mod_perl 1.0 User Guide

Deploying mod_perl technology to give rocket speed to your CGI/Perl scripts.

Last modified Sun Feb 16 01:32:59 2014 GMT

Part I: Getting Started

- 1. Getting Your Feet Wet
  This chapter gives you step-by-step instructions to get a basic statically-compiled mod_perl-enabled Apache server up and running. Having a running server allows you to experiment with mod_perl as you learn more about it.

- 2. Introduction and Incentives
  An introduction to what mod_perl is all about, its different features, and some explanations of the C API, `Apache::Registry`, `Apache::PerlRun`, and the Apache/Perl API.

- 3. mod_perl Installation
  An in-depth explanation of the mod_perl installation process, from the basic installation (in 10 steps), to a more complex one (with all the possible options you might want to use, including DSO build). It includes troubleshooting tips too.

- 4. mod_perl Configuration
  This section documents the various configuration options available for Apache and mod_perl, as well as the Perl startup files, and more esoteric possibilites such as configuring Apache with Perl.

Part II: Coding

- 5. CGI to mod_perl Porting. mod_perl Coding guidelines.
  This chapter is relevant both to writing a new CGI script or perl handler from scratch and to migrating an application from plain CGI to mod_perl.

- 6. How to use mod_perl's Method Handlers
  Described here are a few examples and hints on how to use method handlers with mod_perl.

- 7. mod_perl and Relational Databases
  Creating dynamic websites with mod_perl often involves using relational databases. `Apache::DBI`, which provides a database connections persistence which boosts the mod_perl performance, is explained in this chapter.

- 8. mod_perl and dbm files
  Small databases can be implemented pretty efficiently using dbm files, but there are still some precautions that must be taken to use them properly under mod_perl.

- 9. Protecting Your Site
  Securing your site should be your first priority, because of the consequences a break-in might have. We discuss the various authentication and authorization techniques available, a very interesting use of mod_perl.

- 10. Code Snippets
  A collection of mod_perl code snippets which you can either adapt to your own use or integrate directly into your own code.

- 11. Apache::* modules
Overview of some of the most popular modules for mod_perl, both to use directly from your code and as mod_perl handlers.

Part III: Advanced Setup and Performance

- 12. Choosing the Right Strategy
This document discusses various mod_perl setup strategies used to get the best performance and scalability of the services.

- 13. Real World Scenarios
This chapter provides a step-by-step installation guide for the various setups discussed in *Choosing the Right Strategy*.

- 14. Performance Tuning
An exhaustive list of various techniques you might want to use to get the most performance possible out of your mod_perl server: configuration, coding, memory use, and more.

Part IV: Troubleshooting

- 15. Frequent mod_perl problems
Some problems come up very often on the mailing list. If there is some important problem that is being reported frequently on the list which is not included below, even if it is found elsewhere in the Guide, please report to *mod_perl mailing list*.

- 16. Warnings and Errors Troubleshooting Index
If you encounter an error or warning you don't understand, check this chapter for solutions to common warnings and errors that the mod_perl community has encountered in the last few years.

- 17. Debugging mod_perl
Tired of Internal Server Errors? Find out how to debug your mod_perl applications, thanks to a number of features of Perl and mod_perl.

- 18. Getting Help
If your question isn't answered by reading this guide, check this section for information on how to get help on mod_perl, Apache, or other topics discussed here.

1 Getting Your Feet Wet

1.1 Description

This chapter gives you step-by-step instructions to get a basic statically-compiled mod_perl-enabled Apache server up and running. Having a running server allows you to experiment with mod_perl as you learn more about it.

(Of course, you'll be experimenting on a private machine, not on a production server, right?) The remainder of the guide, along with the documentation supplied with mod_perl, gives the detailed information required for fine-tuning your mod_perl-enabled server to deliver the best possible performance.

Although there are binary distributions of mod_perl-enabled Apache servers available for various platforms, we recommend that you always build mod_perl from source. It's simple to do (provided you have all the proper tools on your machine), and building from source circumvents possible problems with the binary distributions (such as the reported bugs with the RPM packages built for RedHat Linux).

The mod_perl installation that follows has been tested on many mainstream Unix and Linux platforms. Unless you're using a very non-standard system, you should have no problems when building the basic mod_perl server.

For Windows users, the simplest solution is to use the binary package. Windows users may skip to the Installing mod_perl for Windows.

1.2 Installing mod_perl in Three Steps

You can install mod_perl in three easy steps: Obtaining the source files required to build mod_perl, building mod_perl, and installing it.

Building mod_perl from source requires a machine with basic development tools. In particular, you will need an ANSI-compliant C compiler (such as *gcc*) and the *make* utility. All standard Unix-like distributions include these tools. If a required tool is not already installed, then use the package manager that is provided with the system (*rpm*, *apt*, *yast*, etc.) to install them.

A recent version of Perl is also required, at least version 5.004. Perl is available as an installable package, although most Unix-like distributions will have installed Perl by default. To check that the tools are available and learn about their version numbers, try:

```
% make -v
% gcc -v
% perl -v
```

If any of these responds with `Command not found`, it will need to be installed.

Once all the tools are in place the installation process can begin. Experienced Unix users will need no explanation of the commands that follow and can simply copy and paste them into a terminal window to get the server installed.

Acquire the source code distrubutions of Apache and mod_perl from the Internet, using your favorite web browser or one of the command line clients like *wget*, *lwp-download*, etc. These two distributions are available from *http://www.apache.org/dist/httpd/* and *http://apache.org/dist/perl/*.

Remember that mod_perl 1.0 works only with Apache 1.3, and mod_perl 2.0 requires Apache 2.0. In this chapter we talk about mod_perl 1.0/Apache 1.3, hence the packages that you want are named *apache_1.3.xx.tar.gz* and *mod_perl-1.xx.tar.gz*, where *xx* should be replaced with the real version numbers of mod_perl and Apache.

Move the downloaded packages into a directory of your choice, *e.g.*, */home/stas/src/*, proceed with the described steps and you will have mod_perl installed:

```
% cd /home/stas/src
% tar -zvxf apache_1.3.xx.tar.gz
% tar -zvxf mod_perl-1.xx.tar.gz
% cd mod_perl-1.xx
% perl Makefile.PL APACHE_SRC=../apache_1.3.xx/src \
    APACHE_PREFIX=/home/httpd DO_HTTPD=1 USE_APACI=1 EVERYTHING=1
% make && make test
% su
# make install
```

That’s all!

All that remains is to add a few configuration lines to the Apache configuration file, (*/usr/local/apache/conf/httpd.conf*), start the server, and enjoy mod_perl.

The following detailed explanation of each step should help you solve any problems that may have arisen when executing the commands above.

1.3 Installing mod_perl for Unix Platforms

Here is a more detailed explanation of the installation process, with each step explained in detail and with some troubleshooting advice. If the build worked and you are in a hurry to boot your new *httpd*, you may skip to the next section, talking about configuration of the server.

Before installing Apache and mod_perl, you usually have to become *root* so that the files can be installed in a protected area. A user who does not have *root* access, however, can still install all files under their home directory. The proper approach is to build Apache in an unprivileged location and then only use *root* access to install it. We will talk about the nuances of this approach in the dedicated installation process chapter.

1.3.1 Obtaining and Unpacking the Source Code

The first step is to obtain the source code distributions of Apache, available from http://www.apache.org/dist/httpd/ and mod_perl, available from http://apache.org/dist/perl/. Of course you can use your favorite mirror sites to get these distributions.

Even if you have the Apache server running on your machine, usually you need to download its source distribution, because you need to re-build it from scratch with mod_perl.

The source distributions of Apache and mod_perl should be downloaded into directory of your choice. For the sake of consistency, we assume throughout the guide that all builds are being done in the */home/stas/src* directory. However, using a different location is perfectly possible and merely requires changing this part of the path in the examples to the actual path being used.

The next step is to move to the directory with the source files:

```
% cd /home/stas/src
```

Uncompress and untar both sources. GNU *tar* allows this using a single command per file:

```
% tar -zvxf apache_1.3.xx.tar.gz
% tar -zvxf mod_perl-1.xx.tar.gz
```

For non-GNU *tar*'s, you may need to do this with two steps (which you can combine via a pipe):

```
% gzip -dc apache_1.3.xx.tar.gz | tar -xvf -
% gzip -dc mod_perl-1.xx.tar.gz | tar -xvf -
```

Linux distributions supply `tar` and `gzip` and install them by default; for other systems these utilities should be obtained from *http://www.gnu.org/* or other sources, and installed--the GNU versions are available for every platform that Apache supports.

1.3.2 Building mod_perl

Move into the */home/stas/src/mod_perl-1.xx/* source distribution directory:

```
% cd mod_perl-1.xx
```

The next step is to create the *Makefile*. This step is no different in principle from the creation of the *Makefile* for any other Perl module.

```
% perl Makefile.PL APACHE_SRC=../apache_1.3.xx/src \
  DO_HTTPD=1 USE_APACI=1 EVERYTHING=1
```

(Replace *x.x.x* with the Apache distribution version number.)

mod_perl accepts a variety of parameters. The options specified above will enable almost everything that mod_perl offers. There are many other options for fine-tuning mod_perl to suit particular circumstances; these are explained in detail in the dedicated installation process chapter.

Running *Makefile.PL* will cause Perl to check for prerequisites and identify any required software packages which are missing. If it reports missing Perl packages, these will have to be installed before proceeding. They are all available from CPAN (*http://cpan.org/*) and can be easily downloaded and installed.

An advantage to installing mod_perl with the help of the `CPAN.pm` module is that all the missing modules will be installed with the `Bundle::Apache` bundle:

```
% perl -MCPAN -e 'install("Bundle::Apache")'
```

We will talk about using `CPAN.pm` in-depth in the installation process chapter.

Running *Makefile.PL* also transparently executes the *./configure* script from Apache's source distribution directory, which prepares the Apache build configuration files. If parameters must be passed to Apache's `./configure` script, they can be passed as options to *Makefile.PL*.

The *httpd* executable can now be built by using the `make` utility. (Note that the current working directory is still */home/stas/src/mod_perl-1.xx/*):

```
% make
```

This command prepares the mod_perl extension files, installs them in the Apache source tree and builds the *httpd* executable (the web server itself) by compiling all the required files. Upon completion of the *make* process, the working directory is restored to */home/stas/src/mod_perl-1.xx/*.

Running `make test` will execute various mod_perl tests on the freshly built *httpd* executable.

```
% make test
```

This command starts the server on a non-standard port (8529) and tests whether all parts of the built server function correctly. The process will report anything that does not work properly.

1.3.3 Installing mod_perl

Running `make install` completes the installation process of mod_perl by installing all the Perl files required for mod_perl to run--and, of course, the server documentation (man pages). Typically, you need to be *root* to have permission to do this, but another user account can be used if the appropriate options were set on the `perl Makefile.PL` command line. To become *root*, use the *su* command.

```
% su
# make install
```

If you have the proper permission, you might also chain all three *make* commands into a single command line:

```
# make && make test && make install
```

`&&` in shell program is similar to Perl's `&&`. Each section of the statement will be executed left to right, untill all sections will be executed and return true (success) or one of them will return false (failure).

The single-line version simplifies the installation, since there is no need to wait for each command to complete before starting the next one. Of course, if you need to become *root* in order to run *make install*, you'll either need to run it as a separate command or become *root* before running the single-line version.

If the all-in-one approach is chosen and any of the `make` commands fail, execution will stop at that point. For example if `make` alone fails then `make test` and `make install` will not be attempted; similarly if `make test` fails then `make install` will not be attempted.

Finally, change to the Apache source distribution directory and run make install to create the Apache directory tree and install Apache's header files (**.h*), default configuration files (**.conf*), the *httpd* executable, and a few other programs.

```
# cd ../apache_1.3.xx
# make install
```

Note that, as with a plain Apache installation, any configuration files left from a previous installation will not be overwritten by this process. So there is no need to backup the previously working configuration files before the installation, although backing up is never unwise.

At the end of the *make install* process, it will list the path to the `apachectl` utility that you can use to start and stop the server, and the path to the installed configuration files. It is important to write down these paths since they will frequently be needed when maintaining and configuring Apache. On our machines these two important paths are:

```
/usr/local/apache/bin/apachectl
/usr/local/apache/conf/httpd.conf
```

The mod_perl Apache server is now built and installed. All that needs to be done before it can be run is to edit the configuration file *httpd.conf* and write a little test script.

1.3.4 Configuring and Starting the mod_perl Server

The first thing to do is ensure that Apache was built correctly and that it can serve plain HTML files. This helps to minimize the number of possible problem areas: once you have confirmed that Apache can serve plain HTML files, you know that any problems with mod_perl are with mod_perl itself.

Apache should be configured just the same as when it did not have mod_perl. Set the `Port`, `User`, `Group`, `ErrorLog` and other directives in the *httpd.conf* file. Use the defaults as suggested, customizing only when necessary. Values that will probably need to be customized are `ServerName`, `Port`, `User`, `Group`, `ServerAdmin`, `DocumentRoot` and a few others. There are helpful hints preceding each directive in the configuration files themselves, with further information in Apache's documentation. Follow the advice in the files and documentation if in doubt.

When the configuration file has been edited, it is time to start the server. One of the ways to start and stop the server is to use the `apachectl` utility. This can be used to start the server with:

```
# /usr/local/apache/bin/apachectl start
```

And stop it with:

```
# /usr/local/apache/bin/apachectl stop
```

Note that if the server is going to listen on port `80` or another privileged port (Any port with a number less than 1024 can be accessed only by the programs running as *root*.), the user executing `apachectl` must be *root*.

After the server has started, check in the *error_log* file (*/usr/local/apache/logs/error_log* by default) to see if the server has indeed started. Do not rely on the status `apachectl` reports. The *error_log* should contain something like the following:

```
[Thu Jun 22 17:14:07 2000] [notice] Apache/1.3.12 (Unix)
mod_perl/1.24 configured -- resuming normal operations
```

Now point the browser to *http://localhost/* or *http://example.name/* as configured with the `ServerName` directive. If the `Port` directive has been set with a value different from `80`, add this port number at the end of the server name. For example, if the port is 8080, test the server with *http://localhost:8080/* or *http://example.com:8080/*. The infamous *"It worked"* page should appear in the browser, which is an `index.html` file that `make install` in the Apache source tree installs automatically. If this page does not appear, something went wrong and the contents of the *error_log* file should be checked. The path of the error log file is specified in the `ErrorLog` directive section in *httpd.conf*.

If everything works as expected, shut the server down, open *httpd.conf* with a plain text editor, and scroll to the end of the file. The mod_perl configuration directives are added to the end of *httpd.conf* by convention. It is possible to place mod_perl's configuration directives anywhere in *httpd.conf*, but adding them at the end seems to work best in practice.

Assuming that all the scripts that should be executed by the mod_perl enabled server are located in the */home/stas/modperl* directory, add the following configuration directives:

```
Alias /perl/ /home/stas/modperl/

PerlModule Apache::Registry
<Location /perl/>
    SetHandler perl-script
    PerlHandler Apache::Registry
    Options +ExecCGI
    PerlSendHeader On
    allow from all
</Location>
```

Save the modified file.

This configuration causes every URI starting with */perl* to be handled by the Apache mod_perl module with the handler from the Perl module `Apache::Registry`.

1.4 Installing mod_perl for Windows

Apache runs on many flavors of Unix and Unix-like operating systems. Version 1.3 introduced a port to the Windows family of operating systems, often named `Win32` after the name of the common API. Because of many deep differences between Unix and Windows, the Win32 port of Apache is still branded as beta quality, because it hasn't yet reached the stability and performance of the native Unix counterpart.

Another hindrance to using mod_perl 1.0 on Windows is that current versions of Perl are not thread-safe on Win32. As a consequence, mod_perl calls to the embedded Perl interpreter must be serialized, i.e. executed one at a time. See the discussion on multithreading on Win32 mod_perl 1.xx for details. This situation changes with Apache/mod_perl 2.0, which is based on a multi-process/multi-thread approach using a native Win32 threads implementation - see the mod_perl 2 overview for more details, and the discussion of modperl 2.0 in Win32 on getting modperl-2 for Win32 in particular.

Building mod_perl from source on Windows is a bit of a challenge. Development tools such as a C compiler are not bundled with the operating system, and most users expect a point-and-click installation as with most Windows software. Additionally, all software packages need to be built with the same compiler and compile options. This means building Perl, Apache, and mod_perl from source, quite a daunting task. For details on building mod_perl on Windows, see the documentation for modperl 1.0 in Win32 or modperl 2.0 in Win32.

For those who prefer binary distributions, there are a number of alternatives. For mod_perl 1.0, one can obtain either mod_perl 1.0 PPM packages, suitable for use with ActivePerl's `ppm` utility, or else mod_perl 1.0 all-in-one packages containing binaries of Perl and Apache, including mod_perl. For mod_perl 2.0, similar mod_perl 2.0 PPM packages and mod_perl 2.0 all-in-one packages are available.

1.5 Preparing the Scripts Directory

Now you have to select a directory where all the mod_perl scripts and modules will be placed to. We usually use create a directory *modperl* under our home directory, e.g., */home/stas/modperl*, for this purpose. Your mileage may vary.

First create this directory if it doesn't yet exist.

```
% mkdir /home/stas/modperl
```

What about file permissions? Remember that when scripts are executed from a shell, they are being executed with permissions of the user's account. Usually you want to have a read, write and execute access for yourself, but only read and execute permissions for the server. When the scripts are run by Apache, however, the server needs to be able to read and execute them. Apache runs under an account specified by the `User` directive, typically *nobody*. If you modify the `User` directive to run the server under your username, e.g.,

```
User stas
```

the permissions on all files and directories should usually be `rwx------`, so we can set the directory permissions to:

```
% chmod 0700 /home/stas/modperl
```

Now, no-one, but you and the server can access the files in this directory. You should set the same permissions for all the files you place under this directory. (You don't need to set the *x* bit for files that aren't going to be executed, for those mode `0600` would be sufficient.)

If the server is running under account *nobody*, you have to set the permissions to `rwxr-xr-x` or `0755` for your files and directories, which is insecure since other users on the same machine can read your files.

```
# chmod 0755  /home/stas/modperl
```

If you aren't running the server with your username, you have to set these permissions for all the files created under this directory, so Apache can read and execute these.

If you need to have an Apache write files you have to set the file permissions to `rwxrwxrwx` or `0777` which is very undesirable, since any user on the same machine can read and write your files. If this is the case, you should run the server under your username, and then only you and the server have a write access to your files. (Assuming of course that other users have no access to your server, since if they do, they can access your files through this server.)

In the following examples we assume that you run the server under your username, and hence we set the scripts permissions to `0700`.

1.6 A Sample Apache::Registry Script

One of mod_perl's benefits is that it can run existing CGI scripts written in Perl which were previously used under mod_cgi (the standard Apache CGI handler). Indeed mod_perl can be used for running CGI scripts without taking advantage of any of mod_perl's special features, while getting the benefit of the potentially huge performance boost. Here is an example of a very simple CGI-style mod_perl script:

```
mod_perl_rules1.pl
------------------
print "Content-type: text/plain\n\n";
print "mod_perl rules!\n";
```

Save this script in the */home/stas/modperl/mod_perl_rules1.pl* file. Notice that the `#!` line (colloquially known as the *shebang* line) is not needed with mod_perl, although having one causes no problems, as can be seen here:

```
mod_perl_rules1.pl
------------------
#!/usr/bin/perl
print "Content-type: text/plain\n\n";
print "mod_perl rules!\n";
```

Now make the script executable and readable by the server, as explained in the previous section.

```
# chmod 0700   /home/stas/modperl/mod_perl_rules1.pl
```

The *mod_perl_rules1.pl* script can be tested from the command line, since it is essentially a regular Perl script.

```
% perl /home/stas/modperl/mod_perl_rules1.pl
```

This should produce the following output:

```
Content-type: text/plain

mod_perl rules!
```

Make sure the server is running and issue these requests using a browser:

```
http://localhost/perl/mod_perl_rules1.pl
```

If you see it--**congratulations**! You have a working mod_perl server.

If something went wrong, go through the installation process again, making sure that none of the steps are missed and that each is completed successfully. If this does not solve the problem, the installation chapter will attempt to salvage the situation.

If the port being used is not 80, for example 8080, then the port number should be included in the URL:

```
http://localhost:8080/perl/mod_perl_rules1.pl
```

The `localhost` approach will work only if the browser is running on the same machine as the server. If not, use the real server name for this test. For example:

```
http://example.com/perl/mod_perl_rules1.pl
```

If there is any problem please refer to the *error_log* file for the error messages.

Jumping a little bit ahead, we would like to show the same CGI script written using mod_perl API:

```
mod_perl_rules2.pl
------------------
my $r = shift;
$r->send_http_header('text/plain');
$r->print("mod_perl rules!\n");
```

mod_perl API needs a request object `$r` to communicate with Apache. The script gets this object first thing, after that it uses the object to send the HTTP header and print the irrefutable fact about mod_perl's coolness.

This script generates the same output as the previous one.

As you can see it's not much harder to write your code in mod_perl API, all you need to know is the API, but the concepts are the same. As we will show in the later chapters, usually you will want to use mod_perl API for a better performance or when you need a functionality that plain Perl API doesn't provide.

1.6.1 Porting Existing CGI Scripts to run under mod_perl

Now it is time to move any existing CGI scripts from the */somewhere/cgi-bin* directory to */home/stas/modperl*. Once moved they should run much faster when requested from the newly configured base URL (*/perl/*). For example, a CGI script called *test.pl* that was previously accessed as */cgi-bin/test.pl* can now be accessed under mod_perl as */perl/test.pl*.

Some of the scripts might not work immediately and may require some minor tweaking or even a partial rewrite to work properly with mod_perl. We will talk in-depth about these things in the Coding chapter. Most scripts that have been written with care and especially developed with enabled warnings and the `strict` pragma will probably work without any modifications at all.

A quick solution that avoids most rewriting or editing of existing scripts that do not run properly under `Apache::Registry` is to run them under `Apache::PerlRun`. This can be achieved by simply replacing `Apache::Registry` with `Apache::PerlRun` in *httpd.conf*. Put the following configuration directives instead in *httpd.conf* and restart the server:

```
Alias /perl/ /home/stas/modperl/
PerlModule Apache::PerlRun
<Location /perl/>
    SetHandler perl-script
    PerlHandler Apache::PerlRun
    Options ExecCGI
    PerlSendHeader On
    allow from all
</Location>
```

Almost every script should now run without problems; the few exceptions will almost certainly be due to the few minor limitations that mod_perl or its handlers have, but these are all solvable and covered in Coding chapter.

`Apache::PerlRun` is usually useful in the transition period, while the scripts are being cleaned up to run properly under `Apache::Registry`. Though it gives you a significant speedup over mod_cgi, it's still not as fast as `Apache::Registry` and mod_perl handlers.

1.7 A Simple Apache Perl Content Handler

As we mentioned in the beginning of this chapter, mod_perl lets you run both scripts and handlers. The previous example showed a script, since that is probably the most familiar approach to web programming, but the more advanced use of mod_perl involves writing handlers. Have no fear: writing handlers is almost as easy as writing scripts and offers a level of access to Apache's internals that is simply not possible with conventional CGI scripts.

To create a mod_perl handler module, all that is necessary is to wrap the code that would have been the body of a script into a `handler` subroutine, add a statement to return the status to the server when the subroutine has successfully completed, and add a package declaration at the top of the code.

Just as with scripts, the familiar CGI API may be used:

```
ModPerl/Rules1.pm
-----------------
package ModPerl::Rules1;
use Apache::Constants qw(:common);

sub handler {
    print "Content-type: text/plain\n\n";
    print "mod_perl rules!\n";
    return OK; # We must return a status to mod_perl
}
1; # This is a perl module so we must return true to perl
```

Alternatively, the mod_perl API can be used. This API provides almost complete access to the Apache core. In the simple example shown, using either approach is fine, but when lower level access to Apache is required the mod_perl API must be used.

```
ModPerl/Rules2.pm
----------------
package ModPerl::Rules2;
use Apache::Constants qw(:common);

sub handler {
    my $r = shift;
    $r->send_http_header('text/plain');
    $r->print("mod_perl rules!\n");
    return OK; # We must return a status to mod_perl
}
1; # This is a perl module so we must return true to perl
```

Create a directory called *ModPerl* under one of the directories in `@INC` (e.g. under */usr/lib/perl5/site_perl/5.005/*), and put *Rules1.pm* and *Rules2.pm* into it (Note that you will need a *root* access in order to do that.). The files should include the code from the above examples. To find out what the `@INC` directories are, execute:

```
% perl -le 'print join "\n", @INC'
```

On our machine it reports:

```
/usr/lib/perl5/5.00503/i386-linux
/usr/lib/perl5/5.00503
/usr/lib/perl5/site_perl/5.005/i386-linux
/usr/lib/perl5/site_perl/5.005
.
```

So on our machine, we might place the files in the directory */usr/lib/perl5/site_perl/5.005/ModPerl*. By default when you work as *root* the files are created with permissions allowing everybody to read them, so here we don't have to adjust the file permissions, since the server only needs to be able to read those.

Now add the following snippet to */usr/local/apache/conf/httpd.conf* to configure mod_perl to execute the `ModPerl::Rules::handler` subroutine whenever a request to *mod_perl_rules1* is made:

```
PerlModule ModPerl::Rules1
<Location /mod_perl_rules1>
   SetHandler perl-script
   PerlHandler ModPerl::Rules1
   PerlSendHeader On
</Location>
```

Now issue a request to:

```
http://localhost/mod_perl_rules1
```

and just as with the *mod_perl_rules.pl* scripts,

```
mod_perl rules!
```

should be rendered as the response. (Don't forget to include the port number if not using port 80; from now on we will assume this is done, e.g. http://localhost:8080/mod_perl_rules1.)

To test the second module `ModPerl::Rules2` add a similar configuration, while replacing all 1's with 2's:

```
PerlModule ModPerl::Rules2
<Location /mod_perl_rules2>
   SetHandler perl-script
   PerlHandler ModPerl::Rules2
   PerlSendHeader On
</Location>
```

As we will see later in Configuration chapter you can remove the `PerlSendHeader` directive for this particular module.

And to test use the URI:

```
http://localhost/mod_perl_rules2
```

You should see the same response from the server as we have seen when issuing a request for the former mod_perl handler.

1.8 Is This All we Need to Know About mod_perl?

Obviously the next question is: *"Is this all I need to know about mod_perl?"*.

The answer is: "Yes and No".

The *Yes* part:

- Just like with Perl, really cool stuff can be written even with very little mod_perl knowledge. With the basic unoptimized setup presented in this chapter, visitor counters and guest books and any other CGI scripts will run much faster and amaze friends and colleagues, usually without changing a single line of code.

The *No* part:

- A 50 times improvement in guest book response times is great--but when deploying a very heavy service with thousands of concurrent users, a delay of even a few milliseconds might lose a customer, and probably many of them.

Of course when testing a single script with the developer as the only user, squeezing yet another millisecond from the response time seems unimportant. But it becomes a real issue when these milliseconds add up at the production site, with hundreds or thousands of users concurrently generating requests to various scripts on the site. Users are not merciful nowadays. If there is another site that provides the same kind but a significantly faster service, chances are that users will switch to the competing site.

Testing scripts on an unloaded machine can be very misleading, Everything might seem so perfect. But when they are moved into a production machine, things do not behave as well as they did on the development box. For example, the production machine may run out of memory on very busy services. In the performance tuning chapter it will be explained how to optimize code to use less memory and how to make as much memory as possible shared.

Debugging is something that some developers prefer not to think about, since the process can be very tedious at times. Learning how to make the debugging process simpler and efficient is essential for web programmers. This task can be difficult enough when debugging CGI scripts, but it can be even more complicated with mod_perl. The Debugging Chapter explains how to approach debugging in the mod_perl environment.

mod_perl has many features unavailable under mod_cgi when working with databases. Amongst others the most important are persistent database connections. Persistent database connections require a slightly different approach and this is explained in the Databases chapter.

Most web services, especially those which are aimed at an international audience, must run non-stop 24x7. But at the same time new scripts may need to be added, old ones removed, and the server software will need upgrades and security fixes. And if the server goes down, fast recovery is essential. These issues are considered in the Controlling your server chapter.

Finally, the most important aspect of mod_perl is the Apache Perl API, which allows intervention at any or every stage of request processing. This provides incredible flexibility, allowing the creation of scripts and processes which would simply be impossible with mod_cgi.

There are many more things to learn about mod_perl and web programming in general. The rest of this guide will attempt to provide as much information as possible about these and other related matters.

1.9 References

- CPAN is the Comprehensive Perl Archive Network. Comprehensive: its aim is to contain all the Perl material you will need. Archive: close to a gigabyte in size at the time of this writing. Network: CPAN is mirrored at more than one hundred sites around the world.

The CPAN home page: http://cpan.org/

- The libwww-perl distribution is a collection of Perl modules and programs which provide a simple and consistent programming interface (API) to the World Wide Web. The main focus of the library is to provide classes and functions that facilitate writing WWW clients, thus libwww-perl is said to be a WWW client library. The library also contains modules that are of more general use, as well as some useful programs.

  The libwww-perl home page: http://www.linpro.no/lwp/

2 Introduction and Incentives

2.1 Description

An introduction to what mod_perl is all about, its different features, and some explanations of the C API, `Apache::Registry`, `Apache::PerlRun`, and the Apache/Perl API.

2.2 What is mod_perl?

The Apache/Perl integration project brings together the full power of the Perl programming language and the Apache HTTP server. With mod_perl, it is possible to write Apache modules entirely in Perl, letting you easily do things (such as running sub-requests) that are more difficult or impossible in regular CGI programs. In addition, the persistent Perl interpreter embedded in the server saves the overhead of starting an external interpreter, i.e. the penalty of Perl start-up time. And not the least important feature is code caching, where modules and scripts are loaded and compiled only once, and for the rest of the server's life they are served from the cache. Thus the server spends its time only running already loaded and compiled code, which is very fast.

The primary advantages of mod_perl are power and speed. You have full access to the inner workings of the web server and can intervene at any stage of request processing. This allows for customized processing of (to name just a few of the phases) URI->filename translation, authentication, response generation, and logging. There is very little run-time overhead. In particular, it is not necessary to start a separate process, as is often done with web-server extensions. The most wide-spread such extension, the Common Gateway Interface (CGI), can be replaced entirely with Perl code that handles the response generation phase of request processing. mod_perl includes two general purpose modules for this purpose: `Apache::Registry`, which can transparently run existing perl CGI scripts and `Apache::PerlRun`, which does a similar job but allows you to run "dirtier" (to some extent) scripts.

You can configure your httpd server and handlers in Perl (using `PerlSetVar`, and <Perl> sections). You can even define your own configuration directives.

For examples on how you use mod_perl, see our What is mod_perl? section.

Many people ask "How much of a performance improvement does mod_perl give?" Well, it all depends on what you are doing with mod_perl and possibly who you ask. Developers report speed boosts from 200% to 2000%. The best way to measure is to try it and see for yourself! (See Technologie Extraordinaire for the facts.)

2.2.1 mod_cgi

When you run your CGI scripts by using a configuration like this:

```
ScriptAlias /cgi-bin/ /home/httpd/cgi-bin/
```

you are running them under a mod_cgi handler, although you never define it explicitly. Apache does all the configuration work behind the scenes, when you use a ScriptAlias.

By the way, don't confuse `ScriptAlias` with the `ExecCGI` configuration option, which we enable so that the script will be executed rather than returned as a plain text file. For example for mod_perl and `Apache::Registry` you would use a configuration such as:

```
<Location /perl>
  SetHandler perl-script
  PerlHandler Apache::Registry
  Options +ExecCGI
  PerlSendHeader On
</Location>
```

2.2.2 C API

The Apache C API has been present for a long time, and has been the usual way to program extensions (such as mod_perl) to Apache. When you write C extension modules, you write C code that is not independent, but will be linked into the Apache *httpd* executable either at build time (if the module is statically linked), or at runtime (if it is compiled as a Dynamic Shared Object, or DSO). Either way, you do as with any C library: you write functions that receive a certain number of arguments and make use of external API functions, provided by Apache or by other libraries.

The difference is that with Apache extension modules, these functions are *registered* inside a module record: you tell Apache already at compile-time for which phases you wish to run any functions. Of course, you probably won't be handling all the phases. Here is an example of a module handling only the content generation phase:

```
/* Dispatch list of content handlers */
static const handler_rec hello_handlers[] = {
    { "hello", hello_handler },
    { NULL, NULL }
};

/* Dispatch list for API hooks */
module MODULE_VAR_EXPORT hello_module = {
    STANDARD_MODULE_STUFF,
    NULL,                  /* module initializer                  */
    NULL,                  /* create per-dir    config structures */
    NULL,                  /* merge  per-dir    config structures */
    NULL,                  /* create per-server config structures */
    NULL,                  /* merge  per-server config structures */
    NULL,                  /* table of config file commands       */
    hello_handlers,        /* [#8] MIME-typed-dispatched handlers */
    NULL,                  /* [#1] URI to filename translation    */
    NULL,                  /* [#4] validate user id from request  */
    NULL,                  /* [#5] check if the user is ok _here_ */
    NULL,                  /* [#3] check access by host address   */
    NULL,                  /* [#6] determine MIME type            */
    NULL,                  /* [#7] pre-run fixups                 */
    NULL,                  /* [#9] log a transaction              */
    NULL,                  /* [#2] header parser                  */
    NULL,                  /* child_init                          */
    NULL,                  /* child_exit                          */
    NULL                   /* [#0] post read-request              */
};
```

Using this configuration (and a correctly built `hello_handler()` function), you'd then be able to use the following configuration to allow your module to handle the requests for the */hello* URI.

```
<Location /hello>
  SetHandler hello
</Location>
```

When Apache sees a request for the */hello* URI, it will then figure out what the "hello" handler corresponds to by looking it up in the handler record, and match that to the `hello_handler` function pointer, which will execute the `hello_handler` function of your module with a `request_rec *r` as an argument. From that point, your handler is free to do whatever it wants, returning content, declining the request, or doing other bizarre things based on user input.

It is not the object of this guide to explain how to program C handlers. However, this example lets you in on some of the secrets of the Apache core, which you will probably understand anyway by using mod_perl. If you want more information on writing C modules, you should read the Apache API documentation at http://httpd.apache.org/docs/misc/API.html and more importantly *Writing Apache Modules with Perl and C*, which will teach you about *both* mod_perl and C modules!

2.2.3 Perl API

After a while, C modules were found hard to write and difficult to maintain, mostly because code had to be recompiled or just because of the low-level nature of the C language, and because these modules were so intricately linked with Apache that a small bug could put at risk your whole server environment. In comes mod_perl. Programmed in C and using all the techniques described above and more, it allows Perl modules, written in good Perl style, to access the (almost) complete API provided to the conventional C extensions.

However, the structure used for Perl Apache modules is a little different. If you've programmed normal Perl modules (like those found on CPAN) before, you'll be happy to know that programming for mod_perl using the Apache API doesn't involve anything else than writing a Perl module that defines a `handler` subroutine (that is the convention--we'll see that that doesn't necessarily have to be the name). This subroutine accepts an argument, `$r`, which is the Perl API equivalent of the C API `request_rec *r`.

`$r` is your entry point to the whole Perl Apache API. Through it you access methods in good object-oriented fashion, which makes it slightly easier than with C, and looks a lot more familiar to Perl programmers.

Furthermore, Perl Apache modules do not define handler records like C modules. You only need to create your handler subroutine(s), and then control which requests they should handle solely with mod_perl configuration directives inside your Apache configuration.

Let's look at a sample handler that returns a greeting and the current local time.

```
file:My/Greeting.pm
-------------------
package My::Greeting;
use strict;
```

```
use Apache::Constants qw(OK);

sub handler {
    my $r = shift;
    my $now = scalar localtime;
    my $server_name = $r->server->server_hostname;

    $r->send_http_header('text/plain');

    print <<EOT;
Thanks for visiting $server_name.
The local time is $now.
EOT

    return OK;
}
1; # modules must return true
```

As you can see, we're mixing Perl standard functions (like `localtime()`) with Apache functions (`$r->send_http_header()`). To return the above greeting when accessing the */hello* URI, you would configure Apache like this:

```
<Location /hello>
    SetHandler perl-script
    PerlHandler My::Greeting
</Location>
```

When it sees this configuration, mod_perl loads the `My::Greeting` module, finds the `handler()` subroutine, and calls it to allow it to return the appropriate content. There are equivalent `Perl*Handler` directives for the different phases we saw were available to C handlers.

The Perl API gives you an incredible number of possibilities, which you can then use to be more productive or creative. mod_perl is an enabling technology; it won't make you smarter or more creative, but it will do its best to make you lose less time because of *"accidental difficulties"* of programming, and let you concentrate more on the important parts.

2.2.4 Apache::Registry

From the viewpoint of the Perl API, Apache::Registry is simply another handler that's not conceptually different from any other handler. `Apache::Registry` reads in the script file, compiles, executes it and stores into the cache. Since the perl interpreter keeps running from child process' creation to its death, any code compiled by the interpreter is kept in memory until the child dies.

To prevent script name collisions, `Apache::Registry` creates a unique key for each cached script by prepending `Apache::ROOT::` to the mangled path of the script's URI. This key is actually the package name that the script resides in. So if you have requested a script `/perl/project/test.pl`, the scripts would be wrapped in code which starts with a package declaration of:

```
package Apache::ROOT::perl::project::test_e2pl;
```

`Apache::Registry` also stores the script's last modification time. Everytime the script changes, the cached code is discarded and recompiled using the modified source. However, it doesn't check the modification times of any of the perl libraries the script might use.

`Apache::Registry` overrides `CORE::exit()` with `Apache::exit()`, so CGI scripts that use `exit()` will run correctly. We will talk about all these details in depth later.

From the viewpoint of the programmer, there is almost no difference between running a script as a plain CGI script under mod_cgi and running it under mod_perl. There is however a great speed improvement, but at the expense of much heavier memory usage (there is no free lunch :).

When they run under mod_cgi, your CGI scripts are loaded each time they are called and then they exit. Under mod_perl they are loaded once and cached. This gives a big performance boost. But because the code is cached and doesn't exit, it won't cleanup memory as it would under mod_cgi. This can have unexpected effects.

Your scripts will be recompiled and reloaded by mod_perl when it detects that you have changed them, but remember that any libraries that your scripts might require() or use() will not be recompiled when they are changed. You will have to take action yourself to ensure that they are recompiled.

Of course the guide will answer all these issues in depth.

Let's see what happens to your script when it's being executed under `Apache::Registry`. If we take the simplest code of (URI `/perl/project/test.pl`)

```
print "Content-type: text/html\n\n";
print "It works\n";
```

`Apache::Registry` will convert it into the following:

```
package Apache::ROOT::perl::project::test_e2pl;
use Apache qw(exit);
sub handler {
  print "Content-type: text/html\n\n";
  print "It works\n";
}
```

The first line provides a unique namespace for the code to use, and a unique key by which the code can be referenced from the cache.

The second line imports `Apache::exit` which over-rides perl's built-in `exit`.

The `sub handler` subroutine is wrapped around your code. By default (i.e. if you do not specify an alternative), when you use mod_perl and your code's URI is called, mod_perl will seek to execute the URI's associated `handler` subroutine.

`Apache::Registry` is usually configured in this way:

```
Alias /perl/ /usr/local/apache/bin/
<Location /perl>
  SetHandler perl-script
  PerlHandler Apache::Registry
</Location>
```

In short, we see that `Apache::Registry` is just another mod_perl handler, which is executed when requests are made for the */perl* directory, and then does some special handling of the Perl scripts in that directory to turn *them* into Apache handlers.

2.2.5 Apache::PerlRun

Apache::PerlRun is very similar to `Apache::Registry`. It uses the same basic concepts, i.e. it runs CGI scripts under mod_perl for additional speed. However, unlike `Apache::Registry`, `Apache::PerlRun` will not cache scripts. The reason for this is that it's designed for use with CGI scripts that may have been "dirty", which might cause problems when run persistently under mod_perl. Apart from that, the configuration is the same. We discuss `Apache::PerlRun` in Apache::PerlRun, a closer look.

2.3 What you will learn

This document was written in an effort to help you start using Apache's mod_perl extension as quickly and easily as possible. It includes information about the installation and configuration of both Perl and the Apache web server and delves deeply into the issues of writing and porting existing Perl scripts to run under mod_perl. Note that it does not attempt to enter the big world of using the Perl API or C API. You will find pointers to coverage of these topics in the Offsite resources section of this site. This guide tries to cover the most of the `Apache::Registry` and `Apache::PerlRun` modules. Along with mod_perl related topics, there are many more issues related to administering Apache servers, debugging scripts, using databases, mod_perl related Perl, code snippets and more.

It is assumed that you know at least the basics of building and installing Perl and Apache. (If you do not, just read the INSTALL documents which are part of the distribution of each package.) However, in this guide you will find specific Perl and Apache installation and configuration notes, which will help you successfully complete the mod_perl installation and get the server running in a short time.

If after reading this guide and the other documentation you feel that your questions remain unanswered, you could try asking the apache/mod_perl mailing list to help you. But first try to browse the mailing list archive. Often you will find the answer to your question by searching the mailing list archive, since most questions have been asked and answered already! If you ignore this advice, do not be surprised if your question goes unanswered - it bores people when they're asked to answer the same question repeatedly - especially if the answer can be found in the archive or in the documentation. This does not mean that you should avoid asking questions, just do not abuse the available help and **RTFM** before you call for **HELP**. When asking your question, be sure to have read the email-etiquette and How to report problems

If you find errors in these documents, please contact the maintainer, after having read about how to submit documentation patches.

2.4 Maintainers

Maintainer is the person(s) you should contact with updates, corrections and patches.

- Stas Bekman [http://stason.org/]

2.5 Authors

- Stas Bekman [http://stason.org/]

Only the major authors are listed above. For contributors see the Changes file.

3 mod_perl Installation

3.1 Description

An in-depth explanation of the mod_perl installation process, from the basic installation (in 10 steps), to a more complex one (with all the possible options you might want to use, including DSO build). It includes troubleshooting tips too.

First of all:

```
Apache 2.0 doesn't work with mod_perl 1.0.
Apache 1.0 doesn't work with mod_perl 2.0.
```

3.2 A Summary of a Basic mod_perl Installation

The following 10 commands summarize the execution steps required to build and install a basic mod_perl enabled Apache server on almost any standard flavor of Unix OS.

```
% cd /usr/src
% lwp-download http://www.apache.org/dist/httpd/apache_1.3.xx.tar.gz
% lwp-download http://apache.org/dist/perl/mod_perl-1.xx.tar.gz
% tar xzvf apache_1.3.xx.tar.gz
% tar xzvf mod_perl-1.xx.tar.gz
% cd mod_perl-1.xx
% perl Makefile.PL APACHE_SRC=../apache_1.3.xx/src \
  DO_HTTPD=1 USE_APACI=1 EVERYTHING=1
% make && make test && make install
% cd ../apache_1.3.xx
% make install
```

Of course you should replace *1.xx* and *1.3.x* with the real version numbers of mod_perl and Apache.

All that's left is to add a few configuration lines to `httpd.conf`, the Apache configuration file, start the server and enjoy mod_perl.

If you have stumbled upon a problem at any of the above steps, don't despair, the next sections will explain in detail each and every step.

Of course there is a way of installing mod_perl in only a couple of minutes if you are using a Linux distribution that uses RPM packages:

```
% rpm -i apache-1.3.xx.rpm
% rpm -i mod_perl-1.xx.rpm
```

or apt system:

```
% apt-get install libapache-mod-perl
```

These should set up both Apache and mod_perl correctly for your system. Using a packaged distribution can make installing and reinstalling a lot quicker and easier. (Note that the filenames will vary, and *xx* will differ.)

Since mod_perl can be configured in many different ways (features can be enabled or disabled, directories can be modified, etc.) it's preferable to use a manual installation, as a prepackaged version might not suit your needs. Manual installation will allow you to make the fine tuning for the best performance as well.

In this chapter we will talk extensively about the prepackaged versions, and ways to prepare your own packages for reuse on many machines. Win32 users should consult the Win32 documentation for details on that platform.

3.3 The Gory Details

We saw that the basic mod_perl installation is quite simple and takes about 10 commands. You can copy and paste them from these pages. The parameter `EVERYTHING=1` selects a lot of options, but sometimes you will need different ones. You may need to pass only specific parameters, to bundle other components with mod_perl etc. You may want to build mod_perl as a loadable object instead of compiling it into Apache, so that it can be upgraded without rebuilding Apache itself.

To accomplish this you will need to understand various techniques for mod_perl configuration and building. You need to know what configuration parameters are available to you and when and how to use them.

As with Perl, with mod_perl simple things are simple. But when you need to accomplish more complicated tasks you may have to invest some time to gain a deeper understanding of the process. In this chapter I will take the following route. I'll start with a detailed explanation of the four stages of the mod_perl installation process, then continue with the different paths each installation might take according to your goal, followed by a few copy-and-paste real world installation scenarios. Towards the end of the chapter I will show you various approaches that make the installations easier, automating most of the steps. Finally I'll cover some of the general issues that can cause new users to stumble while installing mod_perl.

We can clearly separate the installation process into the following stages:

- **Source Configuration,**
- **Building,**
- **Testing and**
- **Installation.**

3.3.1 Source Configuration (perl Makefile.PL ...)

Before building and installing mod_perl you have to configure it. You configure mod_perl just like any other Perl module:

```
% perl Makefile.PL [parameters]
```

In this section we will go through most of the parameters mod_perl can accept and explain each one of them.

First let's see what configuration mechanisms we have available. Basically they all define a special set of parameters to be passed to `perl Makefile.PL`. Depending on the chosen configuration, the final product might be a stand-alone httpd binary or a loadable object.

The source configuration mechanism in Apache 1.3 provides four major features (which of course are available to mod_perl):

- **Per-module configuration scripts (ConfigStart/End)**

  This is a mechanism modules can use to link themselves into the configuration process. It is useful for automatically adjusting the configuration and build parameters from the modules sources. It is triggered by `ConfigStart`/`ConfigEnd` sections inside *modulename.module* files (e.g. *libperl.module*).

- **Apache Autoconf-style Interface (APACI)**

  This is the new top-level `configure` script from Apache 1.3 which provides a GNU Autoconf-style interface. It is useful for configuring the source tree without manually editing any *src/Configuration* files. Any parameterization can be done via command line options to the `configure` script. Internally this is just a nifty wrapper to the old `src/Configure` script.

  Since Apache 1.3 this is the way to install mod_perl as cleanly as possible. Currently this is a pure Unix-based solution because at present the complete Apache 1.3 source configuration mechanism is only available under Unix. It doesn't work on the Win32 platform for example.

- **Dynamic Shared Object (DSO) support**

  Besides Windows NT support this is one of most interesting features in Apache 1.3. Its a way to build Apache modules as so-called *dynamic shared objects* (usually named *modulename.so*) which can be loaded via the `LoadModule` directive in Apache's *httpd.conf* file. The benefit is that the modules are part of the `httpd` executable only on demand, i.e. only when the user wants a module it is loaded into the address space of the `httpd` executable. This is interesting for instance in relation to memory consumption and upgrading.

  The DSO mechanism is provided by Apache's `mod_so` module which needs to be compiled into the `httpd` binary. This is done automatically when DSO is enabled for module `mod_foo` via:

  ```
  ./configure --enable-module=foo
  ```

  or by explicitly adding `mod_so` via:

  ```
  ./configure --enable-module=so
  ```

- **APache eXtenSion (APXS) support tool**

  This is a new support tool from Apache 1.3 which can be used to build an Apache module as a DSO even **outside** the Apache source-tree. One can say `APXS` is for Apache what `MakeMaker` and `XS` are for Perl. It knows the platform dependent build parameters for making DSO files and provides an easy way to run the build commands with them.

  (`MakeMaker` allows an easy automatic configuration, build, testing and installation of the Perl modules, and `XS` allows to call functions implemented in C/C++ from Perl code.)

Taking these four features together provides a way to integrate mod_perl into Apache in a very clean and smooth way. *No patching* of the Apache source tree is needed in the standard situation and in the APXS situation not even the Apache source tree is needed.

To benefit from the above features a new hybrid build environment was created for the Apache side of mod_perl. The Apache-side consists of the mod_perl C source files which have to be compiled into the `httpd` program. They are usually copied to the subdirectory *src/modules/perl/* in the Apache source tree. To integrate this subtree into the Apache build process a lot of adjustments were done by mod_perl's *Makefile.PL* in the past. And additionally the `Makefile.PL` controlled the Apache build process.

This approach is problematic in several ways. It is very restrictive and not very clean because it assumes that mod_perl is the only third-party module which has to be integrated into Apache.

The new approach described below avoids these problems. It prepares only the *src/modules/perl/* subtree inside the Apache source tree *without* adjusting or editing anything else. This way, no conflicts can occur. Instead, mod_perl is activated later (when the Apache source tree is configured, via APACI calls) and then it configures itself.

We will return to each of the above configuration mechanisms when describing different installation passes, once the overview of the four building steps is completed.

3.3.1.1 Configuration parameters

The command `perl Makefile.PL`, which is also known as a *"configuration stage"*, accepts various parameters. In this section we will learn what they are, and when should they be used.

3.3.1.1.1 APACHE_SRC

If you specify neither the `DO_HTTPD` nor the `NO_HTTPD` parameters you will be asked the following question during the configuration stage:

```
Configure mod_perl with ../apache_1.3.xx/src ?
```

`APACHE_SRC` should be used to define Apache's source tree directory. For example:

```
APACHE_SRC=../apache_1.3.xx/src
```

Unless `APACHE_SRC` is specified, *Makefile.PL* makes an intelligent guess by looking at the directories at the same level as the mod_perl sources and suggests a directory with the highest version of Apache found there.

Answering *'y'* confirms either *Makefile.PL*'s guess about the location of the tree, or the directory you have specified with `APACHE_SRC`.

If you use `DO_HTTPD=1` or `NO_HTTPD`, the first Apache source tree found or the one you have defined will be used for the rest of the build process.

3.3.1.1.2 DO_HTTPD, NO_HTTPD, PREP_HTTPD

Unless any of `DO_HTTPD`, `NO_HTTPD` or `PREP_HTTPD` is used, you will be prompted by the following question:

```
Shall I build httpd in ../apache_1.3.xx/src for you?
```

Answering *'y'* will make sure an httpd binary will be built in *../apache_1.3.xx/src* when you run `make`.

To avoid this prompt when the answer is *Yes* specify the following argument:

```
DO_HTTPD=1
```

Note that if you set `DO_HTTPD=1`, but do not use `APACHE_SRC=../apache_1.3.xx/src` then the first apache source tree found will be used to configure and build against. Therefore it's highly advised to always use an explicit `APACHE_SRC` parameter, to avoid confusion.

`PREP_HTTPD=1` just means default 'n' to the latter prompt, meaning: *Do not build (make) httpd in the Apache source tree*. But it will still ask you about Apache's source location even if you have used the `APACHE_SRC` parameter. Providing the `APACHE_SRC` parameter will just eliminate the need for `perl Makefile.PL` to make a guess.

To avoid the two prompts:

```
Configure mod_perl with ../apache_1.3.xx/src ?
Shall I build httpd in ../apache_1.3.xx/src for you?
```

and avoid building httpd, use:

```
NO_HTTPD=1
```

If you choose not to build the binary you will have to do that manually. We will talk about it later. In any case you will need to run `make install` in the mod_perl source tree, so the Perl side of mod_perl will be installed. Note that, `make test` won't work until you have built the server.

3.3.1.1.3 Callback Hooks

A callback hook (abbrev. *callback*) is a reference to a subroutine. In Perl we create callbacks with the $callback = \&subroutine syntax, where in this example, `$callback` contains a reference to the subroutine called *"subroutine"*. Callbacks are used when we want some action (subroutine call) to occur when some event takes place. Since we don't know exactly when the event will take place we give the event handler a callback to the subroutine we want executed. The handler will call our subroutine at the right time.

By default, most of the callback hooks except for `PerlHandler`,`PerlChildInitHandler`, `PerlChildExitHandler`, `PerlConnectionApi`, and `PerlServerApi` are turned off. You may enable them by editing *src/modules/perl/Makefile*, or when running `perl Makefile.PL`.

The possible parameters for the appropriate hooks are:

```
PERL_POST_READ_REQUEST
PERL_TRANS
PERL_INIT
PERL_RESTART (experimental)

PERL_HEADER_PARSER
PERL_AUTHEN
PERL_AUTHZ
PERL_ACCESS
PERL_TYPE
PERL_FIXUP
PERL_LOG
PERL_CLEANUP
PERL_CHILD_INIT
PERL_CHILD_EXIT
PERL_DISPATCH

PERL_STACKED_HANDLERS
PERL_METHOD_HANDLERS
PERL_SECTIONS
PERL_SSI
```

As with any parameters that are either defined or not, use `PERL_hookname=1` to enable them (e.g. `PERL_AUTHEN=1`).

To enable all, but the last 4 callback hooks use:

```
ALL_HOOKS=1
```

3.3.1.1.4 EVERYTHING

To enable everything set:

```
EVERYTHING=1
```

3.3.1.1.5 PERL_TRACE

To enable debug tracing set: `PERL_TRACE=1`

3.3.1.1.6 APACHE_HEADER_INSTALL

By default, the Apache source headers files are installed into the *$Config{sitearch-exp}/auto/Apache/include* directory.

The reason for installing the header files is to make life simpler for module authors/users when building/installing a module that taps into some Apache C functions, e.g. `Embperl`, `Apache::Peek`, etc.

If you don’t wish to install these files use:

```
APACHE_HEADER_INSTALL=0
```

3.3.1.1.7 PERL_STATIC_EXTS

Normally, if an extension is statically linked with Perl it is listed in `Config.pm`'s `$Config{static_exts}`, in which case mod_perl will also statically link this extension with httpd. However, if an extension is statically linked with Perl after it is installed, it is not listed in `Config.pm`. You may either edit `Config.pm` and add these extensions, or configure mod_perl like this:

```
perl Makefile.PL "PERL_STATIC_EXTS=Something::Static Another::One"
```

3.3.1.1.8 APACI_ARGS

When you use the `USE_APACI=1` parameter, you can tell `Makefile.PL` to pass any arguments you want to the Apache `./configure` utility, e.g:

```
% perl Makefile.PL USE_APACI=1 \
APACI_ARGS='--sbindir=/usr/local/httpd_perl/sbin, \
       --sysconfdir=/usr/local/httpd_perl/etc, \
       --localstatedir=/usr/local/httpd_perl/var, \
       --runtimedir=/usr/local/httpd_perl/var/run, \
       --logfiledir=/usr/local/httpd_perl/var/logs, \
       --proxycachedir=/usr/local/httpd_perl/var/proxy'
```

Notice that **all** `APACI_ARGS` (above) must be passed as one long line if you work with `t?csh`!!! However it works correctly as shown above (breaking the long lines with '\') with `(ba)?sh`. If you use `t?csh` it does not work, since `t?csh` passes the `APACI_ARGS` arguments to `./configure` leaving the newlines untouched, but stripping the backslashes. This causes all the arguments except the first to be ignored by the configuration process.

3.3.1.1.9 APACHE_PREFIX

Alternatively to:

```
APACI_ARGS='--prefix=/usr/local/httpd_perl'
```

from the previous section you can use the `APACHE_PREFIX` parameter. When `USE_APACI` is enabled, this attribute will specify the *--prefix* option just like the above setting does.

In addition when the `APACHE_PREFIX` option is used `make install` be executed in the Apache source directory, which makes these two equivalent:

```
% perl Makefile.PL APACHE_SRC=../apache_1.3.xx/src \
  DO_HTTPD=1 USE_APACI=1 EVERYTHING=1 \
  APACI_ARGS='--prefix=/usr/local/httpd_perl'
% make && make test && make install
% cd ../apache_1.3.xx
% make install

% perl Makefile.PL APACHE_SRC=../apache_1.3.xx/src \
  DO_HTTPD=1 USE_APACI=1 EVERYTHING=1 \
  APACHE_PREFIX=/usr/local/httpd_perl
% make && make test && make install
```

Now you can pick your favorite installation method.

3.3.1.2 Environment Variables

There are a few environment variables that influence the build/test process.

3.3.1.2.1 APACHE_USER and APACHE_GROUP

You can use the environment variables `APACHE_USER` and `APACHE_GROUP` to override the default `User` and `Group` settings in the *httpd.conf* used for 'make test' stage. (Introduced in mod_perl v1.23.)

3.3.1.3 Reusing Configuration Parameters

When you have to upgrade the server, it's quite hard to remember what parameters were used in a mod_perl build. So it's better to save them in a file. For example if you create a file at *~/.mod_perl_build_options*, with contents:

```
APACHE_SRC=../apache_1.3.xx/src DO_HTTPD=1 USE_APACI=1 \
EVERYTHING=1
```

You can build the server with the following command:

```
% perl Makefile.PL `cat ~/.mod_perl_build_options`
% make && make test && make install
```

But mod_perl has a standard method to perform this trick. If a file named *makepl_args.mod_perl* is found in the same directory as the mod_perl build location with any of these options, it will be read in by *Makefile.PL*. Parameters supplied at the command line will override the parameters given in this file.

```
% ls -1 /usr/src
apache_1.3.xx/
makepl_args.mod_perl
mod_perl-1.xx/

% cat makepl_args.mod_perl
APACHE_SRC=../apache_1.3.xx/src DO_HTTPD=1 USE_APACI=1 \
EVERYTHING=1

% cd mod_perl-1.xx
% perl Makefile.PL
% make && make test && make install
```

Now the parameters from *makepl_args.mod_perl* file will be used, as if they were directly typed in.

Notice that this file can be located in your home directory or in the *../* directory relative to the mod_perl distribution directory. This file can also start with dot (*.makepl_args.mod_perl*) so you can keep it nicely hidden along with the rest of the dot files in your home directory.

There is a sample *makepl_args.mod_perl* in the *eg/* directory of the mod_perl distribution package, in which you might find a few options to enable experimental features to play with too!

If you are faced with a compiled Apache and no trace of the parameters used to build it, you can usually still find them if the sources were not `make clean`'d. You will find the Apache specific parameters in `apache_1.3.xx/config.status` and the mod_perl parameters in `mod_perl-1.xx/apaci/mod_perl.config`.

3.3.1.4 Discovering Whether Some Option Was Configured

mod_perl version 1.25 has introduced `Apache::Myconfig`, which provides access to the various hooks and features set when mod_perl is built. This circumvents the need to set up a live server just to find out if a certain callback hook is available.

To see whether some feature was built in or not, check the `%Apache::MyConfig::Setup` hash. For example after installing mod_perl with the following options:

```
panic% perl Makefile.PL EVERYTHING=1
```

but on the next day we don't remember what callback hooks were enabled, and we want to know whether `PERL_LOG` callback hook is enabled. One of the ways to find this out is to run the following code:

```
panic% perl -MApache::MyConfig \
-e 'print $Apache::MyConfig::Setup{PERL_LOG}'
1
```

which prints '1'--meaning that `PERL_LOG` callback hook was enabled. (That's because `EVERYTHING=1` enables them all.)

Another approach is to configure `Apache::Status` and run http://localhost/perl-status?hooks to check for enabled hooks.

You also may try to look at the symbols inside the httpd executable with help of `nm(1)` or a similar utility. For example if you want to see whether you enabled `PERL_LOG=1` while building mod_perl, we can search for a symbol with the same name but in lowercase:

```
panic% nm httpd | grep perl_log
08071724 T perl_logger
```

Indeed we can see that in our example `PERL_LOG=1` was enabled. But this will only work if you have an unstripped httpd binary. By default, `make install` strips the binary before installing it. Use the `--without-execstrip ./Configure` option to prevent stripping during *make install* phase.

Yet another approach that will work in most of the cases is to try to use the feature in question. If it wasn't configured Apache will give an error message

3.3.1.5 Using an Alternative Configuration File

By default mod_perl provides its own copy of the *Configuration* file to Apache's `./configure` utility. If you wish to pass it your own version, do this:

```
% perl Makefile.PL CONFIG=Configuration.custom
```

Where *Configuration.custom* is the pathname of the file *relative to the Apache source tree you build against*.

3.3.1.6 perl Makefile.PL Troubleshooting

3.3.1.6.1 "A test compilation with your Makefile configuration failed..."

When you see this during the `perl Makefile.PL` stage:

```
** A test compilation with your Makefile configuration
** failed. This is most likely because your C compiler
** is not ANSI. Apache requires an ANSI C Compiler, such
** as gcc. The above error message from your compiler
** will also provide a clue.
 Aborting!
```

you've got a problem with your compiler. It is possible that it's improperly installed or not installed at all. Sometimes the reason is that your Perl executable was built on a different machine, and the software installed on your machine is not the same. Generally this happens when you install the prebuilt packages, like RPM or deb. The dependencies weren't properly defined in the Perl binary package and you were allowed to install it, although some essential package is not installed.

The most frequent pitfall is a missing gdbm library. See Missing or Misconfigured libgdbm.so for more info.

But why guess, when we can actually see the real error message and understand what the real problem is. To get a real error message, edit the Apache *src/Configure* script. Down around line 2140 you will see a line like this:

```
if ./helpers/TestCompile sanity; then
```

change it to:

```
if ./helpers/TestCompile -v sanity; then
```

and try again. Now you should get a useful error message.

3.3.1.6.2 Missing or Misconfigured libgdbm.so

On some Linux RedHat systems you might encounter a problem during the `perl Makefile.PL` stage, when the installed from the rpm package Perl was built with the `gdbm` library, but the library isn't actually installed. If this is your situation make sure you install it before proceeding with the build process.

You can check how Perl was built by running the `perl -V` command:

```
% perl -V | grep libs
```

On my machine I get:

```
libs=-lnsl -lndbm -lgdbm -ldb -ldl -lm -lc -lposix -lcrypt
```

Sometimes the problem is even more obscure: you do have `libgdbm` installed but it's not properly installed. Do this:

```
% ls /usr/lib/libgdbm.so*
```

If you get at least three lines like I do:

```
lrwxrwxrwx   /usr/lib/libgdbm.so -> libgdbm.so.2.0.0
lrwxrwxrwx   /usr/lib/libgdbm.so.2 -> libgdbm.so.2.0.0
-rw-r--r--   /usr/lib/libgdbm.so.2.0.0
```

you are all set. On some installations the *libgdbm.so* symbolic link is missing, so you get only:

```
lrwxrwxrwx   /usr/lib/libgdbm.so.2 -> libgdbm.so.2.0.0
-rw-r--r--   /usr/lib/libgdbm.so.2.0.0
```

To fix this problem add the missing symbolic link:

```
% cd /usr/lib
% ln -s libgdbm.so.2.0.0 libgdbm.so
```

Now you should be able to build mod_perl without any problems.

Note that you might need to prepare this symbolic link as well:

```
lrwxrwxrwx   /usr/lib/libgdbm.so.2 -> libgdbm.so.2.0.0
```

with:

```
% ln -s libgdbm.so.2.0.0 libgdbm.so.2
```

Of course if when you read this a new version of the `libgdbm` library will be released, you will have to adjust the version numbers. We didn't use the usual *xx* version replacement here, to make it easier to understand how the symbolic links should be set.

3.3.1.6.3 About gdbm, db and ndbm libraries

Both the gdbm and db libraries offer ndbm emulation, which is the interface that Apache actually uses, so when you build mod_perl you end up with whichever library was linked first by the perl compile. If you build apache without mod_perl you end up with whatever appears to be your ndbm library which will vary between systems, and especially Linux distributions. You may have to work a bit to get both Apache and Perl to use the same library and you are likely to have trouble copying the dbm file from one system to another or even using it after an upgrade.

3.3.1.6.4 Undefined reference to 'PL_perl_destruct_level'

When manually building mod_perl using the shared library:

```
cd mod_perl-1.xx
perl Makefile.PL PREP_HTTPD=1
make
make test
make install

cd ../apache_1.3.xx
./configure --with-layout=RedHat --target=perlhttpd
--activate-module=src/modules/perl/libperl.a
```

you might get:

```
gcc -c  -I./os/unix -I./include   -DLINUX=2 -DTARGET=\"perlhttpd\" -DUSE_HSREGEX
-DUSE_EXPAT -I./lib/expat-lite `./apaci` buildmark.c
gcc  -DLINUX=2 -DTARGET=\"perlhttpd\" -DUSE_HSREGEX -DUSE_EXPAT
-I./lib/expat-lite `./apaci`    \
      -o perlhttpd buildmark.o modules.o modules/perl/libperl.a
modules/standard/libstandard.a main/libmain.a ./os/unix/libos.a ap/libap.a
regex/libregex.a lib/expat-lite/libexpat.a  -lm -lcrypt
modules/perl/libperl.a(mod_perl.o): In function `perl_shutdown':
mod_perl.o(.text+0xf8): undefined reference to `PL_perl_destruct_level'
mod_perl.o(.text+0x102): undefined reference to `PL_perl_destruct_level'
mod_perl.o(.text+0x10c): undefined reference to `PL_perl_destruct_level'
mod_perl.o(.text+0x13b): undefined reference to `Perl_av_undef'
[more errors snipped]
```

This happens when you have a statically linked perl build (i.e. without a shared *libperl.so* library). Build a dynamically linked perl (with *libperl.so*) and the problem will disappear. See the "Building a shared Perl library" section from the *INSTALL* file that comes with Perl source.

Also see "Chapter 15.4 - Perl Build Options" from Practical mod_perl.

3.3.1.6.5 Further notes on libperl.(a|so)

Library files such as *libfoo.a* are archives that are used at linking time - these files are completely included in the final application that linked it.

Whereas *libfoo.so* indicates that it's a shared library. At the linking time the application only knows which library it wants. Only at the loading time (runtime) that shared library will be loaded.

One of the benefits of using a shared library, is that it's loaded only once. If there are two application linking to *libperl.so* that run at the same time, only the first application will need to load it. The second application will share that loaded library (that service is provided by the OS kernel). In the case of static *libfoo.a*, it'll be loaded as many times as there are applications that included it, thus consuming more memory. Of course this is not the only benefit of using shared libs.

In mod_perl 1.0, the library file is unfortunately named *libperl.(so|a)*. So you have *libperl.(so|a)* which is perl, and you have *libperl.(so|a)* which is modperl. You are certainly looking at the modperl version of *libperl.a* if you find it in the apache directory. perl's *libperl.(so|a)* lives under the perl tree (e.g. in *5.8.6/i686-linux/CORE/libperl.so*).

Some distributions (notably Debian) have chosen to put *libperl.so* and *libperl.a* into the global library loader path (e.g., */usr/lib*) which will cause linking problems when compiling mod_perl (if compiling against static perl), in which case you should move aside the *libperl.a* while building mod_perl or else will likely encounter further errors. If building against the dynamic perl's *libperl.so*, you may have similar problems but at startup time. It's the best to get rid of perl that installs its libs into */usr/lib* (or similar) and reinstall a new perl, which puts its library under the perl tree. Also see libperl.so and libperl.a.

3.3.2 mod_perl Building (make)

After completing the configuration you build the server, by calling:

```
% make
```

which compiles the source files and creates an httpd binary and/or a separate library for each module, which can either be inserted into the httpd binary when `make` is called from the Apache source directory or loaded later, at run time.

Note: don't put the mod_perl directory inside the Apache directory. This confuses the build process.

3.3.2.1 make Troubleshooting

3.3.2.1.1 Undefined reference to 'Perl_newAV'

This and similar error messages may show up during the `make` process. Generally it happens when you have a broken Perl installation. Make sure it's not installed from a broken RPM or another binary package. If it is, build Perl from source or use another properly built binary package. Run `perl -V` to learn what version of Perl you are using and other important details.

3.3.2.1.2 Unrecognized format specifier for...

This error usually reported due to the problems with some versions of SFIO library. Try to use the latest version to get around this problem. Or if you don't really need SFIO, rebuild Perl without this library.

3.3.3 Built Server Testing (make test)

After building the server, it's a good idea to test it thoroughly, by calling:

```
% make test
```

Fortunately mod_perl comes with a bunch of tests, which attempt to use all the features you asked for at the configuration stage. If any of the tests fails, the `make test` stage will fail.

Running `make test` will start a freshly built httpd on port 8529 running under the uid and gid of the `perl Makefile.PL` process. The httpd will be terminated when the tests are finished.

Each file in the testing suite generally includes more than one test, but when you do the testing, the program will only report how many tests were passed and the total number of tests defined in the test file. However if only some of the tests in the file fail you want to know which ones failed. To gain this information you should run the tests in verbose mode. You can enable this mode by using the `TEST_VERBOSE` parameter:

```
% make test TEST_VERBOSE=1
```

To change the default port (8529) used for the test do this:

```
% perl Makefile.PL PORT=xxxx
```

To start the newly built Apache:

```
% make start_httpd
```

To shutdown Apache:

```
% make kill_httpd
```

NOTE to Ben-SSL users: httpsd does not seem to handle */dev/null* as the location of certain files (for example some of the configuration files mentioned in *httpd.conf* can be ignored by reading them from */dev/null*) so you'll have to change these by hand. The tests are run with the `SSLDisable` directive.

3.3.3.1 Manual Testing

Tests are invoked by running the `./TEST` script located in the *./t* directory. Use the *-v* option for verbose tests. You might run an individual test like this:

```
% t/TEST -v modules/file.t
```

or all tests in a test sub-directory:

```
% t/TEST modules
```

The `TEST` script starts the server before the test is executed. If for some reason it fails to start, use `make start_httpd` to start it manually.

3.3.3.2 make test Troubleshooting

3.3.3.2.1 make test fails

You cannot run `make test` before you build Apache, so if you told `perl Makefile.PL` not to build the httpd executable, there is no httpd to run the test against. Go to the Apache source tree and run `make`, then return to the mod_perl source tree and continue with the server testing.

3.3.3.2.2 mod_perl.c is incompatible with this version of Apache

If you had a stale old Apache header layout in one of the *include* paths during the build process you will see this message when you try to execute httpd. Run the `find` (or `locate`) utility in order to locate the file *ap_mmn.h*. Delete it and rebuild Apache. RedHat installed a copy of */usr/local/include/ap_mmn.h* on my system.

For all RedHat fans, before you build Apache yourself, do:

```
% rpm -e apache
```

to remove the pre-installed RPM package first!

Users with apt systems would do:

```
% apt-get remove apache
```

instead.

3.3.3.2.3 make test.....skipping test on this platform

While doing `make test` you will notice that some of the tests are reported as *skipped*. The reason is that you are missing some optional modules for these test to be passed. For a hint you might want to peek at the content of each test (you will find them all in the `./t` directory (mnemonic - t, tests). I'll list a few examples, but of course things may change in the future.

```
modules/cookie......skipping test on this platform
modules/request.....skipping test on this platform
```

Install libapreq package which includes among others the `Apache::Request` and `Apache::Cookie` modules.

```
modules/psections...skipping test on this platform
```

Install `Devel::Symdump` and `Data::Dumper`

```
modules/sandwich....skipping test on this platform
```

Install Apache::Sandwich

```
modules/stage.......skipping test on this platform
```

Install Apache::Stage

```
modules/symbol......skipping test on this platform
```

Install Devel::Symdump

Chances are that all of these are installed if you use `CPAN.pm` to `install Bundle::Apache`.

3.3.3.2.4 make test Fails Due to Misconfigured localhost Entry

The `make test` suite uses *localhost* to run the tests that require a network. Make sure you have this entry in */etc/hosts*:

```
127.0.0.1       localhost.localdomain   localhost
```

Also make sure that you have the loopback device [lo] configured. [Hint: try 'ifconfig lo' to test for its existence.]

3.3.4 Installation (make install)

After testing the server, the last step left is to install it. First install all the Perl side files:

```
% make install
```

Then go to the Apache source tree and complete the Apache installation (installing the configuration files, httpd and utilities):

```
% cd ../apache_1.3.xx
% make install
```

Now the installation should be considered complete. You may now configure your server and start using it.

3.3.5 Building Apache and mod_perl by Hand

If you wish to build httpd separately from mod_perl, you should use the `NO_HTTPD=1` option during the `perl Makefile.PL` (mod_perl build) stage. Then you will need to configure various things by hand and proceed to build Apache. You shouldn't run `perl Makefile.PL` before following the steps described in this section.

If you choose to manually build mod_perl, there are three things you may need to set up before the build stage:

- **mod_perl's Makefile**

  When `perl Makefile.PL` is executed, *$APACHE_SRC/modules/perl/Makefile* may need to be modified to enable various options (e.g. `ALL_HOOKS=1`).

Optionally, instead of tweaking the options during `perl Makefile.PL` you may edit *mod_perl-1.xx/src/modules/perl/Makefile* before running `perl Makefile.PL`.

- **Configuration**

  Add to *apache_1.3.xx/src/Configuration* :

  ```
  AddModule modules/perl/libperl.a
  ```

  We suggest you add this entry at the end of the *Configuration* file if you want your callback hooks to have precedence over core handlers.

  Add the following to `EXTRA_LIBS`:

  ```
  EXTRA_LIBS=‘perl -MExtUtils::Embed -e ldopts‘
  ```

  Add the following to `EXTRA_CFLAGS`:

  ```
  EXTRA_CFLAGS=‘perl -MExtUtils::Embed -e ccopts‘
  ```

- **mod_perl Source Files**

  Return to the mod_perl directory and copy the mod_perl source files into the apache build directory:

  ```
  % cp -r src/modules/perl apache_1.3.xx/src/modules/
  ```

When you have done with the configuration parts, run:

```
% perl Makefile.PL NO_HTTPD=1 DYNAMIC=1  EVERYTHING=1\
 APACHE_SRC=../apache_1.3.xx/src
```

`DYNAMIC=1` enables a build of the shared mod_perl library. Add other options if required.

```
% make install
```

Now you may proceed with the plain Apache build process. Note that in order for your changes to the *apache_1.3.xx/src/Configuration* file to take effect, you must run `apache_1.3.xx/src/Configure` instead of the default *apache_1.3.xx/configure* script:

```
% cd ../apache_1.3.xx/src
% ./Configure
% make
% make install
```

3.4 Installation Scenarios for Standalone mod_perl

There are various ways available to build Apache with the new hybrid build environment (using `USE_APACI=1`):

3.4.1 The All-In-One Way

If your goal is just to build and install Apache with mod_perl out of their source trees and have no special interest in further adjusting or enhancing Apache proceed as before:

```
% tar xzvf apache_1.3.xx.tar.gz
% tar xzvf mod_perl-1.xx.tar.gz
% cd mod_perl-1.xx
% perl Makefile.PL APACHE_SRC=../apache_1.3.xx/src \
  DO_HTTPD=1 USE_APACI=1 EVERYTHING=1
% make && make test && make install
% cd ../apache_1.3.xx
% make install
```

This builds Apache statically with mod_perl, installs Apache under the default `/usr/local/apache` tree and mod_perl into the `site_perl` hierarchy of your existing Perl installation. All in one step.

3.4.2 The Flexible Way

This is the normal situation where you want to be flexible while building. Statically building mod_perl into the Apache binary (`httpd`) but via different steps, so you have a chance to include other third-party Apache modules, etc.

1. **Prepare the Apache source tree**

   The first step is as before, extract the distributions:

   ```
   % tar xvzf apache_1.3.xx.tar.gz
   % tar xzvf mod_perl-1.xx.tar.gz
   ```

2. **Install mod_perl's Perl-side and prepare the Apache-side**

   The second step is to install the Perl-side of mod_perl into the Perl hierarchy and prepare the `src/modules/perl/` subdirectory inside the Apache source tree:

   ```
   $ cd mod_perl-1.xx
   $ perl Makefile.PL \
       APACHE_SRC=../apache_1.3.xx/src \
       NO_HTTPD=1 \
       USE_APACI=1 \
       PREP_HTTPD=1 \
       EVERYTHING=1 \
       [...]
   $ make
   $ make install
   $ cd ..
   ```

   The `APACHE_SRC` option sets the path to your Apache source tree, the `NO_HTTPD` option forces this path and only this path to be used, the `USE_APACI` option triggers the new hybrid build environment and the `PREP_HTTPD` option forces preparation of the `APACHE_SRC/modules/perl/` tree but no automatic build.

Then the configuration process prepares the Apache-side of mod_perl in the Apache source tree but doesn't touch anything else in it. It then just builds the Perl-side of mod_perl and installs it into the Perl installation hierarchy.

Important: If you use `PREP_HTTPD` as described above, to complete the build you must go into the Apache source directory and run `make` and `make install`.

3. **Additionally prepare other third-party modules**

   Now you have a chance to prepare third-party modules. For instance the PHP language can be added in a manner similar to the mod_perl procedure.

4. **Build the Apache Package**

   Finally it's time to build the Apache package and thus also the Apache-side of mod_perl and any other third-party modules you've prepared:

```
$ cd apache_1.3.xx
$ ./configure \
    --prefix=/path/to/install/of/apache \
    --activate-module=src/modules/perl/libperl.a \
    [...]
$ make
$ make install
```

   The `--prefix` option is needed if you want to change the default target directory of the Apache installation and the `--activate-module` option activates mod_perl for the configuration process and thus also for the build process. If you choose `--prefix=/usr/share/apache` the Apache directory tree will be installed in */usr/share/apache*.

   Note that the files activated by `--activate-module` do not exist at this time. They will be generated during compilation.

   The last three steps build, test and install the Apache-side of the mod_perl enabled server. Presumably your new server includes third-party components, otherwise you probably won't choose this method of building.

The method described above enables you to insert mod_perl into Apache without having to mangle the Apache source tree for mod_perl. It also gives you the freedom to add third-party modules.

3.4.3 When DSO can be Used

Perl versions prior to 5.6.0, built with `-Dusemymalloc`, and versions 5.6.0 and newer, built with `-Dusemymalloc` and `-Dbincompat5005`, pollute the main `httpd` program with *free* and *malloc* symbols. When `httpd` restarts (happens at startup too), any references in the main program to *free* and *malloc* become invalid, and this causes memory leaks and segfaults.

To determine if you can use a DSO mod_perl with your version of Perl, first find out which malloc your Perl was built with by running:

```
% perl -V:usemymalloc
```

If you get:

```
usemymalloc='n';
```

then it means that Perl is using the system malloc, so mod_perl will work fine as DSO.

If you get:

```
usemymalloc='y';
```

it means that Perl is using its own malloc. If you are running Perl older than 5.6.0, you *must* rebuild Perl with `-Uusemymalloc` in order to use it with a DSO mod_perl. If you are running Perl 5.6.0 or newer, you must either rebuild Perl with `-Uusemymalloc`, or make sure that binary compatibility with Perl 5.005 turned off. To find out, run:

```
% perl -V:bincompat5005
```

If you get:

```
bincompat5005='define';
```

then you *must* either rebuild Perl with `-Ubincompat5005` or with `-Uusemymalloc` to use it with a DSO mod_perl. We recommend that you rebuild Perl with `-Ubincompat5005` if Perl's malloc is a better choice for your OS.

Note that mod_perl's build system issues a warning about this problem.

If you needed to rebuild Perl don't forget to rebuild mod_perl too.

3.4.4 Build mod_perl as a DSO inside the Apache Source Tree via APACI

We have already said that the new mod_perl build environment (`USE_APACI`) is a hybrid. What does it mean? It means for instance that the same `src/modules/perl/` stuff can be used to build mod_perl as a DSO or not, without having to edit anything especially for this. When you want to build `libperl.so` all you have to do is to add one single option to the above steps.

3.4.4.1 libperl.so and libperl.a

`libmodperl.so` would be more correct for the mod_perl file, but the name has to be `libperl.so` because of prehistoric Apache issues. Don't confuse the `libperl.so` for mod_perl with the file of the same name which comes with Perl itself. They are two different things. It is unfortunate that they happen to have the same name.

There is also a `libperl.a` which comes with the Perl installation. That's different too.

You have two options here, depending on which way you have chosen above: If you choose the All-In-One way from above then add

```
USE_DSO=1
```

to the `perl Makefile.PL` options. If you choose the Flexible way then add:

```
--enable-shared=perl
```

to Apache's `./configure` options.

As you can see only an additional `USE_DSO=1` or `--enable-shared=perl` option is needed. Everything else is done automatically: `mod_so` is automatically enabled, the Makefiles are adjusted automatically and even the `install` target from APACI now additionally installs `libperl.so` into the Apache installation tree. And even more: the `LoadModule` and `AddModule` directives (which dynamically load and insert mod_perl into httpd) are automatically added to *httpd.conf*.

3.4.5 Build mod_perl as a DSO outside the Apache Source Tree via APXS

Above we've seen how to build mod_perl as a DSO *inside* the Apache source tree. But there is a nifty alternative: building mod_perl as a DSO *outside* the Apache source tree via the new Apache 1.3 support tool `apxs` (APache eXtension). The advantage is obvious: you can extend an already installed Apache with mod_perl even if you don't have the sources (for instance, you may have installed an Apache binary package from your vendor).

Here are the build steps:

```
% tar xzvf mod_perl-1.xx.tar.gz
% cd mod_perl-1.xx
% perl Makefile.PL \
  USE_APXS=1 \
  WITH_APXS=/path/to/bin/apxs \
  EVERYTHING=1 \
   [...]
% make && make test && make install
```

This will build the DSO `libperl.so` *outside* the Apache source tree with the new Apache 1.3 support tool `apxs` and install it into the existing Apache hierarchy.

3.5 Installation Scenarios for mod_perl and Other Components

([ReaderMETA]: Please send more scenarios of mod_perl + other components installation guidelines. Thanks!)

You have now seen very detailed installation instructions for specific cases, but since mod_perl is used with many other components that plug into Apache, you will definitely want to know how to build them together with mod_perl.

Since all the steps are simple, and assuming that you now understand how the build process works, I'll show only the commands to be executed with no comments unless there is something we haven't discussed before.

Generally every example that I'm going to show consist of:

1. downloading the source distributions of the components to be used
2. un-packing them
3. configuring them
4. building Apache using the parameters appropriate to each component
5. `make test` and `make install`.

All these scenarios were tested on a Linux platform, you might need to refer to the specific component's documentation if something doesn't work for you as described below. The intention of this section is not to show you how to install other non-mod_perl components alone, but how to do this in a bundle with mod_perl.

Also, notice that the links I've used below are very likely to have changed by the time you read this document. That's why I have used the *x.x.x* convention, instead of using hardcoded version numbers. Remember to replace the *xx* place-holders with the version numbers of the distributions you are about to use. To find out the latest stable version number, visit the components' sites. So if the instructions say:

```
http://apache.org/dist/perl/mod_perl-1.xx.tar.gz
```

go to http://perl.apache.org/download/ in order to learn the version number of the latest stable release and download the appropriate file.

Unless otherwise noted, all the components install themselves into a default location. When you run `make install` the installation program tells you where it's going to install the files.

3.5.1 mod_perl and mod_ssl (+openssl)

mod_ssl provides strong cryptography for the Apache 1.3 webserver via the Secure Sockets Layer (SSL v2/v3) and Transport Layer Security (TLS v1) protocols by the help of the Open Source SSL/TLS toolkit OpenSSL, which is based on SSLeay from Eric A. Young and Tim J. Hudson.

Download the sources:

```
% lwp-download http://www.apache.org/dist/apache_1.3.xx.tar.gz
% lwp-download http://apache.org/dist/perl/mod_perl-1.xx.tar.gz
% lwp-download http://www.modssl.org/source/mod_ssl-x.x.x-x.x.x.tar.gz
% lwp-download http://www.openssl.org/source/openssl-x.x.x.tar.gz
```

Un-pack:

```
% tar xvzf mod_perl-1.xx
% tar xvzf apache_1.3.xx.tar.gz
% tar xvzf mod_ssl-x.x.x-x.x.x.tar.gz
% tar xvzf openssl-x.x.x.tar.gz
```

Configure, build and install openssl:

```
% cd openssl-x.x.x
% ./config
% make && make test && make install
```

Configure mod_ssl:

```
% cd mod_ssl-x.x.x-x.x.x
% ./configure --with-apache=../apache_1.3.xx
```

Configure mod_perl:

```
% cd ../mod_perl-1.xx
% perl Makefile.PL USE_APACI=1 EVERYTHING=1 \
      DO_HTTPD=1 SSL_BASE=/usr/local/ssl \
      APACHE_PREFIX=/usr/local/apachessl \
      APACHE_SRC=../apache_1.3.xx/src \
      APACI_ARGS='--enable-module=ssl,--enable-module=rewrite'
```

Note: Do not forget that if you use `csh` or `tcsh` you may need to put all the arguments to 'perl Makefile.PL' on a single command line.

Note: If when specifying `SSL_BASE=/usr/local/ssl` Apache's configure doesn't find the ssl libraries, please refer to the mod_ssl documentation to figure out what `SSL_BASE` argument it expects (usually needed when ssl is not installed in the standard places). This topic is out of scope of this document. For some setups using `SSL_BASE=/usr/local` does the trick.

Build, test and install:

```
% make && make test && make install
% cd ../apache_1.3.xx
% make certificate
% make install
```

Now proceed with the mod_ssl and mod_perl parts of the server configuration before starting the server.

When the server starts you should see the following or similar in the *error_log* file:

```
[Fri Nov 12 16:14:11 1999] [notice] Apache/1.3.9 (Unix)
mod_perl/1.21_01-dev mod_ssl/2.4.8 OpenSSL/0.9.4 configured
-- resuming normal operations
```

3.5.2 mod_perl and mod_ssl Rolled from RPMs

As in the previous section this shows an installation of mod_perl and mod_ssl, but this time using sources/binaries prepackaged in RPMs.

As always, replace *xx* with the proper version numbers. And replace `i386` with the identifier for your platform if it is different.

1.
   ```
   % get apache-mod_ssl-x.x.x.x-x.x.x.src.rpm
   ```
   Source: http://www.modssl.org
2.
   ```
   % get openssl-x.x.x.i386.rpm
   ```
   Source: http://www.openssl.org/
3.
   ```
   % lwp-download http://apache.org/dist/perl/mod_perl-1.xx.tar.gz
   ```
   Source: http://apache.org/dist/perl
4.
   ```
   % lwp-download http://www.engelschall.com/sw/mm/mm-x.x.xx.tar.gz
   ```
   Source: http://www.engelschall.com/sw/mm/
5.
   ```
   % rpm -ivh openssl-x.x.x.i386.rpm
   ```
6.
   ```
   % rpm -ivh apache-mod_ssl-x.x.x.x-x.x.x.src.rpm
   ```
7.
   ```
   % cd /usr/src/redhat/SPECS
   ```
8.
   ```
   % rpm -bp apache-mod_ssl.spec
   ```
9.

```
% cd /usr/src/redhat/BUILD/apache-mod_ssl-x.x.x.x-x.x.x
```

10.

```
% tar xvzf mod_perl-1.xx.tar.gz
```

11.

```
% cd mod_perl-1.xx
```

12.

```
% perl Makefile.PL APACHE_SRC=../apache_1.3.xx/src \
   DO_HTTPD=1 \
   USE_APACI=1 \
   PREP_HTTPD=1 \
   EVERYTHING=1
```

Add or remove parameters if appropriate.

13.

```
% make
```

14.

```
% make install
```

15.

```
% cd ../mm-x.x.xx/
```

16.

```
% ./configure --disable-shared
```

17.

```
% make
```

18.

```
% cd ../mod_ssl-x.x.x-x.x.x
```

19.

```
% ./configure \
      --with-perl=/usr/bin/perl \
      --with-apache=../apache_1.3.xx\
      --with-ssl=SYSTEM \
      --with-mm=../mm-x.x.x \
      --with-layout=RedHat \
      --disable-rule=WANTHSREGEX \
      --enable-module=all \
      --enable-module=define \
      --activate-module=src/modules/perl/libperl.a \
      --enable-shared=max \
      --disable-shared=perl
```

20.

```
% make
```

21.

```
% make certificate
```

with whatever option is suitable to your configuration.

22.

```
% make install
```

You should be all set.

Note: If you use the standard config for mod_ssl don’t forget to run Apache like this:

```
% httpd -DSSL
```

3.5.3 mod_perl and apache-ssl (+openssl)

Apache-SSL is a secure Webserver, based on Apache and SSLeay/OpenSSL. It is licensed under a BSD-style license which means, in short, that you are free to use it for commercial or non-commercial purposes, so long as you retain the copyright notices.

Download the sources:

```
% lwp-download http://www.apache.org/dist/apache_1.3.xx.tar.gz
% lwp-download http://apache.org/dist/perl/mod_perl-1.xx.tar.gz
% lwp-download http://www.apache-ssl.org/.../apache_1.3.xx+ssl_x.xx.tar.gz
% lwp-download http://www.openssl.org/source/openssl-x.x.x.tar.gz
```

Un-pack:

```
% tar xvzf mod_perl-1.xx
% tar xvzf apache_1.3.xx.tar.gz
% tar xvzf openssl-x.x.x.tar.gz
```

Configure and install openssl:

```
% cd openssl-x.x.x
% ./config
% make && make test && make install
```

Patch Apache with SSLeay paths

```
% cd apache_x.xx
% tar xzvf ../apache_1.3.xx+ssl_x.xx.tar.gz
% FixPatch
Do you want me to apply the fixed-up Apache-SSL patch for you? [n] y
```

Now edit the *src/Configuration* file if needed and then configure:

```
% cd ../mod_perl-1.xx
% perl Makefile.PL USE_APACI=1 EVERYTHING=1 \
      DO_HTTPD=1 SSL_BASE=/usr/local/ssl \
      APACHE_SRC=../apache_1.3.xx/src
```

Build, test and install:

```
% make && make test && make install
% cd ../apache_1.3.xx/src
% make certificate
% make install
```

Note that you might need to modify the ’make test’ stage, as it takes much longer for this server to get started and `make test` waits only a few seconds for Apache to start before it times out.

Now proceed with configuration of the apache_ssl and mod_perl parts of the server configuration files, before starting the server.

3.5.4 mod_perl and Stronghold

Stronghold is a secure SSL Web server for Unix which allows you to give your web site full-strength, 128-bit encryption.

You must first build and install Stronghold without mod_perl, following Stronghold’s install procedure. For more information visit http://www.c2.net/products/sh2/ .

Having done that, download the sources:

```
% lwp-download http://apache.org/dist/perl/mod_perl-1.xx.tar.gz
```

Unpack:

```
% tar xvzf mod_perl-1.xx.tar.gz
```

Configure (assuming that you have the Stronghold sources extracted at */usr/local/stronghold*):

```
% cd mod_perl-1.xx
% perl Makefile.PL APACHE_SRC=/usr/local/stronghold/src \
  DO_HTTPD=1 USE_APACI=1 EVERYTHING=1
```

Build:

```
% make
```

Before running `make test`, you must add your `StrongholdKey` to *t/conf/httpd.conf*. If you are configuring by hand, be sure to edit *src/modules/perl/Makefile* and uncomment the `#APACHE_SSL` directive.

Test and Install:

```
% make test && make install
% cd /usr/local/stronghold
% make install
```

3.5.4.1 Note For Solaris 2.5 users

There has been a report related to the REGEX library that comes with Stronghold, that after building Apache with mod_perl it would produce core dumps. To work around this problem, in *$STRONGHOLD/src/Configuration* change:

```
Rule WANTHSREGEX=default
```

to:

```
Rule WANTHSREGEX=no
```

3.5.5 mod_perl and mod_php

This is a simple installation scenario of the mod_perl and mod_php in Apache server:

1. Configure Apache.

   ```
   % cd apache_1.3.xx
   % ./configure --prefix=/usr/local/etc/httpd
   ```

   (this step might be redundant with the recent versions of mod_php, but it's harmless.)

2. Build mod_perl.

   ```
   % cd ../mod_perl-1.xx
   % perl Makefile.PL APACHE_SRC=../apache_1.3.xxx/src NO_HTTPD=1 \
     USE_APACI=1 PREP_HTTPD=1 EVERYTHING=1
   % make
   ```

3. Build mod_php.

   ```
   % cd ../php-x.x.xx
   % ./configure --with-apache=../apache_1.3.xxx \
      --with-mysql --enable-track-vars
   % make
   % make install
   ```

4. Build Apache:

   ```
   % cd ../apache_1.3.xxx
   % ./configure --prefix=/usr/local/etc/httpd \
      --activate-module=src/modules/perl/libperl.a \
      --activate-module=src/modules/php4/libphp4.a \
      --enable-module=stats \
      --enable-module=rewrite
   % make
   ```

Note: *libperl.a* and *libphp4.a* do not exist at this time. They will be generated during compilation.

5. Test and install mod_perl

```
% cd ../mod_perl-1.xx
% make test
# make install.
```

6. Complete the Apache installation.

```
% cd ../apache_1.3.xxx
# make install
```

Note: If you need to build mod_ssl as well, make sure that you add the mod_ssl first.

3.6 mod_perl Installation with the CPAN.pm Interactive Shell

Installation of mod_perl and all the required packages is much easier with help of the `CPAN.pm` module, which provides you among other features with a shell interface to the CPAN repository. CPAN is the Comprehensive Perl Archive Network, a repository of thousands of Perl modules, scripts as well as a vast amount of documentation. See http://cpan.org for more information.

The first thing first is to download the Apache source code and unpack it into a directory -- the name of which you will need very soon.

Now execute:

```
% perl -MCPAN -eshell
```

If it's the first time that you have used it, `CPAN.pm` will ask you about a dozen questions to configure the module. It's quite easy to accomplish this task, and very helpful hints come along with the questions. When you are finished you will see the `CPAN` prompt:

```
cpan>
```

It can be a good idea to install a special `CPAN` bundle of modules to make using the CPAN module easier. Installation is as simple as typing:

```
cpan> install Bundle::CPAN
```

The `CPAN` shell can download mod_perl for you, unpack it, check for prerequisites, detect any missing third party modules, and download and install them. All you need to do to install mod_perl is to type at the prompt:

```
cpan> install mod_perl
```

You will see (I'll use `x.xx` as a placeholder for real version numbers, since these change very frequently):

```
Running make for DOUGM/mod_perl-1.xx.tar.gz
Fetching with LWP:
http://www.cpan.org/authors/id/DOUGM/mod_perl-1.xx.tar.gz

CPAN.pm: Going to build DOUGM/mod_perl-1.xx.tar.gz

Enter 'q' to stop search
Please tell me where I can find your apache src
[../apache_1.3.xx/src]
```

`CPAN.pm` will search for the latest Apache sources and suggest a directory. Here, unless the CPAN shell found it and suggested the right directory, you need to type the directory into which you unpacked Apache. The next question is about the `src` directory, which resides at the root level of the unpacked Apache distribution. In most cases the CPAN shell will suggest the correct directory.

```
Please tell me where I can find your apache src
[../apache_1.3.xx/src]
```

Answer yes to all the following questions, unless you have a reason not to do that.

```
Configure mod_perl with /usr/src/apache_1.3.xx/src ? [y]
Shall I build httpd in /usr/src/apache_1.3.xx/src for you? [y]
```

Now we will build Apache with mod_perl enabled. Quit the `CPAN` shell, or use another terminal. Go to the Apache sources root directory and run:

```
% make install
```

which will complete the installation by installing Apache's headers and the binary in the appropriate directories.

The only caveat of the process I've described is that you don't have control over the configuration process. Actually, that problem is easy to solve -- you can tell `CPAN.pm` to pass whatever parameters you want to `perl Makefile.PL`. You do this with `o conf makepl_arg` command:

```
cpan> o conf makepl_arg 'DO_HTTPD=1 USE_APACI=1 EVERYTHING=1'
```

Just list all the parameters as if you were passing them to the familiar `perl Makefile.PL`. If you add the `APACHE_SRC=/usr/src/apache_1.3.xx/src` and `DO_HTTPD=1` parameters, you will not be asked a single question. Of course you must give the correct path to the Apache source distribution.

Now proceed with `install mod_perl` as before. When the installation is completed, remember to unset the `makepl_arg` variable by executing:

```
cpan> o conf makepl_arg ''
```

If you have previously set `makepl_arg` to some value, before you alter it for the mod_perl installation you will probably want to save it somewhere so that you can restore it when you have finished with the mod_perl installation. To see the original value, use:

```
cpan> o conf makepl_arg
```

You can now install all the modules you might want to use with mod_perl. You install them all by typing a singe command:

```
cpan> install Bundle::Apache
```

This will install mod_perl if isn't yet installed, and many other packages such as: `ExtUtils::Embed`, `MIME::Base64`, `URI::URL`, `Digest::MD5`, `Net::FTP`, `LWP`, `HTML::TreeBuilder`, `CGI`, `Devel::Symdump`, `Apache::DB`, `Tie::IxHash`, `Data::Dumper` etc.

A helpful hint: If you have a system with all the Perl modules you use and you want to replicate them all elsewhere, and if you cannot just copy the whole `/usr/lib/perl5` directory because of a possible binary incompatibility on the other system, making your own bundle is a handy solution. To accomplish this the command `autobundle` can be used on the CPAN shell command line. This command writes a bundle definition file for all modules that are installed for the currently running perl interpreter.

With the clever bundle file you can then simply say

```
cpan> install Bundle::my_bundle
```

and after answering a few questions, go out for a coffee.

3.7 Installing on multiple machines

You may wish to build httpd once, then copy it to other machines. The Perl side of mod_perl needs the Apache headers files to compile. To avoid dragging and build Apache on all your other machines, there are a few Makefile targets to help you out:

```
% make tar_Apache
```

This will tar all files mod_perl installs in your Perl's *site_perl* directory, into a file called *Apache.tar*. You can then unpack this under the *site_perl* directory on another machine.

```
% make offsite-tar
```

This will copy all the header files from the Apache source directory which you configured mod_perl against, then it will `make dist` which creates a mod_perl-1.xx.tar.gz, ready to unpack on another machine to compile and install the Perl side of mod_perl.

If you really want to make your life easy you should use one of the more advanced packaging systems. For example, almost all Linux OS distributions use packaging tools on top of plain tar.gz, allowing you to track prerequisites for each package, and providing easy installation, upgrade and cleanup. One of the most widely-used packagers is RPM (Red Hat Package Manager). See http://www.rpm.org for more information.

All you have to do is prepare a SRPM (source distribution package), then build a binary release. This can be installed on any number of machines in a matter of seconds.

It will even work on live machines! If you have two identical machines (identical software and hardware, although depending on your setup hardware may be less critical). Let's say that one is a live server and the other is in development. You build an RPM with a mod_perl binary distribution, install it on the development machine and satisfy yourself that it is working and stable. You can then install the RPM package on the live server without any fear. Make sure that *httpd.conf* is correct, since it generally includes parameters such as hostname which are unique to the live machine.

When you have installed the package, just restart the server. It can be a good idea to keep a package of the previous system, in case something goes wrong. You can then easily remove the installed package and put the old one back.

([ReaderMETA]: Dear reader, Can you please share a step by step scenario of preparation of SRPMs for mod_perl? Thanks!!!)

3.8 using RPM and other packages to install mod_perl

[ReaderMETA]: Currently only RPM package. Please submit info about other available packages if you use such.

3.8.1 A word on mod_perl RPM packages

The virtues of RPM packages is a subject of much debate among mod_perl users. While RPMs do take the pain away from package installation and maintenance for most applications, the nuances of mod_perl make RPMs somewhat less than ideal for those just getting started. The following help and advice is for those new to mod_perl, Apache, Linux, and RPMs. If you know what you are doing, this is probably Old Hat - contributing your past experiences is, as always, welcomed by the community.

3.8.2 Getting Started

If you are new to mod_perl and are using this Guide and the Eagle Book to help you on your way, it is probably better to grab the latest Apache and mod_perl sources and compile the sources yourself. Not only will you find that this is less daunting than you suspect, but it will probably save you a few headaches down the line for several reasons.

First, given the pace at which the open source community produces software, RPMs, especially those found on distribution CDs, are often several versions out of date. The most recent version will not only be more stable, but will likely incorporate some new functionality that you will eventually want to play with.

It is also unlikely that the file system layout of an RPM package will match what you see in the Eagle Book and this Guide. If you are new to mod_perl, Apache, or both you will probably want to get familiar with the file system layout used by the examples given here before trying something non-standard.

Finally, the RPMs found on a typical distribution's CDs use mod_perl built with Apache's Dynamic Shared Objects (`DSO`) support. While mod_perl can be successfully used as a DSO module, it adds a layer of complexity that you may want to live without for now.

All that being said, should you still feel that rolling your own mod_perl enabled Apache server is not likely, here are a few helpful hints...

3.8.3 Compiling RPM source files

It is possible to compile the source files provided by RPM packages, but if you are using RPMs to ease mod_perl installation, that is not the way to do it. Both Apache and mod_perl RPMs are designed to be install-and-go. If you really want to compile mod_perl to your own specific needs, your best bet is to get the most recent sources from CPAN.

3.8.4 Mix and Match RPM and source

It is probably not the best idea to use a self-compiled Apache with a mod_perl RPM (or vice versa). Sticking with one format or the other at first will result in fewer headaches and more hair.

3.8.5 Installing a single apache+mod_perl RPM

If you use an Apache+mod_perl RPM, chances are `rpm -i` or `glint` (GUI for RPM) will have you up and running immediately, no compilation necessary. If you encounter problems, try downloading from another mirror site or searching http://rpmfind.net/ for a different package - there are plenty out there to choose from.

David Harris has started an effort to build better RPM/SRPM mod_perl packages. You will find the link to David’s site from Binary distributions.

Features of this RPM:

- Installs mod_perl as an "add in" to the RedHat Apache package, but does not install mod_perl as a DSO.
- Includes the four header files required for building `libapreq` (`Apache::Request`)
- Distributes plain text forms of the pod documentation files that come with mod_perl.
- Checks the module magic number on the existing Apache package to see if things are compatible

Notes on this un-conventional RPM packaging of mod_perl

by David Harris <dharris (at) drh.net> on Oct 13, 1999

This package will install the mod_perl library files on your machine along with the following two Apache files:

```
/usr/lib/apache/mod_include_modperl.so
/usr/sbin/httpd_modperl
```

This package does not install a complete Apache subtree built with mod_perl, but rather just the two above files that are different for mod_perl. This conceptually thinks of mod_perl as a kind of an "add on" that we would like to add to the regular Apache tree. However, we are prevented from distributing mod_perl as an actual DSO, because it is not recommended by the mod_perl developers and various features must be turned off. So, instead, we distribute an httpd binary with mod_perl statically linked (httpd_modperl) and the special modified mod_include.so required for this binary (mod_include_modperl.so). You can use the exact same configuration files and other DSO modules, but you just "enable" the mod_perl "add on" by following the directions below.

To enable mod_perl, do the following:

```
(1) Configure /etc/rc.d/init.d/httpd to run httpd_modperl instead of
    httpd by changing the "daemon" command line.
(2) Replace mod_include.so with mod_include_modperl.so in the
    module loading section of /etc/httpd/conf/httpd.conf
(3) Uncomment the "AddModule mod_perl.c" line in /etc/httpd/conf/httpd.conf
```

Or run the following command:

```
/usr/sbin/modperl-enable on
```

and to disable mod_perl:

```
/usr/sbin/modperl-enable off
```

3.8.6 Compiling libapreq (Apache::Request) with the RH 6.0 mod_perl RPM

Libapreq provides the Apache::Request module.

Despite many reports of libapreq not working properly with various RPM packages, it is possible to integrate libapreq with mod_perl RPMs. It just requires a few additional steps.

1. Make certain you have the `apache-devel-x.x.x-x.i386.rpm` package installed. Also, download the latest version of libapreq from CPAN.

2. Install the source RPM for your mod_perl RPM and then do a build prep, (with `rpm -bp apache-devel-x.x.x-x.src.rpm`) which unpacks the sources. From there, copy the four header files (*mod_perl.h*, *mod_perl_version.h*, *mod_perl_xs.h*, and *mod_PL.h*) to `/usr/include/apache`.

   - 2.1 Get the SRPM from `somemir-ror.../redhat-x.x/SRPMS/mod_perl-1.xx-x.src.rpm`.

   - 2.2 Install the SRPM. This creates files in `/usr/src/redhat/SPECS` and `/usr/src/redhat/SOURCES`. Run:

```
% rpm -ih mod_perl-1.xx-x.src.rpm
```

- 2.3 Do a "prep" build of the package, which just unpackages the sources and applies any patches.

```
% rpm -bp /usr/src/redhat/SPECS/mod_perl.spec
Executing: %prep
+ umask 022
+ cd /usr/src/redhat/BUILD
+ cd /usr/src/redhat/BUILD
+ rm -rf mod_perl-1.19
+ /bin/gzip -dc /usr/src/redhat/SOURCES/mod_perl-1.19.tar.gz
+ tar -xf -
+ STATUS=0
+ [ 0 -ne 0 ]
+ cd mod_perl-1.19
++ /usr/bin/id -u
+ [ 0 = 0 ]
+ /bin/chown -Rf root .
++ /usr/bin/id -u
+ [ 0 = 0 ]
+ /bin/chgrp -Rf root .
+ /bin/chmod -Rf a+rX,g-w,o-w .
+ echo Patch #0:
Patch #0:
+ patch -p1 -b --suffix .rh -s
+ exit 0
```

NOTE: Steps 2.1 through 2.3 are just a fancy un-packing of the source tree that builds the RPM into /usr/src/redhat/BUILD/mod_perl-1.xx. You could unpack the mod_perl-1.xx.tar.gz file somewhere and then do the following steps on that source tree. The method shown above is more "pure" because you're grabbing the header files from the same tree that built the RPM. But this does not matter because RedHat is not patching that file. So, it might be better if you just grab the mod_perl source and unpack it to get these files. Less fuss and mess.

- 2.4 Look at the files you will copy: (this is not really a step, but useful to show)

```
% find /usr/src/redhat/BUILD/mod_perl-1.19 -name '*.h'
/usr/src/redhat/BUILD/mod_perl-1.19/src/modules/perl/mod_perl.h
/usr/src/redhat/BUILD/mod_perl-1.19/src/modules/perl/mod_perl_xs.h
/usr/src/redhat/BUILD/mod_perl-1.19/src/modules/perl/mod_perl_version.h
/usr/src/redhat/BUILD/mod_perl-1.19/src/modules/perl/perl_PL.h
```

- 2.5 Copy the files into /usr/include/apache.

```
% find /usr/src/redhat/BUILD/mod_perl-1.19 -name '*.h' \
  -exec cp {} /usr/include/apache \;
```

NOTE: You should not have to do:

```
% mkdir /usr/include/apache
```

because that directory should be created by apache-devel.

3. Apply this patch to libapreq: http://www.davideous.com/modperl-rpm/distrib/libapreq-0.31_include.patch

4. Follow the libapreq directions as usual:

```
% perl Makefile.PL
% make && make test && make install
```

3.8.7 Installing separate Apache and mod_perl RPMs

If you are trying to install separate Apache and mod_perl RPMs, like those provided by the RedHat distributions, you may be in for a bit of a surprise. Installing the Apache RPM will go just fine, and http://localhost will bring up some type of web page for you. However, after installation of the mod_perl RPM, the How can I tell whether mod_perl is running test will show that Apache is not mod_perl enabled. This is because mod_perl needs to be added as a separate module using Apache's Dynamic Shared Objects.

To use mod_perl as a DSO, make the following modifications to your Apache configuration files:

```
httpd.conf:
----------
LoadModule perl_module modules/libperl.so
AddModule mod_perl.c

PerlModule Apache::Registry
Alias /perl/ /home/httpd/perl/
<Location /perl>
  SetHandler perl-script
  PerlHandler Apache::Registry
  PerlSendHeader On
  Options +ExecCGI
</Location>
```

After a complete shutdown and startup of the server, mod_perl should be up and running.

3.8.8 Testing the mod_perl API

Some people have reported that even when the server responds positively to the How can I tell whether mod_perl is running tests, the mod_perl API will not function properly. You may want to run the following script to verify the availability of the mod_perl API.

```
use strict;
my $r = shift;
$r->send_http_header('text/html');
$r->print("It worked!!!\n");
```

3.9 Installation Without Superuser Privileges

As you have already learned, mod_perl enabled Apache consists of two main components: Perl modules and Apache itself. Let's tackle the tasks one at a time.

I'll show a complete installation example using *stas* as a username, assuming that */home/stas* is the home directory of that user.

3.9.1 Installing Perl Modules into a Directory of Choice

Since without superuser permissions you aren't allowed to install modules into system directories like */usr/lib/perl5*, you need to find out how to install the modules under your home directory. It's easy.

First you have to decide where to install the modules. The simplest approach is to simulate the portion of the / file system relevant to Perl under your home directory. Actually we need only two directories:

```
/home/stas/bin
/home/stas/lib
```

We don't have to create them, since that will be done automatically when the first module is installed. 99% of the files will go into the *lib* directory. Occasionally, when some module distribution comes with Perl scripts, these will go into the *bin* directory. This directory will be created if it doesn't exist.

Let's install the *CGI.pm* package, which includes a few other `CGI::*` modules. As usual, download the package from the CPAN repository, unpack it and `chdir` to the newly-created directory.

Now do a standard `perl Makefile.PL` to prepare a *Makefile*, but this time tell `MakeMaker` to use your Perl installation directories instead of the defaults.

```
% perl Makefile.PL PREFIX=/home/stas
```

`PREFIX=/home/stas` is the only part of the installation process which is different from usual. Note that if you don't like how `MakeMaker` chooses the rest of the directories, or if you are using an older version of it which requires an explicit declaration of all the target directories, you should do this:

```
% perl Makefile.PL PREFIX=/home/stas \
  INSTALLPRIVLIB=/home/stas/lib/perl5 \
  INSTALLSCRIPT=/home/stas/bin \
  INSTALLSITELIB=/home/stas/lib/perl5/site_perl \
  INSTALLBIN=/home/stas/bin \
  INSTALLMAN1DIR=/home/stas/lib/perl5/man  \
  INSTALLMAN3DIR=/home/stas/lib/perl5/man3
```

The rest is as usual:

```
% make
% make test
% make install
```

`make install` installs all the files in the private repository. Note that all the missing directories are created automatically, so there is no need to create them in first place. Here (slightly edited) is what it does :

```
Installing /home/stas/lib/perl5/CGI/Cookie.pm
Installing /home/stas/lib/perl5/CGI.pm
Installing /home/stas/lib/perl5/man3/CGI.3
Installing /home/stas/lib/perl5/man3/CGI::Cookie.3
Writing /home/stas/lib/perl5/auto/CGI/.packlist
Appending installation info to /home/stas/lib/perl5/perllocal.pod
```

If you have to use the explicit target parameters, instead of a single `PREFIX` parameter, you will find it useful to create a file called for example *~/.perl_dirs* (where ~ is `/home/stas` in our example) containing:

```
PREFIX=/home/stas \
INSTALLPRIVLIB=/home/stas/lib/perl5 \
INSTALLSCRIPT=/home/stas/bin \
INSTALLSITELIB=/home/stas/lib/perl5/site_perl \
INSTALLBIN=/home/stas/bin \
INSTALLMAN1DIR=/home/stas/lib/perl5/man  \
INSTALLMAN3DIR=/home/stas/lib/perl5/man3
```

From now on, any time you want to install perl modules locally you simply execute:

```
% perl Makefile.PL ‘cat ~/.perl_dirs‘
% make
% make test
% make install
```

Using this method you can easily maintain several Perl module repositories. For example, you could have one for production Perl and another for development:

```
% perl Makefile.PL ‘cat ~/.perl_dirs.production‘
```

or

```
% perl Makefile.PL ‘cat ~/.perl_dirs.develop‘
```

3.9.2 Making Your Scripts Find the Locally Installed Modules

Perl modules are generally placed in four main directories. To find these directories, execute:

```
% perl -V
```

The output contains important information about your Perl installation. At the end you will see:

```
Characteristics of this binary (from libperl):
Built under linux
Compiled at Apr  6 1999 23:34:07
@INC:
  /usr/lib/perl5/5.00503/i386-linux
  /usr/lib/perl5/5.00503
  /usr/lib/perl5/site_perl/5.005/i386-linux
  /usr/lib/perl5/site_perl/5.005
  .
```

It shows us the content of the Perl special variable `@INC`, which is used by Perl to look for its modules. It is equivalent to the `PATH` environment variable in Unix shells which is used to find executable programs.

Notice that Perl looks for modules in the . directory too, which stands for the current directory. It's the last entry in the above output.

Of course this example is from version *5.00503* of Perl installed on my x86 architecture PC running Linux. That's why you see *i386-linux* and *5.00503*. If your system runs a different version of Perl, operating system, processor or chipset architecture, then some of the directories will have different names.

I also have a perl-5.6.0 installed under `/usr/local/lib/` so when I do:

```
% /usr/local/bin/perl5.6.0 -V
```

I see:

```
@INC:
  /usr/local/lib/perl5/5.6.0/i586-linux
  /usr/local/lib/perl5/5.6.0
  /usr/local/lib/site_perl/5.6.0/i586-linux
  /usr/local/lib/site_perl
```

Note that it's still *Linux*, but the newer Perl version uses the version of my Pentium processor (thus the *i586* and not *i386*). This makes use of compiler optimizations for Pentium processors when the binary Perl extensions are created.

All the platform specific files, such as compiled C files glued to Perl with `XS` or `SWIG`, are supposed to go into the `i386-linux`-like directories.

Important: As we have installed the Perl modules into non-standard directories, we have to let Perl know where to look for the four directories. There are two ways to accomplish this. You can either set the `PERL5LIB` environment variable, or you can modify the `@INC` variable in your scripts.

Assuming that we use perl-5.00503, in our example the directories are:

```
/home/stas/lib/perl5/5.00503/i386-linux
/home/stas/lib/perl5/5.00503
/home/stas/lib/perl5/site_perl/5.005/i386-linux
/home/stas/lib/perl5/site_perl/5.005
```

As mentioned before, you find the exact directories by executing `perl -V` and replacing the global Perl installation's base directory with your home directory.

Modifying `@INC` is quite easy. The best approach is to use the `lib` module (pragma), by adding the following snippet at the top of any of your scripts that require the locally installed modules.

```
use lib qw(/home/stas/lib/perl5/5.00503/
           /home/stas/lib/perl5/site_perl/5.005);
```

Another way is to write code to modify `@INC` explicitly:

```
BEGIN {
  unshift @INC,
    qw(/home/stas/lib/perl5/5.00503
       /home/stas/lib/perl5/5.00503/i386-linux
       /home/stas/lib/perl5/site_perl/5.005
       /home/stas/lib/perl5/site_perl/5.005/i386-linux);
      }
```

Note that with the `lib` module we don't have to list the corresponding architecture specific directories, since it adds them automatically if they exist (to be exact, when *$dir/$archname/auto* exists).

Also, notice that both approaches *prepend* the directories to be searched to `@INC`. This allows you to install a more recent module into your local repository and Perl will use it instead of the older one installed in the main system repository.

Both approaches modify the value of `@INC` at compilation time. The `lib` module uses the *BEGIN* block as well, but internally.

Now, let's assume the following scenario. I have installed the `LWP` package in my local repository. Now I want to install another module (e.g. mod_perl) and it has `LWP` listed in its prerequisites list. I know that I have `LWP` installed, but when I run `perl Makefile.PL` for the module I'm about to install I'm told that I don't have `LWP` installed.

There is no way for Perl to know that we have some locally installed modules. All it does is search the directories listed in `@INC`, and since the latter contains only the default four directories (plus the . directory), it cannot find the locally installed `LWP` package. We cannot solve this problem by adding code to modify `@INC`, but changing the `PERL5LIB` environment variable will do the trick. If you are using `t?csh` for interactive work, do this:

```
setenv PERL5LIB /home/stas/lib/perl5/5.00503:
/home/stas/lib/perl5/site_perl/5.005
```

It should be a single line with directories separated by colons (`:`) and no spaces. If you are a `(ba)?sh` user, do this:

```
export PERL5LIB=/home/stas/lib/perl5/5.00503:
/home/stas/lib/perl5/site_perl/5.005
```

Again make it a single line. If you use bash you can use multi-line commands by terminating split lines with a backslash (\), like this:

```
export PERL5LIB=/home/stas/lib/perl5/5.00503:\
/home/stas/lib/perl5/site_perl/5.005
```

As with `use lib`, perl automatically prepends the architecture specific directories to `@INC` if those exist.

When you have done this, verify the value of the newly configured `@INC` by executing `perl -V` as before. You should see the modified value of `@INC`:

```
% perl -V

Characteristics of this binary (from libperl):
Built under linux
Compiled at Apr  6 1999 23:34:07
%ENV:
  PERL5LIB="/home/stas/lib/perl5/5.00503:
  /home/stas/lib/perl5/site_perl/5.005"
@INC:
  /home/stas/lib/perl5/5.00503/i386-linux
  /home/stas/lib/perl5/5.00503
  /home/stas/lib/perl5/site_perl/5.005/i386-linux
  /home/stas/lib/perl5/site_perl/5.005
  /usr/lib/perl5/5.00503/i386-linux
  /usr/lib/perl5/5.00503
  /usr/lib/perl5/site_perl/5.005/i386-linux
  /usr/lib/perl5/site_perl/5.005
  .
```

When everything works as you want it to, add these commands to your *.tcshrc* or *.bashrc* file. The next time you start a shell, the environment will be ready for you to work with the new Perl.

Note that if you have a `PERL5LIB` setting, you don't need to alter the `@INC` value in your scripts. But if for example someone else (who doesn't have this setting in the shell) tries to execute your scripts, Perl will fail to find your locally installed modules. The best example is a crontab script that *might* use a different SHELL environment and therefore the `PERL5LIB` setting won't be available to it.

So the best approach is to have both the `PERL5LIB` environment variable and the explicit `@INC` extension code at the beginning of the scripts as described above.

3.9.3 The CPAN.pm Shell and Locally Installed Modules

As we saw in the section describing the usage of the `CPAN.pm` shell to install mod_perl, it saves a great deal of time. It does the job for us, even detecting the missing modules listed in prerequisites, fetching and installing them. So you might wonder whether you can use `CPAN.pm` to maintain your local repository as well.

When you start the `CPAN` interactive shell, it searches first for the user's private configuration file and then for the system wide one. When I'm logged as user `stas` the two files on my setup are:

```
/home/stas/.cpan/CPAN/MyConfig.pm
/usr/lib/perl5/5.00503/CPAN/Config.pm
```

If there is no `CPAN` shell configured on your system, when you start the shell for the first time it will ask you a dozen configuration questions and then create the *Config.pm* file for you.

If you've got it already system-wide configured, you should have a `/usr/lib/perl5/5.00503/CPAN/Config.pm`. If you have a different Perl version, alter the path to use your Perl's version number, when looking up the file. Create the directory (`mkdir -p` creates the whole path at once) where the local configuration file will go:

```
% mkdir -p /home/stas/.cpan/CPAN
```

Now copy the system wide configuration file to your local one.

```
% cp /usr/lib/perl5/5.00503/CPAN/Config.pm \
  /home/stas/.cpan/CPAN/MyConfig.pm
```

The only thing left is to change the base directory of *.cpan* in your local file to the one under your home directory. On my machine I replace `/usr/src/.cpan` (that's where my system's `.cpan` directory resides) with `/home/stas`. I use Perl of course!

```
% perl -pi -e 's|/usr/src|/home/stas|' \
/home/stas/.cpan/CPAN/MyConfig.pm
```

Now you have the local configuration file ready, you have to tell it what special parameters you need to pass when executing the `perl Makefile.PL` stage.

Open the file in your favorite editor and replace line:

```
'makepl_arg' => q[],
```

with:

```
'makepl_arg' => q[PREFIX=/home/stas],
```

Now you've finished the configuration. Assuming that you are logged in as the same user you have prepared the local installation for (*stas* in our example), start it like this:

```
% perl -MCPAN -e shell
```

From now on any module you try to install will be installed locally. If you need to install some system modules, just become the superuser and install them in the same way. When you are logged in as the superuser, the system-wide configuration file will be used instead of your local one.

If you have used more than just the `PREFIX` variable, modify *MyConfig.pm* to use them. For example if you have used these variables:

```
perl Makefile.PL PREFIX=/home/stas \
INSTALLPRIVLIB=/home/stas/lib/perl5 \
INSTALLSCRIPT=/home/stas/bin \
INSTALLSITELIB=/home/stas/lib/perl5/site_perl \
INSTALLBIN=/home/stas/bin \
INSTALLMAN1DIR=/home/stas/lib/perl5/man  \
INSTALLMAN3DIR=/home/stas/lib/perl5/man3
```

then replace `PREFIX=/home/stas` in the line:

```
'makepl_arg' => q[PREFIX=/home/stas],
```

with all the variables from above, so that the line becomes:

```
'makepl_arg' => q[PREFIX=/home/stas \
  INSTALLPRIVLIB=/home/stas/lib/perl5 \
  INSTALLSCRIPT=/home/stas/bin \
  INSTALLSITELIB=/home/stas/lib/perl5/site_perl \
  INSTALLBIN=/home/stas/bin \
  INSTALLMAN1DIR=/home/stas/lib/perl5/man  \
  INSTALLMAN3DIR=/home/stas/lib/perl5/man3],
```

If you arrange all the above parameters in one line, you can remove the backslashes (\).

3.9.4 Making a Local Apache Installation

Just like with Perl modules, if you don't have permissions to install files into the system area you have to install them locally under your home directory. It's almost the same as a plain installation, but you have to run the server listening to a port number greater than 1024 since only root processes can listen to lower numbered ports.

Another important issue you have to resolve is how to add startup and shutdown scripts to the directories used by the rest of the system services. You will have to ask your system administrator to assist you with this issue.

To install Apache locally, all you have to do is to tell `.configure` in the Apache source directory what target directories to use. If you are following the convention that I use, which makes your home directory look like the `/` (base) directory, the invocation parameters would be:

```
./configure --prefix=/home/stas
```

Apache will use the prefix for the rest of its target directories instead of the default `/usr/local/apache`. If you want to see what they are, before you proceed add the *--show-layout* option:

```
./configure --prefix=/home/stas --show-layout
```

You might want to put all the Apache files under `/home/stas/apache` following Apache's convention:

```
./configure --prefix=/home/stas/apache
```

If you want to modify some or all of the names of the automatically created directories:

```
./configure --prefix=/home/stas/apache \
  --sbindir=/home/stas/apache/sbin
  --sysconfdir=/home/stas/apache/etc
  --localstatedir=/home/stas/apache/var \
  --runtimedir=/home/stas/apache/var/run \
  --logfiledir=/home/stas/apache/var/logs \
  --proxycachedir=/home/stas/apache/var/proxy
```

That's all!

Also remember that you can start the script only under a user and group you belong to. You must set the `User` and `Group` directives in *httpd.conf* to appropriate values.

3.9.5 Manual Local mod_perl Enabled Apache Installation

Now when we have learned how to install local Apache and Perl modules separately, let's see how to install mod_perl enabled Apache in our home directory. It's almost as simple as doing each one separately, but there is one wrinkle you need to know about which I'll mention at the end of this section.

Let's say you have unpacked the Apache and mod_perl sources under */home/stas/src* and they look like this:

```
% ls /home/stas/src
/home/stas/src/apache_1.3.xx
/home/stas/src/mod_perl-1.xx
```

where *xx* are the version numbers as usual. You want the Perl modules from the mod_perl package to be installed under */home/stas/lib/perl5* and the Apache files to go under */home/stas/apache*. The following commands will do that for you:

```
% perl Makefile.PL \
PREFIX=/home/stas \
APACHE_PREFIX=/home/stas/apache \
APACHE_SRC=../apache_1.3.xx/src \
DO_HTTPD=1 \
USE_APACI=1 \
EVERYTHING=1
% make && make test && make install
% cd ../apache_1.3.xx
% make install
```

If you need some parameters to be passed to the `.configure` script, as we saw in the previous section use `APACI_ARGS`. For example:

```
APACI_ARGS='--sbindir=/home/stas/apache/sbin, \
  --sysconfdir=/home/stas/apache/etc, \
  --localstatedir=/home/stas/apache/var, \
  --runtimedir=/home/stas/apache/var/run, \
  --logfiledir=/home/stas/apache/var/logs, \
  --proxycachedir=/home/stas/apache/var/proxy'
```

Note that the above multiline splitting will work only with `(ba)?sh`, `t?csh` users will have to list all the parameters on a single line.

Basically the installation is complete. The only remaining problem is the `@INC` variable. This won't be correctly set if you rely on the `PERL5LIB` environment variable unless you set it explicitly in a startup file which is `require`'d before loading any other module that resides in your local repository. A much nicer approach is to use the `lib` pragma as we saw before, but in a slightly different way--we use it in the startup file and it affects all the code that will be executed under mod_perl handlers. For example:

```
PerlRequire /home/stas/apache/perl/startup.pl
```

where `startup.pl` starts with:

```
use lib qw(/home/stas/lib/perl5/5.00503/
           /home/stas/lib/perl5/site_perl/5.005);
```

Note that you can still use the hard-coded `@INC` modifications in the scripts themselves, but be aware that scripts modify `@INC` in `BEGIN` blocks and mod_perl executes the `BEGIN` blocks only when it performs script compilation. As a result, `@INC` will be reset to its original value after the scripts are compiled and the hard-coded settings will be forgotten. See the section '@INC and mod_perl' for more information.

The only place you can alter the "original" value is during the server configuration stage either in the startup file or by putting

```
PerlSetEnv Perl5LIB \
/home/stas/lib/perl5/5.00503/:/home/stas/lib/perl5/site_perl/5.005
```

in *httpd.conf*, but the latter setting will be ignored if you use the `PerlTaintcheck` setting, and I hope you do use it.

The rest of the mod_perl configuration and use is just the same as if you were installing mod_perl as superuser.

3.9.5.1 Resource Usage

Another important thing to keep in mind is the consumption of system resources. mod_perl is memory hungry. If you run a lot of mod_perl processes on a public, multiuser machine, most likely the system administrator of this machine will ask you to use less resources and may even shut down your mod_perl server and ask you to find another home for it. You have a few options:

- Reduce resources usage (see Preventing Your Processes from Growing).

- Ask your ISP's system administrator whether they can setup a dedicated machine for you, so that you will be able to install as much memory as you need. If you get a dedicated machine the chances are that you will want to have root access, so you may be able to manage the administration yourself. Then you should consider keeping on the list of the system administrator's responsibilities the following items: a reliable electricity supply and network link. And of course making sure that the important security patches get applied and the machine is configured to be secure. Finally having the machine physically protected, so no one will turn off the power or break it.

- Look for another ISP with lots of resources or one that supports mod_perl. You can find a list of these ISPs on this site.

3.9.6 Local mod_perl Enabled Apache Installation with CPAN.pm

Again, CPAN makes installation and upgrades simpler. You have seen how to install a mod_perl enabled server using `CPAN.pm`'s interactive shell. You have seen how to install Perl modules and Apache locally. Now all you have to do is to merge these techniques into a single "local mod_perl Enabled Apache Installation with CPAN.pm" technique.

Assuming that you have configured `CPAN.pm` to install Perl modules locally, the installation is very simple. Start the `CPAN.pm` shell, set the arguments to be passed to `perl Makefile.PL` (modify the example setting to suit your needs), and tell `CPAN.pm` to do the rest for you:

```
% perl -MCPAN -eshell
cpan> o conf makepl_arg 'DO_HTTPD=1 USE_APACI=1 EVERYTHING=1 \
      PREFIX=/home/stas APACHE_PREFIX=/home/stas/apache'
cpan> install mod_perl
```

When you use `CPAN.pm` for local installations, after the mod_perl installation is complete you must make sure that the value of `makepl_arg` is restored to its original value.

The simplest way to do this is to quit the interactive shell by typing *quit* and reenter it. But if you insist here is how to make it work without quitting the shell. You really want to skip this :)

If you want to continue working with `CPAN` *without* quitting the shell, you must:

1. **remember the value of `makepl_arg`**
2. **change it to suit your new installation**
3. **build and install mod_perl**
4. **restore it after completing mod_perl installation**

this is quite a cumbersome task as of this writing, but I believe that `CPAN.pm` will eventually be improved to handle this more easily.

So if you are still with me, start the shell as usual:

```
% perl -MCPAN -eshell
```

First, read the value of the `makepl_arg`:

```
cpan> o conf makepl_arg

PREFIX=/home/stas
```

It will be something like `PREFIX=/home/stas` if you configured `CPAN.pm` to install modules locally. Save this value:

```
cpan> o conf makepl_arg.save PREFIX=/home/stas
```

Second, set a new value, to be used by the mod_perl installation process. (You can add parameters to this line, or remove them, according to your needs.)

```
cpan> o conf makepl_arg 'DO_HTTPD=1 USE_APACI=1 EVERYTHING=1 \
      PREFIX=/home/stas APACHE_PREFIX=/home/stas/apache'
```

Third, let `CPAN.pm` build and install mod_perl for you:

```
cpan> install mod_perl
```

Fourth, reset the original value to `makepl_arg`. We do this by printing the value of the saved variable and assigning it to `makepl_arg`.

```
cpan> o conf makepl_arg.save

PREFIX=/home/stas

cpan> o conf makepl_arg PREFIX=/home/stas
```

Not so neat, but a working solution. You could have written the value on a piece of paper instead of saving it to `makepl_arg.save`, but you are more likely to make a mistake that way.

3.10 Automating installation

- **Apache Builder**

  James G Smith wrote an Apache Builder, that can install a combination of Apache, mod_perl, and mod_ssl -- it also has limited support for including mod_php in the mix. The builder is available from James' CPAN directory: $CPAN/authors/id/J/JS/JSMITH/ in the package *build-apache-xx.tar.gz*.

- **Aphid Apache Installer**

  Aphid provides a facility for bootstrapping SSL-enabled Apache web servers (mod_ssl) with an embedded Perl interpreter (mod_perl). Source is downloaded from the Internet, compiled, and the resulting system is installed in the directory you specify.

  http://sourceforge.net/projects/aphid/

3.11 How can I tell whether mod_perl is running?

There are a few ways. In older versions of apache (< 1.3.6 ?) you could check that by running `httpd -v`, but it no longer works. Now you should use `httpd -l`. Please note that it is not enough to have it installed, you have to configure it for mod_perl and restart the server too.

3.11.1 Checking the error_log

When starting the server, just check the `error_log` file for the following message:

```
[Thu Dec  3 17:27:52 1998] [notice] Apache/1.3.1 (Unix) mod_perl/1.15 configured
                                    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  -- resuming normal operations
```

3.11.2 Testing by viewing /perl-status

Assuming that you have configured the <Location /perl-status> section in the server configuration file fetch: http://www.example.com/perl-status using your favorite Mozilla browser :-)

You should see something like this:

```
Embedded Perl version 5.00503 for Apache/1.3.9 (Unix) mod_perl/1.21
process 50880, running since Mon Dec 6 14:31:45 1999
```

3.11.3 Testing via telnet

Knowing the port you have configured apache to listen on, you can use `telnet` to talk directly to it.

Assuming that your mod_perl enabled server listens to port 8080, telnet to your server at port 8080, and type `HEAD / HTTP/1.0` then press the `ENTER` key TWICE:

```
% telnet localhost 8080<ENTER>
HEAD / HTTP/1.0<ENTER><ENTER>
```

You should see a response like this:

```
HTTP/1.1 200 OK
Date: Mon, 06 Dec 1999 12:27:52 GMT
Server: Apache/1.3.9 (Unix) mod_perl/1.21
Connection: close
Content-Type: text/html

Connection closed.
```

The line

```
Server: Apache/1.3.9 (Unix) mod_perl/1.21
```

confirms that you have mod_perl installed and its version is `1.21`.

However, just because you have got mod_perl linked in there, that does not mean that you have configured your server to handle Perl scripts with mod_perl. You will find configuration assistance at ModPerlConfiguration

3.11.4 Testing via a CGI script

Another method is to invoke a CGI script which dumps the server's environment.

I assume that you have configured the server so that scripts running under location */perl/* are handled by the `Apache::Registry` handler and that you have the `PerlSendHeader` directive set to `On`.

Copy and paste the script below (no need for a shebang line!). Let's say you name it *test.pl*, save it at the root of the CGI scripts and CGI root is mapped directly to the */perl* location of your server.

```
print "Content-type: text/plain\r\n\r\n";
print "Server's environment\n";
foreach ( keys %ENV ) {
    print "$_\t$ENV{$_}\n";
}
```

Make it readable and executable by server (you may need to tune these permissions on a public host):

```
% chmod a+rx test.pl
```

Now fetch the URL `http://www.example.com:8080/perl/test.pl` (replace 8080 with the port your mod_perl enabled server is listening to). You should see something like this (the output has been edited):

```
SERVER_SOFTWARE Apache/1.3.10-dev (Unix) mod_perl/1.21_01-dev
GATEWAY_INTERFACE       CGI-Perl/1.1
DOCUMENT_ROOT   /home/httpd/docs
REMOTE_ADDR     127.0.0.1
[more environment variables snipped]
MOD_PERL        mod_perl/1.21_01-dev
[more environment variables snipped]
```

If you see the that the value of `GATEWAY_INTERFACE` is `CGI-Perl/1.1` everything is OK.

If there is an error you might have to add a shebang line `#!/usr/bin/perl` as a first line of the CGI script and then try it again. If you see:

```
GATEWAY_INTERFACE       CGI/1.1
```

it means that you have configured this location to run under mod_cgi and not mod_perl.

Also note that there is a `MOD_PERL` environment variable if you run under a mod_perl handler, it's set to the *mod_perl/x.xx* string, where *x.xx* is the version number of mod_perl.

Based on this difference you can write code like this:

```
BEGIN {
    # perl5.004 or better is a must under mod_perl
  require 5.004 if $ENV{MOD_PERL};
}
```

You might wonder why in the world you would need to know what handler you are running under. Well, for example you will want to use `Apache::exit()` and not `CORE::exit()` in your modules, but if you think that your script might be used in both environments (mod_cgi and mod_perl) you will have to override the `exit()` subroutine and to make decision what method to use at the runtime.

Note that if you run scripts under the `Apache::Registry` handler, it takes care of overriding the `exit()` call for you, so it's not an issue. For reasons and implementations see: Terminating requests and processes, exit() function and also Writing Mod Perl scripts and Porting plain CGIs to it.

3.11.5 Testing via lwp-request

Yet another one. Why do I show all these approaches? While here they serve a very simple purpose, they can be helpful in other situations.

Assuming you have the `libwww-perl` (`LWP`) package installed (you will need it installed in order to pass mod_perl's `make test` anyway):

```
% lwp-request -e -d http://www.example.com
```

Will show you all the headers. The `-d` option disables printing the response content.

```
% lwp-request -e -d http://www.example.com | egrep '^Server:'
```

To see the server version only.

Specify the port number if your server is listening to a port other than port 80. For example: `http://www.example.com:8080`.

This technique works only if `ServerTokens` directive is set to `Full` or not specified in *httpd.conf*. That's because this directive controls whether the components information is displayed or not.

3.12 General Notes

3.12.1 Is it possible to run mod_perl enabled Apache as suExec?

The answer is **No**. The reason is that you can't *"suid"* a part of a process. mod_perl lives inside the Apache process, so its UID and GID are the same as the Apache process.

You have to use mod_cgi if you need this functionality.

Another solution is to use a crontab to call some script that will check whether there is something to do and will execute it. The mod_perl script will be able to create and update this todo list.

3.12.2 Should I Rebuild mod_perl if I have Upgraded Perl?

Yes, you should. You have to rebuild the mod_perl enabled server since it has a hard-coded `@INC` variable. This points to the old Perl and it is probably linked to an old `libperl` library. If for some reason you need to keep the old Perl version around you can modify `@INC` in the startup script, but it is better to build afresh to save you getting into a mess.

3.12.3 Perl installation requirements

Make sure you have Perl installed! The latest stable version if possible. Minimum perl 5.004! If you don't have it, install it. Follow the instructions in the distribution's `INSTALL` file.

During the configuration stage (while running `./Configure`), to be able to dynamically load Perl module extensions, make sure you answer `YES` to the question:

```
Do you wish to use dynamic loading? [y]
```

3.12.4 mod_auth_dbm nuances

If you are a `mod_auth_dbm` or `mod_auth_db` user you may need to edit Perl's `Config` module. When Perl is configured it attempts to find libraries for ndbm, gdbm, db, etc., for the DB*_File modules. By default, these libraries are linked with Perl and remembered by the `Config` module. When mod_perl is configured with apache, the `ExtUtils::Embed` module requires these libraries to be linked with httpd so Perl extensions will work under mod_perl. However, the order in which these libraries are stored in **Config.pm** may confuse `mod_auth_db*`. If `mod_auth_db*` does not work with mod_perl, take a look at the order with the following command:

```
% perl -V:libs
```

Here's an example:

```
libs='-lnet -lnsl_s -lgdbm -lndbm -ldb -ldld -lm -lc -lndir -lcrypt';
```

If `-lgdbm` or `-ldb` is before `-lndbm` (as it is in the example) edit *Config.pm* and move `-lgdbm` and `-ldb` to the end of the list. Here's how to find *Config.pm*:

```
% perl -MConfig -e 'print "$Config{archlibexp}/Config.pm\n"'
```

Under Solaris, another solution for building Apache/mod_perl+mod_auth_dbm is to remove the DBM and NDBM "emulation" from *libgdbm.a*. It seems that Solaris already provides its own DBM and NDBM, and in our installation we found there's no reason to build GDBM with them.

In our Makefile for GDBM, we changed

```
OBJS = $(DBM_OF) $(NDBM_OF) $(GDBM_OF)
```

to

```
OBJS = $(GDBM_OF)
```

Rebuild libgdbm before Apache/mod_perl.

3.12.5 Stripping Apache to make it almost a Perl-server

Since most of the functionality that various apache mod_* modules provide is implemented in the `Apache::{*}` Perl modules, it was reported that one can build an Apache server with mod_perl only. If you can reduce the requirements to whatever mod_perl can handle, you can eliminate almost every other module. Then basically you will have a Perl-server, with C code to handle the tricky HTTP bits. The only module you will need to leave in is `mod_actions`.

3.12.6 Saving the config.status Files with mod_perl, php, ssl and Other Components

Typically, when building the bloated Apache that sits behind Squid or whatever, you need mod_perl, php, mod_ssl and the rest. As you install each they typically overwrite each other's `config.status` files. Save them after each step, so you will be able to reuse them later.

3.12.7 What Compiler Should Be Used to Build mod_perl?

All Perl modules that use C extensions must be compiled using the same compiler that your copy of Perl was built with and the same compile options.

When you run `perl Makefile.PL`, a *Makefile* is created. This *Makefile* includes the same compilation options that were used to build Perl itself. They are stored in the *Config.pm* module and can be displayed with the `Perl -V` command. All these options are re-applied when compiling Perl modules.

If you use a different compiler to build Perl extensions, chances are that the options that a different compiler uses won't be the same, or they might be interpreted in a completely different way. So the code either won't compile or it will dump core when run or maybe it will behave in most unexpected ways.

Since mod_perl uses Perl, Apache and third party modules, and they all work together, it's essential to use the same compiler while building each of the components.

You shouldn't worry about this when compiling Perl modules since Perl will choose what's right automatically. Unless you override things. If you do that, you are on your own...

Similarly, if you compile a non-Perl component separately, you should make sure to use both the same compiler and the same options used to build Perl.

3.12.8 Unescaping error_log

Starting from 1.3.30, the Apache logging API escapes everything that goes to *error_log*, therefore if you're annoyed by this feature during the development phase (as your error messages will be all messed up) you can disable the escaping during the Apache build time:

```
% CFLAGS="-DAP_UNSAFE_ERROR_LOG_UNESCAPED" ./configure ...
```

Or if you build a static perl

```
% perl Makefile.PL ... PERL_EXTRA_CFLAGS=-DAP_UNSAFE_ERROR_LOG_UNESCAPED
```

Do **not** use that CFLAGS in production unless you know what you are doing.

3.13 OS Related Notes

- Gary Shea <shea (at) xmission.com> discovered a nasty BSDI bug (seen in versions 2.1 and 3.0) related to dynamic loading and found two workarounds:

  It turns out that they use `argv[0]` to determine where to find the link tables at run-time, so if a program either changes `argv[0]`, or does a chdir() (like Apache!) it can easily confuse the dynamic loader. The short-term solutions to the problem are simple. Either of the following will work:

  1) Call httpd with a full path, e.g. /opt/www/bin/httpd

  2) Put the httpd you wish to run in a directory in your PATH *before* any other directory containing a version of httpd, then call it as 'httpd'. Don't use a relative path!

3.14 Pros and Cons of Building mod_perl as DSO

On modern Unix derivatives there is a nifty mechanism usually called dynamic linking/loading of Dynamic Shared Objects (DSO), which provides a way to build a piece of program code in a special format for loading in at run-time into the address space of an executable program.

As of Apache 1.3, the configuration system supports two optional features for taking advantage of the modular DSO approach: compilation of the Apache core program into a DSO library for shared usage and compilation of the Apache modules into DSO files for explicit loading at run-time.

Should you use this method? Read the pros and cons and decide for yourself.

Pros:

- The server package is more flexible at run-time because the actual server process can be assembled at run-time via `LoadModule` *httpd.conf* configuration commands instead of *Configuration* `AddModule` commands at build-time. For instance this way one is able to run different server instances (standard & SSL version, with and without mod_perl) with only one Apache installation.

- The server package can be easily extended with third-party modules even after installation. This is at least a great benefit for vendor package maintainers who can create an Apache core package and additional packages containing extensions like PHP4, mod_perl, mod_fastcgi, etc.

- Easier Apache module prototyping because with the DSO/apxs pair you can both work outside the Apache source tree and only need an apxs -i command followed by an apachectl restart to bring a new version of your currently developed module into the running Apache server.

Cons:

- The DSO mechanism cannot be used on every platform because not all operating systems support dynamic loading of code into the address space of a program.

- The server starts up approximately 20% slower because of the symbol resolving overhead the Unix loader now has to do.

- The server runs approximately 5% slower on some platforms because position independent code (PIC) sometimes needs complicated assembler tricks for relative addressing which are not necessarily as fast as absolute addressing.

- Because DSO modules cannot be linked against other DSO-based libraries (ld -lfoo) on all platforms (for instance a.out-based platforms usually don’t provide this functionality while ELF-based platforms do) you cannot use the DSO mechanism for all types of modules. Or in other words, modules compiled as DSO files are restricted to only use symbols from the Apache core, from the C library (libc) and all other dynamic or static libraries used by the Apache core, or from static library archives (libfoo.a) containing position independent code. The only way you can use other code is to either make sure the Apache core itself already contains a reference to it, loading the code yourself via dlopen() or enabling the SHARED_CHAIN rule while building Apache when your platform supports linking DSO files against DSO libraries.

- Under some platforms (many SVR4 systems) there is no way to force the linker to export all global symbols for use in DSO’s when linking the Apache httpd executable program. But without the visibility of the Apache core symbols no standard Apache module could be used as a DSO. The only workaround here is to use the SHARED_CORE feature because this way the global symbols are forced to be exported. As a consequence the Apache src/Configure script automatically enforces SHARED_CORE on these platforms when DSO features are used in the Configuration file or on the configure command line.

3.15 Maintainers

Maintainer is the person(s) you should contact with updates, corrections and patches.

- Stas Bekman [http://stason.org/]

3.16 Authors

- Stas Bekman [http://stason.org/]

Only the major authors are listed above. For contributors see the Changes file.

4 mod_perl Configuration

4.1 Description

This section documents the various configuration options available for Apache and mod_perl, as well as the Perl startup files, and more esoteric possibilites such as configuring Apache with Perl.

4.2 Server Configuration

The next step after building and installing your new mod_perl enabled Apache server is to configure the server. There are two separate parts to configure: Apache and mod_perl. Each has its own set of directives.

To configure your mod_perl enabled Apache server, the only file that you should need to edit is *httpd.conf*. By default, *httpd.conf* is put into the *conf* directory under the server root directory. The default server root is */usr/local/apache/* on many UNIX platforms, but within reason it can be any directory you choose. If you are new to Apache and mod_perl, you will probably find it helpful to keep to the directory layouts we use in this Guide if you can.

Apache versions 1.3.4 and later are distributed with the configuration directives in a single file -- *httpd.conf*. This Guide uses the same approach in its examples. Prior to version 1.3.4, the default Apache installation used three configuration files -- *httpd.conf, srm.conf,* and *access.conf*. If you wish you can still use all three files, by setting the AccessConfig and ResourceConfig directives in *httpd.conf*. You will also see later on that we use other files, for example *perl.conf* and *startup.pl*. This is just for our convenience, you could still do everything in *httpd.conf* if you wished.

4.3 Apache Configuration

Apache configuration can be confusing. To minimize the number of things that can go wrong, it can be a good idea first to configure Apache itself without mod_perl. This will give you the confidence that it works and maybe that you have some idea how to configure it.

There is a warning in the *httpd.conf* distributed with Apache about simply editing *httpd.conf* and running the server, without understanding all the implications. This is another warning. Modifying the configuration file and adding new directives can introduce security problems, and have performance implications.

The Apache distribution comes with an extensive configuration manual, and in addition each section of the distributed configuration file includes helpful comments explaining how every directive should be configured and what the defaults values are.

If you haven't moved Apache's directories around, the installation program will have configured everything for you. You can just start the server and test it. To start the server use the `apachectl` utility which comes bundled with the Apache distribution. It resides in the same directory as `httpd`, the Apache server itself. Execute:

```
/usr/local/apache/bin/apachectl start
```

Now you can test the server, for example by accessing http://localhost from a browser running on the same host.

4.3.1 Configuration Directives

For a basic setup there are just a few things to configure. If you have moved any directories you have to update them in *httpd.conf*. There are many of them, here are just a couple of examples:

```
ServerRoot   "/usr/local/apache"
DocumentRoot "/home/httpd/docs"
```

If you want to run it on a port other than port 80 edit the `Port` directive:

```
Port 8080
```

You might want to change the user and group names the server will run under. Note that if started as the *root* user (which is generally the case), the parent process will continue to run as *root*, but its children will run as the user and group you have specified. For example:

```
User httpd
Group httpd
```

There are many other directives that you might need to configure as well. In addition to directives which take a single value there are whole sections of the configuration (such as the `<Directory>` and `<Location>` sections) which apply only to certain areas of your Web space. As mentioned earlier you will find them all in *httpd.conf*.

4.3.2 .htaccess files

If there is a file with the name *.htaccess* in any directory, Apache scans it for further configuration directives which it then applies only to that directory (and its subdirectories). The name *.htaccess* is confusing because it can contain any configuration directives, not just those related to access to resources. You will not be surprised to find that a configuration directive can change the names of the files used in this way.

Note that if there is a

```
<Directory />
  AllowOverride None
</Directory>
```

directive in *httpd.conf*, Apache will not try to look for *.htaccess* at all.

4.3.3 <Directory>, <Location> and <Files> Sections

I'll explain just the basics of the `<Directory>`, `<Location>` and `<Files>` sections. Remember that there is more to know and the rest of the information is available in the Apache documentation. The information I'll present here is just what is important for understanding the mod_perl configuration sections.

Apache considers directories and files on your machine all to be resources. For each resource you can determine a particular behaviour which will apply to every request for information from that particular resource.

Obviously the directives in `<Directory>` sections apply to specific directories on your host machine, and those in `<Files>` sections apply only to specific files (actually groups of files with names which have something in common). In addition to these sections, Apache has the concept of a `<Location>`, which is also just a resource. `<Location>` sections apply to specific URIs. Locations are based at the document root, directories are based at the filesystem root. For example, if you have the default server directory layout where the server root is */usr/local/apache* and the document root is */usr/local/apache/htdocs* then static files in the directory */usr/local/apache/htdocs/pub* are in the location */pub*.

It is up to you to decide which directories on your host machine are mapped to which locations. You should be careful how you do it, because the security of your server may be at stake.

Locations do not necessarily have to refer to existing physical directories, but may refer to virtual resources which the server creates for the duration of a single browser request. As you will see, this is often the case for a mod_perl server.

When a browser asks for a resource from your server, Apache determines from its configuration whether or not to serve the request, whether to pass the request to another server, what (if any) authorization is required for access to the resource, and how to reply. For any given resource, the various sections in your configuration may provide conflicting information. For example you may have a `<Directory>` section which tells Apache that authorization is required for access to the resource but you may have a `<Files>` section which says that it is not. It is not always obvious which directive takes precedence in these cases. This can be a trap for the unwary.

- **<Directory directoryPath> ... </Directory>**

  Can appear in server and virtual host configurations.

  `<Directory>` and `</Directory>` are used to enclose a group of directives which will apply only to the named directory and sub-directories of that directory. Any directive which is allowed in a directory context (see the Apache documentation) may be used.

  The path given in the `<Directory>` directive is either the full path to a directory, or a wild-card string. In a wild-card string, `?` matches any single character, `*` matches any sequence of characters, and `[]` matches character ranges. (This is similar to the shell's file globs.) None of the wildcards will match a `/` character. For example:

  ```
  <Directory /home/httpd/docs>
    Options Indexes
  </Directory>
  ```

  If you want to use a regular expression to match then you should use the syntax `<Directory-Match regex>` … `</DirectoryMatch>`.

If multiple (non-regular expression) directory sections match the directory (or its parents) containing a document, then the directives are applied in the order of shortest match first, interspersed with the directives from any *.htaccess* files. For example, with

```
<Directory />
  AllowOverride None
</Directory>

<Directory /home/httpd/docs/*>
  AllowOverride FileInfo
</Directory>
```

for access to the document */home/httpd/docs/index.html* the steps are:

- **Apply directive `AllowOverride None` (disabling *.htaccess* files).**
- **Apply directive `AllowOverride FileInfo` for directory */home/httpd/docs/* (which now enables *.htaccess* in */home/httpd/docs/* and its sub-directories).**
- **Apply any `FileInfo` directives in */home/httpd/docs/.htaccess*.**

- **`<Files filename> ... </Files>`**

  Can appear in server and virtual host configurations, and *.htaccess* files as well.

  The `<Files>` directive provides for access control by filename. It is comparable to the `<Directory>` and `<Location>` directives. It should be closed with the `</Files>` directive. The directives given within this section will be applied to any object with a basename (last component of filename) matching the specified filename.

  `<Files>` sections are processed in the order they appear in the configuration file, after the `<Directory>` sections and *.htaccess* files are read, but before `<Location>` sections. Note that `<Files>` can be nested inside `<Directory>` sections to restrict the portion of the filesystem they apply to. `<Files>` cannot be nested inside `<Location>` sections however.

  The filename argument should include a filename, or a wild-card string, where `?` matches any single character, and `*` matches any sequence of characters. Extended regular expressions can also be used, simply place a tilde character ~ between the directive and the regular expression. The regular expression should be in quotes. The dollar symbol refers to the end of the string. The pipe character indicates alternatives. Special characters in extended regular expressions must escaped with a backslash. For example:

  ```
  <Files ~ "\.(gif|jpe?g|png)$">
  ```

  would match most common Internet graphics formats. Alternatively you can use the `<FilesMatch regex>` ... `</FilesMatch>` syntax.

- **<Location URL> ... </Location>**

  Can appear in server and virtual host configurations.

The <Location> directive provides for access control by URL. It is similar to the <Directory> directive, and starts a section which is terminated with the </Location> directive.

<Location> sections are processed in the order they appear in the configuration file, after the <Directory> sections, *.htaccess* files and <Files> sections are read.

The <Location> section is the directive that is used most often with mod_perl.

URLs *do not* have to refer to real directories or files within the filesystem at all, <Location> operates completely outside the filesystem. Indeed it may sometimes be wise to ensure that <Location>s do not match real paths to avoid confusion.

The URL may use wildcards. In a wild-card string, ? matches any single character, and * matches any sequences of characters, [] groups characters to match. For regular expression matches use the <LocationMatch regex> ... </LocationMatch> syntax.

The <Location> functionality is especially useful when combined with the SetHandler directive. For example to enable status requests, but allow them only from browsers at *example.com*, you might use:

```
<Location /status>
  SetHandler server-status
  order deny,allow
  deny from all
  allow from .example.com
</Location>
```

4.3.4 How Directory, Location and Files Sections are Merged

When configuring the server, it's important to understand the order in which the rules of each section apply to requests. The order of merging is:

1. **<Directory> (except regular expressions) and *.htaccess* are processed simultaneously, with *.htaccess* overriding <Directory>**
2. **<DirectoryMatch>, and <Directory> with regular expressions**
3. **<Files> and <FilesMatch> are processed simultaneously**
4. **<Location> and <LocationMatch> are processed simultaneously**

Apart from <Directory>, each group is processed in the order that it appears in the configuration files. <Directory> (group 1 above) is processed in the order shortest directory component to longest. If multiple <Directory> sections apply to the same directory then they are processed in the configuration file order.

Sections inside <VirtualHost> sections are applied as if you were running several independent servers. The directives inside <VirtualHost> sections do not interact with each other. They are applied after first processing any sections outside the virtual host definition. This allows virtual host configurations to override the main server configuration.

Later sections override earlier ones.

4.3.5 Sub-Grouping of <Location>, <Directory> and <Files> Sections

Let's say that you want all files, except for a few of the files in a specific directory and below, to be handled in the same way. For example if you want all the files in */home/http/docs* to be served as plain files, but any files with ending *.html* and *.txt* to be processed by the content handler of your `Apache::MyFilter` module.

```
<Directory /home/httpd/docs>
  <FilesMatch "\.(html|txt)$">
    SetHandler perl-script
    PerlHandler Apache::MyFilter
  </FilesMatch>
</Directory>
```

Thus it is possible to embed sections inside sections to create subgroups which have their own distinct behavior. Alternatively you could use a `<Files>` section inside an *.htaccess* file.

Note that you can't put `<Files>` or `<FilesMatch>` sections inside a `<Location>` section, but you can put them inside a `<Directory>` section.

4.3.6 Options Directive

Normally, if multiple `Options` directives apply to a directory, then the most specific one is taken complete; the options are not merged.

However if all the options on the `Options` directive are preceded by a `+` or `-` symbol, the options are merged. Any options preceded by `+` are added to the options currently in force, and any options preceded by `-` are removed.

For example, without any `+` and `-` symbols:

```
<Directory /home/httpd/docs>
  Options Indexes FollowSymLinks
</Directory>
<Directory /home/httpd/docs/shtml>
  Options Includes
</Directory>
```

then only `Includes` will be set for the */home/httpd/docs/shtml* directory. However if the second `Options` directive uses the `+` and `-` symbols:

```
<Directory /home/httpd/docs>
  Options Indexes FollowSymLinks
</Directory>
<Directory /home/httpd/docs/shtml>
  Options +Includes -Indexes
</Directory>
```

then the options `FollowSymLinks` and `Includes` are set for the */home/httpd/docs/shtml* directory.

4.4 mod_perl Configuration

When you have tested that the Apache server works on your machine, it's time to configure mod_perl. Some of the configuration directives are already familiar to you, but mod_perl introduces a few new ones.

It can be a good idea to keep all the mod_perl related configuration at the end of the configuration file, after the native Apache configuration directives.

To ease maintenance and to simplify multiple server installations, the Apache/mod_perl configuration system allows you several alternative ways to keep your configuration directives in separate places. The `Include` directive in *httpd.conf* allow you to include the contents of other files, just as if the information were all contained in *httpd.conf*. This is a feature of Apache itself. For example if you want all your mod_perl configuration to be placed in a separate file *mod_perl.conf* you can do that by adding to *httpd.conf* this directive:

```
Include conf/mod_perl.conf
```

If you want to include this configuration conditionally, depending on whether your apache has been compiled with mod_perl, you can use the `IfModule` directive:

```
<IfModule mod_perl.c>
    Include conf/mod_perl.conf
</IfModule>
```

mod_perl adds two further directives: `<Perl>` sections allow you to execute Perl code from within any configuration file at server startup time, and as you will see later, a file containing any Perl program can be executed (also at server startup time) simply by mentioning its name in a `PerlRequire` or `PerlModule` directive.

4.4.1 Alias Configurations

The `ScriptAlias` and `Alias` directives provide a mapping of a URI to a file system directory. The directive:

```
Alias /foo /home/httpd/foo
```

will map all requests starting with */foo* onto the files starting with */home/httpd/foo/*. So when Apache gets a request http://www.example.com/foo/test.pl the server will map this into the file *test.pl* in the directory */home/httpd/foo/*.

In addition `ScriptAlias` assigns all the requests that match the URI (i.e. */cgi-bin*) to be executed under mod_cgi.

```
ScriptAlias /cgi-bin /home/httpd/cgi-bin
```

is actually the same as:

```
Alias /cgi-bin /home/httpd/cgi-bin
<Location /cgi-bin>
  SetHandler cgi-script
  Options +ExecCGI
</Location>
```

where latter directive invokes mod_cgi. You shouldn't use the `ScriptAlias` directive unless you want the request to be processed under mod_cgi. Therefore when you configure mod_perl sections use `Alias` instead.

Under mod_perl the `Alias` directive will be followed by two further directives. The first is the SetHandler perl-script directive, which tells Apache to invoke mod_perl to run the script. The second directive (for example `PerlHandler`) tells mod_perl which handler (Perl module) the script should be run under, and hence for which phase of the request. Refer to the section Perl*Handlers for more information about handlers for the various request phases.

When you have decided which methods to use to run your scripts and where you will keep them, you can add the configuration directive(s) to *httpd.conf*. They will look like those below, but they will of course reflect the locations of your scripts in your file-system and the decisions you have made about how to run the scripts:

```
ScriptAlias /cgi-bin/ /home/httpd/cgi-bin/
Alias       /perl/    /home/httpd/perl/
```

In the examples above all the requests issued for URIs starting with */cgi-bin* will be served from the directory */home/httpd/cgi-bin/*, and starting with */perl* from the directory */home/httpd/perl/*.

4.4.1.1 Running CGI, PerlRun, and Registry Scripts Located in the Same Directory

```
# Typical for plain cgi scripts:
ScriptAlias /cgi-bin/  /home/httpd/perl/

# Typical for Apache::Registry scripts:
Alias       /perl/     /home/httpd/perl/

# Typical for Apache::PerlRun scripts:
Alias       /cgi-perl/ /home/httpd/perl/
```

In the examples above we have mapped the three different URIs (*http://www.example.com/perl/test.pl http://www.example.com/cgi-bin/test.pl* and *http://www.example.com/cgi-perl/test.pl*) all to the same file */home/httpd/perl/test.pl*. This means that we can have all our CGI scripts located at the same place in the file-system, and call the script in any of three ways simply by changing one component of the URI (*cgi-bin*|*perl*|*cgi-perl*).

This technique makes it easy to migrate your scripts to mod_perl. If your script does not seem to be working while running under mod_perl, then in most cases you can easily call the script in straight mod_cgi mode or under `Apache::PerlRun` without making any script changes. Simply change the URL you use to invoke it.

Although in the configuration above we have configured all three *Aliases* to point to the same directory within our file system, you can of course have them point to different directories if you prefer.

You should remember that it is undesirable to run scripts in plain mod_cgi mode from a mod_perl-enabled server--the resource consumption is too high. It is better to run these on a plain Apache server. See Standalone mod_perl Enabled Apache Server.

4.4.2 <Location> Configuration

The `<Location>` section assigns a number of rules which the server should follow when the request's URI matches the *Location*. Just as it is the widely accepted convention to use */cgi-bin* for your mod_cgi scripts, it is conventional to use */perl* as the base URI of the perl scripts which you are running under mod_perl. Let's review the following very widely used `<Location>` section:

```
Alias /perl/ /home/httpd/perl/
PerlModule Apache::Registry
<Location /perl>
  SetHandler perl-script
  PerlHandler Apache::Registry
  Options ExecCGI
  allow from all
  PerlSendHeader On
</Location>
```

This configuration causes all requests for URIs starting with */perl* to be handled by the mod_perl Apache module with the handler from the `Apache::Registry` Perl module. Let's review the directives inside the `<Location>` section in the example:

```
<Location /perl>
```

Remember the `Alias` from the above section? We use the same `Alias` here; if you were to use a `<Location>` that does not have the same `Alias`, the server would fail to locate the script in the file system. You need the `Alias` setting only if the code that should be executed is located in the file. So `Alias` just provides the URI to filepath translation rule.

Sometimes there is no script to be executed. Instead there is some module whose method is being executed, similar to */perl-status*, where the code is stored in an Apache module. In such cases we don't need `Alias` settings for those `<Location>`s.

```
SetHandler perl-script
```

This assigns the mod_perl Apache module to handle the content generation phase.

```
PerlHandler Apache::Registry
```

Here we tell Apache to use the `Apache::Registry` Perl module for the actual content generation.

```
Options ExecCGI
```

The `Options` directive accepts various parameters (options), one of which is `ExecCGI`. This tells the server that the file is a program and should be executed, instead of just being displayed like a static file (like HTML file). If you omit this option then the script will either be rendered as plain text or else it will trigger a *Save-As* dialog, depending on the client's configuration.

```
allow from all
```

This directive is used to set access control based on domain. The above settings allow clients from any domain to run the script.

```
PerlSendHeader On
```

`PerlSendHeader On` tells the server to send an HTTP headers to the browser on every script invocation. You will want to turn this off for nph (non-parsed-headers) scripts.

The `PerlSendHeader On` setting invokes the Apache's `ap_send_http_header()` method after parsing the headers generated by the script. It is only meant for emulation of mod_cgi behavior with regard to headers.

To send the HTTP headers it's always better either to use the `$r->send_http_header` method using the Apache Perl API or to use the `$q->header` method from the `CGI.pm` module.

```
</Location>
```

Closes the `<Location>` section definition.

Note that sometimes you will have to preload the module before using it in the `<Location>` section. In the case of `Apache::Registry` the configuration will look like this:

```
PerlModule Apache::Registry
<Location /perl>
  SetHandler perl-script
  PerlHandler Apache::Registry
  Options ExecCGI
  allow from all
  PerlSendHeader On
</Location>
```

`PerlModule` is equivalent to Perl's native `use()` function call.

No changes are required to the */cgi-bin* location (mod_cgi), since it has nothing to do with mod_perl.

Here is another very similar example, this time using `Apache::PerlRun` (For more information see Apache::PerlRun):

```
<Location /cgi-perl>
  SetHandler perl-script
  PerlHandler Apache::PerlRun
  Options ExecCGI
  allow from all
  PerlSendHeader On
</Location>
```

The only difference from the `Apache::Registry` configuration is the argument of the `PerlHandler` directive, where `Apache::Registry` has been replaced with `Apache::PerlRun`.

4.4.3 Overriding <Location> Setting in "Sub-Location"

So if you have:

```
<Location /foo>
  SetHandler perl-script
  PerlHandler My::Module
 </Location>
```

If you want to remove a mod_perl handler setting from a location beneath a location where the handler was set (i.e. */foo/bar*), all you have to do is to reset it, like this:

```
<Location /foo/bar>
  SetHandler default-handler
</Location>
```

Now, all the requests starting with */foo/bar* would be served by Apache's default handler.

4.4.4 PerlModule and PerlRequire Directives

As we saw earlier, a module should be loaded before it is used. `PerlModule` and `PerlRequire` are the two mod_perl directives which are used to load modules and code. They are almost equivalent to Perl's `use()` and `require()` functions respectively and called from the Apache configuration file. You can pass one or more module names as arguments to `PerlModule`:

```
PerlModule Apache::DBI CGI DBD::Mysql
```

Generally the modules are preloaded from the startup script, which is usually called *startup.pl*. This is a file containing plain Perl code which is executed through the `PerlRequire` directive. For example:

```
PerlRequire  /home/httpd/perl/lib/startup.pl
```

A `PerlRequire` file name can be absolute or relative to `ServerRoot` or a path in `@INC`.

As with any file with Perl code that gets `use()`'d or `require()`'d, it must return a *true* value. To ensure that this happens don't forget to add `1;` at the end of *startup.pl*.

Notice that unless mod_perl is compiled as DSO and unless `PerlFreshRestart` is set to `On`, one needs to fully stop and start Apache for any changes to take affect, if the files and modules have been modified.

*4.4.5 Perl*Handlers*

As you probably know Apache traverses a loop for each HTTP request it receives.

After you have compiled and installed mod_perl, your Apache mod_perl configuration directives tell Apache to invoke the module mod_perl as the handler for some request which it receives. Although it could in fact handle all the phases of the request loop, usually it does not. You tell mod_perl which phases it is to handle (and so which to leave to other modules, or to the default Apache routines) by putting `Perl*Handler` directives in the configuration files.

Because you need the Perl interpreter to be present for your Perl script to do any processing at all, there is a slight difference between the way that you configure Perl and C handlers to handle parts of the request loop. Ordinarily a C module is written, compiled and configured to hook into a specific phase of the request loop. For a Perl handler you compile mod_perl itself to hook into the appropriate phases, as if it were to handle the phases itself. Then you put `Perl*Handler` directives in your configuration file to tell mod_perl that it is to pass the responsibility for handling that part of the request phase to your Perl module.

mod_perl is an Apache module written in C. As most programmers will only need to handle the response phase, in the default compilation most of the `Perl*Handler`s are disabled. When you configure the *Makefile.PL* file for its compilation, you must specify whether or not you will want to handle parts of the request loop other than the usual content generation phase. If so you need to specify which parts. See the "Callback Hooks" section for how to do this.

Apache specifies about eleven phases of the request loop, namely (and in order of processing): Post-Read-Request, URI Translation, Header Parsing, Access Control, Authentication, Authorization, MIME type checking, FixUp, Response (also known as the Content handling phase), Logging and finally Cleanup. These are the stages of a request where the Apache API allows a module to step in and do something. There is a dedicated `Perl*Handler` for each of these stages plus a couple of others which don’t correspond to parts of the request loop.

We call them `Perl*Handler` directives because the names of the many mod_perl handler directives for the various phases of the request loop all follow the same format. The `*` in `Perl*Handler` is a placeholder to be replaced by something which identifies the phase to be handled. For example `PerlLogHandler` is a Perl Handler which (fairly obviously) handles the logging phase.

The slight exception is `PerlHandler`, which you can think of as `PerlResponseHandler`. It is the content generation handler and so it is probably the one that you will use most frequently.

Note that it is mod_perl which recognizes these directives, and not Apache. They are mod_perl directives, and an ordinary Apache does not recognize them. If you get error messages about these directives being *"perhaps mis-spelled"* it is a sure sign that the appropriate part of mod_perl (or the entire mod_perl module!) is not present in your copy of Apache executable.

The full list of `Perl*Handler`s follows. They are in the order that they are processed by Apache and mod_perl:

```
PerlChildInitHandler
PerlPostReadRequestHandler
PerlInitHandler
PerlTransHandler
PerlHeaderParserHandler
PerlAccessHandler
```

```
PerlAuthenHandler
PerlAuthzHandler
PerlTypeHandler
PerlFixupHandler
PerlHandler
PerlLogHandler
PerlCleanupHandler
PerlChildExitHandler
PerlDispatchHandler
PerlRestartHandler
```

`PerlChildInitHandler` and `PerlChildExitHandler` do not refer to parts of the request loop, they are to allow your modules to initialize data structures and to clean up at the child process start-up and shutdown respectively, for example by allocating and deallocating memory.

All `<Location>`, `<Directory>` and `<Files>` sections contain a physical path specification. Like `PerlChildInitHandler` and `PerlChildExitHandler`, the directives `PerlPostRead-RequestHandler` and `PerlTransHandler` cannot be used in these sections, nor in *.htaccess* files, because it is not until the end of the Translation Handler (`PerlTransHandler`) phase that the path translation is completed and a physical path is known.

`PerlInitHandler` changes its behaviour depending upon where it is used. In any case it is the first handler to be invoked in serving a request. If found outside any `<Location>`, `<Directory>` or `<Files>` section (at the top level), it is an alias for `PerlPostReadRequestHandler`. When inside any such section it is an alias for `PerlHeaderParserHandler`.

Starting from `PerlHeaderParserHandler` the requested URI has been mapped to a physical server pathname, and thus it can be used to match a `<Location>`, `<Directory>` or `<Files>` configuration section, or to look in a *.htaccess* file if such a file exists in the specified directory in the translated path.

`PerlDispatchHandler` and `PerlRestartHandler` do not correspond to parts of the Apache API, but allow you to fine-tune the mod_perl API.

The Apache documentation will tell you all about these stages and what your modules can do. By default, most of these hooks are disabled at compile time, see the"Callback Hooks" section for information on enabling them.

4.4.6 The handler subroutine

By default the mod_perl API expects a subroutine called `handler()` to handle the request in the registered `Perl*Handler` module. Thus if your module implements this subroutine, you can register the handler with mod_perl like this:

```
Perl*Handler Apache::Foo
```

Replace `Perl*Handler` with the name of a specific handler from the list given above. mod_perl will preload the specified module for you. Please note that this approach will not preload the module at startup. To make sure it gets loaded you have three options: you can explicitly preload it with the `PerlModule` directive:

```
PerlModule Apache::Foo
```

You can preload it at the startup file:

```
use Apache::Foo ();
```

Or you can use a nice shortcut that the `Perl*Handler` syntax provides:

```
Perl*Handler +Apache::Foo
```

Note the leading `+` character. This directive is equivalent to:

```
PerlModule Apache::Foo
Perl*Handler Apache::Foo
```

If you decide to give the handler routine a name other than `handler`, for example `my_handler`, you must preload the module and explicitly give the name of the handler subroutine:

```
PerlModule Apache::Foo
Perl*Handler Apache::Foo::my_handler
```

As you have seen, this will preload the module at server startup.

If a module needs to know which handler is currently being run, it can find out with the *current_callback* method. This method is most useful to *PerlDispatchHandlers* which wish to take action for certain phases only.

```
if ($r->current_callback eq "PerlLogHandler") {
    $r->warn("Logging request");
}
```

4.4.7 Stacked Handlers

With the mod_perl stacked handlers mechanism, during any stage of a request it is possible for more than one `Perl*Handler` to be defined and run.

`Perl*Handler` directives (in your configuration files) can define any number of subroutines. For example:

```
PerlTransHandler OneTrans TwoTrans RedTrans BlueTrans
```

With the method `Apache->push_handlers()`, callbacks (handlers) can be added to a stack *at runtime* by mod_perl scripts.

`Apache->push_handlers()` takes the callback hook name as its first argument and a subroutine name or reference as its second.

Here’s an example:

```
use Apache::Constants qw(:common);
sub my_logger {
  #some code here
  return OK;
}
Apache->push_handlers("PerlLogHandler", \&my_logger);
```

Here's another one:

```
use Apache::Constants qw(:common);
$r->push_handlers("PerlLogHandler", sub {
    print STDERR "__ANON__ called\n";
    return OK;
});
```

After each request, this stack is erased.

All handlers will be called unless a handler returns a status other than `OK` or `DECLINED`.

Example uses:

`CGI.pm` maintains a global object for its plain function interface. Since the object is global, it does not go out of scope, `DESTROY` is never called. `CGI->new` can call:

```
Apache->push_handlers("PerlCleanupHandler", \&CGI::_reset_globals);
```

This function will be called during the final stage of a request, refreshing `CGI.pm`'s globals before the next request comes in.

`Apache::DCELogin` establishes a DCE login context which must exist for the lifetime of a request, so the `DCE::Login` object is stored in a global variable. Without stacked handlers, users must set

```
PerlCleanupHandler Apache::DCELogin::purge
```

in the configuration files to destroy the context. This is not "user-friendly". Now, `Apache::DCELogin::handler` can call:

```
Apache->push_handlers("PerlCleanupHandler", \&purge);
```

Persistent database connection modules such as `Apache::DBI` could push a `PerlCleanupHandler` handler that iterates over `%Connected`, refreshing connections or just checking that connections have not gone stale. Remember, by the time we get to `PerlCleanupHandler`, the client has what it wants and has gone away, so we can spend as much time as we want here without slowing down response time to the client (although the process itself is unavailable for serving new requests before the operation is completed).

`PerlTransHandlers` (e.g. `Apache::MsqlProxy`) may decide, based on the URI or some arbitrary condition, whether or not to handle a request. Without stacked handlers, users must configure it themselves:

```
PerlTransHandler Apache::MsqlProxy::translate
PerlHandler      Apache::MsqlProxy
```

`PerlHandler` is never actually invoked unless `translate()` sees that the request is a proxy request (`$r->proxyreq`). If it is a proxy request, `translate()` sets `$r->handler("perl-script")`, and only then will `PerlHandler` handle the request. Now users do not have to specify `PerlHandler Apache::MsqlProxy`, the `translate()` function can set it with `push_handlers()`.

Imagine that you want to include footers, headers, etc., piecing together a document, without using SSI. The following example shows how to implement it. First we prepare the code as follows:

```
Test/Compose.pm
---------------
package Test::Compose;
use Apache::Constants qw(:common);

sub header {
   my $r = shift;
   $r->content_type("text/plain");
   $r->send_http_header;
   $r->print("header text\n");
   return OK;
}
sub body   { shift->print("body text\n")   ; return OK}
sub footer { shift->print("footer text\n") ; return OK}
1;
__END__

# in httpd.conf or perl.conf
PerlModule Test::Compose
<Location /foo>
   SetHandler "perl-script"
   PerlHandler Test::Compose::header Test::Compose::body Test::Compose::footer
 </Location>
```

Parsing the output of another PerlHandler? This is a little more tricky, but consider:

```
<Location /foo>
  SetHandler "perl-script"
  PerlHandler OutputParser SomeApp
</Location>

<Location /bar>
  SetHandler "perl-script"
  PerlHandler OutputParser AnotherApp
</Location>
```

Now, `OutputParser` goes first, but it `untie()`'s `*STDOUT` and re-`tie()`'s it to its own package like so:

```
package OutputParser;

sub handler {
    my $r = shift;
    untie *STDOUT;
```

```
    tie *STDOUT => 'OutputParser', $r;
 }

sub TIEHANDLE {
    my ($class, $r) = @_;
    bless { r => $r}, $class;
}

sub PRINT {
    my $self = shift;
    for (@_) {
        #do whatever you want to $_ for example:
        $self->{r}->print($_ . "[insert stuff]");
    }
}

1;
__END__
```

To build in this feature, configure with:

```
% perl Makefile.PL PERL_STACKED_HANDLERS=1 [ ... ]
```

If you want to test whether your running mod_perl Apache can stack handlers, the method `Apache->can_stack_handlers` will return `TRUE` if mod_perl was configured with `PERL_STACKED_HANDLERS=1`, and `FALSE` otherwise.

4.4.8 Perl Method Handlers

If a `Perl*Handler` is prototyped with `$$`, this handler will be invoked as a method. For example:

```
package MyClass;
@ISA = qw(BaseClass);

sub handler ($$) {
    my ($class, $r) = @_;
    ...;
}

package BaseClass;

sub method ($$) {
    my ($class, $r) = @_;
    ...;
}

1;
```

Configuration:

```
PerlHandler MyClass
```

or

```
PerlHandler MyClass->handler
```

Since the handler is invoked as a method, it may inherit from other classes:

```
PerlHandler MyClass->method
```

In this case, the `MyClass` class inherits this method from `BaseClass`. This means that any method of `MyClass` or any of its parent classes can serve as a mod_perl handler, and that you can apply good OO methodology within your mod_perl handlers.

For instance, you could have this base class:

```
package ServeContent;

use Apache::Constants qw(OK);

sub handler($$) {
   my ($class, $r) = @_;

   $r->send_http_header('text/plain');
   $r->print($class->get_content());

   return OK;
}

sub get_content {
   return 'Hello World';
}

1;
```

And then use the same base class for different contents:

```
package HelloWorld;

use ServeContent;
@ISA = qw(ServeContent);

sub get_content {
   return 'Hello, happy world!';
}

package GoodbyeWorld;

use ServeContent;
@ISA = qw(ServeContent);

sub get_content {
   return 'Goodbye, cruel world!';
}

1;
```

Now you can keep the same handler subroutine for a group of modules which are similiar. The following configuration will enable the handlers from the subclasses:

```
<Location /hello>
    SetHandler perl-script
    PerlHandler HelloWorld->handler
</Location>

<Location /bye>
    SetHandler perl-script
    PerlHandler GoodbyeWorld->handler
</Location>
```

To build in this feature, configure with:

```
% perl Makefile.PL PERL_METHOD_HANDLERS=1 [ ... ]
```

4.4.9 PerlFreshRestart

To reload `PerlRequire`, `PerlModule` and other `use()`'d modules, and to flush the `Apache::Registry` cache on server restart, add to *httpd.conf*:

```
PerlFreshRestart On
```

Make sure you read Evil things might happen when using PerlFreshRestart.

Starting from mod_perl version 1.22 `PerlFreshRestart` is ignored when mod_perl is compiled as a DSO. But it almost doesn't matter, since mod_perl as a DSO will do a full tear-down (perl_destruct()). So it's still a *FreshRestart*, just fresher than static (non-DSO) mod_perl :)

But note that even if you have

```
PerlFreshRestart Off
```

and mod_perl as a DSO you will still get a *FreshRestart*.

4.4.10 PerlSetEnv and PerlPassEnv

```
PerlSetEnv key val
PerlPassEnv key
```

`PerlPassEnv` passes, `PerlSetEnv` sets and passes *ENVironment* variables to your scripts. You can access them in your scripts through `%ENV` (e.g. `$ENV{"key"}`). These commands are useful to pass information to your handlers or scripts, or to any modules you use that require some additional configuration.

For example, the Oracle RDBMS requires a number of `ORACLE_*` environment variables to be set so that you can connect to it through `DBI`. So you might want to put this in your *httpd.conf*:

```
PerlSetEnv ORACLE_BASE /oracle
PerlSetEnv ORACLE_HOME /oracle
      :             :
```

You can then use `DBI` to access your oracle server without having to set the environment variables in your handlers.

`PerlPassEnv` proposes another approach: you might want to set the corresponding environment variables in your shell, and not reproduce the information in your *httpd.conf*. For example, you might have this in your *.bash_profile*:

```
ORACLE_BASE=/oracle
ORACLE_HOME=/oracle
export ORACLE_BASE ORACLE_HOME
```

However, Apache (or mod_perl) don't pass on environment variables from the shell by default; you'll have to specify these using either the standard `PassEnv` or mod_perl's `PerlPassEnv` directives.

```
PerlPassEnv ORACLE_BASE ORACLE_HOME
```

One thing to be aware of is that when you start Apache under a shell different than the one you are logged in from, the environment variables could be totally different, so don't be surprised if you get a different value when using `Passenv`/`PerlPassEnv` or none at all. Check the environment Apache is started from. Often it's started from a special account like *apache*, or *nobody*, and can be anything else. Check the value of `User` variable in *httpd.conf* to find out the right answer. Once you figure that out, make sure that the shell Apache starts from has the desired environment variables right. And may be it's a better idea not to rely on the shell variables, but instead set those explicitly using `Setenv`/`PerlSetEnv`.

Regarding the setting of `PerlPassEnv PERL5LIB` in *httpd.conf*: if you turn on taint checks (`PerlTaintCheck On`), `$ENV{PERL5LIB}` will be ignored (unset). See the 'Switches -w, -T' section.

While the Apache's `SetEnv`/`PassEnv` and mod_perl's `PerlSetEnv`/`PerlPassEnv` apparently do the same thing, the former doesn't happen until the fixup phase, the latter happens as soon as possible, so those variables are available before then, e.g. in `PerlAuthenHandler` for `$ENV{ORACLE_HOME}` (or another environment variable that you need in these early request processing stages).

4.4.11 PerlSetVar and PerlAddVar

`PerlSetVar` is very similar to `PerlSetEnv`; however, variables set using `PerlSetVar` are only available through the mod_perl API, and is thus more suitable for configuration. For example, environment variables are available to all, and might show up on casual "print environment" scripts, which you might not like. `PerlSetVar` is well-suited for modules needing some configuration, but not wanting to implement first-class configuration handlers just to get some information.

```
PerlSetVar foo bar
```

or

```
<Perl>
  push @{ $Location{"/"}->{PerlSetVar} }, [ foo => 'bar' ];
</Perl>
```

and in the code you read it with:

```
my $r = Apache->request;
print $r->dir_config('foo');
```

The above prints:

```
bar
```

Note that you cannot do this:

```
push @{ $Location{"/"}->{PerlSetVar} }, [ foo => \%bar ];
```

All values are treated as strings, so you will get a stringified reference to a hash as a value (something which will look like "`HASH(0x87a5108)`"). This cannot be turned back into a reference and therefore into the original hash upon retrieval.

However you can use the `PerlAddVar` directive to push more values into the variable, emulating arrays. For example:

```
PerlSetVar foo bar
PerlAddVar foo bar1
PerlAddVar foo bar2
```

or the equivalent:

```
PerlAddVar foo bar
PerlAddVar foo bar1
PerlAddVar foo bar2
```

To retrieve the values use the `$r->dir_config->get()` method:

```
my @foo = $r->dir_config->get('foo');
```

or

```
my %foo = $r->dir_config->get('foo');
```

Make sure that you use an even number of elements if you store the retrieved values in a hash, like this:

```
PerlAddVar foo key1
PerlAddVar foo value1
PerlAddVar foo key2
PerlAddVar foo value2
```

Then `%foo` will have a structure like this:

```
%foo = (
    key1 => 'value1',
    key2 => 'value2',
);
```

There are some things you should know about sub requests and `$r->dir_config`. For `$r->lookup_uri`, everything works as expected, because all normal phases are run. You can then retrieve variables set in the server scope of the configuration, in `<VirtualHost>` sections, in `<Location>` sections, etc.

However, when using the `$r->lookup_file` method, you are effectively skipping the URI translation phase. This means that the URI won't be known by Apache, only the file name to retrieve. As such, `<Location>` sections won't be applied. This means that if you were using:

```
Alias /perl-subr/ /home/httpd/perl-subr/
<Location /perl-subr>
  PerlSetVar foo bar
  PerlSetVar foo2 bar2
</Location>
```

And issue a subrequest using `$r->lookup_file` and try to retrieve its directory configuration (`Apache::SubRequest` class is just a subclass of `Apache`):

```
my $subr = $r->lookup_file('/home/httpd/perl-subr/script.pl');
print $subr->dir_config('foo');
```

You won't get the results you wanted.

As a side note: the issue we discussed here means that */perl-subr/script.pl* won't even run under mod_perl if configured in the normal Apache::Registry way (using a `<Location>` section), because the `<Location>` blocks won't be applied. You'd have to use a `<Directory>` or `<Files>` section configuration to achieve the desired effect. As to the `PerlSetVar` discussion, using `<Directory>` or `<Files>` section would solve the problem.

4.4.12 PerlSetupEnv

`PerlSetupEnv On` will allow you to access the environment variables like `$ENV{REQUEST_URI}`, which are available under CGI. However, when programming handlers, there are always better ways to access these variables through the Apache API. Therefore, it is recommended to turn it `Off` except for scripts which absolutely require it. See PerlSetupEnv Off.

4.4.13 PerlWarn and PerlTaintCheck

For **PerlWarn** and **PerlTaintCheck** directives see the 'Switches -w, -T' section.

4.4.14 MinSpareServers MaxSpareServers StartServers MaxClients MaxRequestsPerChild

`MinSpareServers`, `MaxSpareServers`, `StartServers` and `MaxClients` are standard Apache configuration directives that control the number of servers that will be launched at server startup and kept alive during the server's operation.

`MaxRequestsPerChild` lets you specify the maximum number of requests which each child will be allowed to serve. When a process has served `MaxRequestsPerChild` requests the parent kills it and replaces it with a new one. There may also be other reasons why a child is killed, so it does not mean that each child will in fact serve this many requests, only that it will not be allowed to serve more than that number.

These five directives are very important for achieving the best performance from your server. The section ' Performance Tuning by Tweaking Apache Configuration' provides all the details.

4.5 The Startup File

At server startup, before child processes are spawned to receive incoming requests, there is more that can be done than just preloading files. You might want to register code that will initialize a database connection for each child when it is forked, tie read-only dbm files, etc.

The *startup.pl* file is an ideal place to put the code that should be executed when the server starts. Once you have prepared the code, load it in *httpd.conf* before the rest of the mod_perl configuration directives like this:

```
PerlRequire  /home/httpd/perl/lib/startup.pl
```

I must stress that all the code that is run at server initialization time is run with root privileges if you are executing it as the root user (which you have to do unless you choose to run the server on an unprivileged port, above 1024). This means that anyone who has write access to a script or module that is loaded by `PerlModule` or `PerlRequire` effectively has root access to the system. You might want to take a look at the new and experimental `PerlOpmask` directive and `PERL_OPMASK_DEFAULT` compile time option to try to disable some of the more dangerous operations.

Since the startup file is a file written in plain Perl, one can validate its syntax with:

```
% perl -c /home/httpd/perl/lib/startup.pl
```

4.5.1 The Sample Startup File

Let's look at a real world startup file:

```
startup.pl
----------
use strict;

# Extend @INC if needed
```

```
use lib qw(/dir/foo /dir/bar);

# Make sure we are in a sane environment.
$ENV{MOD_PERL} or die "not running under mod_perl!";

# For things in the "/perl" URL
use Apache::Registry;

# Load Perl modules of your choice here
# This code is interpreted *once* when the server starts
use LWP::UserAgent ();
use Apache::DBI ();
use DBI ();

# Tell me more about warnings
use Carp ();
$SIG{__WARN__} = \&Carp::cluck;

# Load CGI.pm and call its compile() method to precompile
# (but not to import) its autoloaded methods.
use CGI ();
CGI->compile(':all');

# Initialize the database connections for each child
Apache::DBI->connect_on_init
("DBI:mysql:database=test;host=localhost",
 "user","password",
 {
  PrintError => 1, # warn() on errors
  RaiseError => 0, # don't die on error
  AutoCommit => 1, # commit executes immediately
 }
);

1;
```

Now we'll review the code explaining why each line is used.

```
use strict;
```

This pragma is worth using in every script longer than half a dozen lines. It will save a lot of time and debugging later on.

```
use lib qw(/dir/foo /dir/bar);
```

The only chance to permanently modify `@INC` before the server is started is with this command. Later the running code can modify `@INC` just for the moment it `require()`'s some file, and then `@INC`'s value gets reset to what it was originally.

```
$ENV{MOD_PERL} or die "not running under mod_perl!";
```

A sanity check, if Apache/mod_perl wasn't properly built, the above code will abort the server startup.

```
use Apache::Registry;
use LWP::UserAgent ();
use Apache::DBI ();
use DBI ();
```

Preload the modules that get used by our Perl code serving the requests. Unless you need the symbols (variables and subroutines) exported by the modules you preload to accomplish something within the startup file, don't import them, since it's just a waste of startup time. Instead use the empty list `()` to tell the `import()` function not to import anything.

```
use Carp ();
$SIG{__WARN__} = \&Carp::cluck;
```

This is a useful snippet to enable extended warnings logged in the error_log file. In addition to basic warnings, a trace of calls is added. This makes the tracking of the potential problem a much easier task, since you know who called whom. For example, with normal warnings you might see:

```
Use of uninitialized value at
    /usr/lib/perl5/site_perl/5.005/Apache/DBI.pm  line 110.
```

but you have no idea where it was called from. When we use `Carp` as shown above we might see:

```
Use of uninitialized value at
          /usr/lib/perl5/site_perl/5.005/Apache/DBI.pm line 110.
    Apache::DBI::connect(undef, 'mydb::localhost', 'user',
       'passwd', 'HASH(0x87a5108)') called at
          /usr/lib/perl5/site_perl/5.005/i386-linux/DBI.pm line 382
    DBI::connect('DBI', 'DBI:mysql:mydb::localhost', 'user',
       'passwd', 'HASH(0x8375e4c)') called at
          /usr/lib/perl5/site_perl/5.005/Apache/DBI.pm line 36
    Apache::DBI::__ANON__('Apache=SCALAR(0x87a50c0)') called at
          PerlChildInitHandler subroutine
          'Apache::DBI::__ANON__' line 0
    eval {...} called at PerlChildInitHandler subroutine
          'Apache::DBI::__ANON__' line 0
```

we clearly see that the warning was triggered by eval()'uating the `Apache::DBI::__ANON__` which called `DBI::connect` (with the arguments that we see as well), which in turn called the `Apache::DBI::connect` method. Now we know where to look for our problem.

```
use CGI ();
CGI->compile(':all');
```

Some modules create their subroutines at run time to improve their load time. This helps when the module includes many subroutines, but only a few are actually used. `CGI.pm` falls into this category. Since with mod_perl the module is loaded only once, it might be a good idea to precompile all or a part of its methods.

`CGI.pm`'s `compile()` method performs this task. Notice that this is a proprietary function of this module, other modules can implement this feature or not and use this or some other name for this functionality. As with all modules we preload in the startup file, we don't import symbols from them as they will be lost when they go out of the file's scope.

Note that starting with `CGI.pm` version 2.46, the recommended method to precompile the code in `CGI.pm` is:

```
use CGI qw(-compile :all);
```

But the old method is still available for backward compatibility.

```
1;
```

As *startup.pl* is run through Perl's `require()`, it has to return a true value so that Perl can make sure it has been successfully loaded. Don't forget this (it's very easy to forget it).

See also the 'Apache::Status -- Embedded interpreter status information' section.

4.5.2 What Modules You Should Add to the Startup File and Why

Every module loaded at server startup will be shared among the server children, saving a lot of RAM on your machine. Usually I put most of the code I develop into modules and preload them.

You can even preload your CGI script with `Apache::RegistryLoader` (See Preload Perl modules at server startup) and you can get the children to preopen their database connections with `Apache::DBI`.

4.5.3 The Confusion with use() in the Server Startup File

Some people wonder why you need to duplicate the `use()` clause in the startup file and in the script itself. The confusion arises due to misunderstanding the `use()` function. `use()` normally performs two operations, namely `require()` and `import()`, called within a `BEGIN` block. See the section "use()" for a detailed explanation of the use(), require() and import() functions.

In the startup file we don't want to import any symbols since they will be lost when we leave the scope of the startup file anyway, i.e. they won't be visible to any of the child processes which run our mod_perl scripts. Instead we want to preload the module in the startup file and then import any symbols that we actually need in each script individually.

Normally when we write `use MyModule;`, `use()` will both load the module and import its symbols; so for the startup file we write `use MyModule ();` and the empty parentheses will ensure that the module is loaded but that no symbols are imported. Then in the actual mod_perl script we write `use()` in the standard way, e.g. `use MyModule;`. Since the module has already been preloaded, the only action taken is to import the symbols. For example in the startup file you write:

```
use CGI ();
```

since you probably don't need any symbols to be imported there. But in your code you would probably write:

```
use CGI qw(:html);
```

For example, if you have `use()`'d `Apache::Constants` in the startup file, it does not mean you can have the following handler:

```
package MyModule;
sub handler {
  my $r = shift;
  ## Cool stuff goes here
  return OK;
}
1;
```

You would either need to add:

```
use Apache::Constants qw( OK );
```

Or use the fully qualified name:

```
return Apache::Constants::OK;
```

If you want to use the function interface without exporting the symbols, use fully qualified function names, e.g. `CGI::param`. The same rule applies to variables, you can import variables and you can access them by their full name. e g. `$My::Module::bar`. When you use the object oriented (method) interface you don't need to export the method symbols.

Technically, you aren't required to supply the `use()` statement in your (handler?) code if it was already loaded during server startup (i.e. by '`PerlRequire startup.pl`'). When writing your code, however, you should not assume the module code has been preloaded. In the future, you or someone else will revisit this code and will not understand how it is possible to use a module's methods without first loading the module itself.

Read the `Exporter` and `perlmod` manpages for more information about `import()`.

4.6 Apache Configuration in Perl

With `<Perl>...</Perl>` sections, it is possible to configure your server entirely in Perl.

4.6.1 Usage

`<Perl>` sections can contain *any* and as much Perl code as you wish. These sections are compiled into a special package whose symbol table mod_perl can then walk and grind the names and values of Perl variables/structures through the Apache core configuration gears. Most of the configuration directives can be represented as scalars (`$scalar`) or lists (`@list`). A `@list` inside these sections is simply converted into a space delimited string for you. Here is an example:

```
 httpd.conf
------------
<Perl>
@PerlModule = qw(Mail::Send Devel::Peek);

#run the server as whoever starts it
```

```
$User  = getpwuid($>) || $>;
$Group = getgrgid($)) || $);

$ServerAdmin = $User;

</Perl>
```

Block sections such as `<Location>..</Location>` are represented in a `%Location` hash, e.g.:

```
<Perl>

$Location{"/~dougm/"} = {
  AuthUserFile => '/tmp/htpasswd',
  AuthType => 'Basic',
  AuthName => 'test',
  DirectoryIndex => [qw(index.html index.htm)],
  Limit => {
    METHODS => 'GET POST',
    require => 'user dougm',
  },
};

</Perl>
```

If an Apache directive can take two or three arguments you may push strings (the lowest number of arguments will be shifted off the `@list`) or use an array reference to handle any number greater than the minimum for that directive:

```
push @Redirect, "/foo", "http://www.foo.com/";

push @Redirect, "/imdb", "http://www.imdb.com/";

push @Redirect, [qw(temp "/here" "http://www.there.com")];
```

Other section counterparts include `%VirtualHost`, `%Directory` and `%Files`.

To pass all environment variables to the children with a single configuration directive, rather than listing each one via `PassEnv` or `PerlPassEnv`, a `<Perl>` section could read in a file and:

```
push @PerlPassEnv, [$key => $val];
```

or

```
Apache->httpd_conf("PerlPassEnv $key $val");
```

These are somewhat simple examples, but they should give you the basic idea. You can mix in any Perl code you desire. See *eg/httpd.conf.pl* and *eg/perl_sections.txt* in the mod_perl distribution for more examples.

Assume that you have a cluster of machines with similar configurations and only small distinctions between them: ideally you would want to maintain a single configuration file, but because the configurations aren't *exactly* the same (e.g. the `ServerName` directive) it's not quite that simple.

<Perl> sections come to rescue. Now you have a single configuration file and the full power of Perl to tweak the local configuration. For example to solve the problem of the ServerName directive you might have this <Perl> section:

```
<Perl>
$ServerName = ‘hostname‘;
</Perl>
```

For example if you want to allow personal directories on all machines except the ones whose names start with *secure*:

```
<Perl>
$ServerName = ‘hostname‘;
if ( $ServerName !~ /^secure/) {
  $UserDir = "public.html";
} else {
  $UserDir = "DISABLED";
}
</Perl>
```

Behind the scenes, mod_perl defines a package called Apache::ReadConfig. Here it keeps all the variables that you define inside the <Perl> sections. Therefore it’s not necessarily to configure the server within the <Perl> sections. Actually what you can do is to write the Perl code to configure the server just like you’d do in the <Perl> sections, but instead place it into a separate file that should be called during the configuration parsing with either PerlModule or PerlRequire directives, or from within the startup file. All you have to do is to declare the package Apache::ReadConfig within this file. Using the last example:

```
apache_config.pl
----------------
package Apache::ReadConfig;

$ServerName = ‘hostname‘;
if ( $ServerName !~ /^secure/) {
  $UserDir = "public.html";
} else {
  $UserDir = "DISABLED";
}

1;

httpd.conf
----------
PerlRequire /home/httpd/perl/lib/apache_config.pl
```

4.6.2 Enabling

To enable <Perl> sections you should build mod_perl with perl Make-
file.PL PERL_SECTIONS=1 [...].

4.6.3 Caveats

Be careful when you declare package names inside `<Perl>` sections, for example this code has a problem:

```
<Perl>
  package My::Trans;
  use Apache::Constants qw(:common);
  sub handler{ OK }

  $PerlTransHandler = "My::Trans";
</Perl>
```

When you put code inside a `<Perl>` section, by default it actually goes into the `Apache::ReadConfig` package, which is already declared for you. This means that the `PerlTransHandler` we have tried to define above is actually undefined. If you define a different package name within a `<Perl>` section you must make sure to close the scope of that package and return to the `Apache::ReadConfig` package when you want to define the configuration directives, like this:

```
<Perl>
  package My::Trans;
  use Apache::Constants qw(:common);
  sub handler{ OK }

  package Apache::ReadConfig;
  $PerlTransHandler = "My::Trans";
</Perl>
```

4.6.4 Verifying

This section shows how to check and dump the configuration you have made with the help of `<Perl>` sections in *httpd.conf*.

To check the `<Perl>` section syntax outside of httpd, we make it look like a Perl script:

```
<Perl>
# !perl
# ... code here ...
__END__
</Perl>
```

Now you may run:

```
perl -cx httpd.conf
```

In a running httpd you can see how you have configured the `<Perl>` sections through the URI /perl-status, by choosing *Perl Section Configuration* from the menu. In order to make this item show up in the menu you should set `$Apache::Server::SaveConfig` to a true value. When you do that the *Apache::ReadConfig* namespace (in which the configuration data is stored) will not be flushed, making configuration data available to Perl modules at request time.

Example:

```
<Perl>
$Apache::Server::SaveConfig = 1;

$DocumentRoot = ...
...
</Perl>
```

At request time, the value of **$DocumentRoot** can be accessed with the fully qualified name **$Apache::ReadConfig::DocumentRoot**.

You can dump the configuration of `<Perl>` sections like this:

```
<Perl>
use Apache::PerlSections();
...
# Configuration Perl code here
...
print STDERR Apache::PerlSections->dump();
</Perl>
```

Alternatively you can store it in a file:

```
Apache::PerlSections->store("httpd_config.pl");
```

You can then `require()` that file in some other `<Perl>` section.

4.6.5 Strict `<Perl>` Sections

If the Perl code doesn't compile, the server won't start. If the generated Apache config is invalid, `<Perl>` sections have always just logged an error and carried on, since there might be globals in the section that are not intended for the config.

The variable `$Apache::Server::StrictPerlSections` has been added in mod_perl version 1.22. If you set this variable to a true value, for example

```
$Apache::Server::StrictPerlSections = 1;
```

then mod_perl will not tolerate invalid Apache configuration syntax and will `croak` (die) if this is the case. At the time of writing the default value is `0`.

4.6.6 Debugging

If you compile mod_perl with `PERL_TRACE=1` and set the environment variable MOD_PERL_TRACE then you should see some useful diagnostics when mod_perl is processing `<Perl>` sections.

4.6.7 Perl Section Tricks

- The Perl `%ENV` is cleared during startup, but the C environment is left intact and so you can use it to set `@PassEnv`.

4.6.8 References

For more info see *Writing Apache Modules with Perl and C*, Chapter 8: http://modperl.com:9000/book/chapters/ch8.html

4.7 Validating the Configuration Syntax

`apachectl configtest` tests the configuration file without starting the server. You can safely validate the configuration file on your production server, if you run this test before you restart the server with `apachectl restart`. Of course it is not 100% perfect, but it will reveal any syntax errors you might have made while editing the file.

'`apachectl configtest`' is the same as '`httpd -t`' and it doesn't just parse the code in *startup.pl*, it actually executes it. `<Perl>` configuration has always started Perl during the configuration read, and `Perl{Require,Module}` do so as well.

Of course we assume that the code that gets called during this test cannot cause any harm to your running production environment. The following hint shows how to prevent the code in the startup script and `<Perl>` from being executed during the syntax check, if that's what you want.

If you want your startup code to get control over the `-t` (`configtest`) server launch, start the server configuration test with:

```
httpd -t -Dsyntax_check
```

and, if for example you want to prevent your startup code from being executed, at the top of the code add:

```
return if Apache->define('syntax_check');
```

4.8 Enabling Remote Server Configuration Reports

The nifty mod_info module displays the complete server configuration in your browser. In order to use it you have compile it in or, if the server was compiled with DSO mode enabled, load it as an object. Then just uncomment the ready-prepared section in the *httpd.conf* file:

```
<Location /server-info>
  SetHandler server-info
  Order deny,allow
  Deny from all
  Allow from www.example.com
</Location>
```

Now restart the server and issue the request:

```
http://www.example.com/server-info
```

4.9 Publishing Port Numbers other than 80

If you are using a two-server setup, with a mod_perl server listening on a high port, it is advised that you do not publish the number of the high port number in URLs. Rather use a proxying rewrite rule in the non-mod_perl server:

```
RewriteEngine      On
RewriteLogLevel    0
RewriteRule       ^/perl/(.*) http://localhost:8080/perl/$1 [P]
ProxyPassReverse   /          http://localhost/
```

I was told one problem with publishing high port numbers is that IE 4.x has a bug when re-posting data to a non-port-80 URL. It drops the port designator, and uses port 80 anyway.

Another reason is that firewalls probably will have the high port closed, therefore users behind the firewalls will be unable to reach your service, running on the blocked port.

4.10 Configuring Apache + mod_perl with mod_macro

mod_macro is an Apache module written by Fabien Coelho that lets you define and use macros in the Apache configuration file.

mod_macro can be really useful when you have many virtual hosts, and where each virtual host has a number of scripts/modules, most of them with a moderately complex configuration setup.

First download the latest version of mod_macro from http://www.cri.ensmp.fr/~coelho/mod_macro/ , and configure your Apache server to use this module.

Here are some useful macros for mod_perl users:

```
# set up a registry script
<Macro registry>
SetHandler "perl-script"
PerlHandler Apache::Registry
Options +ExecCGI
</Macro>

# example
Alias /stuff /usr/www/scripts/stuff
<Location /stuff>
Use registry
</Location>
```

If your registry scripts are all located in the same directory, and your aliasing rules consistent, you can use this macro:

```
# set up a registry script for a specific location
<Macro registry $location $script>
Alias /$location /home/httpd/perl/scripts/$script
<Location /$location>
SetHandler "perl-script"
PerlHandler Apache::Registry
Options +ExecCGI
</Location>
</Macro>

# example
Use registry stuff stuff.pl
```

If you're using content handlers packaged as modules, you can use the following macro:

```
# set up a mod_perl content handler module
<Macro modperl $module>
SetHandler "perl-script"
Options +ExecCGI
PerlHandler $module
</Macro>

#examples
<Location /perl-status>
PerlSetVar StatusPeek On
PerlSetVar StatusGraph On
PerlSetVar StatusDumper On
Use modperl Apache::Status
</Location>
```

The following macro sets up a Location for use with `HTML::Embperl`. Here we define all ".html" files to be processed by `Embperl`.

```
<Macro embperl>
SetHandler "perl-script"
Options +ExecCGI
PerlHandler HTML::Embperl
PerlSetEnv EMBPERL_FILESMATCH \.html$
</Macro>

# examples
<Location /mrtg>
Use embperl
</Location>
```

Macros are also very useful for things that tend to be verbose, such as setting up Basic Authentication:

```
# Sets up Basic Authentication
<Macro BasicAuth $realm $group>
Order deny,allow
Satisfy any
AuthType Basic
AuthName $realm
AuthGroupFile /usr/www/auth/groups
AuthUserFile /usr/www/auth/users
Require group $group
```

```
Deny from all
</Macro>

# example of use
<Location /stats>
Use BasicAuth WebStats Admin
</Location>
```

Finally, here is a complete example that uses macros to set up simple virtual hosts. It uses the `BasicAuth` macro defined previously (yes, macros can be nested!).

```
<Macro vhost $ip $domain $docroot $admingroup>
<VirtualHost $ip>
ServerAdmin webmaster@$domain
DocumentRoot /usr/www/htdocs/$docroot
ServerName www.$domain
<Location /stats>
Use BasicAuth Stats-$domain $admingroup
</Location>
</VirtualHost>
</Macro>

# define some virtual hosts
Use vhost 10.1.1.1 example.com example example-admin
Use vhost 10.1.1.2 example.net examplenet examplenet-admin
```

mod_macro is also useful in a non vhost setting. Some sites for example have lots of scripts which people use to view various statistics, email settings and etc. It is much easier to read things like:

```
use /forwards email/showforwards
use /webstats web/showstats
```

The actual macros for the last example are left as an exercise to reader. These can be easily constructed based on the examples presented in this section.

4.11 General Pitfalls

4.11.1 My CGI/Perl Code Gets Returned as Plain Text Instead of Being Executed by the Webserver

Check your configuration files and make sure that the `ExecCGI` is turned on in your configurations.

```
<Location /perl>
  SetHandler perl-script
  PerlHandler Apache::Registry
  Options ExecCGI
  allow from all
  PerlSendHeader On
</Location>
```

4.11.2 My Script Works under mod_cgi, but when Called via mod_perl I Get a 'Save-As' Prompt

Did you put **PerlSendHeader On** in the configuration part of the `<Location foo></Location>`.

4.11.3 Is There a Way to Provide a Different startup.pl File for Each Individual Virtual Host

No. Any virtual host will be able to see the routines from a *startup.pl* loaded for any other virtual host.

4.11.4 Is There a Way to Modify @INC on a Per-Virtual-Host or Per-Location Basis.

You can use `PerlSetEnv PERL5LIB ...` or a `PerlFixupHandler` with the `lib` pragma (`use lib qw(...)`).

A better way is to use Apache::PerlVINC

4.11.5 A Script From One Virtual Host Calls a Script with the Same Path From the Other Virtual Host

This has been a bug before, last fixed in 1.15_01, i.e. if you are running 1.15, that could be the problem. You should set this variable in a startup file (which you load with `PerlRequire` in *httpd.conf*):

```
$Apache::Registry::NameWithVirtualHost = 1;
```

But, as we know sometimes a bug turns out to be a feature. If the same script is running for more than one Virtual host on the same machine, this can be a waste, right? Set it to `0` in a startup script if you want to turn it off and have this bug as a feature. (Only makes sense if you are sure that there will be no *other* scripts with the same path/name). It also saves you some memory as well.

```
$Apache::Registry::NameWithVirtualHost = 0;
```

4.11.6 the Server no Longer Retrieves the DirectoryIndex Files for a Directory

The problem was reported by users who declared mod_perl configuration inside a `<Directory>` section for all files matching *.pl. The problem went away after placing the directives in a `<Files>` section.

The mod_alias and mod_rewrite are both Trans handlers in the normal case. So in the setup where both are used, if mod_alias runs first and matches it will return OK and mod_rewrite won't see the request.

The opposite can happen as well, where mod_rewrite rules apply but the `Alias` directives are completely ignored.

The behavior is not random, but depends on the Apache modules loading order. Apache modules are being executed in *reverse* order, i.e. module that was *Added* first will be executed last.

The solution is not to mix mod_rewrite and mod_alias. mod_rewrite does everything mod_alias does--except for `ScriptAlias` which is not really relevant to mod_perl anyway. Don't rely on the module ordering, but use explicitly disjoint URL namespaces for `Alias` and `Rewrite`. In other words any URL regex that can potentially match a `Rewrite` rule should not be used in an `Alias`, and vice versa. Given that mod_rewrite can easily do what mod_alias does, it's no problem.

Here is one of the examples where `Alias` is replaced with `RedirectMatch`. This is a snippet of configuration at the light non-mod_perl Apache server:

```
RewriteEngine     on
RewriteLogLevel   0
RewriteRule       ^/(perl.*)$    http://127.0.0.1:8045/$1  [P,L]
RewriteRule       ^/(mail.*)$    http://127.0.0.1:8045/$1  [P,L]
NoCache           *
ProxyPassReverse  / http://www.example.com/

RedirectMatch permanent ^/$      /pages/index
RedirectMatch permanent ^/foo$   /pages/bar
```

This configuration works fine because any URI that matches one of the redirects will never match one of the rewrite rules.

In the above setup we proxy requests starting with */perl* or */mail* to the mod_perl server, forbid proxy requests to the external sites, and make sure that the proxied requests will use the *http://www.example.com/* as their URL on the way back to the client.

The `RedirectMatch` settings work exactly like if you'd write:

```
Alias /      /pages/index
Alias /foo   /pages/bar
```

But as we told before we don't want to mix the two.

Here is another example where the redirect is done by a rewrite rule:

```
RewriteEngine     on
RewriteLogLevel   0
RewriteMap        lowercase int:tolower
RewriteRule       ^/(perl.*)$  http://127.0.0.1:8042/$1   [P,L]
RewriteRule       ^/$           /pages/welcome.htm        [R=301,L]
RewriteRule       ^(.*)$        ${lowercase:$1}
NoCache           *
ProxyPassReverse  /  http://www.example.com/
```

If we omit the rewrite rule that matches `^/$`, and instead use a redirect, it will never be called, because the URL is still matched by the last rule `^(.*)$`. This is a somewhat contrived example because that last regex could be rewritten as `^(/.+)$` and all would be well.

4.11.7 Do Perl Directives Affect Code Running under mod_cgi?*

No, they don't.

So for example if you do:

```
PerlSetEnv foo bar
```

It'll be seen from mod_perl, but not mod_cgi or any other module.

4.12 Configuration Security Concerns

The more modules you have in your web server, the more complex the code.

The more complex the code in your web server, the more chances for bugs.

The more chances for bugs, the more chance that some of those bugs may involve security breaches.

4.12.1 Choosing User and Group

Because mod_perl runs within an httpd child process, it runs with the `User` ID and `Group` ID specified in the *httpd.conf* file. This `User`/`Group` should have the lowest possible privileges. It should only have access to world readable files, even better only files that belongs to this user. Even so, careless scripts can give away information. You would not want your */etc/passwd* file to be readable over the net, for instance, even if you use shadow passwords.

When a handler needs write permissions, make sure that only the user, the server is running under, has write permissions to the files. Sometimes you need group write permissions, but be very careful, because a buggy or malicious code run in the server may destroy files writable by the server.

4.12.2 Taint Checking

Make sure to run the server under:

```
PerlTaintCheck On
```

setting in the *httpd.conf* file. Taint checking doesn't ensure that your code is completely safe from external hacks, but it does forces you to improve your code to prevent many potential security problems.

4.12.3 Exposing Information About the Server's Component

It is better not to expose the mod_perl server to the outside world, for it creates a potential security risk by revealing which Apache modules used by the server and the OS the server is running on.

You can see what information is revealed by your server, by telneting to it and issuing some request. For example:

```
% telnet localhost 8080
Trying 127.0.0.1
Connected to localhost
Escape character is '^]'.
HEAD / HTTP1.0

HTTP/1.1 200 OK
Date: Sun, 16 Apr 2000 11:06:25 GMT
Server: Apache/1.3.12 (Unix) mod_perl/1.22 mod_ssl/2.6.2 OpenSSL/0.9.5
[more lines snipped]
```

So as you see that a lot of information is revealed and a `Full ServerTokens` has been used.

We never were completely sure why the default of the `ServerTokens` directive in Apache is `Full` rather than `Minimal`. Seems like you would only make it `Full` if you are debugging. Probably the reason for using the `ServerTokens Full` is for a show-off, so Netcraft (http://netcraft.com) and other similar survey services will count more Apache servers, which is good for all of us, but you really want to reveal as little information as possible to the potential crackers.

Another approach is to modify httpd sources to reveal no unwanted information, so all responses will return an empty or phony `Server:` field.

From the other point of view, security by obscurity is a lack of security. Any determined cracker will eventually figure out what version of Apache run and what third party modules you have built in.

An even better approach is to completely hide the mod_perl server behind a front-end or a proxy server, so the server cannot be accessed directly.

4.13 Apache Restarts Twice On Start

When the server is restarted, the configuration and module initialization phases are called twice in total before the children are forked. The second restart is done in order to ensure that future restarts will work correctly, by making sure that all modules can survive a restart (`SIGHUP`). This is very important if you restart a production server.

You can control what code will be executed on the start or restart by checking the value of `$Apache::Server::Starting` and `$Apache::Server::ReStarting` respectively. The former variable is *true* when the server is starting and the latter is *true* when it's restarting.

For example:

```
<Perl>
print STDERR "Server is Starting\n"   if $Apache::Server::Starting;
print STDERR "Server is ReStarting\n" if $Apache::Server::ReStarting;
</Perl>
```

The *startup.pl* file and similar loaded via `PerlModule` or `PerlRequire` are compiled only once. Because once the module is compiled it enters the special `%INC` hash. When Apache restarts--Perl checks whether the module or script in question is already registered in `%INC` and won't try to compile it again.

So the only code that you might need to protect from running on restart is the one in the `<Perl>` sections. But since one usually uses the `<Perl>` sections mainly for on the fly configuration creation, there shouldn't be a reason why it'd be undesirable to run the code more than once.

4.14 Knowing the proxy_pass'ed Connection Type

Let's say that you have a frontend server running mod_ssl, mod_rewrite and mod_proxy. You want to make sure that your user is using a secure connection for some specific actions like login information submission. You don't want to let the user login unless the request was submitted through a secure port.

Since you have to proxy_pass the request between front and backend servers, you cannot know where the connection has come from. Neither is using the HTTP headers reliable.

A possible solution for this problem is to have the mod_perl server listen on two different ports (e.g. 8000 and 8001) and have the mod_rewrite proxy rule in the regular server redirect to port 8000 and the mod_rewrite proxy rule in the SSL virtual host redirect to port 8001. In the mod_perl server just check the `PORT` variable to tell if the connection is secure.

4.15 Adding Custom Configuration Directives

This is covered in the Eagle Book in a great detail. This is just a simple example, showing how to add your own Configuration directives.

```
Makefile.PL
-----------
package Apache::TestDirective;

use ExtUtils::MakeMaker;

use Apache::ExtUtils qw(command_table);
use Apache::src ();

my @directives = ({
    name        =>  'Directive4',
    errmsg      =>  'Anything',
    args_how    =>  'RAW_ARGS',
    req_override=>  'OR_ALL',
});
```

```
command_table(\@directives);

WriteMakefile(
    NAME         => 'Apache::TestDirective',
    VERSION_FROM => 'TestDirective.pm',
    INC          => Apache::src->new->inc,
);

TestDirective.pm
----------------
package Apache::TestDirective;

use strict;
use Apache::ModuleConfig ();
use DynaLoader ();

if ($ENV{MOD_PERL}) {
    no strict;
    $VERSION = '0.01';
    @ISA = qw(DynaLoader);
    __PACKAGE__->bootstrap($VERSION); #command table, etc.
}

sub Directive4 {

    warn "Directive4 @_\n";
}

1;
__END__
```

In the mod_perl source tree, add this to *t/docs/startup.pl*:

```
use blib qw(/home/dougm/test/Apache/TestDirective);
```

and at the bottom of *t/conf/httpd.conf*:

```
PerlModule Apache::TestDirective
Directive4 hi
```

Test it:

```
% make start_httpd
% make kill_httpd
```

You should see:

```
Directive4 Apache::TestDirective=HASH(0x83379d0)
Apache::CmdParms=SCALAR(0x862b80c) hi
```

And in the error log file:

```
% grep Directive4 t/logs/error_log
Directive4 Apache::TestDirective=HASH(0x83119dc)
Apache::CmdParms=SCALAR(0x8326878) hi
```

If it didn't work as expected try building mod_perl with PERL_TRACE=1, then do:

```
setenv MOD_PERL_TRACE all
```

before starting the server. Now you should get some useful diagnostics.

4.16 Maintainers

Maintainer is the person(s) you should contact with updates, corrections and patches.

- Stas Bekman [http://stason.org/]

4.17 Authors

- Stas Bekman [http://stason.org/]

Only the major authors are listed above. For contributors see the Changes file.

5 CGI to mod_perl Porting. mod_perl Coding guidelines.

5.1 Description

This chapter is relevant both to writing a new CGI script or perl handler from scratch and to migrating an application from plain CGI to mod_perl.

It also addresses the situation where the CGI script being ported does the job, but is too dirty to be altered easily to run as a mod_perl program. (`Apache::PerlRun` mode)

If you are at the porting stage, you can use this chapter as a reference for possible problems you might encounter when running an existing CGI script in the new mode.

If your project schedule is tight, I would suggest converting to mod_perl in the following steps: Initially, run all the scripts in the `Apache::PerlRun` mode. Then as time allows, move them into `Apache::Registry` mode. Later if you need Apache Perl API functionality you can always add it.

If you are about to write a new CGI script from scratch, it would be a good idea to learn about possible mod_perl related pitfalls and to avoid them in the first place.

If you don't need mod_cgi compatibility, it's a good idea to start writing using the mod_perl API in first place. This will make your application a little bit more efficient and it will be easier to use the full mod_perl feature set, which extends the core Perl functionality with Apache specific functions and overridden Perl core functions that were reimplemented to work better in mod_perl environment.

5.2 Before you start to code

It can be a good idea to tighten up some of your Perl programming practices, since mod_perl doesn't tolerate sloppy programming.

This chapter relies on a certain level of Perl knowledge. Please read through the Perl Reference chapter and make sure you know the material covered there. This will allow me to concentrate on pure mod_perl issues and make them more prominent to the experienced Perl programmer, which would otherwise be lost in the sea of Perl background notes.

Additional resources:

- **Perl Module Mechanics**

  This page describes the mechanics of creating, compiling, releasing, and maintaining Perl modules. http://world.std.com/~swmcd/steven/perl/module_mechanics.html

  The information is very relevant to a mod_perl developer.

- **The Eagle Book**

  "Writing Apache Modules with Perl and C" is a "must have" book!

See the details at http://www.modperl.com .

- **"Programming Perl" Book**
- **"Perl Cookbook" Book**
- **"Object Oriented Perl" Book**

5.3 Exposing Apache::Registry secrets

Let’s start with some simple code and see what can go wrong with it, detect bugs and debug them, discuss possible pitfalls and how to avoid them.

I will use a simple CGI script, that initializes a `$counter` to 0, and prints its value to the browser while incrementing it.

```
counter.pl:
----------
#!/usr/bin/perl -w
use strict;

print "Content-type: text/plain\r\n\r\n";

my $counter = 0; # Explicit initialization technically redundant

for (1..5) {
  increment_counter();
}

sub increment_counter{
  $counter++;
  print "Counter is equal to $counter !\r\n";
}
```

You would expect to see the output:

```
Counter is equal to 1 !
Counter is equal to 2 !
Counter is equal to 3 !
Counter is equal to 4 !
Counter is equal to 5 !
```

And that’s what you see when you execute this script the first time. But let’s reload it a few times... See, suddenly after a few reloads the counter doesn’t start its count from 1 any more. We continue to reload and see that it keeps on growing, but not steadily starting almost randomly at 10, 10, 10, 15, 20... Weird...

```
Counter is equal to 6 !
Counter is equal to 7 !
Counter is equal to 8 !
Counter is equal to 9 !
Counter is equal to 10 !
```

We saw two anomalies in this very simple script: Unexpected increment of our counter over 5 and inconsistent growth over reloads. Let's investigate this script.

5.3.1 The First Mystery

First let's peek into the `error_log` file. Since we have enabled the warnings what we see is:

```
Variable "$counter" will not stay shared
at /home/httpd/perl/conference/counter.pl line 13.
```

The *Variable "$counter" will not stay shared* warning is generated when the script contains a named nested subroutine (a named - as opposed to anonymous - subroutine defined inside another subroutine) that refers to a lexically scoped variable defined outside this nested subroutine. This effect is explained in my () Scoped Variable in Nested Subroutines.

Do you see a nested named subroutine in my script? I don't! What's going on? Maybe it's a bug? But wait, maybe the perl interpreter sees the script in a different way, maybe the code goes through some changes before it actually gets executed? The easiest way to check what's actually happening is to run the script with a debugger.

But since we must debug it when it's being executed by the webserver, a normal debugger won't help, because the debugger has to be invoked from within the webserver. Luckily Doug MacEachern wrote the `Apache::DB` module and we will use this to debug my script. While `Apache::DB` allows you to debug the code interactively, we will do it non-interactively.

Modify the `httpd.conf` file in the following way:

```
PerlSetEnv PERLDB_OPTS "NonStop=1 LineInfo=/tmp/db.out AutoTrace=1 frame=2"
PerlModule Apache::DB
<Location /perl>
  PerlFixupHandler Apache::DB
  SetHandler perl-script
  PerlHandler Apache::Registry
  Options ExecCGI
  PerlSendHeader On
</Location>
```

Restart the server and issue a request to *counter.pl* as before. On the surface nothing has changed--we still see the correct output as before, but two things happened in the background:

Firstly, the file */tmp/db.out* was written, with a complete trace of the code that was executed.

Secondly, if you have loaded the `Carp` module already, *error_log* now contains the real code that was actually executed. This is produced as a side effect of reporting the *Variable "$counter" will not stay shared at...* warning that we saw earlier. To load the Carp module, you can add:

```
use Carp;
```

in your *startup.pl* file or in the executed code.

Here is the code that was actually executed:

```
  package Apache::ROOT::perl::conference::counter_2epl;
  use Apache qw(exit);
  sub handler {
    BEGIN {
      $^W = 1;
    };
    $^W = 1;

    use strict;

    print "Content-type: text/plain\r\n\r\n";

    my $counter = 0; # Explicit initialization technically redundant

    for (1..5) {
      increment_counter();
    }

    sub increment_counter{
      $counter++;
      print "Counter is equal to $counter !\r\n";
    }
  }
```

The code in the *error_log* wasn't indented. I've indented it for you to stress that the code was wrapped inside the handler() subroutine.

What do we learn from this?

Well firstly that every CGI script is cached under a package whose name is formed from the `Apache::ROOT::` prefix and the relative part of the script's URL (`perl::conference::counter_2epl`) by replacing all occurrences of `/` with `::` and `.` with `_2e`. That's how mod_perl knows what script should be fetched from the cache--each script is just a package with a single subroutine named `handler`.

If we were to add `use diagnostics` to the script we would also see a reference in the error text to an inner (nested) subroutine--`increment_counter` is actually a nested subroutine.

With mod_perl, each subroutine in every `Apache::Registry` script is nested inside the `handler` subroutine.

It's important to understand that the *inner subroutine* effect happens only with code that `Apache::Registry` wraps with a declaration of the `handler` subroutine. If you put all your code into modules, which the main script `use()`s, this effect doesn't occur.

Do not use Perl4-style libraries. Subroutines in such libraries will only be available to the first script in any given interpreter thread to `require()` a library of any given name. This can lead to confusing sporadic failures.

The easiest and the fastest way to solve the nested subroutines problem is to switch every lexically scoped variable foe which you get the warning for to a package variable. The `handler` subroutines are never called re-entrantly and each resides in a package to itself. Most of the usual disadvantates of package scoped variables are, therefore, not a concern. Note, however, that whereas explicit initialization is not always necessary for lexical variables it is usually necessary for these package variables as they persist in subsequent executions of the handler and unlike lexical variables, don't get automatically destroyed at the end of each handler.

```
counter.pl:
----------
#!/usr/bin/perl -w
use strict;

print "Content-type: text/plain\r\n\r\n";

# In Perl <5.6 our() did not exist, so:
#   use vars qw($counter);
our $counter = 0; # Explicit initialization now necessary

for (1..5) {
  increment_counter();
}

sub increment_counter{
  $counter++;
  print "Counter is equal to $counter !\r\n";
}
```

If the variable contains a reference it may hold onto lots of unecessary memory (or worse) if the reference is left to hang about until the next call to the same handler. For such variables you should use `local` so that the value is removed when the `handler` subroutine exits.

```
my $query = CGI->new;
```

becomes:

```
local our $query = CGI->new;
```

All this is very interesting but as a general rule of thumb, unless the script is very short, I tend to write all the code in external libraries, and to have only a few lines in the main script. Generally the main script simply calls the main function of my library. Usually I call it `init()` or `run()`. I don't worry about nested subroutine effects anymore (unless I create them myself :).

The section 'Remedies for Inner Subroutines' discusses many other possible workarounds for this problem.

You shouldn't be intimidated by this issue at all, since Perl is your friend. Just keep the warnings mode `On` and Perl will gladly tell you whenever you have this effect, by saying:

```
Variable "$counter" will not stay shared at ...[snipped]
```

Just don’t forget to check your *error_log* file, before going into production!

By the way, the above example was pretty boring. In my first days of using mod_perl, I wrote a simple user registration program. I’ll give a very simple representation of this program.

```
use CGI;
$q = CGI->new;
my $name = $q->param('name');
print_response();

sub print_response{
  print "Content-type: text/plain\r\n\r\n";
  print "Thank you, $name!";
}
```

My boss and I checked the program at the development server and it worked OK. So we decided to put it in production. Everything was OK, but my boss decided to keep on checking by submitting variations of his profile. Imagine the surprise when after submitting his name (let’s say "The Boss" :), he saw the response "Thank you, Stas Bekman!".

What happened is that I tried the production system as well. I was new to mod_perl stuff, and was so excited with the speed improvement that I didn’t notice the nested subroutine problem. It hit me. At first I thought that maybe Apache had started to confuse connections, returning responses from other people’s requests. I was wrong of course.

Why didn’t we notice this when we were trying the software on our development server? Keep reading and you will understand why.

5.3.2 The Second Mystery

Let’s return to our original example and proceed with the second mystery we noticed. Why did we see inconsistent results over numerous reloads?

That’s very simple. Every time a server gets a request to process, it hands it over one of the children, generally in a round robin fashion. So if you have 10 httpd children alive, the first 10 reloads might seem to be correct because the effect we’ve just talked about starts to appear from the second re-invocation. Subsequent reloads then return unexpected results.

Moreover, requests can appear at random and children don’t always run the same scripts. At any given moment one of the children could have served the same script more times than any other, and another may never have run it. That’s why we saw the strange behavior.

Now you see why we didn’t notice the problem with the user registration system in the example. First, we didn’t look at the `error_log`. (As a matter of fact we did, but there were so many warnings in there that we couldn’t tell what were the important ones and what were not). Second, we had too many server children running to notice the problem.

A workaround is to run the server as a single process. You achieve this by invoking the server with the `-X` parameter (`httpd -X`). Since there are no other servers (children) running, you will see the problem on the second reload.

But before that, let the `error_log` help you detect most of the possible errors--most of the warnings can become errors, so you should make sure to check every warning that is detected by perl, and probably you should write your code in such a way that no warnings appear in the `error_log`. If your `error_log` file is filled up with hundreds of lines on every script invocation, you will have difficulty noticing and locating real problems--and on a production server you’ll soon run out of disk space if your site is popular.

Of course none of the warnings will be reported if the warning mechanism is not turned `On`. Refer to the section "Tracing Warnings Reports" to learn about warnings in general and to the "Warnings" section to learn how to turn them on and off under mod_perl.

5.4 Sometimes it Works, Sometimes it Doesn’t

When you start running your scripts under mod_perl, you might find yourself in a situation where a script seems to work, but sometimes it screws up. And the more it runs without a restart, the more it screws up. Often the problem is easily detectable and solvable. You have to test your script under a server running in single process mode (`httpd -X`).

Generally the problem is the result of using global variables. Because global variables don’t change from one script invocation to another unless you change them, you can find your scripts do strange things.

Let’s look at three real world examples:

5.4.1 An Easy Break-in

The first example is amazing--Web Services. Imagine that you enter some site where you have an account, perhaps a free email account. Having read your own mail you decide to take a look at someone else’s.

You type in the username you want to peek at and a dummy password and try to enter the account. On some services this will work!!!

You say, why in the world does this happen? The answer is simple: **Global Variables**. You have entered the account of someone who happened to be served by the same server child as you. Because of sloppy programming, a global variable was not reset at the beginning of the program and voila, you can easily peek into someone else’s email! Here is an example of sloppy code:

```
use vars ($authenticated);
my $q = new CGI;
my $username = $q->param('username');
my $passwd   = $q->param('passwd');
authenticate($username,$passwd);
  # failed, break out
unless ($authenticated){
  print "Wrong passwd";
  exit;
}
```

```
  # user is OK, fetch user's data
show_user($username);

sub authenticate{
  my ($username,$passwd) = @_;
  # some checking
  $authenticated = 1 if SOME_USER_PASSWD_CHECK_IS_OK;
}
```

Do you see the catch? With the code above, I can type in any valid username and any dummy password and enter that user's account, provided she has successfully entered her account before me using the same child process! Since `$authenticated` is global--if it becomes 1 once, it'll stay 1 for the remainder of the child's life!!! The solution is trivial--reset `$authenticated` to 0 at the beginning of the program.

A cleaner solution of course is not to rely on global variables, but rely on the return value from the function.

```
my $q = CGI->new;
my $username = $q->param('username');
my $passwd   = $q->param('passwd');
my $authenticated = authenticate($username,$passwd);
  # failed, break out
unless ($authenticated){
  print "Wrong passwd";
  exit;
}
  # user is OK, fetch user's data
show_user($username);

sub authenticate{
  my ($username,$passwd) = @_;
  # some checking
  return (SOME_USER_PASSWD_CHECK_IS_OK) ? 1 : 0;
}
```

Of course this example is trivial--but believe me it happens!

5.4.2 Thinking mod_cgi

Just another little one liner that can spoil your day, assuming you forgot to reset the `$allowed` variable. It works perfectly OK in plain mod_cgi:

```
$allowed = 1 if $username eq 'admin';
```

But using mod_perl, and if your system administrator with superuser access rights has previously used the system, anybody who is lucky enough to be served later by the same child which served your administrator will happen to gain the same rights.

The obvious fix is:

```
$allowed = $username eq 'admin' ? 1 : 0;
```

5.4.3 Regular Expression Memory

Another good example is usage of the `/o` regular expression modifier, which compiles a regular expression once, on its first execution, and never compiles it again. This problem can be difficult to detect, as after restarting the server each request you make will be served by a different child process, and thus the regex pattern for that child will be compiled afresh. Only when you make a request that happens to be served by a child which has already cached the regex will you see the problem. Generally you miss that. When you press reload, you see that it works (with a new, fresh child). Eventually it doesn't, because you get a child that has already cached the regex and won't recompile because of the `/o` modifier.

An example of such a case would be:

```
my $pat = $q->param("keyword");
foreach( @list ) {
  print if /$pat/o;
}
```

To make sure you don't miss these bugs always test your CGI in single process mode.

To solve this particular `/o` modifier problem refer to Compiled Regular Expressions.

5.5 Script's name space

Scripts under `Apache::Registry` do not run in package `main`, they run in a unique name space based on the requested URI. For example, if your URI is `/perl/test.pl` the package will be called `Apache::ROOT::perl::test_2epl`.

5.6 @INC and mod_perl

The basic Perl `@INC` behaviour is explained in section use(), require(), do(), %INC and @INC Explained.

When running under mod_perl, once the server is up `@INC` is frozen and cannot be updated. The only opportunity to *temporarily* modify `@INC` is while the script or the module are loaded and compiled for the first time. After that its value is reset to the original one. The only way to change `@INC` permanently is to modify it at Apache startup.

Two ways to alter `@INC` at server startup:

- In the configuration file. For example add:

  ```
  PerlSetEnv PERL5LIB /home/httpd/perl
  ```

  or

```
PerlSetEnv PERL5LIB /home/httpd/perl:/home/httpd/mymodules
```

Note that this setting will be ignored if you have the `PerlTaintCheck` mode turned on.

- In the startup file directly alter the `@INC`. For example

```
startup.pl
----------
use lib qw(/home/httpd/perl /home/httpd/mymodules);
1;
```

and load the startup file from the configuration file by:

```
PerlRequire /path/to/startup.pl
```

5.7 Reloading Modules and Required Files

You might want to read the "use(), require(), do(), %INC and @INC Explained" before you proceed with this section.

When you develop plain CGI scripts, you can just change the code, and rerun the CGI from your browser. Since the script isn't cached in memory, the next time you call it the server starts up a new perl process, which recompiles it from scratch. The effects of any modifications you've applied are immediately present.

The situation is different with `Apache::Registry`, since the whole idea is to get maximum performance from the server. By default, the server won't spend time checking whether any included library modules have been changed. It assumes that they weren't, thus saving a few milliseconds to stat() the source file (multiplied by however many modules/libraries you use() and/or require() in your script.)

The only check that is done is to see whether your main script has been changed. So if you have only scripts which do not use() or require() other perl modules or packages, there is nothing to worry about. If, however, you are developing a script that includes other modules, the files you use() or require() aren't checked for modification and you need to do something about that.

So how do we get our mod_perl-enabled server to recognize changes in library modules? Well, there are a couple of techniques:

5.7.1 Restarting the server

The simplest approach is to restart the server each time you apply some change to your code. See Server Restarting techniques.

After restarting the server about 100 times, you will tire of it and you will look for other solutions.

5.7.2 Using Apache::StatINC for the Development Process

Help comes from the `Apache::StatINC` module. When Perl pulls a file via require(), it stores the full pathname as a value in the global hash `%INC` with the file name as the key. `Apache::StatINC` looks through `%INC` and immediately reloads any files that have been updated on disk.

To enable this module just add two lines to `httpd.conf`.

```
PerlModule Apache::StatINC
PerlInitHandler Apache::StatINC
```

To be sure it really works, turn on debug mode on your development box by adding `PerlSetVar StatINCDebug On` to your config file. You end up with something like this:

```
PerlModule Apache::StatINC
<Location /perl>
  SetHandler perl-script
  PerlHandler Apache::Registry
  Options ExecCGI
  PerlSendHeader On
  PerlInitHandler Apache::StatINC
  PerlSetVar StatINCDebug On
</Location>
```

Be aware that only the modules located in `@INC` are reloaded on change, and you can change `@INC` only before the server has been started (in the startup file).

Nothing you do in your scripts and modules which are pulled in with require() after server startup will have any effect on `@INC`.

When you write:

```
use lib qw(foo/bar);
```

`@INC` is changed only for the time the code is being parsed and compiled. When that's done, `@INC` is reset to its original value.

To make sure that you have set `@INC` correctly, configure /perl-status location, fetch http://www.example.com/perl-status?inc and look at the bottom of the page, where the contents of `@INC` will be shown.

Notice the following trap:

While `"."` is in `@INC`, perl knows to require() files with pathnames given relative to the current (script) directory. After the script has been parsed, the server doesn't remember the path!

So you can end up with a broken entry in `%INC` like this:

```
$INC{bar.pl} eq "bar.pl"
```

If you want Apache::StatINC to reload your script--modify `@INC` at server startup, or use a full path in the require() call.

5.7.3 Using Apache::Reload

`Apache::Reload` comes as a drop-in replacement for `Apache::StatINC`. It provides extra functionality and better flexibility.

If you want `Apache::Reload` to check all the loaded modules on each request, you just add to *httpd.conf*:

```
PerlInitHandler Apache::Reload
```

If you want to reload only specific modules when these get changed, you have two ways to do that.

5.7.3.1 Register Modules Implicitly

The first way is to turn *Off* the `ReloadAll` variable, which is *On* by default

```
PerlInitHandler Apache::Reload
PerlSetVar ReloadAll Off
```

and add:

```
use Apache::Reload;
```

to every module that you want to be reloaded on change.

5.7.3.2 Register Modules Explicitly

The second way is to explicitly specify modules to be reloaded in *httpd.conf*:

```
PerlInitHandler Apache::Reload
PerlSetVar ReloadModules "My::Foo My::Bar Foo::Bar::Test"
```

Note that these are split on whitespace, but the module list **must** be in quotes, otherwise Apache tries to parse the parameter list.

You can register groups of modules using the metacharacter (`*`).

```
PerlSetVar ReloadModules "Foo::* Bar::*"
```

In the above example all modules starting with *Foo::* and *Bar::* will become registered. This features allows you to assign the whole project modules tree in one pattern.

5.7.3.3 Special "Touch" File

You can also set a file that you can touch(1) that causes the reloads to be performed. If you set this, and don't touch(1) the file, the reloads don't happen (no matter how have you registered the modules to be reloaded).

```
PerlSetVar ReloadTouchFile /tmp/reload_modules
```

Now when you're happy with your changes, simply go to the command line and type:

```
% touch /tmp/reload_modules
```

This feature is very convenient in a production server environment, but compared to a full restart, the benefits of preloaded modules memory sharing are lost, since each child will get it's own copy of the reloaded modules.

5.7.3.4 Caveats

This module might have a problem with reloading single modules that contain multiple packages that all use pseudo-hashes.

Also if you have modules loaded from directories which are not in `@INC`, `Apache::Reload` will fail to find the files, due the fact that `@INC` is reset to its original value even if it gets temporary modified in the script. The solution is to extend `@INC` at the server startup to include directories you load the files from which aren't in `@INC`.

For example, if you have a script which loads *MyTest.pm* from */home/stas/myproject*:

```
use lib qw(/home/stas/myproject);
require MyTest;
```

`Apache::Reload` won't find this file, unless you alter `@INC` in *startup.pl* (or *httpd.conf*):

```
startup.pl
----------
use lib qw(/home/stas/myproject);
```

and restart the server. Now the problem is solved.

5.7.3.5 Availability

This module is available from CPAN.

5.7.4 Configuration Files: Writing, Dynamically Updating and Reloading

Checking all the modules in `%INC` on every request can add a large overhead to server response times, and you certainly would not want the `Apache::StatINC` module to be enabled in your production site's configuration. But sometimes you want a configuration file reloaded when it is updated, without restarting

the server.

This is an especially important feature if for example you have a person that is allowed to modify some of the tool configuration, but for security reasons it's undesirable for him to telnet to the server to restart it.

5.7.4.1 Writing Configuration Files

Since we are talking about configuration files, I would like to show you some good and bad approaches to configuration file writing.

If you have a configuration file of just a few variables, it doesn't really matter how you do it. But generally this is not the case. Configuration files tend to grow as a project grows. It's very relevant to projects that generate HTML files, since they tend to demand many easily configurable parameters, like headers, footers, colors and so on.

So let's start with the approach that is most often taken by CGI scripts writers. All configuration variables are defined in a separate file.

For example:

```
$cgi_dir = "/home/httpd/perl";
$cgi_url = "/perl";
$docs_dir = "/home/httpd/docs";
$docs_url = "/";
$img_dir = "/home/httpd/docs/images";
$img_url = "/images";
... many more config params here ...
$color_hint   = "#777777";
$color_warn   = "#990066";
$color_normal = "#000000";
```

The `use strict;` pragma demands that all the variables be declared. When we want to use these variables in a mod_perl script we must declare them with `use vars` in the script. (Under Perl v5.6.0 `our()` has replaced `use vars`.)

So we start the script with:

```
use strict;
use vars qw($cgi_dir $cgi_url $docs_dir $docs_url
            ... many more config params here ....
            $color_hint  $color_warn $color_normal
           );
```

It is a nightmare to maintain such a script, especially if not all the features have been coded yet. You have to keep adding and removing variable names. But that's not a big deal.

Since we want our code clean, we start the configuration file with `use strict;` as well, so we have to list the variables with `use vars` pragma here as well. A second list of variables to maintain.

If you have many scripts, you may get collisions between configuration files. One of the best solutions is to declare packages, with unique names of course. For example for our configuration file we might declare the following package name:

```
package My::Config;
```

The moment you add a package declaration and think that you are done, you realize that the nightmare has just begun. When you have declared the package, you cannot just require() the file and use the variables, since they now belong to a different package. So you have either to modify all your scripts to use a fully qualified notation like `$My::Config::cgi_url` instead of just `$cgi_url` or to import the needed variables into any script that is going to use them.

Since you don't want to do the extra typing to make the variables fully qualified, you'd go for importing approach. But your configuration package has to export them first. That means that you have to list all the variables again and now you have to keep at least three variable lists updated when you make some changes in the naming of the configuration variables. And that's when you have only one script that uses the configuration file, in the general case you have many of them. So now our example configuration file looks like this:

```
package My::Config;
use strict;

BEGIN {
  use Exporter ();

  @My::HTML::ISA       = qw(Exporter);
  @My::HTML::EXPORT    = qw();
  @My::HTML::EXPORT_OK = qw($cgi_dir $cgi_url $docs_dir $docs_url
                            ... many more config params here ....
                            $color_hint $color_warn $color_normal);
}

use vars qw($cgi_dir $cgi_url $docs_dir $docs_url
            ... many more config params here ....
            $color_hint  $color_warn $color_normal
           );

$cgi_dir = "/home/httpd/perl";
$cgi_url = "/perl";
$docs_dir = "/home/httpd/docs";
$docs_url = "/";
$img_dir = "/home/httpd/docs/images";
$img_url = "/images";
... many more config params here ...
$color_hint   = "#777777";
$color_warn   = "#990066";
$color_normal = "#000000";
```

And in the code:

```
use strict;
use My::Config qw($cgi_dir $cgi_url $docs_dir $docs_url
                  ... many more config params here ....
                  $color_hint  $color_warn $color_normal
                 );
use vars       qw($cgi_dir $cgi_url $docs_dir $docs_url
                  ... many more config params here ....
                  $color_hint  $color_warn $color_normal
                 );
```

This approach is especially bad in the context of mod_perl, since exported variables add a memory overhead. The more variables exported the more memory you use. If we multiply this overhead by the number of servers we are going to run, we get a pretty big number which could be used to run a few more servers instead.

As a matter of fact things aren't so bad. You can group your variables, and call the groups by special names called tags, which can later be used as arguments to the import() or use() calls. You are probably familiar with:

```
use CGI qw(:standard :html);
```

We can implement this quite easily, with the help of export_ok_tags() from `Exporter`. For example:

```
BEGIN {
  use Exporter ();
  use vars qw( @ISA @EXPORT @EXPORT_OK %EXPORT_TAGS);
  @ISA         = qw(Exporter);
  @EXPORT      = qw();
  @EXPORT_OK   = qw();

  %EXPORT_TAGS = (
    vars => [qw($fname $lname)],
    subs => [qw(reread_conf untaint_path)],
  );
  Exporter::export_ok_tags('vars');
  Exporter::export_ok_tags('subs');
}
```

You export subroutines exactly like variables, since what's actually being exported is a symbol. The definition of these subroutines is not shown here.

Notice that we didn't use export_tags(), as it exports the variables automatically without the user asking for them in first place, which is considered bad style. If a module automatically exports variables with export_tags() you can stop this by not exporting at all:

```
use My::Config ();
```

In your code you can now write:

```
use My::Config qw(:subs :vars);
```

Groups of group tags:

The `:all` tag from `CGI.pm` is a group tag of all other groups. It will require a little more effort to implement, but you can always save time by looking at the solution in `CGI.pm`'s code. It's just a matter of a little code to expand all the groups recursively.

After going through the pain of maintaining a list of variables in a big project with a huge configuration file (more than 100 variables) and many files actually using them, I came up with a much simpler solution: keeping all the variables in a single hash, which is built from references to other anonymous scalars, arrays and hashes.

Now my configuration file looks like this:

```
package My::Config;
use strict;

BEGIN {
  use Exporter ();

  @My::Config::ISA       = qw(Exporter);
  @My::Config::EXPORT    = qw();
  @My::Config::EXPORT_OK = qw(%c);
}

use vars qw(%c);

%c = (
   dir => {
       cgi  => "/home/httpd/perl",
       docs => "/home/httpd/docs",
       img  => "/home/httpd/docs/images",
   },
   url => {
       cgi  => "/perl",
       docs => "/",
       img  => "/images",
   },
   color => {
       hint   => "#777777",
       warn   => "#990066",
       normal => "#000000",
   },
  );
```

Good perl style suggests keeping a comma at the end of lists. That's because additional items tend to be added to the end of the list. If you keep that last comma in place, you don't have to remember to add one when you add a new item.

So now the script looks like this:

```
use strict;
use My::Config qw(%c);
use vars       qw(%c)
print "Content-type: text/plain\r\n\r\n";
print "My url docs root: $c{url}{docs}\n";
```

Do you see the difference? The whole mess has gone, there is only one variable to worry about.

There is one small downside to taking this approach: auto-vivification. For example, if we wrote `$c{url}{doc}` by mistake, perl would silently create this element for us with the value *undef*. When we `use strict;` Perl will tell us about any misspelling of this kind for a simple scalar, but this check is not performed for hash elements. This puts the onus of responsibility back on us since we must take greater care. A possible solution to this is to use pseudo-hashes, but they are still considered experimental so we won't cover them here.

The benefits of the hash approach are significant and we can make do even better. I would like to get rid of the `Exporter` stuff completely. I remove all the exporting code so my config file now looks like:

```
package My::Config;
use strict;
use vars qw(%c);

%c = (
   dir => {
       cgi  => "/home/httpd/perl",
       docs => "/home/httpd/docs",
       img  => "/home/httpd/docs/images",
   },
   url => {
       cgi  => "/perl",
       docs => "/",
       img  => "/images",
   },
   color => {
       hint   => "#777777",
       warn   => "#990066",
       normal => "#000000",
   },
  );
```

And the code:

```
use strict;
use My::Config ();
print "Content-type: text/plain\r\n\r\n";
print "My url docs root: $My::Config::c{url}{docs}\n";
```

Since we still want to save lots of typing, and since now we need to use a fully qualified notation like `$My::Config::c{url}{docs}`, let's use the magical Perl aliasing feature. I'll modify the code to be:

```
use strict;
use My::Config ();
use vars qw(%c);
*c = \%My::Config::c;
print "Content-type: text/plain\r\n\r\n";
print "My url docs root: $c{url}{docs}\n";
```

I have aliased the `*c` glob with `\%My::Config::c`, a reference to a hash. From now on, `%My::Config::c` and `%c` are the same hash and you can read from or modify either of them.

Just one last little point. Sometimes you see a lot of redundancy in the configuration variables, for example:

```
$cgi_dir  = "/home/httpd/perl";
$docs_dir = "/home/httpd/docs";
$img_dir  = "/home/httpd/docs/images";
```

Now if you want to move the base path `"/home/httpd"` into a new place, it demands lots of typing. Of course the solution is:

```
$base     = "/home/httpd";
$cgi_dir  = "$base/perl";
$docs_dir = "$base/docs";
$img_dir  = "$docs_dir/images";
```

You cannot do the same trick with a hash, since you cannot refer to its values before the definition is finished. So this wouldn't work:

```
%c =
  (
   base => "/home/httpd",
   dir => {
       cgi  => "$c{base}/perl",
       docs => "$c{base}/docs",
       img  => "$c{base}{docs}/images",
   },
  );
```

But nothing stops us from adding additional variables, which are lexically scoped with my (). The following code is correct.

```
my $base = "/home/httpd";
%c =
  (
   dir => {
       cgi  => "$base/perl",
       docs => "$base/docs",
       img  => "$base/docs/images",
   },
  );
```

You have just learned how to make configuration files easily maintainable, and how to save memory by avoiding the export of variables into a script's namespace.

5.7.4.2 Reloading Configuration Files

First, lets look at a simple case, when we just have to look after a simple configuration file like the one below. Imagine a script that tells you who is the patch pumpkin of the current Perl release.

Sidenote: *Pumpkin* A humorous term for the token (notional or real) that gives its possessor (the "pumpking" or the "pumpkineer") exclusive access to something, e.g. applying patches to a master copy of some source (for which the token is called the "patch pumpkin").

```
use CGI ();
use strict;

my $fname = "Larry";
my $lname = "Wall";
my $q = CGI->new;

print $q->header(-type=>'text/html');
print $q->p("$fname $lname holds the patch pumpkin" .
            "for this Perl release.");
```

The script has a hardcoded value for the name. It's very simple: initialize the CGI object, print the proper HTTP header and tell the world who is the current patch pumpkin.

When the patch pumpkin changes we don't want to modify the script. Therefore, we put the `$fname` and `$lname` variables into a configuration file.

```
$fname = "Gurusamy";
$lname = "Sarathy";
1;
```

Please note that there is no package declaration in the above file, so the code will be evaluated in the caller's package or in the `main::` package if none was declared. This means that the variables `$fname` and `$lname` will override (or initialize if they weren't yet) the variables with the same names in the caller's namespace. This works for global variables only--you cannot update variables defined lexically (with my ()) using this technique.

You have started the server and everything is working properly. After a while you decide to modify the configuration. How do you let your running server know that the configuration was modified without restarting it? Remember we are in production and server restarting can be quite expensive for us. One of the simplest solutions is to poll the file's modification time by calling stat() before the script starts to do real work. If we see that the file was updated, we force a reconfiguration of the variables located in this file. We will call the function that reloads the configuration reread_conf() and have it accept a single argument, which is the relative path to the configuration file.

`Apache::Registry` calls a chdir() to the script's directory before it starts the script's execution. So if your CGI script is invoked under the `Apache::Registry` handler you can put the configuration file in the same directory as the script. Alternatively you can put the file in a directory below that and use a path relative to the script directory. You have to make sure that the file will be found, somehow. Be aware that do() searches the libraries in the directories in `@INC`.

```
use vars qw(%MODIFIED);
sub reread_conf{
    my $file = shift;
    return unless defined $file;
    return unless -e $file and -r _;
    my $mod = -M _;
    unless (exists $MODIFIED{$file} and $MODIFIED{$file} == $mod) {
        my $result;
        unless ($result = do $file) {
            warn "couldn't parse $file: $@" if $@;
            warn "couldn't do $file: $!"    unless defined $result;
            warn "couldn't run $file"       unless         $result;
        }
        $MODIFIED{$file} = $mod; # Update the MODIFICATION times
    }
} # end of reread_conf
```

Notice that we use the == comparison operator when checking file's modification timestamp, because all we want to know whether the file was changed or not.

When the require(), use() and do() operators successfully return, the file that was passed as an argument is inserted into %INC (the key is the name of the file and the value the path to it). Specifically, when Perl sees require() or use() in the code, it first tests %INC to see whether the file is already there and thus loaded. If the test returns true, Perl saves the overhead of code re-reading and re-compiling; however calling do() will (re)load regardless.

You generally don't notice with plain perl scripts, but in mod_perl it's used all the time; after the first request served by a process all the files loaded by require() stay in memory. If the file is preloaded at server startup, even the first request doesn't have the loading overhead.

We use do() to reload the code in this file and not require() because while do() behaves almost identically to require(), it reloads the file unconditionally. If do() cannot read the file, it returns undef and sets $! to report the error. If do() can read the file but cannot compile it, it returns undef and sets an error message in $@. If the file is successfully compiled, do() returns the value of the last expression evaluated.

The configuration file can be broken if someone has incorrectly modified it. We don't want the whole service that uses that file to be broken, just because of that. We trap the possible failure to do() the file and ignore the changes, by the resetting the modification time. If do() fails to load the file it might be a good idea to send an email to the system administrator about the problem.

Notice however, that since do() updates %INC like require() does, if you are using Apache::StatINC it will attempt to reload this file before the reread_conf() call. So if the file wouldn't compile, the request will be aborted. Apache::StatINC shouldn't be used in production (because it slows things down by stat()'ing all the files listed in %INC) so this shouldn't be a problem.

Note that we assume that the entire purpose of this function is to reload the configuration if it was changed. This is fail-safe, because if something goes wrong we just return without modifying the server configuration. The script should not be used to initialize the variables on its first invocation. To do that, you would need to replace each occurrence of return() and warn() with die(). If you do that, take a look at the section "Redirecting Errors to the Client instead of error_log".

I used the above approach when I had a huge configuration file that was loaded only at server startup, and another little configuration file that included only a few variables that could be updated by hand or through the web interface. Those variables were initialized in the main configuration file. If the webmaster breaks the syntax of this dynamic file while updating it by hand, it won't affect the main (write-protected) configuration file and so stop the proper execution of the programs. Soon we will see a simple web interface which allows us to modify the configuration file without actually breaking it.

A sample script using the presented subroutine would be:

```
use vars qw(%MODIFIED $fname $lname);
use CGI ();
use strict;

my $q = CGI->new;
print $q->header(-type=>'text/plain');
my $config_file = "./config.pl";
reread_conf($config_file);
print $q->p("$fname $lname holds the patch pumpkin" .
            "for this Perl release.");

sub reread_conf{
    my $file = shift;
    return unless defined $file;
    return unless -e $file and -r _;
    my $mod = -M _;
    unless ($MODIFIED{$file} and $MODIFIED{$file} == $mod) {
        my $result;
        unless ($result = do $file) {
            warn "couldn't parse $file: $@" if $@;
            warn "couldn't do $file: $!"    unless defined $result;
            warn "couldn't run $file"       unless $result;
        }
        $MODIFIED{$file} = $mod; # Update the MODIFICATION times
    }
} # end of reread_conf
```

Remember that you should be using `(stat $file)[9]` instead of `-M $file` if you are modifying the `$^T` variable. In some of my scripts, I reset `$^T` to the time of the script invocation with `"$^T = time()"`. That way I can perform `-M` and the similar (`-A`, `-C`) file status tests relative to the script invocation time, and not the time the process was started.

If your configuration file is more sophisticated and it declares a package and exports variables, the above code will work just as well. Even if you think that you will have to import() variables again, when do() recompiles the script the originally imported variables get updated with the values from the reloaded code.

5.7.4.3 Dynamically updating configuration files

The CGI script below allows a system administrator to dynamically update a configuration file through the web interface. Combining this with the code we have just seen to reload the modified files, you get a system which is dynamically reconfigurable without needing to restart the server. Configuration can be performed from any machine having just a web interface (a simple browser connected to the Internet).

Let's say you have a configuration file like this:

```
package MainConfig;

use strict;
use vars qw(%c);

%c = (
      name     => "Larry Wall",
      release  => "5.000",
      comments => "Adding more ways to do the same thing :)",

      other    => "More config values",

      hash     => { foo => "ouch",
                    bar => "geez",
                  },

      array    => [qw( a b c)],

     );
```

You want to make the variables `name`, `release` and `comments` dynamically configurable. You want to have a web interface with an input form that allows you to modify these variables. Once modified you want to update the configuration file and propagate the changes to all the currently running processes. Quite a simple task.

Let's look at the main stages of the implementation. Create a form with preset current values of the variables. Let the administrator modify it and submit the changes. Validate the submitted information (numeric fields should carry numbers, literals--words, etc). Update the configuration file. Update the modified value in the memory of the current process. Present the form as before but with updated fields if any.

The only part that seems to be complicated to implement is a configuration file update, for a couple of reasons. If updating the file breaks it, the whole service won't work. If the file is very big and includes comments and complex data structures, parsing the file can be quite a challenge.

So let's simplify the task. If all we want is to update a few variables, why don't we create a tiny configuration file with just those variables? It can be modified through the web interface and overwritten each time there is something to be changed. This way we don't have to parse the file before updating it. If the main configuration file is changed we don't care, we don't depend on it any more.

The dynamically updated variables are duplicated, they will be in the main file and in the dynamic file. We do this to simplify maintenance. When a new release is installed the dynamic configuration file won't exist at all. It will be created only after the first update. As we just saw, the only change in the main code is to add a snippet to load this file if it exists and was changed.

This additional code must be executed after the main configuration file has been loaded. That way the updated variables will override the default values in the main file.

META: extend on the comments:

```
# remember to run this code in taint mode

use strict;
use vars qw($q %c $dynamic_config_file %vars_to_change %validation_rules);

use CGI ();

use lib qw(.);
use MainConfig ();
*c = \%MainConfig::c;

$dynamic_config_file = "./config.pl";

# load the dynamic configuration file if it exists, and override the
# default values from the main configuration file
do $dynamic_config_file if -e $dynamic_config_file and -r _;

# fields that can be changed and their titles
%vars_to_change =
  (
   'name'     => "Patch Pumpkin's Name",
   'release'  => "Current Perl Release",
   'comments' => "Release Comments",
  );

%validation_rules =
  (
   'name'     => sub { $_[0] =~ /^[\w\s\.]+$/;   },
   'release'  => sub { $_[0] =~ /^\d+\.[\d_]+$/; },
   'comments' => sub { 1;                        },
  );

$q = CGI->new;
print $q->header(-type=>'text/html'),
      $q->start_html();

my %updates = ();

# We always rewrite the dynamic config file, so we want all the
# vars to be passed, but to save time we will only do checking

# of vars that were changed.  The rest will be retrieved from
# the 'prev_foo' values.
foreach (keys %vars_to_change) {
  # copy var so we can modify it
  my $new_val = $q->param($_) || '';

  # strip a possible ^M char (DOS/WIN)
  $new_val =~ s/\cM//g;

  # push to hash if was changed
  $updates{$_} = $new_val
    if defined $q->param("prev_".$_)
       and $new_val ne $q->param("prev_".$_);
}
```

```
# Note that we cannot trust the previous values of the variables
# since they were presented to the user as hidden form variables,
# and the user can mangle those. We don't care: it cannot do any
# damage, as we verify each variable by rules which we define.

# Process if there is something to process. Will be not called if
# it's invoked a first time to display the form or when the form
# was submitted but the values weren't modified (we know that by
# comparing with the previous values of the variables, which are
# the hidden fields in the form)

# process and update the values if valid
process_change_config(%updates) if %updates;

# print the update form
conf_modification_form();

# update the config file but first validate that the values are correct ones
##########################
sub process_change_config{
  my %updates = @_;

    # we will list here all the malformatted vars
  my %malformatted = ();

  print $q->b("Trying to validate these values<BR>");
  foreach (keys %updates) {
    print "<DT><B>$_</B> => <PRE>$updates{$_}</PRE>";

    # now we have to handle each var to be changed very carefully
    # since this file goes immediately into production!
    $malformatted{$_} = delete $updates{$_}
      unless $validation_rules{$_}->($updates{$_});

  } # end of foreach (keys %updates)

  # print warnings if there are any invalid changes
  print $q->hr,
    $q->p($q->b(qq{Warning! These variables were changed
         but found malformed, thus the original
         values will be preserved.})
   ),
    join(",<BR>",
   map { $q->b($vars_to_change{$_}) . " : $malformatted{$_}\n"
       } keys %malformatted)
     if %malformatted;

  # Now complete the vars that weren't changed from the
  # $q->param('prev_var') values
  map { $updates{$_} = $q->param('prev_'.$_) unless exists $updates{$_}
      } keys %vars_to_change;

  # Now we have all the data that should be written into the dynamic
  # config file

    # escape single quotes "'" while creating a file
```

```
    my $content = join "\n",
      map { $updates{$_} =~ s/([’\\])/\\$1/g;
      ’$c{’ . $_ . "}  =  ’" . $updates{$_} . "’;\n"
    } keys %updates;

      # now add ’1;’ to make require() happy
    $content .= "\n1;";

      # keep the dummy result in $res so it won’t complain

    eval {my $res = $content};
    if ($@) {
      print qq{Warning! Something went wrong with config file
         generation!<P> The error was : <BR><PRE>$@</PRE>};
      return;
    }

    print $q->hr;

      # overwrite the dynamic config file
    use Symbol ();
    my $fh = Symbol::gensym();
    open $fh, ">$dynamic_config_file.bak"
      or die "Can’t open $dynamic_config_file.bak for writing :$! \n";
    flock $fh,2; # exclusive lock
    seek $fh,0,0;   # rewind to the start
    truncate $fh, 0; # the file might shrink!
       print $fh $content;
    close $fh;

      # OK, now we make a real file
    rename "$dynamic_config_file.bak",$dynamic_config_file
      or die "Failed to rename: $!";

      # rerun it to update variables in the current process! Note that
      # it won’t update the variables in other processes. Special
      # code that watches the timestamps on the config file will do this
      # work for each process. Since the next invocation will update the
      # configuration anyway, why do we need to load it here? The reason
      # is simple: we are going to fill the form’s input fields with
      # the updated data.
    do $dynamic_config_file;

  } # end sub process_change_config

  ############################
  sub conf_modification_form{

    print $q->center($q->h3("Update Form"));

    print $q->hr,
      $q->p(qq{This form allows you to dynamically update the current
         configuration. You don\’t need to restart the server in
         order for changes to take an effect}
           );

      # set the previous settings in the form’s hidden fields, so we
```

```
    # know whether we have to do some changes or not
  map {$q->param("prev_$_",$c{$_}) } keys %vars_to_change;

    # rows for the table, go into the form
  my @configs = ();

    # prepare one textfield entries
  push @configs,
    map {
      $q->td(
       $q->b("$vars_to_change{$_}:"),
      ),
      $q->td(
       $q->textfield(-name      => $_,
             -default   => $c{$_},
             -override  => 1,
             -size      => 20,
             -maxlength => 50,
            )
      ),
    } qw(name release);

    # prepare multiline textarea entries
  push @configs,
    map {
      $q->td(
       $q->b("$vars_to_change{$_}:"),
      ),
      $q->td(
       $q->textarea(-name    => $_,
            -default => $c{$_},
            -override  => 1,
            -rows    => 10,

            -columns => 50,
            -wrap    => "HARD",
            )
      ),
    } qw(comments);

  print $q->startform('POST',$q->url),"\n",
  $q->center($q->table(map {$q->Tr($_),"\n",} @configs),
        $q->submit('','Update!'),"\n",
       ),
  map ({$q->hidden("prev_".$_, $q->param("prev_".$_))."\n" }
       keys %vars_to_change), # hidden previous values
  $q->br,"\n",
  $q->endform,"\n",
  $q->hr,"\n",
  $q->end_html;

} # end sub conf_modification_form
```

Once updated the script generates a file like:

```
$c{release}  =  '5.6';

$c{name}  =  'Gurusamy Sarathy';

$c{comments}  =  'Perl rules the world!';

1;
```

5.7.5 Reloading handlers

If you want to reload a perlhandler on each invocation, the following trick will do it:

```
PerlHandler "sub { do 'MyTest.pm'; MyTest::handler(shift) }"
```

do() reloads `MyTest.pm` on every request.

5.8 Name collisions with Modules and libs

This section requires an in-depth understanding of use(), require(), do(), %INC and @INC .

To make things clear before we go into details: each child process has its own `%INC` hash which is used to store information about its compiled modules. The keys of the hash are the names of the modules and files passed as arguments to require() and use(). The values are the full or relative paths to these modules and files.

Suppose we have `my-lib.pl` and `MyModule.pm` both located at `/home/httpd/perl/my/`.

- `/home/httpd/perl/my/` is in `@INC` at server startup.

  ```
  require "my-lib.pl";
  use MyModule;
  print $INC{"my-lib.pl"},"\n";
  print $INC{"MyModule.pm"},"\n";
  ```

  prints:

  ```
  /home/httpd/perl/my/my-lib.pl
  /home/httpd/perl/my/MyModule.pm
  ```

  Adding `use lib`:

  ```
  use lib qw(.);
  require "my-lib.pl";
  use MyModule;
  print $INC{"my-lib.pl"},"\n";
  print $INC{"MyModule.pm"},"\n";
  ```

  prints:

```
my-lib.pl
MyModule.pm
```

- `/home/httpd/perl/my/` isn't in `@INC` at server startup.

  ```
  require "my-lib.pl";
  use MyModule;
  print $INC{"my-lib.pl"},"\n";
  print $INC{"MyModule.pm"},"\n";
  ```

  wouldn't work, since perl cannot find the modules.

  Adding `use lib`:

  ```
  use lib qw(.);
  require "my-lib.pl";
  use MyModule;
  print $INC{"my-lib.pl"},"\n";
  print $INC{"MyModule.pm"},"\n";
  ```

  prints:

  ```
  my-lib.pl
  MyModule.pm
  ```

Let's look at three scripts with faults related to name space. For the following discussion we will consider just one individual child process.

- **Scenario 1**

  First, You can't have two identical module names running on the same server! Only the first one found in a use() or require() statement will be compiled into the package, the request for the other module will be skipped, since the server will think that it's already compiled. This is a direct result of using `%INC`, which has keys equal to the names of the modules. Two identical names will refer to the same key in the hash. (Refer to the section 'Looking inside the server' to find out how you can know what is loaded and where.)

  So if you have two different `Foo` modules in two different directories and two scripts `script1.pl` and `script2.pl`, placed like this:

  ```
  ./tool1/Foo.pm
  ./tool1/tool1.pl
  ./tool2/Foo.pm
  ./tool2/tool2.pl
  ```

  Where some sample code could be:

  ```
  ./tool1/tool1.pl
  ----------------
  use Foo;
  print "Content-type: text/plain\r\n\r\n";
  print "I'm Script number One\n";
  foo();
  ```

```
./tool1/Foo.pm
--------------
sub foo{
  print "<B>I'm Tool Number One!</B>\n";
}
1;

./tool2/tool2.pl
----------------
use Foo;
print "Content-type: text/plain\r\n\r\n";
print "I'm Script number Two\n";
foo();

./tool2/Foo.pm
--------------
sub foo{
  print "<B>I'm Tool Number Two!</B>\n";
}
1;
```

Both scripts call `use Foo;`. Only the first one called will know about `Foo`. When you call the second script it will not know about `Foo` at all--it's like you've forgotten to write `use Foo;`. Run the server in single server mode to detect this kind of bug immediately.

You will see the following in the error_log file:

```
Undefined subroutine
&Apache::ROOT::perl::tool2::tool2_2epl::foo called at
/home/httpd/perl/tool2/tool2.pl line 4.
```

- **Scenario 2**

  If the files do not declare a package, the above is true for libraries (i.e. *my-lib.pl"*) you require() as well:

  Suppose that you have a directory structure like this:

```
./tool1/config.pl
./tool1/tool1.pl
./tool2/config.pl
./tool2/tool2.pl
```

  and both scripts contain:

```
use lib qw(.);
require "config.pl";
```

  while *./tool1/config.pl* can be something like this:

```
$foo = 0;
1;
```

and *./tool2/config.pl*:

```
$foo = 1;
1;
```

The second scenario is not different from the first, there is almost no difference between use() and require() if you don't have to import some symbols into a calling script. Only the first script served will actually do the require(), for the same reason as the example above. `%INC` already includes the key *"config.pl"*!

- **Scenario 3**

  It is interesting that the following scenario will fail too!

  ```
  ./tool/config.pl
  ./tool/tool1.pl
  ./tool/tool2.pl
  ```

  where `tool1.pl` and `tool2.pl` both require() the *same* `config.pl`.

There are three solutions for this:

- **Solution 1**

  The first two faulty scenarios can be solved by placing your library modules in a subdirectory structure so that they have different path prefixes. The file system layout will be something like:

  ```
  ./tool1/Tool1/Foo.pm
  ./tool1/tool1.pl
  ./tool2/Tool2/Foo.pm
  ./tool2/tool2.pl
  ```

  And modify the scripts:

  ```
  use Tool1::Foo;
  use Tool2::Foo;
  ```

  For require() (scenario number 2) use the following:

  ```
  ./tool1/tool1-lib/config.pl
  ./tool1/tool1.pl
  ./tool2/tool2-lib/config.pl
  ./tool2/tool2.pl
  ```

  And each script contains respectively:

  ```
  use lib qw(.);
  require "tool1-lib/config.pl";

  use lib qw(.);
  require "tool2-lib/config.pl";
  ```

This solution isn't good, since while it might work for you now, if you add another script that wants to use the same module or `config.pl` file, it would fail as we saw in the third scenario.

Let's see some better solutions.

- **Solution 2**

  Another option is to use a full path to the script, so it will be used as a key in `%INC`;

  ```
  require "/full/path/to/the/config.pl";
  ```

  This solution solves the problem of the first two scenarios. I was surprised that it worked for the third scenario as well!

  With this solution you lose some portability. If you move the tool around in the file system you will have to change the base directory or write some additional script that will automatically update the hardcoded path after it was moved. Of course you will have to remember to invoke it.

- **Solution 3**

  Make sure you read all of this solution.

  Declare a package name in the required files! It should be unique in relation to the rest of the package names you use. `%INC` will then use the unique package name for the key. It's a good idea to use at least two-level package names for your private modules, e.g. `MyProject::Carp` and not `Carp`, since the latter will collide with an existing standard package. Even though a package may not exist in the standard distribution now, a package may come along in a later distribution which collides with a name you've chosen. Using a two part package name will help avoid this problem.

  Even a better approach is to use three level naming, like `CompanyName::Project-Name::Module`, which is most unlikely to have conflicts with later Perl releases. Foresee problems like this and save yourself future trouble.

  What are the implications of package declaration?

  Without package declarations, it is very convenient to use() or require() files because all the variables and subroutines are part of the `main::` package. Any of them can be used as if they are part of the main script. With package declarations things are more awkward. You have to use the `Package::function()` method to call a subroutine from `Package` and to access a global variable `$foo` inside the same package you have to write `$Package::foo`.

  Lexically defined variables, those declared with my () inside `Package` will be inaccessible from outside the package.

  You can leave your scripts unchanged if you import the names of the global variables and subroutines into the namespace of package `main::` like this:

```
use Module qw(:mysubs sub_b $var1 :myvars);
```

You can export both subroutines and global variables. Note however that this method has the disadvantage of consuming more memory for the current process.

See `perldoc Exporter` for information about exporting other variables and symbols.

This completely covers the third scenario. When you use different module names in package declarations, as explained above, you cover the first two as well.

- **A Hack**

  The following solution should be used only as a short term bandaid. You can force reloading of the modules by either fiddling with `%INC` or replacing use() and require() calls with do().

  If you delete the module entry from the `%INC` hash, before calling require() or use() the module will be loaded and compiled again. For example:

```
./project/runA.pl
-----------------
BEGIN {
    delete $INC{"MyConfig.pm"};
}
use lib qw(.);
use MyConfig;
print "Content-type: text/plain\n\n";
print "Script A\n";
print "Inside project: ", project_name();
```

  Apply the same fix to *runB.pl*.

  Another alternative is to force module reload via do():

```
./project/runA.pl
-----------------
use lib qw(.);
do "MyConfig.pm";
print "Content-type: text/plain\n\n";
print "Script B\n";
print "Inside project: ", project_name();
```

  Apply the same fix to *runB.pl*.

  If you needed to import() something from the loaded module, call the import() method explicitly. For example if you had:

```
use MyConfig qw(foo bar);
```

  now the code will look as:

```
do "MyConfig.pm";
MyConfig->import(qw(foo bar));
```

Both presented solutions are ineffective, since the modules in question will be reloaded on each request, slowing down the response times. Therefore use these only when a very quick fix is needed and provide one of the more robust solutions discussed in the previous sections.

See also the `perlmodlib` and `perlmod` manpages.

From the above discussion it should be clear that you cannot run development and production versions of the tools using the same apache server! You have to run a separate server for each. They can be on the same machine, but the servers will use different ports.

5.9 More package name related issues

If you have the following:

```
PerlHandler Apache::Work::Foo
PerlHandler Apache::Work::Foo::Bar
```

And you make a request that pulls in `Apache/Work/Foo/Bar.pm` first, then the `Apache::Work::Foo` package gets defined, so mod_perl does not try to pull in `Apache/Work/Foo.pm`

5.10 __END__ and __DATA__ tokens

`Apache::Registry` scripts cannot contain `__END__` or `__DATA__` tokens.

Why? Because `Apache::Registry` scripts are being wrapped into a subroutine called `handler`, like the script at URI `/perl/test.pl`:

```
print "Content-type: text/plain\r\n\r\n";
print "Hi";
```

When the script is being executed under `Apache::Registry` handler, it actually becomes:

```
package Apache::ROOT::perl::test_2epl;
use Apache qw(exit);
sub handler {
  print "Content-type: text/plain\r\n\r\n";
  print "Hi";
}
```

So if you happen to put an `__END__` tag, like:

```
print "Content-type: text/plain\r\n\r\n";
print "Hi";
__END__
Some text that wouldn't be normally executed
```

it will be turned into:

```
package Apache::ROOT::perl::test_2epl;
use Apache qw(exit);
sub handler {
  print "Content-type: text/plain\r\n\r\n";
  print "Hi";
  __END__
  Some text that wouldn't be normally executed
}
```

and you try to execute this script, you will receive the following error:

```
Missing right bracket at .... line 4, at end of line
```

Perl cuts everything after the __END__ tag. The same applies to the __DATA__ tag.

Also, remember that whatever applies to `Apache::Registry` scripts, in most cases applies to `Apache::PerlRun` scripts.

5.11 Output from system calls

The output of `system()`, `exec()`, and `open(PIPE,"|program")` calls will not be sent to the browser unless your Perl was configured with `sfio`.

You can use backticks as a possible workaround:

```
print ‘command here‘;
```

But you're throwing performance out the window either way. It's best not to fork at all if you can avoid it. See the "Forking or Executing subprocesses from mod_perl" section to learn about implications of forking.

Also read about Apache::SubProcess for overridden system() and exec() implementations that work with mod_perl.

5.12 Using format() and write()

The interface to filehandles which are linked to variables with Perl's tie() function is not yet complete. The format() and write() functions are missing. If you configure Perl with `sfio`, write() and format() should work just fine.

Otherwise you could use sprintf() to replace format(): `##.##` becomes `%2.2f` and `####.##` becomes `%4.2f`.

Pad all strings with (" " x 80) before using, and set their length with: %.25s for a max 25 char string. Or prefix the string with (" " x 80) for right-justifying.

Another alternative is to use the `Text::Reform` module.

5.13 Terminating requests and processes, the exit() and child_terminate() functions

Perl's `exit()` built-in function (all versions prior to 5.6) cannot be used in mod_perl scripts. Calling it causes the mod_perl process to exit (which defeats the purpose of using mod_perl). The `Apache::exit()` function should be used instead. Starting from Perl version 5.6 mod_perl will override exit() behind the scenes, using `CORE::GLOBAL::`, a new *magical* package.

You might start your scripts by overriding the exit() subroutine (if you use `Apache::exit()` directly, you will have a problem testing the script from the shell, unless you put `use Apache ();` into your code.) I use the following code:

```
use constant IS_MODPERL => $ENV{MOD_PERL};
use subs qw(exit);
# Select the correct exit function
*exit = IS_MODPERL ? \&Apache::exit : sub { CORE::exit };
```

Now the correct `exit()` is always chosen, whether the script is running under mod_perl, ordinary CGI or from the shell. Notice that since we are using the constant pragma, there is no runtime overhead to select one of the code references, since `IS_MODPERL` constant is folded, that block is optimized away at compile time outside of mod_perl.

Note that if you run the script under `Apache::Registry`, **The Apache function `exit()` overrides the Perl core built-in function**. While you see exit() listed in the `@EXPORT_OK` list of the Apache package, `Apache::Registry` does something you don't see and imports this function for you. This means that if your script is running under the `Apache::Registry` handler you don't have to worry about exit(). The same applies to `Apache::PerlRun`.

If you use `CORE::exit()` in scripts running under mod_perl, the child will exit, but neither a proper exit nor logging will happen on the way. `CORE::exit()` cuts off the server's legs.

Note that `Apache::exit(Apache::Constants::DONE)` will cause the server to exit gracefully, completing the logging functions and protocol requirements etc. (Apache::Constants::DONE == -2, Apache::Constants::OK == 0.)

If you need to shut down the child cleanly after the request was completed, use the `$r->child_terminate` method. You can call it anywhere in the code, and not just at the "end". This sets the value of the `MaxRequestsPerChild` configuration variable to 1 and clears the `keepalive` flag. After the request is serviced, the current connection is broken, because of the `keepalive` flag, and the parent tells the child to cleanly quit, because `MaxRequestsPerChild` is smaller than the number of requests served.

In an `Apache::Registry` script you would do:

```
Apache->request->child_terminate;
```

or in httpd.conf:

```
PerlFixupHandler "sub { shift->child_terminate }"
```

You would want to use the latter example only if you wanted the child to terminate every time the registered handler is called. Probably this is not what you want.

Even if you don't need to call child_terminate() at the end of the request if you want the process to quit afterwards, here is an example of assigning the postprocessing handler. You might do this if you wanted to execute your own code a moment before the process quits.

```
my $r = shift;
$r->post_connection(\&exit_child);
sub exit_child{
   # some logic here if needed
  $r->child_terminate;
}
```

The above is the code that is used by the `Apache::SizeLimit` module which terminates processes that grow bigger than a value you choose.

Apache::GTopLimit (based on *libgtop* and `GTop.pm`) is a similar module. It does the same thing, plus you can configure it to terminate processes when their shared memory shrinks below some specified size.

5.14 die() and mod_perl

When you write:

```
open FILE, "foo" or die "Cannot open foo file for reading: $!";
```

in a perl script and execute it--the script would die() if it is unable to open the file, by aborting the script execution, printing the death reason and quitting the Perl interpreter.

You will hardly find a properly written Perl script that doesn't have at least one die() statement in it, if it has to cope with system calls and the like.

A CGI script running under mod_cgi exits on its completion. The Perl interperter exits as well. So it doesn't really matter whether the interpreter quits because the script died by natural death (when the last statement was executed) or was aborted by a die() statement.

In mod_perl we don't want the interpreter to quit. We already know that when the script completes its chores the interpeter won't quit. There is no reason why it should quit when the script has stopped because of die(). As a result calling die() won't quit the process.

And this is how it works--when the die() gets triggered, it's mod_perl's `$SIG{__DIE__}` handler that logs the error message and calls Apache::exit() instead of CORE::die(). Thus the script stops, but the process doesn't quit.

Here is an example of such trapping code, although it isn't the real code:

```
$SIG{__DIE__} = sub { print STDERR @_; Apache::exit(); }
```

5.15 Return Codes

`Apache::Registry` normally assumes a return code of `OK` (`200`). If you want to send another return code, use `$r->status()`:

```
use Apache::Constants qw(NOT_FOUND);
$r->status(NOT_FOUND);
```

Of course if you do that, you don't have to call `$r->send_http_header()` (assuming that you have `PerlSendHeader Off`).

5.16 Testing the Code from the Shell

Your CGI scripts will **not** yet run from the command line unless you use `CGI::Switch` or `CGI.pm` and have Perl 5.004 or later. They must not make any direct calls to Apache's Perl API methods.

5.17 I/O is different

If you are using Perl 5.004 or later, most CGI scripts can run under mod_perl untouched.

If you're using 5.003, Perl's built-in `read()` and `print()` functions do not work as they do under CGI. If you're using `CGI.pm`, use `$query->print` instead of plain ol' `print()`.

5.18 STDIN, STDOUT and STDERR streams

In mod_perl both `STDIN` and `STDOUT` are tied to the socket the request came from. Because the C level STDOUT is not hooked up to the client, you can re-open the STDOUT filehandler using tie(). For example if you want to dup an STDOUT filehandler and for the code to work with mod_perl and without it, the following example will do:

```
use constant IS_MODPERL => $ENV{MOD_PERL};
if (IS_MODPERL) {
  tie *OUT, 'Apache';
} else {
  open (OUT, ">-");
}
```

Note that `OUT` was picked just as an example -- there is nothing special about it. If you are looking to redirect the STDOUT stream into a scalar, see the Redirecting STDOUT into a String section.

`STDERR` is tied to the file defined by the `ErrorLog` directive.

5.19 Redirecting STDOUT into a Scalar

Sometimes you have a situation where a black box functions prints the output to `STDOUT` and you want to get this output into a scalar. This is just as valid under mod_perl, where you want the `STDOUT` to be tied to the `Apache` object. So that's where the `IO::String` package comes to help. You can re-tie() the STDOUT (or any other filehandler to a string) by doing a simple select() on the `IO::String` object and at the end to re-tie() the STDOUT back to its original stream:

```
my $str;
my $str_fh = IO::String->new($str);
my $old_fh = select($str_fh);

# some function that prints to currently selected file handler.
print_stuff()

# reset default fh to previous value
select($old_fh) if defined $old_fh;
```

5.20 Apache::print() and CORE::print()

Under mod_perl `CORE::print()` will redirect its data to `Apache::print()` since the STDOUT filehandle is tied to the *Apache* module. This allows us to run CGI scripts unmodified under `Apache::Registry` by chaining the output of one content handler to the input of the other handler.

`Apache::print()` behaves mostly like the built-in *print()* function. In addition it sets a timeout so that if the client connection is broken the handler won't wait forever trying to print data downstream to the client.

There is also an optimization built into `Apache::print()`. If any of the arguments to the method are scalar references to strings, they are automatically dereferenced for you. This avoids needless copying of large strings when passing them to subroutines. For example:

```
$long_string = "A" x 10000000;
$r->print(\$long_string);
```

If you still want to print the reference you can always call:

```
$r->print(\\$foo);
```

or by forcing it into a scalar context:

```
print(scalar($foo));
```

5.21 Global Variables Persistence

Since the child process generally doesn't exit before it has serviced several requests, global variables persist inside the same process from request to request. This means that you must never rely on the value of the global variable if it wasn't initialized at the beginning of the request processing. See "Variables globally, lexically scoped and fully qualified" for more information.

You should avoid using global variables unless it’s impossible without them, because it will make code development harder and you will have to make certain that all the variables are initialized before they are used. Use my () scoped variables wherever you can.

You should be especially careful with Perl Special Variables which cannot be lexically scoped. You have to use `local()` instead.

Here is an example with Perl hash variables, which store the iteration state in the hash variable and that state persists between requests unless explicitly reset. Consider the following registry script:

```
#file:hash_iteration.pl
#----------------------
our %hash;
%hash = map {$_ => 1 } ’a’..’c’ unless %hash;

print "Content-type: text/plain\n\n";

for (my ($k, $v) = each %hash) {
    print "$k $v\n";
    last;
}
```

That script prints different values on the first 3 invocations and prints nothing on the 4th, and then repeats the loop. (when you run with httpd -X). There are 3 hash key/value pairs in the global variable `%hash`.

In order to get the iteration state to its initial state at the beginning of each request, you need to reset the iterator as explained in the manpage for the `each()` operator. So adding:

```
keys %hash;
```

before using `%hash` solves the problem for the current example.

5.22 Generating correct HTTP Headers

A HTTP response header consists of at least two fields. HTTP response and MIME type header `Content-type`:

```
HTTP/1.0 200 OK
Content-Type: text/plain
```

After adding one more new line, you can start printing the content. A more complete response includes the date timestamp and server type, for example:

```
HTTP/1.0 200 OK
Date: Tue, 28 Dec 1999 18:47:58 GMT
Server: Apache/1.3.10-dev (Unix) mod_perl/1.21_01-dev
Content-Type: text/plain
```

To notify that the server was configured with KeepAlive Off, you need to tell the client that the connection was closed, with:

```
Connection: close
```

There can be other headers as well, like caching control and others specified by the HTTP protocol. You can code the response header with a single print():

```
  print qq{HTTP/1.1 200 OK
Date: Tue, 28 Dec 1999 18:49:41 GMT
Server: Apache/1.3.10-dev (Unix) mod_perl/1.21_01-dev
Connection: close
Content-type: text/plain

  };
```

or with a *"here"* style print:

```
  print <<EOT;
HTTP/1.1 200 OK
Date: Tue, 28 Dec 1999 18:49:41 GMT
Server: Apache/1.3.10-dev (Unix) mod_perl/1.21_01-dev
Connection: close
Content-type: text/plain

EOT
```

Notice the double new line at the end. But you have to prepare a timestamp string (`Apache::Util::ht_time()` does just this) and to know what server you are running under. You needed to send only the response MIME type (`Content-type`) under mod_cgi, so why would you want to do this manually under mod_perl?

Actually sometimes you do want to set some headers manually, but not every time. So mod_perl gives you the default set of headers, just like in the example above. And if you want to override or add more headers you can do that as well. Let’s see how to do that.

When writing your own handlers and scripts with the Perl Apache API the proper way to send the HTTP header is with the send_http_header() method. If you need to add or override methods you can use the header_out() method:

```
$r->header_out("Server" => "Apache Next Generation 10.0");
$r->header_out("Date" => "Tue, 28 Dec 1999 18:49:41 GMT");
```

When you have prepared all the headers you send them with:

```
$r->send_http_header;
```

Some headers have special aliases:

```
$r->content_type('text/plain');
```

is the same as:

```
$r->header_out("Content-type" => "text/plain");
```

A typical handler looks like this:

```
$r->content_type('text/plain');
$r->send_http_header;
return OK if $r->header_only;
```

If the client issues an HTTP `HEAD` request rather than the usual `GET`, to be compliant with the HTTP protocol we should not send the document body, but only the HTTP header. When Apache receives a `HEAD` request, *header_only()* returns *true*. If we see that this has happened, we return from the handler immediately with an `OK` status code.

Generally, you don't need the explicit content type setting, since Apache does this for you, by looking up the MIME type of the request and by matching the extension of the URI in the MIME tables (from the *mime.types* file). So if the request URI is */welcome.html*, the `text/html` content-type will be picked. However for CGI scripts or URIs that cannot be mapped by a known extension, you should set the appropriate type by using content_type() method.

The situation is a little bit different with `Apache::Registry` and similar handlers. If you take a basic CGI script like this:

```
print "Content-type: text/plain\r\n\r\n";
print "Hello world";
```

it wouldn't work, because the HTTP header will not be sent out. By default, mod_perl does not send any headers itself. You may wish to change this by adding

```
PerlSendHeader On
```

in the `Apache::Registry` `<Location>` section of your configuration. Now, the response line and common headers will be sent as they are by mod_cgi. Just as with mod_cgi, `PerlSendHeader` will not send the MIME type and a terminating double newline. Your script must send that itself, e.g.:

```
print "Content-type: text/html\r\n\r\n";
```

According to HTTP specs, you should send "\cM\cJ", "\015\012" or "\0x0D\0x0A" string. The "\r\n" is the way to do that on UNIX and MS-DOS/Windows machines. However, on a Mac "\r\n" eq "\012\015", exactly the other way around.

Note, that in most UNIX CGI scripts, developers use a simpler "\n\n" and not "\r\n\r\n". There are occasions where sending "\n" without "\r" can cause problems, make it a habit to always send "\r\n" every time.

If you use an OS which uses the EBCDIC as character set (e.g. BS2000-Posix), you should use this method to send the Content-type header:

```
shift->send_http_header('text/html');
```

The `PerlSendHeader On` directive tells mod_perl to intercept anything that looks like a header line (such as `Content-Type: text/plain`) and automatically turn it into a correctly formatted HTTP/1.0 header, the same way it happens with CGI scripts running under mod_cgi. This allows you to keep your CGI scripts unmodified.

You can use `$ENV{PERL_SEND_HEADER}` to find out whether `PerlSendHeader` is `On` or `Off`. You use it in your module like this:

```
if($ENV{PERL_SEND_HEADER}) {
    print "Content-type: text/html\r\n\r\n";
}
else {
    my $r = Apache->request;
    $r->content_type('text/html');
    $r->send_http_header;
}
```

Note that you can always use the code in the else part of the above example, no matter whether the `PerlSendHeader` directive is On or Off.

If you use `CGI.pm`'s `header()` function to generate HTTP headers, you do not need to activate this directive because `CGI.pm` detects *mod_perl* and calls `send_http_header()` for you.

There is no free lunch--you get the mod_cgi behavior at the expense of the small but finite overhead of parsing the text that is sent. Note that mod_perl makes the assumption that individual headers are not split across print statements.

The `Apache::print()` routine has to gather up the headers that your script outputs, in order to pass them to `$r->send_http_header`. This happens in `src/modules/perl/Apache.xs` (`print`) and `Apache/Apache.pm` (`send_cgi_header`). There is a shortcut in there, namely the assumption that each print statement contains one or more complete headers. If for example you generate a `Set-Cookie` header by multiple `print()` statements, like this:

```
print "Content-type: text/plain\n";
print "Set-Cookie: iscookietext\; ";
print "expires=Wednesday, 09-Nov-1999 00:00:00 GMT\; ";
print "path=\/\; ";
print "domain=\.mmyserver.com\; ";
print "\r\n\r\n";
print "hello";
```

Your generated `Set-Cookie` header is split over a number of print() statements and gets lost. The above example wouldn't work! Try this instead:

```
my $cookie = "Set-Cookie: iscookietext\; ";
$cookie .= "expires=Wednesday, 09-Nov-1999 00:00:00 GMT\; ";
$cookie .= "path=\/\; ";
$cookie .= "domain=\.mmyserver.com\; ";
print "Content-type: text/plain\n",
print "$cookie\r\n\r\n";
print "hello";
```

Of course using a special purpose cookie generator modules, like `Apache::Cookie`, `CGI::Cookie` etc is an even cleaner solution.

Sometimes when you call a script you see an ugly "Content-Type: text/html" displayed at the top of the page, and of course the rest of the HTML code won't be rendered correctly by the browser. As you have seen above, this generally happens when your code has already sent the header so you see the duplicate header rendered into the browser's page. This might happen when you call the `CGI.pm` `$q->header` method or mod_perl's `$r->send_http_header`.

If you have a complicated application where the header might be generated from many different places, depending on the calling logic, you might want to write a special subroutine that sends a header, and keeps track of whether the header has been already sent. Of course you can use a global variable to flag that the header has already been sent:

```
use strict;
use vars qw{$header_printed};
$header_printed = 0;

print_header("text/plain");
print "It worked!\n";
print_header("text/plain");

sub print_header {
  my $type = shift || "text/html";
  unless  ($header_printed) {
    $header_printed = 1;
    my $r = Apache->request;
    $r->content_type($type);
    $r->send_http_header;
  }
}
```

`$header_printed` is the variable that flags whether the header was sent or not and it gets initialized to false (0) at the beginning of each code invocation. Note that the second invocation of print_header() within the same code, will do nothing, since `$header_printed` will become true after print_header() will be executed for the first time.

A solution that is a little bit more memory friendly is to use a fully qualified variable instead:

```
use strict;
$main::header_printed = 0;

print_header("text/plain");
print "It worked!\n";
print_header("text/plain");

sub print_header {
  my $type = shift || "text/html";
  unless  ($main::header_printed) {
    $main::header_printed = 1;
    my $r = Apache->request;
```

```
    $r->content_type($type);
    $r->send_http_header;
  }
}
```

We just removed the global variable predeclaration, which allowed us to use `$header_printed` under `"use strict"` and replaced `$header_printed` with `$main::header_printed`;

You may become tempted to use a more elegant Perl solution--the nested subroutine effect which seems to be a natural approach to take here. Unfortunately it will not work. If the process was starting fresh for each script or handler, like with plain mod_cgi scripts, it would work just fine:

```
use strict;

print_header("text/plain");
print "It worked!\n";
print_header("text/plain");

{
  my $header_printed = 0;
  sub print_header {
    my $type = shift || "text/html";
    unless  ($header_printed) {
      $header_printed = 1;
      my $r = Apache->request;
      $r->content_type($type);
      $r->send_http_header;
    }
  }
}
```

In this code `$header_printed` is declared as lexically scoped (with my ()) outside the subroutine print_header() and modified inside of it. Curly braces define the block which limits the scope of the lexically variable.

This means that once print_header() sets it to 1, it will stay 1 as long as the code is running. So all subsequent calls to this subroutine will just return without doing a thing. This would serve our purpose, but unfortunately it will work only for the first time the script is invoked within a process. When the script is executed for the second or subsequent times and is served by the same process--the header will not be printed anymore, since print_header() will remember that the value of `$header_printed` is equal to 1--it won't be reinitialized, since the subroutine won't be recompiled.

Why can't we use a lexical without hitting the nested subroutine effect? Because when we've discussed Apache::Registry secrets we have seen that the code is wrapped in a `handler` routine, effectively turning any subroutines within the file a script resides in into nested subroutines. Hence we are forced to use a global in this situation.

Let's make our smart method more elaborate with respect to the `PerlSendHeader` directive, so that it always does the right thing. It's especially important if you write an application that you are going to distribute, hopefully under one of the Open Source or GPL licenses.

You can continue to improve this subroutine even further to handle additional headers, such as cookies.

See also Correct Headers--A quick guide for mod_perl users

5.23 NPH (Non Parsed Headers) scripts

To run a Non Parsed Header CGI script under mod_perl, simply add to your code:

```
local $| = 1;
```

And if you normally set `PerlSendHeader On`, add this to your server's configuration file:

```
<Files */nph-*>
  PerlSendHeader Off
</Files>
```

5.24 BEGIN blocks

Perl executes `BEGIN` blocks as soon as possible, at the time of compiling the code. The same is true under mod_perl. However, since mod_perl normally only compiles scripts and modules once, either in the parent server or once per-child, `BEGIN` blocks in that code will only be run once. As the `perlmod` manpage explains, once a `BEGIN` block has run, it is immediately undefined. In the mod_perl environment, this means that `BEGIN` blocks will not be run during the response to an incoming request unless that request happens to be the one that causes the compilation of the code.

`BEGIN` blocks in modules and files pulled in via `require()` or `use()` will be executed:

- Only once, if pulled in by the parent process.
- Once per-child process if not pulled in by the parent process.
- An additional time, once per child process if the module is pulled in off disk again via `Apache::StatINC`.
- An additional time, in the parent process on each restart if `PerlFreshRestart` is `On`.
- Unpredictable if you fiddle with `%INC` yourself.

`BEGIN` blocks in `Apache::Registry` scripts will be executed, as above plus:

- Only once, if pulled in by the parent process via

  `Apache::RegistryLoader`.
- Once per-child process if not pulled in by the parent process.

- An additional time, once per child process, each time the script file changes on disk.
- An additional time, in the parent process on each restart if pulled in by the parent process via `Apache::RegistryLoader` and `PerlFreshRestart` is `On`.

Make sure you read Evil things might happen when using PerlFreshRestart.

5.25 END blocks

As the `perlmod` manpage explains, an `END` subroutine is executed as late as possible, that is, when the interpreter exits. In the mod_perl environment, the interpreter does not exit until the server shuts down. However, mod_perl does make a special case for `Apache::Registry` scripts.

Normally, `END` blocks are executed by Perl during its `perl_run()` function. This is called once each time the Perl program is executed, i.e. under mod_cgi, once per invocation of the CGI script. However, mod_perl only calls `perl_run()` once, during server startup. Any `END` blocks encountered during main server startup, i.e. those pulled in by `PerlRequire`, `PerlModule` and the startup file, are suspended.

Except during the cleanup phase, any `END` blocks encountered during compilation of `Apache::Registry` scripts (including those defined in the packages `use()`'d by the script), including subsequent invocations when the script is cached in memory, are called after the script has completed.

All other `END` blocks encountered during other `Perl*Handler` call-backs, e.g. `PerlChildInitHandler`, will be suspended while the process is running and called during `child_exit()` when the process is shutting down. Module authors might wish to use `$r->register_cleanup()` as an alternative to `END` blocks if this behavior is not desirable. `$r->register_cleanup()` is called at the CleanUp processing phase of each request and thus can be used to emulate plain perl's `END{}` block behavior.

The last paragraph is very important for handling the case of 'User Pressed the Stop Button'.

If you only want something to run once in the parent on shutdown or restart you can use `$r->register_cleanup()` in the *startup.pl*.

```
#PerlRequire startup.pl
warn "parent pid is $$\n";
Apache->server->register_cleanup
  (sub { warn "server cleanup in $$\n"});
```

This is usually useful when some server wide cleanup should be performed when the server is stopped or restarted.

5.26 CHECK And INIT Blocks

These blocks run when compilation is complete, but before the program starts. `CHECK` can mean "checkpoint" or "double-check" or even just "stop". `INIT` stands for "initialization". The difference is subtle; `CHECK` blocks are run just after the compilation ends, `INIT` just before the runtime begins. (Hence the `-c` command-line flag to perl runs `CHECK` blocks but not `INIT` blocks.)

Perl only calls these blocks during perl_parse(), which mod_perl calls once at startup time. Therefore `CHECK` and `INIT` blocks don't work for the same reason these don't:

```
% perl -e 'eval qq(CHECK { print "ok\n" })'
% perl -e 'eval qq(INIT  { print "ok\n" })'
```

5.27 Command Line Switches (-w, -T, etc)

Normally when you run perl from the command line, you have the shell invoke it with `#!/bin/perl` (sometimes referred to as the shebang line). In scripts running under mod_cgi, you may use perl execution switch arguments as described in the `perlrun` manpage, such as `-w`, `-T` or `-d`. Since scripts running under mod_perl don't need the shebang line, all switches except `-w` are ignored by mod_perl. This feature was added for a backward compatibility with CGI scripts.

Most command line switches have a special variable equivalent which allows them to be set/unset in code. Consult the `perlvar` manpage for more details.

5.27.1 Warnings

There are three ways to enable warnings:

- **Globally to all Processes**

  Setting:

  ```
  PerlWarn On
  ```

  in `httpd.conf` will turn warnings `On` in any script.

  You can then fine tune your code, turning warnings `Off` and `On` by using the `warnings` pragma in your scripts (or by setting the `$^W` variable, if you prefer to be compatible with older, pre-5.6, perls).

- **Locally to a script**

  ```
  #!/usr/bin/perl -w
  ```

  will turn warnings `On` for the scope of the script. You can turn them `Off` and `On` in the script with `no warnings;` and `use warnings;` as noted above.

- **Locally to a block**

  This code turns warnings mode `On` for the scope of the block.

  ```
  {
    use warnings;
    # some code
  }
  # back to the previous mode here
  ```

This turns it `Off`:

```
{
  no warnings;
  # some code
}
# back to the previous mode here
```

This turns `Off` only the warnings from the listed categories : (warnings categories are explicited in the `perldiag` manpage.)

```
{
  no warnings qw(uninitialized unopened);
  # some code
}
# back to the previous mode here
```

If you want to turn warnings *On* for the scope of the whole file, you can do this by adding:

```
use warnings;
```

at the beginning of the file.

While having warning mode turned `On` is essential for a development server, you should turn it globally `Off` in a production server, since, for example, if every served request generates only one warning, and your server serves millions of requests per day, your log file will eat up all of your disk space and your system will die.

5.27.2 Taint Mode

Perl's `-T` switch enables *Taint* mode. (META: Link to security chapter). If you aren't forcing all your scripts to run under `Taint` mode you are looking for trouble from malicious users. (See the *perlsec* manpage for more information. Also read the *re* pragma's manpage.)

If you have some scripts that won't run under Taint mode, run only the ones that run under mod_perl with Taint mode enabled and the rest on another server with Taint mode disabled -- this can be either a mod_cgi in the front-end server or another back-end mod_perl server. You can use the mod_rewrite module and redirect requests based on the file extensions. For example you can use *.tcgi* for the taint-clean scripts, and *cgi* for the rest.

When you have this setup you can start working toward cleaning the rest of the scripts, to make them run under the Taint mode. Just because you have a few dirty scripts doesn't mean that you should jeopardize your whole service.

Since the `-T` switch doesn't have an equivalent perl variable, mod_perl provides the `PerlTaintCheck` directive to turn on taint checks. In `httpd.conf`, enable this mode with:

```
PerlTaintCheck On
```

Now any code compiled inside httpd will be taint checked.

If you use the `-T` switch, Perl will warn you that you should use the `PerlTaintCheck` configuration directive and will otherwise ignore it.

5.27.3 Other switches

Finally, if you still need to set additional perl startup flags such as `-d` and `-D`, you can use an environment variable `PERL5OPT`. Switches in this variable are treated as if they were on every Perl command line.

Only the `-[DIMUdmw]` switches are allowed.

When the `PerlTaintCheck` variable is turned on, the value of `PERL5OPT` will be ignored.

[META: verify]

See also Apache::PerlRun.

5.28 The strict pragma

It’s _absolutely_ mandatory (at least for development) to start all your scripts with:

```
use strict;
```

If needed, you can always turn off the ’strict’ pragma or a part of it inside the block, e.g:

```
{
  no strict ’refs’;
  ... some code
}
```

It’s more important to have the `strict` pragma enabled under mod_perl than anywhere else. While it’s not required by the language, its use cannot be too strongly recommended. It will save you a great deal of time. And, of course, clean scripts will still run under mod_cgi (plain CGI)!

5.29 Passing ENV variables to CGI

To pass an environment variable from *httpd.conf*, add to it:

```
PerlSetEnv key val
PerlPassEnv key
```

e.g.:

```
PerlSetEnv PERLDB_OPTS "NonStop=1 LineInfo=/tmp/db.out AutoTrace=1"
```

will set $ENV{PERLDB_OPTS}, and it will be accessible in every child.

%ENV is only set up for CGI emulation. If you are using the API, you should use $r->subprocess_env, $r->notes or $r->pnotes for passing data around between handlers. %ENV is slow because it must update the underlying C environment table. It also insecure since its use exposes the data on systems which allow users to see the environment with ps.

In any case, %ENV and the tables used by those methods are all cleared after the request is served.

The Perl %ENV is cleared during startup, but the C environment is left intact. With a combo of forking 'env' and <Perl> sections you can do even do wildcards matching. For example, this passes all environment variables that begin with the letter H:

```
<Perl>
  local $ENV{PATH} = '/usr/bin';
  local $_;

  for ('env') {
      next unless /^(H.*)=/;
      push @PassEnv, $1;
  }
</Perl>
```

See also PerlSetupEnv which can enable/disable environment variables settings.

5.30 -M and other time() file tests under mod_perl

Under mod_perl, files that have been created after the server's (child) startup are reported as having a negative age with -M (-C -A) test. This is obvious if you remember that you will get the negative result if the server was started before the file was created. It's normal behavior with perl.

If you want to have -M report the time relative to the current request, you should reset the $^T variable just as with any other perl script. Add:

```
local $^T = time;
```

at the beginning of the script.

Another even simpler solution would be to specify a fixup handler, which will be executed before each script is run:

```
sub Apache::PerlBaseTime::handler {
  $^T = shift->request_time;
  return Apache::Constants::DECLINED;
}
```

and then in the *httpd.conf*:

```
PerlFixupHandler Apache::PerlBaseTime
```

This technique is better performance-wise as it skips the time() system call, and uses the already available time of the request has been started at via `$r->request_time` method.

5.31 Apache and syslog

When native syslog support is enabled, the stderr stream will be redirected to `/dev/null`!

It has nothing to do with mod_perl (plain Apache does the same). Doug wrote the Apache::LogSTDERR module to work around this.

5.32 File tests operators

Remember that with mod_perl you might get negative times when you use file test operators like `-M` -- last modification time, `-A` -- last access time, `-C` -- last inode-change time, and others. `-M` returns the difference in time between the modification time of the file and the time the script was started. Because the `^T` variable is not reset on each script invocation, and is equal to the time when the process was forked, you might want to perform:

```
$^T = time;
```

at the beginning of your scripts to simulate the regular perl script behaviour of file tests.

META: Above is near duplicate of "-M and other time() file tests under mod_perl" make a link instead

5.33 Filehandlers and locks leakages

META: duplication at debug.pod: =head3 Safe Resource Locking

When you write a script running under mod_cgi, you can get away with sloppy programming, like opening a file and letting the interpreter close it for you when the script had finished its run:

```
open IN, "in.txt" or die "Cannot open in.txt for reading : $!\n";
```

For mod_perl, before the end of the script you **must** `close()` any files you opened!

```
close IN;
```

If you forget to `close()`, you might get file descriptor leakage and (if you `flock()ed` on this file descriptor) also unlock problems.

Even if you do call close(), if for some reason the interpreter was stopped before the `close()` call, the leakage will still happen. See for example Handling the 'User pressed Stop button' case. After a long run without restarting Apache your machine might run out of file descriptors, and worse, files might be left locked and unusable.

What can you do? Use `IO::File` (and the other `IO::*` modules). This allows you to assign the file handler to variable which can be `my ()` (lexically) scoped. When this variable goes out of scope the file or other file system entity will be properly closed (and unlocked if it was locked). Lexically scoped variables will always go out of scope at the end of the script's invocation even if it was aborted in the middle. If the variable was defined inside some internal block, it will go out of scope at the end of the block. For example:

```
{
  my $fh = IO::File->new("filename") or die $!;
  # read from $fh
} # ...$fh is closed automatically at end of block, without leaks.
```

As I have just mentioned, you don't have to create a special block for this purpose. A script in a file is effectively written in a block with the same scope as the file, so you can simply write:

```
my $fh = IO::File->new("filename") or die $!;
  # read from $fh
  # ...$fh is closed automatically at end of script, without leaks.
```

Using a `{ BLOCK }`) makes sure is that the file is closed the moment that the end of the block is reached.

An even faster and lighter technique is to use `Symbol.pm`:

```
my $fh = Symbol::gensym();
open $fh, "filename" or die $!;
```

Use these approaches to ensure you have no leakages, but don't be too lazy to write `close()` statements. Make it a habit.

Under perl 5.6.0 we can do this instead:

```
open my $fh, $filename or die $! ;
```

5.34 Code has been changed, but it seems the script is running the old code

Files pulled in via `use` or `require` statements are not automatically reloaded when they change on disk. See Reloading Modules and Required Files for more information.

5.35 The Script Is Too Dirty, But It Does The Job And I Cannot Afford To Rewrite It.

You still can win from using mod_perl.

One approach is to replace the `Apache::Registry` handler with `Apache::PerlRun` and define a new location. The script can reside in the same directory on the disk.

```
# httpd.conf
Alias /cgi-perl/ /home/httpd/cgi/

<Location /cgi-perl>
  #AllowOverride None
  SetHandler perl-script
  PerlHandler Apache::PerlRun
  Options ExecCGI
  allow from all
  PerlSendHeader On
</Location>
```

See Apache::PerlRun--a closer look

Another "bad", but workable method is to set `MaxRequestsPerChild` to 1, which will force each child to exit after serving only one request. You will get the preloaded modules, etc., but the script will be compiled for each request, then be thrown away. This isn't good for "high-traffic" sites, as the parent server will need to fork a new child each time one is killed. You can fiddle with `MaxStartServers` and `MinSpareServers`, so that the parent pre-spawns more servers than actually required and the killed one will immediately be replaced with a fresh one. Probably that's not what you want.

5.36 Apache::PerlRun--a closer look

`Apache::PerlRun` gives you the benefit of preloaded Perl and its modules. This module's handler emulates the CGI environment, allowing programmers to write scripts that run under CGI or mod_perl without any change. Unlike `Apache::Registry`, the `Apache::PerlRun` handler does not cache the script inside a subroutine. Scripts will be "compiled" on each request. After the script has run, its name space is flushed of all variables and subroutines. Still, you don't have the overhead of loading the Perl interpreter and the compilation time of the standard modules. If your script is very light, but uses lots of standard modules, you will see no difference between `Apache::PerlRun` and `Apache::Registry`!.

Be aware though, that if you use packages that use internal variables that have circular references, they will be not flushed!!! `Apache::PerlRun` only flushes your script's name space, which does not include any other required packages' name spaces. If there's a reference to a `my ()` scoped variable that's keeping it from being destroyed after leaving the eval scope (of `Apache::PerlRun`), that cleanup might not be taken care of until the server is shutdown and `perl_destruct()` is run, which always happens after running command line scripts. Consider this example:

```
package Foo;
sub new { bless {} }
sub DESTROY {
  warn "Foo->DESTROY\n";
}

eval <<'EOF';
package my_script;
my $self = Foo->new;
#$self->{circle} = $self;
```

```
EOF

print $@ if $@;
print "Done with script\n";
```

When executed as a plain script you'll see:

```
Foo->DESTROY
Done with script
```

Then, uncomment the line where `$self` makes a circular reference, and you'll see:

```
Done with script
Foo->DESTROY
```

If you run this example with the circular reference enabled under mod_perl you won't see `Foo->DESTROY` until server shutdown, or until your module properly takes care of things. Note that the warn() call logs its messages to the *error_log* file, so you should expect the output there and not together with STDOUT.

5.37 Sharing variables between processes

META: to be completed

- Global variables initialized at server startup, through the Perl startup file, can be shared between processes, until modified by some of the processes. e.g. when you write:

  ```
  $My::debug = 1;
  ```

  all processes will read the same value. If one of the processes changes that value to `0`, it will still be equal to `1` for any other process, but not for the one which actually made the change. When a process modifies a shared variable, it becomes the process' private copy.

- `IPC::Shareable` can be used to share variables between children.

- libmm

- other methods?

5.38 Preventing Apache::Constants Stringification

In mod_perl, you are going to use a certain number of constants in your code, mainly exported from `Apache::Constants`. However, in some cases, Perl will not understand that the constant you're trying to call is really a constant, but interprets it as a string. This is the case with the hash notation `=>`, which automatically stringifies the key.

For example:

```
$r->custom_response(FORBIDDEN => "File size exceeds quota.");
```

This will not set a custom response for `FORBIDDEN`, but for the string `"FORBIDDEN"`, which clearly isn't what is expected. You'll get an error like this:

```
[Tue Apr 23 19:46:14 2002] null: Argument "FORBIDDEN" isn't numeric
                           in subroutine entry at ...
```

Therefore, you can avoid this by not using the hash notation for things that don't require it.

```
$r->custom_response(FORBIDDEN,  "File size exceeds quota.");
```

There are other workarounds, which you should avoid using unless you really have to use hash notation:

```
my %hash = (
    FORBIDDEN()    => 'this is forbidden',
    +AUTH_REQUIRED => "You aren't authorized to enter!",
);
```

Another important note is that you should be using the correct constants defined here, and not direct HTTP codes. For example:

```
sub handler {
    return 200;
}
```

Is not correct. The correct use is:

```
use Apache::Constants qw(OK);

sub handler {
    return OK;
}
```

Also remember that `OK != HTTP_OK`.

5.39 Transitioning from Apache::Registry to Apache handlers

Even if you are a CGI script die-hard at some point you might want to move a few or all your scripts to Apache Perl handlers. Actually this is an easy task, since we saw already what `Apache::Registry` makes our scripts appear to Apache to be Perl handlers.

When you no longer need backward mod_cgi compatibility you can benefit from the Perl libraries working only under mod_perl. We will see why in a moment.

Let's see an example. We will start with a mod_cgi compatible CGI script running under `Apache::Registry`, transpose it into a Perl content handler and then convert it to use `Apache::Request` and `Apache::Cookie`.

5.39.1 Starting with mod_cgi Compatible Script

This is the original script's code we are going to work with:

```
cookie_script.pl
----------------
use strict;
use CGI;
use CGI::Cookie;
use vars qw($q $switch $status $sessionID);

init();
print_header();
print_status();

### <-- subroutines --> ###

# the init code
############
sub init{
  $q = new CGI;

  $switch = $q->param("switch") ? 1 : 0;

    # try to retrieve the session ID
            # fetch existing cookies
  my %cookies = CGI::Cookie->fetch;
  $sessionID = exists $cookies{'sessionID'}
     ? $cookies{'sessionID'}->value : '';

    # 0 = not running, 1 = running
  $status = $sessionID ? 1 : 0;

    # switch status if asked to
  $status = ($status+1) % 2 if $switch;

  if ($status){
      # preserve sessionID if exists or create a new one
    $sessionID ||= generate_sessionID() if $status;
  } else {
      # delete the sessionID
    $sessionID = '';
  }

} # end of sub init

#################
sub print_header{
    # prepare a cookie
  my $c = CGI::Cookie->new
    ('-name'    => 'sessionID',
     '-value'   => $sessionID,
     '-expires' => '+1h');

  print $q->header
    (-type   => 'text/html',
```

```
      -cookie => $c);

} # end of sub print_header


# print the current Session status and a form to toggle the status
#################
sub print_status{

  print qq{<HTML><HEAD><TITLE>Cookie</TITLE></HEAD><BODY>};

    # print status
  print "<B>Status:</B> ",
        $status
          ? "Session is running with ID: $sessionID"
          : "No session is running";


    # change status form
  my $button_label = $status ? "Stop" : "Start";
  print qq{<HR>
       <FORM>
         <INPUT TYPE=SUBMIT NAME=switch VALUE=" $button_label ">
       </FORM>
         };

  print qq{</BODY></HTML>};

} # end of sub print_status

# A dummy ID generator
# Replace with a real session ID generator
########################
sub generate_sessionID {
  return scalar localtime;
} # end of sub generate_sessionID
```

The code is very simple. It creates a session if you've pressed the *'Start'* button or deletes it if you've pressed the *'Stop'* button. The session is stored and retrieved using the cookies technique.

Note that we have split the obviously simple and short code into three logical units, by putting the code into three subroutines. init() to initialize global variables and parse incoming data, print_header() to print the HTTP headers including the cookie header, and finally print_status() to generate the output. Later we will see that this logical separation will allow us an easy conversion to Perl content handler code.

We have used global variables for a few variables since we didn't want to pass them from function to function. In a big project you should be very restrictive about what variables should be allowed to be global, if any at all. In any case, the init() subroutine makes sure all these variables are re-initialized for each code reinvocation.

Note that we have used a very simple generate_sessionID() function that returns a date string (i.e. Wed Apr 12 15:02:23 2000) as a session ID. You want to replace this one with code which generates a unique session every time it was called. And it should be secure, i.e. users will not be able to forge one and do nasty things.

5.39.2 Converting into Perl Content Handler

Now let's convert this script into a content handler. There are two parts to this task; the first one is to configure Apache to run the new code as a Perl handler, the second one is to modify the code itself.

First we add the following snippet to *httpd.conf*:

```
PerlModule Test::Cookie
<Location /test/cookie>
  SetHandler perl-script
  PerlHandler Test::Cookie
</Location>
```

After we restart the server, when there is a request whose URI starts with */test/cookie*, Apache will execute the `Test::Cookie::handler()` subroutine as a content handler. We made sure to preload the `Test::Cookie` module at server start-up, with the `PerlModule` directive.

Now we are going to modify the script itself. We copy the content to the file *Cookie.pm* and place it into one of the directories listed in `@INC`. For example if */home/httpd/perl* is a part of `@INC` and since we want to call this package `Test::Cookie`, we can put *Cookie.pm* into the */home/httpd/perl/Test/* directory.

So this is the new code. Notice that all the subroutines were left unmodified from the original script, so to make the differences clear we do not repeat them here.

```
Test/Cookie.pm
--------------
package Test::Cookie;
use Apache::Constants qw(:common);

use strict;
use CGI;
use CGI::Cookie;
use vars qw($q $switch $status $sessionID);

sub handler{
  my $r = shift;
  Apache->request($r);

  init();
  print_header();
  print_status();

  return OK;
}

### <-- subroutines --> ###
# all subroutines as before

1;
```

As you see there are two lines added to the beginning of the code:

```
package Test::Cookie;
use Apache::Constants qw(:common);
```

The first one declares the package name and the second one imports some symbols commonly used in Perl handlers to return status codes.

```
use strict;
use CGI;
use CGI::Cookie;
use vars qw($q $switch $status $sessionID);
```

This code is left unchanged just as before.

```
sub handler{
  my $r = shift;
  Apache->request($r);

  init();
  print_header();
  print_status();

  return OK;
}
```

Each content handler (and any other handler) should begin with a subroutine called handler(). This subroutine is called when a request's URI starts with */test/cookie* as per our configuration. Of course you can choose a different name, for example execute(), but then you must explicitly use it in the configuration directives in the following way:

```
PerlModule Test::Cookie
<Location /test/cookie>
  SetHandler perl-script
  PerlHandler Test::Cookie::execute
</Location>
```

But we will use the default name, handler().

The handler() subroutine is just like any other subroutine, but generally it has the following structure:

```
sub handler{
  my $r = shift;

  # the code

  # status (OK, DECLINED or else)
  return OK;
}
```

First we get the request object by shifting it from `@_` and assigning it to the `$r` variable.

Second we write the code that does the processing of the request.

Third we return the status of the execution. There are many possible statuses, the most commonly used are `OK` and `DECLINED`, which tell the server whether they have completed the request phase that the handler was assigned to do or not. If not, another handler must complete the processing. `Apache::Constants` imports these two and other some commonly used status codes.

So in our example all we had to do was to wrap the three calls:

```
    init();
    print_header();
    print_status();
```

inside:

```
  sub handler{
    my $r = shift;
    Apache->request($r);

    return OK;
  }
```

There is one line we didn't discuss:

```
    Apache->request($r);
```

Since we use `CGI.pm`, it relies on the fact that `$r` was set in the `Apache` module. `Apache::Registry` did that behind the scenes. Since we don't use `Apache::Registry` here, we have to do that ourselves.

The one last thing we should do is to add `1;` at the end of the module, just like with any Perl module, so `PerlModule` will not fail when it tries to load `Test::Cookie`.

So to summarize, we took the original script's code and added the following eight lines:

```
  package Test::Cookie;
  use Apache::Constants qw(:common);

  sub handler{
    my $r = shift;
    Apache->request($r);

    return OK;
  }
  1;
```

and now we have a fully fledged Perl Content Handler.

5.39.3 Converting to use Apache Perl Modules

So now we have a complete PerlHandler, let's convert it to use Apache Perl modules. This breaks the backward compatibility, but gives us better performance, mainly because the internals of many of these Perl modules are implemented in C, therefore we should get a significant improvement in speed. The section "TMTOWTDI: Convenience and Performance" compares the three approaches.

What we are going to do is to replace `CGI.pm` and `CGI::Cookie` with `Apache::Request` and `Apache::Cookie` respectively. The two modules are written in C with the XS interface to Perl, which makes code much faster if it utilizes any of these modules a lot. `Apache::Request` uses an API similar to the one `CGI` uses, the same goes for `Apache::Cookie` and `CGI::Cookie`. This allows an easy porting process. Basically we just replace:

```
use CGI;
$q = new CGI;
```

with:

```
use Apache::Request ();
my $q = Apache::Request->new($r);
```

and

```
use CGI::Cookie ();
my $cookie = CGI::Cookie->new(...)
```

with

```
use Apache::Cookie ();
my $cookie = Apache::Cookie->new($r, ...);
```

This is the new code for `Test::Cookie2`:

```
Test/Cookie2.pm
--------------
package Test::Cookie2;
use Apache::Constants qw(:common);

use strict;
use Apache::Request;
use Apache::Cookie ();
use vars qw($r $q $switch $status $sessionID);

sub handler{
  $r = shift;

  init();
  print_header();
  print_status();

  return OK;
}

### <-- subroutines --> ###

# the init code
############
sub init{

  $q = Apache::Request->new($r);
  $switch = $q->param("switch") ? 1 : 0;
```

```
      # fetch existing cookies
  my %cookies = Apache::Cookie->fetch;
    # try to retrieve the session ID
  $sessionID = exists $cookies{'sessionID'}
     ? $cookies{'sessionID'}->value : '';

    # 0 = not running, 1 = running
  $status = $sessionID ? 1 : 0;

    # switch status if asked to
  $status = ($status+1) % 2 if $switch;


  if ($status){
      # preserve sessionID if exists or create a new one
    $sessionID ||= generate_sessionID() if $status;
  } else {
      # delete the sessionID
    $sessionID = '';
  }


} # end of sub init


#################
sub print_header{
    # prepare a cookie
  my $c = Apache::Cookie->new
    ($r,
     -name    => 'sessionID',
     -value   => $sessionID,
     -expires => '+1h');

    # Add a Set-Cookie header to the outgoing headers table
  $c->bake;

  $r->send_http_header('text/html');

} # end of sub print_header


# print the current Session status and a form to toggle the status
#################
sub print_status{

  print qq{<HTML><HEAD><TITLE>Cookie</TITLE></HEAD><BODY>};

    # print status
  print "<B>Status:</B> ",
        $status
          ? "Session is running with ID: $sessionID"

          : "No session is running";


    # change status form
```

```
    my $button_label = $status ? "Stop" : "Start";
    print qq{<HR>
         <FORM>
           <INPUT TYPE=SUBMIT NAME=switch VALUE=" $button_label ">
         </FORM>
           };

    print qq{</BODY></HTML>};

  } # end of sub print_status

  # replace with a real session ID generator
  ########################
  sub generate_sessionID {
    return scalar localtime;
  }

  1;
```

The only other changes are in the print_header() function, where instead of passing the cookie code to the `CGI`'s header() to return a proper HTTP header:

```
    print $q->header
      (-type   => 'text/html',
       -cookie => $c);
```

we do it in two stages.

```
    $c->bake;
```

Adds a `Set-Cookie` header to the outgoing headers table, and:

```
    $r->send_http_header('text/html');
```

sends out the header itself. We have also eliminated:

```
    Apache->request($r);
```

since we don't rely on `CGI.pm` any more and in this case we don't need it.

The rest of the code is unchanged.

Of course we add the following snippet to *httpd.conf*:

```
  PerlModule Test::Cookie2
  <Location /test/cookie2>
    SetHandler perl-script
    PerlHandler Test::Cookie2
  </Location>
```

So now the magic URI that will trigger the above code execution will be the one starting with */test/cookie2* . We save the code in the file */home/httpd/perl/Test/Cookie2.pm* since we have called this package `Test::Cookie2`.

5.39.4 Conclusion

If your took care to write the original plain CGI script's code in a clean and modular way, you can see that the transition is a very simple one and doesn't take a lot of effort. Almost no code was modified.

5.40 Maintainers

Maintainer is the person(s) you should contact with updates, corrections and patches.

- Stas Bekman [http://stason.org/]

5.41 Authors

- Stas Bekman [http://stason.org/]

Only the major authors are listed above. For contributors see the Changes file.

6 How to use mod_perl’s Method Handlers

6.1 Description

Described here are a few examples and hints on how to use method handlers with mod_perl.

This document assumes familiarity with at least the *perltoot* manpage and "normal" usage of the `Perl*Handlers`.

It isn’t strictly mod_perl related, more like "what I use objects for in my mod_perl environment".

6.2 Synopsis

If a **Perl*Handler** is prototyped with ’`$$`’, this handler will be invoked as method, being passed a class name or blessed object as its first argument and the blessed *request_rec* as the second argument, e.g.

```
package My;
@ISA = qw(BaseClass);

sub handler ($$) {
    my ($class, $r) = @_;
    ...;
}

package BaseClass;

sub method ($$) {
    my ($class, $r) = @_;
    ...;
}

__END__
```

Configuration:

```
PerlHandler My
```

or

```
PerlHandler My->handler
```

Since the handler is invoked as a method, it may inherit from other classes:

```
PerlHandler My->method
```

In this case, the ’`My`’ class inherits this method from ’`BaseClass`’.

To build in this feature, configure with:

```
% perl Makefile.PL PERL_METHOD_HANDLERS=1 [PERL_FOO_HOOK=1,etc]
```

6.3 Why?

The short version: For pretty much the same reasons we're using OO perl everywhere else. :-) See the *perltoot* manpage.

The slightly longer version would include some about code reusage and more clean interface between modules.

6.4 Simple example

Let's start with a simple example.

In httpd.conf:

```
<Location /obj-handler>
SetHandler perl-script
PerlHandler $My::Obj->method
</Location>
```

In startup.pl or another PerlRequire'd file:

```
package This::Class;

$My::Obj = bless {};

sub method ($$) {
    my ($obj, $r) = @_;
    $r->send_http_header("text/plain");
    print "$obj isa ", ref($obj);
    0;
}
```

which displays:

```
This::Class=HASH(0x8411edc) isa This::Class
```

6.5 A little more advanced

That wasn't really useful, so let's try something little more advanced.

I've a little module which creates a graphical 'datebar' for a client. It's reading a lot of small gifs with numbers and weekdays, and keeping them in memory in GD.pm's native format, ready to be copied together and served as gifs.

Now I wanted to use it at another site too, but with a different look. Obviously something to do with a object. Hence I changed the module to a object, and can now do a

```
$Client1::Datebar = new Datebar(
        -imagepath => '/home/client1/datebar/',
        -size      => [131,18],
        -elements  => 'wday mday mon year hour min',
);

$Client2::Datebar = new Datebar
        -imagepath => '/home/client2/datebar/',
        -size      => [90,14],
        -elements  => 'wday hour min',
);
```

And then use `$Client1::Datebar` and `$Client2::Datebar` as PerlHandlers in my Apache configuration. Remember to pass them in literal quotes (’’) and not "" which will be interpolated!

I've a webinterface system to our content-database. I've created objects to handle the administration of articles, banners, images and other content. It's then very easy (a few lines of code) to enable certain modules for each client, depending on their needs.

Another area where I use objects with great success in my modperl configurations is database abstraction. All our clients using the webinterface to handle f.x. articles will use a simple module to handle everything related to the database. Each client have

```
$Client::Article = new WebAjour::Article(-host => 'www.client.com');
```

in a module what will be run at server startup.

I can then use some simple methods from the $Client::Article object in my embperl documents, like:

```
[- $c = $Client::Article->GetCursor(-layout=>'Frontpage') -]
[$ while($c->Fetch) $]
  <h2>[+ $c->f('header') +]</h2>
  [+ $c->f('textfield') +]
[$ endwhile $]
```

Very very useful!

6.6 Traps

mod_perl expects object handlers to be in the form of a string, which it will thaw for you. That means that something like

```
$r->push_handlers(PerlHandler => '$self->perl_handler_method');
```

This doesn't work as you might expect, since Perl isn't able to see $self once it goes to PerlHandler.

The best solution to this is to use an anonymous subroutine and pass it $r yourself, like this:

```
$r->push_handlers(PerlHandler =>
    sub {
        my $r = shift;
        $self->perl_handler_method($r);
    }
);
```

6.7 Author

This document is written by Ask Bjoern Hansen <ask@netcetera.dk> or <ask@apache.org>. Corrections and suggestions are most welcome. In particular would more examples be appreciated, most of my own code is way too integrated with our system, which isn't suitable for public release.

Some codesnippets is from Doug MacEachern.

6.8 See Also

The *Apache*, the *perltoot* manpages (also available at `http://www.perl.com/CPAN/doc/FMTEYEWTK/perltoot.html`)

6.9 Maintainers

Maintainer is the person(s) you should contact with updates, corrections and patches.

- modperl docs list

6.10 Authors

- Ask Bjoern Hansen, <ask (at) netcetera.dk> or <ask (at) apache.org>.

Only the major authors are listed above. For contributors see the Changes file.

7 mod_perl and Relational Databases

7.1 Description

Creating dynamic websites with mod_perl often involves using relational databases. `Apache::DBI`, which provides a database connections persistence which boosts the mod_perl performance, is explained in this chapter.

7.2 Why Relational (SQL) Databases

Nowadays millions of people surf the Internet. There are millions of Terabytes of data lying around. To manipulate the data new smart techniques and technologies were invented. One of the major inventions was the relational database, which allows us to search and modify huge stores of data very quickly. We use **SQL** (Structured Query Language) to access and manipulate the contents of these databases.

7.3 Apache::DBI - Initiate a persistent database connection

When people started to use the web, they found that they needed to write web interfaces to their databases. CGI is the most widely used technology for building such interfaces. The main limitation of a CGI script driving a database is that its database connection is not persistent - on every request the CGI script has to re-connect to the database, and when the request is completed the connection is closed.

`Apache::DBI` was written to remove this limitation. When you use it, you have a database connection which persists for the process' entire life. So when your mod_perl script needs to use a database, `Apache::DBI` provides a valid connection immediately and your script starts work right away without having to initiate a database connection first.

This is possible only with CGI running under a mod_perl enabled server, since in this model the child process does not quit when the request has been served.

It's almost as straightforward as is it sounds; there are just a few things to know about and we will cover them in this section.

7.3.1 Introduction

The DBI module can make use of the `Apache::DBI` module. When it loads, the DBI module tests if the environment variable `$ENV{MOD_PERL}` is set, and if the `Apache::DBI` module has already been loaded. If so, the DBI module will forward every connect() request to the `Apache::DBI` module. `Apache::DBI` uses the ping() method to look for a database handle from a previous connect() request, and tests if this handle is still valid. If these two conditions are fulfilled it just returns the database handle.

If there is no appropriate database handle or if the ping() method fails, `Apache::DBI` establishes a new connection and stores the handle for later re-use. When the script is run again by a child that is still connected, `Apache::DBI` just checks the cache of open connections by matching the *host*, *username* and *password* parameters against it. A matching connection is returned if available or a new one is initiated and then returned.

There is no need to delete the disconnect() statements from your code. They won't do anything because the `Apache::DBI` module overloads the disconnect() method with an empty one.

7.3.2 When should this module be used and when shouldn't it be used?

You will want to use this module if you are opening several database connections to the server. `Apache::DBI` will make them persistent per child, so if you have ten children and each opens two different connections (with different connect() arguments) you will have in total twenty opened and persistent connections. After the initial connect() you will save the connection time for every connect() request from your `DBI` module. This can be a huge benefit for a server with a high volume of database traffic.

You must **not** use this module if you are opening a special connection for each of your users (meaning that the login arguments are different for each user). Each connection will stay persistent and after a certain period the number of open connections will reach the allowed limit (configured by the database server) and new database connection opening requests will be refused, rendering your service unusable for some of your users.

If you want to use `Apache::DBI` but you have both situations on one machine, at the time of writing the only solution is to run two Apache/mod_perl servers, one which uses `Apache::DBI` and one which does not.

7.3.3 Configuration

After installing this module, the configuration is simple - add the following directive to `httpd.conf`

```
PerlModule Apache::DBI
```

Note that it is important to load this module before any other `Apache*DBI` module and before the `DBI` module itself!

You can skip preloading `DBI`, since `Apache::DBI` does that. But there is no harm in leaving it in, as long as it is loaded after `Apache::DBI`.

7.3.4 Preopening DBI connections

If you want to make sure that a connection will already be opened when your script is first executed after a server restart, then you should use the `connect_on_init()` method in the startup file to preload every connection you are going to use. For example:

```
Apache::DBI->connect_on_init
("DBI:mysql:myDB:myserver",
 "username",
 "passwd",
 {
  PrintError => 1, # warn() on errors
  RaiseError => 0, # don't die on error
  AutoCommit => 1, # commit executes immediately
 }
);
```

As noted above, use this method only if you want all of apache to be able to connect to the database server as one user (or as a very few users), i.e. if your user(s) can effectively share the connection. Do **not** use this method if you want for example one unique connection per user.

Be warned though, that if you call `connect_on_init()` and your database is down, Apache children will be delayed at server startup, trying to connect. They won't begin serving requests until either they are connected, or the connection attempt fails. Depending on your DBD driver, this can take several minutes!

7.3.5 Debugging Apache::DBI

If you are not sure if this module is working as advertised, you should enable Debug mode in the startup script by:

```
$Apache::DBI::DEBUG = 1;
```

Starting with `ApacheDBI-0.84`, setting `$Apache::DBI::DEBUG = 1` will produce only minimal output. For a full trace you should set `$Apache::DBI::DEBUG = 2`.

After setting the DEBUG level you will see entries in the `error_log` both when `Apache::DBI` initializes a connection and when it returns one from its cache. Use the following command to view the log in real time (your `error_log` might be located at a different path, it is set in the Apache configuration files):

```
tail -f /usr/local/apache/logs/error_log
```

I use `alias` (in `tcsh`) so I do not have to remember the path:

```
alias err "tail -f /usr/local/apache/logs/error_log"
```

7.3.6 Database Locking Risks

Be very careful when locking the database (`LOCK TABLE ...`) or singular rows if you use `Apache::DBI` or similar persistent connections. MySQL threads keep tables locked until the thread ends (connection is closed) or the tables are unlocked. If your session die()'s while tables are locked, they will stay neatly locked as your connection won't be closed either.

See the section Handling the 'User pressed Stop button' case for more information on prevention.

7.3.7 Troubleshooting

7.3.7.1 The Morning Bug

The SQL server keeps a connection to the client open for a limited period of time. In the early days of `Apache::DBI` developers were bitten by so called *Morning bug*, when every morning the first users to use the site received a `No Data Returned` message, but after that everything worked fine.

The error was caused by `Apache::DBI` returning a handle of the invalid connection (the server closed it because of a timeout), and the script was dying on that error. The `ping()` method was introduced to solve this problem, but it didn't worked properly till `Apache::DBI` version 0.82 was released. In that version and afterwards ping() was called inside the `eval` block, which resolved the problem.

It's possible that some `DBD::` drivers don't have the ping() method implemented. The `Apache::DBI` manpage explains how to write one.

Another solution was found - to increase the timeout parameter when starting the database server. Currently we startup `MySQL` server with a script `safe_mysql`, so we have modified it to use this option:

```
nohup $ledir/mysqld [snipped other options] -O wait_timeout=172800
```

172800 seconds is equal to 48 hours. This change solves the problem, but the ping() method works properly in `DBD::mysql` as well.

7.3.7.2 Opening Connections With Different Parameters

When `Apache::DBI` receives a connection request, before it decides to use an existing cached connection it insists that the new connection be opened in exactly the same way as the cached connection. If you have one script that sets `AutoCommit` and one that does not, `Apache::DBI` will make two different connections. So if for example you have limited Apache to 40 servers at most, instead of having a maximum of 40 open connections you may end up with 80.

So these two connect() calls will create two different connections:

```
my $dbh = DBI->connect
    ("DBI:mysql:test:localhost", '', '',
     {
      PrintError => 1, # warn() on errors
      RaiseError => 0, # don't die on error
      AutoCommit => 1, # commit executes immediately
     }
    ) or die "Cannot connect to database: $DBI::errstr";

my $dbh = DBI->connect
    ("DBI:mysql:test:localhost", '', '',
     {
      PrintError => 1, # warn() on errors
      RaiseError => 0, # don't die on error
      AutoCommit => 0, # don't commit executes immediately
     }
    ) or die "Cannot connect to database: $DBI::errstr";
```

Notice that the only difference is in the value of `AutoCommit`.

However, you are free to modify the handle immediately after you get it from the cache. So always initiate connections using the same parameters and set `AutoCommit` (or whatever) afterwards. Let's rewrite the second connect call to do the right thing (not to create a new connection):

```
my $dbh = DBI->connect
    ("DBI:mysql:test:localhost", '', '',
     {
      PrintError => 1, # warn() on errors
      RaiseError => 0, # don't die on error
      AutoCommit => 1, # commit executes immediately
     }
    ) or die "Cannot connect to database: $DBI::errstr";
$dbh->{AutoCommit} = 0; # don't commit if not asked to
```

When you aren't sure whether you're doing the right thing, turn debug mode on.

However, when the `$dbh` attribute is altered after connect() it affects all other handlers retrieving this database handle. Therefore it's best to restore the modified attributes to their original value at the end of database handle usage. As of `Apache::DBI` version 0.88 the caller has to do it manually. The simplest way to handle this is to localize the attributes when modifying them:

```
my $dbh = DBI->connect(...) ...
{
  local $dbh->{LongReadLen} = 40;
}
```

Here the `LongReadLen` attribute overrides the value set in the connect() call or its default value only within the enclosing block.

The problem with this approach is that prior to Perl version 5.8.0 this causes memory leaks. So the only clean alternative for older Perl versions is to manually restore the `dbh`'s values:

```
my @attrs = qw(LongReadLen PrintError);
my %orig = ();

my $dbh = DBI->connect(...) ...
# store the values away
$orig{$_} = $dbh->{$_} for @attrs;
# do local modifications
$dbh->{LongReadLen} = 40;
$dbh->{PrintError}  = 1;
# do something with the filehandle
# ...
# now restore the values
$dbh->{$_} = $orig{$_} for @attrs;
```

Another thing to remember is that with some database servers it's possible to access more than one database using the same database connection. MySQL is one of those servers. It allows you to use a fully qualified table specification notation. So if there is a database *foo* with a table *test* and database *bar* with its own table *test*, you can always use:

```
SELECT from foo.test ...
```

or:

```
SELECT from bar.test ...
```

So no matter what database you have used in the database name string in the connect() call (e.g.: `DBI:mysql:foo:localhost`) you can still access both tables by using a fully qualified syntax.

Alternatively you can switch databases with `USE foo` and `USE bar`, but this approach seems less convenient, and therefore error-prone.

7.3.7.3 Cannot find the DBI handler

You must use `DBI::connect()` as in normal DBI usage to get your $dbh database handler. Using the `Apache::DBI` does not eliminate the need to write proper `DBI` code. As the `Apache::DBI` man page states, you should program as if you are not using `Apache::DBI` at all. `Apache::DBI` will override the DBI methods where necessary and return your cached connection. Any `disconnect()` call will be just ignored.

7.3.7.4 Apache:DBI does not work

Make sure you have it installed.

Make sure you configured mod_perl with either:

```
PERL_CHILD_INIT=1 PERL_STACKED_HANDLERS=1
```

or

```
EVERYTHING=1
```

Use the example script eg/startup.pl (in the mod_perl distribution). Remove the comment from the line.

```
# use Apache::DebugDBI;
```

and adapt the connect string. Do not change anything in your scripts for use with `Apache::DBI`.

7.3.7.5 Skipping connection cache during server startup

Does your error_log look like this?

```
10169 Apache::DBI PerlChildInitHandler
10169 Apache::DBI skipping connection cache during server startup
Database handle destroyed without explicit disconnect at
/usr/lib/perl5/site_perl/5.005/Apache/DBI.pm line 29.
```

If so you are trying to open a database connection in the parent httpd process. If you do, children will each get a copy of this handle, causing clashes when the handle is used by two processes at the same time. Each child must have its own, unique, connection handle.

To avoid this problem, `Apache::DBI` checks whether it is called during server startup. If so the module skips the connection cache and returns immediately without a database handle.

You must use the `Apache::DBI->connect_on_init()` method in the startup file.

7.3.7.6 Debugging code which deploys DBI

To log a trace of `DBI` statement execution, you must set the `DBI_TRACE` environment variable. The `PerlSetEnv DBI_TRACE` directive must appear before you load `Apache::DBI` and `DBI`.

For example if you use `Apache::DBI`, modify your `httpd.conf` with:

```
PerlSetEnv DBI_TRACE "3=/tmp/dbitrace.log"
PerlModule Apache::DBI
```

Replace `3` with the TRACE level you want. The traces from each request will be appended to `/tmp/dbitrace.log`. Note that the logs might interleave if requests are processed concurrently.

Within your code you can control trace generation with the trace() method:

```
DBI->trace($trace_level)
DBI->trace($trace_level, $trace_filename)
```

DBI trace information can be enabled for all handles using this DBI class method. To enable trace information for a specific handle use the similar `$h->trace` method.

Using the handle trace option with a `$dbh` or `$sth` is handy for limiting the trace info to the specific bit of code that you are interested in.

Trace Levels:

- **0 - trace disabled.**
- **1 - trace DBI method calls returning with results.**
- **2 - trace method entry with parameters and exit with results.**
- **3 - as above, adding some high-level information from the driver and also adding some internal information from the DBI.**
- **4 - as above, adding more detailed information from the driver and also including DBI mutex information when using threaded perl.**
- **5 and above - as above but with more and more obscure information.**

7.4 mysql_use_result vs. mysql_store_result.

Since many mod_perl developers use mysql as their preferred SQL engine, these notes explain the difference between `mysql_use_result()` and `mysql_store_result()`. The two influence the speed and size of the processes.

The `DBD::mysql` (version 2.0217) documentation includes the following snippet:

```
mysql_use_result attribute: This forces the driver to use
mysql_use_result rather than mysql_store_result. The former is
faster and less memory consuming, but tends to block other
processes. (That's why mysql_store_result is the default.)
```

Think about it in client/server terms. When you ask the server to spoon-feed you the data as you use it, the server process must buffer the data, tie up that thread, and possibly keep any database locks open for a long time. So if you read a row of data and ponder it for a while, the tables you have locked are still locked, and the server is busy talking to you every so often. That is `mysql_use_result()`.

If you just suck down the whole dataset to the client, then the server is free to go about its business serving other requests. This results in parallelism since the server and client are doing work at the same time, rather than blocking on each other doing frequent I/O. That is `mysql_store_result()`.

As the mysql manual suggests: you should not use `mysql_use_result()` if you are doing a lot of processing for each row on the client side. This can tie up the server and prevent other threads from updating the tables.

7.5 Transactions Not Committed with MySQL InnoDB Tables

Sometimes, when using MySQL's InnoDB table type, you may notice that changes you committed in one process don't seem to be visible to other processes. You may not be aware that InnoDB tables use a default approach to transactions that is actually more cautious than PostgreSQL or Oracle's default. It's called "repeatable read", and the gist of it is that you don't see updates made in other processes since your last commit. There is an explanation of this here: http://dev.mysql.com/doc/mysql/en/InnoDB_Consistent_read_example.html

This is actually not directly related to mod_perl, but you wouldn't notice this issue when using CGI because reconecting to the database on each request resets things just as doing a commit does. It is the persistent connections used with mod_perl that make this issue visible.

If you suspect this is causing you problems, the simplest way to deal with it is to change the isolation level to "read committed" -- which is more like what PostgreSQL and Oracle do by default -- with the "set transaction" command, described here: http://dev.mysql.com/doc/mysql/en/InnoDB_transaction_isolation.html

7.6 Optimize: Run Two SQL Engine Servers

Sometimes you end up running many databases on the same machine. These might have very varying database needs (such as one db with sessions, very frequently updated but tiny amounts of data, and another with large sets of data that's hardly ever updated) you might be able to gain a lot by running two differently configured databases on one server. One would benefit from lots of caching, the other would probably reduce the efficiency of the cache but would gain from fast disk access. Different usage profiles require vastly different performance needs.

This is basically a similar idea to having two Apache servers, each optimized for its specific requirements.

7.7 Some useful code snippets to be used with relational Databases

In this section you will find scripts, modules and code snippets to help you get started using relational Databases with mod_perl scripts. Note that I work with `mysql` (http://www.mysql.com), so the code you find here will work out of box with mysql. If you use some other SQL engine, it might work for you or it might need some changes. YMMV.

7.7.1 Turning SQL query writing into a short and simple task

Having to write many queries in my CGI scripts, persuaded me to write a stand alone module that saves me a lot of time in coding and debugging my code. It also makes my scripts much smaller and easier to read. I will present the module here, with examples following:

Notice the `DESTROY` block at the end of the module, which makes various cleanups and allows this module to be used under mod_perl and `mod_cgi` as well. Note that you will not get the benefit of persistent database handles with mod_cgi.

7.7.2 The My::DB module

The *code/My-DB.pm*:

```
package My::DB;

use strict;
use 5.004;

use DBI;

use vars qw(%c);
use constant DEBUG => 0;

%c =
  (
   db => {
          DB_NAME      => 'foo',
          SERVER       => 'localhost',
          USER         => 'put_username_here',
          USER_PASSWD  => 'put_passwd_here',
         },

  );

use Carp qw(croak verbose);
#local $SIG{__WARN__} = \&Carp::cluck;

# untaint the path by explicit setting
local $ENV{PATH} = '/bin:/usr/bin';
```

```
#######
sub new {
  my $proto = shift;
  my $class = ref($proto) || $proto;
  my $self  = {};

    # connect to the DB, Apache::DBI takes care of caching the connections
    # save into a dbh - Database handle object
  $self->{dbh} = DBI->connect("DBI:mysql:$c{db}{DB_NAME}::$c{db}{SERVER}",
                              $c{db}{USER},
                              $c{db}{USER_PASSWD},
                              {
                               PrintError => 1, # warn() on errors
                               RaiseError => 0, # don't die on error
                               AutoCommit => 1, # commit executes immediately
                              }
                             )
    or die "Cannot connect to database: $DBI::errstr";

    # we want to die on errors if in debug mode
  $self->{dbh}->{RaiseError} = 1 if DEBUG;

    # init the sth - Statement handle object
  $self->{sth} = '';

  bless ($self, $class);

  $self;

} # end of sub new



##############################################################################
               ####################################
               ###                              ###
               ###       SQL Functions          ###
               ###                              ###
               ####################################
##############################################################################

# print debug messages
sub d{
   # we want to print the trace in debug mode
  print "".join("", @_)."\n" if DEBUG;

} # end of sub d


##############################################################################
# return a count of matched rows, by conditions
#
#  $count = sql_count_matched($table_name,\@conditions,\@restrictions);
#
# conditions must be an array so we can pass more than one column with
# the same name.
#
#  @conditions =  ( column => ['comp_sign','value'],
#                   foo    => ['>',15],
```

```
#                  foo    => ['<',30],
#                );
#
# The sub knows automatically to detect and quote strings
#
# Restrictions are the list of restrictions like ('order by email')
#
##########################
sub sql_count_matched{
  my $self    = shift;
  my $table   = shift || '';
  my $r_conds = shift || [];
  my $r_restr = shift || [];

    # we want to print the trace in debug mode
  d( "[".(caller(2))[3]." - ".(caller(1))[3]." - ". (caller(0))[3]."]") if DEBUG;

    # build the query
  my $do_sql = "SELECT COUNT(*) FROM $table ";
  my @where = ();
  for(my $i=0;$i<@{$r_conds};$i=$i+2) {
    push @where, join " ",
      $$r_conds[$i],
      $$r_conds[$i+1][0],
      sql_quote(sql_escape($$r_conds[$i+1][1]));
  }
    # Add the where clause if we have one
  $do_sql .= "WHERE ". join " AND ", @where if @where;

    # restrictions (DONT put commas!)
  $do_sql .= " ". join " ", @{$r_restr} if @{$r_restr};

  d("SQL: $do_sql") if DEBUG;

    # do query
  $self->{sth} = $self->{dbh}->prepare($do_sql);
  $self->{sth}->execute();
  my ($count) = $self->{sth}->fetchrow_array;

  d("Result: $count") if DEBUG;

  $self->{sth}->finish;

  return $count;

} # end of sub sql_count_matched


#############################################################################
# return a count of matched distinct rows, by conditions
#
#  $count = sql_count_matched_distinct($table_name,\@conditions,\@restrictions);
#
# conditions must be an array so we can path more than one column with
# the same name.
#
#  @conditions =  ( column => ['comp_sign','value'],
#                   foo    => ['>',15],
#                   foo    => ['<',30],
```

```
#                );
#
# The sub knows automatically to detect and quote strings
#
# Restrictions are the list of restrictions like ('order by email')
#
# This a slow implementation - because it cannot use select(*), but
# brings all the records in first and then counts them. In the next
# version of mysql there will be an operator 'select (distinct *)'
# which will make things much faster, so we will just change the
# internals of this sub, without changing the code itself.
#
##############################
sub sql_count_matched_distinct{
  my $self    = shift;
  my $table   = shift || '';
  my $r_conds = shift || [];
  my $r_restr = shift || [];

    # we want to print the trace in debug mode
  d( "[".(caller(2))[3]." - ".(caller(1))[3]." - ". (caller(0))[3]."]") if DEBUG;

    # build the query
  my $do_sql = "SELECT DISTINCT * FROM $table ";
  my @where = ();
  for(my $i=0;$i<@{$r_conds};$i=$i+2) {
    push @where, join " ",
      $$r_conds[$i],
      $$r_conds[$i+1][0],
      sql_quote(sql_escape($$r_conds[$i+1][1]));
  }
    # Add the where clause if we have one
  $do_sql .= "WHERE ". join " AND ", @where if @where;

    # restrictions (DONT put commas!)
  $do_sql .= " ". join " ", @{$r_restr} if @{$r_restr};

  d("SQL: $do_sql") if DEBUG;

    # do query
#  $self->{sth} = $self->{dbh}->prepare($do_sql);
#  $self->{sth}->execute();

  my $count = @{$self->{dbh}->selectall_arrayref($do_sql)};

#  my ($count) = $self->{sth}->fetchrow_array;

  d("Result: $count") if DEBUG;

#  $self->{sth}->finish;

  return $count;

} # end of sub sql_count_matched_distinct



#########################################################################
# return a single (first) matched value or undef, by conditions and
```

```
# restrictions
#
# sql_get_matched_value($table_name,$column,\@conditions,\@restrictions);
#
# column is a name of the column
#
# conditions must be an array so we can path more than one column with
# the same name.
#  @conditions =  ( column => ['comp_sign','value'],
#                   foo    => ['>',15],
#                   foo    => ['<',30],
#                 );
# The sub knows automatically to detect and quote strings
#
# restrictions is a list of restrictions like ('order by email')
#
############################
sub sql_get_matched_value{
  my $self    = shift;
  my $table   = shift || '';
  my $column  = shift || '';
  my $r_conds = shift || [];
  my $r_restr = shift || [];

    # we want to print in the trace debug mode
  d( "[".(caller(2))[3]." - ".(caller(1))[3]." - ". (caller(0))[3]."]") if DEBUG;

    # build the query
  my $do_sql = "SELECT $column FROM $table ";

  my @where = ();
  for(my $i=0;$i<@{$r_conds};$i=$i+2) {
    push @where, join " ",
      $$r_conds[$i],
      $$r_conds[$i+1][0],
      sql_quote(sql_escape($$r_conds[$i+1][1]));
  }
    # Add the where clause if we have one
  $do_sql .= " WHERE ". join " AND ", @where if @where;

    # restrictions (DONT put commas!)
  $do_sql .= " ". join " ", @{$r_restr} if @{$r_restr};

  d("SQL: $do_sql") if DEBUG;

    # do query
  return $self->{dbh}->selectrow_array($do_sql);

} # end of sub sql_get_matched_value




#############################################################################
# return a single row of first matched rows, by conditions and
# restrictions. The row is being inserted into @results_row array
# (value1,value2,...)  or empty () if none matched
#
# sql_get_matched_row(\@results_row,$table_name,\@columns,\@conditions,\@restrictions);
```

```
#
# columns is a list of columns to be returned (username, fname,...)
#
# conditions must be an array so we can path more than one column with
# the same name.
#  @conditions =  ( column => ['comp_sign','value'],
#                   foo    => ['>',15],
#                   foo    => ['<',30],
#                 );
# The sub knows automatically to detect and quote strings
#
# restrictions is a list of restrictions like ('order by email')
#
############################
sub sql_get_matched_row{
  my $self    = shift;
  my $r_row   = shift || {};
  my $table   = shift || '';
  my $r_cols  = shift || [];
  my $r_conds = shift || [];
  my $r_restr = shift || [];

    # we want to print in the trace debug mode
  d( "[".(caller(2))[3]." - ".(caller(1))[3]." - ". (caller(0))[3]."]") if DEBUG;

    # build the query
  my $do_sql = "SELECT ";
  $do_sql .= join ",", @{$r_cols} if @{$r_cols};
  $do_sql .= " FROM $table ";

  my @where = ();
  for(my $i=0;$i<@{$r_conds};$i=$i+2) {
    push @where, join " ",
      $$r_conds[$i],
      $$r_conds[$i+1][0],
      sql_quote(sql_escape($$r_conds[$i+1][1]));
  }
    # Add the where clause if we have one
  $do_sql .= " WHERE ". join " AND ", @where if @where;

    # restrictions (DONT put commas!)
  $do_sql .= " ". join " ", @{$r_restr} if @{$r_restr};

  d("SQL: $do_sql") if DEBUG;

    # do query
  @{$r_row} = $self->{dbh}->selectrow_array($do_sql);

} # end of sub sql_get_matched_row



############################################################################
# return a ref to hash of single matched row, by conditions
# and restrictions. return undef if nothing matched.
# (column1 => value1, column2 => value2) or empty () if non matched
#
# sql_get_hash_ref($table_name,\@columns,\@conditions,\@restrictions);
#
```

```
# columns is a list of columns to be returned (username, fname,...)
#
# conditions must be an array so we can path more than one column with
# the same name.
#  @conditions =  ( column => ['comp_sign','value'],
#                   foo    => ['>',15],
#                   foo    => ['<',30],
#                 );
# The sub knows automatically to detect and quote strings
#
# restrictions is a list of restrictions like ('order by email')
#
############################
sub sql_get_hash_ref{
  my $self    = shift;
  my $table   = shift || '';
  my $r_cols  = shift || [];
  my $r_conds = shift || [];
  my $r_restr = shift || [];

    # we want to print in the trace debug mode
  d( "[".(caller(2))[3]." - ".(caller(1))[3]." - ". (caller(0))[3]."]") if DEBUG;

    # build the query
  my $do_sql = "SELECT ";
  $do_sql .= join ",", @{$r_cols} if @{$r_cols};
  $do_sql .= " FROM $table ";

  my @where = ();
  for(my $i=0;$i<@{$r_conds};$i=$i+2) {
    push @where, join " ",
      $$r_conds[$i],
      $$r_conds[$i+1][0],
      sql_quote(sql_escape($$r_conds[$i+1][1]));
  }
    # Add the where clause if we have one
  $do_sql .= " WHERE ". join " AND ", @where if @where;

    # restrictions (DONT put commas!)
  $do_sql .= " ". join " ", @{$r_restr} if @{$r_restr};

  d("SQL: $do_sql") if DEBUG;

    # do query
  $self->{sth} = $self->{dbh}->prepare($do_sql);
  $self->{sth}->execute();

  return $self->{sth}->fetchrow_hashref;

} # end of sub sql_get_hash_ref




#############################################################################
# returns a reference to an array, matched by conditions and
# restrictions, which contains one reference to array per row. If
# there are no rows to return, returns a reference to an empty array:
```

```
# [
#  [array1],
#   ......
#  [arrayN],
# ];
#
# $ref = sql_get_matched_rows_ary_ref($table_name,\@columns,\@conditions,\@restrictions);
#
# columns is a list of columns to be returned (username, fname,...)
#
# conditions must be an array so we can path more than one column with
# the same name. @conditions are being cancatenated with AND
#  @conditions =  ( column => ['comp_sign','value'],
#                   foo    => ['>',15],
#                   foo    => ['<',30],
#                 );
# results in
# WHERE foo > 15 AND foo < 30
#
#  to make an OR logic use (then ANDed )
#  @conditions =  ( column => ['comp_sign',['value1','value2']],
#                   foo    => ['=',[15,24] ],
#                   bar    => ['=',[16,21] ],
#                 );
# results in
# WHERE (foo = 15 OR foo = 24) AND (bar = 16 OR bar = 21)
#
# The sub knows automatically to detect and quote strings
#
# restrictions is a list of restrictions like ('order by email')
#
############################
sub sql_get_matched_rows_ary_ref{
  my $self    = shift;
  my $table   = shift || '';
  my $r_cols  = shift || [];
  my $r_conds = shift || [];
  my $r_restr = shift || [];

    # we want to print in the trace debug mode
  d( "[".(caller(2))[3]." - ".(caller(1))[3]." - ". (caller(0))[3]."]") if DEBUG;

    # build the query
  my $do_sql = "SELECT ";
  $do_sql .= join ",", @{$r_cols} if @{$r_cols};
  $do_sql .= " FROM $table ";

  my @where = ();
  for(my $i=0;$i<@{$r_conds};$i=$i+2) {

    if (ref $$r_conds[$i+1][1] eq 'ARRAY') {
        # multi condition for the same field/comparator to be ORed
      push @where, map {"($_)"} join " OR ",
        map { join " ",
                $r_conds->[$i],
                $r_conds->[$i+1][0],
                sql_quote(sql_escape($_));
            } @{$r_conds->[$i+1][1]};
    } else {
```

```
        # single condition for the same field/comparator
       push @where, join " ",
         $r_conds->[$i],
         $r_conds->[$i+1][0],
         sql_quote(sql_escape($r_conds->[$i+1][1]));
     }
   } # end of for(my $i=0;$i<@{$r_conds};$i=$i+2

     # Add the where clause if we have one
   $do_sql .= " WHERE ". join " AND ", @where if @where;

     # restrictions (DONT put commas!)
   $do_sql .= " ". join " ", @{$r_restr} if @{$r_restr};

   d("SQL: $do_sql") if DEBUG;

     # do query
   return $self->{dbh}->selectall_arrayref($do_sql);

} # end of sub sql_get_matched_rows_ary_ref



###########################################################################
# insert a single row into a DB
#
#  sql_insert_row($table_name,\%data,$delayed);
#
# data is hash of type (column1 => value1 ,column2 => value2 , )
#
# $delayed: 1 => do delayed insert, 0 or none passed => immediate
#
# * The sub knows automatically to detect and quote strings
#
# * The insert id delayed, so the user will not wait untill the insert
# will be completed, if many select queries are running
#
############################
sub sql_insert_row{
   my $self    = shift;
   my $table   = shift || '';
   my $r_data = shift || {};
   my $delayed = (shift) ? 'DELAYED' : '';

     # we want to print in the trace debug mode
   d( "[".(caller(2))[3]." - ".(caller(1))[3]." - ". (caller(0))[3]."]") if DEBUG;

     # build the query
   my $do_sql = "INSERT $delayed INTO $table ";
   $do_sql   .= "(".join(",",keys %{$r_data}).")";
   $do_sql   .= " VALUES (";
   $do_sql   .= join ",", sql_quote(sql_escape( values %{$r_data} ) );
   $do_sql   .= ")";

   d("SQL: $do_sql") if DEBUG;

     # do query
   $self->{sth} = $self->{dbh}->prepare($do_sql);
```

```
  $self->{sth}->execute();

} # end of sub sql_insert_row


#########################################################################
# update rows in a DB by condition
#
#  sql_update_rows($table_name,\%data,\@conditions,$delayed);
#
# data is hash of type (column1 => value1 ,column2 => value2 , )
#
# conditions must be an array so we can path more than one column with
# the same name.
#  @conditions =  ( column => ['comp_sign','value'],
#                   foo    => ['>',15],
#                   foo    => ['<',30],
#                 );
#
# $delayed: 1 => do delayed insert, 0 or none passed => immediate
#
# * The sub knows automatically to detect and quote strings
#
#
############################
sub sql_update_rows{
  my $self    = shift;
  my $table   = shift || '';
  my $r_data = shift || {};
  my $r_conds = shift || [];
  my $delayed = (shift) ? 'LOW_PRIORITY' : '';

    # we want to print in the trace debug mode
  d( "[".(caller(2))[3]." - ".(caller(1))[3]." - ". (caller(0))[3]."]") if DEBUG;

    # build the query
  my $do_sql = "UPDATE $delayed $table SET ";
  $do_sql   .= join ",",
    map { "$_=".join "",sql_quote(sql_escape($$r_data{$_})) } keys %{$r_data};

  my @where = ();
  for(my $i=0;$i<@{$r_conds};$i=$i+2) {
    push @where, join " ",
      $$r_conds[$i],
      $$r_conds[$i+1][0],
      sql_quote(sql_escape($$r_conds[$i+1][1]));
  }
    # Add the where clause if we have one
  $do_sql .= " WHERE ". join " AND ", @where if @where;


  d("SQL: $do_sql") if DEBUG;

    # do query
  $self->{sth} = $self->{dbh}->prepare($do_sql);

  $self->{sth}->execute();

#  my ($count) = $self->{sth}->fetchrow_array;
```

```
#
#  d("Result: $count") if DEBUG;

} # end of sub sql_update_rows


##########################################################################
# delete rows from DB by condition
#
# sql_delete_rows($table_name,\@conditions);
#
# conditions must be an array so we can path more than one column with
# the same name.
#  @conditions =  ( column => ['comp_sign','value'],
#                   foo    => ['>',15],
#                   foo    => ['<',30],
#                 );
#
# * The sub knows automatically to detect and quote strings
#
#
############################
sub sql_delete_rows{
  my $self    = shift;
  my $table   = shift || '';
  my $r_conds = shift || [];

    # we want to print in the trace debug mode
  d( "[".(caller(2))[3]." - ".(caller(1))[3]." - ". (caller(0))[3]."]") if DEBUG;

    # build the query
  my $do_sql = "DELETE FROM $table ";

  my @where = ();
  for(my $i=0;$i<@{$r_conds};$i=$i+2) {
    push @where, join " ",
      $$r_conds[$i],
      $$r_conds[$i+1][0],
      sql_quote(sql_escape($$r_conds[$i+1][1]));
  }

    # Must be very careful with deletes, imagine somehow @where is
    # not getting set, "DELETE FROM NAME" deletes the contents of the table
  warn("Attempt to delete a whole table $table from DB\n!!!"),return unless @where;

    # Add the where clause if we have one
  $do_sql .= " WHERE ". join " AND ", @where;

  d("SQL: $do_sql") if DEBUG;

    # do query
  $self->{sth} = $self->{dbh}->prepare($do_sql);
  $self->{sth}->execute();

} # end of sub sql_delete_rows


##########################################################################
# executes the passed query and returns a reference to an array which
```

```
# contains one reference per row. If there are no rows to return,
# returns a reference to an empty array.
#
# $r_array = sql_execute_and_get_r_array($query);
#
#
##########################
sub sql_execute_and_get_r_array{
  my $self     = shift;
  my $do_sql   = shift || '';

    # we want to print in the trace debug mode
  d( "[".(caller(2))[3]." - ".(caller(1))[3]." - ". (caller(0))[3]."]") if DEBUG;

  d("SQL: $do_sql") if DEBUG;

  $self->{dbh}->selectall_arrayref($do_sql);

} # end of sub sql_execute_and_get_r_array


############################################################################
# lock the passed tables in the requested mode (READ|WRITE) and set
# internal flag to handle possible user abortions, so the tables will
# be unlocked thru the END{} block
#
# sql_lock_tables('table1','lockmode',..,'tableN','lockmode'
# lockmode = (READ | WRITE)
#
# _side_effect_ $self->{lock} = 'On';
#
##########################
sub sql_lock_tables{
  my $self   = shift;
  my %modes = @_;

  return unless %modes;

  my $do_sql = 'LOCK TABLES ';
  $do_sql .= join ",", map {"$_ $modes{$_}"} keys %modes;

    # we want to print the trace in debug mode
  d( "[".(caller(2))[3]." - ".(caller(1))[3]." - ". (caller(0))[3]."]") if DEBUG;

  d("SQL: $do_sql") if DEBUG;

  $self->{sth} = $self->{dbh}->prepare($do_sql);
  $self->{sth}->execute();

    # Enough to set only one lock, unlock will remove them all
  $self->{lock} = 'On';

} # end of sub sql_lock_tables



############################################################################
# unlock all tables, unset internal flag to handle possible user
# abortions, so the tables will be unlocked thru the END{} block
```

```
#
# sql_unlock_tables()
#
# _side_effect_: delete $self->{lock}
#
##########################
sub sql_unlock_tables{
  my $self   = shift;

    # we want to print the trace in debug mode
  d( "[".(caller(2))[3]." - ".(caller(1))[3]." - ". (caller(0))[3]."]") if DEBUG;

  $self->{dbh}->do("UNLOCK TABLES");

    # Enough to set only one lock, unlock will remove them all
  delete $self->{lock};

} # end of sub sql_unlock_tables

#
#
# return current date formatted for a DATE field type
# YYYYMMDD
#
# Note: since this function actually doesn't need an object it's being
# called without parameter as well as procedural call
#############
sub sql_date{
  my $self     = shift;

  my ($mday,$mon,$year) = (localtime)[3..5];
  return sprintf "%0.4d%0.2d%0.2d",1900+$year,++$mon,$mday;

} # end of sub sql_date

#
#
# return current date formatted for a DATE field type
# YYYYMMDDHHMMSS
#
# Note: since this function actually doesn't need an object it's being
# called without parameter as well as procedural call
#############
sub sql_datetime{
  my $self     = shift;

  my ($sec,$min,$hour,$mday,$mon,$year) = localtime();
  return sprintf "%0.4d%0.2d%0.2d%0.2d%0.2d%0.2d",1900+$year,++$mon,$mday,$hour,$min,$sec;

} # end of sub sql_datetime


# Quote the list of parameters.  Parameters consisting entirely of
# digits (i.e. integers) are unquoted.
# print sql_quote("one",2,"three"); => 'one', 2, 'three'
#############
sub sql_quote{ map{ /^(\d+|NULL)$/ ? $_ : "\'$_\'" } @_ }

# Escape the list of parameters (all unsafe chars like ",' are escaped)
```

```
# We make a copy of @_ since we might try to change the passed values,
# producing an error when modification of a read-only value is attempted
###############
sub sql_escape{ my @a = @_; map { s/([\'\\])/\\$1/g;$_} @a }


# DESTROY makes all kinds of cleanups if the fuctions were interuppted
# before their completion and haven't had a chance to make a clean up.
############
sub DESTROY{
  my $self = shift;

  $self->sql_unlock_tables() if $self->{lock};
  $self->{sth}->finish       if $self->{sth};
  $self->{dbh}->disconnect   if $self->{dbh};

} # end of sub DESTROY

# Don't remove
1;
```

module

(Note that you will not find this on CPAN. at least not yet :)

7.7.3 My::DB Module's Usage Examples

To use `My::DB` in your script, you first have to create a `My::DB` object:

```
  use vars qw($db_obj);
  my $db_obj = new My::DB or croak "Can't initialize My::DB object: $!\n";
```

Now you can use any of `My::DB`'s methods. Assume that we have a table called *tracker* where we store the names of the users and what they are doing at each and every moment (think about an online community program).

I will start with a very simple query--I want to know where the users are and produce statistics. `tracker` is the name of the table.

```
    # fetch the statistics of where users are
  my $r_ary = $db_obj->sql_get_matched_rows_ary_ref
    ("tracker",
     [qw(where_user_are)],
    );

  my %stats = ();
  my $total = 0;
  foreach my $r_row (@$r_ary){
    $stats{$r_row->[0]}++;
    $total++;
  }
```

Now let's count how many users we have (in table `users`):

```
my $count = $db_obj->sql_count_matched("users");
```

Check whether a user exists:

```
my $username = 'stas';
my $exists = $db_obj->sql_count_matched
("users",
 [username => ["=",$username]]
);
```

Check whether a user is online, and get the time since she went online (`since` is a column in the `tracker` table, it tells us when a user went online):

```
my @row = ();
$db_obj->sql_get_matched_row
(\@row,
 "tracker",
 ['UNIX_TIMESTAMP(since)'],
 [username => ["=",$username]]
);

if (@row) {
  my $idle = int( (time() - $row[0]) / 60);
  return "Current status: Is Online and idle for $idle minutes.";
}
```

A complex query. I join two tables, and I want a reference to an array which will store a slice of the matched query (`LIMIT $offset,$hits`) sorted by `username`. Each row in the array is to include the fields from the `users` table, but only those listed in `@verbose_cols`. Then we print it out.

```
my $r_ary = $db_obj->sql_get_matched_rows_ary_ref
  (
   "tracker STRAIGHT_JOIN users",
   [map {"users.$_"} @verbose_cols],
   [],
   ["WHERE tracker.username=users.username",
    "ORDER BY users.username",
    "LIMIT $offset,$hits"],
  );

foreach my $r_row (@$r_ary){
  print ...
}
```

Another complex query. The user checks checkboxes to be queried by, selects from lists and types in match strings, we process input and build the `@where` array. Then we want to get the number of matches and the matched rows as well.

```
my @search_keys = qw(choice1 choice2);
my @where = ();
  # Process the checkboxes - we turn them into a regular expression
foreach (@search_keys) {
  next unless defined $q->param($_) and $q->param($_);
```

```
  my $regexp = "[".join("",$q->param($_))."]";
  push @where, ($_ => ['REGEXP',$regexp]);
}

  # Add the items selected by the user from our lists
  # selected => exact match
push @where,(country => ['=',$q->param('country')]) if $q->param('country');

  # Add the parameters typed by the user
foreach (qw(city state)) {
  push @where,($_ => ['LIKE',$q->param($_)]) if $q->param($_);
}

   # Count all that matched the query
my $total_matched_users =  $db_obj->sql_count_matched
  (
   "users",
   \@where,
  );

  # Now process the orderby
my $orderby = $q->param('orderby') || 'username';

   # Do the query and fetch the data
my $r_ary = $db_obj->sql_get_matched_rows_ary_ref
(
 "users",
 \@display_columns,
 \@where,
 ["ORDER BY $orderby",
  "LIMIT $offset,$hits"],
);
```

`sql_get_matched_rows_ary_ref` knows to handle both `OR`ed and `AND`ed params. This example shows how to use `OR` on parameters:

This snippet is an implementation of a watchdog. Our users want to know when their colleagues go online. They register the usernames of the people they want to know about. We have to make two queries: one to get a list of usernames, the second to find out whether any of these users is online. In the second query we use the `OR` keyword.

```
# check who we are looking for
$r_ary = $db_obj->sql_get_matched_rows_ary_ref
  ("watchdog",
   [qw(watched)],
   [username => ['=',$username)],
   ],
  );

  # put them into an array
my @watched = map {$_->[0]} @{$r_ary};

my %matched = ();
  # Does the user have some registered usernames?
if (@watched) {
```

```
  # Try to fetch all the users who match the usernames exactly.
  # Put it into an array and compare it with a hash!
    $r_ary = $db_obj->sql_get_matched_rows_ary_ref
      ("tracker",
       [qw(username)],
       [username => ['=',\@watched],
       ]
      );

    map {$matched{$_->[0]} = 1} @{$r_ary};
  }

  # Now %matched includes the usernames of the users who are being
  # watched by $username and currently are online.
```

7.8 Maintainers

Maintainer is the person(s) you should contact with updates, corrections and patches.

- Stas Bekman [http://stason.org/]

7.9 Authors

- Stas Bekman [http://stason.org/]

Only the major authors are listed above. For contributors see the Changes file.

8 mod_perl and dbm files

8.1 Description

Small databases can be implemented pretty efficiently using dbm files, but there are still some precautions that must be taken to use them properly under mod_perl.

8.2 Where and Why to use dbm files

Some of the earliest databases implemented on Unix were dbm files, and many are still in use today. As of this writing the Berkeley DB is the most powerful dbm implementation (http://www.sleepycat.com).

If you need a light database, with an easy API, using simple key-value pairs to store and manipulate a relatively small number of records, this is a solution that should be amongst the first you consider.

With dbm, it is rare to read the whole database into memory. Combine this feature with the use of smart storage techniques, and dbm files can be manipulated much faster than flat files. Flat file databases can be very slow on insert, update and delete operations, when the number of records starts to grow into the thousands. Sort algorithms on flat files can be very time-consuming.

The maximum practical size of a dbm database depends on many factors--your data, your hardware and the desired response times of course included--but as a rough guide consider 5,000 to 10,000 records to be reasonable.

We will talk mostly about the Berkley DB version 1.x, as it provides the best functionality while having a good speed and almost no limitations. Other implementations might be faster in some cases, but they are either limited in the length of the maximum value or the total number of records.

There is a number of Perl interfaces to the major dbm implementations, to list a few: `DB_File`, `NDBM_File`, `ODBM_File`, `GDBM_File`, and `SDBM_File`. The original Perl module for Berkeley DB was DB_File, which was written to interface to Berkeley DB version 1.85. The newer Perl module for Berkeley DB is `BerkeleyDB`, which was written to interface to version 2.0 and subsequent releases. Because Berkeley DB version 2.X has a compatibility API for version 1.85, you can (and should!) build `DB_File` using version 2.X of Berkeley DB, although `DB_File` will still only support the 1.85 functionality.

Several different indexing algorithms (known also as access methods) can be used with dbm implementations:

- The `HASH` access method gives an `0(1)` complexity of search and update, fast insert and delete, but a slow sort (which you have to implement yourself). (Used by almost all dbm implementations)

- The `BTREE` access method allows arbitrary key/value pairs to be stored in a sorted, balanced binary tree. This allows us to get a sorted sequence of data pairs in `0(1)`, but at the expense of much slower insert, update, delete operations than is the case with `HASH`. (Available mostly in Berkeley DB)

- The `RECNO` access method is more complicated, and enables both fixed-length and variable-length flat text files to be manipulated using the same key/value pair interface as in `HASH` and `BTREE`. In this case the key will consist of a record (line) number. (Available mostly in Berkeley DB)

- The QUEUE access method stores fixed-length records with logical record numbers as keys. It is designed for fast inserts at the tail and has a special cursor consume operation that deletes and returns a record from the head of the queue. The QUEUE access method uses record level locking. (Available only in Berkeley DB version 3.x)

Most often you will want to use the HASH method, but there are many considerations and your choice may be dictated by your application.

In recent years dbm databases have been extended to allow you to store more complex values, including data structures. The MLDBM module can store and restore the whole symbol table of your script, including arrays and hashes.

It is important to note that you cannot simply switch a dbm file from one storage algorithm to another. The only way to change the algorithm is to copy all the records one by one into a new dbm file, which was initialized according to a desired access method. You can use a script like this:

```
#!/usr/bin/perl -w

#
# This script takes as its parameters a list of Berkeley DB
# file(s) which are stored with the DB_BTREE algorithm.  It
# will back them up using the .bak extension and create
# instead dbms with the same records but stored using the
# DB_HASH algorithm
#
# Usage: btree2hash.pl filename(s)

use strict;
use DB_File;
use Fcntl;

  # Do checks
die "Usage: btree2hash.pl filename(s))\n" unless @ARGV;

foreach my $filename (@ARGV) {

  die "Can't find $filename: $!\n"
    unless -e $filename and -r $filename;

    # First backup the file
  rename "$filename", "$filename.btree"
    or die "can't rename $filename $filename.btree:$!\n";

    # tie both dbs (db_hash is a fresh one!)
  tie my %btree , 'DB_File',"$filename.btree", O_RDWR|O_CREAT,
      0660, $DB_BTREE or die "Can't tie $filename.btree: $!";
  tie my %hash ,  'DB_File',"$filename" , O_RDWR|O_CREAT,
      0660, $DB_HASH  or die "Can't tie $filename: $!";

    # copy DB
  %hash = %btree;
```

```
    # untie
  untie %btree ;
  untie %hash ;
}
```

Note that some dbm implementations come with other conversion utilities as well.

8.3 mod_perl and dbm

Where does mod_perl fit into the picture?

If you need to access a dbm file in your mod_perl code in the read only mode the operation would be much faster if you keep the dbm file open (tied) all the time and therefore ready to be used. This will work with dynamic (read/write) databases accesses as well, but you need to use locking and data flushing to avoid data corruption.

Although mod_perl and dbm can give huge performance gains compared to the use of flat file databases you should be very careful. In addition to the need for locking, you need to consider the consequences of `die()` and unexpected process death.

If your locking mechanism cannot handle dropped locks, a stale lock can deactivate your whole site. You can enter a deadlock situation if two processes simultaneously try to acquire locks on two separate databases. Each has locked only one of the databases, and cannot continue without locking the second. Yet this will never be freed because it is locked by the other process. If your processes all ask for their DB files in the same order, this situation cannot occur.

If you modify the DB you should be make very sure that you flush the data and synchronize it, especially when the process serving your handler unexpectedly dies. In general your application should be tested very thoroughly before you put it into production to handle important data.

8.4 Locking dbm Handlers and Write Lock Starvation Hazards

One has to deploy dbm file locking if there is chance that some process will want to write to it. Note that once you need to do locking you do it even when all you want is to read from the file. Since if you don’t, it’s possible that someone writes to the file at this very moment and you may read partly updated data.

Therefore we should distinguish between *READ* and *WRITE* locks. Before doing an operation on the dbm file, we first issue either a *READ* or a *WRITE* lock request, according to our needs.

If we are making a *READ* lock request, it is granted as soon as the *WRITE* lock on the file is removed if any or if it is already *READ* locked. The lock status becomes *READ* on success.

If we make a *WRITE* lock request, it is granted as soon as the file becomes unlocked. The lock status becomes *WRITE* on success.

The treatment of the *WRITE* lock request is most important.

If the DB is *READ* locked, a process that makes a *WRITE* request will poll until there are no reading or writing processes left. Lots of processes can successfully read the file, since they do not block each other. This means that a process that wants to write to the file may never get a chance to squeeze in, since it needs to obtain an exclusive lock.

The following diagram represents a possible scenario where everybody can read but no one can write (pX's represent different processes running for different times and all acquiring the read lock on the dbm file):

```
  [-p1-]                 [--p1--]
     [--p2--]               [--p2--]
   [---------p3---------] [--------p3---.....
                 [------p4-----]
```

The result is a starving process, which will timeout the request, and it will fail to update the DB. Ken Williams solved the above problem with his `Tie::DB_Lock` module, which is discussed in one of the following sections.

There are several locking wrappers for `DB_File` in CPAN right now. Each one implements locking differently and has different goals in mind. It is therefore worth knowing the difference, so that you can pick the right one for your application.

8.5 Flawed Locking Methods Which Must Not Be Used

Caution: The suggested locking methods in the Camel book and `DB_File` man page (before version 1.72, fixed in 1.73) are flawed. If you use them in an environment where more than one process can modify the dbm file, it can get corrupted!!! The following is an explanation of why this happens.

You may not use a tied file's filehandle for locking, since you get the filehandle after the file has been already tied. It's too late to lock. The problem is that the database file is locked *after* it is opened. When the database is opened, the first 4k (in Berkley dbm library) is read and then cached in memory. Therefore, a process can open the database file, cache the first 4k, and then block while another process writes to the file. If the second process modifies the first 4k of the file, when the original process gets the lock is now has an inconsistent view of the database. If it writes using this view it may easily corrupt the database on disk.

This problem can be difficult to trace because it does not cause corruption every time a process has to wait for a lock. One can do quite a bit of writing to a database file without actually changing the first 4k. But once you suspect this problem you can easily reproduce it by making your program modify the records in the first 4k of the DB.

You better resort to using the standard modules for locking instead of inventing your own.

If your dbm file is used only in the read-only mode generally there is no need for locking at all. If you access the dbm file in read/write mode, the safest method is to tie() the dbm file after acquiring an external lock and untie() before the lock is released. So to access the file in shared mode (FLOCK_SH) one should

following this pseudo-code:

```
flock FLOCK_SH <===== start critical section
tie()
read...
untie()
flock FLOCK_UN <===== end critical section
```

Similar for the exclusive (EX), write access:

```
flock FLOCK_EX <===== start critical section
tie()
write...
sync()
untie()
flock FLOCK_UN <===== end critical section
```

However you might want to save a few tie()/untie() calls if the same request accesses the dbm file more than once. You should be careful though. Based on the caching effect explained above, a process can perform an atomic downgrade of an exclusive lock to a shared one without re-tie()ing the file:

```
flock FLOCK_EX <===== start critical section
tie()
write...
sync()
               <===== end critical section
flock FLOCK_SH <===== start critical section
read...
untie()
flock FLOCK_UN <===== end critical section
```

because it has the updated data in its cache. By atomic, we mean it’s ensured that the lock status gets changed, without any other process getting an exclusive access in between.

If you can ensure that one process safely upgrades a shared lock with an exclusive lock, one can save on tie()/untie(). But this operation might lead to a dead-lock if two processes try to upgrade a shared lock with exclusive at the same time. Remember that in order to acquire an exclusive lock, all other processes need to release *all* locks. If your OS locking implementation resolves this deadlock by denying one of the upgrade requests, make sure your program handles that appropriately. The process that were denied has to untie() the dbm file and then ask for an exclusive lock.

A dbm file has always to be untie()’ed before the locking is released (unless you do an atomic downgrade from exclusive to shared as we have just explained). Remember that if at any given moment a process wants to lock and access the dbm file it has to re-tie() this file, if it was tied already. If this is not done, the integrity of the dbm file is not ensured.

To conclude, the safest method of reading from dbm file is to lock the file before tie()-ing it, untie() it before releasing the lock, and in the case of write to call sync() before untie()-ing it.

8.6 Locking Wrappers Overview

Here are some of the correctly working dbm locking wrappers on (three of them are available from CPAN):

- `Tie::DB_Lock` -- `DB_File` wrapper which creates copies of the dbm file for read access, so that you have kind of a multiversioning concurrent read system. However, updates are still serial. After each update the read-only copies of the dbm file are recreated. Use this wrapper in situations where reads may be very lengthy and therefore write starvation problem may occur. On the other hand if you have big dbm files, it may create a big load on the system if the updates are quite frequent. More information.

- `Tie::DB_FileLock` -- `DB_File` wrapper that has the ability to lock and unlock the database while it is being used. Avoids the tie-before-flock problem by simply re-tie-ing the database when you get or drop a lock. Because of the flexibility in dropping and re-acquiring the lock in the middle of a session, this can be used in a system that will work with long updates and/or reads. Refer to the `Tie::DB_FileLock` manpage for more information.

- `DB_File::Lock` -- extremely lightweight `DB_File` wrapper that simply flocks an external lock-file before tie-ing the database and drops the lock after untie. Allows one to use the same lockfile for multiple databases to avoid deadlock problems, if desired. Use this for databases where updates and reads are quick and simple flock locking semantics are enough. Refer to `DB_File::Lock` manpage for more information.

- `DB_File::Lock2` -- does the same thing as `DB_File::Lock`, but has a slightly different implementation. I wrote it before David Harris released his `DB_File::Lock` and I didn't want to kill mine, so I'll keep it here for a while :).

- On some Operating Systems (FreeBSD is one example) it is possible to lock on tie:

  ```
  tie my %t, 'DB_File', $TOK_FILE, O_RDWR | O_EXLOCK, 0664;
  ```

  and only release the lock by un-tie()-ing the file. Check if the `O_EXLOCK` flag is available on your operating system before you try to use this method!

8.7 Tie::DB_Lock

`Tie::DB_Lock` ties hashes to databases using shared and exclusive locks. This module, by Ken Williams, solves the problems raised in the previous section.

The main difference from what I have described above is that `Tie::DB_Lock` copies a dbm file on read. Reading processes do not have to keep the file locked while they read it, and writing processes can still access the file while others are reading. This works best when you have lots of long-duration reading, and a few short bursts of writing.

The drawback of this module is the heavy IO performed when every reader makes a fresh copy of the DB. With big dbm files this can be quite a disadvantage and can slow the server down considerably.

An alternative would be to have one copy of the dbm image shared by all the reading processes. This can cut the number of files that are copied, and puts the responsibility of copying the read-only file on the writer, not the reader. It would need some care to make sure it does not disturb readers when putting a new read-only copy into place.

8.8 DB_File::Lock2

Here is *code/DB_File-Lock2.pm*:

```
package DB_File::Lock2;
require 5.004;

use strict;

BEGIN {
    # RCS/SVN compliant:  must be all one line, for MakeMaker
  $DB_File::Lock2::VERSION = do { my @r = (q$Revision: 420028 $ =~ /\d+/g); sprintf "%d."."%02d" x $#r, @r };

}

use DB_File ();
use Fcntl qw(:flock O_RDWR O_CREAT);
use Carp qw(croak carp verbose);
use Symbol ();

@DB_File::Lock2::ISA    = qw( DB_File );
%DB_File::Lock2::lockfhs = ();

use constant DEBUG => 0;

  # file creation permissions mode
use constant PERM_MODE => 0660;

  # file locking modes
%DB_File::Lock2::locks =
  (
   read  => LOCK_SH,
   write => LOCK_EX,
  );

# SYNOPSIS:
# tie my %mydb, 'DB_File::Lock2', $filepath,
#     ['read' || 'write', 'HASH' || 'BTREE']
# while (my ($k,$v) = each %mydb) {
#   print "$k => $v\n";
# }
# untie %mydb;
#########
sub TIEHASH {
  my $class     = shift;
  my $file      = shift;
  my $lock_mode = lc shift || 'read';
  my $db_type   = shift || 'HASH';

  die "Dunno about lock mode: [$lock_mode].\n
       Valid modes are 'read' or 'write'.\n"
    unless $lock_mode eq 'read' or $lock_mode eq 'write';

  # Critical section starts here if in write mode!

    # create an external lock
  my $lockfh = Symbol::gensym();
```

```
  open $lockfh, ">$file.lock" or die "Cannot open $file.lock for writing: $!\n";
  unless (flock $lockfh, $DB_File::Lock2::locks{$lock_mode}) {
    croak "cannot flock: $lock_mode => $DB_File::Lock2::locks{$lock_mode}: $!\n";
  }

  my $self = $class->SUPER::TIEHASH
    ($file,
     O_RDWR|O_CREAT,
     PERM_MODE,
     ($db_type eq 'BTREE' ? $DB_File::DB_BTREE : $DB_File::DB_HASH )
    );

    # remove the package name in case re-blessing occurs
  (my $id = "$self") =~ s/^[^=]+=//;

    # cache the lock fh
  $DB_File::Lock2::lockfhs{$id} = $lockfh;

  return $self;

} # end of sub new


# DESTROY is automatically called when a tied variable
# goes out of scope, on explicit untie() or when the program is
# interrupted, e.g. with a die() call.
#
# It unties the db by forwarding it to the parent class,
# unlocks the file and removes it from the cache of locks.
############
sub DESTROY{
  my $self = shift;

  $self->SUPER::DESTROY(@_);

    # now it safe to unlock the file, (close() unlocks as well). Since
    # the object has gone we remove its lock filehandler entry
    # from the cache.
  (my $id = "$self") =~ s/^[^=]+=//; # see 'sub TIEHASH'
  close delete $DB_File::Lock2::lockfhs{$id};

    # Critical section ends here if in write mode!

  print "Destroying ".__PACKAGE__."\n" if DEBUG;

}

####
END {
  print "Calling the END from ".__PACKAGE__."\n" if DEBUG;

}

1;
```

which does the locking by using an external lockfile.

This allows you to gain the lock before the file is tied. Note that it's not yet on CPAN and so is linked from here in its entirety. Note also that this code still needs some testing, so *be careful* if you use it on a production machine.

You use it like this:

```
  use DB_File::Lock2 ();
```

A simple tie, READ lock and untie

```
use DB_File::Lock2 ();
my $dbfile = "/tmp/test";
tie my %mydb, 'DB_File::Lock2', $dbfile, 'read';
print $mydb{foo} if exists $mydb{foo};
untie %mydb;
```

You can even skip the `untie()` call. When `$mydb` goes out of scope everything will be done automatically. However it is better to use the explicit call, to make sure the critical sections between lock and unlock are as short as possible. This is especially important when requesting an exclusive (write) lock.

The following example shows how it might be convenient to skip the explicit `untie()`. In this example, we don't need to save the intermediate result, we just return and the cleanup is done automatically.

```
use DB_File::Lock2 ();
my $dbfile = "/tmp/test";
print user_exists("stas") ? "Yes" : "No";
sub user_exists{
  my $username = shift || '';

  warn("No username passed\n"), return 0 unless $username;

  tie my %mydb, 'DB_File::Lock2', $dbfile, 'read';

  # if we match the username return 1, else 0
  return $mydb{$username} ? 1 : 0;

} # end of sub user_exists
```

Now let's write all the upper case characters and their respective ASCII values to a dbm file. Then read the file and print the contents of the DB, unsorted.

```
use DB_File::Lock2 ();
my $dbfile = "/tmp/test";

  # write
tie my %mydb, 'DB_File::Lock2', $dbfile,'write';
for (0..26) {
  $mydb{chr 65+$_} = $_;
}
untie %mydb;

  # now, read them and printout (unsorted)
  # notice that 'read' is a default lock mode
tie %mydb, 'DB_File::Lock2', $dbfile;
while (my ($k,$v) = each %mydb) {
  print "$k => $v\n";
}
untie %mydb;
```

If your CGI script is interrupted, the `DESTROY` block will take care of unlocking the dbm file and flush any changes. So your DB will be safe against possible corruption because of unclean program termination.

8.9 Maintainers

Maintainer is the person(s) you should contact with updates, corrections and patches.

- Stas Bekman [http://stason.org/]

8.10 Authors

- Stas Bekman [http://stason.org/]

Only the major authors are listed above. For contributors see the Changes file.

9 Protecting Your Site

9.1 Description

Securing your site should be your first priority, because of the consequences a break-in might have. We discuss the various authentication and authorization techniques available, a very interesting use of mod_perl.

9.2 The Importance of Your site's Security

Let's face it, your site or service can easily become a target for Internet "terrorists". It can be because of something you said, the success of your site, or for no obvious reason whatever. If your site security is compromised, all your data can be deleted or important information can be stolen. You may risk legal action or the sack if this happens.

Your site can be paralyzed through a _simple_ *denial of service* (DoS) attack.

Whatever you do, as long as you are connected to the network your site will be vulnerable. Cut the connections, turn off your machine and put it into a safe. Now it is protected--but useless.

So what can you do?

Let's first get acquainted with some security related terminology:

- **Authentication**

  When you want to make sure that a user is who he claims to be, you generally ask her for a username and a password. Once you have both, you can check them against your database of username/password pairs. If they match, the user has passed the **Authentication** stage. From now on if you keep the session open all you need to do is to remember the username.

- **Authorization**

  You might want to allow user **foo** to have access to some resource, but restrict her from accessing another resource, which in turn is accessible only for user **bar**. The process of checking access rights is called **Authorization**. For **Authorization** all you need is an authenticated username or some other attribute which you can authorize against. For example, you can authorize against IP number, allowing only your local users to use some service. But be warned that IP numbers or session_ids can be spoofed (forged), and that is why you should not do **Authorization** without **Authentication**.

Actually you've been familiar with both these concepts for a while.

When you telnet to your account on some machine you go through a login process (**Authentication**).

When you try to read some file from your file systems, the kernel checks the permissions on this file (**Authorization**). You may hear about **Access control** which is another name for the same thing.

9.3 Illustrated Security Scenarios

I am going to present some real world security requirements and their implementations.

9.3.1 Non authenticated access for internal IPs, Authenticated for external IPs

An **Extranet** is very similar to an **Intranet**, but at least partly accessible from outside your organization. If you run an Extranet you might want to let your internal users have unrestricted access to your web server. If these same users call from outside your organization you might want to make sure that they are in fact your employees.

These requirements are achieved very simply by putting the IP patterns of the organization in a Perl Access Handler in an `.htaccess` file. This sets the REMOTE_USER environment variable to the organization's generic username. Scripts can test the `REMOTE_USER` environment variable to determine whether to allow unrestricted access or else to require authentication.

Once a user passes the authentication stage, either bypassing it because of his IP address or after entering a correct login/password pair, the `REMOTE_USER` variable is set. Then we can talk about authorization.

Let's see the implementation of the authentication stage. First we modify *httpd.conf*:

```
PerlModule My::Auth

<Location /private>
  PerlAccessHandler My::Auth::access_handler
  PerlSetVar Intranet "10.10.10.1 => userA, 10.10.10.2 => userB"
  PerlAuthenHandler My::Auth::authen_handler
  AuthName realm
  AuthType Basic
  Require valid-user
  Order deny, allow
  Deny from all
</Location>
```

Now the code of My/Auth.pm:

```
   sub access_handler {

       my $r = shift;

       unless ($r->some_auth_required) {
               $r->log_reason("No authentication has been configured");
               return FORBIDDEN;
       }
       # get list of IP addresses
       my %ips = split /\s*(?:=>|,)\s*/, $r->dir_config("Intranet");

       if (my $user = $ips{$r->connection->remote_ip}) {

               # update connection record
```

```
            $r->connection->user($user);

            # do not ask for a password
            $r->set_handlers(PerlAuthenHandler => [\&OK]);
    }
    return OK;
}

sub authen_handler {

    my $r = shift;

    # get user's authentication credentials
    my ($res, $sent_pw) = $r->get_basic_auth_pw;
    return $res if $res != OK;
    my $user = $r->connection->user;

    # authenticate through DBI
    my $reason = authen_dbi($r, $user, $sent_pw);

    if ($reason) {
            $r->note_basic_auth_failure;
            $r->log_reason($reason, $r->uri);
            return AUTH_REQUIRED;
    }
    return OK;

}

sub authen_dbi{
  my ($r, $user, $sent_pw) = @_;

  # validate username/passwd

  return 0 if (*PASSED*) # replace with real code!!!

  return "Failed for X reason";

}
# don't forget 1;
1;
```

You can implement your own `authen_dbi()` routine, or you can replace `authen_handler()` with an existing authentication handler such as `Apache::AuthenDBI`.

If one of the IP addresses is matched, `access_handler()` sets `REMOTE_USER` to be either `userA` or `userB`.

If neither IP address is matched, `PerlAuthenHandler` will not be set to OK, and the Authentication stage will ask the user for a login and password.

9.4 Authentication code snippets

9.4.1 Forcing re-authentication

To force an authenticated user to reauthenticate just send the following header to the browser:

```
WWW-Authenticate: Basic realm="My Realm"
HTTP/1.0 401 Unauthorized
```

This will pop-up (in Netscape at least) a window saying **Authorization Failed. Retry?** with **OK** and a **Cancel** buttons. When that window pops up you know that the password has been discarded. If the user hits the **Cancel** button the username will also be discarded. If she hits the **OK** button, the authentication window will be brought up again with the previous username already in place.

In the Perl API you would use the note_basic_auth_failure() method to force reauthentication.

This may not work! The browser's behaviour is in no way guaranteed.

9.4.2 OK, AUTH_REQUIRED and FORBIDDEN in Authentication handlers

When your authentication handler returns OK, it means that user has correctly authenticated and now `$r->connection->user` will have the username set for subsequent requests. For `Apache::Registry` and friends, where the environment variable settings weren't erased, an equivalent `$ENV{REMOTE_USER}` variable will be available.

The password is available only through the Perl API with the help of the get_basic_auth_pw() method.

If there is a failure, unless it's the first time, the `AUTH_REQUIRED` flag will tell the browser to pop up an authentication window, to try again. For example:

```
my ($status, $sent_pw) = $r->get_basic_auth_pw;
unless($r->connection->user and $sent_pw) {
    $r->note_basic_auth_failure;
    $r->log_reason("Both a username and password must be provided");
    return AUTH_REQUIRED;
}
```

Let's say that you have a mod_perl authentication handler, where the user's credentials are checked against a database. It returns either `OK` or `AUTH_REQUIRED`. One of the possible authentication failure case might happen when the username/password are correct, but the user's account has been suspended temporarily.

If this is the case you would like to make the user aware of this, by displaying a page, instead of having the browser pop up the authentication dialog again. You will also refuse authentication, of course.

The solution is to return `FORBIDDEN`, but before that you should set a custom error page for this specific handler, with help of `$r->custom_response`. It looks something like this:

```
use Apache::Constants qw(:common);
$r->custom_response(SERVER_ERROR, "/errors/suspended_account.html");

return FORBIDDEN if $suspended;
```

9.5 Apache::Auth* modules

- **PerlAuthenHandler's**

```
Apache::AuthAny           Authenticate with any username/password
Apache::AuthenCache       Cache authentication credentials
Apache::AuthCookie        Authen + Authz via cookies
Apache::AuthenDBI         Authenticate via Perl's DBI
Apache::AuthExpire        Expire Basic auth credentials
Apache::AuthenGSS         Generic Security Service (RFC 2078)
Apache::AuthenIMAP        Authentication via an IMAP server
Apache::AuthenPasswdSrv   External authentication server
Apache::AuthenPasswd      Authenticate against /etc/passwd
Apache::AuthLDAP          LDAP authentication module
Apache::AuthPerLDAP       LDAP authentication module (PerLDAP)
Apache::AuthenNIS         NIS authentication
Apache::AuthNISPlus       NIS Plus authentication/authorization
Apache::AuthenRaduis      Authentication via a Radius server
Apache::AuthenSmb         Authenticate against NT server
Apache::AuthenURL         Authenticate via another URL
Apache::DBILogin          Authenticate to backend database
Apache::DCELogin          Obtain a DCE login context
Apache::PHLogin           Authenticate via a PH database
Apache::TicketAccess      Ticket based access/authentication
```

- **PerlAuthzHandler's**

```
Apache::AuthCookie        Authen + Authz via cookies
Apache::AuthzAge          Authorize based on age
Apache::AuthzDCE          DFS/DCE ACL based access control
Apache::AuthzDBI          Group authorization via Perl's DBI
Apache::AuthzGender       Authorize based on gender
Apache::AuthzNIS          NIS authorization
Apache::AuthzPasswd       Authorize against /etc/passwd
Apache::AuthzSSL          Authorize based on client cert
Apache::RoleAuthz         Role-based authorization
```

- **PerlAccessHandler's**

```
Apache::AccessLimitNum    Limit user access by number of requests
Apache::BlockAgent        Block access from certain agents
Apache::DayLimit          Limit access based on day of week
Apache::IPThrottle        Limit bandwith consumption by IP
Apache::RobotLimit        Limit access of robots
Apache::SpeedLimit        Control client request rate
```

9.6 Maintainers

Maintainer is the person(s) you should contact with updates, corrections and patches.

- Stas Bekman [http://stason.org/]

9.7 Authors

- Stas Bekman [http://stason.org/]

Only the major authors are listed above. For contributors see the Changes file.

10 Code Snippets

10.1 Description

A collection of mod_perl code snippets which you can either adapt to your own use or integrate directly into your own code.

10.2 File Upload with Apache::Request

The `Apache::Request` module gives you an easy way to get form content, including uploaded files. In order to add file upload functionality to your form, you need to add two things.

First, you'll need to add a form field which is type *file*. This will put a `browse` button on the form that will allow the user to choose a file to upload.

Second, you'll neet to make sure to add, to the `form` tag the following:

```
    enctype="multipart/form-data"
```

You won't be able to upload a file unless you have added this to the `form` tag.

In your code, you'll need to take a few extra steps to actually retrieve that file that has been uploaded. Using the following `form()` method will allow you to have a standard function that handles all of your forms, and does the right thing in the event that there was a file uploaded. You can put this function in your `mod_perl` handler, or in whatever module you want.

```
sub form {
    use Apache::Request;
    my $r = Apache->request();
    my $apr = Apache::Request->new($r);
    my @keys = $apr->param;

    my %form;
    foreach my $key(@keys) {

        my @value = $apr->param($key);
        next unless scalar @value;

        if ( @value > 1 ) {
            $form{$key} = \@value;
          } else {
            $form{$key} = $value[0];
        }
    }

    my $upload = $apr->upload;
    if ($upload) {
        $form{UPLOAD} = $upload;
    }

   return \%form;
}
```

In your code, you can get the contents of the form by calling this function:

```
my $form = Your::Class::form(); # Wherever you put this function
```

The value returned from this function is compatible with `CGI.pm` and other modules such as `CGI::Lite` Which is to say, the function returns a hashref. The keys of the hash are the names in your form. The values in the hash are the values entered in those fields, with the exception that a multiple select list with multiple things selected will return a listref of the selected values.

If your form contained a file upload element, then `$form{UPLOAD}` will contain a file upload object, which you can make calls back into.

For example:

```
my $form = Your::Class::form(); # Wherever you put this function
if (my $file = $form->{UPLOAD}) {
    my $filename = $file->filename; # If you need the name

    # And, if you want to save the file at $filelocation ...
    open F, ">$filelocation";
    my $filehandle = $file->fh;
    while (my $d = <$filehandle>) {
        print F $d;
    }
    close F;
}
```

That should give you the general idea of how this works. This lets you have a generic form handler that does "normal" forms as well as file upload forms, in mod_perl, without having to mess with `CGI.pm`, and without having to do custom things when you have a file upload.

You will need to see the documentation for `Apache::Upload` for more information about how to deal with the file upload object once you have it. Note that the `Apache::Upload` docs are embeded in the `Apache::Request` documentation, so you'll need to look there for that information.

10.3 Redirecting Errors to the Client Instead of error_log

Many error conditions result in an *exception* (or *signal* -- same thing) which is *raised* in order to tell the operating system that a condition has arisen which needs more urgent attention than can be given by other means. One of the most familiar ways of raising a signal is to hit `Ctrl-C` on your terminal's keyboard. The signal interrupts the processor. In the case of `Ctrl-C` the `INT` signal is generated and the interrupt is usually *trapped* by a default *signal handler* supplied by OS, which causes the operating system to stop the process currently attached to the terminal.

Under mod_perl, a Perl runtime error causes an exception. By default this exception is trapped by default mod_perl handler. The handler logs information about the error (such as the date and time that the error occurred) to *error_log*. If you want to redirect this information to the client instead of to *error_log* you have to take the responsibility yourself, by writing your own exception handler to implement this behaviour. See the section "Exception Handling for mod_perl" for more information.

The code examples below can be useful with your own exception handlers as well as with the default handlers.

META: Integrate the 2 sections

The CGI::Carp package implements handlers for signals. To trap (almost) all Perl run-time errors and send the output to the client instead of to Apache's `error_log` add this line to your script:

```
use CGI::Carp qw(fatalsToBrowser);
```

Refer to the `CGI::Carp` man page for more detailed information.

You can trap individual exceptions: for example you can write custom `__DIE__` and `__WARN__` signal handlers. The special `%SIG` hash contains references to signal handlers. The signal handler is just a subroutine, in the example below it is called "mydie". To install the handler we assign a reference to our handler to the appropriate element of the `%SIG` hash. This causes the signal handler to call mydie(error_message) whenever the `die()` sub is called as a result of something which happened when our script was executing.

Do not forget the `local` keyword! If you do, then after the signal handler has been loaded it will be called whenever `die()` is called by *any* script executed by the same process. Probably that's not what you want. If it is, you can put the assignment statement in any module, as long as it will be executed at the right time.

Here is an example of a handler which I wrote because I wanted users to know that there was an error, without displaying the error message, but instead email it to me. If the error is caused by user (e.g. uploading image whose size is bigger than the limit I had set) I want to tell them about it. I wrote this handler for the mod_perl environment, but it works correctly when called from the shell. The code shown below is a stripped-down version with additional comments.

The following code must be added to the script:

```
# Using the local() keyword restricts the scope of the directive to
# the block in which it is found, so this line must be added at the
# right place in the right script.  It will not affect other blocks
# unless the local() keyword is removed.  Usually you will want the
# directive to affect the entire script, so you just place it near
# the beginning of the file, where the innermost enclosing block is
# the file itself.
local $SIG{__DIE__} = \&mydie;

# The line above assumes that the subroutine "mydie" is in the same script.
# Alternatively you can create a separate module for the error handler.
# If you put the signal handler in a separate module, e.g. Error.pm,
# you must explicitly give the package name to set the handler in your
# script, using a line like this instead of the one above:
local $SIG{__DIE__} = \&Error::mydie;
# again within the script!

# Do not forget the C<local()>, unless you want this signal handler to
# be invoked every time any scripts dies (including events where this
# treatment may be undesirable).
```

```
my $max_image_size = 100*1024; # 100k
my $admin_email    = 'foo@example.com';

# and the handler itself
# Here is the handler itself:
# The handler is called with a text message in a scalar argument
sub mydie{
  my $why = shift;

  chomp $why;
  my $orig_why = $why;        # an ASCII copy for email report

  # handle the shell execution case (so we will not get all the HTML)
  print("Error: $why\n"), exit unless $ENV{MOD_PERL};

  my $should_email = 0;
  my $message = '';

  $why =~ s/[<&>]/"&#".ord($&).";"/ge;    # entity escape


  # Now we need to trap various kinds of errors that come from CGI.pm
  # We don't want these errors to be emailed to us, since
  # they aren't programmatical errors
  if ($orig_why =~ /Client attempted to POST (\d+) bytes/o) {

    $message = qq{
        You cannot POST messages bigger than
        @{[1024*$max_image_size]} bytes.<BR>
        You have tried to post $1 bytes<BR>
        If you are trying to upload an image, make sure its
        size is no bigger than @{[1024*$max_image_size]}
        bytes.<P>
        Thank you!
       };

  } elsif ($orig_why =~ /Malformed multipart POST/o) {

    $message = qq{
        Have you tried to upload an image in the wrong way?<P>
        To successfully upload an image you must use a browser that supports
        image upload and use the 'Browse' button to select that image.
        DO NOT type the path to the image into the upload field.<P>
        Thank you!
       };

  } elsif ($orig_why =~ /closed socket during multipart read/o) {

    $message = qq{
        Have you pressed a 'STOP' button?<BR>
        Please try again!<P>
        Thank you!
       };

  } else {
```

```
    $message = qq{
        <B>You need take no action since
        the error report has already been
        sent to the webmaster. <BR><P>

        <B>Thank you for your patience!</B>
       };

    $should_email = 1;
  }


  print qq{Content-type: text/html

<HTML><BODY BGCOLOR="white">
<B>Oops, Something went wrong.</B><P>
$message
</BODY></HTML>};

    # send email report if appropriate
  if ($should_email){

      # import sendmail subs
    use Mail ();
      # prepare the email error report:
    my $subject ="Error Report";
    my $body = qq|
  An error has happened:

  $orig_why

    |;

      # send error reports to admin
    send_mail($admin_email,$admin_email,$subject,$body);
    print STDERR "[".scalar localtime()."] [SIGDIE] Sending Error Email\n";
  }

     # print to error_log so we will know there was an error
  print STDERR "[".scalar localtime()."] [SIGDIE] $orig_why \n";

  exit 1;
}             # end of sub mydie
```

You may have noticed that I trap the CGI.pm's die() calls here, I don't see any reason why my users should see ugly error messages, but that's the way CGI.pm written. The workaround is to trap them yourself.

Please note that as of version 2.49, CGI.pm provides the cgi_error() method to print the errors and won't die() unless you want it to.

10.4 Reusing Data from POST request

What happens if you need to access the POSTed data more than once, say to reuse it in subsequent handlers of the same request? POSTed data comes directly from the socket, and at the low level data can only be read from a socket once. So you have to store it to make it available for reuse.

There is an experimental option for `Makefile.PL` called `PERL_STASH_POST_DATA`. If you turn it on, you can get at it again with `$r->subprocess_env("POST_DATA")`. This is not *enabled* by default because it adds a processing overhead for each POST request.

But what do we do with large multipart file uploads? Because `POST` data is not all read in one clump, it's a problem that's not easy to solve in a general way. A transparent way to do this is to switch the request method from POST to GET, and store the POST data in the query string. This handler does exactly this:

```
Apache/POST2GET.pm
------------------
package Apache::POST2GET;
use Apache::Constants qw(M_GET OK DECLINED);

sub handler {
  my $r = shift;
  return DECLINED unless $r->method eq "POST";
  $r->args(scalar $r->content);
  $r->method('GET');
  $r->method_number(M_GET);
  $r->headers_in->unset('Content-length');
  return OK;
}
1;
__END__
```

In *httpd.conf* add:

```
PerlInitHandler Apache::POST2GET
```

or even this:

```
<Limit POST>
  PerlInitHandler Apache::POST2GET
</Limit>
```

To save a few more cycles, so the handler will be called only for POST requests.

Effectively, this trick turns the POST request into a GET request internally. Now when `CGI.pm`, `Apache::Request` or whatever module parses the client data, it can do so more than once since `$r->args` doesn't go away (unless you make it go away by resetting it).

If you are using `Apache::Request`, it solves this problem for you with its instance() class method, which allows `Apache::Request` to be a singleton. This means that whenever you call `Apache::Request`->instance() within a single request you always get the same `Apache::Request` object back.

10.5 Redirecting POST Requests

Under mod_cgi it's not easy to redirect POST requests to some other location. With mod_perl you can easily redirect POST requests. All you have to do is read in the content, set the method to `GET`, populate `args()` with the content to be forwarded and finally do the redirect:

```
use Apache::Constants qw(M_GET);
my $r = shift;
my $content = $r->content;
$r->method("GET");
$r->method_number(M_GET);
$r->headers_in->unset("Content-length");
$r->args($content);
$r->internal_redirect_handler("/new/url");
```

Of course that last line can be any other kind of redirect.

10.6 Redirecting While Maintaining Environment Variables

Let's say you have a module that sets some environment variables.

If you redirect, that's most likely telling the web browser to fetch the new page. This makes it a totally new request, so no environment variables are preserved.

However, if you're using internal_redirect(), you can make the environment variables seen in the sub-process via subprocess_env(). The only nuance is that the `%ENV` keys will be prefixed with `REDIRECT_`.

10.7 Terminating a Child Process on Request Completion

If you want to terminate the child process serving the current request, upon completion of processing anywhere in the code call:

```
$r->child_terminate;
```

Apache won't actually terminate the child until everything it needs to do is done and the connection is closed.

10.8 Setting Content-type and Content-encoding headers in non-OK responses

You cannot set *Content-type* and *Content-encoding* headers in non-OK responses, since Apache overrides these in *http_protocol.c*, *ap_send_error_response()*:

```
r->content_type = "text/html; charset=iso-8859-1";
```

10.9 More on Relative Paths

Many people use relative paths for `require`, `use`, etc., and when they open files in their scripts they make assumptions about the current directory. This will fail if you don't `chdir()` to the correct directory first (as could easily happen if you have another script which calls the first script by its full path).

For example:

```
/home/httpd/perl/test.pl:
-------------------------
#!/usr/bin/perl
open IN, "./foo.txt";
-------------------------
```

This snippet would work if we call the script like this:

```
% chdir /home/httpd/perl
% ./test.pl
```

since `foo.txt` is located in the current directory. But when the current directory isn't */home/httpd/perl*, if we call the script like this:

```
% /home/httpd/perl/test.pl
```

then the script will fail to find `foo.txt`. Think about `crontab`s!

Notice that you cannot use the `FindBin.pm` package, something that you'd do in the regular Perl scripts, because it relies on the `BEGIN` block it won't work under mod_perl. It's loaded and executed only for the first script executed inside the process, all the others will use the cached value, which would be probably incorrect if they reside in different directories. Perl 5.9.1 provides a new function `FindBin::again` which will do the right thing. Also the CPAN module `FindBin::Real` provides a working alternative working under mod_perl.

10.10 Watching the error_log File Without Telneting to the Server

I wrote this script a long time ago, when I had to debug my CGI scripts but didn't have access to the `error_log` file. I asked the admin to install this script and have used it happily since then.

If your scripts are running on these 'Get-free-site' servers, and you cannot debug your script because you can't telnet to the server or can't see the `error_log`, you can ask your sysadmin to install this script.

Note, that it was written for plain Apache, and isn't prepared to handle the complex multiline error and warning messages generated by mod_perl. It also uses a system() call to do the main work with the tail() utility, probably a more efficient perl implementation is due (take a look at `File::Tail` module). You are welcome to fix it and contribute it back to mod_perl community. Thank you!

Here is the code:

```
# !/usr/bin/perl -Tw

use strict;

my $default   = 10;
my $error_log = "/usr/local/apache/logs/error_log";
use CGI;

# untaint $ENV{PATH}
$ENV{'PATH'} = '/bin:/usr/bin';
delete @ENV{'IFS', 'CDPATH', 'ENV', 'BASH_ENV'};

my $q = new CGI;

my $counts = (defined $q->param('count') and $q->param('count'))
  ? $q->param('count') : $default;

print $q->header,
      $q->start_html(-bgcolor => "white",
                     -title   => "Error logs"),
      $q->start_form,
      $q->center(
               $q->b('How many lines to fetch? '),
               $q->textfield('count',10,3,3),
               $q->submit('', 'Fetch'),
               $q->reset,
              ),
      $q->end_form,
      $q->hr;

# untaint $counts
$counts = ($counts =~ /(\d+)/) ? $1 : 0;

print($q->b("$error_log doesn't exist!!!")),exit unless -e $error_log;

open LOG, "tail -$counts $error_log|"
  or die "Can't tail $error_log :$!\n";
my @logs = <LOG>;
close LOG;
  # format and colorize each line nicely

foreach (@logs) {
    s{
     \[(.*?)\]\s* # date
     \[(.*?)\]\s* # type of error
     \[(.*?)\]\s* # client part
     (.*)         # the message
    }
    {
      "[$1] <BR> [".
      colorize($2,$2).
      "] <BR> [$3] <PRE>".
      colorize($2,$4).
      "</PRE>"
    }ex;
```

```
  print "<BR>$_<BR>";
}



#############
sub colorize{
  my ($type,$context) = @_;

  my %colors =
    (
     error  => 'red',
     crit   => 'black',
     notice => 'green',
     warn   => 'brown',
    );

  return exists $colors{$type}
      ? qq{<B><FONT COLOR="$colors{$type}">$context</FONT></B>}
      : $context;
}
```

10.11 Accessing Variables from the Caller's Package

Sometimes you want to access variables from the caller's package. One way is to do something like this:

```
{
  no strict 'vars' ;
  my $caller = caller;
  print qq[$caller --- ${"${caller}::var"}];
}
```

10.12 Handling Cookies

Unless you use some well known module like `CGI::Cookie` or `Apache::Cookie`, you need to handle cookies yourself.

Cookies come in the `$ENV{HTTP_COOKIE}` variable. You can print the raw cookie string as `$ENV{HTTP_COOKIE}`.

Here is a fairly well-known bit of code to take cookie values and put them into a hash:

```
sub get_cookies {
    # cookies are separated by a semicolon and a space, this will
    # split them and return a hash of cookies
  local(@rawCookies) = split (/; /,$ENV{'HTTP_COOKIE'});
  local(%cookies);

  foreach(@rawCookies){
    ($key, $val) = split (/=/,$_);
    $cookies{$key} = $val;
```

```
    }

    return %cookies;
}
```

Or a slimmer version:

```
    sub get_cookies {
        map { split /=/, $_, 2 } split /; /, $ENV{'HTTP_COOKIE'} ;
    }
```

10.13 Sending Multiple Cookies with the Perl API

Given that you have prepared your cookies in `@cookies`, the following code will submit all the cookies:

```
for (@cookies){
 $r->headers_out->add( 'Set-Cookie' => $_ );
}
```

10.14 Sending Cookies in REDIRECT Response

You should use err_headers_out() and not headers_out() when you want to send cookies in the `REDIRECT` response.

```
use Apache::Constants qw(REDIRECT OK);
my $r = shift;
# prepare the cookie in $cookie
$r->err_headers_out->add('Set-Cookie' => $cookie);
$r->headers_out->set(Location => $location);
$r->status(REDIRECT);
$r->send_http_header;
return OK;
```

10.15 Apache::Cookie example: Login Pages by Setting Cookies and Refreshing

On occassion you will need to set a cookie and then redirect the user to another page. This is probably most common when you want a Location to be password protected, and if the user is unauthenticated, display to them a login page, otherwise display another page, but both at the same URL.

10.15.1 Logic

The logic goes something like this:

- Check for login cookie

- If found, display the page
- If not found, display a login page
- Get username/password from a POST
- Authenticate username/password
- If the authentication failed, re-display the login page
- If the authentication passed, set a cookie and redirect to the same page, and display

10.15.2 Example Situation

Let's say that we are writing a handler for the location */dealers* which is a protected area to be accessed only by people who can pass a username / password authentication check.

We will use `Apache::Cookie` here as it runs pretty fast under mod_perl, but `CGI::Cookie` has pretty much the same syntax, so you can use that if you prefer.

For the purposes of this example, we'll assume that we already have any passed parameters in a *%params* hash, the `authenticate()` routine returns **true** or **false**, *display_login()* shows the username and password prompt, and *display_main_page()* displays the protected content.

10.15.2.1 Code

```
if( $params{user} and $params{pass} ) {
    if(!authenticate(%params)) {
```

Authentication failed, send them back to the login page. **NOTE:** It's a good idea to use `no_cache()` to make sure that the client browser doesn't cache the login page.

```
        $r->content_type('text/html');
        $r->no_cache(1);
        $r->send_http_header;
        display_login();
    } else {
```

The user is authenticated, create the cookie with `Apache::Cookie`

```
        my $c = Apache::Cookie->new( $r,
            -name => 'secret',
            -value => 'foo'
            -expires => '+3d',
            -path => '/dealers'
        );
```

NOTE: when setting the 'expires' tag you must set it with *either* a leading + or -, as if either of these is missing, it will be put literally into the cookie header.

Now send them on their way via the authenticated page

```
        $r->content_type('text/html');
        $c->bake;
        $r->header_out("Refresh"=>"0;url=/dealers");
        $r->no_cache(1);
        $r->send_http_header;
        $r->print( "Authenticated... heading to main page! );
```

The above code will set the headers to refresh (this is the same syntax as for the HTML meta tag) after 0 seconds. The page that is flashed on the screen will have the text in the `$r->print`

```
    }
}
elsif( $cookies{secret} ) {
```

If they already have a secret cookie, display the main (protected) page. Don't forget to check the validity of cookie data!

```
    display_main_page();
}
```

10.16 Passing and Preserving Custom Data Structures Between Handlers

Let's say that you wrote a few handlers to process a request, and they all need to share some custom Perl data structure. The pnotes() method comes to your rescue.

```
# a handler that gets executed first
my %my_data = (foo => 'mod_perl', bar => 'rules');
$r->pnotes('my_data' => \%my_data);
```

The handler prepares the data in hash `%my_data` and calls pnotes() method to store the data internally for other handlers to re-use. All the subsequently called handlers can retrieve the stored data in this way:

```
my $info = $r->pnotes('my_data');
print $info->{foo};
```

prints:

```
mod_perl
```

The stored information will be destroyed at the end of the request.

10.17 Passing Notes Between mod_perl and other (non-Perl) Apache Modules

The notes() method can be used to make various Apache modules talk to each other. In the following snippet the PHP module talks to the mod_perl code (PHP code):

```
if (isset($user) && substr($user,0,1) == "+") {
  apache_note("user", substr($user,1));
  virtual("/internal/getquota");
  $quota      = apache_note("quota");
  $usage_pp   = apache_note("usage_pp");
  $percent_pp = apache_note("percent_pp");
  if ($quota)
    $message .= " | Using $percent_pp% of $quota_pp limit";
}
```

The PHP code sets the *user* and the username pair using the notes mechanism. Then issuing a sub-request to a perl handler:

```
use Apache::Constants qw(REDIRECT OK);
my $r = shift;
my $notes = $r->main->notes();
my ($quota,usage_pp,percent_pp) = getquota($notes->{user}||'');
$r->notes('quota',$quota);
$r->notes('usage_pp',$usage_pp);
$r->notes('percent_pp',$percent_pp);
return OK;
```

which retrieves the username from the notes (using `$r->main->notes`), uses some getquota() function to get the quota related data and then sets the acquired data in the notes for the PHP code. Now the PHP code reads the data from the notes and proceeds with setting `$message` if `$quota` is set.

So any Apache modules can communicate with each other over the Apache notes() mechanism.

You can use notes along with the sub-request methods lookup_uri() and lookup_filename() too. To make it work, you need to set a note in the sub-request. For example if you want to call a php sub-request from within mod_perl and pass it a note, you can do it in the following way:

```
my $subr = $r->lookup_uri('wizard.php3');
$subr->notes('answer' => 42);
$subr->run;
```

As of the time of this writing you cannot access the parent request tables from a PHP handler, therefore you must set this note for the sub-request. Whereas if the sub-request is running in the mod_perl domain, you can always keep the notes in the parent request notes table and access them via the method main():

```
$r->main->notes('answer');
```

10.18 Passing Environment Variables Between Handlers

This is a simple example of passing environment variables between handlers:

Having a configuration:

```
PerlAccessHandler My::Access
PerlLogHandler My::Log
```

and in *startup.pl*:

```
sub My::Access::handler {
  my $r = shift;
  $r->subprocess_env(TICKET => $$);
  $r->notes(TICKET => $$);
}

sub My::Log::handler {
  my $r = shift;
  my $env = $r->subprocess_env('TICKET');
  my $note = $r->notes('TICKET');
  warn "env=$env, note=$note\n";
}
```

Adding `%{TICKET}e` and `%{TICKET}n` to the `LogFormat` for access_log works fine too.

10.19 Verifying Whether A Request Was Received Over An SSL Connection

Just like `$ENV{MODPERL}` is checked to see whether the code is run under mod_perl, `$ENV{HTTPS}` is set by ssl modules and therefore can be used to check whether a request is running over SSL connection. For example:

```
print "SSL" if $ENV{HTTPS};
```

If `PerlSetupEnv Off` setting is in effect, `$ENV{HTTPS}` won't be available, and then:

```
print "SSL" if $r->subprocess_env('https');
```

should be used instead.

Note that it's also possible to check the scheme:

```
print "SSL" if Apache::URI->parse($r)->scheme =~ m/^https/;
```

but it's not one hundred percent certain unless you control the server and you know that you run a secure server on the port 443.

10.20 CGI::params in the mod_perl-ish Way

You can retrieve the request parameters in a similar to `CGI::params` way using this technique:

```
my $r = shift;  # or $r = Apache->request
my %params = $r->method eq 'POST' ? $r->content : $r->args;
```

assuming that all your variables are single key-value pairs.

Also take a look at `Apache::Request` which has the same API as `CGI.pm` for extracting and setting request parameters.

10.21 Subclassing Apache::Request

To subclass a package you simply modify @ISA, for example:

```
package My::TestAPR;

use strict;
use vars qw/@ISA/;
@ISA = qw/Apache::Request/;

sub new {
      my ($proto, $apr) = @_;
      my $class = ref($proto) || $proto;
      bless { _r => $apr }, $class;
}

sub param {
      my ($self, $key) = @_;
      my $apr = $self->{_r};
      # Here we are calling the Apache::Request object's param method
      $apr->param($key);
}

sub sum {
      my ($self, $key) = @_;
      my $apr = $self->{_r};
      my @values = $apr->param($key);
      my $sum = 0;
      for (@values) {
              $sum += $_;
      }
      $sum;
}
1;
__END__
```

10.22 Sending Email from mod_perl

There is nothing special about sending email from mod_perl, it's just that we do it a lot. There are a few important issues. The most widely used approach is starting a `sendmail` process and piping the headers and the body to it. The problem is that `sendmail` is a very heavy process and it makes mod_perl processes less efficient.

If you don’t want your process to wait until delivery is complete, you can tell `sendmail` not to deliver the email straight away, but either do it in the background or just queue the job until the next queue run. This can significantly reduce the delay for the mod_perl process which would otherwise have to wait for the `sendmail` process to complete. This can be specified for all deliveries in *sendmail.cf* or on each invocation on the sendmail command line:

- `-odb` (deliver in the background)
- `-odq` (queue-only) or
- `-odd` (queue, and also defer the DNS/NIS lookups).

The trend is to move away from sendmail(1) and switch to using lighter mail delivery programs like qmail(1) or postfix(1). You should check the manpage of your favorite mailer application for equivalent configuration presented for sendmail(1).

The most efficient approach is to talk directly to the SMTP server. Luckily `Net::SMTP` modules makes this very easy. The only problem is when `Net::SMTP` fails to deliver the mail, because the destination peer server is temporarily down. But from the other side `Net::SMTP` allows you to send email much faster, since you don’t have to invoke a dedicated process. Here is an example of a subroutine that sends email.

```
use Net::SMTP ();
use Carp qw(carp verbose);

#
# Sends email by using the SMTP Server
#
# The SMTP server as defined in Net::Config
# Alternatively you can hardcode it here, look for $smtp_server below
#
sub send_mail{
  my ($from, $to, $subject, $body) = @_;

  carp "From missing" unless defined $from ; # Prefer to exit early if errors
  carp "To missing"   unless defined $to ;

  my $mail_message = <<__END_OF_MAIL__;
To: $to
From: $from
Subject: $subject

$body

__END_OF_MAIL__

    # Set this parameter if you don’t have a valid Net/Config.pm
    # entry for SMTP host and uncomment it in the Net::SMTP->new
    # call
  # my $smtp_server = ’localhost’;

    # init the server
  my $smtp = Net::SMTP->new(
```

```
              # $smtp_server,
              Timeout => 60,
              Debug   => 0,
             );

  $smtp->mail($from) or carp ("Failed to specify a sender [$from]\n");
  $smtp->to($to) or carp ("Failed to specify a recipient [$to]\n");
  $smtp->data([$mail_message]) or carp ("Failed to send a message\n");

  $smtp->quit or carp ("Failed to quit\n");

} #  end of sub send_mail
```

10.23 A Simple Handler To Print The Environment Variables

The code:

```
package MyEnv;
use Apache;
use Apache::Constants;
sub handler{
  my $r = shift;
  print $r->send_http_header("text/plain");
  print map {"$_ => $ENV{$_}\n"} keys %ENV;
  return OK;
}
1;
```

The configuration:

```
PerlModule MyEnv
<Location /env>
  SetHandler perl-script
  PerlHandler MyEnv
</Location>
```

The invocation:

```
http://localhost/env
```

10.24 mod_rewrite in Perl

We can easily implement everything mod_rewrite does in Perl. We do this with help of PerlTransHandler, which is invoked at the beginning of request processing. For example consider that we need to perform a redirect based on query string and URI, the following handler does that.

```
package Apache::MyRedirect;
use Apache::Constants qw(OK REDIRECT);
use constant DEFAULT_URI => 'http://www.example.org';

sub handler {
```

```
  my $r    = shift;
  my %args = $r->args;
  my $path = $r->uri;

  my $uri = (($args{'uri'}) ? $args{'uri'} : DEFAULT_URI) . $path;

  $r->header_out(Location => $uri);
  $r->status(REDIRECT);
  $r->send_http_header;

  return OK;
}
```

Set it up in *httpd.conf* as:

```
PerlTransHandler Apache::MyRedirect
```

The code consists of three parts: request data retrieval, deciding what to do based on this data and finally setting the headers and the status and issuing redirect.

So if a client submits a request of this kind:

```
http://www.example.com/news/?uri=http://www2.example.com/
```

`$uri` will hold *http://www2.example.com/news/* and that's where the request will be redirected.

10.25 URI Rewrite in PerlTransHandler

Suppose that before a content handler is invoked you want to make this translation:

```
/articles/10/index.html  =>  /articles/index.html?id=10
```

This *TransHandler* will do that for you:

```
My/Trans.pm
-----------
package My::Trans;
use Apache::Constants qw(:common);
sub handler {
  my $r = shift;
  my $uri = $r->uri;
  my ($id) = ($uri =~ m|^/articles/(.*?)/|);
  $r->uri("/articles/index.html");
  $r->args("id=$id");
  return DECLINED;
}
1;
```

and in *httpd.conf*:

```
PerlModule My::Trans
PerlTransHandler My::Trans
```

The handler code retrieves the request object and the URI. Then it retrieves the *id* using the regular expression. Finally it sets the new value of the URI and the arguments string. The handler returns `DECLINED` so the default Apache transhandler will take care of URI to filename remapping.

Notice the technique to set the arguments. By the time the Apache-request object has been created, arguments are handled in a separate slot, so you cannot just push them into the original URI. Therefore the args() method should be used.

10.26 Setting PerlHandler Based on MIME Type

It's very easy to implement a dispatching module based on the MIME type of request. So a different content handler will be called for a different MIME type. This is an example of such a dispatcher:

```
package My::MimeTypeDispatch;
use Apache::Constants qw(DECLINED);

my %mime_types = (
  'text/html'  => \&HTML::Template::handler,
  'text/plain' => \&My::Text::handler,
);

sub handler {
  my $r = shift;
  if (my $h = $mime_types{$r->content_type}) {
    $r->push_handlers(PerlHandler => $h);
    $r->handler('perl-script');
  }
return DECLINED;
}
1;
__END__
```

And in *httpd.conf* we add:

```
PerlFixupHandler My::MimeTypeDispatch
```

After declaring the package name and importing constants, we set a translation table of MIME types and corresponding handlers to be called. Then comes the handler, where the request object is retrieved and if its MIME type is found in our translation table we set the handler that should handle this request. Otherwise we do nothing. At the end we return `DECLINED` so some other fixup handler could take over.

10.27 SSI and Embperl -- Doing Both

This handler lets you use both SSI and Embperl in the same request:

Use it in a `<FilesMatch>` Section or similar:

```
PerlModule Apache::EmbperlFilter Apache::SSI
<FilesMatch "\.epl">
   PerlSetVar Filter On
   PerlHandler Apache::EmbperlFilter Apache::SSI
```

```
</FilesMatch>

package Apache::EmbperlFilter;

use Apache::Util qw(parsedate);
use HTML::Embperl;
use Apache::SSI ();
use Apache::Constants;

use strict;
use vars qw($VERSION);

$VERSION = ’0.03’;
my ($r, %param, $input, $output);

sub handler {
    $r = shift;
    my ($fh, $status) = $r->filter_input();
    unless ($status == OK) {
        return $status
    }
    local $/ = undef;
    $input = scalar(<$fh>);
    %param = ();
    $param{input} = \$input;
    $param{req_rec} = $r;
    $param{output} = \$output;
    $param{mtime} = mtime();
    $param{inputfile} = $r->filename();
    HTML::Embperl::ScanEnvironement(\%param);
    HTML::Embperl::Execute(\%param);
    print $output;
    return OK;
}

sub mtime {

    my $mtime = undef;
    if (my $last_modified = $r->headers_out->{’Last-Modified’}) {
        $mtime = parsedate $last_modified;
    }
    $mtime;
}

1;
__END__
```

10.28 Getting the Front-end Server’s Name in the Back-end Server

Assume that you have more than one front-end server, and you want to dynamically figure out the front-end server name in the back-end server. mod_proxy and mod_rewrite provide the solution.

Compile apache with both mod_proxy and mod_rewrite, then use a directive something like this:

```
RewriteEngine On
RewriteLog /somewhere/rewrite.log
RewriteLogLevel 3
RewriteRule ^/foo/bar(.*)$ \
http://example.com:8080/foo/bar/$1?IP=%{REMOTE_HOST} [QSA,P]
```

This will have all the urls starting with */some/url* proxied off to the other server at the same url. It will append the `REMOTE_HOST` header as a query string argument. (QSA = Query String Append, P = Proxy). There is probably a way to remap it as an X-Header of some sort, but if query string is good enough for you, then this should work really nicely.

10.29 Authentication Snippets

Getting the authenticated username: `$r->connection->user()`, or `$ENV{REMOTE_USER}` if you're in a CGI emulation.

Example:

```
my $r = shift;

my ($res, $sent_pwd) = $r->get_basic_auth_pw;
return $res if $res; #decline if not Basic

my $user = $r->connection->user;
```

10.30 Emulating the Authentication Mechanism

You can provide your own mechanism to authenticate users, instead of the standard one. If you want to make Apache think that the user was authenticated by the standard mechanism, set the username with:

```
$r->connection->user('username');
```

Now you can use this information for example during the logging, so that you can have your "username" passed as if it was transmitted to Apache through HTTP authentication.

10.31 An example of using Apache::Session::DBI with cookies

META: should be annotated at some point. (an example was posted to the mod_perl list)

```
use strict;
use DBI;
use Apache::Session::DBI;
use CGI;

# [...]
```

```
# Initiate a session ID
my $session = ();
my $opts = {  autocommit => 0,
              lifetime   => 3600 };     # 3600 is one hour

# Read in the cookie if this is an old session
my $r = Apache->request;
my $no_cookie = '';
my $cookie = $r->header_in('Cookie');
{
    # eliminate logging from Apache::Session::DBI's use of 'warn'
    local $^W = 0;

    if (defined($cookie) && $cookie ne '') {
        $cookie =~ s/SESSION_ID=(\w*)/$1/;
        $session = Apache::Session::DBI->open($cookie, $opts);
        $no_cookie = 'Y' unless defined($session);
    }
    # Could have been obsolete - get a new one
    $session = Apache::Session::DBI->new($opts) unless defined($session);
}

# Might be a new session, so let's give them a cookie back
if (! defined($cookie) || $no_cookie) {
    local $^W = 0;

    my $session_cookie = "SESSION_ID=$session->{'_ID'}";
    $r->header_out("Set-Cookie" => $session_cookie);
}
```

10.32 Detecting a Client Abort

META: should be annotated at some point. (an example was posted to the mod_perl list)

```
# IsClientConnected? Might already be disconnected for busy
# site, if a user hits stop/reload
my $conn = $r->connection;
my $is_connected = $conn->aborted ? 0 : 1;
if ($is_connected) {
    my $fileno = $conn->fileno;
    if (defined $fileno) {
        my $s = IO::Select->new($fileno);
        $is_connected = $s->can_read(0) ? 0 : 1;
    }
}
```

More comments in this thread: http://marc.theaims-group.com/?l=apache-modperl&m=100057943909683&w=2

10.33 Using DESTROY to Finalize Output

Well, as always with Perl -- TMTOWTDI (There's More Than One Way To Do It), one of the readers is using DESTROY to finalize output, and as a cheap means of buffering.

```
package buffer;
use Apache;

sub new {
   my $class = shift;
   my $self = bless {
       'r' => shift,
       'message' => ""
   }, $class;
   $self->{apr} = Apache::Request->new($self->{r},
                                       POST_MAX=>(32*1024));
   $self->content_type('text/plain');
   $self->{r}->no_cache(1);
}

sub message {
      my $self = shift;
      $self->{message} .= join("\n", @_);
}

sub DESTROY {
      my $self = shift;
      $self->{apr}->send_http_header;
      $self->{apr}->print($self->{message});
}
1;
```

Now you can have perl scripts like:

```
use buffer;
my $b = new buffer(shift);

$b->message(p("Hello World"));
# end
```

and save a bunch of duplicate code across otherwise inconvenient gaggles of small scripts.

But suppose you also want to redirect the client under some circumstances, and send the HTTP status code 302. You might try this:

```
sub redir {
      my $self = shift;
      $self->{redirect} = shift;
      exit;
}
```

and re-code DESTROY as:

```
sub DESTROY {
      my $self = shift;
      if ($self->{redirect}) {
              $self->{apr}->status{REDIRECT};
              $self->{apr}->header_out("Location", $self->{redirect});
              $self->{apr}->send_http_header;
              $self->{apr}->print($self->{redirect});
      } else {
              $self->{apr}->send_http_header;
              $self->{apr}->print($self->{message});
      }
}
```

But you'll find that while the browser redirects itself, mod_perl logs the result code as 200. It turns out that status() only touches the Apache response, and the log message is determined by the Apache return code.

Aha! So we'll change the exit() in redir() to exit(REDIRECT). This fixes the log code, but causes a bogus *"[error] 302"* line in the *error_log*. That comes from Apache::Registry:

```
my $errsv = "";
if($@) {
    $errsv = $@;
    $@ = ''; #XXX fix me, if we don't do this Apache::exit() breaks
    $@{$uri} = $errsv;
}

if($errsv) {
    $r->log_error($errsv);
    return SERVER_ERROR unless $Debug && $Debug & 2;
    return Apache::Debug::dump($r, SERVER_ERROR);
}
```

So you see that any time the return code causes $@ to return true, we'll get an error line. Not wanting this, what can we do?

We can hope that a future version of mod_perl will allow us to set the HTTP result code independent from the handler return code (perhaps a log_status() method? or at least an Apache::LOG_HANDLER_RESULT config variable?).

In the meantime, there's Apache::RedirectLogFix, distributed with mod_perl.

Add to your *httpd.conf*:

```
PerlLogHandler Apache::RedirectLogFix
```

and take a look at the source code below. Note that it requires us to return the HTTP status code 200.

```
package Apache::RedirectLogFix;

use Apache::Constants qw(OK DECLINED REDIRECT);
```

```
sub handler {
   my $r = shift;
   return DECLINED unless $r->handler && ($r->handler eq "perl-script");

   if(my $loc = $r->header_out("Location")) {
       if($r->status == 200 and substr($loc, 0, 1) ne "/") {
           $r->status(REDIRECT);
           return OK
       }
   }
   return DECLINED;
}

1;
```

Now, if we wanted to do the same sort of thing for an error 500 handler, we could write another `PerlLogHandler` (call it `ServerErrorLogFix`). But we'll leave that as an exercise for the reader, and hope that it won't be needed in the next mod_perl release. After all, it's a little awkward to need a `LogHandler` to clean up after ourselves....

10.34 Passing Arguments to a SSI script

Consider the following `Apache::Include` snippet:

```
<!--#perl sub="Apache::Include" arg="/perl/ssi.pl" -->
```

Now if you want to pass arguments, you cannot do that with `Apache::Include`. The solution is to define a subroutine that's pulled in at the startup:

```
sub My::ssi {
    my ($r, $one, $two, $three) = @_;
   ...
}
```

In the html file:

```
<!--#perl sub="My::ssi" arg="one" arg="two" arg="three" -->
```

10.35 Setting Environment Variables For Scripts Called From CGI.

Perl uses `sh()` for its `system()` and `open()` calls. So if you want to set a temporary variable when you call a script from your CGI you do something like this:

```
open UTIL, "USER=stas ; script.pl | " or die "...: $!\n";
```

or

```
system "USER=stas ; script.pl";
```

This is useful, for example, if you need to invoke a script that uses CGI.pm from within a mod_perl script. We are tricking the Perl script into thinking it's a simple CGI, which is not running under mod_perl.

```
open(PUBLISH, "GATEWAY_INTERFACE=CGI/1.1 ; script.cgi
     \"param1=value1&param2=value2\" |") or die "...: $!\n";
```

Make sure that the parameters you pass are shell safe -- all "unsafe" characters like single-quote and back-tick should be properly escaped.

Unfortunately mod_perl uses fork() to run the script, so you have probably thrown out the window most of the performance gained from using mod_perl. To avoid the fork, change script.cgi to a module containing a subroutine which you can then call directly from your mod_perl script.

10.36 Mysql Backup and Restore Scripts

This is somewhat off-topic, but since many of us use mysql or some other RDBMS in their work with mod_perl driven sites, it's good to know how to backup and restore the databases in case of database corruption.

First we should tell mysql to log all the clauses that modify the databases (we don't care about SELECT queries for database backups). Modify the `safe_mysql` script by adding the *--log-update* options to the `mysql` server startup parameters and restart the server. From now on all the non-select queries will be logged to the */var/lib/mysql/www.bar.com* logfile. Your hostname will show up instead of *www.bar.com*.

Now create a *dump* directory under */var/lib/mysql/*. That's where the backups will be stored (you can name the directory as you wish of course).

Prepare the backup script and store it in a file, e.g: */usr/local/sbin/mysql/mysql.backup.pl*

This is the original code *code/mysql-3.22.29_backup.pl*:

```
#!/usr/bin/perl -w

# this script should be run from the crontab every night or in shorter
# intervals. This scripts does a few things.
# 1. dump all the tables into a separate dump files (these dump files
# are ready for DB restore)
# 2. backups the last update log file and create a new log file


use strict;
my $data_dir = "/var/lib/mysql";
my $update_log = "$data_dir/www.bar.com";
my $dump_dir  = "$data_dir/dump";
my $gzip_exec = "/bin/gzip";
my @db_names = qw(bugs mysql bonsai);
my $mysql_admin_exec = "/usr/bin/mysqladmin ";

    # convert unix time to date + time
my ($sec,$min,$hour,$mday,$mon,$year) = localtime(time);
```

```
my $time  = sprintf("%0.2d:%0.2d:%0.2d",$hour,$min,$sec);
my $date  = sprintf("%0.2d.%0.2d.%0.4d",++$mon,$mday,$year+1900);
my $timestamp = "$date.$time";

# dump all the DBs we want to backup
foreach my $db_name (@db_names) {
  my $dump_file = "$dump_dir/$timestamp.$db_name.dump";
  my $dump_command = "/usr/bin/mysqldump -c -e -l -q --flush-logs $db_name > $dump_file";
  system $dump_command;
}

# move update log to backup for later restore if needed
rename $update_log, "$dump_dir/$timestamp.log" if -e $update_log;

# restart the update log to log to a new file!
‘/usr/bin/mysqladmin refresh‘;

# compress all the created files
system "$gzip_exec $dump_dir/$timestamp.*";
```

This is the code modified to work with mysql-3.22.30+ *code/mysql-3.22.30+_backup.pl*:

```
#!/usr/bin/perl -w

# this script should be run from the crontab every night or in shorter
# intervals. This scripts does a few things.
# 1. dump all the tables into a separate dump files (these dump files
# are ready for DB restore)
# 2. backups the last update log file and create a new log file

#This script originates from the perl.apache.org site, but I have adapted it to work
#properly with the newer versions of MySQL, where the log files are named differently
#WVW 14/02/2000 w@ba.be

use strict;

my $data_dir = "/var/lib/mysql";
my $update_log = "$data_dir/central2.001";
my $dump_dir  = "$data_dir/backup";
my $gzip_exec = "/bin/gzip";
my @db_names = qw(mysql besup);
my $mysql_admin_exec = "/usr/bin/mysqladmin ";
my $hostname = "central2";

my $password = "babedb";

# convert unix time to date + time
my ($sec,$min,$hour,$mday,$mon,$year) = localtime(time);
my $time  = sprintf("%0.2d:%0.2d:%0.2d",$hour,$min,$sec);
my $date  = sprintf("%0.2d.%0.2d.%0.4d",++$mon,$mday,$year+1900);
my $timestamp = "$date.$time";

# dump all the DBs we want to backup
foreach my $db_name (@db_names) {
  my $dump_file = "$dump_dir/$timestamp.$db_name.dump";
  my $dump_command = "/usr/bin/mysqldump -c -e -l -q --flush-logs -p$password $db_name > $dump_file";
  system $dump_command;
}

mkdir "$dump_dir/$timestamp.log", 0;
‘mv $data_dir/$hostname.[0-9]* $dump_dir/$timestamp.log‘;

# move update log to backup for later restore if needed
```

```
#rename $update_log, "$dump_dir/$timestamp.log" if -e $update_log;

# restart the update log to log to a new file!
‘/usr/bin/mysqladmin refresh -p$password‘;

# compress all the created files
system "$gzip_exec $dump_dir/$timestamp.log/*";
system "$gzip_exec $dump_dir/$timestamp.*.dump*";
```

You might need to change the executable paths according to your system. List the names of the databases you want to backup using the `db_names` array.

Here is another version using `File::Backup`:

```
  #!/usr/bin/perl
  # written by Miroslav Madzarevic, mire@modperldev.com
  use strict;

  umask 0177;

  use File::Backup qw|backup|;

  backup(
     ’from’          => "",
     ’to’            => "/opt/backup/mysql/backup",
     ’torootname’    => "example_backup_",
     ’keep’          => 4,
     ’tar’           => "/usr/bin/mysqldump",
     ’compress’      => "/usr/bin/bzip2",
     ’tarflags’      => "example -uroot -proot_pass -a >",
     ’compressflags’ => "",
     ’tarsuffix’     => ’.sql’,
  );
```

Now make the script executable and arrange the crontab entry to run the backup script nightly. Note that the disk space used by the backups will grow without bound and you should remove the old backups. Here is a sample crontab entry to run the script at 4am every day:

```
  0 4 * * * /usr/local/sbin/mysql/mysql.backup.pl > /dev/null 2>&1
```

So now at any moment we have the dump of the databases from the last execution of the backup script and the log file of all the clauses that have updated the databases since then. If the database gets corrupted we have all the information to restore it to the state it was in at our last backup. We restore it with the following script, which I put in: */usr/local/sbin/mysql/mysql.restore.pl*

This is the original code *code/mysql-3.22.29_restore.pl*:

```
#!/usr/bin/perl -w

# this scripts restores the DBs

# Usage: mysql.restore.pl update.log.gz dump.db1.gz [... dump.dbn.gz]
# all files dump* are compressed as we expect them to be created by
# mysql.backup utility
```

```
# example:
# % mysql.restore.pl myhostname.log.gz 12.10.1998.16:37:12.*.dump.gz

# .dump.gz extension.

use strict;

use FindBin qw($Bin);

my $data_dir   = "/var/lib/mysql";
my $dump_dir   = "$data_dir/dump";
my $gzip_exec  = "/bin/gzip";
my $mysql_exec = "/usr/bin/mysql -f ";
my $mysql_backup_exec = "$Bin/mysql.backup.pl";
my $mysql_admin_exec  = "/usr/bin/mysqladmin ";

my $update_log_file = '';
my @dump_files = ();

# split input files into an update log and the dump files
foreach (@ARGV) {
  push(@dump_files, $_),next unless /\.log\.gz/;
  $update_log_file = $_;
}

die "Usage: mysql.restore.pl update.log.gz dump.db1.gz [... dump.dbn.gz]\n"
  unless defined @dump_files and @dump_files > 0;

# load the dump files
foreach (@dump_files) {

    # check the file exists
  warn("Can't locate $_"),next unless -e $_;

    # extract the db name from the dump file
  my $db_name = $1 if /\d\d\.\d\d.\d\d.\d\d:\d\d:\d\d\.(\w+)\.dump\.gz/;

  warn("Can't extract DB name from the file name,
        probably an error in the file format"),
          next unless defined $db_name and $db_name;

    # we want to drop the table since restore will rebuild it!
    # force to drop the db without confirmation
  my $drop_command = "$mysql_admin_exec -f drop $db_name";
  system $drop_command;

  $drop_command = "$mysql_admin_exec create $db_name";
  system $drop_command;

    # build the command and execute it
  my $restore_command = "$gzip_exec -cd $_ | $mysql_exec $db_name";
  system $restore_command;
}

# now load the update_log file (update the db with the changes since
# the last dump
warn("Can't locate $update_log_file"),next unless  -e $update_log_file;
```

```
my $restore_command =
  "$gzip_exec -cd $update_log_file |$mysql_exec";
system $restore_command;

# rerun the mysql.backup.pl since we have reloaded the dump files
# and update log , and we must rebuild backups!
system $mysql_backup_exec;
```

This is the code modified to work with mysql-3.22.30+ *code/mysql-3.22.30+_restore.pl*:

```
#!/usr/bin/perl -w

# this scripts restores the DBs

# Usage: mysql.restore.pl update.log.gz dump.db1.gz [... dump.dbn.gz]
# all files dump* are compressed as we expect them to be created by
# mysql.backup utility

# example:
# % mysql.restore.pl myhostname.log.gz 12.10.1998.16:37:12.*.dump.gz

# .dump.gz extension.

use strict;

use FindBin qw($Bin);

my $data_dir   = "/var/lib/mysql";
my $dump_dir   = "$data_dir/backup";
my $gzip_exec  = "/bin/gzip";
my $mysql_exec = "/usr/bin/mysql -f -pbabedb";
my $mysql_backup_exec = "$Bin/mysql_backup.pl";
my $mysql_admin_exec  = "/usr/bin/mysqladmin -pbabedb";

my $update_log_dir = '';
my @dump_files = ();

# split input files into an update log and the dump files
foreach (@ARGV) {
  push(@dump_files, $_),next unless /\.log/;
  $update_log_dir = $_;
}

die "Usage: mysql.restore.pl update.log.dir dump.db1.gz [... dump.dbn.gz]\n"
  unless defined @dump_files and @dump_files > 0;

# load the dump files
foreach (@dump_files) {

  # check the file exists
  warn("Can't locate $_"),next unless -e $_;

  # extract the db name from the dump file
  my $db_name = $1 if /\d\d\.\d\d.\d\d\d\d\d.\d\d:\d\d:\d\d\.(\w+)\.dump\.gz/;

  warn("Can't extract DB name from the file name,
```

```
        probably an error in the file format"),
        next unless defined $db_name and $db_name;

  # we want to drop the table since restore will rebuild it!
  # force to drop the db without confirmation
  my $drop_command = "$mysql_admin_exec -f drop $db_name";
  system $drop_command;

  $drop_command = "$mysql_admin_exec create $db_name";
  system $drop_command;

  # build the command and execute it
  my $restore_command = "$gzip_exec -cd $_ | $mysql_exec $db_name";
  system $restore_command;
}

# now load the update_log file (update the db with the changes since
# the last dump
warn("Can't locate $update_log_dir"),next unless  -d $update_log_dir;

my $restore_command =
  "$gzip_exec -cd $update_log_dir/* |$mysql_exec";
system $restore_command;

# rerun the mysql.backup.pl since we have reloaded the dump files
# and update log , and we must rebuild backups!
system $mysql_backup_exec;
```

These are kinda dirty scripts, but they work... if you come up with cleaner scripts, please contribute them... thanks

Update: there is now a "mysqlhotcopy" utility distributed with MySQL that can make an atomic snapshot of a database. (by Tim Bunce) So you may consider using it instead.

10.37 Maintainers

Maintainer is the person(s) you should contact with updates, corrections and patches.

- Stas Bekman [http://stason.org/]

10.38 Authors

- Stas Bekman [http://stason.org/]
- Alan Bailward, <alan (at) ufies.org>

Only the major authors are listed above. For contributors see the Changes file.

11 Apache::* modules

11.1 Description

Overview of some of the most popular modules for mod_perl, both to use directly from your code and as mod_perl handlers.

Over the time, mod_perl has collected an impressive amount of modules which are distributed in the standard Perl way, over CPAN. Found in the `Apache::` namespace, these implement various functionalities you might need when creating a mod_perl-based website. For mod_perl, we can actually make a distinction between two types of modules:

- Apache handlers, which handle request phases or whole requests and are standalone (`Apache::GTopLimit` for example).
- Convenience modules, which are like standard Perl modules, implementing some useful aspect of web programming, usually using mod_perl API for a greater performance or functionality unavailable in plain Perl. (A good example of this is `Apache::Session`.) These modules exist under the `Apache::` namespace because they can only be used under mod_perl.

For a complete list of modules, see the Apache/Perl Modules .

11.2 Apache::Session - Maintain session state across HTTP requests

This module provides the Apache/mod_perl user with a mechanism for storing persistent user data in a global hash, which is independent of the underlying storage mechanism. Currently you can choose from these storage mechanisms `Apache::Session::DBI`, `Apache::Session::Win32`, `Apache::Session::File`, `Apache::Session::IPC`. Read the man page of the mechanism you want to use for a complete reference.

`Apache::Session` provides persistence to a data structure. The data structure has an ID number, and you can retrieve it by using the ID number. In the case of Apache, you would store the ID number in a cookie or the URL to associate it with one browser, but the method of dealing with the ID is completely up to you. The flow of things is generally:

```
Tie a session to Apache::Session.
Get the ID number.
Store the ID number in a cookie.
End of Request 1.

(time passes)

Get the cookie.
Restore your hash using the ID number in the cookie.
Use whatever data you put in the hash.
End of Request 2.
```

Using `Apache::Session` is easy: simply tie a hash to the session object, stick any data structure into the hash, and the data you put in automatically persists until the next invocation. Here is an example which uses cookies to track the user's session.

```
# pull in the required packages
use Apache::Session::DBI;
use Apache;

use strict;

# read in the cookie if this is an old session
my $r = Apache->request;
my $cookie = $r->header_in('Cookie');
$cookie =~ s/SESSION_ID=(\w*)/$1/;

# create a session object based on the cookie we got from the
# browser, or a new session if we got no cookie
my %session;
tie %session, 'Apache::Session::DBI', $cookie,
    {DataSource => 'dbi:mysql:sessions',
     UserName   => $db_user,
     Password   => $db_pass
    };

# might be a new session, so lets give them their cookie back
my $session_cookie = "SESSION_ID=$session{_session_id};";
$r->header_out("Set-Cookie" => $session_cookie);
```

After setting this up, you can stick anything you want into `%session` (except file handles and code references and using *_session_id*), and it will still be there when the user invokes the next page.

It is possible to write an Apache authentication handler using `Apache::Session`. You can put your authentication token into the session. When a user invokes a page, you open their session, check to see if they have a valid token, and authenticate or forbid based on that.

By way of comparison note that IIS's sessions are only valid on the same web server as the one that issued the session. `Apache::Session`'s session objects can be shared amongst a farm of many machines running different operating systems, including even Win32. IIS stores session information in RAM. `Apache::Session` stores sessions in databases, file systems, or RAM. IIS's sessions are only good for storing scalars or arrays. `Apache::Session`'s sessions allow you to store arbitrarily complex objects. IIS sets up the session and automatically tracks it for you. With `Apache::Session`, you setup and track the session yourself. IIS is proprietary. `Apache::Session` is open-source. `Apache::Session::DBI` can issue 400+ session requests per second on light Celeron 300A running Linux. IIS?

An alternative to `Apache::Session` is `Apache::ASP`, which has session tracking abilities. `HTML::Embperl` hooks into `Apache::Session` for you.

11.3 Apache::DBI - Initiate a persistent database connection

See mod_perl and relational Databases

11.4 Apache::Watchdog::RunAway - Hanging Processes Monitor and Terminator

This module monitors hanging Apache/mod_perl processes. You define the time in seconds after which the process is to be counted as *hanging* or *run away*.

When the process is considered to be *hanging* it will be killed and the event logged in a log file.

Generally you should use the `amprapmon` program that is bundled with this module's distribution package, but you can write your own code using the module as well. See the *amprapmon* manpage for more information about it.

Note that it requires the `Apache::Scoreboard` module to work.

Refer to the `Apache::Watchdog::RunAway` manpage for the configuration details.

11.5 Apache::VMonitor -- Visual System and Apache Server Monitor

`Apache::VMonitor` is the next generation of mod_status. It provides all the information mod_status provides and much more.

This module emulates the reporting functions of the top(), mount(), df() and ifconfig() utilities. There is a special mode for mod_perl processes. It has visual alert capabilities and a configurable *automatic refresh* mode. It provides a Web interface, which can be used to show or hide all the sections dynamically.

The are two main modes:

- Multi processes mode -- All system processes and information is shown.
- Single process mode -- In-depth information about a single process is shown.

The main advantage of this module is that it reduces the need to telnet to the machine in order to monitor it. Indeed it provides information about mod_perl processes that cannot be acquired from telneting to the machine.

11.5.0.1 Configuration

```
# Configuration in httpd.conf
<Location /sys-monitor>
  SetHandler perl-script
  PerlHandler Apache::VMonitor
```

```
</Location>

# startup file or <Perl> section:
use Apache::VMonitor();
$Apache::VMonitor::Config{BLINKING} = 0; # Blinking is evil
$Apache::VMonitor::Config{REFRESH}  = 0;
$Apache::VMonitor::Config{VERBOSE}  = 0;
$Apache::VMonitor::Config{SYSTEM}   = 1;
$Apache::VMonitor::Config{APACHE}   = 1;
$Apache::VMonitor::Config{PROCS}    = 1;
$Apache::VMonitor::Config{MOUNT}    = 1;
$Apache::VMonitor::Config{FS_USAGE} = 1;
$Apache::VMonitor::Config{NETLOAD}  = 1;

@Apache::VMonitor::NETDEVS    = qw(lo eth0);
$Apache::VMonitor::PROC_REGEX = join "\|", qw(httpd mysql squid);
```

More information is available in the module's extensive manpage.

It requires `Apache::Scoreboard` and `GTop` to work. `GTop` in turn requires the `libgtop` library but is not available for all platforms. See the docs in the source at ftp://ftp.gnome.org/pub/GNOME/stable/sources/gtop/ to check whether your platform/flavor is supported.

11.6 Apache::GTopLimit - Limit Apache httpd processes

This module allows you to kill off Apache processes if they grow too large or if they share too little of their memory. You can choose to set up the process size limiter to check the process size on every request:

The module is thoroughly explained in the section: Preventing Your Processes from Growing

11.7 Apache::Request (libapreq) - Generic Apache Request Library

This package contains modules for manipulating client request data via the Apache API with Perl and C. Functionality includes:

- **parsing of application/x-www-form-urlencoded data**
- **parsing of multipart/form-data**
- **parsing of HTTP Cookies**

The Perl modules are simply a thin xs layer on top of libapreq, making them a lighter and faster alternative to CGI.pm and CGI::Cookie. See the `Apache::Request` and `Apache::Cookie` documentation for more details and eg/perl/ for examples.

`Apache::Request` and libapreq are tied tightly to the Apache API, to which there is no access in a process running under mod_cgi.

(Apache::Request)

11.8 Apache::RequestNotes - Allow Easy, Consistent Access to Cookie and Form Data Across Each Request Phase

`Apache::RequestNotes` provides a simple interface allowing all phases of the request cycle access to cookie or form input parameters in a consistent manner. Behind the scenes, it uses libapreq Apache::Request) functions to parse request data and puts references to the data in `pnotes()`.

Once the request is past the PerlInit phase, all other phases can have access to form input and cookie data without parsing it themselves. This relieves some strain, especially when the GET or POST data is required by numerous handlers along the way.

See the `Apache::RequestNotes` manpage for more information.

11.9 Apache::PerlRun - Run unaltered CGI scripts under mod_perl

See Apache::PerlRun - a closer look.

11.10 Apache::RegistryNG -- Apache::Registry New Generation

`Apache::RegistryNG` is the same as `Apache::Registry`, aside from using filenames instead of URIs for namespaces. This feature ensures that if the same CGI script is requested from different URIs (e.g. different hostnames) it'll be compiled and cached only once, thus saving memory.

`Apache::RegistryNG` uses an Object Oriented interface.

```
PerlModule Apache::RegistryNG
<Location /perl>
  SetHandler perl-script
  PerlHandler Apache::RegistryNG->handler
</Location>
```

`Apache::RegistryNG` inherits from `Apache::PerlRun`, but the handler() is overridden. Aside from the handler(), the rest of `Apache::PerlRun` contains all the functionality of `Apache::Registry` broken down into several subclass-able methods. These methods are used by `Apache::RegistryNG` to implement the exact same functionality of `Apache::Registry`, using the `Apache::PerlRun` methods.

There is no compelling reason to use `Apache::RegistryNG` over `Apache::Registry`, unless you want to do add or change the functionality of the existing *Registry.pm* or if you want to use filenames instead of URIs for namespaces. For example, `Apache::RegistryBB` (Bare-Bones) is another subclass that skips the stat() call performed by `Apache::Registry` on each request.

11.11 Apache::RegistryBB -- Apache::Registry Bare Bones

It works just like `Apache::Registry`, but does not test the x bit (-x file test for executable mode), only compiles the file once (no stat() call is made per request), skips the `OPT_EXECCGI` checks and does not `chdir()` into the script parent directory. It uses the Object Oriented interface.

Configuration:

```
PerlModule Apache::RegistryBB
<Location /perl>
  SetHandler perl-script
  PerlHandler Apache::RegistryBB->handler
</Location>
```

11.12 Apache::OutputChain -- Chain Stacked Perl Handlers

Apache::OutputChain was written as a way of exploring the possibilities of stacked handlers in mod_perl. It ties STDOUT to an object which catches the output and makes it easy to build a chain of modules that work on output data stream.

Examples of modules that are build on this idea are `Apache::SSIChain`, `Apache::GzipChain` and `Apache::EmbperlChain` -- the first processes the SSI's in the stream, the second compresses the output on the fly, the last adds Embperl processing.

The syntax goes like this:

```
<Files *.html>
  SetHandler perl-script
  PerlHandler Apache::OutputChain Apache::SSIChain Apache::PassHtml
</Files>
```

The modules are listed in the reverse order of their execution -- here the `Apache::PassHtml` module simply picks a file's content and sends it to STDOUT, then it's processed by `Apache::SSIChain`, which sends its output to STDOUT again. Then it's processed by `Apache::OutputChain`, which finally sends the result to the browser.

An alternative to this approach is `Apache::Filter`, which has a more natural *forward* configuration order and is easier to interface with other modules.

It works with `Apache::Registry` as well, for example:

```
Alias /foo /home/httpd/perl/foo
<Location /foo>
  SetHandler "perl-script"
  Options +ExecCGI
  PerlHandler Apache::OutputChain Apache::GzipChain Apache::Registry
</Location>
```

It's really a regular `Apache::Registry` setup, except for the added modules in the PerlHandler line.

(Apache::GzipChain allows to compress the output on the fly.)

11.13 Apache::Filter - Alter the output of previous handlers

`Apache::Filter`, like `Apache::OutputChain`, allows you to chain stacked handlers. It's not very different from `Apache::OutputChain`, except for the way you configure the filters. A normal configuration with `Apache::Filter` would be the following:

```
PerlModule Apache::Filter Apache::RegistryFilter Apache::SSI Apache::Gzip
Alias /perl /home/httpd/perl
<Location /perl>
   SetHandler "perl-script"
   Options +ExecCGI
   PerlSetVar Filter On
   PerlHandler Apache::RegistryFilter Apache::SSI Apache::Gzip
</Location>
```

This accomplishes some things many CGI programmers want: you can output SSI code from your `Apache::Registry` scripts, have it parsed by `Apache::SSI`, and then compressed with `Apache::Gzip` (see Apache::Gzip below).

Thanks to `Apache::Filter`, you can also write your own filter modules, which allow you to read in the output from the previous handler in the chain and modify it. You would do something like this in your `handler` subroutine:

```
$r = $r->filter_register();  # Required
my $fh = $r->filter_input(); # Optional (you might not need the input FH)
while (<$fh>) {
    s/ something / something else /;
    print;
}
```

Another interesting thing to do with `Apache::Filter` would be to use it for XML output from your scripts(these modules are hypothetical, this is handled much better by AxKit, Matt Seargeant's XML application server for mod_perl (see http://www.axkit.org/).

```
<Location /perl/xml-output>
   SetHandler perl-script
   Options +ExecCGI
   PerlSetVar Filter On
   PerlHandler Apache::RegistryFilter Apache::XSLT
</Location>
```

As you can see, you can get a lot of freedom by using stacked handlers, allowing you to separate various parts of your programs and leave those tasks up to other modules, which may already be available from CPAN (this is much better than the CGI time when your script would have to do *everything* itself, because you couldn't do much with its output).

11.14 Apache::GzipChain - compress HTML (or anything) in the OutputChain

Have you ever served a huge HTML file (e.g. a file bloated with JavaScript code) and wondered how could you send it compressed, thus dramatically cutting down the download times? After all Java applets can be compressed into a jar and benefit from faster download times. Why can't we do the same with plain ASCII (HTML, JS etc.)? ASCII text can often be compressed by a factor of 10.

`Apache::GzipChain` comes to help you with this task. If a client (browser) understands `gzip` encoding, this module compresses the output and sends it downstream. The client decompresses the data upon receipt and renders the HTML as if it were fetching plain HTML.

For example to compress all html files on the fly, do this:

```
<Files *.html>
  SetHandler perl-script
  PerlHandler Apache::OutputChain Apache::GzipChain Apache::PassFile
</Files>
```

Remember that it will work only if the browser claims to accept compressed input, by setting the `Accept-Encoding` header. `Apache::GzipChain` keeps a list of user-agents, thus it also looks at the `User-Agent` header to check for browsers known to accept compressed output.

For example if you want to return compressed files which will in addition pass through the Embperl module, you would write:

```
<Location /test>
  SetHandler perl-script
  PerlHandler Apache::OutputChain Apache::GzipChain Apache::EmbperlChain Apache::PassFile
</Location>
```

Hint: Watch the *access_log* file to see how many bytes were actually sent, and compare that with the bytes sent using a regular configuration.

(See also `Apache::GzipChain`).

Notice that the rightmost PerlHandler must be a content producer. Here we are using `Apache::PassFile` but you can use any module which creates output.

11.15 Apache::Gzip - Auto-compress web files with Gzip

Similar to `Apache::GzipChain` but works with `Apache::Filter`.

This configuration:

```
PerlModule Apache::Filter
<Files ~ "*\.html">
  SetHandler perl-script
  PerlSetVar Filter On
  PerlHandler Apache::Gzip
</Files>
```

will send all the **.html* files compressed if the client accepts the compressed input.

And this one:

```
PerlModule Apache::Filter
Alias /home/http/perl /perl
<Location /perl>
  SetHandler perl-script
  PerlSetVar Filter On
  PerlHandler Apache::RegistryFilter Apache::Gzip
</Location>
```

will compress the output of the `Apache::Registry` scripts. Yes, you should use `Apache::RegistryFilter` instead of `Apache::Registry` for it to work.

You can use as many filters as you want:

```
PerlModule Apache::Filter
<Files ~ "*\.blah">
  SetHandler perl-script
  PerlSetVar Filter On
  PerlHandler Filter1 Filter2 Apache::Gzip
</Files>
```

You can test that it works by either looking at the size of the response in the *access.log* or by telnet:

```
panic% telnet localhost 8000
Trying 127.0.0.1
Connected to 127.0.0.1
Escape character is '^]'.
GET /perl/test.pl HTTP 1.1
Accept-Encoding: gzip
User-Agent: Mozilla
```

And you will get the data compressed if configured correctly.

11.16 Apache::PerlVINC - Allows Module Versioning in Location blocks and Virtual Hosts

With this module you can have different `@INC` for different virtual hosts, locations and equivalent configuration blocks.

Suppose two versions of `Apache::Status` are being hacked on the same server. In this configuration:

```
PerlModule Apache::PerlVINC

<Location /status-dev/perl>
  SetHandler       perl-script
  PerlHandler      Apache::Status

  PerlINC          /home/httpd/dev/lib
  PerlFixupHandler Apache::PerlVINC
  PerlVersion      Apache/Status.pm
</Location>

<Location /status/perl>
  SetHandler       perl-script
  PerlHandler      Apache::Status

  PerlINC          /home/httpd/prod/lib
  PerlFixupHandler Apache::PerlVINC
  PerlVersion      Apache/Status.pm
</Location>
```

The `Apache::PerlVINC` is loaded and then two different locations are specified for the same handler `Apache::Status`, whose development version resides in */home/httpd/dev/lib* and production version in */home/httpd/prod/lib*.

In case the */status/perl* request is issued (the latter configuration section), the fixup handler will internally do:

```
delete $INC{Apache/Status.pm};
unshift @INC, /home/httpd/prod/lib;
require "Apache/Status.pm";
```

which will load the production version of the module and it'll be used to process the request. If on the other hand if the request to the */status-dev/perl* location will be issued, as configured in the former configuration section, a similar thing will happen, but a different path (*/home/httpd/dev/lib*) will be prepended to `@INC`:

```
delete $INC{Apache/Status.pm};
unshift @INC, /home/httpd/dev/lib;
require "Apache/Status.pm";
```

It's important to be aware that a changed `@INC` is effective only inside the `<Location>` or a similar configuration directive. `Apache::PerlVINC` subclasses the `PerlRequire` directive, marking the file to be reloaded by the fixup handler, using the value of `PerlINC` for `@INC`. That's local to the fixup handler, so you won't actually see `@INC` changed in your script.

In addition the modules with different versions can be unloaded at the end of request, using the `PerlCleanupHandler` handler:

```
<Location /status/perl>
  SetHandler          perl-script
  PerlHandler         Apache::Status

  PerlINC             /home/httpd/prod/lib
  PerlFixupHandler    Apache::PerlVINC
  PerlCleanupHandler Apache::PerlVINC
  PerlVersion         Apache/Status.pm
</Location>
```

Also notice that `PerlVersion` effect things differently depending on where it was placed. If it was placed inside a `<Location>` or a similar block section, the files will only be reloaded on requests to that location. If it was placed in a server section, all requests to the server or virtual hosts will have these files reloaded.

As you can guess, this module slows the response time down because it reloads some modules on a per-request basis. Hence, this module should only be used in a development environment, not a production one.

11.17 Apache::LogSTDERR

When Apache's builtin syslog support is used, the stderr stream is redirected to `/dev/null`. This means that Perl warnings, any messages from `die()`, `croak()`, etc., will also end up in the black hole. The *HookStderr* directive will hook the stderr stream to a file of your choice, the default is shown in this example:

```
PerlModule Apache::LogSTDERR
HookStderr logs/stderr_log
```

[META: see http://mathforum.org/epigone/modperl/vixquimwhen]

11.18 Apache::RedirectLogFix

Because of the way mod_perl handles redirects, the status code is not properly logged. The `Apache::RedirectLogFix` module works around that bug until mod_perl can deal with this. All you have to do is to enable it in the *httpd.conf* file.

```
PerlLogHandler Apache::RedirectLogFix
```

For example, you will have to use it when doing:

```
$r->status(304);
```

and do some manual header sending, like this:

```
$r->status(304);
$r->send_http_header();
```

11.19 Apache::SubProcess

The output of `system()`, `exec()`, and `open(PIPE,"|program")` calls will not be sent to the browser unless your Perl was configured with `sfio`.

One workaround is to use backticks:

```
print ‘command here‘;
```

But a cleaner solution is provided by the `Apache::SubProcess` module. It overrides the exec() and system() calls with calls that work correctly under mod_perl.

Let’s see a few examples:

```
use Apache::SubProcess qw(system);
my $r = shift;
$r->send_http_header(’text/plain’);

system "/bin/echo hi there";
```

overrides built-in system() function and sends the output to the browser.

```
use Apache::SubProcess qw(exec);
my $r = shift;
$r->send_http_header(’text/plain’);

exec "/usr/bin/cal";

print "NOT REACHED\n";
```

overrides built-in exec() function and sends the output to the browser. As you can see the print statement after the exec() call will be never executed.

```
use Apache::SubProcess ();
my $r = shift;
$r->send_http_header(’text/plain’);

my $efh = $r->spawn_child(\&env);
$r->send_fd($efh);

sub env {
    my $r = shift;
    $r->subprocess_env(HELLO => ’world’);
    $r->filename("/bin/env");
    $r->call_exec;
}
```

env() is a function that sets an environment variable that can be seen by the main and sub-processes, then it executes */bin/env* program via call_exec(). The main code spawn a process, and tells it to execute the env() function. This call returns an output filehandler from the spawned child process. Finally it takes the output generated by the child process and sends it to the browser via send_fd(), that expects the filehandler as an argument.

```
use Apache::SubProcess ();
my $r = shift;
$r->send_http_header('text/plain');

my $fh = $r->spawn_child(\&banner);
$r->send_fd($fh);

sub banner {
    my $r = shift;
    # /usr/games/banner on many Unices
    $r->filename("/usr/bin/banner");
    $r->args("-w40+Hello%20World");
    $r->call_exec;
}
```

This example is very similar to the previous, but shows how can you pass arguments to the external process. It passes the string to print as a banner to via a subprocess.

```
use Apache::SubProcess ();
my $r = shift;
$r->send_http_header('text/plain');

use vars qw($String);
$String = "hello world";
my ($out, $in, $err) = $r->spawn_child(\&echo);
print $out $String;
$r->send_fd($in);

sub echo {
    my $r = shift;
    $r->subprocess_env(CONTENT_LENGTH => length $String);
    $r->filename("/tmp/pecho");
    $r->call_exec;
}
```

The last example shows how you can have a full access to STDIN, STDOUT and STDERR streams of the spawned sub process, so you can pipe data to a program and send its output to the browser. The echo() function is similar to the earlier example's env() function. The */tmp/pecho* is as follows:

```
!/usr/bin/perl
read STDIN, $buf, $ENV{CONTENT_LENGTH};
print "STDIN: '$buf' ($ENV{CONTENT_LENGTH})\n";
```

So in the last example a string is defined as a global variable, so it's length could be calculated in the echo() function. The subprocess reads from STDIN, to which the main process writes the string (*hello world*). It reads only a number of bytes specified by `CONTENT_LENGTH` passed to the external program via environment variable. Finally the external program prints the data that it read to STDOUT, the main program intercepts it and sends to the client's socket (browser in most cases).

11.20 Module::Use - Log and Load used Perl modules

`Module::Use` records the modules used over the course of the Perl interpreter's lifetime. If the logging module is able, the old logs are read and frequently used modules are automatically loaded.

For example if configured as:

```
<Perl>
    use Module::Use (Counting, Logger => "Debug");
</Perl>

PerlChildExitHandler Module::Use
```

it will only record the used modules when the child exists, logging everything (debug level).

11.21 Apache::ConfigFile - Parse an Apache style httpd.conf config file

This module parses *httpd.conf*, or any compatible config file, and provides methods for accessing the values from the parsed file.

See the module manpage for more information.

11.22 Apache::Admin::Config - Object oriented access to Apache style config files

`Apache::Admin::Config` provides an object oriented interface for reading and writing Apache-like configuration files without affecting comments, indentation, or truncated lines. You can easily extract informations from the apache configuration, or manage htaccess files.

See http://rs.rhapsodyk.net/devel/apache-admin-config/ for more information.

11.23 Maintainers

Maintainer is the person(s) you should contact with updates, corrections and patches.

- Stas Bekman [http://stason.org/]

11.24 Authors

- Stas Bekman [http://stason.org/]

Only the major authors are listed above. For contributors see the Changes file.

12 Choosing the Right Strategy

12.1 Description

This document discusses various mod_perl setup strategies used to get the best performance and scalability of the services.

12.2 Do it like I do it!?

There is no such thing as the **right** strategy in the web server business, although there are many wrong ones. Never believe a person who says: *"Do it this way, this is the best!"*. As the old saying goes: *"Trust but verify"*. There are too many technologies out there to choose from, and it would take an enormous investment of time and money to try to validate each one before deciding which is the best choice for your situation.

With this in mind, I will present some ways of using standalone mod_perl, and some combinations of mod_perl and other technologies. I'll describe how these things work together, offer my opinions on the pros and cons of each, the relative degree of difficulty in installing and maintaining them, and some hints on approaches that should be used and things to avoid.

To be clear, I will not address all technologies and tools, but limit this discussion to those complementing mod_perl.

Please let me stress it again: **do not** blindly copy someone's setup and hope for a good result. Choose what is best for your situation -- it might take **some** effort to find out what that is.

In this chapter we will discuss

- **Deployment of mod_perl in Overview, with the pros and cons.**
- **Alternative architectures for running one and two servers.**
- **Proxy servers (Squid, and Apache's mod_proxy).**

12.3 mod_perl Deployment Overview

There are several different ways to build, configure and deploy your mod_perl enabled server. Some of them are:

1. Having one binary and one configuration file (one big binary for mod_perl).

2. Having two binaries and two configuration files (one big binary for mod_perl and one small binary for static objects like images.)

3. Having one DSO-style binary and two configuration files, with mod_perl available as a loadable object.

4. Any of the above plus a reverse proxy server in http accelerator mode.

If you are a newbie, I would recommend that you start with the first option and work on getting your feet wet with Apache and mod_perl. Later, you can decide whether to move to the second one which allows better tuning at the expense of more complicated administration, or to the third option -- the more state-of-the-art-yet-suspiciously-new DSO system, or to the fourth option which gives you even more power.

1. The first option will kill your production site if you serve a lot of static data from large (4 to 15MB) webserver processes. On the other hand, while testing you will have no other server interaction to mask or add to your errors.

2. This option allows you to tune the two servers individually, for maximum performance.

   However, you need to choose between running the two servers on multiple ports, multiple IPs, etc., and you have the burden of administering more than one server. You have to deal with proxying or fancy site design to keep the two servers in synchronization.

3. With DSO, modules can be added and removed without recompiling the server, and their code is even shared among multiple servers.

   You can compile just once and yet have more than one binary, by using different configuration files to load different sets of modules. The different Apache servers loaded in this way can run simultaneously to give a setup such as described in the second option above.

   On the down side, you are playing at the bleeding edge.

   You are dealing with a new solution that has weak documentation and is still subject to change. It is still somewhat platform specific. Your mileage may vary.

   The DSO module (`mod_so`) adds size and complexity to your binaries.

   Refer to the section "Pros and Cons of Building mod_perl as DSO for more information.

   Build details: Build mod_perl as DSO inside Apache source tree via APACI

4. The fourth option (proxy in http accelerator mode), once correctly configured and tuned, improves the performance of any of the above three options by caching and buffering page results.

12.4 Alternative architectures for running one and two servers

The next part of this chapter discusses the pros and the cons of each of these presented configurations. Real World Scenarios Implementation describes the implementation techniques of these schemes.

We will look at the following installations:

- **Standalone mod_perl Enabled Apache Server**
- **One Plain Apache and One mod_perl-enabled Apache Servers**
- **One light non-Apache and One mod_perl enabled Apache Servers**
- **Adding a Proxy Server in http Accelerator Mode**

12.4.1 Standalone mod_perl Enabled Apache Server

The first approach is to implement a straightforward mod_perl server. Just take your plain Apache server and add mod_perl, like you add any other Apache module. You continue to run it at the port it was using before. You probably want to try this before you proceed to more sophisticated and complex techniques.

The advantages:

- Simplicity. You just follow the installation instructions, configure it, restart the server and you are done.

- No network changes. You do not have to worry about using additional ports as we will see later.

- Speed. You get a very fast server and you see an enormous speedup from the first moment you start to use it.

The disadvantages:

- The process size of a mod_perl-enabled Apache server is huge (maybe 4MB at startup and growing to 10MB and more, depending on how you use it) compared to typical plain Apache. Of course if memory sharing is in place, RAM requirements will be smaller.

  You probably have a few tens of child processes. The additional memory requirements add up in direct relation to the number of child processes. Your memory demands are growing by an order of magnitude, but this is the price you pay for the additional performance boost of mod_perl. With memory prices so cheap nowadays, the additional cost is low -- especially when you consider the dramatic performance boost mod_perl gives to your services with every 100MB of RAM you add.

  While you will be happy to have these monster processes serving your scripts with monster speed, you should be very worried about having them serve static objects such as images and html files. Each static request served by a mod_perl-enabled server means another large process running, competing for system resources such as memory and CPU cycles. The real overhead depends on the static object request rate. Remember that if your mod_perl code produces HTML code which includes images, each one will turn into another static object request. Having another plain webserver to serve the static objects solves this unpleasant obstacle. Having a proxy server as a front end, caching the static objects and freeing the mod_perl processes from this burden is another solution. We will discuss both below.

- Another drawback of this approach is that when serving output to a client with a slow connection, the huge mod_perl-enabled server process (with all of its system resources) will be tied up until the response is completely written to the client. While it might take a few milliseconds for your script to complete the request, there is a chance it will be still busy for some number of seconds or even minutes if the request is from a slow connection client. As in the previous drawback, a proxy solution

can solve this problem. More on proxies later.

Proxying dynamic content is not going to help much if all the clients are on a fast local net (for example, if you are administering an Intranet.) On the contrary, it can decrease performance. Still, remember that some of your Intranet users might work from home through slow modem links.

If you are new to mod_perl, this is probably the best way to get yourself started.

And of course, if your site is serving only mod_perl scripts (close to zero static objects, like images), this might be the perfect choice for you!

For implementation notes, see the "One Plain and One mod_perl enabled Apache Servers" section in the implementations chapter.

12.4.2 One Plain Apache and One mod_perl-enabled Apache Servers

As I have mentioned before, when running scripts under mod_perl you will notice that the httpd processes consume a huge amount of virtual memory -- from 5MB to 15MB and even more. That is the price you pay for the enormous speed improvements under mod_perl. (Again -- shared memory keeps the real memory that is being used much smaller :)

Using these large processes to serve static objects like images and html documents is overkill. A better approach is to run two servers: a very light, plain Apache server to serve static objects and a heavier mod_perl-enabled Apache server to serve requests for dynamic (generated) objects (aka CGI).

From here on, I will refer to these two servers as **httpd_docs** (vanilla Apache) and **httpd_perl** (mod_perl enabled Apache).

The advantages:

- The heavy mod_perl processes serve only dynamic requests, which allows the deployment of fewer of these large servers.
- `MaxClients`, `MaxRequestsPerChild` and related parameters can now be optimally tuned for both `httpd_docs` and `httpd_perl` servers, something we could not do before. This allows us to fine tune the memory usage and get better server performance.

  Now we can run many lightweight `httpd_docs` servers and just a few heavy `httpd_perl` servers.

An **important** note: When a user browses static pages and the base URL in the **Location** window points to the static server, for example `http://www.example.com/index.html` -- all relative URLs (e.g. `<A HREF="/main/download.html">`) are being served by the light plain Apache server. But this is not the case with dynamically generated pages. For example when the base URL in the **Location** window points to the dynamic server -- (e.g. `http://www.example.com:8080/perl/index.pl`) all relative URLs in the dynamically generated HTML will be served by the heavy mod_perl processes. You must use fully qualified URLs and not relative ones!
`http://www.example.com/icons/arrow.gif` is a full URL, while `/icons/arrow.gif` is a

relative one. Using `<BASE HREF="http://www.example.com/">` in the generated HTML is another way to handle this problem. Also, the `httpd_perl` server could rewrite the requests back to `httpd_docs` (much slower) and you still need the attention of the heavy servers. This is not an issue if you hide the internal port implementations, so the client sees only one server running on port `80`. (See Publishing Port Numbers other than 80)

The disadvantages:

- An administration overhead.
  - The need for two configuration files.
  - The need for two sets of controlling scripts (startup/shutdown) and watchdogs.
  - If you are processing log files, now you probably will have to merge the two separate log files into one before processing them.
- Just as in the one server approach, we still have the problem of a mod_perl process spending its precious time serving slow clients, when the processing portion of the request was completed a long time ago. Deploying a proxy solves this, and will be covered in the next section.

  As with the single server approach, this is not a major disadvantage if you are on a fast network (i.e. Intranet). It is likely that you do not want a buffering server in this case.

Before you go on with this solution you really want to look at the Adding a Proxy Server in http Accelerator Mode section.

For implementation notes see the "One Plain and One mod_perl enabled Apache Servers" section in implementations chapter.

12.4.3 One light non-Apache and One mod_perl enabled Apache Servers

If the only requirement from the light server is for it to serve static objects, then you can get away with non-Apache servers having an even smaller memory footprint. `thttpd` has been reported to be about 5 times faster then Apache (especially under a heavy load), since it is very simple and uses almost no memory (260K) and does not spawn child processes.

The Advantages:

- All the advantages of the 2 servers scenario.
- More memory saving. Apache is about 4 times bigger than **thttpd**, if you spawn 30 children you use about 30M of memory, while **thttpd** uses only 260K - 100 times less! You could use the 30M you've saved to run a few more mod_perl servers.

The memory savings are significantly smaller if your OS supports memory sharing with Dynamically Shared Objects (DSO) and you have configured Apache to use it. If you do allow memory sharing, 30 light Apache servers ought to use only about 3 to 4MB, because most of it will be shared. There is no memory sharing if Apache modules are statically compiled into the httpd executable.

- Reported to be about 5 times faster then plain Apache serving static objects.

The Disadvantages:

- Lacks some of Apache's features, like access control, error redirection, customizable log file formats, and so on.

Another interesting choice is a kHTTPd webserver for Linux. kHTTPd is different from other webservers in that it runs from within the Linux-kernel as a module (device-driver). kHTTPd handles only static (file based) web-pages, and passes all requests for non-static information to a regular userspace-webserver such as Apache. For more information see http://www.fenrus.demon.nl/.

Also check out the Boa webserver: http://www.boa.org/

12.5 Adding a Proxy Server in http Accelerator Mode

At the beginning there were two servers: one plain Apache server, which was *very light*, and configured to serve static objects, the other mod_perl enabled (*very heavy*) and configured to serve mod_perl scripts and handlers. As you remember we named them `httpd_docs` and `httpd_perl` respectively.

In the dual-server setup presented earlier the two servers coexist at the same IP address by listening to different ports: `httpd_docs` listens to port 80 (e.g. http://www.example.com/images/test.gif) and `httpd_perl` listens to port 8080 (e.g. http://www.example.com:8080/perl/test.pl). Note that we did not write http://www.example.com:80 for the first example, since port 80 is the default port for the http service. Later on, we will be changing the configuration of the `httpd_docs` server to make it listen to port 81.

This section will attempt to convince you that you really **want** to deploy a proxy server in the http accelerator mode. This is a special mode that in addition to providing the normal caching mechanism, accelerates your CGI and mod_perl scripts.

The advantages of using the proxy server in conjunction with mod_perl are:

- Certainly the benefits of the usual use of the proxy server which allows serving of static objects from the proxy's cache. You get less I/O activity reading static objects from the disk (proxy serves the most "popular" objects from RAM -- of course you benefit more if you allow the proxy server to consume more RAM). Since you do not wait for the I/O to be completed, you are able to serve static objects much faster.

- And the extra functionality provided by the http-acceleration mode, which makes the proxy server act as a sort of output buffer for the dynamic content. The mod_perl server sends the entire response to the proxy and is then free to deal with other requests. The proxy server is responsible for sending the response to the browser. So if the transfer is over a slow link, the mod_perl server is not waiting

around for the data to move.

Using numbers is always more convincing than just words. So we are going to show a simple example from the real world.

First let's explain the abbreviation used in the networking world. If someone claims to have a 56 kbps connection -- it means that the connection is of 56 kilo-bits per second (~56000 bits/sec). It's not 56 kilo-bytes per second, but 7 kilo-bytes per second, because 1 byte equals to 8 bits. So don't let the merchants fool you--your modem gives you 7 kilo-bytes per second connection at most and not 56 kilo-bytes per second as one might think.

Another convention used in computer literature is that if you see 10Kb it usually means 10 kilo-bits and 10KB is 10 kilo-bytes. So if you see upper case B it generally refers to bytes, and lower case b to bits (and K of course means kilo and equals to 1024 or to 1000 depending on the field it's used in). Remember that the latter convention is not followed everywhere, so use this knowledge with care. This document is following this convention.

So here is the real world example. Let's look at the typical scenario with a user connected to your site with 56Kbps modem. It means that the speed of the user's link is 56/8 = 7KBytes per sec. Let's assume an average generated HTML page to be of 42KB and an average mod_perl script that generates this response in 0.5 second. How many responses this script could produce during the time it took for the output to be delivered to user? A simple calculation reveals pretty scary numbers:

```
42KB / (0.5s * 7KB/s) = 12
```

12 other dynamic requests could be served at the same time, if we could put mod_perl to do only what it's best at: generating responses.

This very simple example shows us that we need only one twelfth the number of children running, which means that we will need only one twelfth of the memory (not quite true because some parts of the code are shared).

But you know that nowadays scripts often return pages which are blown up with JavaScript code and similar, which can make them 100kb size and the download time will be of the order of... (This calculation is left to you as an exercise :)

Moreover many users like to open many browser windows and do many things at once (download files and browse graphically *heavy* sites). So in the speed of 7KB/sec we have assumed before, may in reality be 5-10 times slower.

- This technique allows us to hide the details of the server's implementation. Users will never see ports in the URLs (more on that topic later). You can have a few boxes serving the requests, and only one serving as a front end, which spreads the jobs between the servers in a way that you can control. You can actually shut down a server, without the user even noticing, because the front end server will dispatch the jobs to other servers. (This is called *Load Balancing* and it's a pretty big issue which will take a book on its own to cover and therefore will not be discussed here. There is a plenty of information available at the Internet though. For more information see 'High-Availability Linux Project')

- For security reasons, using any httpd accelerator (or a proxy in httpd accelerator mode) is essential because you do not let your internal server get directly attacked by arbitrary packets from whomever. The httpd accelerator and internal server communicate in expected HTTP requests. This allows for only your public "bastion" accelerating www server to get hosed in a successful attack, while leaving your internal data safe.

  This is true if your server runs on your localhost (127.0.0.1) which makes it impossible to connect to you back end machine from the outside. But you don't need to connect from the outside anymore. You will see why when you proceed to this techniques' implementation notes.

The disadvantages are:

- Of course there are drawbacks. Luckily, these are not functionality drawbacks, but they are more administration hassle. You have another daemon to worry about, and while proxies are generally stable, you have to make sure to prepare proper startup and shutdown scripts, which are run at boot and reboot as appropriate. This is something that you do once and never come back to this issue again. Also, you might want to set up the crontab to run a watchdog script that will make sure that the proxying server is running and restart it if it detects a problem, reporting the problem to the administrator on the way.

- Proxy servers can be configured to be light or heavy. The administrator must decide what gives the highest performance for his application. A proxy server like *squid* is light in the sense of having only one process serving all requests. But it can consume a lot of memory when it loads objects into memory for faster service.

- If you use the default logging mechanism for all requests on the front- and back-end servers the requests that will be forwarded to the back-end server will be logged twice, which makes it tricky to merge the two logfiles, should you want to.

  One solution is to tell the heavy Apache not to bother logging requests that seem to come from the light Apache's machine. You might do this by installing a custom `PerlLogHandler` or just piping to *access_log* via `grep -v` (match all but this pattern) for the requests coming from the light Apache server. In this scenario, the *access_log* written by the light Apache is the file you should work with. But you need to look for any direct accesses to the heavy server in case the proxy server is sometimes bypassed, which can be eliminated if the server is listening only to the localhost (127.0.0.1).

  If you still decide to log proxied requests at the back-end server they will be useless since instead of real remote IP of the user, you will get always the same IP of the front-end server. Later in this Chapter on page XXX (mod_proxy_add_forward) we present a solution for this problem.

Have I succeeded in convincing you that you want a proxy server?

Of course if you are on a very fast local area network (LAN) (which means that all your users are connected from this LAN and not from the outside), then the big benefit of the proxy buffering the output and feeding a slow client is gone. You are probably better off sticking with a straight mod_perl server in this case.

12.6 Implementations of Proxy Servers

As of this writing, two proxy implementations are known to be widely used with mod_perl, the **squid** proxy server and **mod_proxy** which is a part of the Apache server. Let's compare them.

12.6.1 The Squid Server

The Advantages:

- Caching of static objects. These are served much faster, assuming that your cache size is big enough to keep the most frequently requested objects in the cache.
- Buffering of dynamic content, by taking the burden of returning the content generated by mod_perl servers to slow clients, thus freeing mod_perl servers from waiting for the slow clients to download the data. Freed servers immediately switch to serve other requests, thus your number of required servers goes down dramatically.
- Non-linear URL space / server setup. You can use Squid to play some tricks with the URL space and/or domain based virtual server support.

The Disadvantages:

- Proxying dynamic content is not going to help much if all the clients are on a fast local net. Also, by default squid only buffers in 16KB chunks so it would not allow mod_perl to complete immediately if the output is larger. (`READ_AHEAD_GAP` which is 16KB by default, can be enlarged in defines.h if your OS allows that).
- Speed. Squid is not very fast today when compared with the plain file based web servers available. Only if you are using a lot of dynamic features such as mod_perl or similar is there a reason to use Squid, and then only if the application and the server are designed with caching in mind.
- Memory usage. Squid uses quite a bit of memory. In fact, it caches a good part of its content in memory, to be able to serve it directly from RAM, a technique which is a lot quicker than serving from disk. However, as you already have your mod_perl server caching its code in RAM, you might not want another RAM-hogging beast taking up your precious memory (see the Squid FAQ for reference: http://www.squid-cache.org/Doc/FAQ/FAQ-8.html).
- HTTP protocol level. Squid is pretty much a `HTTP/1.0` server, which seriously limits the deployment of `HTTP/1.1` features, such as `Keep-Alive` requests.
- HTTP headers, dates and freshness. The squid server might give out stale pages, confusing downstream/client caches. (You update some documents on the site, but squid will still serve the old ones.)
- Stability. Compared to plain web servers, Squid is not the most stable.

The pros and cons presented above lead to the idea that you might want to use squid for its dynamic content buffering features, but only if your server serves mostly dynamic requests. So in this situation, when performance is the goal, it is better to have a plain Apache server serving static objects, and squid proxying the mod_perl enabled server only.

For implementation details, see the sections Running One Webserver and Squid in httpd Accelerator Mode and Running Two Webservers and Squid in httpd Accelerator Mode in the implementations chapter.

12.6.2 Apache's mod_proxy

I do not think the difference in speed between Apache's **mod_proxy** and **squid** is relevant for most sites, since the real value of what they do is buffering for slow client connections. However, squid runs as a single process and probably consumes fewer system resources.

The trade-off is that mod_rewrite is easy to use if you want to spread parts of the site across different back end servers, while mod_proxy knows how to fix up redirects containing the back-end server's idea of the location. With squid you can run a redirector process to proxy to more than one back end, but there is a problem in fixing redirects in a way that keeps the client's view of both server names and port numbers in all cases.

The difficult case is where you have DNS aliases that map to the same IP address and you want the redirect to port 80 and the server is on a different port and you want to keep the specific name the browser has already sent, so that it does not change in the client's **Location** window.

The Advantages:

- No additional server is needed. We keep the one plain plus one mod_perl enabled Apache servers. All you need is to enable `mod_proxy` in the `httpd_docs` server and add a few lines to `httpd.conf` file.

- The `ProxyPass` and `ProxyPassReverse` directives allow you to hide the internal redirects, so if `http://example.com/modperl/` is actually `http://localhost:81/modperl/`, it will be absolutely transparent to the user. `ProxyPass` redirects the request to the mod_perl server, and when it gets the response, `ProxyPassReverse` rewrites the URL back to the original one, e.g:

```
ProxyPass        /modperl/ http://localhost:81/modperl/
ProxyPassReverse /modperl/ http://localhost:81/modperl/
```

- It does mod_perl output buffering like squid does. See the Using mod_proxy notes for more details.

- It even does caching. You have to produce correct `Content-Length`, `Last-Modified` and `Expires` http headers for it to work. If some of your dynamic content does not change frequently, you can dramatically increase performance by caching it with mod_proxy.

- `ProxyPass` happens before the authentication phase, so you do not have to worry about authenticating twice.

- Apache is able to accelerate secure HTTP requests completely, while also doing accelerated HTTP. With Squid you have to use an external redirection program for that.
- The latest (Apache 1.3.6 and later) Apache proxy accelerated module is reported to be very stable.

For implementation see the "Using mod_proxy" section in the implementation chapter.

12.6.3 Closing Lingering Connections with Lingerd

Because of some technical complications in TCP/IP, at the end of each client connection, it is not enough for Apache to close the socket and forget about it; instead, it needs to spend about one second *lingering* on the client. (More details can be found at http://httpd.apache.org/docs/misc/fin_wait_2.html)

Lingerd is a daemon (service) designed to take over the job of properly closing network connections from an http server like Apache and immediately freeing it to handle a new connection.

`lingerd` can only do an effective job if HTTP `Keep-Alive`s are turned off; since `Keep-Alive`s are useful for images, the recommended setup is to have `lingerd` serving mod_perl enabled Apache and plain Apache for images and other static objects.

With a `lingerd` setup, you don't have the proxy, so the buffering chain we have presented before for the proxy setup is much shorter here:

FIGURE:

```
|   Apache     Kernel   |TCP/IP  'o'
| [mod_perl]=>[sendbuf] |======> /|\
|                       |        / \
```

Hence in this setup it becomes more important to have a big enough kernel send buffer.

With lingerd, a big enough kernel send buffer, and keep-alives off, the job of spoonfeeding the data to a slow client is done by the OS kernel in the background. As a result, `lingerd` makes it possible to serve the same load using considerably fewer Apache processes. This translates into a reduced load on the server. It can be used as an alternative to the proxy setups we have seen so far.

For more information about `lingerd` see: http://www.iagora.com/about/software/lingerd/

12.7 When One Machine is not Enough for RDBMS Database and mod_perl

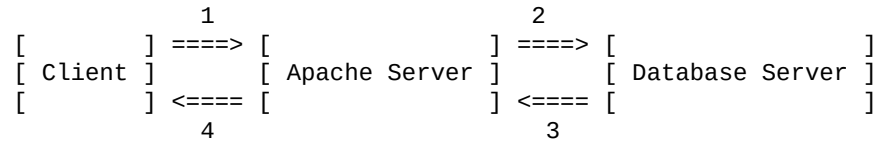
Imagine a scenario where you start your business as a small service providing web-site. After a while your business becomes very popular and at some point you understand that it has outgrown the capacity of your machine. Therefore you decide to upgrade your current machine with lots of memory, the cutting edge super expensive CPU and an ultra-fast hard disk. As a result the load goes back to normal but not for a long, as the demand for your services keeps on growing and just a little time after you've upgraded your machine, once again it cannot cope the load. Should you buy an even stronger and very expensive machine

or start looking for another solution? Let's explore the possible solution for this problem.

A typical web service consists of two main software components, the database server and the web server.

A typical user-server interaction consists of accepting the query parameters entered into an HTML form and submitted to the web server by a user, converting these parameters into a database query, sending it to the database server, accepting the results of the executed query, formatting them into a nice HTML page, and sending it to a user's Internet browser or another application that created the request (e.g. WAP).

This figure depicts the above description:



This schema is known as a 3-tier architecture in the computing world.

A 3-tier architecture means splitting up several processes of your computing solution between different machines.

- **Tier 1**

  The client, who will see the data on its screen and can give instructions to modify or process the data. In our case, an Internet browser.

- **Tier 2**

  The application server, which does the actual processing of the data and sends it back to the client. In our case, a mod_perl enabled Apache server.

- **Tier 3**

  The database server, which stores and retrieves all the data for the application server.

We are interested only in the second and the third tiers; we don't specify user machine requirements, since mod_perl is all about server side programming. The only thing the client should be able to do is to render the generated HTML from the response, which any simple browser will do. Of course I'm not talking about the case where you return some heavy Java applets, but that movie is screened in another theater.

12.7.1 Servers' Requirements

Let's first understand what kind of software the web and database servers are, what they need to run fast and what implications they have on the rest of the system software.

The three important machine components are the hard disk, the amount of RAM and the CPU type.

Typically the mod_perl server is mostly RAM hungry, while the SQL database server mostly needs a very fast hard-disk. Of course if your mod_perl process reads a lot from the disk (which is a quite infrequent phenomenon) you will need a fast disk too. And if your database server has to do a lot of sorting of big tables and do lots of big table joins, you will need a lot of RAM too.

If we would specify average "virtual" requirements for each machine, that's what we'd get:

An *"ideal"* mod_perl machine:

```
* HD:  low-end (no real IO, mostly logging)
* RAM: the more the better
* CPU: medium to high (according to needs)
```

An *"ideal"* database server machine:

```
* HD:  high-end
* RAM: large amounts   (big joins, sorting of many records)
       small amounts (otherwise)
* CPU: medium to high (according to needs)
```

12.7.2 The Problem

With the database and the httpd on the same machine, you have conflicting interests.

During peak loads, Apache will spawn more processes and use RAM that the database server might have been using, or that the kernel was using on its behalf in the form of cache. You will starve your database of resources at the time when it needs those resources the most.

Disk I/O contention is the biggest time issue. Adding another disk wouldn't cut I/O times because the database is the only one who does I/O - since mod_perl processes have all their code loaded in memory. (I'm talking about code that does pure perl and SQL processing) so it's clear that the DB is I/O and CPU bounded (RAM only if there are big joins to make) and mod_perl CPU and mostly RAM bounded.

The problem exists, but it doesn't mean that you cannot run the application and the web servers on the same machine. There is a very high degree of parallelism in modern PC architecture. The I/O hardware is helpful here. The machine can do many things while a SCSI subsystem is processing a command, or the network hardware is writing a buffer over the wire.

If a process is not runnable (that is, it is blocked waiting for I/O or similar), it is not using significant CPU time. The only CPU time that will be required to maintain a blocked process is the time it takes for the operating system's scheduler to look at the process, decide that it is still not runnable, and move on to the next process in the list. This is hardly any time at all. If there are two processes and one of them is blocked on I/O and the other is CPU bound, the blocked process is getting 0% CPU time, the runnable process is getting 99.9% CPU time, and the kernel scheduler is using the remainder.

12.7.3 The Solution

Adding another machine, which allows a set-up where both the database and the web servers run on their own dedicated machines.

12.7.3.1 Pros

- **Hardware Requirements**

  That allows you to scale two requirements independently.

  If your httpd processes are heavily weighted with respect to RAM consumption, you can easily add another machine to accommodate more httpd processes, without changing your database machine.

  If your database is CPU intensive, but your httpd doesn't need much CPU time, you can get low end machines for the httpd and a high end machine with a very fast CPU for the database server.

- **Scalability**

  Since your web server is not depending on the database server location any more, you can add more web servers hitting the same database server, using the existing infrastructure.

- **Database Security**

  Once you have multiple web server boxes the backend database becomes a single point of failure so it's a good idea to shield it from direct Internet access, something you couldn't do when you had both servers on the same machine.

12.7.3.2 Cons

- **Network latency**

  A database request from a webserver to a database server running on the same machine uses UNIX sockets, compared to the TCP/IP sockets used when the client submits the query from another machine. UNIX sockets are very fast since all the communications happens within the same box, eliminating network delays. TCP/IP sockets communication totally depends on the quality and the speed of the network the two machines are connected with.

  Basically, you can have almost the same client-server speed if you install a very fast and dedicated network between the two machines. It might impose a cost of additional NICs but it's probably insignificant compared to the speed up you gain.

  But even the normal network that you have would probably fit as well, because the networks delays are probably much smaller than the time it takes to execute the query. In contrast to the previous paragraph, you really want to test the added overhead, since the network can be quite slow especially at the peak hours.

How do you know what overhead is a significant one? All you have to measure is the average time spent in the web server and the database server. If any of the two numbers is at least 20 times bigger than the added overhead of the network you are all set.

To give you some numbers -- if your query takes about 20 milliseconds to process and only 1 millisecond to deliver the results, it's good. If the delivery takes about half of the time the processing takes you should start considering switching to a faster and/or dedicated network.

The consequences of a slow network can be quite bad. If the network is slow mod_perl processes remain open waiting for data from the database server and eat even more RAM as new child processes pop up to handle new requests. So the overall machine performance can be worse than it was originally when you had just a single machine for both servers.

12.7.4 Three Machines Model

Since we are talking about using a dedicated machine for each server, you might consider adding a third machine to do the proxy work; this will make your setup even more flexible since it will enable you to proxy-pass all request to not just one mod_perl running box, but to many of them. It will enable you to do load balancing if and when you need it.

Generally the proxy machine can be very light when it serves just a little traffic and mainly proxy-passes to the mod_perl processes. Of course you can use this machine to serve the static content and then the hardware requirement will depend on the number of objects you will have to serve and the rate at which they are requested.

12.8 Running More than One mod_perl Server on the Same Machine.

Let's assume that you have two different sets of code which have little or nothing in common--different Perl modules, no code sharing. Typical numbers can be four megabytes of unshared and four megabytes of shared memory for each code set, plus three megabytes of shared basic mod_perl stuff. Which makes each process 17MB in size when the two code sets are loaded. (3MB (server core shared) + 4MB (shared 1st code set) + 4MB (unshared 1st code set) + 4MB (shared 2nd code set) + 4MB (unshared 2nd code set). Under this scenario:

```
Shared_RAM_per_Child :  11MB
Max_Process_Size     :  17MB
Total_RAM            : 251MB
```

We assume that four megabytes is the size of each code sets unshared memory. This is a pretty typical size of unshared memory, especially when connecting to databases, as the database connections cannot be shared. Databases like Oracle can take even more RAM per connection on top of this.

Let's assume that we have 251 megabytes of RAM dedicated to the webserver.

According to the equation developed in the section: "Choosing MaxClients":

```
                    Total_RAM - Shared_RAM_per_Child
MaxClients = ------------------------------------
              Max_Process_Size - Shared_RAM_per_Child

MaxClients = (251 - 11)/(17-11) = 40
```

We see that we can run 40 processes, using the given memory and the two code sets in the same server.

Now consider this practical decision. Since we have recognized that the code sets are very distinct in nature and there is no significant memory sharing in place, the wise thing to do is to split the two code sets between two mod_perl servers (a single mod_perl server actually is a set of the parent process and a number of the child processes). So instead of running everything on one server, now we move the second code set onto another mod_perl server. At this point we are talking about a single machine.

Let's look at the figures again. After the split we will have 20 servers of eleven megabytes (4MB unshared + 7mb shared) and another 20 more of the same kind.

How much memory do we need now? From the above equation we derive:

```
Total_RAM = MaxClients * (Max_Process_Size - Shared_RAM_per_Child)
            + Shared_RAM_per_Child
```

And using the numbers (the total of 40 servers):

```
Total_RAM = 2 * (20*(11-7)+7) = 174
```

A total of 174MB of memory required. But, hey, we have 251MB of memory. We've got 77MB of memory freed up. If we recalculate again the `MaxClients` we will see that we can run almost 60 servers:

```
MaxClients = (251 - 7*2)/(11-7) = 59
```

So we can run about 19 more servers using the same memory size. Almost 30 servers for each code set instead of 20 originally. We have enlarged the servers pool by half without changing the machine's hardware.

Moreover this new setup allows us to fine tune the two code sets, since in reality the smaller in size code base might have a higher hit rate, so we can benefit even more.

Let's assume that based on the usage statistics we know that the first code set is called in 70% of requests and the other 30% are used by the second set. Now we assume that the first code set requires only 5MB of RAM (3MB shared plus 2MB unshared) over the basic mod_perl server size, and the second set needs 11MBytes (7MB shared and 4MB unshared).

Lets compare this new requirement with our original 50:50 setup (here we have assigned the same number of clients for each code set).

So now the first mod_perl server running the first code set will have all its processes using 8MB (3MB (server shared) + 3MB (code shared) + 2MB (code unshared), and the second 14MB (3+7+4). Given that we have a 70:30 hits relation and that we have 251MB of available memory, we have to solve these two equations:

```
X/Y = 7/3

X*(8-6) + 6 + Y*(14-10) + 10 = 251
```

where X is the total number of the processes the first code set can use and Y the second. The first equation reflect the 70:30 hits relation, and the second uses the equation for the total memory requirements for the given number of servers and the shared and unshared memory sizes.

When we solve these equations, we find that X equals 63 and Y equals 27. So we have a total of 90 servers -- two and a half times the number of servers running compared to the original setup using the same memory size.

The hits rate optimized solution and the fact that the code sets can be different in their memory requirements, allowed us to run 30 more servers in total and gave us 33 more servers (63 versus 30) for the most wanted code base, relative to the simple 50:50 split as in the first example.

Of course if you identify more than two distinct sets of code based on your hit rate statistics, more complicated solutions may be required. You could make even more splits and run three or more mod_perl servers.

Remember that having too many running processes doesn't necessarily mean better performance because all of them will contend for CPU time slices. The more processes that are running the less CPU time each gets and the slower overall performance will be. Therefore after hitting a certain load you might want to start spreading servers over different machines.

In addition to the obvious memory saving you gain the power to troubleshoot problems that occur more easily when you have different components running on different servers. It's quite possible that a small change in the server configuration to fix or improve something for one code set, might completely break the second code set. For example if you upgrade the first code set and it requires an update of some modules that both code bases rely on. But there is a chance that the second code set won't work with a new version of a module it was relying on.

12.9 SSL functionality and a mod_perl Server

If you need SSL functionality, you can get it by adding the mod_ssl or equivalent Apache_ssl to the light front-end server (httpd_docs) or the heavy back-end mod_perl server (httpd_perl). (The configuration and installation instructions are located here.)

The question is: Is it a good idea to add mod_ssl into the back-end mod_perl enabled server? Given that your internal network is secured, or if both the front and back end servers are running on the same machine and you can ensure a safe communication between the processes, there is no need for an encrypted traffic between them.

If this is the situation you don't have to put mod_ssl into the already too heavy mod_perl server. You will have the external traffic encrypted by the front-end server, which will proxy-pass the unencrypted request and response data internally.

Another important point is if you put the mod_ssl on the back-end, you have to tunnel back your images to it (i.e. have the back-end serve the images) defeating the whole purpose of having the front-end lightweight server.

You cannot serve a secure page which includes non-secured information. If you fetch an html page over SSL and have an `<IMG>` tag that fetches the image from the non-secure server, the image is shown broken. This is true for any other non-secured objects as well. Of course if the generated response doesn't include any embedded objects, like images -- this is not a problem.

Choosing the front-end machine to have an SSL functionality also simplifies configuration of mod_perl by eliminating VirtualHost duplication for SSL. mod_perl configuration files can be plenty difficult without the mod_ssl overhead.

Also assuming that you have front-end machines under-worked anyway, especially if you run a high-volume web service deploying a cluster of machines to serve requests, you save some CPU as it's known that SSL connections are about 100 times more CPU intensive than non-SSL connections.

Of course caching session keys so you don't have to set up a new symmetric key for every single connection, improves the situation. If you use the shared memory session caching mechanism that mod_ssl supports, then the overhead is actually rather small except for the initial connection.

But then on the other hand, why even bother to run a full scale mod_ssl in front? You might as well just choose a small tunnel/port forwarding application like Stunnel or one of the many other mentioned at http://www.openssl.org/related/apps.html .

Of course if you do heavy SSL processing ideally you should really be offloading it to a dedicated cryptography server. But this advice can be misleading based on the current status of the crypto hardware. If you use hardware you get extra speed now, but you're locked into a proprietary solution; in 6 months/one year software will have caught up with whatever hardware you're using and because software is easier to adapt you'll have more freedom to change what software you're using and more control of things. So the choice is in your hand.

12.10 Maintainers

Maintainer is the person(s) you should contact with updates, corrections and patches.

- Stas Bekman [http://stason.org/]

12.11 Authors

- Stas Bekman [http://stason.org/]

Only the major authors are listed above. For contributors see the Changes file.

13 Real World Scenarios

13.1 Description

This chapter provides a step-by-step installation guide for the various setups discussed in Choosing the Right Strategy.

13.2 Assumptions

I will assume for this section that you are familiar with plain (not mod_perl enabled) Apache, its compilation and configuration. In all configuration and code examples I will use *localhost* or *www.example.com* as a hostname. For the testing on a local machine, *localhost* would be just fine. If you are using the real name of your machine make sure to replace *www.example.com* with the name of your machine.

13.3 Standalone mod_perl Enabled Apache Server

13.3.1 Installation in 10 lines

The Installation is very simple. This example shows installation on the Linux operating system.

```
% cd /usr/src
% lwp-download http://www.apache.org/dist/apache_x.x.x.tar.gz
% lwp-download http://apache.org/dist/perl/mod_perl-x.xx.tar.gz
% tar xzvf apache_x.x.x.tar.gz
% tar xzvf mod_perl-x.xx.tar.gz
% cd mod_perl-x.xx
% perl Makefile.PL APACHE_SRC=../apache_x.x.x/src \
  DO_HTTPD=1 USE_APACI=1 EVERYTHING=1
% make && make test && make install
% cd ../apache_x.x.x
% make install
```

That’s all!

Notes: Replace x.xx and x.x.x with the real version numbers of mod_perl and Apache respectively. The `z` flag tells Gnu `tar` to uncompress the archive as well as extract the files. You might need superuser permissions to do the make install steps.

13.3.2 Installation in 10 paragraphs

If you have the `lwp-download` utility installed, you can use it to download the sources of both packages:

```
% lwp-download http://www.apache.org/dist/apache_x.x.x.tar.gz
% lwp-download http://apache.org/dist/perl/mod_perl-x.xx.tar.gz
```

`lwp-download` is a part of the LWP module (from the `libwww` package), and you will need to have it installed in order for mod_perl’s `make test` step to pass.

Extract both sources. Usually I open all the sources in */usr/src/*, but your mileage may vary. So move the sources and `chdir` to the directory that you want to put the sources in. If you have a non-GNU `tar` utility it will be unable to decompress so you will have to unpack in two steps: first uncompress the packages with:

```
gzip -d apache_x.x.x.tar.gz
gzip -d mod_perl-x.xx.tar.gz
```

then un-tar them with:

```
tar xvf apache_x.x.x.tar
tar xvf mod_perl-x.xx.tar
```

You can probably use gunzip instead of gzip -d if you prefer.

```
% cd /usr/src
% tar xzvf apache_x.x.x.tar.gz
% tar xzvf mod_perl-x.xx.tar.gz
```

`chdir` to the mod_perl source directory:

```
% cd mod_perl-x.xx
```

Now build the Makefile. For your first installation and most basic work the parameters in the example below are the only ones you will need. `APACHE_SRC` tells the Makefile.PL where to find the Apache *src* directory. If you have followed my suggestion and have extracted both sources under the directory */usr/src*, then issue the command:

```
% perl Makefile.PL APACHE_SRC=../apache_x.x.x/src \
  DO_HTTPD=1 USE_APACI=1 EVERYTHING=1
```

There are many additional optional parameters. You can find some of them later in this section and in the Server Configuration section.

While running `perl Makefile.PL ...` the process will check for prerequisites and tell you if something is missing. If you are missing some of the perl packages or other software, you will have to install them before you proceed.

Next `make` the project. The command `make` builds the mod_perl extension and also calls `make` in the Apache source directory to build `httpd`. Then we run the *test* suite, and finally *install* the mod_perl modules in their proper places.

```
% make && make test && make install
```

Note that if `make` fails, neither `make test` nor `make install` will be executed. If `make test` fails, `make install` will be not executed.

Now change to the Apache source directory and run `make install`. This will install Apache's headers, default configuration files, build the Apache directory tree and put `httpd` in it.

```
% cd ../apache_x.x.x
% make install
```

When you execute the above command, the Apache installation process will tell you how to start a freshly built webserver (you need to know the path of `apachectl`, more about that later) and where to find the configuration files. Write down both, since you will need this information very soon. On my machine the two important paths are:

```
/usr/local/apache/bin/apachectl
/usr/local/apache/conf/httpd.conf
```

Now the build and installation processes are complete.

13.3.3 Configuration

First, a simple configuration. Configure Apache as you usually would (set `Port`, `User`, `Group`, `ErrorLog`, other file paths etc).

Start the server and make sure it works, then shut it down. The `apachectl` utility can be used to start and stop the server:

```
% /usr/local/apache/bin/apachectl start
% /usr/local/apache/bin/apachectl stop
```

Now we will configure Apache to run perl CGI scripts under the `Apache::Registry` handler.

You can put configuration directives in a separate file and tell *httpd.conf* to include it, but for now we will simply add them to the main configuration file. We will add the mod_perl configuration directives to the end of *httpd.conf*. In fact you can place them anywhere in the file, but they are easier to find at the end.

For the moment we will assume that you will put all the scripts which you want to be executed by the mod_perl enabled server under the directory */home/httpd/perl*. We will alias this directory to the URI */perl*

Add the following configuration directives to *httpd.conf*:

```
Alias /perl/ /home/httpd/perl/

PerlModule Apache::Registry
<Location /perl>
  SetHandler perl-script
  PerlHandler Apache::Registry
  Options ExecCGI
  PerlSendHeader On
  allow from all
</Location>
```

Now create a four-line test script in */home/httpd/perl/*:

```
test.pl
-------
#!/usr/bin/perl -w
use strict;
print "Content-type: text/html\r\n\r\n";
print "It worked!!!\n";
```

Note that the server is probably running as a user with a restricted set of privileges, perhaps as user `nobody` or `www`. Look for the `User` directive in *httpd.conf* to find the userid of the server.

Make sure that you have read and execute permissions for *test.pl*.

```
% chmod u+rx /home/httpd/perl/test.pl
```

Test that the script works from the command line, by executing it:

```
% /home/httpd/perl/test.pl
```

You should see:

```
Content-type: text/html

It worked!!!
```

Assuming that the server's userid is `nobody`, make the script owned by this user. We already made it executable and readable by user.

```
% chown nobody /home/httpd/perl/test.pl
```

Now it is time to test that mod_perl enabled Apache can execute the script.

Start the server ('`apachectl start`'). Check in *logs/error_log* to see that the server has indeed started--verify the correct date and time of the log entry.

To get Apache to execute the script we simply fetch its URI. Assuming that your *httpd.conf* has been configured with the directive `Port 80`, start your favorite browser and fetch the following URI:

```
http://www.example.com/perl/test.pl
```

If you have the loop-back device (127.0.0.1) configured, you can use the URI:

```
http://localhost/perl/test.pl
```

In either case, you should see:

```
It worked!!!
```

If your server is listening on a port other than 80, for example 8000, then fetch the URI:

```
http://www.example.com:8000/perl/test.pl
```

or whatever is appropriate.

If something went wrong, go through the installation process again, and make sure you didn't make a mistake. If that doesn't help, read the `INSTALL` pod document (`perlpod INSTALL`) in the mod_perl distribution directory.

Now that your mod_perl server is working, copy some of your Perl CGI scripts into the directory */home/httpd/perl/* or below it.

If your programming techniques are good, chances are that your scripts will work with no modifications at all. With the mod_perl enabled server you will see them working very much faster.

If your programming techniques are sloppy, some of your scripts will not work and they may exhibit strange behaviour. Depending on the degree of sloppiness they may need anything from minor tweaking to a major rewrite to make them work properly. (See Sometimes My Script Works, Sometimes It Does Not)

The above setup is very basic, but as with Perl, you can start to benefit from mod_perl from the very first moment you try it. As you become more familiar with mod_perl you will want to start writing Apache handlers and make more use of its power.

13.4 One Plain and One mod_perl enabled Apache Servers

Since we are going to run two Apache servers we will need two complete (and different) sets of configuration, log and other files. In this scenario we'll use a dedicated root directory for each server, which is a personal choice. You can choose to have both servers living under the same roof, but it might lead to a mess, since it requires a slightly more complicated configuration. This decision might be nice since you will be able to share some directories like *include* (which contains Apache headers), but in fact this can become a problem later, when you decide to upgrade one server but not the other. You will have to solve this problem then, so why not to avoid it in first place.

From now on we will refer to these two servers as **httpd_docs** (plain Apache) and **httpd_perl** (Apache/mod_perl). We will use */usr/local* as our *root* directory.

First let's prepare the sources. We will assume that all the sources go into the */usr/src* directory. Since you will probably want to tune each copy of Apache separately, it is better to use two separate copies of the Apache source for this configuration. For example you might want only the httpd_docs server to be built with the mod_rewrite module.

Having two independent source trees will prove helpful unless you use dynamically shared objects (`DSO`) which is covered later in this chapter.

Make two subdirectories:

```
% mkdir /usr/src/httpd_docs
% mkdir /usr/src/httpd_perl
```

Next put a set of the Apache sources into the */usr/src/httpd_docs* directory (replace the directory */tmp* with the path to the downloaded file and `x.x.x` with the version of Apache that you have downloaded):

```
% cd /usr/src/httpd_docs
% gzip -dc /tmp/apache_x.x.x.tar.gz | tar xvf -
```

or if you have GNU tar:

```
% tar xvzf /tmp/apache_x.x.x.tar.gz
```

Just to check we have extracted in the right way:

```
% ls -l
drwxr-xr-x  8 stas  stas 2048 Apr 29 17:38 apache_x.x.x/
```

Now prepare the httpd_perl server sources:

```
% cd /usr/src/httpd_perl
% gzip -dc /tmp/apache_x.x.x.tar.gz | tar xvf -
% gzip -dc /tmp/modperl-x.xx.tar.gz | tar xvf -

% ls -l
drwxr-xr-x  8 stas  stas 2048 Apr 29 17:38 apache_x.x.x/
drwxr-xr-x  8 stas  stas 2048 Apr 29 17:38 modperl-x.xx/
```

We are going to use a default Apache directory layout, and place each server directories under its dedicated directory. The two directories are as you have already guessed:

```
/usr/local/httpd_perl/
/usr/local/httpd_docs/
```

The next step is to configure and compile the sources: Below are the procedures to compile both servers, using the directory layout I have just suggested.

13.4.1 Configuration and Compilation of the Sources.

As usual we use *x.x.x* instead of real version numbers so this document will never become obsolete. But the most important thing -- it's not misleading. It's quite possible that since the moment this document was written a new version has come out and you will be not aware of this fact if you will not check for it.

13.4.1.1 Building the httpd_docs Server

- **Sources Configuration:**

  ```
  % cd /usr/src/httpd_docs/apache_x.x.x
  % make clean
  %  ./configure --prefix=/usr/local/httpd_docs \
     --enable-module=rewrite --enable-module=proxy
  ```

  We need the *mod_rewrite* and *mod_proxy* modules as we will see later, so we tell *./configure* to build them in.

You might want to add `--layout` to see the resulting directories' layout without actually running the configuration process.

- **Source Compilation and Installation**

```
% make
% make install
```

Rename `httpd` to `http_docs`:

```
% mv /usr/local/httpd_docs/bin/httpd \
     /usr/local/httpd_docs/bin/httpd_docs
```

Now modify the **apachectl** utility to point to the renamed httpd via your favorite text editor or by using perl:

```
% perl -pi -e 's|bin/httpd|bin/httpd_docs|' \
  /usr/local/httpd_docs/bin/apachectl
```

Another approach would be to use the *--target* option while configuring the source, which makes the last two commands unnecessary.

```
% ./configure --prefix=/usr/local/httpd_docs \
   --target=httpd_docs \
   --enable-module=rewrite --enable-module=proxy
% make && make install
```

Since we told *./configure* that we want the executable to be called `httpd_docs` (via `--target=httpd_docs`) -- it performs all the naming adjustment for us.

The only thing that you might find unusual, is that *apachectl* will be now called *httpd_docsctl* and the configuration file *httpd.conf* now will be called *httpd_docs.conf*.

We will leave the decision making about the preferred configuration and installation way to the reader. In the rest of the guide we will continue using the regular names resulted from using the standard configuration and the manual executable name adjustment as described at the beginning of this section .

13.4.1.2 Building the httpd_perl Server

Now we proceed with the sources configuration and installation of the *httpd_perl* server. First make sure the sources are clean:

```
% cd /usr/src/httpd_perl/apache_x.x.x
% make clean
% cd /usr/src/httpd_perl/mod_perl-x.xx
% make clean
```

It is important to **make clean** since some of the versions are not binary compatible (e.g apache 1.3.3 vs 1.3.4) so any "third-party" C modules need to be re-compiled against the latest header files.

```
% cd /usr/src/httpd_perl/mod_perl-x.xx

% /usr/bin/perl Makefile.PL \
      APACHE_SRC=../apache_x.x.x/src \
  DO_HTTPD=1 USE_APACI=1 EVERYTHING=1 \
      APACHE_PREFIX=/usr/local/httpd_perl \
  APACI_ARGS='--prefix=/usr/local/httpd_perl'
```

If you need to pass any other configuration options to Apache's `configure`, add them after the *--prefix* option. e.g:

```
  APACI_ARGS='--prefix=/usr/local/httpd_perl \
                 --enable-module=status'
```

Notice that **all** `APACI_ARGS` (above) must be passed as one long line if you work with `t?csh`!!! However with `(ba)?sh` it works correctly the way it is shown above, breaking the long lines with '\'. As of tcsh version 6.08.0, when it passes the `APACI_ARGS` arguments to `configure` it does not alter the newlines, but it strips the backslashes, thus breaking the configuration process.

Notice that just like in httpd_docs configuration you can use `--target=httpd_perl` instead of specifying each directory separately. Note that this option has to be the very last argument in `APACI_ARGS`, otherwise 'make test' tries to run `'httpd_perl'`, which fails.

[META: It's very important to use the same compiler you build the perl with. See the section 'What Compiler Should Be Used to Build mod_perl' for more information.

[META: --- Hmm, what's the option that overrides the compiler when building Apache from mod_perl. Check also whether mod_perl supplies the right compiler (the one used for building itself) -- if it does there is no need for the above note.]

Now, build, test and install the `httpd_perl`.

```
% make && make test && make install
```

Upon installation Apache puts a stripped version of `httpd` at */usr/local/httpd_perl/bin/httpd*. The original version which includes debugging symbols (if you need to run a debugger on this executable) is located at */usr/src/httpd_perl/apache_x.x.x/src/httpd*.

You may have noticed that we did not run `make install` in the Apache source directory. When `USE_APACI` is enabled, `APACHE_PREFIX` will specify the `--prefix` option for Apache's `configure` utility, which gives the installation path for Apache. When this option is used, mod_perl's `make install` will also `make install` for Apache, installing the httpd binary, the support tools, and the configuration, log and document trees. If this option is not used you will have to explicitly run make install in the Apache source directory.

If `make test` fails, look into */usr/src/httpd_perl/mod_perl-x.xx/t/logs* and read the error_log file. Also see make test fails.

While doing `perl Makefile.PL ...` mod_perl might complain by warning you about a missing library `libgdbm`. This is a crucial warning. See Missing or Misconfigured libgdbm.so for more info.

Now rename `httpd` to `httpd_perl`:

```
% mv /usr/local/httpd_perl/bin/httpd \
  /usr/local/httpd_perl/bin/httpd_perl
```

Update the *apachectl* utility to drive the renamed httpd:

```
% perl -p -i -e 's|bin/httpd|bin/httpd_perl|' \
  /usr/local/httpd_perl/bin/apachectl
```

13.4.2 Configuration of the servers

Now when we have completed the building process, the last stage before running the servers is to configure them.

13.4.2.1 Basic httpd_docs Server Configuration

Configuring of the `httpd_docs` server is a very easy task. Starting from version 1.3.4 of Apache, there is only one file to edit. Open */usr/local/httpd_docs/conf/httpd.conf* in your favorite text editor and configure it as you usually would, except make sure that you configure the log file directory (*/usr/local/httpd_docs/logs* and so on) and the other paths according to the layout you have decided to use.

Start the server with:

```
/usr/local/httpd_docs/bin/apachectl start
```

13.4.2.2 Basic httpd_perl Server Configuration

Edit the */usr/local/httpd_perl/conf/httpd.conf*. As with the `httpd_docs` server configuration, make sure that `ErrorLog` and other file location directives are set to point to the right places, according to the chosen directory layout.

The first thing to do is to set a `Port` directive - it should be different from that used by the plain Apache server (`Port 80`) since we cannot bind two servers to the same port number on the same machine. Here we will use `8080`. Some developers use port `81`, but you can bind to ports below 1024 only if the server has root permissions. If you are running on a multiuser machine, there is a chance that someone already uses that port, or will start using it in the future, which could cause problems. If you are the only user on your machine, basically you can pick any unused port number. Many organizations use firewalls which may block some of the ports, so port number choice can be a controversial topic. From my experience the most popular port numbers are: `80`, `81`, `8000` and `8080`. Personally, I prefer the port `8080`. Of course with the two server scenario you can hide the nonstandard port number from firewalls and users, by using either mod_proxy's `ProxyPass` directive or a proxy server like Squid.

For more details see Publishing Port Numbers other than 80, Running One Webserver and Squid in httpd Accelerator Mode, Running Two Webservers and Squid in httpd Accelerator Mode and Using mod_proxy.

Now we proceed to the mod_perl specific directives. It will be a good idea to add them all at the end of `httpd.conf`, since you are going to fiddle with them a lot in the early stages.

First, you need to specify the location where all mod_perl scripts will be located.

Add the following configuration directive:

```
  # mod_perl scripts will be called from
Alias /perl/ /usr/local/httpd_perl/perl/
```

From now on, all requests for URIs starting with */perl* will be executed under mod_perl and will be mapped to the files in */usr/local/httpd_perl/perl/*.

Now we configure the */perl* location.

```
PerlModule Apache::Registry

<Location /perl>
  #AllowOverride None
  SetHandler perl-script
  PerlHandler Apache::Registry
  Options ExecCGI
  allow from all
  PerlSendHeader On
</Location>
```

This configuration causes any script that is called with a path prefixed with */perl* to be executed under the `Apache::Registry` module and as a CGI (hence the `ExecCGI`--if you omit this option the script will be printed to the user's browser as plain text or will possibly trigger a '**Save-As**' window). The `Apache::Registry` module lets you run your (carefully written) Perl CGI scripts virtually unchanged under mod_perl. The `PerlModule` directive is the equivalent of Perl's require(). We load the `Apache::Registry` module before we use it by giving the `PerlHandler Apache::Registry` directive.

`PerlSendHeader On` tells the server to send an HTTP header to the browser on every script invocation. You will want to turn this off for nph (non-parsed-headers) scripts.

This is only a very basic configuration. The Server Configuration section covers the rest of the details.

Now start the server with:

```
/usr/local/httpd_perl/bin/apachectl start
```

13.5 Running Two webservers and Squid in httpd Accelerator Mode

While I have detailed the mod_perl server installation, you are on your own with installing the Squid server (See Getting Helped for more details). I run Linux, so I downloaded the RPM package, installed it, configured the */etc/squid/squid.conf*, fired off the server and all was set.

Basically once you have Squid installed, you just need to modify the default `squid.conf` as I will explain below, then you are ready to run it.

The configuration that I'm going to present works with Squid server version 2.3.STABLE2. It's possible that some directives will change in future versions.

First, let's take a look at what we have already running and what we want from squid.

Previously we have had the `httpd_docs` and `httpd_perl` servers listening on ports 80 and 8080. Now we want squid to listen on port 80, to forward requests for static objects (plain HTML pages, images and so on) to the port which the httpd_docs server listens to, and dynamic requests to httpd_perl's port. And of course collecting the generated responses, which will be delivered to the client by Squid. As mentioned before this mode is known as *httpd-accelerator* mode in proxy dialect.

Therefore we have to reconfigure the httpd_docs server to listen to port 81 instead, since port 80 will be taken by Squid. Remember that in our scenario both copies of Apache will reside on the same machine as Squid.

A proxy server makes all the magic behind it transparent to users. Both Apache servers return the data to Squid (unless it was already cached by Squid). The client never sees the other ports and never knows that there might be more than one server running. Do not confuse this scenario with `mod_rewrite`, where a server redirects the request somewhere according to the rewrite rules and forgets all about it. (i.e. works as a one way dispatcher, which dispatches the jobs but is not responsible for.)

Squid can be used as a straightforward proxy server. ISPs and other companies generally use it to cut down the incoming traffic by caching the most popular requests. However we want to run it in `httpd accelerator mode`. Two directives (`httpd_accel_host` and `httpd_accel_port`) enable this mode. We will see more details shortly.

If you are currently using Squid in the regular proxy mode, you can extend its functionality by running both modes concurrently. To accomplish this, you can extend the existing Squid configuration with **httpd accelerator mode**'s related directives or you can just create one from scratch.

Let's go through the changes we should make to the default configuration file. Since the file with default settings (*/etc/squid/squid.conf*) is huge (about 60KB) and we will not alter 95% of its default settings, my suggestion is to write a new one including only the modified directives.

We want to enable the redirect feature, to be able to serve requests by more than one server (in our case we have two: the httpd_docs and httpd_perl servers). So we specify `httpd_accel_host` as virtual. This assumes that your server has multiple interfaces - Squid will bind to all of them.

```
httpd_accel_host virtual
```

Then we define the default port the requests will be sent to, unless redirected. We assume that most requests will be for static documents (also it's easier to define redirect rules for the mod_perl server because of the URI that starts with *perl* or similar). We have our httpd_docs listening on port 81:

```
httpd_accel_port 81
```

And as described before, squid listens to port 80.

```
http_port 80
```

We do not use `icp` (`icp` is used for cache sharing between neighboring machines, which is more relevant in the proxy mode).

```
icp_port 0
```

`hierarchy_stoplist` defines a list of words which, if found in a URL, causes the object to be handled directly by the cache. Since we told Squid in the previous directive that we aren't going to share the cache between neighboring machines this directive is irrelevant. In case that you do use this feature, make sure to set this directive to something like:

```
hierarchy_stoplist /cgi-bin /perl
```

where the */cgi-bin* and */perl* are aliases for the locations which handle the dynamic requests.

Now we tell Squid not to cache dynamically generated pages.

```
acl QUERY urlpath_regex /cgi-bin /perl
no_cache deny QUERY
```

Please note that the last two directives are controversial ones. If you want your scripts to be more compliant with the HTTP standards, according to the HTTP specification the headers of your scripts should carry the *Caching Directives*: `Last-Modified` and `Expires`.

What are they for? If you set the headers correctly, there is no need to tell the Squid accelerator **NOT** to try to cache anything. Squid will not bother your mod_perl servers a second time if a request is (a) cacheable and (b) still in the cache. Many mod_perl applications will produce identical results on identical requests if not much time has elapsed between the requests. So your Squid might have a hit ratio of 50%, which means that the mod_perl servers will have only half as much work to do as they did before you installed Squid (or mod_proxy).

Even if you insert a user-ID and date in your page, caching can save resources when you set the expiration time to 1 second. A user might double click where a single click would do, thus sending two requests in parallel. Squid could serve the second request.

But this is only possible if you set the headers correctly. Refer to the chapter Correct Headers - A quick guide for mod_perl users to learn more about generating the proper caching headers under mod_perl. In case where only the scripts under */perl/caching-unfriendly* are not *caching friendly* fix the above setting to be:

```
acl QUERY urlpath_regex /cgi-bin /perl/caching-unfriendly
no_cache deny QUERY
```

But if you are lazy, or just have too many things to deal with, you can leave the above directives the way we described. Just keep in mind that one day you will want to reread this section and the headers generation tutorial to squeeze even more power from your servers without investing money in more memory and better hardware.

While testing you might want to enable the debugging options and watch the log files in the directory */var/log/squid/*. But make sure to turn debugging off in your production server. Below we show it commented out, which makes it disabled, since it's disabled by default. Debug option 28 enables the debugging of the access control routes, for other debug codes see the documentation embedded in the default configuration file that comes with squid.

```
# debug_options 28
```

We need to provide a way for Squid to dispatch requests to the correct servers. Static object requests should be redirected to httpd_docs unless they are already cached, while requests for dynamic documents should go to the httpd_perl server. The configuration below tells Squid to fire off 10 redirect daemons at the specified path of the redirect daemon and (as suggested by Squid's documentation) disables rewriting of any `Host:` headers in redirected requests. The redirection daemon script is listed below.

```
redirect_program /usr/lib/squid/redirect.pl
redirect_children 10
redirect_rewrites_host_header off
```

The maximum allowed request size is in kilobytes, which is mainly useful during `PUT` and `POST` requests. A user who attempts to send a request with a body larger than this limit receives an "Invalid Request" error message. If you set this parameter to a zero, there will be no limit imposed. If you are using `POST` to upload files, then set this to the largest file's size plus a few extra KB.

```
request_body_max_size 1000 KB
```

Then we have access permissions, which we will not explain. You might want to read the documentation, so as to avoid any security problems.

```
acl all src 0.0.0.0/0.0.0.0
acl manager proto cache_object
acl localhost src 127.0.0.1/255.255.255.255
acl myserver src 127.0.0.1/255.255.255.255
acl SSL_ports port 443 563
acl Safe_ports port 80 81 8080 81 443 563
acl CONNECT method CONNECT

http_access allow manager localhost
http_access allow manager myserver
http_access deny manager
http_access deny !Safe_ports
http_access deny CONNECT !SSL_ports
# http_access allow all
```

Since Squid should be run as a non-root user, you need these if you are invoking the Squid as root. The user *squid* is created when the Squid server is installed.

```
cache_effective_user squid
cache_effective_group squid
```

Now configure a memory size to be used for caching. The Squid documentation warns that the actual size of Squid can grow to be three times larger than the value you set.

```
cache_mem 20 MB
```

We want to keep pools of allocated (but unused) memory available for future use if we have the memory available of course. Otherwise turn it off.

```
memory_pools on
```

Now tighten the runtime permissions of the cache manager CGI script (`cachemgr.cgi`, which comes bundled with squid) on your production server.

```
cachemgr_passwd disable shutdown
```

If you are not using this script to manage the Squid server from remote, you should disable it:

```
cachemgr_passwd disable all
```

Now the redirection daemon script (you should put it at the location you have specified in the `redirect_program` parameter in the config file above, and make it executable by the webserver of course):

```
#!/usr/local/bin/perl -p
BEGIN{ $|=1 }
s|www.example.com(?::81)?/perl/|www.example.com:8080/perl/|o ;
```

Here is what the regular expression from above does; it matches all the URIs that include either the string *www.example.com/perl/* or the string *www.example.com:81/perl/* and replaces either of these strings with *www.example.com:8080/perl*. No matter whether the regular expression worked or not, the `$_` variable is automagically printed.

We can write the above code as the following code as well:

```
#!/usr/local/bin/perl

$|=1;

while (<>) {
    # redirect to mod_perl server (httpd_perl)
  print($_), next
    if s|www.example.com(:81)?/perl/|www.example.com:8080/perl/|o;

    # send it unchanged to plain apache server (http_docs)
  print;
}
```

The above redirector can be more complex of course, but you know Perl, right?

A few notes regarding the redirector script:

You must disable buffering. `$|=1;` does the job. If you do not disable buffering, `STDOUT` will be flushed only when its buffer becomes full--and its default size is about 4096 characters. So if you have an average URL of 70 chars, only after about 59 (4096/70) requests will the buffer be flushed, and the requests will finally reach the server. Your users will not wait that long, unless you have hundreds requests per second and then the buffer will be flushed very frequently because it'll get full very fast.

If you think that this is a very ineffective way to redirect, you should consider the following explanation. The redirector runs as a daemon, it fires up N redirect daemons, so there is no problem with Perl interpreter loading. Exactly as with mod_perl, the perl interpreter is loaded all the time in memory and the code has already been compiled, so the redirect is very fast (not much slower than if the redirector was written in C). Squid keeps an open pipe to each redirect daemon, thus there is no overhead of the system calls.

Now it is time to restart the server, at linux I do it with:

```
/etc/rc.d/init.d/squid restart
```

Now the Squid server setup is complete.

Almost... When you try the new setup, you will be surprised and upset to discover port 81 showing up in the URLs of the static objects (like htmls). Hey, we did not want the user to see the port 81 and use it instead of 80, since then it will bypass the squid server and the hard work we went through was just a waste of time!

The solution is to make both squid and httpd_docs listen to the same port. This can be accomplished by binding each one to a specific interface (so they are listening to different **sockets**). Modify *httpd_docs/conf/httpd.conf*:

```
Port 80
BindAddress 127.0.0.1
Listen 127.0.0.1:80
```

Now the httpd_docs server is listening only to requests coming from the local server. You cannot access it directly from the outside. Squid becomes a gateway that all the packets go through on the way to the httpd_docs server.

Modify *squid.conf*:

```
http_port 80
tcp_outgoing_address 127.0.0.1
httpd_accel_host 127.0.0.1
httpd_accel_port 80
```

Now restart the Squid and httpd_docs servers (it doesn't matter which one you start first), and voila--the port number has gone.

You must also have in the file */etc/hosts* the following entry (chances are that it's already there):

```
127.0.0.1 localhost.localdomain localhost
```

Now if your scripts are generating HTML including fully qualified self references, using 8080 or the other port, you should fix them to generate links to point to port 80 (which means not using the port at all in the URI). If you do not do this, users will bypass Squid and will make direct requests to the mod_perl server's port. As we will see later just like with httpd_docs, the httpd_perl server can be configured to listen only to requests coming from the localhost (with Squid forwarding these requests from the outside) and therefore users will not be able to bypass Squid.

To save you some keystrokes, here is the whole modified `squid.conf`:

```
http_port 80
tcp_outgoing_address 127.0.0.1
httpd_accel_host 127.0.0.1
httpd_accel_port 80

icp_port 0

acl QUERY urlpath_regex /cgi-bin /perl
no_cache deny QUERY

# debug_options 28

redirect_program /usr/lib/squid/redirect.pl
redirect_children 10
redirect_rewrites_host_header off

request_body_max_size 1000 KB

acl all src 0.0.0.0/0.0.0.0
acl manager proto cache_object
acl localhost src 127.0.0.1/255.255.255.255
acl myserver src 127.0.0.1/255.255.255.255
acl SSL_ports port 443 563
acl Safe_ports port 80 81 8080 81 443 563
acl CONNECT method CONNECT

http_access allow manager localhost
http_access allow manager myserver
http_access deny manager
http_access deny !Safe_ports
http_access deny CONNECT !SSL_ports
# http_access allow all

cache_effective_user squid
cache_effective_group squid

cache_mem 20 MB

memory_pools on
```

```
cachemgr_passwd disable shutdown
```

Note that all directives should start at the beginning of the line, so if you cut and paste from the text make sure you remove the leading whitespace from each line.

13.6 Running One Webserver and Squid in httpd Accelerator Mode

When I was first told about Squid, I thought: "Hey, now I can drop the `httpd_docs` server and have just Squid and the `httpd_perl` servers". Since all my static objects will be cached by squid, I do not need the light `httpd_docs` server.

But I was a wrong. Why? Because I still have the overhead of loading the objects into Squid the first time. If a site has many of them, unless a huge chunk of memory is devoted to Squid they won't all be cached and the heavy mod_perl server will still have the task of serving static objects.

How do we measure the overhead? The difference between the two servers is in memory consumption, everything else (e.g. I/O) should be equal. So you have to estimate the time needed to fetch each static object for the first time at a peak period and thus the number of additional servers you need for serving the static objects. This will allow you to calculate the additional memory requirements. I imagine that this amount could be significant in some installations.

So on for production servers I have decided to stick with the Squid, httpd_docs and httpd_perl scenario, where I can optimize and fine tune everything. But if in your case there is almost no static objects to serve, the httpd_docs server is definitely redundant. And all you need are the mod_perl server and Squid to buffer the output from it.

If you want to proceed with this setup, install mod_perl enabled Apache and Squid. Then use a configuration similar to the previous section, but now httpd_docs is not there anymore. Also we do not need the redirector anymore and we specify `httpd_accel_host` as a name of the server and not `virtual`. Because we do not redirect there is no need to bind two servers on the same port so there are neither `Bind` nor `Listen` directives in *httpd.conf*.

The modified configuration for this simplified setup (see the explanations in the previous section):

```
httpd_accel_host put.your.hostname.here
httpd_accel_port 8080
http_port 80
icp_port 0

acl QUERY urlpath_regex /cgi-bin /perl
no_cache deny QUERY

# debug_options 28

# redirect_program /usr/lib/squid/redirect.pl
# redirect_children 10
# redirect_rewrites_host_header off
```

```
request_body_max_size 1000 KB

acl all src 0.0.0.0/0.0.0.0
acl manager proto cache_object
acl localhost src 127.0.0.1/255.255.255.255
acl myserver src 127.0.0.1/255.255.255.255
acl SSL_ports port 443 563
acl Safe_ports port 80 81 8080 81 443 563
acl CONNECT method CONNECT

http_access allow manager localhost
http_access allow manager myserver
http_access deny manager
http_access deny !Safe_ports
http_access deny CONNECT !SSL_ports
# http_access allow all

cache_effective_user squid
cache_effective_group squid

cache_mem 20 MB

memory_pools on

cachemgr_passwd disable shutdown
```

13.7 mod_proxy

mod_proxy implements a proxy/cache for Apache. It implements proxying capability for FTP, CONNECT (for SSL), HTTP/0.9, and HTTP/1.0. The module can be configured to connect to other proxy modules for these and other protocols.

13.7.1 Concepts and Configuration Directives

In the following explanation, we will use *www.example.com* as the main server users access when they want to get some kind of service and *backend.example.com* as a machine that does the heavy work. The main and the back-end are different servers, they may or may not coexist on the same machine.

The mod_proxy module is built into the server that answers requests to the *www.example.com* hostname. For the matter of this discussion it doesn't matter what functionality is built into the *backend.example.com* server, obviously it'll be mod_perl for most of us.

13.7.1.1 ProxyPass

You can use the `ProxyPass` configuration directive for mapping remote hosts into the space of the local server; the local server does not act as a proxy in the conventional sense, but appears to be a mirror of the remote server.

Let's explore what this rule does:

```
ProxyPass   /modperl/ http://backend.example.com/modperl/
```

When a user initiates a request to http://www.example.com/modperl/foo.pl, the request will be redirected to http://backend.example.com/modperl/foo.pl, and starting from this moment user will see http://backend.example.com/ in her location window, instead of http://www.example.com/.

You have probably noticed many examples of this from real life Internet sites you've visited. Free-email service providers and other similar heavy online services display the login or the main page from their main server, and then when you log-in you see something like *x11.example.com*, then *w59.example.com*, etc. These are the back-end servers that do the actual work.

Obviously this is not an ideal solution, but usually users don't really care about what they see in the location window. So you can get away with this approach. As I'll show in a minute there is a better solution which removes this caveat and provides even more useful functionalities.

13.7.1.2 ProxyPassReverse

This directive lets Apache adjust the URL in the `Location` header on HTTP redirect responses. This is essential for example, when Apache is used as a reverse proxy to avoid by-passing the reverse proxy because of HTTP redirects on the back-end servers which stay behind the reverse proxy. Generally used in conjunction with the `ProxyPass` directive to build a complete front-end proxy server.

```
ProxyPass          /modperl/  http://backend.example.com/modperl/
ProxyPassReverse   /modperl/  http://backend.example.com/modperl/
```

When a user initiates a request to http://www.example.com/modperl/foo.pl, the request will be redirected to http://backend.example.com/modperl/foo.pl but on the way back `ProxyPassReverse` will correct the location URL to become http://www.example.com/modperl/foo.pl . This happens completely transparently. The end user will never know that something has happened to his request behind the scenes.

Note that this `ProxyPassReverse` directive can also be used in conjunction with the proxy pass-through feature:

```
RewriteRule ... [P]
```

from mod_rewrite because its doesn't depend on a corresponding `ProxyPass` directive.

13.7.1.3 Security Issues

Whenever you use mod_proxy you need to make sure that your server will not become a proxy for free riders. Allowing clients to issue proxy requests is controlled by the `ProxyRequests` directive. Its default setting is `off`, which means proxy requests are handled only if generated internally (by `ProxyPass` or `RewriteRule...[P]` directives.) Do not use the `ProxyRequests` directive on your reverse proxy servers.

13.7.2 Buffering Feature

In addition to correcting the URI on its way back from the back-end server, mod_proxy also provides buffering services which mod_perl and similar heavy modules benefit from. The buffering feature allows mod_perl to pass the generated data to mod_proxy and move on to serve new requests, instead of waiting for a possibly slow client to receive all the data.

This figure depicts this feature:

```
               [socket]                    wire     ‘o’
[mod_perl] => [      ] => [mod_proxy] => ____ =>  /|\
               [buffer]                           / \
```

From looking at this figure it’s easy to see that the bottleneck is the socket buffer; it has to be able to absorb all the data that mod_perl has generated in order to free the mod_perl process immediately; mod_proxy will take the data as fast as mod_perl can deliver it, freeing the mod_perl server to service new requests as soon as possible while mod_proxy feeds the client at whatever rate the client requires.

`ProxyReceiveBufferSize` is the name of the parameter that specifies the size of the socket buffer. Configuring:

```
ProxyReceiveBufferSize 16384
```

will create a buffer of 16KB in size. If mod_perl generates output which is less than 16KB, the process will be immediately untied and allowed to serve new requests, if the output is bigger than 16KB, the following process will take place:

1. The first 16KB will enter the system buffer.

2. mod_proxy picks the first 8KB and sends it down the wire.

3. mod_perl writes the next 8KB into the place of the 8KB of data that was just sent off by mod_proxy.

Stages 2 and 3 are repeated until mod_perl has no more data to send. When this happens, mod_perl can serve a new request while stage 2 is repeated until all the data was picked from the system buffer and sent down the wire.

Of course you want to set the buffer size as large as possible, since you want the heavy mod_perl processes to be utilized in the most efficient way, so you don’t want them to waste their time waiting for a client to receive the data, especially if a client has a slow downstream connection.

As the `ProxyReceiveBufferSize` name states, its buffering feature applies only to *downstream data* (coming from the origin server to the proxy) and not upstream data. There is no buffering of data uploaded from the client browser to the proxy, thus you cannot use this technique to prevent the heavy mod_perl server from being tied up during a large POST such as a file upload. Falling back to mod_cgi seems to be the best solution for these specific scripts whose major function is receiving large amounts of upstream data.

[META: check this: --]

Of course just like mod_perl, mod_proxy writes the data it proxy-passes into its outgoing socket buffer, therefore the mod_proxy process gets released as soon as the last chunk of data is deposited into this buffer, even if the client didn't complete the download. Its the OS's problem to complete the transfer and release the TCP socket used for this transfer.

Therefore if you don't use mod_proxy and mod_perl sends its data directly to the client, and you have a big socket buffer, the mod_perl process will be released as soon as the last chunk of data enters the buffer. Just like with mod_proxy, the OS will deal with completing the data transfer.

[based on this comment] yes, too (but receive and transmit buffer may be of different size, depending on the OS)

The problem I don't know is, does the call to close the socket wait, until all data is actually send successfully or not. If it doesn't wait, you may not be noticed of any failure, but because the proxying Apache can write as fast to the socket transmission buffer as it can read, it should be possible that the proxying Apache copies all the data from the receive to the transmission buffer and after that releasing the receive buffer, so the mod_perl Apache is free to do other things, while the proxying Apache still wait until the client returns the success of data transmission. (The last, is the part I am not sure on)

[/META]

Unfortunately you cannot set the socket buffer size as large as you want because there is a limit of the available physical memory and OSs have their own upper limits on the possible buffer size.

This doesn't mean that you cannot change the OS imposed limits, but to do that you have to know the techniques for doing that. In the next section we will present a few OSs and the ways to increase their socket buffer sizes.

To increase the physical memory limits you just have to add more memory.

13.7.3 Setting the Buffering Limits on Various OSs

As we just saw there are a few kinds of parameters we might want to adjust for our needs.

13.7.3.1 IOBUFSIZE Source Code Definition

The first parameter is used by *proxy_util.c:ap_proxy_send_fb()* to loop over content being proxy passed in 8KB chunks (as of this writing), passing that on to the client. In other words it specifies the size of the data that is sent down the wire.

This parameter is defined by the `IOBUFSIZE`:

```
#define IOBUFSIZE 8192
```

You have no control over this setting in the server configuration file, therefore you might want to change it in the source files, before you compile the server.

13.7.3.2 ProxyReceiveBufferSize Configuration Directive

You can control the socket buffer size with the `ProxyReceiveBufferSize` directive:

```
ProxyReceiveBufferSize 16384
```

The above setting will set a buffer size of 16KB. If it is not set explicitly, or if it is set to 0, then the default buffer size is used. The number should be an integral multiple of 512.

Note that if you set the value of `ProxyReceiveBufferSize` larger than the OS limit, the default value will be used.

Both the default and the maximum possible value of `ProxyReceiveBufferSize` depend on the Operating System.

- **Linux**

  For 2.2 kernels the maximum limit is in */proc/sys/net/core/rmem_max* and the default value is in */proc/sys/net/core/rmem_default*. If you want to increase `RCVBUF` size above 65535, the default maximum value, you have to raise first the absolute limit in */proc/sys/net/core/rmem_max*. To do that at the run time, execute this command to raise it to 128KB:

  ```
  % echo 131072 > /proc/sys/net/core/rmem_max
  ```

  You probably want to put this command into */etc/rc.d/rc.local* so the change will take effect at system reboot.

  On Linux OS with kernel 2.2.5 the maximum and default values are either 32KB or 64KB. You can also change the default and maximum values during kernel compilation; for that you should alter the `SK_RMEM_DEFAULT` and `SK_RMEM_MAX` definitions respectively. (Since kernel source files tend to change, use grep(1) utility to find the files.)

- **FreeBSD**

  Under FreeBSD it’s possible to configure the kernel to have bigger socket buffers:

  ```
  % sysctl -w kern.ipc.maxsockbuf=2621440
  ```

- **Solaris**

  Under Solaris this upper limit is specified by *tcp_max_buf* parameter and is 256KB.

- **Other OSs**

  [ReaderMeta]: If you use an OS that is not listed here and know how to increase the socket buffer size please let me know.

When you tell the kernel to use bigger sockets you can set bigger values for *ProxyReceiveBufferSize*. e.g. 1048576 (1MB).

13.7.3.3 Hacking the Code

Some folks have patched the Apache's 1.3.x source code to make the application buffer configurable as well. After the patch there are two configuration directives available:

- ProxyReceiveBufferSize -- sets the socket buffer size
- ProxyInternalBufferSize -- sets the application buffer size

To patch the source, rename ap_breate() to ap_bcreate_size() and add a size parameter, which defaults to *IOBUFSIZE* if 0 is passed. Then add

```
#define ap_bcreate(p,flags) ap_bcreate(p,flags,0)
```

and add a new ap_bcreate() which calls ap_bcreate_size() for binary compatibility.

Actually the `ProxyReceiveBufferSize` should be called `ProxySocketBufferSize`. This would also remove some of the confusion about what it actually does.

13.7.4 Caching Feature

META: complete the conf details

Apache does caching as well. It's relevant to mod_perl only if you produce proper headers, so your scripts' output can be cached. See the Apache documentation for more details on the configuration of this capability.

13.7.5 Build Process

To build mod_proxy into Apache just add *--enable-module=proxy* during the Apache **./configure** stage. Since you probably will need the mod_rewrite capability enable it as well with *--enable-module=rewrite*.

13.8 Front-end Back-end Proxying with Virtual Hosts

This section explains a configuration setup for proxying your back-end mod_perl servers when you need to use Virtual Hosts.

The term *Virtual Host* refers to the practice of maintaining more than one server on one machine, as differentiated by their apparent hostname. For example, it is often desirable for companies sharing a web server to have their own domains, with web servers accessible as *www.company1.com* and *www.company2.com*, without requiring the user to know any extra path information.

The approach is to use a unique port number for each virtual host at the back-end server, so you can redirect from the front-end server to localhost:1234, and name-based virtual servers on the front end, though any technique on the front-end will do.

If you run the front-end and the back-end servers on the same machine you can prevent any direct outside connections to the back-end server if you bind tightly to address `127.0.0.1` (*localhost*) as you will see in the following configuration example.

The front-end (light) server configuration:

```
<VirtualHost 10.10.10.10>
  ServerName www.example.com
  ServerAlias example.com
  RewriteEngine On
  RewriteOptions 'inherit'
  RewriteRule \.(gif|jpg|png|txt|html)$ - [last]
  RewriteRule ^/(.*)$ http://localhost:4077/$1 [proxy]
</VirtualHost>

<VirtualHost 10.10.10.10>
  ServerName foo.example.com
  RewriteEngine On
  RewriteOptions 'inherit'
  RewriteRule \.(gif|jpg|png|txt|html)$ - [last]
  RewriteRule ^/(.*)$ http://localhost:4078/$1 [proxy]
</VirtualHost>
```

The above front-end configuration handles two virtual hosts: *www.example.com* and *foo.example.com*. The two setups are almost identical.

The front-end server will handle files with the extensions *.gif*, *.jpg*, *.png*, *.txt* and *.html* internally, the rest will be proxied to be handled by the back-end server.

The only difference between the two virtual hosts settings is that the former rewrites requests to port `4077` at the back-end machine and the latter to port `4078`.

If your server is configured to run traditional CGI scripts (under mod_cgi) as well as mod_perl CGI programs, then it would be beneficial to configure the front-end server to run the traditional CGI scripts directly. This can be done by altering the `gif|jpg|png|txt` *Rewrite* rule to add `|cgi` at the end if all your mod_cgi scripts have the *.cgi* extension, or adding a new rule to handle all `/cgi-bin/*` locations locally.

The back-end (heavy) server configuration:

```
Port 80

PerlPostReadRequestHandler My::ProxyRemoteAddr

Listen 4077
<VirtualHost localhost:4077>
  ServerName www.example.com
  DocumentRoot /home/httpd/docs/www.example.com
  DirectoryIndex index.shtml index.html
```

```
</VirtualHost>

Listen 4078
<VirtualHost localhost:4078>
  ServerName foo.example.com
  DocumentRoot /home/httpd/docs/foo.example.com
  DirectoryIndex index.shtml index.html
</VirtualHost>
```

The back-end server knows to tell which virtual host the request is made to, by checking the port number the request was proxied to and using the appropriate virtual host section to handle it.

We set "Port 80" so that any redirects don't get sent directly to the back-end port.

To get the *real* remote IP addresses from proxy, the My::ProxyRemoteAddr handler is used based on the `mod_proxy_add_forward` Apache module. Prior to mod_perl 1.22 this setting must have been set per-virtual host, since it wasn't inherited by the virtual hosts.

The following configuration is yet another useful example showing the other way around. It specifies what to be proxied and then the rest is served by the front end:

```
RewriteEngine     on
RewriteLogLevel   0
RewriteRule       ^/(perl.*)$  http://127.0.0.1:8052/$1   [P,L]
NoCache           *
ProxyPassReverse  /  http://www.example.com/
```

So we don't have to specify the rule for static objects to be served by the front-end as we did in the previous example to handle files with the extensions *.gif*, *.jpg*, *.png* and *.txt* internally.

13.9 Getting the Remote Server IP in the Back-end server in the Proxy Setup

Ask Bjoern Hansen has written the `mod_proxy_add_forward` module for Apache. It sets the `X-Forwarded-For` field when doing a `ProxyPass`, similar to what Squid can do. Its location is specified in the download section.

Basically, this module adds an extra HTTP header to proxying requests. You can access that header in the mod_perl-enabled server, and set the IP address of the remote server. You won't need to compile anything into the back-end server.

13.9.1 Build

Download the module and use its location as a value of the *--activate-module* argument for the *./configure* utility within the Apache source code, so the module can be found.

```
./configure \
"--with-layout=Apache" \
"--activate-module=src/modules/extra/mod_proxy_add_forward.c" \
"--enable-module=proxy_add_forward" \
... other options ...
```

--enable-module=proxy_add_forward enables this module as you have guessed already.

13.9.2 Usage

If you are using `Apache::Registry` or `Apache::PerlRun` modules just put the following code into *startup.pl*:

```
use Apache::Constants ();
sub My::ProxyRemoteAddr ($) {
  my $r = shift;

  # we'll only look at the X-Forwarded-For header if the requests
  # comes from our proxy at localhost
  return Apache::Constants::OK
      unless ($r->connection->remote_ip eq "127.0.0.1")
          and $r->header_in('X-Forwarded-For');

  # Select last value in the chain -- original client's ip
  if (my ($ip) = $r->headers_in->{'X-Forwarded-For'} =~ /([^,\s]+)$/) {
    $r->connection->remote_ip($ip);
  }

  return Apache::Constants::OK;
}
```

And in the mod_perl's *httpd.conf*:

```
PerlPostReadRequestHandler My::ProxyRemoteAddr
```

and the right thing will happen transparently for your scripts. Otherwise if you write your own mod_perl content handler, you can retrieve it directly in your code using a similar code.

13.9.3 Security

Different sites have different needs. If you use the header to set the IP address, Apache believes it. This is reflected in the logging for example. You really don't want anyone but your own system to set the header, which is why the *recommended code* above checks where the request really came from before changing `remote_ip`.

Generally you shouldn't trust the `X-Forwarded-For` header. You only want to rely on `X-Forwarded-For` headers from proxies you control yourself. If you know how to spoof a cookie you've probably got the general idea on making HTTP headers and can spoof the `X-Forwarded-For` header as well. The only address you can count on as being a reliable value is the one from `r->connection->remote_ip`.

From that point on, the remote IP address is correct. You should be able to access `$ENV{REMOTE_ADDR}` environment variable as usual.

13.9.4 Caveats

It was reported that Ben Laurie's Apache-SSL does not seem to put the IP addresses in the `X-Forwarded-For` header--it does not set up such a header at all. However, the `$ENV{REMOTE_ADDR}` environment variable it sets up contains the IP address of the original client machine.

Prior to mod_perl 1.22 there was a need to repeat the PerlPostReadRequestHandler My::ProxyRemoteAddr directive for each virtual host, since it wasn't inherited by the virtual hosts.

13.9.5 mod_proxy_add_forward Module's Order Precedence

Some users report that they cannot get this module to work as advertised. They verify that the module is built in, but the front-end server is not generating the `X-Forwarded-For` header when requests are being proxied to the back-end server. As a result, the back-end server has no idea what the remote IP is.

As it turns out, *mod_proxy_add_forward* needs to be configured in Apache before *mod_proxy* in order to operate properly, since Apache gives highest precedence to the last defined module.

Moving the two build options required to enable *mod_proxy_add_forward* while configuring Apache appears to have no effect on the default configuration order of modules, since in each case, the resulting builds show *mod_proxy_add_forward* last in the list (or first via */server-info*).

One solution is to explicitly define the configuration order in the *http.conf* file, so that *mod_proxy_add_forward* appears before *mod_proxy*, and therefore gets executed after *mod_proxy*. (Modules are being executed in *reverse* order, i.e. module that was *Added* first will be executed last.)

Obviously, this list would need to be tailored to match the build environment, but to ease this task just insert an `AddModule` directive before each entry reported by `httpd -l` (and removing *httpd_core.c*, of course):

```
ClearModuleList
AddModule mod_env.c
[more modules snipped]
AddModule mod_proxy_add_forward.c
AddModule mod_proxy.c
AddModule mod_rewrite.c
AddModule mod_setenvif.c
```

Note that the above snippet is added to *httpd.conf* of the front-end server.

Another solution is to reorder the module list during configuration by using one or more `--permute-module` arguments to the *./configure* utility. (Try `./configure --help` to see if your version of Apache supports this option.) `--permute-module=foo:bar` will swap the position of *mod_foo* and *mod_bar* in the list, `--permute-module=BEGIN:foo` will move *mod_foo* to the beginning of the list, and `--permute-module=foo:END` will move *mod_foo* to the end. For example

suppose your module list from `httpd -l` looks like:

```
http_core.c
[more modules snipped]
mod_proxy.c
mod_setenvif.c
mod_proxy_add_forward.c
```

You might add the following arguments to *./configure* to move *mod_proxy_add_forward* to the position in the list just before *mod_proxy*:

```
./configure \
"--with-layout=Apache" \
"--activate-module=src/modules/extra/mod_proxy_add_forward.c" \
"--enable-module=proxy_add_forward" \
... other options ...
"--permute-module=proxy:proxy_add_forward" \
"--permute-module=setenvif:END"
```

With this change, the `X-Forwarded-For` header is now being sent to the back-end server, and the remote IP appears in the back-end server's *access_log* file.

13.10 HTTP Authentication With Two Servers Plus a Proxy

Assuming that you have a setup of one "front-end" server, which proxies the "back-end" (mod_perl) server, if you need to perform authentication in the "back-end" server it should handle all authentication itself. If Apache proxies correctly, it will pass through all authentication information, making the "front-end" Apache somewhat "dumb", as it does nothing but pass through the information.

In the configuration file your `Auth` configuration directives need to be inside the `<Directory ...>` ... `</Directory>` sections because if you use the section `<Location ...>` ... `</Location>` the proxy server will take the authentication information for itself and not pass it on.

The same applies to mod_ssl and similar Apache SSL modules. If it gets plugged into a front-end server, it will properly encode/decode all the SSL requests. So if your machine is secured from inside, your back-end server can do secure transactions.

13.11 mod_rewrite Examples

Example code for using mod_rewrite with mod_perl application servers. Several examples were taken from the mailing list.

13.11.1 Rewriting Requests Based on File Extension

In the mod_proxy + mod_perl servers scenario, `ProxyPass` was used to redirect all requests to the mod_perl server, by matching the beginning of the relative URI (e.g. */perl*). What should you do if you want everything, but files with extensions like *.gif, .cgi* and similar, to be proxypassed to the mod_perl server. These files are to be served by the light Apache server which carries the mod_proxy module.

The following example rewrites everything to the mod_perl server. It locally handles all requests for files with extensions *gif*, *jpg*, *png*, *css*, *txt*, *cgi* and relative URIs starting with */cgi-bin* (e.g. if you want some scripts to be executed under mod_cgi).

```
RewriteEngine On
# handle GIF and JPG images and traditional CGI's directly
RewriteRule \.(gif|jpg|png|css|txt|cgi)$ - [last]
RewriteRule ^/cgi-bin - [last]
# pass off everything but images to the heavy-weight server via proxy
RewriteRule ^/(.*)$ http://localhost:4077/$1 [proxy]
```

That is, first, handle locally what you want to handle locally, then hand off everything else to the back-end guy.

This is the configuration of the logging facilities.

```
RewriteLogLevel 1
RewriteLog "| /usr/local/apache_proxy/bin/rotatelogs \
/usr/local/apache-common/logs/r_log 86400"
```

It says: log all the rewrites thru the pipe to the `rotatelogs` utility which will rotate the logs every 2 hours (86400 secs).

13.11.2 Internet Exporer 5 favicon.ico 404

Redirect all those IE5 requests for *favicon.ico* to a central image:

```
RewriteRule .*favicon.ico /wherever/favicon.ico [PT,NS]
```

13.11.3 Hiding Extensions for Dynamic Pages

A quick way to make dynamic pages look static:

```
RewriteRule ^/wherever/([a-zA-Z]+).html /perl-bin/$1.cgi [PT]
```

13.11.4 Serving Static Content Locally and Rewriting Everything Else

Instead of keeping all your Perl scripts in */perl* and your static content everywhere else, you could keep your static content in special directories and keep your Perl scripts everywhere else. You can still use the light/heavy apache separation approach described before, with a few minor modifications.

In the *light* Apache's *httpd.conf* file, turn rewriting on:

```
RewriteEngine On
```

Now list all directories that contain only static objects. For example if the only relative to `Document-Root` directories are */images* and *style* you can set the following rule:

```
RewriteRule ^/(images|style) - [L]
```

The `[L]` (*Last*) means that the rewrite engine should stop if it has a match. This is necessary because the very last rewrite rule proxies everything to the *heavy* server:

```
RewriteRule ^/(.*) http://www.example.com:8080/$1 [P]
```

This line is the difference between a server for which static content is the default and one for which dynamic (perlish) content is the default.

You should also add the *reverse rewrite rule* as before:

```
ProxyPassReverse / http://www.example.com/
```

so that the user doesn't see the port number `:8080` in the browser's location window.

It is possible to use `localhost` in the `RewriteRule` above if the heavy and light servers are on the same machine. So if we sum up the above setup we get:

```
RewriteEngine On
RewriteRule ^/(images|style) - [L]
RewriteRule ^/(.*) http://www.example.com:8080/$1 [P]
ProxyPassReverse / http://www.example.com/
```

13.11.5 Upgrading mod_perl Heavy Application Instances

When using a light/heavy separation method one of the challenges of running a production environment is being able to upgrade to newer versions of mod_perl or your own application. The following method can be used without having to do a server restart.

Add the following rewrite rule to your httpd.conf file:

```
RewriteEngine On
RewriteMap maps txt:/etc/httpd.maps
RewriteRule ^(.*) http://${maps:appserver}$1 [proxy]
```

Create the file /etc/httpd.maps and add the following entry:

```
appserver foo.com:9999
```

Mod_rewrite rereads (or checks the mtime of) the file on every request so the change takes effect immediately. To seamlessly upgrade your application server to a new version, install a new version on a different port. After checking for a quality installation, edit /etc/httpd.maps to point to the new server. After the file is written the next request the server processes will be redirected to the new installation.

13.11.6 Blocking IP Addresses

The following rewrite code blocks IP addresses:

```
RewriteCond /web/site/var/blocked/REMOTE_ADDR-%{REMOTE_ADDR} -f
RewriteRule .* http://YOUR-HOST-BLOCKED-FOR-EXCESSIVE-CONSUMPTION [redirect,last]
```

To block IP address 10.1.2.3, simply touch

```
/web/site/var/blocked/REMOTE_ADDR-10.1.2.3
```

This has an advantage over Apache parsing a long file of addresses in that the OS is better at a file lookup.

13.12 Caching in mod_proxy

This is not really mod_perl related, so I'll just stress one point. If you want the caching to work the following HTTP headers should be supplied: `Last-Modified`, `Content-Length` and `Expires`.

13.13 Maintainers

Maintainer is the person(s) you should contact with updates, corrections and patches.

- Stas Bekman [http://stason.org/]

13.14 Authors

- Stas Bekman [http://stason.org/]

Only the major authors are listed above. For contributors see the Changes file.

14 Performance Tuning

14.1 Description

An exhaustive list of various techniques you might want to use to get the most performance possible out of your mod_perl server: configuration, coding, memory use, and more.

14.2 The Big Picture

To make the user's Web browsing experience as painless as possible, every effort must be made to wring the last drop of performance from the server. There are many factors which affect Web site usability, but speed is one of the most important. This applies to any webserver, not just Apache, so it is very important that you understand it.

How do we measure the speed of a server? Since the user (and not the computer) is the one that interacts with the Web site, one good speed measurement is the time elapsed between the moment when she clicks on a link or presses a *Submit* button to the moment when the resulting page is fully rendered.

The requests and replies are broken into packets. A request may be made up of several packets, a reply may be many thousands. Each packet has to make its own way from one machine to another, perhaps passing through many interconnection nodes. We must measure the time starting from when the first packet of the request leaves our user's machine to when the last packet of the reply arrives back there.

A webserver is only one of the entities the packets see along their way. If we follow them from browser to server and back again, they may travel by different routes through many different entities. Before they are processed by your server the packets might have to go through proxy (accelerator) servers and if the request contains more than one packet, packets might arrive to the server by different routes with different arrival times, therefore it's possible that some packets that arrive earlier will have to wait for other packets before they could be reassembled into a chunk of the request message that will be then read by the server. Then the whole process is repeated in reverse.

You could work hard to fine tune your webserver's performance, but a slow Network Interface Card (NIC) or a slow network connection from your server might defeat it all. That's why it's important to think about the Big Picture and to be aware of possible bottlenecks between the server and the Web.

Of course there is little that you can do if the user has a slow connection. You might tune your scripts and webserver to process incoming requests ultra quickly, so you will need only a small number of working servers, but you might find that the server processes are all busy waiting for slow clients to accept their responses.

But there are techniques to cope with this. For example you can deliver the respond after it was compressed. If you are delivering a pure text respond--gzip compression will sometimes reduce the size of the respond by 10 times.

You should analyze all the involved components when you try to create the best service for your users, and not the web server or the code that the web server executes. A Web service is like a car, if one of the parts or mechanisms is broken the car may not go smoothly and it can even stop dead if pushed too far without first fixing it.

And let me stress it again--if you want to have a success in the web service business you should start worrying about the client's browsing experience and **not only** how good your code benchmarks are.

14.3 System Analysis

Before we try to solve a problem we need to identify it. In our case we want to get the best performance we can with as little monetary and time investment as possible.

14.3.1 Software Requirements

Covered in the section "Choosing an Operating System".

14.3.2 Hardware Requirements

(META: Only partial analysis. Please submit more points. Many points are scattered around the document and should be gathered here, to represent the whole picture. It also should be merged with the above item!)

You need to analyze all of the problem's dimensions. There are several things that need to be considered:

- How long does it take to process each request?
- How many requests can you process simultaneously?
- How many simultaneous requests are you planning to get?
- At what rate are you expecting to receive requests?

The first one is probably the easiest to optimize. Following the performance optimization tips in this and other documents allows a perl (mod_perl) programmer to exercise their code and improve it.

The second one is a function of RAM. How much RAM is in each box, how many boxes do you have, and how much RAM does each mod_perl process use? Multiply the first two and divide by the third. Ask yourself whether it is better to switch to another, possibly just as inefficient language or whether that will actually cost more than throwing another powerful machine into the rack.

Also ask yourself whether switching to another language will even help. In some applications, for example to link Oracle runtime libraries, a huge chunk of memory is needed so you would save nothing even if you switched from Perl to C.

The last two are important. You need a realistic estimate. Are you really expecting 8 million hits per day? What is the expected peak load, and what kind of response time do you need to guarantee? Remember that these numbers might change drastically when you apply code changes and your site becomes popular. Remember that when you get a very high hit rate, the resource requirements don't grow linearly but exponentially!

More coverage is provided in the section "Choosing Hardware".

14.4 Essential Tools

In order to improve performance we need measurement tools. The main tool categories are benchmarking and code profiling.

It's important to understand that in a major number of the benchmarking tests that we will execute we will not look at the absolute result numbers but the relation between the two and more result sets, since in most cases we would try to show which coding approach is preferable and the you shouldn't try to compare the absolute results collected while running the same benchmarks on your machine, since you won't have the exact hardware and software setup anyway. So this kind of comparison would be misleading. Compare the relative results from the tests running on your machine, don't compare your absolute results with those in this Guide.

14.4.1 Benchmarking Applications

How much faster is mod_perl than mod_cgi (aka plain perl/CGI)? There are many ways to benchmark the two. I'll present a few examples and numbers below. Check out the `benchmark` directory of the mod_perl distribution for more examples.

If you are going to write your own benchmarking utility, use the `Benchmark` module for heavy scripts and the `Time::HiRes` module for very fast scripts (faster than 1 sec) where you will need better time precision.

There is no need to write a special benchmark though. If you want to impress your boss or colleagues, just take some heavy CGI script you have (e.g. a script that crunches some data and prints the results to STDOUT), open 2 xterms and call the same script in mod_perl mode in one xterm and in mod_cgi mode in the other. You can use `lwp-get` from the `LWP` package to emulate the browser. The `benchmark` directory of the mod_perl distribution includes such an example.

See also two tools for benchmarking: ApacheBench and crashme test

14.4.1.1 Benchmarking Perl Code

If you are going to write your own benchmarking utility, use the `Benchmark` module and the `Time::HiRes` module where you need better time precision (less than 10msec).

An example of the `Benchmark.pm` module usage:

```
benchmark.pl
------------
use Benchmark;

timethis (1_000,
 sub {
  my $x = 100;
  my $y = log ($x ** 100)  for (0..10000);
```

```
});

% perl benchmark.pl
timethis 1000: 25 wallclock secs (24.93 usr +  0.00 sys = 24.93 CPU)
```

If you want to get the benchmark results in micro-seconds you will have to use the `Time::HiRes` module, its usage is similar to `Benchmark`'s.

```
use Time::HiRes qw(gettimeofday tv_interval);
my $start_time = [ gettimeofday ];
sub_that_takes_a_teeny_bit_of_time();
my $end_time = [ gettimeofday ];
my $elapsed = tv_interval($start_time,$end_time);
print "The sub took $elapsed seconds."
```

See also the crashme test.

14.4.1.2 Benchmarking a Graphic Hits Counter with Persistent DB Connections

Here are the numbers from Michael Parker's mod_perl presentation at the Perl Conference (Aug, 98). (Sorry, there used to be links here to the source, but they went dead one day, so I removed them). The script is a standard hits counter, but it logs the counts into a mysql relational DataBase:

```
Benchmark: timing 100 iterations of cgi, perl...  [rate 1:28]

cgi: 56 secs ( 0.33 usr 0.28 sys = 0.61 cpu)
perl: 2 secs ( 0.31 usr 0.27 sys = 0.58 cpu)

Benchmark: timing 1000 iterations of cgi,perl...  [rate 1:21]

cgi: 567 secs ( 3.27 usr 2.83 sys = 6.10 cpu)
perl: 26 secs ( 3.11 usr 2.53 sys = 5.64 cpu)

Benchmark: timing 10000 iterations of cgi, perl   [rate 1:21]

cgi: 6494 secs (34.87 usr 26.68 sys = 61.55 cpu)
perl: 299 secs (32.51 usr 23.98 sys = 56.49 cpu)
```

We don't know what server configurations were used for these tests, but I guess the numbers speak for themselves.

The source code of the script was available at http://www.realtime.net/~parkerm/perl/conf98/sld006.htm. It's now a dead link. If you know its new location, please let me know.

14.4.1.3 Benchmarking Response Times

In the next sections we will talk about tools that allow us to benchmark response times.

14.4.1.3.1 ApacheBench

ApacheBench (`ab`) is a tool for benchmarking your Apache HTTP server. It is designed to give you an idea of the performance that your current Apache installation can give. In particular, it shows you how many requests per second your Apache server is capable of serving. The `ab` tool comes bundled with the Apache source distribution.

Let's try it. We will simulate 10 users concurrently requesting a very light script at `www.example.com/perl/test.pl`. Each simulated user makes 10 requests.

```
% ./ab -n 100 -c 10 www.example.com/perl/test.pl
```

The results are:

```
Document Path:          /perl/test.pl
Document Length:        319 bytes

Concurrency Level:      10
Time taken for tests:   0.715 seconds
Complete requests:      100
Failed requests:        0
Total transferred:      60700 bytes
HTML transferred:       31900 bytes
Requests per second:    139.86
Transfer rate:          84.90 kb/s received

Connection Times (ms)
              min   avg   max
Connect:        0     0     3
Processing:    13    67    71
Total:         13    67    74
```

We can see that under load of ten concurrent users our server is capable of processing 140 requests per second. Of course this benchmark is correct only when the script under test is used. We can also learn about the average processing time, which in this case was 67 milli-seconds. Other numbers reported by `ab` may or may not be of interest to you.

For example if we believe that the script *perl/test.pl* is not efficient we will try to improve it and run the benchmark again, to see whether we have any improve in performance.

`HTTPD::Bench::ApacheBench`, available from CPAN, provides a Perl interface for `ab`.

14.4.1.3.2 httperf

httperf is a utility written by David Mosberger. Just like ApacheBench, it measures the performance of the webserver.

A sample command line is shown below:

```
httperf --server hostname --port 80 --uri /test.html \
 --rate 150 --num-conn 27000 --num-call 1 --timeout 5
```

This command causes httperf to use the web server on the host with IP name hostname, running at port 80. The web page being retrieved is */test.html* and, in this simple test, the same page is retrieved repeatedly. The rate at which requests are issued is 150 per second. The test involves initiating a total of 27,000 TCP connections and on each connection one HTTP call is performed. A call consists of sending a request and receiving a reply.

The timeout option defines the number of seconds that the client is willing to wait to hear back from the server. If this timeout expires, the tool considers the corresponding call to have failed. Note that with a total of 27,000 connections and a rate of 150 per second, the total test duration will be approximately 180 seconds (27,000/150), independently of what load the server can actually sustain. Here is a result that one might get:

```
Total: connections 27000 requests 26701 replies 26701 test-duration 179.996 s

Connection rate: 150.0 conn/s (6.7 ms/conn, <=47 concurrent connections)
Connection time [ms]: min 1.1 avg 5.0 max 315.0 median 2.5 stddev 13.0
Connection time [ms]: connect 0.3

Request rate: 148.3 req/s (6.7 ms/req)
Request size [B]: 72.0

Reply rate [replies/s]: min 139.8 avg 148.3 max 150.3 stddev 2.7 (36 samples)
Reply time [ms]: response 4.6 transfer 0.0
Reply size [B]: header 222.0 content 1024.0 footer 0.0 (total 1246.0)
Reply status: 1xx=0 2xx=26701 3xx=0 4xx=0 5xx=0

CPU time [s]: user 55.31 system 124.41 (user 30.7% system 69.1% total 99.8%)
Net I/O: 190.9 KB/s (1.6*10^6 bps)

Errors: total 299 client-timo 299 socket-timo 0 connrefused 0 connreset 0
Errors: fd-unavail 0 addrunavail 0 ftab-full 0 other 0
```

httperf download

14.4.1.3.3 http_load

`http_load` is yet another utility that does webserver load testing. It can simulate 33.6kbps modem connection (*-throttle*) and allows you to provide a file with a list of URLs, which we be fetched randomly. You can specify how many parallel connections to run using the *-parallel N* option, or you can specify the number of requests to generate per second with *-rate N* option. Finally you can tell the utility when to stop by specifying either the test time length (*-seconds N*) or the total number of fetches (*-fetches N*).

A sample run with the file *urls* including:

```
http://www.example.com/foo/
http://www.example.com/bar/
```

We ask to generate three requests per second and run for only two seconds. Here is the generated output:

```
% ./http_load -rate 3 -seconds 2 urls
http://www.example.com/foo/: check-connect SUCCEEDED, ignoring
http://www.example.com/bar/: check-connect SUCCEEDED, ignoring
http://www.example.com/bar/: check-connect SUCCEEDED, ignoring
http://www.example.com/bar/: check-connect SUCCEEDED, ignoring
http://www.example.com/foo/: check-connect SUCCEEDED, ignoring
5 fetches, 3 max parallel, 96870 bytes, in 2.00258 seconds
19374 mean bytes/connection
2.49678 fetches/sec, 48372.7 bytes/sec
msecs/connect: 1.805 mean, 5.24 max, 0.79 min
msecs/first-response: 291.289 mean, 560.338 max, 34.349 min
```

So you can see that it has reported 2.5 requests per second. Of course for the real test you will want to load the server heavily and run the test for a longer time to get more reliable results.

Note that when you provide a file with a list of URLs make sure that you don't have empty lines in it. If you do -- the utility won't work complaining:

```
./http_load: unknown protocol -
```

http_load download

14.4.1.3.4 the crashme Script

This is another crashme suite originally written by Michael Schilli (and was located at http://www.linux-magazin.de site, but now the link has gone). I made a few modifications, mostly adding my () operators. I also allowed it to accept more than one url to test, since sometimes you want to test more than one script.

The tool provides the same results as **ab** above but it also allows you to set the timeout value, so requests will fail if not served within the time out period. You also get values for **Latency** (seconds per request) and **Throughput** (requests per second). It can do a complete simulation of your favorite Netscape browser :) and give you a better picture.

I have noticed while running these two benchmarking suites, that **ab** gave me results from two and a half to three times better. Both suites were run on the same machine, with the same load and the same parameters, but the implementations were different.

Sample output:

```
URL(s):          http://www.example.com/perl/access/access.cgi
Total Requests:  100
Parallel Agents: 10
Succeeded:       100 (100.00%)
Errors:          NONE
Total Time:      9.39 secs
Throughput:      10.65 Requests/sec
Latency:         0.85 secs/Request
```

And the code:

The LWP::Parallel::UserAgent benchmark: *code/lwp-bench.pl*:

```
#!/usr/bin/perl -w

use LWP::Parallel::UserAgent;
use Time::HiRes qw(gettimeofday tv_interval);
use strict;

###
# Configuration
###

my $nof_parallel_connections = 10;
my $nof_requests_total = 100;
my $timeout = 10;
my @urls = (
            'http://www.example.com:81/perl/faq_manager/faq_manager.pl',
            'http://www.example.com:81/perl/access/access.cgi',
           );


#####################################################
# Derived Class for latency timing
#####################################################

package MyParallelAgent;
@MyParallelAgent::ISA = qw(LWP::Parallel::UserAgent);
use strict;

###
# Is called when connection is opened
###
sub on_connect {
  my ($self, $request, $response, $entry) = @_;
  $self->{__start_times}->{$entry} = [Time::HiRes::gettimeofday];
}

###
# Are called when connection is closed
###
sub on_return {
  my ($self, $request, $response, $entry) = @_;
  my $start = $self->{__start_times}->{$entry};
  $self->{__latency_total} += Time::HiRes::tv_interval($start);
}

sub on_failure {
  on_return(@_);  # Same procedure
}

###
# Access function for new instance var
###
sub get_latency_total {
  return shift->{__latency_total};
```

```
}

##################################################
package main;
##################################################
###
# Init parallel user agent
###
my $ua = MyParallelAgent->new();
$ua->agent("pounder/1.0");
$ua->max_req($nof_parallel_connections);
$ua->redirect(0);    # No redirects

###
# Register all requests
###
foreach (1..$nof_requests_total) {
  foreach my $url (@urls) {
    my $request = HTTP::Request->new('GET', $url);
    $ua->register($request);
  }
}

###
# Launch processes and check time
###
my $start_time = [gettimeofday];
my $results = $ua->wait($timeout);
my $total_time = tv_interval($start_time);

###
# Requests all done, check results
###

my $succeeded     = 0;
my %errors = ();

foreach my $entry (values %$results) {
  my $response = $entry->response();
  if($response->is_success()) {
    $succeeded++; # Another satisfied customer
  } else {
    # Error, save the message
    $response->message("TIMEOUT") unless $response->code();
    $errors{$response->message}++;
  }
}

###
# Format errors if any from %errors
###
my $errors = join(',', map "$_ ($errors{$_})", keys %errors);
$errors = "NONE" unless $errors;

###
# Format results
###
```

```
#@urls = map {($_,".")} @urls;
my @P = (
      "URL(s)"          => join("\n\t\t ", @urls),
      "Total Requests"  => $nof_requests_total * @urls,
      "Parallel Agents" => $nof_parallel_connections,
      "Succeeded"       => sprintf("$succeeded (%.2f%%)\n",
                                   $succeeded * 100 / ( $nof_requests_total * @urls ) ),
      "Errors"          => $errors,
      "Total Time"      => sprintf("%.2f secs\n", $total_time),
      "Throughput"      => sprintf("%.2f Requests/sec\n",
                                   ( $nof_requests_total * @urls ) / $total_time),
      "Latency"         => sprintf("%.2f secs/Request",
                                   ($ua->get_latency_total() || 0) /
                                   ( $nof_requests_total * @urls ) ),
     );

my ($left, $right);
###
# Print out statistics
###
format STDOUT =
@<<<<<<<<<<<<<<< @*
"$left:",        $right
.

while(($left, $right) = splice(@P, 0, 2)) {
  write;
}
```

14.4.1.4 Benchmarking PerlHandlers

The `Apache::Timeit` module does `PerlHandler` Benchmarking. With the help of this module you can log the time taken to process the request, just like you'd use the `Benchmark` module to benchmark a regular Perl script. Of course you can extend this module to perform more advanced processing like putting the results into a database for a later processing. But all it takes is adding this configuration directive inside *httpd.conf*:

```
  PerlFixupHandler Apache::Timeit
```

Since scripts running under `Apache::Registry` are running inside the PerlHandler these are benchmarked as well.

An example of the lines which show up in the *error_log* file:

```
  timing request for /perl/setupenvoff.pl:
    0 wallclock secs ( 0.04 usr +  0.01 sys =  0.05 CPU)
  timing request for /perl/setupenvoff.pl:
    0 wallclock secs ( 0.03 usr +  0.00 sys =  0.03 CPU)
```

The `Apache::Timeit` package is a part of the *Apache-Perl-contrib* files collection available from CPAN.

14.4.1.5 Other Benchmarking Tools

Other tools you may want to take a look at:

- **`HTTP::WebTest`**

  `HTTP::WebTest` module runs tests on remote URLs or local web files containing Perl/JSP/HTML/JavaScript/etc. and generates a detailed test report.

  It's available from CPAN.

- **`HTTP::Monkeywrench`**

  `HTTP::Monkeywrench` is a test-harness application to test the integrity of a user's path through a web site.

  It's available from CPAN.

- **`Apache::Recorder` and `HTTP::RecordedSession`**

  `Apache::Recorder` is a mod_perl handler that records an HTTP session and stores it on the web server's file system. `HTTP::RecordedSession` reads the recorded session from the file system, and formats it for playback using `HTTP::WebTest` or `HTTP::Monkeywrench`. This is useful when writing acceptance and regression tests.

  It's available from CPAN.

- **`Webstone`**

  This tool is somewhat complex to set up, but once you get it running it gives you stats that you could only duplicate with ab or http_load if you did quite a bit of extra scripting around them. It also allows multiple client machines to be used for providing heavy loads. This tool is useful if you need to know things like at what point people start finding your sight slow, as opposed to at what point the server becomes unresponsive.

  Webstone download

- **`Flood`**

  Flood is a load-tester being developed through the Apache Software Foundation. From the Flood FAQ:

  "Flood is a profile-driven HTTP load tester. In layman's terms, it means that flood is capable of generating large amounts of web traffic. Flood's flexibility and power arises in its configuration syntax. It is able to work well with dynamic content."

  Flood download

14.4.2 Code Profiling Techniques

The profiling process helps you to determine which subroutines or just snippets of code take the longest time to execute and which subroutines are called most often. Probably you will want to optimize those.

When do you need to profile your code? You do that when you suspect that some part of your code is called very often and may be there is a need to optimize it to significantly improve the overall performance.

For example if you have ever used the `diagnostics` pragma, which extends the terse diagnostics normally emitted by both the Perl compiler and the Perl interpreter, augmenting them with the more verbose and endearing descriptions found in the `perldiag` manpage. You know that it might tremendously slow you code down, so let's first prove that it is correct.

We will run a benchmark, once with diagnostics enabled and once disabled, on a subroutine called *test_code*.

The code inside the subroutine does an arithmetic and a numeric comparison of two strings. It assigns one string to another if the condition tests true but the condition always tests false. To demonstrate the `diagnostics` overhead the comparison operator is intentionally *wrong*. It should be a string comparison, not a numeric one.

```
use Benchmark;
use diagnostics;
use strict;

my $count = 50000;

disable diagnostics;
my $t1 = timeit($count,\&test_code);

enable  diagnostics;
my $t2 = timeit($count,\&test_code);

print "Off: ",timestr($t1),"\n";
print "On : ",timestr($t2),"\n";

sub test_code{
  my ($a,$b) = qw(foo bar);
  my $c;
  if ($a == $b) {
    $c = $a;
  }
}
```

For only a few lines of code we get:

```
Off:  1 wallclock secs ( 0.81 usr +  0.00 sys =  0.81 CPU)
On : 13 wallclock secs (12.54 usr +  0.01 sys = 12.55 CPU)
```

With `diagnostics` enabled, the subroutine test_code() is 16 times slower, than with `diagnostics` disabled!

Now let's fix the comparison the way it should be, by replacing `==` with `eq`, so we get:

```
my ($a,$b) = qw(foo bar);
my $c;
if ($a eq $b) {
  $c = $a;
}
```

and run the same benchmark again:

```
Off:  1 wallclock secs ( 0.57 usr +  0.00 sys =  0.57 CPU)
On :  1 wallclock secs ( 0.56 usr +  0.00 sys =  0.56 CPU)
```

Now there is no overhead at all. The `diagnostics` pragma slows things down only when warnings are generated.

After we have verified that using the `diagnostics` pragma might adds a big overhead to execution runtime, let's use the code profiling to understand why this happens. We are going to use `Devel::DProf` to profile the code. Let's use this code:

```
diagnostics.pl
--------------
use diagnostics;
print "Content-type:text/html\n\n";
test_code();
sub test_code{
  my ($a,$b) = qw(foo bar);
  my $c;
  if ($a == $b) {
    $c = $a;
  }
}
```

Run it with the profiler enabled, and then create the profiling stastics with the help of dprofpp:

```
% perl -d:DProf diagnostics.pl
% dprofpp

Total Elapsed Time = 0.342236 Seconds
  User+System Time = 0.335420 Seconds
Exclusive Times
%Time ExclSec CumulS #Calls sec/call Csec/c  Name
 92.1   0.309  0.358      1   0.3089 0.3578  main::BEGIN
 14.9   0.050  0.039   3161   0.0000 0.0000  diagnostics::unescape
 2.98   0.010  0.010      2   0.0050 0.0050  diagnostics::BEGIN
 0.00   0.000 -0.000      2   0.0000      -  Exporter::import
 0.00   0.000 -0.000      2   0.0000      -  Exporter::export
 0.00   0.000 -0.000      1   0.0000      -  Config::BEGIN
 0.00   0.000 -0.000      1   0.0000      -  Config::TIEHASH
 0.00   0.000 -0.000      2   0.0000      -  Config::FETCH
 0.00   0.000 -0.000      1   0.0000      -  diagnostics::import
 0.00   0.000 -0.000      1   0.0000      -  main::test_code
```

```
  0.00   0.000 -0.000      2   0.0000      -  diagnostics::warn_trap
  0.00   0.000 -0.000      2   0.0000      -  diagnostics::splainthis
  0.00   0.000 -0.000      2   0.0000      -  diagnostics::transmo
  0.00   0.000 -0.000      2   0.0000      -  diagnostics::shorten
  0.00   0.000 -0.000      2   0.0000      -  diagnostics::autodescribe
```

It’s not easy to see what is responsible for this enormous overhead, even if `main::BEGIN` seems to be running most of the time. To get the full picture we must see the OPs tree, which shows us who calls whom, so we run:

```
 % dprofpp -T
```

and the output is:

```
main::BEGIN
  diagnostics::BEGIN
     Exporter::import
        Exporter::export
  diagnostics::BEGIN
     Config::BEGIN
     Config::TIEHASH
     Exporter::import
        Exporter::export
  Config::FETCH
  Config::FETCH
  diagnostics::unescape
  ......................
  3159 times [diagnostics::unescape] snipped
  ......................
  diagnostics::unescape
  diagnostics::import
diagnostics::warn_trap
  diagnostics::splainthis
     diagnostics::transmo
     diagnostics::shorten
     diagnostics::autodescribe
main::test_code
  diagnostics::warn_trap
     diagnostics::splainthis
        diagnostics::transmo
        diagnostics::shorten
        diagnostics::autodescribe
  diagnostics::warn_trap
     diagnostics::splainthis
        diagnostics::transmo
        diagnostics::shorten
       diagnostics::autodescribe
```

So we see that two executions of `diagnostics::BEGIN` and 3161 of `diagnostics::unescape` are responsible for most of the running overhead.

If we comment out the `diagnostics` module, we get:

```
Total Elapsed Time = 0.079974 Seconds
  User+System Time = 0.059974 Seconds
Exclusive Times
%Time ExclSec CumulS #Calls sec/call Csec/c  Name
 0.00   0.000 -0.000      1   0.0000      -  main::test_code
```

It is possible to profile code running under mod_perl with the `Devel::DProf` module, available on CPAN. However, you must have apache version 1.3b3 or higher and the `PerlChildExitHandler` enabled during the httpd build process. When the server is started, `Devel::DProf` installs an `END` block to write the *tmon.out* file. This block will be called at server shutdown. Here is how to start and stop a server with the profiler enabled:

```
% setenv PERL5OPT -d:DProf
% httpd -X -d ‘pwd‘ &
... make some requests to the server here ...
% kill ‘cat logs/httpd.pid‘
% unsetenv PERL5OPT
% dprofpp
```

The `Devel::DProf` package is a Perl code profiler. It will collect information on the execution time of a Perl script and of the subs in that script (remember that `print()` and `map()` are just like any other subroutines you write, but they come bundled with Perl!)

Another approach is to use `Apache::DProf`, which hooks `Devel::DProf` into mod_perl. The `Apache::DProf` module will run a `Devel::DProf` profiler inside each child server and write the *tmon.out* file in the directory `$ServerRoot/logs/dprof/$$` when the child is shutdown (where `$$` is the number of the child process). All it takes is to add to *httpd.conf*:

```
PerlModule Apache::DProf
```

Remember that any PerlHandler that was pulled in before `Apache::DProf` in the *httpd.conf* or *startup.pl*, will not have its code debugging information inserted. To run `dprofpp`, chdir to `$Server-Root/logs/dprof/$$` and run:

```
% dprofpp
```

(Lookup the `ServerRoot` directive’s value in *httpd.conf* to figure out what’s your `$ServerRoot`.)

14.4.3 Measuring the Memory of the Process

Very important aspect of performance tuning is to make sure that your applications don’t use much memory, since if they do you cannot run many servers and therefore in most cases under a heavy load the overall performance degrades.

In addition the code may not be clean and leak memory, which is even worse, since if the same process serves many requests and after each request more memory is used, after awhile all RAM will be used and machine will start swapping (use the swap partition) which is a very undesirable event, since it may lead to a machine crash.

The simplest way to figure out how big the processes are and see whether they grow is to watch the output of top(1) or ps(1) utilities.

For example the output of top(1):

```
  8:51am  up 66 days,  1:44,  1 user,  load average: 1.09, 2.27, 2.61
95 processes: 92 sleeping, 3 running, 0 zombie, 0 stopped
CPU states: 54.0% user,  9.4% system,  1.7% nice, 34.7% idle
Mem:  387664K av, 309692K used,  77972K free, 111092K shrd,  70944K buff
Swap: 128484K av,  11176K used, 117308K free                170824K cached

   PID USER PRI NI SIZE  RSS SHARE STAT LIB %CPU %MEM   TIME COMMAND
29225 nobody 0  0  9760 9760  7132 S      0 12.5  2.5   0:00 httpd_perl
29220 nobody 0  0  9540 9540  7136 S      0  9.0  2.4   0:00 httpd_perl
29215 nobody 1  0  9672 9672  6884 S      0  4.6  2.4   0:01 httpd_perl
29255 root   7  0  1036 1036   824 R      0  3.2  0.2   0:01 top
  376 squid  0  0 15920  14M   556 S      0  1.1  3.8 209:12 squid
29227 mysql  5  5  1892 1892   956 S N    0  1.1  0.4   0:00 mysqld
29223 mysql  5  5  1892 1892   956 S N    0  0.9  0.4   0:00 mysqld
29234 mysql  5  5  1892 1892   956 S N    0  0.9  0.4   0:00 mysqld
```

Which starts with overall information of the system and then displays the most active processes at the given moment. So for example if we look at the `httpd_perl` processes we can see the size of the resident (RSS) and shared (SHARE) memory segments. This sample was taken on the production server running linux.

But of course we want to see all the apache/mod_perl processes, and that's where ps(1) comes to help. The options of this utility vary from one Unix flavor to another, and some flavors provide their own tools. Let's check the information about mod_perl processes:

```
% ps -o pid,user,rss,vsize,%cpu,%mem,ucomm -C httpd_perl
  PID USER      RSS   VSZ %CPU %MEM COMMAND
29213 root     8584 10264  0.0  2.2 httpd_perl
29215 nobody   9740 11316  1.0  2.5 httpd_perl
29216 nobody   9668 11252  0.7  2.4 httpd_perl
29217 nobody   9824 11408  0.6  2.5 httpd_perl
29218 nobody   9712 11292  0.6  2.5 httpd_perl
29219 nobody   8860 10528  0.0  2.2 httpd_perl
29220 nobody   9616 11200  0.5  2.4 httpd_perl
29221 nobody   8860 10528  0.0  2.2 httpd_perl
29222 nobody   8860 10528  0.0  2.2 httpd_perl
29224 nobody   8860 10528  0.0  2.2 httpd_perl
29225 nobody   9760 11340  0.7  2.5 httpd_perl
29235 nobody   9524 11104  0.4  2.4 httpd_perl
```

Now you can see the resident (RSS) and virtual (VSZ) memory segments (and shared memory segment if you ask for it) of all mod_perl processes. Please refer to the top(1) and ps(1) man pages for more information.

You probably agree that using top(1) and ps(1) is cumbersome if we want to use memory size sampling during the benchmark test. We want to have a way to print memory sizes during the program execution at desired places. If you have `GTop` modules installed, which is a perl glue to the `libgtop` library, it's exactly what we need.

Note: GTop requires the libgtop library but is not available for all platforms. See the docs in the source at ftp://ftp.gnome.org/pub/GNOME/stable/sources/gtop/ to check whether your platform/flavor is supported.

GTop provides an API for retrieval of information about processes and the whole system. We are interested only in memory sampling API methods. To print all the process related memory information we can execute the following code:

```
use GTop;
my $gtop = GTop->new;
my $proc_mem = $gtop->proc_mem($$);
for (qw(size vsize share rss)) {
    printf "   %s => %d\n", $_, $proc_mem->$_();
}
```

When executed we see the following output (in bytes):

```
     size => 1900544
    vsize => 3108864
    share => 1392640
      rss => 1900544
```

So if we are interested in to print the process resident memory segment before and after some event we just do it: For example if we want to see how much extra memory was allocated after a variable creation we can write the following code:

```
use GTop;
my $gtop = GTop->new;
my $before = $gtop->proc_mem($$)->rss;
my $x = 'a' x 10000;
my $after  = $gtop->proc_mem($$)->rss;
print "diff: ",$after-$before, " bytes\n";
```

and the output

```
diff: 20480 bytes
```

So we can see that Perl has allocated extra 20480 bytes to create $x (of course the creation of after needed a few bytes as well, but it's insignificant compared to a size of $x)

The Apache::VMonitor module with help of the GTop module allows you to watch all your system information using your favorite browser from anywhere in the world without a need to telnet to your machine. If you are looking at what information you can retrieve with GTop, you should look at Apache::VMonitor as it deploys a big part of the API GTop provides.

If you are running a true BSD system, you may use BSD::Resource::getrusage instead of GTop. For example:

```
print "used memory = ".(BSD::Resource::getrusage)[2]."\n"
```

For more information refer to the `BSD::Resource` manpage.

14.4.4 Measuring the Memory Usage of Subroutines

With help of `Apache::Status` you can find out the size of each and every subroutine.

1. Build and install mod_perl as you always do, make sure it's version 1.22 or higher.
2. Configure /perl-status if you haven't already:

```
<Location /perl-status>
  SetHandler perl-script
  PerlHandler Apache::Status
  order deny,allow
  #deny from all
  #allow from ...
</Location>
```

3. Add to httpd.conf

```
PerlSetVar StatusOptionsAll On
PerlSetVar StatusTerse On
PerlSetVar StatusTerseSize On
PerlSetVar StatusTerseSizeMainSummary On

PerlModule B::TerseSize
```

4. Start the server (best in httpd -X mode)
5. From your favorite browser fetch http://localhost/perl-status
6. Click on 'Loaded Modules' or 'Compiled Registry Scripts'
7. Click on the module or script of your choice (you might need to run some script/handler before you will see it here unless it was preloaded)
8. Click on 'Memory Usage' at the bottom
9. You should see all the subroutines and their respective sizes.

Now you can start to optimize your code. Or test which of the several implementations is of the least size.

For example let's compare `CGI.pm`'s OO vs. procedural interfaces:

As you will see below the first OO script uses about 2k bytes while the second script (procedural interface) uses about 5k.

Here are the code examples and the numbers:

1.

```
cgi_oo.pl
---------
use CGI ();
my $q = CGI->new;
print $q->header;
print $q->b("Hello");
```

2.

```
cgi_mtd.pl
---------
use CGI qw(header b);
print header();
print b("Hello");
```

After executing each script in single server mode (-X) the results are:

1.

```
Totals: 1966 bytes | 27 OPs

handler 1514 bytes | 27 OPs
exit     116 bytes |  0 OPs
```

2.

```
Totals: 4710 bytes | 19 OPs

handler  1117 bytes | 19 OPs
basefont  120 bytes |  0 OPs
frameset  120 bytes |  0 OPs
caption   119 bytes |  0 OPs
applet    118 bytes |  0 OPs
script    118 bytes |  0 OPs
ilayer    118 bytes |  0 OPs
header    118 bytes |  0 OPs
strike    118 bytes |  0 OPs
layer     117 bytes |  0 OPs
table     117 bytes |  0 OPs
frame     117 bytes |  0 OPs
style     117 bytes |  0 OPs
Param     117 bytes |  0 OPs
small     117 bytes |  0 OPs
embed     117 bytes |  0 OPs
font      116 bytes |  0 OPs
span      116 bytes |  0 OPs
exit      116 bytes |  0 OPs
big       115 bytes |  0 OPs
div       115 bytes |  0 OPs
sup       115 bytes |  0 OPs
Sub       115 bytes |  0 OPs
TR        114 bytes |  0 OPs
```

```
td        114 bytes |  0 OPs
Tr        114 bytes |  0 OPs
th        114 bytes |  0 OPs
b         113 bytes |  0 OPs
```

Note, that the above is correct if you didn't precompile all `CGI.pm`'s methods at server startup. Since if you did, the procedural interface in the second test will take up to 18k and not 5k as we saw. That's because the whole of `CGI.pm`'s namespace is inherited and it already has all its methods compiled, so it doesn't really matter whether you attempt to import only the symbols that you need. So if you have:

```
use CGI  qw(-compile :all);
```

in the server startup script. Having:

```
use CGI qw(header);
```

or

```
use CGI qw(:all);
```

is essentially the same. You will have all the symbols precompiled at startup imported even if you ask for only one symbol. It seems to me like a bug, but probably that's how `CGI.pm` works.

BTW, you can check the number of opcodes in the code by a simple command line run. For example comparing 'my %hash' vs. 'my %hash = ()'.

```
% perl -MO=Terse -e 'my %hash' | wc -l
-e syntax OK
    4

% perl -MO=Terse -e 'my %hash = ()' | wc -l
-e syntax OK
   10
```

The first one has less opcodes.

Note that you shouldn't use `Apache::Status` module on production server as it adds quite a bit of overhead for each request.

14.5 Know Your Operating System

In order to get the best performance it helps to get intimately familiar with the Operating System (OS) the web server is running on. There are many OS specific things that you may be able to optimize which will improve your web server's speed, reliability and security.

The following sections will reveal some of the most important details you should know about your OS.

14.5.1 Sharing Memory

The sharing of memory is one very important factor. If your OS supports it (and most sane systems do), you might save memory by sharing it between child processes. This is only possible when you preload code at server startup. However, during a child process' life its memory pages tend to become unshared.

There is no way we can make Perl allocate memory so that (dynamic) variables land on different memory pages from constants, so the **copy-on-write** effect (we will explain this in a moment) will hit you almost at random.

If you are pre-loading many modules you might be able to trade off the memory that stays shared against the time for an occasional fork by tuning `MaxRequestsPerChild`. Each time a child reaches this upper limit and dies it should release its unshared pages. The new child which replaces it will share its fresh pages until it scribbles on them.

The ideal is a point where your processes usually restart before too much memory becomes unshared. You should take some measurements to see if it makes a real difference, and to find the range of reasonable values. If you have success with this tuning the value of `MaxRequestsPerChild` will probably be peculiar to your situation and may change with changing circumstances.

It is very important to understand that your goal is not to have `MaxRequestsPerChild` to be 10000. Having a child serving 300 requests on precompiled code is already a huge overall speedup, so if it is 100 or 10000 it probably does not really matter if you can save RAM by using a lower value.

Do not forget that if you preload most of your code at server startup, the newly forked child gets ready very fast, because it inherits most of the preloaded code and the perl interpreter from the parent process.

During the life of the child its memory pages (which aren't really its own to start with, it uses the parent's pages) gradually get 'dirty' - variables which were originally inherited and shared are updated or modified -- and the *copy-on-write* happens. This reduces the number of shared memory pages, thus increasing the memory requirement. Killing the child and spawning a new one allows the new child to get back to the pristine shared memory of the parent process.

The recommendation is that `MaxRequestsPerChild` should not be too large, otherwise you lose some of the benefit of sharing memory.

See Choosing MaxRequestsPerChild for more about tuning the `MaxRequestsPerChild` parameter.

14.5.1.1 How Shared Is My Memory?

You've probably noticed that the word shared is repeated many times in relation to mod_perl. Indeed, shared memory might save you a lot of money, since with sharing in place you can run many more servers than without it. See the Formula and the numbers.

How much shared memory do you have? You can see it by either using the memory utility that comes with your system or you can deploy the `GTop` module:

```
use GTop ();
print "Shared memory of the current process: ",
  GTop->new->proc_mem($$)->share,"\n";

print "Total shared memory: ",
  GTop->new->mem->share,"\n";
```

When you watch the output of the `top` utility, don't confuse the RES (or RSS) columns with the SHARE column. RES is RESident memory, which is the size of pages currently swapped in.

14.5.1.2 Calculating Real Memory Usage

I have shown how to measure the size of the process' shared memory, but we still want to know what the real memory usage is. Obviously this cannot be calculated simply by adding up the memory size of each process because that wouldn't account for the shared memory.

On the other hand we cannot just subtract the shared memory size from the total size to get the real memory usage numbers, because in reality each process has a different history of processed requests, therefore the shared memory is not the same for all processes.

So how do we measure the real memory size used by the server we run? It's probably too difficult to give the exact number, but I've found a way to get a fair approximation which was verified in the following way. I have calculated the real memory used, by the technique you will see in the moment, and then have stopped the Apache server and saw that the memory usage report indicated that the total used memory went down by almost the same number I've calculated. Note that some OSs do smart memory pages caching so you may not see the memory usage decrease as soon as it actually happens when you quit the application.

This is a technique I've used:

1. For each process sum up the difference between shared and system memory. To calculate a difference for a single process use:

   ```
   use GTop;
   my $proc_mem = GTop->new->proc_mem($$);
   my $diff     = $proc_mem->size - $proc_mem->share;
   print "Difference is $diff bytes\n";
   ```

2. Now if we add the shared memory size of the process with maximum shared memory, we will get all the memory that actually is being used by all httpd processes, except for the parent process.

3. Finally, add the size of the parent process.

Please note that this might be incorrect for your system, so you use this number on your own risk.

I've used this technique to display real memory usage in the module Apache::VMonitor, so instead of trying to manually calculate this number you can use this module to do it automatically. In fact in the calculations used in this module there is no separation between the parent and child processes, they are all counted indifferently using the following code:

```
use GTop ();
my $gtop = GTop->new;
my $total_real = 0;
my $max_shared = 0;
# @mod_perl_pids is initialized by Apache::Scoreboard, irrelevant here
my @mod_perl_pids = some_code();
for my $pid (@mod_perl_pids)
  my $proc_mem = $gtop->proc_mem($pid);
  my $size     = $proc_mem->size($pid);
  my $share    = $proc_mem->share($pid);
  $total_real += $size - $share;
  $max_shared  = $share if $max_shared < $share;
}
my $total_real += $max_shared;
```

So as you see we that we accumulate the difference between the shared and reported memory:

```
  $total_real  += $size-$share;
```

and at the end add the biggest shared process size:

```
my $total_real += $max_shared;
```

So now `$total_real` contains approximately the really used memory.

14.5.1.3 Are My Variables Shared?

How do you find out if the code you write is shared between the processes or not? The code should be shared, except where it is on a memory page with variables that change. Some variables are read-only in usage and never change. For example, if you have some variables that use a lot of memory and you want them to be read-only. As you know the variable becomes unshared when the process modifies its value.

So imagine that you have this 10Mb in-memory database that resides in a single variable, you perform various operations on it and want to make sure that the variable is still shared. For example if you do some matching regular expression (regex) processing on this variable and want to use the pos() function, will it make the variable unshared or not?

The `Apache::Peek` module comes to rescue. Let's write a module called *MyShared.pm* which we preload at server startup, so all the variables of this module are initially shared by all children.

```
MyShared.pm
---------
package MyShared;
use Apache::Peek;

my $readonly = "Chris";

sub match    { $readonly =~ /\w/g;               }
sub print_pos{ print "pos: ",pos($readonly),"\n";}
sub dump     { Dump($readonly);                  }
1;
```

This module declares the package `MyShared`, loads the `Apache::Peek` module and defines the lexically scoped `$readonly` variable which is supposed to be a variable of large size (think about a huge hash data structure), but we will use a small one to simplify this example.

The module also defines three subroutines: match() that does a simple character matching, print_pos() that prints the current position of the matching engine inside the string that was last matched and finally the dump() subroutine that calls the `Apache::Peek` module's Dump() function to dump a raw Perl data-type of the `$readonly` variable.

Now we write the script that prints the process ID (PID) and calls all three functions. The goal is to check whether pos() makes the variable *dirty* and therefore unshared.

```
share_test.pl
-------------
use MyShared;
print "Content-type: text/plain\r\n\r\n";
print "PID: $$\n";
MyShared::match();
MyShared::print_pos();
MyShared::dump();
```

Before you restart the server, in *httpd.conf* set:

```
MaxClients 2
```

for easier tracking. You need at least two servers to compare the print outs of the test program. Having more than two can make the comparison process harder.

Now open two browser windows and issue the request for this script several times in both windows, so you get different processes PIDs reported in the two windows and each process has processed a different number of requests to the *share_test.pl* script.

In the first window you will see something like that:

```
PID: 27040
pos: 1
SV = PVMG(0x853db20) at 0x8250e8c
  REFCNT = 3
  FLAGS = (PADBUSY,PADMY,SMG,POK,pPOK)
  IV = 0
  NV = 0
  PV = 0x8271af0 "Chris"\0
  CUR = 5
  LEN = 6
  MAGIC = 0x853dd80
    MG_VIRTUAL = &vtbl_mglob
    MG_TYPE = 'g'
    MG_LEN = 1
```

And in the second window:

```
PID: 27041
pos: 2
SV = PVMG(0x853db20) at 0x8250e8c
  REFCNT = 3
  FLAGS = (PADBUSY,PADMY,SMG,POK,pPOK)
  IV = 0
  NV = 0
  PV = 0x8271af0 "Chris"\0
  CUR = 5
  LEN = 6
  MAGIC = 0x853dd80
    MG_VIRTUAL = &vtbl_mglob
    MG_TYPE = 'g'
    MG_LEN = 2
```

We see that all the addresses of the supposedly big structures are the same, 0x8250e8c for SV, and 0x8271af0 for PV, therefore the variable data structure is almost completely shared. The only difference is in `SV.MAGIC.MG_LEN` record, which is not shared.

So given that the `$readonly` variable is a big one, its value is still shared between the processes, while part of the variable data structure is non-shared. But it's almost insignificant because it takes a very little memory space.

Now if you need to compare more than variable, doing it by hand can be quite time consuming and error prune. Therefore it's better to correct the testing script to dump the Perl data-types into files (e.g */tmp/dump.$$*, where `$$` is the PID of the process) and then using diff(1) utility to see whether there is some difference.

So correcting the dump() function to write the info to the file will do the job. Notice that we use `Devel::Peek` and not `Apache::Peek`. The both are almost the same, but `Apache::Peek` prints it output directly to the opened socket so we cannot intercept and redirect the result to the file. Since `Devel::Peek` dumps results to the STDERR stream we can use the old trick of saving away the default STDERR handler, and open a new filehandler using the STDERR. In our example when `Devel::Peek` now prints to STDERR it actually prints to our file. When we are done, we make sure to restore the original STDERR filehandler.

So this is the resulting code:

```
MyShared2.pm
---------
package MyShared2;
use Devel::Peek;

my $readonly = "Chris";

sub match    { $readonly =~ /\w/g;               }
sub print_pos{ print "pos: ",pos($readonly),"\n";}
sub dump{
  my $dump_file = "/tmp/dump.$$";
  print "Dumping the data into $dump_file\n";
  open OLDERR, ">&STDERR";
  open STDERR, ">".$dump_file or die "Can't open $dump_file: $!";
  Dump($readonly);
```

```
  close STDERR ;
  open STDERR, ">&OLDERR";
}
1;
```

When if we modify the code to use the modified module:

```
share_test2.pl
--------------
use MyShared2;
print "Content-type: text/plain\r\n\r\n";
print "PID: $$\n";
MyShared2::match();
MyShared2::print_pos();
MyShared2::dump();
```

And run it as before (with MaxClients 2), two dump files will be created in the directory */tmp*. In our test these were created as */tmp/dump.1224* and */tmp/dump.1225*. When we run diff(1):

```
% diff /tmp/dump.1224 /tmp/dump.1225
12c12
<       MG_LEN = 1
---
>       MG_LEN = 2
```

We see that the two padlists (of the variable `readonly`) are different, as we have observed before when we did a manual comparison.

In fact we if we think about these results again, we get to a conclusion that there is no need for two processes to find out whether the variable gets modified (and therefore unshared). It's enough to check the datastructure before the script was executed and after that. You can modify the `MyShared2` module to dump the padlists into a different file after each invocation and than to run the diff(1) on the two files.

If you want to watch whether some lexically scoped (with my ()) variables in your `Apache::Registry` script inside the same process get changed between invocations you can use the `Apache::RegistryLexInfo` module instead. Since it does exactly this: it makes a snapshot of the padlist before and after the code execution and shows the difference between the two. This specific module was written to work with `Apache::Registry` scripts so it won't work for loaded modules. Use the technique we have described above for any type of variables in modules and scripts.

Surely another way of ensuring that a scalar is readonly and therefore sharable is to either use the `constant` pragma or `readonly` pragma. But then you won't be able to make calls that alter the variable even a little, like in the example that we just showed, because it will be a true constant variable and you will get compile time error if you try this:

```
MyConstant.pm
-------------
package MyConstant;
use constant readonly => "Chris";

sub match    { readonly =~ /\w/g;               }
sub print_pos{ print "pos: ",pos(readonly),"\n";}
1;
```

```
% perl -c MyConstant.pm

Can't modify constant item in match position at MyConstant.pm line
5, near "readonly)"
MyConstant.pm had compilation errors.
```

However this code is just right:

```
MyConstant1.pm
--------------
package MyConstant1;
use constant readonly => "Chris";

sub match { readonly =~ /\w/g; }
1;
```

14.5.1.4 Preloading Perl Modules at Server Startup

You can use the `PerlRequire` and `PerlModule` directives to load commonly used modules such as `CGI.pm`, `DBI` and etc., when the server is started. On most systems, server children will be able to share the code space used by these modules. Just add the following directives into *httpd.conf*:

```
PerlModule CGI
PerlModule DBI
```

But an even better approach is to create a separate startup file (where you code in plain perl) and put there things like:

```
use DBI ();
use Carp ();
```

Don't forget to prevent importing of the symbols exported by default by the module you are going to preload, by placing empty parentheses `()` after a module's name. Unless you need some of these in the startup file, which is unlikely. This will save you a few more memory bits.

Then you `require()` this startup file in *httpd.conf* with the `PerlRequire` directive, placing it before the rest of the mod_perl configuration directives:

```
PerlRequire /path/to/start-up.pl
```

`CGI.pm` is a special case. Ordinarily `CGI.pm` autoloads most of its functions on an as-needed basis. This speeds up the loading time by deferring the compilation phase. When you use mod_perl, FastCGI or another system that uses a persistent Perl interpreter, you will want to precompile the functions at initialization time. To accomplish this, call the package function compile() like this:

```
use CGI ();
CGI->compile(':all');
```

The arguments to `compile()` are a list of method names or sets, and are identical to those accepted by the `use()` and `import()` operators. Note that in most cases you will want to replace `':all'` with the tag names that you actually use in your code, since generally you only use a subset of them.

Let’s conduct a memory usage test to prove that preloading, reduces memory requirements.

In order to have an easy measurement we will use only one child process, therefore we will use this setting:

```
MinSpareServers 1
MaxSpareServers 1
StartServers 1
MaxClients 1
MaxRequestsPerChild 100
```

We are going to use the `Apache::Registry` script *memuse.pl* which consists of two parts: the first one preloads a bunch of modules (that most of them aren’t going to be used), the second part reports the memory size and the shared memory size used by the single child process that we start. and of course it prints the difference between the two sizes.

```
memuse.pl
---------
use strict;
use CGI ();
use DB_File ();
use LWP::UserAgent ();
use Storable ();
use DBI ();
use GTop ();

my $r = shift;
$r->send_http_header('text/plain');
my $proc_mem = GTop->new->proc_mem($$);
my $size  = $proc_mem->size;
my $share = $proc_mem->share;
my $diff  = $size - $share;
printf "%10s %10s %10s\n", qw(Size Shared Difference);
printf "%10d %10d %10d (bytes)\n",$size,$share,$diff;
```

First we restart the server and execute this CGI script when none of the above modules preloaded. Here is the result:

```
    Size   Shared     Diff
4706304  2134016  2572288 (bytes)
```

Now we take all the modules:

```
use strict;
use CGI ();
use DB_File ();
use LWP::UserAgent ();
use Storable ();
use DBI ();
use GTop ();
```

and copy them into the startup script, so they will get preloaded. The script remains unchanged. We restart the server and execute it again. We get the following.

```
    Size   Shared    Diff
4710400  3997696  712704 (bytes)
```

Let's put the two results into one table:

```
Preloading    Size   Shared     Diff
   Yes     4710400  3997696   712704 (bytes)
    No     4706304  2134016  2572288 (bytes)
-----------------------------------------
Difference    4096  1863680 -1859584
```

You can clearly see that when the modules weren't preloaded the shared memory pages size, were about 1864Kb smaller relative to the case where the modules were preloaded.

Assuming that you have had 256M dedicated to the web server, if you didn't preload the modules, you could have:

```
268435456 = X * 2572288 + 2134016

X = (268435456 - 2134016) / 2572288 = 103
```

103 servers.

Now let's calculate the same thing with modules preloaded:

```
268435456 = X * 712704 + 3997696

X = (268435456 - 3997696) / 712704 = 371
```

You can have almost 4 times more servers!!!

Remember that we have mentioned before that memory pages gets dirty and the size of the shared memory gets smaller with time? So we have presented the ideal case where the shared memory stays intact. Therefore the real numbers will be a little bit different, but not far from the numbers in our example.

Also it's obvious that in your case it's possible that the process size will be bigger and the shared memory will be smaller, since you will use different modules and a different code, so you won't get this fantastic ratio, but this example is certainly helps to feel the difference.

14.5.1.5 Preloading Registry Scripts at Server Startup

What happens if you find yourself stuck with Perl CGI scripts and you cannot or don't want to move most of the stuff into modules to benefit from modules preloading, so the code will be shared by the children. Luckily you can preload scripts as well. This time the `Apache::RegistryLoader` modules comes to aid. `Apache::RegistryLoader` compiles `Apache::Registry` scripts at server startup.

For example to preload the script */perl/test.pl* which is in fact the file */home/httpd/perl/test.pl* you would do the following:

```
use Apache::RegistryLoader ();
Apache::RegistryLoader->new->handler("/perl/test.pl",
                  "/home/httpd/perl/test.pl");
```

You should put this code either into `<Perl>` sections or into a startup script.

But what if you have a bunch of scripts located under the same directory and you don't want to list them one by one. Take the benefit of Perl modules and put them to a good use. The `File::Find` module will do most of the work for you.

The following code walks the directory tree under which all `Apache::Registry` scripts are located. For each encountered file with extension *.pl*, it calls the `Apache::RegistryLoader::handler()` method to preload the script in the parent server, before pre-forking the child processes:

```
use File::Find qw(finddepth);
use Apache::RegistryLoader ();
{
  my $scripts_root_dir = "/home/httpd/perl/";
  my $rl = Apache::RegistryLoader->new;
  finddepth
    (
     sub {
       return unless /\.pl$/;
       my $url = "$File::Find::dir/$_";
       $url =~ s|$scripts_root_dir/?|/|;
       warn "pre-loading $url\n";
         # preload $url
       my $status = $rl->handler($url);
       unless($status == 200) {
     warn "pre-load of ‘$url’ failed, status=$status\n";
       }
     },
     $scripts_root_dir);
}
```

Note that we didn't use the second argument to `handler()` here, as in the first example. To make the loader smarter about the URI to filename translation, you might need to provide a `trans()` function to translate the URI to filename. URI to filename translation normally doesn't happen until HTTP request time, so the module is forced to roll its own translation. If filename is omitted and a `trans()` function was not defined, the loader will try using the URI relative to **ServerRoot**.

A simple trans() function can be something like that:

```
sub mytrans {
  my $uri = shift;
  $uri =~ s|^/perl/|/home/httpd/perl/|;
  return $uri;
}
```

You can easily derive the right translation by looking at the `Alias` directive. The above mytrans() function is matching our `Alias`:

```
Alias /perl/ /home/httpd/perl/
```

After defining the URI to filename translation function you should pass it during the creation of the `Apache::RegistryLoader` object:

```
my $rl = Apache::RegistryLoader->new(trans => \&mytrans);
```

I won't show any benchmarks here, since the effect is absolutely the same as with preloading modules.

See also BEGIN blocks

14.5.1.6 Modules Initializing at Server Startup

We have just learned that it's important to preload the modules and scripts at the server startup. It turns out that it's not enough for some modules and you have to prerun their initialization code to get more memory pages shared. Basically you will find an information about specific modules in their respective manpages. We will present a few examples of widely used modules where the code can be initialized.

14.5.1.6.1 Initializing DBI.pm

The first example is the `DBI` module. As you know `DBI` works with many database drivers falling into the `DBD::` category, e.g. `DBD::mysql`. It's not enough to preload `DBI`, you should initialize `DBI` with driver(s) that you are going to use (usually a single driver is used), if you want to minimize memory use after forking the child processes. Note that you want to do this under mod_perl and other environments where the shared memory is very important. Otherwise you shouldn't initialize drivers.

You probably know already that under mod_perl you should use the `Apache::DBI` module to get the connection persistence, unless you open a separate connection for each user--in this case you should not use this module. `Apache::DBI` automatically loads `DBI` and overrides some of its methods, so you should continue coding like there is only a `DBI` module.

Just as with modules preloading our goal is to find the startup environment that will lead to the smallest *"difference"* between the shared and normal memory reported, therefore a smaller total memory usage.

And again in order to have an easy measurement we will use only one child process, therefore we will use this setting in *httpd.conf*:

```
MinSpareServers 1
MaxSpareServers 1
StartServers 1
MaxClients 1
MaxRequestsPerChild 100
```

We always preload these modules:

```
use Gtop();
use Apache::DBI(); # preloads DBI as well
```

We are going to run memory benchmarks on five different versions of the *startup.pl* file.

- **option 1**

  Leave the file unmodified.

- **option 2**

  Install MySQL driver (we will use MySQL RDBMS for our test):

  ```
  DBI->install_driver("mysql");
  ```

  It's safe to use this method, since just like with `use()`, if it can't be installed it'll die().

- **option 3**

  Preload MySQL driver module:

  ```
  use DBD::mysql;
  ```

- **option 4**

  Tell `Apache::DBI` to connect to the database when the child process starts (`ChildInitHandler`), no driver is preload before the child gets spawned!

  ```
  Apache::DBI->connect_on_init('DBI:mysql:test::localhost',
                               "",
                               "",
                               {
                                PrintError => 1, # warn() on errors
                                RaiseError => 0, # don't die on error
                                AutoCommit => 1, # commit executes
                                # immediately
                               }
                              )
  or die "Cannot connect to database: $DBI::errstr";
  ```

- **option 5**

  Options 2 and 4: using connect_on_init() and install_driver().

Here is the `Apache::Registry` test script that we have used:

```
preload_dbi.pl
--------------
use strict;
use GTop ();
use DBI ();

my $dbh = DBI->connect("DBI:mysql:test::localhost",
                       "",
                       "",
                       {
                        PrintError => 1, # warn() on errors
```

```
                        RaiseError => 0, # don't die on error
                        AutoCommit => 1, # commit executes
                                         # immediately
                       }
                      )
  or die "Cannot connect to database: $DBI::errstr";

my $r = shift;
$r->send_http_header('text/plain');

my $do_sql = "show tables";
my $sth = $dbh->prepare($do_sql);
$sth->execute();
my @data = ();
while (my @row = $sth->fetchrow_array){
  push @data, @row;
}
print "Data: @data\n";
$dbh->disconnect(); # NOP under Apache::DBI

my $proc_mem = GTop->new->proc_mem($$);
my $size  = $proc_mem->size;
my $share = $proc_mem->share;
my $diff  = $size - $share;
printf "%8s %8s %8s\n", qw(Size Shared Diff);
printf "%8d %8d %8d (bytes)\n",$size,$share,$diff;
```

The script opens a opens a connection to the database *'test'* and issues a query to learn what tables the databases has. When the data is collected and printed the connection would be closed in the regular case, but `Apache::DBI` overrides it with empty method. When the data is processed a familiar to you already code to print the memory usage follows.

The server was restarted before each new test.

So here are the results of the five tests that were conducted, sorted by the *Diff* column:

1. After the first request:

```
Test type                            Size   Shared     Diff
-----------------------------------------------------------
install_driver (2)                 3465216  2621440   843776
install_driver & connect_on_init (5) 3461120  2609152   851968
preload driver (3)                 3465216  2605056   860160
nothing added (1)                  3461120  2494464   966656
connect_on_init (4)                3461120  2482176   978944
```

2. After the second request (all the subsequent request showed the same results):

```
Test type                            Size   Shared     Diff
-----------------------------------------------------------
install_driver (2)                 3469312  2609152   860160
install_driver & connect_on_init (5) 3481600  2605056   876544
preload driver (3)                 3469312  2588672   880640
nothing added  (1)                 3477504  2482176   995328
connect_on_init (4)                3481600  2469888  1011712
```

Now what do we conclude from looking at these numbers. First we see that only after a second reload we get the final memory footprint for a specific request in question (if you pass different arguments the memory usage might and will be different).

But both tables show the same pattern of memory usage. We can clearly see that the real winner is the *startup.pl* file's version where the MySQL driver was installed (2). Since we want to have a connection ready for the first request made to the freshly spawned child process, we generally use the version (5) which uses somewhat more memory, but has almost the same number of shared memory pages. The version (3) only preloads the driver which results in smaller shared memory. The last two versions having nothing initialized (1) and having only the connect_on_init() method used (4). The former is a little bit better than the latter, but both significantly worse than the first two versions.

To remind you why do we look for the smallest value in the column *diff*, recall the real memory usage formula:

```
RAM_dedicated_to_mod_perl = diff * number_of_processes
                          + the_processes_with_largest_shared_memory
```

Notice that the smaller the diff is, the bigger the number of processes you can have using the same amount of RAM. Therefore every 100K difference counts, when you multiply it by the number of processes. If we take the number from the version (2) vs. (4) and assume that we have 256M of memory dedicated to mod_perl processes we will get the following numbers using the formula derived from the above formula:

```
             RAM - largest_shared_size
N_of Procs = -------------------------
                       Diff

              268435456 - 2609152
(ver 2)  N =  ------------------- = 309
                    860160

              268435456 - 2469888
(ver 4)  N =  ------------------- = 262
                   1011712
```

So you can tell the difference (17% more child processes in the first version).

14.5.1.6.2 Initializing CGI.pm

`CGI.pm` is a big module that by default postpones the compilation of its methods until they are actually needed, thus making it possible to use it under a slow mod_cgi handler without adding a big overhead. That's not what we want under mod_perl and if you use `CGI.pm` you should precompile the methods that you are going to use at the server startup in addition to preloading the module. Use the compile method for that:

```
use CGI;
CGI->compile(':all');
```

where you should replace the tag group `:all` with the real tags and group tags that you are going to use if you want to optimize the memory usage.

We are going to compare the shared memory foot print by using the script which is back compatible with mod_cgi. You will see that you can improve performance of this kind of scripts as well, but if you really want a fast code think about porting it to use `Apache::Request` for CGI interface and some other module for HTML generation.

So here is the `Apache::Registry` script that we are going to use to make the comparison:

```
preload_cgi_pm.pl
-----------------
use strict;
use CGI ();
use GTop ();

my $q = new CGI;
print $q->header('text/plain');
print join "\n", map {"$_ => ".$q->param($_) } $q->param;
print "\n";

my $proc_mem = GTop->new->proc_mem($$);
my $size  = $proc_mem->size;
my $share = $proc_mem->share;
my $diff  = $size - $share;
printf "%8s %8s %8s\n", qw(Size Shared Diff);
printf "%8d %8d %8d (bytes)\n",$size,$share,$diff;
```

The script initializes the `CGI` object, sends HTTP header and then print all the arguments and values that were passed to the script if at all. At the end as usual we print the memory usage.

As usual we are going to use a single child process, therefore we will use this setting in *httpd.conf*:

```
MinSpareServers 1
MaxSpareServers 1
StartServers 1
MaxClients 1
MaxRequestsPerChild 100
```

We are going to run memory benchmarks on three different versions of the *startup.pl* file. We always preload this module:

```
use Gtop();
```

- **option 1**

  Leave the file unmodified.

- **option 2**

  Preload `CGI.pm`:

```
use CGI ();
```

- **option 3**

  Preload `CGI.pm` and pre-compile the methods that we are going to use in the script:

  ```
  use CGI ();
  CGI->compile(qw(header param));
  ```

The server was restarted before each new test.

So here are the results of the five tests that were conducted, sorted by the *Diff* column:

1. After the first request:

   ```
   Version     Size   Shared     Diff        Test type
   ------------------------------------------------------------------
         1  3321856  2146304  1175552  not preloaded
         2  3321856  2326528   995328  preloaded
         3  3244032  2465792   778240  preloaded & methods+compiled
   ```

2. After the second request (all the subsequent request showed the same results):

   ```
   Version     Size   Shared    Diff         Test type
   ------------------------------------------------------------------
         1  3325952  2134016  1191936 not preloaded
         2  3325952  2314240  1011712 preloaded
         3  3248128  2445312   802816 preloaded & methods+compiled
   ```

The first version shows the results of the script execution when `CGI.pm` wasn't preloaded. The second version with module preloaded. The third when it's both preloaded and the methods that are going to be used are precompiled at the server startup.

By looking at the version one of the second table we can conclude that, preloading adds about 20K of shared size. As we have mention at the beginning of this section that's how `CGI.pm` was implemented--to reduce the load overhead. Which means that preloading CGI is almost hardly change a thing. But if we compare the second and the third versions we will see a very significant difference of 207K (1011712-802816), and we have used only a few methods (the *header* method loads a few more method transparently for a user). Imagine how much memory we are going to save if we are going to precompile all the methods that we are using in other scripts that use `CGI.pm` and do a little bit more than the script that we have used in the test.

But even in our very simple case using the same formula, what do we see? (assuming that we have 256MB dedicated for mod_perl)

```
              RAM - largest_shared_size
N_of Procs = ------------------------
                       Diff

                268435456 - 2134016
(ver 1)  N =  ------------------- = 223
                     1191936

                268435456 - 2445312
(ver 3)  N =  ------------------- = 331
                     802816
```

If we preload `CGI.pm` and precompile a few methods that we use in the test script, we can have 50% more child processes than when we don't preload and precompile the methods that we are going to use.

META: I've heard that the 3.x generation will be less bloated, so probably I'll have to rerun this using the new version.

14.5.2 Increasing Shared Memory With mergemem

`mergemem` is an experimental utility for linux, which looks *very* interesting for us mod_perl users: http://www.complang.tuwien.ac.at/ulrich/mergemem/

It looks like it could be run periodically on your server to find and merge duplicate pages. It won't halt your httpds during the merge, this aspect has been taken into consideration already during the design of mergemem: Merging is not performed with one big systemcall. Instead most operation is in userspace, making a lot of small systemcalls.

Therefore blocking of the system should not happen. And, if it really should turn out to take too much time you can reduce the priority of the process.

The worst case that can happen is this: `mergemem` merges two pages and immediately afterwards they will be split. The split costs about the same as the time consumed by merging.

This software comes with a utility called `memcmp` to tell you how much you might save.

14.5.3 Forking and Executing Subprocesses from mod_perl

It's desirable to avoid forking under mod_perl. Since when you do, you are forking the entire Apache server, lock, stock and barrel. Not only is your Perl code and Perl interpreter being duplicated, but so is mod_ssl, mod_rewrite, mod_log, mod_proxy, mod_speling (it's not a typo!) or whatever modules you have used in your server, all the core routines, etc.

Modern Operating Systems come with a very light version of fork which adds a little overhead when called, since it was optimized to do the absolute minimum of memory pages duplications. The *copy-on-write* technique is the one that allows to do so. The gist of this technique is as follows: the parent process memory pages aren't immediately copied to the child's space on fork(), but this is done only when the child or the parent modifies the data in some memory pages. Before the pages get modified they get marked as dirty and the child has no choice but to copy the pages that are to be modified since they cannot

be shared any more.

If you need to call a Perl program from your mod_perl code, it's better to try to covert the program into a module and call it a function without spawning a special process to do that. Of course if you cannot do that or the program is not written in Perl, you have to call via system() or is equivalent, which spawn a new process. If the program written in C, you may try to write a Perl glue code with help of XS or SWIG architectures, and then the program will be executed as a perl subroutine.

Also by trying to spawn a sub-process, you might be trying to do the *"wrong thing"*. If what you really want is to send information to the browser and then do some post-processing, look into the `PerlCleanupHandler` directive. The latter allows you to tell the child process after request has been processed and user has received the response. This doesn't release the mod_perl process to serve other requests, but it allows to send the response to the client faster. If this is the situation and you need to run some cleanup code, you may want to register this code during the request processing via:

```
my $r = shift;
$r->register_cleanup(\&do_cleanup);
sub do_cleanup{ #some clean-up code here }
```

But when a long term process needs to be spawned, there is not much choice, but to use fork(). We cannot just run this long term process within Apache process, since it'll first keep the Apache process busy, instead of letting it do the job it was designed for. And second, if Apache will be stopped the long term process might be terminated as well, unless coded properly to detach from Apache processes group.

In the following sections we are going to discuss how to properly spawn new processes under mod_perl.

14.5.3.1 Forking a New Process

This is a typical way to call fork() under mod_perl:

```
defined (my $kid = fork) or die "Cannot fork: $!\n";
if ($kid) {
  # Parent runs this block
} else {
  # Child runs this block
  # some code comes here
  CORE::exit(0);
}
# possibly more code here usually run by the parent
```

When using fork(), you should check its return value, since if it returns `undef` it means that the call was unsuccessful and no process was spawned. Something that can happen when the system is running too many processes and cannot spawn new ones.

When the process is successfully forked--the parent receives the PID of the newly spawned child as a returned value of the fork() call and the child receives 0. Now the program splits into two. In the above example the code inside the first block after *if* will be executed by the parent and the code inside the first block after *else* will be executed by the child process.

It's important not to forget to explicitly call exit() at the end of the child code when forking. Since if you don't and there is some code outside the *if/else block*, the child process will execute it as well. But under mod_perl there is another nuance--you must use `CORE::exit()` and not `exit()`, which would be automatically overridden by `Apache::exit()` if used in conjunction with `Apache::Registry` and similar modules. And we want the spawned process to quit when its work is done, otherwise it'll just stay alive use resources and do nothing.

The parent process usually completes its execution path and enters the pool of free servers to wait for a new assignment. If the execution path is to be aborted earlier for some reason one should use Apache::exit() or die(), in the case of `Apache::Registry` or `Apache::PerlRun` handlers a simple exit() will do the right thing.

The child shares with parent its memory pages until it has to modify some of them, which triggers a *copy-on-write* process which copies these pages to the child's domain before the child is allowed to modify them. But this all happens afterwards. At the moment the fork() call executed, the only work to be done before the child process goes on its separate way is setting up the page tables for the virtual memory, which imposes almost no delay at all.

14.5.3.2 Freeing the Parent Process

In the child code you must also close all the pipes to the connection socket that were opened by the parent process (i.e. `STDIN` and `STDOUT`) and inherited by the child, so the parent will be able to complete the request and free itself for serving other requests. If you need the `STDIN` and/or `STDOUT` streams you should re-open them. You may need to close or re-open the `STDERR` filehandle. It's opened to append to the *error_log* file as inherited from its parent, so chances are that you will want to leave it untouched.

Under mod_perl, the spawned process also inherits the file descriptor that's tied to the socket through which all the communications between the server and the client happen. Therefore we need to free this stream in the forked process. If we don't do that, the server cannot be restarted while the spawned process is still running. If an attempt is made to restart the server you will get the following error:

```
[Mon Dec 11 19:04:13 2000] [crit]
(98)Address already in use: make_sock:
  could not bind to address 127.0.0.1 port 8000
```

`Apache::SubProcess` comes to help and provides a method cleanup_for_exec() which takes care of closing this file descriptor.

So the simplest way is to freeing the parent process is to close all three STD* streams if we don't need them and untie the Apache socket. In addition you may want to change process' current directory to */* so the forked process won't keep the mounted partition busy, if this is to be unmounted at a later time. To summarize all this issues, here is an example of the fork that takes care of freeing the parent process.

```
use Apache::SubProcess;
defined (my $kid = fork) or die "Cannot fork: $!\n";
if ($kid) {
  # Parent runs this block
} else {
  # Child runs this block
    $r->cleanup_for_exec(); # untie the socket
```

```
    chdir '/' or die "Can't chdir to /: $!";
    close STDIN;
    close STDOUT;
    close STDERR;

  # some code comes here

    CORE::exit(0);
}
# possibly more code here usually run by the parent
```

Of course between the freeing the parent code and child process termination the real code is to be placed.

14.5.3.3 Detaching the Forked Process

Now what happens if the forked process is running and we decided that we need to restart the web-server? This forked process will be aborted, since when parent process will die during the restart it'll kill its child processes as well. In order to avoid this we need to detach the process from its parent session, by opening a new session with help of setsid() system call, provided by the `POSIX` module:

```
use POSIX 'setsid';

defined (my $kid = fork) or die "Cannot fork: $!\n";
if ($kid) {
  # Parent runs this block
} else {
  # Child runs this block
    setsid or die "Can't start a new session: $!";
    ...
}
```

Now the spawned child process has a life of its own, and it doesn't depend on the parent anymore.

14.5.3.4 Avoiding Zombie Processes

Now let's talk about zombie processes.

Normally, every process has its parent. Many processes are children of the `init` process, whose `PID` is `1`. When you fork a process you must wait() or waitpid() for it to finish. If you don't wait() for it, it becomes a zombie.

A zombie is a process that doesn't have a parent. When the child quits, it reports the termination to its parent. If no parent wait()s to collect the exit status of the child, it gets *"confused"* and becomes a ghost process, that can be seen as a process, but not killed. It will be killed only when you stop the parent process that spawned it!

Generally the ps(1) utility displays these processes with the `<defunc>` tag, and you will see the zombies counter increment when doing top(). These zombie processes can take up system resources and are generally undesirable.

So the proper way to do a fork is:

```
my $r = shift;
$r->send_http_header('text/plain');

defined (my $kid = fork) or die "Cannot fork: $!";
if ($kid) {
  waitpid($kid,0);
  print "Parent has finished\n";
} else {
    # do something
    CORE::exit(0);
}
```

In most cases the only reason you would want to fork is when you need to spawn a process that will take a long time to complete. So if the Apache process that spawns this new child process has to wait for it to finish, you have gained nothing. You can neither wait for its completion (because you don't have the time to), nor continue because you will get yet another zombie process. This is called a blocking call, since the process is blocked to do anything else before this call gets completed.

The simplest solution is to ignore your dead children. Just add this line before the fork() call:

```
$SIG{CHLD} = 'IGNORE';
```

When you set the `CHLD` (`SIGCHLD` in C) signal handler to `'IGNORE'`, all the processes will be collected by the `init` process and are therefore prevented from becoming zombies. This doesn't work everywhere, however. It proved to work at least on Linux OS.

Note that you cannot localize this setting with `local()`. If you do, it won't have the desired effect.

[META: Can anyone explain why localization doesn't work?]

So now the code would look like this:

```
my $r = shift;
$r->send_http_header('text/plain');

$SIG{CHLD} = 'IGNORE';

defined (my $kid = fork) or die "Cannot fork: $!\n";
if ($kid) {
    print "Parent has finished\n";
} else {
    # do something time-consuming
    CORE::exit(0);
}
```

Note that waitpid() call has gone. The $SIG{CHLD} = 'IGNORE'; statement protects us from zombies, as explained above.

Another, more portable, but slightly more expensive solution is to use a double fork approach.

```
my $r = shift;
$r->send_http_header('text/plain');

defined (my $kid = fork) or die "Cannot fork: $!\n";
if ($kid) {
  waitpid($kid,0);
} else {
  defined (my $grandkid = fork) or die "Kid cannot fork: $!\n";
  if ($grandkid) {
    CORE::exit(0);
  } else {
    # code here
    # do something long lasting
    CORE::exit(0);
  }
}
```

Grandkid becomes a *"child of init"*, i.e. the child of the process whose PID is 1.

Note that the previous two solutions do allow you to know the exit status of the process, but in our example we didn't care about it.

Another solution is to use a different *SIGCHLD* handler:

```
use POSIX 'WNOHANG';
$SIG{CHLD} = sub { while( waitpid(-1,WNOHANG)>0 ) {} };
```

Which is useful when you fork() more than one process. The handler could call wait() as well, but for a variety of reasons involving the handling of stopped processes and the rare event in which two children exit at nearly the same moment, the best technique is to call waitpid() in a tight loop with a first argument of `-1` and a second argument of `WNOHANG`. Together these arguments tell waitpid() to reap the next child that's available, and prevent the call from blocking if there happens to be no child ready for reaping. The handler will loop until waitpid() returns a negative number or zero, indicating that no more reapable children remain.

While you test and debug your code that uses one of the above examples, You might want to write some debug information to the error_log file so you know what happens.

Read *perlipc* manpage for more information about signal handlers.

14.5.3.5 A Complete Fork Example

Now let's put all the bits of code together and show a well written fork code that solves all the problems discussed so far. We will use an `Apache::Registry` script for this purpose:

```
proper_fork1.pl
---------------
use strict;
use POSIX 'setsid';
use Apache::SubProcess;
```

```
my $r = shift;
$r->send_http_header("text/plain");

$SIG{CHLD} = 'IGNORE';
defined (my $kid = fork) or die "Cannot fork: $!\n";
if ($kid) {
  print "Parent $$ has finished, kid's PID: $kid\n";
} else {
    $r->cleanup_for_exec(); # untie the socket
    chdir '/'               or die "Can't chdir to /: $!";
    open STDIN, '/dev/null'  or die "Can't read /dev/null: $!";
    open STDOUT, '>/dev/null'
        or die "Can't write to /dev/null: $!";
    open STDERR, '>/tmp/log' or die "Can't write to /tmp/log: $!";
    setsid or die "Can't start a new session: $!";

    my $oldfh = select STDERR;
    local $| = 1;
    select $oldfh;
    warn "started\n";
    # do something time-consuming
    sleep 1, warn "$_\n" for 1..20;
    warn "completed\n";

    CORE::exit(0); # terminate the process
}
```

The script starts with the usual declaration of the strict mode, loading the `POSIX` and `Apache::SubProcess` modules and importing of the setsid() symbol from the `POSIX` package.

The HTTP header is sent next, with the *Content-type* of *text/plain*. The parent process gets ready to ignore the child, to avoid zombies and the fork is called.

The program gets its personality split after fork and the if conditional evaluates to a true value for the parent process, and to a false value for the child process, therefore the first block is executed by the parent and the second by the child.

The parent process announces his PID and the PID of the spawned process and finishes its block. If there will be any code outside it will be executed by the parent as well.

The child process starts its code by disconnecting from the socket, changing its current directory to `/`, opening the STDIN and STDOUT streams to */dev/null*, which in effect closes them both before opening. In fact in this example we don't need neither of these, so we could just close() both. The child process completes its disengagement from the parent process by opening the STDERR stream to */tmp/log*, so it could write there, and creating a new session with help of setsid(). Now the child process has nothing to do with the parent process and can do the actual processing that it has to do. In our example it performs a simple series of warnings, which are logged into */tmp/log*:

```
my $oldfh = select STDERR;
local $| = 1;
select $oldfh;
warn "started\n";
# do something time-consuming
sleep 1, warn "$_\n" for 1..20;
warn "completed\n";
```

The localized setting of `$|=1` unbuffers the STDERR stream, so we can immediately see the debug output generated by the program. In fact this setting is not required when the output is generated by warn().

Finally the child process terminates by calling:

```
CORE::exit(0);
```

which make sure that it won't get out of the block and run some code that it's not supposed to run.

This code example will allow you to verify that indeed the spawned child process has its own life, and its parent is free as well. Simply issue a request that will run this script, watch that the warnings are started to be written into the */tmp/log* file and issue a complete server stop and start. If everything is correct, the server will successfully restart and the long term process will still be running. You will know that it's still running, if the warnings will still be printed into the */tmp/log* file. You may need to raise the number of warnings to do above 20, to make sure that you don't miss the end of the run.

If there are only 5 warnings to be printed, you should see the following output in this file:

```
started
1
2
3
4
5
completed
```

14.5.3.6 Starting a Long Running External Program

But what happens if we cannot just run a Perl code from the spawned process and we have a compiled utility, i.e. a program written in C. Or we have a Perl program which cannot be easily converted into a module, and thus called as a function. Of course in this case we have to use system(), exec(), qx() or `` (back ticks) to start it.

When using any of these methods and when the *Taint* mode is enabled, we must at least add the following code to untaint the *PATH* environment variable and delete a few other insecure environment variables. This information can be found in the *perlsec* manpage.

```
$ENV{'PATH'} = '/bin:/usr/bin';
delete @ENV{'IFS', 'CDPATH', 'ENV', 'BASH_ENV'};
```

Now all we have to do is to reuse the code from the previous section.

First we move the core program into the *external.pl* file, add the shebang first line so the program will be executed by Perl, tell the program to run under *Taint* mode (-T) and possibly enable the *warnings* mode (-w) and make it executable:

```
external.pl
-----------
#!/usr/bin/perl -Tw

open STDIN, '/dev/null'  or die "Can't read /dev/null: $!";
open STDOUT, '>/dev/null'
    or die "Can't write to /dev/null: $!";
open STDERR, '>/tmp/log' or die "Can't write to /tmp/log: $!";

my $oldfh = select STDERR;
local $| = 1;
select $oldfh;
warn "started\n";
# do something time-consuming
sleep 1, warn "$_\n" for 1..20;
warn "completed\n";
```

Now we replace the code that moved into the external program with exec() to call it:

```
proper_fork_exec.pl
-------------------
use strict;
use POSIX 'setsid';
use Apache::SubProcess;

$ENV{'PATH'} = '/bin:/usr/bin';
delete @ENV{'IFS', 'CDPATH', 'ENV', 'BASH_ENV'};

my $r = shift;
$r->send_http_header("text/html");

$SIG{CHLD} = 'IGNORE';

defined (my $kid = fork) or die "Cannot fork: $!\n";
if ($kid) {
  print "Parent has finished, kid's PID: $kid\n";
} else {
    $r->cleanup_for_exec(); # untie the socket
    chdir '/'                or die "Can't chdir to /: $!";
    open STDIN, '/dev/null'  or die "Can't read /dev/null: $!";
    open STDOUT, '>/dev/null'
        or die "Can't write to /dev/null: $!";
    open STDERR, '>&STDOUT'  or die "Can't dup stdout: $!";
    setsid or die "Can't start a new session: $!";

    exec "/home/httpd/perl/external.pl" or die "Cannot execute exec: $!";
}
```

Notice that exec() never returns unless it fails to start the process. Therefore you shouldn't put any code after exec()--it will be not executed in the case of success. Use system() or back-ticks instead if you want to continue doing other things in the process. But then you probably will want to terminate the process

after the program has finished. So you will have to write:

```
system "/home/httpd/perl/external.pl" or die "Cannot execute system: $!";
CORE::exit(0);
```

Another important nuance is that we have to close all STD* stream in the forked process, even if the called program does that.

If the external program is written in Perl you may pass complicated data structures to it using one of the methods to serialize Perl data and then to restore it. The `Storable` and `FreezeThaw` modules come handy. Let's say that we have program *master.pl* calling program *slave.pl*:

```
master.pl
---------
# we are within the mod_perl code
use Storable ();
my @params = (foo => 1, bar => 2);
my $params = Storable::freeze(\@params);
exec "./slave.pl", $params or die "Cannot execute exec: $!";

slave.pl
--------
#!/usr/bin/perl -w
use Storable ();
my @params = @ARGV ? @{ Storable::thaw(shift)||[] } : ();
# do something
```

As you can see, *master.pl* serializes the `@params` data structure with `Storable::freeze` and passes it to *slave.pl* as a single argument. *slave.pl* restores the it with `Storable::thaw`, by shifting the first value of the `ARGV` array if available. The `FreezeThaw` module does a very similar thing.

14.5.3.7 Starting a Short Running External Program

Sometimes you need to call an external program and you cannot continue before this program completes its run and optionally returns some result. In this case the fork solution doesn't help. But we have a few ways to execute this program. First using system():

```
system "perl -e 'print 5+5'"
```

We believe that you will never call the perl interperter for doing this simple calculation, but for the sake of a simple example it's good enough.

The problem with this approach is that we cannot get the results printed to `STDOUT`, and that's where back-ticks or qx() come to help. If you use either:

```
my $result = `perl -e 'print 5+5'`;
```

or:

```
my $result = qx{perl -e 'print 5+5'};
```

the whole output of the external program will be stored in the `$result` variable.

Of course you can use other solutions, like opening a pipe (`|` to the program) if you need to submit many arguments and more evolved solutions provided by other Perl modules like `IPC::Open2` which allows to open a process for both reading and writing.

14.5.3.8 Executing system() or exec() in the Right Way

The exec() and system() system calls behave identically in the way they spawn a program. For example let's use system() as an example. Consider the following code:

```
system("echo","Hi");
```

Perl will use the first argument as a program to execute, find `/bin/echo` along the search path, invoke it directly and pass the *Hi* string as an argument.

Perl's system() is **not** the `system(3)` call [C-library]. This is how the arguments to system() get interpreted. When there is a single argument to system(), it'll be checked for having shell metacharacters first (like `*`,`?`), and if there are any--Perl interpreter invokes a real shell program (/bin/sh -c on Unix platforms). If you pass a list of arguments to system(), they will be not checked for metacharacters, but split into words if required and passed directly to the C-level `execvp()` system call, which is more efficient. That's a *very* nice optimization. In other words, only if you do:

```
system "sh -c 'echo *'"
```

will the operating system actually exec() a copy of `/bin/sh` to parse your command. But even then since *sh* is almost certainly already running somewhere, the system will notice that (via the disk inode reference) and replace your virtual memory page table with one pointing to the existing program code plus your data space, thus will not create this overhead.

14.5.4 OS Specific Parameters for Proxying

Most of the mod_perl enabled servers use a proxy front-end server. This is done in order to avoid serving static objects, and also so that generated output which might be received by slow clients does not cause the heavy but very fast mod_perl servers from idly waiting.

There are very important OS parameters that you might want to change in order to improve the server performance. This topic is discussed in the section: Setting the Buffering Limits on Various OSes

14.6 Performance Tuning by Tweaking Apache Configuration

Correct configuration of the `MinSpareServers`, `MaxSpareServers`, `StartServers`, `MaxClients`, and `MaxRequestsPerChild` parameters is very important. There are no defaults. If they are too low, you will under-use the system's capabilities. If they are too high, the chances are that the

server will bring the machine to its knees.

All the above parameters should be specified on the basis of the resources you have. With a plain apache server, it's no big deal if you run many servers since the processes are about 1Mb and don't eat a lot of your RAM. Generally the numbers are even smaller with memory sharing. The situation is different with mod_perl. I have seen mod_perl processes of 20Mb and more. Now if you have `MaxClients` set to 50: 50x20Mb = 1Gb. Do you have 1Gb of RAM? Maybe not. So how do you tune the parameters? Generally by trying different combinations and benchmarking the server. Again mod_perl processes can be of much smaller size with memory sharing.

Before you start this task you should be armed with the proper weapon. You need the **crashme** utility, which will load your server with the mod_perl scripts you possess. You need it to have the ability to emulate a multiuser environment and to emulate the behavior of multiple clients calling the mod_perl scripts on your server simultaneously. While there are commercial solutions, you can get away with free ones which do the same job. You can use the ApacheBench **`ab`** utility which comes with the Apache distribution, the crashme script which uses `LWP::Parallel::UserAgent`, httperf or http_load.

It is important to make sure that you run the load generator (the client which generates the test requests) on a system that is more powerful than the system being tested. After all we are trying to simulate Internet users, where many users are trying to reach your service at once. Since the number of concurrent users can be quite large, your testing machine must be very powerful and capable of generating a heavy load. Of course you should not run the clients and the server on the same machine. If you do, your test results would be invalid. Clients will eat CPU and memory that should be dedicated to the server, and vice versa.

14.6.1 Configuration Tuning with ApacheBench

We are going to use `ApacheBench` (`ab`) utility to tune our server's configuration. We will simulate 10 users concurrently requesting a very light script at `http://www.example.com/perl/access/access.cgi`. Each simulated user makes 10 requests.

```
% ./ab -n 100 -c 10 http://www.example.com/perl/access/access.cgi
```

The results are:

```
Document Path:          /perl/access/access.cgi
Document Length:        16 bytes

Concurrency Level:      10
Time taken for tests:   1.683 seconds
Complete requests:      100
Failed requests:        0
Total transferred:      16100 bytes
HTML transferred:       1600 bytes
Requests per second:    59.42
Transfer rate:          9.57 kb/s received

Connnection Times (ms)
```

```
              min   avg   max
Connect:        0    29   101
Processing:    77   124  1259
Total:         77   153  1360
```

The only numbers we really care about are:

```
Complete requests:      100
Failed requests:        0
Requests per second:    59.42
```

Let's raise the request load to 100 x 10 (10 users, each makes 100 requests):

```
% ./ab -n 1000 -c 10  http://www.example.com/perl/access/access.cgi
Concurrency Level:      10
Complete requests:      1000
Failed requests:        0
Requests per second:    139.76
```

As expected, nothing changes -- we have the same 10 concurrent users. Now let's raise the number of concurrent users to 50:

```
% ./ab -n 1000 -c 50  http://www.example.com/perl/access/access.cgi
Complete requests:      1000
Failed requests:        0
Requests per second:    133.01
```

We see that the server is capable of serving 50 concurrent users at 133 requests per second! Let's find the upper limit. Using `-n 10000 -c 1000` failed to get results (Broken Pipe?). Using `-n 10000 -c 500` resulted in 94.82 requests per second. The server's performance went down with the high load.

The above tests were performed with the following configuration:

```
MinSpareServers 6
MaxSpareServers 8
StartServers 10
MaxClients 50
MaxRequestsPerChild 1500
```

Now let's kill each child after it serves a single request. We will use the following configuration:

```
MinSpareServers 6
MaxSpareServers 8
StartServers 10
MaxClients 100
MaxRequestsPerChild 1
```

Simulate 50 users each generating a total of 20 requests:

```
% ./ab -n 1000 -c 50  http://www.example.com/perl/access/access.cgi
```

The benchmark timed out with the above configuration.... I watched the output of **ps** as I ran it, the parent process just wasn't capable of respawning the killed children at that rate. When I raised the `MaxRequestsPerChild` to 10, I got 8.34 requests per second. Very bad - 18 times slower! You can't benchmark the importance of the `MinSpareServers`, `MaxSpareServers` and `StartServers` with this kind of test.

Now let's reset `MaxRequestsPerChild` to 1500, but reduce `MaxClients` to 10 and run the same test:

```
MinSpareServers 6
MaxSpareServers 8
StartServers 10
MaxClients 10
MaxRequestsPerChild 1500
```

I got 27.12 requests per second, which is better but still 4-5 times slower. (I got 133 with `MaxClients` set to 50.)

Summary: I have tested a few combinations of the server configuration variables (`MinSpareServers`, `MaxSpareServers`, `StartServers`, `MaxClients` and `MaxRequestsPerChild`). The results I got are as follows:

`MinSpareServers`, `MaxSpareServers` and `StartServers` are only important for user response times. Sometimes users will have to wait a bit.

The important parameters are `MaxClients` and `MaxRequestsPerChild`. `MaxClients` should be not too big, so it will not abuse your machine's memory resources, and not too small, for if it is your users will be forced to wait for the children to become free to serve them. `MaxRequestsPerChild` should be as large as possible, to get the full benefit of mod_perl, but watch your server at the beginning to make sure your scripts are not leaking memory, thereby causing your server (and your service) to die very fast.

Also it is important to understand that we didn't test the response times in the tests above, but the ability of the server to respond under a heavy load of requests. If the test script was heavier, the numbers would be different but the conclusions very similar.

The benchmarks were run with:

```
HW: RS6000, 1Gb RAM
SW: AIX 4.1.5 . mod_perl 1.16, apache 1.3.3
Machine running only mysql, httpd docs and mod_perl servers.
Machine was _completely_ unloaded during the benchmarking.
```

After each server restart when I changed the server's configuration, I made sure that the scripts were preloaded by fetching a script at least once for every child.

It is important to notice that none of the requests timed out, even if it was kept in the server's queue for more than a minute! That is the way **ab** works, which is OK for testing purposes but will be unacceptable in the real world - users will not wait for more than five to ten seconds for a request to complete, and the client (i.e. the browser) will time out in a few minutes.

Now let's take a look at some real code whose execution time is more than a few milliseconds. We will do some real testing and collect the data into tables for easier viewing.

I will use the following abbreviations:

```
NR    = Total Number of Request
NC    = Concurrency
MC    = MaxClients
MRPC  = MaxRequestsPerChild
RPS   = Requests per second
```

Running a mod_perl script with lots of mysql queries (the script under test is mysqld limited) (http://www.example.com/perl/access/access.cgi?do_sub=query_form), with the configuration:

```
MinSpareServers    8
MaxSpareServers   16
StartServers      10
MaxClients        50
MaxRequestsPerChild 5000
```

gives us:

```
    NR   NC    RPS comment
-----------------------------------------------
    10   10    3.33    # not a reliable figure
   100   10    3.94
  1000   10    4.62
  1000   50    4.09
```

Conclusions: Here I wanted to show that when the application is slow (not due to perl loading, code compilation and execution, but limited by some external operation) it almost does not matter what load we place on the server. The RPS (Requests per second) is almost the same. Given that all the requests have been served, you have the ability to queue the clients, but be aware that anything that goes into the queue means a waiting client and a client (browser) that might time out!

Now we will benchmark the same script without using the mysql (code limited by perl only): (http://www.example.com/perl/access/access.cgi), it's the same script but it just returns the HTML form, without making SQL queries.

```
MinSpareServers    8
MaxSpareServers   16
StartServers      10
MaxClients        50
MaxRequestsPerChild 5000

    NR   NC      RPS   comment
-----------------------------------------------
    10   10    26.95   # not a reliable figure
   100   10    30.88
  1000   10    29.31
  1000   50    28.01
  1000  100    29.74
 10000  200    24.92
100000  400    24.95
```

Conclusions: This time the script we executed was pure perl (not limited by I/O or mysql), so we see that the server serves the requests much faster. You can see the number of requests per second is almost the same for any load, but goes lower when the number of concurrent clients goes beyond `MaxClients`. With 25 RPS, the machine simulating a load of 400 concurrent clients will be served in 16 seconds. To be more realistic, assuming a maximum of 100 concurrent clients and 30 requests per second, the client will be served in 3.5 seconds. Pretty good for a highly loaded server.

Now we will use the server to its full capacity, by keeping all `MaxClients` clients alive all the time and having a big `MaxRequestsPerChild`, so that no child will be killed during the benchmarking.

```
MinSpareServers   50
MaxSpareServers   50
StartServers      50
MaxClients        50
MaxRequestsPerChild 5000

   NR   NC      RPS   comment
-----------------------------------------------
  100   10    32.05
 1000   10    33.14
 1000   50    33.17
 1000  100    31.72
10000  200    31.60
```

Conclusion: In this scenario there is no overhead involving the parent server loading new children, all the servers are available, and the only bottleneck is contention for the CPU.

Now we will change `MaxClients` and watch the results: Let's reduce `MaxClients` to 10.

```
MinSpareServers    8
MaxSpareServers   10
StartServers      10
MaxClients        10
MaxRequestsPerChild 5000

   NR   NC      RPS   comment
-----------------------------------------------
   10   10    23.87   # not a reliable figure
  100   10    32.64
 1000   10    32.82
 1000   50    30.43
 1000  100    25.68
 1000  500    26.95
 2000  500    32.53
```

Conclusions: Very little difference! Ten servers were able to serve almost with the same throughput as 50 servers. Why? My guess is because of CPU throttling. It seems that 10 servers were serving requests 5 times faster than when we worked with 50 servers. In that case, each child received its CPU time slice five times less frequently. So having a big value for `MaxClients`, doesn't mean that the performance will be better. You have just seen the numbers!

Now we will start drastically to reduce `MaxRequestsPerChild`:

```
MinSpareServers    8
MaxSpareServers   16
StartServers      10
MaxClients        50

   NR   NC    MRPC     RPS    comment
-----------------------------------------------
  100   10      10    5.77
  100   10       5    3.32
 1000   50      20    8.92
 1000   50      10    5.47
 1000   50       5    2.83
 1000  100      10    6.51
```

Conclusions: When we drastically reduce `MaxRequestsPerChild`, the performance starts to become closer to plain mod_cgi.

Here are the numbers of this run with mod_cgi, for comparison:

```
MinSpareServers    8
MaxSpareServers   16
StartServers      10
MaxClients        50

   NR   NC    RPS comment
-----------------------------------------------
  100   10    1.12
 1000   50    1.14
 1000  100    1.13
```

Conclusion: mod_cgi is much slower. :) In the first test, when NR/NC was 100/10, mod_cgi was capable of 1.12 requests per second. In the same circumstances, mod_perl was capable of 32 requests per second, nearly 30 times faster! In the first test each client waited about 100 seconds to be served. In the second and third tests they waited 1000 seconds!

14.6.2 Choosing MaxClients

The `MaxClients` directive sets the limit on the number of simultaneous requests that can be supported. No more than this number of child server processes will be created. To configure more than 256 clients, you must edit the `HARD_SERVER_LIMIT` entry in `httpd.h` and recompile. In our case we want this variable to be as small as possible, because in this way we can limit the resources used by the server children. Since we can restrict each child's process size (see Preventing Your Processes from Growing), the calculation of `MaxClients` is pretty straightforward:

```
              Total RAM Dedicated to the Webserver
MaxClients = ------------------------------------
                    MAX child's process size
```

So if I have 400Mb left for the webserver to run with, I can set `MaxClients` to be of 40 if I know that each child is limited to 10Mb of memory (e.g. with `Apache::SizeLimit`).

You will be wondering what will happen to your server if there are more concurrent users than `MaxClients` at any time. This situation is signified by the following warning message in the `error_log`:

```
[Sun Jan 24 12:05:32 1999] [error] server reached MaxClients setting,
consider raising the MaxClients setting
```

There is no problem -- any connection attempts over the `MaxClients` limit will normally be queued, up to a number based on the `ListenBacklog` directive. When a child process is freed at the end of a different request, the connection will be served.

It **is an error** because clients are being put in the queue rather than getting served immediately, despite the fact that they do not get an error response. The error can be allowed to persist to balance available system resources and response time, but sooner or later you will need to get more RAM so you can start more child processes. The best approach is to try not to have this condition reached at all, and if you reach it often you should start to worry about it.

It's important to understand how much real memory a child occupies. Your children can share memory between them when the OS supports that. You must take action to allow the sharing to happen - See Preload Perl modules at server startup. If you do this, the chances are that your `MaxClients` can be even higher. But it seems that it's not so simple to calculate the absolute number. If you come up with a solution please let us know! If the shared memory was of the same size throughout the child's life, we could derive a much better formula:

$$\text{MaxClients} = \frac{\text{Total_RAM} + \text{Shared_RAM_per_Child} * (\text{MaxClients} - 1)}{\text{Max_Process_Size}}$$

which is:

$$\text{MaxClients} = \frac{\text{Total_RAM} - \text{Shared_RAM_per_Child}}{\text{Max_Process_Size} - \text{Shared_RAM_per_Child}}$$

Let's roll some calculations:

```
Total_RAM            = 500Mb
Max_Process_Size     =  10Mb
Shared_RAM_per_Child =   4Mb

             500 - 4
MaxClients = --------- = 82
              10 - 4
```

With no sharing in place

```
               500
MaxClients = --------- = 50
               10
```

With sharing in place you can have 64% more servers without buying more RAM.

If you improve sharing and keep the sharing level, let's say:

```
Total_RAM            = 500Mb
Max_Process_Size     =  10Mb
Shared_RAM_per_Child =   8Mb

             500 - 8
MaxClients = --------- = 246
              10 - 8
```

392% more servers! Now you can feel the importance of having as much shared memory as possible.

14.6.3 Choosing MaxRequestsPerChild

The `MaxRequestsPerChild` directive sets the limit on the number of requests that an individual child server process will handle. After `MaxRequestsPerChild` requests, the child process will die. If `MaxRequestsPerChild` is 0, then the process will live forever.

Setting `MaxRequestsPerChild` to a non-zero limit solves some memory leakage problems caused by sloppy programming practices, whereas a child process consumes more memory after each request.

If left unbounded, then after a certain number of requests the children will use up all the available memory and leave the server to die from memory starvation. Note that sometimes standard system libraries leak memory too, especially on OSes with bad memory management (e.g. Solaris 2.5 on x86 arch).

If this is your case you can set `MaxRequestsPerChild` to a small number. This will allow the system to reclaim the memory that a greedy child process consumed, when it exits after `MaxRequestsPerChild` requests.

But beware -- if you set this number too low, you will lose some of the speed bonus you get from mod_perl. Consider using `Apache::PerlRun` if this is the case.

Another approach is to use the Apache::SizeLimit or Apache::GTopLimit modules. By using either of these modules you should be able to discontinue using the `MaxRequestPerChild`, although for some developers, using both in combination does the job. In addition these modules allow you to kill httpd processes whose shared memory size drops below a specified limit or unshared memory size crosses a specified threshold.

See also Preload Perl modules at server startup and Sharing Memory.

14.6.4 Choosing MinSpareServers, MaxSpareServers and StartServers

With mod_perl enabled, it might take as much as 20 seconds from the time you start the server until it is ready to serve incoming requests. This delay depends on the OS, the number of preloaded modules and the process load of the machine. It's best to set `StartServers` and `MinSpareServers` to high numbers, so that if you get a high load just after the server has been restarted the fresh servers will be ready to serve requests immediately. With mod_perl, it's usually a good idea to raise all 3 variables higher than normal.

In order to maximize the benefits of mod_perl, you don't want to kill servers when they are idle, rather you want them to stay up and available to handle new requests immediately. I think an ideal configuration is to set `MinSpareServers` and `MaxSpareServers` to similar values, maybe even the same. Having the `MaxSpareServers` close to `MaxClients` will completely use all of your resources (if `MaxClients` has been chosen to take the full advantage of the resources), but it'll make sure that at any given moment your system will be capable of responding to requests with the maximum speed (assuming that number of concurrent requests is not higher than `MaxClients`).

Let's try some numbers. For a heavily loaded web site and a dedicated machine I would think of (note 400Mb is just for example):

```
Available to webserver RAM:   400Mb
Child's memory size bounded:  10Mb
MaxClients:           400/10 = 40 (larger with mem sharing)
StartServers:         20
MinSpareServers:      20
MaxSpareServers:      35
```

However if I want to use the server for many other tasks, but make it capable of handling a high load, I'd think of:

```
Available to webserver RAM:   400Mb
Child's memory size bounded:  10Mb
MaxClients:           400/10 = 40
StartServers:         5
MinSpareServers:      5
MaxSpareServers:      10
```

These numbers are taken off the top of my head, and shouldn't be used as a rule, but rather as examples to show you some possible scenarios. Use this information with caution!

14.6.5 Summary of Benchmarking to tune all 5 parameters

OK, we've run various benchmarks -- let's summarize the conclusions:

- **MaxRequestsPerChild**

  If your scripts are clean and don't leak memory, set this variable to a number as large as possible (10000?). If you use `Apache::SizeLimit` or `Apache::GTopLimit`, you can set this parameter to 0 (treated as infinity).

- **StartServers**

  If you keep a small number of servers active most of the time, keep this number low. Keep it low especially if `MaxSpareServers` is also low, as if there is no load Apache will kill its children before they have been utilized at all. If your service is heavily loaded, make this number close to `MaxClients`, and keep `MaxSpareServers` equal to `MaxClients`.

- **MinSpareServers**

  If your server performs other work besides web serving, make this low so the memory of unused children will be freed when the load is light. If your server's load varies (you get loads in bursts) and you want fast response for all clients at any time, you will want to make it high, so that new children will be respawned in advance and are waiting to handle bursts of requests.

- **MaxSpareServers**

  The logic is the same as for `MinSpareServers` - low if you need the machine for other tasks, high if it's a dedicated web host and you want a minimal delay between the request and the response.

- **MaxClients**

  Not too low, so you don't get into a situation where clients are waiting for the server to start serving them (they might wait, but not for very long). However, do not set it too high. With a high MaxClients, if you get a high load the server will try to serve all requests immediately. Your CPU will have a hard time keeping up, and if the child size * number of running children is larger than the total available RAM your server will start swapping. This will slow down everything, which in turn will make things even slower, until eventually your machine will die. It's important that you take pains to ensure that swapping does not normally happen. Swap space is an emergency pool, not a resource to be used routinely. If you are low on memory and you badly need it, buy it. Memory is cheap.

  But based on the test I conducted above, even if you have plenty of memory like I have (1Gb), increasing `MaxClients` sometimes will give you no improvement in performance. The more clients are running, the more CPU time will be required, the less CPU time slices each process will receive. The response latency (the time to respond to a request) will grow, so you won't see the expected improvement. The best approach is to find the minimum requirement for your kind of service and the maximum capability of your machine. Then start at the minimum and test like I did, successively raising this parameter until you find the region on the curve of the graph of latency and/or throughput against MaxClients where the improvement starts to diminish. Stop there and use it. When you make the measurements on a production server you will have the ability to tune them more precisely, since you will see the real numbers.

  Don't forget that if you add more scripts, or even just modify the existing ones, the processes will grow in size as you compile in more code. Probably the parameters will need to be recalculated.

14.6.6 KeepAlive

If your mod_perl server's *httpd.conf* includes the following directives:

```
KeepAlive On
MaxKeepAliveRequests 100
KeepAliveTimeout 15
```

you have a real performance penalty, since after completing the processing for each request, the process will wait for `KeepAliveTimeout` seconds before closing the connection and will therefore not be serving other requests during this time. With this configuration you will need many more concurrent processes on a server with high traffic.

If you use some server status reporting tools, you will see the process in *K* status when it's in `KeepAlive` status.

The chances are that you don't want this feature enabled. Set it Off with:

```
KeepAlive Off
```

the other two directives don't matter if `KeepAlive` is `Off`.

You might want to consider enabling this option if the client's browser needs to request more than one object from your server for a single HTML page. If this is the situation the by setting `KeepAlive On` then for each page you save the HTTP connection overhead for all requests but the first one.

For example if you have a page with 10 ad banners, which is not uncommon today, you server will work more effectively if a single process serves them all during a single connection. However, your client will see a slightly slower response, since banners will be brought one at a time and not concurrently as is the case if each `IMG` tag opens a separate connection.

Since keepalive connections will not incur the additional three-way TCP handshake they are kinder to the network.

SSL connections benefit the most from `KeepAlive` in case you didn't configure the server to cache session ids.

You have probably followed the advice to send all the requests for static objects to a plain Apache server. Since most pages include more than one unique static image, you should keep the default `KeepAlive` setting of the non-mod_perl server, i.e. keep it `On`. It will probably be a good idea also to reduce the timeout a little.

One option would be for the proxy/accelerator to keep the connection open to the client but make individual connections to the server, read the response, buffer it for sending to the client and close the server connection. Obviously you would make new connections to the server as required by the client's requests.

14.6.7 PerlSetupEnv Off

`PerlSetupEnv Off` is another optimization you might consider. This directive requires mod_perl 1.25 or later.

When this option is enabled, *mod_perl* fiddles with the environment to make it appear as if the code is called under the mod_cgi handler. For example, the `$ENV{QUERY_STRING}` environment variable is initialized with the contents of *Apache::args()*, and the value returned by *Apache::server_hostname()* is put into `$ENV{SERVER_NAME}`.

But `%ENV` population is expensive. Those who have moved to the Perl Apache API no longer need this extra `%ENV` population, and can gain by turning it `Off`. Scripts using the `CGI.pm` module require `PerlSetupEnv On` because that module relies on a properly populated CGI environment table.

By default it is turned `On`.

Note that you can still set enviroment variables when `PerlSetupEnv` is turned `Off`. For example when you use the following configuration:

```
PerlSetupEnv Off
PerlModule Apache::RegistryNG
<Location /perl>
  PerlSetEnv TEST hi
  SetHandler perl-script
  PerlHandler Apache::RegistryNG
  Options +ExecCGI
</Location>
```

and you issue a request for this script:

```
setupenvoff.pl
--------------
use Data::Dumper;
my $r = Apache->request();
$r->send_http_header('text/plain');
print Dumper(\%ENV);
```

you should see something like this:

```
$VAR1 = {
          'GATEWAY_INTERFACE' => 'CGI-Perl/1.1',
          'MOD_PERL' => 'mod_perl/1.25',
          'PATH' => '/usr/lib/perl5/5.00503:... snipped ...',
          'TEST' => 'hi'
        };
```

Note that we got the value of the *TEST* environment variable we set in *httpd.conf*.

14.6.8 Reducing the Number of stat() Calls Made by Apache

If you watch the system calls that your server makes (using *truss* or *strace* while processing a request, you will notice that a few stat() calls are made. For example when I fetch http://localhost/perl-status and I have my DocRoot set to */home/httpd/docs* I see:

```
[snip]
stat("/home/httpd/docs/perl-status", 0xbffff8cc) = -1
                    ENOENT (No such file or directory)
stat("/home/httpd/docs", {st_mode=S_IFDIR|0755,
                              st_size=1024, ...}) = 0
[snip]
```

If you have some dynamic content and your virtual relative URI is something like */news/perl/mod_perl/summary* (i.e., there is no such directory on the web server, the path components are only used for requesting a specific report), this will generate five(!) stat() calls, before the `DocumentRoot` is found. You will see something like this:

```
stat("/home/httpd/docs/news/perl/mod_perl/summary", 0xbffff744) = -1
                    ENOENT (No such file or directory)
stat("/home/httpd/docs/news/perl/mod_perl",         0xbffff744) = -1
                    ENOENT (No such file or directory)
stat("/home/httpd/docs/news/perl",                  0xbffff744) = -1
                    ENOENT (No such file or directory)
stat("/home/httpd/docs/news",                       0xbffff744) = -1
                    ENOENT (No such file or directory)
stat("/home/httpd/docs",
                    {st_mode=S_IFDIR|0755, st_size=1024, ...})  =  0
```

How expensive those calls are? Let's use the `Time::HiRes` module to find out.

```
stat_call_sample.pl
-------------------
use Time::HiRes qw(gettimeofday tv_interval);
my $calls = 1_000_000;

my $start_time = [ gettimeofday ];

stat "/foo" for 1..$calls;

my $end_time = [ gettimeofday ];

my $elapsed = tv_interval($start_time,$end_time) / $calls;

print "The average execution time: $elapsed seconds\n";
```

This script takes a time sample at the beginnig, then does 1_000_000 `stat()` calls to a non-existing file, samples the time at the end and prints the average time it took to make a single `stat()` call. I'm sampling a 1M stats, so I'd get a correct average result.

Before we actually run the script one should distinguish between two different situation. When the server is idle the time between the first and the last system call will be much shorter than the same time measured on the loaded system. That is because on the idle system, a process can use CPU very often, and on the

loaded system lots of processes compete over it and each process has to wait for a longer time to get the same amount of CPU time.

So first we run the above code on the unloaded system:

```
% perl stat_call_sample.pl
The average execution time: 4.209645e-06 seconds
```

So it takes about 4 microseconds to execute a stat() call. Now let start a CPU intensive process in one console. The following code keeps CPU busy all the time.

```
% perl -e '1**1 while 1'
```

And now run the *stat_call_sample.pl* script in the other console.

```
% perl stat_call_sample.pl
The average execution time: 8.777301e-06 seconds
```

You can see that the average time has doubled (about 8 microseconds). And this is obvious, since there were two processes competing over CPU. Now if run 4 occurrences of the above code:

```
% perl -e '1**1 while 1' &
% perl -e '1**1 while 1' &
% perl -e '1**1 while 1' &
% perl -e '1**1 while 1' &
```

And when running our script in parallel with these processes, we get:

```
% perl stat_call_sample.pl
2.0853558e-05 seconds
```

about 20 microseconds. So the average stat() system call is 5 times longer now. Now if you have 50 mod_perl processes that keep the CPU busy all the time, the stat() call will be 50 times slower and it'll take 0.2 milliseconds to complete a series of call. If you have five redundant calls as in the strace example above, they adds up to one millisecond. If you have more processes constantly consuming CPU, this time adds up. Now multiply this time by the number of processes that you have and you get a few seconds lost. As usual, for some services this loss is insignificant, while for others a very significant one.

So why Apache does all these redundant `stat()` calls? You can blame the default installed `TransHandler` for this inefficiency. Of course you could supply your own, which will be smart enough not to look for this virtual path and immediately return `OK`. But in cases where you have a virtual host that serves only dynamically generated documents, you can override the default `PerlTransHandler` with this one:

```
PerlModule Apache::Constants
<VirtualHost 10.10.10.10:80>
  ...
  PerlTransHandler  Apache::Constants::OK
  ...
</VirtualHost>
```

As you see it affects only this specific virtual host.

This has the effect of short circuiting the normal `TransHandler` processing of trying to find a filesystem component that matches the given URI -- no more 'stat's!

Watching your server under strace/truss can often reveal more performance hits than trying to optimize the code itself!

For example unless configured correctly, Apache might look for the *.htaccess* file in many places, if you don't have one and add many open() calls.

Let's start with this simple configuration, and will try to reduce the number of irrelevant system calls.

```
DocumentRoot "/home/httpd/docs"
<Location /foo/test>
  SetHandler perl-script
  PerlHandler Apache::Foo
</Location>
```

The above configuration allows us to make a request to */foo/test* and the Perl handler() defined in `Apache::Foo` will be executed. Notice that in the test setup there is no file to be executed (like in `Apache::Registry`). There is no *.htaccess* file as well.

This is a typical generated trace.

```
stat("/home/httpd/docs/foo/test", 0xbffff8fc) = -1 ENOENT
  (No such file or directory)
stat("/home/httpd/docs/foo",      0xbffff8fc) = -1 ENOENT
  (No such file or directory)
stat("/home/httpd/docs",
  {st_mode=S_IFDIR|0755, st_size=1024, ...}) = 0
open("/.htaccess", O_RDONLY)                 = -1 ENOENT
  (No such file or directory)
open("/home/.htaccess", O_RDONLY)            = -1 ENOENT
  (No such file or directory)
open("/home/httpd/.htaccess", O_RDONLY)      = -1 ENOENT
  (No such file or directory)
open("/home/httpd/docs/.htaccess", O_RDONLY) = -1 ENOENT
  (No such file or directory)
stat("/home/httpd/docs/test", 0xbffff774)    = -1 ENOENT
  (No such file or directory)
stat("/home/httpd/docs",
  {st_mode=S_IFDIR|0755, st_size=1024, ...}) = 0
```

Now we modify the `<Directory>` entry and add AllowOverride None, which among other things disables *.htaccess* files and will not try to open them.

```
<Directory />
  AllowOverride None
</Directory>
```

We see that the four open() calls for *.htaccess* have gone.

```
stat("/home/httpd/docs/foo/test", 0xbffff8fc) = -1 ENOENT
  (No such file or directory)
stat("/home/httpd/docs/foo",      0xbffff8fc) = -1 ENOENT
  (No such file or directory)
stat("/home/httpd/docs",
  {st_mode=S_IFDIR|0755, st_size=1024, ...}) = 0
stat("/home/httpd/docs/test", 0xbffff774)    = -1 ENOENT
  (No such file or directory)
stat("/home/httpd/docs",
  {st_mode=S_IFDIR|0755, st_size=1024, ...}) = 0
```

Let's try to shortcut the *foo* location with:

```
Alias /foo /
```

Which makes Apache to look for the file in the / directory and not under */home/httpd/docs/foo*. Let's run it:

```
stat("//test", 0xbffff8fc) = -1 ENOENT (No such file or directory)
```

Wow, we've got only one stat call left!

Let's remove the last `Alias` setting and use:

```
  PerlModule Apache::Constants
  PerlTransHandler  Apache::Constants::OK
```

as explained above. When we issue the request, we see no stat() calls. But this is possible only if you serve only dynamically generated documents, i.e. no CGI scripts. Otherwise you will have to write your own *PerlTransHandler* to handle requests as desired.

For example this *PerlTransHandler* will not lookup the file on the filesystem if the URI starts with */foo*, but will use the default *PerlTransHandler* otherwise:

```
PerlTransHandler 'sub { return shift->uri() =~ m|^/foo| \
                     ? Apache::Constants::OK \
                     : Apache::Constants::DECLINED; }'
```

Let's see the same configuration using the `<Perl>` section and a dedicated package:

```
<Perl>
  package My::Trans;
  use Apache::Constants qw(:common);
  sub handler{
     my $r = shift;
     return OK if $r->uri() =~ m|^/foo|;
     return DECLINED;
  }

  package Apache::ReadConfig;
  $PerlTransHandler = "My::Trans";
</Perl>
```

As you see we have defined the `My::Trans` package and implemented the handler() function. Then we have assigned this handler to the `PerlTransHandler`.

Of course you can move the code in the module into an external file, (e.g. *My/Trans.pm*) and configure the `PerlTransHandler` with

```
PerlTransHandler My::Trans
```

in the normal way (no `<Perl>` section required).

There is an even simpler way to save that last `stat()` call. Instead of using `PerlTransHandler` combined with:

```
Alias /foo /
```

we can use:

```
AliasMatch ^/foo /
```

which in the current implementation (at least in apache-1.3.28) doesn't incur the `stat()` call. Using the regex instead of prefix matching might slow things a bit, but is probably still faster than the `stat()` call.

14.7 TMTOWTDI: Convenience and Habit vs. Performance

TMTOWTDI (sometimes pronounced *"tim toady"*), or *"There's More Than One Way To Do It"* is the main motto of Perl. In other words, you can gain the same goal by coding in many different styles, using different modules and deploying the same modules in different ways.

Unfortunately when you come to the point where performance is the goal, you might have to learn what's more efficient and what's not. Of course it might mean that you will have to use something that you don't really like, it might be less convenient or it might be just a matter of habit that one should change.

So this section is about performance trade-offs. For almost each comparison we will provide the theoretical difference and then run benchmarks to support the theory, since however good the theory its the numbers we get in practice that matter.

"Premature optimizations are evil", the saying goes. I believe that knowing how to write an efficient code in first place, where it doesn't make the quality and clarity suffer saves time in the long run. That's what this section is mostly about.

In the following benchmarks, unless told different the following Apache configuration has been used:

```
MinSpareServers 10
MaxSpareServers 20
StartServers 10
MaxClients 20
MaxRequestsPerChild 10000
```

14.7.1 Apache::Registry PerlHandler vs. Custom PerlHandler

At some point you have to decide whether to use `Apache::Registry` and similar handlers and stick to writing scripts for the content generation or to write pure Perl handlers.

`Apache::Registry` maps a request to a file and generates a subroutine to run the code contained in that file. If you use a PerlHandler My::Handler instead of `Apache::Registry`, you have a direct mapping from request to subroutine, without the steps in between. These steps include:

1. run the stat() on the script's filename ($r->filename)
2. check that the file exists and is executable
3. generate a Perl package name based on the request's URI ($r->uri)
4. go to the directory the script resides in (chdir basename $r->filename)
5. compare the file's and stored in memory compiled subroutine's last modified time (if it was compiled already)
6. if modified or not compiled, compile the subroutine
7. go back to the previous directory (chdir $old_cwd)

If you cut out those steps, you cut out some overhead, plain and simple. Do you *need* to cut out that overhead? May be yes, may be not. Your requirements determine that.

You should take a look at the sister `Apache::Registry` modules (e.g. `Apache::RegistryNG` and `Apache::RegistryBB`) that don't perform all these steps, so you can still choose to stick to using scripts to generate the content. The greatest added value of scripts is that you don't have to modify the configuration file to add the handler configuration and restarting the server for each newly written content handler.

Now let's run benchmarks and compare.

We want to see the overhead that `Apache::Registry` adds compared to the custom handler and whether it becomes insignificant when used for the heavy and time consuming code. In order to do that we will run two benchmarks sets: the first so called a *light* set will use an almost empty script, that only sends a basic header and one word as content; the second will be a *heavy* set which will add some time consuming operation to the script's and the handler's code.

For the *light* set we are going to use the *registry.pl* script running under `Apache::Registry`:

```
benchmarks/registry.pl
----------------------
use strict;
print "Content-type: text/plain\r\n\r\n";
print "Hello";
```

And the following content generation handler:

```
Benchmark/Handler.pm
--------------------
package Benchmark::Handler;
use Apache::Constants qw(:common);

sub handler{
  $r = shift;
  $r->send_http_header('text/html');
  $r->print("Hello");
  return OK;
}
1;
```

We will add this settings to *httpd.conf*:

```
PerlModule Benchmark::Handler
<Location /benchmark_handler>
  SetHandler perl-script
  PerlHandler Benchmark::Handler
</Location>
```

The first directive worries to preload and compile the `Benchmark::Handler` module. The rest of the lines tell Apache to execute the subroutine `Benchmark::Handler::handler` when a request with relative URI */benchmark_handler* is made.

We will use the usual configuration for `Apache::Registry` scripts, where all the URIs starting with */perl* are remapped to the files residing under */home/httpd/perl/* directory.

```
Alias /perl/ /home/httpd/perl/
<Location /perl>
  SetHandler perl-script
  PerlHandler +Apache::Registry
  Options ExecCGI
  PerlSendHeader On
</Location>
```

We will use the `Apache::RegistryLoader` to preload and compile the script at the server startup as well, so the benchmark will be fair through the benchmark and only the processing time will be measured. To accomplish the preloading we add the following code to the *startup.pl* file:

```
use Apache::RegistryLoader ();
Apache::RegistryLoader->new->handler(
            "/perl/benchmarks/registry.pl",
 "/home/httpd/perl/benchmarks/registry.pl");
```

To create the *heavy* benchmark set let's leave the above code examples unmodified but add some CPU intensive processing operation (it can be also an IO operation or a database query.)

```
my $x = 100;
my $y = log ($x ** 100)  for (0..10000);
```

This code does lots of mathematical processing and therefore very CPU intensive.

Now we are ready to proceed with the benchmark. We will generate 5000 requests with 15 as a concurrency level using the `Apache::Benchmark` module.

Here are the reported results:

```
  ------------------------------
      name        | avtime   rps
  ------------------------------
  light handler   |     15   911
  light registry  |     21   680
  ------------------------------
  heavy handler   |    183    81
  heavy registry  |    191    77
  ------------------------------
```

Let's look at the results and answer the previously asked questions.

First let's compare the results from the *light* set. We can see that the average overhead added by `Apache::Registry` (compared to the custom handler) is about:

```
  21 - 15 = 6 milliseconds
```

per request.

Thus the difference in speed is about 40% (15 vs. 21). Note that this doesn't mean that the difference in the real world applications is such big. And the results of the *heavy* set confirm that.

In the *heavy* set the average processing time is almost the same for the `Apache::Registry` and the custom handler. You can clearly see that the difference between the two is almost the same one that we have seen in the *light* set's results. It has grown from 6 milliseconds to 8 milliseconds (191-183). Which means that the identical heavy code that has been added was running for about 168 milliseconds (183-15). It doesn't mean that the added code itself has been running for 168 milliseconds. It means that it took 168 milliseconds for this code to be completed in a multi-process environment where each process gets a time slice to use the CPU. The more processes are running the more time the process will have to wait to get the next time slice when it can use the CPU.

We have the second question answered as well. You can see that when the code is not just the *hello* script, the overhead of the extra operations done but the `Apache::Registry` module, is almost insignificant. It's a non zero though, so it depends on your requirements, and if another 5-10 millisecons overhead are quite tolerable, you may choose to use `Apache::Registry`.

The interesting thing is that when the server under test runs on a very slow machine the results are completely different. I'll present them here for comparison:

```
------------------------------
    name        | avtime   rps
------------------------------
light handler   |     50   196
light registry  |    160    61
------------------------------
heavy handler   |    149    67
heavy registry  |    822    12
------------------------------
```

First of all the difference of 6 milliseconds in the average processing time we have seen on the fast machine when running the *light* set, now has grown to 110 milliseconds. Which means that a few extra operations, that `Apache::Registry` does, turn to be very expensive on the slow machine.

Second, you can see that when the *heavy* set is used, there is no preservation of the 110 milliseconds as we have seen on the fast machine, which we obviously would expect to see, since the code that was added should take the same time to execute in the handler and the script. But instead we see a difference of 673 milliseconds (822-149).

The explanation lies in fact that the difference between the machines isn't merely in the CPU speed. It's possible that there are many other things that are different. For example the size of the processor cache. If one machine has a processor cache large enough to hold the whole handler and the other doesn't this can be very significant, given that in our *heavy* benchmark set, 99.9% of the CPU activity was dedicated to running the calculation code.

But this also shows you again, that none of the results and conclusion made here should be taken for granted. Certainly, most chances are that you will see a similar behavior on your machine, but only after you have run the benchmarks and analyzed the received results, you can be sure what is the best for you using the setup under test. If you later you happen to use a different machine, make sure to run the tests again, as they can lead to complete different decision as we have just seen when we have tried the same benchmark on a different machine.

14.7.2 "Bloatware" modules

Perl modules like IO:: are very convenient, but let's see what it costs us to use them. (perl5.6.0 over OpenBSD)

```
% wc ‘perl -MIO -e ’print join("\n", sort values %INC, "")’‘
 124     696    4166 /usr/local/lib/perl5/5.6.0/Carp.pm
 580    2465   17661 /usr/local/lib/perl5/5.6.0/Class/Struct.pm
 400    1495   10455 /usr/local/lib/perl5/5.6.0/Cwd.pm
 313    1589   10377 /usr/local/lib/perl5/5.6.0/Exporter.pm
 225     784    5651 /usr/local/lib/perl5/5.6.0/Exporter/Heavy.pm
  92     339    2813 /usr/local/lib/perl5/5.6.0/File/Spec.pm
 442    1574   10276 /usr/local/lib/perl5/5.6.0/File/Spec/Unix.pm
 115     398    2806 /usr/local/lib/perl5/5.6.0/File/stat.pm
 406    1350   10265 /usr/local/lib/perl5/5.6.0/IO/Socket/INET.pm
 143     429    3075 /usr/local/lib/perl5/5.6.0/IO/Socket/UNIX.pm
7168   24137  178650 /usr/local/lib/perl5/5.6.0/OpenBSD.i386-openbsd/Config.pm
 230    1052    5995 /usr/local/lib/perl5/5.6.0/OpenBSD.i386-openbsd/Errno.pm
 222     725    5216 /usr/local/lib/perl5/5.6.0/OpenBSD.i386-openbsd/Fcntl.pm
```

```
   47     101     669 /usr/local/lib/perl5/5.6.0/OpenBSD.i386-openbsd/IO.pm
  239     769    5005 /usr/local/lib/perl5/5.6.0/OpenBSD.i386-openbsd/IO/Dir.pm
  169     549    3956 /usr/local/lib/perl5/5.6.0/OpenBSD.i386-openbsd/IO/File.pm
  594    2180   14772 /usr/local/lib/perl5/5.6.0/OpenBSD.i386-openbsd/IO/Handle.pm
  252     755    5375 /usr/local/lib/perl5/5.6.0/OpenBSD.i386-openbsd/IO/Pipe.pm
   77     235    1709 /usr/local/lib/perl5/5.6.0/OpenBSD.i386-openbsd/IO/Seekable.pm
  428    1419   10219 /usr/local/lib/perl5/5.6.0/OpenBSD.i386-openbsd/IO/Socket.pm
  452    1401   10554 /usr/local/lib/perl5/5.6.0/OpenBSD.i386-openbsd/Socket.pm
  127     473    3554 /usr/local/lib/perl5/5.6.0/OpenBSD.i386-openbsd/XSLoader.pm
   52     161    1050 /usr/local/lib/perl5/5.6.0/SelectSaver.pm
  139     541    3754 /usr/local/lib/perl5/5.6.0/Symbol.pm
  161     609    4081 /usr/local/lib/perl5/5.6.0/Tie/Hash.pm
  109     390    2479 /usr/local/lib/perl5/5.6.0/strict.pm
   79     370    2589 /usr/local/lib/perl5/5.6.0/vars.pm
  318    1124   11975 /usr/local/lib/perl5/5.6.0/warnings.pm
   30      85     722 /usr/local/lib/perl5/5.6.0/warnings/register.pm
13733   48195  349869 total
```

Moreover, that requires 116 happy trips through the kernel's namei(). It syscalls open() a remarkable 57 times, 17 of which failed but leaving 38 that were successful. It also syscalled read() a curiously identical 57 times, ingesting a total of 180,265 plump bytes. To top it off, this ***increases your resident set size by two megabytes!***

Happy mallocking...

It seems that `CGI.pm` suffers from the same disease:

```
% wc ‘perl -MCGI -le ’print for values %INC’‘
1368    6920   43710 /usr/local/lib/perl5/5.6.0/overload.pm
6481   26122  200840 /usr/local/lib/perl5/5.6.0/CGI.pm
7849   33042  244550 total
```

You have 16 trips through namei, 7 successful opens, 2 unsuccessful ones, and 213k of data read in.

This is a *perlbloat.pl* that shows how much memory is acquired by Perl when you run some. So we can easily test the overhead of loading some modules.

```
#!/usr/bin/perl -w

use GTop ();

my $gtop = GTop->new;
my $before = $gtop->proc_mem($$)->size;

for (@ARGV) {
    if (eval "require $_") {
        eval {
            $_->import;
        };
    }
    else {
        eval $_;
        die $@ if $@;
    }
```

```
}

my $after = $gtop->proc_mem($$)->size;
printf "@ARGV added %s\n", GTop::size_string($after - $before);
```

Now let's try to load `IO`, which loads `IO::Handle`, `IO::Seekable`, `IO::File`, `IO::Pipe`, `IO::Socket` and `IO::Dir`:

```
% ./perlbloat.pl 'use IO;'
use IO; added  1.5M
```

"Only" 1.5 MB overhead. Now let's load CGI (v2.74) and compile all its methods:

```
% ./perlbloat.pl 'use CGI; CGI->compile(":all")'
use CGI; CGI->compile(":all") added  1.8M
```

Almost 2MB extra memory. Let's compare `CGI.pm` with its younger sister, whose internals are implemented in C.

```
%. /perlbloat.pl 'use Apache::Request'
use Apache::Request added   48k
```

48KB. A significant difference isn't it?

The following numbers show memory sizes in KB (virtual and resident) for v5.6.0 of Perl on four different operating systems, The three calls each are without any modules, with just -MCGI, and with -MIO (never with both):

```
            OpenBSD       FreeBSD    Redhat Linux    Solaris
            vsz   rss     vsz  rss     vsz  rss    vsz    rss
Raw Perl    736   772     832 1208    2412  980    2928  2272
w/ CGI     1220  1464    1308 1828    2972 1768    3616  3232
w/ IO      2292  2580    2456 3016    4080 2868    5384  4976
```

Anybody who's thinking of choosing one of these might do well to digest these numbers first.

14.7.3 Apache::args vs. Apache::Request::param vs. CGI::param

`Apache::args`, `Apache::Request::param` and `CGI::param` are the three most common ways to process input arguments in mod_perl handlers and scripts. Let's write three `Apache::Registry` scripts that use `Apache::args`, `Apache::Request::param` and `CGI::param` to process a form's input and print it out. Notice that `Apache::args` is considered identical to `Apache::Request::param` only when you have single valued keys. In the case of multi-valued keys (e.g. when using check-box groups) you will have to write some extra code: If you do a simple:

```
my %params = $r->args;
```

only the last value will be stored and the rest will collapse, because that's what happens when you turn a list into a hash. Assuming that you have the following list:

```
(rules => 'Apache', rules => 'Perl', rules => 'mod_perl')
```

and assign it to a hash, the following happens:

```
$hash{rules} = 'Apache';
$hash{rules} = 'Perl';
$hash{rules} = 'mod_perl';
```

So at the end only the:

```
rules => 'mod_perl'
```

pair will get stored. With `CGI.pm` or `Apache::Request` you can solve this by extracting the whole list by its key:

```
my @values = $q->params('rules');
```

In addition `Apache::Request` and `CGI.pm` have many more functions that ease input processing, like handling file uploads. However `Apache::Request` is much faster since its guts are implemented in C, glued to Perl using XS code.

Assuming that the only functionality you need is the parsing of key-value pairs, and assuming that every key has a single value, we will compare the following almost identical scripts, by trying to pass various query strings.

Here's the code:

```
file:processing_with_apache_args.pl
-----------------------------------
use strict;
my $r = shift;
$r->send_http_header('text/plain');
my %args = $r->args;
print join "\n", map {"$_ => ".$args{$_} } keys %args;

file:processing_with_apache_request.pl
--------------------------------------
use strict;
use Apache::Request ();
my $r = shift;
my $q = Apache::Request->new($r);
$r->send_http_header('text/plain');
my %args = map {$_ => $q->param($_) } $q->param;
print join "\n", map {"$_ => ".$args{$_} } keys %args;

file:processing_with_cgi_pm.pl
------------------------------
use strict;
use CGI;
my $r = shift;
$r->send_http_header('text/plain');
my $q = new CGI;
my %args = map {$_ => $q->param($_) } $q->param;
print join "\n", map {"$_ => ".$args{$_} } keys %args;
```

All three scripts are preloaded at server startup:

```
<Perl>
    use Apache::RegistryLoader ();
    Apache::RegistryLoader->new->handler(
                              "/perl/processing_with_cgi_pm.pl",
                    "/home/httpd/perl/processing_with_cgi_pm.pl"
                       );
    Apache::RegistryLoader->new->handler(
                              "/perl/processing_with_apache_request.pl",
                    "/home/httpd/perl/processing_with_apache_request.pl"
                       );
    Apache::RegistryLoader->new->handler(
                              "/perl/processing_with_apache_args.pl",
                    "/home/httpd/perl/processing_with_apache_args.pl"
                       );
</Perl>
```

We use four different query strings, generated by:

```
my @queries = (
    join("&", map {"$_=" . 'e' x 10}  ('a'..'b')),
    join("&", map {"$_=" . 'e' x 50}  ('a'..'b')),
    join("&", map {"$_=" . 'e' x 5 }  ('a'..'z')),
    join("&", map {"$_=" . 'e' x 10}  ('a'..'z')),
);
```

The first string is:

```
a=eeeeeeeeee&b=eeeeeeeeee
```

which is 25 characters in length and consists of two key/value pairs. The second string is also made of two key/value pairs, but the value is 50 characters long (total 105 characters). The third and the forth strings are made from 26 key/value pairs, with the value lengths of 5 and 10 characters respectively, with total lengths of 207 and 337 characters respectively. The `query_len` column in the report table is one of these four total lengths.

We conduct the benchmark with concurrency level of 50 and generate 5000 requests for each test.

And the results are:

```
-----------------------------------------------
name   val_len pairs query_len |  avtime  rps
-----------------------------------------------
apreq     10      2       25   |    51    945
apreq     50      2      105   |    53    907
r_args    50      2      105   |    53    906
r_args    10      2       25   |    53    899
apreq      5     26      207   |    64    754
apreq     10     26      337   |    65    742
r_args     5     26      207   |    73    665
r_args    10     26      337   |    74    657
cgi_pm    50      2      105   |    85    573
```

```
cgi_pm    10      2       25   |    87    559
cgi_pm     5     26      207   |   188    263
cgi_pm    10     26      337   |   188    262
-----------------------------------------------
```

Where `apreq` stands for `Apache::Request::param()`, `r_args` stands for `Apache::args()` or `$r->args()` and `cgi_pm` stands for `CGI::param()`.

You can see that `Apache::Request::param` and `Apache::args` have similar performance with a few key/value pairs, but the former is faster with many key/value pairs. `CGI::param` is significantly slower than the other two methods.

14.7.4 Using $|=1 Under mod_perl and Better print() Techniques.

As you know, `local $|=1;` disables the buffering of the currently selected file handle (default is `STDOUT`). If you enable it, `ap_rflush()` is called after each `print()`, unbuffering Apache's IO.

If you are using multiple `print()` calls (_bad_ style in generating output) or if you just have too many of them, then you will experience a degradation in performance. The severity depends on the number of print() calls that you make.

Many old CGI scripts were written like this:

```
print "<BODY BGCOLOR=\"black\" TEXT=\"white\">";
print "<H1>";
print "Hello";
print "</H1>";
print "<A HREF=\"foo.html\"> foo </A>";
print "</BODY>";
```

This example has multiple `print()` calls, which will cause performance degradation with `$|=1`. It also uses too many backslashes. This makes the code less readable, and it is also more difficult to format the HTML so that it is easily readable as the script's output. The code below solves the problems:

```
print qq{
  <BODY BGCOLOR="black" TEXT="white">
    <H1>
      Hello
    </H1>
    <A HREF="foo.html"> foo </A>
  </BODY>
};
```

I guess you see the difference. Be careful though, when printing a `<HTML>` tag. The correct way is:

```
print qq{<HTML>
  <HEAD></HEAD>
  <BODY>
}
```

If you try the following:

```
print qq{
  <HTML>
  <HEAD></HEAD>
  <BODY>
}
```

Some older browsers expect the first characters after the headers and empty line to be `<HTML>` with *no* spaces before the opening left angle-bracket. If there are any other characters, they might not accept the output as HTML and print it as a plain text. Even if it works with your browser, it might not work for others.

One other approach is to use ‘here’ documents, e.g.:

```
print <<EOT;
<HTML>
<HEAD></HEAD>
<BODY>
EOT
```

Now let’s go back to the `$|=1` topic. I still disable buffering, for two reasons:

- **I use relatively few `print()` calls. I achieve this by arranging for my `print()` statements to print multiline HTML, and not one line per `print()` statement.**
- **I want my users to see the output immediately. So if I am about to produce the results of a DB query which might take some time to complete, I want users to get some text while they are waiting. This improves the usability of my site. Ask yourself which you like better: getting the output a bit slower, but steadily from the moment you’ve pressed the Submit button, or having to watch the "falling stars" for a while and then get the whole output at once, even if it’s a few milliseconds faster - assuming the browser didn’t time out during the wait.**

An even better solution is to keep buffering enabled, and use a Perl API `rflush()` call to flush the buffers when needed. This way you can place the first part of the page that you are going to send to the user in the buffer, and flush it a moment before you are going to do some lengthy operation, like a DB query. So you kill two birds with one stone: you show some of the data to the user immediately, so she will feel that something is actually happening, and you have no performance hit from disabled buffering.

```
use CGI ();
my $r = shift;
my $q = new CGI;
print $q->header('text/html');
print $q->start_html;
print $q->p("Searching...Please wait");
$r->rflush;
  # imitate a lengthy operation
for (1..5) {
  sleep 1;
}
print $q->p("Done!");
```

Conclusion: Do not blindly follow suggestions, but think what is best for you in each case.

Note: It might happen that some browsers do not render the page before they have received a significant amount. This is especially true if you insert `<link<` or `<script>` tags in your HTML header that require the browser to load a separate file. In that case, the user won't be able to see the content at once, no matter if you flush the buffers or not.

A workaround for this might be to use an output filter that replaces these tags with the files they refer to.

14.7.5 Global vs. Fully Qualified Variables

It's always a good idea to avoid using global variables where it's possible. Some variables must be either global, such as `@ISA` or else fully qualified such as `@MyModule::ISA`, so that Perl can see them from different packages.

A combination of `strict` and `vars` pragmas keeps modules clean and reduces a bit of noise. However, the `vars` pragma also creates aliases, as does `Exporter`, which eat up more memory. When possible, try to use fully qualified names instead of `use vars`.

For example write:

```
package MyPackage1;
use strict;
use vars; # added only for fair comparison
@MyPackage1::ISA = qw(CGI);
$MyPackage1::VERSION = "1.00";
1;
```

instead of:

```
package MyPackage2;
use strict;
use vars qw(@ISA $VERSION);
@ISA = qw(CGI);
$VERSION = "1.00";
1;
```

Note that we have added the `vars` pragma in the package that doesn't use it so the memory comparison will be fair.

Here are the numbers under Perl version 5.6.0

```
% perl -MGTop -MMyPackage1 -le 'print GTop->new->proc_mem($$)->size'
  2023424
% perl -MGTop -MMyPackage2 -le 'print GTop->new->proc_mem($$)->size'
  2031616
```

We have a difference of 8192 bytes. So every few global variables declared with `vars` pragma add about 8KB overhead.

Note that Perl 5.6.0 introduced a new our() pragma which works like my () scope-wise, but declares global variables.

```
package MyPackage3;
use strict;
use vars; # not needed, added only for fair comparison
our @ISA = qw(CGI);
our $VERSION = "1.00";
1;
```

which uses the same amount of memory as a fully qualified global variable:

```
% perl -MGTop -MMyPackage3 -le 'print GTop->new->proc_mem($$)->size'
2023424
```

Imported symbols act just like global variables, they can add up quick:

```
% perlbloat.pl 'use POSIX ()'
use POSIX () added  316k

% perlbloat.pl 'use POSIX'
use POSIX added  696k
```

That's 380k worth of aliases. Now let's say 6 different `Apache::Registry` scripts `'use POSIX;'` for strftime() or some other function: 6 * 380k = 2.3Mb

One could save 2.3Mb per single process with `'use POSIX ();'` and using fully qualifying `POSIX::` function calls.

14.7.6 Object Methods Calls vs. Function Calls

Which subroutine calling form is more efficient: Object methods or functions?

14.7.6.1 The Overhead with Light Subroutines

Let's do some benchmarking. We will start doing it using empty methods, which will allow us to measure the real difference in the overhead each kind of call introduces. We will use this code:

```
bench_call1.pl
--------------
package Foo;

use strict;
use Benchmark;

sub bar { };

timethese(50_000, {
               method   => sub { Foo->bar()      },
               function => sub { Foo::bar('Foo');},
              });
```

The two calls are equivalent, since both pass the class name as their first parameter; *function* does this explicitly, while *method* does this transparently.

The benchmarking result:

```
Benchmark: timing 50000 iterations of function, method...
  function:  0 wallclock secs ( 0.80 usr +  0.05 sys =  0.85 CPU)
    method:  1 wallclock secs ( 1.51 usr +  0.08 sys =  1.59 CPU)
```

We are interested in the 'total CPU times' and not the 'wallclock seconds'. It's possible that the load on the system was different for the two tests while benchmarking, so the wallclock times give us no useful information.

We see that the *method* calling type is almost twice as slow as the *function* call, 0.85 CPU compared to 1.59 CPU real execution time. Why does this happen? Because the difference between functions and methods is the time taken to resolve the pointer from the object, to find the module it belongs to and then the actual method. The function form has one parameter less to pass, less stack operations, less time to get to the guts of the subroutine.

perl5.6+ does better method caching, `Foo->method()` is a little bit faster (some constant folding magic), but not `Foo->$method()`. And the improvement does not address the `@ISA` lookup that still happens in either case.

14.7.6.2 The Overhead with Heavy Subroutines

But that doesn't mean that you shouldn't use methods. Generally your functions do something, and the more they do the less significant is the time to perform the call, because the calling time is effectively fixed and is probably a very small overhead in comparison to the execution time of the method or function itself. Therefore the longer execution time of the function the smaller the relative overhead of the method call. The next benchmark proves this point:

```
bench_call2.pl
--------------
package Foo;

use strict;
use Benchmark;

sub bar {
  my $class = shift;

  my ($x,$y) = (100,100);
  $y = log ($x ** 10)  for (0..20);
};

timethese(50_000, {
               method   => sub { Foo->bar()      },
               function => sub { Foo::bar('Foo');},
              });
```

We get a very close benchmarks!

```
function: 33 wallclock secs (15.81 usr +  1.12 sys = 16.93 CPU)
  method: 32 wallclock secs (18.02 usr +  1.34 sys = 19.36 CPU)
```

Let's make the subroutine *bar* even slower:

```
sub bar {
  my $class = shift;

  my ($x,$y) = (100,100);
  $y = log ($x ** 10)  for (0..40);
};
```

And the result is amazing, the *method* call convention was faster than *function*:

```
function: 81 wallclock secs (25.63 usr +  1.84 sys = 27.47 CPU)
  method: 61 wallclock secs (19.69 usr +  1.49 sys = 21.18 CPU)
```

In case your functions do very little, like the functions that generate HTML tags in `CGI.pm`, the overhead might become a significant one. If your goal is speed you might consider using the *function* form, but if you write a big and complicated application, it's much better to use the *method* form, as it will make your code easier to develop, maintain and debug, saving programmer time which, over the life of a project may turn out to be the most significant cost factor.

14.7.6.3 Are All Methods Slower than Functions?

Some modules' API is misleading, for example `CGI.pm` allows you to execute its subroutines as functions or as methods. As you will see in a moment its function form of the calls is slower than the method form because it does some voodoo work when the function form call is used.

```
use CGI;
my $q = new CGI;
$q->param('x',5);
my $x = $q->param('x');
```

vs

```
use CGI qw(:standard);
param('x',5);
my $x = param('x');
```

As usual, let's benchmark some very light calls and compare. Ideally we would expect the *methods* to be slower than *functions* based on the previous benchmarks:

```
bench_call3.pl
---------------
use Benchmark;

use CGI qw(:standard);
$CGI::NO_DEBUG = 1;
my $q = new CGI;
my $x;
```

```
timethese
  (20000, {
    method   => sub {$q->param('x',5); $x = $q->param('x'); },
    function => sub {    param('x',5); $x =     param('x'); },
   });
```

The benchmark is written is such a way that all the initializations are done at the beginning, so that we get as accurate performance figures as possible. Let's do it:

```
% ./bench_call3.pl

function: 51 wallclock secs (28.16 usr +  2.58 sys = 30.74 CPU)
  method: 39 wallclock secs (21.88 usr +  1.74 sys = 23.62 CPU)
```

As we can see methods are faster than functions, which seems to be wrong. The explanation lays in the way `CGI.pm` is implemented. `CGI.pm` uses some *fancy* tricks to make the same routine act both as a *method* and a plain *function*. The overhead of checking whether the arguments list looks like a *method* invocation or not, will mask the slight difference in time for the way the function was called.

If you are intrigued and want to investigate further by yourself the subroutine you want to explore is called *self_or_default*. The first line of this function short-circuits if you are using the object methods, but the whole function is called if you are using the functional forms. Therefore, the functional form should be slightly slower than the object form.

14.7.7 Imported Symbols and Memory Usage

There is a real memory hit when you import all of the functions into your process' memory. This can significantly enlarge memory requirements, particularly when there are many child processes.

In addition to polluting the namespace, when a process imports symbols from any module or any script it grows by the size of the space allocated for those symbols. The more you import (e.g. qw(:standard) vs qw(:all)) the more memory will be used. Let's say the overhead is of size X. Now take the number of scripts in which you deploy the function method interface, let's call that Y. Finally let's say that you have a number of processes equal to Z.

You will need X*Y*Z size of additional memory, taking X=10k, Y=10, Z=30, we get 10k*10*30 = 3Mb!!! Now you understand the difference.

Let's benchmark `CGI.pm` using `GTop.pm`. First we will try it with no exporting at all.

```
use GTop ();
use CGI ();
print GTop->new->proc_mem($$)->size;

1,949,696
```

Now exporting a few dozens symbols:

```
use GTop ();
use CGI qw(:standard);
print GTop->new->proc_mem($$)->size;

1,966,080
```

And finally exporting all the symbols (about 130)

```
use GTop ();
use CGI qw(:all);
print GTop->new->proc_mem($$)->size;

1,970,176
```

Results:

```
import symbols  size(bytes)  delta(bytes) relative to ()
--------------------------------------
()              1949696             0
qw(:standard)   1966080         16384
qw(:all)        1970176         20480
```

So in my example above X=20k => 20K*10*30 = 6Mb. You will need 6Mb more when importing all the `CGI.pm`'s symbols than when you import none at all.

Generally you use more than one script, run more than one process and probably import more symbols from the additional modules that you deploy. So the real numbers are much bigger.

The function method is faster in the general case, because of the time overhead to resolve the pointer from the object.

If you are looking for performance improvements, you will have to face the fact that having to type `My::Module::my_method` might save you a good chunk of memory if the above call must not be called with a reference to an object, but even then it can be passed by value.

I strongly endorse Apache::Request (libapreq) - Generic Apache Request Library. Its core is written in C, giving it a significant memory and performance benefit. It has all the functionality of `CGI.pm` except the HTML generation functions.

14.7.8 Interpolation, Concatenation or List

Somewhat overlapping with the previous section we want to revisit the various approaches of mungling with strings, and compare the speed of using lists of strings compared to interpolation. We will add a string concatenation angle as well.

When the strings are small, it almost doesn't matter whether interpolation or a list is used. Here is a benchmark:

```
use Benchmark;
use Symbol;
my $fh = gensym;
open $fh, ">/dev/null" or die;
```

```
my ($one, $two, $three, $four) = (’a’..’d’);

timethese(1_000_000,
    {
     interp => sub {
       print $fh "$one$two$three$four";
     },
     list => sub {
       print $fh $one, $two, $three, $four;
     },
     conc => sub {
       print $fh $one.$two.$three.$four;
     },
    });
```

```
Benchmark: timing 1000000 iterations of conc, interp, list...
     conc:  3 wallclock secs ( 3.38 usr +  0.00 sys =  3.38 CPU)
   interp:  3 wallclock secs ( 3.45 usr + -0.01 sys =  3.44 CPU)
     list:  2 wallclock secs ( 2.58 usr +  0.00 sys =  2.58 CPU)
```

The concatenation technique is very similar to interpolation. The list technique is a little bit faster than interpolation. But when the strings are large, lists are significantly faster. We have seen this in the previous section and here is another benchmark to increase our confidence in our conclusion. This time we use 1000 character long strings:

```
use Benchmark;
use Symbol;
my $fh = gensym;
open $fh, ">/dev/null" or die;

my ($one, $two, $three, $four) = map { $_ x 1000 } (’a’..’d’);

timethese(500_000,
    {
     interp => sub {
       print $fh "$one$two$three$four";
     },
     list => sub {
       print $fh $one, $two, $three, $four;
     },
     conc => sub {
       print $fh $one.$two.$three.$four;
     },
    });
```

```
Benchmark: timing 500000 iterations of interp, list...
    conc:  5 wallclock secs ( 4.47 usr +  0.27 sys =  4.74 CPU)
  interp:  4 wallclock secs ( 4.25 usr +  0.26 sys =  4.51 CPU)
    list:  4 wallclock secs ( 2.87 usr +  0.16 sys =  3.03 CPU)
```

In this case using a list is about 30% faster than interpolation. Concatenation is a little bit slower than interpolation.

Let's look at this code:

```
$title = 'My Web Page';
print "<h1>$title</h1>";         # Interpolation (slow)
print '<h1>' . $title . '</h1>'; # Concatenation (slow)
print '<h1>', $title, '</h1>';   # List (fast for long strings)
```

When you use *"<h1>$title</h1>"* Perl does interpolation (since `""` is an operator in Perl), which must parse the contents of the string and replace any variables or expressions it finds with their respective values. This uses more memory and is slower than using a list. Of course if there are no variables to interpolate it makes no difference whether to use `"string"` or `'string'`.

Concatenation is also potentially slow since Perl might create a temporary string which it then prints.

Lists are fast because Perl can simply deal with each element in turn. This is true if you don't run join() on the list at the end to create a single string from the elements of list. This operation might be slower than direct append to the string whenever a new string springs into existence.

[ReaderMETA]: Please send more mod_perl relevant Perl performance hints

14.7.9 Using Perl stat() Call's Cached Results

When you do a stat() (or its variations `-M` -- last modification time, `-A` -- last access time, `-C` -- last inode-change time, etc), the returned information is cached internally. If you need to make an additional check for the same file, use the _ magic variable and save the overhead of an unnecessary stat() call. For example when testing for existence and read permissions you might use:

```
my $filename = "./test";
  # three stat() calls
print "OK\n" if -e $filename and -r $filename;
my $mod_time = (-M $filename) * 24 * 60 * 60;
print "$filename was modified $mod_time seconds before startup\n";
```

or the more efficient:

```
my $filename = "./test";
  # one stat() call
print "OK\n" if -e $filename and -r _;
my $mod_time = (-M _) * 24 * 60 * 60;
print "$filename was modified $mod_time seconds before startup\n";
```

Two stat() calls were saved!

14.7.10 Optimizing Code

Here are some other resources that explain how to optimize your code, which are usually applied when you profile your code and need to optimize it but in many cases are useful to know when you develop the code.

- Interesting C code optimization notes, most applying to Perl code as well: http://www.utsc.utoronto.ca/~harper/cscb09/lecture11.html#code

[ReaderMETA]: please send me similar resources if you know of such.

14.8 Apache::Registry and Derivatives Specific Notes

These are the sections that deal solely with `Apache::Registry` and derived modules, like `Apache::PerlRun` and `Apache::RegistryBB`. No Perl handlers code is discussed here, so if you don't use these modules, feel free to skip this section.

14.8.1 Be Careful with Symbolic Links

As you know `Apache::Registry` caches the scripts in the packages whose names are constructed by scripts' URI. If you have the same script that can be reached by different URIs, which is possible if you have used symbolic links, you will get the same script stored twice in the memory.

For example:

```
% ln -s /home/httpd/perl/news/news.pl /home/httpd/perl/news.pl
```

Now the script can be reached through the both URIs */news/news.pl* and */news.pl*. It doesn't really matter until you advertise the two URIs, and users reach the same script from both of them.

So let's assume that you have issued the requests to the both URIs:

```
http://localhost/perl/news/news.pl
http://localhost/perl/news.pl
```

To spot the duplication you should use the `Apache::Status` module. Amongst other things, it shows all the compiled `Apache::Registry` scripts (using their respective packages):

If you are using the default configuration directives you should either use this URI:

```
http://localhost/perl-status?rgysubs
```

or just go to the main menu at:

```
http://localhost/perl-status
```

And click on `Compiled Registry Scripts` menu item.

META: we need a screen snapshot here!!!

If you the script was accessed through the URI that was remapped to the real file and through the URI that was remapped to the symbolic link, you will see the following output:

```
Apache::ROOT::perl::news::news_2epl
Apache::ROOT::perl::news_2epl
```

You should run the server in the single mode, to see it immediately. If you test it in the normal mode--it's possible that some child processes would show only one entry or none at all, since they might not serve the same requests as the others. For more hints see the section "Run the server in single mode".

14.9 Improving Performance by Prevention

There are two ways to improve performance: one is by tuning to squeeze the most out of your hardware and software; and the other is preventing certain bad things from happening, like impolite robots that crawl your site without pausing between requests, memory leakages, getting the memory unshared, making sure that some processes won't take up all the CPU etc.

In the following sections we are going to discuss about the tools and programming techniques that would help you to keep your service in order, even if you are not around.

14.9.1 Memory leakage

Scripts under mod_perl can very easily leak memory! Global variables stay around indefinitely, lexically scoped variables (declared with `my ()`) are destroyed when they go out of scope, provided there are no references to them from outside that scope.

Perl doesn't return the memory it acquired from the kernel. It does reuse it though!

14.9.1.1 Reading In A Whole File

```
open IN, $file or die $!;
local $/ = undef; # will read the whole file in
$content = <IN>;
close IN;
```

If your file is 5Mb, the child which served that script will grow by exactly that size. Now if you have 20 children, and all of them will serve this CGI, they will consume 20*5M = 100M of RAM in total! If that's the case, try to use other approaches to processing the file, if possible. Try to process a line at a time and print it back to the file. If you need to modify the file itself, use a temporary file. When finished, overwrite the source file. Make sure you use a locking mechanism!

14.9.1.2 Copying Variables Between Functions

Now let's talk about passing variables by value. Let's use the example above, assuming we have no choice but to read the whole file before any data processing takes place. Now you have some imaginary `process()` subroutine that processes the data and returns it. What happens if you pass the `$content` by value? You have just copied another 5M and the child has grown in size by **another** 5M. Watch your swap space! Now multiply it again by factor of 20 you have 200M of wasted RAM, which will apparently be reused, but it's a waste! Whenever you think the variable can grow bigger than a few Kb, pass it by reference!

Once I wrote a script that passed the contents of a little flat file database to a function that processed it by value -- it worked and it was fast, but after a time the database became bigger, so passing it by value was expensive. I had to make the decision whether to buy more memory or to rewrite the code. It's obvious that adding more memory will be merely a temporary solution. So it's better to plan ahead and pass variables by reference, if a variable you are going to pass might eventually become bigger than you envisage at the time you code the program. There are a few approaches you can use to pass and use variables passed by reference. For example:

```
my $content = qq{foobarfoobar};
process(\$content);
sub process{
  my $r_var = shift;
  $$r_var =~ s/foo/bar/gs;
    # nothing returned - the variable $content outside has already
    # been modified
}
```

If you work with arrays or hashes it's:

```
@{$var_lr}  dereferences an array
%{$var_hr}  dereferences a hash
```

We can still access individual elements of arrays and hashes that we have a reference to without dereferencing them:

```
$var_lr->[$index]  get $index'th element of an array via a ref
$var_hr->{$key}    get $key'th element of a hash via a ref
```

For more information see `perldoc perlref`.

Another approach would be to use the `@_` array directly. This has the effect of passing by reference:

```
process($content);
sub process{
  $_[0] =~ s/foo/bar/gs;
    # nothing returned - the variable $content outside has been
    # already modified
}
```

From `perldoc perlsub`:

```
    The array @_ is a local array, but its elements are aliases for
    the actual scalar parameters.  In particular, if an element
    $_[0] is updated, the corresponding argument is updated (or an
    error occurs if it is not possible to update)...
```

Be careful when you write this kind of subroutine, since it can confuse a potential user. It's not obvious that call like `process($content);` modifies the passed variable. Programmers (the users of your library in this case) are used to subroutines that either modify variables passed by reference or expressly return a result (e.g. `$content=process($content);`).

14.9.1.3 Work With Databases

If you do some DB processing, you will often encounter the need to read lots of records into your program, and then print them to the browser after they are formatted. I won't even mention the horrible case where programmers read in the whole DB and then use Perl to process it!!! Use a relational DB and let the SQL do the job, so you get only the records you need!

We will use `DBI` for this (assume that we are already connected to the DB--refer to `perldoc DBI` for a complete reference to the `DBI` module):

```
$sth->execute;
while(@row_ary  = $sth->fetchrow_array) {
  # do DB accumulation into some variable
}
# print the output using the data returned from the DB
```

In the example above the httpd_process will grow by the size of the variables that have been allocated for the records that matched the query. Again remember to multiply it by the number of the children your server runs!

A better approach is not to accumulate the records, but rather to print them as they are fetched from the DB. Moreover, we will use the `bind_col()` and `$sth->fetchrow_arrayref()` (aliased to `$sth->fetch()`) methods, to fetch the data in the fastest possible way. The example below prints an HTML table with matched data, the only memory that is being used is a `@cols` array to hold temporary row values. The table will be rendered by the client browser only when the whole table will be out though.

```
my @select_fields = qw(a b c);
    # create a list of cols values
my @cols = ();
@cols[0..$#select_fields] = ();
$sth = $dbh->prepare($do_sql);
$sth->execute;
  # Bind perl variables to columns.
$sth->bind_columns(undef,\(@cols));
print "<TABLE>";
while($sth->fetch) {
   print "<TR>",
         map("<TD>$_</TD>", @cols),
         "</TR>";
}
print "</TABLE>";
```

Note: the above method doesn't allow you to know how many records have been matched. The workaround is to run an identical query before the code above where you use `SELECT count(*) ...` instead of `'SELECT * ...`, to get the number of matched records. It should be much faster, since you can remove any **SORTBY** and similar attributes.

For those who think that **$sth->rows** will do the job, here is the quote from the `DBI` manpage:

```
rows();

$rv = $sth->rows;

Returns the number of rows affected by the last database altering
command, or -1 if not known or not available.  Generally you can
only rely on a row count after a do or non-select execute (for some
specific operations like update and delete) or after fetching all
the rows of a select statement.

For select statements it is generally not possible to know how many
rows will be returned except by fetching them all.  Some drivers
will return the number of rows the application has fetched so far
but others may return -1 until all rows have been fetched. So use of
the rows method with select statements is not recommended.
```

As a bonus, I wanted to write a single sub that flexibly processes any query. It would accept conditions, a call-back closure sub, select fields and restrictions.

```
# Usage:
# $o->dump(\%conditions,\&callback_closure,\@select_fields,@restrictions);
#
sub dump{
  my $self = shift;
  my %param = %{+shift}; # dereference hash
  my $rsub = shift;
  my @select_fields = @{+shift}; # dereference list
  my @restrict = shift || '';

    # create a list of cols values
  my @cols = ();
  @cols[0..$#select_fields] = ();

  my $do_sql = '';
  my @where = ();

    # make a @where list
  map { push @where, "$_=\'$param{$_}\'" if $param{$_};} keys %param;

    # prepare the sql statement
  $do_sql = "SELECT ";
  $do_sql .= join(" ", @restrict) if @restrict;    # append restriction list
  $do_sql .= " " .join(",", @select_fields) ;      # append select list
  $do_sql .= " FROM $DBConfig{TABLE} ";            # from table

    # we will not add the WHERE clause if @where is empty
  $do_sql .= " WHERE " . join " AND ", @where if @where;

  print "SQL: $do_sql \n" if $debug;

  $dbh->{RaiseError} = 1; # do this, or check every call for errors
  $sth = $dbh->prepare($do_sql);
  $sth->execute;
    # Bind perl variables to columns.
  $sth->bind_columns(undef,\(@cols));
```

```
    while($sth->fetch) {
      &$rsub(@cols);
    }
      # print the tail or "no records found" message

      # according to the previous calls
    &$rsub();

  } # end of sub dump
```

Now a callback closure sub can do lots of things. We need a closure to know what stage are we in: header, body or tail. For example, we want a callback closure for formatting the rows to print:

```
  my $rsub = eval {
      # make a copy of @fields list, since it might go
      # out of scope when this closure is called
    my @fields = @fields;
    my @query_fields = qw(user dir tool act);   # no date field!!!
    my $header = 0;
    my $tail   = 0;
    my $counter = 0;
    my %cols = ();                    # columns name=> value hash

    # Closure with the following behavior:
    # 1. Header's code will be executed on the first call only and
    #    if @_ was set
    # 2. Row's printing code will be executed on every call with @_ set
    # 3. Tail's code will be executed only if Header's code was
    #    printed and @_ isn't set
    # 4. "No record found" code will be executed if Header's code
    #    wasn't executed

    sub {
          # Header
    if (@_ and !$header){
      print "<TABLE>\n";
      print $q->Tr(map{ $q->td($_) } @fields );
      $header = 1;
    }

      # Body
    if (@_) {
      print $q->Tr(map{$q->td($_)} @_ );
      $counter++;
      return;
    }

          # Tail, will be printed only at the end
    if ($header and !($tail or @_)){
      print "</TABLE>\n $counter records found";
      $tail = 1;
      return;
    }
```

```
        # No record found
  unless ($header){
    print $q->p($q->center($q->b("No record was found!\n")));
  }

    }  #  end of sub {}
};  #  end of my $rsub = eval {
```

You might also want to check the section Preventing Your Processes from Growing and Limiting Other Resources Used by Apache Child Processes.

14.9.2 Avoid the Evil Troika

The *perlre* manpage says:

```
WARNING: Once Perl sees that you need one of "$&", "$‘", or "$’"
anywhere in the program, it has to provide them for every pattern
match. This may substantially slow your program.
```

The mere existence of these variables will trigger this behavior, regardless of whether or not the code that accesses them will be executed. Removing these variables should significantly improve the regex performance.

How do you know whether some code loads them? You could grep(1), but it’s hard to remember to do that as you include more modules from CPAN and write new code. Luckily `Devel::SawAmpersand` comes to help. (http://search.cpan.org/dist/Devel-SawAmpersand/lib/Devel/SawAmpersand.pm) This module will alert you if it detects any of the evil troika variables present.

14.9.3 Preventing Your Processes from Growing

If you have already worked with mod_perl, you have probably noticed that it can be difficult to keep your mod_perl processes from using a lot of memory. The less memory you have, the fewer processes you can run and the worse your server will perform, especially under a heavy load. This chapter presents several common situations which can lead to unnecessary consumption of RAM, together with preventive measures.

When you need to control the size of your httpd processes, use one of the two modules `Apache::GTopLimit` and `Apache::SizeLimit` which kill Apache httpd processes when the latter grow too large or lose a big chunk of their shared memory. The two modules differ in methods for finding out the memory usage. `Apache::GTopLimit` relies on the *libgtop* library to perform this task, therefore if this library can be built on your platform you can use this module. `Apache::SizeLimit` includes different methods for different platforms, you will have to check the modules’ manpage to figure out which platforms are supported.

14.9.3.1 Defining the Minimum Shared Memory Size Threshold

As we have already discussed, when it is first created an Apache child process usually has a large fraction of it memory shared with its parent. During the child process' life some of its data structures are modified and a part of its memory becomes unshared (pages become *"dirty"*), leading to an increase in memory consumption. You will remember that the `MaxRequestsPerChild` directive allows you to specify the number of requests a child process should serve before it is killed. One way to limit the memory consumption of a process is to kill it and let Apache replace it with a newly started process, which again will have all its memory shared with the Apache parent. The new child process serves requests and eventually the cycle is repeated.

This is a fairly crude means of limiting unshared memory and you will probably need to tune `MaxRequestsPerChild`, eventually finding an optimum value. If, as is likely, your service is undergoing constant changes then this is an inconvenient solution. You have to re-tune this number again and again to adapt to the ever changing code base.

You really want to set some guardian to watch the shared size and kill the process if it goes below some limit. This way, processes will not be killed unnecessarily.

To set a shared memory lower limit of 4MB using `Apache::GTopLimit` add the following code into the *startup.pl* file:

```
use Apache::GTopLimit;
$Apache::GTopLimit::MIN_PROCESS_SHARED_SIZE = 4096;
```

and in *httpd.conf*:

```
PerlFixupHandler Apache::GTopLimit
```

don't forget to restart the server for the changes to take effect.

This has the effect that as soon as the child process shares less than 4MB, (the corollary being that it must therefore be occupying a lot of memory with its unique pages), it will be killed after completing to serve the last request, and, as a consequence, a new child will take its place.

If you use `Apache::SizeLimit` you can accomplish the same with the adding to *startup.pl*:

```
use Apache::SizeLimit;
$Apache::SizeLimit::MIN_SHARE_SIZE = 4096;
```

and in *httpd.conf*:

```
PerlFixupHandler Apache::SizeLimit
```

If you only want to set this limit for some requests (presumably the ones which you think are likely to cause memory to become unshared) then you can register a post-processing check using the set_min_shared_size() function. For example:

```
use Apache::GTopLimit;
if ($need_to_limit) {
    # make sure that at least 4MB are shared
    Apache::GTopLimit->set_min_shared_size(4096);
}
```

or for `Apache::SizeLimit`:

```
use Apache::SizeLimit;
if ($need_to_limit) {
    # make sure that at least 4MB are shared
    Apache::SizeLimit->setmin(4096);
}
```

Since accessing the process information adds a little overhead, you may want to only check the process size every N times. In this case set the `$Apache::GTopLimit::CHECK_EVERY_N_REQUESTS` variable. For example to test the size every other time, put in your *startup.pl*:

```
$Apache::GTopLimit::CHECK_EVERY_N_REQUESTS = 2;
```

or for `Apache::SizeLimit`:

```
$Apache::SizeLimit::CHECK_EVERY_N_REQUESTS = 2;
```

You can run the `Apache::GTopLimit` module in the debug mode by setting:

```
PerlSetVar Apache::GTopLimit::DEBUG 1
```

in *httpd.conf*. It's important that this setting should happen before the `Apache::GTopLimit` module is loaded.

When debug mode is turned *on* the module reports in the *error_log* file the memory usage of the current process and also when it detects that at least one of the thresholds was crosses and the process is going to be killed.

`Apache::SizeLimit` controls the debug level via `$Apache::SizeLimit::DEBUG` variable:

```
$Apache::SizeLimit::DEBUG = 1;
```

which can be modified any time, even after the module was loaded.

14.9.3.2 Potential Drawbacks of Memory Sharing Restriction

It's very important that the system won't be heavily engaged in swapping process. Some systems do swap in and out every so often even if they have plenty of real memory available and it's OK. The following applies to conditions when there is hardly any free memory available.

So if the system uses almost all of its real memory (including the cache), there is a danger of parent's process memory pages being swapped out (written to a swap device). If this happens the memory usage reporting tools will report all those swapped out pages as non-shared, even though in reality these pages are still shared on most OSs. When these pages are getting swapped in, the sharing will be reported back to normal after a certain amount of time. If a big chunk of the memory shared with child processes is

swapped out, it's most likely that `Apache::SizeLimit` or `Apache::GTopLimit` will notice that the shared memory floor threshold was crossed and as a result kill those processes. If many of the parent process' pages are swapped out, and the newly created child process is already starting with shared memory below the limit, it'll be killed immediately after serving a single request (assuming that we the `$CHECK_EVERY_N_REQUESTS` is set to one). This is a very bad situation which will eventually lead to a state where the system won't respond at all, as it'll be heavily engaged in swapping process.

This effect may be less or more severe depending on the memory manager's implementation and it certainly varies from OS to OS, and different kernel versions. Therefore you should be aware of this potential problem and simply try to avoid situations where the system needs to swap at all, by adding more memory, reducing the number of child servers or spreading the load across more machines, if reducing the number of child servers is not an options because of the request rate demands.

14.9.3.3 Defining the Maximum Memory Size Threshold

Not less important than maximizing shared memory is restricting the absolute size of the processes. If the processes grow after each request, and if nothing restricts them from growing, you can easily run out of memory.

Again you can set the `MaxRequestPerChild` directive to kill the processes after a few requests have been served. But as we have explained in the previous section this solution is not as good as one which monitors the process size and kills it only when some limit is reached.

If you have `Apache::GTopLimit` (described in the previous section) you can limit process' memory usage by setting the `$Apache::GTopLimit::MAX_PROCESS_SIZE` directive. For example if you want the processes to be killed when they reach 10MB you should put the following in your *startup.pl* file:

```
$Apache::GTopLimit::MAX_PROCESS_SIZE = 10240;
```

Just as when limiting shared memory, you can set a limit for the current process using the set_max_size() method in your code:

```
use Apache::GTopLimit;
Apache::GTopLimit->set_max_size(10000);
```

For `Apache::SizeLimit` the equivalents are:

```
use Apache::SizeLimit;
$Apache::SizeLimit::MAX_PROCESS_SIZE = 10240;
```

and:

```
use Apache::SizeLimit;
Apache::SizeLimit->setmax(10240);
```

14.9.3.4 Defining the Maximum Unshared Memory Size Threshold

Instead of setting the shared and total memory usage thresholds, you can set a single threshold which measures the amount of unshared memory, by subtracting the shared memory size from the total memory size.

Both modules allow you to set the thresholds in similar ways. With `Apache::GTopLimit` you can set the unshared memory threshold server-wide with:

```
$Apache::GTopLimit::MAX_PROCESS_UNSHARED_SIZE = 6144;
```

and locally for a handler with:

```
Apache::GTopLimit->set_max_unshared_size(6144);
```

If you are using `Apache::SizeLimit` the corresponding settings would be:

```
$Apache::SizeLimit::MAX_UNSHARED_SIZE = 6144;
```

and:

```
Apache::SizeLimit->setmax_unshared(6144);
```

14.9.4 Limiting Other Resources Used by Apache Child Processes

In addition to the absolute and shared memory sizes limiting, you might need to prevent the processes from excessive consumption of the system resources. Like limiting the CPU usage, the number of files that can be opened, or memory segment usage and more.

The `Apache::Resource` module allows this all by deploying the `BSD::Resource` module, which in turn uses the C function `setrlimit()` to set limits on system resources.

A resource limit is specified as a soft limit and a hard limit. When a soft limit is exceeded a process may receive a signal (for example, if the CPU time or file size is exceeded), but it will be allowed to continue execution until it reaches the hard limit (or modifies its resource limit). The rlimit structure is used to specify the hard and soft limits on a resource. (See the manpage for *setrlimit* for your OS specific information.)

If the value of the variable is of the form `S:H`, `S` is treated as the soft limit, and `H` is the hard limit. If it is just a single number, it is used for both soft and hard limits. So if you set `10:20`, the soft limit is 10 and the hard limit is 20. If you set just `10`--both the soft and the hard limits are set to 20.

The mostly spread usage of this module is to limit the CPU usage. The environment variable `PERL_RLIMIT_CPU` defines the maximum amount of CPU time the process can use. If it runs for longer than this, it gets killed, no matter what it does, either processing a new request or just waiting. This is very useful when you have a code with a bug and the process starts to spin in an infinite loop or alike using a lot of CPU and never completing the request.

The value is measured in seconds. The following example sets the soft limit of the CPU usage to 120 seconds (the default is 360).

```
PerlModule Apache::Resource
PerlSetEnv PERL_RLIMIT_CPU 120
```

Of course you should tell mod_perl to use this module, which is done by adding the following directive to *httpd.conf*:

```
PerlChildInitHandler Apache::Resource
```

There are other resources that you might want to limit. For example you can limit the memory data and stack segment sizes (`PERL_RLIMIT_DATA` and `PERL_RLIMIT_STACK`), the maximum process file size (`PERL_RLIMIT_FSIZE`), the core file size (`PERL_RLIMIT_CORE`), the address space (virtual memory) limit (`PERL_RLIMIT_AS`), etc. Refer to the setrlimit(2) man page on your OS for other possible resources. Remember to prepend `PERL_` before the resource types you will see in the man page.

If you configure `Apache::Status`, it will let you review the resources set in this way. Remember that `Apache::Status` must be loaded before `Apache::Resource` in order to enable the resources display menu.

If you want to set the debug mode set the `$Apache::Resource::Debug` before loading the module, for example by using the Perl sections in *httpd.conf*.

```
<Perl>
  $Apache::Resource::Debug = 1;
  require Apache::Resource;
</Perl>
PerlChildInitHandler Apache::Resource
```

Now open in the *error_log* file using tell and watch the debug messages showing up, when the requests are served.

14.9.4.1 OS Specific notes

Note that under Linux malloc() uses mmap() instead of brk(). This is done to conserve virtual memory - that is, when you malloc a large block of memory, it isn't actually given to your program until you initialize it. The old-style brk() system call obeyed resource limits on data segment size as set in setrlimit() - mmap() doesn't.

`Apache::Resource`'s defaults put caps on data size and stack size. Linux's current memory allocation scheme doesn't honor these limits, so if you just do

```
PerlSetEnv PERL_RLIMIT_DEFAULTS On
PerlModule Apache::Resource
PerlChildInitHandler Apache::Resource
```

Your Apache processes are still free to use as much memory as they like.

However, `BSD::Resource` also has a limit called `RLIMIT_AS` (Address Space) which limits the total number of bytes of virtual memory assigned to a process. Happily, Linux's memory manager *does* honor this limit.

Therefore, you *can* limit memory usage under Linux with `Apache::Resource` -- simply add a line to *httpd.conf*:

```
PerlSetEnv PERL_RLIMIT_AS  67108864
```

This example sets a hard and soft limit of 64MB of total address space.

Refer to the `Apache::Resource` and `setrlimit(2)` manpages for more information.

14.9.5 Limiting the Number of Processes Serving the Same Resource

If you want to limit number of Apache children that could simultaneously be serving the (nearly) same resource, you should take a look at the `mod_throttle_access` module.

It solves the problem of too many concurrent request accessing the same URI, if for example the handler that serves this URI uses some resource that has a limitation on the maximum number of possible users or the handlers code is very CPU intensive and you cannot afford more than a certain number of concurrent requests to this specific URI.

Imagine that your service provides the three following URIs:

```
/perl/news/
/perl/webmail/
/perl/morphing/
```

The first two URIs are response critical as people want to read news and their email. The third URI is very CPU and RAM intensive image morphing service, provided as a bonus to your users. Since you don't want users to abuse this service, you have to set some limits on the number of concurrent requests for this resource, since if you don't--the other two critical resources can be hurt.

When you compile in and enable the Apache mod_throttle_access module, the `MaxConcurrentReqs` directive becomes available. For example, the following setting:

```
<Location "/perl/morphing">
  <Limit PUT GET POST>
    MaxConcurrentReqs 10
  </Limit>
</Location>
```

will allow only 10 concurrent PUT, GET or POST requests under the URI */perl/morphing* to be processed at one time. The other two URIs remain unlimited.

14.9.6 Limiting the Request Rate Speed (Robot Blocking)

A limitation of using pattern matching to identify robots is that it only catches the robots that you know about, and then only those that identify themselves by name. A few devious robots masquerade as users by using user agent strings that identify themselves as conventional browsers. To catch such robots, you'll have to be more sophisticated.

`Apache::SpeedLimit` comes to your aid, see:

http://www.modperl.com/chapters/ch6.html#Blocking_Greedy_Clients

14.10 Perl Modules for Performance Improvement

These sections are about Perl modules that improve performance without requiring changes to your code. Mostly you just need to tweak the configuration file to plug these modules in.

14.10.1 Sending Plain HTML as Compressed Output

See Apache::GzipChain - compress HTML (or anything) in the OutputChain

14.10.2 Caching Components with HTML::Mason

META: complete the full description

`HTML::Mason` is a system that makes use of components to build HTML pages.

If most of your output is generated dynamically, but each finished page can be separated into different components, `HTML::Mason` can cache those components. This can really improve the performance of your service and reduce the load on the system.

Say for example that you have a page consisting of five components, each generated by a different SQL query, but for four of the five components it's the same four queries for each user so you don't have to rerun them again and again. Only one component is generated by a unique query and will not use the cache.

META: HTML::Mason docs (v 8.0) said Mason was 2-3 times slower than pure mod_perl, implying that the power & convenience made up for this.

META: Should also mention Embperl (especially since its C + XS)

14.11 Efficient Work with Databases under mod_perl

Most of the mod_perl enabled servers work with database engines, so in this section we will learn about two things: how mod_perl makes working with databases faster and a few tips for a more efficient DBI coding in Perl. (DBI provides an identical Perl interface to many database implementations.)

14.11.1 Persistent DB Connections

Another popular use of mod_perl is to take advantage of its ability to maintain persistent open database connections.

You want to have a persistent database connection because the most expensive part of a network transaction for most databases is the business of building and tearing down connections.

Of course the persistence doesn't help with the latency problems during the actual use of the database connections. Oracle is notoriously latency-sensitive which in most cases generates a network transaction per row returned which slows things down if the query execution matches many rows. You may want to read the Tim Bunce's Advanced DBI talk at http://dbi.perl.org/doc/conferences/tim_1999/index.html which covers a lot of techniques to reduce latency.

So here is the basic approach of making the connection persistent:

```
# Apache::Registry script
-------------------------
use strict;
use vars qw($dbh);

$dbh ||= SomeDbPackage->connect(...);
```

Since `$dbh` is a global variable for the child, once the child has opened the connection it will use it over and over again, unless you perform `disconnect()`.

Be careful to use different names for handlers if you open connections to different databases!

`Apache::DBI` allows you to make a persistent database connection. With this module enabled, every `connect()` request to the plain `DBI` module will be forwarded to the `Apache::DBI` module. This looks to see whether a database handle from a previous `connect()` request has already been opened, and if this handle is still valid using the ping method. If these two conditions are fulfilled it just returns the database handle. If there is no appropriate database handle or if the ping method fails, a new connection is established and the handle is stored for later re-use. **There is no need to delete the `disconnect()` statements from your code**. They will not do anything, the `Apache::DBI` module overloads the `disconnect()` method with a NOP. When a child exits there is no explicit disconnect, the child dies and so does the database connection. You may leave the `use DBI;` statement inside the scripts as well.

The usage is simple -- add to *httpd.conf*:

```
PerlModule Apache::DBI
```

It is important to load this module before any other `DBI`, `DBD::*` and `ApacheDBI*` modules!

```
db.pl
------------
use DBI ();
use strict;

my $dbh = DBI->connect( 'DBI:mysql:database', 'user', 'password',
                        { autocommit => 0 }
                      ) || die $DBI::errstr;

...rest of the program
```

14.11.1.1 Preopening Connections at the Child Process' Fork Time

If you use `DBI` for DB connections, and you use `Apache::DBI` to make them persistent, it also allows you to preopen connections to the DB for each child with the `connect_on_init()` method, thus saving a connection overhead on the very first request of every child.

```
use Apache::DBI ();
Apache::DBI->connect_on_init("DBI:mysql:test",
                 "login",
                 "passwd",
                 {
               RaiseError => 1,
               PrintError => 0,
               AutoCommit => 1,
                 }
                );
```

This is a simple way to have Apache children establish connections on server startup. This call should be in a startup file `require()d` by `PerlRequire` or inside a <Perl> section. It will establish a connection when a child is started in that child process. See the `Apache::DBI` manpage for the requirements for this method.

14.11.1.2 Caching prepare() Statements

You can also benefit from persistent connections by replacing prepare() with prepare_cached(). That way you will always be sure that you have a good statement handle and you will get some caching benefit. The downside is that you are going to pay for DBI to parse your SQL and do a cache lookup every time you call prepare_cached().

Be warned that some databases (e.g PostgreSQL and Sybase) don't support caches of prepared plans. With Sybase you could open multiple connections to achieve the same result, although this is at the risk of getting deadlocks depending on what you are trying to do!

14.11.2 mod_perl Database Performance Improving

14.11.2.1 Analysis of the Problem

A common web application architecture is one or more application servers which handle requests from client browsers by consulting one or more database servers and performing a transform on the data. When an application must consult the database on every request, the interaction with the database server becomes the central performance issue. Spending a bit of time optimizing your database access can result in significant application performance improvements. In this analysis, a system using Apache, mod_perl, `DBI`, and Oracle will be considered. The application server uses Apache and mod_perl to service client requests, and `DBI` to communicate with a remote Oracle database.

In the course of servicing a typical client request, the application server must retrieve some data from the database and execute a stored procedure. There are several steps that need to be performed to complete the request:

```
1: Connect to the database server
2: Prepare a SQL SELECT statement
3: Execute the SELECT statement
4: Retrieve the results of the SELECT statement
5: Release the SELECT statement handle
6: Prepare a PL/SQL stored procedure call
7: Execute the stored procedure
8: Release the stored procedure statement handle
9: Commit or rollback
10: Disconnect from the database server
```

In this document, an application will be described which achieves maximum performance by eliminating some of the steps above and optimizing others.

14.11.2.2 Optimizing Database Connections

A naive implementation would perform steps 1 through 10 from above on every request. A portion of the source code might look like this:

```
# ...
my $dbh = DBI->connect('dbi:Oracle:host', 'user', 'pass')
  || die $DBI::errstr;

my $baz = $r->param('baz');

eval {
  my $sth = $dbh->prepare(qq{
      SELECT foo
            FROM bar
           WHERE baz = $baz
  });
  $sth->execute;

  while (my @row = $sth->fetchrow_array) {
      # do HTML stuff
  }

  $sth->finish;
```

```
    my $sph = $dbh->prepare(qq{
        BEGIN
            my_procedure(
                arg_in => $baz
            );
        END;
    });
    $sph->execute;
    $sph->finish;

    $dbh->commit;
  };
  if ($@) {
    $dbh->rollback;
  }

  $dbh->disconnect;
  # ...
```

In practice, such an implementation would have hideous performance problems. The majority of the execution time of this program would likely be spent connecting to the database. An examination shows that step 1 is comprised of many smaller steps:

```
1: Connect to the database server
1a: Build client-side data structures for an Oracle connection
1b: Look up the server's alias in a file
1c: Look up the server's hostname
1d: Build a socket to the server
1e: Build server-side data structures for this connection
```

The naive implementation waits for all of these steps to happen, and then throws away the database connection when it is done! This is obviously wasteful, and easily rectified. The best solution is to hoist the database connection step out of the per-request lifecycle so that more than one request can use the same database connection. This can be done by connecting to the database server once, and then not disconnecting until the Apache child process exits. The `Apache::DBI` module does this transparently and automatically with little effort on the part of the programmer.

`Apache::DBI` intercepts calls to `DBI`'s connect and disconnect methods and replaces them with its own. `Apache::DBI` caches database connections when they are first opened, and it ignores disconnect commands. When an application tries to connect to the same database, `Apache::DBI` returns a cached connection, thus saving the significant time penalty of repeatedly connecting to the database. You will find a full treatment of `Apache::DBI` at Persistent DB Connections

When `Apache::DBI` is in use, none of the code in the example needs to change. The code is upgraded from naive to respectable with the use of a simple module! The first and biggest database performance problem is quickly dispensed with.

14.11.2.3 Utilizing the Database Server's Cache

Most database servers, including Oracle, utilize a cache to improve the performance of recently seen queries. The cache is keyed on the SQL statement. If a statement is identical to a previously seen statement, the execution plan for the previous statement is reused. This can be a considerable improvement over building a new statement execution plan.

Our respectable implementation from the last section is not making use of this caching ability. It is preparing the statement:

```
SELECT foo FROM bar WHERE baz = $baz
```

The problem is that `$baz` is being read from an HTML form, and is therefore likely to change on every request. When the database server sees this statement, it is going to look like:

```
SELECT foo FROM bar WHERE baz = 1
```

and on the next request, the SQL will be:

```
SELECT foo FROM bar WHERE baz = 42
```

Since the statements are different, the database server will not be able to reuse its execution plan, and will proceed to make another one. This defeats the purpose of the SQL statement cache.

The application server needs to make sure that SQL statements which are the same look the same. The way to achieve this is to use placeholders and bound parameters. The placeholder is a blank in the SQL statement, which tells the database server that the value will be filled in later. The bound parameter is the value which is inserted into the blank before the statement is executed.

With placeholders, the SQL statement looks like:

```
SELECT foo FROM bar WHERE baz = :baz
```

Regardless of whether `baz` is 1 or 42, the SQL always looks the same, and the database server can reuse its cached execution plan for this statement. This technique has eliminated the execution plan generation penalty from the per-request runtime. The potential performance improvement from this optimization could range from modest to very significant.

Here is the updated code fragment which employs this optimization:

```
# ...
my $dbh = DBI->connect('dbi:Oracle:host', 'user', 'pass')
  || die $DBI::errstr;

my $baz = $r->param('baz');

eval {
  my $sth = $dbh->prepare(qq{
      SELECT foo
        FROM bar
       WHERE baz = :baz
  });
```

```
    $sth->bind_param(':baz', $baz);
    $sth->execute;

    while (my @row = $sth->fetchrow_array) {
        # do HTML stuff
    }

    $sth->finish;

    my $sph = $dbh->prepare(qq{
        BEGIN
            my_procedure(
                arg_in => :baz
            );
        END;
    });
    $sph->bind_param(':baz', $baz);
    $sph->execute;
    $sph->finish;

    $dbh->commit;
  };
  if ($@) {
    $dbh->rollback;
  }
  # ...
```

14.11.2.4 Eliminating SQL Statement Parsing

The example program has certainly come a long way and the performance is now probably much better than that of the first revision. However, there is still more speed that can be wrung out of this server architecture. The last bottleneck is in SQL statement parsing. Every time `DBI`'s prepare() method is called, `DBI` parses the SQL command looking for placeholder strings, and does some housekeeping work. Worse, a context has to be built on the client and server sides of the connection which the database will use to refer to the statement. These things take time, and by eliminating these steps the time can be saved.

To get rid of the statement handle construction and statement parsing penalties, we could use `DBI`'s prepare_cached() method. This method compares the SQL statement to others that have already been executed. If there is a match, the cached statement handle is returned. But the application server is still spending time calling an object method (very expensive in Perl), and doing a hash lookup. Both of these steps are unnecessary, since the SQL is very likely to be static and known at compile time. The smart programmer can take advantage of these two attributes to gain better database performance. In this example, the database statements will be prepared immediately after the connection to the database is made, and they will be cached in package scalars to eliminate the method call.

What is needed is a routine that will connect to the database and prepare the statements. Since the statements are dependent upon the connection, the integrity of the connection needs to be checked before using the statements, and a reconnection should be attempted if needed. Since the routine presented here does everything that `Apache::DBI` does, it does not use `Apache::DBI` and therefore has the added benefit of eliminating a cache lookup on the connection.

Here is an example of such a package:

```
package My::DB;

use strict;
use DBI ();

sub connect {
  if (defined $My::DB::conn) {
      eval {
          $My::DB::conn->ping;
      };
      if (!$@) {
          return $My::DB::conn;
      }
  }

  $My::DB::conn = DBI->connect(
      'dbi:Oracle:server', 'user', 'pass', {
          PrintError => 1,
          RaiseError => 1,
          AutoCommit => 0
      }
  ) || die $DBI::errstr; #Assume application handles this

  $My::DB::select = $My::DB::conn->prepare(q{
      SELECT foo
        FROM bar
       WHERE baz = :baz
  });

  $My::DB::procedure = $My::DB::conn->prepare(q{
      BEGIN
          my_procedure(
              arg_in => :baz
          );
      END;
  });

  return $My::DB::conn;
}

1;
```

Now the example program needs to be modified to use this package.

```
# ...
my $dbh = My::DB->connect;

my $baz = $r->param('baz');

eval {
  my $sth = $My::DB::select;
  $sth->bind_param(':baz', $baz);
  $sth->execute;
```

```
    while (my @row = $sth->fetchrow_array) {
        # do HTML stuff
    }

    my $sph = $My::DB::procedure;
    $sph->bind_param(':baz', $baz);
    $sph->execute;

    $dbh->commit;
  };
  if ($@) {
    $dbh->rollback;
  }
  # ...
```

Notice that several improvements have been made. Since the statement handles have a longer life than the request, there is no need for each request to prepare the statement, and no need to call the statement handle's finish method. Since `Apache::DBI` and the prepare_cached() method are not used, no cache lookups are needed.

14.11.2.5 Conclusion

The number of steps needed to service the request in the example system has been reduced significantly. In addition, the hidden cost of building and tearing down statement handles and of creating query execution plans is removed. Compare the new sequence with the original:

```
1: Check connection to database
2: Bind parameter to SQL SELECT statement
3: Execute SELECT statement
4: Fetch rows
5: Bind parameters to PL/SQL stored procedure
6: Execute PL/SQL stored procedure
7: Commit or rollback
```

It is probably possible to optimize this example even further, but I have not tried. It is very likely that the time could be better spent improving your database indexing scheme or web server buffering and load balancing.

14.12 Using 3rd Party Applications

It's been said that no one can do everything well, but one can do something specific extremely well. This seems to be true for many software applications, when you don't try to do everything but instead concentrate on something specific you can do it really well.

Based on the above introduction, while the mod_perl server can do many many things, there are other applications (or Apache server modules) that can do some specific operations faster or do a really great job for the mod_perl server by unloading it when doing some operations by themselves.

Let’s take a look at a few of these.

14.12.1 Proxying the mod_perl Server

Proxy gives you a great performance increase in most cases. It’s discussed in the section Adding a Proxy Server in http Accelerator Mode.

14.13 Upload and Download of Big Files

You don’t want to tie up your precious mod_perl backend server children doing something as long and simple as transferring a file, especially a big one. The overhead saved by mod_perl is typically under one second, which is an enormous saving for the scripts whose run time is under one second. The user won’t really see any important performance benefits from mod_perl, since the upload may take up to several minutes.

If some particular script’s main functionality is the uploading or downloading of big files, you probably want it to be executed on a plain apache server under mod_cgi (i.e. performing this operation on the front-end server, if you use a dual-server setup.

This of course assumes that the script requires none of the functionality of the mod_perl server, such as custom authentication handlers.

14.14 Apache/mod_perl Build Options

It’s important how you build mod_perl enabled Apache. It influences the size of the httpd executable, some irrelevant modules might slow the performance.

[ReaderMETA: Any other building time things that influence performance?]

14.14.1 mod_perl Process Size as a Function of Compiled in C Modules and mod_perl Features

You might wonder whether it’s better to compile in only the required modules and mod_perl hooks, or it doesn’t really matter. To answer on this question lets first make a few compilation and compare the results.

So we are going to build mod_perl starting with:

```
% perl Makefile.PL APACHE_SRC=../apache_x.x.x/src \
       DO_HTTPD=1 USE_APACI=1
```

and followed by one of these option groups:

1. **Default**

no arguments

2. **Minimum**

```
APACI_ARGS='--disable-module=env, \
            --disable-module=negotiation, \
            --disable-module=status, \
            --disable-module=info, \
            --disable-module=include, \
            --disable-module=autoindex, \
            --disable-module=dir, \
            --disable-module=cgi, \
            --disable-module=asis, \
            --disable-module=imap, \
            --disable-module=userdir, \
            --disable-module=access, \
            --disable-module=auth'
```

3. **Everything**

```
EVERYTHING=1
```

4. **Everything + Debug**

```
EVERYTHING=1 PERL_DEBUG=1
```

After re-compiling with arguments of each of these groups, we can summarize the results:

```
Build group    httpd size (bytes)  Difference
---------------------------------------------
Minimum              892928         +     0
Default              994316         +101388
Everything          1044432         +151504
Everything+Debug    1162100         +269172
```

Indeed when you strip most of the default things, the server size is slimmer. But the savings are insignificant since you don't multiply the added size by the number of child processes if your OS supports sharing memory. The parent processes is a little bigger, but it shares these memory pages with its child processes. Of course not everything will be shared, if some module you add does some process memory modification particular to the process, but the most will.

And of course this was just an example to show the difference is size. It doesn't mean that you can everything away, since there will be Apache modules and mod_perl options that you won't be able to work without.

But as a good system administrator's rule says: *"Run the absolute minimum of the applications. If you don't know or need something, disable it"*. Following this rule to decide on the required Apache components and disabling the unneeded default components, makes you a good Apache administrator.

14.15 Perl Build Options

The Perl interpreter lays in the brain of the mod_perl server and if we can optimize perl into doing things faster under mod_perl we make the whole server faster. Generally, optimizing the Perl interpreter means enabling or disabling some command line options. Let's see a few important ones.

14.15.1 -DTWO_POT_OPTIMIZE and -DPACK_MALLOC Perl Build Options

Newer Perl versions also have build time options to reduce runtime memory consumption. These options might shrink the size of your httpd by about 150k -- quite a big number if you remember to multiply it by the number of children you use.

The `-DTWO_POT_OPTIMIZE` macro improves allocations of data with size close to a power of two; but this works for big allocations (starting with 16K by default). Such allocations are typical for big hashes and special-purpose scripts, especially image processing.

Perl memory allocation is by bucket with sizes close to powers of two. Because of these the malloc() overhead may be big, especially for data of size exactly a power of two. If `PACK_MALLOC` is defined, perl uses a slightly different algorithm for small allocations (up to 64 bytes long), which makes it possible to have overhead down to 1 byte for allocations which are powers of two (and appear quite often).

Expected memory savings (with 8-byte alignment in `alignbytes`) is about 20% for typical Perl usage. Expected slowdown due to additional malloc() overhead is in fractions of a percent and hard to measure, because of the effect of saved memory on speed.

You will find these and other memory improvement details in `perl5004delta.pod`.

Important: both options are On by default in perl versions 5.005 and higher.

14.15.2 -Dusemymalloc Perl Build Option

You have a choice to use the native or Perl's own malloc() implementation. The choice depends on your Operating System. Unless you know which of the two is better on yours, you better try both and compare the benchmarks.

To build without Perl's malloc(), you can use the Configure command:

```
% sh Configure -Uusemymalloc"
```

Note that:

```
-U == undefine usemymalloc (use system malloc)
-D == define   usemymalloc (use Perl's malloc)
```

It seems that Linux still defaults to system malloc so you might want to configure Perl with -Dusemymalloc. Perl's malloc is not much of a win under linux, but makes a **huge** difference under Solaris.

14.16 Architecture Specific Compile Options

When you build Apache and Perl you can optimize the compiled applications to take the benefits of your machine's architecture.

Everything depends on the kind of compiler that you use, the kind of CPU and

For example if you use gcc(1) you might want to use:

- *-march=pentium* if you have a pentium CPU
- *-march=pentiumpro* for pentiumpro and above (but the binary won't run on i386)
- *-fomit-frame-pointer* makes extra register available but disables debugging
- you can try these options were reported to improve the performance: *-ffast-math*, *-malign-double*, *-funroll-all-loops*, *-fno-rtti*, *-fno-exceptions*.

  see the gcc(1) manpage for the details about these
- and of course you may want to change the usually default `-02` flag with a higher number like *-O3*. *-OX* (where X is a number between 1 and 6) defines a collection of various optimization flags, the higher the number the more flags are bundled. The gcc man page will tell you what flags are used for each number.

Test your applications thoroughly when you change the default optimization flags, especially when you go beyond `-02`. It's possible that the optimization will make the code work incorrectly and/or cause segmentation faults.

See your preferred compiler's man page for detailed information about optimization.

14.17 Maintainers

Maintainer is the person(s) you should contact with updates, corrections and patches.

- Stas Bekman [http://stason.org/]

14.18 Authors

- Stas Bekman [http://stason.org/]

Only the major authors are listed above. For contributors see the Changes file.

15 Frequent mod_perl problems

15.1 Description

Some problems come up very often on the mailing list. If there is some important problem that is being reported frequently on the list which is not included below, even if it is found elsewhere in the Guide, please report to mod_perl mailing list.

15.2 my () scoped variable in nested subroutines

See the section "my () Scoped Variable in Nested Subroutines".

15.3 Segfaults caused by PerlFreshRestart

See the section Evil things might happen when using PerlFreshRestart

15.4 Maintainers

Maintainer is the person(s) you should contact with updates, corrections and patches.

- Stas Bekman [http://stason.org/]

15.5 Authors

- Stas Bekman [http://stason.org/]

Only the major authors are listed above. For contributors see the Changes file.

16 Warnings and Errors Troubleshooting Index

16.1 Description

If you encounter an error or warning you don’t understand, check this chapter for solutions to common warnings and errors that the mod_perl community has encountered in the last few years.

16.2 General Advice

Perl’s warnings mode is immensely helpful in detecting possible problems. Make sure you always turn on warnings while you are developing code. See The Importance of Warnings.

Enabling `use diagnostics;` generally helps you to determine the source of the problem and how to solve it. See diagnostics pragma for more information.

16.3 Building and Installation

16.3.1 Bareword "gensym" not allowed while "strict subs"

When building mod_perl one may get the following failure:

```
make[1]: Entering directory ‘.../mod_perl-1.29/Symbol’
Running Mkbootstrap for Apache::Symbol ()
...
Bareword "gensym" not allowed while "strict subs" in use at
/usr/lib/perl5/5.8.3/Pod/Parser.pm line 1158.
...
make: *** [subdirs] Error 2
```

That happens when you you’ve modify `@INC` to push `"."` before all other directories, whereas it should be last. When `"."` is first in `@INC` it picks *mod_perl-1.29/Symbol/Symbol.pm* when it’s inside the directory *mod_perl-1.29/Symbol*. This shouldn’t happen if your system paths are coming first. To check your `@INC`, run:

```
% perl -V
```

and it’ll appear at the end of the output. Usually `"."` is pushed first by setting a `PERL5LIB` or a similar environment variable. Unset it and repeat the build process to solve the problem.

16.3.2 `No rule to make target ... CORE/config.h`

If while running `'make'` you get a message:

```
make: *** No rule to make target
‘/usr/lib/perl5/5.8.3/i386-linux/CORE/config.h’,
needed by ‘Makefile’.  Stop.
```

That means that your Perl installation is incomplete. Usually this is the case on package based distros, where perl is split across multiple packages. Usually you need to install the Perl-devel package to be able to build any other Perl modules that include XS extensions.

16.4 make test Issues

See make Troubleshooting and make test Troubleshooting

16.5 Configuration and Startup

This section talks about errors reported when you attempt to start the server.

16.5.1 'relocation errors' or 'undefined symbol'

Although the httpd executable was successfully built, you can still have make failures. The output could include the following errors:

```
relocation error:
undefined symbol: PL_dowarn
```

This class of errors is often due to multiple installations of Perl. Having *libperl.so* in */usr/lib/* is a great reason for lots of obscure problems, when you have one more perl installed elsewhere. That's why perl puts its *libperl.so* under its private tree (e.g., */usr/lib/perl5/5.8.3/i686/CORE/*). But some distributions decide to put it along with the rest of system libraries, not expecting that users will have extra perl installations.

To resolve the problem you need to check what perl library the application finds. This is easy to check with the help of `ldd(1)`.

If your mod_perl is statically linked with httpd, you need to check the httpd executable whether it's linked against *libperl.so*:

```
% ldd /path/to/apache/bin/httpd
```

If the output includes *libperl.so* check that the path is to the version of Perl you've built mod_perl with. If your httpd executable is reported to link against the wrong *libperl.so* file, you've found the cause of the problem. You should either ask your distributor to not put the perl library into the global system libs directory, or use some other solution to force the loading of the right library, which is usually very platform specific. For example on Linux one can preload a specific library path using the `LD_PRELOAD` environment variable. So if the wanted library is located at */usr/lib/perl5/5.8.3/i686/CORE/libperl.so* you can make httpd use it with:

```
% LD_PRELOAD=/usr/lib/perl5/5.8.3/i686/CORE/libperl.so httpd
```

for more information on Linux loader referer to the `ld.so(8)` `ldconfig(8)` manpages.

If mod_perl is built as DSO, you will need to check the mod_perl module (and not `httpd`) with `ldd`. Confusingly, mod_perl 1.0 module name is the same as of Perl: *libperl.so*. If for example the mod_perl module is located in */path/to/apache/libexec/*, the command would be:

```
% ldd /path/to/apache/libexec/libperl.so
```

There could be another variation of the problem where a Perl used to build mod_perl is statically linked and again, during the build time a wrong static archive (e.g., */usr/lib/libperl.a*) is picked by the linker. If Perl is statically linked, running:

```
% perl -V:useshrplib
```

will say:

```
useshrplib='false'
```

Again, the solution may vary from system to system, but moving */usr/lib/libperl.a* away while building mod_perl is probably the simplest.

16.5.2 SegFaults During Startup

Possible reasons and solutions:

- You have to build mod_perl with the same compiler as Perl was built with. Otherwise you may see segmentation faults and other weird things happen.

- This could be the XSLoader vs. DynaLoader problem, where the list of dl_handles created by XSLoader is wiped out by DynaLoader. Try adding:

  ```
  use DynaLoader ();
  ```

  to your *startup.pl* before any other module is loaded.

  This has been resolved in perl 5.8.0. For earlier versions of Perl you need to comment out:

  ```
  #@dl_librefs         = ();       # things we have loaded
  #@dl_modules         = ();       # Modules we have loaded
  ```

  in *DynaLoader.pm*.

However if none of this cases applies and you still experience segmentation faults, please report the problem using the following guidelines.

16.5.3 libexec/libperl.so: open failed: No such file or directory

If when you run the server you get the following error:

```
libexec/libperl.so: open failed: No such file or directory
```

it probably means that Perl was compiled with a shared library. mod_perl does detect this and links the Apache executable to the Perl shared library (*libperl.so*).

First of all make sure you have Perl installed on the machine, and that you have *libperl.so* in *<perl-root>/<version>/<architecture>/CORE*. For example in */usr/local/lib/perl5/5.00503/sun4-solaris/CORE*.

Then make sure that directory is included in the environment variable `LD_LIBRARY_PRELOAD`. Under normal circumstances, Apache should have the path configured at compile time, but this way you can override the library path.

16.5.4 install_driver(Oracle) failed: Can't load '.../DBD/Oracle/Oracle.so' for module DBD::Oracle

```
install_driver(Oracle) failed: Can't load
'/usr/lib/perl5/site_perl/5.005/i386-linux/auto/DBD/Oracle/Oracle.so'
for module DBD::Oracle:
libclntsh.so.8.0: cannot open shared object file:
No such file or directory at
/usr/lib/perl5/5.00503/i386-linux/DynaLoader.pm line 169.
at (eval 27) line 3
Perhaps a required shared
library or dll isn't installed where expected at
/usr/local/apache/perl/tmp.pl line 11
```

On BSD style filesystems `LD_LIBRARY_PATH` is not searched for setuid programs (a.k.a., Apache). This isn't a problem for CGI script since they don't do a setuid (and are forked off), but Apache does, and mod_perl is in Apache. Therefore the first solution is to explicitly load the library from the system wide *ldconfig* configuration file:

```
# echo $ORACLE_HOME/lib >> /etc/ld.so.conf
# ldconfig
```

Another solution to this problem is to modify the resulting *Makefile* (after running `perl Makefile.PL`) as follows:

1. search for the line `LD_RUN_PATH=`

2. replace it with `LD_RUN_PATH=(my_oracle_home)/lib`

(my_oracle_home) is, of course, the home path to your oracle installation. In particular, the file libclntsh.so.8.0 should exist in that directory. (If you use CPAN, the build directory for DBD::Oracle should be in ~/.cpan/build/DBD-Oracle-1.06/ if you're logged in as root.)

Then, just type `make install`, and all should go well.

FYI, setting `LD_RUN_PATH` has the effect of hard-coding the path to *(my_oracle_home)/lib* in the resulting `Oracle.so` file generated by the `DBD::Oracle` so that at run-time, it doesn't have to go searching through `LD_LIBRARY_PATH` or the default directories used by `ld`.

For more information see the *ld* manpage and an essay on `LD_LIBRARY_PATH`: http://www.visi.com/~barr/ldpath.html

16.5.5 Invalid command 'PerlHandler'...

```
Syntax error on line 393 of /etc/httpd/conf/httpd.conf: Invalid
command 'PerlHandler', perhaps mis-spelled or defined by a module
not included in the server configuration [FAILED]
```

This can happen when you have a mod_perl enabled Apache compiled with DSO (generally it's an installed RPM or other binary package) but the mod_perl module isn't loaded. In this case you have to tell Apache to load mod_perl by adding:

```
AddModule mod_perl.c
```

in your *httpd.conf*.

This can also happen when you try to run a non-mod_perl Apache server using the configuration from a mod_perl server.

16.5.6 symbol __floatdisf: referenced symbol not found

This problem is experienced by users on certain Solaris versions. When the server is built with modules that use the `__floatdisf` symbol it can't be started. e.g.:

```
Cannot load /usr/local/apache/libexec/libproxy.so into server:
ld.so.1: ../bin/httpd: fatal: relocation error: file
/usr/local/apache/libexec/libproxy.so: symbol __floatdisf: referenced
symbol not found
```

The missing symbol is in *libgcc.a*. Use

```
% gcc -print-libgcc-file-name
```

to see where that file is. Once found you have to relink the module with that file. You can also look for it in the gcc tree, e.g. under *gcc-3.2.1/gcc*.

First, configure and install Apache. Next, relink *mod_proxy.so* or *mod_negotiation.so*, or whatever the module that reports the problem with *libgcc.a*.

```
% cd apache_1.3.27/src/modules
% ld -G -o mod_proxy.so mod_proxy.lo /pathto/libgcc.a
```

(adjust the */pathto/* to point to the right file from the gcc output stage.)

You can now verify with `nm` that *mod_proxy.so* includes that symbol.

16.5.7 RegistryLoader: Translation of uri [...] to filename failed

```
RegistryLoader: Translation of uri [/home/httpd/perl/test.pl] to filename
        failed [tried: /home/httpd/docs/home/httpd/perl/test.pl]
```

This error shows up when `Apache::RegistryLoader` fails to translate the URI into the corresponding filesystem path. Most failures happen when one passes a file path instead of URI. (A reminder: */home/httpd/perl/test.pl* is a file path, while */perl/test.pl* is a URI). In most cases all you have to do is to pass something that `Apache::RegistryLoader` expects to get - the URI, but there are more complex cases. `Apache::RegistryLoader`'s man page shows how to handle these cases as well (look for the trans() sub).

16.5.8 "Apache.pm failed to load!"

If your server startup fails with:

```
Apache.pm failed to load!
```

try adding this to *httpd.conf*:

```
PerlModule Apache
```

16.6 Code Parsing and Compilation

16.6.1 Value of $x will not stay shared at - line 5

my () Scoped Variable in Nested Subroutines.

16.6.2 Value of $x may be unavailable at - line 5.

my () Scoped Variable in Nested Subroutines.

16.6.3 Can't locate loadable object for module XXX

There is no object built for this module. e.g. when you see:

```
Can't locate loadable object for module Apache::Util in @INC...
```

make sure to give mod_perl's `Makefile.PL` `PERL_UTIL_API=1`, `EVERYTHING=1` or `DYNAMIC=1` parameters to enable and build all the components of `Apache::Util`.

16.6.4 Can't locate object method "get_handlers"...

```
Can't locate object method "get_handlers" via package "Apache"
```

You need to rebuild your mod_perl with stacked handlers, i.e. `PERL_STACKED_HANDLERS=1` or more simply `EVERYTHING=1`.

16.6.5 Missing right bracket at line ...

Most often you will find that you really do have a syntax error. However the other reason might be that a script running under `Apache::Registry` is using `__DATA__` or `__END__` tokens. Learn why.

16.6.6 Can't load '.../auto/DBI/DBI.so' for module DBI

Check that all your modules are compiled with the same Perl that is compiled into mod_perl. Perl 5.005 and 5.004 are not binary compatible by default.

Other known causes of this problem:

OS distributions that ship with a broken binary Perl installation.

The 'perl' program and 'libperl.a' library are somehow built with different binary compatibility flags.

The solution to these problems is to rebuild Perl and any extension modules from a fresh source tree. Tip for running Perl's Configure script: use the '-des' flags to accepts defaults and '-D' flag to override certain attributes:

```
% ./Configure -des -Dcc=gcc ... && make test && make install
```

Read Perl's INSTALL document for more details.

Solaris OS specific:

"`Can't load DBI`" or similar error for the IO module or whatever dynamic module mod_perl tries to pull in first. The solution is to re-configure, re-build and re-install Perl and dynamic modules with the following flags when Configure asks for "`additional LD flags`":

```
-Xlinker --export-dynamic
```

or

```
-Xlinker -E
```

This problem is only known to be caused by installing gnu ld under Solaris.

16.7 Runtime

16.7.1 print() doesn’t send anything on Mac OS X

On certain Mac OS X builds mod_perl doesn’t seem to able to print() anything to the client. That’s because `STDOUT` is not tied to the `Apache` module. Most likely some core module on Mac OS X untie’s/re-tie’s `STDOUT` after mod_perl had it tied. The workaround is to add:

```
PerlHeaderParserHandler "sub { tie *STDOUT, 'Apache' unless tied *STDOUT; }"
```

Another solution is not to use un unqualified `print()`, but `$r->print()`, for example replace:

```
print "Hello";
```

with:

```
$r->print("Hello");
```

If you don’t have `$r`, you can obtain it with:

```
my $r = Apache->request;
```

16.7.2 "exit signal Segmentation fault (11)" with mysql

If you build mod_perl and mod_php in the same binary, you might get Segmentation fault followed by this error:

```
exit signal Segmentation fault (11)
```

The solution is to not rely on PHP’s built-in MySQL support, and instead build mod_php with your local MySQL support files by adding `--with-mysql=/path/to/mysql` during `./configure`.

16.7.3 foo ... at /dev/null line 0

Under mod_perl you may receive a warning or an error in the *error_log* which specifies */dev/null* as the source file, and line 0 as an line number where the printing of the message was triggered. This is absolutely normal if the code is executed from within a handler, because there is no actual file associated with the handler. Therefore `$0` is set to */dev/null* and that’s what you see.

16.7.4 Preventing mod_perl Processes From Going Wild

See the sections "Non-Scheduled Emergency Log Rotation" and "All RAM Consumed"

16.7.5 Segfaults when using XML::Parser

If you have some of the processes segfault when using `XML::Parser` you should use

```
--disable-rule=EXPAT
```

during the Apache configuration step.

Starting from mod_perl version 1.23 this option is disabled by default.

16.7.6 My CGI/Perl Code Gets Returned as Plain Text Instead of Being Executed by the Webserver

See My CGI/Perl Code Gets Returned as Plain Text Instead of Being Executed by the Webserver.

16.7.7 Incorrect line number reporting in error/warn log messages

See Use of uninitialized value at (eval 80) line 12.

16.7.8 rwrite returned -1

This message happens when the client breaks the connection while your script is trying to write to the client. With Apache 1.3.x, you should only see the rwrite messages if `LogLevel` is set to `debug`.

There was a bug that reported this debug message regardless of the value of the `LogLevel` directive. It was fixed in mod_perl 1.19_01.

Generally `LogLevel` is either `debug` or `info`. `debug` logs everything, `info` is the next level, which doesn't include debug messages. You shouldn't use "debug" mode on your production server. At the moment there is no way to prevent users from aborting connections.

16.7.9 Can't upgrade that kind of scalar ...

Fixed in mod_perl 1.23.

16.7.10 caught SIGPIPE in process

```
[modperl] caught SIGPIPE in process 1234
[modperl] process 1234 going to Apache::exit with status...
```

That's the `$SIG{PIPE}` handler installed by mod_perl/`Apache::SIG`, which is called if a connection times out or if the client presses the 'Stop' button. It gives you an opportunity to do cleanups if the script was aborted in the middle of its execution. See Handling the 'User pressed Stop button' case for more info.

If your mod_perl version is earlier than 1.17 you might also get the message in the following section...

16.7.11 Client hit STOP or Netscape bit it!

```
Client hit STOP or Netscape bit it!
Process 2493 going to Apache::exit with status=-2
```

You may see this message in mod_perl versions less than 1.17. See also caught SIGPIPE in process.

16.7.12 Global symbol "$foo" requires explicit package name

The script below will print a warning like that above, moreover it will print the whole script as a part of the warning message:

```
#!/usr/bin/perl -w
use strict;
print "Content-type: text/html\n\n";
print "Hello $undefined";
```

The warning:

```
Global symbol "$undefined" requires
explicit package name at /usr/apps/foo/cgi/tmp.pl line 4.
        eval 'package Apache::ROOT::perl::tmp_2epl;
use Apache qw(exit);sub handler {
#line 1 /usr/apps/foo/cgi/tmp.pl
BEGIN {$^W = 1;}#!/usr/bin/perl -w
use strict;
print "Content-type: text/html\\n\\n";
print "Hello $undefined";


}
;' called at
/usr/apps/lib/perl5/site_perl/5.005/aix/Apache/Registry.pm
line 168
        Apache::Registry::compile('package
  Apache::ROOT::perl::tmp_2epl;use Apache qw(exit);sub han...')
  called at
      /usr/apps/lib/perl5/site_perl/5.005/aix/Apache/Registry.pm
      line 121
      Apache::Registry::handler('Apache=SCALAR(0x205026c0)')
      called at /usr/apps/foo/cgi/tmp.pl line 4
        eval {...} called at /usr/apps/foo/cgi/tmp.pl line 4
[Sun Nov 15 15:15:30 1998] [error] Undefined subroutine
&Apache::ROOT::perl::tmp_2epl::handler called at
/usr/apps/lib/perl5/site_perl/5.005/aix/Apache/Registry.pm
line 135.

[Sun Nov 15 15:15:30 1998] [error] Goto undefined subroutine
&Apache::Constants::SERVER_ERROR at
/usr/apps/lib/perl5/site_perl/5.005/aix/Apache/Constants.pm
line 23.
```

The error is simple to fix. When you use the `use strict;` pragma (and you should...), Perl will insist that all variables are defined before being used, so the error will not arise.

The bad thing is that sometimes the whole script (possibly, thousands of lines) is printed to the *error_log* file as code that the server has tried to `eval()`uate.

May be you have a `$SIG{__DIE__}` handler installed (`Carp::confess()`?). If so that's what's expected.

You might wish to try something more terse such as "local $SIG{__WARN__} = \&Carp::cluck;" The confess method is *very* verbose and will tell you more than you might wish to know including full source.

16.7.13 Use of uninitialized value at (eval 80) line 12.

Your code includes some undefined variable that you have used as if it was already defined and initialized. For example:

```
$param = $q->param('test');
print $param;
```

vs.

```
$param = $q->param('test') || '';
print $param;
```

In the second case, `$param` will always be *defined*, either with `$q->param('test')`'s return value or the default value (`''` empty string in our example).

Also read about Finding the Line Which Triggered the Error or Warning.

16.7.14 Undefined subroutine &Apache::ROOT::perl::test_2epl::some_function called at

See Names collisions with Modules and libs.

16.7.15 Callback called exit

Callback called exit is just a generic message when some unrecoverable error occurs inside Perl during `perl_call_sv()` (which mod_perl uses to invoke all handler subroutines. Such problems seem to occur far less with 5.005_03 than 5.004.

Sometimes you discover that your server is not responding and its error_log has filled up the remaining space on the file system. When you get to see the contents of the error_log -- it includes millions of lines, like:

```
Callback called exit at -e line 33, <HTML> chunk 1.
```

Why the looping?

Perl can get *very* confused inside an infinite loop in your code. It doesn't necessarily mean that your code did call `exit()`. Perl's malloc went haywire and called `croak()`, but no memory is left to properly report the error, so Perl is stuck in a loop writing that same message to stderr.

Perl 5.005+ plus is recommended for its improved malloc.c and other features that improve mod_perl and are turned on by default.

See also Out of memory

16.7.16 Out of memory!

If something goes really wrong with your code, Perl may die with an "Out of memory!" message and/or "Callback called exit". Common causes of this are never-ending loops, deep recursion, or calling an undefined subroutine.

If you are using perl 5.005 or later, and perl is compiled to use it's own malloc rutines, you can trap out of memory errors by setting aside an extra memory pool in the special variable `$^M`. By default perl uses the operating system malloc for many popular systems, so unless you build perl with 'usemymalloc=y' you probably wont be able to use `$^M`. Run:

```
% perl -V:usemymalloc
```

if your mod_perl was compiled against perl which uses internal `malloc()` the answer will be 'y'.

Here is an explanation of `$^M` from the `perlvar` manpage:

```
By default, running out of memory is an untrap- pable, fatal
error.  However, if suitably built, Perl can use the contents of
$^M as an emergency memory pool after die()ing.  Suppose that
your Perl were compiled with -DPERL_EMERGENCY_SBRK and used
Perl's malloc.  Then

    $^M = 'a' x (1 << 16);

would allocate a 64K buffer for use in an emer- gency.  See the
INSTALL file in the Perl distribu- tion for information on how
to enable this option.  To discourage casual use of this
advanced feature, there is no English long name for this
variable.
```

If your perl installation supports $^M and you add '`use Apache::Debug level => 4;`' to your Perl script, it will allocate the `$^M` emergency pool and the `$SIG{__DIE__}` handler will call `Carp::confess`, giving you a stack trace which should reveal where the problem is. See the `Apache::Resource` module for ways to control httpd processes.

Note that Perl 5.005 and later have `PERL_EMERGENCY_SBRK` turned on by default.

Another trick is to have a startup script initialize `Carp::confess`, like so:

```
use Carp ();
eval { Carp::confess("init") };
```

this way, when the real problem happens, `Carp::confess` doesn't eat memory in the emergency pool (`$^M`).

Some other mod_perl users have reported that this works well for them:

```
# Allocate 64K as an emergency memory pool for use in out of
# memory situation
$^M = 0x00 x 65536;

# Little trick to initialize this routine here so that in the case
# of OOM, compiling this routine doesn't eat memory from the
# emergency memory pool $^M
use CGI::Carp ();
eval { CGI::Carp::confess('init') };

# Importing CGI::Carp sets $main::SIG{__DIE__} = \&CGI::Carp::die;
# Override that to additionally give a stack backtrace
$main::SIG{__DIE__} = \&CGI::Carp::confess;
```

Discussion of `$^M` has come up on PerlMonks, and there is speculation that `$^M` is a forgotten feature that's not well supported. See http://perlmonks.org/index.pl?node_id=287850 for more information.

16.7.17 server reached MaxClients setting, consider raising the MaxClients setting

See Choosing MaxClients.

16.7.18 syntax error at /dev/null line 1, near "line arguments:"

```
syntax error at /dev/null line 1, near "line arguments:"
Execution of /dev/null aborted due to compilation errors.
parse: Undefined error: 0
```

There is a chance that your `/dev/null` device is broken. Try:

```
% echo > /dev/null
```

Alternatively you should try to remove this special file and recreate it:

```
# rm /dev/null
# mknod /dev/null c 1 3
# chmod a+rw /dev/null
```

16.7.19 Can’t call method "register_cleanup" (CGI.pm)

```
Can’t call method "register_cleanup" on an
undefined value at /usr/lib/perl5/5.00503/CGI.pm line 263.
```

caused by this code snippet in *CGI.pm*:

```
if ($MOD_PERL) {
  Apache->request->register_cleanup(\&CGI::_reset_globals);
  undef $NPH;
}
```

One solution is to add to *httpd.conf*:

```
PerlPostReadRequestHandler ’sub { Apache->request(shift) }’
```

But even better, switch to `Apache::Cookie`:

```
use Apache;
use Apache::Cookie;

sub handler {
  my $r = shift;
  my $cookies = Apache::Cookie->new($r)->parse;
  my %bar = $cookies->{foo}->value;
}
```

16.7.20 readdir() not working

If readdir() either fails with an exception, or in the list context it returns the correct number of items but each item as an empty string, you have a binary compatibility between mod_perl and Perl problem. Most likely the two have been built against different *glibc* versions, which have incompatible `struct dirent`.

To solve this problem rebuild mod_perl and Perl against the same *glibc* version or get new binary packages built against the same *glibc* version.

16.8 Shutdown and Restart

16.8.1 Evil things might happen when using PerlFreshRestart

`PerlFreshRestart` affects the graceful restart process for non-DSO mod_perl builds. If you have mod_perl enabled Apache built as DSO and you restart it, the whole Perl interpreter is completely torn down (perl_destruct())and restarted. The value of `PerlFreshRestart` is irrelevant at this point.

Unfortunately, not all perl modules are robust enough to survive reload. For them this is an unusual situation. `PerlFreshRestart` does not much more than:

```
while (my ($k,$v) = each %INC) {
  delete $INC{$k};
  require $k;
}
```

Besides that, it flushes the `Apache::Registry` cache, and empties any dynamic stacked handlers (e.g. `PerlChildInitHandler`).

Some users, who had turned `PerlFreshRestart` **On**, reported having segfaults, others have seen no problems starting the server, no errors written to the logs, but no server running after a restart. Most of the problems have gone away when it was turned `Off`.

It doesn't mean that you shouldn't use it, if it works for you. Just beware of the dragons...

16.8.2 Constant subroutine XXX redefined

That's a mandatory warning inside Perl which happens only if you modify your script and Apache::Registry reloads it. Perl is warning you that the subroutine(s) were redefined. It is mostly harmless. If you don't like seeing these warnings, just `kill -USR1` (graceful restart) Apache when you modify your scripts.

You aren't supposed to see these warnings if you don't modify the code with perl 5.004_05 or 5.005+.and higher. If you still experience a problem with code within a CGI script, moving all the code into a module (or a library) and require()ing it should solve the problem.

16.8.3 Can't undef active subroutine

```
Can't undef active subroutine at
/usr/apps/lib/perl5/site_perl/5.005/aix/Apache/Registry.pm line 102.
Called from package Apache::Registry, filename
/usr/apps/lib/perl5/site_perl/5.005/aix/Apache/Registry.pm, line 102
```

This problem is caused when a client drops the connection while httpd is in the middle of a write. httpd times out, sending a SIGPIPE, and Perl (in that child) is stuck in the middle of its eval context. This is fixed by the Apache::SIG module which is called by default. This should not happen unless you have code that is messing with **$SIG{PIPE}**. It's also triggered only when you've changed your script on disk and mod_perl is trying to reload it.

16.8.4 [warn] child process 30388 did not exit, sending another SIGHUP

From mod_perl.pod: With Apache versions 1.3.0 and higher, mod_perl will call the perl_destruct() Perl API function during the child exit phase. This will cause proper execution of `END` blocks found during server startup as well as invoking the `DESTROY` method on global objects which are still alive. It is possible that this operation may take a long time to finish, causing problems during a restart. If your code does not contain any `END` blocks or `DESTROY` methods which need to be run during child server shutdown, this destruction can be avoided by setting the `PERL_DESTRUCT_LEVEL` environment variable to `-1`. Be aware that 'your code' includes any modules you use and *they* may well have `DESTROY` and `END` blocks...

16.8.5 Processes Get Stuck on Graceful Restart

If you see a process stuck in "G" (Gracefully finishing) after a doing a graceful restart (sending kill -SIGUSR1) it means that the process is hanging in perl_destruct() while trying to cleanup. This cleanup normally isn't a requirement, you can disable it by setting the PERL_DESTRUCT_LEVEL environment variable to -1. See the section "Speeding up the Apache Termination and Restart" for more information.

16.8.6 httpd keeps on growing after each restart

See the *HUP Signal* explanation at the section: Server Stopping and Restarting

Also, be aware that `<Perl>` sections can also cause leaks during graceful restarts. See the (sub)thread: http://aspn.activestate.com/ASPN/Mail/Message/modperl/304620 (META: we don't have code samples, so if you have such please let us know).

16.9 OS Specific Notes

16.9.1 RedHat Linux

16.9.1.1 RH 8 and 9 Locale Issues

RedHat 8 and 9 ship Perl with a locale setting that breaks perl. To solve the problem either change the locale to en_US.ISO8859-1 or simply remove the UTF8 part.

For more information refer to: http://twiki.org/cgi-bin/view/Codev/UsingPerl58OnRedHat8 http://bugzilla.redhat.com/bugzilla/show_bug.cgi?id=87682

16.9.2 OpenBSD

16.9.2.1 Issues Caused by httpd being `chroot`'ed

Since OpenBSD 3.2, the Apache `httpd(8)` server has been `chroot(2)`ed by default.

A mod_perl enabled OpenBSD httpd will NOT work as is. A proper `chroot(2)` environment must be setup for the default chrooted behavior to work. The `-u` option to httpd disables the chroot behavior, and returns the httpd to the expanded "unsecure" behavior. See the OpenBSD `httpd(8)` man page.

For more information see *section 10.16 of the OpenBSD FAQ*.
http://www.openbsd.org/faq/faq10.html#httpdchroot.

16.9.3 Win FU

16.9.3.1 Apache::DBI

`Apache::DBI` causes the server to exit when it starts up, with:

```
[Mon Oct 25 15:06:11 1999] file .\main\http_main.c, line 5890,
assertion "start_mutex" failed
```

Solution: build mod_perl with `PERL_STARTUP_DONE_CHECK` set (e.g. insert

```
#define PERL_STARTUP_DONE_CHECK 1
```

at the top of *mod_perl.h* or add it to the defines in the MSVC++ and similar applications' `Options` dialog).

Apache loads all Apache modules twice, to make sure the server will successfully restart when asked to. This flag disables all `PerlRequire` and `PerlModule` statements on the first load, so they can succeed on the second load. Without that flag, the second load fails.

16.9.4 HP-UX

16.9.4.1 `Apache::Status` Dumps Core Under HP-UX 10.20

HP-UX 10.20 may dump core when accessing *perl-status?sig*. This issue is known to happen when perl's *./Configure* doesn't detect that `SIGRTMAX` is defined, but not implemented on that platform.

One solution is to upgrade to a recent version of Perl (5.8.0?) that properly detects the implementation of that signal.

Another solution is to modify *Apache/Status.pm* to skip that broken signal by replacing the line:

```
sort keys %SIG),
```

with:

```
sort grep { $_ ne 'RTMAX' } keys %SIG),
```

16.10 Problematic Perl Modules

Here is the list of Perl Modules that are known to have problems under mod_perl 1.0 and possible workarounds, solutions.

- **`IPC::Open*`**

  use `IPC::Run` instead. It provides the same functionality as the `IPC::Open*` family and more... and it works fine with mod_perl.

16.11 Maintainers

Maintainer is the person(s) you should contact with updates, corrections and patches.

- Stas Bekman [http://stason.org/]

16.12 Authors

- Stas Bekman [http://stason.org/]

Only the major authors are listed above. For contributors see the Changes file.

17 Debugging mod_perl

17.1 Description

Tired of Internal Server Errors? Find out how to debug your mod_perl applications, thanks to a number of features of Perl and mod_perl.

17.2 Warning and Errors Explained

Let’s talk first about things that bother most web (and non-web) programmers. *The bothering things* are warning and errors reported by Perl. We are going to learn how to take the best out of both, by turning this obvious to the newbie programmer enemies into our best friends.

17.2.1 Curing The "Internal Server Error"

You have just installed this new CGI script and when you try it out you see the grey screen of death saying "Internal Server Error"... Or even worse you have a script running on a production server for a long time without problems, when the same grey screen starts to show up occasionally for no apparent reason.

How can we find out what the problem is?

First problem:

You have been coding in Perl for years, and whenever an error occurred in the past it was displayed in the same terminal window that you started the script from. But when you work with a webserver there is no terminal to show you the errors, since the server in most cases has no terminal to send the error messages to.

Actually, the error messages don’t disappear, they end up in the *error_log* file. It is located in the directory specified by the `ErrorLog` directive in *httpd.conf*. The default setting is generally:

```
ErrorLog /usr/local/apache/logs/error_log
```

So whenever you see *"Internal Server Error"* it’s time to look at this file.

First problem solved!

There are cases when errors don’t go to the error_log file. For example, some errors go to the httpd process’ STDERR. If you haven’t redirected httpd’s STDERR then the messages are printed to the console (tty, terminal) from which you executed the httpd. This happens when the server didn’t get as far as opening the error_log file for writing before it needed to write an error message.

For example, if you have entered a non-existent directory path in your `ErrorLog` directive, the error message will be printed to STDERR. If the error happens when the server executes a `PerlRequire` or `PerlModule` directive you might also see output sent to STDERR.

You are probably wondering where all the errors go when you are running the server in single process mode (`httpd -X`). They go to STDERR. This is because the error logging for all the httpd children is normally done by the parent httpd. When httpd runs in single process mode, it has no parent httpd process

to perform all the logging. The output to the terminal includes all the status messages that normally go to the error_log file.

Finally with a `PerlLogHandler` you can take away from Apache its control of the error logging process for all HTTP transactions. If you do this, then you are responsible for generating and storing the error messages. You can do whatever you like with the information, (including throwing it away -- don't do it!) and, depending on how you implement you `LogHandler`, the `ErrorLog` directive may have no effect. But you can also do something at this handler and then return `DECLINED` status, so the default Apache LogHandler will do the work as usual.

Second problem:

The usefulness of the error message depends to some extent on the programmer's coding style. An uninformative message might not help you to spot and fix the error.

For example, let's take a function which opens a file passed to it as a parameter. It does nothing else with the file. Here's our first version of the code:

```
my $r = shift;
$r->send_http_header('text/plain');

sub open_file{
  my $filename = shift || '';
  die "No filename passed!" unless $filename;

  open FILE, $filename or die;
}

open_file("/tmp/test.txt");
```

Let's assume that `/tmp/test.txt` doesn't exist so the open() will fail to open the file. When we call this script from our browser, the browser returns an *"internal error"* message and we see the following error appended to *error_log*:

```
Died at /home/httpd/perl/test.pl line 9.
```

We can use the hint Perl kindly gave to us to find where in the code the die() was called. However, we still don't know what filename was passed to this subroutine to cause the program termination.

If we have only one function call as in the example above, the task of finding the problematic filename will be trivial. Now let's add two more open_file() function calls and assume that among the three files only */tmp/test2.txt* exists:

```
open_file("/tmp/test.txt");
open_file("/tmp/test2.txt");
open_file("/tmp/test3.txt");
```

When you execute the above call, you will see the same error message twice:

```
Died at /home/httpd/perl/test.pl line 9.
Died at /home/httpd/perl/test.pl line 9.
```

Based on this error message, can you tell what files your program failed to open? Probably not. Let's fix it by passing the name of the file to die():

```
sub open_file{
  my $filename = shift || '';
  die "No filename passed!" unless $filename;
  open FILE, $filename or die "failed to open $filename";
}

open_file("/tmp/test.txt");
```

When we execute the above code, we see:

```
failed to open /tmp/test.txt at /home/httpd/perl/test.pl line 9.
```

which makes a big difference.

By the way, if you append a newline to the end of the message you pass to die(), Perl won't report the line number the error has happened at, so if you code:

```
open FILE, $filename or die "failed to open a file\n";
```

The error message will be:

```
failed to open a file
```

Which gives you very little to go on. It's very hard to debug with such uninformative error messages.

The warn() function, a kinder sister of die(), which logs the message but doesn't cause program termination, behaves in the same way. If you add a newline to the end of the message, the line number warn() was called at won't be logged, otherwise it will.

You might want to use warn() instead of die() if the failure isn't critical. Consider the following code:

```
if(open FILE, $filename){
  # do something with file
} else {
  warn "failed to open $filename";
}
# more code here...
```

Now we've improved our code, by reporting the names of the problematic files, but we still don't know the reason for the failure. Let's try to improve the warn() example. The `-r` operator tests whether the file is readable:

```
if(-r $filename){
  open FILE, $filename;
  # do something with file
} else {
  warn "Couldn't open $filename - doesn't exist or is not readable";
}
```

Now if we cannot read the file we do not even try to open it. But we still see a warning in error_log:

```
Couldn't open /tmp/test.txt - doesn't exist or is not readable
at /home/httpd/perl/test.pl line 9.
```

The warning tells us the reason for the failure, so we don't have to go to the code and check what it was trying to do with the file.

It could be quite a coding overhead to explain all the possible failure reasons that way, but why reinvent the wheel? We already have the reason for the failure stored in the `$!` variable. Let's go back to the open_file() function:

```
sub open_file{
  my $filename = shift || '';
  die "No filename passed!" unless $filename;
  open FILE, $filename or die "failed to open $filename: $!";
}

open_file("/tmp/test.txt");
```

This time, if open() fails we see:

```
failed to open /tmp/test.txt: No such file or directory
at /home/httpd/perl/test.pl line 9.
```

Now we have all the information we need to debug these problems: we know what line of code triggered die(), we know what file we were trying to open, and last but not least we know the reason, given to us through Perl's `$!` variable.

Now let's create the file */tmp/test.txt*.

```
% touch /tmp/test.txt
```

When we execute the latest version of the code, we see:

```
failed to open /tmp/test.txt: Permission denied
at /home/httpd/perl/test.pl line 9.
```

Here we see a different reason: we created a file that doesn't belong to the user which the server runs as (usually *nobody*). It does not have permission to read the file.

Now you can see that it's much easier to debug your code if you validate the return values of the system calls, and properly code arguments to die() and warn() calls. The open() function is just one of the many system calls perl provides to your convenience.

So now you can code and debug CGI scripts and modules as easily as if they were plain Perl scripts that you execute from a shell.

Second problem solved!

17.2.2 Helping error_log to Help Us

It's a good idea to keep it open all the time in a dedicated terminal with the help of *tail -f* or *less -S*, whichever you prefer (the latter allows you to page around the file, search etc.)

```
% tail -f /usr/local/apache/logs/error_log
```

or

```
% less -S /usr/local/apache/logs/error_log
```

So you will see all the errors and warning as they happen.

Another tip is to create a shell *alias*, to make it easier to execute the above command. In tcsh you would do something like this:

```
% alias err "tail -f /usr/local/apache/logs/error_log"
```

For bash users the command is:

```
% alias err='tail -f /var/log/apache/error_log'
```

and from now on in the shell you set the alias in, executing

```
% err
```

will call *tail -f /usr/local/apache/logs/error_log*. Since you want this alias to be available to you all the time, you should put it into your *.tcshrc* file or its equivalent. For *bash* users this is *.bashrc*, or you can put it in */etc/profile* for use by all users.

If you cannot access your *error_log* file because you are unable to telnet to your machine (generally the case when an ISP provides user CGI support but no telnet access), you might want to use a CGI script I wrote to fetch the latest lines from the file (with a bonus of colored output for easier reading). You might need to ask your ISP to install this script for general use. See Watching the error_log file without telneting to the server .

17.2.3 The Importance of Warnings

Just like errors, Perl's mandatory warnings go to the *error_log* file, if the they are enabled. Of course you have enabled them in your development server, haven't you?

The code you write lives a dual life. In the first life it's being written, tested, debugged, improved, tested, debugged, rewritten, tested, debugged. In the second life it's *just* used.

A significant part of the script's first life is spent on the developer's machine. The other part is spent on the production server where the creature is supposed to be perfect.

So when you develop the code you want all the help in the world to help you spot possible problems, and that's where enabling warnings is a must. Whenever you see an error or warning in the *error_log*, you want to get rid of it. That's very important.

Why?

- If there are warnings, your code is not clean. If they are waved away, expect them to come back on the production server in the form of errors, when it's too late.
- If each invocation of a script generates more than about five lines of warnings, it will be very hard to catch real problems. You just can't see them among all the other warnings which you used to think were unimportant.

On the other hand, on a production server, you really *want* to turn warnings off. And there are good reasons for that:

- There is no added value in having the same warning showing up, again and again, triggered by thousands of script invocations. If your code isn't very clean and generates even a single warning per script invocation, on the heavily loaded server you will end up with a huge *error_log* file in a short time.

  The warning elimination phase is supposed to be a part of the development process, and should be done before the code goes live.

- In any Perl script, not just under mod_perl, enabling runtime warnings has a performance impact.

mod_perl gives you a very simple solution to this warnings saga, don't enable warnings in the scripts unless you really have to. Let mod_perl control this mode globally. All you need to do is put the directive

```
PerlWarn On
```

in *httpd.conf* on your development machine and the directive

```
PerlWarn Off
```

on the live box. Here is a complete description on how to manipulate warning modes under mod_perl.

If there is a piece of code that generates warnings and you want to disable them only in this code, you can do that too. The Perl special variable `$^W` allows you dynamically to turn on and off warnings mode. So just put the code into a block, and disable the warnings in the scope of this block. The original value of `$^W` will be restored upon exit from the block.

```
{
 local $^W=0;
  # some code that generates innocuous warnings
}
```

Unless you have a really good reason, for your own sake the advice is *avoid this technique*.

Don't forget the `local()` operand! If you do, setting `$^W` will affect **all** the requests handled by the Apache child that changed this variable. And for **all** the scripts it executes, not just the one which changed `$^W`!

The `diagnostics` pragma can shed more light on errors and warnings, as you will see in a moment.

17.2.3.1 diagnostics pragma

This module extends the terse diagnostics normally emitted by both the Perl compiler and the Perl interpreter, augmenting them with the more verbose and endearing descriptions found in the `perldiag` manpage. Like the other pragmata, it affects the compilation phase of your scripts as well as the execution phase.

To use in your program as a pragma, merely invoke

```
use diagnostics;
```

at or near the start of your program. This also turns on `-w` mode.

This pragma is especially useful when you are new to perl, and want a better explanation of the errors and warnings. It's also helpful when you encounter some warning you've never seen before, e.g. when a new warning has been introduced in an upgraded version of Perl.

You may not want to leave `diagnostics` mode On for your production server. For each warning, `diagnostics` mode generates ten times more output than warnings mode. If your code generates warnings, with the `diagnostics` pragma you will use disk space much faster.

`diagnostics` mode adds a large performance overhead in comparison with just having warnings mode On. You can see the benchmark results in the section 'Code Profiling Techniques'.

17.3 Handling the 'User pressed Stop button' case

When a user presses a **STOP** or **RELOAD** button, the current socket connection goes broken (aborted). It would be nice if Apache could always immediately detect this event. Unfortunately there is no way to tell whether the connection is still valid unless an attempt to read from or write to connection is made.

Unfortunately the detection technique we are going to present doesn't work if the connection to the back-end mod_perl server is coming from the front-end mod_proxy, as the latter doesn't break the connection to the back-end when user has aborted the connection.

If the reading of the request's data is completed and the code does processing without writing anything back to the client the broken connection won't be noticed. When an attempt is made to send at least one character to the client, the broken connection would be noticed and the `SIGPIPE` signal (Broken pipe) would be sent to the process. The program could then halt its execution and perform all the cleanup stuff it has to do.

Prior to Apache version 1.3.6, `SIGPIPE` was handled by Apache. Currently Apache is not handling SIGPIPE anymore and mod_perl takes care of it.

Under mod_perl, `$r->print` (or just print()) returns a *true* value on success, a *false* value on failure. The latter usually happens when the connection is broken.

If you want a similar to the old `SIGPIPE` behaviour (as it was before Apache version 1.3.6), add the following configuration directive:

```
PerlFixupHandler Apache::SIG
```

When Apache's `SIGPIPE` handler is used, Perl may be left in the middle of it's eval context, causing bizarre errors during subsequent requests are handled by that child. When `Apache::SIG` is used, it installs a different `SIGPIPE` handler which rewinds the context to make sure Perl is back to normal state, preventing these bizarre errors.

But in general case, you don't need to use the above setting.

If you use this setting and you would like to log when a request was canceled by a SIGPIPE in your Apache *access_log*, you must define a custom `LogFormat` in your *httpd.conf*, like so:

```
PerlFixupHandler Apache::SIG
LogFormat "%h %l %u %t \"%r\" %s %b %{SIGPIPE}e"
```

If the server has noticed that the request was canceled via a `SIGPIPE`, then the log line will end with `1`, otherwise it will just be a dash. e.g.:

```
127.0.0.1 - - [09/Jan/2001:10:27:15 +0100]
"GET /perl/stopping_detector.pl HTTP/1.0" 200 16 1
127.0.0.1 - - [09/Jan/2001:10:28:18 +0100]
"GET /perl/test.pl HTTP/1.0"              200 10 -
```

17.3.1 Detecting Aborted Connections

Let's use the knowledge we have acquired to trace the execution of the code and watch all the events as they happen.

Let's take a little script that obviously *"hangs"* the server process:

```
stopping_detector.pl
--------------------
my $r = shift;
$r->send_http_header('text/plain');

print "PID = $$\n";
$r->rflush;

while(1){
  $i++;
  sleep 1;
}
```

The script gets a request object `$r` by shift()ing it from the `@_` argument list passed by the handler() subroutine. (This magic is done by `Apache::Registry`). Then the script sends a `Content-type` header, telling the client that we are going to send a plain text as a response.

Next the script prints out a single line telling us the id of the process that handles this request, which we need to know in order to run the tracing utility. Then we flush Apache's buffer. (If we don't flush the buffer we will never see this short information printed. That's because our output is shorter than the buffer size and the script intentionally hangs, so the buffer won't be auto-flushed as the script hangs at the end.)

Then we enter an infinite `while(1)` loop, which just increments a dummy variable and sleeps for a second.

Running `strace -p PID`, where *PID* is the process ID as printed to the browser, we see the following output printed every second:

```
SYS_175(0, 0xbffff41c, 0xbffff39c, 0x8, 0) = 0
SYS_174(0x11, 0, 0xbffff1a0, 0x8, 0x11)    = 0
SYS_175(0x2, 0xbffff39c, 0, 0x8, 0x2)      = 0
nanosleep(0xbffff308, 0xbffff308,
          0x401a61b4, 0xbffff308, 0xbffff41c) = 0
time([941281947])                       = 941281947
time([941281947])                       = 941281947
```

Let's leave `strace` running and press the **STOP** button. Did anything change? No, the same system calls trace is printed every second. Which means that Apache didn't detect the broken connection.

Now we are going to write the `\0` (NULL) character to the client in attempt to detect the broken connection as close as possible to the time the **Stop** button is pressed at. Therefore we modify the loop code in the following way:

```
while(1){
  $r->print("\0");
  last if $r->connection->aborted;
  $i++;
  sleep 1;
}
```

We add a print() statement to print a NULL character and then we check whether the connection was aborted with help of the `$r->connection->aborted` method. If the connection is broken, we break out of the loop.

We run this script and strace on it as before, but we see that it still doesn't work. The trouble is we aren't flushing the buffer, which leaves the characters in the buffer and they won't be printed before the buffer will get full and will be autoflushed. Since we want to attempt to write to the connection pipe all the time, after printing the NULL, we add $r->rflush(). Here is a new version of the code:

```
stopping_detector2.pl
---------------------
my $r = shift;
$r->send_http_header('text/plain');

print "PID = $$\n";
```

```
$r->rflush;

while(1){
  $r->print("\0");
  $r->rflush;

  last if $r->connection->aborted;

  $i++;
  sleep 1;
}
```

After starting the `strace` utility on the running process as we did before and pressing the **Stop** button, we have seen the following output.

```
SYS_175(0, 0xbffff41c, 0xbffff39c, 0x8, 0) = 0
SYS_174(0x11, 0, 0xbffff1a0, 0x8, 0x11) = 0
SYS_175(0x2, 0xbffff39c, 0, 0x8, 0x2)   = 0
nanosleep(0xbffff308, 0xbffff308, 0x401a61b4, 0xbffff308, 0xbffff41c) = 0
time([941284358])                       = 941284358
write(4, "\0", 1)                       = -1 EPIPE (Broken pipe)
--- SIGPIPE (Broken pipe) ---
select(5, [4], NULL, NULL, {0, 0})      = 1 (in [4], left {0, 0})
time(NULL)                              = 941284358
write(17, "127.0.0.1 - - [30/Oct/1999:13:52"..., 81) = 81
gettimeofday({941284359, 39113}, NULL)  = 0
times({tms_utime=9, tms_stime=8, tms_cutime=0, tms_cstime=0}) = 41551400
close(4)                                = 0
SYS_174(0xa, 0xbffff4e0, 0xbffff454, 0x8, 0xa) = 0
SYS_174(0xe, 0xbffff46c, 0xbffff3e0, 0x8, 0xe) = 0
fcntl(18, F_SETLKW, {type=F_WRLCK, whence=SEEK_SET, start=0, len=0}
```

Apache detects the broken pipe as you see from this snippet:

```
write(4, "\0", 1)                       = -1 EPIPE (Broken pipe)
--- SIGPIPE (Broken pipe) ---
```

Then it stops the script and does all the cleanup work, like access logging:

```
write(17, "127.0.0.1 - - [30/Oct/1999:13:52"..., 81) = 81
```

where 17 is a file descriptor of the opened *access_log* file

17.3.2 The Importance of Cleanup Code

Cleanup code is a critical issue with aborted scripts.

What happens to locked resources if there are any? Will they be freed or not? If not, scripts using these resources and the same locking scheme will hang, waiting for them to be freed.

First let’s go one step back and recall what are the problems and solutions for this issue under mod_cgi.

Under mod_cgi the resource locking issue is a problem only if you happened to create external lock files and use them for lock indication, instead of using flock(). If the script running under mod_cgi is aborted between the lock and the unlock code, and you didn't bother to write cleanup code to remove old dead locks then you are in big trouble.

The solution is to use an `END` block to place the cleanup code in:

```
END {
  # some code that ensures that locks are removed
}
```

When the script is aborted, Apache will run the `END` blocks.

If you use `flock()` things are much simpler, since all opened files will be closed when the script exits. When the file is closed, the lock is removed as well--all the locked resources get freed. There are systems where flock(2) is unavailable, and for those you can use Perl's emulation of this function.

With mod_perl things can be more complex when you use global variables as a filehandlers. Because the processes don't exit after processing a request, files won't be closed unless you explicitly close() them or reopen with the open() call, which first closes a file. Let's see what problems we might encounter, and possible solutions for them.

17.3.2.1 Critical Section

First we want to make a little detour to discuss the *"critical section"* issue.

Let's start with a resource locking scheme. A schematic representation of a proper locking technique is as follows:

```
1. lock a resource
   <critical section starts>
2. do something with the resource
   <critical section ends>
3. unlock the resource
```

If the locking is exclusive, only one process can hold the resource at any given time, which means that all the other processes will have to wait, therefore the code between the locking and unlocking functions can become a service bottleneck. That's why this code section is called critical and once started it should be finished as soon as possible.

Even if you use a shared locking scheme, where many processes are allowed to concurrently access the resource, if there are processes that sometimes want to get an exclusive lock it's also important to keep the critical section as short as possible.

The next example uses a shared lock, but has a poorly-designed critical section:

```
critical_section_sh.pl
------------------
use Fcntl qw(:flock);
use Symbol;
my $fh = gensym;
```

```
open $fh, "/tmp/foo" or die $!;
flock $fh, LOCK_SH;
  # start critical section

seek $fh, 0, 0;
my @lines = <$fh>;
for(@lines){
  print if /foo/;
}

  # end critical section
close $fh; # close unlocks the file
```

The code opens the file for reading, locks and rewinds it to the beginning, reads all the lines from the file and prints out the lines that contain the string *'foo'*.

The `gensym()` function imported by the `Symbol` module creates an anonymous glob and returns a reference to it. Such a glob reference can be used as a file or directory handle. and therefore allows using lexically scoped variables as filehandlers. `Fcntl` imports into the script's namespace file locking symbols like: `LOCK_SH`, `LOCK_EX` and more. Refer to the `Fcntl` manpage for more information.

If the file the script reads is big, it'd take a relatively long time for this code to complete. All this time the file remains open and locked. While it's other processes may access this file for reading (shared lock), the process that wants to modify the file (which requires an acquisition of the exclusive lock), will be blocked waiting for this section to complete.

We can optimize the critical section this way:

Once the file has been read, we have all the information we need from it. In order to make the example simpler we've chosen to just print out the matching lines. In reality the code might be much longer.

We don't need the file to be open while the loop executes, because we don't access it inside the loop. If we close the file before we start the loop, we will allow other processes to have an exclusive access to the file if they need it, instead of blocking them for no reason.

In the following corrected version of the previous example, we only read the content of the file during the critical section and process it afterwards, without creating a possible bottleneck.

```
critical_section_sh2.pl
-----------------------
use Fcntl qw(:flock);
use Symbol;
my $fh = gensym;

open $fh, "/tmp/foo" or die $!;
flock $fh, LOCK_SH;
  # start critical section

seek $fh, 0, 0;
my @lines = <$fh>;

  # end critical section
```

```
close $fh; # close unlocks the file

for(@lines){
  print if /foo/;
}
```

Here is another similar example, but now it uses an exclusive lock. The script reads in a file and writes it back, adding a number of new text lines to the head of the file.

```
critical_section_ex.pl
----------------------
use Fcntl qw(:flock);
use Symbol;
my $fh = gensym;

open $fh, "+>>/tmp/foo" or die $!;
flock $fh, LOCK_EX;

  # start critical section
seek $fh, 0, 0;
my @add_lines =
  (
   qq{Complete documentation for Perl, including FAQ lists,\n},
   qq{should be found on this system using 'man perl' or\n},
   qq{'perldoc perl'. If you have access to the Internet, point\n},
   qq{your browser at http://www.perl.com/, the Perl Home Page.\n},
  );

my @lines = (@add_lines, <$fh>);
seek $fh, 0, 0;
truncate $fh, 0;
print $fh @lines;
  # end critical section

close $fh; # close unlocks the file
```

Since we want to read the file, modify and write it back, without anyone else changing it on the way, we open it for read and write with the help of +>> and lock it with an exclusive lock. You cannot safely accomplish this task by opening the file first for read and then reopening for write, since another process might change the file between the two events. (You could get away with +< as well, please refer to the *perlfunc* manpage for more information about the open() function.)

Next, the code prepares the lines of text it wants to add to the head of the file, and assigns them and the content of the file to the `@lines` array. Now we have our data ready to be written back to the file, so we seek() to the start of the file and truncate() it to zero size. In our example the file always grows, so in this case there is no need to truncate it, but if there was a chance that the file might shrink then truncating would be necessary. However it's good practice to always use truncate(), as you never know what changes your code might undergo in the future. The truncate() operation does not carry any significant performance penalty. Finally we write the data back to the file and close it, which unlocks it as well.

Did you notice that we created the text lines to be added as close to the place of usage as possible? This complies with good *"locality of code"* style, but it makes the critical section longer. In such cases you should sacrifice style, in order to make the critical section as short as possible. An improved version of this

script with a shorter critical section looks like this:

```
critical_section_ex2.pl
-----------------------
use Fcntl qw(:flock);
use Symbol;

my @lines =
  (
   qq{Complete documentation for Perl, including FAQ lists,\n},
   qq{should be found on this system using ‘man perl’ or\n},
   qq{‘perldoc perl’. If you have access to the Internet, point\n},
   qq{your browser at http://www.perl.com/, the Perl Home Page.\n},
  );

my $fh = gensym;
open $fh, "+>>/tmp/foo" or die $!;
flock $fh, LOCK_EX;
  # start critical section

seek $fh, 0, 0;
push @lines, <$fh>;

seek $fh, 0, 0;
truncate $fh, 0;
print $fh @lines;

  # end critical section
close $fh; # close unlocks the file
```

There are two important differences. First, we prepare the text lines to be added *before* the file is locked. Second, instead of creating a new array and copying lines from one array to another, we append the file directly to the `@lines` array.

17.3.2.2 Safe Resource Locking and Cleanup Code

Let’s get back to the main issue of this section, which is safe resource locking.

Unless you use the `Apache::PerlRun` handler that does the cleanup for you, if you don’t make a habit of closing all the files that you open--in some cases you will encounter lots of problems. If you open a file but don’t close it, you may have file descriptor leakage. Since the number of file descriptors available to you is finite, at some point you may run out of them and your service will fail. This is bad, but you can live with it until you run out of file descriptors (which will happen much faster on a heavily used server).

You can use system utilities to observe the opened and locked files, as well as the processes that has opened (and locked) the files. On FreeBSD you would use the fstat(1) utility. On many other UN*X flavors the lsof(1) utility is available.

But this is nothing compared to the trouble you will give yourself if the code terminates and the file stays locked. Any other process requesting a lock on the same file (or resource) will wait indefinitely for it to become unlocked. Since this will not happen until the server reboots, all these processes trying to use this resource will hang.

Here is an example of such a terrible mistake:

```
flock.pl
--------
use Fcntl qw(:flock);
open IN, "+>>filename" or die "$!";
flock IN, LOCK_EX;
  # do something
  # quit without closing and unlocking the file
```

Is this safe code? No - we forgot to close the file. So let's add the close():

```
flock2.pl
---------
use Fcntl qw(:flock);
open IN, "+>>filename" or die "$!";
flock IN, LOCK_EX;
  # do something
close IN;
```

Is it safe code now? Unfortunately it is not. There is a chance that the user may abort the request (for example by pressing his browser's `Stop` or `Reload` buttons) during the critical section. The script will be aborted before it has had a chance to close() the file, which is just as bad as if we forgot to close it.

In fact if the same process will run the same code again, an open() call will close the file first, which will unlock the resource. This is because `IN` is a global variable. But it's quite possible that the process that created the lock, will not serve the same request for a while, since it would be busy serving other requests. So relying on it to reopen the file is a bad idea.

This problem happens **only** if you use global variables as file handles. The following example has the same problem.

```
flock3.pl
---------
use Fcntl qw(:flock);
use Symbol ();
use vars qw($fh);
$fh = Symbol::gensym();
open $fh, "+>>filename" or die "$!";
flock $fh, LOCK_EX;
  # do something
close $fh;
```

`$fh` is still a global variable and therefore the code using it suffers from the same problem.

The simplest solution to this problem is to always use lexically scoped variables (created with my ()). Whether script gets aborted before close() is called or you forgot the use close() the lexically scoped variable will always go out of scope and therefore if the file was locked it will be unlocked. Here is a good version of the code:

```
flock4.pl
---------
use Fcntl qw(:flock);
use Symbol ();
my $fh = Symbol::gensym();
open $fh, "+>>filename" or die "$!";
flock $fh, LOCK_EX;
  # do something
close $fh;
```

Please don't conclude from this example that you don't have to close files anymore, since they will be automatically closed for you. It's a bad style and should be avoided.

mod_perl comes with its own implementation of gensym(), so you don't even need to load the Symbol module in order to use this function. In mod_perl this function resides in the `Apache` package. For example:

```
use Apache;
my $fh = Apache::gensym();
open $fh, "+>>filename" or die "$!";
...
```

If you insist on using the file globs, at least make sure that you local()'ize these, and then if the flow of the code is interrupted before close() was called the filehandle will be automatically closed, since the local()'ized variable will go out of the scope. The following example shows that the file is indeed closed even when there is no close():

```
/tmp/io.pl
--------------
#!/usr/bin/perl
# /dev/null so strace output is more readable
open my $fh, ">/dev/null";
select $fh;
$| = 1;
{
  print "enter";
  local *FH;
  open FH, $0;
  print "leave"
}
print "done";
```

This simple script opens the */dev/null* and tells Perl to send all the STDOUT there, which is also made unbuffered. Then the block is created in which the `FH` file glob is localized. Then it's used to open the source code of the script (which resides in `$0`). In order to separate event of entering the block scope and leaving it, the debug print statements are used. Now let's run the script under strace(1), which proves once again to be very useful in the tool bag of the mod_perl programmer:

```
% strace /tmp/io.pl
write(3, "enter", 5)                     = 5
-> open("/tmp/io.pl", O_RDONLY) = 4
fstat(4, {st_mode=S_ISGID|S_ISVTX|0401, st_size=0, ...}) = 0
fcntl(4, F_SETFD, FD_CLOEXEC)           = 0
write(3, "leave", 5)                    = 5
-> close(4)                               = 0
write(3, "done", 4)                     = 4
```

So you can see that */tmp/io.pl* is actually close()'d.

Under Perl version 5.6 `Symbol.pm`-like functionality is a built-in feature, so you can do:

```
open my $fh, ">/tmp/foo" or die $!;
```

and `$fh` will be automatically vivified as a valid filehandle, so you don't need to use the `Symbol` module anymore, if backward compatibility is not a requirement.

You can also use the `IO::*` modules, such as `IO::File` or `IO::Dir`. These are much bigger than the `Symbol` module, and worth using for files or directories only if you are already using them for the other features which they provide. As a matter of fact, these modules use the `Symbol` module themselves. Here is an example of their usage:

```
use IO::File;
use IO::Dir;
my $fh = IO::File->new(">filename");
my $dh = IO::Dir->new("dirname");
```

If you still have to use global filehandles, there are a few approaches we can take to solving the locking problem.

If you are running under `Apache::Registry` and friends, the `END` block will perform the cleanup work for you. You might use `END` in the same way for scripts running under mod_cgi, or in plain Perl scripts. Just add the cleanup code to this block and you are safe.

For example if you work with dbm files just like with locking it's important to flush the dbm buffers, by calling a sync() method:

```
END{
  # make sure that the DB is flushed
   $dbh->sync();
}
```

Normally the `END` blocks will not be executed after the completion of a request, but only when an Apache child process exits, then if you are writing your own handlers you will need to use the register_cleanup() function to supply cleanup code similar to that used in `END` blocks instead of using `END` blocks.

Under mod_perl, the above will work only for `Apache::Registry` scripts. Otherwise execution of the `END` block will be postponed until the process terminates. If you write a handler in the Perl API use the `register_cleanup()` method instead. It accepts a reference to a subroutine as an argument:

```
$r->register_cleanup(sub { $dbh->sync() });
```

Even better would be to check whether the client connection has been aborted. If you don't check, the cleanup code will always be executed and for normally terminated scripts this may not be what you want:

```
$r->register_cleanup(
  # make sure that the DB is flushed
  sub{
    $dbh->sync() if Apache->request->connection->aborted();
  }
);
```

So in the case of `END` block usage you would use:

```
END{
  # make sure that the DB is flushed
  $dbh->sync() if Apache->request->connection->aborted();
}
```

Note that if you use `register_cleanup()` it should be called at the beginning of the script, or as soon as the variables you want to use in this code become available. If you use it at the end of the script, and the script happens to be aborted before this code is reached, there will be no cleanup performed.

For example `CGI.pm` registers the cleanup subroutine in its new() method:

```
sub new {
  # code snipped
  if ($MOD_PERL) {
      Apache->request->register_cleanup(\&CGI::_reset_globals);
      undef $NPH;
  }
  # more code snipped
}
```

There is another way to register a section of cleanup code for Perl API handlers. You may use `PerlCleanupHandler` in the configuration file, like this:

```
<Location /foo>
  SetHandler perl-script
  PerlHandler        Apache::MyModule
  PerlCleanupHandler Apache::MyModule::cleanup()
  Options ExecCGI
</Location>
```

`Apache::MyModule::cleanup()` performs the cleanup, obviously.

17.4 Handling Server Timeout Cases and Working with $SIG{ALRM}

A similar situation to Pressed Stop button disease happens when the browser times out the connection (is it about 2 minutes?). There are cases when your script is about to perform a very long operation and there is a chance that its duration will be longer than the client's timeout. One example is database interaction, where the DB engine hangs or needs a long time to return the results. If this is the case, use `$SIG{ALRM}` to prevent the timeouts:

```
  $timeout = 10; # seconds
eval {
  local $SIG{ALRM} =
      sub { die "Sorry timed out. Please try again\n" };
  alarm $timeout;
  ... db stuff ...
  alarm 0;
};

die $@ if $@;
```

It was recently discovered that `local $SIG{'ALRM'}` does not restore the original underlying C handler. This was fixed in mod_perl 1.19_01. As a matter of fact none of the `local $SIG{FOO}` signals restores the original C handler - read Debugging Signal Handlers ($SIG{FOO}) for a debug technique and a possible workaround.

17.5 Looking inside the server

Your server is up and running, but something appears to be wrong. You want to see the numbers to tune your code or server configuration. You just want to know what's really going on inside the server.

How do you do it?

There are a few tools that allow you to look inside the server.

17.5.1 Apache::Status -- Embedded Interpreter Status Information

This is a very useful module. It lets you watch what happens to the Perl parts of the server. You can see the size of all subroutines and variables, variable dumps, lexical information, OPcode trees, and more.

You shouldn't use it on production server as it adds quite a bit of overhead for each request.

17.5.1.1 Minimal Configuration

This configuration enables the `Apache::Status` module with its minimum feature set. Add this to *httpd.conf*:

```
<Location /perl-status>
  SetHandler perl-script
  PerlHandler Apache::Status
  order deny,allow
  #deny from all
  #allow from
</Location>
```

If you are going to use `Apache::Status` it's important to put it as the first module in the start-up file, or in *httpd.conf*:

```
# startup.pl
use Apache::Status ();
use Apache::Registry ();
use Apache::DBI ();
```

If you don't put `Apache::Status` before `Apache::DBI`, you won't get the `Apache::DBI` menu entry in the status. For more about `Apache::DBI` see Persistent DB Connections.

17.5.1.2 Extended Configuration

There are several variables which you can use to modify the behaviour of `Apache::Status`.

- **PerlSetVar StatusOptionsAll On**

  This single directive will enable all of the options described below.

- **PerlSetVar StatusDumper On**

  When you are browsing symbol tables, you can view the values of your arrays, hashes and scalars with `Data::Dumper`.

- **PerlSetVar StatusPeek On**

  With this option On and the `Apache::Peek` module installed, functions and variables can be viewed in `Devel::Peek` style.

- **PerlSetVar StatusLexInfo On**

  With this option On and the `B::LexInfo` module installed, subroutine lexical variable information can be viewed.

- **PerlSetVar StatusDeparse On**

  With this option On and `B::Deparse` version 0.59 or higher (included in Perl 5.005_59+), subroutines can be "deparsed".

  Options can be passed to `B::Deparse::new` like so:

  ```
  PerlSetVar StatusDeparseOptions "-p -sC"
  ```

  See the `B::Deparse` manpage for details.

- **PerlSetVar StatusTerse On**

  With this option On, text-based op tree graphs of subroutines can be displayed, thanks to `B::Terse`.

- **PerlSetVar StatusTerseSize On**

  With this option On and the `B::TerseSize` module installed, text-based op tree graphs of subroutines and their size can be displayed. See the `B::TerseSize` docs for more info.

- **PerlSetVar StatusTerseSizeMainSummary On**

  With this option On and the `B::TerseSize` module installed, "Memory Usage" will be added to the `Apache::Status` main menu. This option is disabled by default, as it can be rather cpu intensive to summarize memory usage for the entire server. It is strongly suggested that this option only be used with a development server running in -X mode, as the results will be cached.

  Remember to preload `B::TerseSize` with:

  ```
  PerlModule B::Terse
  ```

- **PerlSetVar StatusGraph**

  When `StatusDumper` (see above) is enabled, another link *"OP Tree Graph"* will be present with the dump if this configuration variable is set to On.

  This requires the B module (part of the Perl compiler kit) and the `B::Graph` module version 0.03 or higher to be installed along with the ‘dot’ program. Dot is part of the graph visualization toolkit from AT&T: http://www.research.att.com/sw/tools/graphviz/.

  WARNING: Some graphs may produce very large images, and some graphs may produce no image if `B::Graph`’s output is incorrect.

There is more information about `Apache::Status` in its manpage.

17.5.1.3 Usage

Assuming that your mod_perl server listens on port 81, fetch http://www.myserver.com:81/perl-status

```
Embedded Perl version 5.00502 for Apache/1.3.2 (Unix) mod_perl/1.16
process 187138, running since Thu Nov 19 09:50:33 1998
```

Below all the sections are links when you view them through */perl-status*

```
Signal Handlers
Enabled mod_perl Hooks
PerlRequire’d Files
Environment
Perl Section Configuration
Loaded Modules
Perl Configuration
ISA Tree
Inheritance Tree
Compiled Registry Scripts
Symbol Table Dump
```

Let's follow, for example, `PerlRequire`'d Files. We see:

```
PerlRequire                   Location
/home/perl/apache-startup.pl  /home/perl/apache-startup.pl
```

From some menus you can move deeper to peek into the internals of the server, to see the values of the global variables in the packages, to see the cached scripts and modules, and much more. Just click around...

17.5.1.4 Compiled Registry Scripts section seems to be empty.

Sometimes when you fetch */perl-status* and look at the **Compiled Registry Scripts** you see no listing of scripts at all. This is correct: `Apache::Status` shows the registry scripts compiled in the httpd child which is serving your request for */perl-status*. If the child has not yet compiled the script you are asking for, */perl-status* will just show you the main menu.

17.5.2 mod_status

The Status module allows a server administrator to find out how well the server is performing. An HTML page is presented that gives the current server statistics in an easily readable form. If required, given a compatible browser this page can be automatically refreshed. Another page gives a simple machine-readable list of the current server state.

This Apache module is written in C. It is compiled by default, so all you have to do to use it is enable it in your configuration file:

```
<Location /status>
  SetHandler server-status
</Location>
```

For security reasons you will probably want to limit access to it. If you have installed Apache according to the instructions you will find a prepared configuration section in *httpd.conf*: to enable use of the mod_status module, just uncomment it.

```
ExtendedStatus On
<Location /status>
  SetHandler server-status
  order deny,allow
  deny from all
  allow from localhost
</Location>
```

You can now access server statistics by using a Web browser to access the page http://localhost/status (as long as your server recognizes localhost:).

The details given by mod_status are:

- **The number of children serving requests**
- **The number of idle children**
- **The status of each child, the number of requests that child has performed and the total number**

of bytes served by the child

- **A total number of accesses and the total bytes served**
- **The time the server was last started/restarted and how long it has been running for**
- **Averages giving the number of requests per second, the number of bytes served per second and the average number of bytes per request**
- **The current percentage CPU used by each child and in total by Apache**
- **The current hosts and requests being processed**

17.5.3 Apache::VMonitor -- Visual System and Apache Server Monitor

This module is covered in the section "Apache::* Modules"

17.6 Sometimes My Script Works, Sometimes It Does Not

See Sometimes it Works Sometimes it does Not

17.7 Code Debug

When the code doesn't perform as expected, either never or just sometimes, we say that the code needs debugging. There are several levels of debugging complexity.

The basic level is when Perl terminates the program during the compilation phase, before it tries to run the resulting byte-code. This usually happens because there are syntax errors in the code, or perhaps a module is missing. Sometimes it takes quite an effort to solve these problems, since code that uses Apache CORE modules generally won't compile when executed from the shell. We will learn how to solve syntax problems in mod_perl code quite easily.

Once the program compiles and begins to run, there might be logical problems, when the program doesn't do what you thought you had programmed it to do. These are somewhat harder to solve, especially when there is a lot of code to be inspected and reviewed, but it's just a matter of time. Perl can help a lot, for example to locate typos, when we enable warnings. For example, if you wanted to compare two numbers, but you omitted the second '=' character so that you had something like `if $yes = 1` instead of `if $yes == 1`, it warns us about the missing '='.

The next level is when the program does what it's expected to do most of the time, but occasionally misbehaves. Often you find that print() statements or the Perl debugger can help, but inspection of the code generally doesn't. Often it's quite easy to debug with print(), but sometimes typing the debug messages can become very tedious. That's where the Perl debugger comes into its own.

While print() statements always work, running the perl debugger for CGI scripts might be quite a challenge. But with the right knowledge and tools handy the debug process becomes much easier. Unfortunately there is no one easy way to debug your programs, as the debugging depends entirely on your code. It can be a nightmare to debug really complex code, but as your style matures you can learn ways to write simpler code that is easier to debug. You will probably find that when you write simpler clearer code it

does not need so much debugging in the first place.

One of the most difficult cases to debug, is when the process just terminates in the middle of processing a request and dumps core. Often when there is a bug the program tries to access a memory area that doesn't belong to it. The operating system halts the process, tidies up and dumps core (it creates a file called *core* in the current directory of the process that was running). This is something that you rarely see with plain perl scripts, but it can easily happen if you use modules written in *C* or *C++* and something goes wrong with them. Occasionally you will come across a bug in mod_perl itself (mod_perl is written in C), that was in a deep slumber before your code awakened it.

In the following sections we will go through in detail each of the problems presented, thoroughly discuss them and present a few techniques to solve them.

17.7.1 Locating and correcting Syntax Errors

While developing code we often make syntax mistakes, like forgetting to put a comma in a list, or a semicolon at the end of a statement.

Even at the end of a {} block, where a semicolon is not required at the end of the last statement, it may be better to put one in: there is a chance that you will add more code later, and when you do you might forget to add the now required semicolon. Similarly, more items might be added later to a list; unlike many other languages, Perl has no problem when you end a list with a redundant comma.

One approach to locating syntactically incorrect code is to execute the script from the shell with the *-c* flag. This tells Perl to check the syntax but not to run the code (actually, it will execute `BEGIN`, `END` blocks, and *use()* calls, because these are considered as occurring outside the execution of your program). (Note also that Perl 5.6.0 has introduced a new special variable, `$^C`, which is set to true when perl is run with the *-c* flag; this provides an opportunity to have some further control over `BEGIN` and `END` blocks during syntax checking.) Also it's a good idea to add the `-w` switch to enable warnings:

```
perl -cw test.pl
```

If there are errors in the code, Perl will report the errors, and tell you at which line numbers in your script the errors were found.

The next step is to execute the script, since in addition to syntax errors there may be run time errors. These are the errors that cause the *"Internal Server Error"* page when executed from a browser. With plain CGI scripts it's the same as running plain Perl scripts -- just execute them and see that they work.

The whole thing is quite different with scripts that use `Apache::*` modules which can be used only from within the mod_perl server environment. These scripts rely on other code, and an environment which isn't available when you attempt to execute the script from the shell. There is no Apache request object available to the code when it is executed from the shell.

If you have a problem when using `Apache::*` modules, you can make a request to the script from a browser and watch the errors and warnings as they are logged to the *error_log* file. Alternatively you can use the `Apache::FakeRequest` module.

17.7.2 Using Apache::FakeRequest to Debug Apache Perl Modules

`Apache::FakeRequest` is used to set up an empty Apache request object that can be used for debugging. The `Apache::FakeRequest` methods just set internal variables with the same names as the methods and return the value of the internal variables. Initial values for methods can be specified when the object is created. The print method prints to STDOUT.

Subroutines for Apache constants are also defined so that you can use `Apache::Constants` while debugging, although the values of the constants are hard-coded rather than extracted from the Apache source code.

Let's write a very simple module, which prints *"OK"* to the client's browser:

```
package Apache::Example;
use Apache::Constants;

sub handler{
  my $r = shift;
  $r->send_http_header('text/plain');
  print "You are OK ", $r->get_remote_host, "\n";
  return OK;
}

1;
```

You cannot debug this module unless you configure the server to run it, by calling its handler from somewhere. So for example you could put in *httpd.conf*:

```
<Location /ex>
  SetHandler perl-script
  PerlHandler Apache::Example
</Location>
```

Then after restarting the server you could start a browser, request the location http://localhost/ex and examine the output. Tedious, no?

But with the help of `Apache::FakeRequest` you can write a little script that will emulate a request and return the output.

```
#!/usr/bin/perl

use Apache::FakeRequest ();
use Apache::Example ();

my $r = Apache::FakeRequest->new('get_remote_host'=>'www.foo.com');
Apache::Example::handler($r);
```

when you execute the script from the command line, you will see the following output:

```
You are OK www.foo.com
```

17.7.3 Finding the Line Which Triggered the Error or Warning

Perl has no problem with the line numbers and file names for modules that are read from disk in the normal way, but modules that are compiled via eval() such as `Apache::Registry` and `Apache::PerlRun` sometimes with some versions of Perl get confused.

There is the Perl <<HEREDOC inside eval "" problem that confuses the Perl current linenumber counter, newer Perls fix this. For older Perls compiling with the experimental **PERL_MARK_WHERE=1** should solve this.

Also if you happen to use line continuation \ your script, the line numbering will be off as well, since eval eats those \'s and combines the lines together. The solution: do not use \'s to mark lines as continued in Perl. Unlike shell languages, Perl doesn't require you to do that.

There are compiler directives to reset its counter to some value that you decide. You can always pepper your code with these to help you locate the problem. At the beginning of the line you could write something of the form:

```
#line nnn label
```

For example:

```
#line 298 myscript.pl
or
#line 890 some_label_to_be_used_in_the_error_message
```

The '#' must be in the first column, so if you cut and paste from this text you must remember to remove any leading white space.

The label is optional - the filename of the script will be used by default. This directive sets the line number of the **following** line, not the line the directive is on. You can use a little script to stuff every N lines of your code with these directives, but then you will have to remember to rerun this script every time you add or remove code lines. The script:

```
#!/usr/bin/perl
# Puts Perl line markers in a Perl program for debugging purposes.
# Also takes out old line markers.
die "No filename to process.\n" unless @ARGV;
my $filename = shift;
my $lines = 100;
open IN, $filename or die "Cannot open file: $filename: $!\n";
open OUT, ">$filename.marked"
    or die "Cannot open file: $filename.marked: $!\n";
my $counter = 1;
while (<IN>) {
  print OUT "#line $counter\n" unless $counter++ % $lines;
  next if /^#line /;
  print OUT $_;
}
close OUT;
close IN;
chmod 0755, "$filename.marked";
```

Another way of narrowing down the area to be searched is to move most of the code into a separate modules. This ensures that the line number will be reported correctly.

To have a complete trace of calls add:

```
use Carp ();
local $SIG{__WARN__} = \&Carp::cluck;
```

17.7.4 Using print() for Debugging

The universal debugging tool across nearly all platforms and programming languages is printf() or the equivalent output function. This can send data to the console, a file, an application window and so on. In perl we generally use the print() function. With an idea of where and when the bug is triggered, a developer can insert print() statements in the source code to examine the value of data at certain stages of execution.

However, it is rather difficult to anticipate all possible directions a program might take and what data to suspect of causing trouble. In addition, inline debugging code tends to add bloat and degrade the performance of an application and can also make the code harder to read and maintain. And you have to comment out or remove the debugging print() calls when you think that you have solved the problem. But if later you discover that you need to debug the same code again, you need at best to uncomment the debugging code lines or, at worst, to write them again from scratch.

Let’s see a few examples where we use print() to debug some problem. In one of my applications I wrote a function that returns the date that was one week ago. Here it is:

```
print "Content-type: text/plain\r\n\r\n";

print "A week ago the date was ",date_a_week_ago(),"\n";

# return a date one week ago as a string in format: MM/DD/YYYY
####################
sub date_a_week_ago{

  my @month_len   = (31,28,31,30,31,30,31,31,30,31,30,31);

  my ($day,$month,$year) = (localtime)[3..5];
  for (my $j = 0; $j < 7; $j++) {

    $day--;
    if ($day == 0) {

      $month--;
      if ($month == 0) {
        $year--;
        $month = 12;
      }

        # there are 29 days in February in a leap year
      $month_len[1] =
        (($year % 4 or $year % 100 == 0) and $year % 400 )
  ? 28 : 29;
```

```
        # set $day to be the last day of the previous month
      $day = $month_len[$month - 1];

    }   # end of if ($day == 0)
  }     # end of for ($i = 0;$i < 7;$i++)

  return sprintf "%02d/%02d/%04d",$month,$day,$year+1900;
}
```

This code is pretty straightforward. We get today's date and subtract one from the value of the day we get, updating the month and the year on the way if boundaries are being crossed (end of month, end of year). If we do it seven times in loop then at the end we should get a date that was a week ago.

Note that since locatime() returns the year as a value of `current_four_digits_format_year-1900` (which means that we don't have a century boundary to worry about) then if we are in the middle of the first week of the year 2000, the value of year returned by localtime() will be `100` and not `0` as you might mistakenly assume. So when the code does `$year--` it becomes `99` and not `-1`. At the end we add 1900 to get back the correct four-digit year format. (This is all correct as long as you don't go to the years prior to 1900)

Also note that we have to account for leap years where there are 29 days in February. For the other months we have prepared an array containing the month lengths.

Now when we run this code and check the result, we see that something is wrong. For example, if today is `10/23/1999` we expect the above code to print `10/16/1999`. In fact it prints `09/16/1999`, which means that we have lost a month. The above code is buggy!

Let's put a few debug print() statements in the code, near the `$month` variable:

```
sub date_a_week_ago{

  my @month_len   = (31,28,31,30,31,30,31,31,30,31,30,31);

  my ($day,$month,$year) = (localtime)[3..5];
  print "[set] month : $month\n"; # DEBUG
  for (my $j = 0; $j < 7; $j++) {

    $day--;
    if ($day == 0) {

      $month--;
      if ($month == 0) {
        $year--;
        $month = 12;
      }
      print "[loop $i] month : $month\n"; # DEBUG

        # there are 29 days in February in a leap year
      $month_len[1] =
        (($year % 4 or $year % 100 == 0) and $year % 400 )
  ? 28 : 29;

        # set $day to be the last day of the previous month
```

```
        $day = $month_len[$month - 1];

      }   # end of if ($day == 0)
    }     # end of for ($i = 0;$i < 7;$i++)

    return sprintf "%02d/%02d/%04d",$month,$day,$year+1900;
  }
```

When we run it we see:

```
[set] month : 9
```

It is supposed to be the number of the current month (`10`), but actually it is not. We have spotted a bug, since the only code that sets the `$month` variable consists of a call to localtime(). So did we find a bug in Perl? let's look at the manpage of the localtime() function:

```
% perldoc -f localtime

Converts a time as returned by the time function to a 9-element
array with the time analyzed for the local time zone.  Typically
used as follows:

  #  0    1    2     3     4    5     6     7     8
  ($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) =
                                              localtime(time);

All array elements are numeric, and come straight out of a struct
tm.  In particular this means that C<$mon> has the range C<0..11>
and C<$wday> has the range C<0..6> with Sunday as day C<0>.  Also,
C<$year> is the number of years since 1900, that is, C<$year> is
C<123> in year 2023, and I<not> simply the last two digits of the
year.  If you assume it is, then you create non-Y2K-compliant
programs--and you wouldn't want to do that, would you?
[more info snipped]
```

Which reveals to us that if we want to count months from 1 to 12 and not 0 to 11 we are supposed to increment the value of `$month`. Among other interesting facts about locatime() we also see an explanation of `$year`, which as I've mentioned before is set to the number of years since 1900.

We have found the bug in our code and learned new things about localtime(). To correct the above code we just increment the month after we call localtime():

```
my ($day,$month,$year) = (localtime)[3..5];
$month++;
```

17.7.5 Using print() and Data::Dumper for Debugging

Sometimes you need to peek into complex data structures, and trying to print them out can be tricky. That's where `Data::Dumper` comes to our rescue. For example if we create this complex data structure:

```
$data =
  {
   array => [qw(a b c d)],
   hash  => {
         foo => "oof",
         bar => "rab",
        },
  };
```

How do we print it out? Very easily:

```
use Data::Dumper;
print Dumper \$data;
```

What we get is a pretty-printed `$data`:

```
$VAR1 = \{
          'hash' => {
                      'foo' => 'oof',
                      'bar' => 'rab'
                    },
          'array' => [
                      'a',
                      'b',
                      'c',
                      'd'
                     ]
        };
```

While writing this example I made a mistake and wrote `qw(a b c d)` instead of `[qw(a b c d)]`. When I pretty-printed the contents of `$data` I immediately saw my mistake:

```
$VAR1 = \{
          'b' => 'c',
          'd' => 'hash',
          'HASH(0x80cd79c)' => undef,
          'array' => 'a'
        };
```

That's not what I wanted of course, but I spotted the bug and corrected it, as you saw in the original example from above.

Of course you can use

```
print STDERR $variable;
```

or:

```
warn $variable;
```

instead of print to have all the debug messages in the error_log, which makes it even easier to debug your code.

17.7.6 The Importance of a Good Concise Coding Style

Don't strive for elegant, clever code. Try to develop a good coding style by writing code which is concise yet easy to understand. It's much easier to find bugs in concise, simple code. And such code tends to have less bugs.

The *'one week ago'* example from the previous section is not concise. There is a lot of redundancy in it, and as a result it is harder to debug than it needs to be. Here is a condensed version of the main loop. As you can see, this version won't make it easier to understand the code:

```
for (0..6) {
  next if --$day;
  $year--,$month=12 unless --$month;
  $day = $month != 1
      ? $month_len[$month-1]
      : (($year % 4 or $year % 100 == 0) and $year % 400 )
           ? 28
           : 29;
}
```

Don't do that at home :)

Why did I present this version? Because it is too obscure, which makes it difficult to understand and maintain. On the other hand a part of this code is easier to understand.

Larry Wall, the author of Perl, is a linguist. He tried to define the syntax of Perl in a way that makes working in Perl much like working in English. So it can be a good idea to learn Perl coding idioms, some of which might seem odd at first but once you get used to them, you will find it difficult to understand how you could have lived without them before. I'll show just a few of the most common Perl coding idioms.

It's a good idea to write code which is more readable but which avoids redundancy, so it's better to write:

```
unless ($i) {...}
```

rather than:

```
if ($i == 0) {...}
```

if you want to test for trueness only.

Use a much more concise, Perlish style:

```
for my $j (0..6) {...}
```

instead of the syntax used in some other languages:

```
for (my $j=0; $j<=6; $j++) {...}
```

It’s much simpler to write and comprehend code like this:

```
print "something" if $debug;
```

than this:

```
if($debug){
  print "something";
}
```

A good style that improves understanding, readability and reduces the chances of having a bug is shown below in the form of yet another rewrite of our *‘one week ago’* code:

```
for (0..6) {
  $day--;
  next if $day;

  $month--;
  unless ($month){
    $year--;
    $month=12
  }

  if($month == 1){
    $day = (($year % 4 or $year % 100 == 0) and $year % 400 )
         ? 28 : 29;
  } else {
    $day = $month_len[$month-1];
  }
}
```

which is a happy medium between the excessively verbose style of the first version and very obscure second version.

And of course a two liner, which is much faster and easier to understand is:

```
sub date_a_week_ago{
  my ($day,$month,$year) = (localtime(time-604800))[3..5];
  return sprintf "%02d/%02d/%04d",$month+1,$day,$year+1900;
}
```

Just take the current date in seconds since epoch as time() returns, subtract a week in seconds (7*24*60*60 = 604800) and feed the result to localtime() - voila we’ve got the date of one week ago!

Why is the last version important, when the first one works just fine? Not because of performance issues (although this last one is twice as fast as the first), but because there are more ways to put a bug in the first version than there are in the last one.

17.7.7 Introduction to the Perl Debugger

As we saw earlier, it's *almost* always possible to debug code with the help of print(). However, it is impossible to anticipate all the possible directions which a program might take, and difficult to know what code to suspect when trouble occurs. In addition, inline debugging code tends to add bloat and degrade the performance of an application, although most applications offer inline debugging as a compile time option to avoid these hits. In any case, this information tends to only be useful to the programmer who added the print statements in the first place.

Sometimes you have to debug tens of thousands lines of Perl in an application, and while you may be a very experienced Perl programmer who can understand Perl code quite well by just looking at it, no mere mortal can even begin to understand what will actually happen in such a large application, until the code is running. So you just don't know where to start adding your trusty print() statements to see what is happening inside.

The most effective way to track down a bug is to run the program inside an interactive debugger. The majority of programming languages have such a tool available, allowing one to see what is happening inside an application while it is running. The basic features of an interactive debugger allow you to:

- Stop at a certain point in the code, based on a routine name or source file and line number
- Stop at a certain point in the code, based on conditions such as the value of a given variable
- Perform an action without stopping, based on the criteria above
- View and modify the value of variables at any given point
- Provide context information such as stack traces and source windows

It does take practice to learn the most effective ways of using an interactive debugger, but the time and effort will be paid back many-fold in the long run.

Most C and C++ programmers are familiar with the interactive GNU debugger (`gdb`). `gdb` is a stand-alone program that requires your code to be compiled with debugging symbols to be useful. While `gdb` can be used to debug the Perl interpreter itself, it cannot be used to debug your Perl scripts.

Not to worry, Perl provides its own interactive debugger, called `perldb`. Giving control of your Perl program to the interactive debugger is simply a matter of specifying the `-d` command line switch. When this switch is used, Perl inserts debugging hooks into the program syntax tree, but it leaves the job of debugging to a Perl module separate from the perl binary itself.

I will start by introducing a few of the basic concepts and commands of the Perl interactive debugger. These warm-up examples all run from the command line, independent of mod_perl, but are all still relevant when we do finally go inside Apache.

It might be useful to keep the *perldebug* manpage handy for reference while reading this section, and for future debugging sessions on your own.

The interactive debugger will attach to the current terminal and present you with a prompt just before the first program statement is executed. For example:

```
% perl -d -le 'print "mod_perl rules the world"'

Loading DB routines from perl5db.pl version 1.0402

Emacs support available.

Enter h or 'h h' for help.

main::(-e:1):   print "mod_perl rules the world"
  DB<1>
```

The source line shown is the line which Perl is *about* to execute, the `next` command (or just `n`) will cause this line to be executed after which execution will stop again just before the next line:

```
main::(-e:1):   print "mod_perl rules the world"
  DB<1> n
mod_perl rules the world
Debugged program terminated.  Use q to quit or R to restart,
use O inhibit_exit to avoid stopping after program termination,
h q, h R or h O to get additional info.
DB<1>
```

In this case, our example code is only one line long, so we have finished interacting after the first line of code is executed. Let's try again with slightly longer example which is the following script:

```
my $word = 'mod_perl';
my @array = qw(rules the world);

print "$word @array\n";
```

Save the script in a file called *domination.pl* and run with the `-d` switch:

```
% perl -d domination.pl

main::(domination.pl:1):      my $word = 'mod_perl';
  DB<1> n
main::(domination.pl:2):      my @array = qw(rules the world);
  DB<1>
```

At this point, the first line of code has been executed and the variable `$word` has been assigned the value *mod_perl*. We can check this by using the `p` command (an abbreviation for the `print` command, the two are interchangeable):

```
main::(domination.pl:2):      my @array = qw(rules the world);
  DB<1> p $word
mod_perl
```

The `print` command works just like the Perl's built-in print() function, but adds a trailing newline and outputs to the `$DB::OUT` file handle, which is normally opened on the terminal where Perl was launched from. Let's carry on:

```
  DB<2> n
main::(domination.pl:4):      print "$word @array\n";
  DB<2> p @array
rulestheworld
  DB<3> n
mod_perl rules the world
Debugged program terminated.  Use q to quit or R to restart,
use O inhibit_exit to avoid stopping after program termination,
h q, h R or h O to get additional info.
```

Ouch, `p @array` printed `rulestheworld` and not `rules the world`, as you might expect it to, but that's absolutely correct. If you print an array without expanding it first into a string it will be printed without adding the content of the `$"` variable (otherwise known as `$LIST_SEPARATOR` if the `English` pragma is being used) between the elements of the array.

If you type:

```
print "@array";
```

the output will be `rules the world` since the default value of the `$"` variable is a single space.

You should have noticed by now that there is some valuable information to the left of each executable statement:

```
main::(domination.pl:4):      print "$word @array\n";
  DB<2>
```

First is the current package name, in this case `main::`. Next is the current filename and statement line number, *domination.pl* and 4 in the example above. The number presented at the prompt is the command number which can be used to recall commands from the session history, using the `!` command followed by this number. For example, `!1` would repeat the first command:

```
% perl -d -e0

main::(-e:1):   0
  DB<1> p $]
5.00503
  DB<2> !1
p $]5.00503
  DB<3>
```

Where `$]` is the perl's version number. As you see `!1` prints the value of `$]`, preceded by the command that was executed.

Things start to get more interesting as the code does. In the example script below (save it to a file called *test.pl*) we've increased the number of source files and packages by including the standard `Symbol` module, along with an invocation of its gensym() function:

```
use Symbol ();

my $sym = Symbol::gensym();

print "$sym\n";
```

```
% perl -d test.pl

main::(test.pl:3):      my $sym = Symbol::gensym();
  DB<1> n
main::(test.pl:5):      print "$sym\n";
  DB<1> n
GLOB(0x80c7a44)
```

First, notice the debugger did not stop at the first line of the file. This is because `use ...` is a compile-time statement, not a run-time statement. Also notice there was more work going on than the debugger revealed. That's because the `next` command does not enter subroutine calls. To step into a subroutine code use the `step` command (or its abbreviated form `s`):

```
% perl -d test.pl

main::(test.pl:3):      my $sym = Symbol::gensym();
  DB<1> s
Symbol::gensym(/usr/lib/perl5/5.00503/Symbol.pm:86):
86:         my $name = "GEN" . $genseq++;
  DB<1>
```

Notice the source line information has changed to the `Symbol::gensym` package and the `Symbol.pm` file. We can carry on by hitting the return key at each prompt, which causes the debugger to repeat the last `step` or `next` command. It won't repeat a `print` command though. The debugger will eventually return from the subroutine back to our main program:

```
  DB<1>
Symbol::gensym(/usr/lib/perl5/5.00503/Symbol.pm:87):
87:         my $ref = \*{$genpkg . $name};
  DB<1>
Symbol::gensym(/usr/lib/perl5/5.00503/Symbol.pm:88):
88:         delete $$genpkg{$name};
  DB<1>
Symbol::gensym(/usr/lib/perl5/5.00503/Symbol.pm:89):
89:         $ref;
  DB<1>
main::(test.pl:5):      print "$sym\n";
  DB<1>
GLOB(0x80c7a44)
```

Our line-by-line debugging approach has served us well for this small program, but imagine the time it would take to step through a large application at the same pace. There are several ways to speed up a debugging session, one of which is known as *setting a breakpoint*. The `breakpoint` command (`b`) can be used for instructing the debugger to stop at a named subroutine or at any line of any file. In this example session, at the first debugger prompt we will set a breakpoint at the `Symbol::gensym` subroutine, telling the debugger to stop at the first line of this routine when it is called. Rather than move along with `next` or `step` we give the `continue` command (`c`) which tells the debugger to execute the script without stopping until it reaches a breakpoint:

```
% perl -d test.pl

main::(test.pl:3):      my $sym = Symbol::gensym();
  DB<1> b Symbol::gensym
  DB<2> c
Symbol::gensym(/usr/lib/perl5/5.00503/Symbol.pm:86):
86:         my $name = "GEN" . $genseq++;
```

Now let's pretend we are debugging a large application where `Symbol::gensym` might be called in various places. When the subroutine breakpoint is reached, by default the debugger does not reveal where it was called from. One way to find out this information is with the `Trace` command (`T`):

```
  DB<2> T
$ = Symbol::gensym() called from file 'test.pl' line 3
```

In this example, the call stack is only one level deep, so only that line is printed. We'll look at an example with a deeper stack later. The left-most character reveals the context in which the subroutine was called. `$` represents scalar context, in other examples you may see `@` which represents list context or `.` which represents void context. In our case we have called:

```
my $sym = Symbol::gensym();
```

which calls the `Symbol::gensym()` in scalar context.

Below we've made our *test.pl* example a little more complex. First, we've added a `My::World` package declaration at the top of the script, so we are no longer working in the `main::` package. Next, we've added a subroutine named do_work() which invokes the familiar `Symbol::gensym`, along with another function called `Symbol::qualify` and then returns a hash reference of the results. The do_work() routine is invoked inside a *for* loop which will be run twice:

```
package My::World;

use Symbol ();

for (1,2) {
  do_work("now");
}

sub do_work {
  my ($var) = @_;

  return undef unless $var;

  my $sym  = Symbol::gensym();
  my $qvar = Symbol::qualify($var);

  my $retval = {
               'sym' => $sym,
               'var' => $qvar,
               };

  return $retval;
}
```

We'll start by setting a few breakpoints and then we use the `List` command (`L`) to display them:

```
% perl -d test.pl

My::World::(test.pl:5):   for (1,2) {
  DB<1> b Symbol::qualify
  DB<2> b Symbol::gensym
  DB<3> L
/usr/lib/perl5/5.00503/Symbol.pm:
 86:        my $name = "GEN" . $genseq++;
   break if (1)
 95:        my ($name) = @_;
   break if (1)
```

The filename and line number of the breakpoint are displayed just before the source line itself. Because both breakpoints are located in the same file, the filename is displayed only once. After the source line we see the condition on which to stop. In this case, as the constant value 1 indicates, we will always stop at these breakpoints. Later on you'll see how to specify a condition.

As we will see, when the `continue` command is executed, the execution of the program stops at one of these breakpoints, either on line 86 or 95 of the `/usr/lib/perl5/5.00503/Symbol.pm` file, whichever is reached first. The displayed code lines are the first rows of the two subroutines from `Symbol.pm`. Breakpoints may only be applied to lines of run-time executable code, you cannot put breakpoints on empty lines or comments for example.

In our example the `List` command shows which lines the breakpoints were set on, but we cannot tell which breakpoint belongs to which subroutine. There are two ways to find this out. One is to run the `continue` command and when it stops, execute the `Trace` command we saw before:

```
  DB<3> c
Symbol::gensym(/usr/lib/perl5/5.00503/Symbol.pm:86):
86:        my $name = "GEN" . $genseq++;
  DB<3> T
$ = Symbol::gensym() called from file 'test.pl' line 14
. = My::World::do_work('now') called from file 'test.pl' line 6
```

So we see that it was `Symbol::gensym`. The other way is to ask for a listing of a range of lines from the code. For example, let's check which subroutine line 86 is a part of. We use the `list` (lowercase!) command (`l`), which displays parts of the code. The `list` command accepts various arguments, the one that we want to use here is a range of lines. Since the breakpoint is at line 86, let's print a few lines above and below that line:

```
  DB<3> l 85-87
85      sub gensym () {
86==>b      my $name = "GEN" . $genseq++;
87:         my $ref = \*{$genpkg . $name};
```

Now we know it's the `gensym` sub and we also see the breakpoint displayed with the help of the `==>b` markup. We could also use the name of the sub to display its code:

```
  DB<4> l Symbol::gensym
85      sub gensym () {
86==>b      my $name = "GEN" . $genseq++;
87:         my $ref = \*{$genpkg . $name};
88:         delete $$genpkg{$name};
89:         $ref;
90      }
```

The `delete` command (`d`) is used to remove a breakpoint by specifying the line number of the breakpoint. Let's remove the first one:

```
  DB<5> d 95
```

The `Delete` command (with a capital 'D') or `D` removes all currently installed breakpoints.

Now let's look again at the trace produced at the breakpoint:

```
  DB<3> c
Symbol::gensym(/usr/lib/perl5/5.00503/Symbol.pm:86):
86:         my $name = "GEN" . $genseq++;
  DB<3> T
$ = Symbol::gensym() called from file 'test.pl' line 14
. = My::World::do_work('now') called from file 'test.pl' line 6
```

As you can see, the stack trace prints the values which are passed into the subroutine. Ah, and perhaps we've found our first bug, as we can see do_work() was called in void context, so the return value was lost into thin air. Let's change the *'for'* loop to check the return value of do_work():

```
for (1,2) {
  my $stuff = do_work("now");
  if ($stuff) {
      print "work is done\n";
  }
}
```

In this session we will set a breakpoint at line 7 of `test.pl` where we check the return value of do_work():

```
% perl -d test.pl

My::World::(test.pl:5):   for (1,2) {
  DB<1> b 7
  DB<2> c
My::World::(test.pl:7):     if ($stuff) {
  DB<2>
```

Our program is still small, but already it is getting more difficult to understand the context of just one line of code. The `window` command (`w`) will list a few lines of code that surround the current line:

```
  DB<2> w
4
5:        for (1,2) {
6:          my $stuff = do_work("now");
7==>b       if ($stuff) {
8:              print "work is done\n";
9           }
10        }
11
12        sub do_work {
13:         my ($var) = @_;
```

The arrow points to the line which is about to be executed and also contains a `'b'` indicating that we have set a breakpoint at this line. The breakable lines of code include a `':'` immediately after the line number.

Please notice that this demonstration was done before perl 5.8 was released, which redefined some of the letters to have a different meaning. For example `w` was replaced with `v`. Please see the `perldebug` manpage.

Now, let's take a look at the value of the `$stuff` variable with the trusty old `print` command:

```
  DB<2> p $stuff
HASH(0x82b89b4)
```

That's not very useful information. Remember, the `print` command works just like the built-in print() function does. The debugger's `x` command evaluates a given expression and prints the results in a "pretty" fashion:

```
  DB<3> x $stuff
0  HASH(0x82b89b4)
   'sym' => GLOB(0x826a944)
      -> *Symbol::GEN0
   'var' => 'My::World::now'
```

There, things seem to be okay, let's double check by calling do_work() with a different value and print the results:

```
  DB<4> x do_work('later')
0  HASH(0x82bacc8)
   'sym' => GLOB(0x818f16c)
      -> *Symbol::GEN1
   'var' => 'My::World::later'
```

We can see the symbol was incremented from `GEN0` to `GEN1` and the variable later was qualified, as expected.

Now let's change the test program a little to iterate over a list of arguments held in `@args` and print a slightly different message:

```
package My::World;

use Symbol ();
```

```
my @args = qw(now later);
for my $arg (@args) {
  my $stuff = do_work($arg);
  if ($stuff) {
      print "do your work $arg\n";
  }
}

sub do_work {
  my ($var) = @_;

  return undef unless $var;

  my $sym = Symbol::gensym();
  my $qvar = Symbol::qualify($var);

  my $retval = {
      'sym' => $sym,
      'var' => $qvar,
  };

  return $retval;
}
```

There are only two arguments in the list, so stopping to look at each one isn't too time consuming, but consider the debugging pace if we had a large list of 100 or so entries. It is possible to customize breakpoints by specifying a condition. Each time a breakpoint is reached, the condition is evaluated, stopping only if the condition is true. In the session below, the `window` command shows breakable lines and we set a breakpoint at line 7 with the condition `$arg eq 'later'`. As we continue, the breakpoint is skipped when `$arg` has the value of *now* but not when it has the value of *later*:

```
% perl -d test.pl

My::World::(test.pl:5): my @args = qw(now later);
  DB<1> w
2
3:      use Symbol ();
4
5==>    my @args = qw(now later);
6:      for my $arg (@args) {
7:          my $stuff = do_work($arg);
8:          if ($stuff) {
9:              print "do your work $arg\n";
10         }
11     }
```

The `==>` symbol shows us the line of code that's about to be executed.

```
  DB<1> b 7 $arg eq 'later'
  DB<2> c
do your work now
My::World::(test.pl:7):     my $stuff = do_work($arg);
  DB<2> n
My::World::(test.pl:8):     if ($stuff) {
  DB<2> x $stuff
```

```
0  HASH(0x82b90e4)
   'sym' => GLOB(0x82b9138)
      -> *Symbol::GEN1
   'var' => 'My::World::later'
  DB<5> c
do your work later
Debugged program terminated.  Use q to quit or R to restart,
```

There are plenty more tricks left to pull from the perldb bag, but you should now understand enough about the debugger to try them on your own with the perldebug manpage by your side. Quick online help from inside the debugger can be reached by typing the `h` command. It will display a list of the most useful commands and a short explanation of what they do.

17.7.8 Interactive Perl Debugging under mod_cgi

`Devel::ptkdb` is a visual Perl debugger that uses perlTk for the user interface and requires a windows system like X-Windows or Windows to run.

To debug a plain perl script with ptkdb, invoke it as:

```
% perl -d:ptkdb myscript.pl
```

The Tk application will be loaded. Now you can do most of the debugging you did with the command line Perl debugger, but using a simple GUI to set/remove breakpoints, browse the code, step through it and more.

With the help of ptkdb you can debug your CGI scripts running under mod_cgi. Be sure that the web server's Perl installation includes the Tk package. In order to enable the debugger you should change your "shebang" line from

```
#! /usr/local/bin/perl -Tw
```

to

```
#! /usr/local/bin/perl -Twd:ptkdb
```

You can debug scripts remotely if you're using a Unix based server and if the machine where you are writing the script has an X-server. The X-server can be another Unix workstation, or a Macintosh or Win32 platform with an appropriate X-Windows package. You must insert the following `BEGIN` subroutine into your script:

```
BEGIN {
  $ENV{'DISPLAY'} = "myHostname:0.0" ;
}
```

You can use either the IP (*123.123.123.123:0.0*) or the DNS convention (*myhost.com:0.0*). You must be sure that your web server has permission to open windows on your X-server (see the *xhost* manpage for more info).

Access the web page with the browser and *Submit* the script as normal. The ptkdb window should appear on the monitor if you have correctly set the `$ENV{'DISPLAY'}` variable. At this point you can start debugging your script. Be aware that the browser may timeout waiting for the script to run.

To expedite debugging you may want to set your breakpoints in advance with a *.ptkdbrc* file and use the `$DB::no_stop_at_start` variable. NOTE: for debugging web scripts you may have to have the *.ptkdbrc* file installed in the server account's home directory (~www) or whatever username the webserver is running under. Also try installing a *.ptkdbrc* file in the same directory as the target script.

META: insert snapshots of ptkdb screen

ptkdb is not part of the standard perl distribution; it is available from CPAN: http://www.perl.com/CPAN/authors/id/A/AE/AEPAGE/

Some fixes for ptkdb to work under mod_perl are listed here:
http://article.gmane.org/gmane.comp.apache.mod-perl/7562

17.7.9 Non-Interactive Perl Debugging under mod_perl

To debug scripts running under mod_perl either use Apache::DB (interactive Perl debugging) or an older non-interactive method as described below.

The `NonStop` debugger option enables you to get some decent debugging information when running under mod_perl. For example, before starting the server:

```
% setenv PERL5OPT -d
% setenv PERLDB_OPTS "NonStop=1 LineInfo=db.out AutoTrace=1 frame=2"
```

Now watch db.out for line:filename info. This is most useful for tracking those core dumps that normally leave us guessing, even with a stack trace from gdb. *db.out* will show you what Perl code triggered the core dump. *'man perldebug'* for more `PERLDB_OPTS`. Note that Perl will ignore `PERL5OPT` if `PerlTaintCheck` is `On`.

17.7.10 Interactive mod_perl Debugging

Now we'll turn to looking at how the interactive debugger is used in a mod_perl environment. The `Apache::DB` module available from CPAN provides a wrapper around `perldb` for debugging Perl code running under mod_perl.

The server must be run in non-forking mode to use the interactive debugger, this mode is turned on by passing the `-X` flag to the httpd executable. It is convenient to use an `IfDefine` section around the `Apache::DB` configuration, the example below does this using the name *PERLDB*. With this setup, debugging is only turned on when starting the server with the `httpd -D PERLDB` command.

This section should be at the top of the Perl configuration section of the configuration file, before any other Perl code is pulled in, so that debugging symbols will be inserted into the syntax tree, triggered by the call to `Apache::DB->init`. The `Apache::DB::handler` can be configured using any of the `Perl*Handler` directives, in this case you use a `PerlFixupHandler` so handlers in the response

phase will bring up the debugger prompt:

```
<IfDefine PERLDB>

  <Perl>
    use Apache::DB ();
    Apache::DB->init;
  </Perl>

  <Location />
    PerlFixupHandler Apache::DB
  </Location>

</IfDefine>
```

Since we have used `/` as the argument to the `Location` directive, the debugger will be invoked for any kind of request (even for static documents and images) but of course it will immediately quit unless there is some Perl module registered to handle these requests.

In our first example, we will debug the standard `Apache::Status` module, which is configured like this:

```
PerlModule Apache::Status
<Location /perl-status>
  PerlHandler Apache::Status
  SetHandler perl-script
</Location>
```

When the server is started with the debugging flag, a notice will be printed to the console:

```
% httpd -X -D PERLDB
[notice] Apache::DB initialized in child 950
```

The debugger prompt will not be available until the first request is made, in our case to http://localhost/perl-status. Once we are at the prompt, all the standard debugging commands are available. First we run window to get some of the context for the code being debugged, then we move to the next statement after a value has been assigned to `$r`, and finally we print the request URI. If no breakpoints are set, the `continue` command will give control back to Apache and the request will finish with the `Apache::Status` main menu showing in the browser window:

```
Loading DB routines from perl5db.pl version 1.0402
Emacs support available.

Enter h or ‘h h’ for help.

Apache::Status::handler(/usr/lib/perl5/site_perl/5.005/i386-linux/Apache/Status.pm:55):
55:         my ($r) = @_;
  DB<1> w
52      }
53
54      sub handler {
55==>       my ($r) = @_;
56:         Apache->request($r); #for Apache::CGI
57:         my $qs = $r->args || "";
```

```
  58:         my $sub = "status_$qs";
  59:         no strict 'refs';
  60
  61:         if($qs =~ s/^(noh_\w+).*/$1/) {
    DB<1> n
 Apache::Status::handler(/usr/lib/perl5/site_perl/5.005/i386-linux/Apache/Status.pm:56):
  56:         Apache->request($r); #  for Apache::CGI
    DB<1> p $r->uri
  /perl-status
    DB<2> c
```

All the techniques we saw while debugging plain perl scripts can be applied to this debugging session.

Debugging `Apache::Registry` scripts is somewhat different, because the handler routine does quite a bit of work before it reaches your script. In this example, we make a request for `/perl/test.pl`, which consists of this code:

```
  use strict;

  my $r = shift;
  $r->send_http_header('text/plain');

  print "mod_perl rules";
```

When a request is issued, the debugger stops at line 28 of *Apache/Registry.pm*. We set a breakpoint at line 140, which is the line that actually calls the script wrapper subroutine. The `continue` command will bring us to that line, where we can step into the script handler:

```
  Apache::Registry::handler(/usr/lib/perl5/site_perl/5.005/i386-linux/Apache/Registry.pm:28):
28:         my $r = shift;
    DB<1> b 140
    DB<2> c
  Apache::Registry::handler(/usr/lib/perl5/site_perl/5.005/i386-linux/Apache/Registry.pm:140):
  140:            eval { &{$cv}($r, @_) } if $r->seqno;
    DB<2> s
  Apache::ROOT::perl::test_2epl::handler((eval 87):3):
  3:        my $r = shift;
```

Notice the funny package name, that's generated from the URI of the request for namespace protection. The filename is not displayed, since the code was compiled via eval(), but the `print` command can be used to show you `$r->filename`:

```
    DB<2> n
  Apache::ROOT::perl::test_2epl::handler((eval 87):4):
  4:        $r->send_http_header('text/plain');
    DB<2> p $r->filename
  /home/httpd/perl/test.pl
```

The line number might seem off too, but the window command will give you a better idea where you are:

```
  DB<4> w
1:      package Apache::ROOT::perl::test_2epl;use Apache qw(exit);
sub handler {  use strict;
2
3:        my $r = shift;
4==>      $r->send_http_header('text/plain');
5
6:        print "mod_perl rules";
7
8       }
9       ;
```

The code from the *test.pl* file is between lines 2 and 7, the rest is the `Apache::Registry` magic to cache your code inside a *handler* subroutine.

It will always take some practice and patience when putting together debugging strategies that make effective use of the interactive debugger for various situations. Once you have a good strategy, bug squashing can actually be quite a bit of fun!

17.7.11 ptkdb and Interactive mod_perl Debugging

As you saw earlier you can use the `ptkdb` visual debugger to debug CGI scripts running under mod_cgi. But it won't work for mod_perl using the same configuration as used in mod_cgi. We have to tweak the *Apache/DB.pm* module to use *Devel/ptkdb.pm* instead of *Apache/perl5db.pl*.

Open the file in your favorite editor and replace:

```
require 'Apache/perl5db.pl';
```

with:

```
require 'Devel/ptkdb.pm';
```

Now when you use the interactive mod_perl debugger configuration from the previous section and issue a request, the *ptkdb* visual debugger will be loaded.

If you are debugging `Apache::Registry` scripts, as in the terminal debugging mode example, go to line 140 (or to whatever line the `eval { &{$cv}($r, @_) } if $r->seqno;` statement is located) and press the *step in* button to start the debug of the script itself.

Note that you can use Apache with `ptkdb` in plain multi-server mode, you don't have to start `httpd` with the `-X` option.

META: One caveat:

When the request is completed, `ptkdb` hangs. Does anyone know what code should be registered for it to exit on completion? To replace the original `Apache::DB` cleanup code, as:

```
if (ref $r) {
$SIG{INT} = \&DB::catch;
$r->register_cleanup(sub {
    $SIG{INT} = \&DB::ApacheSIGINT();
});
}
```

Any Perl/Tk guru to assist???

17.7.12 Debugging when Server Crashes on Startup before Writing to Log File.

If your server crashes on startup, you need to start it under gdb and ask it to generate a stack trace.

I'll emulate a faulty server by starting a startup file with the dump() command:

```
startup.pl
----------
dump;
1;
```

and then requiring this file from the *httpd.conf*:

```
PerlRequire /path/to/startup.pl
```

Make sure no server is running on port 80 or use an alternate config with an alternate port if using a production server.

```
% gdb /path/to/httpd
(gdb) set args -X
```

Use:

```
set args -X -f /path/to/alternate/serverconfig_ifneeded.conf
```

if the server must be started from an alternative configuration file.

Now run the program:

```
(gdb) run

Starting program: /usr/local/apache/bin/httpd -X

Program received signal SIGABRT, Aborted.
0x400da4e1 in __kill () from /lib/libc.so.6
```

At this point the server should die because of the call to `dump()`. When that happens we use `bt` or `where` to ask for a stack back trace.

```
(gdb) where

#0  0x400da4e1 in __kill () from /lib/libc.so.6
#1  0x80d43bc in Perl_my_unexec ()
#2  0x8119544 in Perl_pp_goto ()
#3  0x8118990 in Perl_pp_dump ()
#4  0x812b2ad in Perl_runops_standard ()
#5  0x80d3a9c in perl_eval_sv ()
#6  0x807ef1c in perl_do_file ()
#7  0x807ef4f in perl_load_startup_script ()
#8  0x807b7ec in perl_cmd_require ()
#9  0x8092af7 in ap_clear_module_list ()
#10 0x8092f43 in ap_handle_command ()
#11 0x8092fd7 in ap_srm_command_loop ()
#12 0x80933e0 in ap_process_resource_config ()
#13 0x8093ca2 in ap_read_config ()
#14 0x809db63 in main ()
#15 0x400d41eb in __libc_start_main (main=0x809d8dc <main>, argc=2,
    argv=0xbffffab4, init=0x80606f8 <_init>, fini=0x812b38c <_fini>,
    rtld_fini=0x4000a610 <_dl_fini>, stack_end=0xbffffaac)
    at ../sysdeps/generic/libc-start.c:90
```

If you do not know what this trace means, you could send it to the mod_perl mailing list to ask for help. Make sure to include the version numbers of Apache, mod_perl and Perl, and use a subject line that says something about the problem rather than ’help’.

In our case we already know that the server is supposed to die when compiling the startup file and we can clearly see that from the trace. We always read it from the bottom upward:

We are in config file:

```
#13 0x8093ca2 in ap_read_config ()
```

We do require:

```
#8  0x807b7ec in perl_cmd_require ()
```

We load the file and compile it:

```
#6  0x807ef1c in perl_do_file ()
#5  0x80d3a9c in perl_eval_sv ()
```

`dump()` gets executed:

```
#3  0x8118990 in Perl_pp_dump ()
```

`dump()` calls `__kill()`:

```
#0  0x400da4e1 in __kill () from /lib/libc.so.6
```

17.8 Hanging Processes: Detection and Diagnostics

Sometimes a httpd process might hang in the middle of processing a request, either because there is a bug in your code (e.g. the code is stuck in a while loop), it gets blocked by some system call or because of a resource deadlock) or for some other reason. In order to fix the problem we need to learn what circumstances the process hangs in (detection), so we can reproduce the problem and after that to discover why there is problem (diagnostics).

17.8.1 Hanging because of the OS Problem

Sometimes you can find a process hanging because of some kind of the system problem. For example if the processes was doing some disk IO operation it might get stuck in uninterruptable sleep (`'D'` disk wait in ps(1) report, `'U'` in top(1)) which indicates that either something is broken in your kernel or that you're using NFS. Or and you cannot kill -9 this process.

Another process that cannot be killed with kill -9 is a zombie process (`'Z'` disk wait in ps(1) report, `<defunc>` in top(1)), in which case the process is already dead and Apache didn't wait on it properly.

In the case of *disk wait* you can actually get the *wait* channel from ps(1) and look it up in your kernel symbol table to find out what resource it was waiting on. It might point the way to what component of the system was misbehaving if the problem occurred frequently.

17.8.2 An Example of Code that Might Hang a Process

Deadlock is the situation where, for example, two processes, say X and Y, need two resources, A and B to continue. X holds onto A and Y holds onto B. There is no possibility for Y to continue before X releases A. But X cannot release A before it gets Y.

Look at the following example. Your process has to gain a lock on some resource (e.g. a file) before it continues. So it makes an attempt, and if that fails it sleep()s for a second and increments a counter:

```
until(gain_lock()){
  $tries++;
  sleep 1;
}
```

Because there are many processes competing for this resource, or perhaps because there is a deadlock, gain_lock() always fails. The process is hung.

Another situation that you may very often encounter is exclusive lock starvation. Generally there are two lock types in use: *SHARED* locks, which allow many processes to perform *READ* operations simultaneously, and *EXCLUSIVE* locks. The latter permits access only by a single process and so makes a safe *WRITE* operation possible.

You can lock any kind of resource, although in our examples we will talk about files.

If there is a *READ* lock request, it is granted as soon as the file becomes unlocked or immediately if it is already *READ* locked. The lock status becomes *READ* on success.

If there is a *WRITE* lock request, it is granted as soon as the file becomes unlocked. Lock status becomes *WRITE* on success.

Normally it is the *WRITE* lock request which is the most important. If the file is being *READ* locked, a process that requests to write will poll until there are no reading or writing process left. However, lots of processes can successfully read the file, since they do not block each other from doing so. This means that a process that wants to write to the file (first obtaining an exclusive lock) never gets a chance to squeeze in. The following diagram represents a possible scenario where everybody can read but no one can write:

```
[-p1-]                 [--p1--]
   [--p2--]
 [---------p3---------]
              [------p4-----]
   [--p5--]   [----p5----]
```

Let's look at some real code and see it in action. The following script imports flock() related parameters from the `Fcntl` module, and opens a file that will be locked. It then defines and sets two variables: `$lock_type` and `$lock_type_verbose`. These are set to `LOCK_EX` and `EX` respectively if the first command line argument (`$ARGV[0]) is defined and equal to w. This indicates that this process will try to gain a WRITE (exclusive) lock. Otherwise the two are set to LOCK_SH and <SH` for a *SHARED* (read) lock.

Once the variables are set, we enter the infinite `while(1)` loop that attempts to lock the file by the mode set in `$lock_type`. It report success and the type of lock that was gained, then it sleeps for a random period between 0 and 9 seconds and unlocks the file. The loop then starts from the beginning.

```
lock.pl
--------------------
#!/usr/bin/perl -w
use Fcntl qw(:flock);

$lock = "/tmp/lock";

open LOCK, ">$lock" or die "Cannot open $lock for writing: $!";
my $lock_type         = LOCK_SH;
my $lock_type_verbose = 'SH';
if (defined $ARGV[0] and $ARGV[0] eq 'w'){
  $lock_type         = LOCK_EX;
  $lock_type_verbose = 'EX';
}

while(1){
  flock LOCK,$lock_type;
    # start of critical section
  print "$$: $lock_type_verbose\n";
  sleep int(rand(10));
    # end of critical section
  flock LOCK, LOCK_UN;
}
close LOCK;
```

It’s very easy to see *WRITE* process starvation if you spawn a few of the above scripts simultaneously. Start the first few as *READ* processes and then start one *WRITE* process like this:

```
% ./lock.pl r & ; ./lock.pl r & ; ./lock.pl r & ; ./lock.pl w &
```

You see something like:

```
24233: SH
24232: SH
24232: SH
24233: SH
24232: SH
24233: SH
24231: SH
24231: SH
24231: SH
```

and not a single `EX` line... When you kill off the reading processes, then the write process will gain its lock. Note that as this is a rough example, I used the sleep() function. To simulate a real situation you need to use the `Time::HiRes` module, which allows you to choose more precise intervals to sleep.

The interval between lock and unlock is called a *Critical Section*, which should be kept as short as possible (in terms of the time taken to execute the code, and not in terms of the number of lines of code). As you just saw, a single sleep statement can make the critical section long.

To summarize, if you have a script that uses both *READ* and *WRITE* locks and the critical section isn’t very short, the writing process might be starved. After a while a browser that initiated this request will timeout the connection and abort the request, but it’s much more likely that user will press the *Stop* or *Reload* button before that happens. Since the process in question is just waiting, there is no way for Apache to know that the request was aborted. It will hang until the lock is gained. Only when a write to a client’s broken connection is attempted will Apache terminate the script.

17.8.3 Detecting hanging processes

It’s not so easy to detect hanging processes. There is no way you can tell how long the request is taking to process by using plain system utilities like ps() and top(). The reason is that each Apache process serves many requests without quitting. System utilities can tell how long the process has been running since its creation, but this information is useless in our case, since Apache processes normally run for extended periods.

However there are a few approaches that can help to detect a hanging process.

If the process hangs and demands lots of resources it’s quite easy to spot it by using the top() utility. You will see the same process show up in the first few lines of the automatically refreshed report. But often the hanging process uses few resources, e.g. when waiting for some event to happen.

Another easy case is when some process thrashes the *error_log*, writing millions of error messages there. Generally this process uses lots of resources and is also easily spotted by using top().

There are other tools that report the status of Apache processes.

- **The mod_status module, which is usually accessed from the */server_status* location.**
- **The `Apache::VMonitor` module.**

Both tools provide counters of processed requests per Apache process.

You can watch the report for a few minutes, and try to spot any process which has the same number of processed requests while its status is 'W' (waiting). This means that it has hung.

But if you have fifty processes, it can be quite hard to spot such a process. Apache::Watchdog::RunAway is a hanging processes monitor and terminator that implements this feature and should be used to solve this kind of problem.

If you've got a real problem, and the processes hang one after the other, the time will come when the number of hanging processes is equal to the value of `MaxClients`. This means that no more processes will be spawned. As far as the users are concerned your server is down. It is easy to detect this situation, attempt to resolve it and notify the administrator using a simple crontab watchdog that requests some very light script periodically. (See Monitoring the Server. A watchdog.)

In the watchdog you set a timeout appropriate for your service, which may be anything from a few seconds to a few minutes. If the server fails to respond before the timeout expires, the watchdog has spotted trouble and attempts to restart the server. After a restart an email report is sent to the administrator saying that there was a problem and whether or not the restart was successful.

If you get such reports constantly something is wrong with your web service and you should revise your code. Note that it's possible that your server is being overloaded by more requests than it can handle, so the requests are being queued and not processed for a while, which triggers the watchdog's alarm. If this is a case you may need to add more servers or more memory, or perhaps split your single machine across a cluster of machines.

17.8.4 Determination of the reason

Given the process id (PID), there are three ways to find out where the server is hanging.

1. Deploying the Perl calls tracing mechanism. This will allow to spot the location of the Perl code that has triggered the problem.

2. Using the system calls tracing utilities, like strace(1) or truss(1). This approach reveals low level details about a potential misbehavior of some part of the system.

3. Using an interactive debugger, like gdb(1). When the process is stuck, and you don't know what it was doing just before it has got stuck, with gdb you can attach to this process and print its calls stack, to reveal where the last call was made from. Just like with strace or truss you see the system call trace and not the Perl calls.

17.8.4.1 Using the Perl Trace

To see where an httpd is "spinning", try adding this to your script or a startup file:

```
use Carp ();
$SIG{'USR2'} = sub {
   Carp::confess("caught SIGUSR2!");
};
```

The above code assigns a signal handler for the `USR2` signal. This signal has been chosen because it's least likely to be used by the other parts of the server.

We check the registered signal handlers with help of Apache::Status. What we see at http://localhost/perl-status?sig is :

```
USR2 = \&MyStartUp::__ANON__
```

`MyStartUp` is the name of the package I've used in mine *startup.pl*.

After applying this server configuration, let's use this simple code example, where sleep(10000) will emulate a hanging process:

```
debug/perl_trace.pl
-------------------
$|=1;
print "Content-type:text/plain\r\n\r\n";
print "[$$] Going to sleep\n";
hanging_sub();
sub hanging_sub {sleep 10000;}
```

We execute the above script as *http://localhost/perl/debug/perl_trace.pl*, we have used `$|=1;` and printed the PID with `$$` to learn what process ID we want to work with.

No we issue the command line, using the PID we have just saw being printed to the browser's window:

```
% kill -USR2 PID
```

And watch this showing up at the *error_log* file:

```
caught SIGUSR2!
    at /home/httpd/perl/startup/startup.pl line 32
MyStartUp::__ANON__('USR2') called
    at /home/httpd/perl/debug/perl_trace.pl line 5
Apache::ROOT::perl::debug::perl_trace_2epl::hanging_sub() called
    at /home/httpd/perl/debug/perl_trace.pl line 4
Apache::ROOT::perl::debug::perl_trace_2epl::handler('Apache=SCALAR(0x8309d08)')
  called
    at /usr/lib/perl5/site_perl/5.005/i386-linux/Apache/Registry.pm
      line 140
eval {...} called
    at /usr/lib/perl5/site_perl/5.005/i386-linux/Apache/Registry.pm
      line 140
```

```
Apache::Registry::handler('Apache=SCALAR(0x8309d08)') called
    at PerlHandler subroutine 'Apache::Registry::handler' line 0
eval {...} called
    at PerlHandler subroutine 'Apache::Registry::handler' line 0
```

We can clearly see that the process "hangs" in the code executed at line 5 of the */home/httpd/perl/debug/perl_trace.pl* script, and it was called by the hanging_sub() routine defined at line 4.

17.8.4.2 Using the System Calls Trace

Depending on the operating system you should have one of the `truss(1)` or `strace(1)` utilities available. In the following examples we will use `strace(1)`.

There are two ways to get the trace of the process with strace(1) (similar to gdb(1)). The first one is to tell strace(1) to start the process and do the tracing on it:

```
% strace perl -le 'print "mod_perl rules"'
```

The second is tell strace(1) to attach to the process that's already running. You need to know the PID of the process.

```
% strace -p PID
```

Replace PID with the process number you want to check on.

There are many more useful arguments accepted by strace(1) that you might find useful. For example you can tell it to trace only specific system calls:

```
% strace -e trace=open,write,close,nanosleep \
    perl -le 'print "mod_perl rules"'
```

In this example we have asked strace(1) to show us only the *open*, *write*, *close*, *nanosleep* which simplifies the observing of the output generated by strace(1) if you know what you are looking for.

Let's write a mod_perl script that hangs, and deploy `strace(1)` to find the point it hangs at:

```
hangme.pl
---------
$|=1;
my $r = shift;
$r->send_http_header('text/plain');

print "PID = $$\n";

while(1){
  $i++;
  sleep 1;
}
```

The reason this simple code hangs is obvious. It never breaks from the while loop. As you have noticed, it prints the PID of the current process to the browser. Of course in a real situation you cannot use the same trick. In the previous section I have presented a few ways to detect the runaway processes and their PIDs.

I save the above code in a file and execute it from the browser. Note that I've made STDOUT unbuffered with `$|=1;` so I will immediately see the process ID. Once the script is requested, the script prints the process PID and obviously hangs. So we press the `'Stop'` button, but the process continues to hang in this code. Isn't apache supposed to detect the broken connection and abort the request? *Yes* and *No*, you will understand soon what's really happening.

First let's attach to the process and see what it's doing. I use the PID the script printed to the browser, which is 10045 in this case:

```
% strace -p 10045

[...truncated identical output...]
SYS_175(0, 0xbffff41c, 0xbffff39c, 0x8, 0) = 0
SYS_174(0x11, 0, 0xbffff1a0, 0x8, 0x11) = 0
SYS_175(0x2, 0xbffff39c, 0, 0x8, 0x2)   = 0
nanosleep(0xbffff308, 0xbffff308, 0x401a61b4, 0xbffff308, 0xbffff41c) = 0
time([940973834])                       = 940973834
time([940973834])                       = 940973834
[...truncated the identical output...]
```

It isn't what we expected to see, is it? These are some system calls we don't see in our little example. What we actually see is how Perl translates our code into system calls. Since we know that our code hangs in this snippet:

```
while(1){
  $i++;
  sleep 1;
}
```

We *"easily"* figure out that the first three system calls implement the `$i++`, while the other three are responsible for the `sleep 1` call.

Generally the situation is the reverse of our example. You detect the hanging process, you attach to it and watch the trace of calls it does (or the last few commands if the process is hanging waiting for something, e.g. when blocking on a file lock request). From watching the trace you figure out what it's actually doing, and probably find the corresponding lines in your Perl code. For example let's see how one process *"hangs"* while requesting an exclusive lock on a file exclusively locked by another process:

```
excl_lock.pl
---------
use Fcntl qw(:flock);
use Symbol;

if ( fork() ) {
  my $fh = gensym;
  open $fh, ">/tmp/lock" or die "cannot open /tmp/lock $!";
  print "$$: I'm going to obtain the lock\n";
  flock $fh, LOCK_EX;
```

```
    print "$$: I’ve got the lock\n";
    sleep 20;
    close $fh;

  } else {
    my $fh = gensym;
    open $fh, ">/tmp/lock" or die "cannot open /tmp/lock $!";
    print "$$: I’m going to obtain the lock\n";
    flock $fh, LOCK_EX;
    print "$$: I’ve got the lock\n";
    sleep 20;
    close $fh;
  }
```

The code is simple. The process executing the code forks a second process, and both do the same thing: generate a unique symbol to be used as a file handler, open the lock file for writing using the generated symbol, lock the file in exclusive mode, sleep for 20 seconds (pretending to do some lengthy operation) and close the lock file, which also unlocks the file.

The `gensym` function is imported from the `Symbol` module. The `Fcntl` module provides us with a symbolic constant `LOCK_EX`. This is imported via the `:flock` tag, which imports this and other flock() constants.

The code used by both processes is identical, therefore we cannot predict which one will get its hands on the lock file and succeed in locking it first, so we add print() statements to find the PID of the process blocking (waiting to get the lock) on a lock request.

When the above code executed from the command line, we see that one of the processes gets the lock:

```
% ./excl_lock.pl

3038: I’m going to obtain the lock
3038: I’ve got the lock
3037: I’m going to obtain the lock
```

Here we see that process 3037 is blocking, so we attach to it:

```
% strace -p 3037

about to attach c10
flock(3, LOCK_EX
```

It’s clear from the above trace, that the process waits for an exclusive lock. (Note, that the missing closing parentheses is not a typo!)

As you become familiar with watching the traces of different processes, you will understand what is happening more easily.

17.8.4.3 Using the Interactive Debugger

Another approach to see a trace of the running code is to use a debugger such as `gdb` (the GNU debugger). It's supposed to work on any platform which supports the GNU development tools. Its purpose is to allow you to see what is going on *inside* a program while it executes, or what it was doing at the moment it crashed.

To trace the execution of a process, `gdb` needs to know the process id (PID) and the path to the binary that the process is executing. For Perl code it's */usr/bin/perl* (or whatever is the path to your Perl), for httpd processes it will be the path to your httpd executable.

Here are a few examples using gdb.

Let's go back to our last locking example, execute it as before and attach to the process that didn't get the lock:

```
% gdb /usr/bin/perl 3037
```

After starting the debugger we execute the `where` command to see the trace:

```
(gdb) where
#0  0x40131781 in __flock ()
#1  0x80a5421 in Perl_pp_flock ()
#2  0x80b148d in Perl_runops_standard ()
#3  0x80592b8 in perl_run ()
#4  0x805782f in main ()
#5  0x400a6cb3 in __libc_start_main (main=0x80577c0 <main>, argc=2,
    argv=0xbffff7f4, init=0x8056af4 <_init>, fini=0x80b14fc <_fini>,
    rtld_fini=0x4000a350 <_dl_fini>, stack_end=0xbffff7ec)
    at ../sysdeps/generic/libc-start.c:78
```

That's not what we expected to see and now it's a different trace. `#0` tells us the most recent call that was executed, which is a C language flock() implementation. But the previous call (`#1`) isn't print(), as we would expect, but a higher level of Perl's internal flock(). If we follow the trace of calls what we actually see is an Opcodes tree, which can be better presented as:

```
__libc_start_main
  main ()
    perl_run ()
      Perl_runops_standard ()
        Perl_pp_flock ()
          __flock ()
```

So I would say that it's less useful than `strace`, since if there are several flock()s it's almost impossible to know which of them was called. This problem is solved by `strace`, which shows the sequence of the system calls executed. Using this sequence we can locate the corresponding lines in the code.

(META: the above is wrong - you can ask to display the previous command executed by the program (not gdb)! What is it?)

When you attach to a running process with debugger, the program stops executing and control of the program is passed to the debugger. You can continue the normal program run with the `continue` command or execute it step by step with the `next` and `step` commands which you type at the `gdb` prompt. (`next` steps over any function calls in the line, while `step` steps into them).

C/C++ debuggers are a very large topic and beyond the scope of this document, but the gdb man page is quite good and you can try `info gdb` as well. You might also want to check the `ddd` (Data Display Debugger) which provides a visual interface to `gdb` and other debuggers. It even knows how to debug Perl programs!

For completeness, let's see the gdb trace of the httpd process that's still hanging in the `while(1)` loop of the first example in this section:

```
% gdb /usr/local/apache/bin/httpd 1005

(gdb) where
#0  0x4014a861 in __libc_nanosleep ()
#1  0x4014a7ed in __sleep (seconds=1) at ../sysdeps/unix/sysv/linux/sleep.c:78
#2  0x8122c01 in Perl_pp_sleep ()
#3  0x812b25d in Perl_runops_standard ()
#4  0x80d3721 in perl_call_sv ()
#5  0x807a46b in perl_call_handler ()
#6  0x8079e35 in perl_run_stacked_handlers ()
#7  0x8078d6d in perl_handler ()
#8  0x8091e43 in ap_invoke_handler ()
#9  0x80a5109 in ap_some_auth_required ()
#10 0x80a516c in ap_process_request ()
#11 0x809cb2e in ap_child_terminate ()
#12 0x809cd6c in ap_child_terminate ()
#13 0x809ce19 in ap_child_terminate ()
#14 0x809d446 in ap_child_terminate ()
#15 0x809dbc3 in main ()
#16 0x400d3cb3 in __libc_start_main (main=0x809d88c <main>, argc=1,
    argv=0xbffff7e4, init=0x80606f8 <_init>, fini=0x812b33c <_fini>,
    rtld_fini=0x4000a350 <_dl_fini>, stack_end=0xbffff7dc)
    at ../sysdeps/generic/libc-start.c:78
```

As before we can see a complete trace of the last executed call.

As you have noticed, I still haven't explained why the process hanging in the `while(1)` loop isn't aborted by Apache. The next section covers this.

To easily detect the hanging location, you can go through these steps while running gdb:

```
(gdb) where
(gdb) source ~/.gdbinit
(gdb) curinfo
```

(adjust the path to *.gdbinit* if needed.)

After loading the special macros file (*.gdbinit*) you can use the *curinfo* gdb macro to figure out the file and line number the code stuck in.

17.9 Debugging Hanging processes (continued)

META: incomplete

mod_perl comes with a number of useful of gdb macros to ease the debug process. You will find the file with macros in the mod_perl source distribution in the *.gdbinit* file (mod_perl-x.xx/.gdbinit). You might want to modify the macro definitions.

In order to use this you need to compile mod_perl with `PERL_DEBUG=1`.

To debug the server, start it:

```
% httpd -X
```

Issue a request to the offending script that hangs. Find the PID number of the process that hangs.

Go to the server root:

```
% cd /usr/local/apache
```

Now attach to it with gdb (replace the `PID` with the actual process id) and load the macros from *.gdbinit*:

```
% gdb /path/to/httpd PID
% source /usr/src/mod_perl-x.xx/.gdbinit
```

Now you can start the server (*httpd* below is a gdb macro):

```
(gdb) httpd
```

Now run the `curinfo` macro:

```
(gdb) curinfo
```

It should tell you the line/filename of the offending Perl code.

Add this to *.gdbinit*:

```
define longmess
  set $sv = perl_eval_pv("Carp::longmess()", 1)
  printf "%s\n", ((XPV*) ($sv)->sv_any )->xpv_pv
end
```

and when you reload the macros, run:

```
(gdb) longmess
```

to produce a Perl stacktrace.

17.9.1 Debugging core Dumping Code

```
$ perl -e dump
 Abort(coredump)
```

META: should I move the `Apache::StatINC` here? (I think not, since it relates to other topics like reloading config files, but you should mention it here with a pointer to it)

17.10 PERL_DEBUG=1 Build Option

Building mod_perl with `PERL_DEBUG=1`:

```
perl Makefile.PL PERL_DEBUG=1
```

will:

1. Add '-g' to EXTRA_CFLAGS
2. Turn on PERL_TRACE
3. Set PERL_DESTRUCT_LEVEL=2
4. Link against `libperld` if -e $Config{archlibexp}/CORE/libperld$Config{lib_ext}

17.11 Apache::Debug

(META: to be written)

```
use Apache::Debug ();
Apache::Debug::dump($r, SERVER_ERROR, "Uh Oh!");
```

This module sends what may be helpful debugging information to the client rather than to *error_log*.

Also, you could try using a larger emergency pool, try this instead of Apache::Debug:

```
$^M = 'a' x (1<<18);  #256k buffer
use Carp ();
$SIG{__DIE__} = \&Carp::confess;
eval { Carp::confess("init") };
```

17.12 Debug Tracing

To enable mod_perl debug tracing, configure mod_perl with the PERL_TRACE option:

```
perl Makefile.PL PERL_TRACE=1
```

The trace levels can then be enabled via the `MOD_PERL_TRACE` environment variable which can contain any combination of:

- **c**

  Trace directive handling during *Apache* (non-mod_perl) **c**onfiguration directive handling. (Startup.)

- **d**

  Trace directive handling during *mod_perl* **d**irective processing during configuration read. (Startup.)

- **s**

  Trace processing of <*Perl*> **s**ections. (Startup.)

- **h**

  Trace Perl **h**andler callbacks. (RunTime.)

- **g**

  Trace **g**lobal variable handling, interpreter construction, `END` blocks, etc. (RunTime.)

- **all**

  all of the options listed above. (Startup + RunTime.)

One way of setting this variable is by adding this directive to *httpd.conf*:

```
PerlSetEnv MOD_PERL_TRACE all
```

For example if you want to see a trace of the `PerlRequire` and `PerlModule` directives as they are executed, use:

```
PerlSetEnv MOD_PERL_TRACE d
```

Of course you can use the command line environment setting:

```
% setenv MOD_PERL_TRACE all
% httpd -X
```

17.13 gdb says there are no debugging symbols

During *make install* Apache strips all the debugging symbols. To prevent this you should use *--without-execstrip* `./configure` option. So if you configure Apache via mod_perl, you should do:

```
panic% perl Makefile.PL USE_APACI=1 \
  APACI_ARGS='--without-execstrip' [other options]
```

Alternatively you can copy the unstripped binary manually. For example we did:

```
panic# cp apache_1.3.17/src/httpd /home/httpd/httpd_perl/bin/httpd_perl
```

As you know you need an unstripped executable to be able to debug it. While you can compile mod_perl with `-g` (or `PERL_DEBUG=1`), the Apache `install` strips the symbols.

Makefile.tmpl contains a line:

```
IFLAGS_PROGRAM  = -m 755 -s
```

Removing the -s does the trick (If you cannot find it in *Makefile.tmpl* do it directly in *Makefile*). Alternatively you rerun `make` and copy the unstripped httpd binary away.

17.14 Debugging Signal Handlers ($SIG{FOO})

The current Perl implementation does not restore the original Apache C handler when you use the `local $SIG{FOO}` clause. While the save/restore of `$SIG{ALRM}` was fixed in mod_perl 1.19_01 (SVN version), other signals are not yet fixed. The real fix should probably be in Perl itself.

Until recently `local $SIG{ALRM}` restored the `SIGALRM` handler to Perl's handler, not the handler it was in the first place (Apache's `alrm_handler()`). If you build mod_perl with `PERL_TRACE=1` and set the `MOD_PERL_TRACE` environment variable to **g**, you will see this in the *error_log* file:

```
mod_perl: saving SIGALRM (14) handler 0x80b1ff0
mod_perl: restoring SIGALRM (14) handler from: 0x0 to: 0x80b1ff0
```

If nobody has touched `$SIG{ALRM}`, `0x0` will be the same address as the others.

If you work with signal handlers you should take a look at the `Sys::Signal` module, which solves the problem:

`Sys::Signal` - Set signal handlers with restoration of the existing C sighandler. Get it from CPAN.

The usage is simple. If the original code was:

```
# If a timeout happens and C<SIGALRM> is thrown, the alarm() will be
# reset, otherwise C<alarm 0> is reached and timer is reset as well.
eval {
  local $SIG{ALRM} = sub { die "timeout\n" };
  alarm $timeout;
  ... db stuff ...
  alarm 0;
};
die $@ if $@;
```

Now you would write:

```
use Sys::Signal ();
eval {
  my $h = Sys::Signal->set(ALRM => sub { die "timeout\n" });
  alarm $timeout;
  ... do something that may timeout ...
  alarm 0;
};
die $@ if $@;
```

This should be fixed in Perl 5.6.1, so if you use this version of Perl, chances are that you don't need to use `Sys::Signal`.

mod_perl tries to deal only with those signals that cause conflict with Apache's. Currently this is only `SIGALRM`. If there is another one that gives you trouble, you can add it to the list in *perl_config.c* after *"ALRM"*, before *NULL*.

```
static char *sigsave[] = { "ALRM", NULL };
```

17.15 Code Profiling

(META: duplication??? I've started to write about profiling somewhere in this file)

It is possible to profile code run under mod_perl with the `Devel::DProf` module available on CPAN. However, you must have apache version 1.3b3 or higher and the `PerlChildExitHandler` enabled. When the server is started, `Devel::DProf` installs an `END` block (to write the `tmon.out` file) which will be run when the server is shutdown. Here's how to start and stop a server with the profiler enabled:

```
% setenv PERL5OPT -d:DProf
% httpd -X -d ‘pwd‘ &
... make some requests to the server here ...
% kill ‘cat logs/httpd.pid‘
% unsetenv PERL5OPT
% dprofpp
```

See also: `Apache::DProf`

17.16 Devel::Peek

Devel::Peek - A data debugging tool for the XS programmer

Let's see an example of Perl allocating a buffer only once, regardless of my () scoping, although it will realloc() if the size is bigger than `SvLEN`:

```
use Devel::Peek;

for (1..3) {
    foo();
}
```

```
sub foo {
    my $sv;
    Dump $sv;
    $sv = 'x' x 100_000;
    $sv = "";
}
```

The output:

```
SV = NULL(0x0) at 0x8138008
  REFCNT = 1
  FLAGS = (PADBUSY,PADMY)
SV = PV(0x80e5794) at 0x8138008
  REFCNT = 1
  FLAGS = (PADBUSY,PADMY)
  PV = 0x815f808 ""\0
  CUR = 0
  LEN = 100001
SV = PV(0x80e5794) at 0x8138008
  REFCNT = 1
  FLAGS = (PADBUSY,PADMY)
  PV = 0x815f808 ""\0
  CUR = 0
```

We can see that on the second and subsequent calls `$sv` already has previously allocated memory.

So, if you can afford the memory, a larger buffer means fewer `brk()` syscalls. If you watch that example with `strace` you will only see calls to `brk()` the first time through the loop. So this is a case where your module might want to pre-allocate the buffer like this:

```
package Your::Proxy;

my $buffer = ' ' x 100_000;
$buffer = "";
```

Now only the parent has to brk() at server startup, each child already will already have an allocated buffer. Just reset to "" when you are done.

Note: Previously allocating a scalar in this way saves reallocation in v5.005 but may not do so in other versions.

17.17 How can I find out if a mod_perl code has a memory leak

The `Apache::Leak` module (derived from `Devel::Leak`) should help you detecting the leakages in your code. For example:

```
leaktest.pl
-----------
use Apache::Leak;

my $global = "FooAAA";

leak_test {
  $$global = 1;
  ++$global;
};
```

The argument to `leak_test()` is an anonymous sub, so you can just throw it any code you suspect might be leaking. Beware, it will run the code twice! The first time in, new SVs are created, but does not mean you are leaking. The second pass will give better evidence. You do not need to be inside mod_perl to use it. From the command line, the above script outputs:

```
ENTER: 1482 SVs
new c28b8 : new c2918 :
LEAVE: 1484 SVs
ENTER: 1484 SVs
new db690 : new db6a8 :
LEAVE: 1486 SVs
!!! 2 SVs leaked !!!
```

Build a debuggable Perl to see dumps of the SVs. The simple way to have both a normal Perl and debuggable Perl is to follow hints in the SUPPORT doc for building `libperld.a`. When that is built, copy the `perl` from that directory to your Perl bin directory, but name it `dperl`.

Our example's leak explanation: `$$global = 1;` : new global variable FooAAA created with value of `1`, this will not be destroyed until this module is destroyed. Under mod_perl the module doesn't get destroyed until the process quits.

`Apache::Leak` is not very user-friendly, have a look at `B::LexInfo`. It is possible to see something that might appear to be a leak, but is actually just a Perl optimization. e.g. consider this code:

```
sub foo {
  my $string = shift;
}

foo("a string");
```

`B::LexInfo` will show you that Perl does not release the value from $string, unless you undef() it. This is because Perl anticipates the memory will be needed for another string, the next time the subroutine is entered. You'll see similar behaviour for `@array` length, `%hash` keys, and scratch areas of the pad-list for OPs such as `join()`, '`.`', etc.

`Apache::Status` includes a `StatusLexInfo` option which can show you the internals of your code.

17.18 Debugging your code in Single Server Mode

Running in httpd -X mode is good only for testing during the development phase.

You want to test that your application correctly handles global variables (if you have any - the less you have of them the better of course - but sometimes you just can't do without them). It's hard to test with multiple servers serving your cgi since each child has a different value for its global variables. Imagine that you have a `random()` sub that returns a random number and you have the following script.

```
use vars qw($num);
$num ||= random();
print ++$num;
```

This script initializes the variable `$num` with a random value, then increments it on each request and prints it out. Running this script in a multiple server environments will result in something like `1`, `9`, `4`, `19` (a different number each time you hit the browser's reload button) since each time your script will be served by a different child. (On some operating systems, e.g. AIX, the parent httpd process will assign all of the requests to the same child process if all of the children are idle). But if you run in `httpd -X` single server mode you will get `2`, `3`, `4`, `5`... (assuming that `random()` returned `1` at the first call)

But do not get too obsessive with this mode, since working in single server mode sometimes hides problems that show up when you switch to normal (multi-server) mode.

Consider an application that allows you to change the configuration at run time. Let's say the script produces a form to change the background color of the page. It's not good design, but for the sake of demonstrating the potential problem we will assume that our script doesn't write the changed background color to the disk, but simply changes it in memory, like this:

```
use vars qw($bgcolor);
  # assign default value at first invocation
$bgcolor ||= "white";
  # modify the color if requested to
$bgcolor = $q->param('bgcolor') || $bgcolor;
```

So you have typed in a new color, and in response, your script prints back the html with a new color - you think that's it! It was so simple. If you keep running in single server mode you will never notice that you have a problem...

If you run the same code in normal server mode, after you submit the color change you will get the result as expected, but when you call the same URL again (not reload!) the chances are that you will get back the original default color (white in our case), since only the child which processed the color change request knows about the global variable change. Just remember that children can't share information, other than that which they inherited from their parent on their birth. Of course you could use a hidden variable for the color to be remembered, or store it on the server side (database, shared memory, etc).

If you use the Netscape client while your server is running in single-process mode, if the output returns HTML with `<IMG>` tags, then the loading of the images will take a long time, since Netscape's `KeepAlive` feature gets in the way. Netscape tries to open multiple connections and keep them open. Because there is only one server process listening, each connection has to time-out before the next

succeeds. Turn off `KeepAlive` in *httpd.conf* to avoid this effect. Alternatively (assuming you use the image size parameters, so that Netscape will be able to render the rest of the page) you can press **STOP** after a few seconds.

In addition you should be aware that when running with `-X` you will not see the status messages that the parent server normally writes to the error_log. ("server started", "server stopped", etc.). Since `httpd -X` causes the server to handle all requests itself, without forking any children, there is no controlling parent to write the status messages.

17.19 Apache::DumpHeaders - Watch HTTP Transaction Via Headers

This module is used to watch an HTTP transaction, looking at client and servers headers.

With `Apache::ProxyPassThru` configured, you are able to watch your browser talk to any server besides the one with this module living inside.

`Apache::DumpHeaders` has the ability to filter on IP addresses, has an interface for other modules to decide if the headers should be dumped or not and a function to only dump *n%* of the transactions.

For more information read the module's manpage.

Download the module from CPAN.

17.20 Apache::DebugInfo - Log Various Bits Of Per-Request Data

`Apache::DebugInfo` offers the ability to monitor various bits of per-request data. Its functionality is similar to Apache::DumpHeaders while offering several additional features, including the ability to:

- **- separate inbound from outbound HTTP headers**
- **- view the contents of $r->notes and $r->pnotes**
- **- view any of these at the various points in the request cycle**
- **- add output for any request phase from a single entry point**
- **- use as a PerlInitHandler or with direct method calls**
- **- use partial IP addresses for filtering by IP**
- **- offer a subclassable interface**

See the module's manpage for more details.

17.21 Maintainers

Maintainer is the person(s) you should contact with updates, corrections and patches.

- Stas Bekman [http://stason.org/]

17.22 Authors

- Stas Bekman [http://stason.org/]

Only the major authors are listed above. For contributors see the Changes file.

18 Getting Help

18.1 Description

If your question isn't answered by reading this guide, check this section for information on how to get help on mod_perl, Apache, or other topics discussed here.

18.2 READ ME FIRST

If, after reading this guide, the general docs and the other documents listed on this site, you still don't have the answers, please ask for help on the mod_perl users mailing list. But please, first try to browse the mailing list archive. Most of the time you will find the answers to your questions by searching the archive, since it is very likely that someone else has already encountered the same problem and found a solution for it. If you ignore this advice, you should not be surprised if your question is left unanswered -- it bores people to answer the same question again and again. This does not mean that you should avoid asking questions, but you should not abuse the available help and you should *RTFM* before you call for *HELP*. (Remember the fable of the shepherd boy and the wolves).

Another possibilty is to look in the general Getting Help section of this site for a commercial training or consulting company.

18.2.1 Please Ask Only Questions Related to mod_perl

If you have general Apache questions, please refer to: http://httpd.apache.org/lists.html.

If you have general Perl questions, please refer to: http://lists.perl.org/.

For other remotely related to mod_perl questions see the references to other documentation.

Finally, if you are posting to the list for the first time, please refer to the mod_perl mailing lists' Guidelines.

18.3 How to Report Problems

Make sure to include a good subject like explaining the problem in a few words. Also please mention that this a problem with mod_perl 1.0 (since now we have mod_perl 2.0 too). Here is an example of a good subject:

```
Subject: [mp1] response handler randomly segfaults
```

The most important thing to understand is that you should try hard to provide **all** the information that may assist to understand and reproduce the problem. When you prepare a bug report, put yourself in the position of a person who is going to try to help you, realizing that a guess-work on behalf of that helpful person, more often doesn't work than it does. Unfortunately most people don't realize that, and it takes several emails to squeeze the needed details from the person reporting the bug, a process which may drag for days.

Always send the following details:

- Anything in the *error_log* file that looks suspicious and possibly related to the problem.
- Output of `perl -V`
- Version of mod_perl (hint: it's logged into the error_log file when the server has just started)
- Version of apache (hint: it's logged into the error_log file when the server has just started)
- Options given to mod_perl's Makefile.PL while building it. If you are using a pre-compiled binary (e.g., rpm), find the source package that was used to build this binary and retrieve this information from there.
- Server configuration details (that's the relevant parts of your *httpd.conf*, usually just the relevant mod_perl configuration sections).
- Relevant sections of your `ErrorLog` (make test's is: t/logs/error_log)
- If some other code doesn't work, minimize that code to a minimal size while it reproduces the problem and attach it to the report.
- If you build from source, make sure that `make test` passes 100%.

  If `make test` fails, run the failing tests separately, using the verbose mode and attach the output of the run and the relevant sections of your `ErrorLog` to the report.

If this is a script which doesn't use mod_perl api, try to test under mod_cgi if applicable

You should try to isolate the problem and send the smallest possible code snippet, that reproduces the problem.

Remember, that if we cannot reproduce the problem, we might not be able to solve it.

To further increase the chances that bugs your code exposes will be investigated, try using `Apache-Test` to create a self-contained environment that core developers can use to easily reproduce your bug. To get you started, an `Apache-Test` bug skeleton has been created:

http://perl.apache.org/~geoff/bug-reporting-skeleton-mp1.tar.gz

Detailed instructions are contained within the `README`.

- **Getting the Backtrace From Core Dumps**

  If you get a *core* file dump (*Segmentation fault*), please send a backtrace if possible. Before you try to produce it, re-build mod_perl with:

```
panic% perl Makefile.PL PERL_DEBUG=1
```

which will:

- **add `-g` to `EXTRA_CFLAGS`**
- **turn on `PERL_TRACE`**
- **set `PERL_DESTRUCT_LEVEL=2` (additional checks during Perl cleanup)**
- **link against *libperld* if it exists**

Here is a summary of how to get a backtrace:

```
% cd mod_perl-x.xx
% touch t/conf/srm.conf
% gdb ../apache_x.xx/src/httpd
(gdb) run -X -f ‘pwd‘/t/conf/httpd.conf -d ‘pwd‘/t
[now make request that causes core dump]
(gdb) bt
```

So you go to the mod_perl source directory, create an empty *srm.conf* file, and start gdb with a path to the httpd binary, which is at least located in the Apache source tree after you built it. (Of course replace *x* with real version numbers). Next step is to start the httpd from within gdb and issue a request, which causes a core dump. when the code has died with SEGV sygnal, run *bt* to get the back-trace.

Alternatively you can also attach to an already running process like so:

```
% gdb httpd <process id number>
```

If the dump is happening in *libperl* you have to rebuild Perl with `-DDEBUGGING` enabled. A quick way to this is to go to your Perl source tree and run these commands:

```
% rm *.[oa]
% make LIBPERL=libperld.a
% cp libperld.a $Config{archlibexp}/CORE
```

where `$Config{archlibexp}` is:

```
% perl -V:archlibexp
```

If the trace includes Apache calls and you see no arguments, you need to rebuild Apache with *--without-execstrip*. If building a static mod_perl, you need to rebuild it with:

```
% perl Makefile.PL ... APACI_ARGS='--without-execstrip'
```

- **Spinning Processes**

  The gdb attaching to the live process approach is helpful when debugging a *spinning* process. You can also get a Perl stacktrace of a *spinning* process by install a `$SIG{USR1}` handler in your code:

```
use Carp ();
$SIG{USR1} = \&Carp::confess;
```

While the process is spinning, send it a *USR1* signal:

```
% kill -USR1 <process id number>
```

and the Perl stack trace will be printed.

alternatively you can use gdb to find which Perl code is causing the spin:

```
% gdb httpd <pid of spinning process>
(gdb) where
(gdb) source mod_perl-x.xx/.gdbinit
(gdb) curinfo
```

After loading the special macros file (*.gdbinit*) you can use the *curinfo* gdb macro to figure out the file and line number the code stuck in.

Finally send all these details to the modperl mailing list.

18.4 Help on Related Topics

When developing with mod_perl, you often find yourself having questions regarding other projects and topics like Apache, Perl, SQL, etc. This document will help you find the right resource where you can find the answers to your questions.

18.5 Maintainers

Maintainer is the person(s) you should contact with updates, corrections and patches.

- Stas Bekman [http://stason.org/]

18.6 Authors

- Stas Bekman [http://stason.org/]

Only the major authors are listed above. For contributors see the Changes file.

Table of Contents:

mod_perl 1.0 User Guide 1
Getting Your Feet Wet 4
1 Getting Your Feet Wet 4
1.1 Description 5
1.2 Installing mod_perl in Three Steps 5
1.3 Installing mod_perl for Unix Platforms 6
1.3.1 Obtaining and Unpacking the Source Code 6
1.3.2 Building mod_perl 7
1.3.3 Installing mod_perl 8
1.3.4 Configuring and Starting the mod_perl Server 9
1.4 Installing mod_perl for Windows 10
1.5 Preparing the Scripts Directory 11
1.6 A Sample Apache::Registry Script 12
1.6.1 Porting Existing CGI Scripts to run under mod_perl 14
1.7 A Simple Apache Perl Content Handler 14
1.8 Is This All we Need to Know About mod_perl? 16
1.9 References 17
Introduction and Incentives 19
2 Introduction and Incentives 19
2.1 Description 20
2.2 What is mod_perl? 20
2.2.1 mod_cgi 20
2.2.2 C API 21
2.2.3 Perl API 22
2.2.4 Apache::Registry 23
2.2.5 Apache::PerlRun 25
2.3 What you will learn 25
2.4 Maintainers 26
2.5 Authors 26
mod_perl Installation 27
3 mod_perl Installation 27
3.1 Description 28
3.2 A Summary of a Basic mod_perl Installation 28
3.3 The Gory Details 29
3.3.1 Source Configuration (perl Makefile.PL ...) 29
3.3.1.1 Configuration parameters 31
3.3.1.1.1 APACHE_SRC 31
3.3.1.1.2 DO_HTTPD, NO_HTTPD, PREP_HTTPD 32
3.3.1.1.3 Callback Hooks 32
3.3.1.1.4 EVERYTHING 33
3.3.1.1.5 PERL_TRACE 33
3.3.1.1.6 APACHE_HEADER_INSTALL 33
3.3.1.1.7 PERL_STATIC_EXTS 34
3.3.1.1.8 APACI_ARGS 34

3.3.1.1.9 APACHE_PREFIX 34
3.3.1.2 Environment Variables 35
3.3.1.2.1 APACHE_USER and APACHE_GROUP 35
3.3.1.3 Reusing Configuration Parameters 35
3.3.1.4 Discovering Whether Some Option Was Configured 36
3.3.1.5 Using an Alternative Configuration File 37
3.3.1.6 perl Makefile.PL Troubleshooting 37
3.3.1.6.1 "A test compilation with your Makefile configuration failed..." 37
3.3.1.6.2 Missing or Misconfigured libgdbm.so 38
3.3.1.6.3 About gdbm, db and ndbm libraries 39
3.3.1.6.4 Undefined reference to ‘PL_perl_destruct_level’ 39
3.3.1.6.5 Further notes on libperl.(a|so) 39
3.3.2 mod_perl Building (make) 40
3.3.2.1 make Troubleshooting 40
3.3.2.1.1 Undefined reference to ’Perl_newAV’ 40
3.3.2.1.2 Unrecognized format specifier for... 40
3.3.3 Built Server Testing (make test) 41
3.3.3.1 Manual Testing 41
3.3.3.2 make test Troubleshooting 42
3.3.3.2.1 make test fails 42
3.3.3.2.2 mod_perl.c is incompatible with this version of Apache 42
3.3.3.2.3 make test.....skipping test on this platform 42
3.3.3.2.4 make test Fails Due to Misconfigured localhost Entry 43
3.3.4 Installation (make install) 43
3.3.5 Building Apache and mod_perl by Hand 43
3.4 Installation Scenarios for Standalone mod_perl 44
3.4.1 The All-In-One Way 45
3.4.2 The Flexible Way 45
3.4.3 When DSO can be Used 46
3.4.4 Build mod_perl as a DSO inside the Apache Source Tree via APACI 47
3.4.4.1 libperl.so and libperl.a 47
3.4.5 Build mod_perl as a DSO outside the Apache Source Tree via APXS 48
3.5 Installation Scenarios for mod_perl and Other Components 48
3.5.1 mod_perl and mod_ssl (+openssl) 49
3.5.2 mod_perl and mod_ssl Rolled from RPMs 51
3.5.3 mod_perl and apache-ssl (+openssl) 53
3.5.4 mod_perl and Stronghold 54
3.5.4.1 Note For Solaris 2.5 users 55
3.5.5 mod_perl and mod_php 55
3.6 mod_perl Installation with the CPAN.pm Interactive Shell 56
3.7 Installing on multiple machines 58
3.8 using RPM and other packages to install mod_perl 59
3.8.1 A word on mod_perl RPM packages 59
3.8.2 Getting Started 59
3.8.3 Compiling RPM source files 60
3.8.4 Mix and Match RPM and source 60
3.8.5 Installing a single apache+mod_perl RPM 60

3.8.6 Compiling libapreq (Apache::Request) with the RH 6.0 mod_perl RPM 61
3.8.7 Installing separate Apache and mod_perl RPMs 63
3.8.8 Testing the mod_perl API 63
3.9 Installation Without Superuser Privileges 64
3.9.1 Installing Perl Modules into a Directory of Choice 64
3.9.2 Making Your Scripts Find the Locally Installed Modules 65
3.9.3 The CPAN.pm Shell and Locally Installed Modules 68
3.9.4 Making a Local Apache Installation 70
3.9.5 Manual Local mod_perl Enabled Apache Installation 71
3.9.5.1 Resource Usage 72
3.9.6 Local mod_perl Enabled Apache Installation with CPAN.pm 73
3.10 Automating installation 74
3.11 How can I tell whether mod_perl is running? 75
3.11.1 Checking the error_log 75
3.11.2 Testing by viewing /perl-status 75
3.11.3 Testing via telnet 75
3.11.4 Testing via a CGI script 76
3.11.5 Testing via lwp-request 77
3.12 General Notes 77
3.12.1 Is it possible to run mod_perl enabled Apache as suExec? 77
3.12.2 Should I Rebuild mod_perl if I have Upgraded Perl? 78
3.12.3 Perl installation requirements 78
3.12.4 mod_auth_dbm nuances 78
3.12.5 Stripping Apache to make it almost a Perl-server 79
3.12.6 Saving the config.status Files with mod_perl, php, ssl and Other Components . . 79
3.12.7 What Compiler Should Be Used to Build mod_perl? 79
3.12.8 Unescaping *error_log* 80
3.13 OS Related Notes 80
3.14 Pros and Cons of Building mod_perl as DSO 80
3.15 Maintainers 81
3.16 Authors 82
mod_perl Configuration 83
4 mod_perl Configuration 83
4.1 Description 84
4.2 Server Configuration 84
4.3 Apache Configuration 84
4.3.1 Configuration Directives 85
4.3.2 .htaccess files 85
4.3.3 <Directory>, <Location> and <Files> Sections 85
4.3.4 How Directory, Location and Files Sections are Merged 88
4.3.5 Sub-Grouping of <Location>, <Directory> and <Files> Sections 89
4.3.6 Options Directive 89
4.4 mod_perl Configuration 90
4.4.1 Alias Configurations 90
4.4.1.1 Running CGI, PerlRun, and Registry Scripts Located in the Same Directory . . 91
4.4.2 <Location> Configuration 92
4.4.3 Overriding <Location> Setting in "Sub-Location" 94

4.4.4 PerlModule and PerlRequire Directives 94
4.4.5 Perl*Handlers 94
4.4.6 The handler subroutine 96
4.4.7 Stacked Handlers 97
4.4.8 Perl Method Handlers 100
4.4.9 PerlFreshRestart 102
4.4.10 PerlSetEnv and PerlPassEnv 102
4.4.11 PerlSetVar and PerlAddVar 103
4.4.12 PerlSetupEnv 105
4.4.13 PerlWarn and PerlTaintCheck 105
4.4.14 MinSpareServers MaxSpareServers StartServers MaxClients MaxRequestsPerChild . 106
4.5 The Startup File 106
4.5.1 The Sample Startup File 106
4.5.2 What Modules You Should Add to the Startup File and Why 109
4.5.3 The Confusion with use() in the Server Startup File 109
4.6 Apache Configuration in Perl 110
4.6.1 Usage 110
4.6.2 Enabling 112
4.6.3 Caveats 113
4.6.4 Verifying 113
4.6.5 Strict <Perl> Sections 114
4.6.6 Debugging 114
4.6.7 Perl Section Tricks 115
4.6.8 References 115
4.7 Validating the Configuration Syntax 115
4.8 Enabling Remote Server Configuration Reports 115
4.9 Publishing Port Numbers other than 80 116
4.10 Configuring Apache + mod_perl with mod_macro 116
4.11 General Pitfalls 118
4.11.1 My CGI/Perl Code Gets Returned as Plain Text Instead of Being Executed by the Webserver 118
4.11.2 My Script Works under mod_cgi, but when Called via mod_perl I Get a 'Save-As' Prompt 119
4.11.3 Is There a Way to Provide a Different startup.pl File for Each Individual Virtual Host 119
4.11.4 Is There a Way to Modify @INC on a Per-Virtual-Host or Per-Location Basis. . . 119
4.11.5 A Script From One Virtual Host Calls a Script with the Same Path From the Other Virtual Host 119
4.11.6 the Server no Longer Retrieves the DirectoryIndex Files for a Directory . . . 119
4.11.7 Do Perl* Directives Affect Code Running under mod_cgi? 121
4.12 Configuration Security Concerns 121
4.12.1 Choosing User and Group 121
4.12.2 Taint Checking 121
4.12.3 Exposing Information About the Server's Component 122
4.13 Apache Restarts Twice On Start 122
4.14 Knowing the proxy_pass'ed Connection Type 123
4.15 Adding Custom Configuration Directives 123
4.16 Maintainers 125
4.17 Authors 125

CGI to mod_perl Porting. mod_perl Coding guidelines. 126
5 CGI to mod_perl Porting. mod_perl Coding guidelines. 126
5.1 Description 127
5.2 Before you start to code 127
5.3 Exposing Apache::Registry secrets 128
5.3.1 The First Mystery 129
5.3.2 The Second Mystery 132
5.4 Sometimes it Works, Sometimes it Doesn't 133
5.4.1 An Easy Break-in 133
5.4.2 Thinking mod_cgi 134
5.4.3 Regular Expression Memory 135
5.5 Script's name space 135
5.6 @INC and mod_perl 135
5.7 Reloading Modules and Required Files 136
5.7.1 Restarting the server 136
5.7.2 Using Apache::StatINC for the Development Process 137
5.7.3 Using Apache::Reload 138
5.7.3.1 Register Modules Implicitly 138
5.7.3.2 Register Modules Explicitly 138
5.7.3.3 Special "Touch" File 139
5.7.3.4 Caveats 139
5.7.3.5 Availability 139
5.7.4 Configuration Files: Writing, Dynamically Updating and Reloading 139
5.7.4.1 Writing Configuration Files 140
5.7.4.2 Reloading Configuration Files 146
5.7.4.3 Dynamically updating configuration files 148
5.7.5 Reloading handlers 154
5.8 Name collisions with Modules and libs 154
5.9 More package name related issues 160
5.10 __END__ and __DATA__ tokens 160
5.11 Output from system calls 161
5.12 Using format() and write() 161
5.13 Terminating requests and processes, the exit() and child_terminate() functions . . . 162
5.14 die() and mod_perl 163
5.15 Return Codes 164
5.16 Testing the Code from the Shell 164
5.17 I/O is different 164
5.18 STDIN, STDOUT and STDERR streams 164
5.19 Redirecting STDOUT into a Scalar 165
5.20 Apache::print() and CORE::print() 165
5.21 Global Variables Persistence 165
5.22 Generating correct HTTP Headers 166
5.23 NPH (Non Parsed Headers) scripts 172
5.24 BEGIN blocks 172
5.25 END blocks 173
5.26 CHECK And INIT Blocks 173
5.27 Command Line Switches (-w, -T, etc) 174

5.27.1 Warnings 174
5.27.2 Taint Mode 175
5.27.3 Other switches 176
5.28 The strict pragma 176
5.29 Passing ENV variables to CGI 176
5.30 -M and other time() file tests under mod_perl 177
5.31 Apache and syslog 178
5.32 File tests operators 178
5.33 Filehandlers and locks leakages 178
5.34 Code has been changed, but it seems the script is running the old code 179
5.35 The Script Is Too Dirty, But It Does The Job And I Cannot Afford To Rewrite It. . . . 179
5.36 Apache::PerlRun--a closer look 180
5.37 Sharing variables between processes 181
5.38 Preventing Apache::Constants Stringification 181
5.39 Transitioning from Apache::Registry to Apache handlers 182
5.39.1 Starting with mod_cgi Compatible Script 183
5.39.2 Converting into Perl Content Handler 185
5.39.3 Converting to use Apache Perl Modules 187
5.39.4 Conclusion 191
5.40 Maintainers 191
5.41 Authors 191
How to use mod_perl's Method Handlers 192
6 How to use mod_perl's Method Handlers 192
6.1 Description 193
6.2 Synopsis 193
6.3 Why? 194
6.4 Simple example 194
6.5 A little more advanced 194
6.6 Traps 195
6.7 Author 196
6.8 See Also 196
6.9 Maintainers 196
6.10 Authors 196
mod_perl and Relational Databases 197
7 mod_perl and Relational Databases 197
7.1 Description 198
7.2 Why Relational (SQL) Databases 198
7.3 Apache::DBI - Initiate a persistent database connection 198
7.3.1 Introduction 198
7.3.2 When should this module be used and when shouldn't it be used? 199
7.3.3 Configuration 199
7.3.4 Preopening DBI connections 199
7.3.5 Debugging Apache::DBI 200
7.3.6 Database Locking Risks 200
7.3.7 Troubleshooting 201
7.3.7.1 The Morning Bug 201
7.3.7.2 Opening Connections With Different Parameters 201

7.3.7.3 Cannot find the DBI handler 203
7.3.7.4 Apache:DBI does not work 203
7.3.7.5 Skipping connection cache during server startup 203
7.3.7.6 Debugging code which deploys DBI 204
7.4 mysql_use_result vs. mysql_store_result. 204
7.5 Transactions Not Committed with MySQL InnoDB Tables 205
7.6 Optimize: Run Two SQL Engine Servers 205
7.7 Some useful code snippets to be used with relational Databases 206
7.7.1 Turning SQL query writing into a short and simple task 206
7.7.2 The My::DB module 206
7.7.3 My::DB Module's Usage Examples 219
7.8 Maintainers 222
7.9 Authors 222
mod_perl and dbm files 223
8 mod_perl and dbm files 223
8.1 Description 224
8.2 Where and Why to use dbm files 224
8.3 mod_perl and dbm 226
8.4 Locking dbm Handlers and Write Lock Starvation Hazards 226
8.5 Flawed Locking Methods Which Must Not Be Used 227
8.6 Locking Wrappers Overview 229
8.7 Tie::DB_Lock 229
8.8 DB_File::Lock2 230
8.9 Maintainers 233
8.10 Authors 233
Protecting Your Site 234
9 Protecting Your Site 234
9.1 Description 235
9.2 The Importance of Your site's Security 235
9.3 Illustrated Security Scenarios 236
9.3.1 Non authenticated access for internal IPs, Authenticated for external IPs 236
9.4 Authentication code snippets 238
9.4.1 Forcing re-authentication 238
9.4.2 OK, AUTH_REQUIRED and FORBIDDEN in Authentication handlers 238
9.5 Apache::Auth* modules 239
9.6 Maintainers 240
9.7 Authors 240
Code Snippets 241
10 Code Snippets 241
10.1 Description 242
10.2 File Upload with Apache::Request 242
10.3 Redirecting Errors to the Client Instead of error_log 243
10.4 Reusing Data from POST request 247
10.5 Redirecting POST Requests 248
10.6 Redirecting While Maintaining Environment Variables 248
10.7 Terminating a Child Process on Request Completion 248
10.8 Setting Content-type and Content-encoding headers in non-OK responses 248

10.9 More on Relative Paths 249
10.10 Watching the error_log File Without Telneting to the Server 249
10.11 Accessing Variables from the Caller's Package 251
10.12 Handling Cookies 251
10.13 Sending Multiple Cookies with the Perl API 252
10.14 Sending Cookies in REDIRECT Response 252
10.15 Apache::Cookie example: Login Pages by Setting Cookies and Refreshing 252
10.15.1 Logic 252
10.15.2 Example Situation 253
10.15.2.1 Code 253
10.16 Passing and Preserving Custom Data Structures Between Handlers 254
10.17 Passing Notes Between mod_perl and other (non-Perl) Apache Modules 254
10.18 Passing Environment Variables Between Handlers 255
10.19 Verifying Whether A Request Was Received Over An SSL Connection 256
10.20 CGI::params in the mod_perl-ish Way 256
10.21 Subclassing Apache::Request 257
10.22 Sending Email from mod_perl 257
10.23 A Simple Handler To Print The Environment Variables 259
10.24 mod_rewrite in Perl 259
10.25 URI Rewrite in PerlTransHandler 260
10.26 Setting PerlHandler Based on MIME Type 261
10.27 SSI and Embperl -- Doing Both 261
10.28 Getting the Front-end Server's Name in the Back-end Server 262
10.29 Authentication Snippets 263
10.30 Emulating the Authentication Mechanism 263
10.31 An example of using Apache::Session::DBI with cookies 263
10.32 Detecting a Client Abort 264
10.33 Using DESTROY to Finalize Output 265
10.34 Passing Arguments to a SSI script 267
10.35 Setting Environment Variables For Scripts Called From CGI. 267
10.36 Mysql Backup and Restore Scripts 268
10.37 Maintainers 273
10.38 Authors 273
Apache::* modules 274
11 Apache::* modules 274
11.1 Description 275
11.2 Apache::Session - Maintain session state across HTTP requests 275
11.3 Apache::DBI - Initiate a persistent database connection 277
11.4 Apache::Watchdog::RunAway - Hanging Processes Monitor and Terminator 277
11.5 Apache::VMonitor -- Visual System and Apache Server Monitor 277
11.5.0.1 Configuration 277
11.6 Apache::GTopLimit - Limit Apache httpd processes 278
11.7 Apache::Request (libapreq) - Generic Apache Request Library 278
11.8 Apache::RequestNotes - Allow Easy, Consistent Access to Cookie and Form Data Across Each Request Phase 279
11.9 Apache::PerlRun - Run unaltered CGI scripts under mod_perl 279
11.10 Apache::RegistryNG -- Apache::Registry New Generation 279

11.11 Apache::RegistryBB -- Apache::Registry Bare Bones 280
11.12 Apache::OutputChain -- Chain Stacked Perl Handlers 280
11.13 Apache::Filter - Alter the output of previous handlers 281
11.14 Apache::GzipChain - compress HTML (or anything) in the OutputChain 282
11.15 Apache::Gzip - Auto-compress web files with Gzip 282
11.16 Apache::PerlVINC - Allows Module Versioning in Location blocks and Virtual Hosts . 283
11.17 Apache::LogSTDERR 285
11.18 Apache::RedirectLogFix 285
11.19 Apache::SubProcess 286
11.20 Module::Use - Log and Load used Perl modules 288
11.21 Apache::ConfigFile - Parse an Apache style httpd.conf config file 288
11.22 Apache::Admin::Config - Object oriented access to Apache style config files . . . 288
11.23 Maintainers 288
11.24 Authors 288
Choosing the Right Strategy 290
12 Choosing the Right Strategy 290
12.1 Description 291
12.2 Do it like I do it!? 291
12.3 mod_perl Deployment Overview 291
12.4 Alternative architectures for running one and two servers 292
12.4.1 Standalone mod_perl Enabled Apache Server 293
12.4.2 One Plain Apache and One mod_perl-enabled Apache Servers 294
12.4.3 One light non-Apache and One mod_perl enabled Apache Servers 295
12.5 Adding a Proxy Server in http Accelerator Mode 296
12.6 Implementations of Proxy Servers 299
12.6.1 The Squid Server 299
12.6.2 Apache's mod_proxy 300
12.6.3 Closing Lingering Connections with Lingerd 301
12.7 When One Machine is not Enough for RDBMS Database and mod_perl 301
12.7.1 Servers' Requirements 302
12.7.2 The Problem 303
12.7.3 The Solution 304
12.7.3.1 Pros 304
12.7.3.2 Cons 304
12.7.4 Three Machines Model 305
12.8 Running More than One mod_perl Server on the Same Machine. 305
12.9 SSL functionality and a mod_perl Server 307
12.10 Maintainers 308
12.11 Authors 309
Real World Scenarios 310
13 Real World Scenarios 310
13.1 Description 311
13.2 Assumptions 311
13.3 Standalone mod_perl Enabled Apache Server 311
13.3.1 Installation in 10 lines 311
13.3.2 Installation in 10 paragraphs 311
13.3.3 Configuration 313

13.4 One Plain and One mod_perl enabled Apache Servers 315
13.4.1 Configuration and Compilation of the Sources. 316
13.4.1.1 Building the httpd_docs Server 316
13.4.1.2 Building the httpd_perl Server 317
13.4.2 Configuration of the servers 319
13.4.2.1 Basic httpd_docs Server Configuration 319
13.4.2.2 Basic httpd_perl Server Configuration 319
13.5 Running Two webservers and Squid in httpd Accelerator Mode 321
13.6 Running One Webserver and Squid in httpd Accelerator Mode 327
13.7 mod_proxy 328
13.7.1 Concepts and Configuration Directives 328
13.7.1.1 ProxyPass 328
13.7.1.2 ProxyPassReverse 329
13.7.1.3 Security Issues 329
13.7.2 Buffering Feature 330
13.7.3 Setting the Buffering Limits on Various OSs 331
13.7.3.1 IOBUFSIZE Source Code Definition 331
13.7.3.2 ProxyReceiveBufferSize Configuration Directive 332
13.7.3.3 Hacking the Code 333
13.7.4 Caching Feature 333
13.7.5 Build Process 333
13.8 Front-end Back-end Proxying with Virtual Hosts 333
13.9 Getting the Remote Server IP in the Back-end server in the Proxy Setup 335
13.9.1 Build 335
13.9.2 Usage 336
13.9.3 Security 336
13.9.4 Caveats 337
13.9.5 mod_proxy_add_forward Module's Order Precedence 337
13.10 HTTP Authentication With Two Servers Plus a Proxy 338
13.11 mod_rewrite Examples 338
13.11.1 Rewriting Requests Based on File Extension 338
13.11.2 Internet Exporer 5 favicon.ico 404 339
13.11.3 Hiding Extensions for Dynamic Pages 339
13.11.4 Serving Static Content Locally and Rewriting Everything Else 339
13.11.5 Upgrading mod_perl Heavy Application Instances 340
13.11.6 Blocking IP Addresses 340
13.12 Caching in mod_proxy 341
13.13 Maintainers 341
13.14 Authors 341
Performance Tuning 342
14 Performance Tuning 342
14.1 Description 343
14.2 The Big Picture 343
14.3 System Analysis 344
14.3.1 Software Requirements 344
14.3.2 Hardware Requirements 344
14.4 Essential Tools 345

14.4.1 Benchmarking Applications 345
14.4.1.1 Benchmarking Perl Code 345
14.4.1.2 Benchmarking a Graphic Hits Counter with Persistent DB Connections . . . 346
14.4.1.3 Benchmarking Response Times 346
14.4.1.3.1 ApacheBench 347
14.4.1.3.2 httperf 347
14.4.1.3.3 http_load 348
14.4.1.3.4 the crashme Script 349
14.4.1.4 Benchmarking PerlHandlers 352
14.4.1.5 Other Benchmarking Tools 353
14.4.2 Code Profiling Techniques 354
14.4.3 Measuring the Memory of the Process 357
14.4.4 Measuring the Memory Usage of Subroutines 360
14.5 Know Your Operating System 362
14.5.1 Sharing Memory 363
14.5.1.1 How Shared Is My Memory? 363
14.5.1.2 Calculating Real Memory Usage 364
14.5.1.3 Are My Variables Shared? 365
14.5.1.4 Preloading Perl Modules at Server Startup 369
14.5.1.5 Preloading Registry Scripts at Server Startup 371
14.5.1.6 Modules Initializing at Server Startup 373
14.5.1.6.1 Initializing DBI.pm 373
14.5.1.6.2 Initializing CGI.pm 376
14.5.2 Increasing Shared Memory With mergemem 379
14.5.3 Forking and Executing Subprocesses from mod_perl 379
14.5.3.1 Forking a New Process 380
14.5.3.2 Freeing the Parent Process 381
14.5.3.3 Detaching the Forked Process 382
14.5.3.4 Avoiding Zombie Processes 382
14.5.3.5 A Complete Fork Example 384
14.5.3.6 Starting a Long Running External Program 386
14.5.3.7 Starting a Short Running External Program 388
14.5.3.8 Executing system() or exec() in the Right Way 389
14.5.4 OS Specific Parameters for Proxying 389
14.6 Performance Tuning by Tweaking Apache Configuration 389
14.6.1 Configuration Tuning with ApacheBench 390
14.6.2 Choosing MaxClients 395
14.6.3 Choosing MaxRequestsPerChild 397
14.6.4 Choosing MinSpareServers, MaxSpareServers and StartServers 398
14.6.5 Summary of Benchmarking to tune all 5 parameters 398
14.6.6 KeepAlive 400
14.6.7 PerlSetupEnv Off 401
14.6.8 Reducing the Number of stat() Calls Made by Apache 402
14.7 TMTOWTDI: Convenience and Habit vs. Performance 406
14.7.1 Apache::Registry PerlHandler vs. Custom PerlHandler 407
14.7.2 "Bloatware" modules 410
14.7.3 Apache::args vs. Apache::Request::param vs. CGI::param 412

14.7.4 Using $|=1 Under mod_perl and Better print() Techniques. 415
14.7.5 Global vs. Fully Qualified Variables 417
14.7.6 Object Methods Calls vs. Function Calls 418
14.7.6.1 The Overhead with Light Subroutines 418
14.7.6.2 The Overhead with Heavy Subroutines 419
14.7.6.3 Are All Methods Slower than Functions? 420
14.7.7 Imported Symbols and Memory Usage 421
14.7.8 Interpolation, Concatenation or List 422
14.7.9 Using Perl stat() Call's Cached Results 424
14.7.10 Optimizing Code 424
14.8 Apache::Registry and Derivatives Specific Notes 425
14.8.1 Be Careful with Symbolic Links 425
14.9 Improving Performance by Prevention 426
14.9.1 Memory leakage 426
14.9.1.1 Reading In A Whole File 426
14.9.1.2 Copying Variables Between Functions 426
14.9.1.3 Work With Databases 428
14.9.2 Avoid the Evil Troika 431
14.9.3 Preventing Your Processes from Growing 431
14.9.3.1 Defining the Minimum Shared Memory Size Threshold 432
14.9.3.2 Potential Drawbacks of Memory Sharing Restriction 433
14.9.3.3 Defining the Maximum Memory Size Threshold 434
14.9.3.4 Defining the Maximum Unshared Memory Size Threshold 435
14.9.4 Limiting Other Resources Used by Apache Child Processes 435
14.9.4.1 OS Specific notes 436
14.9.5 Limiting the Number of Processes Serving the Same Resource 437
14.9.6 Limiting the Request Rate Speed (Robot Blocking) 438
14.10 Perl Modules for Performance Improvement 438
14.10.1 Sending Plain HTML as Compressed Output 438
14.10.2 Caching Components with HTML::Mason 438
14.11 Efficient Work with Databases under mod_perl 438
14.11.1 Persistent DB Connections 439
14.11.1.1 Preopening Connections at the Child Process' Fork Time 440
14.11.1.2 Caching prepare() Statements 440
14.11.2 mod_perl Database Performance Improving 440
14.11.2.1 Analysis of the Problem 441
14.11.2.2 Optimizing Database Connections 441
14.11.2.3 Utilizing the Database Server's Cache 443
14.11.2.4 Eliminating SQL Statement Parsing 444
14.11.2.5 Conclusion 446
14.12 Using 3rd Party Applications 446
14.12.1 Proxying the mod_perl Server 447
14.13 Upload and Download of Big Files 447
14.14 Apache/mod_perl Build Options 447
14.14.1 mod_perl Process Size as a Function of Compiled in C Modules and mod_perl Features 447
14.15 Perl Build Options 449
14.15.1 -DTWO_POT_OPTIMIZE and -DPACK_MALLOC Perl Build Options . . . 449

14.15.2 -Dusemymalloc Perl Build Option 449
14.16 Architecture Specific Compile Options 450
14.17 Maintainers 450
14.18 Authors 450
Frequent mod_perl problems 451
15 Frequent mod_perl problems 451
15.1 Description 452
15.2 my () scoped variable in nested subroutines 452
15.3 Segfaults caused by PerlFreshRestart 452
15.4 Maintainers 452
15.5 Authors 452
Warnings and Errors Troubleshooting Index 453
16 Warnings and Errors Troubleshooting Index 453
16.1 Description 454
16.2 General Advice 454
16.3 Building and Installation 454
16.3.1 Bareword "gensym" not allowed while "strict subs" 454
16.3.2 `No rule to make target ... CORE/config.h` 454
16.4 make test Issues 455
16.5 Configuration and Startup 455
16.5.1 'relocation errors' or 'undefined symbol' 455
16.5.2 SegFaults During Startup 456
16.5.3 libexec/libperl.so: open failed: No such file or directory 456
16.5.4 install_driver(Oracle) failed: Can't load '.../DBD/Oracle/Oracle.so' for module DBD::Oracle 457
16.5.5 Invalid command 'PerlHandler'... 458
16.5.6 symbol __floatdisf: referenced symbol not found 458
16.5.7 RegistryLoader: Translation of uri [...] to filename failed 459
16.5.8 "Apache.pm failed to load!" 459
16.6 Code Parsing and Compilation 459
16.6.1 Value of $x will not stay shared at - line 5 459
16.6.2 Value of $x may be unavailable at - line 5. 459
16.6.3 Can't locate loadable object for module XXX 459
16.6.4 Can't locate object method "get_handlers"... 459
16.6.5 Missing right bracket at line 460
16.6.6 Can't load '.../auto/DBI/DBI.so' for module DBI 460
16.7 Runtime 460
16.7.1 print() doesn't send anything on Mac OS X 461
16.7.2 "exit signal Segmentation fault (11)" with mysql 461
16.7.3 foo ... at /dev/null line 0 461
16.7.4 Preventing mod_perl Processes From Going Wild 461
16.7.5 Segfaults when using XML::Parser 461
16.7.6 My CGI/Perl Code Gets Returned as Plain Text Instead of Being Executed by the Webserver 462
16.7.7 Incorrect line number reporting in error/warn log messages 462
16.7.8 rwrite returned -1 462
16.7.9 Can't upgrade that kind of scalar 462

16.7.10 caught SIGPIPE in process . . . 462
16.7.11 Client hit STOP or Netscape bit it! . . . 463
16.7.12 Global symbol "$foo" requires explicit package name . . . 463
16.7.13 Use of uninitialized value at (eval 80) line 12. . . . 464
16.7.14 Undefined subroutine &Apache::ROOT::perl::test_2epl::some_function called at . . . 464
16.7.15 Callback called exit . . . 464
16.7.16 Out of memory! . . . 465
16.7.17 server reached MaxClients setting, consider raising the MaxClients setting . . . 466
16.7.18 syntax error at /dev/null line 1, near "line arguments:" . . . 466
16.7.19 Can't call method "register_cleanup" (CGI.pm) . . . 467
16.7.20 readdir() not working . . . 467
16.8 Shutdown and Restart . . . 467
16.8.1 Evil things might happen when using PerlFreshRestart . . . 467
16.8.2 Constant subroutine XXX redefined . . . 468
16.8.3 Can't undef active subroutine . . . 468
16.8.4 [warn] child process 30388 did not exit, sending another SIGHUP . . . 468
16.8.5 Processes Get Stuck on Graceful Restart . . . 469
16.8.6 httpd keeps on growing after each restart . . . 469
16.9 OS Specific Notes . . . 469
16.9.1 RedHat Linux . . . 469
16.9.1.1 RH 8 and 9 Locale Issues . . . 469
16.9.2 OpenBSD . . . 469
16.9.2.1 Issues Caused by httpd being `chroot`'ed . . . 469
16.9.3 Win FU . . . 470
16.9.3.1 Apache::DBI . . . 470
16.9.4 HP-UX . . . 470
16.9.4.1 `Apache::Status` Dumps Core Under HP-UX 10.20 . . . 470
16.10 Problematic Perl Modules . . . 470
16.11 Maintainers . . . 471
16.12 Authors . . . 471
Debugging mod_perl . . . 472
17 Debugging mod_perl . . . 472
17.1 Description . . . 473
17.2 Warning and Errors Explained . . . 473
17.2.1 Curing The "Internal Server Error" . . . 473
17.2.2 Helping error_log to Help Us . . . 477
17.2.3 The Importance of Warnings . . . 477
17.2.3.1 diagnostics pragma . . . 479
17.3 Handling the 'User pressed Stop button' case . . . 479
17.3.1 Detecting Aborted Connections . . . 480
17.3.2 The Importance of Cleanup Code . . . 482
17.3.2.1 Critical Section . . . 483
17.3.2.2 Safe Resource Locking and Cleanup Code . . . 486
17.4 Handling Server Timeout Cases and Working with $SIG{ALRM} . . . 490
17.5 Looking inside the server . . . 491
17.5.1 Apache::Status -- Embedded Interpreter Status Information . . . 491
17.5.1.1 Minimal Configuration . . . 491

17.5.1.2 Extended Configuration 492
17.5.1.3 Usage 493
17.5.1.4 Compiled Registry Scripts section seems to be empty. 494
17.5.2 mod_status 494
17.5.3 Apache::VMonitor -- Visual System and Apache Server Monitor 495
17.6 Sometimes My Script Works, Sometimes It Does Not 495
17.7 Code Debug 495
17.7.1 Locating and correcting Syntax Errors 496
17.7.2 Using Apache::FakeRequest to Debug Apache Perl Modules 497
17.7.3 Finding the Line Which Triggered the Error or Warning 498
17.7.4 Using print() for Debugging 499
17.7.5 Using print() and Data::Dumper for Debugging 501
17.7.6 The Importance of a Good Concise Coding Style 503
17.7.7 Introduction to the Perl Debugger 505
17.7.8 Interactive Perl Debugging under mod_cgi 514
17.7.9 Non-Interactive Perl Debugging under mod_perl 515
17.7.10 Interactive mod_perl Debugging 515
17.7.11 ptkdb and Interactive mod_perl Debugging 518
17.7.12 Debugging when Server Crashes on Startup before Writing to Log File. 519
17.8 Hanging Processes: Detection and Diagnostics 521
17.8.1 Hanging because of the OS Problem 521
17.8.2 An Example of Code that Might Hang a Process 521
17.8.3 Detecting hanging processes 523
17.8.4 Determination of the reason 524
17.8.4.1 Using the Perl Trace 525
17.8.4.2 Using the System Calls Trace 526
17.8.4.3 Using the Interactive Debugger 529
17.9 Debugging Hanging processes (continued) 531
17.9.1 Debugging core Dumping Code 532
17.10 PERL_DEBUG=1 Build Option 532
17.11 Apache::Debug 532
17.12 Debug Tracing 532
17.13 gdb says there are no debugging symbols 533
17.14 Debugging Signal Handlers ($SIG{FOO}) 534
17.15 Code Profiling 535
17.16 Devel::Peek 535
17.17 How can I find out if a mod_perl code has a memory leak 536
17.18 Debugging your code in Single Server Mode 538
17.19 Apache::DumpHeaders - Watch HTTP Transaction Via Headers 539
17.20 Apache::DebugInfo - Log Various Bits Of Per-Request Data 539
17.21 Maintainers 540
17.22 Authors 540
Getting Help 541
18 Getting Help 541
18.1 Description 542
18.2 READ ME FIRST 542
18.2.1 Please Ask Only Questions Related to mod_perl 542

18.3 How to Report Problems 542
18.4 Help on Related Topics 545
18.5 Maintainers 545
18.6 Authors 545