ApacheCon US
April 4, 2001
Santa Clara, CA

Tutorial: Improving scripts and handlers performance under mod_perl

by **Stas Bekman**
http://stason.org/
<stas@stason.org>
eXtropia.com, Senior Engineer

This talk is available from: http://stason.org/talks/

This document is originally written in **POD**, converted to **HTML**, **PostScript** and **PDF** by `Pod::HtmlPsPdf` Perl module.

(you will find a Table of Contents at the end of the Tutorial)

1 Agenda

1.1 Agenda

- I will start the presentation with a very basic introduction into mod_perl, 10 lines installation instructions, a simple configuration and a few code examples. These should help you get your feet wet if you are really new to mod_perl.

- Afterwards I'll talk about boosting a performance of web applications working with RDBMS databases under mod_perl. We will see what modules allow us to make the work with database faster.

- Finally we will see some performance improvement tips, these should get you programmers produce more efficient code. We will how one should measure performance and DO's and DON'T's to make the code run faster and use less memory.

- The is one more section left for the post-conference reading. It includes additional information about mod_perl and related products and resources. You should use it to find the answer to the questions that you might need to get answered, on your way to becoming a mod_perl guru or when you need some general help.

;o)

2 Getting Started Fast

2.1 mod_perl in Four Slides

Each tutorial will concentrate on different aspects of running a mod_perl server and mod_perl programming. In case you don't know how to get started with it, or you think it's a difficult task, these slides will take away any worries you might have had when you came to this tutorial.

In just four slides you will be able to install and configure a mod_perl server. And, of course, to write new code and reuse the existing code under mod_perl.

The four slides (sections) are:

- Installation
- Configuration
- The ''mod_perl rules'' Apache::Registry Scripts
- The ''mod_perl rules'' Apache Perl Module

2.2 What is mod_perl?

But before we go any further, there is a chance that you don't know what mod_perl is. So let's make a little introduction to mod_perl.

Everybody knows that Perl scripts running under mod_cgi have numerous shortcomings. There are many of them, but code recompilation and Perl interpreter loading overhead at each request is the hardest one to overcome.

Among various attempts to improve on mod_cgi's shortcomings, mod_perl has proved to be one of the better ones and has been widely adopted by CGI developers. According to the http://perl.apache.org/netcraft/ as of January 2001 about 2 million hosts use mod_perl. Doug MacEachern fathered the core code of this Apache module and licensed it under the Apache Software License.

mod_perl does away with mod_cgi's forking by reusing the existing child processes. In this new model, the child process doesn't exit anymore when it has processed a request. The Perl interpreter is loaded only once, when the process is started. Since the interpreter is persistent throughout the process' lifetime, all code is loaded and compiled only once, the first time it is seen. This makes all subsequent requests run much faster because everything is already loaded and compiled. Response processing is now reduced to running your code. This improves response times by a factor of 10 to 100, depending on the code being executed.

Doug didn't stop here, he went and extended mod_cgi's functionality by adding a complete Perl API to the Apache core. This makes it possible to write a complete Apache module in Perl, a feat that used to require coding in C. From then on mod_perl enabled the programmer to handle all phases of request processing in Perl.

The new Perl API also allows complete server configuration in Perl. This has which made the lives of many server administrators much easier, as they could now benefit from dynamically generating the configuration, freed from hunting for bugs in huge configuration files full of similar directives for virtual hosts and the like.

To provide backwards compatibility for plain CGI scripts that used to be run under mod_cgi, while still benefiting from a preloaded perl and modules, a few special handlers were written, each allowing a different level of proximity to pure mod_perl functionality. Some take full advantage of mod_perl, while others only a partial one.

mod_perl embeds a copy of the Perl interpreter into the Apache httpd executable, providing complete access to Perl functionality within Apache. This enables a set of mod_perl-specific configuration directives, all of which start with the string Perl*. Most, but not all, of these directives are used to specify handlers for various phases of the request.

It might occur to you that sticking a large executable (Perl) into another large executable (Apache) makes a very, very large program. mod_perl certainly makes httpd significantly bigger and you will need more RAM on your production server to be able to run many mod_perl processes, but in reality the situation is different. Since mod_perl processes requests much faster, the number of the processes needed to handle the same request rate is much lower relative to the mod_cgi approach. Generally you need slightly more memory available, and the speed improvements you will see are well worth every megabyte of memory you can add.

Now let's get back to the *All-In-Four-Slides*...

2.3 Installation

Did you know that it takes about 10 minutes to build and install a mod_perl enabled Apache server on a computer with a pretty average processor and a decent amount of system memory? It goes like this:

```
% cd /usr/src
% lwp-download http://www.apache.org/dist/apache_x.x.x.tar.gz
% lwp-download http://perl.apache.org/dist/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!

- Of course you must replace *x.x.x* with the actual version numbers of the mod_perl and Apache releases that you use.
- The GNU `tar` utility knows how to uncompress a gzipped tar archive (use the `z` option).

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

2.4 Configuration

Add the following to the configuration file *httpd.conf*:

```
  # for Apache::Registry mode
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 every URI starting with */perl* to be handled by the Apache mod_perl module. It will use the handler from the Perl module `Apache::Registry`.

2.5 The "mod_perl rules" Apache::Registry Scripts

You can write plain perl/CGI scripts just as under mod_cgi:

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

Of course you can write them in the Apache Perl API:

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

Save both files under the */home/httpd/perl* directory, make them executable and readable by server, and issue these requests using your favorite browser:

```
http://localhost/perl/mod_perl_rules1.pl
http://localhost/perl/mod_perl_rules2.pl
```

In both cases you will see on the following response:

```
mod_perl rules!
```

2.6 The "mod_perl rules" Apache Perl Module

To create an Apache Perl module, all you have to do is to wrap the code into a `handler` subroutine and return the status to the server.

```
ModPerl/Rules.pm
----------------
package ModPerl::Rules;
use Apache::Constants;

sub handler{
  my $r = shift;
  $r->send_http_header('text/plain');
  print "mod_perl rules!\n";
  return OK;
}
1;
```

Create a directory called *ModPerl* under one of the directories in `@INC`, and put *Rules.pm* into it. Then add the following snippet to *httpd.conf*:

```
PerlModule ModPerl::Rules
<Location /mod_perl_rules>
  SetHandler perl-script
  PerlHandler ModPerl::Rules
</Location>
```

Now you can issue a request to:

```
http://localhost/mod_perl_rules
```

and just as with our *mod_perl_rules.pl* scripts you will see:

```
mod_perl rules!
```

as the response.

2.7 Is That All I Need To Know About mod_perl?

Definitely not!

These slides are intended to show you that you can install and start using a mod_perl server within 30 minutes of downloading the sources.

There is much more to mod_perl than this, you will need to plan your study around the projects you want to implement. Fortunately, there are many resources and lots of help freely available to you.

At the end of this titorial you will find a chapter describing the available resources and pointers to them.

;o)

3 RDBMS and mod_perl

3.1 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.

3.1.1 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.

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. Each connection will stay persistent and in a short time the number of connections will be so big that your machine will scream in agony and die.

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.

3.1.2 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`.

3.1.3 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!

3.1.4 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"
```

3.1.5 Opening connections with different parameters

When it receives a connection request, before it decides to use an existing cached connection, `Apache::DBI` insists that the new connection be opened in exactly the same way as the cached connection. If I have one script that sets `LongReadLen` and one that does not, `Apache::DBI` will make two different connections. So instead of having a maximum of 40 open connections, I can end up with 80.

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 `LongReadLen` (or whatever) afterwards.

3.1.6 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!

;o)

4 Performance Tuning

4.1 What we will learn in this chapter

- The Big Picture
- Essential Tools
- Choosing MaxClients
- KeepAlive
- PerlSetupEnv Off
- Reducing the Number of `stat()` Calls Made by Apache
- Cached `stat()` Calls by Perl
- Limiting the Size of the Processes
- Sharing Memory
- How Shared My Memory Is
- Preload Perl modules at server startup
- Preload Registry Scripts
- Some numbers: Initializing DBI.pm
- Keeping the Shared Memory Limit
- Limiting the Resources Used by httpd Children
- Upload/Download of Big Files
- Global vs Fully Qualified Variables
- Forking or Executing subprocesses from mod_perl
- Using $|=1 under mod_perl and better `print()` techniques.
- Sending plain HTML as a compressed output

4.2 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, and it is very important for you to 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 rendered complete.

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 elements 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 they will all have to wait for the last one so that the full request message can be reassembled at 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 fast, 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. You will see more examples in this chapter.

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.

4.3 Essential Tools

In order to improve performance we need measurement tools. We use benchmarking for this purpose. We can benchmark the code and we can benchmark the response time which in addition to the code execution measures the request arrival and response delivery time amongst other things.

4.3.1 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 (<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)
```

An example of the `Time::HiRes` module usage:

```
hi-res.pl
---------
use Time::HiRes qw(gettimeofday tv_interval);
sub sub_that_takes_a_teeny_bit_of_time{1+1;};
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.\n"
```

```
% perl hi-res.pl
The sub took 0.000262 seconds.
```

4.3.2 Benchmarking Response Times

To measure response times all we need is a client that will generate parallel requests, process the responses and print the results of the test. You can use either an existing tool that performs this task or you can develop your own.

4.3.2.1 ApacheBench

From existing tools you can try ApacheBench (ab) that comes bundled with Apache source distribution. 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.

Let's try it. We will simulate 10 users concurrently requesting a very light script at `www.example.com:81/test/test.pl`. Each simulated user makes 10 requests.

```
% ./ab -n 100 -c 10 www.example.com:81/test/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
```

4.3.2.2 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), independent 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
```

4.3.3 Using LWP::Parallel::UserAgent

You can use `LWP::Parallel::UserAgent` to write your own bechmarking utility.

This is another crashme suite originally written by Michael Schilli (and used to be located at http://www.linux-magazin.de/ausgabe.1998.08/Pounder/pounder.html, but it's 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:81/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:

```
#!/usr/apps/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",
      "Parallel Agents" => $nof_parallel_connections,
      "Succeeded"       => sprintf("$succeeded (%.2f%%)\n",
                                   $succeeded * 100 / $nof_requests_total),
      "Errors"          => $errors,
      "Total Time"      => sprintf("%.2f secs\n", $total_time),
      "Throughput"      => sprintf("%.2f Requests/sec\n",
                                   $nof_requests_total / $total_time),
      "Latency"         => sprintf("%.2f secs/Request",
                                   ($ua->get_latency_total() || 0) /
                                   $nof_requests_total),
     );
```

```
my ($left, $right);
###
# Print out statistics
###
format STDOUT =
@<<<<<<<<<<<<<<< @*
"$left:",        $right
.

while(($left, $right) = splice(@P, 0, 2)) {
  write;
}
```

4.4 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, 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 accompanied 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. 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 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:

```
              Total_RAM + Shared_RAM_per_Child * (MaxClients - 1)
MaxClients = ----------------------------------------------------
                            Max_Process_Size
```

which is:

```
                   Total_RAM - Shared_RAM_per_Child
MaxClients = ----------------------------------------
             Max_Process_Size - 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.

4.5 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 each request processing, the process will wait for `KeepAliveTimeout` seconds before closing the connection and thus not serving other requests at 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 then by setting `KeepAlive Off` 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, turning it off will be 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.

Also you should know that `KeepAlive` requests only work with responses that contain a `Content-Length` header. To send this header do:

```
$r->header_out('Content-Length', $length);
```

4.6 PerlSetupEnv Off

`PerlSetupEnv Off` is another optimization you might consider.

mod_perl fiddles with the environment to make it appear as if the script were being called under the CGI protocol. 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, can gain by turning it **Off**.

By default it is On.

Note that you can still set enviroment variables. For example when you use the following configuration:

```
<Location /perl>
  SetHandler perl-script
  PerlHandler +Apache::RegistryNG

  PerlSetupEnv Off
  PerlSetEnv TEST hi

  Options +ExecCGI
</Location>
```

and you issue a request (for example http://localhost/perl/setupenvoff.pl) 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/perl5/5.6.1:... snipped ...',
          'TEST' => 'hi'
        };
```

Notice that we have gotten the environment variable *TEST* set.

4.7 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 many `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
```

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:

```
<VirtualHost 10.10.10.10:80>
  ...
  PerlTransHandler  Apache::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 you have `AllowOverride None` directive, Apache will 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 ot 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:

```
  PerlTransHandler  Apache::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::OK : Apache::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.

4.8 Cached stat() Calls by Perl

When you do a `stat()` (or its variations `-M` -- last modification time, `-A` -- last access time, `-C` -- last inode-change time, and others), the information is cached. If you need to make an additional check for the same file, use the _ variable and save the overhead of this check. For example when testing for existence and read permissions you might use:

```
my $filename = "./test";
  # two 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";
  # two stat() calls
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()` syscalls saved!

4.9 Be carefull with symbolic links

As you know `Apache::Registry` caches the scripts based on their 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 cached twice!

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.

To detect this, use the `/perl-status` (`Apache::Status`) handler to see all the compiled scripts and their packages. In our example, when requesting: http://localhost/perl-status?rgysubs you would see:

```
Apache::ROOT::perl::news::news_2epl
Apache::ROOT::perl::news_2epl
```

after the both URIs have been requested from the same child process that happened to serve your request. To make the debugging easier see run the server in single mode.

4.10 Limiting the Size of the Processes

`Apache::SizeLimit` allows you to kill off Apache httpd processes if they grow too large.

Configuration:

In your *startup.pl*:

```
use Apache::SizeLimit;
$Apache::SizeLimit::MAX_PROCESS_SIZE = 10000;
# in KB, so this is 10MB
```

In your *httpd.conf*:

```
PerlFixupHandler Apache::SizeLimit
```

See perldoc `Apache::SizeLimit` for more details.

By using this module, you should be able to avoid using the Apache configuration directive `MaxRequestsPerChild`, although for some folks, using both in combination does the job.

4.11 Sharing Memory

A very important point is the sharing of memory. If your OS supports this (and most sane systems do), you might save more 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 becomes unshared and there is no way we can control perl to make it allocate memory so (dynamic) variables land on different memory pages than constants, that's why the **copy-on-write** effect (will explain in a moment) will hit almost at random. If you are pre-loading many modules you might be able to balance the memory that stays shared against the time for an occasional fork by tuning the `MaxRequestsPerChild` to a point where you restart before too much becomes unshared. In this case the `MaxRequestsPerChild` is very specific to your scenario. You should do some measurements and you might see if this really makes a difference and what a reasonable number might be. Each time a child reaches this upper limit and restarts it should release the unshared copies and the new child will inherit pages that are shared until it scribbles on them.

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 speedup, so if it is 100 or 10000 it does not really matter if it saves you the RAM by sharing. Do not forget that if you preload most of your code at the server startup, the fork to spawn a new child will be very very fast, because it inherits most of the preloaded code and the perl interpreter from the parent process. But than, during the work of the child, its memory pages (which aren't really its yet, it uses the parent's pages) are getting dirty (originally inherited and shared variables are getting updated/modified) and the **copy-on-write** happens, which reduces the number of shared memory pages - thus enlarging the memory demands. Killing the child and respawning a new one, allows to get the pristine shared memory from the parent process again.

The conclusion is that `MaxRequestsPerChild` should not be too big, otherwise you loose the benefits of the memory sharing.

4.12 How Shared My Memory Is

You've probably noticed that the word shared is being repeated many times in many things related 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.

How much shared memory do you have? You can see it by either using the memory utils that comes with your system or you can deploy `GTop` module:

```
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 **RSS** (or **RES**) column with **SHARE** column -- **RES** is a RESident memory, which is a size of pages currently swapped in.

4.13 Keeping the Shared Memory Limit

`Apache::GTopLimit` module allows you to kill off Apache httpd processes if they grow too large (just like `Apache::SizeLimit`) or have too little of shared memory.

Configuration:

In your *startup.pl*:

```
use Apache::GTopLimit;

# Control the life based on memory size
# in KB, so this is 10MB
$Apache::GTopLimit::MAX_PROCESS_SIZE = 10000;

# Control the life based on Shared memory size
# in KB, so this is 4MB
$Apache::GTopLimit::MIN_PROCESS_SHARED_SIZE = 4000;

# watch what happens
$Apache::GTopLimit::DEBUG = 1;
```

In your *httpd.conf*:

```
PerlFixupHandler Apache::GTopLimit
```

4.14 Preload Perl modules at server startup

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 even a better approach is to create a separate startup file (where you code in plain perl) and put there things like:

```
use DBI;
use Carp;
```

Then you `require()` this startup file with help of `PerlRequire` directive from `httpd.conf`, by 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. However, if you are using mod_perl, FastCGI or another system that uses a persistent Perl interpreter, you will want to precompile the methods 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 tag names you really use in your code, since generally only a subset of subs is actually being used.

4.15 Some numbers: 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 are going to run memory benchmarks on five different versions of the *startup.pl* file. We always preload these modules:

```
use Gtop();
use Apache::DBI(); # preloads DBI as well
```

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";
```

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 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:

```
Version     Size   Shared     Diff        Test type
-------------------------------------------------------------------
      1  3465216  2621440   843776  install_driver
      2  3461120  2609152   851968  install_driver & connect_on_init
      3  3465216  2605056   860160  preload driver
      4  3461120  2494464   966656  nothing added
      5  3461120  2482176   978944  connect_on_init
```

2. After the second request (all the subsequent request showed the same results):

```
Version     Size   Shared    Diff         Test type
-------------------------------------------------------------------
      1  3469312  2609152   860160  install_driver
      2  3481600  2605056   876544  install_driver & connect_on_init
      3  3469312  2588672   880640  preload driver
      4  3477504  2482176   995328  nothing added
      5  3481600  2469888  1011712  connect_on_init
```

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 (1). Since we want to have a connection ready for the first request made to the freshly spawned child process, we generally use the second version (2) which uses somewhat more memory, but has almost the same number of shared memory pages. The third version only preloads the driver which results in smaller shared memory. The last two versions having nothing initialized (4) and having only the `connect_on_init()` method used (5). 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:

```
                    Total_RAM - Shared_RAM_per_Child
MaxClients = ----------------------------------------
             Max_Process_Size - Shared_RAM_per_Child
```

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 version (1) 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 1)  N =  ------------------- = 309
                     860160
```

```
               268435456 - 2469888
(ver 5)  N =  ------------------- = 262
                    1011712
```

So you can tell the difference (17% more child processes in the first version).

4.16 Preload Registry Scripts

`Apache::RegistryLoader` compiles `Apache::Registry` scripts at server startup. It can be a good idea to preload the scripts you are going to use as well. So the code will be shared among the children.

Here is an example of the use of this technique. This code is included in a `PerlRequire`'d file, and walks the directory tree under which all registry scripts are installed. For each `.pl` file encountered, it calls the `Apache::RegistryLoader::handler()` method to preload the script in the parent server (before pre-forking the child processes):

```
use File::Find 'finddepth';
use Apache::RegistryLoader ();
{
    my $perl_dir = "perl/";
    my $rl = Apache::RegistryLoader->new;
    finddepth(sub {
        return unless /\.pl$/;
        my $url = "/$File::Find::dir/$_";
        print "pre-loading $url\n";

        my $status = $rl->handler($url);
        unless($status == 200) {
            warn "pre-load of '$url' failed, status=$status\n";
        }
    }, $perl_dir);
}
```

Note that we didn't use the second argument to `handler()` here, as module's manpage suggests. To make the loader smarter about the uri->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()` routine was not defined, the loader will try using the URI relative to **ServerRoot**.

4.17 Limiting the Resources Used by httpd Children

`Apache::Resource` uses the `BSD::Resource` module, which in turn uses the C function `setrlimit()` to set limits on system resources such as memory and cpu usage.

To configure:

```
PerlModule Apache::Resource
  # set child memory limit in megabytes
  # (default is 64 Meg)
PerlSetEnv PERL_RLIMIT_DATA 32:48

  # set child CPU limit in seconds
  # (default is 360 seconds)
PerlSetEnv PERL_RLIMIT_CPU 120

PerlChildInitHandler Apache::Resource
```

If you configure `Apache::Status`, it will let you review the resources set in this way.

The following limit values are in megabytes: `DATA`, `RSS`, `STACK`, `FSIZE`, `CORE`, `MEMLOCK`; all others are treated as their natural unit. Prepend `PERL_RLIMIT_` for each one you want to use. Refer to the `setrlimit` man page on your OS for other possible 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.

4.18 Upload/Download of Big Files

You don't want to tie up your precious mod_perl backend server children doing something as long and dumb as transfering a file. The user won't really see any important performance benefits from mod_perl anyway, since the upload may take up to several minutes, and the overhead saved by mod_perl is typically under one second.

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.

This of course assumes that the script requires none of the functionality of the mod_perl server, such as custom authentication handlers.

4.19 Global vs Fully Qualified Variables

It's always a good idea to stay away from global variables when possible. Some variables must be global so Perl can see them, such as a module's `@ISA` or `$VERSION` variables (or fully qualified **@MyModule::ISA**). In common practice, a combination of `strict` and `vars` pragmas keeps modules clean and reduces a bit of noise. However, `vars` pragma also creates aliases as the `Exporter` does, which eat up more memory. When possible, try to use fully qualified names instead of use vars. Example:

```
package MyPackage;
use strict;
@MyPackage::ISA = qw(...);
$MyPackage::VERSION = "1.00";
```

vs.

```
package MyPackage;
use strict;
use vars qw(@ISA $VERSION);
@ISA = qw(...);
$VERSION = "1.00";
```

4.20 Forking or Executing subprocesses from mod_perl

Generally you should not fork from your mod_perl scripts, since when you do -- you are forking the entire apache web server, lock, stock and barrel. Not only is your perl code being duplicated, but so is mod_ssl, mod_rewrite, mod_log, mod_proxy, mod_spelling or whatever modules you have used in your server, all the core routines and so on.

A much wiser approach would be to spawn a sub-process, hand it the information it needs to do the task, and have it detach (close `STD*` + `setsid()`). This is wise only if the parent who spawns this process, immediately continues, you do not wait for the sub-process to complete. This approach is suitable for a situation when you want to trigger a long time taking process through the web interface, like processing some data, sending email to thousands of subscribed users and etc. Otherwise, you should convert the code into a module, and use its functions or methods to call from CGI script.

Just making a `system()` call defeats the whole idea behind mod_perl, perl interpreter and modules should be loaded again for this external program to run.

Basically, you would do:

```
$params=FreezeThaw::freeze(
      [all data to pass to the other process]
      );
system("program.pl $params");
```

and in `program.pl`:

```
use POSIX qw(setsid);
@params=FreezeThaw::thaw(shift @ARGV);
# check that @params is ok
close STDIN;
close STDOUT;
close STDERR;
# you might need to reopen the STDERR
# open STDERR, ">/dev/null";
setsid(); # to detach
```

At this point, `program.pl` is running in the ‘‘background’’ while the `system()` returns and permits apache to get on with life.

This has obvious problems. Not the least of which is that `@params` must not be bigger then whatever your architecture’s limit is (could depend on your shell).

Also, the communication is only one way.

However, you might want be trying to do the ‘‘wrong thing’’. If what you want is to send information to the browser and then do some post-processing, look into `PerlCleanupHandler`.

If you are interested in more deep level details, this is what actually happens when you fork and make a system call, like

```
system("echo Hi"),CORE::exit(0) unless fork();
```

which is might be more familiar in this form:

```
if (fork){
  #do nothing
} else {
  system("echo Hi");
  CORE::exit(0);
}
```

What happens is that `fork()` gives you 2 execution paths and the child gets virtual memory sharing a copy of the program text (read only) and sharing a copy of the data space copy-on-write (remember why you pre-load modules in mod_perl?). In the above code a parent will immediately continue with the code that comes up after the fork, while the forked process will execute `system("echo Hi")` and then terminate itself.

Notice that I use `CORE::exit` and not `exit` which would be automatically overriden by `Apache::exit` if used in conjunction with `Apache::Registry` and friends.

The only work is setting up the page tables for the virtual memory and the second process goes on its separate way.

Next, Perl will find `/bin/echo` along the search path, and invoke it directly. Perl `system()` is *not* `system(3)` [C-library]. Only when the command has shell meta-chars does Perl invoke a real shell. That's a *very* nice optimization.

Only if you do:

```
system "sh -c 'echo foo'"
```

OS actually parses your command with a shell so you `exec()` a copy of `/bin/sh`, but since one 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 pointed at the already-loaded program code plus your own data space. Then the shell parses the passed command.

Since it is `echo`, it will execute it as a built-in in the latter example or a `/bin/echo` in the former and be done, but this is only an example. You aren't calling `system("echo Hi")` in your mod_perl scripts, right? Since most other real things (heavy programs executed as a subprocess) would involve repeating the process to load the specified command or script (it might involve some actual demand paging from the program file if you execute new code).

The only place you see real overhead from this scheme is when the parent process is huge (unfortunately like mod_perl...) and the page table becomes large as a side effect. The whole point of mod_perl is to avoid having to `fork()` / `exec()` something on every hit, though. Perl can do just about anything by itself. However, you probably won't get in trouble until you hit about 30 forks/sec on a so-so pentium.

Now let's get to the gory details of forking.

4.20.1 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.

4.20.2 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.

4.20.3 Avoiding Zombie Processes

Normally, every process has its parent. Many processes are children of the `init` process, whose `PID` equals to `1`. When you fork a process you must `wait()` or `waitpid()` for it to finish. If you don't wait for it becomes a zombie.

Zombie, is a process that doesn't have a father. When the child quits, it reports the termination to his parent. If no one `wait()s` to collect the exit status of the child, it gets ''confused'' and becomes a ghost process, that can be seen, but not killed. It will be killed only when you stop the httpd process that spawned it! (generally `top()/ps()` utilities display these processes with `<defunc>` tag, and you will see an increment of the zombies counter reported when doing `top().)` These zombie processes can take up system resources and are generally undesirable.

So the proper fork is:

```
print "Content-type: text/plain\n\n";

defined (my $kid = fork) or die "Cannot fork: $!\n";
if ($kid) {
  waitpid($kid,0);
  print "Parent has finished\n";
} else {
    # do something
    CORE::exit(0);
}
```

But in most cases the only reason you would want to fork is when you need to spawn a process that would take a lot of time to complete. So if the server child that spawns this process has to wait for it to finish, you gained nothing. You cannot neither wait for its completion, nor continue because you will get yet another zombie process.

The simplest solution is to ignore your dead children (this doesn't work everywhere, however).

```
$SIG{CHLD} = IGNORE;
```

When you set `CHLD` signal handler to `IGNORE`, all the processes will be collected by the `init` process and prevent from them to become zombies.

Note, that you cannot localize this setting with `local()`. If you do, it wouldn't take the desired effect.

The other thing that you must do -- is to close all the pipes to the connection socket that were opened by the parent process (a `STDIN` and a `STDOUT`) and inherited by the child, so the parent will be able to complete the request and free itself for serving other requests. You may need to close and reopen a `STDERR` filehandler (It's opened to append to the error_log file as inhereted by parent, so chances are that you want it to leave untouched).

So now the code would look like:

```
print "Content-type: text/plain\n\n";

$SIG{CHLD} = IGNORE;

defined (my $kid = fork) or die "Cannot fork: $!\n";
if ($kid) {
  waitpid($kid,0);
  print "Parent has finished\n";
} else {
    close STDIN;
    close STDOUT;
    close STDERR;
    # do something long lasting
    CORE::exit(0);
}
```

Another more portable, but slightly more expensive solution is to use a double fork approach.

```
print "Content-type: text/plain\n\n";

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
    close STDIN;
    close STDOUT;
    close STDERR;
    # do something long lasting
    CORE::exit(0);
  }
}
```

Grandkid becomes a *"child of init"* (parent process ID is 1).

Note that the last two solutions do allow you to know the exit status of the process, but in our case we don't want to.

One more solution is to use a different *SIGCHLD* handler:

```
use POSIX 'WNOHANG';
$SIG{CHLD} = sub { while( waitpid(-1,WNOHANG)>0 ) {} };
```

Which is usefull when you `fork()` more than once process. The handler could call `wait()` as well, but for a variety of reasons involving tge 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 from reaping. The handler will loop untill `waitpid()` returns a negative number or zero, indicating that no more reapable children remain.

You will probably want to open your own log file in the spawned process and log some info so you know what have happened there. At least while debugging your code.

4.20.4 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 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
```

4.21 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 a _bad_ style in generating output, which consist of multiple `print()` calls, or you just have too many of them, you will experience a degradation in performance. The severity depends on the number of the calls you make.

Many old CGIs were written in the style of:

```
print "<BODY BGCOLOR=\"black\" TEXT=\"white\">";
print "<H1>";
print "Hello";
print "</H1>";
print "<A HREF=\"foo.html\"> foo </A>";
print "</BODY>";
```

which reveals the following drawbacks: multiple `print()` calls - performance degradation with `$|=1`, backslashism which makes the code less readable and more difficult to format the HTML to be easily readable as CGI's output. The code below solves them all:

```
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 might not accept the output as HTML, but rather print it as a plain text, since they expect the first characters after the headers and empty line to be `<HTML>` and not spaces and/or additional newline and then `<HTML>`. Even if it works with your browser, it might not work for others.

Now let's go back to the `$|=1` topic. I still disable buffering, for 2 reasons: I use few `print()` calls by printing out multiline HTML and not a line per `print()` and I want my users to see the output immediately. So if I am about to produce the results of the DB query, which might take some time to complete, I want users to get some titles ahead. This improves the usability of my site. Recall yourself: What do 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 awhile and then to receive the whole output at once, even a few millisecs faster (if the client (browser) did not time out till then).

An even better solution is to keep the buffering enabled, and use a Perl API `rflush()` call to flush the buffers when wanted. This way you can aggregate in the buffer the top of the page you are going to send to user, and flush it a moment before you are going to do some lenghty operation, like DB query. So you kill the two birds in one shoot: You show some of the data to the user immediately, so user will feel that something is actually happening, and you almost have no performance hit caused by

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 lenghty operation
for (1..5) {
  sleep 1;
}
print $q->p("Done!");
```

Conclusion: Do not blindly follow suggestions, but think what is best for you in every given case.

4.22 Sending plain HTML as a compressed output

Have you ever served a huge HTML file (e.g. a file bloated with JavaScript code) and wandered how could you send it compressed, thus drammatically cutting down the download times. After all java applets can be compressed into a jar and benefit from a faster download times. Why cannot we do the same with a plain ASCII (HTML,JS and etc), it is a known fact that ASCII text can 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. A client decompresses the data upon receive and renders the HTML as if it was a plain HTML fetch.

For example to compress all html files on the fly, do:

```
<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, thru `Accept-Encoding` header. `Apache::GzipChain` keeps a list of user-agents, thus it also looks at `User-Agent` header, for known to accept compressed output browsers.

For example if you want to return compressed files which should pass in addition through Embperl module, you would write:

```
<Location /test>
  SetHandler perl-script
  PerlHandler Apache::OutputChain Apache::GzipChain Apache::EmbperlChain Apache::PassFile
</Location>
```

Hint: Watch an `access_log` file to see how many bytes were actually send, compare with a regular configuration send.

(See `perldoc Apache::GzipChain`).

Notice that the rightmost PerlHandler must be a content producer. Use `Apache::PassFile` or another similar module.

;o)

5 Getting Help and Further Learning

5.1 What we will learn in this chapter

- Getting help
- Get help with mod_perl
- Get help with Perl
- Get help with Perl/CGI
- Get help with Apache
- Get help with DBI
- Get help with Squid

5.2 Getting help

If after reading this guide and other documents listed in this section, you feel that your question is not yet answered, please ask the apache/mod_perl mailing list to help you. But first try to browse the mailing list archive. Most of the time you will find the answer for your question by searching the mailing archive, since there is a big chance someone else has already encountered the same problem and found a solution for it. If you ignore this advice, do not be surprised if your question will be left unanswered - it bores people to answer the same question more than once. It does not mean that you should avoid asking questions. Just do not abuse the available help and **RTFM** before you call for **HELP**. (You have certainly heard the infamous fable of the shepherd boy and the wolves)

5.3 Get help with mod_perl

- **mod_perl home**

  http://perl.apache.org

- **News and Resources**

  Take23: News and Resources for the mod_perl world http://take23.org

- **mod_perl Books**

  - **'Apache Modules' Book**

    http://www.modperl.com is the home site of The Apache Modules Book, a book about creating Web server modules using the Apache API, written by Lincoln Stein and Doug MacEachern.

    Now you can purchase the book at your local bookstore or from the online dealer. O'Reilly lists this book as:

```
Writing Apache Modules with Perl and C
By Lincoln Stein & Doug MacEachern
1st Edition March 1999
1-56592-567-X, Order Number: 567X
746 pages, $34.95
```

- **'Managing and Programming mod_perl' Book**

  http://www.modperlbook.com is the home site of the new mod_perl book, that Eric Cholet and Stas Bekman are co-authoring. We expect the book to be published in 2001.

  Ideas, suggestions and comments are welcome. Please send them to info@modperlbook.com .

- **mod_perl Quick Reference Card**

  mod_perl Pocket Reference by Andrew Ford was published by O'Reilly and Associates http://www.oreilly.com/catalog/modperlpr/

  You should probably get also the *Apache Pocket Reference* by the same author and the same publisher: http://www.oreilly.com/catalog/apachepr/

  See also Andrew's collection of reference card for Apache and other programs: http://www.refcards.com.

- **mod_perl Guide**

  by Stas Bekman at http://perl.apache.org/guide

- **mod_perl FAQ**

  by Frank Cringle at http://perl.apache.org/faq/ .

- **mod_perl performance tuning guide**

  by Vivek Khera at http://perl.apache.org/tuning/ .

- **mod_perl plugin reference guide**

  by Doug MacEachern at http://perl.apache.org/src/mod_perl.html .

- **Quick guide for moving from CGI to mod_perl**

  at http://perl.apache.org/dist/cgi_to_mod_perl.html .

- **mod_perl_traps, common traps and solutions for mod_perl users**

  at http://perl.apache.org/dist/mod_perl_traps.html .

- **mod_perl Resources Page**

  http://www.perlreference.com/mod_perl/

- **mod_perl mailing list**

  The Apache/Perl mailing list (modperl@apache.org) **is available for mod_perl users and developers to share ideas, solve problems and discuss things related to mod_perl and the Apache::* modules.** To subscribe to this list, send mail to modperl-subscribe@apache.org with empty `Subject` and with `Body`:

  ```
  subscribe modperl
  ```

  A **searchable** mod_perl mailing list archive available at http://forum.swarthmore.edu/epigone/modperl . We owe it to Ken Williams.

  More archives available:

  - http://www.geocrawler.com/lists/3/web/182/0/
  - http://www.bitmechanic.com/mail-archives/modperl/
  - http://www.mail-archive.com/modperl%40apache.org/
  - http://www.davin.ottawa.on.ca/archive/modperl/
  - http://www.progressive-comp.com/Lists/?l=apache-modperl&r=1&w=2#apache-modperl
  - http://www.egroups.com/group/modperl/

5.4 Get help with Perl

- **The Perl FAQ**

  http://www.perl.com/CPAN/doc/FAQs/FAQ/PerlFAQ.html

- **The Perl home**

  http://www.perl.com/

- **The Perl Journal**

  http://www.tpj.com/

- **Perl Module Mechanics**

  http://world.std.com/~swmcd/steven/perl/module_mechanics.html - This page describes the mechanics of creating, compiling, releasing and maintaining Perl modules.

5.5 Get help with Perl/CGI

- **Perl/CGI FAQ**

at http://www.perl.com/CPAN/doc/FAQs/cgi/perl-cgi-faq.html

- **Answers to some bothering Perl and Perl/CGI questions**

  http://stason.org/TULARC/webmaster/myfaq.html

- **Idiot's Guide to CGI programming**

  http://www.perl.com/CPAN/doc/FAQs/cgi/idiots-guide.html

- **WWW Security FAQ**

  http://www.w3.org/Security/Faq/www-security-faq.html

- **CGI/Perl Taint Mode FAQ**

  http://www.gunther.web66.com/FAQS/taintmode.html (by Gunther Birznieks)

5.6 Get help with Apache

- **Apache Project's Home**

  http://www.apache.org

- **Apache Quick Reference Card**

  http://www.refcards.com (Apache and other refcards are available from this link)

- **The Apache FAQ**

  http://www.apache.org/docs/misc/FAQ.html

- **Apache Server Documentation**

  http://www.apache.org/docs/

- **Apache Handlers**

  http://www.apache.org/docs/handler.html

- **mod_rewrite Guide**

  http://www.engelschall.com/pw/apache/rewriteguide/

5.7 Get help with DBI

- **Perl DBI examples**

  http://www.saturn5.com/~jwb/dbi-examples.html (by Jeffrey William Baker).

- **DBI Homepage**

  http://www.symbolstone.org/technology/perl/DBI/

- **DBI mailing list information**

  http://www.fugue.com/dbi/

- **DBI mailing list archives**

  http://outside.organic.com/mail-archives/dbi-users/
  http://www.xray.mpe.mpg.de/mailing-lists/dbi/

- **Persistent connections with mod_perl**

  http://perl.apache.org/src/mod_perl.html#PERSISTENT_DATABASE_CONNECTIONS

5.8 Get help with Squid - Internet Object Cache

- Home page - http://squid.nlanr.net/
- FAQ - http://squid.nlanr.net/Squid/FAQ/FAQ.html
- Users Guide - http://squid.nlanr.net/Squid/Users-Guide/
- Mailing lists - http://squid.nlanr.net/Squid/mailing-lists.html

;o)

Table of Contents:

Tutorial: Improving scripts and handlers performance under mod_perl 1
mod_perl Tutorial: Agenda 3
1 Agenda 3
1.1 Agenda 4
mod_perl Tutorial: Getting Started Fast 5
2 Getting Started Fast 5
2.1 mod_perl in Four Slides 6
2.2 What is mod_perl? 6
2.3 Installation 7
2.4 Configuration 8
2.5 The "mod_perl rules" Apache::Registry Scripts 8
2.6 The "mod_perl rules" Apache Perl Module 9
2.7 Is That All I Need To Know About mod_perl? 9
mod_perl Tutorial: RDBMS and mod_perl 11
3 RDBMS and mod_perl 11
3.1 Apache::DBI - Initiate a persistent database connection 12
3.1.1 Introduction 12
3.1.2 Configuration 13
3.1.3 Preopening DBI connections 13
3.1.4 Debugging Apache::DBI 13
3.1.5 Opening connections with different parameters 14
3.1.6 Caching prepare() Statements 14
mod_perl Tutorial: Performance Tuning 15
4 Performance Tuning 15
4.1 What we will learn in this chapter 16
4.2 The Big Picture 16
4.3 Essential Tools 17
4.3.1 Benchmarking Perl Code 17
4.3.2 Benchmarking Response Times 18
4.3.2.1 ApacheBench 18
4.3.2.2 httperf 19
4.3.3 Using LWP::Parallel::UserAgent 19
4.4 Choosing MaxClients 23
4.5 KeepAlive 25
4.6 PerlSetupEnv Off 26
4.7 Reducing the Number of stat() Calls Made by Apache 27
4.8 Cached stat() Calls by Perl 30
4.9 Be carefull with symbolic links 31
4.10 Limiting the Size of the Processes 31
4.11 Sharing Memory 32
4.12 How Shared My Memory Is 32
4.13 Keeping the Shared Memory Limit 33
4.14 Preload Perl modules at server startup 33
4.15 Some numbers: Initializing DBI.pm 34
4.16 Preload Registry Scripts 38
4.17 Limiting the Resources Used by httpd Children 38
4.18 Upload/Download of Big Files 39
4.19 Global vs Fully Qualified Variables 39

4.20 Forking or Executing subprocesses from mod_perl 40
4.20.1 Freeing the Parent Process 42
4.20.2 Detaching the Forked Process 43
4.20.3 Avoiding Zombie Processes 43
4.20.4 A Complete Fork Example 45
4.21 Using $|=1 under mod_perl and better print() techniques. 47
4.22 Sending plain HTML as a compressed output 49
mod_perl Tutorial: Getting Help and Further Learning 51
5 Getting Help and Further Learning 51
5.1 What we will learn in this chapter 52
5.2 Getting help 52
5.3 Get help with mod_perl 52
5.4 Get help with Perl 54
5.5 Get help with Perl/CGI 54
5.6 Get help with Apache 55
5.7 Get help with DBI 55
5.8 Get help with Squid - Internet Object Cache 56