

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>](mailto:stas@stason.org)
eXtropia.com, Senior Engineer

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

1 Agenda

1.1 Agenda

- mod_perl Introduction
- Basic Configuration
- Basic Scripts and Handlers
- mod_perl and RDBMS
- Improving Performance.

1.2 Off-tutorial reading

- Getting Help

```
;0
```


2 Getting Started Fast

2.1 mod_perl in Four Slides

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

2.2 What is mod_perl?

Solves numerous mod_cgi shortcomings:

- Embedded Perl Interpreter -- no loading overhead
- Code compiled only once per process life -- no compilation overhead
- No forking per request -- process reuse
- Response processing is now reduced to running your code.
- Response times improve by a factor of 10 to 100

- A bigger size, but just a few processes can handle a much bigger load
- `mod_cgi` compatibility preserved (Apache::Registry and Apache::PerlRun modules)
- Persistent database connections

Extended mod_cgi's functionality:

- A complete Perl API added to the Apache core
- Handling of all phases of request processing in Perl.
- Writing complete Apache modules in Perl
- Complete server configuration in Perl.
- Numerous 3rd party modules are available

Logistics:

- Developed by Doug MacEachern
- Licensed under the Apache Software License.
- Home page <http://perl.apache.org>
- Mailing list: send an empty email to modperl-subscribe@apache.org
- January 2001 -- 2 Million mod_perl hosts (according to <http://perl.apache.org/netcraft/>)

2.3 Installation

```
% 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!**

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>
```

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:

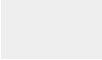
```
ModPerl/Rules.pm
-----
package ModPerl::Rules;
use Apache::Constants;

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.
- Fortunately, there are many resources and lots of help freely available to you.

- See the last chapter of this tutorial for the help references.

```
;0
```


3 RDBMS and mod_perl

3.1 Apache::DBI - Initiate a persistent database connection

- `mod_cgi` limitation -- its database connection is not persistent
- `Apache::DBI` removes this limitation.
- A database connection persists for the process' entire life.
- Available only under `mod_perl`
- No changes to your code required

3.1.1 Introduction

- The DBI module can make use of the `Apache::DBI` module.
- Detects `mod_perl` by testing `$ENV{MOD_PERL}`
- Forwards every `connect()` request to the `Apache::DBI` module.

```
if ( $dbh->ping and
    matched( $host, $username, $password, $arguments ) ) {
    return $dbh;
} else {
    connect() && cache();
}
```

- There is no need to delete the `disconnect()` statements from your code.

When should this module be used?

- **DO** use it if you use one connection (or a few) for all your users.
- **DO NOT** use it if you are opening a special connection for each of your users.
- Why: Each connection will stay persistent.
- $\text{\#conn} = \text{\#processes} * \text{\#users}$
- e.g. 20 procs * 100 users = 2000 connections
- The number of connections is limited by an SQL engine

- If you have both kinds of apps
- Run two Apache/mod_perl servers:
- One which uses `Apache::DBI`
- And one which does not.

3.1.2 Configuration

- Install the module
- Add to *httpd.conf*:

PerlModule Apache::DBI

- It is important to load this module before any other Apache*DBI module and before the DBI module itself!

3.1.3 Preopening DBI connections

- Open connection at a child startup:

```
Apache::DBI->connect_on_init
( "DBI:mysql:mysql::myserver" ,
  "username" ,
  "passwd" ,
  {
    PrintError => 1, # warn() on errors
    RaiseError => 0, # don't die on error
    AutoCommit => 1, # commit executes immediately
  }
);
```

- Warning: 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

- Enable Debug mode in the startup script if something is not working

```
$Apache::DBI::DEBUG = 2;
```

- After setting the DEBUG level view the *error_log* file

3.1.5 Opening connections with different parameters

- `connect()` method should use **identical** args to get the connection reused
- If one script sets `LongReadLen` and another one does not
Apache::DBI will make two different connections.
- So for example instead of having a maximum of 40 open connections, you get 80.
- Solution: modify the handle immediately after you get it from the cache.

```
$dbh->{LongReadLen} = 1;
```

- Always initiate connections using the same parameters and set LongReadLen (or whatever) afterwards.

3.1.6 Caching `prepare()` Statements

- Replace `prepare()` with `prepare_cached()`.

Pros:

- Makes sure that you have a good statement handle
- and you will get some caching benefit.

Cons:

- You are going to pay for DBI to parse your SQL
- and do a cache lookup every time you call `prepare_cached()`.

```
;0
```


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

The goal: **User Experience**

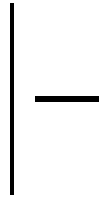
- There are many factors which affect Web site usability
- But speed is one of the most important.

Which speed do we measure?

- Start: link has been clicked
- Finish: the resulting page has been rendered
- The rest is of a little interest to end user

Take into account more than just the code

- NIC and Network: a packet travels from a client and a server and backwards
- Machine's hardware: Each component can be a bottleneck
- Ultrafast webserver but slow modem connection defeats it all -- a need for a proxy server
- Of course the code itself



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

Tools to measure performance

- Benchmarking Perl Code
- Benchmarking Response Time

4.3.1 *Benchmarking Perl Code*

- The Benchmark module
- The Time::HiRes module where you need better time precision (<10msec).

- `Benchmark.pm`:

```
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)
```

- `Time::HiRes:`

```
use Time::HiRes qw(gettimeofday tv_interval);
sub sub_that_takes_a_teeny_bit_of_time{1+1;}

my $start_time = [ gettimeofday ];
sub 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

Goals:

- Generate parallel requests,
- Process the responses and
- Print the results of the test.

Options:

- Re-use the existing stuff
- Develop your own.

4.3.2.1 ApacheBench

- ApacheBench (ab) comes bundled with Apache source distribution.
- Shows Requests/Sec capability of Apache with your code
- An example:
- Simulate 10 concurrent users.
- Each simulated user makes 10 requests.

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

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 httpperf

- httpperf was written by David Mosberger.

```
% httpperf --server hostname --port 80 --uri /test.html \  
--rate 150 --num-conn 27000 --num-call 1 --timeout 5
```

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

- ...more

```
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::::UserAgent

- Write your own tool with LWP::::UserAgent.
- This is a slightly adjusted code originally written by Michael Schilli
- It accepts more than one url to test
- The code is in your handouts

- **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
```

4.4 Choosing MaxClients

- `MaxClients` sets the limit on the number of simultaneous requests that can be supported.
- We want it to be as small as possible

`MaxClients` = ----- **Total RAM Dedicated to the Webserver**
MAX child's process size

- An example:

400MB / 10MB = 40 (clients)

What if there are more concurrent requests than servers?

- 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
```

- Connections can be queued through the `ListenBackLog` directive.
- 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.

- Try not to reach this condition

Real memory use

- Your children can share memory between them when the OS supports that.
- You must take action to allow the sharing to happen.
- Code should be preloaded at the server startup
- You can raise `MaxClients` when you get memory shared

```

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

```

which is:

```

MaxClients = Total_RAM - Shared_RAM_per_Child
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
```

- **Conclusion: With sharing in place you can have 64% more servers without adding 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!

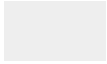
4.5 KeepAlive

- If your mod_perl server's *httpd.conf* includes:

```
KeepAlive On
MaxKeepAliveRequests 100
KeepAliveTimeout 15
```

- You have a real performance penalty,
- The process will wait for `KeepAliveTimeout` seconds before closing the connection
- With this configuration you will need many more concurrent processes on a server with high traffic.

- Set it Off with:



KeepAlive Off

- the other two directives don't matter if `KeepAlive` is Off.

When you want it Enabled?

- A client requests more than one object from your server for a single HTML page.
- You save the HTTP connection overhead for all requests but the first one.
- Example: a page with 10 ad banners
- A server will work more effectively if a single process serves them all during a single connection.
- Your client will see a slightly slower response, though

- SSL connections benefit the most from KeepAlive in case you didn't configure the server to cache session ids.

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.
- Examples:
- the `$ENV{QUERY_STRING}` environment variable is initialized with the contents of `Apache::args()`
- The value returned by `Apache::server_hostname()` is put into `$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**.

- You can still set environment variables.

```
<Location /perl>  
    SetHandler perl-script  
    PerlHandler +Apache::RegistryNG  
  
    PerlSetupEnv Off  
    PerlSetEnv TEST hi  
  
    Options +ExecCGI  
</Location>
```

- and you issue a request for this script:

```
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'
};
```

4.7 Reducing the Number of `stat()` Calls Made by Apache

- Many `stat()` calls are made when the request is being processed.
- Use *truss* or *strace*
- For example: having `DocRoot` set to `/home/httpd/docs`
- and fetching `http://localhost/perl-status (Apache::Status)`

```
[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]
```

Another example:

- Assume the request to */news/perl/mod_perl/summary*
- The URI is completely virtual -- there is no corresponding dirs.
- It will generate 5 `stat()` calls! before the `DocumentRoot` is found.

```
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/news", 0xbffff744) = -1
ENOENT (No such file or directory)
stat("/home/httpd/docs", 0xbffff744) = -1
ENOENT (No such file or directory)
stat("/home/httpd/docs",
      {st_mode=S_IFDIR|0755, st_size=1024, ...}) = 0
```

- Blame the default installed `TransHandler` for this inefficiency.
- Solution: write your own `TransHandler` to return OK immediately when this URI has been seen.
- You cannot put `TransHandler` setting inside the `<Location>`
- But you can do that for a virtual host.

```
<VirtualHost 10.10.10.10:80>  
    ...  
    PerlTransHandler Apache::OK  
    ...  
</VirtualHost>
```

- **no more `stat()`'s for this virtual host!**

- Watching your server under `strace/truss` to reveal more performance hits.
- If you don't have `.htaccess` file, Apache will look for it in the parent directories and add many `open()` calls.
- Use the `AllowOverride None` directive to avoid the lookup.
- Your handouts carry an example of a more complicated `TransHandler` that overrides the default one and reduces the number of the `stat()` calls from many to only one.

4.8 Cached stat() Calls by Perl

- `stat()` (or its variations `-M`, `-A`, `-C`, etc) caches the information.
- Reuse this info by accessing the `_` variable
- So instead of:

```
print "OK\n" if -e $filename and -r $filename;  
my $mod_time = ( -M $filename ) * 24 * 60 * 60;
```

- use the more efficient:

```
print "OK\n" if -e $filename and -r _;  
my $mod_time = ( -M _ ) * 24 * 60 * 60;
```

- `Two stat() syscalls saved!`

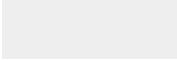
4.8.1 *Be carefull with symbolic links*

- Apache::Registry caches the scripts based on their URI.
- Be ware of symlinks or you will get the same code cached twice

```
% 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.
- The same code will be stored twice in the memory

Detection:

- Use the `/perl-status` (Apache::Status) handler
- `http://localhost/perl-status?rgysubs` you would see:


```
Apache::ROOT::perl::news::news_2ep1  
Apache::ROOT::perl::news_2ep1
```
- run the server in single mode (`httpd -X`) to debug.

4.9 Limiting the Size of the Processes

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

- `startup.pl`:

```
use Apache::SizeLimit; # 10MB
$Apache::SizeLimit::MAX_PROCESS_SIZE = 10000;
```

- `httpd.conf`:

PerlFixupHandler Apache::SizeLimit

- No need for `MaxRequestsPerChild` anymore!

4.10 Sharing Memory

- Saving memory by sharing it between child processes.
- It's possible only when preloading code at server startup
- During a child process' life, its memory pages becomes unshared
- This process is called **Copy-On-Write**
- Perl doesn't have strong typed variables.
- No clear separation between DATA and CODE memory pages.
- That's why the **Copy-On-Write** effect hits almost at random.

- 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.
- Try using `MaxRequestsPerChild` to balance the memory that stays shared against the time
- You should do some measurements and you might see if this really makes a difference and what a reasonable number might be.
- The newly started child process has all of its memory shared.

- `MaxRequestsPerChild` doesn't have to be big (10000?)
- Having a child serving 300 requests on precompiled code is already a huge speedup

4.11 How Shared My Memory Is

- How much shared memory do you have?
- You can see it by either using the memory utils that comes with your system like `top(1)` and `ps(1)`.
- 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";
```

4.12 Keeping the Shared Memory Limit

- `Apache::GTopLimit` module
- like `Apache::SizeLimit` (max memory limitation)
- plus shared memory limitation
- In *httpd.conf*:

■ `PerlFixupHandler Apache::GTopLimit`

- Configuration: *startup.pl*:

```
use Apache::GTopLimit;

# Control the life based on memory size
$Apache::GTopLimit::MAX_PROCESS_SIZE = 10000; # 10MB

# Control the life based on shared memory size
$Apache::GTopLimit::MIN_PROCESS_SHARED_SIZE = 4000; # 4MB

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

4.13 Preload Perl modules at server startup

- Use the `PerlRequire` and `PerlModule` directives to preload commonly used modules such as `CGI.pm`, `DBI...`

```
PerlModule CGI;  
PerlModule DBI;
```

- Do the same in Perl from *startup.pl*:

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

- and require the IPreload Registry Scripts

- `Apache::RegistryLoader` compiles `Apache::Registry` scripts at server startup.
- precompile the scripts just like modules, and save memory

```
use Apache::RegistryLoader ();  
Apache::RegistryLoader->new->handler($url);
```

- Handouts: the code to recursively load of all scripts

4.14 Some numbers: Initializing DBI.pm

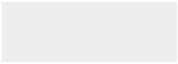
- Let's see how one can calculate the actual improvements of modules preloading practice.
- We will use a very widely used module: DBI
- We will try to initialize DBI with the MySQL driver
DBD : :mysql
- And try to see how it minimizes memory use after forking the child processes.

- 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
```

- or `httpd -X` (single server mode)

- 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()`, it can't be installed it'll `die()`.

option 3

- Preload MySQL driver module:

```
use DBD::mysql;
```

option 4

- Use Apache::DBI->connect_on_init()

- 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";
```

- The `Apache::Registry` test script that was used for testing is in your handouts
- The script opens a connection to the database `'test'`
- issues a query to learn what tables the databases has.
- and prints the memory usage
- The server was restarted before each new test.

- The results 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 requests showed the same results):

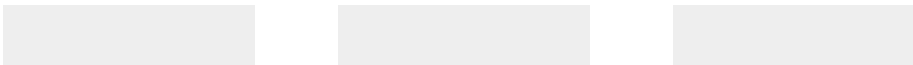
Version	Size	Shared	Diff	Test type
1	3469312	2609152	860160	install_driver
2	3481600	2605056	876544	install_driver & connect_on_init
3	3469312	2588672	880640	preload_driver
4	3477504	2482176	995328	nothing added
5	3481600	2469888	1011712	connect_on_init

Conclusions:

- Only after a second reload we get the final memory footprint for a specific request in question.
- The test with preinstalled driver wins, since it allows to share more memory.
- Since we almost always want to use `connect_on_init()` we will go with option number 2.

- Given that we have 256M of memory dedicated to mod_perl processes

- The good old equation:


$$\begin{array}{r} \text{N_of Procs} = \text{RAM - largest_shared_size} \\ \text{Diff} \\ \text{(ver 1) N} = 268435456 - 2609152 = 309 \\ \text{860160} \\ \text{(ver 5) N} = 268435456 - 2469888 = 262 \\ \text{1011712} \end{array}$$

- So you can tell the difference
- 17% more child processes in the first version

4.15 Limiting the Resources Used by `httpd` Children

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

```
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.

- Refer to the `setrlimit` man page on your OS to find the supported resource names.

4.16 Upload/Download of Big Files

- Don't use mod_perl servers for big file transfers!
- Upload/Download might take minutes
- The second saved by mod_perl gives nothing.
- The server stays tied and cannot do other more important work.
- Solution: Use plain apache server and mod_cgi.

- Assumes that no `mod_perl` functionality is required (e.g. `auth`)

4.17 Global vs Fully Qualified Variables

- Stay away from global variables when possible.
- Sometimes you must have them...
- e.g.: `@ISA` or `$VERSION` variables (or fully qualified `@MyModule::ISA`).
- 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.

- 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";
```

4.18 Forking or Executing subprocesses from mod_perl

- Process' forking might be expensive on some systems.
- To start a long running process (e.g. sending emails to many users: No SPAM please!) and let the parent continue:
- spawn a sub-process,
- hand it the information it needs to do the task,
- and have it detach (close STD* + setsid())

```
$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);  
close STDIN;  
close STDOUT;  
close STDERR;  
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.
- Shell limitation on the size of `@params`.
- Also, the communication is only one way.
- For fast execution of the postprocess code use `PerlCleanupHandler`
- But you will keep the process busy, which is not a good idea..

Forking example:

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

- Parent immediately continues with the code that comes up after the fork
- Child executes `system("echo Hi")` and then terminates itself.

- **!!!: use CORE::exit (exit) == Apache::exit unde Registry and friends)**

...now the gory details:

4.18.1 Freeing the Parent Process

- Closing all the inherited communication pipes opened by the parent.
- allow the parent to complete the request and free itself for serving other requests.
- 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.
- If this socket is not freed, the server cannot be restarted:

```
[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::cleanup_for_exec() takes care of it:

```
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);
}
```

4.18.2 *Detaching the Forked Process*

- The parent cannot continue before the child detaches.
- If the parent is restarted, the child process gets killed

```
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: $!";
    ...
}
```

- The spawned child process has a life of its own

4.18.3 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.
- **Ghostbuster**: It will be killed only when you stop the `httpd` process that spawned it!
- `top(1)/ps(1)` mark those processes as `<defunc>`.
- **Zombies counter** in `top(1)` goes up

- These zombie processes can take up system resources and are generally undesirable.

The proper fork is:

```
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:

```
█ $SIG{CHLD} = IGNORE;
```

- All the processes will be collected by the `init` process and prevent from them to become zombies.
- ... however this doesn't work everywhere.

- You cannot localize this setting with `local()`. If you do, it wouldn't take the desired effect.
- The child must close all the pipes to the connection sockets that were opened by the parent process (`STDIN` and `STDOUT`).
- You may need to close and reopen a `STDERR` filehandler

So now the code would look like:

```
$SIG{CHLD} = IGNORE;

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

- Notice that `waitpid()` call has gone

A double fork approach:

```
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.

use a different *SIGCHLD* handler:

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

- Which is useful when you `fork()` more than once process.
- The 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 until `waitpid()` returns a negative number or zero, indicating that no more reapable children remain.

4.18.4 *A Complete Fork Example*

- Your handouts include the complete example covering all the nuances discussed so far.

4.19 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 $n = 1,$

- 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>
};
```

- Now let's go back to the `$|=1` topic.
- Sometimes I still disable buffering, for 2 reasons:
- I use few `print()` calls.
- Displaying the top of the page, for long queries
- This improves the usability of my services

- A better solution:
- - keep the buffering enabled
- - use a Perl API `rfFlush()` call to flush the buffers when wanted.
- So you kill the two birds in one shoot:
- You show some of the data to the user immediately,
- 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 lengthy operation
    for (1..5) {
        sleep 1;
    }
print $q->p("Done!");
```

4.20 Sending plain HTML as a compressed output

- HTML files with JS code are huge!!
- Compressing them will allow us to deliver them to users much faster
- java applets can be compressed into a jar and benefit from a faster download times.
- 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.
- The 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>
```

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

- 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 `Emberl` module, you would write:

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

- See `perlDoc Apache::GzipChain`

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

```
;0
```

