1   Debugging mod_perl

1.1 Description

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

1.2 Warning and Errors Explained

Let's talk first about things that bother most web (and non-web) programmers. *The bothering things* are warning and errors reported by Perl. We are going to learn how to take the best out of both, by turning this obvious to the newbie programmer enemies into our best friends.

1.2.1 Curing The "Internal Server Error"

You have just installed this new CGI script and when you try it out you see the grey screen of death saying "Internal Server Error"... Or even worse you have a script running on a production server for a long time without problems, when the same grey screen starts to show up occasionally for no apparent reason.

How can we find out what the problem is?

First problem:

You have been coding in Perl for years, and whenever an error occurred in the past it was displayed in the same terminal window that you started the script from. But when you work with a webserver there is no terminal to show you the errors, since the server in most cases has no terminal to send the error messages to.

Actually, the error messages don't disappear, they end up in the *error_log* file. It is located in the directory specified by the `ErrorLog` directive in *httpd.conf*. The default setting is generally:

```
ErrorLog /usr/local/apache/logs/error_log
```

So whenever you see *"Internal Server Error"* it's time to look at this file.

First problem solved!

There are cases when errors don't go to the error_log file. For example, some errors go to the httpd process' STDERR. If you haven't redirected httpd's STDERR then the messages are printed to the console (tty, terminal) from which you executed the httpd. This happens when the server didn't get as far as opening the error_log file for writing before it needed to write an error message.

For example, if you have entered a non-existent directory path in your `ErrorLog` directive, the error message will be printed to STDERR. If the error happens when the server executes a `PerlRequire` or `PerlModule` directive you might also see output sent to STDERR.

You are probably wondering where all the errors go when you are running the server in single process mode (`httpd -X`). They go to STDERR. This is because the error logging for all the httpd children is normally done by the parent httpd. When httpd runs in single process mode, it has no parent httpd process

to perform all the logging. The output to the terminal includes all the status messages that normally go to the error_log file.

Finally with a `PerlLogHandler` you can take away from Apache its control of the error logging process for all HTTP transactions. If you do this, then you are responsible for generating and storing the error messages. You can do whatever you like with the information, (including throwing it away -- don't do it!) and, depending on how you implement you `LogHandler`, the `ErrorLog` directive may have no effect. But you can also do something at this handler and then return `DECLINED` status, so the default Apache LogHandler will do the work as usual.

Second problem:

The usefulness of the error message depends to some extent on the programmer's coding style. An uninformative message might not help you to spot and fix the error.

For example, let's take a function which opens a file passed to it as a parameter. It does nothing else with the file. Here's our first version of the code:

```
my $r = shift;
$r->send_http_header('text/plain');

sub open_file{
  my $filename = shift || '';
  die "No filename passed!" unless $filename;

  open FILE, $filename or die;
}

open_file("/tmp/test.txt");
```

Let's assume that `/tmp/test.txt` doesn't exist so the open() will fail to open the file. When we call this script from our browser, the browser returns an *"internal error"* message and we see the following error appended to *error_log*:

```
Died at /home/httpd/perl/test.pl line 9.
```

We can use the hint Perl kindly gave to us to find where in the code the die() was called. However, we still don't know what filename was passed to this subroutine to cause the program termination.

If we have only one function call as in the example above, the task of finding the problematic filename will be trivial. Now let's add two more open_file() function calls and assume that among the three files only *tmp/test2.txt* exists:

```
open_file("/tmp/test.txt");
open_file("/tmp/test2.txt");
open_file("/tmp/test3.txt");
```

When you execute the above call, you will see the same error message twice:

```
Died at /home/httpd/perl/test.pl line 9.
Died at /home/httpd/perl/test.pl line 9.
```

Based on this error message, can you tell what files your program failed to open? Probably not. Let's fix it by passing the name of the file to die():

```
sub open_file{
  my $filename = shift || '';
  die "No filename passed!" unless $filename;
  open FILE, $filename or die "failed to open $filename";
}

open_file("/tmp/test.txt");
```

When we execute the above code, we see:

```
failed to open /tmp/test.txt at /home/httpd/perl/test.pl line 9.
```

which makes a big difference.

By the way, if you append a newline to the end of the message you pass to die(), Perl won't report the line number the error has happened at, so if you code:

```
open FILE, $filename or die "failed to open a file\n";
```

The error message will be:

```
failed to open a file
```

Which gives you very little to go on. It's very hard to debug with such uninformative error messages.

The warn() function, a kinder sister of die(), which logs the message but doesn't cause program termination, behaves in the same way. If you add a newline to the end of the message, the line number warn() was called at won't be logged, otherwise it will.

You might want to use warn() instead of die() if the failure isn't critical. Consider the following code:

```
if(open FILE, $filename){
  # do something with file
} else {
  warn "failed to open $filename";
}
# more code here...
```

Now we've improved our code, by reporting the names of the problematic files, but we still don't know the reason for the failure. Let's try to improve the warn() example. The -r operator tests whether the file is readable:

```
  if(-r $filename){
    open FILE, $filename;
    # do something with file
  } else {
    warn "Couldn't open $filename - doesn't exist or is not readable";
  }
```

Now if we cannot read the file we do not even try to open it. But we still see a warning in error_log:

```
  Couldn't open /tmp/test.txt - doesn't exist or is not readable
  at /home/httpd/perl/test.pl line 9.
```

The warning tells us the reason for the failure, so we don't have to go to the code and check what it was trying to do with the file.

It could be quite a coding overhead to explain all the possible failure reasons that way, but why reinvent the wheel? We already have the reason for the failure stored in the $! variable. Let's go back to the open_file() function:

```
  sub open_file{
    my $filename = shift || '';
    die "No filename passed!" unless $filename;
    open FILE, $filename or die "failed to open $filename: $!";
  }

  open_file("/tmp/test.txt");
```

This time, if open() fails we see:

```
  failed to open /tmp/test.txt: No such file or directory
  at /home/httpd/perl/test.pl line 9.
```

Now we have all the information we need to debug these problems: we know what line of code triggered die(), we know what file we were trying to open, and last but not least we know the reason, given to us through Perl's $! variable.

Now let's create the file */tmp/test.txt*.

```
  % touch /tmp/test.txt
```

When we execute the latest version of the code, we see:

```
  failed to open /tmp/test.txt: Permission denied
  at /home/httpd/perl/test.pl line 9.
```

Here we see a different reason: we created a file that doesn't belong to the user which the server runs as (usually *nobody*). It does not have permission to read the file.

Now you can see that it's much easier to debug your code if you validate the return values of the system calls, and properly code arguments to die() and warn() calls. The open() function is just one of the many system calls perl provides to your convenience.

So now you can code and debug CGI scripts and modules as easily as if they were plain Perl scripts that you execute from a shell.

Second problem solved!

1.2.2 Helping error_log to Help Us

It's a good idea to keep it open all the time in a dedicated terminal with the help of *tail -f* or *less -S*, whichever you prefer (the latter allows you to page around the file, search etc.)

```
% tail -f /usr/local/apache/logs/error_log
```

or

```
% less -S /usr/local/apache/logs/error_log
```

So you will see all the errors and warning as they happen.

Another tip is to create a shell *alias*, to make it easier to execute the above command. In tcsh you would do something like this:

```
% alias err "tail -f /usr/local/apache/logs/error_log"
```

For bash users the command is:

```
% alias err='tail -f /var/log/apache/error_log'
```

and from now on in the shell you set the alias in, executing

```
% err
```

will call *tail -f /usr/local/apache/logs/error_log*. Since you want this alias to be available to you all the time, you should put it into your *.tcshrc* file or its equivalent. For *bash* users this is *.bashrc*, or you can put it in */etc/profile* for use by all users.

If you cannot access your *error_log* file because you are unable to telnet to your machine (generally the case when an ISP provides user CGI support but no telnet access), you might want to use a CGI script I wrote to fetch the latest lines from the file (with a bonus of colored output for easier reading). You might need to ask your ISP to install this script for general use. See Watching the error_log file without telneting to the server .

1.2.3 The Importance of Warnings

Just like errors, Perl's mandatory warnings go to the *error_log* file, if the they are enabled. Of course you have enabled them in your development server, haven't you?

The code you write lives a dual life. In the first life it's being written, tested, debugged, improved, tested, debugged, rewritten, tested, debugged. In the second life it's *just* used.

A significant part of the script's first life is spent on the developer's machine. The other part is spent on the production server where the creature is supposed to be perfect.

So when you develop the code you want all the help in the world to help you spot possible problems, and that's where enabling warnings is a must. Whenever you see an error or warning in the *error_log*, you want to get rid of it. That's very important.

Why?

- If there are warnings, your code is not clean. If they are waved away, expect them to come back on the production server in the form of errors, when it's too late.

- If each invocation of a script generates more than about five lines of warnings, it will be very hard to catch real problems. You just can't see them among all the other warnings which you used to think were unimportant.

On the other hand, on a production server, you really *want* to turn warnings off. And there are good reasons for that:

- There is no added value in having the same warning showing up, again and again, triggered by thousands of script invocations. If your code isn't very clean and generates even a single warning per script invocation, on the heavily loaded server you will end up with a huge *error_log* file in a short time.

  The warning elimination phase is supposed to be a part of the development process, and should be done before the code goes live.

- In any Perl script, not just under mod_perl, enabling runtime warnings has a performance impact.

mod_perl gives you a very simple solution to this warnings saga, don't enable warnings in the scripts unless you really have to. Let mod_perl control this mode globally. All you need to do is put the directive

```
PerlWarn On
```

in *httpd.conf* on your development machine and the directive

```
PerlWarn Off
```

on the live box. Here is a complete description on how to manipulate warning modes under mod_perl.

If there is a piece of code that generates warnings and you want to disable them only in this code, you can do that too. The Perl special variable `$^W` allows you dynamically to turn on and off warnings mode. So just put the code into a block, and disable the warnings in the scope of this block. The original value of `$^W` will be restored upon exit from the block.

```
{
 local $^W=0;
   # some code that generates innocuous warnings
}
```

Unless you have a really good reason, for your own sake the advice is *avoid this technique*.

Don't forget the `local()` operand! If you do, setting `$^W` will affect **all** the requests handled by the Apache child that changed this variable. And for **all** the scripts it executes, not just the one which changed `$^W`!

The `diagnostics` pragma can shed more light on errors and warnings, as you will see in a moment.

1.2.3.1 diagnostics pragma

This module extends the terse diagnostics normally emitted by both the Perl compiler and the Perl interpreter, augmenting them with the more verbose and endearing descriptions found in the `perldiag` manpage. Like the other pragmata, it affects the compilation phase of your scripts as well as the execution phase.

To use in your program as a pragma, merely invoke

```
use diagnostics;
```

at or near the start of your program. This also turns on `-w` mode.

This pragma is especially useful when you are new to perl, and want a better explanation of the errors and warnings. It's also helpful when you encounter some warning you've never seen before, e.g. when a new warning has been introduced in an upgraded version of Perl.

You may not want to leave `diagnostics` mode On for your production server. For each warning, `diagnostics` mode generates ten times more output than warnings mode. If your code generates warnings, with the `diagnostics` pragma you will use disk space much faster.

`diagnostics` mode adds a large performance overhead in comparison with just having warnings mode On. You can see the benchmark results in the section 'Code Profiling Techniques'.

1.3 Handling the 'User pressed Stop button' case

When a user presses a **STOP** or **RELOAD** button, the current socket connection goes broken (aborted). It would be nice if Apache could always immediately detect this event. Unfortunately there is no way to tell whether the connection is still valid unless an attempt to read from or write to connection is made.

Unfortunately the detection technique we are going to present doesn't work if the connection to the back-end mod_perl server is coming from the front-end mod_proxy, as the latter doesn't break the connection to the back-end when user has aborted the connection.

If the reading of the request's data is completed and the code does processing without writing anything back to the client the broken connection won't be noticed. When an attempt is made to send at least one character to the client, the broken connection would be noticed and the `SIGPIPE` signal (Broken pipe) would be sent to the process. The program could then halt its execution and perform all the cleanup stuff it has to do.

Prior to Apache version 1.3.6, `SIGPIPE` was handled by Apache. Currently Apache is not handling SIGPIPE anymore and mod_perl takes care of it.

Under mod_perl, `$r->print` (or just print()) returns a *true* value on success, a *false* value on failure. The latter usually happens when the connection is broken.

If you want a similar to the old `SIGPIPE` behaviour (as it was before Apache version 1.3.6), add the following configuration directive:

```
PerlFixupHandler Apache::SIG
```

When Apache's `SIGPIPE` handler is used, Perl may be left in the middle of it's eval context, causing bizarre errors during subsequent requests are handled by that child. When `Apache::SIG` is used, it installs a different `SIGPIPE` handler which rewinds the context to make sure Perl is back to normal state, preventing these bizarre errors.

But in general case, you don't need to use the above setting.

If you use this setting and you would like to log when a request was canceled by a SIGPIPE in your Apache *access_log*, you must define a custom `LogFormat` in your *httpd.conf*, like so:

```
PerlFixupHandler Apache::SIG
LogFormat "%h %l %u %t \"%r\" %s %b %{SIGPIPE}e"
```

If the server has noticed that the request was canceled via a `SIGPIPE`, then the log line will end with `1`, otherwise it will just be a dash. e.g.:

```
127.0.0.1 - - [09/Jan/2001:10:27:15 +0100]
"GET /perl/stopping_detector.pl HTTP/1.0" 200 16 1
127.0.0.1 - - [09/Jan/2001:10:28:18 +0100]
"GET /perl/test.pl HTTP/1.0"              200 10 -
```

1.3.1 Detecting Aborted Connections

Let's use the knowledge we have acquired to trace the execution of the code and watch all the events as they happen.

Let's take a little script that obviously *"hangs"* the server process:

```
stopping_detector.pl
-------------------
my $r = shift;
$r->send_http_header('text/plain');

print "PID = $$\n";
$r->rflush;

while(1){
  $i++;
  sleep 1;
}
```

The script gets a request object $r by shift()ing it from the @_ argument list passed by the handler() subroutine. (This magic is done by Apache::Registry). Then the script sends a Content-type header, telling the client that we are going to send a plain text as a response.

Next the script prints out a single line telling us the id of the process that handles this request, which we need to know in order to run the tracing utility. Then we flush Apache's buffer. (If we don't flush the buffer we will never see this short information printed. That's because our output is shorter than the buffer size and the script intentionally hangs, so the buffer won't be auto-flushed as the script hangs at the end.)

Then we enter an infinite while(1) loop, which just increments a dummy variable and sleeps for a second.

Running strace -p PID, where *PID* is the process ID as printed to the browser, we see the following output printed every second:

```
SYS_175(0, 0xbffff41c, 0xbffff39c, 0x8, 0) = 0
SYS_174(0x11, 0, 0xbffff1a0, 0x8, 0x11)    = 0
SYS_175(0x2, 0xbffff39c, 0, 0x8, 0x2)      = 0
nanosleep(0xbffff308, 0xbffff308,
          0x401a61b4, 0xbffff308, 0xbffff41c) = 0
time([941281947])                          = 941281947
time([941281947])                          = 941281947
```

Let's leave strace running and press the **STOP** button. Did anything change? No, the same system calls trace is printed every second. Which means that Apache didn't detect the broken connection.

Now we are going to write the \0 (NULL) character to the client in attempt to detect the broken connection as close as possible to the time the **Stop** button is pressed at. Therefore we modify the loop code in the following way:

```
while(1){
  $r->print("\0");
  last if $r->connection->aborted;
  $i++;
  sleep 1;
}
```

We add a print() statement to print a NULL character and then we check whether the connection was aborted with help of the $r->connection->aborted method. If the connection is broken, we break out of the loop.

We run this script and strace on it as before, but we see that it still doesn't work. The trouble is we aren't flushing the buffer, which leaves the characters in the buffer and they won't be printed before the buffer will get full and will be autoflushed. Since we want to attempt to write to the connection pipe all the time, after printing the NULL, we add $r->rflush(). Here is a new version of the code:

```
stopping_detector2.pl
---------------------
my $r = shift;
$r->send_http_header('text/plain');

print "PID = $$\n";
```

```
$r->rflush;

while(1){
  $r->print("\0");
  $r->rflush;

  last if $r->connection->aborted;

  $i++;
  sleep 1;
}
```

After starting the `strace` utility on the running process as we did before and pressing the **Stop** button, we have seen the following output.

```
SYS_175(0, 0xbffff41c, 0xbffff39c, 0x8, 0) = 0
SYS_174(0x11, 0, 0xbffff1a0, 0x8, 0x11) = 0
SYS_175(0x2, 0xbffff39c, 0, 0x8, 0x2)   = 0
nanosleep(0xbffff308, 0xbffff308, 0x401a61b4, 0xbffff308, 0xbffff41c) = 0
time([941284358])                       = 941284358
write(4, "\0", 1)                       = -1 EPIPE (Broken pipe)
--- SIGPIPE (Broken pipe) ---
select(5, [4], NULL, NULL, {0, 0})      = 1 (in [4], left {0, 0})
time(NULL)                              = 941284358
write(17, "127.0.0.1 - - [30/Oct/1999:13:52"..., 81) = 81
gettimeofday({941284359, 39113}, NULL)  = 0
times({tms_utime=9, tms_stime=8, tms_cutime=0, tms_cstime=0}) = 41551400
close(4)                                = 0
SYS_174(0xa, 0xbffff4e0, 0xbffff454, 0x8, 0xa) = 0
SYS_174(0xe, 0xbffff46c, 0xbffff3e0, 0x8, 0xe) = 0
fcntl(18, F_SETLKW, {type=F_WRLCK, whence=SEEK_SET, start=0, len=0}
```

Apache detects the broken pipe as you see from this snippet:

```
write(4, "\0", 1)                       = -1 EPIPE (Broken pipe)
--- SIGPIPE (Broken pipe) ---
```

Then it stops the script and does all the cleanup work, like access logging:

```
write(17, "127.0.0.1 - - [30/Oct/1999:13:52"..., 81) = 81
```

where 17 is a file descriptor of the opened *access_log* file

1.3.2 The Importance of Cleanup Code

Cleanup code is a critical issue with aborted scripts.

What happens to locked resources if there are any? Will they be freed or not? If not, scripts using these resources and the same locking scheme will hang, waiting for them to be freed.

First let's go one step back and recall what are the problems and solutions for this issue under mod_cgi.

Under mod_cgi the resource locking issue is a problem only if you happened to create external lock files and use them for lock indication, instead of using flock(). If the script running under mod_cgi is aborted between the lock and the unlock code, and you didn't bother to write cleanup code to remove old dead locks then you are in big trouble.

The solution is to use an END block to place the cleanup code in:

```
END {
   # some code that ensures that locks are removed
}
```

When the script is aborted, Apache will run the END blocks.

If you use flock() things are much simpler, since all opened files will be closed when the script exits. When the file is closed, the lock is removed as well--all the locked resources get freed. There are systems where flock(2) is unavailable, and for those you can use Perl's emulation of this function.

With mod_perl things can be more complex when you use global variables as a filehandlers. Because the processes don't exit after processing a request, files won't be closed unless you explicitly close() them or reopen with the open() call, which first closes a file. Let's see what problems we might encounter, and possible solutions for them.

1.3.2.1 Critical Section

First we want to make a little detour to discuss the *"critical section"* issue.

Let's start with a resource locking scheme. A schematic representation of a proper locking technique is as follows:

```
1. lock a resource
   <critical section starts>
2. do something with the resource
   <critical section ends>
3. unlock the resource
```

If the locking is exclusive, only one process can hold the resource at any given time, which means that all the other processes will have to wait, therefore the code between the locking and unlocking functions can become a service bottleneck. That's why this code section is called critical and once started it should be finished as soon as possible.

Even if you use a shared locking scheme, where many processes are allowed to concurrently access the resource, if there are processes that sometimes want to get an exclusive lock it's also important to keep the critical section as short as possible.

The next example uses a shared lock, but has a poorly-designed critical section:

```
critical_section_sh.pl
-------------------
use Fcntl qw(:flock);
use Symbol;
my $fh = gensym;
```

```
open $fh, "/tmp/foo" or die $!;
flock $fh, LOCK_SH;
  # start critical section

seek $fh, 0, 0;
my @lines = <$fh>;
for(@lines){
  print if /foo/;
}

  # end critical section
close $fh; # close unlocks the file
```

The code opens the file for reading, locks and rewinds it to the beginning, reads all the lines from the file and prints out the lines that contain the string *'foo'*.

The `gensym()` function imported by the `Symbol` module creates an anonymous glob and returns a reference to it. Such a glob reference can be used as a file or directory handle. and therefore allows using lexically scoped variables as filehandlers. `Fcntl` imports into the script's namespace file locking symbols like: `LOCK_SH`, `LOCK_EX` and more. Refer to the `Fcntl` manpage for more information.

If the file the script reads is big, it'd take a relatively long time for this code to complete. All this time the file remains open and locked. While it's other processes may access this file for reading (shared lock), the process that wants to modify the file (which requires an acquisition of the exclusive lock), will be blocked waiting for this section to complete.

We can optimize the critical section this way:

Once the file has been read, we have all the information we need from it. In order to make the example simpler we've chosen to just print out the matching lines. In reality the code might be much longer.

We don't need the file to be open while the loop executes, because we don't access it inside the loop. If we close the file before we start the loop, we will allow other processes to have an exclusive access to the file if they need it, instead of blocking them for no reason.

In the following corrected version of the previous example, we only read the content of the file during the critical section and process it afterwards, without creating a possible bottleneck.

```
critical_section_sh2.pl
-------------------
use Fcntl qw(:flock);
use Symbol;
my $fh = gensym;

open $fh, "/tmp/foo" or die $!;
flock $fh, LOCK_SH;
  # start critical section

seek $fh, 0, 0;
my @lines = <$fh>;

  # end critical section
```

```
close $fh; # close unlocks the file

for(@lines){
  print if /foo/;
}
```

Here is another similar example, but now it uses an exclusive lock. The script reads in a file and writes it back, adding a number of new text lines to the head of the file.

```
critical_section_ex.pl
--------------------
use Fcntl qw(:flock);
use Symbol;
my $fh = gensym;

open $fh, "+>>/tmp/foo" or die $!;
flock $fh, LOCK_EX;

  # start critical section
seek $fh, 0, 0;
my @add_lines =
  (
   qq{Complete documentation for Perl, including FAQ lists,\n},
   qq{should be found on this system using 'man perl' or\n},
   qq{'perldoc perl'. If you have access to the Internet, point\n},
   qq{your browser at http://www.perl.com/, the Perl Home Page.\n},
  );

my @lines = (@add_lines, <$fh>);
seek $fh, 0, 0;
truncate $fh, 0;
print $fh @lines;
  # end critical section

close $fh; # close unlocks the file
```

Since we want to read the file, modify and write it back, without anyone else changing it on the way, we open it for read and write with the help of +>> and lock it with an exclusive lock. You cannot safely accomplish this task by opening the file first for read and then reopening for write, since another process might change the file between the two events. (You could get away with +< as well, please refer to the *perlfunc* manpage for more information about the open() function.)

Next, the code prepares the lines of text it wants to add to the head of the file, and assigns them and the content of the file to the @lines array. Now we have our data ready to be written back to the file, so we seek() to the start of the file and truncate() it to zero size. In our example the file always grows, so in this case there is no need to truncate it, but if there was a chance that the file might shrink then truncating would be necessary. However it's good practice to always use truncate(), as you never know what changes your code might undergo in the future. The truncate() operation does not carry any significant performance penalty. Finally we write the data back to the file and close it, which unlocks it as well.

Did you notice that we created the text lines to be added as close to the place of usage as possible? This complies with good *"locality of code"* style, but it makes the critical section longer. In such cases you should sacrifice style, in order to make the critical section as short as possible. An improved version of this

script with a shorter critical section looks like this:

```
critical_section_ex2.pl
-------------------
use Fcntl qw(:flock);
use Symbol;

my @lines =
  (
   qq{Complete documentation for Perl, including FAQ lists,\n},
   qq{should be found on this system using 'man perl' or\n},
   qq{'perldoc perl'. If you have access to the Internet, point\n},
   qq{your browser at http://www.perl.com/, the Perl Home Page.\n},
  );

my $fh = gensym;
open $fh, "+>>/tmp/foo" or die $!;
flock $fh, LOCK_EX;
  # start critical section

seek $fh, 0, 0;
push @lines, <$fh>;

seek $fh, 0, 0;
truncate $fh, 0;
print $fh @lines;

  # end critical section
close $fh; # close unlocks the file
```

There are two important differences. First, we prepare the text lines to be added *before* the file is locked. Second, instead of creating a new array and copying lines from one array to another, we append the file directly to the @lines array.

1.3.2.2 Safe Resource Locking and Cleanup Code

Let's get back to the main issue of this section, which is safe resource locking.

Unless you use the Apache::PerlRun handler that does the cleanup for you, if you don't make a habit of closing all the files that you open--in some cases you will encounter lots of problems. If you open a file but don't close it, you may have file descriptor leakage. Since the number of file descriptors available to you is finite, at some point you may run out of them and your service will fail. This is bad, but you can live with it until you run out of file descriptors (which will happen much faster on a heavily used server).

You can use system utilities to observe the opened and locked files, as well as the processes that has opened (and locked) the files. On FreeBSD you would use the fstat(1) utility. On many other UN*X flavors the lsof(1) utility is available.

But this is nothing compared to the trouble you will give yourself if the code terminates and the file stays locked. Any other process requesting a lock on the same file (or resource) will wait indefinitely for it to become unlocked. Since this will not happen until the server reboots, all these processes trying to use this resource will hang.

Here is an example of such a terrible mistake:

```
flock.pl
--------
use Fcntl qw(:flock);
open IN, "+>>filename" or die "$!";
flock IN, LOCK_EX;
   # do something
   # quit without closing and unlocking the file
```

Is this safe code? No - we forgot to close the file. So let's add the close():

```
flock2.pl
---------
use Fcntl qw(:flock);
open IN, "+>>filename" or die "$!";
flock IN, LOCK_EX;
   # do something
close IN;
```

Is it safe code now? Unfortunately it is not. There is a chance that the user may abort the request (for example by pressing his browser's `Stop` or `Reload` buttons) during the critical section. The script will be aborted before it has had a chance to close() the file, which is just as bad as if we forgot to close it.

In fact if the same process will run the same code again, an open() call will close the file first, which will unlock the resource. This is because IN is a global variable. But it's quite possible that the process that created the lock, will not serve the same request for a while, since it would be busy serving other requests. So relying on it to reopen the file is a bad idea.

This problem happens **only** if you use global variables as file handles. The following example has the same problem.

```
flock3.pl
---------
use Fcntl qw(:flock);
use Symbol ();
use vars qw($fh);
$fh = Symbol::gensym();
open $fh, "+>>filename" or die "$!";
flock $fh, LOCK_EX;
   # do something
close $fh;
```

$fh is still a global variable and therefore the code using it suffers from the same problem.

The simplest solution to this problem is to always use lexically scoped variables (created with my ()). Whether script gets aborted before close() is called or you forgot the use close() the lexically scoped variable will always go out of scope and therefore if the file was locked it will be unlocked. Here is a good version of the code:

```
flock4.pl
---------
use Fcntl qw(:flock);
use Symbol ();
my $fh = Symbol::gensym();
open $fh, "+>>filename" or die "$!";
flock $fh, LOCK_EX;
  # do something
close $fh;
```

Please don't conclude from this example that you don't have to close files anymore, since they will be automatically closed for you. It's a bad style and should be avoided.

mod_perl comes with its own implementation of gensym(), so you don't even need to load the Symbol module in order to use this function. In mod_perl this function resides in the `Apache` package. For example:

```
use Apache;
my $fh = Apache::gensym();
open $fh, "+>>filename" or die "$!";
...
```

If you insist on using the file globs, at least make sure that you local()'ize these, and then if the flow of the code is interrupted before close() was called the filehandle will be automatically closed, since the local()'ized variable will go out of the scope. The following example shows that the file is indeed closed even when there is no close():

```
/tmp/io.pl
--------------
#!/usr/bin/perl
# /dev/null so strace output is more readable
open my $fh, ">/dev/null";
select $fh;
$| = 1;
{
  print "enter";
  local *FH;
  open FH, $0;
  print "leave"
}
print "done";
```

This simple script opens the */dev/null* and tells Perl to send all the STDOUT there, which is also made unbuffered. Then the block is created in which the `FH` file glob is localized. Then it's used to open the source code of the script (which resides in `$0`). In order to separate event of entering the block scope and leaving it, the debug print statements are used. Now let's run the script under strace(1), which proves once again to be very useful in the tool bag of the mod_perl programmer:

```
% strace /tmp/io.pl
write(3, "enter", 5)                        = 5
-> open("/tmp/io.pl", O_RDONLY) = 4
fstat(4, {st_mode=S_ISGID|S_ISVTX|0401, st_size=0, ...}) = 0
fcntl(4, F_SETFD, FD_CLOEXEC)        = 0
write(3, "leave", 5)                        = 5
-> close(4)                                  = 0
write(3, "done", 4)                         = 4
```

So you can see that */tmp/io.pl* is actually close()'d.

Under Perl version 5.6 `Symbol.pm`-like functionality is a built-in feature, so you can do:

```
open my $fh, ">/tmp/foo" or die $!;
```

and `$fh` will be automatically vivified as a valid filehandle, so you don't need to use the `Symbol` module anymore, if backward compatibility is not a requirement.

You can also use the `IO::*` modules, such as `IO::File` or `IO::Dir`. These are much bigger than the `Symbol` module, and worth using for files or directories only if you are already using them for the other features which they provide. As a matter of fact, these modules use the `Symbol` module themselves. Here is an example of their usage:

```
use IO::File;
use IO::Dir;
my $fh = IO::File->new(">filename");
my $dh = IO::Dir->new("dirname");
```

If you still have to use global filehandles, there are a few approaches we can take to solving the locking problem.

If you are running under `Apache::Registry` and friends, the `END` block will perform the cleanup work for you. You might use `END` in the same way for scripts running under mod_cgi, or in plain Perl scripts. Just add the cleanup code to this block and you are safe.

For example if you work with dbm files just like with locking it's important to flush the dbm buffers, by calling a sync() method:

```
END{
   # make sure that the DB is flushed
    $dbh->sync();
}
```

Normally the `END` blocks will not be executed after the completion of a request, but only when an Apache child process exits, then if you are writing your own handlers you will need to use the register_cleanup() function to supply cleanup code similar to that used in `END` blocks instead of using `END` blocks.

Under mod_perl, the above will work only for `Apache::Registry` scripts. Otherwise execution of the `END` block will be postponed until the process terminates. If you write a handler in the Perl API use the `register_cleanup()` method instead. It accepts a reference to a subroutine as an argument:

```
$r->register_cleanup(sub { $dbh->sync() });
```

Even better would be to check whether the client connection has been aborted. If you don't check, the cleanup code will always be executed and for normally terminated scripts this may not be what you want:

```
$r->register_cleanup(
  # make sure that the DB is flushed
  sub{
    $dbh->sync() if Apache->request->connection->aborted();
  }
);
```

So in the case of END block usage you would use:

```
END{
  # make sure that the DB is flushed
  $dbh->sync() if Apache->request->connection->aborted();
}
```

Note that if you use register_cleanup() it should be called at the beginning of the script, or as soon as the variables you want to use in this code become available. If you use it at the end of the script, and the script happens to be aborted before this code is reached, there will be no cleanup performed.

For example CGI.pm registers the cleanup subroutine in its new() method:

```
sub new {
  # code snipped
  if ($MOD_PERL) {
      Apache->request->register_cleanup(\&CGI::_reset_globals);
      undef $NPH;
  }
  # more code snipped
}
```

There is another way to register a section of cleanup code for Perl API handlers. You may use Perl-CleanupHandler in the configuration file, like this:

```
<Location /foo>
  SetHandler perl-script
  PerlHandler       Apache::MyModule
  PerlCleanupHandler Apache::MyModule::cleanup()
  Options ExecCGI
</Location>
```

Apache::MyModule::cleanup() performs the cleanup, obviously.

1.4 Handling Server Timeout Cases and Working with $SIG{ALRM}

A similar situation to Pressed Stop button disease happens when the browser times out the connection (is it about 2 minutes?). There are cases when your script is about to perform a very long operation and there is a chance that its duration will be longer than the client's timeout. One example is database interaction, where the DB engine hangs or needs a long time to return the results. If this is the case, use `$SIG{ALRM}` to prevent the timeouts:

```
  $timeout = 10; # seconds
eval {
  local $SIG{ALRM} =
      sub { die "Sorry timed out. Please try again\n" };
  alarm $timeout;
  ... db stuff ...
  alarm 0;
};

die $@ if $@;
```

It was recently discovered that `local $SIG{'ALRM'}` does not restore the original underlying C handler. This was fixed in mod_perl 1.19_01. As a matter of fact none of the `local $SIG{FOO}` signals restores the original C handler - read Debugging Signal Handlers ($SIG{FOO}) for a debug technique and a possible workaround.

1.5 Looking inside the server

Your server is up and running, but something appears to be wrong. You want to see the numbers to tune your code or server configuration. You just want to know what's really going on inside the server.

How do you do it?

There are a few tools that allow you to look inside the server.

1.5.1 Apache::Status -- Embedded Interpreter Status Information

This is a very useful module. It lets you watch what happens to the Perl parts of the server. You can see the size of all subroutines and variables, variable dumps, lexical information, OPcode trees, and more.

You shouldn't use it on production server as it adds quite a bit of overhead for each request.

1.5.1.1 Minimal Configuration

This configuration enables the `Apache::Status` module with its minimum feature set. Add this to *httpd.conf*:

```
<Location /perl-status>
  SetHandler perl-script
  PerlHandler Apache::Status
  order deny,allow
  #deny from all
  #allow from
</Location>
```

If you are going to use `Apache::Status` it's important to put it as the first module in the start-up file, or in *httpd.conf*:

```
# startup.pl
use Apache::Status ();
use Apache::Registry ();
use Apache::DBI ();
```

If you don't put `Apache::Status` before `Apache::DBI`, you won't get the `Apache::DBI` menu entry in the status. For more about `Apache::DBI` see Persistent DB Connections.

1.5.1.2 Extended Configuration

There are several variables which you can use to modify the behaviour of `Apache::Status`.

- **PerlSetVar StatusOptionsAll On**

  This single directive will enable all of the options described below.

- **PerlSetVar StatusDumper On**

  When you are browsing symbol tables, you can view the values of your arrays, hashes and scalars with `Data::Dumper`.

- **PerlSetVar StatusPeek On**

  With this option On and the `Apache::Peek` module installed, functions and variables can be viewed in `Devel::Peek` style.

- **PerlSetVar StatusLexInfo On**

  With this option On and the `B::LexInfo` module installed, subroutine lexical variable information can be viewed.

- **PerlSetVar StatusDeparse On**

  With this option On and `B::Deparse` version 0.59 or higher (included in Perl 5.005_59+), subroutines can be "deparsed".

  Options can be passed to `B::Deparse::new` like so:

  ```
  PerlSetVar StatusDeparseOptions "-p -sC"
  ```

  See the `B::Deparse` manpage for details.

- **PerlSetVar StatusTerse On**

  With this option On, text-based op tree graphs of subroutines can be displayed, thanks to `B::Terse`.

- **PerlSetVar StatusTerseSize On**

  With this option On and the `B::TerseSize` module installed, text-based op tree graphs of subroutines and their size can be displayed. See the `B::TerseSize` docs for more info.

- **PerlSetVar StatusTerseSizeMainSummary On**

  With this option On and the `B::TerseSize` module installed, "Memory Usage" will be added to the `Apache::Status` main menu. This option is disabled by default, as it can be rather cpu intensive to summarize memory usage for the entire server. It is strongly suggested that this option only be used with a development server running in -X mode, as the results will be cached.

  Remember to preload `B::TerseSize` with:

  ```
  PerlModule B::Terse
  ```

- **PerlSetVar StatusGraph**

  When `StatusDumper` (see above) is enabled, another link *"OP Tree Graph"* will be present with the dump if this configuration variable is set to On.

  This requires the B module (part of the Perl compiler kit) and the `B::Graph` module version 0.03 or higher to be installed along with the 'dot' program. Dot is part of the graph visualization toolkit from AT&T: http://www.research.att.com/sw/tools/graphviz/.

  WARNING: Some graphs may produce very large images, and some graphs may produce no image if `B::Graph`'s output is incorrect.

There is more information about `Apache::Status` in its manpage.

1.5.1.3 Usage

Assuming that your mod_perl server listens on port 81, fetch http://www.myserver.com:81/perl-status

```
Embedded Perl version 5.00502 for Apache/1.3.2 (Unix) mod_perl/1.16
process 187138, running since Thu Nov 19 09:50:33 1998
```

Below all the sections are links when you view them through */perl-status*

```
Signal Handlers
Enabled mod_perl Hooks
PerlRequire'd Files
Environment
Perl Section Configuration
Loaded Modules
Perl Configuration
ISA Tree
Inheritance Tree
Compiled Registry Scripts
Symbol Table Dump
```

Let's follow, for example, `PerlRequire`'d Files. We see:

```
PerlRequire                   Location
/home/perl/apache-startup.pl  /home/perl/apache-startup.pl
```

From some menus you can move deeper to peek into the internals of the server, to see the values of the global variables in the packages, to see the cached scripts and modules, and much more. Just click around...

1.5.1.4 Compiled Registry Scripts section seems to be empty.

Sometimes when you fetch */perl-status* and look at the **Compiled Registry Scripts** you see no listing of scripts at all. This is correct: `Apache::Status` shows the registry scripts compiled in the httpd child which is serving your request for */perl-status*. If the child has not yet compiled the script you are asking for, */perl-status* will just show you the main menu.

1.5.2 mod_status

The Status module allows a server administrator to find out how well the server is performing. An HTML page is presented that gives the current server statistics in an easily readable form. If required, given a compatible browser this page can be automatically refreshed. Another page gives a simple machine-readable list of the current server state.

This Apache module is written in C. It is compiled by default, so all you have to do to use it is enable it in your configuration file:

```
<Location /status>
  SetHandler server-status
</Location>
```

For security reasons you will probably want to limit access to it. If you have installed Apache according to the instructions you will find a prepared configuration section in *httpd.conf*: to enable use of the mod_status module, just uncomment it.

```
ExtendedStatus On
<Location /status>
  SetHandler server-status
  order deny,allow
  deny from all
  allow from localhost
</Location>
```

You can now access server statistics by using a Web browser to access the page http://localhost/status (as long as your server recognizes localhost:).

The details given by mod_status are:

- **The number of children serving requests**
- **The number of idle children**
- **The status of each child, the number of requests that child has performed and the total number**

of bytes served by the child
- **A total number of accesses and the total bytes served**
- **The time the server was last started/restarted and how long it has been running for**
- **Averages giving the number of requests per second, the number of bytes served per second and the average number of bytes per request**
- **The current percentage CPU used by each child and in total by Apache**
- **The current hosts and requests being processed**

1.5.3 Apache::VMonitor -- Visual System and Apache Server Monitor

This module is covered in the section "Apache::* Modules"

1.6 Sometimes My Script Works, Sometimes It Does Not

See Sometimes it Works Sometimes it does Not

1.7 Code Debug

When the code doesn't perform as expected, either never or just sometimes, we say that the code needs debugging. There are several levels of debugging complexity.

The basic level is when Perl terminates the program during the compilation phase, before it tries to run the resulting byte-code. This usually happens because there are syntax errors in the code, or perhaps a module is missing. Sometimes it takes quite an effort to solve these problems, since code that uses Apache CORE modules generally won't compile when executed from the shell. We will learn how to solve syntax problems in mod_perl code quite easily.

Once the program compiles and begins to run, there might be logical problems, when the program doesn't do what you thought you had programmed it to do. These are somewhat harder to solve, especially when there is a lot of code to be inspected and reviewed, but it's just a matter of time. Perl can help a lot, for example to locate typos, when we enable warnings. For example, if you wanted to compare two numbers, but you omitted the second '=' character so that you had something like if `$yes = 1` instead of `if $yes == 1`, it warns us about the missing '='.

The next level is when the program does what it's expected to do most of the time, but occasionally misbehaves. Often you find that print() statements or the Perl debugger can help, but inspection of the code generally doesn't. Often it's quite easy to debug with print(), but sometimes typing the debug messages can become very tedious. That's where the Perl debugger comes into its own.

While print() statements always work, running the perl debugger for CGI scripts might be quite a challenge. But with the right knowledge and tools handy the debug process becomes much easier. Unfortunately there is no one easy way to debug your programs, as the debugging depends entirely on your code. It can be a nightmare to debug really complex code, but as your style matures you can learn ways to write simpler code that is easier to debug. You will probably find that when you write simpler clearer code it does not need so much debugging in the first place.

One of the most difficult cases to debug, is when the process just terminates in the middle of processing a request and dumps core. Often when there is a bug the program tries to access a memory area that doesn't belong to it. The operating system halts the process, tidies up and dumps core (it creates a file called *core* in the current directory of the process that was running). This is something that you rarely see with plain perl scripts, but it can easily happen if you use modules written in *C* or *C++* and something goes wrong with them. Occasionally you will come across a bug in mod_perl itself (mod_perl is written in C), that was in a deep slumber before your code awakened it.

In the following sections we will go through in detail each of the problems presented, thoroughly discuss them and present a few techniques to solve them.

1.7.1 Locating and correcting Syntax Errors

While developing code we often make syntax mistakes, like forgetting to put a comma in a list, or a semi-colon at the end of a statement.

Even at the end of a {} block, where a semicolon is not required at the end of the last statement, it may be better to put one in: there is a chance that you will add more code later, and when you do you might forget to add the now required semicolon. Similarly, more items might be added later to a list; unlike many other languages, Perl has no problem when you end a list with a redundant comma.

One approach to locating syntactically incorrect code is to execute the script from the shell with the *-c* flag. This tells Perl to check the syntax but not to run the code (actually, it will execute BEGIN, END blocks, and *use()* calls, because these are considered as occurring outside the execution of your program). (Note also that Perl 5.6.0 has introduced a new special variable, $^C, which is set to true when perl is run with the *-c* flag; this provides an opportunity to have some further control over BEGIN and END blocks during syntax checking.) Also it's a good idea to add the -w switch to enable warnings:

```
perl -cw test.pl
```

If there are errors in the code, Perl will report the errors, and tell you at which line numbers in your script the errors were found.

The next step is to execute the script, since in addition to syntax errors there may be run time errors. These are the errors that cause the *"Internal Server Error"* page when executed from a browser. With plain CGI scripts it's the same as running plain Perl scripts -- just execute them and see that they work.

The whole thing is quite different with scripts that use Apache::* modules which can be used only from within the mod_perl server environment. These scripts rely on other code, and an environment which isn't available when you attempt to execute the script from the shell. There is no Apache request object available to the code when it is executed from the shell.

If you have a problem when using Apache::* modules, you can make a request to the script from a browser and watch the errors and warnings as they are logged to the *error_log* file. Alternatively you can use the Apache::FakeRequest module.

1.7.2 Using Apache::FakeRequest to Debug Apache Perl Modules

Apache::FakeRequest is used to set up an empty Apache request object that can be used for debugging. The Apache::FakeRequest methods just set internal variables with the same names as the methods and return the value of the internal variables. Initial values for methods can be specified when the object is created. The print method prints to STDOUT.

Subroutines for Apache constants are also defined so that you can use Apache::Constants while debugging, although the values of the constants are hard-coded rather than extracted from the Apache source code.

Let's write a very simple module, which prints *"OK"* to the client's browser:

```
package Apache::Example;
use Apache::Constants;

sub handler{
  my $r = shift;
  $r->send_http_header('text/plain');
  print "You are OK ", $r->get_remote_host, "\n";
  return OK;
}

1;
```

You cannot debug this module unless you configure the server to run it, by calling its handler from somewhere. So for example you could put in *httpd.conf*:

```
<Location /ex>
  SetHandler perl-script
  PerlHandler Apache::Example
</Location>
```

Then after restarting the server you could start a browser, request the location http://localhost/ex and examine the output. Tedious, no?

But with the help of Apache::FakeRequest you can write a little script that will emulate a request and return the output.

```
#!/usr/bin/perl

use Apache::FakeRequest ();
use Apache::Example ();

my $r = Apache::FakeRequest->new('get_remote_host'=>'www.foo.com');
Apache::Example::handler($r);
```

when you execute the script from the command line, you will see the following output:

```
You are OK www.foo.com
```

1.7.3 Finding the Line Which Triggered the Error or Warning

Perl has no problem with the line numbers and file names for modules that are read from disk in the normal way, but modules that are compiled via eval() such as `Apache::Registry` and `Apache::PerlRun` sometimes with some versions of Perl get confused.

There is the Perl <<HEREDOC inside eval "" problem that confuses the Perl current linenumber counter, newer Perls fix this. For older Perls compiling with the experimental **PERL_MARK_WHERE=1** should solve this.

Also if you happen to use line continuation \ your script, the line numbering will be off as well, since eval eats those \'s and combines the lines together. The solution: do not use \'s to mark lines as continued in Perl. Unlike shell languages, Perl doesn't require you to do that.

There are compiler directives to reset its counter to some value that you decide. You can always pepper your code with these to help you locate the problem. At the beginning of the line you could write something of the form:

```
#line nnn label
```

For example:

```
#line 298 myscript.pl
or
#line 890 some_label_to_be_used_in_the_error_message
```

The '#' must be in the first column, so if you cut and paste from this text you must remember to remove any leading white space.

The label is optional - the filename of the script will be used by default. This directive sets the line number of the **following** line, not the line the directive is on. You can use a little script to stuff every N lines of your code with these directives, but then you will have to remember to rerun this script every time you add or remove code lines. The script:

```perl
#!/usr/bin/perl
# Puts Perl line markers in a Perl program for debugging purposes.
# Also takes out old line markers.
die "No filename to process.\n" unless @ARGV;
my $filename = shift;
my $lines = 100;
open IN, $filename or die "Cannot open file: $filename: $!\n";
open OUT, ">$filename.marked"
    or die "Cannot open file: $filename.marked: $!\n";
my $counter = 1;
while (<IN>) {
  print OUT "#line $counter\n" unless $counter++ % $lines;
  next if /^#line /;
  print OUT $_;
}
close OUT;
close IN;
chmod 0755, "$filename.marked";
```

Another way of narrowing down the area to be searched is to move most of the code into a separate modules. This ensures that the line number will be reported correctly.

To have a complete trace of calls add:

```
use Carp ();
local $SIG{__WARN__} = \&Carp::cluck;
```

1.7.4 Using print() for Debugging

The universal debugging tool across nearly all platforms and programming languages is printf() or the equivalent output function. This can send data to the console, a file, an application window and so on. In perl we generally use the print() function. With an idea of where and when the bug is triggered, a developer can insert print() statements in the source code to examine the value of data at certain stages of execution.

However, it is rather difficult to anticipate all possible directions a program might take and what data to suspect of causing trouble. In addition, inline debugging code tends to add bloat and degrade the performance of an application and can also make the code harder to read and maintain. And you have to comment out or remove the debugging print() calls when you think that you have solved the problem. But if later you discover that you need to debug the same code again, you need at best to uncomment the debugging code lines or, at worst, to write them again from scratch.

Let's see a few examples where we use print() to debug some problem. In one of my applications I wrote a function that returns the date that was one week ago. Here it is:

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

print "A week ago the date was ",date_a_week_ago(),"\n";

# return a date one week ago as a string in format: MM/DD/YYYY
####################
sub date_a_week_ago{

  my @month_len   = (31,28,31,30,31,30,31,31,30,31,30,31);

  my ($day,$month,$year) = (localtime)[3..5];
  for (my $j = 0; $j < 7; $j++) {

    $day--;
    if ($day == 0) {

      $month--;
      if ($month == 0) {
        $year--;
        $month = 12;
      }

        # there are 29 days in February in a leap year
      $month_len[1] =
        (($year % 4 or $year % 100 == 0) and $year % 400 )
  ? 28 : 29;
```

```
        # set $day to be the last day of the previous month
      $day = $month_len[$month - 1];

   }    # end of if ($day == 0)
 }       # end of for ($i = 0;$i < 7;$i++)

  return sprintf "%02d/%02d/%04d",$month,$day,$year+1900;
}
```

This code is pretty straightforward. We get today's date and subtract one from the value of the day we get, updating the month and the year on the way if boundaries are being crossed (end of month, end of year). If we do it seven times in loop then at the end we should get a date that was a week ago.

Note that since locatime() returns the year as a value of `current_four_digits_format_year-1900` (which means that we don't have a century boundary to worry about) then if we are in the middle of the first week of the year 2000, the value of year returned by localtime() will be `100` and not `0` as you might mistakenly assume. So when the code does `$year--` it becomes `99` and not `-1`. At the end we add 1900 to get back the correct four-digit year format. (This is all correct as long as you don't go to the years prior to 1900)

Also note that we have to account for leap years where there are 29 days in February. For the other months we have prepared an array containing the month lengths.

Now when we run this code and check the result, we see that something is wrong. For example, if today is `10/23/1999` we expect the above code to print `10/16/1999`. In fact it prints `09/16/1999`, which means that we have lost a month. The above code is buggy!

Let's put a few debug print() statements in the code, near the `$month` variable:

```
  sub date_a_week_ago{

    my @month_len   = (31,28,31,30,31,30,31,31,30,31,30,31);

    my ($day,$month,$year) = (localtime)[3..5];
    print "[set] month : $month\n"; # DEBUG
    for (my $j = 0; $j < 7; $j++) {

      $day--;
      if ($day == 0) {

        $month--;
        if ($month == 0) {
          $year--;
          $month = 12;
        }
        print "[loop $i] month : $month\n"; # DEBUG

          # there are 29 days in February in a leap year
        $month_len[1] =
          (($year % 4 or $year % 100 == 0) and $year % 400 )
    ? 28 : 29;

          # set $day to be the last day of the previous month
```

```
        $day = $month_len[$month - 1];

    }   # end of if ($day == 0)
  }     # end of for ($i = 0;$i < 7;$i++)

  return sprintf "%02d/%02d/%04d",$month,$day,$year+1900;
}
```

When we run it we see:

```
[set] month : 9
```

It is supposed to be the number of the current month (`10`), but actually it is not. We have spotted a bug, since the only code that sets the `$month` variable consists of a call to localtime(). So did we find a bug in Perl? let's look at the manpage of the localtime() function:

```
% perldoc -f localtime

Converts a time as returned by the time function to a 9-element
array with the time analyzed for the local time zone.  Typically
used as follows:

  #  0    1    2     3     4    5     6     7     8
  ($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) =
                                         localtime(time);

All array elements are numeric, and come straight out of a struct
tm.  In particular this means that C<$mon> has the range C<0..11>
and C<$wday> has the range C<0..6> with Sunday as day C<0>.  Also,
C<$year> is the number of years since 1900, that is, C<$year> is
C<123> in year 2023, and I<not> simply the last two digits of the
year.  If you assume it is, then you create non-Y2K-compliant
programs--and you wouldn't want to do that, would you?
[more info snipped]
```

Which reveals to us that if we want to count months from 1 to 12 and not 0 to 11 we are supposed to increment the value of `$month`. Among other interesting facts about localtime() we also see an explanation of `$year`, which as I've mentioned before is set to the number of years since 1900.

We have found the bug in our code and learned new things about localtime(). To correct the above code we just increment the month after we call localtime():

```
    my ($day,$month,$year) = (localtime)[3..5];
    $month++;
```

1.7.5 Using print() and Data::Dumper for Debugging

Sometimes you need to peek into complex data structures, and trying to print them out can be tricky. That's where `Data::Dumper` comes to our rescue. For example if we create this complex data structure:

```
$data =
  {
   array => [qw(a b c d)],
   hash  => {
           foo => "oof",
           bar => "rab",
         },
  };
```

How do we print it out? Very easily:

```
use Data::Dumper;
print Dumper \$data;
```

What we get is a pretty-printed `$data`:

```
$VAR1 = \{
          'hash' => {
                      'foo' => 'oof',
                      'bar' => 'rab'
                    },
          'array' => [
                       'a',
                       'b',
                       'c',
                       'd'
                     ]
        };
```

While writing this example I made a mistake and wrote `qw(a b c d)` instead of `[qw(a b c d)]`. When I pretty-printed the contents of `$data` I immediately saw my mistake:

```
$VAR1 = \{
          'b' => 'c',
          'd' => 'hash',
          'HASH(0x80cd79c)' => undef,
          'array' => 'a'
        };
```

That's not what I wanted of course, but I spotted the bug and corrected it, as you saw in the original example from above.

Of course you can use

```
print STDERR $variable;
```

or:

```
warn $variable;
```

instead of print to have all the debug messages in the error_log, which makes it even easier to debug your code.

1.7.6 The Importance of a Good Concise Coding Style

Don't strive for elegant, clever code. Try to develop a good coding style by writing code which is concise yet easy to understand. It's much easier to find bugs in concise, simple code. And such code tends to have less bugs.

The *'one week ago'* example from the previous section is not concise. There is a lot of redundancy in it, and as a result it is harder to debug than it needs to be. Here is a condensed version of the main loop. As you can see, this version won't make it easier to understand the code:

```
for (0..6) {
  next if --$day;
  $year--,$month=12 unless --$month;
  $day = $month != 1
      ? $month_len[$month-1]
      : (($year % 4 or $year % 100 == 0) and $year % 400 )
          ? 28
          : 29;
}
```

Don't do that at home :)

Why did I present this version? Because it is too obscure, which makes it difficult to understand and maintain. On the other hand a part of this code is easier to understand.

Larry Wall, the author of Perl, is a linguist. He tried to define the syntax of Perl in a way that makes working in Perl much like working in English. So it can be a good idea to learn Perl coding idioms, some of which might seem odd at first but once you get used to them, you will find it difficult to understand how you could have lived without them before. I'll show just a few of the most common Perl coding idioms.

It's a good idea to write code which is more readable but which avoids redundancy, so it's better to write:

```
unless ($i) {...}
```

rather than:

```
if ($i == 0) {...}
```

if you want to test for trueness only.

Use a much more concise, Perlish style:

```
for my $j (0..6) {...}
```

instead of the syntax used in some other languages:

```
for (my $j=0; $j<=6; $j++) {...}
```

It's much simpler to write and comprehend code like this:

```
print "something" if $debug;
```

than this:

```
if($debug){
  print "something";
}
```

A good style that improves understanding, readability and reduces the chances of having a bug is shown below in the form of yet another rewrite of our *'one week ago'* code:

```
for (0..6) {
  $day--;
  next if $day;

  $month--;
  unless ($month){
    $year--;
    $month=12
  }

  if($month == 1){
    $day = (($year % 4 or $year % 100 == 0) and $year % 400 )
        ? 28 : 29;
  } else {
    $day = $month_len[$month-1];
  }
}
```

which is a happy medium between the excessively verbose style of the first version and very obscure second version.

And of course a two liner, which is much faster and easier to understand is:

```
sub date_a_week_ago{
  my ($day,$month,$year) = (localtime(time-604800))[3..5];
  return sprintf "%02d/%02d/%04d",$month+1,$day,$year+1900;
}
```

Just take the current date in seconds since epoch as time() returns, subtract a week in seconds (7*24*60*60 = 604800) and feed the result to localtime() - voila we've got the date of one week ago!

Why is the last version important, when the first one works just fine? Not because of performance issues (although this last one is twice as fast as the first), but because there are more ways to put a bug in the first version than there are in the last one.

1.7.7 Introduction to the Perl Debugger

As we saw earlier, it's *almost* always possible to debug code with the help of print(). However, it is impossible to anticipate all the possible directions which a program might take, and difficult to know what code to suspect when trouble occurs. In addition, inline debugging code tends to add bloat and degrade the performance of an application, although most applications offer inline debugging as a compile time option to avoid these hits. In any case, this information tends to only be useful to the programmer who added the print statements in the first place.

Sometimes you have to debug tens of thousands lines of Perl in an application, and while you may be a very experienced Perl programmer who can understand Perl code quite well by just looking at it, no mere mortal can even begin to understand what will actually happen in such a large application, until the code is running. So you just don't know where to start adding your trusty print() statements to see what is happening inside.

The most effective way to track down a bug is to run the program inside an interactive debugger. The majority of programming languages have such a tool available, allowing one to see what is happening inside an application while it is running. The basic features of an interactive debugger allow you to:

- Stop at a certain point in the code, based on a routine name or source file and line number

- Stop at a certain point in the code, based on conditions such as the value of a given variable

- Perform an action without stopping, based on the criteria above

- View and modify the value of variables at any given point

- Provide context information such as stack traces and source windows

It does take practice to learn the most effective ways of using an interactive debugger, but the time and effort will be paid back many-fold in the long run.

Most C and C++ programmers are familiar with the interactive GNU debugger (gdb). gdb is a stand-alone program that requires your code to be compiled with debugging symbols to be useful. While gdb can be used to debug the Perl interpreter itself, it cannot be used to debug your Perl scripts.

Not to worry, Perl provides its own interactive debugger, called perldb. Giving control of your Perl program to the interactive debugger is simply a matter of specifying the -d command line switch. When this switch is used, Perl inserts debugging hooks into the program syntax tree, but it leaves the job of debugging to a Perl module separate from the perl binary itself.

I will start by introducing a few of the basic concepts and commands of the Perl interactive debugger. These warm-up examples all run from the command line, independent of mod_perl, but are all still relevant when we do finally go inside Apache.

It might be useful to keep the *perldebug* manpage handy for reference while reading this section, and for future debugging sessions on your own.

The interactive debugger will attach to the current terminal and present you with a prompt just before the first program statement is executed. For example:

```
% perl -d -le 'print "mod_perl rules the world"'

Loading DB routines from perl5db.pl version 1.0402

Emacs support available.

Enter h or 'h h' for help.

main::(-e:1):   print "mod_perl rules the world"
  DB<1>
```

The source line shown is the line which Perl is *about* to execute, the `next` command (or just `n`) will cause this line to be executed after which execution will stop again just before the next line:

```
main::(-e:1):   print "mod_perl rules the world"
  DB<1> n
mod_perl rules the world
Debugged program terminated.  Use q to quit or R to restart,
use O inhibit_exit to avoid stopping after program termination,
h q, h R or h O to get additional info.
DB<1>
```

In this case, our example code is only one line long, so we have finished interacting after the first line of code is executed. Let's try again with slightly longer example which is the following script:

```
my $word = 'mod_perl';
my @array = qw(rules the world);

print "$word @array\n";
```

Save the script in a file called *domination.pl* and run with the −d switch:

```
% perl -d domination.pl

main::(domination.pl:1):    my $word = 'mod_perl';
  DB<1> n
main::(domination.pl:2):    my @array = qw(rules the world);
  DB<1>
```

At this point, the first line of code has been executed and the variable `$word` has been assigned the value *mod_perl*. We can check this by using the `p` command (an abbreviation for the `print` command, the two are interchangeable):

```
main::(domination.pl:2):    my @array = qw(rules the world);
  DB<1> p $word
mod_perl
```

The `print` command works just like the Perl's built-in print() function, but adds a trailing newline and outputs to the `$DB::OUT` file handle, which is normally opened on the terminal where Perl was launched from. Let's carry on:

```
  DB<2> n
main::(domination.pl:4):        print "$word @array\n";
  DB<2> p @array
rulestheworld
  DB<3> n
mod_perl rules the world
Debugged program terminated.  Use q to quit or R to restart,
use O inhibit_exit to avoid stopping after program termination,
h q, h R or h O to get additional info.
```

Ouch, `p @array` printed `rulestheworld` and not `rules the world`, as you might expect it to, but that's absolutely correct. If you print an array without expanding it first into a string it will be printed without adding the content of the `$"` variable (otherwise known as `$LIST_SEPARATOR` if the `English` pragma is being used) between the elements of the array.

If you type:

```
print "@array";
```

the output will be `rules the world` since the default value of the `$"` variable is a single space.

You should have noticed by now that there is some valuable information to the left of each executable statement:

```
main::(domination.pl:4):        print "$word @array\n";
  DB<2>
```

First is the current package name, in this case `main::`. Next is the current filename and statement line number, *domination.pl* and 4 in the example above. The number presented at the prompt is the command number which can be used to recall commands from the session history, using the `!` command followed by this number. For example, `!1` would repeat the first command:

```
% perl -d -e0

main::(-e:1):   0
  DB<1> p $]
5.00503
  DB<2> !1
p $]5.00503
  DB<3>
```

Where `$]` is the perl's version number. As you see `!1` prints the value of `$]`, preceded by the command that was executed.

Things start to get more interesting as the code does. In the example script below (save it to a file called *test.pl*) we've increased the number of source files and packages by including the standard `Symbol` module, along with an invocation of its gensym() function:

```
use Symbol ();

my $sym = Symbol::gensym();

print "$sym\n";
```

```
% perl -d test.pl

main::(test.pl:3):      my $sym = Symbol::gensym();
  DB<1> n
main::(test.pl:5):      print "$sym\n";
  DB<1> n
GLOB(0x80c7a44)
```

First, notice the debugger did not stop at the first line of the file. This is because `use ...` is a compile-time statement, not a run-time statement. Also notice there was more work going on than the debugger revealed. That's because the `next` command does not enter subroutine calls. To step into a subroutine code use the `step` command (or its abbreviated form `s`):

```
% perl -d test.pl

main::(test.pl:3):      my $sym = Symbol::gensym();
  DB<1> s
Symbol::gensym(/usr/lib/perl5/5.00503/Symbol.pm:86):
86:       my $name = "GEN" . $genseq++;
  DB<1>
```

Notice the source line information has changed to the `Symbol::gensym` package and the `Symbol.pm` file. We can carry on by hitting the return key at each prompt, which causes the debugger to repeat the last `step` or `next` command. It won't repeat a `print` command though. The debugger will eventually return from the subroutine back to our main program:

```
  DB<1>
Symbol::gensym(/usr/lib/perl5/5.00503/Symbol.pm:87):
87:       my $ref = \*{$genpkg . $name};
  DB<1>
Symbol::gensym(/usr/lib/perl5/5.00503/Symbol.pm:88):
88:       delete $$genpkg{$name};
  DB<1>
Symbol::gensym(/usr/lib/perl5/5.00503/Symbol.pm:89):
89:       $ref;
  DB<1>
main::(test.pl:5):      print "$sym\n";
  DB<1>
GLOB(0x80c7a44)
```

Our line-by-line debugging approach has served us well for this small program, but imagine the time it would take to step through a large application at the same pace. There are several ways to speed up a debugging session, one of which is known as *setting a breakpoint*. The `breakpoint` command (b) can be used for instructing the debugger to stop at a named subroutine or at any line of any file. In this example session, at the first debugger prompt we will set a breakpoint at the `Symbol::gensym` subroutine, telling the debugger to stop at the first line of this routine when it is called. Rather than move along with `next` or `step` we give the `continue` command (c) which tells the debugger to execute the script without stopping until it reaches a breakpoint:

```
% perl -d test.pl

main::(test.pl:3):      my $sym = Symbol::gensym();
  DB<1> b Symbol::gensym
  DB<2> c
Symbol::gensym(/usr/lib/perl5/5.00503/Symbol.pm:86):
86:          my $name = "GEN" . $genseq++;
```

Now let's pretend we are debugging a large application where `Symbol::gensym` might be called in various places. When the subroutine breakpoint is reached, by default the debugger does not reveal where it was called from. One way to find out this information is with the `Trace` command (`T`):

```
  DB<2> T
$ = Symbol::gensym() called from file 'test.pl' line 3
```

In this example, the call stack is only one level deep, so only that line is printed. We'll look at an example with a deeper stack later. The left-most character reveals the context in which the subroutine was called. `$` represents scalar context, in other examples you may see `@` which represents list context or `.` which represents void context. In our case we have called:

```
my $sym = Symbol::gensym();
```

which calls the `Symbol::gensym()` in scalar context.

Below we've made our *test.pl* example a little more complex. First, we've added a `My::World` package declaration at the top of the script, so we are no longer working in the `main::` package. Next, we've added a subroutine named do_work() which invokes the familiar `Symbol::gensym`, along with another function called `Symbol::qualify` and then returns a hash reference of the results. The do_work() routine is invoked inside a *for* loop which will be run twice:

```
package My::World;

use Symbol ();

for (1,2) {
  do_work("now");
}

sub do_work {
  my ($var) = @_;

  return undef unless $var;

  my $sym  = Symbol::gensym();
  my $qvar = Symbol::qualify($var);

  my $retval = {
              'sym' => $sym,
              'var' => $qvar,
              };

  return $retval;
}
```

We'll start by setting a few breakpoints and then we use the `List` command (`L`) to display them:

```
% perl -d test.pl

My::World::(test.pl:5):   for (1,2) {
  DB<1> b Symbol::qualify
  DB<2> b Symbol::gensym
  DB<3> L
/usr/lib/perl5/5.00503/Symbol.pm:
 86:         my $name = "GEN" . $genseq++;
   break if (1)
 95:         my ($name) = @_;
   break if (1)
```

The filename and line number of the breakpoint are displayed just before the source line itself. Because both breakpoints are located in the same file, the filename is displayed only once. After the source line we see the condition on which to stop. In this case, as the constant value 1 indicates, we will always stop at these breakpoints. Later on you'll see how to specify a condition.

As we will see, when the `continue` command is executed, the execution of the program stops at one of these breakpoints, either on line 86 or 95 of the `/usr/lib/perl5/5.00503/Symbol.pm` file, whichever is reached first. The displayed code lines are the first rows of the two subroutines from `Symbol.pm`. Breakpoints may only be applied to lines of run-time executable code, you cannot put breakpoints on empty lines or comments for example.

In our example the `List` command shows which lines the breakpoints were set on, but we cannot tell which breakpoint belongs to which subroutine. There are two ways to find this out. One is to run the `continue` command and when it stops, execute the `Trace` command we saw before:

```
  DB<3> c
Symbol::gensym(/usr/lib/perl5/5.00503/Symbol.pm:86):
 86:         my $name = "GEN" . $genseq++;
  DB<3> T
$ = Symbol::gensym() called from file 'test.pl' line 14
. = My::World::do_work('now') called from file 'test.pl' line 6
```

So we see that it was `Symbol::gensym`. The other way is to ask for a listing of a range of lines from the code. For example, let's check which subroutine line 86 is a part of. We use the `list` (lowercase!) command (`l`), which displays parts of the code. The `list` command accepts various arguments, the one that we want to use here is a range of lines. Since the breakpoint is at line 86, let's print a few lines above and below that line:

```
  DB<3> l 85-87
85       sub gensym () {
86==>b       my $name = "GEN" . $genseq++;
87:          my $ref = \*{$genpkg . $name};
```

Now we know it's the `gensym` sub and we also see the breakpoint displayed with the help of the `==>b` markup. We could also use the name of the sub to display its code:

```
  DB<4> l Symbol::gensym
85      sub gensym () {
86==>b     my $name = "GEN" . $genseq++;
87:        my $ref = \*{$genpkg . $name};
88:        delete $$genpkg{$name};
89:        $ref;
90      }
```

The `delete` command (d) is used to remove a breakpoint by specifying the line number of the breakpoint. Let's remove the first one:

```
  DB<5> d 95
```

The `Delete` command (with a capital 'D') or D removes all currently installed breakpoints.

Now let's look again at the trace produced at the breakpoint:

```
  DB<3> c
Symbol::gensym(/usr/lib/perl5/5.00503/Symbol.pm:86):
86:        my $name = "GEN" . $genseq++;
  DB<3> T
$ = Symbol::gensym() called from file 'test.pl' line 14
. = My::World::do_work('now') called from file 'test.pl' line 6
```

As you can see, the stack trace prints the values which are passed into the subroutine. Ah, and perhaps we've found our first bug, as we can see do_work() was called in void context, so the return value was lost into thin air. Let's change the *'for'* loop to check the return value of do_work():

```
for (1,2) {
  my $stuff = do_work("now");
  if ($stuff) {
     print "work is done\n";
  }
}
```

In this session we will set a breakpoint at line 7 of `test.pl` where we check the return value of do_work():

```
% perl -d test.pl

My::World::(test.pl:5):   for (1,2) {
  DB<1> b 7
  DB<2> c
My::World::(test.pl:7):      if ($stuff) {
  DB<2>
```

Our program is still small, but already it is getting more difficult to understand the context of just one line of code. The `window` command (w) will list a few lines of code that surround the current line:

```
  DB<2> w
4
5:          for (1,2) {
6:             my $stuff = do_work("now");
7==>b          if ($stuff) {
8:                print "work is done\n";
9:             }
10         }
11
12         sub do_work {
13:           my ($var) = @_;
```

The arrow points to the line which is about to be executed and also contains a 'b' indicating that we have set a breakpoint at this line. The breakable lines of code include a ':' immediately after the line number.

Please notice that this demonstration was done before perl 5.8 was released, which redefined some of the letters to have a different meaning. For example w was replaced with v. Please see the perldebug manpage.

Now, let's take a look at the value of the $stuff variable with the trusty old print command:

```
  DB<2> p $stuff
HASH(0x82b89b4)
```

That's not very useful information. Remember, the print command works just like the built-in print() function does. The debugger's x command evaluates a given expression and prints the results in a "pretty" fashion:

```
  DB<3> x $stuff
0  HASH(0x82b89b4)
    'sym' => GLOB(0x826a944)
       -> *Symbol::GEN0
    'var' => 'My::World::now'
```

There, things seem to be okay, let's double check by calling do_work() with a different value and print the results:

```
  DB<4> x do_work('later')
0  HASH(0x82bacc8)
    'sym' => GLOB(0x818f16c)
       -> *Symbol::GEN1
    'var' => 'My::World::later'
```

We can see the symbol was incremented from GEN0 to GEN1 and the variable later was qualified, as expected.

Now let's change the test program a little to iterate over a list of arguments held in @args and print a slightly different message:

```
package My::World;

use Symbol ();
```

```
my @args = qw(now later);
for my $arg (@args) {
  my $stuff = do_work($arg);
  if ($stuff) {
      print "do your work $arg\n";
  }
}

sub do_work {
  my ($var) = @_;

  return undef unless $var;

  my $sym = Symbol::gensym();
  my $qvar = Symbol::qualify($var);

  my $retval = {
      'sym' => $sym,
      'var' => $qvar,
  };

  return $retval;
}
```

There are only two arguments in the list, so stopping to look at each one isn't too time consuming, but consider the debugging pace if we had a large list of 100 or so entries. It is possible to customize break-points by specifying a condition. Each time a breakpoint is reached, the condition is evaluated, stopping only if the condition is true. In the session below, the window command shows breakable lines and we set a breakpoint at line 7 with the condition $arg eq 'later'. As we continue, the breakpoint is skipped when $arg has the value of *now* but not when it has the value of *later*:

```
% perl -d test.pl

My::World::(test.pl:5): my @args = qw(now later);
  DB<1> w
2
3:      use Symbol ();
4
5==>    my @args = qw(now later);
6:      for my $arg (@args) {
7:          my $stuff = do_work($arg);
8:          if ($stuff) {
9:              print "do your work $arg\n";
10          }
11      }
```

The ==> symbol shows us the line of code that's about to be executed.

```
  DB<1> b 7 $arg eq 'later'
  DB<2> c
do your work now
My::World::(test.pl:7):     my $stuff = do_work($arg);
  DB<2> n
My::World::(test.pl:8):     if ($stuff) {
  DB<2> x $stuff
```

```
 0  HASH(0x82b90e4)
    'sym' => GLOB(0x82b9138)
       -> *Symbol::GEN1
    'var' => 'My::World::later'
  DB<5> c
do your work later
Debugged program terminated.  Use q to quit or R to restart,
```

There are plenty more tricks left to pull from the perldb bag, but you should now understand enough about the debugger to try them on your own with the perldebug manpage by your side. Quick online help from inside the debugger can be reached by typing the h command. It will display a list of the most useful commands and a short explanation of what they do.

1.7.8 Interactive Perl Debugging under mod_cgi

`Devel::ptkdb` is a visual Perl debugger that uses perlTk for the user interface and requires a windows system like X-Windows or Windows to run.

To debug a plain perl script with ptkdb, invoke it as:

```
  % perl -d:ptkdb myscript.pl
```

The Tk application will be loaded. Now you can do most of the debugging you did with the command line Perl debugger, but using a simple GUI to set/remove breakpoints, browse the code, step through it and more.

With the help of ptkdb you can debug your CGI scripts running under mod_cgi. Be sure that the web server's Perl installation includes the Tk package. In order to enable the debugger you should change your "shebang" line from

```
  #! /usr/local/bin/perl -Tw
```

to

```
  #! /usr/local/bin/perl -Twd:ptkdb
```

You can debug scripts remotely if you're using a Unix based server and if the machine where you are writing the script has an X-server. The X-server can be another Unix workstation, or a Macintosh or Win32 platform with an appropriate X-Windows package. You must insert the following BEGIN subroutine into your script:

```
  BEGIN {
    $ENV{'DISPLAY'} = "myHostname:0.0" ;
  }
```

You can use either the IP (*123.123.123.123:0.0*) or the DNS convention (*myhost.com:0.0*). You must be sure that your web server has permission to open windows on your X-server (see the *xhost* manpage for more info).

Access the web page with the browser and *Submit* the script as normal. The ptkdb window should appear on the monitor if you have correctly set the $ENV{'DISPLAY'} variable. At this point you can start debugging your script. Be aware that the browser may timeout waiting for the script to run.

To expedite debugging you may want to set your breakpoints in advance with a *.ptkdbrc* file and use the $DB::no_stop_at_start variable. NOTE: for debugging web scripts you may have to have the *.ptkdbrc* file installed in the server account's home directory (~www) or whatever username the webserver is running under. Also try installing a *.ptkdbrc* file in the same directory as the target script.

META: insert snapshots of ptkdb screen

ptkdb is not part of the standard perl distribution; it is available from CPAN: http://www.perl.com/CPAN/authors/id/A/AE/AEPAGE/

Some fixes for ptkdb to work under mod_perl are listed here: http://article.gmane.org/gmane.comp.apache.mod-perl/7562

1.7.9 Non-Interactive Perl Debugging under mod_perl

To debug scripts running under mod_perl either use Apache::DB (interactive Perl debugging) or an older non-interactive method as described below.

The NonStop debugger option enables you to get some decent debugging information when running under mod_perl. For example, before starting the server:

```
% setenv PERL5OPT -d
% setenv PERLDB_OPTS "NonStop=1 LineInfo=db.out AutoTrace=1 frame=2"
```

Now watch db.out for line:filename info. This is most useful for tracking those core dumps that normally leave us guessing, even with a stack trace from gdb. *db.out* will show you what Perl code triggered the core dump. *'man perldebug'* for more PERLDB_OPTS. Note that Perl will ignore PERL5OPT if Perl-TaintCheck is On.

1.7.10 Interactive mod_perl Debugging

Now we'll turn to looking at how the interactive debugger is used in a mod_perl environment. The Apache::DB module available from CPAN provides a wrapper around perldb for debugging Perl code running under mod_perl.

The server must be run in non-forking mode to use the interactive debugger, this mode is turned on by passing the -X flag to the httpd executable. It is convenient to use an IfDefine section around the Apache::DB configuration, the example below does this using the name *PERLDB*. With this setup, debugging is only turned on when starting the server with the httpd -D PERLDB command.

This section should be at the top of the Perl configuration section of the configuration file, before any other Perl code is pulled in, so that debugging symbols will be inserted into the syntax tree, triggered by the call to Apache::DB->init. The Apache::DB::handler can be configured using any of the Perl*Handler directives, in this case you use a PerlFixupHandler so handlers in the response

phase will bring up the debugger prompt:

```
<IfDefine PERLDB>

  <Perl>
    use Apache::DB ();
    Apache::DB->init;
  </Perl>

  <Location />
    PerlFixupHandler Apache::DB
  </Location>

</IfDefine>
```

Since we have used / as the argument to the `Location` directive, the debugger will be invoked for any kind of request (even for static documents and images) but of course it will immediately quit unless there is some Perl module registered to handle these requests.

In our first example, we will debug the standard `Apache::Status` module, which is configured like this:

```
PerlModule Apache::Status
<Location /perl-status>
  PerlHandler Apache::Status
  SetHandler perl-script
</Location>
```

When the server is started with the debugging flag, a notice will be printed to the console:

```
% httpd -X -D PERLDB
[notice] Apache::DB initialized in child 950
```

The debugger prompt will not be available until the first request is made, in our case to http://local-host/perl-status. Once we are at the prompt, all the standard debugging commands are available. First we run window to get some of the context for the code being debugged, then we move to the next statement after a value has been assigned to `$r`, and finally we print the request URI. If no breakpoints are set, the `continue` command will give control back to Apache and the request will finish with the `Apache::Status` main menu showing in the browser window:

```
Loading DB routines from perl5db.pl version 1.0402
Emacs support available.

Enter h or 'h h' for help.

Apache::Status::handler(/usr/lib/perl5/site_perl/5.005/i386-linux/Apache/Status.pm:55):
55:         my ($r) = @_;
  DB<1> w
52        }
53
54        sub handler {
55==>         my ($r) = @_;
56:           Apache->request($r); #for Apache::CGI
57:           my $qs = $r->args || "";
```

```
  58:            my $sub = "status_$qs";
  59:            no strict 'refs';
  60
  61:            if($qs =~ s/^(noh_\w+).*/$1/) {
    DB<1> n
 Apache::Status::handler(/usr/lib/perl5/site_perl/5.005/i386-linux/Apache/Status.pm:56):
  56:            Apache->request($r); #  for Apache::CGI
    DB<1> p $r->uri
 /perl-status
    DB<2> c
```

All the techniques we saw while debugging plain perl scripts can be applied to this debugging session.

Debugging `Apache::Registry` scripts is somewhat different, because the handler routine does quite a bit of work before it reaches your script. In this example, we make a request for `/perl/test.pl`, which consists of this code:

```
use strict;

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

print "mod_perl rules";
```

When a request is issued, the debugger stops at line 28 of *Apache/Registry.pm*. We set a breakpoint at line 140, which is the line that actually calls the script wrapper subroutine. The `continue` command will bring us to that line, where we can step into the script handler:

```
  Apache::Registry::handler(/usr/lib/perl5/site_perl/5.005/i386-linux/Apache/Registry.pm:28):
28:         my $r = shift;
    DB<1> b 140
    DB<2> c
  Apache::Registry::handler(/usr/lib/perl5/site_perl/5.005/i386-linux/Apache/Registry.pm:140):
 140:            eval { &{$cv}($r, @_) } if $r->seqno;
    DB<2> s
 Apache::ROOT::perl::test_2epl::handler((eval 87):3):
 3:          my $r = shift;
```

Notice the funny package name, that's generated from the URI of the request for namespace protection. The filename is not displayed, since the code was compiled via eval(), but the `print` command can be used to show you `$r->filename`:

```
    DB<2> n
 Apache::ROOT::perl::test_2epl::handler((eval 87):4):
 4:          $r->send_http_header('text/plain');
    DB<2> p $r->filename
 /home/httpd/perl/test.pl
```

The line number might seem off too, but the window command will give you a better idea where you are:

```
   DB<4> w
1:       package Apache::ROOT::perl::test_2epl;use Apache qw(exit);
sub handler {  use strict;
2
3:           my $r = shift;
4==>         $r->send_http_header('text/plain');
5
6:           print "mod_perl rules";
7
8:       }
9        ;
```

The code from the *test.pl* file is between lines 2 and 7, the rest is the `Apache::Registry` magic to
cache your code inside a *handler* subroutine.

It will always take some practice and patience when putting together debugging strategies that make effec-
tive use of the interactive debugger for various situations. Once you have a good strategy, bug squashing
can actually be quite a bit of fun!

1.7.11 ptkdb and Interactive mod_perl Debugging

As you saw earlier you can use the `ptkdb` visual debugger to debug CGI scripts running under mod_cgi.
But it won't work for mod_perl using the same configuration as used in mod_cgi. We have to tweak the
Apache/DB.pm module to use *Devel/ptkdb.pm* instead of *Apache/perl5db.pl*.

Open the file in your favorite editor and replace:

```
require 'Apache/perl5db.pl';
```

with:

```
require 'Devel/ptkdb.pm';
```

Now when you use the interactive mod_perl debugger configuration from the previous section and issue a
request, the *ptkdb* visual debugger will be loaded.

If you are debugging `Apache::Registry` scripts, as in the terminal debugging mode example, go to
line 140 (or to whatever line the `eval { &{$cv}($r, @_) } if $r->seqno;` statement is
located) and press the *step in* button to start the debug of the script itself.

Note that you can use Apache with `ptkdb` in plain multi-server mode, you don't have to start `httpd`
with the `-X` option.

META: One caveat:

When the request is completed, `ptkdb` hangs. Does anyone know what code should be registered for it to
exit on completion? To replace the original `Apache::DB` cleanup code, as:

```
if (ref $r) {
$SIG{INT} = \&DB::catch;
$r->register_cleanup(sub {
    $SIG{INT} = \&DB::ApacheSIGINT();
});
}
```

Any Perl/Tk guru to assist???

1.7.12 Debugging when Server Crashes on Startup before Writing to Log File.

If your server crashes on startup, you need to start it under gdb and ask it to generate a stack trace.

I'll emulate a faulty server by starting a startup file with the dump() command:

```
startup.pl
----------
dump;
1;
```

and then requiring this file from the *httpd.conf*:

```
PerlRequire /path/to/startup.pl
```

Make sure no server is running on port 80 or use an alternate config with an alternate port if using a production server.

```
% gdb /path/to/httpd
(gdb) set args -X
```

Use:

```
set args -X -f /path/to/alternate/serverconfig_ifneeded.conf
```

if the server must be started from an alternative configuration file.

Now run the program:

```
(gdb) run

Starting program: /usr/local/apache/bin/httpd -X

Program received signal SIGABRT, Aborted.
0x400da4e1 in __kill () from /lib/libc.so.6
```

At this point the server should die because of the call to `dump()`. When that happens we use `bt` or `where` to ask for a stack back trace.

```
(gdb) where

#0  0x400da4e1 in __kill () from /lib/libc.so.6
#1  0x80d43bc in Perl_my_unexec ()
#2  0x8119544 in Perl_pp_goto ()
#3  0x8118990 in Perl_pp_dump ()
#4  0x812b2ad in Perl_runops_standard ()
#5  0x80d3a9c in perl_eval_sv ()
#6  0x807ef1c in perl_do_file ()
#7  0x807ef4f in perl_load_startup_script ()
#8  0x807b7ec in perl_cmd_require ()
#9  0x8092af7 in ap_clear_module_list ()
#10 0x8092f43 in ap_handle_command ()
#11 0x8092fd7 in ap_srm_command_loop ()
#12 0x80933e0 in ap_process_resource_config ()
#13 0x8093ca2 in ap_read_config ()
#14 0x809db63 in main ()
#15 0x400d41eb in __libc_start_main (main=0x809d8dc <main>, argc=2,
    argv=0xbffffab4, init=0x80606f8 <_init>, fini=0x812b38c <_fini>,
    rtld_fini=0x4000a610 <_dl_fini>, stack_end=0xbffffaac)
    at ../sysdeps/generic/libc-start.c:90
```

If you do not know what this trace means, you could send it to the mod_perl mailing list to ask for help. Make sure to include the version numbers of Apache, mod_perl and Perl, and use a subject line that says something about the problem rather than 'help'.

In our case we already know that the server is supposed to die when compiling the startup file and we can clearly see that from the trace. We always read it from the bottom upward:

We are in config file:

```
#13 0x8093ca2 in ap_read_config ()
```

We do require:

```
#8  0x807b7ec in perl_cmd_require ()
```

We load the file and compile it:

```
#6  0x807ef1c in perl_do_file ()
#5  0x80d3a9c in perl_eval_sv ()
```

`dump()` gets executed:

```
#3  0x8118990 in Perl_pp_dump ()
```

`dump()` calls `__kill()`:

```
#0  0x400da4e1 in __kill () from /lib/libc.so.6
```

1.8 Hanging Processes: Detection and Diagnostics

Sometimes a httpd process might hang in the middle of processing a request, either because there is a bug in your code (e.g. the code is stuck in a while loop), it gets blocked by some system call or because of a resource deadlock) or for some other reason. In order to fix the problem we need to learn what circumstances the process hangs in (detection), so we can reproduce the problem and after that to discover why there is problem (diagnostics).

1.8.1 Hanging because of the OS Problem

Sometimes you can find a process hanging because of some kind of the system problem. For example if the processes was doing some disk IO operation it might get stuck in uninterruptable sleep ('D' disk wait in ps(1) report, 'U' in top(1)) which indicates that either something is broken in your kernel or that you're using NFS. Or and you cannot kill -9 this process.

Another process that cannot be killed with kill -9 is a zombie process ('Z' disk wait in ps(1) report, `<defunc>` in top(1)), in which case the process is already dead and Apache didn't wait on it properly.

In the case of *disk wait* you can actually get the *wait* channel from ps(1) and look it up in your kernel symbol table to find out what resource it was waiting on. It might point the way to what component of the system was misbehaving if the problem occurred frequently.

1.8.2 An Example of Code that Might Hang a Process

Deadlock is the situation where, for example, two processes, say X and Y, need two resources, A and B to continue. X holds onto A and Y holds onto B. There is no possibility for Y to continue before X releases A. But X cannot release A before it gets Y.

Look at the following example. Your process has to gain a lock on some resource (e.g. a file) before it continues. So it makes an attempt, and if that fails it sleep()s for a second and increments a counter:

```
until(gain_lock()){
  $tries++;
  sleep 1;
}
```

Because there are many processes competing for this resource, or perhaps because there is a deadlock, gain_lock() always fails. The process is hung.

Another situation that you may very often encounter is exclusive lock starvation. Generally there are two lock types in use: *SHARED* locks, which allow many processes to perform *READ* operations simultaneously, and *EXCLUSIVE* locks. The latter permits access only by a single process and so makes a safe *WRITE* operation possible.

You can lock any kind of resource, although in our examples we will talk about files.

If there is a *READ* lock request, it is granted as soon as the file becomes unlocked or immediately if it is already *READ* locked. The lock status becomes *READ* on success.

If there is a *WRITE* lock request, it is granted as soon as the file becomes unlocked. Lock status becomes *WRITE* on success.

Normally it is the *WRITE* lock request which is the most important. If the file is being *READ* locked, a process that requests to write will poll until there are no reading or writing process left. However, lots of processes can successfully read the file, since they do not block each other from doing so. This means that a process that wants to write to the file (first obtaining an exclusive lock) never gets a chance to squeeze in. The following diagram represents a possible scenario where everybody can read but no one can write:

```
[-p1-]                      [--p1--]
   [--p2--]
 [---------p3---------]
               [------p4-----]
   [--p5--]   [----p5----]
```

Let's look at some real code and see it in action. The following script imports flock() related parameters from the `Fcntl` module, and opens a file that will be locked. It then defines and sets two variables: `$lock_type` and `$lock_type_verbose`. These are set to `LOCK_EX` and `EX` respectively if the first command line argument (`$ARGV[0]`) is defined and equal to *w*. This indicates that this process will try to gain a *WRITE* (exclusive) lock. Otherwise the two are set to `LOCK_SH` and `<SH` for a *SHARED* (read) lock.

Once the variables are set, we enter the infinite `while(1)` loop that attempts to lock the file by the mode set in `$lock_type`. It report success and the type of lock that was gained, then it sleeps for a random period between 0 and 9 seconds and unlocks the file. The loop then starts from the beginning.

```perl
lock.pl
-------------------
#!/usr/bin/perl -w
use Fcntl qw(:flock);

$lock = "/tmp/lock";

open LOCK, ">$lock" or die "Cannot open $lock for writing: $!";
my $lock_type         = LOCK_SH;
my $lock_type_verbose = 'SH';
if (defined $ARGV[0] and $ARGV[0] eq 'w'){
  $lock_type         = LOCK_EX;
  $lock_type_verbose = 'EX';
}

while(1){
  flock LOCK,$lock_type;
    # start of critical section
  print "$$: $lock_type_verbose\n";
  sleep int(rand(10));
    # end of critical section
  flock LOCK, LOCK_UN;
}
close LOCK;
```

It's very easy to see *WRITE* process starvation if you spawn a few of the above scripts simultaneously. Start the first few as *READ* processes and then start one *WRITE* process like this:

```
 % ./lock.pl r & ; ./lock.pl r & ; ./lock.pl r & ; ./lock.pl w &
```

You see something like:

```
  24233: SH
  24232: SH
  24232: SH
  24233: SH
  24232: SH
  24233: SH
  24231: SH
  24231: SH
  24231: SH
```

and not a single `EX` line... When you kill off the reading processes, then the write process will gain its lock. Note that as this is a rough example, I used the sleep() function. To simulate a real situation you need to use the `Time::HiRes` module, which allows you to choose more precise intervals to sleep.

The interval between lock and unlock is called a *Critical Section*, which should be kept as short as possible (in terms of the time taken to execute the code, and not in terms of the number of lines of code). As you just saw, a single sleep statement can make the critical section long.

To summarize, if you have a script that uses both *READ* and *WRITE* locks and the critical section isn't very short, the writing process might be starved. After a while a browser that initiated this request will timeout the connection and abort the request, but it's much more likely that user will press the *Stop* or *Reload* button before that happens. Since the process in question is just waiting, there is no way for Apache to know that the request was aborted. It will hang until the lock is gained. Only when a write to a client's broken connection is attempted will Apache terminate the script.

1.8.3 Detecting hanging processes

It's not so easy to detect hanging processes. There is no way you can tell how long the request is taking to process by using plain system utilities like ps() and top(). The reason is that each Apache process serves many requests without quitting. System utilities can tell how long the process has been running since its creation, but this information is useless in our case, since Apache processes normally run for extended periods.

However there are a few approaches that can help to detect a hanging process.

If the process hangs and demands lots of resources it's quite easy to spot it by using the top() utility. You will see the same process show up in the first few lines of the automatically refreshed report. But often the hanging process uses few resources, e.g. when waiting for some event to happen.

Another easy case is when some process thrashes the *error_log*, writing millions of error messages there. Generally this process uses lots of resources and is also easily spotted by using top().

There are other tools that report the status of Apache processes.

- **The mod_status module, which is usually accessed from the */server_status* location.**
- **The `Apache::VMonitor` module.**

Both tools provide counters of processed requests per Apache process.

You can watch the report for a few minutes, and try to spot any process which has the same number of processed requests while its status is 'W' (waiting). This means that it has hung.

But if you have fifty processes, it can be quite hard to spot such a process. Apache::Watchdog::RunAway is a hanging processes monitor and terminator that implements this feature and should be used to solve this kind of problem.

If you've got a real problem, and the processes hang one after the other, the time will come when the number of hanging processes is equal to the value of `MaxClients`. This means that no more processes will be spawned. As far as the users are concerned your server is down. It is easy to detect this situation, attempt to resolve it and notify the administrator using a simple crontab watchdog that requests some very light script periodically. (See Monitoring the Server. A watchdog.)

In the watchdog you set a timeout appropriate for your service, which may be anything from a few seconds to a few minutes. If the server fails to respond before the timeout expires, the watchdog has spotted trouble and attempts to restart the server. After a restart an email report is sent to the administrator saying that there was a problem and whether or not the restart was successful.

If you get such reports constantly something is wrong with your web service and you should revise your code. Note that it's possible that your server is being overloaded by more requests than it can handle, so the requests are being queued and not processed for a while, which triggers the watchdog's alarm. If this is a case you may need to add more servers or more memory, or perhaps split your single machine across a cluster of machines.

1.8.4 Determination of the reason

Given the process id (PID), there are three ways to find out where the server is hanging.

1. Deploying the Perl calls tracing mechanism. This will allow to spot the location of the Perl code that has triggered the problem.

2. Using the system calls tracing utilities, like strace(1) or truss(1). This approach reveals low level details about a potential misbehavior of some part of the system.

3. Using an interactive debugger, like gdb(1). When the process is stuck, and you don't know what it was doing just before it has got stuck, with gdb you can attach to this process and print its calls stack, to reveal where the last call was made from. Just like with strace or truss you see the system call trace and not the Perl calls.

1.8.4.1 Using the Perl Trace

To see where an httpd is "spinning", try adding this to your script or a startup file:

```
use Carp ();
$SIG{'USR2'} = sub {
    Carp::confess("caught SIGUSR2!");
};
```

The above code assigns a signal handler for the `USR2` signal. This signal has been chosen because it's least likely to be used by the other parts of the server.

We check the registered signal handlers with help of Apache::Status. What we see at http://localhost/perl-status?sig is :

```
USR2 = \&MyStartUp::__ANON__
```

`MyStartUp` is the name of the package I've used in mine *startup.pl*.

After applying this server configuration, let's use this simple code example, where sleep(10000) will emulate a hanging process:

```
debug/perl_trace.pl
-------------------
$|=1;
print "Content-type:text/plain\r\n\r\n";
print "[$$] Going to sleep\n";
hanging_sub();
sub hanging_sub {sleep 10000;}
```

We execute the above script as *http://localhost/perl/debug/perl_trace.pl*, we have used `$|=1;` and printed the PID with `$$` to learn what process ID we want to work with.

No we issue the command line, using the PID we have just saw being printed to the browser's window:

```
% kill -USR2 PID
```

And watch this showing up at the *error_log* file:

```
caught SIGUSR2!
    at /home/httpd/perl/startup/startup.pl line 32
MyStartUp::__ANON__('USR2') called
    at /home/httpd/perl/debug/perl_trace.pl line 5
Apache::ROOT::perl::debug::perl_trace_2epl::hanging_sub() called
    at /home/httpd/perl/debug/perl_trace.pl line 4
Apache::ROOT::perl::debug::perl_trace_2epl::handler('Apache=SCALAR(0x8309d08)')
  called
    at /usr/lib/perl5/site_perl/5.005/i386-linux/Apache/Registry.pm
      line 140
eval {...} called
    at /usr/lib/perl5/site_perl/5.005/i386-linux/Apache/Registry.pm
      line 140
```

```
Apache::Registry::handler('Apache=SCALAR(0x8309d08)') called
    at PerlHandler subroutine 'Apache::Registry::handler' line 0
eval {...} called
    at PerlHandler subroutine 'Apache::Registry::handler' line 0
```

We can clearly see that the process "hangs" in the code executed at line 5 of the
/home/httpd/perl/debug/perl_trace.pl script, and it was called by the hanging_sub() routine defined at line
4.

1.8.4.2 Using the System Calls Trace

Depending on the operating system you should have one of the `truss(1)` or `strace(1)` utilities avail-
able. In the following examples we will use `strace(1)`.

There are two ways to get the trace of the process with strace(1) (similar to gdb(1)). The first one is to tell
strace(1) to start the process and do the tracing on it:

```
% strace perl -le 'print "mod_perl rules"'
```

The second is tell strace(1) to attach to the process that's already running. You need to know the PID of
the process.

```
% strace -p PID
```

Replace PID with the process number you want to check on.

There are many more useful arguments accepted by strace(1) that you might find useful. For example you
can tell it to trace only specific system calls:

```
% strace -e trace=open,write,close,nanosleep \
    perl -le 'print "mod_perl rules"'
```

In this example we have asked strace(1) to show us only the *open*, *write*, *close*, *nanosleep* which simplifies
the observing of the output generated by strace(1) if you know what you are looking for.

Let's write a mod_perl script that hangs, and deploy `strace(1)` to find the point it hangs at:

```
hangme.pl
---------
$|=1;
my $r = shift;
$r->send_http_header('text/plain');

print "PID = $$\n";

while(1){
  $i++;
  sleep 1;
}
```

The reason this simple code hangs is obvious. It never breaks from the while loop. As you have noticed, it prints the PID of the current process to the browser. Of course in a real situation you cannot use the same trick. In the previous section I have presented a few ways to detect the runaway processes and their PIDs.

I save the above code in a file and execute it from the browser. Note that I've made STDOUT unbuffered with $|=1; so I will immediately see the process ID. Once the script is requested, the script prints the process PID and obviously hangs. So we press the 'Stop' button, but the process continues to hang in this code. Isn't apache supposed to detect the broken connection and abort the request? *Yes* and *No*, you will understand soon what's really happening.

First let's attach to the process and see what it's doing. I use the PID the script printed to the browser, which is 10045 in this case:

```
% strace -p 10045

[...truncated identical output...]
SYS_175(0, 0xbffff41c, 0xbffff39c, 0x8, 0) = 0
SYS_174(0x11, 0, 0xbffff1a0, 0x8, 0x11) = 0
SYS_175(0x2, 0xbffff39c, 0, 0x8, 0x2)   = 0
nanosleep(0xbffff308, 0xbffff308, 0x401a61b4, 0xbffff308, 0xbffff41c) = 0
time([940973834])                       = 940973834
time([940973834])                       = 940973834
[...truncated the identical output...]
```

It isn't what we expected to see, is it? These are some system calls we don't see in our little example. What we actually see is how Perl translates our code into system calls. Since we know that our code hangs in this snippet:

```
while(1){
  $i++;
  sleep 1;
}
```

We *"easily"* figure out that the first three system calls implement the $i++, while the other three are responsible for the sleep 1 call.

Generally the situation is the reverse of our example. You detect the hanging process, you attach to it and watch the trace of calls it does (or the last few commands if the process is hanging waiting for something, e.g. when blocking on a file lock request). From watching the trace you figure out what it's actually doing, and probably find the corresponding lines in your Perl code. For example let's see how one process *"hangs"* while requesting an exclusive lock on a file exclusively locked by another process:

```
excl_lock.pl
---------
use Fcntl qw(:flock);
use Symbol;

if ( fork() ) {
  my $fh = gensym;
  open $fh, ">/tmp/lock" or die "cannot open /tmp/lock $!";
  print "$$: I'm going to obtain the lock\n";
  flock $fh, LOCK_EX;
```

```
    print "$$: I've got the lock\n";
    sleep 20;
    close $fh;

} else {
    my $fh = gensym;
    open $fh, ">/tmp/lock" or die "cannot open /tmp/lock $!";
    print "$$: I'm going to obtain the lock\n";
    flock $fh, LOCK_EX;
    print "$$: I've got the lock\n";
    sleep 20;
    close $fh;
}
```

The code is simple. The process executing the code forks a second process, and both do the same thing: generate a unique symbol to be used as a file handler, open the lock file for writing using the generated symbol, lock the file in exclusive mode, sleep for 20 seconds (pretending to do some lengthy operation) and close the lock file, which also unlocks the file.

The `gensym` function is imported from the `Symbol` module. The `Fcntl` module provides us with a symbolic constant `LOCK_EX`. This is imported via the `:flock` tag, which imports this and other flock() constants.

The code used by both processes is identical, therefore we cannot predict which one will get its hands on the lock file and succeed in locking it first, so we add print() statements to find the PID of the process blocking (waiting to get the lock) on a lock request.

When the above code executed from the command line, we see that one of the processes gets the lock:

```
% ./excl_lock.pl

3038: I'm going to obtain the lock
3038: I've got the lock
3037: I'm going to obtain the lock
```

Here we see that process 3037 is blocking, so we attach to it:

```
% strace -p 3037

about to attach c10
flock(3, LOCK_EX
```

It's clear from the above trace, that the process waits for an exclusive lock. (Note, that the missing closing parentheses is not a typo!)

As you become familiar with watching the traces of different processes, you will understand what is happening more easily.

1.8.4.3 Using the Interactive Debugger

Another approach to see a trace of the running code is to use a debugger such as `gdb` (the GNU debugger). It's supposed to work on any platform which supports the GNU development tools. Its purpose is to allow you to see what is going on *inside* a program while it executes, or what it was doing at the moment it crashed.

To trace the execution of a process, `gdb` needs to know the process id (PID) and the path to the binary that the process is executing. For Perl code it's */usr/bin/perl* (or whatever is the path to your Perl), for httpd processes it will be the path to your httpd executable.

Here are a few examples using gdb.

Let's go back to our last locking example, execute it as before and attach to the process that didn't get the lock:

```
% gdb /usr/bin/perl 3037
```

After starting the debugger we execute the `where` command to see the trace:

```
(gdb) where
#0  0x40131781 in __flock ()
#1  0x80a5421 in Perl_pp_flock ()
#2  0x80b148d in Perl_runops_standard ()
#3  0x80592b8 in perl_run ()
#4  0x805782f in main ()
#5  0x400a6cb3 in __libc_start_main (main=0x80577c0 <main>, argc=2,
    argv=0xbffff7f4, init=0x8056af4 <_init>, fini=0x80b14fc <_fini>,
    rtld_fini=0x4000a350 <_dl_fini>, stack_end=0xbffff7ec)
    at ../sysdeps/generic/libc-start.c:78
```

That's not what we expected to see and now it's a different trace. `#0` tells us the most recent call that was executed, which is a C language flock() implementation. But the previous call (`#1`) isn't print(), as we would expect, but a higher level of Perl's internal flock(). If we follow the trace of calls what we actually see is an Opcodes tree, which can be better presented as:

```
__libc_start_main
  main ()
    perl_run ()
      Perl_runops_standard ()
        Perl_pp_flock ()
          __flock ()
```

So I would say that it's less useful than `strace`, since if there are several flock()s it's almost impossible to know which of them was called. This problem is solved by `strace`, which shows the sequence of the system calls executed. Using this sequence we can locate the corresponding lines in the code.

(META: the above is wrong - you can ask to display the previous command executed by the program (not gdb)! What is it?)

When you attach to a running process with debugger, the program stops executing and control of the program is passed to the debugger. You can continue the normal program run with the `continue` command or execute it step by step with the `next` and `step` commands which you type at the `gdb` prompt. (`next` steps over any function calls in the line, while `step` steps into them).

C/C++ debuggers are a very large topic and beyond the scope of this document, but the gdb man page is quite good and you can try `info gdb` as well. You might also want to check the `ddd` (Data Display Debugger) which provides a visual interface to `gdb` and other debuggers. It even knows how to debug Perl programs!

For completeness, let's see the gdb trace of the httpd process that's still hanging in the `while(1)` loop of the first example in this section:

```
% gdb /usr/local/apache/bin/httpd 1005

(gdb) where
#0  0x4014a861 in __libc_nanosleep ()
#1  0x4014a7ed in __sleep (seconds=1) at ../sysdeps/unix/sysv/linux/sleep.c:78
#2  0x8122c01 in Perl_pp_sleep ()
#3  0x812b25d in Perl_runops_standard ()
#4  0x80d3721 in perl_call_sv ()
#5  0x807a46b in perl_call_handler ()
#6  0x8079e35 in perl_run_stacked_handlers ()
#7  0x8078d6d in perl_handler ()
#8  0x8091e43 in ap_invoke_handler ()
#9  0x80a5109 in ap_some_auth_required ()
#10 0x80a516c in ap_process_request ()
#11 0x809cb2e in ap_child_terminate ()
#12 0x809cd6c in ap_child_terminate ()
#13 0x809ce19 in ap_child_terminate ()
#14 0x809d446 in ap_child_terminate ()
#15 0x809dbc3 in main ()
#16 0x400d3cb3 in __libc_start_main (main=0x809d88c <main>, argc=1,
    argv=0xbffff7e4, init=0x80606f8 <_init>, fini=0x812b33c <_fini>,
    rtld_fini=0x4000a350 <_dl_fini>, stack_end=0xbffff7dc)
    at ../sysdeps/generic/libc-start.c:78
```

As before we can see a complete trace of the last executed call.

As you have noticed, I still haven't explained why the process hanging in the `while(1)` loop isn't aborted by Apache. The next section covers this.

To easily detect the hanging location, you can go through these steps while running gdb:

```
(gdb) where
(gdb) source ~/.gdbinit
(gdb) curinfo
```

(adjust the path to *.gdbinit* if needed.)

After loading the special macros file (*.gdbinit*) you can use the *curinfo* gdb macro to figure out the file and line number the code stuck in.

1.9 Debugging Hanging processes (continued)

META: incomplete

mod_perl comes with a number of useful of gdb macros to ease the debug process. You will find the file with macros in the mod_perl source distribution in the *.gdbinit* file (mod_perl-x.xx/.gdbinit). You might want to modify the macro definitions.

In order to use this you need to compile mod_perl with `PERL_DEBUG=1`.

To debug the server, start it:

```
% httpd -X
```

Issue a request to the offending script that hangs. Find the PID number of the process that hangs.

Go to the server root:

```
% cd /usr/local/apache
```

Now attach to it with gdb (replace the `PID` with the actual process id) and load the macros from *.gdbinit*:

```
% gdb /path/to/httpd PID
% source /usr/src/mod_perl-x.xx/.gdbinit
```

Now you can start the server (*httpd* below is a gdb macro):

```
(gdb) httpd
```

Now run the `curinfo` macro:

```
(gdb) curinfo
```

It should tell you the line/filename of the offending Perl code.

Add this to *.gdbinit*:

```
define longmess
  set $sv = perl_eval_pv("Carp::longmess()", 1)
  printf "%s\n", ((XPV*) ($sv)->sv_any )->xpv_pv
end
```

and when you reload the macros, run:

```
(gdb) longmess
```

to produce a Perl stacktrace.

1.9.1 Debugging core Dumping Code

```
    $ perl -e dump
     Abort(coredump)
```

META: should I move the `Apache::StatINC` here? (I think not, since it relates to other topics like reloading config files, but you should mention it here with a pointer to it)

1.10 PERL_DEBUG=1 Build Option

Building mod_perl with `PERL_DEBUG=1`:

```
  perl Makefile.PL PERL_DEBUG=1
```

will:

1. Add '-g' to EXTRA_CFLAGS

2. Turn on PERL_TRACE

3. Set PERL_DESTRUCT_LEVEL=2

4. Link against `libperld` if -e $Config{archlibexp}/CORE/libperld$Config{lib_ext}

1.11 Apache::Debug

(META: to be written)

```
  use Apache::Debug ();
  Apache::Debug::dump($r, SERVER_ERROR, "Uh Oh!");
```

This module sends what may be helpful debugging information to the client rather than to *error_log*.

Also, you could try using a larger emergency pool, try this instead of Apache::Debug:

```
$^M = 'a' x (1<<18);  #256k buffer
use Carp ();
$SIG{__DIE__} = \&Carp::confess;
eval { Carp::confess("init") };
```

1.12 Debug Tracing

To enable mod_perl debug tracing, configure mod_perl with the PERL_TRACE option:

```
 perl Makefile.PL PERL_TRACE=1
```

The trace levels can then be enabled via the `MOD_PERL_TRACE` environment variable which can contain any combination of:

- **c**

  Trace directive handling during *Apache* (non-mod_perl) **c**onfiguration directive handling. (Startup.)

- **d**

  Trace directive handling during *mod_perl* **d**irective processing during configuration read. (Startup.)

- **s**

  Trace processing of *<Perl>* **s**ections. (Startup.)

- **h**

  Trace Perl **h**andler callbacks. (RunTime.)

- **g**

  Trace **g**lobal variable handling, interpreter construction, `END` blocks, etc. (RunTime.)

- **all**

  all of the options listed above. (Startup + RunTime.)

One way of setting this variable is by adding this directive to *httpd.conf*:

```
 PerlSetEnv MOD_PERL_TRACE all
```

For example if you want to see a trace of the `PerlRequire` and `PerlModule` directives as they are executed, use:

```
 PerlSetEnv MOD_PERL_TRACE d
```

Of course you can use the command line environment setting:

```
 % setenv MOD_PERL_TRACE all
 % httpd -X
```

1.13 gdb says there are no debugging symbols

During *make install* Apache strips all the debugging symbols. To prevent this you should use *--without-execstrip* `./configure` option. So if you configure Apache via mod_perl, you should do:

```
panic% perl Makefile.PL USE_APACI=1 \
   APACI_ARGS='--without-execstrip' [other options]
```

Alternatively you can copy the unstripped binary manually. For example we did:

```
panic# cp apache_1.3.17/src/httpd /home/httpd/httpd_perl/bin/httpd_perl
```

As you know you need an unstripped executable to be able to debug it. While you can compile mod_perl with `-g` (or `PERL_DEBUG=1`), the Apache `install` strips the symbols.

Makefile.tmpl contains a line:

```
IFLAGS_PROGRAM  = -m 755 -s
```

Removing the -s does the trick (If you cannot find it in *Makefile.tmpl* do it directly in *Makefile*). Alternatively you rerun `make` and copy the unstripped httpd binary away.

1.14 Debugging Signal Handlers ($SIG{FOO})

The current Perl implementation does not restore the original Apache C handler when you use the `local $SIG{FOO}` clause. While the save/restore of `$SIG{ALRM}` was fixed in mod_perl 1.19_01 (SVN version), other signals are not yet fixed. The real fix should probably be in Perl itself.

Until recently `local $SIG{ALRM}` restored the `SIGALRM` handler to Perl's handler, not the handler it was in the first place (Apache's `alrm_handler()`). If you build mod_perl with `PERL_TRACE=1` and set the `MOD_PERL_TRACE` environment variable to **g**, you will see this in the *error_log* file:

```
mod_perl: saving SIGALRM (14) handler 0x80b1ff0
mod_perl: restoring SIGALRM (14) handler from: 0x0 to: 0x80b1ff0
```

If nobody has touched `$SIG{ALRM}`, `0x0` will be the same address as the others.

If you work with signal handlers you should take a look at the `Sys::Signal` module, which solves the problem:

`Sys::Signal` - Set signal handlers with restoration of the existing C sighandler. Get it from CPAN.

The usage is simple. If the original code was:

```
# If a timeout happens and C<SIGALRM> is thrown, the alarm() will be
# reset, otherwise C<alarm 0> is reached and timer is reset as well.
eval {
  local $SIG{ALRM} = sub { die "timeout\n" };
  alarm $timeout;
  ... db stuff ...
  alarm 0;
};
die $@ if $@;
```

Now you would write:

```
use Sys::Signal ();
eval {
  my $h = Sys::Signal->set(ALRM => sub { die "timeout\n" });
  alarm $timeout;
  ... do something that may timeout ...
  alarm 0;
};
die $@ if $@;
```

This should be fixed in Perl 5.6.1, so if you use this version of Perl, chances are that you don't need to use
`Sys::Signal`.

mod_perl tries to deal only with those signals that cause conflict with Apache's. Currently this is only
`SIGALRM`. If there is another one that gives you trouble, you can add it to the list in *perl_config.c* after
"ALRM", before *NULL*.

```
static char *sigsave[] = { "ALRM", NULL };
```

1.15 Code Profiling

(META: duplication??? I've started to write about profiling somewhere in this file)

It is possible to profile code run under mod_perl with the `Devel::DProf` module available on CPAN.
However, you must have apache version 1.3b3 or higher and the `PerlChildExitHandler` enabled.
When the server is started, `Devel::DProf` installs an `END` block (to write the `tmon.out` file) which
will be run when the server is shutdown. Here's how to start and stop a server with the profiler enabled:

```
% setenv PERL5OPT -d:DProf
% httpd -X -d `pwd` &
... make some requests to the server here ...
% kill `cat logs/httpd.pid`
% unsetenv PERL5OPT
% dprofpp
```

See also: `Apache::DProf`

1.16 Devel::Peek

Devel::Peek - A data debugging tool for the XS programmer

Let's see an example of Perl allocating a buffer only once, regardless of my () scoping, although it will
realloc() if the size is bigger than `SvLEN`:

```
use Devel::Peek;

for (1..3) {
    foo();
}
```

```
sub foo {
    my $sv;
    Dump $sv;
    $sv = 'x' x 100_000;
    $sv = "";
}
```

The output:

```
SV = NULL(0x0) at 0x8138008
  REFCNT = 1
  FLAGS = (PADBUSY,PADMY)
SV = PV(0x80e5794) at 0x8138008
  REFCNT = 1
  FLAGS = (PADBUSY,PADMY)
  PV = 0x815f808 ""\0
  CUR = 0
  LEN = 100001
SV = PV(0x80e5794) at 0x8138008
  REFCNT = 1
  FLAGS = (PADBUSY,PADMY)
  PV = 0x815f808 ""\0
  CUR = 0
```

We can see that on the second and subsequent calls `$sv` already has previously allocated memory.

So, if you can afford the memory, a larger buffer means fewer `brk()` syscalls. If you watch that example with `strace` you will only see calls to `brk()` the first time through the loop. So this is a case where your module might want to pre-allocate the buffer like this:

```
package Your::Proxy;

my $buffer = ' ' x 100_000;
$buffer = "";
```

Now only the parent has to brk() at server startup, each child already will already have an allocated buffer. Just reset to "" when you are done.

Note: Previously allocating a scalar in this way saves reallocation in v5.005 but may not do so in other versions.

1.17 How can I find out if a mod_perl code has a memory leak

The `Apache::Leak` module (derived from `Devel::Leak`) should help you detecting the leakages in your code. For example:

```
leaktest.pl
-----------
use Apache::Leak;

my $global = "FooAAA";

leak_test {
  $$global = 1;
  ++$global;
};
```

The argument to `leak_test()` is an anonymous sub, so you can just throw it any code you suspect might be leaking. Beware, it will run the code twice! The first time in, new `SVs` are created, but does not mean you are leaking. The second pass will give better evidence. You do not need to be inside mod_perl to use it. From the command line, the above script outputs:

```
ENTER: 1482 SVs
new c28b8 : new c2918 :
LEAVE: 1484 SVs
ENTER: 1484 SVs
new db690 : new db6a8 :
LEAVE: 1486 SVs
!!! 2 SVs leaked !!!
```

Build a debuggable Perl to see dumps of the `SVs`. The simple way to have both a normal Perl and debuggable Perl is to follow hints in the `SUPPORT` doc for building `libperld.a`. When that is built, copy the `perl` from that directory to your Perl bin directory, but name it `dperl`.

Our example's leak explanation: `$$global = 1;` : new global variable `FooAAA` created with value of `1`, this will not be destroyed until this module is destroyed. Under mod_perl the module doesn't get destroyed until the process quits.

`Apache::Leak` is not very user-friendly, have a look at `B::LexInfo`. It is possible to see something that might appear to be a leak, but is actually just a Perl optimization. e.g. consider this code:

```
sub foo {
  my $string = shift;
}

foo("a string");
```

`B::LexInfo` will show you that Perl does not release the value from $string, unless you undef() it. This is because Perl anticipates the memory will be needed for another string, the next time the subroutine is entered. You'll see similar behaviour for `@array` length, `%hash` keys, and scratch areas of the pad-list for OPs such as `join()`, '.', etc.

`Apache::Status` includes a `StatusLexInfo` option which can show you the internals of your code.

1.18 Debugging your code in Single Server Mode

Running in httpd -X mode is good only for testing during the development phase.

You want to test that your application correctly handles global variables (if you have any - the less you have of them the better of course - but sometimes you just can't do without them). It's hard to test with multiple servers serving your cgi since each child has a different value for its global variables. Imagine that you have a `random()` sub that returns a random number and you have the following script.

```
use vars qw($num);
$num ||= random();
print ++$num;
```

This script initializes the variable `$num` with a random value, then increments it on each request and prints it out. Running this script in a multiple server environments will result in something like 1, 9, 4, 19 (a different number each time you hit the browser's reload button) since each time your script will be served by a different child. (On some operating systems, e.g. AIX, the parent httpd process will assign all of the requests to the same child process if all of the children are idle). But if you run in `httpd -X` single server mode you will get 2, 3, 4, 5... (assuming that `random()` returned 1 at the first call)

But do not get too obsessive with this mode, since working in single server mode sometimes hides problems that show up when you switch to normal (multi-server) mode.

Consider an application that allows you to change the configuration at run time. Let's say the script produces a form to change the background color of the page. It's not good design, but for the sake of demonstrating the potential problem we will assume that our script doesn't write the changed background color to the disk, but simply changes it in memory, like this:

```
use vars qw($bgcolor);
  # assign default value at first invocation
$bgcolor ||= "white";
  # modify the color if requested to
$bgcolor = $q->param('bgcolor') || $bgcolor;
```

So you have typed in a new color, and in response, your script prints back the html with a new color - you think that's it! It was so simple. If you keep running in single server mode you will never notice that you have a problem...

If you run the same code in normal server mode, after you submit the color change you will get the result as expected, but when you call the same URL again (not reload!) the chances are that you will get back the original default color (white in our case), since only the child which processed the color change request knows about the global variable change. Just remember that children can't share information, other than that which they inherited from their parent on their birth. Of course you could use a hidden variable for the color to be remembered, or store it on the server side (database, shared memory, etc).

If you use the Netscape client while your server is running in single-process mode, if the output returns HTML with <IMG> tags, then the loading of the images will take a long time, since Netscape's `KeepAlive` feature gets in the way. Netscape tries to open multiple connections and keep them open. Because there is only one server process listening, each connection has to time-out before the next

succeeds. Turn off `KeepAlive` in *httpd.conf* to avoid this effect. Alternatively (assuming you use the image size parameters, so that Netscape will be able to render the rest of the page) you can press **STOP** after a few seconds.

In addition you should be aware that when running with `-X` you will not see the status messages that the parent server normally writes to the error_log. ("server started", "server stopped", etc.). Since `httpd -X` causes the server to handle all requests itself, without forking any children, there is no controlling parent to write the status messages.

1.19 Apache::DumpHeaders - Watch HTTP Transaction Via Headers

This module is used to watch an HTTP transaction, looking at client and servers headers.

With `Apache::ProxyPassThru` configured, you are able to watch your browser talk to any server besides the one with this module living inside.

`Apache::DumpHeaders` has the ability to filter on IP addresses, has an interface for other modules to decide if the headers should be dumped or not and a function to only dump *n%* of the transactions.

For more information read the module's manpage.

Download the module from CPAN.

1.20 Apache::DebugInfo - Log Various Bits Of Per-Request Data

`Apache::DebugInfo` offers the ability to monitor various bits of per-request data. Its functionality is similar to Apache::DumpHeaders while offering several additional features, including the ability to:

- **- separate inbound from outbound HTTP headers**
- **- view the contents of $r->notes and $r->pnotes**
- **- view any of these at the various points in the request cycle**
- **- add output for any request phase from a single entry point**
- **- use as a PerlInitHandler or with direct method calls**
- **- use partial IP addresses for filtering by IP**
- **- offer a subclassable interface**

See the module's manpage for more details.

1.21 Maintainers

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

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

1.22 Authors

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

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

Table of Contents: