CHAPTER 10

Improving Performance with Shared Memory and Proper Forking

In this chapter we will talk about two issues that play an important role in optimizing server performance: sharing memory and forking.

Firstly, mod_perl Apache processes can become quite large, and it is therefore very important to make sure that the memory used by the Apache processes is shared between them as much as possible.

Secondly, if you need the Apache processes to fork new processes, it is important to perform the `fork()` calls in the proper way.

Sharing Memory

The sharing of memory is a very important factor. If your OS supports it (and most sane systems do), a lot of memory can be saved by sharing it between child processes. This is possible only when code is preloaded at server startup. However, during a child process's life, its memory pages tend to become unshared. Here is why.

There is no way to make Perl allocate memory so that (dynamic) variables land on different memory pages from constants or the rest of your code (which is really just data to the Perl interpreter), so the *copy-on-write* effect (explained in a moment) will hit almost at random.

If many modules are preloaded, you can trade off the memory that stays shared against the time for an occasional fork of a new Apache child by tuning the `MaxRequestsPerChild` Apache directive. Each time a child reaches this upper limit and dies, it will release its unshared pages. The new child will have to be forked, but it will share its fresh pages until it writes on them (when some variable gets modified).

The ideal is a point where processes usually restart before too much memory becomes unshared. You should take some measurements, to see if it makes a real difference and to find the range of reasonable values. If you have success with this tuning, bear in mind that the value of `MaxRequestsPerChild` will probably be specific to your situation and may change with changing circumstances.

It is very important to understand that the goal is not necessarily to have the highest `MaxRequestsPerChild` that you can. Having a child serve 300 requests on precompiled code is already a huge overall speedup. If this value also provides a substantial memory saving, that benefit may outweigh using a higher `MaxRequestsPerChild` value.

A newly forked child inherits the Perl interpreter from its parent. If most of the Perl code is preloaded at server startup, then most of this preloaded code is inherited from the parent process too. Because of this, less RAM has to be written to create the process, so it is ready to serve requests very quickly.

During the life of the child, its memory pages (which aren't really its own to start with—it uses the parent's pages) gradually get *dirty*—variables that were originally inherited and shared are updated or modified—and *copy-on-write* happens. This reduces the number of shared memory pages, thus increasing the memory requirement. Killing the child and spawning a new one allows the new child to use the pristine shared memory of the parent process.

The recommendation is that `MaxRequestsPerChild` should not be too large, or you will lose some of the benefit of sharing memory. With memory sharing in place, you can run many more servers than without it. In Chapter 11 we will devise a formula to calculate the optimum value for the `MaxClients` directive when sharing is taking place.

As we mentioned in Chapter 9, you can find the size of the shared memory by using the *ps(1)* or *top(1)* utilities, or by using the `GTop` module:

```
use GTop ();
print "Shared memory of the current process: ",
    GTop->new->proc_mem($$)->share, "\n";

print "Total shared memory: ",
    GTop->new->mem->share, "\n";
```

Calculating Real Memory Usage

We have shown how to measure the size of the process's shared memory, but we still want to know what the real memory usage is. Obviously this cannot be calculated simply by adding up the memory size of each process, because that wouldn't account for the shared memory.

On the other hand, we cannot just subtract the shared memory size from the total size to get the real memory-usage numbers, because in reality each process has a different history of processed requests, which makes different memory pages dirty; therefore, different processes have different memory pages shared with the parent process.

So how do we measure the real memory size used by all running web-server processes? It is a difficult task—probably too difficult to make it worthwhile to find the exact number—but we have found a way to get a fair approximation.

This is the calculation technique that we have devised:

1. Calculate all the unshared memory, by summing up the difference between shared and system memory of each process. To calculate a difference for a single process, use:

```
use GTop;
my $proc_mem = GTop->new->proc_mem($$);
my $diff     = $proc_mem->size - $proc_mem->share;
print "Difference is $diff bytes\n";
```

2. Add the system memory use of the parent process, which already includes the shared memory of all other processes.

Figure 10-1 helps to visualize this.

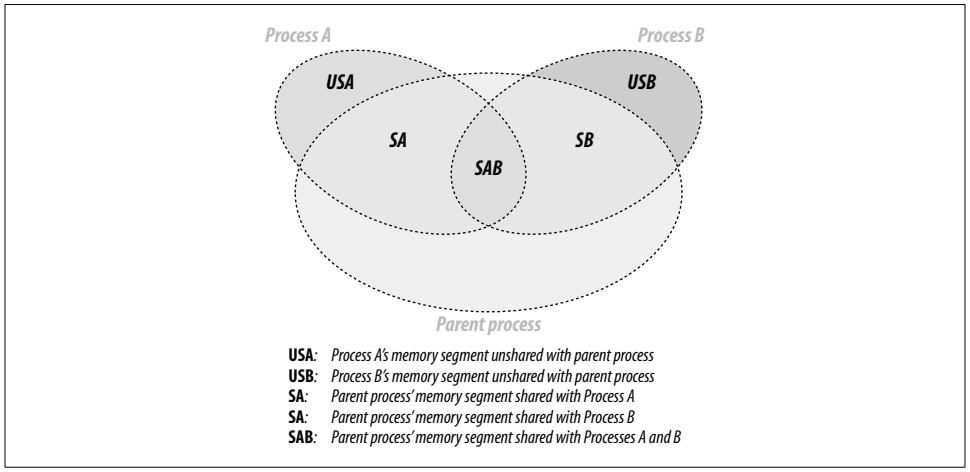

Figure 10-1. Child processes sharing memory with the parent process

The `Apache::VMonitor` module uses this technique to display real memory usage. In fact, it makes no separation between the parent and child processes. They are all counted indifferently using the following code:

```
use GTop ();
my $gtop = GTop->new;
my ($parent_pid, @child_pids) = some_code();
# add the parent proc memory size
my $total_real = $gtop->proc_mem($parent_pid)->size;
# add the unshared memory sizes
for my $pid (@child_pids) {
    my $proc_mem = $gtop->proc_mem($pid);
    $total_real += $proc_mem->size - $proc_mem->share;
}
```

Now `$total_real` contains approximately the amount of memory really used.

This method has been verified in the following way. We calculate the real memory used using the technique described above. We then look at the system memory report for the total memory usage. We then stop Apache and look at the total memory usage for a second time. We check that the system memory usage report indicates that the total memory used by the whole system has gone down by about the same number that we've calculated.

Note that some OSes do smart memory-page caching, so you may not see the memory usage decrease immediately when you stop the server, even though it is actually happening. Also, if your system is swapping, it's possible that your swap memory was used by the server as well as the real memory. Therefore, to get the verification right you should use a tool that reports real memory usage, cached memory, and swap memory. For example, on Linux you can use the *free* command. Run this command before and after stopping the server, then compare the numbers reported in the column called *free*.

Based on this logic we can devise a formula for calculating the maximum possible number of child processes, taking into account the shared memory. From now on, instead of adding the memory size of the parent process, we are going to add the maximum shared size of the child processes, and the result will be approximately the same. We do that approximation because the size of the parent process is usually unknown during the calculation.

Therefore, the formula to calculate the maximum number of child processes with minimum shared memory size of `Min_Shared_RAM_per_Child` MB that can run simultaneously on a machine that has a total RAM of `Total_RAM` MB available for the web server, and knowing the maximum process size, is:

$$MaxClients = \frac{\text{Total\_RAM - Min\_Shared\_RAM\_per\_Child}}{\text{Max\_Process\_Size - Min\_Shared\_RAM\_per\_Child}}$$

which can also be rewritten as:

$$MaxClients = \frac{\text{Total\_RAM - Shared\_RAM\_per\_Child}}{\text{Max\_UnShared\_RAM\_per\_Child}}$$

since the denominator is really the maximum possible amount of a child process's unshared memory.

In Chapter 14 we will see how we can enforce the values used in calculation during runtime.

Memory-Sharing Validation

How do you find out if the code you write is shared between processes or not? The code should remain shared, except when it is on a memory page used by variables that change. As you know, a variable becomes unshared when a process modifies its

value, and so does the memory page it resides on, because the memory is shared in memory-page units.

Sometimes you have variables that use a lot of memory, and you consider their usage read-only and expect them to be shared between processes. However, certain operations that seemingly don't modify the variable values do modify things internally, causing the memory to become unshared.

Imagine that you have a 10 MB in-memory database that resides in a single variable, and you perform various operations on it and want to make sure that the variable is still shared. For example, if you do some regular expression (regex)–matching processing on this variable and you want to use the `pos( )` function, will it make the variable unshared or not? If you access the variable once as a numerical value and once as a string value, will the variable become unshared?

The `Apache::Peek` module comes to the rescue.

Variable unsharing caused by regular expressions

Let's write a module called `Book::MyShared`, shown in Example 10-1, which we will preload at server startup so that all the variables of this module are initially shared by all children.

Example 10-1. Book/MyShared.pm

```
package Book::MyShared;
use Apache::Peek;

my $readonly = "Chris";

sub match     { $readonly =~ /\w/g;               }
sub print_pos { print "pos: ",pos($readonly),"\n";}
sub dump      { Dump($readonly);                  }
1;
```

This module declares the package `Book::MyShared`, loads the `Apache::Peek` module and defines the lexically scoped `$readonly` variable. In most instances, the `$readonly` variable will be very large (perhaps a huge hash data structure), but here we will use a small variable to simplify this example.

The module also defines three subroutines: `match( )`, which does simple character matching; `print_pos( )`, which prints the current position of the matching engine inside the string that was last matched; and finally `dump( )`, which calls the `Apache::Peek` module's `Dump( )` function to dump a raw Perl representation of the `$readonly` variable.

Now we write a script (Example 10-2) that prints the process ID (PID) and calls all three functions. The goal is to check whether `pos( )` makes the variable dirty and therefore unshared.

Example 10-2. share_test.pl

```
use Book::MyShared;
print "Content-type: text/plain\n\n";
print "PID: $$\n";
Book::MyShared::match( );
Book::MyShared::print_pos( );
Book::MyShared::dump( );
```

Before you restart the server, in *httpd.conf*, set:

```
MaxClients 2
```

for easier tracking. You need at least two servers to compare the printouts of the test program. Having more than two can make the comparison process harder.

Now open two browser windows and issue requests for this script in each window, so that you get different PIDs reported in the two windows and so that each process has processed a different number of requests for the *share_test.pl* script.

In the first window you will see something like this:

```
PID: 27040
pos: 1
SV = PVMG(0x853db20) at 0x8250e8c
  REFCNT = 3
  FLAGS = (PADBUSY,PADMY,SMG,POK,pPOK)
  IV = 0
  NV = 0
  PV = 0x8271af0 "Chris"\0
  CUR = 5
  LEN = 6
  MAGIC = 0x853dd80
    MG_VIRTUAL = &vtbl_mglob
    MG_TYPE = 'g'
    MG_LEN = 1
```

And in the second window:

```
PID: 27041
pos: 2
SV = PVMG(0x853db20) at 0x8250e8c
  REFCNT = 3
  FLAGS = (PADBUSY,PADMY,SMG,POK,pPOK)
  IV = 0
  NV = 0
  PV = 0x8271af0 "Chris"\0
  CUR = 5
  LEN = 6
  MAGIC = 0x853dd80
    MG_VIRTUAL = &vtbl_mglob
    MG_TYPE = 'g'
    MG_LEN = 2
```

All the addresses of the supposedly large data structure are the same (`0x8250e8c` and `0x8271af0`)—therefore, the variable data structure is almost completely shared. The

only difference is in the SV.MAGIC.MG_LEN record, which is not shared. This record is used to track where the last m//g match left off for the given variable, (e.g., by pos( )) and therefore it cannot be shared. See the *perlre* manpage for more information.

Given that the $readonly variable is a big one, its value is still shared between the processes, while part of the variable data structure is nonshared. The nonshared part is almost insignificant because it takes up very little memory space.

If you need to compare more than one variable, doing it by hand can be quite time consuming and error prone. Therefore, it's better to change the test script to dump the Perl datatypes into files (e.g., */tmp/dump.$$*, where *$$* is the PID of the process). Then you can use the *diff(1)* utility to see whether there is some difference.

Changing the dump( ) function to write the information to a file will do the job. Notice that we use Devel::Peek and not Apache::Peek, so we can easily reroute the STDERR stream into a file. In our example, when Devel::Peek tries to print to STDERR, it actually prints to our file. When we are done, we make sure to restore the original STDERR file handle.

The resulting code is shown in Example 10-3.

Example 10-3. Book/MyShared2.pm

```
package Book::MyShared2;
use Devel::Peek;

my $readonly = "Chris";

sub match     { $readonly =~ /\w/g;               }
sub print_pos { print "pos: ",pos($readonly),"\n";}
sub dump {
    my $dump_file = "/tmp/dump.$$";
    print "Dumping the data into $dump_file\n";
    open OLDERR, ">&STDERR";
    open STDERR, ">$dump_file" or die "Can't open $dump_file: $!";
    Dump($readonly);
    close STDERR ;
    open STDERR, ">&OLDERR";
}
1;
```

Now we modify our script to use the modified module, as shown in Example 10-4.

Example 10-4. share_test2.pl

```
use Book::MyShared2;
print "Content-type: text/plain\n\n";
print "PID: $$\n";
Book::MyShared2::match( );
Book::MyShared2::print_pos( );
Book::MyShared2::dump( );
```

Now we can run the script as before (with `MaxClients 2`). Two dump files will be created in the directory */tmp*. In our test these were created as */tmp/dump.1224* and */tmp/dump.1225*. When we run *diff(1)*:

```
panic% diff -u /tmp/dump.1224 /tmp/dump.1225
12c12
-       MG_LEN = 1
+       MG_LEN = 2
```

we see that the two padlists (of the variable `$readonly`) are different, as we observed before, when we did a manual comparison.

If we think about these results again, we come to the conclusion that there is no need for two processes to find out whether the variable gets modified (and therefore unshared). It's enough just to check the data structure twice, before the script was executed and again afterward. We can modify the `Book::MyShared2` module to dump the padlists into a different file after each invocation and then to run *diff(1)* on the two files.

Suppose you have some lexically scoped variables (i.e., variables declared with `my()`) in an `Apache::Registry` script. If you want to watch whether they get changed between invocations inside one particular process, you can use the `Apache::RegistryLexInfo` module. It does exactly that: it takes a snapshot of the padlist before and after the code execution and shows the difference between the two. This particular module was written to work with `Apache::Registry` scripts, so it won't work for loaded modules. Use the technique we described above for any type of variables in modules and scripts.

Another way of ensuring that a scalar is read-only and therefore shareable is to use either the `constant` pragma or the `readonly` pragma, as shown in Example 10-5. But then you won't be able to make calls that alter the variable even a little, such as in the example that we just showed, because it will be a true constant variable and you will get a compile-time error if you try this.

Example 10-5. Book/Constant.pm

```
package Book::Constant;
use constant readonly => "Chris";

sub match     { readonly =~ /\w/g;               }
sub print_pos { print "pos: ",pos(readonly),"\n";}
1;

panic% perl -c Book/Constant.pm

Can't modify constant item in match position at Book/Constant.pm
line 5, near "readonly)"
Book/Constant.pm had compilation errors.
```

However, the code shown in Example 10-6 is OK.

Example 10-6. Book/Constant1.pm

```
package Book::Constant1;
use constant readonly => "Chris";

sub match { readonly =~ /\w/g; }
1;
```

It doesn't modify the variable flags at all.

Numerical versus string access to variables

Data can get unshared on read as well—for example, when a numerical variable is accessed as a string. Example 10-7 shows some code that proves this.

Example 10-7. numerical_vs_string.pl

```
#!/usr/bin/perl -w

use Devel::Peek;
my $numerical = 10;
my $string    = "10";
$|=1;

dump_numerical();
read_numerical_as_numerical();
dump_numerical();
read_numerical_as_string();
dump_numerical();

dump_string();
read_string_as_numerical();
dump_string();
read_string_as_string();
dump_string();

sub read_numerical_as_numerical {
    print "\nReading numerical as numerical: ", int($numerical), "\n";
}
sub read_numerical_as_string {
    print "\nReading numerical as string: ", "$numerical", "\n";
}
sub read_string_as_numerical {
    print "\nReading string as numerical: ", int($string), "\n";
}
sub read_string_as_string {
    print "\nReading string as string: ", "$string", "\n";
}
sub dump_numerical {
    print "\nDumping a numerical variable\n";
    Dump($numerical);
}
sub dump_string {
    print "\nDumping a string variable\n";
```

Example 10-7. numerical_vs_string.pl (continued)

```
    Dump($string);
}
```

The test script defines two lexical variables: a number and a string. Perl doesn't have strong data types like C does; Perl's scalar variables can be accessed as strings and numbers, and Perl will try to return the equivalent numerical value of the string if it is accessed as a number, and vice versa. The initial internal representation is based on the initially assigned value: a numerical value* in the case of `$numerical` and a string value† in the case of `$string`.

The script accesses `$numerical` as a number and then as a string. The internal representation is printed before and after each access. The same test is performed with a variable that was initially defined as a string (`$string`).

When we run the script, we get the following output:

```
Dumping a numerical variable
SV = IV(0x80e74c0) at 0x80e482c
  REFCNT = 4
  FLAGS = (PADBUSY,PADMY,IOK,pIOK)
  IV = 10

Reading numerical as numerical: 10

Dumping a numerical variable
SV = PVNV(0x810f960) at 0x80e482c
  REFCNT = 4
  FLAGS = (PADBUSY,PADMY,IOK,NOK,pIOK,pNOK)
  IV = 10
  NV = 10
  PV = 0

Reading numerical as string: 10

Dumping a numerical variable
SV = PVNV(0x810f960) at 0x80e482c
  REFCNT = 4
  FLAGS = (PADBUSY,PADMY,IOK,NOK,POK,pIOK,pNOK,pPOK)
  IV = 10
  NV = 10
  PV = 0x80e78b0 "10"\0
  CUR = 2
  LEN = 28

Dumping a string variable
SV = PV(0x80cb87c) at 0x80e8190
```

* `IV`, for signed integer value, or a few other possible types for floating-point and unsigned integer representations.

† `PV`, for pointer value (`SV` is already taken by a scalar data type)

```
  REFCNT = 4
  FLAGS = (PADBUSY,PADMY,POK,pPOK)
  PV = 0x810f518 "10"\0
  CUR = 2
  LEN = 3

Reading string as numerical: 10

Dumping a string variable
SV = PVNV(0x80e78d0) at 0x80e8190
  REFCNT = 4
  FLAGS = (PADBUSY,PADMY,NOK,POK,pNOK,pPOK)
  IV = 0
  NV = 10
  PV = 0x810f518 "10"\0
  CUR = 2
  LEN = 3

Reading string as string: 10

Dumping a string variable
SV = PVNV(0x80e78d0) at 0x80e8190
  REFCNT = 4
  FLAGS = (PADBUSY,PADMY,NOK,POK,pNOK,pPOK)
  IV = 0
  NV = 10
  PV = 0x810f518 "10"\0
  CUR = 2
  LEN = 3
```

We know that Perl does the conversion from one type to another on the fly, and that's where the variables get modified—during the automatic conversion behind the scenes. From this simple test you can see that variables may change internally when accessed in different contexts. Notice that even when a numerical variable is accessed as a number for the first time, its internals change, as Perl has intialized its `PV` and `NV` fields (the string and floating-point represenations) and adjusted the `FLAGS` fields.

From this example you can clearly see that if you want your variables to stay shared and there is a chance that the same variable will be accessed both as a string and as a numerical value, you have to access this variable as a numerical and as a string, as in the above example, before the fork happens (e.g., in the startup file). This ensures that the variable will be shared if no one modifies its value. Of course, if some other variable in the same page happens to change its value, the page will become unshared anyway.

Preloading Perl Modules at Server Startup

As we just explained, to get the code-sharing effect, you should preload the code before the child processes get spawned. The right place to preload modules is at server startup.

You can use the PerlRequire and PerlModule directives to load commonly used modules such as CGI.pm and DBI when the server is started. On most systems, server children will be able to share the code space used by these modules. Just add the following directives into *httpd.conf*:

```
PerlModule CGI
PerlModule DBI
```

An even better approach is as follows. First, create a separate startup file. In this file you code in plain Perl, loading modules like this:

```
use DBI ();
use Carp ();
1;
```

(When a module is loaded, it may export symbols to your package namespace by default. The empty parentheses ( ) after a module's name prevent this. Don't forget this, unless you need some of these in the startup file, which is unlikely. It will save you a few more kilobytes of memory.)

Next, require( ) this startup file in *httpd.conf* with the PerlRequire directive, placing the directive before all the other mod_perl configuration directives:

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

As usual, we provide some numbers to prove the theory. Let's conduct a memory-usage test to prove that preloading reduces memory requirements.

To simplify the measurement, we will use only one child process. We will use these settings in *httpd.conf*:

```
MinSpareServers 1
MaxSpareServers 1
StartServers 1
MaxClients 1
MaxRequestsPerChild 100
```

We are going to use *memuse.pl* (shown in Example 10-8), an Apache::Registry script that consists of two parts: the first one loads a bunch of modules (most of which aren't going to be used); the second reports the memory size and the shared memory size used by the single child process that we start, and the difference between the two, which is the amount of unshared memory.

Example 10-8. memuse.pl

```
use strict;
use CGI ();
use DB_File ();
use LWP::UserAgent ();
use Storable ();
use DBI ();
use GTop ();
```

Example 10-8. memuse.pl (continued)

```
my $r = shift;
$r->send_http_header('text/plain');
my $proc_mem = GTop->new->proc_mem($$);
my $size  = $proc_mem->size;
my $share = $proc_mem->share;
my $diff  = $size - $share;
printf "%10s %10s %10s\n", qw(Size Shared Unshared);
printf "%10d %10d %10d (bytes)\n", $size, $share, $diff;
```

First we restart the server and execute this CGI script with none of the above modules preloaded. Here is the result:

```
Size     Shared   Unshared
4706304  2134016  2572288 (bytes)
```

Now we take the following code:

```
use strict;
use CGI ();
use DB_File ();
use LWP::UserAgent ();
use Storable ();
use DBI ();
use GTop ();
1;
```

and copy it into the *startup.pl* file. The script remains unchanged. We restart the server (now the modules are preloaded) and execute it again. We get the following results:

```
Size     Shared   Unshared
4710400  3997696  712704 (bytes)
```

Let's put the two results into one table:

```
Preloading    Size   Shared  Unshared
-----------------------------------------------
Yes        4710400  3997696    712704 (bytes)
No         4706304  2134016   2572288 (bytes)
-----------------------------------------------
Difference    4096  1863680  -1859584
```

You can clearly see that when the modules weren't preloaded, the amount of shared memory was about 1,864 KB smaller than in the case where the modules were preloaded.

Assuming that you have 256 MB dedicated to the web server, if you didn't preload the modules, you could have 103 servers:

```
268435456 = X * 2572288 + 2134016

X = (268435456 - 2134016) / 2572288 = 103
```

(Here we have used the formula that we devised earlier in this chapter.)

Now let's calculate the same thing with the modules preloaded:

```
268435456 = X * 712704 + 3997696

X = (268435456 - 3997696) / 712704 = 371
```

You can have almost four times as many servers!!!

Remember, however, that memory pages get dirty, and the amount of shared memory gets smaller with time. We have presented the ideal case, where the shared memory stays intact. Therefore, in use, the real numbers will be a little bit different.

Since you will use different modules and different code, obviously in your case it's possible that the process sizes will be bigger and the shared memory smaller, and vice versa. You probably won't get the same ratio we did, but the example certainly shows the possibilities.

Preloading Registry Scripts at Server Startup

Suppose you find yourself stuck with self-contained Perl CGI scripts (i.e., all the code placed in the CGI script itself). You would like to preload modules to benefit from sharing the code between the children, but you can't or don't want to move most of the stuff into modules. What can you do?

Luckily, you can preload scripts as well. This time the `Apache::RegistryLoader` module comes to your aid. `Apache::RegistryLoader` compiles `Apache::Registry` scripts at server startup.

For example, to preload the script */perl/test.pl*, which is in fact the file */home/httpd/perl/test.pl*, you would do the following:

```
use Apache::RegistryLoader ();
Apache::RegistryLoader->new->handler("/perl/test.pl",
                             "/home/httpd/perl/test.pl");
```

You should put this code either in `<Perl>` sections or in a startup script.

But what if you have a bunch of scripts located under the same directory and you don't want to list them one by one? Then the `File::Find` module will do most of the work for you.

The script shown in Example 10-9 walks the directory tree under which all `Apache::Registry` scripts are located. For each file with the extension *.pl*, it calls the `Apache::RegistryLoader::handler()` method to preload the script in the parent server. This happens before Apache pre-forks the child processes.

Example 10-9. startup_preload.pl

```
use File::Find qw(finddepth);
use Apache::RegistryLoader ();
{
    my $scripts_root_dir = "/home/httpd/perl/";
```

Example 10-9. startup_preload.pl (continued)

```
    my $rl = Apache::RegistryLoader->new;
    finddepth(
        sub {
            return unless /\.pl$/;
            my $url = $File::Find::name;
            $url =~ s|$scripts_root_dir/?|/|;
            warn "pre-loading $url\n";
            # preload $url
            my $status = $rl->handler($url);
            unless($status == 200) {
                warn "pre-load of '$url' failed, status=$status\n";
           }
        },
        $scripts_root_dir
    );
}
```

Note that we didn't use the second argument to `handler()` here, as we did in the first example. To make the loader smarter about the URI-to-filename translation, you might need to provide a `trans()` function to translate the URI to a filename. URI-to-filename translation normally doesn't happen until an HTTP request is received, so the module is forced to do its own translation. If the filename is omitted and a `trans()` function is not defined, the loader will try to use the URI relative to the ServerRoot.

A simple `trans()` function can be something like this:

```
sub mytrans {
    my $uri = shift;
    $uri =~ s|^/perl/|/home/httpd/perl/|;
    return $uri;
}
```

You can easily derive the right translation by looking at the `Alias` directive. The above `mytrans()` function matches our `Alias`:

```
Alias /perl/ /home/httpd/perl/
```

After defining the URI-to-filename translation function, you should pass it during the creation of the `Apache::RegistryLoader` object:

```
my $rl = Apache::RegistryLoader->new(trans => \&mytrans);
```

We won't show any benchmarks here, since the effect is just like preloading modules. However, we will use this technique later in this chapter, when we will need to have a fair comparison between `PerlHandler` code and `Apache::Registry` scripts. This will require both the code and the scripts to be preloaded at server startup.

Module Initialization at Server Startup

It's important to preload modules and scripts at server startup. But for some modules this isn't enough, and you have to prerun their initialization code to get more

memory pages shared. Usually you will find information about specific modules in their respective manpages. We will present a few examples of widely used modules where the code needs to be initialized.

Initializing DBI.pm

The first example is the DBI module. DBI works with many database drivers from the DBD:: category (e.g., DBD::mysql). If you want to minimize memory use after Apache forks its children, it's not enough to preload DBI—you must initialize DBI with the driver(s) that you are going to use (usually a single driver is used). Note that you should do this only under mod_perl and other environments where sharing memory is very important. Otherwise, you shouldn't initialize drivers.

You probably already know that under mod_perl you should use the Apache::DBI module to get persistent database connections (unless you open a separate connection for each user). Apache::DBI automatically loads DBI and overrides some of its methods. You should continue coding as if you had loaded only the DBI module.

As with preloading modules, our goal is to find the configuration that will give the smallest difference between the shared and normal memory reported, and hence the smallest total memory usage.

To simplify the measurements, we will again use only one child process. We will use these settings in *httpd.conf*:

```
MinSpareServers 1
MaxSpareServers 1
StartServers 1
MaxClients 1
MaxRequestsPerChild 100
```

We always preload these modules:

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

We are going to run memory benchmarks on five different versions of the *startup.pl* file:

Version 1
: Leave the file unmodified.

Version 2
: Install the MySQL driver (we will use the MySQL RDBMS for our test):

```
DBI->install_driver("mysql");
```

It's safe to use this method—as with use( ), if it can't be installed, it will die( ).

Version 3
: Preload the MySQL driver module:

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

Version 4

Tell `Apache::DBI` to connect to the database when the child process starts (`ChildInitHandler`). No driver is preloaded before the child is spawned!

```
Apache::DBI->connect_on_init('DBI:mysql:test::localhost', "", "",
    {
     PrintError => 1, # warn( ) on errors
     RaiseError => 0, # don't die on error
     AutoCommit => 1, # commit executes
     # immediately
    }
) or die "Cannot connect to database: $DBI::errstr";
```

Version 5

Use both `connect_on_init( )` from version 4 and `install_driver( )` from version 2.

The `Apache::Registry` test script that we have used is shown in Example 10-10.

Example 10-10. preload_dbi.pl

```
use strict;
use GTop ( );
use DBI ( );

my $dbh = DBI->connect("DBI:mysql:test::localhost", "", "",
    {
     PrintError => 1, # warn( ) on errors
     RaiseError => 0, # don't die on error
     AutoCommit => 1, # commit executes immediately
    }
) or die "Cannot connect to database: $DBI::errstr";

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

my $do_sql = "SHOW TABLES";
my $sth = $dbh->prepare($do_sql);
$sth->execute( );
my @data = ( );
while (my @row = $sth->fetchrow_array) {
    push @data, @row;
}
print "Data: @data\n";
$dbh->disconnect( ); # NOOP under Apache::DBI

my $proc_mem = GTop->new->proc_mem($$);
my $size  = $proc_mem->size;
my $share = $proc_mem->share;
my $diff  = $size - $share;
printf "%8s %8s %8s\n", qw(Size Shared Unshared);
printf "%8d %8d %8d (bytes)\n", $size, $share, $diff;
```

The script opens a connection to the database *test* and issues a query to learn what tables the database has. Ordinarily, when the data is collected and printed the

connection would be closed, but `Apache::DBI` overrides thsi with an empty method. After processing the data, the memory usage is printed. You will already be familiar with that part of the code.

Here are the results of the five tests. The server was restarted before each new test. We have sorted the results by the *Unshared* column.

1. After the first request:

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

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

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

What do we conclude from analyzing this data? First we see that only after a second reload do we get the final memory footprint for the specific request in question (if you pass different arguments, the memory usage will be different).

But both tables show the same pattern of memory usage. We can clearly see that the real winner is version 2, where the MySQL driver was installed. Since we want to have a connection ready for the first request made to the freshly spawned child process, we generally use version 5. This uses somewhat more memory but has almost the same number of shared memory pages. Version 3 preloads only the driver, which results in less shared memory. Having nothing initialized (version 1) and using only the `connect_on_init()` method (version 4) gave the least shared memory. The former is a little bit better than the latter, but both are significantly worse than the first two.

Notice that the smaller the value of the *Unshared* column, the more processes you can have using the same amount of RAM. If we compare versions 2 and 4 of the script, assuming for example that we have 256 MB of memory dedicated to mod_perl processes, we get the following numbers.

Version 2:

$$N = \frac{268435456 - 2609152}{860160} = 309$$

Version 4:

$$N = \frac{268435456 - 2469888}{1011712} = 262$$

As you can see, there are 17% more child processes with version 2.

Initializing CGI.pm

CGI.pm is a big module that by default postpones the compilation of its methods until they are actually needed, thus making it possible to use it under a slow mod_cgi handler without adding a big startup overhead. That's not what we want under mod_perl—if you use CGI.pm, in addition to preloading the module at server startup, you should precompile the methods that you are going to use. To do that, simply call the compile() method:

```
use CGI;
CGI->compile(':all');
```

You should replace the tag group :all with the real tags and group tags that you are going to use if you want to optimize memory usage.

We are going to compare the shared-memory footprint using a script that is backward compatible with mod_cgi. You can improve the performance of this kind of script as well, but if you really want fast code, think about porting it to use Apache::Request* for the CGI interface and some other module for your HTML generation.

The Apache::Registry script that we are going to use to make the comparison is shown in Example 10-11.

Example 10-11. preload_cgi_pm.pl

```
use strict;
use CGI ();
use GTop ();

my $q = new CGI;
print $q->header('text/plain');
print join "\n", map {"$_ => ".$q->param($_) } $q->param;
print "\n";

my $proc_mem = GTop->new->proc_mem($$);
my $size  = $proc_mem->size;
my $share = $proc_mem->share;
my $diff  = $size - $share;
printf "%8s %8s %8s\n", qw(Size Shared Unshared);
printf "%8d %8d %8d (bytes)\n", $size, $share, $diff;
```

The script initializes the CGI object, sends the HTTP header, and then prints any arguments and values that were passed to it. At the end, as usual, we print the memory usage.

* Apache::Request is significantly faster than CGI.pm because its methods for processing a request's arguments are written in C.

Again, we are going to use a single child process. Here is part of our *httpd.conf* file:

```
MinSpareServers 1
MaxSpareServers 1
StartServers 1
MaxClients 1
MaxRequestsPerChild 100
```

We always preload the `Gtop` module:

```
use Gtop ();
```

We are going to run memory benchmarks on three different versions of the *startup.pl* file:

Version 1
: Leave the file unmodified.

Version 2
: Preload `CGI.pm`:

```
use CGI ();
```

Version 3
: Preload `CGI.pm` and precompile the methods that we are going to use in the script:

```
use CGI ();
CGI->compile(qw(header param));
```

Here are the results of the three tests, sorted by the *Unshared* column. The server was restarted before each new test.

1. After the first request:

```
Test type                         Size     Shared   Unshared
-----------------------------------------------------------
(3) preloaded & methods+compiled  3244032  2465792   778240
(2) preloaded                     3321856  2326528   995328
(1) not preloaded                 3321856  2146304  1175552
```

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

```
Test type                         Size     Shared   Unshared
-----------------------------------------------------------
(3) preloaded & methods+compiled  3248128  2445312   802816
(2) preloaded                     3325952  2314240  1011712
(1) not preloaded                 3325952  2134016  1191936
```

Since the memory usage stabilized after the second request, we are going to look at the second table. By comparing the first (not preloaded) and the second (preloaded) versions, we can see that preloading adds about 180 KB (2314240 – 2134016 bytes) of shared memory size, which is the result we expect from most modules. However, by comparing the second (preloaded) and the third (preloaded and precompiled methods) options, we can see that by precompiling methods, we gain 207 KB (1011712 – 802816 bytes) more of shared memory. And we have used only a few methods (the `header` method loads a few more methods transparently for the user).

The gain grows as more of the used methods are precompiled. If you use CGI.pm's functional interface, all of the above applies as well.

Even in our very simple case using the same formula, what do we see? Let's again assume that we have 256 MB dedicated for mod_perl.

Version 1:

$$N = \frac{268435456 - 2134016}{1191936} = 223$$

Version 3:

$$N = \frac{268435456 - 2445312}{802816} = 331$$

If we preload CGI.pm and precompile a few methods that we use in the test script, we can have 50% more child processes than when we don't preload and precompile the methods that we are going to use.

Note that CGI.pm Versions 3.x are supposed to be much less bloated, but make sure to test your code as we just demonstrated.

Memory Preallocation

Perl reuses allocated memory whenever possible. With Devel::Peek we can actually see this happening by peeking at the variable data structure. Consider the simple code in Example 10-12.

Example 10-12. realloc.pl

```
use Devel::Peek;

foo( ) for 1..2;

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

The code starts by loading the Devel::Peek module and calling the function foo( ) twice in the for loop.

The foo( ) function declares a lexically scoped variable, $sv (scalar value). Then it dumps the $sv data structure and prints a separator, assigns a string of 100,000 *x* characters to $sv, assigns it to an empty string, and prints the $sv data structure again. At the end, a separator of two empty lines is printed.

Let's observe the output generated by this code:

```
SV = NULL(0x0) at 0x80787c0
  REFCNT = 1
  FLAGS = (PADBUSY,PADMY)
----
SV = PV(0x804c6c8) at 0x80787c0
  REFCNT = 1
  FLAGS = (PADBUSY,PADMY,POK,pPOK)
  PV = 0x8099d98 ""\0
  CUR = 0
  LEN = 100001


SV = PV(0x804c6c8) at 0x80787c0
  REFCNT = 1
  FLAGS = (PADBUSY,PADMY)
  PV = 0x8099d98 ""\0
  CUR = 0
  LEN = 100001
----
SV = PV(0x804c6c8) at 0x80787c0
  REFCNT = 1
  FLAGS = (PADBUSY,PADMY,POK,pPOK)
  PV = 0x8099d98 ""\0
  CUR = 0
  LEN = 100001
```

In this output, we are interested in the values of `PV`—the memory address of the string value, and `LEN`—the length of the allocated memory.

When `foo( )` is called for the first time and the `$sv` data structure is dumped for the first time, we can see that no data has yet been assigned to it. The second time the `$sv` data structure is dumped, we can see that while `$sv` contains an empty string, its data structure still kept all the memory allocated for the long string.

Notice that `$sv` is declared with `my( )`, so at the end of the function `foo( )` it goes out of scope (i.e., it is destroyed). To our surprise, when we observe the output from the second call to `foo( )`, we discover that when `$sv` is declared at the beginning of `foo( )`, it reuses the data structure from the previously destroyed `$sv` variable—the `PV` field contains the same memory address and the `LEN` field is still 100,101 characters long.

If we had asked for a longer memory chunk during the second invocation, Perl would have called `realloc( )` and a new chunk of memory would have been allocated.

Therefore, if you have some kind of buffering variable that will grow over the processes life, you may want to preallocate the memory for this variable. For example, if you know a variable `$Book::Buffer::buffer` may grow to the size of 100,000 characters, you can preallocate the memory in the following way:

```
package Book::Buffer;

my $buffer;
sub prealloc { $buffer = ' ' x 100_000; $buffer = ""; 0;}
```

```
# ...
1;
```

You should load this module during the `PerlChildInitHandler`. In *startup.pl*, insert:

```
use Book::Buffer;
Apache->push_handlers(PerlChildInitHandler => \&Book::Buffer::prealloc);
```

so each child will allocate its own memory for the variable. When `$Book::Buffer::buffer` starts growing at runtime, no time will be wasted on memory reallocation as long as the preallocated memory is sufficient.

Forking and Executing Subprocesses from mod_perl

When you fork Apache, you are forking the entire Apache server, lock, stock and barrel. Not only are you duplicating your Perl code and the Perl interpreter, but you are also duplicating all the core routines and whatever modules you have used in your server—for example, mod_ssl, mod_rewrite, mod_log, mod_proxy, and mod_speling (no, that's not a typo!). This can be a large overhead on some systems, so wherever possible, it's desirable to avoid forking under mod_perl.

Modern operating systems have a light version of `fork( )`, optimized to do the absolute minimum of memory-page duplication, which adds little overhead when called. This fork relies on the *copy-on-write* technique. The gist of this technique is as follows: the parent process's memory pages aren't all copied immediately to the child's space on `fork( )`ing; this is done later, when the child or the parent modifies the data in the shared memory pages.

If you need to call a Perl program from your mod_perl code, it's better to try to convert the program into a module and call it as a function without spawning a special process to do that. Of course, if you cannot do that or the program is not written in Perl, you have to call the program via `system( )` or an equivalent function, which spawns a new process. If the program is written in C, you can try to write some Perl glue code with help of the Inline, XS, or SWIG architectures. Then the program will be executed as a Perl subroutine and avoid a `fork( )` call.

Also by trying to spawn a subprocess, you might be trying to do the wrong thing. If you just want to do some post-processing after sending a response to the browser, look into the `PerlCleanupHandler` directive. This allows you to do exactly that. If you just need to run some cleanup code, you may want to register this code during the request processing via:

```
my $r = shift;
$r->register_cleanup(\&do_cleanup);
sub do_cleanup{ #some clean-up code here }
```

But when a lengthy job needs to be done, there is not much choice but to use `fork()`. You cannot just run such a job within an Apache process, since firstly it will keep the Apache process busy instead of letting it do the job it was designed for, and secondly, unless it is coded so as to detach from the Apache processes group, if Apache should happen to be stopped the lengthy job might be terminated as well.

In the following sections, we'll discuss how to properly spawn new processes under mod_perl.

Forking a New Process

The typical way to call `fork()` under mod_perl is illustrated in Example 10-13.

Example 10-13. fork1.pl

```
defined (my $kid = fork) or die "Cannot fork: $!\n";
if ($kid) {
    # Parent runs this block
}
else {
    # Child runs this block
    # some code comes here
    CORE::exit(0);
}
# possibly more code here usually run by the parent
```

When using `fork()`, you should check its return value, since a return of `undef` it means that the call was unsuccessful and no process was spawned. This can happen for example, when the system is already running too many processes and cannot spawn new ones.

When the process is successfully forked, the parent receives the PID of the newly spawned child as a returned value of the `fork()` call and the child receives `0`. Now the program splits into two. In the above example, the code inside the first block after `if` will be executed by the parent, and the code inside the first block after `else` will be executed by the child.

It's important not to forget to explicitly call `exit()` at the end of the child code when forking. If you don't and there is some code outside the `if...else` block, the child process will execute it as well. But under mod_perl there is another nuance—you must use `CORE::exit()` and not `exit()`, which would be automatically overriden by `Apache::exit()` if used in conjunction with `Apache::Registry` and similar modules. You want the spawned process to quit when its work is done, or it'll just stay alive, using resources and doing nothing.

The parent process usually completes its execution and returns to the pool of free servers to wait for a new assignment. If the execution is to be aborted earlier for

some reason, you should use Apache::exit( ) or die( ). In the case of Apache::Registry or Apache::PerlRun handlers, a simple exit( ) will do the right thing.

Freeing the Parent Process

In the child code, you must also close all the pipes to the connection socket that were opened by the parent process (i.e., STDIN and STDOUT) and inherited by the child, so the parent will be able to complete the request and free itself for serving other requests. If you need the STDIN and/or STDOUT streams, you should reopen them. You may need to close or reopen the STDERR file handle, too. As inherited from its parent, it's opened to append to the *error_log* file, so the chances are that you will want to leave it untouched.

Under mod_perl, the spawned process also inherits the file descriptor that's tied to the socket through which all the communications between the server and the client pass. Therefore, you need to free this stream in the forked process. If you don't, the server can't be restarted while the spawned process is still running. If you attempt to restart the server, you will get the following error:

```
[Mon May 20 23:04:11 2002] [crit]
(98)Address already in use: make_sock:
  could not bind to address 127.0.0.1 port 8000
```

Apache::SubProcess comes to help, providing a method called cleanup_for_exec( ) that takes care of closing this file descriptor.

The simplest way to free the parent process is to close the STDIN, STDOUT, and STDERR streams (if you don't need them) and untie the Apache socket. If the mounted partition is to be unmounted at a later time, in addition you may want to change the current directory of the forked process to / so that the forked process won't keep the mounted partition busy.

To summarize all these issues, here is an example of a fork that takes care of freeing the parent process (Example 10-14).

Example 10-14. fork2.pl

```
use Apache::SubProcess;
defined (my $kid = fork) or die "Cannot fork: $!\n";
if ($kid) {
    # Parent runs this block
}
else {
    # Child runs this block
    $r->cleanup_for_exec( ); # untie the socket
    chdir '/' or die "Can't chdir to /: $!";
    close STDIN;
    close STDOUT;
    close STDERR;
```

Example 10-14. fork2.pl (continued)

```
    # some code goes here

    CORE::exit(0);
}
# possibly more code here usually run by the parent
```

Of course, the real code should be placed between freeing the parent code and the child process termination.

Detaching the Forked Process

Now what happens if the forked process is running and we decide that we need to restart the web server? This forked process will be aborted, because when the parent process dies during the restart, it will kill its child processes as well. In order to avoid this, we need to detach the process from its parent session by opening a new session with help of a `setsid( )` system call (provided by the `POSIX` module). This is demonstrated in Example 10-15.

Example 10-15. fork3.pl

```
use POSIX 'setsid';

defined (my $kid = fork) or die "Cannot fork: $!\n";
if ($kid) {
    # Parent runs this block
}
else {
    # Child runs this block
    setsid or die "Can't start a new session: $!";
    # ...
}
```

Now the spawned child process has a life of its own, and it doesn't depend on the parent any more.

Avoiding Zombie Processes

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

A zombie is a process that doesn't have a parent. When the child quits, it reports the termination to its parent. If no parent `wait( )`s to collect the exit status of the child, it gets confused and becomes a ghost process that can be seen as a process but not killed. It will be killed only when you stop the parent process that spawned it.

Generally, the *ps(1)* utility displays these processes with the <defunc> tag, and you may see the zombies counter increment when using *top()*. These zombie processes can take up system resources and are generally undesirable.

The proper way to do a fork, to avoid zombie processes, is shown in Example 10-16.

Example 10-16. fork4.pl

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

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

In most cases, the only reason you would want to fork is when you need to spawn a process that will take a long time to complete. So if the Apache process that spawns this new child process has to wait for it to finish, you have gained nothing. You can neither wait for its completion (because you don't have the time to) nor continue, because if you do you will get yet another zombie process. This is called a *blocking call*, since the process is blocked from doing anything else until this call gets completed.

The simplest solution is to ignore your dead children. Just add this line before the `fork()` call:

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

When you set the `CHLD` (`SIGCHLD` in C) signal handler to `'IGNORE'`, all the processes will be collected by the `init` process and therefore will be prevented from becoming zombies. This doesn't work everywhere, but it has been proven to work at least on Linux.

Note that you cannot localize this setting with `local()`. If you try, it won't have the desired effect.

The latest version of the code is shown in Example 10-17.

Example 10-17. fork5.pl

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

$SIG{CHLD} = 'IGNORE';

defined (my $kid = fork) or die "Cannot fork: $!\n";
if ($kid) {
    print "Parent has finished\n";
```

Example 10-17. fork5.pl (continued)

```
}
else {
    # do something time-consuming
    CORE::exit(0);
}
```

Note that the `waitpid( )` call is gone. The `$SIG{CHLD} = 'IGNORE';` statement protects us from zombies, as explained above.

Another solution (more portable, but slightly more expensive) is to use a double fork approach, as shown in Example 10-18.

Example 10-18. fork6.pl

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

defined (my $kid = fork) or die "Cannot fork: $!\n";
if ($kid) {
    waitpid($kid,0);
}
else {
    defined (my $grandkid = fork) or die "Kid cannot fork: $!\n";
    if ($grandkid) {
        CORE::exit(0);
    }
    else {
        # code here
        # do something long lasting
        CORE::exit(0);
    }
}
```

Grandkid becomes a child of `init`—i.e., a child of the process whose PID is 1.

Note that the previous two solutions do allow you to determine the exit status of the process, but in our example, we don't care about it.

Yet another solution is to use a different `SIGCHLD` handler:

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

This is useful when you `fork( )` more than one process. The handler could call `wait( )` as well, but for a variety of reasons involving the handling of stopped processes and the rare event in which two children exit at nearly the same moment, the best technique is to call `waitpid( )` in a tight loop with a first argument of `-1` and a second argument of `WNOHANG`. Together these arguments tell `waitpid( )` to reap the next child that's available and prevent the call from blocking if there happens to be no child ready for reaping. The handler will loop until `waitpid( )` returns a negative number or zero, indicating that no more reapable children remain.

While testing and debugging code that uses one of the above examples, you might want to write debug information to the *error_log* file so that you know what's happening.

Read the *perlipc* manpage for more information about signal handlers.

A Complete Fork Example

Now let's put all the bits of code together and show a well-written example that solves all the problems discussed so far. We will use an `Apache::Registry` script for this purpose. Our script is shown in Example 10-19.

Example 10-19. proper_fork1.pl

```
use strict;
use POSIX 'setsid';
use Apache::SubProcess;

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

$SIG{CHLD} = 'IGNORE';
defined (my $kid = fork) or die "Cannot fork: $!\n";
if ($kid) {
    print "Parent $$ has finished, kid's PID: $kid\n";
}
else {
    $r->cleanup_for_exec( ); # untie the socket
    chdir '/'                 or die "Can't chdir to /: $!";
    open STDIN, '/dev/null'   or die "Can't read /dev/null: $!";
    open STDOUT, '>/dev/null' or die "Can't write to /dev/null: $!";
    open STDERR, '>/tmp/log'  or die "Can't write to /tmp/log: $!";
    setsid                    or die "Can't start a new session: $!";

    my $oldfh = select STDERR;
    local $| = 1;
    select $oldfh;
    warn "started\n";

    # do something time-consuming
    sleep 1, warn "$_\n" for 1..20;
    warn "completed\n";

    CORE::exit(0); # terminate the process
}
```

The script starts with the usual declaration of `strict` mode, then loads the `POSIX` and `Apache::SubProcess` modules and imports the `setsid( )` symbol from the `POSIX` package.

The HTTP header is sent next, with the `Content-Type` of `text/plain`. To avoid zombies, the parent process gets ready to ignore the child, and the fork is called.

The `if` condition evaluates to a true value for the parent process and to a false value for the child process; therefore, the first block is executed by the parent and the second by the child.

The parent process announces its PID and the PID of the spawned process, and finishes its block. If there is any code outside the `if` statement, it will be executed by the parent as well.

The child process starts its code by disconnecting from the socket, changing its current directory to `/`, and opening the `STDIN` and `STDOUT` streams to */dev/null* (this has the effect of closing them both before opening them). In fact, in this example we don't need either of these, so we could just `close( )` both. The child process completes its disengagement from the parent process by opening the `STDERR` stream to */tmp/log*, so it can write to that file, and creates a new session with the help of `setsid( )`. Now the child process has nothing to do with the parent process and can do the actual processing that it has to do. In our example, it outputs a series of warnings, which are logged to */tmp/log*:

```
my $oldfh = select STDERR;
local $| = 1;
select $oldfh;
warn "started\n";
# do something time-consuming
sleep 1, warn "$_\n" for 1..20;
warn "completed\n";
```

We set `$|=1` to unbuffer the `STDERR` stream, so we can immediately see the debug output generated by the program. We use the keyword `local` so that buffering in other processes is not affected. In fact, we don't really need to unbuffer output when it is generated by `warn( )`. You want it if you use `print( )` to debug.

Finally, the child process terminates by calling:

```
CORE::exit(0);
```

which makes sure that it terminates at the end of the block and won't run some code that it's not supposed to run.

This code example will allow you to verify that indeed the spawned child process has its own life, and that its parent is free as well. Simply issue a request that will run this script, see that the process starts writing warnings to the file */tmp/log*, and issue a complete server stop and start. If everything is correct, the server will successfully restart and the long-term process will still be running. You will know that it's still running if the warnings are still being written into */tmp/log*. If Apache takes a long time to stop and restart, you may need to raise the number of warnings to make sure that you don't miss the end of the run.

If there are only five warnings to be printed, you should see the following output in the */tmp/log* file:

```
started
1
2
3
4
5
completed
```

Starting a Long-Running External Program

What happens if we cannot just run Perl code from the spawned process? We may have a compiled utility, such as a program written in C, or a Perl program that cannot easily be converted into a module and thus called as a function. In this case, we have to use `system()`, `exec()`, `qx()` or `` `` (backticks) to start it.

When using any of these methods, and when taint mode is enabled, we must also add the following code to untaint the `PATH` environment variable and delete a few other insecure environment variables. This information can be found in the *perlsec* manpage.

```
$ENV{'PATH'} = '/bin:/usr/bin';
delete @ENV{'IFS', 'CDPATH', 'ENV', 'BASH_ENV'};
```

Now all we have to do is reuse the code from the previous section.

First we move the core program into the *external.pl* file, then we add the shebang line so that the program will be executed by Perl, tell the program to run under taint mode (*-T*), possibly enable *warnings* mode (*-w*), and make it executable. These changes are shown in Example 10-20.

Example 10-20. external.pl

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

open STDIN, '/dev/null'   or die "Can't read /dev/null: $!";
open STDOUT, '>/dev/null' or die "Can't write to /dev/null: $!";
open STDERR, '>/tmp/log'  or die "Can't write to /tmp/log: $!";

my $oldfh = select STDERR;
local $| = 1;
select $oldfh;
warn "started\n";
# do something time-consuming
sleep 1, warn "$_\n" for 1..20;
warn "completed\n";
```

Now we replace the code that we moved into the external program with a call to `exec()` to run it, as shown in Example 10-21.

Example 10-21. proper_fork_exec.pl

```
use strict;
use POSIX 'setsid';
use Apache::SubProcess;

$ENV{'PATH'} = '/bin:/usr/bin';
delete @ENV{'IFS', 'CDPATH', 'ENV', 'BASH_ENV'};

my $r = shift;
$r->send_http_header("text/html");

$SIG{CHLD} = 'IGNORE';

defined (my $kid = fork) or die "Cannot fork: $!\n";
if ($kid) {
    print "Parent has finished, kid's PID: $kid\n";
}
else {
    $r->cleanup_for_exec( ); # untie the socket
    chdir '/'                 or die "Can't chdir to /: $!";
    open STDIN, '/dev/null'   or die "Can't read /dev/null: $!";
    open STDOUT, '>/dev/null' or die "Can't write to /dev/null: $!";
    open STDERR, '>&STDOUT'   or die "Can't dup stdout: $!";
    setsid                    or die "Can't start a new session: $!";

    exec "/home/httpd/perl/external.pl" or die "Cannot execute exec: $!";
}
```

Notice that exec( ) never returns unless it fails to start the process. Therefore you shouldn't put any code after exec( )—it will not be executed in the case of success. Use system( ) or backticks instead if you want to continue doing other things in the process. But then you probably will want to terminate the process after the program has finished, so you will have to write:

```
system "/home/httpd/perl/external.pl"
    or die "Cannot execute system: $!";
CORE::exit(0);
```

Another important nuance is that we have to close all STD streams in the forked process, even if the called program does that.

If the external program is written in Perl, you can pass complicated data stuctures to it using one of the methods to serialize and then restore Perl data. The Storable and FreezeThaw modules come in handy. Let's say that we have a program called *master.pl* (Example 10-22) calling another program called *slave.pl* (Example 10-23).

Example 10-22. master.pl

```
# we are within the mod_perl code
use Storable ( );
my @params = (foo => 1, bar => 2);
```

Example 10-22. master.pl (continued)

```
my $params = Storable::freeze(\@params);
exec "./slave.pl", $params or die "Cannot execute exec: $!";
```

Example 10-23. slave.pl

```
#!/usr/bin/perl -w
use Storable ( );
my @params = @ARGV ? @{ Storable::thaw(shift)||[ ] } : ( );
# do something
```

As you can see, *master.pl* serializes the `@params` data structure with `Storable::freeze` and passes it to *slave.pl* as a single \argument. *slave.pl* recovers it with `Storable::thaw`, by shifting the first value of the `@ARGV` array (if available). The `FreezeThaw` module does a very similar thing.

Starting a Short-Running External Program

Sometimes you need to call an external program and you cannot continue before this program completes its run (e.g., if you need it to return some result). In this case, the fork solution doesn't help. There are a few ways to execute such a program. First, you could use `system( )`:

```
system "perl -e 'print 5+5'"
```

You would never call the Perl interperter for doing a simple calculation like this, but for the sake of a simple example it's good enough.

The problem with this approach is that we cannot get the results printed to `STDOUT`. That's where backticks or `qx( )` can help. If you use either:

```
my $result = `perl -e 'print 5+5'`;
```

or:

```
my $result = qx{perl -e 'print 5+5'};
```

the whole output of the external program will be stored in the `$result` variable.

Of course, you can use other solutions, such as opening a pipe (`|`) to the program if you need to submit many arguments. And there are more evolved solutions provided by other Perl modules, such as `IPC::Open2` and `IPC::Open3`, that allow you to open a process for reading, writing, and error handling.

Executing system( ) or exec( ) in the Right Way

The Perl `exec( )` and `system( )` functions behave identically in the way they spawn a program. Let's use `system( )` as an example. Consider the following code:

```
system("echo", "Hi");
```

Perl will use the first argument as a program to execute, find the `echo` executable along the search path, invoke it directly, and pass the string "Hi" as an argument.

Note that Perl's `system()` is not the same as the standard *libc* `system(3)` call.

If there is more than one argument to `system()` or `exec()`, or the argument is an array with more than one element in it, the arguments are passed directly to the C-level functions. When the argument is a single scalar or an array with only a single scalar in it, it will first be checked to see if it contains any shell metacharacters (e.g., `*`, `?`). If there are any, the Perl interpreter invokes a real shell program (*/bin/sh -c* on Unix platforms). If there are no shell metacharacters in the argument, it is split into words and passed directly to the C level, which is more efficient.

In other words, only if you do:

```
system "echo *"
```

will Perl actually `exec()` a copy of */bin/sh* to parse your command, which may incur a slight overhead on certain OSes.

It's especially important to remember to run your code with taint mode enabled when `system()` or `exec()` is called using a single argument. There can be bad consequences if user input gets to the shell without proper laundering first. Taint mode will alert you when such a condition happens.

Perl will try to do the most efficient thing no matter how the arguments are passed, and the additional overhead may be incurred only if you need the shell to expand some metacharacters before doing the actual call.

References

- *Mastering Regular Expressions*, by Jeffrey E. F. Friedl (O'Reilly).
- Chapters 2 and 4 in *Operating Systems: Design And Implementation*, by Andrew S. Tanenbaum and Albert S. Woodhull (Prentice Hall).
- Chapter 4 in *Modern Operating Systems*, by Andrew S. Tanenbaum (Prentice Hall).
- Chapters 7 and 9 in *Design of the UNIX Operating System*, by Maurice J. Bach (Prentice Hall).
- Chapter 9 ("Tuning Apache and mod_perl") in *mod_perl Developer's Cookbook*, by Geoffrey Young, Paul Lindner, and Randy Kobes (Sams Publishing).
- The Solaris memory system, sizing, tools, and architecture: *http://www.sun.com/sun-on-net/performance/vmsizing.pdf*.
- Refer to the *Unix Programming Frequently Asked Questions* to learn more about `fork()` and related system calls: *http://www.erlenstar.demon.co.uk/unix/faq_toc.html*.