CHAPTER 14

Defensive Measures for Performance Enhancement

If you have already worked with mod_perl, you have probably noticed that it can be difficult to keep your mod_perl processes from using a lot of memory. The less memory you have, the fewer processes you can run and the worse your server will perform, especially under a heavy load. This chapter presents several common situations that can lead to unnecessary consumption of RAM, together with preventive measures.

Controlling Your Memory Usage

When you need to control the size of your *httpd* processes, use one of the two modules, `Apache::GTopLimit` and `Apache::SizeLimit`, which kill Apache *httpd* processes when those processes grow too large or lose a big chunk of their shared memory. The two modules differ in their methods for finding out the memory usage. `Apache::GTopLimit` relies on the `libgtop` library to perform this task, so if this library can be built on your platform you can use this module. `Apache::SizeLimit` includes different methods for different platforms—you will have to check the module's manpage to figure out which platforms are supported.

Defining the Minimum Shared Memory Size Threshold

As we have already discussed, when it is first created, an Apache child process usually has a large fraction of its memory shared with its parent. During the child process's life some of its data structures are modified and a part of its memory becomes unshared (pages become "dirty"), leading to an increase in memory consumption. You will remember that the `MaxRequestsPerChild` directive allows you to specify the number of requests a child process should serve before it is killed. One way to limit the memory consumption of a process is to kill it and let Apache replace it with a newly started process, which again will have most of its memory shared with the Apache parent. The new child process will then serve requests, and eventually the cycle will be repeated.

This is a fairly crude means of limiting unshared memory, and you will probably need to tune `MaxRequestsPerChild`, eventually finding an optimum value. If, as is likely, your service is undergoing constant changes, this is an inconvenient solution. You'll have to retune this number again and again to adapt to the ever-changing code base.

You really want to set some guardian to watch the shared size and kill the process if it goes below some limit. This way, processes will not be killed unnecessarily.

To set a shared memory lower limit of 4 MB using `Apache::GTopLimit`, add the following code into the *startup.pl* file:

```
use Apache::GTopLimit;
$Apache::GTopLimit::MIN_PROCESS_SHARED_SIZE = 4096;
```

and add this line to *httpd.conf*:

```
PerlFixupHandler Apache::GTopLimit
```

Don't forget to restart the server for the changes to take effect.

Adding these lines has the effect that as soon as a child process shares less than 4 MB of memory (the corollary being that it must therefore be occupying a lot of memory with its unique pages), it will be killed after completing its current request, and, as a consequence, a new child will take its place.

If you use `Apache::SizeLimit` you can accomplish the same by adding this to *startup.pl*:

```
use Apache::SizeLimit;
$Apache::SizeLimit::MIN_SHARE_SIZE = 4096;
```

and this to *httpd.conf*:

```
PerlFixupHandler Apache::SizeLimit
```

If you want to set this limit for only some requests (presumably the ones you think are likely to cause memory to become unshared), you can register a post-processing check using the `set_min_shared_size( )` function. For example:

```
use Apache::GTopLimit;
if ($need_to_limit) {
    # make sure that at least 4MB are shared
    Apache::GTopLimit->set_min_shared_size(4096);
}
```

or for `Apache::SizeLimit`:

```
use Apache::SizeLimit;
if ($need_to_limit) {
    # make sure that at least 4MB are shared
    Apache::SizeLimit->setmin(4096);
}
```

Since accessing the process information adds a little overhead, you may want to check the process size only every *N* times. In this case, set the `$Apache::GTopLimit::`

CHECK_EVERY_N_REQUESTS variable. For example, to test the size every other time, put the following in your *startup.pl* file:

```
$Apache::GTopLimit::CHECK_EVERY_N_REQUESTS = 2;
```

or, for Apache::SizeLimit:

```
$Apache::SizeLimit::CHECK_EVERY_N_REQUESTS = 2;
```

You can run the Apache::GTopLimit module in debug mode by setting:

```
PerlSetVar Apache::GTopLimit::DEBUG 1
```

in *httpd.conf*. It's important that this setting appears before the Apache::GTopLimit module is loaded.

When debug mode is turned on, the module reports in the *error_log* file the memory usage of the current process and also when it detects that at least one of the thresholds was crossed and the process is going to be killed.

Apache::SizeLimit controls the debug level via the $Apache::SizeLimit::DEBUG variable:

```
$Apache::SizeLimit::DEBUG = 1;
```

which can be modified any time, even after the module has been loaded.

Potential drawbacks of memory-sharing restrictions

In Chapter 11 we devised a formula to calculate the optimum value for the MaxClients directive when sharing is taking place. In the same section, we warned that it's very important that the system not be heavily engaged in swapping. Some systems do swap in and out every so often even if they have plenty of real memory available, and that's OK. The following discussion applies to conditions when there is hardly any free memory available.

If the system uses almost all of its real memory (including the cache), there is a danger of the parent process's memory pages being swapped out (i.e., written to a swap device). If this happens, the memory-usage reporting tools will report all those swapped out pages as nonshared, even though in reality these pages are still shared on most OSs. When these pages are getting swapped in, the sharing will be reported back to normal after a certain amount of time. If a big chunk of the memory shared with child processes is swapped out, it's most likely that Apache::SizeLimit or Apache::GTopLimit will notice that the shared memory threshold was crossed and as a result kill those processes. If many of the parent process's pages are swapped out, and the newly created child process is already starting with shared memory below the limit, it'll be killed immediately after serving a single request (assuming that the $CHECK_EVERY_N_REQUESTS variable is set to 1). This is a very bad situation that will eventually lead to a state where the system won't respond at all, as it'll be heavily engaged in the swapping process.

This effect may be less or more severe depending on the memory manager's implementation, and it certainly varies from OS to OS and between kernel versions. Therefore, you should be aware of this potential problem and simply try to avoid situations where the system needs to swap at all, by adding more memory, reducing the number of child servers, or spreading the load across more machines (if reducing the number of child servers is not an option because of the request-rate demands).

Defining the Maximum Memory Size Threshold

No less important than maximizing shared memory is restricting the absolute size of the processes. If the processes grow after each request, and if nothing restricts them from growing, you can easily run out of memory.

Again you can set the `MaxRequestsPerChild` directive to kill the processes after a few requests have been served. But as we explained in the previous section, this solution is not as good as one that monitors the process size and kills it only when some limit is reached.

If you have `Apache::GTopLimit` (described in the previous section), you can limit a process's memory usage by setting the `$Apache::GTopLimit::MAX_PROCESS_SIZE` directive. For example, if you want processes to be killed when they reach 10 MB, you should put the following in your *startup.pl* file:

```
$Apache::GTopLimit::MAX_PROCESS_SIZE = 10240;
```

Just as when limiting shared memory, you can set a limit for the current process using the `set_max_size( )` method in your code:

```
use Apache::GTopLimit;
Apache::GTopLimit->set_max_size(10000);
```

For `Apache::SizeLimit`, the equivalents are:

```
use Apache::SizeLimit;
$Apache::SizeLimit::MAX_PROCESS_SIZE = 10240;
```

and:

```
use Apache::SizeLimit;
Apache::SizeLimit->setmax(10240);
```

Defining the Maximum Unshared Memory Size Threshold

Instead of setting the shared and total memory usage thresholds, you can set a single threshold that measures the amount of unshared memory by subtracting the shared memory size from the total memory size.

Both modules allow you to set the thresholds in similar ways. With `Apache::GTopLimit`, you can set the unshared memory threshold server-wide with:

```
$Apache::GTopLimit::MAX_PROCESS_UNSHARED_SIZE = 6144;
```

and locally for a handler with:

```
Apache::GTopLimit->set_max_unshared_size(6144);
```

If you are using `Apache::SizeLimit`, the corresponding settings would be:

```
$Apache::SizeLimit::MAX_UNSHARED_SIZE = 6144;
```

and:

```
Apache::SizeLimit->setmax_unshared(6144);
```

Coding for a Smaller Memory Footprint

The following sections present proactive techniques that prevent processes from growing large in the first place.

Memory Reuse

Consider the code in Example 14-1.

Example 14-1. memory_hog.pl

```
use GTop ();
my $gtop = GTop->new;
my $proc = $gtop->proc_mem($$);
print "size before:  ", $gtop->proc_mem($$)->size(), " B\n";
{
    my $x = 'a' x 10**7;
    print "size inside: ", $gtop->proc_mem($$)->size(), " B\n";
}
print "size  after: ", $gtop->proc_mem($$)->size(), " B\n";
```

When executed, it prints:

```
size before:  1830912 B
size inside: 21852160 B
size  after: 21852160 B
```

This script starts by printing the size of the memory it occupied when it was first loaded. The opening curly brace starts a new block, in which a lexical variable `$x` is populated with a string 10,000,000 bytes in length. The script then prints the new size of the process and exits from the block. Finally, the script again prints the size of the process.

Since the variable `$x` is lexical, it is destroyed at the end of the block, before the final print statement, thus releasing all the memory that it was occupying. But from the output we can clearly see that a huge chunk of memory wasn't released to the OS—the process's memory usage didn't change. Perl reuses this released memory internally. For example, let's modify the script as shown in Example 14-2.

Example 14-2. memory_hog2.pl

```
use GTop ();
my $gtop = GTop->new;
my $proc = $gtop->proc_mem($$);
print "size before :  ", $gtop->proc_mem($$)->size(), " B\n";
{
    my $x = 'a' x 10**7;
    print "size inside : ", $gtop->proc_mem($$)->size(), " B\n";
}
print "size  after : ", $gtop->proc_mem($$)->size(), " B\n";
{
    my $x = 'a' x 10;
    print "size inside2: ", $gtop->proc_mem($$)->size(), " B\n";
}
print "size  after2: ", $gtop->proc_mem($$)->size(), " B\n";
```

When we execute this script, we will see the following output:

```
size before :  1835008 B
size inside : 21852160 B
size  after : 21852160 B
size inside2: 21852160 B
size  after2: 21852160 B
```

As you can see, the memory usage of this script was no more than that of the previous one.

So we have just learned that Perl programs don't return memory to the OS until they quit. If variables go out of scope, the memory they occupied is reused by Perl for newly created or growing variables.

Suppose your code does memory-intensive operations and the processes grow fast at first, but after a few requests the sizes of the processes stabilize as Perl starts to reuse the acquired memory. In this case, the wisest approach is to find this limiting size and set the upper memory limit to a slightly higher value. If you set the limit lower, processes will be killed unnecessarily and lots of redundant operations will be performed by the OS.

Big Input, Big Damage

This section demonstrates how a malicious user can bring the service down or cause problems by submitting unexpectedly big data.

Imagine that you have a guestbook script/handler, which works fine. But you've forgotten about a small nuance: you don't check the size of the submitted message. A 10 MB core file copied and pasted into the HTML `textarea` entry box intended for a guest's message and submitted to the server will make the server grow by at least 10 MB. (Not to mention the horrible experience users will go through when trying to view the guest book, since the contents of the binary core file will be displayed.) If your

server is short of memory, after a few more submissions like this one it will start swapping, and it may be on its way to crashing once all the swap memory is exhausted.

To prevent such a thing from happening, you could check the size of the submitted argument, like this:

```
my $r = shift;
my %args = $r->args;
my $message = exists $args{message} ? $args{message} : '';
die "the message is too big"
    unless length $message > 8192; # 8KB
```

While this prevents your program from adding huge inputs into the guest book, the size of the process will grow anyway, since you have allowed the code to process the submitted form's data. The only way to really protect your server from accepting huge inputs is not to read data above some preset limit. However, you cannot safely rely on the `Content-Length` header, since that can easily be spoofed.

You don't have to worry about `GET` requests, since their data is submitted via the query string of the URI, which has a hard limit of about 8 KB.

Think about disabling file uploads if you don't use them. Remember that a user can always write an HTML form from scratch and submit it to your program for processing, which makes it easy to submit huge files. If you don't limit the size of the form input, even if your program rejects the faulty input, the data will be read in by the server and the process will grow as a result. Here is a simple example that will readily accept anything submitted by the form, including fields that you didn't create, which a malicious user may have added by mangling the original form:

```
use CGI;
my $q = CGI->new;
my %args = map {$_ => $q->param($_)} $q->params;
```

If you are using `CGI.pm`, you can set the maximum allowed `POST` size and disable file uploads using the following setting:

```
use CGI;
$CGI::POST_MAX = 1048576;  # max 1MB allowed
$CGI::DISABLE_UPLOADS = 1; # disable file uploads
```

The above setting will reject all submitted forms whose total size exceeds 1 MB. Only non–file upload inputs will be processed.

If you are using the `Apache::Request` module, you can disable file uploads and limit the maximum `POST` size by passing the appropriate arguments to the `new( )` function. The following example has the same effect as the `CGI.pm` example shown above:

```
my $apr = Apache::Request->new($r,
                               POST_MAX        => 1048576,
                               DISABLE_UPLOADS => 1
                              );
```

Another alternative is to use the `LimitRequestBody` directive in *httpd.conf* to limit the size of the request body. This directive can be set per-server, per-directory, per-file, or per-location. The default value is 0, which means unlimited. As an example, to limit the size of the request body to 2 MB, you should add:

```
LimitRequestBody 2097152
```

The value is set in bytes (2097152 bytes = = 2 MB).

In this section, we have presented only a single example among many that can cause your server to use more memory than planned. It helps to keep an open mind and to explore what other things a creative user might try to do with your service. Don't assume users will only click where you intend them to.

Small Input, Big Damage

This section demonstrates how a small input submitted by a malicious user may hog the whole server.

Imagine an online service that allows users to create a canvas on the server side and do some fancy image processing. Among the inputs that are to be submitted by the user are the width and the height of the canvas. If the program doesn't restrict the maximum values for them, some smart user may ask your program to create a canvas of 1,000,000×1,000,000 pixels. In addition to working the CPU rather heavily, the processes that serve this request will probably eat all the available memory (including the swap space) and kill the server.

How can the user do this, if you have prepared a form with a pull-down list of possible choices? Simply by saving the form and later editing it, or by using a `GET` request. Don't forget that what you receive is merely an input from a user agent, and it can very easily be spoofed by anyone knowing how to use `LWP::UserAgent` or something equivalent. There are various techniques to prevent users from fiddling with forms, but it's much simpler to make your code check that the submitted values are acceptable and then move on.

If you do some relational database processing, you will often encounter the need to read lots of records from the database and then print them to the browser after they are formatted. Let's look at an example.

We will use `DBI` and `CGI.pm` for this example. Assume that we are already connected to the database server (refer to the `DBI` manpage for a complete reference to the `DBI` module):

```
my $q = new CGI;
my $default_hits = 10;
my $hits = int $q->param("hints") || $default_hits;

my $do_sql = "SELECT from foo LIMIT 0,$hits";
my $sth = $dbh->prepare($do_sql);
$sth->execute;
```

```
while (@row_ary = $sth->fetchrow_array) {
    # do DB accumulation into some variable
}
# print the data
...
```

In this example, the records are accumulated in the program data before they are printed. The variables that are used to store the records that matched the query will grow by the size of the data, in turn causing the *httpd* process to grow by the same amount.

Imagine a search engine interface that allows a user to choose to display 10, 50, or 100 results. What happens if the user modifies the form to ask for 1,000,000 hits? If you have a big enough database, and if you rely on the fact that the only valid choices would be 10, 50, or 100 without actually checking, your database engine may unexpectedly return a million records. Your process will grow by many megabytes, possibly eating all the available memory and swap space.

The obvious solution is to disallow arbitrary inputs for critical variables like this one. Another improvement is to avoid the accumulation of matched records in the program data. Instead, you could use `DBI::bind_columns()` or a similar function to print each record as it is fetched from the database. In Chapter 20 we will talk about this technique in depth.

Think Production, Not Development

Developers often use sample inputs for testing their new code. But sometimes they forget that the real inputs can be much bigger than those they used in development.

Consider code like this, which is common enough in Perl scripts:

```
{
    open IN, $file or die $!;
    local $/;
    $content = <IN>; # slurp the whole file in
    close IN;
}
```

If you know for sure that the input will always be small, the code we have presented here might be fine. But if the file is 5 MB, the child process that executes this script when serving the request will grow by that amount. Now if you have 20 children, and each one executes this code, together they will consume 20×5 MB = 100 MB of RAM! If, when the code was developed and tested, the input file was very small, this potential excessive memory usage probably went unnoticed.

Try to think about the many situations in which your code might be used. For example, it's possible that the input will originate from a source you did not envisage. Your code might behave badly as a result. To protect against this possibility, you might want to try to use other approaches to processing the file. If it has lines, perhaps you can process one line at a time instead of reading them all into a variable at

once. If you need to modify the file, use a temporary file. When the processing is finished, you can overwrite the source file. Make sure that you lock the files when you modify them.

Often you just don't expect the input to grow. For example, you may want to write a birthday reminder process intended for your own personal use. If you have 100 friends and relatives about whom you want to be reminded, slurping the whole file in before processing it might be a perfectly reasonable way to approach the task.

But what happens if your friends (who know you as one who usually forgets their birthdays) are so surprised by your timely birthday greetings that they ask you to allow them to use your cool invention as well? If all 100 friends have yet another 100 friends, you could end up with 10,000 records in your database. The code may not work well with input of this size. Certainly, the answer is to rewrite the code to use a DBM file or a relational database. If you continue to store the records in a flat file and read the whole database into memory, your code will use a lot of memory and be very slow.

Passing Variables

Let's talk about passing variables to a subroutine. There are two ways to do this: you can pass a *copy* of the variable to the subroutine (this is called *passing by value*) or you can instead pass a *reference* to it (a reference is just a pointer, so the variable itself is not copied). Other things being equal, if the copy of the variable is larger than a pointer to it, it will be more efficient to pass a reference.

Let's use the example from the previous section, assuming we have no choice but to read the whole file before any data processing takes place and its size is 5 MB. Suppose you have some subroutine called `process()` that processes the data and returns it. Now say you pass `$content` by value and `process()` makes a copy of it in the familiar way:

```
my $content = qq{foobarfoobar};
$content = process($content);
sub process {
    my $content = shift;
    $content =~ s/foo/bar/gs;
    return $content;
}
```

You have just copied another 5 MB, and the child has grown in size by another 5 MB. Assuming 20 Apache children, you can multiply this growth again by factor of 20—now you have 200 MB of wasted RAM! This will eventually be reused, but it's still a waste. Whenever you think the variable may grow bigger than a few kilobytes, definitely pass it by reference.

There are several forms of syntax you can use to pass and use variables passed by reference. For example:

```
my $content = qq{foobarfoobar};
process(\$content);
sub process {
    my $r_content = shift;
    $$r_content =~ s/foo/bar/gs;
}
```

Here `$content` is populated with some data and then passed by reference to the subroutine `process( )`, which replaces all occurrences of the string *foo* with the string *bar*. `process( )` doesn't have to return anything—the variable `$content` was modified directly, since `process( )` took a reference to it.

If the hashes or arrays are passed by reference, their individual elements are still accessible. You don't need to dereference them:

```
$var_lr->[$index]  get $index'th element of an array via a ref
$var_hr->{$key}    get $key'th element of a hash via a ref
```

Note that if you pass the variable by reference but then dereference it to copy it to a new string, you don't gain anything, since a new chunk of memory will be acquired to make a *copy* of the original variable. The *perlref* manpage provides extensive information about working with references.

Another approach is to use the `@_` array directly. Internally, Perl always passes these variables by reference and dereferences them when they are copied from the `@_` array. This is an efficiency mechanism to allow you to write subroutines that take a variable passed as a value, without copying it.

```
process($content);
sub process {
  $_[0] =~ s/foo/bar/gs;
}
```

From *perldoc perlsub*:

```
The array @_ is a local array, but its elements are aliases for the actual scalar
parameters.  In particular, if an element $_[0] is updated, the corresponding
argument is updated (or an error occurs if it is not possible to update)...
```

Be careful when you write this kind of subroutine for use by someone else; it can be confusing. It's not obvious that a call like `process($content);` modifies the passed variable. Programmers (the users of your library, in this case) are used to subroutines that either modify variables passed by reference or expressly return a result, like this:

```
$content = process($content);
```

You should also be aware that if the user tries to submit a read-only value, this code won't work and you will get a runtime error. Perl will refuse to modify a read-only value:

```
$content = process("string foo");
```

Memory Leakage

It's normal for a process to grow when it processes its first few requests. They may be different requests, or the same requests processing different data. You may try to reload the same request a few times, and in many cases the process will stop growing after only the second reload. In any case, once a representative selection of requests and inputs has been executed by a process, it won't usually grow any more unless the code leaks memory. If it grows after each reload of an identical request, there is probably a memory leak.

The experience might be different if the code works with some external resource that can change between requests. For example, if the code retrieves database records matching some query, it's possible that from time to time the database will be updated and that a different number of records will match the same query the next time it is issued. Depending on the techniques you use to retrieve the data, format it, and send it to the user, the process may increase or decrease in size, reflecting the changes in the data.

The easiest way to see whether the code is leaking is to run the server in single-process mode (*httpd -X*), issuing the same request a few times to see whether the process grows after each request. If it does, you probably have a memory leak. If the code leaks 5 KB per request, then after 1,000 requests to run the leaking code, 5 MB of memory will have leaked. If in production you have 20 processes, this could possibly lead to 100 MB of leakage after a few tens of thousands of requests.

This technique to detect leakage can be misleading if you are not careful. Suppose your process first runs some clean (non-leaking) code that acquires 100 KB of memory. In an attempt to make itself more efficient, Perl doesn't give the 100 KB of memory back to the operating system. The next time the process runs *any* script, some of the 100 KB will be reused. But if this time the process runs a script that needs to acquire only 5 KB, you won't see the process grow even if the code has actually leaked these 5 KB. Now it might take 20 or more requests for the leaking script *served by the same process* before you would see that process start growing again.

A process may leak memory for several reasons: badly written system C/C++ libraries used in the *httpd* binary and badly written Perl code are the most common. Perl modules may also use C libraries, and these might leak memory as well. Also, some operating systems have been known to have problems with their memory-management functions.

If you know that you have no leaks in your code, then for detecting leaks in C/C++ libraries you should either use the technique of sampling the memory usage described above, or use C/C++ developer tools designed for this purpose. This topic is beyond the scope of this book.

The `Apache::Leak` module (derived from `Devel::Leak`) might help you to detect leaks in your code. Consider the script in Example 14-3.

Example 14-3. leaktest.pl

```
use Apache::Leak;

my $global = "FooA";

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

You do not need to be inside mod_perl to use this script. The argument to `leak_test( )` is an anonymous sub or a block, so you can just throw in any code you suspect might be leaking. The script will run the code twice. The first time, new scalar values (SVs) are created, but this does not mean the code is leaking. The second pass will give better evidence.

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 !!!
```

This module uses the simple approach of walking the Perl internal table of allocated SVs. It records them before entering the scope of the code under test and after leaving the scope. At the end, a comparison of the two sets is performed, `sv_dump( )` is called for anything that did not exist in the first set, and the difference in counts is reported. Note that you will see the dumps of SVs only if Perl was built with the *-DDEBUGGING* option. In our example the script will dump two SVs twice, since the same code is run twice. The volume of output is too great to be presented here.

Our example leaks because `$$global = 1;` creates a new global variable, `FooA` (with the value of `1`), which will not be destroyed until this module is destroyed. Under mod_perl the module doesn't get destroyed until the process quits. When the code is run the second time, `$global` will contain `FooB` because of the increment operation at the end of the first run. Consider:

```
$foo = "AAA";
print "$foo\n";
$foo++;
print "$foo\n";
```

which prints:

```
AAA
AAB
```

So every time the code is executed, a new variable (FooC, FooD, etc.) will spring into existence.

Apache::Leak is not very user-friendly. You may want to take 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. Consider this code, for example:

```
sub test { my ($string) = @_;}
test("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 that the memory will be needed for another string, the next time the subroutine is entered. You'll see similar behavior for @array lengths, %hash keys, and scratch areas of the padlist for operations such as join( ), ., etc.

Let's look at how B::LexInfo works. The code in Example 14-4 creates a new B::LexInfo object, then runs cvrundiff( ), which creates two snapshots of the lexical variables' padlists—one before the call to LeakTest1::test( ) and the other, in this case, after it has been called with the argument "a string". Then it calls *diff -u* to generate the difference between the snapshots.

Example 14-4. leaktest1.pl

```
package LeakTest1;
use B::LexInfo ();

sub test { my ($string) = @_;}

my $lexi = B::LexInfo->new;
my $diff = $lexi->cvrundiff('LeakTest1::test', "a string");
print $$diff;
```

In case you aren't familiar with how diff works, - at the beginning of the line means that that line was removed, + means that a line was added, and other lines are there to show the context in which the difference was found. Here is the output:

```
--- /tmp/B_LexInfo_3099.before        Tue Feb 13 20:09:52 2001
+++ /tmp/B_LexInfo_3099.after        Tue Feb 13 20:09:52 2001
@@ -2,9 +2,11 @@
   {
     'LeakTest1::test' => {
       '$string' => {
-        'TYPE' => 'NULL',
+        'TYPE' => 'PV',
+        'LEN' => 9,
         'ADDRESS' => '0x8146d80',
-        'NULL' => '0x8146d80'
+        'PV' => 'a string',
+        'CUR' => 8
       },
       '__SPECIAL__1' => {
         'TYPE' => 'NULL',
```

Perl tries to optimize the speed by keeping the memory allocated for `$string`, even after the variable is destroyed.

Let's run the script from Example 14-3 with `B::LexInfo` (see Example 14-5).

Example 14-5. leaktest2.pl

```
package LeakTest2;
use B::LexInfo ();

my $global = "FooA";

sub test {
    $$global = 1;
    ++$global;
}

my $lexi = B::LexInfo->new;
my $diff = $lexi->cvrundiff('LeakTest2::test');
print $$diff;
```

Here's the result:

```
    --- /tmp/B_LexInfo_3103.before Tue Feb 13 20:12:04 2001
    +++ /tmp/B_LexInfo_3103.after         Tue Feb 13 20:12:04 2001
    @@ -5,7 +5,7 @@
              'TYPE' => 'PV',
              'LEN' => 5,
              'ADDRESS' => '0x80572ec',
    -         'PV' => 'FooA',
    +         'PV' => 'FooB',
              'CUR' => 4
            }
          }
```

We can clearly see the leakage, since the value of the `PV` entry has changed from one string to a different one. Compare this with the previous example, where a variable didn't exist and sprang into existence for optimization reasons. If you find this confusing, probably the best approach is to run *diff* twice when you test your code.

Now let's run the `cvrundiff( )` function on this example, as shown in Example 14-6.

Example 14-6. leaktest3.pl

```
package LeakTest2;
use B::LexInfo ();

my $global = "FooA";

sub test {
    $$global = 1;
    ++$global;
}
```

Example 14-6. leaktest3.pl (continued)

```
my $lexi = B::LexInfo->new;
my $diff = $lexi->cvrundiff('LeakTest2::test');
$diff    = $lexi->cvrundiff('LeakTest2::test');
print $$diff;
```

Here's the output:

```
--- /tmp/B_LexInfo_3103.before Tue Feb 13 20:12:04 2001
+++ /tmp/B_LexInfo_3103.after         Tue Feb 13 20:12:04 2001
@@ -5,7 +5,7 @@
          'TYPE' => 'PV',
          'LEN' => 5,
          'ADDRESS' => '0x80572ec',
-         'PV' => 'FooB',
+         'PV' => 'FooC',
          'CUR' => 4
        }
     }
```

We can see the leak again, since the value of `PV` has changed again, from `FooB` to `FooC`. Now let's run `cvrundiff()` on the second example script, as shown in Example 14-7.

Example 14-7. leaktest4.pl

```
package LeakTest1;
use B::LexInfo ();

sub test { my ($string) = @_;}

my $lexi = B::LexInfo->new;
my $diff = $lexi->cvrundiff('LeakTest1::test', "a string");
   $diff = $lexi->cvrundiff('LeakTest1::test', "a string");
print $$diff;
```

No output is produced, since there is no difference between the second and third runs. All the data structures are allocated during the first execution, so we are sure that no memory is leaking here.

`Apache::Status` includes a `StatusLexInfo` option that can show you the internals of your code via `B::LexInfo`. See Chapter 21 for more information.

Conclusion

The impacts of coding style, efficiency, differences in data, potential abuse by users, and a host of other factors combine to make each web service unique. You will therefore need to consider these things carefully in the light of your unique knowledge of your system and the pointers and guidelines suggested here. In this chapter we have tried to show how a defensive and efficient coding style will make sure that your processes are reasonably small and also unlikely to grow excessively. Knowing that your

processes are well behaved will give you the confidence to make the best use of the available RAM, so that you can run the maximum number of processes while ensuring that the server will be unlikely to swap.

References

- The `mod_limitipconn.c` and `Apache::LimitIPConn` Apache modules: *http://dominia.org/djao/limitipconn.html*

  These modules allow web server administrators to limit the number of simultaneous downloads permitted from a single IP address.
- Chapter 9 ("Tuning Apache and mod_perl") in *mod_perl Developer's Cookbook*, by Geoffrey Young, Paul Lindner, and Randy Kobes (Sams Publishing)
- *Advanced Perl Programming*, by Sriram Srinivasan (O'Reilly)