3

Memory Management

One of the most jarring differences between a managed language like PHP, and an unmanaged language like C is control over memory pointers.

Memory

In PHP, populating a string variable is as simple as `<?php $str = 'hello world'; ?>` and the string can be freely modified, copied, and moved around. In C, on the other hand, although you could start with a simple static string such as `char *str = "hello world";`, that string cannot be modified because it lives in program space. To create a manipulable string, you'd have to allocate a block of memory and copy the contents in using a function such as `strdup()`.

```
{
    char *str;

    str = strdup("hello world");
    if (!str) {
        fprintf(stderr, "Unable to allocate memory!");
    }
}
```

For reasons you'll explore through the course of this chapter, the traditional memory management functions (`malloc()`, `free()`, `strdup()`, `realloc()`, `calloc()`, and so on) are almost never used directly by the PHP source code.

Free the Mallocs

Memory management on nearly all platforms is handled in a request and release fashion. An application says to the layer above it (usually the operating system) "I want some number of bytes of memory to use as I please." If there is space available, the operating system offers it to the program and makes a note not to give that chunk of memory out to anyone else.

When the application is done using the memory, it's expected to give it back to the OS so that it can be allocated elsewhere. If the program doesn't give the memory back, the OS has no way of knowing that it's no longer being used and can be allocated again by another process. If a block of memory is not freed, and the owning application has lost track of it, then it's said to have "leaked" because it's simply no longer available to anyone.

In a typical client application, small infrequent leaks are sometimes tolerated with the knowledge that the process will end after a short period of time and the leaked memory will be implicitly returned to the OS. This is no great feat as the OS knows which program it gave that memory to, and it can be certain that the memory is no longer needed when the program terminates.

With long running server daemons, including web servers like Apache and by extension `mod_php`, the process is designed to run for much longer periods, often indefinitely. Because the OS can't clean up memory usage, any degree of leakage—no matter how small—will tend to build up over time and eventually exhaust all system resources.

Consider the userspace `stristr()` function; in order to find a string using a case-insensitive search, it actually creates a lowercase copy of both the haystack and the needle, and then performs a more traditional case-sensitive search to find the relative offset. After the offset of the string has been located, however, it no longer has use for the lowercase versions of the haystack and needle strings. If it didn't free these copies, then every script that used `stristr()` would leak some memory every time it was called. Eventually the web server process would own all the system memory, but not be able to use it.

The ideal solution, I can hear you shouting, is to write good, clean, consistent code, and that's absolutely true. In an environment like the PHP interpreter, however, that's only half the solution.

Error Handling

In order to provide the ability to bail out of an active request to userspace scripts and the extension functions they rely on, a means needs to exist to jump out of an active request entirely. The way this is handled within the Zend Engine is to set a bailout address at the beginning of a request, and then on any `die()` or `exit()` call, or on encountering any critical error (`E_ERROR`) perform a `longjmp()` to that bailout address.

Although this bailout process simplifies program flow, it almost invariably means that resource cleanup code (such as `free()` calls) will be skipped and memory could get leaked. Consider this simplified version of the engine code that handles function calls:

```
void call_function(const char *fname, int fname_len TSRMLS_DC)
{
    zend_function *fe;
    char *lcase_fname;
```

```
    /* PHP function names are case-insensitive
     * to simplify locating them in the function tables
     * all function names are implicitly
     * translated to lowercase
     */
    lcase_fname = estrndup(fname, fname_len);
    zend_str_tolower(lcase_fname, fname_len);

    if (zend_hash_find(EG(function_table),
            lcase_fname, fname_len + 1, (void **)&fe) == FAILURE) {
        zend_execute(fe->op_array TSRMLS_CC);
    } else {
        php_error_docref(NULL TSRMLS_CC, E_ERROR,
                         "Call to undefined function: %s()", fname);
    }
    efree(lcase_fname);
}
```

When the `php_error_docref()` line is encountered, the internal error handler sees that the error level is critical and invokes `longjmp()` to interrupt the current program flow and leave `call_function()` without ever reaching the `efree(lcase_fname)` line. Again, you're probably thinking that the `efree()` line could just be moved above the `zend_error()` line, but what about the code that called this `call_function()` routine in the first place? Most likely `fname` itself was an allocated string and you can't free that before it has been used in the error message.

> **Note**
>
> The `php_error_docref()` function is an internals equivalent to `trigger_error()`. The first parameter is an optional documentation reference that will be appended to `docref.root` if such is enabled in `php.ini`. The third parameter can be any of the familiar E_* family of constants indicating severity. The fourth and later parameters follow `printf()` style formatting and variable argument lists.

Zend Memory Manager

The solution to memory leaks during request bailout is the Zend Memory Management (ZendMM) layer. This portion of the engine acts in much the same way the operating system would normally act, allocating memory to calling applications. The difference is that it is low enough in the process space to be request-aware so that when one request dies, it can perform the same action the OS would perform when a process dies. That is, it implicitly frees all the memory owned by that request. Figure 3.1 shows ZendMM in relation to the OS and the PHP process.

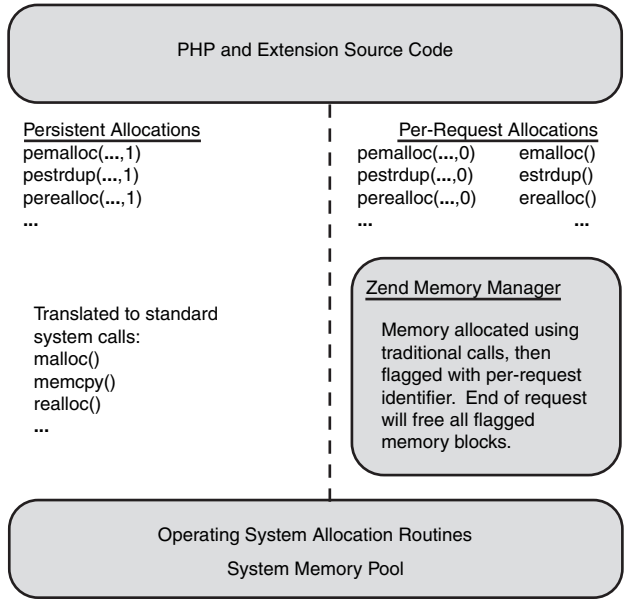

Figure 3.1 Zend Memory Manager replaces system calls for per-request allocations.

In addition to providing implicit memory cleanup, ZendMM also controls the per-request memory usage according to the `php.ini` setting: `memory_limit`. If a script attempts to ask for more memory than is available to the system as a whole, or more than is remaining in its per-request limit, ZendMM will automatically issue an E_ERROR message and begin the bailout process. An added benefit of this is that the return value of most memory allocation calls doesn't need to be checked because failure results in an immediate `longjmp()` to the shutdown part of the engine.

Hooking itself in between PHP internal code and the OS's actual memory management layer is accomplished by nothing more complex than requiring that all memory allocated internally is requested using an alternative set of functions. For example, rather than allocate a 16-byte block of memory using `malloc(16)`, PHP code will use `emalloc(16)`. In addition to performing the actual memory allocation task, ZendMM will flag that block with information concerning what request it's bound to so that when a request bails out, ZendMM can implicitly free it.

Often, memory needs to be allocated for longer than the duration of a single request. These types of allocations, called `persistent` allocations because they persist beyond the end of a request, could be performed using the traditional memory allocators because these do not add the additional per-request information used by ZendMM. Sometimes, however, it's not known until runtime whether a particular allocation will need to be

persistent or not, so ZendMM exports a set of helper macros that act just like the other memory allocation functions, but have an additional parameter at the end to indicate persistence.

If you genuinely want a persistent allocation, this parameter should be set to one, in which case the request will be passed through to the traditional `malloc()` family of allocators. If runtime logic has determined that this block does not need to be persistent however, this parameter may be set to zero, and the call will be channeled to the per-request memory allocator functions.

For example, `pemalloc(buffer_len, 1)` maps to `malloc(buffer_len)`, whereas `pemalloc(buffer_len, 0)` maps to `emalloc(buffer_len)` using the following `#define` in `Zend/zend_alloc.h`:

```
#define pemalloc(size, persistent) \
          ((persistent)?malloc(size):emalloc(size))
```

Each of the allocator functions found in ZendMM can be found below along with their more traditional counterparts.

Table 3.1 shows each of the allocator functions supported by ZendMM and their e/pe counterparts:

Table 3.1 **Traditional versus PHP-specific allocators**

Allocator funtion	e/pe counterpart
`void *malloc(size_t count);`	`void *emalloc(size_t count);`
	`void *pemalloc(size_t count, char persistent);`
`void *calloc(size_t count);`	`void *ecalloc(size_t count);`
	`void *pecalloc(size_t count, char persistent);`
`void *realloc(void *ptr, size_t count);`	`void *erealloc(void *ptr, size_t count);`
	`void *perealloc(void *ptr, size_t count, char persistent);`
`void *strdup(void *ptr);`	`void *estrdup(void *ptr);`
	`void *pestrdup(void *ptr, char persistent);`
`void free(void *ptr);`	`void efree(void *ptr);`
	`void pefree(void *ptr, char persistent);`

You'll notice that even `pefree()` requires the persistency flag. This is because at the time that `pefree()` is called, it doesn't actually know if `ptr` was a persistent allocation or not. Calling `free()` on a non-persistent allocation could lead to a messy double free, whereas

calling `efree()` on a persistent one will most likely lead to a segmentation fault as the memory manager attempts to look for management information that doesn't exist. Your code is expected to remember whether the data structure it allocated was persistent or not.

In addition to the core set of allocator functions, a few additional and quite handy ZendMM specific functions exist:

```
void *estrndup(void *ptr, int len);
```

Allocate `len+1` bytes of memory and copy `len` bytes from `ptr` to the newly allocated block. The behavior of `estrndup()` is roughly the following:

```
void *estrndup(void *ptr, int len)
{
    char *dst = emalloc(len + 1);
    memcpy(dst, ptr, len);
    dst[len] = 0;
    return dst;
}
```

The terminating NULL byte implicitly placed at the end of the buffer here ensures that any function that uses `estrndup()` for string duplication doesn't need to worry about passing the resulting buffer to a function that expects NULL terminated strings such as `printf()`. When using `estrndup()` to copy non-string data, this last byte is essentially wasted, but more often than not, the convenience outweighs the minor inefficiency.

```
void *safe_emalloc(size_t size, size_t count, size_t addtl);
void *safe_pemalloc(size_t size, size_t count, size_t addtl, char persistent);
```

The amount of memory allocated by these functions is the result of `((size * count) + addtl)`. You may be asking, "Why an extra function at all? Why not just use emalloc/pemalloc and do the math myself?" The reason comes in the name: *safe*. Although the circumstances leading up to it would be exceedingly unlikely, it's possible that the end result of such an equation might overflow the integer limits of the host platform. This could result in an allocation for a negative number of bytes, or worse, a positive number that is significantly smaller than what the calling program believed it requested. `safe_emalloc()` avoids this type of trap by checking for integer overflow and explicitly failing if such an overflow occurs.

Note

Not all memory allocation routines have a `p*` counterpart. For example, there is no `pestrndup()`, and `safe_pemalloc()` does not exist prior to PHP 5.1. Occasionally you'll need to work around these gaps in the ZendAPI.

Reference Counting

Careful memory allocation and freeing is vital to the long term performance of a multirequest process like PHP, but it's only half the picture. In order for a server that handles thousands of hits per second to function efficiently, each request needs to use as little memory as possible and perform the bare minimum amount of unnecessary data copying. Consider the following PHP code snippet:

```
<?php
    $a = 'Hello World';
    $b = $a;
    unset($a);
?>
```

After the first call, a single variable has been created, and a 12 byte block of memory has been assigned to it holding the string 'Hello World' along with a trailing NULL. Now look at the next two lines: `$b` is set to the same value as `$a`, and then `$a` is unset (freed).

If PHP treated every variable assignment as a reason to copy variable contents, an extra 12 bytes would need to be copied for the duplicated string and additional processor load would be consumed during the data copy. This action starts to look ridiculous when the third line has come along and the original variable is unset making the duplication of data completely unnecessary. Now take that one further and imagine what could happen when the contents of a 10MB file are loaded into two variables. That could take up 20MB where 10 would have been sufficient. Would the engine waste so much time and memory on such a useless endeavor?

You know PHP is smarter than that.

Remember that variable names and their values are actually two different concepts within the engine. The value itself is a nameless `zval*` holding, in this case, a string value. It was assigned to the variable `$a` using zend `_hash_add()`. What if two variable names could point to the same value?

```
{
    zval *helloval;
    MAKE_STD_ZVAL(helloval);
    ZVAL_STRING(helloval, "Hello World", 1);
    zend_hash_add(EG(active_symbol_table), "a", sizeof("a"),
                                           &helloval, sizeof(zval*), NULL);
    zend_hash_add(EG(active_symbol_table), "b", sizeof("b"),
                                           &helloval, sizeof(zval*), NULL);
}
```

At this point you could actually inspect either `$a` or `$b` and see that they both contain the string `"Hello World"`. Unfortunately, you then come to the third line: `unset($a);`. In this situation, `unset()` doesn't know that the data pointed to by the `$a` variable is also in use by another one so it just frees the memory blindly. Any subsequent accesses to `$b` will be looking at already freed memory space and cause the engine to crash. Hint: You don't want to crash the engine.

This is solved by the third of a zval's four members: `refcount`. When a variable is first created and set, its `refcount` is initialized to 1 because it's assumed to only be in use by the variable it is being created for. When your code snippet gets around to assigning `helloval` to `$b`, it needs to increase that `refcount` to 2 because the value is now "referenced" by two variables:

```
{
    zval *helloval;
    MAKE_STD_ZVAL(helloval);
    ZVAL_STRING(helloval, "Hello World", 1);
    zend_hash_add(EG(active_symbol_table), "a", sizeof("a"),
                                           &helloval, sizeof(zval*), NULL);
    ZVAL_ADDREF(helloval);
    zend_hash_add(EG(active_symbol_table), "b", sizeof("b"),
                                           &helloval, sizeof(zval*), NULL);
}
```

Now when `unset()` deletes the `$a` copy of the variable, it can see from the `refcount` parameter that someone else is interested in that data and it should actually just decrement the `refcount` and otherwise leave it alone.

Copy on Write

Saving memory through refcounting is a great idea, but what happens when you only want to change one of those variables? Consider this code snippet:

```
<?php
    $a = 1;
    $b = $a;
    $b += 5;
?>
```

Looking at the logic flow you would of course expect `$a` to still equal 1, and `$b` to now be 6. At this point you also know that Zend is doing its best to save memory by having `$a` and `$b` refer to the same zval after the second line, so what happens when the third line is reached and `$b` must be changed?

The answer is that Zend looks at refcount, sees that it's greater than one and `separates` it. Separation in the Zend engine is the process of destroying a reference pair and is the opposite of the process you just saw:

```
zval *get_var_and_separate(char *varname, int varname_len TSRMLS_DC)
{
    zval **varval, *varcopy;
    if (zend_hash_find(EG(active_symbol_table),
                       varname, varname_len + 1, (void**)&varval) == FAILURE) {
```

```
        /* Variable doesn't actually exist — fail out */
        return NULL;
    }
    if ((*varval)->refcount < 2) {
        /* varname is the only actual reference,
         * no separating to do
         */
        return *varval;
    }
    /* Otherwise, make a copy of the zval* value */
    MAKE_STD_ZVAL(varcopy);
    varcopy = *varval;
    /* Duplicate any allocated structures within the zval* */
    zval_copy_ctor(varcopy);

    /* Remove the old version of varname
     * This will decrease the refcount of varval in the process
     */
    zend_hash_del(EG(active_symbol_table), varname, varname_len + 1);

    /* Initialize the reference count of the
     * newly created value and attach it to
     * the varname variable
     */
    varcopy->refcount = 1;
    varcopy->is_ref = 0;
    zend_hash_add(EG(active_symbol_table), varname, varname_len + 1,
                                              &varcopy, sizeof(zval*), NULL);

    /* Return the new zval* */
    return varcopy;
}
```

Now that the engine has a `zval*` that it knows is only owned by the `$b` variable, it can convert it to a long and increment it by 5 according to the script's request.

Change on Write

The concept of reference counting also creates a new possibility for data manipulation in the form of what userspace scripters actually think of in terms of "referencing". Consider the following snippet of userspace code:

```
<?php
    $a = 1;
    $b = &$a;
    $b += 5;
?>
```

Being experienced in the ways of PHP code, you'll instinctively recognize that the value of `$a` will now be 6 even though it was initialized to 1 and never (directly) changed. This happens because when the engine goes to increment the value of `$b` by 5, it notices that `$b` is a reference to `$a` and says, "It's okay for me to change the value without separating it, because I want all reference variables to see the change."

But how does the engine know? Simple, it looks at the fourth and final element of the zval struct: `is_ref`. This is just a simple on/off bit value that defines whether the value is, in fact, part of a userspace-style reference set. In the previous code snippet, when the first line is executed, the value created for `$a` gets a refcount of 1, and an `is_ref` value of 0 because its only owned by one variable (`$a`), and no other variables have a change on write reference to it. At the second line, the refcount element of this value is incremented to 2 as before, except that this time, because the script included an ampersand to indicate full-reference, the `is_ref` element is set to 1.

Finally, at the third line, the engine once again fetches the value associated with `$b` and checks if separation is necessary. This time the value is not separated because of a check not included earlier. Here's the refcount check portion of `get_var_and_separate()` again, with an extra condition:

```
    if ((*varval)->is_ref || (*varval)->refcount < 2) {
        /* varname is the only actual reference,
         * or it's a full reference to other variables
         * either way: no separating to be done
         */
        return *varval;
    }
```

This time, even though the refcount is 2, the separation process is short-circuited by the fact that this value is a full reference. The engine can freely modify it with no concern about the values of other variables appearing to change magically on their own.

Separation Anxiety

With all this copying and referencing, there are a couple of combinations of events that can't be handled by clever manipulation of `is_ref` and `refcount`. Consider this block of PHP code:

```
<?php
    $a = 1;
    $b = $a;
    $c = &$a;
?>
```

Here you have a single value that needs to be associated with three different variables, two in a change-on-write full reference pair, and the third in a separable copy-on-write context. Using just `is_ref` and `refcount` to describe this relationship, what values will work?

The answer is: none. In this case, the value must be duplicated into two discrete `zval*`s, even though both will contain the exact same data (see Figure 3.2).

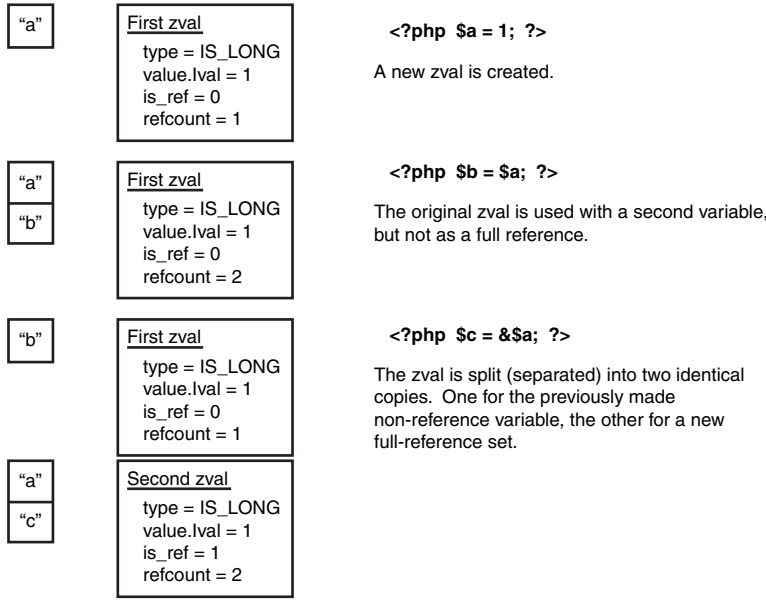

Figure 3.2 Forced separation on reference.

Similarly, the following code block will cause the same conflict and force the value to separate into a copy (see Figure 3.3).

```
<?php
    $a = 1;
    $b = &$a;
    $c = $a;
?>
```

Notice here that in both cases here, `$b` is associated with the original zval object because at the time separation occurs, the engine doesn't know the name of the third variable involved in the operation.

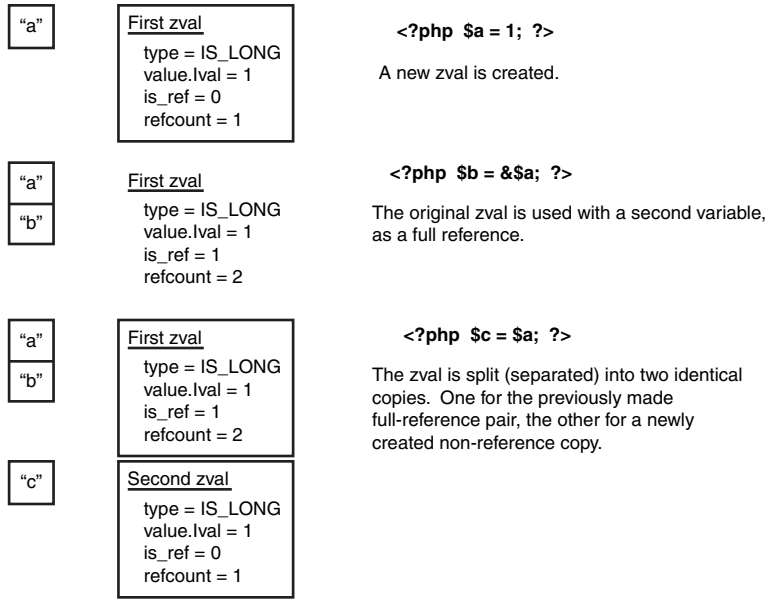

Figure 3.3 Forced separation on copy.

Summary

PHP is a managed language. On the userspace side of things, this careful control of resources and memory means easier prototyping and fewer crashes. After you delve under the hood though, all bets are off and it's up to the responsible developer to maintain the integrity of the runtime environment.