CHAPTER 9

Memory Management

Memory is among the most basic, but also most essential, of resources available to a process. This chapter covers the management of this resource: the allocation, manipulation, and eventual release of memory.

The verb *allocate*, which is the common term for obtaining memory, is a bit misleading, as it conjures up images of rationing a scarce resource for which demand outstrips supply. To be sure, many users would love more memory. On modern systems, however, the problem is not really one of sharing too little among too many, but of properly using and keeping track of the bounty.

In this chapter, you will learn about all of the approaches to allocating memory in various regions of a program, including each method's advantages and disadvantages. We'll also go over some ways to set and manipulate the contents of arbitrary memory regions and look at how to lock memory so it remains in RAM and your program runs no risk of having to wait for the kernel to page in data from swap space.

The Process Address Space

Linux, like any modern operating system, virtualizes its physical resource of memory. Processes do not directly address physical memory. Instead, the kernel associates each process with a unique *virtual address space*. This address space is *linear*, with addresses starting at zero, increasing contiguously to some maximum value. The address space is also *flat*: it exists in one space, directly accessible, without the need for segmentation.

Pages and Paging

Memory is composed of bits, of which (usually) eight make a byte. Bytes compose words, which in turn compose *pages*. For the purposes of memory management, the page is the most important of these: it is the smallest addressable unit of memory that the

memory management unit (MMU) can manage. Thus the virtual address space is carved up into pages. The machine architecture determines the *page size*. Typical sizes include 4 KB for 32-bit systems and 8 KB for 64-bit systems.[1]

A 32-bit address space contains roughly a million 4 KB pages; a 64-bit address space with 8 KB pages contains several magnitudes more. A process cannot necessarily access all of those pages; they may not correspond to anything. Thus, pages are either valid or invalid. A *valid page* is associated with an actual page of data, either in physical memory (RAM) or on secondary storage, such as a swap partition or file on disk. An *invalid page* is not associated with anything and represents an unused, unallocated piece of the address space. Accessing an invalid page results in a segmentation violation.

If a valid page is associated with data on secondary storage, a process cannot access that page until the data is brought into physical memory. When a process attempts to access such a page, the memory management unit generates a *page fault*. The kernel then intervenes, transparently *paging in* the data from secondary storage to physical memory. Because there is considerably more virtual memory than physical memory, the kernel may have to move data out of memory to make room for the data paging in. *Paging out* is the process of moving data from physical memory to secondary storage. To minimize subsequent page ins, the kernel attempts to page out the data that is the least likely to be used in the near future.

Sharing and copy-on-write

Multiple pages of virtual memory, even in different virtual address spaces owned by different processes, may map to a single physical page. This allows different virtual address spaces to *share* the data in physical memory. For example, at any given moment there is a good chance that many processes on the system are using the standard C library. With shared memory, each of these processes may map the library into their virtual address space, but only one copy need exist in physical memory. As a more explicit example, two processes may both map into memory a large database. While both of these processes will have the database in their virtual address spaces, it will exist in RAM only once.

The shared data may be read-only, writable, or both readable and writable. When a process writes to a shared writable page, one of two things can happen. The simplest is that the kernel allows the write to occur, in which case all processes sharing the page can see the results of the write operation. Usually, allowing multiple processes to read from or write to a shared page requires some level of coordination and synchronization

1. Some systems support multiple page sizes. For this reason, the page size is not part of the ABI. Applications must programmatically obtain the page size at runtime. We covered doing so in Chapter 4 and will review the topic in this chapter.

among the processes, but at the kernel level the write "just works" and all processes sharing the data instantly see the modifications.

Alternatively, the MMU can intercept the write operation and raise an exception; the kernel, in response, will transparently create a new copy of the page for the writing process, and allow the write to continue against the new page. We call this approach *copy-on-write* (*COW*).[2] Effectively, processes are allowed read access to shared data, which saves space. But when a process wants to write to a shared page, it receives a unique copy of that page on the fly, thereby allowing the kernel to act as if the process always had its own private copy. As copy-on-write occurs on a page-by-page basis, with this technique a huge file may be efficiently shared among many processes, and the individual processes will receive unique physical pages only for those pages to which they themselves write.

Memory Regions

The kernel arranges pages into blocks that share certain properties, such as access permissions. These blocks are called *mappings*, *memory areas*, or *memory regions*. Certain types of memory regions can be found in every process:

- The *text segment* contains a process's program code, string literals, constant variables, and other read-only data. In Linux, this segment is marked read-only and is mapped in directly from the object file (the program executable or a library).
- The *stack* contains the process's execution stack, which grows and shrinks dynamically as the stack depth increases and decreases. The execution stack contains local variables and function return data. In a multithreaded process, there is one stack per thread.
- The *data segment*, or *heap*, contains a process's dynamic memory. This segment is writable and can grow or shrink in size. `malloc()` (discussed in the next section) can satisfy memory requests from this segment.
- The *bss segment*[3] contains uninitialized global variables. These variables contain special values (essentially, all zeros), per the C standard.

Linux optimizes these variables in two ways. First, because the bss segment is dedicated to uninitialized data, the linker (*ld*) does not actually store the special values in the object file. This reduces the binary's size. Second, when this segment is loaded into memory, the kernel simply maps it on a copy-on-write basis to a page of zeros, efficiently setting the variables to their default values.

2. Recall from Chapter 5 that `fork()` uses copy-on-write to duplicate and share the parent's address space with the child.

3. The name is historic; it comes from *block started by symbol*.

Most address spaces contain a handful of *mapped files*, such as the program executable itself, the C and other shared libraries, and data files. Take a look at */proc/self/maps*, or the output from the *pmap* program for an example of the mapped files in a process.

This chapter covers the interfaces that Linux provides to obtain and return memory, create and destroy new mappings, and everything in between.

Allocating Dynamic Memory

Memory also comes in the form of automatic and static variables, but the foundation of any memory management system is the allocation, use, and eventual return of *dynamic memory*. Dynamic memory is allocated at runtime, not compile time, in sizes that may be unknown until the moment of allocation. As a developer, you need dynamic memory when the amount of memory that you will need, or how long you might need it, varies and is not known before the program runs. For example, you might want to store in memory the contents of a file or input read in from the keyboard. Because the size of the file is unknown, and the user may type any number of keystrokes, the size of the buffer will vary, and you may need to make it dynamically larger as you read more and more data.

There is no C variable that is backed by dynamic memory. For example, C does not provide a mechanism to obtain a `struct pirate_ship` that exists in dynamic memory. Instead, C provides a mechanism for allocating dynamic memory sufficient to hold a `pirate_ship` structure. The programmer then interacts with the memory via a pointer—in this case, a `struct pirate_ship*`.

The classic C interface for obtaining dynamic memory is `malloc()`:

```
#include <stdlib.h>

void * malloc (size_t size);
```

A successful call to `malloc()` allocates `size` bytes of memory and returns a pointer to the start of the newly allocated region. The contents of the memory are undefined; do not expect the memory to be zeroed. Upon failure, `malloc()` returns `NULL`, and `errno` is set to `ENOMEM`.

Usage of `malloc()` may be rather straightforward, as in this example used to allocate a fixed number of bytes:

```
char *p;

/* give me 2 KB! */
p = malloc (2048);
```

```
if (!p)
        perror ("malloc");
```

Or this example used to allocate a structure:

```
struct treasure_map *map;

/*
 * allocate enough memory to hold a treasure_map stucture
 * and point 'map' at it
 */
map = malloc (sizeof (struct treasure_map));
if (!map)
        perror ("malloc");
```

C automatically promotes pointers to `void` to any other pointer type on assignment. Thus, these examples do not need to typecast the return value of `malloc()` to the lvalue's type used in the assignments. The C++ programming language, however, does not perform automatic `void` pointer promotion. Consequently, users of C++ need to typecast `malloc()`'s return as follows:

```
char *name;

/* allocate 512 bytes */
name = (char *) malloc (512);
if (!name)
        perror ("malloc");
```

Some C programmers like to typecast the result of any function that returns a pointer to `void`, `malloc()` included. I argue against this practice because it will hide an error if the return value of the function ever changes to something other than a `void` pointer. Moreover, such a typecast also hides a bug if a function is not properly declared. While the former is not a risk with `malloc()`, the latter certainly is.

Undeclared functions default to returning an `int`. Integer-to-pointer casts are not automatic and generate a warning. The typecast will suppress the resulting warning.

Because `malloc()` can return `NULL`, it is vitally important that developers *always* check for and handle error conditions. Many programs define and use a `malloc()` wrapper that prints an error message and terminates the program if `malloc()` returns `NULL`. By convention, developers call this common wrapper `xmalloc()`:

```
/* like malloc(), but terminates on failure */
void * xmalloc (size_t size)
{
        void *p;
```

```
        p = malloc (size);
        if (!p) {
                perror ("xmalloc");
                exit (EXIT_FAILURE);
        }

        return p;
}
```

Allocating Arrays

Dynamic memory allocation may also be quite complex when the specified `size` is itself dynamic. One such example is the dynamic allocation of arrays, where the size of an array element may be fixed but the number of elements to allocate is dynamic. To simplify this scenario, C provides the `calloc()` function:

```
#include <stdlib.h>

void * calloc (size_t nr, size_t size);
```

A successful call to `calloc()` returns a pointer to a block of memory suitable for holding an array of `nr` elements, each of `size` bytes. Consequently, the amount of memory requested in these two calls is identical (either may end up returning more memory than requested, but never less):

```
int *x, *y;

x = malloc (50 * sizeof (int));
if (!x) {
        perror ("malloc");
        return -1;
}

y = calloc (50, sizeof (int));
if (!y) {
        perror ("calloc");
        return -1;
}
```

The behavior, however, is not identical. Unlike `malloc()`, which makes no such guarantees about the contents of allocated memory, `calloc()` zeros all bytes in the returned chunk of memory. Thus, each of the 50 elements in the array of integers `y` holds the value of `0`, while the contents of the elements in `x` are undefined. Unless the program is going to immediately set all 50 values, programmers should use `calloc()` to ensure that the array elements are not filled with gibberish. Note that binary zero might not be the same as floating-point zero!

On the Etymology of calloc()

There remains no original documentation on the etymology of `calloc()`. Unix historians debate the origin: Does the *c* stand for *count*, since the function accepts a count of array elements? Or perhaps it stands for *clear*, since the function zeros out memory? Take your pick. The debate is fierce.

In pursuit of the truth, I asked Brian Kernighan, the *K* in *K&R* and early Unix contributor, for his recollection. Brian cautioned that he didn't write the original function but that he believes the "*c* is for *clear*." That is likely as authoritative a view as we will get.

Users often want to "zero out" dynamic memory, even when not dealing with arrays. Later in this chapter, we will consider `memset()`, which provides an interface for setting every byte in a chunk of memory to a given value. Letting `calloc()` perform the zeroing, however, is faster because the kernel can provide memory that is already zeroed.

On failure, like `malloc()`, `calloc()` returns `NULL` and sets `errno` to `ENOMEM`.

Why the standards bodies never defined an "allocate and zero" function separate from `calloc()` is a mystery. Developers can easily define their own interface, however:

```
/* works identically to malloc(), but memory is zeroed */
void * malloc0 (size_t size)
{
        return calloc (1, size);
}
```

Conveniently, we can combine this `malloc0()` with our previous `xmalloc()`:

```
/* like malloc(), but zeros memory and terminates on failure */
void * xmalloc0 (size_t size)
{
        void *p;

        p = calloc (1, size);
        if (!p) {
                perror ("xmalloc0");
                exit (EXIT_FAILURE);
        }

        return p;
}
```

Resizing Allocations

The C language provides an interface for resizing (making larger or smaller) existing allocations:

```
#include <stdlib.h>

void * realloc (void *ptr, size_t size);
```

A successful call to `realloc()` resizes the region of memory pointed at by `ptr` to a new size of `size` bytes. It returns a pointer to the newly sized memory, which may or may not be the same as `ptr`. When enlarging a memory region, if `realloc()` is unable to enlarge the existing chunk of memory by growing the chunk *in situ*, the function may allocate a new region of memory `size` bytes in length, copy the old region into the new one, and free the old region. On any operation, the contents of the memory region are preserved up to the minimum of the old and the new sizes. Because of the potentiality of a copy, a `realloc()` operation to enlarge a memory region can be a relatively costly operation.

If `size` is `0`, the effect is the same as an invocation of `free()` on `ptr`.

If `ptr` is `NULL`, the result of the operation is the same as a fresh `malloc()`. If `ptr` is non-`NULL`, it must have been returned via a previous call to `malloc()`, `calloc()`, or `realloc()`.

On failure, `realloc()` returns `NULL` and sets `errno` to `ENOMEM`. The state of the memory pointed at by `ptr` is unchanged.

Let's consider an example of shrinking a memory region. First, we'll use `calloc()` to allocate enough memory to hold a two-element array of `map` structures:

```
struct map *p;

/* allocate memory for two map structures */
p = calloc (2, sizeof (struct map));
if (!p) {
        perror ("calloc");
        return -1;
}

/* use p[0] and p[1]... */
```

Now, let's assume we've found one of the treasures and no longer need the second map, so we decide to resize the memory and give half of the region back to the system. This wouldn't generally be a worthwhile operation, but it might be if the `map` structure were very large and we were going to hold the remaining map for a long time:

```
struct map *r;

/* we now need memory for only one map */
r = realloc (p, sizeof (struct map));
if (!r) {
        /* note that 'p' is still valid! */
        perror ("realloc");
        return -1;
```

```
}

/* use 'r'... */

free (r);
```

In this example, `p[0]` is preserved after the `realloc()` call. Whatever data was there before is still there. If the call returned failure, `p` is untouched and thus still valid. We can continue using it, and will eventually need to free it. Conversely, if the call succeeded, we ignore `p`, and in lieu use `r`. We now have the responsibility to free `r` when we're done.

Freeing Dynamic Memory

Unlike automatic allocations, which are automatically reaped when the stack unwinds, dynamic allocations are permanent parts of the process's address space until they are manually freed. The programmer thus bears the responsibility of returning dynamically allocated memory to the system. (Both static and dynamic allocations, of course, disappear when the entire process exits.)

Memory allocated with `malloc()`, `calloc()`, or `realloc()` must be returned to the system when no longer in use via `free()`:

```
#include <stdlib.h>

void free (void *ptr);
```

A call to `free()` frees the memory at `ptr`. The parameter `ptr` must have been previously returned by `malloc()`, `calloc()`, or `realloc()`. That is, you cannot use `free()` to free partial chunks of memory—say, half of a chunk of memory—by passing in a pointer halfway into an allocated block. Doing so will result in undefined memory, likely manifested as a crash.

`ptr` may be `NULL`, in which case `free()` silently returns. Thus, the oft-seen practice of checking `ptr` for `NULL` before calling `free()` is redundant.

Let's look at an example:

```
void print_chars (int n, char c)
{
        int i;

        for (i = 0; i < n; i++) {
                char *s;
                int j;

                /*
                 * Allocate and zero an i+2 element array
                 * of chars. Note that 'sizeof (char)'
                 * is always 1.
                 */
```

```
                s = calloc (i + 2, 1);
                if (!s) {
                        perror ("calloc");
                        break;
                }

                for (j = 0; j < i + 1; j++)
                        s[j] = c;

                printf ("%s\n", s);

                /* Okay, all done. Hand back the memory. */
                free (s);
        }
}
```

This example allocates `n` arrays of `chars` containing successively larger numbers of elements, ranging from two elements (2 bytes) up to `n` + 1 elements (`n` + 1 bytes). Then, for each array, the loop writes the character `c` into each byte except the last (leaving the 0 that is already in the last byte), prints the array as a string, and then frees the dynamically allocated memory.

Invoking `print_chars()` with `n` equal to `5` and `c` set to `X`, we get the following:

```
X
XX
XXX
XXXX
XXXXX
```

There are, of course, significantly more efficient ways of implementing this function. The point, however, is that we can dynamically allocate and free memory even when the size and the number of said allocations are known only at runtime.

Unix systems such as SunOS and SCO provide a variant of `free()` named `cfree()`, which, depending on the system, behaves the same as `free()` or receives three parameters, mirroring `calloc()`. In Linux, `free()` can handle memory obtained from any of the allocation mechanisms we have discussed thus far. `cfree()` should never be used, except for backward compatibility. The Linux version is the same as `free()`.

Note what the repercussions would be if this example did not invoke `free()`. The program would never return the memory to the system, and, even worse, it would lose its only reference to the memory—the pointer `s`—thereby making it impossible to ever access the memory. We call this type of programming error a *memory leak*. Memory leaks and similar dynamic memory mistakes are among the most common, and, unfortunately, the most detrimental mishaps in C programming. Because the C language

places full responsibility for managing memory on the programmer, C programmers must keep a fastidious eye on all memory allocations.

Another common C programming pitfall is *use-after-free*. This foible occurs when a block of memory is freed and then subsequently accessed. Once `free()` is called on a block of memory, a program must never again access its contents. Programmers must be particularly careful to watch for *dangling pointers*: non-`NULL` pointers that nevertheless point at invalid blocks of memory. An excellent tool to detect memory errors in your programs is *Valgrind*.

Alignment

Data alignment refers to the way data is arranged in memory. A memory address `A` is said to be n-byte aligned when n is a power-of-2 and `A` is a multiple of n. Processors, memory subsystems, and other components in a system have specific alignment requirements. For example, most processors operate on words and can only access memory addresses that are word-size-aligned. Similarly, as discussed, memory management units deal only in page-size-aligned addresses.

A variable located at a memory address that is a multiple of its size is said to be *naturally aligned*. For example, a 32-bit variable is naturally aligned if it is located in memory at an address that is a multiple of 4—in other words, if the address's lowest two bits are 0. Thus, a type that is *2n* bytes in size must have an address with the *n* least-significant bits set to 0.

Rules pertaining to alignment derive from hardware and thus differ from system to system. Some machine architectures have very stringent requirements on the alignment of data. Others are more lenient. Some systems generate a catchable error. The kernel can then choose to terminate the offending process or (more likely) manually perform the unaligned access (generally through multiple aligned accesses). This incurs a performance hit and sacrifices atomicity, but at least the process isn't terminated. When writing portable code, programmers must be careful to avoid violating alignment requirements.

Allocating aligned memory

For the most part, the compiler and the C library transparently handle alignment concerns. POSIX decrees that the memory returned via `malloc()`, `calloc()`, and `realloc()` be properly aligned for use with any of the standard C types. On Linux, these functions always return memory that is aligned along an 8-byte boundary on 32-bit systems and along a 16-byte boundary on 64-bit systems.

Occasionally, programmers require dynamic memory aligned along a larger boundary, such as a page. While motivations vary, the most common is a need to properly align

buffers used in direct block I/O or other software-to-hardware communication. For this purpose, POSIX 1003.1d provides a function named `posix_memalign()`:

```
/* one or the other -- either suffices */
#define _XOPEN_SOURCE 600
#define _GNU_SOURCE

#include <stdlib.h>

int posix_memalign (void **memptr,
                    size_t alignment,
                    size_t size);
```

A successful call to `posix_memalign()` allocates `size` bytes of dynamic memory, ensuring it is aligned along a memory address that is a multiple of `alignment`. The parameter `alignment` must be a power of 2 and a multiple of the size of a `void` pointer. The address of the allocated memory is placed in `memptr`, and the call returns `0`.

On failure, no memory is allocated, `memptr` is undefined, and the call returns one of the following error codes:

`EINVAL`
: The parameter `alignment` is not a power of 2 or is not a multiple of the size of a `void` pointer.

`ENOMEM`
: There is insufficient memory available to satisfy the requested allocation.

Note that `errno` is not set; the function directly returns these errors.

Memory obtained via `posix_memalign()` is freed via `free()`. Usage is simple:

```
char *buf;
int ret;

/* allocate 1 KB along a 256-byte boundary */
ret = posix_memalign (&buf, 256, 1024);
if (ret) {
        fprintf (stderr, "posix_memalign: %s\n",
                 strerror (ret));
        return -1;
}

/* use 'buf'... */

free (buf);
```

Before POSIX defined `posix_memalign()`, BSD and SunOS provided the following interfaces, respectively:

```
#include <malloc.h>
```

```
void * valloc (size_t size);
void * memalign (size_t boundary, size_t size);
```

The function `valloc()` operates identically to `malloc()`, except that the allocated memory is aligned along a page boundary. Recall from Chapter 4 that the system's page size is easily obtained via `getpagesize()`.

The function `memalign()` is similar to `posix_memalign`. It aligns the allocation along a boundary of `boundary` bytes, which must be a power of 2. In this example, both of these allocations return a block of memory sufficient to hold a `ship` structure, aligned along a page boundary:

```
struct ship *pirate, *hms;

pirate = valloc (sizeof (struct ship));
if (!pirate) {
        perror ("valloc");
        return -1;
}

hms = memalign (getpagesize (), sizeof (struct ship));
if (!hms) {
        perror ("memalign");
        free (pirate);
        return -1;
}

/* use 'pirate' and 'hms'... */

free (hms);
free (pirate);
```

On Linux, memory obtained via both of these functions is freeable via `free()`. This may not be the case on other Unix systems, some of which provide no mechanism for safely freeing memory allocated with these functions. Programs concerned with portability may have no choice but to not free memory allocated via these interfaces!

Linux programmers should use these two functions only for the purposes of portability with older systems; `posix_memalign()` is superior and standardized. All three of these interfaces are needed only if an alignment greater than that provided by `malloc()` is required.

Other alignment concerns

Alignment concerns extend beyond natural alignment of the standard types and dynamic memory allocations. For example, nonstandard and complex types have more complex requirements than the standard types. Further, alignment concerns are doubly important when assigning values between pointers of varying types and using typecasting.

Nonstandard and complex data types possess alignment requirements beyond the simple requirement of natural alignment. Four useful rules follow:

- The alignment requirement of a structure is that of its largest constituent type. For example, if a structure's largest type is a 32-bit integer that is aligned along a 4-byte boundary, the structure must be aligned along at least a 4-byte boundary as well.
- Structures also introduce the need for padding, which is used to ensure that each constituent type is properly aligned to that type's own requirement. Thus, if a `char` (with a probable alignment of 1 byte) finds itself followed by an `int` (with a probable alignment of 4 bytes), the compiler will insert 3 bytes of padding between the two types to ensure that the `int` lives on a 4-byte boundary. Programmers sometimes order the members of a structure—for example, by descending size—to minimize the space "wasted" by padding. The gcc option `-Wpadded` can aid in this endeavor, as it generates a warning whenever the compiler inserts implicit padding.
- The alignment requirement of a union is that of the largest unionized type.
- The alignment requirement of an array is that of the base type. Thus, arrays carry no requirement beyond a single instance of their type. This behavior results in the natural alignment of all members of an array.

As the compiler transparently handles most alignment requirements, it takes a bit of effort to expose potential issues. It is not unheard of, however, to encounter alignment concerns when dealing with pointers and casting.

Accessing data via a pointer recast from a lesser-aligned to a larger-aligned block of data can result in the processor loading data that is not properly aligned for the larger type. For example, in the following code snippet, the assignment of `c` to `badnews` attempts to read `c` as an `unsigned long`:

```
char greeting[] = "Ahoy Matey";
char *c = greeting[1];
unsigned long badnews = *(unsigned long *) c;
```

An `unsigned long` is naturally aligned along a 4- or 8-byte boundary; `c` is likely not aligned to that same boundary. Consequently, the load of `c`, when typecast, causes an alignment violation. Depending on the architecture, this can cause results ranging from as minor as a performance hit to as major as a program crash. On machine architectures that can detect but not properly handle alignment violations, the kernel sends the offending process the `SIGBUS` signal, which terminates the process. We will discuss signals in Chapter 10.

Examples such as this are more common than one might think. Real-world examples will not be quite so silly in appearance, but they will likely be less obvious as well.

Strict Aliasing

This typecasting example also violates strict aliasing, one of the least-understood aspects of C and C++. *Strict aliasing* is the requirement that an object is only accessed through the actual type of that object, a qualified (e.g., `const` or `volatile`) version of the actual type, a signed (or unsigned) version of the actual type, a `struct` or `union` that contains the actual type among its members, or a `char` pointer. For example, the common pattern of accessing a `uint32_t` through two `uint16_t` pointers violates strict aliasing.

Here's a succinct summary: *dereferencing a cast of a pointer from one type of variable to a different type is usually a violation of the strict aliasing rule*. If you have ever seen the *gcc* warning, "dereferencing type-punned pointer will break strict-aliasing rules," you have violated the rule. Strict aliasing has been part of C++ for a long time, but it was standardized in C only with C99. gcc, as the warning attests, enforces strict aliasing; doing so allows it to generate more optimal code.

For the curious, the actual rules are laid out in section 6.5 of the ISO C99 standard.

Managing the Data Segment

Unix systems historically have provided interfaces for directly managing the data segment. However, most programs have no direct use for these interfaces because `malloc()` and other allocation schemes are easier to use and more powerful. I'll cover these interfaces here to satisfy the curious and for the rare reader who wants to implement her own heap-based allocation mechanism:

```
#include <unistd.h>

int brk (void *end);
void * sbrk (intptr_t increment);
```

These functions derive their names from old-school Unix systems, where the heap and the stack lived in the same segment. Dynamic memory allocations in the heap grew upward from the bottom of the segment; the stack grew downward toward the heap from the top of the segment. The line of demarcation separating the two was called the *break* or the *break point*. On modern systems where the data segment lives in its own memory mapping, we continue to label the end address of the mapping the break point.

A call to `brk()` sets the break point (the end of the data segment) to the address specified by `end`. On success, it returns `0`. On failure, it returns `-1` and sets `errno` to `ENOMEM`.

A call to `sbrk()` increments the end of the data segment by `increment` bytes, which may be a positive or negative delta. `sbrk()` returns the revised break point. Thus, an `increment` of `0` provides the current break point:

```
printf ("The current break point is %p\n", sbrk (0));
```

Deliberately, both POSIX and the C standard define neither of these functions. Nearly all Unix systems, however, support one or both. Portable programs should stick to the standards-based interfaces.

Anonymous Memory Mappings

Memory allocation in *glibc* uses a combination of the data segment and memory mappings. The classic method of implementing `malloc()` is to divide the data segment into a series of power-of-2 partitions and satisfy allocations by returning the partition that is the closest fit to the requested size. Freeing memory is as simple as marking the partition as "free." If adjacent partitions are free, they can be coalesced into a single, larger partition. If the top of the heap is entirely free, the system can use `brk()` to lower the break point, shrinking the heap and returning memory to the kernel.

This algorithm is called a *buddy memory allocation scheme*. It has the upside of speed and simplicity but the downside of introducing two types of fragmentation. *Internal fragmentation* occurs when more memory than requested is used to satisfy an allocation. This results in inefficient use of the available memory. *External fragmentation* occurs when sufficient memory is free to satisfy a request, but it is split into two or more nonadjacent chunks. This can result in inefficient use of memory (because a larger, less suitable block may be used), or failed memory allocations (if no alternative block exists).

Moreover, this scheme allows one memory allocation to "pin" another, preventing a traditional C library from returning freed memory to the kernel. Imagine that two blocks of memory, block *A* and block *B*, are allocated. Block *A* sits right on the break point, and block *B* sits right below *A*. Even if the program frees *B*, the C library cannot adjust the break point until *A* is likewise freed. In this manner, a long-living allocation can pin all other allocations in memory.

This is not always a concern as C libraries do not strictly return memory to the system. Generally, the heap is not shrunk after each free. Instead, the `malloc()` implementation keeps freed memory around for a subsequent allocation. Only when the size of the heap is significantly larger than the amount of allocated memory does `malloc()` shrink the data segment. A large allocation, however, can prevent this shrinkage.

Consequently, for large allocations, *glibc* does not use the heap. Instead, *glibc* creates an *anonymous memory mapping* to satisfy the allocation request. Anonymous memory mappings are similar to the file-based mappings discussed in Chapter 4, except that they are not backed by any file—hence the "anonymous" moniker. Instead, an anonymous

memory mapping is simply a large, zero-filled block of memory, ready for your use. Think of it as a brand new heap, solely for a single allocation. Because these mappings are located outside of the heap, they do not contribute to the data segment's fragmentation.

Allocating memory via anonymous mappings has several benefits:

- No fragmentation concerns. When the program no longer needs an anonymous memory mapping, the mapping is unmapped, and the memory is immediately returned to the system.
- Anonymous memory mappings are resizable, have adjustable permissions, and can receive advice just like normal mappings (see Chapter 4).
- Each allocation exists in a separate memory mapping. There is no need to manage the global heap.

There are also two downsides to using anonymous memory mappings rather than the heap:

- Each memory mapping is an integer multiple of the system page size in size. Ergo, allocations that are not integer multiples of pages in size result in wasted "slack" space. This slack space is more of a concern with small allocations, where the wasted space is large relative to the allocation size.
- Creating a new memory mapping incurs more overhead than satisfying allocations from the heap, which may not involve any kernel interaction whatsoever. The smaller the allocation, the more this overhead is detrimental.

Juggling the pros against the cons, *glibc*'s `malloc()` uses the data segment to satisfy small allocations and anonymous memory mappings to satisfy large allocations. The threshold is configurable (see "Advanced Memory Allocation" on page 312), and may change from one *glibc* release to another. Currently, the threshold is 128 KB: allocations smaller than or equal to 128 KB derive from the heap, whereas larger allocations derive from anonymous memory mappings.

Creating Anonymous Memory Mappings

Perhaps because you want to force the use of a memory mapping over the heap for a specific allocation, or perhaps because you are writing your own memory allocation system, you may want to manually create your own anonymous memory mapping—either way, Linux makes it easy. Recall from Chapter 4 that the system call `mmap()` creates a memory mapping and the system call `munmap()` destroys a mapping:

```
#include <sys/mman.h>

void * mmap (void *start,
```

```
                size_t length,
                int prot,
                int flags,
                int fd,
                off_t offset);

int munmap (void *start, size_t length);
```

Creating an anonymous memory mapping is actually easier than creating a file-backed mapping, as there is no file to open and manage. The primary difference is the presence of a special flag, signifying that the mapping is anonymous.

Let's look at an example:

```
void *p;

p = mmap (NULL,                         /* do not care where */
          512 * 1024,                   /* 512 KB */
          PROT_READ | PROT_WRITE,       /* read/write */
          MAP_ANONYMOUS | MAP_PRIVATE, /* anonymous, private */
          -1,                           /* fd (ignored) */
          0);                           /* offset (ignored) */

if (p == MAP_FAILED)
        perror ("mmap");
else
        /* 'p' points at 512 KB of anonymous memory... */
```

For most anonymous mappings, the parameters to `mmap()` mirror this example, with the exception, of course, of passing in whatever size (in bytes) the programmer desires. The other parameters are generally as follows:

- The first parameter, `start`, is set to `NULL`, signifying that the anonymous mapping may begin anywhere in memory that the kernel wishes. Specifying a non-`NULL` value here is possible, so long as it is page-aligned, but limits portability. Rarely does a program care where mappings exist in memory.
- The `prot` parameter usually sets both the `PROT_READ` and `PROT_WRITE` bits, making the mapping readable and writable. An empty mapping is of no use if you cannot read from and write to it. On the other hand, executing code from an anonymous mapping is rarely desired, and allowing execution opens up an attack vector.
- The `flags` parameter sets the `MAP_ANONYMOUS` bit, making this mapping anonymous, and the `MAP_PRIVATE` bit, making this mapping private.
- The `fd` and `offset` parameters are ignored when `MAP_ANONYMOUS` is set. Some older systems, however, expect a value of `-1` for `fd`, so it is a good idea to pass that if portability is a concern.

Memory obtained via an anonymous mapping looks the same as memory obtained via the heap. One benefit to allocating from anonymous mappings is that the pages are already filled with zeros. This occurs at no cost, because the kernel maps the application's anonymous pages to a zero-filled page via copy-on-write. Thus, there is no need to `memset()` the returned memory. Indeed, this is one benefit to using `calloc()` as opposed to `malloc()` followed by `memset()`: *glibc* knows that anonymous mappings are already zeroed, and that a `calloc()` satisfied from a mapping does not require explicit zeroing.

The system call `munmap()` frees an anonymous mapping, returning the allocated memory to the kernel:

```
int ret;

/* all done with 'p', so give back the 512 KB mapping */
ret = munmap (p, 512 * 1024);
if (ret)
        perror ("munmap");
```

For a review of `mmap()`, `munmap()`, and mappings in general, see Chapter 4.

Mapping /dev/zero

Other Unix systems, such as BSD, do not have a `MAP_ANONYMOUS` flag. Instead, they implement a similar solution by mapping a special device file, */dev/zero*. This device file provides identical semantics to anonymous memory. A mapping contains copy-on-write pages of all zeros; the behavior is thus the same as with anonymous memory.

Linux has always provided a */dev/zero* device and the ability to map it and obtain zero-filled memory. Indeed, before the introduction of `MAP_ANONYMOUS`, Linux programmers used this BSD-style approach. To provide backward compatibility with older versions of Linux, or portability to other Unix systems, developers can still map */dev/zero* in lieu of creating an anonymous mapping. The syntax is no different from mapping any other file:

```
void *p;
int fd;

/* open /dev/zero for reading and writing */
fd = open ("/dev/zero", O_RDWR);
if (fd < 0) {
        perror ("open");
        return -1;
}

/* map [0,page size) of /dev/zero */
```

```
p = mmap (NULL,                  /* do not care where */
          getpagesize (),        /* map one page */
          PROT_READ | PROT_WRITE, /* map read/write */
          MAP_PRIVATE,           /* private mapping */
          fd,                    /* map /dev/zero */
          0);                    /* no offset */

if (p == MAP_FAILED) {
        perror ("mmap");
        if (close (fd))
                perror ("close");
        return -1;
}

/* close /dev/zero, no longer needed */
if (close (fd))
        perror ("close");

/* 'p' points at one page of memory, use it... */
```

Memory mapped in this manner is, of course, unmapped using `munmap()`.

This approach involves the additional system call overhead of opening and closing the device file. Thus, anonymous memory is a faster solution.

Advanced Memory Allocation

Many of the allocation operations discussed in this chapter are limited and controlled by *glibc* or kernel parameters that the programmer can change. To do so, use the `mallopt()` call:

```
#include <malloc.h>

int mallopt (int param, int value);
```

A call to `mallopt()` sets the memory-management-related parameter specified by `param` to the value specified by `value`. On success, the call returns a nonzero value; on failure, it returns `0`. Note that `mallopt()` does not set `errno`. It also tends to always return success, so avoid any optimism over receiving useful information from the return value.

Linux currently supports seven values for `param`, all defined in `<malloc.h>`:

`M_CHECK_ACTION`
: The value of the `MALLOC_CHECK_` environment variable (discussed in the next section).

`M_MMAP_MAX`
: The maximum number of mappings that the system will create to satisfy dynamic memory requests. When this limit is reached, the data segment will be used for all

allocations until one of the previously created mappings is freed. A value of `0` disables all use of anonymous mappings as a basis for dynamic memory allocations.

`M_MMAP_THRESHOLD`
: The threshold (measured in bytes) over which an allocation request will be satisfied via an anonymous mapping instead of the data segment. Note that allocations smaller than this threshold may also be satisfied via anonymous mappings at the system's discretion. A value of `0` enables the use of anonymous mappings for all allocations, effectively disabling use of the data segment for dynamic memory allocations.

`M_MXFAST`
: The maximum size (in bytes) of a fast bin. *Fast bins* are special chunks of memory in the heap that are never coalesced with adjacent chunks and never returned to the system, allowing for very quick allocations at the cost of increased fragmentation. A value of `0` disables all use of fast bins.

`M_PERTURB`
: Enables memory poisoning, which aids in the detection of memory management errors. If provided a nonzero `value`, *glibc* sets all allocated bytes (except those requested via `calloc()`) to the logical compliment of the least-significant byte in `value`. This helps detect use-before-initialized errors. Moreover, *glibc* also sets all freed bytes to the least-significant byte in `value`. This helps detect use-after-free errors.

`M_TOP_PAD`
: The amount of padding (in bytes) used when adjusting the size of the data segment. Whenever *glibc* uses `brk()` to increase the size of the data segment, it can ask for more memory than needed in the hopes of alleviating the need for an additional `brk()` call in the near future. Likewise, whenever *glibc* shrinks the size of the data segment, it can keep extra memory, giving back a little less than it would otherwise. These extra bytes are the *padding*. A value of `0` disables all use of padding.

`M_TRIM_THRESHOLD`
: The minimum amount of free memory (in bytes) at the top of the data segment before *glibc* invokes `sbrk()` to return memory to the kernel.

The XPG standard, which loosely defines `mallopt()`, specifies three other parameters: `M_GRAIN`, `M_KEEP`, and `M_NLBLKS`. Linux defines these parameters, but setting their value has no effect. See Table 9-1 for a full listing of all valid parameters, their default values, and their ranges of accepted values.

Table 9-1. mallopt() parameters

Parameter	Origin	Default value	Valid values	Special values
M_CHECK_ACTION	Linux-specific	0	0 – 2	
M_GRAIN	XPG standard	Unsupported on Linux		
M_KEEP	XPG standard	Unsupported on Linux		
M_MMAP_MAX	Linux-specific	64 * 1024	>= 0	0 disables use of mmap()
M_MMAP_THRESHOLD	Linux-specific	128 * 1024	>= 0	0 disables use of the heap
M_MXFAST	XPG standard	64	0 - 80	0 disables fast bins
M_NLBLKS	XPG standard	Unsupported on Linux		
M_PERTURB	Linux-specific	0	0 or 1	0 disables perturbation
M_TOP_PAD	Linux-specific	0	>= 0	0 disables top padding
M_TRIM_THRESHOLD	Linux-specific	128 * 1024	>= -1	-1 disables trimming

Programs must make any invocations of mallopt() before their first call to malloc() or any other memory allocation interface. Usage is simple:

```
int ret;

/* use mmap() for all allocations over 64 KB */
ret = mallopt (M_MMAP_THRESHOLD, 64 * 1024);
if (!ret)
        fprintf (stderr, "mallopt failed!\n");
```

Fine-Tuning with malloc_usable_size() and malloc_trim()

Linux provides a couple of functions that offer low-level control of *glibc*'s memory allocation system. The first such function allows a program to ask how many usable bytes a given memory allocation contains:

```
#include <malloc.h>

size_t malloc_usable_size (void *ptr);
```

A successful call to `malloc_usable_size()` returns the actual allocation size of the chunk of memory pointed to by `ptr`. Because *glibc* may round up allocations to fit within an existing chunk or anonymous mapping, the usable space in an allocation can be larger than requested. Of course, the allocation will never be smaller than requested. Here's an example of the function's use:

```
size_t len = 21;
size_t size;
char *buf;

buf = malloc (len);
if (!buf) {
        perror ("malloc");
```

```
        return -1;
}

size = malloc_usable_size (buf);

/* we can actually use 'size' bytes of 'buf' ... */
```

The second of the two functions allows a program to force *glibc* to return all immediately freeable memory to the kernel:

```
#include <malloc.h>

int malloc_trim (size_t padding);
```

A successful call to `malloc_trim()` shrinks the data segment as much as possible, minus `padding` bytes, which are reserved. It then returns `1`. On failure, the call returns `0`. Normally, *glibc* performs such shrinking automatically whenever the freeable memory reaches `M_TRIM_THRESHOLD` bytes. It uses a padding of `M_TOP_PAD`.

You'll almost never want to use these two functions for anything other than debugging or educational purposes. They are not portable and expose low-level details of *glibc*'s memory allocation system to your program.

Debugging Memory Allocations

Programs can set the environment variable `MALLOC_CHECK_` to enable enhanced debugging in the memory subsystem. The additional debugging checks come at the expense of less efficient memory allocations, but the overhead is often worth it during the debugging stage of application development.

Because an environment variable controls the debugging, there is no need to recompile your program. For example, you can simply issue a command like the following:

```
$ MALLOC_CHECK_=1 ./rudder
```

If `MALLOC_CHECK_` is set to `0`, the memory subsystem silently ignores any errors. If it is set to `1`, an informative message is printed to `stderr`. If it is set to `2`, the program is immediately terminated via `abort()`. Because `MALLOC_CHECK_` changes the behavior of the running program, setuid programs ignore this variable.

Obtaining Statistics

Linux provides the `mallinfo()` function for obtaining statistics related to the memory allocation system:

```
#include <malloc.h>

struct mallinfo mallinfo (void);
```

A call to `mallinfo()` returns statistics in a `mallinfo` structure. The structure is returned by value, not via a pointer. Its contents are also defined in `<malloc.h>`:

```
/* all sizes in bytes */

struct mallinfo {
        int arena;    /* size of data segment used by malloc */
        int ordblks;  /* number of free chunks */
        int smblks;   /* number of fast bins */
        int hblks;    /* number of anonymous mappings */
        int hblkhd;   /* size of anonymous mappings */
        int usmblks;  /* maximum total allocated size */
        int fsmblks;  /* size of available fast bins */
        int uordblks; /* size of total allocated space */
        int fordblks; /* size of available chunks */
        int keepcost; /* size of trimmable space */
};
```

Usage is simple:

```
struct mallinfo m;

m = mallinfo ();

printf ("free chunks: %d\n", m.ordblks);
```

Linux also provides the `malloc_stats()` function, which prints memory-related statistics to `stderr`:

```
#include <malloc.h>

void malloc_stats (void);
```

Invoking `malloc_stats()` in a memory-intensive program yields some big numbers:

```
Arena 0:
system bytes     =  865939456
in use bytes     =  851988200
Total (incl. mmap):
system bytes     = 3216519168
in use bytes     = 3202567912
max mmap regions =      65536
max mmap bytes   = 2350579712
```

Stack-Based Allocations

Thus far, ball of the mechanisms for dynamic memory allocation that we have studied have used the heap or memory mappings to obtain dynamic memory. We should expect this because the heap and memory mappings are decidedly dynamic in nature. The other common construct in a program's address space, the stack, is where a program's *automatic variables* live.

There is no reason, however, that a programmer cannot use the stack for dynamic memory allocations. So long as the allocation does not overflow the stack, such an approach should be easy and should perform quite well. To make a dynamic memory allocation from the stack, use the `alloca()` system call:

```
#include <alloca.h>

void * alloca (size_t size);
```

On success, a call to `alloca()` returns a pointer to `size` bytes of memory. This memory lives on the stack and is automatically freed when the invoking function returns. Some implementations return `NULL` on failure, but most `alloca()` implementations cannot fail or are unable to report failure. Failure is manifested as a stack overflow.

Usage is identical to `malloc()`, but you do not need to (indeed, must not) free the allocated memory. Here is an example of a function that opens a given file in the system's configuration directory, which is probably */etc*, but is portably determined at compile time. The function has to allocate a new buffer, copy the system configuration directory into the buffer, and then concatenate this buffer with the provided filename:

```
int open_sysconf (const char *file, int flags, int mode)
{
        const char *etc = SYSCONF_DIR; /* "/etc/" */
        char *name;

        name = alloca (strlen (etc) + strlen (file) + 1);
        strcpy (name, etc);
        strcat (name, file);

        return open (name, flags, mode);
}
```

Upon return, the memory allocated with `alloca()` is automatically freed as the stack unwinds back to the invoking function. This means you cannot use this memory once the function that calls `alloca()` returns! However, because you don't have to do any cleanup by calling `free()`, the resulting code is a bit cleaner. Here is the same function implemented using `malloc()`:

```
int open_sysconf (const char *file, int flags, int mode)
{
        const char *etc = SYSCONF_DIR; /* "/etc/" */
        char *name;
        int fd;

        name = malloc (strlen (etc) + strlen (file) + 1);
        if (!name) {
                perror ("malloc");
                return -1;
        }
```

```
        strcpy (name, etc);
        strcat (name, file);
        fd = open (name, flags, mode);
        free (name);

        return fd;
}
```

Note that you should not use `alloca()`-allocated memory in the parameters to a function call because the allocated memory will then exist in the middle of the stack space reserved for the function parameters. For example, the following is off-limits:

```
/* DO NOT DO THIS! */
ret = foo (x, alloca (10));
```

The `alloca()` interface has a checkered history. On many systems, it behaved poorly or gave way to undefined behavior. On systems with a small and fixed-sized stack, using `alloca()` was an easy way to overflow the stack and kill your program. On still other systems, `alloca()` did not even exist. Over time, the buggy and inconsistent implementations earned `alloca()` a bad reputation.

So, if your program must remain portable, you should avoid `alloca()`. On Linux, however, `alloca()` is a wonderfully useful and underutilized tool. It performs exceptionally well—on many architectures, an allocation via `alloca()` does as little as increment the stack pointer—and handily outperforms `malloc()`. For small allocations in Linux-specific code, `alloca()` can yield excellent performance gains.

Duplicating Strings on the Stack

A very common use of `alloca()` is to temporarily duplicate a string. For example:

```
/* we want to duplicate 'song' */
char *dup;

dup = alloca (strlen (song) + 1);
strcpy (dup, song);

/* manipulate 'dup'... */

return; /* 'dup' is automatically freed */
```

Because of the frequency of this need and the speed benefit that `alloca()` offers, Linux systems provide variants of `strdup()` that duplicate the given string onto the stack:

```
#define _GNU_SOURCE
#include <string.h>

char * strdupa (const char *s);
char * strndupa (const char *s, size_t n);
```

A call to `strdupa()` returns a duplicate of `s`. A call to `strndupa()` duplicates up to `n` characters of `s`. If `s` is longer than `n`, the duplication stops at `n`, and the function appends a null byte. These functions offer the same benefits as `alloca()`. The duplicated string is automatically freed when the invoking function returns.

POSIX does not define the `alloca()`, `strdupa()`, or `strndupa()` functions, and their record on other operating systems is spotty. If portability is a concern, use of these functions is highly discouraged. On Linux, however, `alloca()` and friends perform quite well and can provide an excellent performance boost, replacing the complicated dance of dynamic memory allocation with a mere adjustment of the stack frame pointer.

Variable-Length Arrays

C99 introduced *variable-length arrays* (VLAs), which are arrays whose geometry is set at runtime, not at compile time. GNU C has supported variable-length arrays for some time, but now that C99 has standardized them, there is greater incentive for their use. VLAs avoid the overhead of dynamic memory allocation in much the same way as `alloca()`.

Their use is exactly what you would expect:

```
for (i = 0; i < n; ++i) {
        char foo[i + 1];

        /* use 'foo'... */
}
```

In this snippet, `foo` is an array of `char`s of variable size `i + 1`. On each iteration of the loop, `foo` is dynamically created and automatically cleaned up when it falls out of scope. If we used `alloca()` instead of a VLA, the memory would not be freed until the function returned. Using a VLA ensures that the memory is freed on every iteration of the loop. Thus, using a VLA consumes at worst `n` bytes, whereas `alloca()` would consume `n * (n+1) / 2` bytes.

Using a variable-length array, we can rewrite our `open_sysconf()` function as follows:

```
int open_sysconf (const char *file, int flags, int mode)
{
        const char *etc; = SYSCONF_DIR; /* "/etc/" */
        char name[strlen (etc) + strlen (file) + 1];

        strcpy (name, etc);
        strcat (name, file);

        return open (name, flags, mode);
}
```

The main difference between `alloca()` and variable-length arrays is that memory obtained via the former exists for the duration of the function, whereas memory obtained via the latter exists until the holding variable falls out of scope, which can be before the current function returns. This could be welcome or unwelcome. In the `for` loop we just looked at, reclaiming the memory on each loop iteration reduces net memory consumption without any side effect (we did not need the extra memory hanging around). However, if for some reason we wanted the memory to persist longer than a single loop iteration, using `alloca()` would make more f158.60sense.

Mixing `alloca()` and variable-length arrays in a single function can invite peculiar behavior. Play it safe and use one or the other in a given function.

Choosing a Memory Allocation Mechanism

The myriad memory allocation options discussed in this chapter may leave programmers wondering exactly what solution is best for a given job. In the majority of situations, `malloc()` is your best bet. Sometimes, however, a different approach provides a better tool. Table 9-2 summarizes guidelines for choosing an allocation mechanism.

Table 9-2. Approaches to memory allocation in Linux

Allocation approach	Pros	Cons
`malloc()`	Easy, simple, common.	Returned memory not necessarily zeroed.
`calloc()`	Makes allocating arrays simple, zeros returned memory.	Convoluted interface if not allocating arrays.
`realloc()`	Resizes existing allocations.	Useful only for resizing existing allocations.
`brk()` and `sbrk()`	Provides intimate control over the heap.	Much too low-level for most users.
Anonymous memory mappings	Easy to work with, sharable, allow developer to adjust protection level and provide advice; optimal for large mappings.	Suboptimal for small allocations; `malloc()` automatically uses anonymous memory mappings when optimal.
`posix_memalign()`	Allocates memory aligned to any reasonable boundary.	Relatively new and thus portability is questionable; overkill unless alignment concerns are pressing.
`memalign()` and `valloc()`	More common on other Unix systems than `posix_memalign()`.	Not a POSIX standard, offers less alignment control than `posix_memalign()`.
`alloca()`	Very fast allocation, no need to explicitly free memory; great for small allocations.	Unable to return error, no good for large allocations, broken on some Unix systems.
Variable-length arrays	Same as `alloca()`, but frees memory when array falls out of scope, not when function returns.	Useful only for arrays; `alloca()` freeing behavior may be preferable in some situations; less common on other Unix systems than `alloca()`.

Finally, let us not forget the alternative to all of these options: automatic and static memory allocations. Allocating automatic variables on the stack or global variables on the heap is often easier and does not require that the programmer manage pointers and worry about freeing the memory.

Manipulating Memory

The C language provides a family of functions for manipulating raw bytes of memory. These functions operate in many ways similarly to string-manipulation interfaces such as `strcmp()` and `strcpy()`, but they rely on a user-provided buffer size instead of the assumption that strings are null-terminated. Note that none of these functions can return errors. Preventing errors is up to the programmer—pass in the wrong memory region, and there is no alternative, except the resulting segmentation violation!

Setting Bytes

Among the collection of memory-manipulating functions, the most common is easily `memset()`:

```
#include <string.h>

void * memset (void *s, int c, size_t n);
```

A call to `memset()` sets the `n` bytes starting at `s` to the byte `c` and returns `s`. A frequent use is zeroing a block of memory:

```
/* zero out [s,s+256) */
memset (s, '\0', 256);
```

`bzero()` is an older, deprecated interface introduced by BSD for performing the same task. New code should use `memset()`, but Linux provides `bzero()` for backward compatibility and portability with other systems:

```
#include <strings.h>

void bzero (void *s, size_t n);
```

The following invocation is identical to the preceding `memset()` example:

```
bzero (s, 256);
```

Note that `bzero()` (along with the other `b` interfaces) requires the header `<strings.h>` and not `<string.h>`.

Do Not Use memset() if You Can Use calloc()!

Avoid allocating memory with `malloc()` only to immediately zero the provided memory with `memset()`. While the result may be the same, foregoing the two functions for a single `calloc()`, which returns zeroed memory, is superior. Not only will you make one less function call, but `calloc()` may be able to obtain already zeroed memory from the kernel. In that case, you avoid manually setting each byte to 0, improving performance.

Comparing Bytes

Similar to `strcmp()`, `memcmp()` compares two chunks of memory for equivalence:

```
#include <string.h>

int memcmp (const void *s1, const void *s2, size_t n);
```

An invocation compares the first `n` bytes of `s1` to `s2` and returns `0` if the blocks of memory are equivalent, a value less than zero if `s1` is less than `s2`, and a value greater than zero if `s1` is greater than `s2`.

BSD again provides a now-deprecated interface that performs largely the same task:

```
#include <strings.h>

int bcmp (const void *s1, const void *s2, size_t n);
```

An invocation of `bcmp()` compares the first `n` bytes of `s1` to `s2`, returning `0` if the blocks of memory are equivalent, and a nonzero value if they are different.

Because of structure padding (see "Other alignment concerns" on page 305 earlier in this chapter), comparing two structures for equivalence via `memcmp()` or `bcmp()` is unreliable. There can be uninitialized garbage in the padding that differs between two otherwise identical instances of a structure. Consequently, code such as the following is not safe:

```
/* are two dinghies identical? (BROKEN) */
int compare_dinghies (struct dinghy *a, struct dinghy *b)
{
        return memcmp (a, b, sizeof (struct dinghy));
}
```

Instead, programmers who wish to compare structures should compare each element of the structures, one by one. This approach allows for some optimization, but it's definitely more work than the unsafe `memcmp()` approach. Here's the equivalent code:

```
/* are two dinghies identical? */
int compare_dinghies (struct dinghy *a, struct dinghy *b)
{
        int ret;
```

```
        if (a->nr_oars < b->nr_oars)
                return -1;
        if (a->nr_oars > b->nr_oars)
                return 1;

        ret = strcmp (a->boat_name, b->boat_name);
        if (ret)
                return ret;

        /* and so on, for each member... */
}
```

Moving Bytes

`memmove()` copies the first `n` bytes of `src` to `dst`, returning `dst`:

```
#include <string.h>

void * memmove (void *dst, const void *src, size_t n);
```

Again, BSD provides a deprecated interface for performing the same task:

```
#include <strings.h>

void bcopy (const void *src, void *dst, size_t n);
```

Note that although both functions take the same parameters, the order of the first two is reversed in `bcopy()`.

Both `bcopy()` and `memmove()` can safely handle overlapping memory regions (say, if part of `dst` is inside of `src`). This allows bytes of memory to shift up or down within a given region, for example. As this situation is rare, and a programmer would know if it were the case, the C standard defines a variant of `memmove()` that does not support overlapping memory regions. This variant is potentially faster:

```
#include <string.h>

void * memcpy (void *dst, const void *src, size_t n);
```

This function behaves identically to `memmove()`, except `dst` and `src` may not overlap. If they do, the results are undefined.

Another safe copying function is `memccpy()`:

```
#include <string.h>

void * memccpy (void *dst, const void *src, int c, size_t n);
```

The `memccpy()` function behaves the same as `memcpy()`, except that it stops copying if the function finds the byte `c` within the first `n` bytes of `src`. The call returns a pointer to the next byte in `dst` after `c`, or `NULL` if `c` was not found.

Finally, you can use `mempcpy()` to step through memory:

```
#define _GNU_SOURCE
#include <string.h>

void * mempcpy (void *dst, const void *src, size_t n);
```

The `mempcpy()` function performs the same as `memcpy()`, except that it returns a pointer to the next byte after the last byte copied. This is useful if a set of data is to be copied to consecutive memory locations—but it's not so much of an improvement because the return value is merely `dst + n`. This function is GNU-specific.

Searching Bytes

The functions `memchr()` and `memrchr()` locate a given byte in a block of memory:

```
#include <string.h>

void * memchr (const void *s, int c, size_t n);
```

The `memchr()` function scans the `n` bytes of memory pointed at by `s` for the character `c`, which is interpreted as an `unsigned char`:

```
#define _GNU_SOURCE
#include <string.h>

void * memrchr (const void *s, int c, size_t n);
```

The call returns a pointer to the first byte to match `c`, or `NULL` if `c` is not found.

The `memrchr()` function is the same as the `memchr()` function, except that it searches backward from the end of the `n` bytes pointed at by `s` instead of forward from the front. Unlike `memchr()`, `memrchr()` is a GNU extension and not part of the C language.

For more complicated search missions, the awfully named function `memmem()` searches a block of memory for an arbitrary array of bytes:

```
#define _GNU_SOURCE
#include <string.h>

void * memmem (const void *haystack,
               size_t haystacklen,
               const void *needle,
               size_t needlelen);
```

The `memmem()` function returns a pointer to the first occurrence of the subblock `needle`, of length `needlelen` bytes, within the block of memory `haystack`, of length `haystacklen` bytes. If the function does not find `needle` in `haystack`, it returns `NULL`. This function is also a GNU extension.

Frobnicating Bytes

The Linux C library provides an interface for trivially convoluting bytes of data:

```
#define _GNU_SOURCE
#include <string.h>

void * memfrob (void *s, size_t n);
```

A call to `memfrob()` obscures the first `n` bytes of memory starting at `s` by exclusive-ORing (XORing) each byte with the numberf164.430 42. The call returns `s`.

The effect of a call to `memfrob()` can be reversed by calling `memfrob()` again on the same region of memory. Thus, the following snippet is a no-op with respect to `secret`:

```
memfrob (memfrob (secret, len), len);
```

This function is in no way a proper (or even a poor) substitute for encryption; its use is limited to the trivial obfuscation of strings. It is GNU-specific.

Locking Memory

Linux implements *demand paging*, which means that pages are paged in from disk as needed and paged out to disk when no longer needed. This allows the virtual address spaces of processes on the system to have no direct relationship to the total amount of physical memory, as secondary storage can provide the illusion of a nearly infinite supply of physical memory.

This paging occurs transparently, and applications generally need not be concerned with (or even know about) the Linux kernel's paging behavior. There are, however, two situations in which applications may wish to influence the system's paging behavior:

Determinism
: Applications with timing constraints require deterministic behavior. If some memory accesses result in page faults—which incur costly disk I/O operations—applications can overrun their timing needs. By ensuring that the pages it needs are always in physical memory and never paged to disk, an application can guarantee that memory accesses will not result in page faults, providing consistency, determinism, and improved performance.

Security
: If secrets are kept in memory, the secrets can be paged out and stored unencrypted on disk. For example, if a user's private key is normally stored encrypted on disk, an unencrypted copy of the key in memory can end up in the swap file. In a high-security environment, this behavior may be unacceptable. Applications for which this might be a problem can ask that the memory containing the key always remain in physical memory.

Of course, changing the kernel's paging behavior can result in a negative impact on overall system performance. One application's determinism or security may improve, but while its pages are locked into memory, another application's pages will be paged out instead. The kernel, if we trust its algorithms, chooses the optimal page to page out —that is, the page least likely to be used in the future—so when you change its behavior, it has to swap out a suboptimal page.

Locking Part of an Address Space

POSIX 1003.1b-1993 defines two interfaces for "locking" one or more pages into physical memory, ensuring that they are never paged out to disk. The first locks a given interval of addresses:

```
#include <sys/mman.h>

int mlock (const void *addr, size_t len);
```

A call to `mlock()` locks the virtual memory starting at `addr` and extending for `len` bytes into physical memory. On success, the call returns `0`; on failure, the call returns `-1` and sets `errno` as appropriate.

A successful call locks all physical pages that contain `[addr,addr+len)` in memory. For example, if a call specifies only a single byte, the entire page in which that byte resides is locked into memory. The POSIX standard dictates that `addr` should be aligned to a page boundary. Linux does not enforce this requirement, silently rounding `addr` down to the nearest page if needed. Programs requiring portability to other systems, however, should ensure that `addr` sits on a page boundary.

The valid `errno` codes include:

`EINVAL`
: The parameter `len` is negative.

`ENOMEM`
: The caller attempted to lock more pages than the `RLIMIT_MEMLOCK` resource limit allows (see "Locking Limits" on page 328).

`EPERM`
: The `RLIMIT_MEMLOCK` resource limit was `0`, but the process did not possess the `CAP_IPC_LOCK` capability (again, see "Locking Limits" on page 328).

A child process does not inherit the locked status of memory across a `fork()`. Due to the copy-on-write behavior of address spaces in Linux, however, a child process's pages are effectively locked in memory until the child writes to them.

As an example, assume that a program holds a decrypted string in memory. A process can lock the page containing that string with code such as the following:

```
int ret;

/* lock 'secret' in memory */
ret = mlock (secret, strlen (secret));
if (ret)
        perror ("mlock");
```

Locking All of an Address Space

If a process wants to lock its entire address space into physical memory, `mlock()` is a cumbersome interface. For such a purpose—common to real-time applications—POSIX defines a system call that locks an entire address space:

```
#include <sys/mman.h>

int mlockall (int flags);
```

A call to `mlockall()` locks all of the pages in the current process's address space into physical memory. The `flags` parameter, which is a bitwise OR of the following two values, controls the behavior:

`MCL_CURRENT`
: If set, this value instructs `mlockall()` to lock all currently mapped pages—the stack, data segment, mapped files, and so on—into the process's address space.

`MCL_FUTURE`
: If set, this value instructs `mlockall()` to ensure that all pages mapped into the address space in the future are also locked into memory.

Most applications specify a bitwise OR of both values.

On success, the call returns `0`; on failure, it returns `-1` and sets `errno` to one of the following error codes:

`EINVAL`
: The parameter `flags` is negative.

`ENOMEM`
: The caller attempted to lock more pages than the `RLIMIT_MEMLOCK` resource limit allows (see the later section "Locking Limits" on page 328).

`EPERM`
: The `RLIMIT_MEMLOCK` resource limit was `0`, but the process did not possess the `CAP_IPC_LOCK` capability (again, see "Locking Limits" on page 328).

Unlocking Memory

To unlock pages from physical memory, again allowing the kernel to swap the pages out to disk as needed, POSIX standardizes two more interfaces:

```
#include <sys/mman.h>

int munlock (const void *addr, size_t len);
int munlockall (void);
```

The system call `munlock()` unlocks the pages starting at `addr` and extending for `len` bytes. It undoes the effects of `mlock()`. The system call `munlockall()` undoes the effects of `mlockall()`. Both calls return `0` on success, and on error return `-1` and set `errno` to one of the following:

`EINVAL`
: The parameter `len` is invalid (`munlock()` only).

`ENOMEM`
: Some of the specified pages are invalid.

`EPERM`
: The `RLIMIT_MEMLOCK` resource limit was `0`, but the process did not possess the `CAP_IPC_LOCK` capability (see "Locking Limits").

Memory locks do not nest. Therefore, a single `mlock()` or `munlock()` will unlock a locked page, regardless of how many times the page was locked via `mlock()` or `mlockall()`.

Locking Limits

Because locking memory can affect the overall performance of the system—indeed, if too many pages are locked, memory allocations can fail—Linux places limits on how many pages a process may lock.

Processes possessing the `CAP_IPC_LOCK` capability may lock any number of pages into memory. Processes without this capability may lock only `RLIMIT_MEMLOCK` bytes. By default, this resource limit is 32 KB—large enough to lock a secret or two in memory, but not large enough to adversely affect system performance. (Chapter 6 discusses resource limits and how to retrieve and change this value.)

Is a Page in Physical Memory?

For debugging and diagnostic purposes, Linux provides the `mincore()` function, which can be used to determine whether a given range of memory is in physical memory or swapped out to disk:

```
#include <unistd.h>
#include <sys/mman.h>

int mincore (void *start,
             size_t length,
             unsigned char *vec);
```

A call to `mincore()` provides a vector delineating which pages of a mapping are in physical memory at the time of the system call. The call returns the vector via `vec` and describes the pages starting at `start` (which must be page-aligned) and extending for `length` bytes (which need not be page-aligned). Each byte in `vec` corresponds to one page in the range provided, starting with the first byte describing the first page and moving linearly forward. Consequently, `vec` must be at least large enough to contain `(length - 1 + page size) / page size` bytes. The lowest-order bit in each byte is 1 if the page is resident in physical memory and 0 if it is not. The other bits are currently undefined and reserved for future use.

On success, the call returns `0`. On failure, it returns `-1` and sets `errno` to one of the following:

`EAGAIN`
: Insufficient kernel resources are available to carry out the request.

`EFAULT`
: The parameter `vec` points at an invalid address.

`EINVAL`
: The parameter `start` is not aligned to a page boundary.

`ENOMEM`
: `[address,address+1)` contains memory that is not part of a file-based mapping.

Currently, this system call works properly only for file-based mappings created with `MAP_SHARED`. This greatly limits the call's use.

Opportunistic Allocation

Linux employs an *opportunistic allocation* strategy. When a process requests additional memory from the kernel—say, by enlarging its data segment or by creating a new memory mapping—the kernel *commits* to the memory without actually providing any physical storage. Only when the process writes to the newly allocated memory does the kernel *satisfy* the commitment by converting the commitment for memory to a physical allocation of memory. The kernel does this on a page-by-page basis, performing demand paging and copy-on-writes as needed.

This behavior has several advantages. First, lazily allocating memory allows the kernel to defer most of the work until the last possible moment—if indeed it ever has to satisfy

the allocations. Second, because the requests are satisfied page-by-page and on demand, only physical memory in actual use need consume physical storage. Finally, the amount of committed memory can far exceed the amount of physical memory and even swap space available. This last feature is called *overcommitment*.

Overcommitting and OOM

Overcommitting allows systems to run many more, and much larger, applications than they could if every requested page of memory had to be backed by physical storage at the point of allocation instead of the point of use. Without overcommitment, mapping a 2 GB file copy-on-write would require the kernel to set aside 2 GB of storage. With overcommitment, mapping a 2 GB file requires storage only for each page of data to which the process actually writes. Likewise, without overcommitment, every `fork()` would require enough free storage to duplicate the address space, even though the vast majority of pages never undergo copy-on-writes.

What if, however, processes attempt to satisfy more outstanding commitments than the system has physical memory and swap space? In that case, one or more of the satisfactions must fail. Because the kernel has already committed to the memory—the system call requesting the commitment returned success—and a process is attempting to use that committed memory, the kernel's only recourse is to kill a process, freeing up available memory.

When overcommitment results in insufficient memory to satisfy a committed request, we say that an *out of memory* (OOM) condition has occurred. In response to an OOM condition, the kernel employs the *OOM killer* to pick a process "worthy" of termination. For this purpose, the kernel tries to find the least important process that is consuming the most memory.

OOM conditions are rare—hence the huge utility in allowing overcommitment in the first place. To be sure, however, these conditions are unwelcome, and the indeterministic termination of a process by the OOM killer is often unacceptable.

For systems where this is the case, the kernel allows the disabling of overcommitment via the file */proc/sys/vm/overcommit_memory*, and the analogous *sysctl* parameter `vm.overcommit_memory`.

The default value for this parameter, `0`, instructs the kernel to perform a heuristic overcommitment strategy, overcommitting memory within reason, but disallowing egregious overcommitments. A value of `1` allows all commitments to succeed, throwing caution to the wind. Certain memory-intensive applications, such as those in the scientific field, tend to request so much more memory than they ever need satisfied that such an option makes sense.

A value of `2` disables overcommitments altogether and enables *strict accounting*. In this mode, memory commitments are restricted to the size of the swap area plus a configurable percentage of physical memory. The configuration percentage is set via the file */proc/sys/vm/overcommit_ratio* or the analogous *sysctl* parameter, which is `vm.overcommit_ratio`. The default is `50`, which restricts memory commits to the size of the swap area plus half of the physical memory. Because physical memory contains the kernel, page tables, system-reserved pages, locked pages, and so on, only a portion of it is actually swappable and guaranteed to be able to satisfy commitments.

Be careful with strict accounting! Many system designers, repulsed by the notion of the OOM killer, think strict accounting is a panacea. However, applications often perform many unnecessary allocations that reach far into overcommitment territory, and allowing this behavior was one of the main motivations behind virtual memory.