13

Interlude: Memory API

In this interlude, we discuss the memory allocation interfaces in UNIX systems. The interfaces provided are quite simple, and hence the chapter is short and to the point[1].

13.1 Types of Memory

In running a C program, there are two types of memory that are allocated. The first is called **stack** memory, and allocations and deallocations of it are managed *implicitly* by the compiler for you, the programmer; for this reason it is sometimes called **automatic** memory.

Declaring memory on the stack in C is easy. For example, let's say you need some space in a function `func()` for an integer, called `x`. To declare such a piece of memory, you just do something like this:

```
void func() {
    int x; // declares an integer on the stack
    ...
}
```

The compiler does the rest, making sure to make space on the stack when you call into `func()`. When your return from the function, the compiler deallocates the memory for you; thus, if you want

[1] Indeed, we hope all chapters are short and to the point! But this one is shorter and pointier, we think.

some information to live beyond the call invocation, you had better not leave that information on the stack.

It is this need for long-lived memory that gets us to the second type of memory, called **heap** memory, where all allocations and deallocations are *explicitly* handled by you, the programmer. A heavy responsibility, no doubt! And certainly the cause of many bugs. But if you are careful and pay attention, you will use such interfaces correctly and without too much trouble. Here is an example of how one might allocate a pointer to an integer on the heap:

```
void func() {
    int *x = (int *) malloc(sizeof(int));
    ...
}
```

A couple of notes about this small code snippet. First, you might notice that both stack and heap allocation occur on this line: first the compiler knows to make room for a pointer to an integer when it sees `int *x`; subsequently, when the program calls `malloc()`, it requests space for an integer on the heap; the routine returns the address of such an integer (upon success, or `NULL` on failure), which is then stored on the stack for use by the program.

Because of its explicit nature, and because of its more varied usage, heap memory presents more challenges to both users and systems. Thus, it is the focus of the remainder of our discussion.

13.2 The `malloc()` Call

The `malloc()` call is quite simple: you pass it a size asking for some room on the heap, and it either succeeds and gives you back a pointer to the newly-allocated space, or fails and returns `NULL`[2].

The manual page shows what you need to do to use malloc; type `man malloc` at the command line and you will see:

```
#include <stdlib.h>
...
void *malloc(size_t size);
```

[2]Note that `NULL` in C isn't really anything special at all; it is just a macro for the value zero.

From this information, you can see that all you need to do is include the header file `stdlib.h` to use malloc. In fact, you don't really need to even do this, as the C library, which all C programs link with by default, has the code for `malloc()` inside of it; adding the header just lets the compiler check whether you are calling `malloc()` correctly (e.g., passing the right number of arguments to it, of the right type).

The single parameter `malloc()` takes is of type `size_t` which simply describes how many bytes you need. However, most programmers do not type in a number here directly (such as 10); indeed, it would be considered poor form to do so. Instead, various routines and macros are utilized. For example, to allocate space for a double-precision floating point value, you simply do this:

```
double *d = (double *) malloc(sizeof(double));
```

This invocation of `malloc()` uses the `sizeof()` operator to request the right amount of space; in C, this is generally thought of as a *compile-time* operator, meaning that the actual size is known at *compile time* and thus a number (in this case, 8, for a double) is substituted as the argument to `malloc()`. For this reason, `sizeof()` is correctly thought of as an operator and not a function call (a function call would take place at run time).

TIP: WHEN IN DOUBT, TRY IT OUT

If you aren't sure how some routine or operator you are using behaves, there is no substitute for simply trying it out and making sure it behaves as you expect. While reading the manual pages or other documentation is useful, how it works in practice is what matters. Write some code and test it! That is no doubt the best way to make sure your code behaves as you desire. Indeed, that is what we did to double-check the things we were saying about `sizeof()` were actually true!

You can also pass in the name of a variable (and not just a type) to `sizeof()`, but in some cases you may not get the desired results, so be careful. For example, let's look at the following code snippet:

```
int *x = malloc(10 * sizeof(int));
printf("%d\n", sizeof(x));
```

In the first line, we've declared space for an array of 10 integers, which is fine and dandy. However, when we use `sizeof()` in the next line, it returns a small value, such as 4 (on 32-bit machines) or 8 (on 64-bit machines). The reason is that in this case, `sizeof()` thinks we are simply asking how big a *pointer* to an integer is, not how much memory we have dynamically allocated. However, sometimes `sizeof()` does work as you might expect:

```
int x[10];
printf("%d\n", sizeof(x));
```

In this case, there is enough static information for the compiler to know that 40 bytes have been allocated.

Another place to be careful is with strings. When declaring space for a string, use the following idiom: `malloc(strlen(s) + 1)`, which gets the length of the string using the function `strlen()`, and adds 1 to it in order to make room for the end-of-string character. Using `sizeof()` may lead to trouble here.

You might also notice that `malloc()` returns a pointer to type `void`. Doing so is just the way in C to pass back an address and let the programmer decide what to do with it. The programmer further helps out by using what is called a **cast**; in our example above, the programmer casts the return type of `malloc()` to a pointer to a `double`. Casting doesn't really accomplish anything, other than tell the compiler and other programmers who might be reading your code: "yeah, I know what I'm doing." By casting the result of `malloc()`, the programmer is just giving some reassurance; the cast is not needed for the correctness of the program.

13.3 The `free()` Call

As it turns out, allocating memory is the easy part of the equation; knowing when, how, and even if to free memory is the hard part. To free heap memory that is no longer in use, programmers simply call `free()`, as follows:

```
int *x = malloc(10 * sizeof(int));
...
free(x);
```

The routine takes one argument, a pointer that was returned by `malloc()`. Thus, you might notice, the size of the allocated region

is not passed in by the user, and must be tracked by the memory-allocation library itself.

13.4 Common Errors

There are a number of common errors that arise in the use of `malloc()` and `free()`. Here are some we've seen over and over again in teaching the undergraduate operating systems course. All of these examples compile and run with nary a peep from the compiler; while compiling a C program is necessary to build a correct C program, it is far from sufficient, as you will learn.

Correct memory management has been such a problem, in fact, that many newer languages have support for **automatic memory management**. In such languages, while you call something akin to `malloc()` to allocate memory (usually **new** or something similar to allocate a new object), you never have to call something to free space; rather, a **garbage collector** runs and figures out what memory you no longer have references to and frees it for you.

Forgetting To Allocate Memory

Many routines expect memory to be allocated before you call them. For example, the routine `strcpy(dst, src)` copies a string from a source pointer to a destination pointer. However, if you are not careful, you might do this:

```
char *src = "hello";
char *dst;        // oops! unallocated
strcpy(dst, src); // segfault and die
```

When you run this code, it will likely lead to a **segmentation fault**[3], which is a fancy term for **YOU DID SOMETHING WRONG WITH MEMORY YOU FOOLISH PROGRAMMER AND I AM ANGRY.**

In this case, the proper code might instead look like this:

```
char *src = "hello";
char *dst = (char *) malloc(strlen(src) + 1);
strcpy(dst, src); // work properly
```

[3] Although it sounds arcane, you will soon learn why such an illegal memory access is called a segmentation fault; if that isn't incentive to read on, what is?

Alternately, you could use `strdup()` and make your life even easier. Read the `strdup` man page for more information.

Not Allocating Enough Memory

A related error is not allocating enough memory, sometimes called a **buffer overflow**. In the example above, a common error is to make *almost* enough room for the destination buffer.

```
char *src = "hello";
char *dst = (char *) malloc(strlen(src)); // too small!
strcpy(dst, src); // work properly
```

Oddly enough, depending on how malloc is implemented and many other details, this program will often run seemingly correctly. In some cases, when the string copy executes, it writes one byte too far past the end of the allocated space, but in some cases this is harmless, perhaps overwriting a variable that isn't used anymore. In some cases, these overflows can be incredibly harmful, and in fact are the source of many security vulnerabilities in systems [W06]. In other cases, the malloc library allocated a little extra space anyhow, and thus your program actually doesn't scribble on some other variable's value and works quite fine. In even other cases, the program will indeed fault and crash. And thus we learn another valuable lesson: even though it ran correctly once, doesn't mean it's correct.

TIP: IT COMPILED OR IT RAN $\neq$ IT IS CORRECT

Just because a program compiled(!) or even ran once or many times correctly does not mean the program is correct. Many events may have conspired to get you to a point where you believe it works, but then something changes and it stops. A common student reaction is to say (or yell) "But it worked before!" and then blame the compiler, operating system, hardware, or even (dare we say it) the professor. But the problem is usually right where you think it would be, in your code. Get to work and debug it before you blame those other components.

Forgetting to Initialize Allocated Memory

With this error, you call `malloc()` properly, but forget to fill in some values into your newly-allocated data type. Don't do this! If you do forget, your program will eventually encounter an **uninitialized read**, where it reads from the heap some data of unknown value. Who knows what might be in there? If you're lucky, some value such that the program still works (e.g., zero). If you're not lucky, something random and harmful.

Forgetting To Free Memory

Another common error is known as a **memory leak**, and it occurs when you forget to free memory. In long-running applications or systems (such as the OS itself), this is a huge problem, as slowly leaking memory eventually leads one to run out of memory, at which point a restart is required. Thus, in general, when you are done with a chunk of memory, you should make sure to free it. Note that using a garbage-collected language doesn't help here: if you still have a reference to some chunk of memory, no garbage collector will ever free it.

Note that not all memory need be freed, at least, in certain cases. For example, when you write a short-lived program, you might allocate some space using `malloc()`. The program runs and is about to complete: is there need to call `free()` a bunch of times just before exiting? While it seems wrong not to, it is in this case quite fine to simply exit. After all, when your program exits, the OS will clean up everything about this process, including any memory it has allocated. Calling `free()` a bunch of times and then exiting is thus pointless, and, if you do so incorrectly, will cause the program to crash. Just call exit and be happy instead.

Freeing Memory Before You Are Done With It

Sometimes a program will free memory before it is finished using it; such a mistake is called a **dangling pointer**, and it, as you can guess, is also a bad thing. The subsequent use can crash the program, or overwrite valid memory (e.g., you called `free()`, but then called `malloc()` again to allocate something else, which then recycles the errantly-freed memory).

Freeing Memory Repeatedly

Programs also sometimes free memory more than once; this is known as the **double free**. The result of doing so is undefined. As you can imagine, the memory-allocation library might get confused and do all sorts of weird things; crashes are a common outcome.

Calling `free()` Incorrectly

One last problem we discuss is the call of `free()` incorrectly. After all, `free()` expects you only to pass to it one of the pointers you received from `malloc()` earlier. When you pass in some other value, bad things can (and do) happen. Thus, such **invalid frees** are dangerous and of course should also be avoided.

Summary

As you can see, there are lots of ways to abuse memory. Because of frequent errors with memory, a whole ecosphere of tools have developed to help find such problems in your code. Check out both **purify** [HJ92] and **valgrind** [SN05]; both are excellent at helping you locate the source of your memory-related problems.

13.5 Underlying OS Support

You might have noticed that we haven't been talking about system calls when discussing `malloc()` and `free()`. The reason for this is simple: they are not system calls, but rather library calls. Thus the malloc library manages space within your virtual address space, but itself is built on top of some system calls which call into the OS to ask for more memory or release some back to the system.

One such system call is called `brk`, which is used to change the location of the program's **break**: the location of the end of the heap. It takes one argument (the address of the new break), and thus either increases or decreases the size of the heap based on whether the new break is larger or smaller than the current break. An additional call `sbrk` is passed an increment but otherwise serves a similar purpose.

Note that you should never directly call either `brk` or `sbrk`. They are used by the memory-allocation library; if you try to use them,

you will likely make something go (horribly) wrong. Stick to `malloc()` and `free()` instead.

13.6 Other Calls

There are a few other calls that the memory-allocation library supports. For example, `calloc()` allocates memory and also zeroes it before returning; this prevents some errors where you assume that memory is zeroed and forget to initialize it yourself (see the paragraph on "uninitialized reads" above). The routine `realloc()` can also be useful, when you've allocated space for something (say, an array), and then need to add something to it: `realloc()` logically makes a new larger region of memory, copies the old region into it, and returns the pointer to the new region. Read the man pages to find out more.

13.7 Summary

We have introduced some of the APIs dealing with memory allocation. As always, we have just covered the basics; more details are available elsewhere. Read the C book [KR88] and Stevens [S92] (Chapter 7) for more information. For a cool modern paper on how to detect and correct many of these problems automatically, see Novark et al. [N+07]; this paper also contains a nice summary of common problems and some neat ideas on how to find and fix them.

References

[HJ92] Purify: Fast Detection of Memory Leaks and Access Errors
R. Hastings and B. Joyce
USENIX Winter '92
The paper behind the cool Purify tool, now a commercial product.

[KR88] "The C Programming Language"
Brian Kernighan and Dennis Ritchie
Prentice-Hall 1988
The C book, by the developers of C. Read it once, do some programming, then read it again, and then keep it near your desk or wherever you program.

[N+07] "Exterminator: Automatically Correcting Memory Errors with High Probability"
Gene Novark, Emery D. Berger, and Benjamin G. Zorn
PLDI 2007
A cool paper on finding and correcting memory errors automatically, and a great overview of many common errors in C and C++ programs.

[SN05] "Using Valgrind to Detect Undefined Value Errors with Bit-precision"
J. Seward and N. Nethercote
USENIX '05
How to use valgrind to find certain types of errors.

[SR05] "Advanced Programming in the UNIX Environment"
W. Richard Stevens and Stephen A. Rago
Addison-Wesley, 2005
We've said it before, we'll say it again: read this book many times and use it as a reference whenever you are in doubt. The authors are always surprised at how each time they read something in this book, they learn something new, even after many years of C programming.

[W06] "Survey on Buffer Overflow Attacks and Countermeasures"
Tim Werthman
Available: www.nds.rub.de/lehre/seminar/SS06/Werthmann_BufferOverflow.pdf
A nice survey of buffer overflows and some of the security problems they cause. Refers to many of the famous exploits.