【OS】Day17

【Ch14】Memory API

The main problem we address in this chapter is this:

Crux: How to allocate and manage memory ?

In Unix/C programs, understanding how to allocate and manage memory is critical in building robust and reliable software. What interfaces are commandly used? What mistakes should be avoided?

14.1 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 us. 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 you return from the function, the compiler deallocates the memory for you; thus, if you want 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.

Here is an example of how one might allocate an integer on the heap:

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

A couple of notes about this small code snippet:

1. First, 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 your declaration of said pointer `int*`;
2. 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.

14.2 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.`

The manual page shows us how to use `malloc`:

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

The single paramter `malloc()` takes is of type `size_t` which simply describes how many bytes we need.

However, we do not just type in a number here directly(such as 10); it is 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, we simply do this:

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

`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.

14.3 The free() Call

To free heap memory that is no longer in use, programmers imply call `free()`:

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

The routine takes one argument, a pointer returned by `malloc()`.

14.4 Common Errors

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(void *dst, void *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; //UNALLOCATED!!!
strcpy(dst, src); //segmentation fault
```

This error is called a segmentation fault.

Alternatively, we could use `strdup()` (which returns a pointer to a new string which is a copy of the passed-in string) and make our life even easier.

Not Allocating Enough Memory

A relatred error is not allocating enough memory, sometimes called a buffer overflow. In this example, 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.

This type of error is usually called Buffer Overflow!

Forgetting to Initialize Allocated Memory

With this error, we call `malloc()` properly, but forget to fill in some values into our newly-allocated data type.

If we forget, our program will eventually encounter an uninitialized read, where it reads from the heap some data of unknown value.

Forgetting to Free Memory

Another common error is known as a memory leak, and it occurs when we forget to free memory.

In long-running applications, this is a huge problem, as slowly leaking memory eventually leads one to run out of memory, at which point a restart is required.

In some cases, it may seem like not calling `free()` is reasonable. For example, our program is short-lived, and will soon exit; in this case, when the process dies, the OS will clean up all of its allocated pages and thus no memory leak will take place.

However, it is a bad habit to develop.

Freeing Memory Before we 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 is also a bad thing. The subsequent use can crash the program, or overwrite valid 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.

The library might get confused and do all sorts of weird things; crashes are a common outcome.

Calling free() Incorrectly

`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 happen. Thus, such invalid frees are dangerous and of course should also be avoided.