Operating Systems Development - Preface

by Mike, 2009, Updated 2010

This series is intended to demonstrate and teach operating system development from the ground up.

Introduction

Welcome!

This is a series of chapters, tutorials, and articles about computers and operating systems. The series focuses on a new direction in developing an operating system from scratch during this process, while describing architecture, and concepts that are in system-level programming.

The goal of this series is to provide the most comprehensive guide in operating systems and computer systems, while attempting to cover every bit of it (pun intended).

Before moving on with the series, however, I feel that we should cover our chosen languages, and what is required for our readers to know, along with an overview of important concepts in the languages, and how to work with them. We will also cover concepts that are only used in embedded platforms and system-level software.

This series uses C and x86 Assembly Language. It is very important to have a good understanding of both of these languages before moving on. This chapter includes a review of both of these languages.

Overview of C

It is assumed that you already know how to program in C. This is a quick overview of some of the more important parts of the language, and also how they will work for us.

How to use C in kernel land

16 bit and 32 bit C

In the beginning of programming your system you will find out that there is nothing at all to help you. At power on, the system is also operating in 16 bit **real mode** which 32 bit compilers do not support. This is the first important thing: If you want to create a 16 bit real mode OS, you **must** use a 16 bit C compiler. If, however, you decide that you would like to create a 32 bit OS, you **must** use a 32 bit C compiler. 16 bit C code is not compatible with 32 bit C code.

In the series, we will be creating a 32 bit operating system. Because of this, we will be using a 32 bit C compiler.

C and executable formats

A problem with C is that it does not support the ability to output **flat binary programs**. A flat binary program can basically be defined as a program where the **entry point routine** (such as main()) is always at the first byte of the program file. Wait, what? Why would we want this?

This goes back to the good old days of DOS COM programming. DOS COM programs were flat binary - they had no well-defined entry point nor **symbolic names** at all. To execute the program, all that needed to be done was to "jump" to the first byte of the program. Flat binary programs have no special internal format, so there was no standard. Its just a bunch of 1's and 0's. When the PC is powered on, the system BIOS ROM takes control. When it is ready to start an OS, it has no idea how to do it. Because of this, it runs another program - the **Boot Loader** to load an OS. The BIOS does not at all know what internal format this program file is or what it does. Because of this, it treats the Boot Loader as a **flat binary program**. It loads whatever is on the **Boot Sector** of the **Boot Disk** and "jumps" to the first byte of that program file.

Because of this, the first part of the boot loader, also called the **Boot Code** or **Stage 1** cannot be in C. This is because all C compilers output a program file that has a special internal format - they can be library files, object files, or executable files. There is only one language that natively supports this - assembly language.

How to use C in a boot loader

While it is true that the first part of the boot loader must be in assembly language, it is possible to use C in a boot loader. There are different ways of doing this. One way is used in both Windows and our own in-house operating system, Neptune. We combine an assembly stub program and the C program in a single file. The assembly stub program sets up the system and calls our C program. Because both of these programs are combined into a single file, Stage 1 only needs to load a single file - which in turn loads both our stub program and C program.

This is one method - there are others. Most real boot loaders use C, including GRUB, Neptunes boot loader, Microsoft's NTLDR and Boot Manager. Because we are using 32 bit C, there are also ways that will allow us to mix 16 bit code with our 32 bit C code.

Doing this can be fairly complicated and tricky to implement. Because of this, we stick with just using assembly language in the series boot loader. We might cover an advanced tutorial later that can describe methods of using C later on however if the reader demand is great enough.

Calling a C kernel

When the boot loader is ready, it loads and executes our C kernel by calling its entry point routine. Because the C program follows a specific internal format, the boot loader must know how to parse the file and locate the entry point routine to call it. In the series, we cover how to do this a little later. This allows us to use C for the kernel and other libraries that we build.

Pointers

Introduction

Because you are reading this, I assume that you are already good with pointers. In system software, they are used everywhere. Because of this, it is very important to master pointers.

A pointer is simply a variable that holds the address of something. To define a pointer, we use the * operator:

```
char* pointer;
```

Remember that a pointer stores an "address"? We do not set the above pointer to anything, so what "address" does it refer to? The above code is an example of a **wild pointer**. A wild pointer is a pointer that can point to anything. **Remember that C does not initialize anything for you.** Because of this, the above pointer can point to anything. Another variable, address 0, some other piece of data, your own code, *a hardware address*.

The Physical Address Space (PAS)

The Physical Address Space (PAS) defines all of the "Addresses" that you can use. These addresses can refer to anything that is inside of the PAS. This includes physical memory (RAM), hardware devices, or even nothingness. This is very different then in applications programming in a protected mode OS, like Windows, where all "addresses" are memory.

Here is an example. In applications programming, the following would cause a segmentation fault error and crash your program:

```
char* pointer = 0;
*pointer = 0;
```

This creates a pointer and points it to memory address 0, which you do not "own". Because of this, the system does not allow you to write to it.

Now, lets try that same exact code again in our future C kernel ... no crash! Instead of crashing, it overwrites the first byte of the **Interrupt Vector Table (IVT)**.

From this, we can make a few important differences:

- The system will not crash if you use null pointers
- Pointers can point to any "address" in the PAS, which may or may not be memory

If you attempt to read from a memory address that does not exist, you will get garbage (whatever was on the system data bus at that time.) An attempt to write to a memory address that does not exist does nothing. Writing to a non existent memory address and immediately reading it back may or may not give you the same result just "written"...It depends if the data "written" is still on the data bus or not.

Things get more interesting here. ROM devices are mapped into the same PAS. This means that it is possible for you to read or write certain parts of ROM devices using pointers. A good example of a ROM device is the system BIOS. Because ROM devices are read only, writing to a ROM device is the same effect as writing to a non existent memory location. You can read from ROM devices, however.

Other devices may also be mapped into the PAS. This depends on your system configuration. This means reading or writing different parts of the PAS may yeild different types of results.

As you can tell, pointers play a much bigger role in systems programming then they did in the applications programming world. It may be easier to

think of pointers not as a "variable that points to a memory location" but rather a "variable that points to an address in the PAS" as it may or may not be RAM.

Dynamic Memory Allocation

In the application programming world, you would normally call **malloc()** and **free()** or **new** and **delete** to allocate a block of memory from the heap. This is different in the system programming world. To allocate memory, we do this:

```
char* pointer = (char*)0x5000;
```

That is it. Cool, huh? Because we have control over everything, we can just point a pointer to some address in the PAS (would have to be RAM) and say "theres our new buffer of 1024 bytes" or something like that.

The important thing here is that there is no dynamic memory allocation. Dynamic memory allocation in C and C++ are **system services** and require an OS to be running. But, wait! Aren't we developing our own OS? That is the problem :) We will need to write our own memory management services and routines in order to be able to provide a malloc() and free() or new and delete.

Until then, the only way to "allocate" a buffer is to use some unused location in the address space.

Inline Assembly

There are some things that C cannot natively do. We will be needing to use assembly language for system services and talking to hardware devices.

Most compilers provide a keyword that allows inline assembly. For example, Microsoft Visual C++ uses _asm:

```
_asm cli ; disable interrupts
```

We can also have blocks of assembly code:

```
_asm {
	cli
	hlt
}
```

Standard Library and the Run Time Library (RTL)

You can use external libraries - if and only if those routines do not use system services. Anything like printf(), scanf(), memory routines, or, virtually everything but the bare minimum routines can be used. About 90% of it will be needed to be rewritten for your own OS, so it is best to write your own.

The RTL is the set of services and routines that your application program uses at run time. These, by their nature, require an OS to already be running and to support them. Because of this, you will need to develop your own RTL.

The startup RTL code is responsible for calling C++ constructors and destructors. If you are wanting to use C++, you must develop the RTL code to support it. This uses compiler extensions.

In the series, we develop both an RTL that supports C and C++ features as well as a basic standard library as needed.

Fixing Errors

Debugging

Because there is no printf() or any way to use a debugger, what are you going to do if something is not working? The series uses (and explains) how to use the Bochs Debugger, which is a debugger that comes with the Bochs emulator. This can be used to run your OS as well as for aiding in fixing most of the more common errors that you may run into.

The only other way is to develop your own routines that will allow you to output information. At the most this might be able to tell you how far the software gets to before crashing.

Until next time

That is all that there is for this chapter :) In the next chapter, we begin the adventure into the world of operating systems by looking at what they are, as well as some tools that we will be using throughout the series.

Until next time,

~Mike
BrokenThorn Entertainment. Currently developing DoE and the Neptune Operating System

Questions or comments? Feel free to Contact me.

Would you like to contribute and help improve the articles? If so, please let me know!

Home

Chapter 1