1. Setting up a bootstrap OS

# Introduction

Hi and welcome to the first Fling OS video tutorial. In this tutorial we will look at:

1. How to get started with OS development,
2. Setting up OS development tools (which will also be used in future tutorials),
3. Creating a basic operating system,
4. And running the operating system in a virtual machine.

Fling OS is an x86 based OS and so these tutorials will be based around x86 along with VMWare as the virtual machine software. Don’t worry, all the software required is free (at least for personal use).

If you can’t decide whether you want to target x86 or ARM or some other architecture (such as MIPS) then you can do two things:

1. Remember that most PCs are currently based on x86 so targeting x86 is easiest
2. Read the Fling OS article on CPUs and CPU Architecture that will give you a lot more information to inform your decision.

It should be noted that a lot of low-level knowledge is transferrable knowledge, so targeting x86 doesn’t put you at a big disadvantage if you later want to write an OS for ARM.

# Getting Started

First of all congratulations for finding this tutorial. There are many, many scraps of tutorials and information out on the internet about OS development but Fling OS is different. Unlike other sites which have scraps of info or poor documentation, Fling OS is working towards a complete educational OS that’s up to date (including things like USB support). You can be sure that if you follow Fling OS (and give us a bit of time) you will learn all the core stuff you need for OS development.

The question we really need to answer is: how do you go about developing an OS? Well, the first step is to follow tutorials such as this one to get the basics working, even if you don’t fully understand them yet.

Once you have the basics working, you will have some idea of what is involved in developing an OS. At this point you will need to pick something to aim for (for example, USB support). Then design your architecture and code structure to get you to that goal efficiently but without hacking it together. After that it’s a case of filling in all the gaps with code, which generally involves either watching tutorials or reading reference articles.

Fling OS is building a large documentation site full of reference articles for fundamental OS stuff. Fling OS is also itself a working operating system so you can view the code to give you an idea of what to do.

Other useful sites are:

1. OSDev.org – This has a number of articles often written by other hobbyist developers. Though often incomplete, they do provide a lot of useful, practical info that can help you get your task done.
2. And of course Google. Need I say more?
3. When Googling it is often good to spot blog articles as these can be packed full of info from a hobbyist who’s done something themselves.

Other sources of information include specifications (though these are often hard to read and understand, let alone write code from) and other hobby OS projects. When looking at other code though, check the license and try not to directly copy/paste code. Linux can be useful but because it is production code it can be unintelligible or contain specific hacks for certain processors and similar sorts of real-world issues. Lastly, guestimating what to do is often successful and faster than trawling the internet for the exact, right answer.

I will mention a couple of pre-requisites before we go any further:

1. It is assumed that you know C, C++ or C# to an advanced level.
2. It is assumed you have at least some familiarity with assembly code.
3. It is assumed you have worked on large software projects before and so understand the design and planning issues of a large, unpredictable software project. This is because that is precisely what OS development is. Large and highly unpredictable in many areas, especially for a hobbyist.

# Setup

Moving on to set up. There’s a few practical pre-requisites to these tutorials:

1. You should be running on the Windows OS.
   1. This is so the compiler software works. There are alternatives for Linux and Mac (and even Android) but I don’t have time in these tutorials to cover all the variants. All the basic (i.e. non-C#) tutorial code will still work and compile but you will need to find software to do this. Start by looking at OSDev.org
2. You will also require an internet connection.
   1. This is for downloading software and the package for this tutorial.
3. And lastly admin privileges, for installing software.

Now please take a moment to go and download VMWare Player which is free for personal use and also download the Tutorial Package. Then install VMWare Player but don’t touch the tutorial package (yet). After you’ve done all that, come back and continue with this video.

[PAUSE]

Okay so you should now have VMWare Player installed. So now unzip the Tutorial Package into a folder of your choice. This folder will be the root development folder for this tutorial. Once you’ve unzipped the package, come back and continue with this video.

[PAUSE]

Okay so you should now have unzipped the folder. If you go into that folder you will see a number of subfolders and files. I will briefly explain what each of these is and why it is needed.

* The Cygwin folder:
  + This contains the Cygwin command line tools used for linking and compiling. These are prebuilt for Windows.
* The ISO folder:
  + This contains the files built into the final operating system.
  + They have been pre-built and/or configured for you so you don’t need to worry about them for now. We will look at these in more detail in the next tutorial.
* The ISO Gen folder:
  + This contains a wrapper command line program I wrote for Mosa’s ISO Generator library.
  + It lets you produce ISO 9660 format files which are the final output of the compiler.
  + ISO files are disc image files, in other words they look like a CD. VMWare uses them as the disk to boot from. We will look at setting up VMWare at the end of this tutorial.
* The NASM folder:
  + This contains a copy of the NASM compiler which converts the assembler code into machine code. The output is stored in a temporary binary file (the extension is .bin).
  + The Cygwin tools contain a linker which is used to combine the OS machine code with the code in the ISO folder which produces a temporary object file (the extension is .obj).
  + Lastly the ISO Gen tool converts the object file into the final ISO file.
  + The temporary intermediate files (.bin and .obj) are deleted.
* Bootstrap.asm:
  + This is the main file of interest. It contains a working copy of the assembler code for this tutorial and is the code of your OS.
* Bootstrap.iso is the final output disc image file as mentioned previously. This has been precompiled and tested so it is a working version.
* Compile.bat is a windows batch script which executes all the compile tools in the correct order and handles deleting the intermediate files.

So now the setup is complete, we can move on to some actual code!

# Multiboot

To explain Multiboot we must first understand what a bootloader is. A bootloader is the first small bit of code the processor runs when it starts. The bootloader then loads your OS into memory and then hands off execution to your OS. A bootloader may also offer the user the choice to select which OS to start if multiple are installed. In this simple example we have a good bootloader called IsoLinux which is a version of SysLinux and we only have one OS installed.

So to load your OS the bootloader must be able to find it. This is what Multiboot is. Multiboot is a standard for detecting and loading operating systems. Note at this point that you could write your own bootloader that either supported the Multiboot standard or had its own standard. However, bootloaders are fairly boring and difficult to write and many good ones (such as Syslinux and GRUB) already exist that you are best off using.

Multiboot can also provide us with extra information. In this tutorial we will see how it can tell us the maximum amount of memory available as this is the most practically useful piece of information available.

So looking at the assembler code for Multiboot. The Multiboot signature must come at the very start of your assembly code. It consists of a signature followed by flags and setup info. Also included is some static memory for static variables which we will later with a pointer to the Multiboot Info Structure and memory information.

Note also we put all code (and in the case of Fling OS, data) in the .text section of the file. Files can have various sections which allow chunks of memory to be loaded at different locations. For this example (and Fling OS) we have just one section: the .text section which is expected to be there by NASM.

# Kernel Start

First let’s think about what starting the kernel really means. When the bootloader has loaded the OS, it simply starts it executing wherever we tell it to in the configuration. This is what the kernel start point is. It’s the first line of code that actually executes.

So when the kernel starts, we can assume nothing about how it started or what state the PC is in. Nothing is going to have magically set it up for us. So we need to check the OS has been loaded properly, by making sure the Multiboot signature and information is present. Then we need to retrieve the Multiboot information that we want. And finally we switch from Real mode to Protected mode.

I will not discuss in depth the difference between Real and Protected modes. You should be aware, however, that Real mode is not used on x86 anymore (though a fair few articles even on OSDev will talk about using BIOS interrupts from Real mode). Everything you want to do can and should be done from Protected mode. Real and Protected modes are hardware features, with Protected mode being the most recent version and the sensible one to use.

Now let’s look at the kernel start code. The “global Kernel\_Start “ simply makes the Kernel\_Start label visible to the linker so it can configure the bootloader with where to start the OS execution.

“xchg bx, bx” is from when I constructed this code originally for Fling OS from the OSDev.org sample. It does nothing except in Bochs, which is a piece of virtual machine software, it causes the equivalent of a breakpoint being hit. We won’t be using Bochs (we will use VMWare) so this line is possibly unnecessary but it is useful to know about.

“cli” clears all maskable interrupts so things like the user pressing the keyboard can’t accidentally cause our startup code to be interrupted (which would cause a CPU triple fault and system crash until the interrupts table is initialized).

Then we check for the Multiboot signature and info structure. If it isn’t found we will jump to some code for handling “No valid Multiboot”. If it is found we simply continue and load the some of the specific Multiboot info.

And lastly we enable Protected mode by setting the first bit of config register 0. We cannot perform modifier ops on CR0 directly so we must take a copy of its value into EAX, modify that then set the new value of CR0.

# Handling “No Multiboot”

When the Multiboot signature isn’t detected, we must presume our OS hasn’t been loaded correctly. This gives us an issue – what to do next? There really aren’t very many options since you must assume most of your OS isn’t going to work properly. In this example we simply give up, print a message to the user on the screen then enter an infinite sleep-by-halting loop.

Alternative methods are to implement proper shut down code or to try and drop back into the bootloader but both of these are very difficult and ultimately useless for testing.

Looking at the code we can see it starts with a jump. This ensures that regardless of where the code is placed, it will never execute unless we deliberately jump inside of the code to the starting label. The end label simply comes after all the Handle No Multiboot code.

The handle no Multiboot code itself very simply outputs the characters of “No space Multiboot” to the screen using VGA Text-Mode output. A reference article explaining how this works can be found in the Fling OS documentation. This code outputs the text in white on a red background.

The final part of the code is the infinite loop but we will see how this works later as very similar code is used in the Reset code.

# Stack Initialization

What is the stack? Well hopefully you know what the stack is and how it can be used. In x86 the stack starts at a high address and pushing to it subtracts from the address. When your processor first starts it assumes nothing about where you want the stack memory to be. So the OS must allocate memory and then tell the CPU where that memory is. In terms of how much to allocate, it is arbitrary but integer powers of 4 are recommended. Also if you allocate too little you will hit stack overflow exceptions very quickly. Attempt to allocate too much and there won’t be enough physical memory!

In terms of the code we see it’s very simple to allocate stack memory and even easier to tell the CPU where that memory is. We simply put the value into the stack pointer register.

# Main entry point

The main entry point is the first point in our code where we can assume everything is set up and working. At this point you would usually call into your Main method written in C, C++ or C# or whatever language you choose. In our example, we simply output “Multiboot” in white on green to the screen to identify that we have successfully booted. The main entry point also contains a small bit of stack setup. Lastly our main entry point ends with a jump to the Reset code so it enters an infinite sleep-by-halting loop.

Looking at the code then. First we do a call to the main method. This is just to initialize the stack a bit. Our main entry point method then does some stack and base pointer manipulation as per the calling convention used by Fling OS. In future tutorials we will discuss when you need to change the calling convention.

After that it’s very simple. We output the text then jump to the Reset code.

# Reset

The reset code works just by preventing interrupts then halting the CPU. If the CPU ever restarts it just loops round and halts the CPU again. This does not turn off the PC but it does prevent the CPU from doing any processing so processing-wise it’s a close equivalent.

We need to do this otherwise we either have to provide other code for the CPU to run or it will crash because it won’t know what to do. There are alternatives that we mentioned earlier (and which most OS’es provide). These are things like shutdown, restart and dropping back into the bootloader.

Here’s a quick look at the reset code.

# Compiling the OS

Assuming you haven’t modified the code you should just be able to run Compile.bat and the ISO file will be recreated. Compiling itself is quite an involved process and we will discuss it in more detail in the next tutorial when we will need to modify part of the process.

For now here is a brief look at what the different files and tools do. There are alternative tools that do just the same job and will run on platforms such as Linux and Mac but I don’t have time to cover them in this tutorial. You should look OSDev as a starting point for help.

# Booting the OS

Finally, booting the OS. I will quickly show you how to set up a virtual machine in VMWare player and how to start it.

# Complete Code

Lastly, a look at the complete code. In this tutorial we looked at the code in an order that made sense as a train of thought. The complete code file has the code combined in a different order (shown here). Processing starts from Kernel\_Start and continues in the following order. The blue path is the expected execution path. The red path is the alternative execution path for a conditional jump.