How to write a simple operating system

This document shows you how to write and build your first operating system in x86 assembly language. It explains what you need, the fundamentals of the PC boot process and assembly language, and how to take it further. The resulting OS will be very small (fitting into a bootloader) and have very few features, but it's a starting point for you to explore further.

After you have read the guide, see the MikeOS project for a bigger x86 assembly language OS that you can explore to expand your skills.

Requirements

Prior programming experience is essential. If you've done some coding in a high-level language like PHP or Java, that's good, but ideally you'll have some knowledge of a lower-level language like C, especially on the subject of memory and pointers.

For this guide we're using Linux. You can view the Windows-specific build instructions on this page.

Installing Linux is very easy thesedays; grab Ubuntu and install it in VMware or VirtualBox if you don't want to dual-boot. When you're in Ubuntu, get all the tools you need to follow this guide by entering this in a terminal window:

```
sudo apt-get install build-essential qemu nasm
```

This gets you the development toolchain (compiler etc.), QEMU PC emulator and the NASM assembler, which converts assembly language into raw machine code executable files.

PC primer

If you're writing an OS for x86 PCs (the best choice, due to the huge amount of documentation available), you'll need to understand the basics of how a PC starts up. Fortunately, you don't need to dwell on complicated subjects such as graphics drivers and network protocols, as you'll be focusing on the essential parts first.

When a PC is powered-up, it starts executing the **BIOS** (Basic Input/Output System), which is essentially a mini-OS built into the system. It performs a few hardware tests (eg memory checks) and typically spurts out a graphic (eg Dell logo) or diagnostic text to the screen. Then, when it's done, it starts to load your operating system from any media it can find. Many PCs jump to the hard drive and start executing code they find in the **Master Boot Record** (MBR), a 512-byte section at the start of the hard drive;

some try to find executable code on a floppy disk (boot sector) or CD-ROM.

This all depends on the boot order - you can normally specify it in the BIOS options screen. The BIOS loads 512 bytes from the chosen media into its memory, and begins executing it. This is the bootloader, the small program that then loads the main OS kernel or a larger boot program (eg GRUB/LILO for Linux systems). This 512 byte bootloader has two special numbers at the end to tell the OS that it's a boot sector - we'll cover that later.

Note that PCs have an interesting feature for booting. Historically, most PCs had a floppy drive, so the BIOS was configured to boot from that device. Today, however, many PCs don't have a floppy drive - only a CD-ROM - so a hack was developed to cater for this. When you're booting from a CD-ROM, it can **emulate a floppy disk**; the BIOS reads the CD-ROM drive, loads in a chunk of data, and executes it as if it was a floppy disk. This is incredibly useful for us OS developers, as we can make floppy disk

versions of our OS, but still boot it on CD-only machines. (Floppy disks are really easy to work with, whereas CD-ROM filesystems are much more complicated.)

So, to recap, the boot process is:

1. Power on: the PC starts up and begins executing the BIOS code.
2. The BIOS looks for various media such as a floppy disk or hard drive.
3. The BIOS loads a 512 byte boot sector from the specified media and begins executing it.
4. Those 512 bytes then go on to load the OS itself, or a more complex bootloader.

For MikeOS, we have the 512-byte bootloader, which we write to a floppy disk image file (a virtual floppy). We can then inject that floppy image into a CD, for PCs that only have CD-ROM drives. Either way, the BIOS loads it as if it was on a floppy, and starts executing it. We have control of the system!

Assembly language primer

Most modern operating systems are written in C/C++. That's very useful when portability and code-maintainability are crucial, but it adds an extra layer of complexity to the proceedings. For your very first OS, you're better off sticking with assembly language, as used in MikeOS. It's more verbose and non-portable, but you don't have to worry about compilers and linkers. Besides, you need a bit of assembly to kick-start any OS.

Assembly language (or colloquially "asm") is a textual way of representing the instructions that a CPU executes. For instance, an instruction to move some memory in the CPU may be **11001001 01101110** - but that's hardly memorable! So assembly provides mnemonics to substitute for these instructions, such as **mov ax, 30**. They correlate directly with machine-

code CPU instructions, but without the meaningless binary numbers.

Like most programming languages, assembly is a list of instructions followed in order. You can jump around between various places and set up subroutines/functions, but it's much more minimal than C# and friends. You can't just print "Hello world" to the screen - the CPU has no concept of what a screen is! Instead, you work with memory, manipulating chunks of RAM, performing arithmetic on them and putting the results in the right place. Sounds scary? It's a bit alien at first, but it's not hard to grasp.

At the assembly language level, there is no such thing as variables in the high-level language sense. What you do have, however, is a set of **registers**, which are on-CPU memory stores. You can put numbers into these registers and perform calculations on them. In 16-bit mode, these registers can hold numbers between 0 and 65535. Here's a list of the fundamental registers on a typical x86 CPU:

AX, BX, CX, DX	General-purpose registers for storing numbers that you're using. For instance, you may use AX to store the character that has been pressed on the keyboard, while using CX to act as a counter in a loop. (Note: these 16-bit registers can be split into 8-bit registers such as AH/AL, BH/BL etc.)
SI, DI	Source and destination data index registers. These point to places in memory for retrieving and storing data.
SP	The Stack Pointer (explained in a moment).
IP (sometimes CP)	The Instruction/Code Pointer. This contains the location in memory of the instruction being executed. When an instruction has finished, it is incremented and moves on to the next instruction. You can change the contents of this register to move around in your code.

So you can use these registers to store numbers as you work - a bit like variables, but they're much more fixed in size and purpose. There are a few others, notably **segment registers**. Due to limitations in old PCs, memory was handled in 64K chunks called segments. This is a really messy subject, but thankfully you don't have to worry about it - for the time being, your OS will be less than a kilobyte anyway! In MikeOS, we limit ourselves to a single 64K segment so that we don't have to mess around with segment registers.

The **stack** is an area of your main RAM used for storing temporary information. It's called a stack because numbers are stacked one-on-top of another. Imagine a Pringles tube: if you put in a playing card, an iPod Shuffle and a beermat, you'll pull them out in the reverse order (beermat, then iPod, and finally playing card). It's the same with numbers: if you **push** the numbers 5, 7 and 15 onto the stack, you will **pop** them out as 15 first, then 7, and lastly 5. In assembly, you can push registers onto the

stack and pop them out later - it's useful when you want to store temporarily the value of a register while you use that register for something else.

PC **memory** can be viewed as a linear line of pigeon-holes ranging from byte 0 to whatever you have installed (millions of bytes on modern machines). At byte number 53,634,246 in your RAM, for instance, you may have your web browser code to view this document. But whereas we humans count in powers of 10 (10, 100, 1000 etc. - decimal), computers are better off with powers of two (because they're based on binary). So we use**hexadecimal**, which is **base 16**, as a way of representing numbers. See this chart to understand:

Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Hexadecimal	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14

As you can see, whereas our normal decimal system uses 0 - 9, hexadecimal uses 0 - F in

counting. It's a bit weird at first, but you'll get the hang of it. In assembly programming, we identify hexadecimal (hex) numbers by tagging a**'h'** onto the end - so **0Ah** is hex for the number **10**. (You can also denote hexadecimal in assembly by prefixing the number with 0x - for instance, 0x0A.)

Let's finish off with a few common assembly instructions. These move memory around, compare them and perform calculations. They're the building blocks of your OS - there are hundreds of instructions, but you don't have to memorise them all, because the most important handful are used 90% of the time.

mov	Copies memory from one location or register to another. For instance, **mov ax, 30** places the number 30 into the AX register. Using square brackets, you can get the number at the memory location pointed to by the register. For instance, if BX contains 80, then **mov ax, [bx]** means "get the number in memory location 80, and put it into AX". You can move

	numbers between registers too: **mov bx, cx**.
add / sub	Adds a number to a register. **add ax, FFh** adds FF in hexadecimal (255 in our normal decimal) to the AX register. You can use **sub** in the same way: **sub dx, 50**.
cmp	Compares a register with a number. **cmp cx, 12** compares the CX register with the number 12. It then updates a special register on the CPU called **FLAGS** - a special register that contains information about the last operation. In this case, if the number 12 is bigger than the value in CX, it generates a negative result, and notes that negative in the FLAGS register. We can use this in the following instructions...
jmp / jg / jl...	Jump to a different part of the code. **jmp label** jumps (GOTOs) to the part of our source code where we have **label:** written. But there's more - you can jump conditionally, based on the CPU flags set in the previous command. For instance, if a **cmp** instruction determined that a

	register held a smaller value than the one with which it was compared, you can act on that with **jl label**(jump if less-than to label). Similarly, **jge label** jumps to 'label' in the code if the value in the**cmp** was greater-than or equal to its compared number.
int	Interrupt the program and jump to a specified place in memory. Operating systems set up**interrupts** which are analogous to subroutines in high-level languages. For instance, in MS-DOS, the 21h interrupt provides DOS services (eg as opening a file). Typically, you put a value in the AX register, then call an interrupt and wait for a result (passed back in a register too). When you're writing an OS from scratch, you can call the BIOS with **int 10h, int 13h, int 14h** or**int 16h** to perform tasks like printing strings, reading sectors from a floppy disk etc.

Let's look at some of these instructions in a little more detail. Consider the following code snippet:

```
    mov bx, 1000h
    mov ax, [bx]
    cmp ax, 50
    jge label
    ...

label:
    mov ax, 10
```

In the first instruction, we move the number 1000h into the BX register. Then, in the second instruction, we store in AX whatever is in the memory location pointed to by BX. This is what the **[bx]** means: if we just did **mov ax, bx** it'd simply copy the number 1000h into the AX register. But by using square brackets, we're saying: don't just copy the contents of BX into AX, but copy the contents of the memory address to which BX points. Given that BX contains 1000h, this instruction says: find whatever is at memory location 1000h, and put it into AX.

So, if the byte of memory at location 1000h contains 37, then that number 37 will be put into the AX register via our second instruction. Next

up, we use **cmp** to compare the number in AX with the number 50 (the decimal number 50 - we didn't suffix it with 'h'). The following **jge** instruction acts on the **cmp** comparison, which has set the FLAGS register as described earlier. The **jge label** says: if the result from the previous comparison is greater than or equal, jump to the part of the code denoted by **label:**. So if the number in AX is greater than or equal to 50, execution jumps to label:. If not, execution continues at the '...' stage.

One last thing: you can insert data into a program with the **db** (define byte) directive. For instance, this defines a series of bytes with the number zero at the end, representing a string:

```
mylabel: db 'Message here', 0
```

In our assembly code, we know that a string of characters, terminated by a zero, can be found at the **mylabel:**position. We could also set up single byte to use somewhat like a variable:

```
foo: db 0
```

Now **foo:** points at a single byte in the code, which in the case of MikeOS will be writable as the OS is copied completely to RAM. So you could have this instruction:

```
    mov byte al, [foo]
```

This moves the byte pointed to by **foo** into the AL register.

That's the essentials of x86 PC assembly language, and enough to get you started. When writing an OS, though, you'll need to learn much more as you progress, so see the Resources section for links to more in-depth assembly tutorials.

Your first OS

Now you're ready to write your first operating system kernel! Of course, this is going to be extremely bare-bones, just a 512-byte boot sector as described earlier, but it's a starting

point for you to expand further. Paste the following code into a file called **myfirst.asm** and save it into your home directory - this is the source code to your first OS.

```
    BITS 16

start:
    mov ax, 07C0h       ; Set up 4K stack
space after this bootloader
    add ax, 288     ; (4096 + 512) / 16
bytes per paragraph
    mov ss, ax
    mov sp, 4096

    mov ax, 07C0h       ; Set data segment
to where we're loaded
    mov ds, ax



    mov si, text_string; Put string
position into SI
    call print_string  ; Call our string-
printing routine

    jmp $         ; Jump here - infinite
loop!
```

```
    text_string db 'This is my cool new
OS!', 0


print_string:           ; Routine: output
string in SI to screen
    mov ah, 0Eh     ; int 10h 'print char'
function

.repeat:
    lodsb           ; Get character from
string
    cmp al, 0
    je .done        ; If char is zero, end of
string
    int 10h           ; Otherwise, print it
    jmp .repeat

.done:
    ret


    times 510-($-$$) db 0 ; Pad remainder
of boot sector with 0s
    dw 0xAA55     ; The standard PC boot
signature
```

Let's step through this. The **BITS 16** line isn't an x86 CPU instruction; it just tells the NASM assembler that we're working in 16-bit mode. NASM can then translate the following instructions into raw x86 binary. Then we have the**start:** label, which isn't strictly needed as execution begins right at the start of the file anyway, but it's a good marker. From here onwards, note that the semicolon (;) character is used to denote non-executable text comments - we can put anything there.

The following six lines of code aren't really of interest to us - they simply set up the segment registers so that the stack pointer (SP) knows where our handy stack of temporary data is, and where the data segment (DS) is located. As mentioned, segments are a hideously messy way of handling memory from the old 16-bit days, but we just set up the segment registers and forget about them. (The references to 07C0h are the equivalent segment location at which the BIOS loads our code, so we start from there.)

The next part is where the fun happens. The **mov si, text_string** line says: copy the location of the text string below into the SI register. Simple enough! Then we use **call**, which is like a GOSUB in BASIC or a function call in C. It means: jump to the specified section of code, but prepare to come back here when we're done.

How does the code know how to do that? Well, when we use a **call** instruction, the CPU increments the position of the IP (Instruction Pointer) register and **pushes** it onto the stack. You may recall from the previous explanation of the stack that it's a last-in first-out memory storage mechanism. All that business with the stack pointer (SP) and stack segment (SS) at the start cleared a space for the stack, so that we can drop temporary numbers there without overwriting our code.

So, the **call print_string** says: jump to the print_string routine, but push the location of the next instruction onto the stack, so we can **pop** it off later and resume execution here. Execution

has jumped over to **print_string:** - this routine uses the BIOS to output text to the screen. First we put 0Eh into the AH register (the upper byte of AX). Then we have a **lodsb** (load string byte) instruction, which retrieves a byte of data from the location pointed to by SI, and stores it in AL (the lower byte of AX). Next we use **cmp** to check if that byte is zero - if so, it's the end of the string and we quit printing (jump to the **.done** label).

If it's not zero, we call **int 10h** (interrupt our code and go to the BIOS), which reads the value in the AH register (0Eh) we set up before. Ah, says the BIOS - 0Eh in the AH register means "print the character in the AL register to the screen!". So the BIOS prints the first character in our string, and returns from the **int** call. We then jump to the**.repeat** label, which starts the process again - **lodsb** to load the next byte from SI (it increments SI each time), see if it's zero and decide what to do. The **ret** at the end of our string-printing routine means: "we've finished here - return back to the place where we

were **call**ed by popping the code location from the stack back into the IP register".

So there we have a demonstration of a loop, in a standalone routine. You can see that the **text_string** label is alongside a stream of characters, which we insert into our OS using **db**. The text is in apostrophes so that NASM knows it's not code, and at the end we have a zero to tell our **print_string** routine that we're at the end.

Let's recap: we start off by setting up the segment registers so that our OS knows where the stack pointer and executable code resides. Then we point the SI register at a string in our OS binary, and **call** our string-printing routine. This routine scans through the characters pointed to by SI and displays them until it finds a zero, at which point it **ret**urns back into the code that **call**ed it. Then the **jmp $** line says: keep jumping to the same line. (The '$' in NASM denotes the current point of code.) This sets up an infinite loop, so that the message is displayed

and our OS doesn't try to execute the following string!

The final two lines are interesting. For a PC to recognise a valid floppy disk boot sector, it has to be exactly 512 bytes in size and end with the numbers AAh and 55h (the boot signature). So the first of these lines says: pad out our resulting binary file to be 510 bytes in size. Then the second line uses **dw** (define a word - two bytes) containing the aforementioned boot signature. Voila: a 512 byte boot file with the correct numbers at the end for the BIOS to recognise.

Building

Let's build our new OS. As mentioned, these instructions are for Linux; if you're using Windows, see the separate guide. In a terminal window, in your home directory, enter:

```
nasm -f bin -o myfirst.bin myfirst.asm
```

Here we assemble the code from our text file into a raw binary file of machine-code

instructions. With the **-f bin**flag, we tell NASM that we want a plain binary file (not a complicated Linux executable - we want it as plain as possible!). The **-o myfirst.bin** part tells NASM to generate the resulting binary in a file called myfirst.bin.

Now we need a virtual floppy disk image to which we can write our bootloader-sized kernel. Copy **mikeos.flp** from the **disk_images/** directory of the MikeOS bundle into your home directory, and rename it **myfirst.flp**. Then enter:

```
dd status=noxfer conv=notrunc
if=myfirst.bin of=myfirst.flp
```

This uses the 'dd' utility to directly copy our kernel to the first sector of the floppy disk image. When it's done, we can boot our new OS using the QEMU PC emulator as follows:

```
qemu -fda myfirst.flp
```

And there you are! Your OS will boot up in a virtual PC. If you want to use it on a real PC, you can write the floppy disk image to a real floppy

and boot from it, or generate a CD-ROM ISO image. For the latter, make a new directory called **cdiso** and move the **myfirst.flp** file into it. Then, in your home directory, enter:

```
mkisofs -o myfirst.iso -b myfirst.flp cdiso/
```

This generates a CD-ROM ISO image called **myfirst.iso** with bootable floppy disk emulation, using the virtual floppy disk image from before. Now you can burn that ISO to a CD-R and boot your PC from it! (Note that you need to burn it as a direct ISO image and not just copy it onto a disc.)

Next you'll want to improve your OS - explore the MikeOS source code to get some inspiration. Remember that bootloaders are limited to 512 bytes, so if you want to do a lot more, you'll need to make your bootloader load a separate file from the disk and begin executing it, in the same fashion as MikeOS.

Going further

So, you've now got a very simple bootloader-based operating system running. What next? Here are some ideas:

- **Add more routines** -- You already have *print_string* in your kernel. You could add routines to get strings, move the cursor etc. Search the internet for BIOS calls which you can use to achieve these.
- **Load files** -- The bootloader is limited to 512 bytes, so you don't have much room. You could make the bootloader load subsequent sectors on the disk into RAM, and jump to that point to continue execution. Or you could read up on FAT12, the filesystem used on floppy drives, and implement that. (See source/bootload/bootload.asm in the MikeOS .zip for an implementation.)
- **Join another project** -- This tutorial was written by Mike Saunders, the lead developer of MikeOS. It's a simple 16-bit x86 assembly

language operating system, and once you're fully confident with the concepts covered here, you'll be able to dig into the code and add features. See the System Developer Handbook link on the front page for more info, and join the mailing list!