O.S. Project 1: Booting and System Calls

Part I. Introduction

The term project for this class is a series of programs that result in a tiny yet functional multitasking operating system. The operating system is to be programmed primarily in C with a small amount of assembly functions. It contains no externally written functions or libraries; all the code is written exclusively for this project.

The completed operating system contains the following components:

- A boot loader
- System calls using software interrupts
- A CP/M-like file system, with system calls for creating, reading, and deleting files
- A program loader that can load and execute user programs
- A command-line shell with basic shell commands: dir, copy, delete, type, execute process

Motivation

The planned operating system is similar to CP/M and designed to fit on a 3½ inch floppy disk and to be bootable on any x86 PC. You should be asking yourself what you will gain from writing a museum piece. First of all, it should be clear that modern operating systems have become so big that to construct one in the space of one semester would be virtually impossible. All we could hope to do is study parts, and it makes sense to start with the basics.

More importantly, desktop computers are being rapidly supplanted by smaller devices with limited power consumption and resources. The operating systems for these devices have a lot in common with their text-driven counterparts for old PCs, such as restricted I/O paradigms, reduced resource options and limited functionality/responsibility. The starting point for a hand-held o.s. isn't that much different from where we begin this excursion, making this good practice for the types of devices you will use in the years to come.

Source Files

After compiling the operating system, you will create a floppy disk image file. An image is a byte-by-byte replica of the data stored on a floppy disk. You can then load the disk image onto a computer simulator.

Your final operating system will include the following provided source files:

- *bootload.asm*
  Assembly code to load the kernel from the disk and run it.

- *kernel.asm*
  Assembly functions called by the kernel. Provided to you since a few routines, such as calling interrupts and writing to registers, must be done in assembly.

- *lib.asm*
  This contains a single assembly function allowing the shell to make system calls.

- *map.img*
  This file contains an initial Disk Map sector image for the file system. You will use this when putting together your floppy disk image.

- *dir.img*
  This file contains an initial Directory sector image for the file system. You will also use this when making your floppy disk image.

- *loadFile.c*
  This is a Unix program that copies a file from Unix onto your disk image.

The files you will have to write yourself include:

- *kernel.c*
  The kernel code. The system calls, file handling, program loading, and process scheduling is done here.

- *shell.c*
  The shell code. This program prompts the user for shell commands and performs them by making system calls.

- *compileOS.sh*
  This is a Unix shell script you will write that will allow you to compile the operating system source files.

Tools

You will need the following utilities to complete this project:

- Bochs — An x86 processor simulator
- bcc (Bruce Evans' C Compiler) — A 16-bit C compiler
- as86, ld86 — A 16-bit assembler and linker that comes with bcc
- gcc — The standard 32-bit GNU C compiler
- nasm — The Netwide Assembler
- dd — A standard low-level copying utility
- pico, nano, or vi — A simple text-based editor

All of these utilities except the first two are available for Linux but not for Windows.

To complete this assignment, you will need a Linux account. The department lab machines have all of the above utilities already installed. You will need to obtain an account on the department server (talk to your professor). Once you have an account, you should use an SSH utility (such as PuTTY) to connect to the server and log in to your account. You should be able to complete the project using your account for storage.

Bochs

Bochs is a simulator of an x86 computer. It allows you to simulate an operating system without potentially wrecking a real computer in the process. It is sufficiently elaborate to run both Linux and Windows 95!

To run Bochs you will need a Bochs configuration file. The configuration file tells things like what drives, peripherals, video, and memory the simulated computer has. The second thing you will need is a disk image. A disk image is a single file containing every single byte stored on a simulated floppy disk. In this project you are writing an operating system that will run off of a 3½ inch floppy disk. To get Bochs to recognize the disk, you should name the file **floppya.img** and store it in the same directory as the configuration file.

Running Bochs from the CAS 241/254 Labs

The recommended way to complete the work on this project is to reboot one of the PCs in our public labs to Linux and work from there. From the log-in screen, type <Ctrl><Alt><Delete>, then select *Restart* from the red menu bar in the lower-right-hand corner. After the machine reboots, use the down-arrow key to select the first "Ubuntu" option and hit <Enter>. After Linux starts, log in with your UANet i.d. and password as you did on the Windows side. From the Linux Start menu (left-click in the lower-left-hand corner) select "Applications", “Utilities” then "Xterm". You now have the UNIX-like command line environment to work in. Navigate to your working directory for this project.

To test Bochs, type **firefox &** at the prompt in your Linux window. Proceed to the lab web site

http://www.cs.uakron.edu/~toneil/teaching/cs426/labs/

and download **test.img** and **osxterm.txt** to your working directory by clicking on each file and selecting the “Save file as” option. Quit firefox and type **mv test.img floppya.img** to rename the file. Type **bochs –f osxterm.txt** at the command prompt to start bochs running. If you see the message “Bochs works!” appear in the Bochs window, you are ready to proceed. From the top command bar select **Simulate** then **Stop**. Click **OK** and close the Bochs window.

When completely done, left-click again in the lower-left-hand corner of the desktop and select “Leave” then “Restart” to quit. This should restart the lab machine to Windows.

(There are other possibilities but you’re on your own. I will not do tech support on your home-made Bochs implementations.)

The Boot Loader

The first thing that the computer does after powering on is read the boot loader from the first sector of the floppy disk into memory and start it running. A floppy disk is divided into sectors, where each sector is 512 bytes. All reading and writing to the disk must be in whole sectors - it is impossible to read or write a single byte. The boot loader is required to fit into Sector 0 of the disk, be exactly 512 bytes in size, and end with the special hexadecimal code "55 AA." Since there is not much that can be done with a 510 byte program, the whole purpose of the boot loader is to load the larger operating system from the disk to memory and start it running.

Since boot loaders have to be very small and handle such operations as setting up registers, it does not make sense to write it in any language other than assembly. Consequently you are not required to write a boot loader in this project - one is supplied to you online (**bootload.asm**). You will need to assemble it, however, and start it running.

If you look at **bootload.asm**, you will notice that it is a very small program that does three things. First it sets up the segment registers and the stack to memory 10000 hex. This is where it puts the kernel in memory. Second, it reads 10 sectors (5120 bytes) from the disk starting at sector 3 and puts them at 10000 hex. This would be fairly complicated if it had to talk to the disk driver directly, but fortunately the BIOS already has a disk read function prewritten. This disk read function is accessed by putting the various parameters into various registers, and calling Interrupt 13 (hex). After the interrupt, the program at sectors 3-12 is now in memory at 10000. The last thing that the boot loader does is jump to 10000, starting whatever program it just placed there. That program should be the one that you are going to write. Notice that after the jump it fills out the remaining bytes with 0, and then sets the last two bytes to **55 AA**, telling the computer that this is a valid boot loader.

To install the boot loader, you first have to assemble it. The boot loader is written in x86 assembly language understandable by the NASM assembler. To assemble it, type **nasm bootload.asm**. The output file **bootload** is the actual machine language file understandable by the computer.

Next you should make an image file of a floppy disk that is filled with zeros. You can do this using the **dd** utility. Type

```
dd if=/dev/zero of=floppya.img bs=512 count=2880
```

This will copy 2880 blocks of 512 bytes each from /dev/zero and put it in file **floppya.img**. 2880 is the number of sectors on a 3½ inch floppy, and /dev/zero is a phony file containing only zeros. What you will end up with is a 1.47 megabyte file **floppya.img** filled with zeros.

Finally you should copy **bootload** to the beginning of **floppya.img**. Type

```
dd if=bootload of=floppya.img bs=512 count=1 conv=notrunc
```

Now type **hexdump –C floppya.img**. You should see the following:

```
00000000  b8 00 10 8e d8 8e d0 8e  c0 b8 f0 ff 89 c4 89 c5  |................|
00000010  b1 04 b6 00 b5 00 b4 02  b0 0a b2 00 bb 00 00 cd  |................|
00000020  13 ea 00 00 00 10 00 00  00 00 00 00 00 00 00 00  |................|
00000030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
000001f0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 55 aa  |..............U.|
00000200  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00168000
```

See that the contents of **bootload** are at the beginning of the **floppya.img** file.

If you want, you can try running **floppya.img** with Bochs. Nothing meaningful will happen, however, because the boot loader just loads and runs garbage. You need to write your program now and put it at sector 3 of **floppya.img**.

Scripts Part I

Notice that producing even an empty **floppya.img** file requires you to type several lines that are very tedious to type over and over. An easier alternative is to make a Linux *shell script* file. A shell script is simply a sequence of commands that can be executed by typing one name.

To begin a script, put all commands that appear above in grey boxes into a single text file, with one command per line, and call it **compileOS.sh**. We will continue to build this up as we move forward.

Part II. Interrupt-Driven I/O

When a computer is turned on, it goes through a process known as *booting*. The computer starts executing the BIOS (Basic Input/Output System, which comes with the computer and is stored in read-only memory (ROM)). The BIOS loads and executes a very small program called the *boot loader*, which is located at the beginning of the disk. The boot loader then loads and executes the *kernel*, a larger program that comprises the bulk of the operating system.

One of the major services an operating system provides are system calls. In this project you will learn how to use some of the system calls provided by the BIOS. You will then write your own system calls to print a string to the video and read in a line from the keyboard. Once these are in place, you will write a very small kernel that will print out “Hello World” to the screen and hang up. This will create the foundation needed for the next project.

Introduction to the Kernel

When writing C programs for your operating system, you should note that all the C library functions, such as **printf**, **scanf**, **putchar**... are unavailable to you. This is because these functions make use of services

provided by Linux. Since Linux will not be running when your operating system is running, these functions will not work (or even compile). You can only use the basic C commands.

Download the additional files **kernel.c** and **lib.asm** (kernel.asm version 1) from the lab web page. Rename **lib.asm** as **kernel.asm** and open **kernel.c** in a text editor. A shortened version of this initial kernel seed file appears at right.

As you can see, stopping running is the simple part right now. After performing I/O, you don't want anything else to run. The simplest way to tie up the computer is to put your program into an infinite while loop and stop the simulator by hand. We now have to figure out what to do with the I/O.

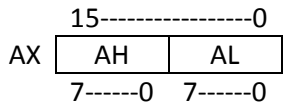
```
void printString(char*);
void readString(char*);

void main()
{
  char line[80];
  /* BlackDOS header */
  printString("Hello world\r\n\0");
  printString("Enter a line: \0");
  readString(line);
  printString("\r\nYou typed: \0");
  printString(line);
  printString("\r\n\0");
  while (1) ;
}

/* more code follows */
```

Overview: x86 Registers

Registers are named storage locations directly inside the CPU, designed to be accessed at much higher speed than RAM. In the Intel x86 architecture the *general-purpose* or *data registers* AX, BX, CX and DX are used for arithmetic and data movement. Each of these is a 16-bit register but can be addressed as their upper or lower 8-bit values. For example AX is a 16-bit register whose upper 8 bits are called AH and whose lower 8 bits are called AL.



Thus instructions can address either the 4 16-bit registers or the 8 8-bit quantities. Changes to the 16-bit registers also modify the corresponding 8-bit registers and vice versa.

Step 1: Printing to the Screen – Interrupt 16 (0x10)/Function 14 (0x0E)

The BIOS provides a software interrupt that will take care of printing to the screen for you. Interrupt 16 calls the BIOS to perform a variety of I/O functions. The one of current interest: if you call interrupt 16 with 14 (0xE) in the AH register, the ASCII character stored in the AL register is printed to the screen at the current cursor location.

Since interrupts may only be called in assembly language, you are provided with a function *interrupt* that makes an interrupt happen. The interrupt function takes five parameters: the interrupt number, and the interrupt parameters passed in the AX, BX, CX, and DX registers, respectively. It returns the value returned from the interrupt in the AL register.

To use interrupt 10 to print out the letter 'Q', you will need to do the following:

```
char al = 'Q';
char ah = 14;
int ax = ah * 256 + al;
interrupt(16, ax, 0, 0, 0);
```

- Figure out the parameters.
  a. To print out 'Q', AH must equal 14 and AL must equal 'Q' (81 decimal or 0x51)
  b. Calculate the value of AX. AX is always AH*256 + AL.
- Call the interrupt routine. Since registers BX, CX, and DX are not used, pass 0 for those parameters.

Alternately you could simply write: **interrupt(16, 14*256+'Q', 0, 0, 0);**

Your Task

To complete step 1, you need to add a *void printString(char*)* function to the kernel. Your *printString* takes a C-string (i.e. character array) as a parameter. The last character in the array should be the unprintable character '\0' (0x0). Your function should print out each character of the array until it reaches 0x0, at which point it should stop.

Notes:

- When adding functions to your C program, make sure they always follow *main()*. *main()* must always be your first function.
- When adding a function, you will need to declare it at the top. Don't forget prototypes.

Step 2: Reading from the keyboard – Interrupt 22 (0x16)

The BIOS interrupt for reading a character from the keyboard is 22 (0x16). When called, AH must equal 0 (actually, it is okay if AX equals 0 since AL does not matter). The interrupt returns the ASCII code for the key pressed; i.e. **char* c = interrupt(22,0,0,0,0)** stores the key stroke in the variable *c*.

You will now write a kernel function *readString*. *readString* should take a character array with at least 80 elements but nothing in them. *readString* should call interrupt 22 repeatedly and save the results in successive elements of the character array until the ENTER key is pressed (ASCII 13 or 0xD). It should then add 0x0 (end of string) as the last character in the array and return.

Two other points:

- All characters typed should be printed to the screen (otherwise the user will not see what the user is typing). After reading a character, the character should be printed to the screen using interrupt 16.
- Your function should be able to handle the BACKSPACE key. When a backspace (ASCII 0x8) is pressed, it should print the backspace to the screen but not store it in the array. Instead it should decrease the array index. (Make sure the array index does not go below zero).

If your function works, when you run the kernel's *main()* function in Bochs, it should prompt you to enter a line. When you press ENTER, it should echo what you typed back to you on the next line.

Compiling the Kernel

To compile your C program you cannot use the standard Linux C compiler **gcc**. This is because **gcc** generates 32-bit machine code, while the computer on start up runs only 16-bit machine code (most real operating systems force the processor to make a transition from 16-bit mode to 32-bit mode, but we are not going to do this). **bcc** is a 16-bit C compiler. It is fairly primitive and requires you to use the early Kernighan and Ritchie C syntax rather than later dialects. For example, you have to use /* */ for all comments; // is not recognized. You should not expect programs that compile with **gcc** to necessarily compile with **bcc**.

To compile your kernel, type

```
bcc -ansi -c -o kernel.o kernel.c
```

The **-c** flag tells the compiler not to use any preexisting C libraries. The **-o** flag tells it to produce an output file called **kernel.o**.

kernel.o is not your final machine code file, however. It needs to be linked with **kernel.asm** so that the final file contains both your code and the **interrupt()** assembly function. You will need to type two more lines:

```
as86 kernel.asm -o kernel_asm.o
ld86 -o kernel -d kernel.o kernel_asm.o
```

The first line assembles **kernel.asm**, the last links all files together to produce the file **kernel**, your program in machine code. To run it, you will need to copy it to **floppya.img** at sector 3, where the boot loader is expecting to load it (in later projects you will find out why sector 3 and not sector 1). To copy it, type

```
dd if=kernel of=floppya.img bs=512 conv=notrunc seek=3
```

where **seek=3** tells it to copy kernel to the third sector.

Try running Bochs. If your program is correct, you should see “Hello World” printed out.

Scripts Part II

As in Lab 1, append all commands that appear above in grey boxes onto the end of your **compileOS.sh** text file. Then type **chmod +x compileOS.sh**, which tells Linux that **compileOS.sh** is an executable. Now, when you change **kernel.c** and want to recompile, simple type **./compileOS.sh**.

Step 3. Clear screen/set colors – Interrupt 16 (0x10)/Functions 2 and 6

It’s time to clean up a bit of the clutter and have some fun. Write *clearScreen*(*bx*,*cx*) to do the following:

- Issue 24 carriage return/newline combinations;
- Issue the command **interrupt(16,512,0,0,0)**;
- Finally if both *bx* and *cx* are bigger than zero execute

interrupt(16, 1536, 4096 * (bx – 1) + 256 * (cx – 1), 0, 6223).

Now for some explanation. The first command calls BIOS function 16/2 which repositions the cursor in the upper left-hand corner (coordinates (0,0)) after the existing text has been cleared away. The other interrupt calls BIOS function 16/6 to scroll the window with these parameters:

- AH = 6, indicating function 6;
- AL = 0, meaning scroll whole screen or window;
- **BH = the attribute byte for blank lines**, explained next;
- CH and CL are the row and column for the upper left-hand corner of the window (0,0); and
- DH and DL are the row and column for the lower right-hand corner, (24,79).

The attribute byte describes the color and intensity of subsequent characters (in its four low bits), as well as the background color (in its four high bits). In the *clearScreen()* function call we specify the background color in the parameter *bx*, the character/foreground color in *cx*. Zero values will mean make no change, otherwise specify a color scheme according to the standard 16-color text palette:

Basic colors (Foreground or background)				**Bright colors (Foreground only)**			
0	Black	4	Red	8	Gray/dark gray	12	Light red
1	Blue	5	Magenta	9	Light blue	13	Light magenta
2	Green	6	Brown	10	Light green	14	Yellow
3	Cyan	7	White/light gray	11	Light cyan	15	Bright white

If *bx* is larger than 8, or *cx* larger than 16, make no change to the color scheme. (Remember that the values passed in are one larger than those in the table.) Finally change the parameters in the initial call to *clearScreen()* in *main()* to create a unique color scheme for your specific version of BlackDOS.

Part III. BlackDOS Function Calls

Historically MS-DOS provided many easy-to-use function calls for providing both console/keyboard and disk I/O, all supported by interrupt 33 (0x21). One advantage was that then any user program (not just the o.s. kernel) could access this functionality. In this lab we will add a couple of extra I/O services to our o.s., enable interrupt 33 and then tie everything done to date to this interrupt.

Integer I/O

So that we can ultimately create some more interesting test programs, let's add I/O for 16-bit unsigned integers to our system. We will need functions **void readInt(int*)** and **void writeInt(int)**.

Of the two **writeInt()** is actually much less obvious so it is provided to you, along with a couple of helper functions. (Refer to the code excerpt at right.) First of all, since we are dealing exclusively with 16-bit unsigned integers, there are at most five digits (MAX_INT = 65535) to print with no sign. (We really should worry about overflow and underflow but won't for now.) Second bcc does not support integer division and modulus as primitive functions, so we have to provide them ourselves.

```
int mod(int a, int b) {
  int x = a;
  while (x >= b) x = x - b;
  return x;
}

int div(int a, int b) {
  int q = 0;
  while (q * b <= a) q++;
  return (q - 1);
}

void writeInt(int x) {
  char number[6], *d;
  int q = x, r;
  d = number + 5;
  *d = 0; d--;
  while (q > 0) {
    r = mod(q,10); q = div(q,10);
    *d = r + 48; d--;
  }
  d++;
  printString(d);
}
```

We now proceed as shown. Work from right-to-left in a 6-character array. Insert a '\0' as your final character, leaving the five digits to be determined. Through the use of div and mod, we can now get the digits and insert them as ASCII characters in our array in reverse order. For example consider the steps in handling 1234.

q=	1234	r=	4	q=	123	Insert '4'
	123		3		12	Insert '3'
	12		2		1	Insert '2'
	1		1		0 (done)	Insert '1'

At the end we advance d to point to the last inserted digit and print the null-terminated string beginning there to the console.

Based on the above discussion **readInt()** should now be straight-forward. Read a number as a character string. From left to right convert the ASCII characters for each individual digit to the corresponding numeric value, correctly weight each value and add them into a running sum. Store the sum at the *address* provided as an argument (must use pass-by-reference here).

When finished rewrite **kernel.c** to match the code at right. Compile the o.s. and test.

```
void printString(char*);
void readString(char*);
void writeInt(int);
void readInt(int*);

void main()
{
  char line[80];
  int x;

  /* BlackDOS header */
  printString("Enter a line: \0");
  readString(line);
  printString("\r\nYou typed: \0");
  printString(line);
  printString("\r\n\0");
  printString("Enter a number: \0");
  readInt(&x);
  printString("You entered \0");
  writeInt(x);
  printString("\r\n\0");
  while (1) ;
}

/* more code follows */
```

Creating Interrupt 33

Creating an interrupt service routine is simply a matter of creating a function, and putting the address of that function in the correct entry of the interrupt vector table. The interrupt vector table sits at the absolute bottom of memory and contains a 4 byte address for each interrupt number. To add a service routine for interrupt 33, write a function to be called on interrupt 33, and then put the address of that function at the right memory address.

Unfortunately, this really has to be done in assembly code. Rename **kernel.asm** as **lib.asm,** then download the new **kernel.asm** file (version 2) from the web site. This revised kernel provides you with two extra functions. **makeInterrupt21()** simply sets up the interrupt 33 service routine. Function *interrupt21ServiceRoutine()* is henceforth automatically called whenever an interrupt 33 happens. It calls a function in your C code **void handleInterrupt21(int ax, int bx, int cx, int dx)** that you will need to write. The AX, BX, CX, DX parameters passed in the interrupt call will show up in your *handleInterrupt21()* function as parameters ax, bx, cx, dx.

In **kernel.c** create the new function at the end of the file. Leave it empty for now. Replace all prototypes with just the one to *handleInterrupt21()*.

Adding Interrupt Function Calls

We are now finally ready to have the interrupt 33 handler provide all developed services. We will write this so that the number of the requested service is stored in AX, all arguments in BX, CX and DX. Define the interrupt handler as follows:

Function	**AX=**	**BX=**	**CX=**	**Action**
Print string	0	Address of string to print		Call **printString(bx)**
Read string	1	Address of character array to store entered keys in		Call **readString(bx)**
Clear screen and set colors	12	Background color (1 – 8)	Foreground color (1 – 16)	Call **clearScreen(bx,cx)**
Write integer	13	Integer to print		Call **writeInt(bx)**
Read integer	14	Address of integer variable to store entered number in		Call **readInt(bx)**

If AX is any other value print an error message. Now rewrite the main() function of **kernel.c** to appear like the code at right. First enable interrupt 33. Next replace all function calls with interrupt calls. It should work exactly like it did previously.

```
void handleInterrupt21(int,int,int,int);

void main()
{
  char line[80];
  int x;

  makeInterrupt21();
  /* BlackDOS header with interrupts */
  Interrupt(33,12,backgroundColor, foregroundColor,0);
  interrupt(33,0,"Hello world\r\n\0",0,0);
  interrupt(33,0,"\r\n\0",0,0);
  interrupt(33,0,"Enter a line: \0",0,0);
  interrupt(33,1,line,0,0);
  interrupt(33,0,"You wrote: \0",0,0);
  interrupt(33,0,line,0,0);
  interrupt(33,0,"\r\n\0",0,0);
  interrupt(33,0,"Enter a number: \0",0,0);
  interrupt(33,14,&x,0,0);
  interrupt(33,0,"You entered \0",0,0);
  interrupt(33,13,x,0,0);
  interrupt(33,0,"\r\n\0",0,0);
  while (1) ;
}

/* more stuff follows */
```

Conclusion

When finished, submit a .tar or .zip file (no .rar files) containing all needed files (**bootload**, **osxterm.txt**, **kernel.asm**, **compileOS.sh**, **kernel.c** and **README**) to the drop box. **README** should explain what you did and how the t.a. can verify it. Your .zip/.tar file name should be your name. Be sure to include your name under "Author" in the opening banner of the kernel.

Last updated 1.22.2016 by T. O'Neil, based on material by M. Black and G. Braught. Previous revisions 1.14.2015, 2.21.2014.