CSC 565 - Operating Systems
Spring, 2008

Project F - Build on It!

Objective

What's the point of making your own operating system if you can't tinker with it? In this project you will make an addition to your operating system. The addition can be of your choosing, though several options are listed below.

The following are possible projects:

A Text User Interface - Make a Graphical Shell

Most users consider a command line to be hostile and threatening. They prefer an easy-to-use graphical interface for their software.

A TUI is like a GUI, except that all the graphics are done with text. Create a TUI shell that presents all the shell commands in a colorful, easy to understand format. Allow the user to select functions with arrow keys instead of typing out commands.

An example of a TUI shell is given in the *floppya.img* posted on Blackboard. Your interface need not look identical or even similar.

Recall the "Hello World" example from the first project. You should be able to write a function that prints a string in a particular color at a specific row/column coordinate of the screen. With such a function, this becomes reasonably simple.

A good reference is located at: *http://my.execpc.com/~geezer/osd/cons/index.htm*.

File Allocation Tables

The file system that you implemented in this operating system is very restrictive. A file system should not limit the number of files to 16. Files should be able to take up the whole disk.

The MS-DOS File Allocation Table system, which itself is starting to become dated, was in its time considered to be exceptionally elegant. For a 3 1/2 disk, this is probably the best file system strategy.

The FAT system should have a several sector File Allocation Table at the beginning of the disk. The table should have a 16 bit entry for each sector on the disk (there are 2880 total). Files are represented as a linked list, with each FAT entry containing the number of the next entry in the file. An entry of 0xFFFF means end-of-file. At 2880 sectors * 2 bytes per entry, you will need 12 sectors to hold the FAT.

Next should follow a directory. Each directory entry now only needs to store the name and the first FAT entry for the file. At 8 bytes per entry (6 for name + 2 for FAT entry), you can now support 64 files in a single directory sector.

You should modify the readFile, writeFile, and deleteFile functions in the kernel. You should then modify (or rewrite) loadFile.c to load files onto your system.

It is not difficult to model the MS-DOS file system exactly: you will need a duplicate FAT table and more extensive directory entries holding the date, time, and attributes. If you do this, you will have the added benefit that Windows and Linux can read and write files to your system directly; you will not have to revise loadFile.c.

Utilities

Most operating systems do not just come with a shell; they provide a suite of utilities. Among these utilities is always a disk formatter (DOS: format.exe) and an application to copy the system (DOS: sys.com).

Formatting is reasonably simple because there is a BIOS interrupt call to do it. You just need to make the call with the correct parameters.

You should write two utility applications for your operating system. The first, *format,* should take an empty floppy, format it, and copy the bootloader and kernel to it. It should also set up a disk map and directory. The second, *filcpy*, should copy a file from one disk to another.

It is okay, by the way, to do this using one drive. Just prompt the user “Enter original disk” and “Enter new disk” and let the user swap them.

Directories

Modify the file system to support subdirectories. A directory can simply be implemented as a file holding directory entries. You will need to add a byte to each directory entries specifying whether it is a file or a subdirectory.

You should also add a “cd” (change directory) function to your shell that works similarly to the DOS & Unix function. The cd function must be able to return to the root.

Protected Mode

Warning: this project will require you to write in assembly language.

Modern operating systems for the PC do not run in 16 bit real mode anymore. Most switch to 32 bit protected mode immediately upon start up. This is not difficult to do; there are step-by-step tutorials online, many including assembly code. The assembly code to switch is simple enough to fit easily into the boot loader. Basically, you need to make a GDT table, an interrupt descriptor table (IDT), and call a special instruction that loads them and changes the processor state.

There are three major advantages to switching to protected mode:

1. You can compile your code with gcc
2. You can address up to 4GB of memory, rather than 640kB.
3. You can make memory truly virtual, completely isolating processes from each other.

There is also one major disadvantage:

1. BIOS interrupts are no longer available. This includes support for floppy disks, and the interrupt 0x10 function for writing to the screen.

You will also need to compile your kernel and shell in 32 bit mode. The compiler and linker have a command-line flag for this.

There are two ways to deal with the BIOS interrupt problem. The first is to write a function that switches back to real mode before calling these interrupts. This is very tricky and next-to-impossible to debug. The second, and better way, is to write the floppy disk driver, print character, and read character functions yourself. A floppy disk driver is very complicated, but fortunately people have already written it (obviously). If you browse around online you are likely to find several source codes.

To complete this project, you should switch to 32-bit protected mode in the kernel and then replace the calls to the various BIOS interrupts with your own code.

Message Passing between Processes

Most operating systems with multiprocessing have some mechanism for passing messages from one process to another. This is used, for example, when you minimize a window (an “event”), or with Unix pipes.

You should create two new system calls: sendMessage and getMessage. sendMessage should take a buffer holding a message and a process number to send it to. getMessage should be called with an empty buffer to hold the message. getMessage should block until a message is received.

If a process calls sendMessage to a process that is not currently waiting for a message, that message should be stored by the kernel. sendMessage should not block. When the destination process calls getMessage, the messages should be delivered in order.

You should demonstrate message passing with two programs passing messages to each other.

Submission

You should submit a .zip or .tar file (no .rar files please) containing all your files and a shell script for compiling on Blackboard Digital Dropbox. Be sure that all files have your name in comments at the top. Your .tar/.zip file name should be your name. You must include a README file that explains 1) what you did, and 2) how to verify it.