Build an Operating System from Scratch: A Project for an Introductory Operating Systems Course

Michael Black
American University
4400 Massachusetts Ave, NW
Washington, DC 20016
(202) 885-2011
mblack@american.edu

ABSTRACT
This paper describes a semester project where students design an operating system from the ground-up, capable of booting from a floppy disk on an actual machine. Unlike previous projects of this kind, this project was designed for students with only one semester of programming experience and no prior exposure to data structures, assembly language, or computer organization. Students nevertheless wrote a full system consisting of system calls, program execution, a file system, a command-line shell, and support for multiprocessing. The project was assigned to a class and successfully completed by nearly every student.

Categories and Subject Descriptors
D.4.7 [**Operating Systems**]: Organization and Design; K.3.2 [**Computer and Information Science Education**]: Computer Science Education

General Terms
Design

Keywords
Operating systems, education

1. INTRODUCTION
Can a second-year computer science student write an operating system from scratch? Furthermore, could a student complete this system in one semester, with no assembly language or computer organization experience, and still boot the system on a personal computer? This paper describes one such project developed for our operating systems course. The project is designed to give undergraduate students concrete, tangible experience in writing an operating system for an actual machine.

Various projects have been designed for operating systems courses over recent years. Possibly the most ambitious projects have been the so-called "bare-metal" operating systems [5], where students write a system to run directly on a computer without simulators or software underneath. Well-known examples of such systems include Minix and GeekOS, and are usually intended for upper-level or graduate courses. These systems, typically approximating Unix, tend to be intricate and complex, and include substantial amounts of prewritten code (14886 lines for Minix and 4202 lines for GeekOS) [5,8]. The complexity of the systems make them difficult to assign to students at smaller liberal arts colleges which offer only one operating systems course. My objective, described in this paper, is to construct a simple "bare-metal" teaching operating system suitable for a small computer science program. The operating system described in this paper is under 1/4 the size of GeekOS, making it one of the smallest and simplest teaching operating systems yet developed.

There are several key advantages to such a project over a higher-level project that isolates students from the machine. First, students gain a deeper understanding of the computer itself, experiencing first-hand how such concepts as segmentation, interrupt-vectors, and memory management actually manifest themselves in their own computers. Second, when forced to program without the familiar POSIX routines and library functions, students realize how much support operating systems give them. Third, there is a thrill in booting a machine on a system that is entirely one's own, making the often-abstract operating systems concepts much more tangible.

Despite these advantages, of the many operating systems projects that have been designed, relatively few are truly "bare-metal." More common are systems to be run on simulated computers such as Nachos or OS/161 [3,5], or projects that simulate operating system components in Java or other high level languages [6]. Bare metal systems are difficult to debug, need special tools to develop, and require background knowledge about the machine. Some assembly language is required. And many advanced concepts typically taught in operating systems courses, such as page replacement or thread management, cannot be easily built into the system. It is a daunting task for undergraduates to reproduce Unix in an introductory course.

However, the first operating systems for personal computers tended to be very simple and lacked most of the advanced concepts taught in typical operating systems courses. CP/M, for example, provided little more than a rudimentary file system, a small set of system calls, and a command-line shell [7]. Because modern PCs are still capable of emulating the IBM 5150, a bootable operating system need not be as complex as Windows

Vista. My project has students constructing an operating system with roughly the same complexity as CP/M [13].

The project is nevertheless ambitious. Our operating systems class is intended for sophomore-level students with only one semester of prior programming experience. Students know Java, but are not expected to know C, assembly language, or any computer organization coming into the course. This project is consequently designed to be as simple and basic as possible. All programming is done in C, a minimum of prewritten assembly code is given, and students are given a substantial amount of step-by-step instructions. The entire finished project consisted of approximately 1250 lines of code, of which 320 lines are simple assembly routines provided to the students and approximately 930 lines are C functions students are expected to write.

The finished project consists of the following components:

- A bootloader to boot from a 3 1/2" floppy disk
- A set of system calls made using an interrupt
- A file system
- The ability to execute a program from a file
- A command-line shell with a minimum of necessary commands (directory, type, copy, delete, execute).
- Multiprocessing and basic memory management

2. DESIGNING THE PROJECT

My objective in designing this project was to make the system as simple as possible yet capable of booting on an actual computer and functioning as a recognizable operating system. I tried to minimize the number of source files, the total lines of code, and the number of prewritten assembly functions provided to the students. No prewritten C code is provided to students, and assembly functions are limited to routines that can only be easily accomplished in assembly. The resulting operating system consists of only five source files (bootload.asm, kernel.c, kernel.asm, shell.c, shell.asm), two of which (kernel.c and shell.c) are entirely written by the students. Figure 1 shows a screenshot of the finished system. Table 1 shows the assembly functions given to students.

Although the project is intended to complement a standard undergraduate operating systems course and give students practical experience in a wide variety of topics, certain areas are emphasized while others are omitted. Topics covered in the projects include system calls and interrupts, file systems, and shells. Process support is included in the last project module due to complexity, and students are only expected to complete round-robin scheduling. Memory management, apart from the 8088's built in segmentation, is largely omitted, and virtual memory and protection are not included in the project for reasons given below. GUIs are also not addressed in the project due to limited time, but a simple extra-credit GUI project was created for students to attempt. On the other hand, file systems and system calls are strongly emphasized because I found them critical to making the operating system behave realistically.

The operating system, when completed, consists of three components: a bootloader, a kernel, and a shell. The bootloader, which is provided to the students, merely loads the kernel from sector 3 of the disk to memory. The kernel contains an interrupt handler that provides set of system calls, and loads the shell. The shell provides a command-line interface. System calls are made from the kernel and shell by calling interrupt 21 (in an homage to MS-DOS). Table 2 shows the list of system calls the students are required to write, in the order that they are assigned. Table 3 shows the shell commands.

Table 1. Assembly functions provided to students

Name	Function	Proj.
bootloader	loads and executes kernel	A
putInMemory	puts a byte in memory	A
interrupt	makes an interrupt call	B
makeInterrupt21	sets up int 21 vector	B
handleInterrupt21	calls C function on int 21	B
launchProgram	calls program at 0x20000	C
makeTimerInterrupt	sets up timer interrupt (8)	E
handleTimerInterrupt	calls C function on timer	E
returnFromTimer	returns to active process	E
initializeProgram	sets up process stack frame	E
setKernelDataSegment	sets data segment to kernel	E
restoreDataSegment	sets data segment to user	E

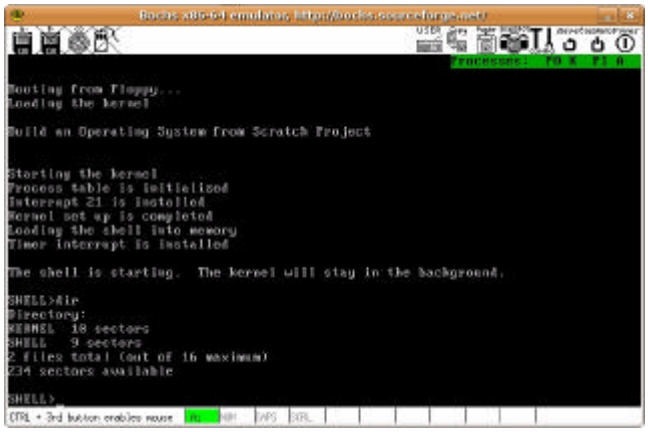

Figure 1. Screenshot of operating system at startup

The file system is illustrated in Figure 2. The first two sectors of the disk, after the bootloader, are reserved for a Disk Map and a Directory. The Directory consists of 16 32-byte file entries, where each entry has room for a 6 byte file name and 26 bytes of sector numbers specifying the file's location. The Map serves as a kind of free list; it consists of 512 bytes representing the first 512 sectors on the disk, with a free sector denoted by 00 and a used sector denoted by FF.

The disadvantages of this primitive file system are obvious; besides restricting the user to 16 files, it also restricts the maximum file size to 13kB (not a problem as the finished kernel is not more than 5kB), is unable to reference sectors above 256, and is utterly unscalable. However, there are a couple key advantages. Unlike most practical file systems, it does not require the students to have prior experience with linked lists or other data structures more complicated than an array. Since every sector is referenced by a single byte, students do not need to understand binary math to write the file system handlers.

Program execution is also very simple. All executables in this operating system consume less than 64kB of memory, have no segmentation, and have their entry point at the beginning of the program. The kernel is loaded into the second 64kB block of

memory (from 0x10000 to 0x1FFFF). All user programs (including the shell) are loaded into the next 64kB (from 0x20000 to 0x2FFFF). Assembly routines are provided to set up the stack and segment registers; students only need to copy the program to memory and call the routine to launch it.

Table 2. System calls written by students

AX=	Function	Project
0	print string	B
1	read string from keyboard	B
2	read disk sector	B
3	read file	C
4	execute program	C
5	terminate program	C
6	write disk sector	D
7	delete file	D
8	create and write to file	D
9	kill process	E
A	wait on process	E

Table 3. Shell commands written by students

Command	Function	Project
type	print out text file	C
execute	load and execute program	C
delete	delete file	D
copy	copy file	D
dir	print out directory	D
create	create a new text file	D
kill	kill process	E
execback	execute program in background	E

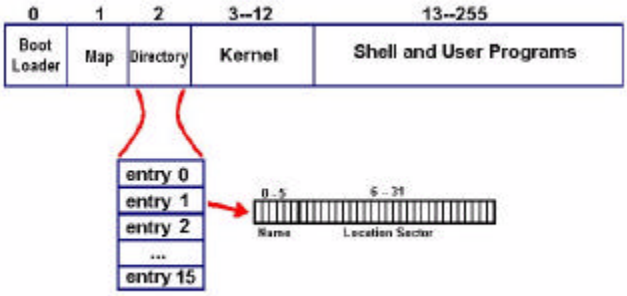

Figure 2. Block Diagram of File System

In the multiprocessing step, user programs may be located in any of the first eight 64kB blocks. A process table holds the identities of currently running programs and their current stack pointer address. On each timer interrupt, the register state is backed up and a new program chosen from the process table in round-robin fashion. Although assembly language routines for handling the timer, backing up and restoring the state are provided to students completing this step, students still must handle the process table and cope with race conditions.

The project makes heavy use of existing BIOS functions to print to the screen, read from the keyboard, and read and write sectors to the disk [11]. These BIOS functions are contained in ROM in all modern PCs and are holdovers from the earlier days of DOS; they are typically not used in modern operating systems. Using existing BIOS functions greatly simplifies my operating system because students do not have to write floppy or console drivers. However, to use these functions, the computer state must be kept in 16-bit real mode. Virtual memory and protection, which require the 32-bit mode, are consequently not addressed.

3. ASSIGNMENTS

I modularized this project by dividing it into five components, labeled Project A through E. Students were given two weeks to complete each component. Before beginning Project A, students were given a warm-up project writing a simple Unix shell to give them experience coding in C. The projects are designed to increase in difficulty as students become more and more accustomed to low-level programming.

All students were given a Linux account on the department server containing the necessary development tools. Students used Bochs [9], rather than a real machine, to test their system. Because gcc and nasm are not capable of producing 16-bit code, students developed their system using the archaic bcc compiler and as86 assembler (from the bin86 toolset). Additional tools included the dd command to produce the binary floppy image and hexedit to verify their binary files. All of these are open source tools available on standard Linux repositories.

3.1 Project A: Getting Started

Project A is simply a getting-started project. Students were provided with step-by-step instructions for getting started. The project consisted of writing a kernel that prints "Hello World" to the screen by writing characters directly to video memory. Students are expected to write a program *kernel.c* to copy each of the letters to memory starting at 0xB8000 using the *putInMemory* function, and then halt the computer with a while(1) loop.

The two assembly functions provided are very simple. The bootloader uses the BIOS read sector call to load the kernel sectors into memory at 0x10000, and jumps directly to the kernel. The *putInMemory* function simply loads the address and data parameters from the stack and writes them to memory.

The Project A description provides students with the commands to run the as86, ld86, and bcc tools to compile the project, and the dd tool to create the image. In addition to the kernel.c deliverable, students are expected to construct a shell script to compile and assemble their project.

3.2 Project B: System Calls

Project B consists of writing the kernel interrupt handler. Students are required to write system call handlers to print a string to the console, read a string from the keyboard, and read a sector to a buffer. At the end of Project B, students' systems read a text file from the disk and print it to the screen.

A new kernel.asm is provided to students with three additional assembly functions. Function *handleInterrupt21* is a stub that calls a C function *interrupt21* that students must write. *makeInterrupt21* modifies the interrupt vector table to call *handleInterrupt21* on an interrupt 21. *interrupt* takes an interrupt number and four parameters, puts the parameters in registers AX-DX, and makes the interrupt call. The *interrupt* function is intended to call both BIOS calls and the students' own interrupt.

Students are required to create the following in kernel.c:

- Write a *printString* function to print out a null terminated string by making repeated print character (interrupt 10) BIOS calls.
- Create a *handleInterrupt21* function that is called on an interrupt 21 and takes the contents of the AX, BX, CX, and DX registers as parameters. Students have *handleInterrupt21* call *printString* when AX is 0. Subsequently, whenever students write a kernel function, they must call it in *handleInterrupt21*.
- Create a *readString* function that reads characters from the keyboard using BIOS interrupt 16. The *readString* function terminates the string when ENTER is pressed. Students are expected to write handling for the BACKSPACE key.
- Create a *readSector* function that reads a single sector from the disk using BIOS interrupt 13. Students must write *mod* and *division* functions to convert a raw sector number to the CHS format required by interrupt 13
- In main(), call *makeInterrupt21*, and test the functions by reading a string, reading a sector, and printing both out.

3.3 Project C: Shell

Project C is much more demanding. Students construct system calls to read files from the disk and execute and terminate programs, and write a rudimentary shell. An additional assembly function *launchProgram* is provided to allow students to start user programs. *launchProgram* sets the stack pointer to the top of the segment, sets the segment registers to 0x20000, and jumps to 0x20000. Students are also provided two additional resources: a shell.asm containing the same interrupt function as in kernel.asm, and a utility to create map and directory sector images.

Students are required to do the following steps:

- Write a *readFile* function. The function takes a file name, reads the directory sector, and tries to match the file name against each entry. When found, it must use the entry's sector list to read the file to a buffer sector-by-sector.
- Write a *loadProgram* function that calls *readFile* to load an executable file, copies it byte-by-byte to memory at 0x20000 using *putInMemory*, and executes it with *launchProgram*.
- Create a new program shell.c that should make the appropriate interrupt 21 calls to prompt the user and read commands. Students must then write shell commands *type* and *execute* to print out text files and execute programs.
- Write a *terminate* function using *loadProgram* to reload the shell.

3.4 Project D: Writing to Files

Project D completes the single-process operating system. No new assembly functions are provided. Using the functions they have already written, students must:

- Make a *writeSector* function that writes a 512-byte buffer to a disk sector.
- Make a *writeFile* function that takes a file name, a buffer containing the file contents, and the length of the file. The function searches through the disk map to find sufficient free sectors and uses *writeSector* to copy the file buffer to the disk. It then must modify the directory to add an entry and record where the file is stored.
- Make a *delete* function that takes a file name and deletes it. *delete* finds the file in the directory, removes each sector of the file from the map, and marks the directory entry as deleted.
- Add shell commands to copy a file, delete a file, and list the directory. Students must also make a simple editor: a *create* shell function that reads in lines of text from the keyboard and writes them to a file.

3.5 Project E: Multiprocessing

At this stage, students were given a choice to complete Project E or do an additional project of their choosing (making a GUI or adding virtual memory were offered as examples). Nearly all students chose to do Project E, adding multiprocessing to their operating system.

Several new assembly functions had to be provided to allow multiprocessing capability. *handleTimerInterrupt* is a stub function that saves the registers on the stack and calls the student's C function on a timer interrupt. *makeTimerInterrupt* initializes the system timer to make periodic interrupt 8 calls, and sets up the interrupt vector table to call *handleTimerInterrupt*. *initializeProgram* creates a stack frame for executing a program but does not actually start the program, while *returnFromTimer* reloads registers backed up on the stack and returns to a preempted program. Functions *setKernelDataSegment* and *restoreDataSegment* set the DS register to the kernel space and user program space respectively, allowing students to create and access a process table in the kernel segment.

Students were required to do the following:

- Start the timer running and print the string "Tic" on each timer interrupt, to verify that the timer works.
- Make a global process table array and *currentProcess* variable in kernel.c, storing up to 8 processes. For each process the table must hold whether the process is active, and the address of the process's last stack pointer. Each process table entry refers to a 64kB segment of memory: entry 0 is 0x20000, 1 is 0x30000, and 7 is 0x90000.
- Modify *executeProgram* to initialize the program instead of loading it, and have it set the appropriate process to active.
- Modify *terminate* to set a process to inactive.
- Write a *handleTimer* function that is called on timer interrupts. The function must select a process from the process table in round-robin fashion, and call *returnFromTimer* with that process's segment number and last stack address.
- Write a *kill* function that sets a process to inactive, and a *kill* shell command.
- For extra credit, allow processes to block on other processes, and revise the execute shell command to block the shell while another process runs. All the students who completed Project E also completed this step.

4. STUDENT EXPERIENCE

I administered the project to a single class of 14 students. The student experiences and results are consequently limited to this single sample. Student project submissions were evaluated solely on whether they functioned correctly and met the stated project requirements. Submissions were not graded on efficiency or code elegance.

The class consisted of 6 sophomores, 4 seniors, and 4 masters-level graduate students. 7 were computer science majors, the rest were non-majors (the most common non-CS major was international studies). All the students had completed the first course in introductory programming, but 3 had not taken the second course and only 4 had taken a computer organization course previously.

10 students correctly completed all 5 components; these included 2 of the 3 students who had only taken the first programming course and 2 of the 4 graduate students. Of the remainder, 3 finished Project D, and the remaining student finished Project C. Since all development work was done on Bochs, students were given the opportunity at the end of the course to test their system on a Pentium 4 computer. 12 of the 14 students' systems acted the same on a real computer as on the simulator.

Most student difficulties were on Projects C and E. In Project C, students tended to have problems reading files, especially files greater than one sector in size. Project E provided two sources of frustration. Many students had difficulty understanding when to switch data segments to the kernel space. Nearly all students attempting Project E had difficulty debugging errors due to multiprocessing race conditions. On the other hand, students did not typically have problems understanding what was required of them, and few students made the mistake of trying to call POSIX functions in their kernel.

A survey was given to the students at the end of the semester asking them to evaluate their project experience. Students unanimously reported that the projects contributed to their understanding of the course material, that they enjoyed the projects, and that they recommended them to future classes. Students were asked whether they felt that the hardest part of the project, the multiprocessing step (Project E), was necessary; 9 answered yes, 2 no, and 3 "not sure".

5. CONCLUSIONS AND FUTURE WORK

In a survey paper of operating systems projects, Anderson and Nguyen address the scarcity of bare-metal teaching operating systems [1]. Some of the well-known previous endeavors, themselves excellent designs, tend to be far too advanced for the typical undergraduate student. Operating systems projects on actual machines, Anderson and Nguyen found, are infrequently assigned in introductory operating systems courses. The consequence is that many computer science graduates lack hands-on operating systems experience.

I set out on this project as a personal challenge to see if a bare-metal operating systems project could actually be programmed from the ground-up by undergraduates lacking prior low-level programming experience. I found that, in fact, such a project can be devised, and that students rise to the challenge. As mentioned above, most students came to the class with only basic programming experience and no hardware background, and nevertheless successfully wrote an operating system.

From my experience, I believe that several factors were necessary to make a project like this successful:

- Providing detailed step-by-step instructions for each stage.
- Keeping prewritten assembly code to an absolute minimum, and providing clear instructions on calling these functions.
- Approximating a simple system like CP/M or early MS-DOS rather than replicating Unix.
- Using BIOS calls instead of custom device drivers.
- Including no advanced data structures, binary math, stack handling, or assembly programming (besides what was provided).

In the future, I hope to extensively document this project, simplify it further, and make it reproducible by instructors at other universities. I am presently developing a GUI shell component, and one of my future plans is to boot from a USB key instead of a floppy disk.

6. REFERENCES

[1] Anderson C.L. and Nguyen M. 2005. A Survey of Contemporary Instructional Operating Systems for Use in Undergraduate Courses. In Consortium for Computing Sciences in Colleges, Northwest, 2005.

[2] Christopher W.A., Procter S.J., Anderson T.E. 1993. The Nachos Instructional Operating System. Winter USENIX Conference.

[3] Holland D.A., Lim A.T., and Seltzer M.I. 2002. A New Instructional Operating System. In Proceedings of SIGCSE Technical Symposium on Computer Science Education.

[4] Liu H., Chen X., Gong Y. 2007. BabyOS: a fresh start, In Proceedings of SIGCSE Technical Symposium on Computer Science Education.

[5] Hovemeyer D., Hollingsworth J.K., Bhattacharjee B. 2004. Running on the Bare Metal with GeekOS. In Proceedings of SIGCSE Technical Symposium on Computer Science Education.

[6] Silberschatz A. and Galvin P. 1999. Applied Operating System Concepts. John Wiley and Sons.

[7] Tanenbaum A.S. 2001. Modern Operating Systems. Prentice Hall.

[8] Tanenbaum A.S., Woodhull A.S. 1997. "Operating Systems: Design and Implementation." Prentice Hall.

[9] Bochs. http://bochs.sourceforge.net. February, 2008.

[10] GeekOS http://geekos.sourceforge.net. December, 2007.

[11] Ralf Brown's Interrupt List. http://www.ctyme.com/intr/int.htm. December, 2007.

[12] OS Development Wiki http://wiki.osdev.org. January, 2008.

[13] CP/M-85. 1982. Zenith Data Systems.