Teaching Operating Systems as How Computers Work

Peter Desnoyers
Northeastern University
360 Huntington Ave.
Boston, MA 02115
pjd@ccs.neu.edu

ABSTRACT

The "Computer Systems" course at Northeastern University is an MS-level core course which attempts to teach students *how computers work*, through a behavioral approach to the concepts involved in operating systems and their interface to the hardware. As an operating system is typically the first reactive system which students encounter in their studies, the goal of the class is to develop an understanding of the tools and reasoning which are involved in understanding and working with the internals of such a system, whether it be a conventional operating system or (as is more commonly found in industry) a consumer product, networking device, or other embedded system. This course is currently in its third year with enthusiastic responses from students, especially those who have been able to apply its lessons in co-operative work assignments, and an undergraduate class teaching substantially the same material is currently underway.

Categories and Subject Descriptors

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

General Terms

Design, Experimentation, Human Factors

Keywords

operating systems, computer science education

1. INTRODUCTION

CS5600, Computer Systems, begins with a simple demonstration — a ridiculously simple one, in fact. The projector displays the console of a computer which has been booted into Linux in text mode; the instructor types the keystrokes 'l', 's', and return. At the beginning of the semester virtually every student can describe what this is doing—listing the files in a directory. By the end of the class, the goal of this class is that these students learn *how* this happened—e.g. the operation of the keyboard driver and scheduler to deliver input to the shell, the virtual memory operations needed to demand page the `ls` executable, and the file system layout, implementation, and actions necessary to locate and enumerate a particular directory.

In effect the goal of this class is to teach students how computers work. For a simple enough computer, this understanding might be obtained through an understanding of the hardware. For any general-purpose computer today, however, most of its working parts are virtual and reside as software components within the operating system.

The approach this class takes to teaching its material matches this goal, as well. The class does not teach how a computer operating system is structured, or what its most important concepts are - it tries to teach *how it works*. In particular, by this we refer specifically to the causal chains of events which occur in response to stimuli such as user requests, and which as a whole comprise its behavior.[1]

This goal is reflected at all levels within the course. Lectures emphasize event-by-event walkthroughs to illustrate concepts. Material is presented in order from low-level components up, so that higher-level behaviors may be seen as combinations of previously-learned lower-level ones. Assignments are used to illustrate these behaviors, with test scripts or other oracles provided so that students may identify correct behavior. Finally, evaluation on exams is done in a way which emphasizes exposition of system behavior in response to state and stimuli, rather than descriptive or computational problems.

With the introduction of this revised course, enrollment has increased by 60%, while the pool of eligible students has remained roughly constant. A university-wide change from paper to online course evaluations coincident with the first offering of this course prevents direct comparison with previous versions; however, evaluations have been well above average, with many students describing the new syllabus and format in highly enthusiastic terms.

2. BACKGROUND

CS 5600, Computer Systems (previously CS G112) in the College of Computer and Information Science at Northeastern is a core course in the professional MS program - students

[1]We note that an operating system is often the first reactive system which students see in their studies, requiring them to be introduced to the idea of reasoning about the evolution of system state given a sequence of inputs over time.

SIGCSE'11, March 9–12, 2011, Dallas, Texas, USA.

must take either this or Managing Software Development as one of 8 courses. In addition, most students undertake a cooperative assignment in industry as part of their degree; a significant fraction of the students in this course have already been on co-op. The students are primarily international, with a broad range of undergraduate preparation that may or may not have included an operating systems course or exposure to the C programming language.

Prior to 2008 the class was a traditional operating systems concepts class. Anecdotal evidence indicated that the class was not meeting the needs of students—enrollment was low, and co-op employers had commented on students' lack of operating systems background. With a change of instructor, the course was re-focused to emphasize topics which would be of long-term use for students pursuing careers in embedded systems, such as networking and storage devices, which represent a significant number of co-op and career opportunities for students in the College.

3. BEHAVIOR AS A PRIMARY CONCEPT

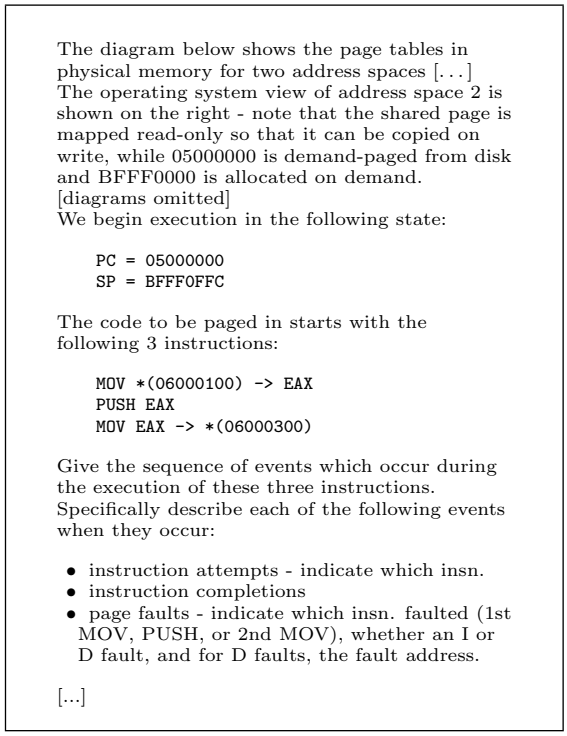
The class is structured around operating system behavior for several reasons. First, because it fits the intuitive idea of "how it works". Part of the goal of the class is to make students comfortable working "under the hood", as would be the case in embedded fields such as consumer or network devices; the author believes that the form of operational understanding emphasized in this class is a key factor in achieving this familiarity. In addition we wish to teach principles that are general, rather than specific to an individual operating system, and while structure and abstractions may vary from one OS to another, behavior such as response to a page fault remains relatively constant.

The level of abstraction at which this behavior is described varies according to the complexity of the interaction. Thus the discussion of keyboard input in the first lecture would serve as a detailed design for the keyboard driver of the simple machine being described, while when discussing virtual memory and demand paging we refer to reading a block from a file (itself a complicated process) as one operation.

Assessment of student comprehension of these behavioral concepts poses a set of challenges. Programming assignments when successfully completed result in code which when executed will reproduce the behavior in question, but do not necessarily prove understanding of the behavior by the student. For in-class exams the problem is worse, as typical problems involving calculation or description fail to capture the behavioral aspects of a system at all.

Instead we place a heavy reliance on questions which ask for pseudo-event traces in response. Like pseudo-code, these questions ask for a description at a prescribed level of detail, in this case describing the sequence of actions which unfold given a specified starting state and stimulus. An example of such an assessment is shown in Figure 1, where students are asked to trace the virtual memory operations which occur as several lines of machine code are faulted in and executed.

In contrast, programming assignments are designed to illustrate the behavior patterns being studied, as well as assessing the students' knowledge of these patterns. For the most part the assignments are accompanied by test scripts; successfully completion of all tests should correspond to full marks on the assignment. The assignment thus acts as a simulator, reproducing the behavior the student heard described in lecture, or failing tests until corrected if that understanding proves to be incorrect.



The diagram below shows the page tables in physical memory for two address spaces [. . .]
The operating system view of address space 2 is shown on the right - note that the shared page is mapped read-only so that it can be copied on write, while 05000000 is demand-paged from disk and BFFF0000 is allocated on demand.
[diagrams omitted]
We begin execution in the following state:

```
PC = 05000000
SP = BFFF0FFC
```

The code to be paged in starts with the following 3 instructions:

```
MOV *(06000100) -> EAX
PUSH EAX
MOV EAX -> *(06000300)
```

Give the sequence of events which occur during the execution of these three instructions. Specifically describe each of the following events when they occur:

- instruction attempts - indicate which insn.
- instruction completions
- page faults - indicate which insn. faulted (1st MOV, PUSH, or 2nd MOV), whether an I or D fault, and for D faults, the fault address.

[...]

Figure 1: Behavior-based exam questions

4. LOADING AND CONTEXT SWITCHING

The beginning of the class is designed to help students develop a mental model of an operating system and programs in memory on a simple machine, and be able to walk through the steps involved in loading a program, invoking OS functions, and context switching. For this purpose we define a simple machine, executing only on the blackboard. Thus we do not need to fully define the CPU; instead we give only its assembly language instructions – the basic MOV, CALL, etc. found on most CPUs, with numbered registers and fixed-length instructions for simplicity.

For our purposes the key characteristic of this machine is its set of simple I/O devices mapped into the top of its 16-bit address space. At present these consist of a text-mode 80x24 video display buffer, a keyboard controller, a primitive disk controller, and 4 serial ports for multiprocessing.

Rather than taking a top-down approach, we begin with a very simple "bare-metal" program, copying the string "Hello World" to the display buffer, as shown in Figure 2. Starting with this program, which may easily be understood in its entirety with little effort, we proceed in stages to develop a simple single-user system, transitioning from assembler to pseudo-code when appropriate. Hardware-specific functionality is split into separate procedures, and then segregated to a separate region of the memory space to allow loading of programs; loading is done by a simple command line which allows the user to specify disk blocks and then begin execution. Next we introduce a system call vector at a well-known location, as well as its software interrupt equivalent,

```
frame_buf   eq   F000
str         .db  'Hello World'
len         eq   11

begin:      MOV  #frame_buf → R0
            MOV  #str → R1
            MOV  #len → R2

loop:       MOV  *(R1++) → R3
            MOV  R3 → *(R0++)
            SUB  1 → R2
            JMP  NZ, loop

done:       JMP  done
```

Figure 2: Our first program

```
switch( uint16 *oldsp, uint16 newsp)
_switch:  PUSH  R7 (caller-saved)
          PUSH  R6
          MOV   *(SP-8) → R0
          MOV   SP → *R0
          MOV   *(SP-10) → SP
          POP   R6
          POP   R7
          RET
```

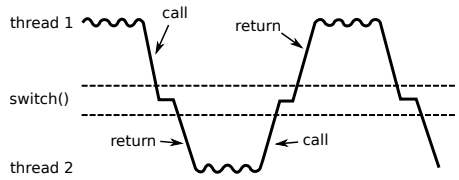


Figure 3: Alternate context switching depictions

providing a stable interface for binary compatibility across instances of the OS.

We then describe the concept of a command-line shell which uses OS functions to load and execute other programs, hardware interrupts and their application in buffering keyboard input, and a file system using a directory of {name, start, length} tuples in block 0. At this point we have explained a system of roughly the sophistication of MS-DOS 1.0, a contemporary description of which is part of the assigned reading. [8]

Using the motivating example of the 4 serial ports with attached terminals, we review subroutine linkage and stack frames and introduce context switching and multiprogramming. On a simple system such as this, process state can be pushed onto the stack and represented by nothing more than a saved stack pointer value, allowing context switches to be easily walked through on the blackboard. Drawing on the prior description of hardware interrupts, a timer device is then introduced, and with that, preemptive context switching.

In each iteration of the system there is an emphasis on the memory map, using the classic organization of code at the bottom, heap growing up, and stack growing down. This organization is used to present the memory map for the multitasking system, leading to a discussion of methods for loading programs at this differing locations in memory—separate compilation, load-time fix-up, position-independent code, and segment registers.

In practice most students have a great deal of difficulty with context switching; although they understand the concept, they have trouble following the exact sequence of events which occur. This difficulty is seen regardless of prior preparation or experience; we believe that it is because a context switch function, which is called in one thread but returns in another, breaks a fundamental programming abstraction which is more deeply ingrained in the more experienced students. We have found it helpful to present multiple ways of visualizing the context switch process, including both the low-level instructions themselves as well as higher-level views of behavior, as shown in Figure 3.

The assignment for this section of the class involves writing a user-space loader, context switching between threads, and scheduling multiple threads to respond to user input. Students are provided with a code framework which creates an executable data segment and two network connections, as well as functions for switching the stack pointer and initializing a stack. In addition, a build script is provided which extracts the contents of simple ELF executables in binary form which may be easily loaded into memory and invoked, as well as an assembler file which defines a vector table these "micro-programs" can use to invoke functions within the main executable.

Students must write several micro-programs and a simple shell to load and execute them, as well as the "system calls" such as `printline()` which are invoked through the vector table. Next they implement context switching between two threads, and finally implement a thread context switch, and modify their input routine to switch between two threads as user input is received on the network connections.

For many students this assignment is a high point of the class — although tiny, they have written an operating system of their own. Where in prior classes they were looking down at the interface to the OS, by creating a sort of micro-OS with basic functions, they become able to look at the remainder of the class as elaborations on top of this base.

5. SYNCHRONIZATION

Having introduced threads and preemptive switching, the concept of race conditions is described via the classic bank balance update example. We walk through the operation of a spinlock at the memory transaction level, and show how to construct a mutex using spinlocks or interrupt disabling in conjunction with a thread scheduler.

For the remainder of the section on synchronization we focus on abstract synchronization primitives rather than lower levels. Using classic problems such as bounded buffer for motivation, we introduce semaphores, monitors, and deadlock.

This is in a sense the most practical unit of the class, as it is likely that most students will have to write and debug threaded code at some point in their career. As such, there is a strong focus on best practices and rules of thumb, e.g.:

- Logical association of mutexes with data, rather than code sections; we therefore downplay the term "critical section".
- Safe handling of shared variables, through means such as creating local copies before releasing a lock.
- Use of lock ranking to avoid deadlocks.

Although semaphores are introduced early on, the primary synchronization mechanism used in this class is the

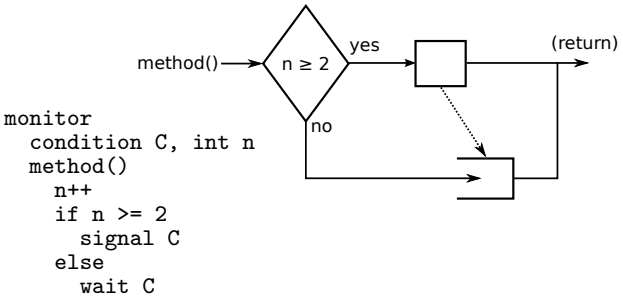


Figure 4: Graphical illustration of synchronization

monitor. We do this for both practical and philosophical reasons. First, monitors are the more likely form in which students will later encounter concurrency control, in either Java (as synchronized objects or re-entrant locks from *java.utils.concurrent*) or Posix threads *pthread_cond* operations. More importantly, we believe monitors are more pedagogically sound, as the intuitive understanding of how a semaphore works must be violated (by calling `wait` in one thread and `signal` in another) in order to use it for any but the simplest of problems.

We focus on reasoning about the operation of multiple threads, an area which presents difficulties for even experienced programmers. Rather than understanding *the* order of events which will occur, students must learn to understand *all possible* orders of events. To cope with this, we introduce a flowchart-based graphical notation for visualizing the flow of control in multiple simultaneous threads, as shown in Figure 4. Although not as precise as the n-dimensional state graphs used in e.g. Bryant and O'Hallaron [2], the ability to visualize multiple threads moving through a 2-dimensional network provides students with an effective metaphor to understand the state space of a multi-threaded system. [2]

We augment this section with a review of continuous-time Markov models, providing students with another tool to reason about state-based behavior. This allows us to give the following assignment:

- Write a pseudo-code description of a monitor-based solution to a concurrency problem (e.g. the sleeping barber)
- Translate this solution into C using Posix thread condition variables, and execute this solution.
- Run the same solution as an event-driven simulation using a virtual-time thread scheduler, with additional instrumentation to gather state occupancy and throughput statistics.
- Finally, formulate the problem as a Markov model and solve the balance equations, deriving expected state occupancies and transition rates.

A significant problem in assigning multi-threaded programming assignments is that unlike the single-threaded case, student observation of program behavior appears to do little to dispel misconceptions about the operation of a particular piece of code; instead, they often need to be shown by some other means that it is operating incorrectly. Using the approach described here, we are able to provide an oracle to test the Markov model solution (e.g. selected digits from the decimal solution), and once students have successfully solved that portion of the problem, they may in turn use it to verify operation of their code.

[2]Graph representations of concurrency (e.g. Petri nets) have a long history, however a review of current textbooks shows no examples of their use in teaching introductory concurrency concepts.

6. VIRTUAL MEMORY

Virtual memory plays a central role in modern operating systems, and without understanding it much of the operation of these systems must remain an unknown black box. We therefore focus heavily on this topic, covering the range from hardware mechanisms through software policies and algorithms.

Early on we introduced the use of segment registers for run-time relocation of executables; we now examine per-process memory protection, leading to simple base-and-bounds translation hardware. A walk though several examples leads to the necessity of separate user and supervisor modes, along with mechanisms (e.g. software interrupts) to allow controlled transitions into supervisor mode. At this point we have covered all the infrastructure needed to understand the non-file system portions of the original Unix paper [14] which is assigned for reading.

There is a brief (and very conventional) discussion of heap allocation strategies (first/best/next fit, buddy) and external and internal fragmentation. These problems motivate the introduction of paged translation. Students become familiar with the steps involved in address translation via page tables, as well as use of a TLB to achieve acceptable performance. To reinforce this knowledge we ask students to (a) given a specified table, describe the steps needed to translate a particular address, and (b) for a specified memory map, construct a page table to implement it.

The central idea of this unit is virtual memory, though, rather than address translation. We thus describe the concept of operating system virtual memory areas, and the page fault handling necessary for demand allocation, copy-on-write, and demand paging. As before, the emphasis is on understanding of the sequence of steps involved, as illustrated in the test question shown in Figure 1.

We describe hardware virtualization next, which is perhaps the most conceptually difficult subject in the class. At this point At this point we have erected enough infrastructure that we can describe the difficulty of the problem—i.e. why one operating system cannot be run directly on top of another. The first solution presented is full software emulation, as in e.g. Bochs [5], which despite its practical complexity is conceptually an almost trivial fetch-switch-execute loop. From this we proceed to the trap-and-emulate strategy used on CPUs meeting the Popek & Goldberg virtualization requirements [10], and finally discuss software and hardware techniques for virtualizing non-virtualizable CPUs such as the x86 series. [11]

The homework for this unit is split between paper exercises, in particular pseudo-event traces, and a programming assignment. Creating meaningful programming assignments for this unit is a challenge, as pure user-mode projects are very restrictive, while a kernel-mode assignment would require an amount of development and debugging infrastructure which really needs to be amortized over multiple assignments. To date our assigment requires students to use the `mprotect` system call and a `sigsegv` handler to emulate actual page faulting. They then implement several differ-

ent page replacement policies - FIFO with differing working set sizes, and the VMS LRU approximation [6] using FIFO replacement from the working set and a victim cache with LRU replacement.

7. I/O AND BLOCK DEVICES

In traditional operating systems classes the block storage layer is woefully under-represented in comparison to the attention it receives in both industry and academia. In this portion of the class we focus on the block interface both internally and externally, again paying close attention to the sequences of operations involved, which are typically the crucial determinants of performance.

On the host side of the interface we focus on driver architecture and the performance characteristics of buses such as PCI and PCI-e. We calculate what I/O performance would be for simple memory-mapped devices given typical bus latencies, and use this to motivate the discussion of a typical I/O device using DMA, a descriptor queue, and interrupts. We discuss driver architectures, and put these together to provide students with a picture of the operation of blocking I/O from the application, through the OS and down through to the PCI bus, and back.

Disk drives are then discussed, from the point of view of performance. The goal here is to give students a basic understanding of the mechanical and geometric properties which determine performance—rotation speed, seek time, and zone density. The primary focus of this portion of the class, however, is on features which may be provided on the device side of the block interface. After a brief discussion of transports (SCSI, Fibre Channel, USB, etc.) we cover standard RAID levels (RAID0, RAID1, RAID4 and RAID5) as well as logical volume management (snapshot, migration, storage virtualization) and storage de-duplication.

To date the assignment for this unit has been combined with that for virtual memory. Students are given specifications for a small but otherwise representative disk drive—rotational speed, zones and sectors per track in each zone, and seek time as a simple function of tracks moved. They must then write a timing simulator, which may be validated against a provided sample sequence. Finally they use a page fault trace collected during the virtual memory portion of the assignment, and use their simulator to determine performance of that trace on the specified drive.

On-paper assessment for this unit is a work in progress; to date it has focused on RAID systems and students' understanding of their operation and performance. Thus they may be asked to indicate the sequence of underlying disk operations in response to reads and writes at the volume level, or to calculate the time taken for certain operations, given specified disk seek and transfer parameters.

8. FILE SYSTEMS AND SECURITY

These are the final sections in the class; to date file systems has been covered first, building on top of the discussion of block devices, with security last. However this order may be revised, as permission checking has been an area of significant confusion in the file system assignment.

Unlike other portions of the class, we begin at an abstract level, presenting the ideas of file system namespace and operations. We do this in order to solidify students' intuitive notions of file and directory, providing a firm basis for discussing concepts such as hard and soft links, inclusion of devices in the file system namespace, and Win32 (device-rooted) vs. Unix (single root) path names.

Kernel implementation of file systems is covered briefly, covering dispatch at the file operations level (e.g. to device driver methods) and at the VFS level. The primary focus, however, is on the externally-visible file system structure. We characterize file systems by their solutions to the following problems:

- Free space management: Do they use linked lists (i.e. the original UNIX file system), a bitmap, some array-based equivalent such as a file allocation table, or extent lists?
- File organization: We discuss contiguous files (e.g. ISO 9660), allocation table-based linking, direct/indirect block structures such as used by most UNIX file systems, and extent lists.
- Directory entries and file meta-data: What information is stored about a file? Is there a separation between file meta-data (i.e. i-node) and directory entry?
- Robustness: Here we discuss file system checking, as well as the operation of log-structured file systems and file system journalling.

The assignment for this unit is typically the most intensive one of the semester. Students are given the definition of a simple file system using a file allocation table and UNIX-like file names and attributes, along with several sample disk images, a `mkfs` utility, and test scripts. In addition to this they are given skeleton code to read and write blocks from a disk image file and to interface with the Linux FUSE (File system in USEr space [13]) library.

Given these materials, students create an implementation of the specified file system, which they are able to mount and use on a Linux system. As with prior assignments, students are provided with test suites which may be used to verify their solutions, so that with sufficient effort most teams should be able to obtain full marks.

On-paper assessment of students' understanding focuses on operations across the block interface boundary, as well as the role of buffering in optimizing these interactions, for both FAT and i-node/indirect block organizations. Thus students might be asked for the sequence of reads necessary for certain operations, and of the effect of e.g. a small buffer cache vs. an i-node cache on the number of operations, or to contrast the worst-case overhead for FAT vs. indirect block organizations for retrieving a block at a large offset within a file.

The final unit, security, is quite brief. It focuses on security mechanisms found in common operating systems (i.e. UNIX, Windows) and covers user/group permissions, access control lists, and capabilities as used in Posix and Windows. The goal of this unit is for students to be able to construct user/group permissions and access control lists to implement a given access policy, a skill which many (although by no means all) students enter the class with.

9. COMPLICATIONS

One of the particular issues with this course, as with many other operating system classes, is the use of C for programming assignments. For many students this is their first exposure to an unsafe language, yet we wish to use class time for teaching course content rather than the assignment language. To some extent this problem has been ameliorated

by student use of Valgrind [7] to detect memory bugs. Unfortunately this is not always feasible; in particular, valgrind is unable to differentiate a user-constructed context switch from a severe memory error. To complement valgrind, we are currently working on a set of grading-enforced style rules — e.g. all pointer variables must be initialized to a valid value or NULL— which we hope will further reduce problems.

The other major issue is that of grading. Due to the open-ended nature of the exam questions and cascading effects of errors, considerable judgment is required to assign marks in a way that reasonably reflects a student's performance. The assignments are similarly subject to a wide range of often subtle errors; in both cases it has been difficult for graduate teaching assistants to achieve the desired quality of grading on their own. (This may change as the course matures.) Our solution to date has been the use of "grading parties", where TAs and the professor mark papers, agree on an ordering of answers, and then assign grades accordingly.

As with any novel course, textbook support is problematic. To date we have used Silbershatz et al. [12], and are using Bryant and O'Hallaron [2] in an undergraduate version of the class, but neither is a good fit. Silbershatz is too abstract, avoiding hardware details, although its treatment of synchronization and file systems fits well. Bryant and O'Hallaron cover more hardware details, but not file systems and concurrency are poorly covered. The gap between the textbook and class material has been filled with readings from the literature, as well as student-produced scribe notes and lecture videos posted on the course website.

The use of lecture videos has been highly successful. These are low quality videos, made using consumer-grade equipment operated by the TA, who also typically edits them for posting. Although no doubt responsible for a small drop in course attendance, many students report that the videos have been a valuable study aid.

10. DISCUSSION

In order to provide students with an in-depth exposure to the concepts in this class, certain compromises have been made. Chief among these is the use of user-mode programming assignments, rather than kernel code running on simulated, virtual, or real hardware. This is a deliberate decision, as the goal of these exercises is to illustrate particular issues and mechanisms, not to teach students how to work with a large body of systems code. The largest amount of "scaffolding" required for the CS 5600 assignments is 600 lines of actual code (including headers) for the file system assignment, compared to 4500 lines for project0 in GeekOS [3, 4], or over 7000 for just the scheduler in a recent Linux kernel.

The only area in which this has presented significant difficulty is the virtual memory assignment, where the mprotect/sigsegv mechanism has a number of deficiencies when compared to the equivalent functions in the kernel. With work we believe this may be remedied, either with better support code, or by moving the entire assignment into kernel mode under e.g. QEMU.

The notion of starting from the bottom up in a computer systems class is not new; in particular, Patt and Patel [9] begin at the lowest level with digital logic. After the hardware level, however, their focus turns directly to application programming, while CS 5600 leaves discussion of hardware implementation to digital design classes, but focuses almost exclusively on the layers underneath the application.

Unlike Bryant and O'Hallaron's Introduction to Computer Systems [2, 1], CS 5600 does not attempt to describe those aspects of a computer system which every application programmer will interact with. This is in part due to purpose of the class; as an elective core, it is taken by students with an interest in low-level programming. In addition, however, we consider some aspects of this material to be a fundamental part of any engineering-oriented computer science education.

The class is still evolving, and this paper does not exactly reflect any single semester of CS5600, but rather those portions which have been successful in the past and are being retained, as well as several new approaches which have been tested in a more advanced version of the same class (CS7600) and will be introduced to CS5600 this year.

The author is in the process of revising class and scribe notes into a form which may be used as a text for this class, either alone or in combination with another text. These materials will be made available under an open documentation license on the author's web page, where existing materials such as prior lecture videos and the pedagogical machine description may be found.

11. REFERENCES

[1] Bryant, R. E., and O'Hallaron, D. R. Introducing computer systems from a programmer's perspective. In *SIGCSE* (Charlotte, NC, 2001), ACM, pp. 90–94.

[2] Bryant, R. E., and O'Hallaron, D. R. *Computer Systems: A Programmer's Perspective*. Prentice Hall, Aug. 2002.

[3] Hovemeyer, D. GeekOS. http://geekos.sourceforge.net/, 2010.

[4] Hovemeyer, D., Hollingsworth, J. K., and Bhattacharjee, B. Running on the bare metal with GeekOS. In *SIGCSE* (Norfolk, VA, 2004), ACM.

[5] Lawton, K. P. Bochs: A portable PC emulator for Unix/X. *Linux J. 1996*, 29es (1996), 7.

[6] Levy, H. M., and Lipman, P. H. Virtual memory management in the VAX/VMS operating system. *Computer 15*, 3 (1982), 35–41.

[7] Nethercote, N., and Seward, J. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *SIGPLAN* (San Diego, CA, 2007).

[8] Paterson, T. An inside look at MS-DOS. *Byte Magazine 8*, 6 (1983), 230–252.

[9] Patt, Y. N., and Patel, S. *Introduction to Computing Systems: From Bits and Gates to C and Beyond*. Osborne/McGraw-Hill, 2000.

[10] Popek, G. J., and Goldberg, R. P. Formal requirements for virtualizable third generation architectures. *Commun. ACM 17*, 7 (1974), 412–421.

[11] Robin, J. S., and Irvine, C. E. Analysis of the intel pentium's ability to support a secure virtual machine monitor. In *USENIX Security Symposium* (Denver, Colorado, 2000), pp. 10–10.

[12] Silberschatz, A., Galvin, P. B., and Gagne, G. *Operating System Concepts*. Wiley, July 2008.

[13] Szeredi, M. FUSE: filesystem in userspace. http://fuse.sourceforge.net/, 2010.

[14] Thompson, K., and Ritchie, D. M. The UNIX timesharing system. *Communications of the ACM 17*, 7 (1974), 365–375.