System Calls and Interrupt Vectors in an Operating Systems Course[1]

Mark A. Holliday
Western Carolina University
Department of Mathematics and Computer Science
Cullowhee, NC, 28723
holliday@wcu.edu

Abstract

The introductory operating systems course has a tendency to appear to the student as a disparate collection of topics such as synchronization primitives, process scheduling algorithms, and page replacement policies. We describe a sequence of material to cover early in the operating systems course that prevents this tendency by clarifying the goal of the course and by providing a framework for understanding how the later course material is used in kernel design. The material centers around two concepts. First is the importance of the abstraction provided by the system call interface, that the kernel is the implementation of that interface, and the analogy with the instruction set interface the student has already encountered. Second is how the interrupt vector mechanism in a broad sense is central to how the kernel functions and underpins the actual implementation of many of the other topics in the course. Illustration through code from a real operating system kernel is a key feature of how this sequence makes clear the workings of an operating system.

1 Introduction

Students often enter their first operating systems course with anticipation. A key part of a computer system, a part that they encounter every day, they will finally understand. Unfortunately, too often the student leaves the course disappointed. A sizable number of synchronization primitives, process scheduling policies, and page replacement policies have been memorized. However, how the material fits together into a functioning operating system is often not clear.

Over many semesters of teaching the first operating systems course I have developed a sequence of material covered early in the course to address this problem. The course starts with two key concepts that establish the goal of the course and explain the basic machinery behind much of the functioning of the kernel.

The first concept is that the goal of the course is to understand the abstraction defined by the system call interface and the software, the kernel, that implements that abstraction. The second concept is that a very concrete, non-magical, sequence of events based on the interrupt vector mechanism is the underpinning for the operation of the kernel. We have found that illustration using code from a real operating system kernel is an important aid in the student's understanding these concepts. With these concepts understood, a framework has been provided which allows the remainder of the course to cover the traditional material more effectively.

Precedents for these concepts can be found in recent operating systems textbooks [2,3]. Also, a recent paper [1] describes a sequence of programming assignments for the operating systems course in which the first assignment involves the use of UNIX system calls. Our proposal is distinctive, however, in the emphasis that we place on these two concepts and the extent to which we integrate the system call and interrupt vector material. The extent to which we integrate the other material in the course within the framework of these two concepts and the use of real code in a coordinated manner to reinforce these two concepts are further distinctions of our approach.

In the next two Sections we describe the sequence of material being proposed. Section Two discusses the material of the first concept: system calls and the primary goal of the course. Once the system call interface is understood, there is a very natural organization of material that goes from the specific steps of executing a system call all the way through I/O devices. This development is outlined in Section Three as the second concept. Section Four reports on our experiences using this organization and how this material can be used to lead into the remainder of the course. We conclude in Section Five.

[1] This research is supported by the National Science Foundation under grant DUE-9650458.

SIGCSE '97 CA, USA
© 1997 ACM 0-89791-889-4/97/0002...$3.50

2 System Calls

The first concept states that the set of system calls is an interface of an abstract machine and the kernel is the software that implements that interface. The primary goal of the course is identified as understanding this interface's abstraction and the design issues in the kernel's implementation. Identifying a primary goal for the course is important. It is true, for example, that the concurrency ideas covered in an operating systems course are not used only in kernel design. Moreover, a first operating systems course usually (and should) discuss distributed systems to some extent. However, presenting those aspects of the course as adjuncts to the central goal of understanding how the kernel on a single processor works helps the student maintain focus later in the semester when a large number of topics are encountered.

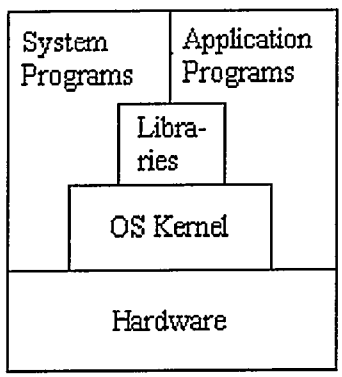

Figure 1: System Calls and Abstract Machines

Figure 1 is the illustration used when describing the computer at run-time as layers of abstract machines. Each layer has its own interface and implementation: machine language for the hardware, system calls for the kernel, and library calls for the libraries. That alternative implementations of the same interface (e.g. both the MINIX and Version 6 UNIX kernels are implementations of the Version 6 UNIX system call interface) are possible is noted. The step shape formed by the rectangles is intentional and shows that the code of either a system program or an application program can contain lines that directly use any of the three levels. For example, the source line **a = b + c;** will translate to a machine instruction that does not involve the kernel or a library. The source line **read(fd, buf, 10);** will translate to a machine instruction that calls a system call function. The source line **cout << a;** will translate to a machine instruction that will call a library function

A key part of this first concept is understanding the system call interface. One approach is to provide a textual description of some generic set of system calls. The alternative that I have found more effective follows our theme of using real code whenever possible. In particular we use examples from the MINIX operating system that was developed by Andrew Tanenbaum and described in his text *Operating Systems: Design and Implementation* [4]. MINIX is an implementation of the Version 6 UNIX system call interface that runs on PCs as the native operating system. The Version 6 Unix system call interface is a good choice since it contains most of the well-known UNIX system calls, but is not as large as the system call interfaces of later versions of UNIX.

2.1 Based on MINIX Examples

Part of the MINIX distribution includes a **tests** directory containing a set of short programs that exercise the system call interface. Tanenbaum included these programs in the distribution so that developers could check for problems after booting a rebuilt kernel. We use abbreviated versions of four of the tests to illustrate in a very concrete manner the version 6 UNIX system call interface. In the order that we present the programs to the students, the test programs are:

1. *Test 0.* Illustrate basic file manipulation by a single process. Cover the **open()**, **close()**, **creat()**, **read()**, **write()**, and **lseek()** system calls.
2. *Test 1.* Illustrate forking a child process with the **fork()** system call. The return value from the **fork()** call is used as the condition of an **if** statement to branch to the parent or child code both of which are in the initial process image. Also use the **pipe()**, **wait()**, and **exit()** system calls for communication between the parent and child processes.
3. *Test 2.* Illustrate use of the **execve()** system call for starting execution of another process image by the current process. Use in combination with the **fork()** system call so that the parent process waits and the child process calls **execve()**. Several versions of **execve()** are used to illustrate **argv[]** versus **envp[]** array passing. Also use the **link()** and **unlink()** system calls to illustrate that file descriptors are part of the environment passed to a forked child.
4. *Test 3.* Illustrate use of signaling with the **signal()**, **kill()**, **pause()** system calls. Use a signal handler and a forked child process.

The original test programs ran on MINIX, but, after trivial changes they have been made to run on SunOS, Ultrix, and Linux machines (and should run on any POSIX-compatible operating system). After being shown the working test programs, the students' understanding of the system call interface is reinforced by an assignment in which they write their own set of short programs that exercise the system calls. Five programs are

assigned (the assignments and solutions are available by contacting the author) as briefly described below.

1. Have a parent process create a pipe and fork a child process. Have the parent read and the child write from the pipe.
2. Have multiple uses of the **fork()** and **execve()** system calls with complex arguments passed using **argv**.
3. Demonstrate extensive file manipulation between a parent and forked child process.
4. Use the **time()**, **signal()**, and **pause()** system calls and a signal handler for delaying a process.
5. Show synchronization through extensive use of signals and signal handlers in both the parent and child processes including trying to ignore a signal using **SIG_IGN**.

Note the analogy with a computer architecture and organization course. The students spend a considerable amount of time in such a course learning example instruction sets and writing short programs that use that instruction set. How the instruction set is implemented is then studied. Similarly, in an operating systems course the students need to understand a concrete system call set and write short programs that use that system call set. How the system call set is implemented by the kernel is then studied.

2.2 Based on Man Pages

A disadvantage of providing the students with complete, working example programs from the MINIX **tests** directory is that they are fairly similar to the assignment programs. To make the assignment programs more challenging, for the last two course offerings instead of the example programs I provided the students with UNIX man pages for all the relevant system calls plus a handout that contained some program fragments. Those program fragments were 1) which header files to include, 2) an example use of **fork()**, 3) the syntax of the **signal()** system call, 4) a fragment showing a signal handler and the associated signal system call, 5) a fragment showing some of the variants of the **execve()** system call and the **argv[]** and **envp[]** arrays, and 6) the prototype for the function **main()** showing **argc**, **argv[]**, and **envp[]**. With this approach the students experience having to work from a system call interface specification.

3 Interrupt Vectors

The second concept can be expressed narrowly as: the kernel uses a single mechanism, the interrupt vector, at its lowest level for generating all exceptions including system call traps. The second concept can be expressed more broadly as: many of the features of an operating system (such as, system call execution, input/output, process scheduling, and virtual memory) have as a key step the use of interrupt vectors. This set of features follows a natural order. The start of that ordering is the events that occur upon a system call invocation since the student has just studied the system call interface. Again, realistic code from MINIX is shown and traced to clarify the events. The end of the ordering is various specific device drivers and their relationships with the rest of the kernel.

3.1 Libraries

The first step introduces libraries as an archive file of object modules of functions that can be linked to a user program. A discussion of **include** files, static linking, and dynamic linking arises. The next step demonstrates that a library can provide through its set of functions an interface that a user program can use through function calls. Thus, the concept of an *API (Application Programmers Interface)* is introduced. As an example of real code the standard I/O library that is in the library **libc.a** is discussed as an API above the I/O system calls. In particular, the header file for the standard I/O library, **stdio.h**, and the file **putc.c** containing the **putc()** function of the standard I/O library are shown. We note that the **putc()** code calls an I/O system call, **write()**.

Knowing that the functions above the system calls can be made into interfaces through libraries leads us to how the system call interface itself is defined through a library. In particular, each system call has a function by the same name in the library **libc.a**. Thus, at compile-time a system call use in a user program simply causes the function by the same name in the library **libc.a** to be linked. Understanding that the library mechanism is being used for both system calls and regular library functions is a key step in the student's understanding of how a user program and the kernel interact. Again, code from **libc.a** is used as reinforcement; the **libc.a** files, **close.c** and **exec.c**, for the **close()** and **execve()** system calls, respectively, are shown.

Next we consider the connection between the system call function in the library and control being passed to the kernel. The use of real code again is helpful. Execution is traced within **libc.a** from the **close.c** and **exec.c** files to a **call.c** file and then to a **sendrec.s** assembly language file that saves registers, calls the trap instruction, and then (after the return from the trap) restores the registers and returns to the **call.c** file. A point made is that the system call invocation behaves to the user program like a regular function call with control eventually returning all the way to the statement in the user program after the system call invocation.

3.2 Hardware Support

With attention focused on the trap instruction in the **sendrec.s** file we turn to the hardware support. The purposes of the Program Counter (PC) and Program Status Word (PSW) registers are reviewed. We reinforce that interrupt vectors exist in low memory and each vector has a PC and PSW value by examining the MINIX code for boot time interrupt vector initialization. Discussion of the PSW naturally leads to a discussion of user and supervisor protection modes, how the protection mode changes, and what additional access rights supervisor protection mode allows. The effect of the trap instruction---saving of the old PC and PSW register values and loading of the new register values from the interrupt vector associated with the trap instruction---is then identified as the general action of the interrupt vector mechanism.

That the new PC value is the address of an exception handler raises the distinction between the state saving done by the hardware and by the software. Examining the assembly language portion of the exception handler in MINIX shows the large amount of context saving (such as register saving) that must be done in software. Examining this code helps make the concept of a context switch more concrete and shows that the cost of a context switch is nontrivial. The assembly portion at the end of the exception handler that restores the initial state and returns from the interrupt is also noted to show the steps in returning control to the library.

3.3 Exceptions Taxonomy

Two natural questions arise at this point. First: what happens between the two assembly language parts of the exception handler for saving and restoring context? Second: what are the other causes of transfer of control through an interrupt vector? We address these questions by introducing a taxonomy of all the types of exceptions (see Figure 2).

Initially we distinguish between synchronous exceptions, or *traps*, that are internal to the processor and asynchronous exceptions, or *interrupts*, which are due to events external to the processor. Traps, in turn, are separated into those due to execution of the trap instruction and those due to an anomalous event. Such anomalous events in turn are separated into those that indicate a problem (such as a divide-by-zero error or an address translation generating a protection violation) and those that serve to notify the kernel of work that the hardware cannot do itself (such as an address translation generating a page fault).

The processing done by the kernel for traps is discussed briefly (reference is made to the deeper discussion later in the course). Trap instructions cause the requested system call actions to be performed. Anomalous events that are problems usually cause the process to terminate. Anomalous events that are requests for the kernel to do something cause branching to kernel code (such as to a page fault handler) to perform that work.

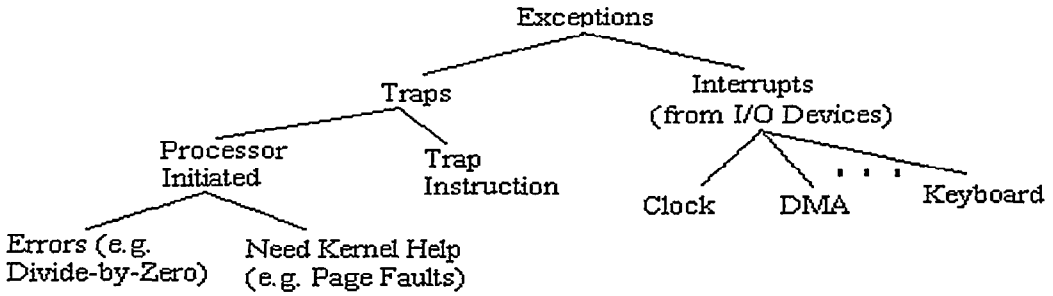

Figure 2: The Exception Taxonomy

The interrupt class of exceptions raises the subject of the input/output subsystem. A general discussion of I/O begins with I/O hardware including buses, devices, and controllers. The associated kernel software, especially interrupt handlers and device drivers, is identified. The code from the MINIX hard disk device driver illustrates the interaction between kernel device drivers and I/O controllers as well as the complex combination of programmed I/O and interrupt-driven I/O that is necessary in practical device drivers. Thus, the students see that the I/O device (actually its controller) interact with the processor and the kernel software at two levels. One level has the controller's interrupts interacting with the processor's interrupt vector mechanism and the kernel's exception handler. The second and higher level has the controller's data, control, and status registers interacting with the corresponding device driver in the kernel.

The in-depth coverage of interrupt handling that the students have seen helps motivate two I/O devices with specialized functions: the DMA device and the clock. The usefulness of DMA devices is explained in conjunction with the discussion of the disk device driver. The DMA device offloads the word-by-word interrupt processing of a disk block read or write from the processor. The clock simply generates periodic interrupts that can be used by the kernel for timing measurements and in particular for quantum expiration. Quantum expiration, in turn, is used for preemptive, short-term process scheduling. Learning that preemptive, short-term process scheduling can cause concurrency problems naturally leads from this section of the course to the later section of the course on concurrency.

4 Experiences

The sequence of material described in this paper evolved during teaching the operating systems course nine times over an eight years. In a semester-length course the first four weeks is sufficient time to cover the material. We now highlight some points found by experience.

1. Of all the system calls, students find the **fork()** system call the hardest, particularly, the idea that there are two returns with different return values from the system call. The assembly language for the fork system call has separate instructions for the system call invocation, the store of the return value, and the branch on the return value. Seeing this assembly language students often find helpful since the assembly language makes clear what actions are done after the fork.
2. Initial attempts at the assignment program on synchronization using signals can lead to the parent and child processes deadlocking. This deadlock can be used to motivate the deadlock problem when encountered later in the course.
3. The exceptions taxonomy helps motivate and integrate the material covered later in the course. Memory management is touched on when discussing address translation traps for protection violations and page faults. Both process scheduling and one source of concurrency problems are touched on when discussing clock interrupts, quantum expirations, and their use in preemptive process scheduling.
4. Illustrating the concepts discussed using carefully selected code from a real kernel is effective in making the material concrete. The next step in the student's learning involves programming assignments involving kernel modification. For that step our experience is that the best choice for the remainder of the course is to use a high-quality kernel simulator such as NACHOS [2]. The alternative of modifying the native kernel of the machine (such as MINIX) is unattractive since debugging is more difficult and the kernel complexity makes many interesting assignments beyond the reach of students in a first operating systems course.
5. An assembly language course precedes the operating systems course in our curriculum. I recently successfully tried moving the material described in this paper into the last four weeks of the earlier course. This provides a natural connection between the two courses and allows a more in-depth treatment of topics in the operating systems course.

Conclusions

It is important that the operating systems course have a well-defined focus, that it be clear how the different algorithms and policies discussed fit together into a kernel, and that the basic mechanisms underpinning the kernel be understood instead of being "magic". We have described a sequence of material that can be used during the first part of an operating systems course to ensure that these objectives are achieved. The sequence of material has been outlined and our experiences in using this organization have been discussed.

The outlined material is based on two concepts. First is the importance of the abstraction provided by the system call interface, that the kernel is the implementation of that interface, and the analogy with the instruction set interface the student has already encountered. Second is that the set of features centered about the interrupt vector mechanism underpins how the kernel functions and relates to many of the later topics in the course. Illustration through code from a real operating system kernel is a key feature of how this sequence makes clear the functioning of the operating system.

References

1. Ramakrishnan, S. and Lancaster, A-M. Operating System Projects: Linking Theory, Practice, and Use. *Proc. of the 24th SIGCSE Tech. Symp.* 24, 1 (Feb 1993).
2. Silberschatz, A. and Galvin, P.B. *Operating System Concepts, Fourth Edition*. Addison-Wesley, Reading, MA, 1994.
3. Tanenbaum, A. *Operating Systems: Design and Implementation*. Prentice-Hall, Englewood Cliffs, NJ, 1987.
4. Tanenbaum, A. *Modern Operating Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1992.