A Real Bottom-Up Operating Systems Course

Daniel P. Bovet*† Marco Cesati*‡

October 6, 2000

1 Introduction

The "Operating Systems" course is included in the Computer Science undergraduate program of almost all universities. It is quite surprising, however, that teachers rarely agree on what should be taught in such courses. Actually, there even isn't a widely agreed definition of what an operating system is.

One of the broadest definitions we ever seen has been given in [2]:

> *The operating system is the collection of functions which make available computing services to people who require them.*

Therefore, the operating system must fill the gap between the computer and the services required by the people using it. However, what kind of services should we consider? Isn't writing a document with a word processor a computing service that people require? And what about playing a strategy game, or browsing the Web?

Much more restricted definitions of operating system have been given. For example, in [6] we read:

> *The purpose of an operating system is to provide an environment in which a user can execute programs. The primary goal of an*

*Department of Computer Science, Systems, and Industrial Engineering, University of Rome "Tor Vergata", via di Tor Vergata 110, I-00133, Rome, Italy.

†E-mail: `bovet@uniroma2.it`

‡E-mail: `cesati@uniroma2.it`

operating system is thus to make the computer system convenient to use. A secondary goal is to use the computer hardware in an efficient manner.

Most Operating Systems courses implicitly adopt this definition or a similar one, because it allows teachers to stick to a well-defined set of canonical topics. Actually, two kinds of operating systems courses may co-exist: one addressed to system administrators and one to system programmers. In the first case, the course should teach how to make use of the many existing features inside an OS, how to tune it up properly, and how to set up proper environments for the users. In the second case, the course should explain how the functions are implemented and which are the main issues and trade-offs to be considered while implementing them.

In fact, the first kind of course is seldom included in an undergraduate Computer Science program: this is likely due to the fact that operating systems are quite different from each other, thus the experience gathered on one of them, say Sun Microsystem's Solaris, cannot be readily transfered to another one, say IBM's OS/390. We thus concentrate in the rest of this paper on the second type of course, the standard elective fourth year course of an Undergraduate program in Computer Science.

2 Basic approaches for teaching an OS course

Three basic approaches have been adopted so far:

- traditional course based on a valid textbook;
- use of simulators or toy operating systems that allow students to experiment and make changes;
- study the kernel of a real operating system.

A detailed analysis of the first approach has recently been made by [2], so we won't spend time discussing it.

The second approach deserves some additional comments. The venerable book by Madnick and Donovan [4] was likely the first one to include a simulator of a very primitive OS (it was a punched card simulator of a simplified version of IBM's OS/360). In some sense, also the book by Brinch

Hansen [3] included a simulated operating system (the description in concurrent Pascal allowed the simulator to be run on any system with a concurrent Pascal run-time environment). The Tanenbaum book [5] is, at the best of our knowledge, the only one that provides the source code for a true operating system, and not a simulated one (the toy OS called Minix has proved to be a very effective teaching tool: as it is well known, Linus Torvalds started working on Linux after taking a course based on Tanenbaum's book).

Until recently, the third approach has not been very popular. In fact, teaching a real operating system requires a laboratory where students can experiment changes to the kernel and reboot the system as many times as they want. Furthermore, an exhaustive documentation of the kernel and system utilities must be available to the students. With very few exceptions, the study of real operating systems like Unix has been confined to postgraduate courses and to research projects.

Things have changed dramatically in the last few years and several undergraduate Computer Science programs are now offering Operating System courses based on real case studies. We think that there are two main reasons for this change:

- Hardware is becoming cheaper and cheaper and most departments can offer laboratories with tens of powerful computers. Moreover, students usually own personal computers that can be used to work on the OS term projects.

- Several Open Source operating systems for personal computers are now available. They are fully fledged operating systems that often successfully compete with commercial ones.

3 The OS course at Rome "Tor Vergata"

We began planning an undergraduate course on Operating System at the end of 1996. Our students were at the third year of a five-year Computer Science degree offered by the School of Engineering at the University of Rome "Tor Vergata". The schedule included fourty 90-minute lectures and about ten lab sessions. Our objective was to teach a course based on the study of the source code of a real OS and to assign term projects consisting of making changes to the kernel and performing tests on the modified version. We were encouraged by the fact that PCs were becoming quite cheap and the source code of several operating systems was available at virtually no cost. By that time we were not aware of other similar courses so we had to start from scratch.

As a first step, we decided that the course should be focused on how the operating system exploits the hardware of the computer system: modern microprocessors include a lot of features that enable the OS to operate in an efficient and safe way. We thought thus crucial to teach how the OS builds on these hardware features in order to provide a friendly environment to the user. We also planned to cover all topics commonly found in operating system textbooks, even if in a non-traditional way.

The second step was to select the OS, among those available as Open Source, to be studied in classrooms. We selected Linux because it was already raising considerable interest among experts and because its "open group" philosophy matched quite well a university environment: for example, interested students could follow in USENET newsgroups the sometimes heated discussions among developers.

Unfortunately, suitable documentation was missing. A real life OS is hard to understand and students need plenty of information in order to grasp the general architecture and the finer details. In particular, we would like our students to know which are the main algorithms and data structures used by Linux, which are the assembly language and C functions that implement the algorithms and, last but not least, what kind of support does the IBM-compatible PC hardware offers to the OS. We thus decided to write lecture notes for our students.

A last decision was to design our course following a bottom-up approach. This is the most natural way for us to describe how hardware and operating system interact. However, ordering course topics following a bottom-up approach was not straightforward and required a lot of efforts and refinements over the past three years. We shall detail in the next section the existing dependencies among topics and we shall present our current preferred order.

The effort involved in carrying out our project was considerable. In the Spring Semester of 1997 we taught a course on Operating Systems based on Linux 2.0 and we prepared course notes for our students about a few critical features of Linux like task switching and task scheduling. We continued along this line in the following years moving on to the Linux 2.1 development version and later to the Linux 2.2 stable version. Starting from our lecture notes, which were becoming larger and larger, we wrote a whole book on the Linux kernel internals [1].

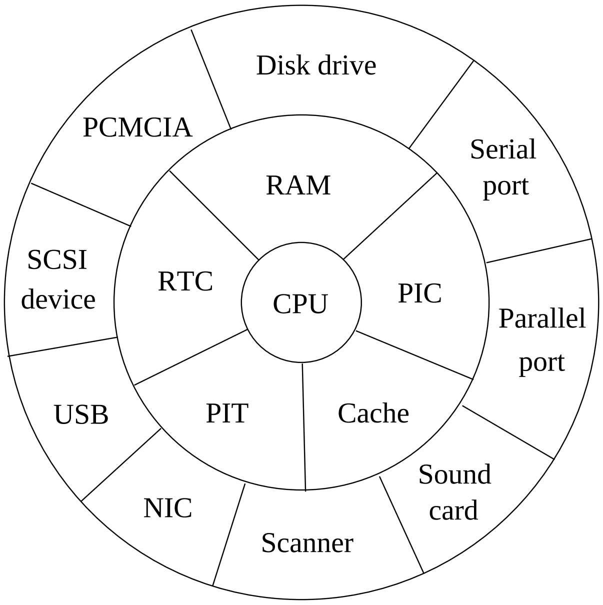

Figure 1: A layered lists of hardware devices inside a computer

4 Course organization

As stated earlier, our OS course should teach how an OS exploits computer's hardware. Obviously, the course cannot afford a detailed and in-depth discussion of all hardware devices included in modern computer systems. However, we select a few hardware devices among those always found in a PC and we teach how these devices are handled by the OS.

What hardware devices should we consider? Explaining how the kernel handles a sound card may be interesting, but it doesn't really teach anything fundamental about a modern operating system. Conversely, explaining how the kernel handles a microprocessor, the system RAM, or a disk drive gives insight to the internal structure of a modern operating system. In fact, some hardware devices are crucial (all computer systems have a CPU), while others are less important or even missing at all.

Figure 1 lists some hardware devices commonly found on a personal computer. The most important device is the CPU, which is thus placed in the center. Next we identify a few crucial devices, like the system dynamic memory (RAM), the high-speed static memory (hardware cache), the Programmable Interrupt Controller (PIC), the Programmable Interrupt Timer (PIT), or the Real Time Clock (RTC). These devices are very important, since they provide crucial services to the computer system. Finally, there are plenty of additional devices, like disk drives, serial and parallel ports, Netword Interface Cards (NIC), Universal Serial Bus controllers (USB), SCSI devices, and so on. While some of them may be quite important (e.g., disk drives), they are not really essential: a computer system can be built without them.

When planning our OS course we sticked to the following rule:

> *Course topics that are somewhat related to most important hardware devices (inner hardware devices in Figure 1) should precede course topics related to less important hardware devices (outer hardware devices in Figure 1).*

In some sense, we start from the center of the circle in the figure and we move towards the outer boundary. (By the way, our course doesn't cover all hardware devices shown in figure.)

A modern operating system kernel like Linux cannot be considered as a simple handler of the hardware devices included in the personal computer. The kernel creates and manages the multitasking environment in which system utilities and user applications run, it enforces system protections on processes and files, and so on. Thus, many course topics are not related to specific hardware devices, and we need a second general rule:

> *Any course topic X should be introduced after the topics whose comprehension is crucial to fully understand X.*

While apparently trivial, this rule is not so easy to apply. The kernel of a modern operating system is a highly-customized collection of programs, which mutually interact and cooperate. Dissecting a program or kernel function may not be so easier, and often circular dependencies among OS topics are difficult to avoid. From a practical point of view, it could be easier to stick to the following rule of thumb:

> *Course topics should be ordered in such a way to minimize the forward references to arguments yet to be explained.*

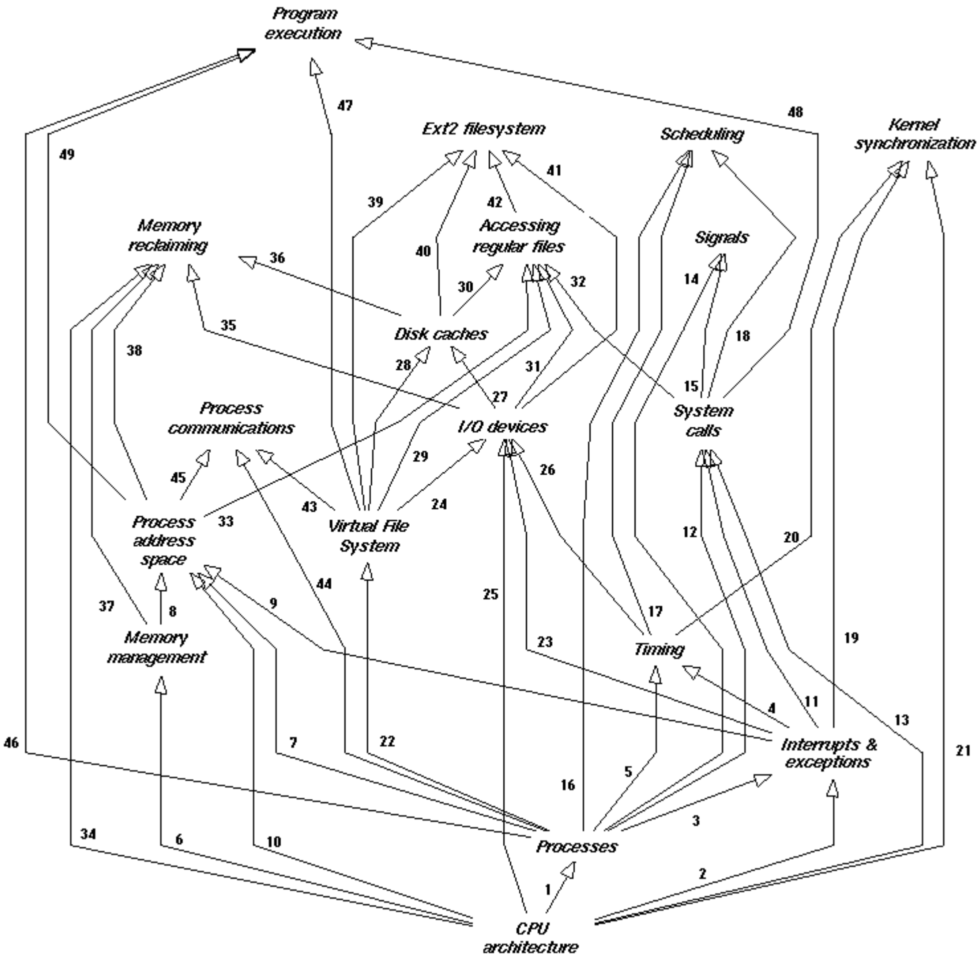

Figure 2: Main dependencies among OS course topics

Figure 2 illustrates the most important dependencies among the topics covered in our OS course. For instance, "CPU architecture" is introduced before "Processes", because we cannot fully explain what a process is without referring to notions like the execution context of a program, that is, the contents of the CPU registers and the memory locations used by the program.

Let us detail our current preferred order of the OS course topics. We briefly explain what arguments are covered, and the dependencies with previously introduced topics.

1: CPU architecture. We introduce the overall architecture of a microprocessor of the Intel 80x86 family, and we explain in detail how dynamic memory is addressed (segmentation and pagination). We also introduce the "User Mode" and "Kernel Mode" levels of execution.

Dependencies. nothing

2: Processes. We give a formal definition of process, and we illustrate how processes are handled by Intel microprocessors. In particular, we describe how Linux performs a context switch, that is, how it replaces a running process with another one.

Dependencies. (1) The process is a software abstraction, ultimately based on the notion of "execution context of a running program", so a clear understanding of the CPU architecture is required.

3: Interrupts and exceptions. We explain how a process may divert from its normal execution flow to execute high-priority functions handling hardware devices (interrupts) or coping with anomalous conditions (exceptions). We also discuss the role of the Programmable Interrupt Controller (PIC).

Dependencies. (2) Students must understand memory addressing and how CPUs keep track of the execution flow of a process. (3) The notion of process is also required, since interrupts and exceptions are handled in the execution context of the currently running process.

4: Timing. We describe how Linux exploits the features offered by the Real Time Clock (RTC) and by the Programmable Interrupt Timer (PIT) to keep track of elapsed time and to implement software timers.

Dependencies. (4) Time book-keeping is delegated to a special interrupt generated by the PIT, so students must know how interrupts are handled.

(5) Timing is also crucial for time-sharing, thus the notion of process should be clearly understood.

5: Memory management. We introduce the main algorithms used by Linux to manage the physical memory (RAM) in such a way to reduce memory fragmentation and to enhance system performances.

Dependencies. (5) Students must be aware of how the microprocessor handles the physical memory. Also, they must know how the static memory (hardware cache) works, so that they can understand what Linux does to fully make use of it.

6: Process address space. We describe how Linux associates an address space to each process in order to catch addressing errors of User Mode applications. We also explain how to allocate memory "on demand" and the "copy on write" technique.

Dependencies. (7) Of course, the notion of process is required. (8) Also, students must know how physical memory is effectively allocated to a process. (9) Exceptions should have been already introduced in order to explain how "allocation on demand" and "copy on write" work. (10) Finally, students should understand the memory access protection mechanisms offered by the CPU.

7: System calls. We explain how User Mode applications may require services to the kernel by means of system calls.

Dependencies. (11) Linux implements system calls for the PC architecture as program-triggered exceptions. (12) Students should clearly understand the process-kernel model. (13) Discussing how the kernel protects itself against buggy addresses passed as parameters of system calls also requires to understand the protection mechanisms offered by the CPU.

8: Signals. We introduce Unix signals and their implementation on Linux.

Dependencies. (14) Every signal is a sort of message sent to a specific process, thus the process notion is definitely required. (15) Students should already know what system calls are, so that the `kill()` system call can be easily introduced.

9: Scheduling. We discuss in detail the process scheduling algorithm of Linux.

Dependencies. (16) Of course, we cannot explain scheduling without having introduced the notion of process in advance. (17) Scheduling is related to the time-sharing mechanism; we must also teach how the time quantum of execution assigned to each process affects system performances. (18) Explaining how the scheduling algorithm can be tuned requires to introduce some system calls like `nice()`.

10: Kernel synchronization. We discuss all kind of concurrency problems and race conditions both for uniprocessor systems and multiprocessor ones, and we explain how the kernel defends itself against them.

Dependencies. (19) The most important source of potential data corruption inside the kernel is due to interrupt handlers, which are executed in a asynchronous fashion at unpredictable time instants. (20) Time-sharing may also cause data corruption, because several processes may end up using the very same system resource, thus acting on the same kernel data structure. (21) As usual, protection against race conditions is ultimately enforced by hardware features offered by the CPU. Moreover, students should understand the problems induced by per-processor hardware caches in multiprocessor systems.

11: Virtual File System. We describe the Virtual File System, which can be considered as an interface between the application's requests and the various filesystem types supported by Linux. Focus is placed on disk-based filesystems; notice that we left the inner layers of hardware devices in Figure 1, and we are moving towards the outer layer.

Dependencies. (22) In order to explain the role of the VFS, we refer to User Mode processes as actors that place requests to the kernel.

12: I/O devices. We give a general overview of how I/O hardware devices are handled by Linux. We discuss how the kernel may interact with them, and how it provides some special "device files" that act as communication channels between User Mode applications and the hardware devices. We also explain the special handling deserved by Linux to the block devices (hard disks).

Dependencies. (23) Students must clearly understand interrupts, because most hardware devices make extensive use of them. (24) In order to explain how device files work, we should have introduced the VFS in advance. (25) A few references to the internal CPU architecture are also required in order to explain how the kernel may grant to selected processes direct access to specific hardware devices. (26) Finally, most device drivers make use of kernel timers and other time-out mechanisms.

13: Disk caches. We discuss the algorithms that allow Linux to make use of dynamic memory in order to avoid slow accesses to the disks.

Dependencies. (27) Students must know why disk accesses are inherently slow, so disk devices should have already been introduced. (28) Moreover, most cached disk data belongs to disk files, so the VFS should have already been understood.

14: Regular files. We detail the kernel operations triggered by a User Mode application that is accessing a regular file on disk.

Dependencies. (29) Of course, the teacher should have already explained the role of the VFS as interface between the application and the low-level block device driver. (30) Students should have understood the role of the disk caches, as well as (31) how the physical disk drive is accessed. (32) Applications normally make use of system calls like `read()` and `write()` to access files; (33) however, files can also be memory mapped in the process address space.

15: Memory reclaiming. We describe several techniques adopted by the Linux kernel to preserve a minimal amount of free dynamic memory: swapping, flushing and shrinking of disk caches, and shrinking of memory caches.

Dependencies. (34) Swapping depends on a hardware feature of the CPU, that is, the possibility to mark some page of memory as "not present". (35) Swapping also concerns transferring pages of dynamic memory on disk devices. (36) It is obvious that disk caches should have already been introduced before explaining how Linux reduces their size in order to reclaim free memory. (37) Of course, students must be aware of how Linux manages free memory. (38) Finally, the swapping algorithm select a process to be penalized according to its memory usage, thus a clear understanding of the process address space is required.

16: Ext2 filesystem. We cover the implementation details of the standard Linux filesystem, considering both the disk data structures and the dynamic memory data structures.

Dependencies. (39) The VFS interface between the Ext2 filesystem and the application's file access requests must have already been understood. (40) References are also made to disk device drivers (41) and to the disk caches. (42) Finally, students will benefit from the previous discussion of how regular files are effectively accessed.

17: Process communications. Many kinds of inter-process communications are described: pipes, FIFOs, IPC messages, IPC semaphores, and IPC shared memory segments.

Dependencies. (43) Many inter-process communication mechanisms are file-based, so the VFS topic should have already been covered. (44) Of course, students must know what processes are. (45) Finally, explaining IPC shared memory segments requires a clear understanding of the process address space.

18: Program executions. We explain how a new program is executed, that is, what happens when a process invokes the `execve()` system call.

Dependencies. (46) Executing a program essentially means changing the execution context of a process. (47) Programs are stored in executable file, so students should know what a Unix file is. (48) Students should also understand what system calls are, because the discussion is focused on `execve()`. (49) When Linux rebuilds the execution context of a process for a new program, it releases the dynamic memory previously allocated to the process and builds a fresh, empty process address space.

5 Conclusion

Several classic Operating System textbooks claim to follow a bottom-up approach. As noticed in [2], a kind of average canonical order of course topics is the following: Processes, Scheduling, Concurrency, Memory Management, Deadlock, Input/Output and Devices, File Management, Multiprocessing and Distributed Systems, Protection and Security.

This approach works out quite well when presenting a "virtual" kernel that is hardware-independent. It doesn't work however when trying to describe a "real" kernel. The alternative course topics order proposed in this paper may be of interest to all those who want to plan an Operating Systems course based on the source code of a successfull, modern kernel.

References

[1] D. P. Bovet, M. Cesati. Understanding the Linux kernel. O'Reilly & Co., 2000.

[2] R. A. Creak, R. Sheehan. *A Top-Down Operating System Course*. Operating Systems Review 34 (3), ACM, July 2000, 69–80.

[3] P. Brinch Hansen. Operating System Principles. Prentice-Hall, 1973.

[4] S. E. Madvick, J. J. Donovan. Operating Systems. McGraw-Hill, 1974.

[5] A. S. Tanenbaum. Operating Systems—Design and Implementation. Prentice Hall, 1987.

[6] A. Silberschatz, P. B. Galvin. Operating System Concepts. Addison-Wesley, 4th edition, 1994.