Addressing Small Computers in the First OS Course

GARY NUTT
University of Colorado, Boulder

Small computers are emerging as important components of the contemporary computing scene. Their operating systems vary from specialized software for an embedded system to the same style of OS used on a generic desktop or server computer. This article describes a course in which systems are classified by their hardware capability and the traditional OS topic areas are then dissected, augmented, reassembled, and reorganized to illustrate the aspects of each topic as applicable to each class of small computers. Ultimately, the course covers the same material as a conventional OS course, but from a new perspective.

Categories and Subject Descriptors: D.4.0 [**Operating Systems**]: General
General Terms: Documentation
Additional Key Words and Phrases: Small computers, embedded systems, OS organization, dedicated systems, trusted processes, managed processes.

1. INTRODUCTION

There is a revolution in the field of computers, where small, communicating computers are becoming a pervasive part of our society (e.g., see MIT Project Oxygen [http://www.oxygen.lcs.mit.edu/H21.html]). Examples of small computers include notebook/laptop computers, mobile computers, palmtop computers, personal digital assistants (PDAs), set-top boxes, cell phones, wireless sensor network (WSN) nodes, and embedded systems. Fig. 1 shows an intuitive description of the space (i.e., this figure describes collections of computers of a labeled type by plotting their purpose (the y-axis) vs their physical size (the x-axis) of small computers, according to their purpose and physical size.

Commerce has driven the formation of small computer markets, creating several new application domains spanning finance (PDAs and cell phones), communication (cell phones and internet appliances), entertainment (consumer electronics and set-top boxes), defense (embedded systems and WSNs), and so on. Application domain characteristics typically limit small computer hardware resources such as CPU cycle times, memory, network bandwidth, wireless operation, and battery power. Domain requirements and hardware limitations can place unique constraints on the operating system (OS) for the small computer, causing designers to seek new resource management mechanisms and policies, and, in extreme cases, to entirely eliminate traditional OS functions such as file systems and virtual memory.

On the other hand, small computers sometimes require basic support for features that might be considered optional in mainstream operating systems. A significant fraction of

Author's address: Department of Computer Science, CB-430, University of Colorado, Boulder, CO 80309-0430.

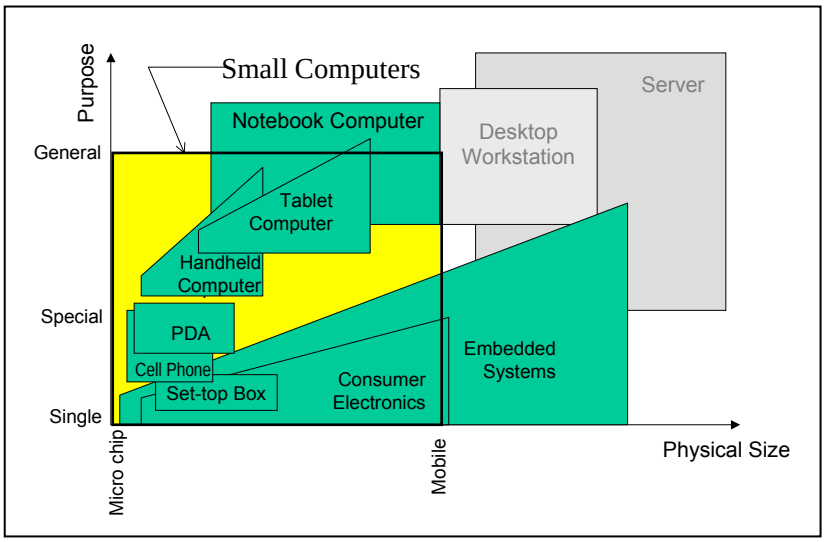

Fig. 1. Informal description of the small computers' size versus purpose.

the application domains make heavy use of distributed systems technology. At times this requirement is a result of the very purpose of the small computer (e.g., use as a node in a WSN [Culler et al. 2004]). At other times, the requirement is a result of the limited amount of storage that can be configured into the small computer (e.g., with a communicating PDA or cell phone). Small, communicating computers create a new set of criteria for trading off network bandwidth for local storage.

Another area where small computers may rely on functionality that is traditionally considered supplementary is in real-time support. For example, many domains require the small computer to playback/relay continuous media (audio and video streams), while others require the system to provide stimulus-response patterns for sensors and actuators as in classic embedded systems. Real-time system technology is most important when the system is running with a significant load; because small computers have limited resources, they often run at or near memory, CPU, and throughput capacities. Small computers do not generally have the reserve capacity to adequately handle real-time tasks with best-effort scheduling, which suggests that many small computers will need to incorporate soft or hard real-time scheduling techniques in order to meet application and domain demands.

The OS technology in contemporary small computers overlaps that used in conventional desktop operating systems such as Unix or Windows, and in embedded systems. Like desktop operating systems, these machines take advantage of multithreaded software, graphics, audio I/O, and network technology, but unlike desktop machines the OS operates in an environment of specialized devices and limited resources. Like embedded systems, small computers are also often limited by resource availability and require real-time responsiveness, but unlike embedded systems, the technology may focus on relatively general-purpose software rather than the hardware environment.

Operating systems for modern small computers are distinct from the mainstream OS designs as well as from classic embedded systems. Today, Linux is used as the operating system for enterprise servers at the same time that the same basic architecture is intended

to work for small computers [Embedded Linux Consortium 2005; Lehrbaum 2002]. There is also some skepticism that the fundamental Linux design is the best one for the smallest computers. Researchers have identified new OS technologies and approaches for various kinds of small computers, including set-top boxes [Furht 1995; Nelson et al. 1995], PDAs (e.g., see the commercial web pages for Palm Pilot and PocketPC), WSN nodes [Bhatti et al. 2005; Hill et al. 2000], and others.

Today's undergraduate operating system curriculum is focused on technology for desktop and server operating systems [IEEE and ACM 2001]. OS textbooks [Tanenbaum 2001; Nutt 2004; Silberschatz et al. 2005] are consistent with the Curriculum 2001 model and generally organize the main body of the material in four primary areas: process and resource management, memory management, input/output device management, and file management. Supplementary material is added in the areas of protection and security, multiprocessors, and other topics (as the textbook author sees fit). As long as the course focuses on operating system technology like that used in Unix and Windows NT, this organization works well, since, for example, these operating systems incorporate all (or nearly all) the contemporary technology related to processes, threads, objects, reusable resources, consumable resources, and so on. The student can immerse in process management and learn all this material, then consider how it is applicable to an OS like Unix or Windows NT, and then move on to the next major functionality.

However, if the instructor wishes to show how process management applies to an OS for a single-purpose (but multiple task) computer, such as an embedded system for greenhouse temperature control, then for process management it is only necessary to focus on cooperative scheduling techniques (like those in the Xerox Alto [Thacker et al. 1979]), without introducing the idea of a virtual machine. But if the instructor wants to describe how a small OS like TinyOS [Hill et al. 2000] is designed, then it is necessary to describe mandatory scheduling (timer interrupts), contexts and context switching, scheduling policy, and dispatching. However it is not necessary to describe processes, address spaces, protection barriers, and so on.

We have experimented with a reformulation of the material provided in a typical undergraduate OS course to make it better suited to the small computer perspective, and yet still cover the concepts used in a course for conventional general-purpose computers. Our reformulation is based on differences in hardware platforms used with different types of small computers.

In the remainder of this article we describe the system classification model and then the aspects of OS technology that are relevant to each class of system. Finally, we discuss our preliminary classroom experience using this organization to teach undergraduates about operating systems.

2. SYSTEM CLASSIFICATION

The system classification is based on the use of interrupts and the CPU mode bit in the supporting hardware. This section explains this taxonomy

2.1 Dedicated Systems

Early von Neumann computers did not incorporate interrupts nor a CPU mode bit. In effect, these computers only supported a single program-in-execution at a time. Work was divided into software modules (e.g., functions; the main program called different functions depending on the purpose of the program). There is a class of operating systems for dedicated computers (such as embedded computers) that do not need a more comprehensive computing model, or a more complex operating system; we refer to these systems as *dedicated systems* (DS).

One would expect to see DS technology in very small systems such as those used to control a sprinkler system, a microwave oven, a gasoline dispenser, or perhaps a point-of-sale terminal. These are single-threaded systems in which the single thread moves from part to part of the application, and from the application to the OS when system services are required. Some system designers prefer the simplicity of DS technology in cases where the CPU is a minor element of the system, but the behavior of the device is paramount. The DS approach is also preferred in cases where the system designer explicitly decides to avoid the inherent software complexity related to concurrency, that is, to the uncontrolled occurrence of interrupts that can add significant complexity to the software model.

2.2 Trusted Process Systems

Computer hardware evolved from the original von Neumann architecture when devices began to incorporate interrupts, thus enabling a radically new class of operating systems with CPU abstraction (multiprogramming) and gave rise to the idea of a classic process. In essence, this class of operating systems supports schedulable units of computation that are able to change any CPU register and access all information stored in executable memory. These classic processes share some similarity with a modern set of threads within a single process, in the sense that they are autonomous units of computation but there are few barriers among the processes. The second class of operating systems is built on this type of hardware; we use the historical term "process" to refer to this class as *trusted process* (TP) operating systems, since the correct behavior of the system depends on the correct behavior of each process.

The first generation of multiprogrammed computers are TP systems, which depend on interrupts, in particular on a timer device that periodically interrupts the CPU and gives control to the OS. The technology in this kind of system is radically different from DS systems, since the OS can control the hardware resources and thereby export an abstract machine (process) model. This class of OS introduces multiprogramming, concurrency, synchronization, scheduling, and so on. It is also a realistic representation of many small computer operating systems since it does not depend on the CPU mode bit. The immediate implication is that the OS can support the process model, but any process can read/write the address space of any other process. Ultimately the correct operation of the entire system depends on the safe operation of every abstract machine (process) in the system.

2.3 Managed Process Systems

The third class of OS is the basic type used in contemporary desktop and server computers, including Unix and Windows operating systems. These machines use the idea of a process execution environment accompanied by one or more threads of execution within the process. Since the activity in one process is explicitly prevented from interfering with the execution environment in other processes, we refer to this class of systems as *managed process* (MP) systems.

MP systems implement safe sharing among abstract machines. For example, they export an abstract machine, such as a process, and they enforce barriers (e.g., address space protection) among the abstract machines. Contemporary desktop, workstation, and server operating systems are MP operating systems.

The organization of the small computer course described in this article presents OS concepts according to the DS/TP/MP system characterization. That is, we first describe DS hardware and then discuss relevant OS technologies in this class of system. Next, we presume that the hardware incorporates devices with interrupts and then reconsider each

of the OS technologies for TP systems. In the new course organization we revisit/augment some technologies such as scheduling, but also introduce new ideas such as abstract machines in classes of systems capable of supporting the notion. When we finally consider MP systems, we are able to show how the previous technologies can be generalized for broader applicability. We are also able to emphasize new approaches and technologies that are possible in this class of systems, e.g., protection and security.

3. DEDICATED SYSTEMS

Dedicated systems are designed to solve one specific problem as cost-effectively as possible. In many dedicated systems the computing requirements can be satisfied with a very simple computer configured with an inexpensive CPU, limited memory, and a few devices. The computer is not required to support compilers, text editors, and other general software; instead, it executes a small, fixed repertoire of applications whose general behavior is well understood by the system designers. Figure 2 illustrates the general types of small computers that might use DS-style operating systems.

DS systems are designed as polling systems. Since the software does not depend on interrupts for context switching, the entire system effectively runs under the control of a *single thread* that executes both the OS and all of the application software.

Why would someone build a contemporary system that does not exploit interrupts? Systems that use interrupts are implicitly concurrent, multithreaded systems. When one (process or) thread is executing a program, the interrupt hardware transparently suspends the first thread of execution and then starts a new one to service the interrupt. The significance of the multithreading is the substantial jump in complexity (due to concurrency) to design and debug such systems: The software must address general, and sometimes very subtle, issues associated with the implied resource and/or information sharing between the two or more threads of execution. The cost of software development in modest systems that do not use interrupts can be orders of magnitude less than for systems which exploit them. If the purpose of the computer is to solve one specific problem, then the designer can avoid the complexity associated with multithreading (and

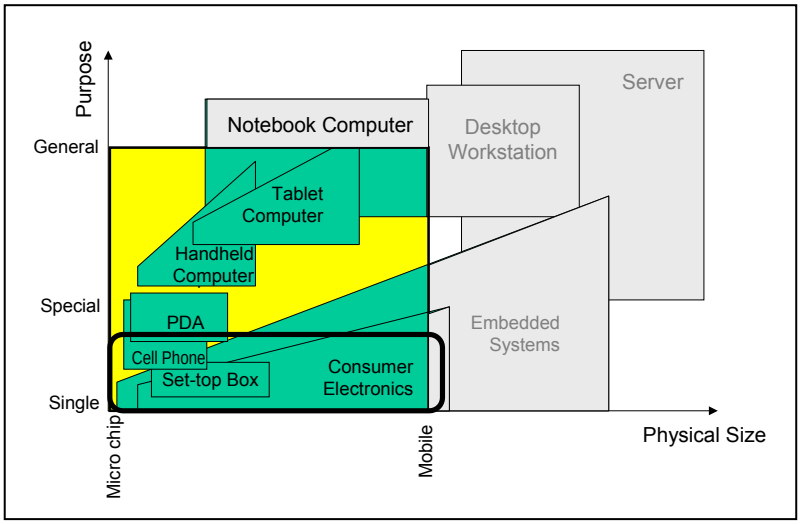

Fig. 2. Target hardware for a DS operating system.

thereby greatly reduce software development and maintenance costs) by tailoring the software environment so that it solves the particular problem with a single-threaded operation. Of course, as the number of independent tasks increases (i.e., the purpose of the computer is generalized from a small to a large number of tasks), interrupt-driven systems become more attractive.

In abstraction, DS software is organized as a collection of library procedures. As usual, library procedures are used to abstract complexity into a simple interface. In our course, we introduce the idea of coroutines so that we can approach the idea of semi-independent units of computation. Logically, a *coroutine* is a package for *autonomous units of computation* – the execution of a semi-independent task in the application programs (such as getting the next input value). Coroutines differ from procedures in that a coroutine can be invoked and then choose to suspend itself to *resume* a sibling coroutine. If a sibling coroutine resumes a suspended coroutine, then execution begins at the point at which the coroutine was last suspended. For example, coroutines are traditionally used to interpret format specifier lists for I/O functions (originally in Fortran format statements, but today to interpret the format specifier argument in the `printf()` and `scanf()` C library functions).

If a new, independent task (application task) is to be added to an existing DS, then a new coroutine is defined to implement the task. Next, the existing coroutines are modified so that the new coroutine is resumed by its new siblings; the new coroutine is responsible for resuming existing sibling coroutines.

Process management first focuses on the simple form of a coroutine system, where coroutines statically schedule themselves. It is then easy to generalize this idea so that the operating system keeps a process descriptor and assigns states to coroutines (ready, running, or blocked). At that point, we can introduce an anonymous scheduler coroutine; every coroutine resumes the anonymous scheduler rather than resuming a particular sibling. The scheduler can then use a policy for selecting among the competing coroutines to dispatch the next coroutine (using the resume primitive).

It works well pedagogically to describe a community of autonomous units of computation as a community of coroutines. Once students understand the concept of a coroutine, they immediately understand the consequence of manual scheduling, and are pleased to learn how a generic scheduler can be built (using conventional scheduler techniques).

By focusing on DS systems, we can also introduce elementary device management (without addressing interrupts). Students learn how the OS treats devices without seeing the complexity of asynchronous operation, top and bottom halves of drivers, and so on.

Classic memory management is introduced by focusing on how the translation system builds an address space in which the application will execute, how the loader(s) relocate addresses, and how dynamic address relocation can greatly simplify the relocation problem. It is possible for the student to clearly see the fundamental aspects of address spaces without being concerned about security and excessive levels of binding (as in virtual memory systems).

Few DS systems incorporate a file system, we, however, introduce file management as a DS system function, without addressing buffering. This perspective emphasizes that files are really just an abstraction of storage devices, without focusing on the performance required in systems that have multiple threads pounding on the storage system. Students learn the ideas of file descriptor, descriptor caching, block management, and directories. Later they will learn about file system performance enhancements, particularly buffering.

The DS summary emphasizes that it is possible to build a nontrivial OS in a limited hardware environment. It also identifies shortcomings due to the lack of independence among the coroutines, and the limited concurrency in the system – which provides a nice segue into TP systems.

4. TRUSTED PROCESS SYSTEMS

Trusted process systems are distinguished from DS systems by the fact that they are able to address multiple, independent problems at once – they are multipurpose rather than single-purpose machines (see Figure 3). TP systems are multiprogramming systems, where the autonomous units of computation – the executing programs – need not have knowledge of one another. TP programs make a great jump in abstraction when there *is* an operating system and *it* is in control of the hardware. As a consequence, the heart of the discussion in this phase of the course is process management.

When the computer is initialized, the OS is loaded and initialized; this creates a software environment composed of multiple abstract machines, each bearing a remarkable resemblance to the underlying physical machine (e.g, the TP abstract machine and the physical machine have the same instruction set, but different types and amounts of resources). A large part of the new material on process management deals with resources: the generalized notion of a resource is introduced, including the idea of abstracted resources (like files). Finally, the discussion addresses resource sharing and isolation.

The OS accepts requests to execute an application program, assigns each request to an abstract machine, and then systematically allocates physical machine components (such as executable memory and the CPU) to the abstract machines. When an abstract machine has been allocated the appropriate resources, it executes; otherwise it is blocked. Each abstract machine is classically ([this changes in modern systems) called a *process*, although we are sorely tempted to refer to them as threads, since they all operate in the same (hardware) address space.

The TP approach depends on the existence of an interval timer that periodically raises an interrupt. When a timer interrupt occurs, the currently executing abstract machine is

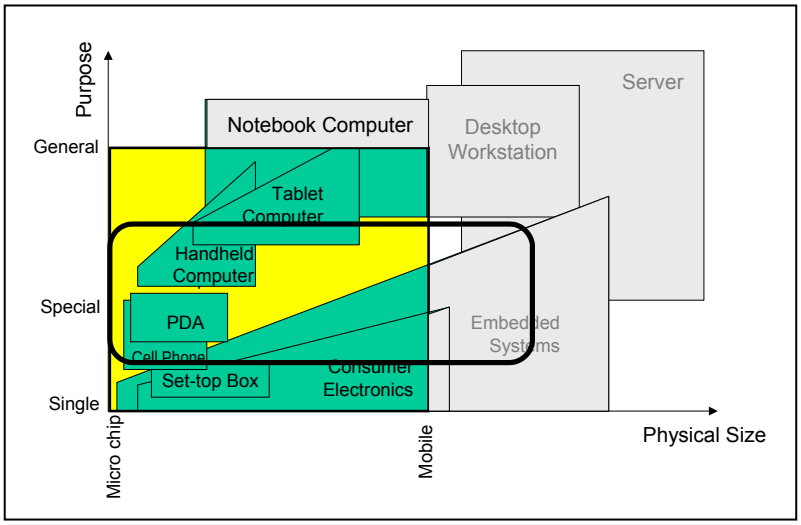

Fig. 3. Target hardware for TP operating systems.

suspended and the OS begins to execute. In particular, it can invoke the scheduler to dispatch an abstract machine according to its scheduling policy. The newly dispatched abstract machine executes until it voluntarily yields the CPU or another interval timer interrupt occurs.

Once the idea of a community of abstract machines has been explained, students can easily see how the individual programs need not incorporate design dependencies as required in a DS system. The instructor can then focus on new problems that arise as a result of this autonomy and its associated concurrency: scheduling, synchronization, and deadlock.

Due to the emphasis on small computers, and because many of the small computers deal with streaming media, it is natural to expand the traditional discussion of scheduling to introduce the ideas behind real-time scheduling. It is generally sufficient to describe periodic tasks, deadlines, service times, and earliest-deadline-first scheduling.

In the discussion of TP systems, device management is revisited and refined to explain how interrupts are managed. Now the full explanation of device driver top halves and interrupt handlers is provided. Once this concept is absorbed, students are ready to revisit the file manager to see how it can buffer ahead and behind in byte-stream file systems. This is also the time that the course can focus on performance enhancement based on CPU-device overlap.

Memory management is revisited next. It is now possible to talk about memory limit registers and the fact that they can raise an exception whenever the process attempts to reference memory outside the block allocated to it (as reflected by the base and limit register contents). This provides the perfect background for explaining how a modern memory manager allocates memory and then sets the base and limit registers once the memory has been allocated. Swapping is a natural topic for this part of the course. Astute students will notice that this is really not an insurmountable boundary among processes, since the process can simply reload the base and limit registers to access any memory it desires. That is, the correct operation of TP systems depends on the correctness of the individual application programs. A TP system can make it difficult, but not impossible, for an application to violate the interprocess protection barriers, which of course provides the justification for MP systems.

Within the class of TP systems, we can illustrate a broad range of systems, beginning with a system like TinyOS [Hill et al. 2000], and progressing to systems that implement more comprehensive scheduling (e.g., the Mantis OS [Bhatti et al. 2005]), and then on to systems that support the full process model (similar to Windows CE [Murray 1998]), but do so without relying on the CPU mode bit for interprocess barriers.

5. MANAGED PROCESS SYSTEMS

Managed process (MP) systems are contemporary, mainstream operating systems like Unix and Windows NT (Windows XP); see Figure 4. MP systems differ from TP systems in that they *enforce* sharing and cooperation policies, rather than relying on the cooperation of the application programs. So we say that software execution is *managed* to prevent it from violating the underlying abstract machine boundaries.

The difference in hardware for MP and TP systems is that MP systems require the presence of the CPU mode bit. This means that the CPU's instruction set is divided into a set of user instructions and a set of privileged instructions that can only be executed when the CPU is in supervisor mode. That is, the process abstract machine executes the user mode instructions and the OS kernel uses the privileged instructions. This is very significant because it enables MP operating systems to enforce barriers among processes.

Unless the OS explicitly allows it, no process can read or write the memory assigned to a different process.

Conventional undergraduate OS courses are organized for MP operating systems; that is, if one had no interest in DS and TP systems, one would simply use the conventional organization to describe process management, memory management, device management, and file management in some desired sequence. In the small computer course organization, this phase of the course adds the "missing materials" to the DS and TP discussion, thus completing the third pass through the traditional topics.

In this final phase of the course, it is important to revisit how the process manager implements the abstract machine such that only user space instructions and system calls are "instructions" executed by an application (user space) program. If traps were not covered when discussing interrupts in TP systems, they will need to be described in this phase to explain how user space applications are only able to access system functions through a well-defined interface. Even if they have been described earlier, this phase of study provides an essential motivation for why they are needed.

Protection and security are typically a supplemental topic in OS courses. In our present organization, they play a far more essential role – we cannot build an MP system without being concerned about security among process abstract machines. This provides a backdrop for discussing contemporary protection policies and security mechanisms. Within the OS, the CPU mode bit is the fundamental security mechanism used to implement a broad range of protection policies.

Finally, the course addresses virtual memory, which is typically not used in DS and TP systems. The student will learn about virtual memory in a context where protected mode operation is essential to the successful implementation of the idea.

6. DISCUSSION

Small computers have played an increasingly important role in the computing landscape over the last five years. Students are curious about the nature of the OS in their PDA, and many have heard that a TiVo DVD recorder uses Linux [Davis 1999; Knudson 2000].

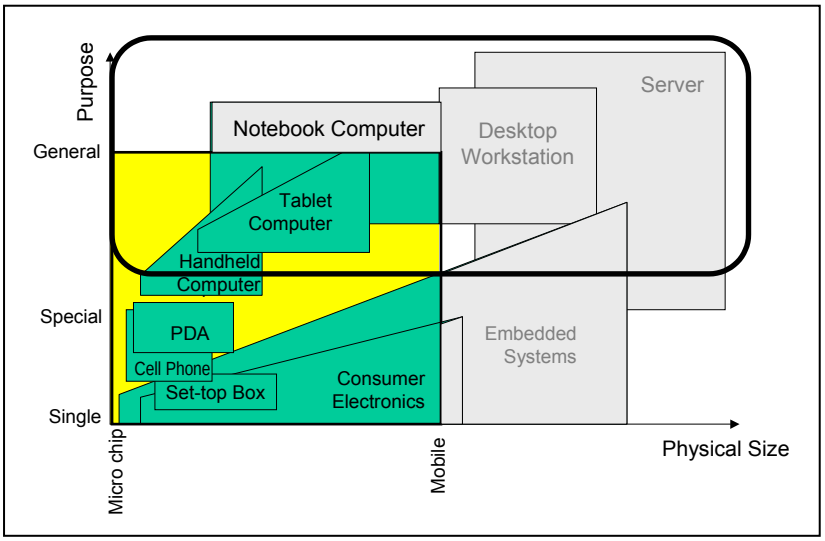

Fig. 4. Target hardware for MP operating systems.

When they learn about the OS for a Unix or Windows desktop computer, they can easily see that these principles do not really describe how their personal digital assistant OS works (how could it incorporate virtual memory if it doesn't even have a disk?).

6.1Topic Ordering

Textbook publishers frequently ask "What will the next generation operating system course look like?" There are many answers, ranging from "just like the current ones," to "they will have more object, multimedia, protection and security, SMP technology, <name_your_favorite_topic>," to "they will encompass operating systems for small computers." Many textbooks have continued to add topics to the traditional core of OS technology, and instructors add topics of their choosing also. As a result, OS courses have evolved to conform to the first two answers. This article illustrates a new approach, in which traditional topics are discussed in three different contexts, corresponding to three different classes of computer.

Traditional textbooks are organized to encourage the coverage of all of the four primary areas in operating systems sequentially (e.g., in the sequence shown in Figure 5).

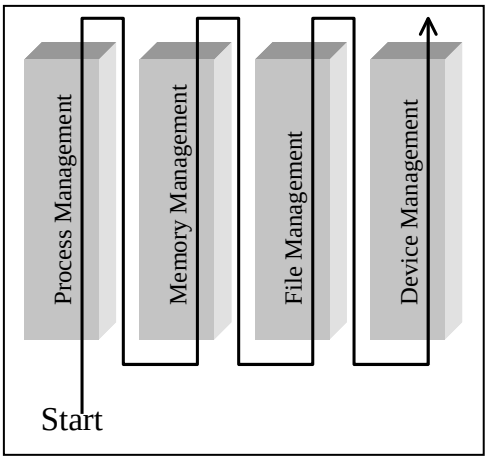

Fig. 5. Typical order of coverage for OS topics.

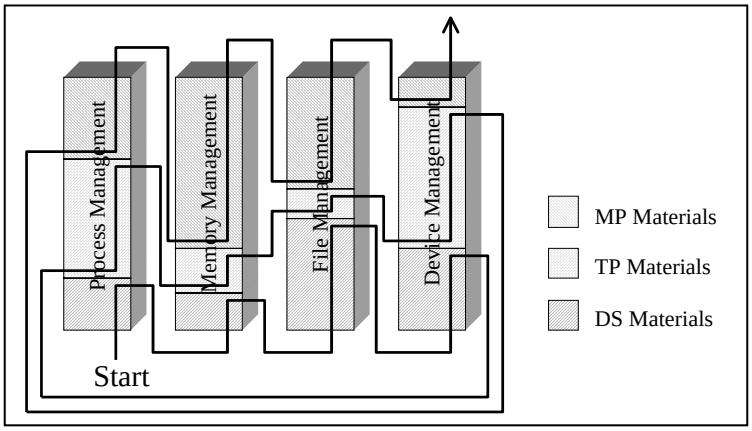

Fig. 6. Revised order of coverage for OS topics.

Teachers choose the order in which they present these four topics, although the textbook certainly influences the order in they way topics are described.

In the small computer OS course described here, each of the four topic areas are visited three times, once each for DS, TP, and MP systems (see Figure 6). Figure 6 is intended to roughly characterize the fraction of each column that is discussed in each phase of the course (the various fill patterns represent the three phases of the course). For example, in the first part of the course, each of the four areas is discussed as the topics apply to DS systems. In the DS discussion the process management portion of this phase focuses on coroutines and their management without introducing the notion of an abstract machine. In memory management, only the basic ideas of address spaces and dynamic relocation hardware are described for DS systems. On the other hand, it is possible to talk about many of the file manager concepts (e.g., except buffering) in the context of DS systems.

In the TP phase of the course, the bulk of the remainder of process management can be introduced, since the OS can now exploit interrupts. Specifically, the idea of an abstract machine is introduced, as well as diverse scheduling policies. We deferred the discussion of virtual memory to the MP phase of the course, since its correct operation depends on the CPU mode bit. There is not much material to add to file management during the TP phase of the course; and, of course, the topic of device management focuses on interrupt-driven I/O.

In the MP phase of the course, we finish process management by focusing on protection and security. For completeness, we introduced kernel threads during this phase of the course. Next we can describe paging virtual memory as it is used in contemporary operating systems such as Unix and Windows. The new file management material introduces the notion of a virtual file system switch (as is used in Linux) and introductory remarks about file servers. There is a little strip of device management in the MP phase to represent the task of ensuring that students understand how device management fits into MP systems.

A conventional first course in OS has a few topics that do not fit naturally fit in the primary four topic areas, e.g., an examination of deadlock theory, performance evaluation, design methodology, and so on. To the extent that an instructor has adequate time, these topics are covered as they are in the conventional course – they are added wherever the instructor likes and has time to address them.

6.2 Classroom Experience

We have used the organization described in this article in an undergraduate special topics course at the University of Colorado. Students who had previously taken the undergraduate OS course were not allowed to take this course for credit, as most of the material was the same; fifteen students enrolled in the course. The Faculty Course Questionnaire results were quite good compared to the instructor's results in the conventional undergraduate OS course (of course that was at least partly due to the size of the special class). Students who enrolled in the course were genuinely curious about small computers and were enthusiastic about the order of topic presentations.

Additional materials (typically not included in the undergraduate OS course) relating to distributed computing and real-time scheduling were included in the prototype offering. There was remarkably little time lost due to covering the primary topic areas (such as process management) in fragments. The additional material on distributed computing and real-time scheduling displaced some material that might be covered in

more depth in the conventional OS course (e.g., deadlock theory). However, we were able to at least address all the topics described in the conventional course.

The coroutine autonomous unit of computation model was new to most students, and they initially had some trouble seeing this as a realistic unit of computation. One alternative might be to replace this model with objects, although doing so would not reflect the reality of how this class of operating systems is designed and built. It is difficult to choose the formulation, since the corresponding model in commercial DS systems does not have a set of strong, underlying, computational model concepts.

There was general appreciation for the classification of systems and for the engineering tradeoffs among the classes of systems. Students seemed to quickly grasp the idea that the OS is really just software. An unanticipated benefit was that these students had a better appreciation of threads and kernel-space execution than is typical in the course that uses the conventional organization.

As in most operating system courses, the choice of homework assignments varies widely according to the teacher's goals. Most of the traditional analytic assignments apply to this course (e.g., analyzing synchronization solutions and scheduling algorithms, or predicting system performance). In terms of programming exercises, the class solved lab exercises similar to those in Nutt [2001; 2004]; relatively little adaptation was required for the small computer perspective. For example, students wrote a rudimentary shell program so they could grasp the fundamental notions of concurrency. Next, they wrote a simple TCP/IP program in order to understand the nature of distributed computing between a client and a server. In the first offering of the course, students solved simple Unix kernel exercises such as learning to write a new OS system call function.

In the next offering of the course, the exercises will be refined so that students use a hardware simulator to enable them to write programs to experiment with voluntary scheduling (like those used with coroutines). There are now many more platforms that can be used to support exercises for Embedded Linux (like the Crossbow Stargate computer); educational programs able to afford such platforms can offer true kernel exercises for small operating systems including Embedded Linux. Since the prototype course was first offered, a graduate student solved various kernel exercises, taken from Nutt [2001], on both a conventional Linux desktop system and on the Embedded Linux Stargate system. Some of these exercises will be phased into the small computer OS course.

7. CONCLUSION

We have created a course organization that emphasizes the differences between operating systems for small and large computers. First, we focus on DS operating system strategies in which sharing is accomplished through cooperative computing. We are able to introduce basic principles of process management, memory management, file management and device management without addressing the full complexity that accompanies multiple process-execution environments.

TP systems exploit interrupts in two especially important ways: interrupts allow the hardware to incorporate an interval timer that, in turn, enables the OS to *control* the machine. It also provides the necessary hardware to support full-fledged concurrency and sophisticated abstract machine models. The student revisits process, memory, file, and device management to understand the increased functionality that the OS can now provide, along with the associated cost in complexity required to create abstract machines and to manage their concurrent operation.

Finally, MP systems incorporate technology that prevents a process from performing operations that affect the operation of other processes. By learning about TP systems, the student will be familiar with the rationale that drove the evolution of address spaces, process boundaries, protection and security, and other abstractions that create a safe computing environment. This pass through the material completes the description of the OS technology that is used in contemporary desktop and server computers.

Our initial experience with the reorganized material is that the course takes an OS perspective that excites at least some faction of undergraduate computer science students. Nevertheless, it is feasible to address all of the same material that is in a conventional OS course (albeit in a little less depth in some cases), so that a student who takes an OS course with this organization understands the same concepts as a student who learned the same material in the conventional order. However, students in the new course also gained an additional understanding of the fundamental principles of operating systems for two emerging, highly relevant classes (DS and TP systems) of contemporary small computer hardware.

REFERENCES

IEEE AND ACM. 2001. *Computing Curriculum 2001: Computer Science Volume.* http://www.sigcse.org/cc2001/.

EMBEDDED LINUX CONSORTIUM. 2007. http://www.embedded-linux.org/.

MIT PROJECT OXYGEN. 2007. Pervasive, human-oriented computing. http://www.oxygen.lcs.mit.edu/H21.html.

BHATTI, S., CARLSON, J., DAI, H., DENG, J., ROSE, J., SHETH, A., SHUCKER, B., GRUENWALD, C., TORGERSON, A., AND HAN, R. 2005. MANTIS OS: An embedded multithreaded operating system for wireless micro sensor platforms. *Mobile Networks & Applications (MONET) 10*, 563-579.

BORKO, F., DEVEN, K., KITSON, F.L., RODRIGUEZ, A.A. AND WALL, W.E. 1995. Design issues for interactive television systems. *IEEE Computer 28*, 25-39.

CULLER, D., ESTRIN, D., AND SRIVASTAVA, M. 2004. Overview of sensor networks. *IEEE Computer* 37, 41-49.

DAVIS, G. 1999. Don't kill it after all: TiVo lets you control television. *LA Weekly 70*. http://www.laweekly.com/ink/99/29/cyber-davis.php.

HILL, J., SZEWCZYK, R., WOO, A., HOLLAR, S., CULLER, D., AND PISTER, K. 2000. System architecture directions for network sensors. In *Proceedings of the ASPLOS-IX Conference* (Cambridge, MA), ACM, New York, 93-104.

KNUDSON, C. 2000. Profile: TiVo. *Linux Journal 70*. http://www.linuxjournal.com/article/3740.

LEHRBAUM, R. 2002. Linux at the Embedded Systems Conference 2002. *Linux Journal 70*. http://www.linuxjournal.com/article/4730.

MURRAY, J. 1998. *Inside Microsoft Windows CE*. Microsoft Press, Redmond, WA.

NELSON, M., LINTON, M., AND OWICKI, S. 1995. A highly available, scalable ITV system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, ACM, New York, 54-67.

NUTT, G. 2001. *Kernel Projects for Linux*, Addison Wesley, Reading, MA.

NUTT, G. 2004. *Operating Systems*, Addison Wesley, Reading, MA.

SILBERSCHATZ, A., GALVIN, P.B., AND GAGNE, G. 2005. *Operating System Concepts*, 7th ed., Wiley, New York.

TANENBAUM,, A. S. 2001. *Modern Operating Systems*, 2nd ed., Prentice Hall, Upper Saddle River, NJ.

THACKER, C. P., MCCREIGH, E. M., LAMPSON, B. W., SPROULL, R. F., AND BOGGS, D. R. 1979. Alto: A personal computer. In *Computer Structures Readings and Examples*, 2nd ed., Sieworek et al. eds., McGraw-Hill, New York.

Received December 2005; revised August 2006; accepted August 2006