PortOS:
An Educational Operating System for the Post-PC Environment

Benjamin Atkin Emin Gün Sirer
Computer Science Department
Cornell University
Ithaca, NY 14853
{batkin, egs}@cs.cornell.edu

Abstract

In this paper, we describe PortOS, an educational operating system designed to complement undergraduate and graduate level classes on operating systems. PortOS is a complete user-level operating system project, with phases covering concurrency, synchronization, networking and file systems. It focuses particularly on ad hoc and peer-to-peer distributed computing on mobile devices. This paper discusses alternative approaches to operating system projects, and presents our particular design point along with pedagogical justifications.

1 Introduction

Operating systems form an integral part of the undergraduate and graduate curriculum at many institutions. A typical operating systems course serves many purposes, covering a range of material from system design principles and concurrent programming to implementation tradeoffs and software engineering for large systems. Operating systems classes frequently rely on a practicum project to span this large space.

In this paper, we describe PortOS, an educational operating system we developed to facilitate operating systems instruction. PortOS is comprised of six assignments that cover concurrency, networking and filesystems. PortOS ultimately culminates in a complete, user-level operating system which supports a GUI application that performs peer-to-peer messaging between mobile hosts without a central server.

This work is supported in part by ONR Grant N00014-01-1-0968. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organisations or of the Government.

The main motivation to develop a new system stemmed from a fundamental and quantifiable shift in the global computing landscape from desktop and centralized mainframe operating systems towards mobile, embedded and wireless systems [5]. Dubbed ubiquitous computing [13] or the Post-PC environment, this trend towards embedding computers in everyday devices shifts the focus of system development away from multi-user timesharing issues towards distributed systems. While the impact of these changes on the theoretical course material is small – after all, most OS courses cover networking and routing – the project component of the course should reflect the types of distributed computing challenges that the students would face in a highly mobile environment. In addition, a secondary motivation in re-examining existing operating systems projects was sheer necessity. Unlike most other educational tools, OS projects are usually highly system-dependent and need constant upkeep to evolve with changing platforms. We found that most of our students had Windows PCs at home, that our computing clusters were mostly composed of PCs, and that mobile handheld devices within our department ran Windows CE. We needed a project that would span these wireless, mobile hosts as well as support personal computers. Consequently, PortOS differs from other educational systems in use in that it focuses on distributed computing on mobile computers, and it supports Windows NT/98/2000/ME as well as the CE/PocketPC/Handheld PC 2000 platforms.

PortOS is a multi-phase operating systems project designed to accompany a traditional presentation of operating systems found in OS textbooks [8, 10, 12]. These texts provide some insight into implementation issues, but mostly focus on theory. PortOS complements them with subprojects covering threads, scheduling, unreliable datagrams, reliable streams, ad hoc routing and file systems. Students program in C against a realistic, but sanitized, machine model. They use a familiar integrated development and debugging environment, even though they are developing systems code. This paper presents the project, and the motivation behind it.

The rest of the paper is structured as follows. In the next section, we outline the guiding principles behind the system design and examine some alternatives. Section 3 describes the subprojects in further detail. We discuss related work on educational operating systems projects in section 4 and conclude in section 5.

2 The PortOS Platform

Three main principles guided the design of PortOS:

1. *Provide a sanitized but realistic environment*: The students should write operating systems code just as they would if they were implementing a native kernel. That is, the machine model, including atomic access primitives and interrupt model, should mimic an actual hardware implementation. At the same time, however, students should be shielded from complications and quirks found in real systems, such as time-dependent device behavior and complex bus signaling protocols.
2. *Support common platforms*: The post-PC environment is characterized by diverse, mobile clients. Each student in the class is provided with a PDA, and we want the students' projects to run seamlessly on PDAs and desktop clusters.
3. *Support familiar development environments*: Kernel programming should not necessitate learning an entirely different programming, debugging and build environment.

These three principles defined our choice of a language and platform for the PortOS project.

We decided to use the C language for PortOS assignments. The previous version of our class was based around a filesystem project written in Java. We decided not to use Java, even though this is the language of instruction for the introductory computer science class at our institution, because it is too high-level. That is, it provides too much of the functionality that we would like to have our students understand deeply and implement. In particular, we wanted the students to understand the implementation of threading, scheduling, synchronization and memory management services, which are provided by the Java runtime. A second problem with Java is that its lack of support for unsigned types and packed bit-fields makes it difficult to write networking code that can interact with existing, legacy services on the network. At the other extreme, we decided against exposing students to assembler because assembler programming is too low-level, which interferes with portability, makes modular programming difficult, and detracts from higher-level issues such as interface design.

Our choice of platform was also determined by the three principles of favoring sanitization, common platforms and familiar tools. Initially, we considered structuring the project as a set of extensions to Linux, NetBSD, FreeBSD or Windows NT source code. These systems are attractive because they are deployed, commercial-grade systems for which many applications exist and with which most students are familiar. But after closer inspection of the software environment in these kernels, we decided that they would be a poor platform for an educational course (though some of our students have gone on to apply their skills to open source development on Linux after our class). These systems have three problems in an educational setting. First, they often provide bad examples of the concepts taught in an OS class. For instance, most OS classes stress the importance and proper usage of universally studied synchronization primitives, such as semaphores and condition variables. Linux, BSD variants and Windows NT do not support either semaphores or condition variables in the kernel (this is not to be confused with the POSIX interface Linux and BSD variants provide to user applications; that interface is not available to in-kernel code on these platforms). In fact, all of these systems support what can at best be called "condimaphores," that is ad hoc synchronization primitives whose semantics are non-standard, obscure, and hard to explain [11]. In addition, lack of preemption among kernel threads makes these systems unsuitable for educational use. In essence, the entire kernel is arranged as a big monitor. Such a design may be suitable for a large, collaborative software project as it simplifies the programming model, but it is undesirable in a classroom setting as it will mask common synchronization errors students make [2]. Finally, a native kernel would expose the students to the vagaries of SCSI drivers, memory layout of VGA cards and other hardware quirks, which would detract from the fundamental course material.

Consequently, we decided to develop PortOS as a portable layer that resides on top of an existing commercial operating system. PortOS in turn provides a machine model to the systems programmer that closely mimics the underlying hardware, while virtualizing devices. Providing such an emulation layer on top of Windows NT and CE was not straightforward. Unlike the actual hardware, the Windows operating systems provide rigidly synchronous semantics for I/O events. Network packets, data blocks from the disk, timer interrupts and screen refresh interrupts are not taken on the stack of the currently executing thread – a thread has to explicitly wait and possibly block to perform these operations. The internals of PortOS are concerned with synthesizing interrupts that match the asynchronous interrupt model found on x86 and StrongARM processors. The students thus build a realistic system, on the emulated virtual hardware provided by the PortOS layer, rather than on a simulator.

3 The PortOS Project

The ultimate goal of the PortOS project is to build a prototype operating system that supports a peer-to-peer messaging application for mobile hosts in an ad hoc network. The project is comprised of six phases that build

up to this objective, which are designed to take two weeks each. The first four phases are intended to be completed in sequence, while the last two are independent of the previous phases, allowing them to be moved or omitted entirely. At each phase, we provide the students with a complete system image to build on, effectively providing a sample solution to the previous phase. Though students are encouraged to reuse their own solutions, in practice many have chosen to use our sample solutions in order to start with a bug-free "clean slate" in each new phase of the project.

In the following subsections, we describe the machine model PortOS provides and outline the subsystems students are expected to implement.

3.1 Phase 1: Threads and Synchronization

The first phase of the project introduces concurrent programming via a threads package which students implement. This phase also includes the implementation of semaphores for synchronization. We supply the students with code for context switching, initial stack setup, and atomic memory access (namely, **stack_initialize**, **stack_switch**, **test_and_set**, and **compare_and_swap**) to obviate the need for writing x86 and StrongARM assembly code. Figure 1 illustrates the interface students implement. For this phase of the project, thread switching is performed via a simple, round robin, non-preemptive scheduler. To test their work, the students implement a variant of the single producer, multiple consumer problem.

```
thread_t thread_fork(proc_t proc,
                     arg_t arg)
thread_t thread_create(proc_t proc,
                       arg_t arg)
void thread_stop()
void thread_start(thread_t t)
void thread_yield()
semaphore_t semaphore_create()
void semaphore_destroy(semaphore_t sem)
void semaphore_initialize(semaphore_t sem,
                          int count)
void semaphore_P(semaphore_t sem)
void semaphore_V(semaphore_t sem)
```

Figure 1. The interface students are expected to implement for Phase 1 includes support for threading and synchronization.

Overall, this phase of the project is quite traditional. An alternative approach is to ask students to write assembler code to start up a thread and perform context switching. We opted not to follow this strategy because it required assembler programming on two different platforms. In addition, concurrency is a complicated concept: it is hard to understand it without a context switch routine, and hard to implement a context switch routine without understanding it. We sidestep this dilemma by providing primitives in assembler.

3.2 Phase 2: Preemptive Scheduling

Applications which interact with the user need good response time. Phase 2 improves response time by extending the thread implementation from the first phase to use preemptive scheduling. Along the way, the students also replace the previous round-robin scheduler with a multi-level feedback queue. The new scheduler awards more, but shorter time slices to any process that routinely blocks before using up its allocated time slice. We provide a test program so that students can visually verify that their scheduler is working correctly, in the same spirit as [1], but with a textual interface.

3.3 Phase 3: Unreliable Networking

The third phase of the project introduces networking via an unreliable datagram service. The students implement a UDP-like protocol on top of a virtual network card PortOS provides. This phase of the project also introduces the concept of ports as communication endpoints, and requires proper synchronization and buffering to coordinate message delivery. Figure 2 describes the interface the students have to implement on top of the I/O interrupts they receive from the virtual network card. Since it incorporates a synchronous receive operation, the students must translate asynchronous packet arrivals into waking up the appropriate waiting thread via a packet classifier.

```
port_t port_local_create()
port_t port_remote_create(
                network_address_t addr,
                int id)
void port_destroy(port_t port)
int msg_send(port_t local, port_t remote,
             msg_t msg, int len)
int msg_receive(port_t local,
                port_t* remote,
                msg_t msg, int *len)
```

Figure 2. The network interface introduces ports as communication endpoints and unreliable datagrams.

3.4 Phase 4: Reliable Streams

Unreliable datagrams are not adequate as a general-purpose foundation for reliable distributed systems. The next phase of the project involves adding a reliable, TCP-like stream protocol. Our protocol is much simplified in comparison to TCP, by omitting handshakes, sender and receiver windows and other performance enhancements. But it captures the essential problem in reliable protocol design: how to deliver packets to a given destination despite transient failures within the network. Students need to implement acknowledgements, resends, and duplicate suppression. At most a single packet is ever required to be in transit between a sender-receiver pair of ports, which precludes a high data rate, but is easier to implement. The interface students implement for this assignment is identical to the

previous phase. To test the robustness of the students' code, the PortOS network card drops or duplicates packets with some probability to simulate failures encountered in wide area networks.

3.5 Phase 5: Ad hoc and Peer-to-Peer Applications

This phase gives students the opportunity to experiment with distributed computing in a wireless setting. In particular, this phase introduces ad hoc networking, where mobile nodes form self-organizing networks over wireless links. Since each node can communicate only with its nearby neighbors within a transmission radius, nodes must cooperate to find a suitable, multi-hop route to forward packets to their destinations.

Implementing projects based on ad hoc networks poses many challenges. Not only is the behavior of wireless links highly variable and hard to reproduce, but also it is impractical to test code with large numbers of mobile computers at once. PortOS simplifies these problems by emulating an ad hoc network within a wired cluster. This enables students to develop application code for handheld devices on the desktop. The main primitives provided by PortOS are unreliable broadcast and unicast functions. These deliver packets only to the neighboring nodes within range of the source node. A configurable network topology map informs the PortOS emulation layer of the network layout. Figure 3 shows an example of a network topology file that provides an adjacency matrix for the nodes comprising the network. PortOS also provides interfaces through which the network topology can be modified at run time to simulate host movement.

```
quark      .xx.x
proton     x.xx.
neutron    xx...
electron   .x..x
neutrino   x..x.
```

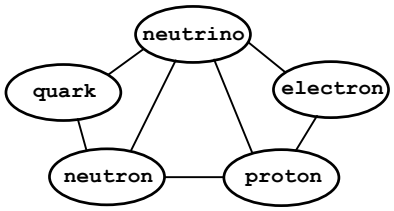

Figure 3. A topology file and the corresponding broadcast network.

The goal of this phase is to introduce self-organization in ad hoc networking in the context of a peer-to-peer application. Students need to develop an ad hoc routing algorithm [6, 9] to handle correct packet forwarding over multiple hops, and on top of the algorithm, implement a peer-to-peer instant messaging service. This requires that a text message typed in by one student and addressed to another is correctly and reliably routed and delivered over an ad hoc network, regardless of the location of the intended recipient.

3.6 Phase 6: File Systems

The final phase of the project introduces storage systems and atomic operations on secondary storage media. While it can be integrated with the instant messaging application to offer email services by saving messages onto the disk, it can also be done independently of phase five.

The interface implemented by the students is a Unix-style, multithreaded file system on top of a low-level disk emulator that PortOS provides. The disk emulator translates requests to read and write blocks into accesses to a regular file in the Windows NT file system, thus saving the students' data onto stable storage and providing continuity between sessions. The virtual disk interface provides a simplified block interface to secondary storage. This interface enables querying the disk block size, reading a given disk block and committing a block to the disk. On top of this interface, students build a traditional Unix filesystem for file manipulation, as shown in Figure 4.

```
file_t file_create(char *filename)
file_t file_open(char *filename,
                 int mode)
int file_read(file_t file, char *data,
              int maxlen)
int file_write(file_t file, char *data,
               int len)
int file_close(file_t file)
int file_unlink(char *filename)
int file_mkdir(char *dirname)
int file_rmdir(char *dirname)
```

Figure 4. The file system interface for phase six, a simplification of the traditional Unix systems calls.

To test the file system, we provide a shell incorporating a set of simple file utilities, such as **ls**, **mv** and **cat**. For the final project, some students implemented RAID storage and file systems based on UFS and LFS on top of the disk emulator. We are currently extending the emulator to reorder disk accesses on the fly and simulate system crashes in a repeatable manner.

3.7 Summary

While PortOS covers a broad spectrum of operating systems topics, the amount of code that the students have to write is modest. Table 1 summarizes the size of the code we provide for each phase of the project, as well typical figures for the amount of code the students write.

4 Related work

PortOS has been used as the project for our honors-level operating systems course. In this respect it is comparable to other instructional operating systems which have been described in the literature [3, 12]. We briefly elaborate the differences between PortOS and these systems, as well as other styles of operating system projects.

Component	Provided Code	Expected Code
PortOS Core	1071	-
Threads and synchronization	219	~600
Preemption	219	~400
Datagram networking	539	~300
Reliable stream networking	539	~400
Ad hoc routing	765	~1000
File-system	958	~1500
Testing code for all phases	1890	-

Table 1. Line counts for PortOS components. Students start with the PortOS Core, and are successively provided extra code in each of the phases. The final column lists the typical length of an adequate solution.

Our work is closest to Nachos [3] in form and content, but focuses more on advanced projects. PortOS enables real code to be linked to the kernel, rather than relying on cross-compilation and simulation, which require a special build, test and debug environment. Finally, it supports the Windows operating system family, including NT/98/2000/ME, but also CE/PocketPC and Handheld 2000.

As discussed in section 2, using a native kernel for studying operating systems is impractical because of the complexity and bad examples posed by a full-fledged system. An additional problem, shared by stand-alone operating systems, such as Minix [12], is that rewriting and debugging the kernel often requires a dedicated machine.

Alternative approaches to ours include building up from a raw machine simulator [4, 7], and using visualization tools to aid students in understanding and implementing operating systems algorithms [1].

5 Conclusions

PortOS is a new educational operating system construction framework for use in an undergraduate-level operating systems course. In addition to covering core material such as concurrency, scheduling, and storage systems, PortOS can serve as a platform for introducing the next generation of challenges posed by ubiquitous computing, ad hoc networks and mobile systems into the curriculum. PortOS runs on the Windows family of operating systems for both mobile and desktop hosts, and can be developed with standard development tools.

PortOS was developed and used at Cornell in spring 2001 as a practicum for an undergraduate class of 64 students. This earlier version of the software was refined by the addition of ad hoc networking and file system components, which are based on final projects written on top of PortOS by the students themselves. This experience has shown that that PortOS is a suitable platform for introducing mobility, issues, ad hoc networking, and ubiquitous computing into the undergraduate systems curriculum.

PortOS can be downloaded from

http://www.cs.cornell.edu/People/egs/portos/index.html

Acknowledgements

We would like to thank the 414/415 class of spring 2001 who had to work the bugs out of the first version of this course, and Sunny Gleason, for his help with the WinCE port. PortOS is based on the minithreads system, developed at CMU by Brian N. Bershad. We would also like to thank Hewlett-Packard and Microsoft for providing the infrastructure necessary to deploy mobile devices in our course.

References

[1] Michael Bedy, Steve Carr, Xianlong Huang, and Ching-Kuang Shene. A visualization system for multithreaded programming. In *Proceedings of the Thirty-First SIGCSE Technical Symposium on Computer Science Education*, pages 1-5, Austin, Texas, March 2000.

[2] Sung-Eun Choi and E Christopher Lewis, A Study of Common Pitfalls in Simple Multi-Threaded Programs. In *Proceedings of the Thirty-first ACM SIGCSE Technical Symposium on Computer Science Education*, March 2000.

[3] W. Christopher, S. Procter, and T. Anderson. The Nachos instructional operating system. In *Proceedings of the 1993 Winter USENIX Conference*, pages 479-488, January 1993.

[4] John Dickinson. Operating systems projects built on a simple hardware simulator. In *Proceedings of the Thirty-First SIGCSE Technical Symposium on Computer Science Education*, pages 320-324, Austin, Texas, March 2000.

[5] J. Hennessy. The Future of Systems Research. In *IEEE Computer*, pages 27-33, August 1999.

[6] David B. Johnson and David A. Maltz. Dynamic Source Routing in Ad Hoc Wireless Networks. In Mobile Computing, edited by Tomasz Imielinski and Hank Korth, Chapter 5, pages 153-181, Kluwer Academic Publishers, 1996.

[7] Mauro Morsiani and Renzo Davoli. Learning operating systems structure and implementation through the MPS computer system simulator. In *Proceedings of the Thirtieth SIGCSE Technical Symposium on Computer Science Education*, pages 63-67, New Orleans, Louisiana, March 1999.

[8] Gary Nutt. Operating Systems: A Modern Perspective. Addison Wesley Longman, 2000.

[9] Charles E. Perkins and Elizabeth M. Royer. Ad hoc On-Demand Distance Vector Routing. Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications, New Orleans, LA, February 1999, pp. 90-100.

[10] Abraham Silberschatz and Peter Galvin. Operating System Concepts. John Wiley and Sons, fifth edition, 1997.

[11] David A. Solomon, Inside Windows NT. Microsoft Press, second edition, 1998.

[12] Andrew S. Tanenbaum and Albert S. Woodhull. Operating Systems: Design and Implementation. Prentice Hall, second edition, 1997.

[13] M. Weiser. Some Computer Science Problems in Ubiquitous Computing. In *Communications of the ACM*, July 1993.