Symmetric MultiProcessing for the Pintos Instructional Operating System

Lance Chao

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science & Application

Godmar V Back, Chair

Ali Butt

Dennis G Kafura

May 9, 2017

Blacksburg, Virginia

Keywords: Pintos, instructional kernel, instructional operating systems, SMP

Copyright 2017, Lance Chao

Symmetric MultiProcessing for the Pintos Instructional Operating System

Lance Chao

(ABSTRACT)

For the last decade, practical limitations have prevented processor speeds from increasing significantly. To increase throughput, the computing industry has turned to multiprocessing; that is, executing computations in parallel on separate processing units. Making use of these additional units requires support from the operating system (OS). Indeed, most modern operating systems do have the capability of recognizing and utilizing multiprocessor hardware.

Pintos is an instructional operating system used by many institutions to teach important operating systems concepts. Pintos aims to increase student engagement by providing challenging programming projects in which students personally implement many core functionalities of an operating system. However, prior to this work, Pintos was a uniprocessor OS. This makes it difficult for Pintos to expose students to the same synchronization challenges that most modern kernel developers face. In addition, the first structured project, aimed at teaching scheduling policies, requires students to implement an uniprocessor variant of MLFQS scheduler which is no longer used in modern systems.

We implemented Symmetric MultiProcessing (SMP) support in Pintos. We also created a new scheduling assignment to expose students to a multiprocessor proportional-share scheduling policy called Completely Fair Scheduler and to introduce them to the concept of load balancing. Finally, we evaluate the effectiveness of our new Pintos framework in augmenting students' knowledge of OS scheduling and enhancing their ability to code and debug in a low-level environment.

Symmetric MultiProcessing for the Pintos Instructional Operating System

Lance Chao

(GENERAL AUDIENCE ABSTRACT)

Operating system education remains a cornerstone of any undergraduate computer science curriculum. Instructional operating systems provide the necessary infrastructure to increase student engagement by allowing students to learn through challenging, hands-on projects. We present PintOS/SMP, an instructional operating system we built.

PintOS/SMP is based on the existing PintOS operating system that has been in use at Virginia Tech and other institutions for several years. PintOS, however, was not a multiprocessor operating system, which meant that it was unable to support additional execution units provided by the underlying hardware. Thus, it lacks realism in an era in which even smartphones are delivered with multiple execution units. We added multiprocessor support to PintOS in order to introduce students to the challenges that multiprocessor systems brought to OS developers. We also developed a new programming assignment in order to expose students to the techniques used to distribute work efficiently in multiprocessor systems.

We deployed PintOS/SMP in a capstone class with 23 students and evaluated it using a survey instrument. Students reported that the projects were useful and interesting and significantly enhanced their level of understanding of operating system concepts, which we confirmed through the use of test questions. Our results indicate that PintOS/SMP provides a challenging but enjoyable learning experience and is successful in reaching its educational goals.

Acknowledgments

First, I would like to thank my primary adviser, Dr. Godmar Back, who supported me throughout my studies both as an undergraduate and as a graduate student. He first exposed me to Pintos in a graduate Operating Systems course. He was very patient in answering my questions and critiquing my implementation of the structured projects, which immensely increased my ability to design and implement a stable and productive system. During my time as a graduate student, he met with me for almost two hours every week to help me improve my design, clear any roadblocks I ran into, and pointing me in the right direction whenever I felt lost. He also personally made some important tweaks to the Pintos base code and documentation during the final stages of Pintos development that made the new SMP framework much better when it was finally in production. He also supported me while I pursued some of my other interests, such as cybersecurity wargaming and competitive programming. Finally, he was able to find me a GRA position, which supported me financially throughout my final year at Virginia Tech, as well giving me experience with web application development. I felt extremely inspired by his passion towards computer science, his willingness to spend large amounts of his own time to helping his students, his drive to

learn more, to improve existing systems, to produce works of the highest quality, to avoid shortcuts and build systems correctly from the ground up. He has had a positive influence on not only me, but everyone who has had the pleasure of studying and working with him.

I would also like to thank Dr. Ali Butt, who was my professor for Computer Systems and the Systems and Networking Capstone. I really enjoyed his lectures and he taught me many systems fundamentals that later became invaluable to me when developing Pintos. He was also very supportive of me when I decided to work by myself instead of with a group as typically required by those classes.

I would also like to thank William Mcquain, who was my professor for Computer Organization (I & II) and Data Structures & Algorithms, and whom I worked with many times as a Teaching Assistant for the Computer Systems course. He was very patient with answering my questions during office hours when I had trouble understanding even basic concepts. He is also one of the most effective lecturers that I have met, and under his guidance I was able to quickly develop my understanding of the C programming language, debugging, memory management, computer architecture, and other fundamentals that I could have not finished my thesis without.

I would like to thank Dennis Kafura, with whom I took an Instructional Design class with. His class helped me design and evaluate the educational component of my project.

Last but not least, I would like to thank the entire Computer Science department at Virginia Tech and everyone else who helped get me to where I am today, including my past professors,

advisers, teaching assistants, and other students.

Contents

1 Introduction **1**

1.1 Pintos 2

1.2 Motivation 4

1.3 Core Contributions 4

1.4 Roadmap 6

2 Background **7**

2.1 Process management 7

2.1.1 Processes 7

2.1.2 Threads 9

2.2 Scheduling and Context Switching 9

2.2.1 Preemptive vs Non-preemptive Scheduling 9

2.2.2 Proportional-share Scheduler 10

2.2.3 Multi-CPU Scheduling 12

2.3 Virtual Memory 13

2.4 Interrupts 14

2.4.1 Exceptions 14

2.4.2 Hardware Interrupts 15

2.4.3 Software Interrupts 15

2.5 Managing Concurrency 16

2.5.1 Data Races 16

2.5.2 Atomic Instructions 17

2.5.3 Critical Sections 18

2.5.4 Atomicity Violations 18

2.5.5 Locks and Mutual Exclusion 19

2.5.6 Spinlocks 20

2.5.7 Deadlocks 20

3 Pintos 2.0 Overview **21**

3.1 Basecode 22

3.1.1 Thread Management 23

3.1.2 Memory Management 23

3.1.3 Managing I/O 25

3.1.4 Filesystem 26

3.2 Student Projects 27

3.2.1 Project 1: Threads 27

3.2.2 Project 2: User Programs 27

3.2.3 Project 3: Virtual Memory 28

3.2.4 Project 4: Filesystem 28

3.3 Running and Debugging Pintos 29

4 Implementation **31**

4.1 Interrupt Management 31

4.1.1 Per-CPU Timer Interrupts 32

4.1.2 Advanced PIC for I/O Interrupts 33

4.1.3 Configuring the APIC 33

4.2 Detection and Bootstrapping of Additional Processors 34

4.3 Synchronization in SMP 35

4.3.1 Sources of Concurrency 35

4.3.2 Managing Concurrency 36

4.3.3 Designing Synchronization Mechanisms 37

4.4 Spinlock Implementation 38

4.5 Extending the Base Scheduler to SMP 39

4.6 Thread Block and Unblock 40

4.7 TLB Shootdown 42

4.8 Atomic Instructions 43

5 New Scheduling Project **45**

5.1 Alarm Clock 45

5.2 Simplified Completely Fair Scheduler 46

5.2.1 Overview 46

5.2.2 Detailed Requirements 47

5.2.3 Differences from Linux 54

5.2.4 Tests 55

5.3 Load Balancer 58

5.3.1 Overview 58

5.3.2 Detailed Requirements 58
5.3.3 Differences from Linux 60
5.3.4 Tests 60

6 Evaluation **63**

6.1 Test Cases 64
6.2 Design Documents 64
6.3 Post-Survey 66
6.3.1 Student Feedback 67
6.3.2 Student Learning Evaluation 68
6.3.3 Student Comments/Suggestions 69
6.4 Interactions with Students 70
6.5 Limitations of our Evaluation Methodology 71

7 Related Work **73**

8 Future Work **77**

9 Conclusion **80**

Bibliography **82**

List of Figures

2.1 Process states. 8

2.2 Incrementing a shared counter 17

2.3 An atomicity violation 19

3.1 A graphical overview of Pintos. 22

3.2 Running Pintos. 30

4.1 The advanced programmable interrupt controller 32

4.2 Sources of concurrency in UP and SMP Pintos 36

4.3 Synchronization mechanisms 37

4.4 Relationship between synchronization mechanisms. 38

4.5 Thread block and unblock in Pintos/SMP. 42

5.1 Mapping of process nice values 48

5.2 Atomicity violation caused by load balancing 62

6.1 Student grades from automatic test cases 65

6.2 Scores received by students on their design documents 66

6.3 Students' feedback about Project 1 67

6.4 Student's learning about multiprocessor operating systems 68

6.5 Student's learning about OS scheduling 69

6.6 Student's learning about load balancing 70

Chapter 1

Introduction

An operating system (OS) is the bridge that connects user software to the hardware on which it runs. It manages system resources, such as CPU, memory, storage, and peripheral devices attached to the system, and provides access to these resources through a simple interface. A deeper understanding of how operating systems manage these resources helps programmers identify areas of contention and design scalable application programs. In addition, designing and implementing operating systems involves making design choices and overcoming challenges that are broadly applicable to other types of systems as well. For these reasons, OS education remains a cornerstone of undergraduate computer science education [26].

Approaches to teaching OS typically fall along two axes: whether to treat the material as abstract or concrete and whether to teach OS from an internal or external perspective [49]. On one axis, the abstract approach discusses OS from a conceptual point of view,

discussing the algorithms and techniques used in modern operating systems to expose the hardware to user processes. A concrete approach, on the other hand, puts an emphasis on the design and implementation of real operating systems software. On the second axis, an internal perspective considers the system from the point of view of an OS designer or developer trying to provide services to higher level programs, whereas an external perspective considers the OS from the point of view of a user or programmer working directly with OS facilities.

1.1 Pintos

Here at Virginia Tech, we offer an advanced systems course that teaches OS with a concrete, internal approach using Pintos [44]. Pintos is a simple educational operating system framework originally developed at Stanford University for the x86 architecture. It supports kernel threads, loading and running user programs, and reading/writing to I/O devices, but it implements all of these in a very simple way. It provides a well-designed and commented code base, as well as a manual which provides a high-level documentation of Pintos as well as specifications for each of four structured projects that are included. Students build core functionality on top of the given code base, which results in a simple but functional operating system. Unlike other educational OS frameworks that target simulated hardware [11, 21], Pintos is designed to run on the dominant x86 ISA. We believe this provides more realism and increases student engagement [44]. Pintos projects are designed to be completed

by groups of 2-4 students. Working in a team provides an environment closely resembling modern industry software development, and introduces students to the challenges of working collaboratively on a relatively large project, including the use of distributed version control systems such as Git.

The first project teaches OS scheduling and thread state management. Students are required to implement a simplified version of Linux's Completely Fair Scheduler (CFS) [43] and its associated load balancing algorithm. The second project explores user programs by illustrating how user processes are protected and isolated from each other via virtual address spaces, and how they interact with the kernel via system calls. The third project focuses on virtual memory, where the students are asked to implement a mechanism that allows user processes to access more memory than is available to the system (also known as paging). The last project involves the file system, where students implement a hierarchical, Unix-style file system to replace the simple one provided by the base code.

Pintos, like most modern operating systems, is a preemptive kernel, which means that threads executing in the kernel can be preempted at any time; furthermore, in an SMP system, additional threads could be executing on other CPUs. Students must ensure that shared data structures are not accessed concurrently by different threads. To achieve this, they must apply explicit synchronization techniques to manage concurrency and avoid possible race conditions, thereby gaining experience programming in multi-threaded environments. These techniques include the use of synchronization constructs such as locks and semaphores to protect critical sections and ensure program order. The same techniques are also often

used for the same purpose in userspace programs; therefore, the experience gathered here is valuable for not only kernel hacking, but also a variety of other programming scenarios.

1.2 Motivation

Pintos is designed to provide a code base that resembles that is used in real production scenarios, yet is not so complicated that it stunts student learning. However, Pintos is a uniprocessor OS, making it unable to support additional execution units provided by the underlying hardware. When Pintos was first developed, multiprocessor hardware was not widely available. Nowadays, most general-purpose OS must include multiprocessor support since even hand-held devices such as smartphones are increasingly built with multiple on-chip processing units. As a uniprocessor OS, Pintos is unable to present the challenges that multiprocessor technology brought to OS developers. This limitation motivated us to introduce SMP support in Pintos.

1.3 Core Contributions

Our work includes three core contributions.

First, we introduced multiprocessor support to Pintos. To do this, we added support to the bootup code to perform CPU discovery and wake up additional CPUs. We also added device driver support for advanced interrupt management hardware, which was introduced by Intel

to support multiprocessing. We removed all uniprocessor assumptions from the code that manages the interaction between threads and between threads and interrupt handlers, which required switching from an interrupt-disabling approach to a spinlock-based approach. We extended the basecode scheduler to be capable of multiprocessor scheduling.

Second, we created a new scheduling project to introduce students to advanced multiprocessor scheduling. We provided specifications for a simplified version of Linux's Completely Fair Scheduler [43] and its associated load balancing algorithm, which students are required to implement. We also provided test cases, including stress tests, to assess the correctness of students' implementations.

Third, we evaluated the ability of our new Pintos framework to help students reach our intended learning objectives. We deployed the revised Pintos framework in an advanced systems course at our university. Our evaluation consisted of several instruments, such as one-to-one meetings with each group of students, a review of students' project submissions, a short quiz to assess student learning, and a post-survey to gather student feedback. Based on our meetings with students and their project submissions, we found that students find the projects challenging yet feasible, with most groups receiving a full score on the automated testing. Our manual inspection of students' code similarly did not reveal any substantial bugs not caught by the automated test cases. Our post-survey showed students displaying a strong understanding of the OS concepts we have set as learning objectives. Student feedback on the Pintos framework was also positive, with most students indicating that the project helped augment their programming abilities and deepen their understanding of operating

systems. From this, we conclude that our revised Pintos can be a successful framework for teaching undergraduate and graduate OS courses.

1.4 Roadmap

Chapter 2 provides background on multiple operating system concepts such as process management, scheduling, virtual memory, and concurrency. Chapter 3 provides a technical overview of Pintos. It discusses what is provided in the basecode and what is expected from students' code. Chapter 4 describes the changes that we made to the basecode to support symmetric multiprocessing. Chapter 5 provides details of our new scheduling project. Chapter 6 evaluates the effectiveness of Pintos/SMP as an educational OS based on a pilot use in an advanced systems course. Chapter 7 reviews related work, and Chapter 8 talks about future work. We conclude our work in Chapter 9.

Chapter 2

Background

2.1 Process management

The role of an OS is to manage system resources allocated to running programs that request them. It must do so in a safe and fair way, ensuring that no programs can interfere with the execution of other programs, and that every program has access to shared system resources.

2.1.1 Processes

A process is the operating system's abstraction for a running user program. The operating system provides the illusion that a process has exclusive access to the CPU, memory, and peripheral devices. Although all systems resources are shared with different processes, maintaining this illusion allows the application to be written without worrying about the possible

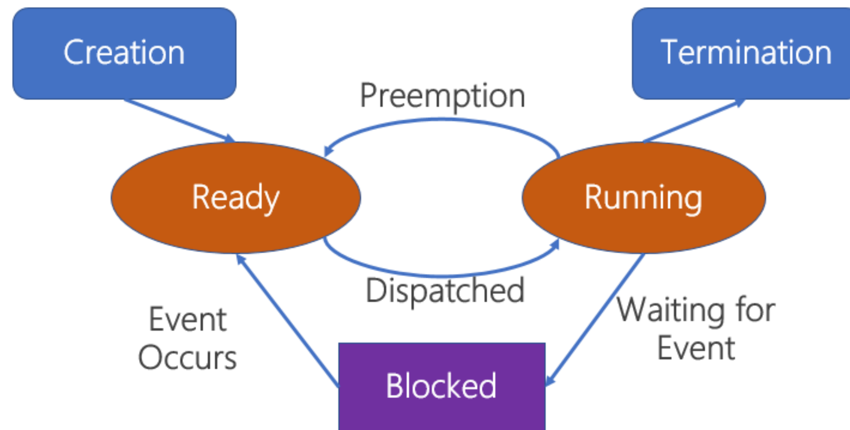

Figure 2.1: Process states. The three general states are RUNNING, READY, and BLOCKED

interference of other processes.

A process passes through several different states during execution. Although the detailed semantics of these states differ between operating systems, they typically include three states corresponding to RUNNING, READY, and BLOCKED, as shown in Figure 2.1. A process is in the RUNNING state when it is executing on a processor. It is placed in the READY state if it is runnable but is not currently running on any processor because it has not been chosen to run by the scheduler. A process is put in the BLOCKED state if it must wait for a resource to become available to make progress, such as user input. The process will not be scheduled again until the resource becomes available, at which point the process will be moved to the READY state.

2.1.2 Threads

Threads provide processes a way to run parts of their code independently of each other. A process has one or more threads, and each thread represents a single flow of execution. Threads can also speed up the execution of a process by allowing different parts to execute in parallel on separate CPUs.

2.2 Scheduling and Context Switching

A scheduler (or scheduling policy) determines how threads share access to the systems processor(s). Based on a chosen strategy, it decides which thread should run on the processor at any given time, and how long that thread runs for before switching to another thread.

When the scheduler decides to switch to another thread, a context switch occurs. During a context switch, the current thread saves the state of its execution so that the next time it is run, it can pick up where it left off. Then the state of the thread chosen to run next is restored, and execution resumes in the context of the chosen thread.

2.2.1 Preemptive vs Non-preemptive Scheduling

Scheduling policies can be categorized as preemptive or non-preemptive.

Non-preemptive

A non-preemptive scheduler, once chosen a thread to run, would run that thread to completion or until it blocks, even if another thread became ready to run that is in higher need of the CPU.

Preemptive

A preemptive scheduler, after choosing a thread, is willing to preempt that thread (suspend its execution) in order to run another thread that it deems to be in graver need of the CPU. How often the scheduler will preempt the running thread is dependent on the system. The system may decide at any time to preempt the running thread; however, in practice preemptions usually occur either at the arrival of a timer interrupt, or when another thread moves from the BLOCKED to READY state, since the newly unblocked thread may be in higher need of the CPU than the currently running thread.

Pintos, like most general-purpose operating systems, employs a preemptive scheduler in order to maintain the illusion of simultaneous multi-tasking, which results in a system with better interactive response to users.

2.2.2 Proportional-share Scheduler

Proportional-share schedulers (also called fair-share schedulers) adopt a time-sharing approach to give threads CPU time. Under this approach, each thread is given a share of the

CPU relative to its importance. Two simple implementations of proportional-share scheduling are lottery scheduling and stride scheduling [6, 51].

Lottery scheduling assigns each thread a number of tickets in proportion to its importance. Tickets are drawn from a contiguous range of numbers. When selecting a thread to run, the scheduler generates a random "winning number." Whichever thread holds the ticket of the winning number is chosen to run. Threads with more tickets are more likely to win; therefore over long periods of time, lottery scheduling achieves probabilistic (and approximate) fairness.

Stride scheduling, on the other hand, creates a deterministic and fair scheduling. Each thread is given a *stride* value, which is inversely proportional to the thread's importance. Threads also keep track of their *pass*, which is a counter that keeps track of how much weighted CPU time a thread received. When choosing a thread to run, the scheduler chooses the thread with the lowest *pass* and increments its *pass* by *stride*. Therefore, threads with lower *stride* values increment *pass* more slowly, causing them to receive more CPU time in comparison to other threads, thus achieving proportional sharing. Stride scheduling is the CPU variant of the WFQ packet scheduling algorithm [33].

Although lottery scheduling and stride scheduling do not meet the goals of general purpose operating systems, they serve as a basis from which more sophisticated schedulers, such as Linux's Completely Fair Scheduler, are derived.

2.2.3 Multi-CPU Scheduling

When multiple CPUs are available in the system, a work-conserving scheduler tries to keep available CPUs busy when there are ready processes to run, a goal pursued by most operating systems. Therefore, schedulers must support multiprocessing.

One of the simplest ways to do this is to keep the threads in a global queue that is shared by all CPUs. An advantage to this approach is that it ensures that no CPU is idle while threads are ready to run (but are not currently running). However, using a global queue has two main weaknesses.

The first weakness is a lack of scalability. The global queue must be locked while choosing the next job to run. Locking greatly reduces performance as the number of CPUs grows. Each CPU will spend more and more time contending for the global queue lock and less time actually running threads.

The second weakness is related to processor affinity. A thread can build up a fair amount of state in the caches and TLB associated with its running CPU. It is advantageous to try run it on the same CPU each time, as it will run faster than if it ran on a different CPU where its data is far less likely to be stored in the CPU cache. A global queue in which threads are equally likely to be chosen by any CPU may not preserve processor affinity.

Because of the weaknesses described above, many operating systems, including Pintos, use per-CPU queues. Each CPU manages only the threads on their own queue, independent of the other CPUs, thereby avoiding the scalability problem outlined above and improving

processor affinity.

Using separate, per-CPU ready queues, however, has the potential drawback in that it may lead to load imbalance between CPUs, which in turn can affect fairness and the ability to use all CPUs fully. For instance, if CPU0 manages thread A, and CPU1 manages two threads B and C, then A has exclusive access to CPU0, while B and C take turns being scheduled on CPU1. A then is given twice as much CPU time as B and C. The imbalance becomes worse if thread A finishes. Then CPU0 would be idle, while CPU1 is still shared between threads B and C. Load balancing is needed to avoid this problem by providing mechanisms and policies to migrate threads between CPUs in order to minimize load imbalance.

2.3 Virtual Memory

Virtual memory is a memory management technique that allows the OS to create an illusion of providing more memory than available as physical memory by using secondary storage. It maps virtual addresses, which are used by programs, to physical addresses in main memory. All memory accesses must first be translated before the data may be accessed. The OS moves data between secondary storage and main memory based on access patterns.

The memory management unit (MMU) is responsible for performing virtual-to-physical address translations. It also provides memory protection by preventing processes from accessing memory that they do not have permission to access. This prevents a malicious process from affecting the kernel or other processes running on the same system.

The translation performed by the MMU is relatively slow because it must traverse the process's page directory (a data structure containing all the virtual-to-physical mappings). For faster retrieval, a translation lookaside buffer (TLB) caches recent translations made by the MMU. When a virtual memory address is referenced by a process, the TLB is first consulted to look for the translation. In the case of a TLB miss (translation is not found), the MMU then traverses the page directory to make the translation.

In an SMP system, a TLB shootdown (see section 4.7) is needed when changing the virtual-to-physical mappings of a process in order to keep them synchronized between CPUs.

2.4 Interrupts

Interrupts are a signal to the CPU that an important event needs to be handled. When an interrupt occurs, the OS saves the execution context of the currently running thread and handles the interrupt. All interrupts have an associated interrupt handler which is invoked as a response to the arrival of that interrupt. The interrupt handler may or may not return control flow back to the running thread.

2.4.1 Exceptions

Exceptions are triggered by the CPU to inform the OS of a condition that prevented the successful completion of an instruction. Some exceptions, such as divide-by-zero or floating-

point exceptions are caused by logical errors in the application, which typically leads to its termination. Others, such as page faults or debug faults, require the intervention of the OS to handle the exception, after which the process can continue running normally.

2.4.2 Hardware Interrupts

Hardware interrupts are caused by hardware devices, such as the keyboard or timer device, or even another CPU. They inform the OS that the state of the hardware device has changed in a way that might be of interest to the OS. For example, a keyboard device may be informing the OS that the user just pressed a key. Sometimes, it can be inconvenient for the OS to attend to a hardware interrupt. In such a case, interrupts can be temporarily blocked. On the x86, for example, this can be achieved by executing a special `cli` instruction. This will prevent interrupts from disrupting execution flow until interrupts are enabled again by the `sti` instruction.

2.4.3 Software Interrupts

Software interrupts are often implemented as system calls. Software interrupts are similar to exceptions, but they are not caused by a condition that occurred when executing an instruction, but rather caused as a direct result of executing a special "trap" instruction. User processes use software interrupts to request services from the operating system. To access these services, a process can perform a system call, which starts with a software

interrupt. For example, a process may execute a system call to request more memory.

2.5 Managing Concurrency

Concurrency bugs are among the most challenging to test and debug. Given their non-deterministic nature, they are difficult to reproduce, impeding both in-house testing and postmortem debugging. Concurrency bugs are varied but often tend to follow common patterns. They include data races, atomicity violations, and deadlock [6]. Avoiding such bugs requires careful analysis of all possible interleaving of shared memory access between concurrent threads and applying proper techniques to synchronize shared access. Many tools [37, 36, 42, 28, 48, 14, 32] are available to aid the testing and debugging of concurrent programs.

2.5.1 Data Races

A data race occurs when two threads try to simultaneously access and modify a shared variable or data structure, causing unexpected behavior. For example, consider if two threads both try to increment a variable; e.g.

```
int counter = 0; //shared counter
void increment()
{
    counter++;
}
```

Figure 2.2: Incrementing a shared counter is a classic data race

The final value of *counter* should be 2. However, depending on the interleaving of the threads' accesses to the shared variable, the value could be 1. This is because incrementing a variable requires three machine instructions, comprising of reading the value of *counter* into a register, incrementing it, and writing back to the memory location containing *counter*. Without additional synchronization, it is possible for both threads to read the original value of *counter* (0), increment by 1, and both write 1 back to *counter*, causing an unexpected final value of *counter*.

2.5.2 Atomic Instructions

The race on the shared variable in the above example stemmed from the fact that incrementing involved three instructions. For that reason, most modern processors provide atomic instructions that allow updates to variables without exposing intermediate states to other threads.

For example, an atomic increment would guarantee that incrementing the counter happens in one, uninterruptible step. Atomic instructions are more expensive than non-atomic in-

structions, but less expensive than most other synchronization mechanisms.

2.5.3 Critical Sections

In many situations, the shared variable or data structure cannot be updated and placed in a stable state with a single operation. For example, inserting a node into a doubly linked list involves setting four pointers, and the list is in an unstable state until the last pointer is set. Therefore, it is important that other threads do not try to use the list while a thread is performing an insertion. Access and modification to the list must therefore be placed inside a critical section; that is, a section of code that can only be entered by one thread at a time to avoid a race condition.

2.5.4 Atomicity Violations

An atomicity violation occurs when programmers fail to enclose memory accesses that should occur atomically inside the same critical section. This causes incorrect program behavior even if all individual accesses to shared data are synchronized, as shown in Figure 2.3.

Two threads calling **increment**() may end up incrementing *counter* only by one, since the lock is released after reading *counter*. To avoid this violation, reading and updating of *counter* must be in the same critical section, as shown to the right of Figure 2.3.

```
int counter; // shared variable
             // protected by lock L

void increment() {
    int temp;

    lock (L);
    temp = counter;
    unlock (L);


    temp++; //Violation! No lock held

    lock (L);
    counter = temp;
    unlock (L);
}
```

(a) Lock dropped too early

```
int counter; // shared variable
             // protected by lock L

void increment_correct() {
    int temp;

    lock (L);
    temp = counter;

    temp++;

    counter = temp;
    unlock (L);
}
```

(b) Read and update of counter should be placed in the same critical section

Figure 2.3: An atomicity violation (left) and how to avoid it (right)

2.5.5 Locks and Mutual Exclusion

Mutual exclusion denotes the property that while any thread is executing within a critical section, other threads are prevented from entering that section.

A common way to provide mutual exclusion is through a lock. Whenever a thread needs to enter a critical section, it must first acquire a lock associated with that section. If it successfully acquires the lock, it enters the critical section, modifies the shared resource, and leaves the critical section, releasing the lock. If the lock is already acquired, it indicates that

another thread is inside the critical section; therefore, it must wait for the lock to be released by the other thread before attempting to modify the shared resource.

2.5.6 Spinlocks

A spinlock is another synchronization mechanism used to protect critical sections. Unlike a blocking lock, which blocks the thread if it failed to acquire the lock and unblocks it later when the lock is free, a spinlock will cause a thread to spin (repeatedly try) to acquire the lock. They avoid the relatively large overhead of blocking the current thread and context switching to a different thread when trying to acquire a lock that is already held by another thread. However, it is crucial to not hold a spinlock for long periods of time in order to avoid wasting CPU time. That means that spinlocks should not be held by a thread while that thread is not running.

Spinlocks can provide better performance in cases where the critical section is short, and therefore the expected wait time to enter it is short.

2.5.7 Deadlocks

A deadlock occurs when one thread *T1* acquires a lock *L1* and tries to acquire another lock *L2*. If another thread *T2* has acquired *L2* and tries to acquire *L1*, neither thread can progress since both are waiting for a lock held by the other thread. To avoid this situation, both threads must acquire the same locks in the same order.

Chapter 3

Pintos 2.0 Overview

This section provides a brief overview of the new, SMP-capable Pintos educational operating system we built. We split this chapter into two sections. The first section discusses what is provided to the students as baseline code in Pintos. The second section provides an overview of the structured projects.

Figure 3.1 provides a graphical overview of Pintos. It shows which parts are provided by the base code, what is expected of student code, and what tests are provided for each project. The components that changed as part of our work are highlighted in dotted yellow lines.

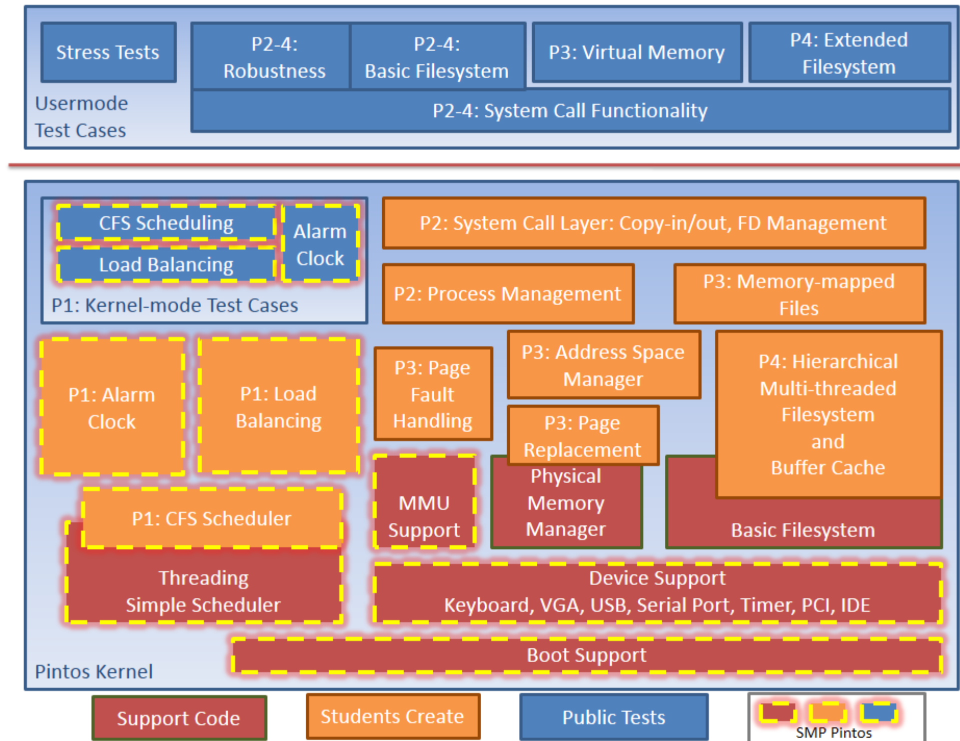

Figure 3.1: A graphical overview of Pintos. Components that changed as part of our work are highlighted in yellow dotted lines.

3.1 Basecode

The basecode can be divided into four logical OS components: CPU virtualization, memory management, I/O, and storage. We provide an overview of how Pintos implements each component.

3.1.1 Thread Management

Pintos maps user processes to kernel threads. The thread maintains a thread control block that contains information about the process, such as its id, status, and the program that the process is running. The thread also allocates kernel stack space for use during a user-to-kernel mode transition and to save CPU registers during a context switch.

The Pintos basecode comes with a simple round-robin scheduler that switches to a new thread every 4 timer interrupts. In project 1, students are asked to implement a more sophisticated scheduler to replace the one given by the basecode.

3.1.2 Memory Management

In addition to virtualizing the CPU, Pintos also provides processes the illusion that they have exclusive access to the entire memory space. It does so with the help of the virtual memory system available to all modern CPUs.

Managing Physical Memory

Pintos has a page allocator that allocates memory in chunks consisting of one or more contiguous pages. The page allocator uses free-space bitmaps to keep track of used and unused memory locations. The page allocator also evenly divides system memory into two "pools" called kernel and user pools. The user pool is solely for user memory, and the kernel pool is used for everything else.

Pintos also provides a malloc implementataion that uses the page allocator. This implementation uses segregated storage. It rounds up blocks to the nearest power of 2 (for small allocations) and uses the page allocator directly for large allocations.

Both the page allocator and malloc performs internal locking so that students do not need to provide their own synchronization mechanisms when allocating memory. However, this also means that memory cannot be allocated from interrupt context, since acquiring a lock can potentially block the caller. We have not found any need to allocate memory from interrupt context, but if necessary the locks could be changed to spinlocks, which are safe to use during interrupt context. Both allocators also provide debugging aid by setting bytes in recently freed memory to *0xcc*. This adds overhead to the allocator, but helps students find use-after-free errors.

Managing virtual addressing

The x86 ISA does not provide any way to directly access physical memory; therefore, Pintos maps kernel virtual memory one-to-one to physical memory. By default, user virtual memory ranges from *0* to *0xc0000000* (3GB) and kernel virtual memory ranges from 3GB to 4GB. Pintos is not able to access more memory than addressable by kernel virtual memory (1GB by default). This limitation is acceptable for Pintos given its educational nature. Pintos also provides functions and macros to facilitate working with virtual addresses.

Managing hardware page tables

The Memory Management Unit (MMU) is responsible for consulting page tables to make virtual-to-physical address translations. The Pintos basecode provides an abstract interface to the x86 hardware page table, which hides the internal representation and ensures consistency with the CPU's translation lookaside buffer (TLB). As a result, students can treat the code as a black box for installing, updating, and removing virtual-to-physical memory mappings. The code also provides helper functions for reading and writing dirty and accessed bits associated with each page table entry. These bits contain information about which pages have recently been touched, allowing OS designers to implement effective page replacement algorithms.

3.1.3 Managing I/O

Peripheral devices

Pintos has device driver support for the following I/O devices: keyboard, VGA, serial port, USB host controllers and PCI controller. The keyboard and serial device can be used to capture user input. The VGA and serial device can display output.

I/O devices in Pintos are interrupt-driven, but the serial device can also be configured to poll instead. Polling the serial device is useful for the first project, where students working intimately in the scheduler and interrupt infrastructure may not be able to print debugging

messages in critical sections of their code.

Disk

Pintos has a device driver written for the legacy IDE controller [2]. It supports reading and writing sectors on up to 4 IDE disks. It does not support SATA or SCSI.

Pintos has support for reading partitions on a disk, allowing a single disk to be separated into multiple segments for independent use. It also provides an abstraction layer for block devices: random-access, disk-like devices organized as arrays of fixed-sized blocks. The filesystem interacts with the disk via the abstraction provided by block devices.

3.1.4 Filesystem

The Pintos basecode provides a limited but working file system. Students have to work with the following limitations:

- No internal synchronization. A coarse-grained lock must be used to ensure that only one process is executing file system code at once.
- Fixed file size. There is no support for files to grow as needed.
- Files are allocated as a single extent, making it vulnerable to external fragmentation
- No subdirectories

In project 4, students will replace this basic filesystem with a more sophisticated one that addresses the limitations listed above (see subsection 3.2.4).

3.2 Student Projects

Pintos includes 4 structured projects, along with test cases for each project. These projects are intended to be completed by students over the course of about 3 months.

3.2.1 Project 1: Threads

The first project is about thread management. We describe threads to be in one of three states: READY, RUNNING, and BLOCKED. From reading the basecode, students gain insight into how the OS transitions the threads between the three states in order to virtualize the CPU. Students then enhance the existing timer facility and scheduler in Pintos.

As part of our work, we created a new scheduling project to replace Project 1. Details of the new project is provided in Chapter 5, so they will not be discussed further here.

3.2.2 Project 2: User Programs

Project 2 is aimed at providing insight into how an OS provides protection and isolation between user processes, how it exposes kernel services to user programs via system calls, and how the program code and data is laid out in the virtual address space.

Students are asked to implement argument passing to user programs. They then implement several system calls for process management, including **exec**(), **wait**(), and **exit**(). They also implement system calls that perform I/O and interact with the console and the file system, including **create**(), **remove**(), **open**(), **close**(), **read**(), and **write**().

3.2.3 Project 3: Virtual Memory

In project 3, students learn how an OS uses of the virtual memory system in order to create an illusion of managing more memory than available in physical memory by using secondary storage.

Students are asked to implement several techniques related to virtual memory, including on-demand paging of programs as well as page replacement. Students also implement stack growth by allocating additional pages as necessary when a process reaches its stack limit. Finally, they implement memory-mapped files and expose two system calls to user processes, **mmap**() and **munmap**().

3.2.4 Project 4: Filesystem

Project 4 asks students to implement a more sophisticated file system to replace the simple filesystem provided.

Students must implement a file system that allows a file to grow as long as free space is available and does not suffer from external fragmentation. We recommend a multi-level

indexed approach with direct, indirect, and doubly-indirect blocks. Students also implement a buffer cache for their file system which is used to improve performance while providing an abstraction to the underlying disk. Finally, students modify the filesystem such that a directory can point to files or to other directories. Students must implement several system calls to support subdirectories, including **mkdir**() and **chdir**().

3.3 Running and Debugging Pintos

Although Pintos targets the PC architecture and could run on real hardware, we typically recommend running Pintos on a virtual machine emulator. We have tested Pintos/SMP on Bochs [31], QEMU [9], KVM [18]. For development purposes, we encourage students to use QEMU or KVM. In our experience, QEMU provides a better remote debugging experience, while KVM runs the fastest. KVM utilizes the Intel architecture's virtualization facilities, and is therefore able to exploit multiple cores provided by the underlying host machine. This allows students to observe truly parallel execution of their Pintos kernel at nearly native speed.

Pintos can be debugged using print statements, which can be called from almost everywhere in the kernel. Alternatively, Pintos can be debugged using QEMU's internal debugger with GDB's remote debugging stub attached. This allows Pintos to be debugged almost as if it were a normal C program.

During a kernel panic, Pintos prints a backtrace of the call stack. This allows students to

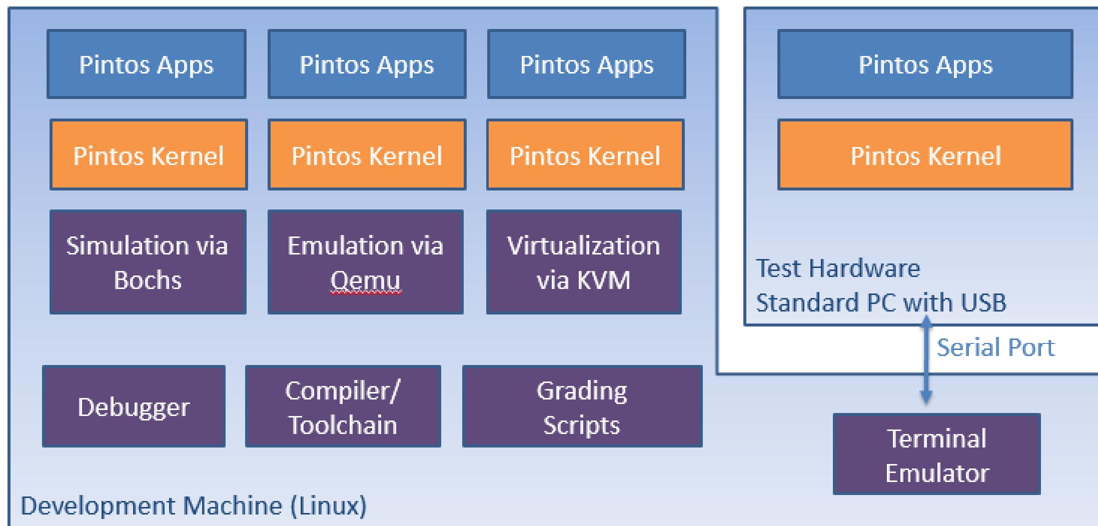

Figure 3.2: Running Pintos. Pintos can run on a variety of emulated environments, but also on real hardware

see how the kernel got to the improper state.

Chapter 4

Implementation

The following sections describe the changes that were made to the Pintos base code in order to support multiple processors.

4.1 Interrupt Management

To support multi-processing, Pintos must be configured to use the interrupt facilities provided by SMP hardware. Pintos originally supported the 8254 Programmable Interval Timer (PIT) [55, 23] and the 8259A Programmable Interrupt Controller [52], which was sufficient for uniprocessor systems, but not for multiprocessor systems. Therefore, we configured Pintos to use the advanced programmable interrupt controller [53], which was invented specifically to support multiprocessing. As shown in Figure 4.1, the APIC consists of two parts: the local APIC (LAPIC) [54, 25] and the I/O APIC [24, 25].

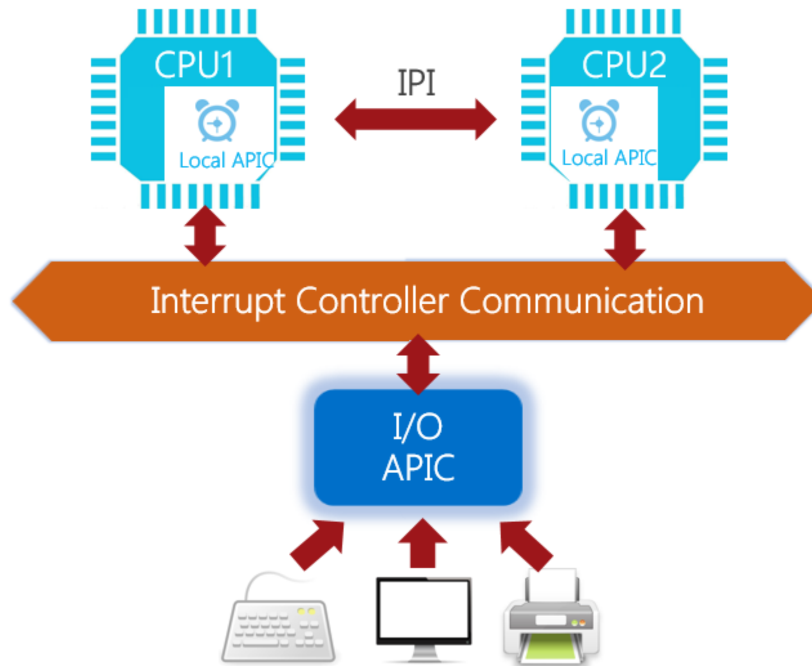

Figure 4.1: The advanced programmable interrupt controller (APIC). In an SMP system, it is used to deliver interrupts to CPUs. It consists of two parts, the Local APIC and the I/O APIC.

4.1.1 Per-CPU Timer Interrupts

The 8254A Programmable Interval Timer was used in Pintos to send timer interrupts to the CPU at a set frequency, but cannot be provide per-CPU timer interrupts. We replaced it with the LAPIC, which provides local timer interrupts.

The LAPIC is local to each CPU and provides three main functionalities. First, the LAPIC is used to send local timer interrupts to each CPU. Second, the LAPIC can issue Inter-Processor Interrupts (IPI) - hardware interrupts to the LAPIC on other CPU cores. Finally,

each LAPIC is assigned a unique ID which is used by CPUs to keep track of their CPU number. This is used to track per-CPU information.

4.1.2 Advanced PIC for I/O Interrupts

A programmable interrupt controller (PIC) is used to deliver hardware interrupts to the CPU. The PIC receives interrupt requests from different hardware sources (keyboard, timer, disk, etc.) and issues a hardware interrupt to the CPU.

The 8259A PIC [52] was a simple programmable interrupt controller used before the invention of multiprocessor systems. However, the 8259A PIC cannot manage interrupts in the presence of multiple CPUs. Therefore, Pintos/SMP disables the PIC and uses the I/O APIC to manage external interrupts.

The I/O APIC manages external interrupts in multiprocessor systems. It is used to prioritize and distribute interrupts across available processors. For simplicity, we do not try to balance the interrupts across different CPUs, and instead configure the I/O APIC to route all external interrupts to CPU0.

4.1.3 Configuring the APIC

The APIC is implemented as a memory-mapped device. Each CPU accesses their own LAPIC at the same addresses. The I/O APIC is mapped to give shared, symmetric I/O access from all processors [25].

By default, the physical base address of the APIC is *0xFE000000* [5]. Since that base address is unused in the kernel's virtual address space, we add a 1:1 virtual memory mapping starting at the base address to the kernel's page directory. We configured the APIC using code examples from the xv6 kernel [13].

4.2 Detection and Bootstrapping of Additional Processors

When the OS boots, only one CPU is running, called the Bootstrap Processor (BSP) as specified by Intel's MultiProcessor Specification [25]. The other CPUs, called Application Processors (AP) are in a halted state. Using code examples from xv6 [13], we added support for parsing the MP Configuration Table that is written to a pre-designated memory location by the BIOS. This system discovery is needed for the OS to determine how many processors are running on the underlying hardware, and the IDs of those processors. After the BSP has finished initializing its own CPU as well as any global initializations, it wakes up each of the APs in turn. To wake up each AP, the BSP fetches a page to serve as the APs initial stack, then copies the boot code for the APs into a free area of memory. Then, a wakeup IPI is sent to each AP in turn, which in turn causes it to execute the boot code. Each AP independently performs its own initializations before sending a signal to the BSP, indicating that it has finished. Each AP then in turn waits for all the other APs to finish initializing. After the last AP finishes its initialization, each AP calls into the scheduler to begin executing threads.

Global initializations, such as memory and I/O devices, are done only once by the BSP. Local CPU initializations, on the other hand, are done by both the BSP and by each AP as they are awoken. They include switching from 16-bit real mode to 32-bit protected mode, turning on paging, configuring the LAPIC, initializing segmentation hardware, and creating an idle thread.

4.3 Synchronization in SMP

The concurrency model in the original Pintos base code relied on the assumption that Pintos runs on only one processor. To support SMP, we had to remove this assumption.

4.3.1 Sources of Concurrency

The only sources of concurrency in UP systems are hardware interrupts arriving during the execution of a critical sections, as shown in Figure 4.2. Interrupts divert execution flow to interrupt handlers, which could also enter the same critical section. Interrupts could also cause preemption, which would allow another thread to be executed that may also enter the critical section. In an SMP world, concurrency is caused not only by interrupts on the current CPU, but also by threads or interrupt handlers running on other CPUs.

	Uniprocessor	Multiprocessor

Sources of concurrency	• Interrupts
	  • Interrupt handlers
	  • Preemption
		• Interrupts
		  • Interrupt handlers
		  • Preemption
		• Other CPUs

Figure 4.2: Sources of concurrency in UP and SMP Pintos

4.3.2 Managing Concurrency

Figure 4.3 shows how concurrency management changed between UP and SMP Pintos.

If the kernel is inside a section of code that can block (the common case) then the synchronization mechanism did not change. In both UP and SMP world, high-level constructs such as semaphores and locks are used to protect critical sections.

However, if the kernel is in a section of code that cannot block (for example, when servicing an interrupt), then a lock cannot be acquired, since a lock can block the caller. In uniprocessor Pintos, since all sources of concurrency came from interrupts executing on the current (and only) CPU, disabling interrupts was sufficient to prevent concurrency. In a multiprocessor world, where the critical section could be entered both from threads and interrupt handlers on the current CPU or other CPUs, a spinlock is needed instead. A spinlock provides mutual exclusion for short critical sections. We identified every instance in Pintos where concurrency was managed by disabling interrupts and replaced those instances with a spinlock. These instances were mostly in the device driver code, where threads communicated with I/O interrupt handlers using shared data structures.

Can block in a critical section?	Uniprocessor	Multiprocessor
YES	Semaphores, Locks	Semaphores, Locks
NO	Disable Interrupts	Spinlocks

Figure 4.3: Synchronization mechanisms used to manage concurrency in UP and SMP Pintos

4.3.3 Designing Synchronization Mechanisms

Figure 4.4 shows the relationship in PintOS between high-level synchronization constructs and the lower-level primitives from which they are built in both UP and SMP Pintos. This relationship is one of many ways to build these high-level synchronization constructs.

In the UP world, disabling interrupts was the most basic synchronization technique. Pintos used it to build semaphores and locks which are used when the execution context could block. In Pintos/SMP, spinlocks became the new "disable interrupt" in the sense that it is the most primitive method of protecting critical sections. Spinlocks are now also used in place of disabling interrupts to construct semaphores and locks.

Users of semaphores and locks were unaffected by the upgrade to SMP because all changes were hidden within the implementation of semaphores, while the interface remained the same. This minimized the amount of refactoring needed to bring SMP to Pintos since instances in the kernel where locks or semaphores were used to protect critical sections required no changes to support SMP.

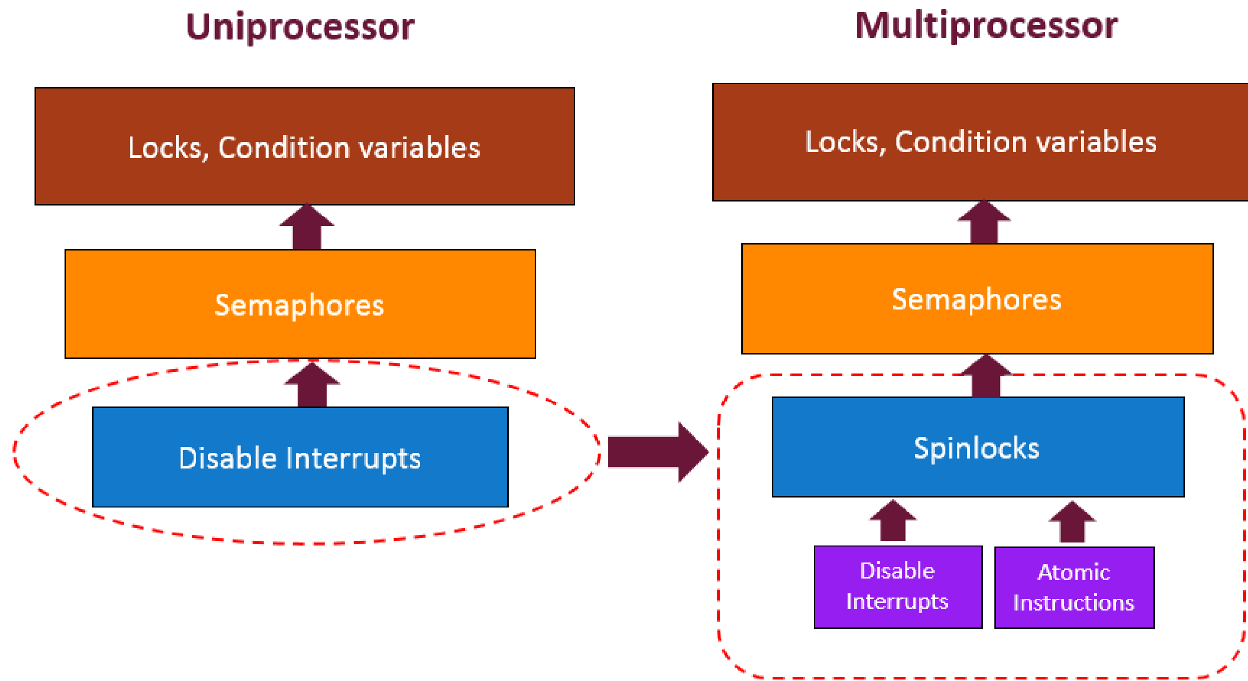

Figure 4.4: Relationship between synchronization mechanisms. Low-level synchronization primitives are used to build high-level constructs.

4.4 Spinlock Implementation

We implemented a spinlock that prevents other CPUs as well as interrupt handlers executing on the local CPU from entering a critical section. We do this by first saving the interrupt state and disabling interrupts, then repeatedly checking whether the spinlock is freeand, if so, set a "locked" flag indicating that the thread has acquired the spinlock. The check and acquisition must happen atomically, so an atomic compare-and-swap (CAS) is used. When releasing a spinlock, we atomically clear the "locked" flag and restore the interrupt state. Therefore, if interrupts were disabled before acquisition of the spinlock, then they remain disabled afterwards (and vice versa).

Spinlocks disable interrupts for two purposes. First, it is to protect against attempts to enter the critical section by interrupt handlers. Second, in order to prevent preemption. If a thread holding a spinlock were preempted, other threads attempting to acquire the spinlock would be required to spin until the first thread was rescheduled.

Other systems separate these two purposes by providing a flavor of spinlocks that disables preemption without disabling interrupts. For simplicity, we chose not to provide such a flavor at this time. However, we found this decision to decrease the applicability of spinlocks because of the risk of increased interrupt latency. In addition, we are unable to use spinlocks in situations where the kernel is expected to respond to interrupts.

To help students debug lock-related errors such as recursive locking or releasing an unacquired lock, every lock acquisition and release is recorded with a full backtrace. Thus, if a thread tries to acquire a lock it already holds, students can see a backtrace that led to the first lock acquisition. Similarly, if a thread tries to release an unacquired lock, students can see where the thread had already released the lock.

Our spinlock implementation is derived in part from the spinlock provided by the xv6 kernel [13].

4.5 Extending the Base Scheduler to SMP

As part of project 1, students implement a multiprocessor scheduler with load balancing. However, we wanted the base scheduler to also be capable of multiprocessor scheduling in

order to provide an SMP environment for project 2 and onwards without requiring a complete project 1 implementation. Therefore, we designed a simple heuristic to distribute load across CPUs.

The base scheduling policy remained the same, but each CPU now maintains its own per-CPU ready queue. When a new thread is created, a CPU is chosen in a round-robin fashion to execute that thread. This is enough to ensure high CPU utilization under general workloads such as those in project 2 and onwards. In addition, this heuristic intentionally suffers the problem of load imbalance in situations where threads on one CPU finish much quicker than that of another CPU. We depend on this weakness to design load balancing tests for Project 1.

4.6 Thread Block and Unblock

The problem of blocking and waking threads on an SMP system is a difficult one [45]. A thread, which we will refer to as a "sleeper," may block waiting for a condition to be met, such as an I/O interrupt to occur or access to a shared resource to be granted. When the condition is fulfilled, other threads or interrupt handlers, which we will refer to as "wakers," are responsible for unblocking the sleeper, resuming its execution.

Blocking on an event involves two parts. First, the sleeper checks whether the condition has been met. If not, then it adds itself to a "sleep queue", which is a list of threads that are blocked waiting for the condition to be fulfilled. Then, it marks its own state as BLOCKED

and calls the scheduler.

Once the condition has been fulfilled, a waker checks the "sleep queue" to find which thread(s) to awaken, removes them from the sleep queue and adds them to the ready queue. The newly unblocked thread(s) can now wake up and find their condition fulfilled.

Special care needs to be taken to avoid the "lost wakeup" problem, which could result in a sleeper blocking indefinitely even though the condition has been fulfilled. A lost wakeup can occur if the condition is fulfilled after the sleeper checks the condition but before it blocks. In this case, the waker will either not find the sleeper on the sleep queue, or find the sleeper on the sleep queue but still in an unblocked state. In the latter case, we detect an inconsistency and panic the kernel. If we did not panic the kernel, the waker would incorrectly believe to have woken the sleeper, and would not try to send another wakeup signal.

The lost wakeup problem is present in both uniprocessor and multiprocessor systems. The general solution is for the sleeper to *atomically* check the condition, add itself to the sleep queue and block. In a uniprocessor system, this atomicity could be guaranteed by disabling interrupts, which guarantees that no attempt can be made to unblock the thread. In multiprocessor systems, this is no longer the case since wakers may be running on other CPUs. Instead, we achieve this atomicity by using a spinlock which is used to protect access to the wakeup condition and the sleep queue. This spinlock must be acquired both when checking for the wakeup condition as well as when unblocking the thread. The caller must pass the spinlock to a function called **thread_block**(), as shown in Figure 4.5.

```
//Called with lk held
//lk protects access to wakeup condition and sleep queue
void thread_block (struct spinlock *lk)
{
    //Our ready queue lock ensures that releasing the lk, changing
    //the thread's state and calling the scheduler happens atomically
    lock_own_ready_queue ();
    spinlock_release (lk);
    thread_current ()->status = THREAD_BLOCKED;
    schedule ();
    unlock_own_ready_queue ();
    spinlock_acquire (lk);
}

//Called with lk held
void thread_unblock (struct thread *t)
{
    //Acquire the same spinlock the sleeper acquired using lock_own_ready_queue()
    spinlock_acquire (&t->cpu->rq.lock);
    //We check for inconsistency that could result in a lost wakeup
    ASSERT (t->status == THREAD_BLOCKED);
    t->status = THREAD_READY;
    unblock_thread(t);
    spinlock_release (&t->cpu->rq.lock);
}
```

Figure 4.5: Thread block and unblock in Pintos/SMP.

We ensure that **thread_block**() will atomically release the spinlock and block the thread. We achieve this atomicity by first acquiring a spinlock associated with the current CPU's ready queue, which also protects thread state changes occurring in the complementary **thread_unblock**() function.

4.7 TLB Shootdown

TLB invalidation is needed when a virtual-to-physical mapping is changed which is cached in the current or other CPU's TLB. Therefore, when a mapping is changed, each CPU must

be notified of the change and invalidate its TLB if necessary. This process is also referred to as TLB shootdown.

To notify other CPUs, we implement inter-processor interrupts (IPI), which are supported by the LAPIC. When a thread changes the virtual mapping of another thread, it writes the address of the page table that has been modified to a shared memory location and broadcasts an IPI to all other CPUs. All other CPUs respond to the IPI by flushing their TLB if the modified page table is used by the running thread. The sender must wait for all the CPUs to complete this process to ensure that all mappings are consistent before proceeding. To avoid deadlocking situations that may arise from multiple CPUs sending invalidation requests to each other and waiting, we mark TLB-invalidation as a critical section and protect it using a lock. This simple TLB shootdown protocol suffices for our purposes although we point out to students that real operating systems such as Linux tend to employ more sophisticated protocols to improve scalability.

4.8 Atomic Instructions

Pintos lacked an atomic operations library, which is needed to build higher-level synchronization mechanisms, and can also be used in lieu of locks in certain situations. Therefore, we provide a small library for atomic instructions based on the GCC __atomic builtins [16], which match the requirements of the C++11 memory model [1]. This requires that Pintos be built with GCC, which is a reasonable limitation as GCC is available in all distributions

of Linux.

The atomic library in Pintos ensures sequential consistency for atomic operations. The GCC builtins emit any necessary hardware instructions such as fences [1, 10, 17].

Chapter 5

New Scheduling Project

We created a new scheduling project to replace project 1. We took out requirements for the priority scheduler and the 4.4BSD scheduler. Instead, we ask students to implement a simplified version of Linux's Completely Fair Scheduler (CFS) [43] and its associated load balancing algorithm. We kept requirements for the alarm clock from the old scheduling assignment.

5.1 Alarm Clock

Although we did not change the requirements of this portion of the assignment, solutions written for UP Pintos will no longer work under SMP Pintos. This is because in UP Pintos, synchronization between threads and interrupt handlers was enforced by disabling interrupts. In SMP Pintos, spinlocks are required if there is a possibility of the interrupt handler

executing on another CPU. In addition, with additional CPUs each executing local timer interrupt handlers, students must choose between maintaining a per-CPU list of sleeping threads or a global list shared between all CPUs.

5.2 Simplified Completely Fair Scheduler

5.2.1 Overview

The scheduler that we ask the students to implement is a simplified version of Linux's Completely Fair Scheduler, which is an algorithm that belongs to the group of virtual time-based algorithms that are derived from the WFQ packet scheduling algorithm. It is similar to stride scheduling but introduces a number of heuristics intended to improve performance in a general purpose operating system.

We provide students full specifications of how their scheduler should behave. Students may make optimizations and changes as they see fit as long as it passes the test cases we provide. An acceptable scheduler implementation will exhibit the following behavior:

- Being "fair" to each thread. If all threads have the same importance to the user, and all threads are asking for CPU time, then it should give each thread an equal amount of CPU time. If threads are given differing degrees of importance, it should assign CPU time proportionally.
- Balance different threads' differing scheduling needs. Threads that perform a lot of I/O

require a fast response time to keep input and output devices busy, but need little CPU time. On the other hand, compute-bound threads need to receive a lot of CPU time to finish their work, but have no requirement for fast response time. Other threads lie somewhere in between, with periods of I/O punctuated by periods of computation, and thus have requirements that vary over time.

- Wherever possible, minimize scheduling overhead. That requires using efficient data structures to manage ready threads as well as minimizing the number of context switches made, while still meeting the aforementioned goals of fairness and interactivity.

5.2.2 Detailed Requirements

The following is the specifications that we provide students for their simplified CFS.

Niceness

Niceness is the traditional way in which Unix systems allow users to express the relative importance of threads. Each thread has an integer *nice* value that determines how "nice" the thread should be to other threads. A positive *nice*, to the maximum of *NICE_MAX* (19), decreases the priority of a thread, thus causing it to give up some CPU time it would otherwise receive. On the other hand, a negative *nice*, to the minimum of *NICE_MIN* (-20), tends to take away CPU time from other threads. By default, each thread has a nice value

of *NICE_DEFAULT* (0).

Computing Weights

The *nice* value is not directly used to make scheduling decisions. Instead, each thread is assigned an integer weight that is derived from its *nice* value. Threads with lower *nice* values have higher priority, and therefore are given higher weight. In contrast, threads with higher *nice* values have lower weight. Threads with higher weights tend to not only be scheduled more often, but run for longer periods of time when scheduled.

The following table shows the mapping from *nice* value to weight, as used in CFS. All values are represented as unsigned 64-bit integers.

```
static const uint32_t prio_to_weight[40] =
  {
    /* -20 */    88761, 71755, 56483, 46273, 36291,
    /* -15 */    29154, 23254, 18705, 14949, 11916,
    /* -10 */    9548, 7620, 6100, 4904, 3906,
    /*  -5 */    3121, 2501, 1991, 1586, 1277,
    /*   0 */    1024, 820, 655, 526, 423,
    /*   5 */    335, 272, 215, 172, 137,
    /*  10 */    110, 87, 70, 56, 45,
    /*  15 */    36, 29, 23, 18, 15,
  }
```

Figure 5.1: Mapping of process nice values to their weight

Calculating Virtual Runtime *vruntime*

Each thread keeps track of its *vruntime*, which is short for ”virtual runtime.” It is a normalized measure of how much CPU time a thread has already consumed. CFS always selects

the thread with the lowest *vruntime* value when picking a task to run, which represents the thread that is farthest behind relative to its desired share.

If multiple threads have the same *vruntime* value, the thread with the lowest tid is scheduled. (This tie breaker is needed only for the tests; it is not used in the actual CFS algorithm.)

A thread's virtual runtime increases linearly as a function of consumed CPU time d where a thread's weight determines the rate of the increase. Hence, given the same amount of CPU consumption, *vruntime* increases more slowly for threads with higher weight and more quickly for a thread with lower weight.

Specifically, *vruntime* is computed based on the thread's consumed CPU time d and its weight w as follows:

$$\text{vruntime} = \text{vruntime}_0 + d * w_0/w$$

where $vruntime_0$ is an initial value for the thread's virtual runtime set when the thread is added to the ready queue, and where w_0 is the weight of a thread with a *nice* value of 0. (64-bit integer arithmetic must be used for all CFS calculations.)

The very first thread's $vruntime_0$ is initialized to 0, but consider what would happen if the $vruntime_0$ values of threads created later were also set to 0 when those threads are added to the ready queue: those threads would appear to have no CPU time consumed at all, and would be preferred by the scheduler until they caught up with the threads that were already running in the system (which would then not receive any CPU time).

Instead, CFS chooses as the initial value of $vruntime_0$ for newly created threads the minimum

value of *vruntime* of all threads already running or ready at that point. This value, called *min_vruntime*, must be maintained for each ready queue.

The scheduler must calculate the *vruntime* values of all threads in accordance with their actual CPU consumption, so that accurate values are used when the scheduler makes decisions. Specifically, the scheduler must use updated values of *vruntime* when a new thread is created and whenever a thread is unblocked.

Calculating *ideal_runtime*

At each timer interrupt the scheduler needs to decide whether to preempt the currently running thread or not. A thread is preempted if it has run for more than its "ideal runtime," which represents the length of this thread's time slice. In CFS, the length of a thread's time slice depends on its niceness: higher priority threads receive longer time slices than lower priority threads.

Specifically, the *ideal_runtime* is computed as

$$\text{ideal_runtime} = 4000000 * n * w/s$$

where n is the number of threads either running or ready to run, w is the weight of the thread, and s is the sum of weights of all threads that are either running or ready to run. Since all of these variables may change as a thread runs, the preemption decision should be made based on their current values when a timer interrupt occurs.

Notice that in the common case where all threads have the same weight ($s = n * w$), the

ideal runtime is 4,000,000ns, or 4ms. For example, assuming a timer frequency of 1000 Hz, if 2 CPU bound threads were running on a CPU, they would be taking turns every 4 clock ticks.

This time interval is long enough to avoid excessive context switch overhead, but short enough so that users can perceive their threads as making progress simultaneously.

Handling I/O Bound Threads

I/O bound threads spend much of their time in the blocked state, waiting for I/O operations to complete. (In the Linux kernel, they are referred to as "sleepers.") An example is a program such as PowerPoint, which may run only when a user presses a key to update a slide, then go back sleeping to wait for more input. To increase responsiveness, the scheduler should schedule such threads as early as possible when they become ready. Most general-purpose schedulers, CFS included, include a special policy for this case.

When a thread is unblocked, its *vruntime* is likely to be lower than that of other threads that did not sleep. As in the case of newly created threads discussed above, without adjustment, those threads would be scheduled by the scheduler until they have caught up with the others. Although this meets the goal of minimizing latency, it is in general undesirable, particularly if the thread now started using the CPU extensively.

To avoid this, CFS sets an unblocked thread's $vruntimee_0$ to a slightly smaller value than

min_vruntime, specifically:

$$\text{vruntime}_0 = \max(\text{vruntime}, \text{min_vruntime} - 20000000)$$

where 20000000 represents the "sleeper bonus" given to I/O bound processes when they wake up (unblock). This adjustment tends to place these threads at the front of the ready queue.

To avoid threads manipulating this system by intentionally sleeping, the previous value of *vruntime* from when the thread began sleeping is included as a lower bound. This ensures that a thread's *vruntime* value cannot decrease, thus threads will not be receiving more CPU time than if they had been continuously ready.

Threads receiving a sleeper bonus may temporarily obtain a *vruntime* value that is less than *min_vruntime*. In this case, do not take the *vruntime* value of the thread receiving the bonus into account when updating the ready queue's *min_vruntime*. You must maintain the invariant that a ready queue's *min_vruntime* value increases monotonically.

Keep in mind that unblocked threads are not necessarily added to the current CPU's ready queue. To preserve processor affinity, unblocked threads are added to the CPU on which they last ran.

Since I/O bound threads require quick response times, the scheduler must arrange for the unblocked thread to preempt the running thread if the idle thread is running, or if the *vruntime* of the newly unblocked thread is lower than that of the currently running thread. To do so, it must return RETURN_YIELD from **sched_unblock**() in this case. As a simplification

for this project, all occasions in which an existing thread is unblocked should be treated as if they were related to I/O.

Summary

A summary of the CFS algorithm is provided below:

- At each timer tick, preempt the current thread if it has run for at least *ideal_runtime*. When choosing which thread to run next, pick the thread with lowest *vruntime*. Break ties by choosing lowest tid.
- Let d be the amount of CPU time consumed since a thread's *vruntime* was last updated, w_0 be the weight of a thread with 0 *nice*, and w be the weight of the thread. Then:

$$\text{vruntime} += d * w_0 / w$$

- Maintain *min_vruntime*, the minimum value of *vruntime* of all running or ready threads. Both *vruntime* and *min_vruntime* are always nonnegative.
- Let n be the number of threads either running or ready to run, w be the weight of the currently running thread, and s be the sum of weights of all threads that are either running or ready to run. Then:

$$\text{ideal_runtime} = 4000000 * n * w / s$$

- When a thread is unblocked for the first time, set its *vruntime* to:

$$\text{vruntime}_0 = \text{min_vruntime}$$

- When a thread is unblocked subsequently, set its *vruntime* to:

$$\text{vruntime}_0 = \max(\text{vruntime}, \text{min_vruntime} - 20000000)$$

  If the *vruntime* value of the unblocked thread is lower than that of the current thread, or if the CPU is idle, arrange for the CPU to yield.

5.2.3 Differences from Linux

This section summarizes the simplifications we made to the Completely Fair Scheduler from the Linux 2.6.24 kernel. We feel that these simplifications make the project more feasible for students while retaining the core aspects of CFS.

- We removed *sysctl_sched_wakeup_granularity*, which defines by how much a thread's *vruntime* must exceed that of a recently woken thread in order to be preempted. Instead, in Pintos a newly woken thread will preempt the running thread if its *vruntime* is strictly less than that of the current thread.
- We removed *sysctl_sched_child_runs_first*, which specifies whether a child or parent should be the first to run after a call to **fork**(). It does so by examining the vruntimes of the child and parent and swapping them if necessary. In Pintos, either process could run first after a call to **fork**(), depending on which process has the lower vruntime.
- We removed *sysctl_sched_latency*, which is a system setting that defines the period in which all threads are guaranteed to be scheduled at least once (subject to increase as the

number of ready threads increases). In Linux, the scheduling period increases linearly if num_threads_ready is greater than 5. We removed the case where num_threads_ready is not greater than 5, and therefore in Pintos the period always increases linearly with num_threads_ready.

- Linux manages ready threads in a red-black tree for $O(\log n)$ insertion and dequeue [41]. We do not include a balanced binary tree implementation in Pintos, and instead provide a sorted list that allows for $O(n)$ insertion and $O(1)$ dequeue, and we encourage students to manage the ready queue with our provided list implementation. We stress to the students that in production systems, such a large time complexity would be unacceptable.

5.2.4 Tests

Testing the correct behavior of a scheduler can be tricky. On the one hand, tests need to verify that the desired policy is implemented correctly, which tends to favor a unit-test based approach. On the other hand, the scheduler implementation must work in an actual kernel environment to schedule a workload of real threads.

We pursue a dual approach to testing that includes both simulation and actual execution. The simulator framework is built into the Pintos kernel, ensuring no changes are needed to students' implementations for testing.

Simulated Tests

The majority of CFS tests is performed under the simulated scheduler. In these tests, we do not actually create or schedule any real threads. Instead, the scheduler simulator simulates how threads would be scheduled under the students' scheduler. The simulator framework asks the students' scheduler for which scheduling decisions to make at which point, but it does not actually switch between threads. Instead, it verifies that the correct thread is selected to run at the correct time, based on the algorithm. As such, it is able to create a wide variety of scheduling scenarios and check if the students' scheduler makes the correct decisions. This is implemented in tests/threads/cfstest.c and tests/threads/simulator.c.

The tests are set up in functions **cfstest_set_up**() and **cfstest_tear_down**(), defined in tests/threads/cfstest.c. The simulator sets up a "fake CPU" that does not represent an actual CPU on the hardware, but rather a virtual environment where the simulator can create threads, execute timer interrupts, etc., without affecting the system. During setup, **change_cpu**() is called so that the CPU local variable *cpu* points to the fake CPU. After that, all OS events are directed towards the simulated CPU, causing the students' scheduler to be invoked in the process. The real CPU is restored at the end of the test.

During the simulated testing, interrupts are disabled, so no real timer interrupts will arrive. Timer interrupts are simulated by setting the system time via **timer_settime**() and then executing **driver_interrupt_tick**(), which in turn invokes **driver_tick**(). These functions are almost identical to **timer_interrupt**() in devices/timer.c and **thread_tick**() in thread.c

respectively.

At the beginning of each test, the system time is set to 0, so any time spent prior to the test does not affect the test. Each test defines a set of OS events that arrive after a certain amount of time. Each OS event is a scheduling event that will invoke the students' scheduler. At the end of each event, the test checks that the thread that the students' scheduler would run at the end of the event is the correct one. The real time is restored at the end of the test.

To ease the maintenance of simulated tests, we developed a test generator, written in Python, to auto-generate the test cases. For each test, the generator takes as input a list of OS events and their arrival time. For each event, it calculates the correct thread that should be selected based on the specifications we provide. It then generates C code that is used to check the correctness of students' submissions.

Real Workload Tests

These tests do not run the simulator, but rather schedule real threads doing work under the students' scheduler implementation. It is not able to check if the students' scheduler conforms to our specifications, but is rather used to pick up possible bugs that may not be caught by the simulated tests.

5.3 Load Balancer

5.3.1 Overview

We ask students to implement a load balancing policy that is largely unchanged from that of Linux. This load balancing policy is specific towards the CFS scheduler because its load metric is CFS specific. Thus students must have at least a partial implementation of their simplified CFS working before attempting to implement load balancing.

The load balancing policy we ask them to implement pursues the following goals:

- Ensuring that no CPU should be idle while there are ready threads in any CPU's ready queue.
- Ensuring that each CPU should be equally loaded to maximize fairness.
- Wherever possible, minimize the migration of threads, since migrating a thread to a different CPU diminishes its processor affinity.

5.3.2 Detailed Requirements

When load balancing, a CPU moves threads from another CPU's ready queue to its own. To decide whether to pull threads from another CPU's ready queue, CFS examines the load on each CPU, represented by a variable *cpu_load*. *cpu_load* is defined as the sum of the weights of all threads in the ready queue.

A CPU's *imbalance* is calculated as follows:

$$\text{imbalance} = (\text{busiest_cpu_load} - \text{my_load})/2$$

where *busiest_cpu_load* is the *cpu_load* of the CPU with highest load and *my_load* is the *cpu_load* of the CPU that is executing the load balancing.

If imbalance is small ($\text{imbalance} * 4 < \text{busiest_cpu_load}$) then no rebalancing occurs. Otherwise, CFS pulls threads from the busiest CPU to the CPU that initiated the load balancing. It continues to do so until the aggregate weight of threads that have been migrated equals or exceeds *imbalance*.

Since threads' *vruntime* values are significant only when compared to the *vruntime* of other threads on a CPU's local queue, the *vruntime* of threads in different CPU's ready queues may be vastly different. Therefore, the *vruntime* values of each of the migrated threads must be adjusted upon migration as follows:

$$\text{vruntime}_0 = \text{vruntime} - \text{busiest_cpu_minvruntime} + \text{my_minvruntime}$$

where *busiest_cpu_minvruntime* is the *min_vruntime* of the busiest CPU and *my_minvruntime* is the *min_vruntime* of the CPU that initiated the load balancing. This adjustment will allow a thread to retain any sleeper bonus it may have enjoyed at the time of the migration. The *min_vruntime* of the receiving CPU is not updated.

5.3.3 Differences from Linux

In Pintos, we allow students to decide for themselves how often they invoke their load balancer, with the only requirement being that it is invoked when the CPU is idle in order to avoid missing when there are available threads in other CPU's ready queues. In Linux, load balancing when the CPU is idle is not sufficient to ensuring that loads are balanced across CPUs, since each CPU could chronically have different loads without any CPU going idle. Therefore, load balancing is invoked by each CPU periodically. In Linux, load balancing is also invoked right before the system goes idle.

5.3.4 Tests

Load balancing tests run real workloads rather than invoking the simulator. When threads are created in Pintos, they are added to the CPU's ready queues in a round-robin fashion. The load automated tests for the load balancer relies on this assumption to ensure that light threads are spawned on one CPU while computationally heavy threads are spawned on another CPU. The tests then check CPU statistics to ensure that all CPUs remain equally loaded despite CPUs receiving unequal loads. The tests are also run multiple times to check that the student implementation appears to be free of race conditions. Unlike the CFS simulated tests, these tests do not check that the load balancing algorithm exactly matches the one described in the specifications. However, inefficient or incorrect implementations that do not result in roughly balanced CPU loads will not be accepted.

We tested using real workloads rather than invoking the scheduling simulator because the simulator does not provide the non-determinism that is needed to reveal possible data races and atomicity violations present in students' implementation. The load balancer must acquire per-CPU locks correctly in order to avoid a data race on the per-CPU local queues. In addition, with load balancing, an atomicity violation can arise in two scenarios. First, it can arise if a thread checks and then uses the value of the current CPU without first disabling interrupts. The thread may be preempted and migrated to a different CPU after the thread checks its CPU, but before it uses that CPU value, causing it to use the previous CPU value despite running on a different CPU. In addition, a thread cannot assume that it will be blocked and woken up on the same CPU, since it may be migrated prior to awakening. Depending on the students' design, this may require them to revisit their alarm clock implementation and make sure it is free of the aforementioned atomicity violation. Figure 5.2 shows a possible implementation of the alarm clock that suffers from both of the aforementioned atomicity violations.

```
void
timer_sleep (int64_t ticks)
{
  struct thread *thread = thread_current();
  thread->next_wakeup = timer_ticks() + ticks;
  //BUG 1! Thread could be migrated after reading get_cpu(),
  //causing it to acquire the wrong spinlock.
  spinlock_acquire(&get_cpu()->lock);
  list_insert(&get_cpu()->sleep_list, &thread->elem);
  thread_block(&get_cpu()->lock);
  //BUG 2! Thread could be migrated after wakeup, causing
  //the following statement to release an unacquired lock.
  spinlock_release(&get_cpu()->lock);
}
```

Figure 5.2: Atomicity violation caused by load balancing

Chapter 6

Evaluation

To evaluate the effectiveness of Pintos/SMP as an instructional operating system, we used it as the basis for teaching a Systems and Networking Capstone class with 21 enrolled students, who worked in 8 groups of two or three. We collected results from their first project only. Our evaluation consisted of several instruments, including one-to-one meetings with each group of students, a review of students' project submissions, a short quiz to assess student learning, and a post-survey to gather student feedback.

We followed standard research procedures for human subjects that were approved by Virginia Tech's Instituational Review Board (FWA00000572). Participation in the study was voluntary and had no effect on students' grades. Our data was collected with the consent of each student. It was stripped of personally identifiable information before it was handed over to the research team.

6.1 Test Cases

Figure 6.1 shows the scores the students received from the automatic test cases. The test cases were provided to the students, so they could check their own code before submitting for grading. Therefore, failure to pass a test case indicates that the group was not able to implement the feature before the project was due, or the test passed during their own testing but failed when we tested their code. There can be numerous reasons for this; the most common reason being that there was a race condition present in the student code that did not manifest itself during students' testing.

From the scores, it seems that students were able to correctly implement the required functionality within the time frame given to them. Only two students (one group) were not able to pass all the test cases that we provided. This indicates that the project, although intended to be challenging, was not overwhelming for the students.

6.2 Design Documents

Design documents are a set of structured questionnaires related to each project that we require students to individually fill out in addition to implementing the projects. We do this for two reasons. First, design documents give students an opportunity to justify their design and discuss the choices and trade-offs they made. Second, since this project is a group effort, requiring a design document individually from each student ensures that students

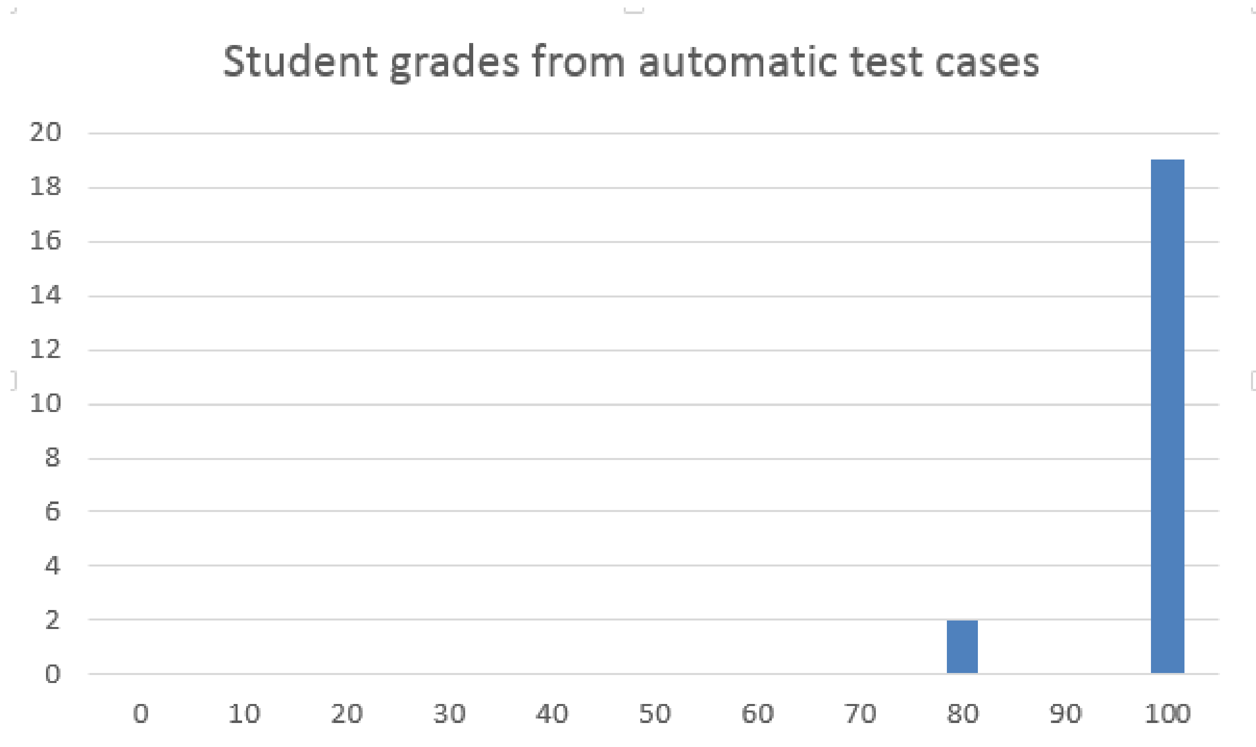

Figure 6.1: Student grades from automatic test cases

understand their combined solution to all parts of the assignment, not just the parts that they personally implemented. Students lost points if they made poor design choices that resulted in a sub-optimal system, or if their responses to the questionnaire demonstrated poor understanding of the components we had asked them to build.

Figure 6.2 shows the grades that the students received on their design documents. Overall the grades were very good, with a majority of students receiving full credit. One student received a grade far lower than his or her class peers - upon reviewing his submission we found that many of the questions were either unanswered, incomplete, or inconsistent with the implementation that his or her group provided. We believe this may be a good example

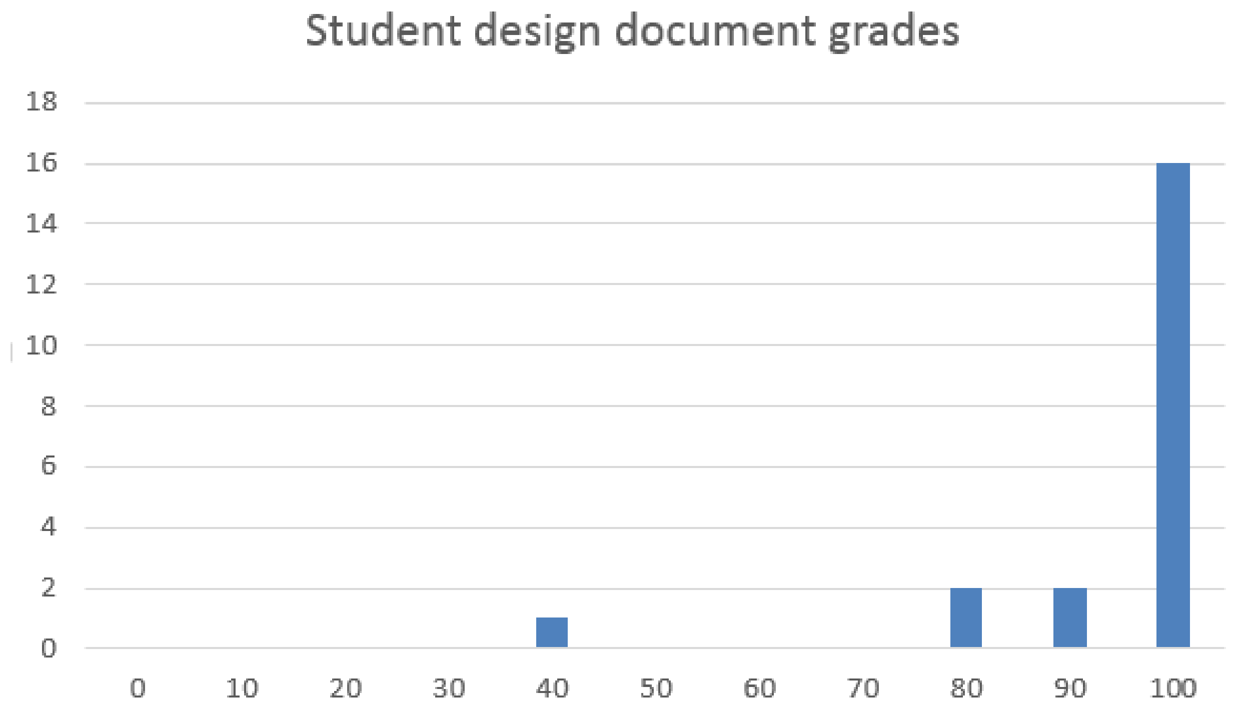

Figure 6.2: Scores received by students on their design documents

of why design documents are needed; the student likely was not as involved as they should have been during the design and implementation of their project.

6.3 Post-Survey

Shortly after the first project was due, we asked students to fill out a survey to give them a chance to provide feedback and also for us to evaluate their learning. Only one student was absent from the survey.

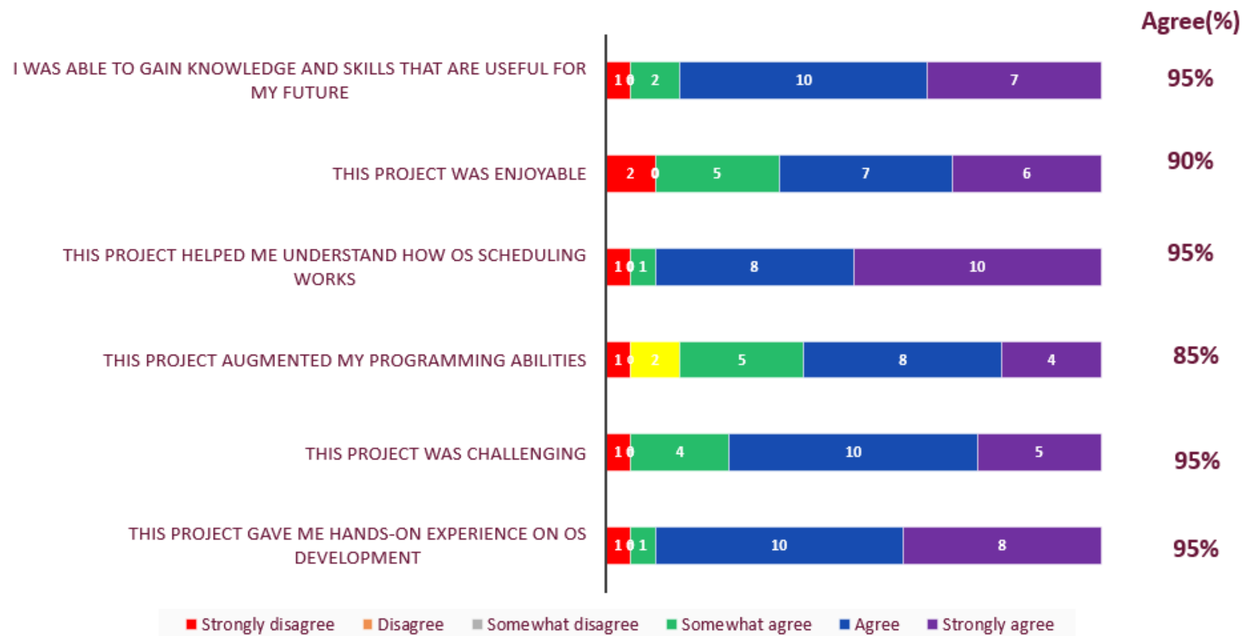

Figure 6.3: Students' feedback about Project 1

6.3.1 Student Feedback

Figure 6.3 shows students' responses to several questions we asked them about their perceived learning as a result of completing project 1. The responses were largely positive, with a large majority of students indicating that they found our new scheduling project to be challenging, enjoyable, and successful in helping them understand OS scheduling. They further indicated that implementing our project allowed them to gain knowledge and skills that increased their "market value" [15]. We believe these characteristics make Pintos a successful educational framework [27].

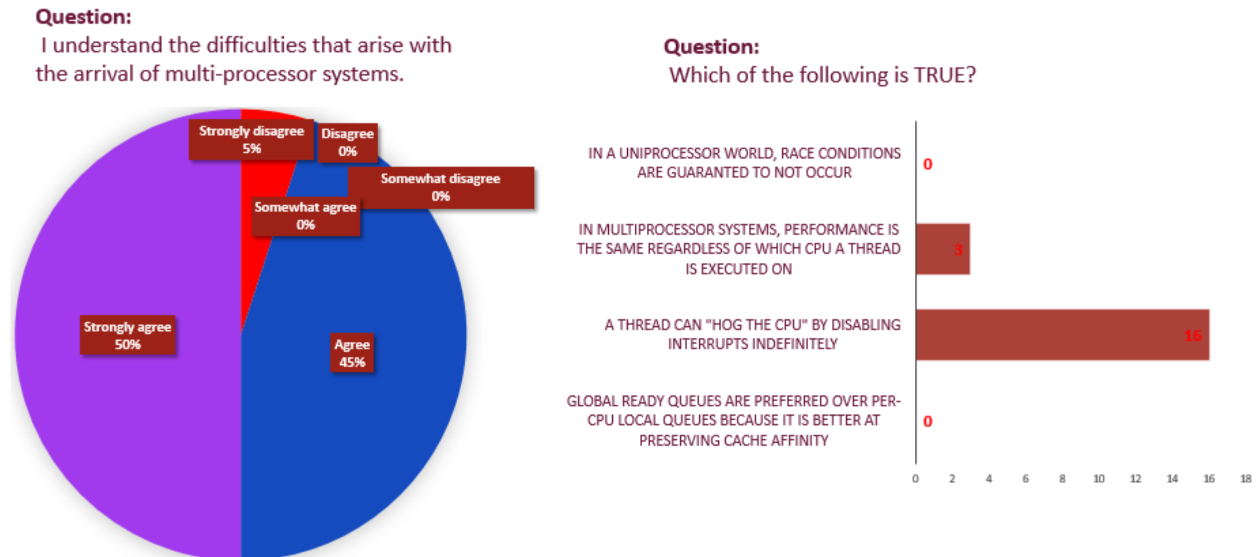

Figure 6.4: Student's learning about multiprocessor operating systems

6.3.2 Student Learning Evaluation

We identified three learning goals of our new project and asked students how much they perceive to have learned about each of our goals as a result of completing the project. We then followed up with a short quiz to see measure their actual learning.

The learning goals we identified related to multiprocessor operating systems, scheduling, and load balancing. Figures 6.4, 6.5, and 6.6 show the students' responses to questions about each of our learning goals.

We found that the students' perceived learning matches their actual learning. From the survey, 95% of the students sampled indicated they were confident in their understanding of each of our learning goals. Their confidence is confirmed by the results of the quiz, with each question having over 80% correct response rate and one being answered correctly by every

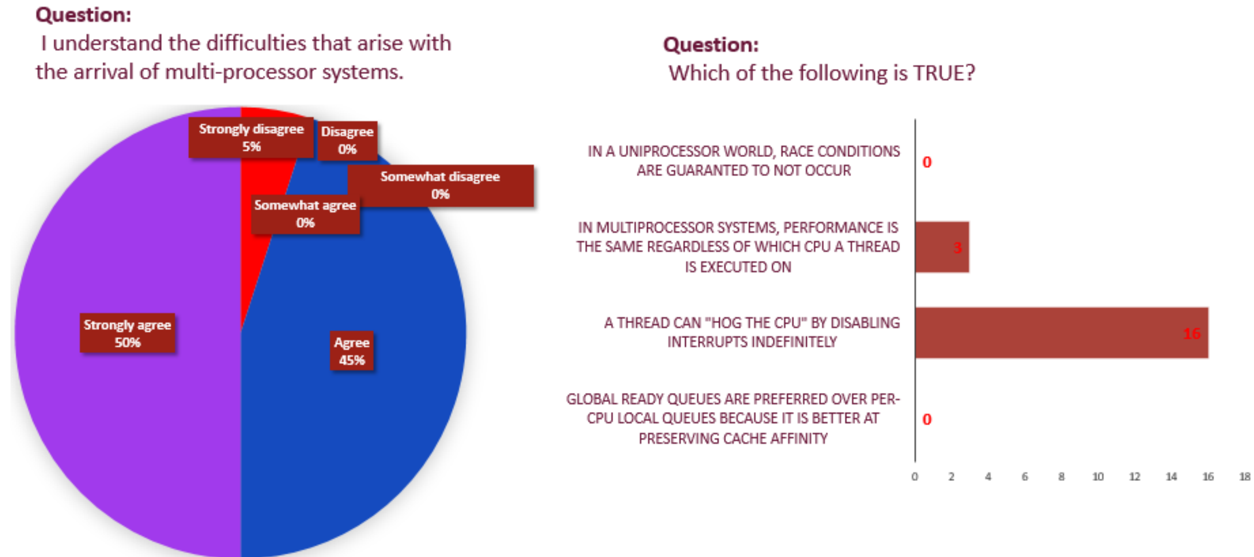

Figure 6.5: Student's learning about OS scheduling

student. Based on these results, we conclude that the practical exposure to a real, functional OS like Pintos may have enabled a majority of students to obtain a grasp of operating system concepts.

6.3.3 Student Comments/Suggestions

Lastly, we gave students an opportunity to leave comments and suggestions for how we can improve project 1.

Students' comments and suggestions on project 1 were varied but there were common themes. Many students indicated they faced difficulty understanding the base code that was provided for them, and indicated that we might want to provide a visual explanation on how our thread, scheduling, and timer modules interact with each other. Some students also indicated

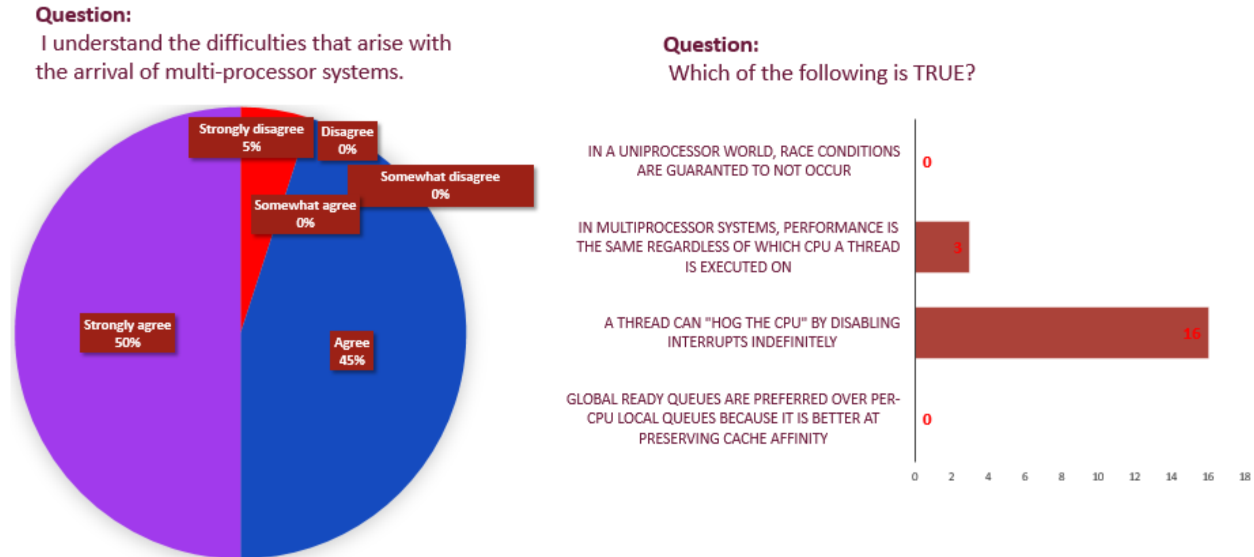

Figure 6.6: Student's learning about load balancing

difficulty debugging and indicated that they would like better tools or documentation to help uncover their bugs. On the other hand, many students also indicated that they felt the specifications were well written and that the project was challenging but not unreasonably so.

6.4 Interactions with Students

We had the opportunity to interact with students directly during their completion of project 1. We held meetings with all groups midway through the assignment to review their progress, answer questions, and provide help. We also held help sessions the week before the project was due to provide guidance to individual students.

From the mid-point meeting, we found that the students had already largely read and understood the base code, and had already begun implementing their project. Most groups came to the meeting with at least a partial implementation of the alarm clock portion of the assignment, and many had already given thought on how to implement the CFS scheduler. We also found that, for the students who had already begun coding, they were able to understand their code and why it worked or did not work. No groups were able to "hack" a solution without understanding how it interacts with the rest of the system. Because of this, we feel that this project is effective for enforcing student learning since completion of the project requires deep understanding and careful analysis of the system from the students.

Most students came to the help sessions because they had difficulty debugging their solution. By far the most common symptom of their bugs was a race condition, which we believe to be the most difficult bugs for students to debug on their own.

6.5 Limitations of our Evaluation Methodology

A post-analysis of our evaluation methodology revealed several limitations.

First, for administrative reasons, we did not create a pre-survey for students to take prior to completing our new project. Our post-survey measures learning from not only our instruction, but also from other sources as well, such as other classes taught at our university. A pre-survey would therefore have been important because it measures students' knowledge prior to completing our assignment, which in turn would have allowed us to measure how

much of students' learning came from our instruction.

Second, since our quiz was optional and did not affect students' grades, we kept it short in order to avoid a situation where students either refuse to take the quiz, or avoid making their best effort in responding to the questions. As a result, we asked a total of only three questions, which is not enough to accurately measure students' understanding of our concepts. If we were to repeat the evaluation, we would increase the number and raise the difficulty of the quiz questions and also incentivize students to answer each question to the best of their ability.

Chapter 7

Related Work

This section first contains a discussion of how operating system courses are taught at other universities. Then we compare and contrast educational operating systems that are similar to Pintos in its design and general use.

Although an overwhelming majority of operating system educators focus on systems for PC machines, many, such as iPodLinux [30], PortOS [7], and others [4] recognize the growing prevalence of mobile devices in the computing landscape and thus introduced frameworks for students to gain hands-on experience programming for those environments, which is different from that of traditional desktops and servers [4]. Others, such as BabyOS [34] and MiniOS [46] are instead geared towards embedded devices, which are often simpler than traditional desktop systems, but are also limited in its hardware resources, thereby introducing additional challenges to the system designer [34].

OS education frameworks can also be categorized based on the complexity of the base code that is given to students. On one end of the spectrum, systems like Yalnix [56], MiniOS [46], and JaeOS [38] provide little to no base code and ask students to implement a limited but functional operating system from scratch. Others use production level operating systems such as Windows [47] and Linux [29, 20, 4, 30] as the base in which students develop other small-scale functionalities, such as implementing a new system call or scheduling class. However, the drawback is that real OSes are full of complexities for coping with real-world issues that have little instructional value but make the code difficult to understand. Furthermore, since these systems are already fully functional, students do not design or implement major subsystems [21]. Special-purpose instructional operating systems such as Pintos and others [39, 22, 11, 21, 8, 7, 34] attempt to capture the best of both worlds by providing students a partially functional operating system which they must study and enhance by implementing additional functionalities or replacing existing subsystems. This goal of "combining realism with simplicity" [34] gives students a high level view of the core components of a fully functional operating system without worrying about the specifics of how lower-level constructs are implemented, thereby maximizing their learning with minimum effort. In the meantime, they gain experience in kernel programming, having implemented many important components of the kernel and can verify their implementation using automated test cases provided by instructors.

Pedagogical operating system frameworks can further be categorized into two types: the "simulated" and the "real" [8]. Simulation systems such as Yalnix [56], PortOS [7], and

MOS [50] are typically user space programs that use user-level constructs to simulate OS facilities. For example, PennOS [8] implements interrupts via UNIX signals - configuring the host OS to send a SIGALRM every 10ms, upon which the "interrupt" is serviced in a signal handling routine. Other simulation systems, including Nachos [11], OS/161 [21], JaeOS [38], and PortOS [7], do not directly simulate an OS. Instead, they run on top of an hardware simulator built specifically for it or (in the case of Nachos) compiled directly into the simulator [3, 11, 21]. This provides a higher degree of realism since students can still see their code interacting with hardware, despite the fact that the hardware is simulated. "Real" systems, such as Pintos [44], GeekOS [22], JOS [39], MiniOS [46], BabyOS [34], and Minix [19] can run on a bare metal x86 machine. This has several advantages over simulation, such as giving students an opportunity to learn about computer architecture, see their code interact with real hardware, and gain first-hand experience in hardware-level programming [22]. Furthermore, they still have access to advanced debugging tools available for simulated systems because they can be debugged under PC emulators such as Bochs [31] and QEMU [9], which provide extensive debugging tools to the students.

The OS concepts taught by different operating system courseware frameworks are largely the same. Most cover core OS topics such as scheduling, servicing user processes, virtual memory, and the filesystem - in other words, the same topics covered by Pintos. Some dive deeper into device drivers [20, 46, 50], networking [11, 7, 34, 39], and inter-process communication [22, 34, 39].

While Pintos follows the tradition of other instructional operating systems, it is unique in

two aspects. First, it supports SMP, a trait shared by only one other instructional OS (JOS). While Pintos provides SMP support in the base code, JOS requires implementing SMP as one of its projects. Second, Pintos is the only framework that teaches multiprocessor scheduling, which we have identified as an important learning objective.

Chapter 8

Future Work

Some possible improvements could be made to Pintos to improve its design, teach additional OS concepts, or make the journey less confusing and frustrating for students.

Pintos could be improved by adding support for another flavor of spinlocks; one that disables preemption, but not interrupts, as discussed in section 4.4. This type of spinlock not only minimizes the interrupt latency by allowing interrupts to be handled while spinning for the lock, but also avoids deadlocks like the one we discovered in our first implementation attempt of our TLB shootdown protocol. The deadlock arose if one thread acquired the TLB shootdown spinlock, sent an TLB shootdown IPI to another CPU, and waited for that CPU to acknowledge the interrupt while that CPU is contending for the TLB shootdown spinlock. That CPU will not re-enable interrupts until it acquires the spinlock that the IPI sender is holding, but the IPI sender is waiting for the CPU to service the interrupt. To

avoid this situation, we now protect TLB shootdown using a lock, rather than a spinlock. However, this means that code paths that can potentially lead to invalidating other CPUs' TLB's must not hold a spinlock, as a thread may block while trying to acquire the TLB shootdown lock.

In Pintos, timer interrupts are issued periodically at a predetermined frequency. Pintos could be improved by adopting a tickless approach similar to the one used in Linux [12], where the timer interrupts are not issued periodically but rather set to arrive on-demand. This provides performance improvements by ensuring that no timer ticks arrive unnecessarily. In the design document for Project 1, we asked students to identify how a tickless kernel would have improved their system, but we did not ask them to implement it, nor did we implement such a feature for them.

A multi-threading project could also potentially be beneficial for students, as our SMP-capable Pintos would allow students to observe truly parallel execution and speedup of their multi-threaded applications. This could be a new standalone project, or added as new requirement for Project 2.

Pintos could also benefit from having more accurate CPU statistics. The current Pintos kernel estimates CPU usage by sampling during timer interrupts. That is, at each timer tick, the CPU records whether it was executing an idle thread, a kernel thread, or a user thread at that time. However, it cannot know anything about what thread executed in between timer ticks, leading to extremely coarse usage statistics. As a result, we had to increase the amount of time for which we run the load balancing tests to avoid spuriously

failing correct implementations due to inaccurate statistics. Pintos could be improved by using hardware cycle counters such as the TSC register to more accurately measure CPU usage.

Finally, kernel support for memory and synchronization checkers similar to those available for the Linux kernel [35, 40] could greatly benefit students debugging intermittent race conditions.

Chapter 9

Conclusion

Teaching Operating Systems remains a cornerstone of the undergraduate computer science curriculum. Several educational operating system frameworks exist to provide students with first-hand experience programming in a simplified, but realistic kernel environment. Pintos is one such framework that has been use for several years in many universities throughout the US, including Stanford and Virginia Tech. With extensive documentation, commented code base, well-defined project specifications, and automated test cases, Pintos is very suitable as a supplement to teaching computer science courses.

In our work, we upgraded Pintos to a multiprocessor OS, capable of true multiprocessing. To expose students to scheduling policies and synchronization techniques used in multiprocessor systems, we created a new scheduling assignment that requires students to implement a simplified version of Linux's Completely Fair Scheduler and its associated load balancing

algorithm. We also created test cases to assess students' submissions. Finally, we evaluated the effectiveness of our new Pintos framework in teaching OS concepts using several instruments, including meetings with students, review of their project submissions, a quiz to assess student learning, and a post-survey. Our feedback was positive, showing that students enjoyed working with our new Pintos framework and demonstrated a deeper understanding of OS concepts as a result of completing our project.

Bibliography

[1] Sarita V. Adve and Hans-J. Boehm. Memory Models: A Case for Rethinking Parallel Languages and Hardware. *Commun. ACM*, 53(8):90–101, August 2010.

[2] American National Standard for Information Systems. *Information Technology -AT Attachment with Packet Interface Extension (ATA/ATAPI-4)*, Aug 1998.

[3] Charles L. Anderson and Minh Nguyen. A Survey of Contemporary Instructional Operating Systems for Use in Undergraduate Courses. *J. Comput. Sci. Coll.*, 21(1):183–190, October 2005.

[4] Jeremy Andrus and Jason Nieh. Teaching Operating Systems Using Android. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, SIGCSE '12, pages 613–618, New York, NY, USA, 2012. ACM.

[5] Lab 4 Introduction. `http://compas.cs.stonybrook.edu/~nhonarmand/courses/sp17/cse506/lab4.html`.

[6] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 0.91 edition, May 2015.

[7] Benjamin Atkin and Emin Gün Sirer. PortOS: An Educational Operating System for the Post-PC Environment. In *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '02, pages 116–120, New York, NY, USA, 2002. ACM.

[8] Adam J. Aviv, Vin Mannino, Thanat Owlarn, Seth Shannin, Kevin Xu, and Boon Thau Loo. Experiences in Teaching an Educational User-level Operating Systems Implementation Project. *SIGOPS Oper. Syst. Rev.*, 46(2):80–86, July 2012.

[9] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[10] Hans-J Boehm and Sarita V. Adve. You don't know jack about shared variables or memory models. *Queue*, 9(12):40:40–40:49, December 2011.

[11] Wayne A. Christopher, Steven J. Procter, and Thomas E. Anderson. The Nachos Instructional Operating System. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, USENIX'93, pages 4–4, Berkeley, CA, USA, 1993. USENIX Association.

[12] Jonathan Corbet. (Nearly) full tickless operation in 3.10. `https://lwn.net/Articles/549580/`.

[13] Russ Cox, Frans Kaashoek, and Robert Morris. xv6, a simple, Unix-like teaching operating system. `http://plan9.bell-labs.com/sys/doc/sleep.html`, 1991.

[14] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. dmp: Deterministic shared memory multiprocessing.

[15] Alessio Gaspar, Naomi Boyer, and Abdel Ejnioui. Role of the C Language in Current Computing Curricula Part 1: Survey Analysis. *J. Comput. Sci. Coll.*, 23(2):120–127, December 2007.

[16] gnudocs. Built-in Functions for Memory Model Aware Atomic Operations. `https://gcc.gnu.org/onlinedocs/gcc/_005f_005fatomic-Builtins.html`.

[17] gnudocs. Memory model synchronization modes. `https://gcc.gnu.org/wiki/Atomic/GCCMM/AtomicSync`.

[18] Irfan Habib. Virtualization with KVM. *Linux J.*, 2008(166), February 2008.

[19] Stephen J. Hartley. Experience with minix in an operating systems lab. *SIGCSE Bull.*, 22(3):34–38, August 1990.

[20] Rob Hess and Paul Paulson. Linux Kernel Projects for an Undergraduate Operating Systems Course. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, SIGCSE '10, pages 485–489, New York, NY, USA, 2010. ACM.

[21] David A. Holland, Ada T. Lim, and Margo I. Seltzer. A New Instructional Operating System. In *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '02, pages 111–115, New York, NY, USA, 2002. ACM.

[22] David Hovemeyer, Jeffrey K. Hollingsworth, and Bobby Bhattacharjee. Running on the bare metal with geekos. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '04, pages 315–319, New York, NY, USA, 2004. ACM.

[23] Intel. *8254 PROGRAMMABLE INTERVAL TIMER*, May 1996.

[24] Intel. *Intel 82093AA I/O Advanced Programmable Interrupt Controller (I/O APIC) Datasheet*, May 1996.

[25] Intel. *MultiProcessor Specification*, May 1997.

[26] Association for Computing Machinery (ACM) Joint Task Force on Computing Curricula and IEEE Computer Society. *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. ACM, New York, NY, USA, 2013. 999133.

[27] Brett D. Jones. Motivating students to engage in learning: The music model of academic motivation. 2010.

[28] Eric Koskinen and Maurice Herlihy. Dreadlocks: Efficient Deadlock Detection. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, pages 297–303, New York, NY, USA, 2008. ACM.

[29] Oren Laadan, Jason Nieh, and Nicolas Viennot. Structured Linux Kernel Projects for Teaching Operating Systems Concepts. In *Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education*, SIGCSE '11, pages 287–292, New York, NY, USA, 2011. ACM.

[30] Barry Lawson and Lewis Barnett. Using iPodLinux in an Introductory OS Course. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '08, pages 182–186, New York, NY, USA, 2008. ACM.

[31] Kevin P. Lawton. Bochs: A Portable PC Emulator for Unix/X. *Linux J.*, 1996(29es), September 1996.

[32] Dongyoon Lee, Mahmoud Said, Satish Narayanasamy, Zijiang Yang, and Cristiano Pereira. Offline Symbolic Analysis for Multi-processor Execution Replay. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 564–575, New York, NY, USA, 2009. ACM.

[33] Tong Li, Dan Baumberger, and Scott Hahn. Efficient and Scalable Multiprocessor Fair Scheduling Using Distributed Weighted Round-robin. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 65–74, New York, NY, USA, 2009. ACM.

[34] Haifeng Liu, Xianglan Chen, and Yuchang Gong. Babyos: A fresh start. In *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '07, pages 566–570, New York, NY, USA, 2007. ACM.

[35] Kernel Memory Leak Detector. `https://01.org/linuxgraphics/gfx-docs/drm/dev-tools/kmemleak.html`.

[36] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 37–48, New York, NY, USA, 2006. ACM.

[37] Brandon Lucia, Joseph Devietti, Luis Ceze, and Karin Strauss. Atom-Aid: Detecting and Surviving Atomicity Violations. *IEEE Micro*, 29(1):73–83, January 2009.

[38] Marco Melletti, Michael Goldweber, and Renzo Davoli. The JaeOS Project and the uARM Emulator. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '15, pages 3–8, New York, NY, USA, 2015. ACM.

[39] MIT. JOS OS Lab. `https://pdos.csail.mit.edu/6.828/2016/`, 2016.

[40] Igor Molnar. Runtime locking correctness validator. `https://www.kernel.org/doc/Documentation/locking/lockdep-design.txt`.

[41] Oakbytes. Linux Scheduler CFS and Red Black Tree. `https://oakbytes.wordpress.com/2012/06/08/linux-scheduler-cfs-and-red-black-tree/`.

[42] Ron Obermarck. Distributed Deadlock Detection Algorithm. *ACM Trans. Database Syst.*, 7(2):187–208, June 1982.

[43] Chandandeep Singh Pabla. Completely Fair Scheduler. *Linux J.*, 2009(184), August 2009.

[44] Ben Pfaff, Anthony Romano, and Godmar Back. The Pintos Instructional Operating System Kernel. In *Proceedings of the 40th ACM Technical Symposium on Computer Science Education*, SIGCSE '09, pages 453–457, New York, NY, USA, 2009. ACM.

[45] Rob Pike, Dave Presotto, Ken Thompson, and Gerard Holzmann. Process Sleep and Wakeup on a Shared-memory Multiprocessor. `http://plan9.bell-labs.com/sys/doc/sleep.html`, 1991.

[46] Rafael Román Otero and Alex A. Aravind. MiniOS: An Instructional Platform for Teaching Operating Systems Projects. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, pages 430–435, New York, NY, USA, 2015. ACM.

[47] Alexander Schmidt, Andreas Polze, and Dave Probert. Teaching Operating Systems: Windows Kernel Projects. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, SIGCSE '10, pages 490–494, New York, NY, USA, 2010. ACM.

[48] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: Data Race Detection in Practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, pages 62–71, New York, NY, USA, 2009. ACM.

[49] Vaishaal Shankar and David Culler. A Modern Student Experience in Systems Programming. In *Proceedings of the Second (2015) ACM Conference on Learning @ Scale*, LS '15, pages 233–236, New York, NY, USA, 2015. ACM.

[50] Onkar P. Sharma and Sean Goldsmith. OS Project Implementation: Multiprogramming with Threads. *J. Comput. Sci. Coll.*, 27(3):108–114, January 2012.

[51] C. A. Waldspurger. Lottery and stride scheduling: Flexible proportional-share resource management. Technical report, Cambridge, MA, USA, 1995.

[52] wiki.osdev. 8259 PIC. `http://wiki.osdev.org/8259_PIC`, Sep 2016.

[53] wiki.osdev. APIC. `http://wiki.osdev.org/APIC`, Feb 2017.

[54] wiki.osdev. APIC timer. `http://wiki.osdev.org/APIC_timer`, Jan 2017.

[55] wiki.osdev. Programmable Interval Timer. `http://wiki.osdev.org/Programmable_Interval_Timer`, April 2017.

[56] The Yalnix Kernel. `http://www.cs.utah.edu/~chojs/projects/uofu/CS5460/yalnix.pdf`, 2001.