An Educational Operating System Supporting Computer Security

Patrick Joseph McGee
Marquette University

AN EDUCATIONAL OPERATING SYSTEM SUPPORTING COMPUTER SECURITY

by

Patrick J. McGee

A Thesis Submitted to the Faculty
of the Graduate School, Marquette University,
in Partial Fulfillment of the Requirements for
the Degree of Master of Science

Milwaukee, Wisconsin

May 2020

ABSTRACT
AN EDUCATIONAL OPERATING SYSTEM SUPPORTING COMPUTER SECURITY

Patrick J. McGee

Marquette University

It is uncommon to teach computer security concepts using an embedded operating system. Few educational operating systems are implemented in systems courses, and a small subset of them are used to teach hardware-related security elements such as system calls. This work explores relevant approaches to teaching low-level security concepts, including difficulties associated with building a hands-on learning environment.

This work also presents additions to the Embedded Xinu kernel that support system calls and memory protection on a Raspberry Pi 3 B+. Provided sample assignments are intended to help give students a solid understanding of intricate hardware details. Results from an assignment run in a computer security course show that the system call interface provides an effective way to teach essential computer security measures.

ACKNOWLEDGMENTS

Patrick J. McGee

I would like to extend my deepest gratitude to:

- Dennis Brylow, my advisor, for continuously supporting my research interests,
- Debbie Perouli, whose guidance helped shape this project,
- Tom Kaczmarek, for providing invaluable advice throughout my graduate program,
- Benjamin Levandowski, my friend and fellow Xinu Team member, for devoting his time to review this thesis, offering well-informed suggestions,
- My family and friends, who are always displaying their love and support for me, and
- My little brother, Sean, who inspires me endlessly.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

1.0.1 Thesis Statement

An embedded, educational operating system can effectively facilitate the teaching of computer security concepts at the collegiate level.

1.0.2 Overview

To teach courses such as operating systems and computer security, it is common for instructors to gravitate toward approaches that offer students limited practical involvement, if any at all. This is because it is seen as too difficult for students to learn such concepts in a hands-on manner, or that available resources for hands-on teaching are unfamiliar to the instructor. Many courses employ a theoretical approach in which students spend time solving problems that are only indirectly related to programming physical systems. Other common approaches include examinations of full-size, production operating systems. While students may be subject to writing or modifying code in these cases, the unnecessary overhead – inherited from the size of the operating system – limits the students' scope to a single component or sub-component. This is undesirable since it is ineffective in displaying the interactions between operating system components, and cannot reasonably present hardware-informed design decisions to a student.

A proven alternative is the use of educational operating systems – software created specifically for students to observe and develop components of an operating system. Embedded Xinu is an educational operating system that is publicly available and maintained by researchers at Marquette University's Systems Laboratory. Xinu is programmed to run on real, inexpensive hardware such as a Linksys router or a Raspberry Pi computer. Therefore, universities can create affordable laboratory environment configurations to ease the load; a group of students need not share a single machine to execute their kernel.

This work presents security-oriented contributions to the Embedded Xinu operating system. These additions cover two low-level components:

- System calls, an interface for unprivileged tasks to access privileged operating system services, and
- Memory protection, a configuration that invokes built-in hardware safeguards to shield computer storage from invalid accesses.

This work includes a study of student performance on a system call assignment run in a computer security course at Marquette University (COSC 5360). Also included is a sample assignment for the memory protection facility, and a more complex version of the system call assignment. This work focuses on implementing software that is as concisely-written as possible, enabling students to be less intimidated when reading source files.

The arrangement of this thesis is as follows:

- Chapter 1 introduces this work and explains its contributions,
- Chapter 2 explains foundational terms upon which this thesis will expand,
- Chapter 3 presents a literature review of tools for systems education and relevant design principles,
- Chapter 4 details the implementation of system calls on this platform, including a sample assignment,
- Chapter 5 describes the established memory security structure, including a sample assignment,
- Chapter 6 details the system call assignment run in a cross-listed computer security course,
- Chapter 7 presents an analysis of student performance on the system call assignment, discussing the results, and

- Chapter 8 summarizes this work and describes directions of potential future work.

1.0.3 Contributions

This thesis presents additions to the Embedded Xinu operating system that facilitate hands-on learning of fundamental topics in computer security. Also discussed are the difficulties encountered while implementing new functionality in the kernel. Teaching difficulties and outcomes are discussed, regarding a system call assignment run in a computer security course. The final student performance evaluation shows that it is achievable to teach such concepts to students of various computing disciplines – even those who have never had experience programming an embedded system.

CHAPTER 2

Background

2.0.1 Embedded Xinu

The Embedded Xinu operating system (O/S) descends from Xinu, an educational O/S created in the 1980s by Douglas Comer [10]. Many books about the parent kernel were written by Comer [12], who also maintains a webpage listing the various applications of Xinu both in industry and in systems education [13]. Xinu was originally supported to run on CISC (complex instruction set computer) architectures, such as Intel's x86 platform, that are equipped with instruction sets containing hundreds of operations [14]. A student learning assembly language for a CISC machine only needs to understand a small subset of the available instructions. Xinu mitigated this overhead when it was ported to modern RISC (reduced instruction set computer) platforms, such as the Linksys wireless router [9] and the Raspberry Pi 1 [8]. These ports established Embedded Xinu and distinguished it from its CISC-supporting counterpart.

2.0.2 Development and Deployment

One of the central goals of Embedded Xinu is to expose students to real hardware. Marquette University researchers have been porting the O/S to new platforms, such as the parallel-oriented Intel Single-Chip Cloud (SCC) computer [42]. The most recent port of Embedded Xinu to the Raspberry Pi 3 B+ [6] allows hardware systems [19] and operating systems courses to provide hands-on experimentation in a multicore execution environment. Earlier ports of Embedded Xinu have been used to teach compilers [22] and embedded systems [32]. Until this work, Embedded Xinu was unsuited to teach hardware security since no explicit security facilities had been implemented.

2.0.3 Computer Security

Modern computing standards demand secure operations. The broad range of security topics includes information security, cryptography, and network transmission.

This thesis focuses on the hardware aspect of computer security, i.e., the security facilities that computer hardware provides to the software that is running on it.

2.0.4 Execution Modes

The hardware component of a computer that executes a program's instructions is the central processing unit. After a computer program is compiled, it runs by invoking the CPU, which processes it one instruction at a time. An operating system is an example of a computer program. The O/S offers abstractions called threads that are units of execution that perform a specific task [34]. Modern O/Ses provide multitasking, a feature that enables the illusion that many threads are running at the same time. In actuality, they are constantly being switched, sharing a single CPU's resources. This procedure is commonly referred to as a "task switch" or "context switch" [35].

A CPU executes in a particular mode, which determines *how* its instructions will be executed. Processors have a set of execution modes which the O/S designer can implement. A privileged mode called System mode has unrestricted access to the processor's instruction set. While simple in design, an O/S that performs all of its tasks in System mode – including running user threads – is susceptible to security instability. For instance, a user of such an operating system can run an instruction that modifies a memory space in use by the kernel. This type of modification – whether it was caused by a malicious actor or a system error – would likely cause the system to fault irrecoverably.

Instead of running all tasks in System mode, a more desirable approach is to make use of a processor's unprivileged User mode. User mode implies that the execution environment is restricted. At the most basic level, a processor running in User mode is unable to execute privileged instructions that may interfere with the underlying operating system. A more advanced computer might also support a memory system that can determine – based on the current execution mode – whether a memory access is valid. Invalid accesses cause a fault, and the O/S designer can choose how to deal with it.

2.0.5 System Calls

How does one design an operating system that is both (a) capable of overseeing privileged execution and (b) able to operate user threads securely? System calls, or Supervisor calls (SVC) provide this feature. A Supervisor call is an interface for unprivileged threads to temporarily access privileged O/S resources. While System mode is all powerful, and User mode is limited, Supervisor calls execute in a privileged environment called Supervisor mode. Generally, this mode has the same high privilege as System mode. Supervisor calls are based on instruction set access. Some instructions, such as those that modify CPU registers, are only accessible from a privileged mode. In many cases, instructions can be allowed by the CPU but restricted by the operating system. In this case, it is up to the O/S designer to decide which operations should require Supervisor privilege. A simple O/S, such as an educational one, may only include a few applications of Supervisor calls in its kernel. A production O/S offers more protection, but with high complexity.

2.0.6 Compilation

Just like any computer program, an operating system must be translated to a language that the machine can understand before it can run. This is the job of a compiler. The Embedded Xinu O/S relies on the GNU cross-compiler (GCC) to convert the kernel source code, written mostly in C, to a series of object files. These productions are then linked together to create the final executable O/S image that can be transferred to and run on a supported machine.

The process of transferring a program, such as a kernel, into the main memory of a machine is called loading. The loader is just another program that can be designed to support the specific initial interaction that the CPU expects. For example, a CPU platform (or architecture) may require the first instruction to be loaded into its main memory at a specific address. If this address is not included in the loader script, the CPU will not begin execution.

Compilers such as GCC have a set of optimization options available. Common benefits of optimizing a program include: smaller code size, faster program execution, and fewer branches taken (i.e., perform an instruction reorder such that instead of jumping to an instruction far away in memory, continue executing it in the current space). The compiler is trusted to perform optimizations without changing the meaning of the program, but in some specific cases, optimization can cause undesirable effects, such as modification of local variables.

2.0.7 Tools

This work was completed using the latest port of Embedded Xinu, running on a Raspberry Pi 3 B+. The Marquette University Systems Laboratory ("Systems Lab") provides the environment necessary to remotely execute a kernel on a Raspberry Pi, supporting input and output. This arrangement is explained in Section 6.0.2.

2.0.8 Summary of Background

Embedded Xinu is a public, educational operating system that has offered practical experience to students across systems courses. System calls allow the O/S to manage and secure its resources while providing an interface for user programs to request specific privileged services.

CHAPTER 3

Related Work

Many small operating systems have been applied in education. Recent papers describe uses of educational operating systems at universities. Although few existing papers describe an educational O/S that supports a hardware-focused security curriculum, some publications have described practices in teaching security concepts that are close to the hardware. This chapter discusses the most recent literature relevant to educational operating systems, and offers context about design decisions that shadow interaction between the hardware and the operating system.

3.0.1 Educational Operating Systems

While Marquette University teaches systems courses using Embedded Xinu on real hardware, both physical and virtual platforms make for effective teaching tools. However, when it comes to examinations of hardware-level interactions, virtual platforms inherently fall short. This section compares Embedded Xinu with existing educational operating systems and examines various approaches in design.

3.0.2 Physical Hardware Ports

The use of an educational operating system is an uncommon approach to teach systems concepts, but several existing educational O/Ses share design decisions with Embedded Xinu. Perhaps the most similar O/S is vmwOS, developed by researchers at the University of Maine to teach a graduate-level operating systems course [16]. Like Xinu, vmwOS is written in C, but it runs on the Raspberry Pi 3 Model B, the predecessor of the Pi 3 B+. Both platforms support the same ARM Cortex A-53 architecture. However, vmwOS supports 64-bit mode, while Xinu runs in ARMv7 32-bit mode for the sake of simplicity. In addition, vmwOS activates the Memory Management Unit (MMU), but employs an "honor system" that does not actually protect a user thread's

interaction with the kernel. Chapter 5 of this thesis presents an effective mechanism to prevent a user thread from modifying kernel space.

Another similar O/S is Minix, developed by Andrew S. Tanenbaum at VU Amsterdam [36]. Like the original Xinu operating system, several O/S books have been written, incorporating Minix source code [37]. Like Embedded Xinu, Minix has been ported to the Raspberry Pi and the Intel SCC [20]. Minix is a microkernel; it a small O/S with industry-standard features that have even been used to support platforms such as x86 [40]. Minix's teaching applications are well-documented, and plenty of academic projects have spawned from Minix [41].

3.0.3 Simulated Ports

Embedded Xinu is designed to expose computing students to a real operating system running on real hardware, as opposed to a simulation on a virtual machine. However, there are many cases in which building a full-scale, physical laboratory environment might not be feasible. An instructor may not have the resources required to expand their current teaching approach and attempt to build a physical system robust enough to handle a full class. Such a laboratory environment usually requires maintenance from a group of dedicated researchers and teaching assistants that may not be immediately available in many CS departments. To cater to this need, some educational O/Ses are strictly built to run on simulators. One such O/S is Nachos, which runs on a MIPS architecture simulator [11]. Moving away from RISC O/Ses, other simulator-supported, educational O/Ses are x86-based gemOS [25] and Pintos [7].

Moving into higher levels of abstraction in teaching operating systems, researchers at the University of Nevada in Reno propose an alternative strategy in which O/S operations are simulated by formatted textual statements rather than an architecture simulator [24]. Such a simulation can be a sufficient programming environment to teach operating systems, but under this model, students are spending time experimenting with a language-defined O/S in place of experimenting with real operating system components.

A similar high-level approach is from Calvin College's CaIOS, an educational operating system simulator written entirely in Python [27]. This tool may be especially useful in an environment where students are unfamiliar with writing programs in assembly language or C. CaIOS is designed such that each O/S component is contained in a Python class, which could make it easier for students who don't know C. However, students might become confused since classes imply potential inheritance, which does not apply for many components that it simulates. For example, creating a new instance of a thread class is a somewhat accurate simulation of an important operating system task, but CaIOS also defines classes that simulate hardware components such as the CPU, which complicates the boundary between hardware and software.

3.0.4 Security Education

Educational O/Ses are capable of teaching essential computing concepts, but only a few include principles of system security. This section details recent work aimed at bringing security concepts to a systems course.

3.0.5 MiniOS

In 2015, researchers from the University of Northern British Columbia chose a course model in which students build their kernel (named MiniOS) from scratch, instead of modifying an existing educational O/S to fit their standards for the course [31]. The O/S course at Marquette University also follows this approach; through a series of cumulative C and ARM assembly assignments, students write essential components and end up with a functional kernel. MiniOS runs on an ARM-based, Atmel SAM4S Xplained Pro prototyping board customized to include components that exist on a Raspberry Pi (i.e., NAND flash memory, GPIO header, and an on-board UART). Using a prototyping board inherently ties the development environment to proprietary development tools; MiniOS students can build and run their O/S projects from Microchip's Atmel Studio 6 – a suite related to Microsoft Visual Studio [5] – while Embedded Xinu is built and executed from a Unix command-line.

An operating system's hardware platform determines the span of programmable features. Choosing to design an embedded system (e.g., using a development board as the base and attaching shields) may ease the depth of programming required to run an operating system because extensional components offer an easily-tailored development experience. In contrast, a Raspberry Pi-like computer has immutable features that O/S designers must learn and program accordingly. As this thesis will cover, it is difficult to program a poorly-documented System-on-Chip (SoC). But while prototyping boards eliminate hurdles such as writing unfamiliar device drivers, their associated development environment may abstract away key O/S components such as the bootloader. Embedded Xinu's bootloader is visible to students in a single assembly file, and it makes for a good demonstration when introducing bootloaders in hardware systems and operating systems.

MiniOS's chosen hardware platform favors kernel simplicity. Their board features program memory, a corresponding memory protection unit (MPU), and a bootloader. As one of its first major projects, the MiniOS course introduces User mode and software interrupts (analogous to this work's Supervisor calls). To simplify this part of the project, the authors of MiniOS give the students the instructions necessary to change modes – a teaching philosophy shared with this thesis. Beginning ARM programmers should not be expected to figure out the instructions and setup required to support Supervisor mode, for instance. The subsequent MiniOS project requires students to protect kernel data via the on-board MPU. Because the MPU is already included and configured, students are unable to directly interact with the hardware.

3.0.6 Summary of Related Work

It is a rarity to use an educational operating system to teach security concepts. For operating systems that run on virtual machines, the scope of projects or laboratory experiments is limited to software-level interaction. Existing educational O/Ses that run on an embedded platform enable hardware visibility, but depending on the platform's design, some hardware details (such as those that provide security) may be hidden from the kernel.

CHAPTER 4

Supervisor Calls

This chapter describes the implementation of SVC calls in Embedded Xinu for the ARM-based Raspberry Pi 3 B+. Education-oriented design goals are discussed, as well as development difficulties. The final section of this chapter introduces a sample assignment and considers how it can be modified to better support a given group of students.

4.0.1 Hardware Considerations

When choosing an embedded platform for an educational operating system, it is important to consider the associated goals of the system as a teaching tool. For the O/S to be most effective, the platform's hardware should align with these goals. One such goal may be laboratory integration. With respect to this goal, some desired O/S functionality may even be included as a hardware unit, as is the case with the networking hardware of the Raspberry Pi 3 B+ that allows the loaded operating system to boot over the network [29]. In developing a lightweight O/S for such a platform, the network driver – which occupies approximately 1700 lines of code in the Xinu kernel [23][15] – need not be written to achieve basic usability. If the network driver is not to be used in application, then the desired goal of laboratory integration can be achieved by choosing a platform that supports network booting.

Another example of a logistical system goal is scalability. The amount of individual machines in a laboratory should fit the needs of the courses they are supporting. Thus, cost is an important factor. Table 4.0.1 weighs essential attributes of platforms that have been used to support educational operating systems.

To teach security concepts in a hands-on course, any of these platforms are qualified. Cost considerations are especially important; a laboratory environment supporting educational O/Ses should be functional, but affordable. Purdue University runs the

Platform	Class	Processor	Arch.	Memory	Price
Atmel SAM4S	Dev.	ARM	32	(2MB)	$48
Raspberry Pi 3 B+	PC	ARM A-53	32/64	1GB	$35
Beaglebone Black Rev C	PC	ARM A-8	32	512MB	$60
Linksys WRT54GL Router	PC	MIPS	32	16MB	$50
QEMU	Emu.	–	–	–	$0
Intel Galileo	Dev.	x86	32	(256MB)	$70

Table 4.1: Comparison of common educational O/S platforms

Xinu operating system on the Beaglebone Black [13], a more feature-packed (and more costly) competitor of the Raspberry Pi. This presents a trade-off: while the feature-packed Beaglebone board may provide active, long-term development of the O/S to be implemented, a less costly alternative is more justifiable for a laboratory setting. At 35 USD, the Pi 3 B+ is a cost-effective solution.

Organizational needs (such as scalability in a laboratory environment) will narrow down the contenders. However, in making a final decision on the platform, course material should ultimately determine the required hardware. For example, for a course that covers hardware-level memory protection, it is desirable to choose a platform that (1) has a memory management unit and (2) does not provide abstractions that will hide the interaction between the memory management unit and corresponding software requests. Similarly, if the course includes mode-based protection, instructions should be available on the platform that allow a temporary mode switch to take place, such as `svc` (Supervisor call) or `swi` (software interrupt).

Another important factor is available documentation, adding complexity to the decision. During the latest port of Embedded Xinu to the Pi 3 B+, lack of documentation for its SoC yielded major development challenges. However, since the release of the Pi 3 B+ in 2017, active online communities such as the Pi Bare Metal Forums [28] have provided direct communication between ARM developers and Pi engineers. Especially in initial design stages of an operating system, this official communication medium has saved the Embedded Xinu Team many hours of programming frustration. On the other hand, the Beaglebone boards are extremely well-documented, with an entire Technical Reference Manual available on their GitHub [18].

The Raspberry Pi has proven to be a canonical teaching tool, from introductory computer courses to advanced systems courses. The Pi Foundation has a site dedicated to providing resources to instructors, including instructions to set up a basic laboratory environment [30]. The creators of the board are also involved in education; a recent textbook written by the Raspberry Pi co-creator focuses on teaching computer architecture with the Raspberry Pi 3 Model B, and includes a detailed chapter written by an engineer of its graphics processing unit [39].

4.0.2 Processor Modes

This research evolved from the idea that students should experiment with protection facilities that the hardware provides, such as mode-based protection. A processor's execution mode defines its current state – that is, the method it will use to execute instructions such as `svc`. The Pi 3 B+'s CPU has seven execution modes:

- System mode,
- User mode.
- Supervisor mode,
- Abort mode,
- Fast Interrupt Request (FIQ) mode,
- Interrupt Request (IRQ) mode, and
- Undefined mode.

Before this work, Embedded Xinu only executed in System mode, IRQ mode, or FIQ mode. System mode is a privileged mode; all instructions are valid in this mode. Using System mode for all basic execution provides an elegant design, but the built-in hardware security facilities go unused. When the processor executes in User mode, privileged instructions become *undefined*. Because the SoC of the Pi 3 B+ has no publicly available documentation, it is difficult to determine exactly which instructions

are considered privileged, and which are available in User mode. Supervisor mode, like System mode, is a privileged mode. However, Supervisor mode is an exception mode, and System mode is not. Hence, to enter Supervisor mode (or any exception mode), the processor generates an exception that is caught by Xinu's exception handler.

Supervisor mode has its own banked stack pointer (a register that keeps track of the runtime stack) and link register (a register that holds the address of the instruction to return to when the currently executing function completes) [1]. Per the ARM calling convention, registers `r0-r3` are pushed onto the stack at the time of a function call [3]. When handling a Supervisor mode exception, it is beneficial to use its respective stack to save these arguments. Therefore, its stack is initialized in the bootloader. Figure 4.1 shows an updated Embedded Xinu memory diagram.

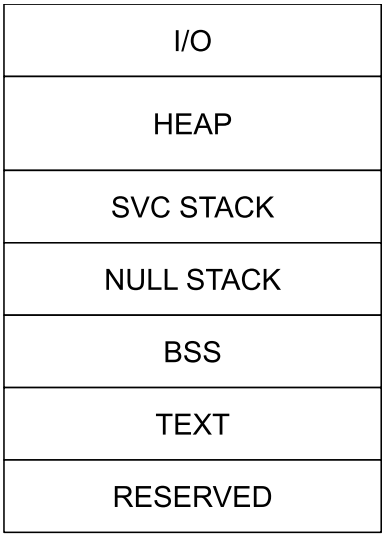

Figure 4.1: Embedded Xinu memory diagram

After the Raspberry Pi powers on, the kernel's bootloader performs initialization steps such as preparing the stack memory for each potential execution mode. When handling an exception such as Supervisor, the platform requires the corresponding stack to be 8-byte aligned, per the ARM procedure call standard [2]. After beginning to execute a function that was branched to by the exception handler, such as a handler routine, the mode switch implies that the stack pointer has been updated. Therefore,

since the handler routine can no longer guarantee alignment of the stack, it is realigned on the proper boundary.

While the Pi 3 B+ is equipped with a four-core processor, this implementation leaves three of the cores disabled. Expanding the kernel becomes a simpler task when there are no issues of multicore concurrency presented. This benefits the developer because unexplored behaviors – regarding system calls across cores – are eliminated. This decision also aligns with Xinu's use as a teaching tool. The student need not be overwhelmed by a multicore system introduced alongside already-new hardware details.

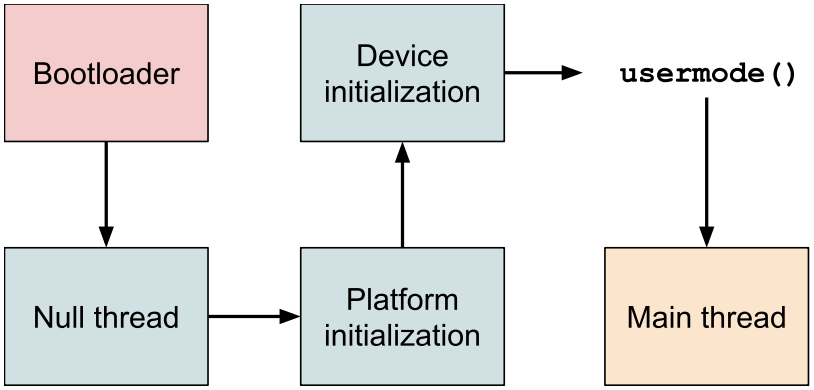

Figure 4.2: Xinu's initialization sequence with system calls enabled

After Embedded Xinu loads the null thread, the processor is still in System mode (see Figure 4.2). Some of the most critical initialization steps follow. To execute the kernel after it is transferred over the network, the Ethernet device must first be initialized. (Of course, network bootable hardware is available on this platform, but because Xinu's `kexec()` [8] function cuts the default boot time in half, the Ethernet driver is initialized). As Section 4.0.6 will explain, interrupts are disabled in User mode. As a quirk of the platform, network communication goes through the USB device. Therefore, the USB subsystem must first be enabled. USB transfers require interrupts to be enabled, so this entire process takes place in System mode. To maintain simplicity, User mode is entered as late as possible – before the main thread begins executing.

4.0.3 SVC Handler

A Supervisor call (SVC) interface, or system call interface, is an abstraction that allows the processor, running in an unprivileged mode, to safely execute privileged instructions and return to normal execution. Figure 4.3 shows the sequence of a system call for `getcpuid()`, a function that returns the numeric identifier of the current processor core.

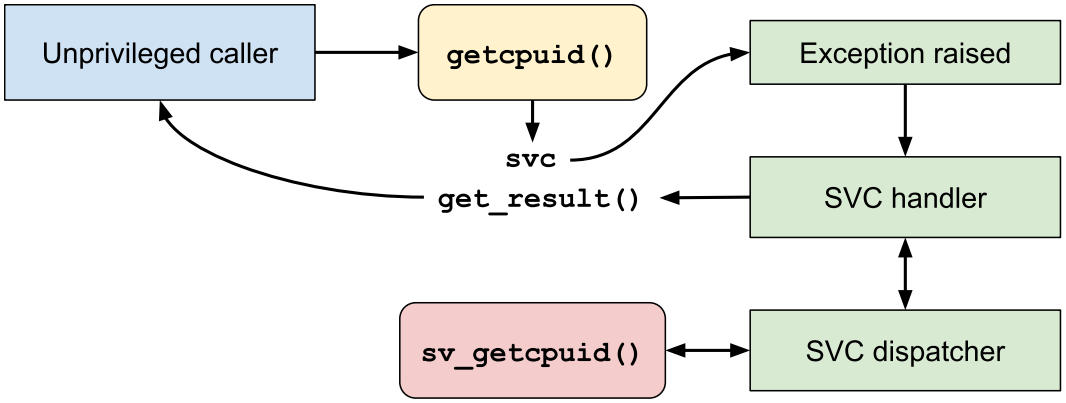

Figure 4.3: Sequence diagram for the `getcpuid()` system call

Before jumping into the SVC handler mechanism, it is necessary to understand how the ARM Cortex A-53 processor operates when switching modes. The Current Program Status Register (CPSR) holds information about the current mode, including execution state and interrupt masks indicating whether the current execution can be interrupted by a running task. The CPSR is accessible in all processor modes. The Saved Program Status Register (SPSR) is only accessible from exception modes such as Supervisor mode. This extra register preserves information about the interrupted mode.

The `svc` instruction is part of both the 32-bit ARM state instruction set, as well as the 16-bit Thumb instruction set. This detail is important when handling the SVC because exceptions generated during Thumb state switch to ARM execution before entering the handler [1]. Further, the processor assumes an ARM execution state and sets the SVC link register (`lr_svc`) to `address_of_svc + 2`, adding a halfword. When an `svc` instruction is executed, the immediate value in its single operand is used to identify the specific service requested. Therefore, to understand the service, it is necessary to be able to extract the immediate value – known as the SVC number – that is sitting in

the `svc` instruction. To differentiate the kernel services provided, each SVC number is uniquely defined in a header file, `include/svc.h`.

```
mrs     r0, spsr          // Save SPSR
tst     r0, #0x20         // If Thumb bit is 1
ldrneh  r0, [lr,#-2]      // then load the halfword
bicne   r0, r0, #0xFF00 // extract 8-bit immediate
ldreq   r0, [lr,#-4]        // Else, load word
biceq   r0, r0, #0xFF000000 // extract 24-bit immediate
// r0 holds SVC_NUMBER
```

Figure 4.4: SVC number extraction

Depending on the state (ARM or Thumb) that the processor was executing in at the time of the instruction, the instruction format will vary, changing the method of accessing the SVC number from the instruction. A series of operations [4] are required to extract the SVC number from this location (see Figure 4.4). After the SVC number is extracted from the instruction, the dispatch call is staged. The SVC dispatcher, written in C, is responsible for both staging and calling the requested service. It takes two arguments: the extracted SVC number and the stack pointer of Supervisor mode. Because the SVC number uniquely identifies a kernel function, a `switch` statement on the SVC number provides a straightforward convention to stage a call.

To differentiate a User mode API from a kernel function, this implementation enforces a naming convention. User function names descend from "normal" function names and are retained. Kernel functions use the prefix `sv_`. For example, if a User task needed to obtain the ID of the currently running thread, it would call `gettid()` (get thread ID). This system call would proceed through the handler. The dispatcher stages a call to `sv_gettid()` which performs the privileged task.

In the dispatcher, after the corresponding `sv_` function finishes executing, the returned result is assigned to a local integer variable that is always returned. It should be noted that the Embedded Xinu kernel defines system errors (`SYSERR`) to be signed integers, thus the SVC dispatcher will return a signed result if the `sv_` function yielded an error. If a `SYSERR` *is* returned, then it must not be handled in the dispatcher.

Rather, because the dispatcher is meant to execute calls as a middle entity, the value must be preserved and returned to the caller.

Because simplicity is a design goal, it is important to use straightforward instructions where possible. In designing a mechanism to return values from the dispatcher to the calling API, many options are available. Unfortunately, because the `svc_call(SVC_NUM)` function must be bound to an assembly instruction (`svc #SVC_NUM`), assignments cannot be made to directly get the result. Instead, one approach is for the dispatcher to store the kernel function result into a global variable, and the API can simply return the global variable. This design has the benefit of familiarity; a student in a computer security course is likely experienced with using global variables to program a feature across separate tasks. But because this work focuses on providing the student with an extensive view of the hardware, it is desirable to use this component as a lesson in consistency with a processor's calling convention. The chosen approach is to preserve the return value in `r0` (in the handler), and use an "empty function" to make the assignment in the corresponding API. This more closely follows the ARM calling convention's use of `r0` as a result register. See Figure 4.5 for an example of a system call API definition. The empty ARM routine, `_get_result()`, is simply defined as `bx lr`. This function leaves `r0` unchanged and allows a normal lefthand assignment of the result.

```
uint getcpuid(){
    uint retval;
    svc_call(SVC_GETCPU);          // svc
    retval = (uint)get_result(); // get r0
    return retval;
}
```

Figure 4.5: Example of a system call API

4.0.4 Difficulties and Lessons Learned

4.0.5 Solving a Tricky Compiler Optimization Issue

Beginning this project, there was a large amount of confidence that the GNU cross-compiler was minimally-configured. In adding major functionality to a an embed-

ded operating system, it is important to square away all assumptions about how the code will run. Failure to do so will surely result in lost time. Much of the initial stage of development was directed at trying to understand a mysterious memory corruption issue that was ultimately resolved by lowering a compiler optimization flag.

Switching out of System mode and into User mode for the first time, results were intermittent. Performing system call tests such as `getmode()` seemed to nullify memory space that had nothing to do with the system call (see Figure 4.6). The initial assumption was that the SVC stack was growing the wrong direction into the BSS segment's space, stepping on global variables. After the issue seemed to disappear, development continued far into the project. Unfortunately, the mysterious memory corruption issue returned. By this time, there was an established set of assumptions in the kernel based on the initial tests from when the issue went away, but all were offset when the issue returned.

initialize.c (test):

```
usermode();
uint *myptr;
myptr = malloc(4);
kprintf("My pointer is: 0x%X\r\n", myptr);
kprintf("My CPUID is: %d\r\n", getcpuid()); // svc
kprintf("My pointer is: 0x%X\r\n", myptr);
```

Output:

```
My pointer is: 0x835820
My CPUID is: 0
My pointer is: 0x0
```

Figure 4.6: Compiler optimizations modify a variable after `svc`

Examining the assembled `.elf` image using the ARM Library's `object-dump` binary utility, memory modifications were not immediately apparent. Luckily, the next test that came to mind was to insert a compiler directive that disables optimizations. When developing system call APIs, it was speculated that the established calling convention might cause the compiler to decide that the order of execution between the `svc` instruction and the following `_get_result()` does not matter. Thus, `__nopt` was defined (see Figure 4.7), turning optimization off for that function.

```
#define __nopt __attribute__((optimize("0")))
```

Figure 4.7: Compiler optimization attribute

After placing `__nopt` in the function definition that caused memory problems, the memory issue was resolved. This begs the question, "what is the relationship between Supervisor calls and compiler optimization?" Examining the Makefile, an optimization flag `-Os` was included. This flag performs many optimizations that shrink the kernel size. Changing this flag back to the default, `-O0` (Optimization level zero) [17], tells the compiler to perform as little optimizations as possible, and the memory issue was resolved. The former compiler optimization is beneficial to have in regular cases, but a working kernel is better than a broken one that is slightly smaller.

4.0.6 Interrupts

On an embedded operating system that uses an on-board device such as a USB (Universal Serial Bus) device, a mechanism must be in place to facilitate communication between the O/S and the device. For many devices, an interrupt request line is the necessary medium of communication. Upon initialization, the operating system will enable a request line for each device it will interface with. Prior to this work, the Xinu kernel enabled interrupts for many devices – most notably, the UART (Universal Asynchronous Receiver-Transmitter) and the USB. UART interrupts allow the receiver to transfer data. This is useful in system functions that get input, such as `getc()`. However, without interrupts enabled, a User mode function is not able to get input.

As explained in 4.0.2, the critical initialization sequence can occur in System mode. However, when the first context switch occurs from User mode, the `msr` (Move to Status from Register) instruction, which updates the CPSR's interrupt bits, is undefined. This is unfortunate because without access to this instruction, the new thread cannot be started with interrupts enabled. A circumvention of this issue was attempted by issuing an `svc` instruction from the context switch routine. Subsequently, after changing the SPSR in the SVC handler, interrupts were enabled. However, upon return from the

`svc`, the system timer immediately interrupted the context switch code. In its current state, the clock handler (called following a system timer interrupt) is unable to execute in User mode. It was decided that, for the sake of time and simplicity, interrupts need not be enabled to achieve what is intended.

4.0.7 Advanced Sample SVC Assignment

The sample assignment proposed in this section is a more advanced version of the one described in Chapter 6. This assignment involves ARM programming and more advanced C programming, allowing more creative freedom.

Students are given a kernel that boots into User mode, but does not yet initialize the Supervisor mode stack in the bootloader. They are tasked with (1) setting up the Supervisor mode stack and (2) implementing a system call for `create()`. This function takes a variable amount of arguments (`...`), making both its API and dispatch cases more challenging to write.

The amount of arguments that will be tested is unknown to the students, requiring them to write an elegant C implementation to receive full credit. Written test cases for this assignment might include readying created threads that perform an argument check. How students think about checking arguments can vary; for example, a summation of a set of numbers would suffice as long as the expected result is returned. Another type of test can be automated, checking arguments operationally.

Further tests can be conducted by calling `create()` multiple times, readying the new thread each time. This type of stress test can determine whether the student's dispatch case properly accesses the variable arguments that were created. To make the assignment even more difficult, a broken SVC handler could be provided such that students need to push argument registers onto its stack and stage the call to the dispatch function.

4.0.8 Summary of Supervisor Calls

An educational operating system should be implemented on a hardware platform that aligns with material for the course. The chosen platform defines the extent to which hardware security protocols, such as system calls, can be implemented. Adding system calls proved to be challenging, but a simple implementation was achieved, allowing assignments to spawn from this work.

CHAPTER 5

Memory Protection

This chapter describes the platform's memory system and the changes required to provide mode-based memory protection. This chapter ends with a detailed explanation of an introductory sample assignment using this work.

5.0.1 Memory Management

5.0.2 Background

The Memory Managemeny Unit (MMU) is a functional unit that oversees the processor's use of available physical memory. Memory management units can be configured to implement memory protection. A cache is an intermediary between the processor and physical memory. For a unit that supports cacheing, regions marked as cacheable are saved accordingly in a Translation Lookaside Buffer (TLB). Figure 5.1 shows a high-level MMU interface.

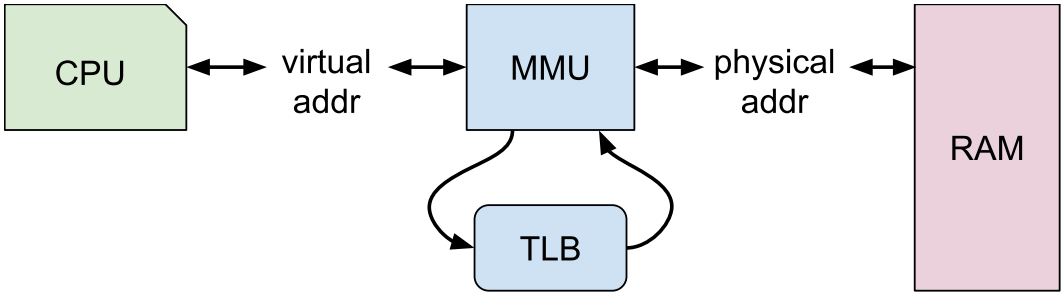

Figure 5.1: Generalized MMU interface

The arrows represent the flow of a memory request, from the processor (requester) to the main memory, and back. The requested memory address is first examined by the MMU to ensure that it is valid. To accomplish this, the MMU examines information stored in a hardware register that contains read and write permissions for a given region. After determining that the address is accessible, an address translation is performed. These translations are stored in the TLB, so the MMU simply references this buffer and grabs the corresponding physical memory address. If a store was performed,

then the memory system need not return a value. However, if a load operation was performed, the request is updated to include the value located at the physical address, and it sent back to the requester.

5.0.3 Enforcing Basic Protection

The Raspberry Pi 3 B+ supports two virtual memory system options: the MMU and the Memory Protection Unit (MPU) [1]. The MPU provides a simpler interface for programming regions, but because Embedded Xinu already enables the MMU, the MMU was chosen for this implementation. This decision is better aligned with hardware systems teaching; the MMU is a necessary computer component that students will be familiar with, while the MPU is more of a specialty hardware unit offered by some systems.

The Pi 3 B+ defines two levels of cache: L1 and L2. By default, L2 cache is disabled on the Rapsberry Pi 3 B+ [1]. The most recent port of Embedded Xinu initializes the MMU, but only uses the enabled L2 system for atomic operations, enforcing mutual exclusion across cores [6]. Prior to this work, the enabled MMU did not make use of its memory protection capabilities.

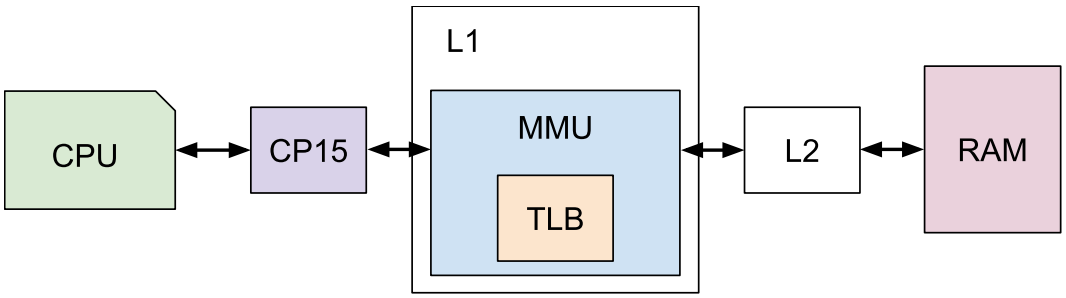

Figure 5.2: Pi 3 B+ memory hierarchy

Figure 5.2 shows the memory system implemented by this device. When the MMU is initialized, the L2 cache is enabled and translations can begin. As described earlier, a hardware register is required to determine access control. On this platform, coprocessor register 15 (CP15) holds information about the memory region being accessed, such as whether it is cacheable, or whether the current execution mode is allowed

to access it [1][21]. Following the address translation, the physical address may reside in either of the two cache hierarchies, or in RAM.

This kernel configures the MMU in such a way that the translation between virtual and physical translations is 1:1. This makes for a simpler design in which translated memory addresses are visible to the requester. The platform's memory architecture allows memory regions to be programmed to one out of many various fixed sizes: 4KB page, 64KB page, 1MB section, or 16MB supersection [1]. The latest kernel port uses 1MB sections, without subpages, for simplicity of design. 1MB sections allow for a concise initialization of the MMU, making the code simpler for a student to understand. This implementation keeps 1MB sections.

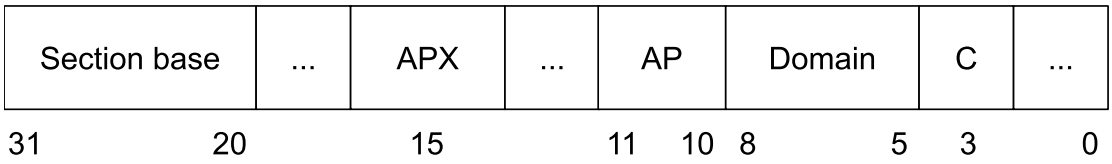

Figure 5.3: Simplified descriptor format for a 1MB section

MMU regions are defined by translation table descriptors. The chosen 1MB section size uses a first-level descriptor format, as shown in Figure 5.3. When the C bit is set, the 1MB region is cacheable. This detail is important when considering peripheral address space; this region's C bit must be disabled or profound effects may occur even with basic output. 12 bits of the descriptor are reserved for the base address of the section. The APX bit allows an extension of the normal access privilege (AP) bits. Instead of four states for privileged and user access control permissions, when the APX bit is set, two extra privilege states are made available, such as privileged read only. This implementation does not set the APX bit, and makes use of either *full access* mode, or *user write disabled* mode using AP bits, as shown in Section 5.0.4. The domain format defines the access type over a collection of memory regions, allowing more flexibility in advanced kernels. For simplicity, in this implementation, a single domain is defined for the entire set of 1MB sections. The previous port of Embedded Xinu set the domain bits to the Manager access type. This prevented accesses from being checked against the access permission bits in the descriptor, thus a permission fault was never generated.

Therefore, to enforce a level of memory protection, it is not only necessary to change the AP bits, but also the domain bits. Changing these bits to Client Mode, accesses are now checked against the AP bits in the entry.

Embedded Xinu's MMU initialization code is written in C and ARM assembly. The `mmu_init()` C function oversees the general initialization of regions using a helper function, `mmu_section()`, to set appropriate descriptor format bits. A for loop cycles through memory addresses, calling the helper. Once all of the section base addresses have been mapped and the peripherals are marked as uncacheable, the MMU can be started. ARM routine `start_mmu()` performs necessary operations through CP15. First, all instruction and data cache lines are invalidated. Before the domain and TLB base is set, the TLB is invalidated to prevent invalid references leftover from a hardware reset.

After a restricted access permission (i.e., User mode read-only) is applied to a region and the MMU is enabled, Abort mode becomes a possible execution mode. As introduced in section 4.0.2, Abort mode is an exception mode entered upon a memory abort. Memory aborts are caused by invalid data memory accesses For example, if a variable exists at memory address `0x00300`, and this region is defined by the first-level descriptor to be User mode read-only, then an attempt to change the value at `0x00300` from User mode would cause a data memory abort. The abort handler can be programmed to deal with such a fault in a certain way. Depending on the handler's design, it might attempt the faulting instruction again after performing a dynamic configuration. Otherwise, a simple abort handler will skip the faulting instruction.

5.0.4 Introductory Sample Assignment

This section presents a sample assignment using the changes described in this chapter and in section 4.0.3. In this sample assignment, students are presented with a main program that creates and runs two threads, in order. The first thread that runs simulates a normal thread, which makes a few system calls (e.g., `gettid()`, `getcpuid()`) and prints each result. The second thread simulates a malicious assignment of a global

variable that causes the kernel to fault irrecoverably. The class is tasked with preventing the fault from occurring, without removing or changing the rogue thread's instructions.

```
void rogue_thread(void){
  kprintf("Attempting to modify core zero's readylist...\r\n");
  readylist[0] = -1;
  kprintf("Readylist modification attempt complete.\r\n");
}
```

Figure 5.4: Example of a memory write to be protected

This sample assignment contains three major parts:

- Making the `insert()` function a system call,
- Modifying the access permission bits during the `mmu_init()` routine, and
- Writing a simple Abort mode handler to skip the invalid instruction.

The global thread ready list is modified by `insert()`, which enqueues the ID of the readied thread into the ready list. Assigning the list index to `-1` will cause the scheduler to malfunction, and a kernel error message is printed. Execution will not continue after the error message is printed. Students can begin by thinking about the protection provided by execution modes. If `insert()` is executed in Supervisor mode, then the ready list will only ever be written from the privileged mode. Therefore, if the address of the ready list is known, the MMU can be configured to only allow write access to the address from a privileged mode.

This implementation uses a scheme to single out the ready list, placing it in a known location away from kernel memory. First, to make the ready list address known, a modification is required. In `initialize.c`, where the ready list queue is declared, a compiler directive – together with a loader script modification – can be used to assign the ready list to a known location (see Figure 5.5). Because the MMU uses 1MB sections with 1:1 translations, the ready list can be defined at a high physical section such as `0x00800000`.

initialize.c:

```
qid_typ readylist[NCORES] __attribute__((section(".readySection")));
```

loader script:

```
.readySegment 0x00800000 : {KEEP(*(.readySection))}
```

Figure 5.5: Placing the ready list at a physical address using the loader script

With the ready list at a known location that is visible to students, they can begin modifying mmu_init() to use the `USER_READONLY` access permission bits. Initially, both AP bits are set, allowing full read and write access to both privileged and unprivileged modes. The modification requires only the second AP bit to be set, granting `USER_READONLY` access for the region holding the ready list.

Once this is complete, the student will notice that the kernel hangs after attempting to make the now-invalid assignment. This is not the desired behavior; the description calls for the invalid instruction to be skipped. Some ARM assembly is required for the next part. The kernel is hanging because it is stuck in Abort mode, due to the fault at the time of the attempted memory write. Students must modify the `abort_handler` to *skip* the faulted instruction. When the processor enters Abort mode from a data fault, the link register is set to `address_of_aborting_instruction` + 8 [1]. Therefore, to return to the instruction that follows the invalid assignment, the abort handler would have to subtract 4 from the link register and branch to it.

5.0.5 Summary of Memory Protection

The MMU provides mode-based memory protection for its programmed regions. This is done by configuring the access privilege bits during initialization of the MMU. A sample assignment shows the practicality of mode-based hardware security. This assignment gives the student a full view of what the MMU is doing and exemplifies an important role of the hardware: to keep the kernel running when malicious instructions enter the kernel.

CHAPTER 6

Teaching With This Platform

This chapter details a version of the proposed system call assignment that was run in a computer security course, including difficulties that students had with the assignment.

6.0.1 Supervisor Call Assignment

In teams of two, computer security students worked to complete a 1-week assignment that involved writing system call APIs. The course was comprised of 30 students: 12 took the course for graduate-level credit, and the other 18 were undergraduates. During the week of the assignment, lectures covered hardware-level security topics such as common kernel protection techniques. While many of the students had programmed on Embedded Xinu in a prior Marquette course, some students had not seen it before. This assignment was written in such a way that it did not require much prior C experience to solve it.

Leading up to the assignment, students had experience programming security concepts in general-purpose languages. Before the system call assignment, their most recent C assignment, a hands-on SEED Lab [33], applied theories in RSA encryption. The system call assignment followed a buffer overflow project [38], implemented in Java.

6.0.2 Laboratory Environment

The Marquette University Systems Lab provides students with a set of Raspberry Pi backend machines. After compiling their kernel, students upload their bootable image to a Raspberry Pi by running a command from a Unix machine. An available backend machine is selected from the pool, and the machine boots over TFTP, a network protocol used to transfer files. Each Raspberry Pi is connected to a rebooter unit and a serial port aggregator (see Figure 6.1).

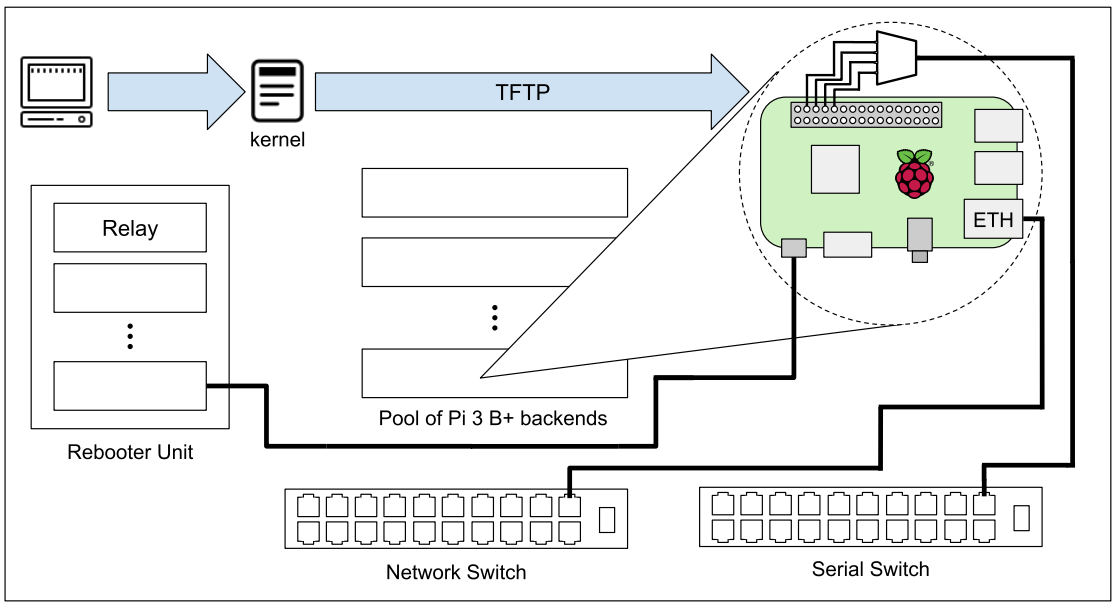

Figure 6.1: Marquette's Systems Laboratory Environment

6.0.3 Implementation: System Call API

The students were given a document that introduced the assignment and detailed their task of writing four system call APIs. The functions were `getcpuid()`, `gettid()`, `malloc()`, and `free()`. The task sheet listed the files necessary to edit and read (see Figure 6.2), explained the relationship between the files, and described how to test their system calls.

Files to edit:

```
include/svc.h
svc/svcdefs.c
system/testcases.c
system/svc-dispatch.c
```

Files to read:

```
svc/setmode.S
system/gettid.c
system/getcpuid.S
system/svc-handler.S
mem/malloc.c
mem/free.c
```

Figure 6.2: SVC assignment: given files to edit and read

Throughout the stripped-down kernel template that was provided, there was a set of `TODO` statements explaining how files should be changed to support the system call

API. For example, a sample block of code was provided for the `getcpuid()` interface, similar to Figure 4.5 in Section 4.0.3, but the API is empty and ready for them to implement.

6.0.4 Evaluation Questions

While evaluating code can help gauge a student's understanding of the task at hand, this is a less stable approach regarding a brand new assignment. There are substantial uncertainties about the programs that will be produced. When creating this assignment, it was unclear whether standalone evaluation of the code will be an effective way to measure how well students understood system calls. Therefore, to create a more pedagogically sound evaluation platform, the assignment description also required students to answer five questions, as listed in Figure 6.3.

1. What do you observe when a Supervisor mode function (such as sv_getcpuid()) is called after the processor enters User mode? Why?

2. What do you observe when a User function (such as getcpuid()) is called before the processor enters User mode? Why?

3. Explain what happens when you try to compile your kernel in the following scenario: A User function (such as getcpuid()) is called after the processor enters User mode, but before you created its API definition.

4. On what core (0−3) is the testcases() thread running?

5. Using malloc(), allocate 4 bytes of memory once. What is the value of the pointer, in hexadecimal, returned by malloc?

Figure 6.3: Evaluation questions for the system call assignment

These questions were meant to target different parts of the assignment. Some questions have less to do with the API implementation and more to do with general systems programming. Questions 1 and 2 test their understanding of their API implementation for the `getcpuid()` system call. `sv_getcpuid()` invokes a coprocessor register to return the value, thus this instruction is undefined in User mode, as Section 4.0.3 establishes. Question 3 tests their C knowledge; if they did not write a definition for `getcpuid()`, the kernel will not compile. This question is also meant to better

differentiate the similarly-named functions `getcpuid()` and `sv_getcpuid()`. Question 4 tests their implementation of the most basic system call, `getcpuid()`. It should be noted that while Section 4.0.2 explains that a single core is used, it is not clear to the student which core is being used – the code to initialize the core was included in the given template as a pre-compiled, non-human-readable object file – *not* a C source file. Question 5 tests their `malloc()` system call, which is perhaps the most difficult. It must take an argument, stage the call after grabbing the argument from the stack (performing appropriate casting), and return the result.

6.0.5 Difficulties

The most common issues from this assignment came from semantics of the system call mechanism. For instance, some students asked about the SVC number definitions in the header file, confused about how many SVC numbers were required. Of course, a single SVC number is defined for a system call. Another question was asked about the necessity of `_get_result()`, as it was presented as an "empty function" used for assignments of the SVC dispatcher's return value. The confusion stemmed from the ARM calling convention's use of r0 as the result register.

During the assignment, a compile error affected a handful of students. The error, "`cannot represent SWI relocation`", was especially confusing to students because they were not sure what "SWI" meant; this assignment only uses the term "SVC" to describe a software interrupt, to be consistent with the platform. This error was caused by a typo in the usage of an SVC number definition.

Other questions were specific to the kernel. Some students had noticed that `gettid()` returns a `tid_typ`, while `getcpuid()` returns an unsigned integer. They were wondering about the difference between the two types. `tid_typ` is actually typedef'd to `int` in Xinu's `stddef.h` header file, which other students were able to find by using Unix command line tools such as `grep`.

6.0.6 Summary of Teaching With This Platform

A system call assignment was run in a computer security course. The assignment was created with the intention of it not being too difficult while still presenting a new challenge to the class. Along with programming, students were required to answer a series of evaluation questions to help quantify their knowledge of what they programmed. The difficulties students experienced were typical for an embedded programming assignment.

CHAPTER 7

Outcomes

7.0.1 Written API Evaluation

To evaluate the written system call interfaces, a set of automatic grading scripts (based upon the Xest tool [26]) provided nightly feedback to students as they built them. At the deadline, final submissions were checked against the same set of nightly test cases. This makes for simpler grading; if a student passed all test cases, then their written solutions are guaranteed to function as expected. After examining the script-generated result, the code was manually checked (by a human) to determine whether the style of programming was adequate.

The programming portion was intended to be the most difficult part of the assignment. However, students performed exceptionally well. Out of 30 students, 26 received full points. The few who did not receive full points came very close. While it is difficult to predict performance when creating a new assignment, this result is not completely unforeseen. This good performance can be attributed to the straightforward nature of the programming required. Working closer to the hardware, students see how elaborate details can affect the way they write programs. In this case, because each system call interface is built almost the same way, if the student was able to solve one, it seems the student had no difficulty writing the rest. Thus, most creative freedom in this assignment is traded off for exposure of the realities of embedded programming.

7.0.2 Analyzing Responses to Evaluation Questions

While examining code can be a reliable way of measuring performance, a more accurate picture can be painted by examining responses to the proposed evaluation questions. This work is about providing students with a deeper understanding of the hardware, but it is worthwhile to examine the possible advantages that students may have held over others.

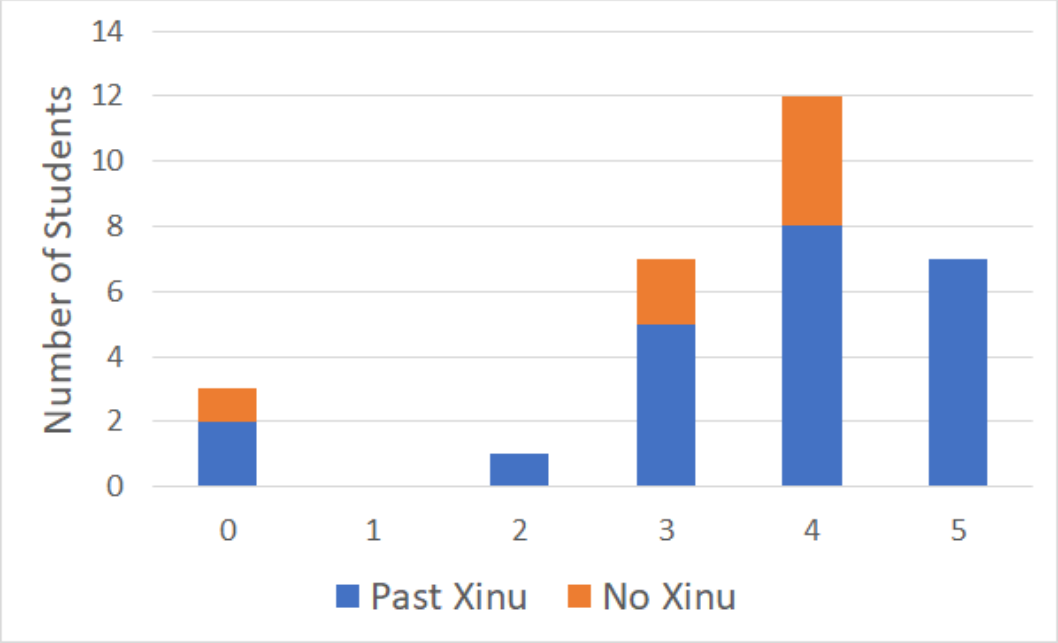

Figure 7.1: Evaluation question performance by prior Xinu experience

The significant advantage in this assignment comes with a student's prior familiarity with Embedded Xinu. Whether Xinu was taught at a high level or the student actually spent time programming Xinu (in operating systems, for example), this qualifies as prior Xinu experience. Figure 7.1 shows the number of questions answered correctly by students (zero being no submission, five being a perfect score), split by prior Xinu exposure.

7.0.3 Discussion of Results

The only students who answered every evaluation question correctly had prior experience with Xinu. However, those who have never seen Xinu before also exhibited a good understanding of the concepts. The most common evaluation question answered incorrectly was question 2. Some students did not describe the correct behavior. Others described correctly that the User mode function can execute in System mode, but did not have the proper reasoning. System mode has full privilege over the instruction set, therefore it must be allowed to execute the task that we call a "User mode function".

In theory, a preliminary survey could have been conducted asking whether the student has had prior Xinu experience. But to increase the difficulty of the assignment based on its responses would be unfair to those who have never seen Xinu before.

7.0.4 Summary of Outcomes

After evaluating performance on the system call assignment, students programmed solutions that received full credit, for the most part. The responses to the evaluation questions yielded more interesting results, showing that students who answered all questions correctly had prior familiarity with Xinu. However, students without prior Xinu experience also performed well. While the assignment did not prioritize creative freedom, it did provide students with a hands-on learning experience about hardware-level security.

CHAPTER 8

Summary and Future Work

8.0.1 Summary

Common approaches in teaching computer security are impractical, or are too complicated to be effective. An embedded platform can offer hardware units that provide a valuable, hands-on learning experience. Recent work has shown a growing interest in using educational operating systems to teach computer security. Few of these systems are taught on a real hardware platform.

This work introduced system calls, a major structural change to the Embedded Xinu operating system. An implementation of mode-based memory protection was also provided. Sample assignments proposed ways that this work can be used in practice. A system call assignment was run in a computer security course, and promising data showed that this work can be used as a foundation for learning elaborate security concepts.

8.0.2 Future Work

Because this work involved adding new hardware-related features to the kernel, plenty of directions of future work involve higher-level expansions. In the future, nested SVC calls can be researched. It is unclear whether they are supported on this platform, but the ability to nest an `svc` instruction would simplify the Xinu kernel, especially if a separate user and kernel space is desired.

Another point of future work is to narrow down the tricky GCC optimization included by the `-Os` flag. If the specific optimization is disabled, then the kernel size can still be reduced by the other optimizations enabled by the flag.

A significant direction of future work is a full integration of system calls and memory protection into the kernel, paving the way to a full release. The MIPS port

of Embedded Xinu defined a platform-specific scheme of memory protection. Perhaps much of its code can be modified to support this platform's use of the MMU for memory protection. Further, a full integration implies that the main thread should be protected from the kernel space. With that, interrupts should be enabled to provide basic interfacing with the shell and other devices. The Ethernet driver should also function properly from User mode. Finally, the other three cores would have to be enabled, but this could be simpler than it sounds, especially if the integration is tested on a single core first.

Bibliography

[1] ARM Ltd. *ARM Architecture Reference Manual Armv8, for Armv8-A architecture profile*. 2019. URL: https://static.docs.arm.com/ddi0487/ea/DDI0487E%5C_a_armv8%5C_arm.pdf?%5C_ga=2.216594262.1753366973.1581287042-1117839992.1566109547.

[2] ARM Ltd. *Eight-byte Stack Alignment*. URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka4127.html.

[3] ARM Ltd. *Procedure Call Standard for the Arm Architecture*. URL: https://static.docs.arm.com/ihi0042/g/aapcs32.pdf.

[4] ARM Ltd. *SVC Handlers*. URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0203j/Cacdfeci.html.

[5] Atmel Corporation. *Atmel Studio 6.1 Release Notes*. URL: http://ww1.microchip.com/downloads/archive/AStudio61readme.pdf.

[6] Priya Bansal et al. "XinuPi3: Teaching Multicore Concepts Using Embedded Xinu". In: *CSERC '17: Proceedings of the 6th Computer Science Education Research Conference* (Nov. 2017), pp. 20–25.

[7] Ben Pfaff. *Pintos*. URL: https://web.stanford.edu/class/cs140/projects/pintos/pintos.pdf.

[8] Eric Biggers et al. "XinuPi: Porting a Lightweight Educational Operating System to the Raspberry Pi". In: *2013 Proceedings of the Workshop on Embedded and Cyber-Physical Systems Education*. 2013. URL: http://www.mscs.mu.edu/~brylow/papers/BiggersHarunaniMuchBrylow-WESE2013.pdf.

[9] Dennis Brylow. "An experimental laboratory environment for teaching embedded operating systems". In: *Proceedings of the 39th SIGCSE technical symposium on Computer science education*. SIGCSE '08. New York, NY, USA: ACM, 2008, pp. 192–196. ISBN: 978-1-59593-799-5. DOI: 10.1145/1352135.1352201. URL: http://doi.acm.org/10.1145/1352135.1352201.

[10] Dennis Brylow. *Embedded Xinu*. URL: https://xinu.mscs.mu.edu/Main_Page.

[11] Wa Christopher and Sj Procter. "The Nachos instructional operating system". In: *the USENIX Winter 1993* (1993), pp. 1–15. URL: http://portal.acm.org/citation.cfm?id=1267303.1267307.

[12] Douglas Comer. *Operating System Design: The Xinu Approach*. 2nd. CRC Press, 2015.

[13] Douglas Comer. *The Xinu Page*. URL: https://xinu.cs.purdue.edu/.

[14] Intel Corporation. *Intel Architecture Instruction Set Reference*. URL: https://software.intel.com/en-us/download/intel-architecture-instruction-set-extensions-programming-reference.

[15] Eric Biggers. *SMSC LAN9512*. 2013. URL: https://embedded-xinu.readthedocs.io/en/latest/arm/rpi/SMSC-LAN9512.html.

[16] P. Francis-Mezger and V. M. Weaver. "A Raspberry Pi Operating System for Exploring Advanced Memory System Concepts". In: (Oct. 2018). URL: http://web.eece.maine.edu/~vweaver/projects/vmwos/2018_memsys_os.pdf.

[17] GNU Project. *Options That Control Optimization*. URL: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.

[18] Jason Kridner. *System Reference Manual*. URL: https://github.com/beagleboard/beaglebone-black/wiki/System-Reference-Manual.

[19] Benjamin Levandowski, Despoina Perouli, and Dennis Brylow. "Using Embedded Xinu and the Raspberry Pi 3 to Teach Parallel Computing in Assembly Programming". In: *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2019, pp. 334–341.

[20] Niek Linnenbank. PhD thesis. VU Amsterdam, 2011.

[21] ARM Ltd. "ARM ® Cortex ® -A53 MPCore Processor Technical Reference Manual". In: (2014). URL: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0500d/DDI0500D_cortex_a53_r0p2_trm.pdf.

[22] Adam B Mallen and Dennis Brylow. “Compiler construction with a dash of concurrency and an embedded twist”. In: *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. SPLASH ’10. New York, NY, USA: ACM, 2010, pp. 161–168. ISBN: 978-1-4503-0240-1. DOI: 10.1145/1869542.1869568. URL: http://doi.acm.org/10.1145/1869542.1869568.

[23] Marquette University Systems Lab. *Pi 1 Ethernet Driver*. URL: https://github.com/xinu-os/xinu/tree/master/device/smsc9512.

[24] Cayler Miley and Michael Leverington. “A Hands-on Approach to Operating Systems through Simulation Projects”. In: *J. Comput. Sci. Coll.* 32.2 (Dec. 2016), pp. 28–33. ISSN: 1937-4771.

[25] Debadatta Mishra. “GemOS: Bridging the Gap between Architecture and Operating System in Computer System Education”. In: *Proceedings of the Workshop on Computer Architecture Education*. WCAE’19. Phoenix, AZ, USA: Association for Computing Machinery, 2019. ISBN: 9781450368421. DOI: 10.1145/3338698.3338887. URL: https://doi.org/10.1145/3338698.3338887.

[26] Matthew H. Netkow and Dennis Brylow. “Xest: An Automated Framework for Regression Testing of Embedded Software”. In: *Proceedings of the 2010 Workshop on Embedded Systems Education*. WESE ’10. Scottsdale, Arizona: Association for Computing Machinery, 2010. ISBN: 9781450305211. DOI: 10.1145/1930277.1930284. URL: https://doi.org/10.1145/1930277.1930284.

[27] Victor Norman. “CaIOS: An Educational Operating System Written in Python”. In: *J. Comput. Sci. Coll.* 33.1 (Oct. 2017), pp. 90–91. ISSN: 1937-4771.

[28] Raspberry Pi Foundation. *Bare Metal, Assembly Language Forum*. URL: https://www.raspberrypi.org/forums/viewforum.php?f=72.

[29] Raspberry Pi Foundation. *Network booting*. URL: https://www.raspberrypi.org/documentation/hardware/raspberrypi/bootmodes/net.md.

[30] Raspberry Pi Foundation. *Teachers’ Guide to Raspberry Pi*. URL: https://www.raspberrypi.org/learning/teachers-guide/.

[31] 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. Kansas City, Missouri, USA: Association for Computing Machinery, 2015, pp. 430–435. ISBN: 9781450329668. DOI: 10.1145/2676723.2677299. URL: https://doi.org/10.1145/2676723.2677299.

[32] Paul Ruth and Dennis Brylow. “An experimental Nexos laboratory using Virtual Xinu”. In: *Proceedings of the 2011 Frontiers in Education Conference*. FIE ’11. Washington, DC, USA: IEEE Computer Society, 2011, S2E–1–1–S2E–6. ISBN: 978-1-61284-468-8. DOI: 10.1109/FIE.2011.6143069. URL: http://dx.doi.org/10.1109/FIE.2011.6143069.

[33] SEED Security Labs. *RSA Encryption and Signature Lab*. URL: https://seedsecuritylabs.org/Labs_16.04/Crypto/Crypto_RSA/.

[34] Abraham Silberschatz, Peter B Galvin, and Greg Gagne. *Operating System Concepts*. 7th ed. J. Wiley & Sons, 2005, p. 139.

[35] William Stallings. *Operating Systems Internals and Design Principles*. 7th ed. Pearson, 2012, p. 139.

[36] Andrew S. Tanenbaum. *MINIX 3*. URL: http://www.minix3.org/.

[37] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating Systems (2nd Ed.): Design and Implementation*. USA: Prentice-Hall, Inc., 1997. ISBN: 0136386776.

[38] Towson University. *Buffer Overflow - CS2 Java*. URL: http://cis1.towson.edu/~cyber4all/modules/nanomodules/Buffer_Overflow-CS2_Java.html.

[39] Eben Upton. *Learning computer architecture with Raspberry Pi*. Hoboken, New Jersey: John Wiley & Sons, Inc, 2016. ISBN: 978-1-119-18393-8.

[40] Albert S. Woodhull. *An Open Letter to Intel*. URL: https://www.cs.vu.nl/~ast/intel/.

[41] Albert S. Woodhull. *Teaching with Minix Howto*. URL: https://minix1.woodhull.com/teaching/index.html#bookquote.

[42] Michael Ziwisky, Kyle Persohn, and Dennis Brylow. “A down-to-earth educational operating system for up-in-the-cloud many-core architectures”. In: *Trans. Comput. Educ.* 13.1 (Feb. 2013), 4:1–4:12. ISSN: 1946-6226. DOI: 10.1145/2414446.2414450. URL: http://doi.acm.org/10.1145/2414446.2414450.