Using Embedded Xinu to Teach Operating Systems on Baremetal RISC-V

Alexander Gebhard
Marquette University
Milwaukee, Wisconsin
alexander.gebhard@marquette.edu

Jack Forden
Marquette University
Milwaukee, Wisconsin
jack.forden@marquette.edu

Oliver Laufenberg
Marquette University
Milwaukee, Wisconsin
oliver.laufenberg@marquette.edu

Dennis Brylow
Marquette University
Milwaukee, Wisconsin
dennis.brylow@marquette.edu

ABSTRACT

RISC-V is an open computer architecture that has gained increasing popularity in recent years. Companies such as Google, Nvidia, and Huawei have all announced or developed CPUs based on the RISC-V architecture. The increasing popularity of RISC-V along with its simplicity make it an ideal platform for students to learn low-level operating system concepts. We have ported Embedded Xinu, a simple, lightweight, and education-focused operating system, to a baremetal RISC-V board. Embedded Xinu has been used to teach thousands of students operating systems over the past two decades. This new port is the first education-focused operating system designed to run on baremetal RISC-V. In the following sections, we describe the challenges in porting Embedded Xinu to support the RISC-V architecture. We describe how practitioners can adopt Embedded Xinu to teach low-level CS systems courses such as operating systems. Finally, we reflect on our experience using Embedded Xinu on RISC-V to teach operating systems in Spring 2023.

CCS CONCEPTS

• **Computer systems organization** → **Embedded software**; • **Applied computing** → **Interactive learning environments**.

KEYWORDS

RISC-V, Operating System, Embedded Xinu, Operating System Education

ACM Reference Format:
Alexander Gebhard, Jack Forden, Oliver Laufenberg, and Dennis Brylow. 2024. Using Embedded Xinu to Teach Operating Systems on Baremetal RISC-V. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2024), March 20–23, 2024, Portland, OR, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3626252.3630959

SIGCSE 2024, March 20–23, 2024, Portland, OR, USA

ACM ISBN 979-8-4007-0423-9/24/03.
https://doi.org/10.1145/3626252.3630959

1 INTRODUCTION

Embedded Xinu is a lightweight research and teaching-focused operating system designed for simplicity. This simplicity has enabled Xinu to be taught to thousands of students at numerous universities in both graduate and undergraduate courses. Embedded Xinu was based on the Xinu operating system developed by Douglas Comer [9] in the 1980s. In the mid 2000s, work on Embedded Xinu refocused away from mini computers and x86-based PCs toward RISC-based embedded consumer devices, such as PowerPC and RISC-powered appliances. Due to its elegant, architecture-agnostic design, Embedded Xinu has maintained the advantage of quickly porting to new platforms and architectures. This has allowed Embedded Xinu to continue to remain relevant in hands-on CS course work with modern platforms and upcoming architectures. To date, Embedded Xinu has run on a litany of hardware, but most notably it has been ported to the Linksys WRT54GL, Linksys WRT160NL, Raspberry Pi 1 B+, and a multicore implementation on Raspberry Pi 3 B+ [4, 9, 10]. The venerable Embedded Xinu OS can also support virtualized platforms such as the QEMU [3] ARM and RISC-V emulators, where physical hardware is impractical or prohibitively expensive.

The contributions of this paper include a technical description of porting a popular education-focused operating system ("OS") to the RISC-V architecture. We outline an example set of OS project assignments for instructors to adopt and follow. Finally, we detail our experience using this sequence of projects within an OS course of 47 students in Spring 2023.

2 BACKGROUND

There are many different computer processor architectures such as Intel x86, ARM, and MIPS. The x86 processor is classified as a Complex Instruction Set Computer (CISC). With CISC systems, a single assembly instruction can perform multiple tasks (such as load from memory, perform arithmetic, set a register, or load to memory). This further complicates CPU design and requires developers, educators, and students to understand multiple syntax patterns for one instruction.

ARM, much like MIPS and RISC-V, is designed using a Reduced Instruction Set Computer (RISC) philosophy. RISC architectures have a minimum, highly optimized set of assembly instructions. This trade-off means that developers usually have to write more instructions for a given program than a CISC architecture. Embedded

Xinu has been ported to run on the ARM/Raspberry Pi platform in 32-bit mode. However, since ARM has released its version 8 (ARMv8), it has increased in complexity. ARMv8 specifies 1070 different instructions in 53 different formats [26] that take 5778 pages to document [15]. When a previous team attempted to port Embedded Xinu to the 64-bit mode on the ARMv8 Raspberry Pi 3B+, they quickly despaired due to the increased complexity students would need to face for many basic operations. Embedded Xinu-based courses thrive when based on current, inexpensive platforms that embrace minimalist design principles but have mainstream support. RISC-V is a simple, open, and modular architecture that Embedded Xinu now supports.

RISC-V started as a research project under Dr. Krste Asanovi, Yunsup Lee, and Andrew Waterman at the University of California - Berkeley. Its simplicity and openness have encouraged its use in the academic community. The RISC-V Instruction Set Architecture (ISA) is designed to be modular, meaning that CPU manufacturers can choose to adopt parts of the ISA, without having to implement the entire ISA. These optional add-ons are called "extensions". Splitting the ISA into optional extensions simplifies the CPU design and does not require the ISA to be *intradependent*. This also makes it simpler for students and researchers to understand, as they can begin by learning the base ISA and explore the extensions only when needed. In contrast to ARMv8, the RISC-V base ISA has 47 instructions with only 6 different instruction formats. This simple yet open architecture sets RISC-V apart from alternatives such as MIPS, ARM, and x86 [12].

3 RELATED WORK

Historically operating systems courses generally fall into three categories: theoretical, virtual, and physical hardware (also sometimes called, "*baremetal*"). Theoretical courses teach the important aspects from a very high-level perspective. Theoretical courses can supplement the latter categories while still training students' innovative abilities [7]. Usually, theoretical courses stop short of going "under the hood" and allowing students to experiment with the writing of kernel code. On the other hand, virtual courses allow students to experiment using virtualized hardware or emulators. Over time, use of emulators in these courses has steadily increased in popularity, even more so as universities grappled with the COVID-19 pandemic. Virtual operating systems can be an attractive option due to their low cost of maintenance, ease of access, and safety, as many of these systems are run in a sandbox environment. While attractive, virtual machines simplify details students would encounter when developing on baremetal hardware. Additionally, virtual machines often differ significantly or completely abstract away their interactions with various hardware components [25]. Baremetal courses seek to teach operating systems on real hardware in a way that is accessible to all, without compromising some aspects required when opting for the virtual approach.

3.1 Virtual Educational Operating Systems

There is an extensive list of emulated learning operating systems; however, most are no longer supported or lack documentation should an educator want to adopt them [13]. Survey papers do not offer much relief, with previous work published in 2005 providing a poor view of the current landscape and available tools [1]. For example, if an educator wanted to use an emulated OS based on the MIPS architecture, they would only have three realistic options: PintOS (2009), Nachos (1993) and most recently Pandos (2021) [8, 13, 20]. Although these systems allow adopters to implement them in their courses, PintOS and Nachos have not had significant maintenance done in several years. Although Pandos provides a more modern option, it suffers from the inherent reality of running on the MIPS architecture. Although beloved for its simplicity in comparison to x86, MIPS is slowly moving towards being considered a legacy architecture, with the parent company ceasing production of all future MIPS cards in lieu of RISC-V [24].

Some attempts have been made to create an ISA-independent educational OS to mitigate some of these deprecation issues. While this is a more manageable task when using virtual hardware, due to the basic nature of virtual machines, it still requires continuous development from a dedicated community. The most widespread example of this is *xv6* [22], which has been ported to MIPS, x86, and most recently RISC-V, using QEMU virtual machines. The relatively large community at several universities, such as MIT and University of Wisconsin–Madison, distinguishes xv6 for both ongoing development and a wealth of adoption resources should an educator wish to implement it. xv6, however, is a "feature complete" operating system. Students do not write any core part of the operating system; rather, they build off the OS features that are already implemented. For example, when teaching file systems, xv6 does not have students implement the file system, rather students build a "file system checker" to ensure the file system is valid. Embedded Xinu takes a very different approach. Embedded Xinu guides students to implement actual parts of the OS from the ground up. Students will be building fundamental parts of modern operating systems, not extensions onto an already working OS.

3.2 Baremetal Educational Operating Systems

There are very few baremetal educational operating systems due to the complexity of dealing with real hardware. Similar issues of maintaining code bases for virtual systems are only amplified when dealing with real hardware systems. Stanford University developed PintOS [21] which can be run natively on x86 hardware; however, in practice, this is most often used in an emulator. Other similar x86 systems are GeekOS and Nanvix; both have the ability to run on both hardware and virtual environments, but differ in their methodology. GeekOS's projects build on each other, while Nanvix allows for a more free-form adaption, allowing instructors to hit individual learning goals [14, 19].

The landscape is barren when trying to adopt RISC-V based systems. While xv6 and Egos [27] have been adapted to run on Field Programmable Gate Arrays (FPGA), these are custom chips that have a high bar, preventing adoption by educators. To our knowledge, Embedded Xinu is the only current educational OS that runs on commercial RISC-V boards that can be purchased easily from retailers. This reinforces Embedded Xinu's continued legacy of being highly portable to new platforms, as it was also the first educational OS to run on commercially available multicore hardware like the Raspberry Pi 3B+ [17].

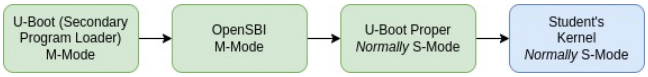

Figure 1: The Sipeed Nezha boot process

4 PORTING EMBEDDED XINU TO RISC-V

When evaluating candidate platforms for the RISC-V port of Embedded Xinu, we sought commercial boards that were both inexpensive and publicly documented. The Sipeed Nezha fits both criteria. The Sipeed Nezha uses an AllWinner D1 single-core SoC with 512MiB/1GiB/2GiB of RAM (depending on the board). These boards can be found for $125 online. The publicly accessible Technical Reference Manual (TRM) makes it less painful to implement hardware features into Embedded Xinu. Previous researchers adopting Embedded Xinu have struggled to find hardware documentation for boards such as the Raspberry Pi 3B+, leading to significant delays in implementing new features. This section describes the changes made to Embedded Xinu to support the RISC-V architecture.

4.1 Boot Sequence

The Sipeed Nezha consists of a multi-stage boot process as seen in Figure 1. The first stage U-Boot Secondary Program Loader (SPL) executes in the most privileged RISC-V mode, machine mode (or M-Mode). U-Boot SPL initializes the hardware (such as RAM) and finds the next stage, OpenSBI. OpenSBI provides a basic, cross platform API to handle hardware such as starting other *harts* (or hardware threads), managing the timer, and writing to the Universal Asynchronous Receiver and Transmitter (UART). Once OpenSBI is initialized, it jumps to U-Boot "proper" running in supervisor mode (S–Mode), the middle privileged mode in RISC-V. We have modified the OpenSBI code to keep U-Boot running in M-Mode so that we can access M-Mode level registers for student assignments. U-Boot "proper" is responsible for finding the OS, loading the OS into memory, and jumping to it. In our lab, we used the latest version of U-Boot flashed onto an SD card. The Sipeed Nezha's do come with U-Boot on the NAND chip, however, this version of U-Boot is older and does not support network booting via TFTP (which our lab uses to load the student's operating system onto the Nezhas). Once the student's kernel is loaded into memory, execution begins in the `start.S` assembly file. Although students are not expected to alter the boot sequence, they are responsible for understanding how the boot sequence works. Teaching students what happens from the moment power is turned on, to the execution of `start.S`, helps students gain a deeper understanding of the material and is an important aspect in understanding the reality of how hardware and software work together [18]. The Embedded Xinu mantra is, "there is no magic in the box."

4.2 Trap Handling

Trap handling in RISC-V Embedded Xinu is a crucial mechanism that enables the operating system to respond quickly and efficiently to hardware interrupts or exceptions. When a hardware device (e.g. a timer or an I/O device) generates an interrupt, the CPU halts the currently executing task and transfers control to an interrupt handler (dispatcher) that decodes the interrupt source and passes control to the appropriate function. Once the interrupt no longer needs attention, the processor will return to its previous state, and allow whatever prior process to continue executing. Interrupt handling is simpler on RISC-V compared to other architectures Embedded Xinu has supported. On RISC-V, the address of the interrupt/exception handler is loaded into the `stvec` or `mtvec` register depending on which privilege mode should handle the interrupt. Once an interrupt or exception occurs, the cause of the interrupt or exception is stored in the `scause` or `mcause` register. Furthermore, the AllWinner D1 stores the interrupt number at a fixed memory address. The interrupt numbers are publicly documented in the TRM for the AllWinner D1. Students write a part of the interrupt handler for two assignments during the semester. Embracing real-world examples within the Embedded Xinu environment, students see first-hand the practical implications of interrupt handling, which provides students a foundation to study operating system concepts [23].

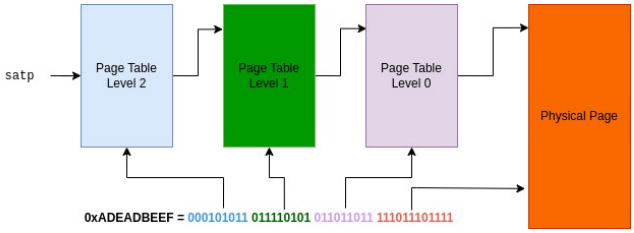

Figure 2: Splitting a virtual address into page table indexes

4.3 Memory Protection and Paging

Embedded Xinu on RISC-V is the first platform for which the OS implements memory protection, bringing it closer to many production OSes. The RISC-V specification defines multiple different paging schemes, however, the AllWinner D1 only supports Sv39 paging with a 3 level hierarchical page table. In Sv39 paging, virtual addresses are 39 bits long. Bits 30-38 are used to index the second level page table, bits 21-29 are used to index the first level page table, and bits 12-20 are used to index the zeroth level page table. The final 12 bits are used as an offset once the address at the third-level page table is found. Figure 2 gives an example of walking the page table using the virtual address `0xADEADBEEF`. The base address of the first-level page table is stored in the `satp` register.

This is the first port of Embedded Xinu that has natively supported paging and memory protection. Previous architectures such as MIPS and ARM have more complicated paging mechanisms, which discouraged student-facing implementations. With just a few structure definitions and initializations, students can be tasked with building out the page table code for initializing new processes or for allocating new physical pages for a given virtual address.

4.4 Kernel Organization

The transition to a RISC-V implementation allowed Embedded Xinu to completely overhaul the kernel organization. Unlike previous versions that followed a monolithic structure, the new Embedded Xinu was designed as a microkernel, embracing a minimalist approach.

This design would have been challenging in the ARM architecture; however, the memory protection in RISC-V makes this possible. The kernel's only responsibility is to manage system resources between user processes. As a result, the updated OS aligns with the latest best practices and industry trends, providing instructors with a valuable teaching tool.

```
syscall send(int pid, int msg)
{
    register pcb *procptr;
    irqmask im;

    im = disable(); // Disable interrupts
    // Check to ensure the proc ID is valid
    if (isbadpid(pid))
    {
        restore(im);
        return SYSERR;
    }
    procptr = &proctab[pid];
    if ((PRFREE == procptr->state)
        || procptr->hasmsg)
    {
        restore(im);
        return SYSERR;
    }
    // Set the message field on the process
    procptr->msg = msg;
    procptr->hasmsg = TRUE;

    /* if receiver proc waits, start it */
    if (PRRECV == procptr->state)
    {
        ready(pid, RESCHED_NO);
    }
    restore(im); // Reenable interrupts
    return OK;
}
```

Listing 1: send() function in Embedded Xinu

```
#define SYSCALL_SEND                28

#define SYSCALL(num) int status; \
    asm("li a7, %0" : : "i"(SYSCALL_##num));
    asm("ecall"); \
    asm("mv %0, a0" : "=r"(status)); \
    return status;

syscall user_send(pid_typ pid, message msg)
{
        SYSCALL(SEND);
}
```

Listing 2: User function for the send() system call

4.5 Inter-Process Communication

RISC-V Embeddded Xinu has two methods of interprocess communication, direct messages received by processes via a field in the process control block, and mailboxes represented by a separate structure that includes a message queue. Both of these variations were present in prior platforms for Embedded Xinu, and are implemented on RISC-V in very similar ways. For direct messaging, the send and receive functions, respectively, write or read a message field contained within each process control block, allowing only one message to be stored at a time (see Listing 1).

The send function in RISC-V Embedded Xinu has been converted to a system call. The user space version of send() calls a `SYSCALL` macro that places the system call ID (in this case, the constant `SYSCALL_SEND`) in the register `a7`. All arguments needed for the system call are kept in `a0` - `a6` argument registers. Once the `ecall` opcode is executed, the processor switches kernel privileged mode and jumps to the interrupt/exception handler in `stvec` - which in Embedded Xinu is the `syscall_dispatch` function. The `syscall_dispatch` function uses an internal table to determine which system call occurred (using the ID in the `a7` register), runs the corresponding function, and returns the result in `a0`.

For mailboxes, each must be allocated before it can be used and the desired size must be specified. Afterwards, the send and receive functions must specify a mailbox ID, and the messages enter a queue where receiving only gives the single oldest message in the mailbox. There is no limit on the number of processes that can communicate with a mailbox, as long as each process has access to the mailbox's ID, so this method is useful for a wider variety of scenarios than direct message passing, at the cost of memory usage.

For both mailbox and direct message passing, a method of ensuring mutual exclusion is necessary to prevent multiple senders from writing to a mailbox at the same time. Mutual exclusion is achieved by disabling interrupts entirely on our single-core implementations. However, the RISC-V implementation required one significant change to the support structures and interface: system calls. On earlier platforms, Embedded Xinu had no memory protection, and so each process was free to modify other process' fields or to disable interrupts for mutual exclusion without any issues. Now, RISC-V Embedded Xinu includes memory protection measures, so, much like with semaphores, all of the functions for messaging must be implemented as system calls, with a set of separate wrapper functions to allow user processes to request the operating system to act on their behalf.

5 TEACHING OPERATING SYSTEMS WITH EMBEDDED XINU

The RISC-V port of Embedded Xinu was used to teach operating systems in Spring 2023. To the best of our knowledge, this marks the first usage of off-the-shelf RISC-V baremetal platforms in an undergraduate OS course. The course used the free, online textbook *Operating Systems: Three Easy Pieces* by Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau [2]. Weekly assignments were designed to follow this textbook. We provide assignment descriptions, necessary starting files, Embedded Xinu source code, and sample testcases on our website [16]. Our lab follows the same setup as

Table 1: Assignments by Week

Topics	Assignments
Introduction and Overview	Project 1: UNIX and C
C Representation, Operators	Project 2: UNIX and C
Serial; Device Drivers	Project 3: Serial Driver
Processes; Context	Project 4: Context Switch
Trap Handlers	Project 5: Trap Handlers
Interrupts; Scheduling	Project 6: Process Scheduling
Address Translation; Paging	Project 7: Memory Protection
Memory Management	Project 8: Heap Management
Threads; Locks; Concurrency	Project 9: Threads

previous researchers [5]. Instructions with solutions are available to verified higher education instructors.

Our 3000-level OS course for undergraduates is required for majors in computer science, as well as computer engineering and biomedical/biocomputing. Course prerequisites include CS1 and CS2 or Data Structures, as well as a course in hardware systems or computer organization. Students are typically familiar with at least two programming languages at this point: Java and either Python or Matlab, depending on their major track. They have been exposed to an assembly language, but have no prior experience in C.

Before students start an assignment, any additional files needed for the assignment are distributed. These files are often ANSI C language header and skeleton files that offer detailed explanations of key structures and methods that students are expected to understand. Creating an OS, even one made for education, is a complex task. Typically, a day of lab was devoted to walking students through the important takeaways and helping them identify a good starting point. This focusing of student attention is aided by Embedded Xinu's design, which by nature, allows students to focus on the core concept of each week's assignment, rather than spending the majority of their time grappling with complex ideas such as system organization and design. An overview of weekly assignments is described in Table 1. A more detailed description of each is below.

In the first two weeks of the course, students are given introductory C assignments, as most students have never programmed in C. A typical Week 1 assignment could be a currency converter that takes in dollars, yen, rupies, pounds, and euros and converts to dollars. Week 1 is designed to get students familiar with functions, control statements, and loops in C. Week 2 builds off of Week 1's assignment by asking students to add dynamic data structures, such as outputting the converted currency list in reverse order. Week 2 assignments usually emphasize C structures and pointers, a common source of confusion for beginners.

Week 3 is the first week students are asked to begin developing on Embedded Xinu. Students are given some basic files which allow the OS to boot, however, they will not see any text appear on the screen. Week 3's assignment asks students to write the code for a synchronous UART (polling, non-interrupt-driven Universal Asynchronous Receiver/Transmitter) driver. Once the students have correctly read the TRM and implemented the UART driver, they will be able to take input and print output. Week 3's assignment teaches students how to write parts of a device driver, read technical documentation, and debug without use of a higher-level debugger.

In Week 4, students are tasked with implementing context switching. Context switching allows the concept of processes within the operating system. Students complete the code to save the context (or state) of the current running process before loading in the context of the next process. Many designs are possible, but attention to detail is crucial. Students require only four RISC-V assembly opcodes to complete this assignment, but must mirror their design in the corresponding C code for building the initial process context structure. This assignment is important as it teaches students about low-level calling convention, registers, and the interface between assembly instructions and higher-level programming.

Week 5's assignment asks students to implement trap handlers. A trap handler is a function that handles a call into the kernel when a user process wants to perform a privileged action. The trap handler is important later in the semester when students implement memory protection. Students are given an incomplete C function and are asked to determine which kernel function was requested. From there, they are asked to call the corresponding kernel function and return the result to the user process. This assignment is the first step in implementing "user" processes.

In Week 6, students are asked to implement a process scheduling algorithm. Until this week, the operating system uses a FIFO queue to select which process runs next. This assignment asks students to implement one of several common scheduling algorithms, such as simple priority, priority queue, or lottery-based scheduling. Additionally, students must complete the implementation of process preemption. Preemption will automatically switch processes if the current running process does not yield the processor after a given time period. This assignment demonstrates how operating systems perform preemptive multitasking of processes.

Week 7's assignment was first run in Spring 2023. The simplicity of the RISC-V specification made this assignment possible. Students are given an incomplete code to support paging with the memory management unit (MMU). Students are asked to write a function that, given a virtual address, walks the page table and returns the final page table entry. Once this function is implemented correctly, memory protection will work on their operating system. Memory protection is the final feature that is needed to implement user processes. From this assignment forward, all user processes cannot touch memory belonging to other user processes or the kernel.

In Week 8, students are asked to build the standard `malloc` and `free` system calls. The `malloc` call requires students to visualize how memory is laid out in the operating system. This assignment also requires the students to use their knowledge in Week 7 to understand how to navigate a linked list.

Week 9's assignment students are tasked with implementing POSIX threads [11] and concurrency into the operating system. Before this assignment is released, the instructor demonstrates a race condition in class with 4 threads updating the same global variable. The goal at the end of this assignment is for the students to be able to run the same program they saw in their operating system. Additionally, they must implement spinlocks to prevent the race condition. They are given most of the thread implementation but must create system calls for `pthread_create`, `pthread_join`,

Table 2: Final Exam Scores

Question Topic	Fall 2020 (Control)	Spring 2023 (Experiment)	Percent Difference (with +16% baseline)
Page Replacement	68.6%	58.5%	+5.8%
Deadlock	70.0%	53.0%	-1.0%
File Systems	42.2%	34.7%	+8.5%
Concurrency	49.1%	42.1%	+9.0%
Memory Management	41.6%	45.3%	+19.7%

`pthread_mutex_lock`, and `pthread_mutex_unlock`. Once implemented correctly, they should be able to run the same program as shown in class without any race conditions. This assignment teaches the importance of concurrency and race conditions.

6 RESULTS AND DISCUSSION

To assess the impact of the RISC-V port, student feedback and exam data were collected. Anecdotally, the students found it "cool" that they were completing important components of an OS designed to run on a real hardware platform (instead of a virtual machine). In addition, they recognized the novelty of working on a course with more hands-on learning than equivalent courses at most other universities. On the other hand, the students expressed that the cognitive complexity of the course was high compared to other computer science courses.

As instructors, the high cognitive complexity of the assignments is not surprising. Developing an operating system on baremetal is challenging. There is no debugger that can stop execution and allow students to view/manipulate variables or control flow. Students need to have a strong understanding of the concepts before trying to apply them in practice. Of particular challenge to the students was *Project 7: Paging & Memory Protection*. Students were especially confused with the structure of the 3 level hierarchical page table. We plan to address decreasing the cognitive complexity of the assignments by developing more instructional aids for students and teaching assistants. As the assignments were new, the TAs felt that they did not have enough time to learn the assignments. This lead to conflicting guidance among the teaching assistants. We plan to create more in-depth instructional aids for teaching assistants. Despite these challenges, exam data shows that students performed relatively better with the RISC-V Embedded Xinu assignments compared to Fall 2020.

To measure the effect of the new curriculum on students, we compared the operating systems class in Spring 2023 (n=47 students) to the operating systems class in Fall 2020 (n=14 students). The Fall 2020 course used an older version of Embedded Xinu which ran on the Raspberry Pi 3B+. Fall 2023 used similar assignments as Spring 2023 (with new assignments described in the last section). Both semesters used the same textbook. However, there are differences to take into account between the two semesters. Fall 2020 has fewer students due to a change in the class schedule. The Fall 2020 semester was also the first semester without COVID-19 lockdowns. Therefore, before any intervention occurred, we measured a baseline between the Fall 2020 semester and the Spring 2023 semester. The start of the course, before introducing Embedded Xinu, is very similar across both semesters. Students received slightly modified introduction assignments and first exam questions (to prevent cheating). Students in the Spring 2023 semester scored an average 16% lower on the first exam (53% in Fall 2020 vs 37% in Spring 2023). Thus, any change between the two semesters should account for the 16% difference before the intervention occurred.

To gauge student learning, we reused four final exam questions from the Fall 2020 semester in the Spring 2023 final exam. The average student scores are shown in Table 2. Of the five questions, students in the experiment semester scored *better* on four final exam questions compared to students in the control semester (taking into account the 16% difference before the intervention). As new assignments focused on memory organization were introduced this semester, it is unsurprising that student understanding in memory management and page replacement increased in the experiment semester. The decrease in exam scores for the Deadlock question is also unsurprising. In the Fall 2020, emphasis was placed on teaching multicore concepts. As the Sipeed Nezha is a single core machine, there was less emphasis on multicore concepts in Spring 2023. Overall, the students' understanding increased with the RISC-V port of Embedded Xinu compared to the previous Fall 2020 semester.

7 FUTURE WORK

The RISC-V architecture provides a sufficient foundation for multicore systems, and extending Embedded Xinu to efficiently utilize multiple cores could boost its performance. The difficulties posed by this involve both the alteration of code to accommodate multiple cores and the redesigning kernel subsystems to prevent deadlock.

We also plan on extending the RISC-V port of Embedded Xinu to support the TCP/IP stack. Previous researchers have implemented the TCP/IP stack in Embedded Xinu to teach networking courses [6], however, the hardware of those ports has fallen out of date. By moving the TCP/IP stack to the RISC-V port, it allows instructors to teach additional courses using modern hardware.

8 SUMMARY AND CONCLUSIONS

Embedded Xinu has existed for many decades, providing an attractive option for educators interested in adopting a baremetal OS course. Teaching on baremetal is often out of reach due to its challenging nature and high costs. However, Embedded Xinu mitigates both of these challenges by running on consumer RISC-V boards. Furthermore, the readily available supporting content for educators allows instructors without prior experience to adapt the course to their requirements and objectives. Embedded Xinu's relatively small code base and documentation enable instructors and students to feel confident in the weekly assignments. Moreover, the port to RISC-V abstracts even more of the confusing aspects associated with ARM or x86, increasing student understanding and allowing instructors to focus more of their efforts on the core concepts.

REFERENCES

[1] Charles L. Anderson and Minh Nguyen. 2005. A Survey of Contemporary Instructional Operating Systems for Use in Undergraduate Courses. *Journal of Computing Sciences in Colleges* 21, 1 (October 2005), 183–190.

[2] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. 2018. *Operating Systems: Three Easy Pieces.* CreateSpace Independent Publishing Platform, North Charleston, SC, USA.

[3] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Anaheim, CA) *(ATEC '05)*. USENIX Association, USA, 41.

[4] Dennis Brylow. 2007. An Experimental Laboratory Environment for Teaching Embedded Hardware Systems. In *Proceedings of the 2007 Workshop on Computer Architecture Education* (San Diego, California) *(WCAE '07)*. Association for Computing Machinery, New York, NY, USA, 44–51. https://doi.org/10.1145/1275633.1275643

[5] Dennis Brylow. 2007. An Experimental Laboratory Environment for Teaching Embedded Hardware Systems. In *Proceedings of the 2007 Workshop on Computer Architecture Education* (San Diego, California) *(WCAE '07)*. Association for Computing Machinery, New York, NY, USA, 44–51. https://doi.org/10.1145/1275633.1275643

[6] Dennis Brylow and Kyle Thurow. 2011. Hands-on Networking Labs with Embedded Routers. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education* (Dallas, TX, USA) *(SIGCSE '11)*. Association for Computing Machinery, New York, NY, USA, 399–404. https://doi.org/10.1145/1953163.1953283

[7] Xiangqun Chen, Weizhen Sun, Yu Luo, Lei Wang, and Yong Xiang. 2020. Research on the knowledge hierarchy and practice teaching of operating system course under the background of emerging engineering. In *2020 15th International Conference on Computer Science & Education (ICCSE)*. 30–34. https://doi.org/10.1109/ICCSE49874.2020.9201856

[8] Wayne A. Christopher, Steven J. Procter, and Thomas E. Anderson. 1993. The Nachos Instructional Operating System. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings* (San Diego, California) *(USENIX'93)*. USENIX Association, USA, 4.

[9] Douglas Comer. 1984. *Operating System Design: The XINU Approach.* Prentice-Hall, Inc., USA.

[10] D.E. Comer and T.V. Fossum. 1988. *Operating System Design: The XINU Approach.* Prentice Hall.

[11] Linux Foundation. 2023. *pthreads(7) — Linux manual page.* Retrieved August 18, 2023 from https://man7.org/linux/man-pages/man7/pthreads.7.html

[12] RISC-V Foundation. 2019. *The RISC-V Instruction Set Manual.*

[13] Mikey Goldweber, Renzo Davoli, and Mattia Biondi. 2021. The Pandos Project and the uMPS3 Emulator. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1* (Virtual Event, Germany) *(ITiCSE '21)*. Association for Computing Machinery, New York, NY, USA, 122–128. https://doi.org/10.1145/3430665.3456331

[14] David Hovemeyer, Jeffrey K. Hollingsworth, and Bobby Bhattacharjee. 2004. Running on the Bare Metal with GeekOS. *SIGCSE Bull.* 36, 1 (mar 2004), 315–319. https://doi.org/10.1145/1028174.971411

[15] ARM Ltd. 2015. *ARMv8-A Reference Manual.* Retrieved August 18, 2023 from https://developer.arm.com/documentation/ddi0487/ah/

[16] Marquette University Systems Lab. 2023. *The Embedded Xinu Wiki.* https://xinu.cs.mu.edu

[17] Patrick J. McGee, Rade Latinovich, and Dennis Brylow. 2020. Using Embedded Xinu and the Raspberry Pi 3 to Teach Operating Systems. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 307–315. https://doi.org/10.1109/IPDPSW50202.2020.00063

[18] James W. McGuffee. 2020. Why Teach Operating Systems? *J. Comput. Sci. Coll.* 35, 9 (apr 2020), 44–51.

[19] Pedro Henrique de Mello Morado Penna, Márcio Bastos Castro, Henrique Cota de Freitas, Jean-François Méhaut, and João Caram. 2017. Using the Nanvix Operating System in Undergraduate Operating System Courses. In *2017 VII Brazilian Symposium on Computing Systems Engineering (SBESC)*. 193–198. https://doi.org/10.1109/SBESC.2017.33

[20] Ben Pfaff, Anthony Romano, and Godmar Back. 2009. The Pintos Instructional Operating System Kernel. *SIGCSE Bull.* 41, 1 (mar 2009), 453–457. https://doi.org/10.1145/1539024.1509023

[21] Ben Pfaff, Anthony Romano, and Godmar Back. 2009. The Pintos Instructional Operating System Kernel. In *Proceedings of the 40th ACM Technical Symposium on Computer Science Education* (Chattanooga, TN, USA) *(SIGCSE '09)*. Association for Computing Machinery, New York, NY, USA, 453–457. https://doi.org/10.1145/1508865.1509023

[22] Russ Cox, Frans Kaashoek, and Robert Morris. 2023. *Xv6, a simple Unix-like teaching operating system.* https://pdos.csail.mit.edu/6.828/2023/xv6.html

[23] Soe Than. 2007. Use of a Simulator and an Assembler in Teaching Input-Output Processing and Interrupt Handling. *J. Comput. Sci. Coll.* 22, 4 (apr 2007), 75–81.

[24] Jim Turley. 2021. *Wait, What? MIPS Becomes RISC-V.* https://www.eejournal.com/article/wait-what-mips-becomes-risc-v/

[25] Maxwell Walter and Sven Karlsson. 2017. Software Tools for Low-Level Software and Operating Systems Classes. In *Proceedings of the 19th Workshop on Computer Architecture Education* (Toronto, ON, Canada) *(WCAE'17)*. Association for Computing Machinery, New York, NY, USA, 16–23. https://doi.org/10.1145/3116214.3116241

[26] Andrew Waterman. 2014. *Instruction Sets Should Be Free: The Case For RISC-V.* Technical Report. Berkeley, CA, USA.

[27] Yunhao Zhang. 2023. *EGoS-2000 Repository.* https://github.com/yhzhang0128/egos-2000