1-1-2017

XinuPi3: Teaching Multicore Concepts Using Embedded Xinu

Priva Bansal
Marquette University

Rade Latinovich
Marquette University

Tom Lazar
Marquette University

Patrick J. McGee
Marquette University

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

CSERC '17 Proceedings of the 6th Computer Science Education Research Conference, Pages 20-25. DOI.

XinuPi3: Teaching Multicore Concepts Using Embedded Xinu

Priya Bansal, Rade Latinovich, Tom Lazar, Patrick J. McGee, and Dennis Brylow
Department of Mathematics, Statistics, and Computer Science
Marquette University
Milwaukee, Wisconsin, USA
{priya.bansal,rade.latinovich,thomas.lazar,patrick.j.mcgee,dennis.brylow}@marquette.edu

ABSTRACT

As computer platforms become more advanced, the need to teach advanced computing concepts grows accordingly. This paper addresses one such need by presenting *XinuPi3*, a port of the lightweight instructional operating system *Embedded Xinu* to the Raspberry Pi 3. The Raspberry Pi 3 improves upon previous generations of inexpensive, credit card-sized computers by including a quad-core, ARM-based processor, opening the door for educators to demonstrate essential aspects of modern computing like inter-core communication and genuine concurrency.

Embedded Xinu has proven to be an effective teaching tool for demonstrating low-level concepts on single-core platforms, and it is currently used to teach a range of systems courses at multiple universities. As of this writing, no other bare metal educational operating system supports multicore computing. XinuPi3 provides a suitable learning environment for beginners on genuinely concurrent hardware. This paper provides an overview of the key features of the XinuPi3 system, as well as the novel embedded system education experiences it makes possible.

KEYWORDS
Raspberry Pi, Embedded Xinu, Embedded systems education, Multicore, Computing

ACM Reference format:
Priya Bansal, Rade Latinovich, Tom Lazar, Patrick J. McGee, and Dennis Brylow. 2017. XinuPi3: Teaching Multicore Concepts Using Embedded Xinu. In *Proceedings of The 6th Computer Science Education Research Conference, Helsinki, Finland, November 14, 2017 (CSERC '17),* 6 pages.
https://doi.org/10.1145/3162087.3162091

1 INTRODUCTION

Embedded Xinu is a lightweight operating system that is designed to assist in teaching embedded operating systems to undergraduate students.[25] As a RISC adaptation of its original CISC ancestor, Xinu,[11] Embedded Xinu has been ported to many embedded platforms, including multiple consumer-grade home networking appliances such as Linksys routers,[25] the experimental Intel Single-Chip Cloud (SCC)[29] processor, and the ubiquitous Raspberry Pi 1.[6] In order to remain viable, Embedded Xinu must be continually updated to run on commercially-available platforms and to support the current state of typical computer hardware.

The Raspberry Pi 3 is the most recent addition to the popular Raspberry Pi series.[18] Its low cost of US$35[21] makes this platform uniquely suited to introduce systems concepts to students.

While the Raspberry Pi and its upgraded models are typically used to run full-featured Linux-based operating systems such as Raspbian,[16] it is increasingly difficult to teach operating systems (O/S) concepts to beginners using exemplars surpassing more than 15 million lines of code.[16] Embedded Xinu contains approximately ten thousand lines of code[15] in its entirety, which is equivalent to the size of one file in the Raspbian kernel's process scheduling subsystem.[12] This multi-tasking, interrupt-driven embedded O/S with device drivers, resource management, and interprocess communication primitives can thus be understood and modified by students in a single collegiate semester, orders of magnitude more quickly than general purpose O/S code bases.

Our port, *XinuPi3*, of Embedded Xinu to the Pi 3, is a lightweight, open source, symmetric multiprocessing instructional operating system. The Xinu kernel has been used in many commercial embedded products over the years, and thus is no mere toy. The simplicity and clarity of its design allows educators to work with multicore concepts such as genuine concurrency on real embedded hardware in their courses, while still remaining comprehensible to undergraduate students.

2 RELATED WORK

Many O/S and systems architecture courses are taught using some form of educational operating system.[13] The exact nature of these educational systems can be split into two main categories: simulated and those designed to run on physical hardware. Some of the systems in these categories can be run on either simulated or physical hardware but this grouping focuses on the platform on which they were primarily designed to run.

Simulated operating systems, such as Nachos,[10] Minix,[27] and JaeOS,[17] run on virtual machines. Simulated operating systems are beneficial in that they do not require the use of any specialized equipment. Because simulated operating systems operate on abstractions from physical hardware, they do not suffer from the same support, cost, and space constraints as physical operating systems. As a consequence of their hardware abstraction, simulated O/Ses typically are of reduced utility in genuine embedded systems, and sharply limited in the types of interactions they may exhibit in student assignments.

Physical hardware systems, sometimes called "bare metal" systems, expose students to actual hardware. Embedded Xinu is one example of such a system supporting the Raspberry Pi 1,[6] the Intel SCC,[29] and several MIPS-based consumer networking appliances such as Linksys wireless routers.[8] These Embedded Xinu ports have been used to teach a variety of systems courses at many universities, including operating systems,[8] embedded systems,[25] compilers,[14] and hardware systems,[7] among others.

Other examples of bare metal educational operating systems include Cornell University's PortOS,[5] which is designed to run through emulation on a variety of handheld devices as well as desktop platforms. Tanenbaum's Minix,[27] a fully built instructional operating system, supports several different platforms. Stanford's PintOS[19] project adapted the venerable Nachos operating system to run on the x86 platform. To date, none of the other widely-used bare metal instructional operating systems have been extended to support a commercially-available multicore platform like XinuPi3.

3 PORTING EMBEDDED XINU TO THE PI 3

The following subsections detail the new hardware of the Pi 3 as it pertains to the progression of Embedded Xinu.

3.1 Hardware Overview and Changes

The Raspberry Pi 3 uses the Broadcom BCM2837 SoC (System on a Chip), containing a quad-core, 1.2GHz ARM Cortex-A53 CPU implementing both 64-bit and 32-bit versions of the ARMv8-A instruction set.[3] The 32-bit version is largely compatible with the ARMv7 instruction set, which is used by XinuPi,[6] the previous port of Embedded Xinu onto the Raspberry Pi 1. XinuPi3 currently runs in 32-bit mode because of this overlap in compatibility as well as the desirability of a simplified instruction set in a teaching setting.

The SoC also contains Broadcom's VideoCore IV GPU and various peripherals such as microSD storage, a 40-pin General Purpose Input and Output (GPIO) header, two Universal Asynchronous Receiver/Transmitter devices, one on-board Bluetooth 4.1, and an 802.11n WiFi chip. It includes a microUSB port for power, an HDMI 1.4, a 3.5 mm audio jack, four USB 2.0 ports, and Ethernet capability.[20]

XinuPi does not require device driver support for all of these peripherals to provide a satisfactory educational user experience. In particular, the Bluetooth, WiFi and audio jack subsystems have been left as future work, and the device driver for accessing microSD storage is not generally included in student builds of the kernel. (Student kernels are loaded into RAM over Ethernet by the XinuPi bootloader; if students inadvertently overwrite the bootloader on microSD, the bootloader image will have to be manually restored before the platform can again be successfully booted.)

3.2 Boot Process

The Pi 3's initial booting process is nearly identical to the boot processes of previous versions of the Pi.[22] When the Pi turns on, all four cores are off and the VideoCore GPU is on. The GPU starts by executing the first stage bootloader that is located on the aforementioned SoC. The first stage bootloader's primary function is to load the second stage bootloader which is located on the removable SD Card. The second stage bootloader initializes much of the hardware, including the RAM and L1 and L2 caches. It also prepares the CPU to execute the kernel. Once the kernel is loaded into memory, the processor starts execution at memory address `0x8000`, where student kernel boot code is loaded.

Cores initially start in Hypervisor mode, but Embedded Xinu requires the cores to be in System mode in order to be able to execute properly.

```
mrs     r0, cpsr           /* Get CPSR value.        */
orr     r0, r0, #0x1F      /* Mask on System Mode.   */
msr     spsr_cxsf, r0      /* Set up return context  */
add     r0, pc, #4         /*  to go to PC+4 in      */
msr     ELR_hyp, r0        /*  System Mode upon      */
eret                       /*  exception return.     */
```

Figure 1: Changing privilege level to system mode

Typically, when it is desired to change modes on the processor, bit manipulation is performed on the Current Program Status Register (CPSR),[4] which can be simplified by using the cps (Change Processor State) opcode. However, in supervisor mode, the only way to change to a lower privilege level, such as system mode, is to use an exception return.[4] An exception return loads the contents of the Saved Program Status Register (SPSR) into the CPSR, and continues execution at the address specified in the Exception Link Register (ELR). Figure 1 shows how it is done in assembly. In the case of Figure 1, the ELR is set to the instruction that comes immediately after `eret`.

XinuPi3's boot code also is responsible for the exception vector table. This is discussed more in-depth in Section 3.5. The BSS memory segment is also set up during the boot process, as BSS initialization is necessary to run any C code. The boot code also initializes each core's stack pointer and configures the Memory Management Unit (MMU) and L1 and L2 caches, as explained in Section 4.3.

From an instructional standpoint, students can be given fewer than 100 lines of initial ARM assembly and C code as a starting point for a stripped down C execution environment on the bare metal ARM core.

3.3 UART Driver

Output is essential when working with embedded software. The Raspberry Pi 3 contains two UARTs, one Mini-UART and one PL011 (PrimeCell) UART,[1] making it possible to interact with the device through a serial connection. The functionality of the UARTs differ in that the PL011 includes break detection, framing errors detection, and timeout interrupts.[23] The PL011 UART is a straightforward starting point for students constructing their first simple device driver, as the overall functioning of the device is comparable to many predecessor UARTs that have been standard on personal computing platforms since the 1980s.

The UART control and status registers, along with the other peripherals, are all memory-mapped, meaning that we access the peripheral registers as if they were addresses in memory. On the Pi 1, the base address for the peripherals was `0x20200000`. As for the Pi 2 and Pi 3, this address has been changed to `0x3F000000`,[24] due to the increased size of RAM.

3.4 Threads and Context Switching

A thread (or task) is a basic unit of CPU utilization, each of which contains an ID, program counter, working register set, and stack of activation records.[26] With multicore XinuPi3, each core now begins with its own null (or "idle") thread. This null thread executes in an infinite loop until the scheduler gives the core a new thread to execute (see Section 4.4).

Scheduling threads across multiple cores is a rich area for educational activities. XinuPi3 includes a simple, priority-based global scheduler as the default, but many extensions and enhancements can delve into the complexities of load balancing, thread migration, and real-time scheduling algorithms.

3.5 Interrupts

An interrupt is a signal to the processor indicating that an event has occurred and that the processor should handle that event accordingly. When the processor receives an interrupt, it stops the task it is currently executing, and goes to a predetermined location in memory to access an exception vector table (EVT). The EVT associates a received exception or interrupt with the proper instruction address to begin handling the exception. The implementation of the vector table is architecture-specific, although the concept is similar across many platforms.

It should be noted that for the Pi 3's architecture, the exception vector table must be 32-byte aligned in memory.[4]

```
ldr r1, =_vectors            /* Load table address */
mcr p15, 0, r1, c12, c0, 0   /*  into VBAR.        */
```

Figure 2: Writing the address of the Exception Vector Table

In order for the processor to find the vector table, following the protocol of the ARMv8-A architecture,[2] the address of the vector table is loaded into the Vector Base Address Register (VBAR) system register (as shown in Figure 2).

4 THE MULTICORE CPU: PREPARATION AND UTILIZATION

The primary contribution of our new XinuPi3 port is the addition of support for the multiple processor cores on the Pi 3 platform. This requires attention to preparing and starting the cores, memory management and cache coherency, cross-core mutual exclusion, and multicore scheduling.

4.1 Core Mailboxes

Upon startup, the XinuPi3 kernel *parks* each of cores 1-3 into its own infinite loop. Only core 0 executes normally during the earliest stages of preparation. Each other core waits looping until a nonzero value is put into one of the following memory addresses: `0x4000009C`, `0x400000AC`, or `0x400000BC`.[28] Each of these memory locations is termed a core *mailbox*. For symmetry, `0x4000008C` is set aside to be used as a mailbox for core 0, but the kernel does not use this mailbox to start core 0, which is already running.

When a nonzero value is put into a parked core's mailbox, the corresponding core will start execution at that address. In order for

```
ldr r0, =core_init_sp   /* Load initial stack */
ldr sp, [r0, #4]        /*  value into SP reg.*/
```

Figure 3: Storing core 1's initial stack pointer.

the core to execute C code properly, among the first instructions it is given must be an instruction to set up its stack pointer (as shown in Figure 3). XinuPi3's startup code for Core 0 allocates main memory space for each core's null process stack, and stores the initial value in the corresponding index of the global array, `core_init_sp`. The array is set up in file `start.S`, the entry point of the XinuPi3 kernel source code.

4.2 Memory Management Unit and Caches

The Memory Management Unit (MMU) is a piece of computer hardware that provides a layer of abstraction between the processor and memory, hiding the details of underlying physical memory hardware. The MMU is primarily used to perform translation of virtual memory addresses to physical memory addresses as well as handle memory protection and cache control.

A cache, in the context of the CPU, is an intermediary piece of memory between the processor and main memory. Typically, CPUs use multiple levels of cache. The Raspberry Pi 3 has two levels of cache: L1 and L2. Each core has its own L1 cache that is probably 32- kilobytes (there is no publicly available documentation specifying this), and all share a larger 512-kilobyte L2 cache. The caches are separated into two parts, one for instructions and another for data.

Some of the features provided by the MMU are not necessary for student-focused Embedded Xinu, such as configuring a virtual address space. Therefore, we use a minimal configuration of the MMU to map virtual addresses one-to-one with physical addresses, so that, for example, virtual address `0x8000` is equivalent to physical address `0x8000`. Embedded Xinu configures the peripheral address space as non-cacheable memory, as caching peripheral address space may cause coherency issues when attempting to access peripherals. The primary focus of this configuration is to enable cache on certain memory spaces rather than providing a virtual memory address space.

4.3 Mutual Exclusion

The ARMv8-Aarchitecture includes two instructions, load-exclusive (`ldrex`) and store-exclusive (`strex`),[3] that provide essential synchronization primitives. Building atop these primitives, Embedded Xinu implements more familiar synchronization mechanisms like mutex locks and semaphores.

A *mutex lock* is one of the simplest tools to solve the *critical-section problem*.[26] Mutex locks are used to address the problem of resource sharing and are used to ensure *mutual exclusion* during a critical section. Mutex locks are essential as they provide a means to prevent race conditions within embedded systems.

While implementing a simple spin-lock mutex, a few caveats were discovered regarding the use of the `ldrex` and `strex` opcodes on the Raspberry Pi 3. Specifically, the memory that is being accessed by these instructions has to be loaded into cache and marked as shareable by the MMU.[3] The reason for the minimal configuration of the MMU and cache, which was described earlier in

```
mutex_acquire:
    pld     [r0]              /* Preload data into cache to increase chance of success. */
    nop
    ldrex   r1, [r0]          /* Load the value of the lock from memory.                */
    cmp     r1, #LOCKED       /* IF already locked,                                     */
    beq     mutex_acquire     /* THEN try again...                                      */
    mov     r1, #LOCKED       /* ELSE unlocked, try to lock.                            */
    strex   r2, r1, [r0]
    cmp     r2, #0x0          /* IF lock acquire failed this time around,               */
    bne     mutex_acquire     /* THEN try again back up at the top.                     */
    dmb                       /* Data Memory Barrier opcode waits for memory accesses   */
    bx      lr                /*   to complete before returning.
```

Figure 4: XinuPi3's spin-lock mutex acquire in ARM assembly.

Section 4.2, is largely because of these requirements for exclusive memory access.

Another caveat that may not be obvious at first can be seen in Figure 4. To load the data into cache the `pld` (Preload Data) instruction is used. This ARM opcode starts a request to load the data into cache, if it is not already there. However, the `pld` instruction does not wait for the data to be loaded into cache, it only sends the request to do so.[2] Therefore, a slight delay is achieved using nops for the purpose of increasing the probability that the data has been loaded in cache before an exclusive memory access is invoked.

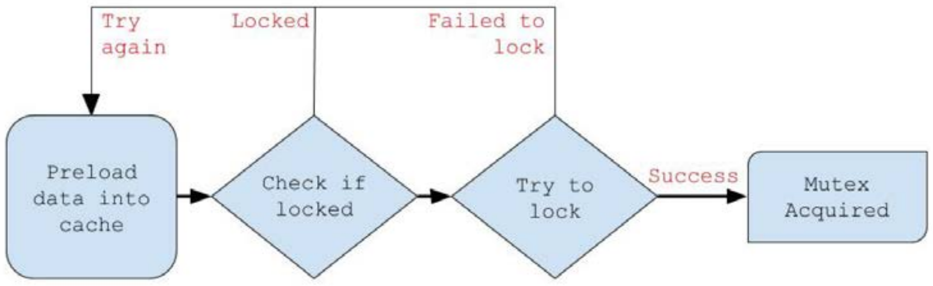

Figure 5: Spin-lock flow Diagram

The spin-lock mutex implementation was intentionally made to be verbose for the purpose of enabling students to understand and conceptualize the workings of a spin-lock mutex. In keeping with most multicore systems, this spin-lock implementation is optimistic. Mutual exclusion is guaranteed, but in the worst case, a core might be perpetually denied the lock by competing cores. In practice, such a degenerate case is highly unlikely.

4.4 XinuPi3's Multicore Scheduler

The Scheduler is the component of the kernel that controls what threads are running, how long they run, and when they stop running. Single core versions of Embedded Xinu use *multitasking*, a scheduler design that emulates concurrent processing by rapidly switching the thread that is running on the CPU. XinuPi3's multicore scheduler allows genuine concurrency, scheduling threads to execute in parallel on distinct cores with shared memory.

Embedded Xinu stores information about each thread in a continuous table structure called the `thrtab`. The structure of this table allows for simple O(N) insertions and O(1) removals, while still being simple enough for students to understand the structure. This table associates a key (`tid`) with a thread entry (`thrent`). Each `thrent` structure stores important information about the current state of the thread, which core the thread is running on, its current execution state, a name, and a stack pointer. This information is vital to maintaining a concurrent multiprocessing operating system.

In order for a thread to begin execution, it must go through several steps in Embedded Xinu. First, it must be created programmatically. Once a thread has been created it is put into the *ready queue* and given the state THRREADY. When a core calls `resched()` to ask the scheduler for a new thread to run, the scheduler dequeues from the ready queue and the core *context switches* to that thread using the `ctxsw()` function. If a core calls `resched()` and there are no `thrent's` in the ready queue then the processor continues to loop in its null thread. The null thread is always ready to run, ensuring that if a CPU has no current jobs it continuously requests a new one from the scheduler.

When a thread is running it can perform the following tasks:

- Voluntarily call `resched()` and allow another thread to run. The calling thread is put back into the ready queue,
- Call `sleep()` and put itself on the sleep queue. After it has slept for the specified length of time, it will be put back into the ready queue,
- Call on an I/O or physical device driver. While waiting for the corresponding interrupt to be called, the thread is stored in the I/O Queue, and

- Wait for a semaphore to be free in order to access a protected resource. Once the semaphore is `signal()`ed, and the waiting thread makes it to the front of that semaphore's wait queue, the thread resumes.

Within XinuPi3, each core has access to the central ready queue which, upon idle, dequeues the next element. This allows for a simple load balancer that affirms each core is equally busy, while giving headroom for students to expand its implementation in future assignments. For example, a possible extension could instruct students to add a thread migration system which would ensure that the number of threads assigned to each core remains balanced, even after certain threads are done executing.

Information about each individual core is placed in a new table called the `coretab` which stores the state of the core and its physical address. In utilizing a single `coretab`, this structure may support platforms that have a varying amount of cores.

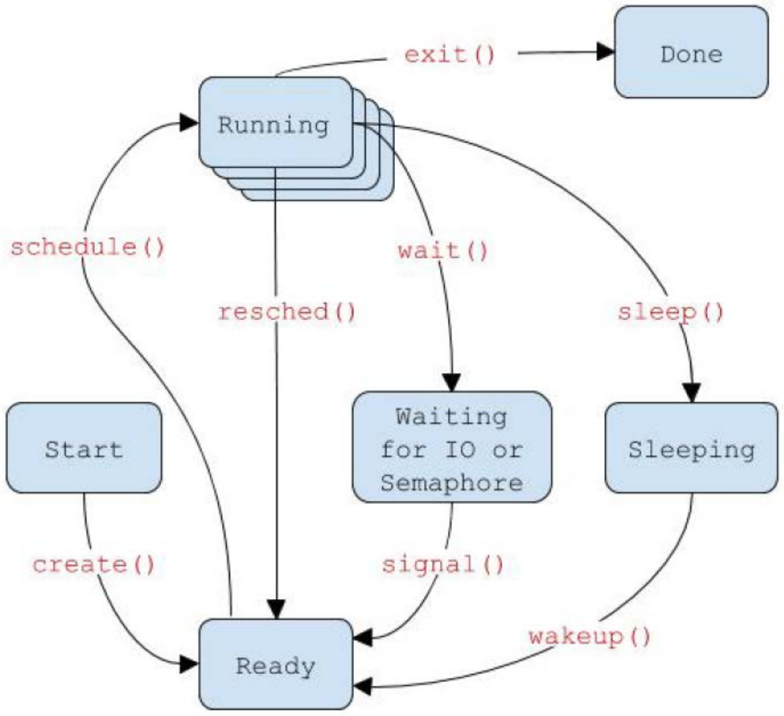

Figure 6: Xinu Process State diagram.

5 TEACHING WITH MULTICORE EMBEDDED XINU

The most obvious course in the standard curriculum to benefit from the first multicore port of Embedded Xinu is Operating Systems. University O/S courses range from "hands-free" surveys of O/S concepts, to hardcore explorations of queuing theory, to introductory systems programming, to deeply applied O/S-building project courses. For any course that includes an applied system programming component, Embedded Xinu on Raspberry Pi 3 offers an inexpensive but powerful platform for students to directly interact with genuinely concurrent embedded software.

As an example, Marquette University's three-credit-hour Operating Systems course is required for our Computer Science majors, Computer Engineering majors, and also counts as a specific technical elective for many Biocomputing majors. Computer Science majors normally take this course during their fourth semester, immediately following the course's prerequisite class, Hardware Systems. Engineering majors typically take the O/S course during their third year. The hands-on, low-level system programming experiences in this course are highly valued by those majors who go on to take more advanced coursework in embedded and real-time systems, as well as courses in internetworking and cybersecurity.

The single core version of the O/S course challenged student teams to design, implement and test significant components of Embedded Xinu O/S, such as the synchronous serial driver, context switch, process scheduler, and memory allocator. Synchronization primitives such as mutexes and semaphores could be implemented inmany variants, and applied to classic interprocess communication problems, such as "Dining Philosophers" or concurrent readers and writers.

Multicore XinuPi3 enables a sequence of new activities that highlight the genuine concurrency capabilities available on modern platforms. An early assignment now asks the students to unpark

```c
/* TODO: Initialize two cores.              */
/*       Have each core output via the      */
/*        shared serial port.               */
void nulluser()
{
         . . .
     /* TODO: use unparkcore(int, void *) */
     /*       method to start core 1      */
     /*       at specified code location. */
}

void testcore0(void)
{
     /* TODO: Add print statements        */
     /*       and other testcases         */
     kprintf("This␣is␣core␣%d.", cpuid);
}

void testcore1(void)
{
     /* TODO: Add print statements        */
     /*       and other testcases         */
     kprintf("This␣is␣core␣%d.", cpuid);
}
```

Figure 7: First multicore demonstration.

each of the cores, initialize them with valid C execution environments, and demonstrate serial port output from each core. See Figure 7.

The scheduling assignment now tasks students with one or more variants on the simple global scheduler with näive load balancing.

For the thread synchronization project, students use the platform-provided `ldrex` and `strex` ARM opcodes to guarantee mutually exclusive updates to a shared, global value, free of race condition contention from other cores. Using this basis, students can construct counting or binary semaphores with queues, bounded-wait mutexes, simple concurrent monitors, or condition variables.

When learning about interprocess communication, (a standard topic in any O/S course,) the multicore system can run genuinely concurrent examples, greatly increasing the opportunities to observe race conditions in action, and to think deeply about proper avoidance techniques. Furthermore, simple parallelizable computations can be deployed, demonstrating measurable speed-up over single-core implementations.

Single-core Embedded Xinu provides the standard C library function `malloc()`, which allocates memory from the Pi's available heap. Completing or improving upon `malloc()` and `free()` is a commonplace student assignment in this context. The Pi 3 platform uses shared memory between all the cores, so the simplest baseline memory allocator for XinuPi3 uses the same malloc() code with an additional mutex to prevent multiple cores from accessing the critical section of the memory allocation code concurrently. This heavy-handed solution works correctly but presents grave efficiency concerns that grow quickly as the number of concurrent cores scales. Multicore embellishments for the memory allocator include cordoning off dedicated "mini heaps" for each core, which would allow faster local heap allocation when needed. Of course, this only raises new resource-balancing questions, as well as a host of intellectually challenging quandaries involving allocation of shared space, migration of heap-allocated blocks, and delegation of responsibility for deallocation.

6 SUMMARY AND CONCLUSIONS

XinuPi3 is a novel port of the venerable Embedded Xinu O/S to the multicore Raspberry Pi 3. The need to provide students with hands-on experience programming modern embedded platforms frequently outstrips the availability of well-tested instructional tools and curricula. XinuPi3 occupies a unique niche as an established bare metal instructional O/S that is actively maintained, used in a wide variety of courses at many schools, and that now supports genuine concurrency on a modern multicore platform.

Maintaining practical embedded system educational tools remains difficult in the face of rapid hardware evolution, and a persistent lack of detailed, low-level documentation and datasheets for widely available commercial platforms. In this paper, we have presented our contribution to this toolset, as well as many of the technical insights that may allow others to port bare metal tools to the multicore Pi 3 board.

6.1 FutureWork

In order for Embedded Xinu to continue its service in computer systems courses at universities, there must be extensive documentation available explaining how to use it as a teaching tool, whether it runs on a Linksys router or a board within the Raspberry Pi series. Currently, such instructional information exists for the past ports of Embedded Xinu on the Embedded Xinu Wiki.[9] We are currently developing instructor-friendly documentation detailing the importance of teaching using XinuPi3 and how to set up the laboratory environment accordingly.

As an instructional tool, it is essential to measure the impact of XinuPi3 on actual student learning in an educational context. XinuPi3 will be deployed as a teaching tool in Spring of 2018 at Marquette University in the courses COSC 3250: Operating Systems, and COEN 4820: Operating Systems and Networks. Previous instances of these courses have collected ample baseline data on student learning outcomes with single-core Embedded Xinu on several platforms, including Raspberry Pi 1 boards. The data collection in 2018 will focus on changes to students' understanding of concurrency and syncronization in the context of hands-on laboratory projects using multicore XinuPi3.

ACKNOWLEDGEMENTS

We are indebted to the many student researchers who have previously contributed to iterations of the Embedded Xinu code base, and the many more students who have provided feedback on our curriculum and tools. This work is supported in part by an NSF REU supplement to CNS-1339392, and ACI-1461264.

REFERENCES

[1] ARM. 2005. PrimeCellR UART (PL011). (2005).
http://infocenter.arm.com/help/topic/com.arm.doc.ddi0183f/DDI0183.pdf

[2] ARM. 2016. ARMR *CortexR -A53 MPCore Processor Technical Reference Manual*. ARM Holdings, Cambridge, England. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0500e/index.html

[3] ARM. 2017. ARM R Architecture Reference Manual ARMv8, for ARMv8-A architecture profile. (2017).

[4] ARM. 2017. Bare-metal Boot Code for ARMv8-A Processors. (2017).

[5] Benjamin Atkin and Emin Gün Sirer. 2002. PortOS: An Educational Operating System for the Post-PC Environment. *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education* (2002), 116–120. https://doi.org/10.1145/563340.563384

[6] Eric Biggers, Farzeen Harunani, Tyler Much, and Dennis Brylow. 2013. XinuPi: Porting a Lightweight Educational Operating System to the Raspberry Pi. In *2013 Proceedings of the Workshop on Embedded and Cyber-Physical Systems Education*. http://www.mscs.mu.edu/~brylow/papers/BiggersHarunaniMuchBrylow-WESE2013.pdf

[7] Dennis Brylow. 2007. An experimental laboratory environment for teaching embedded hardware systems. In *Proceedings of the 2007 workshop on Computer architecture education (WCAE '07).* ACM, New York, NY, USA, 44–51. https://doi.org/10.1145/1275633.1275643

[8] Dennis Brylow. 2008. An experimental laboratory environment for teaching embedded operating systems. In *Proceedings of the 39th SIGCSE technical symposium on Computer science education (SIGCSE '08).* ACM, New York, NY, USA, 192–196. https://doi.org/10.1145/1352135.1352201

[9] Dennis Brylow. 2013. Embedded Xinu - Main Page. (2013). http://xinu.mscs.mu.edu/Main_Page

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

[11] Douglas Comer. 2015. *Operating System Design: The XINU Approach* (2 ed.). CRC Press. 701 pages. https://www.crcpress.com/Operating-System-Design-The-Xinu-Approach-Second-Edition/Comer-Comer/p/book/9781498712439

[12] Mike Galbraith, Dmitry Adamushko, Srivatsa Vaddagiri, Thomas Gleixner, and Peter Zijlstra. 2007. Completely Fair Scheduling. (2007). https://github.com/raspberrypi/linux/blob/rpi-4.9.y/kernel/sched/fair.c

[13] David Jeff Jackson and Paul Caspi. 2005. Embedded Systems Education: Future Directions, Initiatives, and Cooperation. *SIGBED* Rev. 2, 4 (Oct. 2005), 1–4. https://doi.org/10.1145/1121812.1121814

[14] Adam B Mallen and Dennis Brylow. 2010. 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).* ACM, New York, NY, USA, 161–168. https://doi.org/10.1145/1869542.1869568

[15] Marquette University Systems Lab. 2010. Embedded Xinu GitHub. (2010). https://github.com/xinu-os/xinu

[16] Matt Mackal et. al. 2017. Raspberry Pi Foundation Linux Kernel - Github. (2017). https://github.com/raspberrypi/linux

[17] Marco Melletti, Michael Goldweber, and Renzo Davoli. 2015. The JaeOS Project and the µARM Emulator. *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education - ITiCSE '15* (2015), 3–8. https://doi.org/10.1145/2729094.2742596

[18] Paul Miller. 2017. Raspberry Pi sold over 12.5 million boards in five years. (2017). https://www.theverge.com/circuitbreaker/2017/3/17/14962170/raspberry-pi-sales-12-5-million-five-years-beats-commodore-64

[19] Ben Pfaff, Anthony Romano, and Godmar Back. 2009. The pintos instructional operating system kernel. *ACM SIGCSE Bulletin* 41, 1 (2009), 453. https://doi.org/10.1145/1539024.1509023

[20] Raspberry Pi Foundation. 2016. Raspberry Pi 3 Model B Specifications. (2016). https://www.raspberrypi.org/products/raspberry-pi-3-model-b/

[21] Raspberry Pi Foundation. 2016. Raspberry Pi 3 on Sale Now at $35. (2016). https://www.raspberrypi.org/blog/raspberry-pi-3-on-sale/

[22] Raspberry Pi Foundation. 2016. Raspberry Pi Boot Modes. (2016). https://www.raspberrypi.org/documentation/hardware/raspberrypi/bootmodes/

[23] Raspberry Pi Foundation. 2016. The Raspberry Pi UARTs. (2016). https://www.raspberrypi.org/documentation/configuration/uart.md

[24] Raspberry Pi Foundation. 2017. Peripheral Addresses. (2017). https://www.raspberrypi.org/documentation/hardware/raspberrypi/peripheral_addresses.md

[25] Paul Ruth and Dennis Brylow. 2011. An experimental Nexos laboratory using Virtual Xinu. In *Proceedings of the 2011 Frontiers in Education Conference (FIE'11).* IEEE Computer Society, Washington, DC, USA, S2E–1–1–S2E–6. https://doi.org/10.1109/FIE.2011.6143069

[26] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. 2013. *Operating System Concepts* (9 ed.). John Wiley & Sons, Inc.

[27] Andrew S Tanenbaum and Albert S Woodhull. 2006. *Operating Systems Design & Implementation 3rd Edition.* 1099 pages. http://www.amazon.com/s/ref=nb_sb_noss?url=search-alias%3Daps&field-keywords=9780131429383

[28] Gert van Loo. 2014. *ARM Quad A7 core*. ARM. https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2836/QA7_rev3.4.pdf

[29] Michael Ziwisky, Kyle Persohn, and Dennis Brylow. 2013. A down-to-earth educational operating system for up-in-the-cloud many-core architectures. *Trans. Comput. Educ*. 13, 1 (feb 2013), 4:1–4:12. https://doi.org/10.1145/2414446.2414450

ACM ISBN 978-1-4503-6338-9/17/11. . . $15.00
https://doi.org/10.1145/3162087.3162091