Marquette University

e-Publications@Marquette

5-18-2020

Using Embedded Xinu and the Raspberry Pi 3 to Teach Operating Systems

Patrick J. McGee

Rade Latinovich

Dennis Brylow

Using Embedded Xinu and the Raspberry Pi 3 to Teach Operating Systems

Patrick J. McGee
Computer Science, Marquette University, Milwaukee, Wisconsin
Rade Latinovich
Electrical and Computer Engineering, Marquette University, Milwaukee, Wisconsin
Dennis Brylow
Computer Science, Marquette University, Milwaukee, Wisconsin

Abstract:
Multicore processors have become the standard in modern computing platforms. Such complex hardware enables faster execution of the programs it runs, but this is only true if its programmer has the knowledge and ability to make it so. Thus, there is a great need to prepare computing students by establishing robust educational tools. Existing tools often include abstract learning environments such as a virtual machine. While such platforms are widely available and convenient, they are unable to

expose students to concurrency on real hardware.This paper presents multicore Embedded Xinu, an educational operating system used to teach concurrency concepts at the university level. The latest port of Embedded Xinu to the four-core, ARM-based Raspberry Pi 3 B+ enabled an operating systems curriculum in which students build their own concurrency-oriented kernel and execute it on a real machine. Assignments that have been run in the course include concepts of synchronization, scheduling, and memory allocation on a multicore platform. Upon completing the course, students are capable of solving problems commonly found in the field of parallel computing.

SECTION I. Introduction

It is uncommon for an undergraduate-level operating systems (O/S) course to offer a hands-on, project-based curriculum. Some courses use a theoretical approach offering little practical experience, because the logistics of allowing students to directly develop low-level software on laboratory machines is seen as expensive, insecure, and/or difficult to maintain. Studying code of a modern O/S has many pedagogical drawbacks due to both code size and abstractions necessary to support many disparate platforms. Another common approach is to use virtual machines, which can provide greater scalability, protection, and customization. However, virtual machines by their very nature can abstract away some of the key details that students would face when working with real hardware. A proven middle ground has been the use of educational operating systems – code developed specifically for students to experiment with and learn about O/S components, often in a protected execution environment. While many historical educational operating systems have ceased to be maintained, the continued vigor of the Xinu operating system creates an opportunity for combining hands-on O/S projects steeped in important parallel and distributed computing concepts.

A. Background

Embedded Xinu descends from Xinu, an operating system created in the 1980s by Douglas Comer. The educational aspect of Xinu is described in detail in several editions of his popular textbook [1]. Xinu was originally written to run on LSI-11 minicomputers and was later ported to run on various CISC architecture machines, including x86, Sun Microsystems SPARC, and Motorola 68000 [2].

In the 21st century, Xinu was ported to inexpensive RISC platforms and Embedded Xinu was born [3]. Embedded Xinu was first ported to the PowerPC architecture, and later to MIPS hardware in consumer networking appliances [4], [5]. The ARM port of Embedded Xinu was targeted to the ubiquitous Raspberry Pi platform [6]. Other branches of the original Xinu operating system have subsequently been ported to the Intel x86-based Galileo board and the ARM Cortex-A8 BeagleBone Black board [2].

For an undergraduate computer science curriculum, an operating systems course has the potential to incorporate previously-learned software design concepts in such a way that enhances the student's ability to program embedded systems. Recent work porting Embedded Xinu to the multicore Raspberry Pi 3 B+ enabled three semesters of concurrency-oriented projects in Marquette University's Operating Systems course [7]. It should be noted that while the Pi 3 B+ is equipped with an ARMv8-A 64-bit processor [8], multicore Xinu runs in ARMv7 32-bit mode. This design decision was made to avoid the increased complexity of 64-bit programming, thus maintaining the concise nature of a RISC platform used to teach undergraduates.

In the spring of 2018, multicore Embedded Xinu was deployed for the first time in this course. An initial assignment late in the course made use of all four cores to brute-force an encrypted DES message in substantially reduced time – an embarrassingly parallel application. In the fall of 2018, multicore Embedded Xinu was incorporated into the second-year Hardware Systems course at Marquette to demonstrate aspects of parallel computing at the lowest levels of the machine. [9]. In 2019 and 2020, multicore Embedded Xinu was incorporated fully into the O/S course, bringing hardware concurrency topics directly into six of the eleven semester project assignments, as we later describe in Section IV.

B. Laboratory Environment

The laboratory environment of Marquette University's Computer Science Department, ("The Systems Lab",) enables students to run their kernel on a backend machine, such as a Raspberry Pi. With around 100 computing students requiring use of backend machines, efficiency is a top priority. After cross-compiling the kernel on a UNIX Systems Lab machine, a student can upload the bootable image to a backend by issuing a command that selects a machine from the pool and invokes the machine to boot over the network (see Figure 1). This bootstrapping process is similar to that of Purdue's Xinu laboratory [10]. Each backend machine is connected to (1) a rebooter unit, responsible for supplying power to an invoked machine; and (2) a serial port aggregator that provides a facility for input/output between a backend machine and the UNIX machine.

When a student is confident of their implementation, they submit their work via a UNIX Systems Lab machine. A nightly run of automatic, instructor-defined testcases are emailed to students each morning, helping students to come to lecture prepared to ask questions about requirements of the current assignment.

This paper details the technical challenges overcome to support a bare-metal, educational operating system on genuine concurrent multicore hardware. Building on parallel computing concepts covered in prior courses, we have developed a sequence of hands-on development projects for our required second-year Operating Systems course that highlight support for multicore machines. This paper presents the new project sequence, and describes our experiences and lessons learned deploying this material in the classroom.

SECTION II. Related Work

The ACM/IEEE Joint Task Force on Computing Curricula 2013 recommendations include 15 hours of tier-1 and tier-2 coverage for parallel and distributed computing, as well as 15 hours of operating systems content [11]. Major topics under the operating systems rubric include:

- Concurrency
    - Managing atomic access to O/S objects,
    - Implementing synchronization primitives, and
    - Multiprocessor issues (such as spinlocks and reentrancy);
- Scheduling and dispatch
    - Preemptive and non-preemptive scheduling,

- Schedulers and policies, and

- Processes and threads; and

- Memory management.

The parallel and distributed computing recommendations include such foundational concepts as atomicity and race conditions, mutual exclusion, blocking vs. non-blocking messaging, and assembly-level support for parallelism.

However, despite the clear synergy between the operating systems topics and parallel/distributed topics, little prior work has focused on enabling undergraduate students in operating systems courses to gain practical experiences while learning about these concepts. The dearth of suitable teaching materials in this realm is understandable given the complexity of the topic, but the absence of educational operating systems with support for multicore hardware has remained the most formidable stumbling block prior to our previous work porting Embedded Xinu to the multicore Raspberry Pi.

The tendency in the parallel/distributed education community has been to focus either on higher level abstractions – activities that begin with the assumption of a fully-formed operating system and standardized interfaces, such as OpenMP [12] (with a multicore processor) or MPI [13] (between parallel processors); programming language constructs for multithreading and synchronization; or special-purpose hardware, such as GPUs. Much of the effort has accrued toward the implementation of high level undergraduate elective or graduate level courses, such as in [14], but there is growing recognition in the community that introducing parallel and distributed content earlier in the course sequence can yield significant benefits. For example, a 2019 paper [15] presented hands-on teaching modules using OpenMP and the Raspberry Pi to integrate parallel programming concepts into undergraduate courses, particularly in the first and second year.

Other work has concentrated on developing suitable abstractions and tools to be able to introduce younger students to the high level concepts of parallelism well before they acquire the programming proficiency necessary to work on low-level system development [16].

The Parallel and Distributed Computing course at UNC Charlotte, described in [17], has many features in common with our work. They use a dedicated PBS cluster to run student code without disrupting the production cluster, as we use a dedicated pool of multicore Raspberry Pi boards, remotely available on demand. They emphasize basic familiarity with C/C++ and the UNIX toolchain, as do we. They build a series of projects that increase the level of abstraction available, ranging from a pthreads implementation up to the MR-MPI library providing Map Reduce features. Our work differs primarily in level, focusing on a second-year, rather than third-year course, that builds the operating system from lowest level hardware operations through to the equivalent of the pthreads library. The next logical step for us is to adopt a similar Parallel and Distributed Computing course that builds directly upon the work our students have completed in the prerequisite Operating Systems course.

The graduate-level Advanced Operating Systems course at the University of Maine uses vmwOS, an educational O/S that runs on the Raspberry Pi [18]. This O/S shares many features that are central to Xinu, such as multicore support complete with a scheduler, blocking I/O, and device drivers. However,

one design goal of vmwOS is to support a 64-bit architecture, while multicore Xinu implements a 32-bit mode to teach undergraduate-level Operating Systems.
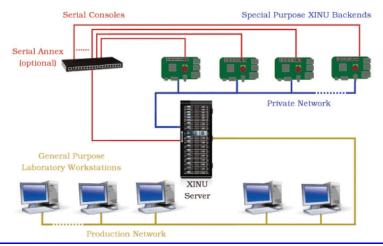

Fig. 1. Marquette University Systems Lab configuration.

Another similar work is from the University of Calgary which uses a simplified bare-metal O/S to teach assembly language in their undergraduate Computing Machinery course [19]. This course employs a unique approach in which students build an interactive video game written entirely in ARM assembly language. At Marquette University, one prerequisite for Operating Systems is Hardware Systems. In this course, first and second-year majors become experienced in writing ARM assembly through a series of weekly projects that begins with concepts such as conditional branching and ends with recursion using activation records. The kernel used in these projects is a more stripped-down version of multicore Xinu that – after bootstrapping – simply executes the student's ARM code. This is a good example of how our work can support curriculum in multiple systems courses that use the Raspberry Pi.

Systems such as the Habanero Autograder [20] can provide valuable scaffolding for students working on parallel and distributed systems. Our automatic evaluation system based upon the Xest tool [21] provides nightly feedback to students as they build the components of their multicore operating system, a similar scaffolding mechanism focused more on multicore remote-target embedded devices.

SECTION III. Multicore Additions

Expanding the educational O/S Embedded Xinu to properly support a multicore platform is challenging because of the tension between its minimalist design and the additional complexities of managing concurrent hardware. Embedded Xinu is designed to be understood in its entirety by a midcareer undergraduate computing major in a single term. While many modern operating systems have already solved the fundamental technical barriers to supporting multiple cores, those solutions rarely scale down suitably to be incorporated into the minimal design aesthetic of a system like Xinu. The challenge is to build the simplest new component that accomplishes the task at hand, without getting lost in the low level architectural details.

Compounding the complexity of this task, poor public documentation of modern processors and platforms makes it particularly challenging to program modern machines without the assistance of a full-blown desktop O/S. The next sections detail specific technical hurdles that must be conquered by a multicore O/S, and describe the Embedded Xinu solution.

A. Process Scheduler

The process scheduler on multicore Xinu is similar in functionality to the scheduler present on the single-core version of Embedded Xinu. Adaptations of the single-core scheduler are required in order to function properly in a multicore environment. Priority-based preemptive process scheduling is employed in both the single-core and multicore versions.

Four ready list queues are used in the scheduler, one for each core. Each ready list is a doubly-linked priority queue. The values stored in the queue are thread ID's (tid). Each tid has a corresponding structure called a thread entry (thrent). Thread entries contain information about the thread such as the thread state, stack pointer, thread name, parent thread, and more as shown in Figure 3. The thrent structure is equivalent to a traditional process control block. The thread entries are kept in a thread table array, (thrtab), which associates a tid with its respective thrent [7].

In Embedded Xinu, a thread has a certain flow, or life cycle with respect to its current state and status. When a thread is first created with the create() function, it is initialized to be in a suspended state (THRSUSP). In order for a thread to begin execution it must be queued in one of the core ready lists with the ready() function. The state will be updated to indicate it is ready to run, or THRREADY. At this point, the scheduler can context switch the currently running thread with one that is in the ready queue. Context switching can be triggered by timer preemption or by the current thread calling resched(). When a thread is running on a core it will have a state of THRCURR. From here, a thread can also be put in a sleeping state (THRSLEEP), a waiting state (THRWAIT), or the thread can be killed. This is summarized in Figure 2.
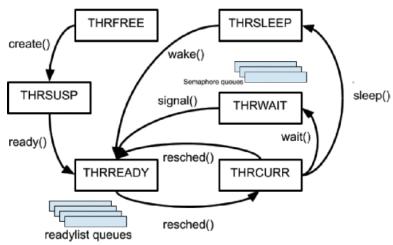

Fig. 2. Embedded Xinu Process State Diagram.

This thread life cycle is very similar to single-core Embedded Xinu [6], the difference being that there are now multiple instances of it occurring at the same time and synchronization is now a critical factor to take into consideration when implementing and adapting the existing scheduling algorithm.

B. Timer Interrupts and Preemption

Timer interrupts are necessary for preemption to occur. For preemptive multicore scheduling, timer interrupts need to occur on each of the four cores. Timer interrupts occur based on a system clock. A standard mechanism for triggering timer interrupts has two basic elements. The first element is a clock count – a free-running count of how many clock ticks have occurred. The second element is a way to compare the free-running clock count to a value, typically referred to as the *compare value*. A timer interrupt is triggered when the clock count value and the compare value intersect with each other in some fashion.

```
struct thrent
{
    uchar state;       /* thread state           */
    int prio;          /* thread priority        */
    void *stkptr;      /* saved stack pointer     */
    void *stkbase;     /* base of run time stack  */
    ulong stklen;      /* stack length in bytes   */
    char name[TNMLEN];      /* thread name        */
    irqmask intmask;   /* interrupt mask          */
    semaphore sem;     /* semaphore waiting for    */
    tid_typ parent;    /* tid for the parent thread */
    message msg;       /* message sent to this thr. */
    bool hasmsg;       /* nonzero iff msg is valid  */
    struct memblock memlist; /* free memory list   */
    int fdesc[NDESC];       /* device descriptors  */
    uint core_affinity;     /* core affinity       */
};
```
Fig. 3. thrent struct

There were two main documents that we found during our investigation of the timer interrupts on this platform. One of the documents was written for the BCM2836 System-on-Chip (SoC) [22], the chip present on the Raspberry Pi 2. The other document is the ARMv8-A Architecture Reference Manual [23] provided by ARM. These two documents provide different details about the timer devices. The BCM2836 document indicates that the timers are accessible by memory-mapped peripheral space, while the ARM document shows that the timer registers are accessible directly via the co-processor registers. We were not able to find any documents that show any direct relation between the two sources. Reading between the lines in both documents yielded working results.

To initialize the timer on a core, the Physical Timer Control Register (CNTP_CTL) is used to enable the timer in conjunction with the timer interrupt control memory-mapped register. The CNTP_CTL register has three bits, ISTATUS, IMASK, and ENABLE [23]. The ISTATUS bit indicates whether the timer condition is met. This refers to the timer value and the compare value, which will be discussed in the next paragraph. The IMASK is the interrupt mask bit, which determines whether the timer interrupt is masked or not. Lastly, the ENABLE bit enables or disables the timer. For the memory-mapped register, we enable interrupt request signals for nCNTPNSIRQ and nCNTPSIRQ, which correspond to Physical Timer Non-Secure IRQ and Physical Timer Secure IRQ. The initialization sequence is done in assembly when a core starts up and is shown in Figure 4.

To trigger an interrupt from a timer, the Timer Count Register (CNTPCT) register is used with the Timer CompareValue register (CNTP_CVAL). The CNTPCT register holds the 64bit physical count value, and the CNTP_CVAL register holds compare value for the timer. If the timer is enabled, the timer condition

is met when CNTPCT - CNTP_CVAL is greater than or equal to zero [23]. An interrupt is generated when the timer condition is met and if the CNTP_CTL.IMASK bit is 0.

```
/* initialize Physical Timer */
mrc     p15, 0, r2, c14, c2, 1 // Read CNTP_CTL
bic     r2, #0b110         // ISTATUS and IMASK to 0
orr     r2, #0b001         // Set enable bit
mcr     p15, 0, r2, c14, c2, 1 // Write CNTP_CTL

ldr     r2, =0x40000044 // Core 1 timer
                        // interrupt control
ldr     r3, [r2]
bic     r3, #0b11111100
orr     r3, #0b11          // nCNTPNSIRQ and
                           // nCNTPSIRQ irq enable
str     r3, [r2]
```

Fig. 4. Initializing ARM Timer on Core 1

C. Support for Atomic Operations

Atomic operations are required in a multicore context to avoid potential issues regarding cache coherency and race conditions. Simple atomic operations can be useful as a form a basic synchronization or can be used to implement more sophisticated methods of synchronization, such as semaphores or condition variables.

Implementing complex atomic operations requires the use of hardware synchronization primitives provided by the platform. Atomic hardware primitives vary between processor architectures. For example, some platforms provide exclusive load and store operations (ldrex/strex) while some others provide test-and-set (tsl) or compare-and-swap (cas) operations. On the Raspberry Pi 3 B+, which is an ARM platform, ldrex and strex opcodes are used to implement these atomic operations [23].

In order to avoid potential issues with different hardware synchronization schemes, Embedded Xinu provides a set of operations that will behave the same, regardless of the platform (see Figure 5). The interface contains three operations that must be implemented on each hardware platform, including atomic increment (_atomic_increment), atomic decrement (_atomic_decrement), and atomic increment modulus (_atomic_increment_mod) operations. Descriptions of the atomic operations implemented are given in Figure 5.

Atomic increment and decrement can be used to implement simple locks shared across cores. The atomic increment with modulus is particularly useful for allocating slots in any static array kernel data structure, such as a new thread ID in the global thread table, without fear of race condition in the initialization phase.

D. USB Driver and DMA Buffers

On the Raspberry Pi 1 single-core platform, the memory caches did not require any specific configuration in Embedded Xinu, as they are disabled by default [6]. With the Raspberry Pi 3 B+, the cache was required to be enabled and configured so that all of memory is cacheable, except for the memory-mapped peripheral address space [7]. One area of memory that was not initially taken into consideration was memory used for Direct Memory Access (DMA) transactions.

```
/**
 * @fn int _atomic_increment(int *var)
 *
 * Atomically increment specified integer.
 * (Equivalent to ++var in C).
 *
 * @param *var pointer to variable
 * @return value the variable held after
 *         being incremented.
 */
int _atomic_increment(int *var);

/**
 * @fn int _atomic_decrement(int *var)
 *
 * Atomically decrement specified integer.
 * (Equivalent to —var in C).
 *
 * @param *var pointer to variable
 * @return value the variable held after
 *         being decremented.
 *
 */
int _atomic_decrement(int *var);

/**
 * @fn int _atomic_increment_mod(int *var, int mod)
 *
 * Atomically increment and mod with value.
 * Equivalent to var = (var + 1) % mod.
 *
 * @param *var  pointer to variable
 * @param mod value to wrap—around if
 *            var exceeds it.
 * @return incremented var value
 */
int _atomic_increment_mod(int *var, int mod);
```

Fig. 5. include/atomic.h - interface for atomic operations

Cache coherency becomes an issue when multiple cores are reading and writing to the same, shared physical memory location. Because DMA transactions are used heavily within Embedded Xinu's USB subsystem, a section of memory is flagged as uncacheable to protect the DMA transactions from cache coherency errors.

SECTION IV. Multicore Operating Systems Course

Within the past year of this writing, multicore Xinu has been used in six assignments in Operating Systems at Marquette University. In this course, approximately 80 students from three different majors (computer science, computer engineering, and biocomputing) work in small teams to build foundational operating system components in a series of weekly, cumulative assignments.

While the full multicore Xinu kernel is lightweight, students are given a stripped-down version of the kernel – consisting of a few hundred lines of embedded C and ARM assembly code – to reduce as much cognitive load as possible. This section unpacks the details of each multicore assignment, including samples of code where instructive.

A. Multicore Synchronization Primitives

Following the first O/S assignment in which students build a synchronous serial driver, this introductory multicore concurrency assignment presents the task of enforcing basic protection among the four cores. At this point in the course, the students have gained experience in C programming through two previous warm-up assignments. In addition, they have had practice working in a UNIX commandline environment, as it is important for them to become comfortable with cross-compiling their kernel and uploading the bootable image to a Raspberry Pi.

In the worst case, all four cores are trying to access the single PL011 UART [24] serial device at the same time. Without protecting each access, there will be undesirable results. For example, if multiple cores are trying to print the string "Hello Xinu World!", and each core has uncoordinated access to the UART control and status registers, then the output will be unreadable (see Figure 6). Like most assignments in this course, the students begin with this inadequate scenario and it is their task to resolve it. Simply by replicating their existing O/S on the other three cores, the students quickly see how concurrency can ruin the perfectly good device driver they constructed in the previous week, because the functions are not reentrant.

```
void core_print(void)
{ while(1)
    kprintf("Hello_Xinu_World!\r\n"); }
...
unparkcore(1, (void *) core_print, NULL);
unparkcore(2, (void *) core_print, NULL);
unparkcore(3, (void *) core_print, NULL);
```

```
Output:
ell   iiuuWWoll!HHllooXXnnuWWrrld!Hell
XXiuuWWrrld!eell XXiuu oolddHeello Xnn
oorldHHello Xin   oolldHHell  iinu World!
HelloXXnn  Wolld!eell  XnnuWWrrddHHello
XinuWWorl!HHello iiuuWWrrd!!elloo Xinu
World!HelooXXinu World!Hello iiuu
Wordd!elll XXnnuWWr
```

Fig. 6. unparkcore() [7] gives printing job to three cores, resulting in garbled output due to improper synchronization.

To solve this problem, the students must implement an assembly routine that makes proper use of on-chip synchronization primitives (ldrex, strex instructions) [25]. Such a routine will be called with each access of the UART. Completion of this routine exposes a central principle: O/Ses for multicore architectures should invoke instructions that provide atomic memory updates. In addition, the class gains immediate familiarity with developing solutions to the concurrency problems posed by a multicore architecture.

B. Non-preemptive Multicore Scheduling

The follow-up assignment to Multicore Synchronization is the Non-Preemptive Multicore Scheduling assignment, in which students add a thread abstraction onto each of the cores. This initial exercise in multithreading uses only cooperative (non-preemptive) scheduling for simplicity. Preemptive multitasking is added in the subsequent assignment. The stages of this assignment require students to:

- Build an assembly routine to switch process contexts;

- Modify the incomplete function create() to consider multicore processes; and

- Test their implementations.

In prior years, this O/S assignment only required students to handle single-core multithreading, but this was already considered to be a challenging project. Students must understand the thread control block structure, complete the functions for initializing a new thread's context, and correctly match that

context to the assembly language context switch routine. With the new multicore implementation, additional scaffolding is required. Provided code includes fields and functions that students may not initially recognize will be necessary later, such as the thread entry structure containing a field named core_affinity, describing the core number (see Figure 3). As a preliminary step, the class must study this structure in order to understand how to properly initialize a thread entry.

For simplicity, we do not allow threads to migrate between cores once started, nor can a thread kill another running thread on a different core.

C. Preemptive Multicore Scheduler

The subsequent assignment tasks students with adding preemption to their multicore scheduler. Students implement round-robin priority scheduling using three priority queues (representing low, medium, and high priority) per core.

For an undergraduate, it may not be obvious that changes to the underlying clock interrupt handlers are required for real preemption on multiple cores (see Figure 7). Therefore, like previous assignments, the several complicated components are already scaffolded; the student is required to establish preemption by modifying the scheduler-related functions.

To make proper use of the priority system, the students must first understand the following:

- The ready list is now a two-dimensional queue;

- Each core has its own timer, so it is rational to store respective timing values in an array; and

- If aging is enabled, a thread may be promoted to a higher priority queue, avoiding starvation.

The value of this assignment comes with students' full interaction with their multicore preemption system. In testing their implementation, students can test prioritization in isolation and then introduce cases for aging.

D. Multicore Semaphores for the Asynchronous Serial Driver

Having written a synchronous serial driver, students understand the implications of writing software that waits for a slow I/O device. Now, students are tasked with developing an asynchronous, interrupt-driven driver for the UART. The challenge presented by a multicore architecture is ensuring the semaphore variables are safe from destructive updates by competing cores.

```
volatile ulong clkticks[NCORES];  /* ticks    */
volatile ulong clktime[NCORES];    /* seconds */
#define QUANTUM 3 /* ticks until preemption */
#if PREEMPT
volatile ulong preempt[NCORES];
#endif
#if AGING
volatile ulong promote_medium[NCORES];
volatile ulong promote_low[NCORES];
#endif

void clkinit(void)
{          int i;
#if PREEMPT
        for (i = 0; i < NCORES; i++)
            preempt[i]         = QUANTUM;
#endif
#if AGING
        for (i = 0; i < NCORES; i++)
        {   promote_medium[i] = QUANTUM;
            promote_low[i]     = QUANTUM;  }
#endif
        for (i = 0; i < NCORES; i++)
        {   clkticks[i]        = 0;
            clktime[i]         = 0;  }
}
```

Fig. 7. A subset of the clock system that initializes preemption values.

The structure of the higher-level functions of this driver (getc(), putc(), printf()) is similar to that of the synchronous one, but after viewing the UART interrupt handler, it will become clear to the student that the given implementation is only functional when a single core is using the UART. The student is tasked with resolving a broken semaphore implementation by using the atomic increment and decrement operations as synchronization primitives. Semaphores will then be used to complete the asynchronous serial driver portion of the assignment.

E. Heap Memory on a Multicore Platform

In the final multicore assignment, the focus of the course switches to memory management in an operating system. A free list of available memory blocks can be accessed by any of the four cores at once, making the environment ripe for race conditions. The student's malloc() and free() procedures require avoidance of such race conditions. If the proper synchronization steps are taken (e.g., using spinlocks), these memory operations can be trusted to be mutually exclusive.

SECTION V. Outcomes and Lessons Learned

In spring of 2019, Marquette Computer Science fully integrated multicore Embedded Xinu into its COSC 3250 Operating Systems course, also cross-listed as COEN 4820 in the Department of Electrical and Computer Engineering. Students implemented the projects described in the previous section in teams of two, typically with partners assigned across majors. (At Marquette, computer science majors take prerequisite courses that expose them to ARM assembler, the UNIX tool chain, and the Embedded Xinu ARM Playground [9]; computer engineering and biocomputing majors take courses that expose them to C programming, more depth in computer architecture topics, and embedded devices. Teams with both majors are better equipped to tackle the complex projects in Operating Systems.) Computer

science majors typically take this required course in their fourth semester, while engineering majors normally take it in their sixth semester.

In this context, it is difficult to quantitatively measure the impact on student learning outcomes. While we included exam questions that might measure a change in students' fluency with concurrency and synchronization concepts, the comparison with a prior "control group" of students in a previous term proved problematic. The students in the multicore group performed *worse* on the concurrency question than the students who answered an identical exam question in a previous term that used only single-core Embedded Xinu assignments. (The average score was lower, but the difference was not statistically significant.) However, they also performed worse on other questions that had nothing to do with concurrency, and averaged markedly lower on the exam scores overall. While some statistical analysis might be possible to tease out the independent effect on the concurrency exam question, other confounding factors – a 75% larger class size, changeover in teaching assistants and graders, and substantial changes in prerequisite curricula for all three groups of students – undermined our efforts to make an "apples to apples" comparison of student learning via exam questions in this quasi-experiment.

Our intent at the outset was to directly compare student performance on each of the multicore project assignments against their single-core counterparts from the previous term. However, in the course of adding multicore design requirements to assignments, we found that the intellectual load had, in some cases, shifted enough that other aspects of the assignment had to be simplified. Moreover, as part of a required, implementation-heavy course with a larger enrollment, the instructors generally avoid reuse of specific assignment variants in adjacent terms, and this further complicated efforts to directly compare projects. Finally, the order of project topics had to be shifted to accomodate multicore needs. For example, where previously students built their single-core operating system context switch in the fourth project, in the multicore sequence it is necessary to built a spinlock mechanism to guard the serial port device driver critical section before the multicore context switch can be written, otherwise intermingled output from each of the cores will be illegible. For all of these reasons, we are left with primarily qualitative data to evaluate our results.

Anecdotally, students were generally excited to be on the "bleeding edge" of curriculum development, and thought the new focus on multicore concepts fit well with other structural changes in their major curricula.

The first assignment, on multicore synchronization primitives, went very well. A large number of teams successfully implemented the multicore spinlock system, and successfully guarded their serial port driver critical sections. Students would rely on this code to get clean output from each of the cores for the rest of the term. Instructors and teaching assistants noted many "aha!" moments in the laboratory and in office hours, with students making connections between the project and content from previous courses, such as seeing a useful application for ARM opcodes ldrex and strex. Students dug into the provided ARM opcode documentation, and expressed high levels of satisfaction when the assignment was completed.

Similarly, the cooperative scheduling assignment went well. Despite historically being one of the harder projects in the sequence, the addition of multicore issues to the project did not appear to substantially alter the cognitive load of an already challenging assignment.

The addition of timer-based preemption in the third assignment caused more consternation, in large part because any minor bugs that crept through in earlier assignments tended to be greatly compounded by the combination of both preemption and multiple cores. Debugging grew substantially more complex with only a slow serial port to observe what was transpiring on each of the cores. (Embedded Xinu has a remote-target debugging capacity that supports single-step, breakpointing, and register modification – but this hardware and software has not yet been ported forward to the multicore Pi 3 B+ boards.) Teams found that minor testcases that had failed in previous assignments took on much greater significance if left to fester. In the current term, we are emphasizing the importance of addressing those minor bug testcases, even though the system may be appear to be working well overall.

The asynchronous device driver with multicore semaphores seemed to be one of the more challenging projects in the term. This was both because of accumulated errors in the student code, and also due to the complexity of managing both multicore concurrency and interrupt-driven device driver buffering. Of note, we had not previously used the interrupt-driven device driver project in the prior five years, because it had not been a favorite of students. However, the importance of introducing interrupt-driven asynchrony in the system to replace inefficient spinlocks seemed like an irresistible justification. In the current term, we are planning to try a different fourth project in the sequence that implements another portion of the pthreads API, such as condition variables.

Finally, the core-safe memory allocation project went smoothly, perhaps because of the relative ease of implementation after the two previous more challenging projects.

In summary, a large number of the teams successfully completed the sequence of multicore O/S component projects, and could correctly claim that they had implemented large segments of a multicore, interrupt-driven, preemptive multitasking operating systems with resource allocation and synchronization primitives. The course continues to have a reputation as a transformative experience that students look back upon fondly, once they have completed it.

SECTION VI. Summary and Conclusion

Embedded Xinu is an established educational operating system that has provided university students a hands-on approach to learning operating systems concepts for decades. While many other educational operating systems have fallen into disuse and disrepair, Xinu's continued vigor creates a unique opportunity for new hands-on experiences with parallel computing content. Our recent work porting Embedded Xinu to the multicore Raspberry Pi 3 B+ platform enables us to create relevant, rigorous project assignments in our required lower-division undergraduate operating systems course that highlight multicore concurrency at the O/S level. Students build the core components of their embedded O/S, and complete a substantial portion of a concurrency API roughly equivalent to a pthreads library by the end of the term.

This paper has presented the major technical hurdles that must be overcome to extend an educational operating system to a multicore platform, while retaining the minimalist design that makes the codebase tractable to undergraduates in a single semester course. We outlined a series of cumulative project assignments, and detailed our experiences and lessons learned fielding these projects in a course with 80+ students across three groups of majors in computer science, computer engineering, and biocomputing.

As the prevalence of parallel computing platforms continues to grow, it is essential to provide appropriate models and teaching tools for students to encounter this material in their lower-division core systems courses.

SECTION VII. Future Work

A. Non-Blocking Concurrent Data Structures

Non-blocking data structures are of interest because they provide many benefits in terms of performance, namely the avoidance of deadlocks. The two places that we would use these data structures would be for thread ready list queues and for the memory freelist. Using non-blocking data structures would increase the complexity of Embedded Xinu. Creating an assignment based off of concurrent data structures would likely be out of scope for an introductory undergraduate course, but could be of value in a more advanced course.

B. Memory Protection and Virtual Memory

Memory protection has been implemented in previous iterations of Embedded Xinu, but has not yet been adapted for the Raspberry Pi platforms. Implementing memory protection would allow each thread to have its own memory space protected from other threads.

An implementation of virtual memory paired with memory protection would make concepts such as having a user-space or process migration become more straightforward to implement.

ACKNOWLEDGEMENTS

References

1. D. Comer, Operating System Design: The Xinu Approach, CRC Press, 2015.

2. D. Comer, The Xinu page, [online] Available: https://xinu.cs.purdue.edu/.

3. D. Brylow, Embedded Xinu, [online] Available: https://www.cs.mu. edu/~brylow/xinu/.

4. D. Brylow, "An experimental laboratory environment for teaching embedded hardware systems", *Proceedings of the 2007 Workshop on Computer Architecture Education*, pp. 44-51, 2007, [online] Available: https://doi.org/10.1145/1275633.1275643.

5. D. Brylow, "An experimental laboratory environment for teaching embedded operating systems", *SIGCSE Bull*, vol. 40, no. 1, pp. 192-196, Mar. 2008, [online] Available: https://doi.org/10.1145/1352322.1352201.

6. E. Biggers, F. Harunani, T. Much and D. Brylow, "XinuPi: Porting a lightweight educational operating system to the raspberry pi", *WESE 2013: Workshop on Embedded and Cyber-Physical Systems Education Montreal Quebec*, Oct 2013.

7. P. Bansal, R. Latinovich, T. Lazar, P. J. McGee and D. Brylow, "Xinupi3: Teaching multicore concepts using embedded xinu", *CSERC '17: Proceedings of the 6th Computer Science Education Research Conference*, pp. 20-25, Nov 2017.

8. Raspberry pi 3 model b+ on sale now at \$35, [online] Available: https://www.raspberrypi.org/blog/raspberry-pi-3-model-bplus-sale-now-35/.

9. B. Levandowski, D. Perouli and D. Brylow, "Using embedded xinu and the raspberry pi 3 to teach parallel computing in assembly programming", *EduPar 2019 the 9th NSF/TCPP Workshop on Parallel and Distributed Computing Education*, May 2019.

10. D. Comer, An example of booting over a network, [online] Available: https://www.cs.purdue.edu/homes/dec/xinu/page-574.pdf.

11. Computer science curriculum 2013, December 2013, [online] Available: https://www.acm.org/education/curricula-recommendations.

12. A. A. Younis, R. Sunderraman, M. Metzler and A. G. Bourgeois, "Case study: Using project based learning to develop parallel programing and soft skills", *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 304-311, May 2019.

13. L. Alvarez, E. Ayguade and F. Mantovani, "Teaching hpc systems and parallel programming with small-scale clusters", *2018 IEEE/ACM Workshop on Education for High-Performance Computing (EduHPC)*, pp. 1-10, Nov 2018.

14. P. Chitra and S. K. Ghafoor, "Activity based approach for teaching parallel computing: An indian experience", *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 290-295, May 2019.

15. S. J. Matthews, J. C. Adams, R. A. Brown and E. Shoop, "Exploring parallel computing with openmp on the raspberry pi", *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, pp. 1234, 2019, [online] Available: https://doi.org/10.1145/3287324.3287535.

16. E. Buzek and M. Kruli?s, "An entertaining approach to parallel programming education", *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 340-346, May 2018.

17. E. Saule, "Experiences on teaching parallel and distributed computing for undergraduates", *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 361-368, 2018.

18. P. Francis-Mezger and V. M. Weaver, A raspberry pi operating system for exploring advanced memory system concepts, Oct 2018, [online] Available: http://web.eece.maine.edu/~vweaver/projects/vmwos/2018memsysos.pdf.

19. J. Kawash, A. Kuipers, L. Manzara and R. Collier, "Undergraduate assembly language instruction sweetened with the raspberry pi", *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, pp. 498-503, 2016, [online] Available: https://doi.org/10.1145/2839509.2844552.

20. M. Grossman, M. Aziz, H. Chi, A. Tibrewal, S. Imam and V. Sarkar, "Pedagogy and tools for teaching parallel computing at the sophomore undergraduate level", *Journal of Parallel and Distributed Computing*, vol. 105, pp. 18-30, 2017, [online] Available: http://www.sciencedirect.com/science/article/pii/S0743731517300047.

21. M. H. Netkow and D. Brylow, "Xest: An automated framework for regression testing of embedded software", *Proceedings of the 2010 Workshop on Embedded Systems Education*, 2010, [online] Available: https://doi.org/10.1145/1930277.1930284.

22. G. van Loo, ARM quad A7 core, 2014, [online] Available: https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2836/QA7_rev3.4.pdf.

23. ARM architecture reference manual armv8 for armv8-a architecture profile, 2019, [online] Available: https://static.docs.arm.com/ddi0487/ea/DDI0487E_a armv8_arm.pdf?_ga=2.216594262.1753366973.1581287042-1117839992.1566109547.

24. Primecell® uart (pl011), 2005, [online] Available: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0183f/DDI0183.pdf.

25. ARM synchronization primitives development article, [online] Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dht0008a/ch01s02s01.html.