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

Patrick McGee, Rade Latinovich, Dennis Brylow

Marquette University

Outline

- Objectives
- Similar teaching tools
- Multicore support for Embedded Xinu
- Operating Systems Assignments
- Outcomes
- Future work

Introduction

- It is uncommon to teach OS in a hands-on way
  - Logistics are seen as expensive and difficult to maintain
- Instead, OS courses are often taught using:
  - Examinations of a large kernel such as Linux
    - Limits student’s scope to a single component
  - Virtualized platforms (e.g., Oracle’s VirtualBox)
    - Hardware details are inherently abstracted from the student

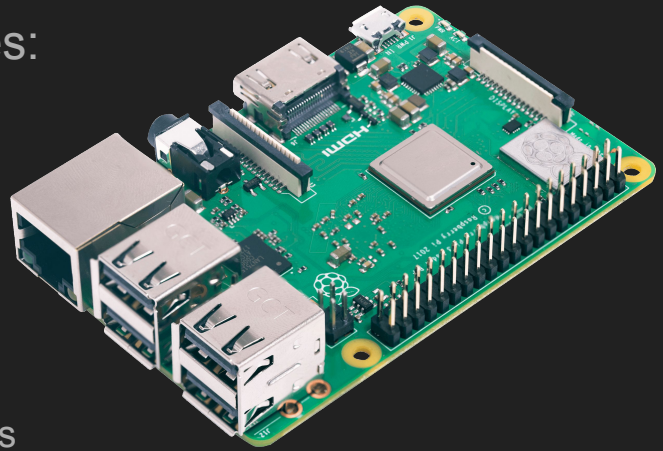
Educational Operating Systems

- Bring real hardware interaction up front
- Embedded Xinu Operating System in courses:
  - Operating Systems
  - Hardware Systems
  - Compilers
  - Embedded Systems
- Used at other universities (XINU Labs)

https://cse.buffalo.edu/~bina/cse321/fall2017/Lectures/XinuIntroOct19.pptx

Multicore Embedded Xinu

- Embedded Xinu descends from Xinu (1980s, Douglas Comer)
  - Xinu originally supported CISC machines
- Embedded Xinu was designed for RISC machines:
  - PowerPC, MIPS, Raspberry Pi
- Ported to Pi 3 B+ in 2017
  - $35 USD
  - ARMv8-A CPU running in 32-bit (ARMv7) mode
  - Enabled 3 semesters of concurrency-oriented OS projects



Recommended Curricula

- ACM/IEEE CS 2013 recommends 15 hours of parallel undergrad education
- Major topics explored in an OS course:
  - Concurrency
  - Scheduling
  - Memory management
- Parallel distributed concepts include:
  - Atomicity & Race conditions
  - Mutual exclusion
  - Blocking and Non-blocking messaging
- Little prior work has focused on teaching parallel concepts in an OS course

Existing Educational OSes

- vmwOS, University of Maine
  - Graduate-level OS Course
  - Features in common with Xinu:
    - Multicore support (including a scheduler)
    - Blocking I/O
    - Device drivers
  - Difference: supports a 64-bit architecture
- University of Calgary's bare-metal OS
  - Undergraduate-level Computing Machinery course
  - Students build an interactive video game written in ARM assembly language
  - Marquette's Hardware Systems course teaches ARM assembly
    - From conditional branching to recursion using activation records

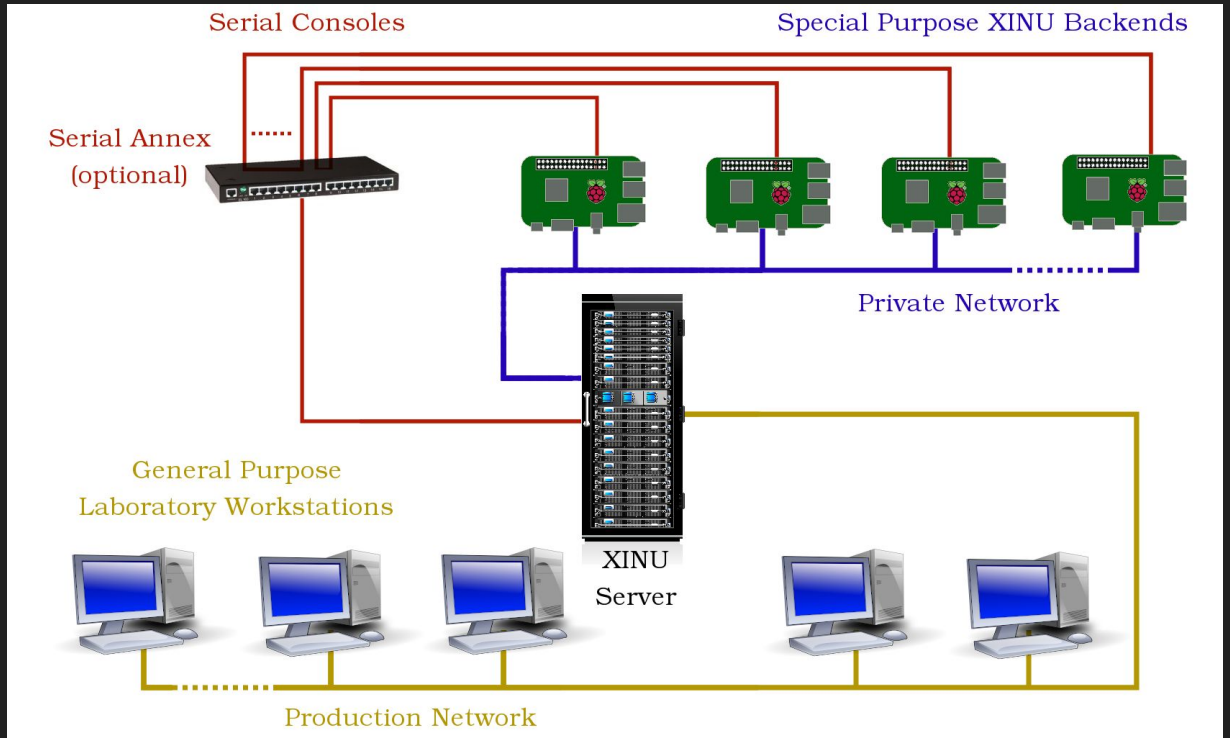
Other Related Work

- UNC Charlotte's Parallel and Distributed Computing course
  - Dedicated cluster runs student code
  - Basic familiarity with C/C++ and UNIX toolchain
  - Build a series of projects to increase level of abstraction available
    - Pthreads
    - MR-MPI library
  - Focuses on a third-year course

- "Exploring Parallel Computing with OpenMP on the Raspberry Pi"
  - SIGSCE '19
  - First and second year undergraduates
  - Shared memory parallelism using OpenMP, a standardized interface

Marquette Systems Lab Environment

- Dedicated pool of Raspberry Pi boards are remotely available on demand

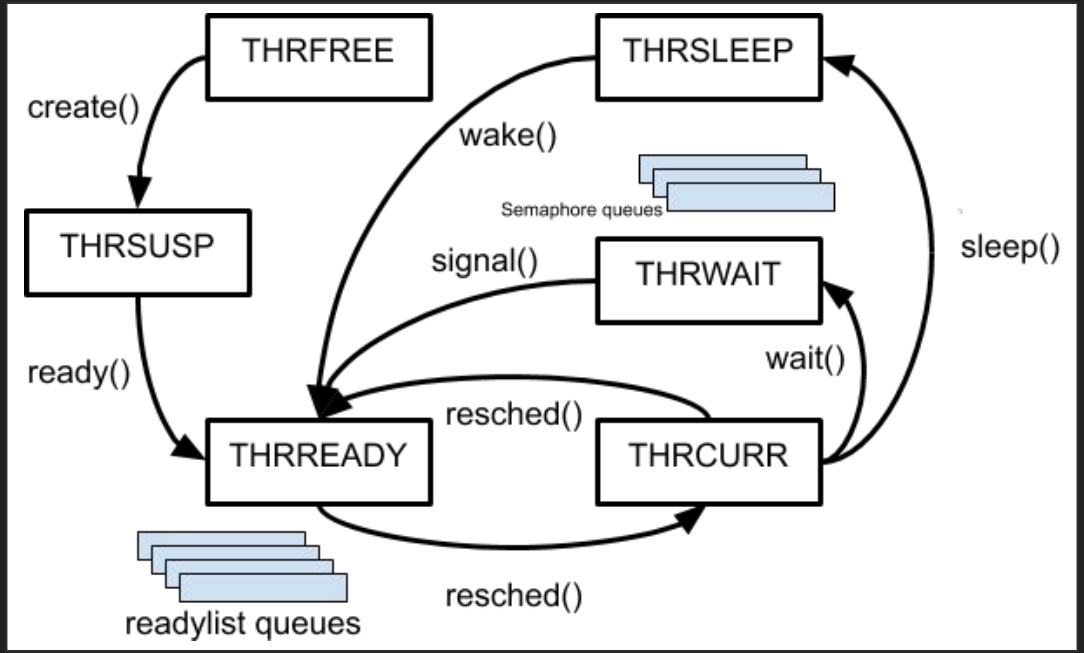


Difficulties

- Expanding Embedded Xinu to support a multicore platform is challenging
- Multicore features should be carefully implemented
  - Otherwise, Xinu's design goal of minimalism will be stepped on
  - Designed to be understood by by a mid-career undergraduate computing major
- Modern OSes do not face this tradeoff
  - Their solutions do not scale down to fit Xinu's design aesthetic
- Poor public documentation of modern processors compounds the difficulty
  - The general ARM Cortex A-53 document is available, but not for the BCM2837B0 SoC

Multicore Additions

- Priority-based preemptive process scheduler
  - 4 doubly-linked ready list priority queues



Multicore Additions (continued)

- Timer interrupts and preemption
  - Timer interrupts need to occur on each of the four cores
  - Therefore, each core must initialize its respective physical timer
    - Coprocessor registers hold information about interrupt control
- Atomic operations
  - Exclusive load and store operations provide synchronization among cores
    - `ldrex`, `strex`
    - `_atomic_increment()` , `_atomic_decrement()`
- Cache coherency and DMA
  - DMA buffer space is flagged as uncacheable to protect USB transfers from stale values

Marquette’s OS Course

- 80 undergraduate students in various majors:
  - Computer Science
  - Computer Engineering
  - Biocomputing
- Students work in small teams to build OS components
  - Weekly, cumulative assignments
  - 5 multicore assignments
- Stripped-down version of the kernel is given
  - Few hundred lines of C and ARM code

Multicore Synchronization Primitives

- Introductory multicore concurrency assignment
  - Students enforce basic protection among the four cores
  - Following 2 warm-up C assignments and the synchronous serial driver assignment
- Worst case: all cores are trying to access the UART at the same time
  - Students complete ARM routines to provide synchronization

```
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
```

Non-preemptive Multicore Scheduling

- Students add a thread abstraction onto each core by:
  - Building an assembly routine to switch process contexts
  - Modifying the incomplete create() function to consider multicore processes
  - Testing their implementations
- Students are required to examine the thread structure
  - Particularly, understanding:
    - The thread life cycle
    - A new field, `core_affinity`, that describes the core running the thread
    - For simplicity, we do not yet allow threads to migrate between cores once started
    - In addition, it is not possible to kill a running thread from a different core

Preemptive Multicore Scheduler

- Students implement round-robin priority scheduling:
  - Using 3 priority queues (representing low, medium, and high priority) per core
  - The ready list is now a two dimensional queue
- Students must understand:
  - Each core has its own timer (the underlying handlers are completed for them)
  - If aging is enabled, a thread may be promoted to a higher priority queue
    - This avoids starvation of a thread
- Testing:
  - Students can test prioritization in isolation and then introduce cases for aging

Multicore Semaphores

- Asynchronous, interrupt-driven UART driver
  - Challenge: semaphore variables should be safe from destructive updates by competing cores
- The high-level functions are similar to the synchronous driver, except:
  - The given asynchronous implementation is only functional on a single core architecture
- The student:
  - Develops the main functionality of the driver
  - Uses atomic increment and decrement as synchronization primitives to resolve the broken semaphores

Heap Memory on a Multicore Platform

- Focus switches to OS-level memory management
- A free list of available memory blocks can be accessed by any of the four cores at once
  - Race conditions must be protected against using spinlocks
- Students develop mutually-exclusive `malloc()` and `free()` memory operations

Outcomes

- Students implemented these projects in teams of two
- Quantitative comparison is difficult, primarily because:
  - Instructors avoid reuse of assignment variants
  - The order of project topics had to be shifted to accommodate multicore needs
    - Guarding the serial port is now necessary before writing the context switch
- Most well-received assignments:
  - Multicore synchronization primitives
  - Non-preemptive multicore scheduling
  - Core-safe memory allocation
- More challenging assignments:
  - Preemptive multicore scheduling (students' latent bugs were seemingly exacerbated)
  - Asynchronous device driver (complexity of managing interrupt-driven I/O alongside multicore)

Future Work

- Non-blocking concurrent data structures
  - Primary interest due to avoidance of deadlocks
  - Applied to the thread ready list queues and the memory free list
  - Probably of more value in a graduate-level course
- Memory protection and virtual memory
  - The virtual memory system is enabled on our port, but not for memory protection
  - Each thread can have its own memory space
  - With the foundational protections in place, concepts such as a user-space or process migration would become more simple to implement