ECE 531 – Advanced Operating Systems
Lecture 22

Vince Weaver

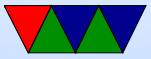
https://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

21 November 2023

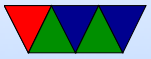
Announcements

- Project topics were due, I am responding
- HW#7 was posted, mostly short answer
- 2nd Midterm moved to November 30th



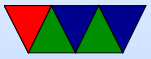
Brief Midterm Review

- Not cumulative, will be mostly topics since last midterm
- Virtual Memory
- Filesystems
- Graphics
- Multicore / Locking



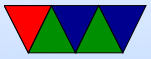
Project Notes – General

- Might be best to start small and gradually build to project
- Start with userspace code on Linux if possible before moving to our OS
- Might be easier to prototype some stuff on the earliest versions of the homework OS rather than the most recent that has more complex features



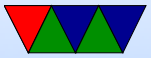
Project Notes – Device Drivers

- Ideally drivers for things like 1-wire / i2c / etc would live in the kernel
- To access from userspace you'd need to add a syscall
- In a perfect world you'd access via /dev but even the most recent HW doesn't support that yet
- Getting things working at all is most important though so it's fine to just provide userspace access

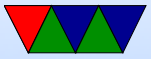


Project Notes – PS/2 Mouse

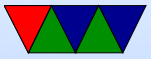
- Mouse is a bit more difficult than keyboard (keyboard will generally work without being written to first, a mouse you have to send a packet to activate)
- Low-level open-collector interface
  - Clock and Data lines pulled high by pull-up
  - To send 0, set GPIO pin to OUTPUT and output a 0
  - To send 1, *don't* output a 1. Instead switch the line to input which will allow the line to be pulled up to a 1



- Packet generally 11 bits, start/data/parity/stop
- Timing, ideally 10-20kHz or so. Hard to generate exact short time delays in kernel
- On real x86 hardware there was a separate micro-controller that ran the keyboard so the OS only had to talk to that, didn't have to bitbang the lines like this
- You can constantly poll for input, but that is wasteful
  - Instead you can set up an interrupt handler
  - Configure to trigger on falling edge (high to low) of GPIO23
  - This is IRQ49. You'll have to update IRQ handler to

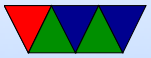


call your read routine when this triggers.



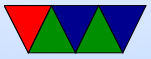
Multi-Processing

- In the old days your computer had a single CPU/core
- That was relatively simple to deal with
- Modern systems (even small embedded systems) have multiple cores



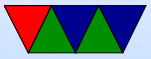
Hardware Concerns – Multi-Processing

- SMP/CMP (Symmetric or Chip Multi-processing): all cores are identical
- Asymmetric: cores can have different features (see ARM big.LITTLE or intel's efficiency cores)

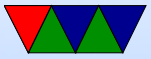


Hardware Concerns – Multi-Threading

- SMT (Simultaneous Multi-threading), Hyperthreading (Intel)
- Pipelined processor might not be able to fill all pipelines each cycle
- Add an extra instruction queue and have two programs issuing instructions to the pipelines
- Less transistors than extra core but usually not as much of a performance gain (can actually be worse!)
- OS often treats extra threads like extra processors for

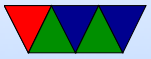


scheduling purposes



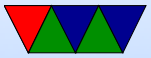
Hardware Concerns – Memory

- Shared memory vs Distributed
  Shared memory, a CPU can write a value to memory, read it back and it will be different (another CPU can write to it)
- How many copies of the OS? One per core or single image? One per core is more like a cluster.

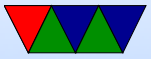


Hardware Concerns – NUMA

- In old days, single CPU with one single range of memory
- Modern CPUs, the memory controller (and DIMMs) might be run by separate packages
- This means some RAM is more distant from a core than others
- This leads to NUMA (non-uniform memory access), some RAM takes longer to access
- OS should take this in account when starting processes / scheduling jobs

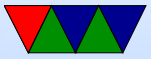


- UMA, NUMA, CC-NUMA (cache-coherent)



Multi-Processor Resource Sharing

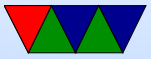
- How are resources shared in SMP system?
- Any core can access any of the devices. Need locking.



Multi-Processor Interrupts

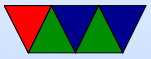
- Have one core handle all interrupts?
  Might have better cache behavior
- Round-robin interrupts to each core?
  Reduces load on core0 but hurts others.

- Balance interrupt load across processors?

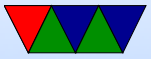


OS Support for SMP

- How can we have multiple cores share one OS-image?
- Big-kernel-lock
  - Simple, only one core can be inside kernel at time
  - Doesn't scale well, can end up with a lot of cores waiting for kernel to be available
- Split up with fine-grained critical sections.
  - Only parts of OS happen at once. Scheduler can run at same time as serial driver or filesystem read or page fault

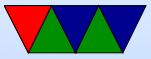


- Much harder to get right



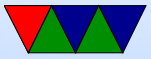
SMP Scheduling

- 4 processors, 5 jobs
  How to avoid ping-ponging? Better to make two processes slow or all of them?
- Gang scheduling – if you have processes that are using IPC (or multithreads) you want to schedule all at the same time so can communicate without having to wait through multiple context switches.
- Keeping jobs on same CPU started on (why is this good?) Cache behavior. TLB, NUMA.



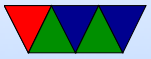
Why might you want to move them?

- When might you want to run everything on one core even though lots available? Power! Can put rest of CPUs to sleep.
- How do you online/offline hotswap processors.



Multiprocessor Scheduling

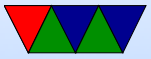
- Currently
  - Timer interrupt (or yield waiting for I/O comes in)
  - We scan the linked-list of processes seeing if any other process is ready to run
  - If it is, run it. If not, keep running current.
  - If no process is ready, run idle task
- Now
  - Timer interrupt: does the timer interrupt go to each core? Do you have separate timers? Do you have one



timer and it broadcasts an IPI to all cores?

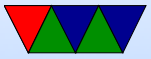
- Multiple cores inside scheduler at once. Is that an issue? Need locking.
- Each core looks at the list to see if anything ready to run.

- Issues
  - affinity – Ideally, a process stays on same core if at all possible.
    Maybe even have separate per-processor queue of jobs to run
  - smart scheduling – if a process has a spinlock held



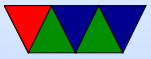
let it have a bit more time to clear so other processes aren't stuck on it

- space scheduling – a job needs say 8 threads, wait until 8 cores are available to run it
- gang scheduling – time and space scheduling if doing IPC with other processes, doesn't make sense to schedule the other side of these at different times



Helper Threads

- Linux has kernel threads (look in top for things starting with k or rcu).
- One of each type of thread per core
- Interrupt handlers have fast handler and worker threads.

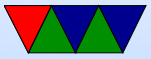


Initializing Multicore on Raspberry Pi

- Bare Metal
  - Detect which processor you are on

```
mrc p15, 0, r3, c0, c0, 5
ands    r3, #3                  /* CPU ID is Bits 0..1 */
bne wait_forever              /* If not CPU zero, go to sleep */
```

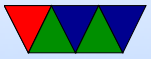
  - "park" the extra CPUs. Put in tight loop, wfe (wait for exception) when wake, check a flag to see if they should start and jump to address if true. Otherwise, back to sleeping.
  - To wake, use SEV to send event
- Raspberry Pi boot firmware does this for you



It copies some code to 0x0 and executes it before jumping to your code at 0x8000

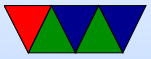
- This code parks the other cores
- each process has a mailbox, if you write an address there it will jump to it core 1: 0x4000009C core 2: 0x400000AC core 3: 0x400000BC
- They are waiting in WFE so have to send SEV too

- Other things you will need to do:
  - Set up stacks for each CPU (why can't they all share the CPU0 stack?)
  - Start up virtual memory and caches

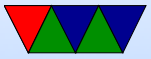


Locking depends on the caches working

- Start them into idle thread
- Start scheduling jobs?



Multicore Concerns

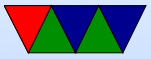


Race Conditions

- Shared variable access (increment memory_free after allocate?)
- Read-Modify-Write on ARM, value starts at 0

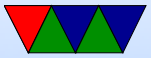
Core A	Core B
Read value from memory	
Increment value in reg	Read value from memory
Write back to memory	Increment value in reg
	Write back to memory

- What is the final result?
- What should the result be?



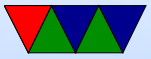
Critical Sections

- Want mutual exclusion, only one can access structure at once
  1. no two processes can be inside critical section at once
  2. no assumption can be made about speed of CPU
  3. no process not in critical section may block other processes
  4. no process should wait forever



How to avoid

- Single core: just disable interrupts
  - Bad for performance
  - OS might not let you do this as user (why?)
- Multicore: Locks/mutex/semaphore

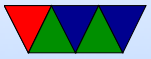


Mutex

- mutex_lock(&lock);
  - if unlocked (0), then it set lock=1 and return
  - if locked, return failure
  - what do we do if failed?

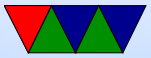
    Busy wait? (spinlock)

    re-schedule (yield)?
- mutex_unlock(&lock): sets lock to zero
- NOTE: we need special instructions for this (see later)



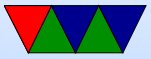
Semaphore

- Up/Down
- Wait in queue
- Blocking
- As lock frees, the job waiting is woken up



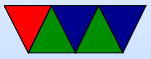
Locking Primitives

- Depend on Atomicity (what's that?)
- fetch and add (bus lock for multiple cores), xadd (x86)
- test and set (atomically test value and set to 1)
- test and test and set
- compare-and-swap
  - Atomic swap instruction SWP (ARM before v6, deprecated)
  - x86 CMPXCHG
  - Does both load and store in one instruction!



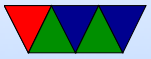
- Why bad? Longer interrupt latency (can't interrupt atomic op)
- Especially bad in multi-core

- load-link/store conditional
  - Load a value from memory
  - Later a store to same memory address.
  - Only succeeds if no other stores to that memory location in interim
  - ldrex/strex (ARMv6 and later)
- Transactional Memory



Locking Primitives

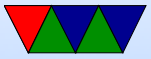
- can be shown to be equivalent
- how swap works:
  - lock is 0 (free). r1=1; swap r1,lock
  - now r1=0 (was free), lock=1 (in use)
  - lock is 1 (not-free). r1=1, swap r1,lock
  - now r1=1 (not-free), lock still==1 (in use)



ARMv7 Mutexes

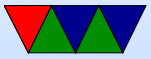
- On ARMv6 could use swap, but deprecated
- Now ldrex/strex
- Locking

```
.equ    MUTEX_UNLOCKED, 0
.equ    MUTEX_LOCKED,   1
.global mutex_lock
mutex_lock:
        ldr     r1, =MUTEX_LOCKED
lock_retry:
        ldrex   r2, [r0]
        cmp     r2, r1          @ are we already locked?
        wfeeq                   @ if so, go to sleep (wait for event)
        beq     lock_retry
        strex   r2, r1, [r0]    @ conditionally store value of r1 into r0
                                @ r2 lets you know if it worked or not
        cmp     r2, #1          @ if this failed
```



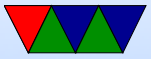
```
        beq     lock_retry      @ then keep retrying
        @ lock was acquired
        dmb                     @ Memory barrier
        bx      lr              @ return

.global mutex_unlock
mutex_unlock:
        ldr     r1, =MUTEX_UNLOCKED
        dmb                     @ memory barrier
        str     r1, [r0]        @ clear the lock
        dmb
        sev                     @ wake other processors waiting in wfe
                                @ by sending wakeup event
        bx      lr
```



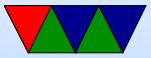
Memory Barriers

- Not a lock, but might be needed when doing locking
- Modern out-of-order processors can execute loads or stores out-of-order
- What happens a load or store bypasses a lock instruction?
- Processor Memory Ordering Models, not fun
- Technically on BCM2835 we need a memory barrier any time we switch between I/O blocks (i.e. from serial to GPIO, etc.) according to documentation, otherwise loads could return out of order



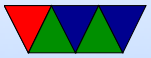
Deadlock

- Two processes both waiting for the other to finish, get stuck
- One possibility is a bad combination of locks, program gets stuck
- P1 takes Lock A. P2 takes Lock B. P1 then tries to take lock B and P2 tries to take Lock A.



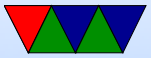
Livelock

- Processes change state, but still no forward progress.
- Two people trying to avoid each other in a hall.
- Can be harder to detect



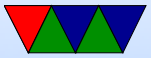
Starvation

- Not really a deadlock, but if there's a minor amount of unfairness in the locking mechanism one process might get "starved" (i.e. never get a chance to run) even though the other processes are properly taking and freeing the locks.



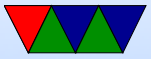
How to avoid Deadlock

- Don't write buggy code
- Reboot the system
- Kill off stuck processes
- Pre-emption (let one of the stuck processes run anyway)
- Rollback (checkpoint occasionally)



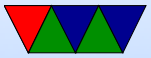
Priority Inversion

- Low-importance task interrupts a high-priority one
- Say you have a camera. Low-priority job takes lock to take picture.
- High-priority task wants to use the camera, spins waiting for it to be free. But since it is high-priority, the low priority task can never run to free the lock.



Locking in your OS

- When?
- Interrupts
- Multi-processor
- Pre-emptive kernel (used for lower latencies)
- Big-kernel lock? Fine-grained locking? Transactional memory?
- Semaphores? Mutexes
- Linux futexes?



Where does our OS need locks?

- Does a single-core operating system need locks?
  Yes, interrupts can cause similar troubles
- Any shared resources
- What if multiple processes try to write the console at the same time?
- What if try to update the memory allocation/free list at same time?