ECE 471 – Embedded Systems Lecture 19

Vince Weaver

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

vincent.weaver@maine.edu

20 October 2025

Announcements

- Keep thinking about project topics
- Don't forget HW#6 (due Friday)
- No class Wednesday – Engineering Career Fair
- Midterm not graded yet, will probably be a week
- Still working on HW#4 and HW#5 grades
- Seniors, don't forget advising for Spring 2026 classes! Will be sending out an e-mail about that

Real Time Operating System

- Can provide multi-tasking/context-switching
- Can provide priority-based scheduling
- Can provide low-overhead interrupts
- Can provide locking primitives

Hard Real Time Operating System

- Can it be hard real time?
- Is it just some switch you can throw? (No)
- Simple ones can be mathematically provable
- Note: that assumes perfect hardware, there are things like transistor metastability where you can't guarantee hardware timings
- Otherwise, it's a best effort

Priority Based, like Vxworks

- Each task has priority 0 (high) to 255 (low)
- When task launched, highest priority gets to run
- Other tasks only get to run when higher is finished or yields
- What if multiple of same priority? Then go round-robin or similar

Free RTOS

- Useful article series about this: `https://www.digikey.com/en/maker/projects/what-is-a-realtime-operating-system-rtos/28d8087f53844decafa5000d89608016`
- Footprint as low as 9K
- Pre-emptive or co-op multitasking
- Regularly scheduled tasks (vTaskDelayUntil(), i.e. blink LED every 10 ticks)
- Task priority
- Semaphores/Mutexes
- Timers

- Stack overflow protection
- Inter-process communication (queues, etc)
- Power management support
- Interrupts (interrupt priority)

Is Regular Linux a RTOS

- Not really
- Can do priorities ("nice") but the default ones are not RT.
- Aside, "nice" comes from old UNIX multi-user days, when you could be nice and give your long-running jobs a low-priority so they wouldn't interfere with other people doing interactive tasks

Is there an RT Version of Linux?

- For years there were outside patches
- You'd have to special patch and compile a kernel to get support
- With the upcoming 6.12 release all the patches will be merged and you can get better RT support
- It still might not be enabled by default on most distros

PREEMPT Kernel

- Linux PREEMPT_RT
- Faster response times
- Remove all unbounded latencies
- Change locks and interrupt threads to be pre-emptible
- Have been gradually merging changes upstream

Typical kernel, when can you pre-empt

- When user code running
- When a system call or interrupt happens
- When kernel blocks on mutex (lock) or voluntarily yields
- If a high priority task wants to run, and the kernel is running, it might be hundreds of milliseconds before you get to run
- Pre-empt patch makes it so almost any part of kernel can be stopped (pre-empted). Also moves interrupt routines into pre-emptible kernel threads.

Linux PREEMPT Kernel

- What latencies can you get?
  10-30us on some x86 machines
- Depends on firmware; SMI interrupts (secret system mode, can't be blocked, emulate USB, etc.)
  Slow hardware; CPU frequency scaling; nohz
- Special patches, recompile kernel

Linux Real Time Priorities

- Linux Nice: -20 to 19 (lowest), use nice command
- Real Time: 0 to 99 (highest)
- Appears in ps as 0 to 139?
- Can set with `chrt` command (see HW#6)

Co-operative real-time Linux

- Xenomai
- Linux run as side process, sort of like hypervisor

List of some other RTOSes

- Vxworks (the Martian)
- Neutrino
- Free RTOS
- Microsoft Windows / RTX64 extensions
- MongooseOS `https://lwn.net/Articles/733297/`
- ThreadX (in the Pi GPU)
  Bought by Microsoft. Azure RTOS / Eclipse ThreadX

Coding Tips when trying to Write Real Time Code

Challenges with Interrupts

- Why are interrupts slow?
- Shared lines, have to run all handlers
- On Cortex-A systems have one IRQ line, have to query all to see what caused it. Cortex-M improves this by having dedicated vector for each piece of hardware
- When can they not be pre-empted? IRQ disabled? If a driver really wanted to pause 1ms for hardware to be ready, would often turn off IRQ and spin rather than sleep

- Higher priority IRQs? FIR on ARM?
- Top Halves / Bottom Halves

Challenges with Threading

- A thread is a unit of executing code with its own program counter and own stack
- It's possible to have one program/process have multiple threads of execution, sharing the same memory space
- Why?
  - Traditionally, to let part of program keep running when another part waiting on I/O (gui keep drawing while waiting for input, sound playing in background during game, etc)

- Lets one program spread work across multiple cores

- This complicates the scheduler, and also makes priority more complex

Challenges with Locks

- When shared hardware/software and more than one thing might access at once
- Example:
  - thread 1 read temperature, write to temperature variable
  - thread 2 read temperature variable to write to display
  - each digit separate byte
  - Temperature was 79.9, but new is 80.0
  - Thread 1 writing this

- What if Thread 2 reads part-way through? Could you get 89.9?

- Is this only a SMP problem? What about interrupts?

More on Locks

- Previous was example of Race Condition (two threads "racing" to access same memory)
- How do you protect this? With a lock
  - Special data structure, allows only one thread inside the locked area at a time
  - This is called a "critical section"

```
lock(&temp_lock);           lock(&temp_lock);
write_display();            read_temperature();
unlock(&temp_lock);         unlock(&temp_lock);
```

Locking even on Single Core

- Can you have race conditions on a single core?
  - Yes, with interrupts
  - On simple systems you can just disable interrupts during critical section
  - Usually can't do that if have an OS

Lock Implementation

- Implemented with special instructions, in assembly language
- Usually you will use a library, like pthreads
- mutex/spinlock
- Atomicity

Memory Allocation in Embedded Systems

- Modern systems memory access is slow and complicated
- Caches add lots of jitter (can you disable them?)
- Virtual Memory also can cause unexpected delays (run on MMU-less system?)
- There are some things you can do in software to make things a bit better

Memory Allocation – Dynamic

- Using `malloc()/calloc()` or `new()`
- In C have to make sure you `free()` at end
- These aren't Linux syscall interfaces. The OS provides typically provides big chunks of RAM through the `brk()` and `mmap()` interfaces
- Some sort of library manages these bigger chunks, tracks usage and size, and tries to avoid fragmentation
- Usually on Linux you use the C library malloc() but you can also use custom libraries instead

Dynamic Memory Downsides

- What to do if fails?
  Can you handle that? What if error code also tries to alloc?
- Timing overhead? Is it deterministic?
  Especially problem with high-level languages and garbage collection
- Fragmentation: when there's plenty of RAM free but it's in small chunks when you need a large chunk

Can You use Static Allocation Instead?

- Allocate all memory you need at startup
- Fail early
- This isn't always possible, but avoids issues with failure, overhead, etc.
- Free RTOS (newer) allows static allocation at compile time

Linux Virtual Memory Issues

- Even if you statically allocate memory, on system with virtual memory it might swap out to disk
- This can suddenly make your code unexpectedly slower, ruin real-time performance
- Can you prevent this?
  - `mlockall()` syscall can lock memory so it stays in RAM, never goes to disk
  - So at start of program, allocate RAM, touch it (or prefault) to bring it in, then mlock() it

- Generally need to be root (why?)

Next up is SPI

Start early on it as there's more than one lecture of material