ECE 471 – Embedded Systems
Lecture 21

Vince Weaver

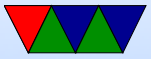
http://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

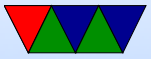
24 October 2022

Announcements

- Don't forget RT homework HW#6
- Don't forget projects
- HW#6 comment: newish assignment, the 3rd/4th experiment might slow down your system to be unusable if logged in at the GUI. If this happens and you can't complete the question, just mention that and complete things the best you can.
- Turn back midterm exam

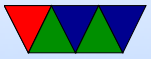


- Briefly go over midterm exam



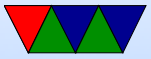
Scheduler Complications – Threading

- 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? Parallelize across more cores, have separate background task (sound in video games), etc.

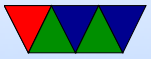


Scheduler Complications – Locking

- 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

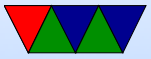


- What if Thread 2 reads part-way through? Could you get 89.9?

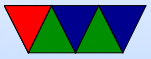


Scheduler Complications – Locking

- 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 access to piece of memory, others have to wait.
- Can this happen on single core? Yes, what about interrupts.
- Implemented with special instructions, in assembly language

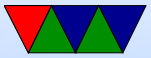


- Usually you will use a library, like pthreads
- mutex/spinlock
- Atomicity



Priority Inversion Example

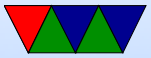
- Task priority 3 takes lock on some piece of hardware (camera for picture)
- Task 2 fires up and pre-empts task 3
- Task 1 fires up and pre-empts task 1, but it needs same HW as task 3. Waits for it. It will never get free. (camera for navigation?)
- Space probes have had issues due to this.



Real Time without an O/S

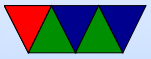
Often an event loop. All parts have to be written so deadlines can be met. This means all tasks must carefully be written to not take too long, this can be extra work if one of the tasks is low-priority/not important

```
main() {
    while(1) {
        do_task1(); // read sensor
        do_task2(); // react to sensor
        do_task3(); // update GUI (low priority)
    }
}
```



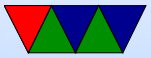
Bare Metal

- What if want priorities?
- Have GUI always run, have the other things happen in timer interrupt handler?
- What if you have multiple hardware all trying to use interrupts (network, serial port, etc)
- At some point it's easier to let an OS handle the hard stuff



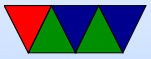
Real Time Operating System

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



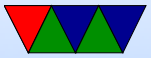
Hard Real Time Operating System

- Can it be hard real time?
- Is it just some switch you can throw? (No)
- Simple ones can be mathematically provable
- Otherwise, it's a best effort



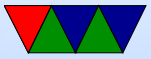
Priority Based, like Vxworks

- 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

- Footprint as low as 9K
- Pre-emptive or co-op multitasking
- Task priority
- Semaphores/Mutexes
- Timers
- Stack overflow protection
- Inter-process communication (queues, etc)
- Power management support
- Interrupts (interrupt priority)



Memory Allocation – Static vs Dynamic

- Using `malloc()` and `new()`
- Not always available
- Code can be large
- Is it thread safe?
- Is it deterministic?
- Can lead to fragmentation
- What to do if fails?
- Free RTOS (newer) allows static allocation at compile time