ECE 471 – Embedded Systems
Lecture 16

Vince Weaver

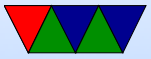
https://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

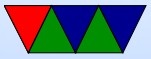
8 October 2025

Announcements

- Don't forget HW#5
- Project document was posted to website
- Midterm on the 17th, will review briefly Friday and also Wednesday
- Two weeks for HW#6 because of break and midterm
- For seniors in the class, advising for Spring 2026 coming up soon.
  Reminder that ECE574 Cluster Computing will not be taught this year

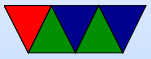


Brief Review of Jitter



Latency in Modern Systems

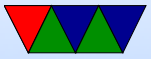
- Modern software stack has sources of latency



Video game keyboard latency example

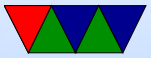
See Dan Luu's Paper "Computer Latency: 1977-2017"
`https://danluu.com/input-lag/`

- 1977 computers can have less latency to getting keypress on screen than fastest 2010s computers
- Having a fast processor only helps so much
- Slow hardware (keyboards, LCD displays), layers of abstraction in the way
- Apple II (1977) 30ms, modern machines 60-100+ms



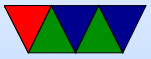
Latency of Apple II

- CPU running code reading memory access
  Each CPU instruction handful of 1MHz cycles (few usec)
- Keypress happens, high bit set along with ASCII code, CPU reads in
- CPU writes out ASCII value to memory
- Video generator hardware running in parallel at 60 fps
- Electron beam scanning, reads out RAM, runs through decode ROM to get 7-bit pattern, writes to screen within one frame worst case



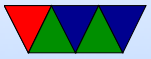
Latency of Modern System

- Press key, keyboard is own embedded system with CPU, scans keyboard, gets value, encodes it up as USB packet
- Sends out over USB bus (complex and with latency)
- USB controller gets packet, sends interrupt to CPU
- CPU gets interrupt, takes packet, notes it, returns from interrupt
- Later bottom half runs, decodes, to input subsystem,
- Operating system sees if anything is waiting for the input, if so it wakes it up (may take a bit if anything

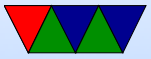


else running)

- If it's a GUI, might have to run and see which window has focus, etc
- Program itself finally gets notified of keypress. scanf(). Immediately printf()
- Terminal emulator, update the graphics for the window (colors, font processing)
- GUI compositor puts together screen, tells OS
- OS sends out over PCIe bus to GPU
- GPU runs shaders/whatever outputs to display via HDMI
- LCD display gets the data, decodes it to display it

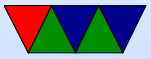


- Display might buffer a few frames to do extra processing (turn this off with “gaming” mode)



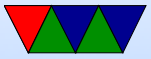
Can you get Real-Time on Modern Systems?

- Small embedded systems w/o operating system easier
- Some will have small PIO (programmable I/O), essentially smaller embedded system you can offload important tasks to (Beaglebone, Pi Pico, Pi5)
- Code directly to hardware
- Turn off interrupts
- Turn off/avoid caches/speculation
- Load all of code into memory



What about on higher end systems?

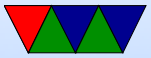
- Modern hardware does make it difficult with potentially unpredictable delay
- Hard to program such machines w/o an operating system
- Some machines provide special, deterministic co-processors to help (PRUs on the beaglebone)
- You can still attempt to get real-time by coding your OS carefully



Real Time Operating Systems

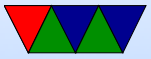
How do RTOSes differ from regular OSes?

- Low-latency of OS calls and interrupts (reduced jitter)
- Fast/Advanced Context switching (especially the scheduler used to pick which jobs to run)
- Often some sort of job priority mechanism that allows high-importance tasks to run first



Software Worst Case – IRQ overhead

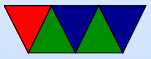
- OS like Linux will split interrupt handlers into top/bottom halfs
- Top half will do the bare minimum: ACK the interrupt, make a note for the OS to handle the rest later, then immediately return. Tries to keep IRQ latency as small as possible.
- Bottom half at some later time when nothing else is going on the OS will carry out the work needed by the IRQ (handle a keypress, or a network packet, etc)



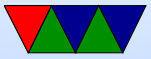
Software Worst Case – Context Switching

- OS provides the illusion of single-user system despite many processes running, by switching between them quickly.

- Switch rate in general 100Hz to 1000Hz, but can vary (and is configurable under Linux). Faster has high overhead but better responsiveness (guis, etc). Slower not good for interactive workloads but better for long-running batch jobs.



- You need to save register state. Can be slow, especially with lots of registers.
- When does context switch happen? Periodic timer interrupt. Certain syscalls (yield, sleep) when a process gives up its timeslice. When waiting on I/O
- Who decided who gets to run next? The scheduler.
- The scheduler is complex.
- Fair scheduling? If two users each have a process, who runs when? If one has 99 and one has 1, which runs



next?

- Linux scheduler was O(N). Then O(1). Now O(log N). Why not O(N^3)