# ECE 531 – Advanced Operating Systems
# Lecture 34

Vince Weaver

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

vincent.weaver@maine.edu

1 December 2025

## Announcements

- NOTE: Class will start 10-15 mins late on Wednesday the 3rd.
  I'll be giving a brief talk to the ECE100 class that meets the same time
- ALSO NOTE: there is no final exam for this class during finals week, only the second midterm
- HW#8 was posted (more on that in a bit)
- Tentative project schedule sent out

## Second Midterm

- Friday December 5th
- Cumulative, but heavily concentrating on topics since the 1st midterm
- Questions will be similar to those on homeworks.
- Topics:
  - Memory allocation
  - Virtual Memory: what it's good for, how two addresses can have the same address
  - Filesystems: Fat, ext4, others. Why use fat?

- ◦ Multi-core/Locking/Deadlock
- ◦ Maybe a brief graphics question

## Project Presentations

- Aim for 8 minute presentation with 2 min for questions/setup
- Can present with slides if you want
- Mostly just showing off your project idea and what you accomplished.
- Topics to include
  - Intro/Overview: what you did and why
  - Brief Related Work: any similar projects, how your project differs (it's OK if it doesn't)

- Hardware: describe any hardware used in the project If it's interfaced via built-in BCM2835 hardware briefly describe that interface too.
- Software: describe the software you wrote and how it ties into an operating system (is it a device driver? is it userspace and talks via syscalls? Does it bypass the OS? is it a proof-of-concept outside the OS?)
- Challenges: Describe any challenges you encountered
- Future Work: if you had more time, what else would you do
- Demo: show off what you did, if possible (it might

not be. Video, screenshots, or even just a description is fine)

## Project Writeup

- Described in the document
- Ideally IEEE format (this is grad course)
- Like a mini published paper. 6 pages or so is fine.
- Document on website describes which sections to include

## HW#8 Preview

- Finally getting graphics going
- You'll note it's slow. Memory copies are slow. Even with L1 dcache enabled
- Aside: how can you make fast memory copies?
- I provide code that sets up a 640x480 24-bit framebuffer using the mailbox interface

## HW#8 Preview – Gradient

- Draw vertical line, with RGB color component incrementing from 0 to 255
- You can use included vertical line code

## HW#8 Preview – Font Printing

- We have a 8x16 bitmap font. On a 640x480 screen this is 80x30 chars
- A call is added to the console driver to send characters to screen via HDMI in addition to serial port
- Right now when a character is drawn it is just a solid 8x16 block
- For homework modify so it prints the character
  - Need to take a row from the font which is a single byte. Look at this bit by bit to see which pixels are on

or off

- There are various ways to do this. Most common would be to use an 8 iteration for loop, and use bitwise AND to mask to see if bit set
- One complication is you want to draw most-significant-bit first
- A provided putpixel routine can be used.

## Kernel Race Conditions

- On a multi-core system, multiple cores can be running the OS at once
- This is fine unless two cores try modifying the same data structures at once as they can conflict and break things
- To avoid this we will need locking, which we will describe later
- A quick example where this is an issue:
  - two different parts of the kernel call `memory_allocate()`
  - Both of them call `find_free_memory()` at same time

- They both search the bitmap and find the same free block
- Now they update the bitmap to mark as not free, but it's too late, they both got the same one and any use of it is going to conflict
- This is called a race condition
- The solution is to *lock* this part of code so that only one core can be executing it at the same time

## SMP Locking Strategies

- Many OSes (like ours) weren't written with SMP in mind
- Instead they were retroactive retro-fitted
- How can an OS be made SMP safe?

## Big-kernel-lock

- Most initial conversions do this
- Simple, only one core can be inside kernel lock at time
- Doesn't scale well, can end up with a lot of cores waiting for kernel to be available
- Especially bad if more than 4 cores

## Fine-grained Locking

- Split up with fine-grained critical sections.
- Need to carefully figure out which code gets which lock
- Much harder to get right
- Can now have deadlock, where one code path takes lock1/lock2 and another lock2/lock1 and then they both get stuck forever waiting for the other to release

## OS Locking Strategies

- Linux initially was big-kernel lock, is now fine-grained
- TODO: lookup a few other OSes

## Scheduler – Single Core Reminder

- 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

## Scheduler – SMP Complications

- 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.

## SMP Scheduler Issues

- 4 processors, 5 jobs
  How to avoid ping-ponging? Better to make two processes slow or all of them?
- affinity – Ideally, a process stays on same core if at all possible.
  Cache behavior. TLB, NUMA.
  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 (or thread) you want to schedule all at the same time so can communicate without having to wait through multiple context switches.
- 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.

## 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?