ECE 531 – Advanced Operating Systems
Lecture 19

Vince Weaver

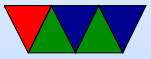
https://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

17 October 2025

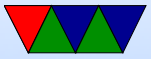
Announcements

- HW#6 will be posted next week
- Ask if planning on going to ECE career fair
  consensus is we should cancel class Wednesday
- Project was posted. Topics due 5 November
  the PDF handout has past project examples



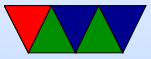
HW#6 Preview

- There are some major code changes building up to getting multitasking going
- Userspace is split off into its own directory
- This lecture is about what changed and also some of the concepts behind what's going on with the updated code



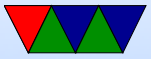
Processes

- Meant to have you do more work on the scheduling side. Getting A/B working is always a huge milestone in making your own OS
- Took me weeks the first time, thought I could simplify it down. But no, still a huge mass of assembly and had trouble sorting it out (lack of comments and code 2 years old!)
- Each process has a structure that holds info on it, plus the save state.



Process Control Blocks

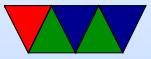
- Each process has some sort of process control block structure that holds all the info in a process
- There is a “processes” linked list that can be iterated by the OS to find processes
  - The scheduler will do this
  - The OS might have other reasons to look for a process struct (killing it, etc)



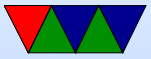
Process Control Structure

Here are the things you might find in a PCB, others are possible. This list is from vmwOS

- Saved State
  - Register saved state (for context switch, r0..r15, spsr)
  - Kernel saved state (maybe needed to restart process blocking in I/O)
- Linked list pointers (note, fancy kernels might use better data structs)
  - Next/Prev Process linked list pointers



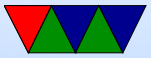
  - Waitqueue linked list pointer
- Process info
  - status (valid/running)
  - time accounting (user/kernel)
  - pid – process id
  - name – for printing
- Parent info
  - exit value, parent pointer – when program ends it needs to stick around until parent acknowledges it and gets the exit value
- Memory info – useful for virtual memory during page



faults

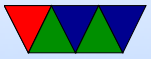
- stack pointer, size
- text pointer, size
- data pointer, size
- bss pointer, size

- Open file info
  - open_files array
  - current working directory

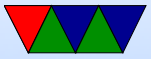


Open Files Array

- Each process tracks its open files
- Indexed via the filedescriptor
- This is UNIX/Linux so “everything is a file”
- Offset/inode/count/flags/name
- Also VFS struct with function pointers to the driver responsible for I/O
  - read()
  - write()
  - llseek()

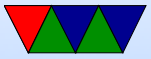


- getdents()
- ioctl()
- open()



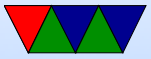
What does the Linux PCB look like?

TODO



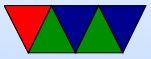
Userspace/Executables

- Entering into userspace for first time is a pain.
- Previous homework just called a function and treated as an exe, but that a bit of a hack.
- So had to implement executables (right now, bare code/data blob. A problem as working on HW#7 issue with BSS not actually being allocated so program crashes) Working on bFLT support.



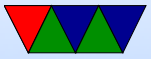
Filesystem

- With many executables, they need a place for them to live
- Set up a simple ramdisk (storage that looks like disk but lives in RAM)
- This will be loaded along with the kernel at boot (to save the trouble of having to write a disk or SD card driver yet)
- The filesystem we use is the "romfs" one from Linux (we'll learn more about filesystems in a few weeks)



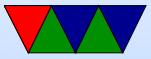
Fork/Exec

- How do executables start?
- We already discussed Unix fork/exec
- It turns out a true fork is a lot easier with virtual memory (we haven't learned about that yet)



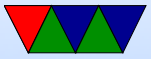
vfork()

- There's a stripped-down version of fork called `vfork()`
- As soon as you fork, the parent goes to sleep and the child is running inside the parent and the *only* thing the child it is allowed to do is either call exec() or _exit() (not even plain exit() as that would exit the parent



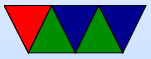
exec()

- We use execve() to launch the program
- you pass in the program you want to run, as well as the command line arguments
- It loads from disk the executable, allocates memory, sets up the process, marks as ready to run.



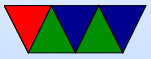
Scheduler

- How does the scheduler work?
- Simple, nothing fancy. There's a doubly linked-list of all processes and when a timer interrupt happens the list is walked to find the next one that's runnable.
- What if none available? Then run the idle thread.



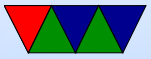
Idle Thread

- Instead of busy-waiting when nothing wants to run, run idle thread
- Process 0 on Linux/UNIX
- Can do background tasks
- Also can just go to low-power mode (wfi on ARM, hlt on x86)
- We need to create and launch this as a kernel thread though and have the scheduler run it when appropriate



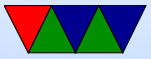
Waitqueues

- Also implements wait queues. If you are sleeping (because of a vfork) or waiting on I/O (waiting for keypress) you get put to sleep and put on a linked-list waitqueue. Then when I/O comes in, you are woken up, removed from the queue, and marked as ready.
- This is tricky as in theory you are sort of sleeping in the kernel and that's how we implement it, so we need to save our kernel register state as well as the user space. There's probably better ways to do this.



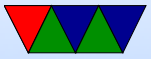
Wait Queue

- There is often separate waitqueue linked lists that can hold everything waiting on a certain kind of I/O
- The process itself doesn't move, it just is added/removed from these lists as needed
- Often waitqueue is per device, so you'll have a serial one, a disk one, a network one, etc

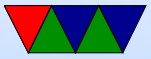


Wait Queue Example – Console Code (vmwos)

- your code `read(stdin,buffer,size)`
- syscall looks up file descriptor, sees fd maps to console
- calls `console_read()`
- console driver has a buffer that gets filled in the background by serial port data
  - if more data avail then requested, return that many bytes
  - if less data avail then requested, return all available

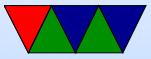


  - if no data available, put on waitqueue for console data
- next time serial port sends more data to console, it will put in buffer, but also wake up everyone in waitqueue
- that marks each process is READY and then removes all from queue

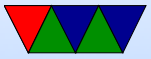


Linux Waitqueues

- `https://lwn.net/Articles/577370/`
- Linux had simple waitqueue implementation
- Mindcraft Linux performance issues vs other oses in late 90s
- "Thundering Herd" problem, you wake up all sleeping processes on waitqueue, they all wake and try to take action, but only enough input for one and have to put to sleep again
- They added a lot of code to avoid this

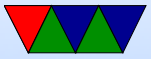


- Turns out they overdid it, and so were looking to back out and go simple again and only have complex in cases that need it.

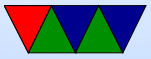


Waitpid

- In UNIX like operating systems once you have children via fork, if they die they don't go away. zombies. You can wait using waitpid() to see when they die, and once you use waitpid they are finally freed.
- So how do you wait in the background like in the HW? Had to implement waitpid(NOHANG) which means check to see if any children have died. If not, continue. Otherwise handle them so they can die.
- So in the shell after every command is typed it does a

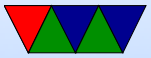


waitpid(NOHANG) to see if any of the background tasks finished.



Final HW#6 Notes

- Might be delayed posting, issues with previous years code
- Will involve getting multitasking running
- Will involve looking at scheduler
- Will involve modifying the kernel memory allocator



Virtual Memory

See next lecture for virtual memory notes