ECE 531 – Advanced Operating Systems
Lecture 15

Vince Weaver

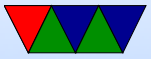
https://www.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

6 October 2025

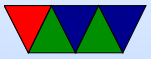
Announcements

- Note: Midterm after Fall break, on 20th
- Homework #5 will be Posted



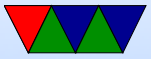
HW#5 – Userspace / Syscalls

- In this homework we will move our shell to userspace
- This will involve switching to user mode as well as implementing system calls
- All calls in the shell to internal kernel routines must be converted to userspace routines, such as printk() to printf() and uart_putc() to putchar()



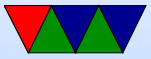
HW#5 – Switching to Userspace

- Need to setup user stack (can do this by switching to special system mode, which is like user but privileged)
- Need to set saved SPSR register to have usermode indicated
- Need to set return link register (LR) value to point to our shell() function
- Then we just return which starts the shell in usermode



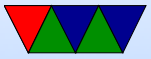
HW#5 – Syscalls

- All I/O from usermode must be done through syscalls
- I provide a syscall handler that does a few common ones
- You will add the `time()` syscall that we will use to report seconds since boot
- Will need to modify timer interrupt handler to track time
- In our shell need to call this. In real life libc (C library) provides syscall wrappers using inline assembly. We don't have that yet so for now will call with something like `syscall1(SYSCALL\_TIME,(long)&time);`

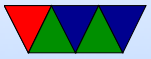


HW#5 Shell Notes

- Review C string handling, strcmp/strncmp and strcpy/strncpy/strlcpy
- Be careful with the `sizeof()` operator, note to get length of string use `strlen()` not `sizeof()`. `sizeof(char[128])` will be 128 `sizeof(char *)` will be 4, possibly neither is what you want if you are measuring a NUL terminated string.
- In this class we haven't talked much about software engineering best practices

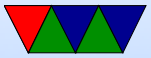


- Unit tests for printf/printk. Test outside of our OS by making it standalone unit tests (avoid SD card swapping)
- Code commenting.
- Source code versioning (git).



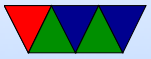
Multi-tasking / Multi-programming

- You could design a computer to only run one thing at a time
- Much more convenient if you can run multiple programs
  - If one program stops to wait for I/O, another can run
  - You can have multiple tasks going, but given the illusion they are all running at once
  - Especially useful when only have one core to run on



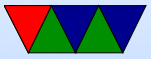
Timer Interrupt

- Usually a timer interrupt is set up to trigger every so often
- 250Hz on Linux
- If more than one process wants to run, the old one is stopped and its context is saved, and a new one is brought in to replace it



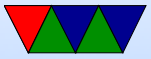
Context switching

- First time you get it working you get excited about having an AAA program and BBB programing printing ABABABA



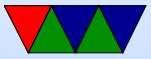
How is Context Switching Implemented?

- Interrupt comes in
- Scheduler (more on that later) decides if it's time to switch to another program

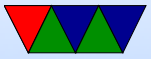


Context Switch – Save Current Program State

- TODO: diagram of Process Control Block
- Userspace registers saved to process control block
  Note this is tricky as we're running in kernel/interrupt mode
- Need to save current PC of program
- Also save SPSR (flags)
- Stats (like time running) can be updated
- There's generally no need on modern systems to copy

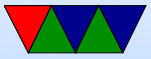


any memory around

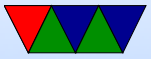


Context Switch – Restore New Program State

- Copy all register state from the new process process control block into user registers (this includes stack pointer as well as the return address to where the code was stopped)
- Also restore SPSR status register (with flags)
- On virtual memory systems update pointer to virtual memory page tables
- Return from timer interrupt and instead of returning to

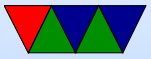


original we end up in the new one

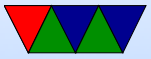


Hardware vs Software Context Switching

- Some processors (like x86) can do context switching in hardware
- You have special pointer to process control block
- Originally it was faster, but doesn't handle a lot of odd cases. Linux moved away from using it.
- It makes x86 OS code more complex though because there are some features that depend on some of the infrastructure
- TODO: look up more on this, things like TSS, LDT,

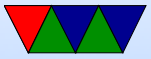


descriptors, ioperm, etc

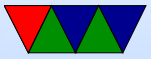


Context Switch Code

```
int32_t process_switch(struct process_control_block_type *old,
                       struct process_control_block_type *new) {

/* Save current state to PCB */
asm("mov␣␣␣␣r2,␣%[save]\n"
    "stmia␣␣r2,{r0-lr}\n"   // Save all registers r0-lr
    "add␣␣␣␣r2,r2,#60\n"
    "mrs␣␣␣␣r0,␣SPSR\n"     // load user SPSR
    "stmia␣␣r2,{r0}\n"      // save SPSR to PCB
    : /* output */
    : [save] "r"(&(old->kernel_state.r[0]))/* input */
    : /* clobbers */
);

current_process=new;

/* Restore current state from PCB */
asm("mov␣␣␣␣r2,␣%[restore]\n"
    "ldr␣␣␣␣r0,[r2,#60]\n"
    "msr␣␣␣␣SPSR,␣r0\n"     // restore SPSR (switch to user mode)
    "ldmia␣␣r2,{r0-r14}\n"  // restore registers
```

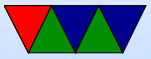


```
    "mov␣␣␣␣pc,lr\n"          // return, restoring SPSR
    : /* output */            //
    : [restore] "r"(&(new->kernel_state.r[0]))
    /* input */
    : /* clobbers */
);
```



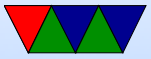
The Scheduler

- If you have multiple processes ready to run, how do you pick which to run next?
- The code that does this is called the scheduler
- This is a complex problem
- You want to run as fast as possible as it runs on every timer tick



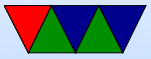
When to Schedule

- If timeslice runs out
- Task voluntarily yields (it has run out of work to do)
- If kernel blocks on I/O (Can be but to sleep instead of busy waiting)



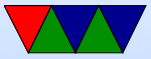
Can you Multi-task without timer tick?

- Co-operative multi-tasking
- Really old MacOS and Windows
- Processes voluntarily yield every so often
- Can this go wrong? What if process doesn't want to give up?



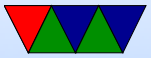
Scheduling Goals

- All: fairness, balance
- Batch: throughput (max jobs/hour), turnaround (time from submission to completion), CPU utilization (want it busy)
- Interactive: fast response, doesn't annoy users
- Real-time: meet deadlines, determinism



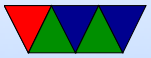
Batch Scheduling

- First-come-first-served (what if 2-day long job submitted first)
- Shortest job first
- Many others



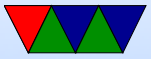
Interactive Scheduling

- Round-robin
- Priority – “nice” on UNIX
- Multiple Queues
- Others (shortest process, guaranteed, lottery)
- Fair scheduling – per user rather than per process



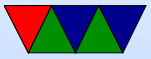
Real-time Scheduling

- Complex, more examples in 471 or real time OS course



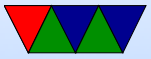
Scheduler example

- Simple: In order the jobs arrive
- Static: (RMS) Rate Monotonic Scheduling – shortest first
- Dynamic: (EDF) Earliest deadline first
- Three tasks come in
  - A: deadline: finish by 10s, takes 4s to run
  - B: deadline: finish by 3s, takes 2s to run
  - C: deadline: finish by 5s, takes 1s to run
- Can they meet the deadline?



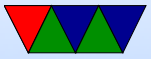
- There is a large body of work on scheduling algorithms.

In-order	A	A	A	A	B	B	C	-	-	-
RMS	C	B	B	A	A	A	A	-	-	-
EDF	B	B	C	A	A	A	A	-	-	-

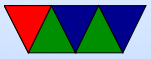


Priority Based Scheduling

- It's actually rare for an OS to let you specify a deadline
- Usually instead they are priority based
  - Have multiple tasks running, assign priority
  - In previous example, B highest, then C, then A
  - B can pre-empt C and A
- What can happen if overcommit resources? Starvation

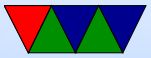


IRQ	-	-	-	-	-	I	-	-	-	-
HIGH	-	-	-	-	B	-	B	-	-	-
MEDIUM	-	-	C	-	-	-	-	-	-	-
LOW	A	A	-	A	-	-	-	A	-	-
OS	!	!	!	!	!	!	!	!	!	!



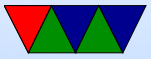
Priority Inversion Example

- 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.



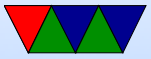
Linux Real Time Priorities

- 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



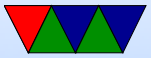
Scheduling Queues

- generally there will be a queue data structure holding all processes ready to run
- There will also be wait queues, where programs waiting on I/O can sleep
- If I/O comes in, the kernel will wake the process by moving it to the ready-to-run queue so it can be scheduled



Process States

- Running – on CPU
- Ready – ready but no CPU available
- Blocked – waiting on I/O or resource
- Terminated – might stick around until parent acknowledges



Scheduler Inputs

- How do you schedule?
- Per-task (5 jobs, each get 20%).
- Per user? (5 users, each get 20%).
- Per-process? Per-thread?
- Multi-processors? Hyper-threading? Heterogeneous cores?
- Power / Thermal issues? Performance counters?