The Context Switch

Dave Eckhardt
de0u@andrew.cmu.edu

Outline

- Project 2 Q&A
- Context switch
  - Motivated by yield()

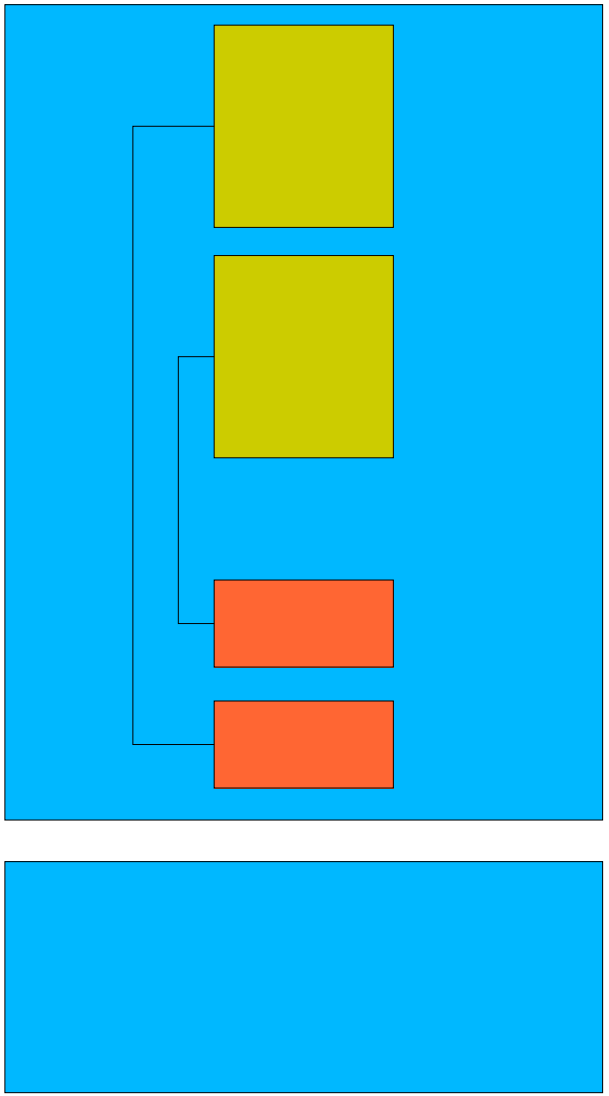
Mysterious yield()

- P1
  - while (1)
    - yield(P2)
- P2
  - while(1)
    - yield(P1)

User-space Yield

- Consider pure user-space threads
  - The opposite of Project 2
- yield(user-thread-3)
  - save registers on stack
  - /* magic happens here */
  - restore registers from stack
  - Return

Memory Picture



No magic!

- yield(user-thread-3)
  - save registers on stack
  - threadblock->pc = &there;
  - threadblock=findtcb(user-thread-3);
  - stackpointer = threadblock->sp;
  - jump(threadblock->pc); /* e.g., asm(...) */
  - there:
    - restore registers from stack
    - return
- What values does the program counter have?

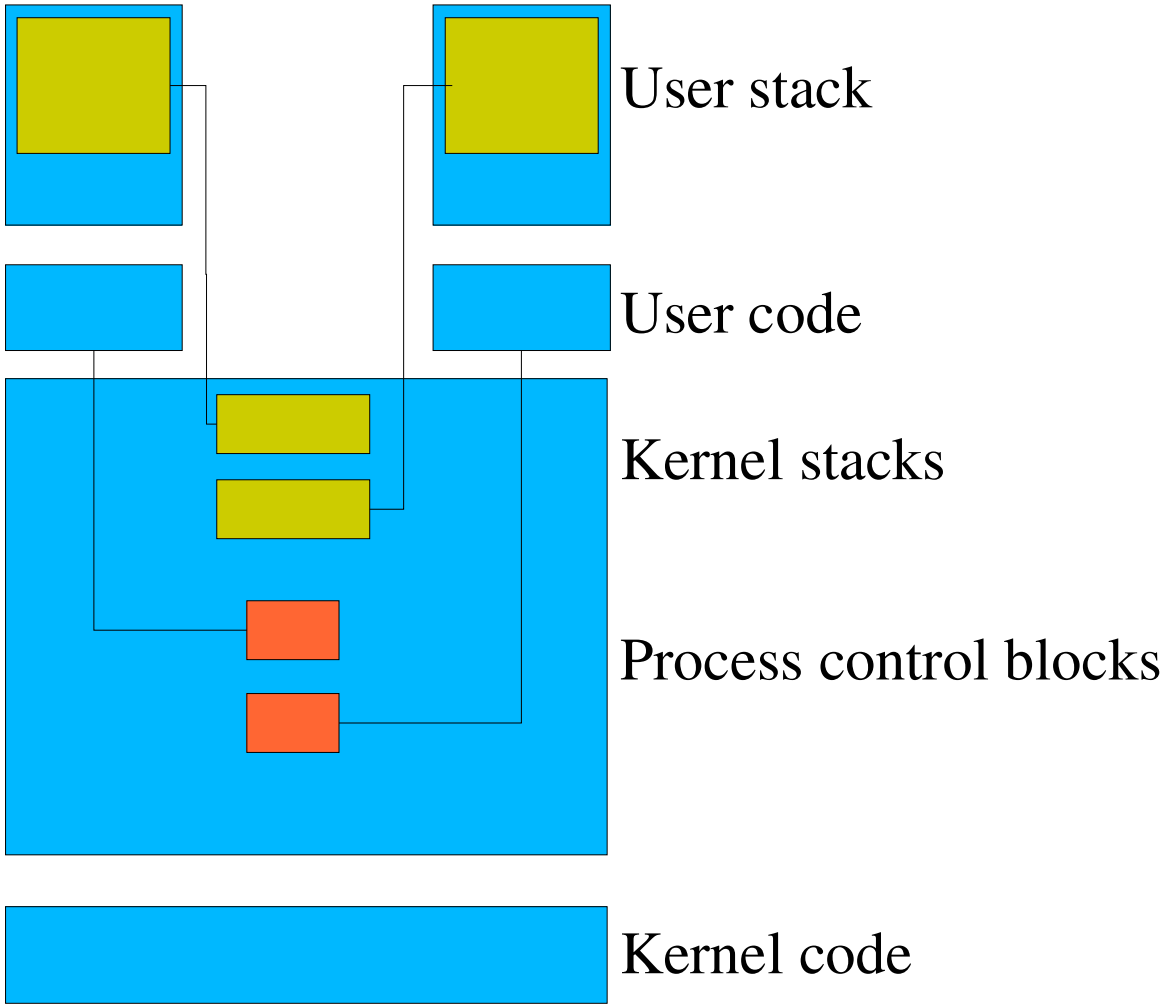
Remove unnecessary work...

- yield(user-thread-3)
  - save registers on stack
  - threadblock=findtcb(user-thread-3);
  - stackpointer = threadblock->sp;
  - restore registers from stack
  - return

User vs. Kernel

- Kernel process vs. user-space thread
  - User-space threads: shared memory
  - Separate kernel processes: indepdendent memory
- Kernel context switches aren't just yield()
  - Message passing from P1 to P2
  - P1 sleeping on disk I/O, so run P2
  - CPU preemption by clock interrupt

Kernel Memory Picture



Yield steps

- P1 calls yield(P2)
- INT 40 -> Boom!
- Processor trap protocol
  - Saves registers on P1's kernel stack
  - Activates kernel virtual memory
  - Loads new registers
  - Starts trap handler

Yield steps

- P1 (in kernel) calls yield(P2)
- yield()
  - return(process_switch(P2))
- P1 trap handler done
- Processor return-from-trap protocol
  - Restores registers from P1's kernel stack
  - Adjusts virtual memory
- INT 40 instruction “completes”
  - Back in user-space
- P1 yield() routine returns

That's not right!

- What about process_switch()?
  - **ATOMICALLY**
    - enqueue_tail(runqueue, cur_pcb);
    - cur_pcb = dequeue(runqueue, P2);
    - save registers (on P1's kernel stack)
    - Stackpointer = cur_pcb->sp;
    - restore registers (from P2's kernel stack)
    - return
- Process_switch() “takes a while to return”
  - When P1 calls it, it “returns to” P2
  - When P2 calls it, it “returns to” P1 – eventually

Clock interrupts

- P1 doesn't “ask for” clock interrupt
  - Clock handler *forces* P1 into kernel
    - Like an “involuntary system call”
    - Looks same way to debugger
- P1 doesn't say who to yield to
  - Scheduler chooses next process

I/O completion

- P1 calls read()
- In kernel
  - read() starts disk read
  - read() calls condition_wait(&buffer);
  - condition_wait() calls process_switch()
- While P2 is running
  - Disk completes read, interrupts P2 into kernel
  - Interrupt handler calls condition_signal(&buffer);
  - condition_signal() MAY call process_switch()
    - P1, P2, P3... will “return” from process_switch()

Summary

- Similar steps for user space, kernel space
- Primary differences
  - Kernel has open-ended competitive scheduler
  - Kernel more interrupt-driven
- Implications for 412 projects
  - P2: understand thread_create() stack setup
  - P3: understand kernel context switch