Slide 1

Administrivia

- Reminder: Reading quiz due today. 11:59pm.
- Next reading quiz coming soon, and/or also homework.

Slide 2

Processes and Virtualizing the CPU — Recap

- A key thing we want from an operating system — allow multiple things to be happening "at the same time" (really or in effect).
- Define in terms of "process" abstraction.
- Implement by virtualizing the CPU. If you keep in mind the big picture, all the details should make sense?

Slide 3

Implementing Processes, Continued

- In the big picture, each process consists of a program (read in from disk) and a machine state. How to execute the program?

  We want execution to be efficient, *but* we also want it to be "safe".
- "Safe"? a couple of very old stories ...

Slide 4

Sidebar: Two Very Old War Stories

- First story ("how I discovered the difference between DOS and a real operating system'):

  I started out on mainframes and other multiuser systems and was not an early adopter of PCs. During that time an employer got me my first PC. I was taking courses part-time and learning Pascal. One of my homework programs failed in strange and puzzling ways ...

  (Details in lecture.)
- Second story (no catchy name but similar idea):

  I'd had access to UNIX desktop systems but not to Windows. In the process of learning my way around Windows (and the whole "WIMP" paradigm) I did something that also produced very bad results ...

  (Details in lecture.)

Slide 5

Running Programs, Continued

- For efficiency, best thing would seem to be "direct execution" (as opposed to, say, emulation).
- But that's potentially unsafe — hence the term "limited direct execution". Requires some support from hardware.

Slide 6

Hardware — Dual Mode Operation

- *In hardware*:

  Distinguish between "kernel mode" and "user mode". Designate some instructions as "in kernel mode only".
- Attempt to execute kernel-mode-only instruction in user mode is an error and usually crashes the program.
- (Connecting to CSCI 2321: Could implement this using a bit in a special-purpose register, which kernel-mode-only instructions check.)

Slide 7

A Dilemma

- But there are things you want user programs to do — e.g., create files — that require kernel-mode-only instructions.
- How to make this possible? “system calls”.
- (Textbook’s discussion of this topic a bit x86-centric, unfortunately. I’ll try to discuss more generally here.)

Slide 8

System Calls — Mechanism

- Library routine (running in user mode) sets up parameters and issues `TRAP` instruction or equivalent. A key parameter says which system call is being made (to create a process, open a file, etc.).
- `TRAP` instruction switches to kernel mode and transfers control to a fixed address.
- At that address is code for “handler” that uses parameters set up by library routine to figure out which system call is being invoked and call appropriate code.
- When processing of system call is finished, control returns to calling program — *if* appropriate. (What are other possibilities? Consider situations involving waiting, errors.) Return to calling program also switches back to user mode.

Slide 9

Example: System Calls in MIPS

- MIPS instruction set includes `syscall` instruction that generates a system-call exception. MIPS interrupts/exceptions use special-purpose registers to hold type of exception and address of instruction causing exception.

  Before issuing `syscall`, program puts value indicating which service it wants in register `$v0`. Parameters for system call are in other registers (can be different ones for different calls).

- Interrupt handler for system calls looks at `$v0` to figure out what service is requested, other registers for other parameters.
- When done, it uses `rfe` instruction to restore calling program's environment, then returns to caller using value from EPC register.

Slide 10

Example: System Calls in MIPS/SPIM

- SPIM simulator — a primitive O/S! — defines a short list of system calls. Example code fragment:

```
la $a0, hello
li $v0, 4 # "print string" syscall
syscall
....
.data
hello: .asciiz "hello, world!\n";
```

Slide 11

System Calls — Services Provided

- Typical services provided include creating processes, creating files and directories, etc., etc. — details depend on (and in some ways define, from application programmer's perspective) operating system.
- Examples from last year's textbook:
  - POSIX (Portable Operating System Interface (for UNIX)) — about 100 calls.
  - Win32 API (Windows 32-bit Application Program Interface) — thousands of calls.

  Worth noting that the actual number of system calls is likely smaller — interface may contain function calls that are implemented completely in user space (no `TRAP` to kernel space).

Slide 12

Time Sharing

- Going back to big picture, remember that we want to share actual processors among processes, and the mechanism for doing that is "time sharing".
- To make this work, have to periodically stop running one process and run another. When to do that?
- Simple way is just to run until interrupted — because running process has to wait (e.g., for I/O) or terminates, or in response to an external interrupt.
- This works fine for batch systems, but interactive systems — what if running program doesn't do any of those? In "cooperative multitasking" can add a system call "yield", but — well, problem is obvious, no?
- Again we need help from hardware . . .

Timer Interrupts

- Idea here is to set a timer that will generate an interrupt after some specified amount of time.
- Before starting a user program, operating system sets the timer.

Slide 13

Sidebar: Interrupts?

- Many situations in which it's useful or necessary to stop current program and do something else, such as:
  - Running program ends normally.
  - An error occurs.
  - Something outside the CPU (e.g., an I/O device) signals it.
  - A program makes a system call.
- All processed similarly as "interrupts". Common goal is to stop what we're doing, go attend to the interrupt ("interrupt handler"), then (maybe) pick up where we left off.
- On some systems, single interrupt handler; one others, different handlers for different kinds of interrupts.

Slide 14

Slide 15

Interrupts, Continued

- Hardware and interrupt-handler code must between them make it possible to "pick up where we left off". So they need to:
- Save the current program counter.
- Save other machine state, such as contents of registers.

Slide 16

Context Switches

- Basic idea: Stop what we're doing and switch to something else.
- Similar to what happens in interrupt handler: Save current program counter and other machine state. Then load new program counter and state from previously-saved values.
- In effect, switch "execution context".

Slide 17

Scheduling

- When a running process blocks or ends, and perhaps after handling an interrupt, want to switch to another process. Often more than one choice.
- Who chooses? “Scheduler” — and many ways to do it. Next topic . . .

Slide 18

Minute Essay

- I’ve missed kind of a lot of classes. I was going to try to record some extra make-up lectures, but I’m not sure that makes sense — might be better to just move on. Thoughts?