Slide 1

Administrivia

- (None.)

Slide 2

Minute Essay From Last Lecture

- Quite a bit of variation in answers, many somewhat unclear(?). Key point is whether the o/s can defend itself.
- To recap . . .

Slide 3

O/S Versus Application Programs — Recap/Review

- Should seem reasonable to make distinction between what O/S can do and what application programs can do.
- But how to enforce that? i.e., how to make it as difficult as possible for buggy or malicious application programs to do what they shouldn't?

  Can this problem be solved completely by clever programming? Consider that most current systems can be asked to load and execute machine-level application code . . .

Slide 4

O/S Versus Application Programs, Continued

- If you don't allow that — how do you decide what's okay?
- If you do allow loading and executing arbitrary code, then some sort of hardware mechanism for limiting what it can do seems like the only way. This is the problem "dual-mode operation" is intended to solve.

Slide 5

O/S Versus Application Programs, Continued

- At hardware level, then, need to keep track of which mode we're in and use that information to allow/disallow certain operations (and maybe memory accesses — though that could be a separate problem/solution).
- To do this efficiently — single bit in a register somewhere, probably a special-purpose one, checked by "privileged" instructions.
- What happens if unprivileged program tries . . . ? Hardware version of exception — interrupt.
- How to set this bit? privileged operation, or no?

Slide 6

O/S Versus Application Programs, Continued

- A solution: Include instruction to generate interrupt, and have hardware, on interrupt, transfer control to a fixed location *and* set the "privileged" bit. If what's at the fixed location is O/S code, then it can do more checking (e.g., passwords).
- What if it's not O/S code?

Slide 7

O/S Versus Application Programs, Continued

- So maybe we need memory protection too? but we probably needed that anyway.
- How to make memory protection work? more about that later, but for now — again, seems like the only way to do this reliably and efficiently is with help from hardware.

Slide 8

System Call / Interrupt Processing — Recap/Review

- Recall(?) typical mechanism for regular program calls: Put parameters in agreed-on locations (registers, stack, etc.), issue instruction that saves current program counter (in another register maybe) and transfers control to called program. Called program returns using saved program counter.
- System calls are similar *except* that the "called program" is at a fixed address *and* the transfer of control also puts the processor in supervisor/kernel mode.

Slide 9

Command Shells

- History — early batch systems had to interpret "control cards"; modern equivalent is to interpret "commands" (usually interactive).
- Not technically part of o/s, but important and related.
- Typical shell functionality:
  - Invocation of programs (optionally in background).
  - Input/output redirection.
  - Program-to-program connections (pipes).
  - "Wildcard" capability.
  - Scripting capability.
- Examples — MS-DOS `command.com`; UNIX `sh`, `bash`, `csh`, `tcsh`, `ksh`, `zsh`, ...

Slide 10

Homework 1 Programming Problem

- The idea is to write a very simple shell based on the sort-of-pseudocode in the textbook, using `fork` and `execve` system calls.
- To do this, you have to solve a couple of problems:
  - Figure out how to use system-call library functions `fork` and `execve`. Overview on next slide; details in `man` pages.
  - Deal with string processing in C (or C++). Some hints in the homework writeup. Remember that C doesn't protect you from "buffer overflows" (e.g., there's a reason `gcc` complains about `gets`).

Slide 11

Homework 1 Programming Problem, Continued

- `fork()` function creates and starts a new process. Both original ("parent") and new ("child") processes execute the same program, continuing at whatever follows call to `fork()`. Return value from function says which process is which.
- `execve()` function discards current program and loads and starts a new one. If it fails, execution continues with whatever follows; otherwise whatever follows is ignored!

Slide 12

Minute Essay

- None — sign in.