Slide 1

Administrivia

- Homework 1 to be on Web later today. Due in a week.

Slide 2

System Calls

- Recall that some things can/should only be done by o/s (e.g., I/O), and hardware can help enforce that.
- But application programs need to be able to request these services. How can we make this work? System calls . . .

Slide 3

System Calls — Mechanism

- Library routine (running in user mode) sets up parameters and issues TRAP instruction or similar. 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 4

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 discussed in 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 5

Interrupts

- Processing of TRAP instructions is similar to interrupts, so worth mentioning here:
- Very useful to have a way to interrupt current processing when an unexpected or don't-know-when event happens — error occurs (e.g., invalid operation), I/O operation completes.
- On interrupt, goal is to save enough of current state to allow us to restart current activity later:
  - Save old value of program counter.
  - Disable interrupts.
  - Transfer control to fixed location ("interrupt handler" or "interrupt vector") — normally o/s code that saves other registers, re-enables interrupts, decides what to do next, etc.

Slide 6

Operating System Structures

- Clearly o/s could involve a whole lot of code (e.g., second edition of textbook says 29M lines of code for Windows 2000). How to structure?
- Choices include:
  - Monolithic systems.
  - Layered systems.
  - Microkernels.
  - Client-server model.
  - Virtual machines.
  - Exokernels.

Slide 7

Monolithic Systems

- Tanenbaum's description in the previous edition of the textbook — "The Big Mess". Maybe an exaggeration, since there can be *some* structure.
- Examples include MS-DOS, early UNIX.
- Arguments for this approach — "works, sort of"?
- Arguments against — easier for one malfunctioning component to crash others.

Slide 8

Layered Systems

- Idea — use layers of abstraction, just as one structures application programs.
- Examples include THE, MULTICS, OS/2, Windows NT (more so in early releases).
- Arguments for — nice separation of concerns, modularity.
- Arguments against — tricky to plan layers, performance can be slow.

Slide 9

Microkernel Systems

- Idea — make kernel itself as small as possible, package other services separately, as independent processes.
- Examples include MINIX (written by Tanenbaum).
- Arguments for — modularity, reliability.
- Arguments against — tricky to plan layers, performance might be reduced.

Slide 10

Virtual Machines

- Idea — o/s provides a simulation of the actual physical machine, this "virtual machine" then runs another o/s – or several of them.
- Examples include VM/370, Windows support for old MS-DOS programs, VMware, Mac-on-Linux, Java Virtual Machine.
- Arguments for — separates multiprogramming from other concerns, emulation aspect can be useful, useful in o/s development.
- Arguments against — another layer, so can be slower. Also, may not be possible for some hardware — e.g., if privileged instructions executed in user mode are simply ignored.
- (Notice how this is an idea that fell out of favor for a while, then came back.)

Slide 11

VM/370

- Idea — provide multiple "virtual machines", each running its own o/s, which could be:
  - "Real" o/s such as MVS (another mainframe o/s) — in turn running many processes.
  - Not-quite-real o/s CMS — interactive single-user system rather like MS-DOS, runs under VM/370 only (not on real hardware).
- Allows sharing of physical resources among multiple "client" o/s's:
  - CPU sharing — similar to multitasking.
  - I/O device sharing — share physical devices, or allow exclusive use.

Slide 12

VM/370, Continued

- How does this work? briefly:
  - Client o/s's run native code, request o/s services in the usual way (interrupt or system call).
  - Interrupt handler is part of VM/370 — so it processes I/O requests/interrupts, errors, etc.
  - Client o/s system code runs in simulated supervisor mode (really user mode).
- Successors to VM/370 (VM/ESA, z/VM) currently being used to run many copies of Linux on a mainframe (!).

Slide 13

Minute Essay

- There is an old adage that says that any programming problem can be solved by adding a layer of abstraction, while any performance problem can be solved by removing a layer of abstraction.

  How (if at all) does this apply to operating systems and how they are structured?

Slide 14

Minute Essay Answer

- Based on the descriptions of the various operating-system structures, it looks like the general principle applies here too — adding layers of abstraction can improve correctness and reliability, but there is likely to be a performance cost.