Slide 1

Administrivia

- The calendar date brings back memories. History to y'all; not just that for me!
- Homework 1 posted; due a week from Monday. Some written problems and a programming problem. Programming problem will only work on something UNIX-like. So . . .
- I'm still working on suggestions for getting a UNIX-like environment on a Windows computer. Will let you know by e-mail. A fallback is just to use the Linux virtual desktop.
- Reading quiz over Chapter 1 coming soon. Also I'll let you know by e-mail.
- If you have a question, pause the video and make a note of it, and then put it in your minute essay.

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 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 4

Example: System Calls in MIPS

- MIPS instruction set includes `syscall` instruction that generate 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 5

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 6

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 7

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`, Cygwin under Windows; UNIX `sh`, `bash`, `csh`, `tcsh`, `ksh`, `zsh`, ...

Slide 8

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. (See Figure 1-19.)

  Note that the shell starts a new process for each command. Why do you think it does that? (Think about what happens if the command crashes.)
- 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++). (But I'm supplying starter code that does most of this.)

Slide 9

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 10

Sidebar: C/C++ Programming Advice

- I *strongly* recommend always compiling with flags to get extra warnings. There are lots of them, but you can get a lot of mileage just from `-Wall`. Add `-pedantic` to flag nonstandard usage.

  Warnings are often a sign that something is wrong. Only rarely should they be ignored! Sometimes the problem is a missing `#include`. `man` pages tell you if you need one.
- If you want to write "new" C (including C++-style comments), you may need to add `-std=c99`.

Slide 11

Sidebar: C/C++ Programming Advice, Continued

- If typing all of these gets tedious, consider using a simple makefile: Create a file called `Makefile` containing the following (the first line for C, the second for C++):

  ```
  CFLAGS = -Wall ....
  CXXFLAGS = -Wall ....
  ```

  and then compile `hello.c` to `hello` by typing `make hello`, or similarly for `hello.cpp`.

Slide 12

Minute Essay

- Questions?