Slide 1

Administrivia

- Programming problem added to Homework 1. Written problems still due Wednesday (hard-copy preferred); programming problem due next Monday (submit by e-mail).
- First quiz next Monday. Topic(s) from Chapter 1.

Slide 2

Minute Essay From Last Lecture

- (Review.)

Slide 3

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, discussed previously. Mechanism similar to interrupts, so . . .

Slide 4

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 5

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 6

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 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`; 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.
- 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++). (This year I'm supplying starter code.)

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

Compiler(s) on the Classroom/Lab Machines

- For the homework you will be writing a C or C++ program. I will test with the appropriate GNU compiler on the lab machines, so you should probably do so too.
- For what it's worth, the current (and just-previous) "build" running on the classroom/lab machines includes multiple versions of `gcc`. If you're using one of the non-default ones (perhaps because it's required for some other course, such as anything Dr. Lewis teaches using C++), it would be helpful to tell me so when you turn something in. More information about all of this on request.

Slide 11

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. 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`.

- 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

Slide 12

similarly for `hello.cpp`.

Slide 13

Minute Essay

- It appears from the pseudocode for the simple command shell that the shell starts a new process for each command. Why do you think it's done that way, rather than simply calling the command's main function?

Slide 14

Minute Essay Answer

- A reason that occurs to me is that it protects the shell itself from buggy or malicious commands.