Slide 1

Administrivia

- Reminder(?): For minute essays where there's some notion of a "right answer", it will be in the final version of the slides online, sometime after class.
- Homework 1 on the Web. Due next Friday. I recommend starting early on the programming problem.
- Reminders/requests about homework:

  All homework is considered pledged work. Write "pledged" on hardcopy work, and include it in comments for programming assignments. Also tell me if you got help from someone else, and/or worked with another student in the class.

  For work submitted by e-mail, *please* include the name or number of the course in the subject line of your message, plus something about which assignment it is, to help me get it into the correct folder for grading.

Slide 2

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.
- This year, though, there are two versions of the GNU compiler suite installed — the one included in the current release of Scientific Linux (gcc 4.4.7) and a newer one that supports more of the C++11 standard (gcc 4.8.1). To get access to the newer one, type

  `module load gcc-4.8.1`

  (You could probably put this in your `.bashrc` file. Ask me for details if need be.)

Slide 3

Minute Essay From Last Lecture

- Many people did figure out that the problem was related to using an unitialized pointer, but beyond that a lot of variation, and some confusion about what pointers are. (In context — basically memory addresses.)
- Key point is that MS-DOS didn't protect its own memory, so my little application program could (and presumably did) overwrite something important in the o/s's memory. Symptoms suggest that "something important" here was something related to processing keyboard input.

  The story may be badly titled, since it's not clear what's at fault — the hardware for not providing memory protection or MS-DOS for not using it. Either way it illustrates the risk of not having and using memory protection?

Slide 4

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 5

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 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

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 8

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 9

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 10

Minute Essay

- Is it really necessary to have a special instruction (such as TRAP) for system calls? Wouldn't it be just as good to call the operating system's code in the way other code is called?

Minute Essay Answer

- The advantage of having the special instruction is that it provides a safe/controlled way to get from user mode into kernel mode (since control is transferred to operating system code, which can do any other authorization that is needed). This switch of modes has to happen at some point, and I can't think of another way to make it happen safely.