Slide 1

Administrivia

- Homework 1 to be on the Web soon; due in a week.
- I say in the syllabus that I try to respond promptly to e-mail. Exceptions are minute essays and homeworks, which I don't always look at right away. If you need a quick reply, make that apparent on the subject line please!

Slide 2

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 3

Minute Essay From Last Lecture — Some Responses

- Pointers form a cycle? (But then why would control-alt-delete not work?)
- "I avoid pointers" since they can lead to even breaking the computer. (For the record — computer was fine after power-cycle, but my *program* . . .)
- No provision for throwing an exception on bad memory access? (Pretty much.)
- Various mentions of segfault — well, no such thing on this system.
- (Aside: "Java doesn't have pointers.")
- System "kept looking" for uninitialized pointer — ??

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.

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";
```

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 11

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 will supply starter code.)

Slide 12

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 13

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 multiple versions of the GNU compiler suite installed — the one included in the current release of Scientific Linux (gcc 4.4.7) and newer ones that support more of the C++11 standard (gcc 4.8.1, gcc 4.9.1). To get access to the newest one, type

  ```
  module load gcc-latest
  ```

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

Slide 14

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), 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 15

similarly for `hello.cpp`.

Slide 16

Minute Essay

- None really — sign in, any questions/comments? this is about all for chapter 1.