Slide 1

Administrivia

- Reminder: Homework 1 written problems due Monday at 5pm.
- I've put a copy of the textbook on 1-day reserve at the library if yours hasn't arrived yet.
- (Anyone notice that Chapter 1 has a whole section on C? Early editions didn't. Guess why it was added. You can feel smug!)

Slide 2

Minute Essay From Last Lecture

- If you're curious — a number of people got the answer I had in mind, or came close, but by no means all. Okay either way!
- Would you agree that irritating as "Segmentation fault" is, it's not as bad as this alternative?

Slide 3

System Calls (Review)

- 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 4

System Calls — Mechanism (Review)

- Library routine (running in user mode) sets up parameters and issues `TRAP` instruction or similar. One 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 5

System Calls — Services Provided (Review)

- 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 6

Interrupts

- (Discuss here partly since processing of `TRAP` instructions is similar to interrupts.)
- What are interrupts? 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. Seems useful, no?
- On interrupt, goal is to save enough of current state to allow us to restart current activity later, and then deal (perhaps minimally) with interrupt.

Slide 7

Interrupt Processing

- 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.
- (See Figure 1-11.)
- (Also see Figure 1-17 — system call processing.)

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

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 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. (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 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 C code (or C++ if you truly don't want to use the starter code). 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 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. 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 15

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 16

Process Abstraction

- We want O/S to manage "things happening at the same time" — applications, hidden tasks such as managing a device, etc.
- Key abstraction for this — "process" — program plus associated data, including program counter.
- True concurrency ("at the same time") requires more than one CPU/processor/core. Can get apparent concurrency via interleaving — model one virtual CPU per process and have the real processor switch back and forth among them ("context switch").

  (Aside: In almost all respects, this turns out to be indistinguishable from true concurrency. "Hm!"?)

Slide 17

Process Abstraction, Continued

- Can also associate with process an "address space" — range of addresses the program can use. Simplifying a little, this is "virtual memory" (like the virtual CPU) that only this process can use. More (lots more) about this later. (Nitpick: Yes, we also want to be able to share memory among processes. More about that later too.)
- How to map this to the real hardware? Chapter 2 talks about how to share the real CPU(s) among processes; chapter 3 talks about how to share the real memory.

Slide 18

Context Switches

- What is it? switch from one process to another.
- When should this happen?

Slide 19

Context Switches, Continued

- Should happen
  - when a process's "time slice" is up.
  - when there's an unrecoverable error.
  - when there's something that needs to be done right away (e.g., deal with input/output).
  - maybe other times? (when a process has to wait for something, e.g.).

  All signalled by some kind of interrupt.
- Goal is to suspend work on a process such that we can later pick up exactly where we left off. How do we make that happen?

  (Think about what the hardware does when an interrupt happens, what's included in that "virtual CPU".)

Slide 20

Context Switches, Continued

- On interrupt, hardware saves program counter (at least — why?), transfers control to fixed location — which contains O/S code.
- That O/S code has to
  - Save CPU state (program counter, registers, etc.) for the current process.
  - Deal with interrupt (details depend on type — I/O versus timer versus . . .).
  - Restore CPU state for "next" process (previously saved), thereby restarting it.

    ("Next" process? yes, O/S might have to choose — more about that later.)

Slide 21

Process Creation and Termination

- When are processes created?
  - At system startup.
  - When another process makes a "create process" system call — e.g., to start a new application.
- When are processes destroyed?
  - At program exit.
  - After some kinds of errors.
  - When another process makes a "kill process" system call.

Slide 22

Process States

- Can think of processes as being in one of three states:
  - "Running" — being executed by a CPU.
  - "Blocked" — waiting for something to happen (I/O to complete, another process to do something, etc.) and unable to do anything useful until it does.
  - "Ready" — not blocked, but waiting because all CPUs are currently executing other processes.
- Possible transitions? Which ones require decision-making?

Slide 23

Process States, Continued

- Possible transitions (Figure 2-2):
  - Running to blocked — happens when, e.g., a process makes an I/O request and can't continue until it's complete.
  - Blocked to ready — happens when the event the blocked process is waiting for occurs.
  - Running to ready, ready to running — needed if we want some sort of time-sharing (give all non-blocked processes "a turn" frequently).
- Notice that moving to and from "blocked" state doesn't involve decision-making, but ready/running transitions do.
- The decision-maker — "scheduler" (to be discussed later). Often "running to ready" is triggered by an interrupt (I/O, timer, etc.), and "ready to running" involves this scheduler.

Slide 24

Implementing Processes

- Think about how you would implement this abstraction . . .
- First, you'd want a data structure to represent each process, to include — what?

Slide 25

Implementing Processes, Continued

- Data structure to represent each process would include some way to represent such things as:
  - Process ID.
  - Process state (running / ready / blocked).
  - Information needed for context switch — a place to save program counter, registers, etc.
  - Other stuff as needed — e.g., a list of data structures for open files.
- Then you'd collect these into a table (or some similar structure) — "process control table", with individual data structures being "entries in the process control table" or "process control blocks".

Slide 26

Implementing Processes, Example — Linux

- Each process ("task") is represented by a C `struct` containing information similar to what we described.
- These `struct`s are chained as a doubly-linked list; there is also a hash table keyed by PID.
- (This is according to online information about the 2.4 kernel.)

Slide 27

Minute Essay

- In a system with 8 CPUs and 100 processes, what are the maximum and minimum number of processes that can be running? ready? blocked?
- How are you doing with regard to getting a copy of the textbook?

Slide 28

Minute Essay Answer

- Blocked: Maximum of 100 (unless you assume that there's an "idle" operating system process that runs when nothing else does and never blocks, and maybe one of these is needed for every CPU). Minimum of 0.
- Running: Maximum of 8, because there are 8 CPUs. Minimum of 0 (again unless you assume that there's an O/S process that runs when nothing else does).
- Ready: Maximum of 92, since all CPUs will be running processes if there are any that can be run. (Depending on details, you might have to add "except during context switches, when the scheduler is choosing the next process to run on a CPU".) Minimum of 0, since they could all be blocked or running.