Slide 1

Administrivia

- Reminder: Homework 1 written problems due Monday. Turn in on paper.
- Homework 1 programming problem on the Web; due next Wednesday. Turn in via e-mail.

Slide 2

Minute Essay From Last Lecture

- (Review.)
- Most people at least were “in the ballpark”.
- Would you agree that irritating as “Segmentation fault” is, it’s not as bad as this alternative?

Slide 3

System Call / Interrupt Processing — Recap/Review

- Recall(?) typical mechanism for regular program calls: Put parameters in agreed-on locations (registers, stack, etc.), issue instruction that saves current program counter (in another register maybe) and transfers control to called program. Called program returns using saved program counter.
- System calls are similar *except* that the "called program" is at a fixed address *and* the transfer of control also puts the processor in supervisor/kernel mode.

Slide 4

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 5

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 6

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 7

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 8

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 9

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 10

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 11

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 12

Context Switches

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

Slide 13

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 14

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 15

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 16

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 17

Process States, Continued

- Possible transitions (figure in textbook, p. 93):
  - 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 18

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 19

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 20

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 21

Processes Versus Threads

- So far I've used "process" in an abstract/general way.
- In typical implementations, though, "process" is more specific — something that has its own address space, list of open files, etc. Often these are called "heavyweight processes".
  - Advantages — such processes don't interfere with each other.
  - Disadvantages — they can't easily share data, switching between them is expensive ("a lot of state" to save/restore).
- For some applications, might be nice to have something that implements the abstract process idea but allows sharing data and faster context switching — "threads".

Slide 22

Threads

- So, threads are another way to implement the process abstraction.
- Typically, a thread is "owned" by a (heavyweight) process, and all threads owned by a process share some of its state — address space, list of open files.
- However, each thread has a "virtual CPU" (a distinct copy of registers, including program counter).
- Implementation involves data structures similar to process table.
- Advantages / disadvantages (compared to processes)?

Slide 23

Threads, Continued

- Advantages: threads can share data (same address space), switching from thread to thread is fairly fast.
- Disadvantages: sharing data has its hazards (more about this later).

Slide 24

Implementing Threads

- Two basic approaches — “in user space” and “in kernel space” Various hybrid schemes also possible.
- Basic idea of “in user space” — operating system thinks it’s managing single-threaded processes, all the work of managing multiple threads happens via library calls within each process.
- Basic idea of “in kernel space” — operating system is involved in managing threads, the work of managing multiple threads happens via system calls (rather than user-level library calls).
- How do they compare?. . .

Slide 25

Implementing Threads, Continued

- Implementing in user space is likely more efficient — fewer system calls.
- Implementing in kernel space avoids some problems, though:
  - If a thread blocks, it may do so in a way that blocks the whole process.
  - Preemptive multitasking is difficult/impossible without help from the kernel, as is using multiple CPUs.

Slide 26

Adding Multithreading

- If you've written multithreaded applications — moving from single-threaded to multithreaded not trivial:
  - Figure out how to split up computation among threads.
  - Coordinate threads' actions (including dealing properly with shared variables).
- Similar problems in adding multithreading to systems-level programs:
  - Deal properly with shared variables (including ones that may be hidden).
  - Deal properly with signals/interrupts.

Slide 27

Implementing Threads, Example — Linux

- Early versions of Linux provided no support for kernel-space threading, but there were libraries for the user-space version.
- More-recent kernels provide support, but in an interesting way — threads in some ways are just processes with with some different flags allowing them to share memory, etc.

  Adding support for threads complicates process creation — the basic mechanism (`fork`) duplicates an existing process, and if that process is multithreaded, things can be interesting. Some details in chapter 10, or read the POSIX standard for `fork`.

Slide 28

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?
- If you did an internship this past summer and you are free Tuesdays at 3:35pm, *please* consider responding to Dr. Zhang's request for student speakers. 10 minutes, no slides . . .

Slide 29

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.