Slide 1

Administrivia

- Correction to schedule: Homework 1 due next Friday (since that's what I said in class!).
- Quiz 1 next week (Friday). Topics TBA next time.
- (A semi-random comment, possibly of interest: Section of chapter 1 about C is new with this edition of Tanenbaum's book. Apparently not all schools make the same choices we do about what language(s) to use in courses!)

Slide 2

Minute Essay From Last Lecture

- Quite a bit of variation in answers. Some people mentioned efficiency. Possibly related (in some way I'm not sure I get?), but the key point is whether the o/s can defend itself.
- 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 3

Overview of Hardware — Recap

- Idea is to get a sense of what o/s designers/developers have to work with.
- Notice also what features seem intended to make it possible to write an o/s that can defend itself!
- (I won't talk in class about the sections on buses and booting, but do read them.)

Slide 4

Overview of Hardware — Recap

- Idea is to get a sense of what o/s designers/developers have to work with.
- Notice also what features seem intended to make it possible to write an o/s that can defend itself!
- (I won't talk in class about the sections on buses and booting, but do read them.)

Slide 5

Operating System Functionality — Recap

- “Operating system as virtual machine” must provide key abstractions (processes, filesystems).
- “Operating system as resource manager” must manage resources (memory, I/O devices, etc.).
- Operating system functionality typically packaged as “system calls”.
- Details obviously vary among systems, but some ideas are common to most/many . . .

Slide 6

Processes — Abstraction

- Basic idea — a program (application or background activity) together with its current state (registers and memory contents).
- In order to have more than one at a time, need some way to share the physical machine among them.
- May be useful to think in terms of each process having its own simulated processor and memory (“address space”), with operating system providing infrastructure to map that onto the hardware. How to do that? (Next slide.)
- Other relevant concepts include process ownership, hierarchical relationships among processes, interprocess communication.
- Relevant system calls — create process, end process, communicate with another process, etc.

Slide 7

Processes — Implementation

- Managing the “simulated processor” aspect requires some way to timeshare physical processor(s). Typically do that by defining a per-process data structure that can save information about process. Collection of these is a “process table”, and each one is a “process table entry”.
- Managing the “address space” aspect requires some way to partition physical memory among processes. To get a system that can defend itself (and keep applications from stepping on each other), memory protection is needed — probably via hardware assist. Some notion of address translation may also be useful, as may a mechanism for using RAM as a cache for the most active parts of address space, with other parts kept on disk.

Slide 8

Filesystems

- Most common systems are hierarchical, with notions of “files” and “folders”/“directories” forming a tree. “Links”/“shortcuts” give the potential for a more general (non-tree) graph.
- Connecting application programs with files — notions of “opening” a file (yielding a data structure programs can use, usually by way of library functions).
- Many, many associated concepts — ownership, permissions, access methods (simple sequence of bytes, or something more complex?), whether/how to include direct access to I/O devices in the scheme.
- Relevant system calls — create file, create directory, remove file, open, close, etc., etc.
- See text for some UNIXisms — single hierarchy, regular versus special files, pipes, etc.

Slide 9

I/O

- As noted previously — hardware is diverse, and communicating with it may involve a lot of messy details.
- So — typically there is an “I/O subsystem”, often involving multiple layers of abstraction. More later!

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`; 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++). Some hints in the homework writeup. Remember that C doesn't protect you from "buffer overflows" (e.g., there's a reason `gcc` complains about `gets`).

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

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 14

similarly for `hello.cpp`.

Slide 15

Hardware, Software, and History

- Textbook has a section called “Ontogeny Recapitulates Phylogeny”. Many interesting general observations:
- What seems like a good idea in software is strongly influenced by what the hardware can do. (I think it goes the other way too, but that’s speculation.)
- As in other areas of human endeavor, evolution of operating systems is in some ways cyclic: What seems brilliant now may be “ready for the scrap heap” in a few years — and then resurface as brilliant later. (This is why it’s not useless to read about approaches not currently in use.)

Slide 16

Minute Essay

- Tell me something you learned from reading chapter 1 of the textbook.