Slide 1

Administrivia

- My plan to teach class asynchronously Monday went awry. (My semester is really not off to a good start! But there's still time to improve.) Instead I plan to record one more lecture for this week, for you to watch before next Monday. To be available by Friday (I hope earlier); I'll send e-mail.
- Homework 1 to be assigned by Friday, due a week from following Monday. Some written problems, one programming problem. More in next lecture.
- (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

- Few people mentioned having used anything especially primitive or clunky, though there were a couple of mentions of a command line.
- A few people mentioned find Macs not as easy to use as Apple seems to think. "Hm!" Intuitive is in the mind of the user. . .
- At least one person mentioned that newer systems seem designed to make access to internals difficult or even impossible. Probably for most users this is a good thing, but irritating for experts and would-be experts?
- (Someone asked me in class one year how I'd answer that? a very small/limited IBM mainframe, with most data stored on punched cards, and several mechanical gadgets for processing them. See the Wikipedia article on "unit record equipment".)

Slide 3

Operating System Functionality — Overview

- Provide a “virtual machine”:
  - Filesystem abstraction — files, directories, ownership, access rights, etc.
  - Process abstraction — “process” is one of a collection of “things happening at the same time” (multiple users, multiple applications, background activities such as print spooling, etc.).
- Manage resources (probably on behalf of multiple users/applications):
  - Memory.
  - CPU cycles (one or more CPUs).
  - I/O devices.
- To do all of this effectively, O/S must be “in charge”, and able to defend itself from buggy or malicious applications, user programs.

Slide 4

Overview of Hardware

- Given these goals, useful to know next what we have to work with — i.e., hardware capabilities.
- So textbook presents a simplified view of hardware (as it appears to programmers) — processor(s), memory, I/O devices, bus.
- Figure 1-6 shows simplified view of overall organization — components connected to a single bus. (Actual processors may have more than one bus.)

Slide 5

Processors

- “Instruction set” of primitive operations — load/store, arithmetic/logical operations, control flow.
- Basic CPU cycle — fetch instruction, decode, execute. (Again, this is simplified — pipelined or “superscalar” architectures overlap these steps.)
- Registers — “local memory” for processor; general-purpose registers for arithmetic and other operations, special-purpose registers (e.g., program counter, stack pointer, program status word (PSW)).

  Note that *all programs* have access to some registers (not clear how they could run otherwise!). Access to some special-purpose registers may be restricted (more shortly).

Slide 6

Processors, Continued

- “Interrupts” — mechanism for interrupting normal flow of control; particularly useful when:
  - Something has gone wrong and it doesn’t make sense to continue. (The infamous-among-C-programmers “Segmentation fault”, e.g.!)
  - Something has happened outside the processor (e.g., an I/O device has completed something the O/S asked it to do).

  On interrupt, flow of control goes to O/S “interrupt handler”. Can eventually return to interrupted program, or not.
- Typically also include features useful in writing an operating system that can “defend itself” . . .

Slide 7

Dual-Mode Operation / Privileged Instructions

- Useful to have mechanism to keep application programs from doing things that should be reserved for O/S (e.g., enable/disable interrupts).
- Usual approach is to define in hardware:
  - Two modes for processor (supervisor/kernel and user). (Note that some systems define more, but basic idea is the same.)
  - Set of privileged instructions, to be executed only in kernel mode.
- Bit in PSW indicates which mode. Attempting to execute privileged instruction in user mode results in exception/interrupt.
- When to switch modes? when O/S starts application program, when application program requests O/S services, on error.
- How to switch? kernel to user seems straightforward, but how about the other way? Usually handled via `TRAP` or similar instruction, which generates an interrupt. (More shortly.)

Slide 8

Sidebar: Multithreaded and Multicore Chips

- Historical note: For many years (at least 30, to my knowledge) advocates of parallel programming were saying that eventually hardware designers would run out of ways to make single processors faster — and finally, some 10–15 years ago, it happened!
- Basic idea — number of transistors one can put on a chip kept increasing, and for a long time hardware designers used that to make single processors faster (e.g., with longer pipelines). But then they apparently ran out of ideas. So, instead, they chose to provide (more) hardware support for parallelism. Various approaches, including "hyperthreading" (fast switching among threads), "multicore" (multiple independent CPUs, possibly sharing cache), "GPGPU" (use of graphic card's many processors for computation).

Memory Hierarchy

Slide 9

- In a perfect world — fast, big, cheap, as permanent as desired.
- In this world — hierarchy of types, from fast but expensive to slow but cheap: registers, cache, RAM, magnetic disk, magnetic tape. (See Figure 1-9.)
- Note also — some types volatile, some non-volatile.

Registers and Caches

Slide 10

- Registers — part of processor, fastest to access but most expensive to build. Managed explicitly in software.
- Caches (possibly multiple levels) — less fast, less expensive, bigger. Mostly managed by hardware.
- Aside: Caching is a widely used strategy in computing! hardware caches, browser caches, etc., etc.

Main Memory (RAM)

- Still less fast, less expensive, bigger.
- Access managed by software.
- Shared among processes — which presents some interesting challenges . . .

Slide 11

Disk

- Can even use disk as sort of overflow area for RAM — “virtual memory” (more later). Even less fast, but cheaper and potentially larger.
- Also managed (mostly) by software.

Slide 12

Slide 13

Memory Protection

- Very useful to have a way to give each process (including O/S) its own variables that other processes can't alter.
- Usual approach — provide a hardware mechanism such that attempting to access memory out of range generates exception/interrupt. Several ways; some simple ones:
  - Limit each process to a range of memory locations; hold starting and ending addresses in special registers.
  - Partition memory into blocks, give each block a numeric key, give each process a key, and only allow processes to access blocks if keys match.

  (Real systems these days mostly use approaches that are more complex. Much more later when we talk about memory management.)

Slide 14

I/O Devices

- What they provide (from the user's perspective):
  - Non-volatile storage (disks, tapes).
  - Connections to outside world (keyboards, microphones, screens, etc., etc.).
- Distance between hardware and "virtual machine" is large here, so usually think in terms of:
  - Layers of software abstraction (as with other parts of O/S).
  - Layers of hardware abstraction too: most devices attached via controller, which provides a hardware layer of abstraction (e.g., "IDE controller").

Slide 15

I/O Basics

- Devices / controllers managed via "device registers" — typically including one you write to to tell the device what to do, one you can read from to check status – and a data buffer.
- CPU communicates with device (controller) by reading/writing device registers. Device controllers then translate to specifics for device.
- Two ways for CPU to communicate: memory-mapped I/O, special I/O instructions.
- Two approaches to coordinating with device: polling, interrupts.
- Normally only O/S allowed to interact with devices. User processes make requests to the O/S.
- Functionality for a particular device packaged as "device driver".

Slide 16

Overview of Hardware — Recap

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

Slide 17

Operating System Functionality — Review

- “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” (more later).
- Details obviously vary among systems, but some ideas are common to most/many (more later).

Slide 18

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 O/S 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.

Slide 19

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 20

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.

I/O

Slide 21

- 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!

Hardware, Software, and History

Slide 22

- 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?)

The Operating System "Zoo"

Slide 23

- Section of this name in textbook talks briefly about different kinds of operating systems.
- Key point here? a lot of variation in situations (combination of hardware and "use case") where an O/S is needed, worth thinking about what implications that might have for O/S.

Operating System Structures

Slide 24

- General-purpose operating systems are big — tens of millions of lines of code (traditionally often in C with some assembly language, though not always). How to organize all of it? several choices, discussed in textbook (read about it there — but okay to skim).
- A possibly-relevant maxim, origin unknown (to me): "Any programming problem can be solved by adding a layer of abstraction. Any performance problem can be solved by removing a layer of abstraction." Not always true, but true enough?

Slide 25

Minute Essay

- I once had a learning experience about “how DOS is different from a real O/S”. Summary version: A program using pointers (possibly uninitialized) caused the whole machine to lock up, so thoroughly that the only recovery was to power-cycle.

  What do you think went wrong?

  (Stick around for five minutes to give everyone a chance to answer. Then we’ll discuss.)

Slide 26

Minute Essay Answer

- The program changed memory at the addresses pointed to by the uninitialized pointers — and this memory was being used by the O/S, possibly to store something related to interrupt handling. A “real” O/S wouldn’t allow this!

  (Then again, the version of MS-DOS in question was supposedly written to run on hardware that didn’t provide memory protection, so maybe it’s not DOS’s fault.)