Slide 1

Administrivia

- About the textbook, it appears that many people have bought electronic copies and most of the rest are ordering or have ordered from online sources. I don't know when the bookstore will have copies.
- Homework 1 on the Web (written problems only, for now). Due in a week and a half (to allow time for textbooks to arrive? — but at least one problem you should be able to do without a textbook). Programming problem, to be added and discussed next time, will be due later.

  Note request for information about collaboration ("honor code statement").

Slide 2

Minute Essay From Last Lecture

- Not everyone had exposure to anything very old/different.
- But some did: post-"jailbreak" Wii, Linux without GUI, debugging instructions on a processor (for firmware company — sounds interesting!), PC without GUI, TI 83, rotary phone, Commodore 64C, various old-and-slow PCs, Windows 98.

Slide 3

Operating System Functionality — Overview

- Provide a “virtual machine”:
  - Filesystem abstraction — files, directories, ownership, access rights, etc.
  - Process abstraction — “process” is a name for 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.

Slide 4

Overview of Hardware

- 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 (program counter, stack pointer, program status word (PSW)).

  Notice that *all programs* have access to some registers (not clear how they could run otherwise!), though one might want to restrict access to some special-purpose registers.

  Not shared among processes in the sense of simultaneous use (but context switches "fake it" — much more later).

Slide 6

Processors, Continued

- Typically also include features useful in writing an operating system . . .

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.
- Usual approach — in hardware, define two modes for processor (supervisor/kernel and user), privileged instructions.
  - Privileged instructions — things only O/S should do, e.g., enable/disable interrupts.
  - Bit in PSW indicates kernel mode (O/S only, privileged instructions okay) or user mode (application programs, privileged instructions not allowed).
  - 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 about interrupts later).

Slide 8

Multithreaded and Multicore Chips

- For many years (at least 30, to my knowledge) advocates of parallel programming have been saying that eventually hardware designers would run out of ways to make single processors faster — and finally it seems to be happening.
- 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).

Slide 9

Memory Hierarchy

- 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.

Slide 10

Registers and Caches

- 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! virtual memory, disk blocks in memory, etc., etc.

Main Memory (RAM)

Slide 11

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

Memory Protection

Slide 12

- 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 ranges 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.

Slide 13

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 s/w abstraction (as with other parts of O/S).
  - Layers of h/w abstraction too: most devices attached via controller, which provides a h/w layer of abstraction (e.g., "IDE controller").

Slide 14

I/O Basics

- CPU communicates with device controller by reading/writing device registers; device controller communicates with device.
- Memory-mapped I/O versus I/O instructions.
- Polling versus interrupts.
- Functionality for a particular device packaged as "device driver".
- I/O in application programs — make system call to invoke O/S services.

Slide 15

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 16

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 17

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.

Slide 18

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.

Filesystems

Slide 19

- 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 20

- 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 21

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 22

The Operating System “Zoo”

- 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.

Slide 23

Operating System Structures

- General-purpose operating systems are big — tens of millions of lines of code (probably mostly in something C-like). How to organize all of it? several choices, discussed in textbook:
  - Monolithic systems.
  - Layered systems.
  - Microkernels.
  - Client-server model.
  - Virtual machines.
  - Exokernels.
- 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 24

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 25

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 26

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 27

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 28

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 29

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 30

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?

  (After you've told me, stick around a few minutes and I'll show my answer.)

Slide 31

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.)