Slide 1

Administrivia

- A reminder: *Please do not* reboot the machines in HAS 340! People depend on these machines to do background processing.

  If a previous user has left a machine in the "locked by screensaver" state, you can bail out by pressing control-alt-backspace to restart X (the graphical subsystem) without disturbing background processes.

  If you log out from the "System" menu, it might be easy to shut down by mistake. Can put an icon on the task bar for logout to avoid this.

- Prox card access should be enabled now, so you should be able to get into the labs after hours.

Slide 2

Operating System Functionality

- 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" (in effect if not in fact), including:
    - In batch systems, user "jobs", plus input/output spooling.
    - In timesharing system, interactive users.
    - In PC o/s, concurrently-executing tasks.

    Here too, idea of ownership / access rights.
- Manage resources (probably on behalf of multiple users/applications):
  - Memory.
  - CPU cycles (one or more CPUs).
  - I/O devices.

Slide 3

Overview of Hardware

- Simplified view of hardware (as it appears to programmers) — processor(s), memory, I/O devices, bus.
- (See figure, p. 21.)
- Next few sections talk about each component — what it does (from user's point of view) and low-level interface to software.

Slide 4

Processors

- "Instruction set" of primitive operations — load/store, arithmetic/logical operations, control flow.
- Basic CPU cycle — fetch instruction, decode, execute.
- Registers — "local memory" for processor; general-purpose registers for arithmetic and other operations, special registers (program counter, stack pointer, program status word (PSW)).
- Now consider what additional features would make it easier to write an operating system . . .

Slide 5

Interrupt Mechanism

- 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.
- Usually have a `TRAP` instruction for generating interrupt.
- Could you write an o/s without this?

Slide 6

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 and user), privileged instructions.
  - Privileged instructions — things only o/s should do, e.g., enable/disable interrupts.
  - Bit in PSW indicates supervisor 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.
- Could you write an o/s without this?

Slide 7

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 ranges generates exception/interrupt; several ways, including:
  - 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.
- Could you write an o/s without this?

Slide 8

Timer

- Useful to have a way to set a timer / "alarm clock" — e.g., to get control back if application program enters infinite loop.
- Usual approach — hardware features that tracks real time and can be set to interrupt CPU.
- Could you write an o/s without this?

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 picture, p. 24.)
- Note also — some types volatile, some non-volatile.

Slide 10

Program Relocation

- At the machine-instruction level, references to memory are in terms of an absolute number. Compilers/assemblers can generate these only by making assumption about where program will reside in memory.
- In the very early days, programs started at 0, so no problem. Now they hardly ever do, so we need a way to relocate programs — when loaded, or "on the fly".
- "On the fly" relocation uses MMU (memory management unit) — which can provide both program relocation and memory protection.

  Logically between CPU and memory, physically usually part of CPU.

  A simple scheme — base and limit registers (described in text). When do values in them need to change?

Slide 11

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 12

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 (more about system calls later).

Slide 13

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?

Slide 14

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.)