Slide 1

Administrivia

- Reminder: Homework 1 due today (written part by 5pm, programming part by 11:59pm). Hardcopy preferred for written part, but okay to submit by e-mail if you must!
- Reminder: Quiz 1 Wednesday. Short question(s) based on chapter 1, lectures. “Open book/notes”, meaning access to textbook, your notes, anything on course Web site.

Slide 2

Minute Essay From Last Lecture

- Eight people (out of 20) did not have a book yet(!).
- Many comments were along the line of “I didn’t realize X was so complicated.” Yes, amazing how much typical UIs these days hide . . .

Slide 3

Operating System Structures

- Clearly o/s could involve a whole lot of code (tens of millions of lines of code, if one can believe Wikipedia). How to structure?
- Choices include:
  - Monolithic systems.
  - Layered systems.
  - Microkernels.
  - Client-server model.
  - Virtual machines.
  - Exokernels.

Slide 4

Monolithic Systems

- Tanenbaum's description in the previous edition of the textbook — "The Big Mess". Maybe an exaggeration, since there can be *some* structure.
- Examples include MS-DOS, early UNIX.
- Arguments for/against?

Slide 5

Monolithic Systems, Continued

- Arguments for this approach — "works, sort of"?
- Arguments against — easier for one malfunctioning component to crash others.

Slide 6

Layered Systems

- Idea — use layers of abstraction, just as one structures application programs.
- Examples include THE, MULTICS, OS/2, Windows NT (more so in early releases).
- Arguments for/against?

Slide 7

Layered Systems, Continued

- Arguments for — it's an extra layer of abstraction.
- Arguments against — it's an extra layer of abstraction.
- (Or in other words — nice separation of concerns, modularity, but tricky to plan layers, performance can be slow.)

Slide 8

Microkernel Systems

- Idea — make kernel itself as small as possible, package other services separately, as independent processes.
- Examples include MINIX (written by Tanenbaum).
- Arguments for/against?

Slide 9

Microkernel Systems, Continued

- Arguments for — modularity, reliability.
- Arguments against — tricky to plan layers, performance might be reduced.

Slide 10

Virtual Machines

- Idea — o/s provides a simulation of the actual physical machine, this “virtual machine” then runs another o/s – or several of them.
- Examples include VM/370, Windows support for old MS-DOS programs, VMware, Java Virtual Machine, other virtualization schemes.
- (Notice how this is an idea that fell out of favor for a while, then came back.)
- Arguments for/against?

Slide 11

Virtual Machines, Continued

- Arguments for — separates multiprogramming from other concerns, emulation aspect can be useful, useful in o/s development.
- Arguments against — another layer, so can be slower. Also, may not be possible for some hardware — e.g., if privileged instructions executed in user mode are simply ignored.

Slide 12

VM/370

- Idea — provide multiple "virtual machines", each running its own o/s, which could be:
  - "Real" o/s such as MVS (another mainframe o/s) — in turn running many processes.
  - Not-quite-real o/s CMS — interactive single-user system rather like MS-DOS, runs under VM/370 only (not on real hardware).
- Allows sharing of physical resources among multiple "client" o/s's:
  - CPU sharing — similar to multitasking.
  - I/O device sharing — share physical devices, or allow exclusive use.

Slide 13

VM/370, Continued

- How does this work? briefly:
  - Client o/s's run native code, request o/s services in the usual way (interrupt or system call).
  - Interrupt handler is part of VM/370 — so it processes I/O requests/interrupts, errors, etc.
  - Client o/s system code runs in simulated supervisor mode (really user mode).
- Successors to VM/370 (VM/ESA, z/VM) currently being used to run many copies of Linux on a mainframe (!).

Slide 14

Words of Wisdom?

- A very smart person I know once said the only interesting part of an o/s course was concurrent algorithms (to be covered soon), and the rest is "just details".

  A student a few years ago said "a lot of this just seems like common sense" (once you understand the basic ideas).

  Both sort of right . . .
- Goal of this course is to learn/retain basic ideas. Details may help with that — and can be interesting in themselves — but should not be the focus.
- (Both things to keep in mind as you continue reading and we continue discussing . . .)

Slide 15

Process Abstraction — Preview

- 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 (more properly now, "more than one CPU/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!"?)
- 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.

Slide 16

Context Switches

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

Slide 17

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 18

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 19

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 20

Minute Essay

- What is/was most difficult about doing Homework 1? What was most interesting?