Slide 1

Administrivia

- Reading quiz(zes) on Chapter 3 coming soon.
- I'm still hoping/planning to make up an extra-credit assignment on concurrency, but it's taking more time than I thought.
- Not a lot of class weeks left! so I'm going to try to stick to essentials so we can address major stuff. Plan is to address important and relevant material in Chapter 3 and high spots of Chapters 4 and 5.

Slide 2

Memory Management — Overview

- One job of operating system is to "manage memory" — assign sections of main memory to processes, keep track of who has what, protect processes' memory from other processes.
- As with CPU scheduling, we'll look at several schemes, starting with the very simple. For each scheme, think about how well it solves the problem, how it compares to others.
- As with processes, tradeoff between simplicity and providing a nice abstraction to user programs.

Simple Schemes — No Abstraction

Slide 3

- Memory (a.k.a. “RAM”) can be thought of as a very long list of numbered cells (usually bytes). (Somewhat simplified view but good enough for our purposes.)
- Simplest schemes for managing it don’t try to hide that view. (Names for these come from older edition of Tanenbaum’s book.)

  Going through these quickly, mostly for historical perspective . . .

  For each consider tradeoffs — complexity versus flexibility, efficient use of memory.

Monoprogramming

Slide 4

- Idea — only one user program/process at a time, stays resident until finished. Only decision to make is how much memory to devote to O/S itself, where to put it.

  (Figure 3-1 in textbook.)
- Very simple (good) but very restrictive.
- Used in very early mainframes, MS-DOS; still used in some embedded systems.

Slide 5

Multiprogramming With Fixed Partitions

- Idea — partition memory into fixed-size “partitions” (maybe different sizes), one for each process. Possibly also add the ability to “swap” programs (later).
- Limits “degree of multiprogramming” (how many processes can run concurrently).
- Probably necessitates “admissions scheduling” (some way of controlling which processes even get to start) — either one input queue per partition, or one combined queue.

  If one combined queue, how to choose from it when a partition becomes available? first job that fits? largest job that fits? etc.
- Fairly simple but pretty restrictive.
- Used in early mainframes.

Slide 6

Multiprogramming With Variable Partitions

- Idea — separate memory into partitions as before, but allow them to vary in size and number. (Figure 3-4 in textbook, sort of.)

  I.e., “contiguous allocation” scheme.
- Like previous scheme, necessitates admissions scheduling.
- Requires that we keep track of locations and sizes of processes’ partitions, free space. Note potential for memory fragmentation.
- Also fairly simple but restrictive.
- Used in early mainframes.

Slide 7

Multiprogramming With Variable Partitions, Continued

- Another implementation issue — how to decide, when starting a process, which of the available free chunks to assign.
- Several strategies possible (more in textbook if interested, but choices include first bit, best fit, worst fit).

Slide 8

Multiprogramming with Fixed/Variable Partitions — Recap

- Comparing the two schemes:
  - Similar admission scheduling issues.
  - Both pretty simple and pretty restrictive, though variable partitions are less so. Neither makes great use of memory.
- Either could be adequate for a simple batch system, maybe with the addition of swapping.

Slide 9

Swapping

- Idea — move processes into / out of main memory (when not in main memory, save on disk).

  (Note that programs do have to be in memory to run!)
- Addresses some problems (what if a process blocks for an extended time?), could also provide a way to “fix” fragmentation.
- Implies another level of scheduling (what to swap in/out).
- Less simple but more flexible.

Slide 10

Sidebar: Three-Level Scheduling

- Basic idea — break up problem of scheduling (batch) work into three parts:
  - Admissions scheduling — choose from input queue which jobs to “let into the system” (create processes for).
  - Memory scheduling — choose from among processes in system which to keep in memory, which to “swap out” to disk.
  - CPU scheduling — choose from among processes in memory which to actually run.
- Points to consider:
  - Are there advantages to limiting how many processes, how many in memory? What criteria could we use?
  - Are there advantages to the explicit three-level scheme?
  - Would this (or a variant) work for interactive systems?
  - Do all three schedulers have to be efficient?

Sidebar: Program Relocation

Slide 11

- Recall(?) that for most systems memory can be thought of one big one-dimensional space. Hardware references to memory are via an index into this space ("absolute address").
- At the machine-instruction level, load/store references to memory use an absolute address.
- You may recall from CSCI 2321 that in the MIPS architecture this address can be computed based on contents of a register or on the program counter, and that doesn't change based on where the program resides in memory. But for some instructions the address comes from the actual instruction (i.e., it's an absolute address). Fairly typical.

Program Relocation, Continued

Slide 12

- You may also recall from the discussion of assembling and linking that generating these absolute addresses is a bit complicated, since they can't be known at least until link time. But even then, they depend on where the program will reside in memory.
- In the very early days, all programs loaded at address 0, so no problem. With monoprogramming, too, all programs reside at the same address, so no problem. (SPIM works that way.)
- What happens, though, if you want to have multiple programs in memory? compilers/assemblers can't generate correct absolute addresses.
- This is the "relocation problem". What to do?

Program Relocation, Continued

Slide 13

- One solution: Generate, as part of the executable, a list of locations where there's an absolute address, and modify it as the program is loaded into memory. (This won't work well if we include swapping.)
- One solution to the memory-protection problem — storage-protection keys (IBM 360, an early mainframe).
- Better solution involves translating addresses "on the fly". Also helps with memory protection (making sure processes don't have access to each other's data, at least without explicit sharing).

Sidebar: The "Address Space" Abstraction

Slide 14

- Basic idea somewhat analogous to process abstraction, in which each process has its own simulated CPU. Here, each process has its own simulated memory.
- As with processes, implementing this abstraction is part of what an operating system can/should do.
- Usually, though, O/S needs help from hardware . . .

Slide 15

Dynamic Address Translation

- Underlying idea — separate program addresses (relative to start of program's "address space") from physical addresses (memory locations), and map program addresses to physical addresses. Also try to identify out-of-bounds addresses.
- Only practical way to implement — hardware "memory management unit" that logically sits between the CPU and memory. (Figure 3-8 in text.)

  Simplifying, CPU references program addresses, MMU turns them into physical addresses, generates interrupt if invalid.

Slide 16

A Simple MMU

- Idea — map each process's address space to a contiguous chunk of real memory, based on base and limit addresses (B and L):

  Program address p maps to memory location $B + p$.

  If $B + p > L$, invalid (out of bounds).

  If B and L are different for each process — solves both problems.
- Turn this into hardware (MMU) by using base and limit registers.
- Solves both the relocation and protection problems.
- Simple but restrictive (each process's memory must be one contiguous chunk).
- Used in some early mainframes and PCs.

Slide 17

Memory Management with Contiguous Allocation

- Simplest MMU (just described) uses two registers, base and limit. More or less implies that each process can have only one contiguous chunk of memory. (Note here the interaction between hardware design and O/S design.)
- Key issues here are keeping track of what space is used by what, and deciding how to assign memory to processes.

Slide 18

Simple Memory Management — Recap

- Contiguous-allocation schemes are simple to understand, implement.
- But not very flexible — each process's memory must be contiguous, swapping is all-or-nothing.
- Can we do better? yes, by relaxing one or both of those requirements — "paging".

Slide 19

Pause

- (Pause to try to combat Zoom fatigue.)

Slide 20

Paging — Overview

- Idea — divide both address spaces and memory into fixed-size blocks (“pages” and “page frames”), allow non-contiguous allocation.
- Seems like this would be more flexible and make better use of memory, but would be much more complex? Yes . . .

Slide 21

Paging — Mapping Program to Physical Addresses

- One consequence — mapping from program addresses to physical addresses is much more complicated.
- How? “page table” for each process maps pages of address space to page frames; use this to calculate physical address from program address.

  (Are there page sizes for which this is easier?)
- As with base/limit scheme, makes more sense to implement this in MMU. (Note again interaction between hardware design and O/S design.)
- Could let page table size vary, but easier to make them all the same (i.e., each process has the same size address space), have a bit to indicate valid/invalid for each entry. Attempt to access page with invalid bit — “page fault”.

  (Figures 3-9, 3-10 in textbook.)

Slide 22

Paging and Virtual Memory

- Idea — extend this scheme to provide “virtual memory” — keep some pages on disk. Allows us to pretend we have more memory than we really do. (Not as important these days as previously, but still, sometimes it seems like however much you have of a resource it isn’t always enough?)
- (Compare to swapping.)

Slide 23

Paging and Memory Protection

- This scheme also provides memory protection. (How?)
- We could also use it to allow processes to share memory. (How?)

Slide 24

Minute Essay

- To do its job, a MMU that uses paging must have access to current process's page table. The textbook mentions two simple schemes for doing this:
  - Keep entire table in (processor) registers.
  - Keep table in memory and have a particular processor register point to its starting location.
- What advantages/disadvantages can you think of for each of these? (Think about context switching between processes and also about how quickly MMU will be able to translate each address.)

Slide 25

Minute Essay Answer

- First scheme almost surely makes for faster translations, but for a large page table it will require a lot of registers, which even if feasible would make context switches slow.

  Second scheme won't slow down context switches, but as stated it isn't going to make for fast translation.