Slide 1

Administrivia

- I'm still hoping/planning to make up an extra-credit assignment, but it's taking more time than I thought.

Slide 2

Homework 3 Essays

- On written problems, no clear consensus, and nothing really stood out, except:

  One person commented on how pseudocode was less satisfactory than real code because you couldn't demonstrate correctness by testing. For sequential programs that's kind of true, though it depends on how good the tests are, no? But for concurrent programs, it's less true because output can vary depending on timing.

- On programming problem, not many people have turned in final versions, but among those that have:

  A few thought this was fun (well, if you like programming? as I do!)

  A few commented that figuring out one algorithm generally made the next one easier. Makes sense!

Slide 3

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, there’s a tradeoff between simplicity and providing a nice abstraction to user programs.

Slide 4

Simple Schemes — No Abstraction

- Memory (a.k.a. “RAM”) can be thought of as a very long list of numbered cells (usually bytes). (This is a 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.)

Slide 5

Monoprogramming

- 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.)
- Consider tradeoffs — complexity versus flexibility, efficient use of memory.
- Used in very early mainframes, MS-DOS; still used in some embedded systems.

Slide 6

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 slide).
- 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.
- Consider tradeoffs — complexity versus flexibility, efficient use of memory.
- Used in early mainframes.

Slide 7

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.)

  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. Notice potential for memory fragmentation.
- Consider tradeoffs — complexity versus flexibility, efficient use of memory.
- Used in early mainframes.

Slide 8

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:
  - First fit.
  - Next fit.
  - Best fit.
  - Worst fit.
  - Quick fit.

Slide 9

Multiprogramming with Fixed/Variable Partitions — Recap

- Comparing the two schemes:
  - Similar admission scheduling issues.
  - Complexity versus flexibility, memory use also roughly similar.
- Either could be adequate for a simple batch system, maybe with the addition of swapping.

Slide 10

Sidebar: Program Relocation

- Recall(?) that for most systems memory can be thought of a 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).

Slide 11

Program Relocation, Continued

- 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.

  (Figure 3-2 in textbook.)
- This is the "relocation problem". What to do?

Slide 12

Program Relocation, Continued

- 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 introduce swapping, discussed soon.)
- A better solution involves translating addresses "on the fly" — and this solution also helps with memory protection (making sure processes don't have access to each other's data, at least without explicit sharing).

Slide 13

Sidebar: The “Address Space” Abstraction

- Basic idea is 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 14

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 15

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.
- Consider tradeoffs — complexity versus flexibility.
- Used in some early mainframes and PCs.

Slide 16

Memory Management with Contiguous Allocation

- Simplest MMU (just described) uses two registers, base and limit. This more or less implies that each process can have only one contiguous chunk of memory. (Notice 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.

  (Figure 3-3 in text.)

Slide 17

Swapping

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

  (Aside — can we run a program directly from disk?)
- Addresses both questions from previous slide; could also provide a way to “fix” fragmentation.
- Implies another level of scheduling (what to swap in/out).
- Makes non-dynamic solutions to relocation problem much less attractive. MMU-based solution still works, though, and adds memory protection.
- Consider tradeoffs again — complexity versus flexibility, efficient use of memory.

Slide 18

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?

Simple Memory Management — Recap

Slide 19

- Contiguous-allocation schemes are simple to understand, implement.
- But they're not very flexible — 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".

Paging — Overview

Slide 20

- 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 . . . (To be continued.)

Minute Essay

- None really — just sign in.