Slide 1

Administrivia

- Homework 4 on the Web. Due next Wednesday.
- Midterm grade summaries (to be) e-mailed.

Slide 2

Memory Management — Review

- The problem we're solving — partition physical memory among processes.
- Two related issues — program relocation and memory protection. Whether program relocation is potentially a problem depends on the processor's instruction set and on the program — are there instructions that use absolute addresses, and does the program use them? (For MIPS, some forms of jump and load/store instructions do.)
- Both nicely solved by defining "address space" abstraction and implementing with help from hardware (MMU). Also makes it easier to move processes around in memory. (Why would you want to?)
- We looked briefly at several schemes in which each process's memory is contiguous. Good fit for simple MMU but not very flexible. Can we do better? Yes . . .

Slide 3

Paging

- Idea — divide both address spaces and memory into fixed-size blocks ("pages" and "page frames"), allow non-contiguous allocation.
- Consider tradeoffs yet again — complexity versus flexibility, efficient use of memory.

Slide 4

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. (Notice 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".

Slide 5

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. Details later.)

Slide 6

Paging and Memory Protection

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

Slide 7

Sidebar: Memory Management Within Processes

- What if we don't know before the program starts how much memory it will want? with very old languages, maybe not an issue, but with more modern ones it is.

  I.e., we might want to manage memory within a process's "address space" (range of possible program/virtual addresses).

- Typical scheme involves
  - Fixed-size allocation for code and any static data.
  - Two variable-size pieces ("heap" and "stack") for dynamically allocated data.
  - Note that combined sizes of these pieces might be less than size of address space, maybe a lot less.

Slide 8

Page Table Entries

- Exactly what's in a page table entry depends partly on hardware.
- Required(?) fields are page frame number, present/absent bit.
- Optional but useful fields include bits that can be used to track usage ("referenced/modified"), bits indicating what access is allowed (e.g., read-only), etc.
- (Figure 3-11 in text.)

Slide 9

Page Sizes and Other Details

- How big to make pages? compare extreme cases (really big, really small).
- If you know how big addresses are, what does that tell you about (maximum) sizes of physical/virtual memory?
- How big are page tables . . .

Slide 10

Page Table Size — Example

- Given a page size of 64K (2^{16}), 64-bit addresses, and 4G (2^{32}) of main memory, at least how much space is required for a page table? Assume that you want to allow each process to have the maximum address space possible with 64-bit addresses, i.e., 2^{64} bytes.
- (Hints: How many entries? How much space for each one? and no, this is not a very realistic system.)

Slide 11

Page Table Size — Example Continued

- Number of entries is $2^{64}/2^{16}$, i.e., 2^{48}.
- Size of each entry — at least enough for page frame number. There are 2^{16} of them, so we need 16 bits for that. Probably should also include a valid/invalid bit, for a total of 17 bits. Rounding up to a multiple of 8 bits (one byte), that's 3 bytes per entry.
- Total space is $2^{48} \times 3$ — bigger than main memory!! so, not realistic.

Slide 12

Performance / Feasibility Concerns

- Even with good choice of page size, serious performance implications — page table can still be big, and every memory reference involves page-table access — how to make this feasible/fast?
- (Remember that the MMU is hardware, and a bit about registers — local to the CPU, faster to access than memory but limited in number, can be general-purpose or dedicated to a particular use (e.g., the program counter).)

Slide 13

Minute Essay

- To do its job the MMU must have access to the current process's page table. The textbook mentions two simple schemes for doing this:
  - Keep the entire table in (processor) registers.
  - Keep the 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 the MMU will be able to translate each address.)
- How did the midterm compare to your expectations? with regard to length, difficulty, etc.? If you didn't do well, why do you think that happened?

Slide 14

Minute Essay Answer

- The first scheme almost surely makes for faster translations, but for a large page table it will require a lot of registers, which would make context switches slow. The second scheme won't slow down context switches, but as stated it isn't going to make for fast translation.