Slide 1

Administrivia

- (None.)

Slide 2

Paging — Review/Recap

- Simple schemes for memory management, in which each process's address space is mapped to a single single contiguous block of physical memory, are simple but not very flexible. Paging is one way to do better.
- 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 3

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 4

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.
- (Compare to swapping. Details later.)

Slide 5

Paging and Memory Protection

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

Slide 6

Page Table Entries

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

Slide 7

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 8

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 9

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 10

Performance / Feasibility Concerns

- Clearly page tables can be impractically large. What to do?
- Also, every memory reference involves page-table access — how to make this feasible/fast?
- (To be continued . . .)

Slide 11

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.)

Slide 12

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.