Slide 1

Administrivia

- (None.)

Slide 2

Paging — Recap

- Recall basic ideas of paging:
  - Divide address spaces into pages, memory into page frames; allocate memory page (frame) by page (frame).
  - Use page tables (one per process) to keep track of things.
  - Use MMU to translate program (virtual) addresses into memory locations — using page table for current process. Generate "page fault" interrupt if impossible.
- Notice that we get memory protection for free; can also get memory sharing. Related issue — might be nice to have "read-only" bit in page table.
- Still some issues to address — performance, large tables, how to use this for virtual memory.

  (Review minute essay from last time.)

Slide 3

Performance / Large Address Spaces

- 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?
- Consider several options — compare access time, cost, context-switch time:
  - Keep page table for current process in registers.
  - Keep whole page table in main memory, pointed to by special register.
  - Use multilevel page tables. (More about this later.)
  - Use inverted page tables (one entry per page frame). (More about this later.)
- If page tables are in memory, performance improves with “translation lookaside buffer” (TLB) — special-purpose cache.

Slide 4

Large Address Spaces

- Clearly page tables can be big. How to make this feasible?
- One approach — multilevel page tables.
- Another approach — inverted page tables (one entry per page frame).

Slide 5

Paging and Virtual Memory

- Idea — if we don't have room for all pages of all processes in main memory, keep some on disk ("pretend we have more memory than we really do").
- Or a simpler view: All address spaces live in secondary memory / swap space / backing store, and we "page in" as needed (demand paging).
- Making this work requires help from both hardware (MMU) and software (operating system).

Slide 6

Processing Memory References — MMU

- Does cache contain data for (virtual) address? If so, done.
- Does TLB contain matching page table entry? If so, generate physical address and send to memory bus.
- Does page table entry (in memory) say page is present? If so, put PTE in TLB and as above.
- If page table entry says page not present, generate page fault interrupt. Transfers control to interrupt handler.

Slide 7

Processing Memory References — Page Fault Interrupt Handler

- Is page on disk or invalid (based on entry in process table, or other o/s data structure)? If invalid, error — terminate process.
- Is there a free page frame? If not, choose one to steal. If it needs to be saved to disk, start I/O to do that. Update process table, PTE, etc., for “victim” process. Block process until I/O done.
- Start I/O to bring needed page in from swap space (or zero out new page). If I/O needed, block process until done.
- Update process table, etc., for process that caused the page fault, and restart it at instruction that generated page fault.

Slide 8

Processing Memory References — Details Still To Fill In

- How to keep track of pages on disk.
- How to keep track of which page frames are free.
- How to “schedule I/O” (but that’s later).
- How to choose a page frame to “steal”.

Minute Essay

- We just mentioned the need to choose “a page frame to steal”. What are some of the things you think need to be considered in making this choice? (current contents? owning process? something else?)

  (We’ll talk about this next time.)

Slide 9