Slide 1

Administrivia

- (None?)

Slide 2

Minute Essay From Last Lecture

- Several right answers (one with a reference to textbook's explanation). Also see slide below.
- Many answers that seem a little confused:

  "Registers" are part of processor, not memory. Contents are part of "state" that must be saved/reloaded on context switch. But access to them is fast.

  Memory (RAM) isn't really involved in context switch but is slower to access than registers.

  Both (probably?) require the same amount of total space, just in different places.
- One person said keeping table in registers might be less safe. I'm skeptical — ?

Slide 3

Paging — Recap

- Idea — divide both address spaces and memory into fixed-size blocks (“pages” and “page frames”), allow non-contiguous allocation.
- Makes for a much more flexible system but at a cost in complexity — keeping track of a process’s memory requires a “page table” to be used by both hardware (MMU) and software (O/S).

Slide 4

Page Tables — Performance Issues

- One possibility is to keep the whole page table for the current process in registers. Could possibly use general-purpose registers for this but likely would not. Should make for fast translation of addresses, but — is this really feasible for a large table? and what about context switches?
- Another possibility is to keep the process table in memory and just have one register (probably a special-purpose one) point to it. Cost/benefit tradeoffs here seem like the opposite of the first scheme, no?

  The big downside is slow lookup, though, and that can be improved with a “translation lookaside buffer” (TLB) — special-purpose cache.

Slide 5

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 6

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).
- (Aside: Why are we even bothering? Can't the processor(s) access disk? Yes, but . . .)
- Making this work requires help from both hardware (MMU) and software (operating system).

Page Fault Interrupts

- We said MMU should generate a “page fault” interrupt for a page that’s not present in real memory. What happens then? It’s an interrupt, so . . .
- Control goes to an interrupt handler. What should it do? (Are there different possibilities for what caused the page faults?)

Slide 7

Page Fault Interrupts, Continued

- One possible cause — an address that’s not valid. You know (sort of) what happens then . . .
- Another cause — an address that’s valid, but the page is on disk rather than in real memory. So — do I/O to read it in. Where to put it? If there’s a free page frame, choice is easy. What if there’s not?

Slide 8

Slide 9

Finding A Free Frame — Page Replacement Algorithms

- Processing a page fault can involve finding a free page frame. Would be easy if the current set of processes aren't taking up all of main memory, but what if they are? Must steal a page frame from someone. How to choose one?
- Several ways to make choice (as with CPU scheduling) — "page replacement algorithms".
- "Good" algorithms are those that result in few page faults. (What happens if there are many page faults?)
- Choice usually constrained by what MMU provides (though that is influenced by what would help o/s designers).
- Many choices (no surprise, right?) . . .

Slide 10

"Optimal" Algorithm

- Idea — if we know for each page when it will next be referenced, choose the one for which that's the furthest away.
- Theoretically optimal, though can't be implemented.
- Useful as a standard of comparison — run program once on simulator to collect data on page references, again to determine performance with this "algorithm". (Not clear that this is really possible with multiprogramming, i.e., more than one process active.)

Slide 11

Sidebar: Page Table Entries, Revisited

- Recall — many architectures' page table entries contain bits called "R (referenced) bit" and "M (modified) bit".
- Idea is that these bits are set by hardware on any memory reference, and cleared by software (o/s) in some way that's useful.

Slide 12

"Not Recently Used" Algorithm

- Idea — choose a page that hasn't been referenced/modified recently, hoping it won't be referenced again soon.
- Implementation — use page table's R and M bits, group pages into four classes:
  - $R = 0$, $M = 0$.
  - $R = 0$, $M = 1$.
  - $R = 1$, $M = 0$.
  - $R = 1$, $M = 1$.

  Choose page to replace at random from first non-empty class.
- How good is this? Easy to understand, reasonably efficient to implement, often gives adequate performance.

“First In, First Out” Algorithm

- Idea — remove page that’s been there the longest.
- Implementation — keep a FIFO queue of pages in memory.
- How good is this? Easy to understand and implement, no MMU support needed, but could be very non-optimal.

Slide 13

“Second Chance” Algorithm

- Idea — modify FIFO algorithm so it only removes the oldest page if it looks inactive.
- Implementation — use page table’s R and M bits, also keep FIFO queue. Choose page from head of FIFO queue, *but* if its R bit is set, just clear R bit and put page back on queue.
- Variant — “clock” algorithm (same idea, keeps pages in a circular queue).
- How good is this? Easy to understand and implement, probably better than FIFO.

Slide 14

Slide 15

“Least Recently Used” (LRU) Algorithm

- Idea — replace least-recently-used page, on the theory that pages heavily used in the recent past will be heavily used in the near future. (Usually true).
- Implementation:
  - Full implementation requires keeping list of pages ordered by time of reference. Must update this list on every memory reference.
  - Only practical with special hardware — e.g.:

    Build 64-bit counter C, incremented after each instruction (or cycle).

    On every memory reference, store C’s value in PTE.

    To find LRU page, scan page table for smallest stored value of C.

    (Is 64 bits enough?)
- How good is this? Could be pretty good, but requires hardware we probably won’t have.

Slide 16

“Not Frequently Used” (NFU) Algorithm

- Idea — simulate LRU in software.
- Implementation:
  - Define a counter for each PTE. Periodically (“every clock-tick interrupt”) update counter for every PTE with R bit set.
  - Choose page with smallest counter.
- How good is this? Reasonable to implement, could be good, but counters track full history, which might not be a good predictor.

Slide 17

“Aging” Algorithm

- Idea — simulate LRU in software (like NFU), but give more weight to recent history.
- Implementation similar to NFU, but increment counters by shifting right and adding to *leftmost* bit — in effect, divide previous count by 2 and add bit for recent references.
- How good is this? Pretty good approximation to LRU, though a little crude, and limited by size of counter.

Slide 18

Sidebar: Working Sets

- Most programs exhibit “locality of reference”, so a process usually isn’t using all its pages.
- A process’s “working set” is the pages it’s using. Changes over time, with size a function of time and also of how far back we look.

Slide 19

“Working Set” Algorithm

- Idea — steal / replace page not in recent working set. Define working set by looking back τ time units (w.r.t. process's virtual time). Value of τ is a tuning parameter, to be set by o/s designer or sysadmin.
- Implementation:
  - For each entry in page table, keep track of time of last reference.
  - When we need to choose a page to replace, scan through page table and for each entry:

    If $R = 1$, update time of last reference.

    Compute time elapsed since last use. If more than τ, page can be replaced.
  - If we don't find a page to replace that way, pick the one with oldest time of last use. If a tie, pick at random.
- How good is this? Good, but could be slow.

Slide 20

“WSClock” Algorithm

- Idea — efficient-to-implement variation of previous algorithm, based on circular list of pages-in-memory for process. (Carr and Hennessy.)
- Implementation — like previous algorithm, but when we need to pick a page to replace, go around the circle and:
  - If $R = 1$, update time of last use. Compute time since last use.
  - If time since last use is more than τ and $M = 1$, schedule I/O to write this page out (so it can maybe be replaced next time — M bit will be cleared when I/O completes). No need to block yet, though.
  - If time since last use is more than τ and $M = 0$, replace this page.

  The idea is to go around the circle until we find a page to replace, then stop. (If we get all the way around the circle, we'll pick some page with $M = 0$.)
- How good is this? Makes good choices, practical to implement, apparently widely used in practice.

Slide 21

Minute Essay

- Another story from long ago: Once upon a time, a mainframe computer was running very slowly. The sysadmins were puzzled, until one of them noticed that one of the disk drives seemed to be very busy and asked “which disk are you using for paging?” The answer made everyone say “aha!” What was wrong (to make the system so slow)?
- Does anything like this still happen?

Slide 22

Minute Essay Answer

- The disk being used for paging was the one that was very busy. So, mostly likely the system was spending so much time paging (“thrashing”) that it wasn’t able to get anything else done. Usually this means that the system isn’t able to keep up with active processes’ demand for memory.
- Memory sizes have increased to a point where the odds aren’t as good as they were. But a few years ago we did run into problems with the machines in one of the classrooms trying to run both Eclipse and a Lewis simulation, and then more recently with some of them attempting to run a background program that asked for memory than its author intended.