Slide 1

Administrivia

- Homework 4 to be on Web later today. I'll send mail. Due next Wednesday (or later).

Slide 2

Paging and Virtual Memory, Review/Clarification

- "Main memory" / "real memory" — directly accessible by the CPU, contents lost on shutdown. With paging, conceptually divided into "page frames".
- "Address space" — process's view of "its" memory, similar to virtual CPU discussed earlier. Can be larger than real memory. With paging, conceptually divided into "pages".

  Process might not be using all of its address space, in which case some pages don't exist anywhere. Pages that *do* exist — are either in main memory, or on disk.
- "Swap space" — area on disk reserved for copies of pages we don't have room for in memory.

Slide 3

Review — Page Replacement Algorithms

- General idea is to minimize number of page faults while keeping cost of picking which page to replace reasonable.
- Many simple algorithms discussed. LRU and approximations seem most promising / practical.
- A few more to discuss, most based on idea that each process has a "working set" of pages that need to be in memory.

Slide 4

Sidebar: Demand Paging, Prepaging, and Working Sets

- The purest form of paging is "demand paging" — processes are started with no pages in memory, and pages are loaded into memory on demand only.
- An alternative is "prepaging" — try to load pages in advance of demand. How?
- 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 5

"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 6

"WSClock" Algorithm

- Idea — efficient-to-implement variation of previous algorithm, based on circular list of pages-in-memory for process.
- 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 7

Review — Page Replacement Algorithms

- Nice summary in textbook, table on p. 228.
- Author says best choices are aging, WSClock.

Slide 8

Modeling Page Replacement Algorithms

- Intuitively obvious that more memory leads to fewer page faults, right? Not always!
- Counterexample — “Belady’s anomaly”, sparked interest in modeling page replacement algorithms.
- Modeling based on simplified version of reality — one process only, known inputs. Can then record “reference string” of pages referenced.
- Given reference string, p.r.a., and number of page frames, we can calculate number of page faults.
- How is this useful? can compare different algorithms, and also determine if a given algorithm is a “stack algorithm” (more memory means fewer page faults).

Slide 9

Paging — Other Design Issues

- In deciding which page to replace, consider all pages ("global allocation"), or just those that belong to the current process ("local allocation")?

  Generally, global approach works better, but not all page replacement algorithms can work that way (e.g., WSClock). Hybrid strategy — combine local approach with some way to vary processes' allocations.

- What happens if combined working sets of all processes don't fit into memory? "Thrashing".

  What to do? temporarily "swap out" some processes, or other forms of "load control".

- Maintaining a supply of free frames — desirable, could do by having a "paging daemon" in background.

Slide 10

Paging — Other Hardware Issues

- What if page to be replaced is waiting for I/O? probably trouble if we replace it anyway.
- One solution — allow pages to be "locked".
- Another solution — do all I/O to o/s pages, then move to user pages.

Slide 11

Minute Essay

- If process p0's working set totals 128M, and process p1's working set totals 160M, and you have 256M of real memory, what will happen if you run p0 and p1 at the same time? (I.e., do they run well together?)

Slide 12

Minute Essay Answer

- Since the combined working sets of the two processes exceeds the size of main memory, the likely result of trying to run them at the same time is lots of paging, and thus poor performance. (We might have this problem even with slightly smaller working sets, since some of real memory needs to be reserved for the operating system itself.)