Slide 1

Administrivia

- Corrected version of sample solution for Homework 2 available in hardcopy. Ask me. Or I'll send a mass mail with corrections?
- Homework 3 coming soon, probably Monday, to be due in a week.

Slide 2

Page Replacement Algorithms, Continued

- Recall context — we want to move (or copy) a page from disk to memory, but all page frames in memory are in use. So we have to "steal" a page frame. How to choose?
- Other things being equal, the fewer page faults the better.
- We looked at several "page replacement algorithms" last time. More today. But first . . .

Slide 3

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 4

"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 5

"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 6

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 7

Page Replacement Algorithms — Recap

- Nice summary in textbook (table at end of section 3.4).
- Tanenbaum says best choices are aging, WSClock.
- Now move on to other issues to consider . . .

Slide 8

Global Versus Local Allocation

- 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.

Slide 9

Thrashing and Load Control

- What happens if combined working sets of all processes don't fit into memory? "Thrashing". (See minute essay from last time!)
- What to do? temporarily "swap out" some processes, or other forms of "load control".

Slide 10

Sharing Pages

- Shared pages can be useful, but can also present problems.
- Multiple processes running the same program is relatively easy (why?) but has one potential downside (what?)
- UNIX `fork` system call is — interesting in this context. POSIX definition says that child process's address space is basically a copy of the parent's address space. What's the easy-to-implement way to do this? What downside does that have in current systems? Is there a way to reduce its impact? And why duplicate in the first place?

Slide 11

Sharing Pages and `fork`

- Duplicating pages is easy but inefficient, especially if the child process is going to call `execve` or something similar right away. Some systems use "copy-on-write" to improve efficiency.
- Why did the people who designed UNIX require this duplication . . . Possibly because it makes some things easy (such as setting up parent/child pipes) and wasn't very costly when designed. Windows' system call for creating processes takes a different approach. Maybe that's better!

Slide 12

Minute Essay

- None — sign in.