CMPSCI 377 Operating Systems **Fall 2008**

Lecture 18: November 19

Lecturer: Prashant Shenoy *Scribe: Shashi Singh*

18.1 LRU Approximations

LRU evicts the page not used for the longest time. It approximates OPT whenever the recent past is a decent prediction of the future. Notice that variants of the LRU policy are currently used in *all* real operating systems.

What about evicting the **Most-Recently Used** page? For one thing, this does very well on LRU's worst case (loops that exceed RAM size), but in general MRU is not a good idea. Another possibility is to evict the oldest page; this is accomplished by a **First-in, First-out** (FIFO) policy. In theory FIFO is as competitive as LRU, but in practice it performs miserably, since it ignores temporal locality: suppose pages ABCAD are in memory; then, when hitting D, a FIFO algorithm would evict A!.

OK, so how do we actually implement a LRU? Idea *1)*: mark everything we touch with a timestamp. Whenever we need to evict a page, we select the oldest page (=least-recently used). It turns out that this simple idea is not so good. Why? Because for every memory load, we would have to read contents of the clock and perform a memory store! So it is clear that keeping timestamps would make the computer *at least* twice as slow. Idea *2)*: keep a queue; everytime we touch something, we move that something to the beginning of the queue. Whenever we need to evict a page, we evict the last element of the queue. Now, we don't need to use timestamps, but we still have to do some pointer manipulation. Notice however that trying to do that for every load or store operation is actually *even worst* than using timestamps! Idea *3)*: using a hash table. In this case, the problem is that on every memort access we have to compute a hash address. Bad idea. Idea *4)*: let's try to do something *close* to LRU by mainting *reference bits* for every page; on each memory access, we set the page's reference bit to 1. Now, a page replacement algorithm (usually implemented on hardware) periodically resets the reference bits. Whenever we need to evict a page, we select one that has a reference bit not set. This algorithm considers whatever is zeroed to be "old enough"; then, although we can't know exactly what is the *least*-recently used, we do know that whatever is marked is a least *more* recent than something not marked. In practice, this approach works pretty well.

One interesting idea is to keep a reference bit per page and to set it to one everytime we accessed that page. Also, all reference bits were to be periodically reset. This algorithm turned out to be a good approximation of LRU. Although it makes us capable of deciding which pages are "newer", it does not do it in a very precise way: both a page that hasn't been accessed in the last 10.000 years, and one that hasn't been accessed in the last minute, both have 0 as their "clock" bit (reference bit).

Now, let us discuss two other algorithms for deciding which pages to evict. The *clock algorithm* is one of the most popular choices; it works by keeping frames in circle. When a page fault occurs, it checks the reference bit of the *next frame*. If that bit is zero, it evicts that page and sets its bit to 1; if the reference bit is 1, the algorithm sets the bit to 0 and advances the pointer ot next frame. For more details, please refer to `http://en.wikipedia.org/wiki/Page_replacement_algorithm`.

Another possibility is to use a *segmented queue*. This type of algorithm divides all existing pages into two sets of pages. A third of all pages (the pages that are the most active) go through a clock algorithm. After that, all pages that are evicted out of the clock are moved to a "uncommon" list, in which we use exact

LRU. This way, we approximate LRU for the frequently-referenced pages (1/3 of the page frames - fast clock algorithm), and at the same time use exact LRU on the infrequently accessed pages (2/3 of all page frames).

18.1.1 Drawbacks

The question of *fairness*, regarding page eviction, is a hard one. How do we decide what is fair? We sure can put all pages from all processes into one large pool, and then manage it using a segmented queue. This approach is easy to implement, but has a considerable drawback. To understand it, suppose you have a computer running both minesweeper and a nuclear reaction control program. The nuclear program does not run very often, and thus is paged out constantly; even though it is "more important", since it is constantly paged out it always runs slower. Is that fair? Of course not. In general, the problem of fairness is that processes that are greedy (or wasteful) are rewarded; processes with poor locality end up squeezing out those with good locality, because they "steal" memory. There is no simple solution to this problem.

18.2 Replacement Policies for Multiprogramming

Thrashing: It occurs when the memory is over-committed and pages are continuously removed while they are still in use. Memory access times approach disk access times since many memory references cause page faults. It results in severe loss of performance. Following are the schemes used to limit thrashing in multiprogrammed environment -

- **Proportional allocation:** number of page frames allocated to a process is proportional to its size. This seems like a logical thing to do.
- **Global replacement:** A global replacement algorithm is free to select any page in memory (belonging to any process). It is flexible in the sense that it adjusts to divergent process needs. On the downside, thrashing might become even more likely because a process cometes with all other processes in the system.
- **Per-process replacement:** Each process has its own pool of pages. Only those processes that fit into memory are run. We need to figure out how many pages a process needs, i.e. its working set size. But working sets are expensive to compute. Its easier to track the page fault frequency of each process instead. If the page fault frequency of a process is greater than some threshold, it is assigned more page frames. If the frequency is less than a second-threshold, we take away some page frames assigned to it. The goal is to make the system-wide mean time between page faults equal to the time it takes to handle a page fault. If the former is smaller than the latter, page fault operations will start queuing up. Thrashing is less likely to happen as a process only competes with itself. This gives more consistent performance independent of system load. On the downside, figuring out how many pages to assign to a process is a non-trivial task.