CMPSCI 377 Operating Systems **Spring 2010**

Lecture 17: March 25

Lecturer: Prashant Shenoy *TA: Vimal Mathew & Tim Wood*

17.1 Memory and Page Replacement

One would expect that adding memory to a system should always reduce the number of page faults seen within the system. In fact, the benefit of adding more memory depends on the page replacement algorithm being used. Adding additional memory to a system that uses a FIFO page replacement policy may or may not provide any benefit depending on the nature of page accesses. In contrast, adding more memory to a Least Recently Used (LRU) based system will always provide equal or better performance than a system with less memory. This is because LRU more intelligently chooses how pages are evicted, and as long as a process exhibits some locality in its memory accesses, it will see improved performance from increased memory.

17.1.1 Perfect LRU

The goal of LRU is to evict the page not used for the longest time. In order to perfectly achieve this goal, it must maintain a time stamp for every page that indicates the time of the last access (requiring both extra bits in the page table and an additional memory operating for each access). When deciding which page to evict, the OS must scan every single page in memory in order to find the one with the oldest time stamp. As a result, maintaining time stamps is expensive and slows down both individual memory accesses and page evictions. Rather than keep extra time stamp bits, it would also be possible to implement LRU by just maintaing pages in an ordered list. Whenever a page is accessed it is moved to the head of the list; when a page must be evicted, the last one in the list is chosen. However, this can still be overly expensive because it requires multiple pointer modifications per access in order to keep the list in the right order. The high cost of these techniques suggest that implementing perfect LRU may be too expensive and that it would be better to design an algorithm that approximates LRU.

17.1.2 LRU Approximations

Variants of the LRU algorithm are used in all modern operating systems because it can approximate the performance of an optimal page replacement scheme. However, in order to make LRU run efficiently, it must be modified. Most LRU approximations take advantage of **reference bits** provided by the hardware. These are one or more bits included with each page table entry in the system. When a page is accessed, the hardware automatically sets the reference bit to 1. By periodically clearing the reference bits and then examining which ones are reset to 1, the OS can get a sense of which pages have been used recently and which have not.

One way to approximate LRU is the **Additional-Reference-Bits** technique that maintains several reference bits. On each page access, the highest order reference bit is set to 1. At regular intervals, the system will shift all of the reference bits one space to the right, inserting a 0 as the new highest order bit. When a page must be evicted from memory, the system can scan through the pages to find the one with the lowest value in its reference bits. For example, a page which has been modified during every interval might have reference bits 1111, while one that was modified every interval except the most recent would have 0111. The second page can thus easily be distinguished as having been accessed the least recently. This technique is more efficient than maintaing full timestamps for every memory access, but it does not provide a perfect total ordering of pages—they are grouped and ordered by whether they have been accessed in recent intervals. Unfortunately, this approach still requires that all pages be scanned before a page is evicted in order to find the page with the lowest count in its reference bits.

The **Second Chance** or **Clock** algorithm is another LRU approximation that is closer to what is used in most modern operating systems. In the Clock algorithm, only a single reference bit is required per page. The OS maintains all memory frames in circular list (imagined as a clock), and it keeps a pointer that moves around this list (the clock's hand). When a page fault occurs, Clock 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 to the next frame. The result of this algorithm is that a page is only evicted if its reference bit is set to zero and the "clock hand" travels fully around the clock to it again before the page is accessed. If the page has been accessed in the intervening time, it is given a "second chance" since its reference bit will have been reset to 1.

The Clock algorithm is less accurate at determining how recently a page has been used compared to the time stamp and multiple reference bit approaches, but it can be implemented very efficiently since only a single bit needs to be updated with each memory access and there is no shifting required. In addition, page faults are faster since the clock hand only needs to cycle through the list of pages until it finds the first 0. However, in the worst case the clock algorithm may still need to cycle through all pages before it will return to the original page it started at in order to find a '0' page to evict.

The Clock algorithm works by partitioning pages into either a young or old category. The **Enhanced Second Chance** algorithm extends this idea to include additional categories in an attempt to more intelligently pick which pages to evict. In particular, it is desirable to evict pages which have not been written to, e.g. code pages that are never modified or data pages which were swapped out and back from disk previously and have not been modified since. For these pages, it is not actually necessary to write the page out to disk because a copy of it already exists there!

In order to tell which pages have been modified, the Enhanced Second Chance algorithm keeps a "modified" bit along with the reference bit. When set, this bit indicates that the page has been modified, and thus is different from any copy kept on disk. If the bit is zero, then the page in memory is identical to one on disk, and will not need to be written out if evicted. The combination of reference and modified bits (r, m) can then be used to determine which page to evict. For example, a value of $(1, 1)$ means a page has both been recently used and is different than the version kept on disk, making it a poor choice for eviction. A page with $(0, 0)$ has both not been used recently and is identical to one on disk—an ideal candidate for eviction.

Attempting to find the page in the best eviction class can actually take multiple sweeps of the clock. During the first pass, pages with bits $(0, 1)$ are locked and scheduled to be written to disk, but are not yet evicted from memory. By the second pass, if one of the $(0, 1)$ pages has completed being written to disk, its value will have changed to $(0, 0)$ and it will be selected for eviction. In the worst case where all pages initially had values of $(1, 1)$, by the third iteration all pages will have been reset to $(0, 0)$ and the first will be selected for eviction.

17.1.3 Fairness

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.

17.2 Replacement Policies for Multiprogramming

Multiprogramming is when the resources of a system are divided up so that many processes run simultaneously. Unfortunately, the interactions between processes can lead to poor performance if the page replacement policies are not designed carefully. **Thrashing** occurs when the memory is over-committed and pages are continuously swapped to disk while they are still in use. As a result, memory access times approach disk access times since many memory references cause page faults. It results in a severe loss of performance. The following are some schemes used to limit thrashing in multiprogrammed environment -

Proportional allocation: makes the number of page frames allocated to a process proportional to its size. Typically, all processes would be given an equal portion of memory frames, but this can lead to poor performance if some applications need more memory or are more important than others. By letting larger applications keep a proportionately larger amount of pages in memory, the chance of thrashing can be reduced. Generally not used in practice.

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 changing process needs. On the downside, thrashing might become even more likely because a single process can cause system wide thrashing by disrupting the global pool. As a result, it is not used in many operating systems.

Per-process replacement: Each process has its own pool of pages, and an LRU queue is maintained for each process. Only those processes that fit into memory are run. However, we need to figure out how many pages each process needs, i.e. its working set size—the set of pages it is actively using at any time. This can be tracked by counting which pages have been accessed over a historical time window. Over time, the pages in the working set, as well as the size of the working set, may change.

Unfortunately, working sets are difficult and expensive to compute. It is often easier to instead track the page fault frequency of each process. If the page fault frequency of a process is greater than some threshold, it must have fewer pages allocated than its working set, so it is assigned more page frames. If the frequency is less than a second-threshold, we take away some of the 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—otherwise page fault operations will start queuing up, leading to trashing.

With per-process replacement, thrashing is less likely to happen because each 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.