CS5600 – Computer Systems – October 13, 2010

Page replacement

In order to maintain reasonable performance, the number of memory accesses that result in a page fault must remain fairly low, and the cost of moving a page into memory must be kept as low as possible.

Various techniques are used to pick which frame will be **evicted** to make room for a new page. The page that is being removed is typically referred to as the **victim**.

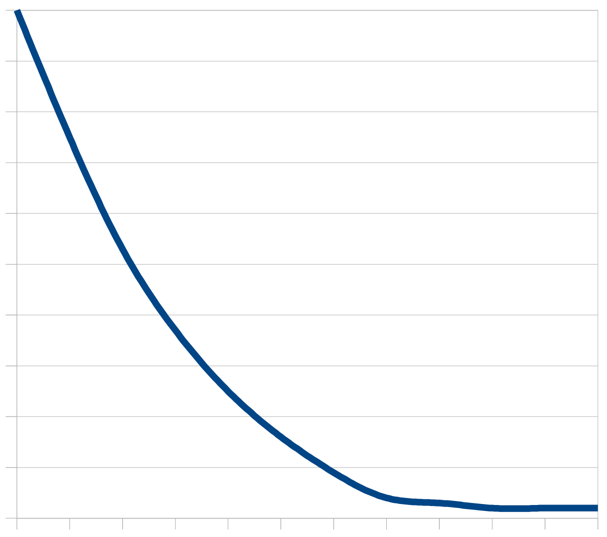
One of the techniques is to keep a **dirty bit** in the page table, which is set whenever a page has been modified in memory. Pages that are **clean** (ie, do not have the dirty bit set) are already stored in the backing store and do not need to be written to disk when they are replaced, so they are better candidates for eviction.

There are other heuristics or algorithms that can be used to pick victim pages; how should we evaluate them? We need two pieces of data to evaluate different choices:

1) The number of physical frames that are available.
2) A stream/series of memory accesses, in terms of virtual pages. (Note that you only need the page number, not the full virtual address, because the virtual memory system operates on full pages only.)

With this information, we can simulate the operation of a particular **page replacement algorithm.**

Typically, the relationship between page fault rate (y-axis) and the number of frames (x-axis) looks something like this:



The intuition behind this is that when a small number of frames are available, adding just a few more makes a big difference in the number of page faults, but when you already have many of them, adding just a few frames makes little difference in the overall page fault rate.

Let's consider an example:

3 frames
Accesses: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5, 1

A simple approach would be FIFO (first in, first out) – when a page must be evicted, choose the first one that was loaded into memory.

Rows represent the content of the frames, each column is an access of the indicated page.

	1	2	3	4	1	2	5	1	2	3	4	5	1
1	1	1	1	4	4	4	5	5	5	5	5	5	1
2		2	2	2	1	1	1	1	1	3	3	3	3
3			3	3	3	2	2	2	2	2	4	4	4
fault	X	X	X	X	X	X	X			X	X		X

This sequence produced 10 page faults in 13 accesses.

Some of these faults are necessary; for instance, a fault must occur the first time any logical page is accessed. But this algorithm is not optimal; for instance, in the fifth access (when page 1 is accessed for the second time), we would have been better off if page one had not been evicted in the previous step.

Let's try this again with four frames:

	1	2	3	4	1	2	5	1	2	3	4	5	1
1	1	1	1	1	1	1	5	5	5	5	4	4	4
2		2	2	2	2	2	2	1	1	1	1	5	5
3			3	3	3	3	3	3	2	2	2	2	1
4				4	4	4	4	4	4	3	3	3	3
fault	X	X	X	X			X	X	X	X	X	X	X

This actually produced **11** page faults! The number of page faults actually increased when an additional frame was provided.

This is due to limitations of the FIFO replacement algorithm. So let's try something else, like LRU (**least recently used**) – replace the frame that was least recently accessed.

	1	2	3	4	1	2	5	1	2	3	4	5	1
1	1	1	1	1	1	1	1	1	1	1	1	5	5
2		2	2	2	2	2	2	2	2	2	2	2	1
3			3	3	3	3	5	5	5	5	4	4	4
4				4	4	4	4	4	4	3	3	3	3
fault	X	X	X	X			X			X	X	X	X

This only produced 9 faults, which is a definite improvement.

This leads to a question: is 9 "good"? What is the best we could do?

The optimal algorithm is actually **farthest in future** – evict the item that has the most time before its next access.

	1	2	3	4	1	2	5	1	2	3	4	5	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1
2		2	2	2	2	2	2	2	2	2	4	4	4
3			3	3	3	3	3	3	3	3	3	3	3
4				4	4	4	5	5	5	5	5	5	5
fault	X	X	X	X			X				X		

This only produced six faults, which is optimal. Of course, in the real world, you can't tell what accesses are coming, so you cannot actually implement a farthest-in-future algorithm. LRU tends to be used in practice, and works pretty well. If you think about trying to predict the future, assuming that the least recently used item will not be needed again for a while is often reasonable.

So what if we want to use LRU in practice? One simple approach would be to have the kernel keep track of memory accesses, and context-switch into the kernel during every page fault to look up which page should be evicted. However, this is likely to be too slow in practice, and would require too much kernel memory. So how about doing something in hardware? Could we add something to the page table to track when a page was used, that the MMU could update for us?

It's probably too expensive to keep an actual counter or timestamp in that table, but it would be possible to add just one or two bits to each table entry. In practice, it's usually one bit per page, called the **reference bit**. So how would we use this? During a timer interrupt, the kernel could examine which reference bits are set, then clear them all. This would allow you to see which pages are accessed by the process during each timer tick. However, it may be too expensive to keep much history about these accesses, since there are many physical pages.

Consider treating the pages in memory as a circular linked list, where we keep a pointer to some particular page. Now, if we just evict out the pointed-to page each time, this implements a FIFO algorithm: pages will be evicted in the order they were inserted in the list. If you augment this by not removing pages where the reference bit is on (and also clearing the bit when you skip a page like this), this implements a **second chance** algorithm – pages are only evicted if they are chosen for replacement twice since the last time they were accessed. Frequently-used pages will quickly have the bit reset after it is cleared, so they are much less likely to be evicted than infrequently-used ones.