Page Replacement Algorithms

This section is about a collection of algorithms used to determine when to kick a page out of memory and swap it to the hard disk. Many of the algorithms here work for any type of cache, not just for paging.

Optimum

Rule: Kick out the page that won't be needed again until the farthest in the future.

Example: We'll assume the cache has capacity to hold 3 pages. Let's suppose the pages are accessed in this order: 0, 1, 2, 0, 1, 3, 0, 3, 1, 2, 1, 2, 3, 0, 1, 2, 3. The table lists these pages, whether they are in the cache or not when they need to be accessed (a hit or a miss), and what the cache looks like after that page access.

page	hit/miss	new cache	comments
0	miss	0	cache is empty to start, so we start with "cold start" misses
1	miss	0,1	
2	miss	0,1,2	
0	hit	0,1,2	
1	hit	0,1,2	
3	miss	0,1,3	kick out 2 because 0, 1, and 3 will be needed sooner
0	hit	0,1,3	
3	hit	0,1,3	
1	hit	0,1,3	
2	miss	1,2,3	kick out 0 because 1, 2, and 3 will be needed sooner
1	hit	1,2,3	
2	hit	1,2,3	
3	hit	1,2,3	
0	miss	1,2,3	don't kick out anything because 0 won't be needed again
1	hit	1,2,3	
2	hit	1,2,3	
3	hit	1,2,3	

Pros: It's mathematically provable that this is the best algorithm in terms of hit/miss percentage.

Cons: To use it, we have to know the future, which is not practical in real systems.

Comments: The Optimum algorithm is useful in simulations of real systems where we do know the future. We can use it as a benchmark to compare other algorithms to see how close to optimum they get.

FIFO

Rule: It's first-in, first out, so the page that has been in the cache the longest is the one kicked out.

Here is an example. Assume the cache has capacity 3.

page	hit/miss	new cache	comments
0	miss	0	cache is empty to start, so we start with "cold start" misses
1	miss	0,1	
2	miss	0,1,2	
0	hit	0,1,2	
1	hit	0,1,2	
3	miss	1,2,3	kick out 0 because it's the oldest
0	miss	2,3,0	kick out 1 because it's the oldest
3	hit	2,3,0	
1	miss	3,0,1	kick out 2 because it's the oldest
2	miss	0,1,2	kick out 3 because its the oldest
1	hit	0,1,2	

Pros: It's fast and easy to implement.

Cons: There is no consideration for how important a page is. Pages that are used a lot can get kicked out in favor of rarely used pages.

There are also certain edge cases that FIFO is really bad with. One of them is the so-called looping workload. Suppose we have a cache that holds 3 pages, and the pages are accessed in the order 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3, etc. The accesses keep looping in the order 0, 1, 2, 3, and whenever a page is needed, it ends up having just been kicked out of the cache right before. So every access ends up being a miss. This is problematic because looping workloads like this are common.

FIFO is also subject to something called Béllády's anomaly, where for certain sequences of page accesses a larger cache can actually have a worse hit/miss percentage than a smaller cache.[1]

Random

Rule: When it comes time to kick out a page, pick a random page to kick out.

Pros: It's almost as fast and easy to implement as FIFO, and it doesn't suffer from edge cases like FIFO does. We would have to be really unlucky for a sequence of page accesses to result in all misses.

Cons: Just like with FIFO, a heavily used page has an equal chance of being kicked out as a rarely used page.

LRU

Rule: Kick out the page whose most recent access was the longest ago.

LRU stands for *least recently used*. The idea is that pages that were accessed recently are likely to be needed again soon, while pages that haven't been accessed in a while are likely not going to be needed for a while. Here is an example. Assume the cache size is 3.

page	hit/miss	new cache	comments
0	miss	0	cache is empty to start, so we start with "cold start" misses
1	miss	0,1	
2	miss	0,1,2	
0	hit	0,1,2	
1	hit	0,1,2	
3	miss	0,1,3	kick out 2 because 0 and 1 have been accessed more recently than 2
0	hit	0,1,3	
3	hit	0,1,3	
1	hit	0,1,3	
2	miss	2,1,3	kick out 0 because 1 and 3 have been accessed more recently than 0
1	hit	2,1,3	

Pros: Its hit/miss percentage is pretty good.

Cons: It's difficult to implement efficiently. It takes some effort keep track of when everything was last accessed and to find the right thing to kick out. Often this is more work than it's worth. The clock algorithm, covered below, gives a reasonably good approximation to LRU with good performance.

Clock algorithm

LRU has one of the best hit/miss percentages of any common algorithm, but it implementing it, especially in hardware, is a pain. The Clock algorithm gives a simple and effective approximation of LRU.

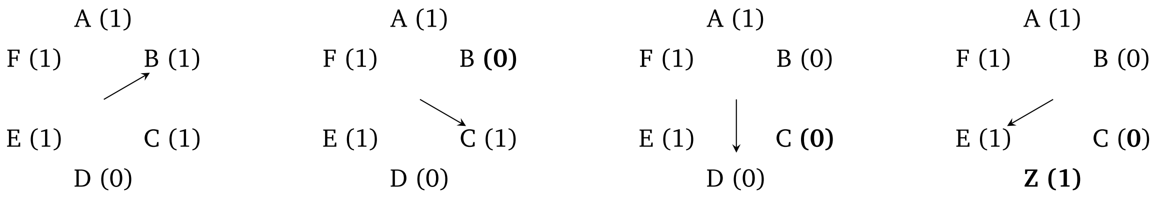
[1] An example is 3, 2, 1, 0, 3, 2, 4, 3, 2, 1, 0, 4 with cache sizes 3 and 4.

Imagine that the pages are arranged around a circle like the numbers on a clock. Along with each page there is the value of its R-bit, which indicates if it has been recently referred to by a process. We don't want to kick out pages that have been recently referenced because they are likely to be used again soon. There is a clock hand that goes around this clock, pointing at various pages.

If it comes time to kick a page out, we do the following:

- If the hand is currently pointing at a page whose R-bit is 1, set the R-bit to 0, move the clock hand forward, and continue.
- If the hand is currently pointing at a page whose R-bit is 0, kick that page out. We will replace it with the new page. The R-bit of the new page is set to 1 and the clock hand is advanced past it.

Here is an example. Assume the clock hand is currently pointing at B, like below on the left. Since B's R-bit is 1, we set it to 0 and move the clock hand to C. The R-bit of C is also 1, so we set it to 0 and move the clock hand forward to D. The R-bit here is 0, so D is the page that is kicked out. We replace it with the new page (which we'll call Z), set its R-bit to 1, and move the clock hand.



The "clock" will stay like this until it comes time to kick something else out. In the interim, some of the R-bits may get set to 1 as some of the pages are used by various processes.

The Clock algorithm is a variation of something called the Second Chance algorithm. The reason for the name is that if the hand is pointing at a page, it gets a second chance before being kicked out if its R-bit is 1. It's a reasonably good approximation to LRU that involves a lot less work.

Pros: It has almost as good a hit/miss percentage as LRU and it can be implemented efficiently.

Cons: Not many. It's not quite as simple to implement as FIFO or random, but it's still pretty easy to implement.

Aging

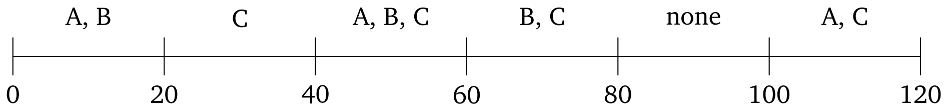
Related to the idea of LRU is LFU (*least frequently used*). The idea is to keep a count for each page of how often it is used and kick out the one that has been used the least. One drawback of this approach is pages that were used a lot a long time ago but not recently might be kept in the cache instead of more recent pages. A fix to this is to restrict the count to only keep track of what has happened in the recent past. The Aging algorithm is a way to do this efficiently.

In the Aging algorithm, every so many milliseconds we record the value of the R-bit for each page. Remember that each time a process uses a page, that page's R-bit will be set to 1. We keep track of these in a counter that has a structure like below:

current R-bit	previous R-bit	R-bit two steps ago	R-bit three steps ago

We have used 4-bit counters here, though in theory any size counter could be used, depending on how far back we want to keep track of. With each new time interval, we shift all the bits over 1 place to the right, and the left bit is updated with a 0 or 1, depending on whether or not the page was accessed in that interval. The bit all the way on the right will fall off the end and be lost.

Here is a graphical depiction of the process. In the figure below, time runs from 0 to 120 ms, and every 20 ms we check the value of the R-bit. Above the timeline are the names of the pages that are accessed in each interval. Below are how the counters change at each step, assuming they all start at 0000.



A	1000	0100	1010	0101	0010	1001
B	1000	0100	1010	1101	0110	0011
C	0000	1000	1100	1110	0111	1011

Here is an explanation of the first few steps: in the 0 to 20 interval, both A and B are accessed, so the leftmost bits of their counters are set to 1. In the 20 to 40 interval, everything shifts right one bit, so the two 1s from the previous step move into the second spot. In this interval, only C is accessed, so we set the leftmost bit of C to 1. In the 40 to 60 interval, we again shift everything 1 spot to the right from the previous step and all of the left bits are set because all three pages are accessed.

When deciding which page to kick out, the rule is simple: treat the counters as binary numbers and kick out the page with the smallest counter. For instance, in the last step above, the counters are 1001, 0011, and 1011. The smallest of these is 0011 (for page B), so that's the page that is kicked out. The aging algorithm is a nice approximation of the LRU algorithm that is much more efficient to implement. It is one of the more widely used page replacement algorithms.

WS Clock algorithm

The WS Clock algorithm is a widely used combination of the Clock algorithm and another algorithm called the WS algorithm (Working Set algorithm). The details of the WS algorithm are a little complex, but the main idea is that it tries to keep track of which pages are in a process's working set. It approximates the working set as being those pages used in the last so many milliseconds. For each page, it keeps track of when it was last accessed, and things in the working set are those whose last access was not too long ago. When choosing a page to kick out, it tries to only kick out things not in the working set.

In addition to the R-bit used by the Clock algorithm, the WS Clock algorithm also uses the M-bit. This bit indicates if the page has been modified since being read in from disk. Remember that to kick out a page, we have to save it on the hard drive. If a page had been previously stored on the hard drive and then copied into memory, if it hasn't been modified since then, then the version on disk is still current. Therefore, there's no slow hard drive write needed when kicking the page out. So it is preferable to kick out pages whose M-bits are 0 rather than those whose M-bits are 1. The WS clock algorithm uses a clock in a similar way to the regular Clock algorithm. The decision-tree is a little more complicated. Here is pseudocode describing what it does:

```
if R-bit == 1:
    set R-bit to 0 and advance the clock hand
else if R-bit == 0:
    if page is in working set:
        advance the clock hand
    else if page is not in working set:
        if M-bit == 1:
            advance the clock hand and schedule a disk write to write the page to disk
        else if M-bit == 0:
            replace the page with the new page and advance the clock hand
```

In short, we first check the R-bit just like with the regular clock algorithm. If the R-bit is 0, then we move on to check if the page is in the working set. If the page is not in the working set, then we finally check the M-bit. We try to avoid kicking out pages whose R-bit is 1, pages in the working set, and pages whose M-bit is 1. The reason for the disk write if the M-bit is 1 is that if the page is no longer in the working set, then the page is a good candidate to kick out in the future. The disk write will eventually happen in the background, and while we're waiting we move the hand forward and look for a better page to kick out. As the hand moves around, it will hopefully find something to kick out, but it might not. If one of the disk writes completes, then we will have something to kick out. But if there were no disk writes scheduled, then we just have to pick something to kick out, such as the current page.