Page Replacement Algorithms

To implement demand paging, there is a need to have an elaborated replacement policy supported by the hardware and software. There are many different page-replacement algorithms. Every operating system probably has its own replacement scheme. Designers try to implement the one to have lowest page-fault rate.

We evaluate an algorithm by running it on a particular string of memory references and computing the number of page faults. The string of memory references (memory addresses in binary or hexadecimal form) is called a reference string. We can generate reference strings artificially (by using a random-number generator, for example), or we can trace a given system and record the address of each memory reference. To reduce the large reference string data, we can easily convert reference string into pages. First, for a given page size we need to consider only the page number, rather than the entire address. If we have a reference to a page p, then any references to page 'P' that immediately follow will never cause a page fault. Page 'P' will be in memory after the first reference, so the immediately following references will not fault.

To determine the number of page faults for a particular reference string and page-replacement algorithm, we also need to know the number of page frames available. Obviously, as the number of frames available increases, the number of page faults decreases. To describe several page replacement algorithms, we will use following page string (trace) to represent reference string and a memory of 3 frames.

Page Trace: 2, 3, 2, 1, 5, 2, 4, 5, 3, 2, 5, 2

FIFO Page Replacement Algorithm

First-In-First-Out (FIFO) algorithm is the simplest page-replacement algorithm. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen. No need to record the time when a page is brought in, we can create FIFO queue to hold all pages in memory. When a page is brought into memory, we insert it at the tail of the queue. We replace the page at the head of the queue.

For our example page trace, initially all three frames are empty. Figure-1 depicts 3 page/frame queue, where page fault is marked as 'x' and page replacement marked in red color. The first two references (2, 3) cause page faults and are brought into empty frames. Next reference (2) is already in memory, we have no page fault. Next reference (1) cause page fault and is brought into empty frame. Next reference (5) cause page fault, no free frame, so we replace page 2, because page 2 was brought in first as compared other two pages (3, 1). Next reference (2) cause page fault, no free frame, so we replace page 3. Next reference (4) cause page fault, no free frame, so we replace page 1. Reference (5) already in memory, no page fault. Next reference (3) cause page fault, no free frame, so we replace page 5. Reference (2) already in memory, no page fault. Next reference (5) cause page fault, no free frame, so we replace page 2. Next reference (2) cause page fault, no free frame, so we replace page 4.

The FIFO page-replacement algorithm is easy to understand and implement. However, its performance is not always good. If we select for replacement a page that is in active use, everything still works correctly. After we replace an active page with a new one,

Page Trace	2	3	2	1	5	2	4	5	3	2	5	2
				1	5	2	4	4	3	3	5	2
		3	3	3	1	5	2	2	4	4	3	5
	2	2	2	2	3	1	5	5	2	2	4	3
Page Fault	x	x	-	x	x	x	x	-	x	-	x	x

Figure-1: FIFO Page Replacement

a fault occurs almost immediately to retrieve the active page. Some other page must be replaced to bring the active page back into memory. Thus, a bad replacement choice increases the page-fault rate and slows process execution. To illustrate the problems with a FIFO page-replacement algorithm, consider the following page trace:

Page Trace: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

We can find page faults for this page trace with 3 frames as shown in Figure-2

.

Page Trace	1	2	3	4	1	2	5	1	2	3	4	5
			3	4	1	2	5	5	5	3	4	4
		2	2	3	4	1	2	2	2	5	3	3
	1	1	1	2	3	4	1	1	1	2	5	5
Page Fault	x	x	x	x	x	x	x	-	-	x	x	-

Figure-2: FIFO Page Replacement with 3 frames

With 3 frames, 9 page faults occur. If we increase one frame and calculate page faults, with four frames, page faults should be decreased. Figure-3 depicts that page faults increased with increase in frame.

Page Trace	1	2	3	4	1	2	5	1	2	3	4	5
			3	4	4	4	5	1	2	3	4	5
			3	3	3	3	4	5	1	2	3	4
		2	2	2	2	2	3	4	5	1	2	3
	1	1	1	1	1	1	2	3	4	5	1	1
Page Fault	x	x	x	x	-	-	x	x	x	x	x	x

Figure-3: FIFO Page Replacement with 4 frames

With 4 frames, ten page faults occur. This most unexpected result is known as *Belady's anomaly*: for some page-replacement algorithms, the page-fault rate may increase as the number of allocated frames increases. Discovery of Belady's anomaly lead the search for an optimal page-replacement algorithm - the algorithm that has the lowest page-fault rate of all algorithms and will never suffer from Belady's anomaly.

Optimal Page Replacement Algorithm

Discovery of Belady's anomaly was the search for an optimal page-replacement algorithm - the algorithm that has the lowest page-fault rate of all algorithms and will never suffer from Belady's anomaly. Such an algorithm does exist and has been called OPT or MIN. It is simply this: *Replace the page that will not be used for the longest period of time.*

Use of this page-replacement algorithm guarantees the lowest possible page fault rate for a fixed number of frames. Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the memory references (page trace). However, the optimal algorithm is used mainly for comparison studies. For instance, a new algorithm is within 13.5 percent of optimal at worst and within 5.4 percent on average.
We can use our sample page trace to calculate page faults with 3 frames to clarify the working of optimal page replacement as shown in Figure-10. In FIFO, we look for when a page was brough in memory, in optimal we look forward when a page will be used as shown in Figure-4. Number of page faults in this case is 6.

Page Trace	2	3	2	1	5	2	4	5	3	2	5	2
				1	5	5	4	4	4	2	2	2
		3	3	3	3	3	5	5	5	4	4	4
	2	2	2	2	2	2	3	3	3	5	5	5
Page Fault	x	x	-	x	x	-	x	-	-	x	-	-

Figure-4: Optimal Page Replacement

Approximation of Optimal Page Replacement Algorithm
Since optimal algorithm is not feasible, perhaps an approximation of the optimal algorithm is possible. Optimal algorithm uses the time when a page is to be used. If we use the recent past as an approximation of the near future, then we can replace the page that has not been used for the longest period of time. This approach is the Least Recently Used (LRU) Algorithm. LRU replacement associates with each page the time of that page's last use. When a page must be replaced, LRU chooses the page that has not been used for the longest period of time.

LRU approach is good and often is used as a page-replacement algorithm. The main problem is how to implement LRU replacement. An LRU page-replacement algorithm may require substantial hardware assistance. The problem is to determine an order for the frames defined by the time of last use. Counters and stack implementations are feasible:
Counters: Associate with each page-table entry a time-of-use field and add to the CPU a logical clock or counter. The clock is incremented for every memory reference. Whenever a reference to a page is made, the contents of the clock register are copied to the time-of-use field in the page-table entry for that page. In this way, we always have the "time" of the last reference to each page. We replace the page with the smallest time value. This scheme requires a search of the page table to find the LRU page and a write to memory (to the time-of-use field in the page table) for each memory access. The times must also be maintained when page tables are changed (due to CPU scheduling). Overflow of the counter clock must be considered.
Stack: Keep a stack of page numbers. Whenever a page is referenced, it is removed from the stack and put on the top. In this way, the most recently used page is always at the top of the stack and the least recently used page is always at the bottom. Because entries must be removed from the middle of the stack, it is best to implement this approach by using a doubly linked list with a head pointer and a tail pointer. Removing a page and putting it on the top of the stack then requires changing six pointers at worst. Each update is a little more expensive, but there is no search for a replacement; the tail pointer points to the bottom of the stack, which is the LRU page. This approach is particularly appropriate for software or microcode implementations of LRU replacement.

Like optimal replacement, LRU replacement does not suffer from Belady's anomaly. Both belong to a class of page-replacement algorithms, called stack algorithms, that can never exhibit Belady's anomaly. With n frames, for LRU replacement, the set of pages in memory would be the n most recently referenced pages. If the number of frames is increased, these n pages will still be the most recently referenced and so will still be in memory. Using a stack (special stack, deletion of entry is performed at the bottom of stack) for LRU replacement policy, we can find number of page faults by using example page trace as shown in Figure-5.

Page Trace	2	3	2	1	5	2	4	5	3	2	5	2
	2	3	2	1	5	2	4	5	3	2	5	2
		2	3	2	1	5	2	4	5	3	2	5
				3	2	1	5	2	4	5	3	3
Page Fault	x	x	-	x	x	-	x	-	x	x	-	

Figure-5: Stack based LRU page replacement.

For our example page trace, initially all three frames are empty. The first two references (2, 3) cause page faults and are brought into these empty frames. Next reference (2) is already in memory, we update stack, by placing page 2 at the top of the stack. no page fault. Next reference (1) cause page fault and is brought into empty frame. Next reference (5) cause page fault, no free frame, so we replace page 3, which is at the bottom of the stack. Next reference (2) is already in memory, so we update the stack for this reference by placing page (2) at the top of the stack. Next reference (4) cause page fault, no free frame, so we replace page 1. Reference (5) is already in memory, so we update the stack for this reference by placing page (5) at the top of the stack. Next reference (3) cause page fault, no free frame, so we replace page 2, which is at the bottom of stack. Reference (2) also cause page fault, no free frame, so we replace page 4. Next reference (5) is already in memory, so we update the stack for this reference by placing page (5) at the top of the stack. Next reference (2) is already in memory, so we update the stack for this reference by placing page (2) at the top of the stack. Page faults for LRU approach using stack is 6, whereas page faults for the same page trace with optimal is 7. In this case LRU is a good approximation.

Neither counter, nor stack implementation of LRU would be imaginable without hardware assistance beyond the standard TLB registers. The updating of stack or the counter fields must be done for every memory reference. If we use an interrupt for every reference to allow software to update such data structures, it would slow every memory reference by a factor of at least ten, which results in slowing every user process by a factor of ten. Overhead for memory management to that level is not tolerable.

LRU-Approximation Page Replacement

A few systems provide sufficient hardware support for true LRU page replacement. Some systems provide no hardware support, and other page-replacement algorithms (such as a FIFO algorithm) must be used. Many systems provide some help, however, in the form of a reference bit. The reference bit for a page is set by the hardware whenever that page is referenced (either a read or a write to any byte in the page). Reference bits are associated with each entry in the page table.

Initially, all bits are cleared (to 0) by the operating system. As a user process executes, the bit associated with each page referenced is set (to 1) by the hardware. After some time, we can determine which pages have been used and which have not been used by

examining the reference bits, although the order of use is not known. This information is the basis for many page-replacement algorithms that approximate LRU replacement.

Additional-Reference-Bits Algorithm

In this algorithm additional ordering information is gained by recording the reference bits at regular intervals. We can keep an 8-bit byte for each page in a table in memory. At regular intervals (say, every 100 milliseconds), a timer interrupt transfers control to OS. The OS shifts the reference bit for each page into the high-order bit of its 8-bit byte, shifting the other bits right by 1 bit and discarding the low-order bit. These 8-bit shift registers contain the history of page use for the last eight time periods. If the shift register contains 00000000, for example, then the page has not been used for the last eight time periods. A page that is used at least once in each period has a shift register value of 11111111. A page with a history register value of 11000100 has been used more recently than one with a value of 01110111. If we interpret these 8-bit bytes as unsigned integers, the page with the lowest number is the LRU page, and it can be replaced. Notice that the numbers are not guaranteed to be unique, however. We can either replace (swap out) all pages with the smallest value or use the FIFO method to choose among them.

The number of bits of history included in the shift register can be varied, of course, and is selected (depending on the hardware available) to make the updating as fast as possible. In the extreme case, the number can be reduced to zero, leaving only the reference bit itself. This algorithm is called the second-chance page-replacement algorithm.

Second-Chance Algorithm

The basic algorithm of second-chance replacement is a FIFO replacement algorithm. When a page has been selected, however, we inspect its reference bit. If the value is 0, we proceed to replace this page; but if the reference bit is set to 1, we give the page a second chance and move on to select the next FIFO page. When a page gets a second chance, its reference bit is cleared, and its arrival time is reset to the current time. Thus, a page that is given a second chance will not be replaced until all other pages have been replaced (or given 2^{nd} chances). In addition, if a page is used often enough to keep its reference bit set, it will never be replaced.

One way to implement the second-chance algorithm is as a circular queue (sometimes referred to as the clock algorithm). A pointer (that is, a hand on the clock) indicates which page is to be replaced next. When a frame is needed, the pointer advances until it finds a page with a reference bit ‘0’. As it advances, it clears the reference bits as shown in Figure-6.

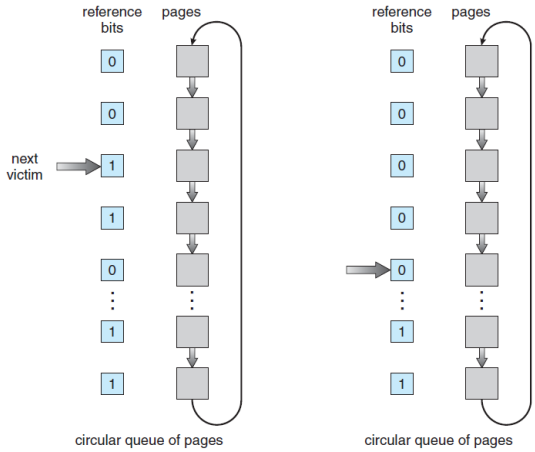
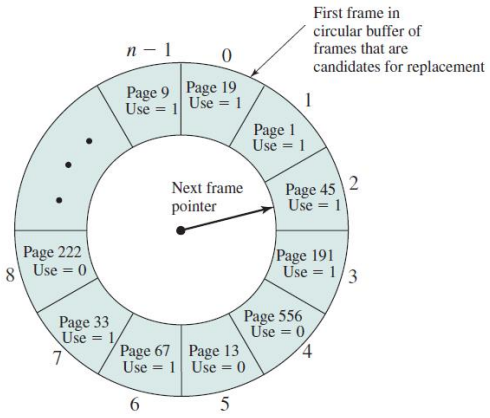
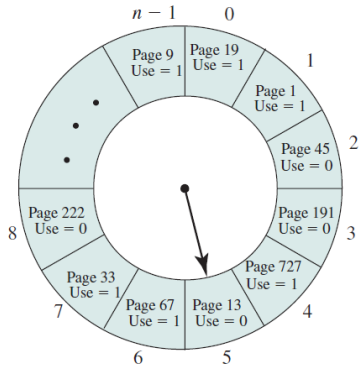


Figure-6: Second Chance (Clock) page replacement

Once a victim page is found, the page is replaced, and the new page is inserted in the circular queue in that position. Notice that, in the worst case, when all bits are set, the pointer cycles through the whole queue, giving each page a second chance. It clears all the reference bits before selecting the next page for replacement. Second-chance replacement degenerates to FIFO replacement if all bits are set.





Page-Buffering Algorithms

Some systems use variants of Page-Buffering Algorithms, by keeping *buffer* (a pool of free frames). When a page fault occurs, a victim frame is chosen as before. However, the desired page is read into a free frame from the pool before the victim is written out. This procedure allows the process to restart as soon as possible, without waiting for the victim page to be written out. When the victim is later written out, its frame is added to the free-frame pool.

In some systems, this idea is further expanded to maintain a list of modified pages. Whenever the paging device is idle, a modified page is selected and is written to the disk. It's modify bit is then reset. This scheme increases the probability that a page will be clean when it is selected for replacement and will not need to be written out.

Some system, even have another modification to keep a pool of free frames but to remember which page was in each frame. Since the frame contents are not modified when

a frame is written to the disk, the old page can be reused directly from the free-frame pool if it is needed before that frame is reused. No I/O operation is needed in this case. When a page fault occurs, we first check whether the desired page is in the free-frame pool. If it is not, we must select a free frame and read into it. This technique is used in the VAX/VMS system along with a FIFO replacement algorithm. When the FIFO replacement algorithm mistakenly replaces a page that is still in active use, that page is quickly retrieved from the free-frame pool, and no I/O is necessary. The free-frame buffer provides protection against the relatively poor, but simple, FIFO replacement algorithm.