ISSN 2348-1196 (print)
International Journal of Computer Science and Information Technology Research ISSN 2348-120X (online)
Vol. 2, Issue 3, pp: (90-99), Month: July - September 2014, Available at: www.researchpublish.com

Page Replacement

S. Jananee

Department of Information Technology & Engineering
Saveetha School Of Engineering, Saveetha University, Chennai

***Abstract*: This paper is about algorithms specific to paging. For outline of general cache algorithms (e.g. processor, disk, database, web),In a computer operating system that uses paging for virtual memory management, page replacement algorithms decide which memory pages to page out (swap out, write to disk) when a page of memory needs to be allocated. Paging happens when a page fault occurs and a free page cannot be used to satisfy the allocation, either because there are none, or because the number of free pages is lower than some threshold. When the page that was selected for replacement and paged out is referenced again it has to be paged in (read in from disk), and this involves waiting for I/O completion. This determines the quality of the page replacement algorithm: the less time waiting for page-ins, the better the algorithm. A page replacement algorithm looks at the limited information about accesses to the pages provided by hardware, and tries to guess which pages should be replaced to minimize the total number of page misses, while balancing this with the costs (primary storage and processor time) of the algorithm itself.**

***Keywords:* Page replacement algorithms.**

I. INTRODUCTION

In our presentation so far, the page-fault rate has not been a serious problem, because each page has faulted at most once, when it is first referenced. This representation is not strictly accurate. If a process of ten pages actually uses only one-half of them, then demand paging saves the I/O necessary to load the five pages that are never used. We could also increase our degree of multiprogramming by running twice as many processes. Thus, if we had 40 frames, we could run eight processes, rather than the four that could run if each required 10 frames (five of which were never used).If we increase our degree of multiprogramming, we are over-allocating memory. If we run six processes, each of which is ten pages in size, but actually uses only five pages, we have higher CPU utilization and throughput, with 10 frames to spare. It is possible, however, that each of these processes, for a particular data set, may suddenly try to use all ten of its pages, resulting in a need for 60 frames, when only 40 are available.

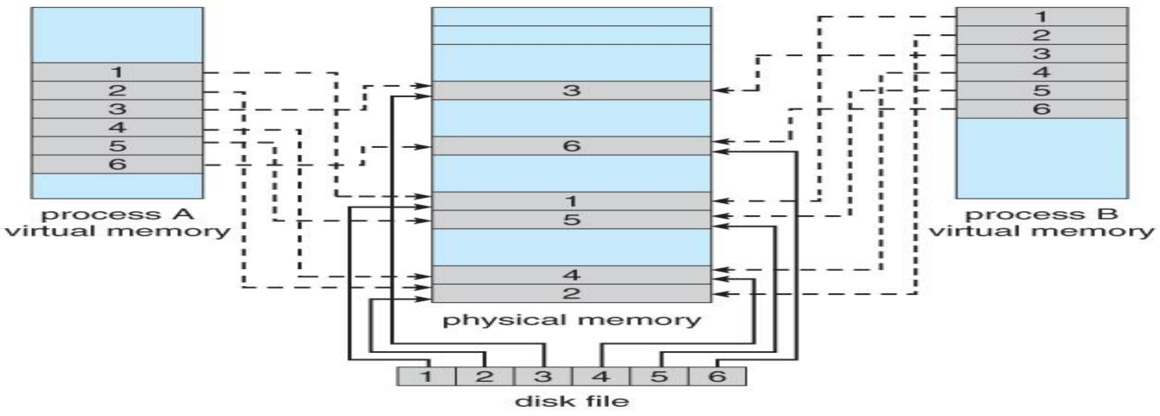

Figure 1 - Memory-mapped files

Although this situation may be unlikely, it becomes much more likely as we increase the multiprogramming level, so that the average memory usage is close to the available physical memory. (In our example, why stop at a multiprogramming level of six, when we can move to a level of seven or eight?) Further, consider that system memory is not used only for holding program pages. Buffers for 1/0 also consume a significant amount of memory. This use can increase the strain on memory-placement algorithms. Deciding how much memory to allocate to I/O and how much to program pages is a significant challenge. Some systems allocate a fixed percentage of memory for I/O buffers, whereas others allow both user processes and the I/O subsystem to compete for all system memory. Over-allocation manifests itself as follows. While a user process is executing, a page fault occurs. The hardware traps to the operating system, which checks its internal tables to see that this page fault is a genuine one rather than an illegal memory access. The operating system determines where the desired page is residing on the disk, but then finds that there are no free frames on the free-frame list: All memory is in use (Figure 2).

The operating system has several options at this point. It could terminate the user process. However, demand paging is the operating system's attempt to improve the computer system's utilization and throughput. Users should not be aware that their processes are running on a paged system-paging should be logically transparent to the user.

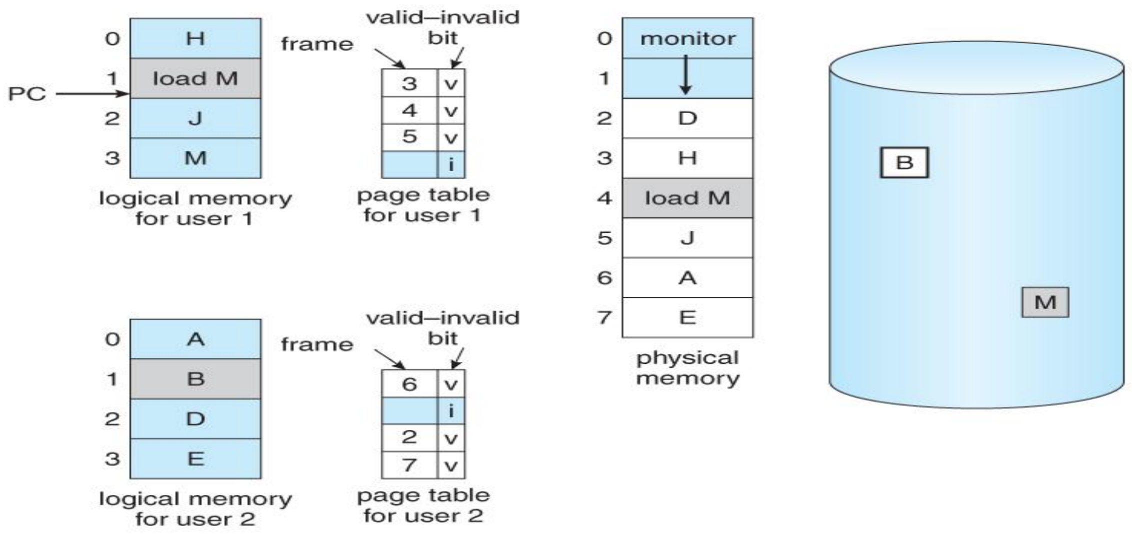

Figure 2- Need for page replacement

So this option is not the best choice. The operating system could swap out a process, freeing all its frames, and reducing the level of multiprogramming. This option is a good one in certain circumstances. Here, we discuss a more intriguing possibility: **page replacement.**

II. BASIC SCHEME

Page replacement takes the following approach. If no frame is free, we find one that is not currently being used and free it. We can free a frame by writing its contents to swap space, and changing the page table (and all tables) to indicate that the page is no longer in memory (Figure 3). We can now use the freed frame to hold the page for which the process faulted. We modify the page-fault service routine to include page replacement:

1. Find the location of the desired page on the disk.

2. Find a free frame:

a. If there is a free frame, use it.

b. If there is no free frame, use a page-replacement algorithm to select a victim frame.

c. Write the victim page to the disk; change the page and frame tables accordingly.

3. Read the desired page into the (newly) free frame; change the page and frame tables.

4. Restart the user process.

ISSN 2348-1196 (print)
International Journal of Computer Science and Information Technology Research ISSN 2348-120X (online)
Vol. 2, Issue 3, pp: (90-99), Month: July - September 2014, Available at: www.researchpublish.com

Notice that, if no frames are free, *two* page transfers (one out and one in) are required. This situation effectively doubles the page-fault service time and increases the effective access time accordingly. We can reduce this overhead by using a modify bit (or dirty bit). Each page or frame may have a modify bit associated with it in the hardware. The modify bit for a page is set by the hardware whenever any word or byte in the page is written into, indicating that the page has been modified. When we select a page for replacement, we examine its modify bit. If the bit is set, we know that the page has been modified since it was read in from the disk. In this case, we must write that page to the disk. If the modify bit is not set, however, the page has not been modified since it was read into memory. Therefore, if the copy of the page on the disk has not been overwritten (by some other page, for example), then we can avoid writing the memory page to the disk: it is already there.

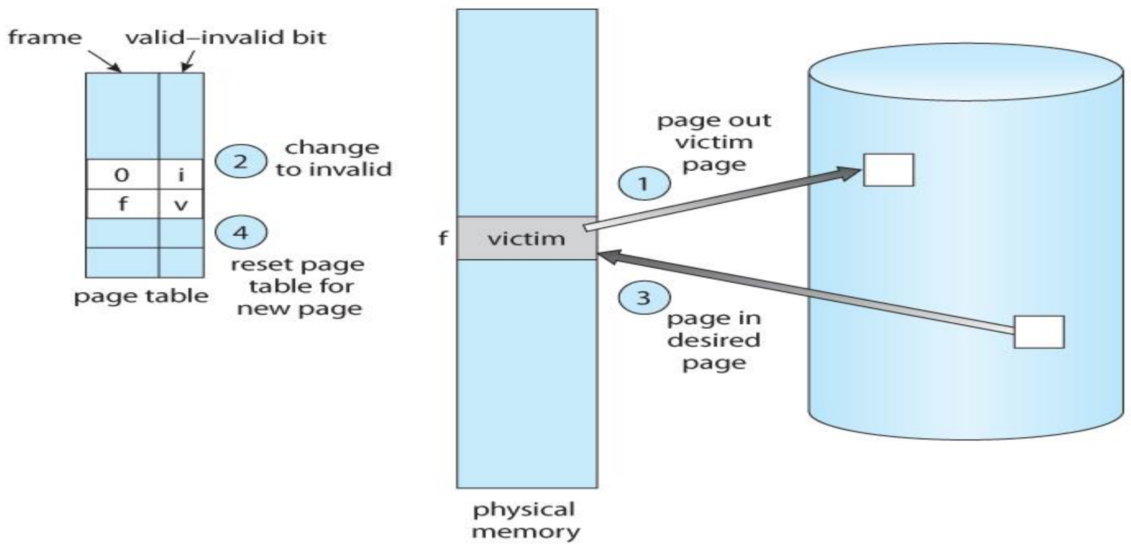

Figure 3- Page Replacement

This technique also applies to read-only pages (for example, pages of binary code).

Such pages cannot be modified; thus, they may be discarded when desired. This scheme can reduce significantly the time required to service a page fault, since it reduces I/O time by one-half if the page is not modified. Page replacement is basic to demand paging. It completes the separation between logical memory and physical memory. With this mechanism, an enormous virtual memory can be provided for programmers on a smaller physical memory. With non-demand paging, user addresses are mapped into physical addresses, so the two sets of addresses can be different. All the pages of a process still must be in physical memory, however. With demand paging, the size of the logical address space is no longer constrained by physical memory. If we have a user process of 20 pages, we can execute it in ten frames simply by using demand paging, and using a replacement algorithm to find a free frame whenever necessary. If a page that has been modified is to be replaced, its contents are copied to the disk. A later reference to that page will cause a page fault. At that time, the page will be brought back into memory, perhaps replacing some other page in the process. We must solve two major problems to implement demand paging: We must develop a **frame-allocation algorithm** and a **page-replacement algorithm.** If we have multiple processes in memory, we must decide how many frames to allocate to each process. Further, when page replacement is required, we must select the frames that are to be replaced. Designing appropriate algorithms to solve these problems is an important task, because disk 1/0 is so expensive.

Even slight improvements in demand-paging methods yield large gains in system performance. There are many different page-replacement algorithms. Every operating system probably has its own replacement scheme. How do we select a particular replacement algorithm? In general, we want the one with the 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 is called a **reference string.** We can generate reference strings artificially (by a random-number generator, for example) or we can trace a given system and record the address of each memory reference. The latter choice produces a large number of data (on the order of 1 million addresses per second). To reduce the number of data, we use two facts.

First, for a given page size (and the page size is generally fixed by the hardware or system), we need to consider only the page number, rather than the entire address. Second, if we have a reference to a page p, then any immediately following references to page p will never cause a page fault. Page p will be in memory after the first reference; the immediately following references will not fault.

For example, if we trace a particular process, we might record the following address sequence:

0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103,
0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105,

Which, at 100 bytes per page, is reduced to the following reference string?

1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1.

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. For the reference string considered previously, for example, if we had three or more frames, we would have only three faults, one fault for the first reference to each page. On the other hand, with only one frame available, we would have a replacement with every reference, resulting in 11 faults. In general, we expect a curve such as that in Figure 4. As the number of frames increases, the number of page faults drops to some minimal level. Of course, adding physical memory increases the number of frames.

To illustrate the page-replacement algorithms, we shall use the reference string

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

for a memory with three frames

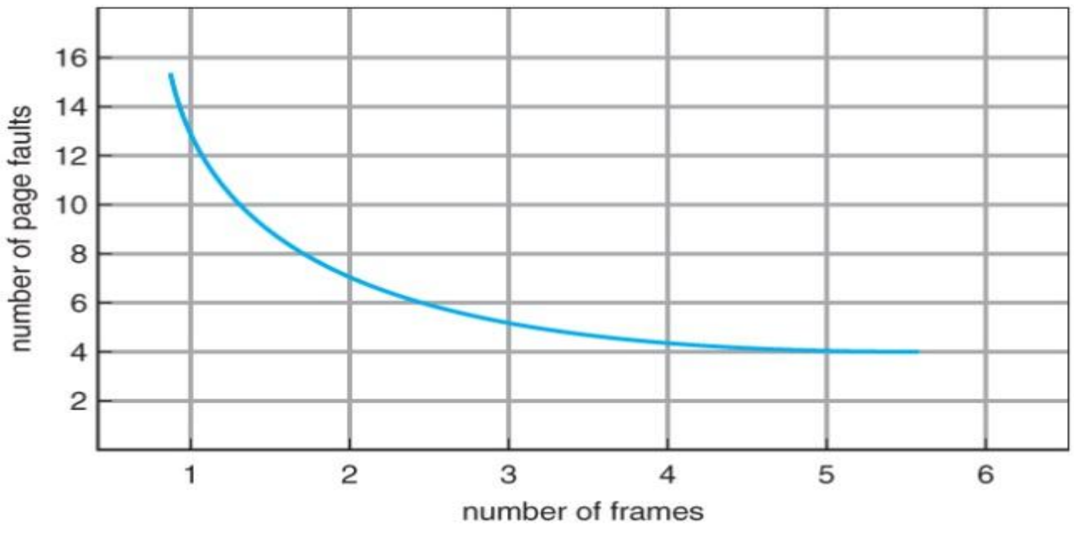

Figure 4- Graph of page faults versus the number of frames

FIFO Page Replacement

The simplest page-replacement algorithm is a FIFO 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. Notice that it is not strictly necessary to record the time when a page is brought in. We can create a FIFO queue to hold all pages in memory. We replace the page at the head of the queue. When a page is brought into memory, we insert it at the tail of the queue.

For our example reference string, our three frames are initially empty. The first three references (7,0,1) cause page faults, and are brought into these empty frames. The next reference (2) replaces page 7, because page 7 was brought in first. Since *0* is the next reference and 0 is already in memory, we have no fault for this reference. The first reference to 3 results in page *0* being replaced, since it was the first of the three pages in memory (0, 1, and 2) to be brought in. Because of this replacement, the next reference, to 0, will fault. Page 1 is then replaced by page 0. This process continues as shown in Figure 5. Every time a fault occurs, we show which pages are in our three frames. There are 15 faults altogether.

The FIFO page-replacement algorithm is easy to understand and program. However, its performance is not always good. The page replaced may be an initialization module that was used a long time ago and is no longer needed.

On the other hand, it could contain a heavily used variable that was initialized early and is in constant use. Notice that, even if we select for replacement a page that is in active use, everything still works correctly. After we page out an active page to bring in a new one, a fault occurs almost immediately to retrieve the active page. Some other page will need to be replaced to bring the active page back into memory. Thus, a bad replacement choice increases the page-fault rate and slows process execution, but does not cause incorrect execution.

To illustrate the problems that are possible with a FIFO page-replacement algorithm, we consider the reference string

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5.

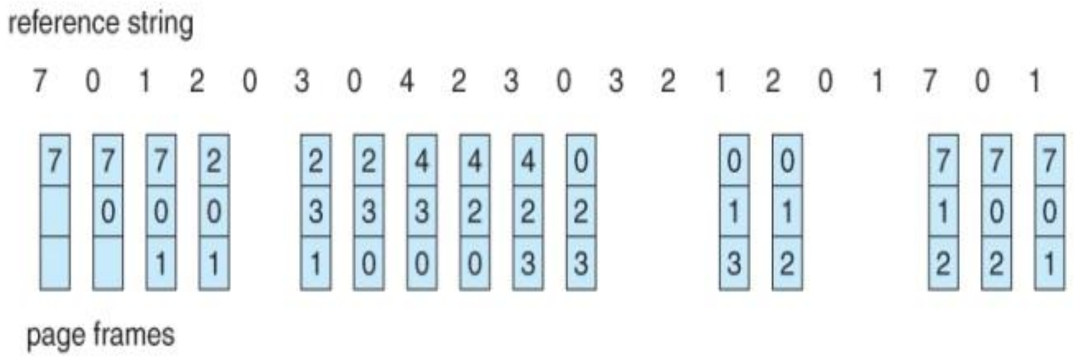

Figure 5- FIFO page-replacement algorithm.

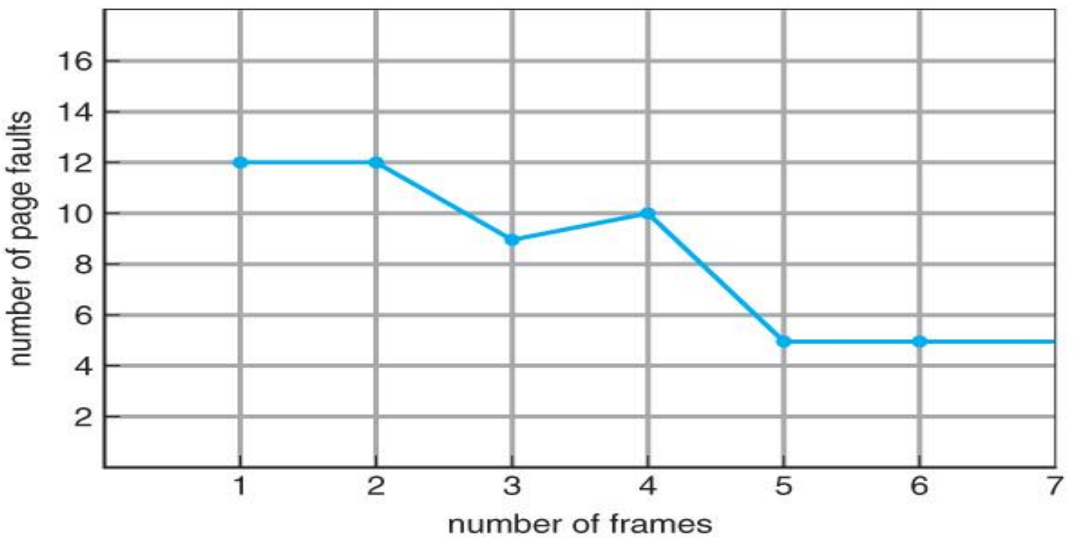

Figure 6- Page-fault curve for FIFO replacement on a reference string

Figure 6 shows the curve of page faults versus the number of available frames. We notice that the number of faults for four frames (10) is *greater* than the number of faults for three frames (nine)! This most unexpected result is known as **Belady's anomaly:** For some page-replacement algorithms, the pagefault rate may *increase* as the number of allocated frames increases. We would expect that giving more memory to a process would improve its performance. In some early research, investigators noticed that this assumption was not always true. Belady's anomaly was discovered as a result.

Optimal Page Replacement

One result of the discovery of Belady's anomaly was the search for an **optimal page-replacement algorithm.** An optimal page-replacement algorithm 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 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.
For example, on our sample reference string, the optimal page-replacement algorithm would yield nine page faults, as shown in Figure 10.11. The first three references cause faults that fill the three empty frames. The reference to page 2

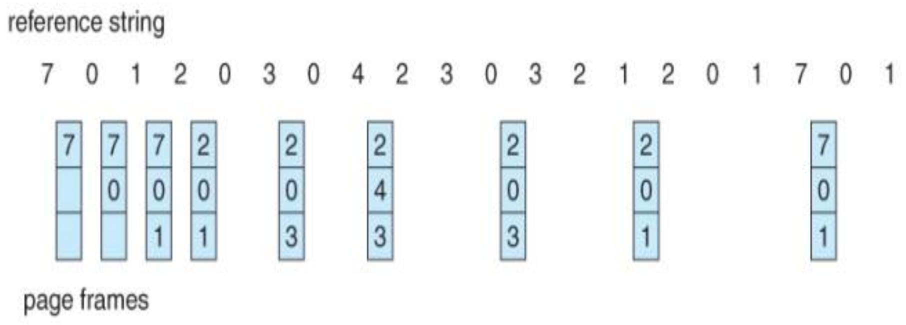

Figure 7- Optimal page-replacement algorithm.

replaces page 7, because 7 will not be used until reference 18, whereas page 0 will be used at 5, and page 1 at 14. The reference to page 3 replaces page 1, as page 1 will be the last of the three pages in memory to be referenced again. With only nine page faults, optimal replacement is much better than a FIFO algorithm, which had 15 faults. (If we ignore the first three, which all algorithms must suffer, then optimal replacement is twice as good as FIFO replacement.) In fact, no replacement algorithm can process this reference string in three frames with less than nine faults. Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string. As a result, the optimal algorithm is used mainly for comparison studies. For instance, it may be useful to know that, although a new algorithm is not optimal, it is within 12.3 percent of optimal at worst, and within 4.7 percent on average.

LRU Page Replacement

If the optimal algorithm is not feasible, perhaps an approximation to the optimal algorithm is possible. The key distinction between the FIFO and OPT algorithms (other than looking backward or forward in time) is that the FIFO algorithm uses the time when a page was brought into memory; the OPT 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 will replace the page that *has not been used* for the longest period of time (Figure 8). 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 that page that has not been used for the longest period of time. This strategy is the optimal page-replacement algorithm looking backward in time, rather than forward. (Strangely, if we let SR be the reverse of a reference string S, then the page-fault rate for the OPT algorithm on S is the same as the page-fault rate for the OPT algorithm on SR.

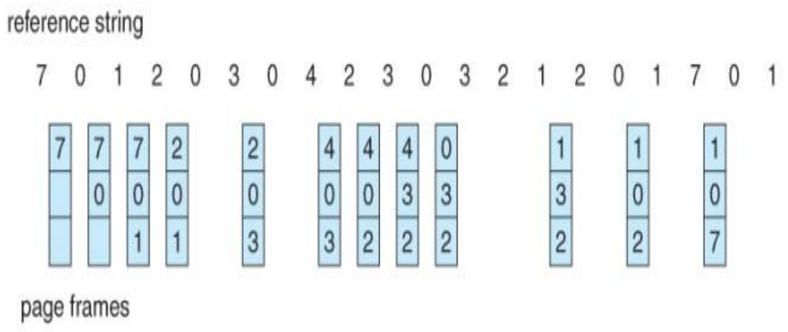

Figure 8- LRU page-replacement algorithm.

Similarly, the page-fault rate for the LRU algorithm on S is the same as the page-fault rate for the LRU algorithm on SR.) The result of applying LRU replacement to our example reference string is shown in Figure 8. The LRU algorithm produces 12 faults. Notice that the first five faults are the same as the optimal replacement. When the reference to page 4 occurs, however, LRU replacement sees that, of the three frames in memory, page 2 was used least recently. The most recently used page is page 0, and just before that page 3 was used. Thus, the LRU algorithm replaces page 2, not knowing that page 2 is about to be used. When it then faults for page 2, the LRU algorithm replaces page 3 since, of the three pages in memory {0,3,4}, page 3 is the least recently used. Despite these problems, LRU replacement with 12 faults is still much better than FIFO replacement with 15. The LRU policy is often used as a page-replacement algorithm and is considered to be good. The major 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. Two implementations are feasible:

Counters:

In the simplest case, we 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 clock must be considered.

Stack:

Another approach to implementing LRU replacement is to 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 top of the stack is always the most recently used page and the bottom is the LRU page (Figure 9). Because entries must be removed from the middle of the stack, it is best implemented by a doubly linked list, with a head and 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. Neither optimal replacement nor LRU replacement suffers from Belady's anomaly. There is a class of page-replacement algorithms, called stack algorithms, that can never exhibit Belady's anomaly. A stack algorithm is an algorithm for which it can be shown that the set of pages in memory for *n* frames is always a *subset* of the set of pages that would be in memory with *n* + 1 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.

Note that neither implementation of LRU would be conceivable without hardware assistance beyond the standard TLB registers. The updating of the clock fields or stack must be done for *every* memory reference. If we were to 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, hence slowing every user process by a factor of ten. Few systems could tolerate that level of overhead for memory management.

LRU Approximation Page Replacement

Few computer systems provide sufficient hardware support for true LRU page replacement. Some systems provide no hardware support, and other page

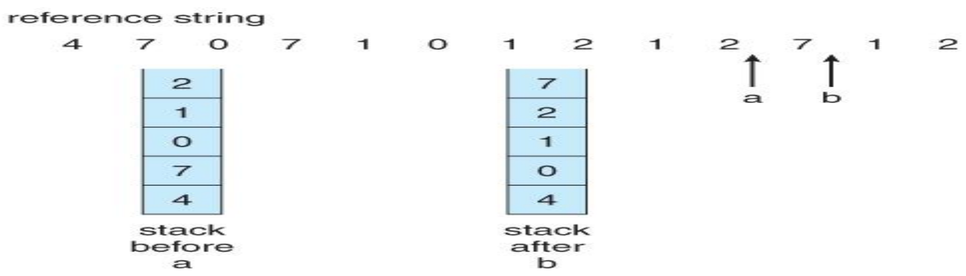

Figure 9- Use of a stack to record the most recent page references

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. We do not know the order of use, but we know which pages were used and which were not used. This partial ordering information leads to many page-replacement algorithms that approximate LRU replacement.

Additional-Reference-Bits Algorithm

We can gain additional ordering information 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 the operating system. The operating system shifts the reference bit for each page into the high-order bit of its 8-bit byte, shifting the other bits right 1 bit, discarding the low-order bit. These &bit shift registers contain the history of page use for the last eight time periods. If the shift register contains 00000000, then the page has not been used for eight time periods; a page that is used at least once each period would have a shift register value of 11111111. A page with a history register value of 11000100 has been used more recently than has one with 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 a FIFO selection among them. The number of bits of history can be varied, of course, and would be 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. If the reference bit is set to 1, however, we give that 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 are replaced (or given second 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 (sometimes referred to as the clock) algorithm is as a circular queue. A pointer indicates which page is to be replaced next. When a frame is needed, the pointer advances until it finds a page with a 0 reference bit. As it advances, it clears the reference bits (Figure 10). 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.

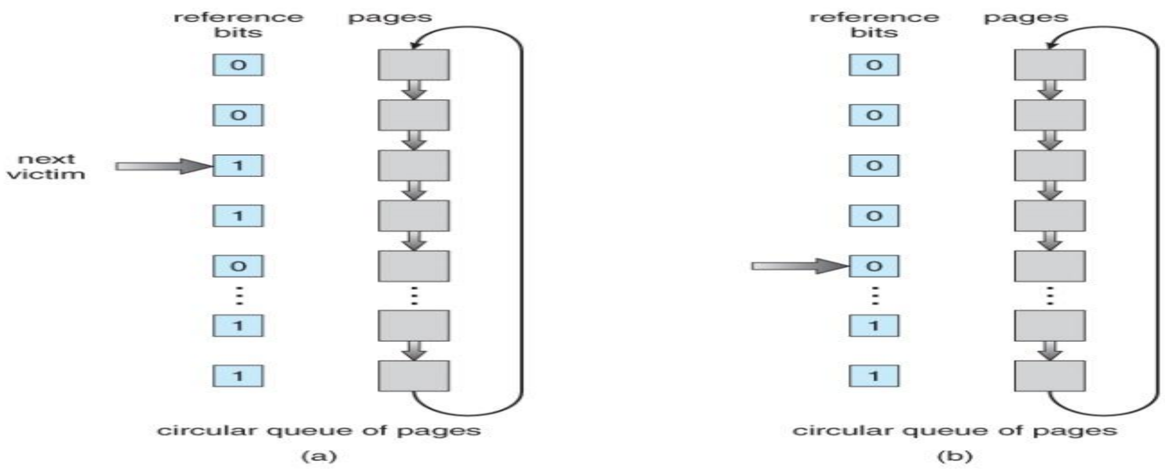

Figure 10- Second-chance (clock) page-replacement algorithm.

Enhanced Second-Chance Algorithm

We can enhance the second-chance algorithm by considering both the reference bit and the modify bit (Section 10) as an ordered pair. With these two bits, we have the following four possible classes:

1. (0,0) neither recently used nor modified-best page to replace
2. (0,1) not recently used but modified-not quite as good, because the page will need to be written out before replacement
3. (1,0) recently used but clean-it probably will be used again soon
4. (1,1) recently used and modified-it probably will be used again soon, and the page will be need to be written out to disk before it can be replaced

When page replacement is called for, each page is in one of these four classes. We use the same scheme as the clock algorithm, but instead of examining whether the page to which we are pointing has the reference bit set to 1, we examine the class to which that page belongs. We replace the first page encountered in the lowest nonempty class. Notice that we may have to scan the circular queue several times before we find a page to be replaced. This algorithm is used in the Macintosh virtual-memory-management scheme. The major difference between this algorithm and the simpler clock algorithm is that here we give preference to those pages that have been modified to reduce the number of I/Os required.

Counting-Based Page Replacement

There are many other algorithms that can be used for page replacement. For example, we could keep a counter of the number of references that have been made to each page, and develop the following two schemes.

The **least frequently used** (LFU) **page-replacement algorithm** requires that the page with the smallest count be replaced. The reason for this selection is that an actively used page should have a large reference count. This algorithm suffers from the situation in which a page is used heavily during the initial phase of a process, but then is never used again. Since it was used heavily, it has a large count and remains in memory even though it is no longer needed. One solution is to shift the counts right by 1 bit at regular intervals, forming an exponentially decaying average usage count.

The **most frequently used** (MFU) **page-replacement algorithm** is based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

As you might expect, neither MFU nor LFU replacement is common. The implementation of these algorithms is expensive, and they do not approximate OPT replacement well.

Page-Buffering Algorithm

Other procedures are often used in addition to a specific page-replacement algorithm. For example, systems commonly keep 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. An expansion of this idea is to maintain a list of modified pages. Whenever the paging device is idle, a modified page is selected and is written to the disk. Its 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. Another modification is to keep a pool of free frames, but to remember the reference bit correctly. 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 1/0 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, 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 buffer, and no I/O is necessary. The free-frame buffer provides protection against the relatively poor, but simple, FIFO replacement algorithm. This method is necessary because the early versions of the VAX did not implement the reference bit correctly.

III. CONCLUSION

Page Replacement Algorithms is used to prevent over-allocation of memory by modifying page-fault service routine to include page replacement. It will use *dirty (modify)* bit to reduce overhead of page transfers; only modified pages written back to disk. Page replacement completes separation between logical memory and physical memory; large virtual memory can be provided on smaller physical memory. It will find location of desired page on disk. It will find free frame only if free frame, use it ,if no free frame, page replacement algorithm selects *victim* frame, if victim *dirty* write back to disk. Read desired page into (newly) free frame; update page and frame tables. Restart faulting process. Want lowest page-fault rate. Evaluate algorithm by running on given string of memory references (reference string) and compute number of page faults. The Optimal Page Replacement Algorithm; the page that will not be referenced again for the longest time is replaced (prediction of the future; purely theoretical, but useful for comparison.).The Not Recently Used Page Replacement Algorithm; algorithm removes a page at random. The First-In, First-Out (FIFO) Page Replacement Algorithm; FIFO the frames are treated as a circular list; the oldest (longest resident) page is replaced. The Second Chance Page Replacement Algorithm; look for an old page that has not been referenced in the previous clock interval, avoids the problem of throwing out of heavily used page. The Least Recently Used (LRU) Page Replacement Algorithm; LRU the frame whose contents have not been used for the longest time is replaced.

REFERENCES

[1] X. Munoz, J. Freixenet, X. Cufi, and J. Marti, Strategies for image segmentation combining region and boundary information,‖ Pattern Recognition Letters, vol. 24, no. 1, pp. 375–392, 2003.

[2] D. Pham, C. Xu, and J. Prince, A survey of current methods in medical image segmentation,‖ In Annual Review of Biomedical Engineering, vol. 2, pp. 315–337, 2000.

[3] Mohammad Ali Balafar, Abd.RahmanRamli, M.IqbalSaripan, SyamsiahMashohor, Medical Image Segmentation Using Fuzzy CMean (Fcm), Bayesian Method And User Interaction,‖ Proceedings of the 2008 International Conference on Wavelet Analysis and Pattern Recognition, pp. 68-73, Aug. 2008.

[4] LászlóSzilágyi, Sándor M. Szilágyi, BalázsBenyó and Zoltán Benyó, Application of Hybrid c-Means Clustering Models in Inhomogeneity Compensation and MR Brain Image Segmentation ,‖ 5th International Symposium on Applied Computational Intelligence and Informatics ,pp.105-110, May. 2009.

[5] Mac Queen ,J.B. Some Methods for classification and Analysis of Multivariate Observations, "Proceedings of 5th Berkeley Symposium on Mathematical Statistics and Probability. University of California Press, pp. 281–297, 1967.

[6] Arthur D,VassilvitskiiS, "How Slow is the k-means Method?," Proceedings of the 2006 Symposium on Computational Geometry , June. 2006.

[7] L. Zadeh, Fuzzy sets,‖ Inf. Control, vol. 8, pp. 338–353, 1965.

[8] J. Udupa and S. Samarasekera, Fuzzy connectedness and object definition: Theory, algorithm and applications in image segmentation,‖ Graphical Models and Image Processing, vol. 58, no. 3, pp. 246–261, 1996.

[9] Y. Tolias and S. Panas, Image segmentation by a fuzzy clustering algorithm using adaptive spatially constrained membership functions,‖ IEEE Transactions on Systems, Man, and Cybernetics, vol. 28, no. 3, pp. 359–369, Mar.1998.

[10] J. Noordam, W. van den Broek, and L. Buydens, Geometrically guided fuzzy C-means clustering for multivariate image segmentation,‖ in Proceedings of the International Conference on Pattern Recognition, 2000, vol. 1, pp.462–465