
CODE DIGITAL
HARDWARE DESIGN
INTERNET
SCREEN
DATA
SERVICE
GRAPHIC
INPUT
APPLICATION
OS
APP
WIRELESS
USER
NETWORK
DEVELOPMENT
INFORMATIONCONCEPT
COMPUTING
CONCEPTUAL
MEMORY
BACKGROUND
DESKTOP
PROGRAMMING
OPERATING
SYSTEM
TECHNOLOGY
ALLOCATION
PROGRAM
DEVICE
COMPUTER
SOFTWARE
STORAGE
MOBILE
BUSINESS
WEB
COMMUNICATION
SYMBOL
OUTPUT

Jump2Learn
The Online Learning Place

OPERATING SYSTEM - II

CHAPTER 5 : VIRTUAL MEMORY MANAGEMENT

AUTHORS

Mr. Yatin Solanki
M.C.A., Ph.D.(Pursuing)

DOLAT USHA INSTITUTE OF APPLIED SCIENCES, VALSAD

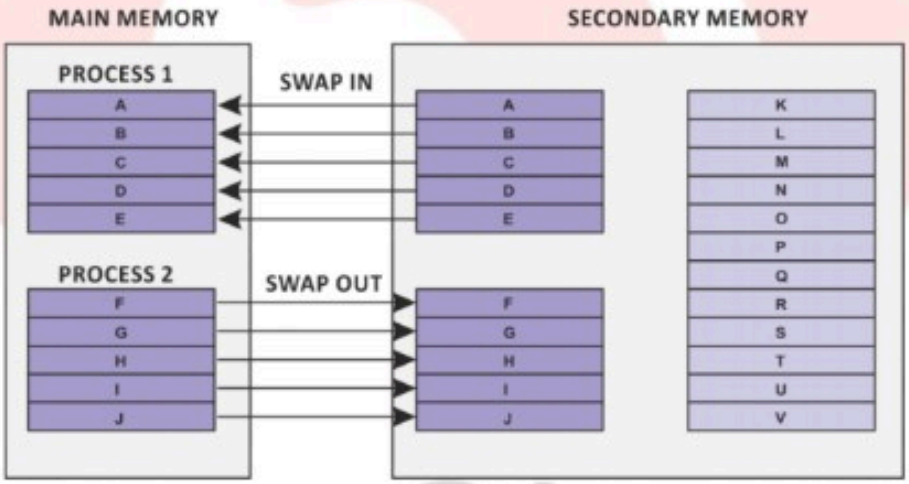
Dr. Jaimin Shukla
M.C.A., M. Phil., Ph.D.(Pursuing)

SUTEX BANK COLLEGE OF COMPUTER APPLICATIONS & SCIENCE, AMROLI

Website : www.jump2learn.com | **Email :** info@jump2learn.com | **Instagram :** www.instagram.com/jump2learn
Facebook : www.facebook.com/Jump2Learn | **Whatsapp :** +91-909-999-0960 | **YouTube :** Jump2Learn

5.1 DEMAND PAGING

- In computer operating systems, demand paging is a method of virtual memory management.
- In a system that uses demand paging, the operating system copies a disk page into physical memory only if an attempt is made to access it and that page is not already in memory.
- A demand paging system is quite similar to a paging system with swapping where processes reside in secondary memory and pages are loaded only on demand, not in advance.
- When a context switch occurs, the operating system does not copy any of the old program's pages out to the disk or any of the new program's pages into the main memory Instead, it just begins executing the new program after loading the first page and fetches that program's pages as they are referenced.
- In computing, a context switch is the process of storing the state of a process or of a thread, so that it can be restored and execution resumed from the same point later. This allows multiple processes to share a single CPU, and is an essential feature of a multitasking operating system.



- While executing a program, if the program references a page which is not available in the main memory because it was swapped out a little ago, the processor treats this invalid memory reference as a **page fault** and transfers control from the program to the operating system to demand the page back into the memory.

ADVANTAGES:

- Large virtual memory.
- More efficient use of memory.
- There is no limit on degree of multiprogramming.

DISADVANTAGES:

- Number of tables and the amount of processor overhead for handling page interrupts are greater than in the case of the simple paged management techniques.

PAGE REPLACEMENT ALGORITHMS

- Page replacement algorithms are the techniques using which an Operating System decides which memory pages to swap out, write to disk when a page of memory needs to be allocated.
- Paging happens whenever a page fault occurs and a free page cannot be used for allocation purpose accounting to reason that pages are not available or the number of free pages is lower than required pages.
- When the page that was selected for replacement and was paged out, is referenced again, it has to read in from disk, and this requires for I/O completion. This process determines the quality of the page replacement algorithm: the lesser the time waiting for page-ins, the better is the algorithm.

REFERENCE STRING:

- The string of memory references is called reference string. Reference strings are generated artificially or by tracing a given system and recording the address of each memory reference.
- For a given page size, we need to consider only the page number, not the entire address.

PAGE FAULT:

- A page fault happens when a running program accesses a memory page that is mapped into the virtual address space, but not loaded in physical memory.
- Since actual physical memory is much smaller than virtual memory, page faults happen. In case of page fault, Operating System might have to replace one of the existing pages with newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce number of page faults.
- An interrupt that occurs when a program requests data that is not currently in real memory. The interrupt triggers the operating system to fetch the data from a virtual memory and load it into RAM. An invalid page fault or page fault error occurs when the operating system cannot find the data in virtual memory.

5.2 ALLOCATION OF FRAMES

- → How do we allocate the fixed amount of free memory among the various processes? If we have 93 free frames and two processes, how many frames does each process get?
- → The simplest case of virtual memory is the single-user system. Consider a single-user system with 128 KB memory composed of pages of size 1 KB. Thus, there are 128 frames. The operating system may take 35 KB, leaving 93 frames for the user process.

→ Under pure demand paging, all 93 frames would initially be put on the free-frame list. When a user process started execution, it would generate a sequence of page faults. The first 93 page faults would all get free frames from the free-frame list. When the free-frame list was exhausted, a page replacement algorithm would be used to select one of the 93 in-memory pages to be replaced with the ninety-fourth, and so on. When the process terminated, the 93 frames would once again be placed on the free-frame list.

*** ALLOCATION ALGORITHMS**

The easiest way to split *m* frames among *n* processes is to give everyone an equal share, *m/n* frames. For instance, if there are 93 frames and five processes, each process will get 18 frames. The leftover three frames could be used as a free-frame buffer pool. This scheme is called **equal allocation.**

→ An alternative is to recognize that various processes will need differing amounts of memory. Consider a system with a 1 KB frame size. If a small student process of 10 KB and an interactive database of 127 KB are the only two processes running in a system with 62 free frames, it does not make much sense to give each process 31 frames. The student process does not need more than ten frames, so the other 21 are strictly wasted.

→ To solve this problem, we can use **proportional allocation.** We allocate available memory to each process according to its size. Let the size of the virtual memory for process ***pi*** be ***si,*** and define

$$S = \sum s_i .$$

Then, if the total number of available frames is ***m,*** we allocate ***ai*** frames to process **pi**, where ***ai*** is approximately

$$a_i = s_i / S \times m.$$

Of course, we must adjust each *ai* to be an integer that is greater than the minimum number of frames required by the instruction set, with a sum not exceeding ***m.***

→ For proportional allocation, we would split 62 frames between two processes, one of 10 pages and one of 127 pages, by allocating 4 frames and 57 frames, respectively,

$$10/137 \times 62 \approx 4, \text{ and}$$
$$127/137 \times 62 \approx 57.$$

since In this way, both processes share the available frames according to their "needs," rather than equally.

→ In both equal and proportional allocation, of course, the allocation to each process may vary according to the multiprogramming level. If the multiprogramming

level is increased, each process will lose some frames to provide the memory needed for the new process. On the other hand, if the multiprogramming level decreases, the frames that had been allocated to the departed process can now be spread over the remaining processes.

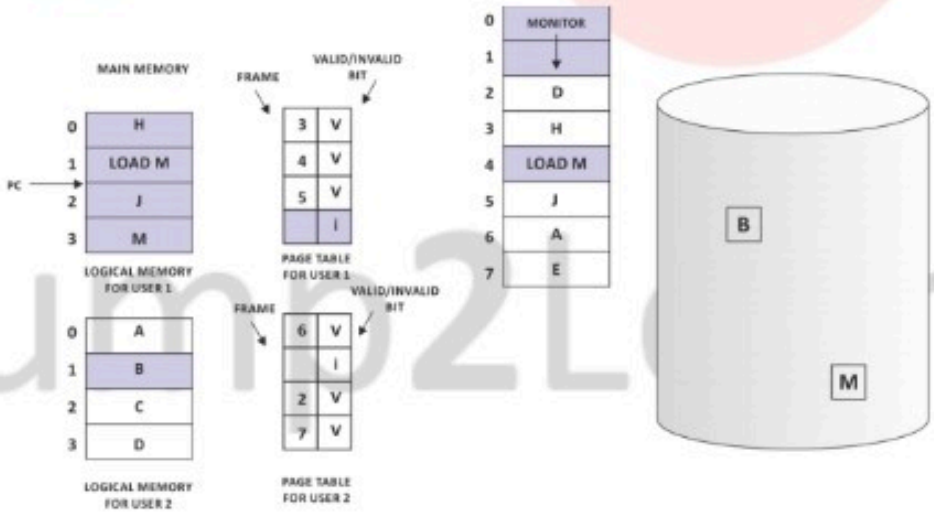
→ Notice that, with either equal or proportional allocation, a high-priority process is treated the same as a low-priority process. By its definition, however, we may want to give the high-priority process more memory to speed its execution, to the detriment of low-priority processes.

→ One approach is to use a proportional allocation scheme where the ratio of frames depends not on the relative sizes of processes, but rather on the priorities of processes, or on a combination of size and priority.

GLOBAL VERSUS LOCAL ALLOCATION

→ Another important factor in the way frames are allocated to the various processes is page replacement. With multiple processes competing for frames, we can classify page-replacement algorithms into two broad categories: **global replacement** and **local replacement.** Global replacement allows a process to select a replacement frame from the set of all frames, even if that frame is currently allocated to some other process; one process can take a frame from another. Local replacement requires that each process select from only its own set of allocated frames.

5.2 PAGE REPLACEMENT TECHNIQUE:



→ 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 9.9).

→ 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. 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.**

INCLUDE PAGE REPLACEMENT:

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

2. Find a free frame:

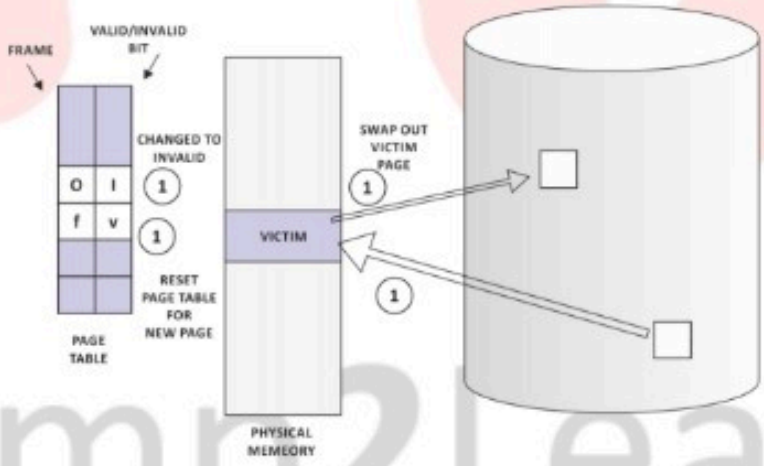
a. If there is a free frame, use it.

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.



→ 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 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.

FIFO PAGE REPLACEMENT

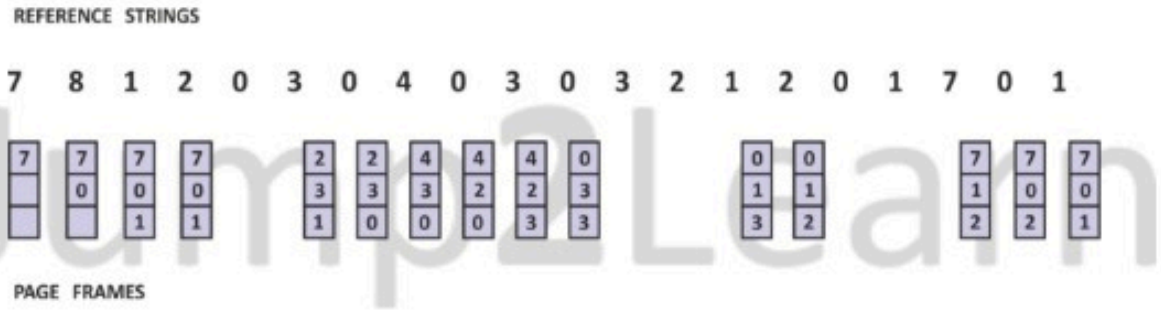
→ The simplest page-replacement algorithm is a FIFO algorithm. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen. Notice that it is not strictly necessary to record the time when a page is brought in.

→ 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 9.12.

→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**.

→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.



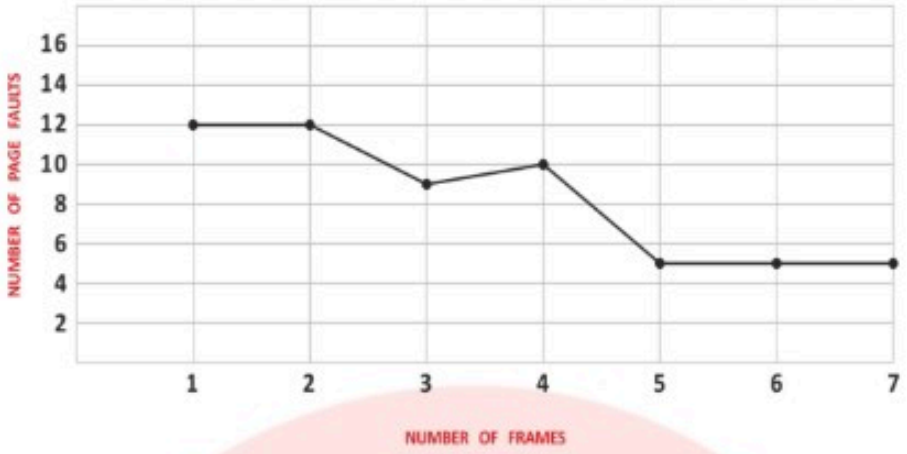


Figure shows the curve of page faults versus the number of available frames. We notice that the number of faults for four frames (10) is *greater* than the number of faults for three frames (nine)! This most unexpected result is known as **Belady's anomaly:** For some page-replacement algorithms, the page fault rate may *increase* as the number of allocated frames increases.

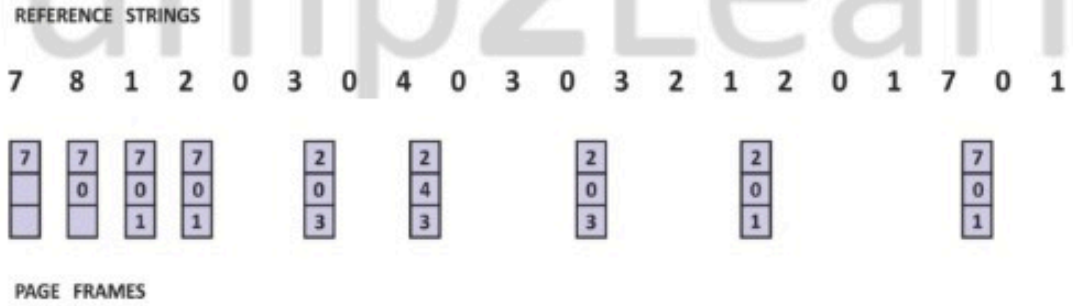
OPTIMAL PAGE REPLACEMENT

→ 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 9.14. The first three references cause faults that fill the three empty frames. The reference to page

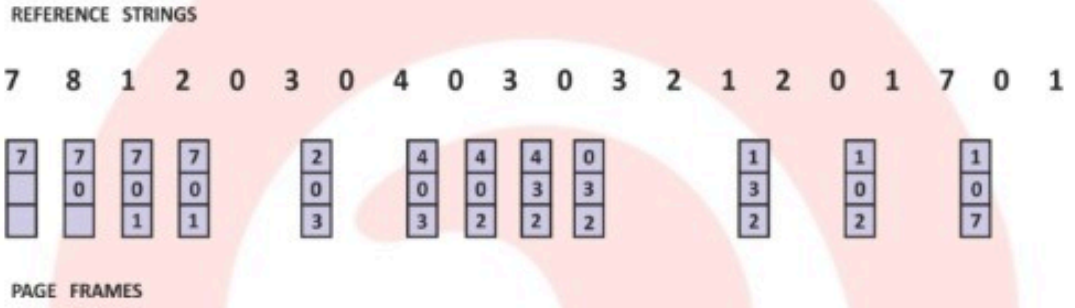


2 replaces page 7, because 7 will not be used until reference 18, whereas page 0 will be used at 5, and page 1 at 14. The reference to page 3 replaces page 1, as page 1 will be the last of the three pages in memory to be referenced again.

→ With only nine page faults, optimal replacement is much better than a FIFO algorithm, which had 15 faults. the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string.

LRU PAGE REPLACEMENT

→ If the optimal algorithm is not feasible. 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 9.15). 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.

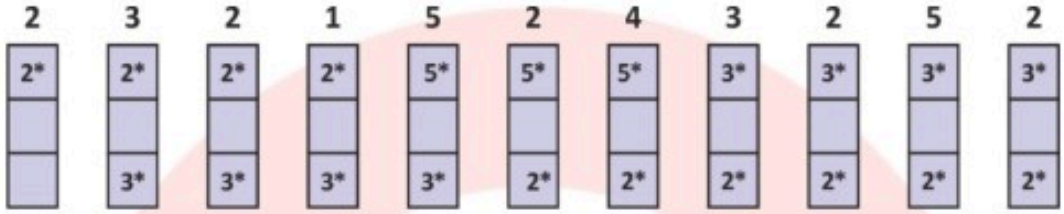
→ The result of applying LRU replacement to our example reference string is shown in Figure 10.12. The LRU algorithm produces 12 faults. Notice that the first five faults are the same as the optimal replacement. When the reference to page 4 occurs, however, LRU replacement sees that, of the three frames 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.

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).

→ **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). 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.



*** Enhanced Second-Chance Algorithm**

(0,0) FIRST BIT FOR REFERENCE BIT AND SECOND BIT IS MODIFY BIT

We can enhance the second-chance algorithm by considering both the **reference bit and the modify bit** 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,l) **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, l) **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.

.* 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 **Most frequently used** (MFU) **page-replacement algorithm** requires that the page with the highest count be replaced.

***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.

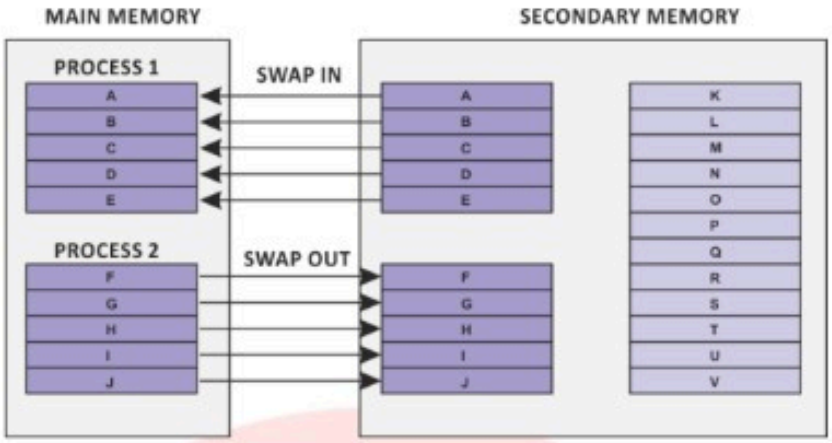
An expansion of this idea is to maintain a list of modified pages. Whenever the paging device is idle, a modified page is selected and is written to the disk. Its modify bit is then reset. This scheme increases the probability that a page will be clean when it is selected for replacement, and will not need to be written out.

Another modification is to keep a pool of free frames, but to remember 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.

5.4 THRASHING

→ When referring to a computer, **thrashing** or **disk thrashing** is a term used to describe when the hard drive is being overworked by moving information between the system memory and virtual memory excessively. Thrashing is often caused when the system does not have enough memory, the system swap file is not properly configured, or too much is running on the computer and it has low system resources.

- In computer operating systems, demand paging is a method of virtual memory management.
- In a system that uses demand paging, the operating system copies a disk page into physical memory only if an attempt is made to access it and that page is not already in memory.
- A demand paging system is quite similar to a paging system with swapping where processes reside in secondary memory and pages are loaded only on demand, not in advance.
- When a context switch occurs, the operating system does not copy any of the old program's pages out to the disk or any of the new program's pages into the main memory Instead, it just begins executing the new program after loading the first page and fetches that program's pages as they are referenced.
- In computing, a context switch is the process of storing the state of a process or of a thread, so that it can be restored and execution resumed from the same point later. This allows multiple processes to share a single CPU, and is an essential feature of a multitasking operating system.



- While executing a program, if the program references a page which is not available in the main memory because it was swapped out a little ago, the processor treats this invalid memory reference as a **page fault** and transfers control from the program to the operating system to demand the page back into the memory.

ADVANTAGES:

- Large virtual memory.
- More efficient use of memory.
- There is no limit on degree of multiprogramming.

DISADVANTAGES:

- Number of tables and the amount of processor overhead for handling page interrupts are greater than in the case of the simple paged management techniques.