Unit 8 Virtual Memory

Structure:

8.1 Introduction
Objectives
8.2 Need for Virtual Memory Technique
8.3 Demand Paging
8.4 Concept of Page Replacement
8.5 Page Replacement Algorithms
FIFO page replacement algorithm
Optimal algorithm
LRU page replacement algorithm
8.6 Thrashing
Causes for thrashing
Working set model
Page fault frequency
8.7 Summary
8.8 Terminal Questions
8.9 Answers

8.1 Introduction

In the last unit you studied about memory management strategies like paging and segmentation which helps to implement the concept of multi-programming. But they have a few disadvantages. One problem with the above strategies is that they require the entire process to be in main memory before execution can begin. Another disadvantage is the limitation on the size of the process. Processes whose memory requirement is larger than the maximum size of the memory available, will never be able to be run, that is, users are desirous of executing processes whose logical address space is larger than the available physical address space.

To address the above said problem in this unit we are going to discuss a new technique called virtual memory that allows execution of processes that may not be entirely in memory. In addition, virtual memory allows mapping of a large virtual address space onto a smaller physical memory. It also raises the degree of multi-programming and increases CPU utilization.

Because of the above features, users are freed from worrying about memory requirements and availability.

Objectives:

After studying this unit, you should be able to:

- explain virtual memory technique and its need
- discuss demand paging
- describe different page replacement algorithms
- explain thrashing and its causes

8.2 Need for Virtual Memory Technique

Every process needs to be loaded into physical memory for execution. One brute force approach to this is to map the entire logical space of the process to physical memory, as in the case of paging and segmentation.

Many a time, the entire process need not be in memory during execution. The following are some of the instances to substantiate the above statement:

- Code used to handle error and exceptional cases is executed only in case errors and exceptional conditions occur, which is usually a rare occurrence, may be one or no occurrences in an execution.
- Static declarations of arrays lists and tables declared with a large upper bound but used with no greater than 10% of the limit.
- Certain features and options provided in the program as a future enhancement, never used, as enhancements are never implemented.
- Even though entire program is needed, all its parts may not be needed at the same time because of overlays.

All the examples show that a program can be executed even though it is partially in memory. This scheme also has the following benefits:

- Physical memory is no longer a constraint for programs and therefore users can write large programs and execute them.
- Physical memory required for a program is less. Hence degree of multi-programming can be increased because of which utilization and throughput increase.
- I/O time needed for load / swap is less.

Virtual memory is the separation of logical memory from physical memory. This separation provides a large logical / virtual memory to be mapped on to a small physical memory (Refer figure 8.1).

Virtual memory is implemented using demand paging. Also demand segmentation could be used. A combined approach using a paged segmentation scheme is also available. Here user view is segmentation but the operating system implements this view with demand paging.

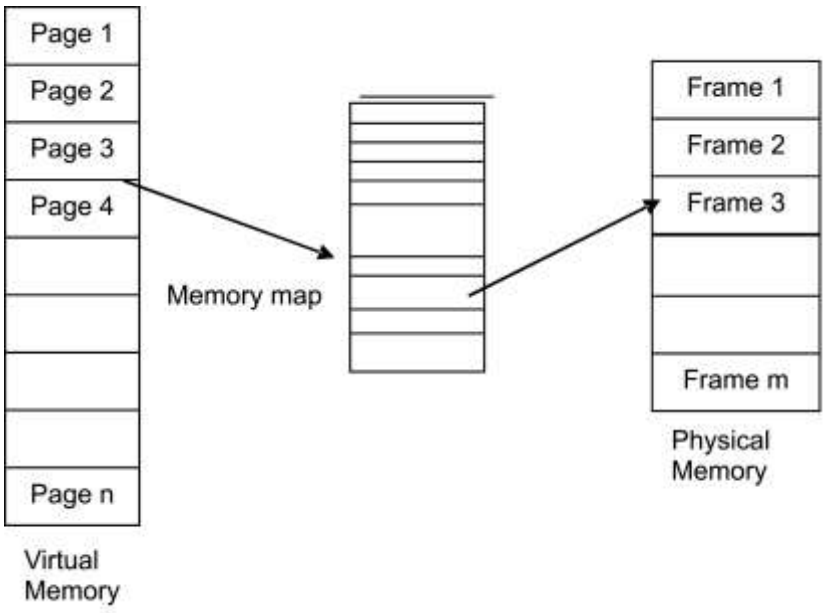


Fig. 8.1: Virtual to physical memory mapping (n >> m)

8.3 Demand Paging

Demand paging is similar to paging with swapping (Refer figure 8.2). When a process is to be executed then only that page of the process, which needs to be currently executed, is swapped into memory. Thus, only necessary pages of the process are swapped into memory thereby decreasing swap time and physical memory requirement.

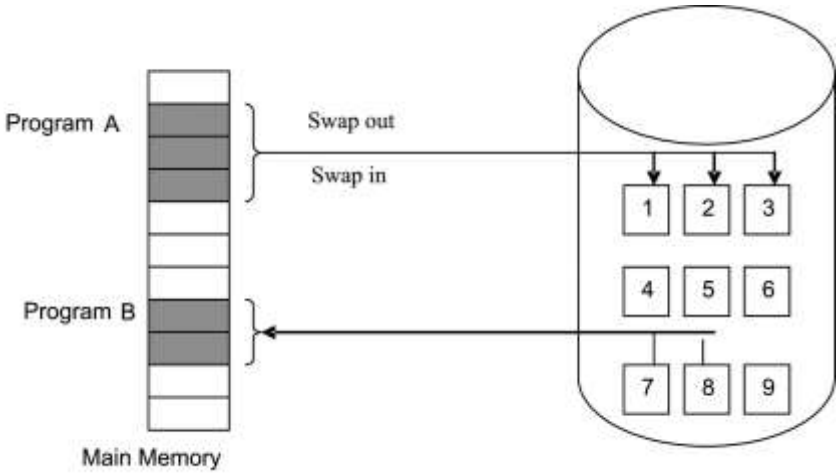


Fig. 8.2: Paging with swapping

The protection valid-invalid bit which is used in paging to determine valid / invalid pages corresponding to a process is used here also (Refer figure 8.3).

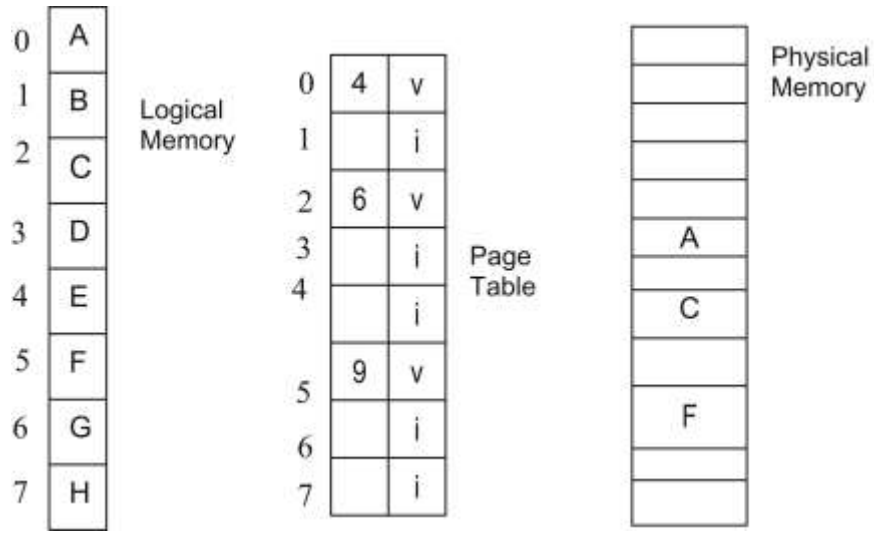


Fig. 8.3: Demand paging with protection

If the valid-invalid bit is set, then the corresponding page is valid and also in physical memory. If the bit is not set, then any of the following can occur:

- Process is accessing a page not belonging to it, that is, an illegal memory access.

- Process is accessing a legal page but the page is currently not in memory.

If the same protection scheme as in paging is used, then in both the above cases a page fault error occurs. The error is valid in the first case but not in the second because in the latter a legal memory access failed due to non-availability of the page in memory which is an operating system fault. Page faults can thus be handled as follows (Refer figure 8.4):

1) Check the valid-invalid bit for validity.
2) If valid, then the referenced page is in memory and the corresponding physical address is generated.
3) If not valid then, an addressing fault occurs.
4) The operating system checks to see if the page is in the backing store. If present, then the addressing error was only due to non-availability of page in main memory and is a valid page reference.
5) Search for a free frame.
6) Bring in the page into the free frame.
7) Update the page table to reflect the change.
8) Restart the execution of the instruction stalled by an addressing fault.

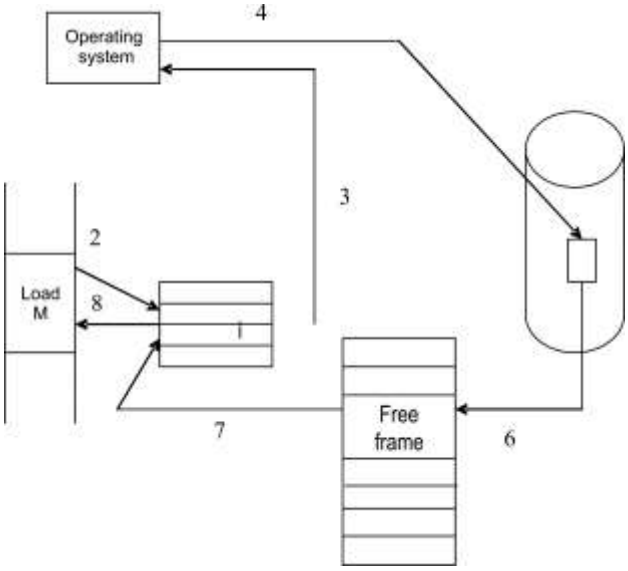


Fig. 8.4: Handling a page fault

In the initial case, a process starts executing with no pages in memory. The very first instruction generates a page fault and a page is brought into

memory. After a while all pages required by the process are in memory with a reference to each page generating a page fault and getting a page into memory. This is known as pure demand paging. The concept, never bring in a page into memory until it is required'.

Hardware required to implement demand paging is the same as that for paging and swapping.

- Page table with valid-invalid bit.
- Secondary memory to hold pages not currently in memory, usually a high speed disk known as a swap space or backing store.
- A page fault at any point in the fetch-execute cycle of an instruction causes the cycle to be repeated.

Self Assessment Questions

1. Every process needs to be loaded into physical memory for execution. (True / False)
2. _____ is implemented using demand paging.
3. When a process is to be executed then only that page of the process, which needs to be currently executed, is swapped into memory. This method is called _____. (Pick the right option)
   a) Demand Paging
   b) Request Paging
   c) Swap Paging
   d) Change Paging

8.4 Concept of Page Replacement

Initially, execution of a process starts with none of its pages in memory. Each of its pages will have page fault at least once when it is first referenced. But it may so happen that some of its pages are never used. In such a case those pages which are not referenced even once will never be brought into memory. This saves load time and memory space. If this is so, the degree of multi-programming can be increased so that more ready processes can be loaded and executed. Now, we may come across a situation wherein all of sudden, a process hitherto not accessing certain pages starts accessing those pages. The degree of multi-programming has been raised without looking into this aspect and the memory is over allocated. Over allocation of memory shows up when there is a page fault for want of page in memory

and the operating system finds the required page in the backing store but cannot bring in the page into memory for want of free frames. More than one option exists at this stage:

- Terminate the process. This is not a good option because the very purpose of demand paging to increase CPU utilization and throughput by increasing the degree of multi-programming is lost.
- Swap out a process to free all its frames. This reduces the degree of multi-programming that again may not be a good option but better than the first.
- Page replacement seems to be the best option in many cases.

The page fault service routine can be modified to include page replacement as follows:

1) Find for the required page in the backing store.
2) Find for a free frame
   a) if there exists one use it
   b) if not, find for a victim using a page replacement algorithm
   c) write the victim into the backing store
   d) modify the page table to reflect a free frame
3) Bring in the required page into the free frame.
4) Update the page table to reflect the change.
5) Restart the process.

When a page fault occurs and no free frame is present, then a swap out and a swap in occur. A swap out is not always necessary. Only a victim that has been modified needs to be swapped out. If not, the frame can be over written by the incoming page. This will save time required to service a page fault and is implemented by the use of a **dirty bit**. Each frame in memory is associated with a dirty bit that is reset when the page is brought into memory. The bit is set whenever the frame is modified. Therefore, the first choice for a victim is naturally that frame with its dirty bit which is not set.

Page replacement is basic to demand paging. The size of the logical address space is no longer dependent on the physical memory. Demand paging uses two important algorithms:

- Page replacement algorithm: When page replacement is necessitated due to non-availability of frames, the algorithm looks for a victim.

- Frame allocation algorithm: In a multi-programming environment with degree of multi-programming equal to n, the algorithm gives the number of frames to be allocated to a process.

8.5 Page Replacement Algorithms

A good page replacement algorithm generates as low a number of page faults as possible. To evaluate an algorithm, the algorithm is run on a string of memory references and a count of the number of page faults is recorded. The string is called a reference string and is generated using either a random number generator or a trace of memory references in a given system.

Illustration:

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

Page size: 100 bytes

Reference string: 1 4 1 6 1 6 1 6 1 6 1

The reference in the reference string is obtained by dividing (integer division) each address reference by the page size. Consecutive occurrences of the same reference are replaced by a single reference.

To determine the number of page faults for a particular reference string and a page replacement algorithm, the number of frames available to the process needs to be known. As the number of frames available increases the number of page faults decreases. In the above illustration, if frames available were 3 then there would be only 3 page faults, one for each page reference. On the other hand, if there were only 1 frame available then there would be 11 page faults, one for every page reference.

8.5.1 FIFO page replacement algorithm

The first-in-first-out page replacement algorithm is the simplest page replacement algorithm. When a page replacement is required the oldest page in memory is the victim.

Illustration:

Reference string: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

Memory frames: 3

Page faults:	7	7	7	2	2	2	4	4	4	0	0	0	7	7	7
		0	0	0	3	3	3	2	2	2	1	1	1	0	0
			1	1	1	0	0	0	3	3	3	2	2	2	1

Number of page faults = 15.

The performance of the FIFO algorithm is not always good. The replaced page may have an initialization module that needs to be executed only once and therefore no longer needed. On the other hand, the page may have a heavily used variable in constant use. Such a page swapped out will cause a page fault almost immediately to be brought in. Thus, the number of page faults increases and results in slower process execution. Consider the following reference string:

Reference string: 1 2 3 4 1 2 5 1 2 3 4 5
Memory frames: 1, 2, 3, 4, 5

The chart below (Figure 8.5) gives the number of page faults generated for each of the 1, 2, 3, 4 and 5 memory frames available.

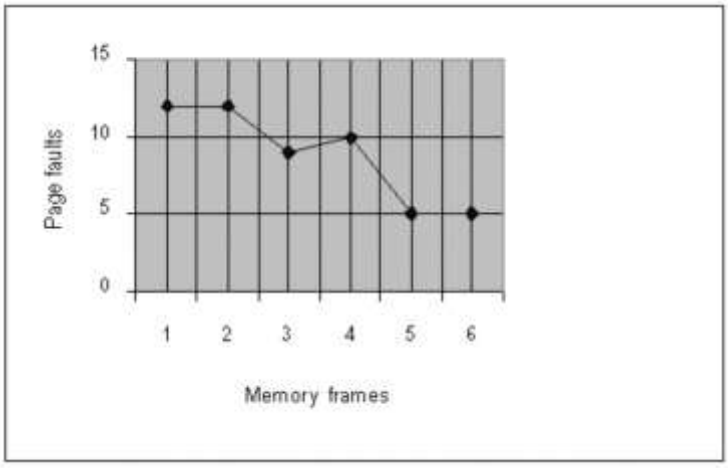


Fig. 8.5: Chart showing Page faults vs. Memory frames

As the number of frames available increases, the number of page faults must decrease. But the chart above shows 9 page faults when memory frames available are 3 and 10 when memory frames available are 4. This unexpected result is known as **Belady's anomaly**.

Implementation of FIFO algorithm is simple. A FIFO queue can hold pages in memory with a page at the head of the queue becoming the victim and the page swapped in joining the queue at the tail.

8.5.2 Optimal algorithm

An optimal page replacement algorithm produces the lowest page fault rate of all algorithms. The algorithm is to replace the page that will not be used for the longest period of time to come. Given a fixed number of memory frame by allocation, the algorithm always guarantees the lowest possible page fault rate and also does not suffer from Belady's anomaly.

Illustration:

Reference string: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

Memory frames: 3

Page faults:	7	7	7	2		2		2		2		7	
		0	0	0		0		4		0		0	
			1	1		3		3		3		1	

Number of page faults = 9.

Ignoring the first three page faults that do occur in all algorithms, the optimal algorithm is twice as better than the FIFO algorithm for the given string.

But implementation of the optimal page replacement algorithm is difficult since it requires future a priori knowledge of the reference string. Hence the optimal page replacement algorithm is more a benchmark algorithm for comparison.

8.5.3 LRU page replacement algorithm

The main distinction between FIFO and optimal algorithm is that the FIFO algorithm uses the time when a page was brought into memory (looks back) whereas the optimal algorithm uses the time when a page is to be used in future (looks ahead). If the recent past is used as an approximation of the near future, then replace the page that has not been used for the longest period of time. This is the least recently used (LRU) algorithm.

Illustration:

Reference string: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

Memory frames: 3

Page faults:	7	7	7	2		2		4	4	4	0		1		1		1
		0	0	0		0		0	0	3	3		3		0		0
			1	1		3		3	2	2	2		2		2		7

Number of page faults = 12.

The LRU page replacement algorithm with 12 page faults is better than the FIFO algorithm with 15 faults. The problem is to implement the LRU algorithm. An order for the frames by time of last use is required. Two options are feasible:

- By use of counters
- By use of stack

In the first option using counters, each page table entry is associated with a variable to store the time when the page was used. When a reference to the page is made, the contents of the clock are copied to the variable in the page table for that page. Every page now has the time of last reference to it. According to the LRU page replacement algorithm the least recently used page is the page with the smallest value in the variable associated with the clock. Overheads here include a search for the LRU page and an update of the variable to hold clock contents each time a memory reference is made.

In the second option a stack is used to keep track of the page numbers. A page referenced is always put on top of the stack. Therefore the top of the stack is the most recently used page and the bottom of the stack is the LRU page. Since stack contents in between need to be changed, the stack is best implemented using a doubly linked list. Update is a bit expensive because of the number of pointers to be changed, but there is no necessity to search for a LRU page.

LRU page replacement algorithm does not suffer from Belady's anomaly. But both of the above implementations require hardware support since either the clock variable or the stack must be updated for every memory reference.

Self Assessment Questions

4. Whenever a page fault occurs instead of using any page replacement algorithms simply terminate the process. (True / False)
5. _____ algorithm is more a benchmark algorithm for comparison.
6. Generally when the number of frames available increases, the number of page faults should decrease. Instead if the page faults also increase, this unexpected result is called _____. (Pick the right option)
   a) Charlie's anamoly

b) Belady's anamoly
c) John's anamoly
d) Pascal's anamoly

8.6 Thrashing

When a process does not have enough frames or when a process is executing with a minimum set of frames allocated to it which are in active use, there is always a possibility that the process will page fault quickly. The page in active use becomes a victim and hence page faults will occur again and again. In this case a process spends more time in paging than executing. This high paging activity is called thrashing.

8.6.1 Causes for thrashing

The operating system closely monitors CPU utilization. When CPU utilization drops below a certain threshold, the operating system increases the degree of multiprogramming by bringing in a new process to increase CPU utilization. Let a global page replacement policy be followed. A process requiring more frames for execution page faults and steals frames from other processes which are using those frames. This causes the other processes also to page fault. Paging activity increases with longer queues at the paging device but CPU utilization drops. Since CPU utilization drops, the job scheduler increases the degree of multiprogramming by bringing in a new process. This only increases paging activity to further decrease CPU utilization. This cycle continues. Thrashing has set in and throughput drops drastically. This is illustrated in the figure 8.6.

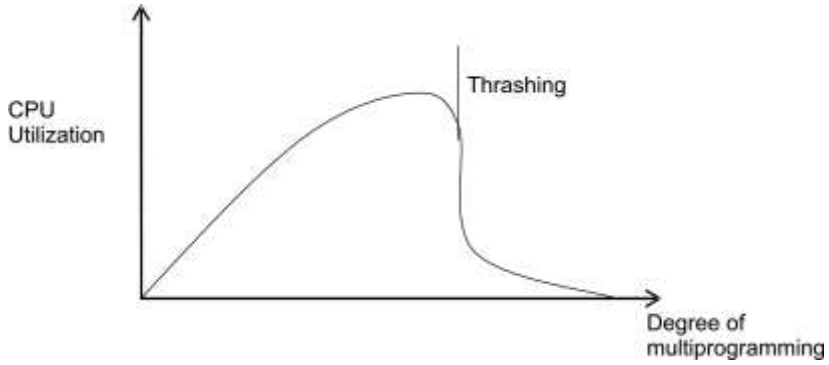


Fig. 8.6: Thrashing

When a local page replacement policy is used instead of a global policy, thrashing is limited to a process only. To prevent thrashing, a process must be provided as many frames as it needs. A working-set strategy determines how many frames a process is actually using by defining what is known as a locality model of process execution.

The locality model states that as a process executes, it moves from one locality to another, where a locality is a set of active pages used together. These localities are strictly not distinct and overlap. For example, a subroutine call defines a locality by itself where memory references are made to instructions and variables in the subroutine. A return from the subroutine shifts the locality with instructions and variables of the subroutine no longer in active use. So localities in a process are defined by the structure of the process and the data structures used therein. The locality model states that all programs exhibit this basic memory reference structure. Allocation of frames enough to hold pages in the current locality will cause faults for pages in this locality until all the required pages are in memory. The process will then not page fault until it changes locality. If allocated frames are less than those required in the current locality then thrashing occurs because the process is not able to keep in memory actively used pages.

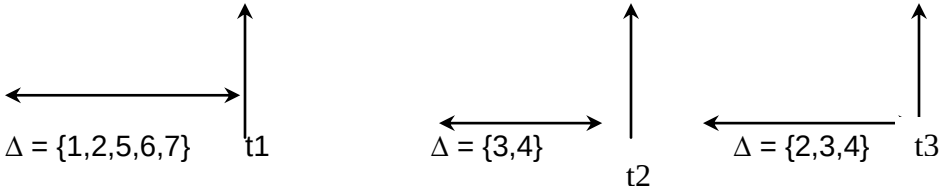
8.6.2 Working set model

The working set model is based on the locality model. A working set window is defined. It is a parameter Δ that maintains the most recent Δ page references. This set of most recent Δ page references is called the working set. An active page always finds itself in the working set. Similarly a page not used will drop off the working set Δ time units after its last reference. Thus the working set is an approximation of the program's locality.

Illustration:

Pages referenced:

…. 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 3 4 4 4 3 4 4 4 ….



If Δ = 10 memory references then a working set {1,2,5,6,7} at t_1 has changed to {3,4} at t_2 and {2,3,4} at t_3. The size of the parameter Δ defines the working set. If too small it does not consist of the entire locality. If it is too big then it will consist of overlapping localities.

Let WSS_i be the working set for a process P_i. Then $D = \sum WSS_i$ will be the total demand for frames from all processes in memory. If total demand is greater than total available, that is, D>m then thrashing has set in.

The operating system thus monitors the working set of each process and allocates to that process enough frames equal to its working set size. If free frames are still available then degree of multi-programming can be increased. If at any instant D>m then the operating system swaps out a process to decrease the degree of multi-programming so that released frames could be allocated to other processes. The suspended process is brought in later and restarted.

The working set window is a moving window. Each memory reference appears at one end of the window while an older reference drops off at the other end.

The working set model prevents thrashing while the degree of multi-programming is kept as high as possible there by increasing CPU utilization.

8.6.3 Page fault frequency

One other way of controlling thrashing is by making use of the frequency of page faults. This page fault frequency (PFF) strategy is a more direct approach.

When thrashing has set in page fault is high. This means to say that a process needs more frames. If page fault rate is low, the process may have more than necessary frames to execute. So upper and lower bounds on page faults can be defined. If the page fault rate exceeds the upper bound, then another frame is allocated to the process. If it falls below the lower bound then a frame already allocated can be removed. Thus monitoring the page fault rate helps prevent thrashing.

As in the working set strategy, some process may have to be suspended only to be restarted later if page fault rate is high and no free frames are available so that the released frames can be distributed among existing processes requiring more frames.

Self Assessment Questions

7. The operating system closely monitors CPU utilization. (True / False)
8. _____ is a high paging activity in which a process spends more time in paging than executing.
9. PFF stands for _____. (Pick the right option)
   a) Page Fault Finder
   b) Page Finding Frequency
   c) Page Fault Frequency
   d) Page Fault Finding

8.7 Summary

Let's summarize the key points:

- In this chapter we have studied a technique called virtual memory that creates an illusion for the user that he/she has a large memory at his/her disposal. But in reality, only a limited amount of main memory is available and that too is shared amongst several users.
- We have also studied demand paging which is the main concept needed to implement virtual memory.
- Demand paging brought in the need for page replacements if required pages are not in memory for execution.

8.8 Terminal Questions

1. What is virtual memory? Distinguish between logical address and physical address.
2. Explain demand paging in virtual memory system.
3. Explain Belady's anomaly with the help of FIFO page replacement algorithm.
4. Discuss the Optimal Page replacement algorithm.
5. What is thrashing and what is its cause?

8.9 Answers

Self Assessment Questions

1. True
2. Virtual Memory
3. a) Demand Paging
4. False

5. Optimal Page Replacement
6. b) Belady's Anamoly
7. True
8. Thrashing
9. c) Page Fault Frequency

Terminal Questions

1. Every process needs to be loaded into physical memory for execution. One brute force approach to this is to map the entire logical space of the process to physical memory, as in the case of paging and segmentation. (Refer Section 8.2 for detail)
2. Demand paging is similar to paging with swapping. When a process is to be executed then only that page of the process, which needs to be currently executed, is swapped into memory. (Refer Section 8.3)
3. The first-in-first-out page replacement algorithm is the simplest page replacement algorithm. When a page replacement is required the oldest page in memory is the victim. (Refer Sub-section 8.5.1)
4. An optimal page replacement algorithm produces the lowest page fault rate of all algorithms. The algorithm is to replace the page that will not be used for the longest period of time to come. (Refer Sub-section 8.5.2)
5. When a process does not have enough frames or when a process is executing with a minimum set of frames allocated to it which are in active use, there is always a possibility that the process will page fault quickly. The page in active use becomes a victim and hence page faults will occur again and again. In this case a process spends more time in paging than executing. This high paging activity is called thrashing. (Refer Section 8.6)