

Virtual –Memory Management

Virtual Memory



1. What is virtual memory? How is it implemented? What are its benefits? [7]
2. What is virtual memory concept? [1]
3. Describe demand paging system. [4]

Virtual memory is a technique that allows the execution of processes that are not completely in memory. One major advantage of this scheme is that programs can be larger than physical memory.

An examination of real programs shows us that, in many cases, the entire program (in memory) is not needed.

- Programs often have code to handle unusual error conditions (seldom used).
- Arrays, lists, and tables are often allocated more memory than they actually need.
- Certain options and features of a program may be used rarely.

The ability to execute a program that is only partially in memory would offer many benefits:

- A program would no longer be constrained by the amount of physical memory that is available (simplifying the programming task).
- Because each user program could take less physical memory, more programs could be run at the same time, with a corresponding increase in CPU utilization and throughput with no increase in response time or turnaround time.
- Less I/O would be needed to load or swap each user program into memory, so each user program would run faster.

Virtual memory is commonly implemented by **demand paging**. The basic idea behind demand paging is that when a process is swapped in, its pages are not swapped in all at once. Rather they are swapped in only when the process needs them (On demand).

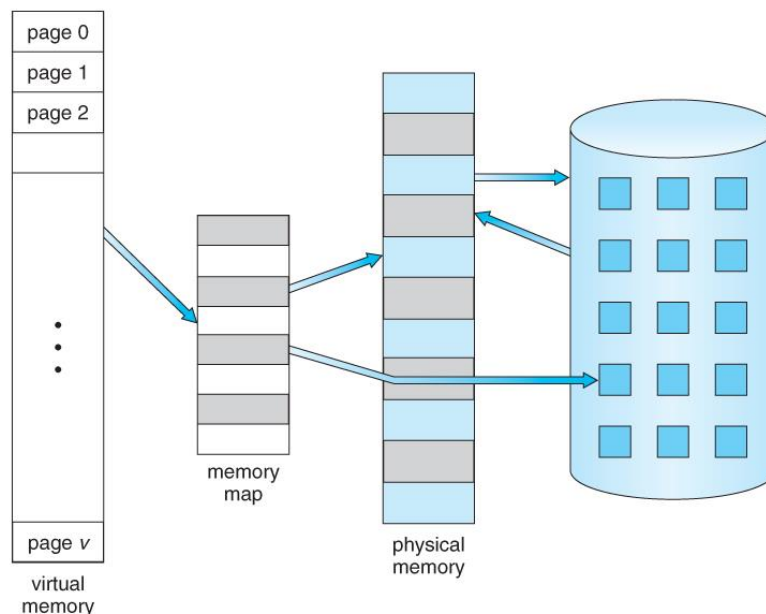


Figure: Diagram showing virtual memory that is larger than physical memory.

In the above scenario, only those pages of the process that are needed for execution are loaded into memory, the other pages reside on the disk. Since only a subset of the process pages are loaded into memory, processes with size larger than the physical memory can easily accommodate in the main memory and execute successfully to completion

Virtual memory **also allows the sharing of files and memory** by multiple processes, with several benefits:

- System libraries can be shared by mapping them into the virtual address space of more than one process.
- Processes can also share virtual memory by mapping the same block of memory to more than one process.
- Process pages can be shared during a `fork()` system call, eliminating the need to copy all of the pages of the original (parent) process.

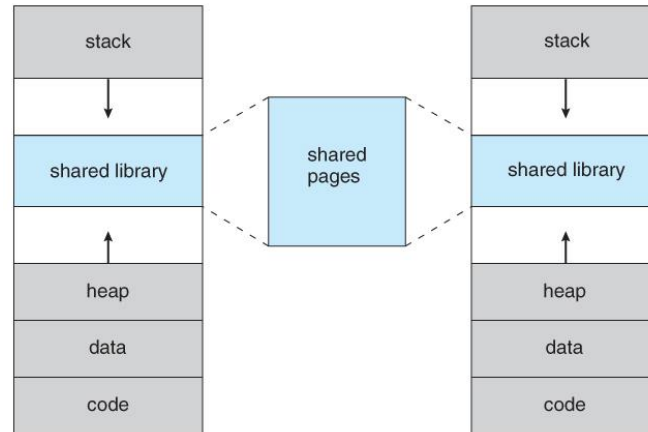

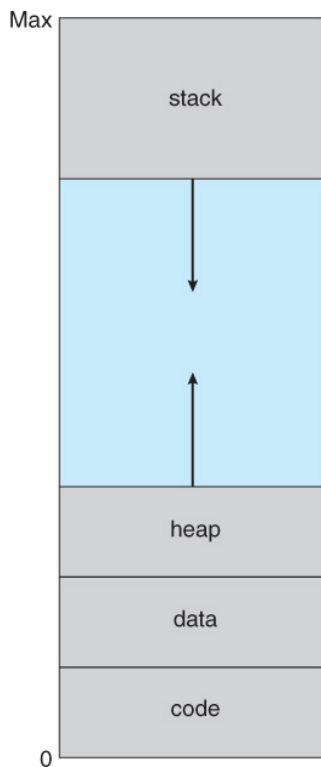


Figure: Shared library using virtual memory.

A **virtual address space** of a process refers to the logical view of how a program is stored in memory. Typically, this view is that a process begins at a certain logical address-say, address 0-and exists in contiguous memory, as shown  in Figure

The heap grows upward in memory as it is used for dynamic memory allocation. Similarly, stack grows downward in memory through successive function calls. The large blank space (or hole) between the heap and the stack is part of the virtual address



Demand Paging

The basic idea behind **demand paging** is that when a process is swapped in, its pages are not swapped in all at once. Rather they are swapped in only when the process needs them. (on demand.) This is termed a **lazy swapper**, although a **pager** is a more accurate term.

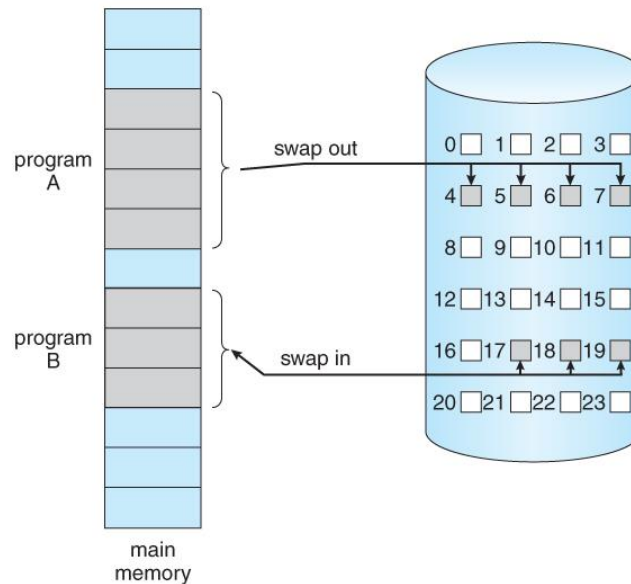


Figure: Transfer of a paged memory to contiguous disk space.

Basic Idea:

- The basic idea behind paging is that when a process is swapped in, the pager only loads into memory those pages that it expects the process to need (right away.)
- Pages that are not loaded into memory are marked as invalid in the page table, using the invalid bit. (The rest of the page table entry may either be blank or contain information about where to find the swapped-out page on the hard drive.)
- When this bit is set to “valid”, the associated page is both legal and in memory. If the bit is set to “invalid”, the page either is not valid (that is, not in the logical address space of the process) or is valid but is currently on the disk.

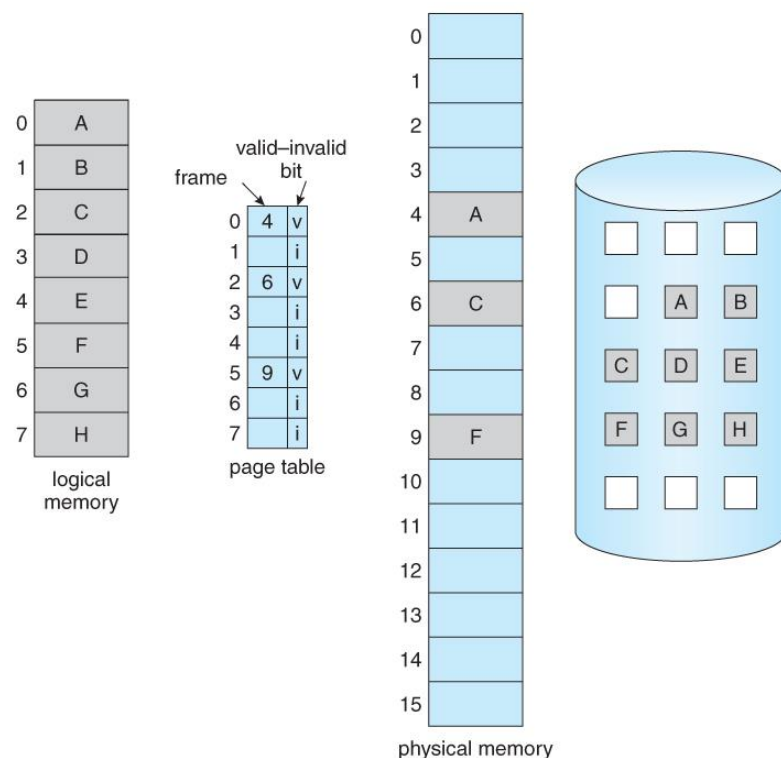


Figure : Page table when some pages are not in main memory.

Page Faults



1. Discuss the steps in handling page fault with the help of a neat diagram [10]

- While the process executes and accesses pages that are memory resident, execution proceeds normally.
- On the other hand, if the requested page is not loaded in memory (marked as invalid), then a **page fault trap** is generated, which must be handled in a series of steps:
 1. The memory address requested is first checked, to make sure it was a valid memory request. Every page in the page table will be marked as valid/invalid.
 - ‘*valid*’ indicates a valid page that belongs to the process and is loaded into memory.
 - ‘*invalid*’ indicates that either the page does not belong to the process or that it is not loaded into memory
 2. An invalid page reference **results in a trap** to the operating system. The Operating system determines whether the trap was because of illegal access to memory or because the page was not loaded into memory. If the reference was illegal, the process is terminated. Otherwise, the OS initiates necessary steps to load the required page into memory.
 3. A free frame is located, possibly from a free-frame list.
 4. A disk operation is scheduled to bring in the necessary page from disk.
 5. When the I/O operation is complete, the *process's page table is updated* with the new frame number, and the invalid bit is changed to indicate that this is now a valid page reference.
 6. The *instruction that caused the page fault must now be restarted* from the beginning,

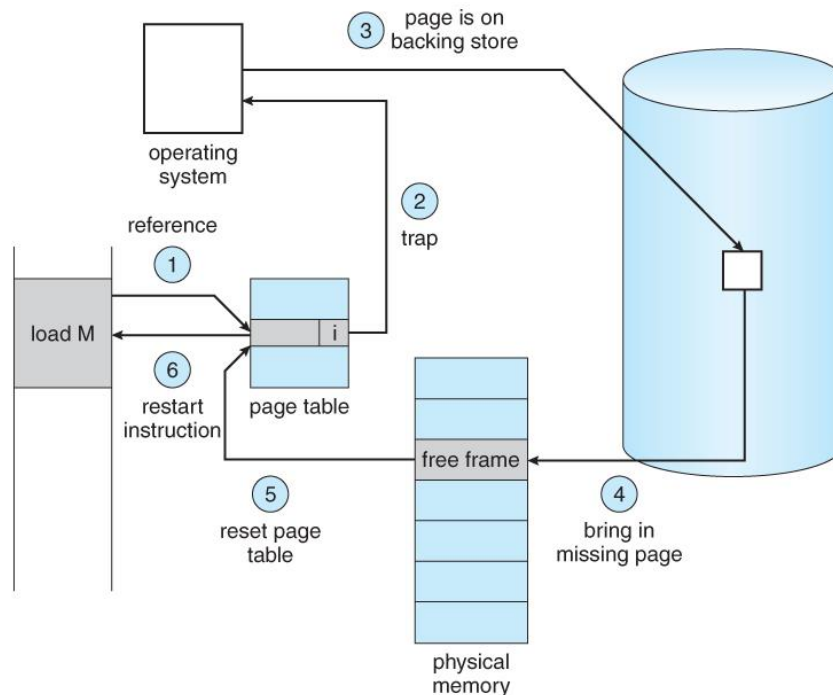


Figure: Steps in handling a page fault.

Pure Demand Paging

- In the extreme case, we can start executing a process with no pages in memory.
- When the OS sets the instruction pointer to the first instruction of the process, which is on a non-memory-resident page, the process immediately faults for the page.
- After this page is brought into memory, the process continues to execute, faulting as necessary until every page that it needs is in memory.
- This scheme is **pure demand paging**: Never bring a page into memory until it is required.

The hardware support for demand paging is same as hardware support for paging and swapping. We need

- **Page table** – An ability to mark a page as invalid through a valid -invalid bit or a special value of protection bits
- **Secondary Memory** – A high speed disk known as *swap device* to hold the pages that are not present in the main memory.

Performance of Demand Paging



1. Discuss on the performance of demand paging [5]
2. Given page fault service time of 8 milli sec, memory access time 100 nano sec. page fault rate 0.0002. Calculate effective access time [6]
3. On a system using demand paging, it takes 12 microsec to satisfy a memory request, if the page is not in memory, the request takes 5000 micro sec, What would the page fault rate need to be to achieve effective access time 1000 micro sec. Assume the system runs only a single process and the CPU is idle during page swaps [8]

Demand paging can significantly affect the performance of a computer system. Whenever the requested page is not in memory, a page fault occurs and necessary steps have to be taken to arrange for a free frame in memory and load the requested page into that frame.

In any case, we are faced with three major components of the page-fault service time:

- *Service the page-fault interrupt.*
- *Read in the page.*
- *Restart the process.*

Effective access time for a demand-paged memory-

- For most computer systems, the memory-access time, denoted ***ma***, ranges from 10 to 200 nanoseconds.
- As long as we have no page faults, the effective access time is equal to the memory access time.
- If a page fault occurs, we must first read the relevant page from disk and then access the desired word.

Let ***p*** be the ***probability of a page fault*** ($0 \leq p \leq 1$).

We would expect ***p*** to be close to zero -that is, we would expect to have only a few page faults. The effective access time is then

Effective Access Time = $(1 - p) \times ma + p \times \text{page fault time}$

With an average page-fault service time of 8 milliseconds and a memory access time of 200 nanoseconds, the effective access time in nanoseconds is

$$\begin{aligned}\text{effective access time} &= (1 - p) \times (200) + p (8 \text{ milliseconds}) \\ &= (1 - p) \times 200 + p \times 8,000,000 \\ &= 200 + 7,999,800 \times p.\end{aligned}$$

If one access out of 1,000 causes a page fault, probability of page fault is 0.001, the effective access time = $200 + 7999800 \times 0.001 = 8199 \text{ ns} = 8.199 \text{ microseconds}$. The computer will be slowed. down by a factor of 40 [8.199 microseconds/200 ns] because of demand paging!

If we want performance degradation to be less than 10% (i.e. computed effective access time should be less than 220 ns), then probability of page fault should be less than 0.0000025

$$220 > 200 + 7999800 * p$$

$$20 > 7999800 * p$$

$$\frac{20}{7999800} > p$$

$$p < 0.0000025$$

It is very important to keep the page-fault rate low in a demand-paging system. Otherwise, the effective access time increases, slowing process execution dramatically

We see, then, that the effective access time is directly proportional to the page fault rate.

Copy-on-Write



What do you mean by copy-on-write? Where is it used? Explain in brief [4]

- The fork() system call creates a child process as a duplicate of its parent, i.e. a copy of all the pages belonging to the parent process is created for the child. However, if the child processes exec() system call immediately after creation, the copying of the parent's address space may be unnecessary.
-
- The idea behind a copy-on-write fork is that the pages for a parent process do not have to be actually copied for the child until one or the other of the processes changes the page. They *can be simply shared between* the two processes in the meantime, with *a bit set* that the page needs to be copied if it ever gets written to. This is a reasonable approach, since the child process usually issues an exec() system call immediately after the fork.

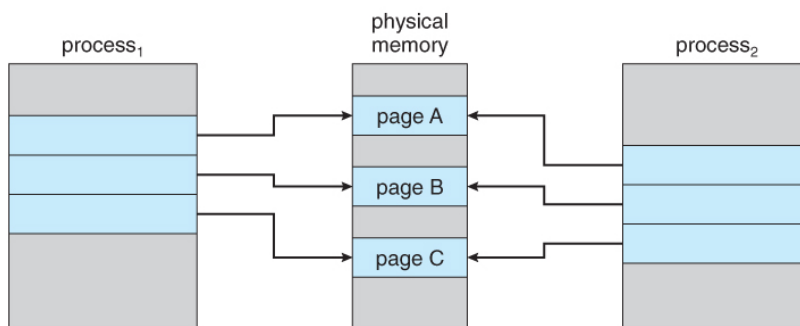


Figure 9.7 - Before process 1 modifies page C.

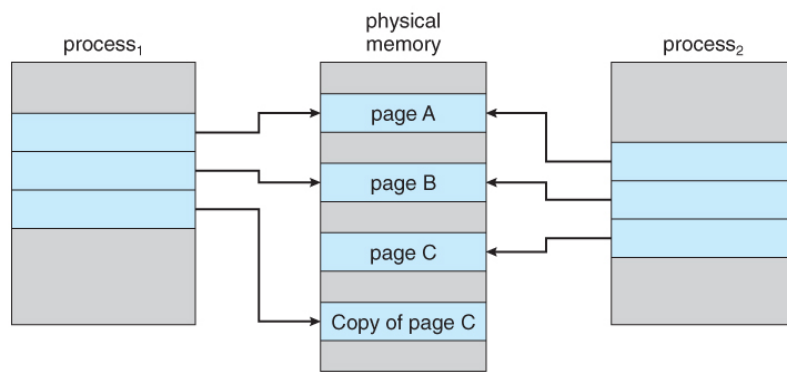


Figure 9.8 - After process 1 modifies page C

These shared pages are marked as copy-on-write pages, meaning that if either process writes to a shared page, a copy of the shared page is created. Only pages that can be modified need be marked as copy-on-write. Pages that cannot be modified (pages containing executable code) can be shared by the parent and child.

Some systems provide an alternative to the `fork()` system call called a virtual memory fork, ***vfork()***. In this case the parent is suspended, and the child uses the parent's memory pages. This is very fast for process creation, but requires that the child not modify any of the shared memory pages before performing the `exec()` system call. With `vfork`, the parent is suspended, allowing the child to execute first until it calls `exec()`, sharing pages with the parent in the meantime.

Page Replacement

In order to make the most use of virtual memory, we load several processes into memory at the same time. Since we only load the pages that are actually needed by each process at any given time, there is room to load many more processes than if we had to load in the entire process.

What happens if some process suddenly decides it needs more pages and there aren't any free frames available? There are several possible solutions to consider:

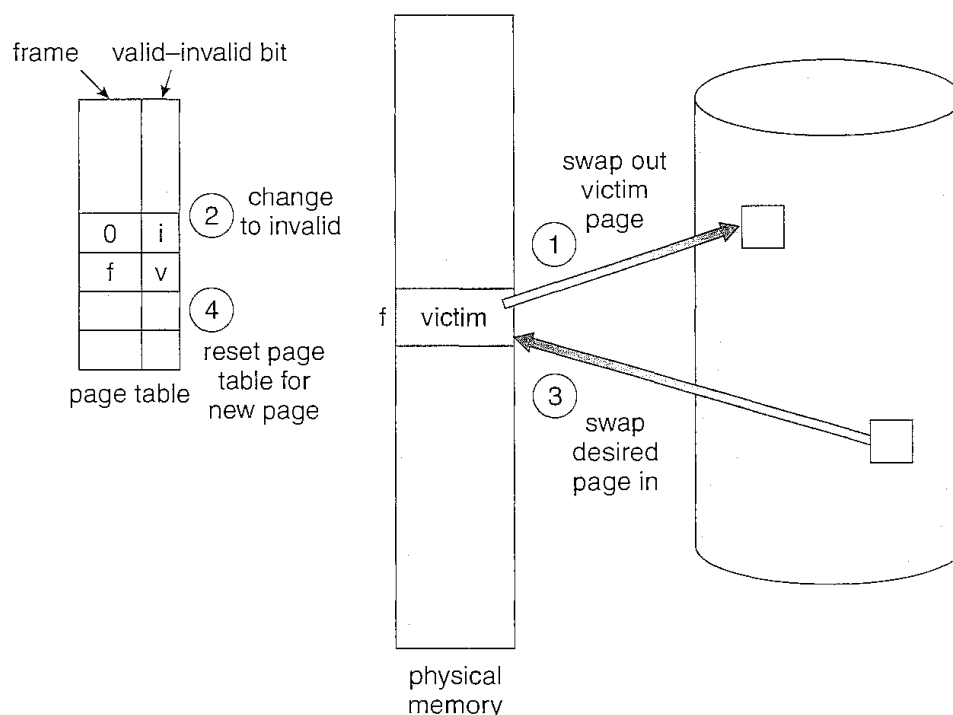
- Put the process requesting more pages into a wait queue until some free frames become available.
- Swap some process out of memory completely, freeing up its page frames.
- Find some page in memory that isn't being used right now, and swap that page only out to disk, freeing up a frame that can be allocated to the process requesting it. This is known as **page replacement**, and is the most common solution.

Basic Page Replacement

The previously discussed page-fault processing assumed that there would be free frames available on the free-frame list. Now the page-fault handling must be modified to free up a frame if necessary, as follows:

1. Find the location of the desired page on the disk, either in swap space or in the file system.
2. Find a free frame:
3. If there is a free frame, use it.
4. If there is no free frame, use a page-replacement algorithm to select an existing frame to be replaced, known as the victim frame.
5. Write the victim frame to disk. Change all related page tables to indicate that this page is no longer in memory.

6. Read in the desired page and store it in the frame. Adjust all related page and frame tables to indicate the change.
7. Restart the instruction that was waiting for this page.



=====No free frames → page fault service time is doubled=====

We can 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. This overhead can be reduced by assigning a **modify bit, or dirty bit** to each page, indicating whether or not it has been changed since it was last loaded in from disk.

If the dirty bit has not been set, then the page is unchanged, and does not need to be written out to disk. Otherwise the page write is required. The page replacement strategies can now look for pages that do not have their dirty bit set (no need to copy these pages to disk), and preferentially select clean pages as victim pages.

There are two major requirements to implement a successful demand paging system → a *frame-allocation algorithm* and a *page-replacement algorithm*.

The former centers around how many frames are allocated to each process and the latter deals with how to select a page for replacement when there are no free frames available.

The overall goal in selecting and tuning these algorithms is to generate the fewest number of overall page faults. Because disk access is so slow relative to memory access, even slight improvements to these algorithms can yield large improvements in overall 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**.

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

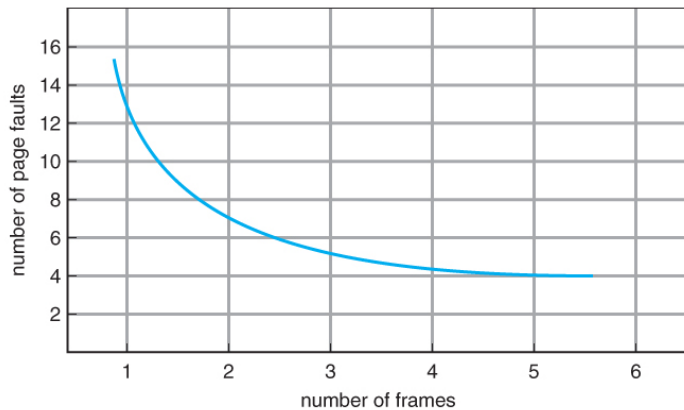
Since an entire page is loaded into memory, we are only interested in the page number and not the entire address. In the above example, considering a system with 100 bytes per page, the address sequence can be reduced to the given reference string

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

Which can be further reduced to

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

In general, as the number of available frames increases, the number of page faults should decrease, as shown in Figure:



1. Consider the following reference string. 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1. Assuming 3 frames, all initially empty, how many page faults would occur for LRU, FIFO, Optimal page replacement algorithm? Which algorithm is the most efficient in this case? [12]
2. Consider the reference string 1, 2, 3, 5, 2, 3, 5, 7, 2, 1, 2, 3, 8, 6, 4, 3, 2, 2, 3, 6. How many page faults would occur in case of LRU, FIFO, and Optimal page replacement algorithm? Assume 3 frames all of which are empty initially. [8]
3. Same question as above : Ref String - 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1. Assume 3 frames
4. List the various page replacement algorithms. Explain any one with an example of your own [6]
5. Explain LRU page replacement algorithm [5]

FIFO Page Replacement

Basic Idea: Choose the oldest page for replacement

Consider the following reference string and a demand paging system with 3 frames:

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

Blank columns indicate 'no page faults'

In the above example 15 page faults occur when accessing a reference string of 20 pages.

Page fault rate = $15/20 = 0.75$.

Belady's Anomaly



1. What is Belady's anomaly? Explain with an example [5]

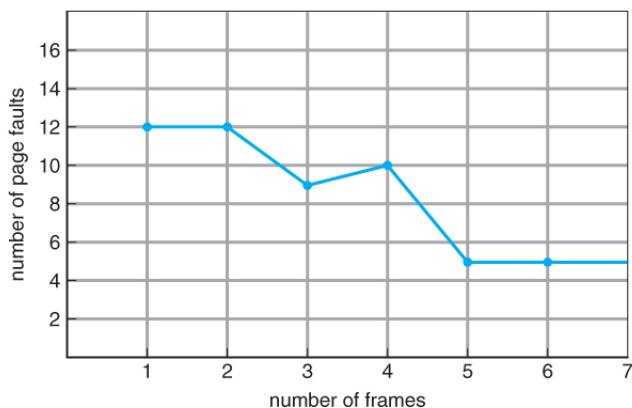
Usually, on increasing the number of frames allocated to a process virtual memory, the process execution is faster, because fewer page faults occur. Sometimes, the reverse happens, i.e., the execution time increases or remains constant even when more frames are allocated to the process. This is Belady's Anomaly.

Belady's anomaly is usually observed in FIFO page replacement scheme.

NOTE: Use your own example to demonstrate how increasing the number of frames in FIFO algorithm may not always reduce the number of page faults.

Drawing the graph of page faults vs no of frames at the end may help to demonstrate the concept

Ex:



Optimal Page Replacement

Basic idea: Replace the page that will not be used for the longest period of time (in the future)

Consider the same reference string and a demand paging system with 3 frames:

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

With optimal page replacement, the number of page faults = 9

Page fault rate = $9/20 = 0.45$

Optimal page replacement algorithm has the lowest page-fault rate of all algorithms and will never suffer from Belady's anomaly. Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string.

Least Recently Used [LRU] Page Replacement

Basic Idea: choose the page that has not been used (referenced) for the longest period of time

Consider the same reference string and a demand paging system with 3 frames:

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

With optimal page replacement, the number of page faults = 12. Page fault rate = $12/20 = 0.6$.

LRU is considered a good replacement policy, and is often used. The problem is how exactly to implement it i.e. *how to determine an order for the frames defined by the time of last use* There are two simple approaches commonly used:

1. **Counters.** Every memory access increments a counter, and the current value of this counter is stored in the page table entry for that page. Then finding the LRU page involves simple *searching the table for the page with the smallest counter value*. Note that overflowing of the counter must be considered.
2. **Stack.** Another approach is to use a stack, and *whenever a page is accessed, pull that page from the middle of the stack and place it on the top. The LRU page will always be at the bottom of the stack*. Because this requires removing objects from the middle of the stack, a doubly linked list is the recommended data structure.

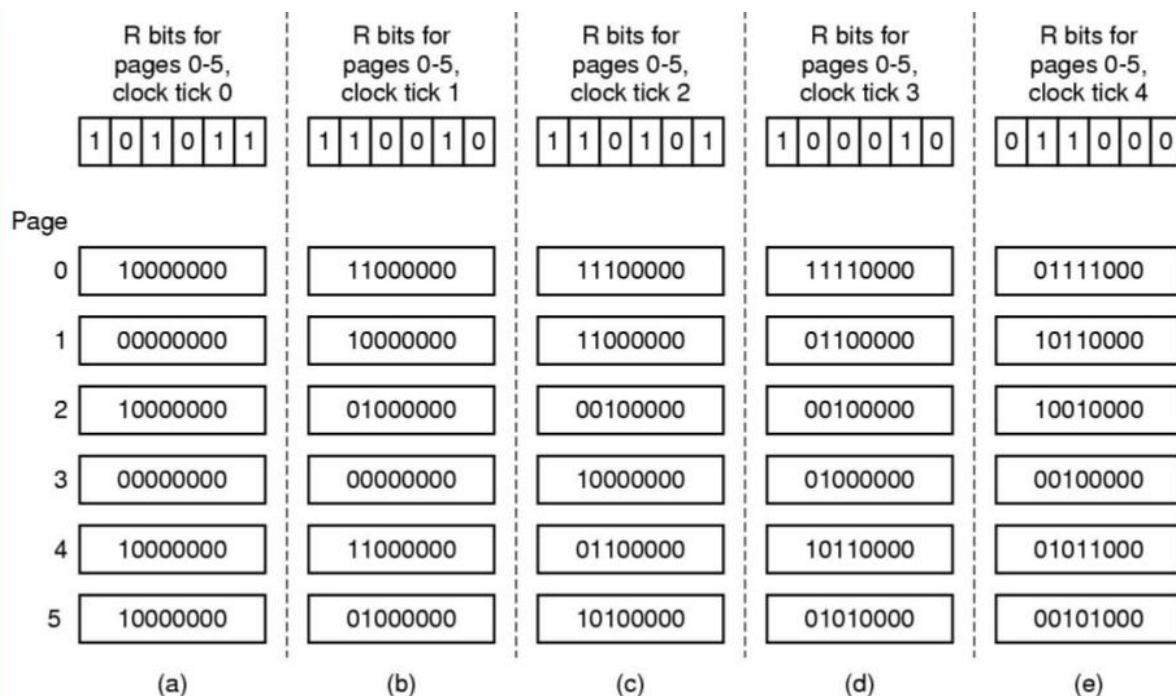
LRU Approximation algorithms

Note that both implementations of LRU require hardware support, either for incrementing the counter or for managing the stack, as these operations must be performed for every memory access.

However many systems offer some degree of HW support, enough to approximate LRU fairly well. In particular, many systems provide a **reference bit** for every entry in a page table, which is set anytime that page is accessed. Initially all bits are set to zero, and they can also all be cleared at any time. One bit of precision is enough to *distinguish pages that have been accessed since the last clear from those that have not* but it does not specify the order in which the pages were accessed.

Additional-Reference-Bits Algorithm

- Finer grain of detail can be obtained by storing the most recent 8 reference bits for each page in an 8-bit byte in the page table entry.

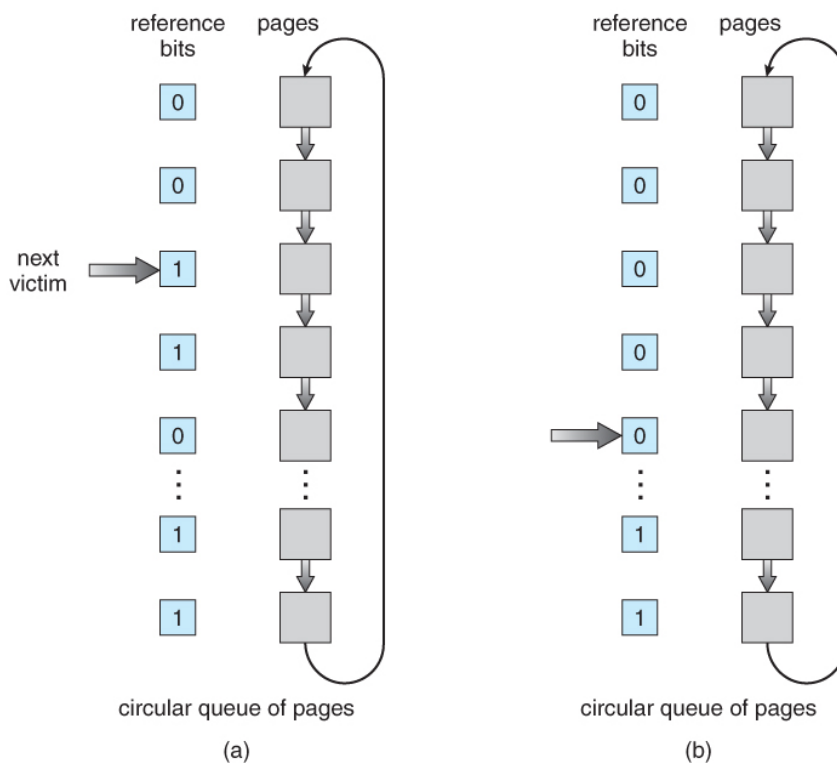


- At periodic intervals (clock interrupts), the OS takes over, and right-shifts each of the reference bytes by one bit.

- The high-order bit is then filled in with the current value of the reference bit, and the reference bits are cleared.
- At any given time, *the page with the smallest value for the reference byte is the LRU page*.

Second-Chance Algorithm

- The second chance algorithm is essentially a FIFO, except the *reference bit is used to give pages a second chance at staying in the page table*.
- When a page must be replaced, the page table is scanned in a FIFO (circular queue) manner. If a page is found with its reference bit not set (un-modified page), then that page is selected as the next victim.
- If, however, the next page in the FIFO does have its reference bit set, then it is given a second chance:
 - The reference bit is cleared, and the FIFO search continues.
 - If some other page is found that did not have its reference bit set, then that page will be selected as the victim, and this page (the one being given the second chance) will be allowed to stay in the page table.
 - If, however, there are no other pages that do not have their reference bit set, then this page will be selected as the victim when the FIFO search circles back around to this page on the second pass.



- If all reference bits in the table are set, then second chance degrades to FIFO, but also requires a complete search of the table for every page-replacement.
- As long as there are some pages whose reference bits are not set, then any page referenced frequently enough gets to stay in the page table indefinitely.
- This algorithm is also known as the clock algorithm, from the hands of the clock moving around the circular queue.

Enhanced Second-Chance Algorithm

The enhanced second chance algorithm looks at the reference bit and the modify bit (dirty bit) as an ordered page, and classifies pages into one of four classes:

(0, 0) - Neither recently used nor modified.

(0, 1) - Not recently used, but modified.

(1, 0) - Recently used, but clean.

(1, 1) - Recently used and modified.

This algorithm searches the page table in a circular fashion (in as many as four passes), looking for the first page it can find in the lowest numbered category. i.e. it first makes a pass looking for a (0, 0), and then if it can't find one, it makes another pass looking for a (0, 1), etc.

The main difference between this algorithm and the previous one is the *preference for replacing clean pages* if possible.

Counting-Based Page Replacement

There are several algorithms based on counting the number of references that have been made to a given page, such as:

- **Least Frequently Used, LFU:** *Replace the page with the lowest reference count.* A problem can occur if a page is used frequently initially and then not used any more, as the reference count remains high. A solution to this problem is to right-shift the counters periodically, yielding a time-decaying average reference count.
- **Most Frequently Used, MFU:** *Replace the page with the highest reference count.* The logic behind this idea is that pages that have already been referenced a lot have been in the system a long time, and we are probably done with them, whereas pages referenced only a few times have only recently been loaded, and we still need them.

In general counting-based algorithms are not commonly used, as their implementation is expensive and they do not approximate Optimal Page replacement well.

Page-Buffering Algorithms

There are a number of page-buffering algorithms that can be *used in conjunction with the afore-mentioned algorithms, to improve overall performance* and sometimes make up for inherent weaknesses in the hardware and/or the underlying page-replacement algorithms:

- **Maintain a certain minimum number of free frames at all times.** When a page-fault occurs, go ahead and allocate one of the free frames from the free list first, to get the requesting process up and running again as quickly as possible, and *then select a victim page to write to disk and free* up a frame as a second step.
- **Keep a list of modified pages,** and *when the I/O system is idle, write these pages out to disk,* and then clear the modify bits, thereby increasing the chance of finding a "clean" page for the next potential victim.
- **Maintain a pool of free frames, but remember what page was stored in it before it was made free.** Next time, the same page is requested, the free frame can be made active again

Allocation of Frames

The two important tasks in virtual memory management: a page-replacement strategy and a frame-allocation strategy.

9.5.1 Minimum Number of Frames

Every system dictates a minimum number of frames for each process. This number is usually the minimum number of pages that is required by the process for each instruction execution.

9.5.2 Allocation Algorithms

- **Equal Allocation** - If there are m frames available and n processes to share them, each process gets m / n frames, and the leftovers are kept in a free-frame buffer pool.
- **Proportional Allocation** - Allocate the frames proportionally to the size of the process, relative to the total size of all processes. So if the size of process i is S_i , and S is the sum of all S_i , the total number of available frames is m , then the allocation for process $i \rightarrow a_i = m * S_i / S$.

9.5.3 Global versus Local Allocation

Allocation of frames to processes can be done using local replacement or global replacement strategy.

- With local replacement, the number of pages allocated to a process is fixed, and page replacement occurs only amongst the pages allocated to this process.
- With global replacement, any page may be a potential victim, whether it currently belongs to the process seeking a free frame or not.

Local page replacement allows processes to better control their own page fault rates, and leads to more consistent performance of a given process over different system load levels.

Global page replacement is overall more efficient, and is the more commonly used approach.

9.5.4 Non-Uniform Memory Access

- The above arguments all assume that all memory is equivalent, or at least has equivalent access times.
- This may not be the case in multiple-processor systems, especially where each CPU is physically located on a separate circuit board which also holds some portion of the overall system memory.
- On such systems, CPUs can access memory that is physically located on the same board much faster than the memory on the other boards.
 - The basic solution is similar to **processor affinity** – Just as we try to schedule processes on the same CPU to minimize cache misses, we also try to allocate memory for those processes on the same boards, to minimize access times.

Thrashing



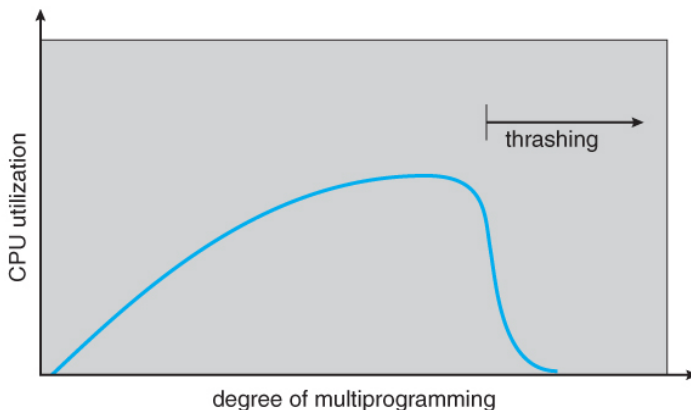
1. What is thrashing? How does the system detect thrashing? [4]
2. **What is Thrashing? How can it be controlled?** [6]
3. Describe thrashing [5]
4. Define [3]
 - a. Thrashing
 - b. Belady's anomaly
 - c. Effective Access time in demand paging
5. What is thrashing? Explain how working set model can be used to solve the same [6]

Thrashing is a condition or a situation when the system is spending a major portion of its time in servicing the page faults, but the actual processing done is very negligible.

If a process is allocated too few frames, then there will be too many and too frequent page faults. As a result, no useful work would be done by the CPU and the CPU utilisation would fall drastically. The long-term scheduler would then try to improve the CPU utilisation by loading some more processes into the memory thereby increasing the degree of multiprogramming.

Cause of Thrashing

- Early process scheduling schemes would control the level of multiprogramming allowed based on CPU utilization, adding in more processes when CPU utilization was low.
- The problem is that when memory filled up and processes started spending lots of time waiting for their pages to page in, then CPU utilization would lower, causing the scheduler to add in even more processes so as to increase CPU utilization



This increased number of processes would result in some more page faults on an already degraded system, thus reducing the performance of the system drastically.

=====How to reduce thrashing?=====

To prevent thrashing we must provide processes with as many frames as they really need "right now"

Methodology for solving thrashing

Approach 1: working set

We know that to prevent thrashing we must provide processes with as many frames as they really need "right now". The number of frames needed by the process right now is defined by the working set.

Informally *Working Sets* refers to the collection of pages a process is using actively, and which must be memory-resident to prevent this process from thrashing

The **working set model** is based on the concept of locality, and defines a **working set window**, of length **delta**.

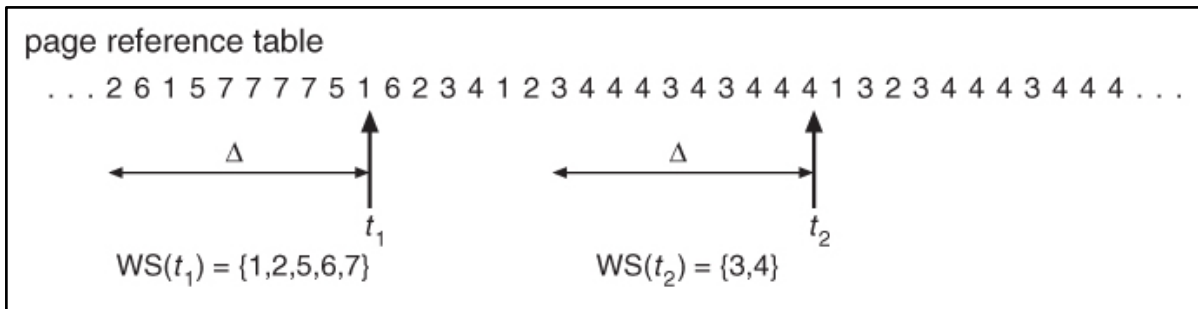


Figure 37: Working-set model

For example, given the sequence of memory references shown in figure, if $\Delta = 10$ memory references, then the working set at time t_1 is $\{1, 2, 5, 6, 7\}$. By time t_2 , the working set has changed to $\{3, 4\}$.

The accuracy of working set depends on the selection of Δ . The most important property of the working set is its size. If the working set size is WSS_i for each process, then

$$D = \sum WSS_i$$

Where D is the total demand for frames.

If the total demand is greater than the total number of available frames (m), thrashing will occur.

Hence, If $D > m$, suspend some processes, create free frames in the memory

Approach 2: page fault frequency

- Since Thrashing has a high page-fault rate, we need to control that
- We can establish upper and lower bounds on the desired page-fault rate

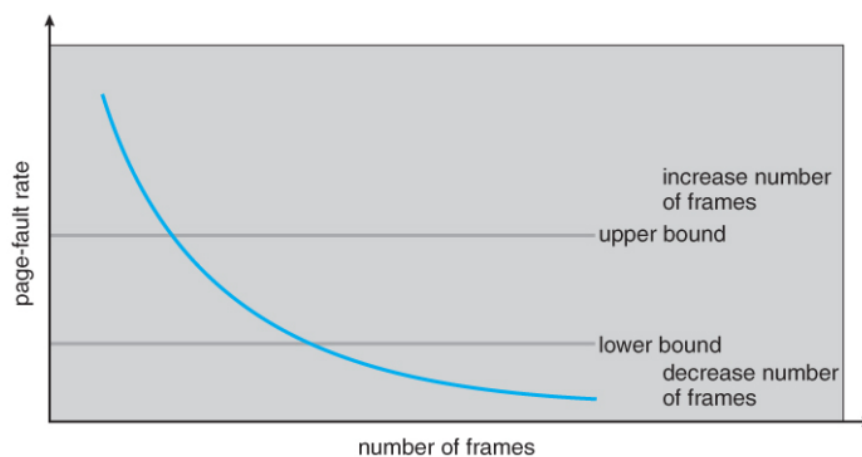


Figure 38: Page-fault frequency

- If the actual page fault rate exceeds the upper limit, we allocate the process another frame. If the page fault rate falls below the lower limit, we remove a frame from the process. Thus we can directly measure and control the page fault rate to prevent thrashing.