UNIT 4

VIRTUAL MEMORY

- It is a technique that allows the **execution of processes that may not be completely** in main memory.
- Virtual memory involves **the separation of logical memory as perceived** by users from physical memory (figure 4.1)
- Virtual memory makes the task of programming much easier, because the programmer no longer needs to worry about the amount of physical memory available.
  - **Advantages:**
    - ✓ Allows the program that can be larger than the physical memory.
    - ✓ Separation of user logical memory from physical memory
    - ✓ Allows processes to easily share files & address space.
    - ✓ Allows for more efficient process creation.
  - Virtual memory can be implemented using
    - ✓ Demand paging
    - ✓ Demand segmentation

Virtual Memory That is Larger than Physical Memory

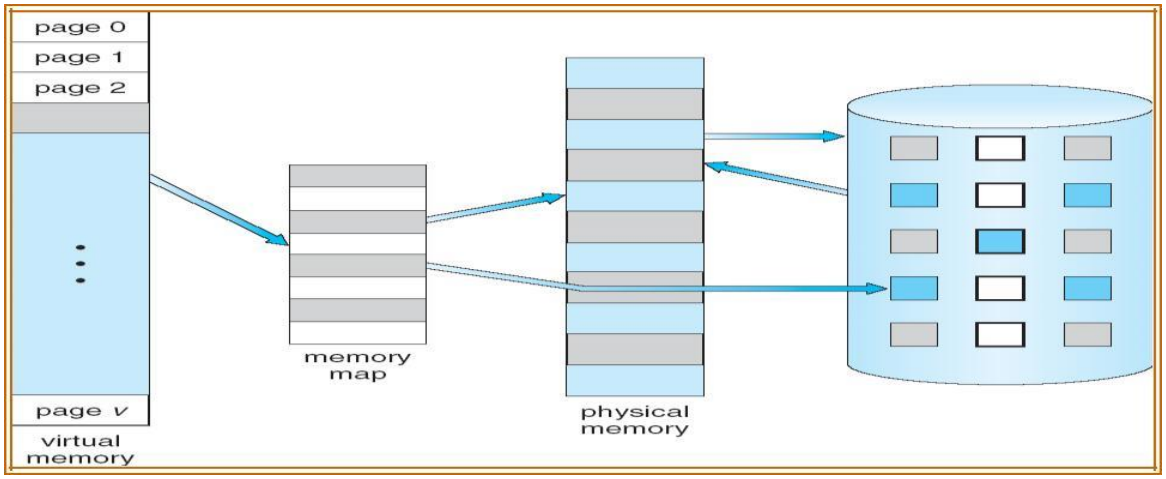


Fig 4.1: Diagram showing virtual memory that is larger than physical memory

Demand Paging

- It is similar to a paging system with swapping.
- Demand Paging - Bring a page into memory only when it is needed
- To execute a process, need to swap entire process into memory. Rather than swapping the entire process into memory however, we use ―Lazy Swapper that swaps the pages when they are demanded during program execution.
- **Lazy Swapper -** Never swaps a page into memory unless that page will be needed.
- **Advantages**
  - ✓ Less I/O needed
  - ✓ Less memory needed

- ✓ Faster response
- ✓ More users

Basic Concepts:

- Instead of swapping in the whole processes, the pager brings only those necessary pages into memory. Thus,
  1. It avoids reading into memory pages that will not be used anyway.
  2. Reduce the swap time.
  3. Reduce the amount of physical memory needed.
- To differentiate between those pages that are in memory & those that are on the disk we use the **Valid-Invalid bit**.

Valid-Invalid bit

- Valid – the associated page is both legal and in memory.
- 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.

Page Fault

- Access to a page marked invalid causes a page fault or trap**. Referred page is not in memory, so need to bring the desired page into memory.**

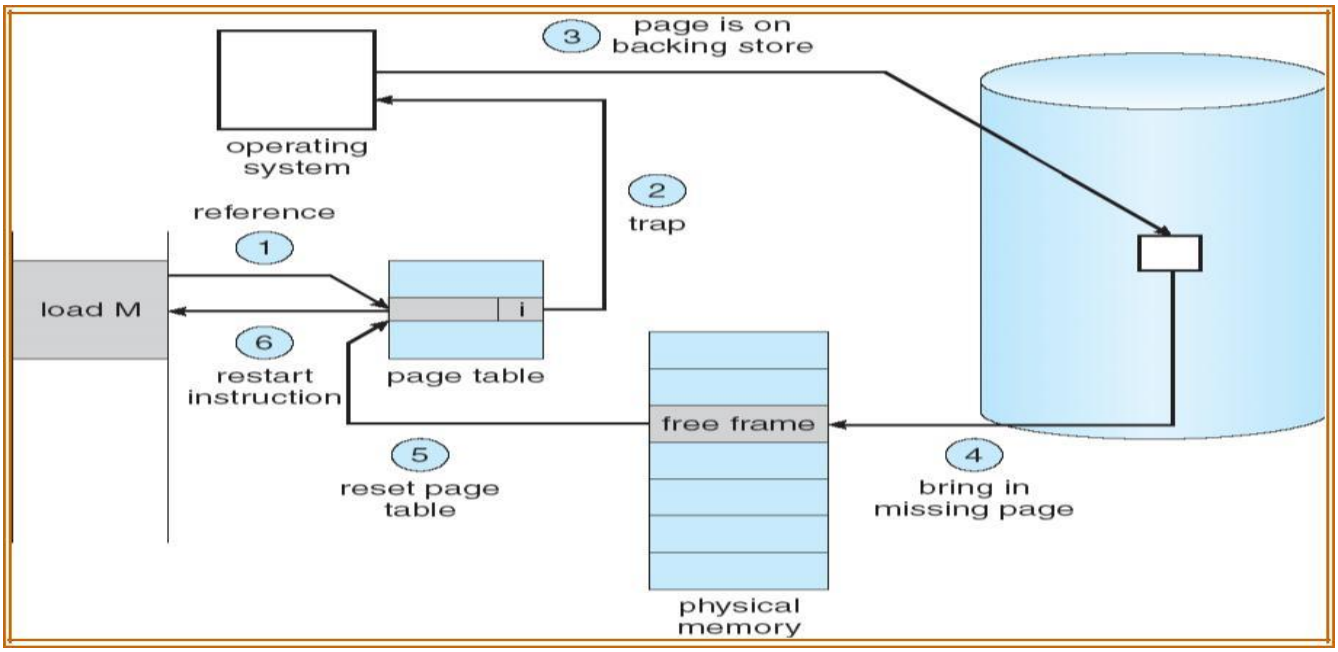


Fig: **Steps in Handling a Page Fault**

1. Determine whether the reference is a valid or invalid memory access by checking an internal table (usually kept with the process control block)

2. a) If the reference is invalid then terminate the process.
   b) If the reference is valid then the page has not been yet brought into main memory.
3. Find a free frame. (by taking one from the free-frame list)
4. Read the desired page into the newly allocated frame.
5. Reset the page table to indicate that the page is now in memory.
6. Restart the instruction that was interrupted.

Pure demand paging

- Never bring a page into memory until it is required.
- We could start a process with no pages in memory**.**
- When the OS sets the instruction pointer to the 1st instruction of the process, which is on the non-memory resident page, then the process immediately faults for the page.
- After this page is bought into the memory, the process continue to execute, faulting as necessary until every page that it needs is in memory.

Performance of demand paging

Demand paging can significantly affect the performance of a computer system. To see why, let's compute the 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, however, a page fault occurs, we must first read the relevant page from disk and then access the desired word from memory.

- Let p be the probability of a page fault $0 \leq p \leq 1$
- Effective Access Time (EAT)

  EAT = (1 – *p*) x ma + *p* x page fault time.

  Where ma → memory access, p → Probability of page fault ($0 \leq p \leq 1$)
- The memory access time denoted ma is in the range 10 to 200 ns.

If there are no page faults then EAT = ma.

- To compute effective access time, we must know how much time is needed to service a page fault.
- A page fault causes the following sequence to occur:
  1. Trap to the OS
  2. Save the user registers and process state.
  3. Determine that the interrupt was a page fault.
  4. Check whether the reference was legal and find the location of page on disk.
  5. Read the page from disk to free frame.
     a. Wait in a queue until read request is serviced.
     b. Wait for seek time and latency time.
     c. Transfer the page from disk to free frame.

6. While waiting, allocate CPU to some other user.
7. Interrupt from disk.
8. Save registers and process state for other users.
9. Determine that the interrupt was from disk.
7. Reset the page table to indicate that the page is now in memory.
8. Wait for CPU to be allocated to this process again.
9. Restart the instruction that was interrupted.

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

1. Service the page-fault interrupt.
2. Read in the page.
3. Restart the process.

Therefore, the effective access time is directly proportional to the page-fault rate. Therefore demand paging can significantly affect the performance of a computer system

Process Creation

- Virtual memory enhances the performance of creating and running processes using:
  - Copy-on-Write
  - Memory-Mapped Files

In general fork() system call creates a child process that is a duplicate of its parent. However, considering that many child processes invoke the exec() system call immediately after creation, the copying of the parent's address space may be unnecessary. Instead, we can use a technique known as copy-on-write.

a) Copy-on-Write

- **Copy-on-Write (CoW**) allows both parent and child processes to initially *share* the same pages in memory. These shared pages are marked as Copy-on-Write pages, meaning that if either process modifies a shared page, a copy of the shared page is created.
  - For example, assume that the child process attempts to modify a page, Which is set as copy-on-write.
  - The operating system will create a copy of this page, mapping it to the address space **of the child process**. The child process will then modify its copied page and not the page belonging to the parent process.
  - If no modification required, they use the same shared pages.
  - Operating systems typically allocate these pages for modification using a technique known as zero-fill-on-demand. Zero-fill-on-demand pages have been zeroed-out before being allocated, thus erasing the previous contents.
  - Copy-on-write is a common technique used by several operating systems, including Windows XP, Linux, and Solaris.

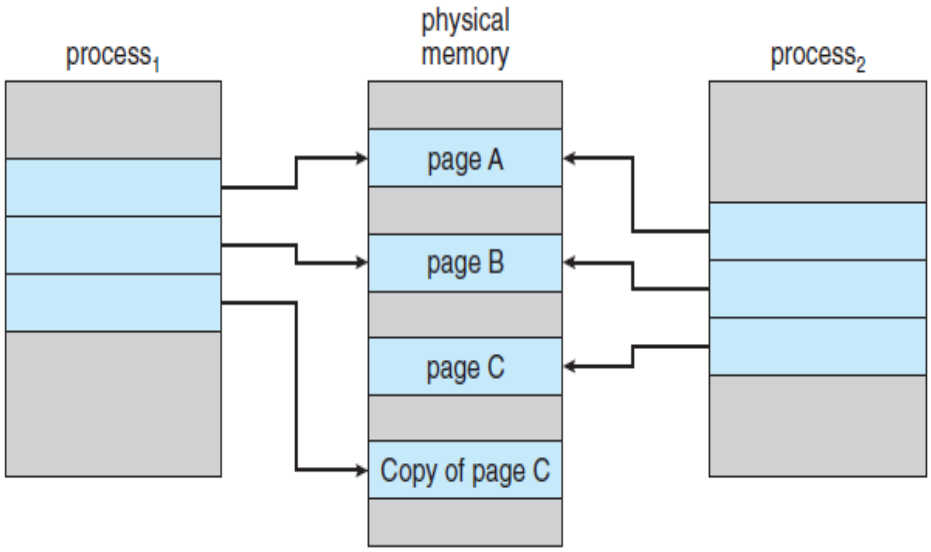


Fig: After child process 1 modifies page C on Copy on Write Technique.

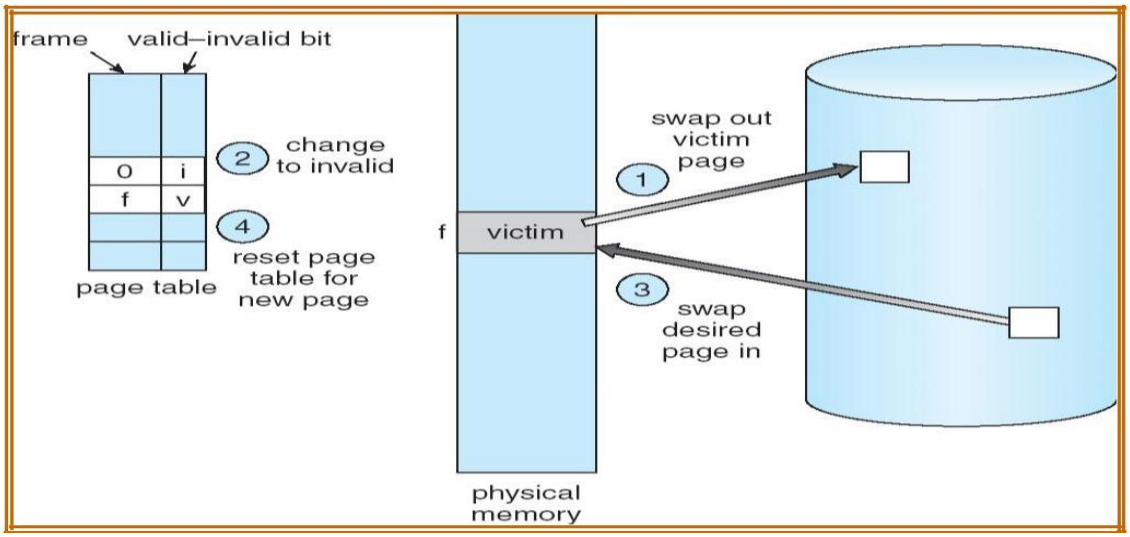
- **vfork():**
  - With this the parent process is suspended & the child process uses the address space of the parent.
  - Because vfork() does not use Copy-on-Write, if the child process changes any pages of the parent‘s address space, the altered pages will be visible to the parent once it resumes.
  - Therefore, vfork() must be used with caution, ensuring that the child process does not modify the address space of the parent.

Page Replacement

- If no frames are free, we could find one that is not currently being used & free it.
- We can free a frame by writing its contents to swap space & changing the page table to indicate that the page is no longer in memory.
- Then we can use that freed frame to hold the page for which the process faulted.

Basic Page Replacement

1. Find the location of the desired page on disk
2. Find a free frame
   - If there is a free frame, then use it.
   - If there is no free frame, use a page replacement algorithm to select a **victim** frame
   - Write the victim page to the disk, change the page & frame tables accordingly.
3. Read the desired page into the (new) free frame. Update the page and frame tables.
4. Restart the process



If no frames are free, two page transfers are required & this situation effectively doubles the page- fault service time.

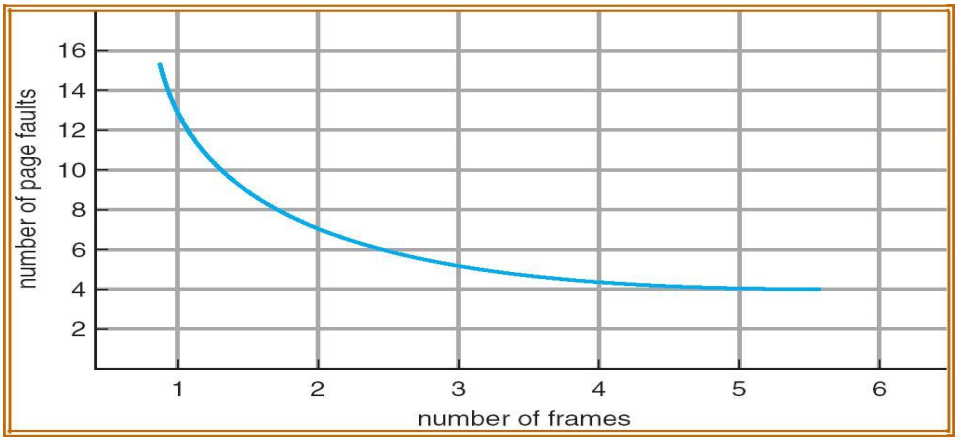
Modify (dirty) bit:

- It indicates that any word or byte in the page is 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 & in this case we must write that page to the disk.
  - If the bit is not set, then if the copy of the page on the disk has not been overwritten, then we can avoid writing the memory page on the disk as it is already there.

Page Replacement Algorithms

1. FIFO Page Replacement
2. Optimal Page Replacement
3. LRU Page Replacement
4. LRU Approximation Page Replacement
5. Counting-Based Page Replacement

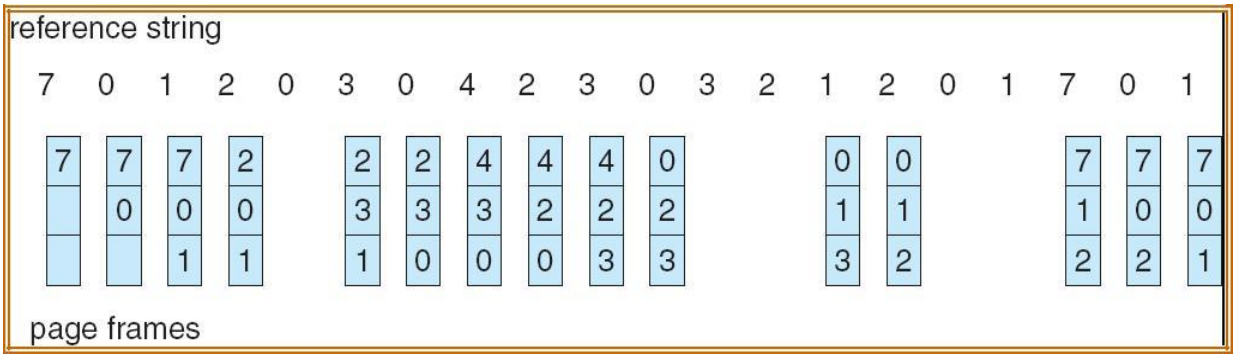
- We evaluate an algorithm by running it on a particular string of memory references & computing the number of page faults. The string of memory reference is called a —reference string.
- The algorithm that provides less number of page faults is termed to be a good one.
- As the number of available frames increases, the number of page faults decreases. This is shown in the following graph:



(a) FIFO page replacement algorithm

- **Replace the oldest page.**
- This algorithm associates with each page, the time when that page was brought in.

Example:
Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1
No.of available frames = 3 (3 pages can be in memory at a time per process)



No. of page faults = 15

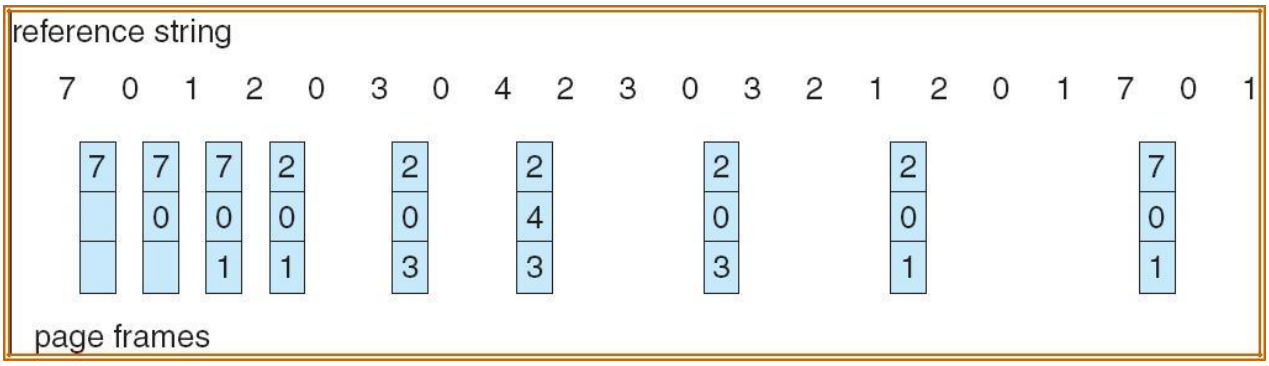
Drawback:

- FIFO page replacement algorithm _s performance is not always good.
- To illustrate this, consider the following example:
  Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- If No.of available frames -= 3 then the no.of page faults =9
- If No.of available frames =4 then the no.of page faults =10
- Here the no. of page faults increases when the no.of frames increases .This is called

as **Belady's Anomaly.**

(b) Optimal page replacement algorithm

- **Replace the page that will not be used for the longest period of time.**
  Example:
  Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1
  No.of available frames = 3



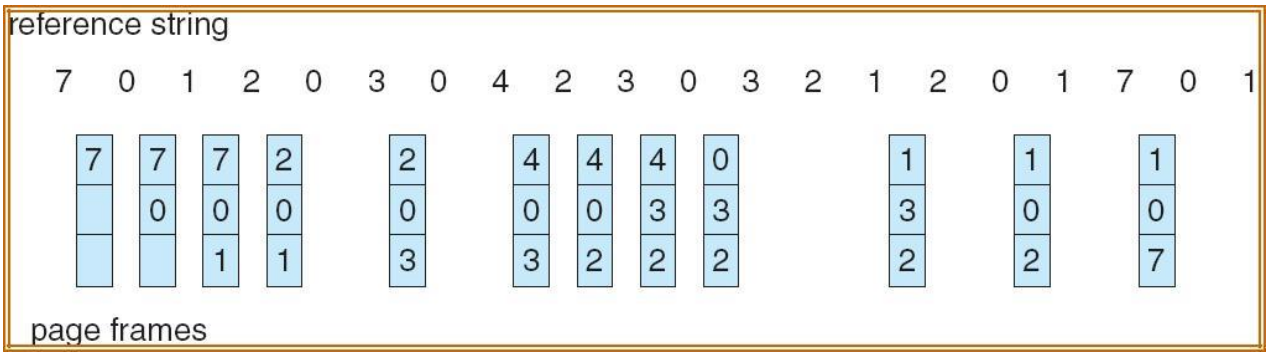
No. of page faults = 9

Drawback:

- It is difficult to implement as it requires future knowledge of the reference string.

(c) LRU(Least Recently Used) page replacement algorithm

- **Replace the page that has not been used for the longest period of time.**
  Example:
  Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1
  No.of available frames = 3



No. of page faults = 12

- LRU page replacement can be implemented using

1. **Counters**
   - ✓ Every page table entry has a time-of-use field and a clock or counter is

associated with the CPU.

- ✓ The counter or clock is incremented for every memory reference.
- ✓ Each time a page is referenced , copy the counter into the time-of-use field.
- ✓ When a page needs to be replaced, replace the page with the smallest counter value.

2. **Stack**
   - ✓ Keep a stack of page numbers
   - ✓ Whenever a page is referenced, remove the page from the stack and put it on top of the stack.
   - ✓ When a page needs to be replaced, replace the page that is at the bottom of the stack.(LRU page)

(d) LRU Approximation Page Replacement

- o Reference bit
  - ✓ With each page associate a reference bit, initially set to 0
  - ✓ When page is referenced, the bit is set to 1
- o When a page needs to be replaced, replace the page whose reference bit is 0
- o The order of use is not known , but we know which pages were used and which were not used.

(i) Additional Reference Bits Algorithm

- o Keep an 8-bit byte for each page in a table in memory.
- o At regular intervals , a timer interrupt transfers control to OS.
- o The OS shifts reference bit for each page into higher- order bit shifting the other bits right 1 bit and discarding the lower-order bit.

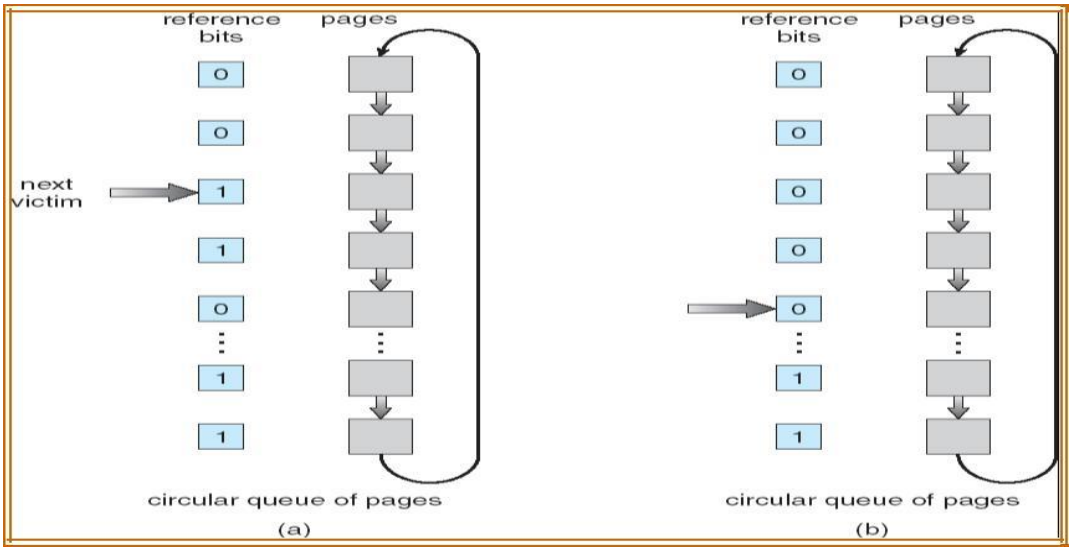
Example:

- o If reference bit is 00000000 then the page has not been used for 8 time periods.
- o If reference bit is 11111111 then the page has been used atleast once each time period.
- o If the reference bit of page 1 is 11000100 and page 2 is 01110111 then page 2 is the LRU page.

(ii) Second Chance Algorithm

- o Basic algorithm is FIFO
- o When a page has been selected , check its reference bit.
  - ✓ If 0 proceed to replace the page
  - ✓ If 1 give the page a second chance and move on to the next FIFO page.
  - ✓ When a page gets a second chance, its reference bit is cleared and arrival time is reset to current time.

- ✓ Hence a second chance page will not be replaced until all other pages are replaced.



(iii) Enhanced Second Chance Algorithm

- Consider both reference bit and modify bit
- There are four possible classes
  1. (0,0) – neither recently used nor modified → Best page to replace
  2. (0,1) – not recently used but modified → page has to be written out before replacement.
  3. (1,0) - recently used but not modified → page may be used again
  4. (1,1) – recently used and modified → page may be used again and page has to be written to disk

(e) Counting-Based Page Replacement

- Keep a counter of the number of references that have been made to each page
  1. **Least Frequently Used (LFU)Algorithm**: replaces page with smallest count
  2. **Most Frequently Used (MFU)Algorithm**: replaces page with largest count
     - ✓ It is based on the argument that the page with the smallest count was probably just brought in and has yet to be used

Page Buffering Algorithm

- These are used along with page replacement algorithms to improve their performance

Technique 1:

- A pool of free frames is kept.
- When a page fault occurs, choose a victim frame as before.

- Read the desired page into a free frame from the pool
- The victim frame is written onto the disk and then returned to the pool of free frames.

Technique 2:

- Maintain a list of modified pages.
- Whenever the paging device is idles, a modified is selected and written to disk and its modify bit is reset.

Technique 3:

- A pool of free frames is kept.
- Remember which page was in each frame.
- If frame contents are not modified then the old page can be reused directly from the free frame pool when needed

6.3 Allocation of Frames

- There are two major allocation schemes
  - ✓ Equal Allocation
  - ✓ Proportional Allocation ○

Equal allocation

- ✓ If there are n processes and m frames then allocate m/n frames to each process.
- ✓ **Example:** If there are 5 processes and 100 frames, give each process 20 frames.

- **Proportional allocation**
  - ✓ Allocate according to the size of process

Let s_i be the size of process i.
Let m be the total no. of frames
Then $S = \sum s_i$
$a_i = s_i / S * m$
where a_i is the no.of frames allocated to process i.

Global vs. Local Replacement

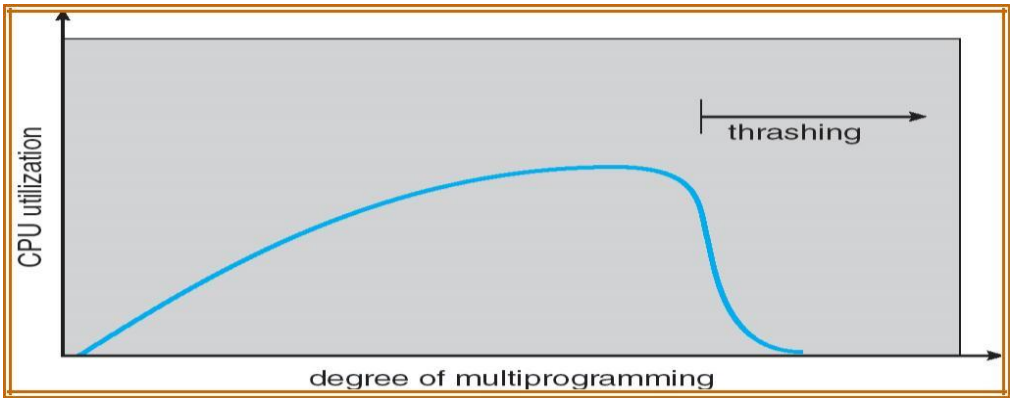
- **Global replacement** – each process selects a replacement frame from the set of all frames; one process can take a frame from another.
- **Local replacement** – each process selects from only its own set of allocated frames.

6.4 Thrashing

- High paging activity is called **thrashing**.
- If a process does not have ―enough‖ pages, the page-fault rate is very high. This leads to:
  - ✓ low CPU utilization

- ✓ operating system thinks that it needs to increase the degree of multiprogramming
- ✓ another process is added to the system

- When the CPU utilization is low, the OS increases the degree of multiprogramming.
- If global replacement is used then as processes enter the main memory they tend to steal frames belonging to other processes.
- Eventually all processes will not have enough frames and hence the page fault rate becomes very high.
- Thus swapping in and swapping out of pages only takes place.
- This is the cause of thrashing.

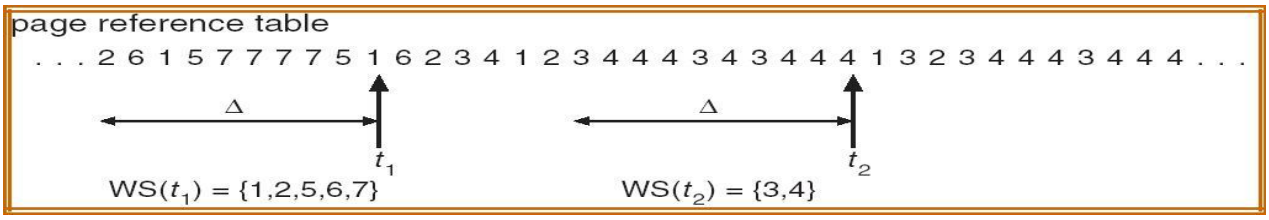


- To **limit thrashing**, we can use a **local replacement** algorithm.
- To prevent thrashing, there are two methods namely ,
  - ✓ Working Set Strategy
  - ✓ Page Fault Frequency

1. Working-Set Strategy

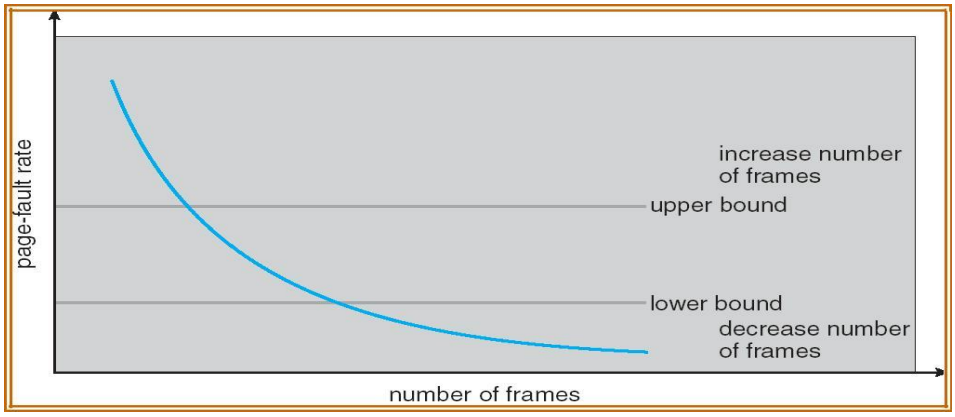
- It is based on the assumption of the model of locality.
- Locality is defined as the set of pages actively used together.
- Working set is the set of pages in the most recent Δ page references
- Δ is the working set window.

  - ✓ if Δ too small , it will not encompass entire locality
  - ✓ if Δ too large ,it will encompass several localities
  - ✓ if $\Delta = \infty \Rightarrow$ it will encompass entire program
- $D = \Sigma\ WSS_i$
  - ✓ Where WSS_i is the working set size for process i.
  - ✓ D is the total demand of frames
- if $D > m$ then Thrashing will occur.



2. Page-Fault Frequency Scheme

- If actual rate too low, process loses frame
- If actual rate too high, process gains frame



Other Issues

- **Prepaging**
  - To reduce the large number of page faults that occurs at process startup
  - Prepage all or some of the pages a process will need, before they are referenced
  - But if prepaged pages are unused, I/O and memory are wasted
- **Page Size**

  Page size selection must take into consideration:
  - fragmentation
  - table size
  - I/O overhead
  - locality
- **TLB Reach**
  - TLB Reach - The amount of memory accessible from the TLB
  - TLB Reach = (TLB Size) X (Page Size)
  - Ideally, the working set of each process is stored in the TLB. Otherwise there is a high degree of page faults.
  - Increase the Page Size. This may lead to an increase in fragmentation as not all applications require a large page size
  - Provide Multiple Page Sizes. This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation.

- **I/O interlock**
  - Pages must sometimes be locked into memory
  - Consider I/O. Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm.