Virtual Memory

Outline

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing

Background

Separation of user logical memory from physical memory, which allows the execution of processes that may not be completely in memory.

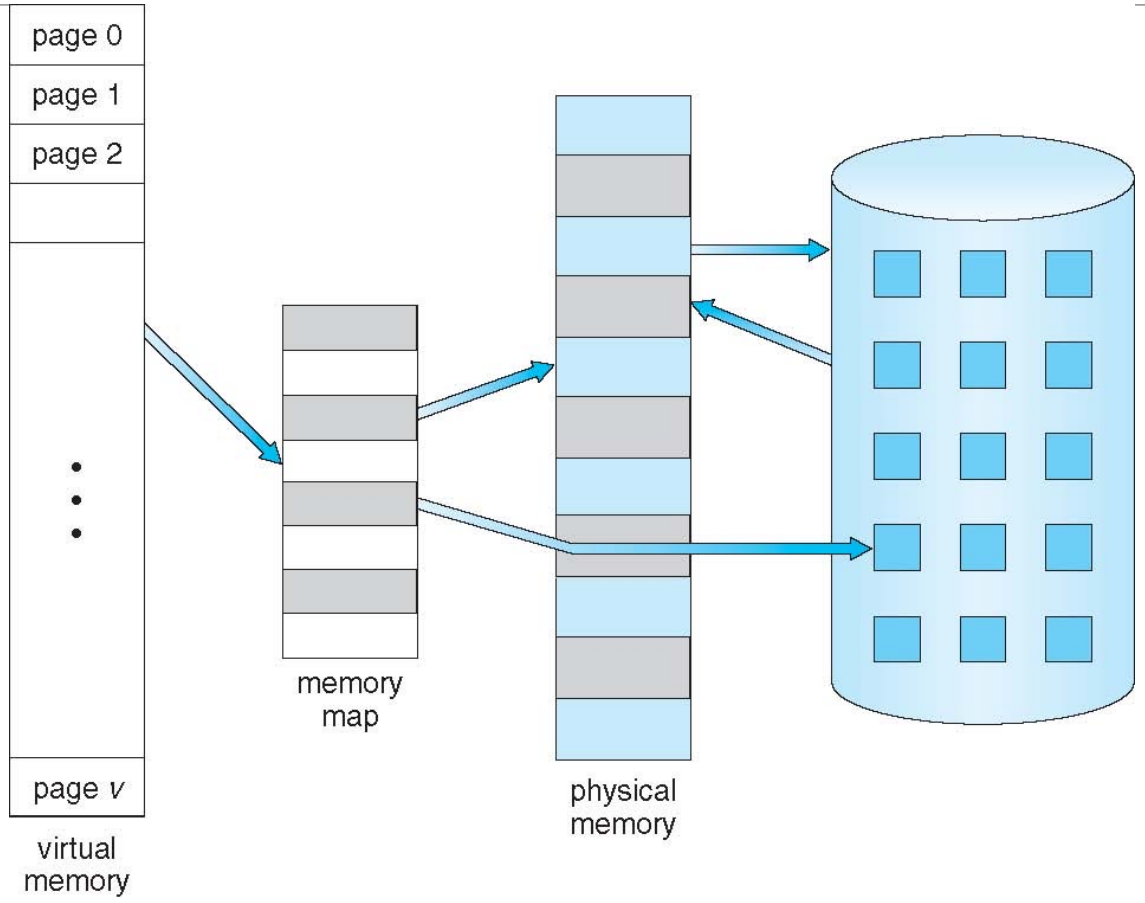
- Only part of the program needs to be in memory for execution.

Advantages

- Logical address space can be much larger than physical address space.i.e Users can write the programs for an extremely large virtual -address space, as it would no longer constrained by the amount of physical memory that is available.
- As each process uses less physical memory, more programs could be run simultaneously. -> increases the throughput .
- Less I/O would be needed to load or swap each user program into memory so each user program would run faster.
- Allows for more efficient process creation.

Virtual memory can be implemented via Demand Paging.

Virtual Memory That is Larger Than Physical Memory



Demand Paging

Could bring entire process into memory at load time

Or bring a page into memory only when it is needed

- Less I/O needed, no unnecessary I/O
- Less memory needed
- Faster response
- More users

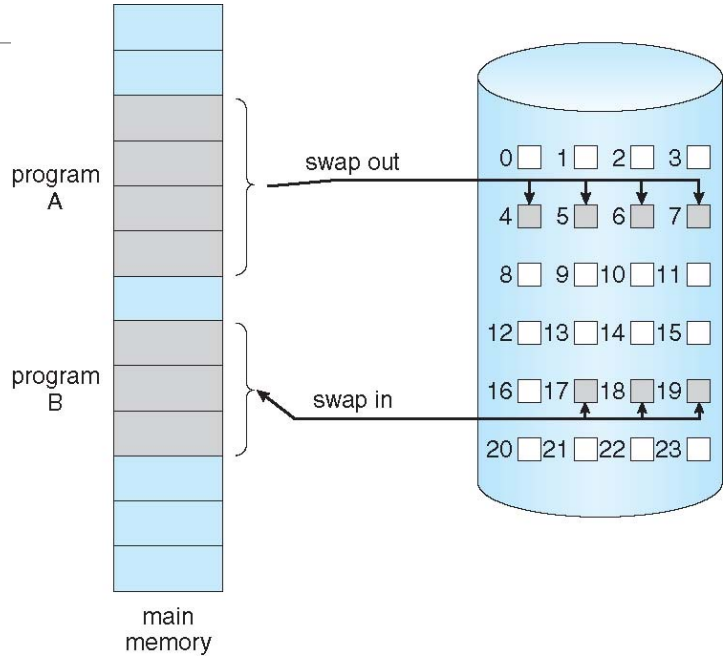
Similar to paging system with swapping

Page is needed ⇒ reference to it

- invalid reference ⇒ abort
- not-in-memory ⇒ bring to memory

Lazy swapper – never swaps a page into memory unless page will be needed

- Swapper that deals with pages is a **pager**



Basic Concepts

With swapping, pager guesses which pages will be used before swapping out again

Instead, pager brings in only those pages into memory

How to determine that set of pages?

- Need new MMU functionality to implement demand paging

If pages needed are already **memory resident**

- No difference from non demand-paging

If page needed and not memory resident

- Need to detect and load the page into memory from storage
  - Without changing program behavior
  - Without programmer needing to change code

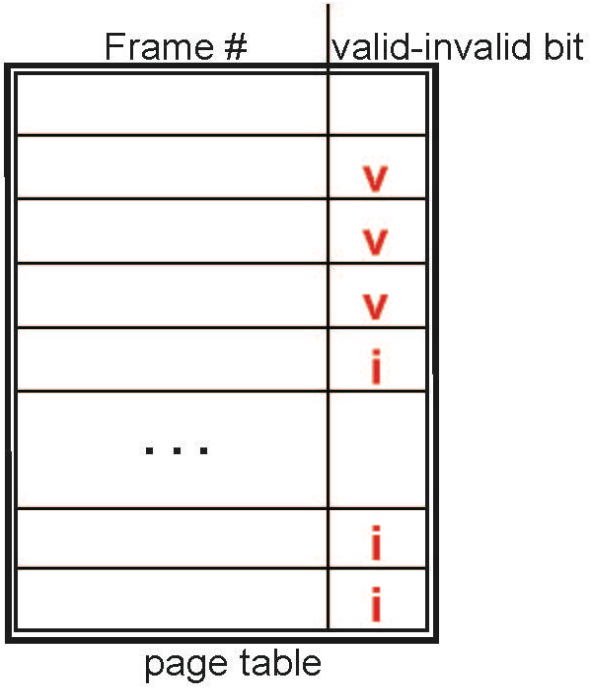
Valid-Invalid Bit

With each page table entry a valid–invalid bit is associated
(**v** ⇒ in-memory – **memory resident**, **i** ⇒ not-in-memory)

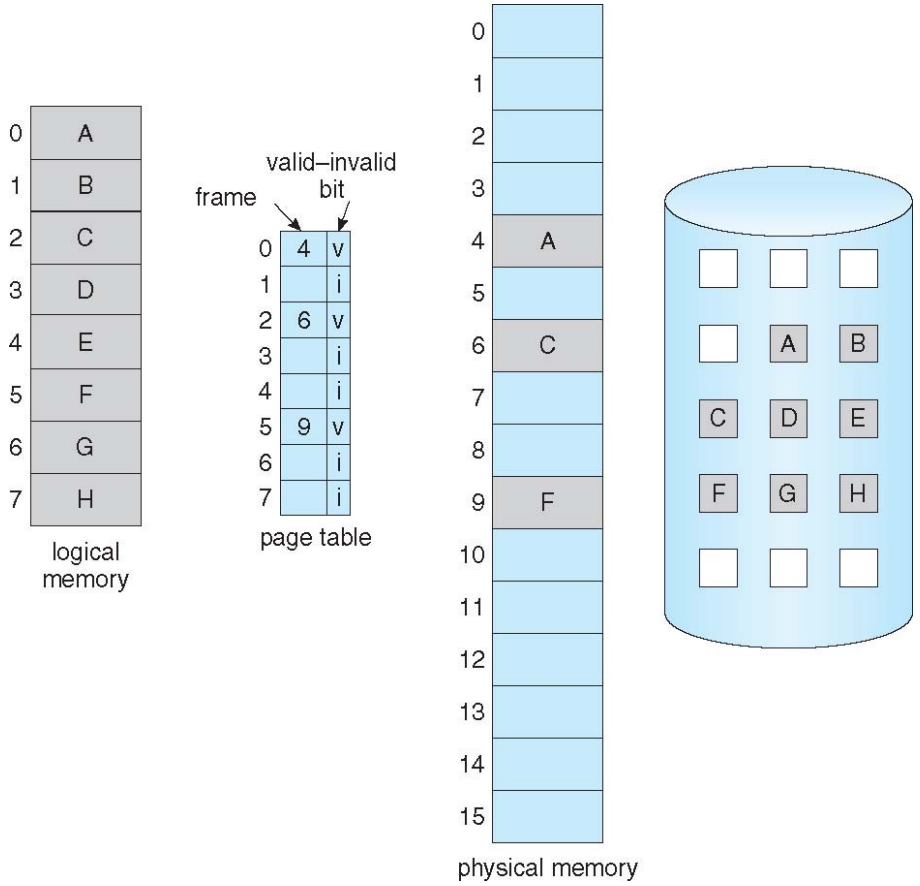
Initially valid–invalid bit is set to **i** on all entries

Example of a page table snapshot:

During MMU address translation, if valid–invalid bit in page table entry is **i** ⇒ page fault



Page Table When Some Pages Are Not in Main Memory



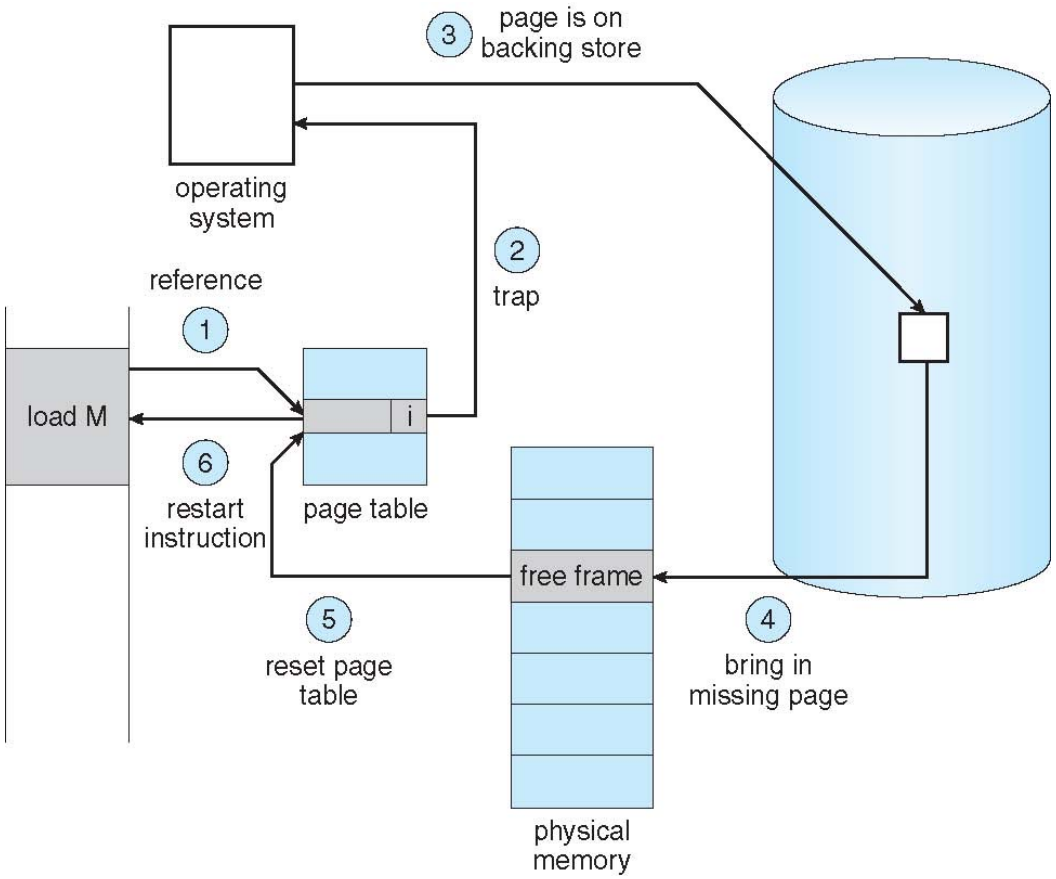
Page Fault

If there is a reference to a page, first reference to that page will trap to operating system:

page fault

1. Operating system looks at another table to decide:
   - Invalid reference ⇒ abort
   - Just not in memory
2. Find free frame
3. Swap page into frame via scheduled disk operation
4. Reset tables to indicate page now in memory
   Set validation bit = **v**
5. Restart the instruction that caused the page fault

Steps in Handling a Page Fault



Aspects of Demand Paging

Extreme case – start process with *no* pages in memory

- OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault
- And for every other process pages on first access
- **Pure demand paging**

Hardware support needed for demand paging

- Page table with valid / invalid bit
- Secondary memory (swap device with **swap space**)
- Instruction restart

Stages in Demand Paging (worse case)

1. Trap to the operating system
2. Save the user registers and process state
3. Determine that the interrupt was a page fault
4. Check that the page reference was legal and determine the location of the page on the disk
5. Issue a read from the disk to a free frame:
    1. Wait in a queue for this device until the read request is serviced
    2. Wait for the device seek and/or latency time
    3. Begin the transfer of the page to a free frame
6. While waiting, allocate the CPU to some other user
7. Receive an interrupt from the disk I/O subsystem (I/O completed)
8. Save the registers and process state for the other user
9. Determine that the interrupt was from the disk
10. Correct the page table and other tables to show page is now in memory
11. Wait for the CPU to be allocated to this process again
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction

Performance of Demand Paging

Three major activities

- Service the interrupt – careful coding means just several hundred instructions needed
- Read the page – lots of time
- Restart the process – again just a small amount of time

Page Fault Rate $0 \le p \le 1$

- if $p = 0$ no page faults
- if $p = 1$, every reference is a fault

Effective Access Time (EAT)

$$\begin{aligned} \text{EAT} = (1 - p) \text{ x memory access} \\ + p \text{ (page fault overhead} \\ + \text{swap page out} \\ + \text{swap page in)} \end{aligned}$$

Example

Memory access time = 200 nanoseconds

Average page-fault service time = 8 milliseconds

EAT = (1 – p) x 200 + p (8 milliseconds)

= (1 – p x 200 + p x 8,000,000

= 200 + p x 7,999,800

If one access out of 1,000 causes a page fault, then

EAT = 8.2 microseconds.

This is a slowdown by a factor of 40!!

If want performance degradation < 10 percent

- 220 > 200 + 7,999,800 x p
  20 > 7,999,800 x p
- p < .0000025
- < one page fault in every 400,000 memory accesses

Copy-on-Write

Copy-on-Write (COW) allows both parent and child processes to initially ***share*** the same pages in memory

- If either process modifies a shared page, only then is the page copied

COW allows more efficient process creation as only modified pages are copied

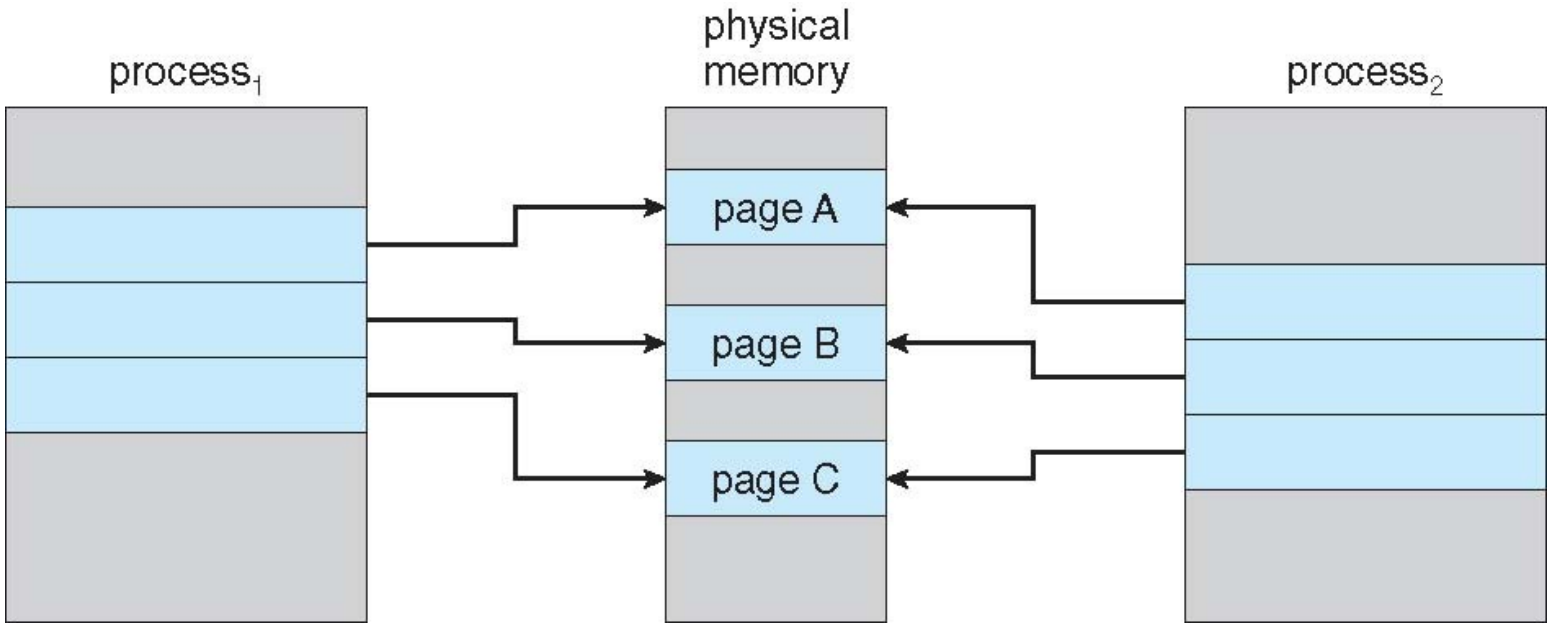
In general, free pages are allocated from a **pool** of **zero-fill-on-demand** pages

- Pool should always have free frames for fast demand page execution

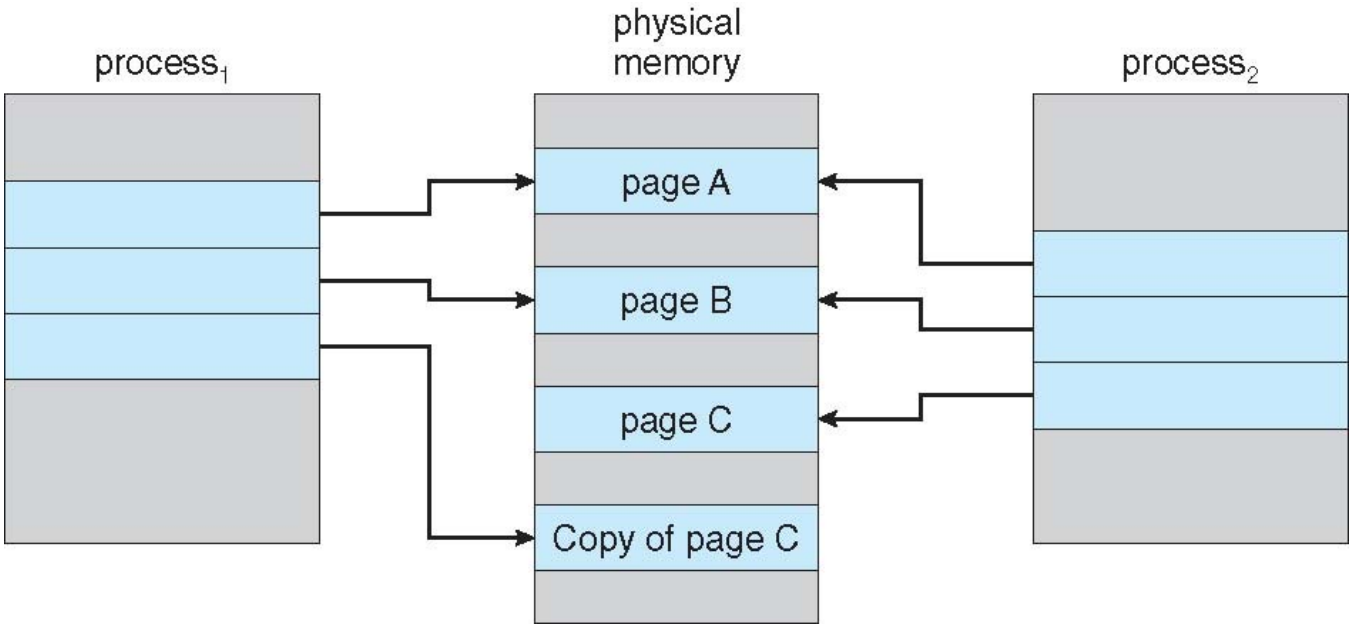
`vfork()` variation on `fork()` system call has parent suspend and child using copy-on-write address space of parent

- Designed to have child call `exec()`
- Very efficient

Before Process 1 Modifies Page C



After Process 1 Modifies Page C



Page Replacement

Two problems must be addressed to implement demand paging

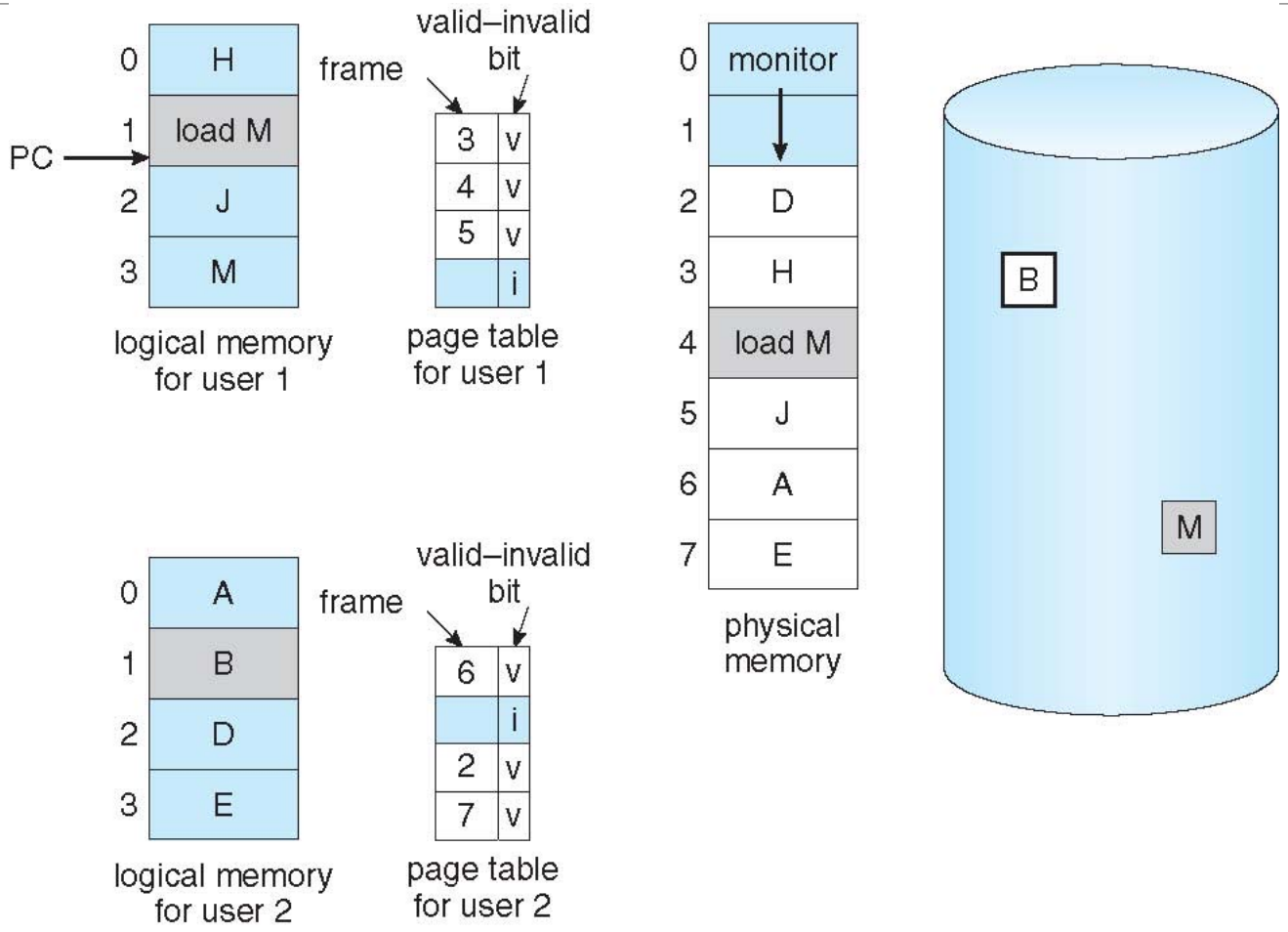
- Frame allocation algorithm
- Page Replacement algorithm

What happens if there is no free frame?

- Terminate a process.
- Swap out a process and release all its frames.
- Page replacement: Find some page in memory that is not really in use and swap it.

Over-allocation of memory (/ dynamic requests) can be handled by modifying the page fault routine to include page replacement.

Need For Page Replacement



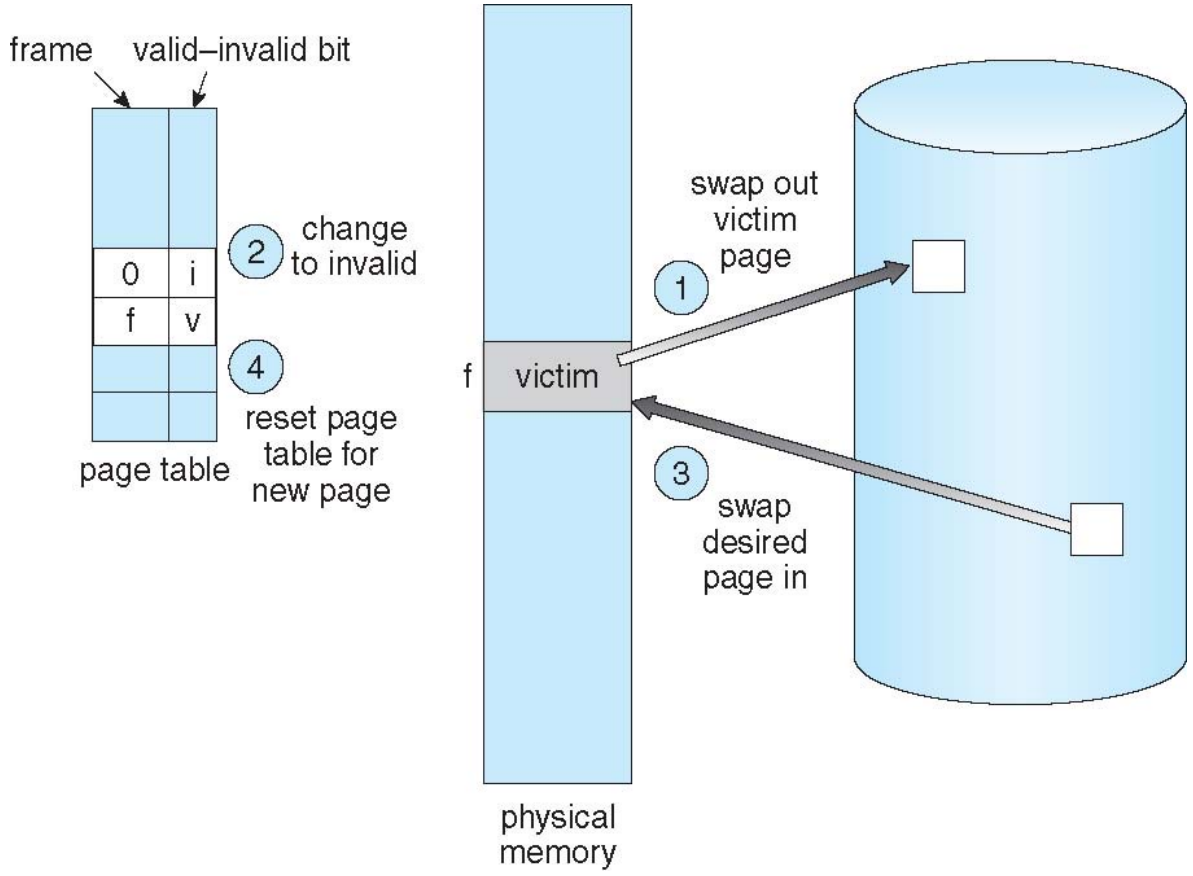
Page Replacement…

Modified version of page-fault service to include page replacement.

1. Find the location of the desired page on disk
2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a **victim frame**
   - Write victim frame to disk if dirty
3. Bring the desired page into the (newly) free frame; update the page and frame tables
4. Continue the process by restarting the instruction that caused the trap

Note now potentially 2 page transfers for page fault – increasing EAT

Page Replacement…



Page Replacement Algorithm

Choose PRA that has lowest page fault rate.

Evaluate the PRA by running it on a particular string of memory references (reference string) and computing the number of page faults on that string.

Reference string can be generated artificially (ex: using random number generator) or trace the given system and record the address of each memory reference.

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

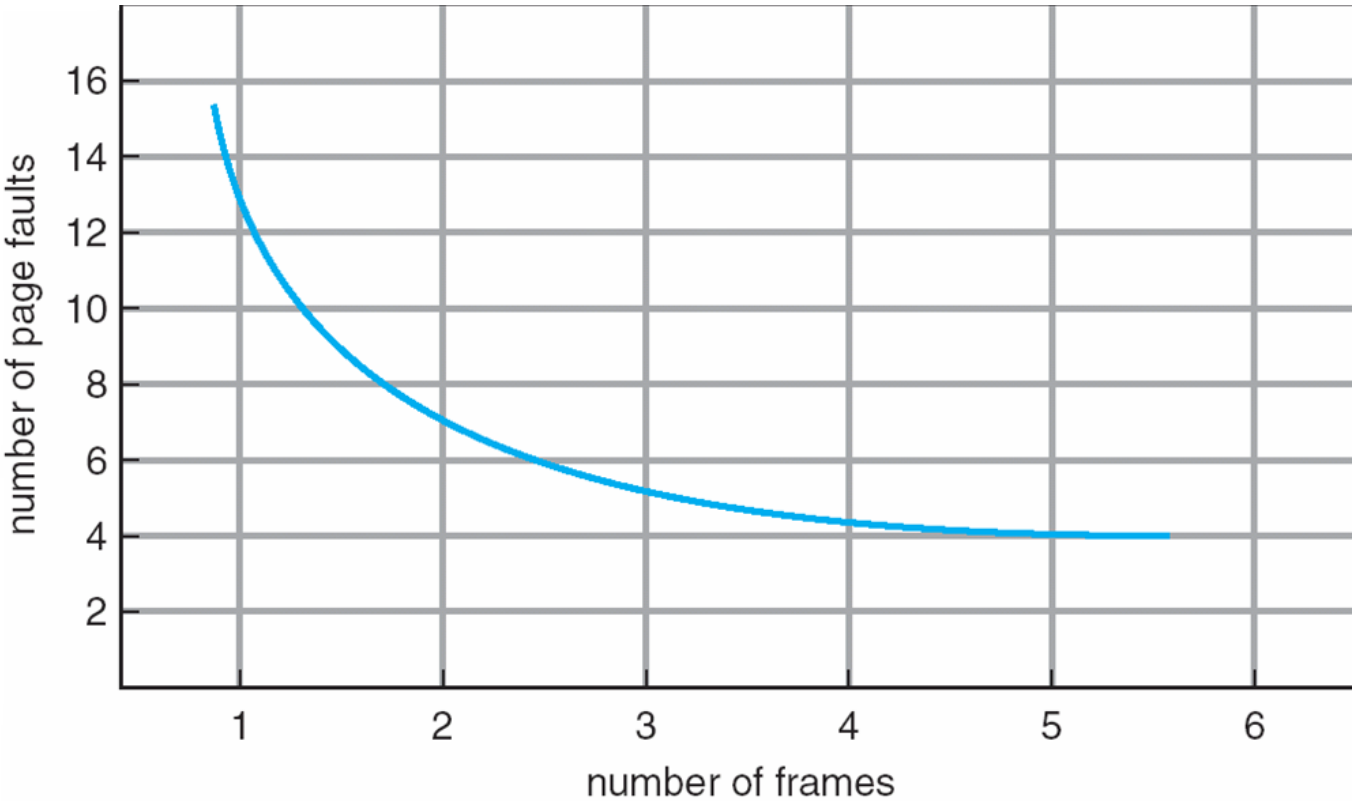
0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102

which, at 100 bytes per page, is reduced to the following reference string:

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

Number of page faults is also dependent on the number of frames(-> physical memory size)

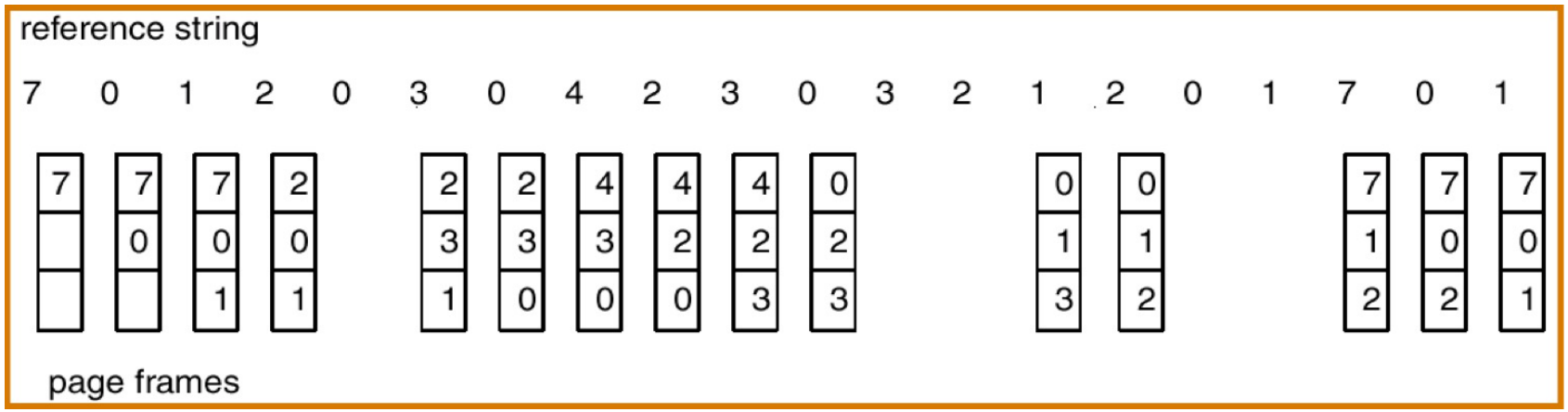
Graph of Page Faults Versus The Number of Frames



FIFO Page Replacement Alg.

Associates time when the page is brought in. When the page must be replaced, the oldest page is chosen.

No need to strictly associate the time rather implemented with FIFO queue (replace the one at the head of the queue & replaced page is put at the tail of queue).



Optimal Page Replacement

- Replace the page that will not be used for longest period of time.
- OPT guarantees the lowest possible page fault rate.
- Requires future knowledge of the reference string.
- Used mainly for comparison studies.
- Example

7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1 With 3 frames

1,2,3,4,1,2,5,1,2,3,4,5 With 4 frames

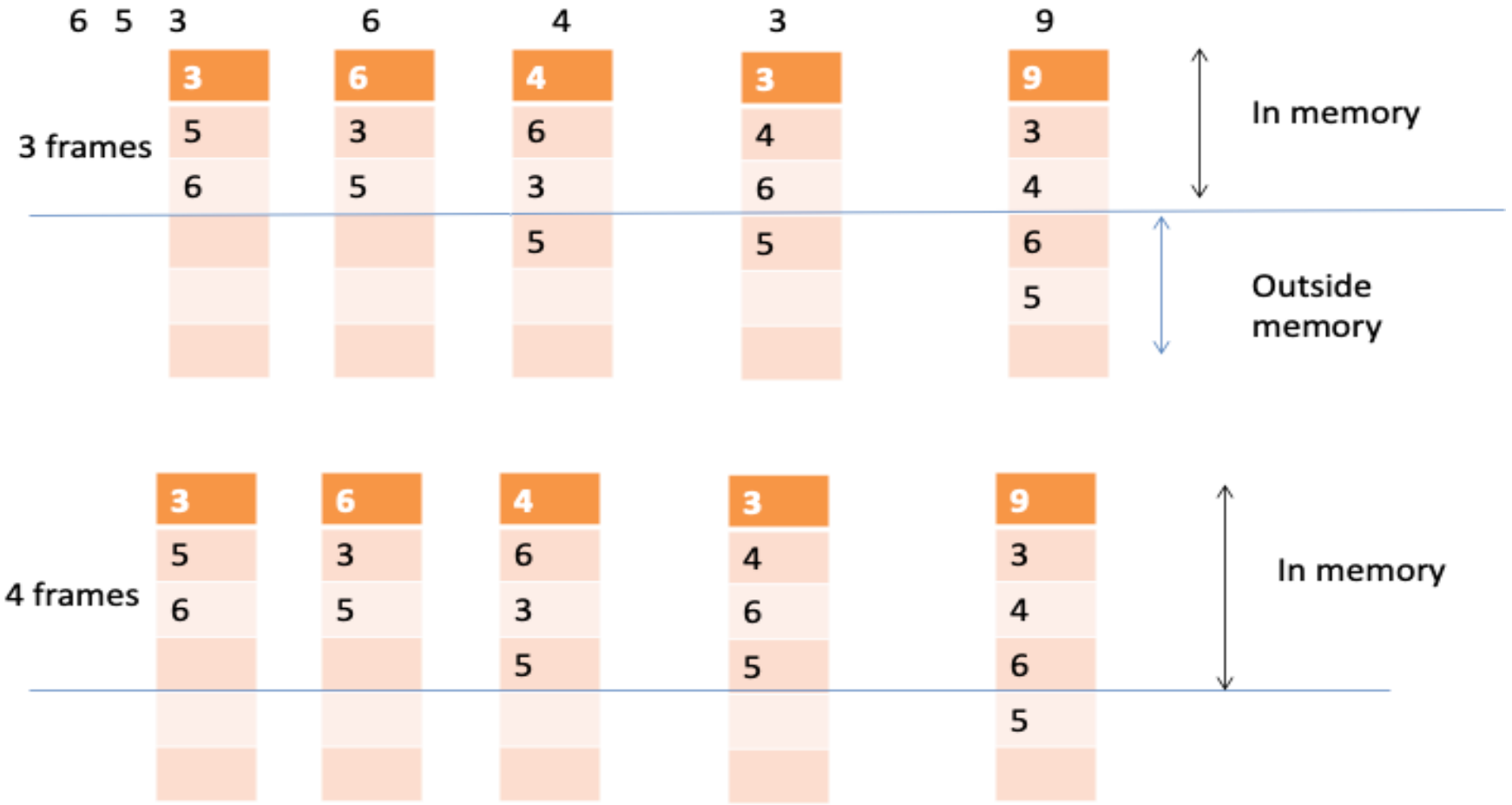
LRU Page Replacement

- Least Recently Used (LRU) associates with each page the time of that page's last use.
- When the 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.
- If S is the reference string and S^R is the reverse of S then the page fault rate for LRU algorithm on S is same as the page-fault rate for LRU algorithm on S^R.
- Example

Stack Algorithms

- Certain page replacement algorithms are more "well behaved" than others
- (In the FIFO example), the problem arises because the set of pages loaded with a memory allocation of 3 frames is not necessarily also loaded with a memory allocation of 4 frames
- There are a set of paging algorithms whereby the set of pages loaded with an allocation of m frames is always a subset of the set of pages loaded with an allocation of m +1 frames. This property is called the inclusion property
- Algorithms that satisfy the inclusion property are not subject to Belady's anomaly. FIFO does not satisfy the inclusion property and is not a stack algorithm

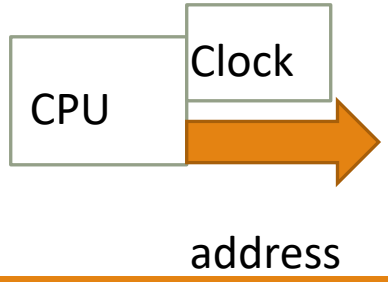
LRU algorithm



LRU Algorithm-Implementation

Counter implementation

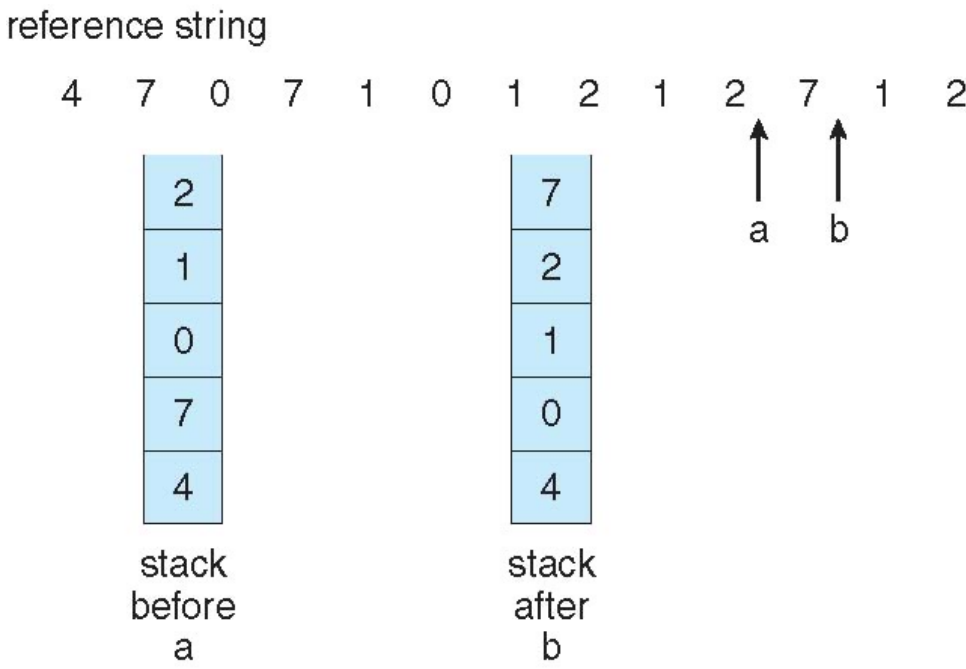
- CPU maintains a clock
- Every page entry has a Time of use;
  - every time page is referenced, copy the clock into the time of use
- When a page needs to be replaced, look at the "Time of use" to find smallest value
  - Search through table needed



			Time of Use
0			
1			
2			
3			

LRU Algorithm-Implementation …

- **Stack implementation**
  - Keep a stack of page numbers in a double linked list form:
  - Page referenced:
    - move it to the top
  - Victim page is the bottom page



Allocation of Frames…

Each process needs *minimum* number of frames

Minimum number is defined by the instruction set

Page fault forces to restart the instruction
- Enough frames to hold all the pages for that instruction

Example:
- Single address instruction (2 frames)
- Two address instruction (3 frames)

Maximum of course is total frames in the system

Two major allocation schemes
- fixed allocation
- proportional allocation

Fixed and proportional Allocation

Equal/Fixed allocation – m frames and n processes

- Each process gets m/n

For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames

- Keep some as free frame buffer pool

Unfair for small and large sized processes

Proportional allocation – Allocate according to the size of process

- Dynamic as degree of multiprogramming, process sizes change

s_i = size of process p_i

$S = \sum s_i$

m = total number of frames

a_i = allocation for $p_i = \frac{s_i}{S} \times m$

$m = 64$

$s_1 = 10$

$s_2 = 127$

$a_1 = \frac{10}{137} \times 64 \approx 5$

$a_2 = \frac{127}{137} \times 64 \approx 59$

Priority Allocation

Allocation of frames

Depends on multiprogramming level

Use a proportional allocation scheme using priorities along with size

Global vs. Local Allocation

Frames are allocated to various processes

If process P_i generates a page fault

- select for replacement one of its frames
- select for replacement a frame from another process

Local replacement – each process selects from only its own set of allocated frames

- More consistent per-process performance
- But possibly underutilized memory

Global replacement – process selects a replacement frame from the set of all frames; one process can take a frame from another

- But then process execution time can vary greatly
- But greater throughput ----- so more common

Processes can not control its own page fault rate

- Depends on the paging behavior of other processes

Thrashing

If a process does not have "enough" active 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 added to the system, causing more page faults.

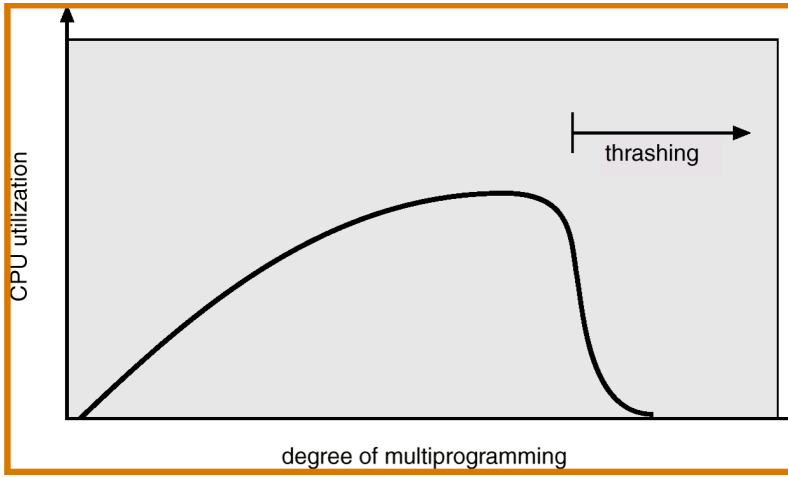
Thrashing ≡ busy swapping pages in and out

≡ doing more paging than executing

A process is thrashing if it is spending more time paging than executing.

Cause of Trashing…

- As the degree of multiprogramming increases , CPU utilization also increases until a maximum is reached. If the degree of multiprogramming is increased even further, trashing sets in and CPU utilization drops sharply.
- The effect of thrashing can be limited by using local replacement algorithm. But this only partially solves the problem.



Demand Paging and Thrashing

Why does demand paging work?

Locality model

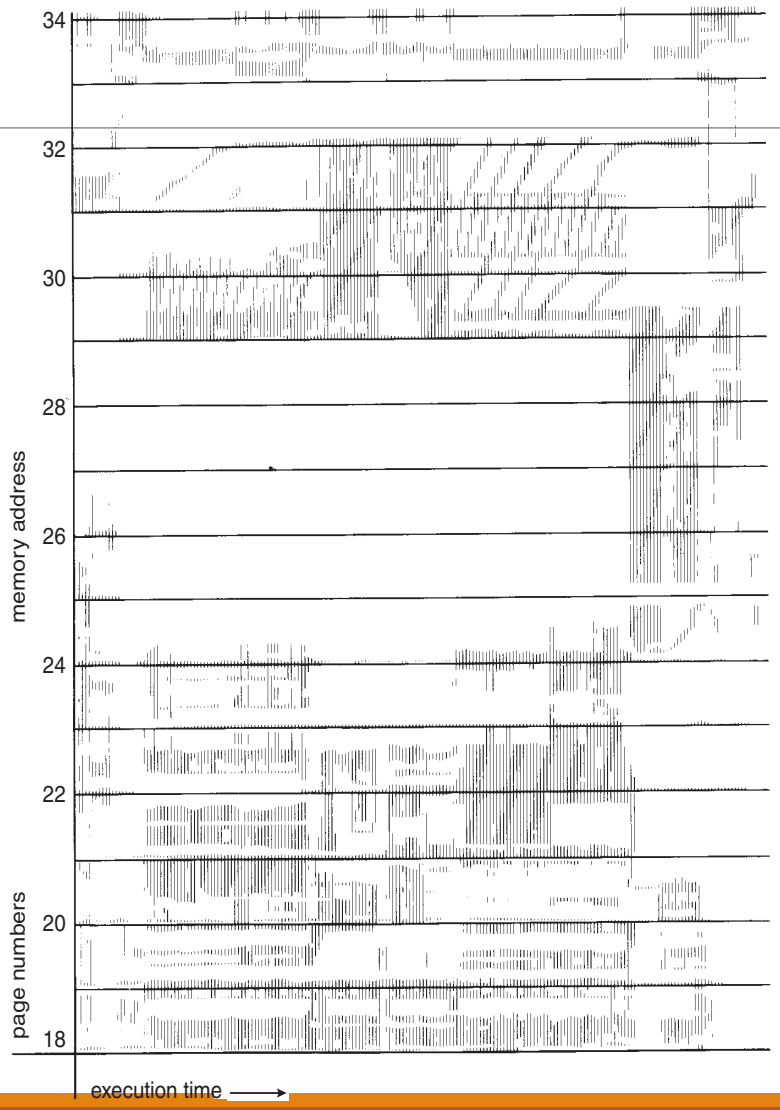
- Process migrates from one locality to another
- Localities may overlap

Why does thrashing occur?

Σ size of locality $>$ total memory size

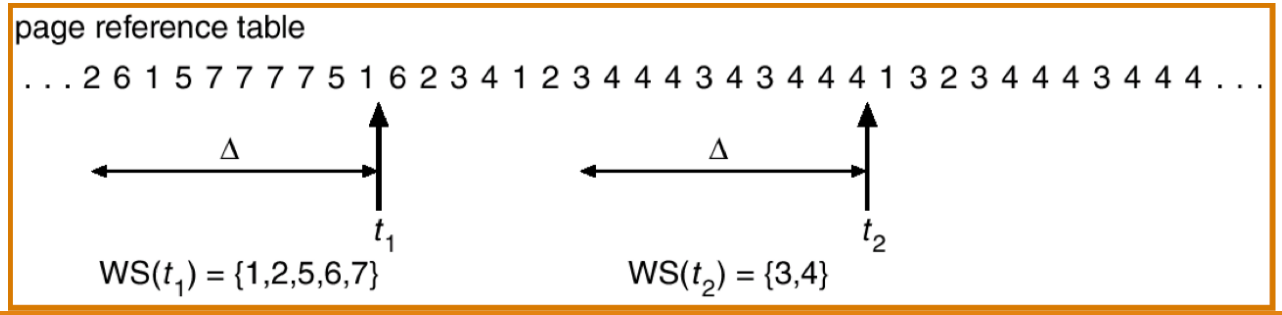
- Limit effects by using local or priority page replacement

Locality In A Memory-Reference Pattern



Working-Set Model

- It is based on the assumption of locality model.
- $\Delta \equiv$ working-set window $\equiv$ a fixed number of the most recent page references
- The set of pages in the most recent Δ page references is the **working set**.
- The working set is an approximation of the program's current locality.
- If a page is in active use, it will be in the working set. If it is no longer being used, it will drop from the working set Δ time units after its last reference.



Δ = 10

Working Set Model…

The accuracy of working set depends on the selection of Δ

- if Δ too small, will not encompass entire locality.
- if Δ too large, will encompass several localities.
- if $\Delta = \infty$, will encompass entire program.

WSS_i (working-set size of Process P_i) = total number of pages referenced in the most recent Δ (varies in time)

$D = \Sigma\ WSS_i \equiv$ total demand for frames

If the total demand is greater than the total number of available frames ($D > m$) $\Rightarrow$ Thrashing

The OS monitors the working set of each process and allocates enough frames to provide it with its working-set size. If there are enough extra frames, another process can be initiated.

Policy: if $D > m$, then suspend one of the processes.

Page-Fault Frequency

Thrashing has high page-fault rate.

Establish the upper and lower bounds on the desired page-fault rate.

- If the actual page fault rate exceeds the upper limit, we allocate that process another frame.
- If the page fault rate falls below the lower limit, we remove a frame from that process.

If the page fault rate increases and no free frames are available, we must select some process and suspend it and distribute the freed frames to the processes with high page-fault rates.

Page-Fault Frequency