



# Operating Systems ICS 431

## Weeks 11-12

# Virtual Memory

**Dr. Tarek Helmy El-Basuny**

# Virtual Memory (VM)

- **Introducing of the Virtual Memory**
  - What do we want to achieve by using virtual memory?
  - What are the problems do we address through virtual memory?
  - How does virtual memory work?
  - The meaning of Demand Paging, Thrashing concept,
  - Page Fault and the Cost of Handling a Page Fault,
  - What are the Advantages of Virtual Memory? (less I/O time, copy on write, memory mapped files)
- **Page Replacement Polices**
  - Local and Global replacement strategies.
  - First In First Out (FIFO), (Examples with different frame numbers allocated),
  - Optimal Page Replacement (OPR),
  - Least Recent Used (LRU),
  - Most Recent Used (MRU),
  - Second Choice Page-Replacement,
  - Enhanced version of Second Choice Page-Replacement,
  - Page-Buffering Algorithms.
  - Relationship between the allocated frames and the page fault frequency.
- **Allocation of Frames to processes to minimize Thrashing**
  - Based on the size of the process or
  - Based on the working set of the process or
  - Based on the Fault frequency rate
    - Other Considerations that affect Thrashing
- **How different Operating Systems manage the MM and VM?**

# Virtual Memory

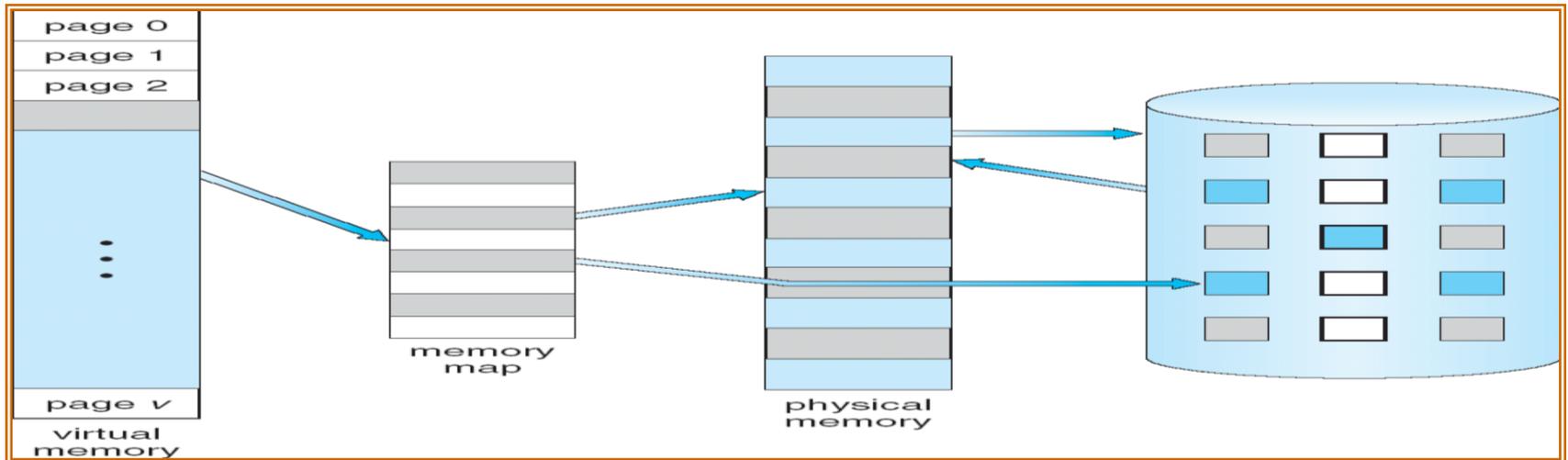
- With virtual memory support, the OS can address memory outside what is physically installed on the system.
- This non-physical memory is called Virtual Memory (VM), and
  - It is implemented by using a piece of your machine's hard disk that's set up to emulate physical memory.
  - This hard disk storage is actually a single file called a Page File, or a Swap File.
- If the physical memory is full and a page is urgently required:
  - The OS makes room for new page by taking infrequently-used page that's currently in physical memory and swaps it out to the page/swap file.
- No matter how much Main Memory (MM) your system has, Windows OS still creates and uses a page/swap file as a VM.

# Virtual Memory

- With Virtual Memory (VM): the OS can execute a process that **may not be** completely in the MM because the process size is larger than the MM size.
- VM is the OS abstraction that gives the programmer the illusion of an address space that may be larger than the physical memory address space.
  - Means allows user programs to be larger than physical memory.
- **Virtual memory:** same as the MM/PM, it can be implemented using either paging or segmentation but paging of the segments is the most common.
- **VM technique is possible due to the following facts:**
  - Only part of the process (**most frequently used pages**) needs to be brought into the MM for execution to maximize the concurrency level. i.e.
    - Error/Exception handler routines are very rarely used. **So, why don't we load them on demand to the MM/PM?**
    - **Arrays, lists or tables** are often allocated more memory than they actually need.
      - An array may be declared **100 by 100 element**, while it is rarely be **10 by 10 element** at a time. **So, the memory assigned to these arrays/tables can be used.**

## How to deal with a Process's size $>$ Size of PM?

- If the address space of the process is  $\leq$  the size of PM, then full load can be done unless we want to maximize the concurrency level through partial load.
- When the address space of the process is  $>$  the available PM.
  - A part of the process will be loaded into the PM/MM.
  - The rest of the process will be left on VM.
  - Swapping in between VM and PM will be supported through a certain policy.
- We will investigate how does the OS support that latter?



## Advantages of Partial Loading

- A process can now execute even if it is larger than the available main memory space.
  - It is even possible to use more bits for **logical addresses** than the bits needed for addressing the **physical memory**.
- More processes can be maintained in MM and this increases both the CPU utilization and throughput.
  - Only **load in** some of processes' pages.
  - With more processes in main memory, the concurrency level will be more.
- **Demand Paging:** bring a page (not a process) into MM only when it is needed. Rather than swapping the entire process, we use a lazy swapper that never swap a page into MM unless that page is needed. It leads to:
  - Less I/O needed for swapping or loading
  - Less memory needed
  - Faster response

## Support Needed for Virtual Memory

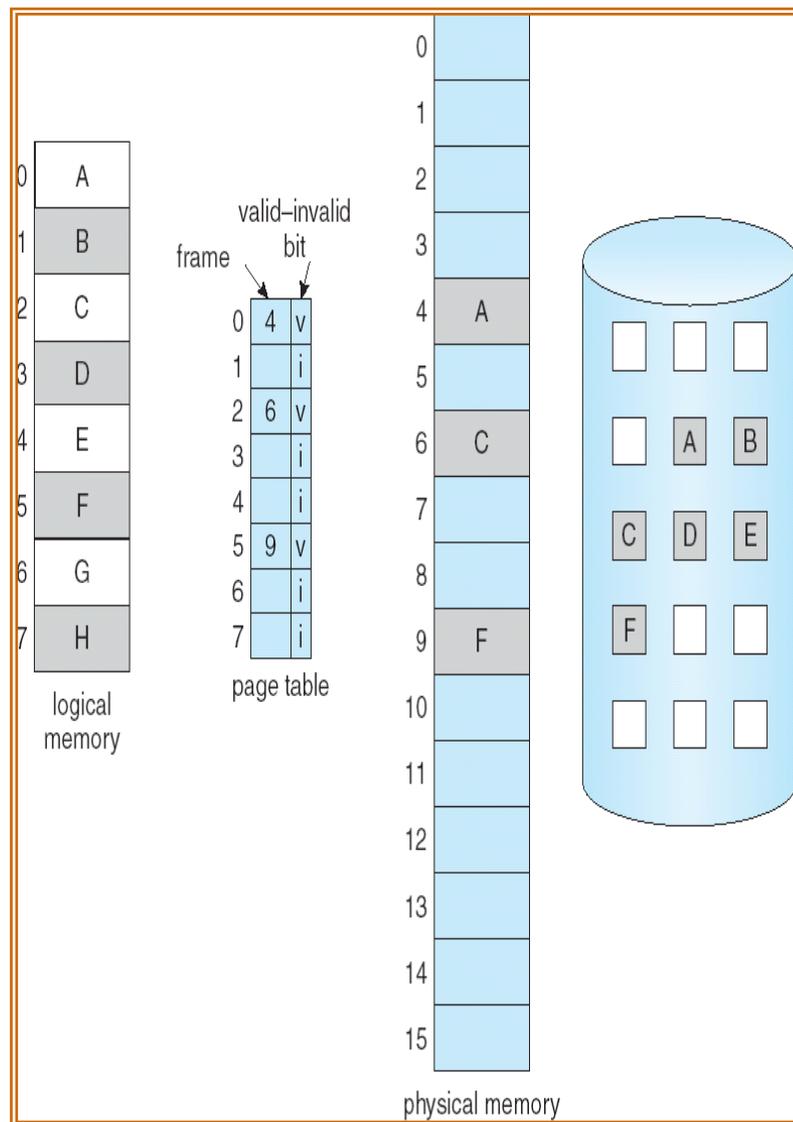
- A process needs to be broken up into **pieces** (pages or segments).
- Pages or segments of the same process do not need to be located contiguously in main memory.
- To accommodate as many processes as possible (**multiprogramming**), **only a few pages of each process is maintained in MM/PM.**
- **Memory references are dynamically translated into physical addresses at the run time.**
- The OS must not swap out a piece of a process just before that piece is needed to avoid **thrashing**.
  - **Swapper** manipulates the entire processes,
  - **Pager** is concerned with the individual pages of a process,
  - **We thus use pager in connection with demand page.**
- OS must be able to manage the movement of pages and/or segments between virtual memory and main memory through **replacement policies.**

## Process Execution with Existence of VM

- The OS brings into the main memory only few pages (**working set**) of the process.
- The **working set** is the most frequently used pages in the process.
- We will discuss later, how the working set of the process will be set?
- Each page/segment table entry has a present bit that is set only if the corresponding page/segment is in MM/PM.
- An interrupt (memory fault) is generated when the memory reference is on a page/segment that is not present in MM/PM.
- The OS places the process in a **blocking** state if it has a page/segment fault.
- The OS issues a disk I/O read request to bring the needed page/segment into the MM.
- The OS may dispatch another process to run while the disk I/O takes place.
- An interrupt is issued when the disk I/O completes to place the blocked process in the **ready** state.

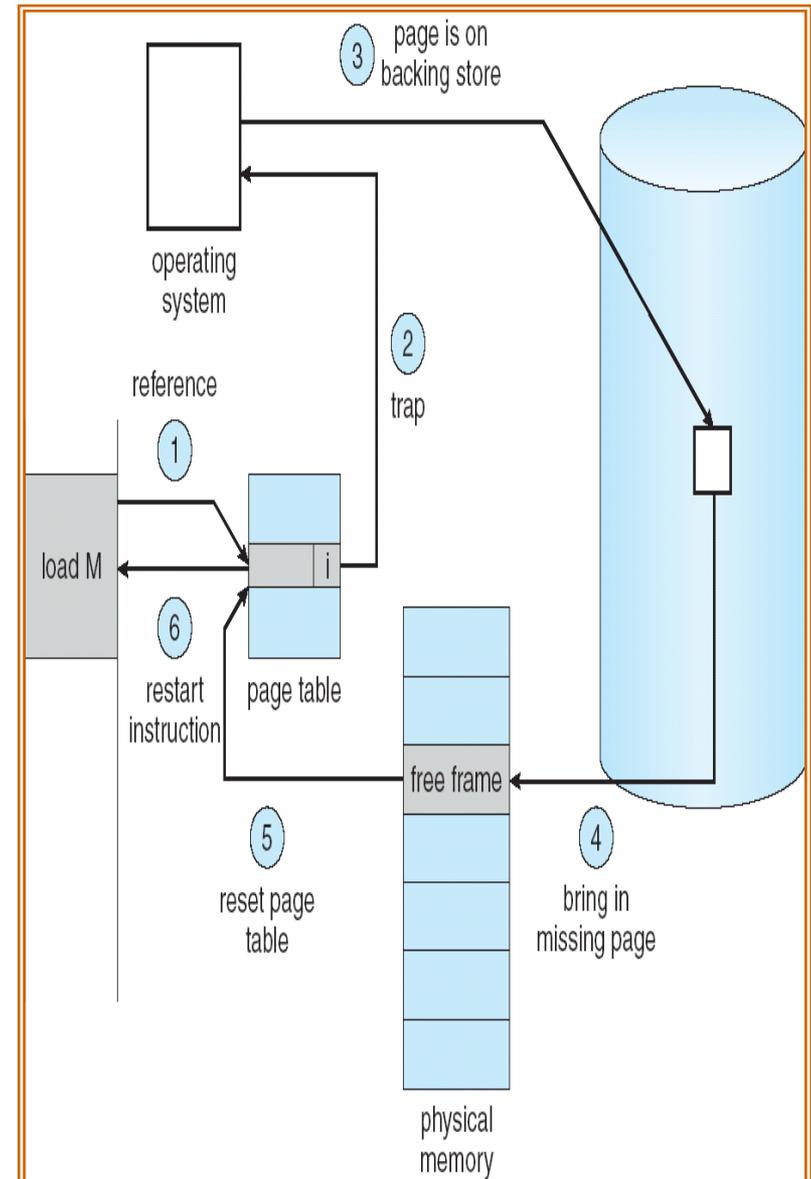
# Demand Paging

- When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again.
- Instead of swapping in a whole process, the pager brings only those necessary pages. **Thus decreasing the swapping time.**
- How to represent fact that a page of VM is not yet in memory and still on a disk?
- **With each page table entry a valid–invalid bit is associated**  
(1  $\Rightarrow$  in-memory, 0  $\Rightarrow$  not-in-memory)
- Page is needed  $\Rightarrow$  reference to valid–invalid bit
  - Invalid reference  $\Rightarrow$  Not-in-memory  $\Rightarrow$  bring to memory
- Initially **valid–invalid** but is set to 0 on all entries.
- During address translation, if **valid–invalid bit** in page table entry is 0  $\Rightarrow$  page fault, means replacement algorithm should be invoked.



# Steps in Handling Page Fault

- When a process needs a page.
  1. The OS checks the page table of this process to determine the validity of the page.
    - **Invalid reference**  $\Rightarrow$  Page fault exception.
  2. Find a free memory frame
  3. Read desired page from disk
    - Under control of I/O controller
  4. Changes invalid bit of page to **valid**
  5. Restarts process that was interrupted by the exception.
- Is it easy to restart a process? **Least Recently Used**
- What happens if there is no free frame?

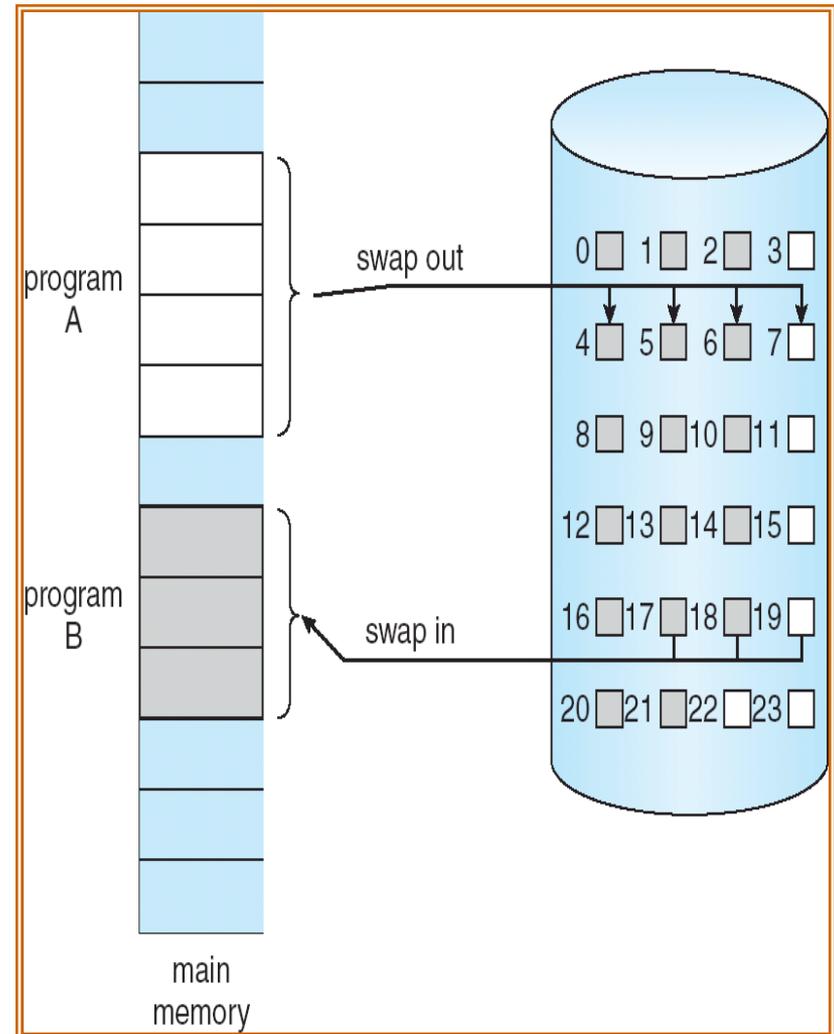


## Cost of Handling a Page Fault with VM

- Checking page table, trapping an error, finding free memory frame (or finding victim page to be swapped out) needs about 200 - 600  $\mu$ s
- Disk seek and read takes about 10 ms (**seek time** is the amount of time required for the read/write heads to move between tracks over the surfaces of the platters)
- Memory access takes about 100 ns
- Page fault = (swap page out + swap page in + restart overhead)
- Page fault degrades the performance by approximately 100 ms!
  - This doesn't even count all the additional things that can happen along the way.
- Better not have too many page faults!
- If we want no more than 10 % degradation, can only have 1 page fault for every 1,000,000 memory accesses.
- OS must do a great job of managing the movement of data between VM and MM.

## Possibility of Thrashing

- To accommodate as many processes as possible, **only a few pages of each process is maintained in MM.**
- **But MM may be full:** when the OS brings one page in, it must swap one page out
- **The OS must not swap out a page of a process just before that page is needed.**
- If it does this too often this leads to **thrashing**:
  - The processor spends most of its time swapping pages in and out rather than executing user's processes.



## Effective Access Time & Page Fault

- Demand paging can have a significant effect on the performance of a system.
- As long as there is no page faults:
- The Effective Access Time (EAT) = Memory Access time (MA).
- Let  $p$  the Page Fault Rate  $0 \leq p \leq 1.0$ 
  - If  $p = 0$  no page faults
  - If  $p = 1$ , every reference causes a fault

$$EAT = (1 - p) * \text{Memory Access (MA)} + p * (\text{page fault cost})$$

- Effective Access Time (EAT) =  $(1 - p) * MA + p (\text{page fault overhead} + \text{swap page out overhead} + \text{swap page in overhead} + \text{restart overhead})$

**Example:** Memory access time = 1 microsecond

- 50 % of the time the page that is being replaced has been modified and therefore needs to be swapped out, (means  $p = .5$ ).
- Page fault times = 15 microsecond

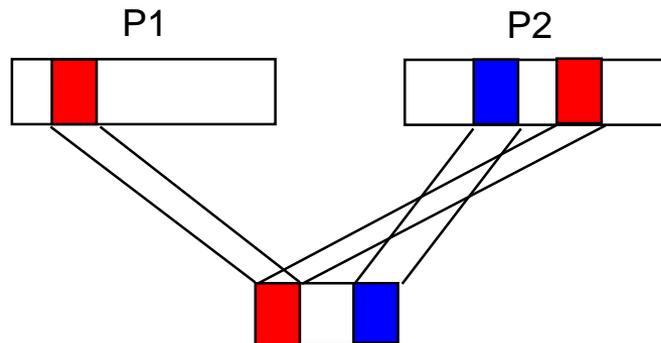
$$EAT = (1 - p) * 1 + p * (15) \text{ microsecond}$$

## Benefits of VM

- **With demand Paging:** Rather than swapping the entire process, we use a lazy swapper that never swap a page into MM unless that page is needed: this leads to:
  - Less I/O needed
  - Less memory needed
  - Faster response
- **Copy-on-Write:** The basic idea of copy-on-write is to allow one or more virtual pages of many processes (with the same contents) to be shared by loading them into the same frame/s in the MM.
- **Memory-Mapped Files:** Uses VM techniques to treat file I/O as a regular memory access.

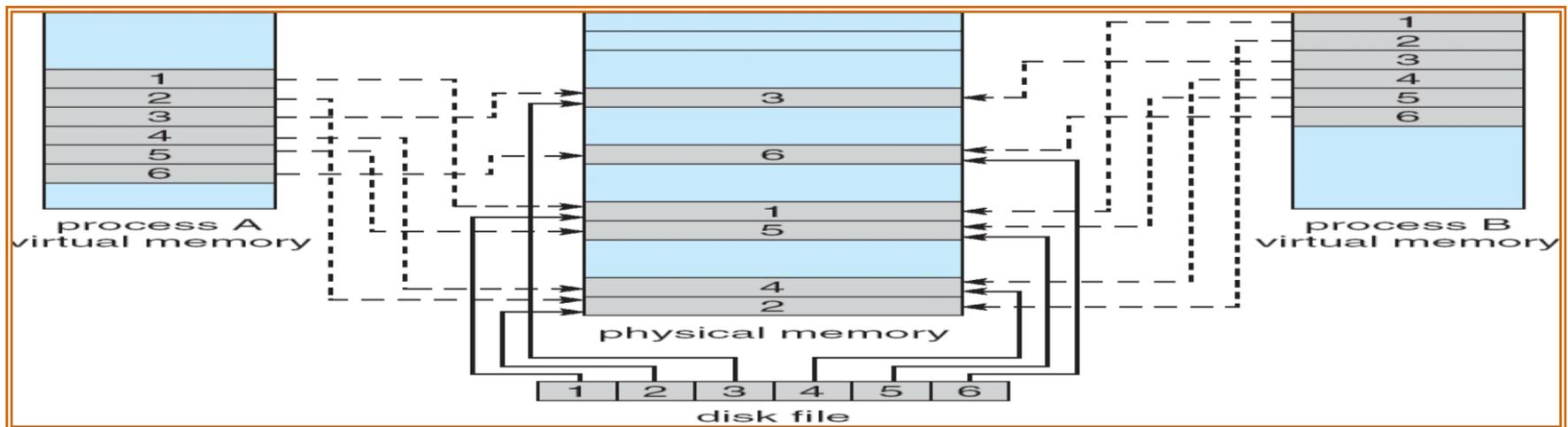
## Copy-on-Write

- Copy-on-Write (COW) allows both parent and child processes to initially **share** the same frame in memory for their pages of the same content.
- **These pages are marked as COW.** That means:
- **If either the parent or the child process modified a shared page, only then the page will be copied.**
- COW allows more efficient process creation as only modified pages are copied.
- Only pages that may be modified need to be marked as **COW**.
- **At the time of duplicating a page using COW, it is important to note where the free frame is going to be allocated from.**
- Many OSs provide a pool of free frames for such requests.
- **Free frames are allocated from a pool.**
- The allocated frames from the pool should **be zeroed-out (erased)** before being allocated. This will be done by using a technique called **zeroed-fill-on-demand**.



# Memory-Mapped Files

- Typically I/O devices have slower access time than CPU and memory.
- A system call and disk access is required every time the file is accessed.
- This requires too much CPU involvement in I/O operations.
- **Memory-mapped Files** allows file I/O operations to be treated as routine memory access by **mapping a disk block to a frame in memory.**
- A file is initially read using demand paging. A page-sized portion of the file is read from the file system into a physical frame.
- Subsequent reads/writes from/to the file are **treated as ordinary memory accesses.**
- Simplifies file access by treating file I/O through memory rather than **read(), write()** system calls.
- Also allows several processes to map the same file into the shared frames in memory if the file is shared.

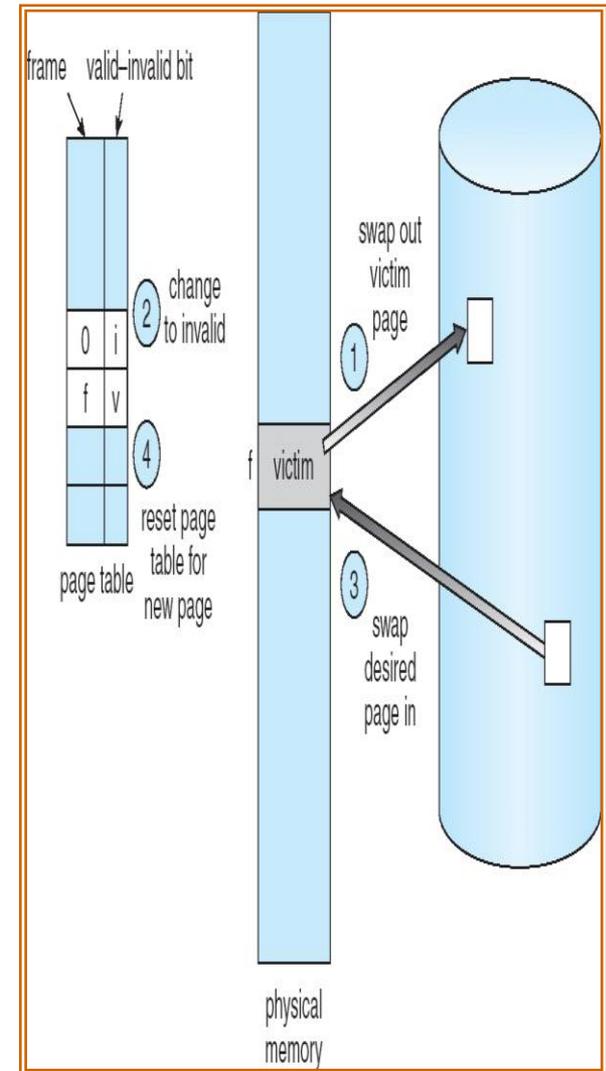


## Page Replacement

- **Page replacement:** Find a page in memory, but not really in use, swap it out. There should be an algorithm:
  - **Performance:** we need an algorithm which will result in minimum number of page faults. **It must consider that same page may be brought into memory several times.**
- **Temporal locality:** Addresses that are referenced at some time  $T_s$  will be accessed in the near future ( $T_s + \text{delta time}$ ) with high probability. Example : **Execution in a loop.**
- **Spatial locality:** Items whose addresses are near one another tend to be referenced close together in time. Example: **Accessing array elements.**

# Page Replacement

- What if there's no free frame left in the MM on a page fault?
  - **Free a frame that's currently being used**
    1. Select the frame to be replaced (**victim**).
    2. Write the victim back to disk **if it has been modified**.
    3. Change page table to reflect that the victim is now invalid.
    4. Read the desired page into the newly freed frame
    5. Change page table to reflect that the new page is now valid.
    6. Restart faulting instruction.
  - **Optimization**: no need to write the victim back if it has not been modified (**need dirty bit per page table entry**).
  - Highly motivated to find a good replacement policy
    - How do we choose the best victim in order to minimize the page fault rate?
  - **Is there an optimal replacement algorithm? If yes, what is it?**



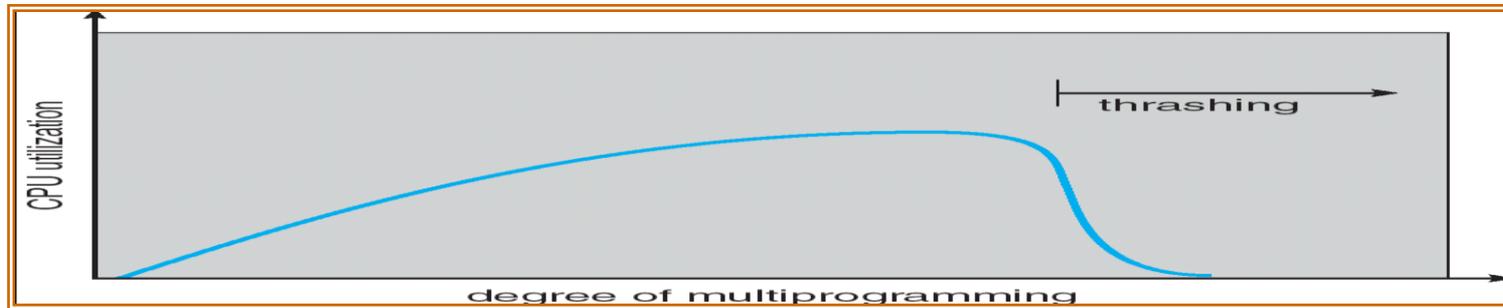
## Replacement Policy

- The OS uses some policies to select a frame in MM to be replaced when a new page is required to be brought into the MM.
- This occurs whenever the MM is full (there is no free frame available).
- Not all frames in the MM can be selected for replacement.
- Some frames are locked (cannot be paged out): i.e.
  - Frames allocated to the OS kernel,
  - Frames used for data structures used in the management,
  - Frames used for memory mapped files or for buffering or spooling.
  - etc..
- The replacement policy should have lowest page-fault rate.
  - We can evaluate the replacement policy by running it on a particular string of memory references (**reference string**) and compute the number of page faults [needed to do the replacement] on that string.

## Global vs. Local Replacement

- The OS might decide that the set of pages considered for replacement be:
  - **Local**: Limited to those of the process that has suffered the page fault.
  - **Global**: The set of all pages in unlocked frames.
- **Local Replacement**: the OS selects for replacement a frame from the allocated frames of the same process.
  - The set of pages in memory for a process is affected by the paging behavior of only that process.
- **Global Replacement (GR)**: OS selects for replacement a frame from the set of all frames assigned to any process; one process can take a frame from another.
  - For a high priority process, the OS can select from either its own frames or from the frames of any lower priority process.
  - **This means a high priority process can increase its frames at the expenses of the low priority process.**
  - A process can not control its own page fault rate.
  - The thrashing depends not only on the paging behavior of that process but also on the paging behavior of other process.
- A bad replacement choice increases the page fault rate and slow process execution, but does not cause incorrect execution.

## Thrashing with Global Replacement



- To maximize the CPU utilization, OS increases the degree of concurrency by loading more processes to the MM.
- With the use of **global replacement**, process's page fault affects each other and this causes propagation of **thrashing**.
- As the degree of multiprocessing increases the CPU utilization increase until a maximum is reached. **If the degree of multiprocessing increases further, thrashing propagates and the CPU utilization drops sharply.**
- **Domino-style thrashing:** if one process has page faults, evicting another process' page, and when this process runs, it will evict yet another process' pages, etc.
- **We can limit the effects of thrashing by:**
- **By using local replacement:** if one process starts thrashing, it can not steal frames from another process and causes it to thrash too.
- **By predicting its working set and provide a process as many frames as it needs.**
- **How do we the working set:** by looking at how many frames a process is currently using and predicts the future needs. We will see how in the coming slides.

# The First-In-First-Out (FIFO) Policy

- **First-In-First-Out**

- Be fair, let every page lives in memory for about the same amount of time, then replaces it.

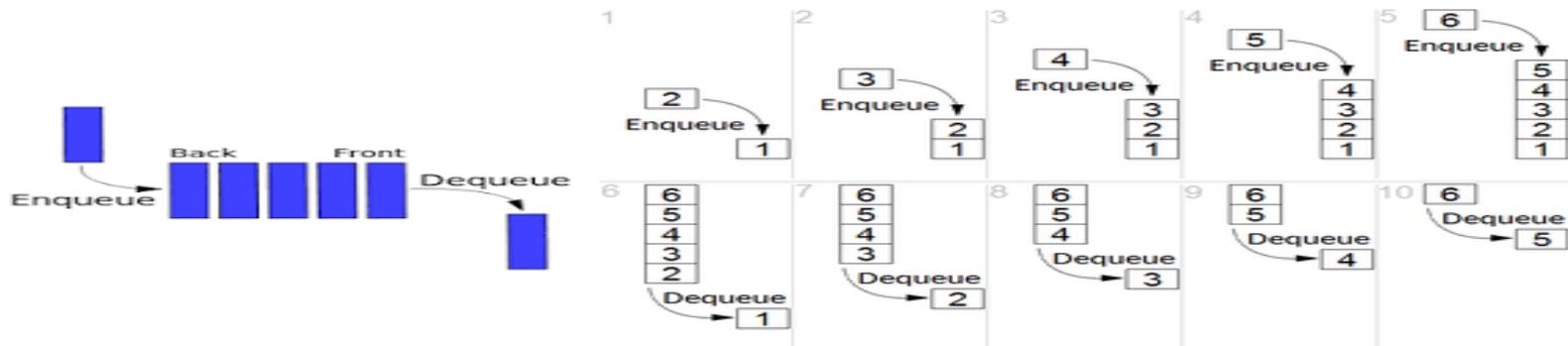
- Treats page frames allocated to a process as a FIFO queue.

- When the buffer is full, the oldest page is replaced, **the one at the head of the queue.**

- A frequently used page is often the oldest, so it will be repeatedly paged out by FIFO.

- Simple to implement

- Requires only a FIFO queue.

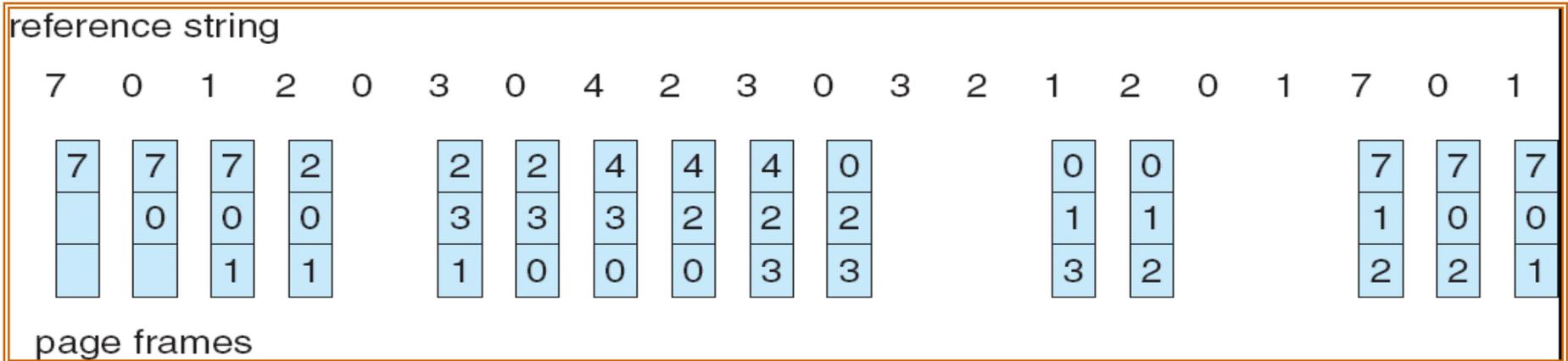


**FIFO Queue**

# FIFO Page Replacement: Example-1

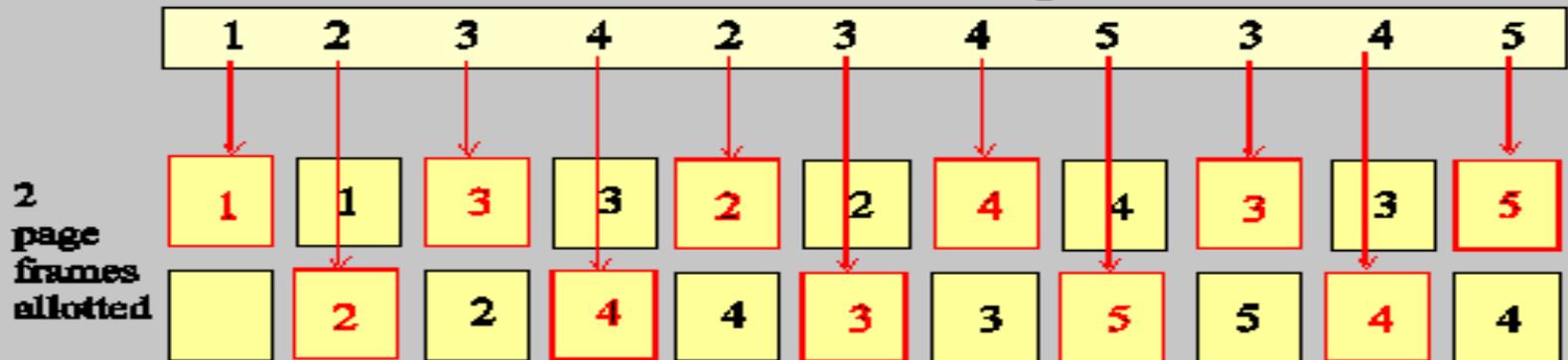
1. For the following example: the three frames are initially empty. The first three references (7,0,1) cause page faults and are brought into these empty frames.
2. The next reference (2) replaces page 7, because page 7 was come in first.
3. Since 0 is the next reference and 0 is already in memory, we have no fault for this reference.
4. The first reference to page 3 results in page 0 being replaced, since it was the first of three pages in the memory (0,1,2).
5. Because of this replacement, next reference to 0 will cause page fault.
6. Page 1 is then replaced by page 0.

15 page faults



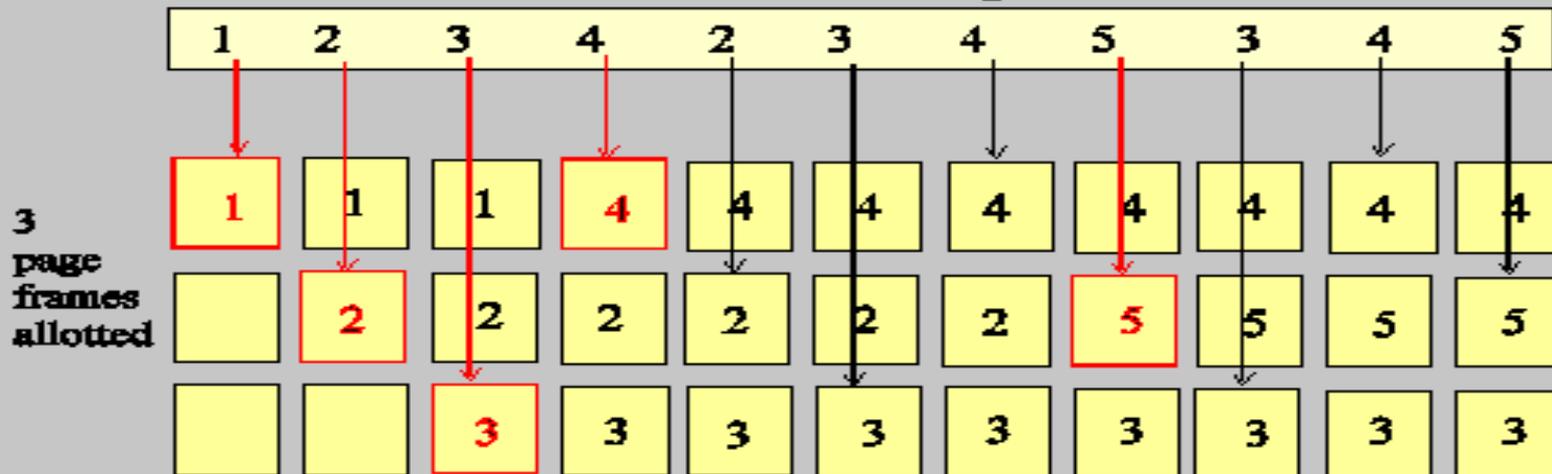
# FIFO Page Replacement: Example-2

☀ Reference String



FIFO Replacement 11 page faults

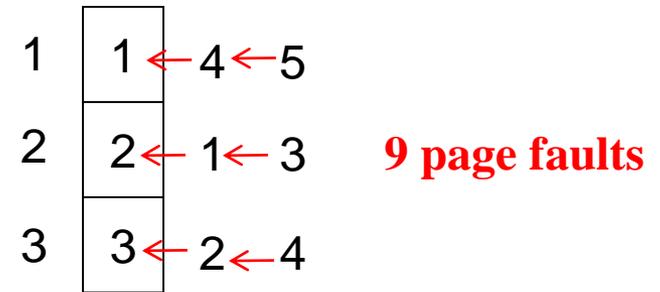
Reference String



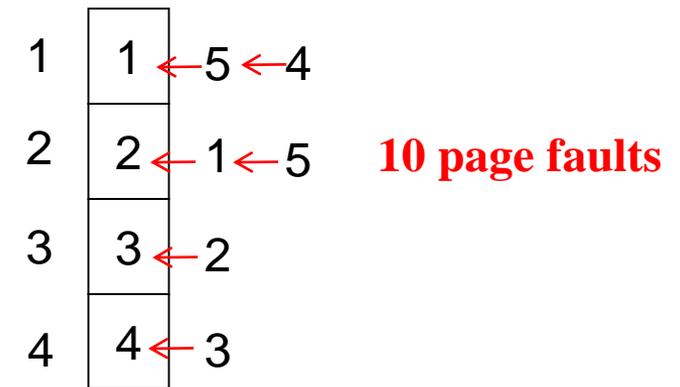
FIFO Replacement 5 page faults

## FIFO Algorithm and the # of Frames

- To demonstrate the relation between the number of allocated frames and the number of faults, consider the following example:
- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames (3 pages can be in memory at a time per process)



- 4 frames (4 pages can be in at a time)



# FIFO Illustrating Example: Example-3

## First In - First Out Page Replacement Algorithm

### Memory Reference String

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

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

17 page faults

### Memory Reference String

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

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

14 page faults

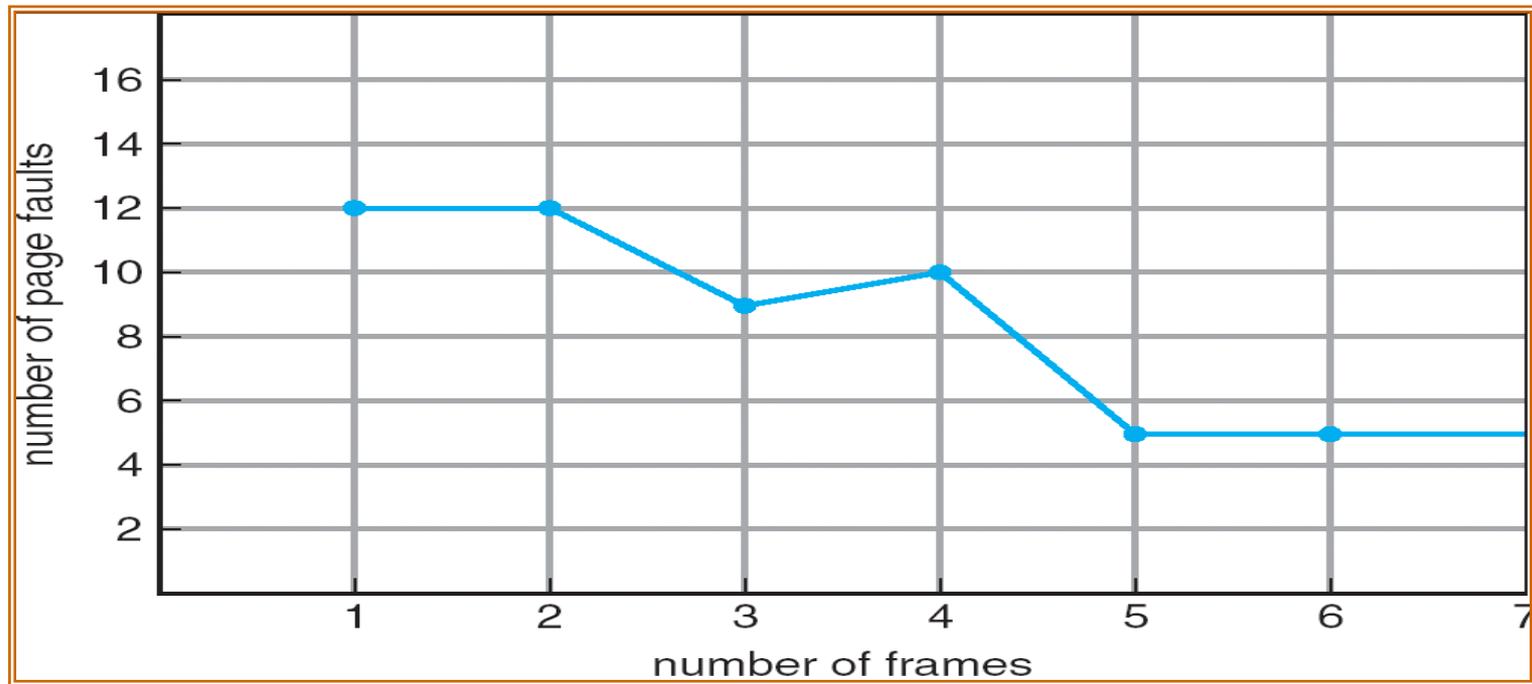
### Memory Reference String

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

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

15 page faults

## FIFO Illustrating Belady's Anomaly



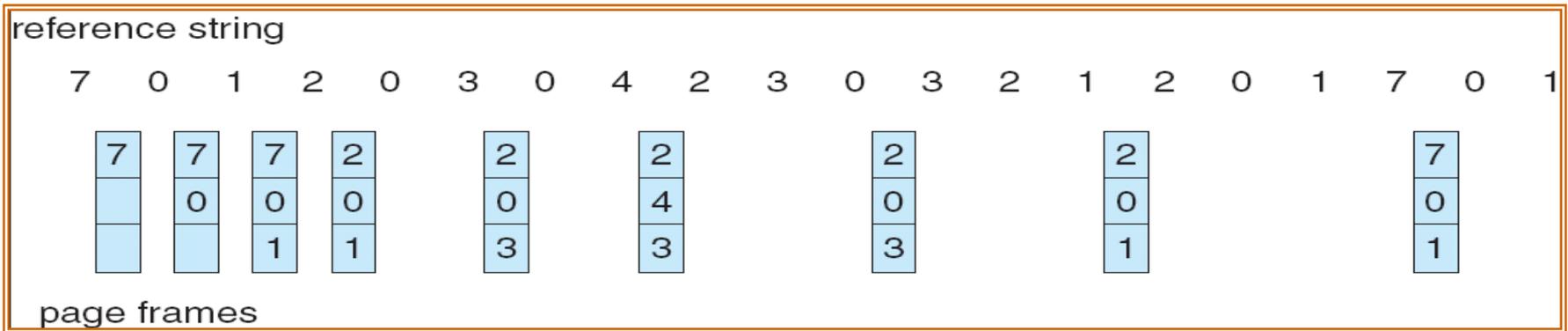
1. We would expect that giving more memory to a process improves the performance, **More frames  $\Rightarrow$  less page faults**
2. This most unexpected result is known as Belady's anomaly (**deviation from the common rule** )

# Optimal Page Replacement (OPR)

- In OPR algorithm/policy, the OS selects for replacement the page that will not be used for the longest period of time.
- Assume pages used recently will be used again soon.
  - Eject out a page that will not be used for longest time
- Keep a time counter in each page table entry
  - Choose page with lowest time value in the counter
  - Periodically zero the counter
- This algorithm has the lowest page fault rate of all algorithms and never suffers from Belady's anomaly.
- What's the problem with this algorithm?
- Policies to predict future references on the basis of past behavior.
- It requires a great deal of prediction and searching overhead.

## OPR Example

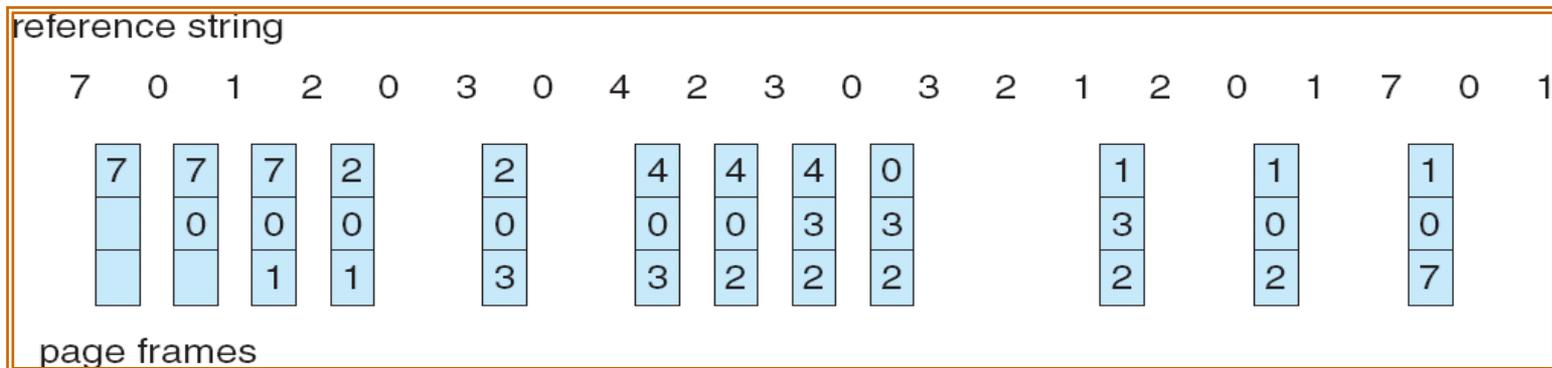
1. The three frames are initially empty. The first three references (7,0,1) cause page faults and are brought into these empty frames.
2. The next reference (2) replaces page 7, because page 7 will not be used until reference 18, whereas page 0 will be used at 5, and page 1 at 14.
3. The reference to page 3 replaces page 1, as page 1 will be the last of the three pages in memory to be referenced again.
4. It makes 9 faults while FIFO made 15 faults.
5. If we consider the first 3 faults common for all algorithms then OPR is twice good of FIFO. The page fault frequency of OPR is 50% less than FIFO.



9 Page Faults

# The Least Recent Used (LRU) Page Replacement

- The OS replaces the page that has not been referenced for the longest time.
- Least Recently Used implementation:
  - On access to a page, timestamp it (Each page could be tagged in the page table entry)
  - The LRU page is the one with the smallest time value
  - This would require expensive hardware and a great deal of searching overhead.
    - On a page fault, choose the one with the oldest timestamp
    - What's the motivation here?
  - In practice, LRU is quite good for most programs
    - Easly to be implemented by using a linked list of pages.
    - Most recently used at front, least at rear and needs to update this list every memory reference !



12 Page Faults

# OPT and LRU Comparison

- **Example:** A process of 5 pages with an OS that allocates 3 frames to the process. What is the number of page faults if we use OPT and LRU?
  - For comparison reasons, **we are not counting initial page faults when the memory is empty.**

## 3 Page Faults

Page address stream

2 3 2 1 5 2 4 5 3 2 5 2

OPT

2	2	2	2	2	2	4	4	4	2	2	2
	3	3	3	3	3	3	3	3	3	3	3
			1	5	5	5	5	5	5	5	5
				F		F			F		

LRU

2	2	2	2	2	2	2	2	3	3	3	3
	3	3	3	5	5	5	5	5	5	5	5
			1	1	1	4	4	4	2	2	2
				F		F		F	F		

## 4 Page Faults

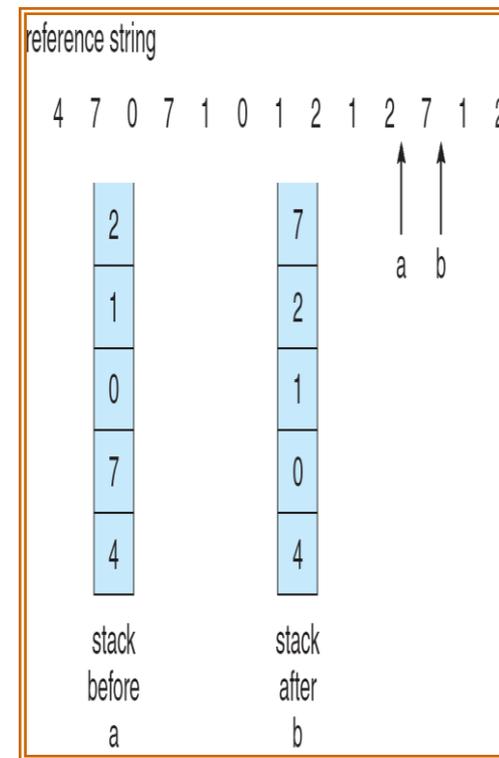
# LRU and MRU Implementation Methods

## □ Counter Implementation

- Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter.
  - When a page needs to be changed, look at the counters to determine which one will be selected.
  - **Least Frequently Used (LFU) Algorithm:** Replaces page with smallest count. Based on the argument that the page with the smallest count was probably just brought in and has yet to be used.
  - **Most Frequently Used (MFU) Algorithm:** Replaces page with largest count
- When we use Most Frequently Used Algorithm to replace pages, it means the page with the largest count will be replaced. In this case if the page has been referenced many times during the initial phase of the process then its counter will be high and will be replaced. One way to solve this problem and to keep this page in memory is to shift the counter right by one at regular intervals to reflect the usage. I.e. if we used 3 bites to represent the counter, then if it has the value “TWO”, means its value in the binary system will be 010. If we shifted the bites to the right, the new value will be 001 “ONE” means shift to right means divide the value by TWO. By this way, the counter will be decreased periodically to keep the page in memory if it is important to avoid its replacement.

# LRU and MRU Implementation Methods

- **Stack implementation:** Keep a stack of page numbers.
  - Whenever a page is referenced:
    - Move it to the top
    - **The top of the stack is always** the most recent used (MRU), **while the bottom is the LRU**
  - No search for replacement
- **Reference bit**
  - With each page associate a reference bit, initially = 0
  - When page is referenced (**read/write to any byte of the page**) **reference bit set to 1.**
  - After some time, the OS can determine which pages have been used and which have not been used by examining the reference bit.
  - **Replace the one which has a reference bit=0** (if one exists).
    - We do not know the order, however.

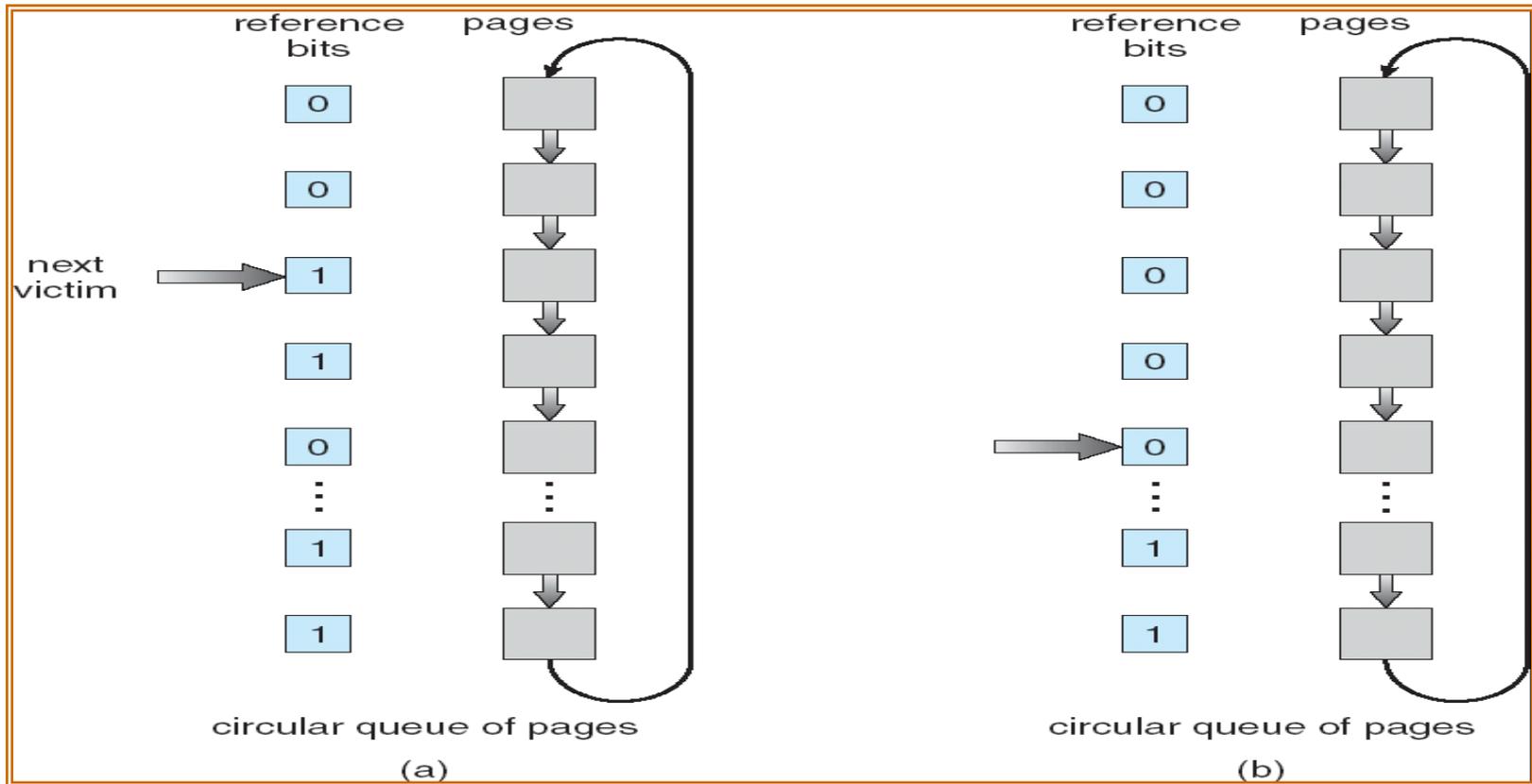


## Second-Chance (Clock) Implementation

- Arrange physical frames in a circle queue, with a clock hand.
- Associate a **reference bit** per frame.
- Sets the **reference bit** on memory reference to a frame.
- If the **reference bit** is not set, a page hasn't been used for a while
- On page fault (a frame is needed):
  1. Advance clock hand
  2. Check the **reference bit**,
    - If it is = 1, means the page has been used recently, clear it, give it second chance and look for the another page.
    - If it is = 0, this is our victim and the new page will be inserted in that position.
    - In the worst case, when all bits are set, the pointer cycles through all of the pages, giving each page a second chance and clear its bit.
- Can we always find a victim? **If no then use FIFO.**

## Second-Chance (clock) Page-Replacement Algorithm

- Second Chance policy is an improved version of FIFO. **This is referred to as the Clock policy.**



## Enhanced 2nd-Chance Algorithm

- Associate a pair of (reference and modify) bits with each frame, (Unix versions) this means we have four possible 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 but modified.

## Page-Buffering Algorithm

- The OS preserves a pool of free frames.
- When a page fault occurs, instead of waiting to look for a victim frame using one of the replacement policies.
- The OS provides a free frame from the pool and reads the required page into the free frame.
- This procedure allows the process to restart soon without waiting for finding the victim page and/or writing it out.
- The OS may employ page replacement policy and when the victim page is written out, its frame is added to the pool of free-frames.

## Allocation of Frames to a Process

- How many frames should be allocated to a process?
- Shall we allocate the frames based on the size of process?
- Shall we allocate the frames based on the working Set of the process?
- Shall we allocate the frames based on the Page Fault Frequency?
- Shall we allocate the frames based on the Process Priority?
  - **Working set** is set of pages in memory and has referenced in the last N seconds/references (not easily known at the startup).
  - Allocating a process fewer frames than its working set can quickly lead to more page faults.
  - One way to prevent many page faults is to avoid scheduling a process unless it is allocated enough frames for its working set.
  - Allocating a process more frames than its working set will minimize the concurrency level (minimize memory utilization).
- **Page fault frequency rate** measures the page fault rate of the process.
  - If it is too low: it means the process has been given more frames than it needs.
  - If it is too high: it means the process has been given less frames than it needs.
  - If too many processes have high page fault frequency then swap out one of these processes and reassign its frames to the faulting processes.

## Frames Allocation Policy

- Shall we allocate frames equally or proportionally to processes?
- Which one is more fair and decreases the page fault frequency?
- **Equal allocation:** If  $m$  (100) frames and  $n$  (5) processes, then give each  $m/n$  (20) frames. The remainder can be used as free-frame pool.
- **Proportional allocation:** Allocate available frames to each process according to the size of process.

$s_i$  = size of process  $p_i$

$$S = \sum s_i$$

$m$  = total number of frames

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

$$m = 64$$

$$s_1 = 10$$

$$s_2 = 127$$

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

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

- A proportional allocation scheme based on priorities rather than size can be used.

## Working Set Model

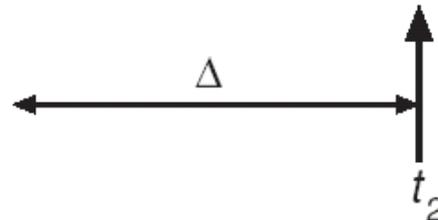
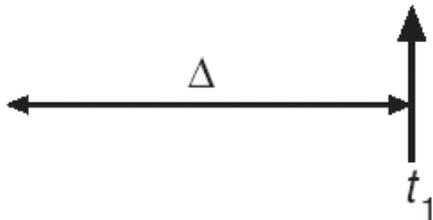
- The set of pages that are in memory and have been referenced in the last time interval  $\Delta$ .
- The working set model is based on the assumption of locality (set of pages that are actively used together and **local replacement is used**).
- Do you think the size of the working set is static or varies during the execution of the process depending on the locality of accesses?
- If the number of frames allocated to a process covers its working set then the number of page faults will be small.
- Schedule a process only if there is enough free frames more than or equal to its working set.

# The Working Set Strategy

- The working set for a process at time  $t_1$ ,  $WS(\Delta, t_1)$ , is the set of pages that have been referenced in the last virtual time units  $t_1$ .
  - Virtual time = time elapsed while the process was in execution (i.e.: number of instructions executed).
  - $\Delta \equiv$  working-set window  $\equiv$  a fixed number of page references.
  - $\Delta$  is used to define the working set window, **the pages that in active use**.
  - If the page is no longer being used, it will drop from  $\Delta$ .
  - $WS(\Delta, t)$  is an approximation of the program's locality.
  - If  $\Delta=10$  as shown, then the working set at time  $t_1$  is 5 (pages #1,2,5,6,7) and at time  $t_2$  is 2 (pages # 3,4).

page reference table

. . . 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 . . .



## Working-Set Model

- We compute the working set size of process  $P_i$  ( $WSS_i$ ) = Total number of pages referenced in the most recent  $\Delta$  (varies in time)
  - The optimal value for  $\Delta$  is unknown and time varying
  - if  $\Delta$  too small will not overlap the entire locality.
  - A **locality** is a set of pages that are actively used together.
  - if  $\Delta$  too large will overlap several localities.
  - if  $\Delta = \infty \Rightarrow$  will overlap entire process locality.
- $D = \sum WSS_i$  = total demand frames from all processes.
- If  $D > m$  (*total number of available frames*)  $\Rightarrow$  **Thrashing** will occur because some processes will not have enough frames.
- The OS monitors the working set of each process and allocates to that process enough frames to cover its working set size.
- **If there enough extra frames, another process can be initiated.**
- If  $D > m$ , then suspend one of the processes.
- The pages of the suspended process are written out and reallocated to other processes. **The suspended process can be restarted later.**
- **The problem here is keeping track of the working set which is a dynamic one, pages may be dropped out or newly come.**

## The Working Set Strategy to avoid Thrashing

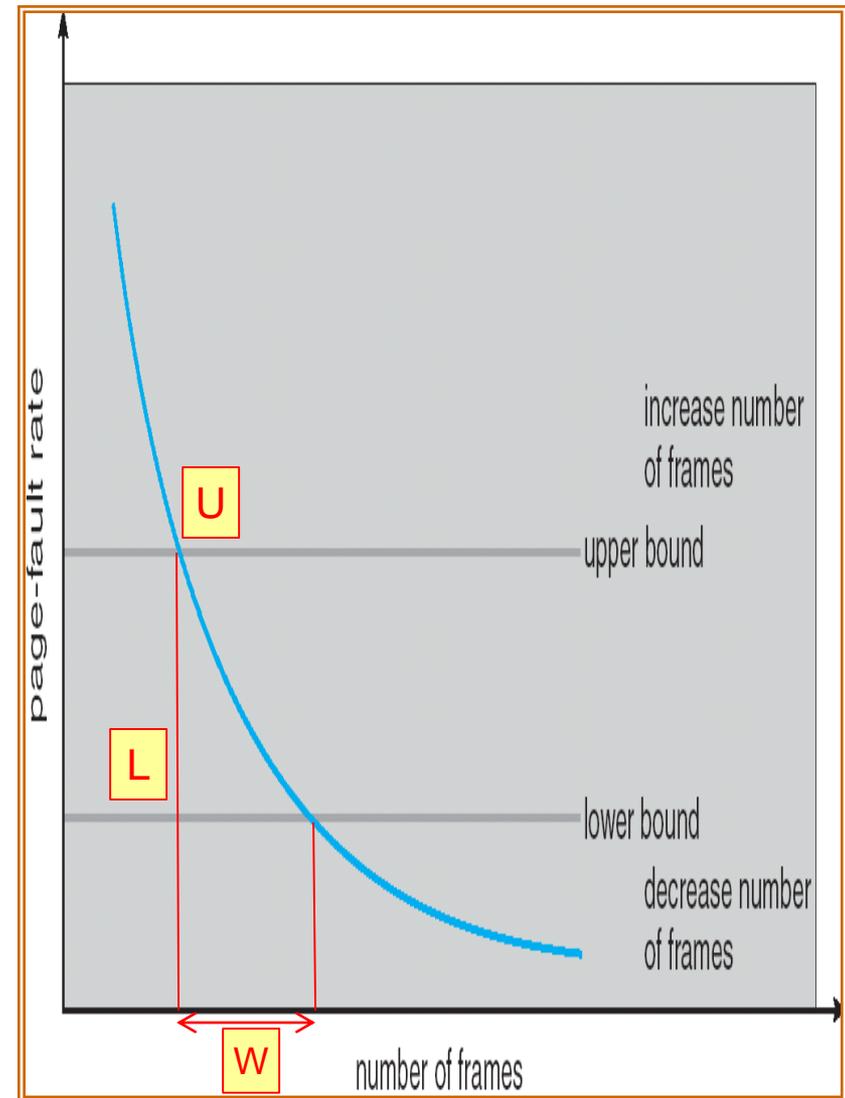
- The working set concept suggest the following strategy to determine the resident set size
  - Monitor the working set for each process.
  - Periodically remove from the resident set of a process those pages that are not in the working set.
  - When the resident set of a process is smaller than its working set, allocate more frames to it.
    - If not enough free frames are available, suspend the process (until more frames are available).
      - i.e.: a process may execute only if its working set is in main memory
- Practical problems with this working set strategy
  - Measurement of the working set for each process is impractical
    - Necessary to time stamp the referenced page at every memory reference.
    - Necessary to maintain a time-ordered queue of referenced pages for each process.
- Solution: rather than monitor the working set, monitor the page fault rate!

## Page-Fault Frequency to avoid Thrashing

- A counter per process stores the # of faults (virtual time between page faults).
- An upper threshold for # of faults (the virtual time) is defined .
- If the # of faults (amount of time since the last fault) is **greater than** the threshold (i.e. page faults are happening at a high rate), then add new frame to the resident set.
- If the # of faults is **less than** a lower threshold then discard frames from the resident set.

# The Page-Fault Rate(PFR) Strategy to avoid Thrashing

- Define an upper bound  $U$  and lower bound  $L$  for Page Fault Rates (PFR).
- Allocate more frames to a process if  $PFR \geq U$ .
- Allocate less frames if  $PFR < L$ .
- The resident set size should be close to the working set size  $W$
- We suspend the process if the  $PFR > U$  and no more free frames are available.



# Ch. 9 Virtual Memory (VM)

- **Introducing of the Virtual Memory**
  - What do we want to achieve by using virtual memory?
  - What are the problems do we address through virtual memory?
  - How does virtual memory work?
  - The meaning of Demand Paging, Thrashing concept,
  - Page Fault and the Cost of Handling a Page Fault,
- **Page Replacement Polices**
  - What are the Advantages of Virtual Memory? (less I/O time, copy on write, memory mapped files)
  - Page replacement strategies (not all pages can be selected for replacement, why?)
  - Local and Global replacement strategies, advantages and disadvantages of each approach.
  - Thrashing with Global Replacement
  - First In First Out (FIFO), (Examples with different frame numbers allocated), Belady's anomaly)
  - Optimal Page Replacement (OPR) with examples,
  - Least Recent Used (LRU), Most Recent Used (MRU),
  - Second Choice Page-Replacement, Enhanced version of Second Choice Page-Replacement,
  - Page-Buffering Algorithms.
  - Relationship between the allocated frames and the page fault frequency.
- **Allocation of Frames to processes to minimize Thrashing**
  - Based on the size of the process or Based on the working set of the process or based on the Fault frequency rate
    - **Other Considerations that affect Thrashing**
- **How different Operating Systems manage the MM and VM?**

## Other Considerations to avoid Thrashing

- Pre-paging

- Means we want to minimize the number of initial page faults by bringing **at once into memory all the pages that will be needed.**
- We keep with each process **a list of the pages in the working set.** If we want to suspend the process due to a lack of free frames or I/O, **we should remember the working set of that process.**
- When the process is to be resumed, **the OS automatically brings back into memory the entire working set for that process.**
- There is a trade of between **the cost of page fault** and **consuming the memory by some of currently unused pages from the working set.**

## Other Considerations to avoid Thrashing

- Related to the **hit ratio** (Percentage of times that a page number is found in the TLB) is a similar metric called **TLB Reach**.
- **TLB Reach**: The amount of memory accessible from the TLB.
- $\text{TLB Reach} = (\text{TLB Size}) \times (\text{Page Size})$  (should be increased)
- **Ideally, the working set of each process is stored in the TLB.** Otherwise the process consumes time resolving memory reference in page table rather than in TLB.
- If we double the number of entries in TLB, we double the TLB reach.  
**Another approach for increasing TLB Reach is either:**
- **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 processes, that require larger page sizes, to use them without an increase in fragmentation.

## Page Size: Trade-off

- Page size selection
  - There is no a decision regarding the best page size, there is a set of factors that support various sizes.
  - Table size
    - Decreases the page size, increases the number of pages and hence increases the size of the page table.
  - Fragmentation
    - Memory is better utilized with smaller pages
  - I/O overhead
    - I/O time is composed of seek, latency, and transfer times where transfer time is proportional of page size.
  - Locality
    - With small page size, locality will be improved, a small page size allows each page to match program locality more accurately.
      - A **Locality** is a set of pages that are actively used together
      - As a process executes, it moves from locality to locality
        - » **Example**: Entering a subroutine defines a new locality
      - Programs generally consist of several localities, some of which overlap

## Other Considerations to avoid Thrashing

- Program structure

- The system performance can be improved if the user/compiler/OS has an awareness of the underlying demand paging.

- Assume **that pages are 128 words** in size. Consider a program whose function is to initialize to 0 each element of 128 by 128 array.

- `int A[][] = new int[128][128];`

- Program 1:

```
for (int j = 0; j < A.length; j++)
    for (int i = 0; i < A.length; i++)
        A[i][j] = 0;
```

Notice that the array **is stored row major** `A[0][0], A[0][1], A[0][2], A[0][3], A[0][127], .....` `A[127][0], A[127][1], ... A[127][127]`.

- If the OS allocates **less than 128 frames to the entire program**, then it causes  $128 \times 128 = 16384$  page faults.

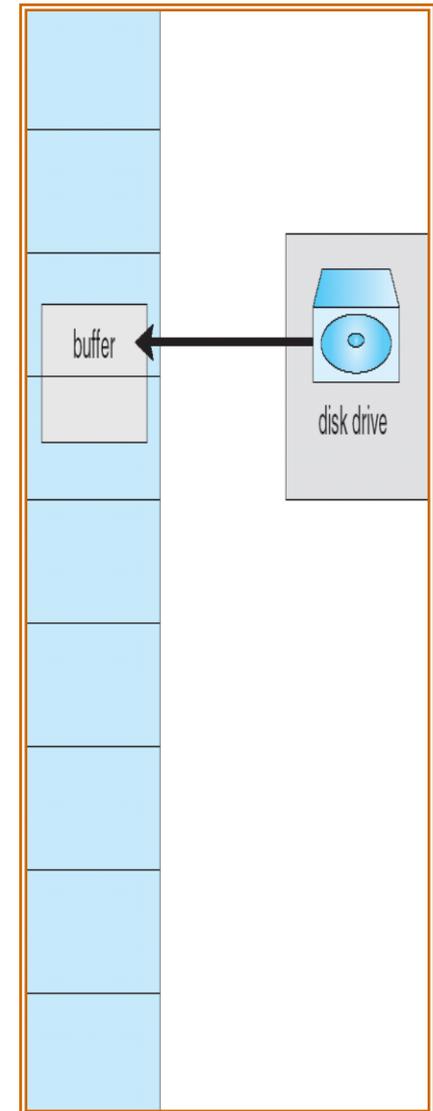
- Changing the code to Program 2:

```
for (int i = 0; i < A.length; i++)
    for (int j = 0; j < A.length; j++)
        A[i][j] = 0;
```

- Zeros all the words on one page before starting the next page reducing the number of page faults to 128.

## Other Considerations to avoid Thrashing

- The compiler and loader can have a significant effect on paging. Separating code and data generating a reentrant code. This means pages can be read only and hence will never be modified. Non modified pages need not to be paged out to be replaced.
- The choice of the programming language can affect paging as well. C and C++ use pointers frequently and **pointers tend to randomize access to memory, thereby potentially diminishing a process locality.**
  - OOPs also tend to have a poor locality of references.
  - **I/O Interlock:** Pages must sometimes be locked into memory if they used as buffers for I/O operations.
  - **Consider I/O:** Pages that are used for file-mapping must be locked from being selected for eviction by a page replacement algorithm.



## OS Examples: Unix Paging Policy

- Demand paging
- Page replacement algorithm
  - Maintain a certain number of free frames (within a min/max range)
  - Swaps out processes when number of free pages is below min.
  - Unix uses 2-handed clock for page replacement policy.

## OS Examples: Linux Paging Policy

- Demand paging
- Maintain a certain range of free frames
- Each process on a 32-bit machine is given 3 GB of virtual address space and 1 GB reserved for page tables and other kernel data.
- 3-level page table
- Kernel is never paged out.

## OS Examples: Windows NT Paging Policy

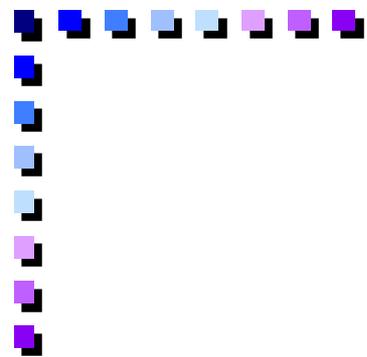
- Demand paging
- Maintain a certain number of free frames
- For 32-bit machine, each process has 4 GB of virtual address space
- Uses working sets (per process)
  - Consists of pages mapped into memory and can be accessed without page fault
  - Has min/max size range that changes over time
    - If page fault occurs and **working set < min**, add page
    - If page fault occurs and **working set > max**, evict page from working set and add new page
    - If too many page faults, then **increase size of working set**
- When evicting pages,
  - Evict from large processes that have been idle for a long time before small active processes.
  - Consider foreground process last

## OS Examples: Windows

- Uses demand paging with clustering. Clustering brings in pages surrounding the faulting page.
- Processes are assigned working set minimum and working set maximum.
- Working set minimum is the minimum number of pages the process is guaranteed to have in memory.
- A process may be assigned as many pages up to its working set maximum.
- When the amount of free memory in the system falls below a threshold, automatic working set trimming is performed to restore the amount of free memory.
- Working set trimming removes pages from processes that have pages in excess of their working set minimum.

## OS Examples: Solaris 2

- Maintains a list of free frames to assign faulting processes.
- **Lots-free** – threshold parameter to begin paging.
- Paging is performed by *page-out* process.
- Page-out scans pages **using modified clock algorithm**.
- **Scan-rate** is the rate at which pages are scanned. This ranged from **slow-scan** to **fast-scan**.
- Page-out is called more frequently depending upon the amount of free memory available.



**The End!!**

**Thank you**

**Any Questions?**

