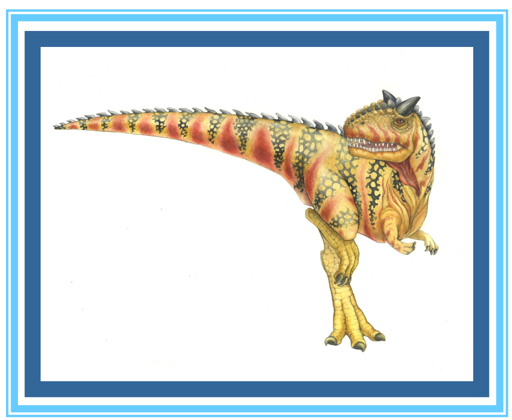
Section 10: Virtual Memory



Virtual Memory

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

Objectives

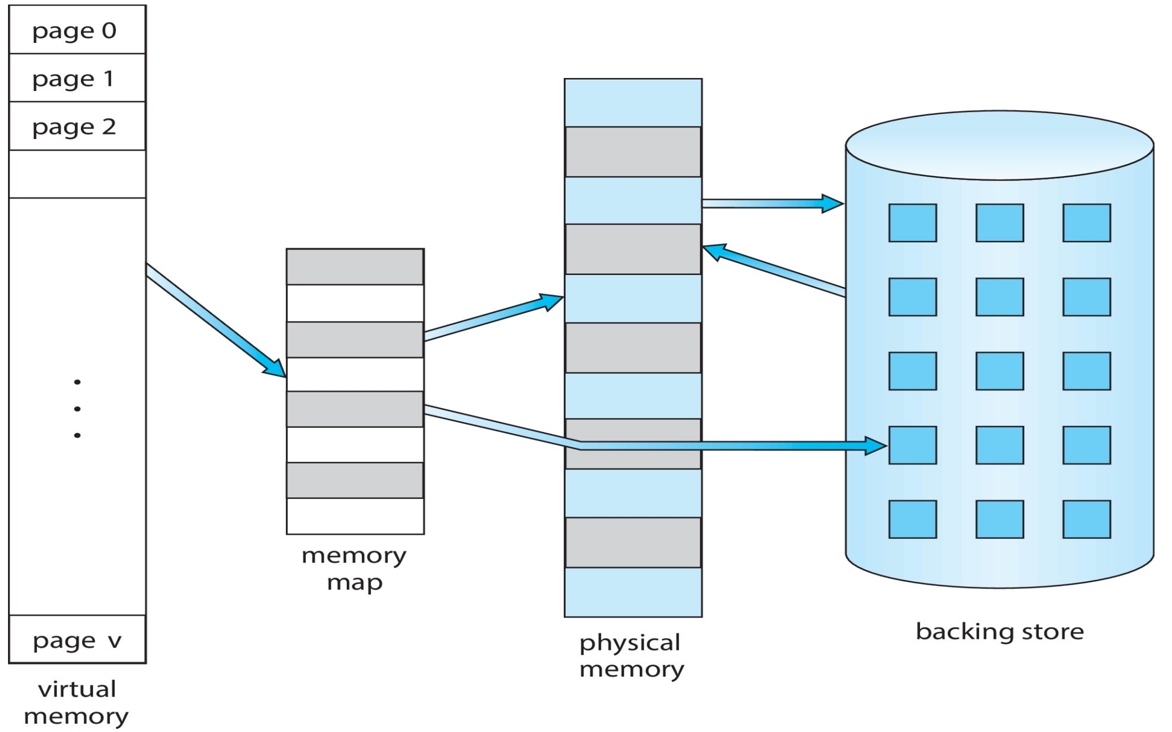
- In the previous section, paging is used to allocate memory blocks (frames) to store processes non-contiguously in main memory
- In this section we show how paging extend logical address space beyond the size of the physical address space
- We introduce demand paging, page-fault handling, page replacement algorithms, frame allocation to processes, etc.

Virtual Memory

- ***Virtual memory*** is a technique to execute processes which are not completely in memory
  - Only part of a process is in main memory during execution
  - The rest of the process stay on the hard disk

- This memory is called “virtual” because the part of a process that is still on the hard disk is managed by the OS as if it was in main memory
  - Hard disk memory is huge, it is seen as part of main memory, so virtual memory is huge as well
  - This large memory is virtual however, if the part of a process on the hard disk is needed, it must be loaded into main memory

Some pages have no frame

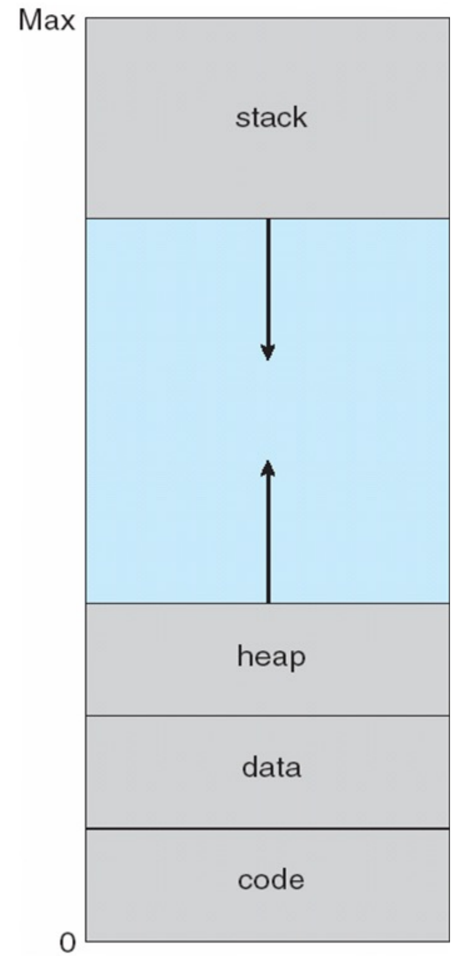


- The virtual address space could be larger than the physical (main) memory, not all pages are allocated frames, the other are on hard disk

- Entries of the page table for which no frame is allocated are empties

Example of virtual memory usefulness

- Virtual address spaces include **holes**
  - Large empty space between the heap and stack
  - Holes can be filled during process execution
  - With virtual memory, wholes are not allocated any frame unless they are filled



Implementation: Demand Paging

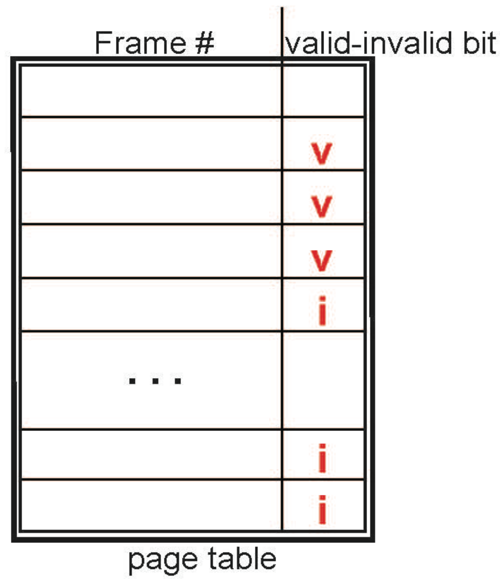
- Pages loaded when they are demanded during program execution
- Pages which are not accessed don't get loaded into physical memory
- Initially, a process resides in secondary memory (hard disk)

Demand Paging

- Need hardware support to distinguish between pages in memory and pages on the disk
- The **valid-invalid bit scheme** (as described in the Memory Management Chapter) can be used for this purpose:
  - When this bit is set to valid, the associated page is both legal and in memory.
  - When invalid, the associated page either is not valid or is valid but is currently not on the disk

Valid-Invalid Bit

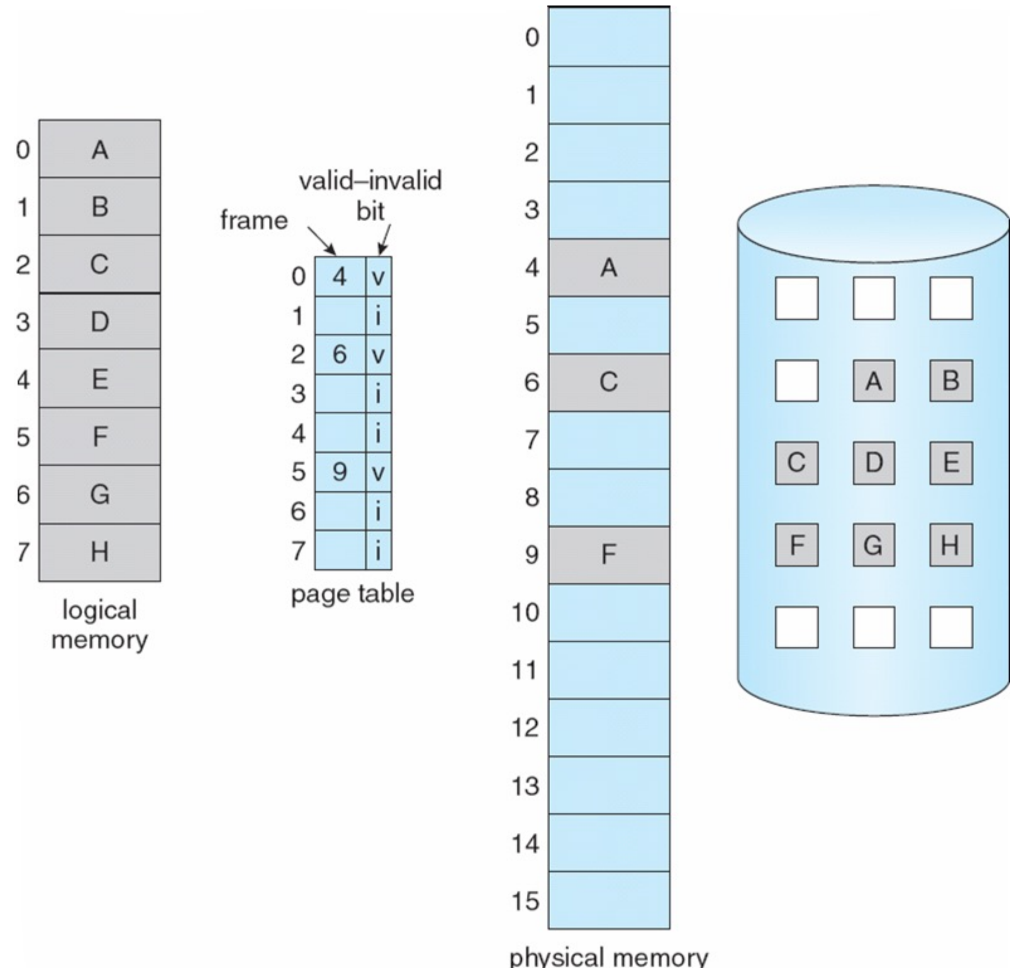
- With each page table entry, a valid–invalid bit is associated (**v** ⇒ in-memory – **memory resident**, **i** ⇒ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:



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

Page Table, some Pages not in Main Memory

- After running for some time, pages have been loaded in memory, so page table has some valid-invalid bits set to v

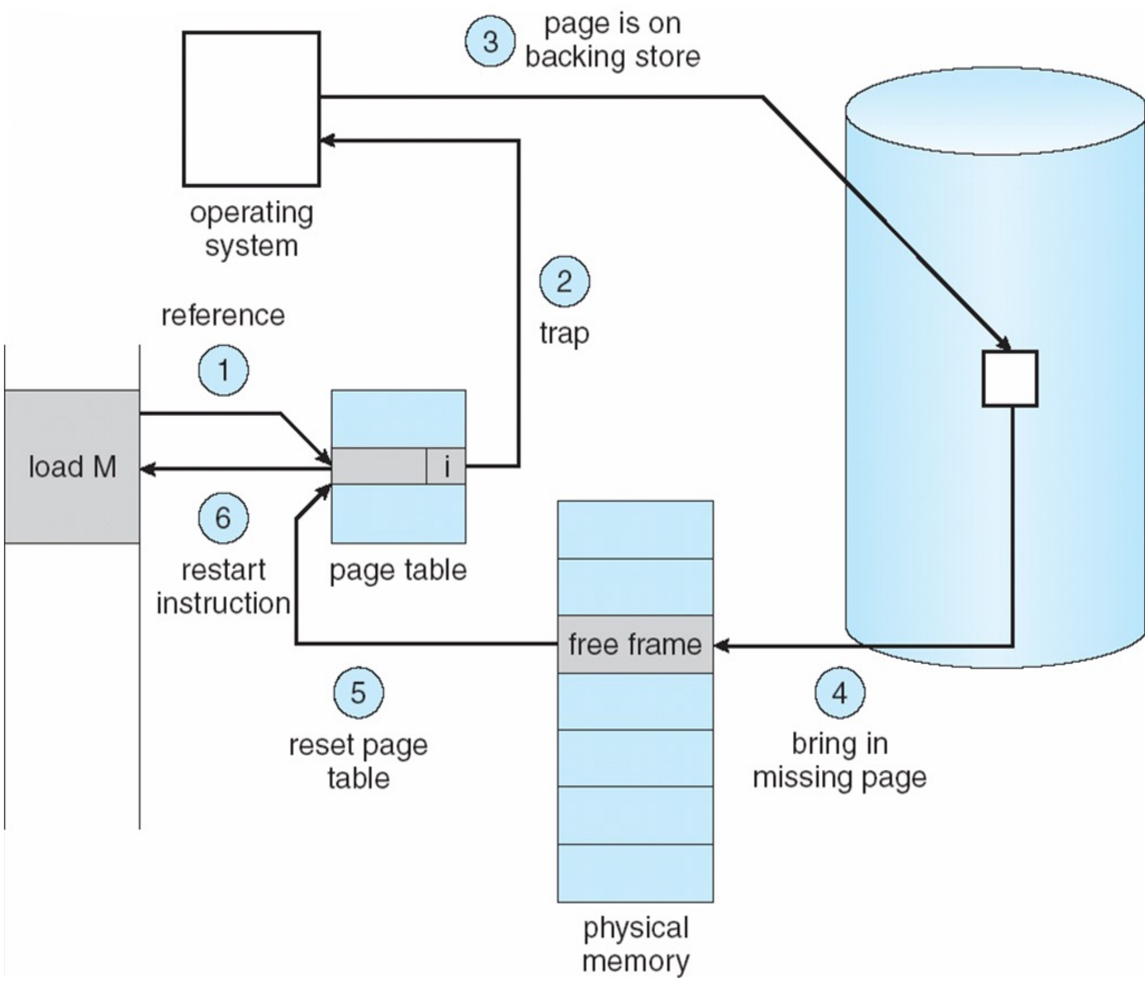


Steps in Handling Page Fault

1. If there is a reference to a page, first reference to that page will be trapped by the operating system
   - Page fault
2. Operating system looks at another table to decide:
   - Invalid reference $\Rightarrow$ abort
   - Just not in memory
3. Find free frame
4. Swap page into frame via scheduled disk operation
5. Reset tables to indicate page now in memory
   Set validation bit = **v**
6. Restart the instruction that caused the page fault

Steps in Handling a Page Fault

1. A reference is made to an entry in the page table that has the invalid bit set (therefore, no frame number in the corresponding table entry).
   Page fault
2. The interrupt handler determine whether the page fault is an invalid address, if so, terminate process
3. Otherwise, page is on disk (back store)
4. Find a free frame and swap in the frame the page from disk
5. Reset tables to indicate page now in memory
   Set validation bit = **v**
6. Restart the instruction that caused the page fault



Pure demand paging

- Pure demand paging: A process is started with no pages loaded into main memory
- Only the page table is in main memory initialized with invalid bits
- First instruction causes a page fault

Stages in Demand Paging – Worse Case

1. Trap to the operating system
2. Save the process registers and process state in the PCB
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:
   a) Wait in a queue for this device until the read request is serviced
   b) Wait for the device seek and/or latency time
   c) Begin the transfer of the page to a free frame

Stages in Demand Paging (Cont.)

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

Overhead 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)

  EAT = $(1 - p)$ x memory access

  $+ p$ (page fault overhead)

Overhead of demand paging: 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

  p = .001

  0.001 x 7,999,800 = 7999.8ns

  EAT = 7999.8 + 200 = 8199.8ns

  8199.8 / 200 = 40.999

  This is a slowdown by a factor of more than 40!!

Overhead of demand paging: Example

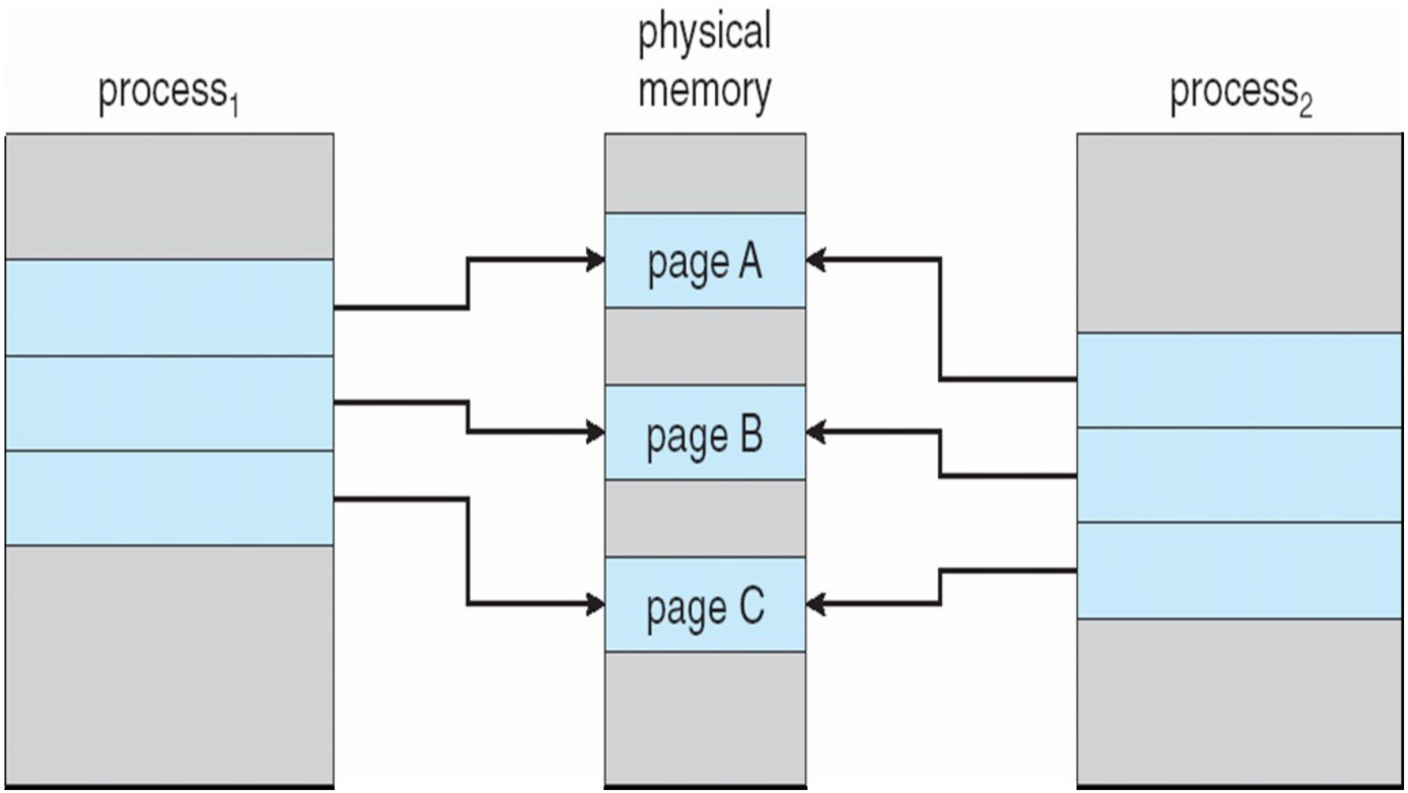
- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- If we want performance degradation < 10 percent, i.e. want EAT < 220ns:
  - $220 > 200 + 7{,}999{,}800 \times p$
    $20 > 7{,}999{,}800 \times p$
  - $p < .0000025$
- The probability p of a page fault should be smaller than 0.0000025, i.e. < one page fault in every 400,000 memory accesses

Process Creation: Copy-on-Write

- Virtual memory allows benefits during process creation:
  - Copy-on-Write
- `fork()` creates a copy of parent's address space for the child
  - Duplicating the pages belonging to the parent.
- Many child processes invoke the `exec()` system call immediately after creation.
  - The copying of the parent's address space may be unnecessary.

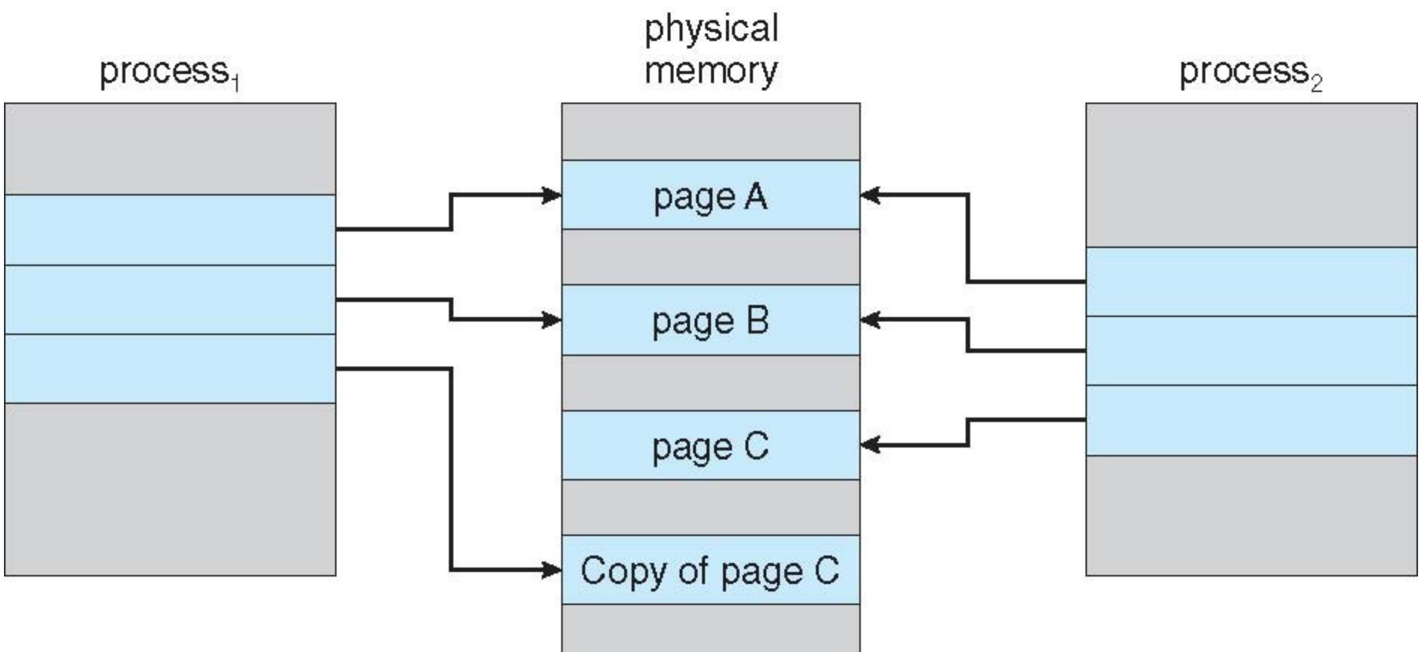
Copy-on-Write

- The page table of the child process is a copy of the parent's page table
- Allows both parent and child processes to initially *share* the same frames in main memory



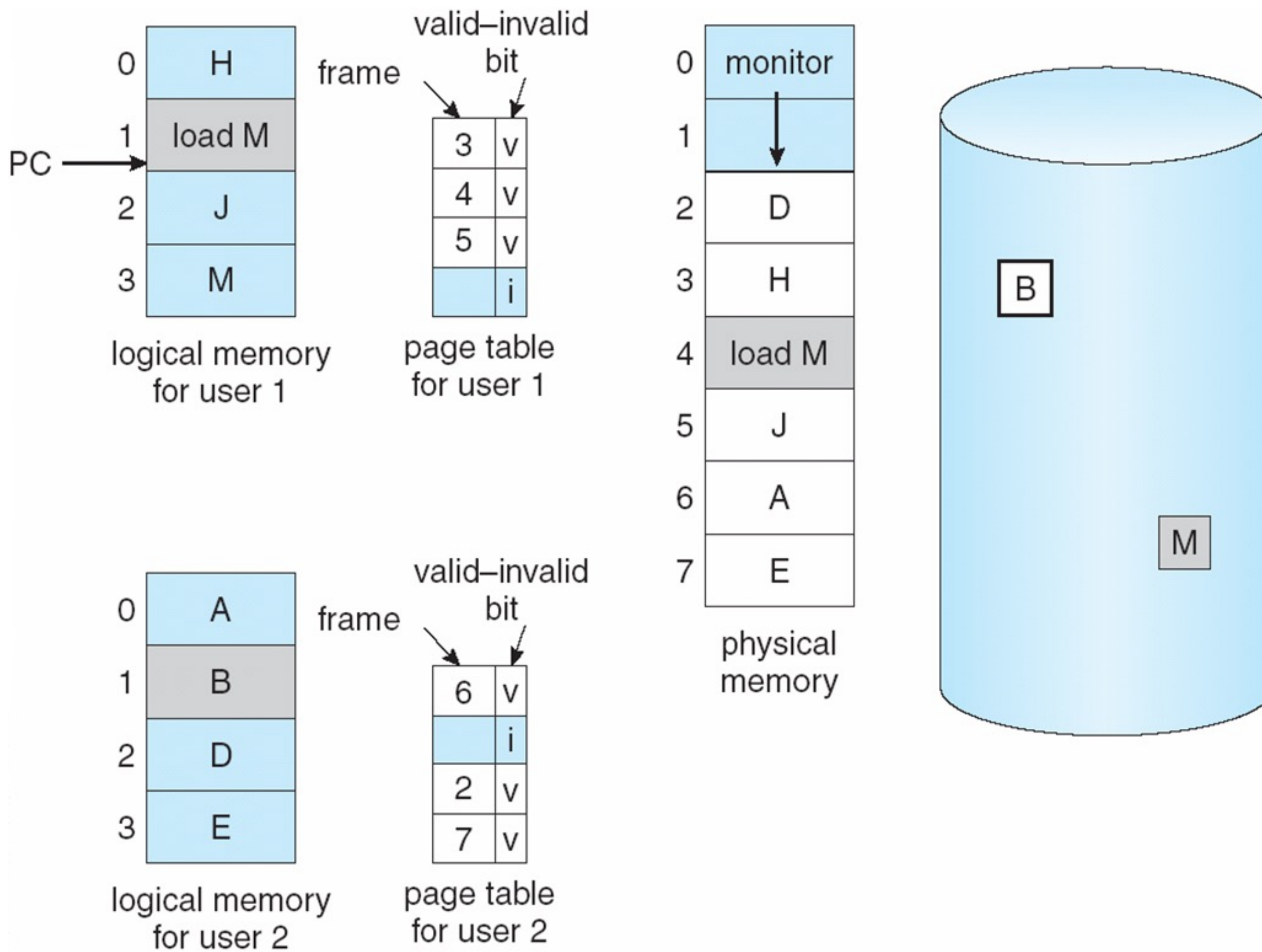
Copy-on-Write

- If either process modifies a shared frame (marked as copy-on-write pages), the child page is copied in a new frame, and the child's page table is updated
- Faster process creation, only modified pages are copied
- Minimize the number of allocated pages.
- Copy-on-Write used by Windows, Linux, and macOS.



What happens if there is no free frame?

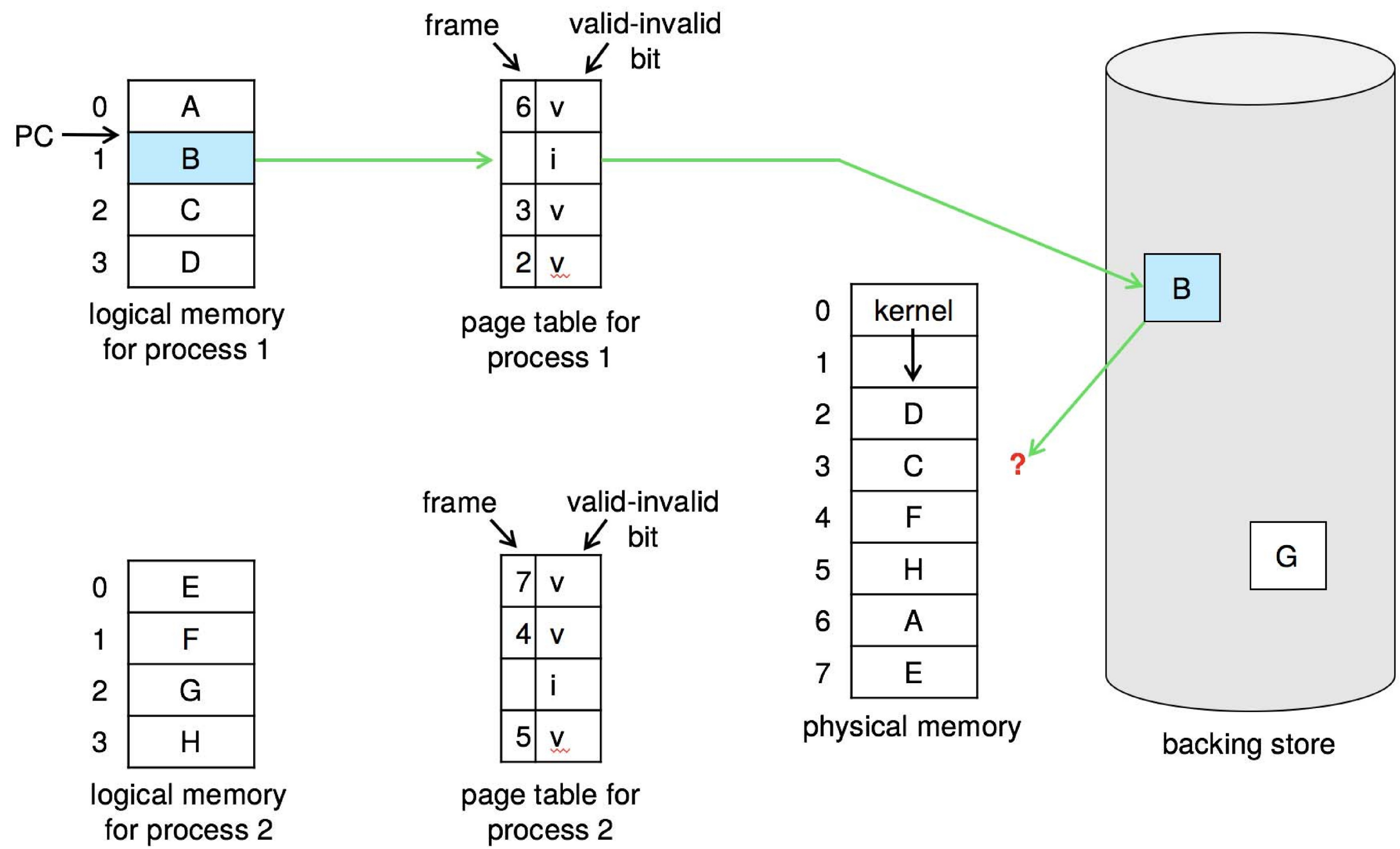
- In order to load a page in main memory, there must be some free frames.



What happens if there is no free frame?

- Need a page replacement strategy
- Page replacement – find some page in memory, but not really in use, swap it out
- Issues:
  - algorithm
  - performance – want an algorithm which will result in minimum number of page faults

Need For Page Replacement

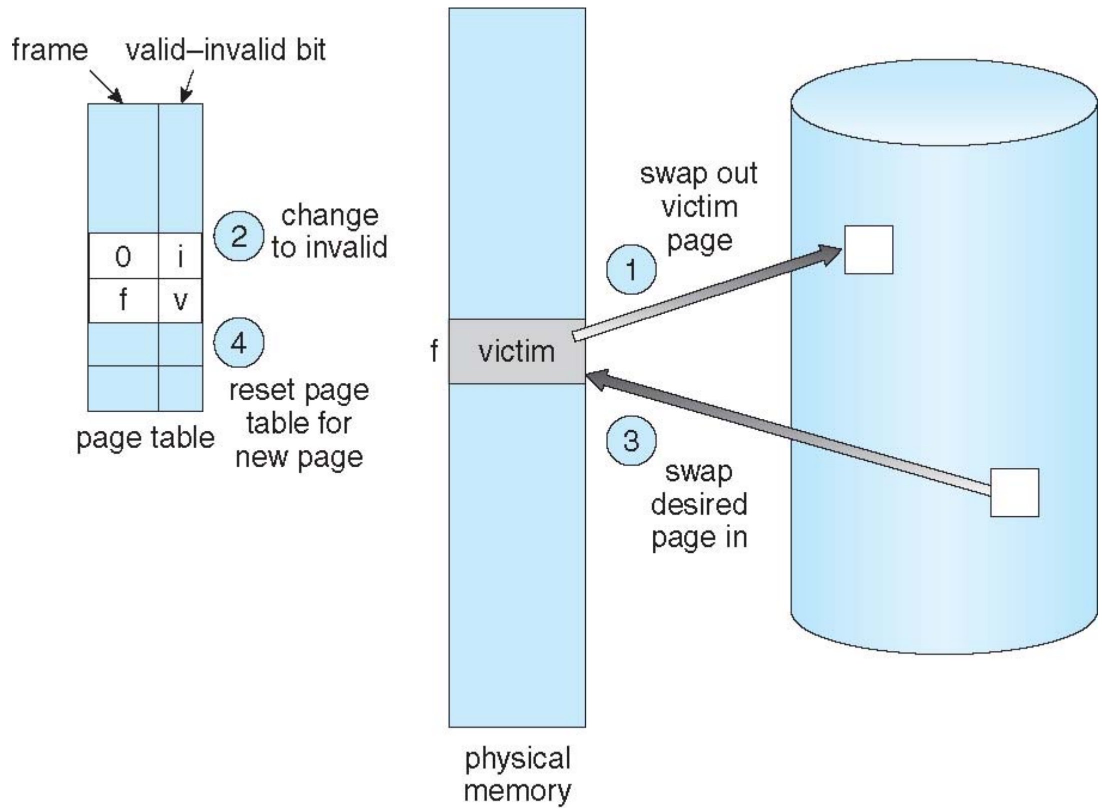


Basic Page Replacement

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
3. Bring the desired page into the (newly) free frame; update the page and frame tables
4. Restart the process

Page Replacement

1. Find the location of the desired page on disk
2. If there is no free frame, use a page replacement algorithm to select a **victim** frame
3. Bring the desired page into the (newly) free frame; update the page and frame tables
4. Restart the process



Page Replacement

- Two-page transfer (one out and one in) needed
  - **Double the page-fault service time!**
- We can reduce the overhead by using a ***modify bit*** (dirty bit)
  - The modify bit for a page is set whenever any word or byte in the page is written into, since it was read in from the disk
- When we select a page for replacement, we examine its modify bit.
  - If set, we must write that page to the disk.
  - If not set, we need not write the page to the disk.
- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory

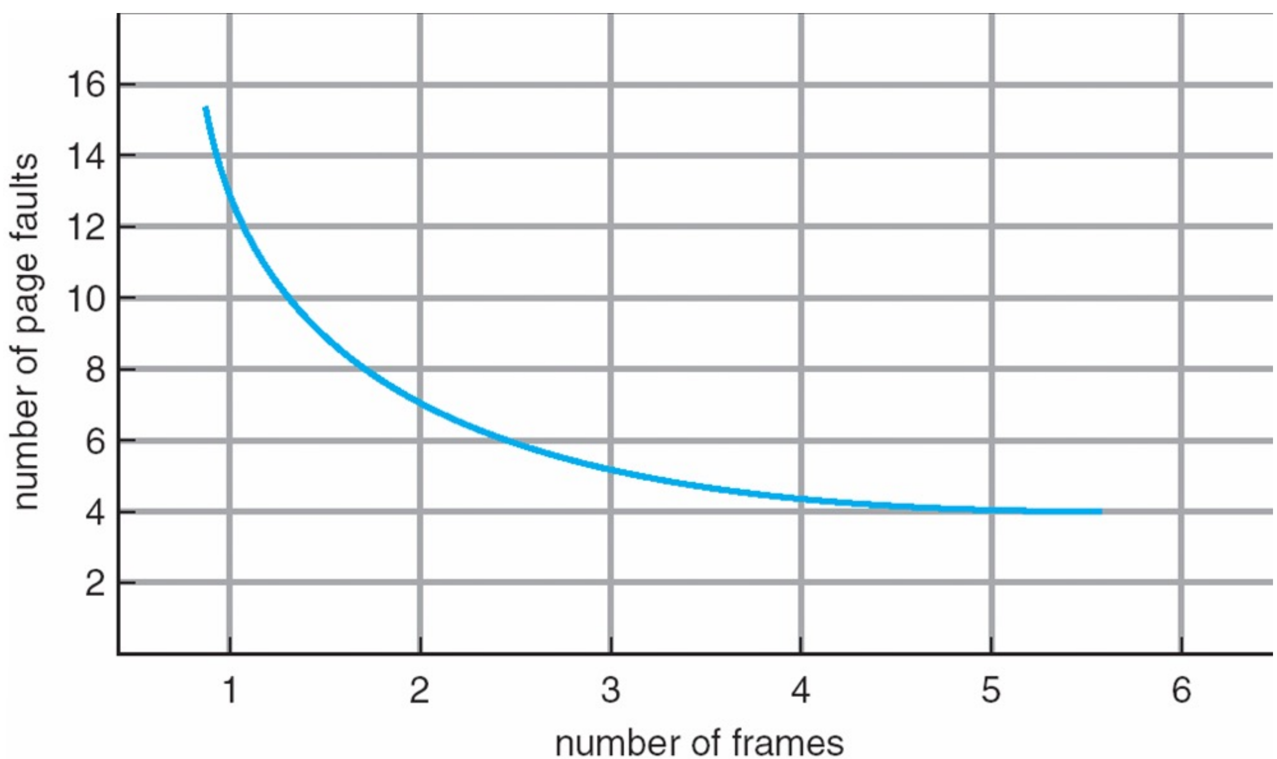
Page and Frame Replacement Algorithms

- **Frame-allocation algorithm** determines
  - How many frames to give each process
  - Which frames to replace
- **Page-replacement algorithm**
  - Want lowest page-fault rate on both first access and re-access
- Evaluate page replacement algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
  - String is just page numbers, not full addresses
  - Repeated access to the same page does not cause a page fault
  - Results depend on number of frames available
- In all our examples, the **reference string** of referenced page numbers is

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

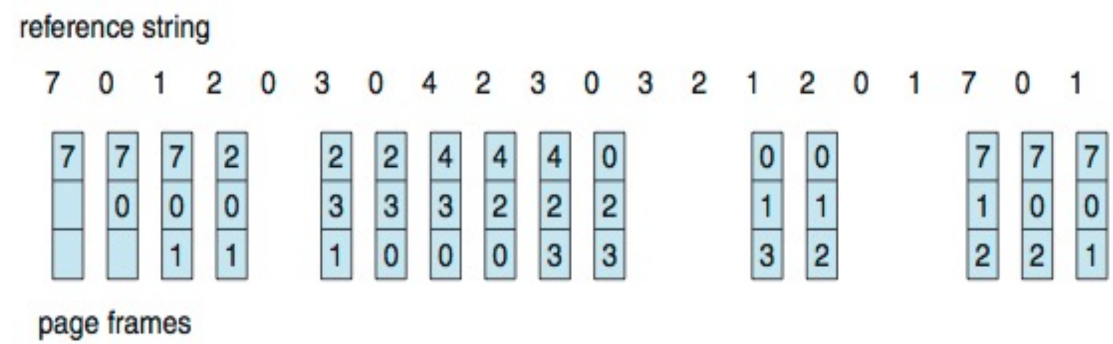
Graph of Page Faults Versus the Number of Frames

- To calculate the number of page faults, we need to know the number of frames available
- Ideally, we expect as the number of frames available increases, the number of page faults decreases



First-In-First-Out (FIFO) Algorithm

- FIFO replacement algorithm: Select the oldest page for replacement
- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)

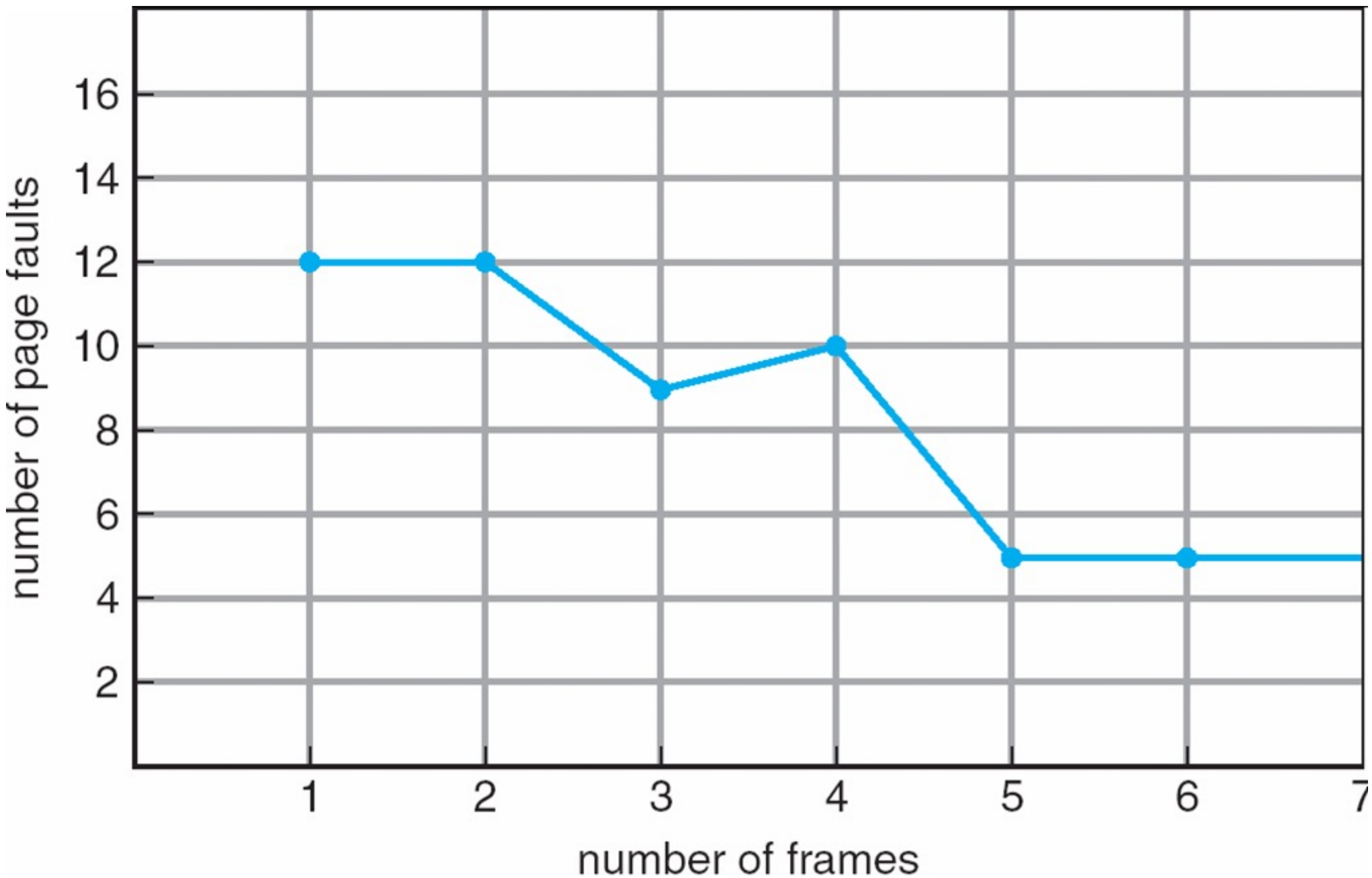


15 page faults

- How to track ages of pages?
  - Just use a FIFO queue

FIFO & Belady's Anomaly

- Page-fault rate may increase as the number of allocated frames increases



Example Belady's Anomaly for FIFO

- 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)

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

9 page faults

- 4 frames

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

10 page faults

Optimal Algorithm

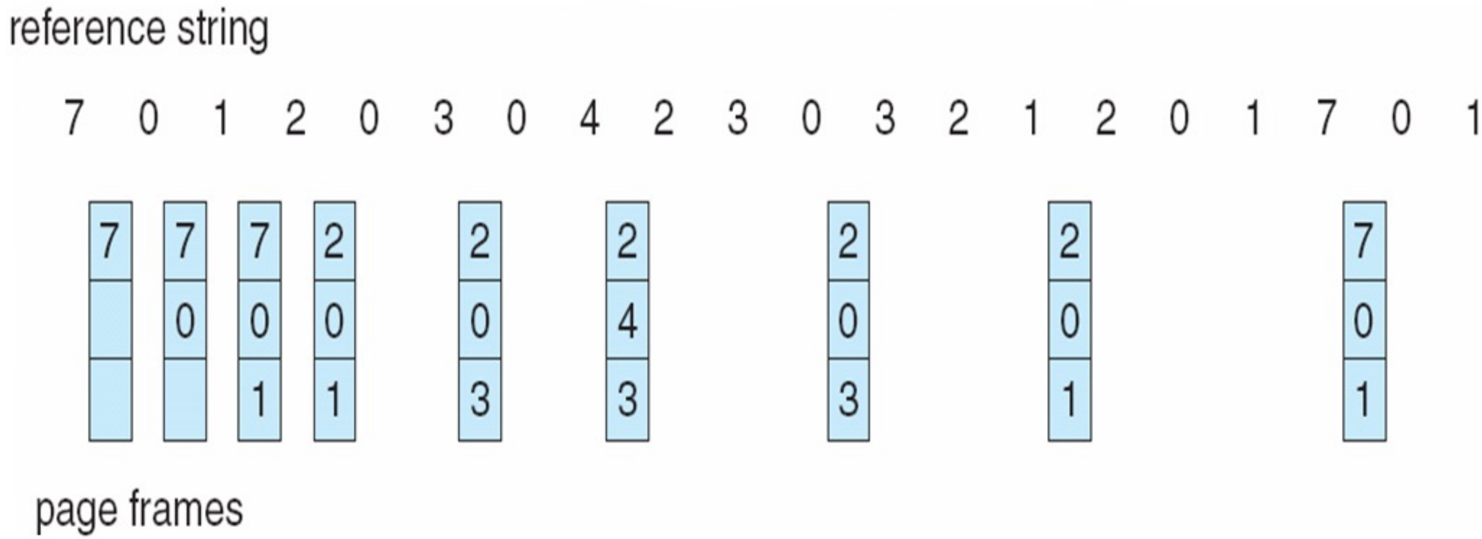
- Replace page that will not be used for longest period of time
- 4 frames example

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

1	4
2	
3	
4	5

6 page faults

Optimal Page Replacement



- For the above example, 9 page faults is optimal
- But optimal page-replacement algorithm is impossible to implement. Requires future knowledge of the reference string.
- Algorithm used mainly for comparison studies.

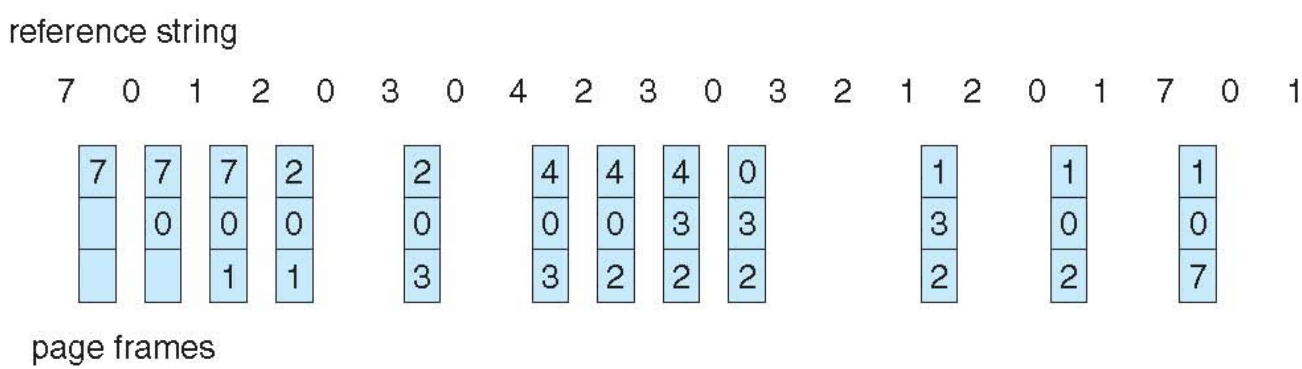
Least Recently Used (LRU) Algorithm

- Replace the page that **has not been used for the longest period of time**
- LRU seen as an approximation of OPT Algo
- Reference string: 1, 2, 3, 4, 1, 2, **5**, 1, 2, **3**, **4**, **5**

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

Least Recently Used (LRU) Algorithm

- Use past knowledge rather than future
- Replace page that has not been used in the most amount of time
- Associate time of last use with each page



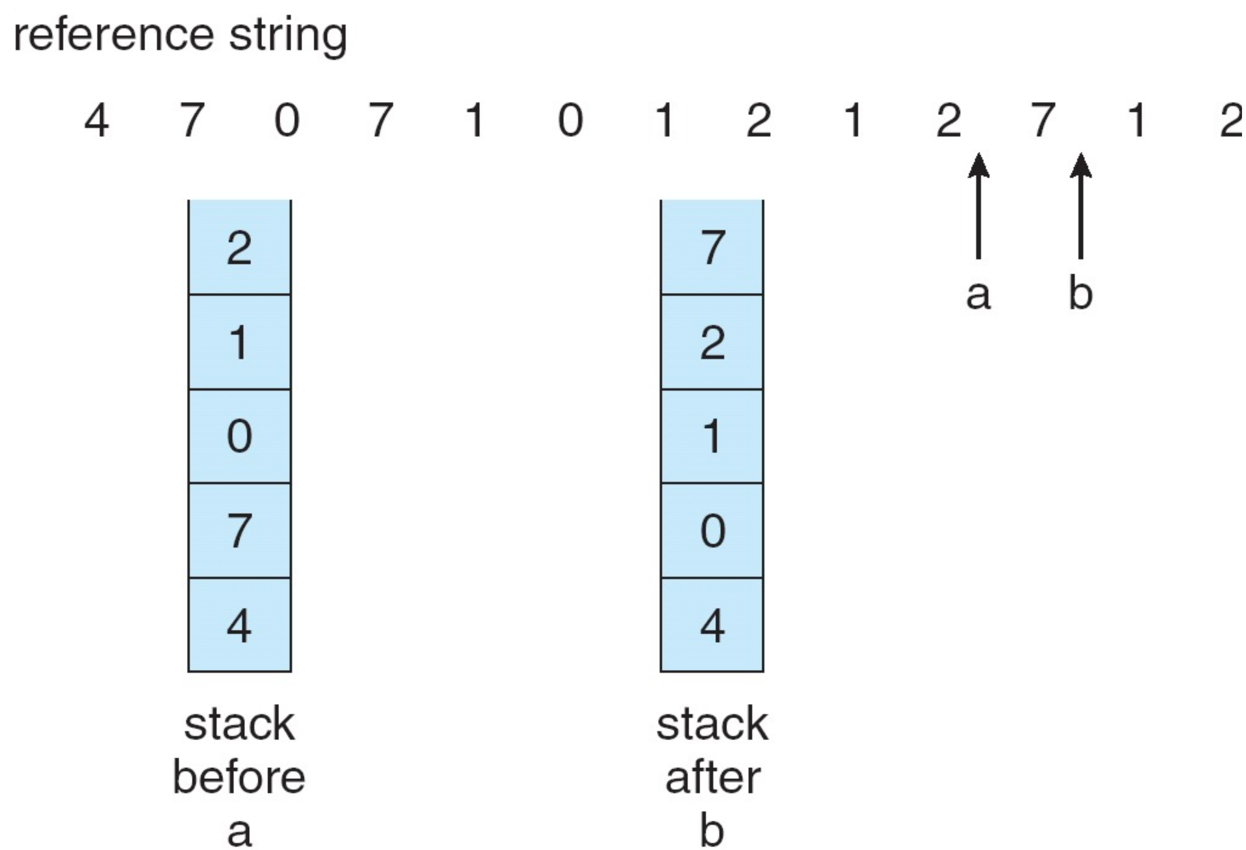
- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- But how to implement?

LRU Algorithm (Cont.)

- 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 find smallest value
    - Search through table needed
- Stack implementation
  - Keep a stack of page numbers in a double link form:
    - Page referenced:
      - move it to the top
      - requires 6 pointers to be changed
  - Each update more expensive
  - No search for replacement

LRU: Stack Implementation

- Keep a stack of page numbers in a doubly link list:
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed
  - No search for replacement



- LRU and OPT are cases of **stack algorithms** that don't have Belady's Anomaly

Allocation of Frames

- Must decide how many frames to allocate for each process
- Each process needs *minimum* number of pages
  - To reduce page-fault rate
  - To resolve a single instruction
    - number of frames that a single instruction can reference
    - this minimum **defined by the computer architecture**
- For example – a system in which all memory-reference instructions have only one memory address.
  - We need one frame for the instruction and one frame for the memory reference.
- Two major allocation schemes
  - fixed allocation
  - priority allocation

Fixed Allocation

- Equal allocation – For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames
- 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 $p_i = \frac{s_i}{S} \times m$

$$m = 64$$
$$s_1 = 10$$
$$s_2 = 127$$
$$a_1 = \frac{10}{137} \times 62 \approx 4$$
$$a_2 = \frac{127}{137} \times 62 \approx 57$$

Global vs. Local Allocation

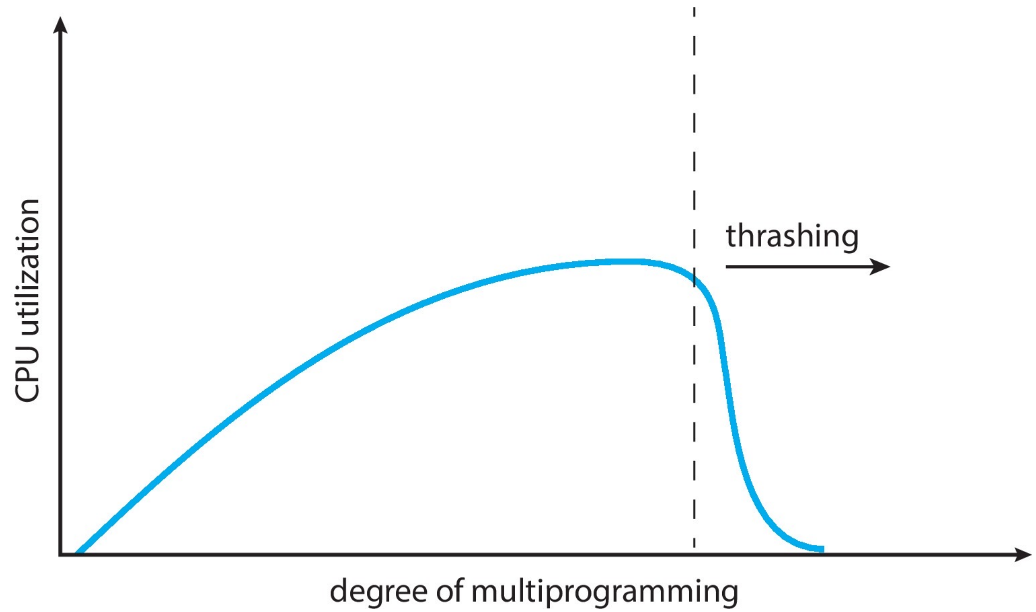
- Frame can be allocated through page replacement
- Page replacement algorithms can be classified in two categories:
  - Local replacement
  - Global replacement
- **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another
  - For a high priority process, replacement can select a victim from a low priority process
  - Number of frames allocated to a process change, so process execution time can vary greatly
  - But greater throughput so more common
- **Local replacement** – each process selects from only its own set of allocated frames
  - More consistent per-process performance
  - But possibly underutilized memory

Thrashing

- **Thrashing** ≡ a process that does not have enough frames, when it page-fault, it replaces page that it needs right away.
- When trashing, processes are busy swapping pages in and out, spending more time paging than executing
- If a process does not have “enough” frames, the page-fault rate is very high. This leads to:
  - low CPU utilization
  - OS thinks that it needs to increase the degree of multiprogramming
  - another process added to the system, stealing frames from the process that is trashing
  - resulting in even more trashing and even lower CPU utilization

Thrashing (Cont.)

- **Thrashing**. A process is busy swapping pages in and out

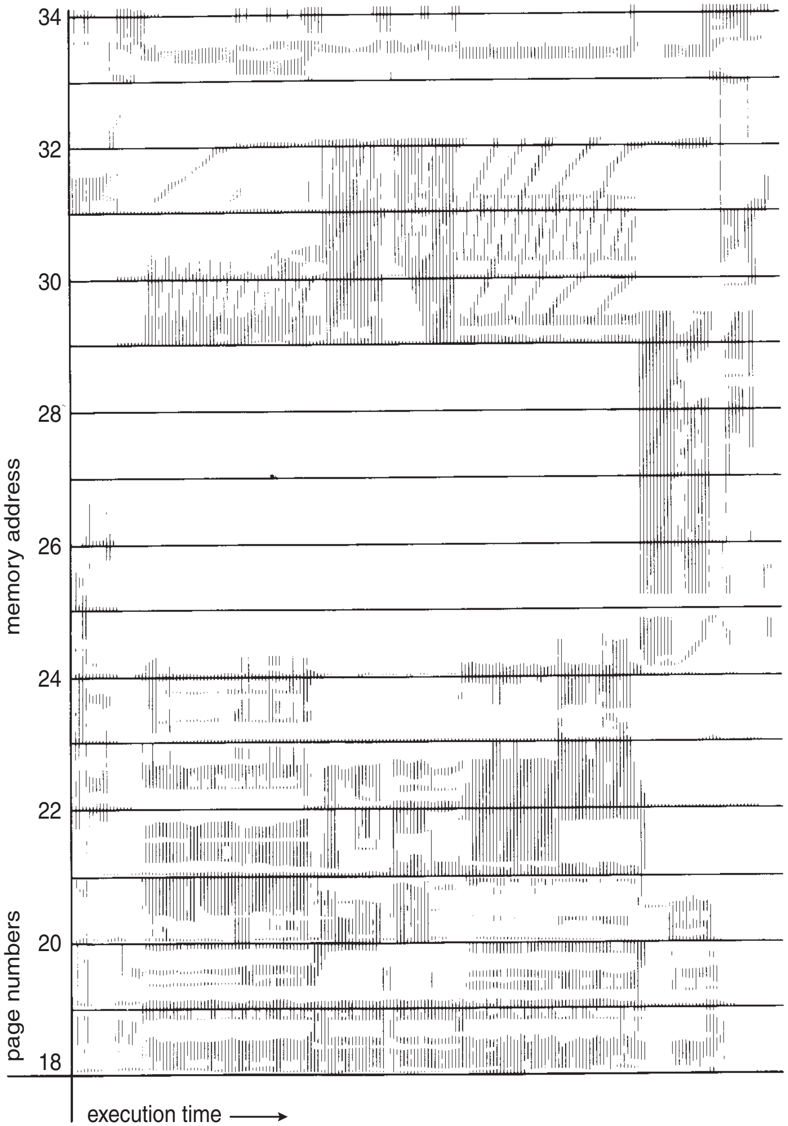


Thrashing

- We can limit the effects of thrashing using local replacement
  - If one process starts thrashing, cannot steal frames from another process
  - However, the queue of the paging device will contain a lot of requests
  - The service time for a page fault of a non-thrashing process will also increase
- To prevent thrashing, must provide a process with as many frames as it needs.
- We don't know how many frames a process needs
- Process's memory references generally exhibit ***locality*** of patterns, or ***locality model*** of execution
- **Locality:** a set of pages actively used together
  - For example, the locality of a function call consists of the memory references of the function instructions, its local variables, and a subset of the global variables.

The **locality model** states that, as a process executes, it moves from locality to locality

Locality In A Memory-Reference Pattern



Working-Set Strategy

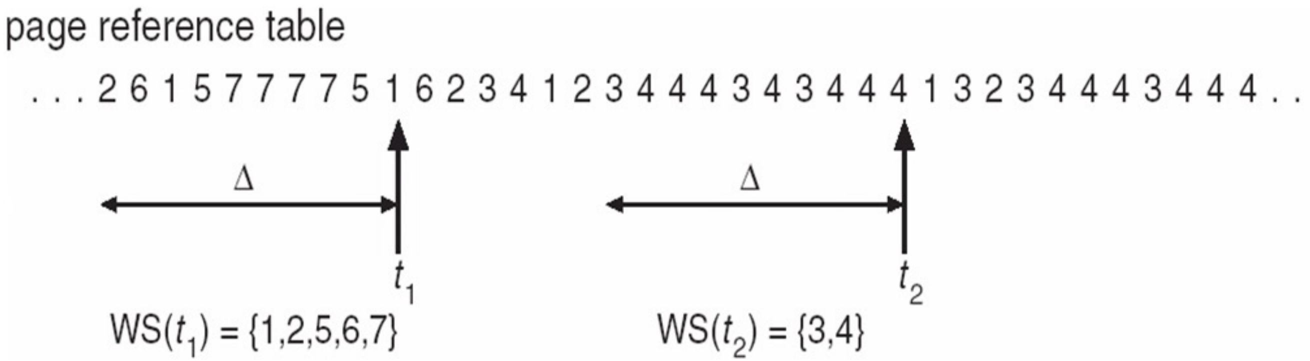
- Why does thrashing occur?
  - We allocate fewer frames than the size of the current locality process
- Based on the locality model, the ***working-set strategy*** tries to identify *how many frames a process is actually using*
- The method uses a parameter, Δ, to define the ***working-set window***
  - The idea is to examine **the most recent page references**.
  - The set of page in the most recent Δ page references is the ***working set***

Working-Set Model

- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  Example: 10,000 instructions
- WSS_i (working set of Process P_i) = total number of pages referenced in the most recent Δ (varies in time)
  - if Δ too small will not encompass entire locality
  - if Δ too large will encompass several localities
  - if $\Delta = \infty \Rightarrow$ will encompass entire program
- $D = \Sigma\ WSS_i \equiv$ total demand frames
  - Approximation of locality

Working-Set Model (Cont.)

- if $D > m \Rightarrow$ Thrashing (where m is the total number of frames)
- Policy if $D > m$, then suspend or swap out one of the processes

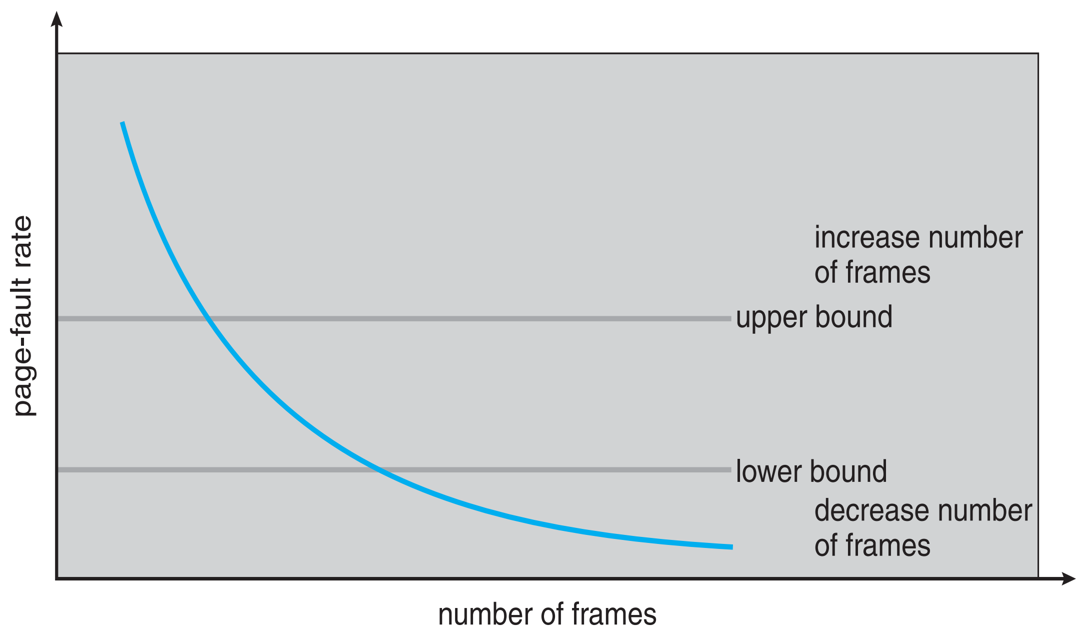


Keeping Track of the Working Set

- Approximate with interval timer + a reference bit
- Example: $\Delta = 10{,}000$
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 bits for each page
  - Whenever a timer interrupts copy and sets the values of all reference bits to 0
  - If one of the bits in memory = 1 $\Rightarrow$ page in working set
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units

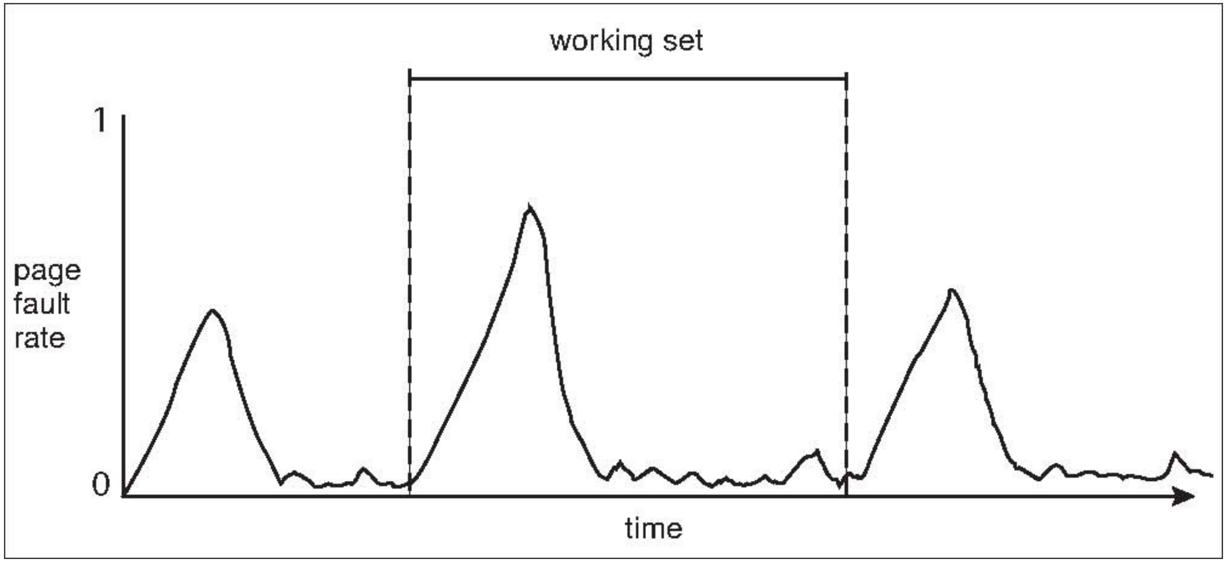
Page-Fault Frequency

- More direct approach than WSS
- Establish "acceptable" **page-fault frequency** (**PFF**) rate and use local replacement policy
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame



Working Sets and Page Fault Rates

- Direct relationship between working set of a process and its page-fault rate
- Working set changes over time
- Peaks and valleys over time



Operating System Examples

- Windows

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

End of Section 10