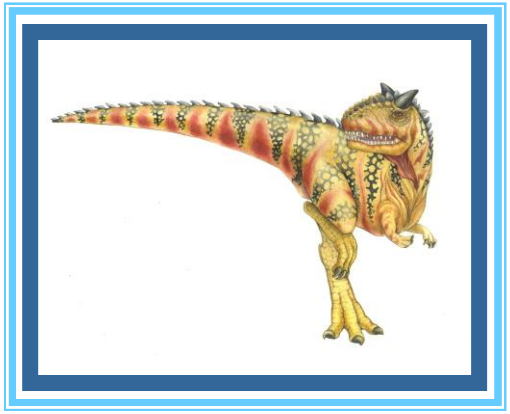
Chapter 10: Virtual Memory

Section 10.1-10.6

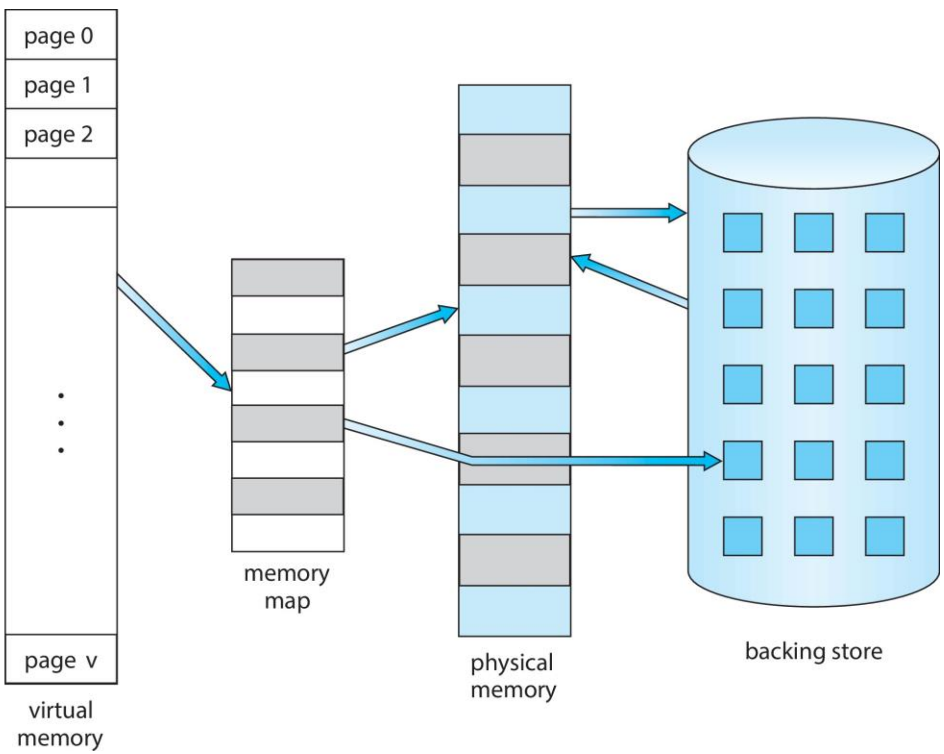


Background

- Code needs to be in memory to execute, but entire program rarely used
  - Code to handle errors, routines rarely used, large data structures
- Even when the entire program is needed, all program code may not be needed at same time
- Consider ability to execute partially-loaded program
  - Program no longer constrained by limits of physical memory
  - Each program takes less memory while running → more programs could be run at the same time
    - Increased CPU utilization and throughput
  - Less I/O needed to load or swap portions of programs into memory → each program runs faster

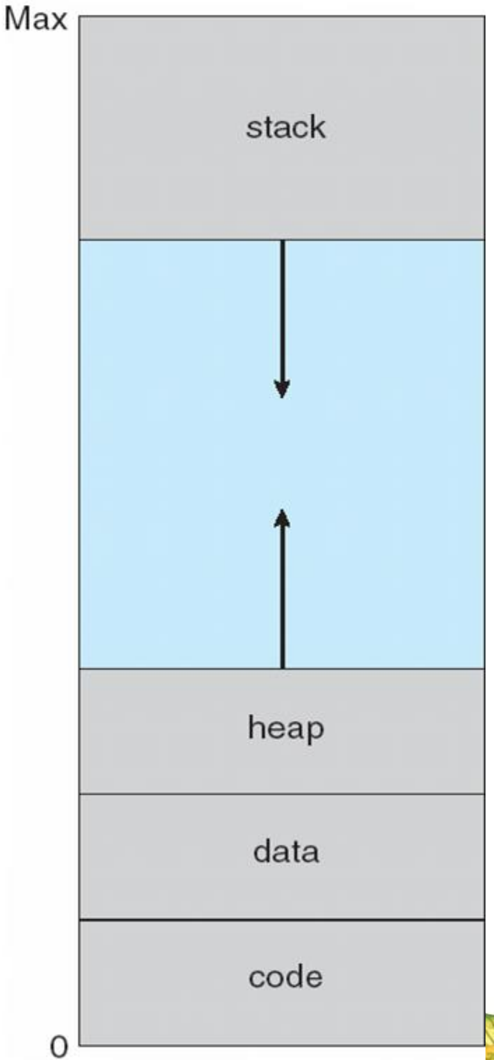
Virtual memory

- **Virtual memory** involves the separation of logical memory as perceived by developers from physical memory
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space



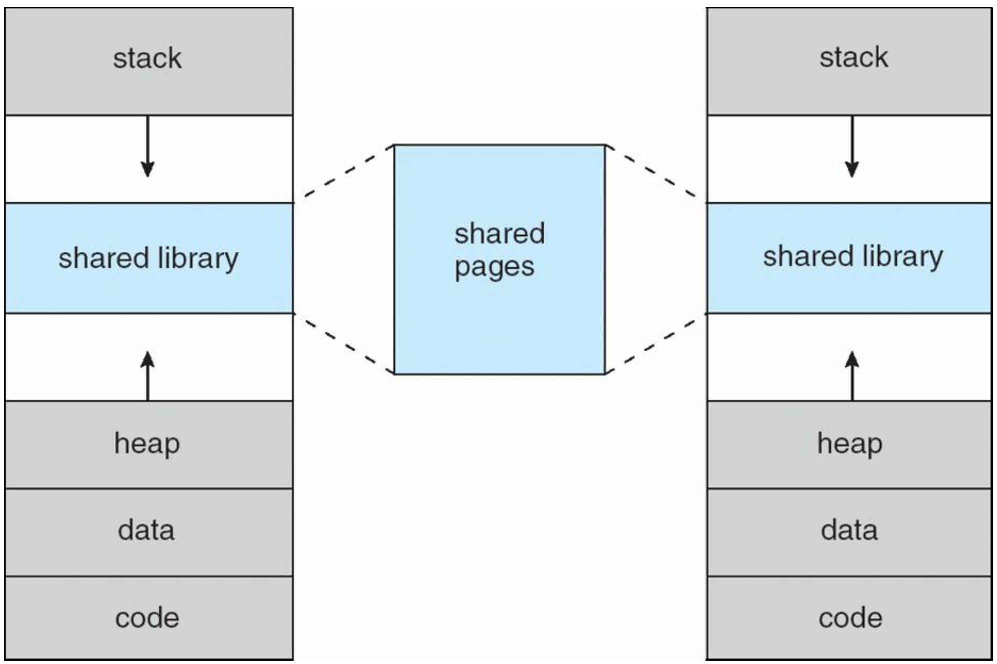
Virtual Address Space

- **Virtual address space** – logical view of how process is stored in memory
  - Usually start at address 0, contiguous addresses until end of address space
  - Meanwhile, physical memory organized in page frames
  - MMU must map logical addresses to physical addresses
- The hole between stack and heap is part of virtual address space
  - No physical memory needed until heap or stack grows to a new page



Shared Library Using Virtual Memory

- Processes can share system libraries by mapping shared library into virtual address space
  - The pages where the libraries reside in physical memory are shared by processes
- Processes can share memory by mapping shared physical pages into virtual address space

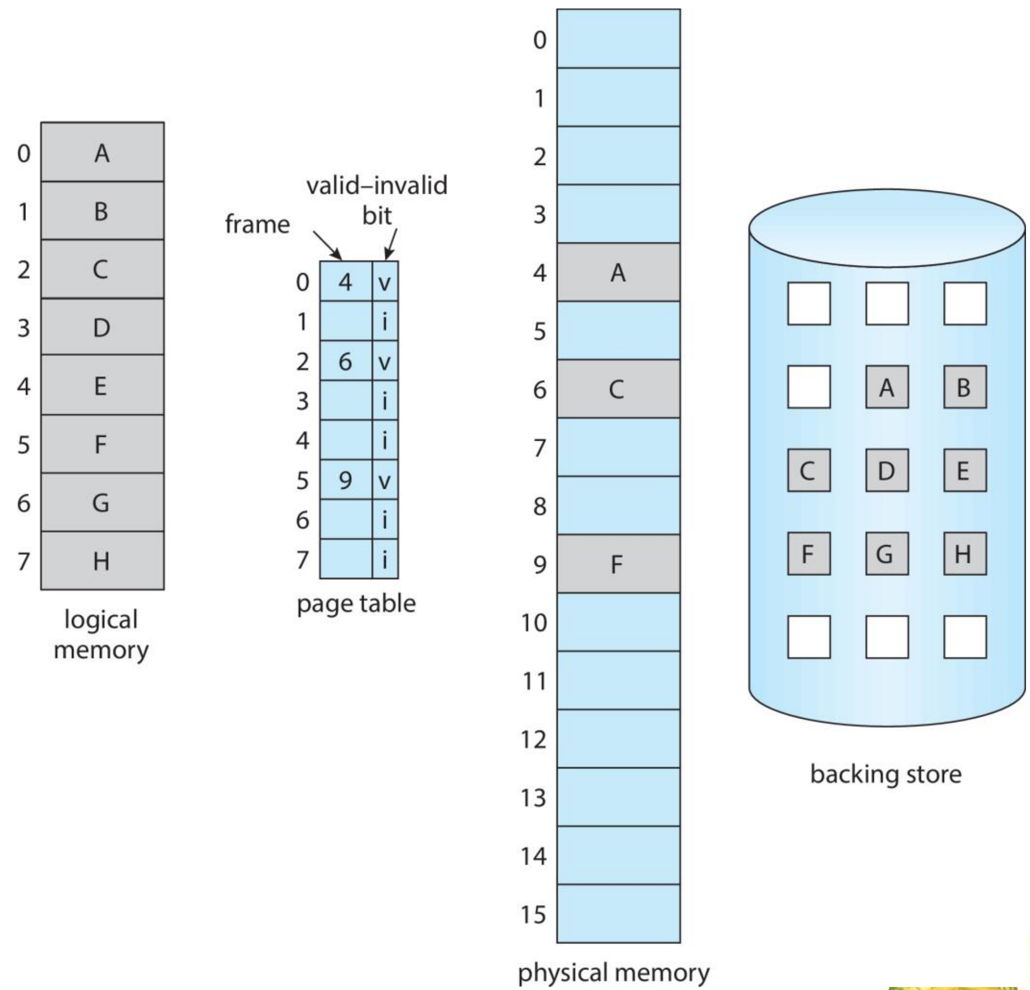


Demand Paging

- **Demand paging –** bring a page into memory only when it is needed
  - No unnecessary I/O
  - Less memory needed
- When a page is referenced
  - invalid reference (i.e., the page is not in the process's logical address space) $\Rightarrow$ abort
    - For example: 14-bit logical address space, 2KB page size → a process can have at most 8 pages
    - If a process has only 6 pages, reference to page 6 or 7 would be invalid
  - not-in-memory $\Rightarrow$ bring to memory

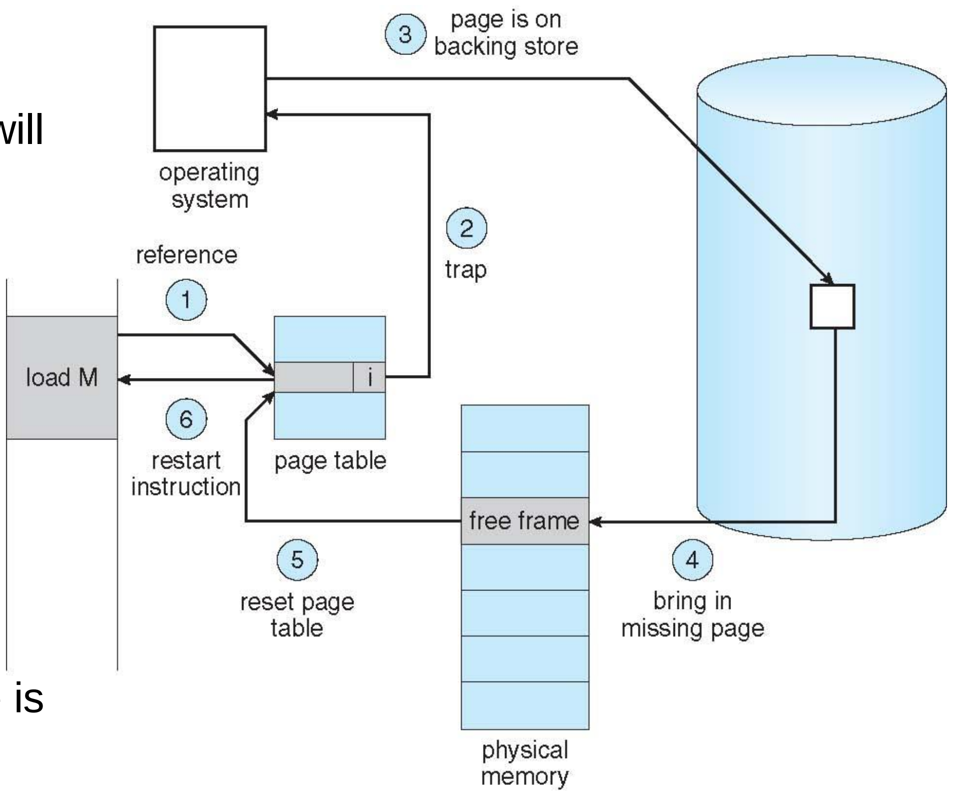
Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated
  - **v** ⇒ page is legal and in-memory
  - **i** ⇒ page is illegal or legal but not-in-memory
- Initially valid–invalid bit is set to **i** on all entries
- Access to a page marked invalid causes a **page fault**



Steps in Handling Page Fault

1. During MMU address translation, referencing a page marked invalid will cause a page fault and the hardware will trap to operating system
2. OS checks an internal table (kept with PCB) for the process to decide:
   - Invalid reference ⇒ abort
   - Just not in memory ⇒ bring the page in memory
3. Find a free frame
4. Swap page into frame via scheduled disk operation
5. Modify the page table to indicate page is now in memory: set validation bit = **v**
6. Restart the instruction that caused the page fault



Aspects of Demand Paging

- **Pure demand paging** – start process with *no* pages in memory
  - OS sets instruction pointer to first instruction of process, page is non-memory-resident → page fault
  - And page fault occurs for every page on first access
- A given instruction could access multiple pages → multiple page faults
  - Consider fetch and decode of instruction which adds 2 numbers from memory and stores result back to memory
  - Pain decreased because of **locality of reference**
- Hardware support needed for demand paging
  - Page table with valid/invalid bit
  - Secondary memory (**swap space**) that holds those pages that are not present in main memory
    - Typically high-speed disk or NVM device

Free-Frame List

- When a page fault occurs, the operating system must bring the desired page from secondary storage into main memory
- Most operating systems maintain a **free-frame list** -- a pool of free frames for satisfying such requests

head → 7 → 97 → 15 → 126 ··· → 75

- Operating system typically allocate free frames using a technique known as **zero-fill-on-demand** -- the content of the frames are "zeroed-out" before being allocated
- When a system starts up, all available memory is placed on the free-frame list

Steps in Page Fault Servicing

1. Trap to the operating system
2. Save the 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

Steps in Page Fault Servicing (Cont.)

6. While waiting, allocate the CPU to some other process
7. Receive an interrupt from the disk I/O subsystem (I/O completed)
8. Save the registers and process state for the other process
9. Determine that the interrupt was from the disk
10. Update the page table to show that the desired 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 (1-100 microseconds)
  - Read the page – lots of time (8 milliseconds)
  - Restart the process – again just a small amount of time (1-100 microseconds)

- Page fault rate p, $0 \le p \le 1$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault
- Effective Access Time (EAT)

  EAT = (1 – *p*) x memory access time

  \+ *p* x page-fault service time

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 (p=0.001), then

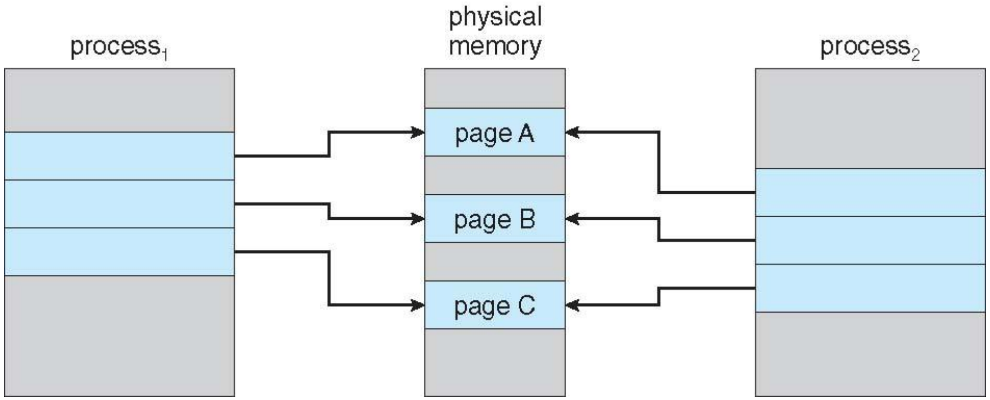
  EAT = 8200 nanoseconds = 8.2 microseconds

  This is a slowdown by a factor of 40!! (since if p=0, EAT=0.2 microseconds)
- If want performance degradation < 10%
  - 220 > 200 + 7,999,800 x p
    20 > 7,999,800 x p
  - p < 0.0000025, i.e., less than one page fault in every 400,000 memory accesses
- It is important to keep the page-fault rate low!

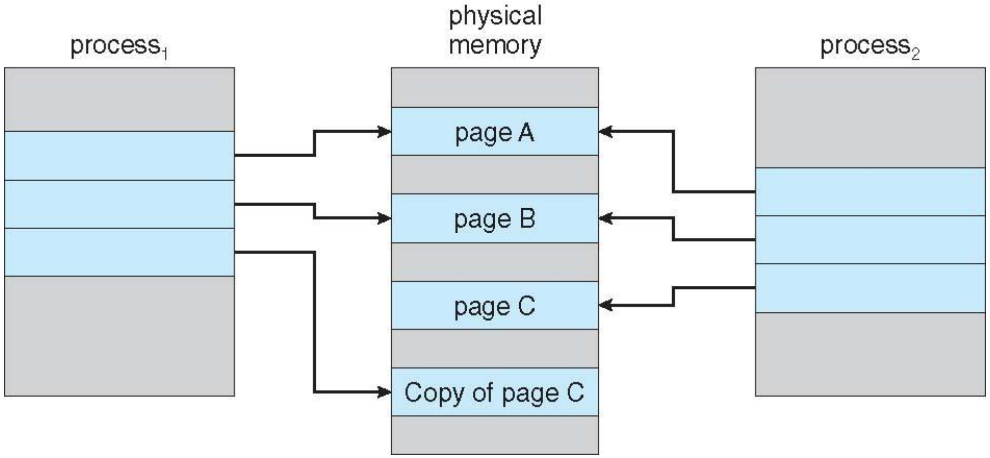
Copy-on-Write

- **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

Before process 1 modifies page C



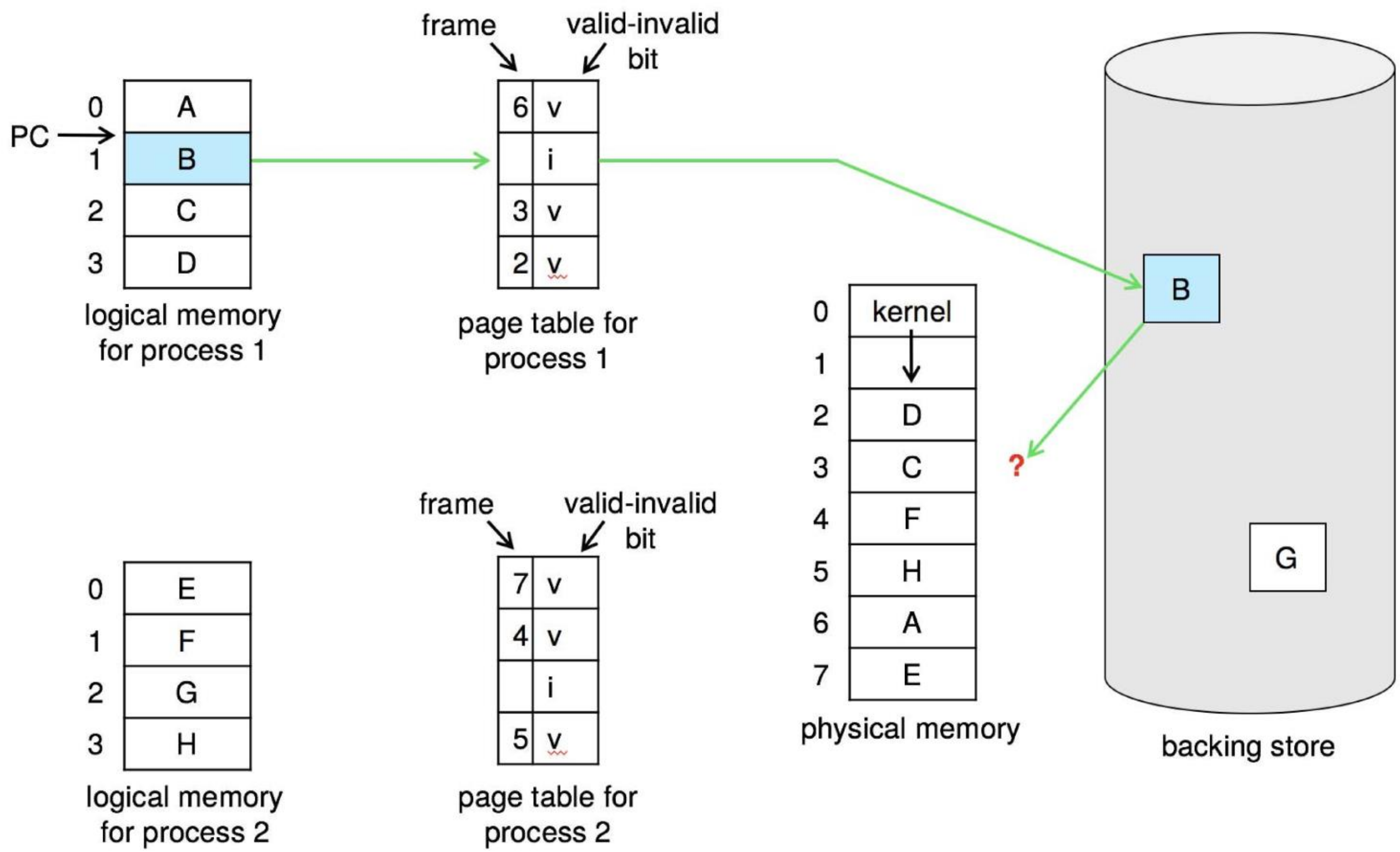
After process 1 modifies page C



Page Replacement

- How many frames should be allocated to each process?
  - Example: 40 frames, each process of size 10 pages
  - If each process uses only 5 pages, then we can run 8 processes
    - We are over-allocating memory: each process may suddenly try to use all 10 pages
- Consequence of over-allocation of memory: when a page fault occurs, OS finds that there is no free frame
- OS options
  - Terminate the process – not good
  - Swap out a process to free all its frames – high overhead
  - Page replacement – find a page in memory that is not in use, swap it out
    - Same page may be brought into memory several times
    - Performance – want an algorithm which will result in minimum number of page faults
    - Used by most operating systems

Need For Page Replacement



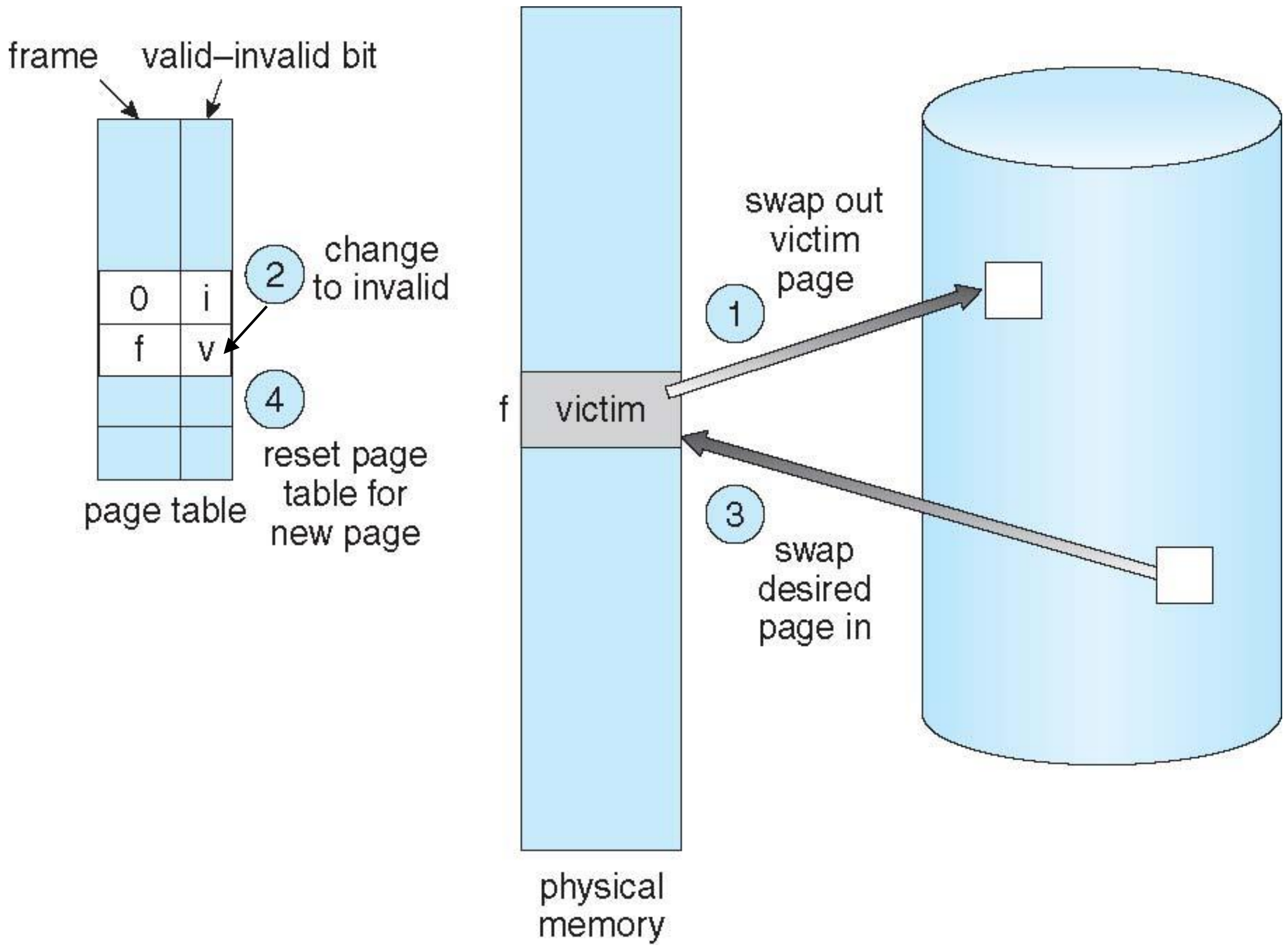
Basic Page Replacement

1. When page fault occurs, 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 page** and write victim page to disk
3. Bring the desired page into the (newly) free frame; update the page table
4. Continue the process by restarting the instruction that caused the page fault

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

- Can use a **modify bit** (or **dirty bit)** to reduce overhead of page transfers
  - Each page has a modified bit associated with it
  - Write victim frame to disk if dirty

Page Replacement

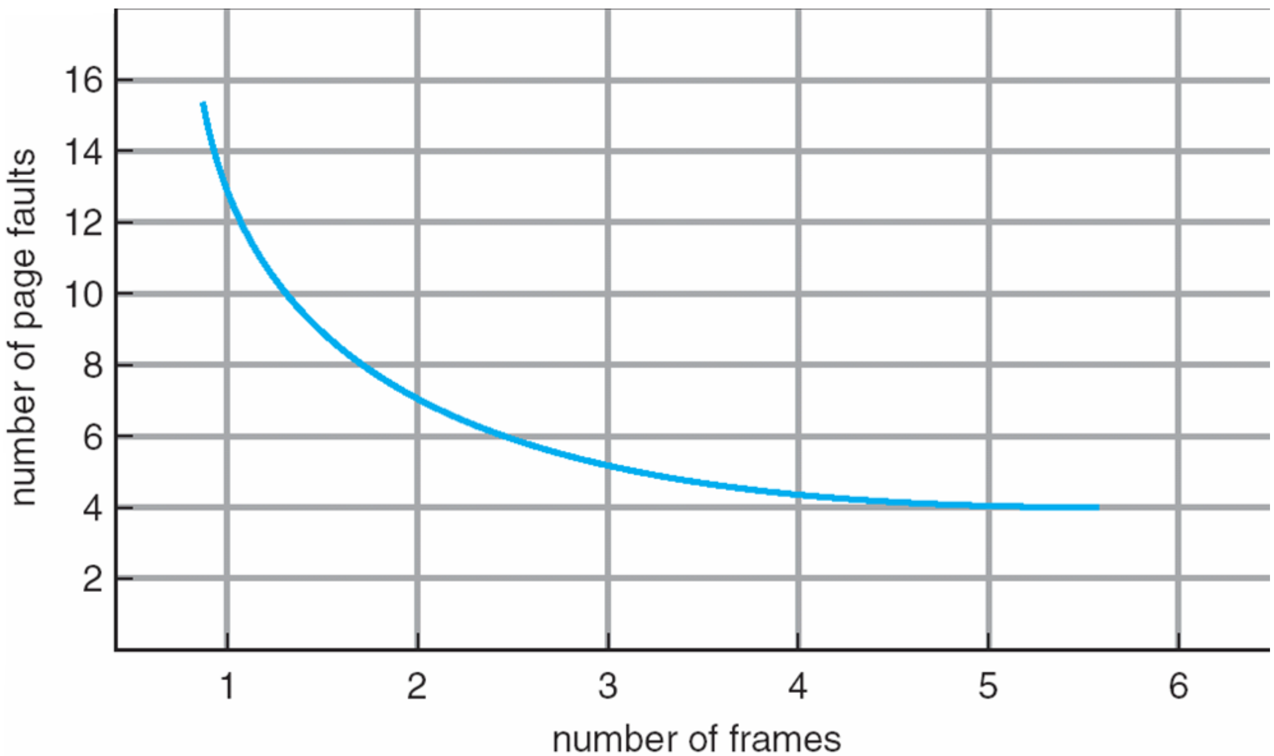


Page Replacement Algorithms

- **Page-replacement algorithm**
  - Select the page that is to be replaced
  - Want lowest page-fault rate
- Evaluate 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
    - E.g., if page size is 100 bytes, then address sequence 0100 0432 0101 0612 0102 0103 0104 0611 0102 is reduced to the reference string 1 4 1 6 1 6 1
  - Number of page faults depends on number of frames available
- In all our examples, the **reference string** is

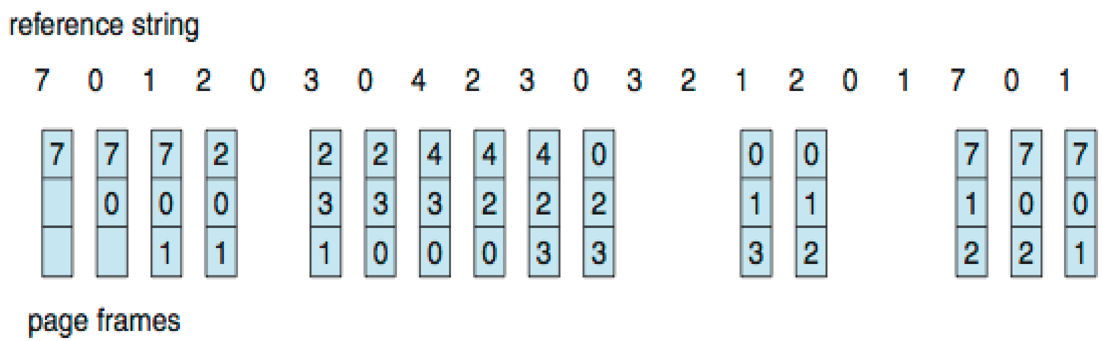
7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

Graph of Page Faults Versus The Number of Frames



First-In-First-Out (FIFO) Algorithm

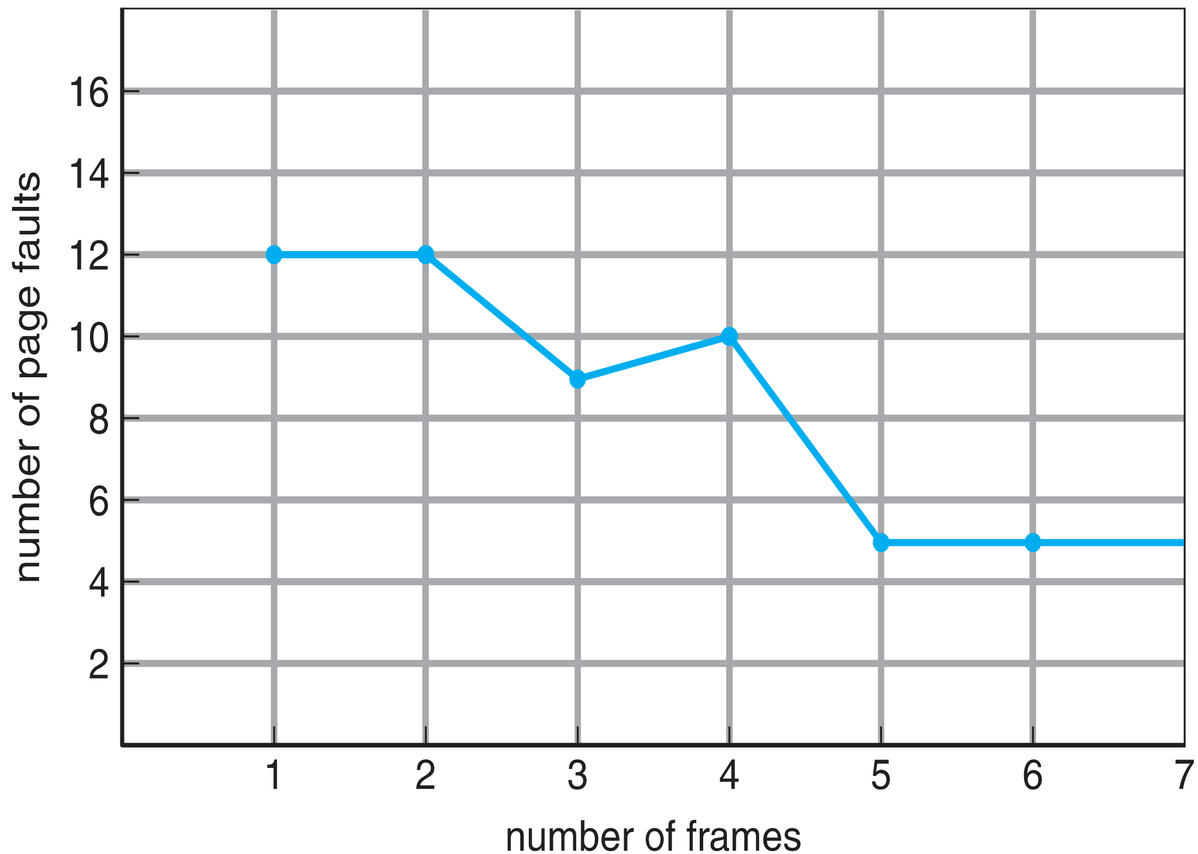
- When a page must be replaced, the oldest page is chosen
  - Can use an FIFO queue to track ages of pages
- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)



15 page faults

- Problem: Adding more frames can cause more page faults!
  - **Belady's Anomaly**

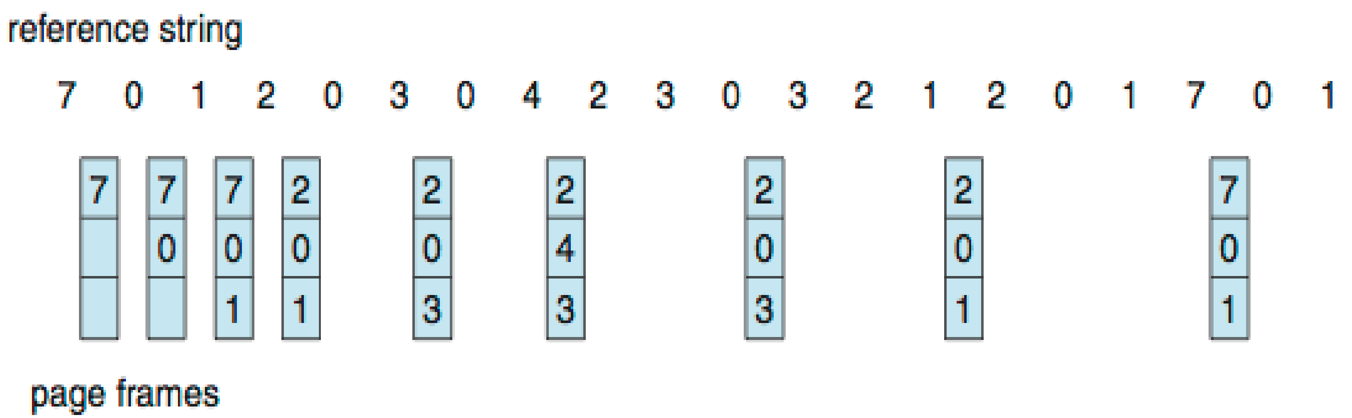
FIFO Illustrating Belady's Anomaly



Reference string: 1,2,3,4,1,2,5,1,2,3,4,5

Optimal Page Replacement Algorithm

- Is there an algorithm that has the lowest page-fault rate and will never suffer from Belady's anomaly?
  - Yes! Such an algorithm is called OPT. It replaces the page that will not be used for longest period of time

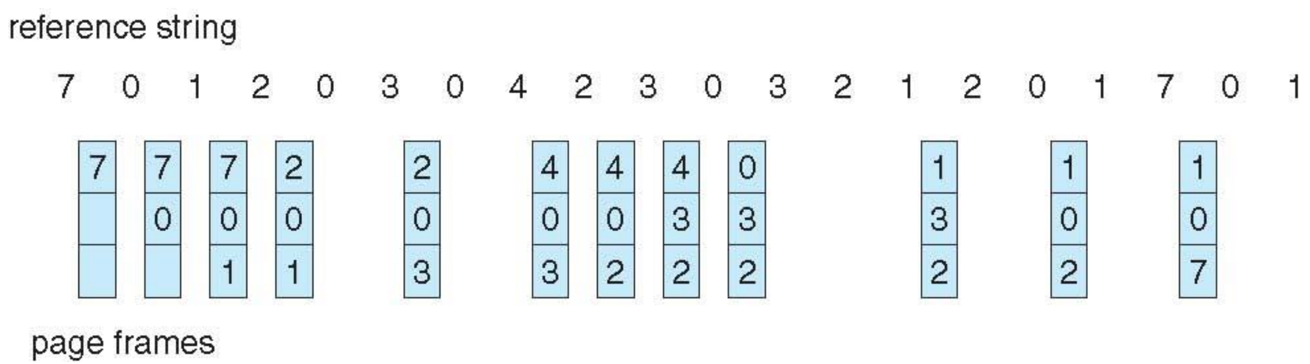


9 page faults

- OPT cannot be implemented because we cannot know the future
  - Used for measuring how well your algorithm performs

Least Recently Used (LRU) Algorithm

- Use past knowledge rather than future knowledge
- Replace the page that has not been used for the longest period of time



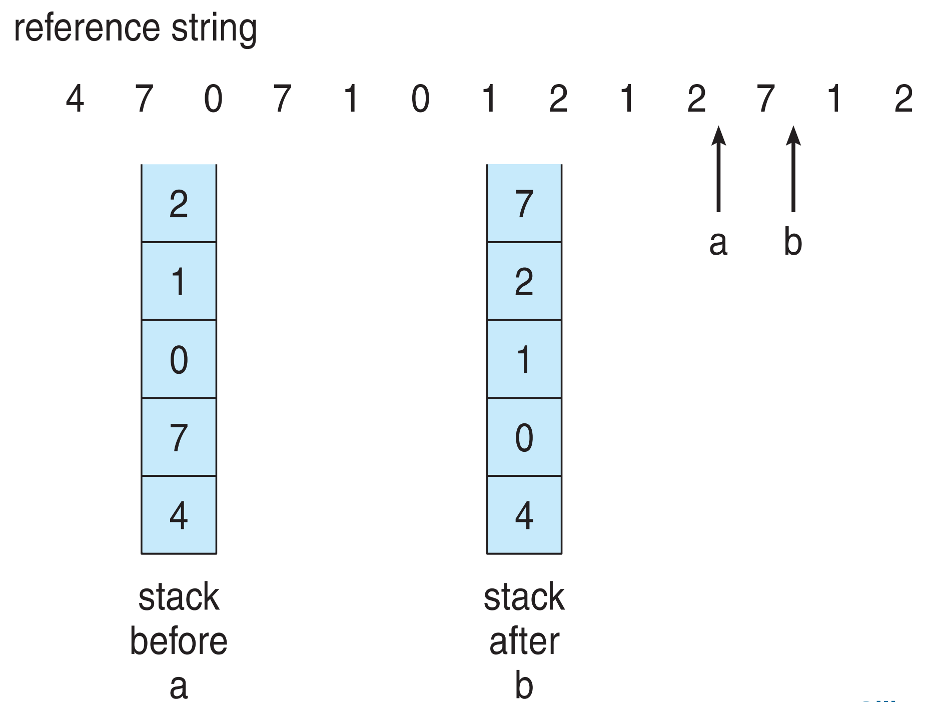
- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- But how to implement?

LRU Algorithm Implementation

- Counter implementation
  - Every page-table entry has a time-of-use field
  - CPU has a logical clock or counter that is incremented for every memory reference
  - Every time a page is referenced, copy the clock value into time-of-use field in the page-table entry for that page
  - When a page needs to be replaced, look at time-of-use fields to find smallest value
    - Search through page table needed

LRU Algorithm Implementation

- Stack implementation
  - Keep a stack of page numbers in a doubly linked list with a head pointer and a tail pointer
  - Whenever a page is referenced, move it to top of stack
    - Most recently used page is always at the top, least recently used page is always at the bottom
  - No search needed for a page replacement



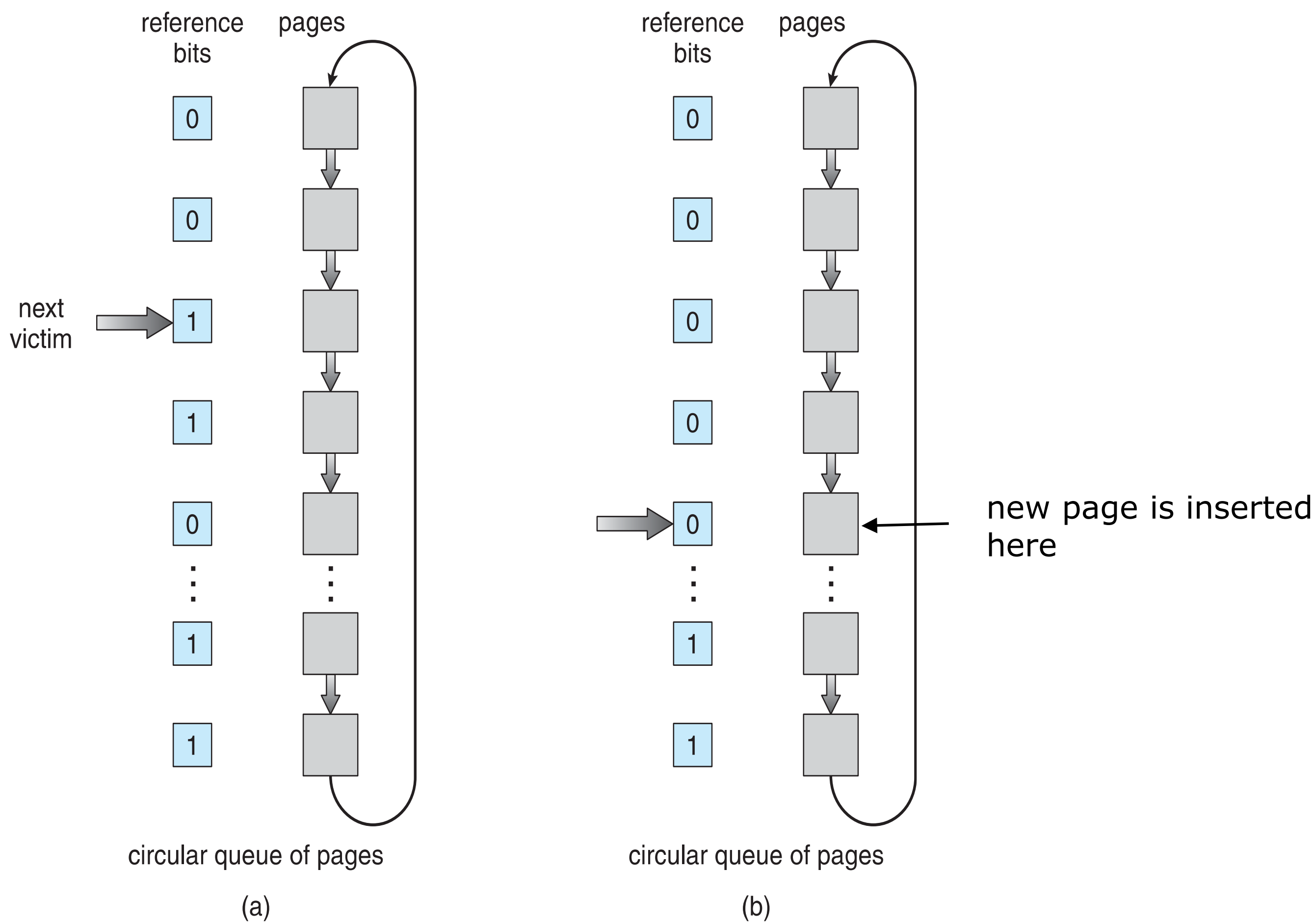
Stack Algorithms

- LRU and OPT are cases of **stack algorithms** that don't have Belady's Anomaly
- A **stack algorithm** is an algorithm with the following property:
  - The set of pages in memory for n frames is always a subset of the set of pages that would be in memory with n+1 frames
- LRU: the set of pages in memory would be the n most recently referenced pages
  - If number of frames is increased, these n pages will still be the most recently referenced and so will still be in memory
- OPT: the set of pages in memory would be the n pages that will not be used for the longest period of time
  - If number of frames is increased, these n pages will still not be used for the longest period of time and so will still be in memory

LRU Approximation Algorithms

- Not many computer systems provide sufficient hardware support for LRU
- Many systems provide hardware support of a **reference bit** to enable LRU approximation
  - With each page associate a reference bit, initially = 0
  - When page is referenced bit set to 1 by hardware
    - We do not know the order, however
- **Second-chance algorithm**
  - FIFO plus hardware-provided reference bit
  - When a page is selected, we inspect its reference bit
    - Reference bit = 0 → replace it
    - Reference bit = 1 then:
      - set reference bit 0, leave page in memory, i.e., give page a second chance
      - consider next page, subject to same rules

Second-Chance (Clock) Page-Replacement Algorithm



- Maintain a circular queue of pages
- A pointer points to the page to be considered next

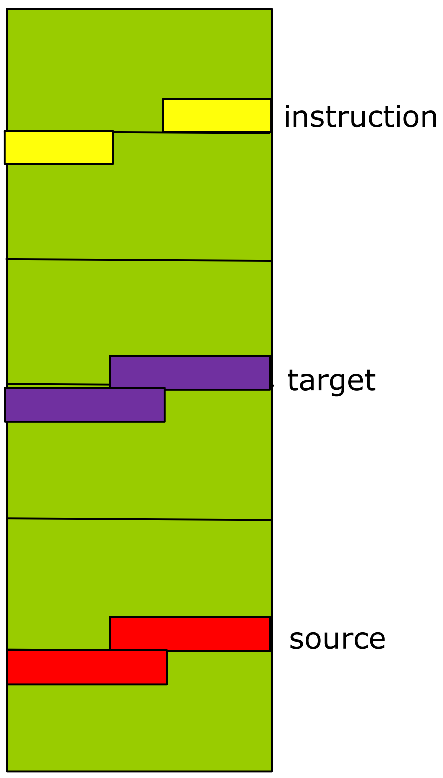
Second Change Algorithm Example

Reference string **0 4 1 4 2 4 3,** 3 frames

- Initially, all frames are empty so after first 3 passes they will be filled with {0, 4, 1} and the reference bit array will be {0, 0, 0}. Also, pointer = 0.
- **Pass-4:** Frame={0, 4, 1}, reference bit = {0, 1, 0}, pointer = 0 (No page needs to be replaced so the candidate is still page in frame 0)
- **Pass-5:** Frame={2, 4, 1}, reference bit = {0, 1, 0} [0 replaced; its reference bit was 0, so it didn't get a second chance], pointer=1
- **Pass-6:** Frame={2, 4, 1}, reference bit ={0, 1, 0}, pointer=1
- **Pass-7:** Frame={2, 4, 3}, reference bit = {0, 0, 0} [4 survived but its reference bit became 0], pointer=0 (as page 1 is replaced)

Allocation of Frames

- **Frame-allocation algorithm** determines
  - How many frames to give each process
  - Which frame to replace
- Each process needs a ***minimum*** number of frames
- Example: IBM 370 needs 6 pages to handle MVC instruction:
  - instruction is 6 bytes, might straddle 2 pages
  - 2 pages to handle *source*
  - 2 pages to handle *target*
- ***Maximum*** number of frames allocated to a process is total frames in the system



Frame Allocation Algorithms

- Equal allocation – For example, if there are 95 frames (after allocating frames for the OS) and 10 processes, give each process 9 frames
  - 5 left over frames can be used as a free-frame buffer pool
- Proportional allocation – Allocate according to the size of process

$$s_i = \text{size of process } p_i$$

$$S = \sum s_i$$

$$m = \text{total number of frames}$$

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

m=64
s_1=10 pages
s_2=120 pages
s=130 pages
a_1=10/130 * 64 ≈ 5
a_2=120/130 * 64 ≈ 59

  - Allocation changes as the degree of multiprogramming changes

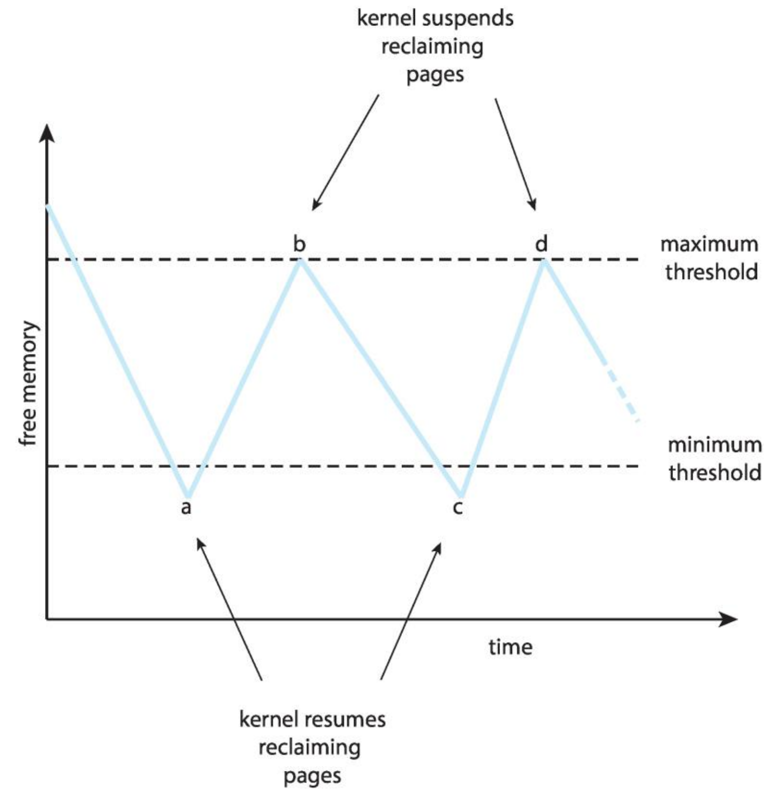
Global vs. Local Allocation

Two categories of page-replacement algorithms:

- **Global replacement** allows a process to select a replacement frame from the set of all frames; one process can take a frame from another
  - Set of pages in memory for a process is affected by the paging behavior of other processes → Process execution time can vary greatly
  - Greater throughput so more common
- **Local replacement** requires that each process select from only its own set of allocated frames
  - More consistent per-process performance
  - But possibly underutilized memory

Reclaiming Pages

- A strategy to implement global page-replacement policy
- All memory requests are satisfied from the free-frame list
  - Page replacement is triggered when the list falls below a minimum threshold
    - A kernel routine begins reclaiming pages from all processes and adding them to the free-frame list
  - The kernel routine stops reclaiming pages when amount of free memory reaches a maximum threshold
- This strategy attempts to ensure there is always sufficient free memory to satisfy new requests



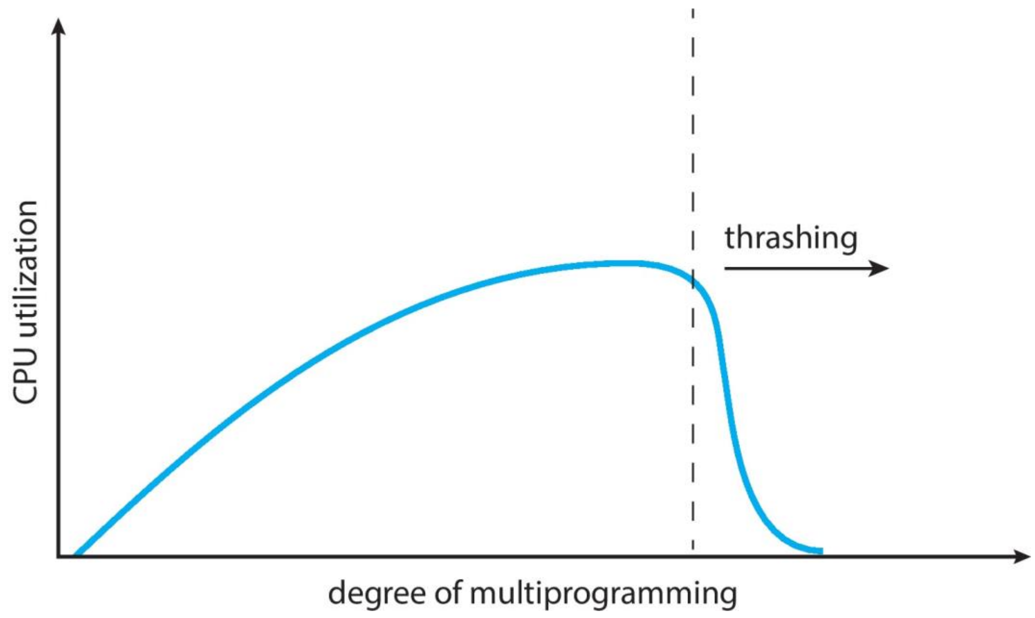
Thrashing

- If a process does not have "enough" frames, the page-fault rate will be very high
  - Page fault to get page
  - Replace existing page (assuming local replacement)
  - But quickly need replaced page back
- **Thrashing** - a process is spending more time paging than executing
  - Leads to low CPU utilization

Thrashing (Cont.)

- Assume a global replacement algorithm is used
- Suppose a new process enters a new phase in its execution and needs more frames
  - It starts faulting and taking frames away from other processes
  - These processes also fault, taking frames from other processes
  - As faulting processes wait for the paging device, CPU utilization decreases
  - OS thinks that it needs to increase the degree of multiprogramming
  - Another process is added to the system
  - The new process takes frames from other processes, causing more page faults
  - CPU utilization drops even further, OS tries to increase the degree of multiprogramming even more
  - Thrashing has occurred and system throughput plunges!
- To stop thrashing, we must **decrease** the degree of multiprogramming

Thrashing (Cont.)

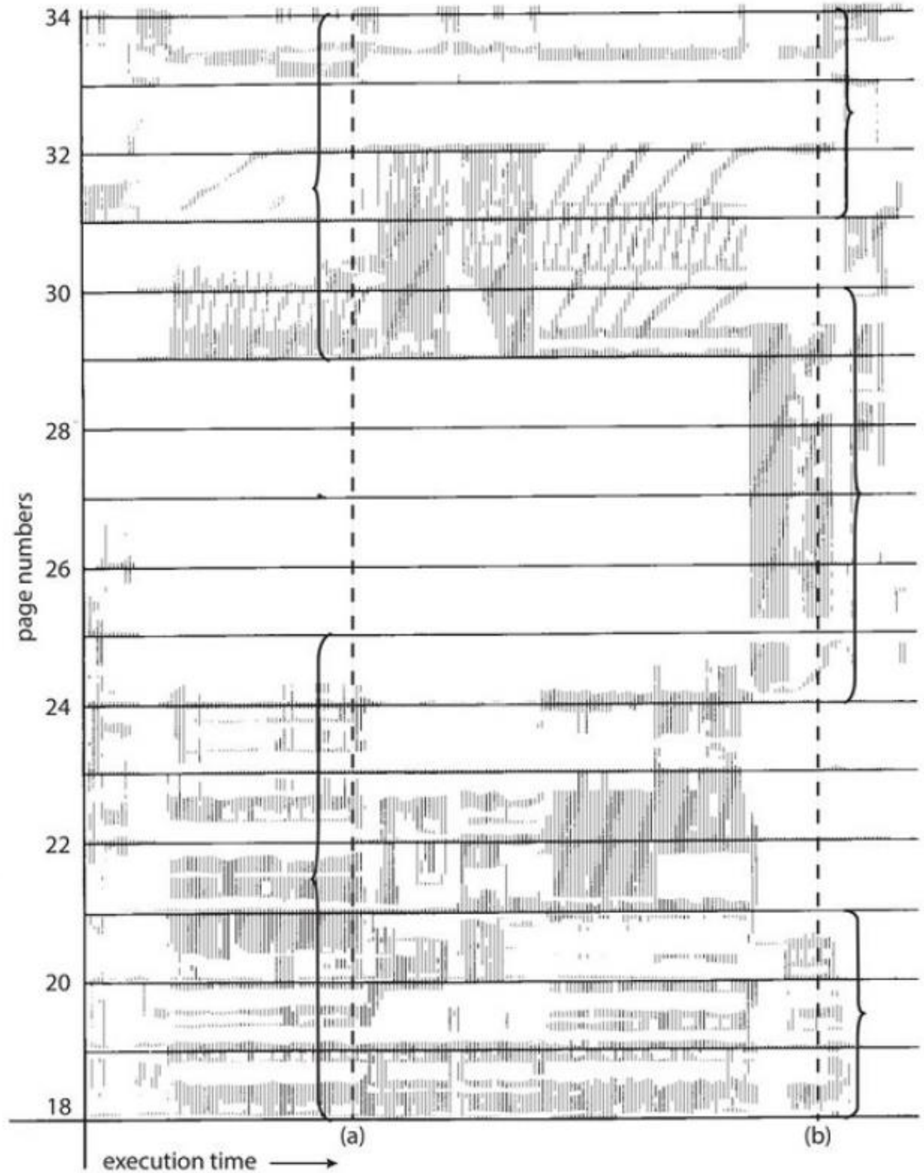


- Symptom of thrashing: low CPU utilization and high page fault rate
- When thrashing sets in, we must **decrease** the degree of multiprogramming

Preventing Thrashing

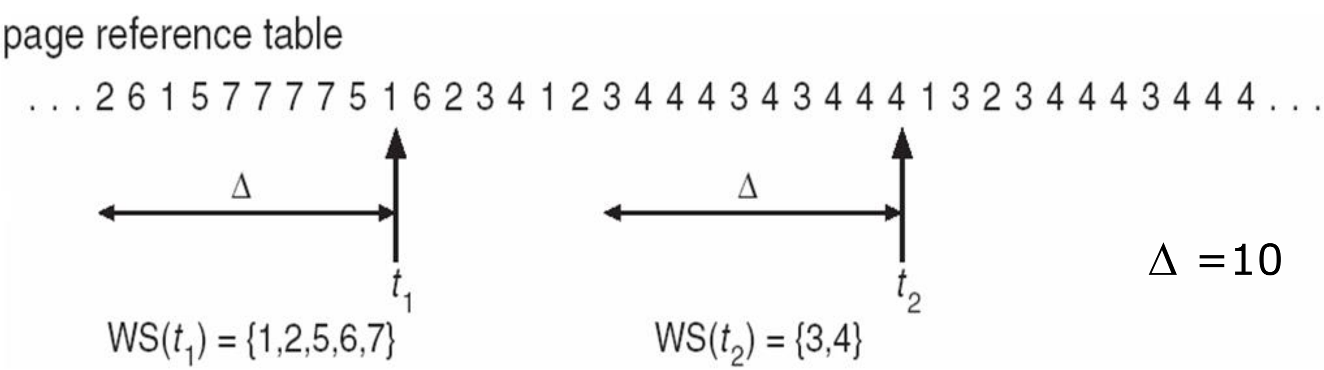
- To prevent thrashing, we must provide a process as many frames as it needs
- How do we know how many frames a process needs?
- **Locality model** of process execution
  - As a process executes, it moves from one locality to another
  - A **locality** is a set of pages that are actively used together
  - Localities may overlap
- A process will thrash if we do not allocate enough frames to accommodate the size of its current locality

Locality In A Memory-Reference Pattern



Working-Set Model

- We can use the **working-set model** to determine how many frames should be allocated to a process
  - The working-set model is based on the assumption of locality
- $\Delta \equiv$ a fixed number of page references, defines the working-set window Example: Δ =10,000
- WS_i (Working Set of process P_i) = Set of pages in the most recent Δ page references (WS_i varies in time)
  - The working set is an approximation of the program's locality
  - WSS_i denotes the size of WS_i



Δ =10

WSS =5

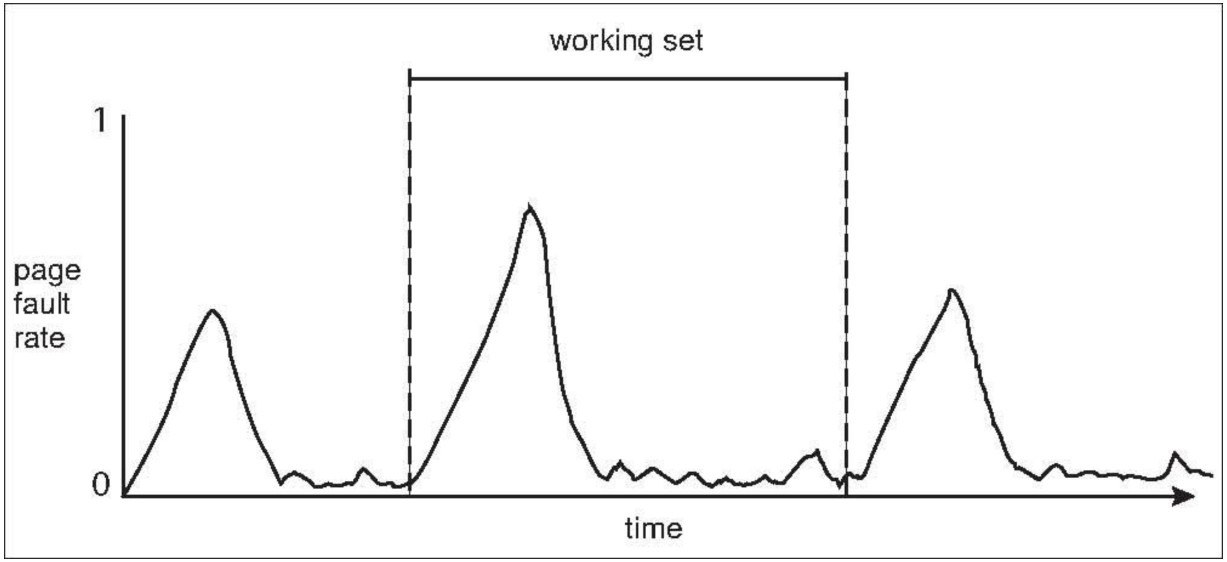
WSS =2

Working-Set Model (Cont.)

- The accuracy of the 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 $\Delta = \infty \Rightarrow$ will encompass all pages touched during the process execution
- Total demand for frames $D = \Sigma\, WSS_i$
  - $D > m$ (m=total number of frames) → Thrashing
- Policy:
  - Each process should be allocated K frames where K = WSS of the process
  - If D > m, then suspend and swap out a process
  - If m-D > certain threshold (i.e., a typical WSS for a process), one more process can be initiated

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



Page-Fault Frequency

- More direct approach than working-set model
- Establish upper bound and lower bound on the desired page-fault rate and use local replacement policy
  - If actual rate too low, process loses a frame
  - If actual rate too high, process gains a frame