Chapter 9: Virtual Memory

Chapter 9: Virtual Memory

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

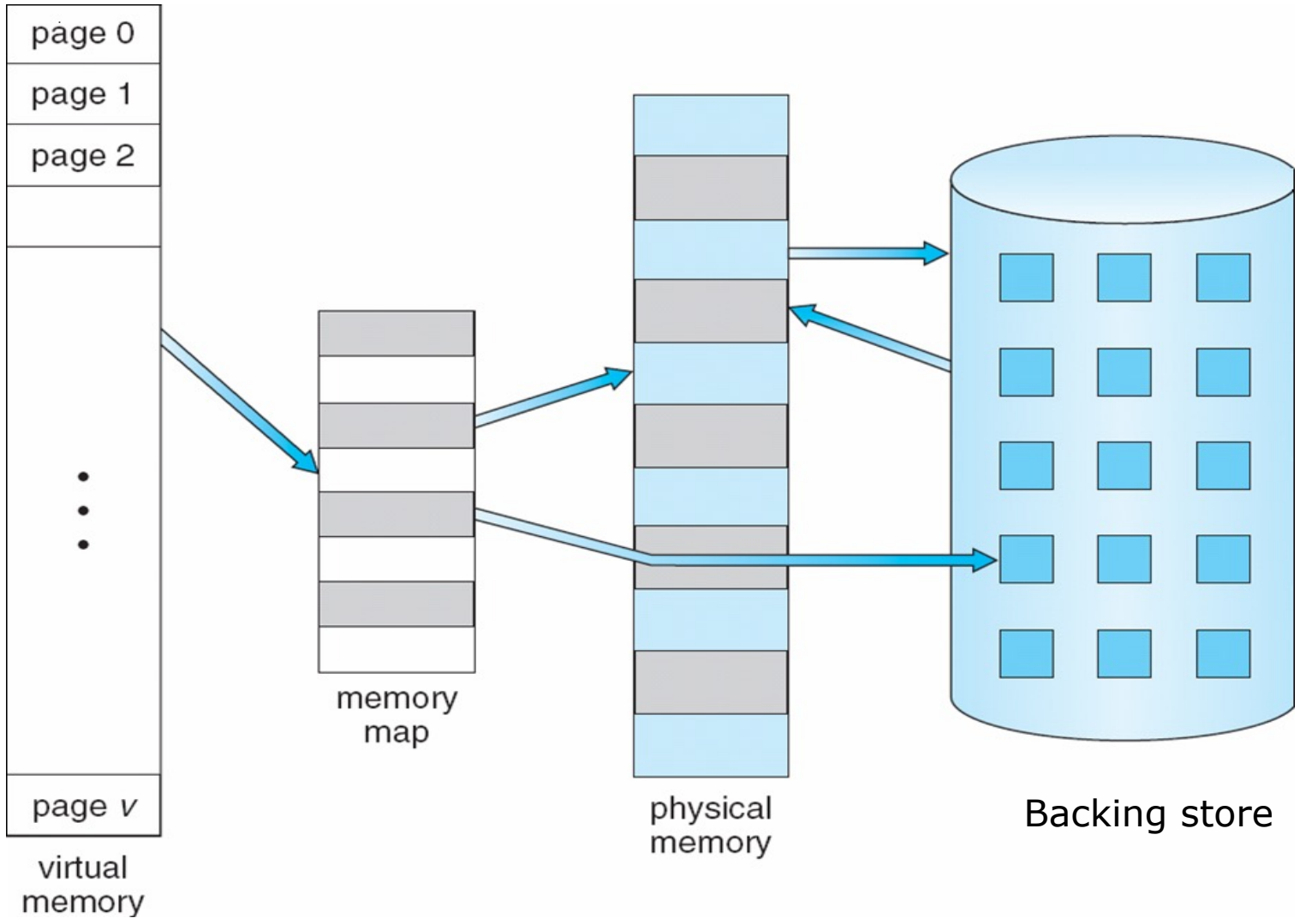
Objectives

- To describe the benefits of a virtual memory system
- To explain the concepts of demand paging, page-replacement algorithms, and allocation of page frames
- To discuss the principle of the working-set model

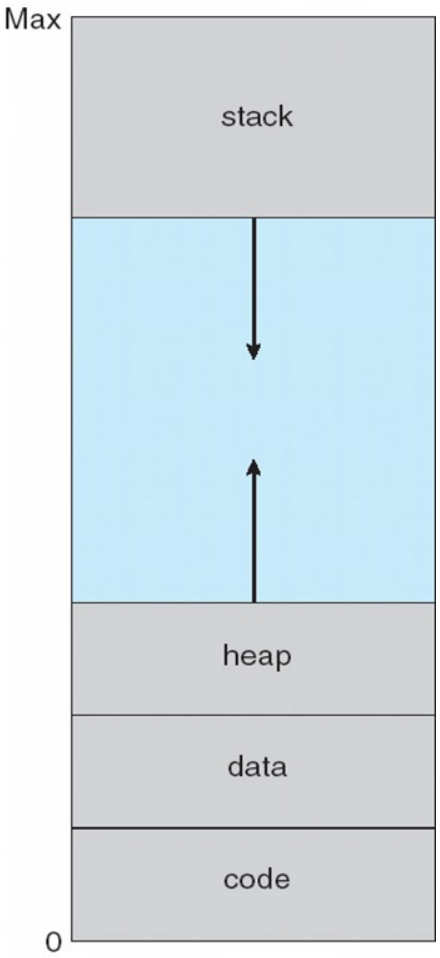
Background

- **Virtual memory** – separation of user logical memory 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
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

Virtual Memory That is Larger Than Physical Memory



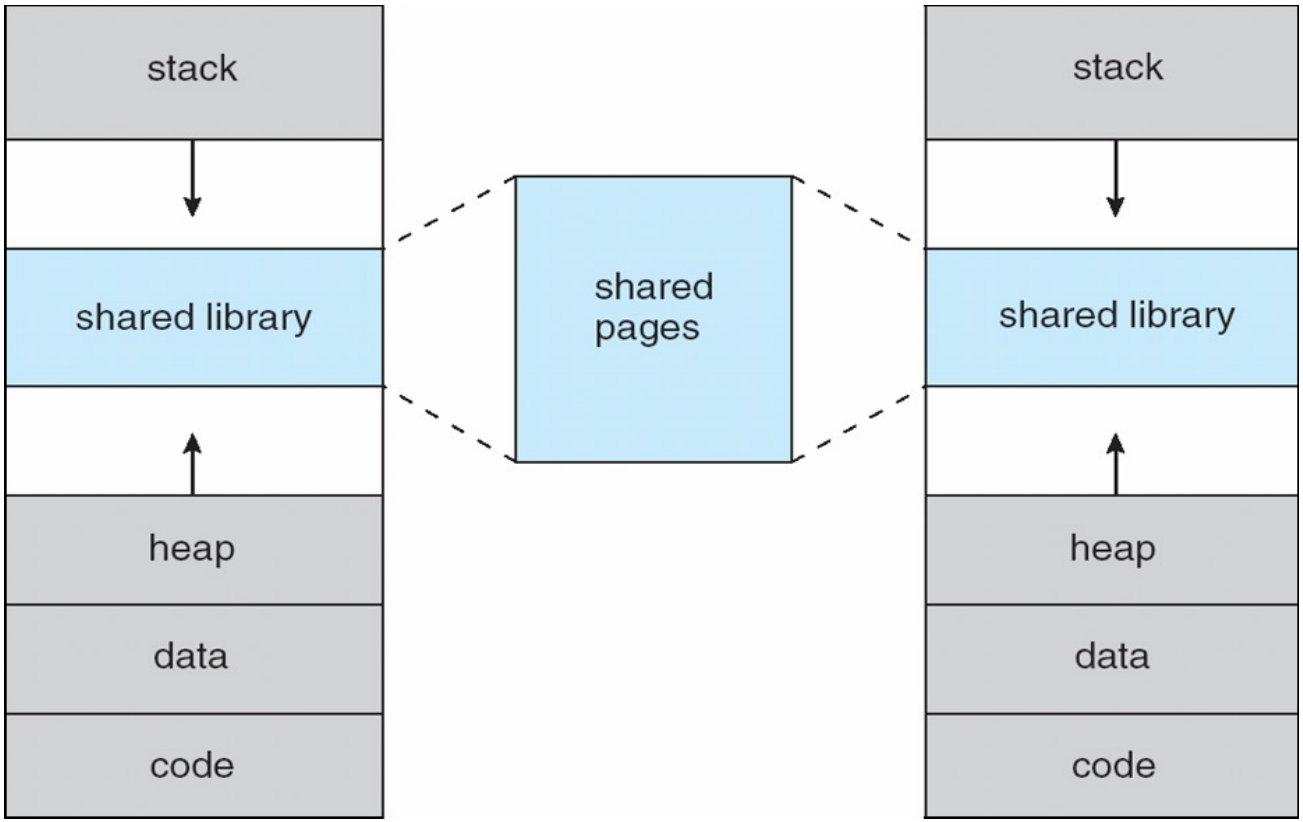
Virtual-address Space



Refers to the logical view of how a process is stored in memory.

A process begins at a certain logical address (such as 0) and exists in contiguous memory. Physical frames may not be contiguous.

Shared Library Using Virtual Memory



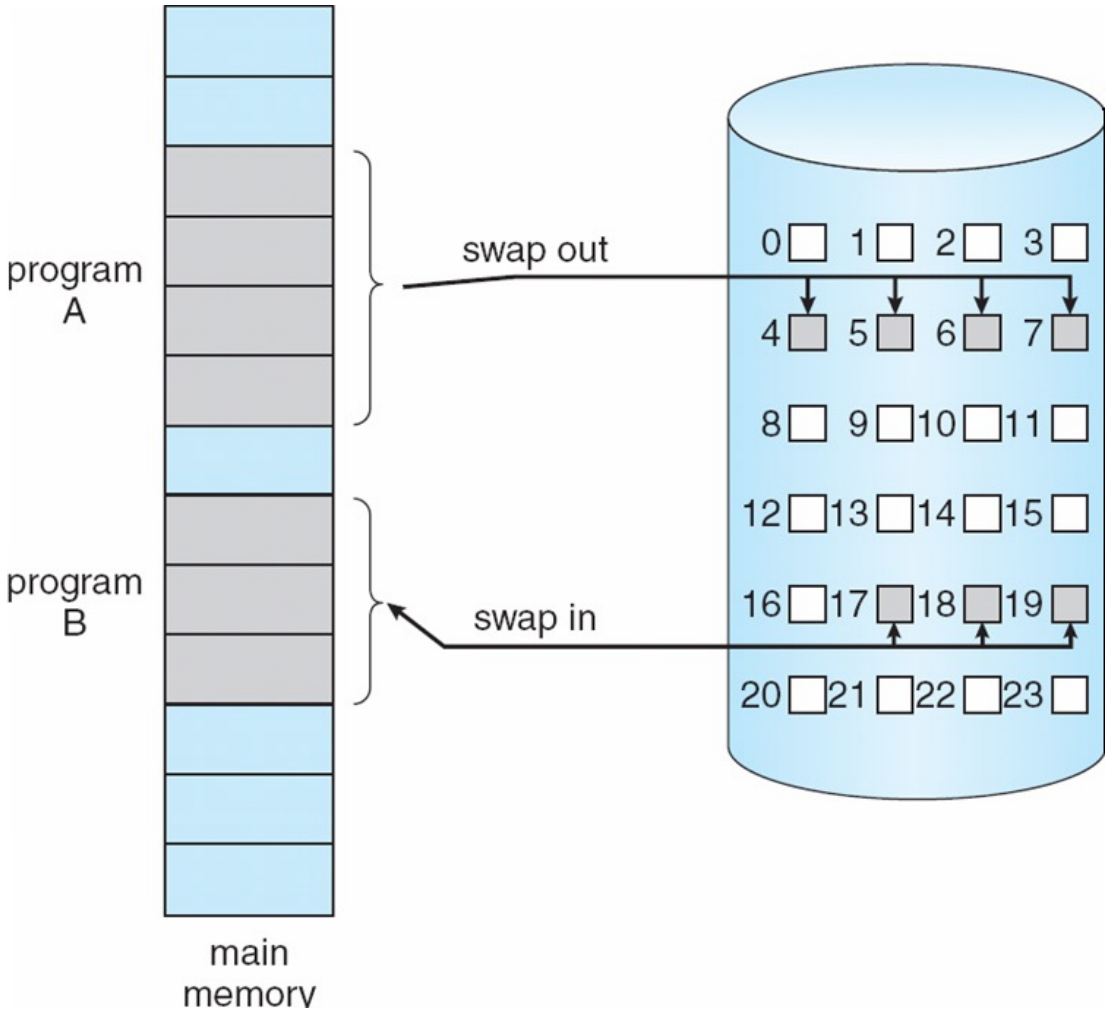
Stack or heap grow if we wish to dynamically link libraries during program execution.

System library can be shared by several process through mapping the shared object into a virtual address space.

Demand Paging

- Bring a page into memory **only when it is needed**
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users
- 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**

Transfer of a Paged Memory to Contiguous Disk Space



We use pager because it is concerned with the individual pages of a process.

Valid-Invalid Bit

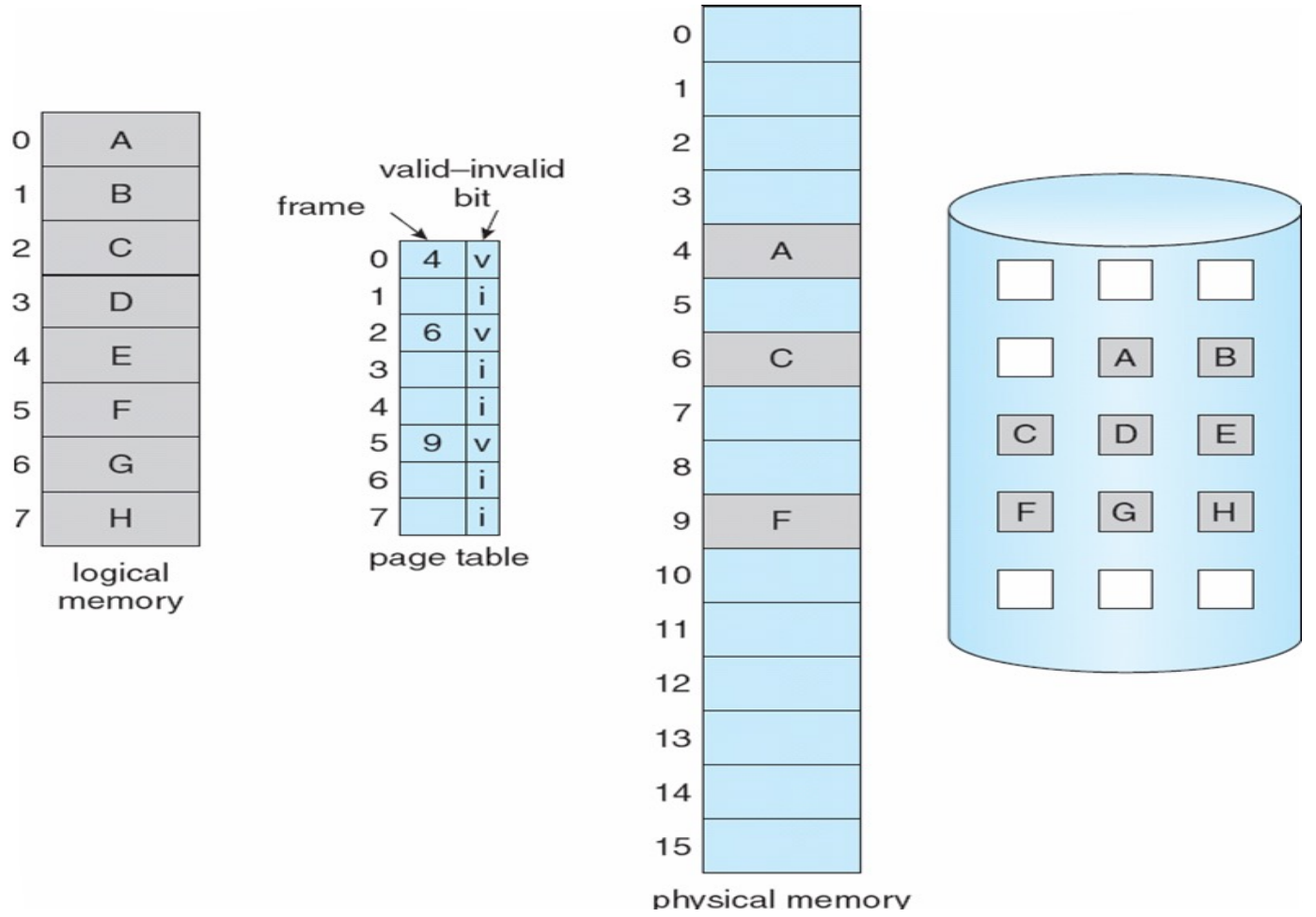
- With each page table entry a valid–invalid bit is associated (**v** ⇒ in-memory, **i** ⇒ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:

Frame #	valid-invalid bit
	v
	v
	v
	v
	i
….	
	i
	i

page table

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

Page Table When Some Pages Are Not in Main Memory



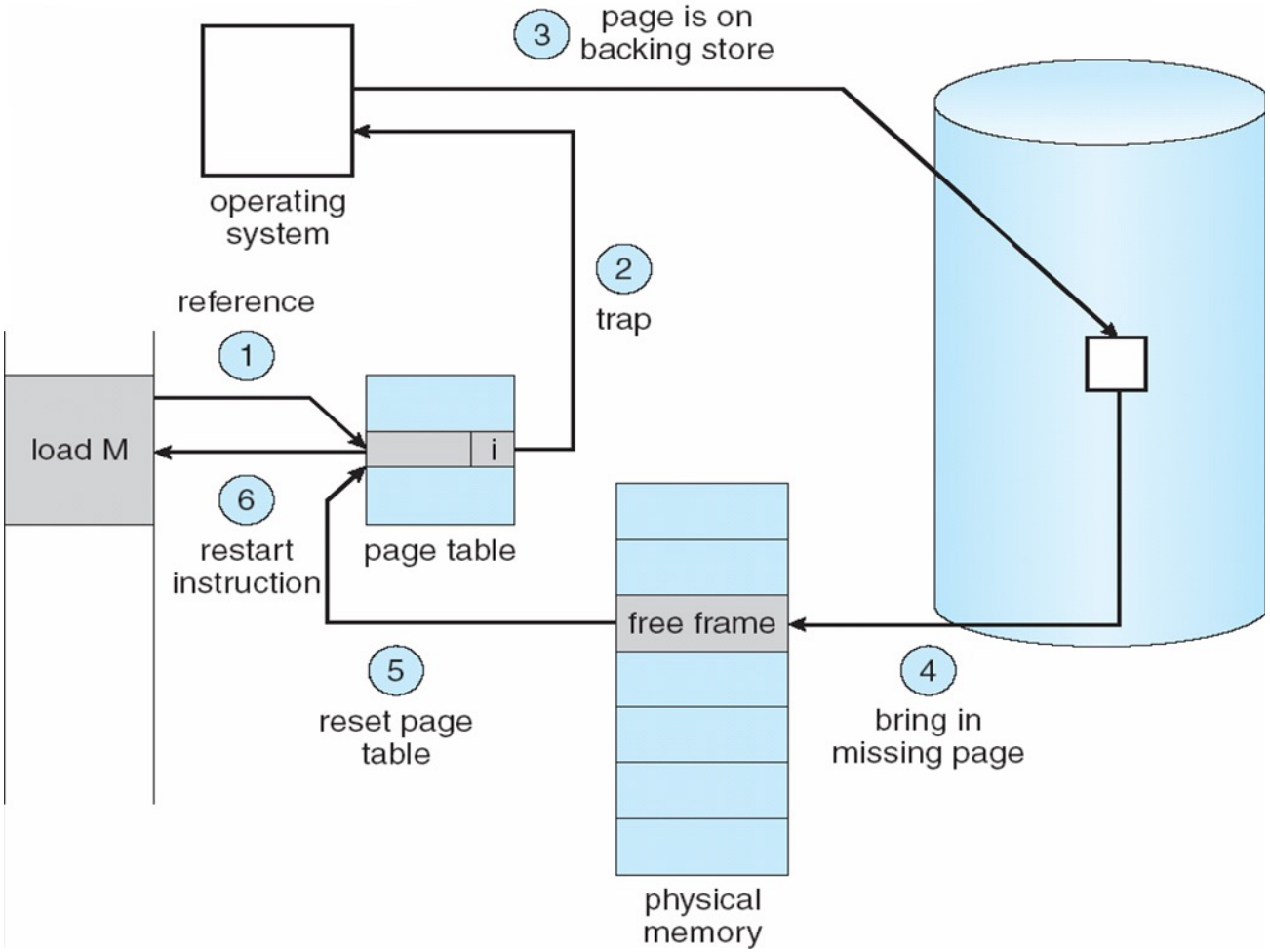
Page Fault

- 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. Get empty frame
3. Swap page into frame
4. Reset tables
5. Set validation bit = **v**
6. Restart the instruction that caused the page fault

Steps in Handling a Page Fault



Page Fault (Cont.)

- Restart instruction
  - The major difficulty arise when one instruction may modify several different locations.
  - One solution is to access both ends of both blocks. If a page faulty is going to occur, it will happen at this step before anything is modified. The move can take place if all relevant pages are in memory.
  - The other solution uses temporary registers to hold the values of overwritten locations.

Performance of Demand Paging

- Page Fault Rate $0 \le p \le 1.0$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault

- Effective Access Time (EAT)

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

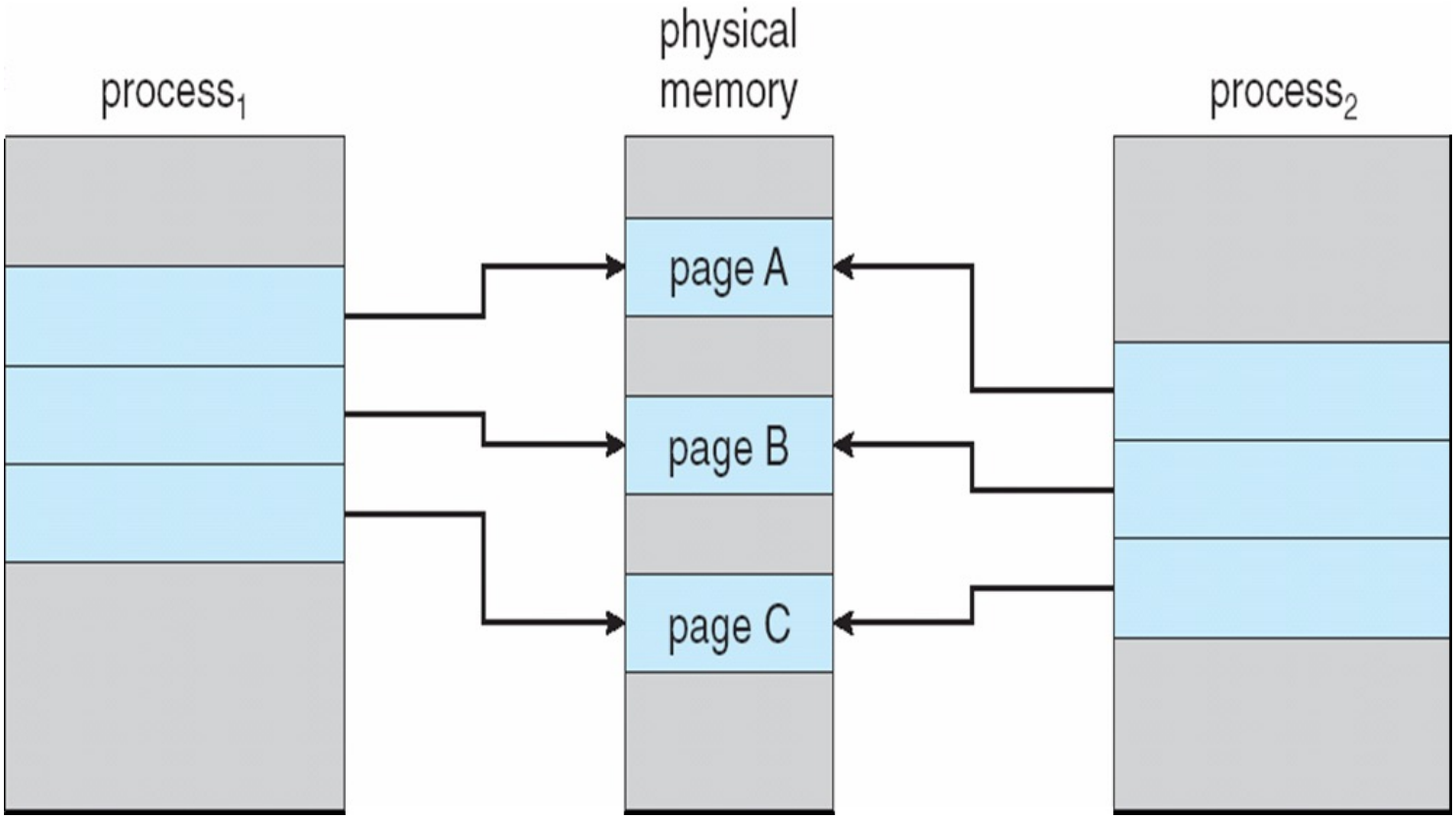
Demand Paging Example

- 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
  EAT = 8.2 microseconds.
  This is a slowdown by a factor of 40!!
- We can allow fewer than one out of 399,990 page-fault.

Process Creation

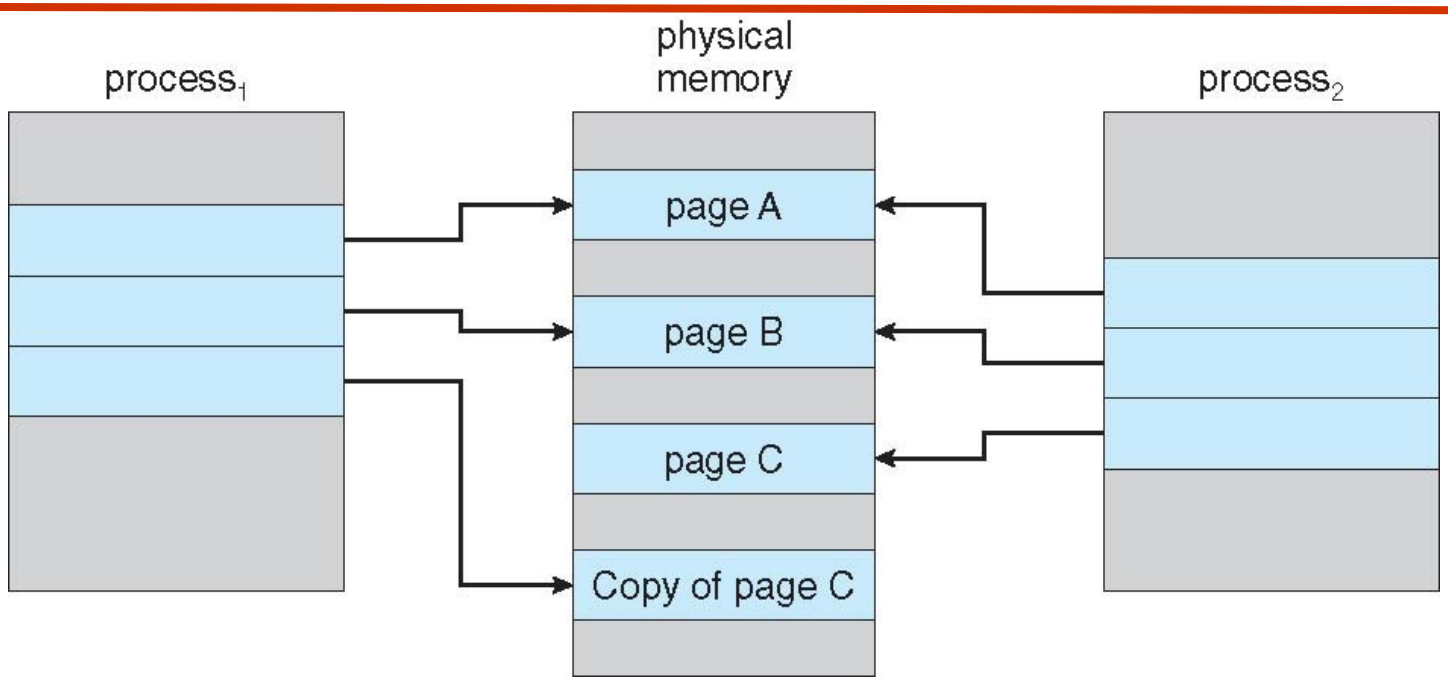
- Virtual memory allows other benefits during process creation:
  - Copy-on-Write (COW) allows both parent and child processes to initially *share* the same pages in memory.
    If either process modifies a shared page, only then is the page copied
  - COW allows more efficient process creation as only modified pages are copied
  - Free pages are allocated from a **pool** of zeroed-out pages

Before Process 1 Modifies Page C



If process 1 attempts to modify page C of the stack, with the page C set to by copy-on-write.

After Process 1 Modifies Page C

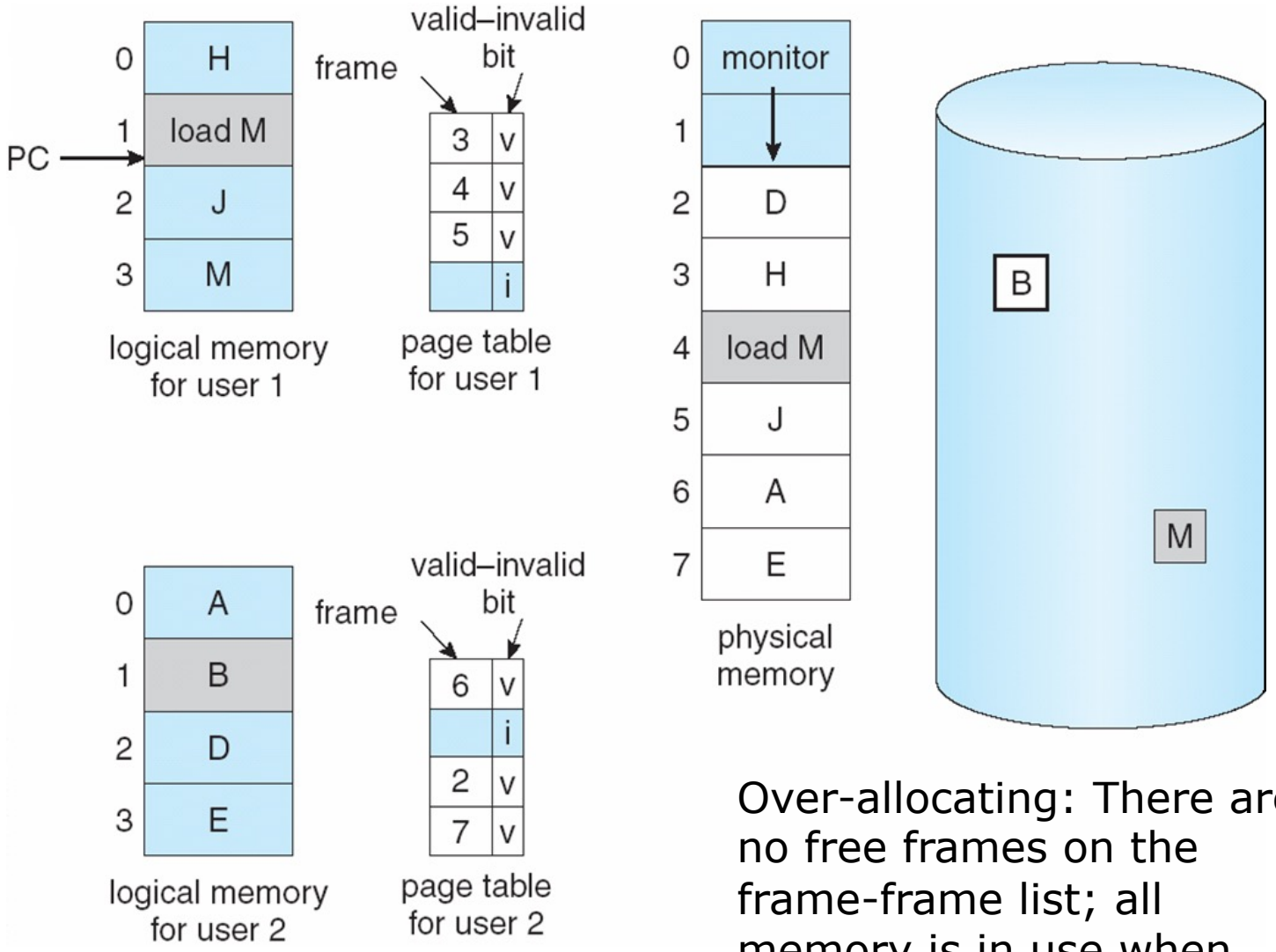


- Only the page C that are modified by process 1 is copied. All unmodified pages can be shared by the parent and child process.
- Only pages that can be modified need to be marked as copy-on-write. Pages that cannot be modified can be shared by the parent and child.
- Copy-on-write is a common technique used by several OS such as Windows, Linux, and Solaris.

What happens if there is no free frame?

- Page replacement – find some page in memory, but not really in use, swap it out
  - algorithm
  - performance – want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times

Need For Page Replacement

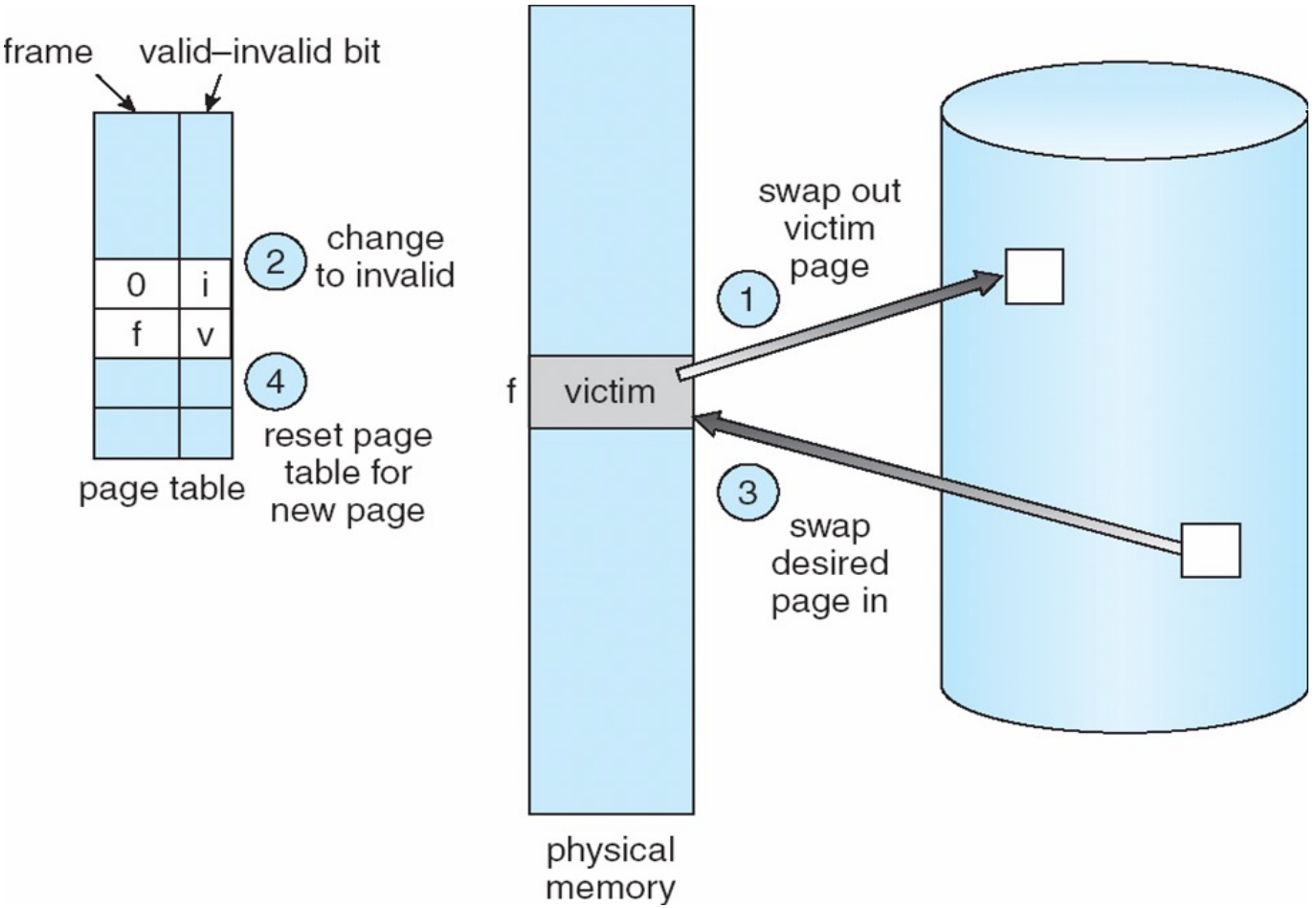


Over-allocating: There are no free frames on the frame-frame list; all memory is in use when user 1 tries to load M.

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



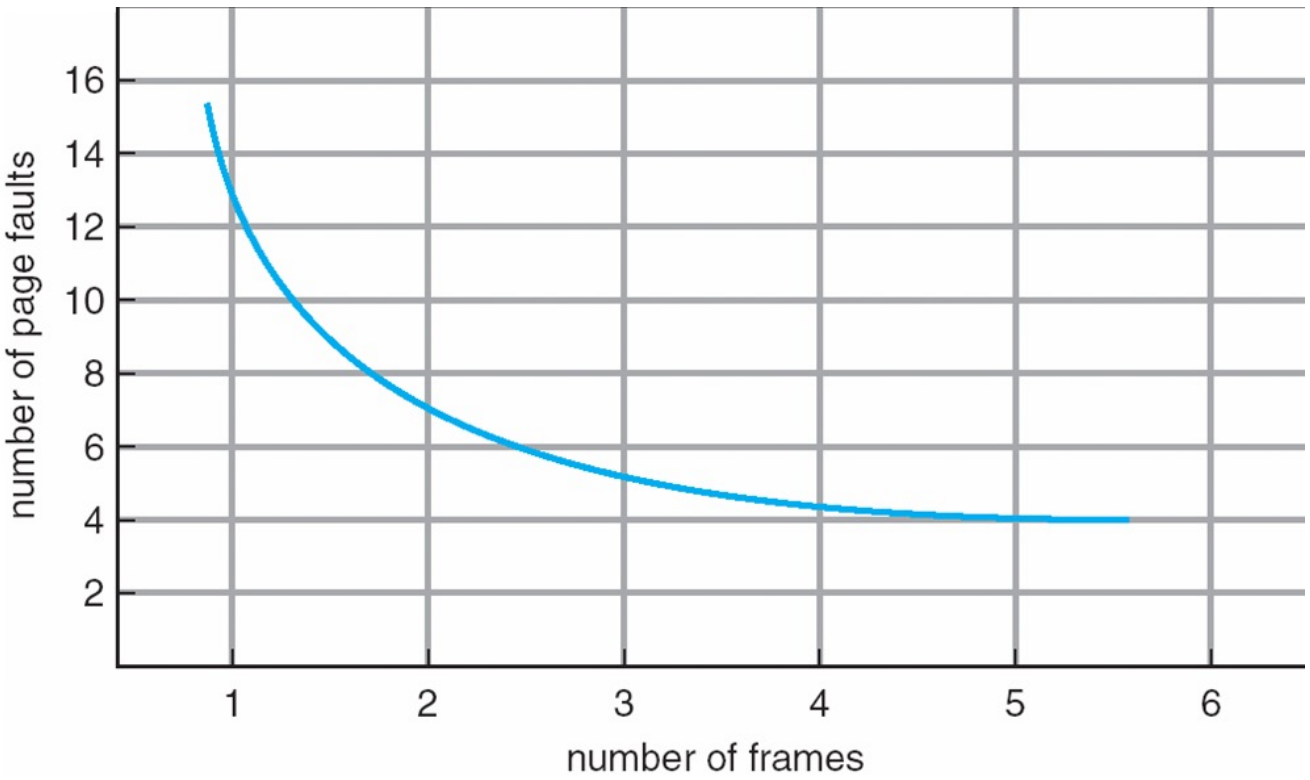
Page Replacement Algorithms

- 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

- In all our examples, the reference string (the sequence of **12** needed page number) is

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

Graph of Page Faults Versus The Number of Frames



- As the number of frames available increases, the number of page faults should decrease.
- If only one frame available, we would have a replacement with every reference, resulting in eleven faults.

First-In-First-Out (FIFO) Algorithm

- 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)

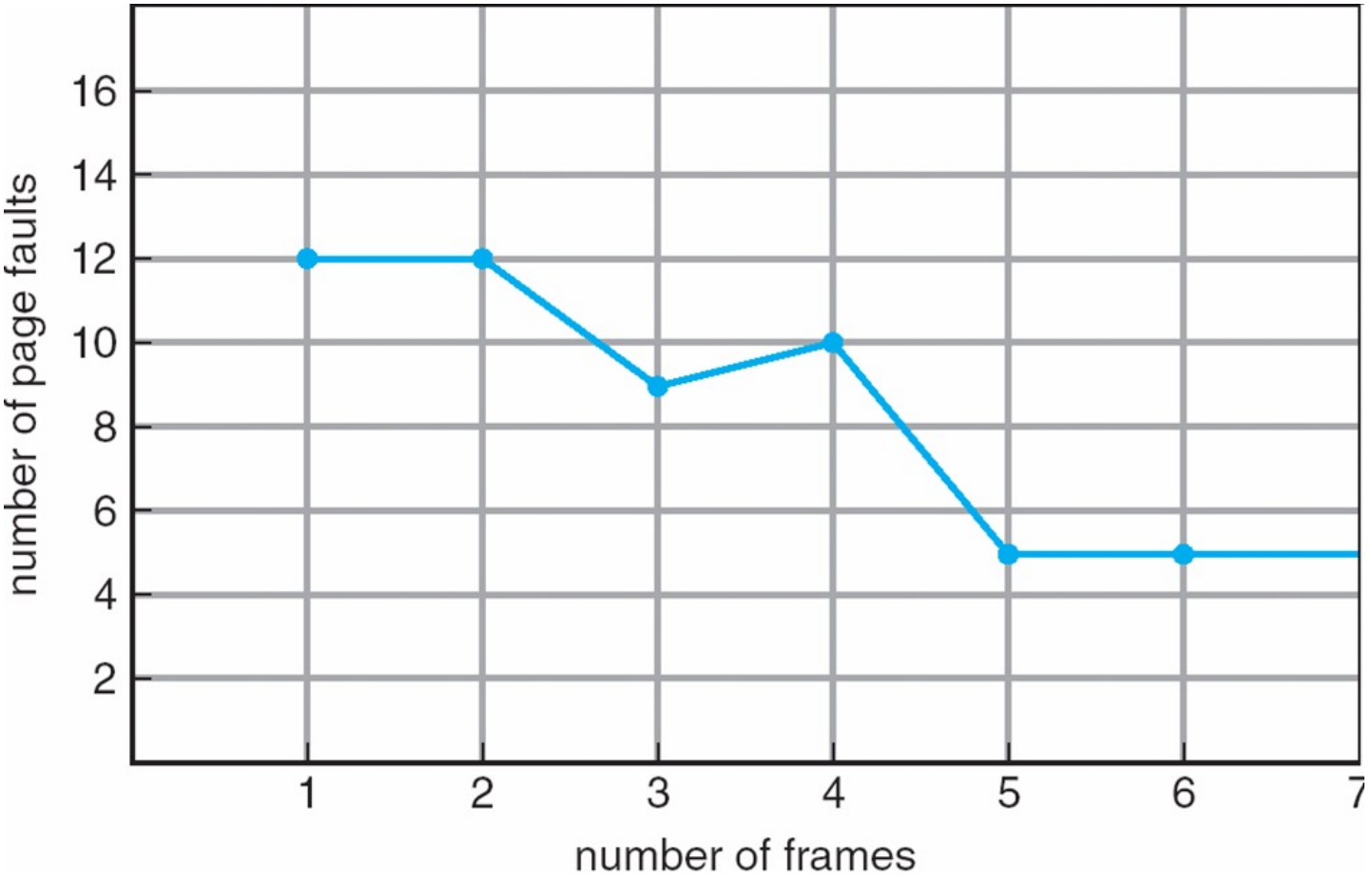
1	1	4	5	
2	2	1	3	9 page faults
3	3	2	4	

- 4 frames

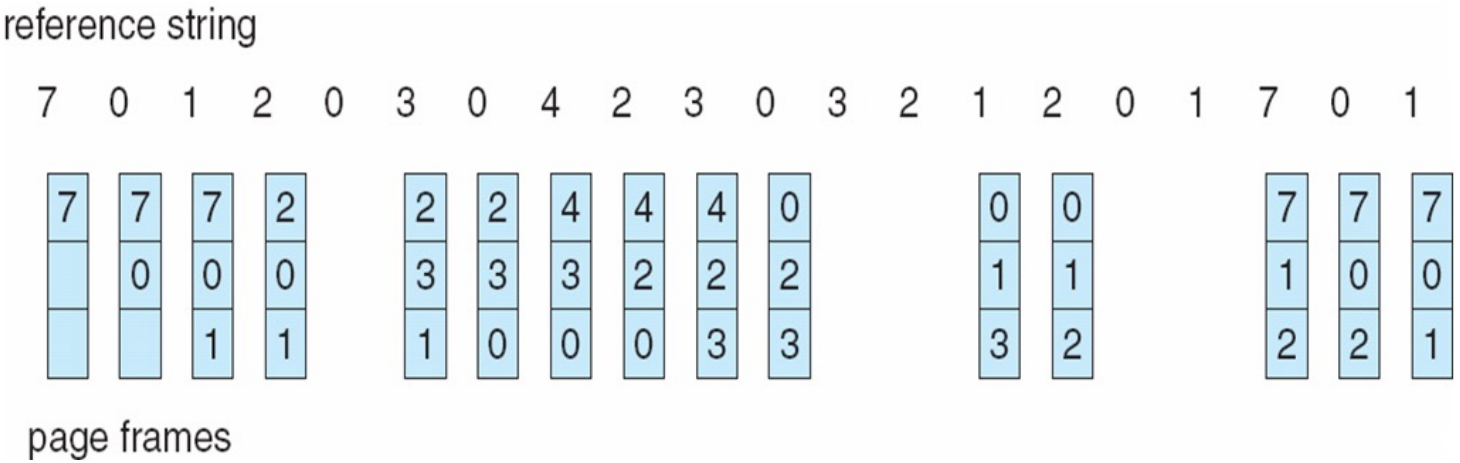
1	1	5	4	
2	2	1	5	10 page faults
3	3	2		
4	4	3		

- Belady's Anomaly: more frames ⇒ more page faults

FIFO Illustrating Belady's Anomaly



More example -- FIFO Page Replacement

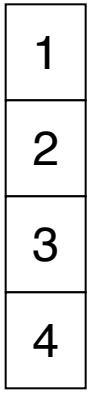


15 page faults

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



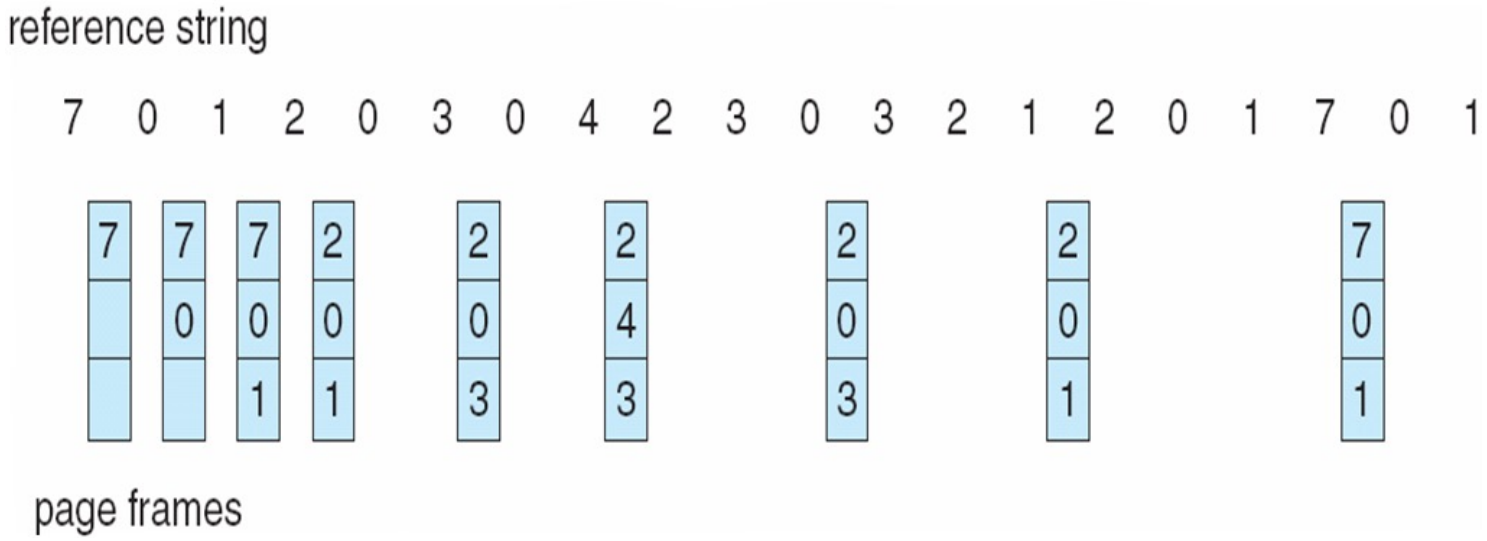
4

5

6 page faults

- How do you know this?
- Used for measuring how well your algorithm performs

Example -- Optimal Page Replacement



The reference to page 2 replaces page 7 because page 7 will not used until reference 18, whereas page 0 will be used at 5, and page 1 at 14.

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.

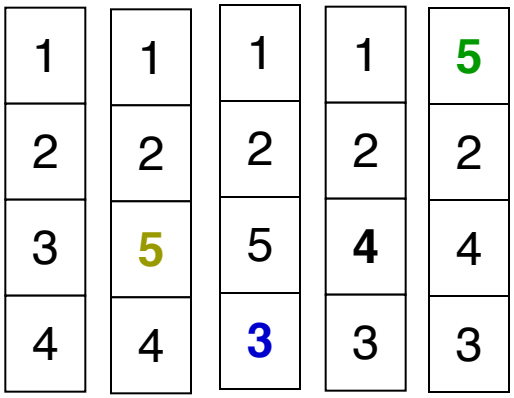
9 page faults which is better than 15 ones in the FIFO algorithm.

Least Recently Used (LRU) Algorithm

- The FIFO algorithm uses the time when a page was brought into memory, whereas the optimal replacement algorithm uses the time when a page is to be used.
- If we use the recent past as an approximation of the near future, we can replace the page that **has not been used for the longest period of time.**

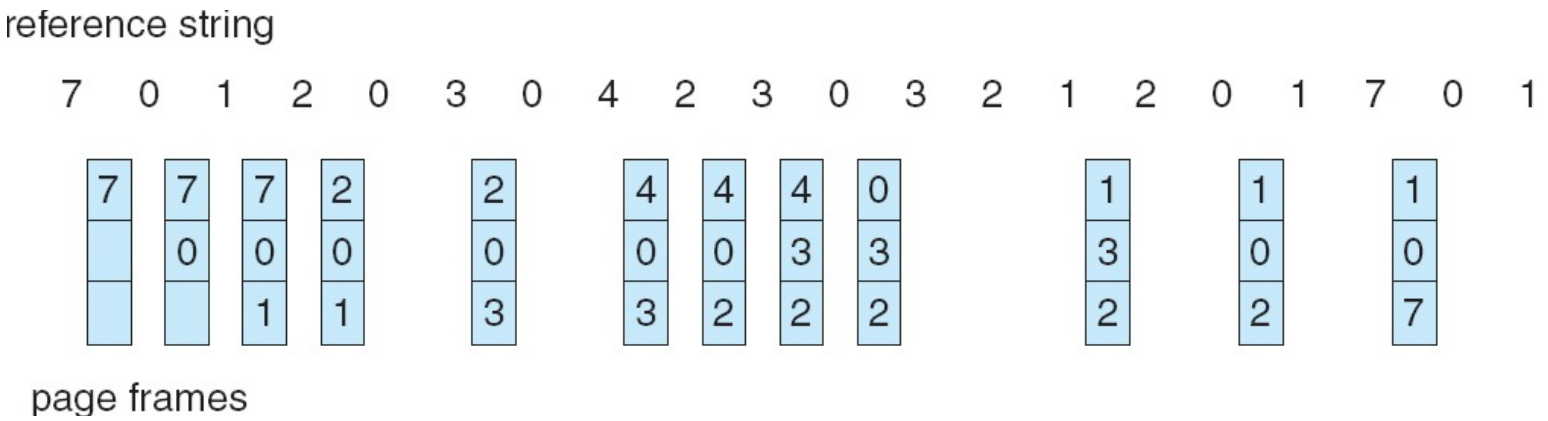
Least Recently Used (LRU) Algorithm

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



- 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 are to change

LRU Page Replacement

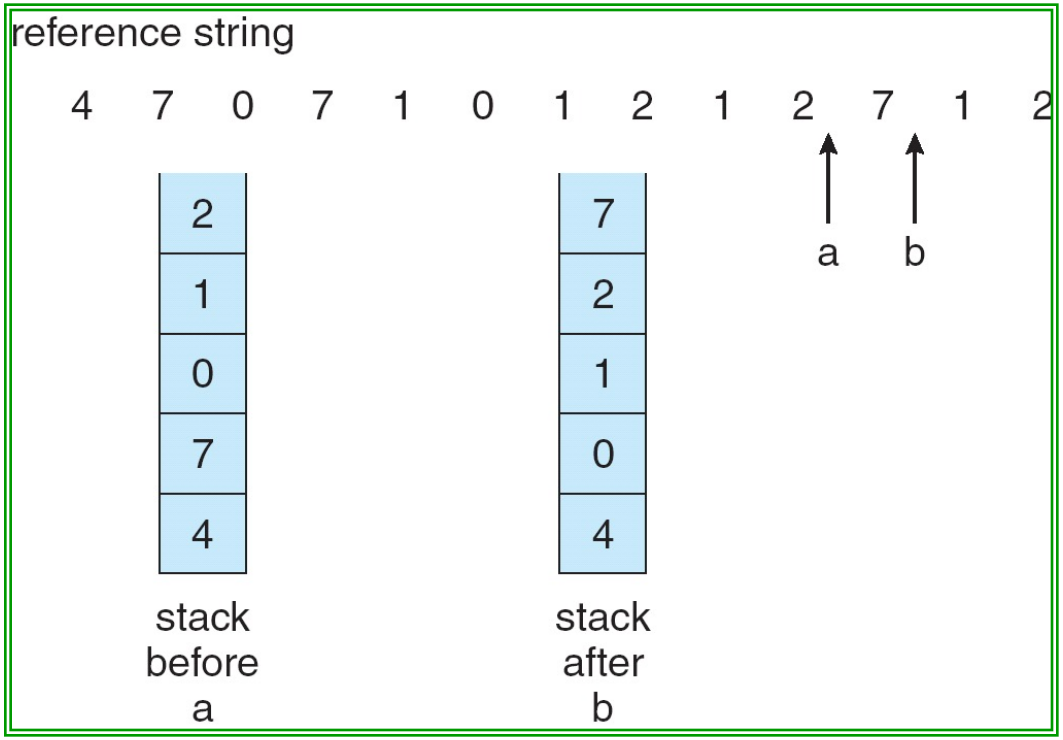


When reference to page 4 occurs, of the three frames in memory, page 2 was used least recently. LRU replaces page 2, not knowing that page 2 is about to be used.

12 page faults of LRU algorithm is still better than 15 from FIFO.

LRU Algorithm (Cont.)

- Stack implementation – keep a stack of page numbers in a double link form:
  - Page referenced:
    - When a page is the stack is referenced, it is moved to the top
    - requires 6 pointers to be changed
  - No search for replacement



LRU Approximation Algorithms

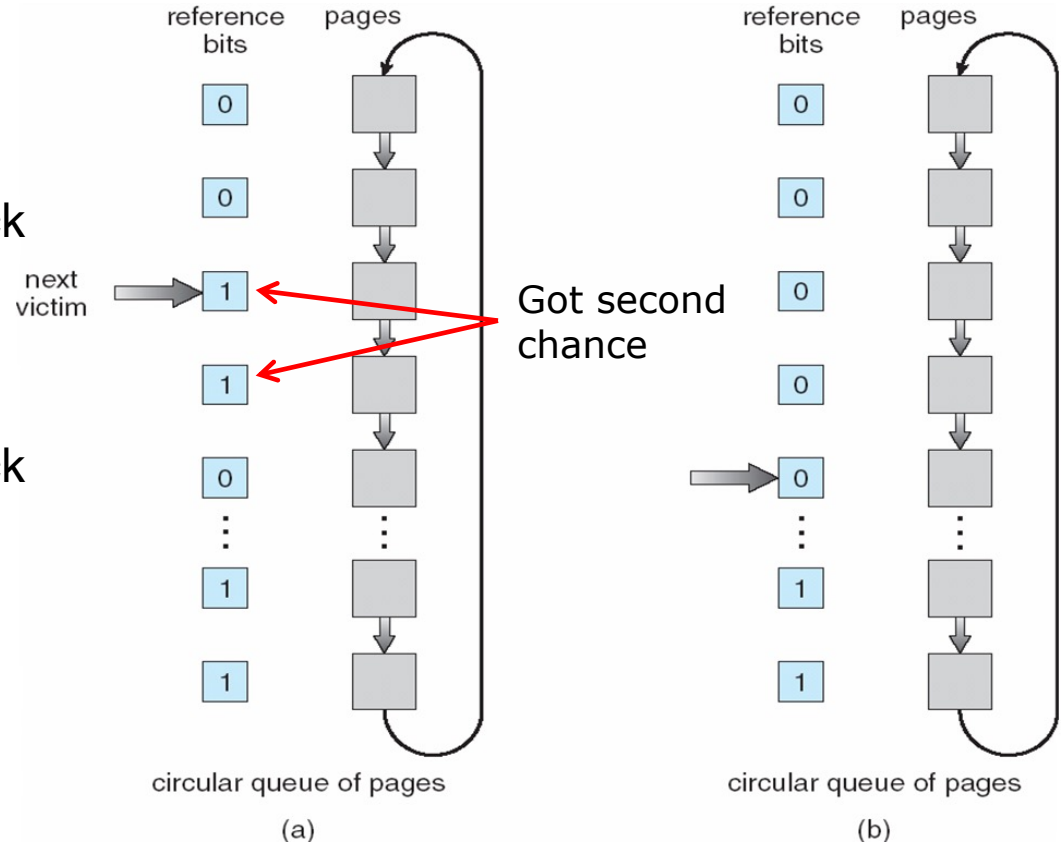
- Reference bit (set by hardware)
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
  - Replace the one which is 0 (if one exists)
    - We do not know the order, however
  - Provide basis for approximate LRU replacement.

Additional Reference Bits Algorithm

- Keep an 8-bit byte for each page in a table in memory.
- At regular intervals, a time interrupt transfers control to the OS.
- The OS shifts the reference bit for each page into the high-order of its 8-bit byte, shifting the other bits right by 1 bit and discarding the low-order bit.
- For example: 0000 1111 becomes 1000 0111
- If a page is never used, it is shift register contains 0000 0000, and 1111 1111 if it is used at least once in each period.
- **1**100 0100 is used more recently than **0**111 0111. The lower one can be replace.

Second-Chance (clock) Page-Replacement Algorithm

- Second chance
  - Need reference bit
  - Clock replacement
  - If page to be replaced (in clock order) has reference bit = 0 then:
    - Replace the page
  - If page to be replaced (in clock order) has reference bit = 1 then:
    - set reference bit 0
    - leave page in memory
    - It arrival time is reset to current the current time



Allocation of Frames

- Each process needs *minimum* number of pages
- Example: IBM 370 – 6 pages to handle SS MOVE instruction:
  - instruction is 6 bytes, might span 2 pages
  - 2 pages to handle *from*
  - 2 pages to handle *to*
- Two major allocation schemes
  - fixed allocation
  - priority allocation

Fixed Allocation

- Equal allocation – For example, if there are 100 frames and 5 processes, give each process 20 frames.
- **Proportional allocation** – Allocate according to the size of process
  - s_i = size of process p_i
  - $S = \sum s_i$
  - m = total number of frames
  - a_i = allocation for $p_i = \frac{s_i}{S} \times m$

$$m = 64$$
$$s_i = 10$$
$$s_2 = 127$$
$$a_1 = \frac{10}{137} \times 64 \approx 5$$
$$a_2 = \frac{127}{137} \times 64 \approx 59$$

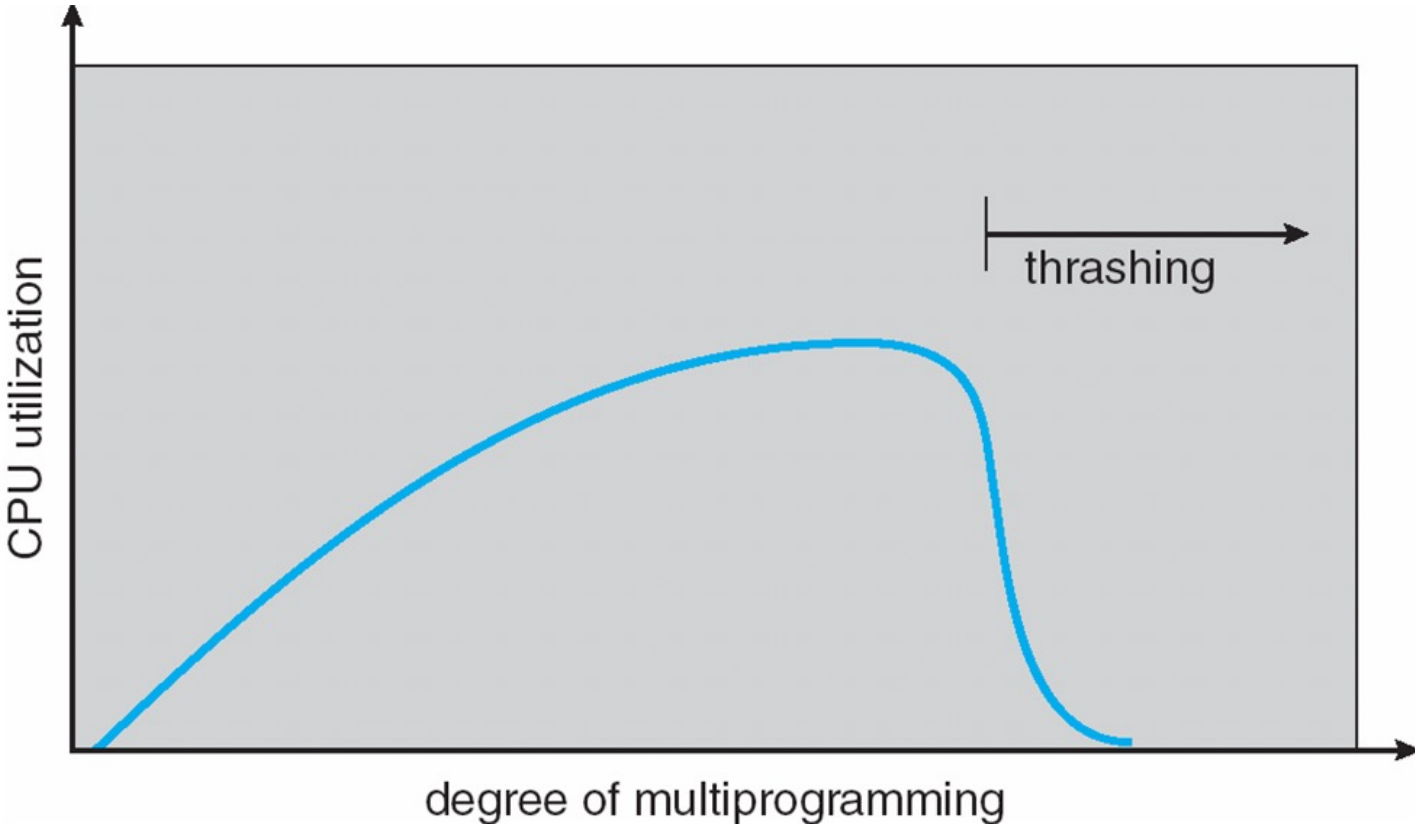
Priority Allocation

- Use a proportional allocation scheme using priorities rather than size
- If process P_i generates a page fault,
  - select for replacement one of its frames
  - select for replacement a frame from a process with lower priority number
- **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another
- **Local replacement** – each process selects from only its own set of allocated frames

Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high. This leads to:
  - low CPU utilization
  - operating system thinks that it needs to increase the degree of multiprogramming
  - another process added to the system
- **Thrashing** ≡ a process is busy swapping pages in and out

Thrashing (Cont.)

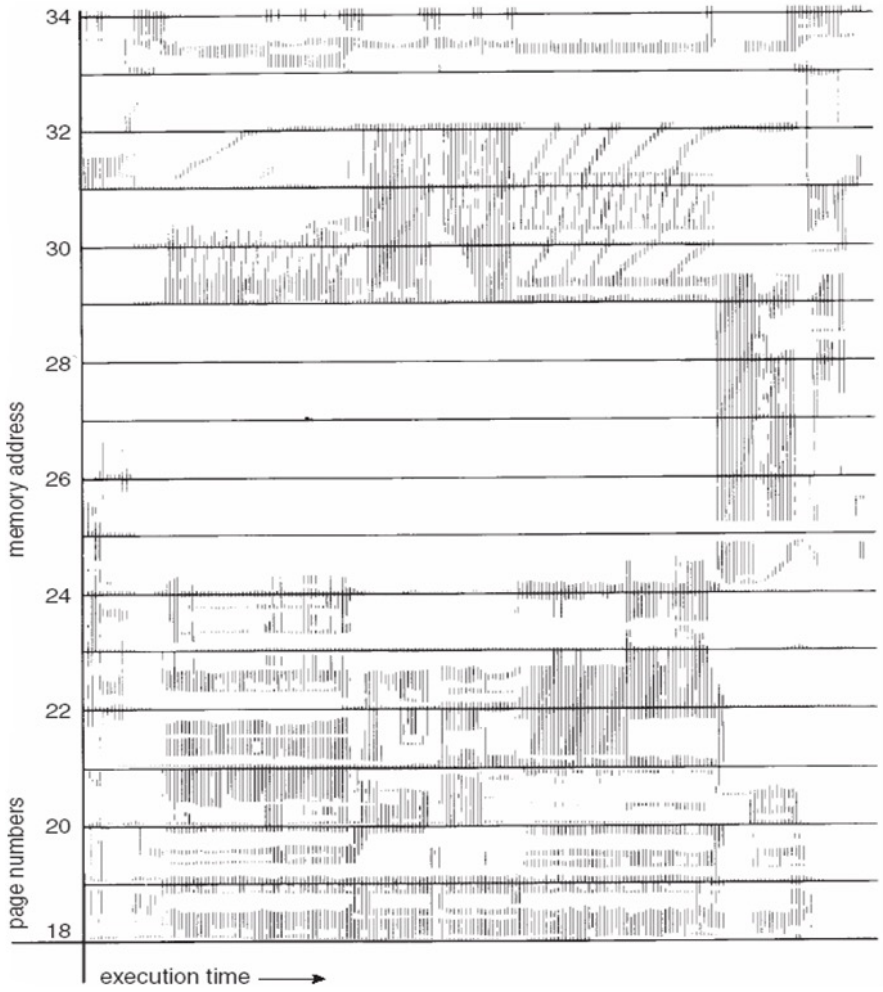


The effective memory-access time increases.
No work is getting done, because the process are spending all their time paging.

Demand Paging and Thrashing

- Why does demand paging work?
  Locality model
  - Process migrates from one locality to another
  - Localities may overlap
  - For example, calling a function need memory references to the instruction of the function call, its local variables and subset of global variables.
- Why does thrashing occur?
  Σ size of locality > total memory size

Locality In A Memory-Reference Pattern

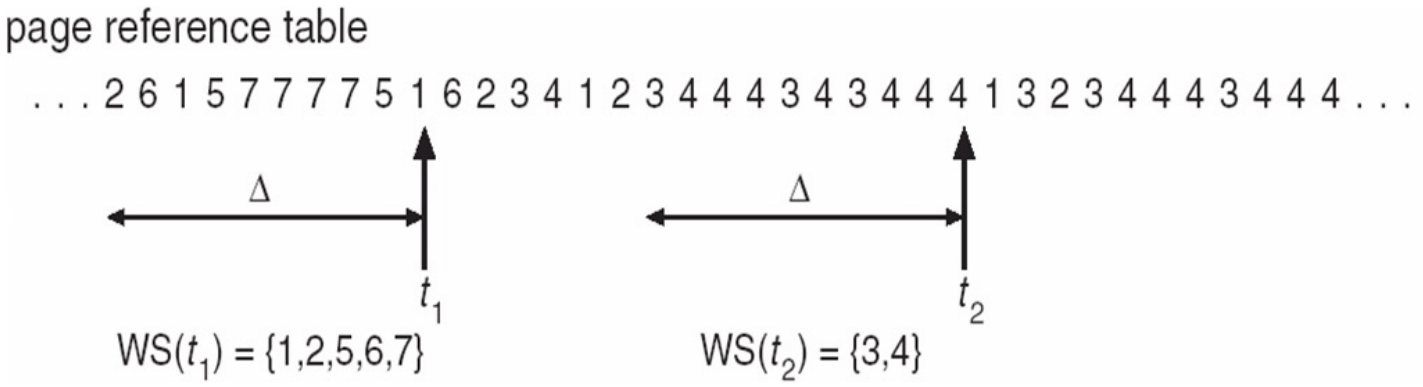


Locality is a set of pages that are actively used together.
Localities are defined by the program structure and its data structure.

Working-Set Model

- The working set model examines the most recent Δ page references.
- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  Example: 10,000 instruction
- 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
- if $D > m \Rightarrow$ Thrashing, m is total available frames
- **Policy:** if $D >$ m, then suspend one of the processes

Working-set model

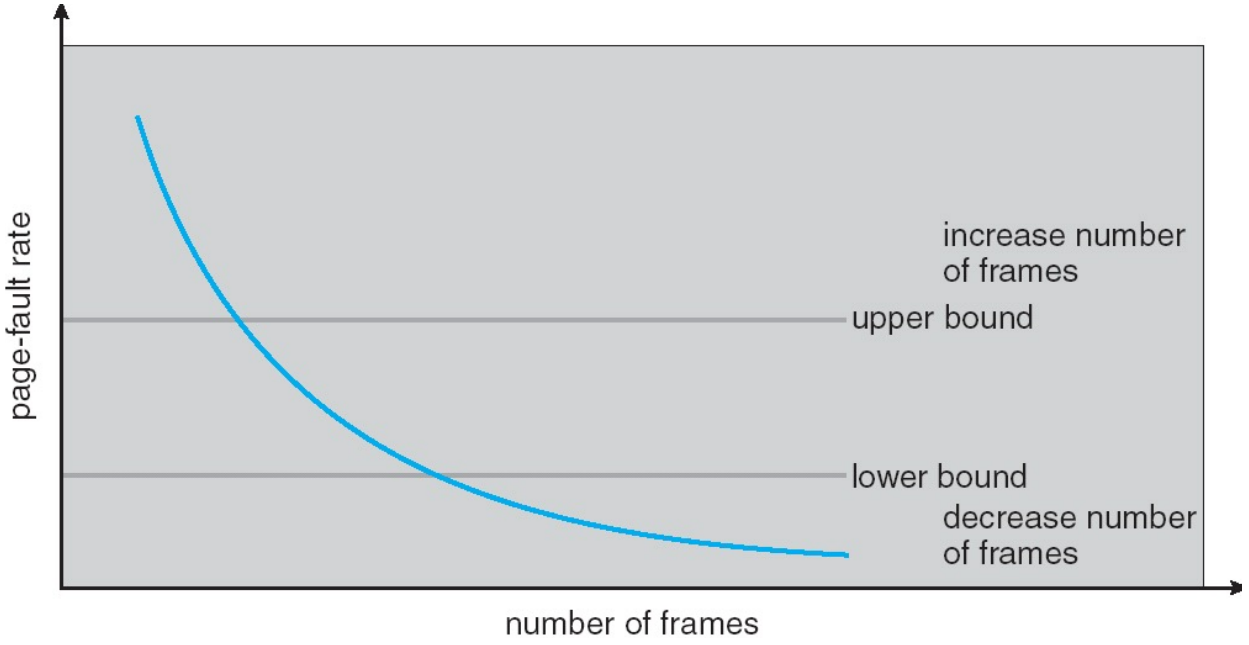


Keeping Track of the Working Set

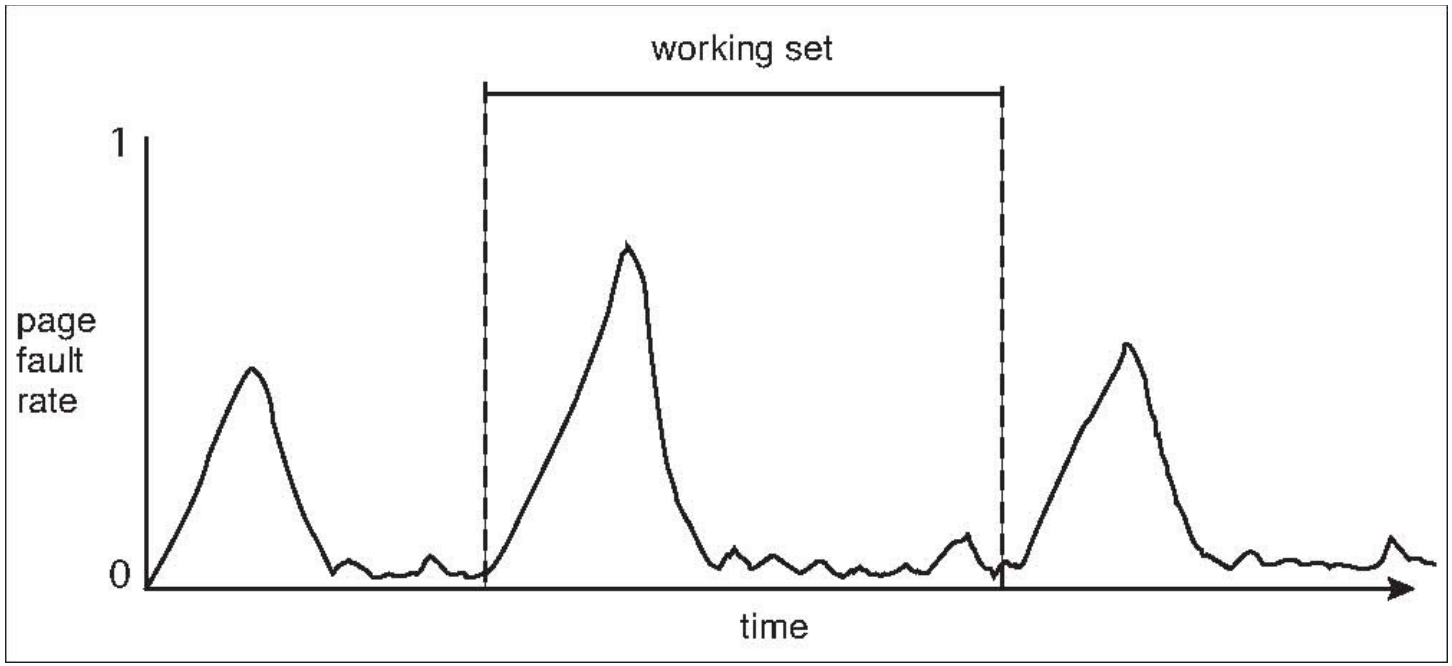
- Approximate with interval timer + a reference bit
- Example: Δ = 10,000
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 bits for each page
  - Whenever a timer interrupts copy and sets the values of all reference bits to 0
  - If one of the bits in memory = 1 $\Rightarrow$ page in working set
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units
- **OS monitors the working set of each process. If the sum exceeds the total available frames, OS selects a process to suspend.**

Page-Fault Frequency Scheme

- Establish “acceptable” page-fault rate
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame



Working Sets and Page Fault Rates



The page fault rate of the process will transition between peaks and valleys over time.

When we begin demand-paging a new locality, a peak occurs.

When the working set of this new locality is in memory, the page-fault falls.

When it moves to new locality, the page-fault rises again.

End of Chapter 9