Chapter 9: Virtual-Memory Management

Chapter 9: Virtual Memory

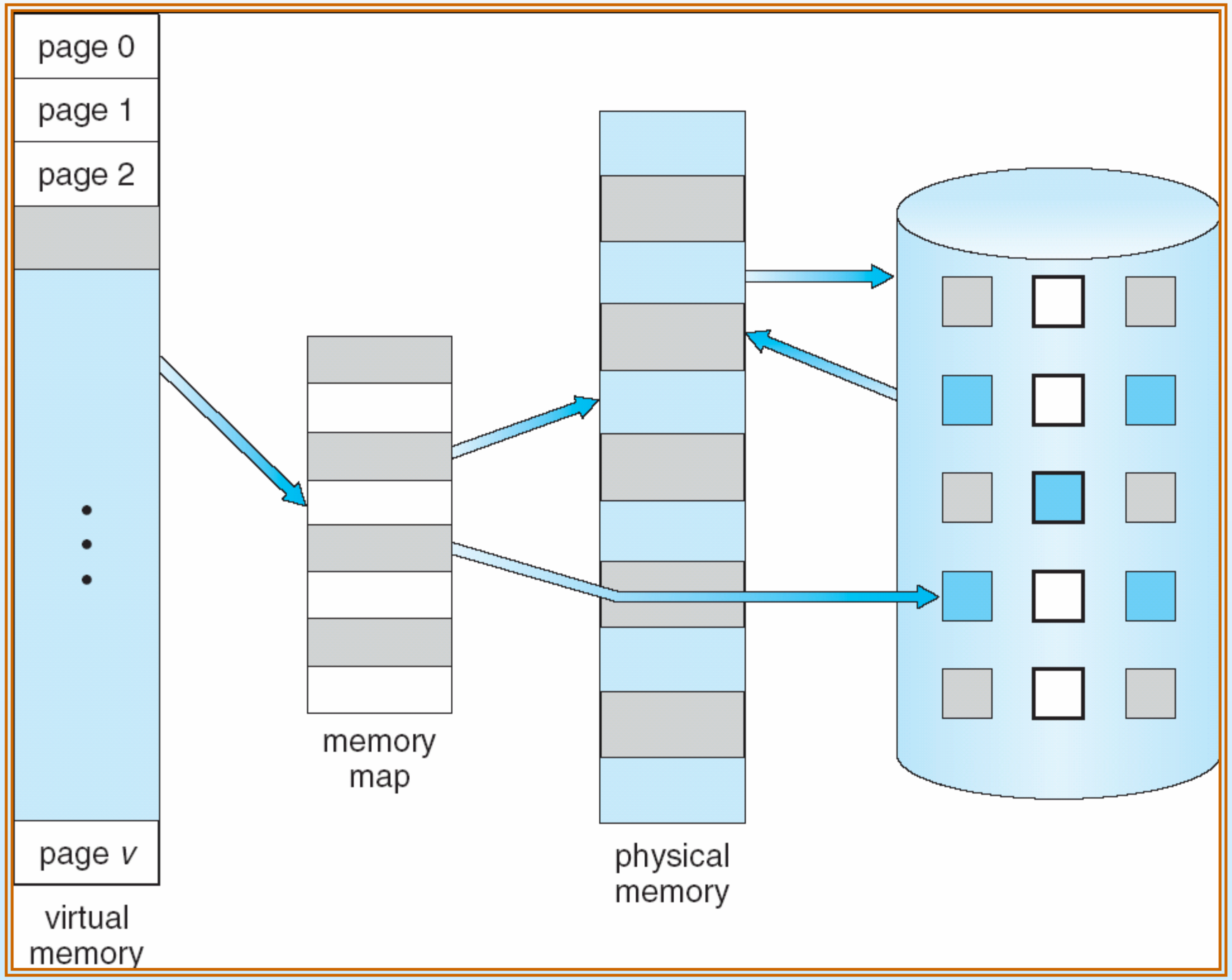
- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped Files
- Allocation Kernel Memory
- Other Consideration

Background

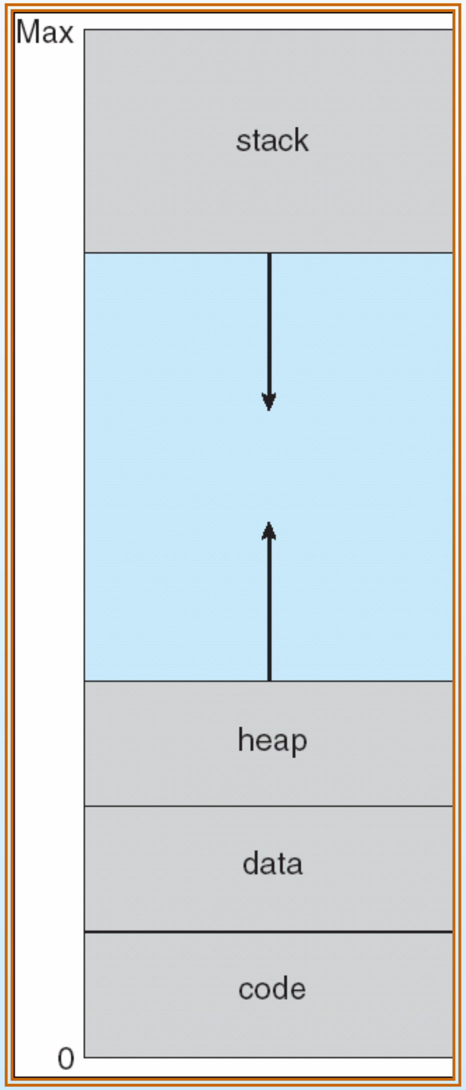
- **Virtual memory** – separation of user logical memory from physical memory.
  - Only part of the program needs to be in memory for execution.
  - Logical address space can therefore be much larger than physical address space.
  - Allows address spaces to be shared by several processes.
  - Allows for more efficient process creation.

- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

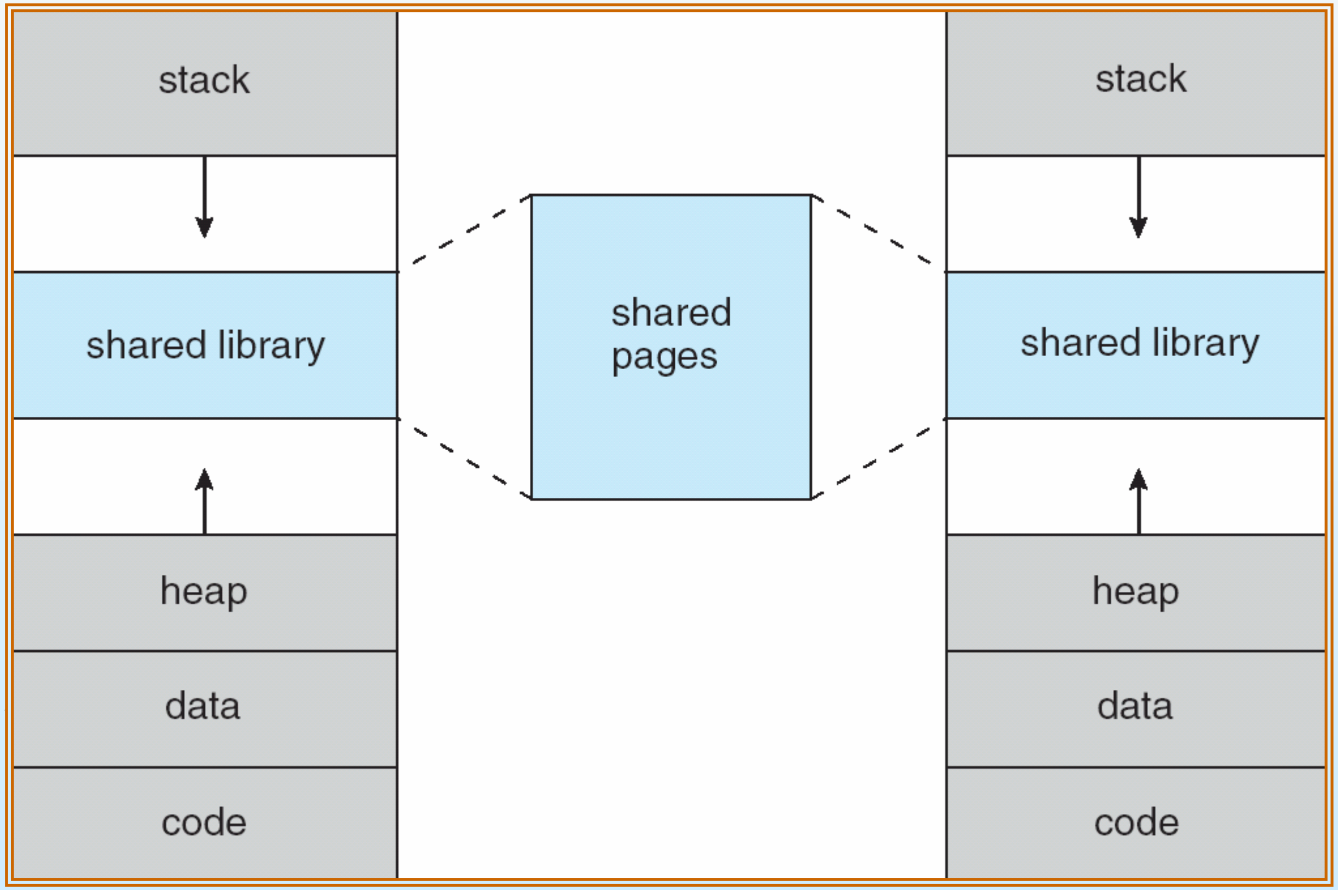
Virtual Memory That is Larger Than Physical Memory



Virtual-address Space



Shared Library Using Virtual Memory

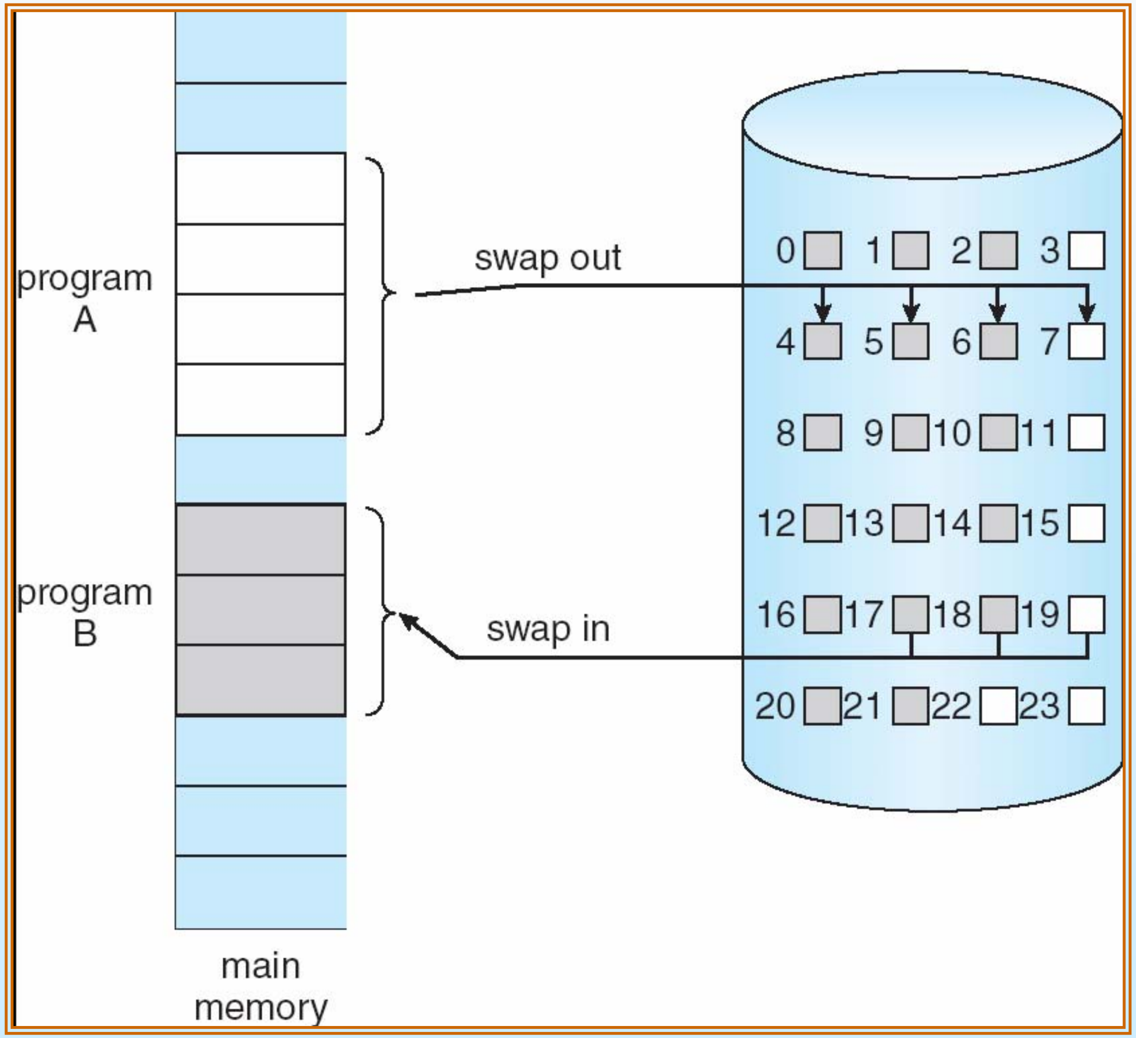


Demand Paging

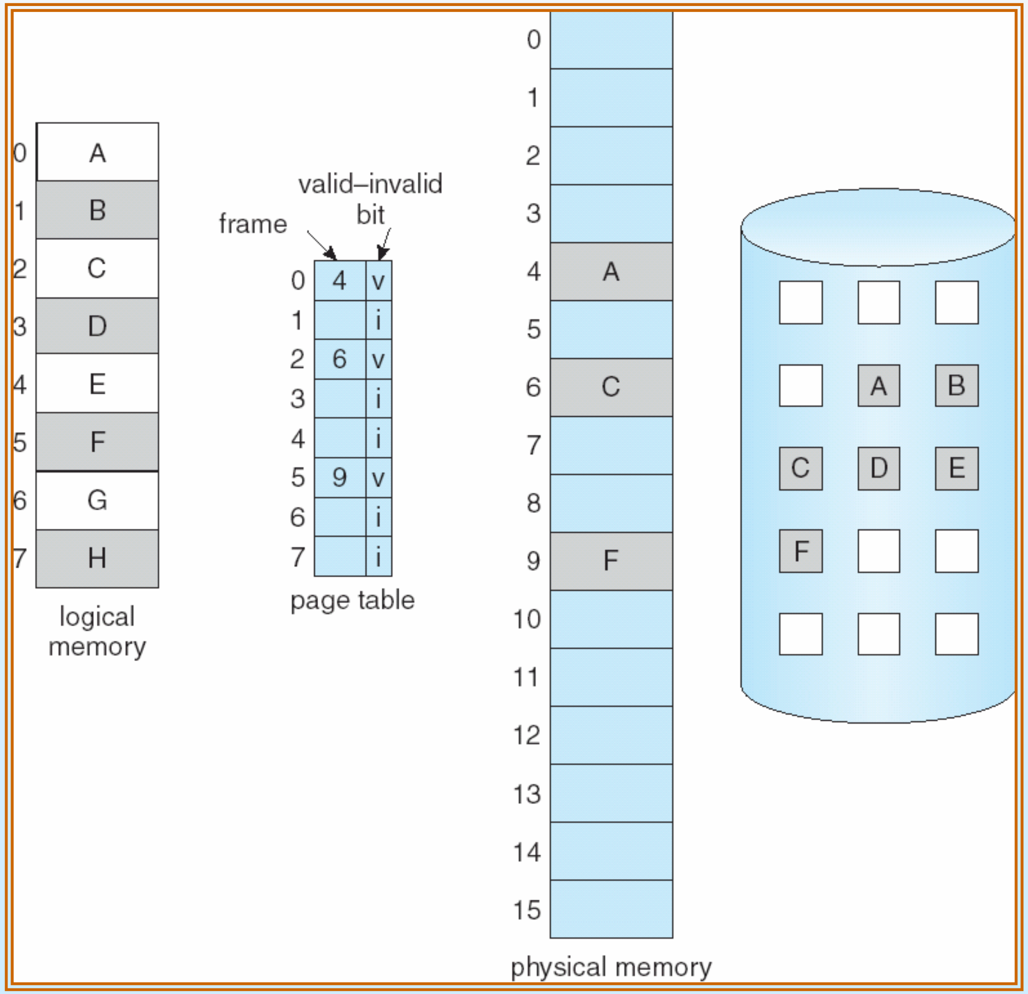
- Bring a page into memory only when it is needed
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users

- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory

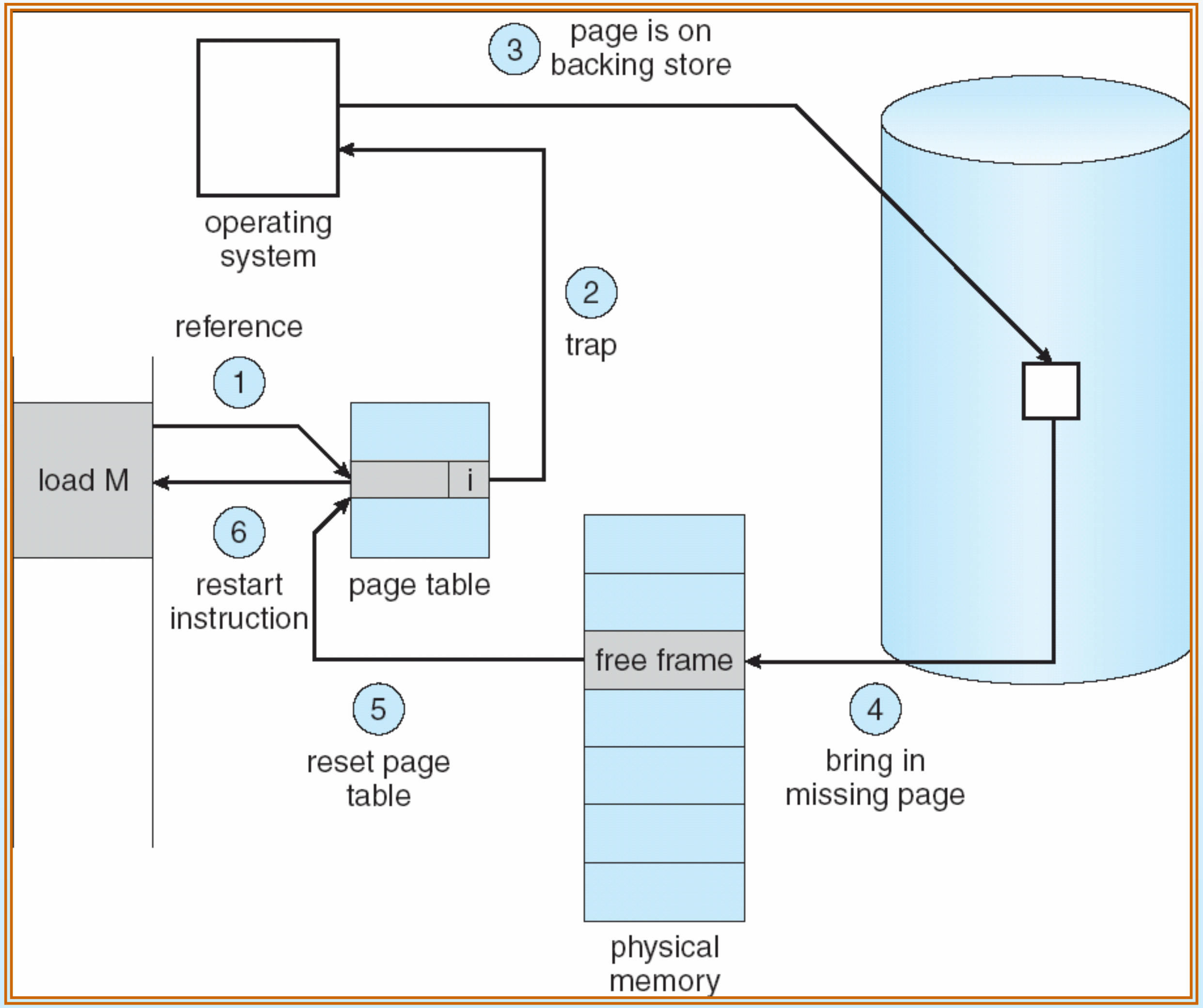
Transfer of a Paged Memory to Contiguous Disk Space



Page Table When Some Pages Are Not in Main Memory



Steps in Handling a Page Fault



A Worst Case Example

- Consider a three-address instruction, ADD the content of A and B and place the result in C
  - Fetch and decode the instruction (ADD)
  - Fetch A
  - Fetch B
  - Add A and B
  - Store the sum in C

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)

EAT = (1 – p) x memory access
+ p x page fault time

For example, page fault time = page fault overhead
+ [swap page out]
+ swap page in
+ restart overhead

Example

- Memory access time = 200 ns, page fault service time = 8 ms
- EAT = $(1-p) \times 200 + p \times 8000000 = 200 + 7999800 \times p$
- If $p = 0.1\%$, EAT = 8.2 μs, slowed down by a factor of 40
- If we want performance degradation to be less than 10%, we need

$$220 > 200 + 7999800 \times p$$

$$20 > 7999900 \times p$$

$$p < 0.0000025$$

- Fewer than one memory access out of 399990 to page fault

Copy-on-Write

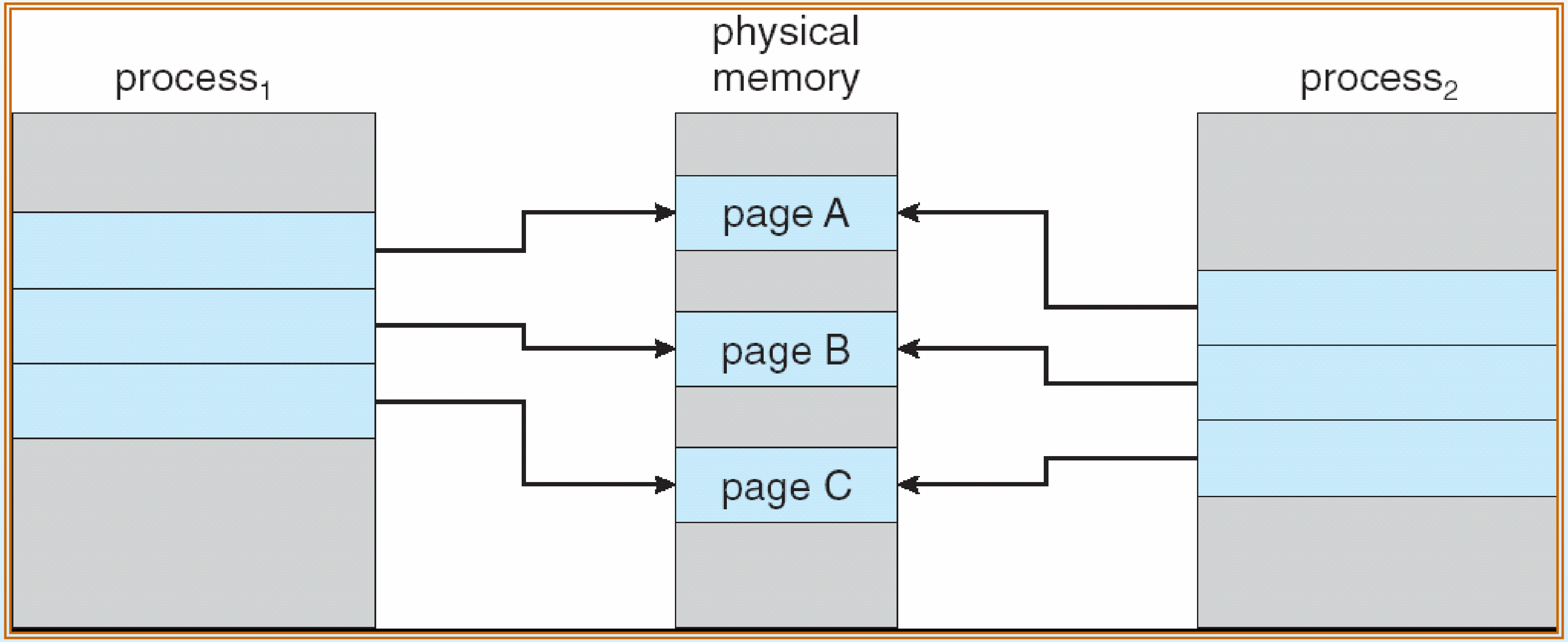
- Copy-on-Write (COW) allows both parent and child processes to initially *share* the same pages in memory

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

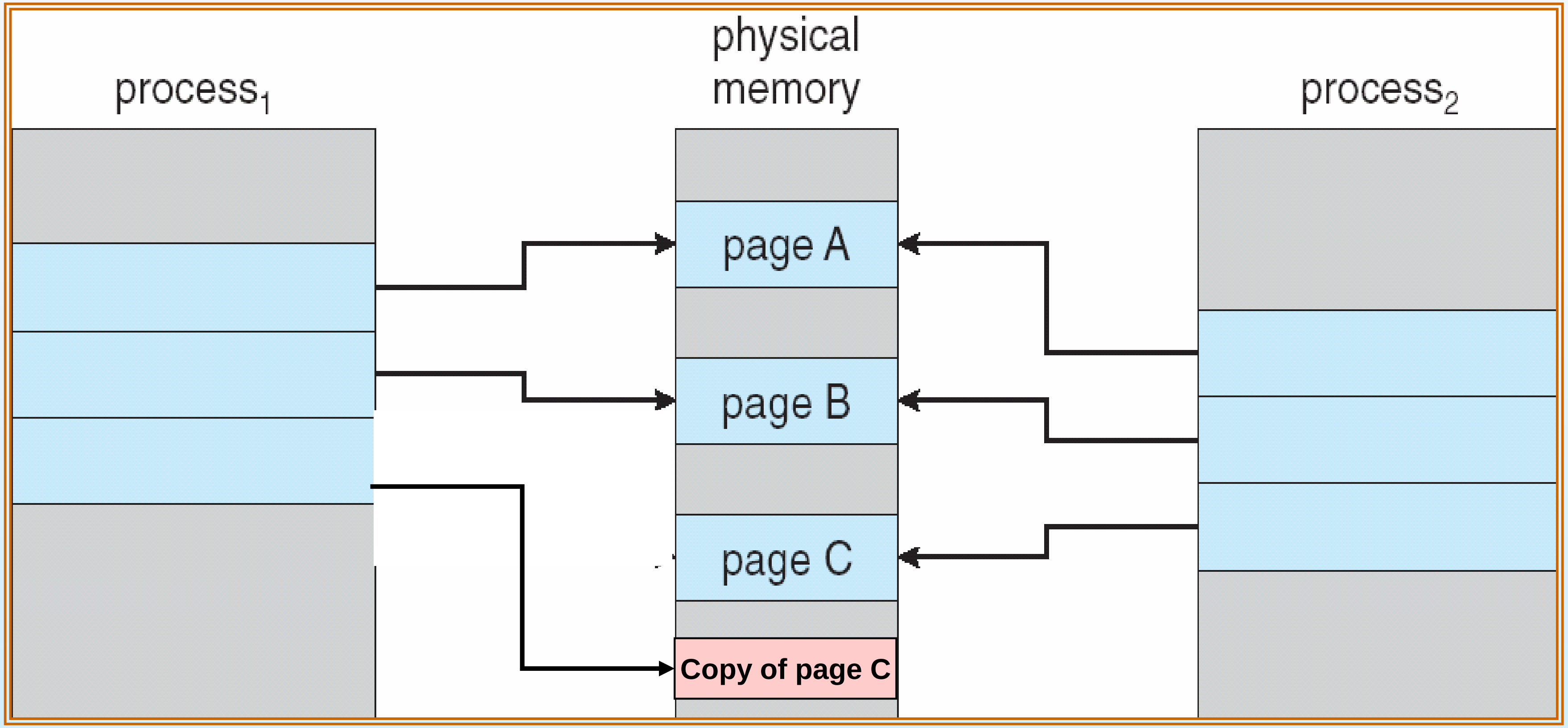
- COW allows more efficient process creation as only modified pages are copied

- Free pages are allocated from a **pool** of zeroed-out pages

Before Process 1 Modifies Page C



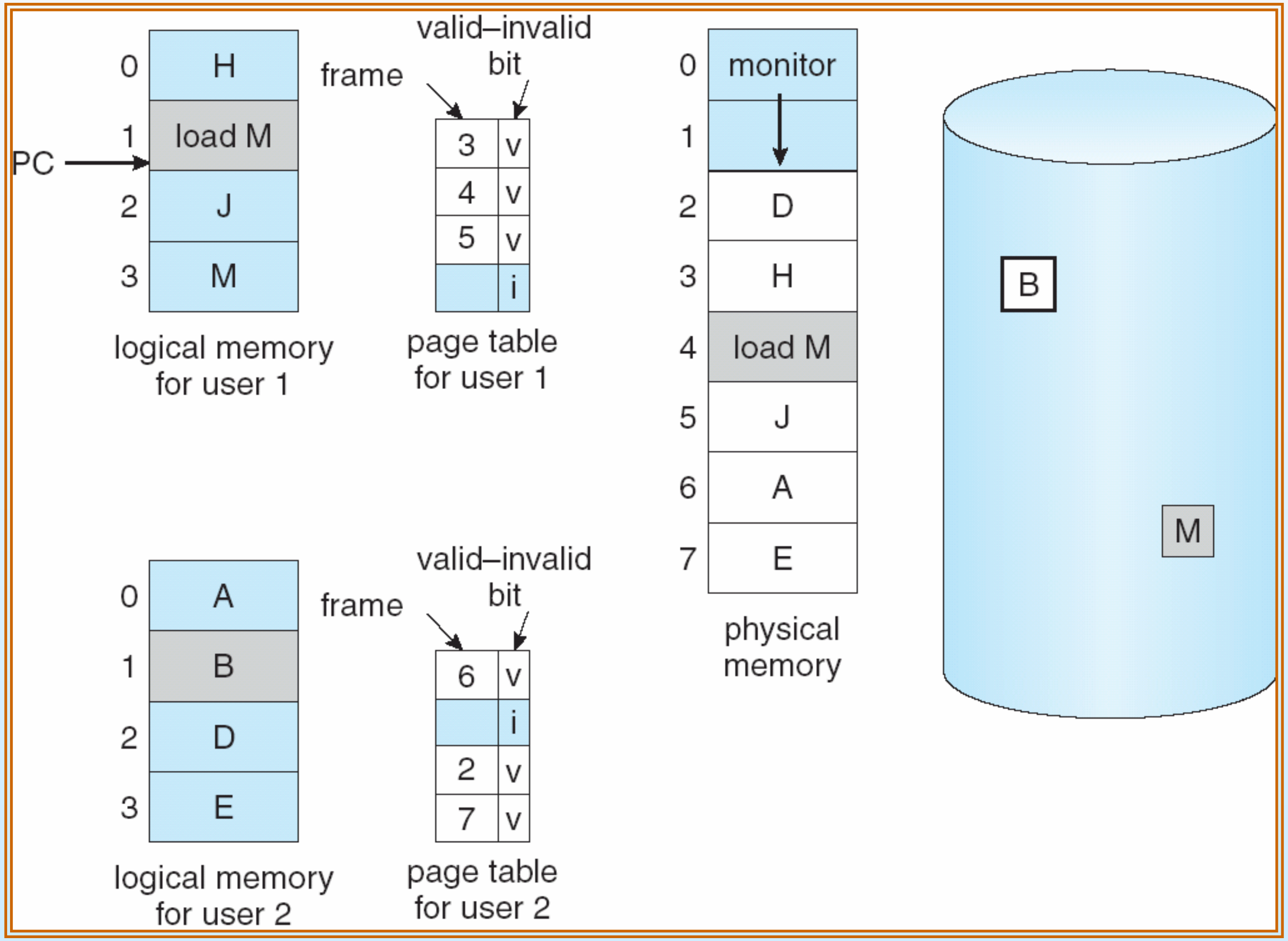
After Process 1 Modifies Page C



Page Replacement

- Prevent over-allocation of memory by modifying page-fault service routine to include page replacement
- Use **modify (dirty) bit** to reduce overhead of page transfers – only modified pages are written to disk
- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory

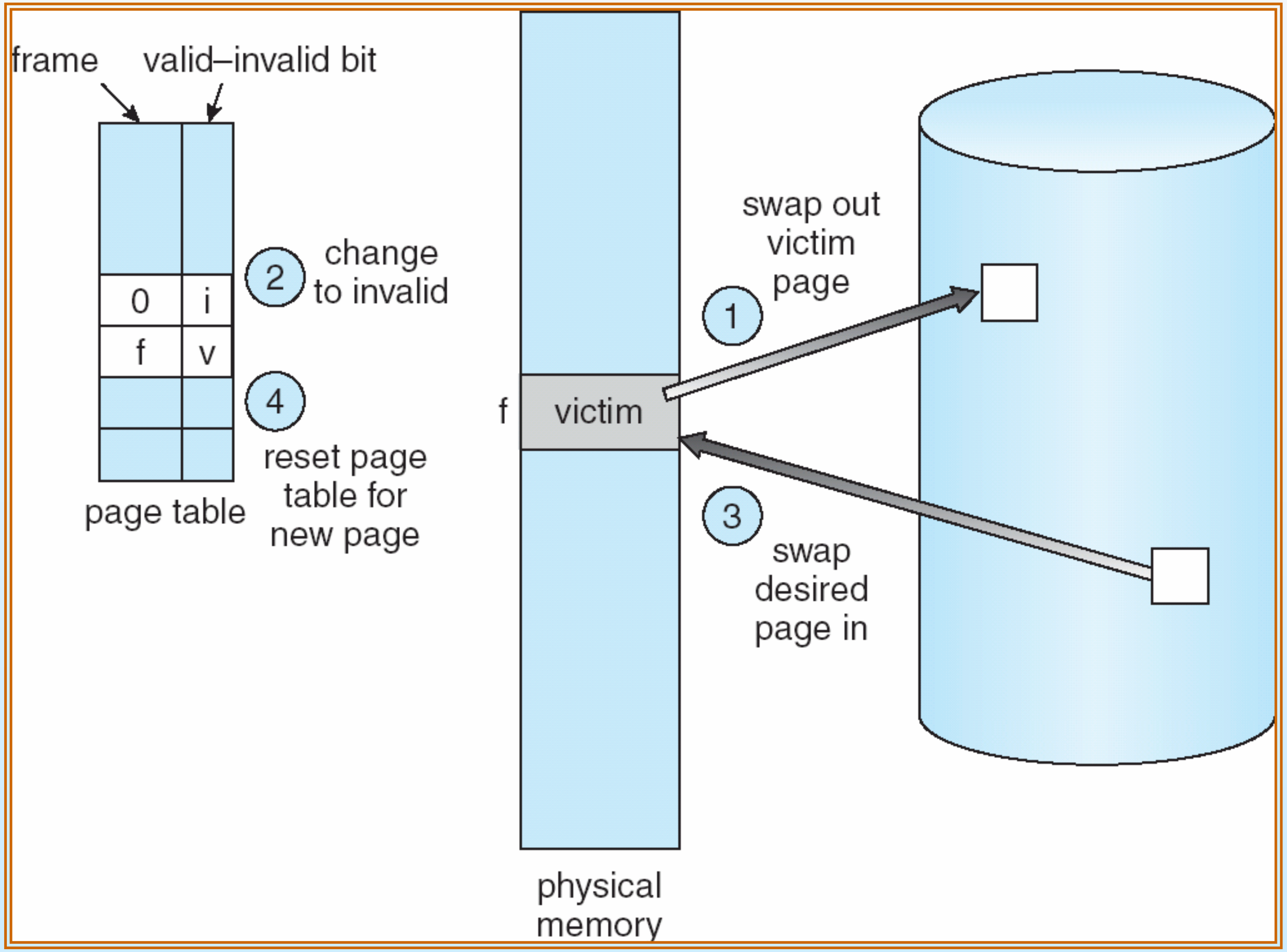
Need For Page Replacement



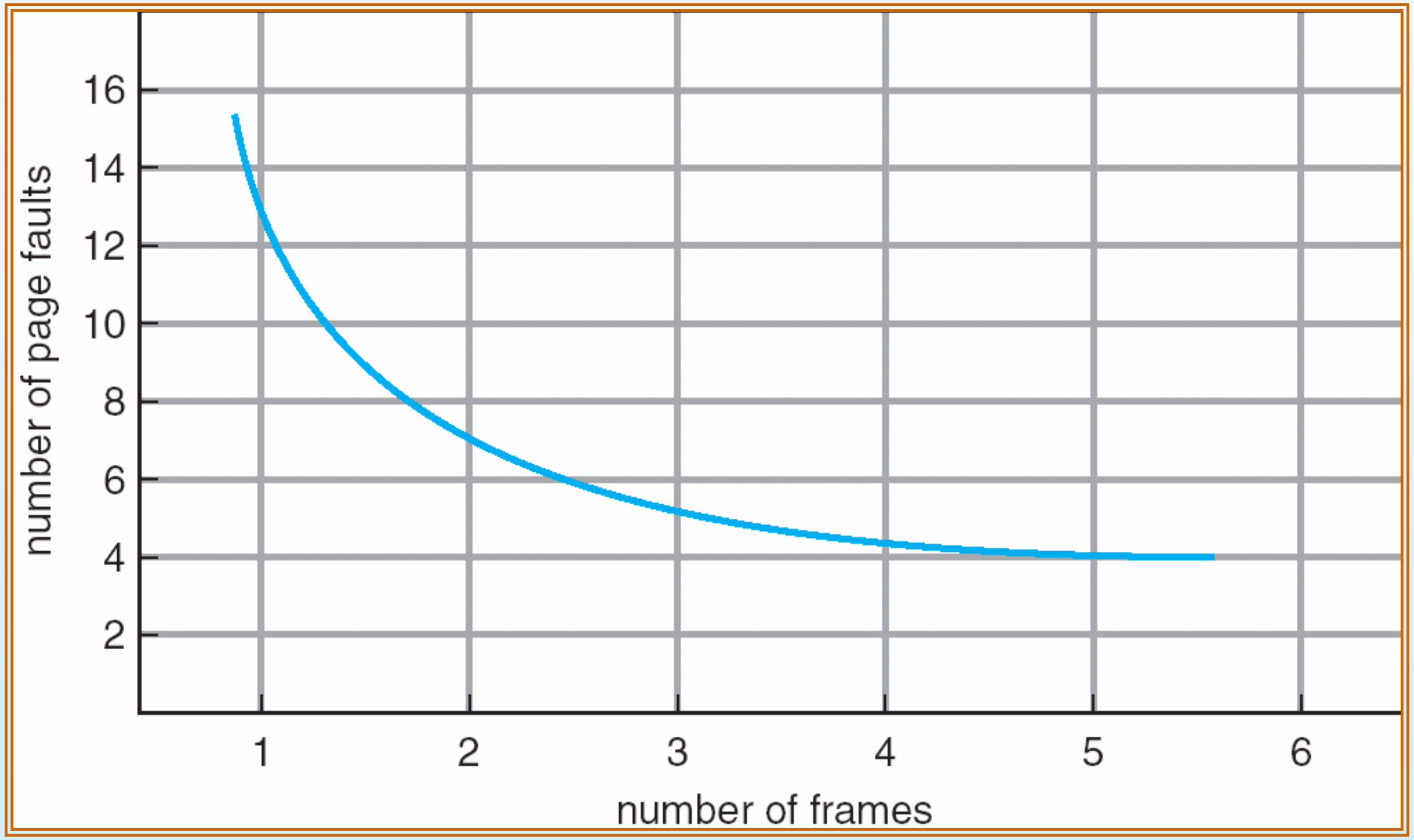
Basic Page Replacement

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. Read the desired page into the (newly) free frame. Update the page and frame tables.
4. Restart the process

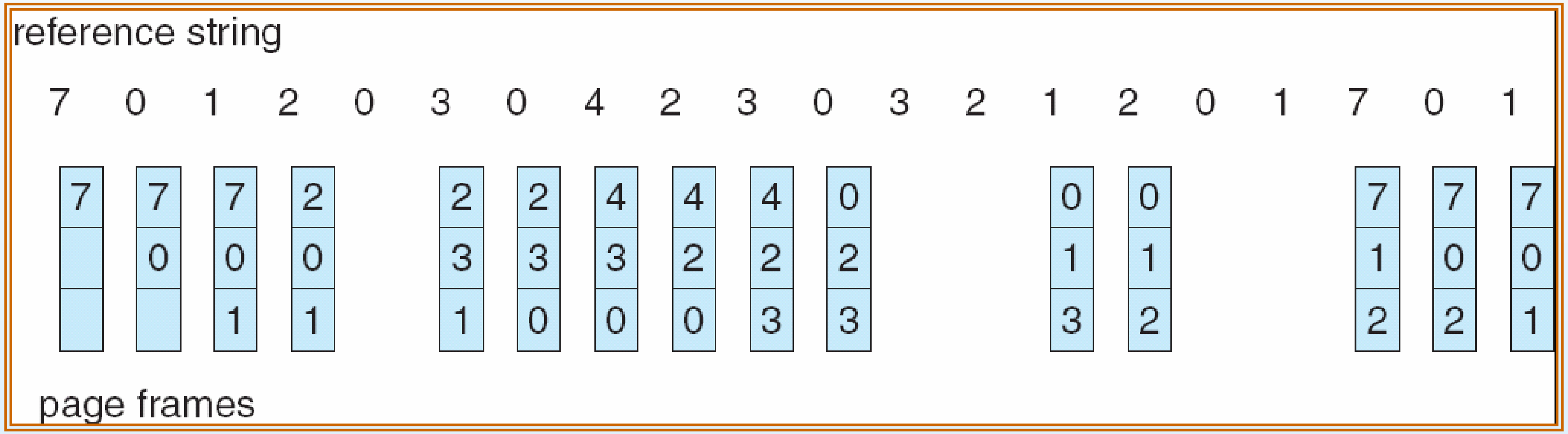
Page Replacement



Graph of Page Faults Versus The Number of Frames



FIFO Page Replacement



reference string

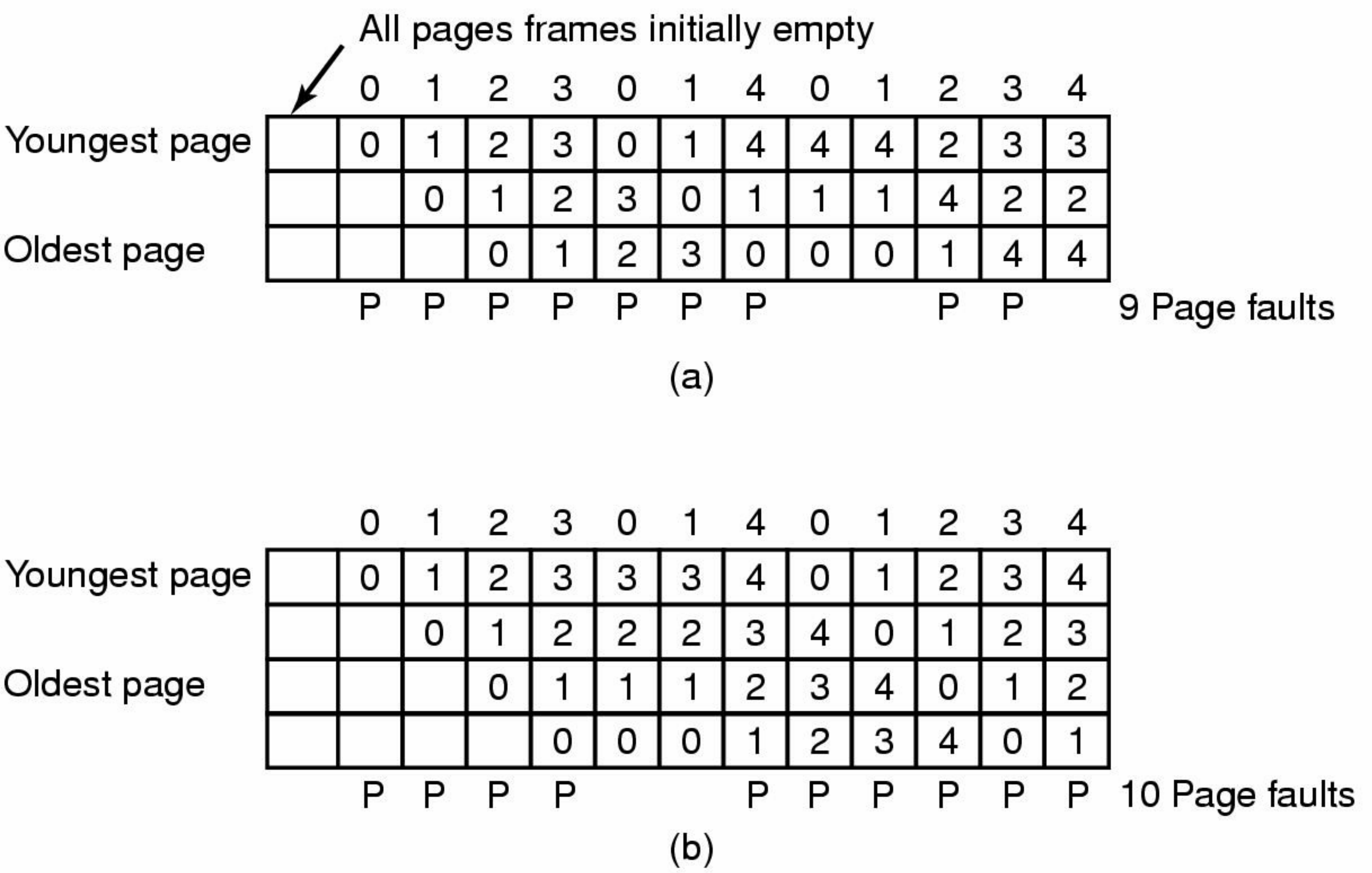
7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2		2	2	4	4	4	0			0	0			7	7	7
	0	0	0		3	3	3	2	2	2			1	1			1	0	0
		1	1		1	0	0	0	3	3			3	2			2	2	1

page frames

Belady's Anomaly

- For some page-replacement algorithm, the page-fault rate may increase as the number of allocated frames increases.
- For reference string 0, 1, 2, 3, 0, 1, 4, 0, 1, 2, 3, 4
  - FIFO demonstrates Belady's anomaly.
  - The number of faults for four frames is greater than three frames.

FIFO Illustrating Belady's Anomaly



All pages frames initially empty

		0	1	2	3	0	1	4	0	1	2	3	4	
Youngest page		0	1	2	3	0	1	4	4	4	2	3	3	
			0	1	2	3	0	1	1	1	4	2	2	
Oldest page				0	1	2	3	0	0	0	1	4	4	
		P	P	P	P	P	P	P			P	P		9 Page faults

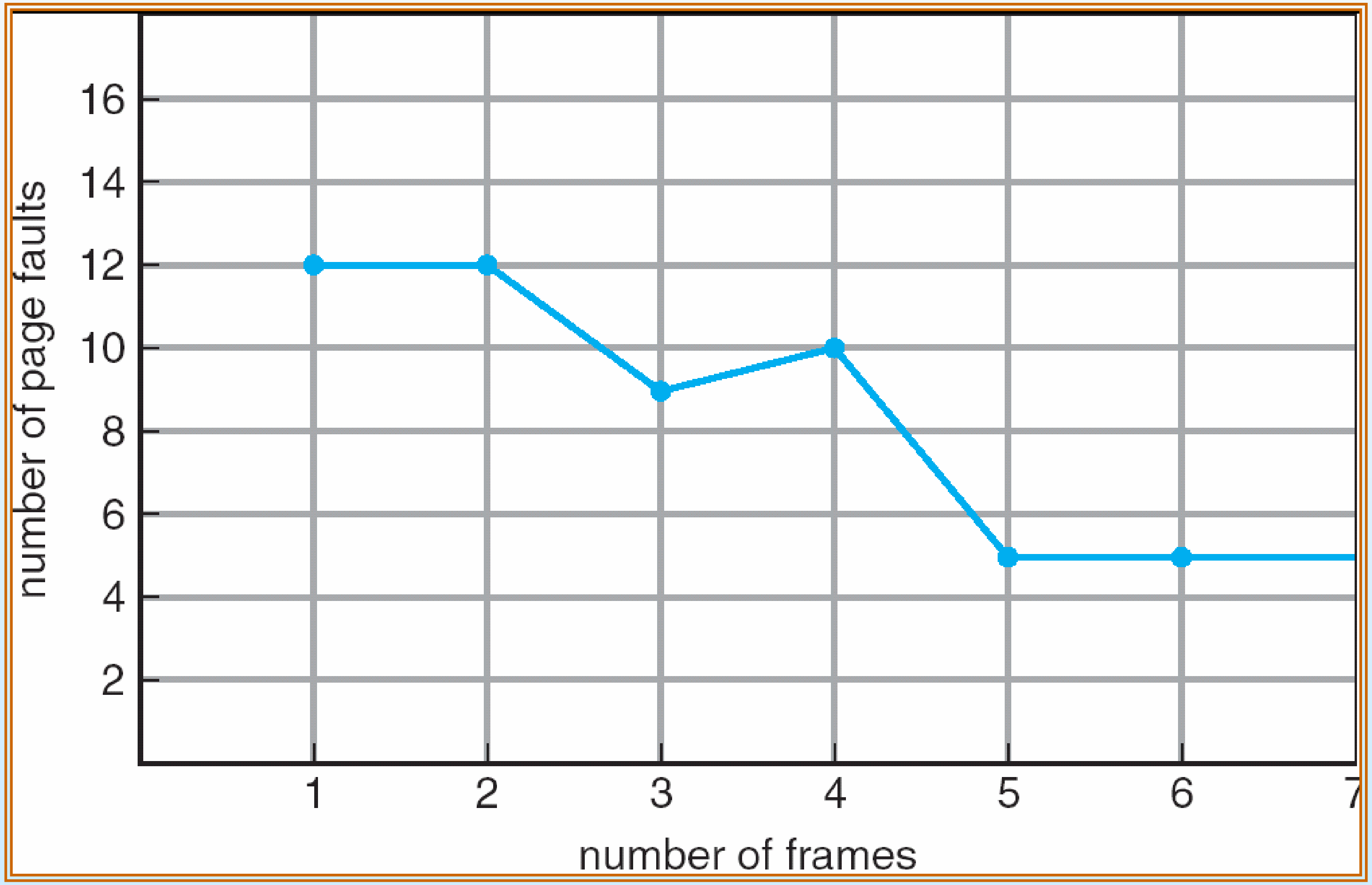
(a)

		0	1	2	3	0	1	4	0	1	2	3	4	
Youngest page		0	1	2	3	3	3	4	0	1	2	3	4	
			0	1	2	2	2	3	4	0	1	2	3	
Oldest page				0	1	1	1	2	3	4	0	1	2	
					0	0	0	1	2	3	4	0	1	
		P	P	P	P			P	P	P	P	P	P	10 Page faults

(b)

- FIFO with 3 page frames
- FIFO with 4 page frames
- *P*'s show which page references show page faults

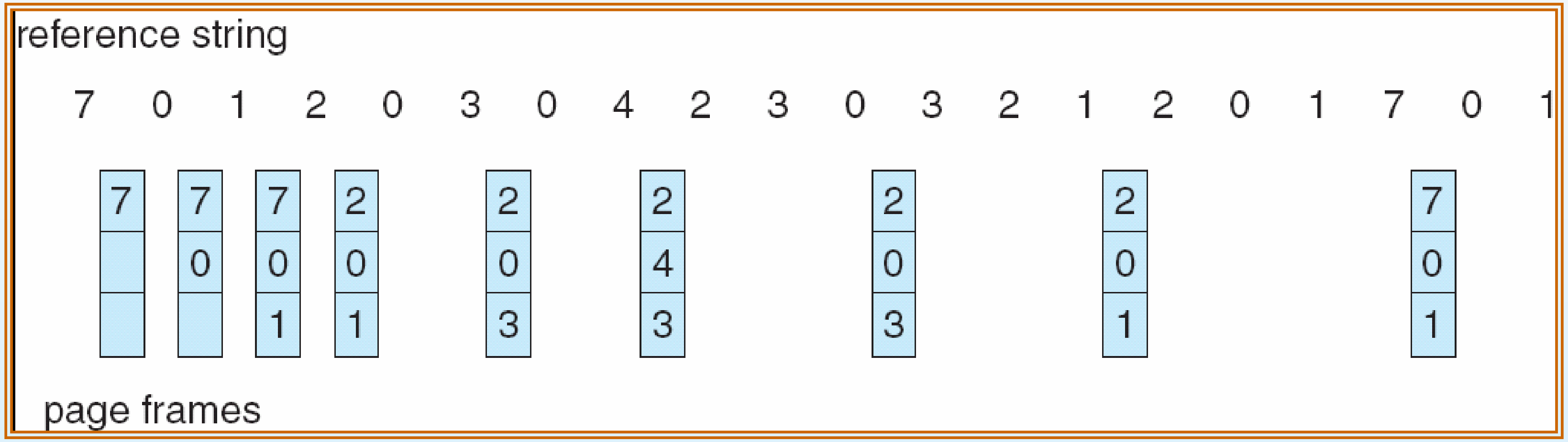
FIFO Illustrating Belady's Anomaly



Optimal Page Replacement

- Replace the page that will not be used for the longest period of time.
- The lowest page-fault rate and no Belady’s anomaly
- Requires future knowledge of page reference

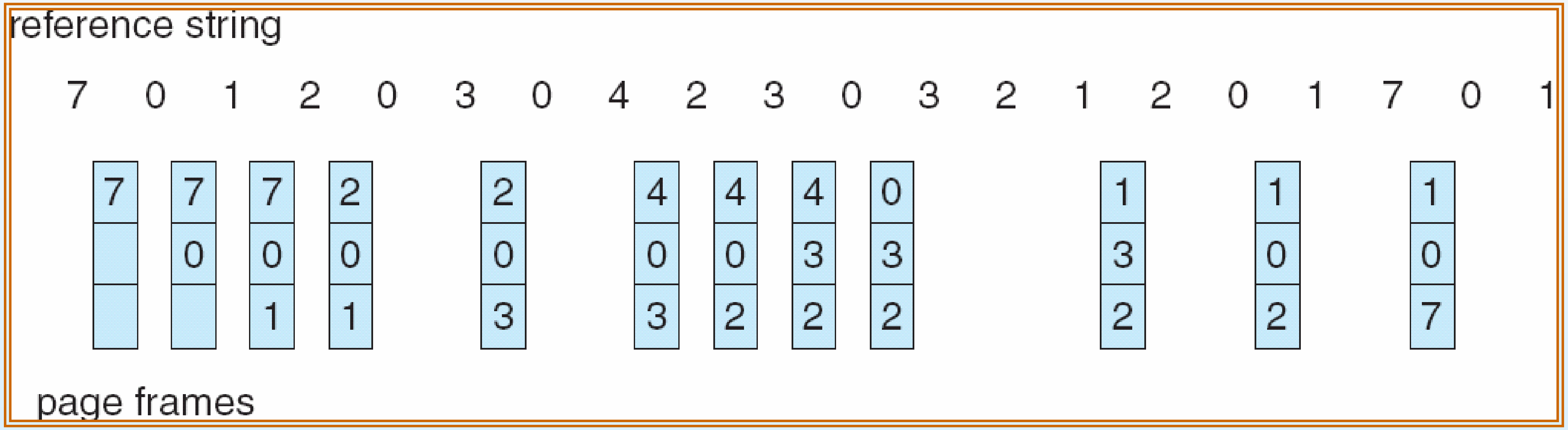
Optimal Page Replacement



LRU Page Replacement

- Time
    - FIFO uses the time when a page was brought into memory
    - OPT uses the time when a page is to be used
    - LRU, least recently used, uses the time that a page has not been used
- LRU: replace the page that has not been used for the longest period of time
    - Like OPT looking backward in time
- Requires hardware support for implementing LRU

LRU Page Replacement



reference string

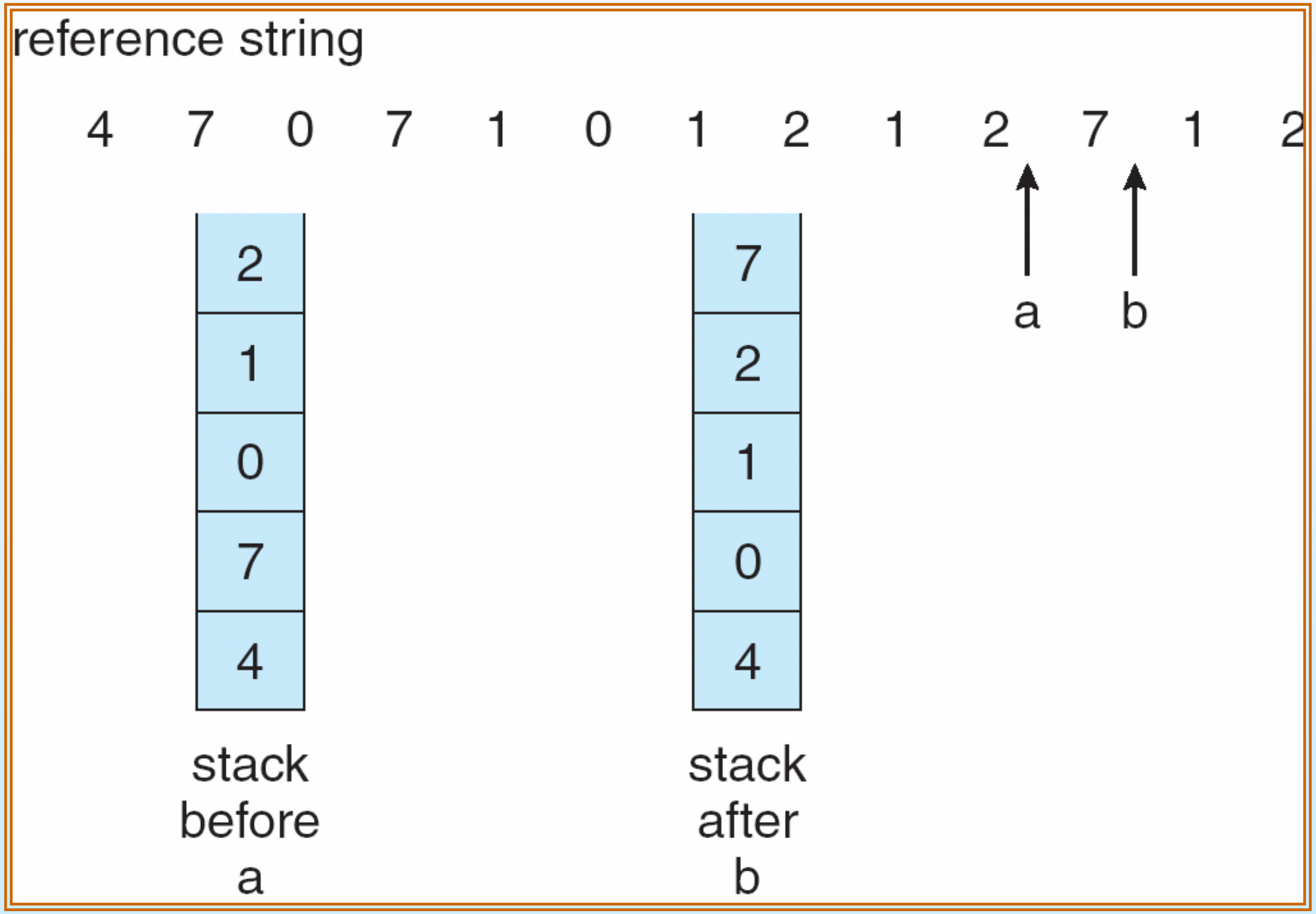
7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2		2		4	4	4	0			1		1		1		
	0	0	0		0		0	0	3	3			3		0		0		
		1	1		3		3	2	2	2			2		2		7		

page frames

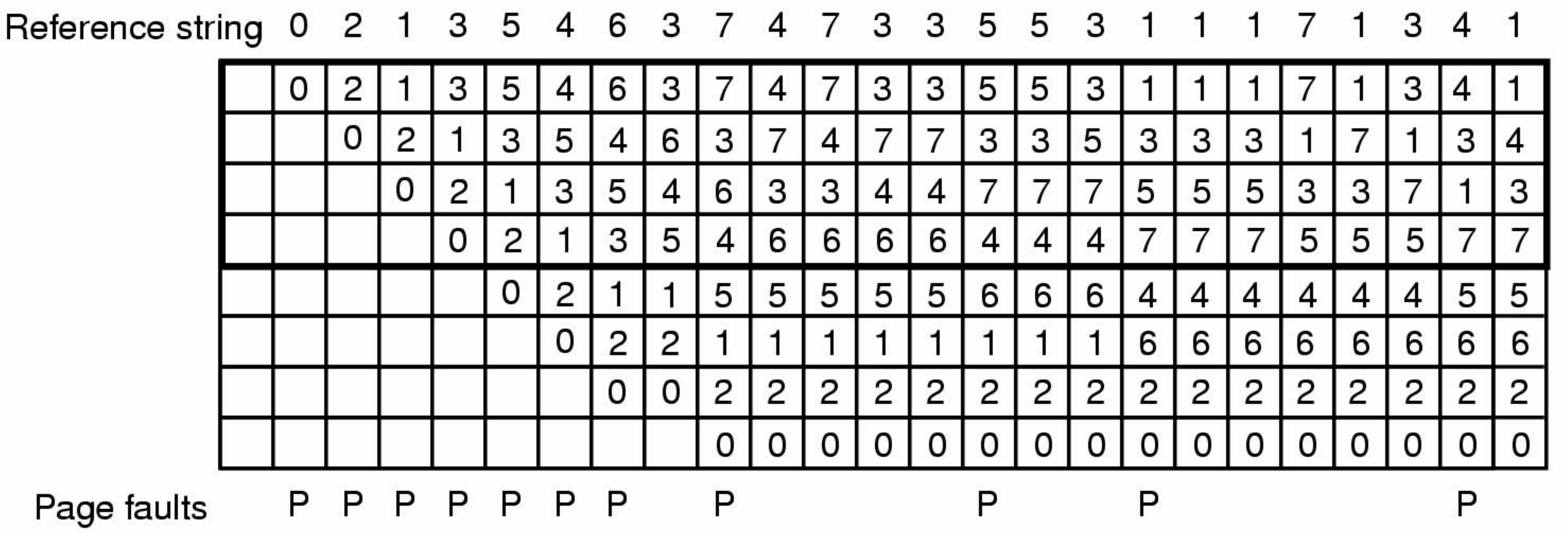
LRU Page Replacement

- Two possible implementation
  - Counters
    - Whenever a page is referenced, the CPU clock counter is copied to the time-of-use field of the page
    - The page with the smallest time value will be replaced
  - Stack
    - Whenever a page is referenced, it is removed from the stack and put on the top
    - The bottom page will be replaced
- Together with OPT, they belong to stack algorithm that never exhibit Belady’s anomaly
  - A stack algorithm is the one that the set of pages in memory of n frames is always a subset of pages in memory of n+1 frames

Use Of A Stack to Record The Most Recent Page References



Stack Algorithms



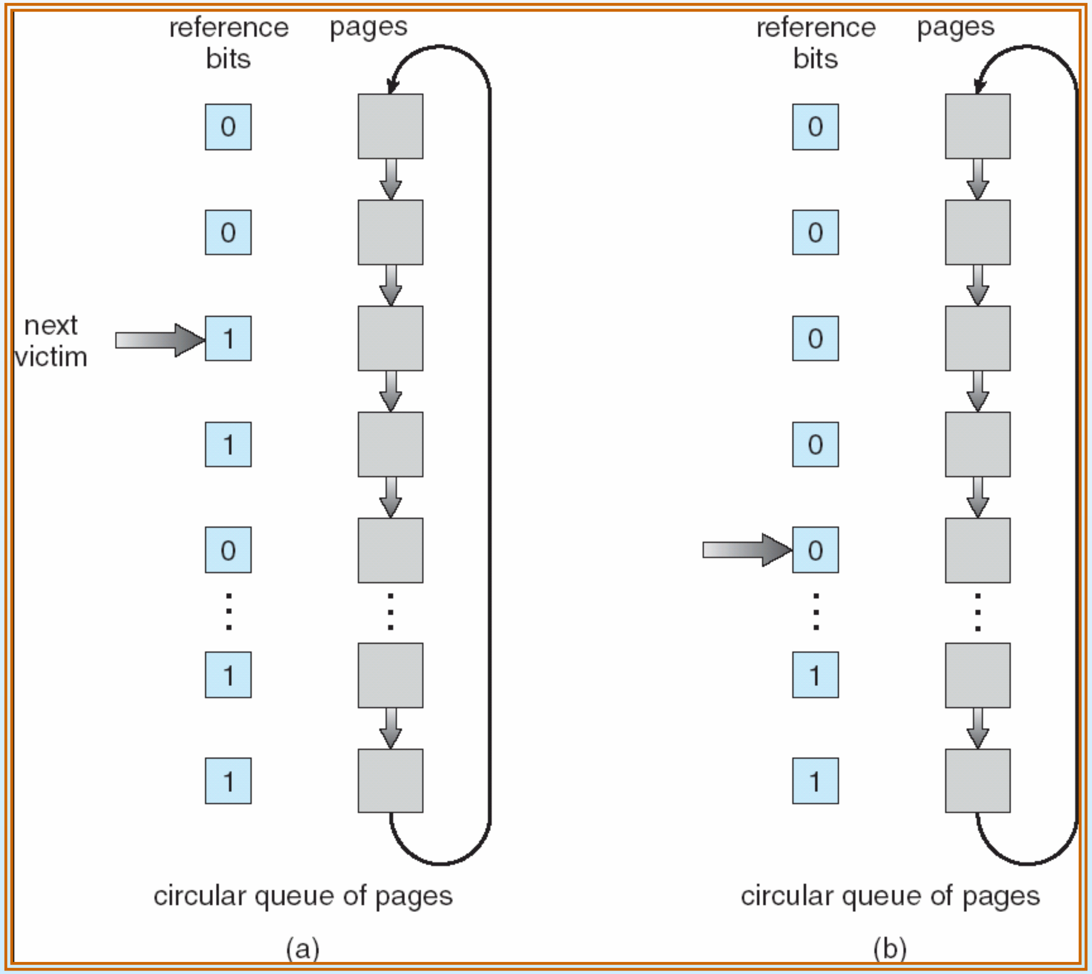
Reference string	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	1	3	4	1
	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	1	3	4	1
		0	2	1	3	5	4	6	3	7	4	7	7	3	3	5	3	3	3	1	7	1	3	4
			0	2	1	3	5	4	6	3	3	4	4	7	7	7	5	5	5	3	3	7	1	3
				0	2	1	3	5	4	6	6	6	6	4	4	4	7	7	7	5	5	5	7	7
					0	2	1	1	5	5	5	5	5	6	6	6	4	4	4	4	4	4	5	5
						0	2	2	1	1	1	1	1	1	1	1	6	6	6	6	6	6	6	6
							0	0	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
									0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Page faults	P	P	P	P	P	P	P		P					P			P						P	

State of memory array, *M*, after each item in reference string is processed

LRU Approximation Algorithms

- Need less hardware support than LRU: reference bit
  - 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.
- Second chance or clock
  - Need reference bit
  - Clock replacement
  - If page to be replaced (in clock order) has reference bit = 1 then:
    - set reference bit 0
    - leave page in memory
    - replace next page (in clock order), subject to same rules

Second-Chance (clock) Page-Replacement Algorithm



Enhanced Second-Chance Algorithm

- Each page has Reference bit, Modified bit
  - bits are set when page is referenced, modified
- Pages are classified
  1. (0, 0): not referenced, not modified
  2. (0, 1): not referenced, modified
  3. (1, 0): referenced, not modified
  4. (1, 1): referenced, modified
- Scan the circular queue and replace the first page from lowest numbered non empty class
  - May scan the circular queue several times

Counting Algorithms

- Keep a counter of the number of references that have been made to each page
  - **LFU Algorithm**: replaces page with smallest count
  - **MFU Algorithm**: based on the argument that the page with the smallest count was probably just brought in and has yet to be used
- Neither MFU or LFU is common
  - The implementation is expensive
  - Do not approximate OPT well

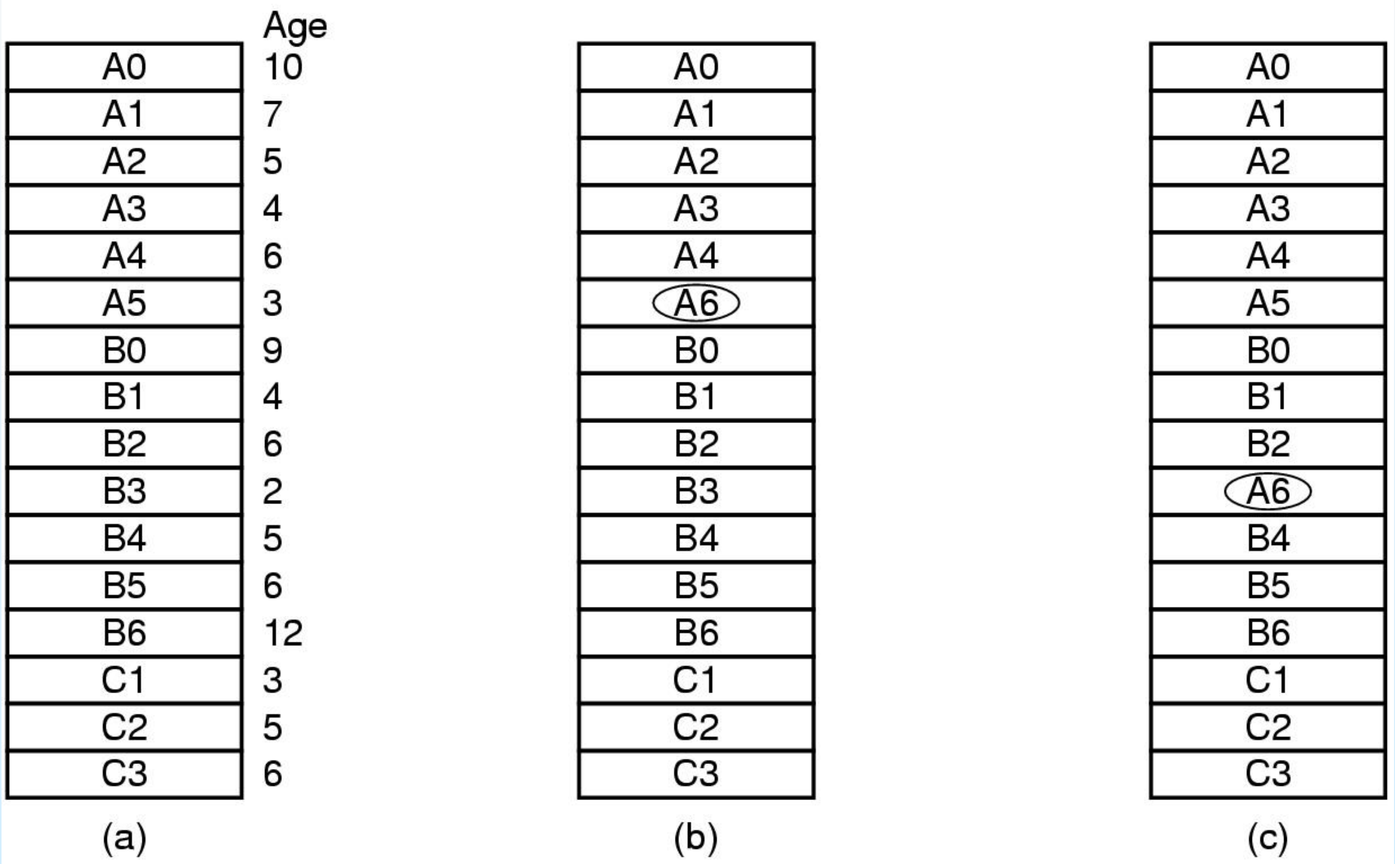
Allocation of Frames

- How to allocate the fixed amount of free memory among various processes?
- 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
    - Equal size or proportional to process size
  - priority allocation
    - High-priority processes may have more memory

Local versus Global Allocation

- Local
  - Each process can select from its own set of allocated frames
- Global
  - A process can select replacement frame from all frames, even it is currently allocated to other process.
  - Has better throughput
  - Performance of a process depends not only on the paging behavior of that process but also on other processes.
    - A process cannot control its own page-fault rate.

Local versus Global Allocation Example



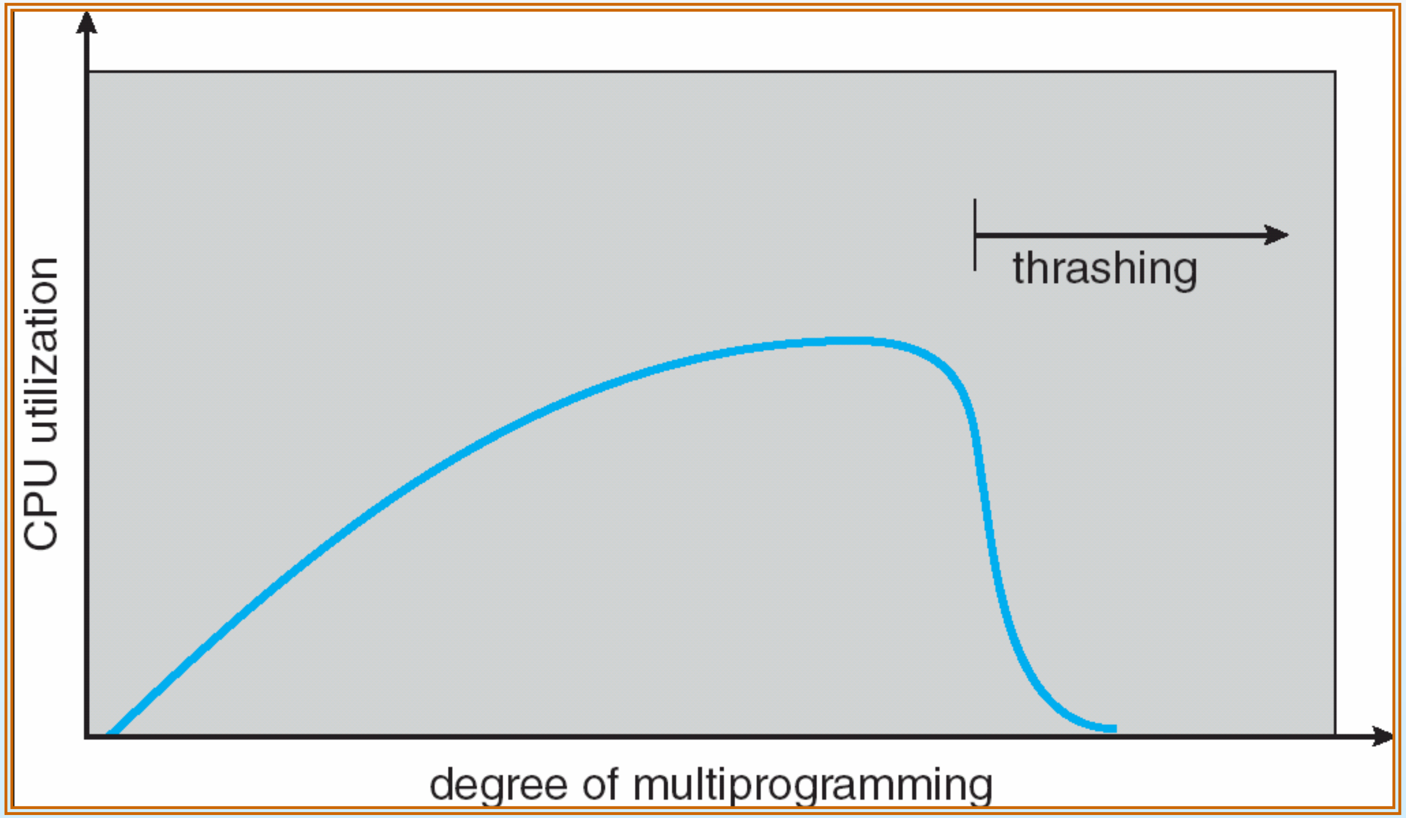
(a)	Age	(b)	(c)
A0	10	A0	A0
A1	7	A1	A1
A2	5	A2	A2
A3	4	A3	A3
A4	6	A4	A4
A5	3	A6	A5
B0	9	B0	B0
B1	4	B1	B1
B2	6	B2	B2
B3	2	B3	A6
B4	5	B4	B4
B5	6	B5	B5
B6	12	B6	B6
C1	3	C1	C1
C2	5	C2	C2
C3	6	C3	C3

- Original configuration
- Local page replacement
- Global page replacement

Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high. Consider the following scenario with global allocation:
  - low CPU utilization
  - operating system thinks that it needs to increase the degree of multiprogramming
  - another process added to the system
  - at some point, each process does not have enough frames and page fault rate is high which cause CPU wait for paging devices
    - CPU utilization
- **Thrashing** ≡ a process is busy swapping pages in and out

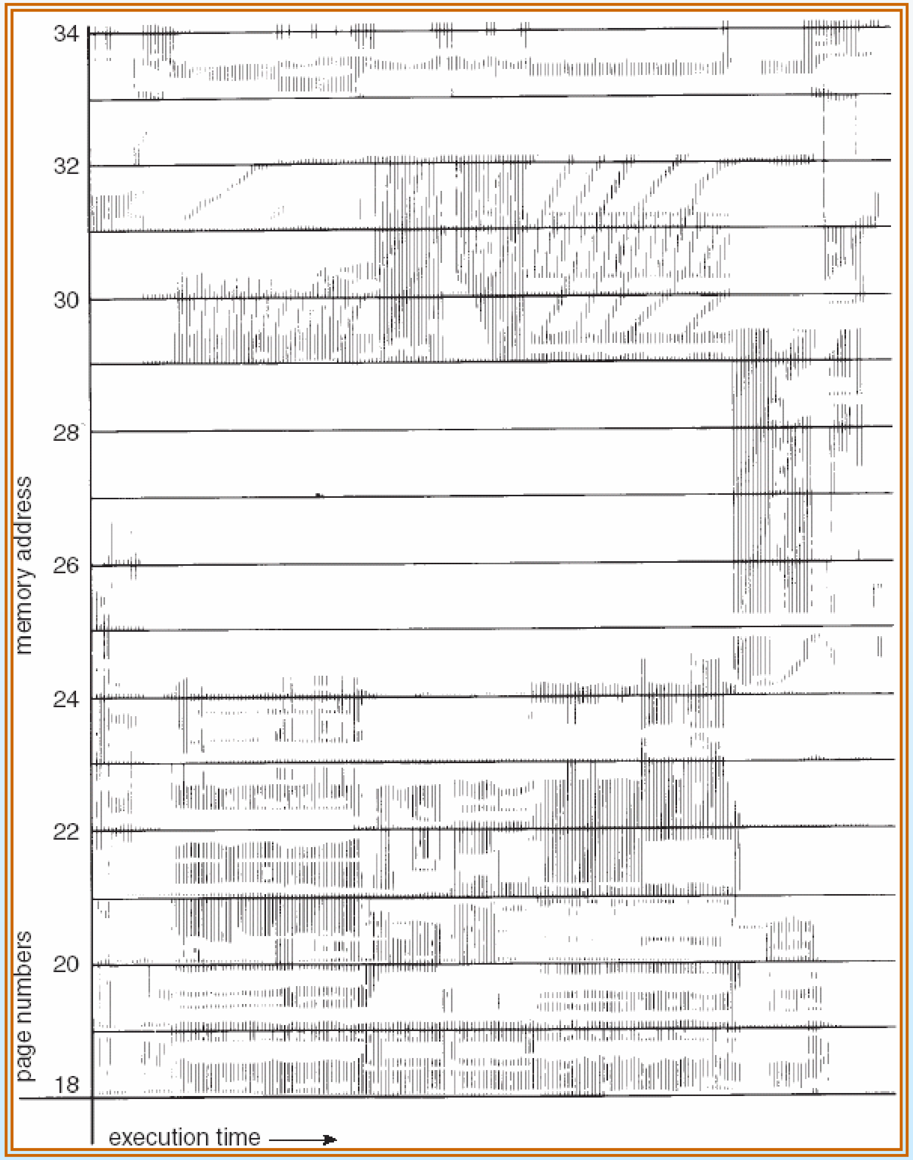
Thrashing (Cont.)



Thrashing (Cont.)

- Thrashing can be limited by using local allocation, but cannot be completely solved
- To prevent thrashing, we must provide a process with as many frames as it needs
- Question:
  - How many it needs?
  - Locality model of process execution
- Process execution tends to work in some locality for some period of time and then move to another
  - Allocate enough frames for current locality then no page faults again during the locality lifetime
  - If not, the process may thrash since it cannot keep enough pages in memory that it is actively using

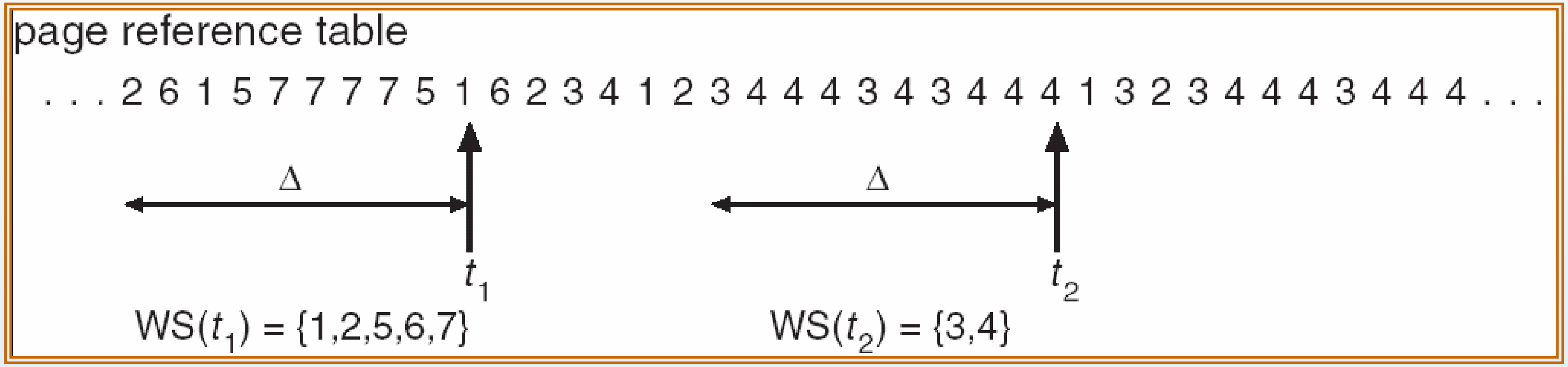
Locality In A Memory-Reference Pattern



Working-Set Model

- $\Delta \equiv$ 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 number of available frames
- Policy
  - if D > m, then suspend one of the processes
  - If enough extra frames, another process can be initiated
- Prevent thrashing while keeping the degree of multiprogramming as high as possible

Working-set model



page reference table

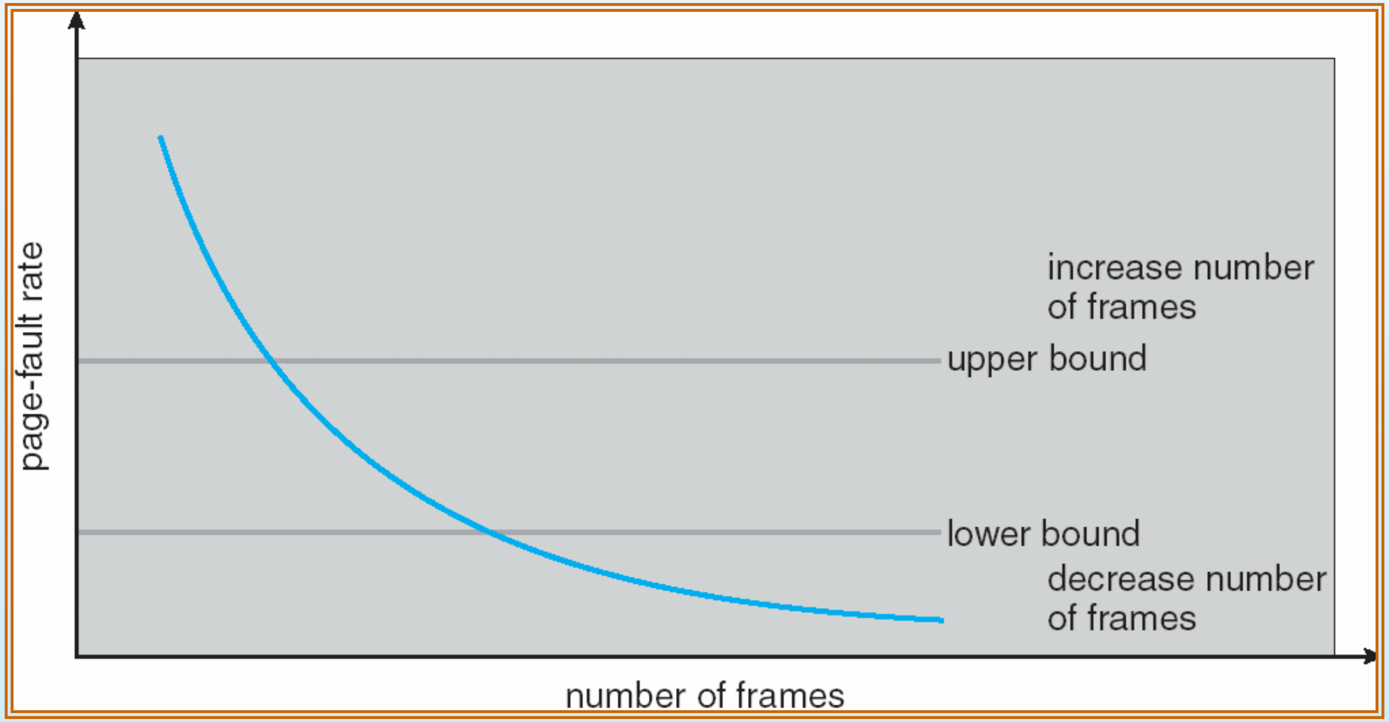
. . . 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 3 4 4 4 3 4 4 4 . . .

$WS(t_1) = \{1,2,5,6,7\}$

$WS(t_2) = \{3,4\}$

Page-Fault Frequency Scheme

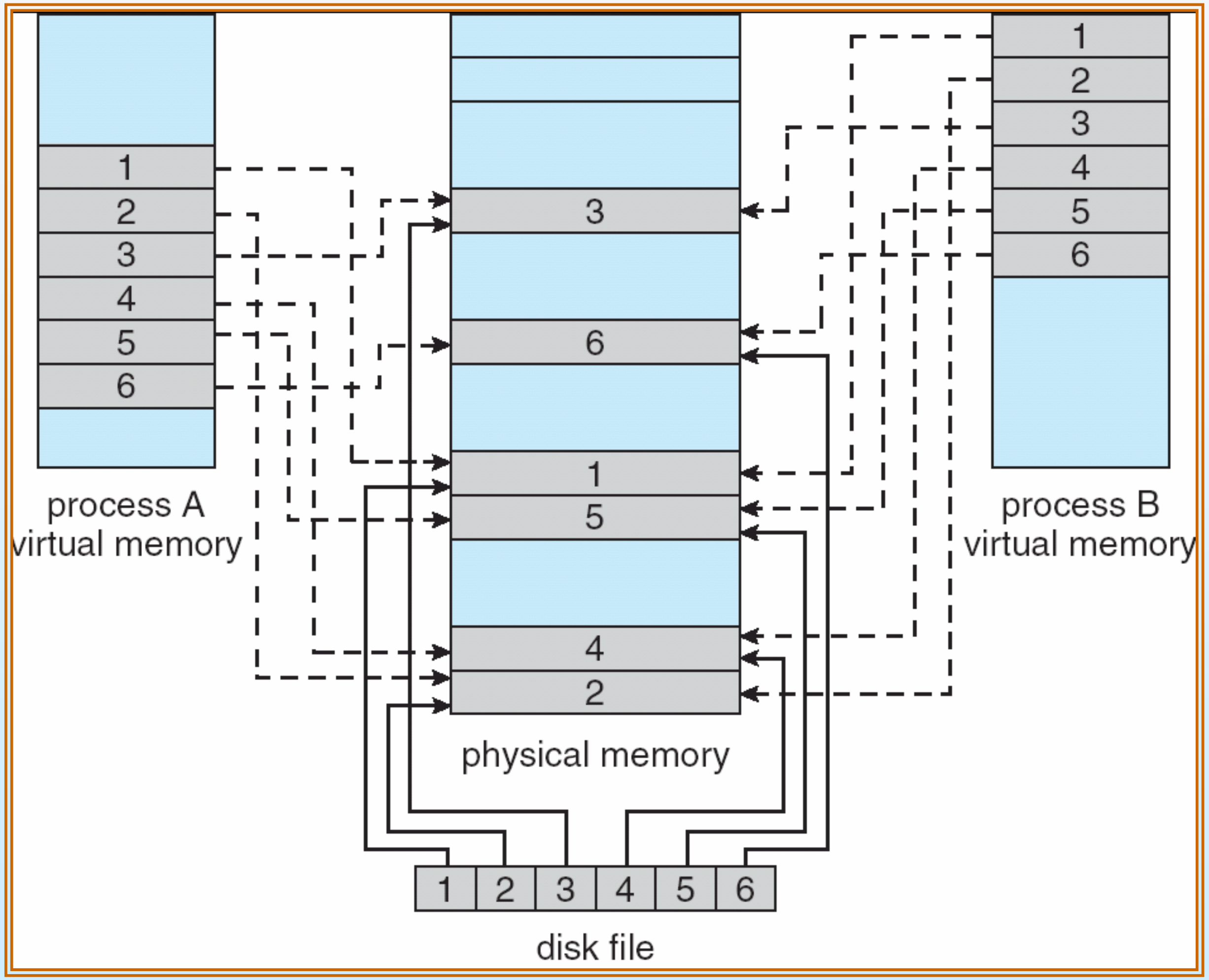
- Establish “acceptable” page-fault rate
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame



Memory-Mapped Files

- Memory-mapped file I/O allows file I/O to be treated as routine memory access by **mapping** a disk block to a page 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 page. Subsequent reads/writes to/from 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 allowing the pages in memory to be shared

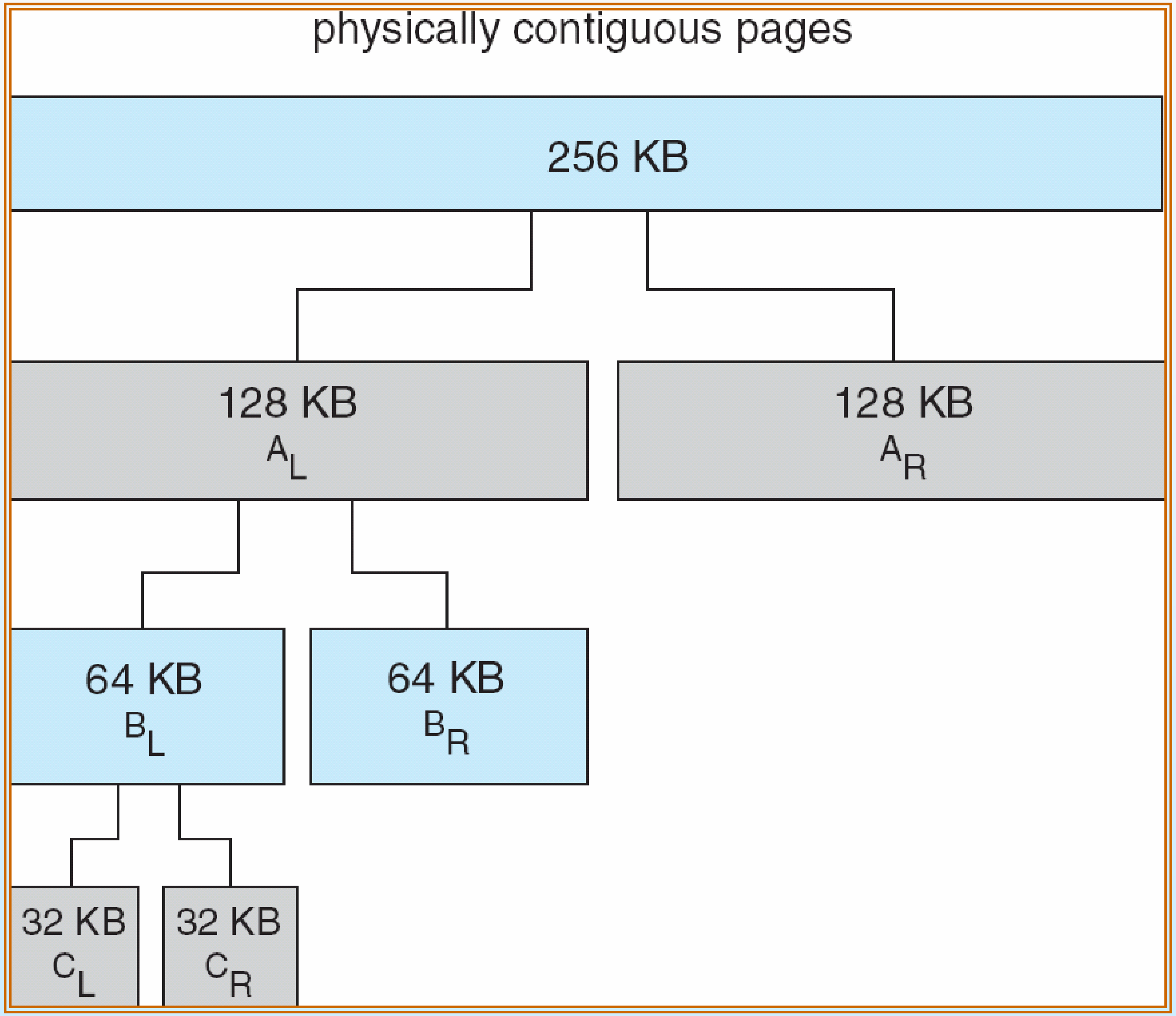
Memory Mapped Files



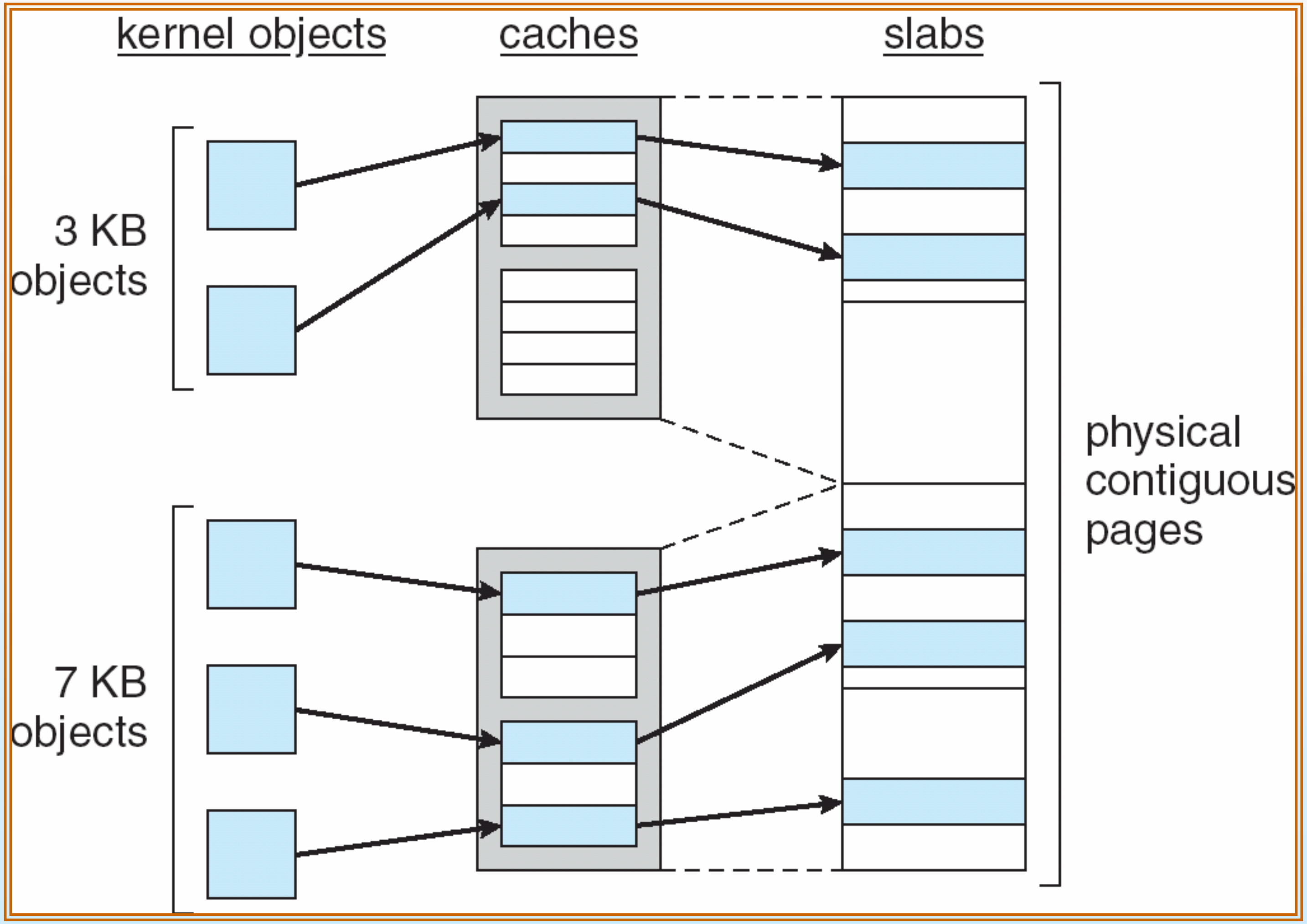
Allocating Kernel Memory

- Kernel memory is often allocated from a free-memory pool different from the list for ordinary user processes
  - To minimize fragmentation: kernel requests memory for data structures of varying sizes, some of which are less than a page
  - Requires contiguous memory: certain hardware devices interact directly with physical memory without virtual memory interface
- Two strategies for managing free memory for kernel processes
  - Buddy system
  - Slab allocation

Buddy System Allocation



Slab Allocation



Prepaging

- Prepaging
  - To reduce the large number of page faults that occurs at process startup
  - Prepage all or some of the pages a process will need, before they are referenced
  - But if prepaged pages are unused, I/O and memory was wasted
  - Assume s pages are prepaged and α of the pages is used
    - Is the cost of $s * \alpha$ saved pages faults > or < than the cost of prepaging $s * (1- \alpha)$ unnecessary pages?
    - α near zero $\Rightarrow$ prepaging loses

Page Size

- Page size selection must take into consideration:
  - fragmentation
  - table size
  - I/O overhead
  - locality

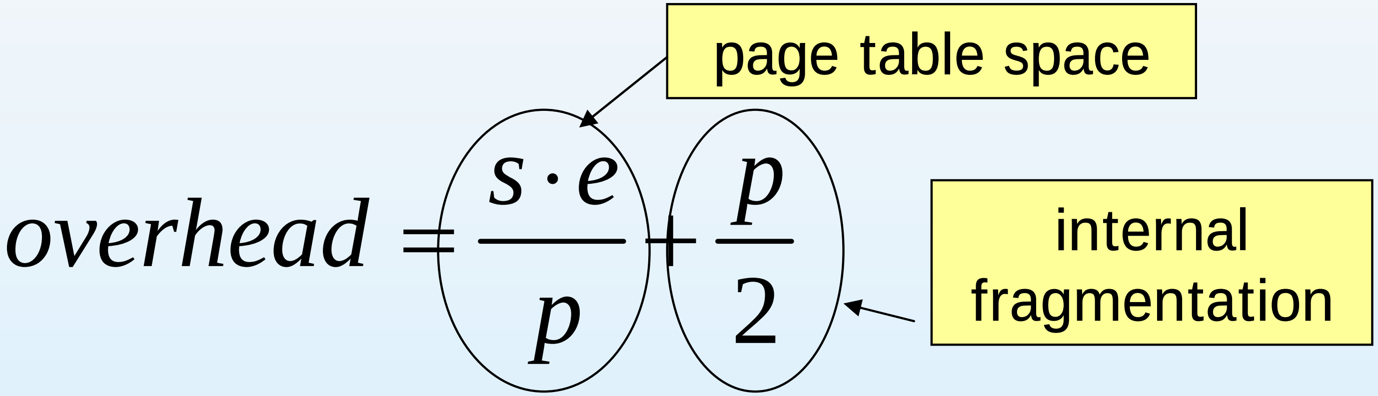
Page Size (Cont.)

Small page size

- Advantages
  - less internal fragmentation
  - better fit for various data structures, code sections
  - less unused program in memory
- Disadvantages
  - programs need many pages, larger page tables

Page Size (Cont.)

- Overhead due to page table and internal fragmentation



$$overhead = \underbrace{\frac{s \cdot e}{p}}_{\text{page table space}} + \underbrace{\frac{p}{2}}_{\text{internal fragmentation}}$$

- Where
  - s = average process size in bytes
  - p = page size in bytes
  - e = page entry size in bytes

Optimized when

$$p = \sqrt{2se}$$

TLB Reach

- TLB Reach: The amount of memory accessible from the TLB
- TLB Reach = (TLB Size) X (Page Size)
- Ideally, the working set of each process is stored in the TLB. Otherwise there is a high degree of page faults.
- 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 applications that require larger page sizes the opportunity to use them without an increase in fragmentation.

Program Structure

- Program structure
  - Int[128,128] data;
  - Each row is stored in one page
  - Program 1

```
for (j = 0; j <128; j++)
      for (i = 0; i < 128; i++)
            data[i,j] = 0;
```

128 x 128 = 16,384 page faults

  - Program 2

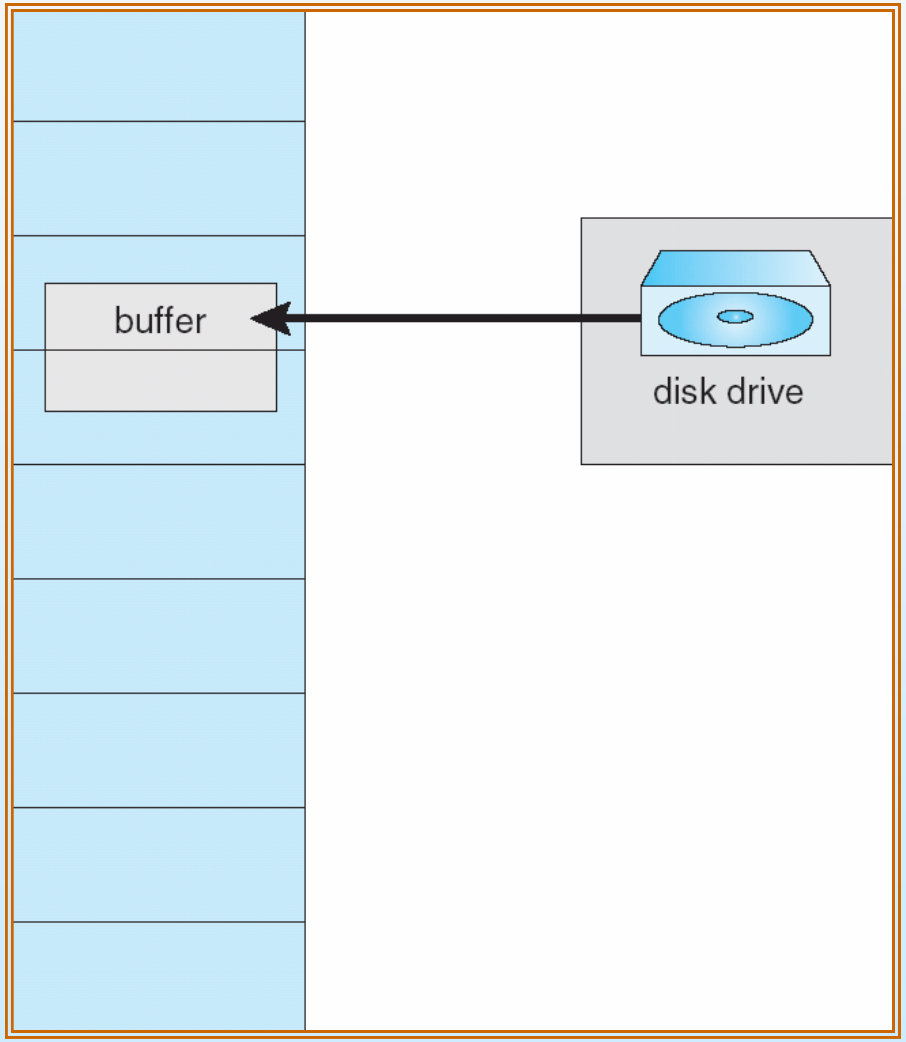
```
for (i = 0; i < 128; i++)
      for (j = 0; j < 128; j++)
            data[i,j] = 0;
```

128 page faults

I/O interlock

- **I/O Interlock** – Pages must sometimes be locked into memory
  - A process is doing I/O and has some pages for buffer
  - After issuing I/O, the process will be suspended while the I/O device is transferring data to buffer
  - A page for I/O buffer may be replaced by other process via global allocation
- Consider I/O. Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm.

Reason Why Frames Used For I/O Must Be In Memory



End of Chapter 9