



# 操作系统原理及应用

---

李 伟

xchlw@seu.edu.cn

计算机科学与工程学院、软件学院  
江苏省网络与信息安全重点实验室



# Chapter 9 Virtual Memory

---



# Outline

---

- **Background**
- **Demand Paging**
- **Page Replacement**
- **Allocation of Frames**
- **Thrashing**
- **Memory-Mapped Files**
- **Allocating Kernel Memory**
- **Other Considerations**
- **Operating-System Examples**

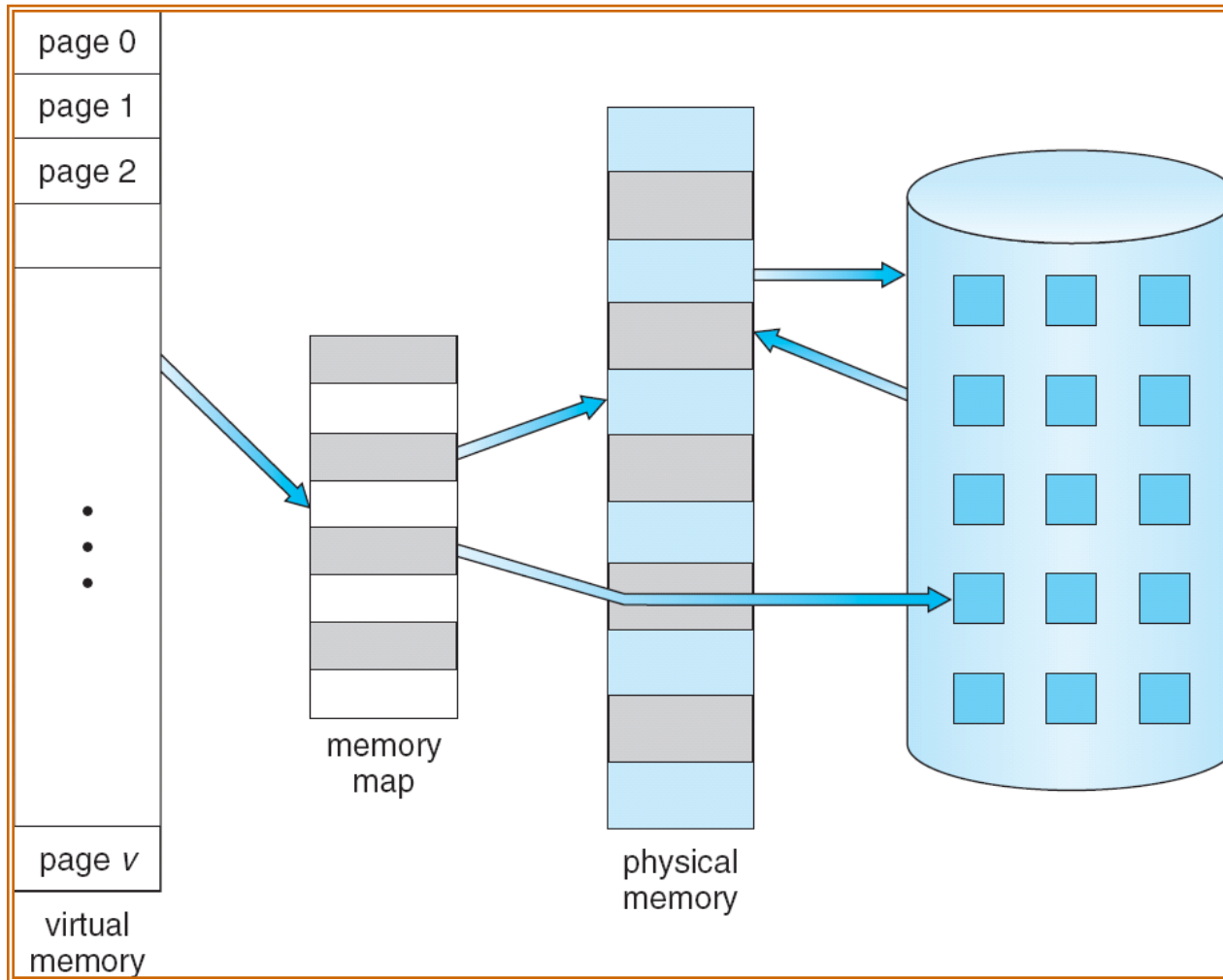


# Background

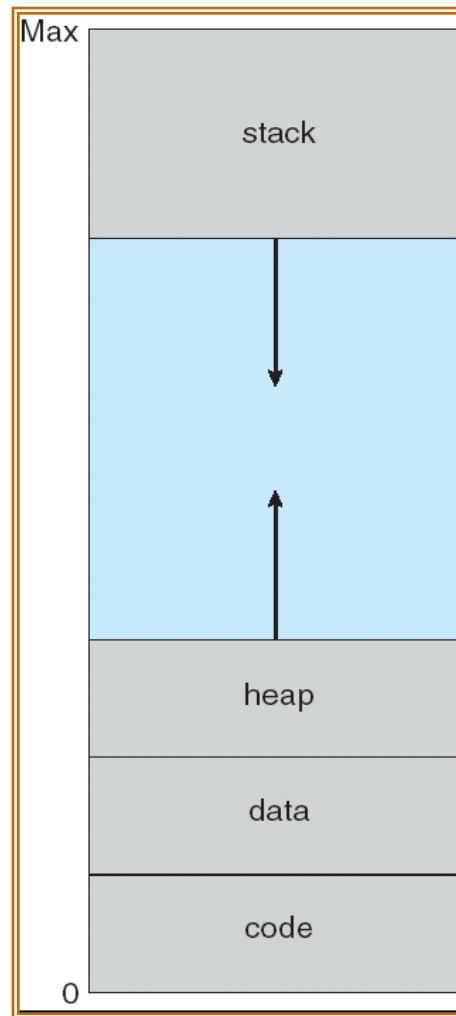
---

- **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.**
  - **More programs can be run at the same time**
  - **Less I/O be needed to load or swap**

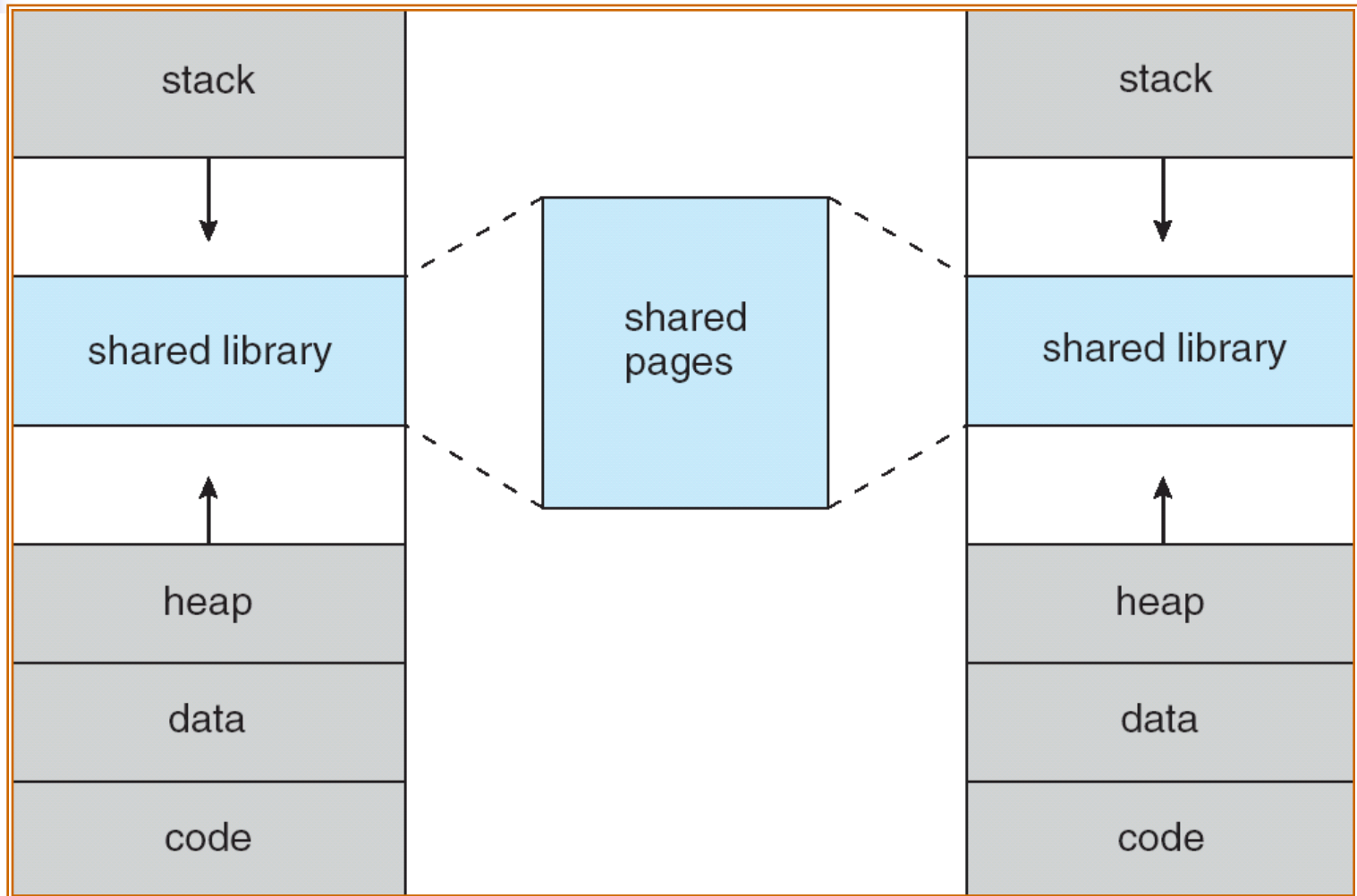
# Virtual Memory is Larger than Physical Memory



# Virtual-address Space



# Shared Library Using Virtual Memory





# Copy-on-Write

---

- 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, then only the page is copied.
- COW allows more efficient process creation as only modified pages are copied.
- Free pages are allocated from **a pool of zeroed-filled pages.**





# Outline

---

- Background
- **Demand Paging**
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples



# 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  $\Rightarrow$  reference to it
  - invalid reference  $\Rightarrow$  abort
  - not-in-memory  $\Rightarrow$  bring to memory
- **Lazy Swapper (Pager)** – never swaps a page into memory unless page will be needed

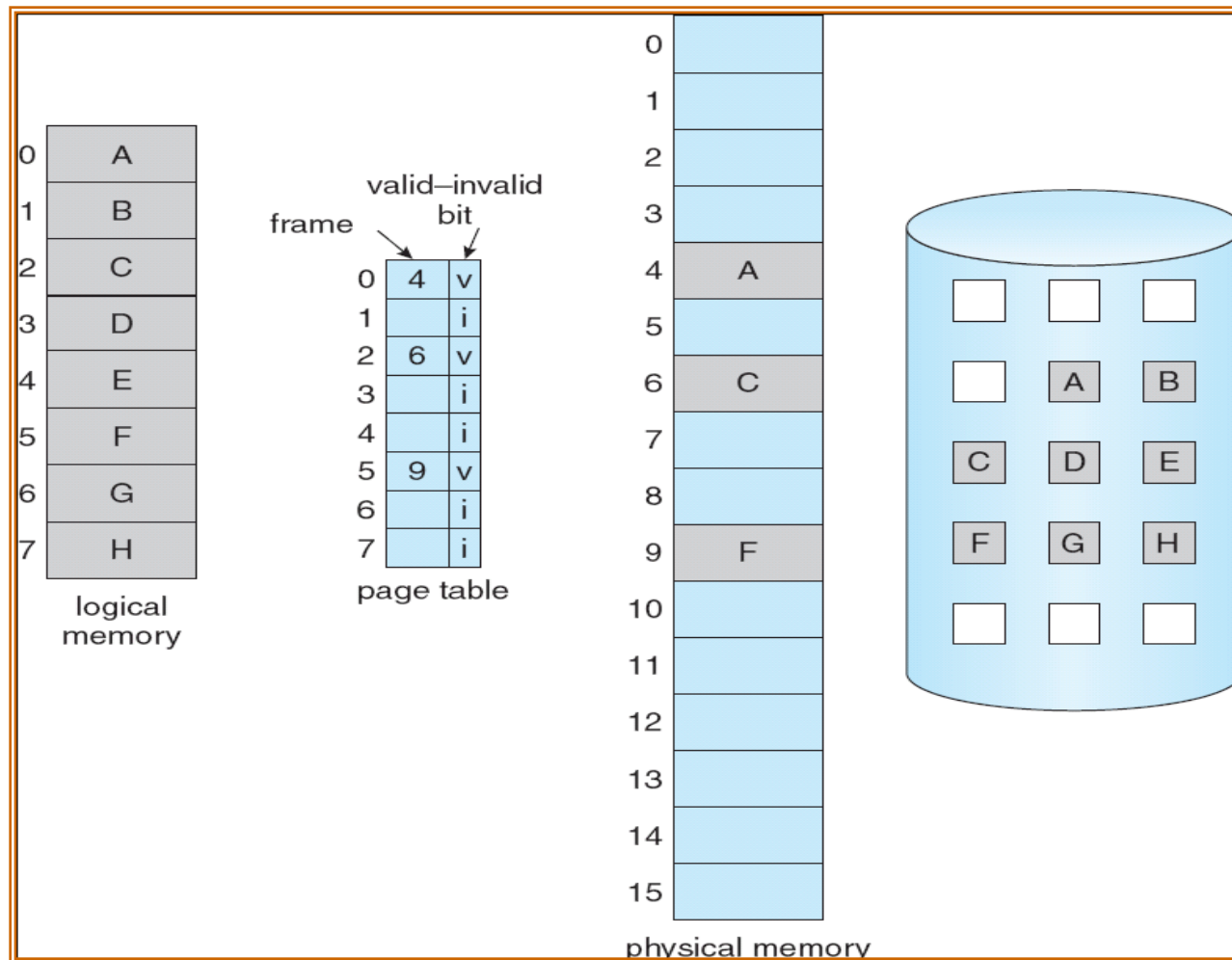


# Valid-Invalid Bit

---

- A valid–invalid bit is associated with each page table entry
  - 1  $\Rightarrow$  valid and in-memory
  - 0  $\Rightarrow$  invalid or not-in-memory
- Initially valid–invalid bit is set to 0 on all entries.
- During address translation, if valid–invalid bit in page table entry is 0  $\Rightarrow$  **page fault**.

# Page Table When Some Pages Are Not in Main Memory



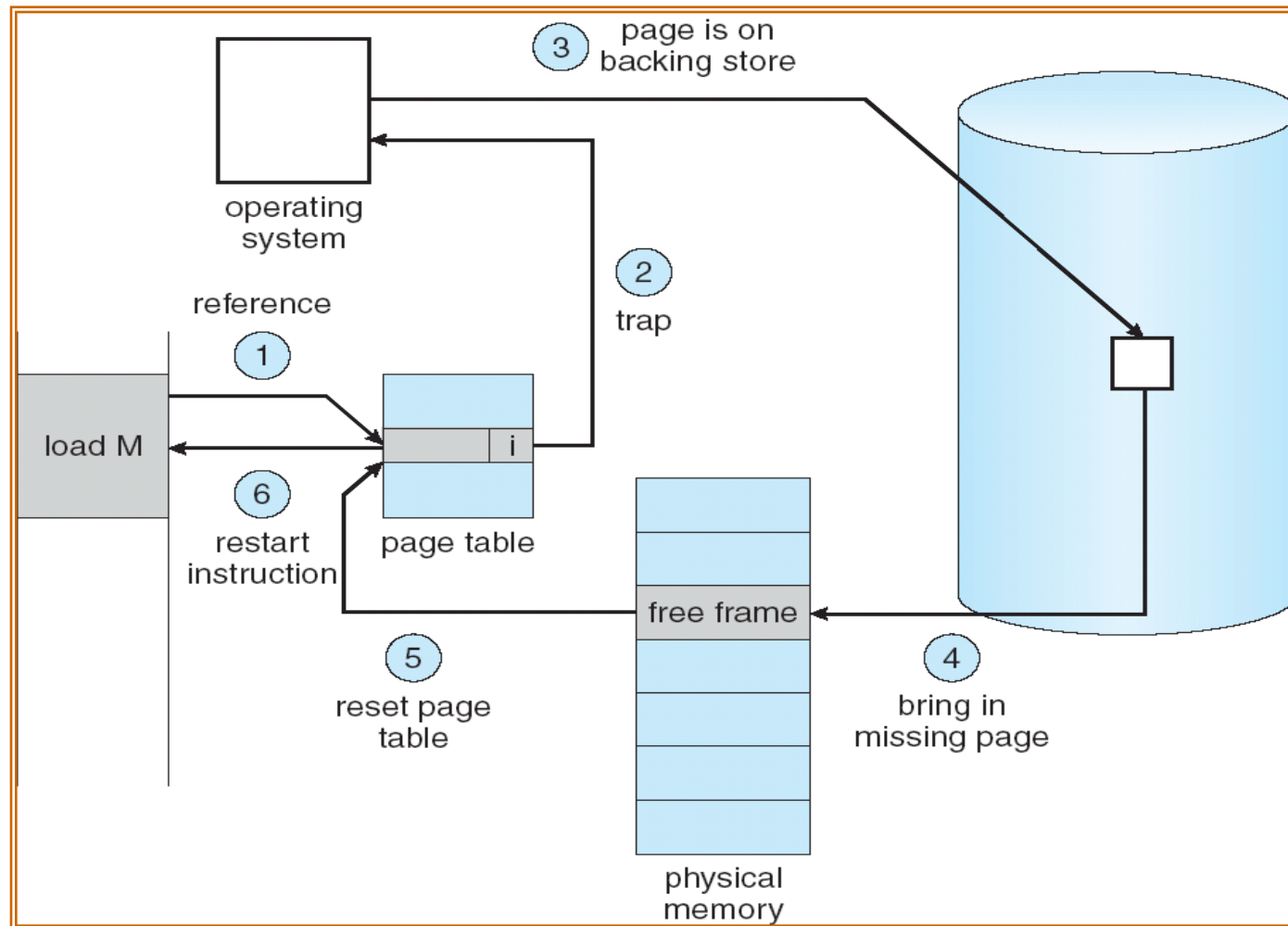


# Page Fault

---

- If there is ever a reference to a page, first reference will trap to OS  $\Rightarrow$  page fault
  - OS looks at internal table to decide:
    - Invalid reference  $\Rightarrow$  abort.
    - Just not in memory.
  - Get empty frame.
  - Swap page into frame.
  - Reset tables, validation bit = 1.
  - Restart instruction

# Steps in Handling a Page Fault





# Performance of Demand Paging

---

- **Page Fault Rate**  $0 \leq p \leq 1$

- if  $p = 0$  no page faults
- if  $p = 1$ , every reference is a fault

- **Effective Access Time (EAT)**

$$\text{EAT} = (1 - p) \times \text{memory access time} \\ + p (\text{page fault overhead})$$

page fault overhead = service the page-fault interrupt  
+ [swap page out ]  
+ swap page in  
+ restart overhead



# Outline

---

- Background
- Demand Paging
- **Page Replacement**
- Allocation of Frames
- Thrashing
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples





# What happens if there is no free frame?

---

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

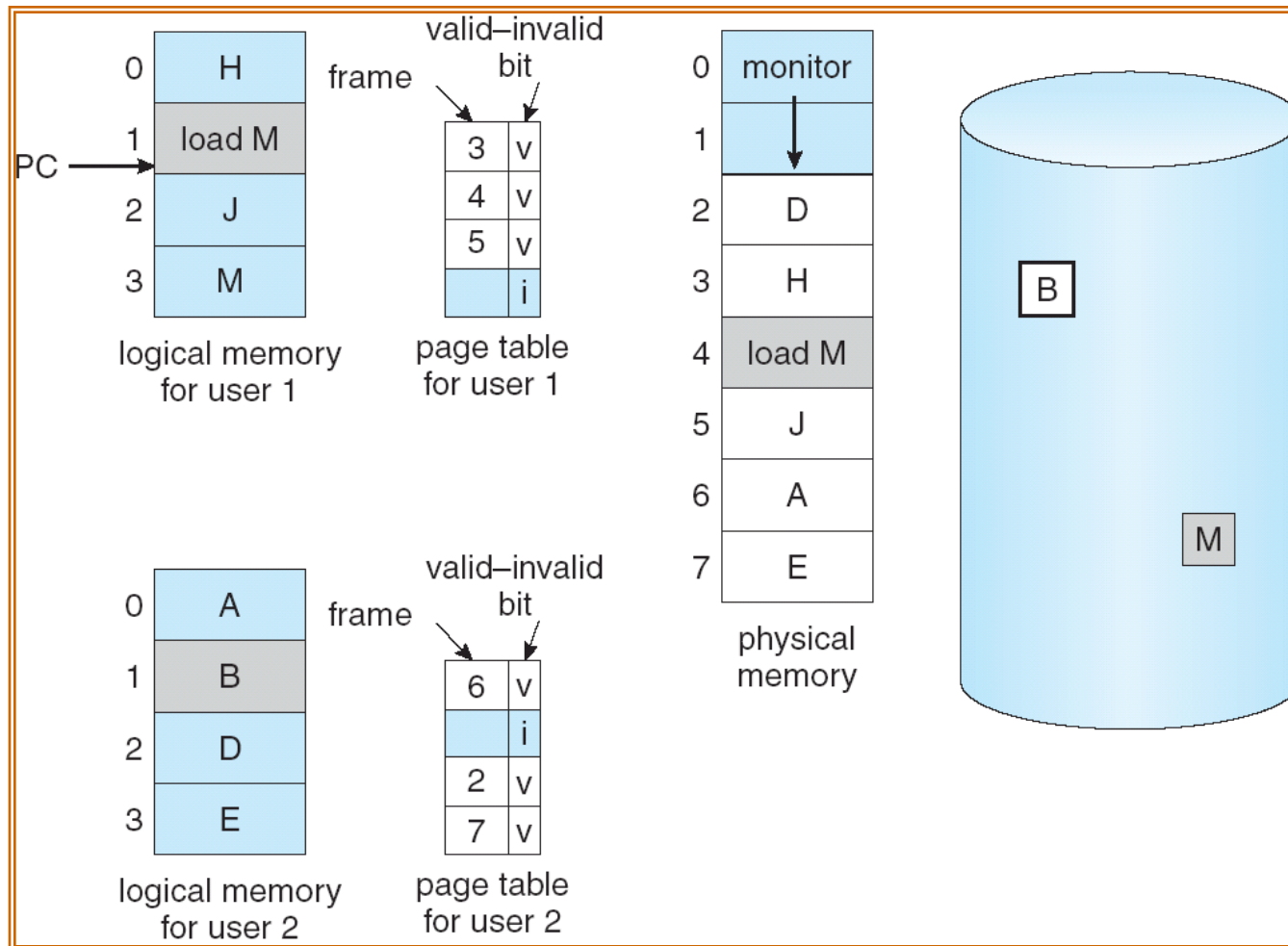


# Page Replacement

---

- Prevent **over-allocation** of memory by modifying page-fault service routine to include page replacement.
- 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



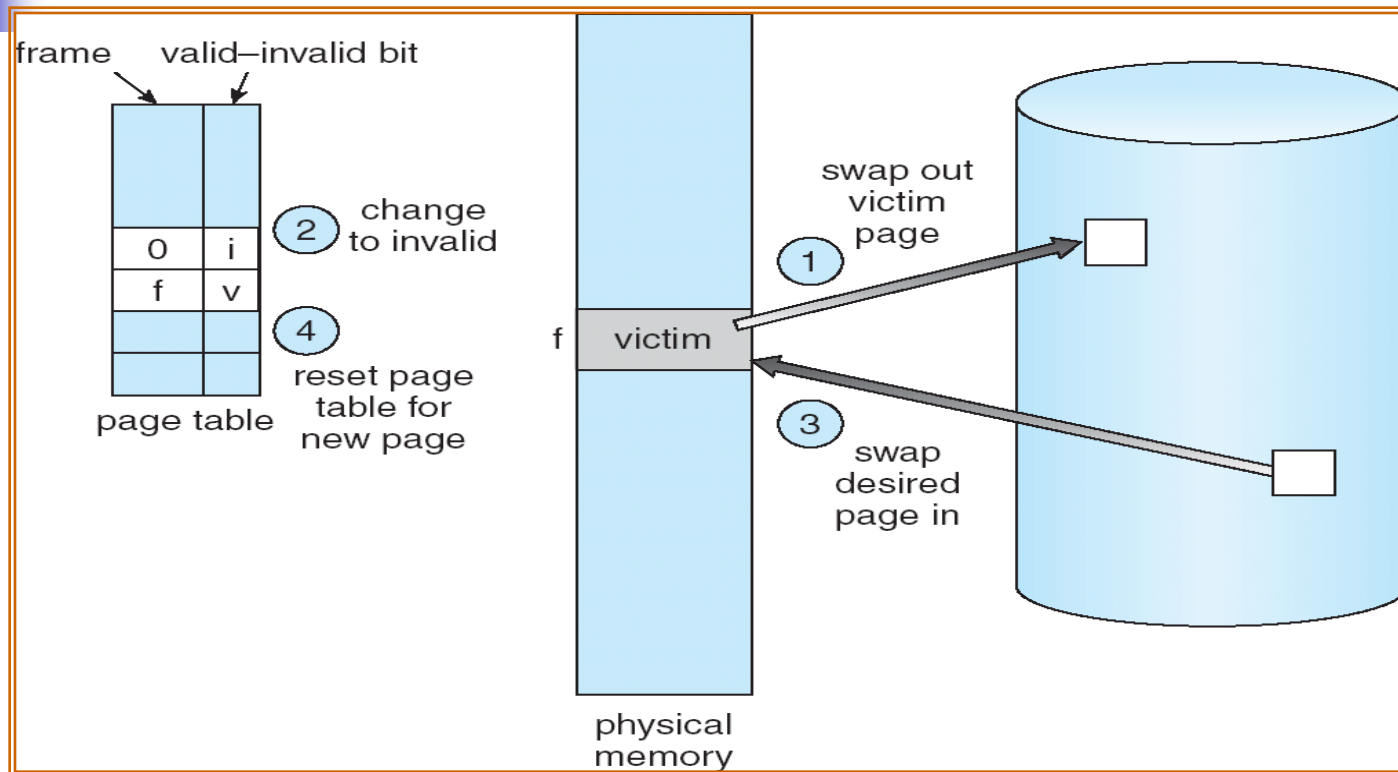


# Basic Page Replacement

---

- Find the location of the desired page on disk.
- 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**.
  - Write the victim frame to the disk and change the page and frame tables.
- Read the desired page into the free frame. Update the page and frame tables.
- Restart the process.

# Basic Page Replacement



Use ***modify (dirty) bit*** to reduce overhead of page transfers  
– only modified pages are written to disk.

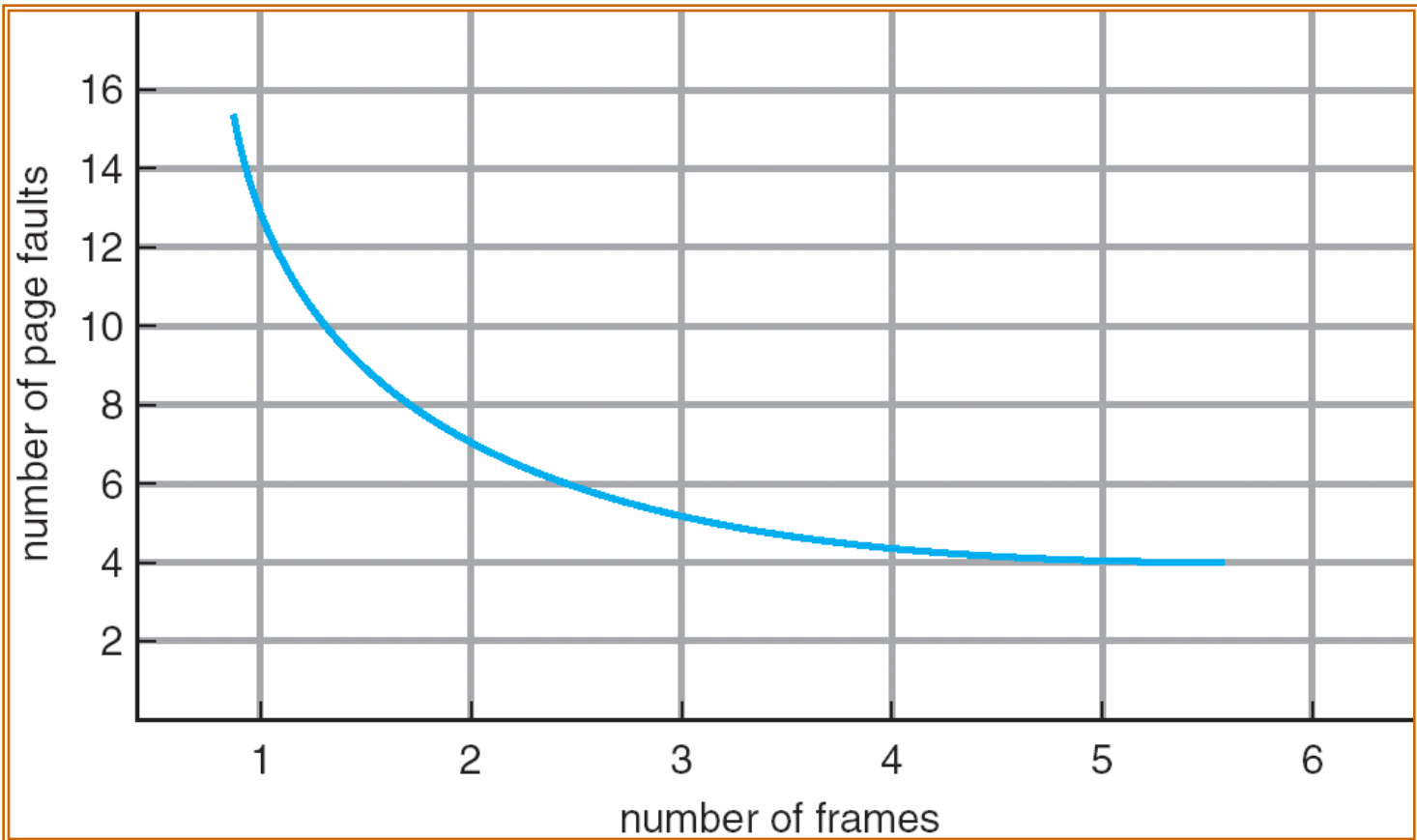


# Page Replacement Algorithms

---

- 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.

# Graph of Page Faults Versus The Number of Frames



# 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	4	5
2	1	3
3	2	4

9 page faults



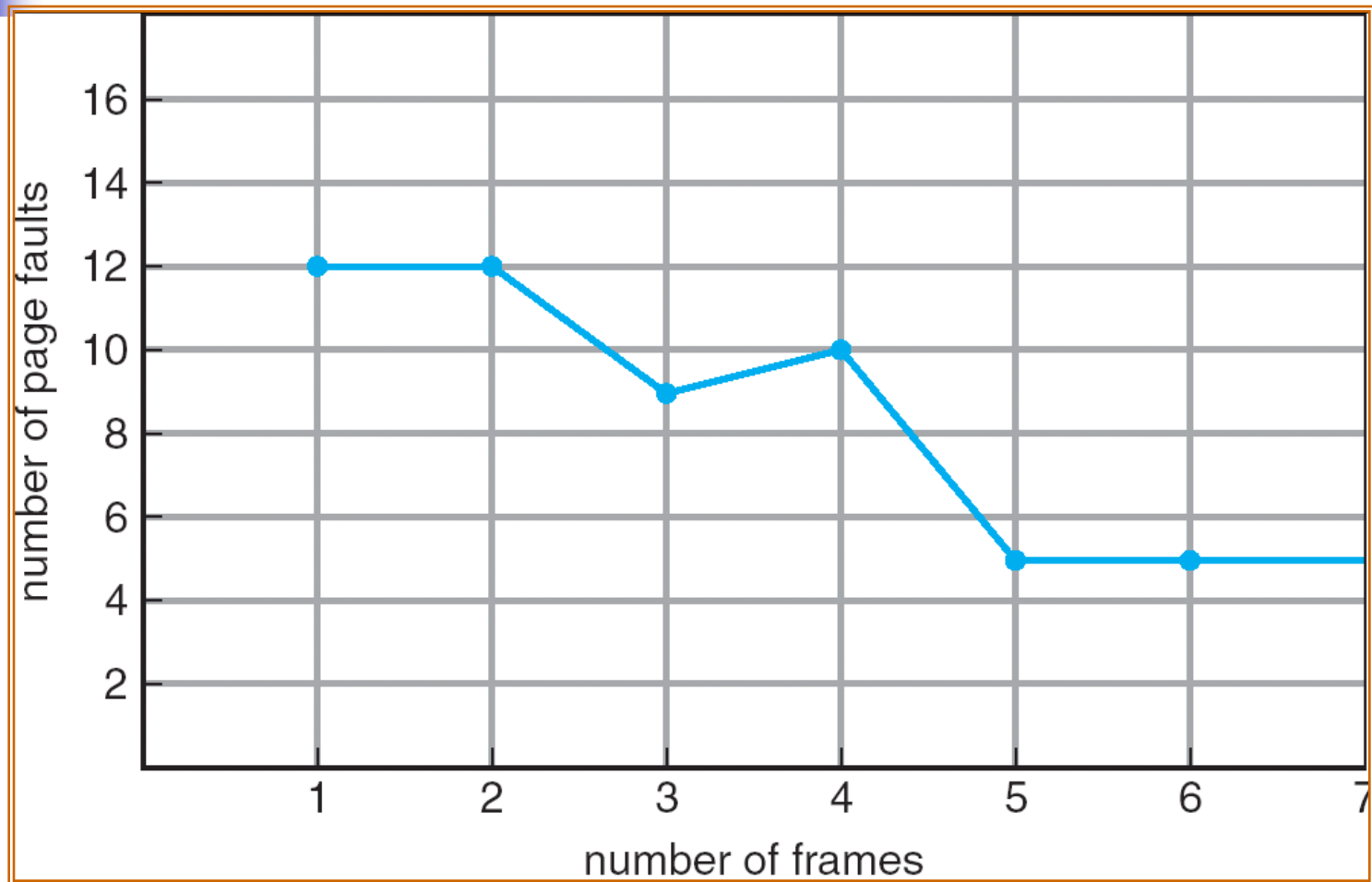
# First-In-First-Out (FIFO) Algorithm

- 4 frames

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

- FIFO Replacement – **Belady's Anomaly**
  - more frames  $\Rightarrow$  more page faults

# FIFO Illustrating Belady's Anamoly





# 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

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



# Least Recently Used (LRU) Algorithm

---

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

1	5
2	
3	5 4
4	3

- **Counter implementation**
  - Every page entry has a **counter**; every time page is referenced through this entry, the clock is copied into the counter.
  - Replace the page with the smallest time value.



# LRU Algorithm

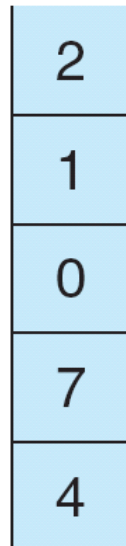
---

- **Stack implementation** – keep a stack of page numbers in a double link form
  - Page referenced
    - remove it from the stack and put it on the top
    - LRU page is always at the bottom
    - requires 6 pointers to be changed at worst
  - No search for replacement
- Optimal replacement and LRU replacement do not suffer from Belady's anomaly

# Use of a Stack to Record the Most Recent Page References

reference string

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



stack  
before  
a



stack  
after  
b

↑  
a

↑  
b



# LRU Approximation Algorithms

---

## ■ Reference bit

- With each entry in the page table associate a bit, initially = 0
- When page is referenced, the bit will be set to 1.
- Replace the one which bit is 0 (if one exists).  
We do not know the order of use, however we can determine which pages have been used and which have not been used.



# LRU Approximation Algorithms

---

- **Additional-Reference-Bits Algorithm**
  - Keep an 8-bit bytes for each page
  - At regular intervals shifts the bits **right** 1 bit, shift the **reference bit** into the high-order bit
  - Interpret these 8-bit bytes as unsigned integers, **the page with lowest number** is the LRU page



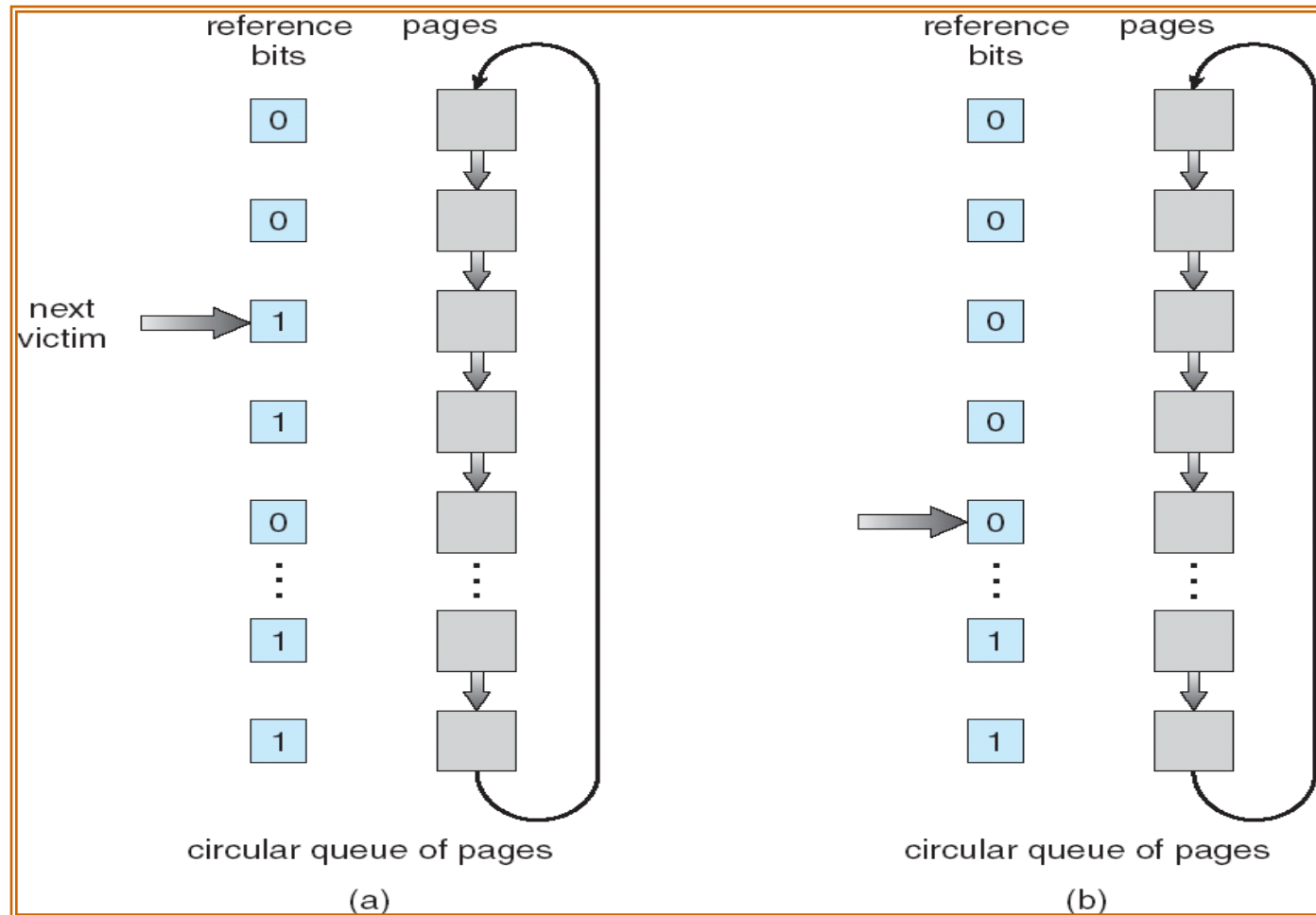


# LRU Approximation Algorithms

---

- **Second-Chance Algorithm**
  - Need reference bit.
  - Clock replacement (**FIFO**).
  - 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 Page-Replacement Algorithm





# 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.



# Outline

---

- Background
- Demand Paging
- Page Replacement
- **Allocation of Frames**
- Thrashing
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples



# Allocation of Frames

---

- Each process needs **minimum** number of pages, **which is decided by the given computer architecture.**
- Example: IBM 370 MVC 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** – e.g., if 100 frames and 5 processes, give each 20 pages.
- **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

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



# 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 vs. Local Allocation

---

- **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.





# Outline

---

- Background
- Demand Paging
- Page Replacement
- Allocation of Frames
- **Thrashing**
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples

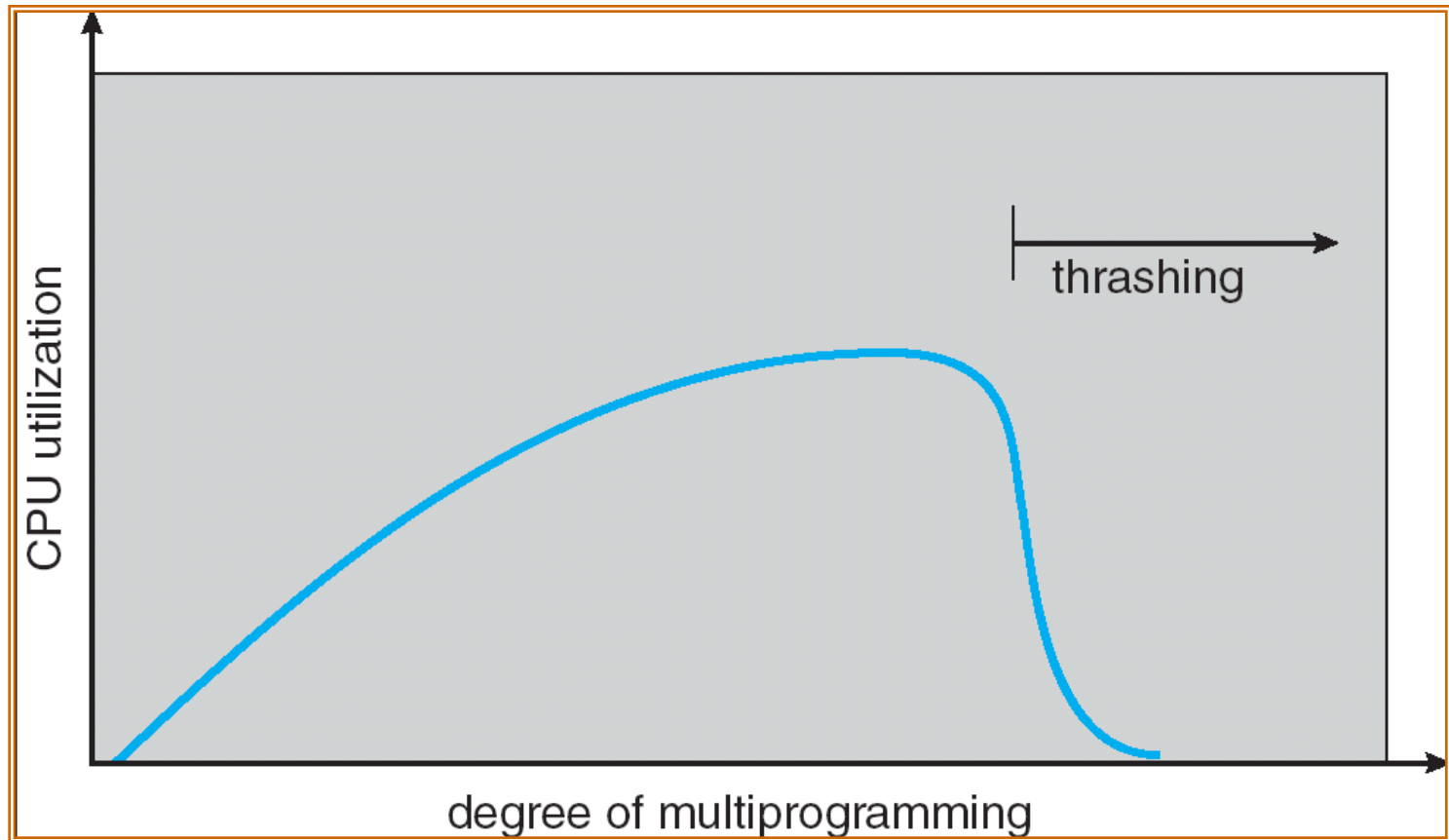


# Thrashing

---

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

# Thrashing





# Thrashing

---

- **Why does paging work?**

- Locality model**

- A locality is a set of pages that are actively used together.
    - Process migrates from one locality to another.
    - Localities may overlap.

- **Why does thrashing occur?**

- size of locality > allocated memory size



# 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  $\Delta$  (varies in time)
  - if  $\Delta$  too small will not encompass entire locality.
  - if  $\Delta$  too large will encompass several localities.
  - if  $\Delta = \infty \Rightarrow$  will encompass entire program.



# Working-Set Model

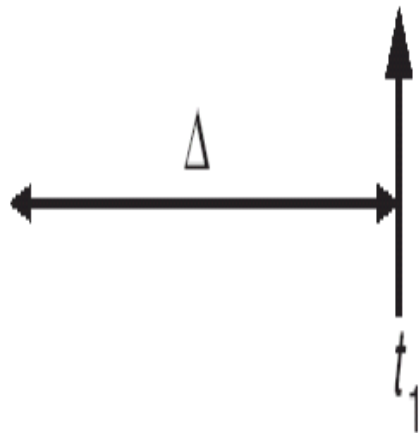
---

- The working set is an approximation of the program's locality.
- $D = \sum WSS_i \equiv$  total demand frames
- if  $D > m \Rightarrow$  Thrashing  $\Rightarrow$  Suspend one of the processes

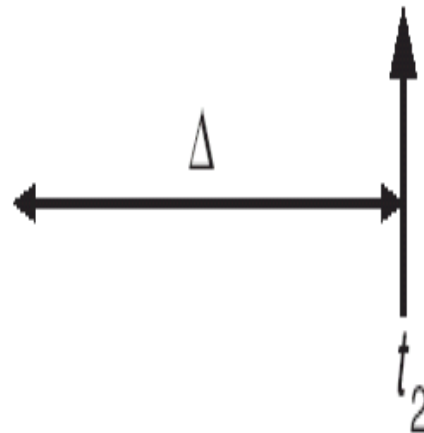
# Working-set model

page reference table

... 2 6 1 5 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 ...

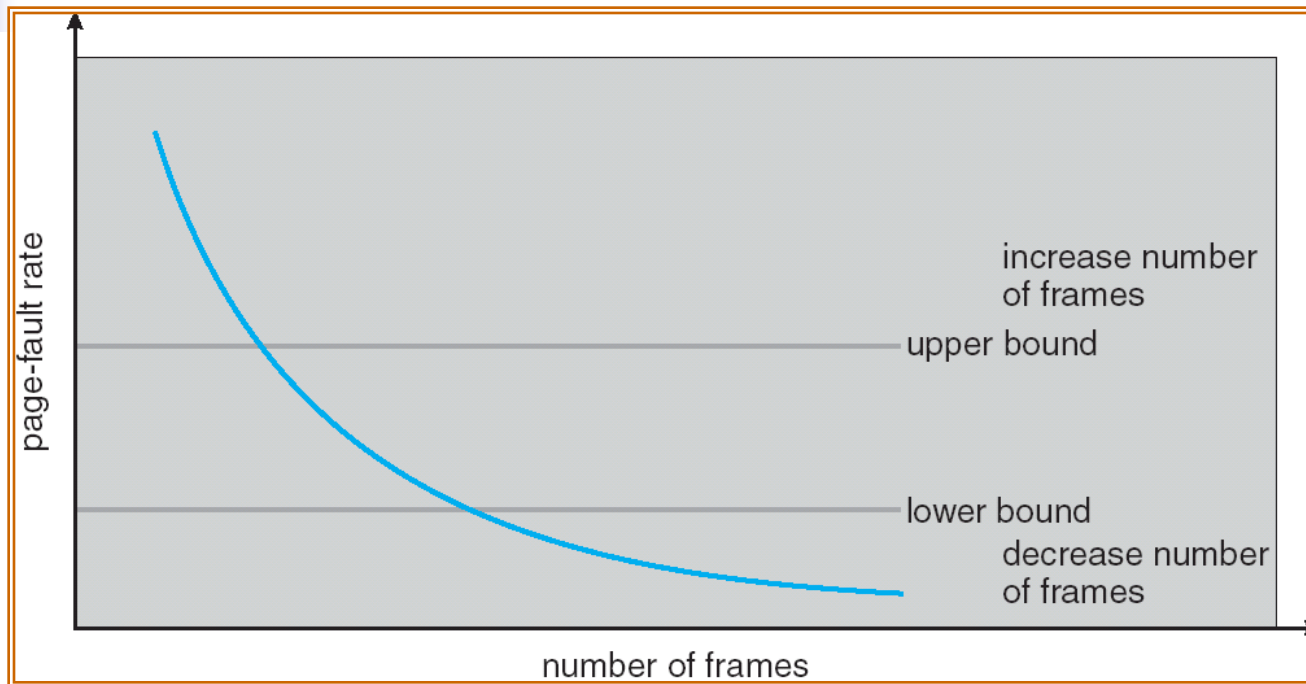


$$WS(t_1) = \{1, 2, 5, 6, 7\}$$



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

# Page-Fault Frequency



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





# Outline

---

- Background
- Demand Paging
- Page Replacement
- Allocation of Frames
- Thrashing
- **Memory-Mapped Files**
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples

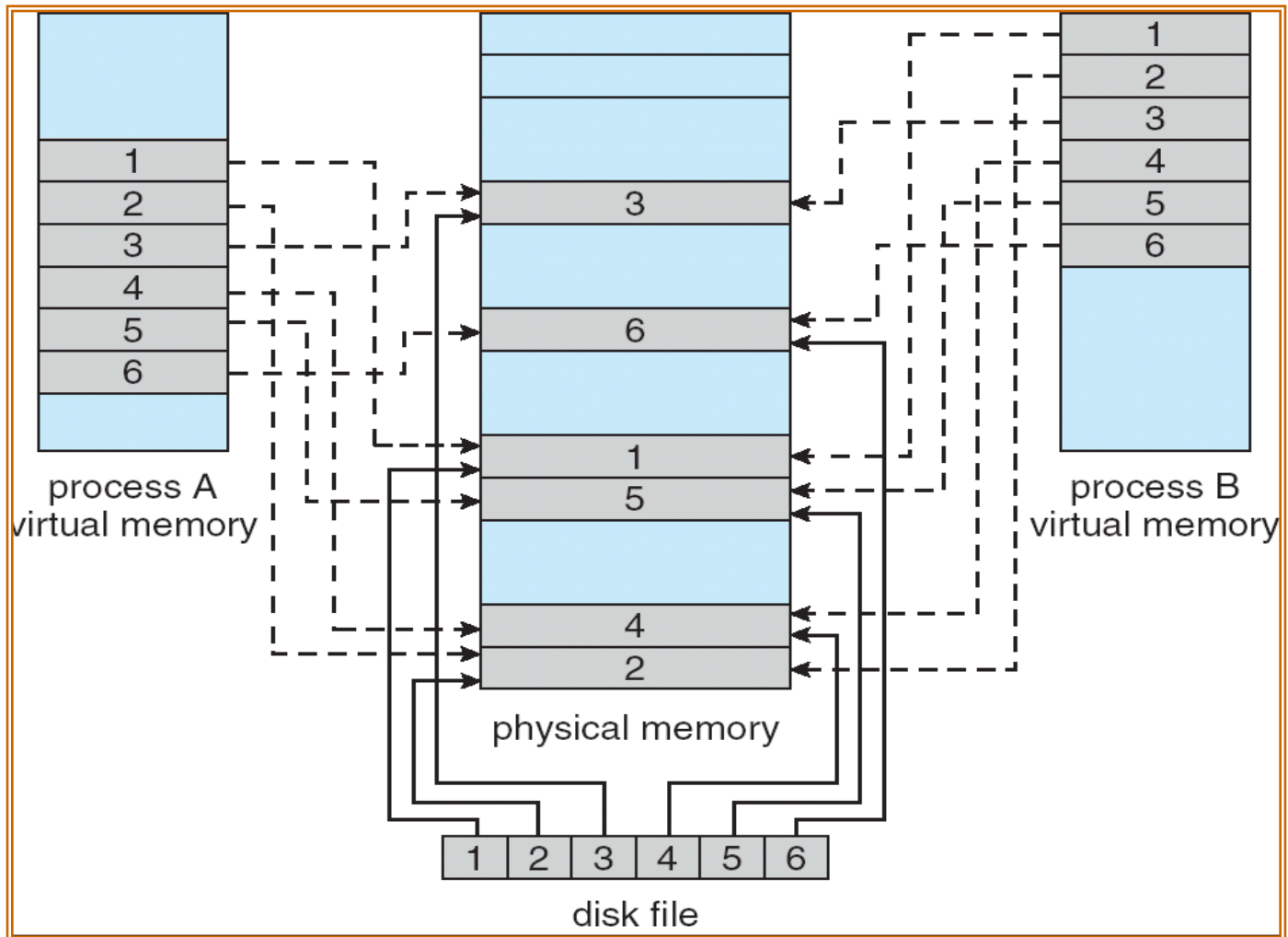


# Memory-Mapped Files

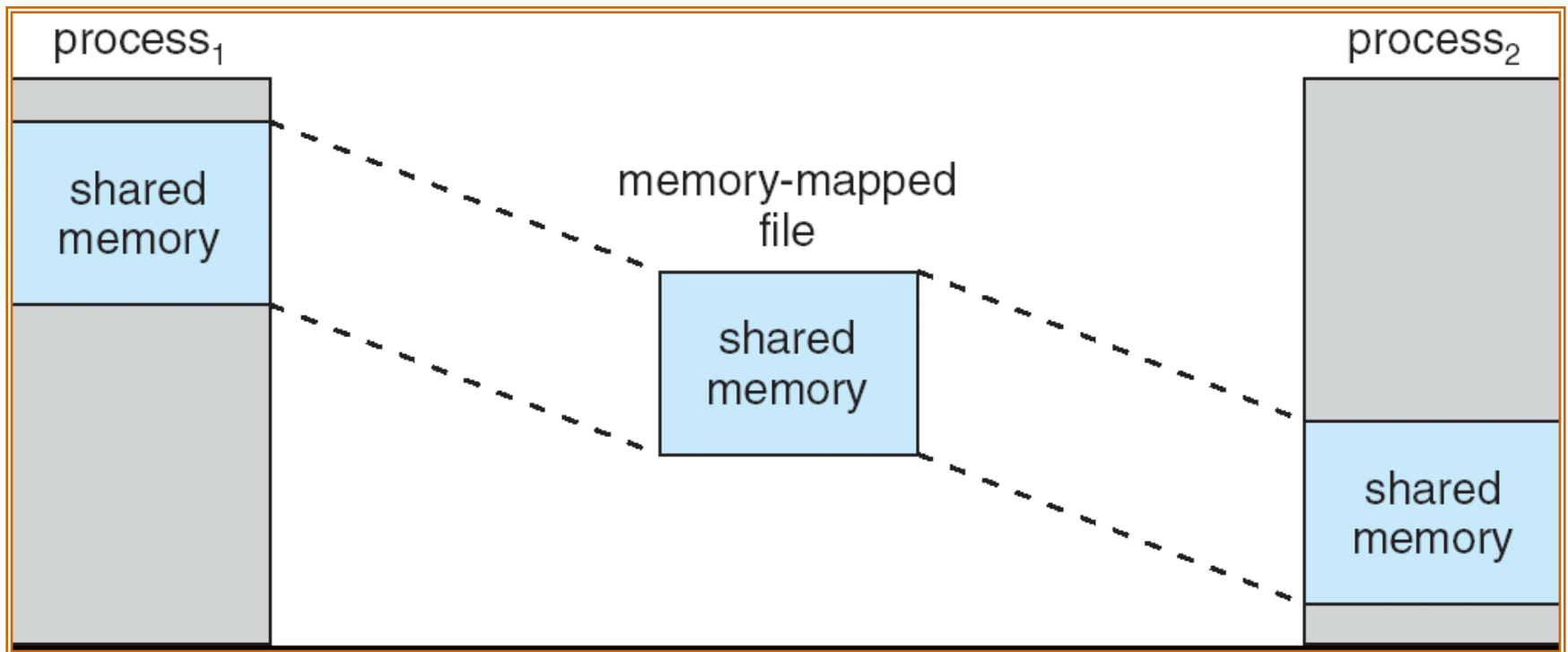
---

- 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



# Memory-Mapped Shared Memory in Windows





# Outline

---

- Background
- Demand Paging
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped Files
- **Allocating Kernel Memory**
- Other Considerations
- Operating-System Examples



# Allocating Kernel Memory

---

- **Treated differently from user memory**
- **Often allocated from a free-memory pool**
  - **Kernel requests memory for structures of varying sizes**
  - **Some kernel memory needs to be contiguous**

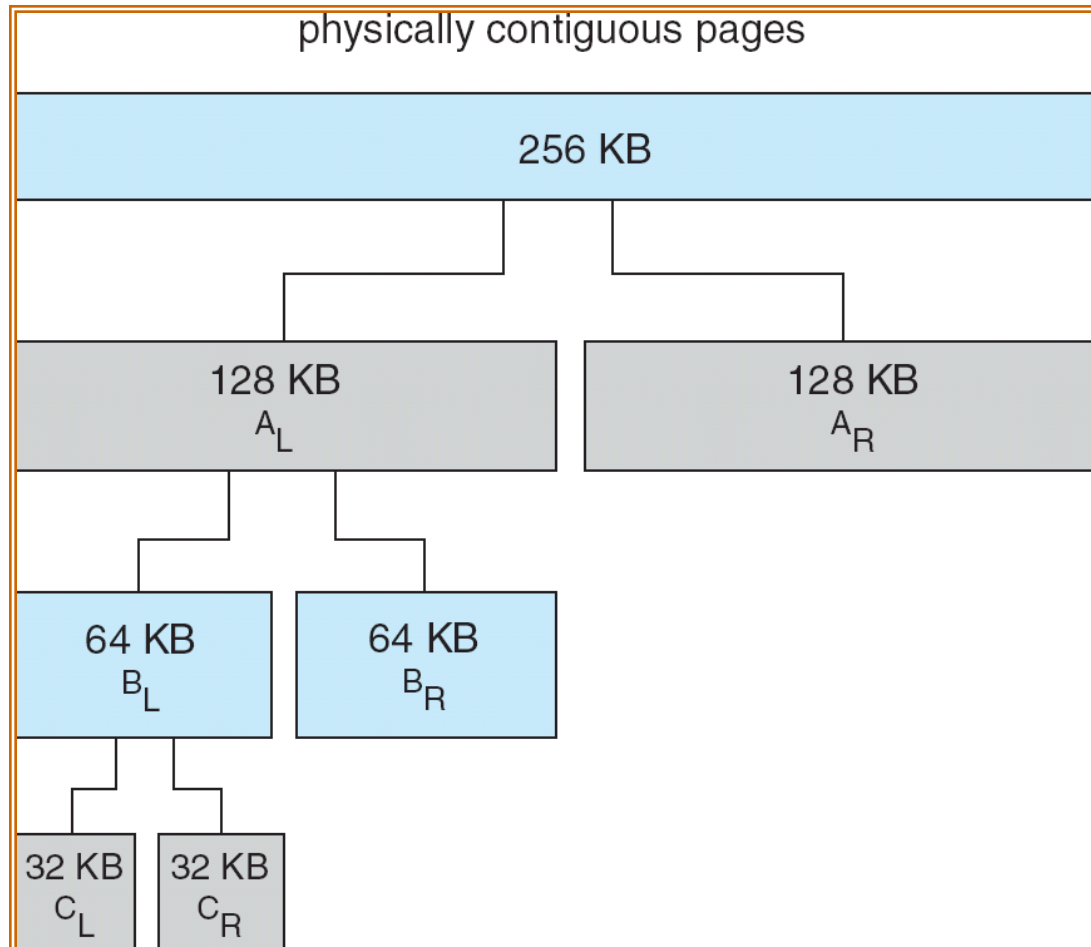


# Buddy System

---

- **Allocates memory from fixed-size segment consisting of physically-contiguous pages**
- **Memory allocated using power-of-2 allocator**
  - **Satisfies requests in units sized as power of 2**
  - **Request rounded up to next highest power of 2**
  - **When smaller allocation needed than is available, current chunk split into two buddies of next-lower power of 2**
    - **Continue until appropriate sized chunk available**

# Buddy System Allocator







# Slab Allocator

---

- Alternate strategy
- **Slab** is one or more physically contiguous pages
- **Cache** consists of one or more slabs
- Single cache for each unique kernel data structure
  - Each cache filled with objects – instantiations of the data structure

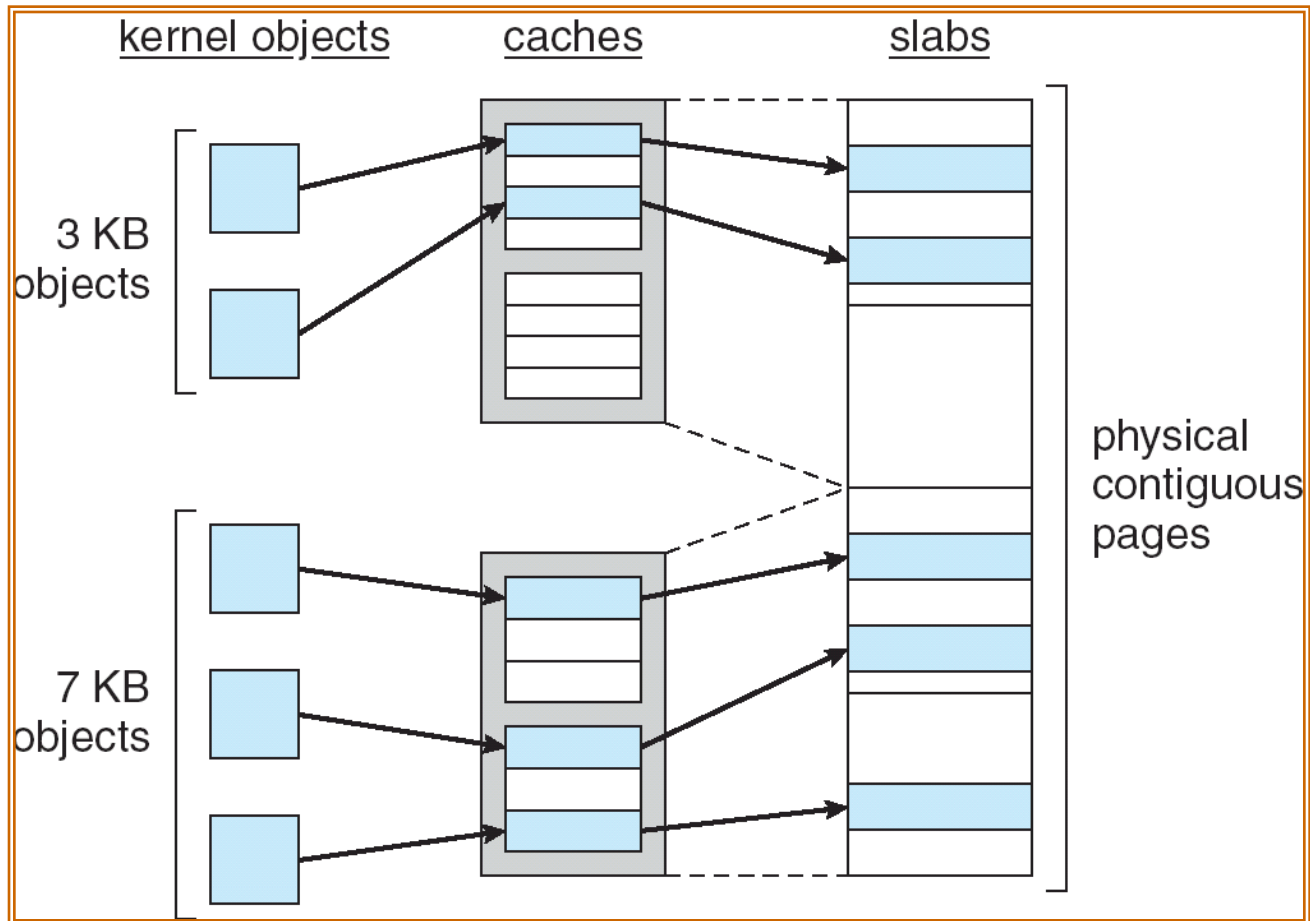


# Slab Allocator

---

- When cache created, filled with objects marked as free
- When structures stored, objects marked as used
- If slab is full of used objects, next object allocated from empty slab
  - If no empty slabs, new slab allocated
- Benefits include no fragmentation, fast memory request satisfaction

# Slab Allocation





# Outline

---

- **Background**
- **Demand Paging**
- **Page Replacement**
- **Allocation of Frames**
- **Thrashing**
- **Memory-Mapped Files**
- **Allocating Kernel Memory**
- **Other Considerations**
- **Operating-System Examples**



# Other Issues -- 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  $\alpha$  of the pages is used**
    - **Is cost of  $s * \alpha$  save pages faults  $>$  or  $<$  the cost of prepaging  $s * (1 - \alpha)$  unnecessary pages?**
    - **$\alpha$  near zero  $\Rightarrow$  prepaging loses**



# Other Issues – Page Size

---

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



# Other Issues – 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**
- **Provide Multiple Page Sizes**
  - **This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation**



# Other Issues – Program Structure

---

- Program structure

- `int A[][] = new int[1024][1024];`

- Each row is stored in one page

- Program 1                      `for (j = 0; j < A.length; j++)`  
   `for (i = 0; i < A.length; i++)`  
   `A[i,j] = 0;`

1024 x 1024 page faults

- Program 2                      `for (i = 0; i < A.length; i++)`  
   `for (j = 0; j < A.length; j++)`  
   `A[i,j] = 0;`

- 1024 page faults



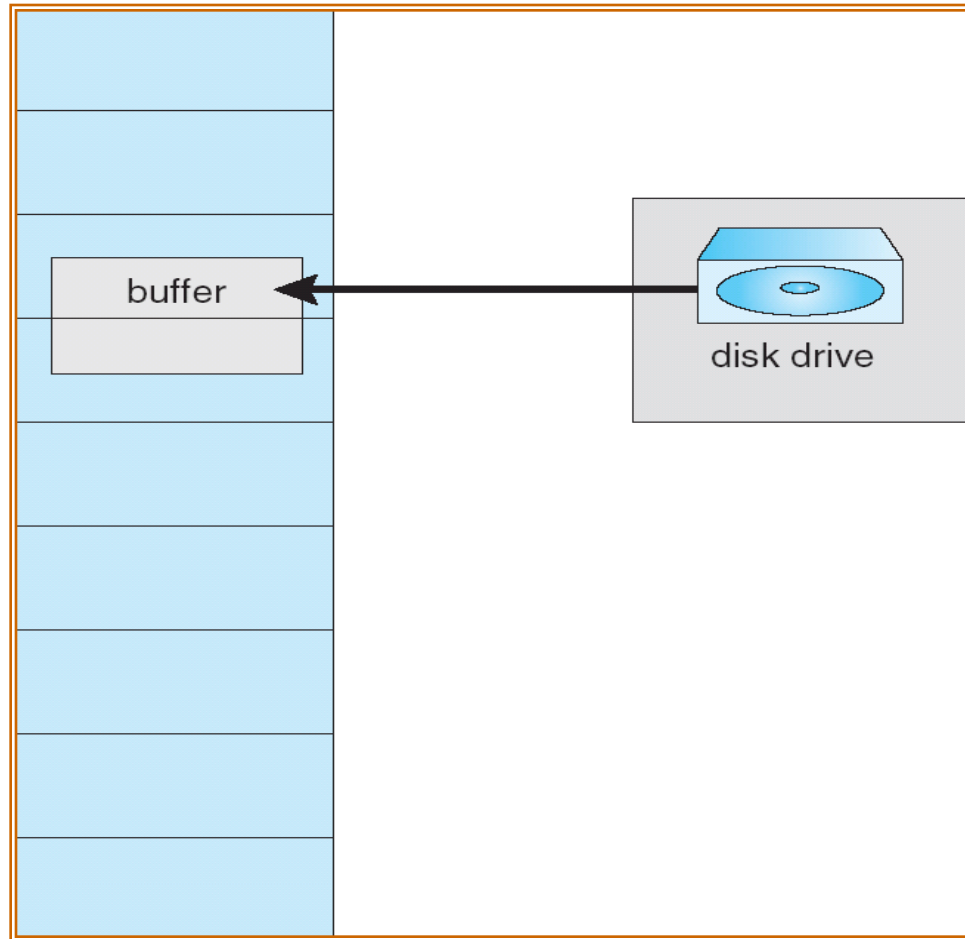


# Other Issues – I/O interlock

---

- I/O Interlock – Pages must sometimes be locked into memory
- 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





# Outline

---

- **Background**
- **Demand Paging**
- **Page Replacement**
- **Allocation of Frames**
- **Thrashing**
- **Memory-Mapped Files**
- **Allocating Kernel Memory**
- **Other Considerations**
- **Operating-System Examples**



# Operating System Examples

---

- **Windows XP**
- **Solaris**



# Windows XP

---

- **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**



# Windows XP

---

- 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



# Solaris

---

- **Maintains a list of free pages to assign faulting processes**
- ***Lotsfree* – threshold parameter (amount of free memory) to begin paging**
- ***Desfree* – threshold parameter to increasing paging**
- ***Minfree* – threshold parameter to being swapping**



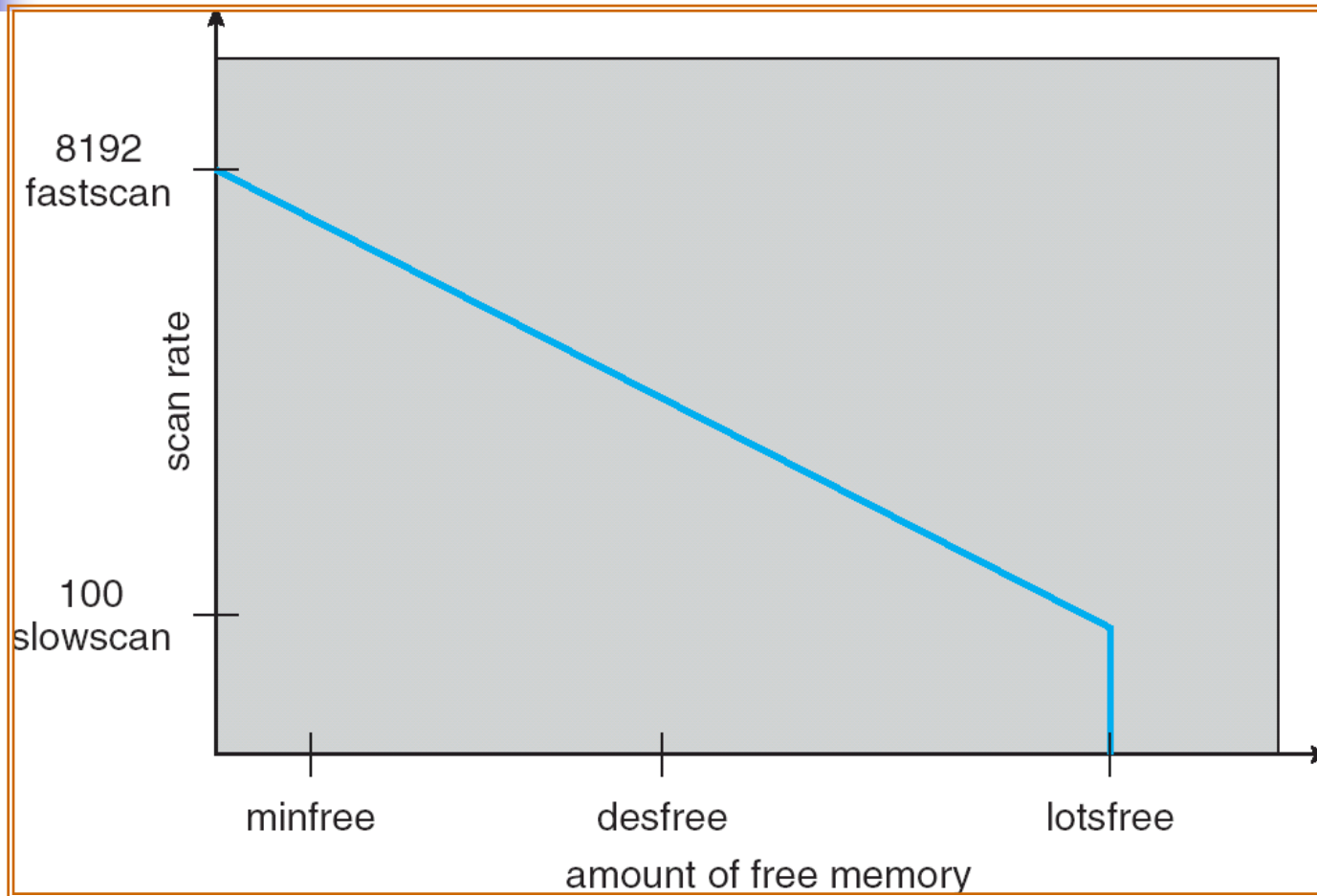
# Solaris

---

- Paging is performed by ***pageout process***
- Pageout scans pages using modified clock algorithm
- ***Scanrate*** is the rate at which pages are scanned. This ranges from ***slowscan*** to ***fastscan***
- Pageout is called more frequently depending upon the amount of free memory available



# Solaris 2 Page Scanner





## Part 3 小结

---

- 基本概念：内存保护（基址寄存器+界限地址寄存器）、地址绑定、逻辑地址空间与物理地址空间、动态加载、动态链接、交换
- 连续分配：固定分区、可变分区（动态分配问题，3种方法）、碎片（内、外）
- 分页：页、帧、页表、逻辑地址结构、页表实现（寄存器、PTBR、TLB）、Hit Ratio（命中率）、内存保护、共享页、页表结构（层次、哈希、反向）、分段
- 按需调页（有效访问时间）、写时复制、页置换算法（FIFO（Belady异常）、最优、LRU、近似LRU、计数）、
- 帧分配（平均、按比例）、系统颠簸（含义、原因、解决方法）、内存映射文件、内核内存分配