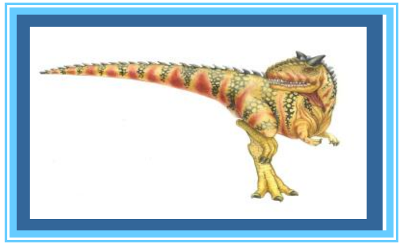
# Chapter 9: Virtual Memory



# Chapter 9:  Virtual Memory

- 9.1 Background
- 9.2 Demand Paging
- 9.3 Copy-on-Write
- 9.4 Page Replacement
- 9.5 Allocation of Frames
- 9.6 Thrashing
- 9.7 Memory-Mapped Files
- 9.8 Allocating Kernel Memory
- 9.9 Other Considerations
- 9.10 Operating-System Examples
- 9.11 Summary

# 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

# 9.1 Background

# Background

- Virtual Memory ： Only part of a running program needs to be loaded into memory for execution
  - Virtual memory separates user logical memory from physical memory
  - Logical (or virtual) address space can be larger than physical address space
  - Allows physical address space to be shared by several processes
  - Enables quicker process creation

What about `fork()`?

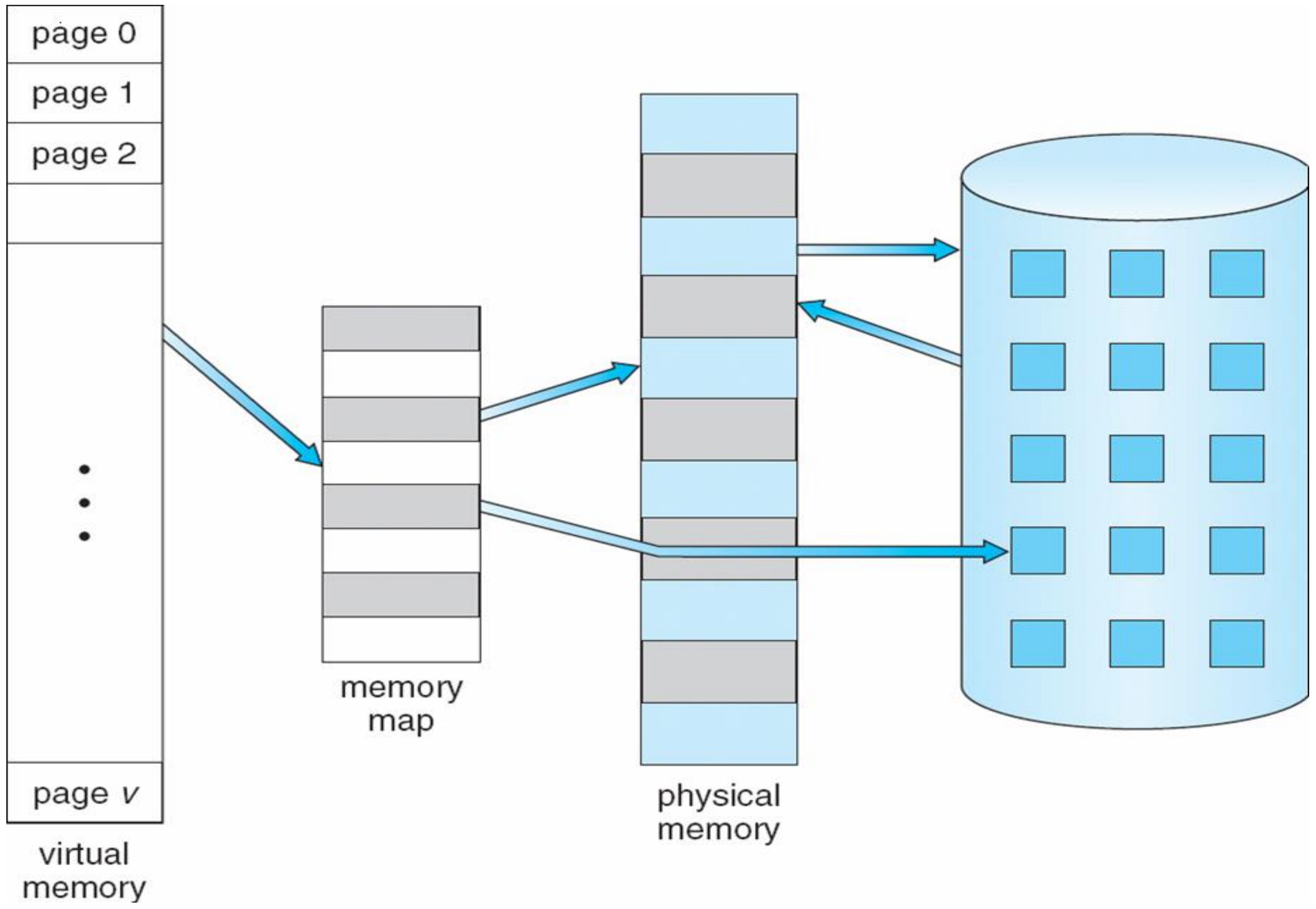
- Virtual memory can be implemented via:
  - Demand paging （请求调页，按需调页，请求页式管理）
  - Demand segmentation（请求段式管理）

# principle of locality

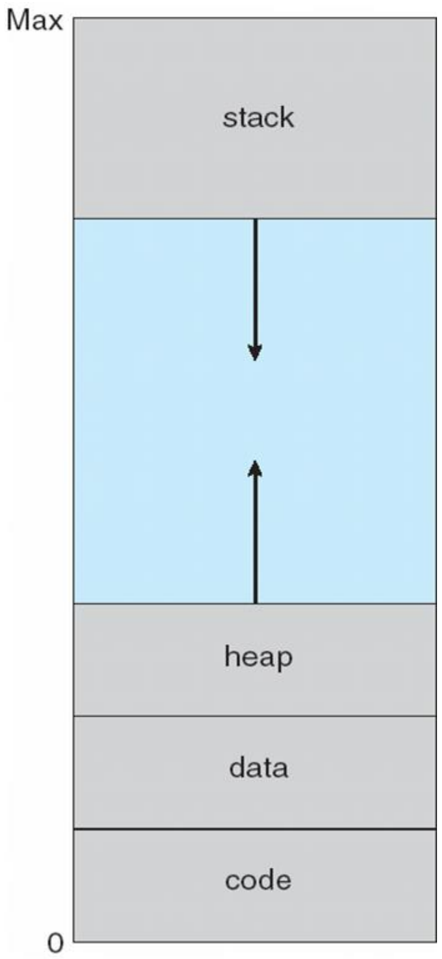
- **局部性原理**(principle of locality)：指程序在执行过程中的一个较短时期，所执行的指令地址和指令的操作数地址，分别局限于一定区域。表现为：
  - **时间局部性**：一条指令的一次执行和下次执行，一个数据的一次访问和下次访问都集中在一个较短时期内；
  - **空间局部性**：当前指令和邻近的几条指令，当前访问的数据和邻近的数据都集中在一个较小区域内。

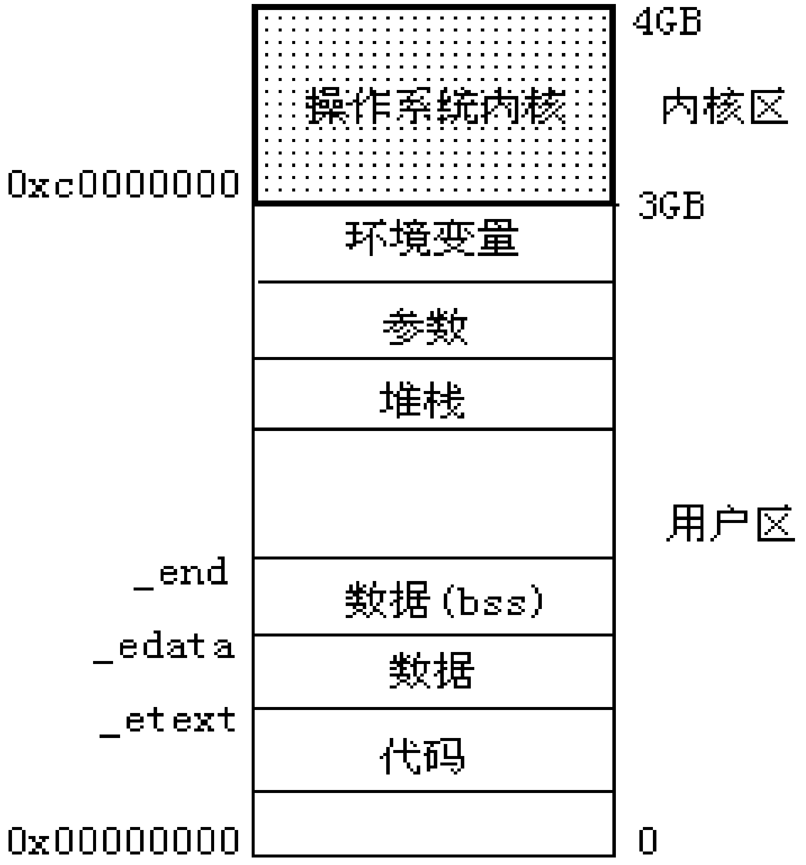
- **虚拟存储器**是具有请求调入功能和置换功能，能仅把进程的一部分装入内存便可运行进程的存储管理系统，它能从逻辑上对内存容量进行扩充的一种虚拟的存储器系统

# Virtual Memory That is Larger Than Physical Memory



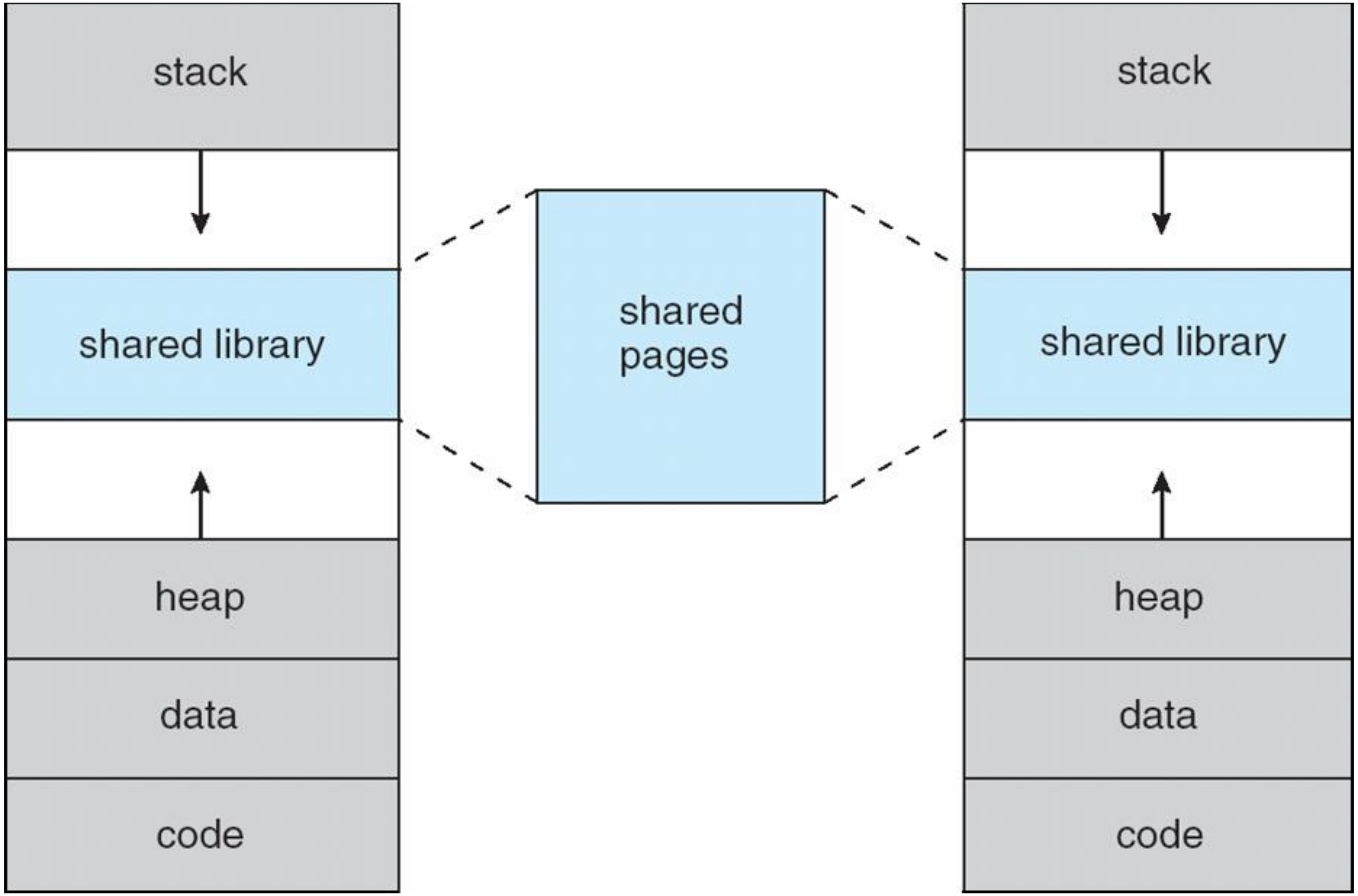
# Virtual-address Space（虚拟地址空间）



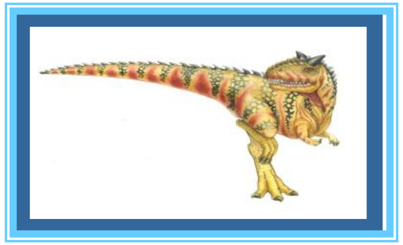




进程的虚存空间

# Shared Library Using Virtual Memory



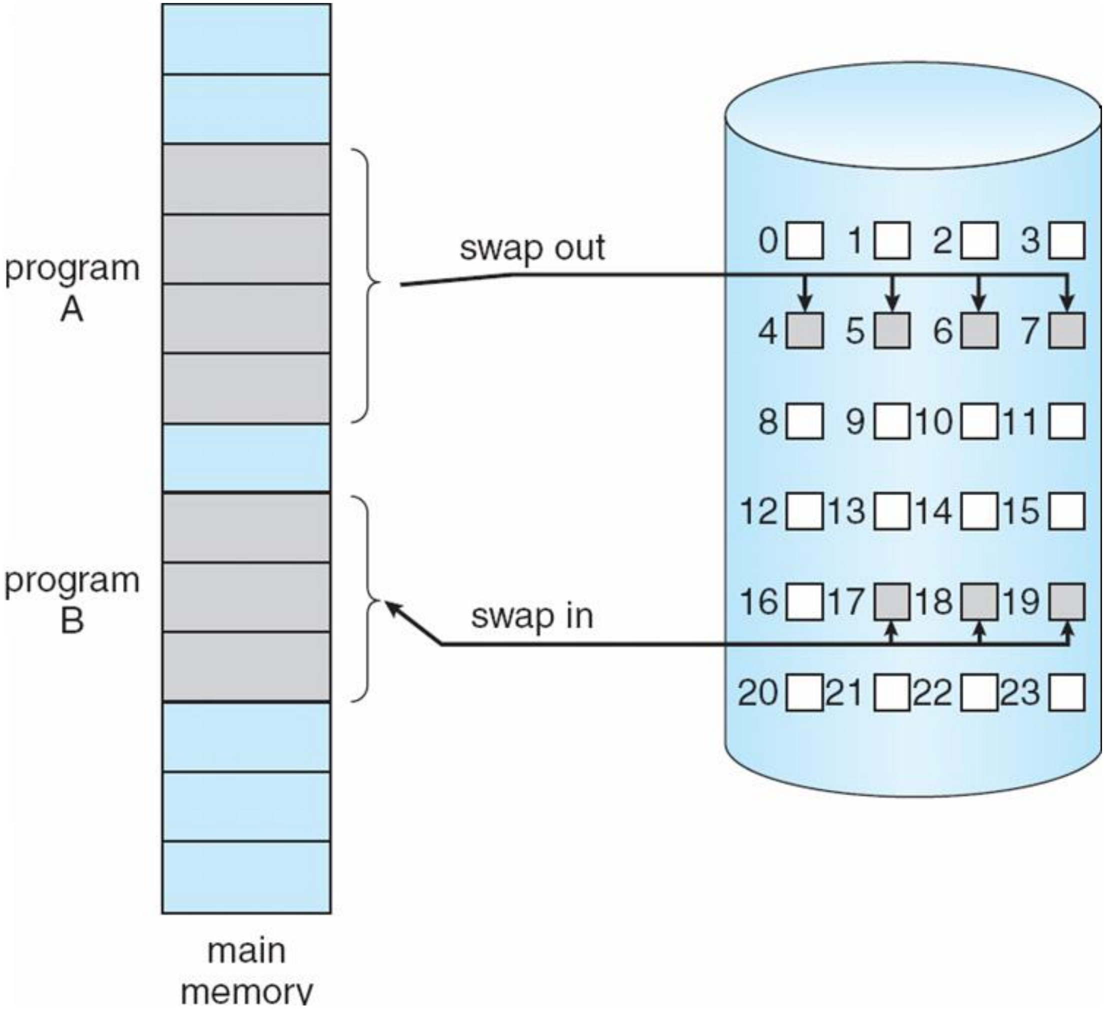
# 9.2 Demand Paging（按需调页、请求调页）



# 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



# Valid-Invalid Bit

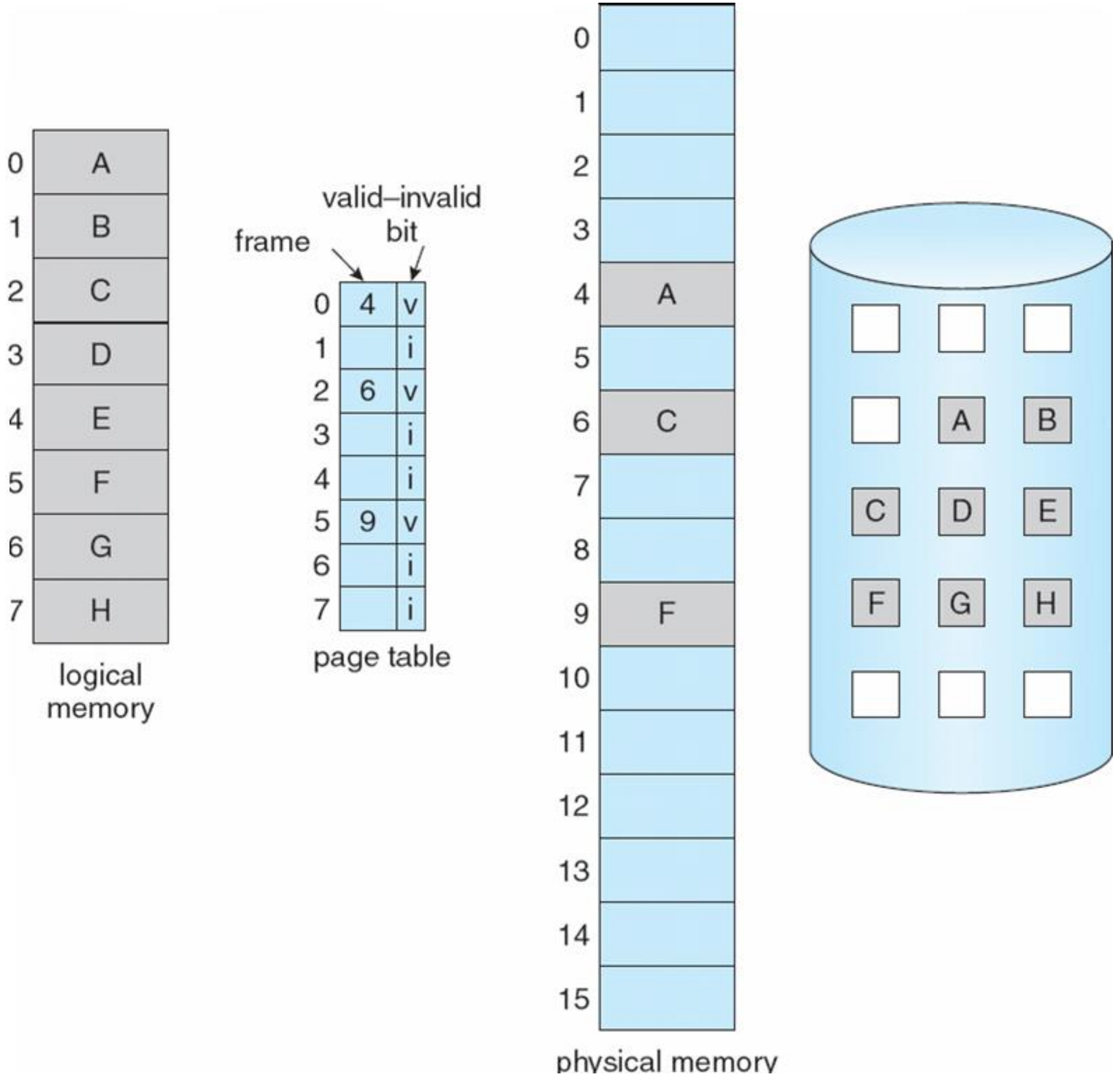
- With each page table entry a valid–invalid bit is associated
  (v ⇒ in-memory, i ⇒ not-in-memory)
- Initially valid–invalid bit is set to i on all entries
- Example of a page table snapshot:

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



# 更完整的页表

- 在请求分页系统中的每个**页表项**如图所示：

| 物理块号 | 状态位P | 访问字段A | 修改位M | 外存地址 |
|---|---|---|---|---|
| | | | | |

- **状态位P**（存在位）：用于指示该页是否已调入内存，供程序访问时参考。
- **访问字段A**：用于记录本页在一段时间内被访问的次数，或最近已有多长时间未被访问，提供给置换算法选择换出页时参考。
- **修改位R/W**：表示该页在调入内存后是否被修改过。
- **外存地址**：用于指出该页在外存上的地址，供调入该页时使用。

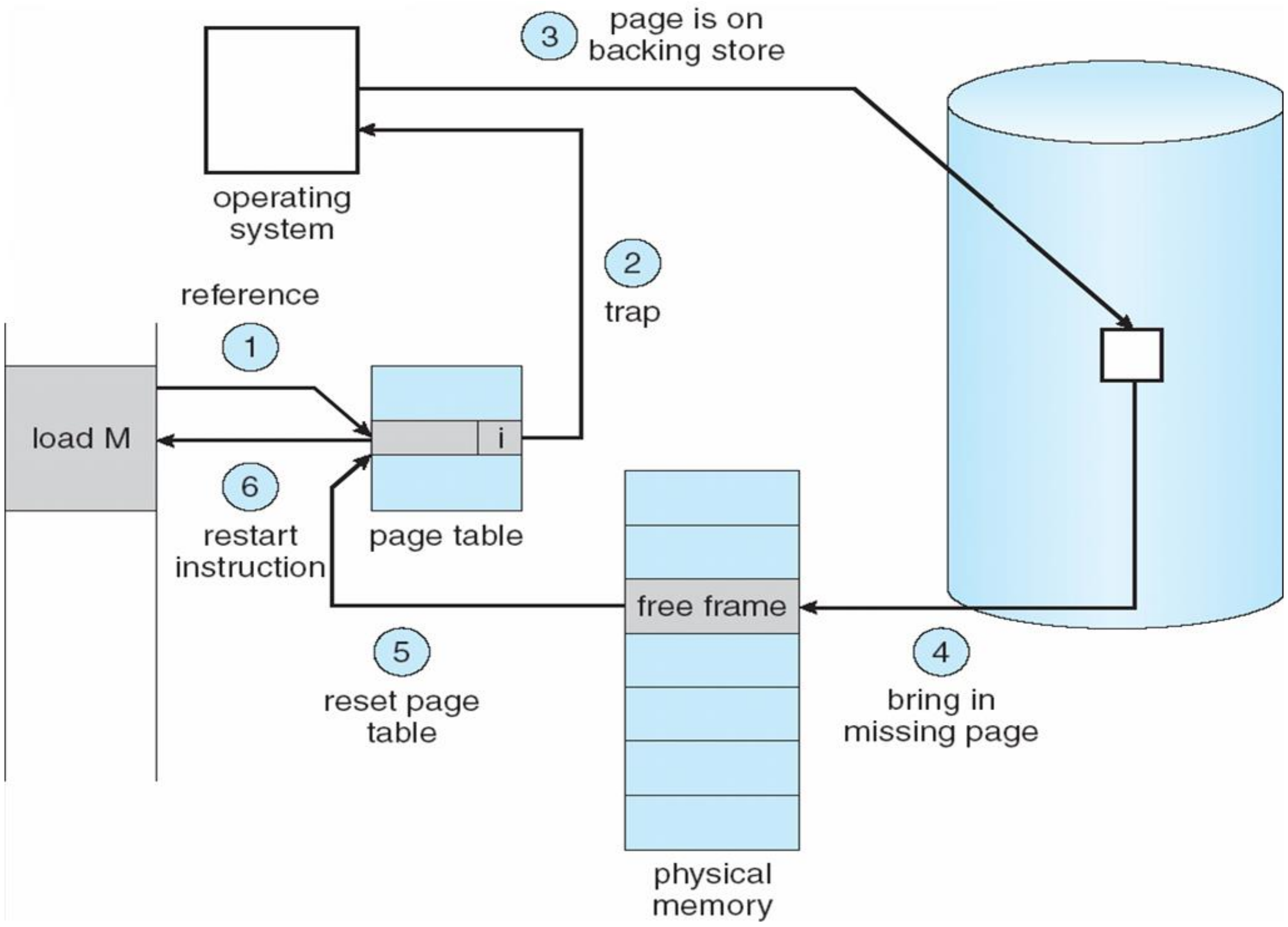
# Page Fault（缺页）

- 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 $\Rightarrow$ abort（非法地址访问）
   - 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



# Demand Paging Performance

- The **page fault rate** *p* is in the range [0.0, 1.0]:
  - If *p* is 0.0, no page faults at all
  - If *p* is 1.0, every page is a page faualt
  - Typically *p* is very low....
- The **effective memory-access time** is
  - (1 – *p*) x physical-memory-access +
    *p* x ( page-fault-overhead +
    swap-page-out +
    swap-page-in +
    restart-overhead )

page-fault
service time

# Performance of Demand Paging

- To compute the **EAT**, we must know how much time is needed to service a page fault. A page fault causes the following sequence to occur:
  1. Trap to the OS.
  2. Save the user 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:
     - Wait in a queue for this device until the read request is serviced.
     - Wait for the device seek time and latency time.
     - Begin the transfer of the page to a free frame.

# Performance of Demand Paging

6. While waiting, allocate the CPU to some other user (CPU scheduling, optional).
7. Interrupt from the disk (I/O completed).
8. Save the registers and process  state for the other user (if step 6 is executed).
9. Determine that the interrupt was from the disk.
10. Correct the page table and other tables 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, then resume the interrupt instruction.

# Performance of Demand Paging

- **Three major components of the page-fault service time**
  - Service the page-fault interrupt(缺页中断服务时间)
  - Read in the page(将缺页读入时间)
  - Restart the process(重新启动进程时间)

# Demand Paging Example

- Memory access time = 200 nanoseconds（*ns*）

**microsecond--us**

- Average page-fault service time = 8 milliseconds（*ms*）

- EAT = (1 – p) x 200 ns + p x8 ms

  = (1 – p  x 200 ns + p x 8,000,000 ns

  = 200 + p x 7,999,800 ns

- If one access out of 1,000 causes a page fault（p=0.001）, then

  EAT = 8200 ns

  This is a slowdown by a factor of 40!!

# 9.3 Process Creation

# Process Creation

- Virtual memory allows other benefits during process creation:
  - **Copy-on-Write（写时拷贝）**

# Copy-on-Write

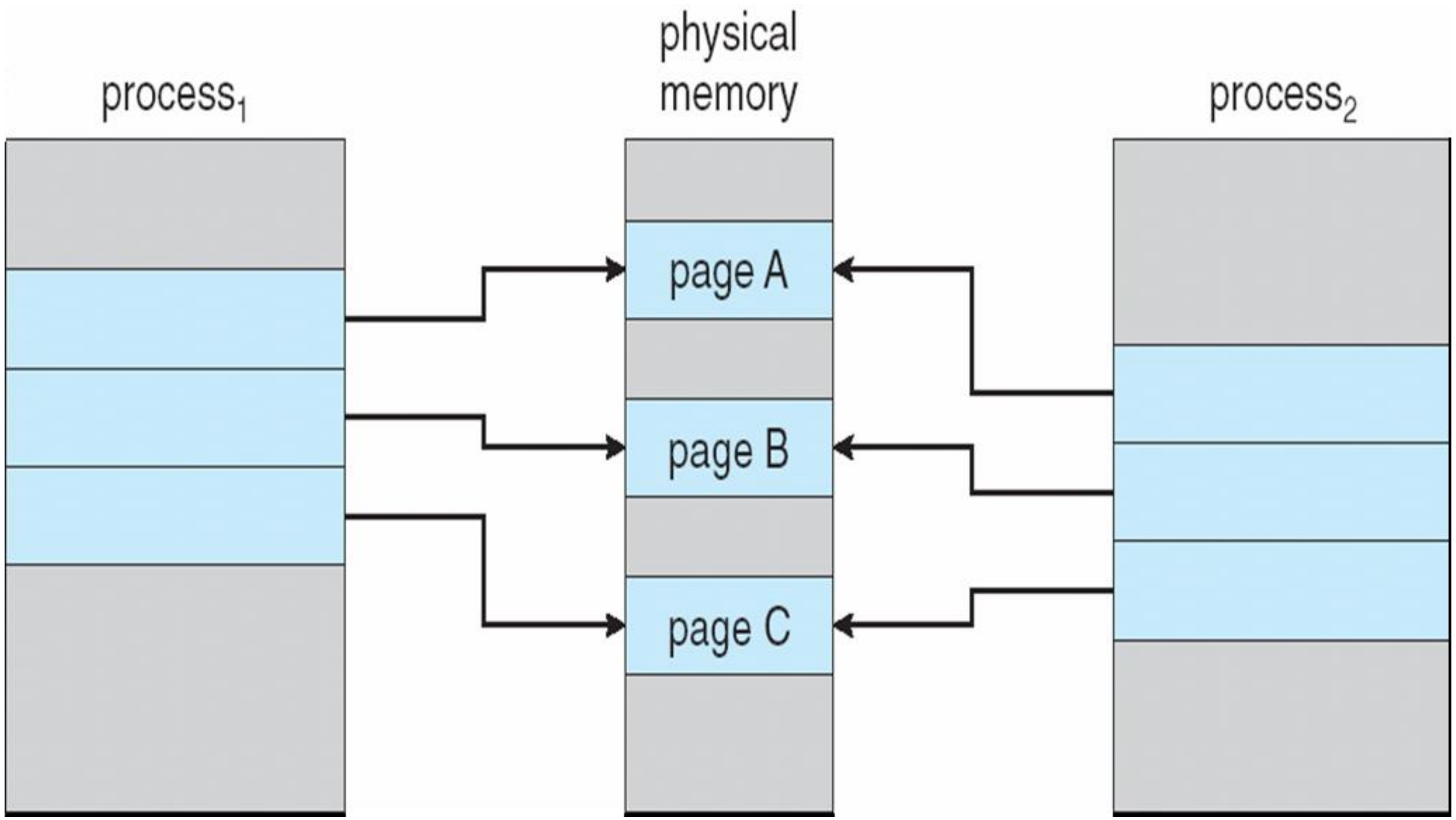
- **Copy-on-Write (COW) allows both parent and child processes to initially *share* the same pages in memory**
  **If either process modifies a shared page, only then is the page copied**

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

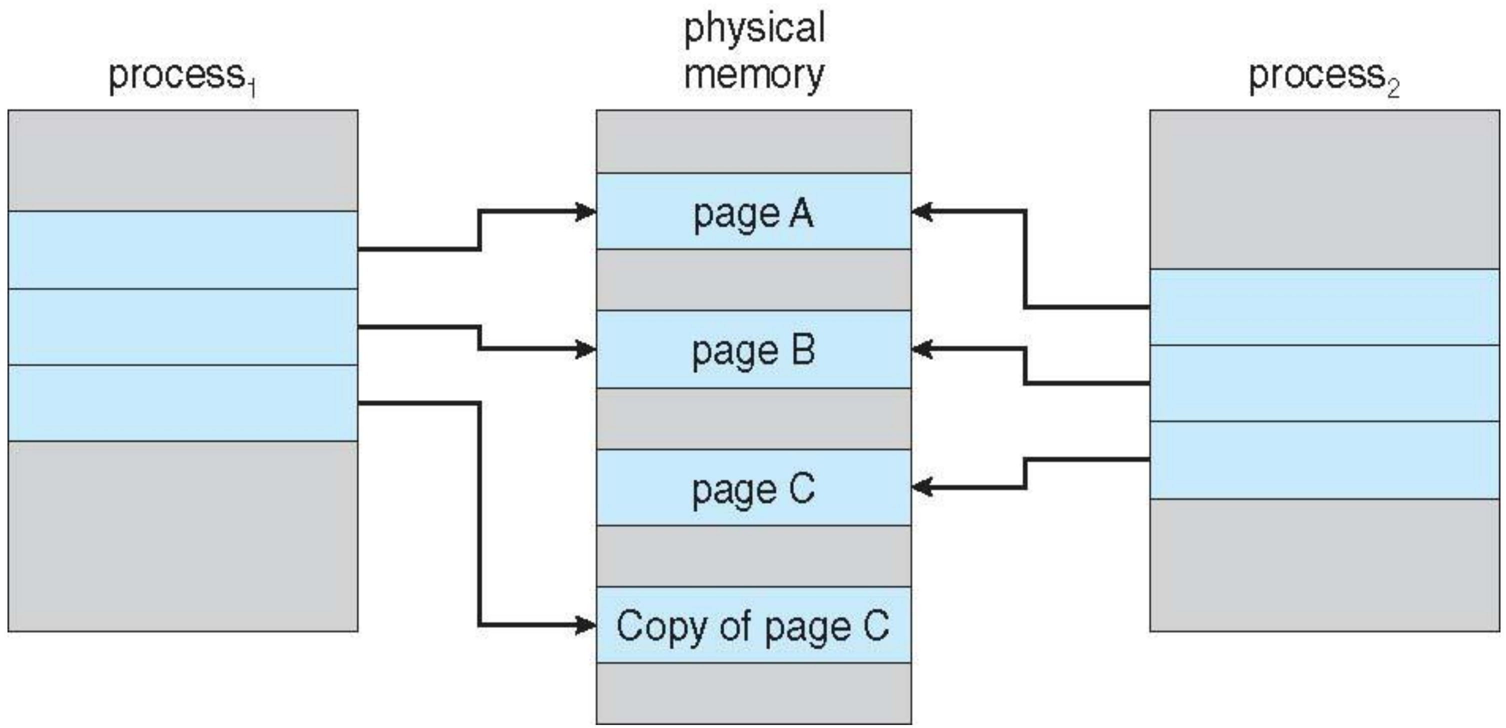
- Free pages are allocated from a pool of zeroed-out pages

- Windows、Linux、Solaris

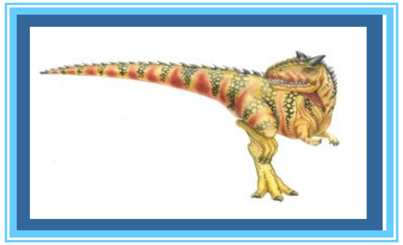
# Before Process 1 Modifies Page C



# After Process 1 Modifies Page C



# 9.4 Page Replacement（页面置换）



# Page Replacement 页面置换

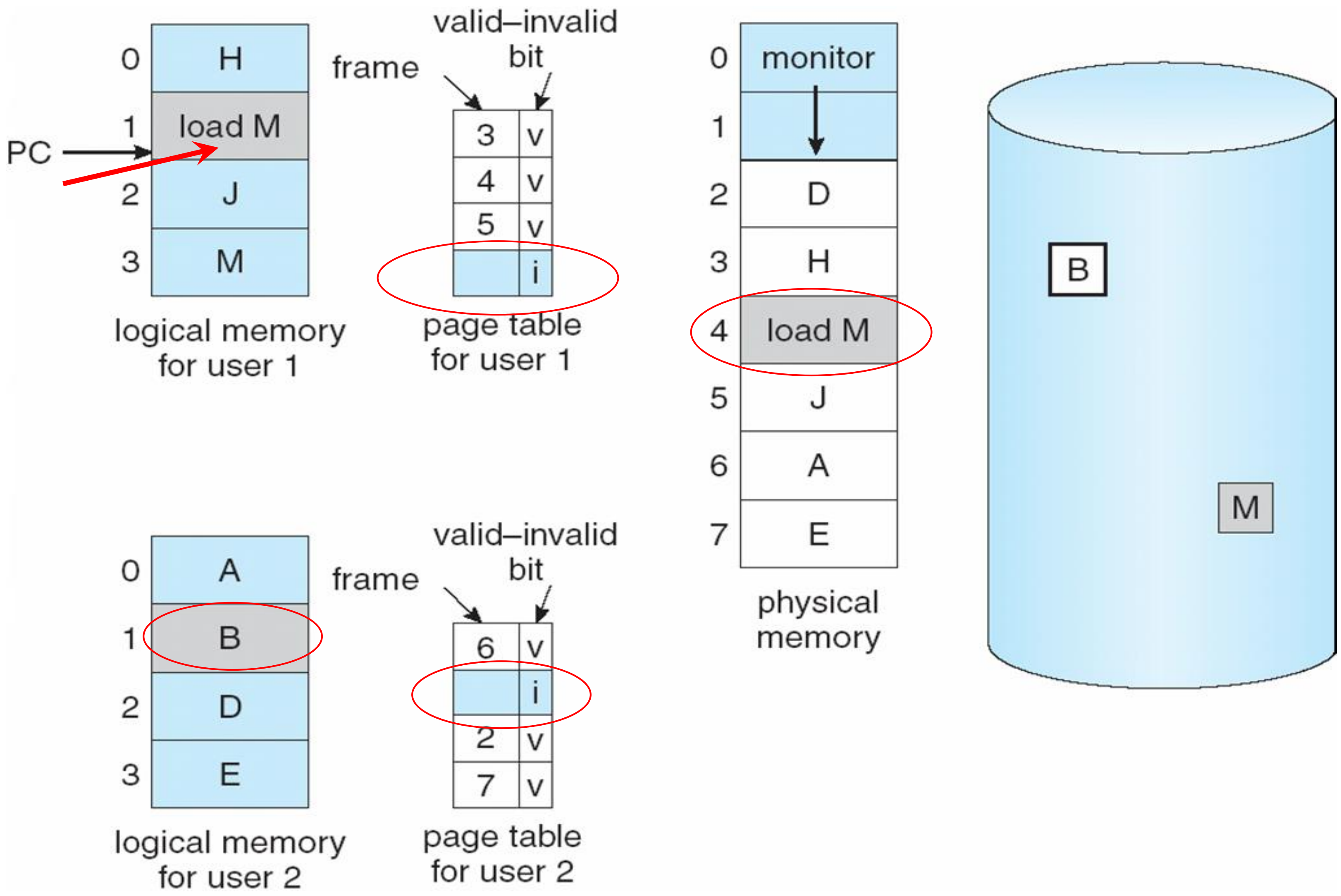
**What happens if there is no free frame?**

- **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

# 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

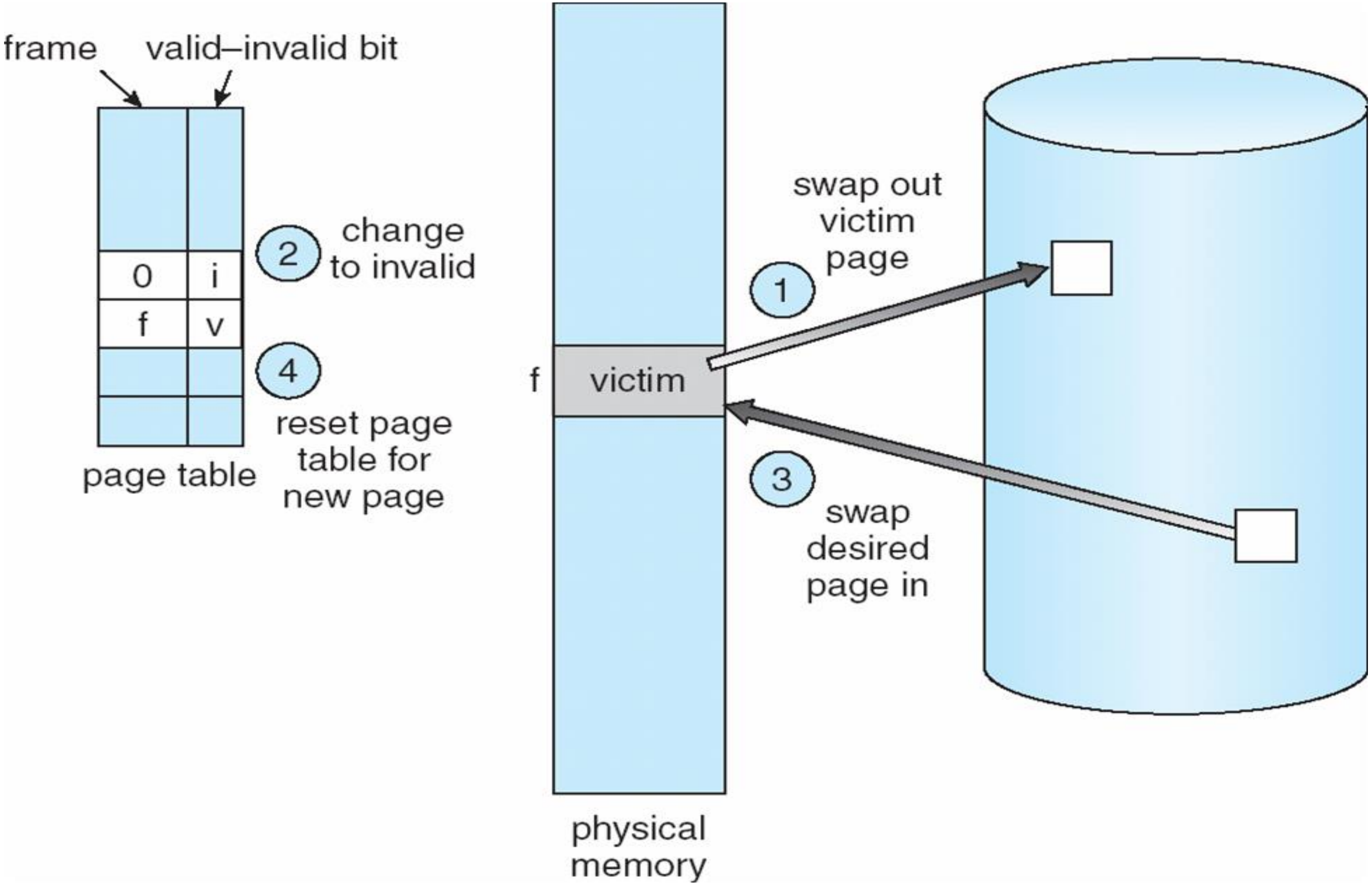
# Need For Page Replacement



# Basic Page Replacement

- 页面置换的过程：
  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 (淘汰) frame
     - Write the victim page to the disk; change the page and frame tables accordingly.
  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，** 引用串) and computing the number of page faults on that string
- reference string:(100 bytes per page)

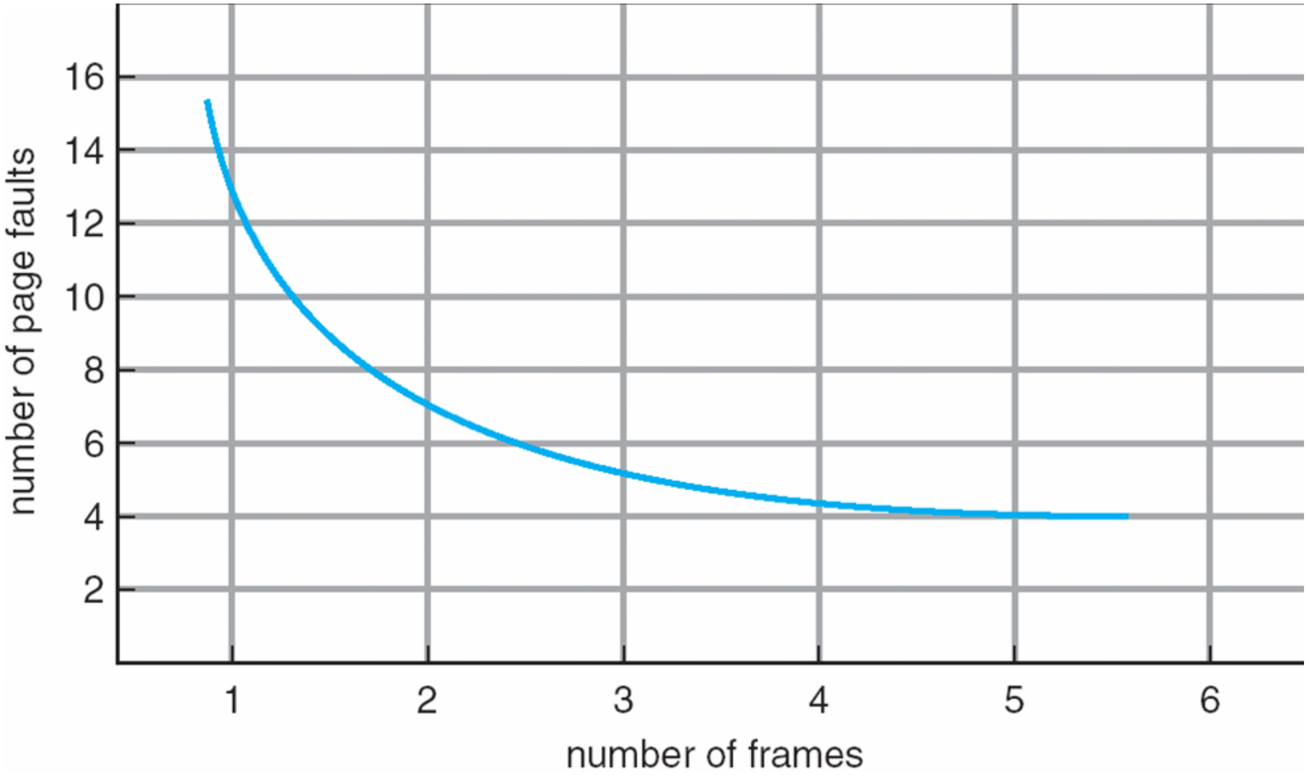
  0100, 0432,0101,0612, 0102,0103,0104,0611,0120 ➔ 1,4,1,6,1,6,1

- In all our examples, the reference string is

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

  7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1.、

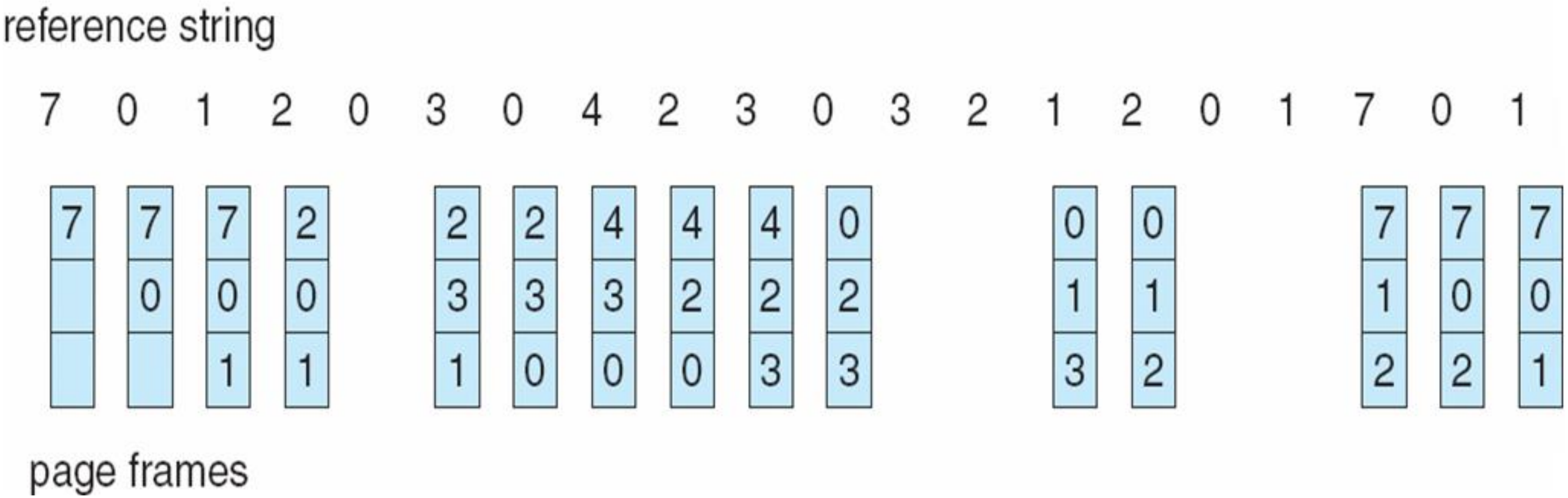
# Graph of Page Faults Versus The Number of Frames



# Page Replacement Algorithms

- First-In-First-Out Algorithm (**FIFO**，先进先出算法)
- Optimal Algorithm （**OPT** 最佳页面置换算法）
- Least Recently Used (**LRU**) Algorithm (最近最久使用算法)
- LRU Approximation Algorithms （近似LRU算法）：
  - Additional-Reference-Bits Algorithm
  - Second-Chance（clock） Algorithm
  - Enhanced Second-Chance Algorithm
- Counting-Base Page Replacement：
  - Least Frequently Used Algorithm (LFU最不经常使用算法）
  - Most Frequently Used Algorithm (MFU引用最多算法)
- Page Buffering Algorithm（页面缓冲算法）

# FIFO Page Replacement




How many page faults occur?

**15 Page 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 |
| 3 | 3 | 2 | 4 |

9 page faults

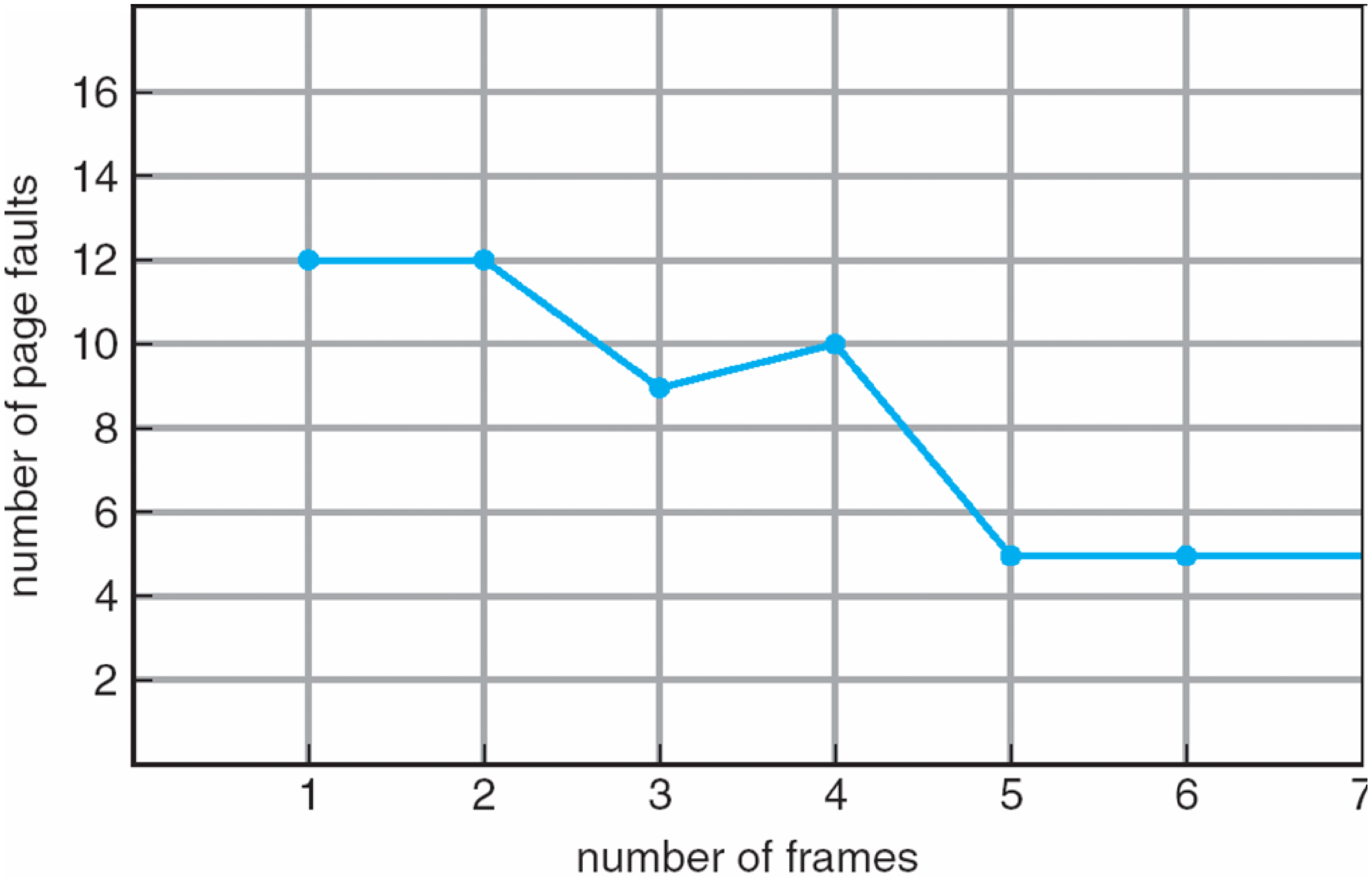
- **4 frames**

| | | | |
|---|---|---|---|
| 1 | 1 | 5 | 4 |
| 2 | 2 | 1 | 5 |
| 3 | 3 | 2 | |
| 4 | 4 | 3 | |

10 page faults

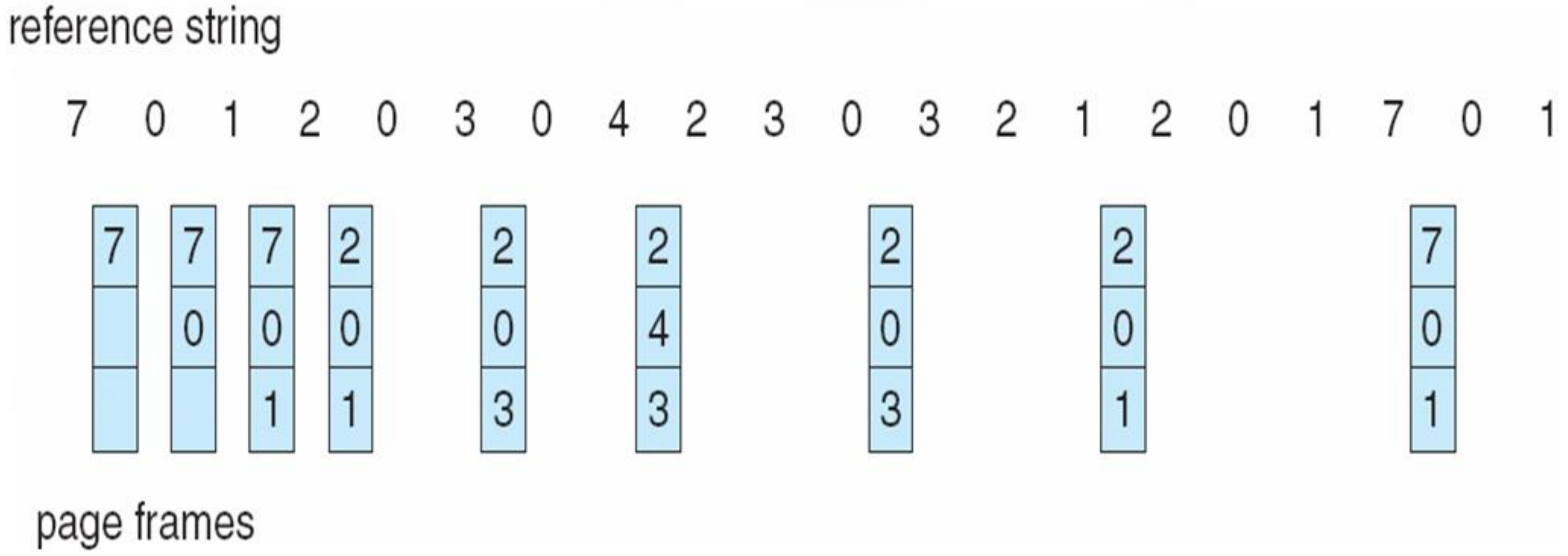
- **Belady's Anomaly: more frames $\Rightarrow$ more page faults**

# FIFO Illustrating Belady's Anomaly



# Optimal Page Replacement

- **OPT(最佳页面置换算法)**：Replace page that will not be used for longest period of time。选择“未来不再使用的”或“在离当前最远位置上出现的”页被置换。
- How do you know this?
- Used for measuring how well your algorithm performs




**9 Page faults**

# Optimal Algorithm

- 4 frames example

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

| | | |
|---|---|---|
| 1 | | 4 |
| 2 | | |
| 3 | | |
| 4 | 5 | |

6 page faults

## Least Recently Used (LRU)（最近最久使用）Algorithm

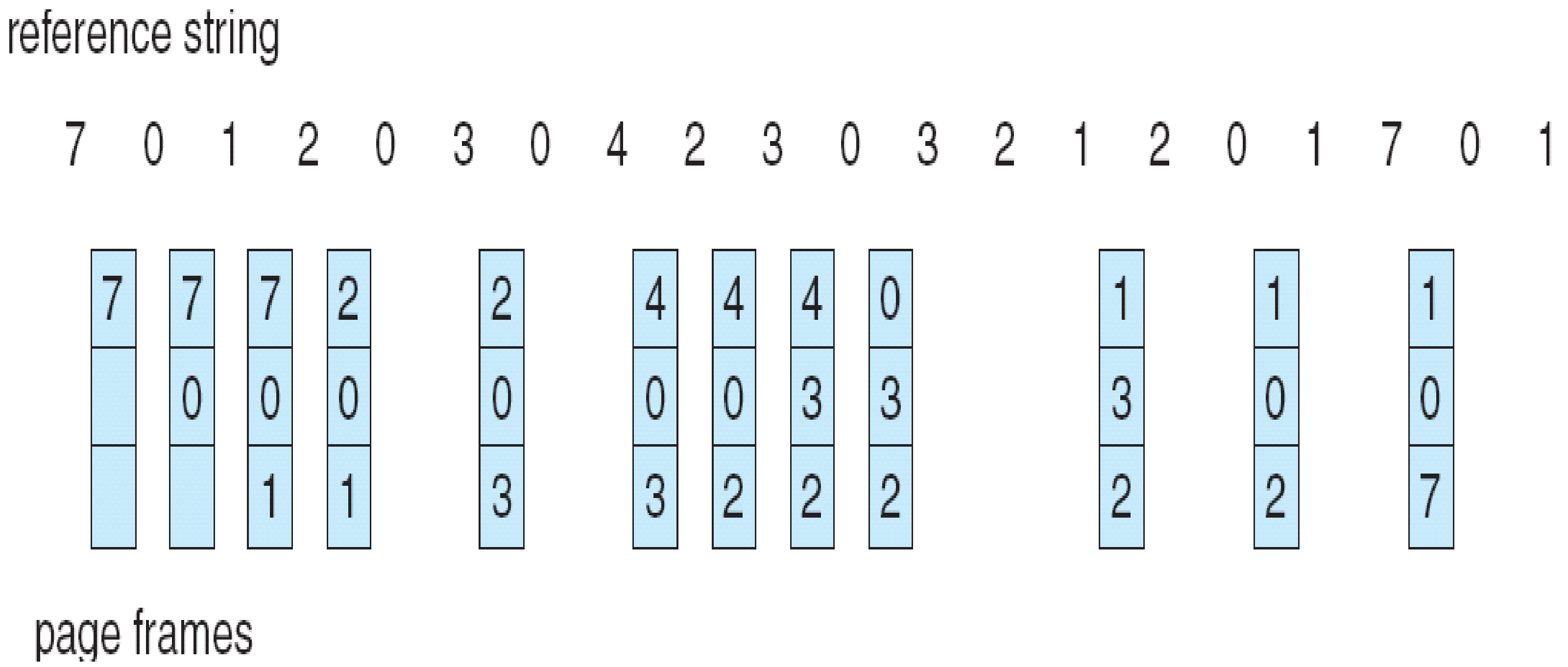
- LRU(最近最少使用算法)：选择内存中最久没有引用的页面被置换。这是局部性原理的合理近似，性能接近最佳算法。但由于需要记录页面使用时间，硬件开销太大。

- 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 |

**8 Page faults**

# LRU Page Replacement

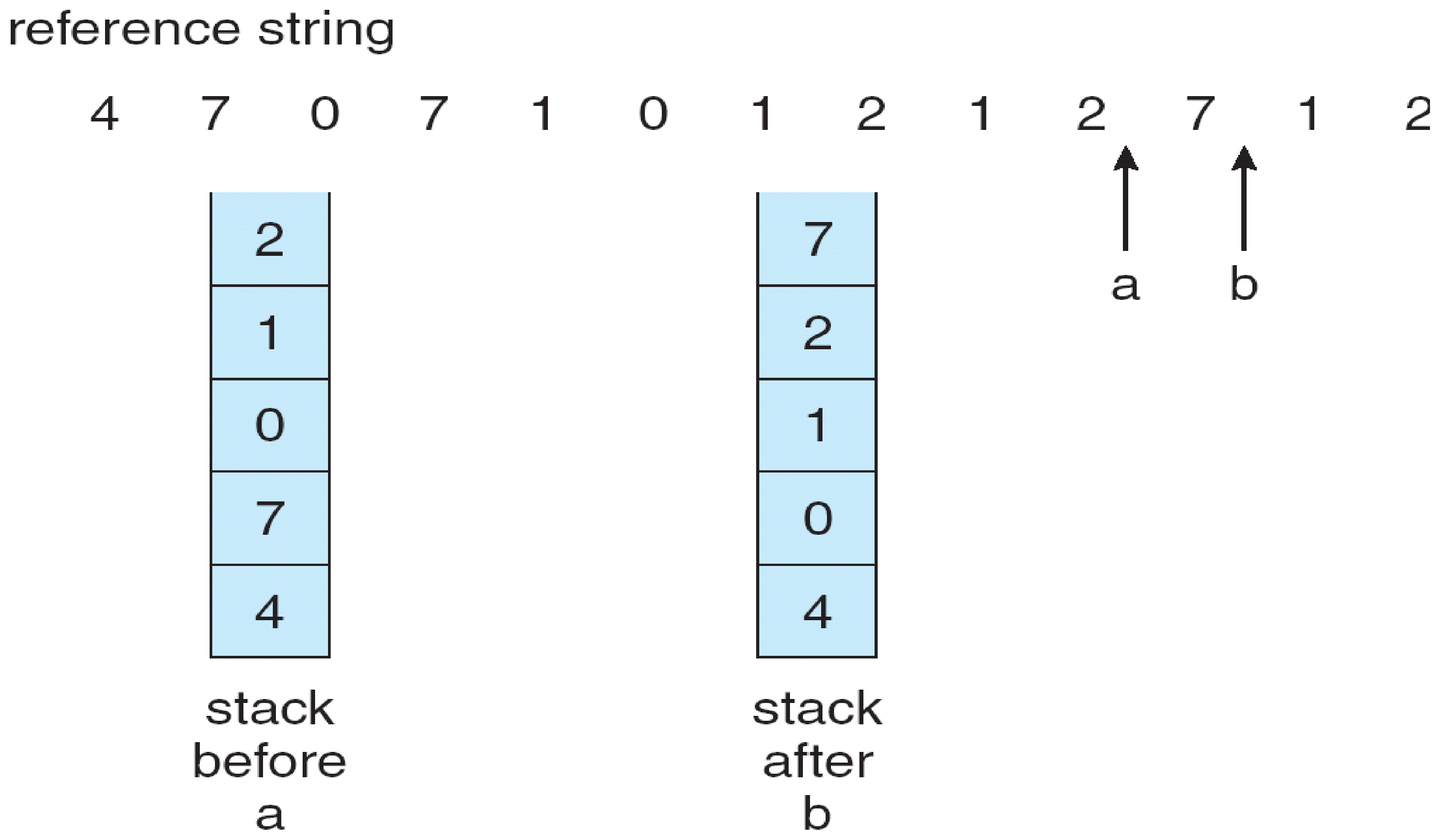



**12 Page faults**

# LRU算法,如何获知“多长时间没引用”?

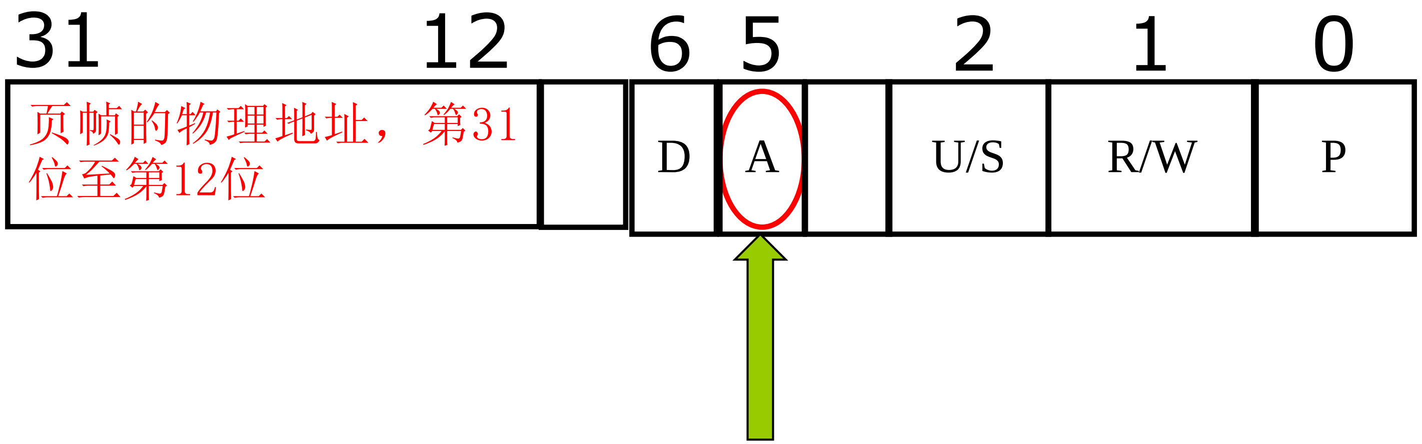
- **Counter implementation**
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to determine which are to change
- **Stack implementation** – keep a stack of page numbers in a double link form:
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed
  - No search for replacement

# Use Of A Stack to Record The Most Recent Page References



# LRU Approximation Algorithms

- 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



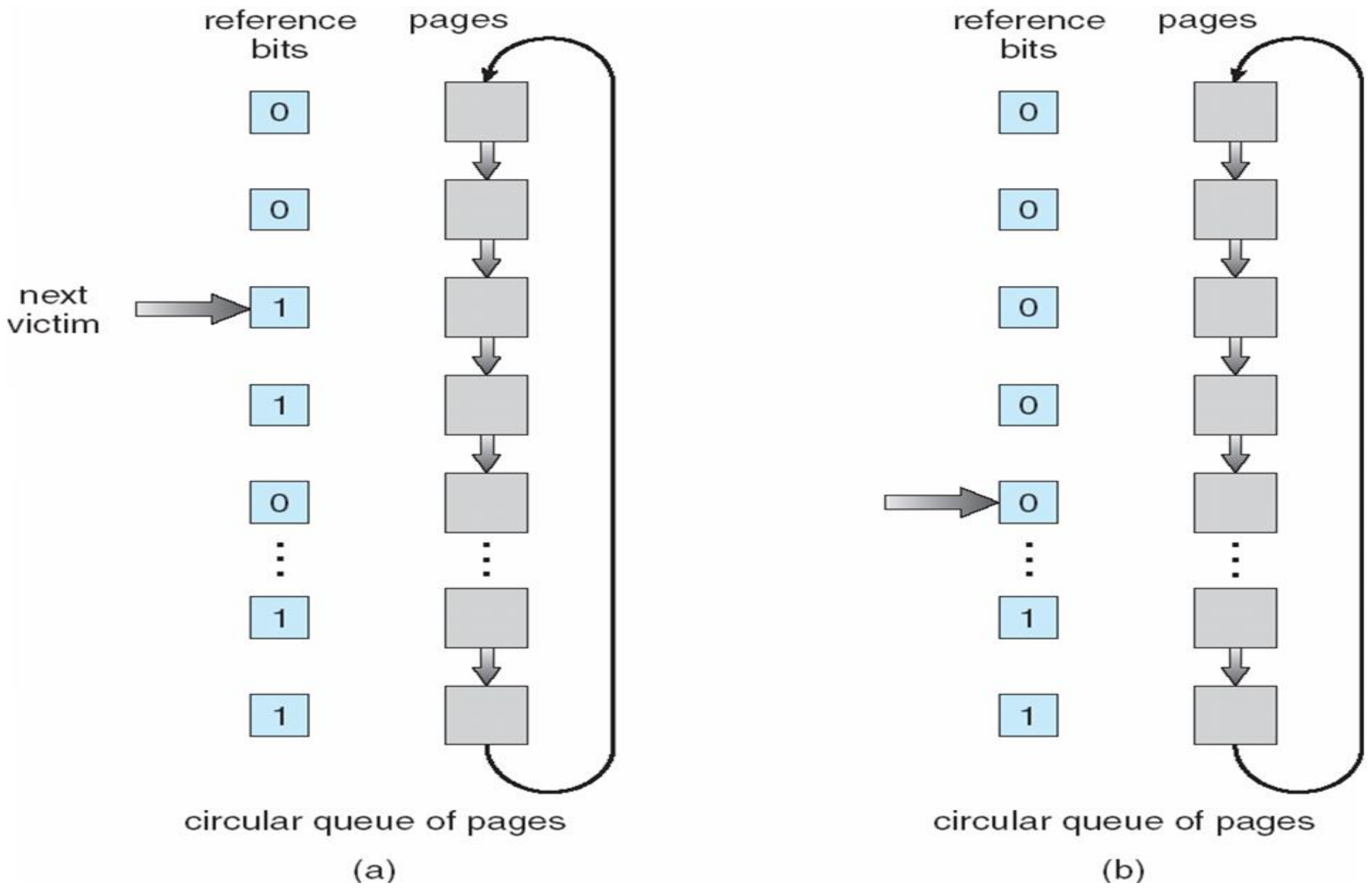
# 1. Additional-Reference-Bits Algorithm

- 附加引用位算法：
  - To keep **an 8-bit byte for each page** in a table in memory
  - At regular intervals (every 100 ms), a timer interrupt transfers control to the OS. The OS shifts the reference bit for each page into the high-order bit of its 8-bit byte, shifting the other bits right 1 bit, discarding the low-order bit. These 8-bit bytes contain the history of the page use for the last eight time periods.
  - If we interpret these 8-bit bytes as unsigned integers, the page with the lowest number is the LRU page and it can be replaced.
  - 被访问时左边最高位置1，定期右移并且最高位补0，于是寄存器数值最小的是最久未使用页面。

# 2. Second-Chance (clock) Algorithm

- Second chance(clock算法)
  - 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 Page-Replacement Algorithm



# 3. Enhanced Second-Chance Algorithm

- 增强二次机会算法（改进型的clock算法）
  - 使用引用位和修改位：引用过或修改过置成1
  - (Reference bit, modified bit) ：
    - (0,0): best page to replace
    - (0,1): not quite good for replacement
    - (1,0): will be used soon
    - (1,1): worst page to replace.
  - 淘汰次序:(0,0) ⇒(0,1)⇒(1,0)⇒(1,1)

- Macintosh系统中使用

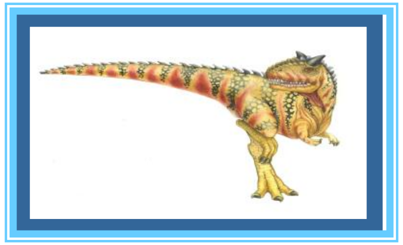
# Counting Algorithms

- Keep a counter of the number of references that have been made to each page
- LFU(Least Frequently Used) Algorithm(最不经常使用算法): replaces page with smallest count
- MFU(Most Frequently Used) Algorithm(经常使用算法): based on the argument that the page with the smallest count was probably just brought in and has yet to be used

# Page Buffering Algorithm页面缓冲算法

- **页面缓冲算法**：通过被置换页面的缓冲，有机会找回刚被置换的页面
  - 被置换页面的选择和处理：用FIFO算法选择被置换页，把被置换的页面放入两个链表之一。即：如果页面未被修改，就将其归入到空闲页面链表的末尾，否则将其归入到已修改页面链表。
  - 需要调入新的页面时，将新页面内容读入到空闲页面链表的第一项所指的页面，然后将第一项删除。
  - 空闲页面和已修改页面，仍停留在内存中一段时间，如果这些页面被再次访问，这些页面还在内存中。
  - 当已修改页面达到一定数目后，再将它们一起调出到外存，然后将它们归入空闲页面链表。
- VAX/VMS系统使用
- Windows、Linux页面置换算法是基于页面缓冲算法。

# 9.5 Allocation of Frames(帧分配)



# Allocation of Frames(帧分配)

- 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（固定分配）

- Equal allocation（平均分配算法） – For example, if there are 100 frames and 5 processes, give each process 20 frames.
- Proportional allocation （按比例分配算法） – Allocate according to the size of process

$$
\begin{aligned}
&- s_i = \text{size of process } p_i \\
&- S = \sum s_i \\
&- m = \text{total number of frames} \\
&- a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m
\end{aligned}
$$

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

# Priority Allocation（优先级分配）

- 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

置换策略：

- Global replacement（全局置换） – process selects a replacement frame from the set of all frames; one process can take a frame from another
- Local replacement （局部置换）– each process selects from only its own set of allocated frames

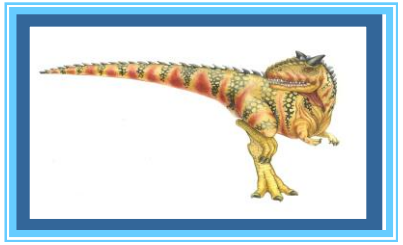
分配策略：

- 固定分配
- 可变分配

# 帧的分配和置换策略

- 组合成三种策略：
  - 固定分配局部置换策略
  - 可变分配全局置换策略
  - 可变分配局部置换

# 9.6 Thrashing （颠簸、抖动）

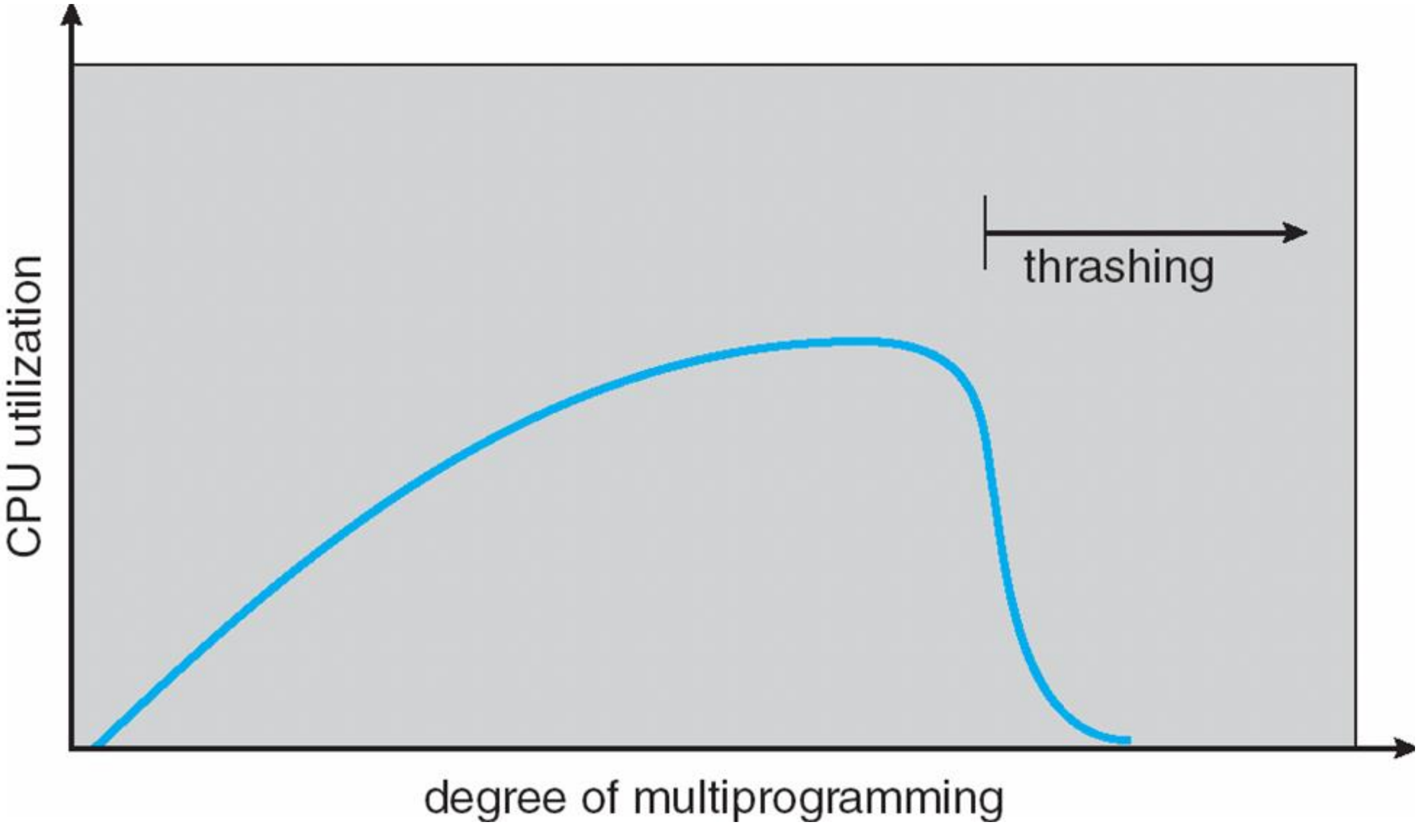


# Thrashing （颠簸、抖动）

- 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.)

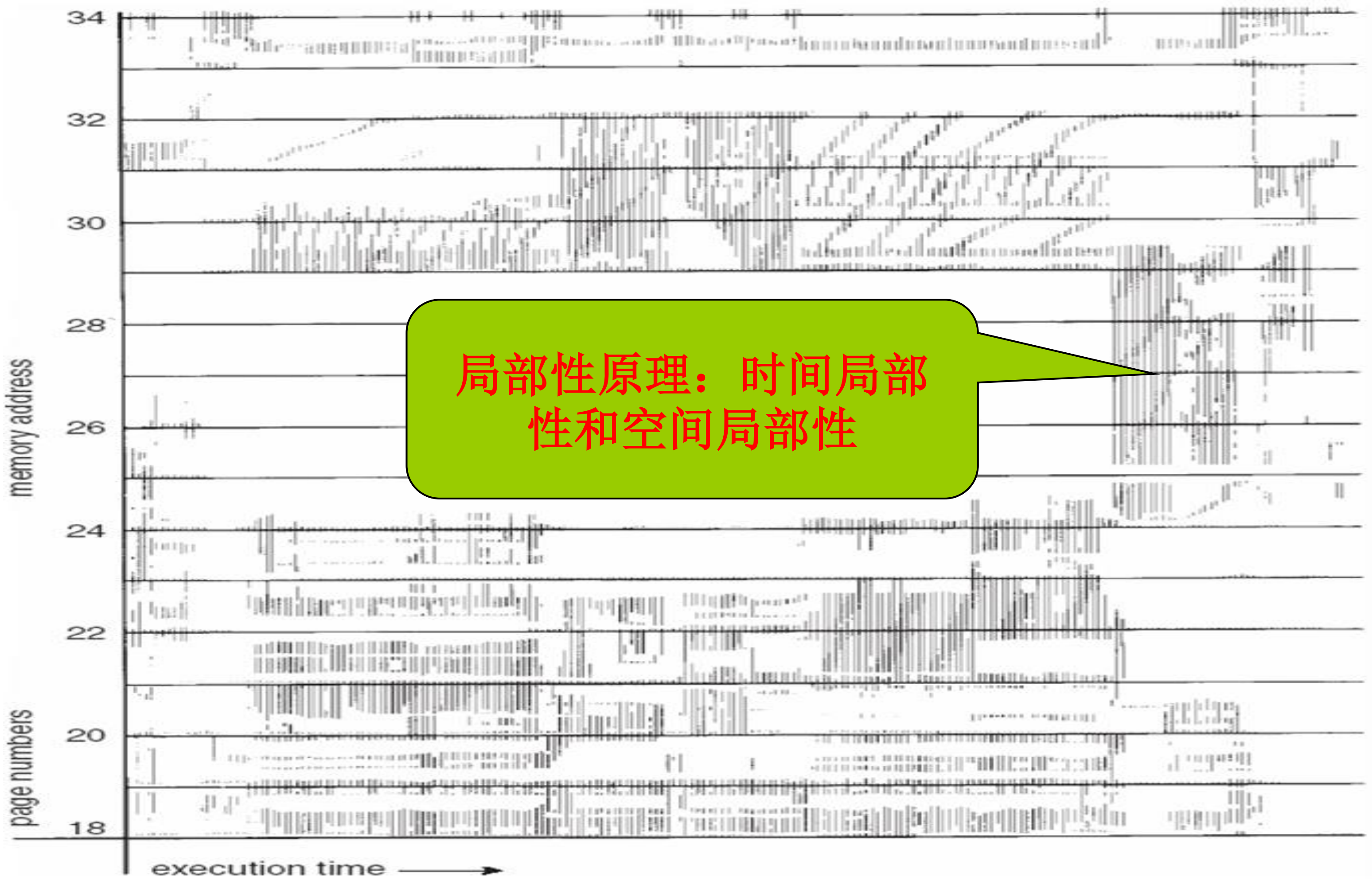


# Demand Paging and Thrashing

- Why does demand paging work?
  Locality（局部性） model
  - Process migrates from one locality to another
  - Localities may overlap

- Why does thrashing occur?
  $\Sigma$ size of locality > total memory size

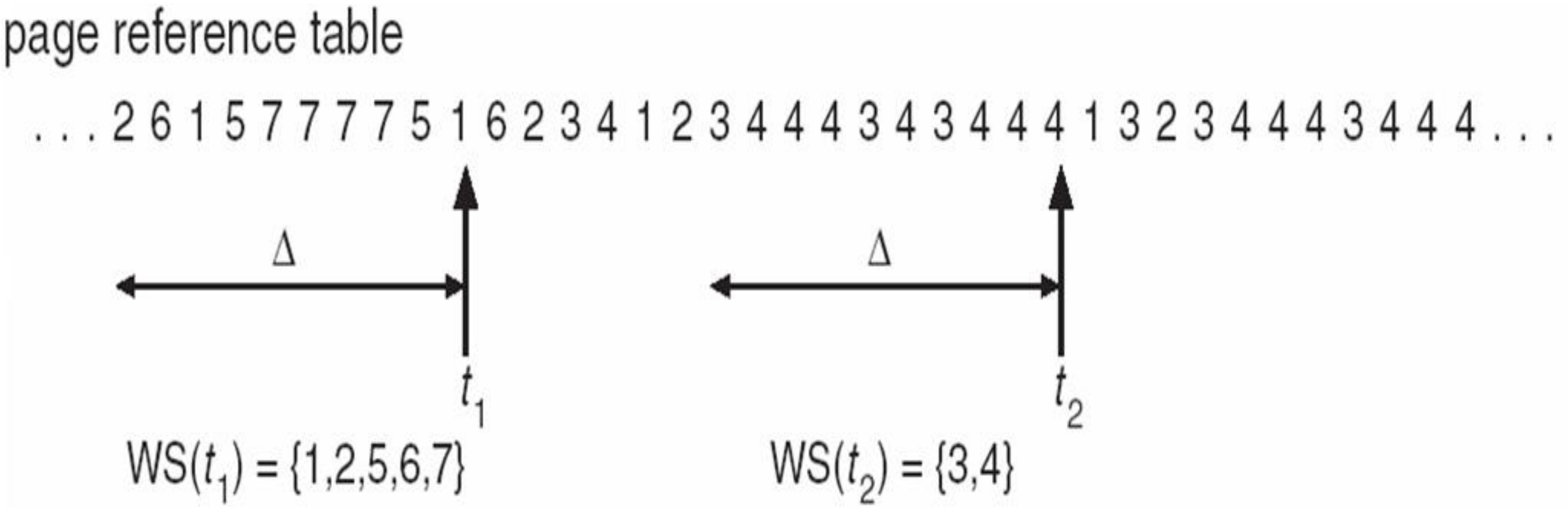
# Locality In A Memory-Reference Pattern



# Working-Set Model

- working set (WS)工作集：The set of pages in the most recent $\Delta$ page references
- $\Delta \equiv$ working-set window(工作集窗口) $\equiv$ a fixed number of page references
  Example: 10,000 instruction
- $WSS_i$ (working set size of Process $P_i$ 工作集大小) =
  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.
- $D = \Sigma\, WSS_i \equiv$ total demand frames; m $\equiv$ total available frames
- if $D > m \Rightarrow$ Thrashing
- **Policy if $D$ > m, then suspend one of the processes.**

# Working-set model




***WSS=5***

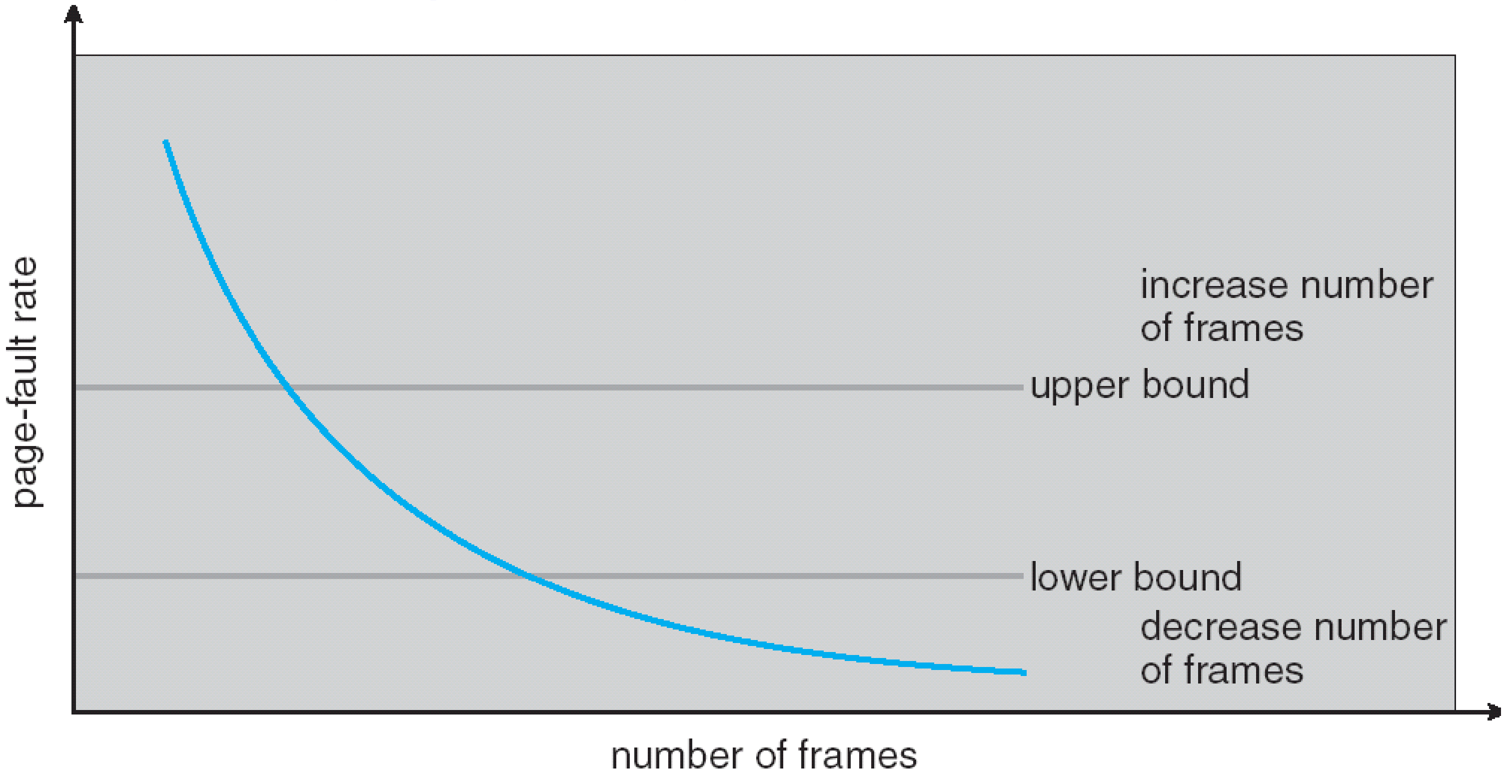
***WSS=2***

# Keeping Track of the Working Set

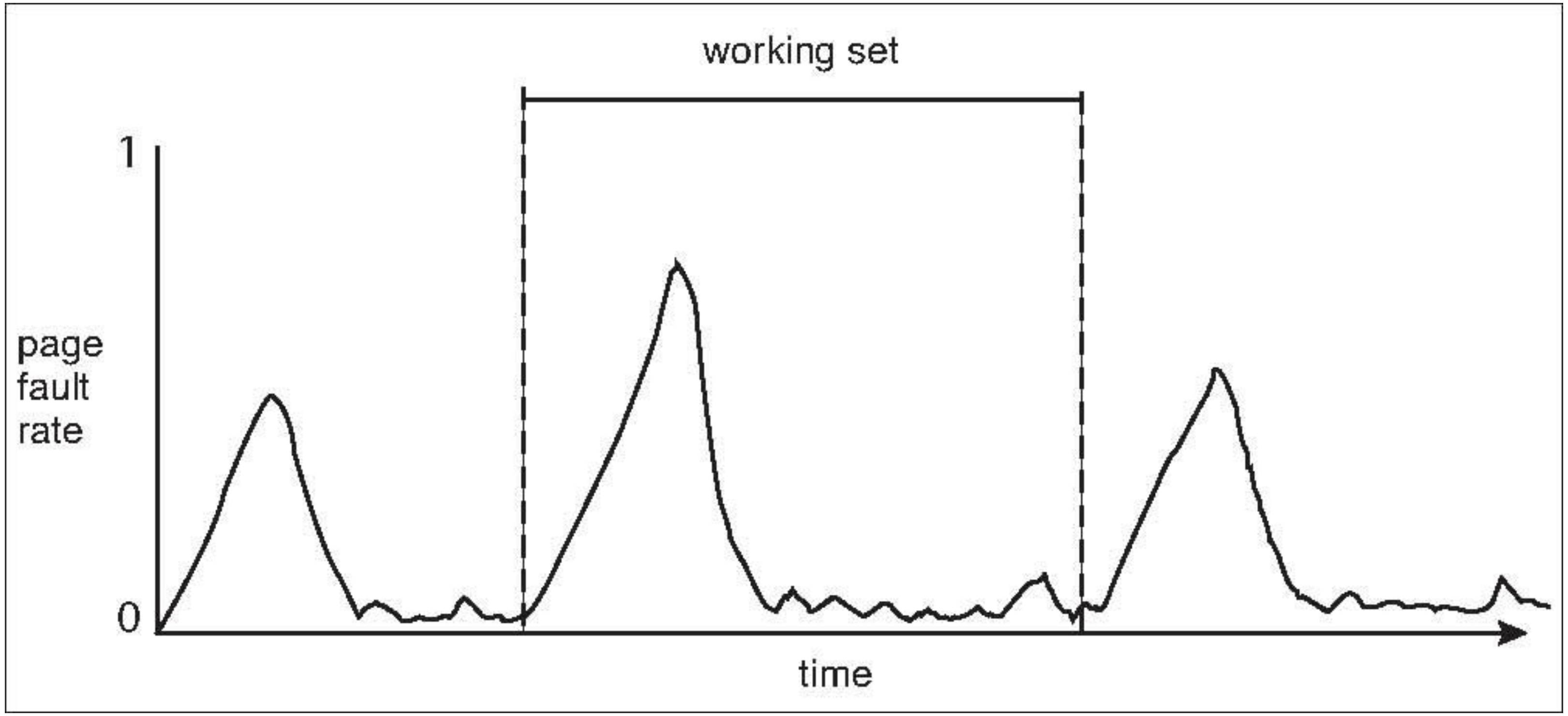
- Approximate with interval timer + a reference bit
- Example: $\Delta = 10{,}000$
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 bits for each page
  - Whenever a timer interrupts copy and sets the values of all reference bits to 0
  - If one of the bits in memory = 1 $\Rightarrow$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

# Page-Fault Frequency Scheme（缺页频率）

- 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



# 9.7 Memory-Mapped Files

# Memory-Mapped Files

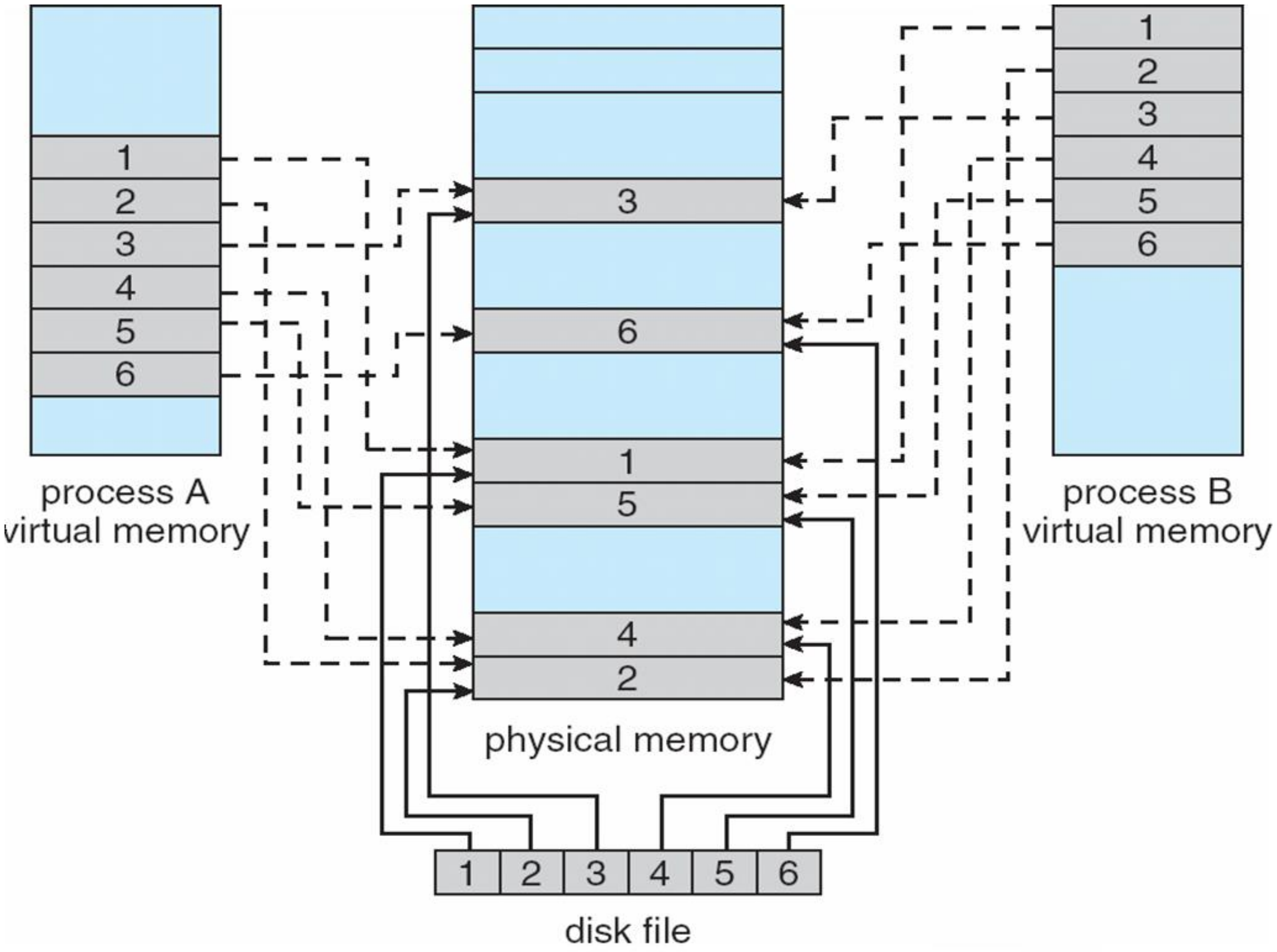
- Memory-mapped file I/O allows file I/O to be treated as routine memory access by mapping a disk block to a page in memory

- 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.

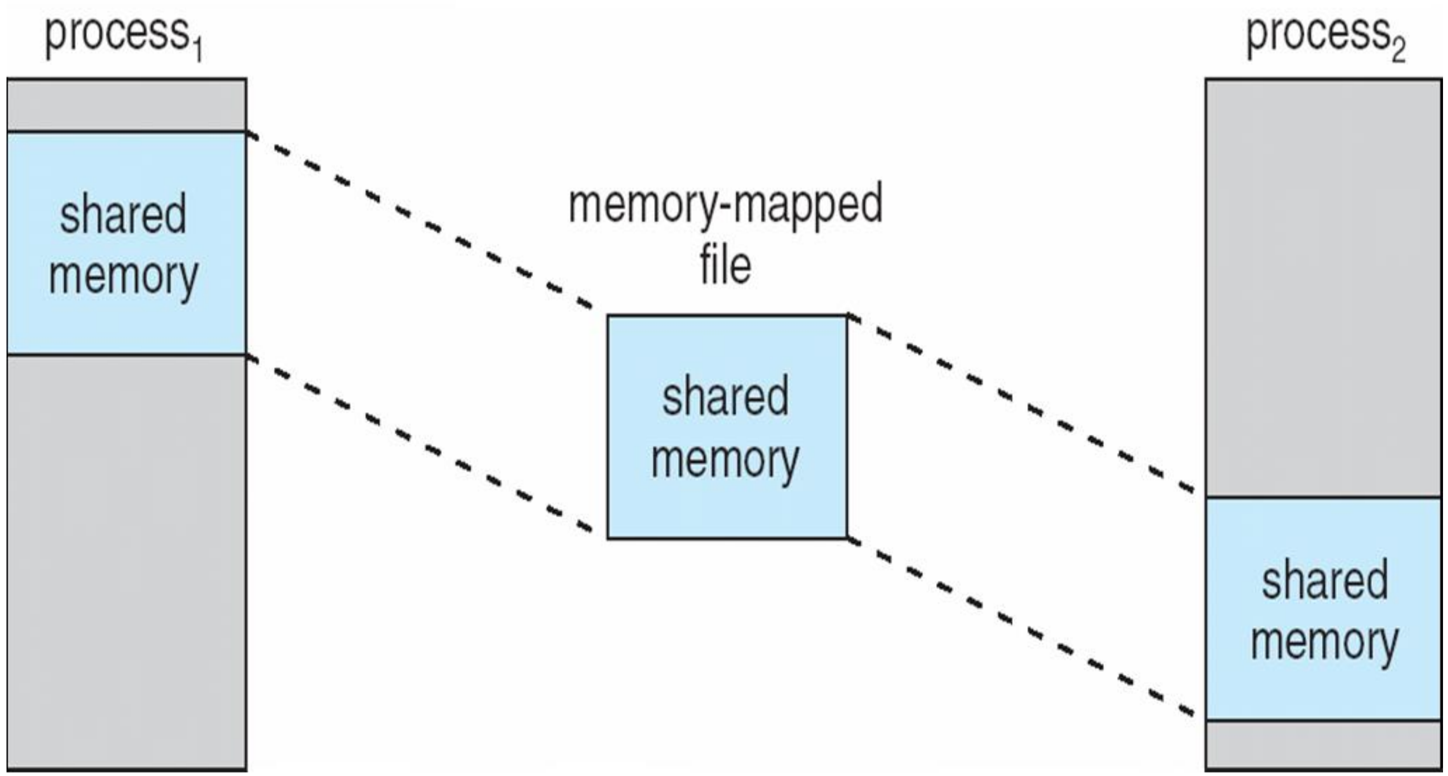
- Simplifies file access by treating file I/O through memory rather than **`read()`** **`write()`** system calls

- 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



# 9.8 Allocating Kernel Memory

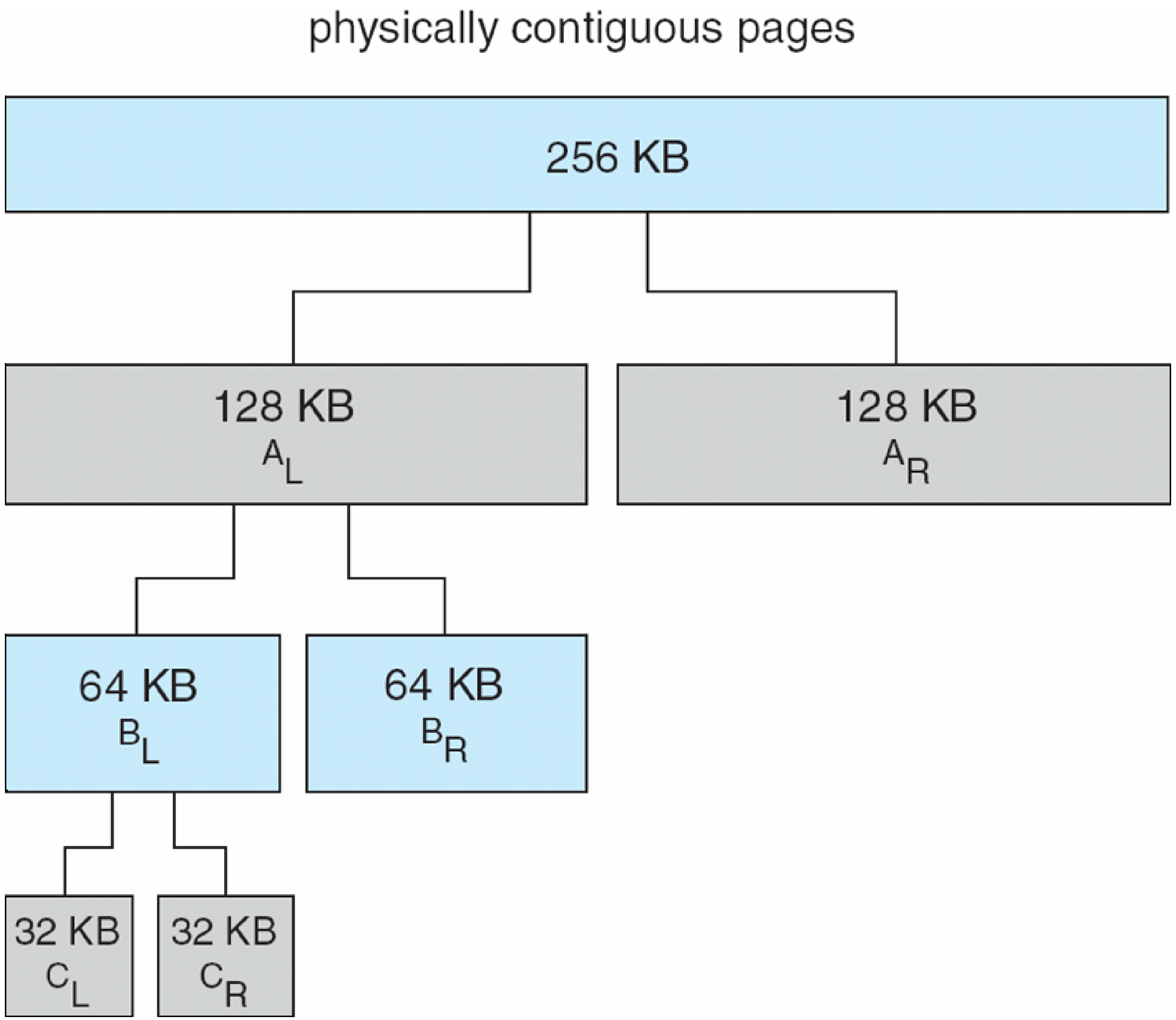
# Allocating Kernel Memory

- 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

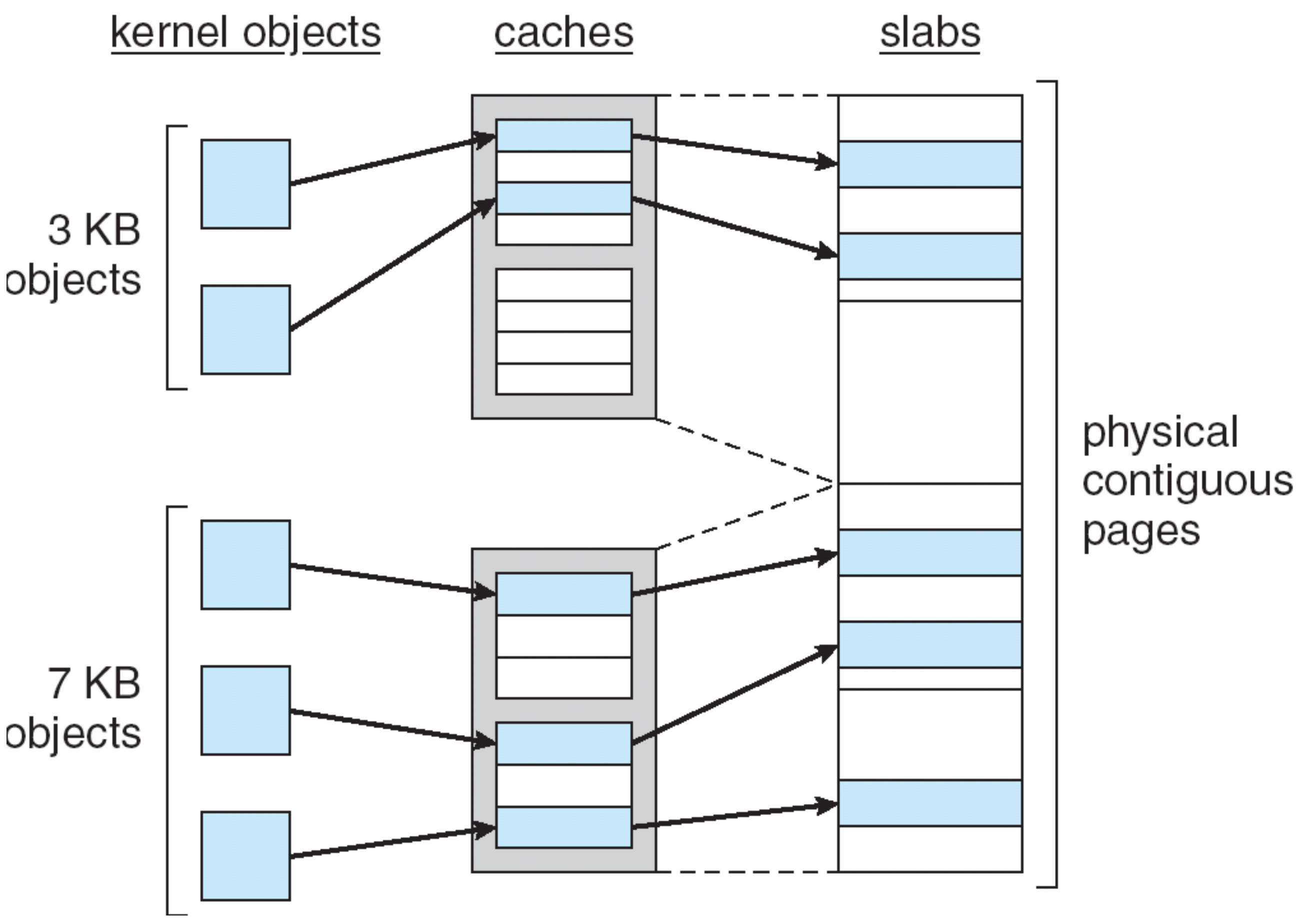
# Buddy System Allocator



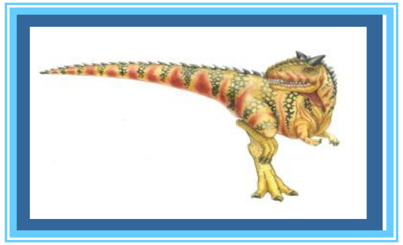
# Slab Allocator

- 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
- 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



# 9.9 Other Considerations



# Other Considerations

- Prepaging （预调页）
- Page Size（页大小）
- TLB Reach（TLB范围）
- Program Structure （程序结构）
- I/O interlock（ I/O 锁定）

# Other Issues -- Prepaging （预调页）

- 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 cost of $s * \alpha$ save pages faults > or < than the cost of prepaging $s * (1- \alpha)$ unnecessary pages?
    - $\alpha$ near zero $\Rightarrow$ prepaging loses

# Other Issues – Page Size （页大小）

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

# Other Issues – TLB Reach（TLB范围）

- 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

# Other Issues – 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;
```

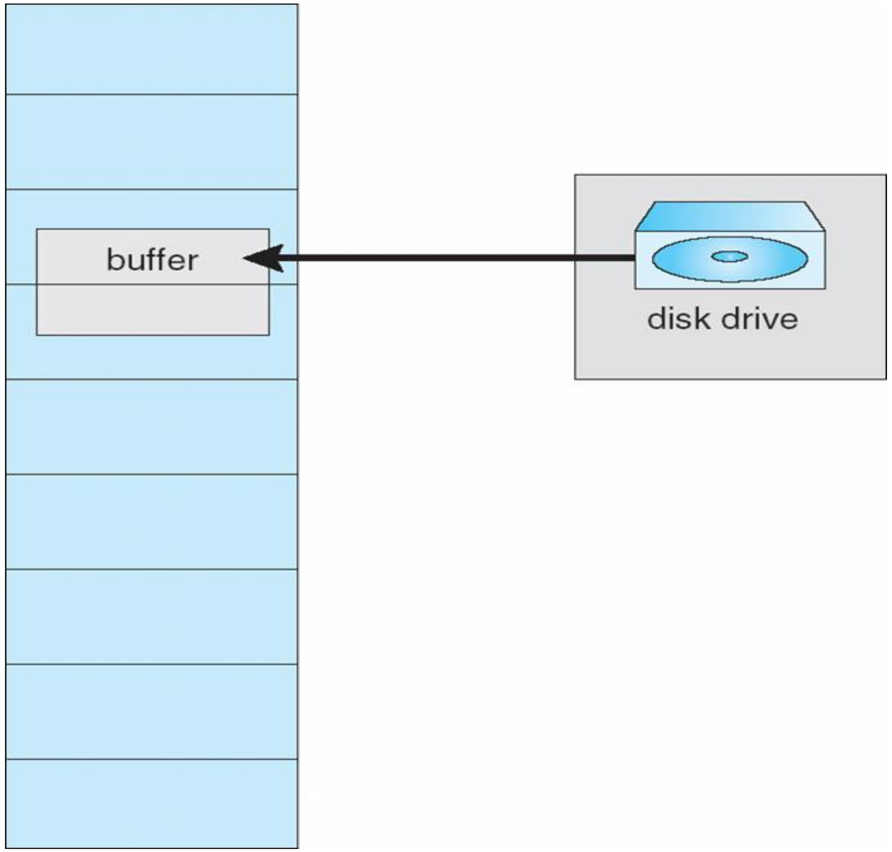
128 page faults

Which method should you program?

# 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



# 9.10 Operating System Examples

# Operating System Examples

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

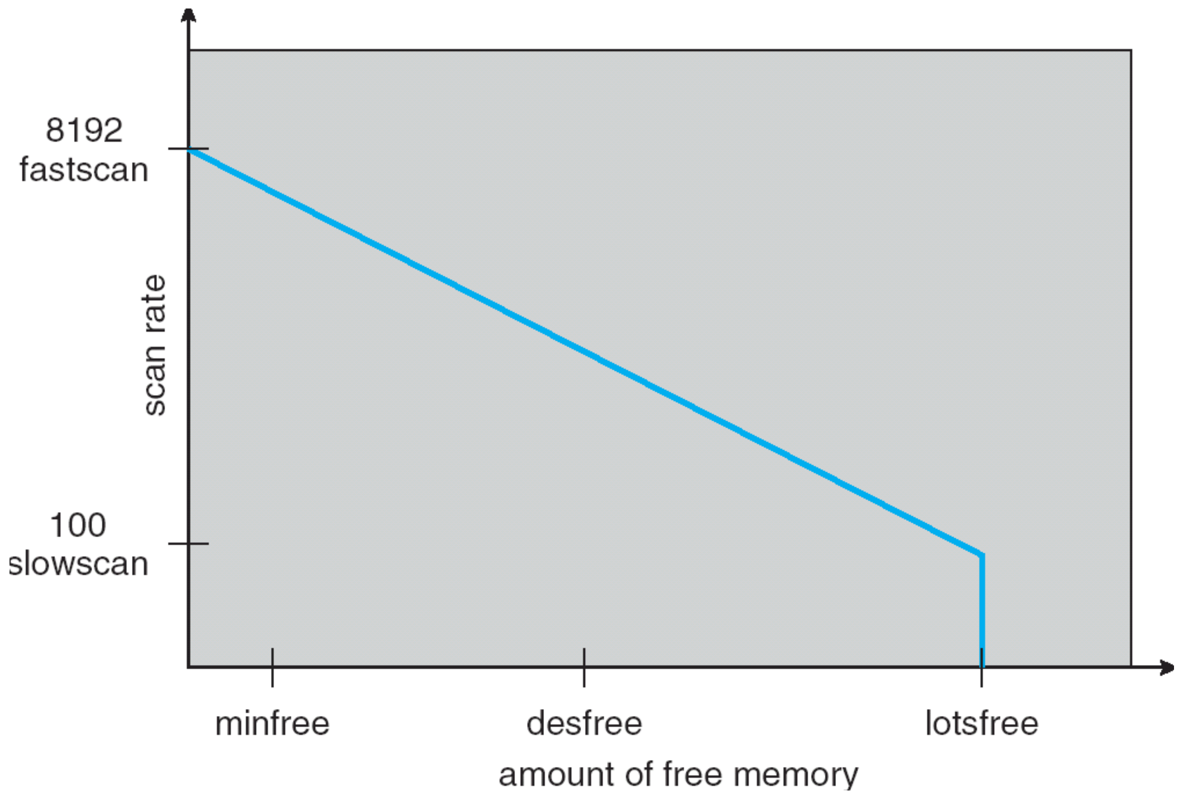
# Windows XP

- 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
- 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
- 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



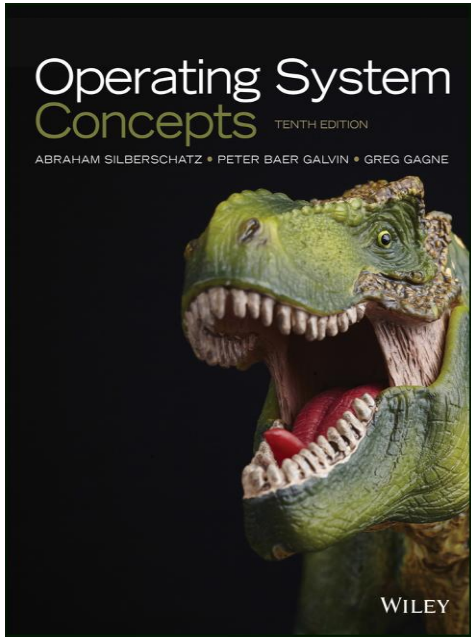
# HOMEWORK

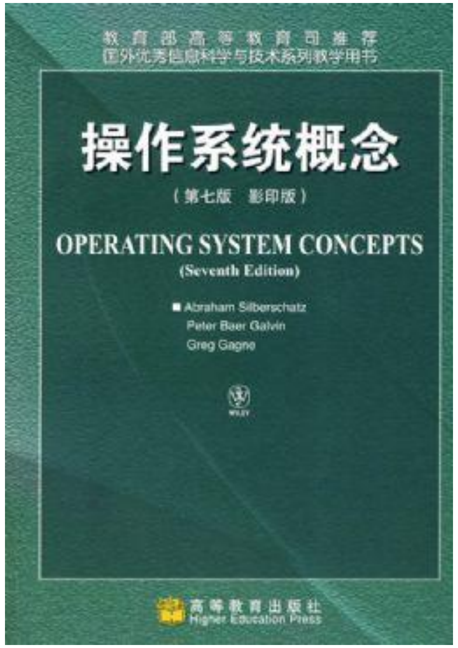
- 作业：
  - 学在浙大

- 习题分析

# Reading Assignments

- Read for this week:
  - Chapters 9
    of the text book:
- Read for next week:
  - Chapters  10
    of the text book:



# End of Chapter 9