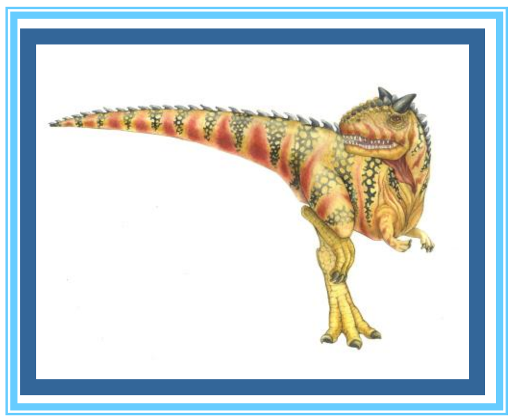
Chapter 9: Virtual Memory



Chapter 9: Virtual Memory

- Background
- Demand Paging
- Copy-on-Write（写时复制）
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples

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 is a technique that allows the execution of processes that may not be completely in 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

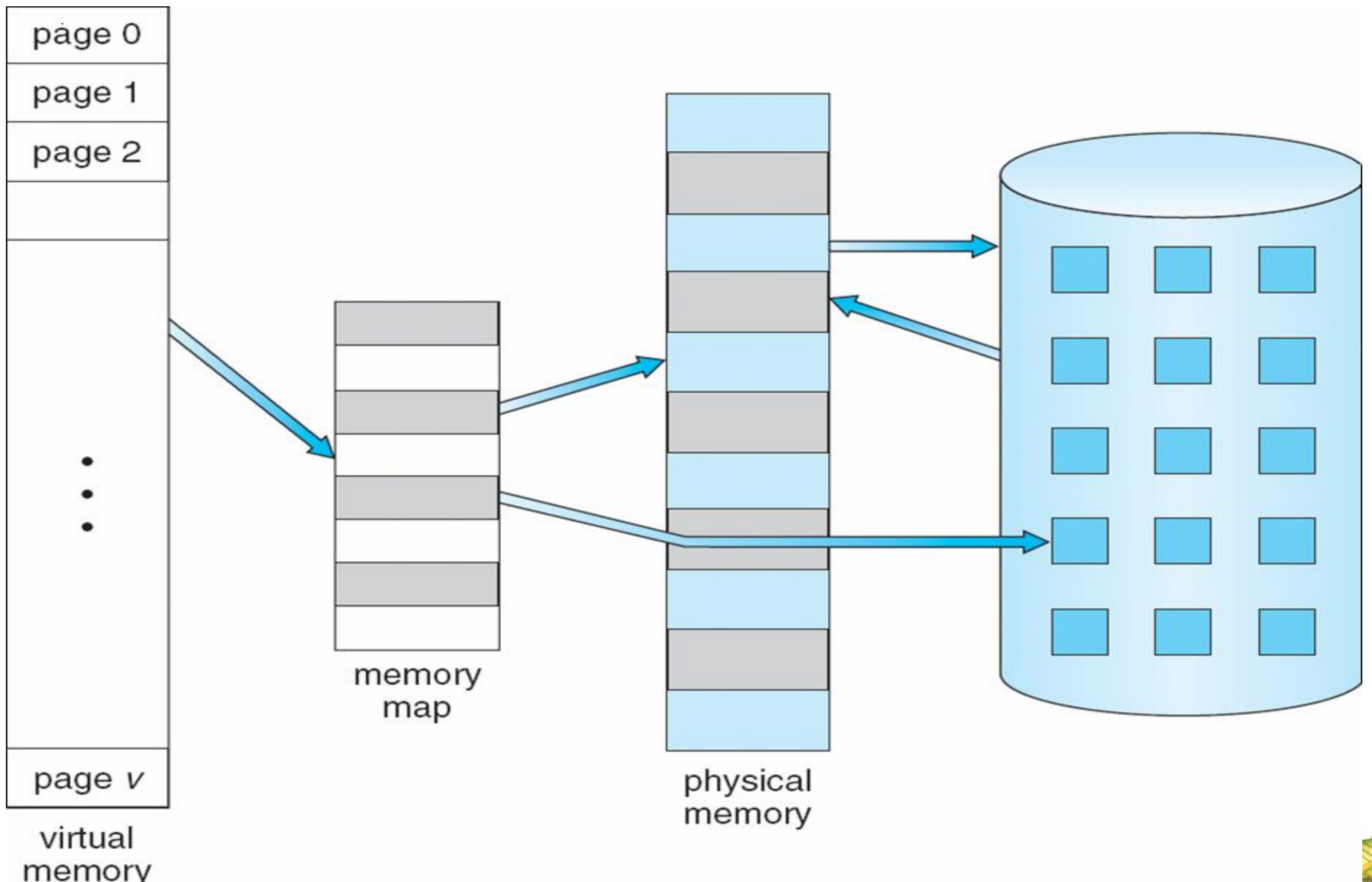
Background

- principle of locality:
  - Temporal Locality (Locality in Time):同一事物的访问在时间上聚集在一起；
  - Spatial Locality (Locality in Space):在时间上被引用的事物在空间上也接近（相邻的存储器地址，磁盘上的邻近扇区等），则引用序列被认为具有**空间局部性(Spatial Locality)**。
  - ✓ sum为时间局限性
  - ✓ 数组元素为空间局限性

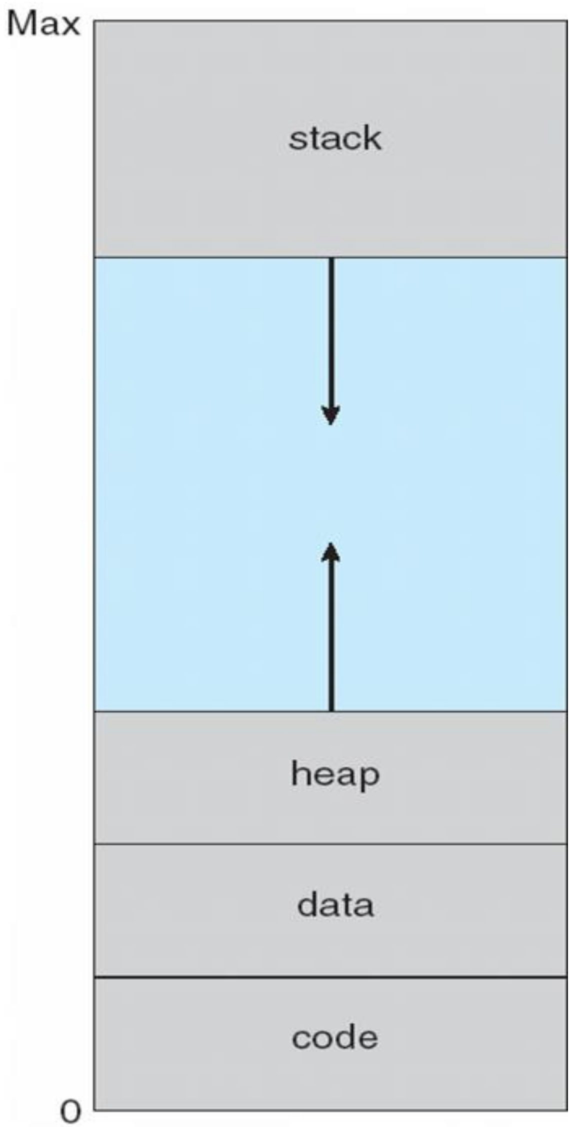
```
// Compute sum of an int array
int  a[N] = {2, 5, 3, 7, ...};
int sum=0;

for(i=0; i<N; i++)
     sum = sum + a[i];
```

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

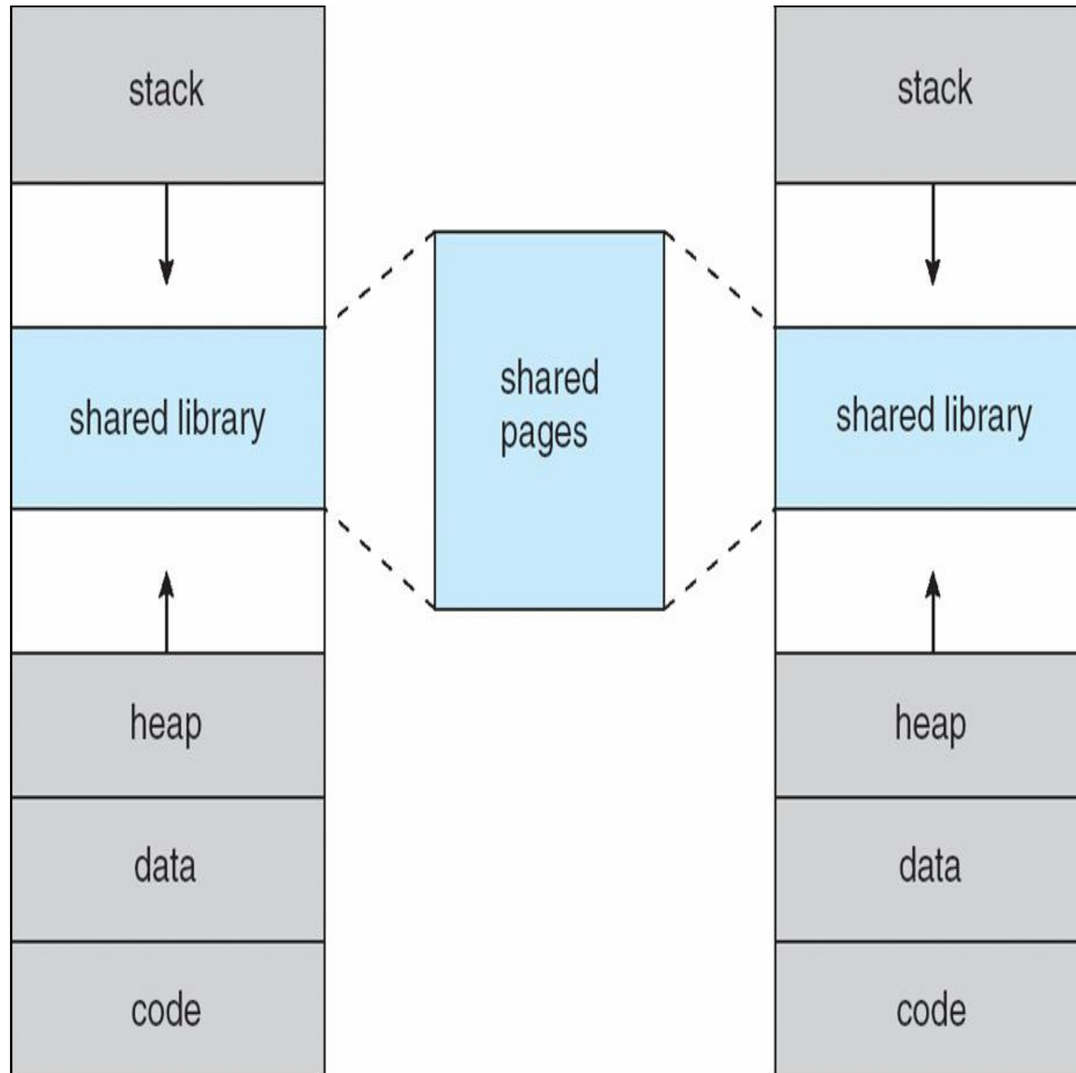
Virtual Memory That is Larger Than Physical Memory



Virtual-address Space



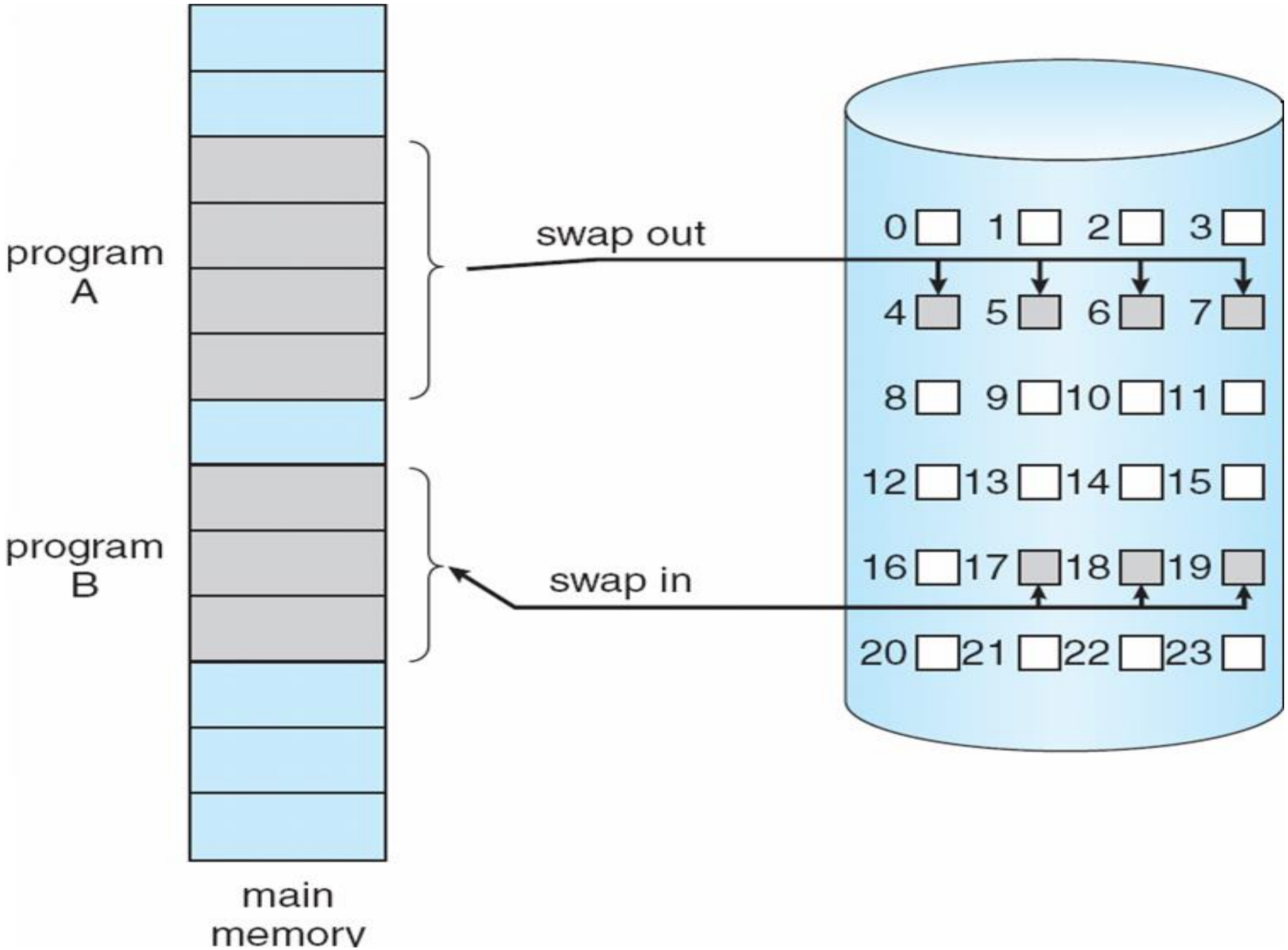
Shared Library Using Virtual Memory



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 $\Rightarrow$ reference to it
  - invalid reference $\Rightarrow$ abort
  - not-in-memory $\Rightarrow$ 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

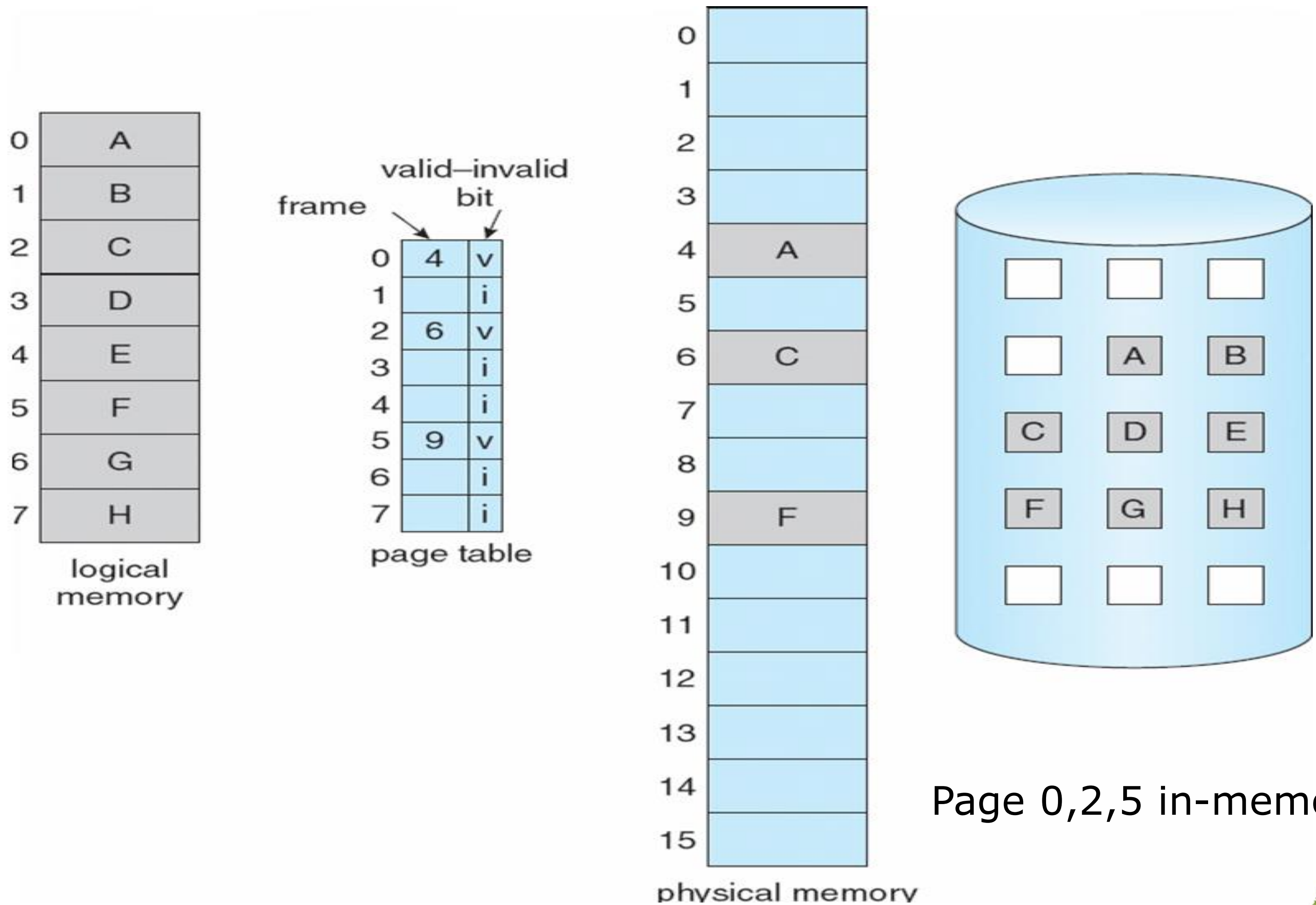
- With each page table entry a valid–invalid bit is associated
(**v** $\Rightarrow$ in-memory, **i** $\Rightarrow$ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:

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** $\Rightarrow$ page fault

Page Table When Some Pages Are Not in Main Memory



Page 0,2,5 in-memory

Page Fault

- If there is a reference to a page, first reference to that page will trap to operating system:

  page fault

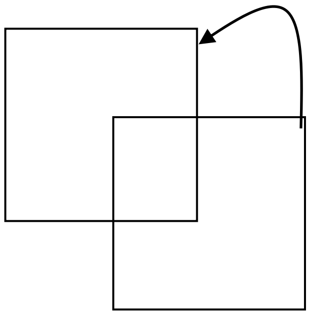
- Operating system looks at another table to decide:
  - Invalid reference ⇒ abort
  - Just not in memory
- Get empty frame
- Swap page into frame
- Reset tables
- Set validation bit = **v**
- Restart the instruction that caused the page fault

Page Fault 缺页中断处理代码示例：

```
// 定义一个页表结构
 struct PageTableEntry {
        int present; // 指示页面是否在内存中
        int frame;   // 页面的内存帧号
        // 其他页表项的信息（例如访问权限等）...
 };
 // 缺页中断处理函数
 void page_fault_handler() { // 获取引起缺页中断的虚拟内存地址
         int virtual_address = get_faulting_address(); // 从虚拟地址中解析出页表项中的索引
         int page_index = get_page_index(virtual_address); // 从页表中获取对应页表项
         PageTableEntry page_table_entry = get_page_table_entry(page_index);
         // 判断对应页面是否在内存中
 if (page_table_entry.present) {
         // 页面已经在内存中，可能是因为其他原因引起的中断，进行相应处理... }
 else { // 页面不在内存中，需要将其加载到内存中
         int frame = allocate_frame(); // 从内存中分配一个合适的页面框
         page_table_entry.frame = frame; // 更新页表项中的页面框号
         page_table_entry.present = 1; // 更新页表项中的页面在内存标志
         load_page_from_disk(page_index, frame); // 从磁盘加载页面到内存中
 } // 更新页表项中的其他信息...
         update_page_table_entry(page_index, page_table_entry); // 恢复被中断的指令执行
         restore_interrupted_instruction();
 }
```

Page Fault (Cont.)

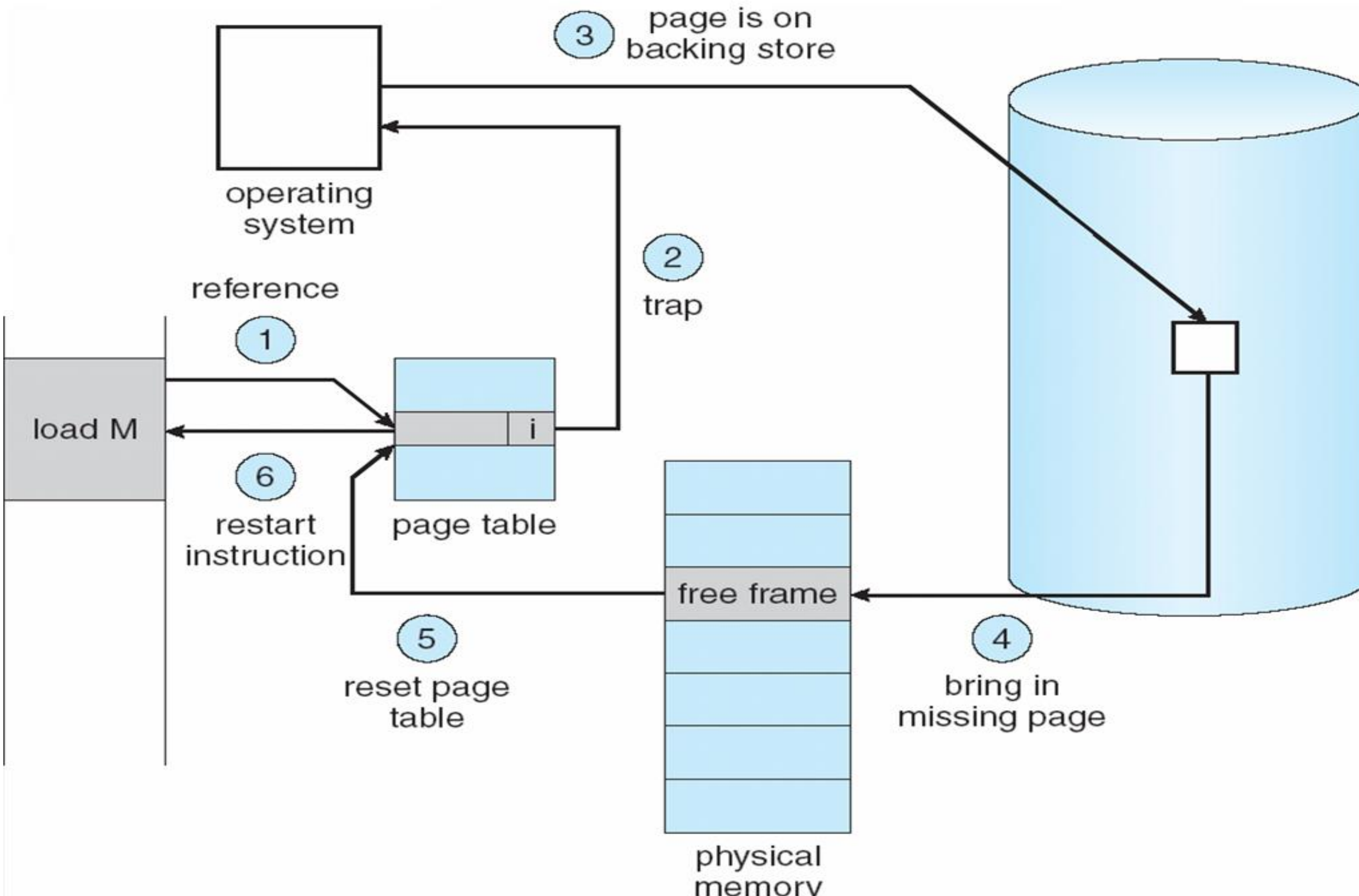
- Restart instruction
  - block move

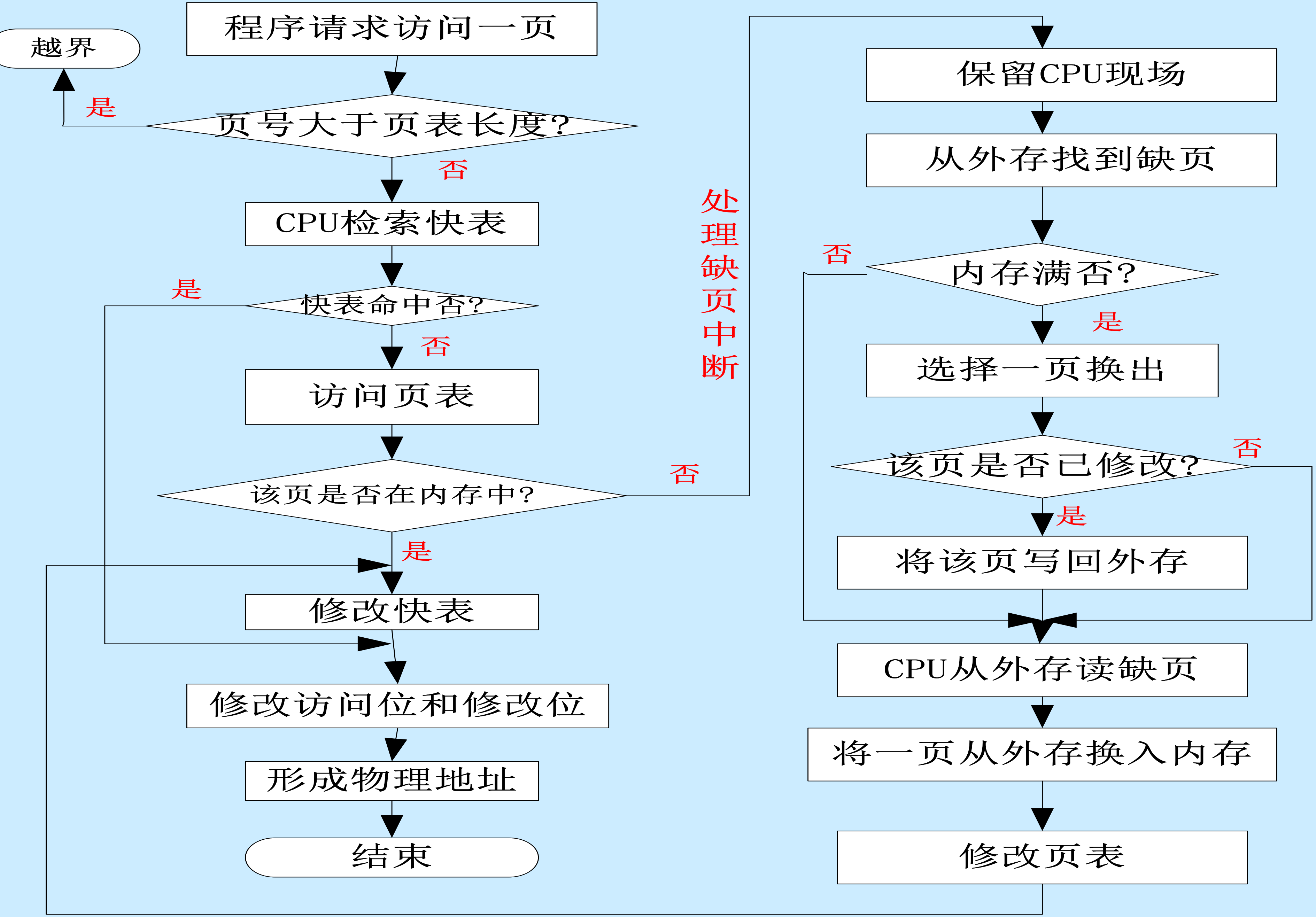


page fault process

  - auto increment/decrement location

Steps in Handling a Page Fault





请求分页系统中地址变换的过程

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 (page fault overhead

  + swap page out

  + swap page in

  + restart overhead)

Demand Paging Example

- Memory access time = 200 nanoseconds

- Average page-fault service time = 8 milliseconds(毫秒)

- 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(微妙).

Process Creation

- Virtual memory allows other benefits during process creation:

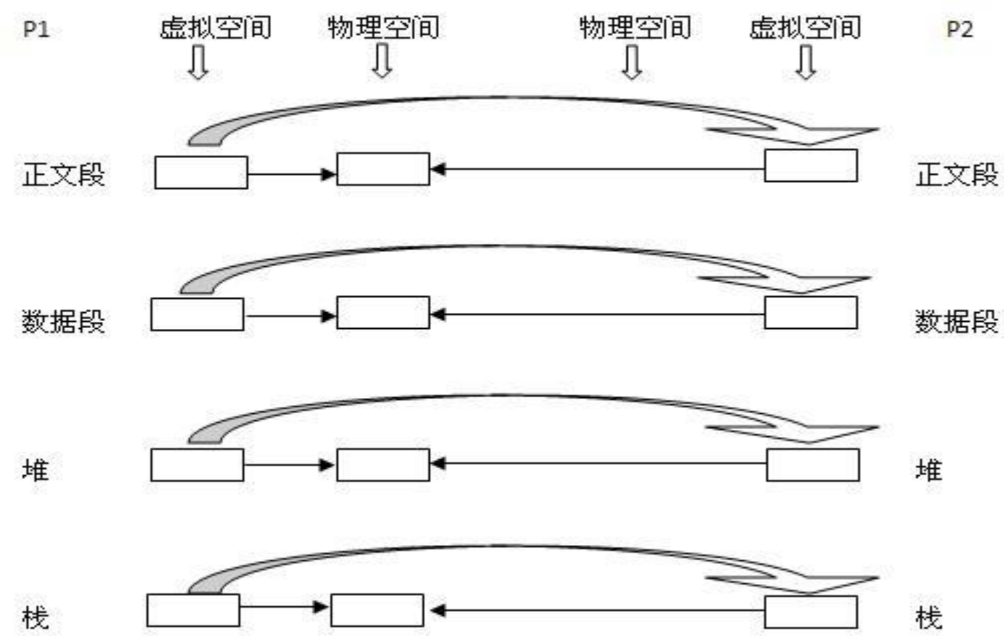
  - Copy-on-Write（写入时复制）

  - Memory-Mapped Files (later)

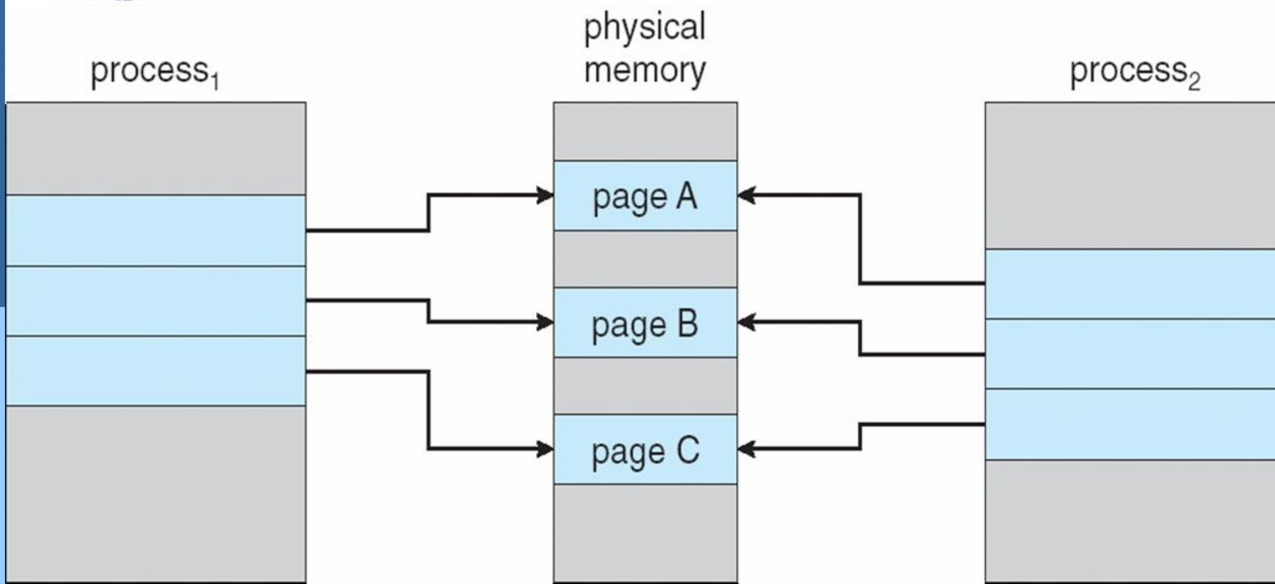
Copy-on-Write

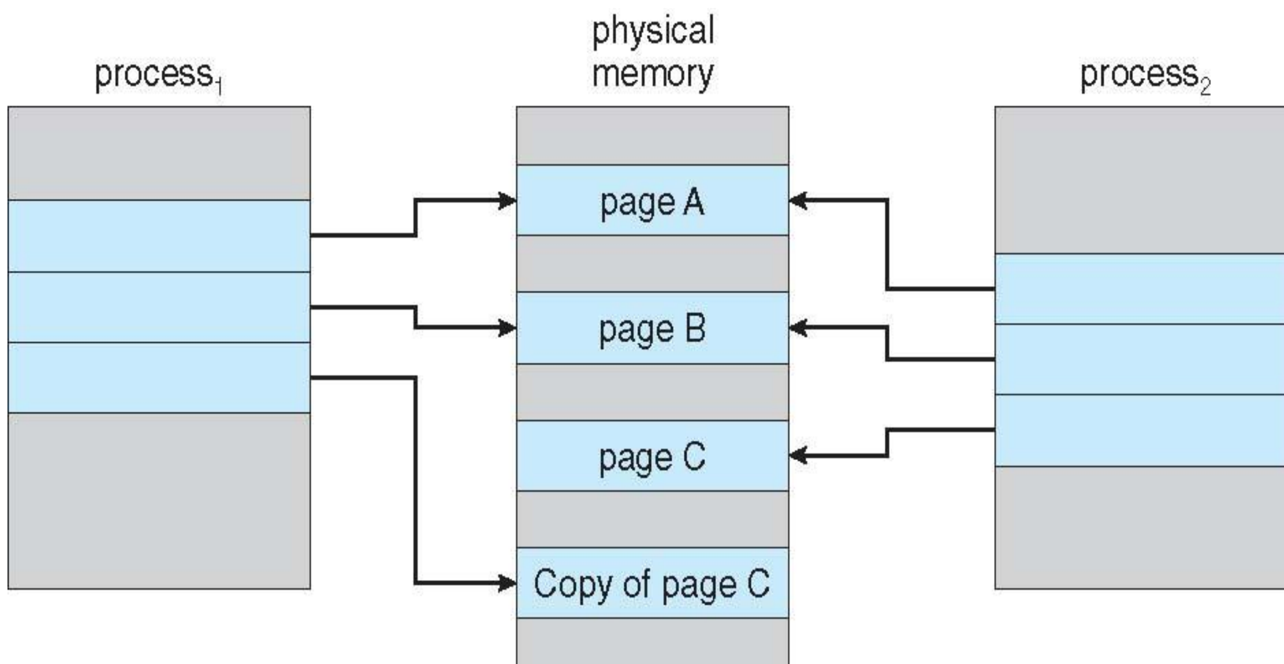
- Copy-on-Write (COW) allows both parent and child processes to initially *share* the same pages in memory, If either process modifies a shared page, only then is the page copied

在Linux程序中，fork（）会产生一个和父进程完全相同的子进程，出于效率考虑，引入了“写时复制“技术，也就是只有进程空间的各段的内容要发生变化时，才会将父进程的内容复制一份给子进程。



Before Process 1 Modifies Page C





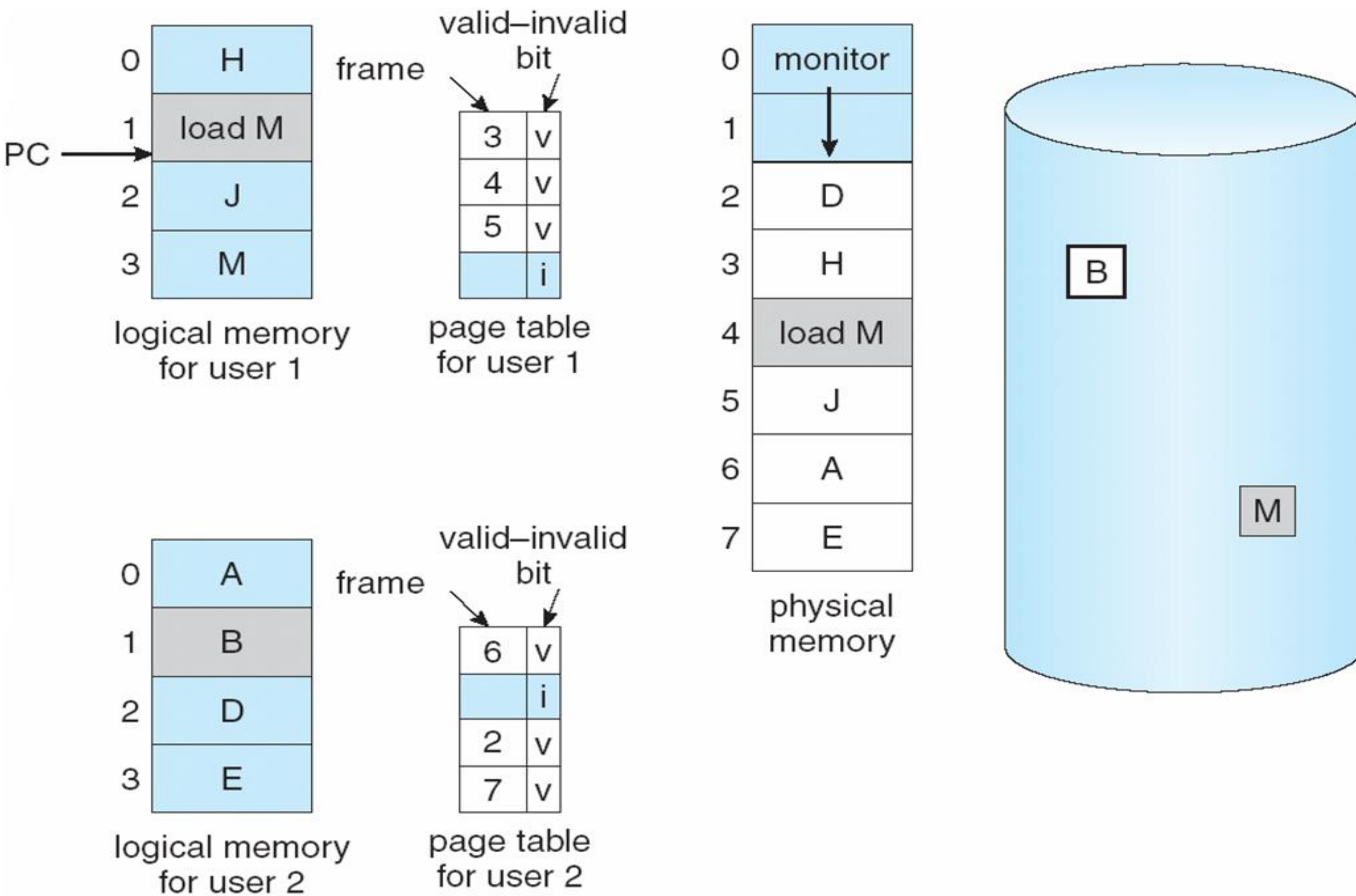
What happens if there is no free frame?

- 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

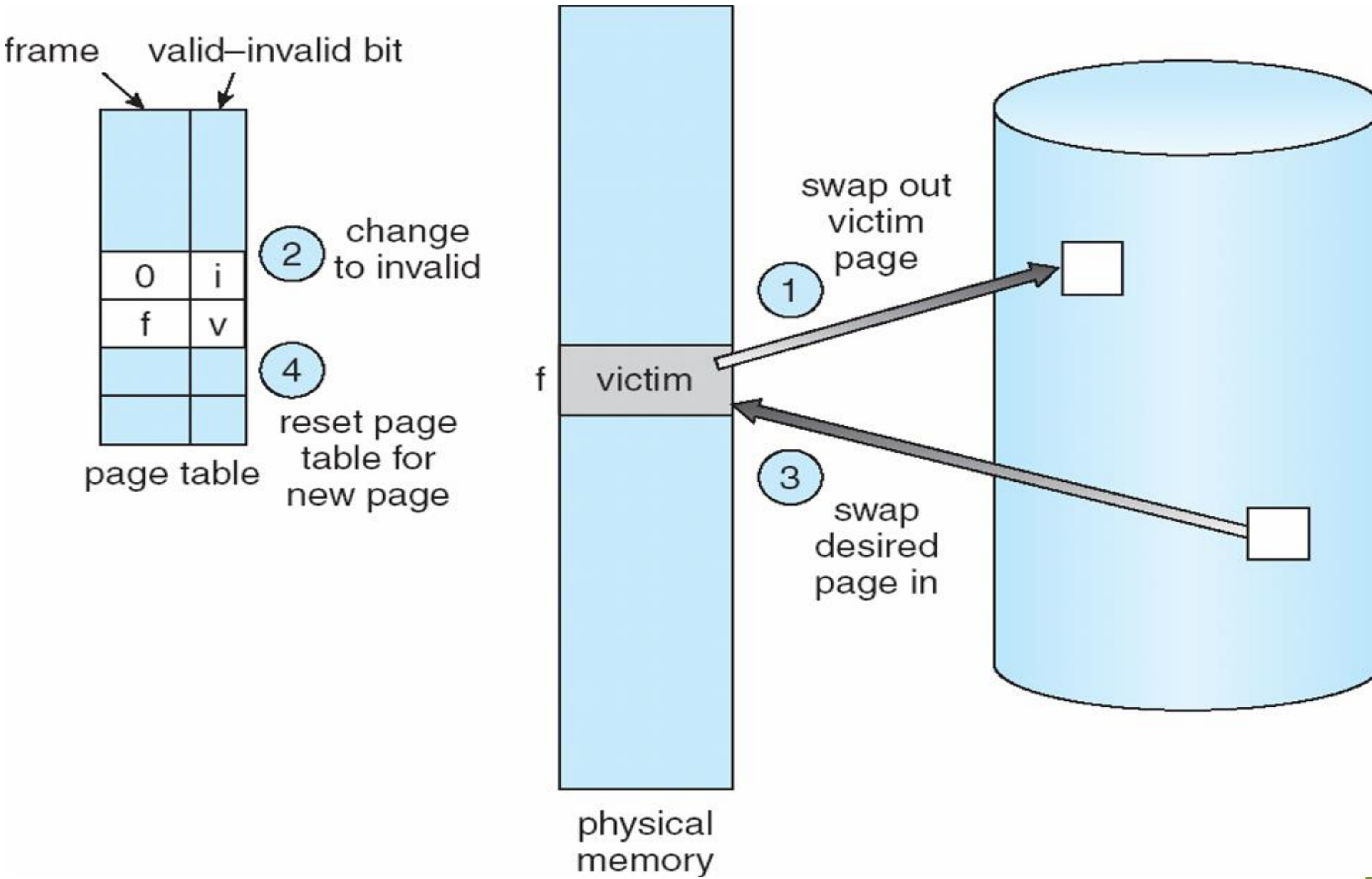
Need For Page Replacement



Basic Page Replacement

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

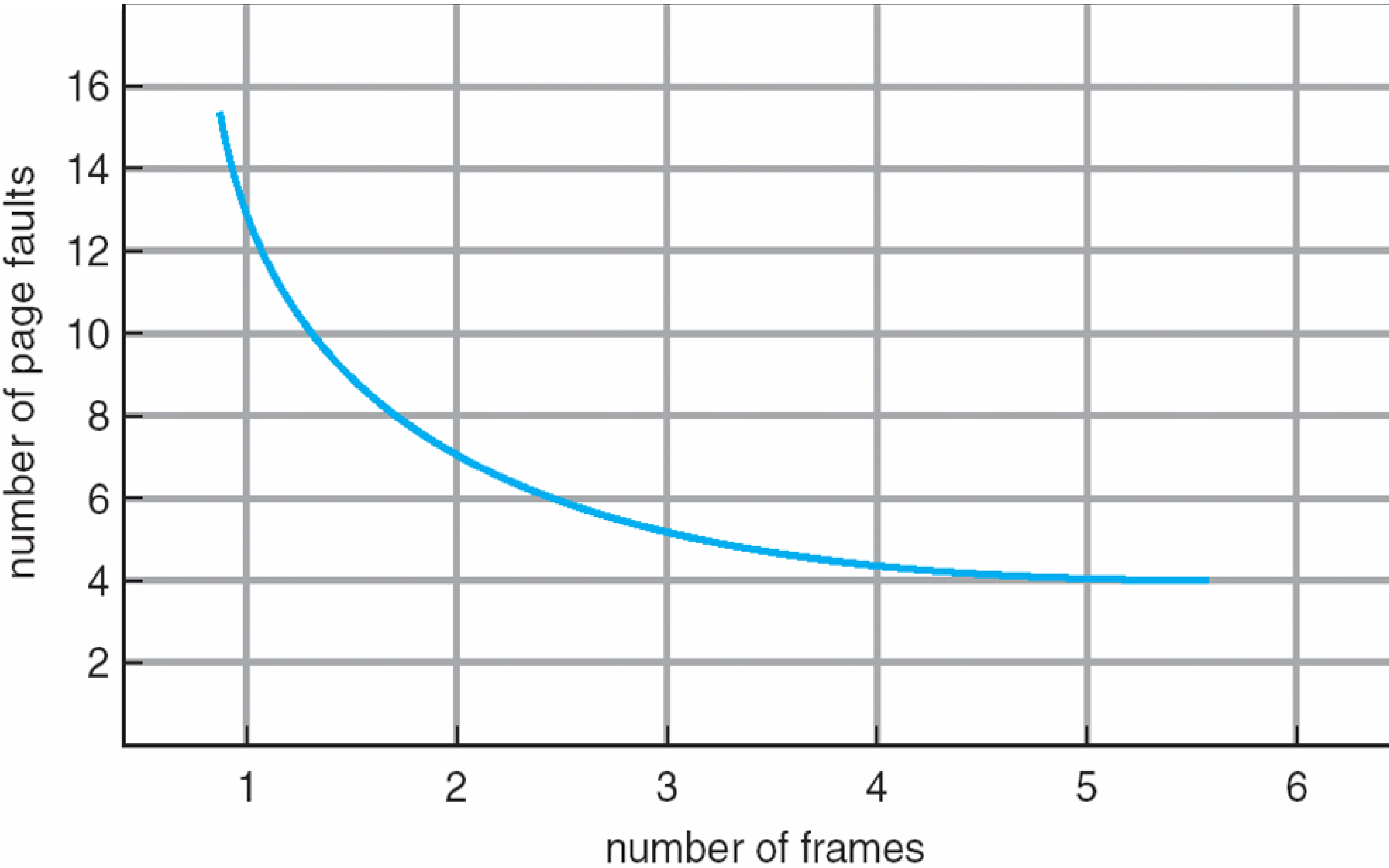


Page Replacement Algorithms

- 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 is

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

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   n=12

- 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

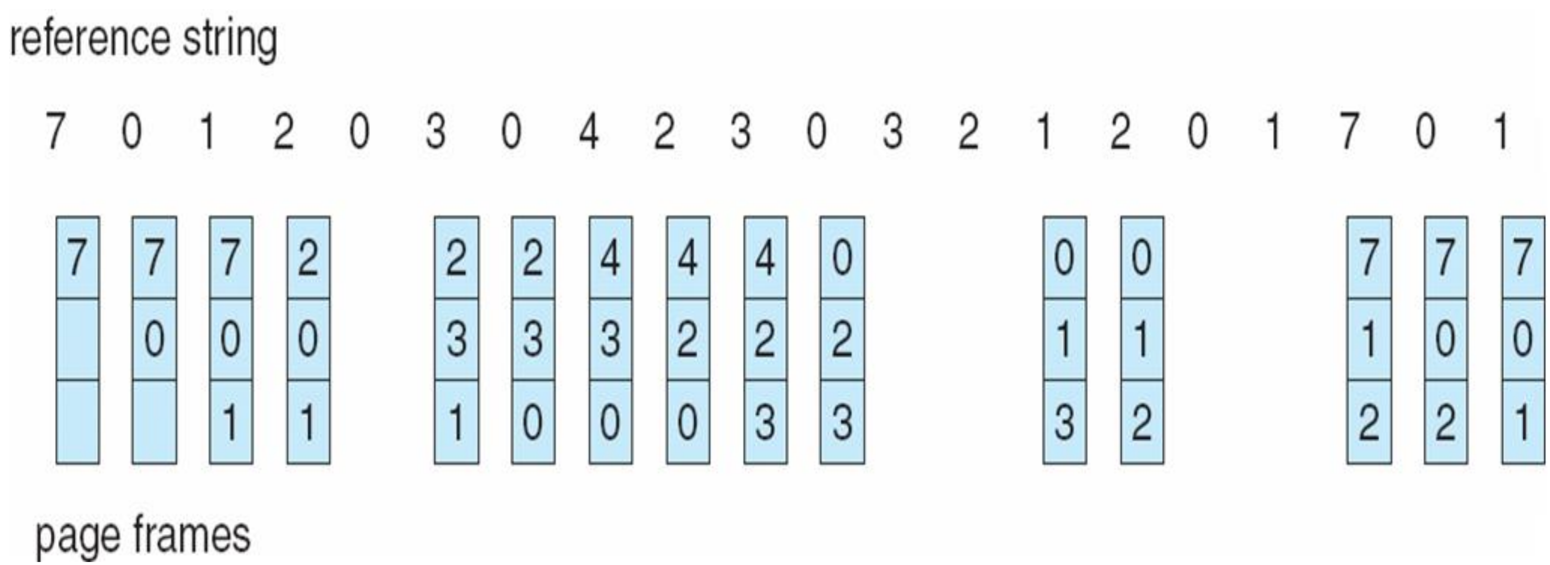
- 4 frames

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

10 page faults

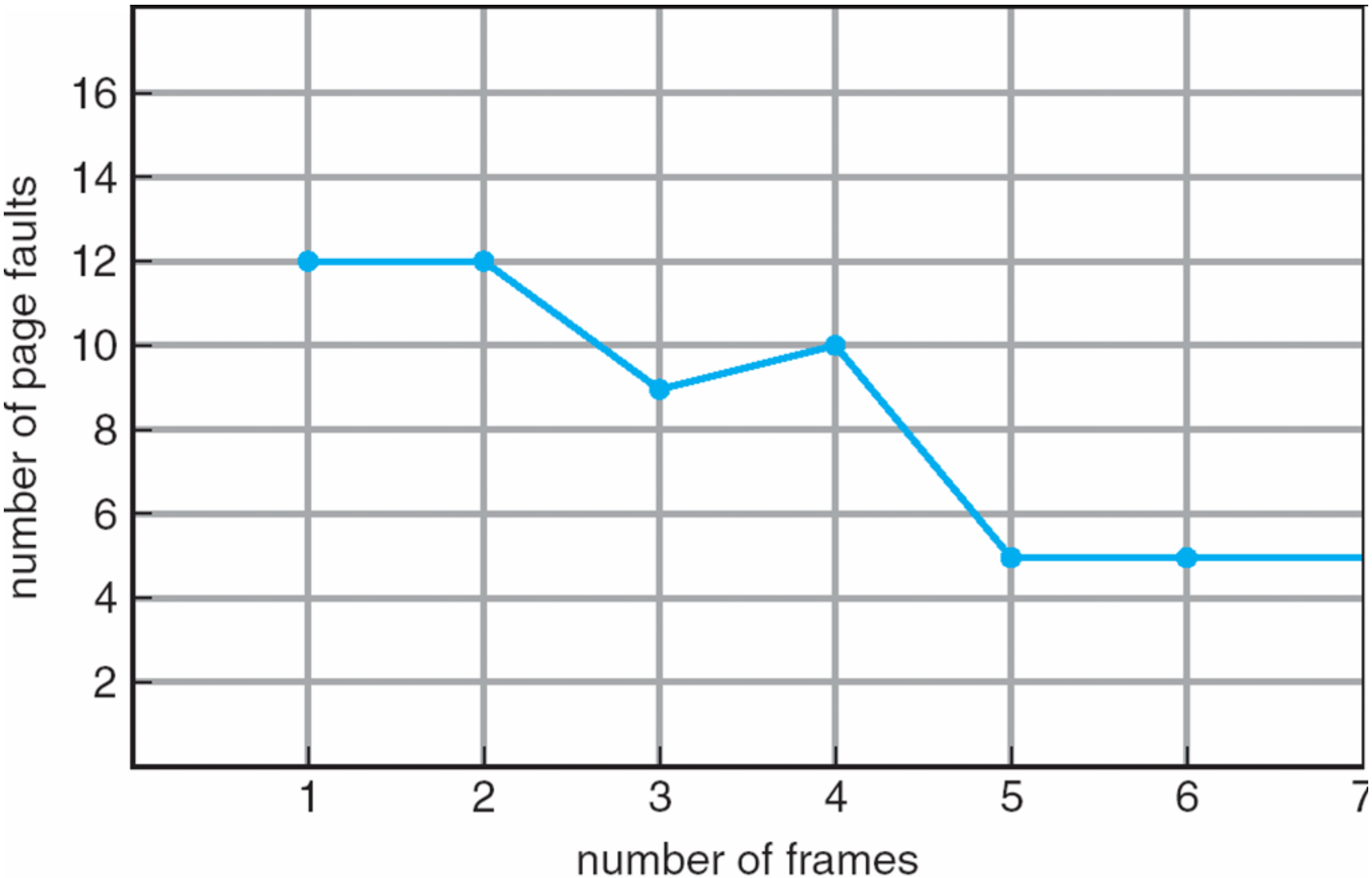
- Belady's Anomaly: more frames $\Rightarrow$ more page faults

FIFO Page Replacement



F=15 page faults → f= F/n =15/20=0.75

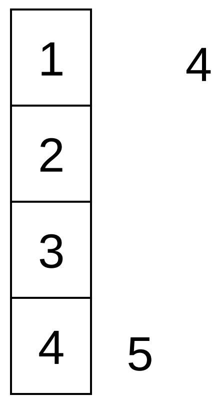
FIFO Illustrating Belady's Anomaly



Optimal Algorithm

- Replace page that will not be used for longest period of time
- 4 frames example

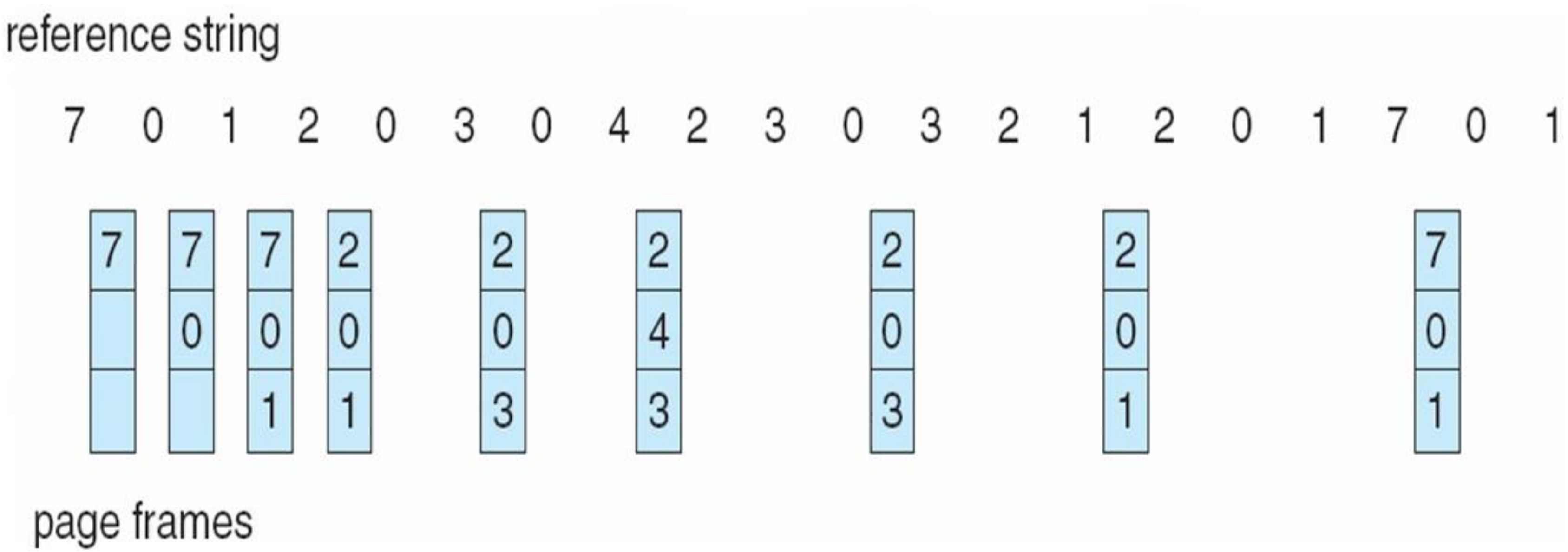
  1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5   n=12



6 page faults

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

Optimal Page Replacement



F=9 page faults

f= F/n =9/20=0.45

Least Recently Used (LRU) Algorithm

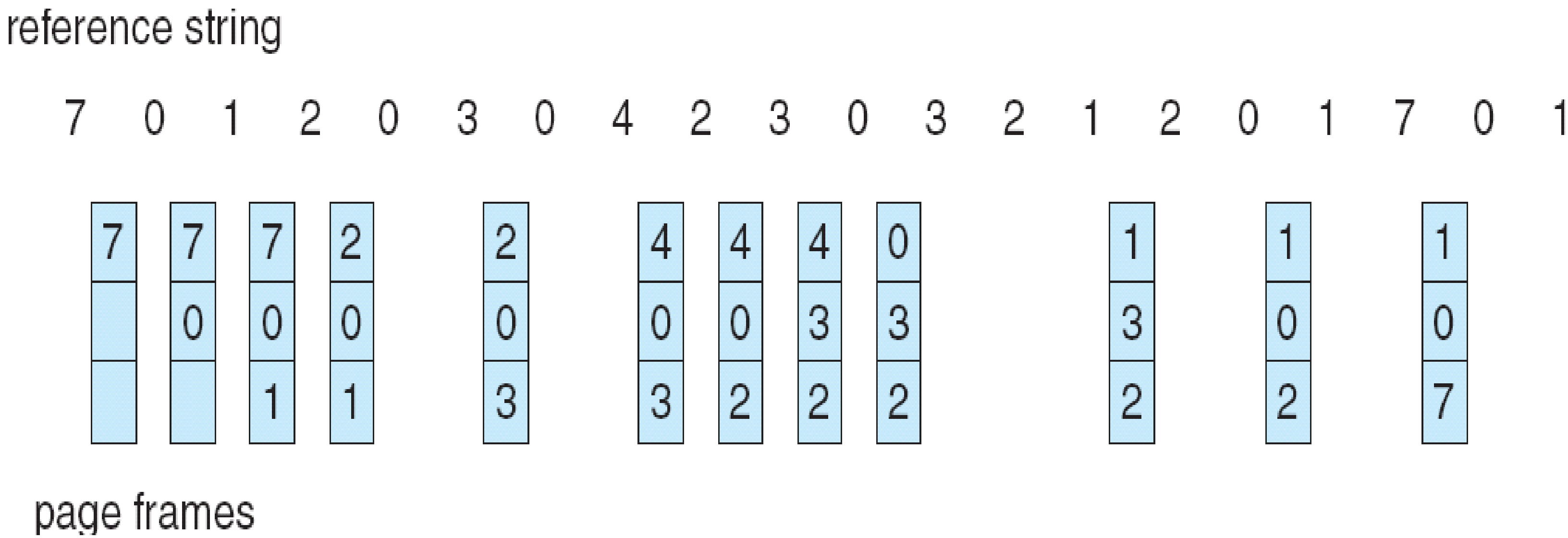
- Reference string: 1, 2, 3, 4, 1, 2, **5**, 1, 2, **3**, **4**, **5**

1	1	1	1	**5**
2	2	2	2	2
3	**5**	5	**4**	4
4	4	**3**	3	3

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

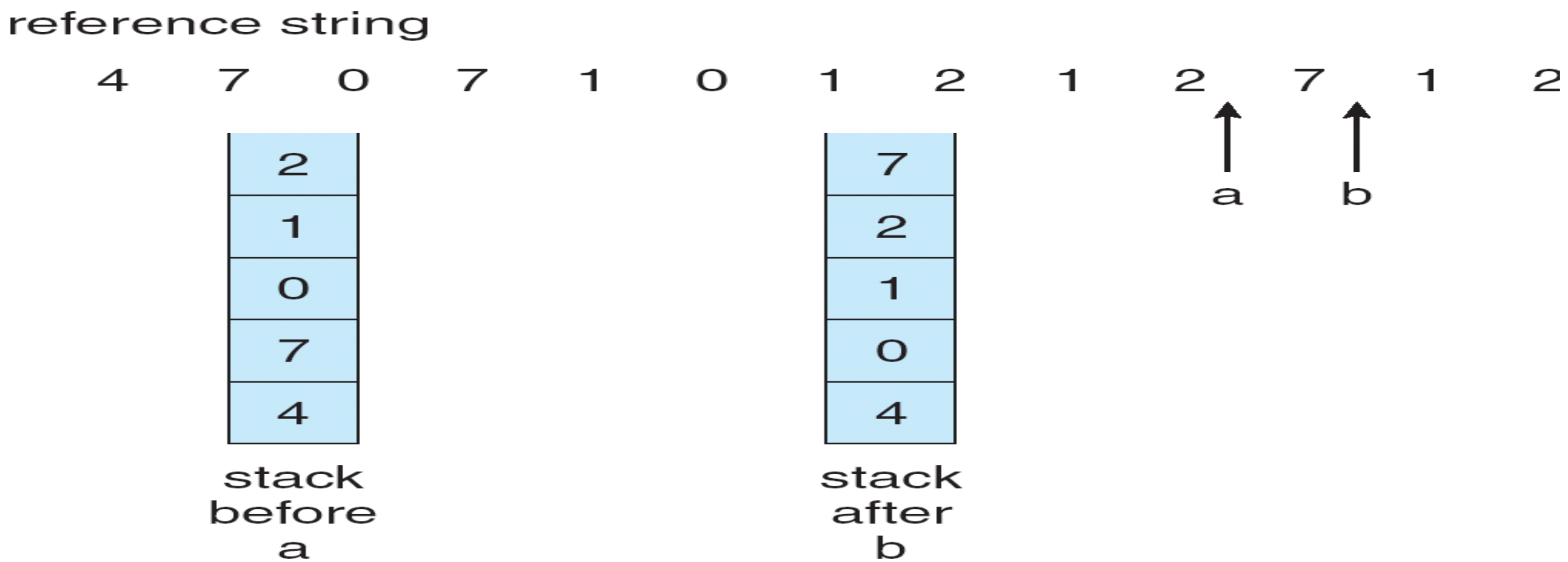


F=12 page faults

f= F/n =12/20=0.6

LRU Algorithm (Cont.)

- 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

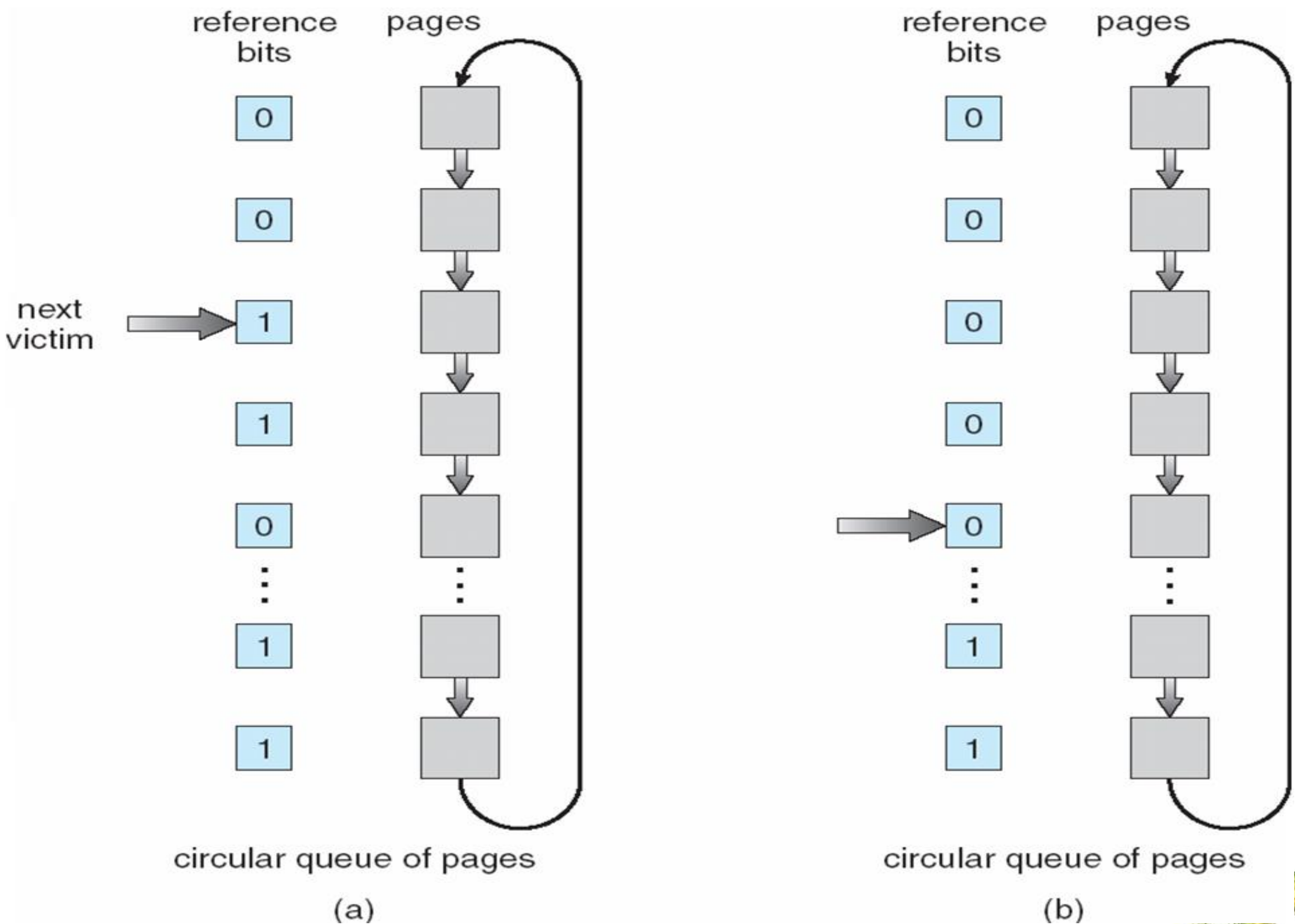


Use Of A Stack to Record The Most Recent Page References

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)替换标示为0的页
    - We do not know the order, however
- Second chance
  - Need reference bit
  - Clock replacement
  - If page to be replaced (in clock order) has reference bit = 1 then:
    (如果被替换的页为1,先置为0，看看被引用的频率)
    - 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

- 不仅考虑页面的使用情况，并考虑置换代价，选择淘汰页面时：选择未访问且未被修改的页面。设置两位
  - 访问位（A），修改位（M）；
  - 启动一个进程时，A与M均置为0;
  - A被周期清零;
- 内存所有页面分成为四类，选择开销最小的置换.
  - 第0类：无访问，无修改(A=0,M=0)
  - 第1类：无访问，有修改(A=0,M=1)
  - 第2类：有访问，无修改(A=1,M=0)
  - 第3类：有访问，有修改(A=1,M=1)

Counting Algorithms

- Keep a counter of the number of references that have been made to each page
- **LFU(least frequently used) Algorithm**: replaces page with smallest count（使用最少的）
- **MFU(Most frequently used) Algorithm**: based on the argument that the page with the smallest count was probably just brought in and has yet to be used（使用最多的）

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

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

  – $S = \sum s_i$

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

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

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

Priority Allocation

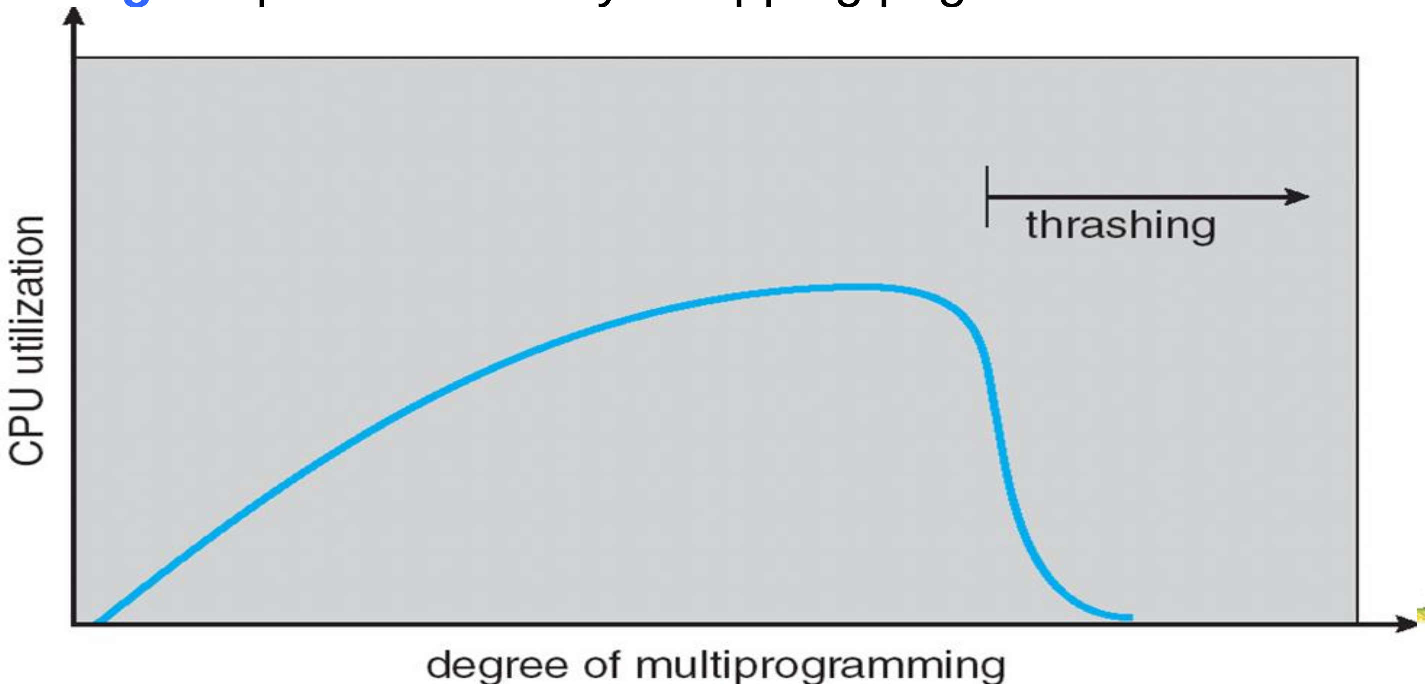
- Use a proportional allocation scheme using priorities rather than size
- If process P_i generates a page fault,
  - select for replacement one of its frames
  - select for replacement a frame from a process with lower priority number

Global 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

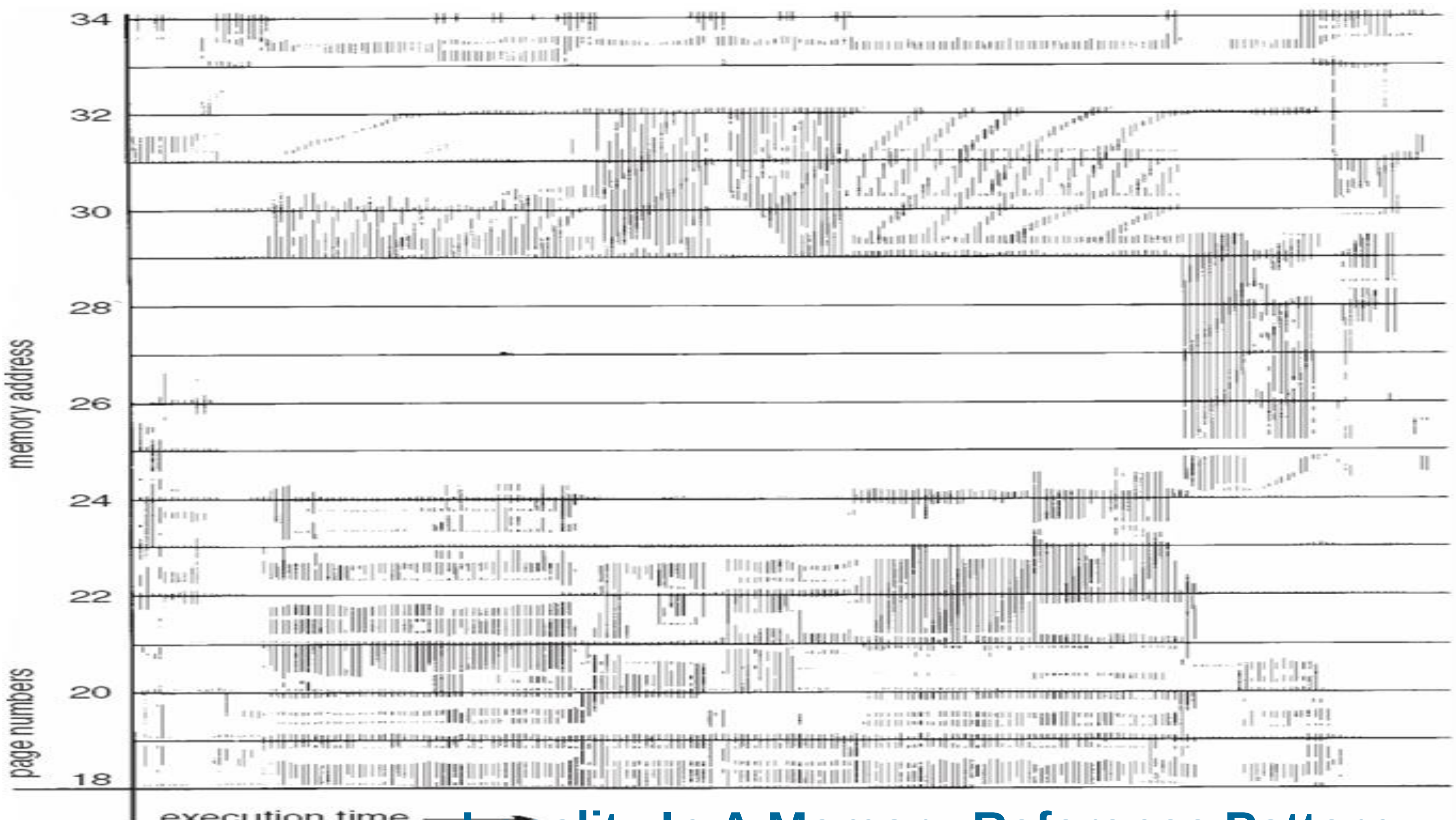
Thrashing

- 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** $\equiv$ a process is busy swapping pages in and out



Demand Paging and Thrashing

- Why does demand paging work? Locality model
  - Process migrates from one locality to another
  - Localities may overlap
- Why does thrashing occur? Σ size of locality > total memory size



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

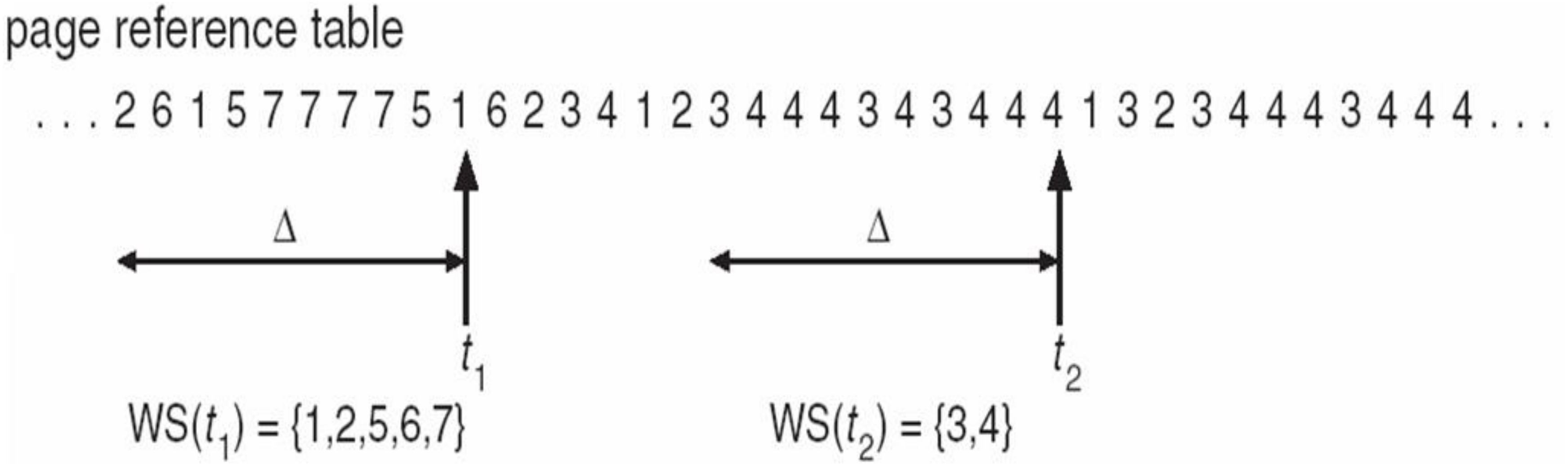
- WSS_i (working set of Process P_i) =

  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
- Policy if D > m, then suspend one of the processes

Working-set model

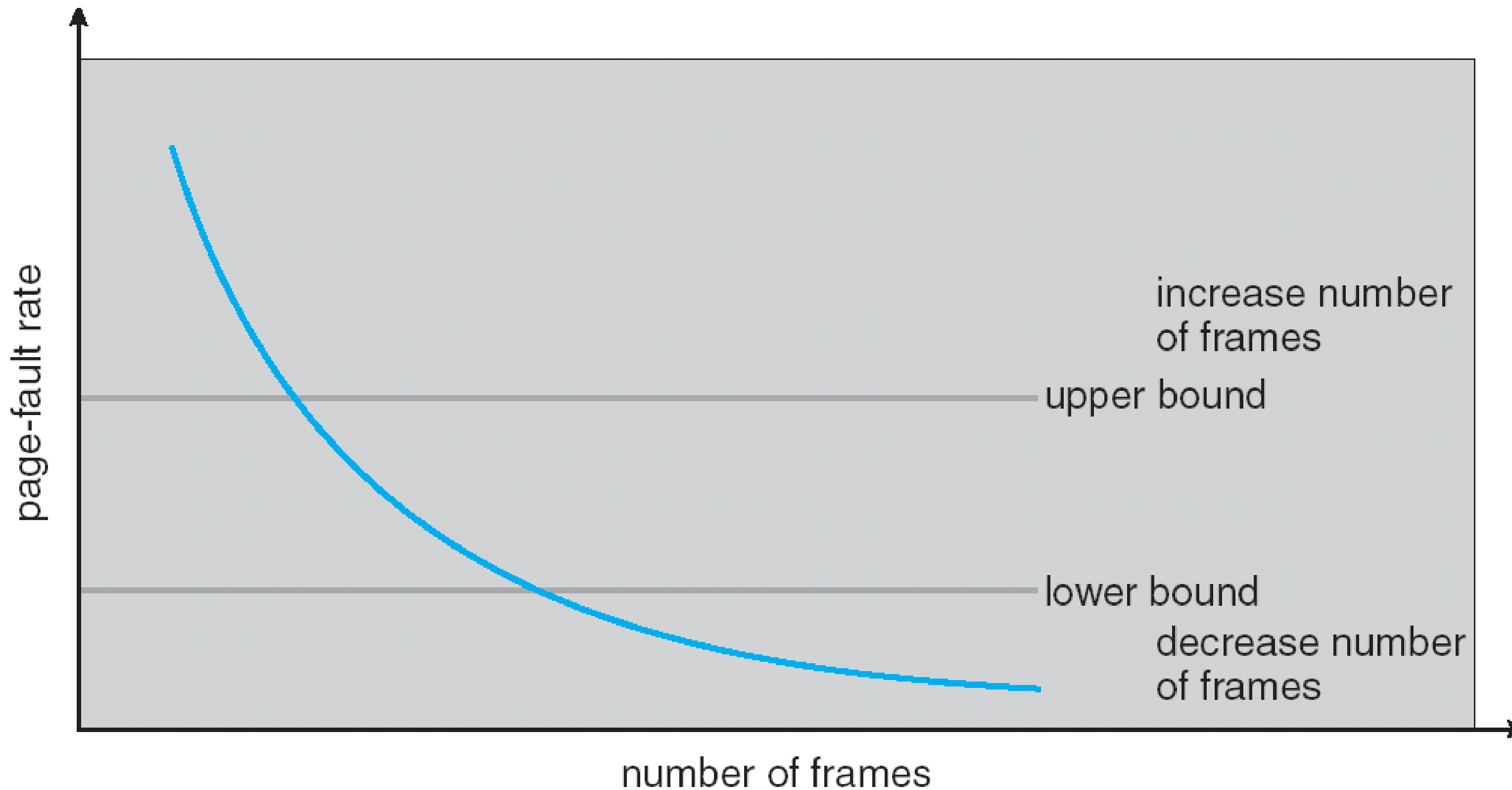


Keeping Track of the Working Set

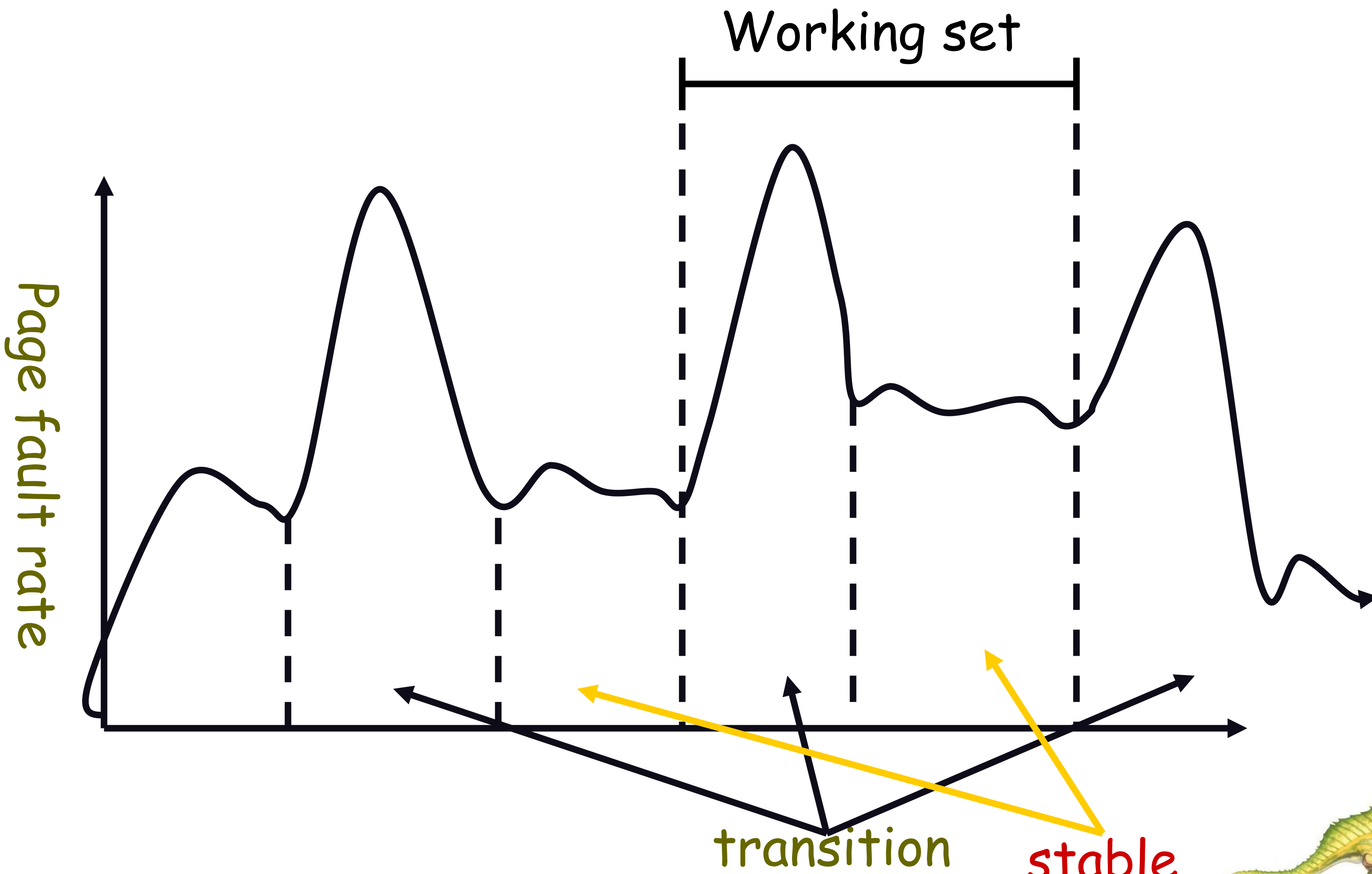
- Approximate with interval timer + a reference bit
- Example: $\Delta = 10{,}000$
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 bits for each page
  - Whenever a timer interrupts copy and sets the values of all reference bits to 0
  - If one of the bits in memory = 1 $\Rightarrow$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

Page-Fault Frequency Scheme

- 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

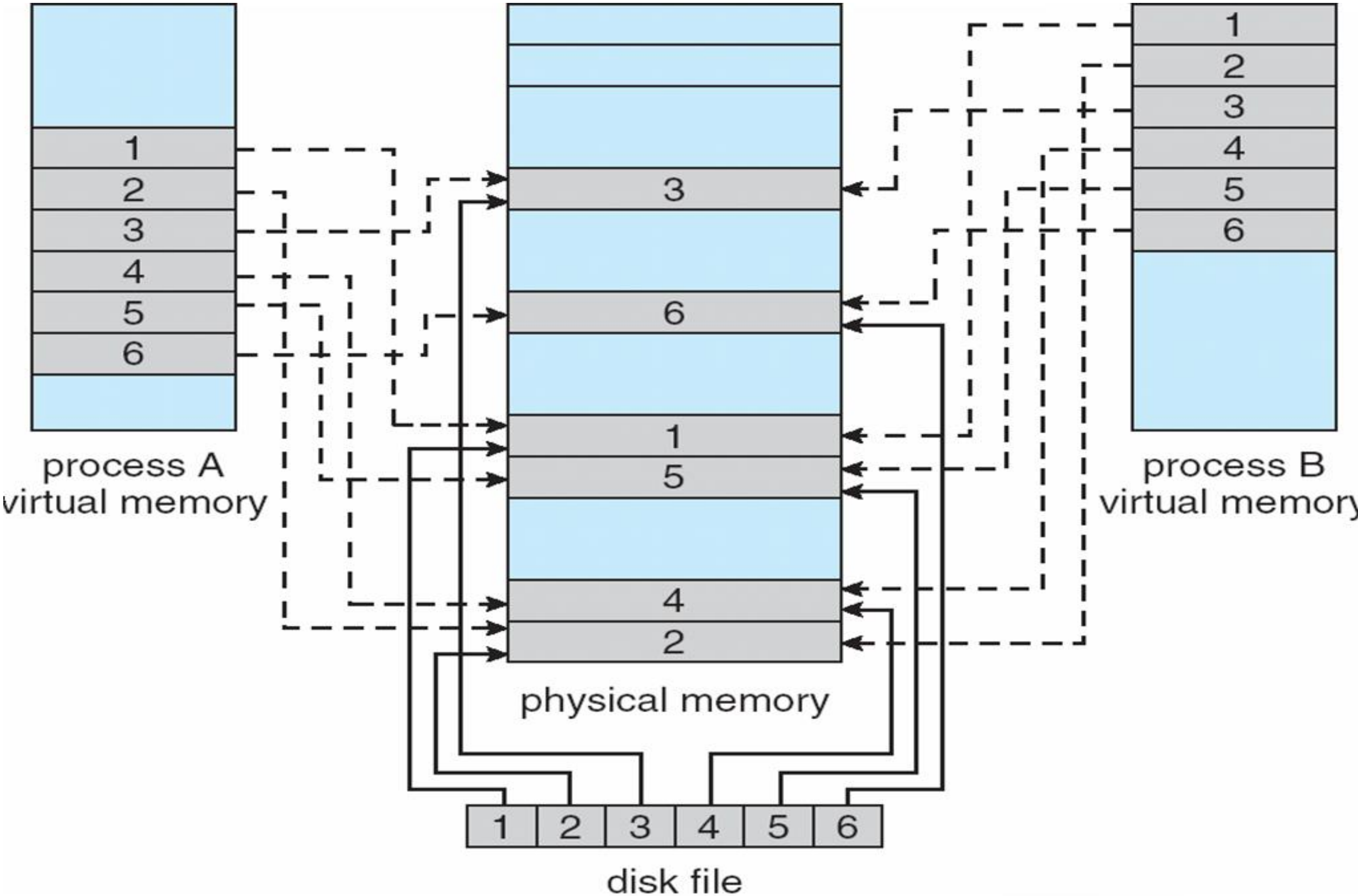


51

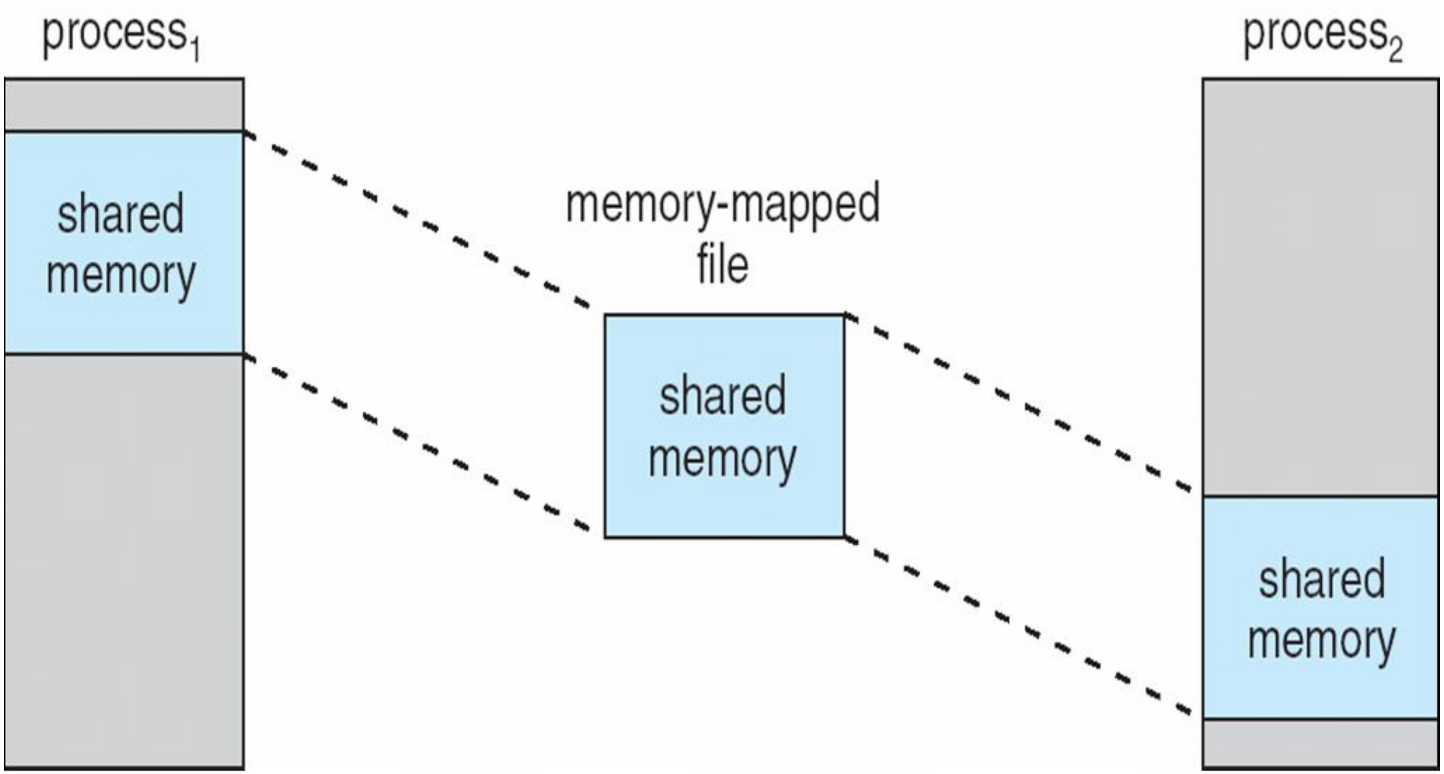
Memory-Mapped Files

- 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



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
- 分配核心级内存不同于用户级的:大小变化/连续

Design features

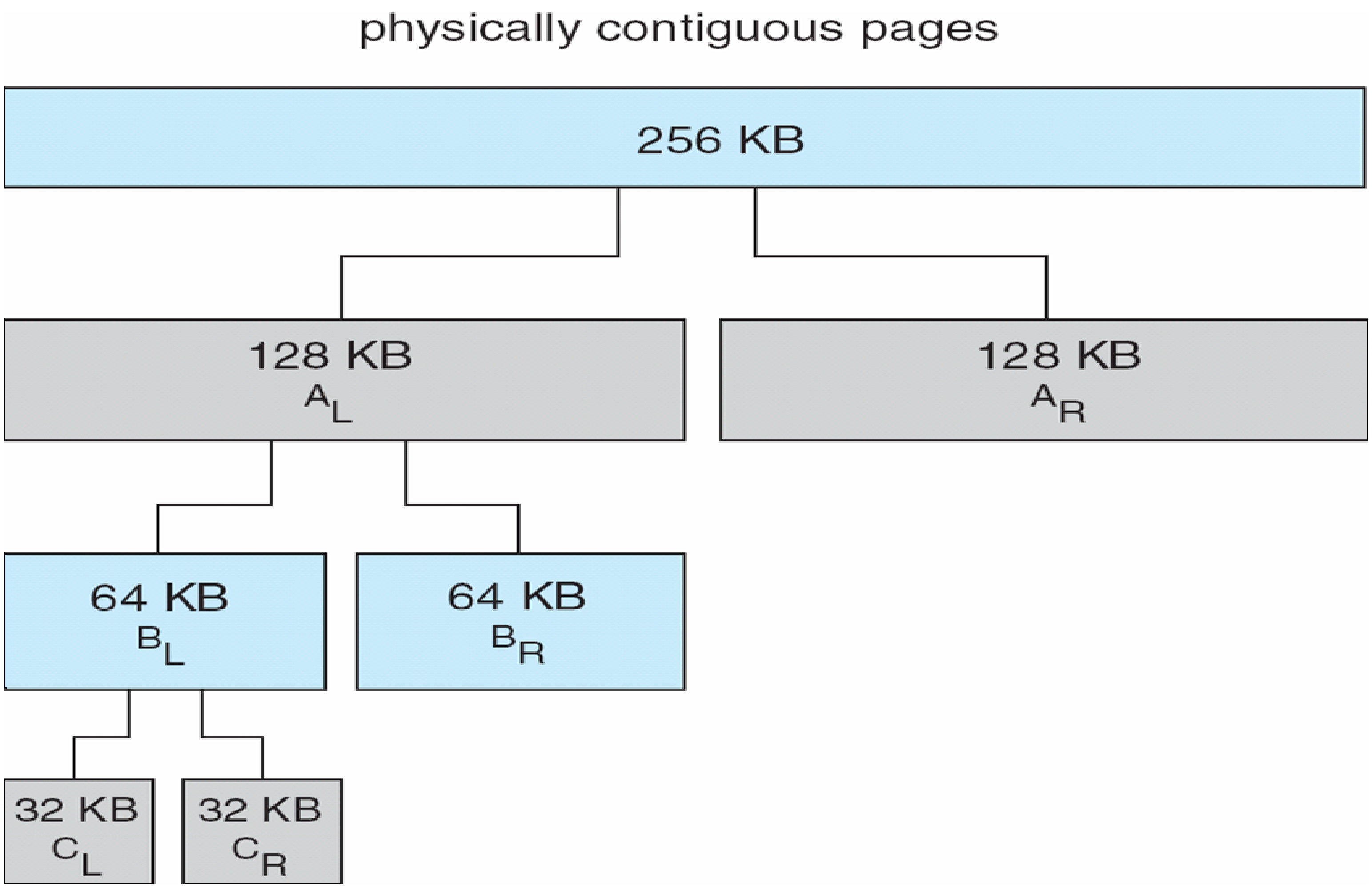
- Which free chunks should service request（分配那个块）
  - Ideally avoid fragmentation… requires future knowledge
- Split free chunks to satisfy smaller requests
  - Avoids internal fragmentation
- Coalesce（合并） free blocks to form larger chunks
  - Avoids external fragmentation

- 分区管理问题?

Buddy System

- Allocates memory from fixed-size segment consisting of physically-contiguous pages
- Memory allocated using **power-of-2 allocator**
  1. Satisfies requests in units sized as power of 2
  2. Request rounded up to next highest power of 2
  3. 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
- 按2的幂次方大小拆分或合并，分配最接近大小的块.

Buddy System Allocator

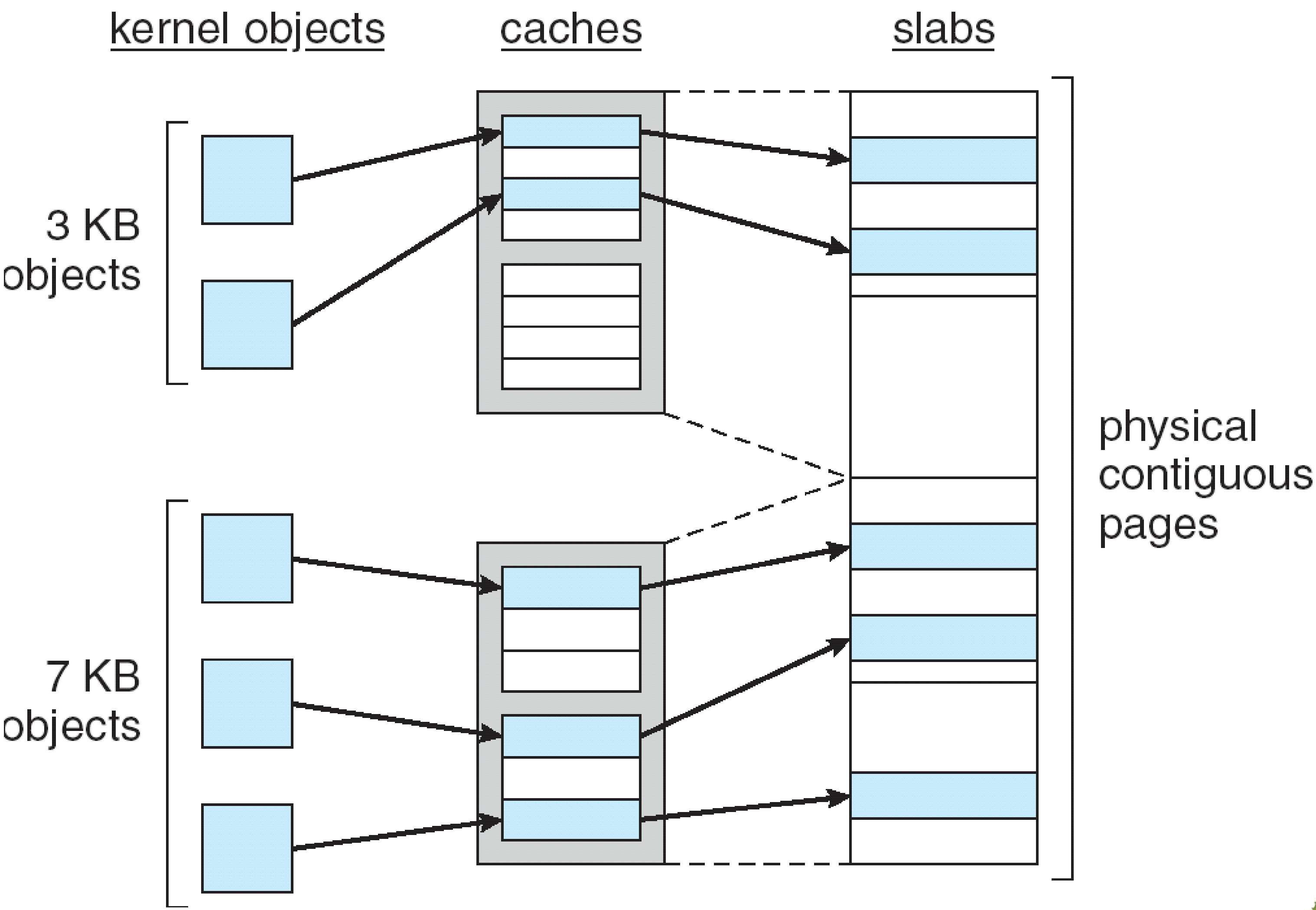


Slab Allocator

- Alternate strategy
- **Slab** is one or more physically contiguous pages
- **Cache** consists of one or more slabs（用于cache管理分配）
- 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

将分配的内存分割成各种尺寸的块（chunk），并把尺寸相同的块分成组（chunk的集合）

Slab Allocation



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 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? α near zero $\Rightarrow$ prepaging loses
    - 缺页中断开销/预分配的页却没有使用的开销=？

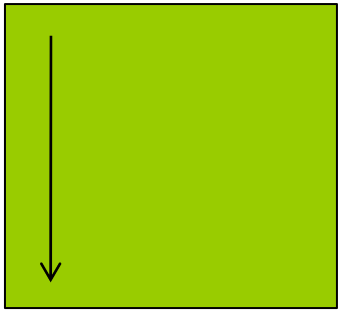
Other Issues – Page Size

- Page size selection must take into consideration:
  - Fragmentation: 页面大，则内碎片大;
  - table size: 页面小，则页表占用的空间大;
  - I/O overhead: 磁盘I/O时间中传输时间和数据量有关系;
  - Locality;

Other Issues – Program Structure

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

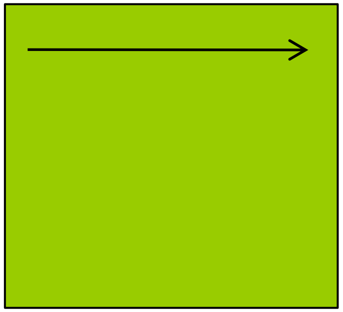
```
for (j = 0; j <128; j++)
     for (i = 0; i < 128; i++)
           data[i,j] = 0;
```



128 x 128 = 16,384 page faults

  - Program 2

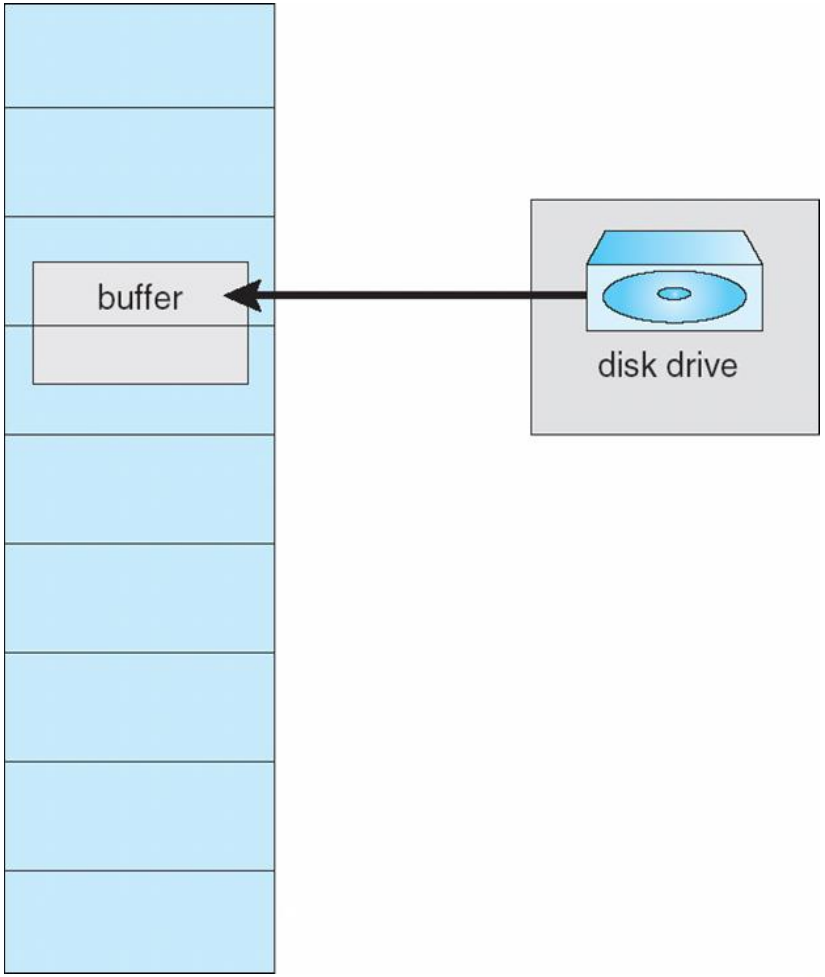
```
for (i = 0; i < 128; i++)
     for (j = 0; j < 128; j++)
           data[i,j] = 0;
```



128 page faults

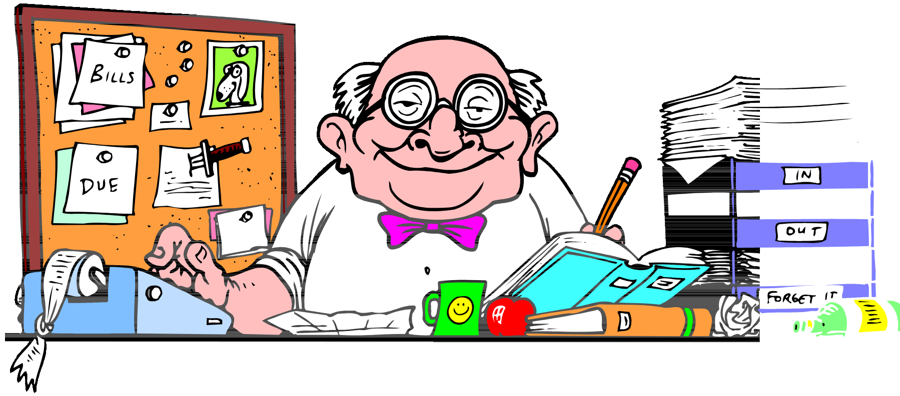
Other Issues – I/O interlock

- **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
- 不能置换



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

Caching

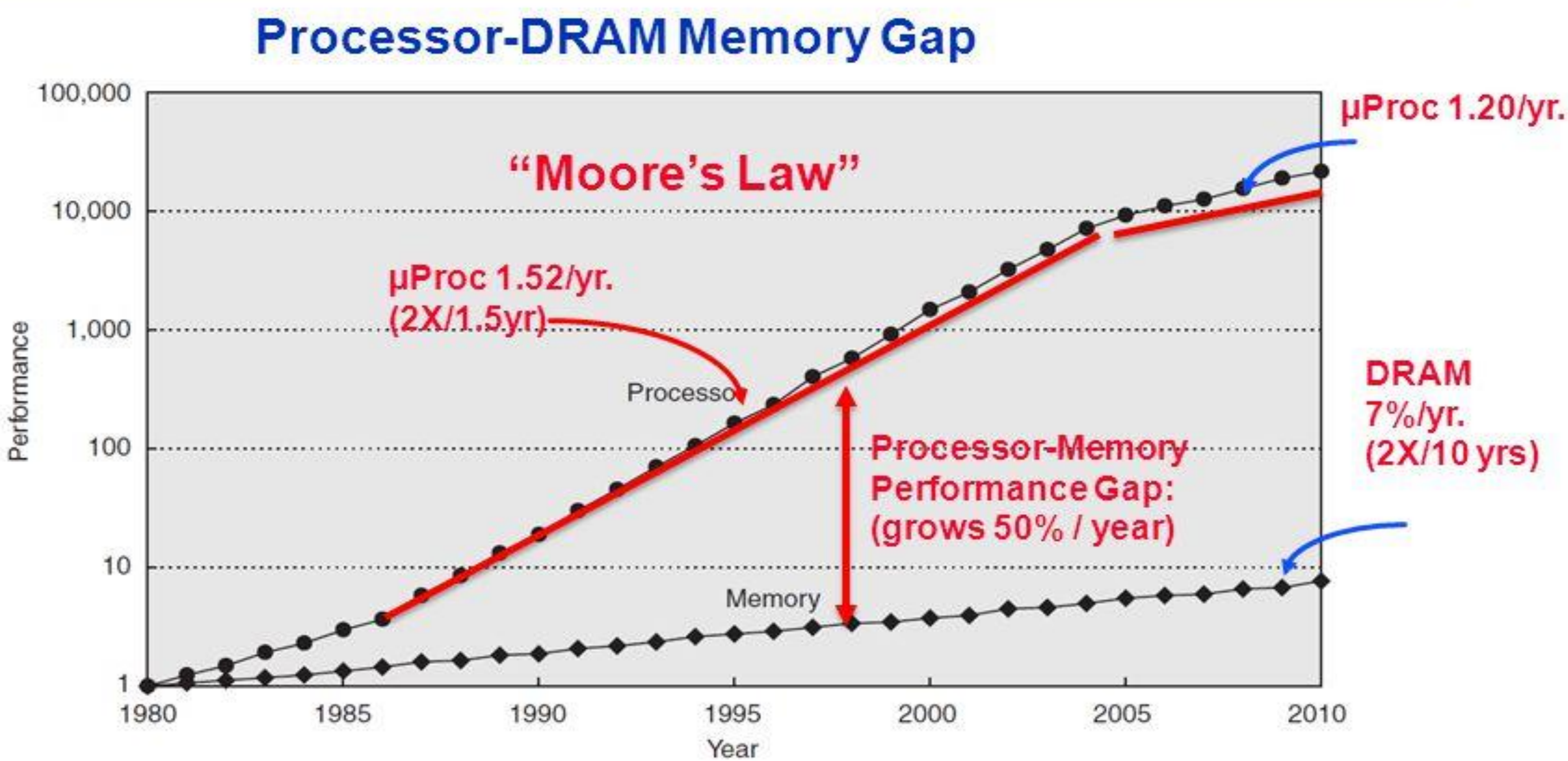


- Cache: a repository for copies that can be accessed more quickly than the original
- It underlies many of the techniques that are used today to make computers fast（提高性能的技术）
  - ✓ Can cache: memory locations, address translations, pages, file blocks, file names, network routes, etc…
- Important measure: Average Access time =

  (Hit Rate x Hit Time) + (Miss Rate x Miss Time)

Why Bother with Caching?

Memory Wall Problem

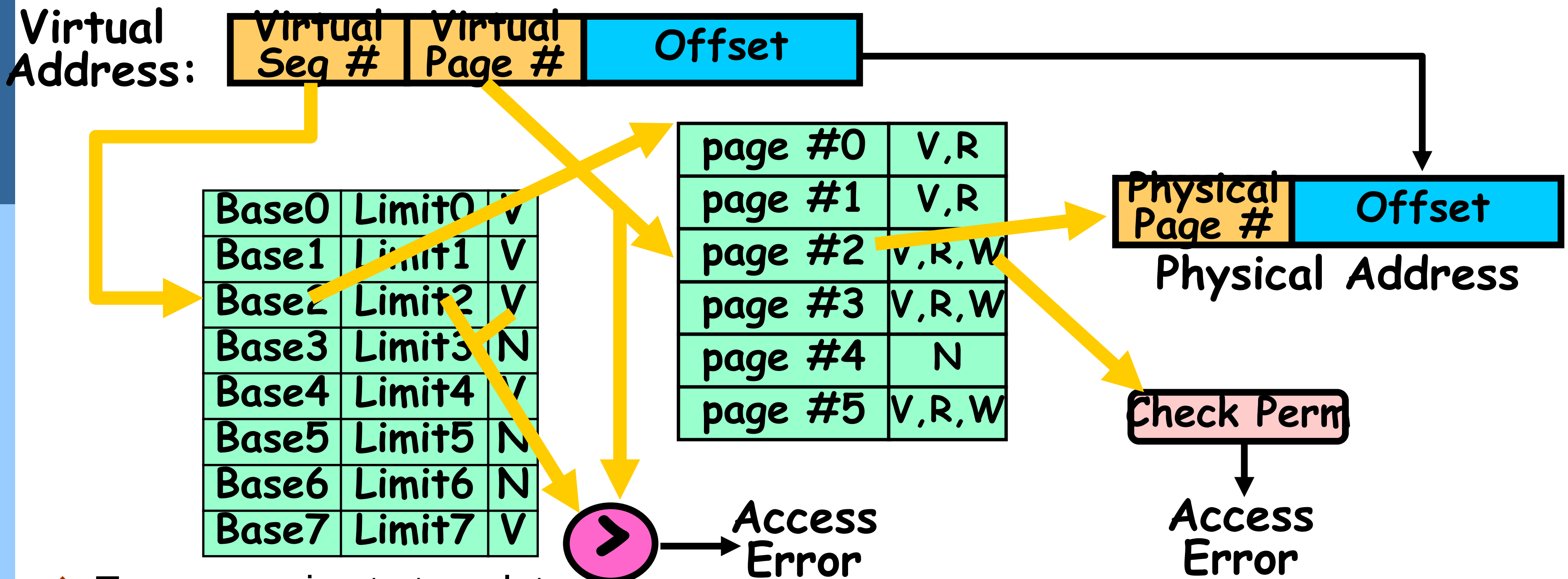


- 1980: no cache in micro-processor; 2010: 3-level cache on chip, 4-level cache off chip
- 1989 the first Intel processor with on-chip L1 cache was Intel 486, 8KB size
- 1995 the first Intel processor with on-chip L2 cache was Intel Pentium Pro, 256KB size
- 2003 the first Intel processor with on-chip L3 cache was Intel Itanium 2, 6MB size

Xian-He Sun

Source: Computer Architecture A Quantitative Approach

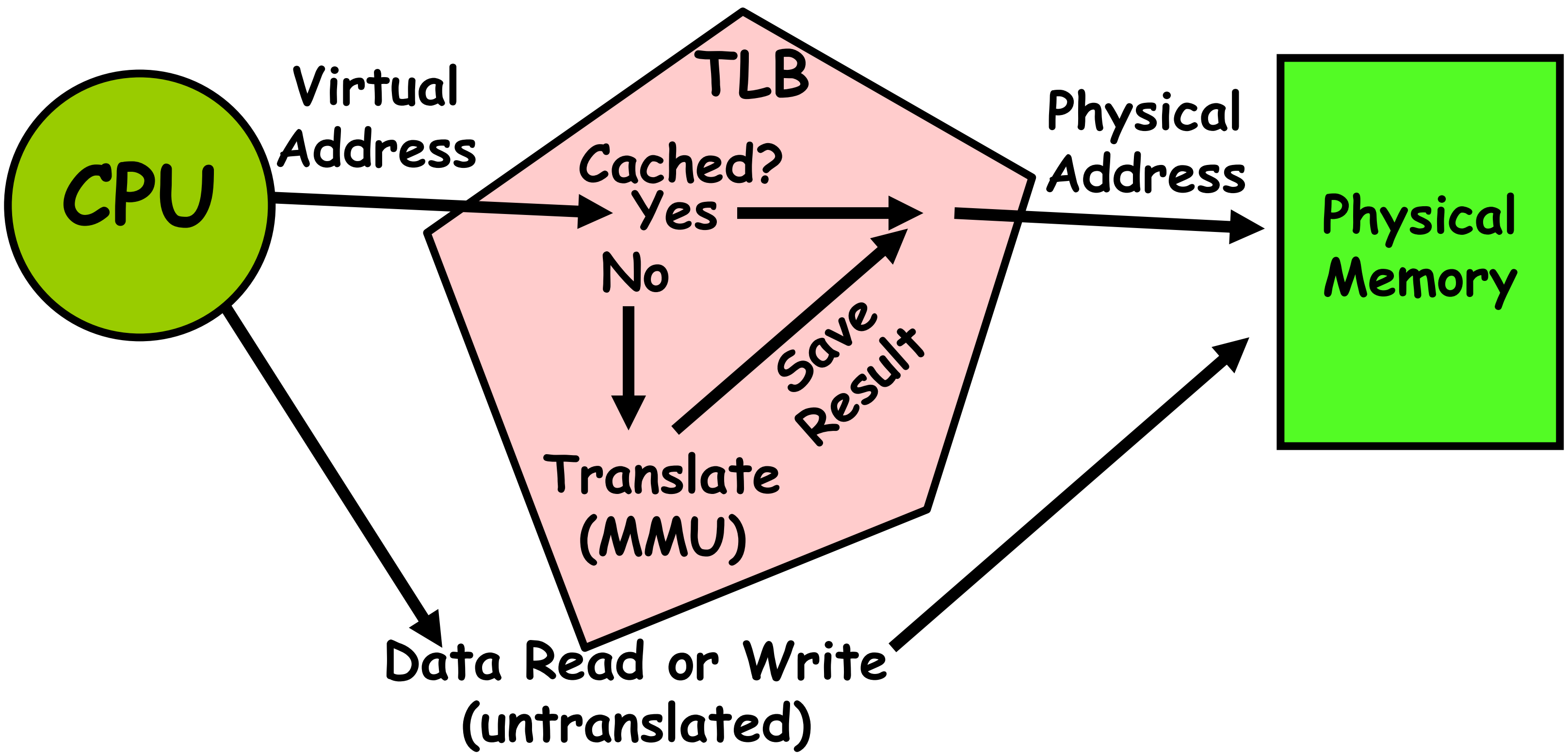
Another Major Reason to Deal with Caching



- Too expensive to translate on every access
  - ✓ At least two DRAM accesses per actual DRAM access
  - ✓ Or: perhaps I/O if page table partially on disk!
- Even worse problem: What if we are using caching to make memory access faster than DRAM access???
- Solution? Cache translations!
  - ✓ Translation Cache: TLB ("Translation Lookaside Buffer")

67

Caching Applied to Address Translation



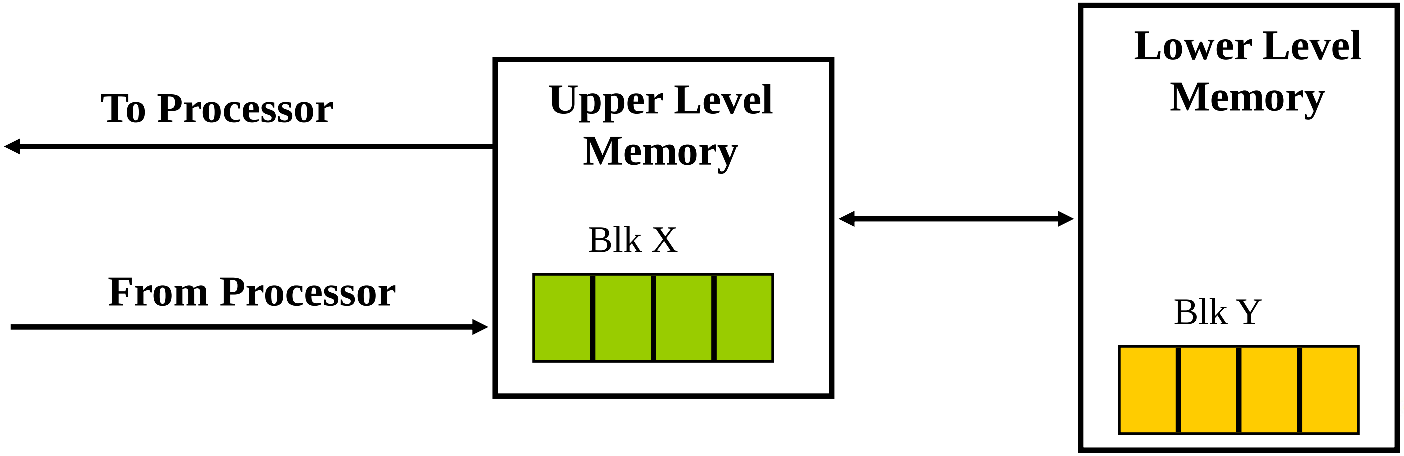
68

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（增加页尺寸产生内碎片）
- Provide Multiple Page Sizes
  - ✓ This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation

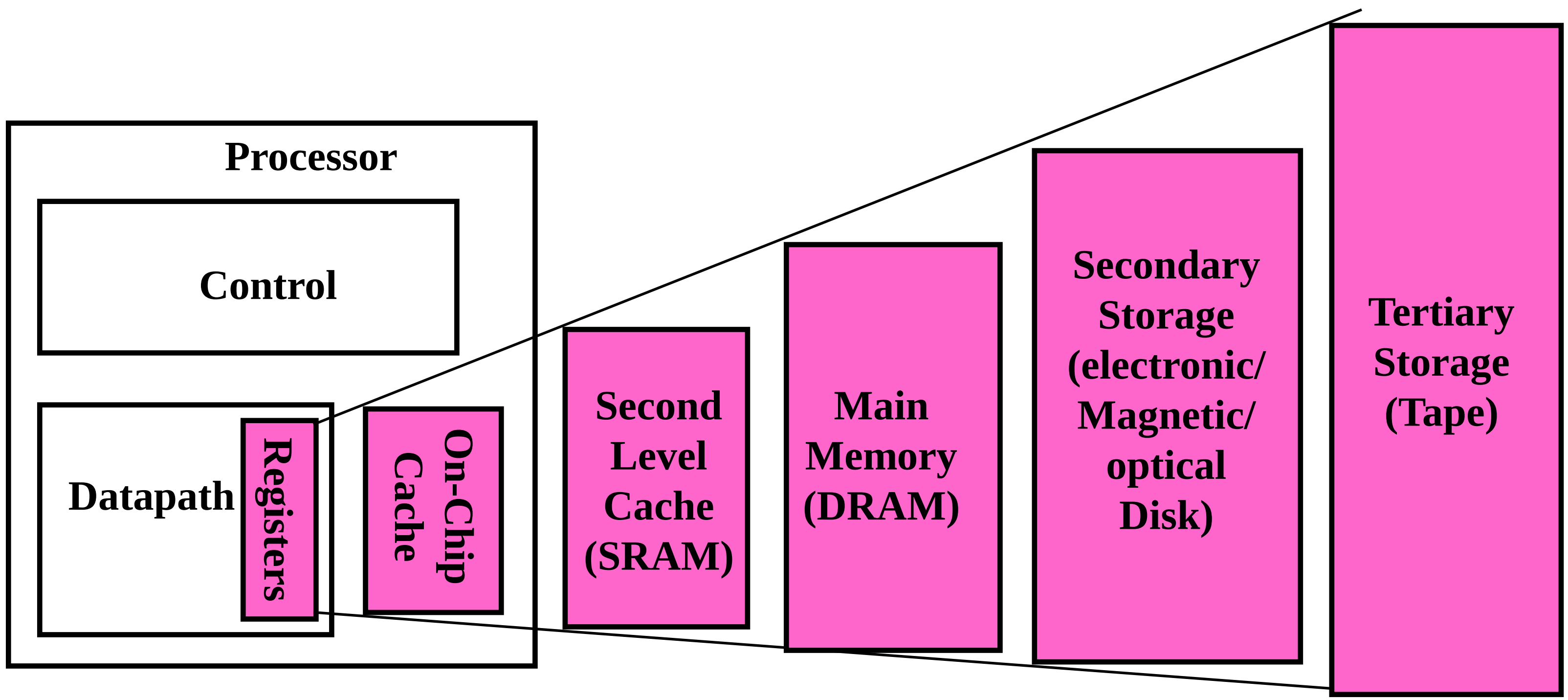
Why Does Caching Help? Locality!

- Temporal Locality (Locality in Time): 最近访问的数据项更接近处理器
- Spatial Locality (Locality in Space):   相邻的数据项更接近处理器



Review: Memory Hierarchy of a Modern Computer System

- Take advantage of the principle of locality to:
  - ✓ Present as much memory as in the cheapest technology
  - ✓ Provide access at speed offered by the fastest technology



Speed (ns):	1s	10s-100s	100s	10,000,000s (10s ms)	10,000,000,000s (10s sec)
Size (bytes):	100s	Ks-Ms	Ms	Gs	Ts

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≤

A process may be assigned as many pages

≤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（空闲页数阈值，大于它则进行分页，小于它则执行回收）
  - *Desfree* – threshold parameter to increasing paging（最小空闲页阈值，小于它则加快页面回收）
  - *Minfree* – threshold parameter to being swapping Paging is performed by *pageout* process, Pageout scans pages using modified clock algorithm（最小空闲页极限阈值，小于它则每次请求都进行页面回收）
  - *Scanrate* is the rate at which pages are scanned. This ranges from *slowscan* to *fastscan*（扫描的频率）
- Pageout is called more frequently depending upon the amount of free memory available

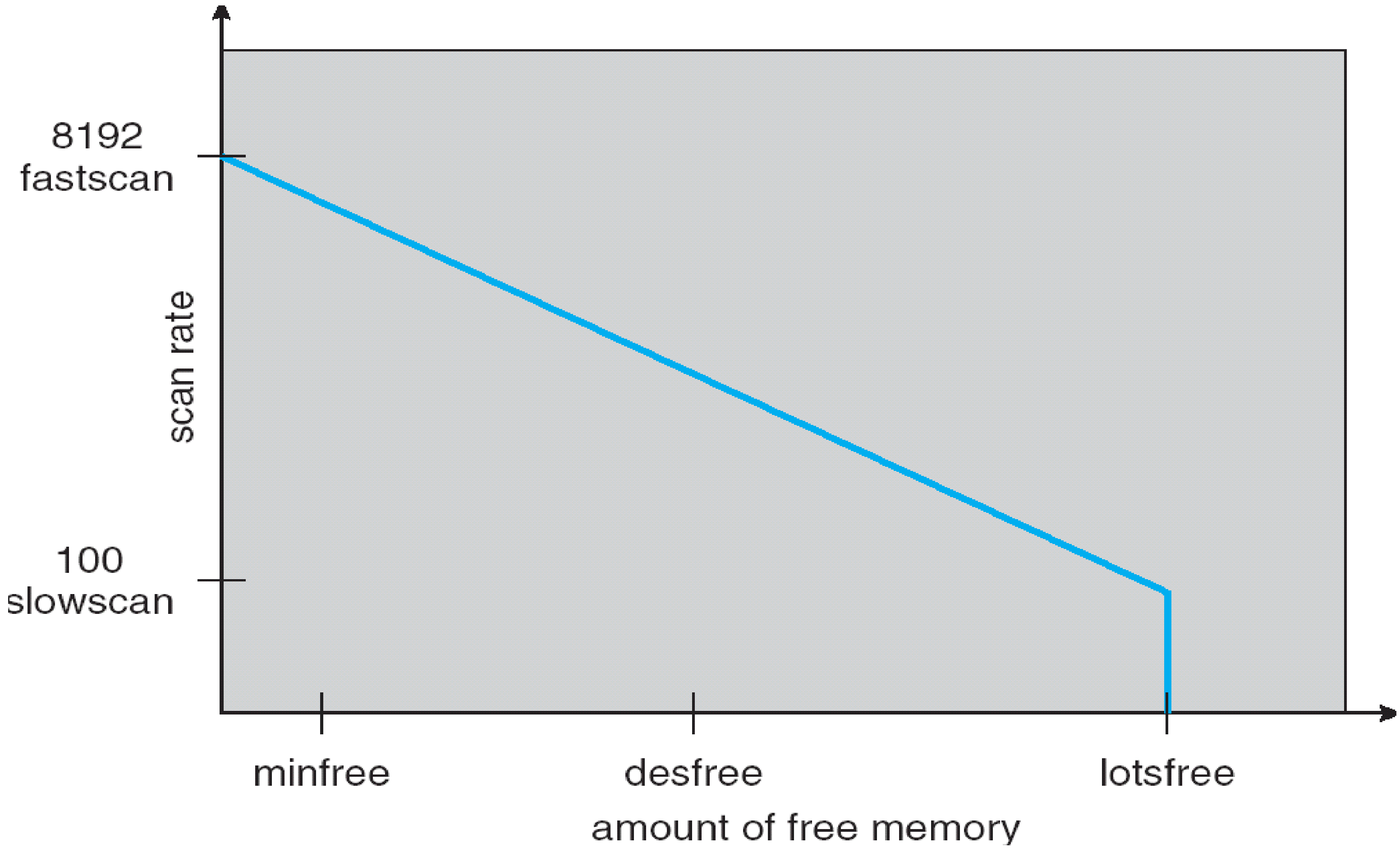
Solaris

- *Lotsfree* – threshold parameter (amount of free memory) to begin paging
  - *Lotsfree* is typically set to 1/64 the size of the physical memory
  - 4 times per second, the kernel checks whether the amount of free memory is less than lotsfree
    - If below *lotsfree*, a process pageout starts up, which is similar to the second-chance algorithm
- If free memory falls below *desfree*, pageout will run 100 times per second with the intention of keeping at least *desfree* free memory available

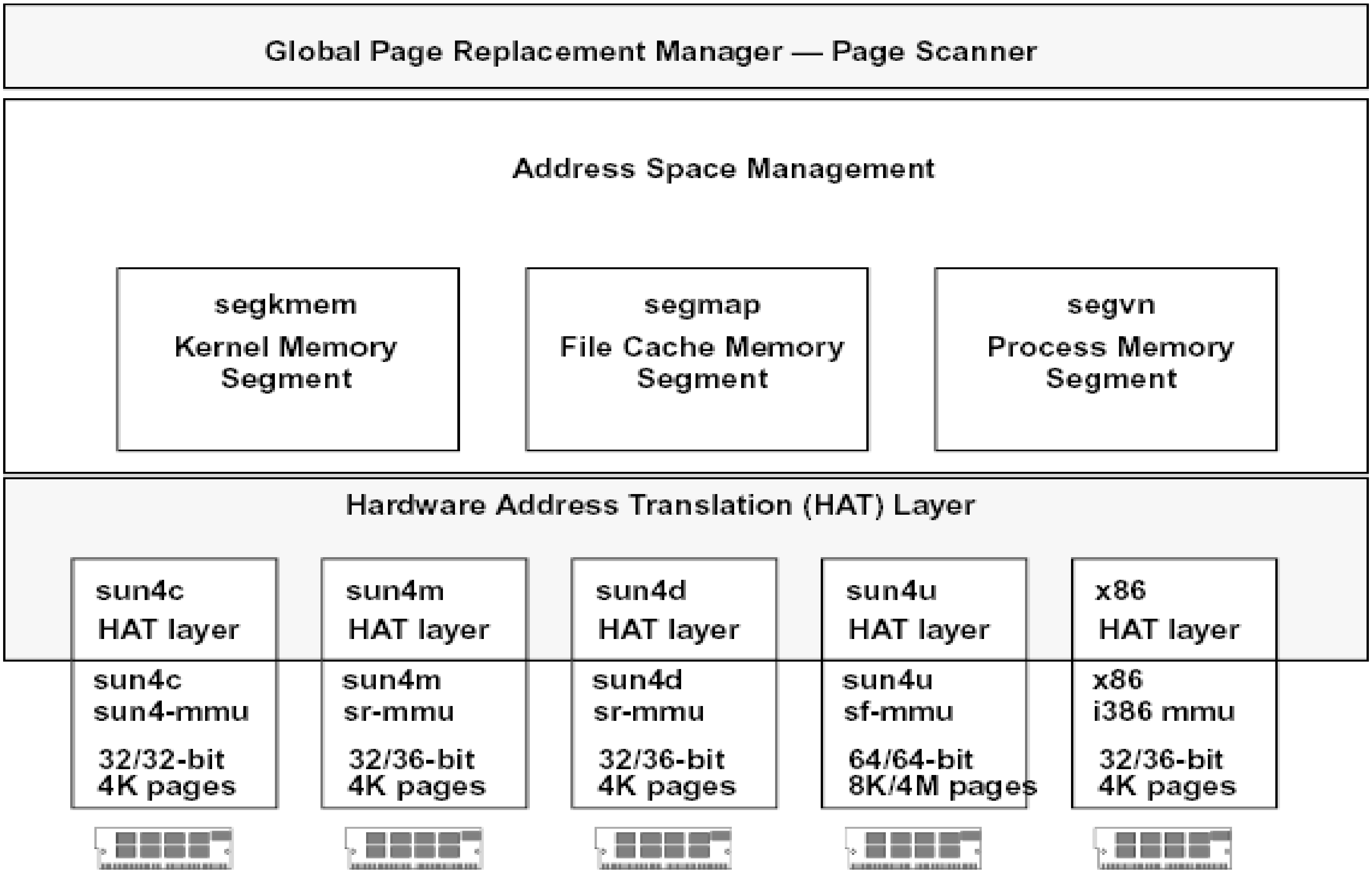
Solaris

- If the pageout is unable to keep free memory at *desfree* for a 30-second average, the kernel begins swapping processes.
  - Free all pages allocated to swappped processes.
  - The kernel looks for processes that have been idle for a long time.
- If the system unable to maintain free memory at *minfree*, the pageout process is called for every request for a new page.

Solaris 2 Page Scanner



Solaris Virtual Memory Layers



assignment

- 9.4
- 9.8
- 编程题：编写一个程序，实现本章所述的FIFO、LRU和最优页面置换算法。首先，生成一个随机的页面引用串，页码范围为0~9.将这个随机页面引用串应用到每个算法，并记录每个算法引起的缺页错误的数量。实现置换算法，以便页面帧的数量可以从1~7。假设采用请求分页。