Chapter 9: Virtual Memory

肖 卿 俊
办公室：计算机楼212室
电邮：csqjxiao@seu.edu.cn
主页：http://cse.seu.edu.cn/PersonalPage/csqjxiao
电话：025-52091022

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 working-set model
- To explain the IPC model based on memory sharing; To examine the differences between shared memory and memory-mapped files
- To explore how kernel memory is managed

Chapter 9: Virtual Memory

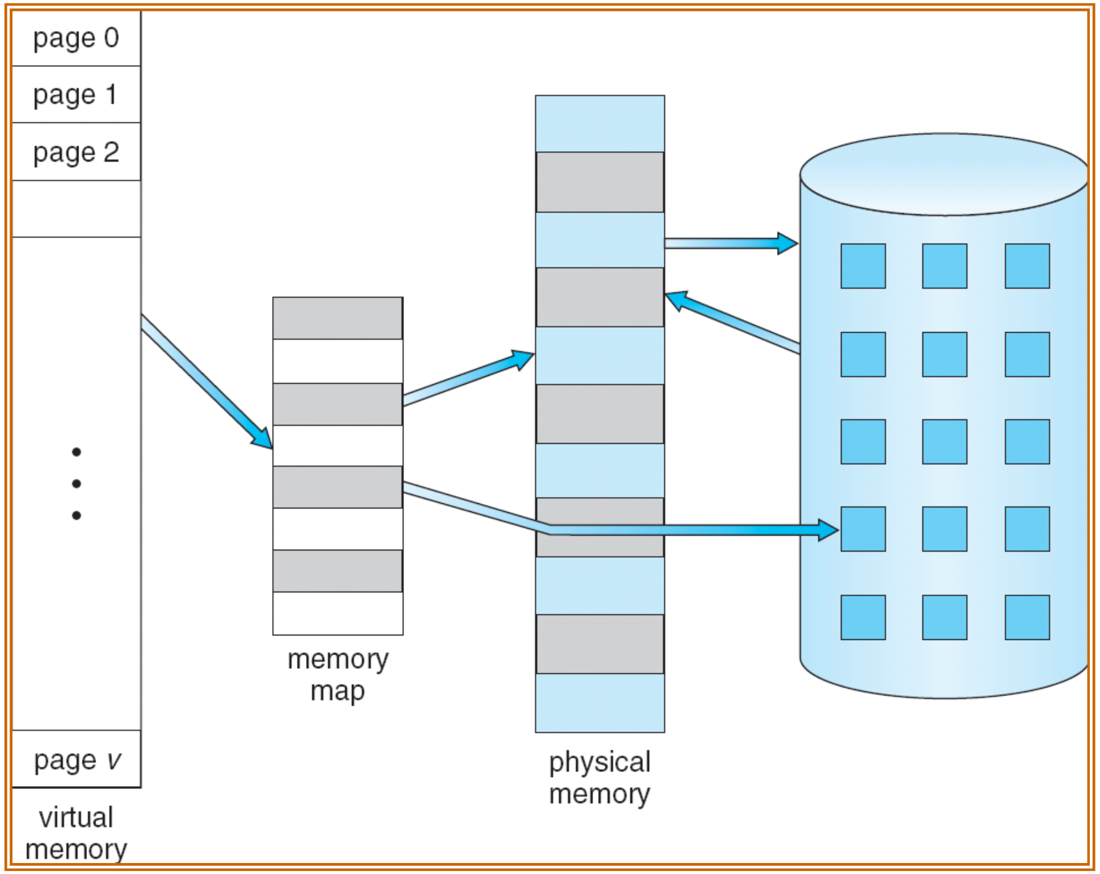
- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing and Working Set Model
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples

Background

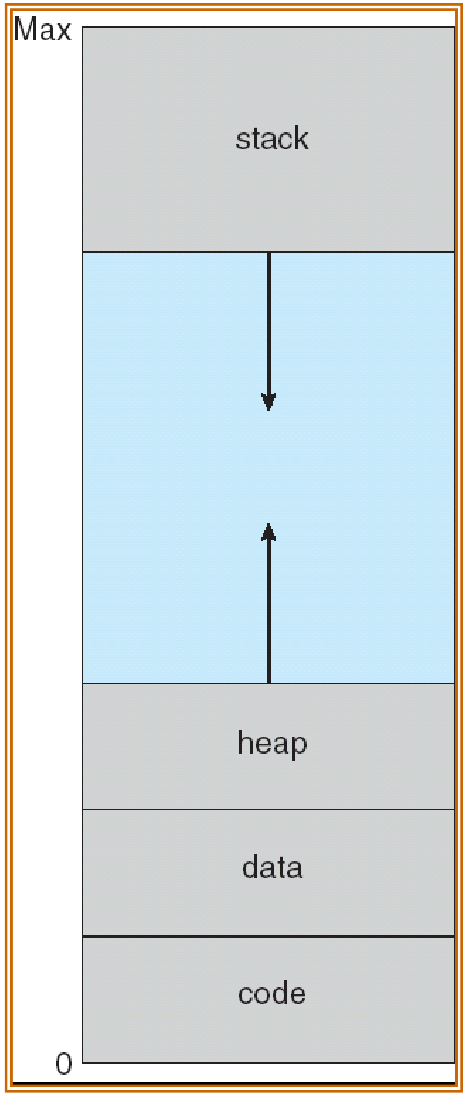
- We previously talked about an entire process swapping into or out of main memory
- **Idea of virtual memory** – separation of user logical memory from physical memory.
  - Only part of the program needs to be kept in main memory for execution.
  - Logical address space can therefore be much larger than physical address space.
  - More programs can be run at the same time
  - Less I/O is needed than loading or swapping

Two Kinds of Implementation for Virtual Memory

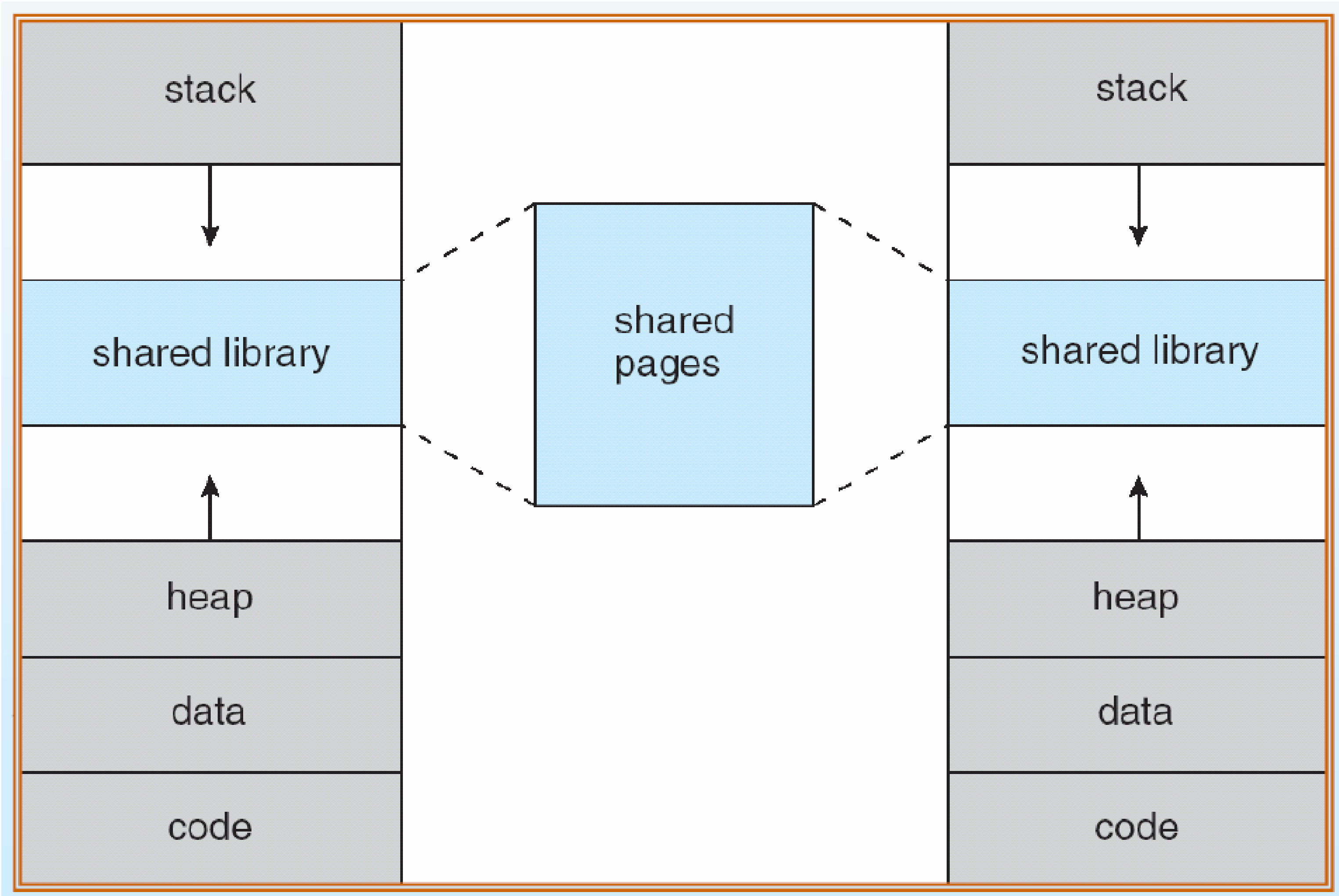
- Virtual memory can be implemented via:
  - Demand paging （按需调页）
  - Demand segmentation（按需调段）



Virtual-Address Space



Shared Library Using Virtual Memory



Chapter 9: Virtual Memory

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing and Working Set Model
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples

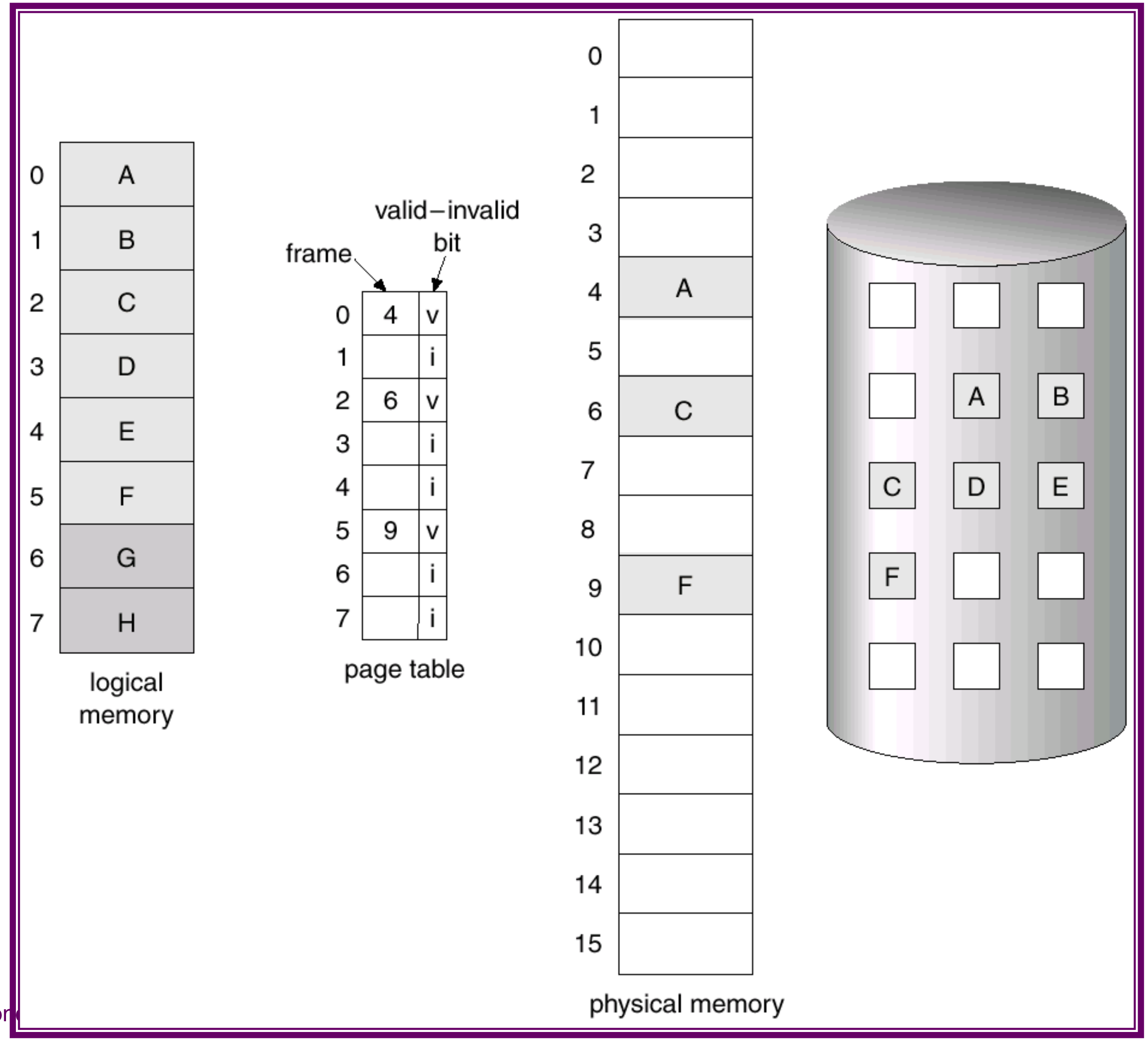
Demand Paging

- 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
- **Pure demand paging**– never bring a page into memory unless page will be needed

Valid-Invalid Bit

- With each page table entry, a valid-invalid bit is associated
  - (1 $\Rightarrow$ in-memory, 0 $\Rightarrow$ 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.

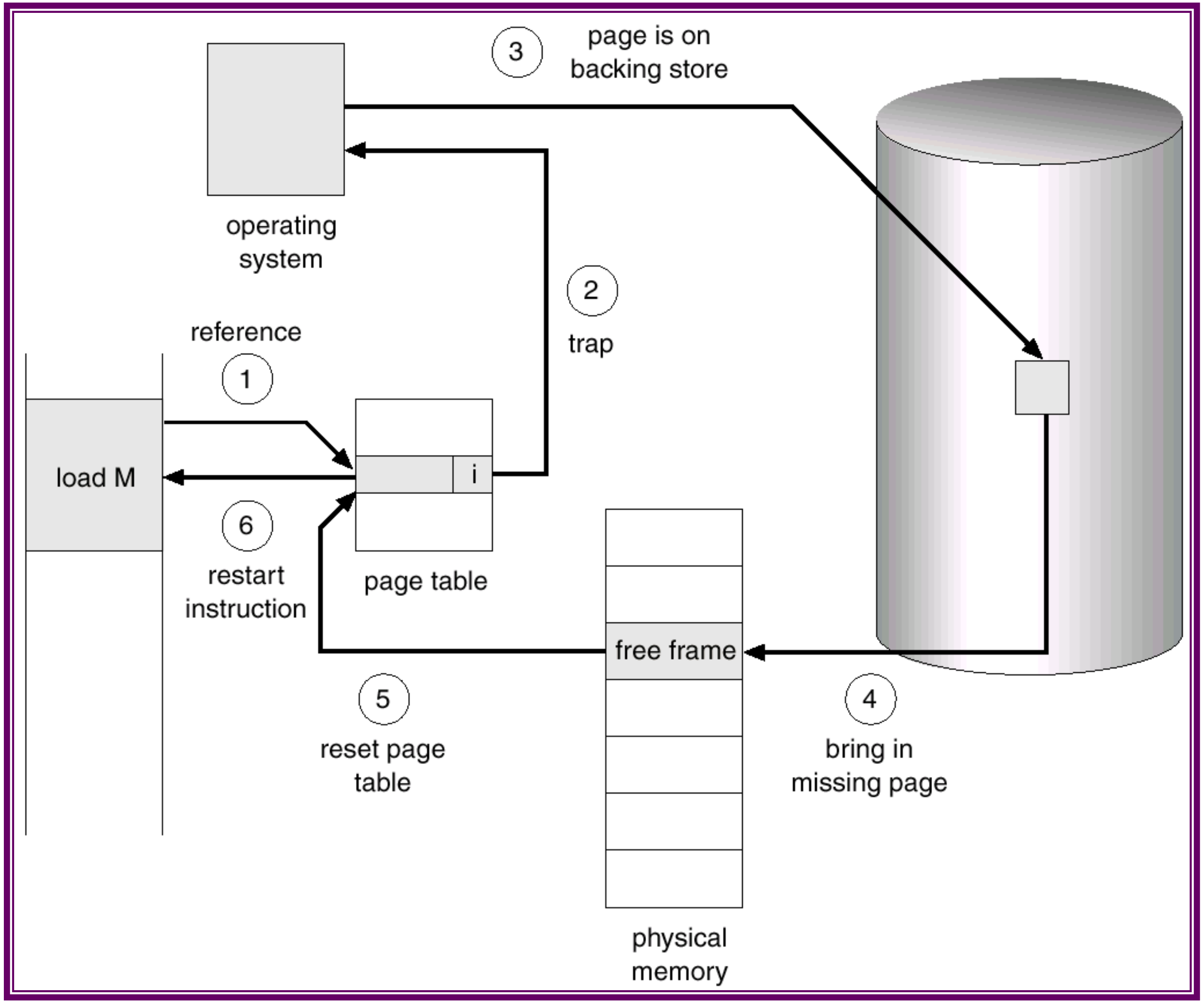
Page Table When Some Pages Are Not in Main Memory



Steps in Handling a Page Fault

- If there is ever a reference to a page, first reference will trap to OS $\Rightarrow$ page fault
- OS looks at another 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

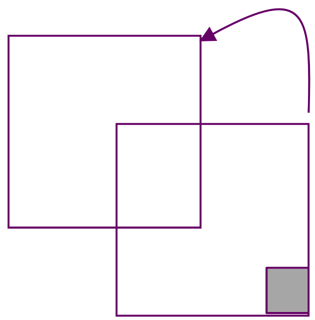
Steps in Handling a Page Fault



More Details about Restarting an Instruction

- The restart will require fetching instruction again, decoding it again, fetching the two operands again, and applying it again

- Difficulty arises when an instruction may modify multiple virtual pages
  - For example, block move operation



  - Auto increment/decrement location
  - Restart the whole operation?
    - ✓What if source and destination overlap?
    - ✓The source may have been modified

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 overhead
[+ swap page out]
+ swap page in
+ restart overhead)

Demand Paging Example

- Memory access time = 1 microsecond
- Swap Page Time = 10 millisec = 10000 microsec
- Assume 50% of the time the page that is being replaced has been modified and therefore needs to be swapped out.
- Ignore the cost of restarting an instruction.

- EAT = (1 - p) x 1 + p x (10000*50%+20000*50%)
  = (1 - p) x 1 + p x (15000)
  = 1 + 14999 x p   (in microsecond)

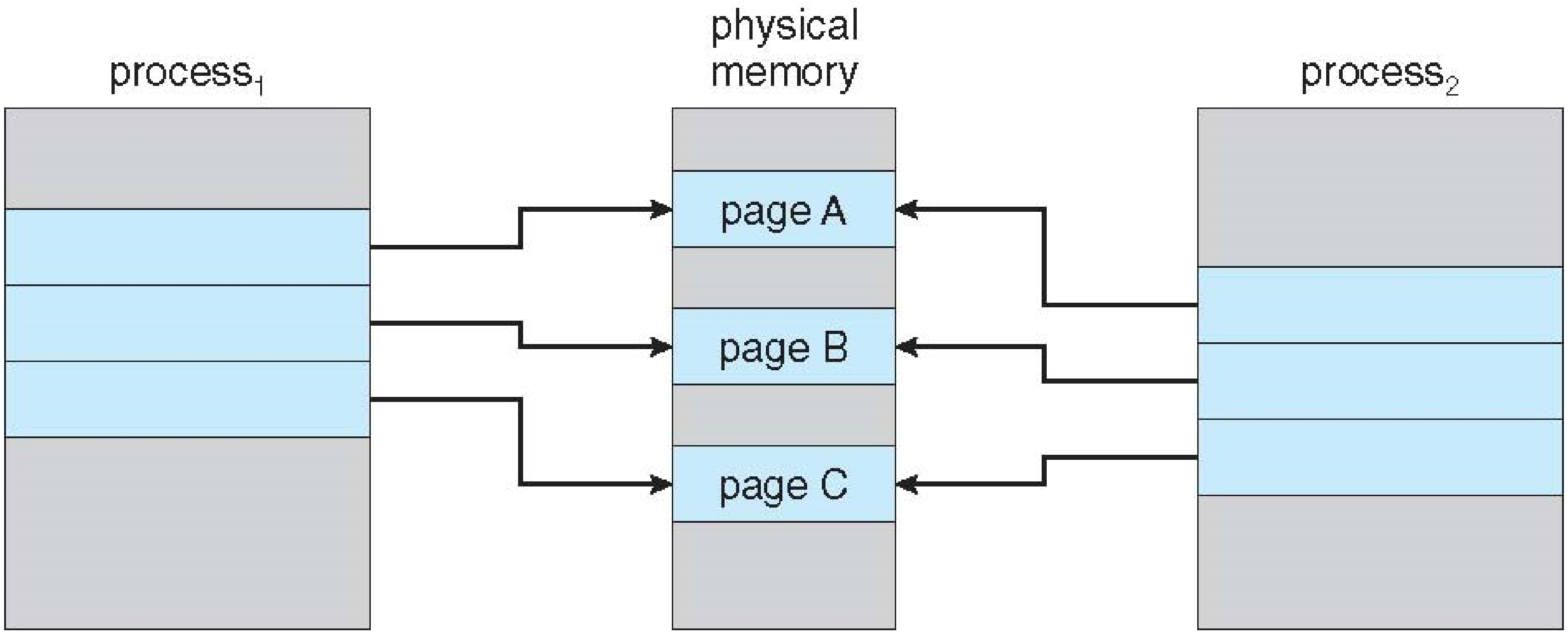
Chapter 9: Virtual Memory

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing and Working Set Model
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples

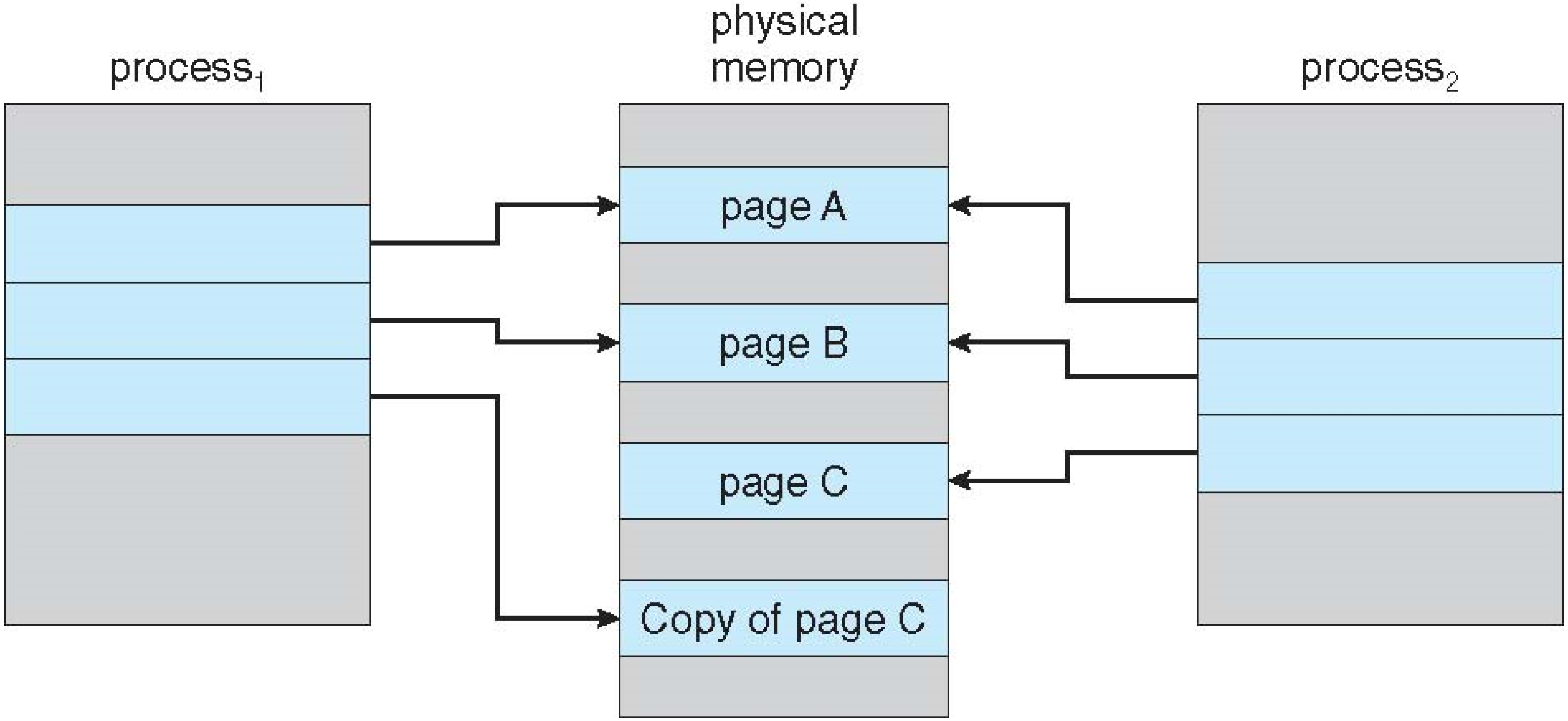
Copy-on-Write

- **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 **zero-fill-on-demand** pages
  - Why do we need to zero-out a page before allocating it to a process?
  - The pool should always have free frames for fast

Before Process 1 Modifies Page C



After Process 1 Modifies Page C



fork() and vfork()

- `vfork()`, a variation of `fork()` system call, has the parent suspend and the child without copying the page table of the parent
  - Useful in performance-sensitive applications where a child is created which then immediately issues an `execve()`.
- 以前的fork很低效，它创建一个子进程时，将会创建一个新的地址空间，并且拷贝父进程的资源，而往往在子进程中会执行exec调用，这样，前面的拷贝工作就是白费了。于是，设计者就想出了vfork，它产生的子进程刚开始暂时与父进程共享地址空间（其实就是线程的概念了）。因为这时候子进程在父进程的地址空间中运行，所以子进程不能进行写操作，并且在儿子“霸占”着老子的房子时候，要委屈父亲一下了，让他在外面歇着（阻塞），一旦儿子执行了execve 或者exit 后，相于儿子买了自己的房子了，这时候就相于分家了。

http://linux.die.net/man/2/vfork

An Example of fork() and vfork()

```
int main() {
    pid_t pid;
    int cnt = 0;
    pid = fork();
    if(pid<0)
        printf("error in fork!\n");
    else if(pid == 0) {
        cnt++;
        printf("Child process %d\n",getpid());
        printf("cnt=%d, ",cnt);
    }  else {
        cnt++;
        printf("Parent process %d\n",getpid());
        printf("cnt=%d, ",cnt);
    }
    return 0;
}
```

Execution Result:

Child process 5077, cnt=1

Parent process 5076, cnt=1

If we replace line 4 by pid = vfork(), then

Execution Result:

Child process 5077, cnt=1

Parent process 5076, cnt=1

段错误

Question: If the cnt variable on stack is shared between parent and child processes, why do we still see cnt =1?

Answer: vfork() differs from fork() in that the calling thread is suspended until the child terminates (either normally by exit() or abnormally after a fatal signal), or it makes a call to execve(). Until that point, the child shares all memory with its parent, including the stack.

Question: What if we insert exit(0) between line 10 and line 11?

Chapter 9: Virtual Memory

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing and Working Set Model
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples

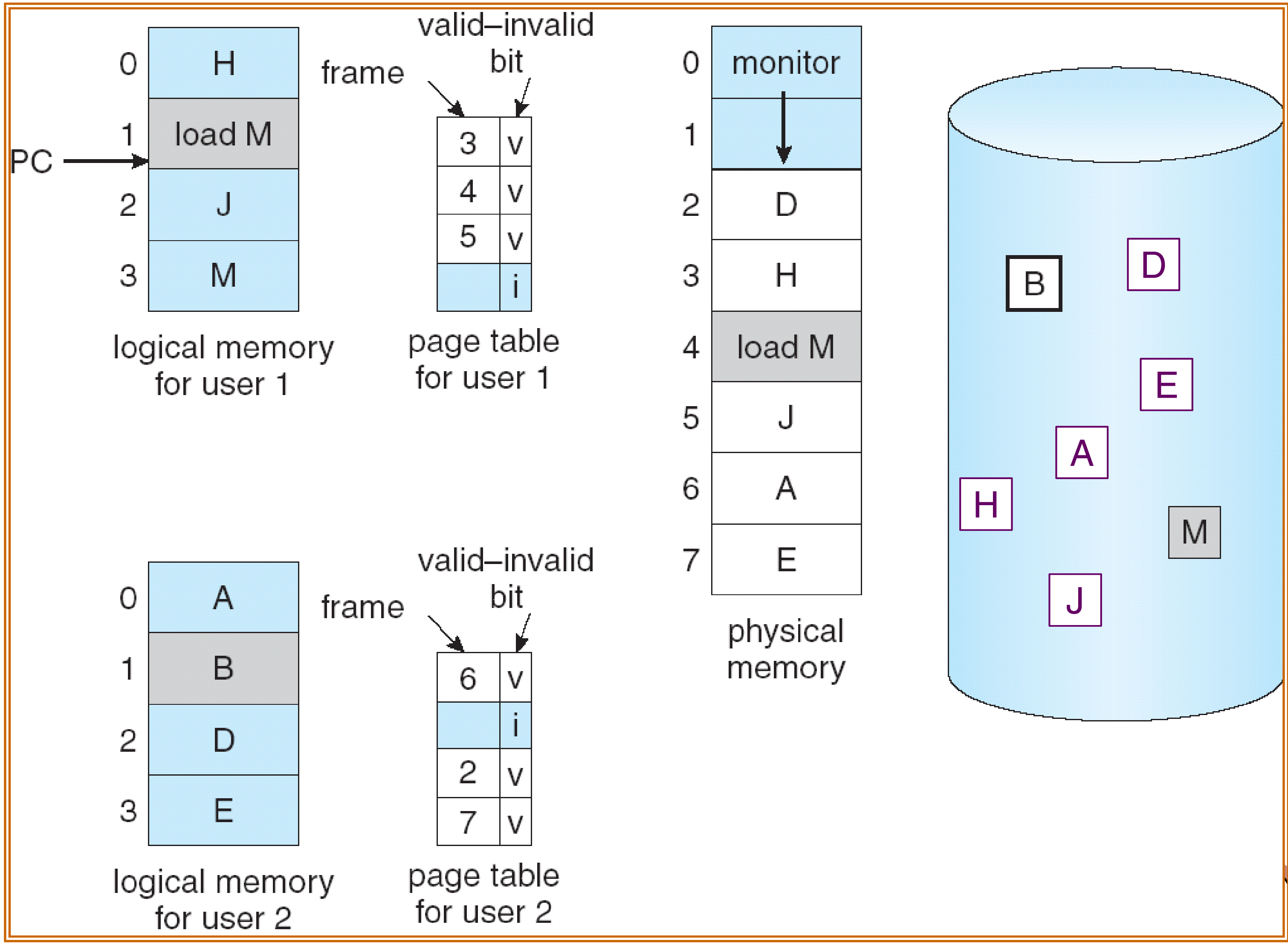
What Happens if There are no Free Frames?

- Used up by process pages
- Also in demand by the kernel, I/O buffers, …
- How much to allocate to each?
- Same page may be brought into memory several times
- Page replacement – find some page in memory, but not really in use, swap it out
  - Algorithm – terminate? swap out? replace the page?
  - 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.
- Use *modify* (*dirty*) *bit* to reduce the 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.

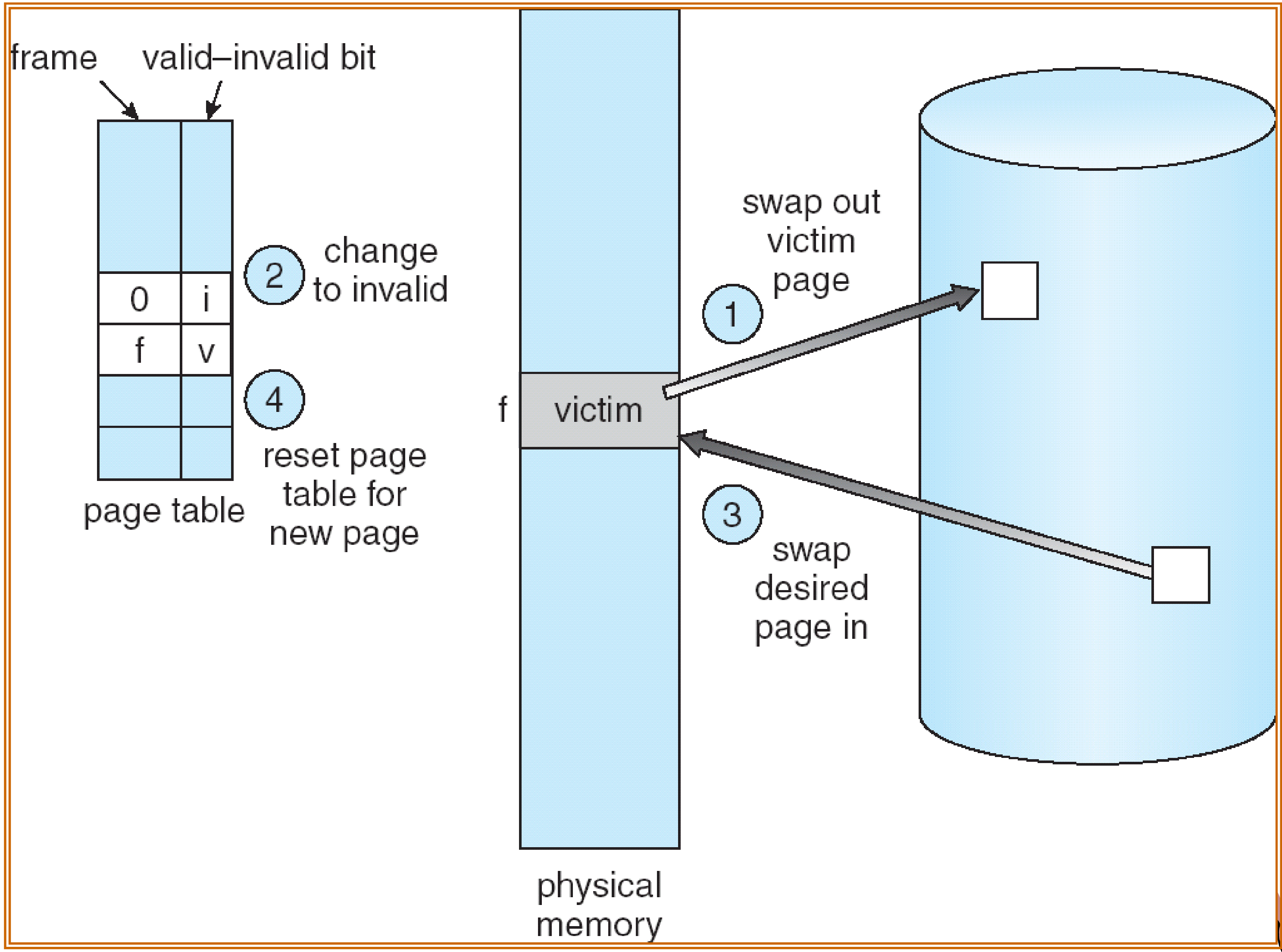
Need For Page Replacement



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 and swap it out
3. Read the desired page into the free frame.
4. Update the page and frame tables.
5. Restart the instruction.

Page Replacement

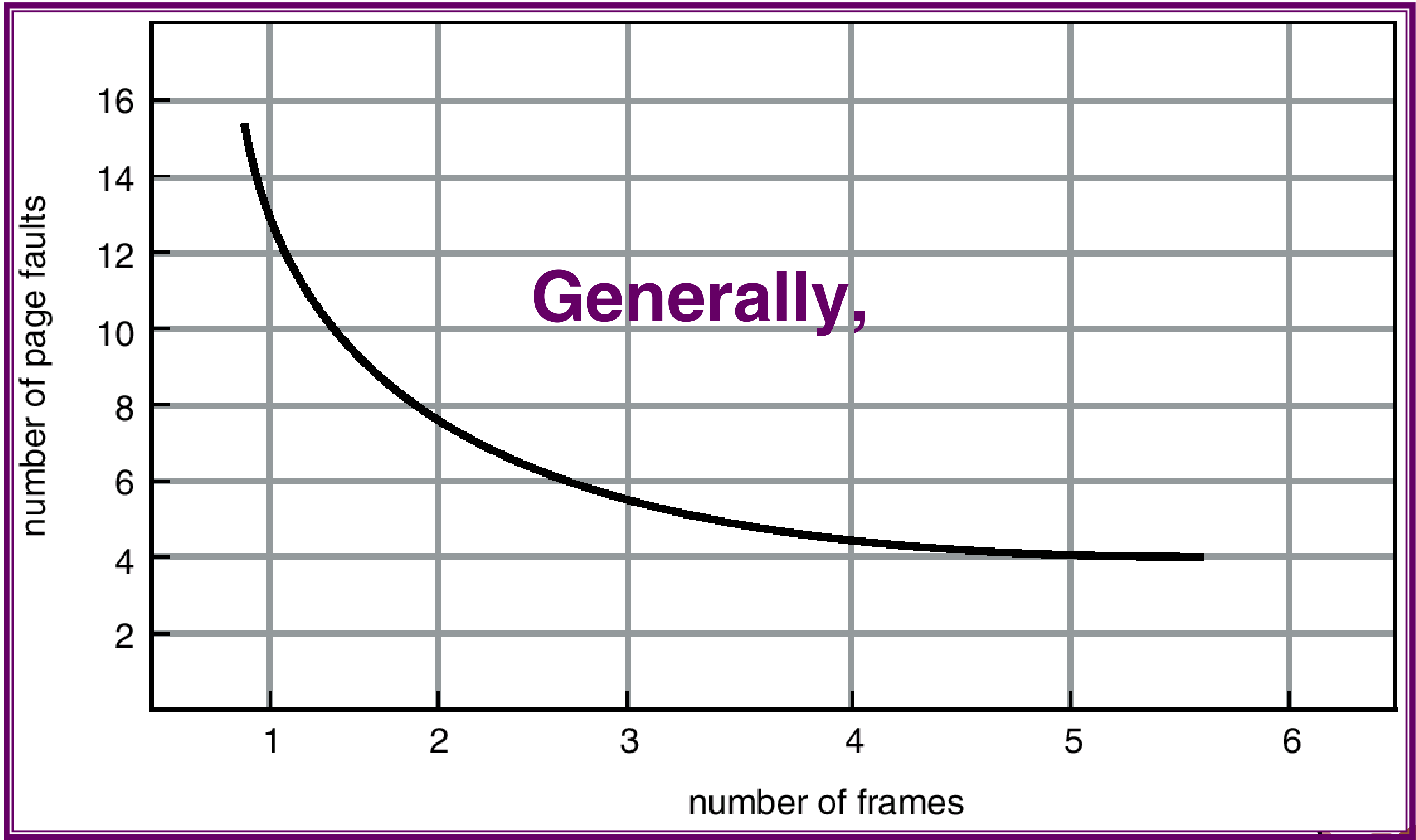


Page Replacement Algorithms

- 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.

The Number of Page Faults vs. The Number of Frames



First-In-First-Out (FIFO) Page Replacement

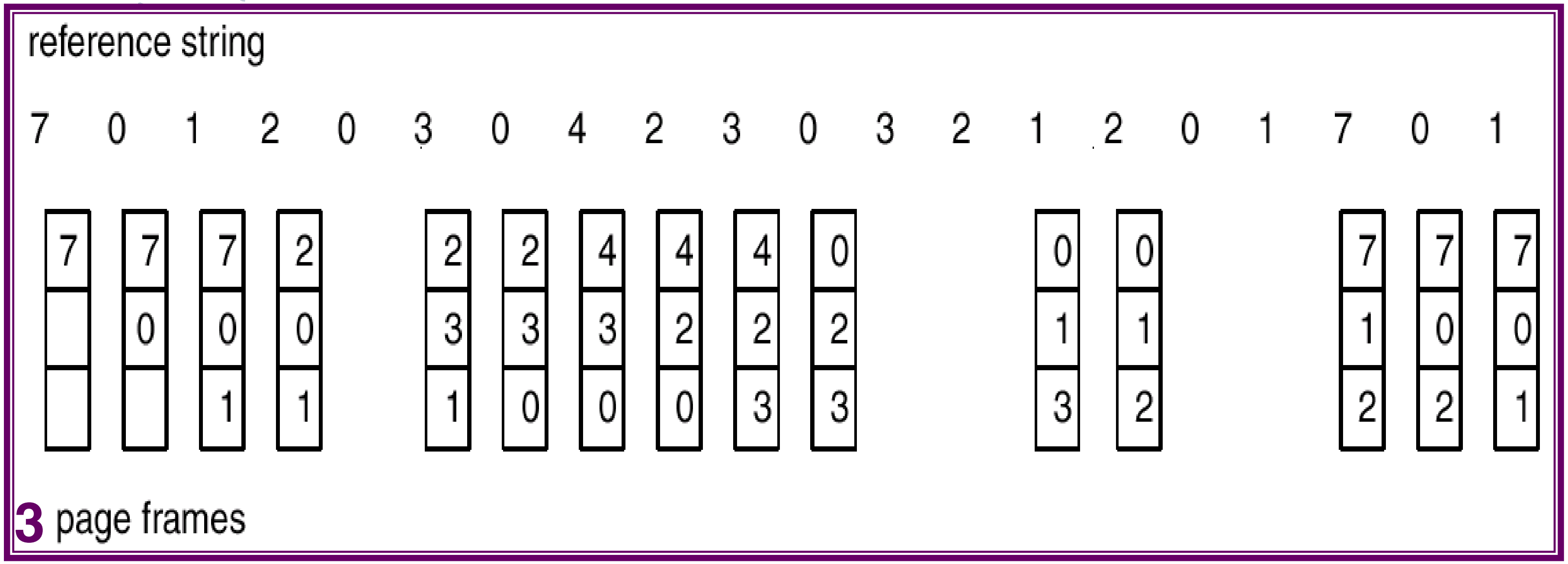
- Reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1
- 3 frames (3 pages can be in memory at a time per process)

reference string

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

3 page frames

FIFO Page Replacement



Belady's Anomaly for FIFO Algorithm

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

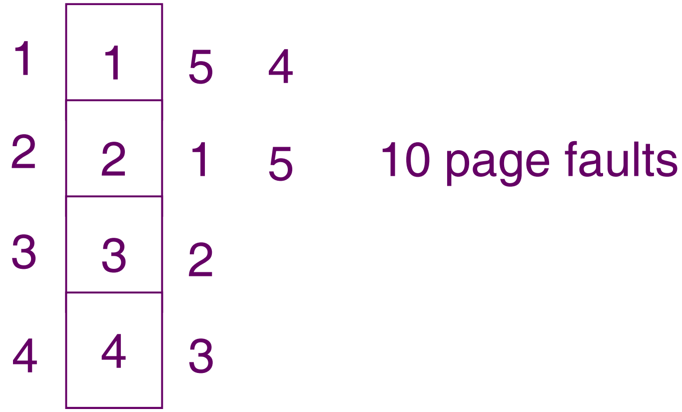
- Where there are 3 frames (3 pages can be in memory at a time per process)

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

9 page faults

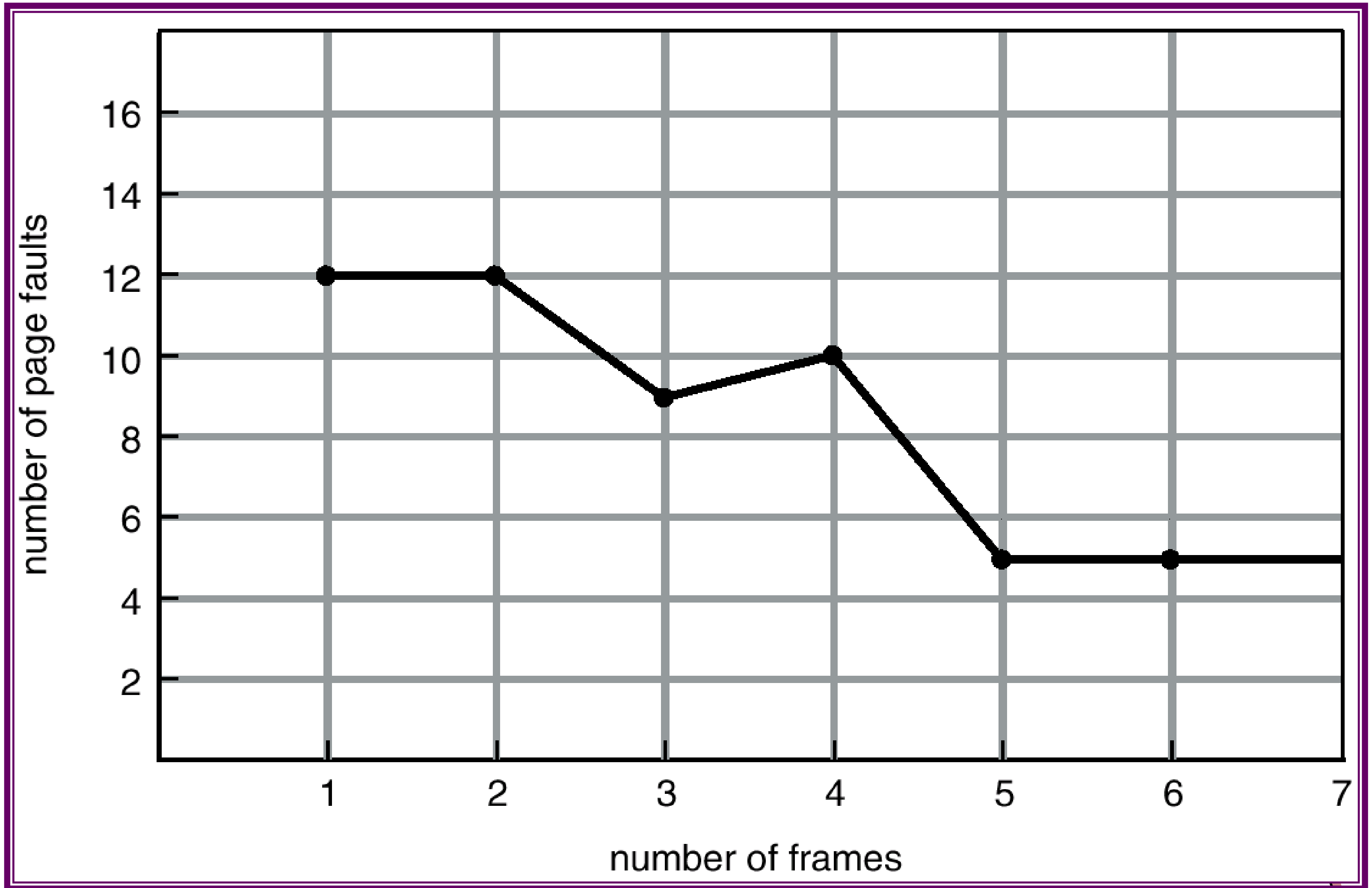
Belady's Anomaly for FIFO Algorithm

- When there are 4 frames



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

FIFO Illustrating Belady's Anamoly



Optimal Algorithm

- Replace page that will not be used for the longest period of time.
- How do you know this?
- 4 frames example

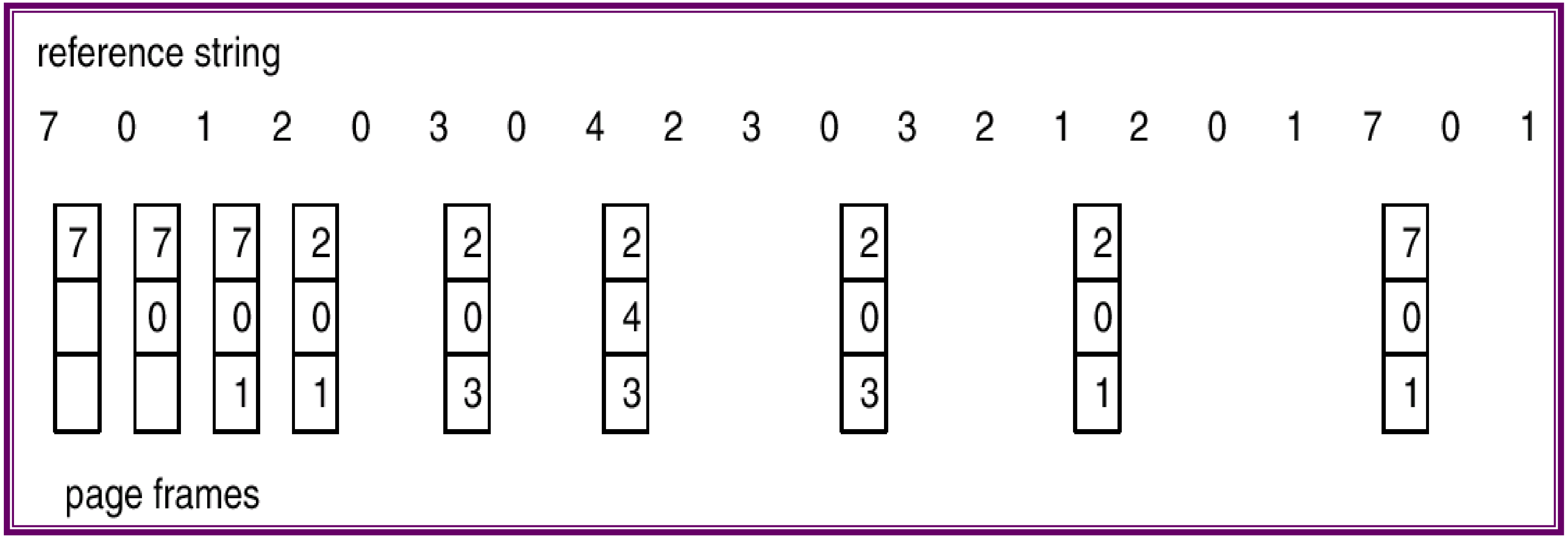
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

1	4
2	
3	
4	5

6 page faults

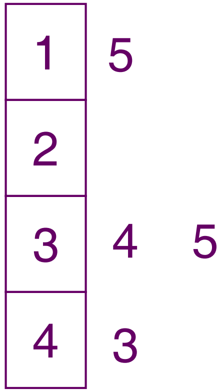
- Used for measuring how well your algorithm performs.

Optimal Page Replacement



Least Recently Used (LRU) Algorithm

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

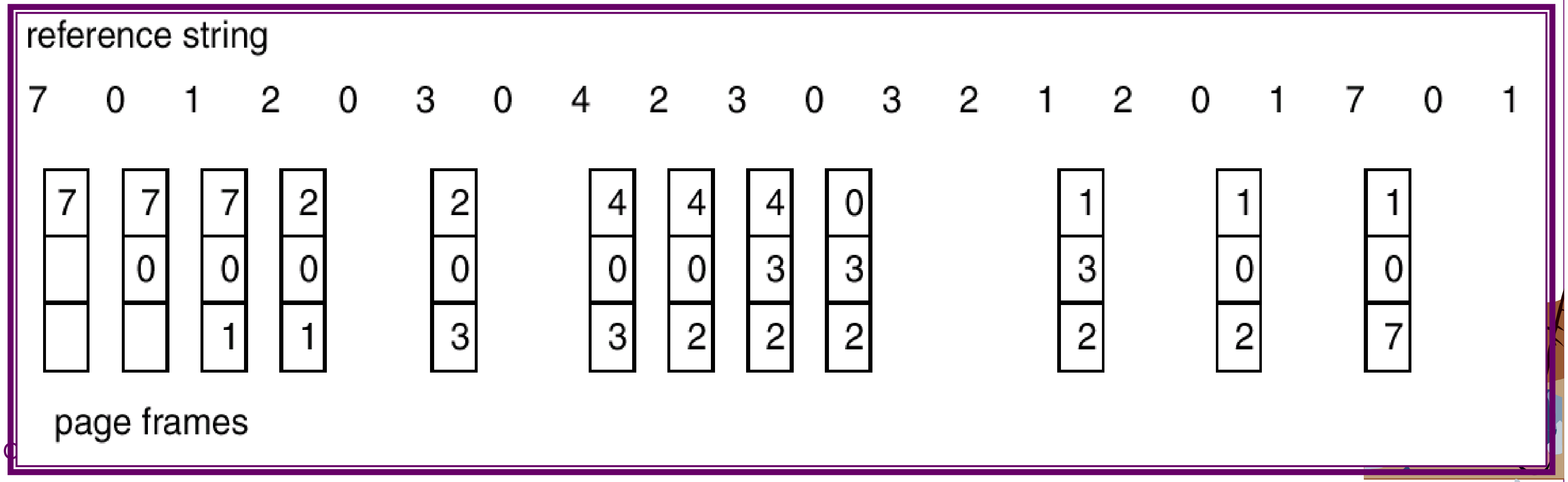


8 page faults,

better than FIFO (10 faults) and

worse than the optimal (6 faults)

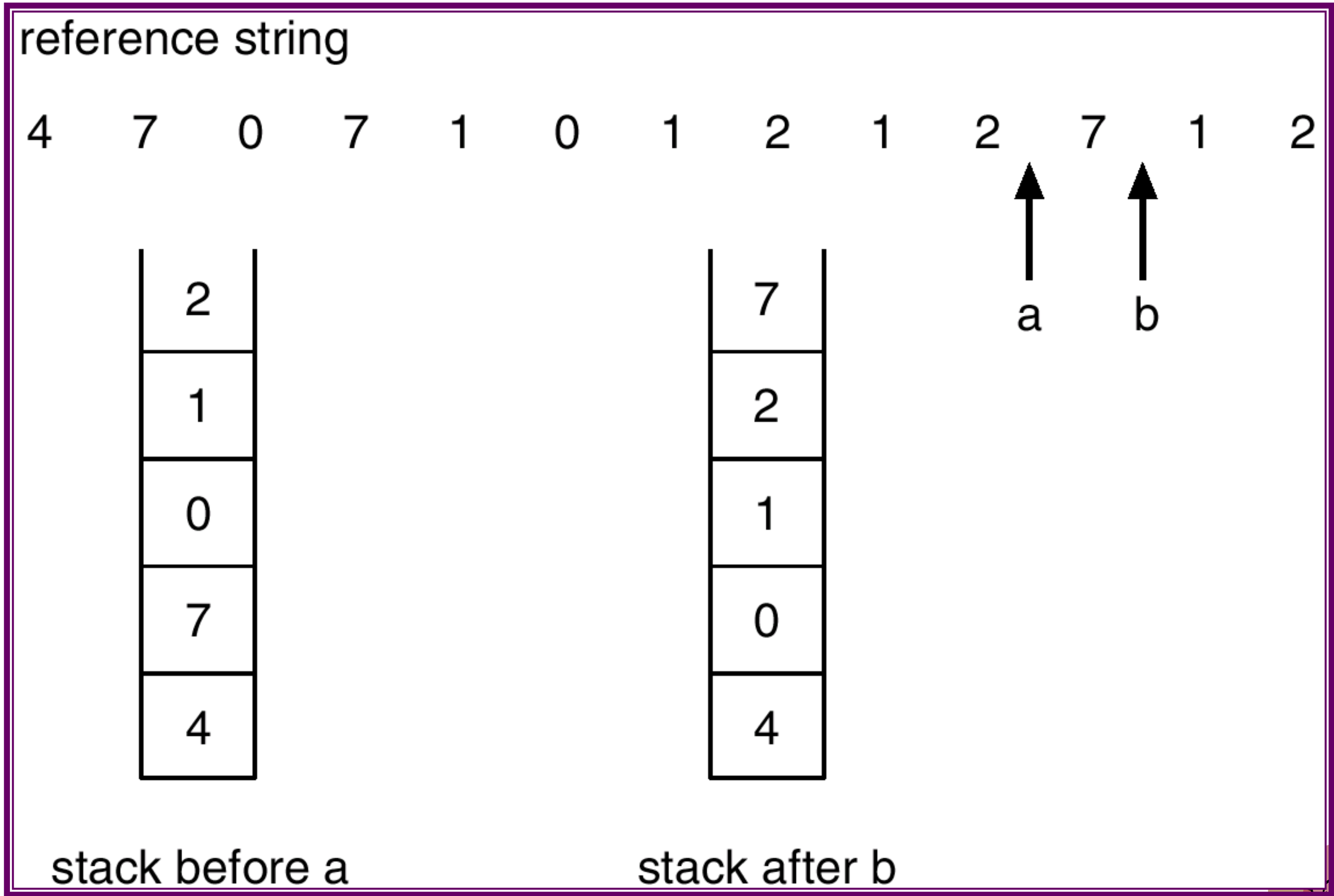
- Another Example



LRU Algorithm Implementations

- 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 doubly linked list:
  - When a page is referenced:
    - ✓move it to the top
    - ✓requires 6 pointers to be changed
  - No search for page replacement

Use A Stack to Record The Most Recent Page References



Problems of Previous LRU Implementations

- As to the previous two LRU implementations,
  - Clock: Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter.
  - Stack: Whenever a page is referenced, it is removed from the stack and put on the top.
- The updating of the clock fields or stack must be done for every memory reference
- Would slow every memory access by a factor of at least ten

LRU Approximation Algorithms

- Reference bit per page (Hardware maintained)
  - Each page is associated with a bit in the page table
  - Initially 0; When page is referenced, set the bit to 1.
  - Replace the one which is 0 (if one exists)
- However, we do not know the order of use.
- This information is the basis for many page-replacement algorithms that approximate LRU replacement

LRU Approximation Algorithms

- Rational: Gain additional ordering information by recording the reference bits at regular intervals

- Additional-Reference-Bits Algorithm
  - Keep an 8-bit bytes for each page in main memory
  - 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 intergers, the page with lowest number is the LRU page

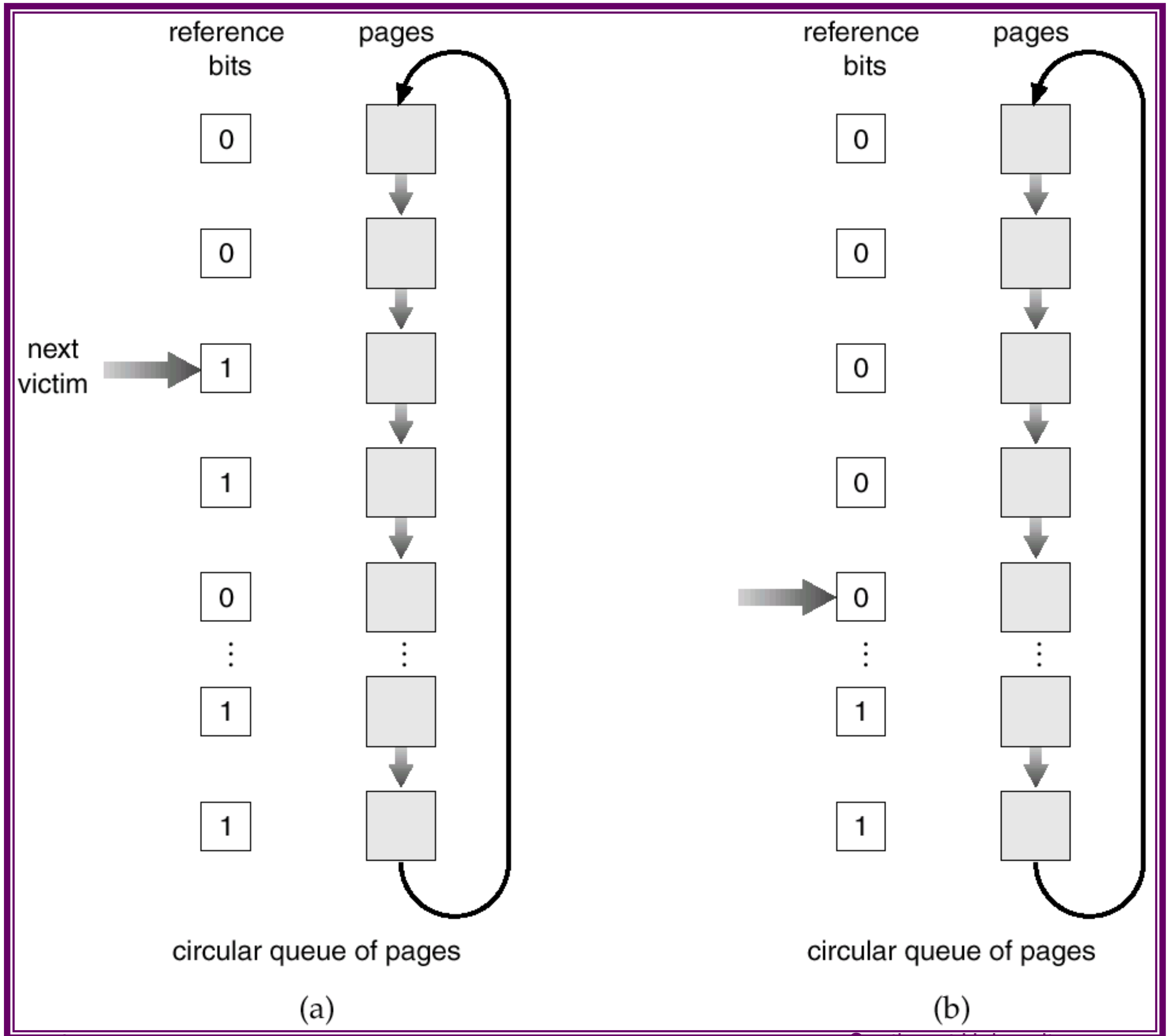
LRU Approximation Algorithms

- Second-Chance Algorithm (FIFO+reference bit)
  - When a page has been selected for replacement, we inspect its reference bit.
  - If the value is 0, we proceed to replace this page;
  - but if the reference bit is set to 1, we give the page a second chance and move on to select the next FIFO page.
  - When a page gets a second chance, its reference bit is cleared, and its arrival time is reset to the current time.

LRU Approximation Algorithms

- Second-Chance Algorithm (clock+reference bit)
  - Given a circular queue, called clock
  - 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



Counting Algorithms

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

Chapter 9: Virtual Memory

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing and Working Set Model
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples

Allocation of Frames

- Each process needs a **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 – e.g., if 100 frames and 5 processes, give each 20 pages.
- Proportional allocation – Allocate according to the size of process.

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

$$S = \sum s_i$$

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

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

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.

Chapter 9: Virtual Memory

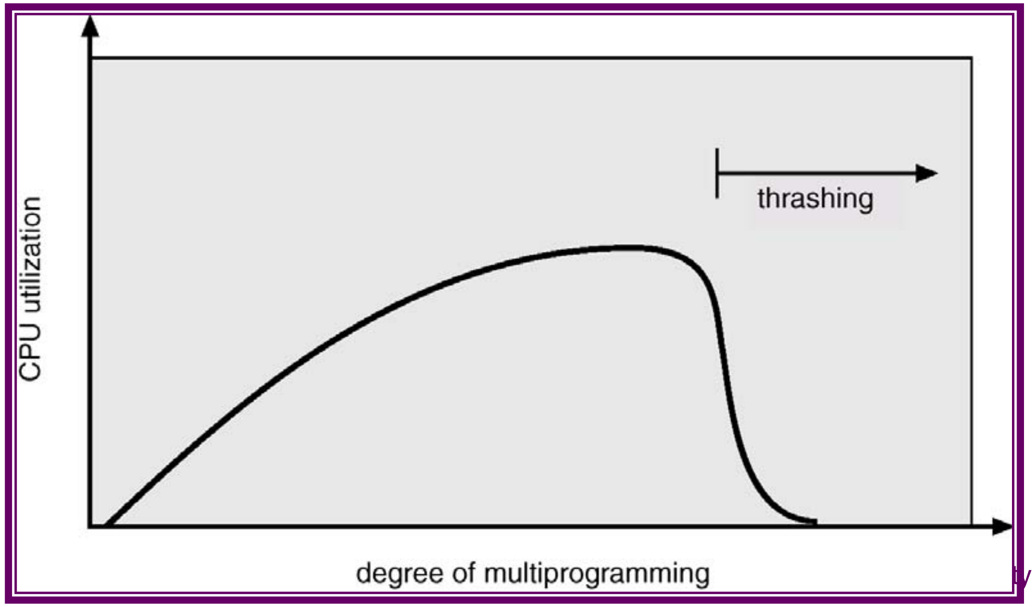
- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing and Working Set Model
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples

Thrashing

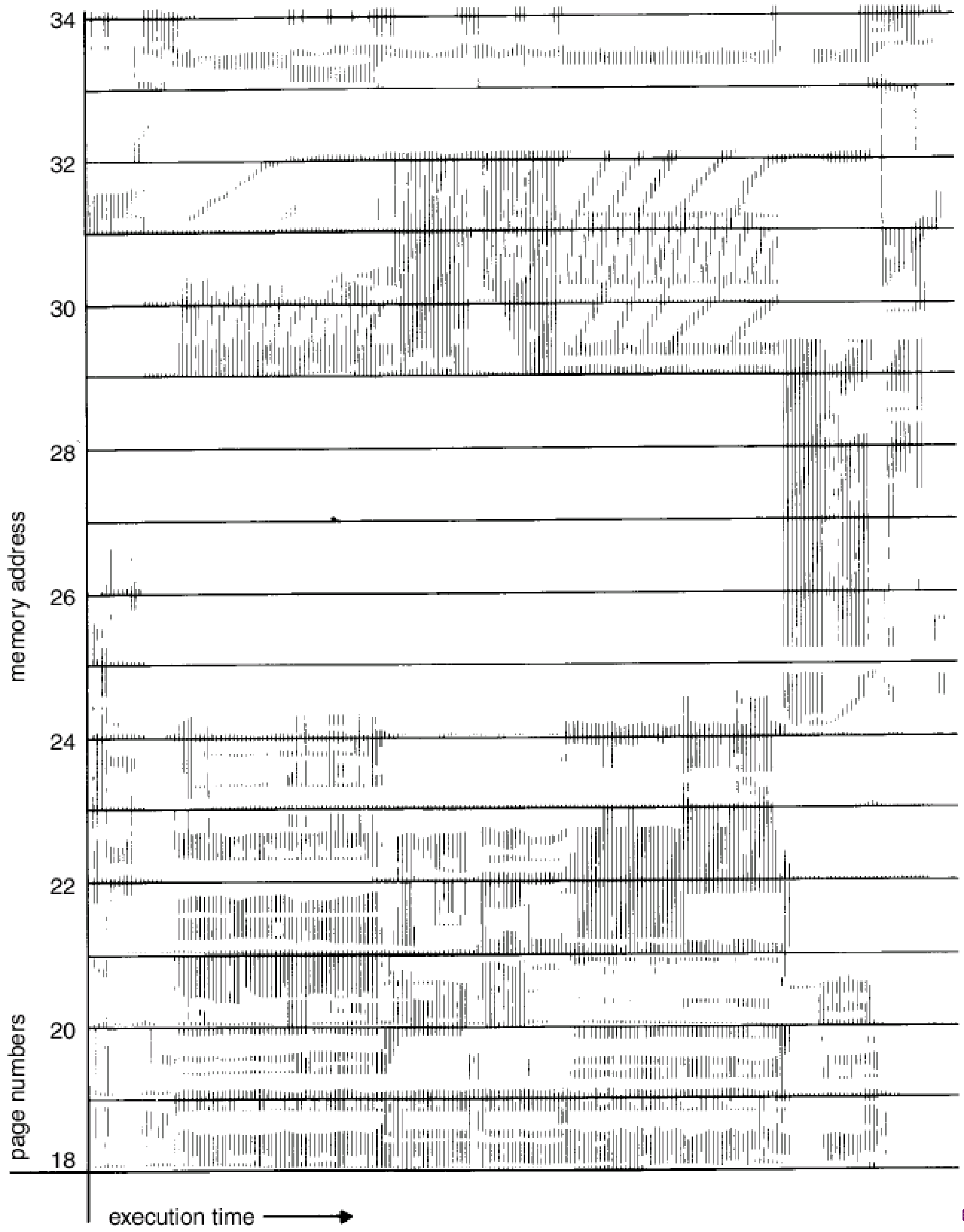
- If a process does not have “enough” frames, the page-fault rate is very high. This leads to:
  - low CPU utilization.
  - operating system thinks that it needs to increase the degree of multiprogramming.
  - another process added to the system.
- **Thrashing** ≡ a process is busy swapping pages in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

Thrashing

- Why does 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 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.

Working-Set Model (cont.)

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 . . .

$WS(t_1) = \{1,2,5,6,7\}$ $WS(t_2) = \{3,4\}$

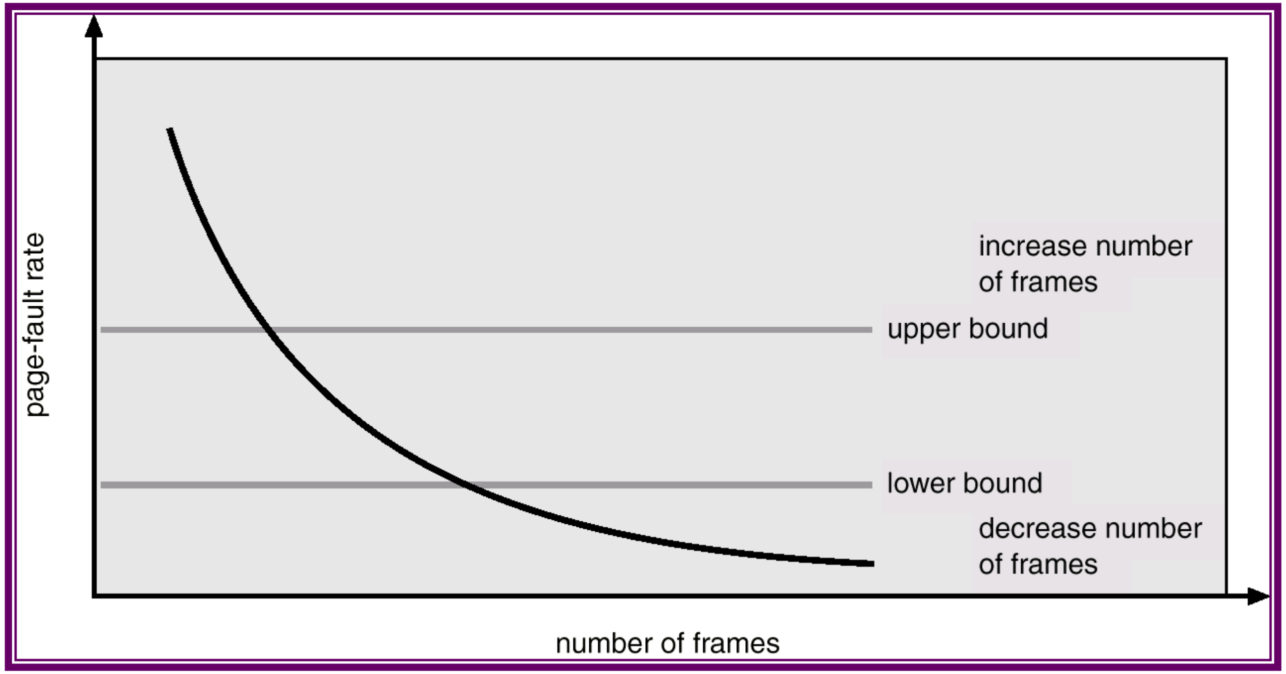
Working-Set Model (cont.)

- $D = \Sigma\ WSS_i \equiv$ total demand frames
- if $D > m \Rightarrow$ Thrashing
- Policy if $D > m$, then suspend one of the processes.

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.

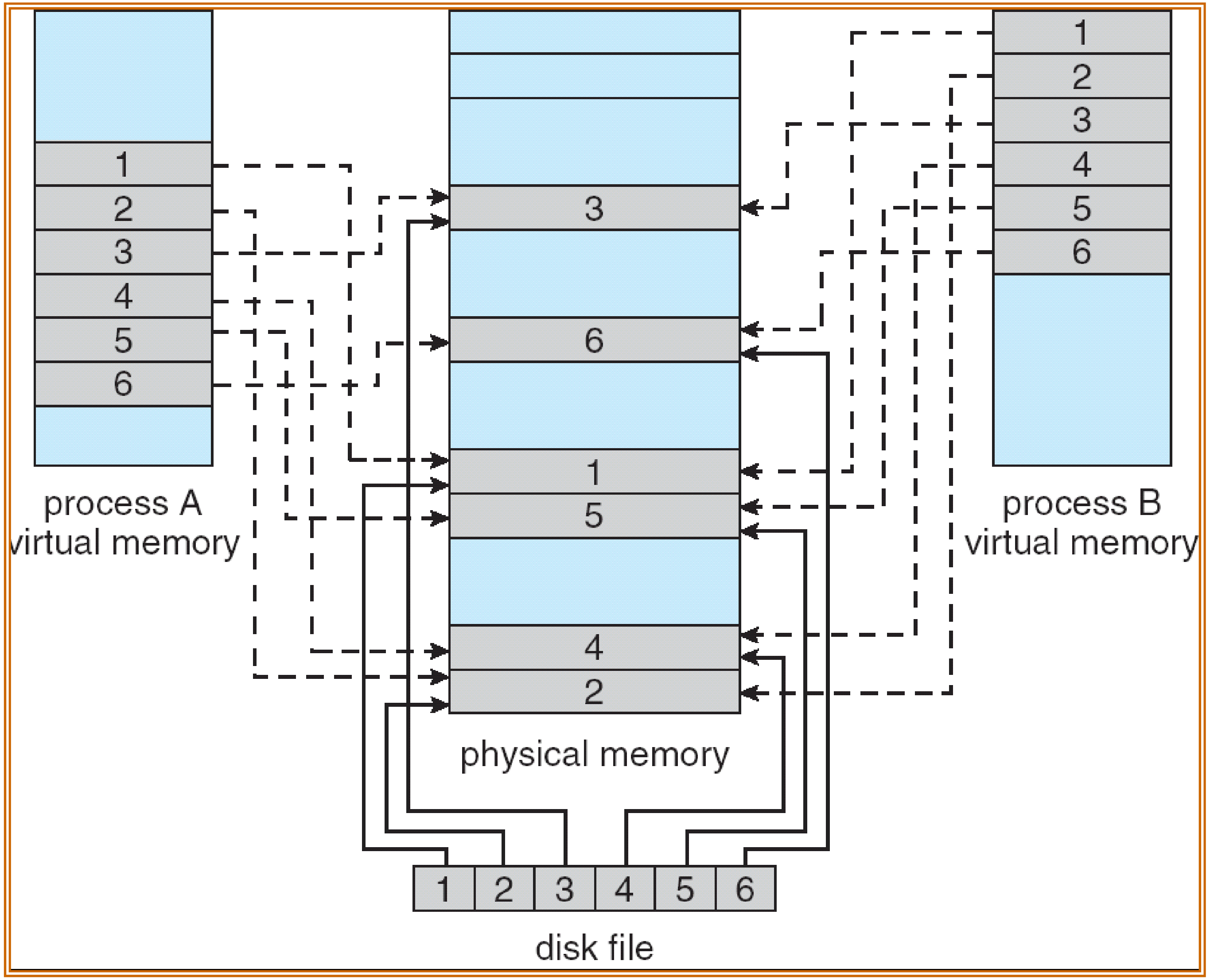
Chapter 9: Virtual Memory

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing and Working Set Model
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples

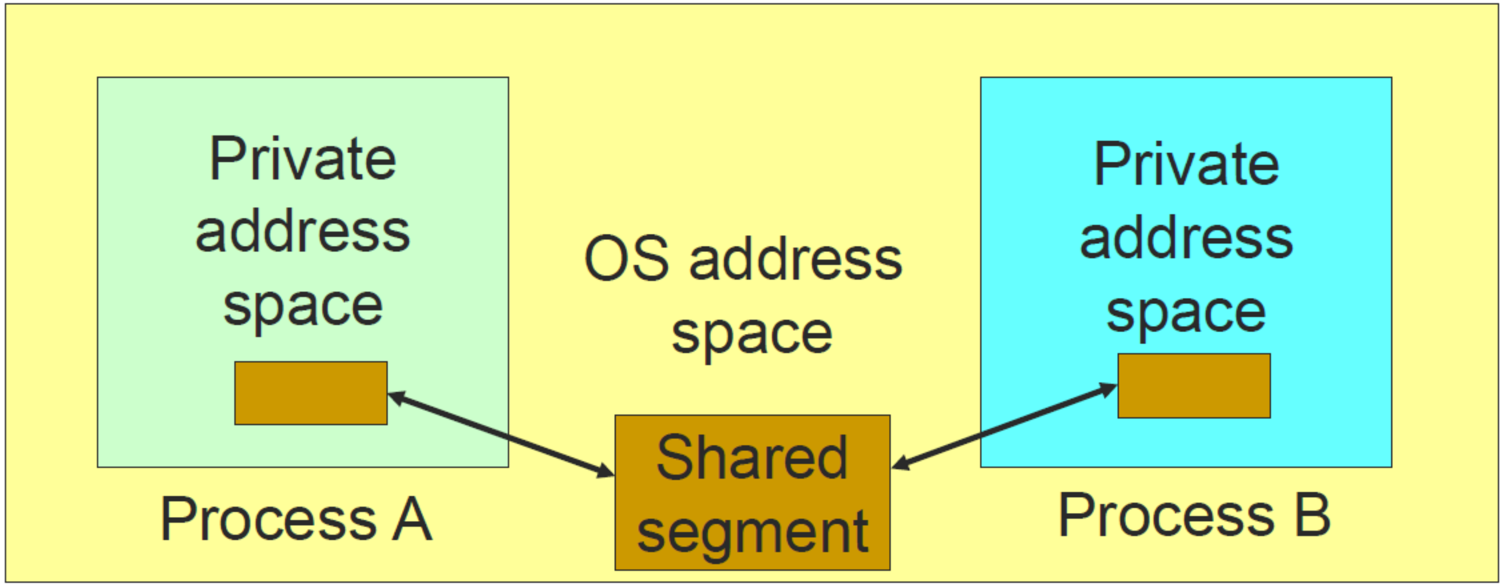
Memory-Mapped Files

- 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



- Processes request the shared segment
- OS maintains the shared segment
- Processes can attach/detach the segment
- Can mark segment for deletion on last detach

Chapter 9: Virtual Memory

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing and Working Set Model
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples

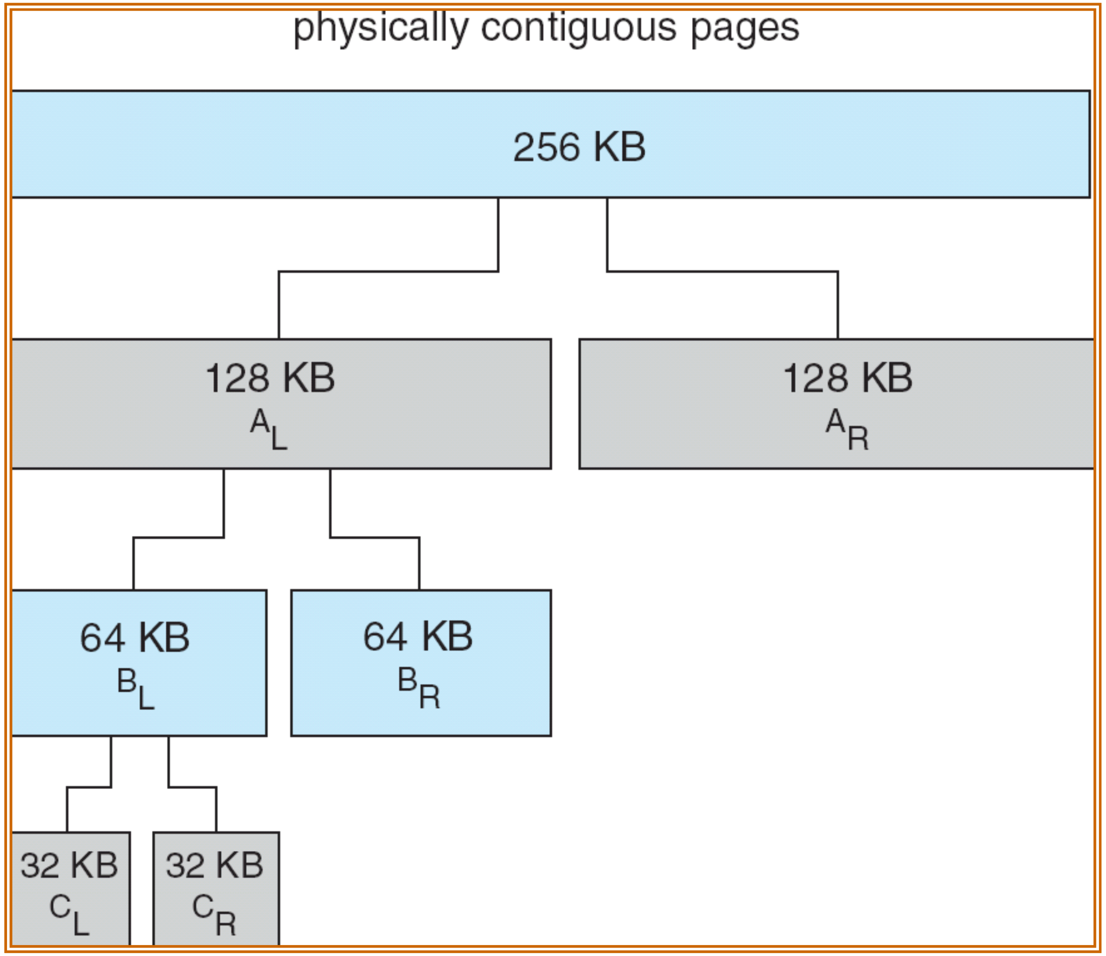
Allocating Kernel Memory

- 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



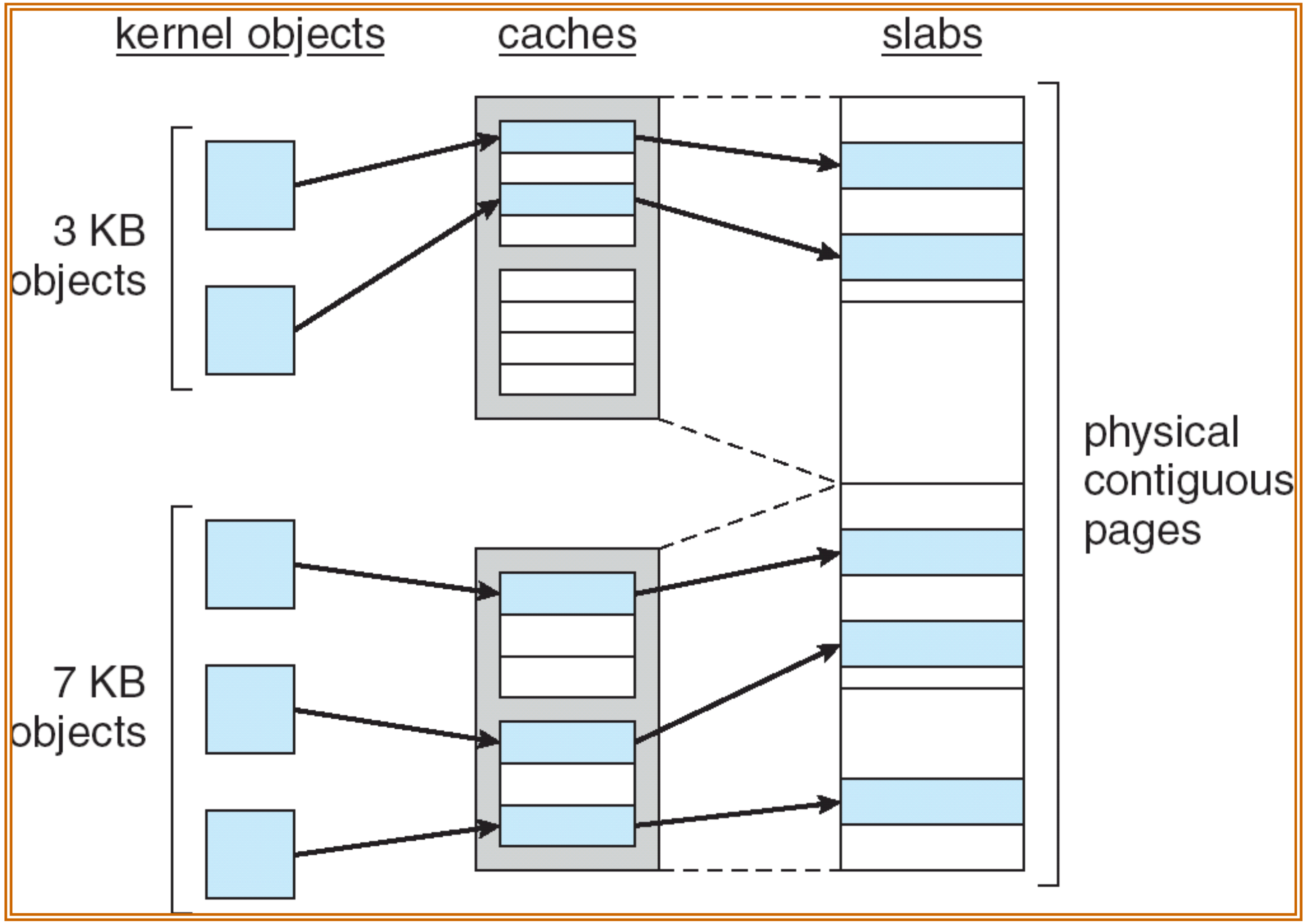
Slab Allocator

- 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

Illustrate the Slab Allocation



Chapter 9: Virtual Memory

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing and Working Set Model
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples

Other Issues -- Prepaging

- Prepaging (预调页)
  - To reduce the large number of page faults that occur 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 benefit of $s * \alpha$ save pages faults larger or smaller than the cost of prepaging $s * (1- \alpha)$ unnecessary pages?
    - ✓ α 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;
```

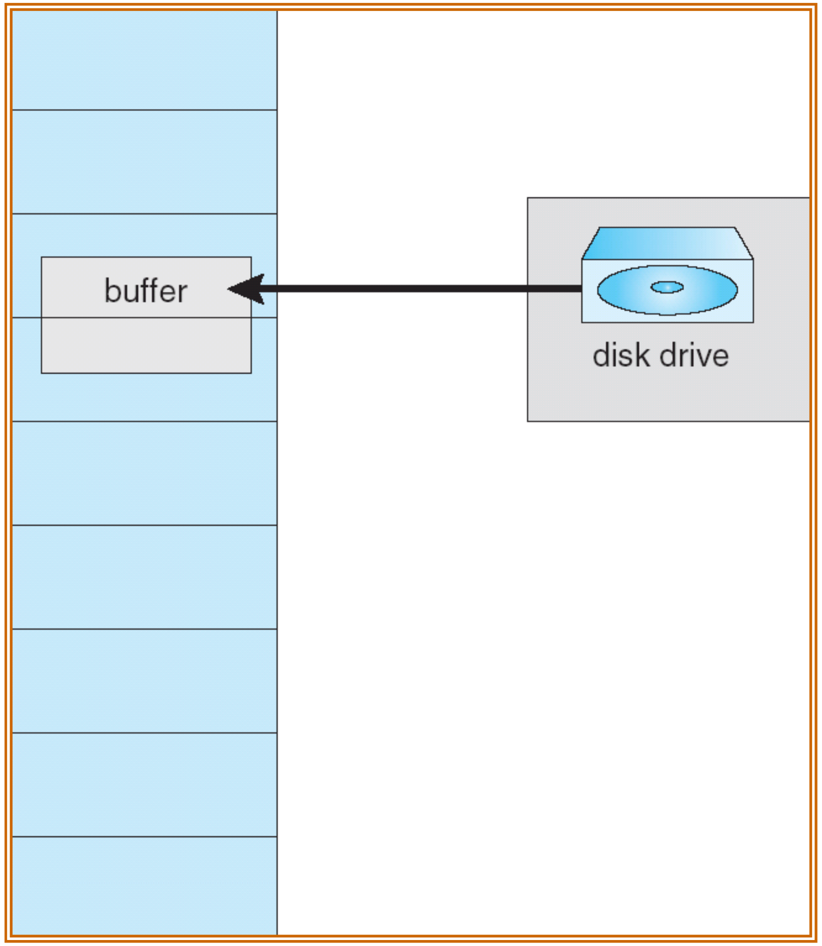
  - 1024 page faults

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 Kept in Memory



Chapter 9: Virtual Memory

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing and Working Set Model
- 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 (Cont.)

- 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 (Cont.)

- 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