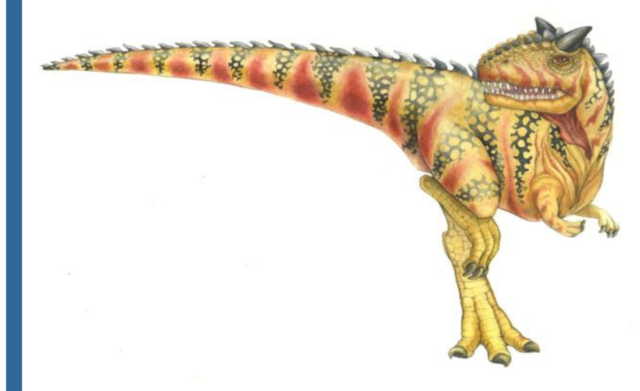
# Chapter 9
# Virtual-Memory Management



# Chapter 9: Virtual-Memory Management

- Background
- Demand Paging
- Copy-on-Write
- 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

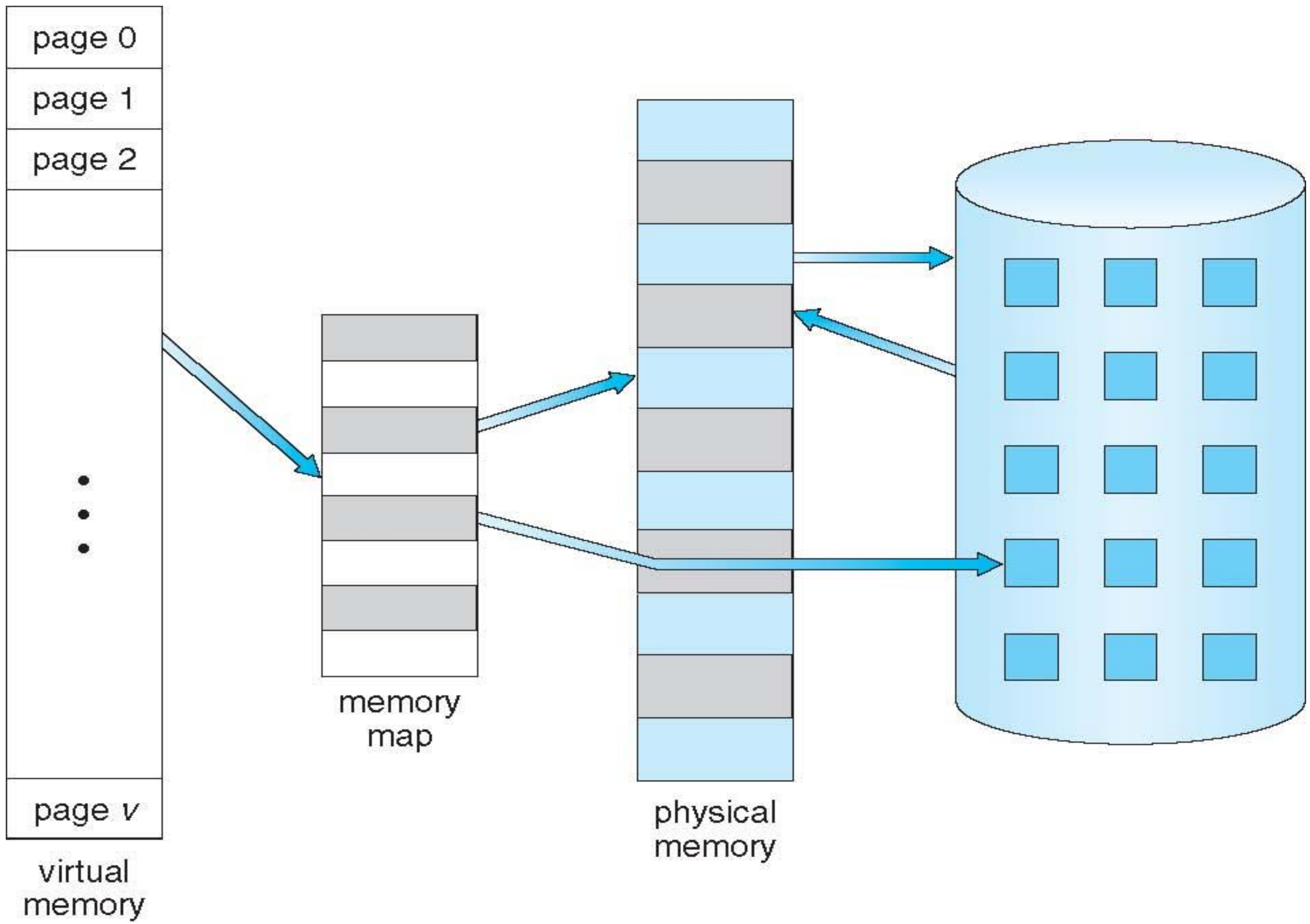
# Background

- Code needs to be in memory to execute, but entire program rarely used
  - Error code, unusual routines, large data structures
- Entire program code not needed at same time
- Consider ability to execute partially-loaded program
  - Program no longer constrained by limits of physical memory
  - Program and programs could be larger than physical memory

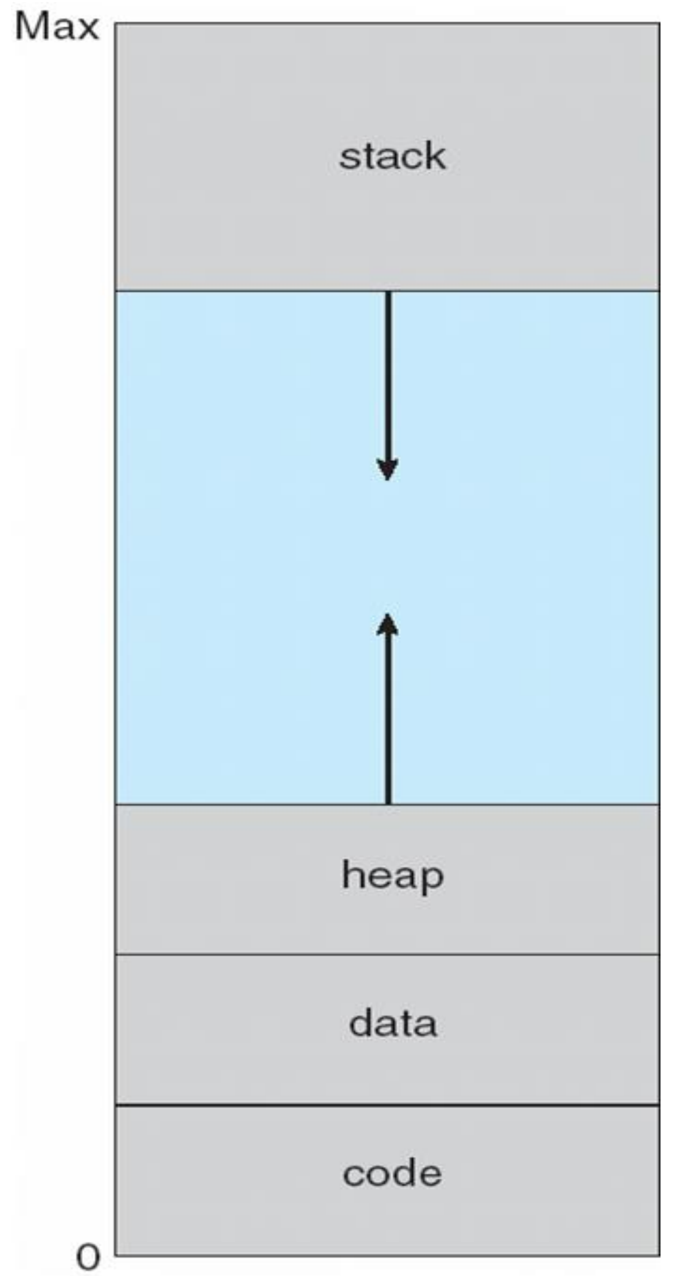
# Background

- **Virtual memory** – separation of user logical memory from physical memory
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation
  - More programs running concurrently
  - Less I/O needed to load or swap processes
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

# Virtual Memory That is Larger Than Physical Memory



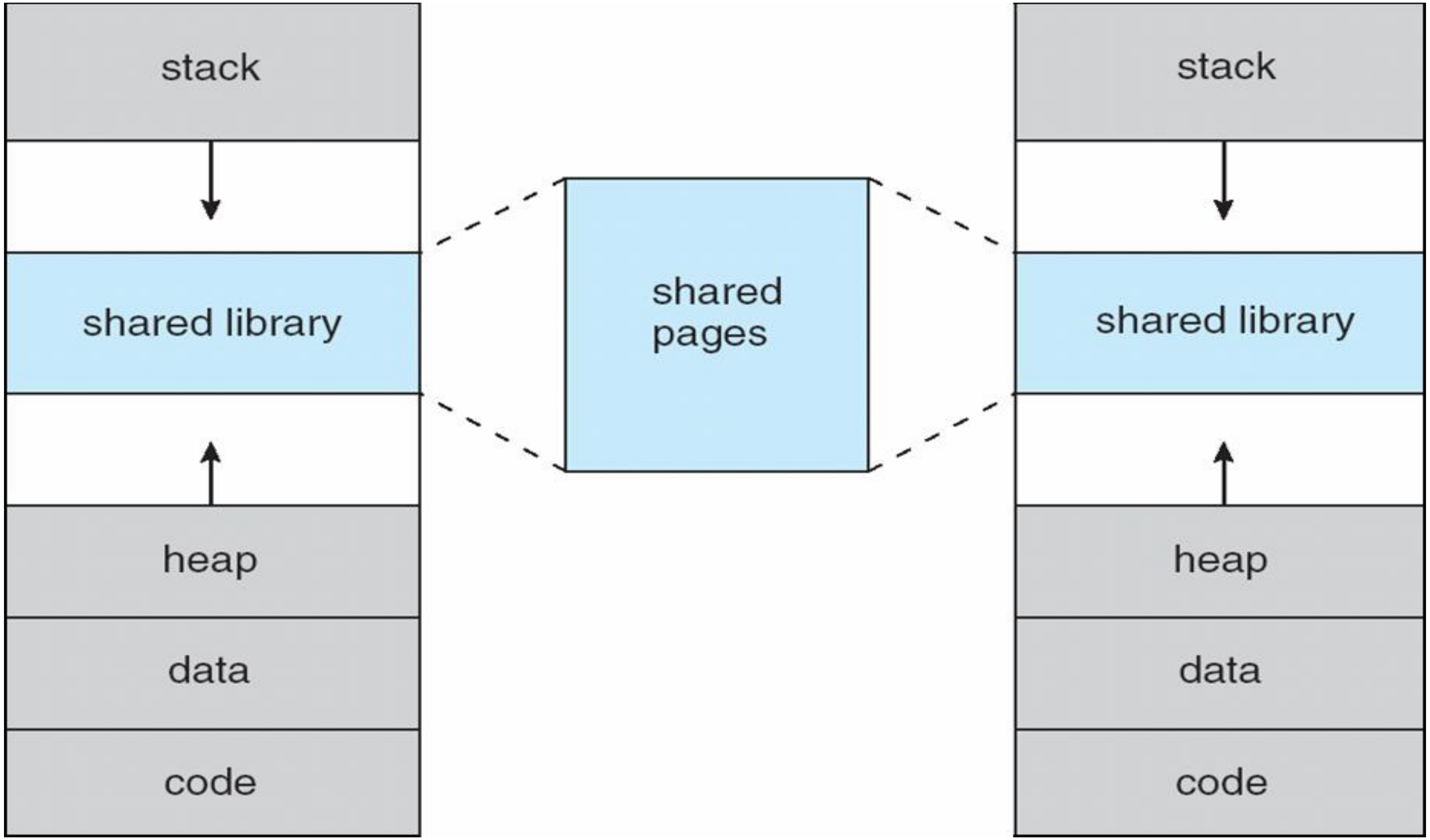
# Virtual-address Space



# Virtual Address Space

- Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc
- System libraries shared via mapping into virtual address space
- Shared memory by mapping pages read-write into virtual address space
- Pages can be shared during `fork()`, speeding process creation

# Shared Library Using Virtual Memory



# A Procedure to Handle A Page Fault

- Pure Demand Paging:
  - Never bring in a page into the memory until it is required!

- Pre-Paging
  - Bring into the memory all of the pages that “will” be needed at one time!
  - Locality of reference

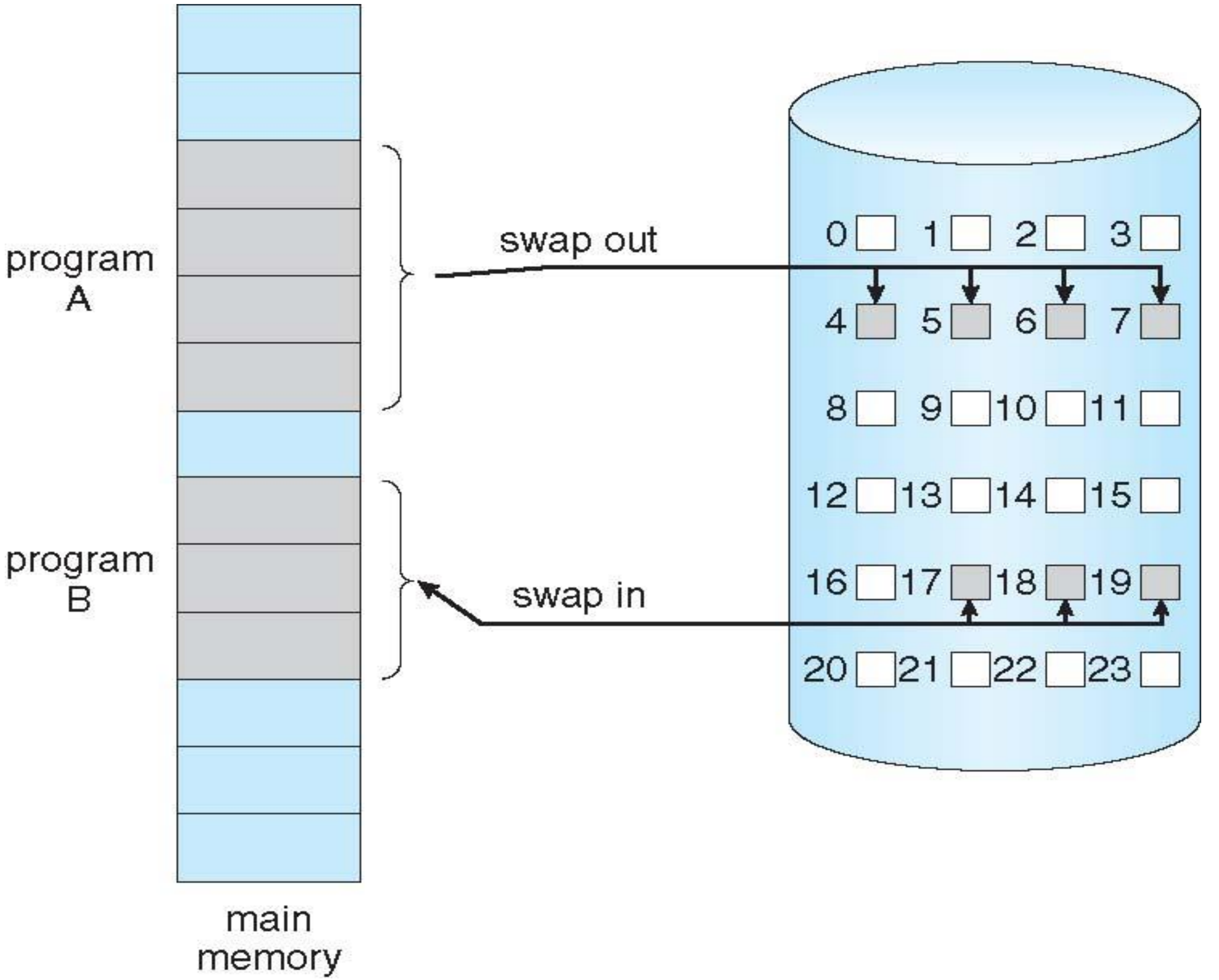
# Demand Paging

- Could bring entire process into memory at load time
- Or bring a page into memory only when it is needed
  - Less I/O needed, no unnecessary I/O
  - 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

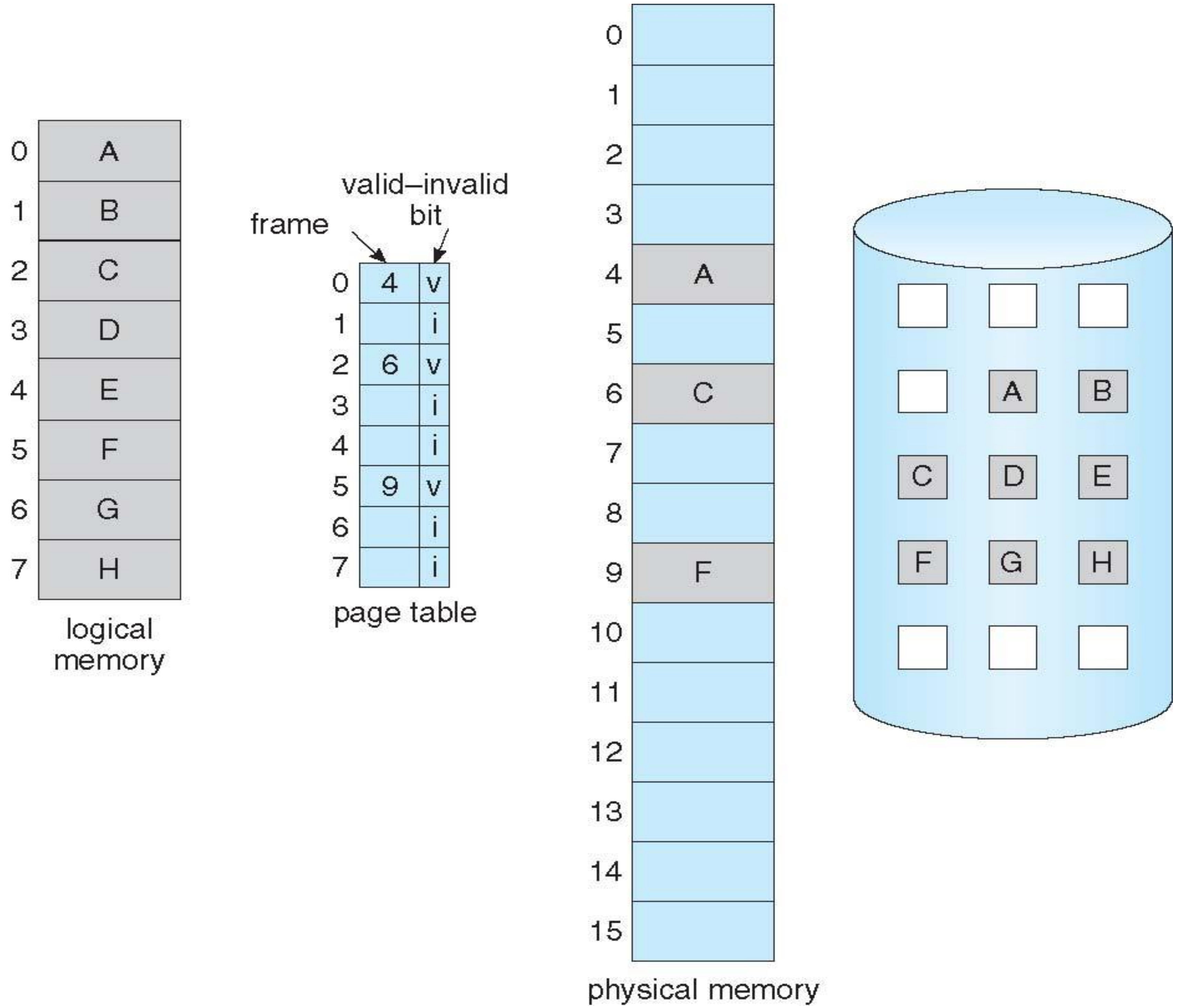
- With each page table entry a valid–invalid bit is associated
(**v** ⇒ in-memory – **memory resident**, **i** ⇒ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:

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



# Page Fault

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 ⇒ abort
   - Just not in memory
2. Get empty frame
3. Swap page into frame via scheduled disk operation
4. Reset tables to indicate page now in memory
5. Set validation bit = **v**
6. Restart the instruction that caused the page fault

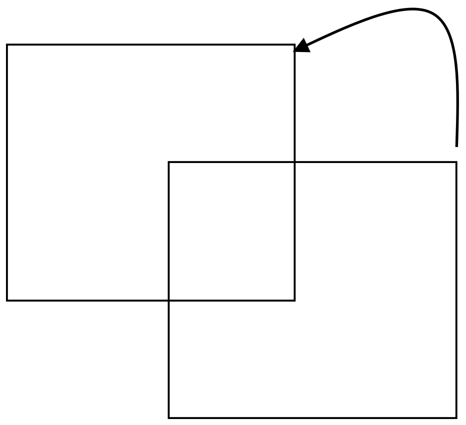
# Aspects of Demand Paging

- Extreme case – start process with *no* pages in memory
  - OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault
  - And for every other process pages on first access
  - **Pure demand paging**
- Actually, a given instruction could access multiple pages -> multiple page faults
  - Pain decreased because of **locality of reference**
- Hardware support needed for demand paging
  - Page table with valid / invalid bit
  - Secondary memory (swap device with **swap space**)
  - Instruction restart

# Instruction Restart

Consider an instruction that could access several different locations

- block move



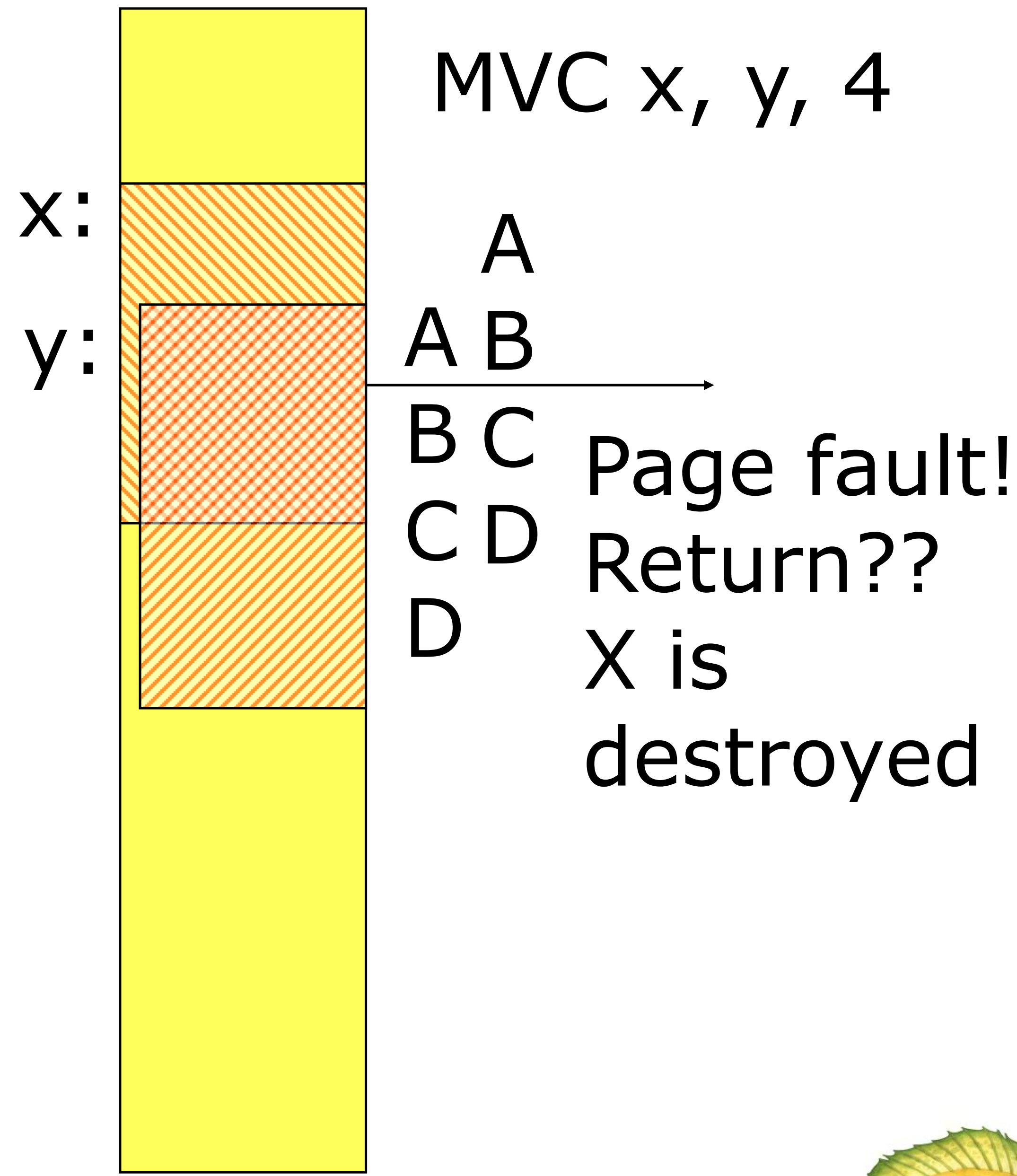
- auto increment/decrement location
- Restart the whole operation?
  - What if source and destination overlap?

# Crucial issues

- Example 1 – *Cost in restarting an instruction*
  - Assembly Instruction: Add a, b, c
  - Only a short job!
    - Re-fetch the instruction, decode, fetch operands, execute, save, etc
  - Strategy:
    - Get all pages and restart the instruction from the beginning!

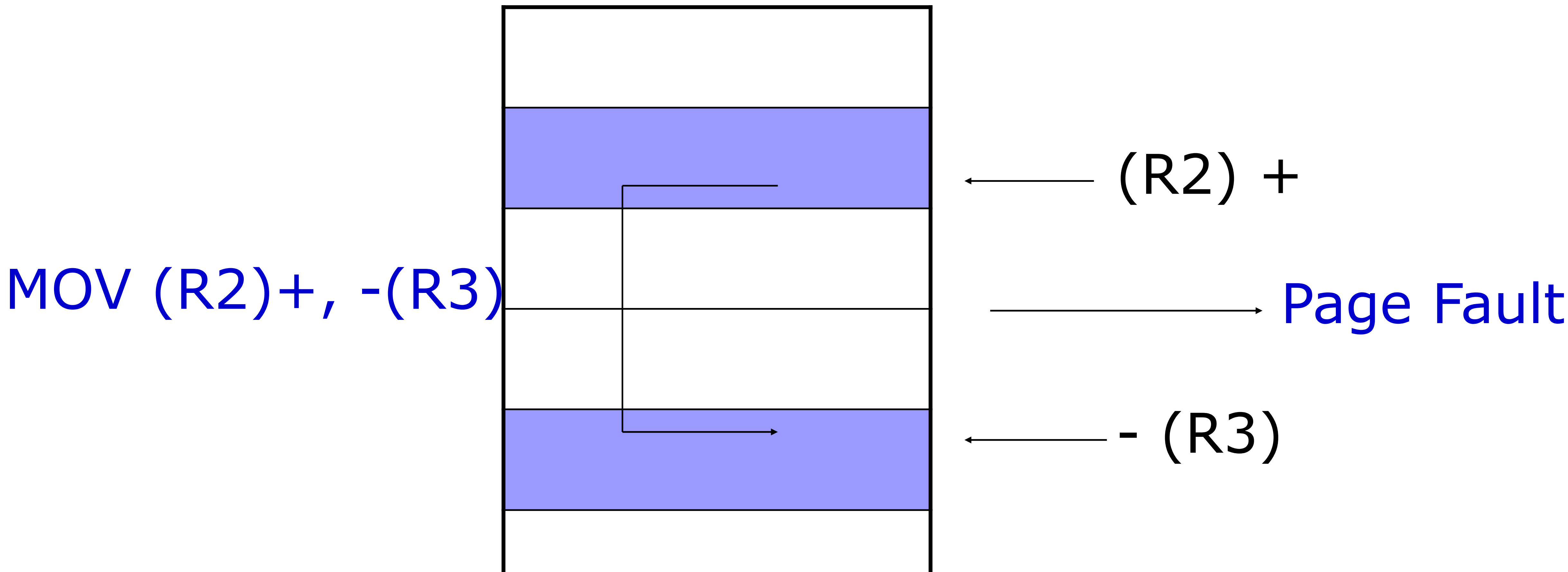
# Crucial Issues

- Example 2 – Block-Moving Assembly Instruction
  - MVC x, y, 256
    - IBM System 360/ 370
  - Characteristics
    - More expensive
    - “self-modifying” “operands”
  - Solutions:
    - Pre-load pages
    - Pre-save & recover before page-fault services



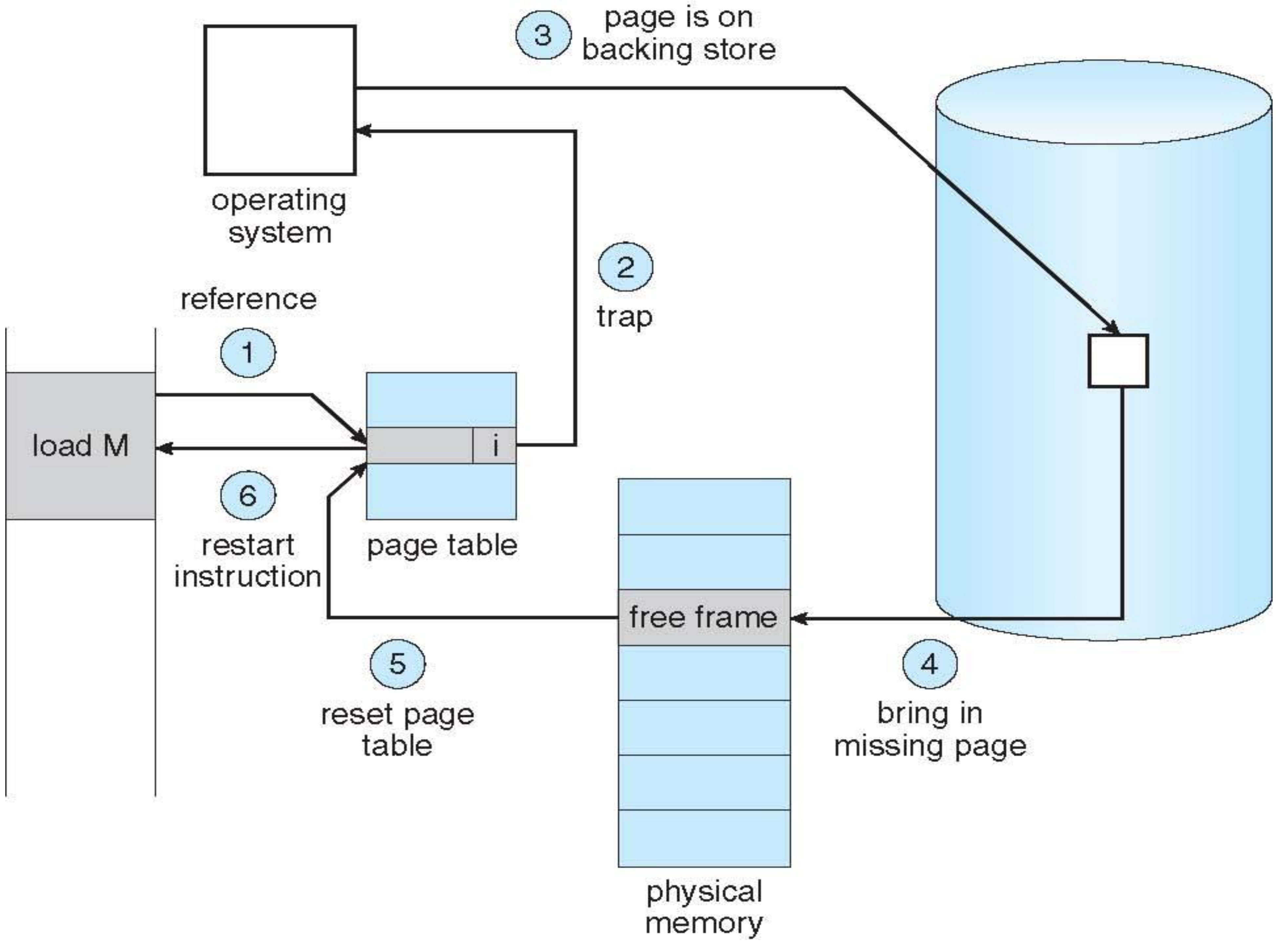
# Crucial Issues

- Example 3 – Addressing Mode




When the page fault is serviced,
R2, R3 are modified!
- Undo Effects!

# Steps in Handling a Page Fault



# Performance of Demand Paging

Stages in Demand Paging

1. Trap to the operating system
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:
    1. Wait in a queue for this device until the read request is serviced
    2. Wait for the device seek and/or latency time
    3. Begin the transfer of the page to a free frame
6. While waiting, allocate the CPU to some other user
7. Receive an interrupt from the disk I/O subsystem (I/O completed)
8. Save the registers and process state for the other user
9. Determine that the interrupt was from the disk
10. Correct the page table and other tables to show 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, and then resume the interrupted instruction

# Performance of Demand Paging (Cont.)

- Page Fault Rate $0 \le p \le 1$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault
- Effective Access Time (EAT)

$$
\begin{aligned}
EAT = (1 - p) \text{ x memory access} \\
+ p \text{ (page fault overhead} \\
+ \text{swap page out} \\
+ \text{swap page in} \\
+ \text{restart overhead} \\
)
\end{aligned}
$$

# Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds

- 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.

  This is a slowdown by a factor of 40!!
- If want performance degradation < 10 percent
  - 220 > 200 + 7,999,800 x p<br>20 > 7,999,800 x p
  - p < .0000025
  - < one page fault in every 400,000 memory accesses

# Demand Paging Optimizations

- How to keep the page fault rate low?
  - Effective Access Time ≈ 100ns + 24,999,900ns * p
- Handling of Swap Space – A Way to Reduce Page Fault Time (pft)
  - Disk I/O to swap space is generally faster than that to the file system.
    - Preload processes into the swap space before they start up.
    - Demand paging from file system but do page replacement to the swap space. (BSD UNIX)
- Demand page in from program binary on disk, but discard rather than paging out when freeing frame
  - Used in Solaris and current BSD

# Process Creation

- Virtual memory allows other benefits during process creation:

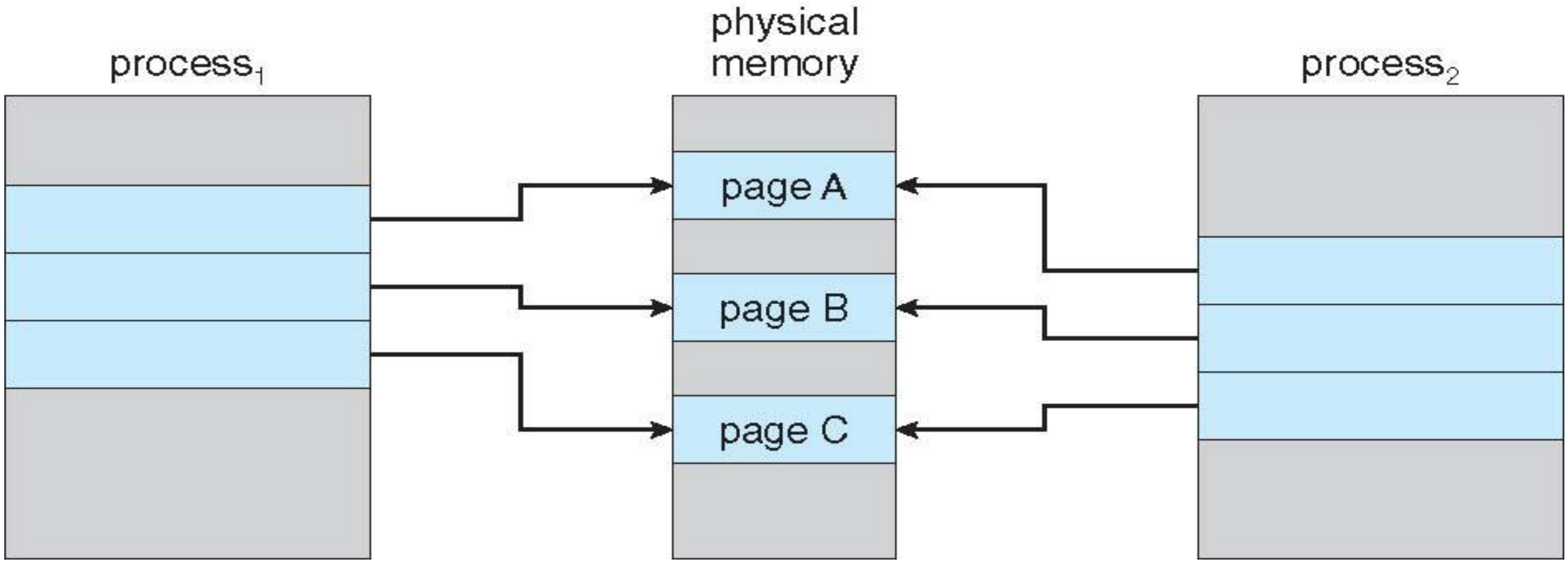
  - Copy-on-Write

  - Memory-Mapped Files (later)

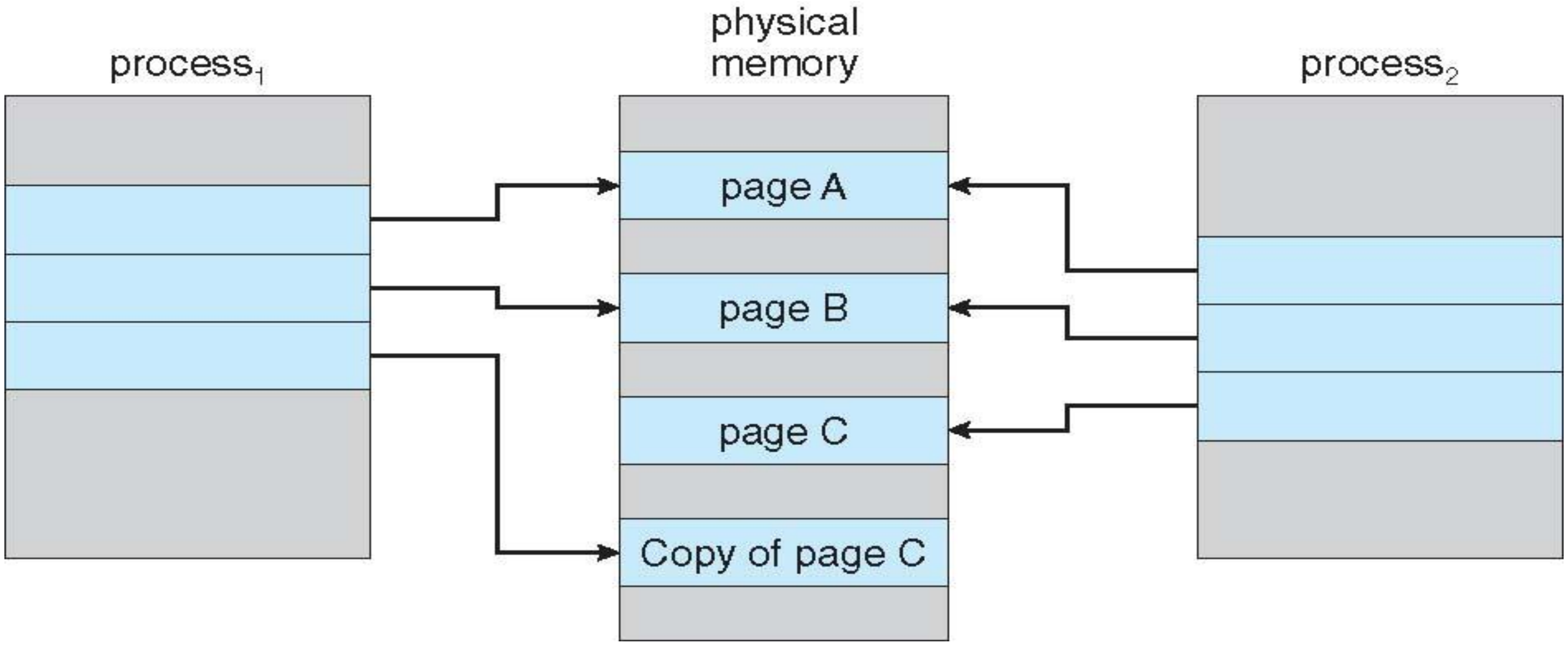
# Copy-on-Write

- **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
- COW allows more efficient process creation as only modified pages are copied
- In general, free pages are allocated from a **pool** of **zero-fill-on-demand** pages
  - Why zero-out a page before allocating it?
- `vfork()` variation on `fork()` system call has parent suspend and child using copy-on-write address space of parent
  - Designed to have child call `exec()`
  - Very efficient

# Before Process 1 Modifies Page C



# After Process 1 Modifies Page C



# What Happens if There is no Free Frame?

- Used up by process pages
- Also in demand from the kernel, I/O buffers, etc
- How much to allocate to each?

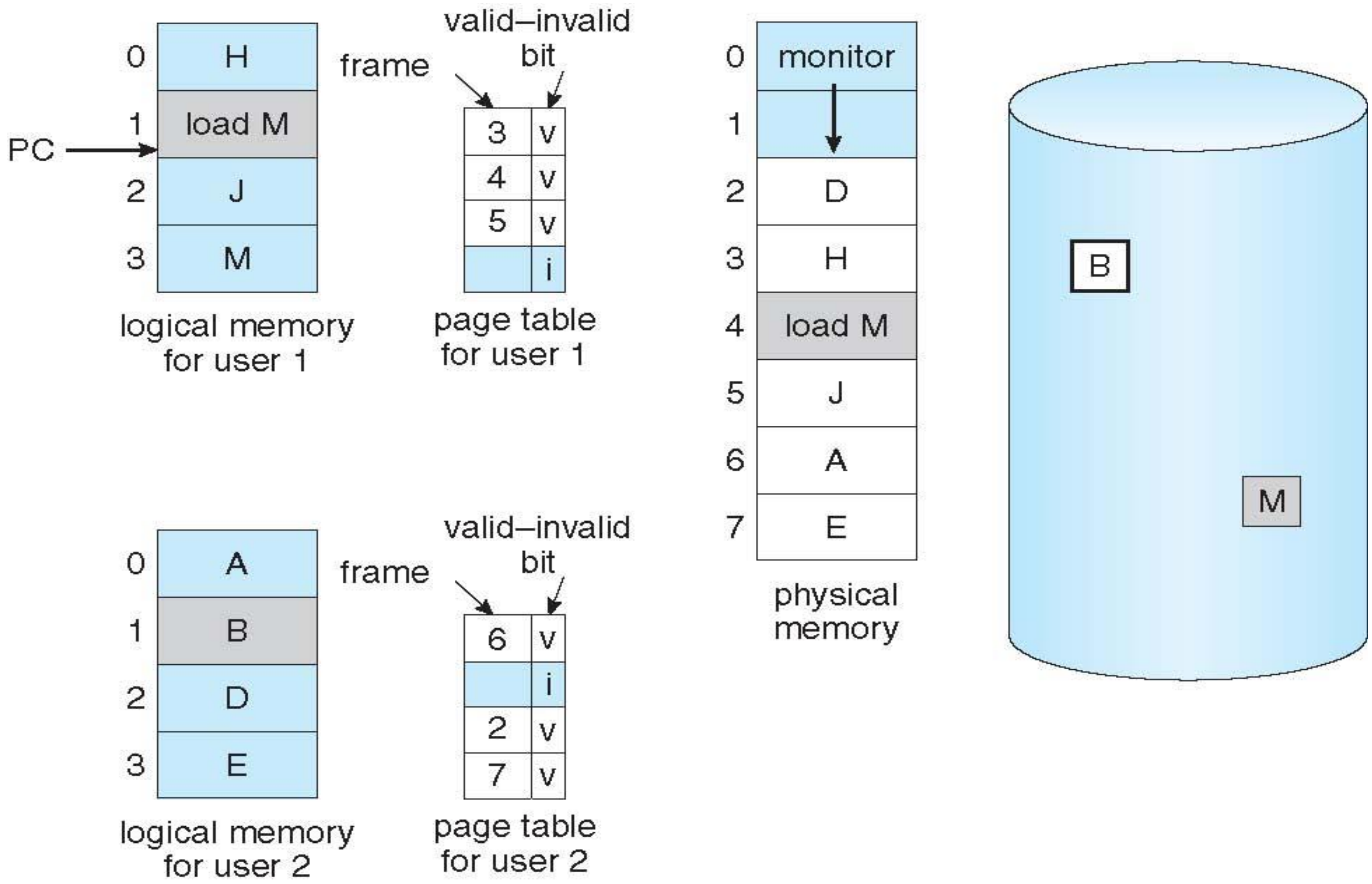
- Page replacement – find some page in memory, but not really in use, page it out
  - Algorithm – terminate? swap out? replace the page?
  - Performance – want an algorithm which will result in minimum number of page faults

- 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

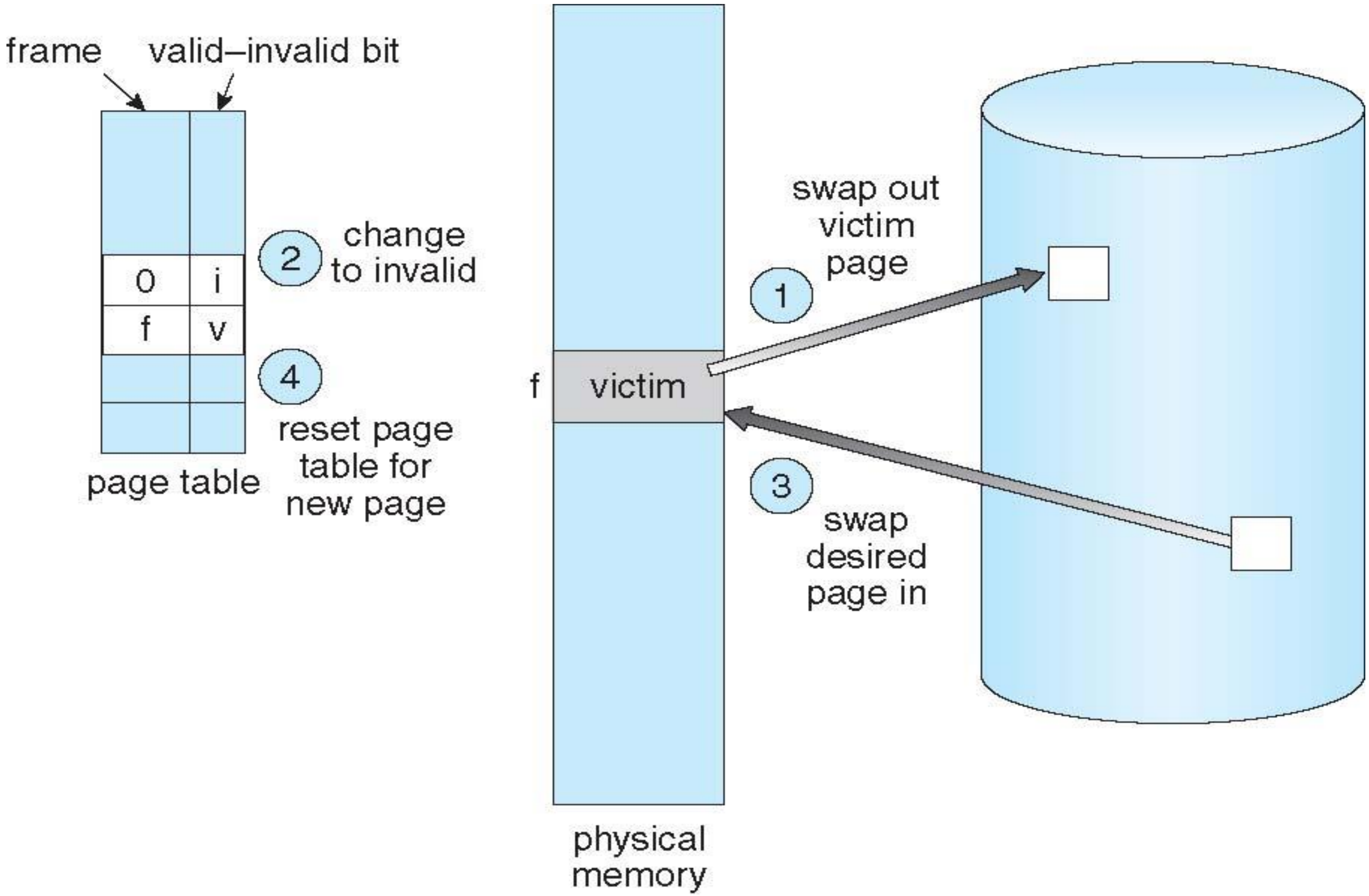
2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a **victim frame**
     - Write victim frame to disk if dirty

3. Bring the desired page into the (newly) free frame; update the page and frame tables

4. Continue the process by restarting the instruction that caused the trap

Note now potentially 2 page transfers for page fault – increasing EAT

# Page Replacement

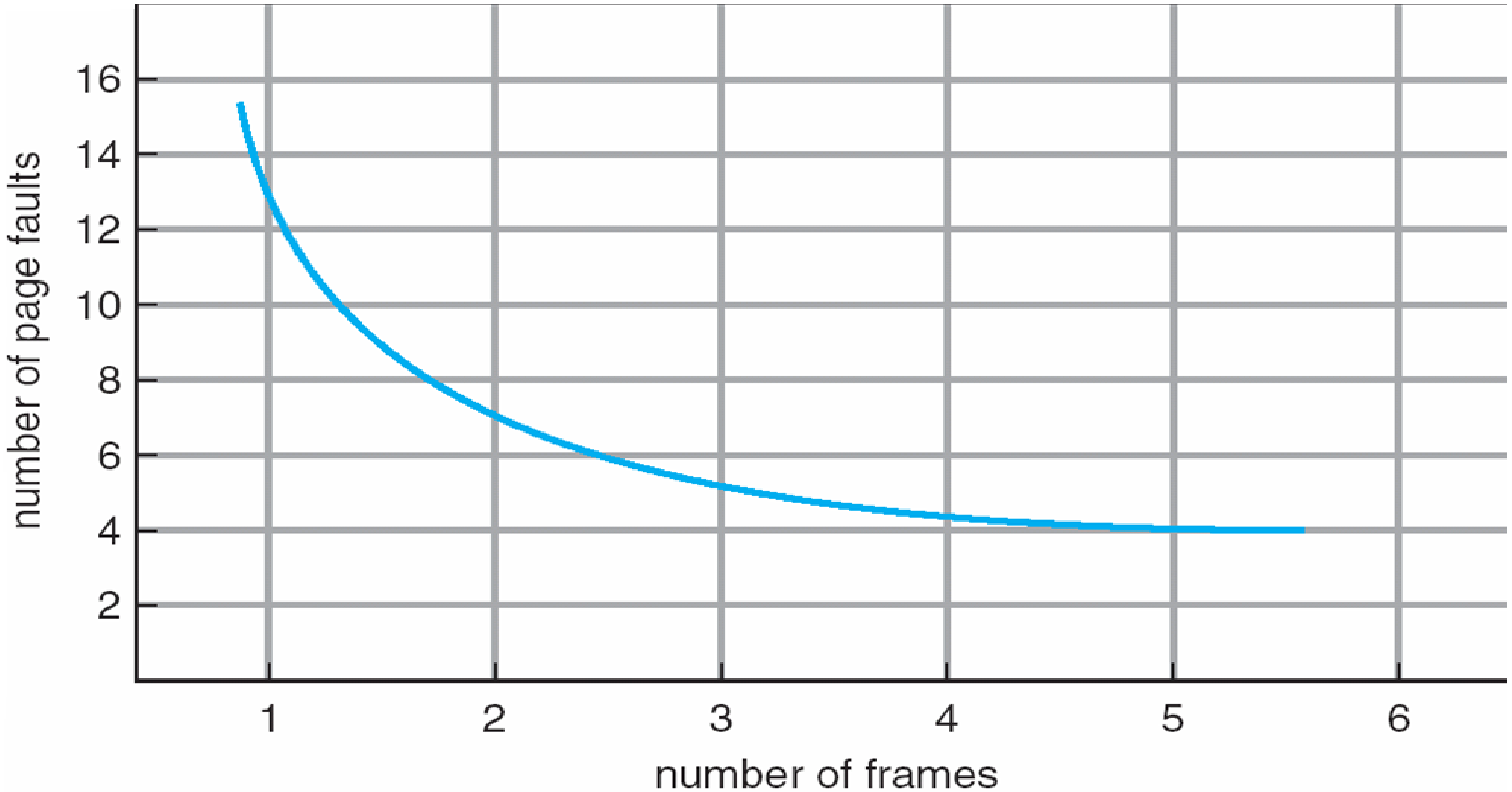


# Page and Frame Replacement Algorithms

- **Frame-allocation algorithm** determines
  - How many frames to give each process
  - Which frames to replace
- **Page-replacement algorithm**
  - Want lowest page-fault rate on both first access and re-access
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
  - String is just page numbers, not full addresses
  - Repeated access to the same page does not cause a page fault
- In all our examples, the reference string is

**7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**

# Graph of Page Faults Versus The Number of Frames



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

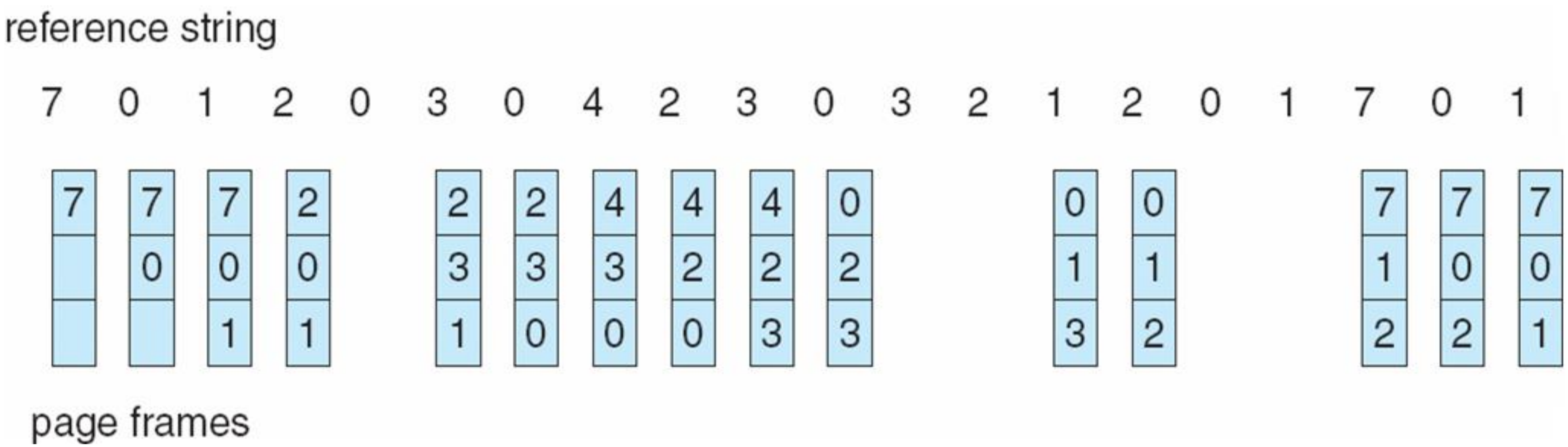
- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)

| | | | | | |
|---|---|---|---|---|---|
| 1 | 7 | 2 | 4 | 0 | 7 |
| 2 | 0 | 3 | 2 | 1 | 0 |
| 3 | 1 | 0 | 3 | 2 | 1 |

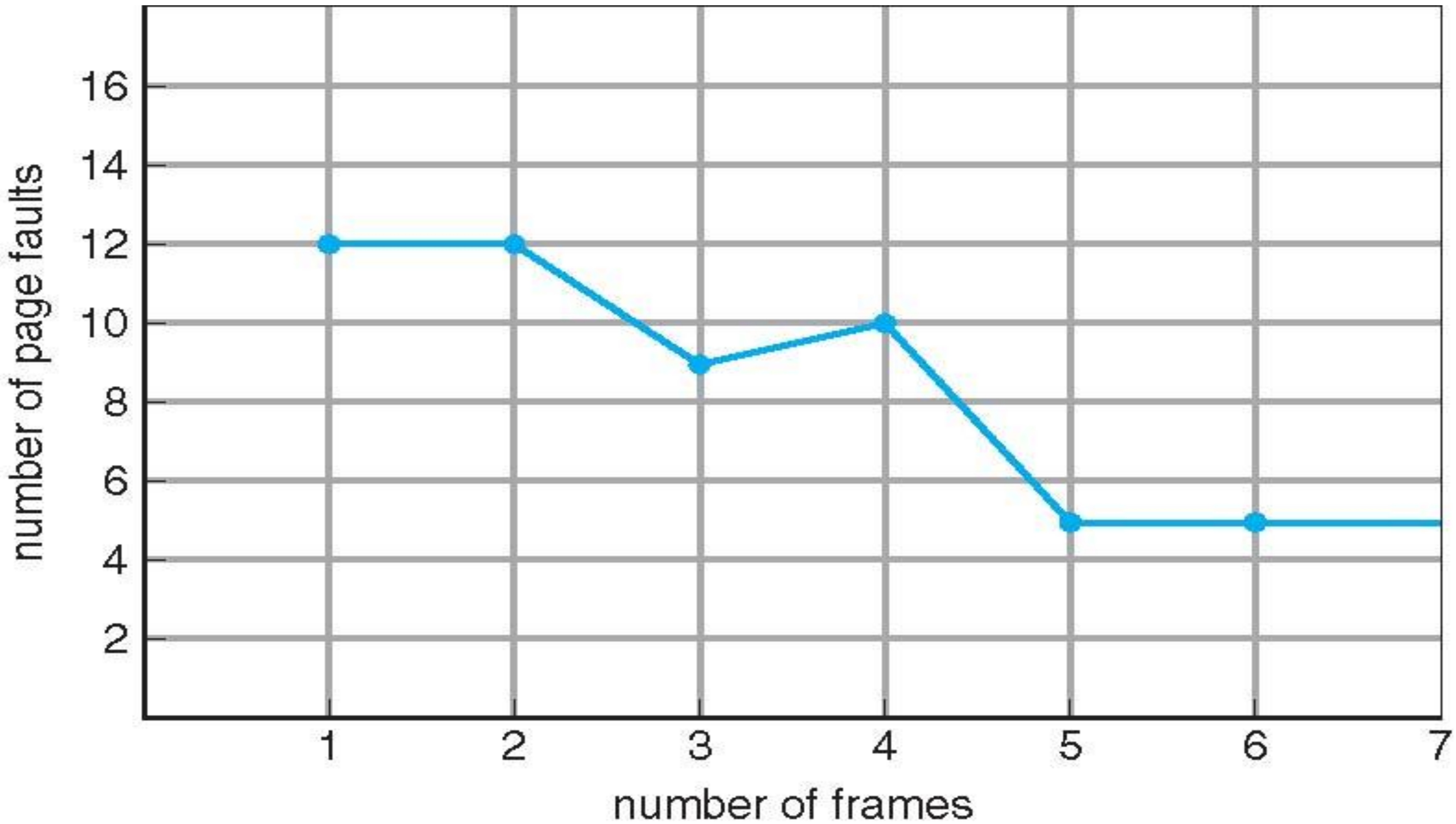
15 page faults

- Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5
  - Adding more frames can cause more page faults!
    - **Belady's Anomaly**
- How to track ages of pages?
  - Just use a FIFO queue

# FIFO Page Replacement



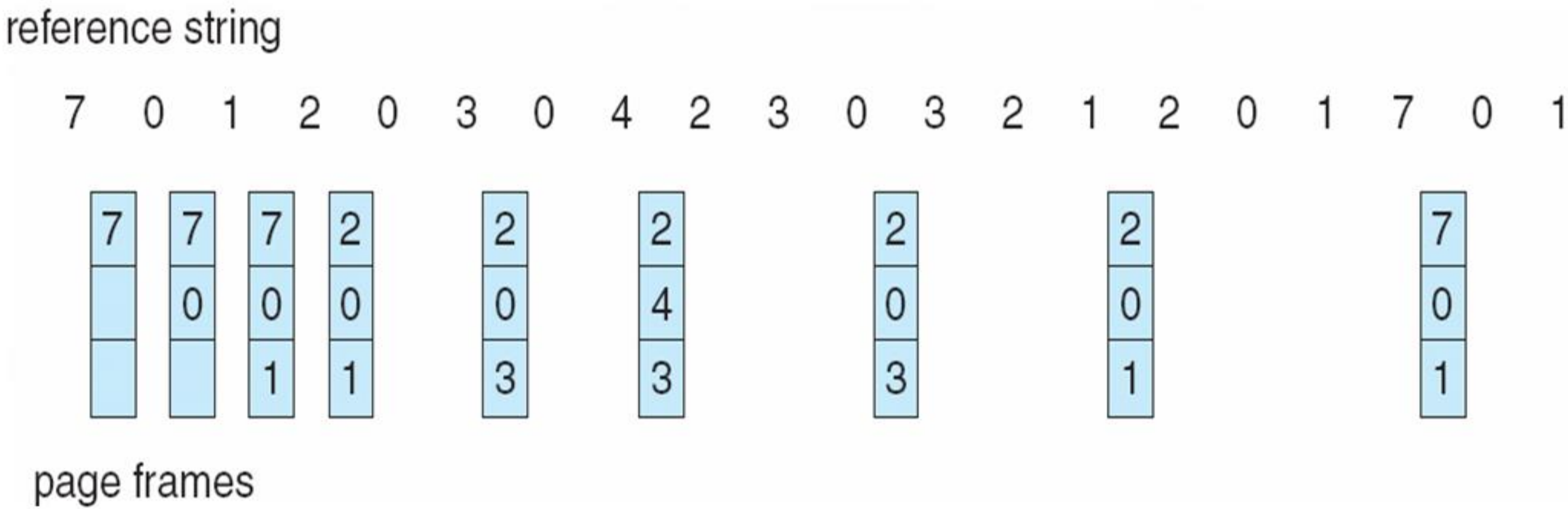
# FIFO Illustrating Belady's Anomaly



# Optimal Algorithm

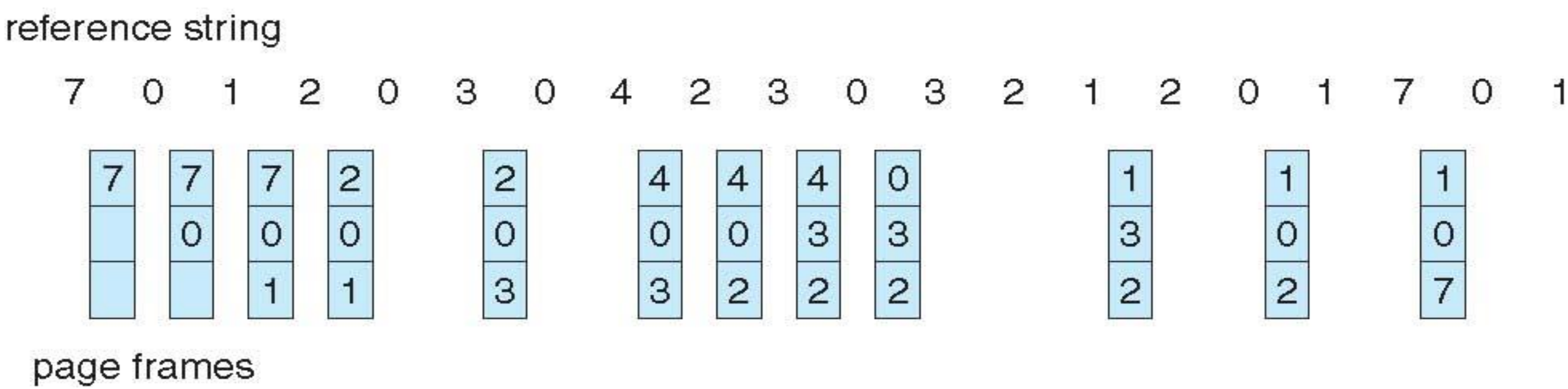
- Replace page that will not be used for longest period of time
  - 9 is optimal for the example on the next slide
- How do you know this?
  - Can’t read the future
- Used for measuring how well your algorithm performs

# Optimal Page Replacement



# Least Recently Used (LRU) Algorithm

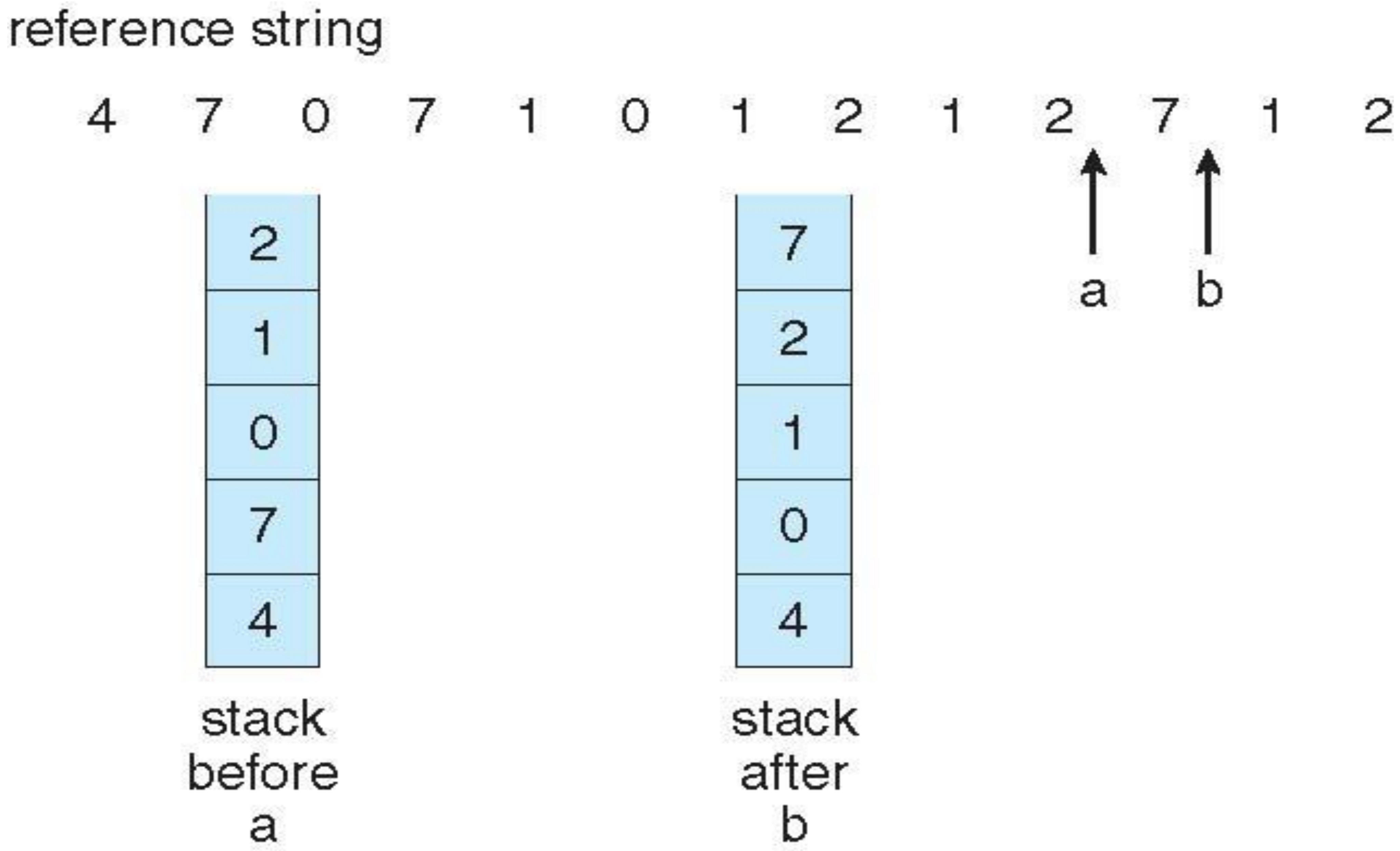
- Use past knowledge rather than future
- Replace page that has not been used in the most amount of time
- Associate time of last use with each page



# LRU Algorithm (Cont.)

- 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 find smallest value
    - Search through table needed
- 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
  - But each update more expensive
  - No search for replacement
- LRU and OPT are cases of **stack algorithms** that don't have Belady's Anomaly

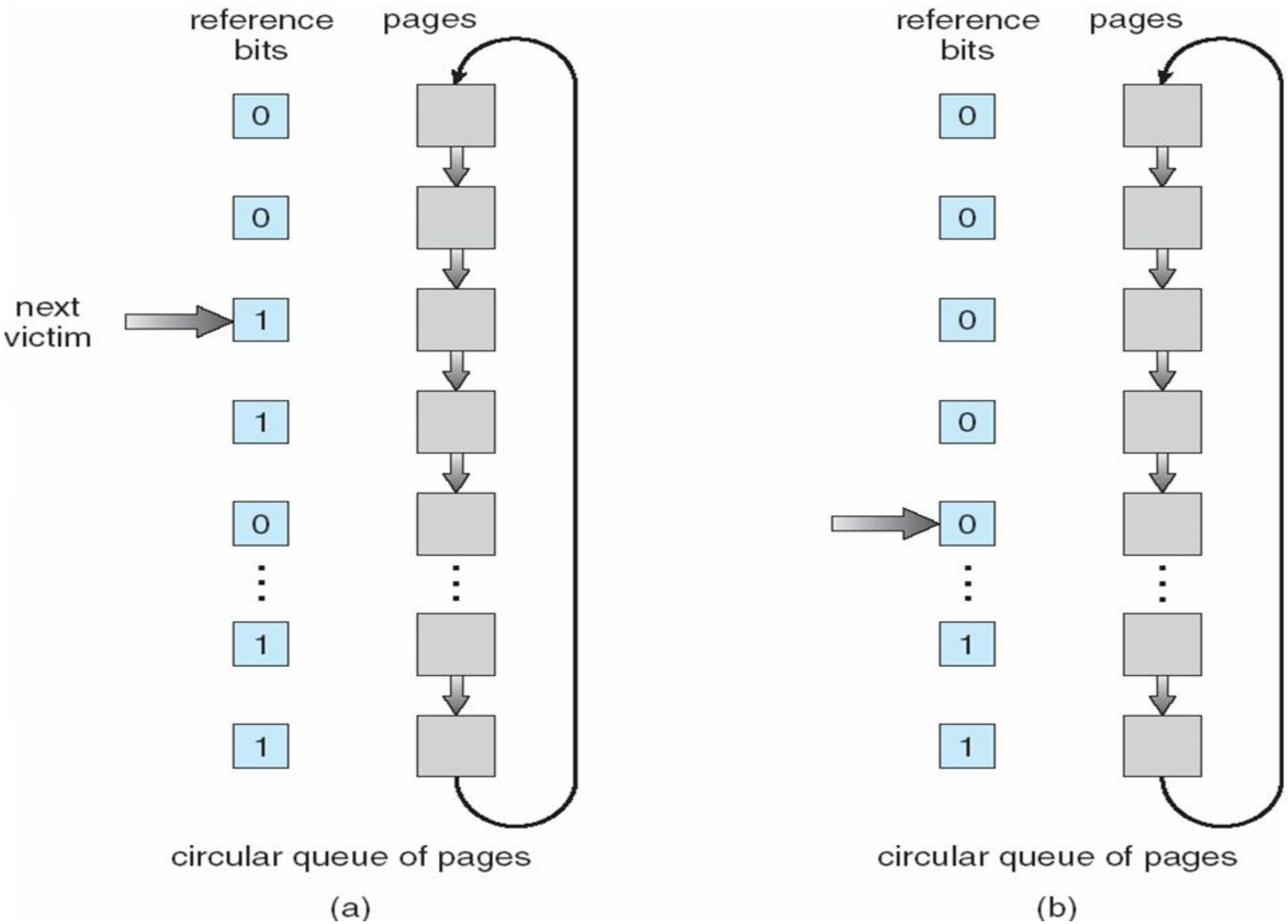
# Use Of A Stack to Record The Most Recent Page References



# LRU Approximation Algorithms

- LRU needs special hardware and still slow
- **Reference bit**
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
  - Replace any with reference bit = 0 (if one exists)
    - We do not know the order, however
- **Second-chance algorithm**
  - Generally FIFO, plus hardware-provided reference bit
  - Clock replacement
  - If page to be replaced has
    - Reference bit = 0 -> replace it
    - reference bit = 1 then:
      - set reference bit 0, leave page in memory
      - replace next page, 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
  - Not common
- **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

# Page-Buffering Algorithms

- Keep a pool of free frames, always
  - Then frame available when needed, not found at fault time
  - Read page into free frame and select victim to evict and add to free pool
  - When convenient, evict victim
- Possibly, keep list of modified pages
  - When backing store otherwise idle, write pages there and set to non-dirty
- Possibly, keep free frame contents intact and note what is in them
  - If referenced again before reused, no need to load contents again from disk
  - Generally useful to reduce penalty if wrong victim frame selected

# Applications and Page Replacement

- All of these algorithms have OS guessing about future page access
- Some applications have better knowledge – i.e. databases
- Memory intensive applications can cause double buffering
  - OS keeps copy of page in memory as I/O buffer
  - Application keeps page in memory for its own work
- Operating system can given direct access to the disk, getting out of the way of the applications
  - **Raw disk** mode
- Bypasses buffering, locking, etc

# Allocation of Frames

- Each process needs *minimum* number of frames
- 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*
- *Maximum* of course is total frames in the system
- Two major allocation schemes
  - fixed allocation
  - priority allocation
- Many variations

# Fixed Allocation

- Equal allocation – For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames
  - Keep some as free frame buffer pool
- Proportional allocation – Allocate according to the size of process
  - Dynamic as degree of multiprogramming, process sizes change

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

$$S = \sum s_i$$

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

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

$$m = 64$$

$$s_1 = 10$$

$$s_2 = 127$$

$$a_1 = \frac{10}{137} \times 64 \approx 5$$

$$a_2 = \frac{127}{137} \times 64 \approx 59$$

# 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
  - But then process execution time can vary greatly
  - But greater throughput so more common
- **Local replacement** – each process selects from only its own set of allocated frames
  - More consistent per-process performance
  - But possibly underutilized memory

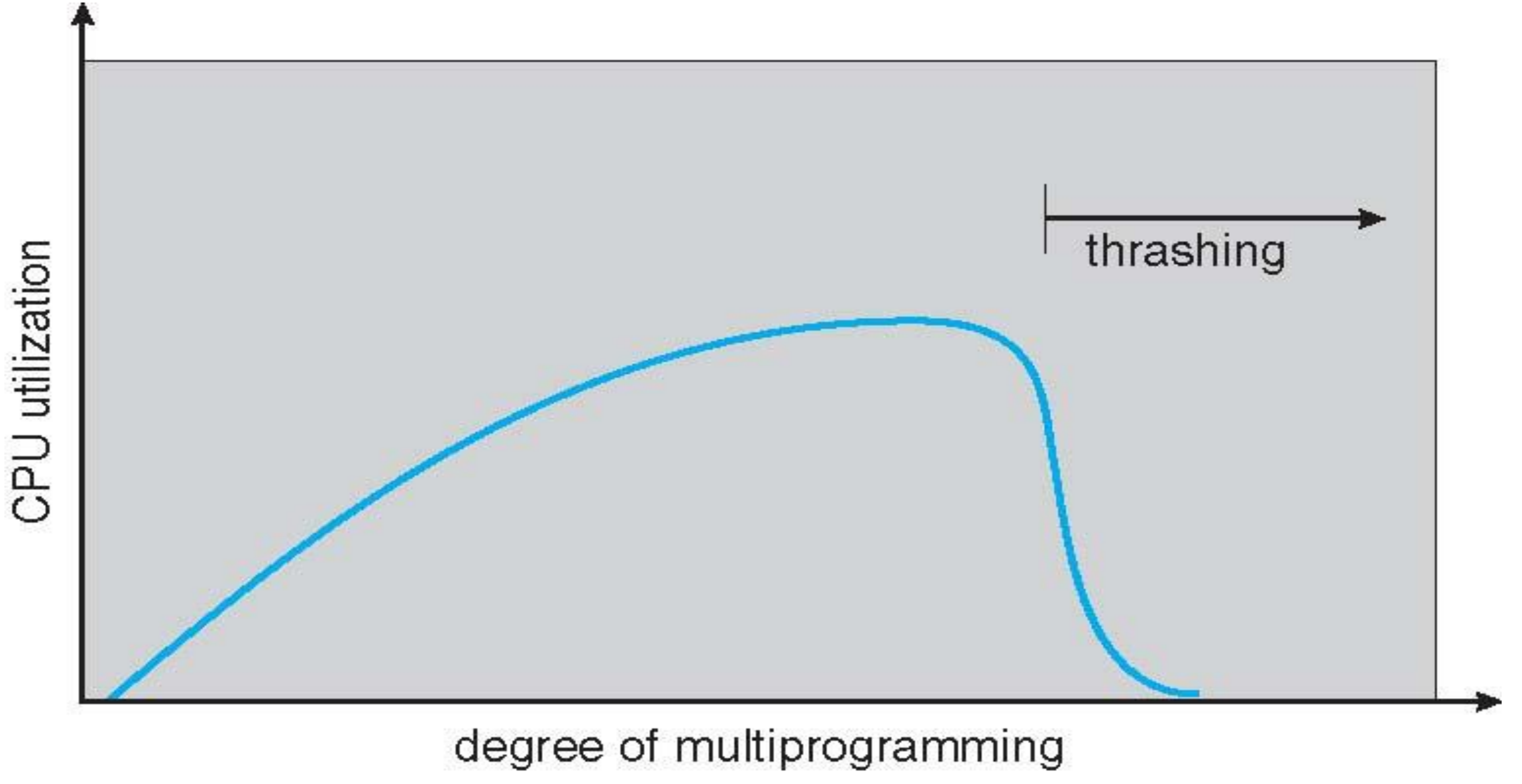
# Non-Uniform Memory Access

- So far all memory accessed equally
- Many systems are NUMA – speed of access to memory varies
  - Consider system boards containing CPUs and memory, interconnected over a system bus
- Optimal performance comes from allocating memory “close to” the CPU on which the thread is scheduled
  - And modifying the scheduler to schedule the thread on the same system board when possible
  - Solved by Solaris by creating **lgroups**
    - Structure to track CPU / Memory low latency groups
    - Used my schedule and pager
    - When possible schedule all threads of a process and allocate all memory for that process within the lgroup

# Thrashing

- If a process does not have "enough" pages, the page-fault rate is very high
  - Page fault to get page
  - Replace existing frame
  - But quickly need replaced frame back
  - This leads to:
    - Low CPU utilization
    - Operating system thinking 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.)

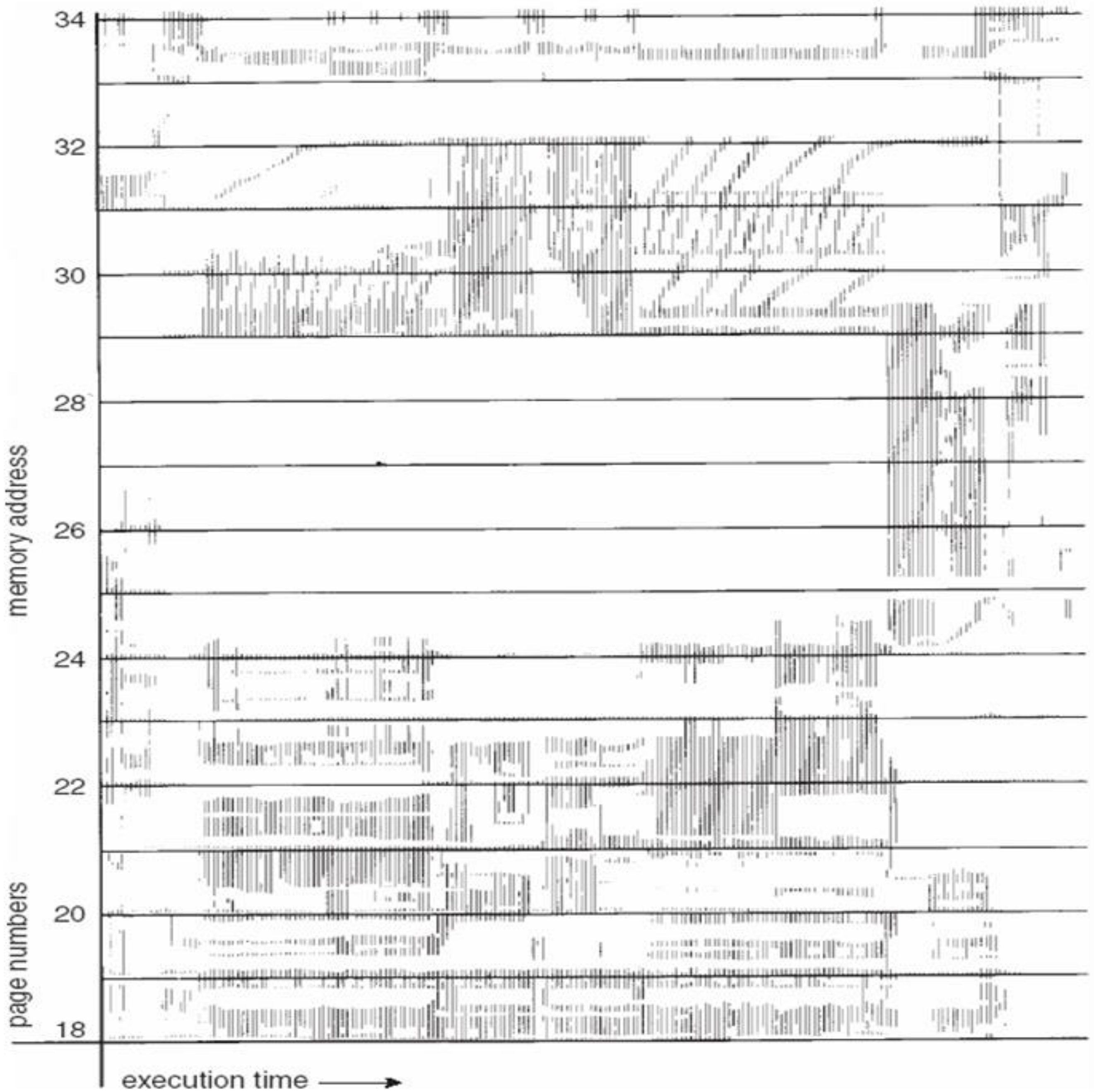


# Demand Paging and Thrashing

- Why does demand paging work?
  **Locality model**
  - Process migrates from one locality to another
  - Localities may overlap

- Why does thrashing occur?
  $\Sigma$ size of locality > total memory size
  - Limit effects by using local or priority page replacement

# Locality In A Memory-Reference Pattern



# Working-Set Model

- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  Example: 10,000 instructions

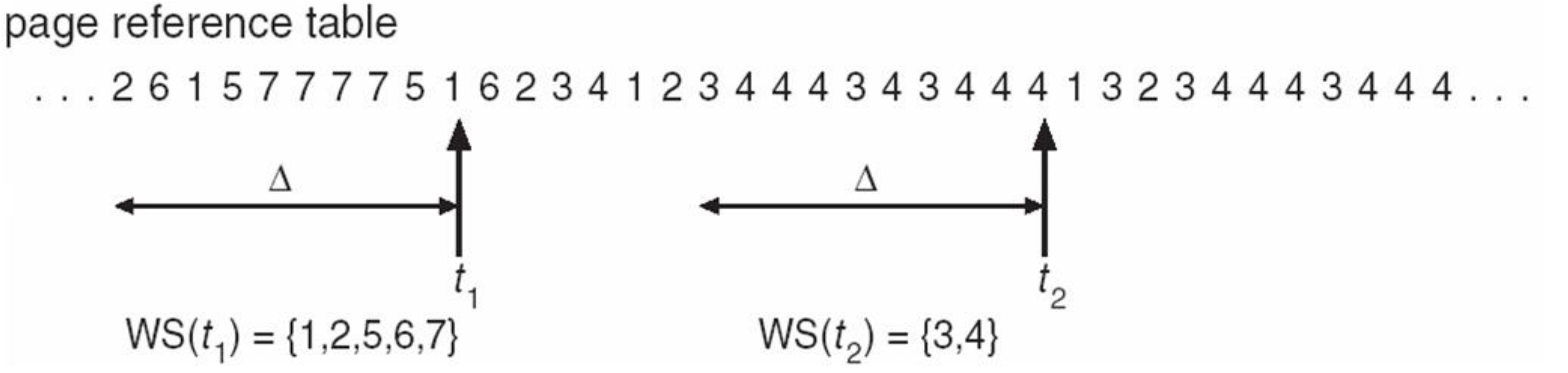
- $WSS_i$ (working set of Process $P_i$) =
  total number of pages referenced in the most recent $\Delta$ (varies in time)
  - if $\Delta$ too small will not encompass entire locality
  - if $\Delta$ too large will encompass several localities
  - if $\Delta = \infty \Rightarrow$ will encompass entire program

- $D = \Sigma \, WSS_i \equiv$ total demand frames
  - Approximation of locality

- if $D > m \Rightarrow$ Thrashing

- Policy if $D$ > m, then suspend or swap out one of the processes

# Working-set model

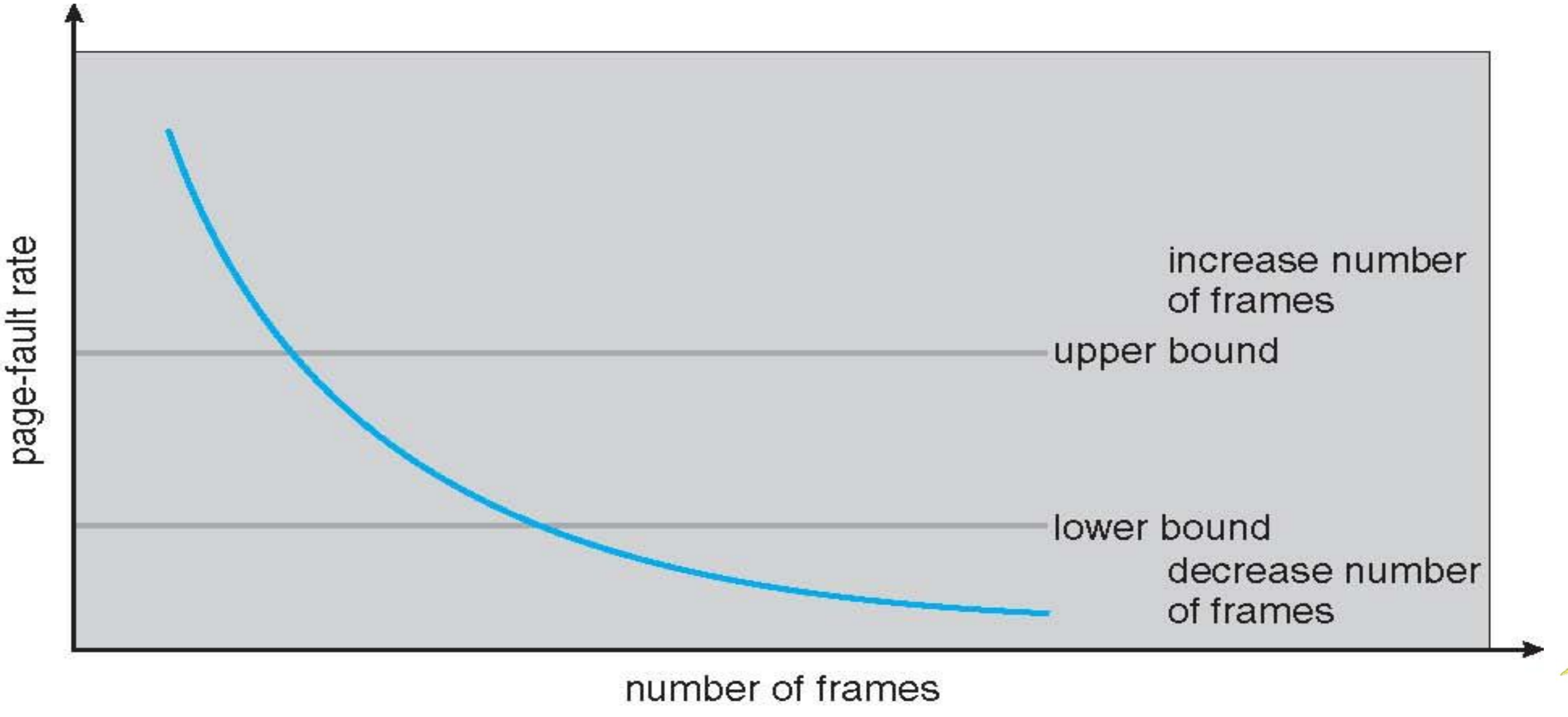


# Keeping Track of the Working Set

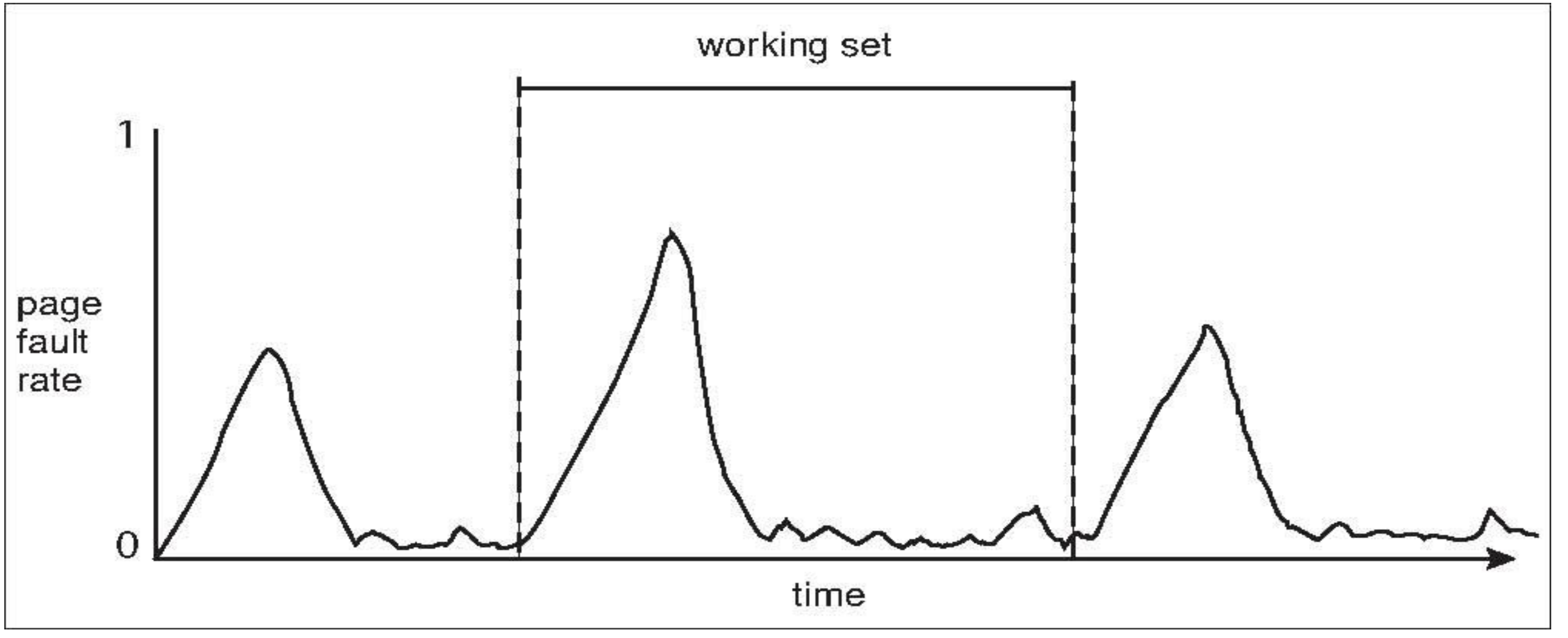
- Approximate with interval timer + a reference bit
- Example: $\Delta$ = 10,000
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 bits for each page
  - Whenever a timer interrupts copy and sets the values of all reference bits to 0
  - If one of the bits in memory = 1 $\Rightarrow$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

# Page-Fault Frequency

- More direct approach than WSS
- Establish "acceptable" **page-fault frequency** rate and use local replacement policy
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame



# Working Sets and Page Fault Rates



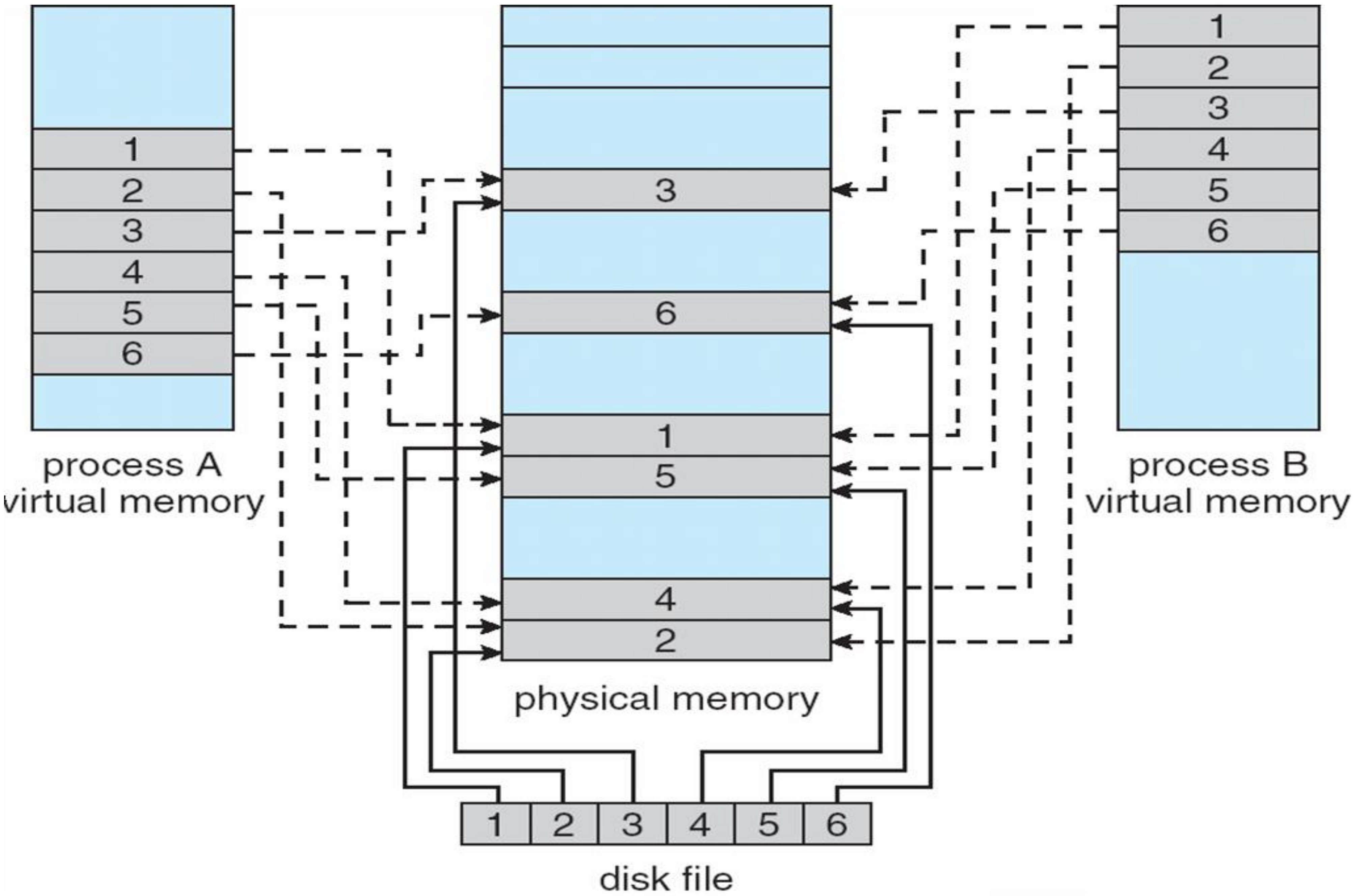
# Memory-Mapped Files

- 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 and speeds file access by driving file I/O through memory rather than `read()` and `write()` system calls
- Also allows several processes to map the same file allowing the pages in memory to be shared
- But when does written data make it to disk?
  - Periodically and / or at file `close()` time
  - For example, when the pager scans for dirty pages

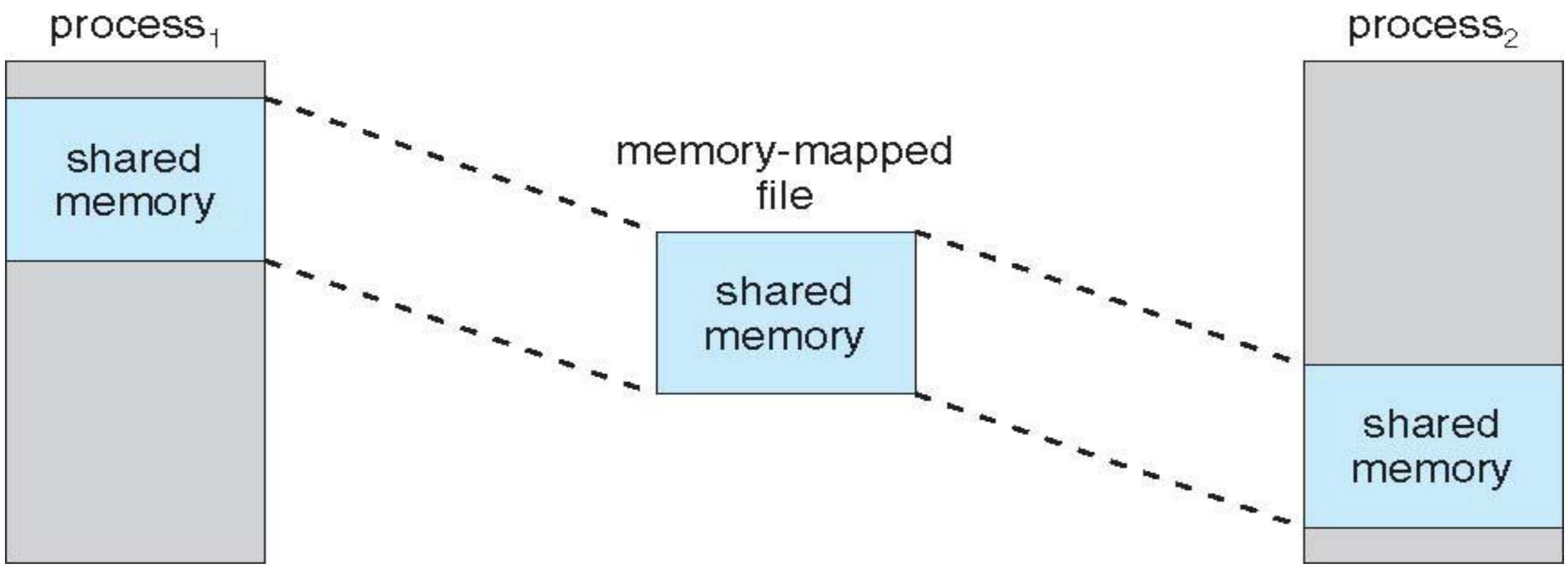
# Memory-Mapped File Technique for all I/O

- Some OSes use memory mapped files for standard I/O
- Process can explicitly request memory mapping a file via `mmap()` system call
  - Now file mapped into process address space
- For standard I/O (`open()`, `read()`, `write()`, `close()`), mmap anyway
  - But map file into kernel address space
  - Process still does read() and write()
    - Copies data to and from kernel space and user space
  - Uses efficient memory management subsystem
    - Avoids needing separate subsystem
- COW can be used for read/write non-shared pages
- Memory mapped files can be used for shared memory (although again via separate system calls)

# Memory Mapped Files



# Memory-Mapped Shared Memory in Windows



# Allocating Kernel Memory

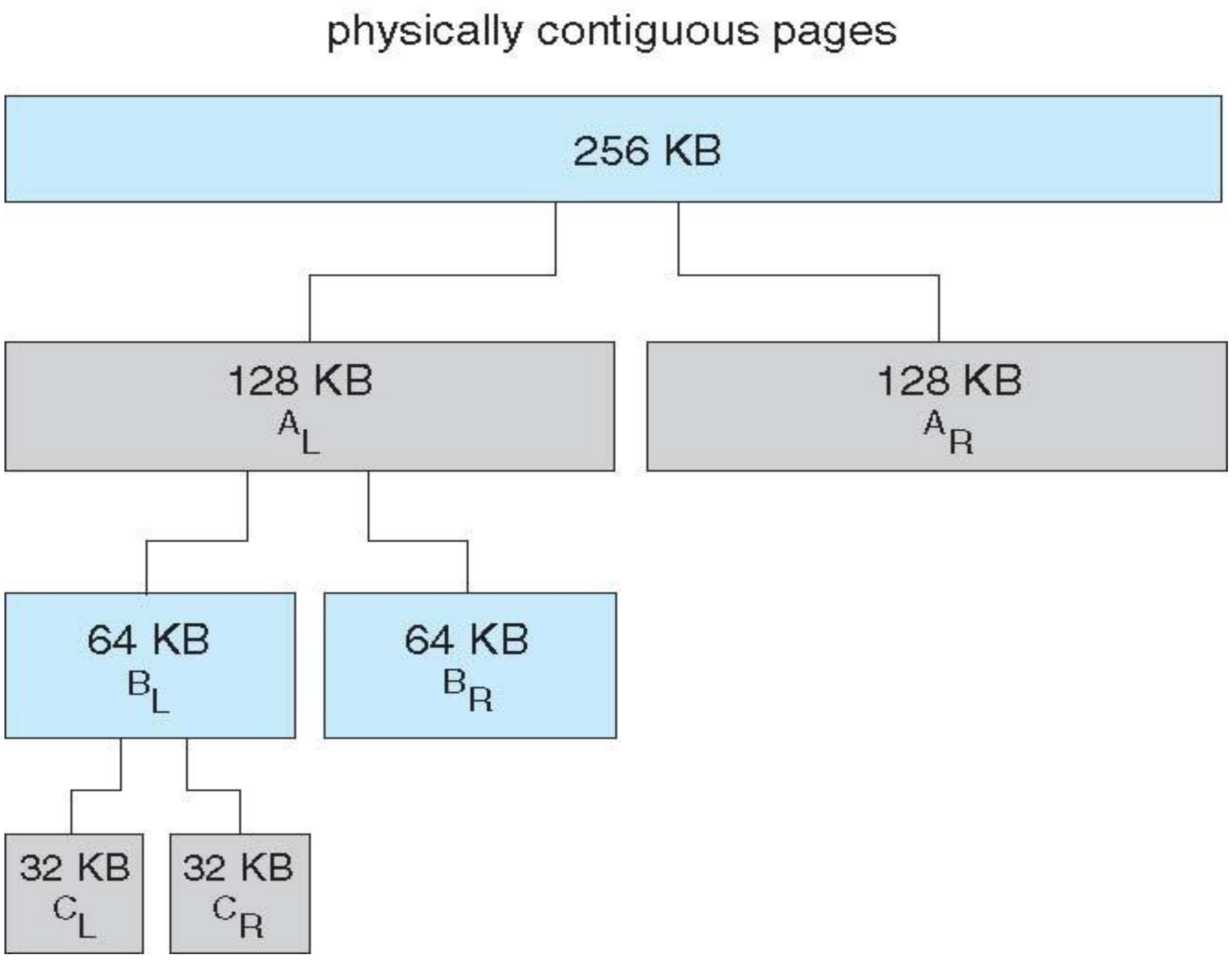
- Treated differently from user memory

- Often allocated from a free-memory pool
  - Kernel requests memory for structures of varying sizes
  - Some kernel memory needs to be contiguous
    - I.e. for device I/O

# 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
- For example, assume 256KB chunk available, kernel requests 21KB
  - Split into $A_{L\ \text{and}}$ $A_r$ of 128KB each
    - One further divided into $B_L$ and $B_R$ of 64KB
      - One further into $C_L$ and $C_R$ of 32KB each – one used to satisfy request
- Advantage – quickly coalesce unused chunks into larger chunk
- Disadvantage - fragmentation

# Buddy System Allocator

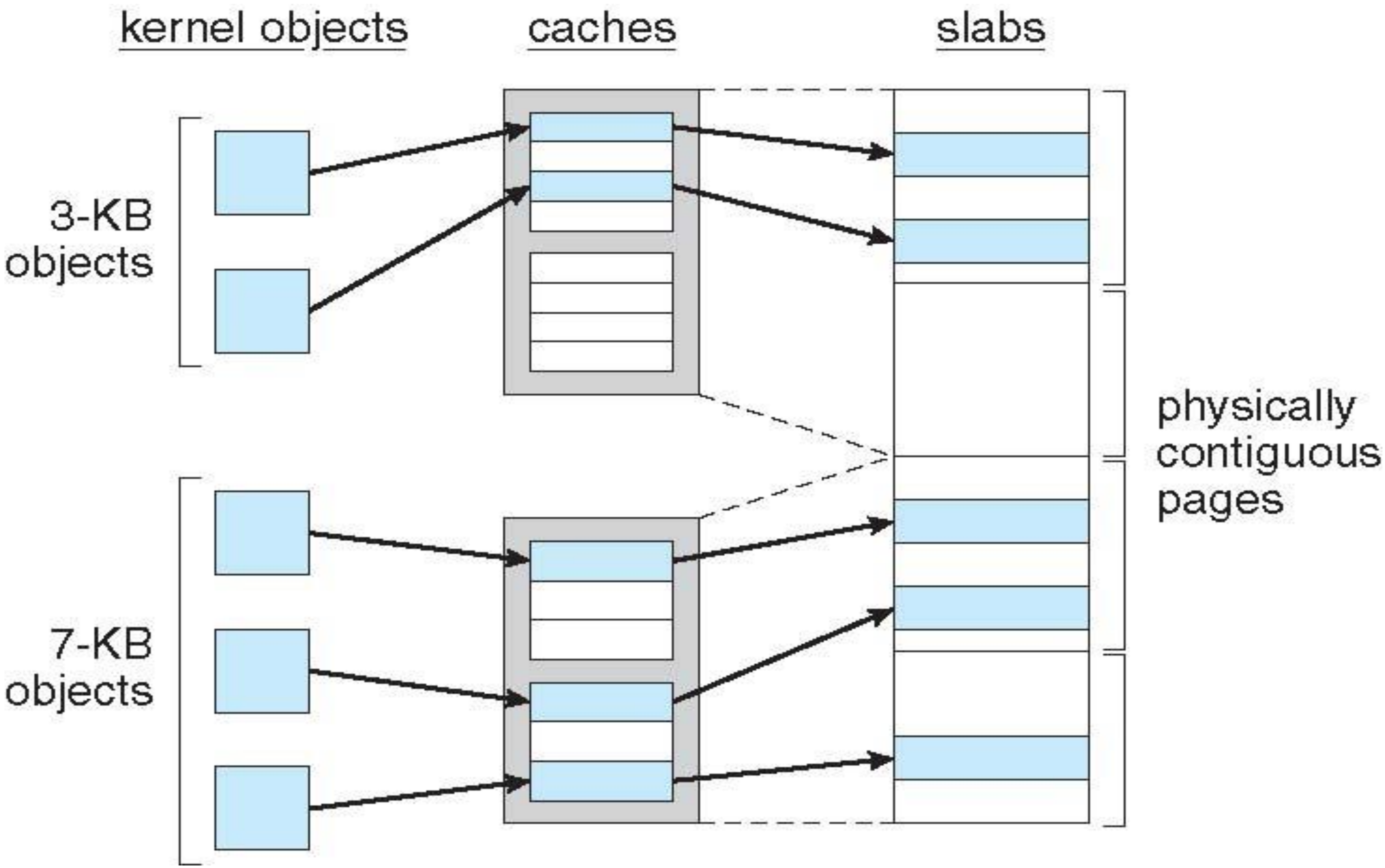


# Allocating Kernel Memory

## 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



# Other Considerations
## 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 < than the cost of prepaging
    $s * (1- \alpha)$ unnecessary pages?
  - $\alpha$ near zero $\Rightarrow$ prepaging loses

# Other Issues
## Page Size

- Sometimes OS designers have a choice
  - Especially if running on custom-built CPU
- Page size selection must take into consideration:
  - Fragmentation
  - Page table size
  - **Resolution**
  - I/O overhead
  - Number of page faults
  - Locality
  - TLB size and effectiveness
- Always power of 2, usually in the range $2^{12}$ (4,096 bytes) to $2^{22}$ (4,194,304 bytes)
- On average, growing over time

# 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
  - 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;
```

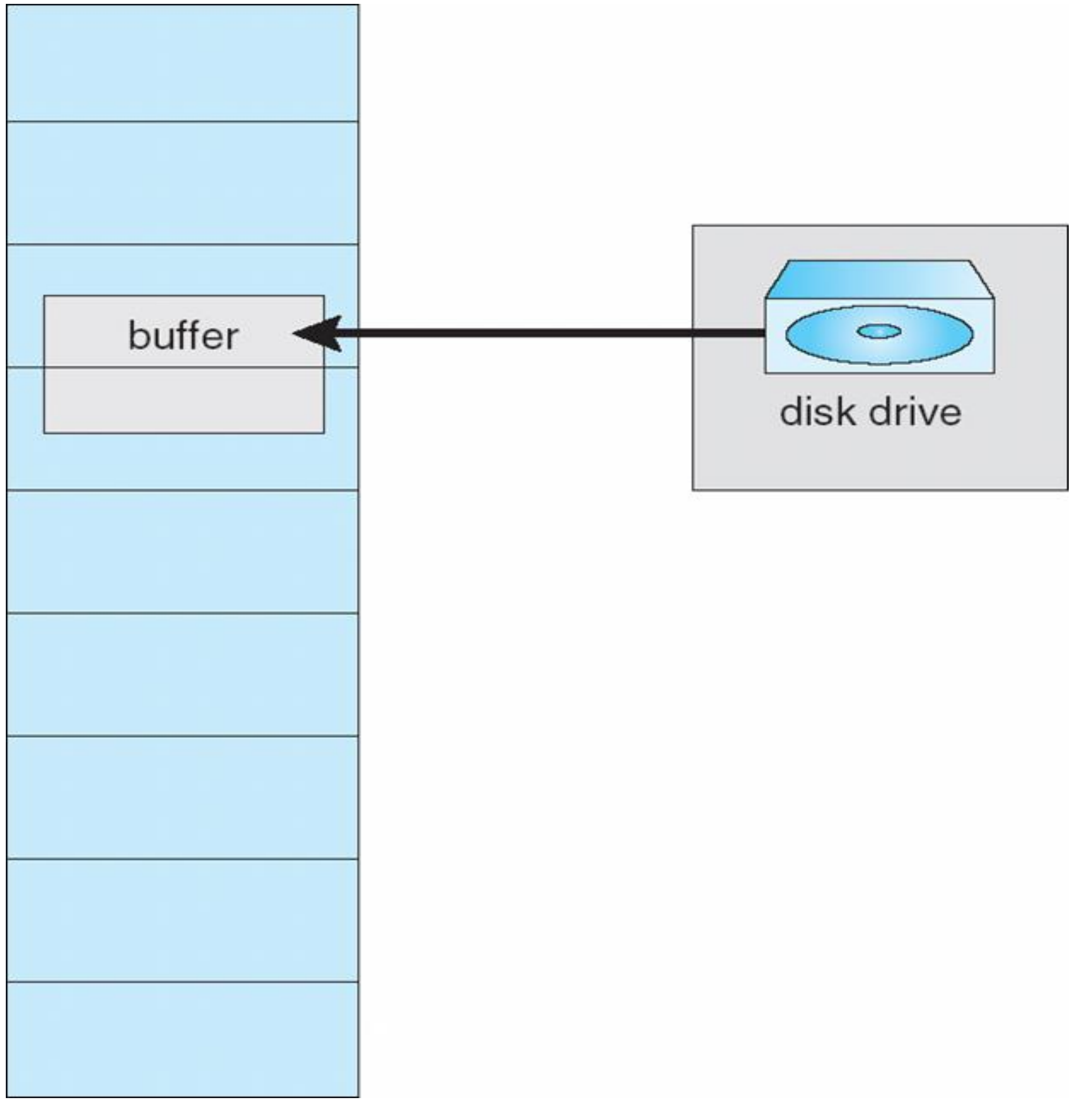
128 page faults

# 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



# Operating System Examples

- Windows XP
- Solaris

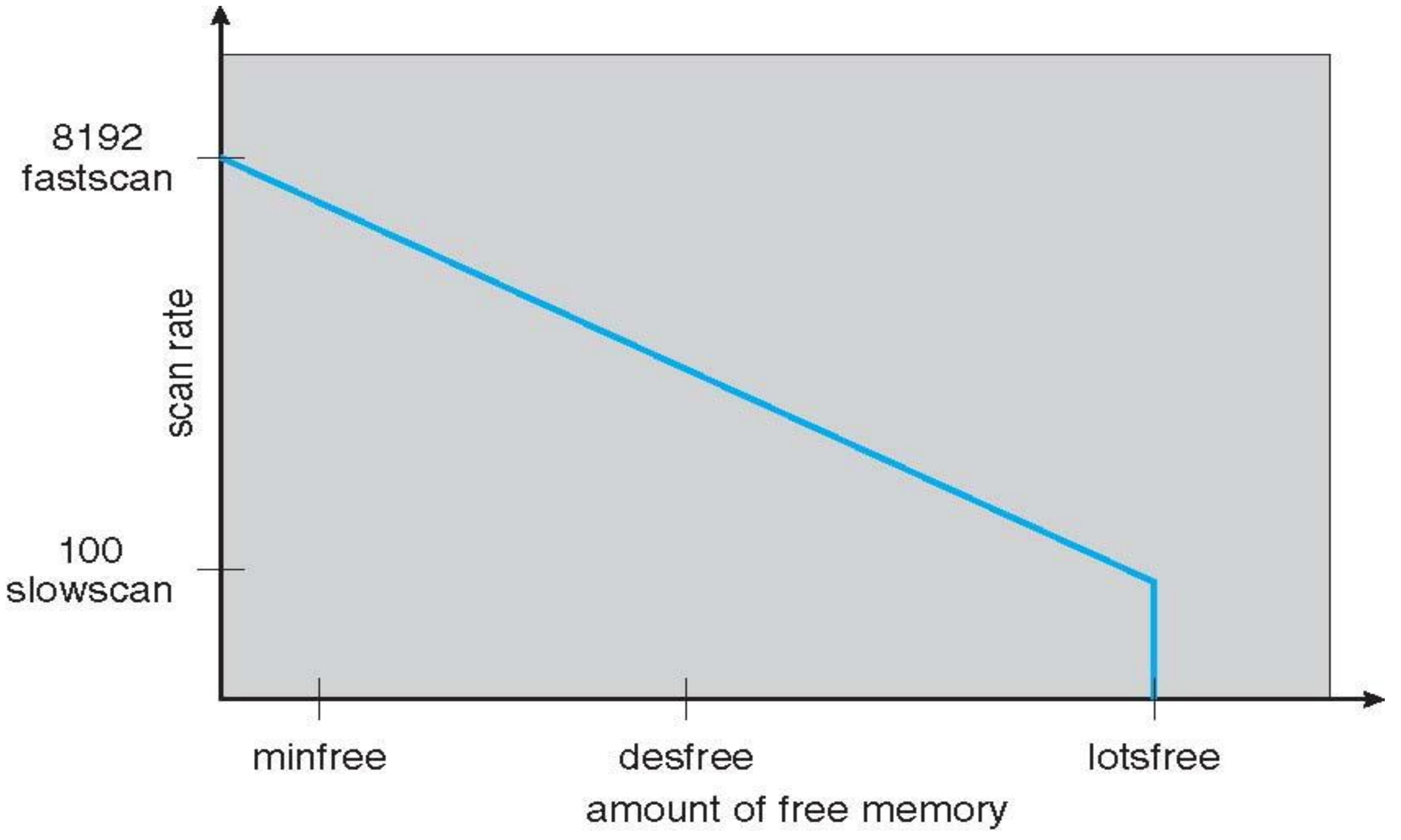
# Windows XP

- 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
- Priority paging gives priority to process code pages

# Solaris 2 Page Scanner



# Exercise (3/1)



Kernel processes typically require memory to be allocated using pages that are physically contiguous. The buddy system allocates memory to kernel processes in units sized according to a power of 2, which often results in fragmentation. Slab allocators assign kernel data structures to caches associated with slabs, which are made up of one or more physically contiguous pages. With slab allocation, no memory is wasted due to fragmentation, and memory requests can be satisfied quickly.

In addition to requiring us to solve the major problems of page replacement and frame allocation, the proper design of a paging system requires that we consider prepaging, page size, TLB reach, inverted page tables, program structure, I/O interlock and page locking, and other issues.

## Exercises

**9.1** Assume that a program has just referenced an address in virtual memory. Describe a scenario in which each of the following can occur. (If no such scenario can occur, explain why.)

- TLB miss with no page fault
- TLB miss and page fault
- TLB hit and no page fault
- TLB hit and page fault

**9.2** A simplified view of thread states is ***Ready***, ***Running***, and ***Blocked***, where a thread is either ready and waiting to be scheduled, is running on the processor, or is blocked (for example, waiting for I/O). This is illustrated in Figure 9.30. Assuming a thread is in the Running state, answer the following questions, and explain your answer:

a. Will the thread change state if it incurs a page fault? If so, to what state will it change?

b. Will the thread change state if it generates a TLB miss that is resolved in the page table? If so, to what state will it change?

c. Will the thread change state if an address reference is resolved in the page table? If so, to what state will it change?

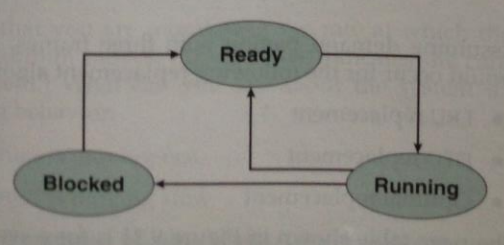



**Figure 9.30** Thread state diagram for Exercise 9.2.



**9.3** Consider a system that uses pure demand paging.

a. When a process first starts execution, how would you characterize the page-fault rate?

b. Once the working set for a process is loaded into memory, how would you characterize the page-fault rate?

c. Assume that a process changes its locality and the size of the new working set is too large to be stored in available free memory. Identify some options system designers could choose from to handle this situation.

**9.4** What is the copy-on-write feature, and under what circumstances is its use beneficial? What hardware support is required to implement this feature?

**9.5** A certain computer provides its users with a virtual memory space of $2^{32}$ bytes. The computer has $2^{22}$ bytes of physical memory. The virtual memory is implemented by paging, and the page size is 4,096 bytes. A user process generates the virtual address 11123456. Explain how the system establishes the corresponding physical location. Distinguish between software and hardware operations.

**9.6** Assume that we have a demand-paged memory. The page table is held in registers. It takes 8 milliseconds to service a page fault if an empty frame is available or if the replaced page is not modified and 20 milliseconds if the replaced page is modified. Memory-access time is 100 nanoseconds.

Assume that the page to be replaced is modified 70 percent of the time. What is the maximum acceptable page-fault rate for an effective access time of no more than 200 nanoseconds?

**9.7** When a page fault occurs, the process requesting the page must block while waiting for the page to be brought from disk into physical memory. Assume that there exists a process with five user-level threads and that the mapping of user threads to kernel threads is one to one. If one user thread incurs a page fault while accessing its stack, would the other user threads belonging to the same process also be affected by the page fault—that is, would they also have to wait for the faulting page to be brought into memory? Explain.

**9.8** Consider the following page reference string:

7, 2, 3, 1, 2, 5, 3, 4, 6, 7, 7, 1, 0, 5, 4, 6, 2, 3, 0 , 1.

Assuming demand paging with three frames, how many page faults would occur for the following replacement algorithms?

- LRU replacement
- FIFO replacement
- Optimal replacement

**9.9** The page table shown in Figure 9.31 is for a system with 16-bit virtual and physical addresses and with 4,096-byte pages. The reference bit is

# Exercise (2/3)



| Page | Page Frame | Reference Bit |
|---|---|---|
| 0 | 9 | 0 |
| 1 | 1 | 0 |
| 2 | 14 | 0 |
| 3 | 10 | 0 |
| 4 | – | 0 |
| 5 | 13 | 0 |
| 6 | 8 | 0 |
| 7 | 15 | 0 |
| 8 | – | 0 |
| 9 | 0 | 0 |
| 10 | 5 | 0 |
| 11 | 4 | 0 |
| 12 | – | 0 |
| 13 | – | 0 |
| 14 | 3 | 0 |
| 15 | 2 | 0 |

**Figure 9.31** Page table for Exercise 9.9.

set to 1 when the page has been referenced. Periodically, a thread zeroes out all values of the reference bit. A dash for a page frame indicates the page is not in memory. The page-replacement algorithm is localized LRU, and all numbers are provided in decimal.

a. Convert the following virtual addresses (in hexadecimal) to the equivalent physical addresses. You may provide answers in either hexadecimal or decimal. Also set the reference bit for the appropriate entry in the page table.

- 0xE12C
- 0x3A9D
- 0xA9D9
- 0x7001
- 0xACA1

b. Using the above addresses as a guide, provide an example of a logical address (in hexadecimal) that results in a page fault.

c. From what set of page frames will the LRU page-replacement algorithm choose in resolving a page fault?

**9.10** Assume that you are monitoring the rate at which the pointer in the clock algorithm moves. (The pointer indicates the candidate page for replacement.) What can you say about the system if you notice the following behavior:

a. Pointer is moving fast.

b. Pointer is moving slow.

**9.11** Discuss situations in which the least frequently used (LFU) page-replacement algorithm generates fewer page faults than the least recently used (LRU) page-replacement algorithm. Also discuss under what circumstances the opposite holds.



**9.12** Discuss situations in which the most frequently used (MFU) page-replacement algorithm generates fewer page faults than the least recently used (LRU) page-replacement algorithm. Also discuss under what circumstances the opposite holds.

**9.13** The VAX/VMS system uses a FIFO replacement algorithm for resident pages and a free-frame pool of recently used pages. Assume that the free-frame pool is managed using the LRU replacement policy. Answer the following questions:

a. If a page fault occurs and the page does not exist in the free-frame pool, how is free space generated for the newly requested page?

b. If a page fault occurs and the page exists in the free-frame pool, how is the resident page set and the free-frame pool managed to make space for the requested page?

c. What does the system degenerate to if the number of resident pages is set to one?

d. What does the system degenerate to if the number of pages in the free-frame pool is zero?

**9.14** Consider a demand-paging system with the following time-measured utilizations:

| | |
|---|---|
| CPU utilization | 20% |
| Paging disk | 97.7% |
| Other I/O devices | 5% |

For each of the following, indicate whether it will (or is likely to) improve CPU utilization. Explain your answers.

a. Install a faster CPU.

b. Install a bigger paging disk.

c. Increase the degree of multiprogramming.

d. Decrease the degree of multiprogramming.

e. Install more main memory.

f. Install a faster hard disk or multiple controllers with multiple hard disks.

g. Add prepaging to the page-fetch algorithms.

h. Increase the page size.

**9.15** Suppose that a machine provides instructions that can access memory locations using the one-level indirect addressing scheme. What sequence of page faults is incurred when all of the pages of a program are currently nonresident and the first instruction of the program is an indirect memory-load operation? What happens when the operating

# Exercise (3/3)



system is using a per-process frame allocation technique and only two pages are allocated to this process?

**9.16** Suppose that your replacement policy (in a paged system) is to examine each page regularly and to discard that page if it has not been used since the last examination. What would you gain and what would you lose by using this policy rather than LRU or second-chance replacement?

**9.17** A page-replacement algorithm should minimize the number of page faults. We can achieve this minimization by distributing heavily used pages evenly over all of memory, rather than having them compete for a small number of page frames. We can associate with each page frame a counter of the number of pages associated with that frame. Then, to replace a page, we can search for the page frame with the smallest counter.

a. Define a page-replacement algorithm using this basic idea. Specifically address these problems:

   i. What is the initial value of the counters?
   ii. When are counters increased?
   iii. When are counters decreased?
   iv. How is the page to be replaced selected?

b. How many page faults occur for your algorithm for the following reference string with four page frames?

   1, 2, 3, 4, 5, 3, 4, 1, 6, 7, 8, 7, 8, 9, 7, 8, 9, 5, 4, 5, 4, 2.

c. What is the minimum number of page faults for an optimal page-replacement strategy for the reference string in part b with four page frames?

**9.18** Consider a demand-paging system with a paging disk that has an average access and transfer time of 20 milliseconds. Addresses are translated through a page table in main memory, with an access time of 1 microsecond per memory access. Thus, each memory reference through the page table takes two accesses. To improve this time, we have added an associative memory that reduces access time to one memory reference if the page-table entry is in the associative memory.

Assume that 80 percent of the accesses are in the associative memory and that, of those remaining, 10 percent (or 2 percent of the total) cause page faults. What is the effective memory access time?

**9.19** What is the cause of thrashing? How does the system detect thrashing? Once it detects thrashing, what can the system do to eliminate this problem?

**9.20** Is it possible for a process to have two working sets, one representing data and another representing code? Explain.

**9.21** Consider the parameter $\Delta$ used to define the working-set window in the working-set model. When $\Delta$ is set to a small value, what is the effect on the page-fault frequency and the number of active (nonsuspended)



processes currently executing in the system? What is the effect when $\Delta$ is set to a very high value?

**9.22** In a 1,024-KB segment, memory is allocated using the buddy system. Using Figure 9.26 as a guide, draw a tree illustrating how the following memory requests are allocated:

- Request 6-KB
- Request 250 bytes
- Request 900 bytes
- Request 1,500 bytes
- Request 7-KB

Next, modify the tree for the following releases of memory. Perform coalescing whenever possible:

- Release 250 bytes
- Release 900 bytes
- Release 1,500 bytes

**9.23** A system provides support for user-level and kernel-level threads. The mapping in this system is one to one (there is a corresponding kernel thread for each user thread). Does a multithreaded process consist of (a) a working set for the entire process or (b) a working set for each thread? Explain

**9.24** The slab-allocation algorithm uses a separate cache for each different object type. Assuming there is one cache per object type, explain why this scheme doesn't scale well with multiple CPUs. What could be done to address this scalability issue?

**9.25** Consider a system that allocates pages of different sizes to its processes. What are the advantages of such a paging scheme? What modifications to the virtual memory system provide this functionality?

## Programming Problems

**9.26** Write a program that implements the FIFO, LRU, and optimal page-replacement algorithms presented in this chapter. First, generate a random page-reference string where page numbers range from 0 to 9. Apply the random page-reference string to each algorithm, and record the number of page faults incurred by each algorithm. Implement the replacement algorithms so that the number of page frames can vary from 1 to 7. Assume that demand paging is used.

**9.27** Repeat Exercise 3.22, this time using Windows shared memory. In particular, using the producer—consumer strategy, design two programs that communicate with shared memory using the Windows API as outlined in Section 9.7.2. The producer will generate the numbers specified in the Collatz conjecture and write them to a shared memory object