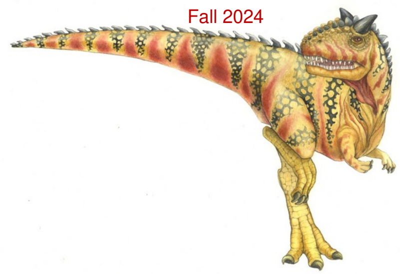
Lecture 7: Virtual Memory

Hossein Asadi (asadi@sharif.edu)

Rasool Jalili (jalili@sharif.edu)

Fall 2024



Lecture 7: Virtual Memory

- 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 Benefits of a Virtual Memory System
- To Explain Concepts of Demand Paging, Page-Replacement Algorithms, and Allocation of Page Frames
- To Discuss Principle of Working-Set Model
- To Examine Relationship between Shared Memory and Memory-Mapped Files
- To Explore how Kernel Memory is Managed

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
  - Each program takes less memory while running ➔ more programs run at the same time
    - Increased CPU utilization and throughput with no increase in response time or turnaround time
  - Less I/O needed to load or swap programs into memory ➔ each user program runs faster

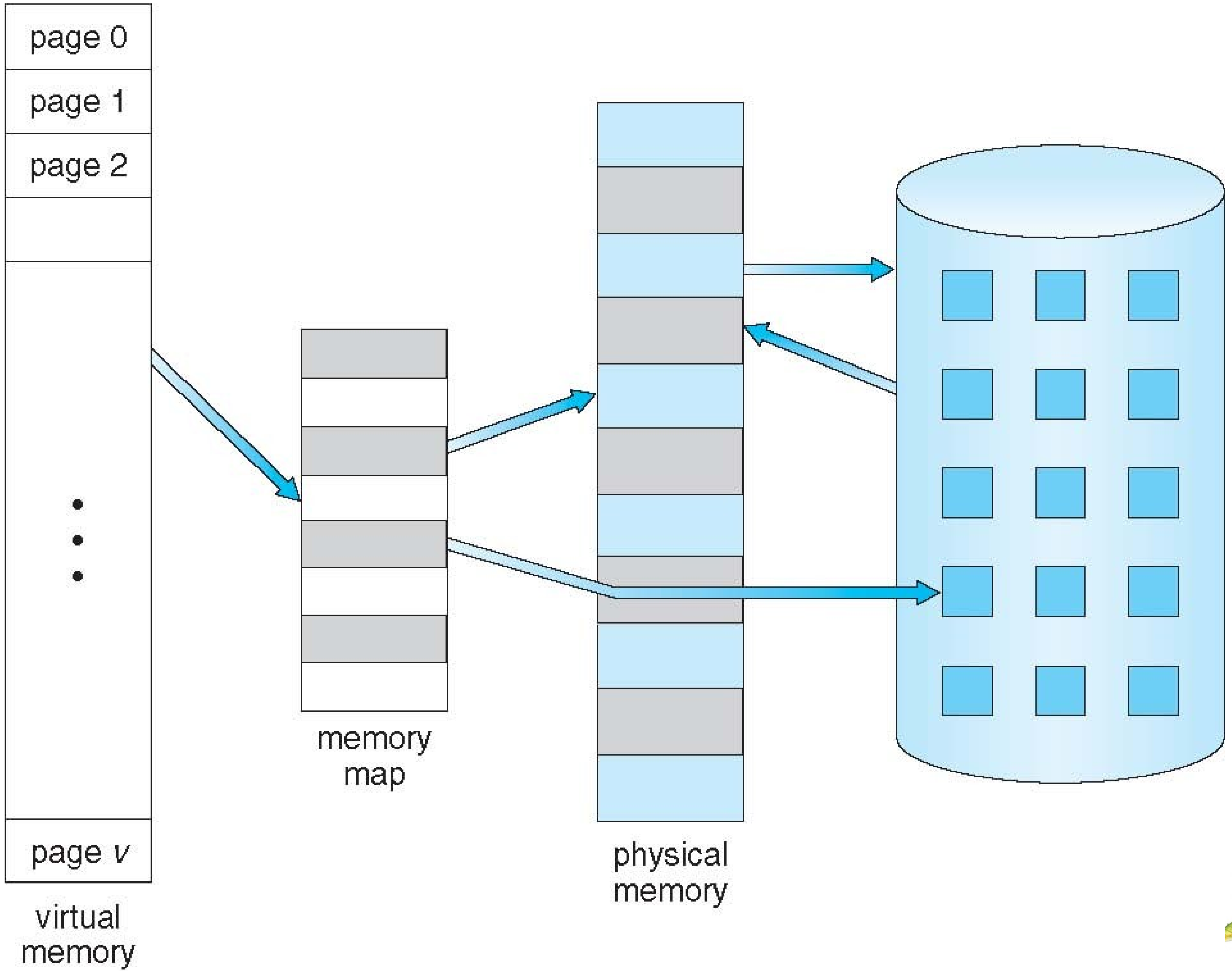
Background (cont.)

- **Virtual Memory** – separation of user logical memory from physical memory
  - Only part of 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

Background (cont.)

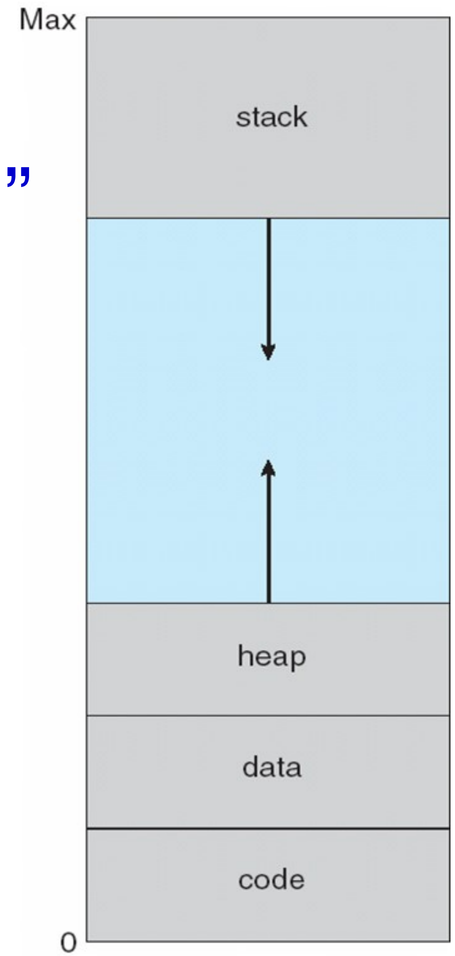
- **Virtual Address Space** – logical view of how process is stored in memory
  - Usually start at address 0, contiguous addresses until end of space
  - Meanwhile, physical memory organized in page frames
  - MMU must map logical to physical
- Virtual Memory can be Implemented via:
  - Demand paging
  - Demand segmentation

Virtual Memory Larger than Physical Memory



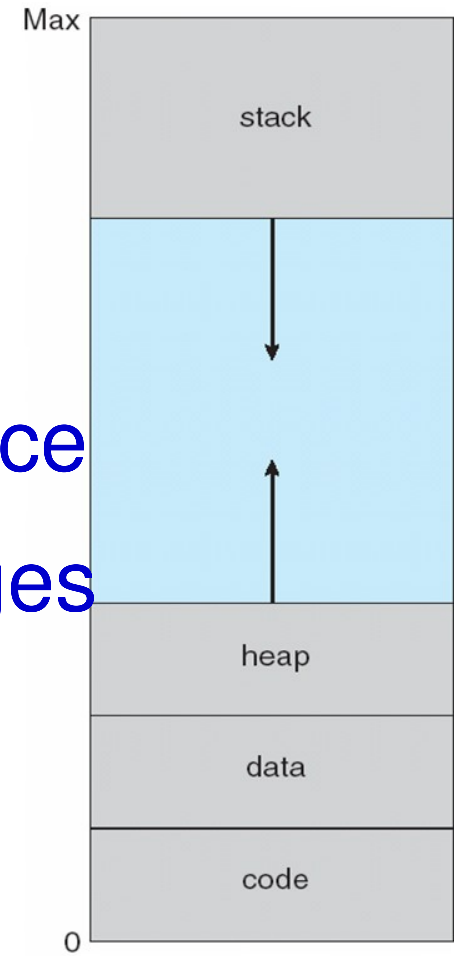
Virtual-Address Space

- Usually design Logical Address Space for Stack to start at Max Logical Address and grow “down” while heap grows “up”
  - Maximizes address space use
  - Unused address space between two is hole
    - No physical memory needed until heap or stack grows to a given new page

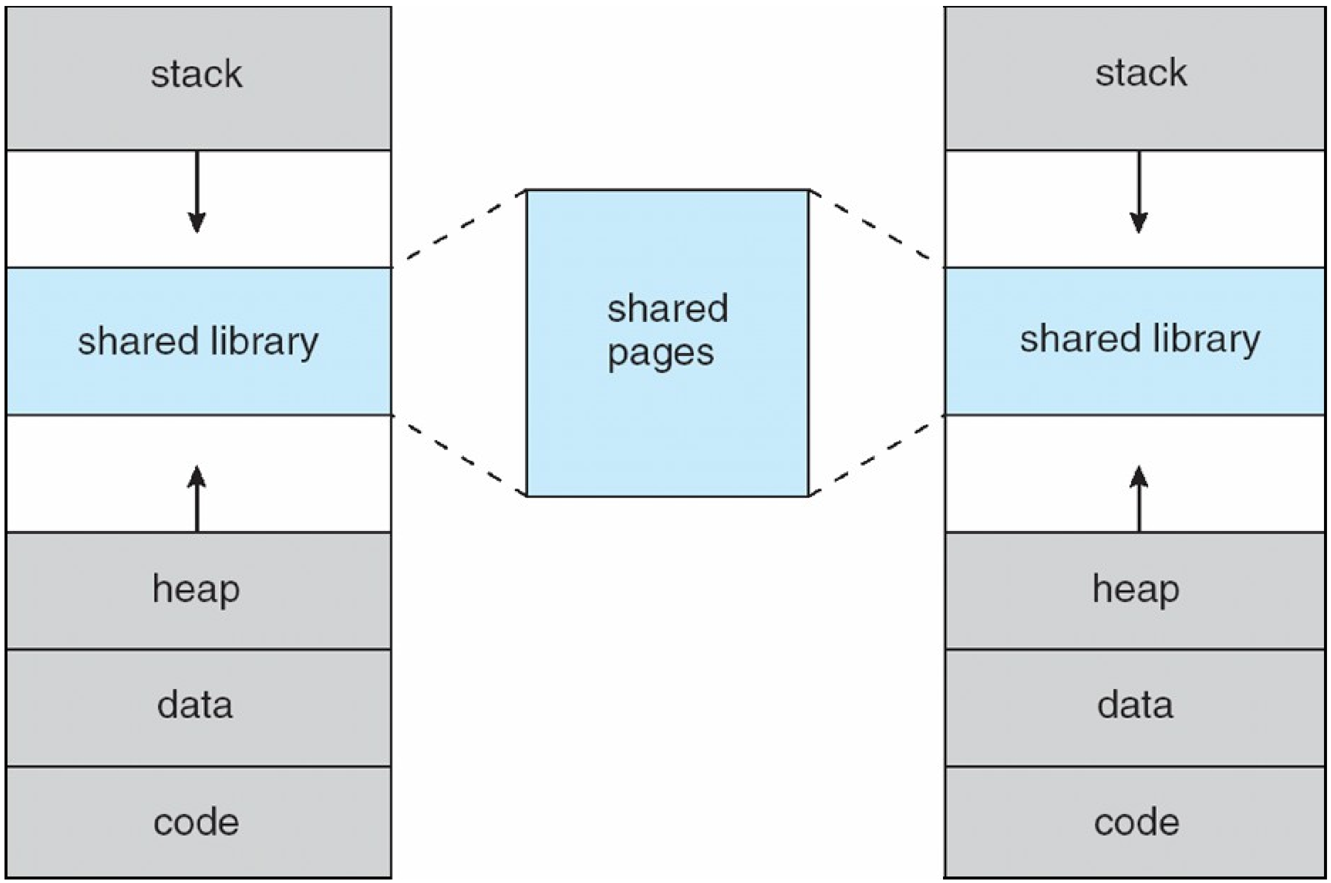


Virtual-Address Space (cont.)

- 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

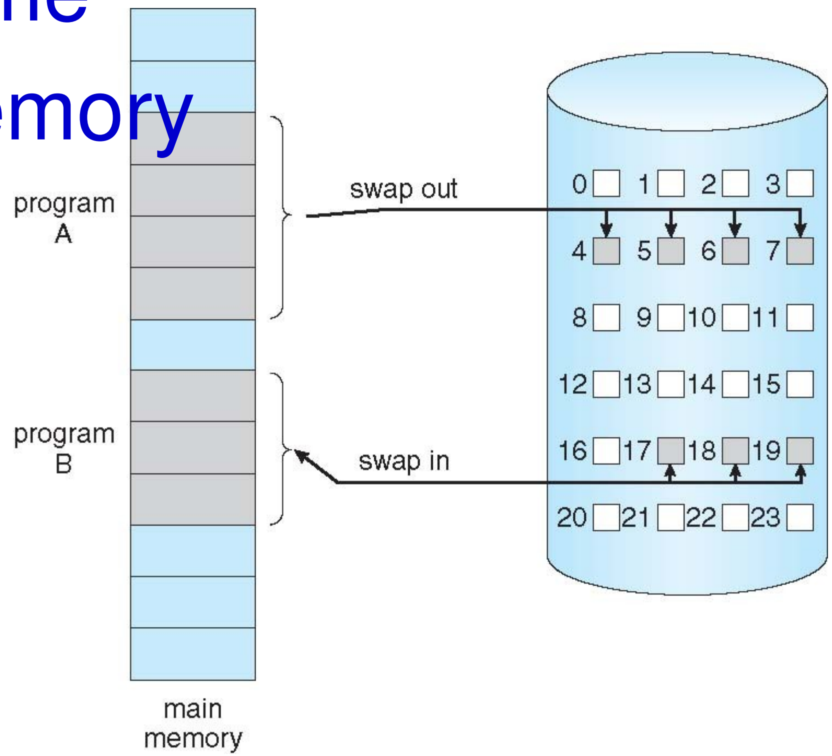


Shared Library Using Virtual Memory



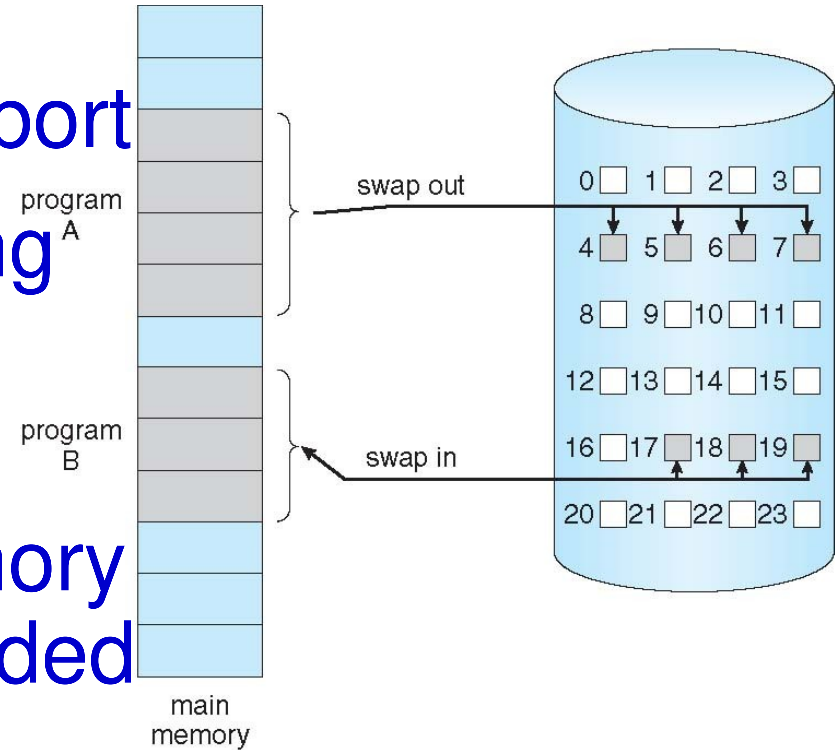
Demand Paging

- 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
- Similar to paging system with swapping



Demand Paging (cont.)

- Page is needed ⇒ reference to it
  - Invalid reference ⇒ abort
  - not-in-memory ⇒ Bring to memory
- **Lazy Swapper** – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**



Basic Concepts

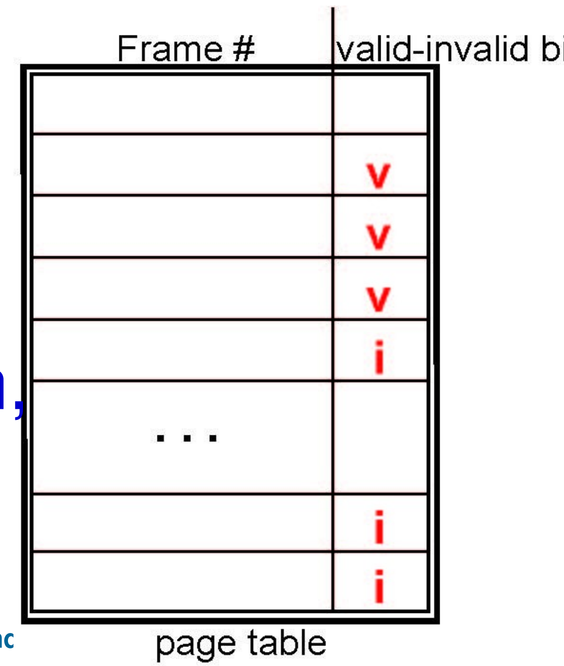
- With swapping, Pager Guesses which pages will be used before swapping out again
  - Instead, pager brings in only those pages into memory
- How to determine that set of pages?
  - Need new **MMU functionality** to implement demand paging
- If pages needed are already **memory resident**
  - No difference from non demand-paging

Basic Concepts (cont.)

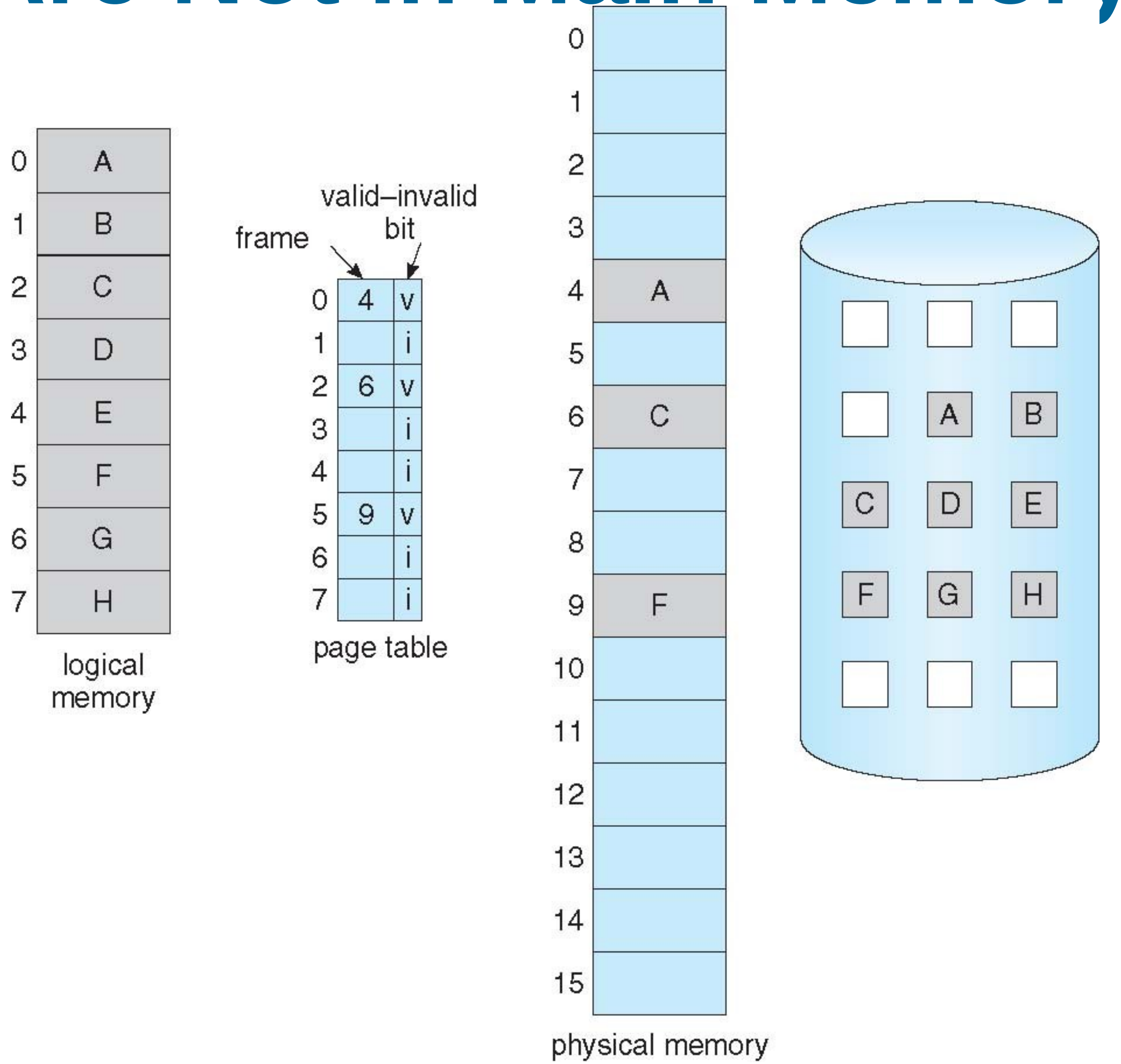
- If page needed and not memory resident
  - Need to detect and load page into memory from storage
    - Without changing program behavior
    - Without programmer needing to change code

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
  - Page table snapshot
  - During MMU address translation, if valid–invalid bit in page table entry is **i** ⇒ page fault



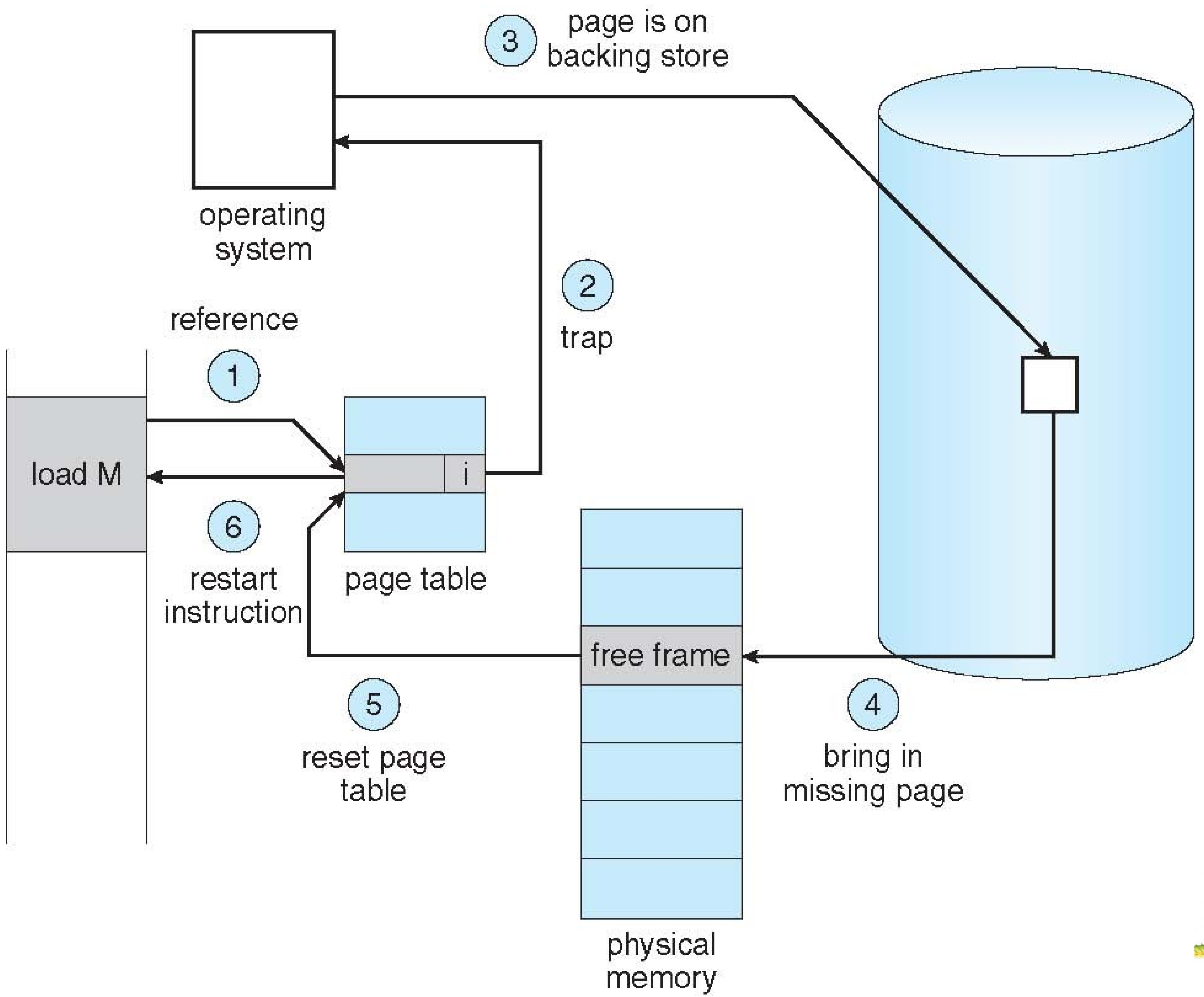
Page Table When Some Pages Are Not in Main Memory



Page Fault

1. If there is a reference to a page, 1st reference to that page will trap to OS: **Page Fault**
2. OS looks at table to decide:
   - Invalid reference ⇒ abort
   - Just not in memory ⇒ proceed with step 3
3. Find Free Frame
4. Swap page into frame via scheduled disk operation
5. Reset tables to indicate page now in memory (Set validation bit = **v**)
6. Restart instruction that caused page fault

Steps in Handling a Page Fault



Aspects of Demand Paging

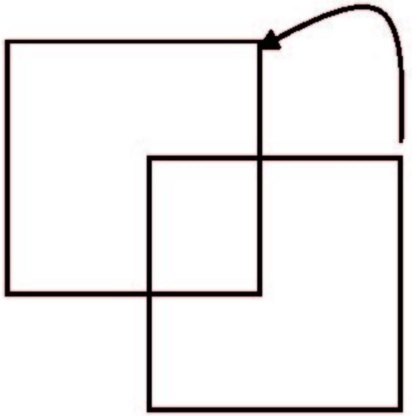
- Extreme case – start process with *no* pages in memory
  - OS sets instruction pointer to first instruction of process, non-memory-resident → page fault
  - For every other process pages on first access
  - **Pure demand paging**

Aspects of Demand Paging (cont.)

- Question:
  - Can an instruction cause multiple page faults?
  - Consider fetch and decode of instruction which adds 2 numbers from memory and stores result back to memory ➔ significant performance drop
  - Not a real case due to **locality of reference**
- HW 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?



Performance of Demand Paging

- Stages in Demand Paging (worse case)

1. Trap to OS
2. Save user registers and process state
3. Determine that the interrupt was a page fault
4. Check that page reference was legal and determine location of page on disk
5. Issue a read from disk to a free frame:
    1. Wait in a queue for this device until read request is serviced
    2. Wait for device seek and/or latency time
    3. Begin transfer of the page to a free frame

Performance of Demand Paging (cont.)

- Stages in Demand Paging (worse case)

6. While waiting, allocate CPU to some other user
7. Receive an interrupt from disk I/O subsystem (I/O completed)
8. Save registers and process state for other user
9. Determine that interrupt was from disk
10. Correct page table and other tables to show page is now in memory
11. Wait for CPU to be allocated to this process again
12. Restore user registers, process state, and new page table, then resume interrupted instruction

Performance of Demand Paging (cont.)

- Three Major Activities
  - Service interrupt – careful coding means just several hundred instructions needed
  - Read page – lots of time
  - Restart process – again just a small amount of time
- 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)

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

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 (40X slowdown)
- If seeking performance degradation less than 10%
  - 220 > 200 + 7,999,800 x p

    20 > 7,999,800 x p
  - ➔ p < .0000025 which means one page fault in every 400,000 memory accesses

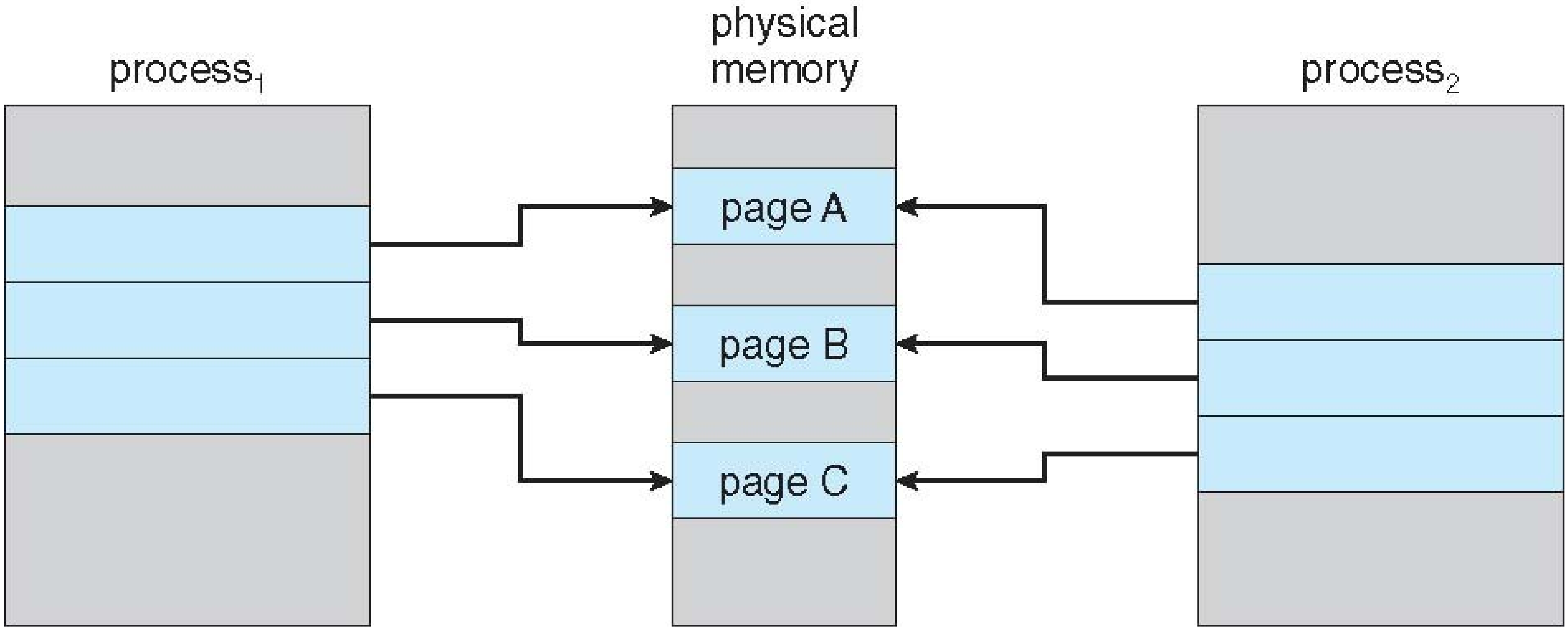
Copy-on-Write

- **Copy-on-Write** (COW) Allows both parent and child processes to initially ***share*** 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

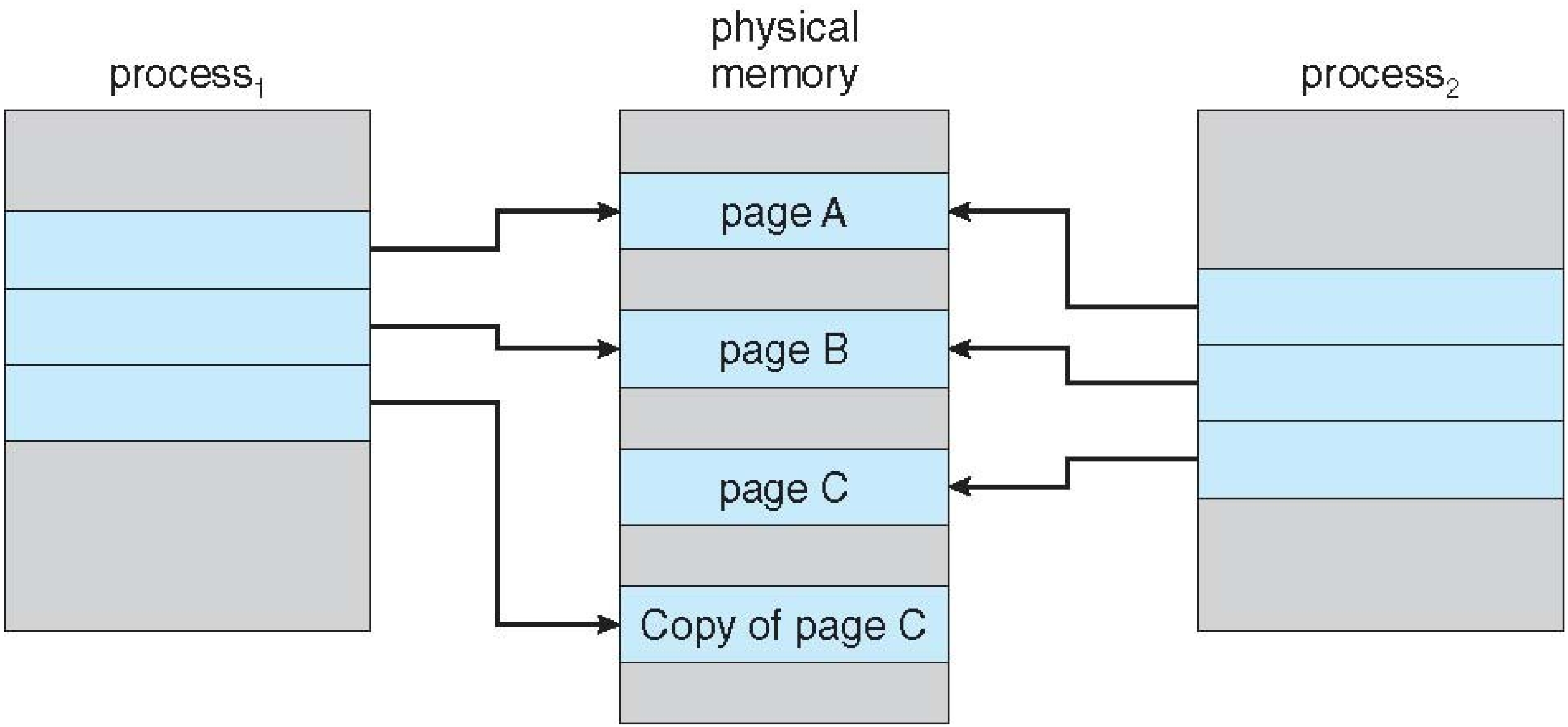
Copy-on-Write (cont.)

- In general, free pages are allocated from a **pool** of **zero-fill-on-demand** pages
  - Pool should always have free frames for fast demand page execution
    - Don’t want to have to free a frame as well as other processing on page fault
- `vfork()`
  - Virtual memory fork() system call
  - Has parent suspend and child using copy-on-write address space of parent

Before Process 1 Modifies Page C



After Process 1 Modifies Page C



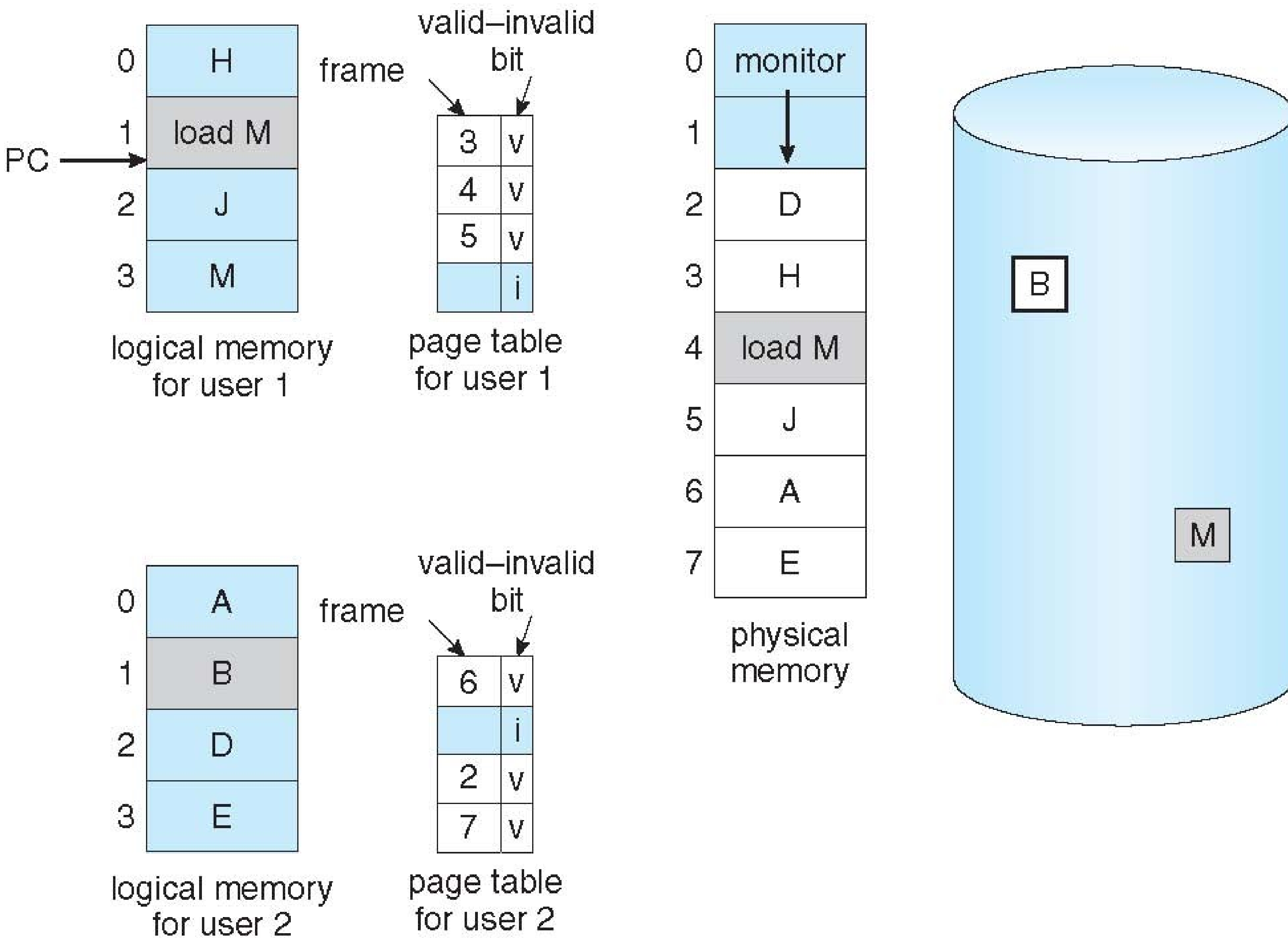
What Happens if There is no Free Frame?

- Used up by Process Pages
- 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 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 controlling degree of multi-programming
  - 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

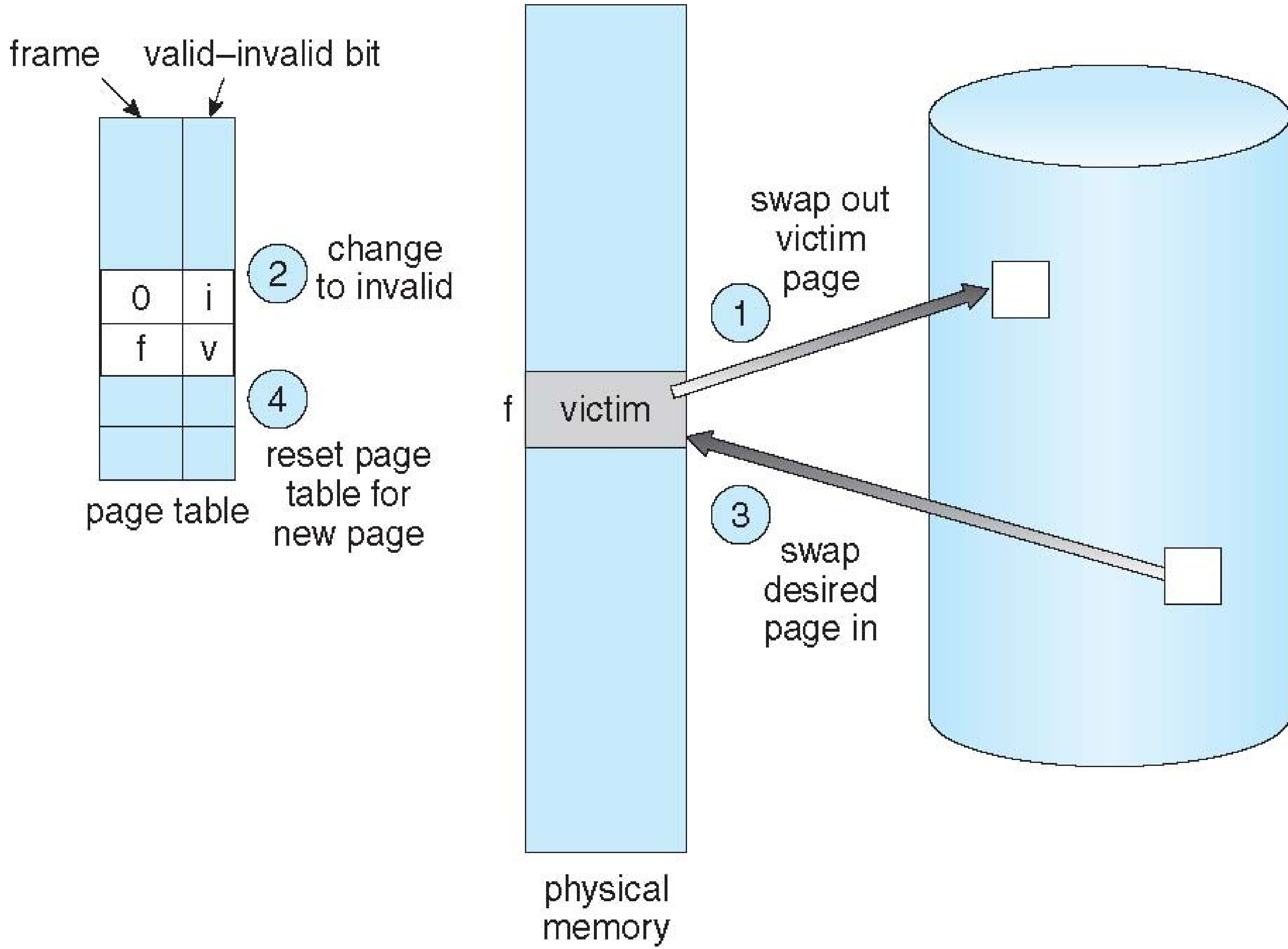


Basic Page Replacement

1. Find Location of 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 Desired Page into (newly) free frame; update page and frame tables
4. Continue Process by Restarting Instruction that caused trap

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

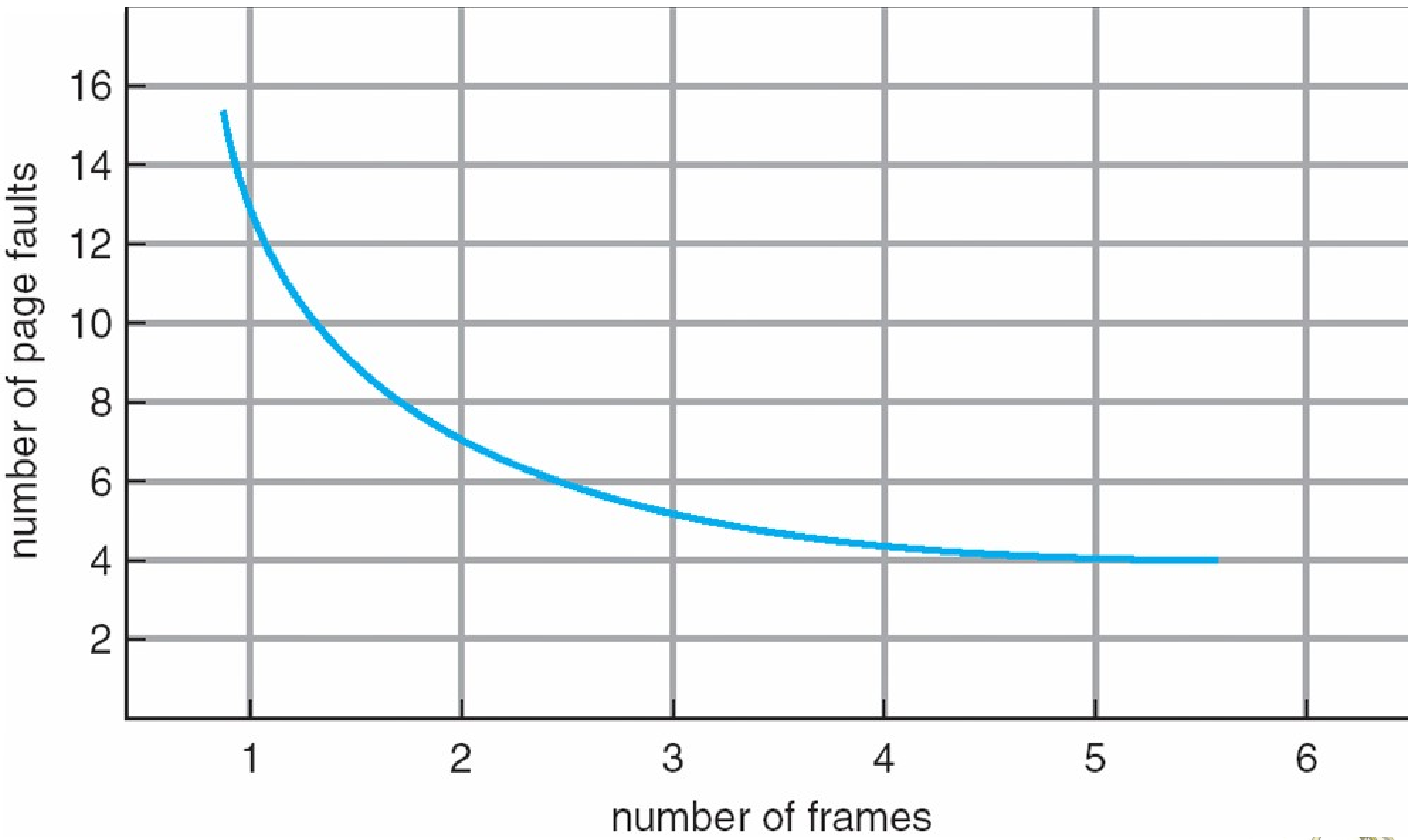
Page Replacement



Page and Frame Replacement Alg.

- **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 number of page faults on that string
  - String is just page numbers, not full addresses
  - Repeated access to same page does not cause a page fault
  - Results depend on number of frames available
- In all our examples, **reference string** of referenced page numbers 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 No of Frames

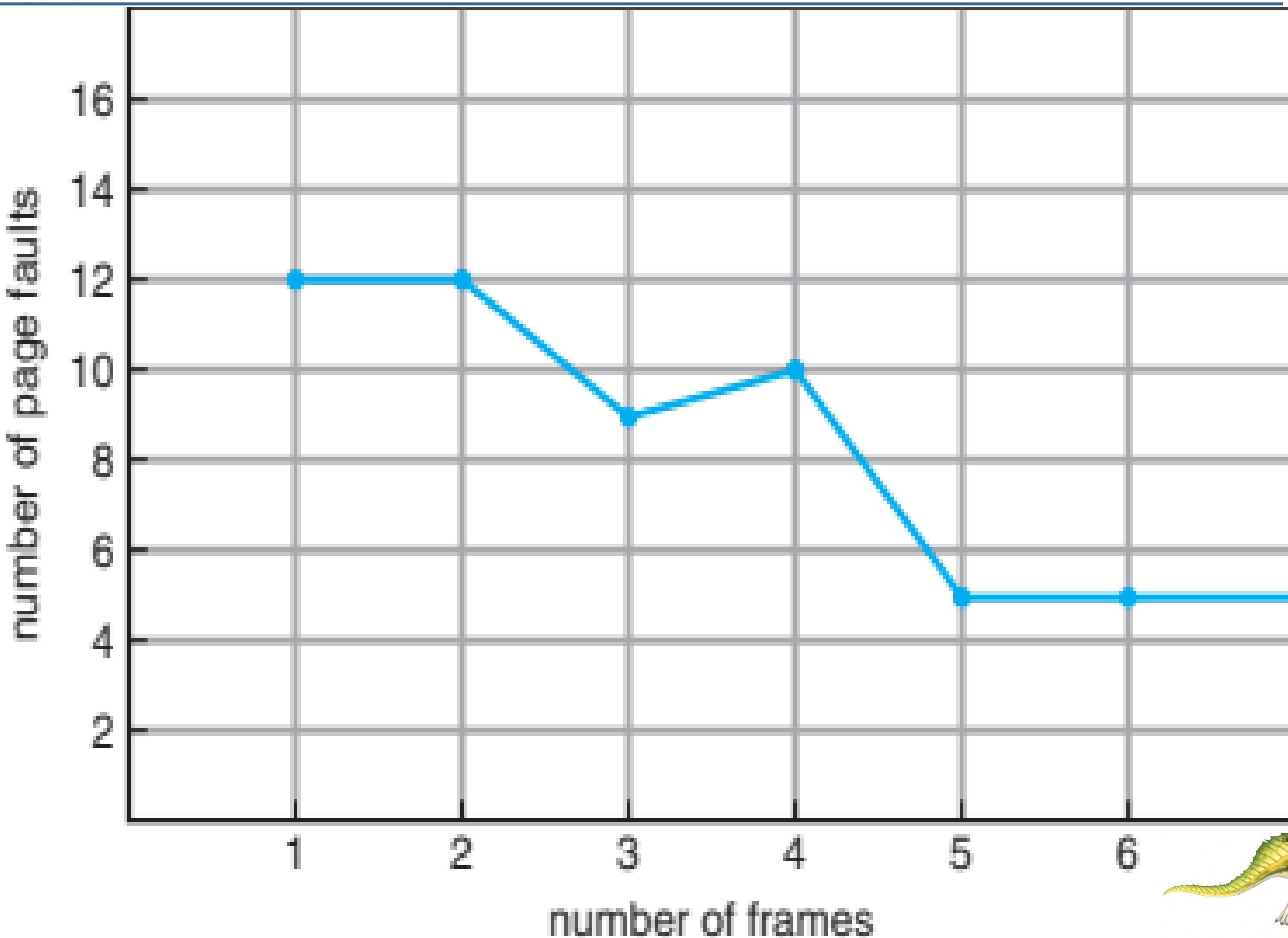


First-In-First-Out (FIFO) Algorithm

- 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)
- 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

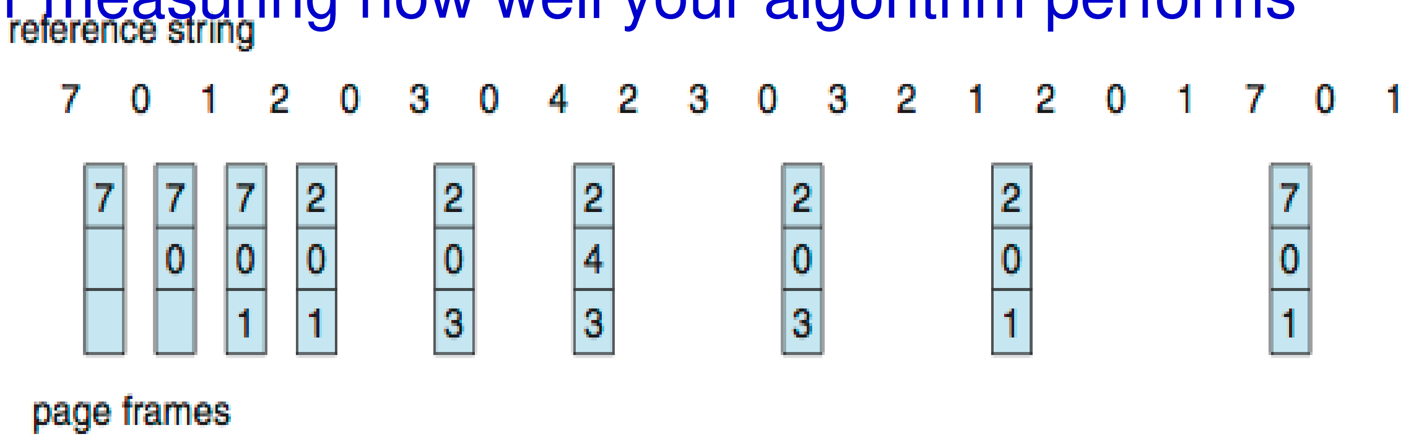
15 page faults

FIFO Illustrating Belady's Anomaly



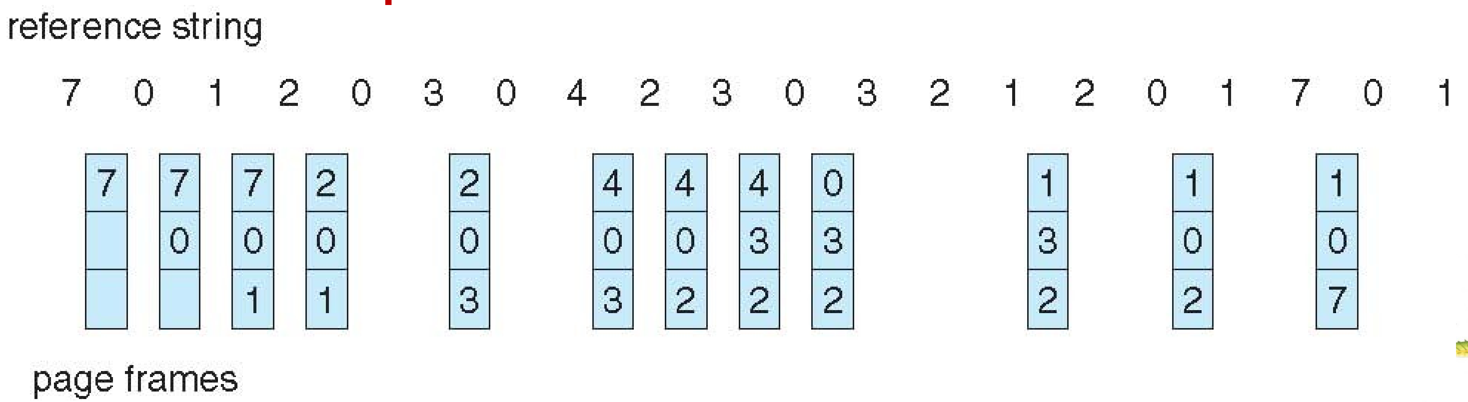
Optimal Algorithm

- Replace Page that will not be used for Longest Period of Time (in future)
  - 9 is optimal for this example
- How do you know this?
  - Can’t read the future
- Used just as a Reference Model
  - For measuring how well your algorithm performs



Least Recently Used (LRU) Algorithm

- Use Past Knowledge rather than Future
- Replace Page that has not been Used in Most Amount of Time
- Associate Time of Last Use with each page
- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- But How to Implement?



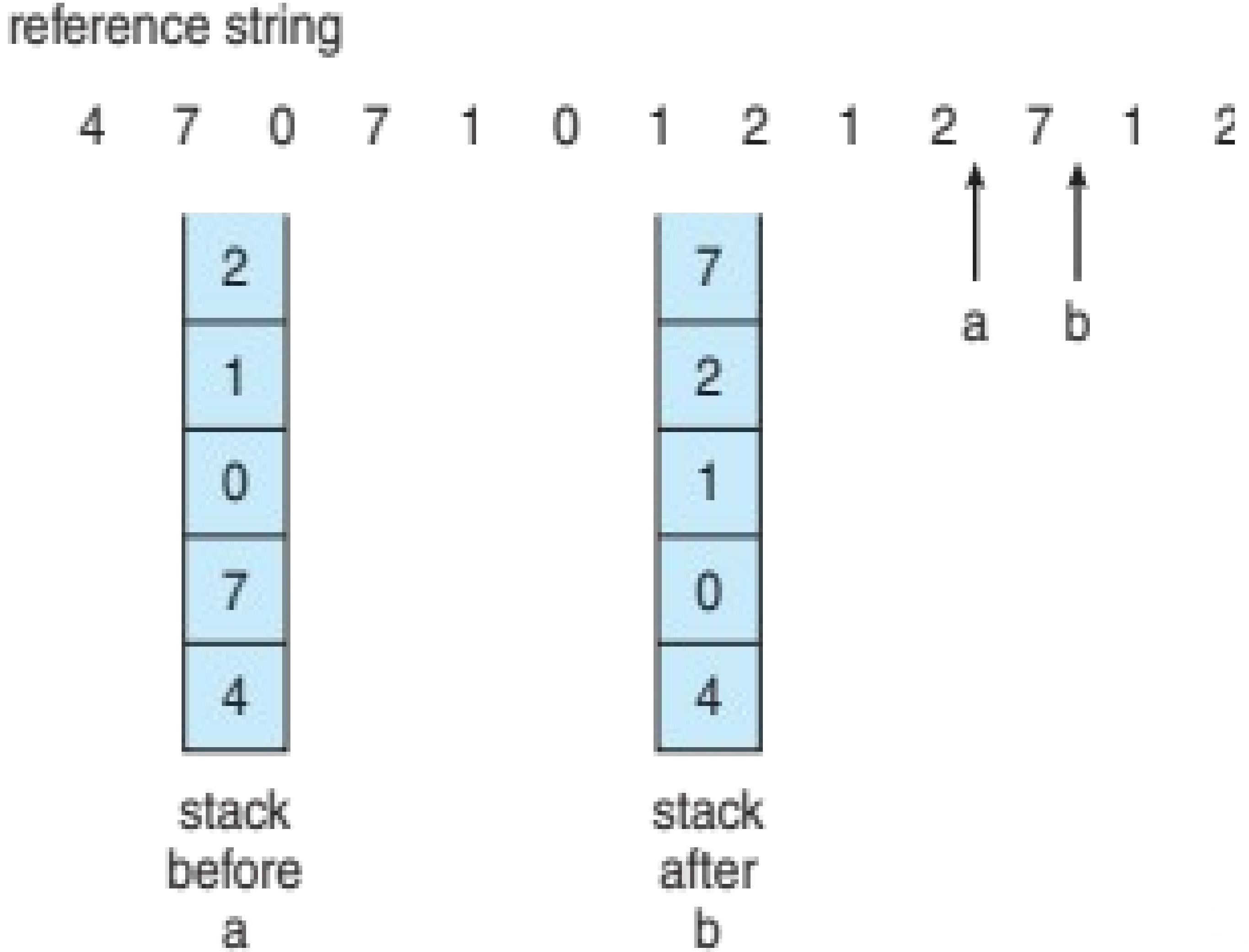
LRU Algorithm (cont.)

- 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
  - **Cons**
    - Write to memory on every memory access ☹
    - Needs to look into all counters for smallest value ☹
    - Counter overflow is an issue ☹

LRU Algorithm (cont.)

- Stack Implementation
  - Keep stack of page numbers in a double link-list
  - Page referenced:
    - Move it to top
    - Requires up to 6 pointers to be changed
  - But each update more expensive
- Pros
  - No search for replacement ☺

Use Of a Stack to Record Most Recent Page References



LRU Algorithm (cont.)

- Stack Algorithms
  - An algorithm for which it can be shown that set of pages in memory for *n* frames is always a ***subset*** of set of pages that would be in memory with *n*+1 frames
  - Informally: Don't have Belady's anomaly
- Example of Stack Algorithms
  - LRU
  - OPT

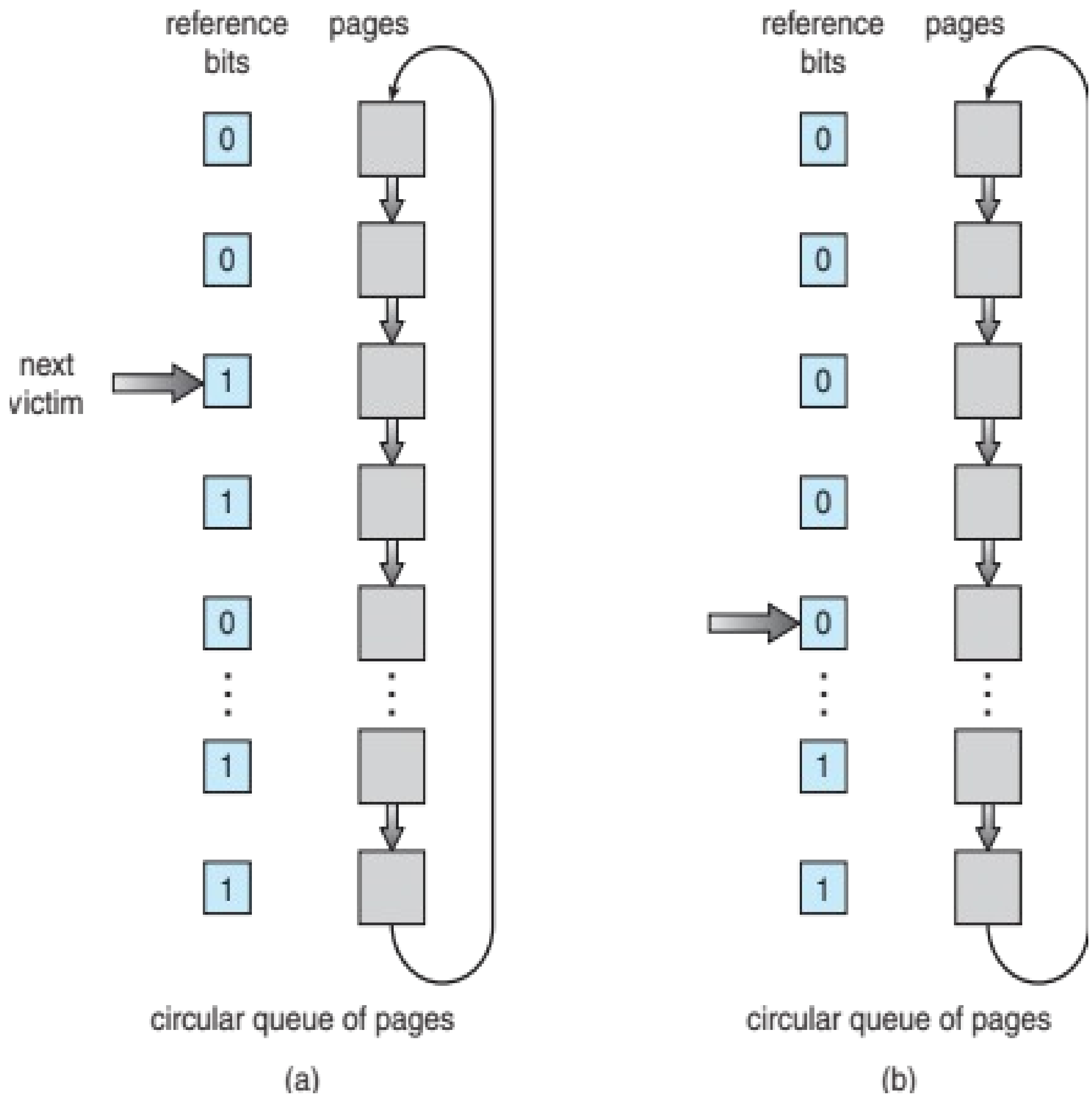
LRU Approximation Algorithms

- LRU needs Special HW 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
- **Additional-Reference-Bits Algorithm**
  - Use multiple history bits
  - OS shifts reference bits at intervals
  - Example: consider 8 reference bits
    - 00000000 ➔ page has not been referenced in non of intervals
    - 11111111 ➔ page has been referenced in eight time intervals
    - 11000000 ➔ ?

LRU Approximation Algorithms

- **Second-Chance Algorithm**
  - Generally FIFO, plus HW-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



Enhanced Second-Chance Algorithm

- Using Reference Bit + Modify Bit
  - Take ordered pair (reference, modify)

1. (0, 0) neither recently used not modified (best candid)
2. (0, 1) not recently used but modified – not quite as good, must write out before replacement
3. (1, 0) recently used but clean – probably will be used again soon
4. (1, 1) recently used and modified – probably will be used again soon and need to write out before replacement

- Use Same Scheme as Clock
  - First search in class “1”, then “2” and “3” and lastly “4”
  - Favors modified pages to reduce number of I/Os
  - Might need to search circular queue several times

Counting Algorithms

- Keep a counter of number of references that have been made to each page
- **Least Frequently Used (LFU) Algorithm**
  - Replaces page with smallest count
  - Typically counter shifted to right at regular intervals to avoid accumulated references
- **Most Frequently Used (MFU) Algorithm**
  - Based on argument that page with smallest count probably just brought in and has yet to be used
- Issues with LFU & MFU → Not common
  - Expensive to implement
  - Do not approximate OPT replacement well

Page-Buffering Algorithms: Enhanced Techniques

- Keep a Pool of Free Frames
  - 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
- Keep List of Modified Pages
  - When backing store otherwise idle, write pages there and set to non-dirty ➔ faster eviction
- 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 (e.g., 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
- OS can Give Direct Access to Disk
  - Array called **raw disk** mode
  - I/O accesses to raw disk called raw I/O
  - Bypasses all file-system services such as demand paging, buffering, locking, and prefetching

Allocation of Frames

- Important Question
  - How to allocate fixed amount of free memory among various processes?
- Main Constraints
  - Cannot allocate more than total # of available frames
    - Unless there is page sharing
  - Must allocate minimum # of frames. Why?
    - Significant page fault rate ➔ degraded performance

Allocation of Frames

- Facts
  - # of allocated frames for a process decreases ➔ page fault rate increases
  - Upon a page fault ➔ instruction must be restarted
  - ➔ Must have enough frames to hold all different pages that any single instruction can reference
- 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*

Allocation of Frames

- Worst Case Scenario
  - When an ISA allows multiple levels of indirection
  - E.g., 100 levels of indirection ➔ would need 101 pages in physical memory
  - Solution: put a limit on maximum # of indirection
- Minimum # of Frames per Process
  - Depends on ISA and locality of references
- Maximum # of Frames per Process
  - Available physical memory & # of processes in system
- Two Major Allocation Schemes
  - Fixed allocation & priority allocation

Fixed Allocation

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

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

$$m = 64$$
$$s_1 = 10$$
$$s_2 = 127$$
$$a_1 = \frac{10}{137} \times 62 \approx 4$$
$$a_2 = \frac{127}{137} \times 62 \approx 57$$

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 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

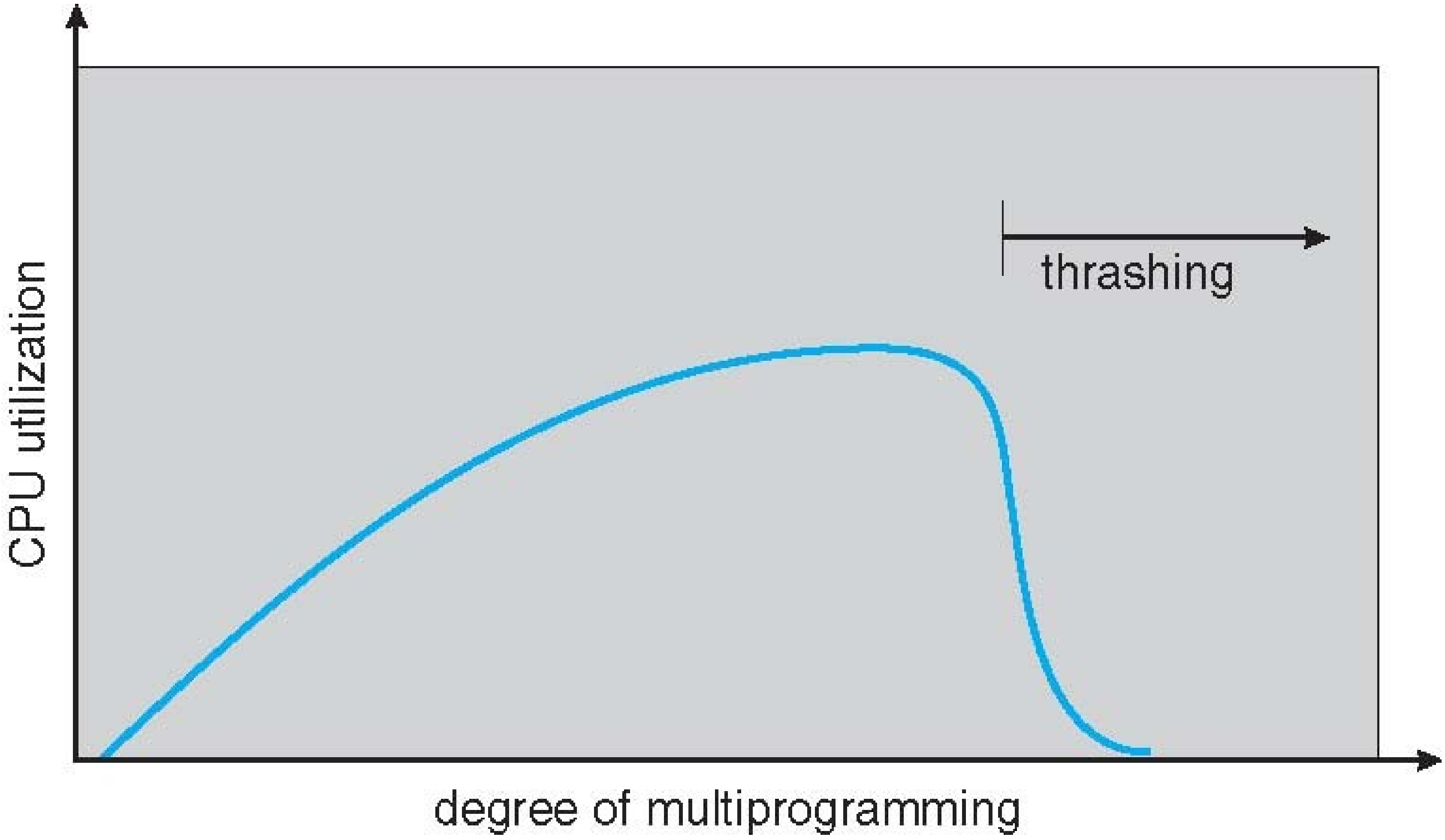
Non-Uniform Memory Access

- 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" CPU on which thread scheduled
  - And modifying scheduler to schedule thread on 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, 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
    - OS thinking that it needs to increase degree of multiprogramming
    - Another process added to system
- **Thrashing** $\equiv$ a Process is Busy Swapping Pages In and Out

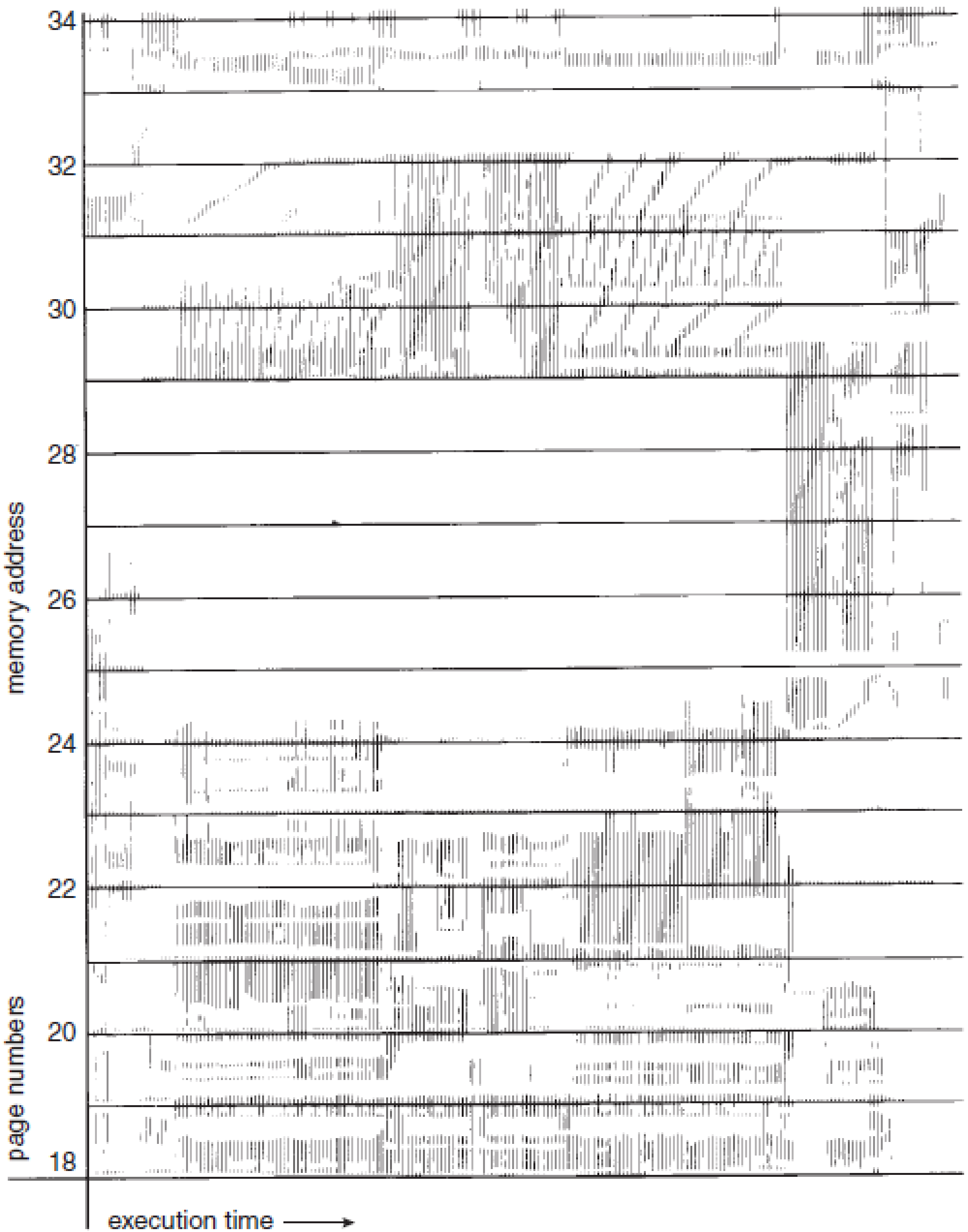
Thrashing (cont.)



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?
- Σ Size of Locality > Total Memory Size
  - Limit effects by using local or priority page replacement

Locality in a Memory-Reference Pattern

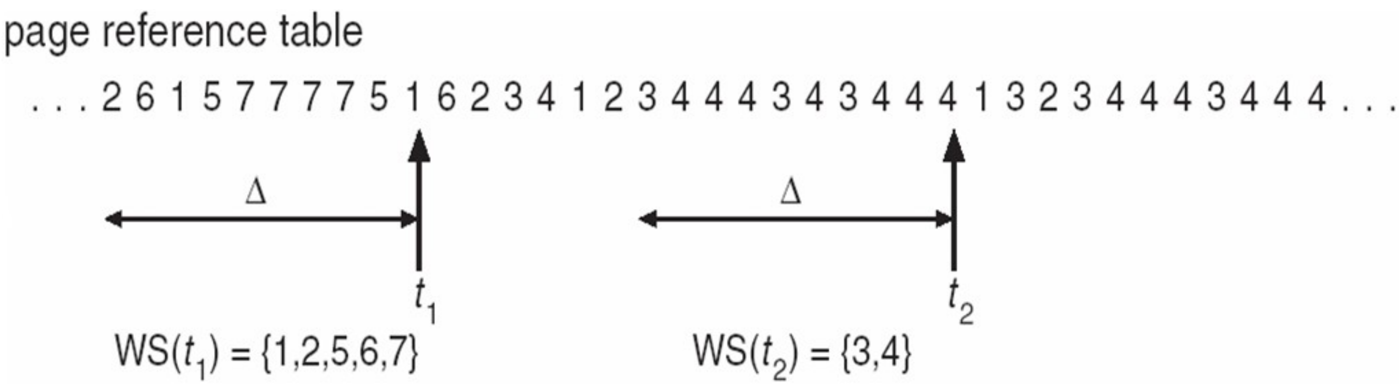


Working-Set Model

- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
- WSS_i (working set of Process P_i) = total number of pages referenced in most recent Δ (varies in time)
  - If Δ too small $\rightarrow$ will not encompass entire locality
  - If Δ too large $\rightarrow$ will encompass several localities
  - If $\Delta = \infty$ $\rightarrow$ will encompass entire program

Working-Set Model (cont.)

- $D = \Sigma\ WSS_i \equiv$ Total Demand Frames
  - Approximation of locality
- if $D > m \Rightarrow$ Thrashing
- Policy: if D > m, then suspend or swap out one of processes

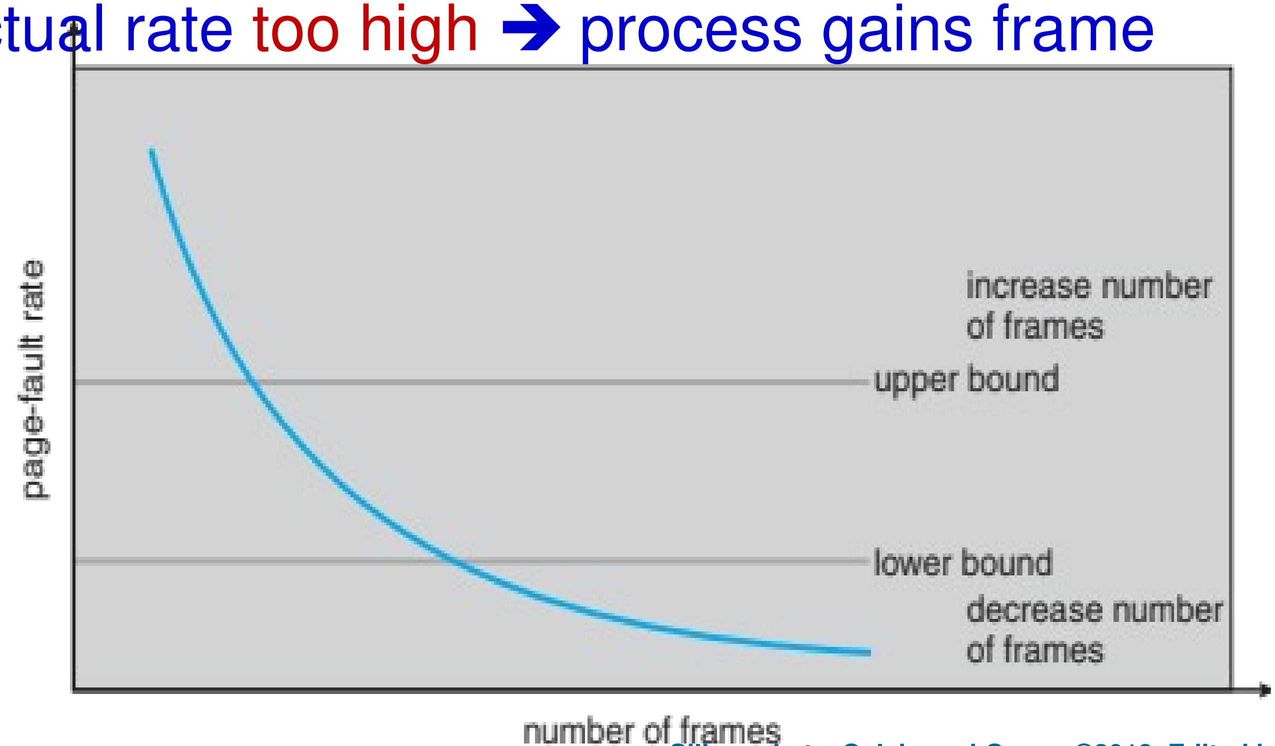


Keeping Track of Working Set

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

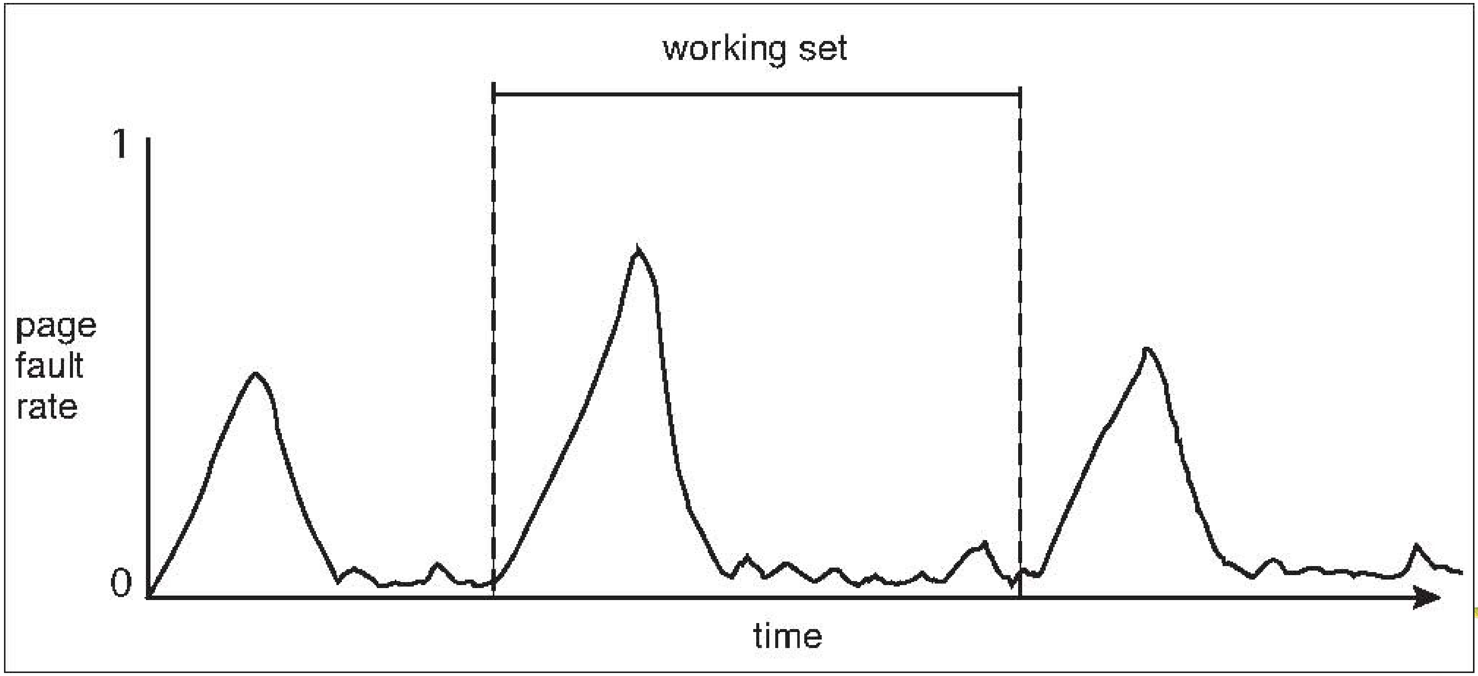
Page-Fault Frequency

- More Direct Approach than WSS
- Establish "acceptable" **Page-Fault Frequency** (**PFF**) 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

- Direct Relationship between Working Set of a Process and its Page-Fault Rate
- Working Set Changes over Time
- Peaks and Valleys over Time



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 file is read from file system into a physical page
  - Subsequent reads/writes to/from 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

Memory-Mapped Files (cont.)

- Also Allows Several Processes to Map same File allowing 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 pager scans for dirty pages

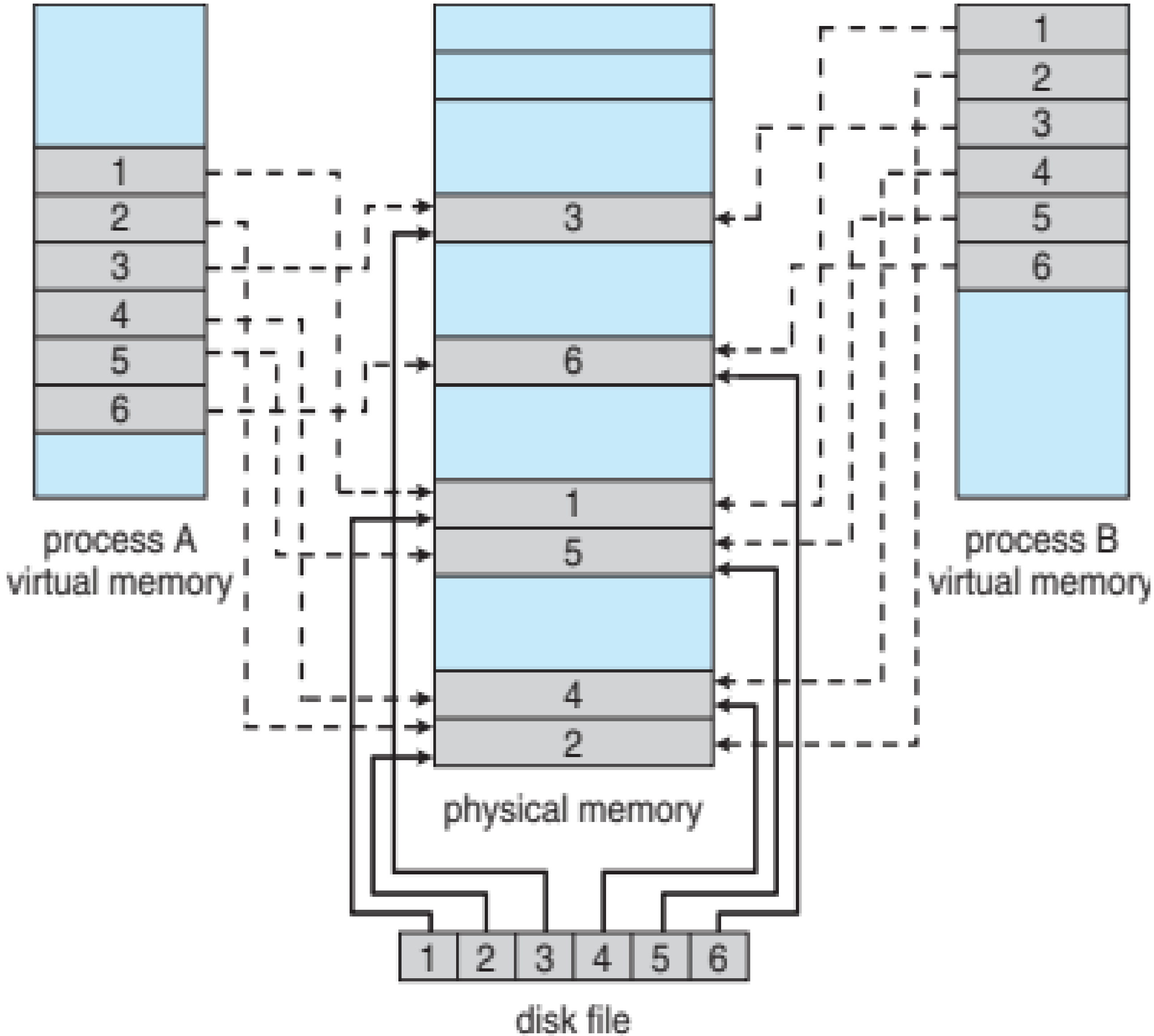
Memory-Mapped File Technique for all I/O

- Some OSes uses 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

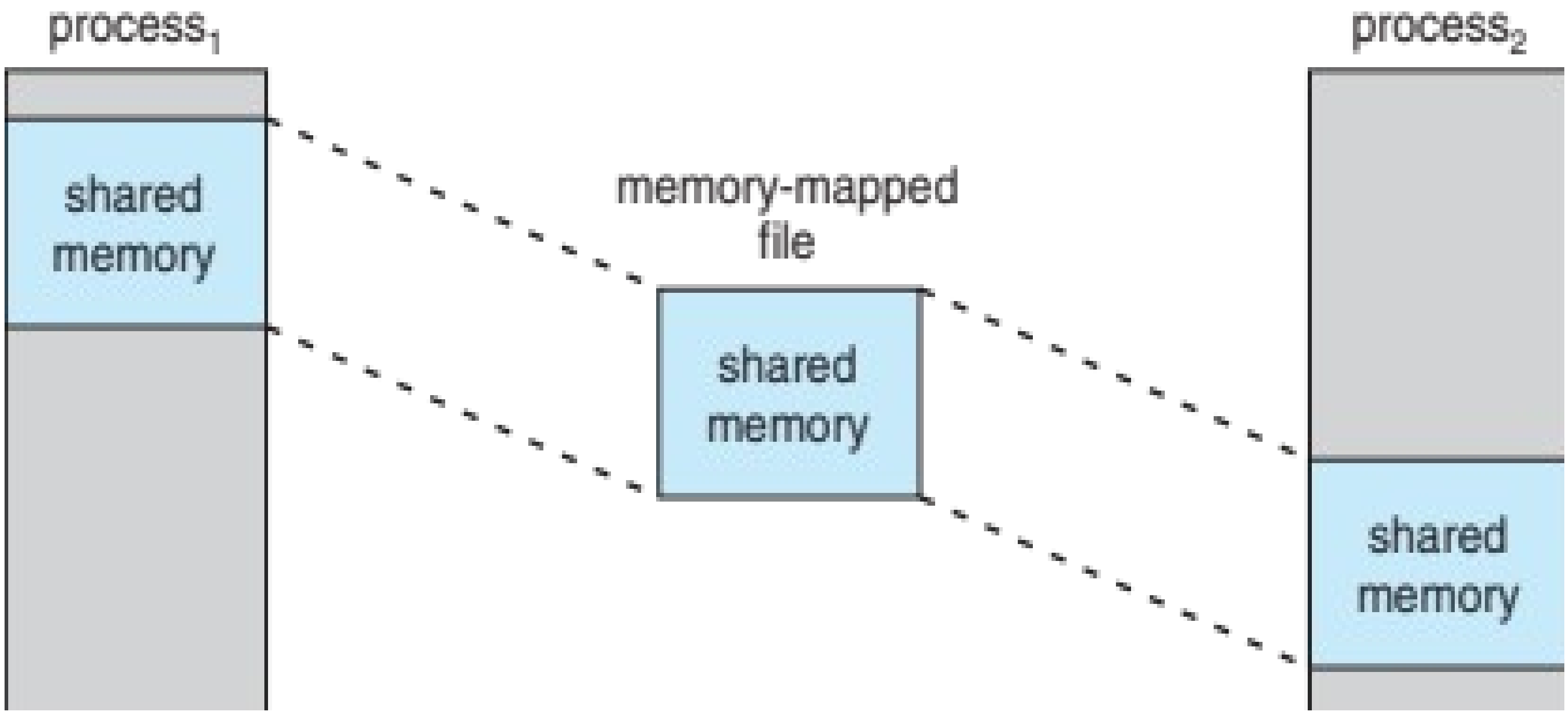
Memory-Mapped File Technique for all I/O

- Solaris: 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



Shared Memory via Memory-Mapped I/O



Allocating Kernel Memory

- 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
    - Structures for device I/O
    - Page tables

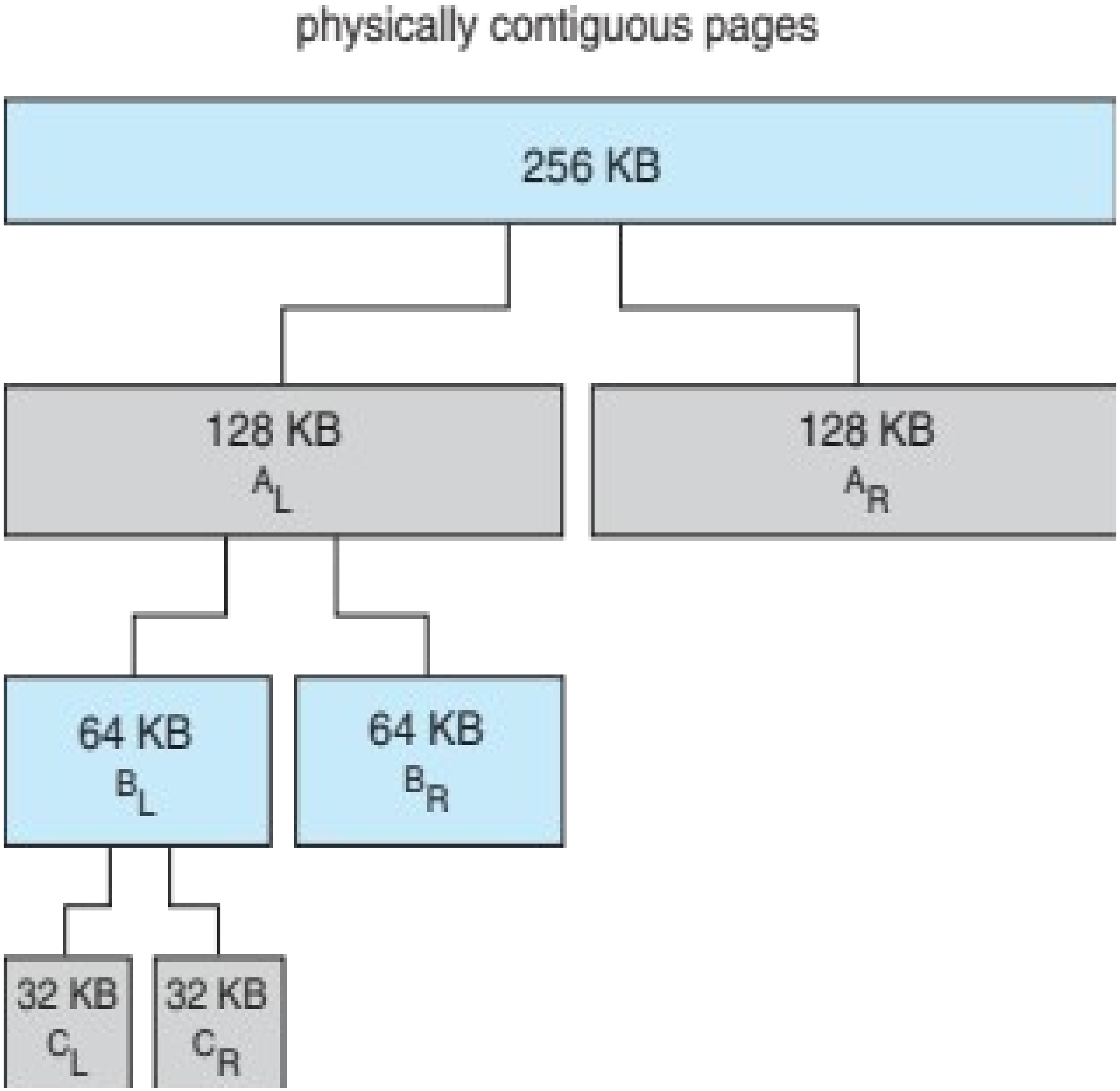
Buddy System

- 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 (cont.)

- For example, assume 256KB chunk available, kernel requests 21KB
  - Split into A_L 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



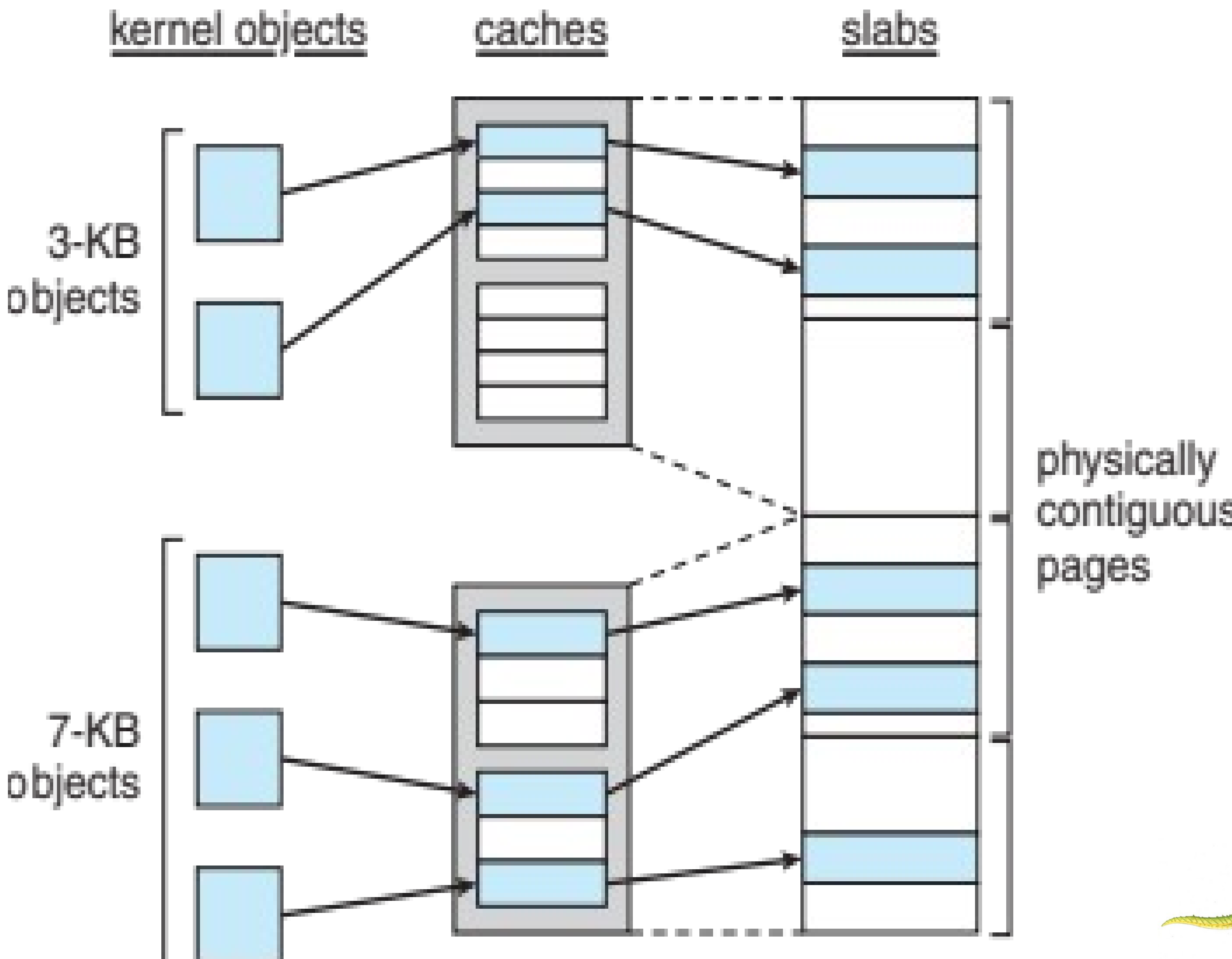
Slab Allocator

- 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
  - A cache for file objects, a cache for semaphores
  - Each cache filled with **objects** – instantiations of data structure
- When cache created, filled with objects marked as **`free`**

Slab Allocator (cont.)

- 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



Further Readings

- Demand Paging Optimizations
- Slab Allocation in Linux
- Operating System Examples
  - Windows
  - Solaris
- Prepaging
- TLB Reach

Does Printf prints Logical or Physical Address?

```c
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>
int main() {
    int my_variable = 1111;
    printf("variable is %d at %p\n", my_variable, &my_variable);
    if (fork() == 0) {
        my_variable = 2222;
        printf("variable in child is %d at %p\n", my_variable, &my_variable);
    } else {
        wait(NULL);
        printf("variable in parent is %d at %p\n", my_variable, &my_variable);
    }
    return 0;
}
```

Slab Allocator in Linux

- For example process descriptor is of type `struct task_struct`
- Approx 1.7KB of memory
- New task -> allocate new struct from cache
  - Will use existing free `struct task_struct`
- Slab can be in three possible states
  1. Full – all used
  2. Empty – all free
  3. Partial – mix of free and used
- Upon request, slab allocator
  1. Uses free struct in partial slab
  2. If none, takes one from empty slab
  3. If no empty slab, create new empty

Slab Allocator in Linux (cont.)

- Slab started in Solaris, now wide-spread for both kernel mode and user memory in various OSes
- Linux 2.2 had SLAB, now has both SLOB and SLUB allocators
  - SLOB for systems with limited memory
    - Simple List of Blocks – maintains 3 list objects for small, medium, large objects
  - SLUB is performance-optimized SLAB removes per-CPU queues, metadata stored in page structure

Shared Memory in Windows API

- First create a **file mapping** for file to be mapped
  - Then establish a view of the mapped file in process's virtual address space
- Consider producer / consumer
  - Producer create shared-memory object using memory mapping features
  - Open file via `CreateFile()`, returning a `HANDLE`
  - Create mapping via `CreateFileMapping()` creating a **named shared-memory object**
  - Create view via `MapViewOfFile()`
- Sample code in Textbook

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 *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 ⇒ 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 range 2^{12} to 2^{22} bytes
- On average, growing over time

Other Issues – TLB Reach

- TLB Reach - amount of memory accessible from TLB
- TLB Reach = (TLB Size) X (Page Size)
- Ideally, working set of each process stored in TLB
  - Otherwise there is a high degree of page faults
- Increase 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 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

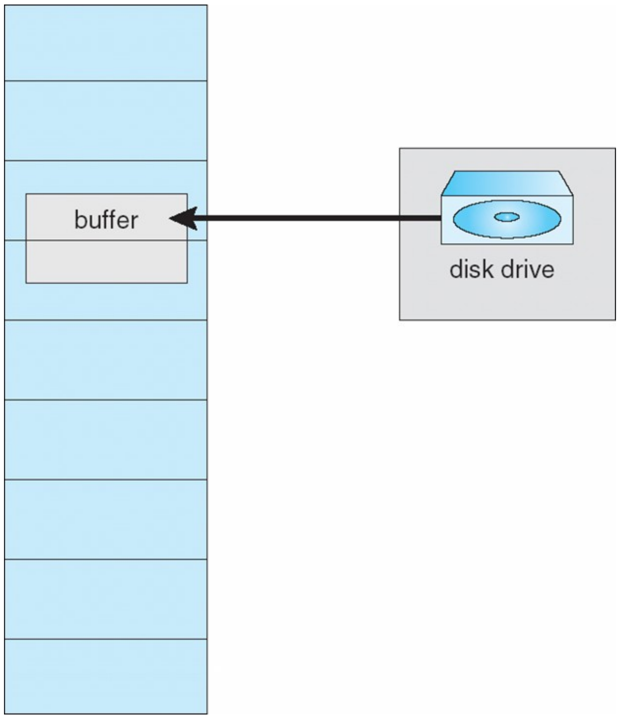
  - Program 2

```
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
- **Pinning** of pages to lock into memory



Demand Paging Optimizations

- Swap space I/O faster than file system I/O even if on the same device
  - Swap allocated in larger chunks, less management needed than file system
- Copy entire process image to swap space at process load time
  - Then page in and out of swap space
  - Used in older BSD Unix

Demand Paging Optimizations (cont.)

- Demand page in from program binary on disk, but discard rather than paging out when freeing frame
  - Used in Solaris and current BSD
  - Still need to write to swap space
    - Pages not associated with a file (like stack and heap) – **anonymous memory**
    - Pages modified in memory but not yet written back to the file system
- Mobile systems
  - Typically don’t support swapping
  - Instead, demand page from file system and reclaim read-only pages (such as code)

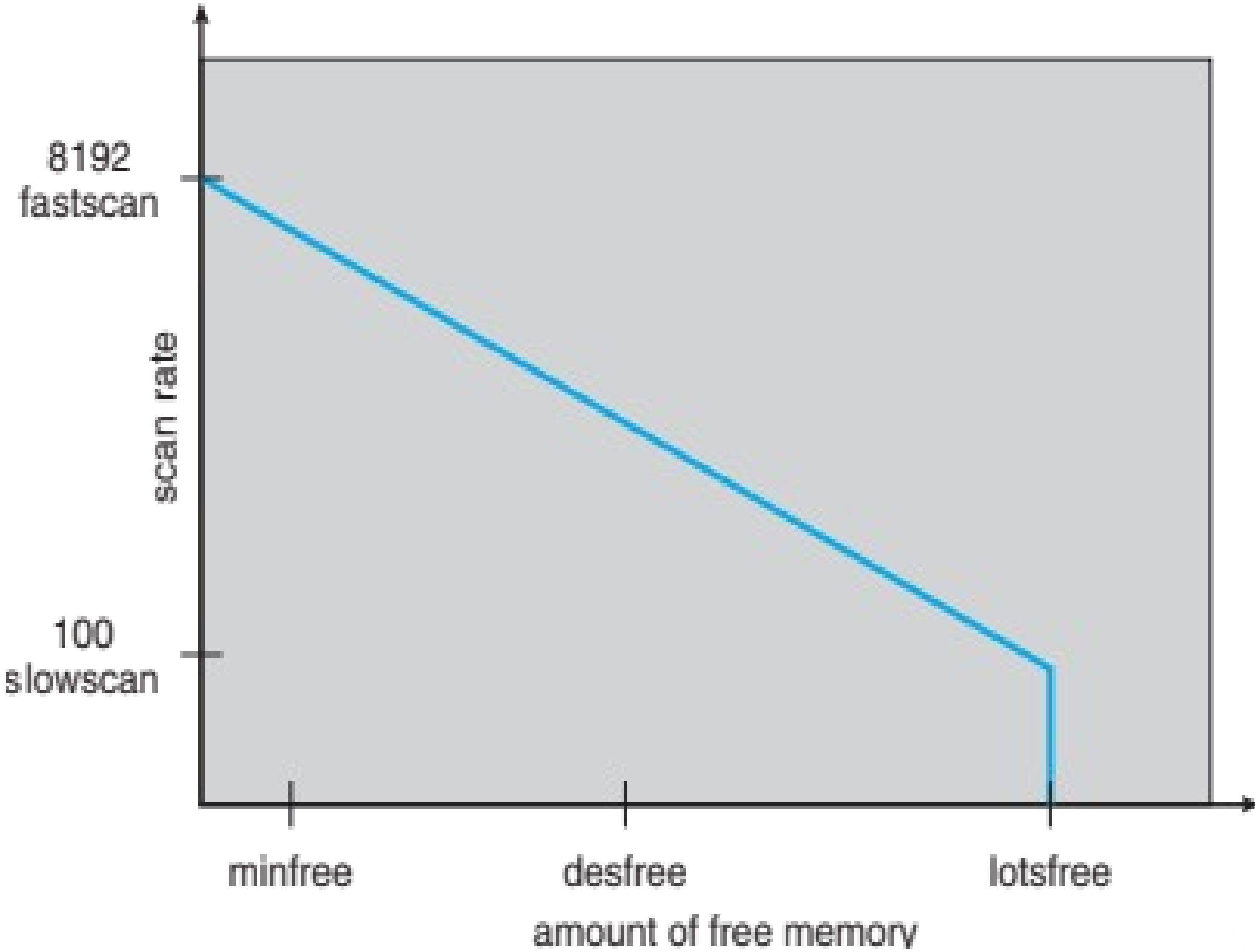
Windows

- 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



End of Lecture 7