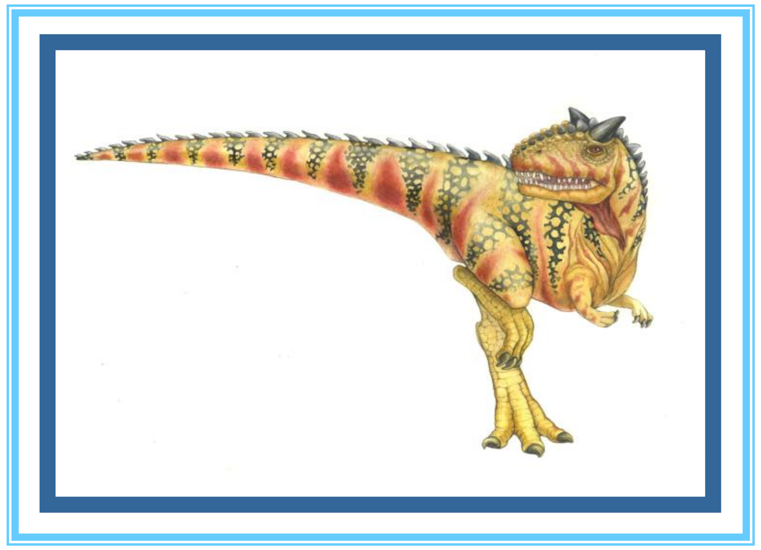
Chapter 9: Virtual-Memory Management



Chapter 9: Virtual-Memory Management

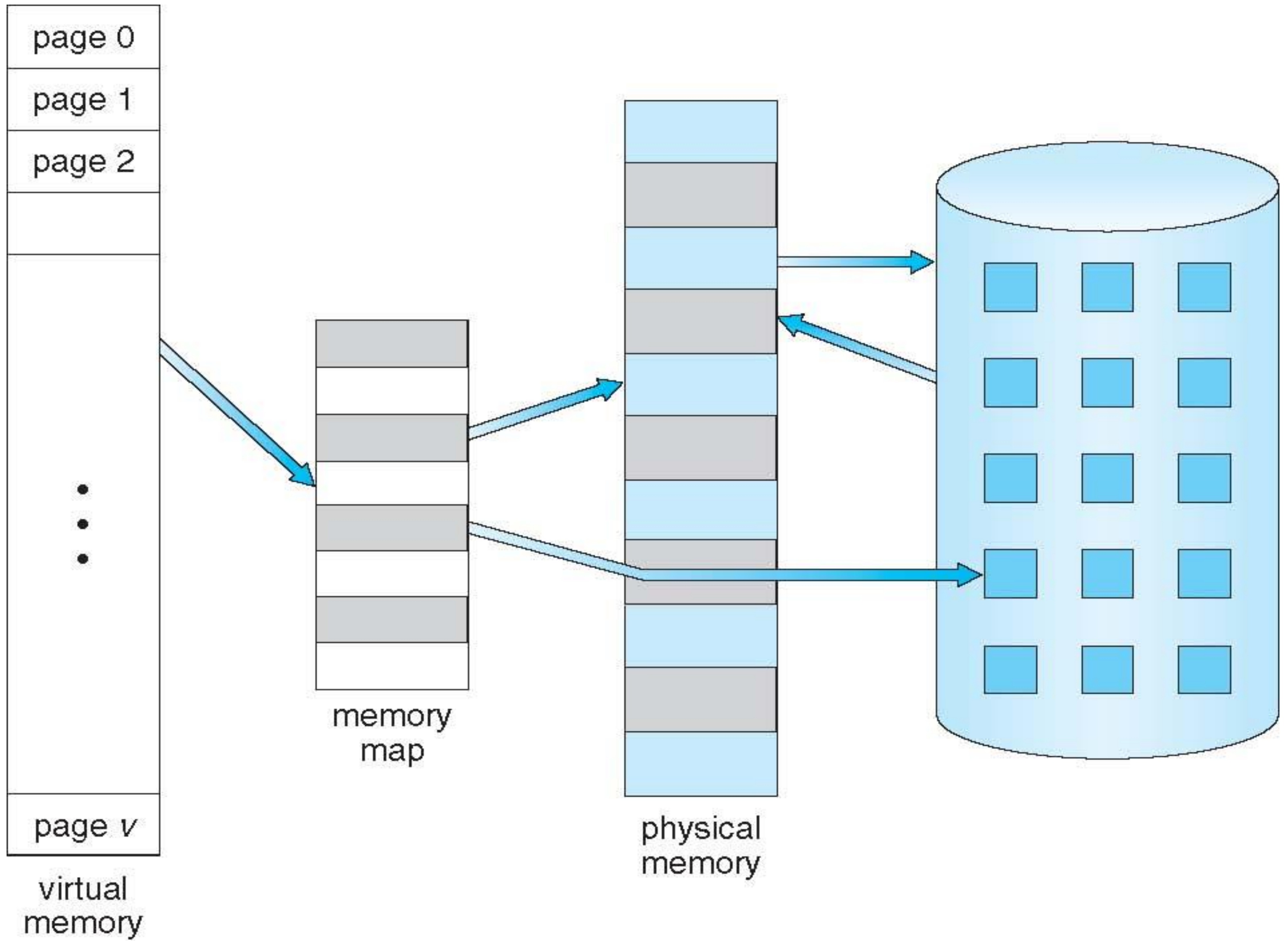
- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped Files
- Allocating Kernel Memory
- Operating-System Examples

- **Objective**
  - 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

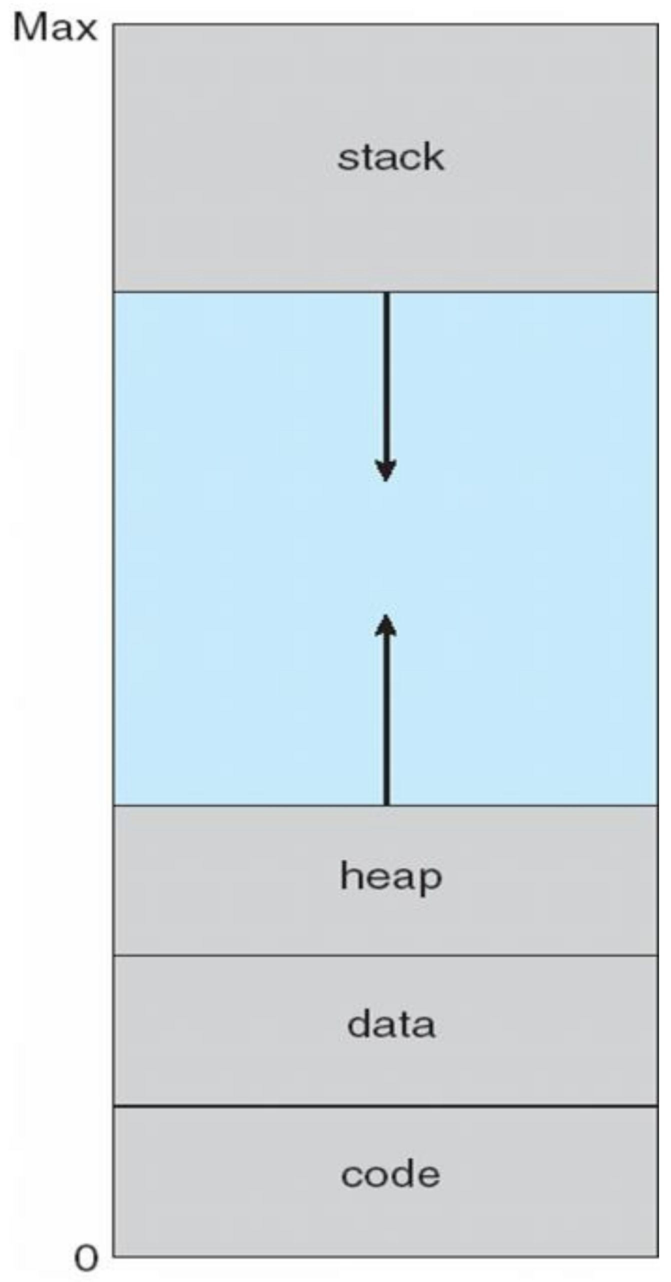
Background

- 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
- **Virtual memory** – separation of user logical memory from physical memory
  - Only part of the program needs to be in memory for execution
  - Allows for more efficient process creation
  - Logical address space can be much larger than physical address space
  - More programs running concurrently
  - Allows address spaces to be shared by several processes

Virtual Memory That is Larger Than Physical Memory

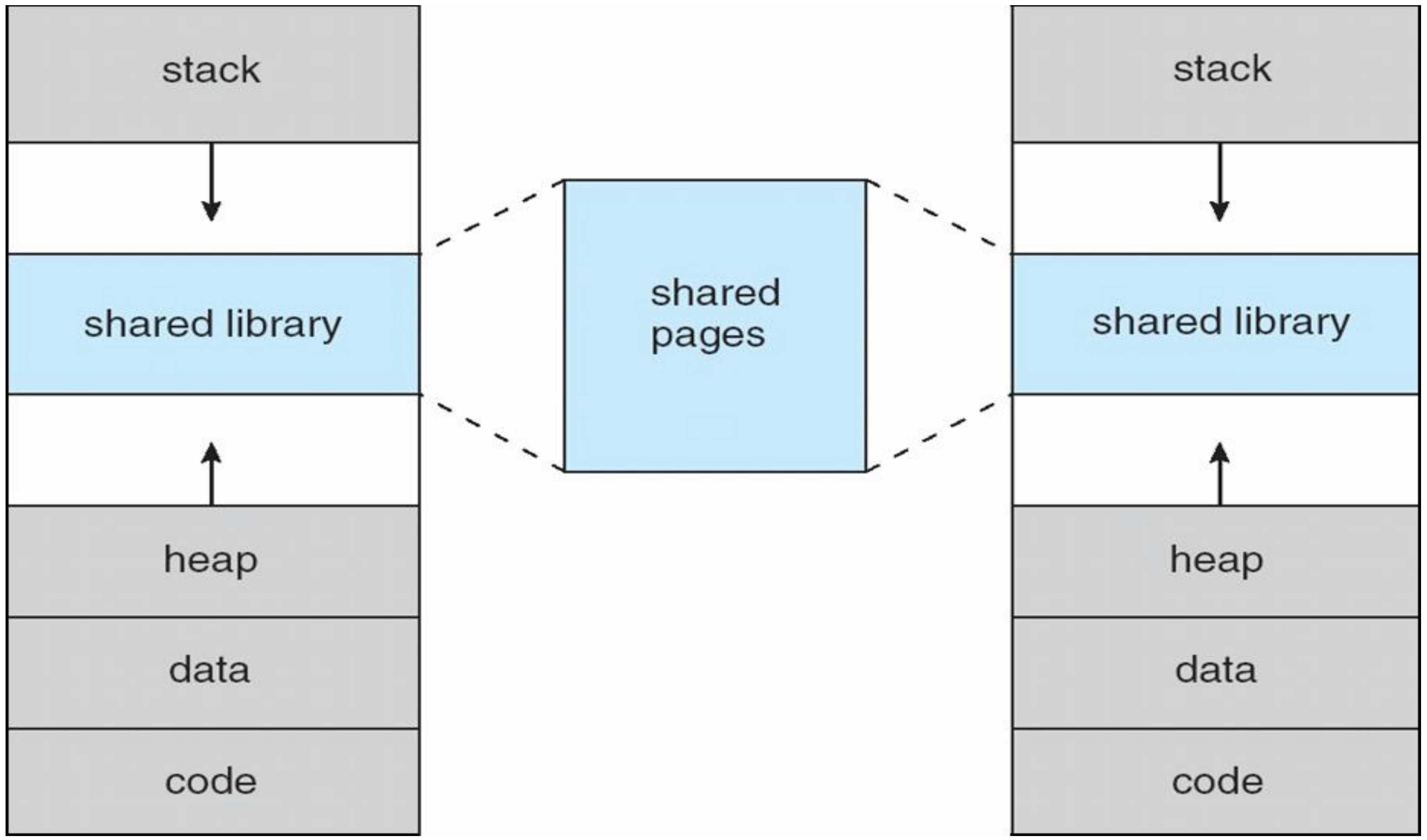


Virtual-address Space



Virtual Address Space

- Enable **sparse** address spaces with holes left for growth, dynamically linked libraries, etc
  - System libraries shared via mapping into virtual address space

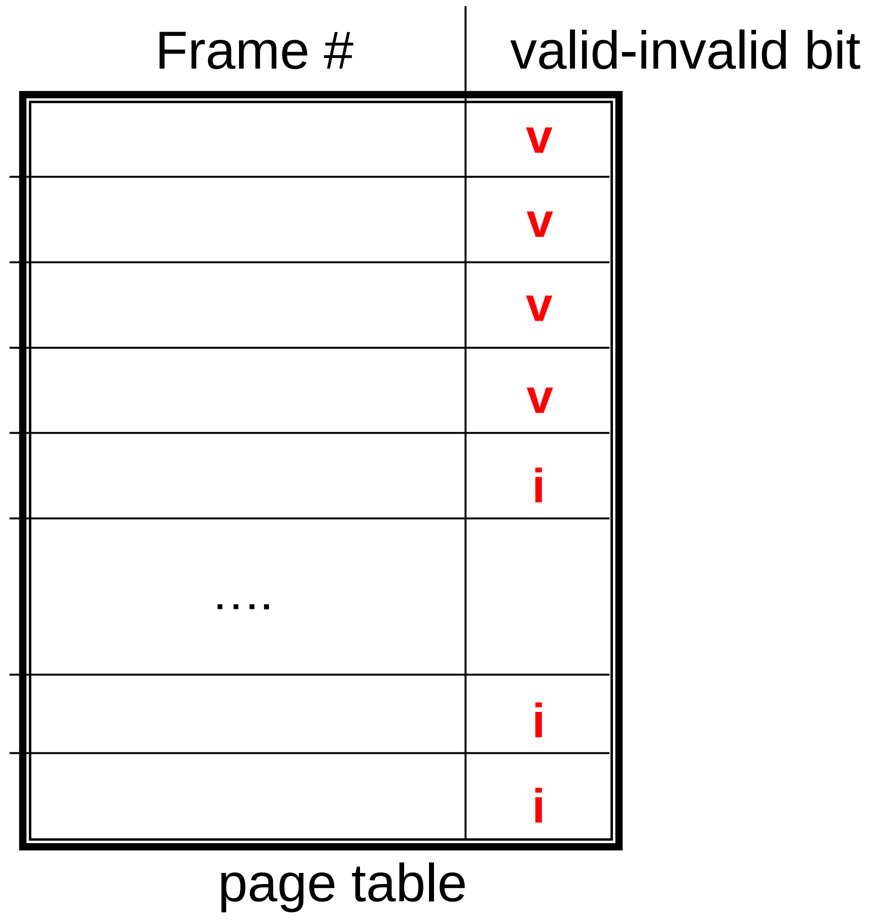


Demand Paging

- To run a process
  - 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 ⇒ 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
  - The code that deals with pages is called **pager**

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:

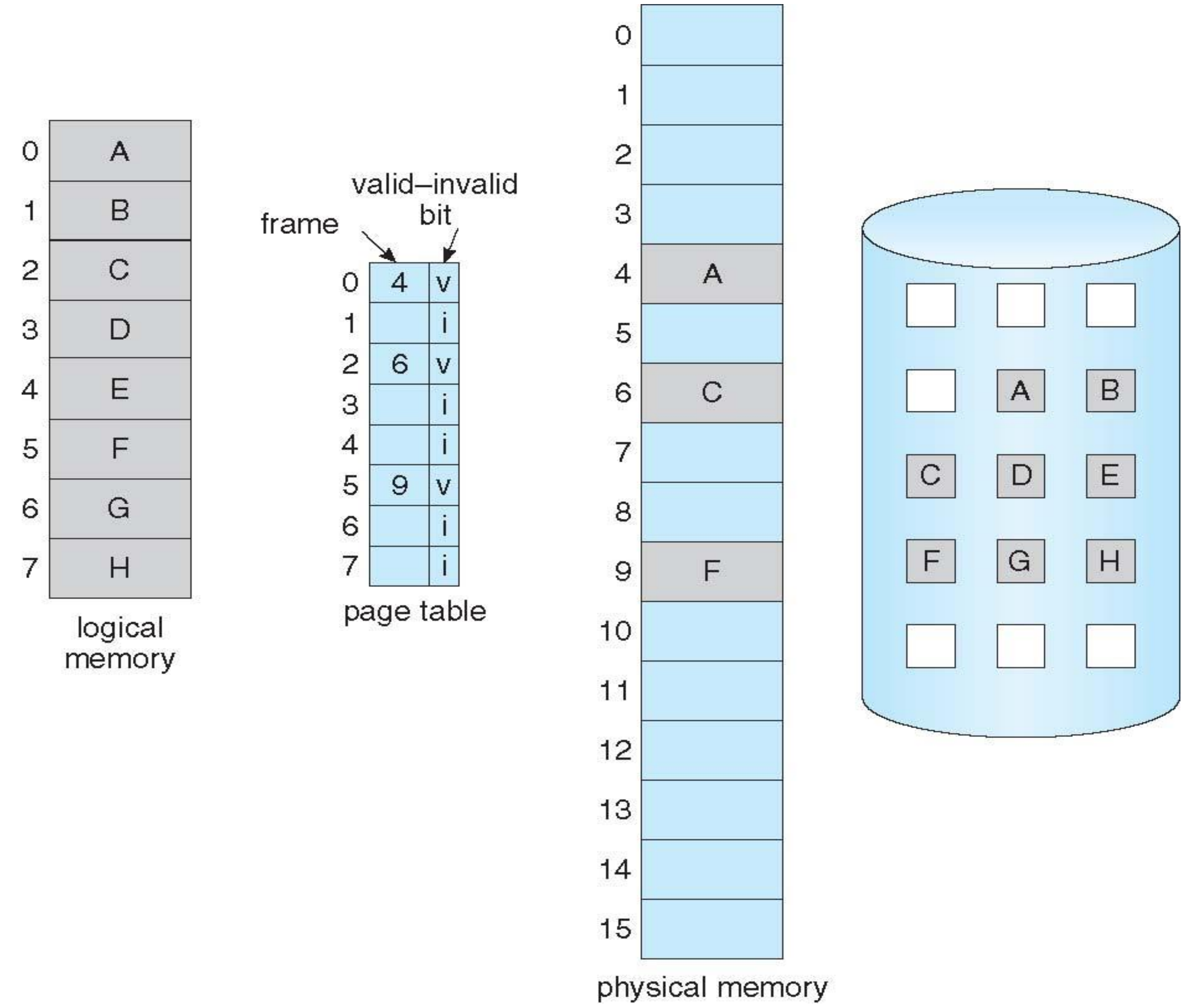


Frame #	valid-invalid bit
	v
	v
	v
	v
	i
….	
	i
	i

page table

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

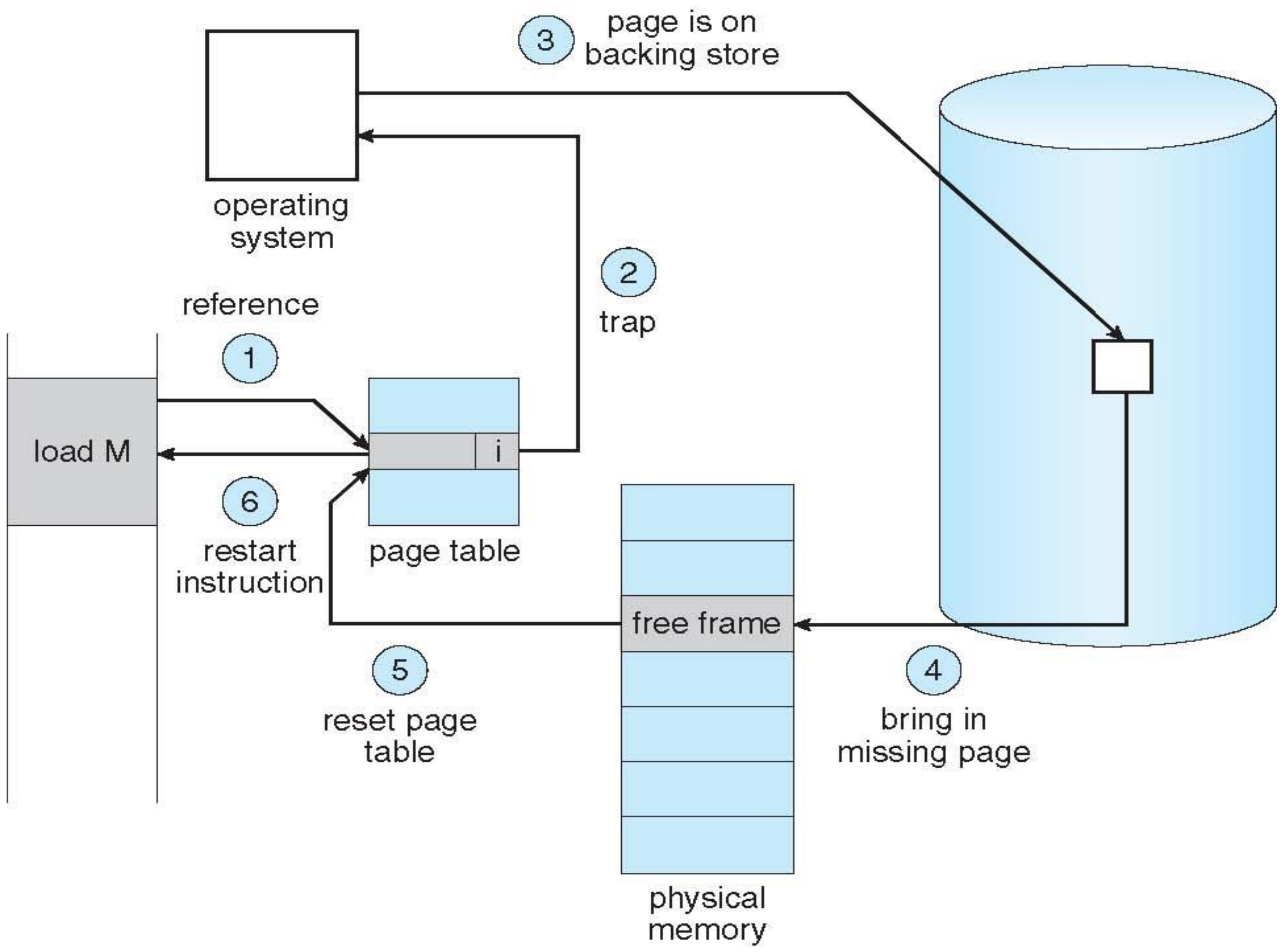
Page Table When Some Pages Are Not in Main Memory



Page Fault

- Access to a page marked invalid causes a **page fault**.
- The paging hardware, in translating the address through the page table, will notice that the invalid bit is set, causing a **trap** to the operating system.
  1. OS checks PCB to see whether the reference is valid or invalid:
     - Invalid reference ⇒ abort
     - Otherwise, invalid bit means "not in memory" → go to step2
  2. Get empty frame
  3. Swap page into frame via scheduled disk operation
  4. Reset tables to indicate page now in memory
     Set validation bit = **v**
  5. Restart the instruction that caused the page fault

Steps in Handling a Page Fault



Pure Demand Paging

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

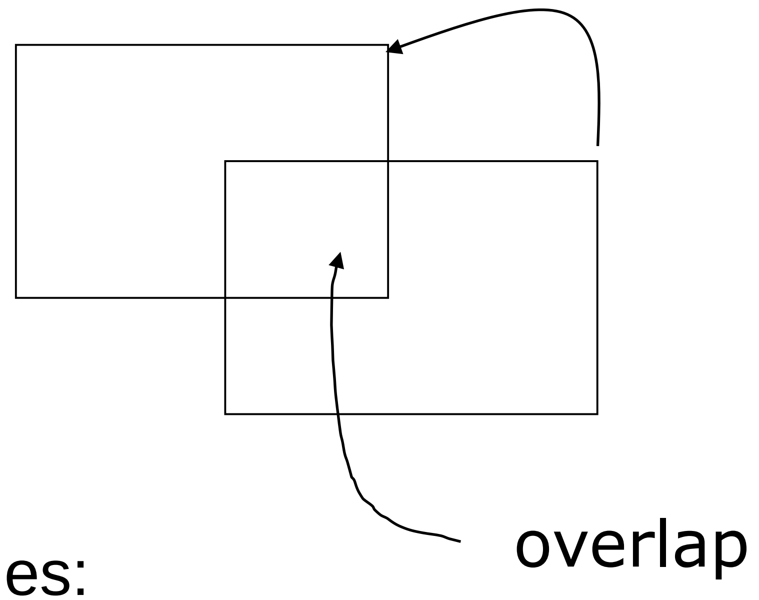
- Actually, a given instruction could access multiple pages -> multiple page faults
  - Pain decreased because of **locality of reference**
- Hardware support needed for demand paging
  - Page table with valid / invalid bit
  - Secondary memory (swap device with **swap space**)
  - Instruction restart

Instruction Restart

- Consider a three-address instruction such as ADD C, A, B
  - Fetch and decode the instruction (ADD)
  - Fetch A
  - Fetch B
  - Add A and B
  - Store the sum in C
- If page fault occurs when store in C, restart instruction includes:
  - fetching instruction again, fetch A again, fetch B again, adding again, …
- Consider an instruction that accesses several locations, such as block move
  - What if source and destination overlap? Restart the whole operation?
  - Two possible solutions:

1. The microcode computes and attempts to access both ends of both blocks. If a page fault is going to occur, it will happen before anything is modified.
2. Uses temporary registers to hold the values of overwritten locations. Restores memory to its state before the instruction was restarted, so the instruction can be repeated if a page fault occurs.

.



Performance of Demand Paging

- 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)**

  EAT = (1 – p) x memory access

  + p page fault time

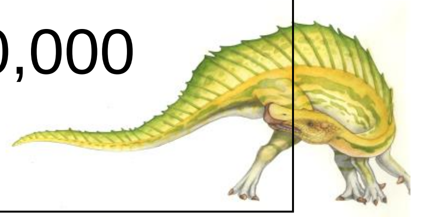
Example

- Memory access time = 200 ns
- Average page-fault service time = 8 ms
- EAT = (1 – p) x 200 + p (8 ms)

  = (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 us

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



Demand Paging Optimizations

- Copy entire file image to swap space at process load time (Disk I/O to swap space is generally faster than that to the file system.) ➔ page in and out of swap space
  - Used in older BSD Unix

Demand pages from the file system

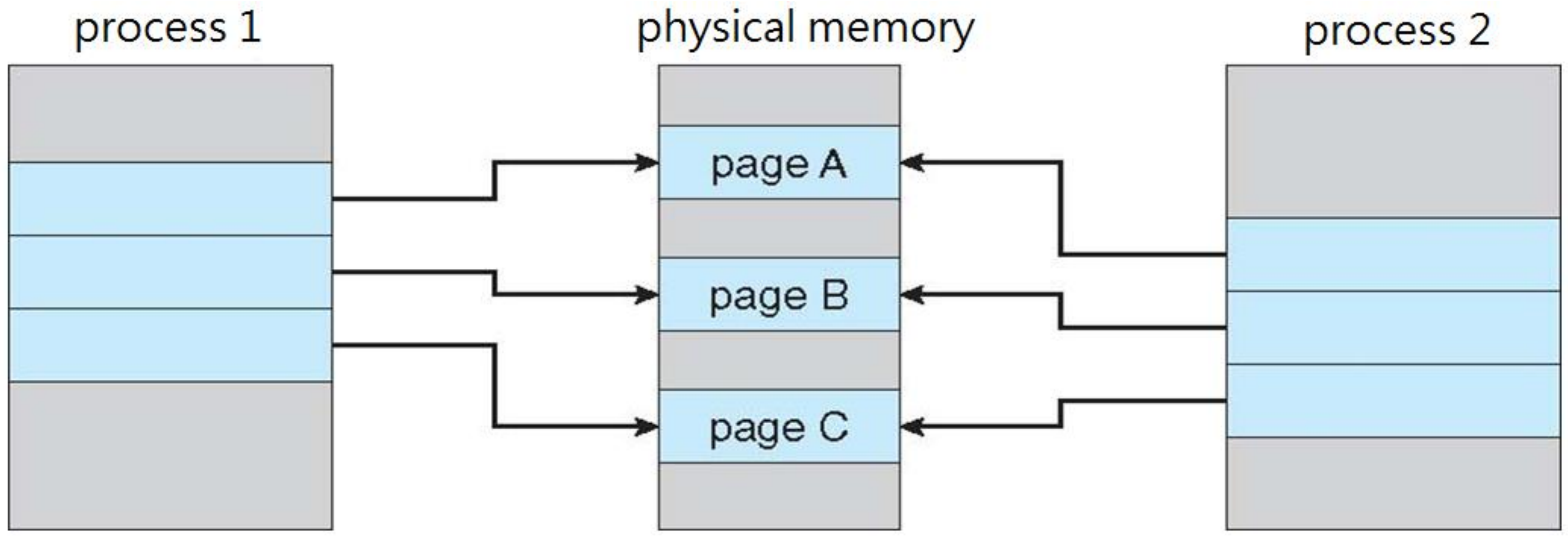
- Alternatively, demand pages from the file system initially but to ***write the pages to swap space*** as they are replaced. (This ensures that only needed pages are read from the file system but all subsequent paging is still from swap space.
- Demand page in from file system on disk, but ***discard rather than paging out*** when freeing frame (to limit the amount of swap space)
  - Swap space still needed for pages not associated with a file (known as **anonymous memory** which includes stack, heap, … for a process).
  - Used in Solaris and current BSD
- Mobile operating systems typically do not support swapping. They demand-page from file system and ***only reclaim read-only pages*** (such as code).
  - In iOS, anonymous memory pages are never reclaimed unless application is terminated or explicitly release the memory.

Copy-on-Write

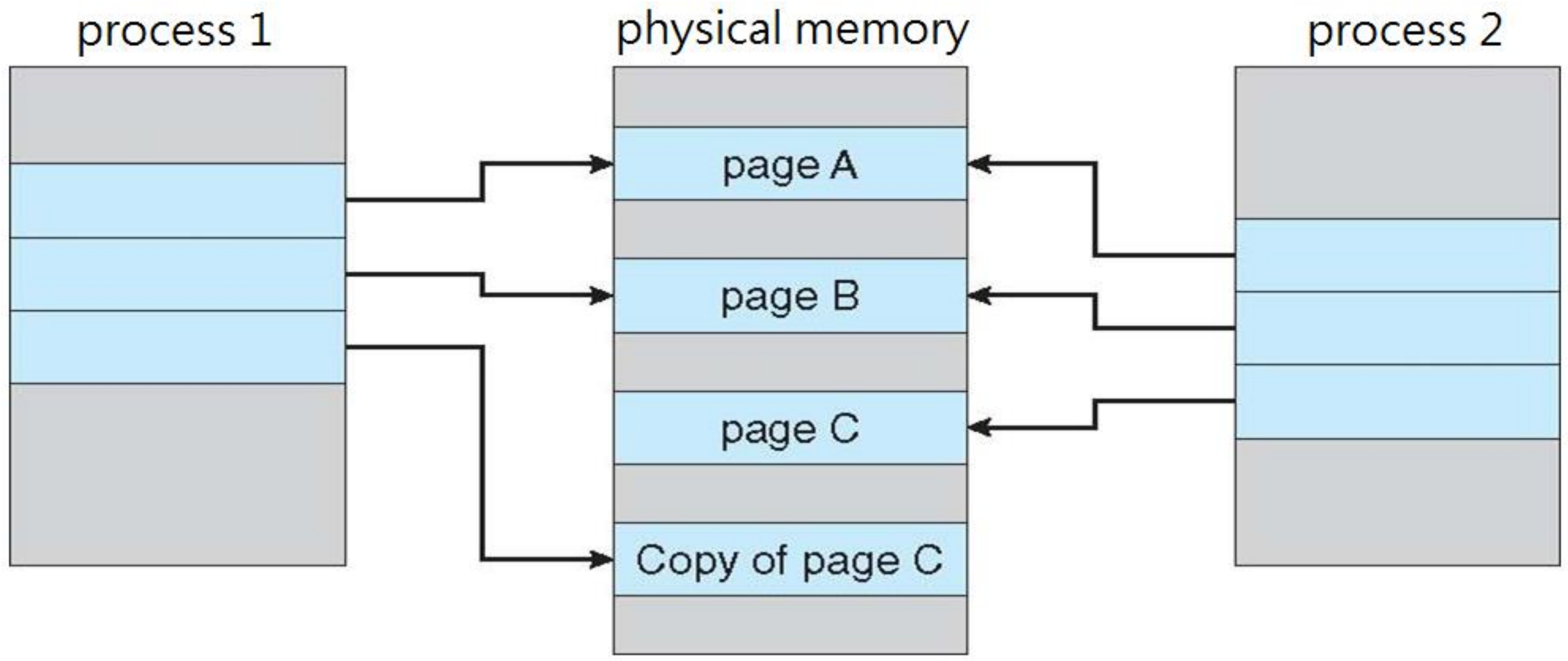
- **Copy-on-Write** (COW) allows both parent and child processes to initially *share* the same pages in memory
  - If either process modifies a shared page, only then is the page copied
- COW allows more efficient process creation as only modified pages are copied
- In general, free pages are allocated from a **pool** of **zero-fill-on-demand** pages
- `vfork()` variation on `fork()` system call has parent suspend and child using the address space of parent
  - vfork() is different from fork() with copy-on-write
  - vfork() does not use copy-on-write ➔ child process is not supposed to modify any page (because all the pages are shared with parents)
  - Designed to have child call `exec()` immediately after creation.
  - Very efficient

Process 1 Modifies Page C

Before



After



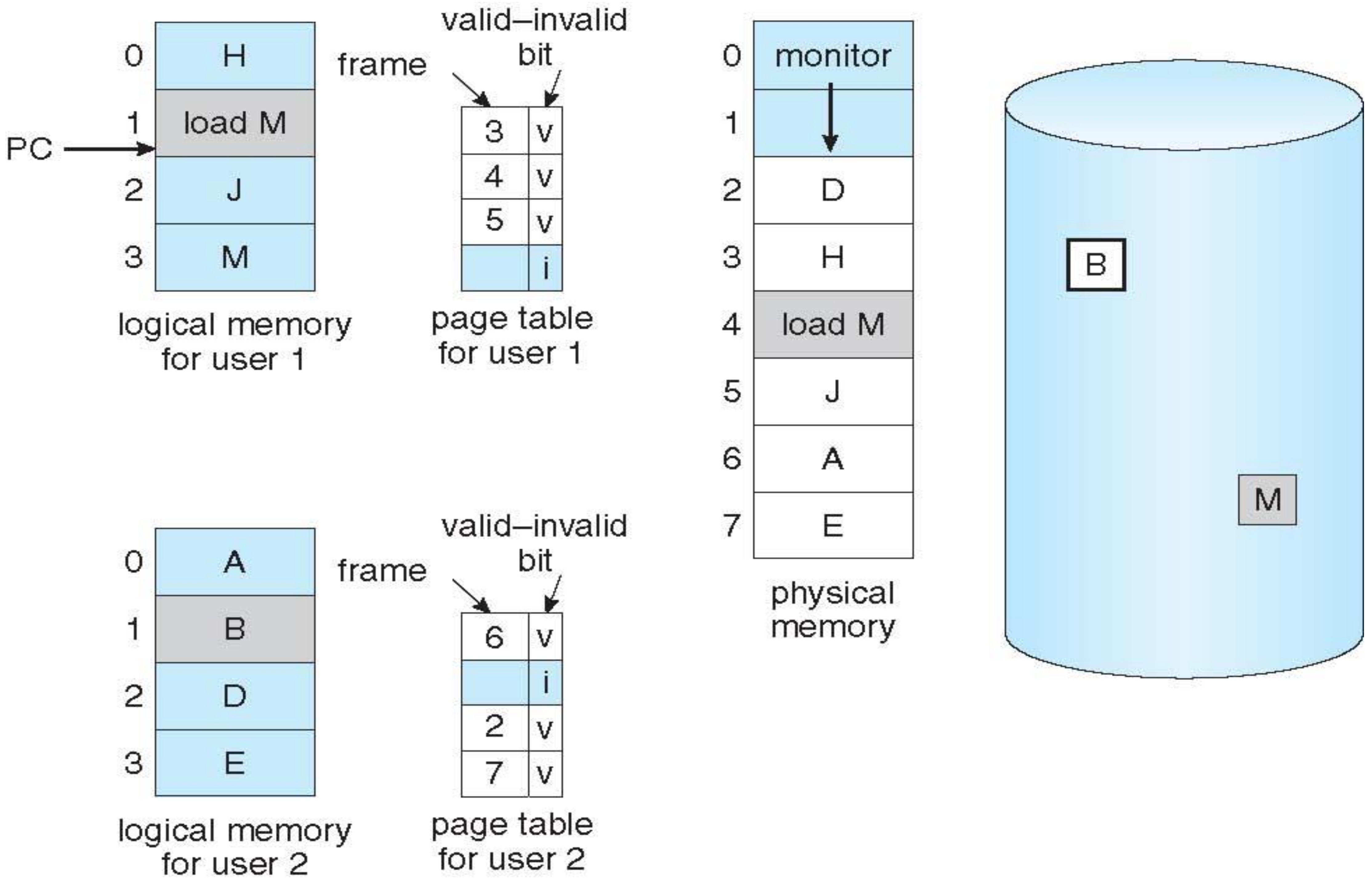
Chapter 9: Virtual-Memory Management

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped Files
- Allocating Kernel Memory
- Operating-System Examples

Page Replacement

- Used up by process pages
  - ➔ terminate? swap out? replace the page?
  - Also in demand from the kernel, I/O buffers, etc
- **Page replacement** – find some page in memory, but not really in use, page it out
  - Same page may be brought out and into memory several times
  - Performance – want an algorithm which will result in ***minimum number of page faults***
- 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 replacement algorithm to select a **victim frame**
   - Write victim frame to disk **if dirty**

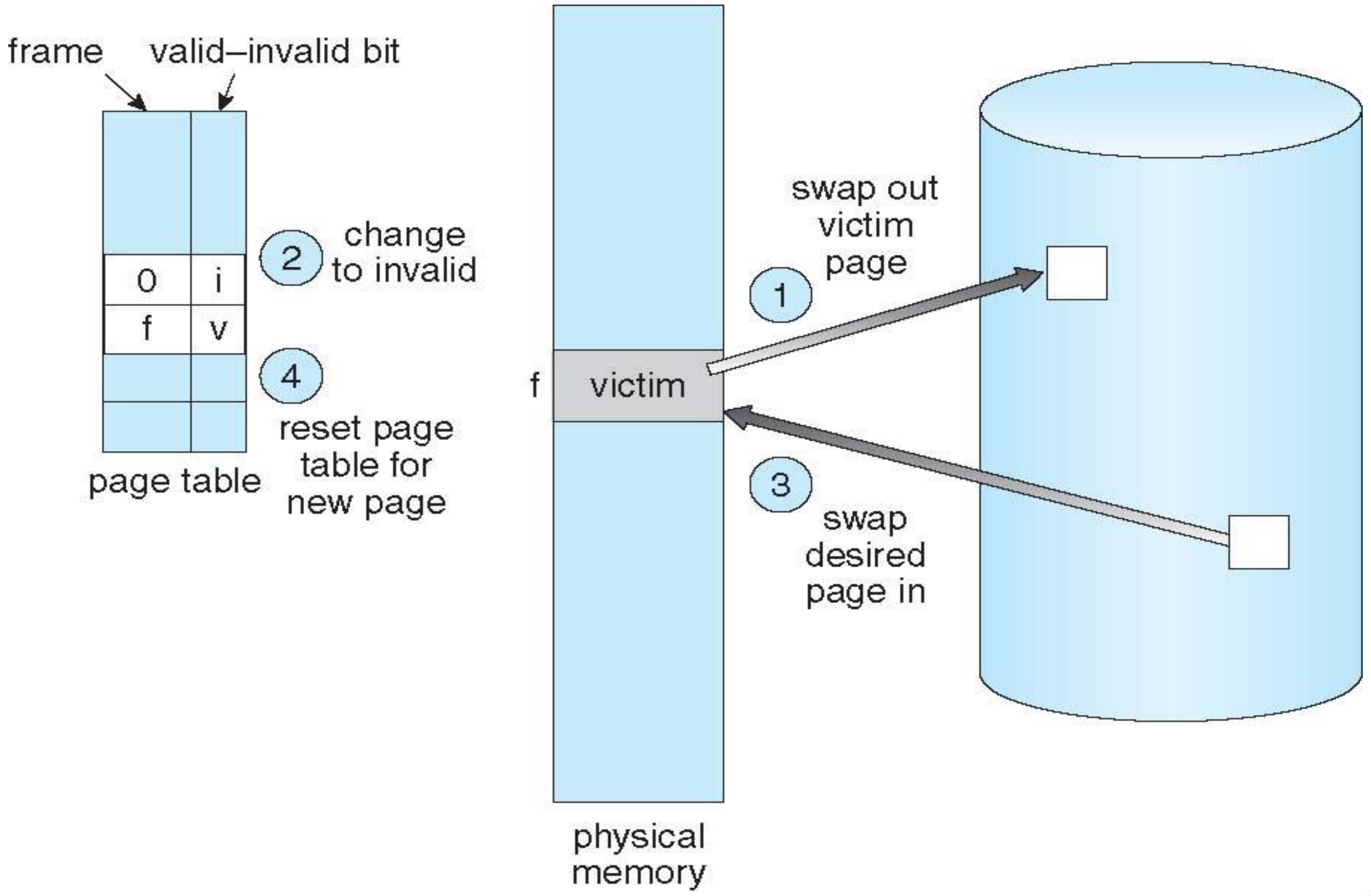
Use **modify (dirty) bit** to reduce overhead of page transfers – only modified pages are written to disk

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

➔ reduce this overhead by using a modify bit (dirty bit)

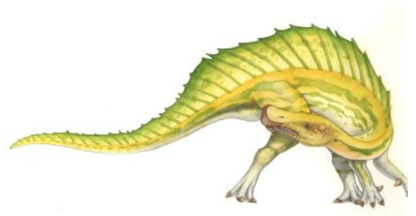
Page Replacement



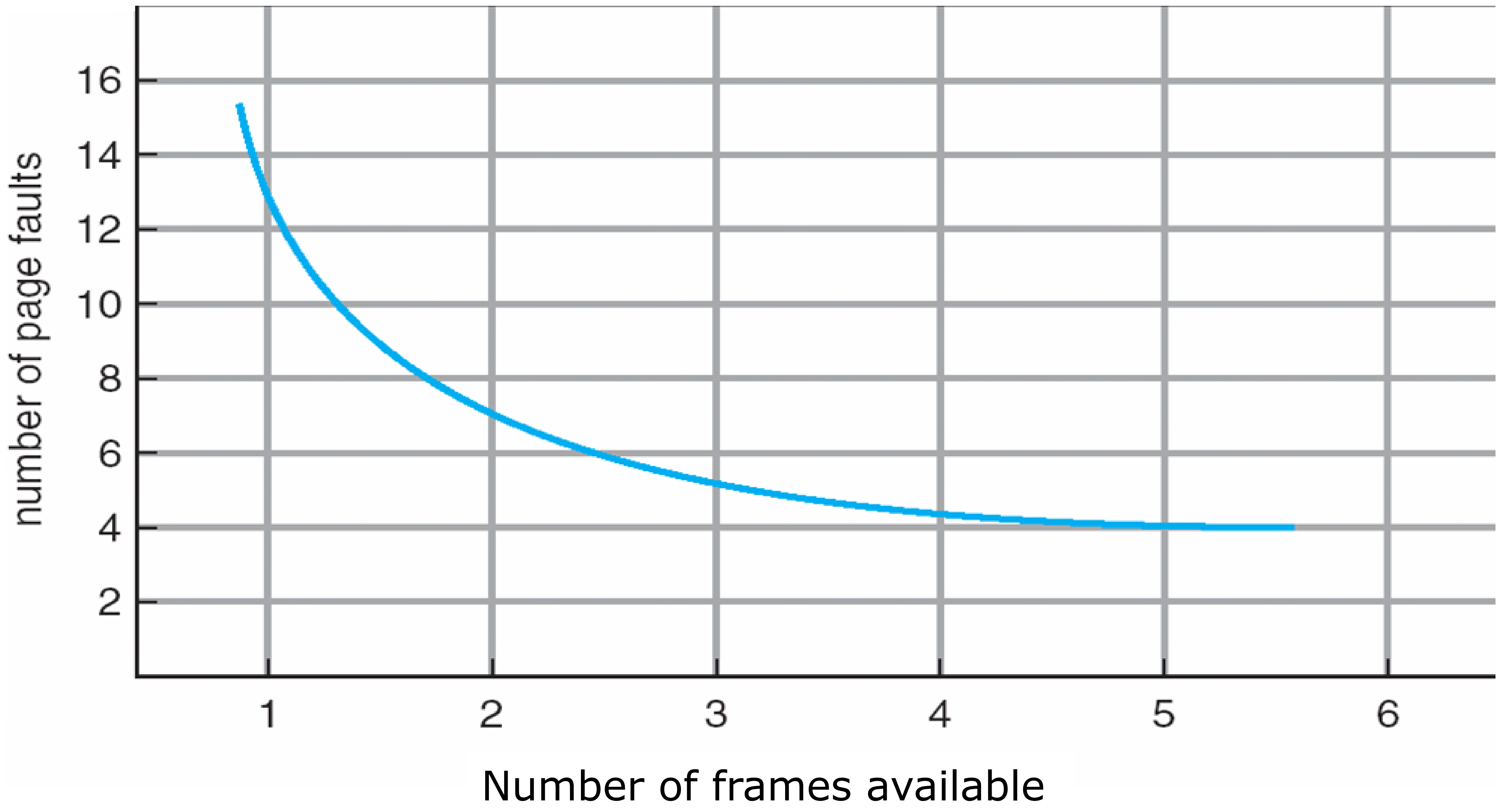
Page and Frame Replacement Algorithms

- **Frame-allocation algorithm** determines
  - How many frames to give each process
- **Page-replacement algorithm** determines
  - Which frames to replace
  - 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
  - If there is a reference to a page *p*, then any Immediately following references to page *p* will 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

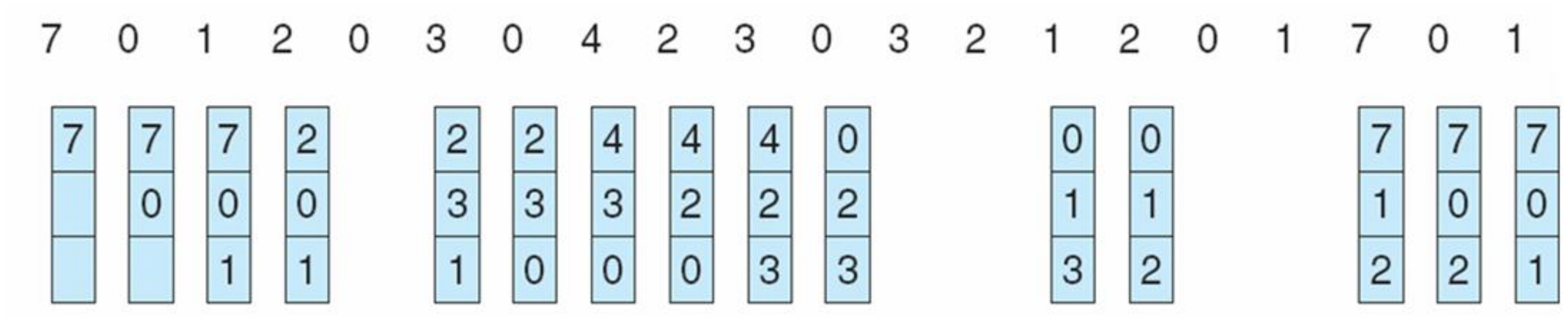


Graph of Page Faults Versus The Number of Frames



First-In-First-Out (FIFO) Algorithm

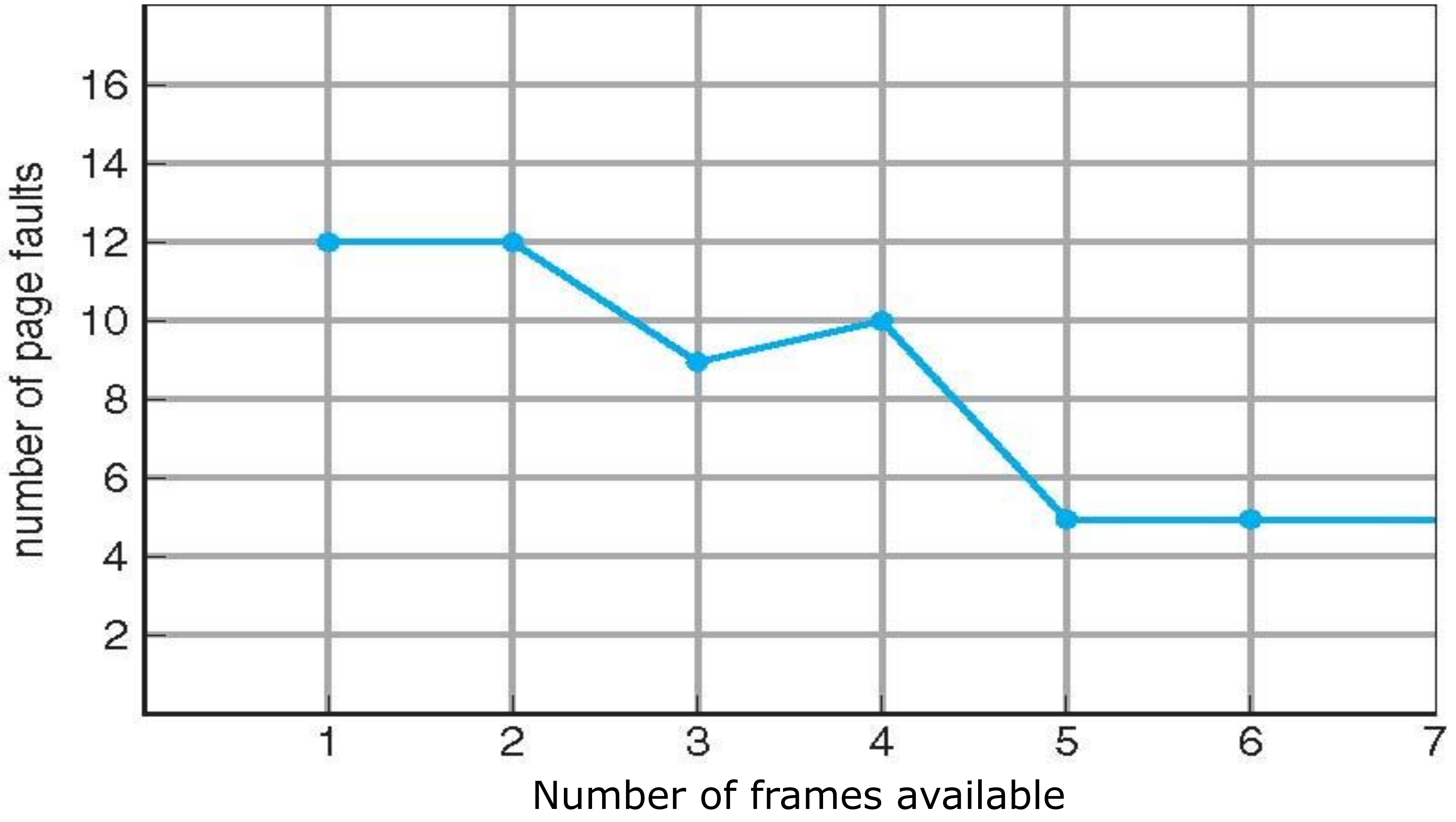
- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)



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

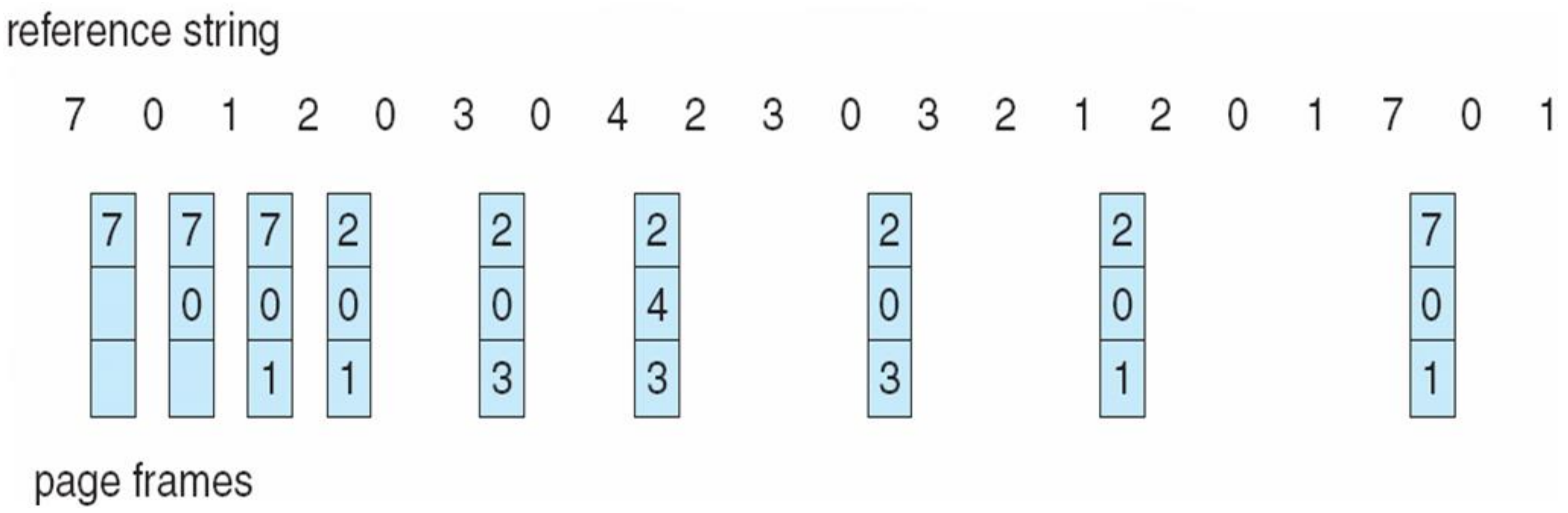
FIFO Illustrating Belady's Anomaly



The number of faults for 4 frames (10) is greater than the number of faults for 3 frames (nine). ➔ **Belady's anomaly**: the page-fault rate may increase as the number of allocated frames increase.

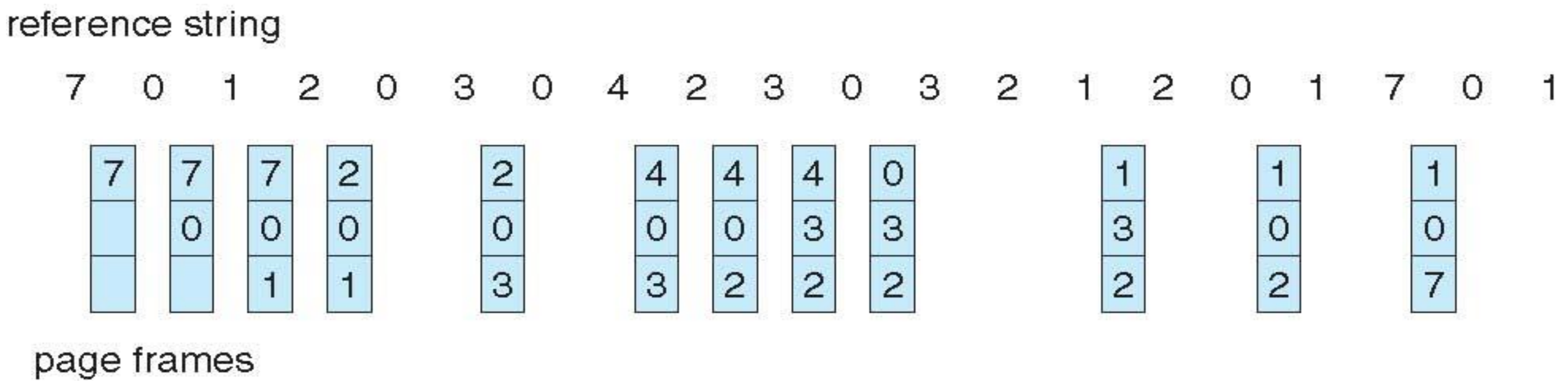
Optimal Algorithm

- 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 implement this?
  - Can’t read the future
- Used 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 the most amount of time
- Associate time of last use with each page



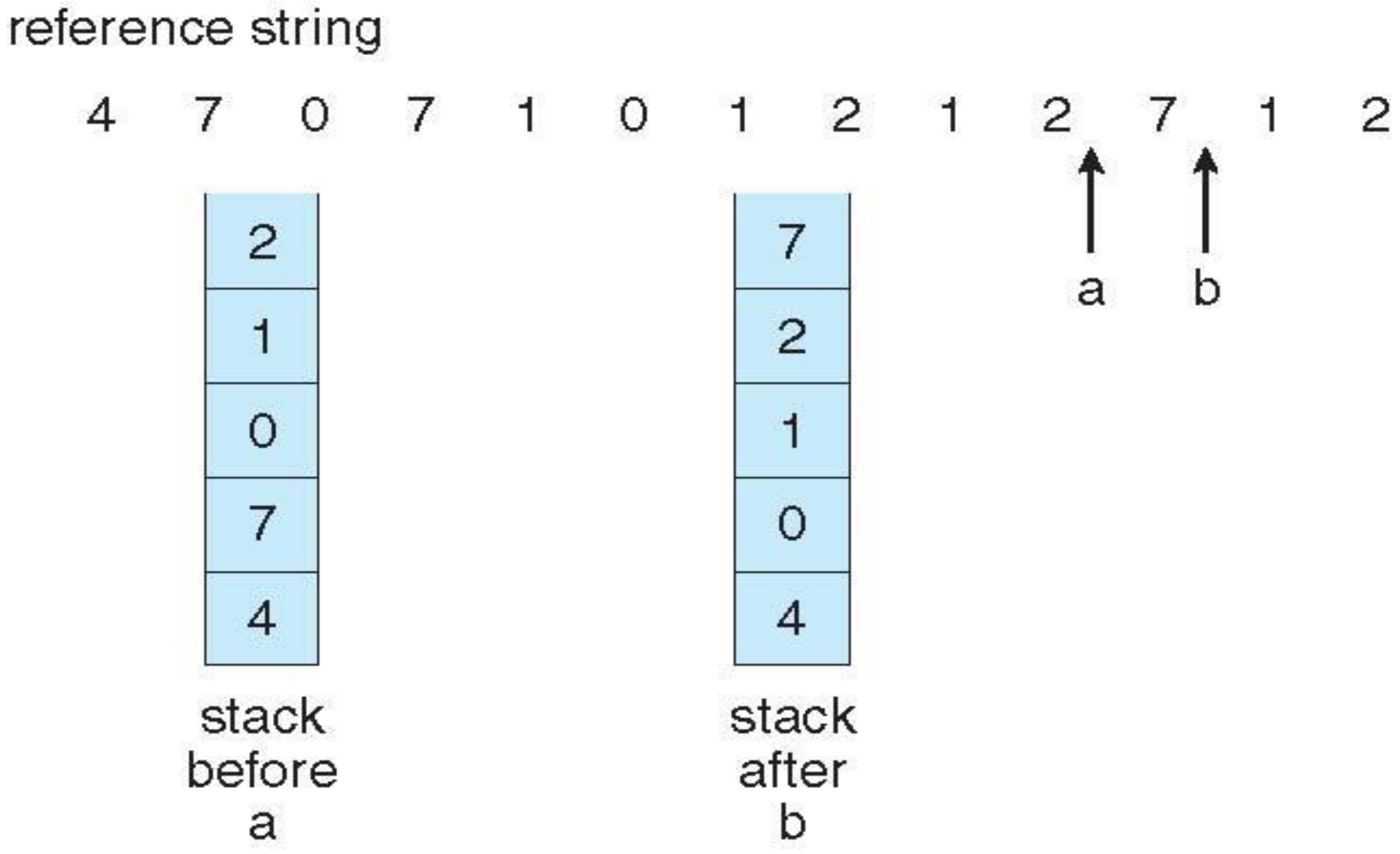
- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- But how to implement? (how to maintain the time of last use?)

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

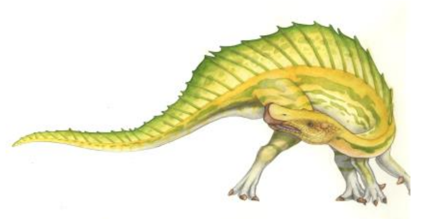
  Belady’s Anomaly

Use Of A Stack to Record The Most Recent Page References

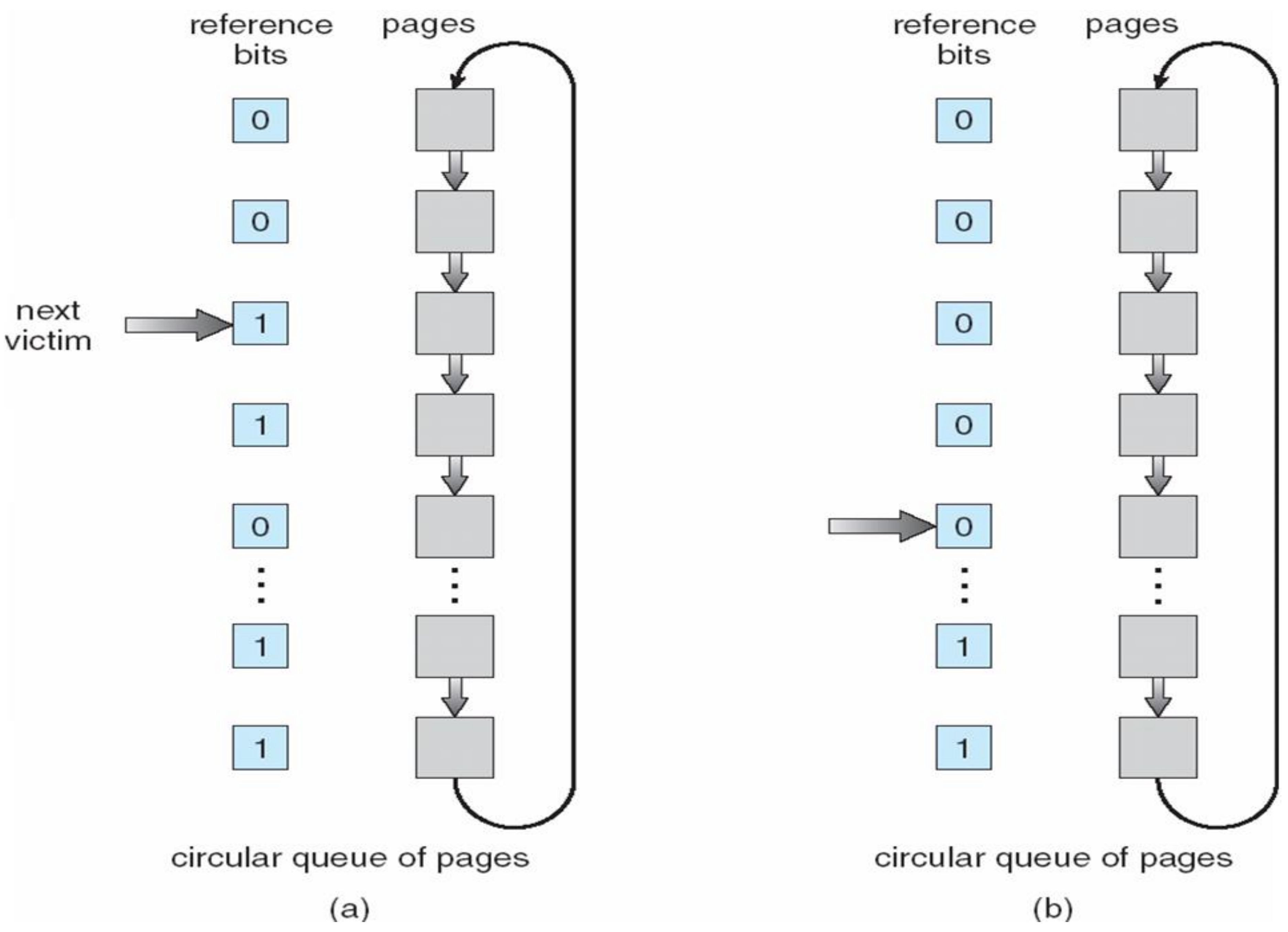


LRU Approximation Algorithms

- 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 (clock replacement)**
  - Generally FIFO, plus hardware-provided reference bit
  - 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



LRU Approximation Algorithms

- **Additional-Reference-Bits Algorithm**
  - Keep more bits (e.g., 8-bit byte) for each page (in a table in memory)
  - At regular interval (e.g., every 100ms), a timer interrupt occurs and OS shifts the reference bit for each page into the high-order bit of its 8-bit byte, shifting the other bits right by 1 bit, and discarding the low-order bit.
    - 00000000 ➔ the page has not been used for 8 time periods.
    - 11111111 ➔ the page is used at least once in each period
    - Considering two pages below, which one is more preferred to be replaced with LRU ?
      - Page A: 11000100   Page B: 01110111
    - If we interpret these 8-bit bytes as unsigned integers, the page with the **lowest number** is the LRU page.

Enhanced Second-Chance Algorithm

- To enhance the second-chance algorithm by considering both the **reference bit** and the **modify bit**.

- Four possible cases

  (ref, modify)

  - (0, 0) : best page to replace
  - (0, 1) : not quite as good, because it needs to be written out before replacement
  - (1, 0) : probably will be used again soon
  - (1, 1) : probably will be used again soon and it needs to be written out to disk.

Counting Algorithms

- Keep a counter of the number of references that have been made to each page ➔ Not common
- **LFU (least frequently used) Algorithm**: replaces page with smallest count. It is based on the reason that an actively used page should have a large reference count.
  - Problem with LFU: When a page is used heavily initially but never used again. It has a large count and thus remains in memory even though it is no longer needed. Solution ➔ shift the counts right by 1 bit at regular intervals, forming an exponentially decaying average usage count.
- **MFU (most frequently used) Algorithm**: based on the argument that the page with the smallest count was probably just brought in and has yet to be used. So the one with largest count should be replaced.

Page-Buffering Algorithms

- Keep a pool of free frames, always
  - Then frame available when needed, not found at fault time
    - ➔ allow the process to restart as soon as possible
  - When a page fault occurs, read desired page into free frame and select victim
  - When convenient, the victim is written out and its frame is added to the pool.
- Possibly, keep list of modified pages
  - When backing store is idle, write pages there and set to non-dirty
    - ➔ increase the probability that a page is clean when it is selected as victim.
- Possibly, keep free frame contents intact and note which pages are in them
  - If referenced again before reused, retrieved it from the free-frame pool and 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, without any file-system data structures
  - **Raw disk** ➔ applications can implement their special-purpose storage service on a raw partition.
    - Bypasses buffering, locking, etc

Chapter 9: Virtual-Memory Management

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped Files
- Allocating Kernel Memory
- Operating-System Examples

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 priority allocation
  - Priority allocation
- Many variations

Fixed Allocation

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

— $s_i = \text{size of process } p_i$

— $S = \sum s_i$

— $m = \text{total number of frames}$

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

$m = 64$

$s_1 = 10 \qquad s_2 = 127$

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

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

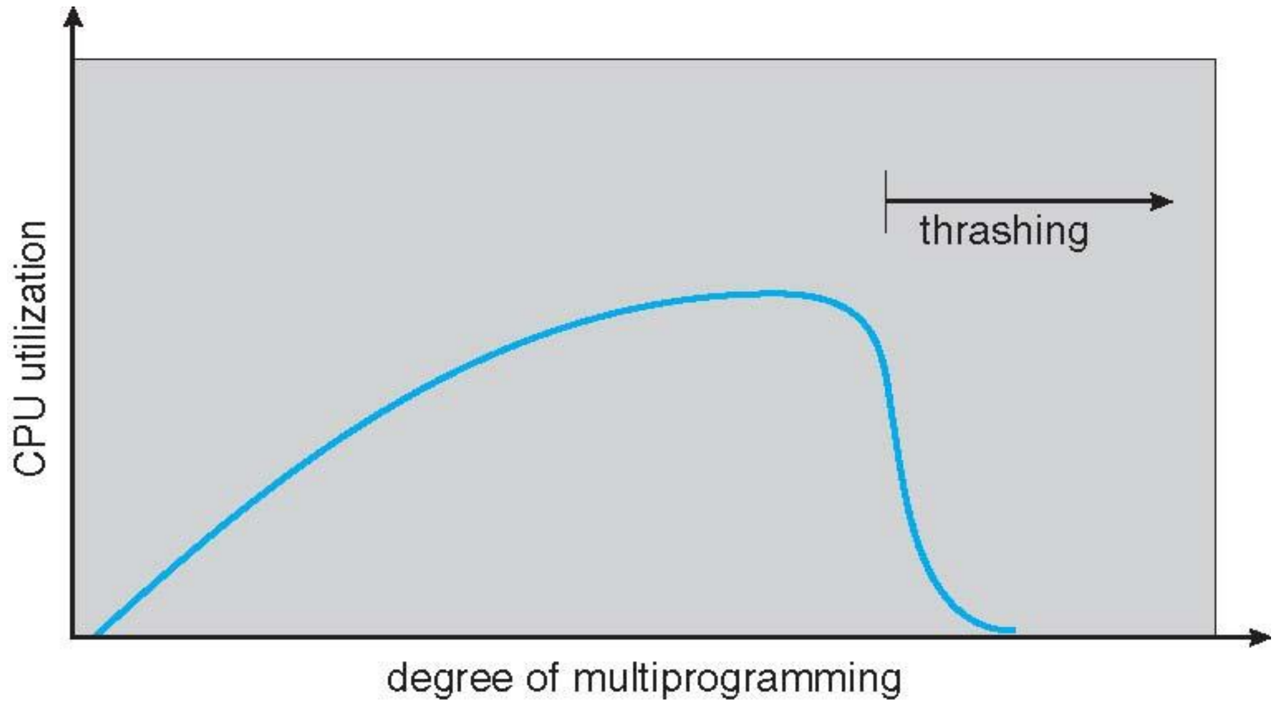
- Priority allocation
  - Use a proportional allocation scheme using priorities rather than size
- If process P_i generates a page fault,
  - select for replacement one of its frames
  - select for replacement a frame from a process with lower priority number

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

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
    - OS 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
  - Limit the trashing effects by local replacement algorithm. If one process starts thrashing, it cannot steal frames from another and cause the latter trashing.
  - Not completely solve the problem because the trashing process will be in the paging queue most of the time, increasing page fault service time for others.

Demand Paging and Thrashing

- Why does demand paging work?

  Locality model

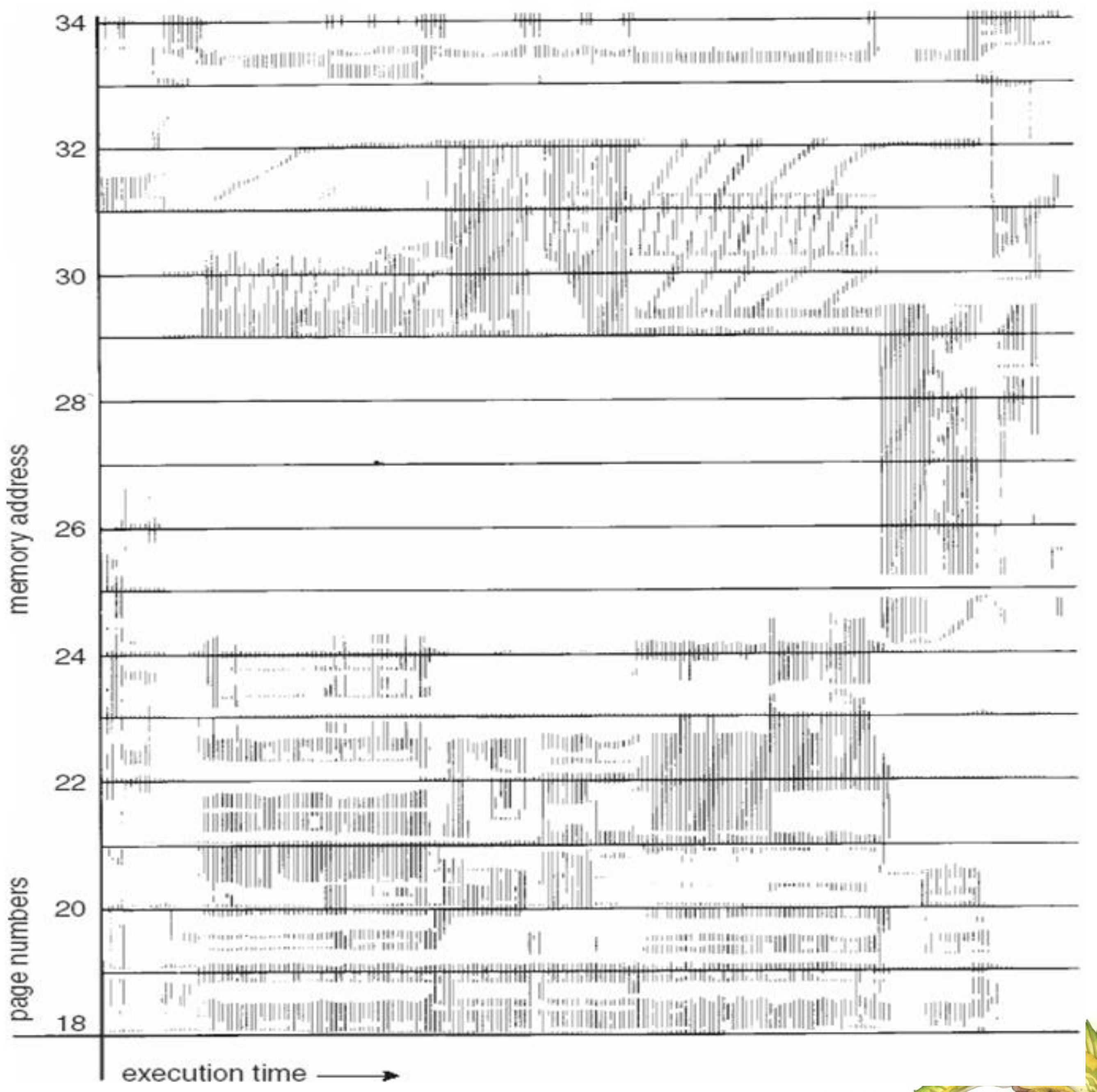
  - Locality: is a set of pages that are actively used together.
  - Process migrates from one locality to another.
  - A program is composed of several localities which may overlap

- Why does thrashing occur?

  Σ size of locality > total memory size

  - Limit effects by using local or priority page replacement

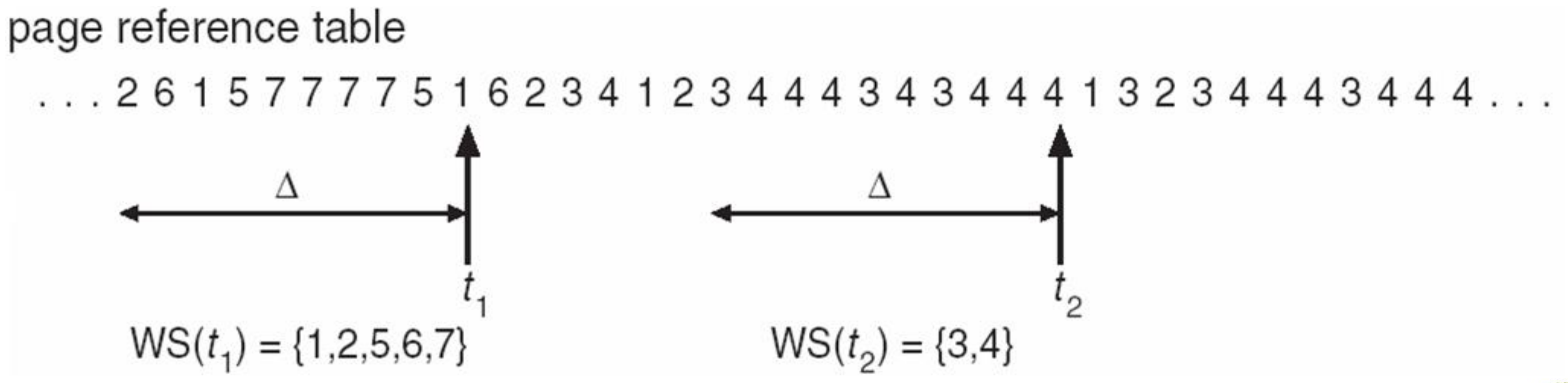
Locality In A Memory-Reference Pattern



Working-Set Model

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

- 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

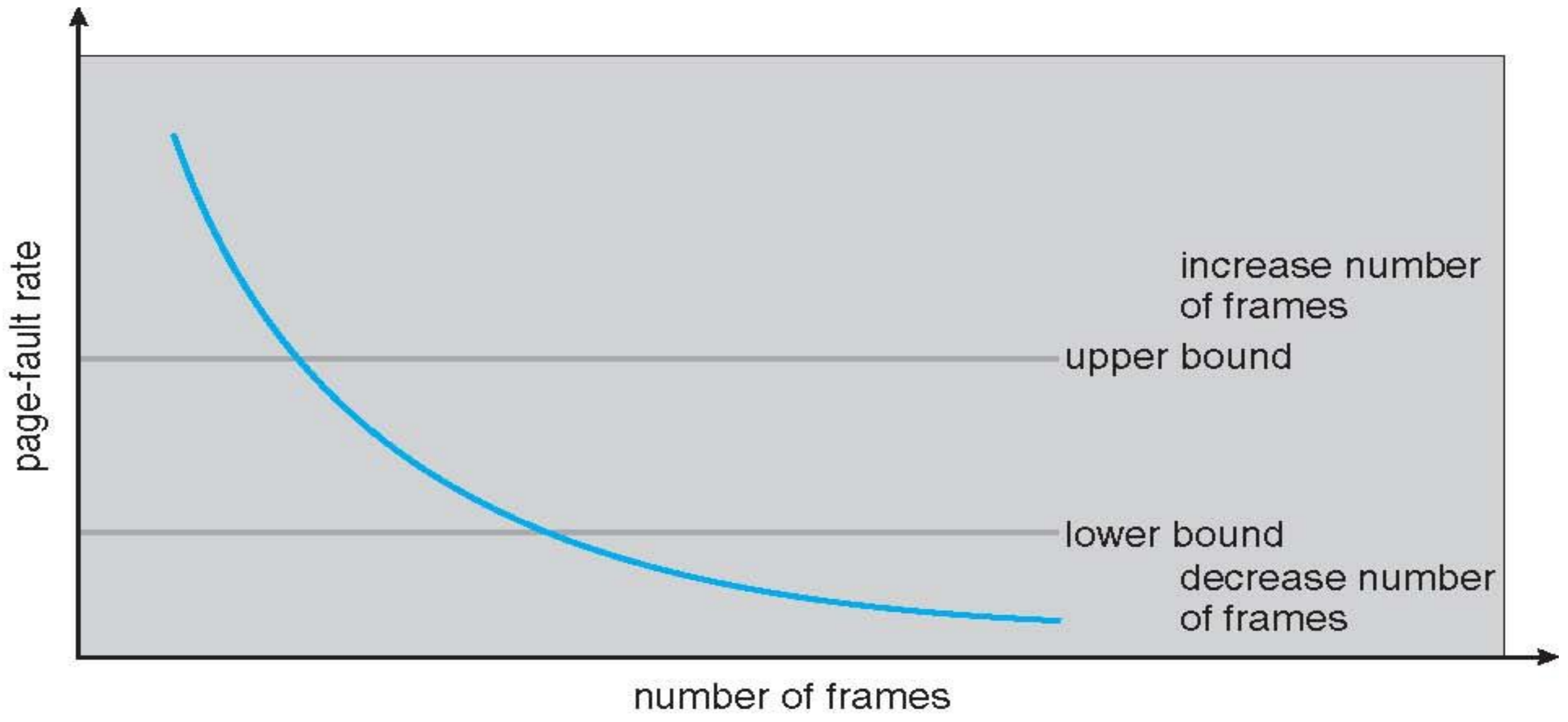


Keeping Track of the Working Set

- $D = \Sigma\, WSS_i \equiv$ total demand frames
- if $D > m \Rightarrow$ Thrashing (*m*: total number of available frames)
- **Policy ➔ if *D* > m, then suspend or swap out one of the processes**
- Approximate the working set with **fixed-interval timer interrupt** + **reference bits**
- 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 🡸 but cost more

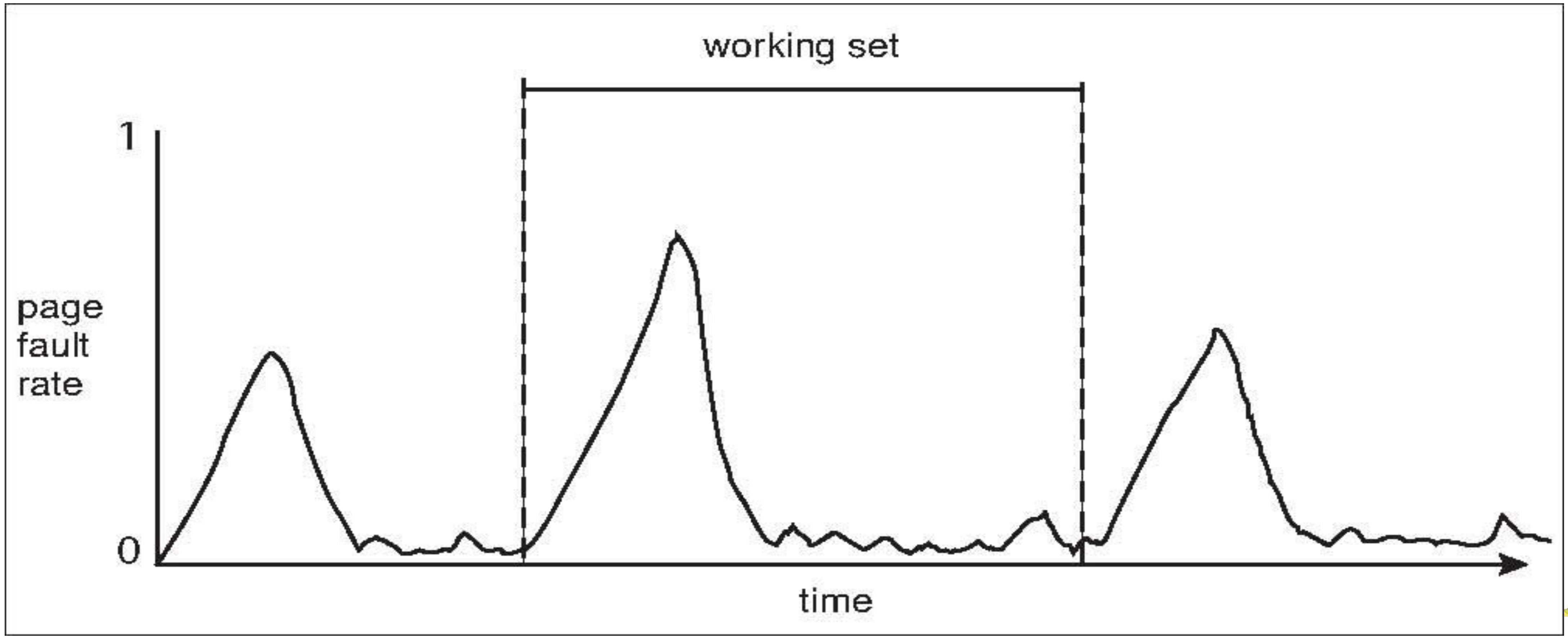
Page-Fault Frequency (PFF)

- More direct approach than WSS
- Establish "acceptable" **page-fault frequency** rate and use local replacement
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame
  - If page-fault rate increases and no free frames are available, select some process and swap



Working Sets and Page Fault Rates

- Assuming there is sufficient memory to store the working set of a process
  - The page-fault rate of a process transitions between peaks and valleys over time.
  - A peak in the page-fault rate occurs when begin a new locality.
  - Once the working set of this new locality is in memory, the page-fault rate falls.



Chapter 9: Virtual-Memory Management

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped Files
- Allocating Kernel Memory
- Operating-System Examples

Memory-Mapped Files

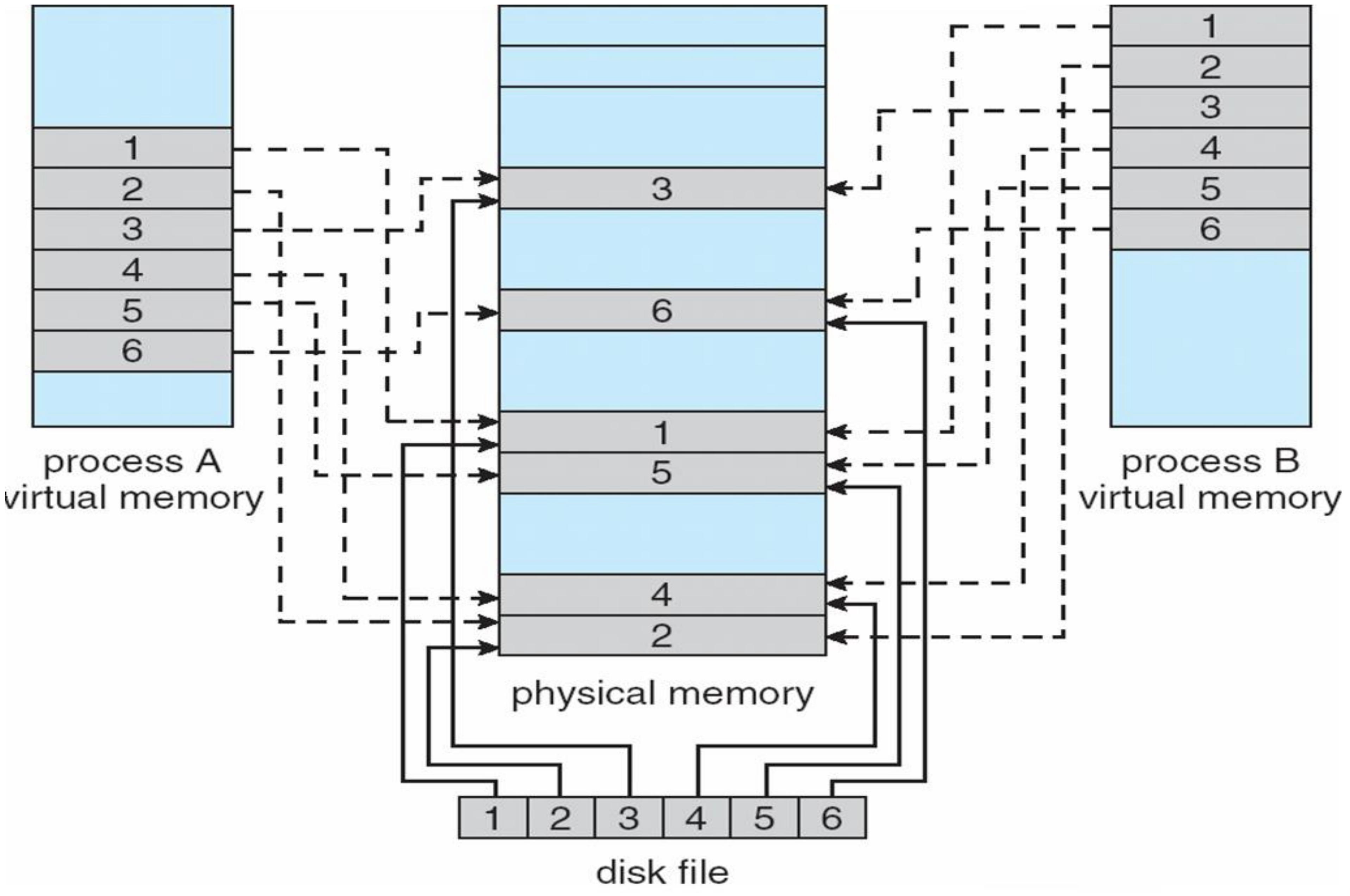
- Memory-mapped file allows file I/O to be treated as routine memory access
  - by **mapping** a disk block to a page in memory
- A file is initially read using demand paging
  - A page-sized portion of the file is read from the file system to a physical page
  - Subsequent reads/writes to/from the file are treated as 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 uses memory mapped files for standard file 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
- Memory mapped files can be used for shared memory (although again via separate system calls)
  - COW (copy on write) can be used for read/write non-shared pages

Memory Mapped Files

- allows several processes to map the same file allowing the pages in memory to be shared



Memory-Mapped I/O

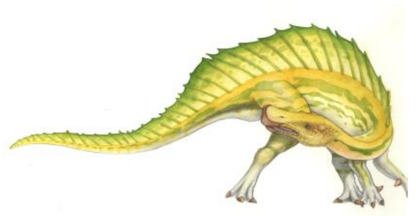
- Ranges of memory addresses are mapped to the device registers.
  - Reads and writes to these memory addresses cause the data to be transferred to and from the device registers.
  - Appropriate for devices with fast response times, such as video controllers.
  - E.g., IBM PC,
    - each location on the screen is mapped to a memory location.
    - displaying text on screen is as easy as writing the text into the memory-mapped locations.

Chapter 9: Virtual-Memory Management

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped Files
- Allocating Kernel Memory
- Operating-System Examples

Allocating Kernel Memory

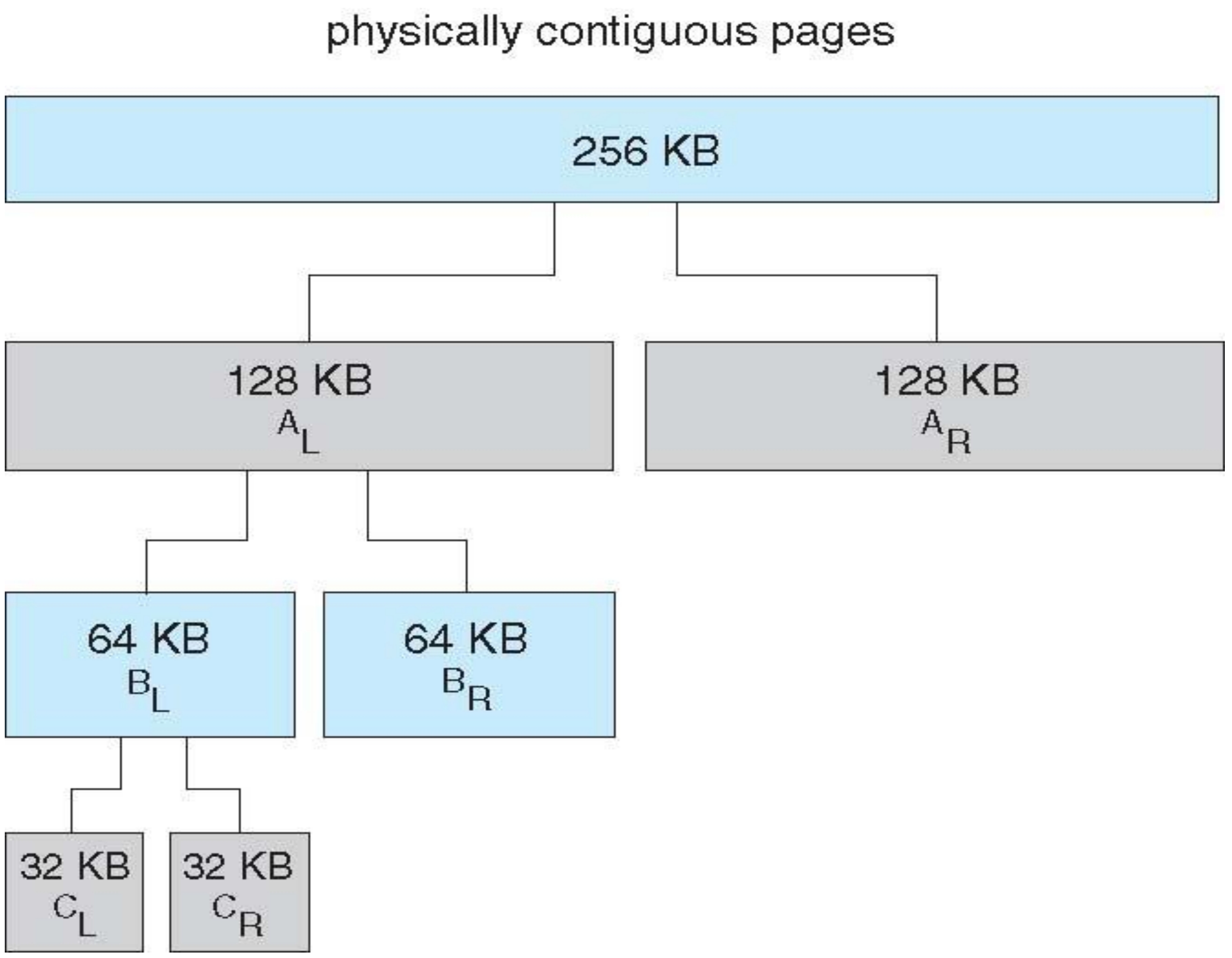
- Treated differently from user memory
- Often allocated from a free-memory pool
  - Kernel requests memory for structures of varying sizes
  - Some are less than a page in size
    - ➔ allocate one page will waste due to internal fragmentation.
  - Some kernel memory needs to be contiguous
    - I.e. for device I/O
- Two strategies for managing memory for Kernel processes:
  - **Buddy system**
  - **Slab allocation**



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 $_{and}$ 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

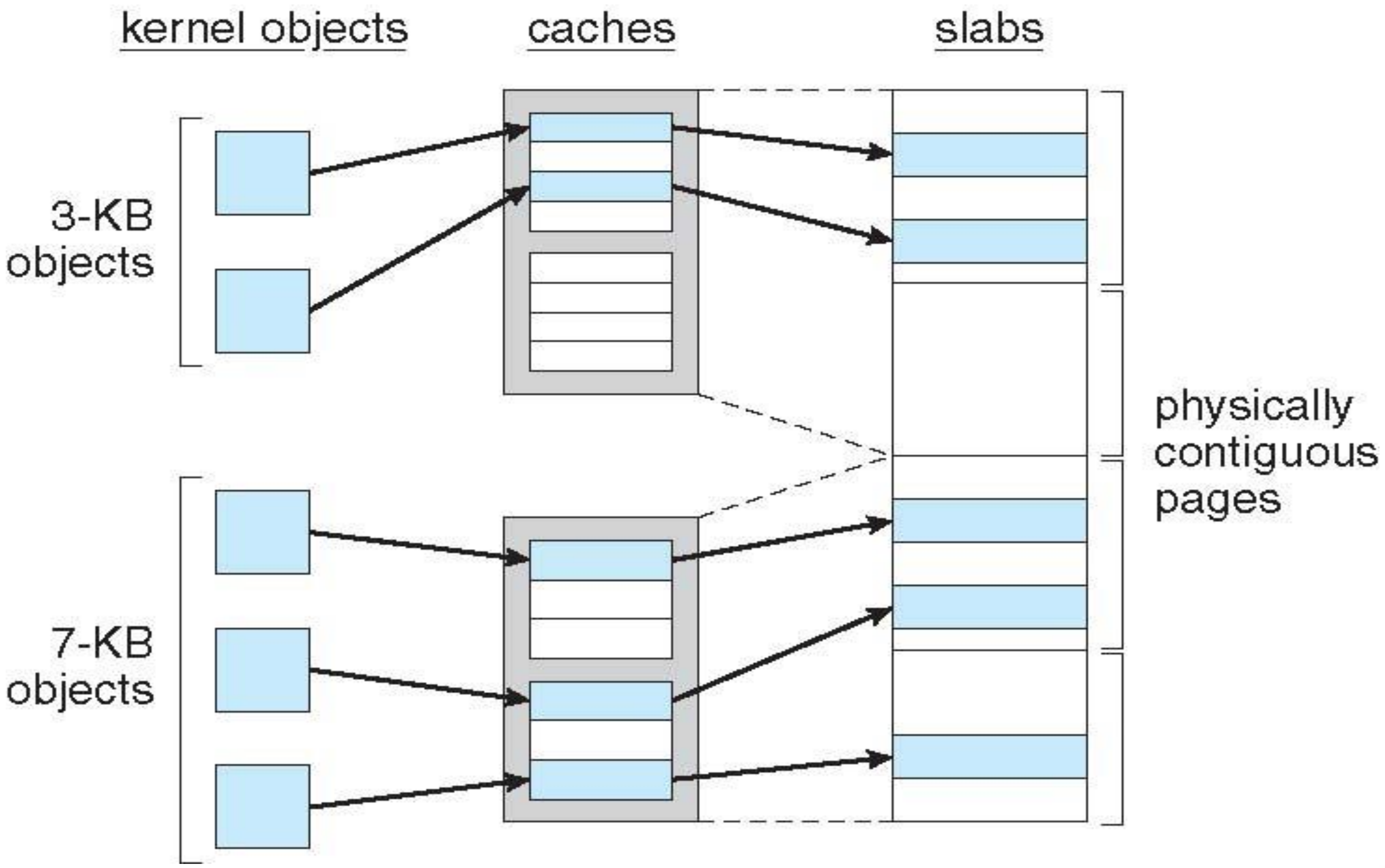
Buddy System Allocator



Slab Allocator

- Alternate strategy for kernel memory allocation
- **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 a new object for kernel data structure is needed, a free object in cache is assigned to satisfy the request. The assigned object is then 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 *α* of the pages is used
    - cost of *s* * *α* *save* pages faults
    - cost of prepaging *s* * *(1- α)* *waste* memory access
      - *α* near zero $\Rightarrow$ prepaging loses
      - *α* near 1 $\Rightarrow$ prepaging wins

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 (internel)
  - Page table size
  - **Resolution** (the smaller page size is , the better resolution is)
  - I/O overhead (time required to read or write a page)
  - 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 ➡ **Row major**

    - *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*

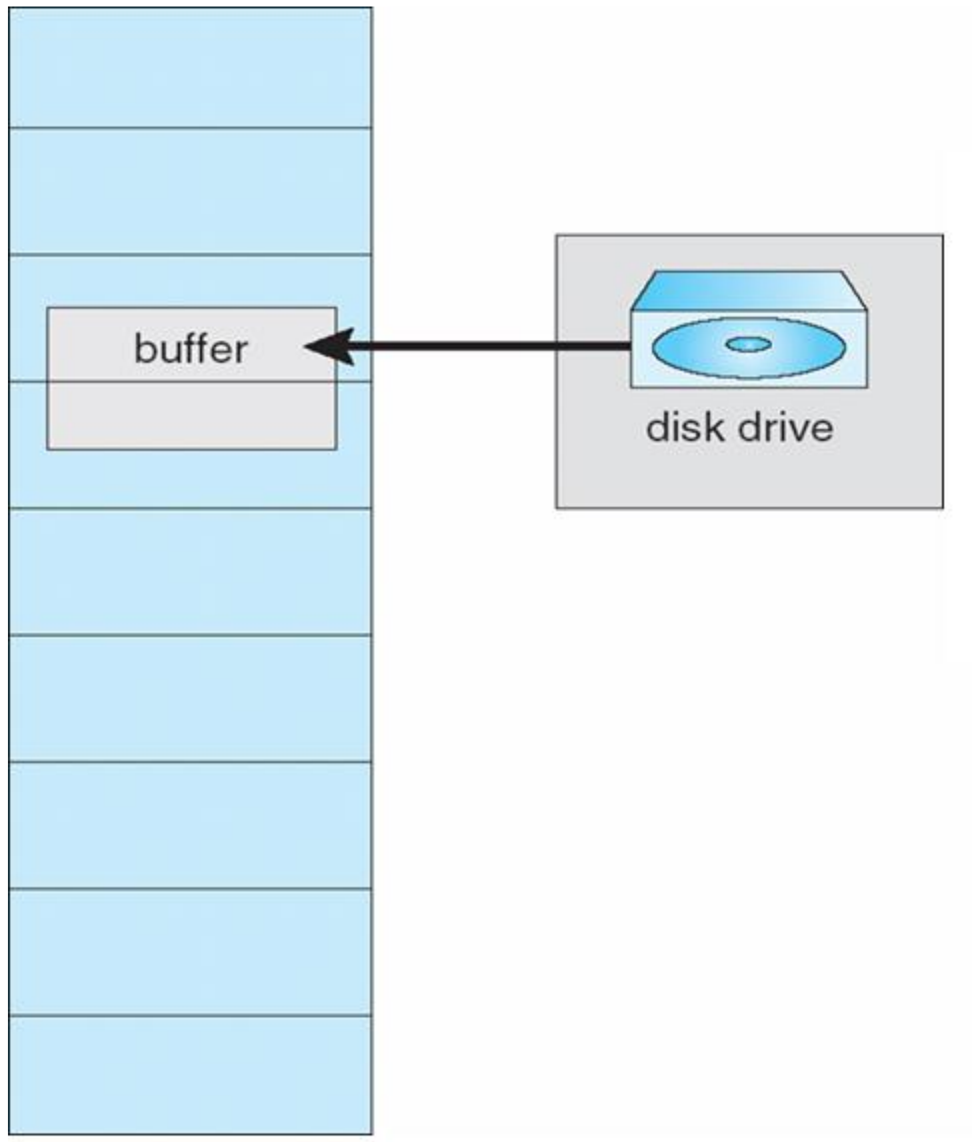
```
for ( i = 0; i < 128; i++ )
        for ( j = 0; j < 128; j++ )
               data[ i , j ] = 0;
```

➔ **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
  - Associate a lock bit with every frame.
  - If a frame is locked, it cannot be selected for replacement.



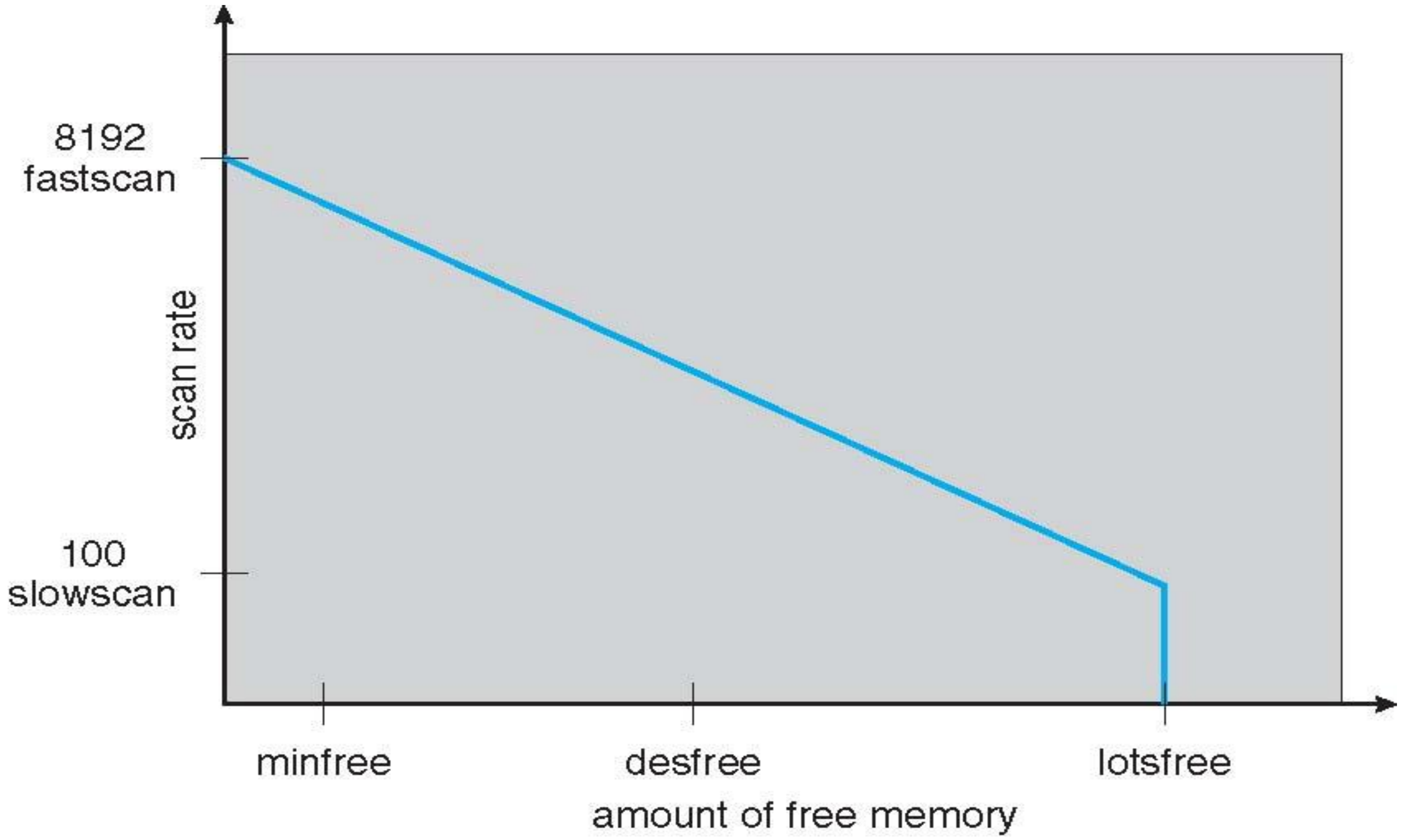
Operating System Examples – Windows XP

- 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

Operating System Examples – Solaris

- Maintains a list of free pages to assign to the processes with page fault
  - *Lotsfree* – threshold parameter (amount of free memory) to begin paging (typically, 1/64 of physical memory size)
    - Paging is performed by *pageout* process (check memory 4 times per sec)
  - *Desfree* – threshold parameter to increasing paging (check memory a hundred times per sec).
  - *Minfree* – threshold parameter to being swapping (*pageout* process is called for every request for a new page)
- Pageout scans pages: using modified clock algorithm.
- *Scanrate* is the rate that pages are scanned, ranging from *slowscan* to *fastscan*
- Pageout is called more frequently depending on the amount of free memory available
- Priority paging gives priority to shared code pages (pages from shared libraries)

Solaris 2 Page Scanner


fastscan
slowscan
scan rate
minfree
desfree
lotsfree
amount of free memory