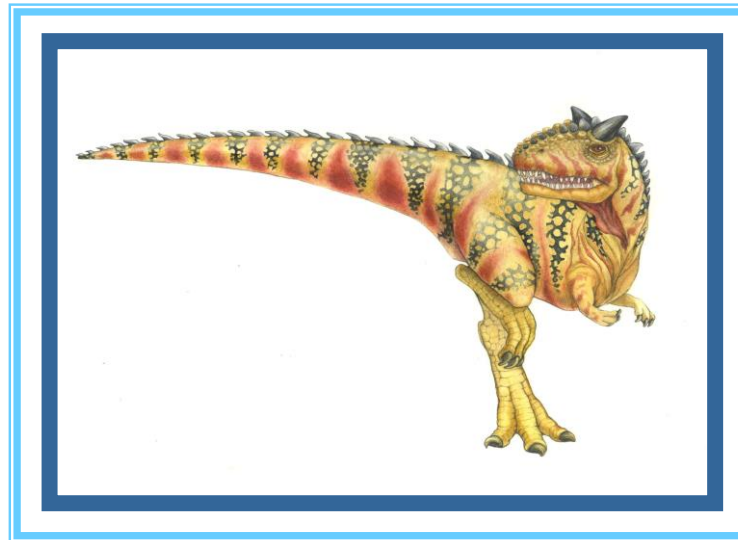


Chapter 9: Virtual-Memory Management





Chapter 9: Virtual-Memory Management

- Background
- Demand Paging
- Page Replacement
- Frame Allocation
- Thrashing
- Other Considerations





Objectives

- ❑ Define virtual memory and describe its benefits.
- ❑ Illustrate how pages are loaded into memory using demand paging.
- ❑ Apply the FIFO, optimal, and LRU page-replacement algorithms.
- ❑ Describe the working set of a process, and explain how it is related to program locality.





Background

- Codes need to be in memory to execute, but not necessarily the entire program.
 - Error code, unusual routines. Some errors seldom, if ever, occur in practice, this code is almost never executed
 - Large data structures such as arrays, lists and tables are often allocated more memory than they actually need. For example, an array may be declared 100x100 elements, even though it is seldom larger than 10x10
- Consider ability to execute partially-loaded programs
 - Programs no longer constrained by limits of physical memory. Programs can be written with an extremely large virtual memory address, simplifying the programming task
 - Each user program could take less physical memory, more programs could be run at the same time, which increases CPU utilization (degree of multiprogramming) and throughput
 - Less I/O would be needed to load or swap user programs into physical memory, so each user program would run faster.





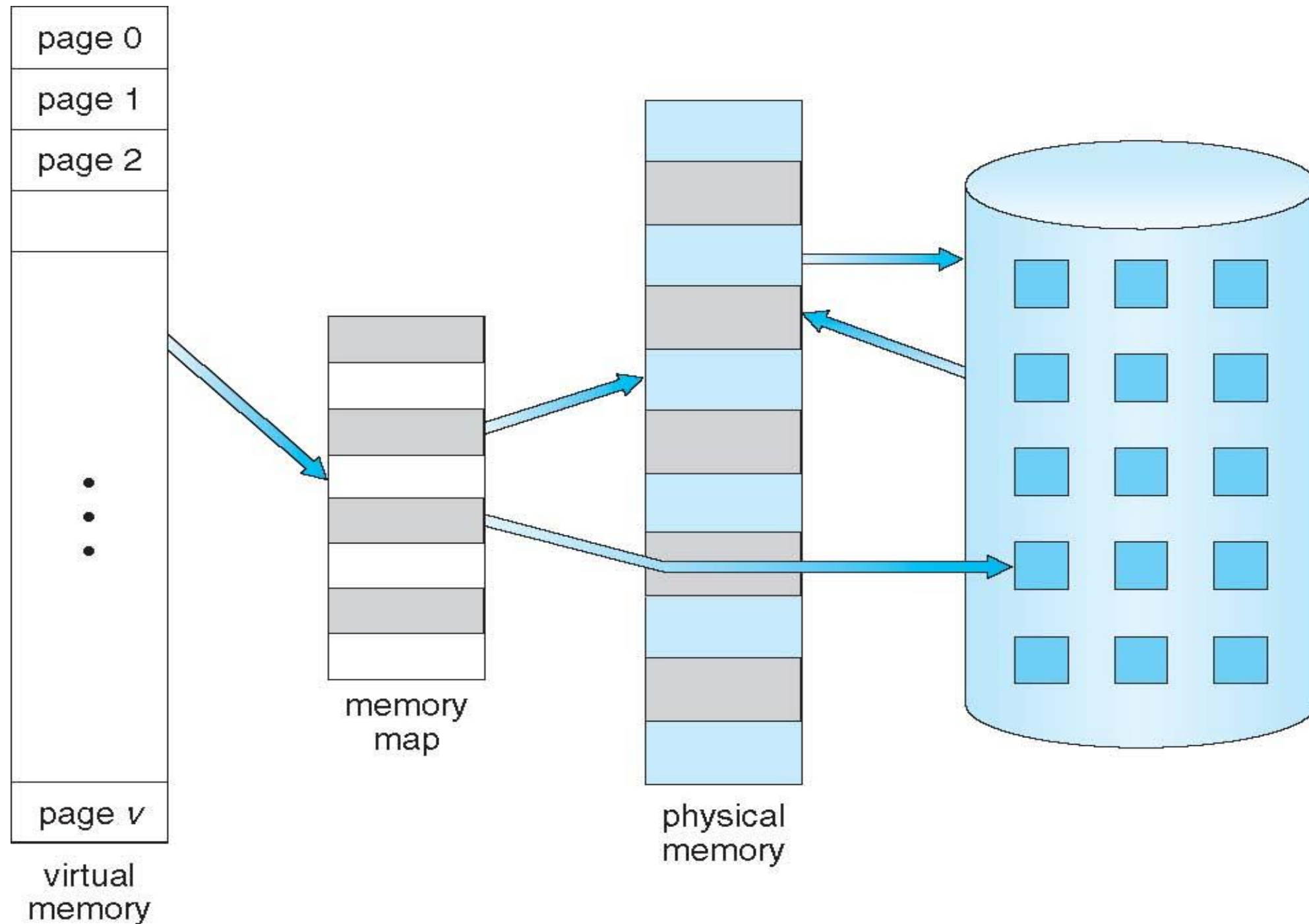
Background

- **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 the actual physical address space
 - Allows address spaces to be shared by several processes. For instance, system libraries can be shared by several processes
 - Allows for more efficient process creation, as pages can be shared during process creation, thus speeding up the process creation
 - More programs running concurrently – increase the degree of multiprogramming
 - Less I/O needed to load or swap processes
- Virtual memory can be implemented via:
 - Demand paging
 - Demand segmentation





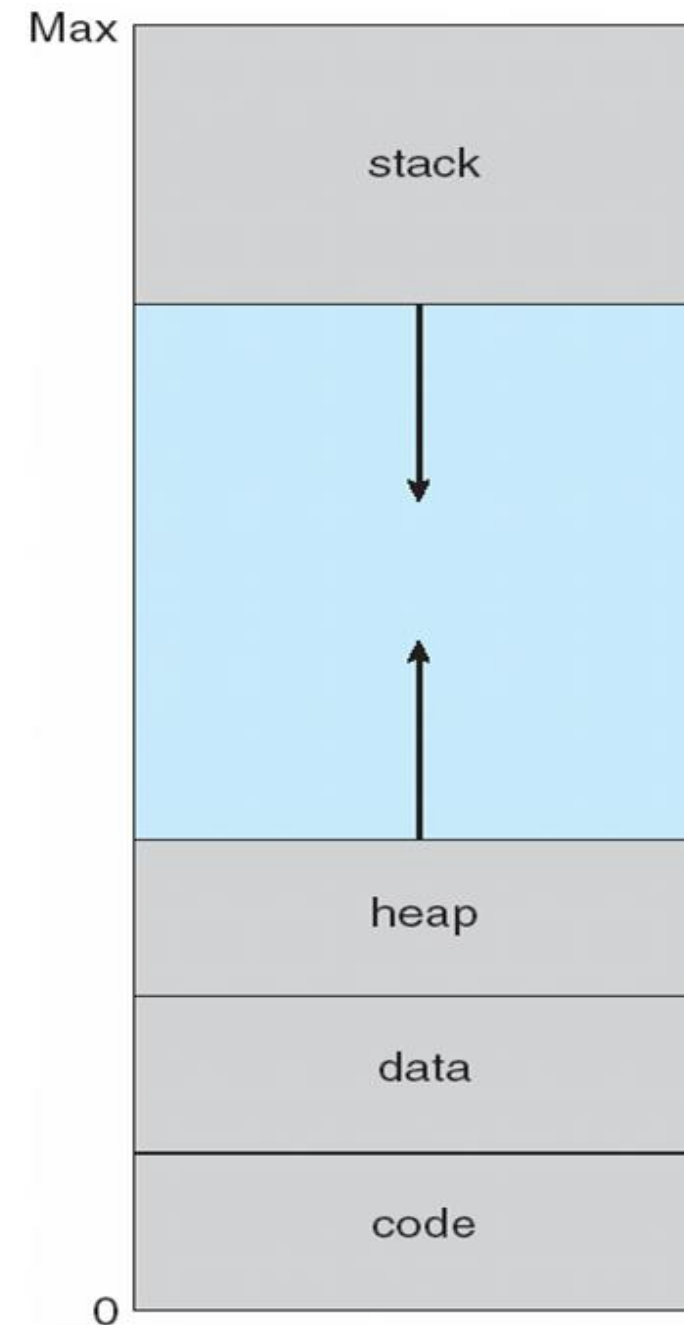
Virtual Memory That is Larger Than Physical Memory





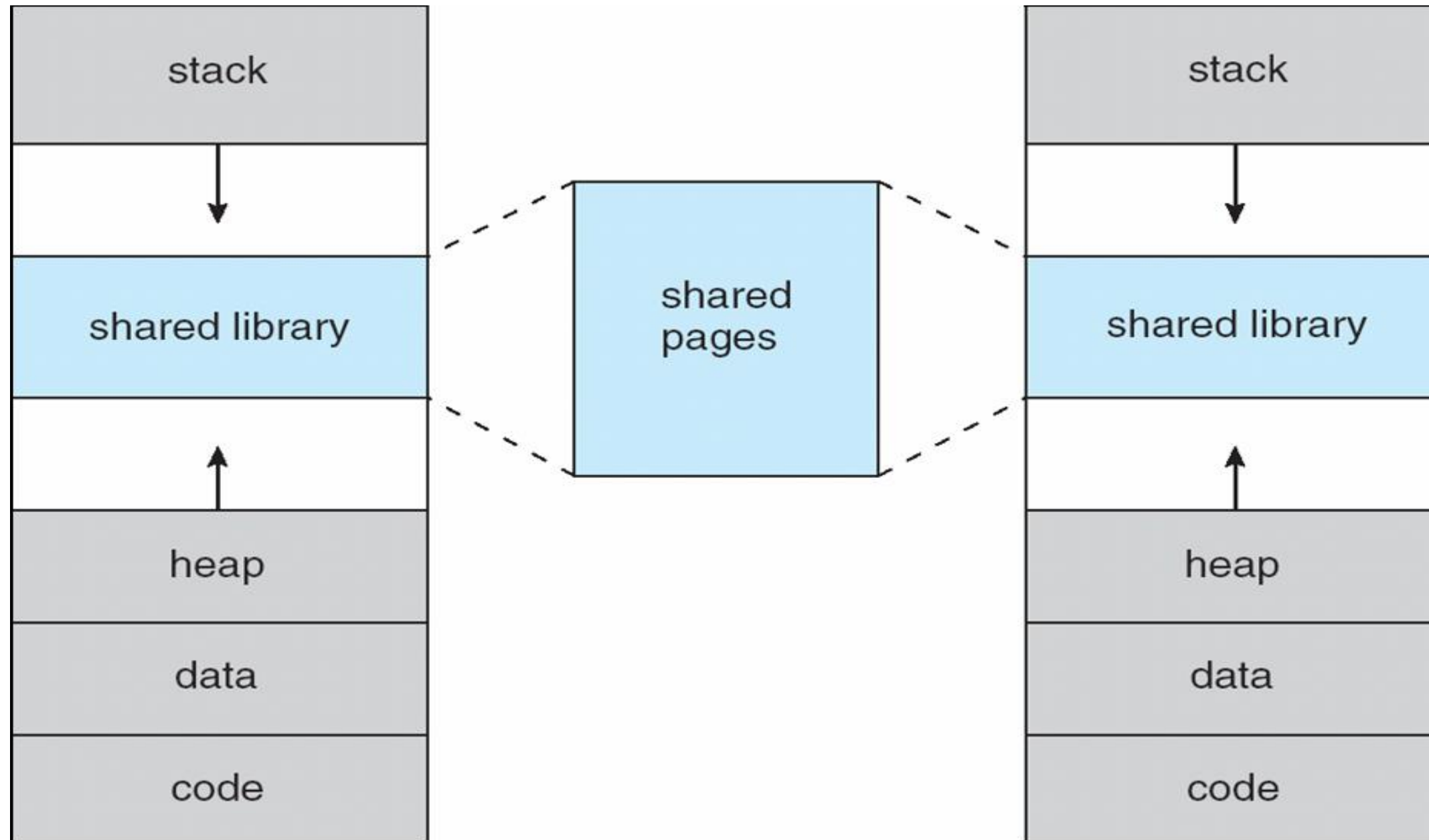
Virtual-address Space

- ❑ **Virtual address space** – 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
- ❑ Heap can grow upward in memory, used in dynamic memory allocation. Stack can grow downward in memory through successive function calls
- ❑ The large blank space (or hole) between the heap and stack is part of the virtual address space, but will require actual physical pages (space) only if the heap or stack grows.
- ❑ Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc
- ❑ System libraries shared via mapping into virtual address space





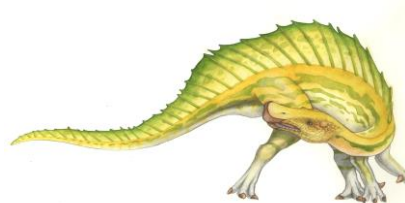
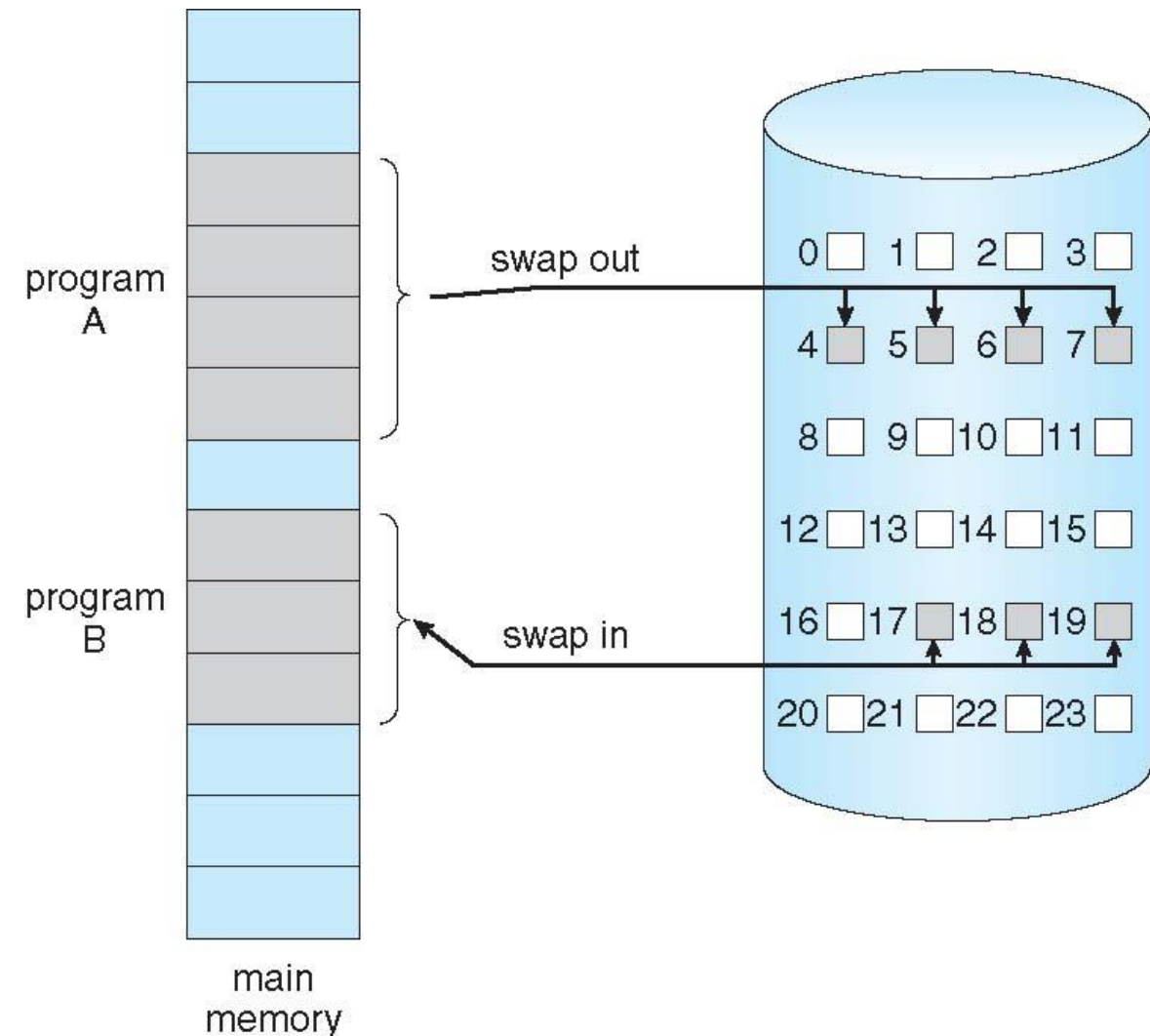
Shared Library Using Virtual Memory





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 to be running
- ❑ Similar to paging system with swapping (diagram on right)
- ❑ Page is needed => reference to it
 - ❑ invalid reference (illegal memory address) => 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
- If page needed and not memory resident
 - Need to detect and load the 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** \Rightarrow in-memory – **memory resident**, **i** \Rightarrow not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:

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

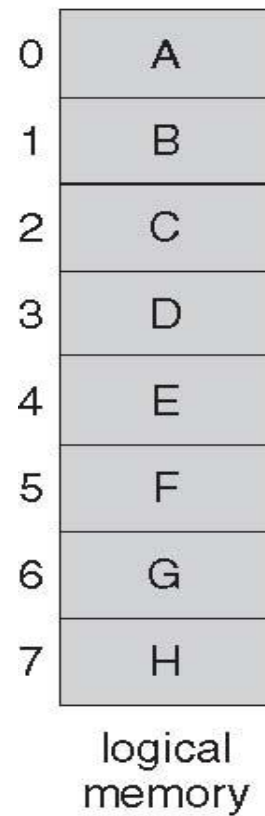
page table

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



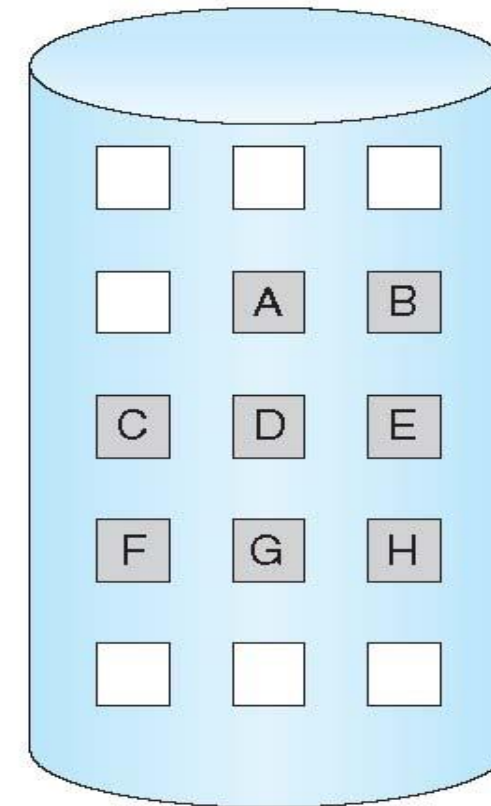
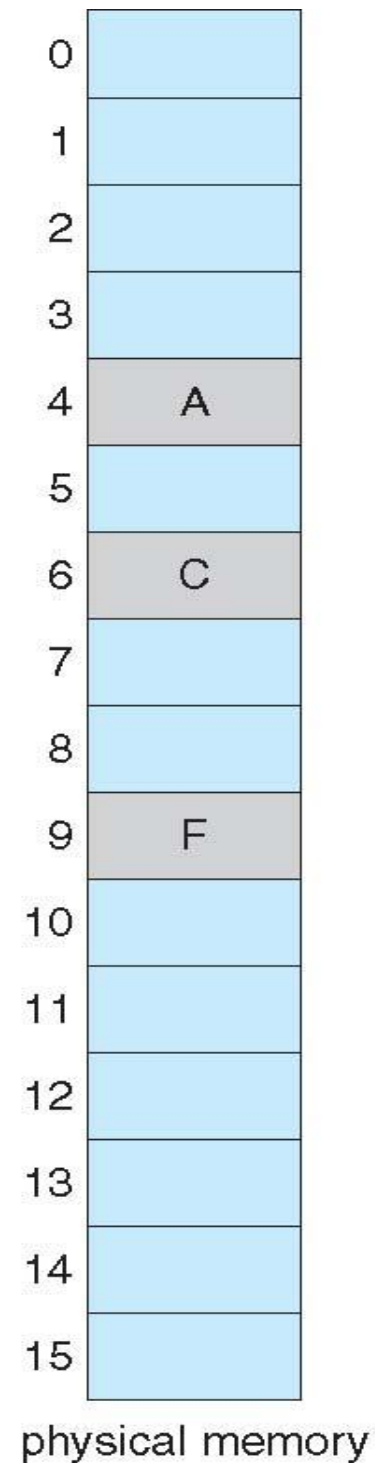


Page Table When Some Pages Are Not in Main Memory



valid-invalid bit		
frame		bit
0	4	v
1		i
2	6	v
3		i
4		i
5	9	v
6		i
7		i

page table





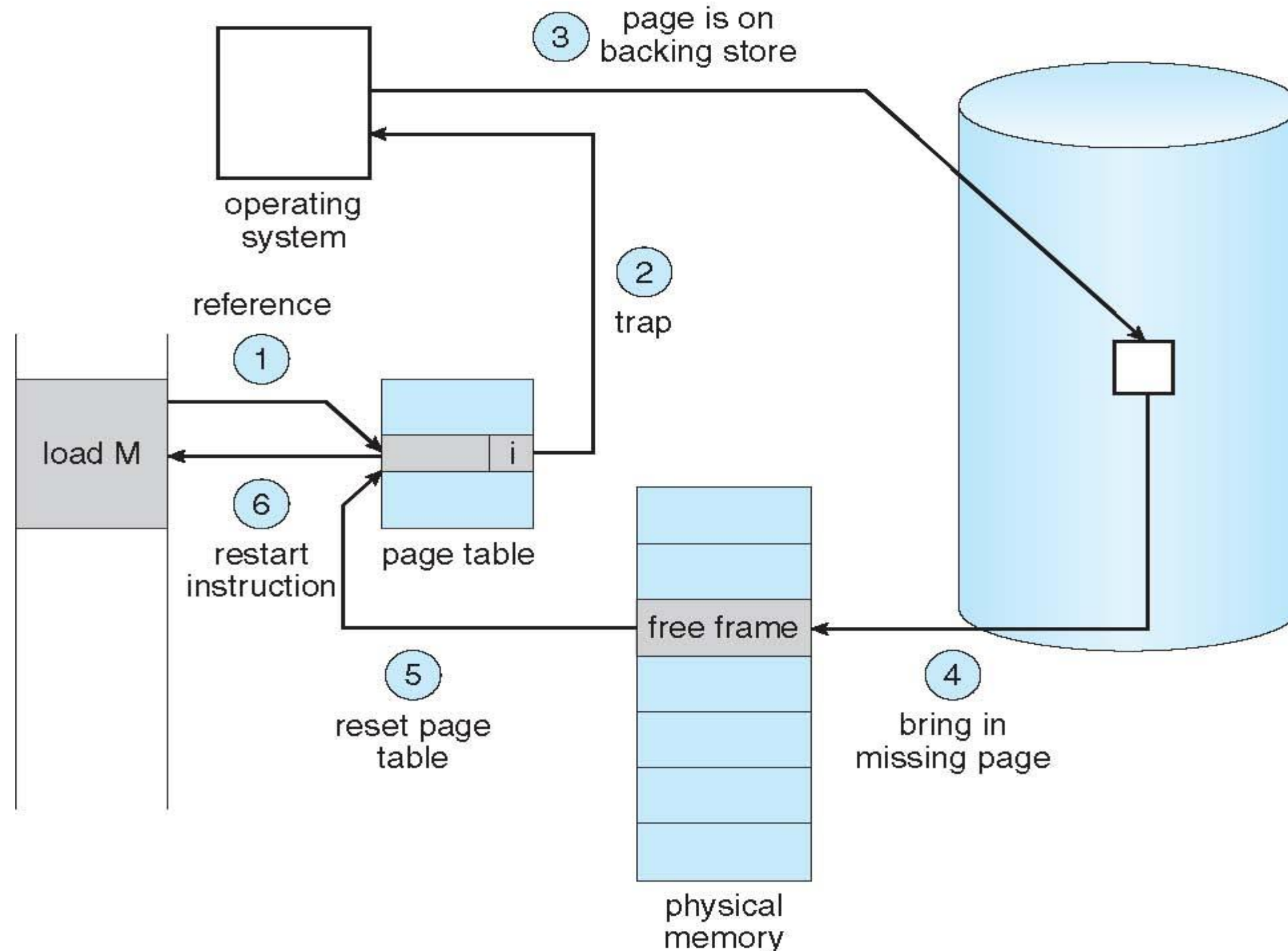
Page Fault

- 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 \Rightarrow abort
 - Just not in memory
- 2. Get an empty frame if any (OS maintains free-frame list)
- 3. Swap the page into the frame via scheduled disk operation
- 4. Reset tables to indicate page now in memory
Set validation bit = **v**
- 5. Restart the instruction (later depending on CPU scheduling) that caused the page fault





Steps in Handling a Page Fault





Aspects of Demand Paging

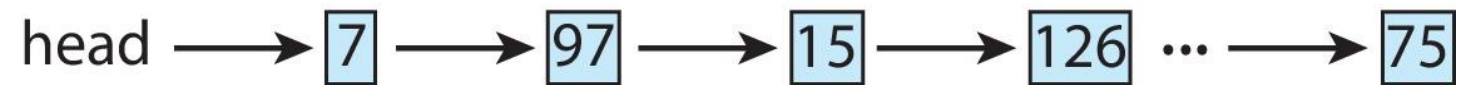
- ❑ Extreme case – start process with *no* pages in memory
 - ❑ OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault
 - ❑ And for every other process pages on first access
 - ❑ Pure demand paging
- ❑ Actually, a given instruction could access multiple pages -> multiple page faults
 - ❑ Consider fetch and decode of instruction which adds two numbers from memory and stores result back to memory
 - ❑ Pain decreased after process starts running for some time because of *locality of reference*
- ❑ Hardware support is needed for demand paging
 - ❑ Page table with valid / invalid bit as indication
 - ❑ Secondary memory (swap device with *swap space*) for page in and page out
 - ❑ Instruction restart





Free-Frame List

- When a page fault occurs, the operating system must bring the desired page from secondary storage into main memory.
- Most operating systems maintain a **free-frame list** -- a pool of free frames for satisfying such requests.



- Operating system typically allocate free frames using a technique known as **zero-fill-on-demand** -- the content of the frames **zeroed-out** before being allocated.
 - The technique of the writing of zeros into a page before it is made available to a process (to keep any old data from being available to the process).
 - Consider the potential security implications of not clearing out the contents of a frame before reassigning it.
- When a system starts up, all available memory is placed on the free-frame list.

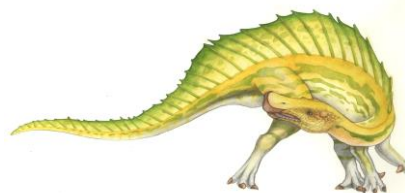




Performance of Demand Paging

□ 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 (if available) in physical memory:
 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 processes
7. Receive an interrupt from the disk I/O subsystem (I/O completed)
8. Save the registers and process state for the other process (depending on the CPU scheduling)
9. Determine that the interrupt was from the disk
10. Update 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.)

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

- Page Fault Rate $0 \leq p \leq 1$
 - if $p = 0$ no page faults
 - if $p = 1$, every reference is a fault

- Effective Access Time (EAT)

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





Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- $EAT = (1 - p) \times 200 + p (8 \text{ milliseconds})$
 $= (1 - p) \times 200 + p \times 8,000,000$
 $= 200 + p \times 7,999,800$
- 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
 - $220 > 200 + 7,999,800 \times p$
 $20 > 7,999,800 \times p$
 - $p < .0000025$
 - < one page fault in every 400,000 memory accesses
- Fortunately, the memory locality usually satisfies this, as each memory miss brings an entire page





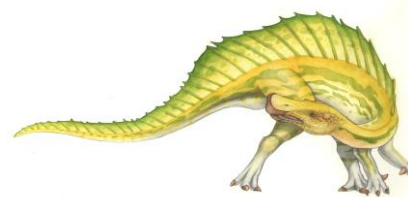
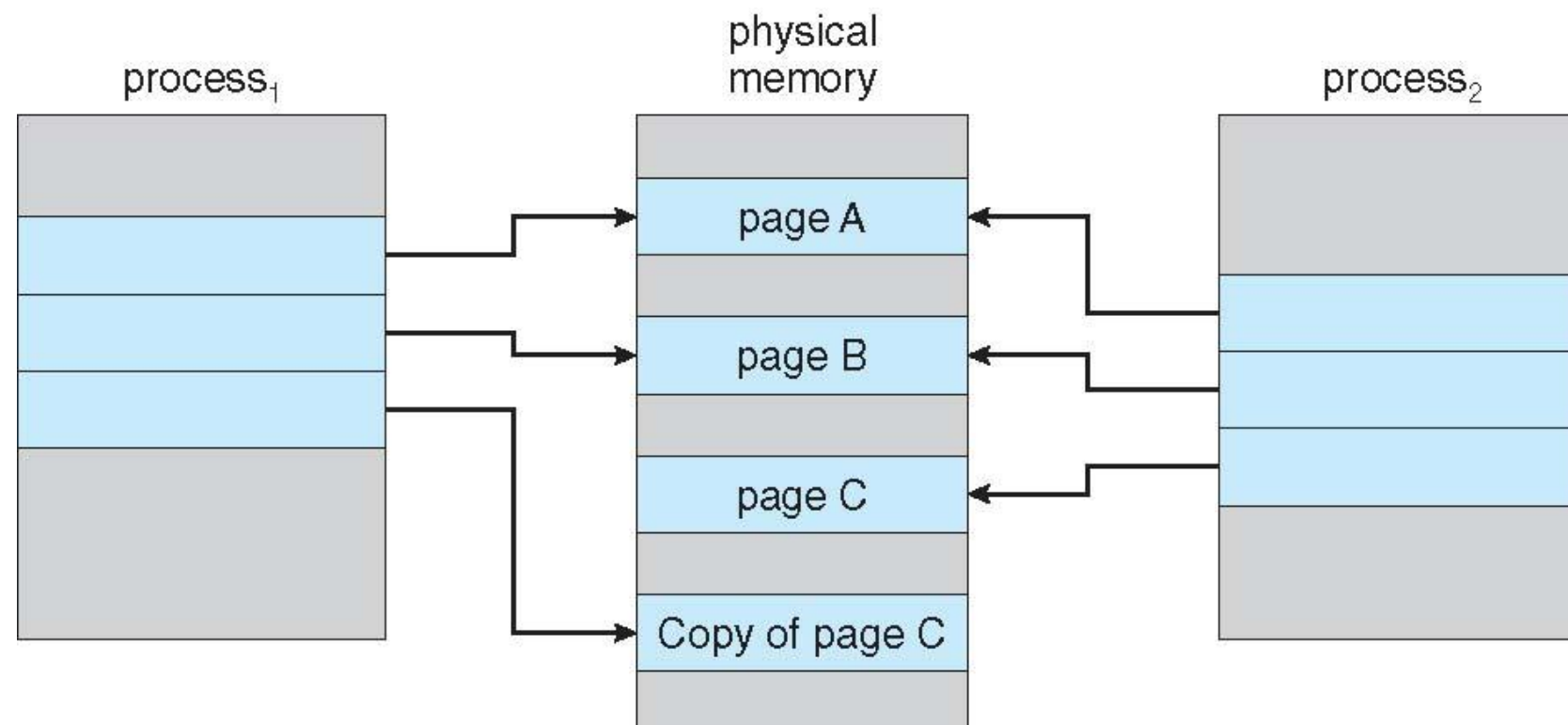
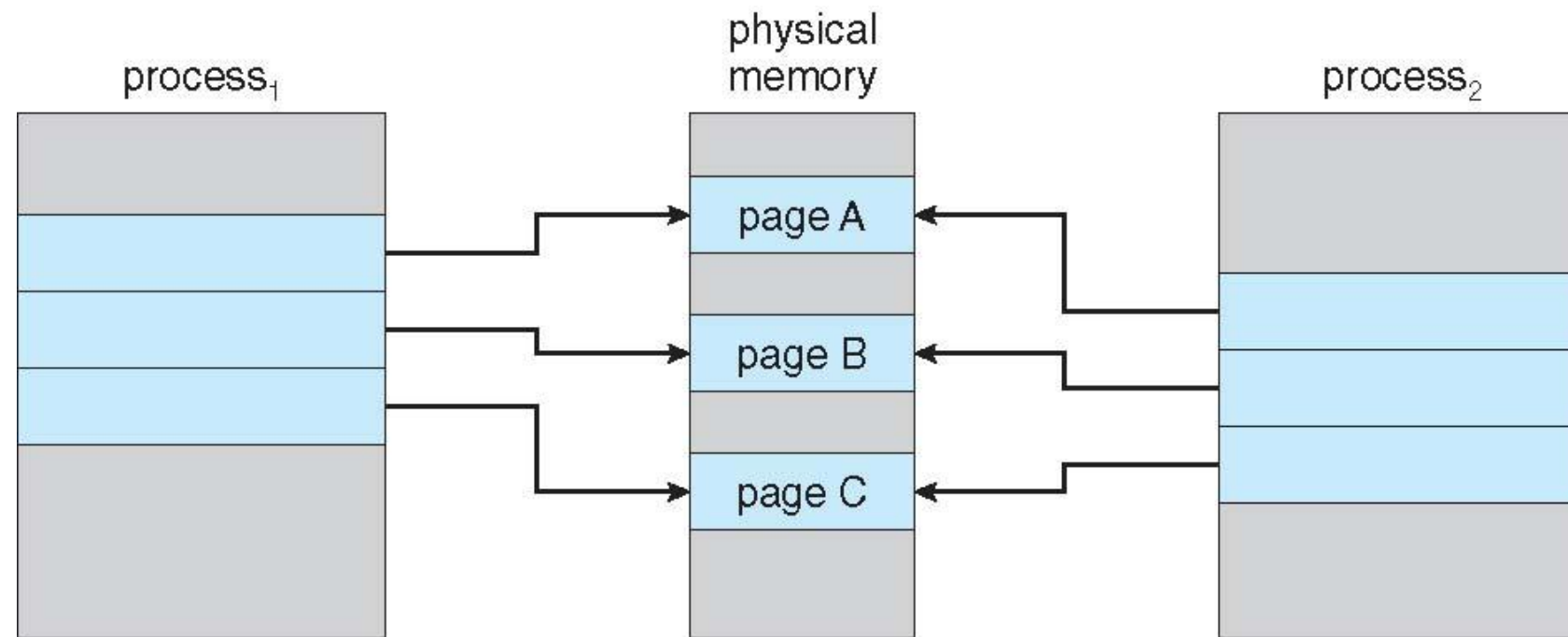
Copy-on-Write

- **Copy-on-Write** (COW) allows both parent and child processes to initially *share* the same pages in memory
 - This technique provides rapid process creation such as `fork()` and minimizes the number of new pages that must be allocated to the newly created process.
 - 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
 - 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
 - Zero-fill-on-demand frames are “zeroed-out” before being allocated, thus erasing their previous contents.
- `vfork()` variation on `fork()` system call has parent suspend and child using copy-on-write address space of parent
 - `vfork()` must be used with caution to ensure that the child process does not modify the address space of the parent
 - This was designed to have child call `exec()`
 - Very efficient





Before and 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 memory to allocate to each process?

- 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 the minimum number of page faults

- Same page may be brought into memory several times





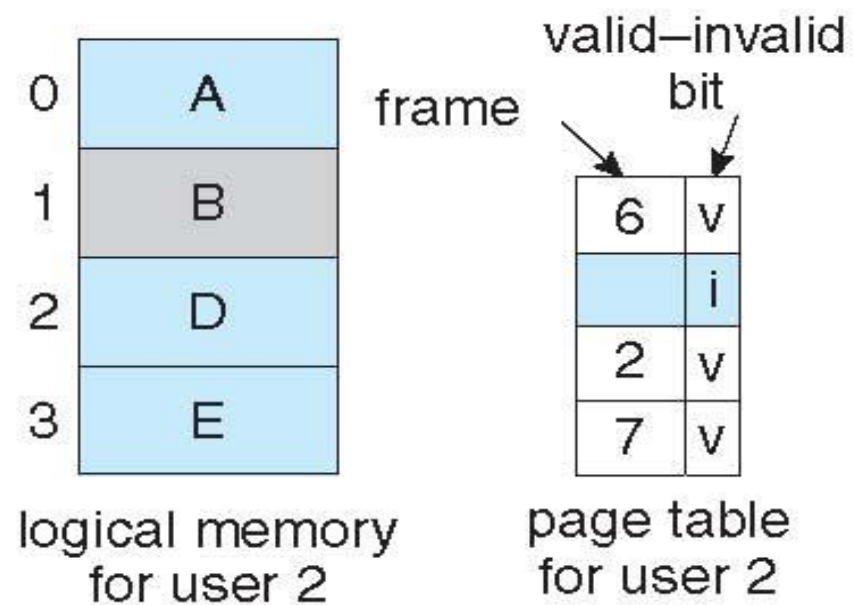
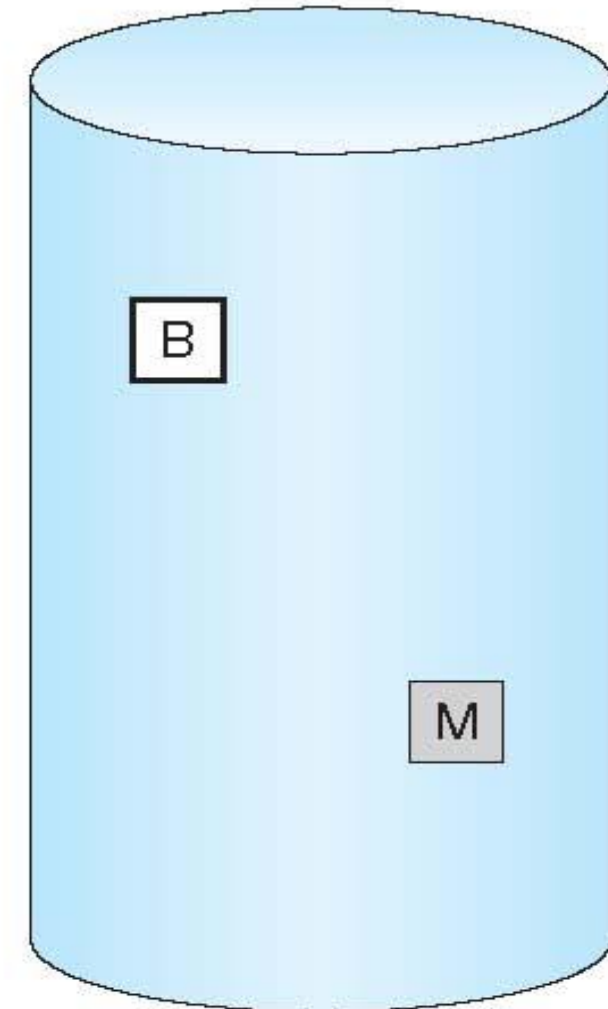
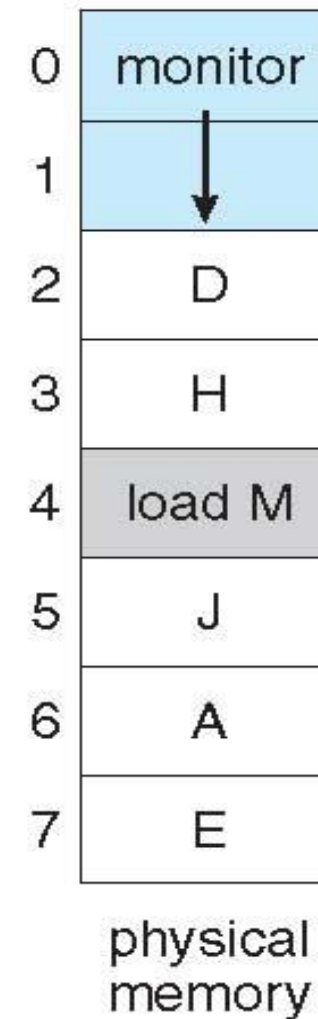
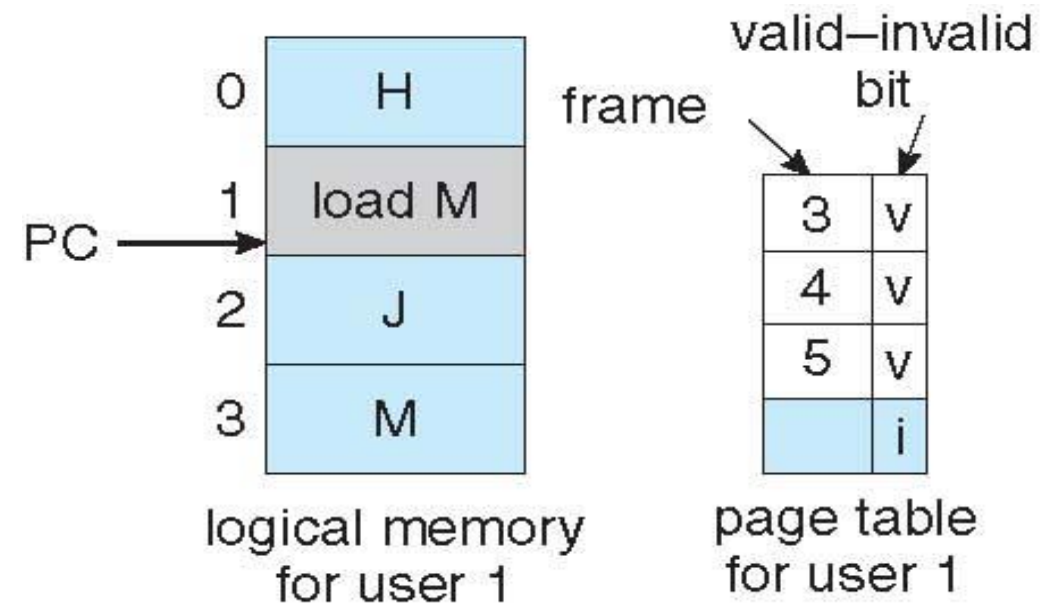
Page Replacement

- ❑ 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 back to disk
- ❑ Page replacement completes separation between logical memory and physical memory – large virtual memory can be supported on a smaller physical memory





Need For Page Replacement

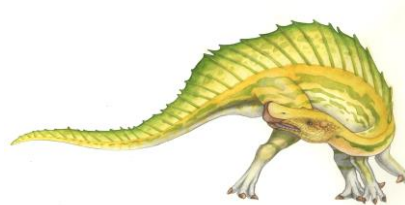




Basic Page Replacement

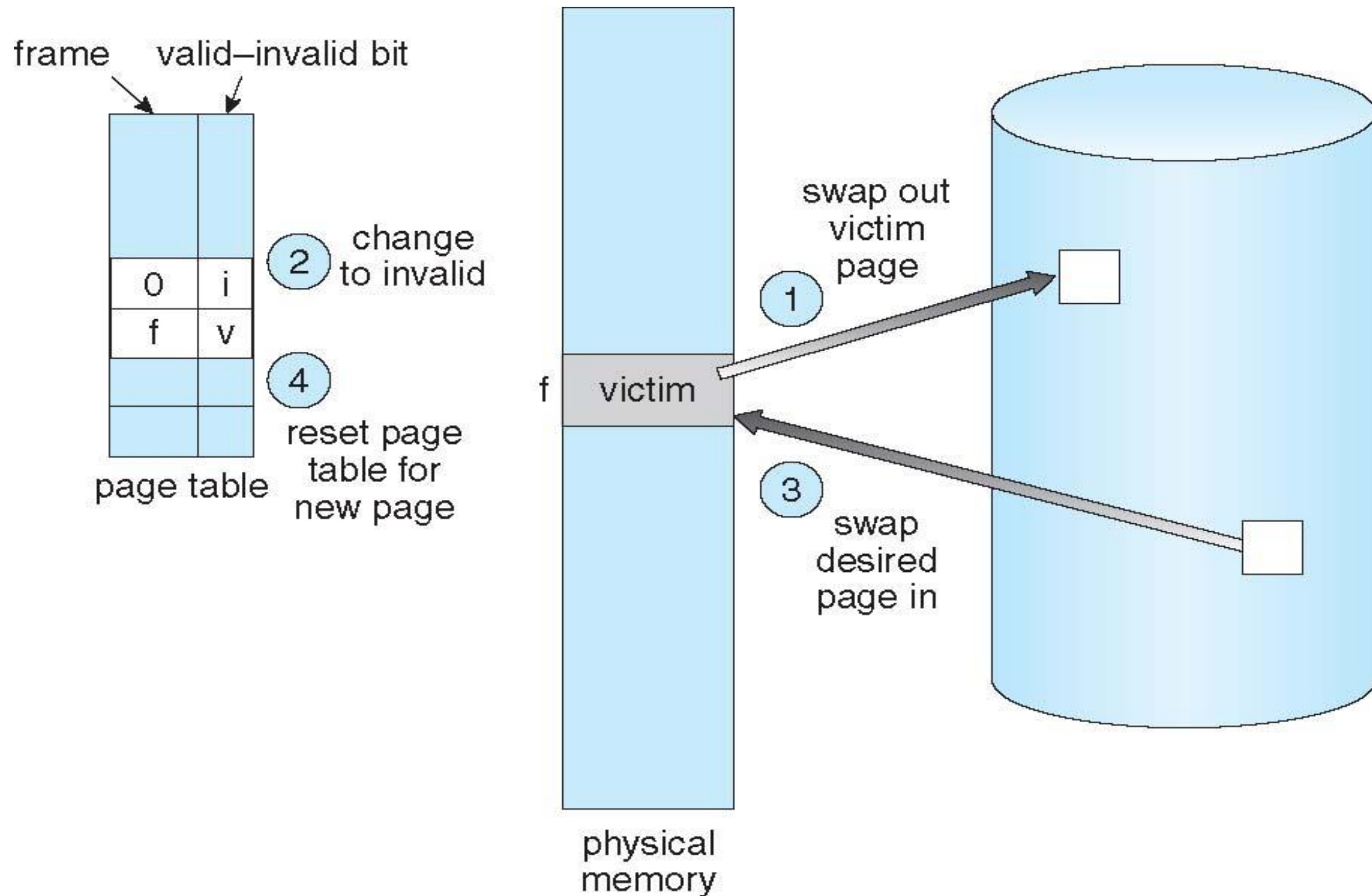
1. Find the location of the desired page on disk
2. Find a free frame:
 - If there is a free frame, use it
 - If there is no free frame, use a page replacement algorithm to select a **victim frame**
 - Write victim frame to disk if “dirty” (modified since last time it was brought into the memory)
3. 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

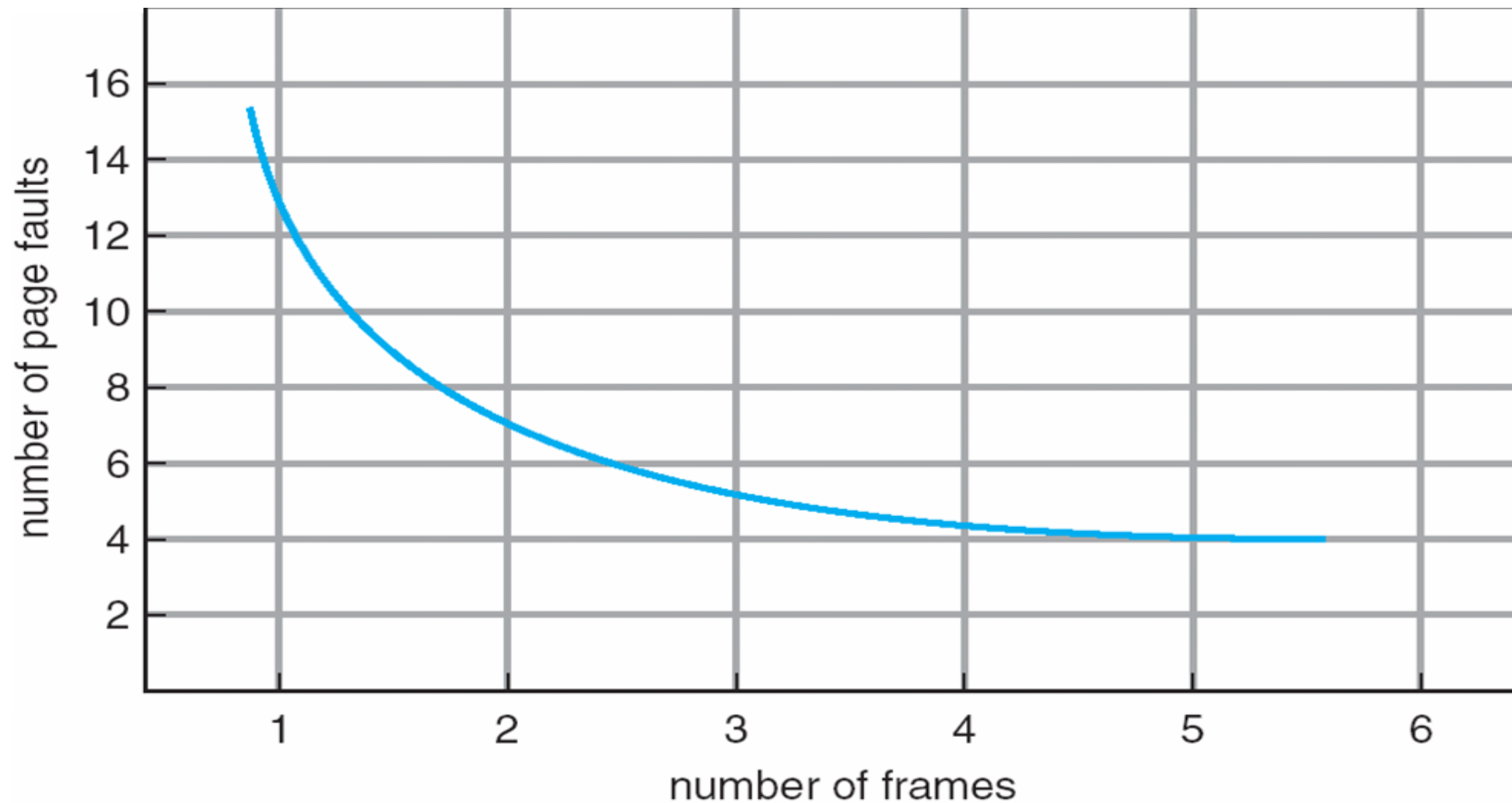
- **Frame-allocation algorithm** determines
 - How many frames to give each process
 - Which frames to replace
- **Page-replacement algorithm**
 - Want the lowest page-fault rate on both first access and re-access
- Evaluate algorithm by running it on a particular string of memory references (**reference string**) and computing the number of page faults on that string
 - String is just page numbers, not full addresses (no need)
 - Repeated access to the same page does not cause a page fault (thus does not show) – usually recent memory accesses are all within the same page, or a few (same) pages
 - Results depend on number of frames available
- 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)

reference string

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

7	7	7	2																		
	0	0	0																		
		1	1																		

page frames

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?
 - Easy - just use a FIFO queue





FIFO Page Replacement

reference string

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

3 frames

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

Page fault:
9

reference string

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

4 frames

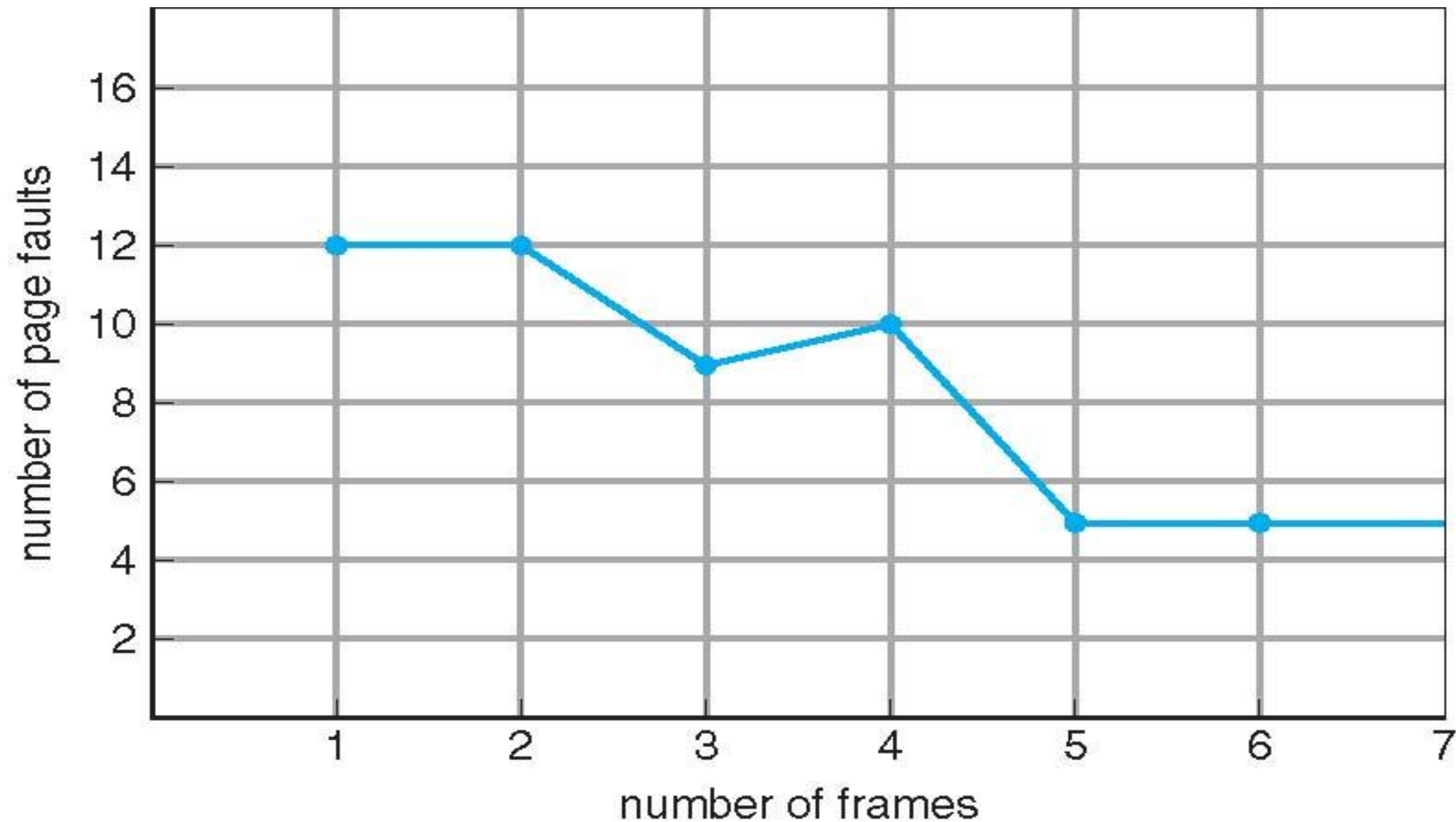
1	1	1	1			5	5	5	5	4	4
	2	2	2			2	1	1	1	1	5
		3	3			3	3	2	2	2	2
			4			4	4	4	3	3	3

Page fault:
10





FIFO Illustrating Belady's Anomaly





Optimal Algorithm

- Replace page that will not be used for longest period of time in the future
 - If possible, ideally select a page that will not be used in the future. In practice, this is not always feasible, so a page can be brought into the memory multiple times
 - 9 is the optimal for the example
- How do you know this?
 - Can't read the future
- Used for measuring how well your algorithm performs

reference string

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

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

page frames





Least Recently Used (LRU) Algorithm

- Use past knowledge rather than future as an approximation for OPT
- Replace page that has not been used in the most amount of time
- Associate time of last use with each page – complexity (update with each memory reference)

reference string

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

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

page frames

- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- But how to implement? - An LRU page-replacement algorithm may require substantial hardware assistance. The problem is to determine an order for the frames defined by the time of last use.





LRU Algorithm Implementation

- ❑ 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 the smallest value
 - ▶ Search through table needed to find the LRU page and a write to memory (to the time-of-use field in the page table) for each memory access
- ❑ 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 upon each reference
 - ❑ Each update more expensive, but no need to search for replacement
- ❑ LRU and OPT are cases of **stack algorithms** that don't have Belady's Anomaly
 - ❑ A **stack algorithm** can be shown (or proved) that the set of pages in memory for n frames is always a **subset** of the set of pages that would be in memory with $n+1$ frames.





Use Of A Stack to Record The Most Recent Page References

reference string

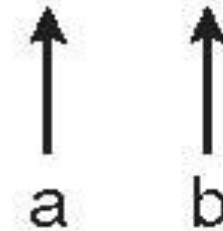
4 7 0 7 1 0 1 2 1 2 7 1 2



stack
before
a



stack
after
b





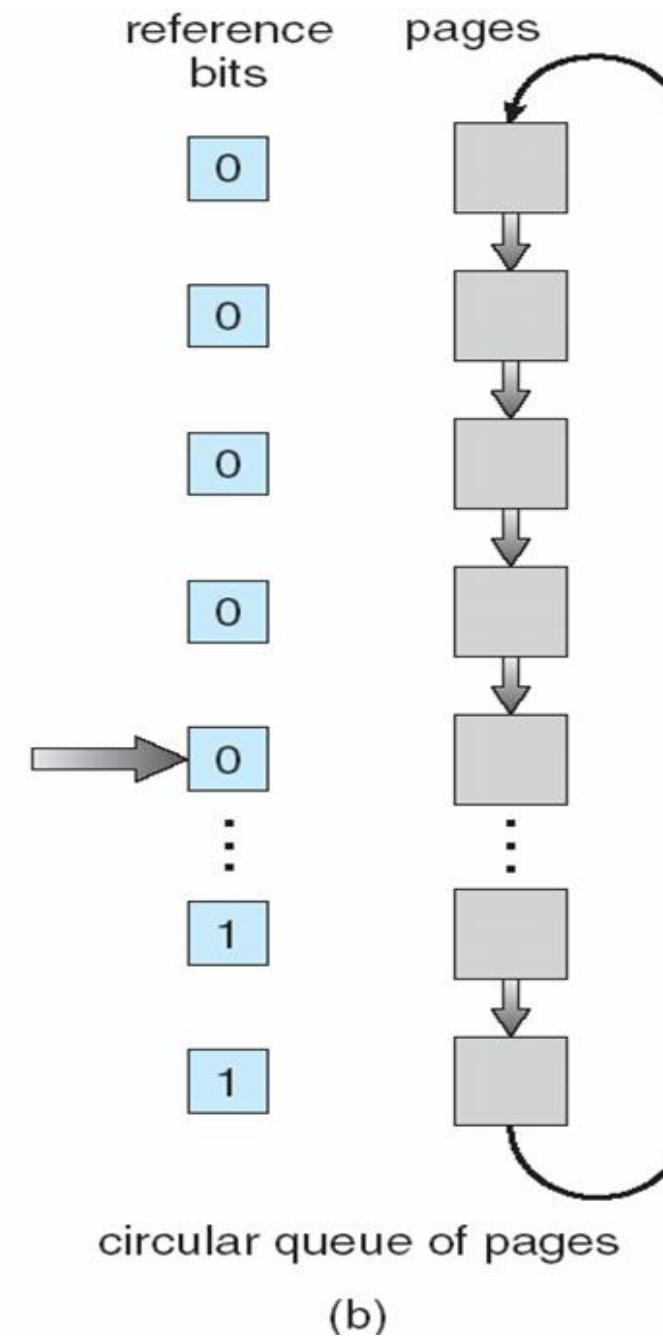
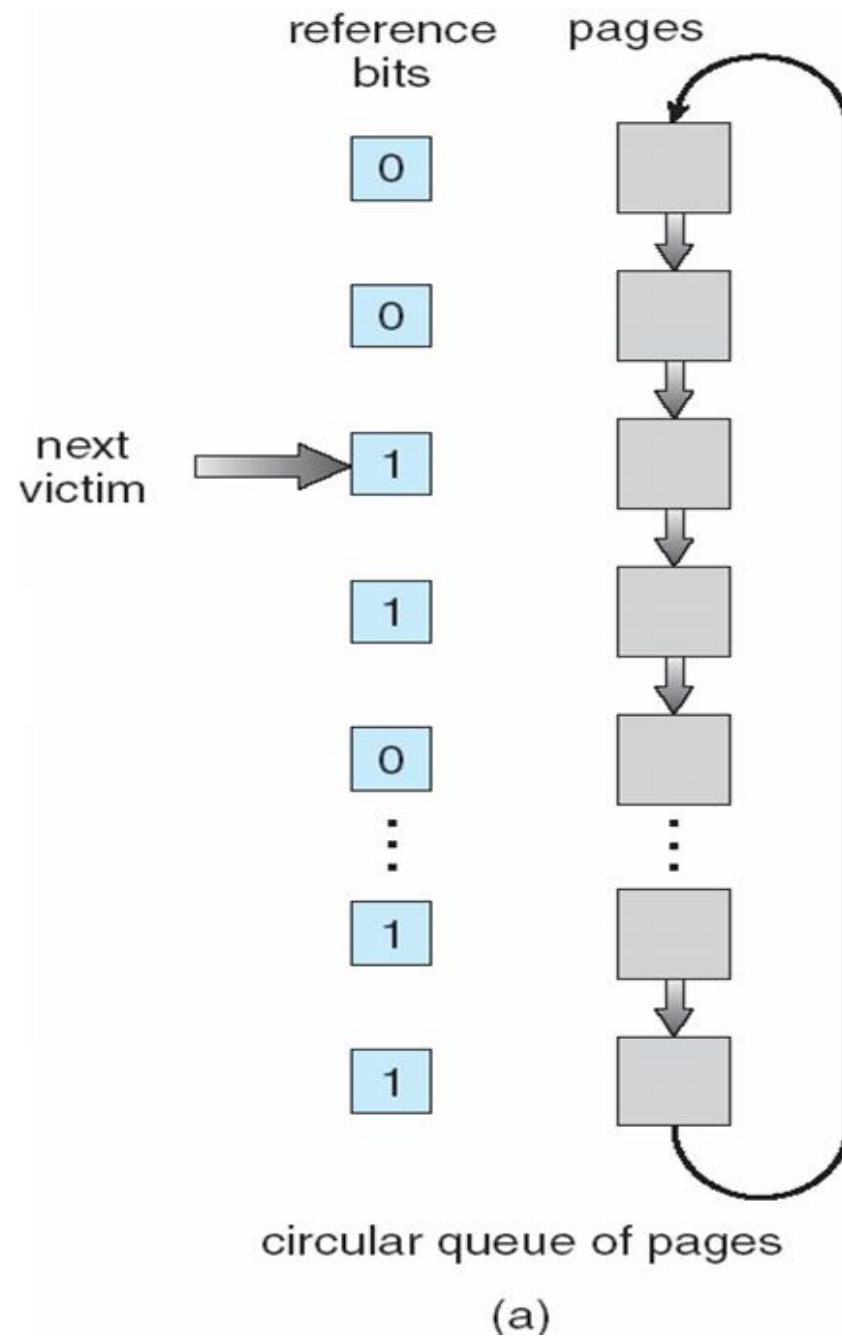
LRU Approximation Algorithms

- LRU needs special hardware and still slow (updates required for each memory reference)
- Reference bit
 - With each page associate a bit, initially = 0, associated with each entry in the page table
 - When page is referenced (read or write), the reference bit set to 1
 - Replace any with reference bit = 0 (if one exists)
 - ▶ We do not know the order of use, 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 (second chance)
 - replace next page, subject to same rules (FIFO and clock)





Second-Chance (Clock) Page-Replacement Algorithm





Counting Algorithms

- Keep a counter of the number of references that have been made to each page
- The **least frequently used (LFU)** Algorithm: replaces the page with the smallest count
- The **most frequently used (MFU)** Algorithm: replace the page with the largest count based on the argument that the page with the smallest count was probably just brought in and has yet to be used
- Neither LFU nor MFU replacement is commonly used. The implementation of such algorithms is expensive, and they do not approximate OPT replacement well





Allocation of Frames

- How do we allocate memory among different processes?
 - Does every process get the same fraction of memory? Different fractions?
 - Should we completely swap some processes out of memory?
- Each process needs certain *minimum* number of frames in order to execute its program
- 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*
- The maximum, of course, is the total frames required for a process
- 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
- **Proportional allocation** – Allocate according to the size of process
 - Dynamic as the degree of multiprogramming and process sizes change over the time

s_i = size of process p_i

$S = \sum s_i$

m = total number of frames

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

$$m = 64$$

$$s_1 = 10$$

$$s_2 = 127$$

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

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





Global vs. Local Allocation

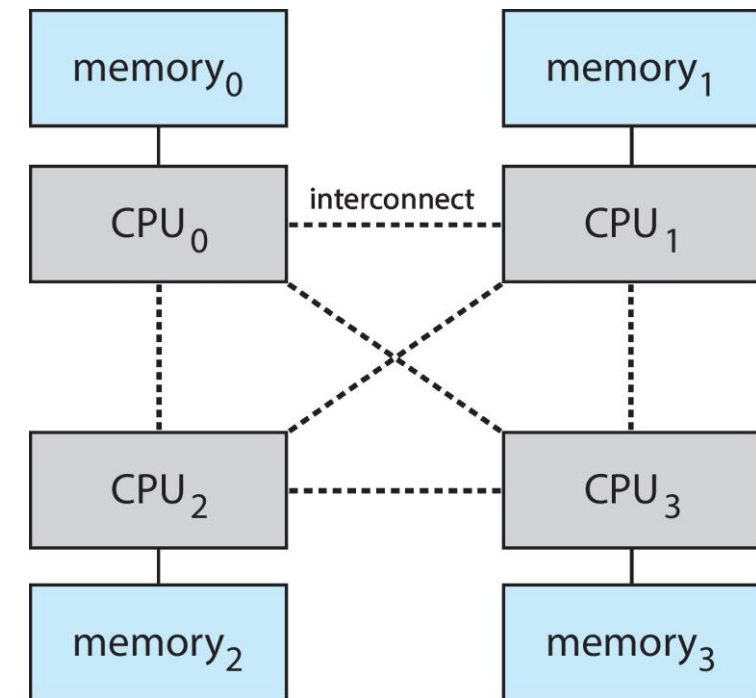
- **Global replacement** – process selects a replacement frame from the set of all frames, even if that frame is currently allocated to some other process; thus, one process can take a frame from another process
 - For instance, this can be based on priority – **priority allocation** in which a process can preempt memory from a lower priority process
 - This can result in better system throughput
 - But process execution time can vary greatly, as a process cannot control its own page-fault rate.
- **Local replacement** – each process selects from only its own set of allocated frames
 - What we used in the page replacement algorithms earlier
 - Set of page for a process is only affected by the paging behavior of only that process
 - More consistent per-process performance
 - But possibly underutilized memory, since pages allocated to a process can not utilized by another process, even if this page is not currently used by the process holding it





Non-Uniform Memory Access

- So far all memory accessed equally
- Many systems are **NUMA** – speed of access to memory varies



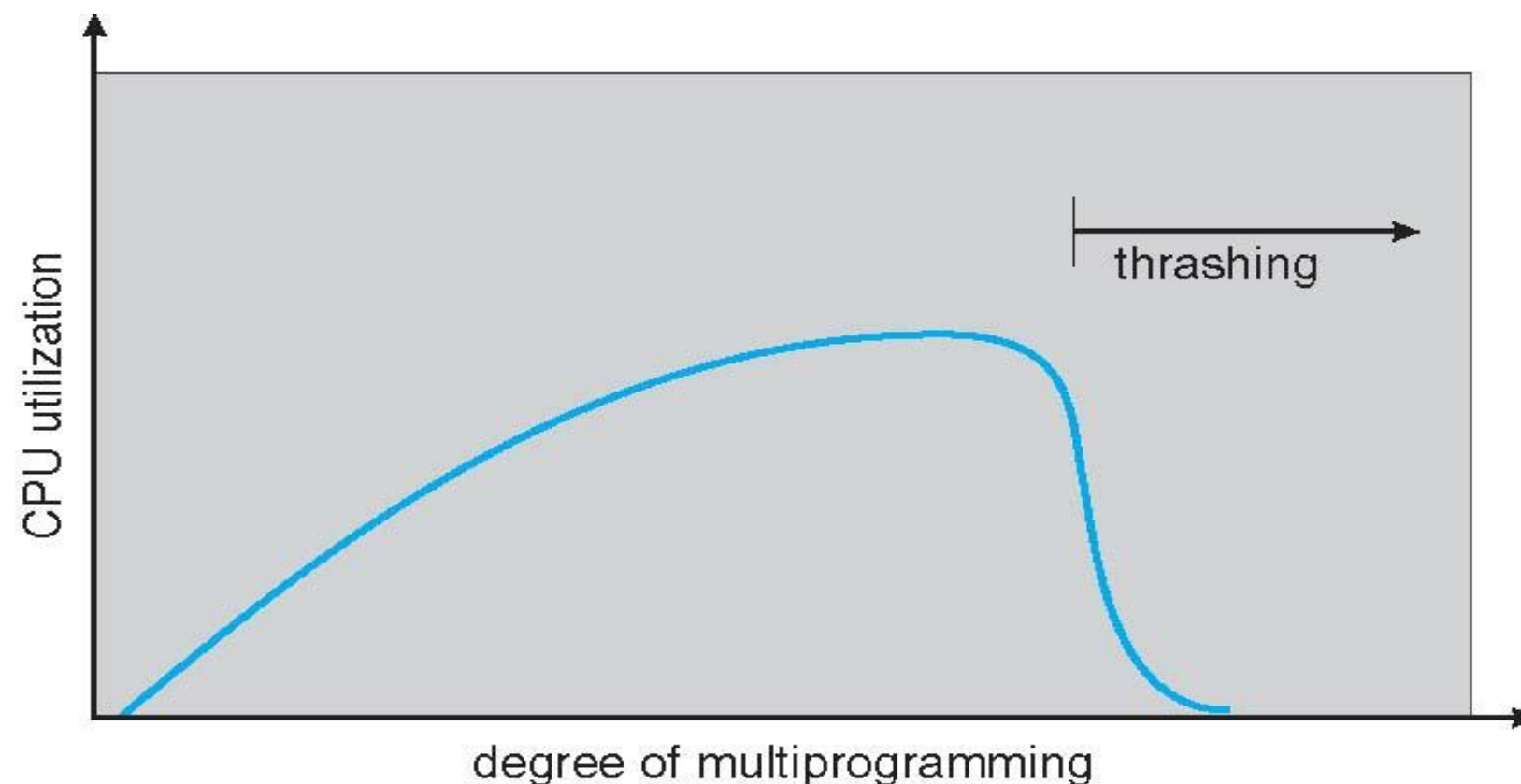
- Optimal performance - allocating memory “close to” the CPU on which the thread is scheduled
 - And modifying the scheduler to schedule the thread on the same system board when possible
 - Solved by Solaris by creating **lggroups** – locality groups
 - ▶ Structure to track CPU / Memory low latency groups
 - ▶ Each lgroup gathers together CPUs and memory, and each CPU in that group can access any memory in the group within a defined latency interval
 - ▶ 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, and replace an existing frame
 - But quickly need replaced frame back
 - This leads to low CPU utilization
 - ▶ Operating system thinking “by mistakes” that it needs to increase the degree of multiprogramming in order improve the CPU utilization
 - ▶ Another process added to the system – aggravate the problem
- **Thrashing** ≡ a process or a set of processes is busy swapping pages in and out





Demand Paging and Thrashing

□ Why does demand paging work?

Locality model

- A locality is a set of pages that are actively used together. A running program is generally composed of several different localities, which may overlap
- Memory access or subsequent memory access tends to stay in the same page(s)
- Process migrates from one locality to another, e.g., operating on a different set of data
- Localities may overlap, instructions or part of the data being manipulated

□ Why does thrashing occur?

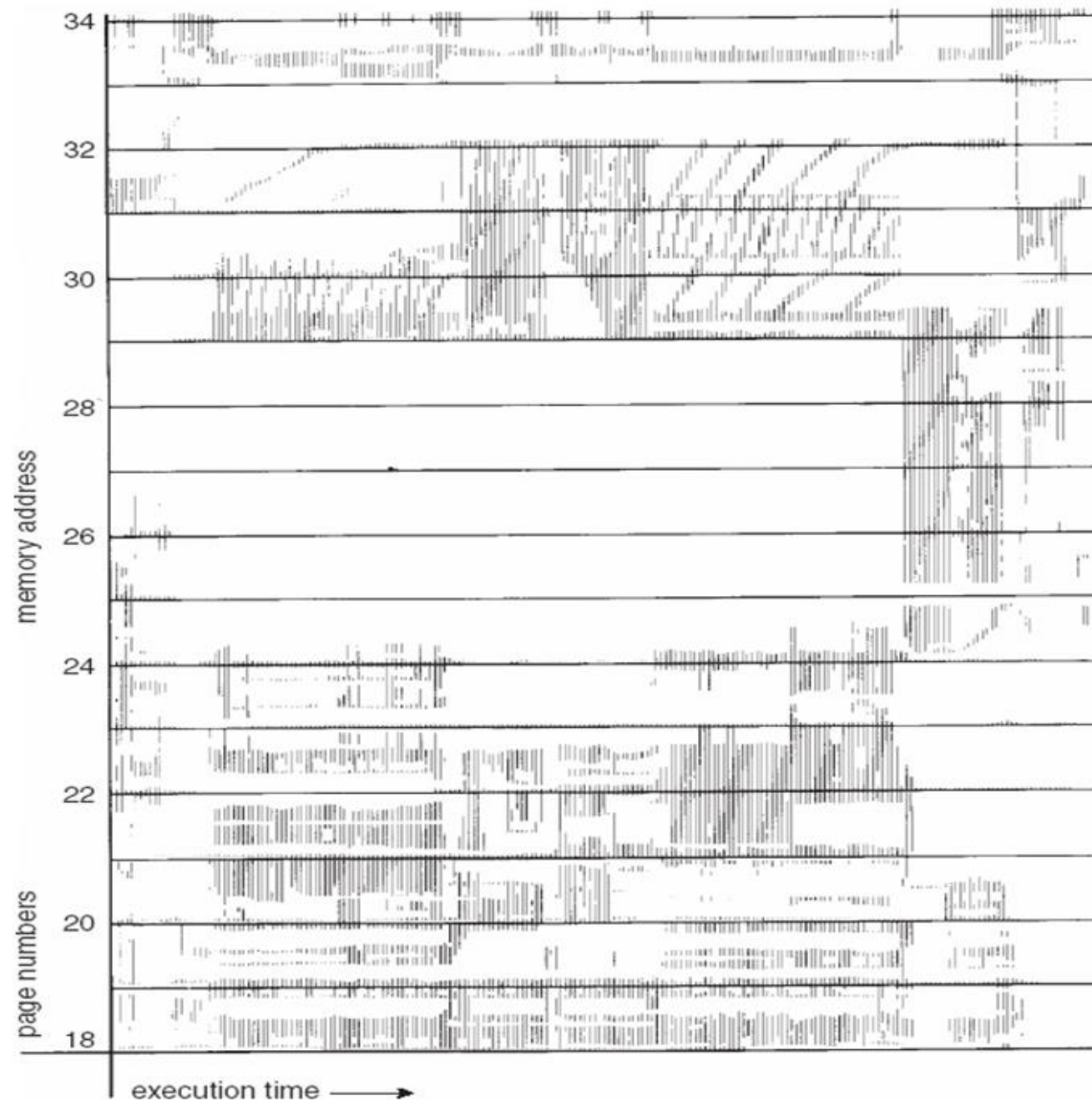
Σ size of locality (of all processes) > total memory size

- We can limit effects by using local or priority page replacement, as thrashing in one process can not steal frames from another process and cause the latter to thrash as well





Locality In A Memory-Reference Pattern



- Recall program memory access patterns exhibit temporal and spatial locality
- The left Figure illustrates the concept of locality and how a process's locality changes over time. At time (a), the locality is the set of pages {18, 19, 20, 21, 22, 23, 24, 29, 30, 33}. At time (b), the locality changes to {18, 19, 20, 24, 25, 26, 27, 28, 29, 31, 32, 33}. Notice the overlap, as some pages (for example, 18, 19, and 20) are part of both localities.





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
- $D = \sum WSS_i \equiv$ total demand frames
 - Approximation of the current locality in the system (of all processes)
- if $D > m \Rightarrow$ Thrashing – at least one process is short of memory
- Policy if $D > m$, then suspend or swap out one of the processes
- The working-set strategy prevent thrashing while keeping the degree of multiprogramming as high as possible, thus optimizes CPU utilization

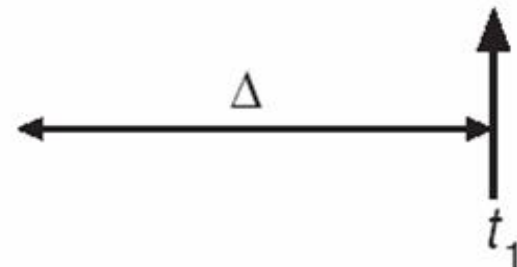




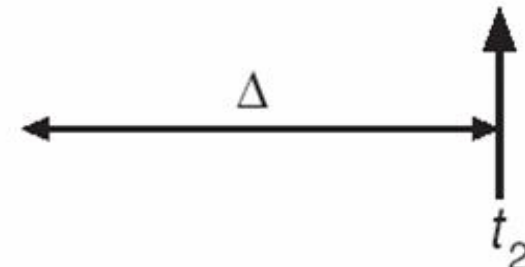
Working-Set Model (Cont.)

page reference table

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



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



$$WS(t_2) = \{3, 4\}$$





Keeping Track of the Working Set

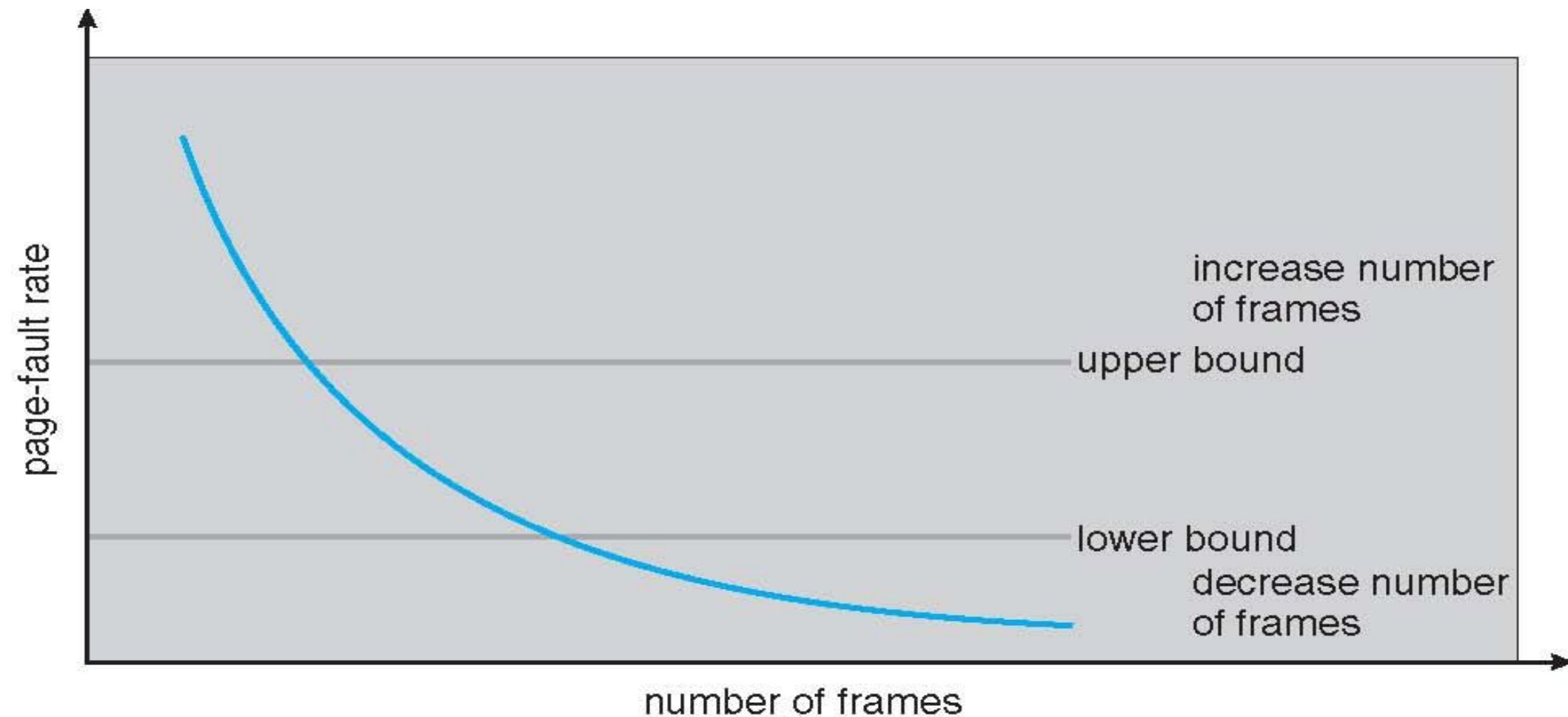
- It is difficult to keep track of the working set, as working-set window is a moving window which needs to be updated for each memory reference
- Approximate with interval timer + a reference bit
- Example: $\Delta = 10,000$
 - Timer interrupts after every 5000 time units
 - Keep in memory 2 bits for each page, (0,0), (0,1),(1,0),(1,1)
 - 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
- This is not completely accurate, as we cannot tell where, within an interval of 5,000, a reference occurred
- Improvement = 10 bits and interrupt every 1000 time units, more accurate but cost is higher
- Accuracy versus complexity





Page-Fault Frequency

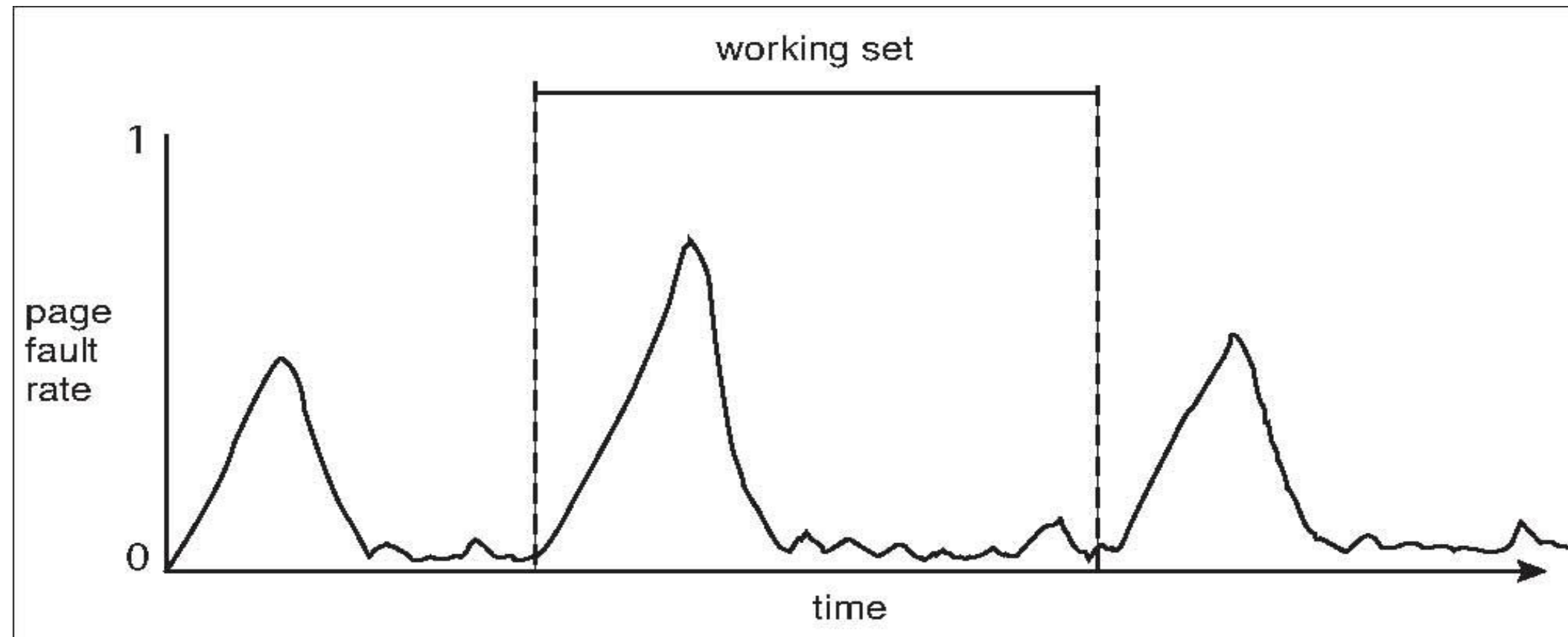
- More direct approach than WSS
- Establish “acceptable” **page-fault frequency (PFF)** rate and use local replacement policy
 - If actual rate too low, process loses frame
 - If actual rate too high, process gains frame





Working Sets and Page Fault Rates

- Direct relationship between working set of a process and its page-fault rate
- Working set changes over time
- The page-fault rate of the process will transition between peaks and valleys over time.





Allocating Kernel Memory

- Pages are allocated from the list of free page frames maintained by the kernel when a process in user mode requests additional memory
 - Pages scattered throughout the physical memory, and suffers from internal fragmentation
- Kernel memory is often allocated from a free-memory pool differently from the list used to satisfy user-mode processes, because:
 - Kernel requests memory for structures of varying sizes, some of which less than a page
 - Some kernel memory needs to be contiguous. For instance, certain hardware devices interact directly with physical memory—without the benefit of a virtual memory interface
- Two strategies for managing free memory for kernel processes - the “buddy system” and slab allocation





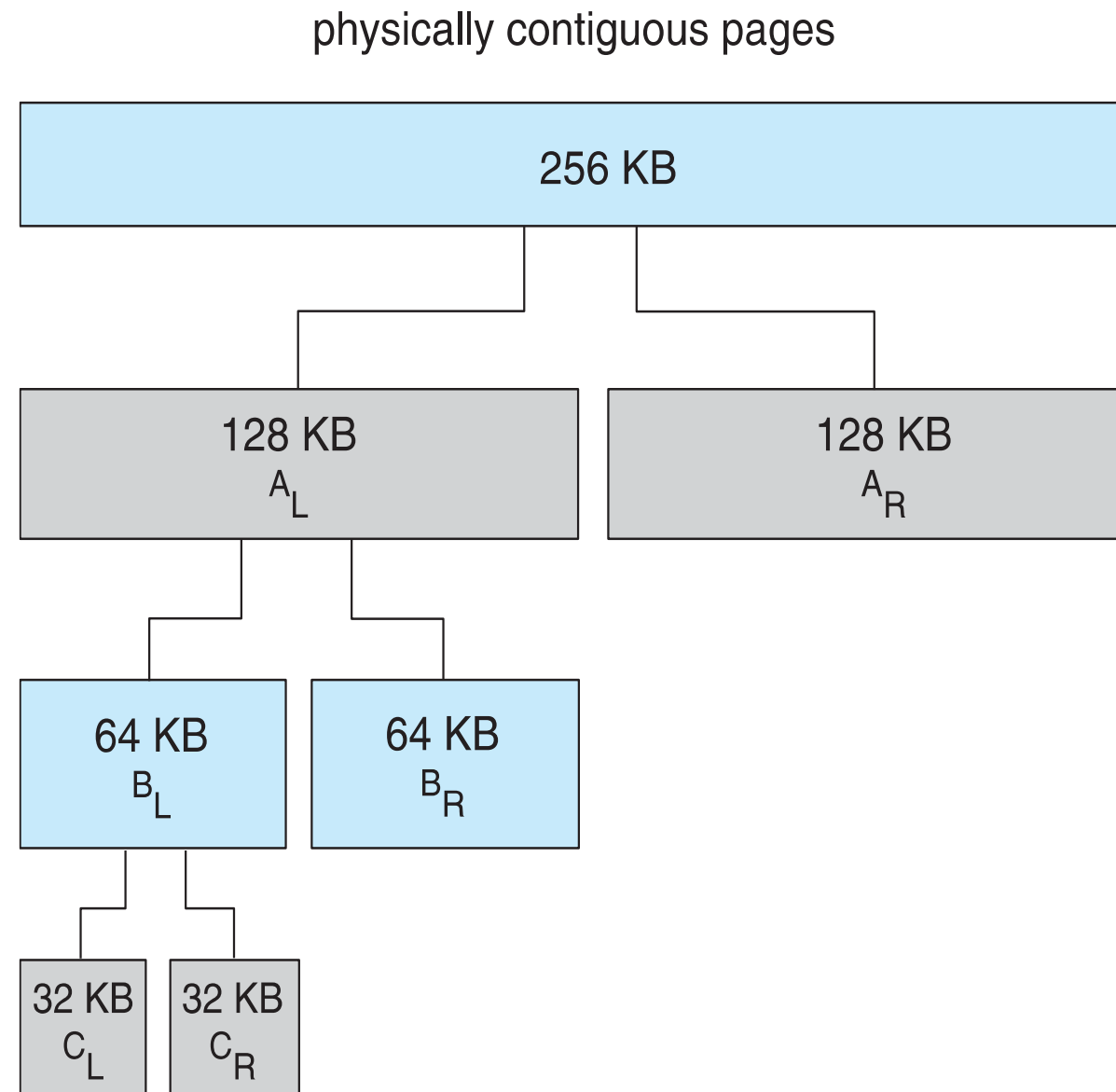
Buddy System

- ❑ 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. For example, a request for 11 KB request is satisfied by a 16-KB segment
 - ❑ 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
 - ❑ Quickly combine adjacent buddies to form larger segments
- ❑ Disadvantage – internal fragmentation





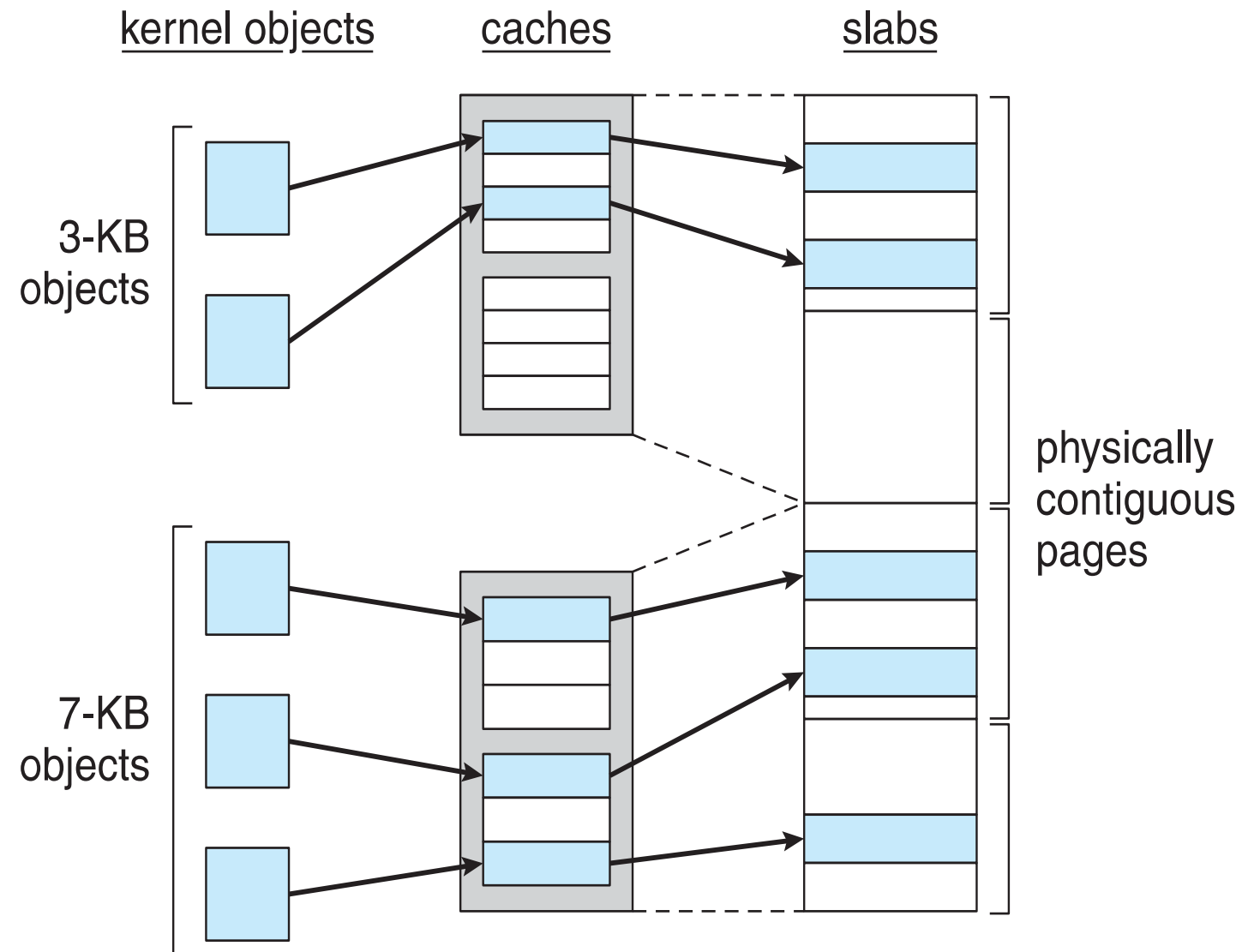
Buddy System Allocator





Slab Allocator

- ❑ **Slab** is made up of one or more physically contiguous pages
- ❑ **Cache** consists of one or more slabs
- ❑ There is a single cache for each unique kernel data structure
 - ❑ Caches for process descriptors, for file objects, for semaphores
 - ❑ Each cache filled with **objects** – instantiations of the data structure
- ❑ The figure shows two kernel objects of 3 KB in size and three objects of 7 KB in size, each stored in a separate cache





Slab Allocation

- ❑ Caches to store kernel objects. When a cache is created, a number of objects—which are initially marked as **free** - are allocated to the cache.
- ❑ The number of objects in a cache depends on the size of the associated slab.
 - ❑ A 12-KB slab (made up of three contiguous 4-KB pages) could store six 2-KB objects
- ❑ When a new object for a kernel data structure is needed, the allocator can assign any free object from the cache to satisfy the request. The object assigned is marked as **used**
- ❑ If slab is full of used objects, next object allocated from an empty slab
 - ❑ If no empty slabs, new slab allocated
- ❑ Two benefits include no fragmentation, and fast memory request satisfaction
 - ❑ Each cache is made up of one or more slabs of chunk size of the objects being represented. Thus, the slab allocator returns the exact amount of memory required to represent the object.
 - ❑ Objects are created in advance and thus can be quickly allocated from the cache. Released objects are marked as **free** and returned to its cache, thus making it immediately available for subsequent requests from the kernel.





Slab Allocator in Linux

- ❑ For example process descriptor is of type `struct task_struct`
- ❑ This requires an approximate 1.7KB of memory
- ❑ When Linux creates a new task, it requests memory for `struct task_struct` object from its cache. The cache fulfill the request using a `struct task_struct` object that has already been allocated in a slab and is marked as **free**.

- ❑ In Linux, a slab can be in one of the three possible states:
 1. **Full** – All objects in the slab are marked as used
 2. **Empty** – All objects in the slab are marked as free
 3. **Partial** – The slab consists of both used and free objects

- ❑ Upon request, slab allocator
 1. first attempts to use a free object in a *partial* slab
 2. If none exists, a free object is assigned from an *empty* slab
 3. If no empty slab, a new slab is allocated from contiguous physical pages and assigned to a cache





Other Considerations

- Prepaging
- Page size
- TLB reach
- Program structure





Prepaging

- To reduce the large number of page faults that occurs at process startup
- Prepage some of the pages a process will need, before they are referenced
- But if prepaged pages are unused, I/O and memory was wasted
- Assume s pages are prepaged and α of the pages is used
 - Is cost of $s * \alpha$ save pages faults $>$ or $<$ than the cost of prepaging $s * (1 - \alpha)$ unnecessary pages?
 - α near zero \Rightarrow prepaging loses





Page Size

- Sometimes OS designers have a choice
 - Especially if running on custom-built CPU
- Page size selection must take into consideration a conflicting set of criteria:
 - Fragmentation – calls for smaller page size
 - Page table size – calls for larger page size
 - Resolution – isolate the memory actually be used
 - I/O overhead – larger page size requires longer I/O time
 - Number of page faults – smaller page size can increase the number of page faults
 - Locality – ideally each page should match the current locality
 - TLB size and effectiveness – larger page size improves the TLB reach
- 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





TLB Reach

- TLB Reach - The amount of memory accessible from the TLB
- $\text{TLB Reach} = (\text{TLB Size}) \times (\text{Page Size})$
- Ideally, the working set of each process is stored in the TLB
 - Otherwise there might be a high degree of page faults, or the access time is slow down
- 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





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



End of Chapter 9

