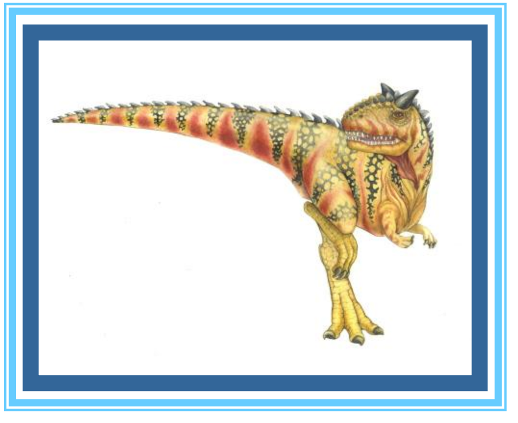
Chapter 9: Virtual Memory



Chapter 9: 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 the benefits of a virtual memory system
  - **Goal of memory-management strategies**: keep many processes in main memory to allow multi-programming; see Chap-8
    - **Problem**: Entire processes must be in memory before they can execute
  - **Virtual Memory** technique: running process need not be in memory entirely
    - Programs can be larger than physical memory
    - Abstraction of main memory; need not concern with storage limitations
    - Allows easy sharing of files and memory
    - Provide efficient mechanism for process creation
- To explain the concepts of demand paging, page-replacement algorithms, and allocation of page frames
- To discuss the principle of the working-set model
- To examine the 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; are all seldom used
    - Ex: declared array of size 100 cells but only 10 cells are used
- Entire program code not needed (in main memory) at the 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
    - Thus, more [partially-loaded] programs can run at the same time
    - Increased CPU utilization and throughput with no increase in response time or turnaround time; more multi-programming and time-sharing
  - Less I/O needed to load or swap programs into/from memory
    - Thus, each user program would run faster

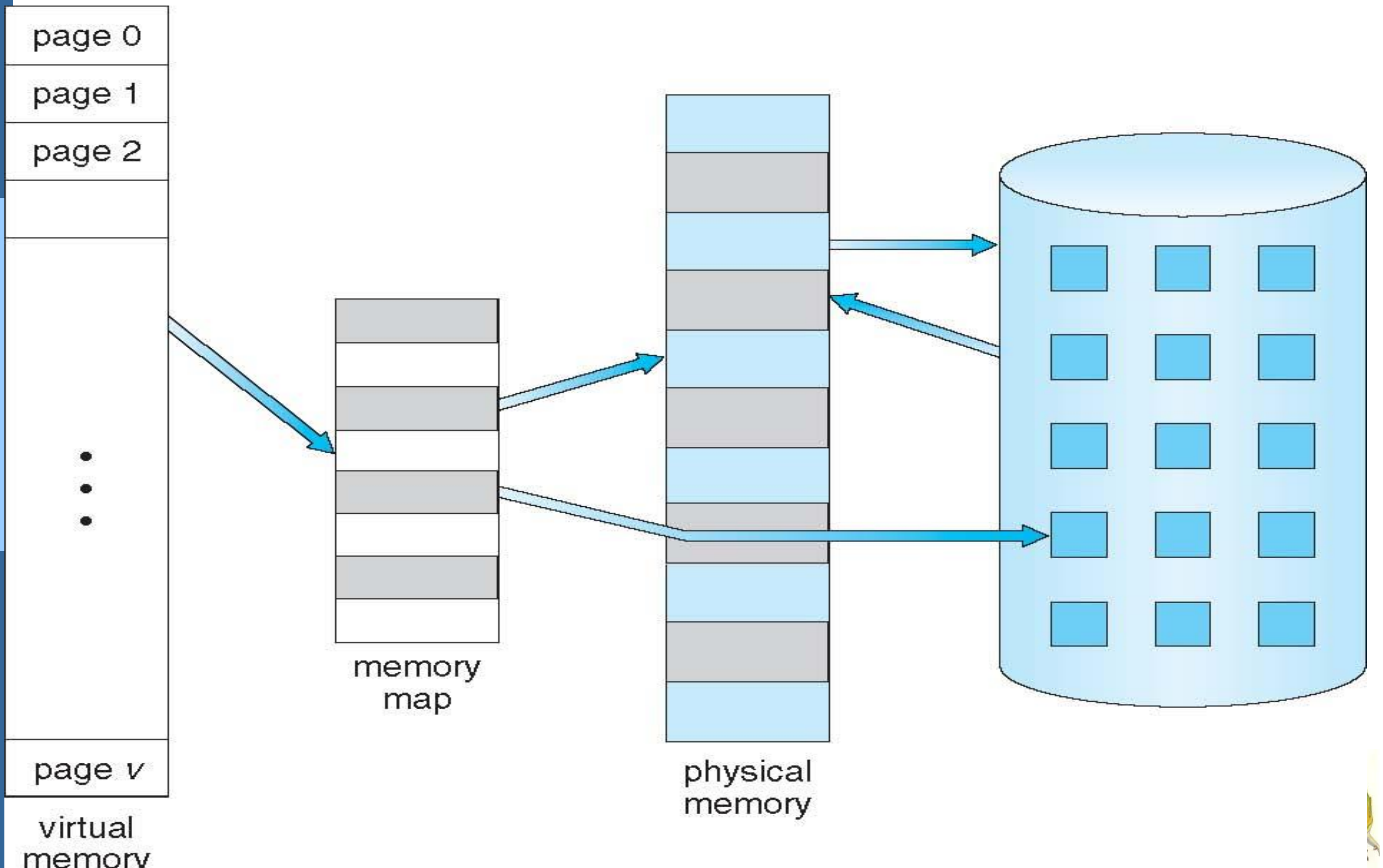
Background

- **Virtual memory** – separation of user logical memory from physical memory
    - As perceived by users; **that programs exist in contiguous memory**
    - Abstracts physical memory: need not worry about memory requirements
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
    - Programmers can work as if memory is an unlimited resource
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation
  - More programs running concurrently; increased multi-programming and/or time-sharing
  - Less I/O needed to load or swap processes; hence, faster program execution

Background (Cont.)

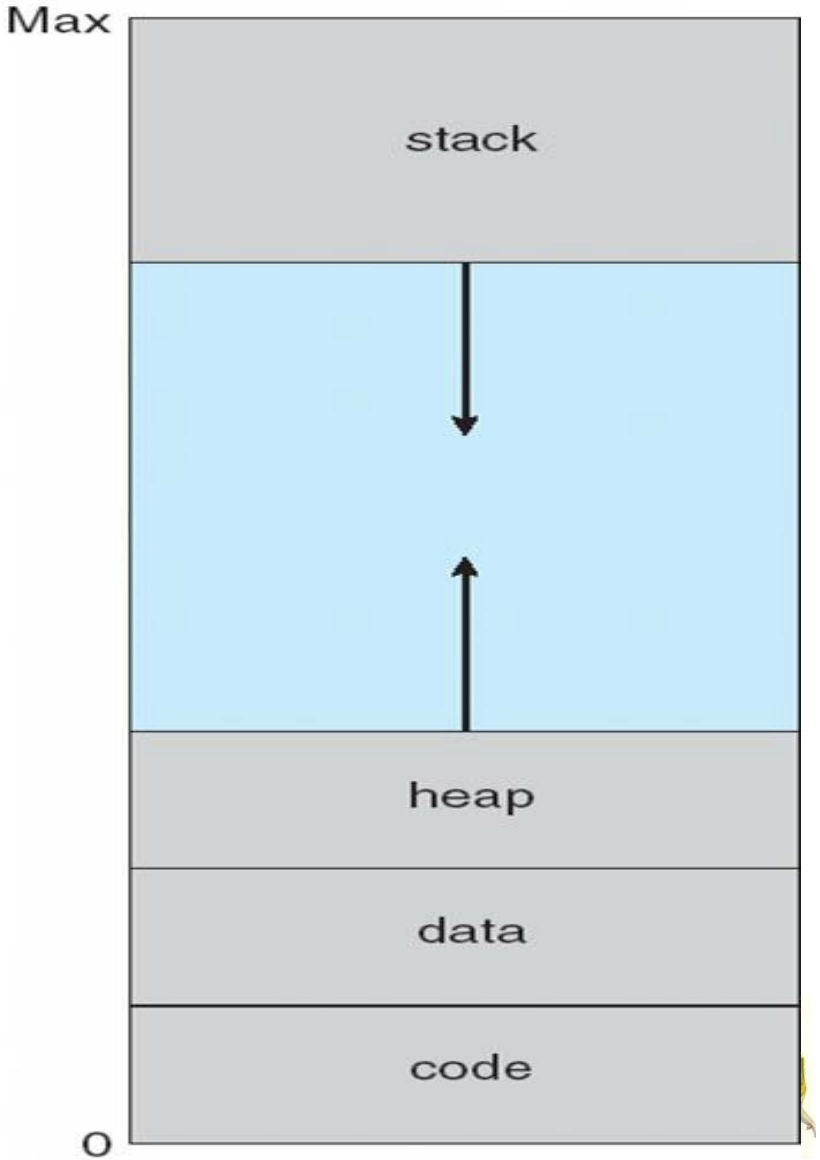
- **Virtual address space** – logical view of how process is stored in memory
  - Process starts at address 0 with contiguous addresses until end of its address space
  - Meanwhile, physical memory organized in page frames; not contiguous (see Chap-8)
  - MMU maps logical pages to physical pages (i.e., frames) in memory
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

Virtual Memory That is Larger Than Physical Memory

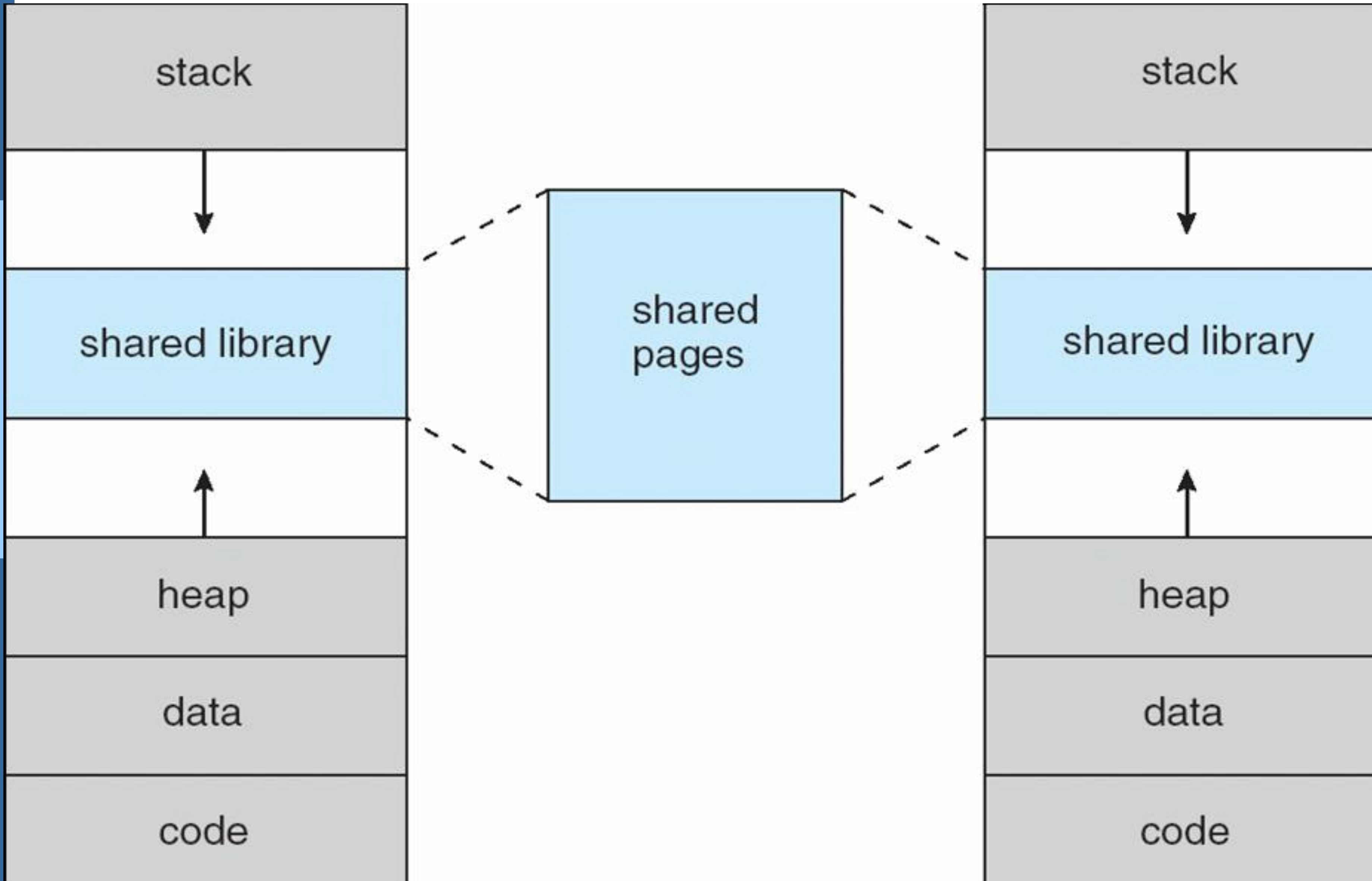


Virtual-Address Space of a Process

- For a process: heap grows upward while stack grows downward in memory; in process's space
    - Unused address space between the two is a hole; part of **virtual-address space**
        - Require actual physical pages only if the heap or the stack grow
    - Maximizes address space use
- Enables **sparse** address spaces with holes left for growth, or to dynamically link libraries, etc
- System libraries can be shared by many processes through mapping of the shared objects into virtual address space
- Processes can share memory by mapping read-write pages into virtual address space
- Pages can be shared during process creation with the `fork()`; speeding up process creation

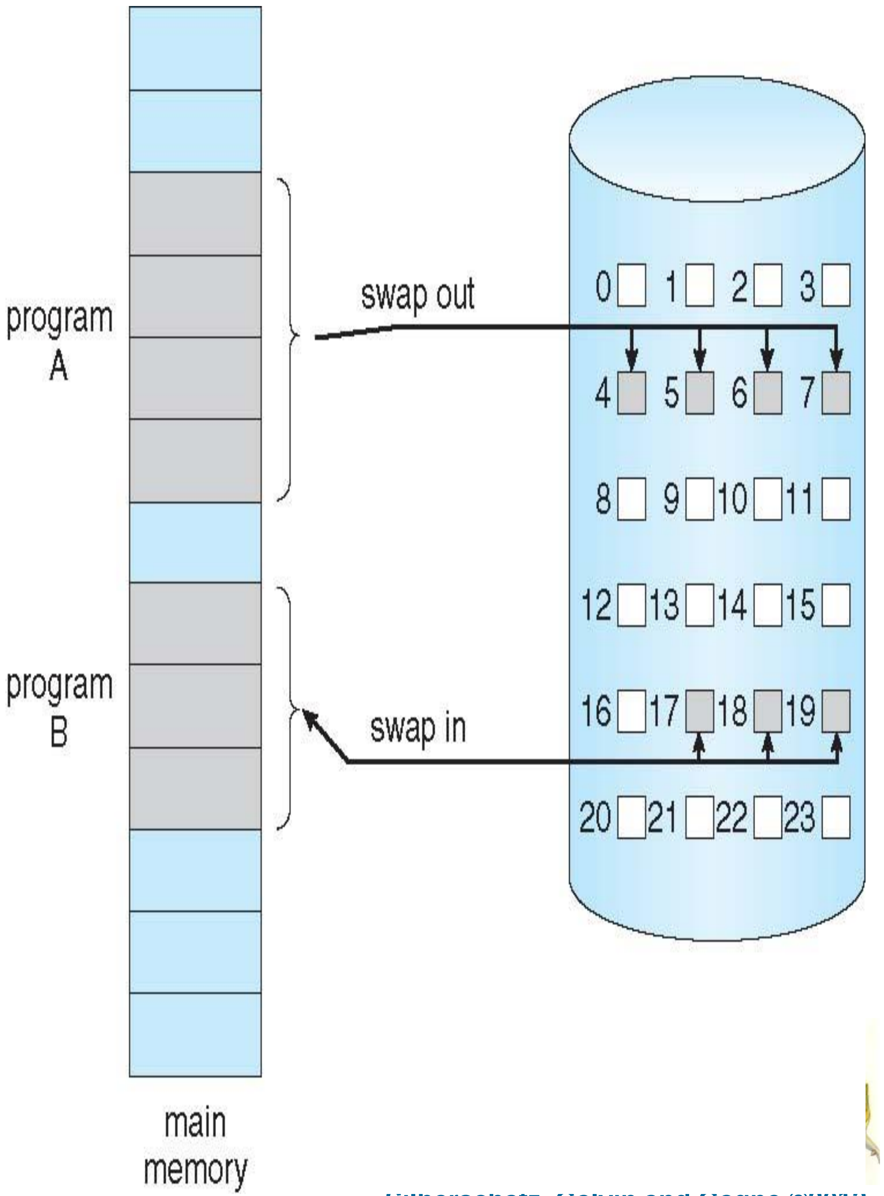


Shared Library Using Virtual Memory



Demand Paging

- Could bring an entire process into memory at load time.
- Or bring a process's 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 a paging system with swapping (diagram on right)
- **Lazy swapper** – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**
- Page is needed ⇒ reference it; see Slide-14
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory

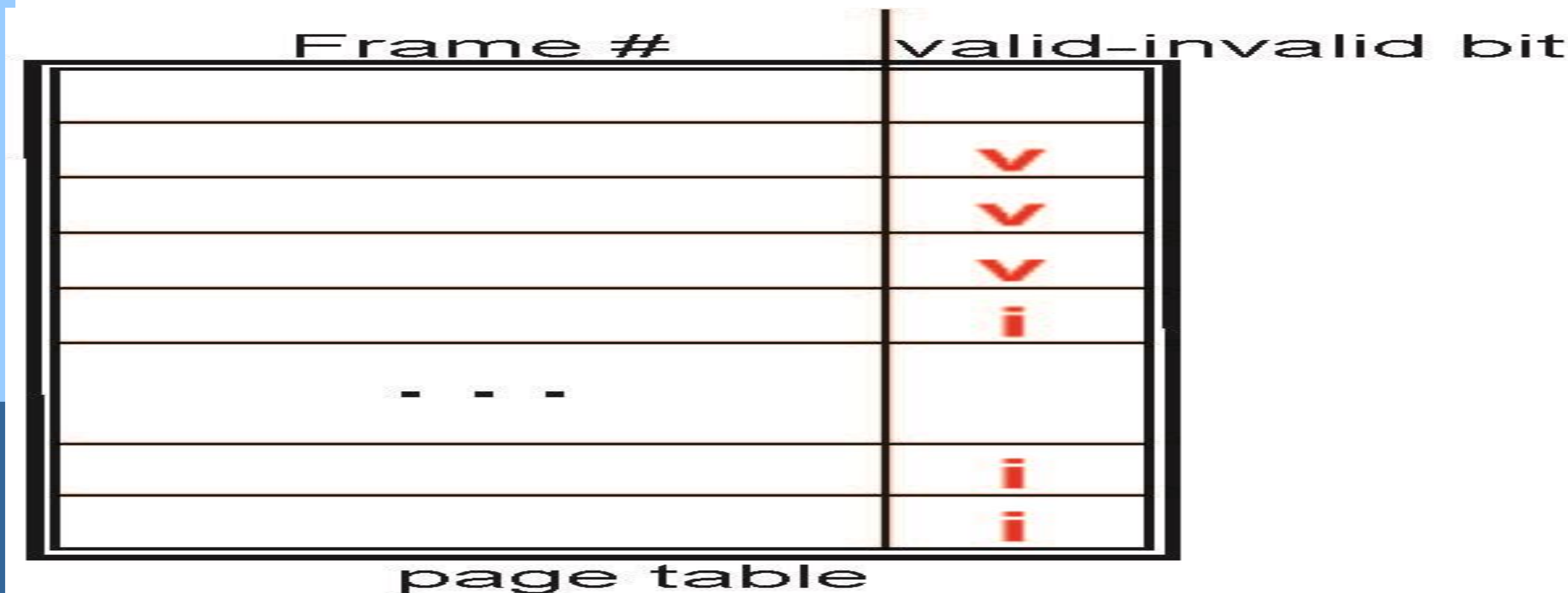


Basic Concepts

- When swapping in a process, the pager guesses which pages will be used before swapping it out again
  - The pager brings in only those **needed** pages into memory
    - Thus, decreases swap time and amount of needed physical memory
- Need new MMU hardware support to implement demand paging; see Slide-15
  - To distinguish between **in-memory** pages and **on-disk** pages
    - Uses the **valid—invalid scheme** of Slide-40 Chap-8
- If pages needed are already **memory resident**
  - Execution proceeds normally
- If page needed and is not memory resident; see Slide-14
  - Need to find the needed page from the disk and load it into memory
    - Without changing program behavior
    - Without programmer needing to change code

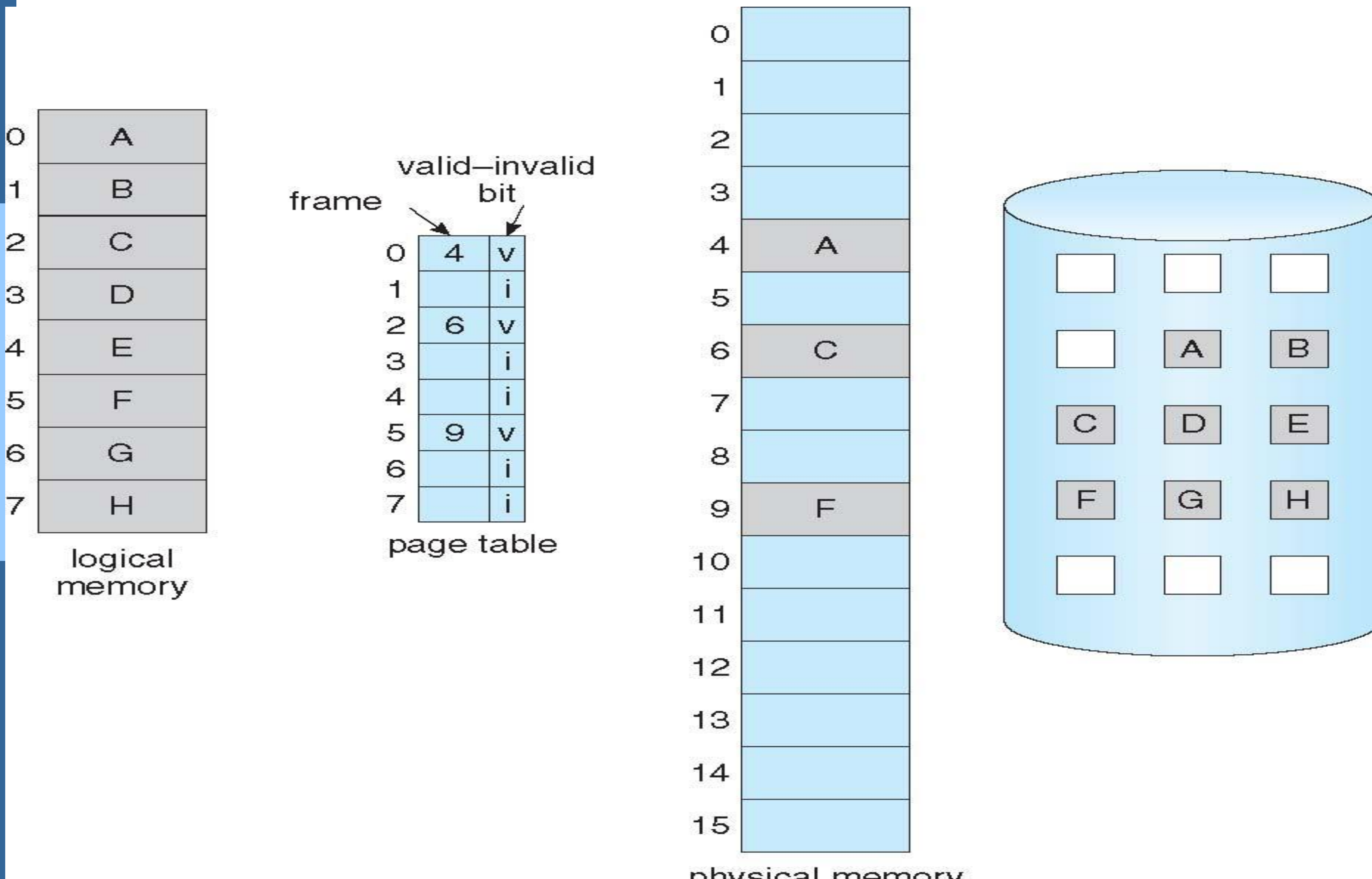
Valid-Invalid Bit

- A valid–invalid bit is associated with each page-table entry; see Chap-8, Slide-40 (**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:



- During MMU address translation:
  - if valid–invalid bit in page-table entry is **i** ⇒ there is a page fault

Page Table When Some Pages Are Not in Main Memory



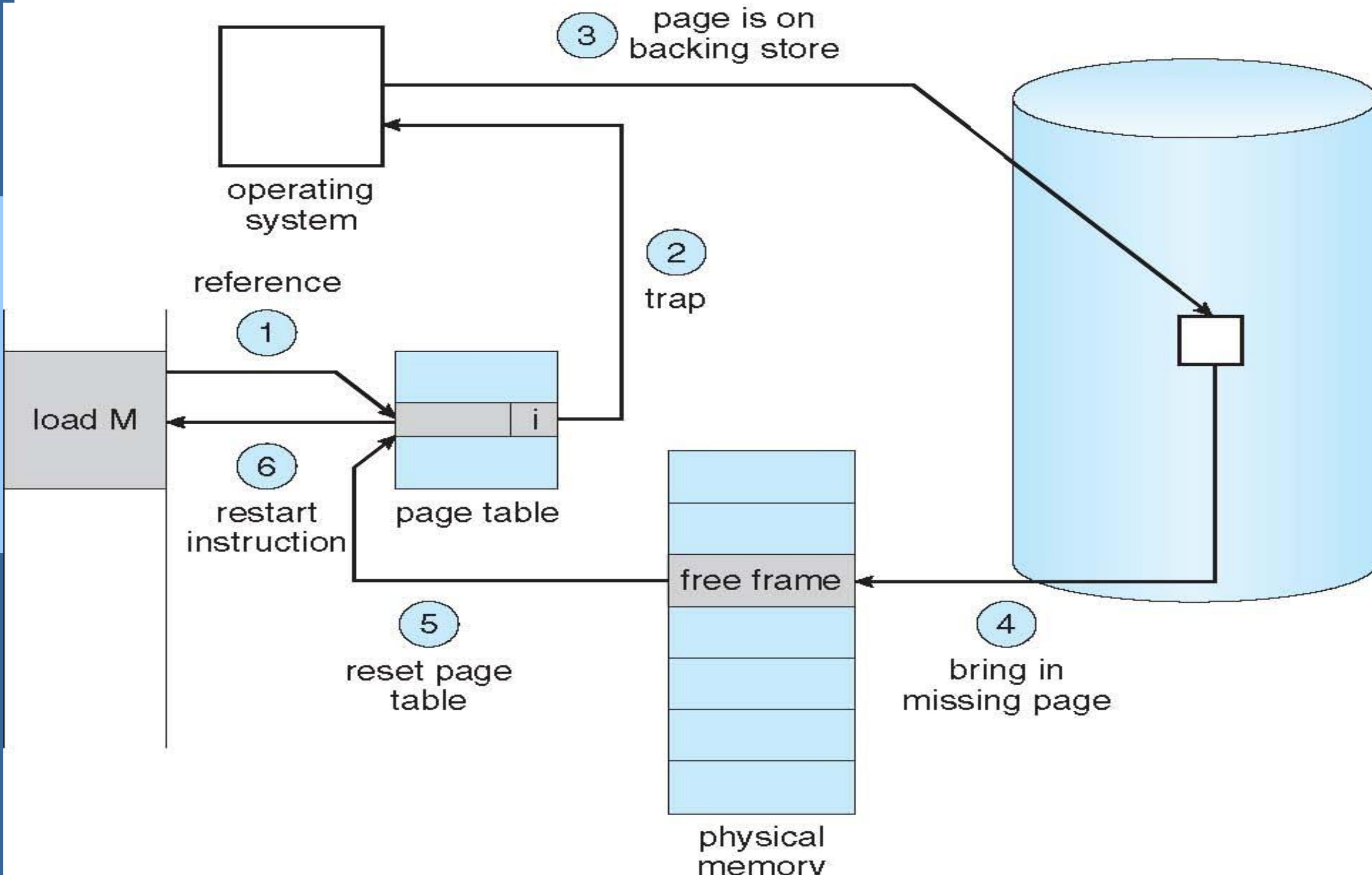
Page Fault

- What if the process refers to (i.e., tries to access) a page not in-memory ?
  - The [first] reference (i.e., address) to that **invalid** page will trap to operating system and causes a **page fault**

- **Procedure for handling a page fault**

1. OS checks an internal table to see if reference is valid or invalid memory access
2. If
   - Invalid reference ⇒ abort the process
     - address is not in logical address space of process
   - Just not in memory ⇒ page in the referred page from the disk
     - logical address is valid but page is simply not in-memory
3. Find a free frame; see Chap-8
4. Read the referred page into this allocated frame via scheduled disk operation
5. Update both internal table and page-table by setting validation bit = **v**
6. Restart the instruction that caused the page fault and resume process execution

Steps in Handling a Page Fault

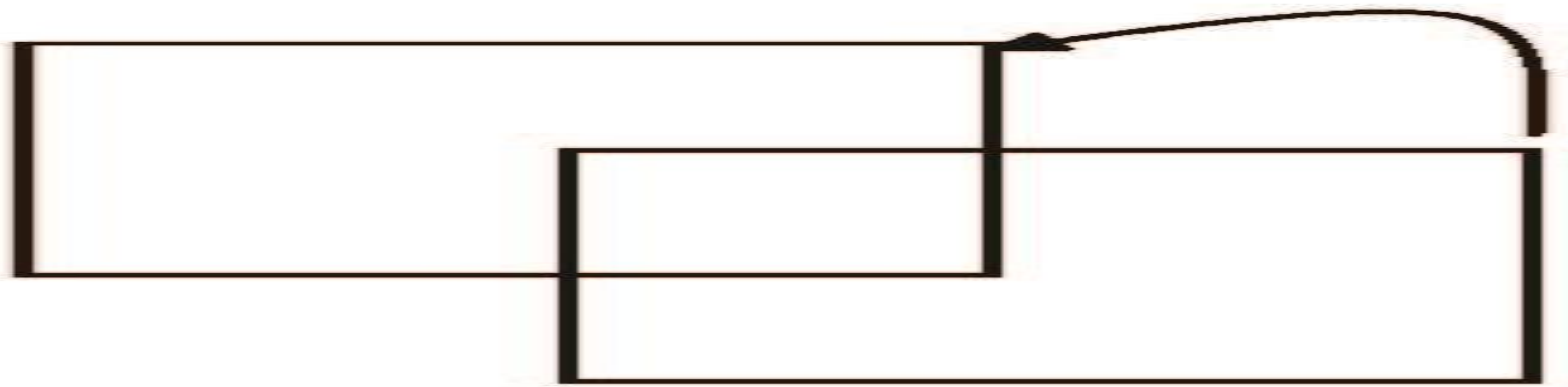


Aspects of Demand Paging

- Extreme case – start process with ***no*** pages in memory
  - OS sets instruction pointer to first instruction of process; logical address = (*p*, *d*)
    - Since page *p* is non-memory-resident then a page fault is issued
    - Page *p* is loaded and… same for all other process pages on first reference
  - This scheme is **pure demand paging**: load a page only when it is needed
- A given instruction may refer to multiple distinct pages; thus, multiple page faults
  - Consider fetching the instruction "`ADD A, B`" and fetching the values of data `A` and `B` from memory and then storing the result back to memory
    - Addresses of "**`ADD A, B`**", "**`A`**", and "**`B`**" may all be in three different pages
  - Multiple page fault per instruction results in unacceptable performance
    - Very unlikely, fortunately, due to **locality of reference**; see Slide-51
- Hardware support needed for demand paging; same as hardware for paging and swapping
  - **Page table** with valid / invalid bit, or special protection bits
  - **Secondary memory**: swap device with **swap space**; for not in-memory pages
  - **Instruction restart**; ability to restart any instruction after a page fault

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

- What is the **E**ffective **A**ccess **T**ime in demand paging? (worst case number of steps)

1. Trap to the operating system
2. Save the user registers and process state
3. Determine that the interrupt was a page fault
4. Check that the page reference was legal and determine the location of the page on the disk
5. Issue a read from the disk to a free frame:
    1. Wait in a queue for this device until the read request is serviced
    2. Wait for the device seek and/or latency time
    3. Begin the transfer of the page to a free frame
6. While waiting, allocate the CPU to some other user (CPU scheduling, optional)
7. Receive an interrupt from the disk I/O subsystem (I/O completed)
8. Save the registers and process state for the other user (if Step-6 is executed)
9. Determine that the interrupt was from the disk
10. Correct the page table and other tables to show page is now in memory
11. Wait for the CPU to be allocated to this process again
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction

Performance of Demand Paging

- Not all steps (in Slide-18) are necessary in every case; **e.g., Step-6**
- **Three major components** of the page-fault service-time
  1. Service the interrupt; between 1 to 100 microseconds
     - Careful coding of the ISR means just several hundred instructions needed
  2. Read in the page – lots of time; **at least 8 milliseconds** + time in device-queue **+ …**
  3. Restart the process; between 1 to 100 microseconds – again, careful coding…
- Page Fault Rate $0 \le p \le 1$; p = probability of a page-fault and we expect $p \approx 0$
  - if $p = 0$ then there is no page faults
  - if $p = 1$ then every memory reference causes a page-fault
- **Effective Access Time (EAT)**
  - **EAT = [(1 – *p*) × memory_access_time] + [*p* × page_fault_time]**
    - **page_fault_time** = page fault overhead + swap page out + swap page in

Demand Paging Example

- Memory access time = 200 nanoseconds; between 10 to 200ns in most computers
- Average page-fault service time = 8 milliseconds

- EAT = [(1 – p) × (200 nanoseconds)] + [p × (8 milliseconds)]

  = [(1 – p) × 200] + [p × 8,000,000] nanoseconds

  = 200 + 7,999,800p nanoseconds; thus, EAT is directly proportional to p

- If one access out of 1,000 causes a page fault, then
  - EAT = 8,199.8 nanoseconds = 8.2 microseconds.
    - This is a slowdown by a factor of 40!!; Because of demand paging

- If want performance degradation < 10 percent
  - 220 > 200 + 7,999,800 × p
    20 > 7,999,800 × p
  - Thus, we must have p < .0000025
  - That is, to keep slowdown due to demand paging
    - p < one page fault in every 399,990 memory accesses

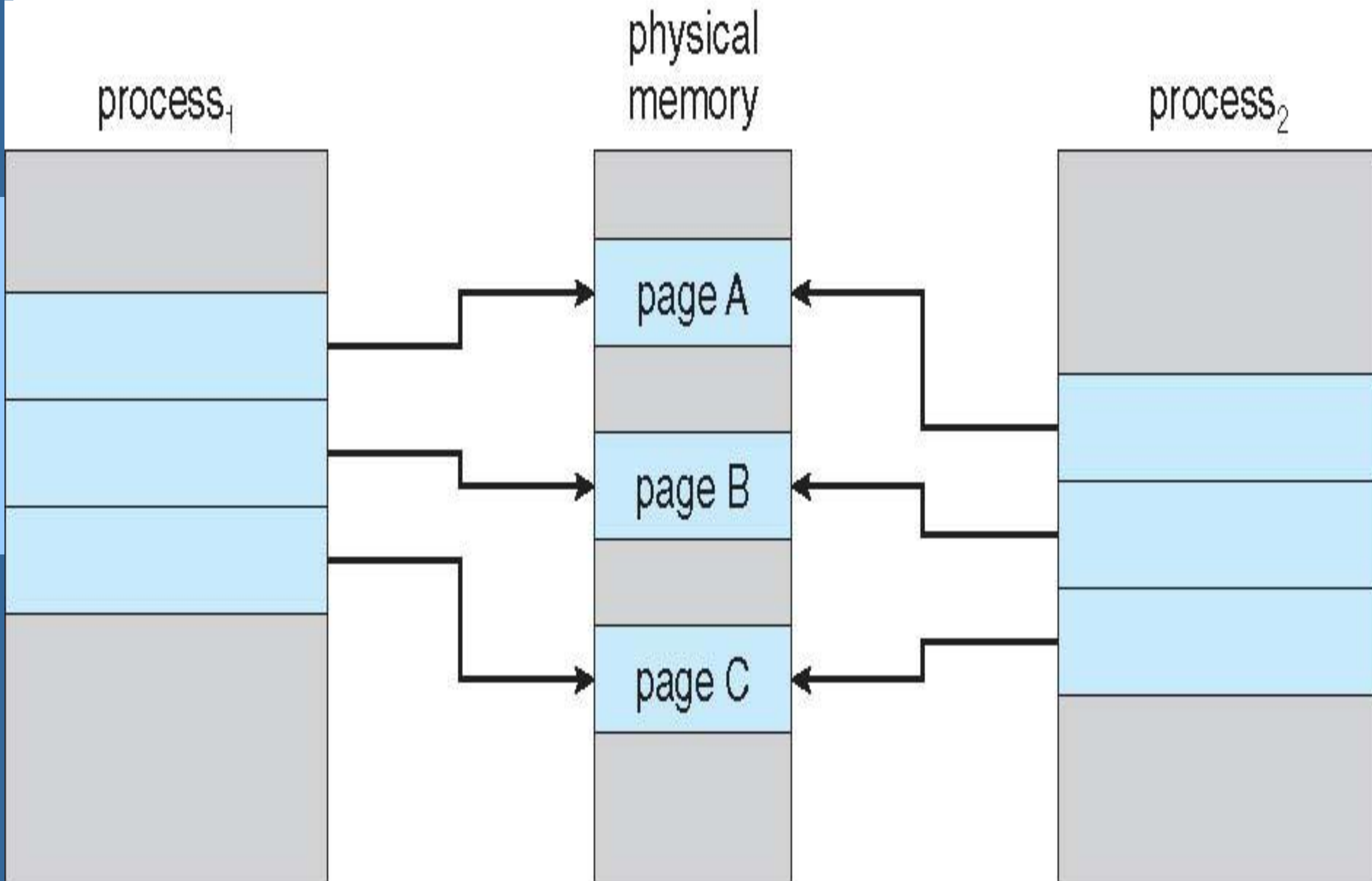
Demand Paging Optimizations

- 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 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)

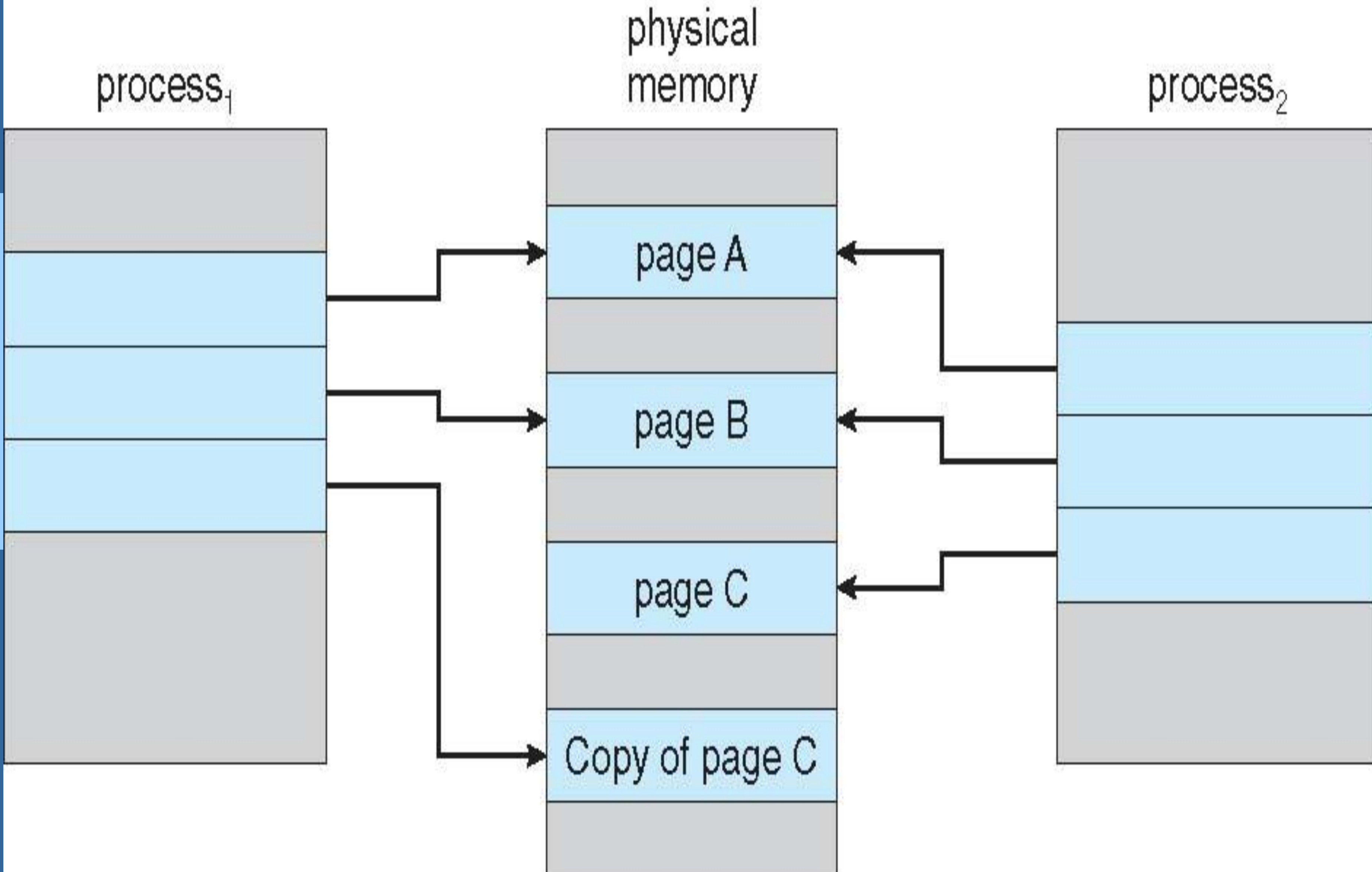
Copy-on-Write

- **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
  - 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
  - Why zero-out a page before allocating it?
- `vfork()` variation on `fork()` system call has parent suspended and child using copy-on-write address space of parent
  - Designed to have child call `exec()`
  - Very efficient

Before Process 1 Modifies Page C



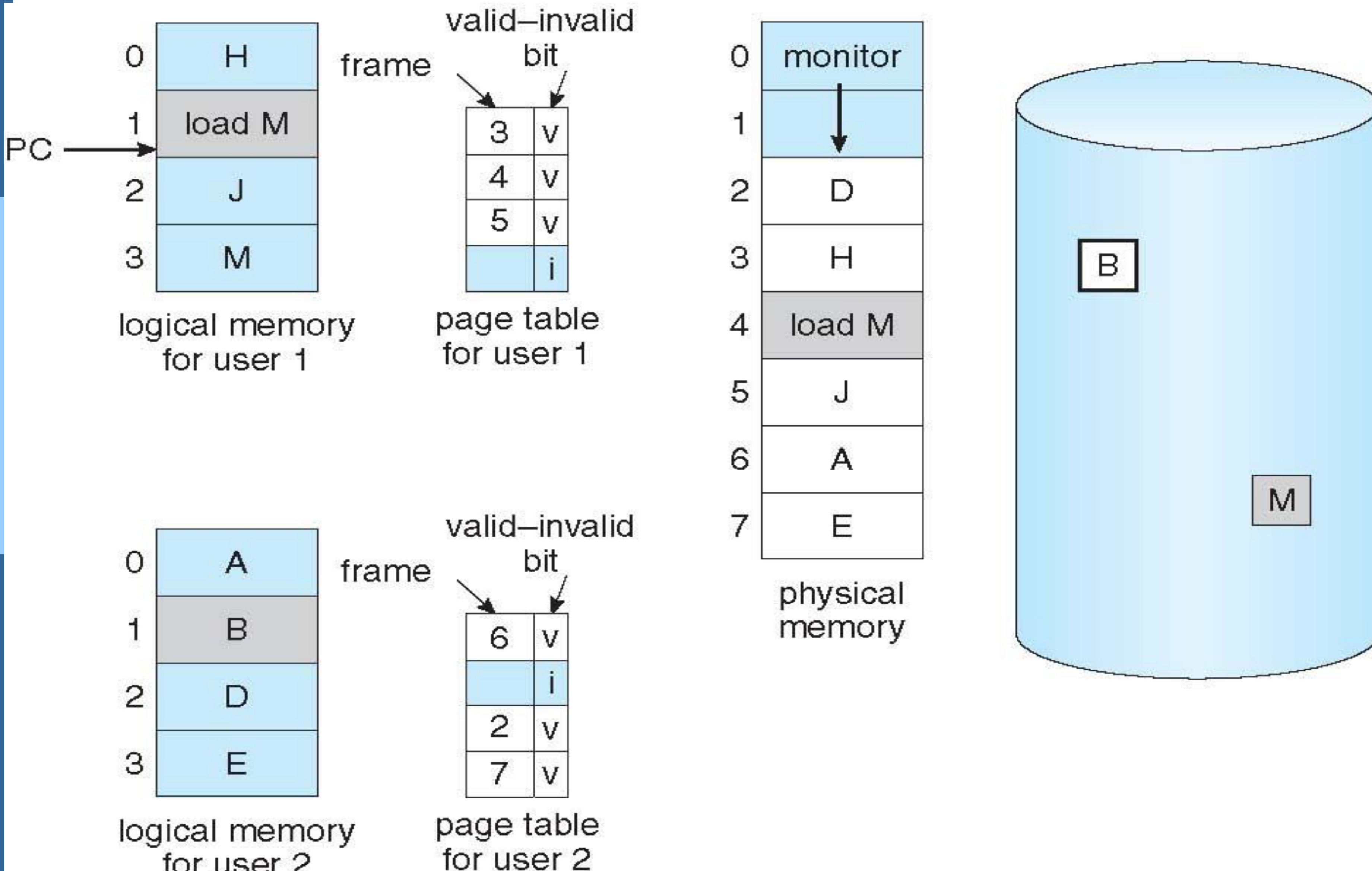
After Process 1 Modifies Page C



What Happens if There is no Free Frame?

- Many pages need to be loaded but not enough free frames available fro them
  - Memory is being used up by process pages; both, user and kernel processes
  - Also memory is in demand from the kernel, I/O buffers, etc
- How much memory to allocate to I/O buffers, kernel, processes, … , etc
- **Solution**: Page replacement; when paging in pages of a process but no free frames
  - Terminate the process? Big fat **no**
  - Swap out some process? **Yes**, but not always a good option
  - Find currently un-used frame to free it; Page it out and page in process page
    - Replacing the un-used memory page with the new page
  - Performance – want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times

Need For Page Replacement

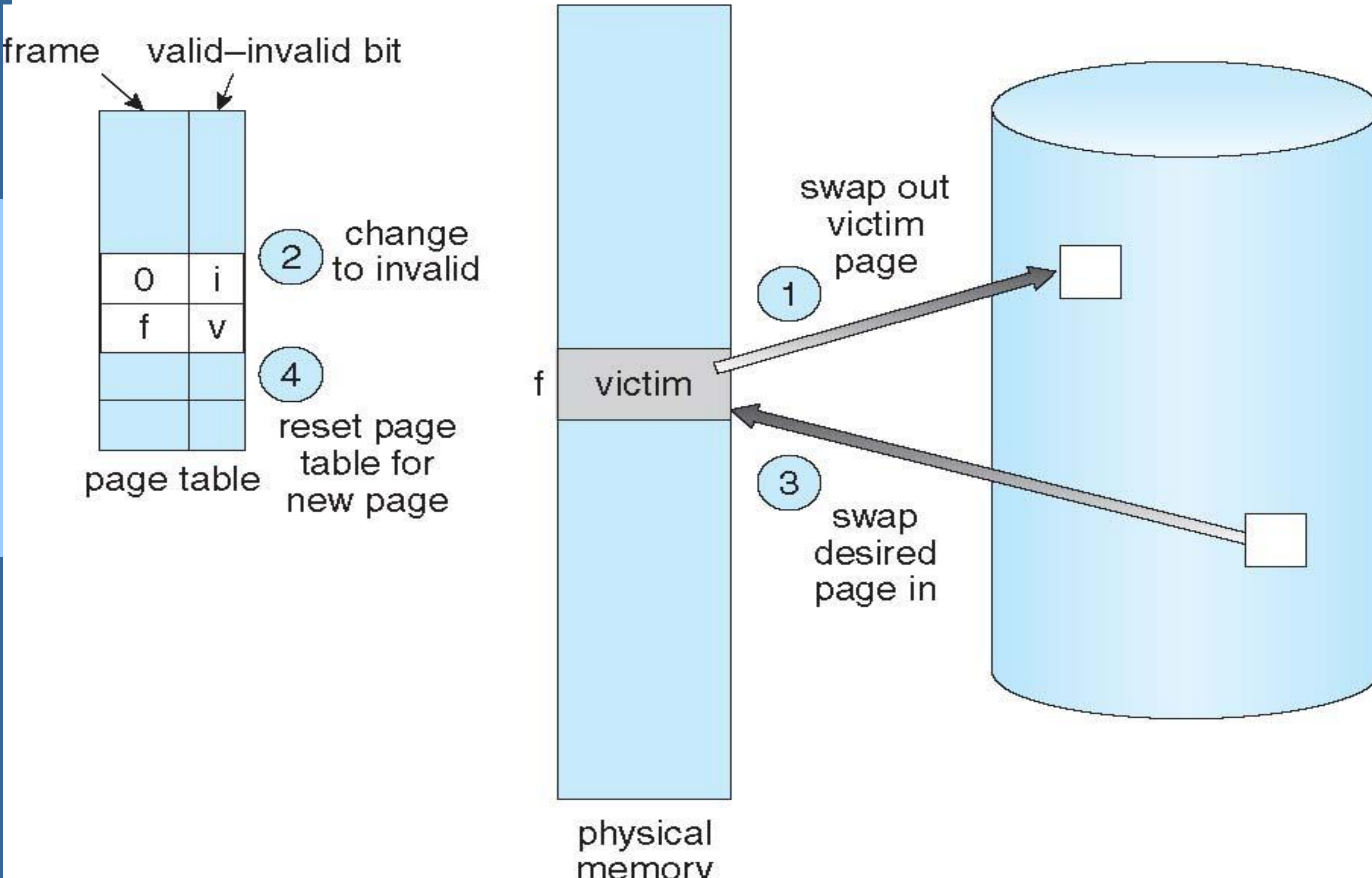


Basic Page Replacement Algorithm

- The page-fault service routine is modified to include page replacement
  1. Find the location of the desired page on disk
  2. Find a free frame:
     1. If there is a free frame, use it
     2. If there is no free frame, use a page-replacement algorithm to select a **victim frame**
     3. Write the victim frame to the disk [if dirty]; change the page and the frame tables accordingly
  3. Read the desired page into the newly freed frame; change the page and frame tables
  4. Continue the user process from where the page fault occurred

- We have potentially ***two*** page transfers to do – increasing EAT
  - Only if no frames are free; one page in required and one page out required

Page Replacement



Page Replacement …

- Prevent **over-allocation** of memory by modifying page-fault service routine to include page replacement
- Use **modify** (**dirty**) **bit** to reduce overhead of page transfers – only modified pages are written to disk ; **see Slide-28**
  - Each page or frame is associated with a **modify bit**
  - Set by the hardware whenever a page is modified
- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory
  - **A user process of 20 pages can be executed in 10 frames** simply by using demand-paging and using a page-replacement algorithm to find a free frame whenever necessary

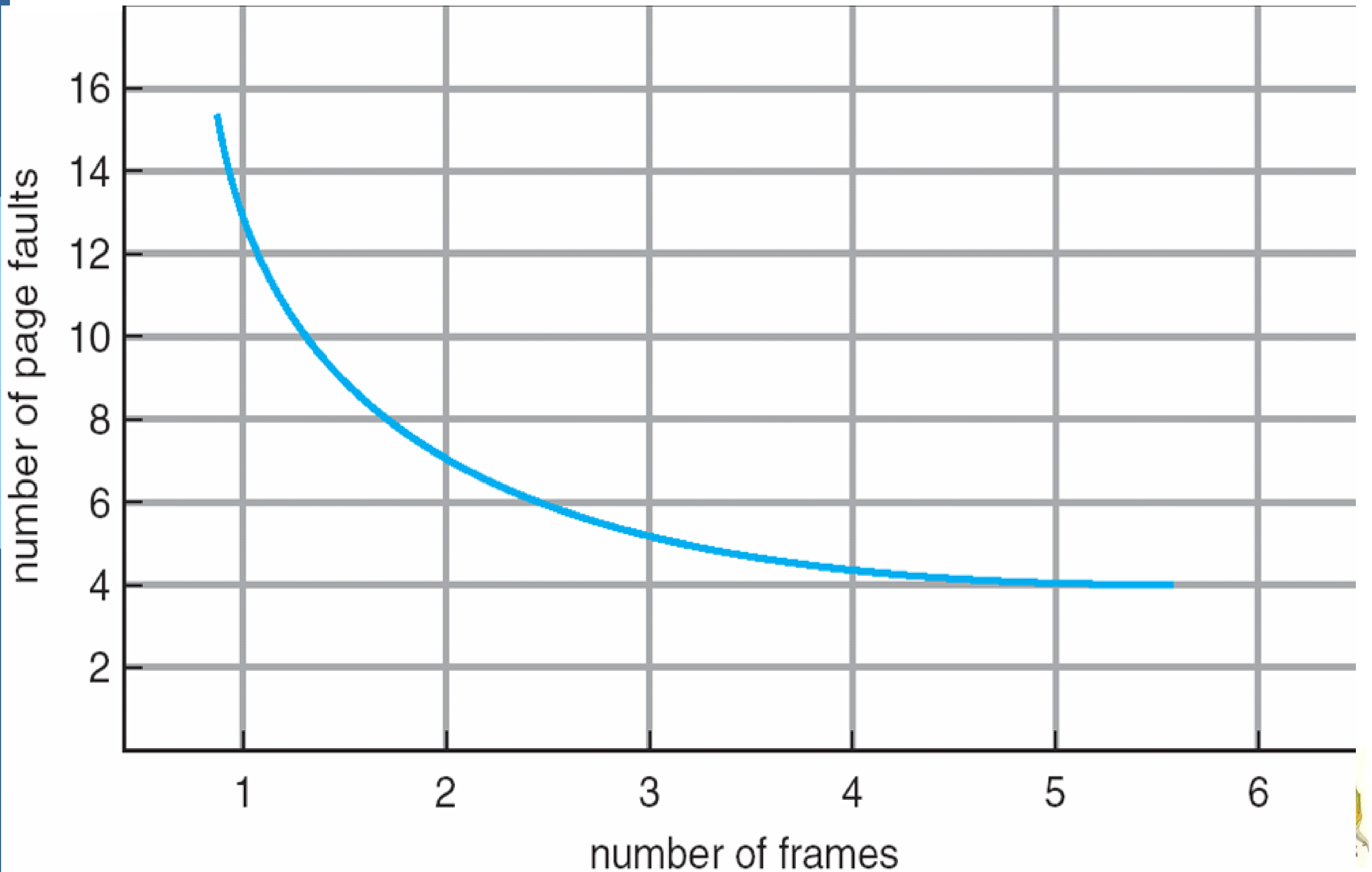
Page- Replacement and Frame-Allocation Algorithms

- Two major demand-paging problems : **frame allocation** and **page replacement**
- **Frame-allocation algorithm** determines
  - How many frames to allocate to each process
  - Which frames to replace; when page replacement is required
- **Page-replacement algorithm**
  - We want an algorithm which yields the lowest page-fault rate
- Evaluate an algorithm by running it on a particular string of memory references (the **reference string**) and computing the number of page faults on that string
  - String is just page numbers ***p***, not full addresses
  - Repeated access to the same page does not cause a page fault
  - Results depend on number of frames available
- In all our examples, the **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

  for a memory with three frames

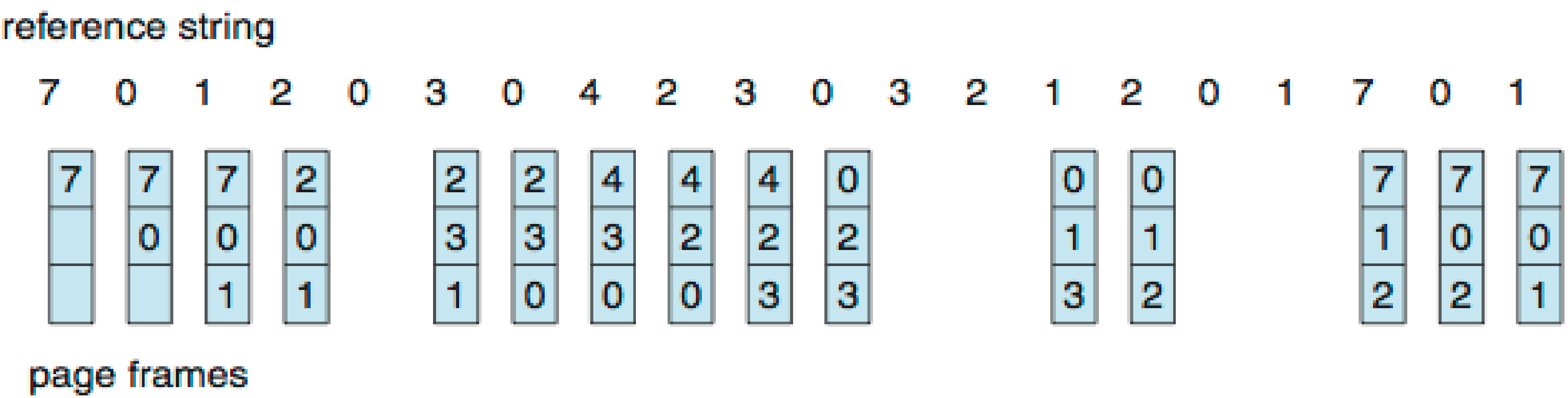
Graph of Page Faults Versus The Number of Frames



FIFO Page Replacement Algorithm

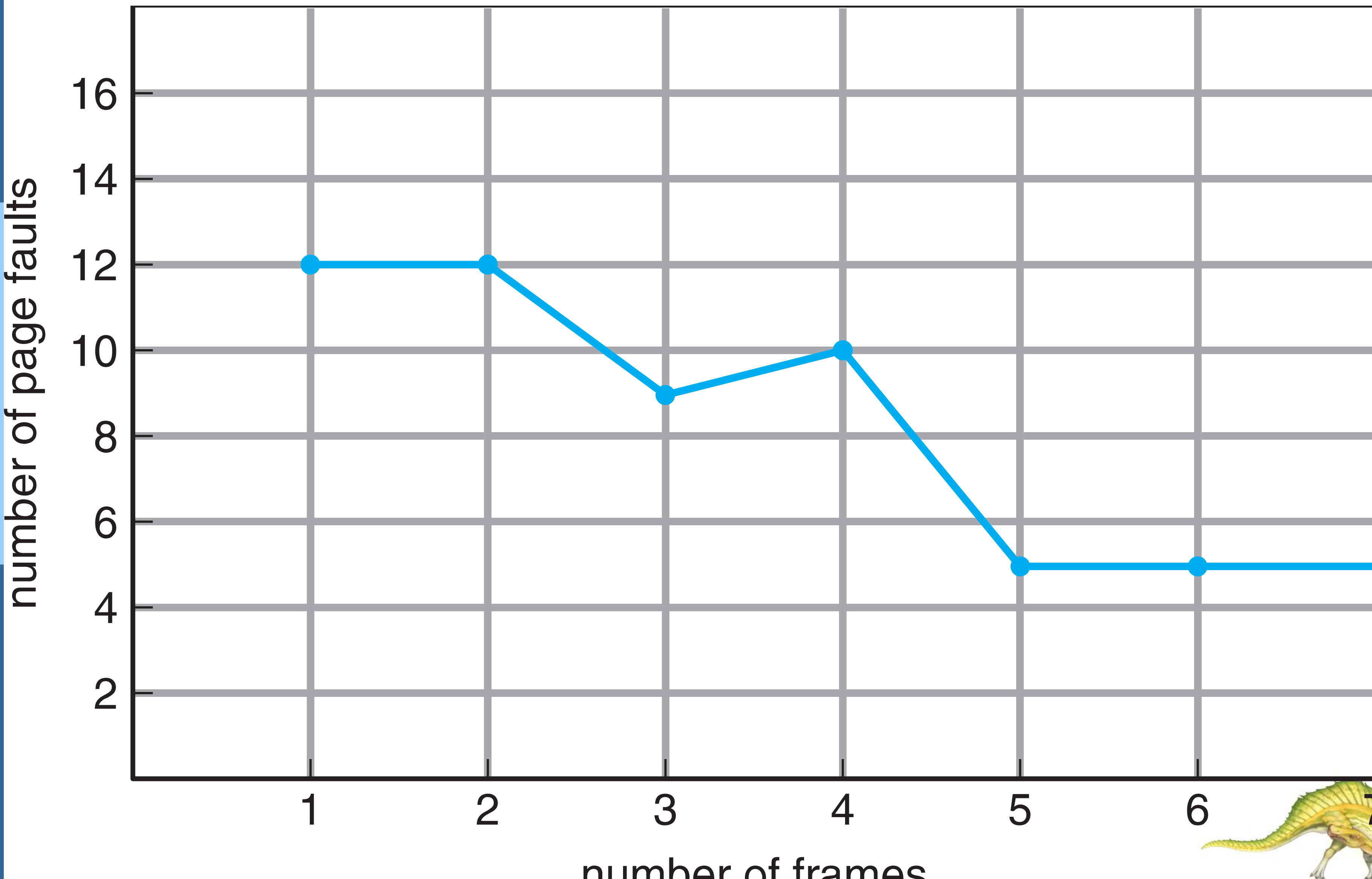
[Reference string = 7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1 and Memory = 3 frames]

- Each page brought into memory is also inserted into a **first-in first-out queue**
  - Page to be replaced is the **oldest** page; the one at the head of the queue
- Our example yields 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**

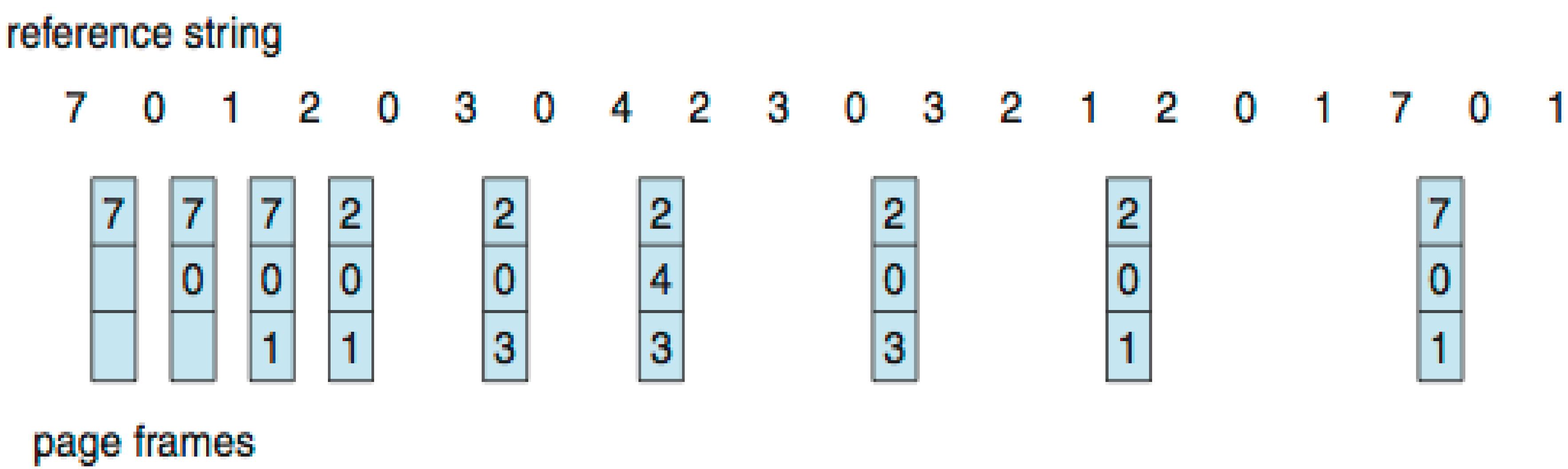
FIFO Illustrating Belady's Anomaly



Optimal Page Replacement 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 and Memory = 3 frames]

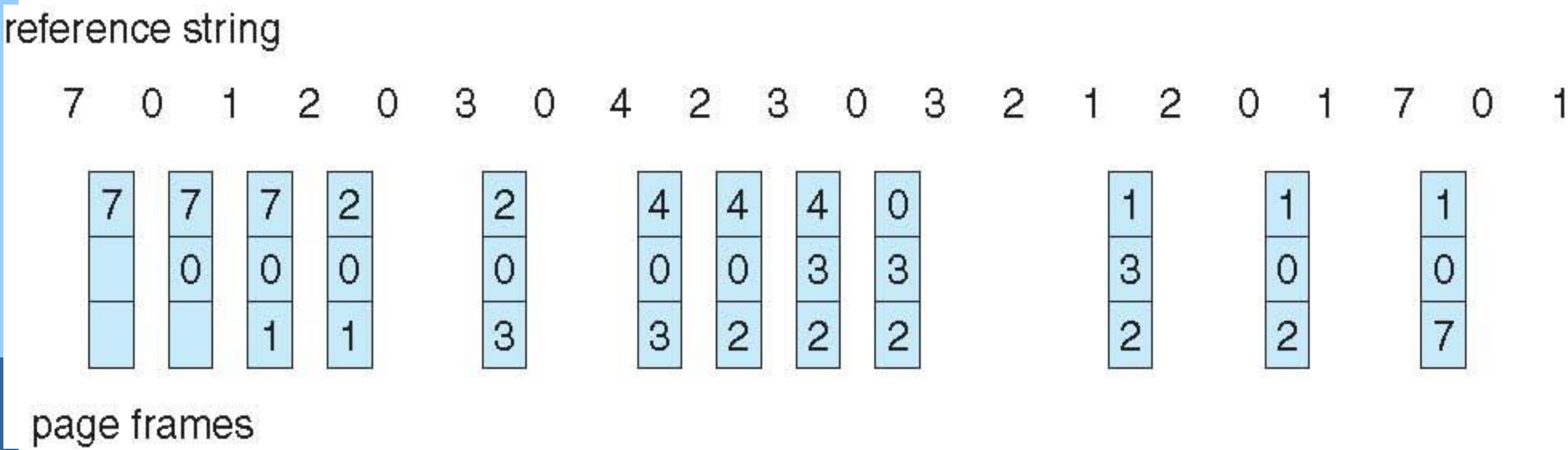
- Replace the page that will not be used for longest period of time
- Our example yields 9 page faults
- Unfortunately, OPR is **not feasible** to implement
  - Because: we can't know the future; i.e., what is the next page?
    - We have assumed that we know the reference string. No, we don't
- OPR is used only for comparing with new algorithms; **how close to the optimal?**



LRU Page Replacement 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 and Memory = 3 frames]

- Use the recent past as an approximation of the near future
- Replace the page that *has not been used* for the longest period of time
  - That is, the **least recently used** page
- Associate time of last use with each page

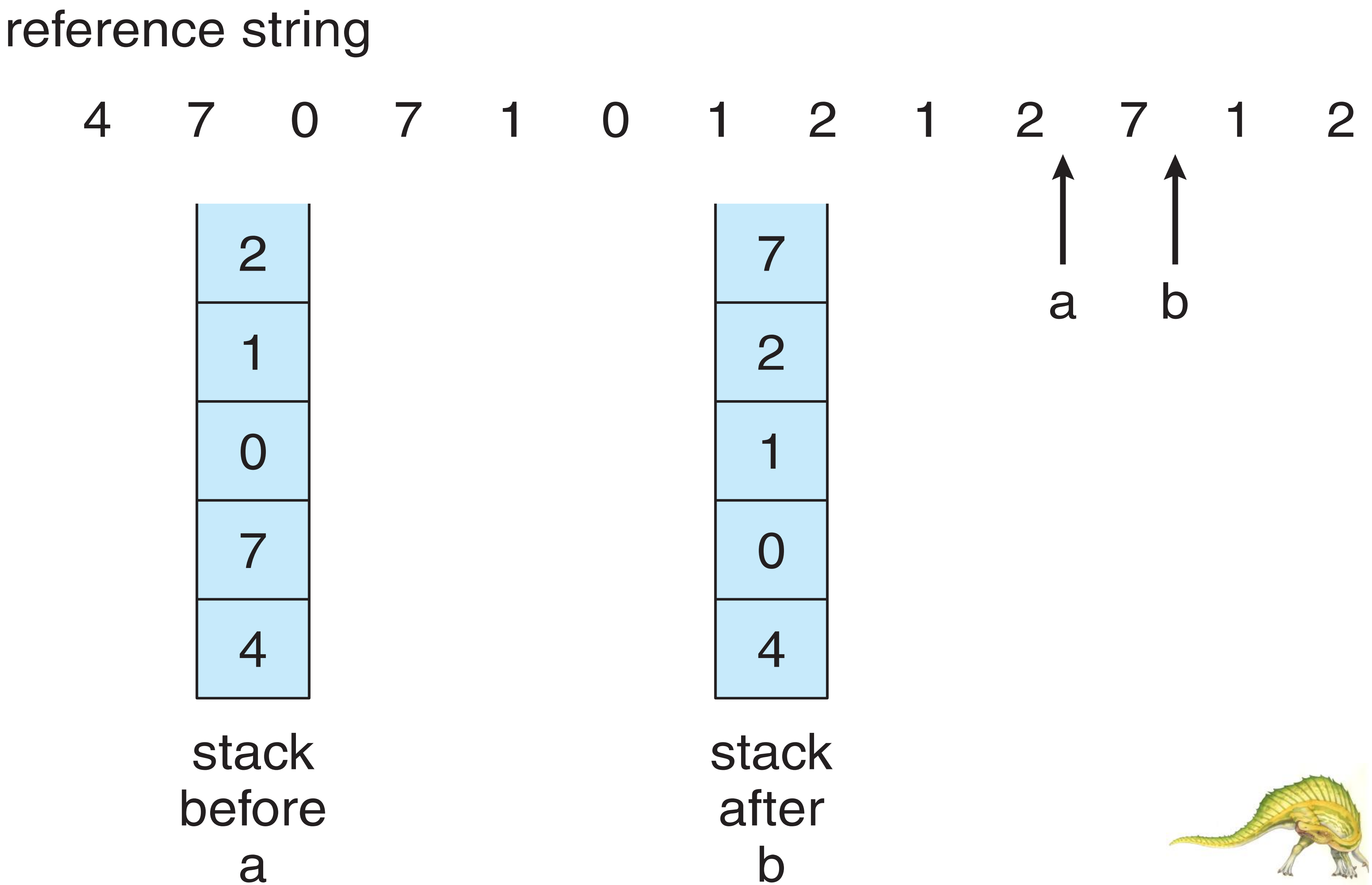


- Our example yields 12 page faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- Algorithm is feasible but not easy to implement.
  - LRU algorithm may require substantial hardware support

LRU Algorithm

- **Counter implementation**
  - Each page-table entry has a counter; every time the page is referenced through this entry, copy the current clock value into the counter
  - When a page needs to be changed, look at the counters to find smallest value
    - Search through the page-table needed; to find the LRU page
- **Stack implementation**
  - Keep a stack of page numbers in a double link form, with head and tail pointers:
  - Whenever a page is referenced:
    - move it to the top; most recently used page is always at the top of stack
    - requires 6 pointers to be changed
  - But each update more expensive
  - No search for replacement; as LRU page is always at the bottom of the stack
- LRU and OPT are cases of **stack algorithms** that don't have Belady's Anomaly

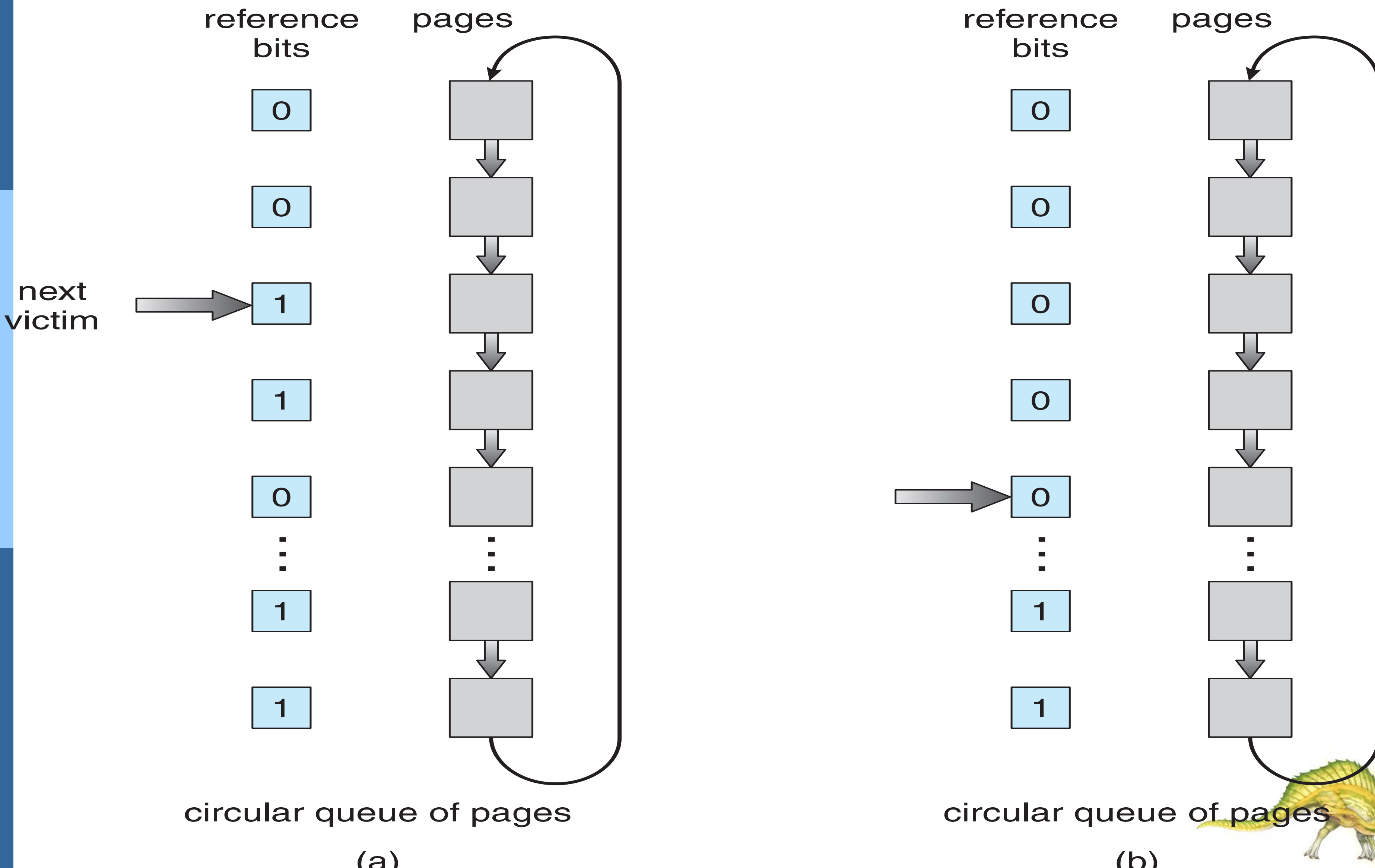
Use of A Stack to Record Most Recent Page References



LRU-Approximation Page Replacement Algorithms

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

Second-Chance (clock) Page-Replacement Algorithm



Enhanced Second-Chance Algorithm

- Improve algorithm by using reference bit and modify bit (if available) in concert
- Take ordered pair (reference, modify)

1. (0, 0) neither recently used not modified – best page to replace
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

- When page replacement called for, use the clock scheme but use the four classes replace page in lowest non-empty class
  - Might need to search circular queue several times

Counting-Based Page Replacement Algorithms

[Reference string = 7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1 and Memory = 3 frames]

- Keep a counter of the number of references that have been made to each page
  - Not common
- **Lease Frequently Used** (**LFU**) **Algorithm**: replaces the page with the smallest count
  - An actively used page should have a large count value
    - But... Pages may be heavily used initially and never used again
- **Most Frequently Used** (**MFU**) **Algorithm**: based on the argument that the page with the smallest count was probably just brought in and has yet to be used
- Counting-based algorithms are very expensive to implement, and they do not approximate OPT replacement well

Page-Buffering Algorithms

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

Applications and Page Replacement

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

Allocation of Frames

- Each process needs be allocated a ***minimum*** number of frames
  - Number of page-faults increases as the # of allocated frames decreases
  - The minimum number of frames is defined by the computer architecture
    - Example: IBM 370 – references 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 the total number of frames in the system
- Two major allocation schemes
  - fixed allocation
  - priority allocation
- Many variations

Fixed Allocation

- **Equal allocation** – For example: after allocating frames to the OS, and if there are 100 free frames and 5 processes, then allocate 20 frames to each process.
  - Keep the leftover free frames as free-frame buffer pool
- **Proportional allocation** – Allocate free frames according to the size of process
  - Dynamic, as the degree of multiprogramming and the 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
  - Ratio of frames depends on the priorities of processes, or, on a combination of both their priorities and their sizes
  - We may want to allocate more frames to a high-priority process, in order to speed up its execution, to the detriment of low-priority processes
- In this case, the replacement algorithm is modified to consider process's priorities
  - If high-priority 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** – allows a process to select a replacement frame from the set of **all** frames; that is, one process can take a frame from another
  - But then process execution time can vary greatly
  - But greater throughput so more common
- **Local replacement** – requires that a process selects from only its own set of allocated frames
  - More consistent per-process performance
  - But possibly underutilized memory

Non-Uniform Memory Access

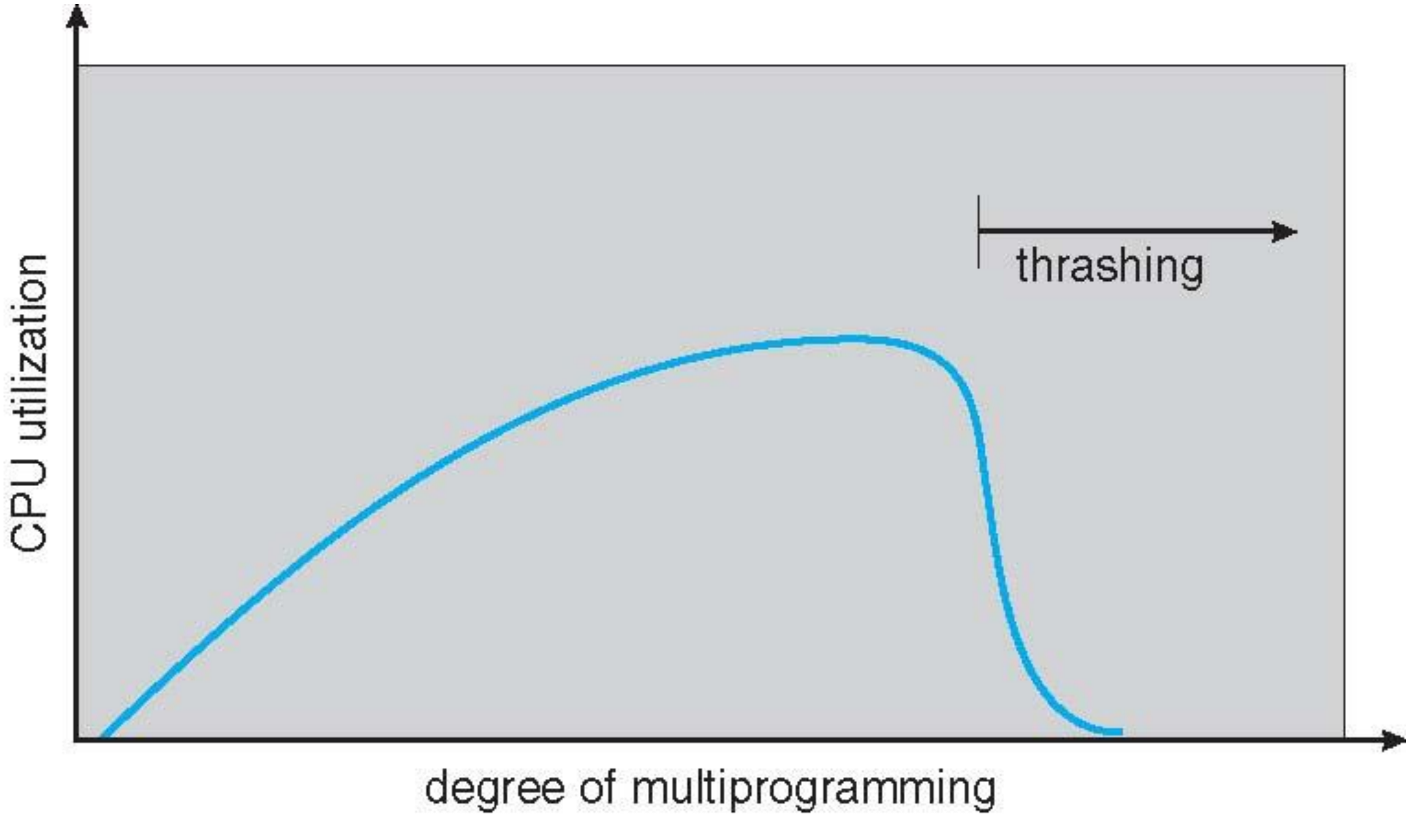
- So far we have assume that all main memory is accessed equally
- Many systems are **NUMA** – speed of access to memory varies
  - Consider system boards containing CPUs and memory, interconnected over a system bus or a high-speed network
- Optimal performance comes from allocating memory frames "*as close as possible to*" the CPU on which the process or thread is running or scheduled
  - And modifying the scheduler to schedule the thread on the same system board when possible; thus, taking NUMA into account
  - Thread issues solved by Solaris by creating **lgroups**; that is, **latency groups**
    - Each lgroup gathers together close CPUs and memories
    - Tries to schedule all threads of a process and allocate all memory of a process within an lgroup
      - If not possible, then pick a nearby lgroup

Thrashing

- If a process does not have "enough" pages, the page-fault rate is very high
  - Page fault to get page
  - Replace existing frame
  - But quickly need replaced frame back
  - This leads to:
    - Low CPU utilization
    - Operating system thinking that it needs to increase the degree of multiprogramming
    - Another process added to the system

- **Thrashing** ≡ a process is busy swapping pages in and out

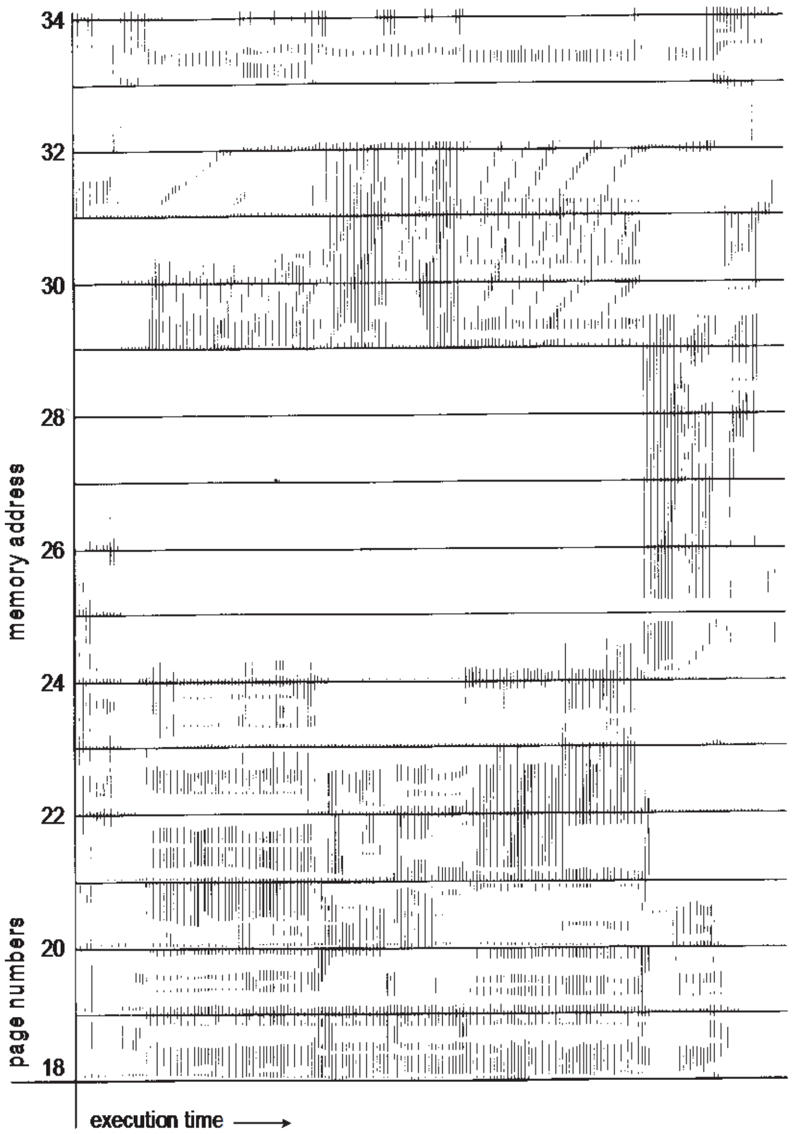
Thrashing (Cont.)



Demand Paging and Thrashing

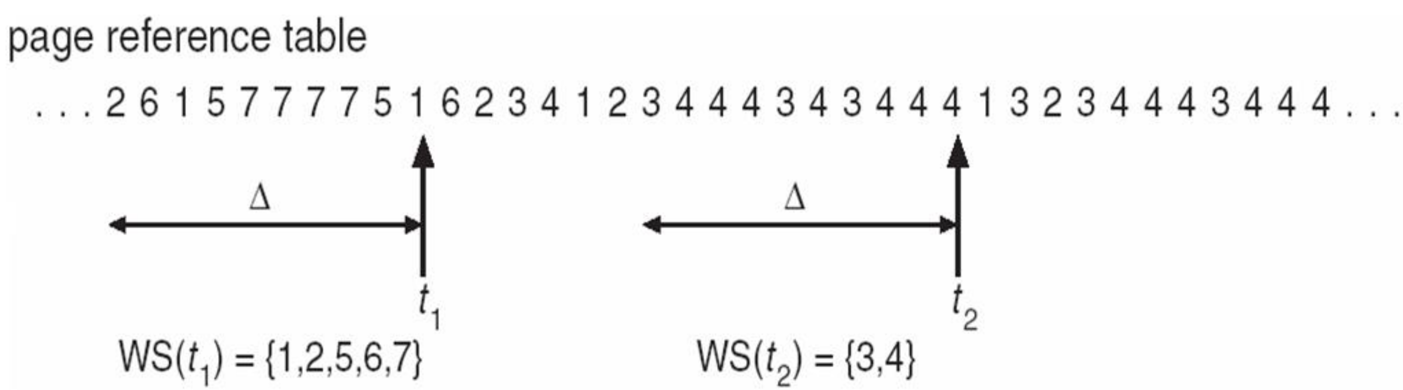
- Why does demand paging work?
  Locality model
  - Process migrates from one locality to another
  - Localities may overlap
- Why does thrashing occur?
  Σ size of locality > total memory size
  - Limit effects by using local or priority page replacement

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

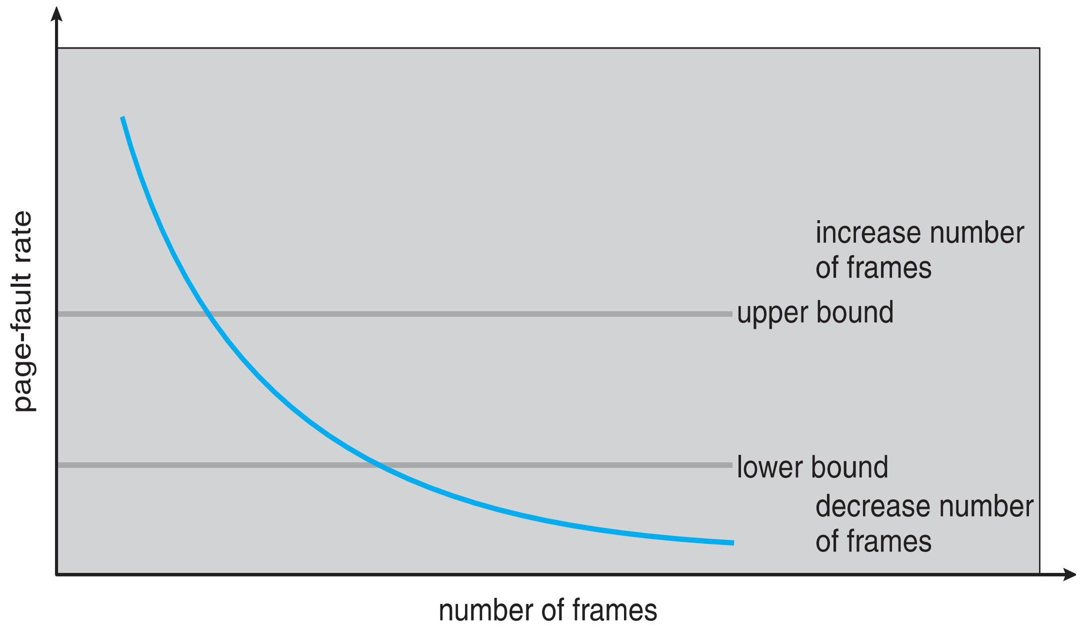


Keeping Track of the Working Set

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

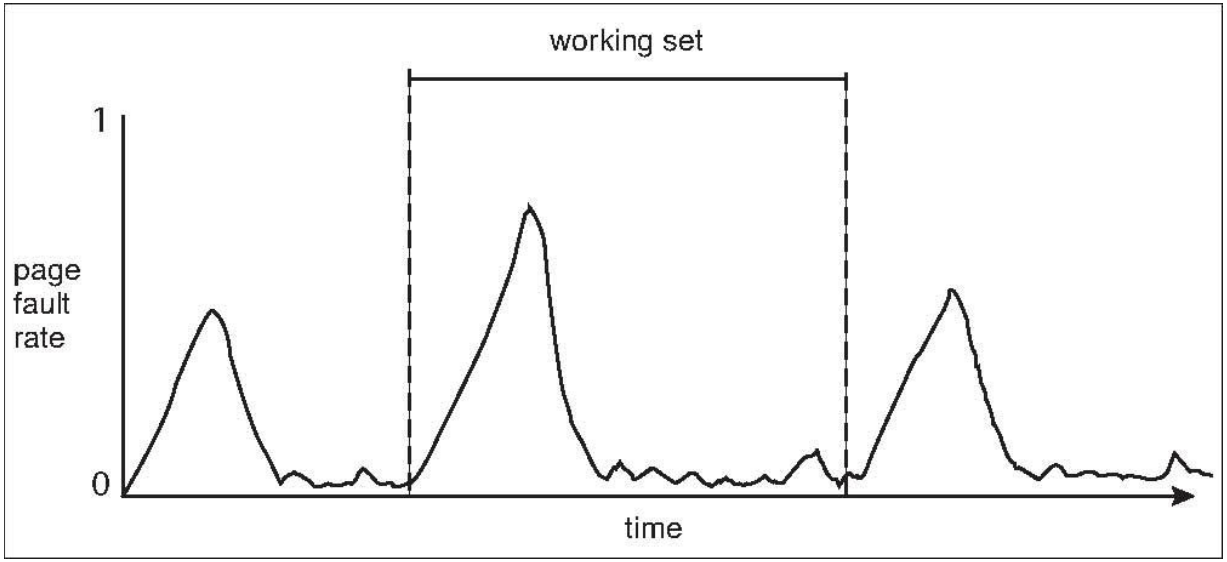
Page-Fault Frequency

- More direct approach than WSS
- Establish "acceptable" **page-fault frequency** (**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



Operating System Examples

- Windows

- Solaris

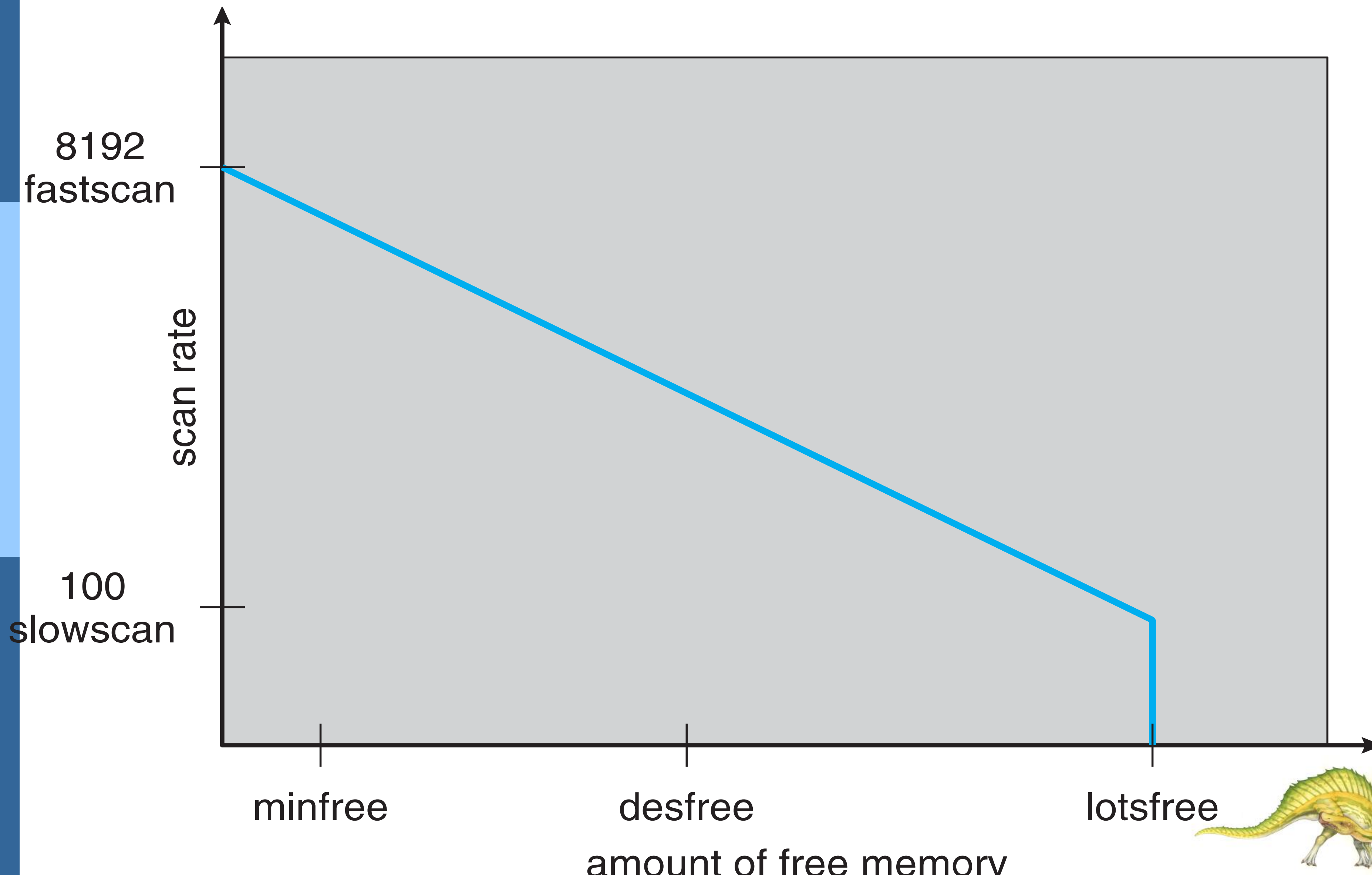
Windows

- 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

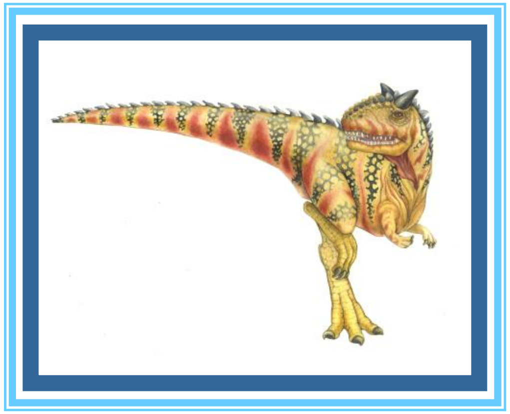
Solaris

- 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 Chapter 9



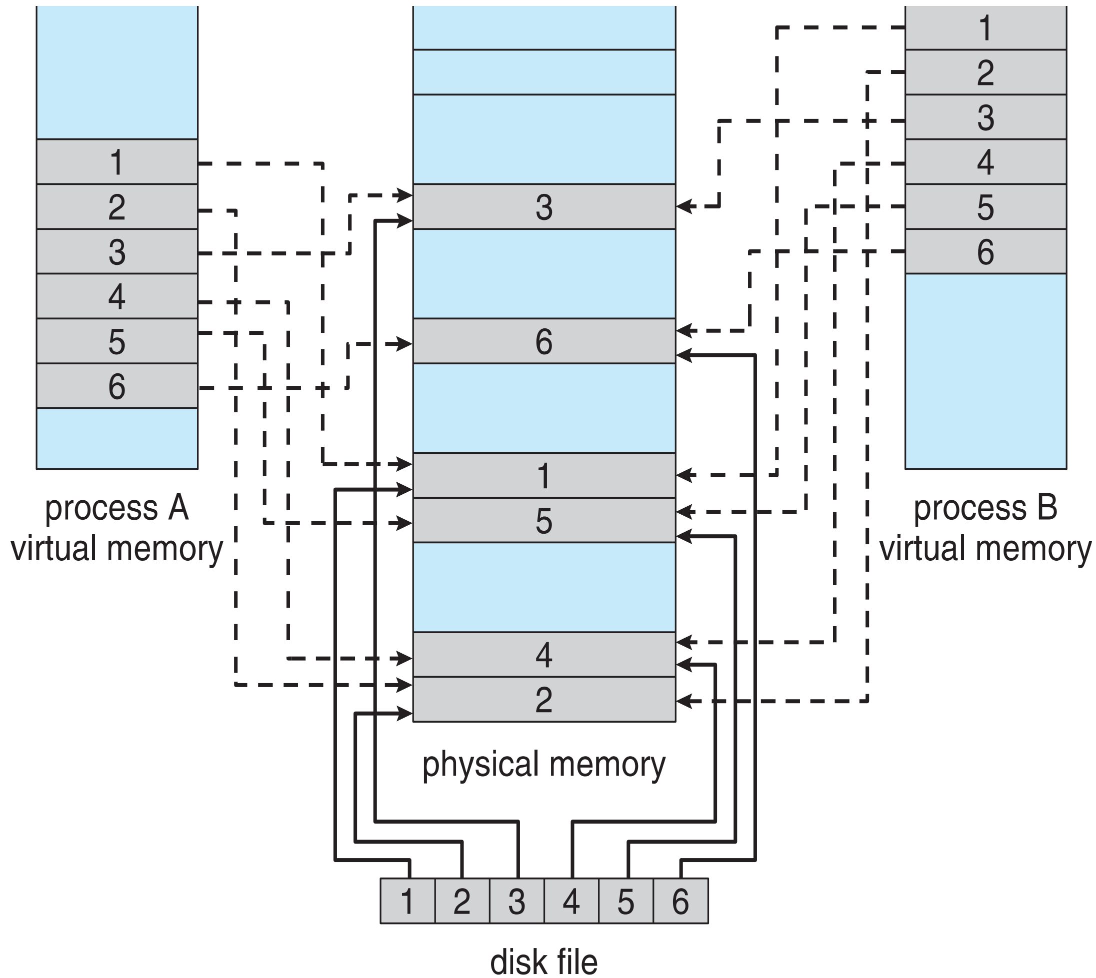
Memory-Mapped Files

- Memory-mapped file I/O allows file I/O to be treated as routine memory access by **mapping** a disk block to a page in memory
- A file is initially read using demand paging
  - A page-sized portion of the file is read from the file system into a physical page
  - Subsequent reads/writes to/from the file are treated as ordinary memory accesses
- Simplifies and speeds file access by driving file I/O through memory rather than `read()` and `write()` system calls
- Also allows several processes to map the same file allowing the pages in memory to be shared
- But when does written data make it to disk?
  - Periodically and / or at file `close()` time
  - For example, when the pager scans for dirty pages

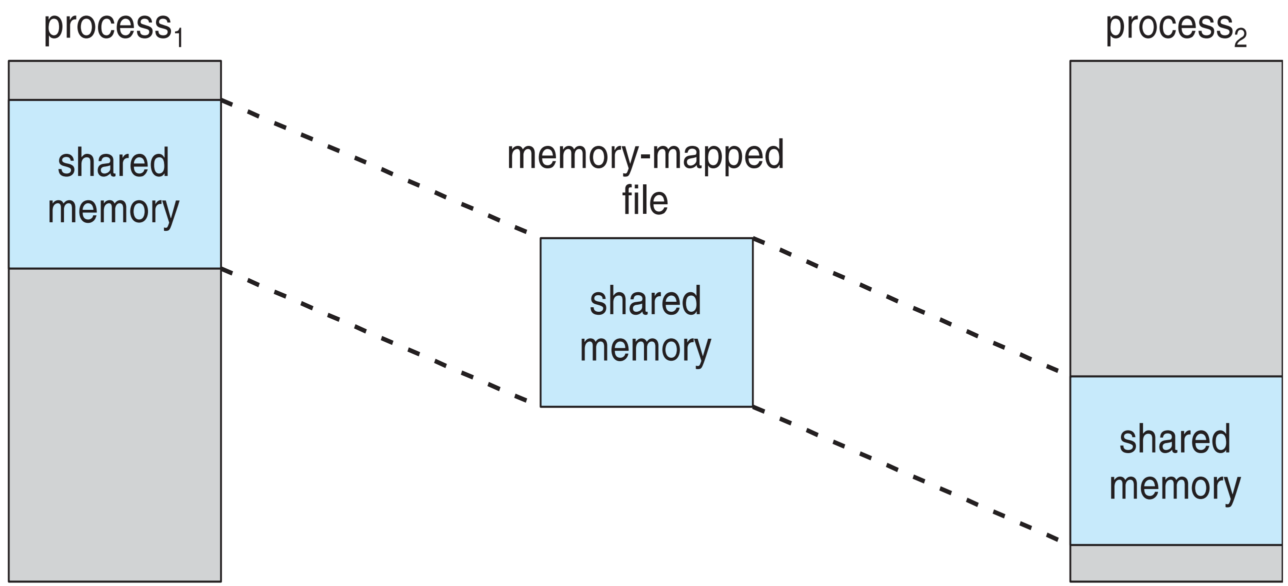
Memory-Mapped File Technique for all I/O

- Some OSes 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
- 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



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

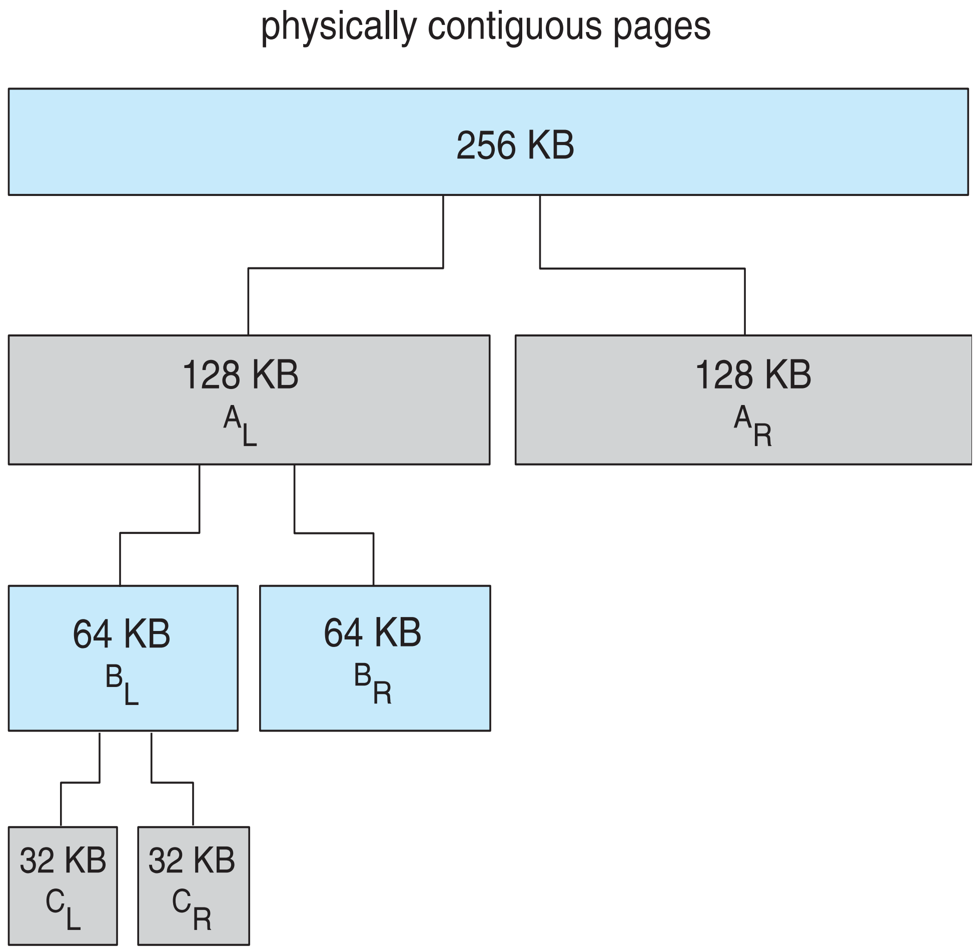
Allocating Kernel Memory

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

Buddy System

- Allocates memory from fixed-size segment consisting of physically-contiguous pages
- Memory allocated using **power-of-2 allocator**
  - Satisfies requests in units sized as power of 2
  - Request rounded up to next highest power of 2
  - When smaller allocation needed than is available, current chunk split into two buddies of next-lower power of 2
    - Continue until appropriate sized chunk available
- For example, assume 256KB chunk available, kernel requests 21KB
  - Split into A_L $_{and}$ 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

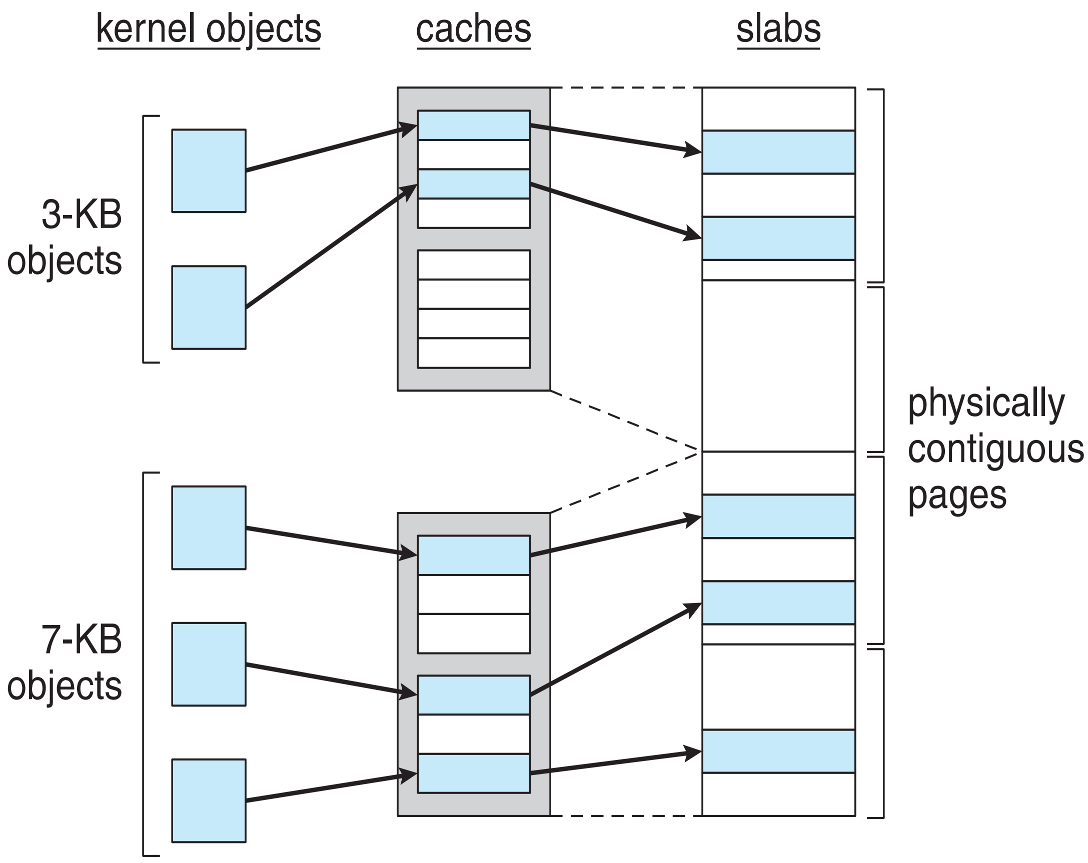
Buddy System Allocator



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
  - Each cache filled with **objects** – instantiations of the data structure
- When cache created, filled with objects marked as `free`
- When structures stored, objects marked as `used`
- If slab is full of used objects, next object allocated from empty slab
  - If no empty slabs, new slab allocated
- Benefits include no fragmentation, fast memory request satisfaction

Slab Allocation



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

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 $\Rightarrow$ prepaging loses

Other Issues – Page Size

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

Other Issues – TLB Reach

- TLB Reach - The amount of memory accessible from the TLB
- TLB Reach = (TLB Size) X (Page Size)
- Ideally, the working set of each process is stored in the TLB
  - Otherwise there is a high degree of page faults
- Increase the Page Size
  - This may lead to an increase in fragmentation as not all applications require a large page size
- Provide Multiple Page Sizes
  - This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation

Other Issues – Program Structure

- Program structure
  - `int[128,128] data;`
  - Each row is stored in one page
  - Program 1

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

128 x 128 = 16,384 page faults

  - Program 2

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

128 page faults

Other Issues – I/O interlock

- **I/O Interlock** – Pages must sometimes be locked into memory
- Consider I/O - Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm
- **Pinning** of pages to lock into memory