



# CS 5323 – OS II

## Lecture 10 – Main Memory and Virtual Management



# Logistics

- Quiz 4 posted and is due: Monday 03/14/2022 11:59 pm
- Assignment 3 posted. Due 03/27/2022 11:59 pm

# Background



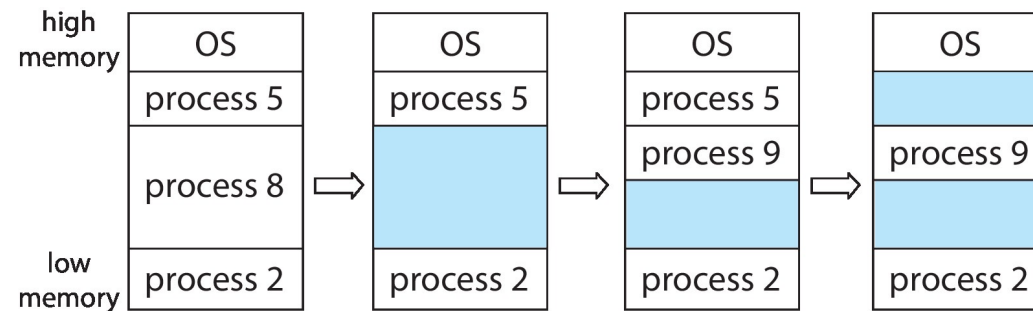
- Program must be brought (from disk) into memory and placed within a process for it to be run
  - Fetch -> Decode -> Execute
  - Fetch next instruction from memory
  - Decode the new instruction – If necessary: fetch operands (data) from memory
  - Execute the instruction – If necessary: save results (new data) to memory
- Main memory and registers are only storage CPU can access directly
  - There are no machine instructions taking disk addresses as arguments
- Memory unit only sees a stream of:
  - addresses + read requests, or
  - address + data and write requests
- Register access is done in one CPU clock (or less); **very fast memories**
- Main memory can take many cycles, causing a **stall**
- **Cache** sits between main memory and CPU registers; **solution to stall issue**
- Protection of memory required to ensure correct operation; **hardware-level soln.**

# Variable Partition



- Multiple-partition allocation

- Degree of multiprogramming limited by number of partitions
- **Variable-partition** sizes for efficiency (sized to a given process' needs)
- **Hole** – block of available memory; holes of various size are scattered throughout memory
- When a process arrives, it is allocated memory from a hole large enough to accommodate it
- Process exiting frees its partition, adjacent free partitions combined
- Operating system maintains information about:
  - a) allocated partitions    b) free partitions (hole)



# Dynamic Storage-Allocation Problem



How to satisfy a request of size  $n$  from a list of free holes?

- **First-fit**: Allocate the *first* hole that is big enough
- **Best-fit**: Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size
  - Produces the smallest leftover hole
- **Worst-fit**: Allocate the *largest* hole; must also search entire list
  - Produces the largest leftover hole

First-fit and best-fit better than worst-fit in terms of speed and storage utilization

# Fragmentation



- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
  - Free memory space is broken into pieces as processes are loaded and removed from memory
  - Both, first-fit and best-fit strategies suffer from external fragmentation
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
  - Example: a process requests 18,462 bytes and is allocated memory hole of 18,464 bytes; we are then left with a hole of 2 bytes
- First fit analysis reveals that given  $N$  blocks allocated,  $0.5 N$  blocks lost to fragmentation
  - $1/3$  may be unusable -> **50-percent rule**

# Fragmentation (Cont.)



- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time
  - I/O problem
    - Latch job in memory while it is involved in I/O
    - Do I/O only into OS buffers
- Now consider that backing store has same fragmentation problems
- Other solutions to external fragmentation problems: Segmentation and Paging
  - Permit the logical address space of each process to be non-contiguous
    - Process can be allocated physical memory wherever it is available

# Paging



- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
  - Avoids external fragmentation **and the need for compaction**
  - Avoids problem of varying sized memory chunks
- Divide physical memory into fixed-sized blocks called **frames**
  - Block size is a power of 2; between 512 bytes and 1 Gbytes
  - Divide logical memory into blocks of same size called **pages**
  - Divide logical memory into blocks of same size as the frames called pages
  - The page size (like the frame size) is defined by the hardware
  - Divide backing store into [clusters of] blocks of same size as the frames
- Keep track of all free frames
- To run a program of size  **$N$**  pages, need to find  **$N$**  free frames and load program
- Set up a **page table** to translate logical to physical addresses
- Backing store likewise split into pages
- Still have **Internal fragmentation**

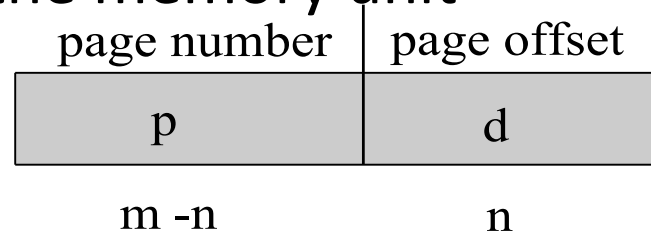


# Address Translation Scheme



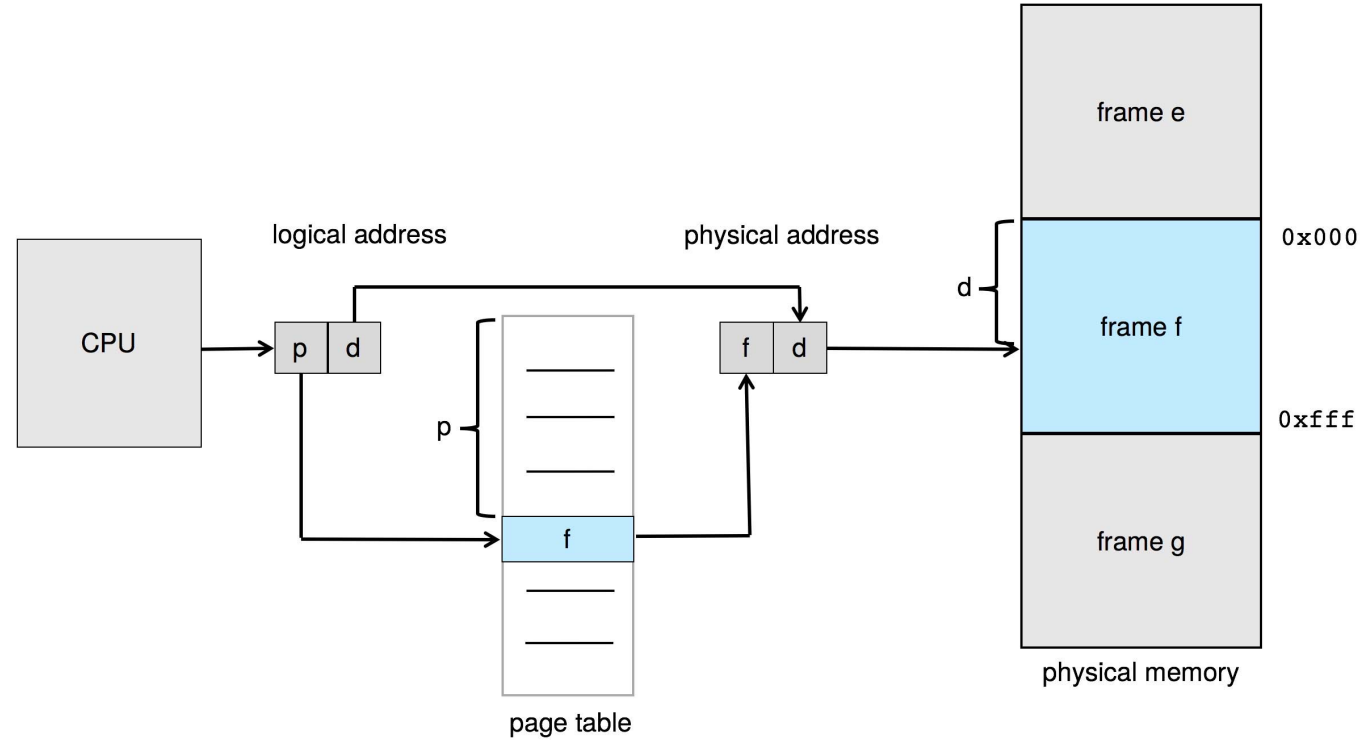
■ Address generated by CPU is divided into:

- **Page number ( $p$ )** – used as an index into a **page table** which contains base address of each page in physical memory
  - Page table contains the base address of each page in physical memory
- **Page offset ( $d$ )** – combined with base address to define the physical memory address that is sent to the memory unit



- If the logical address space is  $2^m$  and the page size is  $2^n$ 
  - The binary representation of the logical address has  $m$  bits, such that
    - The  $m - n$  leftmost bits designate the page number  $p$ 
      - $p$  is index into the page table
    - The rightmost  $n$  bits designate the page offset  $d$ 
      - $d$  is displacement within the page

# Paging Hardware



# Paging Example



- Logical address:  $n = 2$  and  $m = 4$ . Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages)

0	a
1	b
2	c
3	d
4	e
5	f
6	g
7	h
8	i
9	j
10	k
11	l
12	m
13	n
14	o
15	p

logical memory

0	5
1	6
2	1
3	2

page table

0	
4	i j k l
8	m n o p
12	
16	
20	a b c d
24	e f g h
28	

physical memory

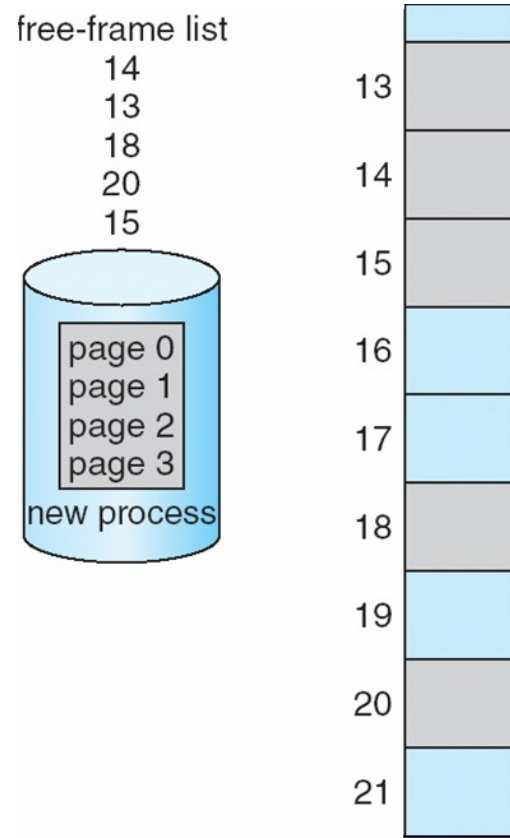
# of pages = total bytes/page size

# Paging



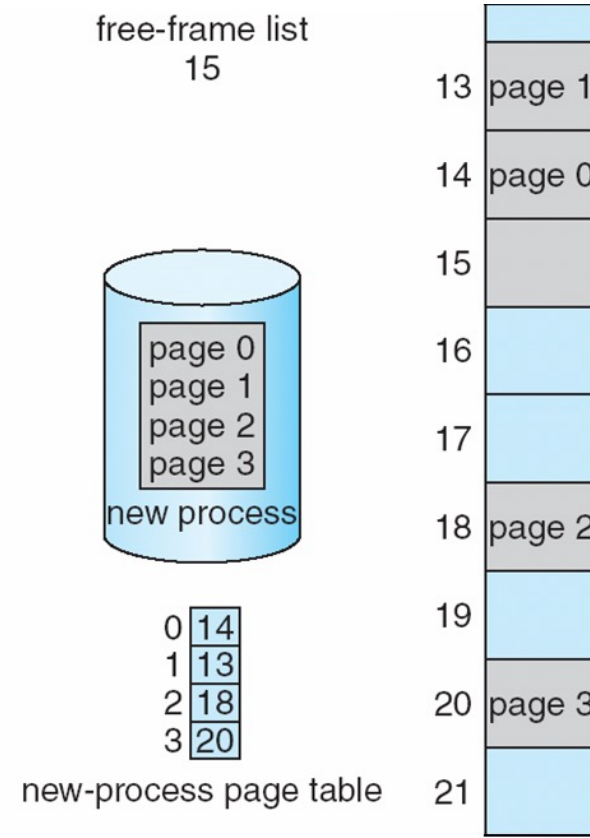
- There is no external fragmentation when using paging scheme
- Internal fragmentation is possible. Calculating internal fragmentation
- Page size = 2,048 bytes
- Process size = 72,766 bytes
  - Process has: 35 pages + 1,086 bytes => 36 frames required
- Internal fragmentation of  $2,048 - 1,086 = 962$  bytes of unused memory
- Worst case fragmentation = 1 frame – 1 byte
  - when 1 frame contains only 1 byte of used memory
- On average fragmentation =  $1 / 2$  frame size
  - So small frame sizes desirable?
  - Not necessarily; each page-table entry takes memory to track (overhead)
  - Large frame sizes better when transferring data to/from disk; efficient disk I/O
- Page size: 4KB ~ 8KB but growing over time, and researching variable page size
  - Solaris supports two page sizes – 8 KB and 4 MB; fixed multiple page sizes

# Free Frames



(a)

Before allocation



(b)

After allocation

# Implementation of Page Table



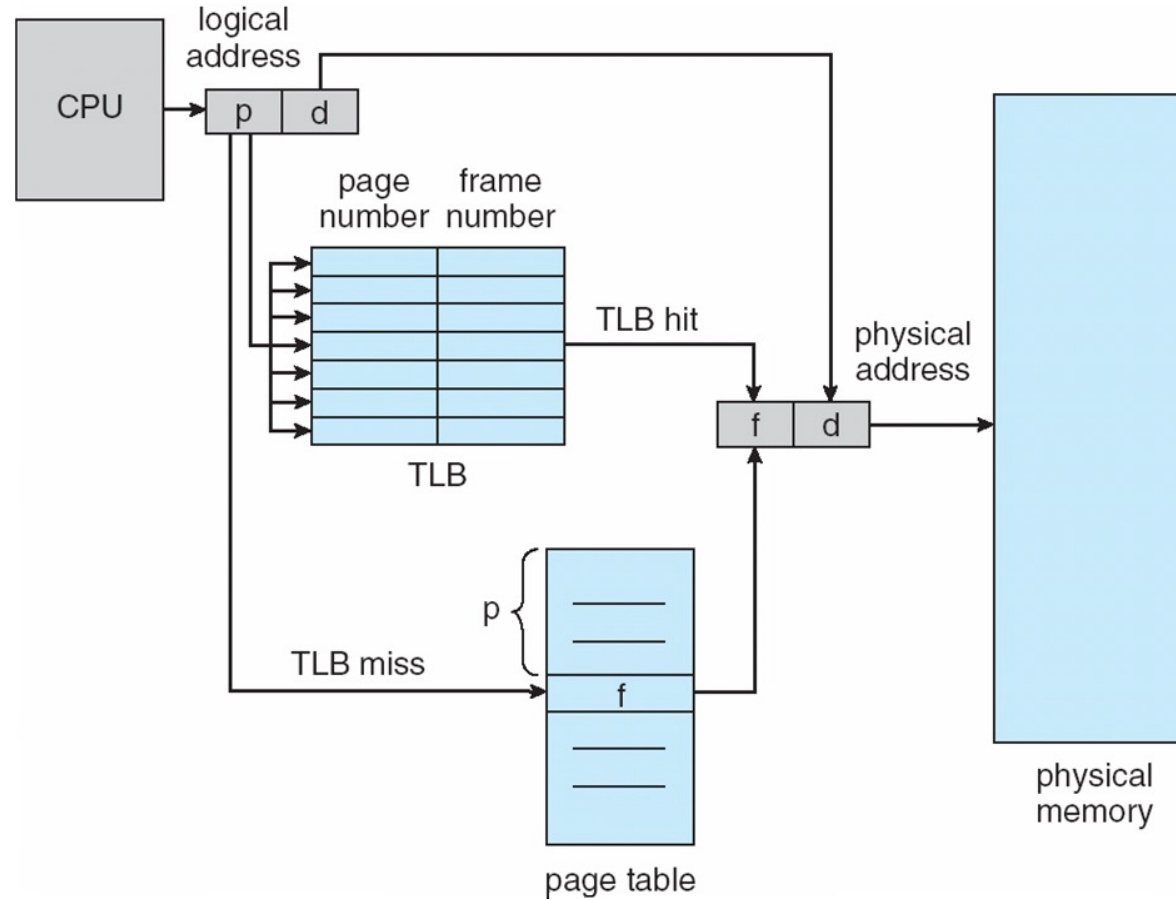
- Page table is kept in main memory
  - **Page-table base register (PTBR)** points to the page table
  - **Page-table length register (PTLR)** indicates size of the page table
  - Both, PTBR and PTLR are also stored in the process's PCB
- In this scheme every data/instruction access requires two memory accesses
  - One for the page table and one for the data / instruction
  - First: access the page table using PTBR value to retrieve its frame number
  - Second: access the actual memory location given the frame number
    - This is a serious time overhead that needs to be reduced
- The two memory access problem can be solved by the use of a **special fast-lookup hardware** cache called **translation look-aside buffers (TLBs)** (also called **associative memory**).

# Translation Look-Aside Buffer



- Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry – uniquely identifies each process to provide address-space protection for that process
  - Otherwise need to flush at every context switch
- TLBs typically small (64 to 1,024 entries)
- On a TLB miss, value is loaded into the TLB for faster access next time
  - Replacement policies must be considered
  - Some entries can be **wired down** for permanent fast access

# Paging Hardware With TLB







# Effective Memory-Access Time

- Associative Lookup Time (TLB) =  $e$  time units
- Hit ratio =  $a$ 
  - Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers

- Consider hit ratio  $a$ , TLB search time  $e$ , and memory access time  $m$

$$\text{EAT} = (m + e) * a + (2m + e) (1 - a)$$

- Consider,  $a = 80\%$ ,  $e = 20\text{ns}$ ,  $m = 100\text{ns}$ ,
  - $\text{EAT} = 0.80 \times (100 + 20) + (1 - 0.80) \times (2 \times 100 + 20) = 140\text{ns}$   
(40% slowdown)
- Consider,  $a = 99\%$ ,  $e = 20\text{ns}$ ,  $m = 100\text{ns}$ ,
  - $\text{EAT} = 0.99 \times (100 + 20) + (1 - 0.99) \times (2 \times 100 + 20) = 121\text{ns}$   
(21% slowdown)

# Memory Protection



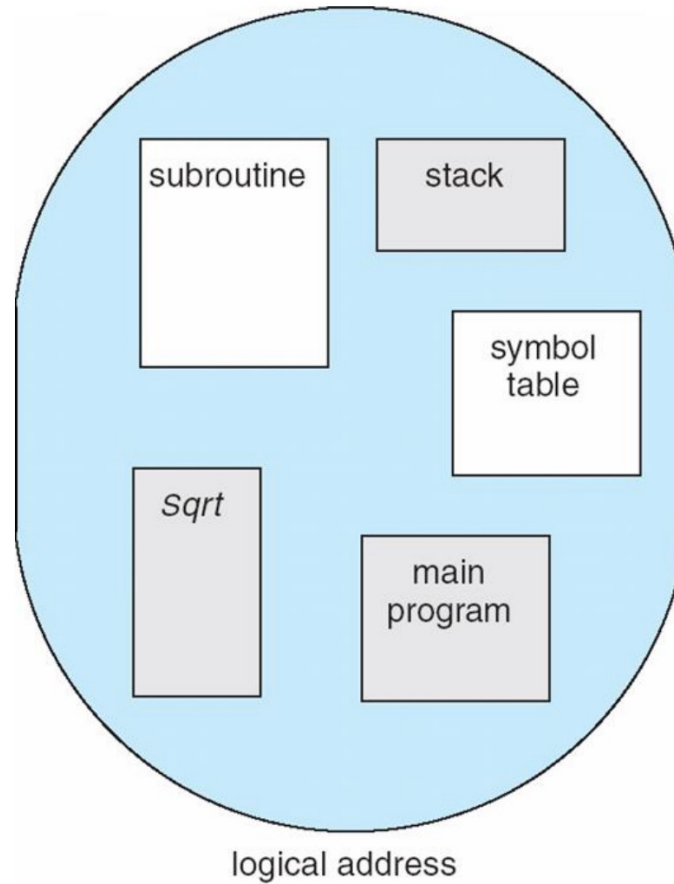
- Memory protection implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed
  - Can also add more bits to indicate page execute-only, and so on
- **Valid-invalid** bit attached to each entry in the page table:
  - “valid” indicates that the associated page is in the process’ logical address space, and is thus a legal page
  - “invalid” indicates that the page is not in the process’ logical address space
  - Or use **page-table length register (PTLR)**
- Any violations result in a trap to the kernel



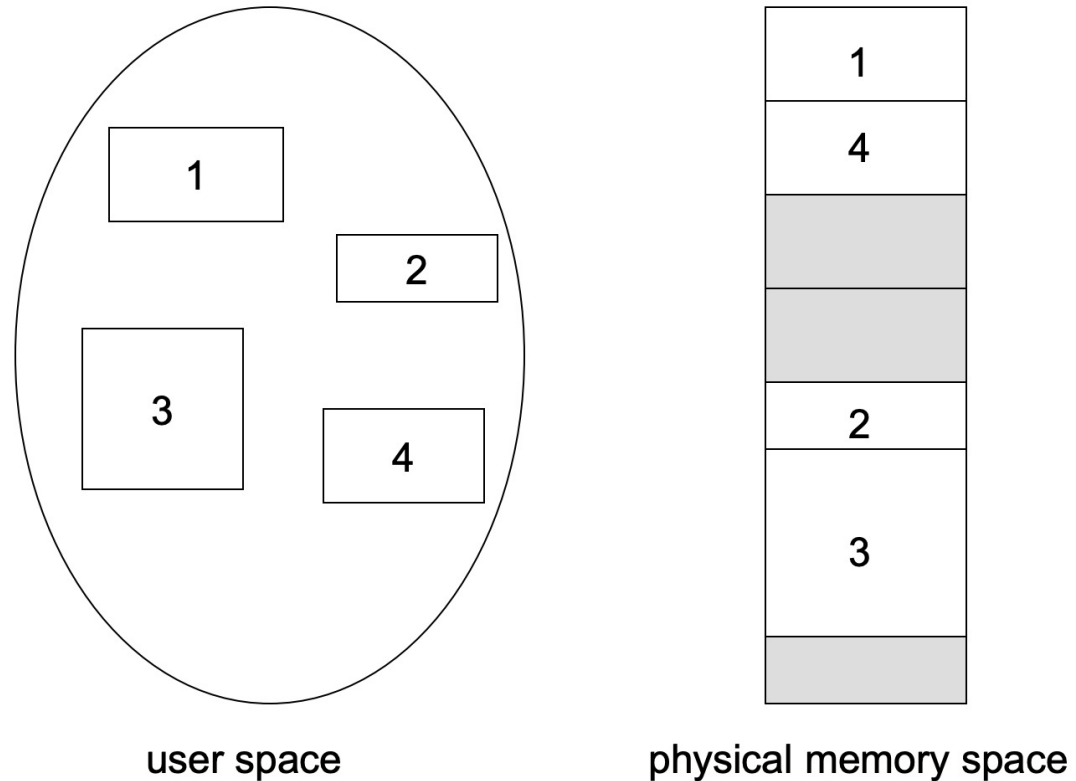
# Segmentation

- Memory-management scheme that supports user view of memory
- A program is a collection of segments
- A segment is a logical unit such as:
  - Main program
  - Procedure
  - Function
  - Method
  - Object
  - local variables and global variables
  - common block
  - stack
  - symbol table
  - arrays

# User view of a program



# Logical View of Segmentation



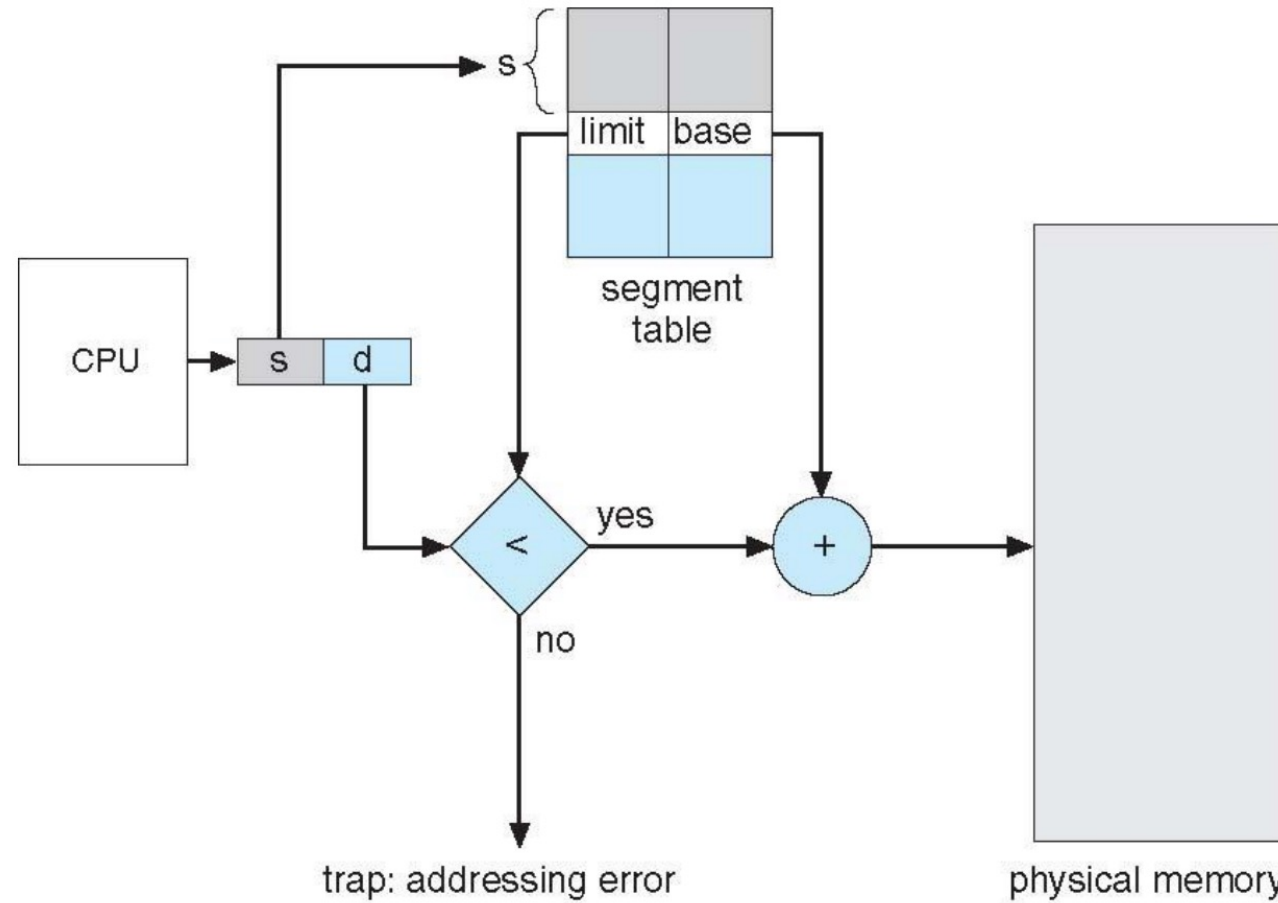


# Segmentation Architecture

- Logical address consists of a two tuple:  
    <segment number, offset>
- **Segment table** – maps two-dimensional physical addresses; each table entry has:
  - **base** – contains the starting physical address where the segments reside in memory
  - **limit** – specifies the length of the segment
- **Segment-table base register (STBR)** points to the segment table's location in memory
- **Segment-table length register (STLR)** indicates number of segments used by a program;

segment number  $s$  is legal if  $s < \text{STLR}$

# Segmentation Architecture



# Structure of the Page Table



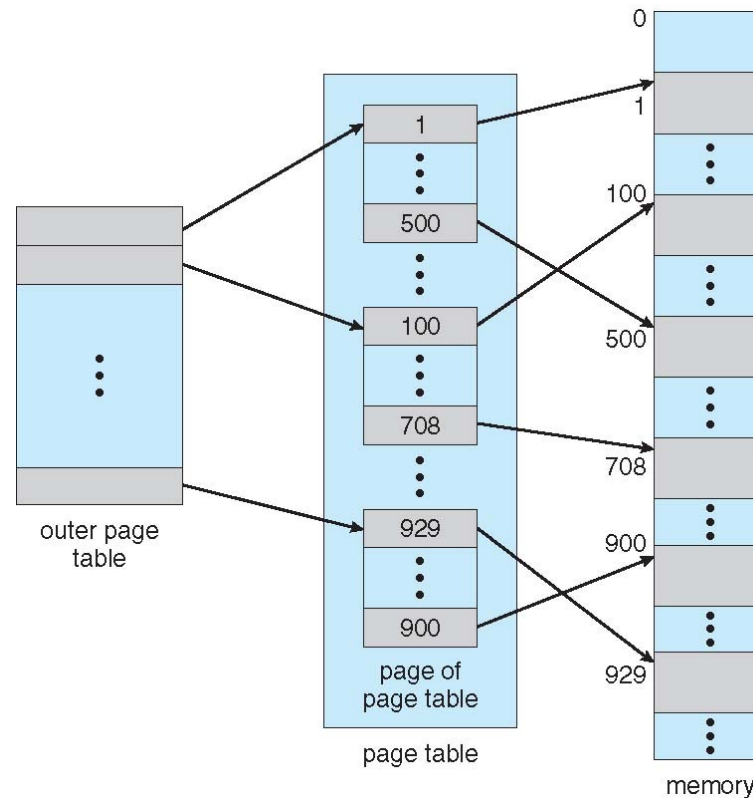
- Memory structures for paging can get huge using straight-forward methods
  - Consider a 32-bit logical address space as on modern computers
  - Page size of 4 KB ( $2^{12}$ )
  - Page table would have 1 million entries ( $2^{32} / 2^{12}$ )
  - If each entry is 4 bytes → each process 4 MB of physical address space for the page table alone
    - Don't want to allocate that contiguously in main memory
    - Very costly to store in main memory
- One simple solution is to divide the page table into smaller units
  - Hierarchical Paging
  - Hashed Page Tables
  - Inverted Page Tables



# Hierarchical Page Tables



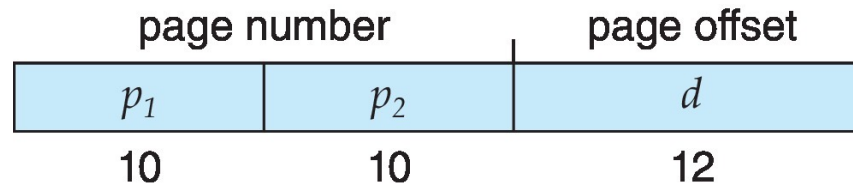
- Two-Level Paging Algorithm
  - The page table itself is also paged
- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
- We then page the page table



# Two-Level Paging Example

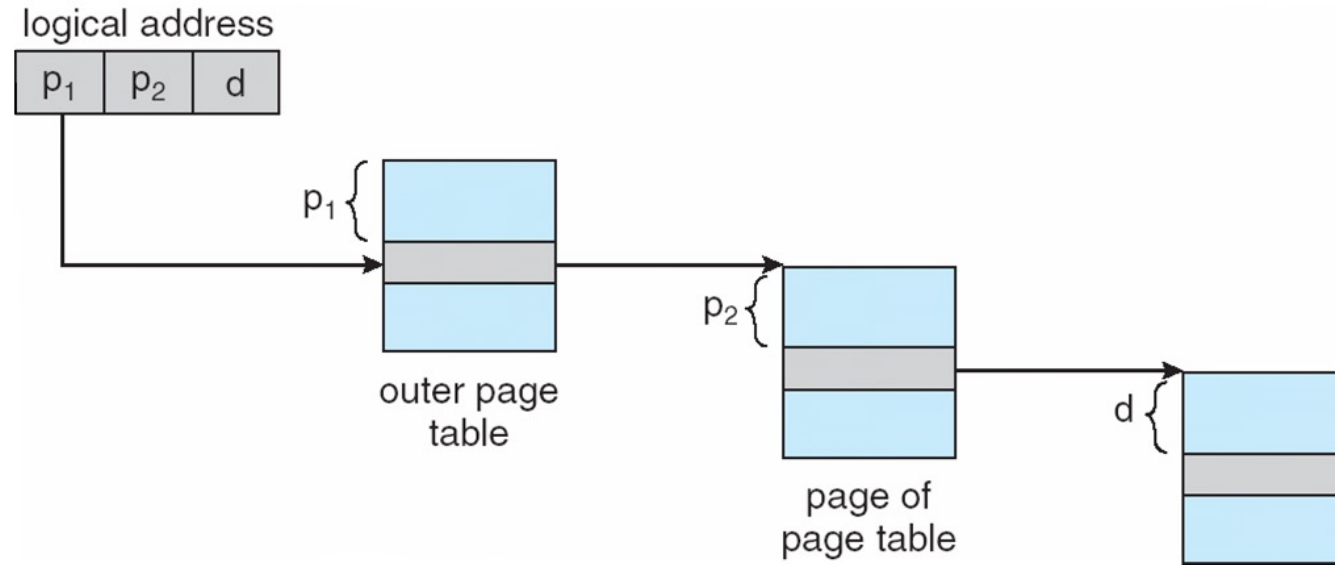


- A logical address (on 32-bit machine with 1K page size) is divided into:
  - a page number consisting of 22 bits (m-n)
  - a page offset consisting of 10 bits (n)
- Since the page table is paged, the page number is further divided into:
  - a 10-bit outer page number
  - a 12-bit inner page offset
- Thus, a logical address is as follows:



- where  $p_1$  is an index into the outer page table, and  $p_2$  is the displacement within the page of the inner page table
- Known as **forward-mapped page table**

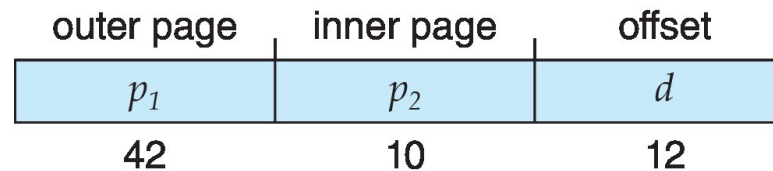
# Address-Translation Scheme



# 64-bit Logical Address Space



- Even two-level paging scheme not sufficient
  - 64-bit addressing =  $2^{64}$  of physical address space to store a page table
- If page size is 4 KB ( $2^{12}$ )
  - Then page table has  $2^{52}$  entries; number of pages
  - If two level scheme, inner page tables could be  $2^{10}$  4-byte entries
  - Address would look like



- Outer page table has  $2^{42}$  entries or  $2^{44}$  bytes
- One solution is to add a 2<sup>nd</sup> outer page table
- But in the following example the 2<sup>nd</sup> outer page table is still  $2^{34}$  bytes in size
  - ▶ And possibly 4 memory access to get to one physical memory location

# Three-level Paging Scheme



outer page	inner page	offset
$p_1$	$p_2$	$d$
42	10	12

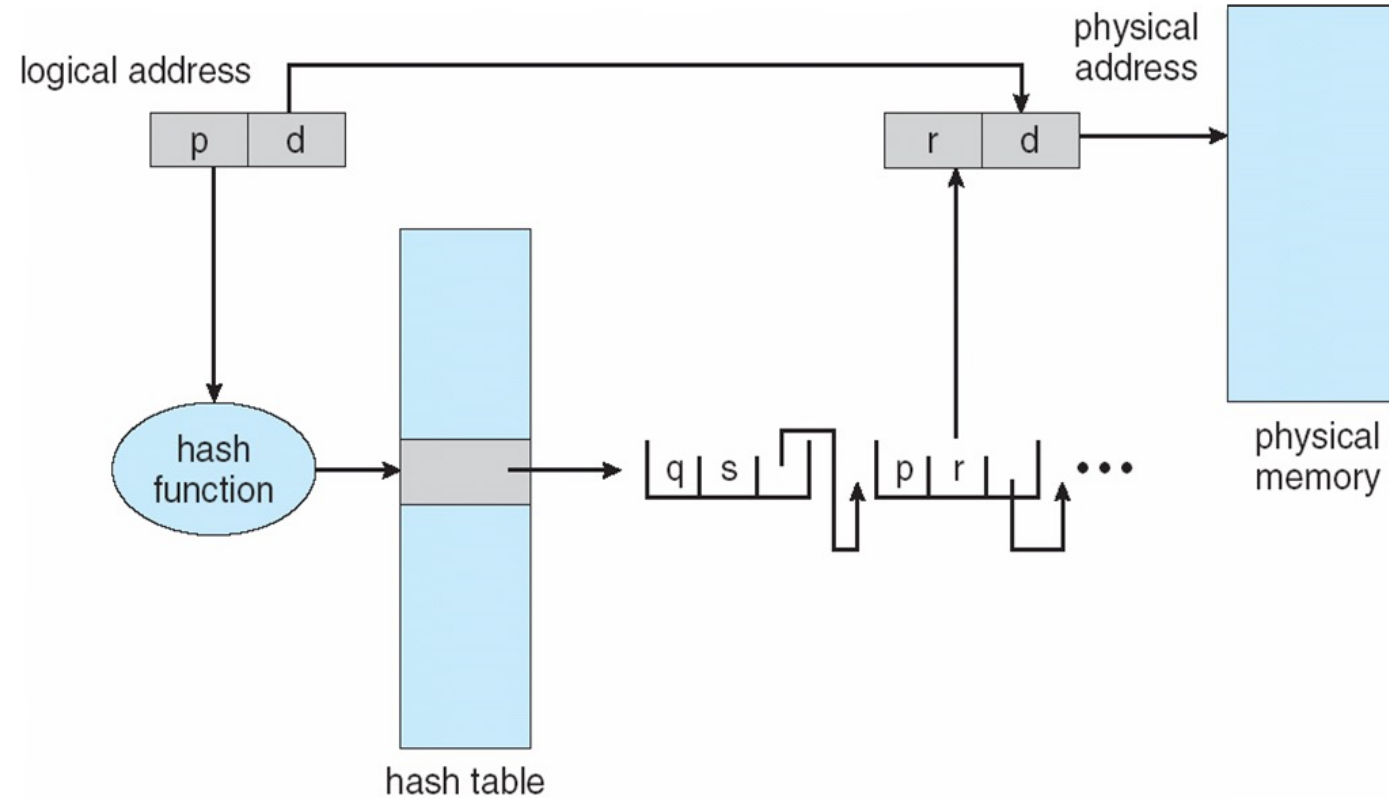
2nd outer page	outer page	inner page	offset
$p_1$	$p_2$	$p_3$	$d$
32	10	10	12

# Hashed Page Tables



- Common in address spaces  $> 32$  bits
- The virtual page number (**hash value**) is hashed into a page table
  - Each entry in table is a **linked list** of elements hashing to the same location
    - **Collision handling**
- Each element contains (1) the virtual page number (2) the value of the mapped page frame (3) a pointer to the next element
- **Algorithm:** Virtual page numbers are compared in this chain searching for a match
  - If a match is found, the corresponding physical frame is extracted
- Variation for 64-bit addresses is **clustered page tables**
  - Similar to hashed but each entry refers to several pages (such as 16) rather than 1
  - Especially useful for **sparse** address spaces (where memory references are non-contiguous and scattered)

# Hashed Page Table



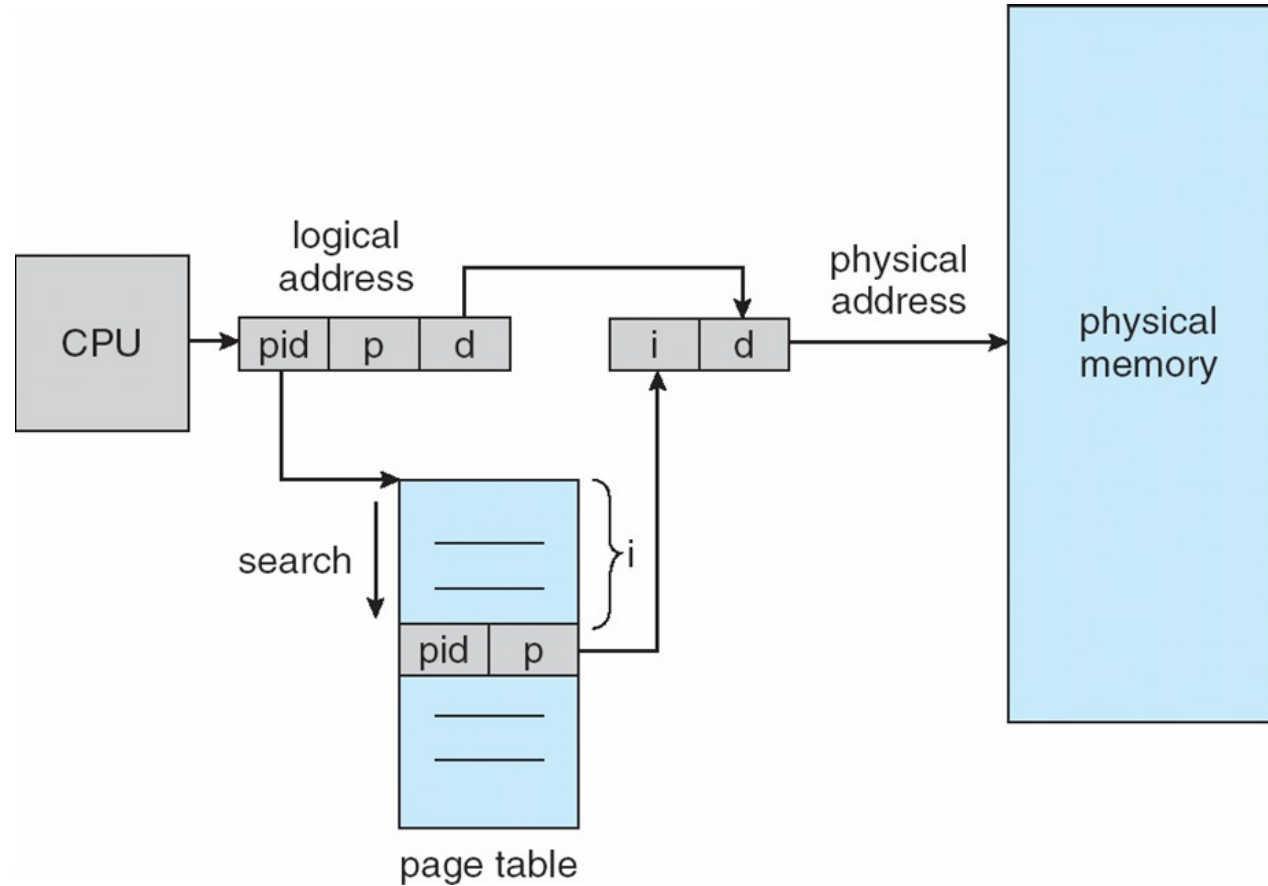
# Inverted Page Table



- Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages
- One entry for each real page of memory
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
- Use hash table to limit the search to one — or at most a few — page-table entries
  - TLB can accelerate access
- But how to implement shared memory?
  - One mapping of a virtual address to the shared physical address



# Inverted Page Table Architecture





# Virtual Memory

# Background



- Code needs to be in memory to execute, but entire program rarely used
  - Error code, unusual routines, large data structures
- Entire program code not needed at same time
- Consider ability to execute partially-loaded program
  - Program no longer constrained by limits of physical memory
  - Each program takes less memory while running -> more programs run at the same time
    - Increased CPU utilization and throughput with no increase in response time or turnaround time
  - Less I/O needed to load or swap programs into memory -> each user program runs faster

# Virtual memory



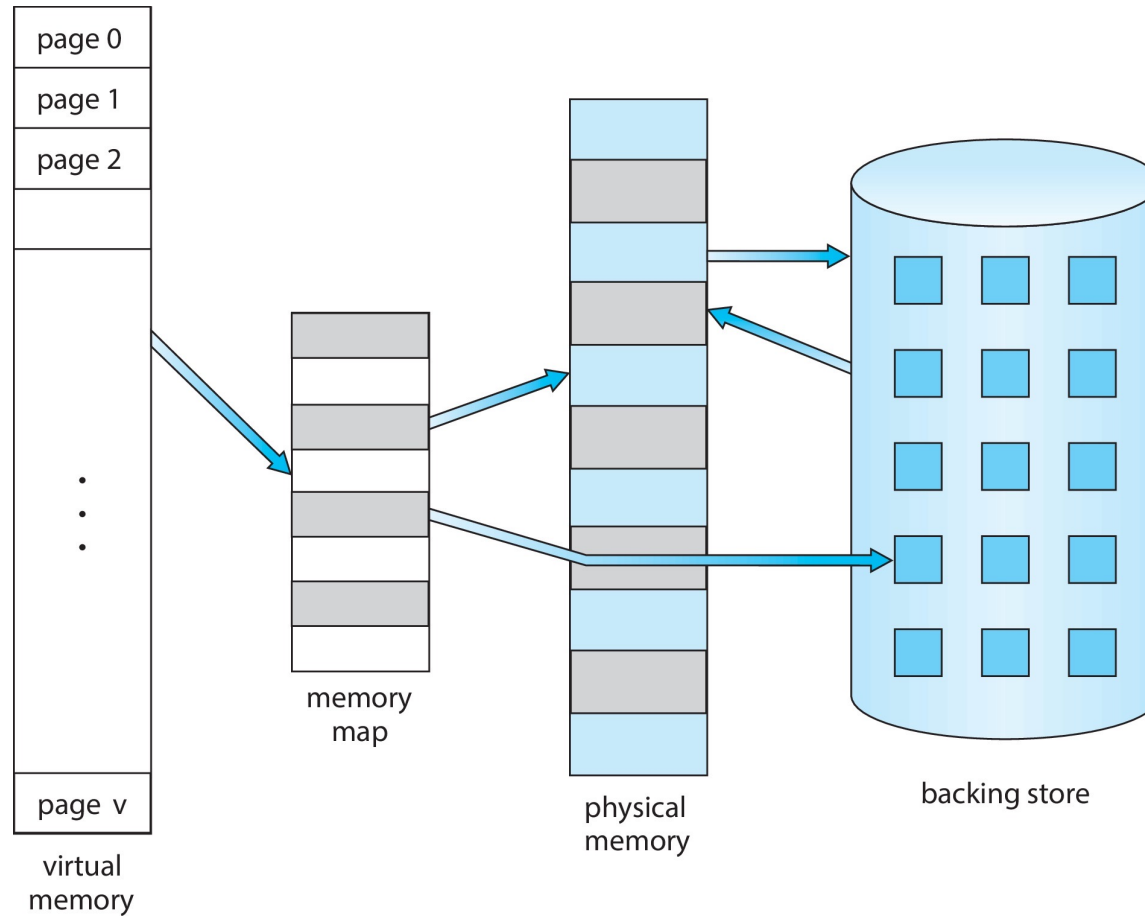
- **Virtual memory** – separation of user logical memory from physical memory
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Allows for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

# Virtual memory (Cont.)



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

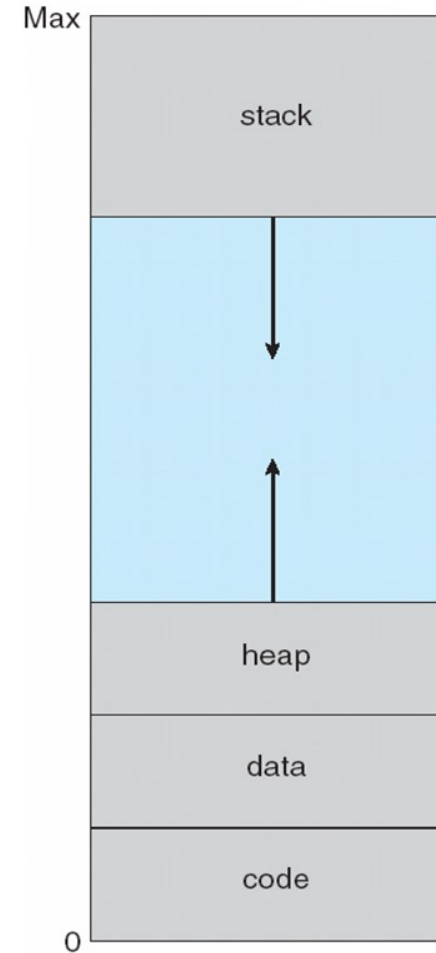
# Virtual Memory That is Larger Than Physical Memory



# Virtual-address Space



- Usually design logical address space for stack to start at Max logical address and grow “down” while heap grows “up”
  - Maximizes address space use
  - Unused address space between the two is hole
    - ▶ No physical memory needed until heap or stack grows to a given new page
- Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc
- System libraries shared via mapping into virtual address space
- Shared memory by mapping pages read-write into virtual address space
- Pages can be shared during `fork()`, speeding process creation



# Demand Paging



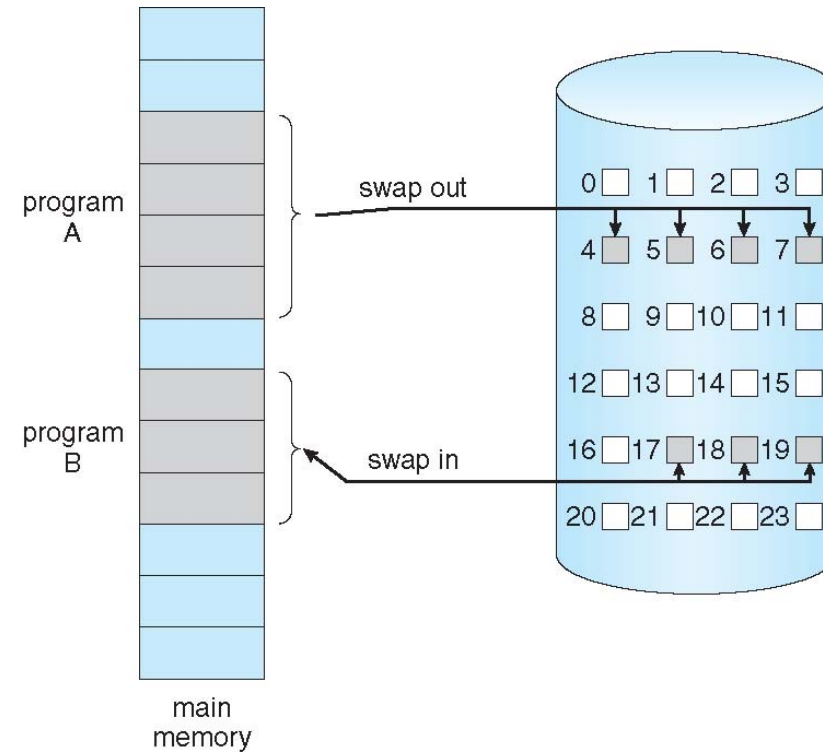
- Could bring entire process into memory at load time
- Or bring a page into memory only when it is needed
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users
- Similar to paging system with swapping
- Page is needed  $\Rightarrow$  reference to it
  - invalid reference  $\Rightarrow$  abort
  - not-in-memory  $\Rightarrow$  bring to memory
- **Lazy swapper** – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**



# Demand Paging



- Could bring entire process into memory at load time
- Or bring a page into memory only when it is needed
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users
- Similar to paging system with swapping (diagram on right)



# 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

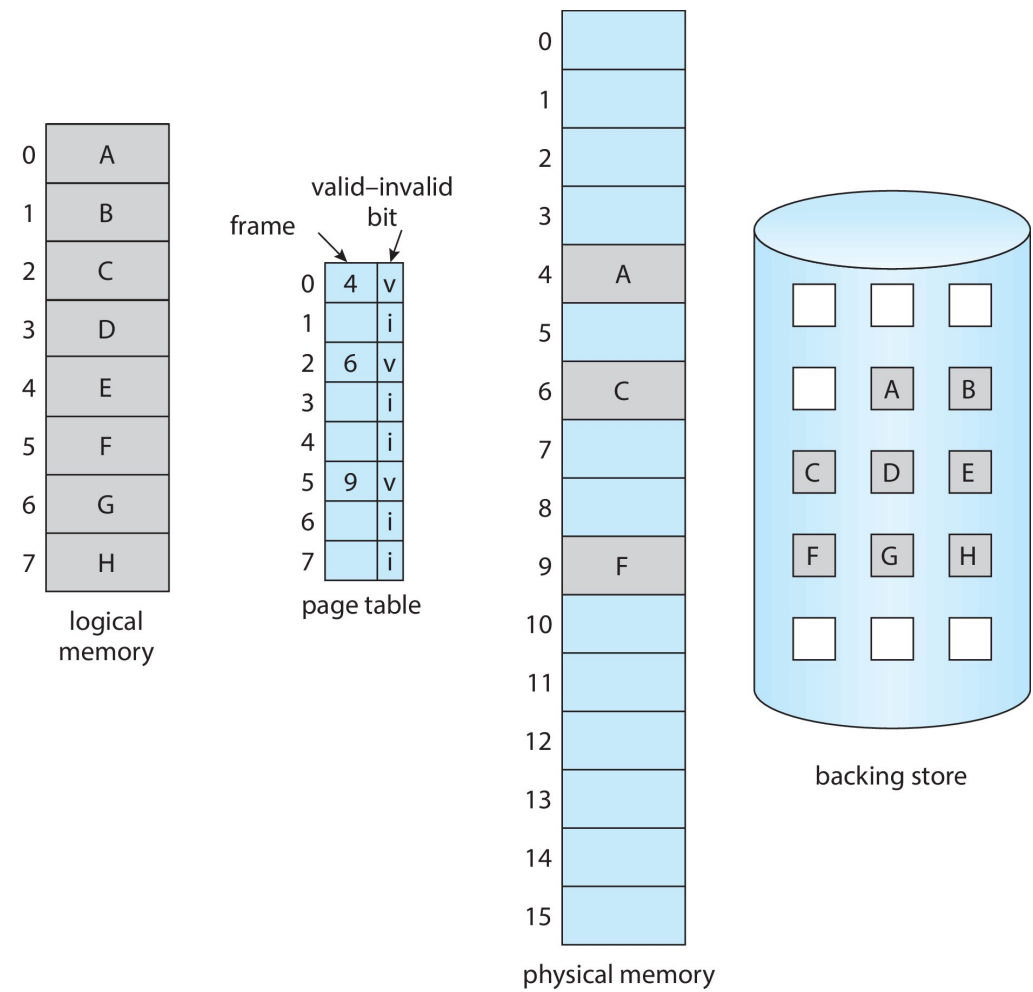
- With each page table entry a valid–invalid bit is associated (**v**  $\Rightarrow$  in-memory – **memory resident**, **i**  $\Rightarrow$  not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:

Frame #	valid-invalid bit
	<b>v</b>
	<b>v</b>
	<b>v</b>
	<b>i</b>
...	
	<b>i</b>
	<b>i</b>

page table

- During MMU 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



# Steps in Handling Page Fault



1.If there is a reference to a page, first reference to that page will trap to operating system

- Page fault

2.Operating system looks at another table to decide:

- Invalid reference  $\Rightarrow$  abort

- Just not in memory

3.Find free frame

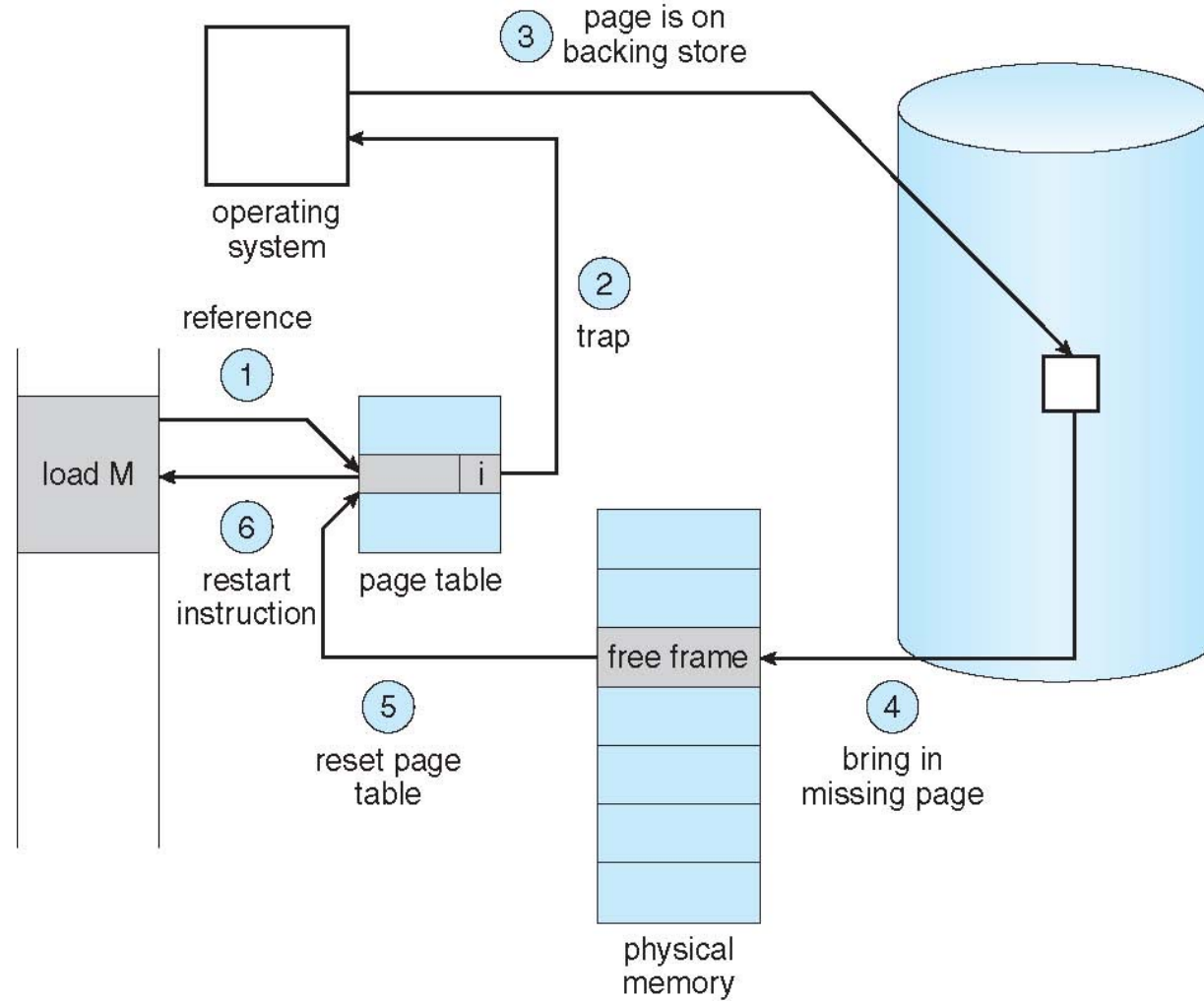
4.Swap page into frame via scheduled disk operation

5.Reset tables to indicate page now in memory

Set validation bit = **v**

6.Restart the instruction that caused the page fault

# Steps in Handling a Page Fault (Cont.)



# Performance of Demand Paging



- 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)
  - EAT =  $(1 - p)$  x memory access
  - +  $p$  (page fault overhead
  - + swap page out
  - + swap page in )

# 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)

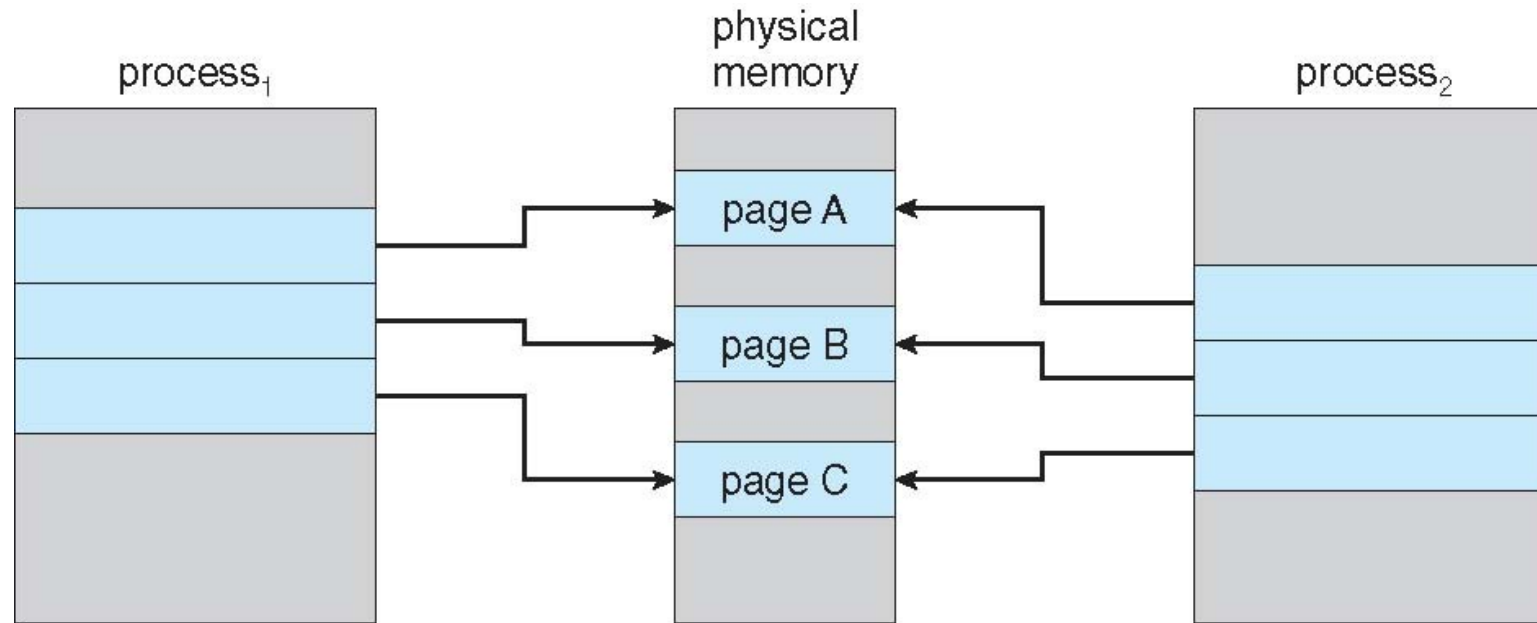


# Copy-on-Write

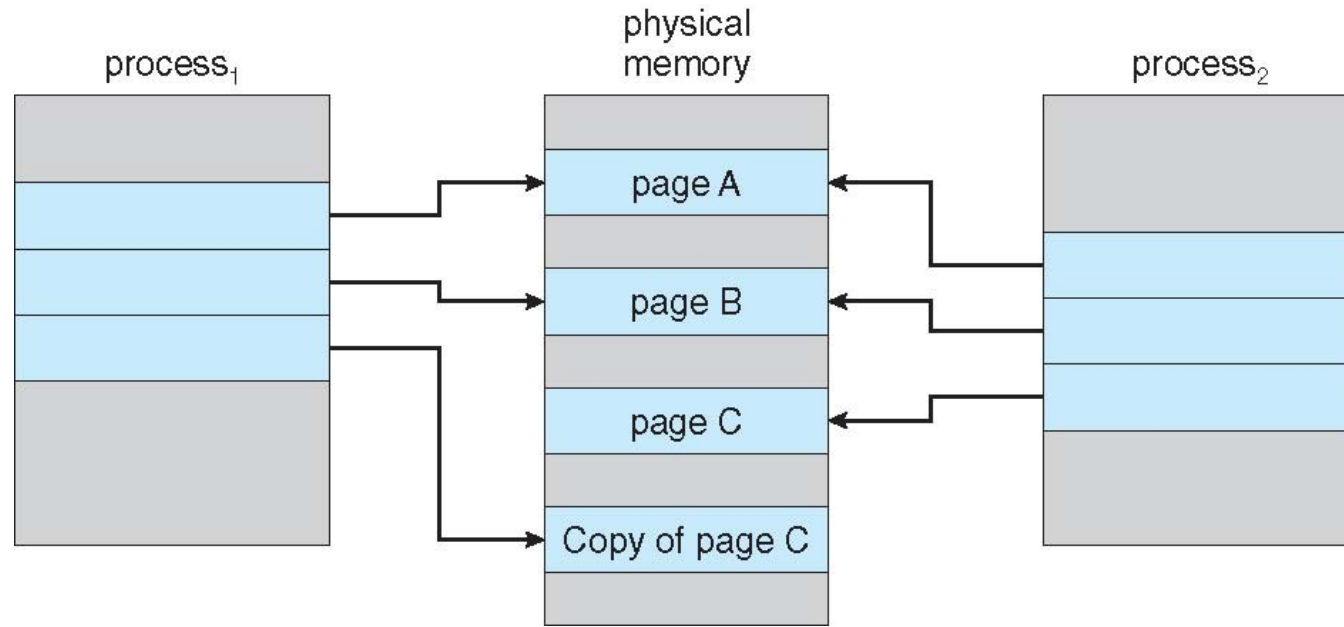


- **Copy-on-Write** (COW) allows both parent and child processes to initially *share* the same pages in memory
  - If either process modifies a shared page, only then is the page copied
- COW allows more efficient process creation as only modified pages are copied
- In general, free pages are allocated 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 suspend and child using copy-on-write address space of parent
  - Designed to have child call `exec()`
  - Very efficient

# Before Process 1 Modifies Page C



# After Process 1 Modifies Page C



# What Happens if There is no Free Frame?



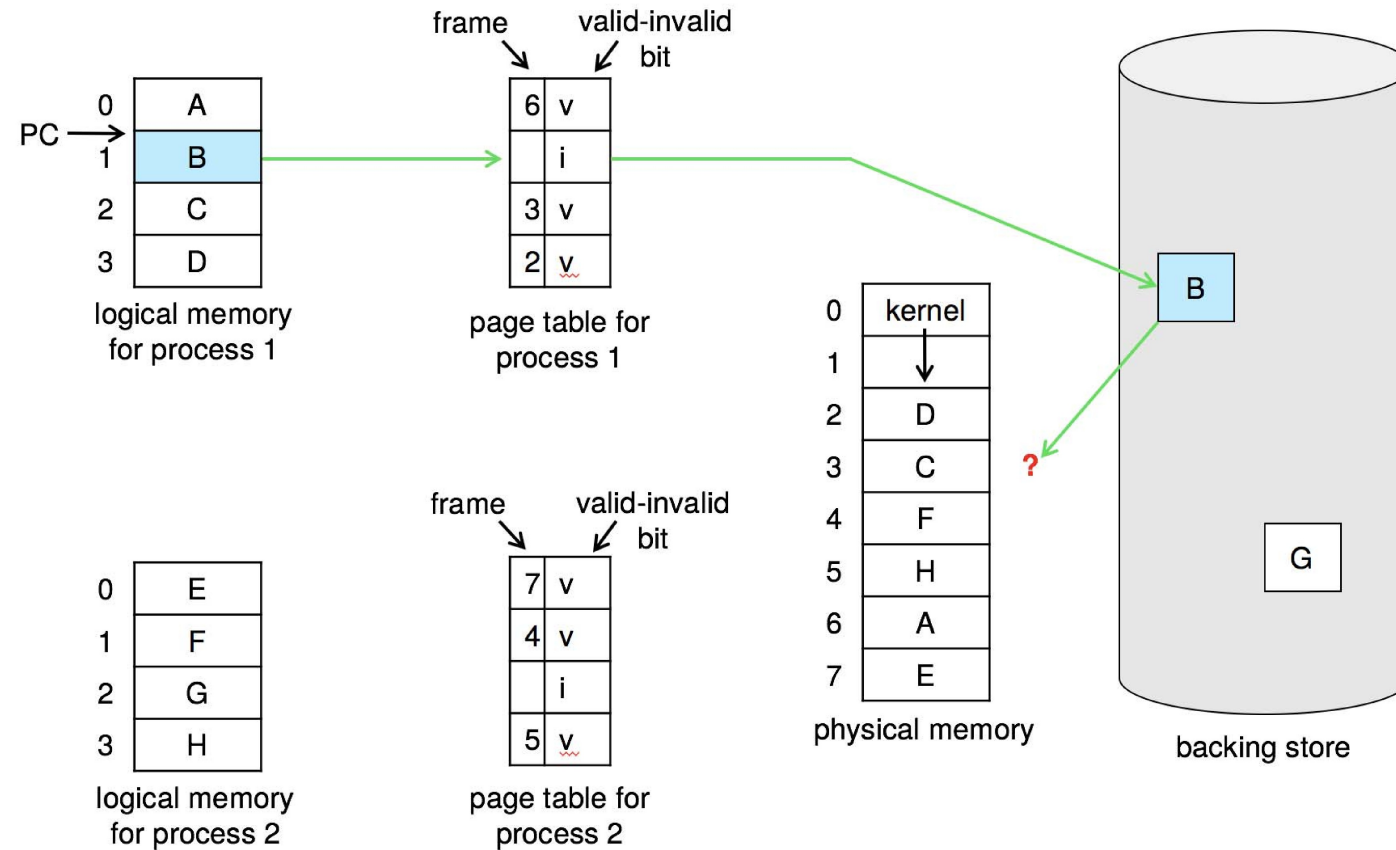
- Used up by process pages
- Also in demand from the kernel, I/O buffers, etc
- How much to allocate to each?
- Page replacement – find some page in memory, but not really in use, page it out
  - Algorithm – terminate? swap out? replace the page?
  - Performance – want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times

# Page Replacement



- Prevent **over-allocation** of memory by modifying page-fault service routine to include page replacement
- Use **modify (dirty) bit** to reduce overhead of page transfers – only modified pages are written to disk
- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory

# Need For Page Replacement



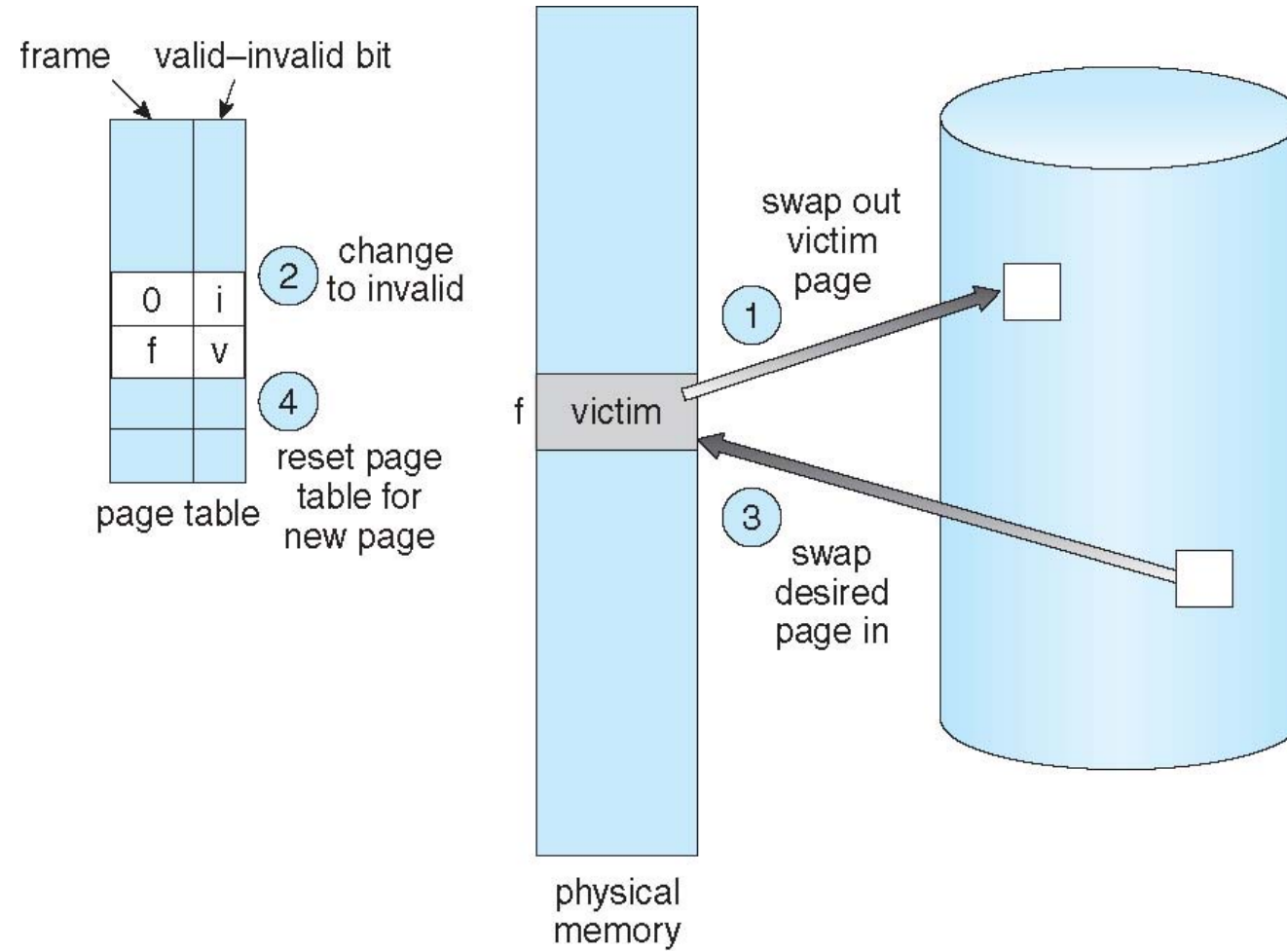
# Basic Page Replacement



1. Find the location of the desired page on disk
2. Find a free frame:
  - If there is a free frame, use it
  - If there is no free frame, use a page replacement algorithm to select a **victim frame**
  - Write victim frame to disk if dirty
3. Bring the desired page into the (newly) free frame; update the page and frame tables
4. Continue the process by restarting the instruction that caused the trap

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

# Page Replacement





# Page and Frame Replacement Algorithms



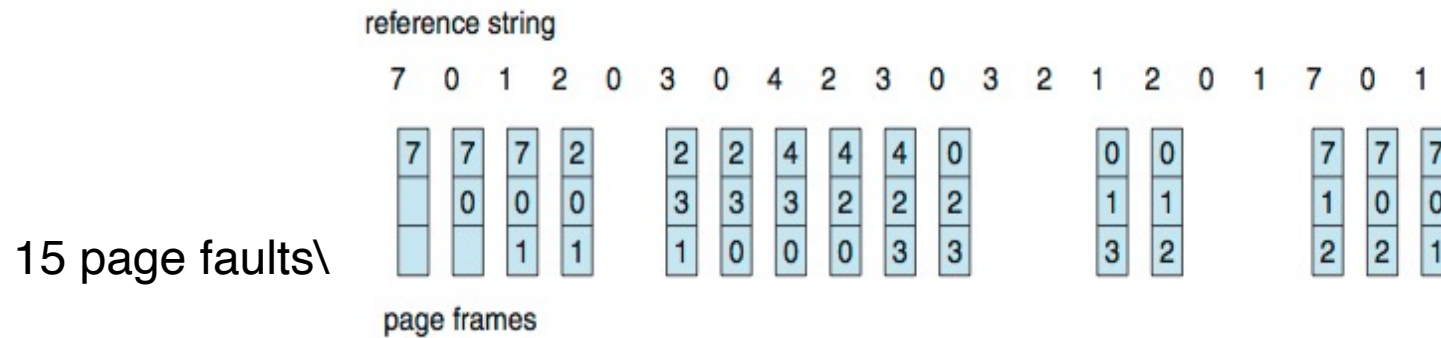
- **Frame-allocation algorithm** determines
  - How many frames to give each process
  - Which frames to replace
- **Page-replacement algorithm**
  - Want lowest page-fault rate on both first access and re-access
- Evaluate algorithm by running it on a particular string of memory references (reference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

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



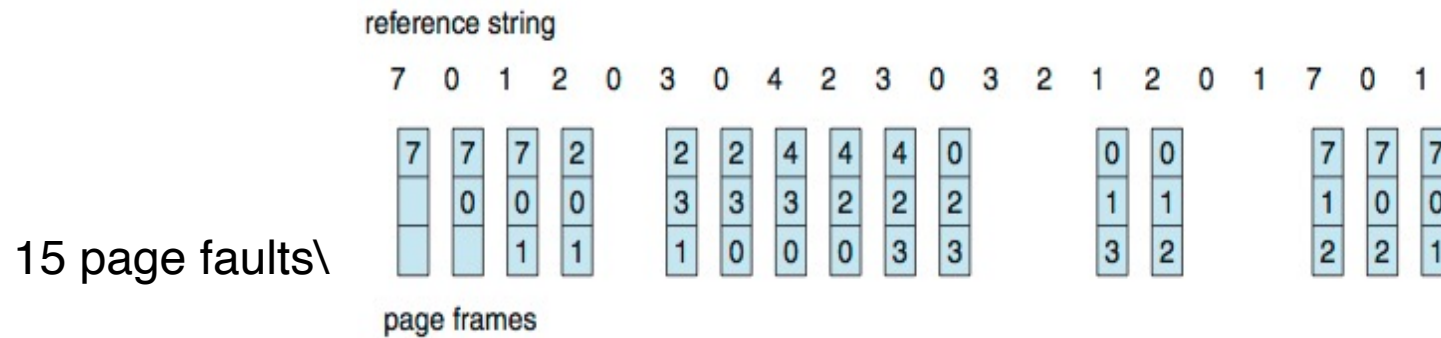
- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)



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

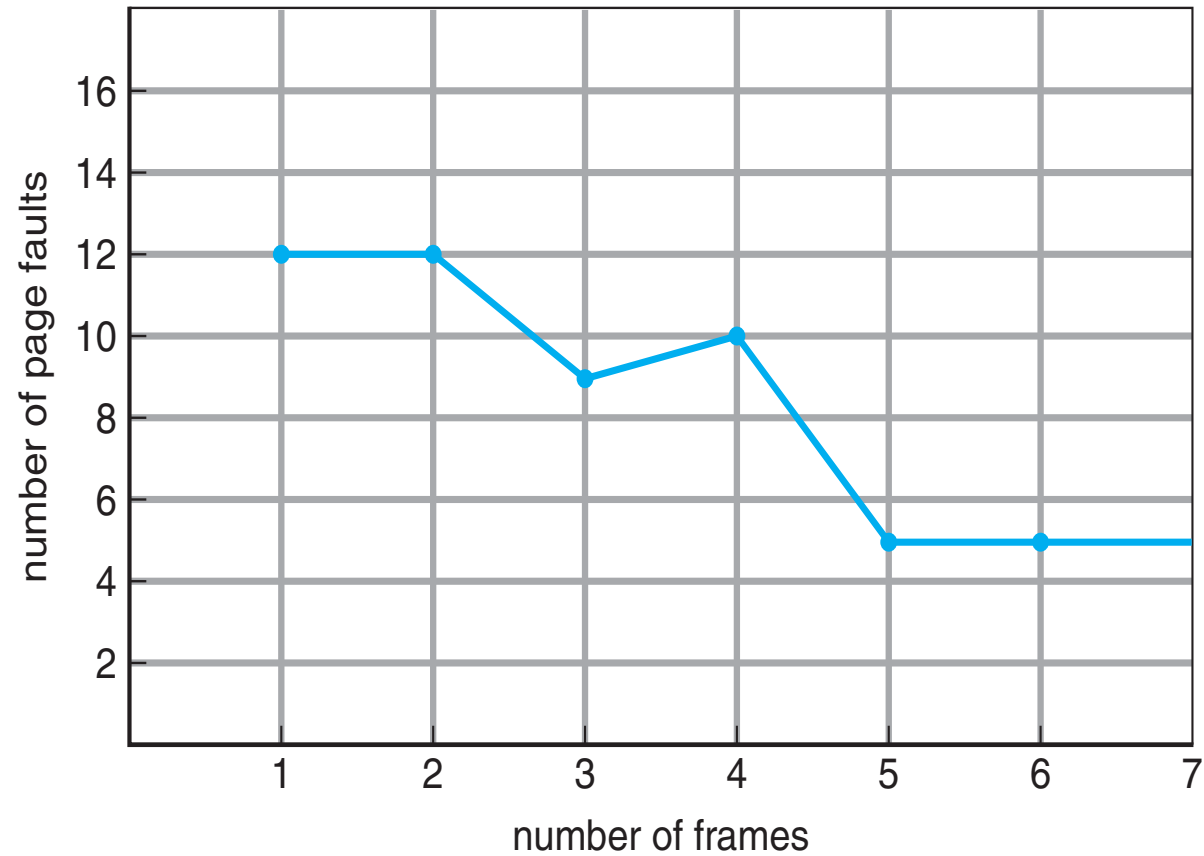


- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)



- Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5

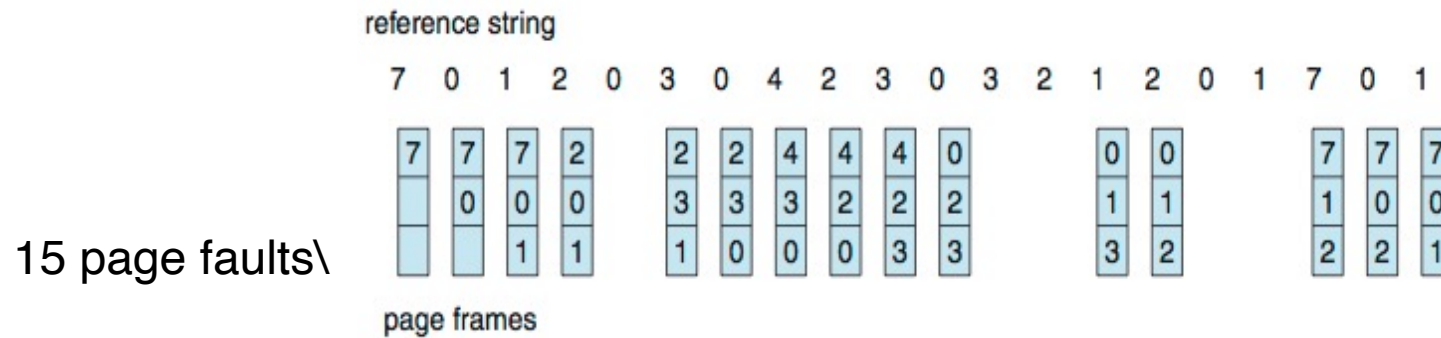
# FIFO Illustrating Belady's Anomaly



# 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)



- 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

# Optimal Algorithm



- Replace page that will not be used for longest period of time
  - 9 is 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

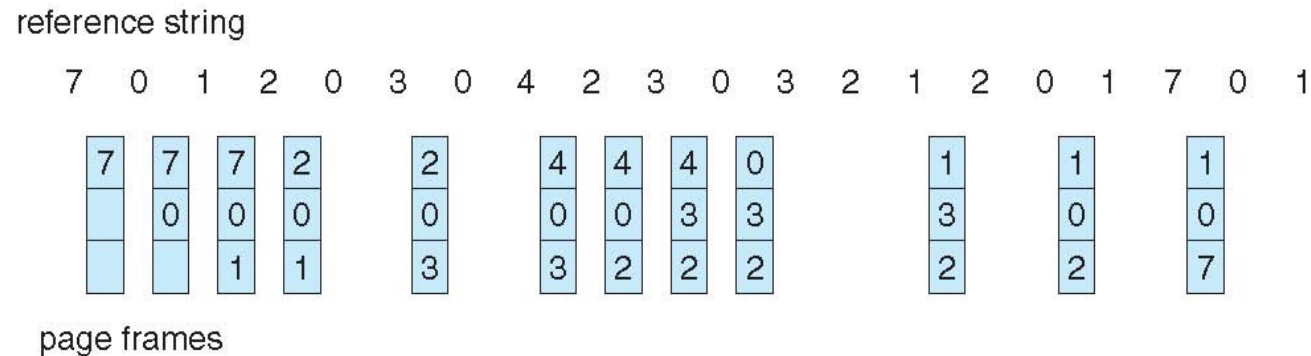
7	7	7	2		2		2		2		2						7		
	0	0	0		0		0		0		0						0		
		1	1		3		3		3		1						1		

page frames

# 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?

# 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

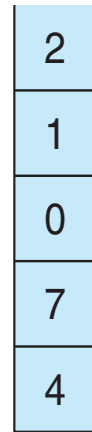


## Use Of A Stack to Record Most Recent Page References



reference string

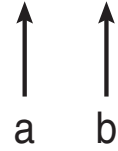
4 7 0 7 1 0 1 2 1 2 7 1 2



stack  
before  
a



stack  
after  
b





# Page replacement

- Consider the following page reference strings:  
7, 2, 3, 1, 2, 5, 3, 4, 6, 7, 7, 1, 0, 5, 4, 6, 2, 3, 0, 1
- Assume demand paging with 3 frames, how many page faults will occur for:
  - LRU Page Replacement Algorithm
  - FIFO Page Replacement Algorithm
  - Optimal Page Replacement Algorithm