Open Source OS

Module 2: Memory Management

Module 2: Memory Management

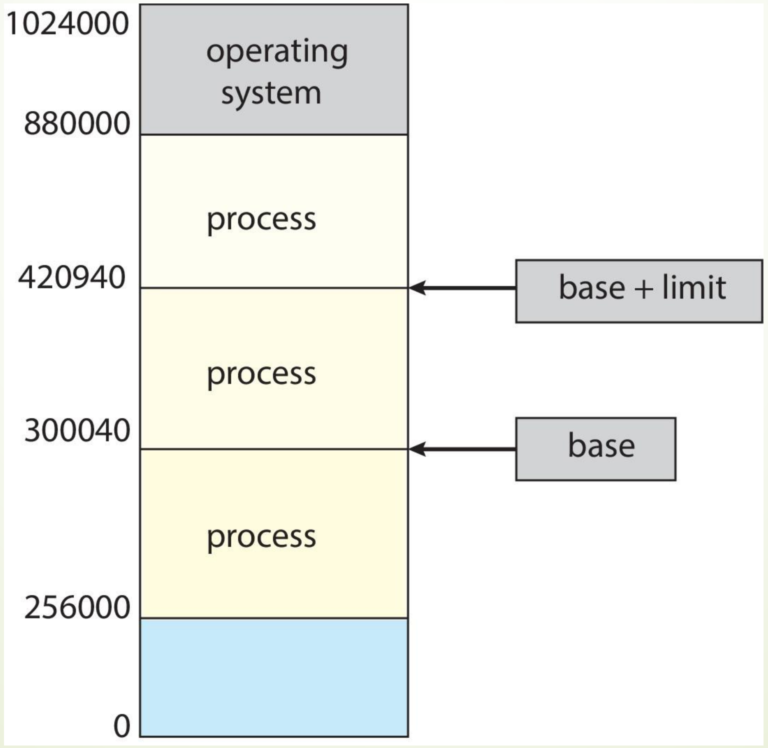
- Memory Allocation Methods
  - Contiguous Memory Allocation
  - Paging
  - Segmentation
- Swapping
- Examples: The Intel 32/64-bit and ARMv8 Architectures
- Virtual Memory
  - Demand Paging
  - Copy-on-Write
  - Page Replacement
  - Thrashing
  - Memory-Mapped Files

Background

- Program must be loaded (from disk) into memory to be run
  - Instruction, data
- Memory unit only sees a stream of:
  - addresses + read requests, or
  - addresses + data and write requests
- The only storage CPU can access directly
  - Register access is done in one CPU clock (or less)
  - Main memory can take many cycles, causing a **stall**
  - **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

Protection

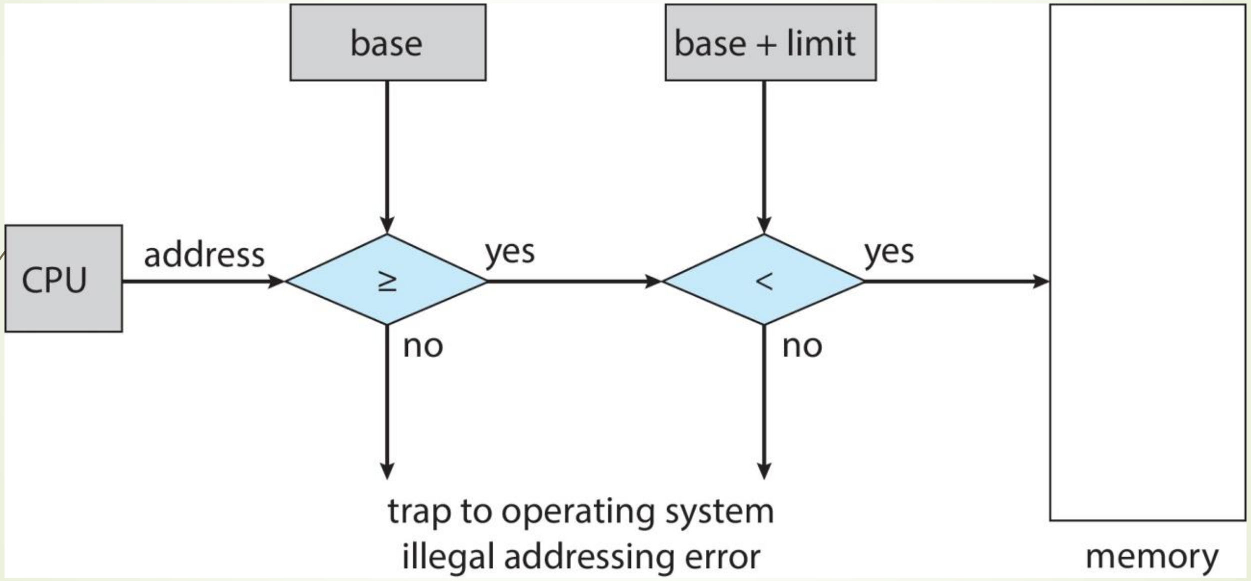
- To ensure a process can only access allowed addresses
- Can be protected by using a pair of **base** and **limit registers** defining the logical address space of a process



[Source: Operating Systems Concepts, 10th ed.]

Hardware Address Protection

- CPU must check every memory access generated in user mode to ensure it's between base and limit for that process



- The instructions to load the base and limit registers are privileged

[Source: Operating Systems Concepts, 10th ed.]

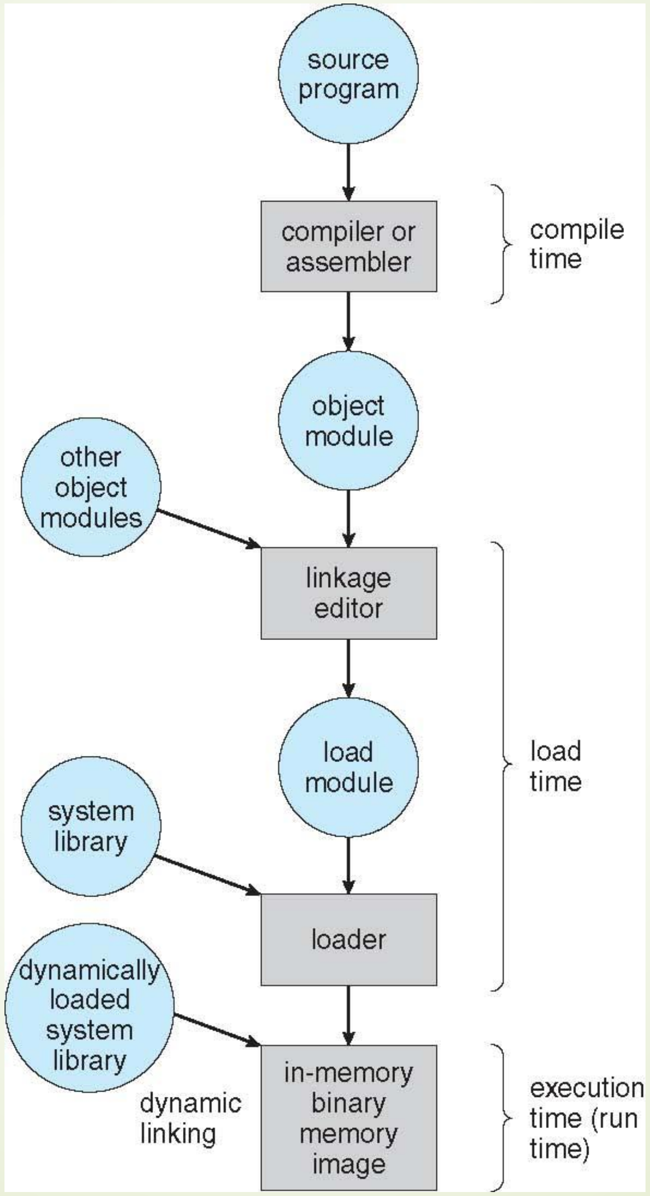
Address Binding

- Programs on disk, ready to be brought into memory to execute form an **input queue**
  - Without support, must be loaded into address 0000
  - Inconvenient to have first user physical address always at 0000
- Addresses represented in different ways at different stages of a program's life
  - Source code addresses usually symbolic
  - Compiled code addresses **bind** to relocatable addresses
    - i.e., "14 bytes from beginning of this module"
  - Linker or loader will bind relocatable addresses to absolute addresses
    - i.e., 74014
  - Each binding maps one address space to another

Binding of Instructions and Data to Memory

- Address binding of instructions and data to memory addresses can happen at three different stages
  - **Compile time**: If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
  - **Load time**: Must generate **relocatable code** if memory location is not known at compile time
  - **Execution time**: Binding delayed until run time if the process can be moved during its execution in memory
    - Need hardware support for address maps (e.g., base and limit registers)

Multistep Processing of a User Program



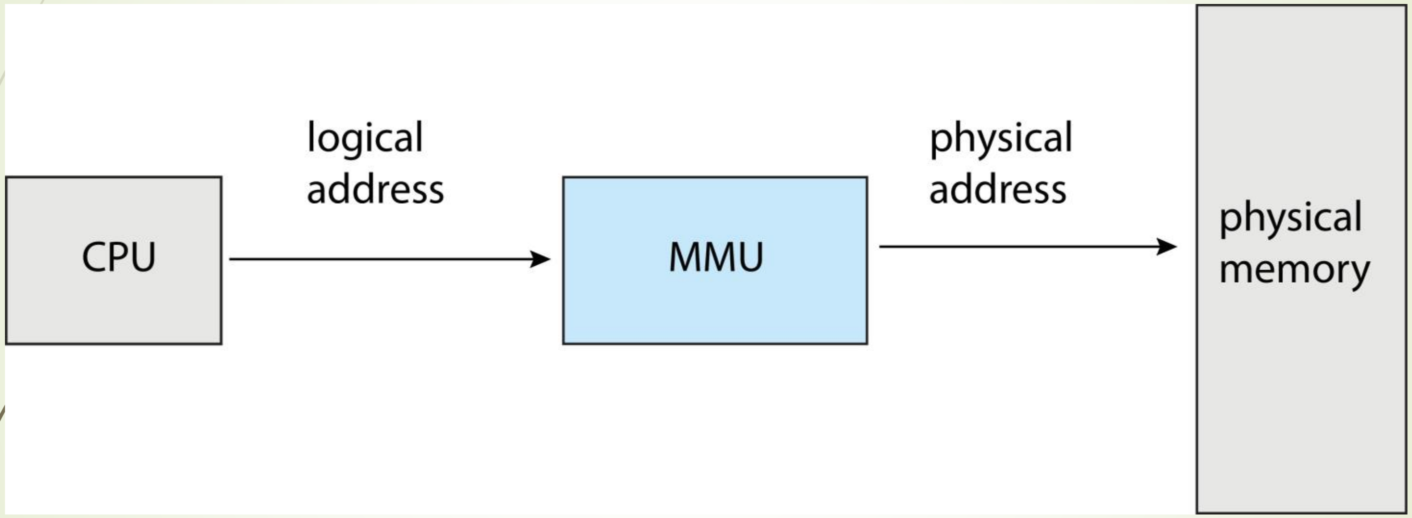
[Source: Operating Systems Concepts, 10th ed.]

Logical vs. Physical Address Space

- The concept of a logical address space that is bound to a separate physical address space is central to proper memory management
  - **Logical address** – generated by the CPU (**virtual address)**
  - **Physical address** – address seen by the memory unit
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; they differ in execution-time address-binding scheme
- **Logical address space** is the set of all logical addresses generated by a program
- **Physical address space** is the set of all physical addresses generated by a program

Memory-Management Unit (MMU)

- Hardware device that maps virtual to physical address at run time



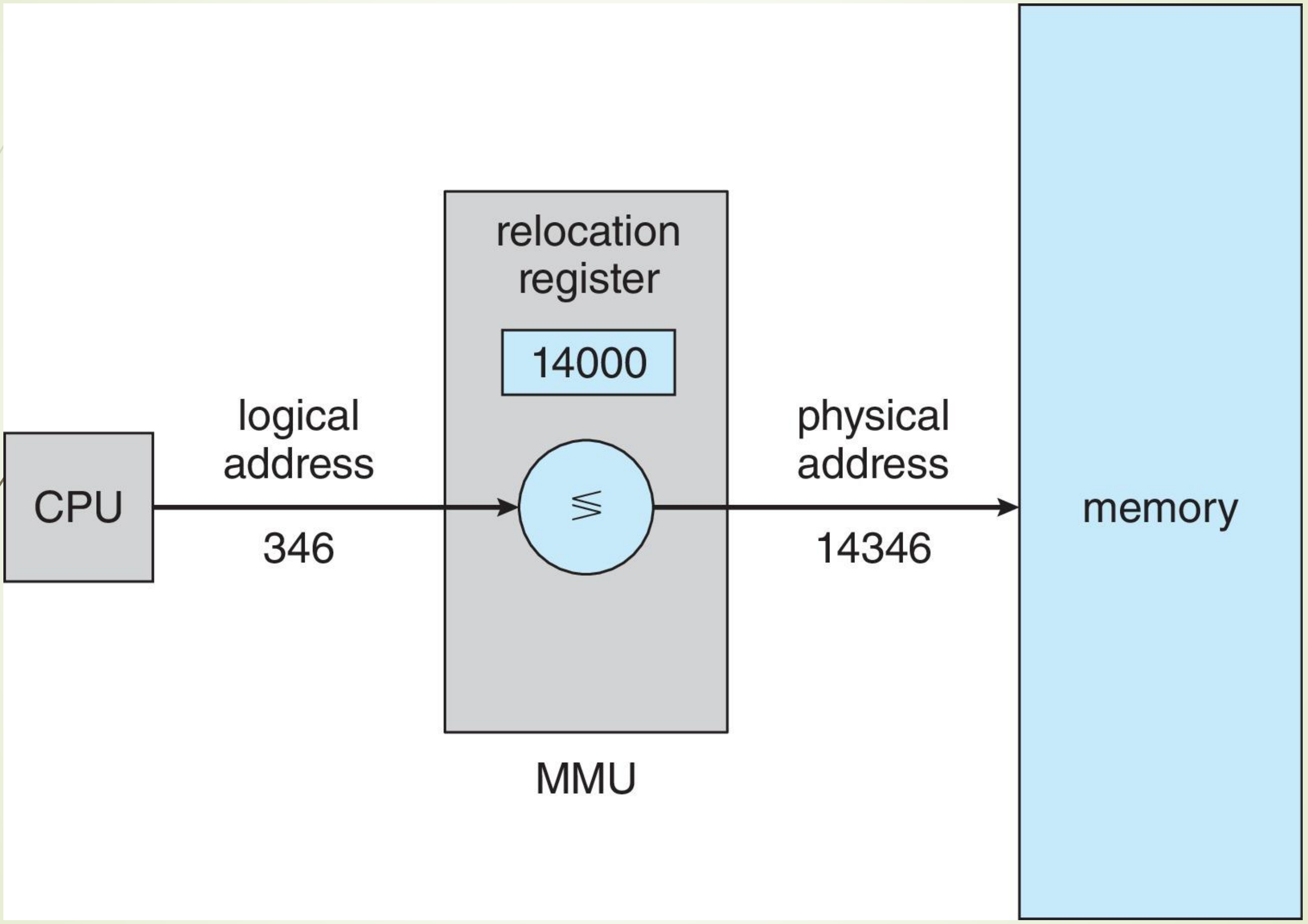
- Many methods possible, covered in the rest of this chapter

[Source: Operating Systems Concepts, 10th ed.]

Memory-Management Unit (Cont.)

- The value in the relocation register is added to every address generated by a user process at the time it's sent to memory
  - The base register now called **relocation register**
- The user program deals with *logical* addresses; it never sees the *real* physical addresses
  - Execution-time binding occurs when reference is made to location in memory
  - Logical address bound to physical addresses

Memory-Management Unit (Cont.)



[Source: Operating Systems Concepts, 10th ed.]

Memory Allocation Methods

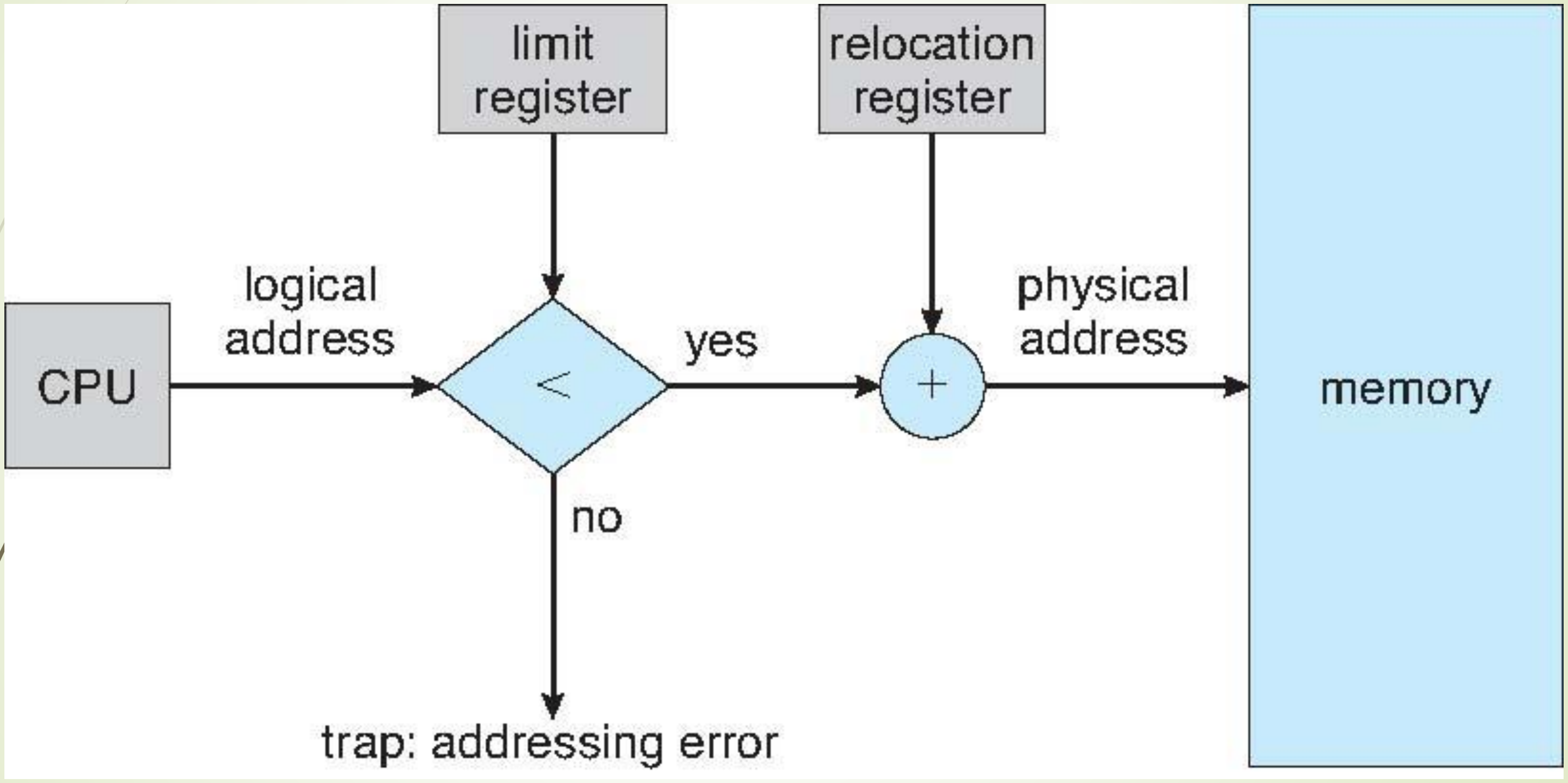
Contiguous Allocation

- Contiguous allocation is one early method
  - Main memory must support both OS and user processes
  - Limited resource, must allocate efficiently
- Main memory usually divided into two **partitions**:
  - Resident OS, usually held in low memory with interrupt vector
  - User processes then held in high memory
    - Each process contained in single contiguous section of memory

Contiguous Allocation (Cont.)

- Relocation registers used to protect user processes from each other, and from changing OS code and data
  - Base register contains value of smallest physical address
  - Limit register contains range of logical addresses – each logical address must be less than the limit register
  - MMU maps logical address *dynamically*
  - Can then allow actions such as kernel code being **transient** and kernel changing size
    - E.g. for device driver not commonly used

Hardware Support for Relocation and Limit Registers



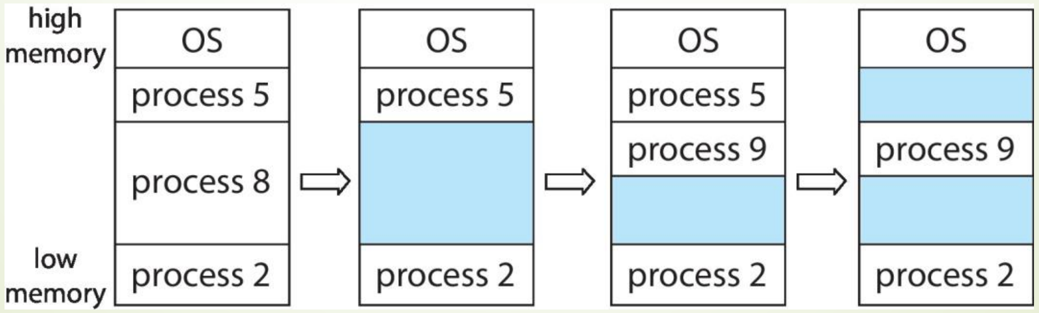
[Source: Operating Systems Concepts, 10th ed.]

Memory Allocation

- Multiple-partition allocation
  - In the simplest method of fixed-size partition, degree of multiprogramming limited by number of partitions
    - It was used by IBM OS/360, but no longer in use
  - A generalization of fixed-partition scheme is used for batch environments, and also applicable to time-sharing systems for pure segmentation

Variable Partition

- **Variable-partition** sizes for efficiency (sized to processes' needs)
  - **Hole** – block of available space scattered in memory (in various sizes)
    - 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
  - OS maintains information about:
    a) allocated partitions   b) free partitions (hole)



[Source: Operating Systems Concepts, 10th ed.]

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
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- 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.)

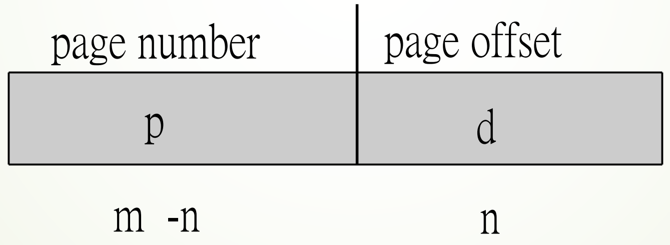
- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time
  - The cost is expensive by moving all processes toward one end of memory
- Another solution is to permit logical address space to be noncontiguous
  - Segmentation
  - Paging

Paging

- Physical address space of a process can be noncontiguous
  - Avoids external fragmentation
  - Avoids problem of varying sized memory chunks
- Divide physical memory into fixed-sized blocks called **frames**
  - Size is power of 2, between 512 bytes and 16 Mbytes
- Divide logical memory into blocks of same size called **pages**
- 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
- Still have Internal fragmentation

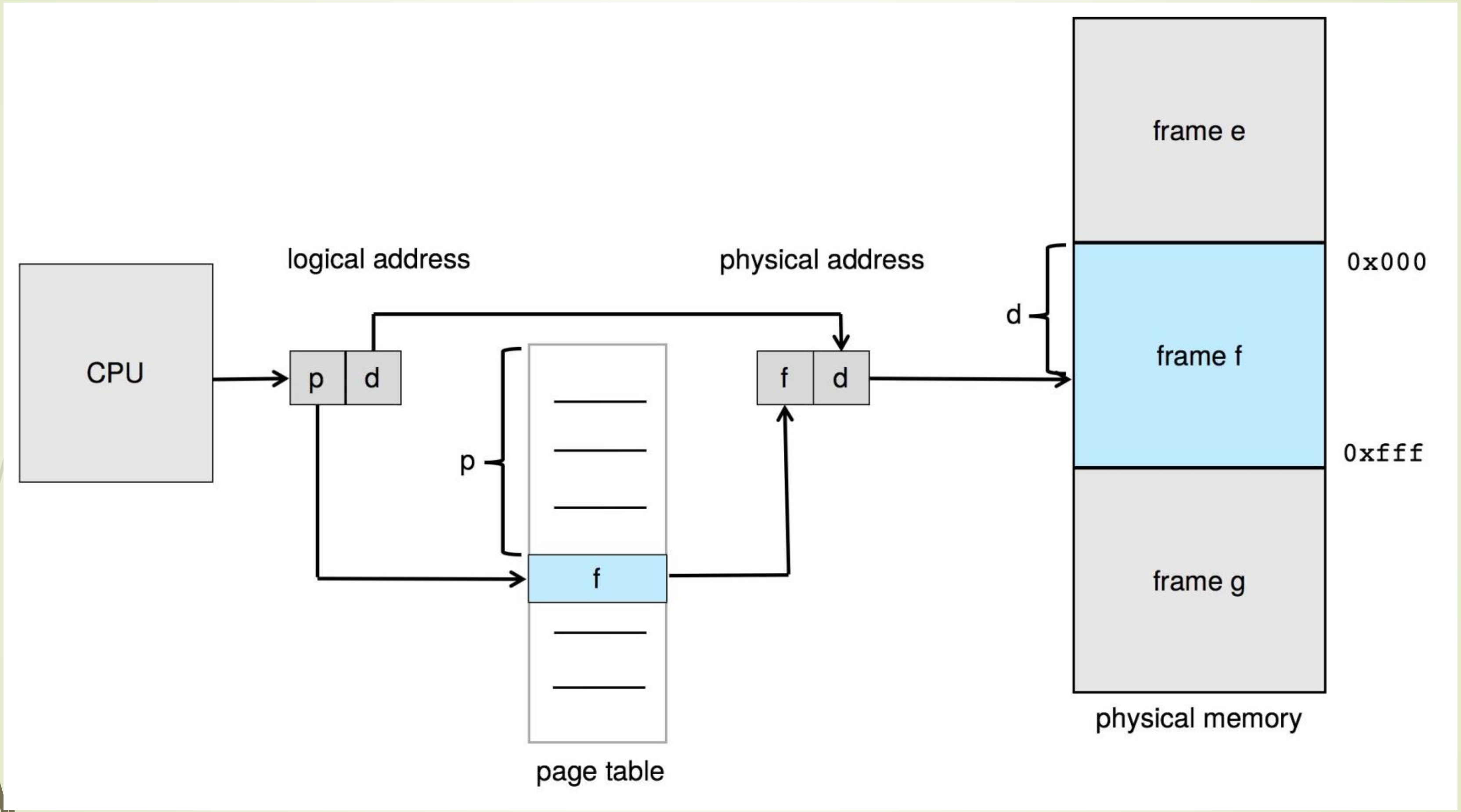
Address Translation Scheme

- 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 offset** (***d***) – combined with base address to define the physical memory address that is sent to the memory unit



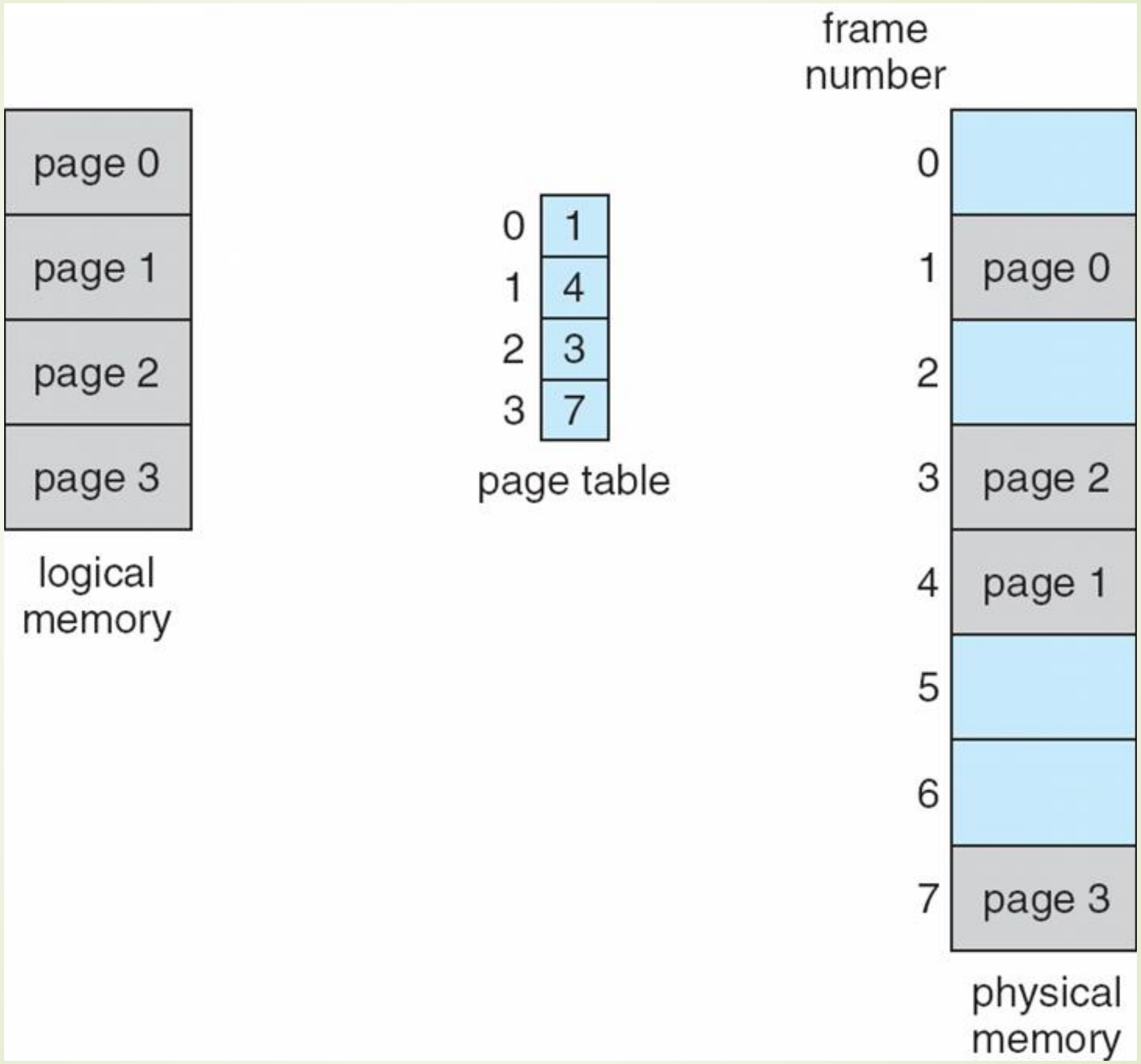
  - For given logical address space 2^m and page size 2^n

Paging Hardware



[Source: Operating Systems Concepts, 10th ed.]

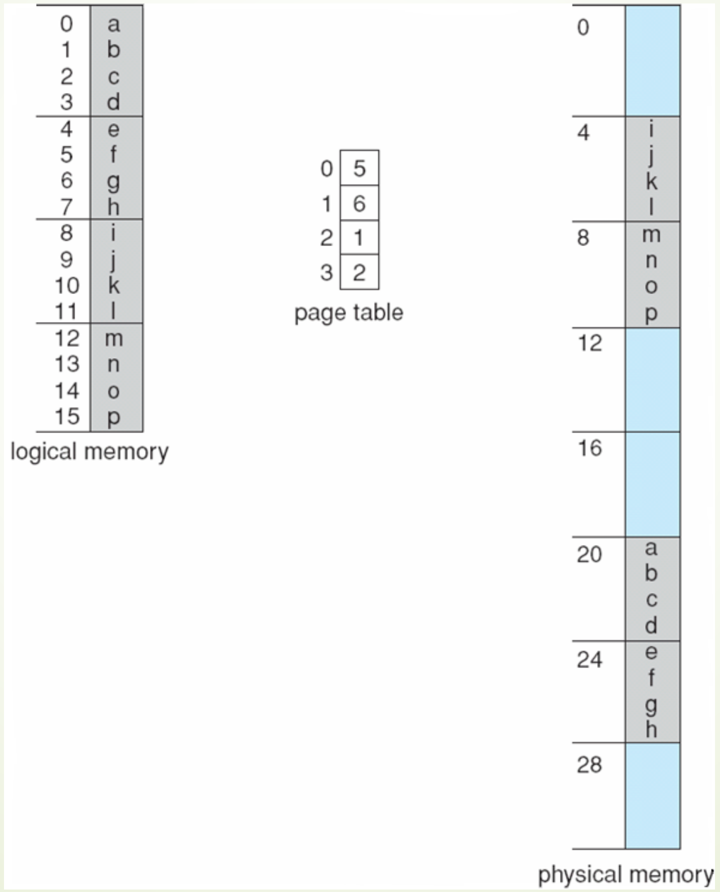
Paging Model of Logical and Physical Memory



[Source: Operating Systems Concepts, 10th ed.]

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)

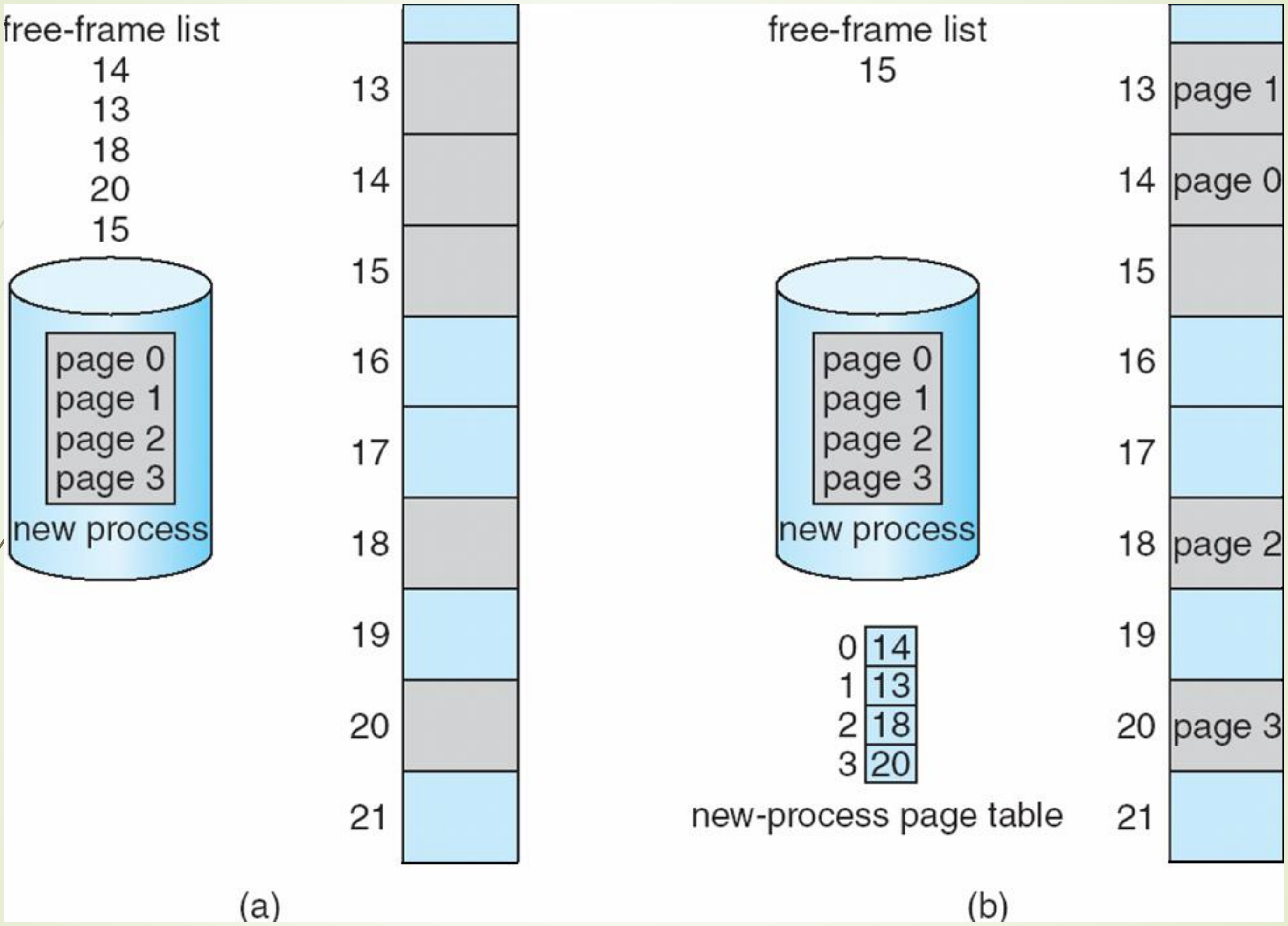


[Source: Operating Systems Concepts, 10th ed.]

Paging -- Calculating internal fragmentation

- Page size = 2,048 bytes
  - For process size of 72,766 bytes
    - 35 pages + 1,086 bytes
    - Internal fragmentation of 2,048 - 1,086 = 962 bytes
- Worst case fragmentation = 1 frame – 1 byte
  - On average fragmentation = 1 / 2 frame size
- So small frame sizes desirable?
  - But each page table entry takes memory to track
- Page sizes growing over time
  - Solaris supports two page sizes – 8 KB and 4 MB

Free Frames



Before allocation

After allocation

[Source: Operating Systems Concepts, 10th ed.]

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
- In this scheme every data/instruction access requires two memory accesses
  - One for the page table and one for the data / instruction
- 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**)

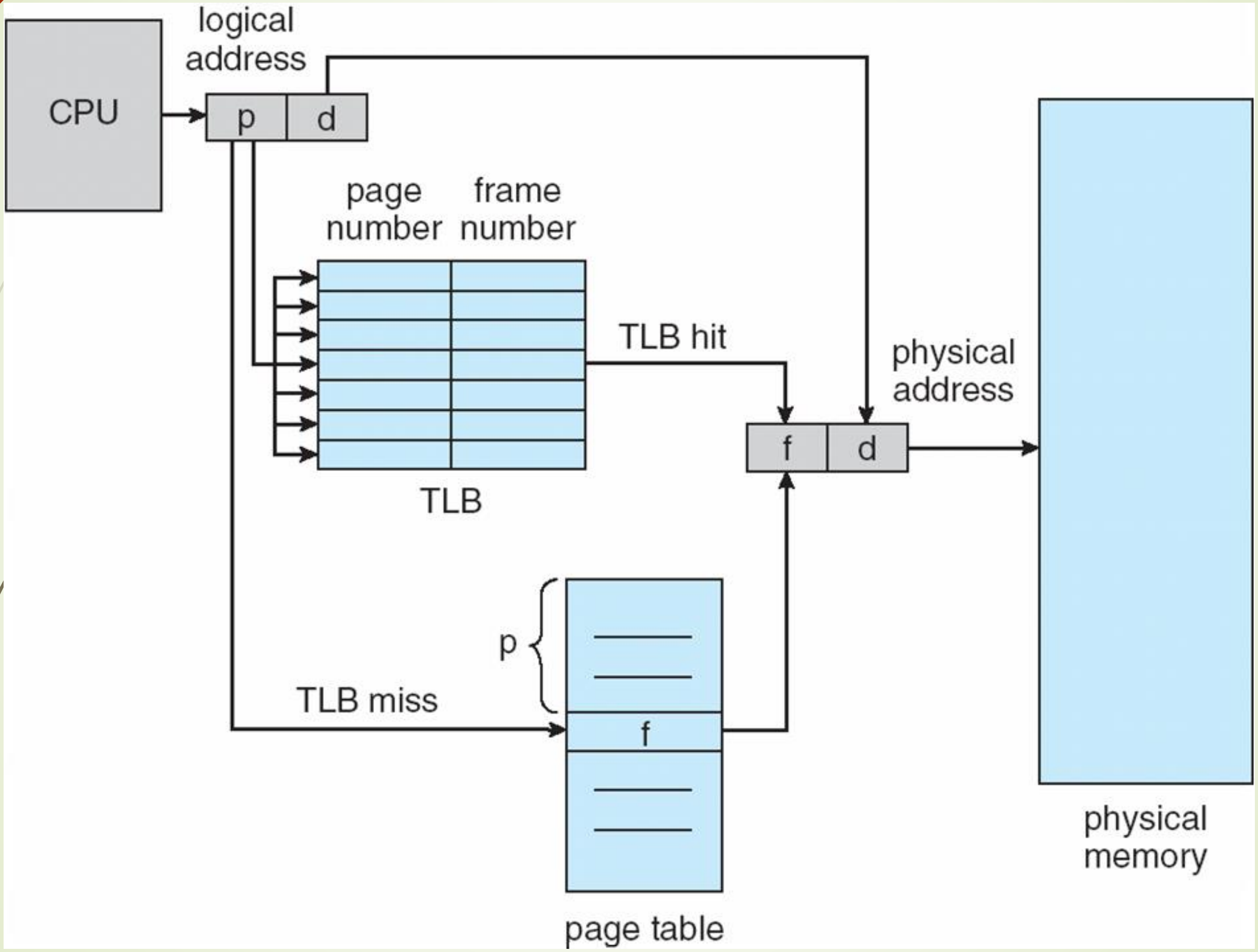
TLB Hardware

- Associative memory – parallel search on key-value pairs

Page #	Frame #

- TLBs typically small (64 to 1,024 entries)
- Address translation (p, d)
  - If p is in associative register, get frame # out
  - Otherwise, for a TLB miss, get frame # from page table in memory, and value is loaded into TLB for faster access next time

Paging Hardware With TLB



[Source: Operating Systems Concepts, 10th ed.]

Effective Access Time

- Hit ratio – percentage of times that a page number is found in the TLB
  - An 80% hit ratio means that we find the desired page number in the TLB 80% of the time
- Suppose that 10 nanoseconds to access memory
  - If we find the desired page in TLB then a mapped-memory access take 10 ns
  - Otherwise we need two memory access so it is 20 ns
- **Effective Access Time** (**EAT**)

  EAT = 0.80 x 10 + 0.20 x 20 = 12 nanoseconds

  implying 20% slowdown in access time
- Consider a more realistic hit ratio of 99%,

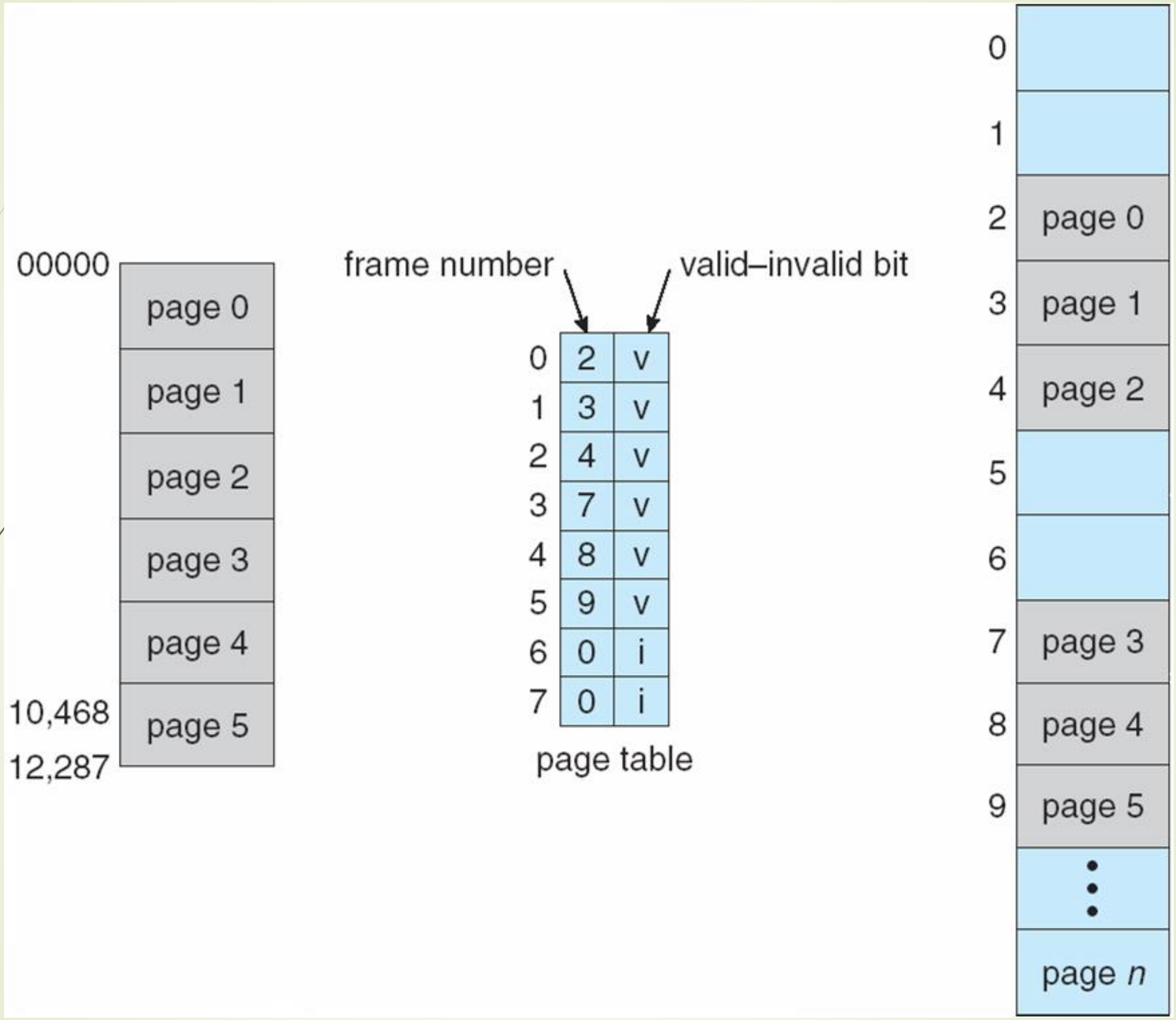
  EAT = 0.99 x 10 + 0.01 x 20 = 10.1ns

  implying only 1% slowdown in access time

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 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

Valid (v) or Invalid (i) Bit in a Page Table

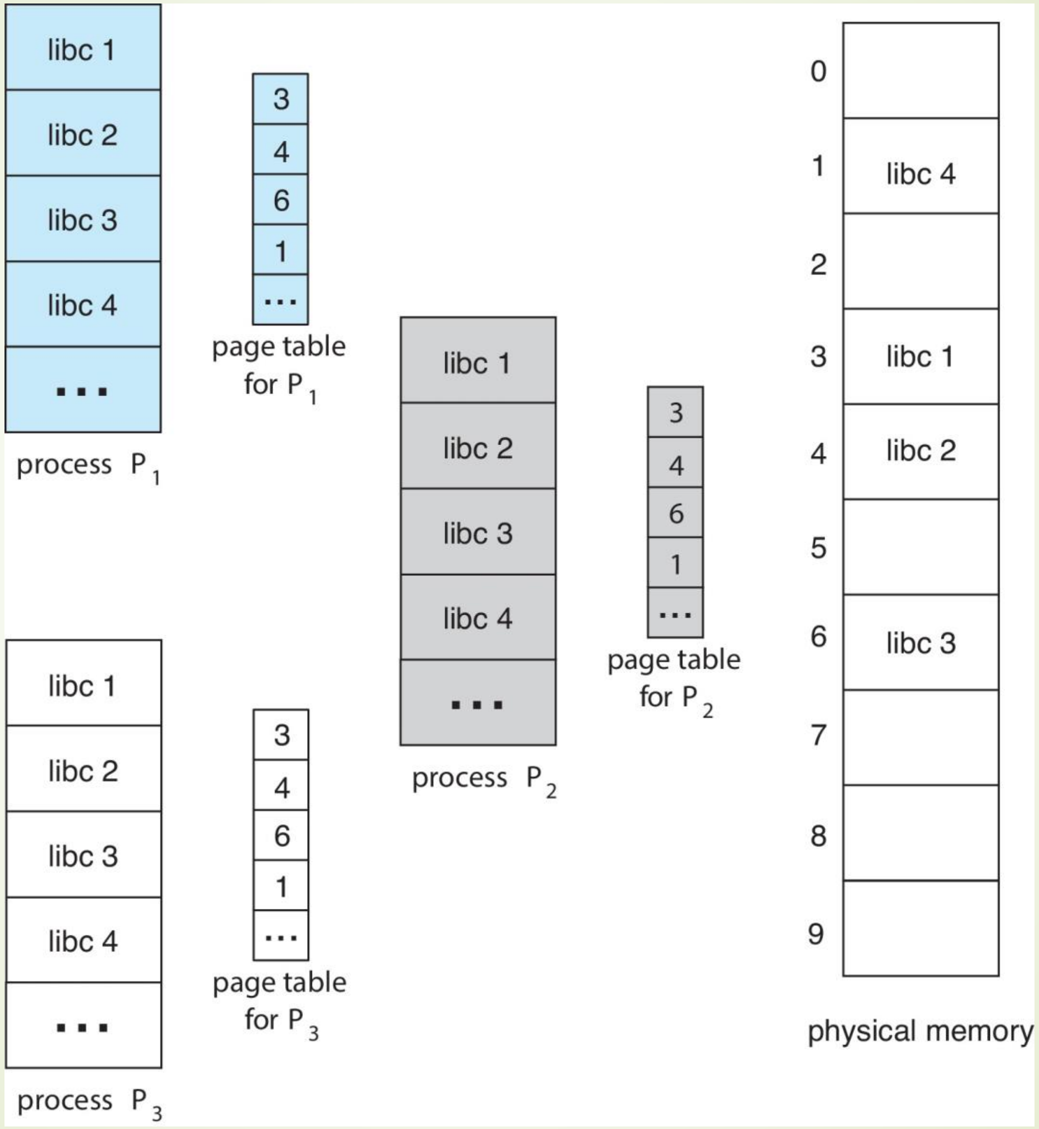


[Source: Operating Systems Concepts, 10th ed.]

Shared Pages

- **Shared code**
  - One copy of read-only (**reentrant**) code shared among processes (i.e., text editors, compilers, window systems)
  - Similar to multiple threads sharing the same process space
  - Also useful for interprocess communication if sharing of read-write pages is allowed
- **Private code and data**
  - Each process keeps a separate copy of the code and data
  - The pages for the private code and data can appear anywhere in the logical address space

Shared Pages Example



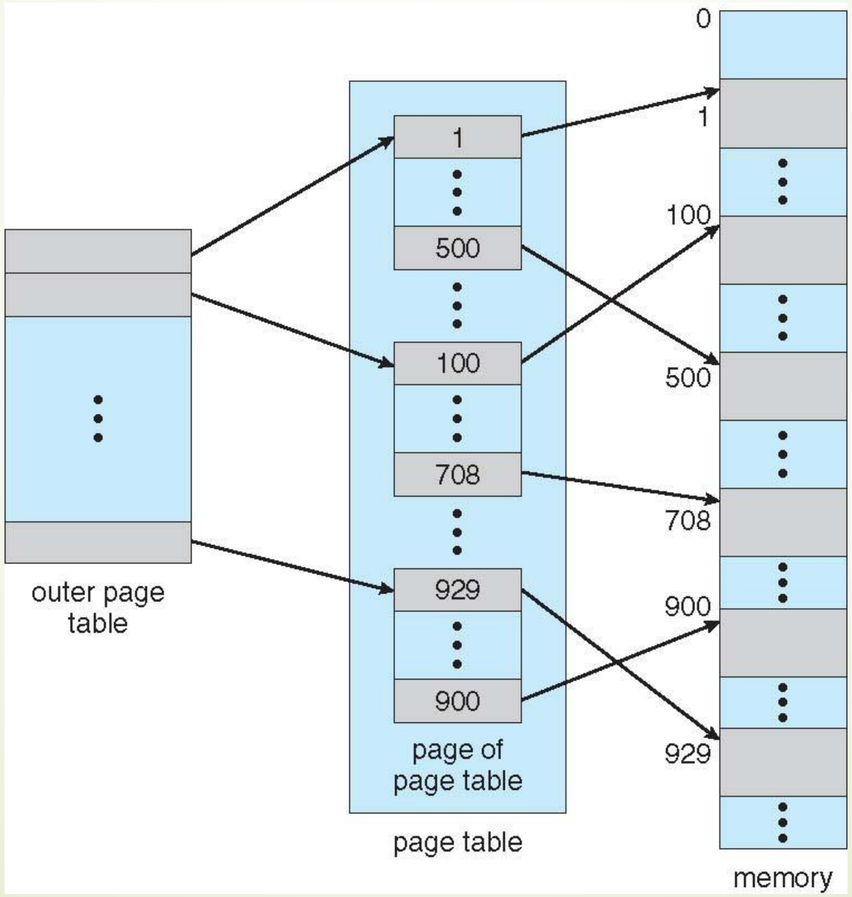
[Source: Operating Systems Concepts, 10th ed.]

Structure of the Page Table

- Memory structures for paging can get huge using straight-forward methods
  - Consider a 32-bit logical address
    - 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
  - One simple solution is to divide the page table into smaller units
    - Hierarchical Paging
    - Hashed Page Tables
    - Inverted Page Tables

Hierarchical Page Tables

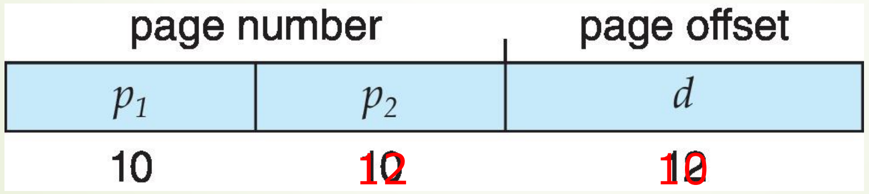
- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
- We then page the page table



[Source: Operating Systems Concepts, 10th ed.]

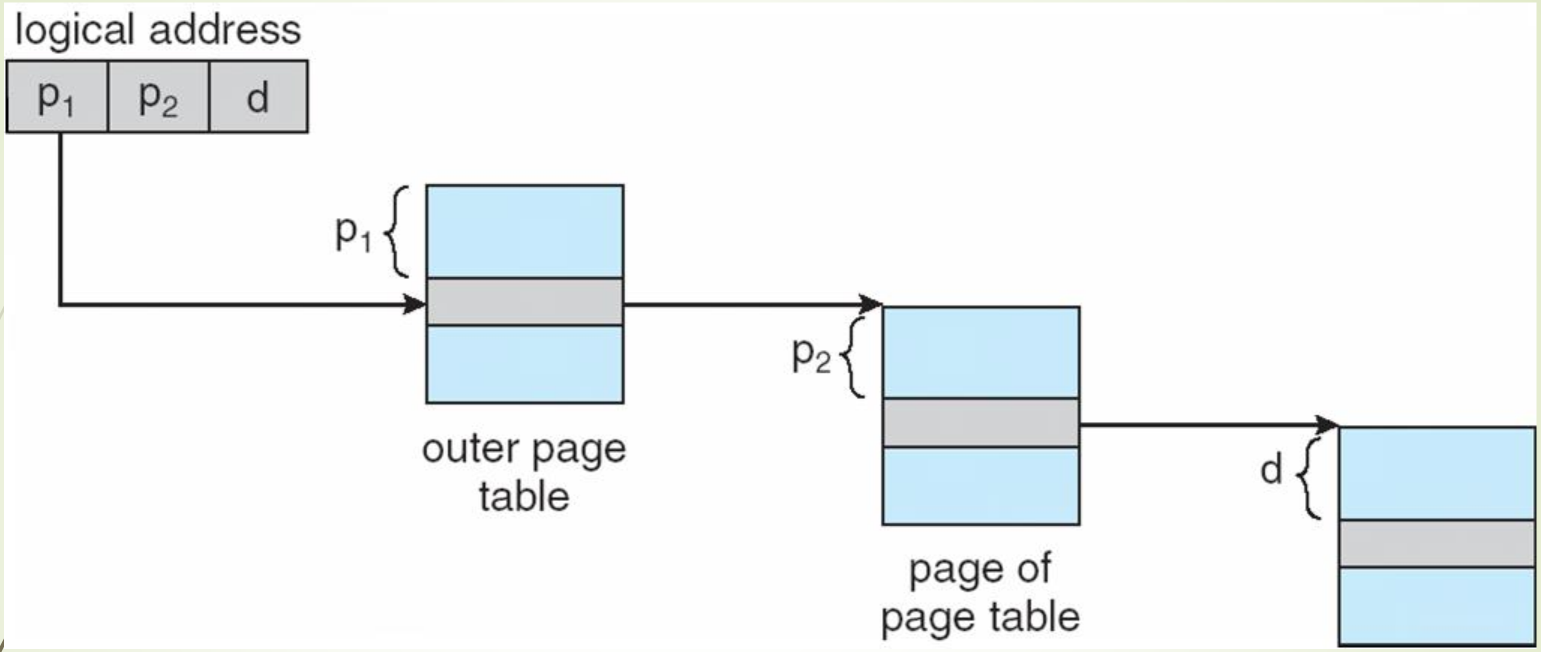
Two-Level Paging Example

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

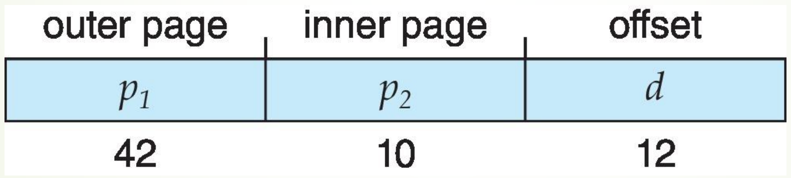
Address-Translation Scheme



[Source: Operating Systems Concepts, 10th ed.]

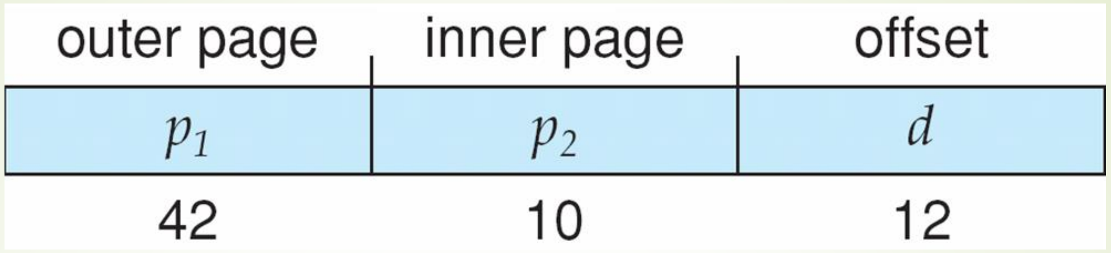
64-bit Logical Address Space

- Even two-level paging scheme not sufficient
- If page size is 4 KB (2^{12})
  - Then page table has 2^{52} entries
  - If two level scheme, inner page tables could be 2^{10} 4-byte entries
  - Address would look like

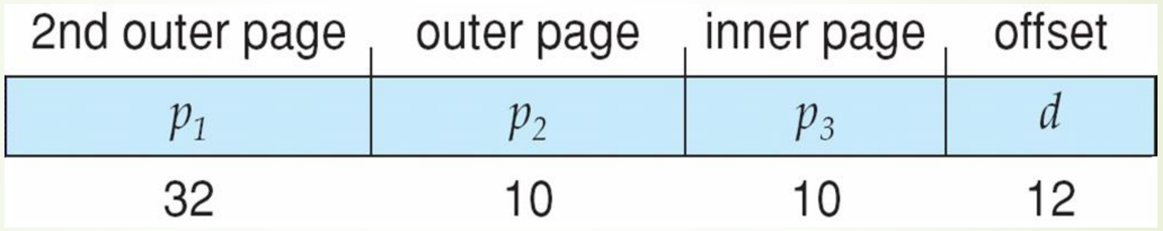


  - Outer page table has 2^{42} entries or 2^{44} bytes
  - One solution is to add a 2nd outer page table
    - But the 2nd outer page table is still 2^{34} bytes in size
    - And possibly 4 memory accesses to get to one physical memory location

Three-level Paging Scheme



outer page	inner page	offset
p_1	p_2	d
42	10	12



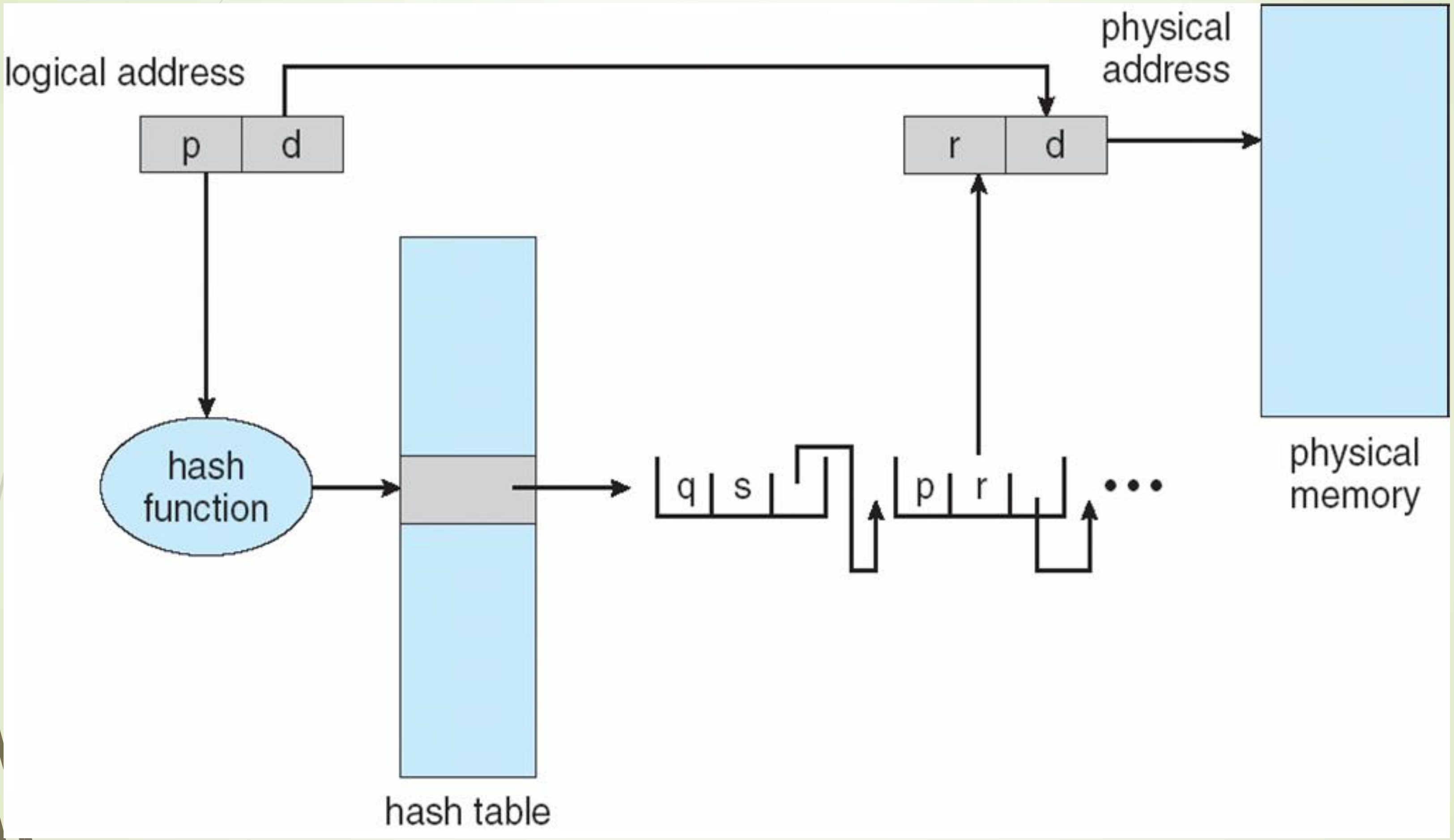
2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12

[Source: Operating Systems Concepts, 10th ed.]

Hashed Page Tables

- The virtual page number is hashed into a page table
  - This page table contains a chain of elements hashing to the same location
  - Each element contains (1) the virtual page number (2) the value of the mapped page frame (3) a pointer to the next element
  - Virtual page numbers are compared in this chain searching for a match
    - If a match is found, the corresponding physical frame is extracted
- Common in address spaces > 32 bits
- 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

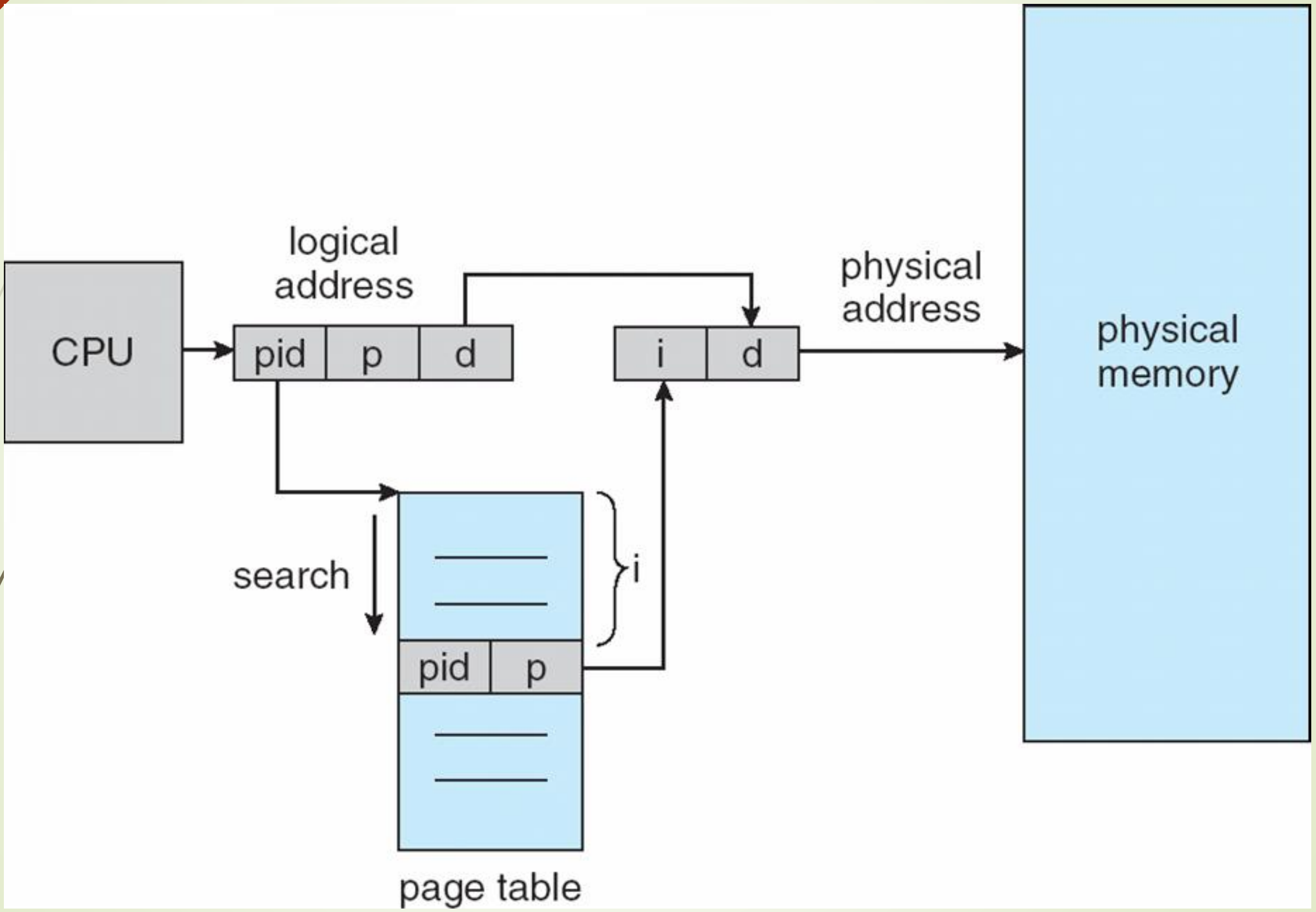


[Source: Operating Systems Concepts, 10th ed.]

Inverted Page Table

- Rather than each process having a page table and keeping track of all possible logical pages, we 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



[Source: Operating Systems Concepts, 10th ed.]

Oracle SPARC Solaris

- Consider modern, 64-bit OS example with tightly integrated HW
  - Goals are efficiency, low overhead
- Based on hashing, but more complex
- Two hash tables
  - One for kernel and one for all user processes
  - Each maps memory addresses from virtual to physical memory
  - Each entry represents a contiguous area of mapped virtual memory
    - More efficient than having a separate hash-table entry for each page
  - Each entry has base address and span (indicating the number of pages the entry represents)

Oracle SPARC Solaris (Cont.)

- TLB holds translation table entries (TTEs) for fast hardware lookups
  - A cache of TTEs reside in a translation storage buffer (TSB)
    - Includes an entry per recently accessed page
- Virtual address reference causes TLB search
  - If miss, hardware walks the in-memory TSB looking for the TTE corresponding to the address
    - If match found, the CPU copies the TSB entry into the TLB and translation completes
    - If no match found, kernel interrupted to search the hash table
      - The kernel then creates a TTE from the appropriate hash table and stores it in the TSB, Interrupt handler returns control to the MMU, which completes the address translation

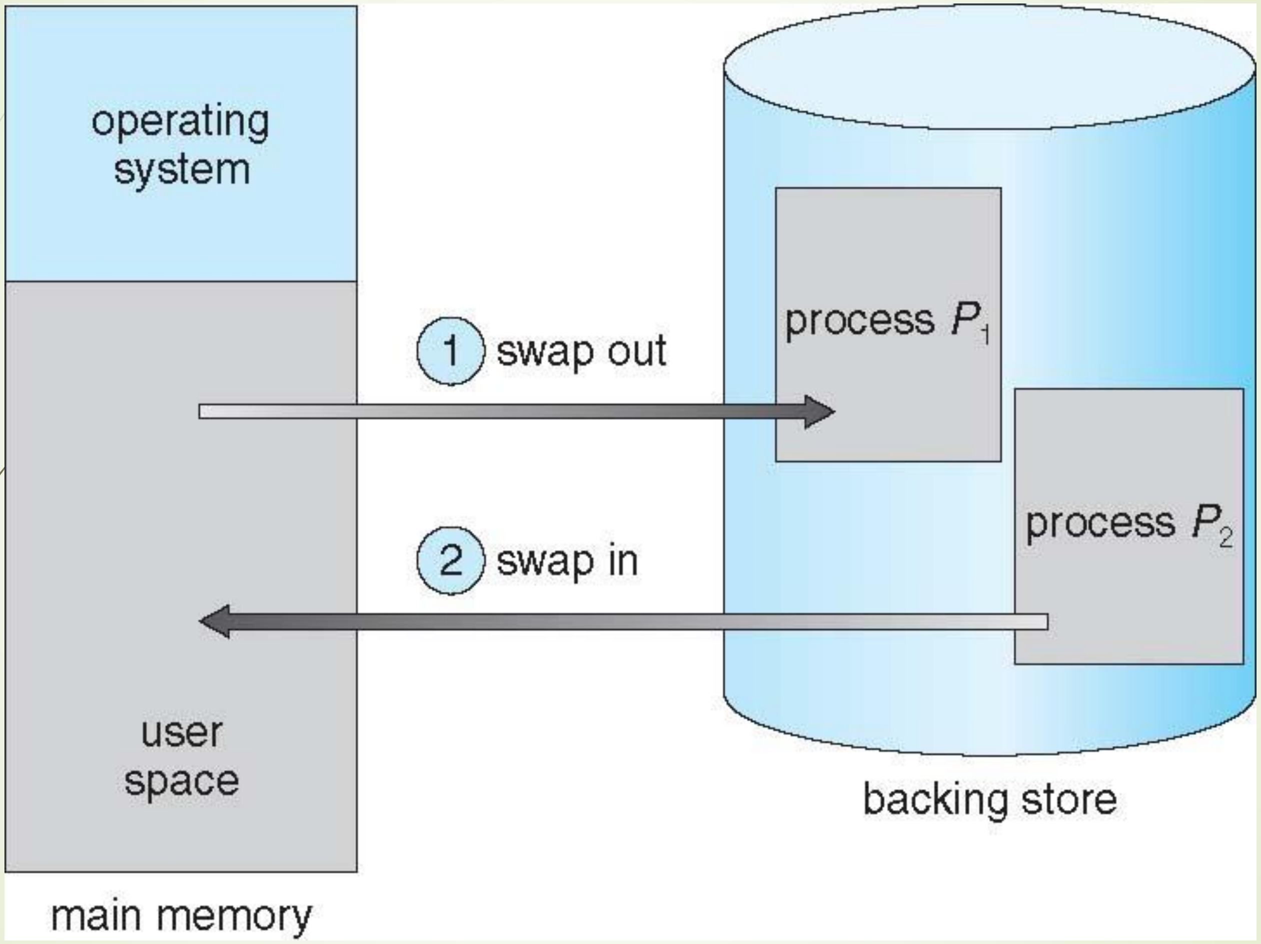
Swapping

- A process can be **swapped** temporarily out of memory to a backing store, and then brought **back** into memory for continued execution
  - Total physical memory space of processes can exceed physical memory
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk

Swapping (Cont.)

- Does the swapped out process need to swap back in to same physical addresses?
  - Depends on address binding method
  - Plus consider pending I/O to / from process memory space
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
  - Swapping normally disabled
  - Started if more than threshold amount of memory allocated
  - Disabled again once memory demand reduced below threshold

Schematic View of Swapping



[Source: Operating Systems Concepts, 10th ed.]

Context Switch Time including Swapping

- If next process to be put on CPU is not in memory, need to swap out a process and swap in target process
  - Context switch time can then be very high
  - 100MB process swapping to hard disk with transfer rate of 50MB/sec
    - Swap out time of 2000 ms
    - Plus swap in of same sized process
    - Total context switch swapping component time of 4000ms (4 seconds)
- Can reduce if reduce size of memory swapped – by knowing how much memory really being used
  - System calls to inform OS of memory use via `request_memory()` and `release_memory()`

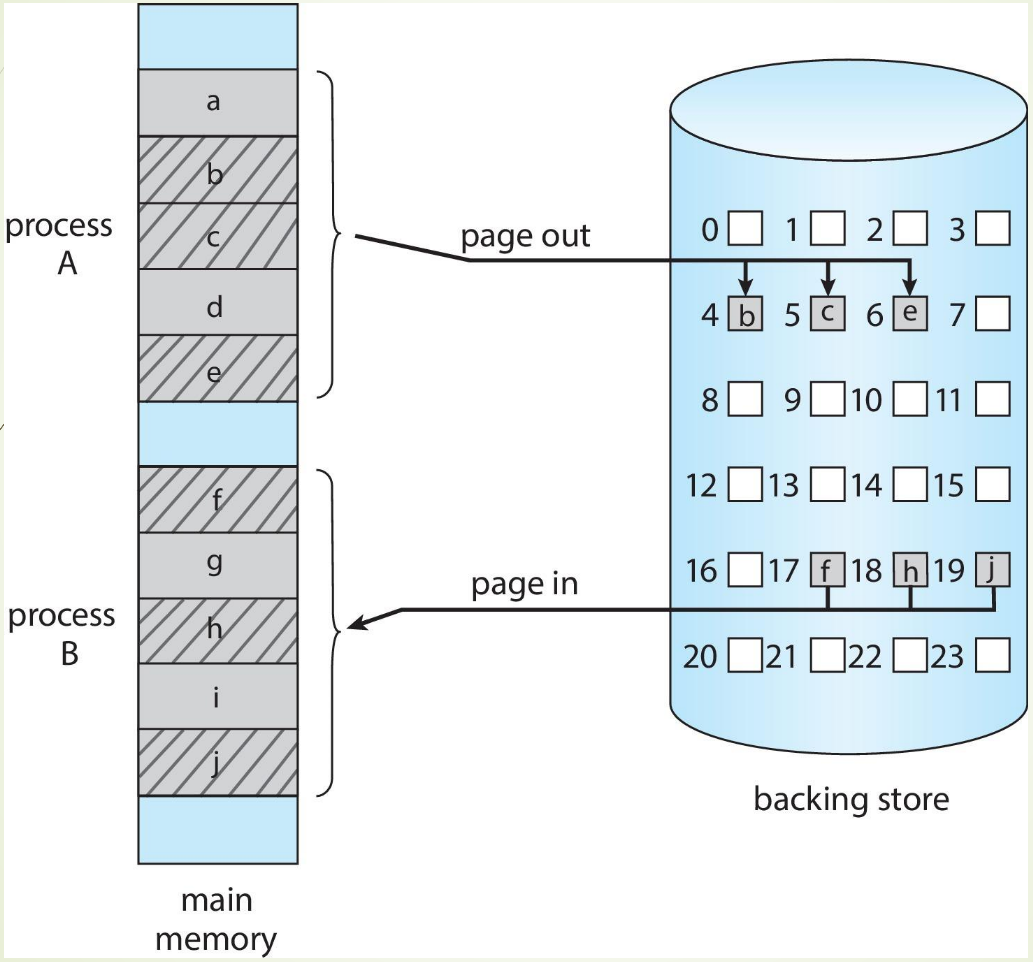
Context Switch Time and Swapping (Cont.)

- Other constraints as well on swapping
  - Pending I/O – can't swap out as I/O would occur to wrong process
  - Or always transfer I/O to kernel space, then to I/O device
    - Known as **double buffering**, adds overhead
- Standard swapping not used in modern OS
  - But modified version common
    - Swap only when free memory extremely low

Swapping on Mobile Systems

- Not typically supported
  - Flash memory based
    - Small amount of space
    - Limited number of write cycles
    - Poor throughput between flash memory and CPU on mobile platform
- Instead use other methods to free memory if low
  - iOS ***asks*** apps to voluntarily relinquish allocated memory
    - Read-only data thrown out and reloaded from flash if needed
    - Failure to free can result in termination
  - Android terminates apps if low free memory, but first writes **application state** to flash for fast restart
  - Both OSes support paging as discussed below

Swapping with Paging



[Source: Operating Systems Concepts, 10th ed.]

Example: The Intel 32 and 64-bit Architectures

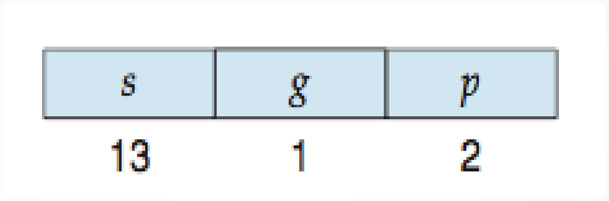
- Dominant industry chips
- Pentium CPUs are 32-bit and called IA-32 architecture
- Current Intel CPUs are 64-bit and called IA-64 architecture
- Many variations in the chips, cover the main ideas here

Example: The Intel IA-32 Architecture

- Supports both segmentation and segmentation with paging
  - Each segment can be 4 GB
  - Up to 16 K segments per process
  - Divided into two partitions
    - First partition of up to 8K segments are private to process (kept in **local descriptor table** (**LDT**))
    - Second partition of up to 8K segments shared among all processes (kept in **global descriptor table** (**GDT**))

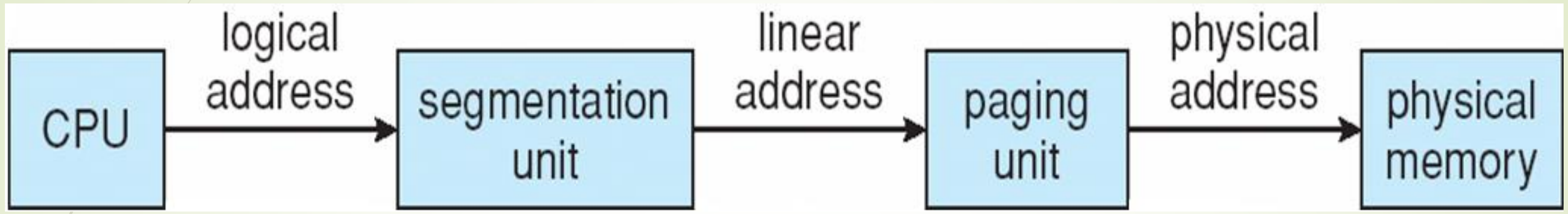
Example: The Intel IA-32 Architecture (Cont.)

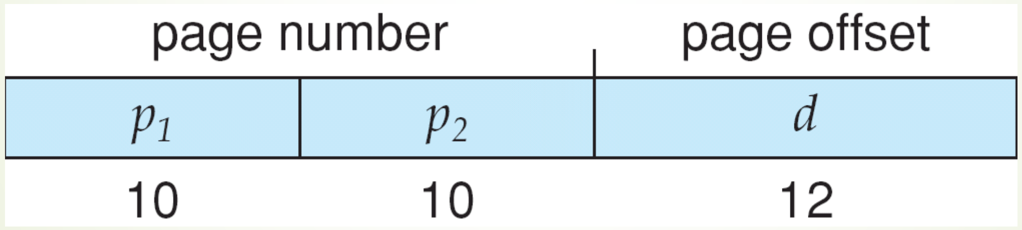
- CPU generates logical address
  - Selector given to segmentation unit
    - Which produces linear addresses



  - Linear address given to paging unit
    - Which generates physical address in main memory
    - Paging units form equivalent of MMU
    - Pages sizes can be 4 KB or 4 MB

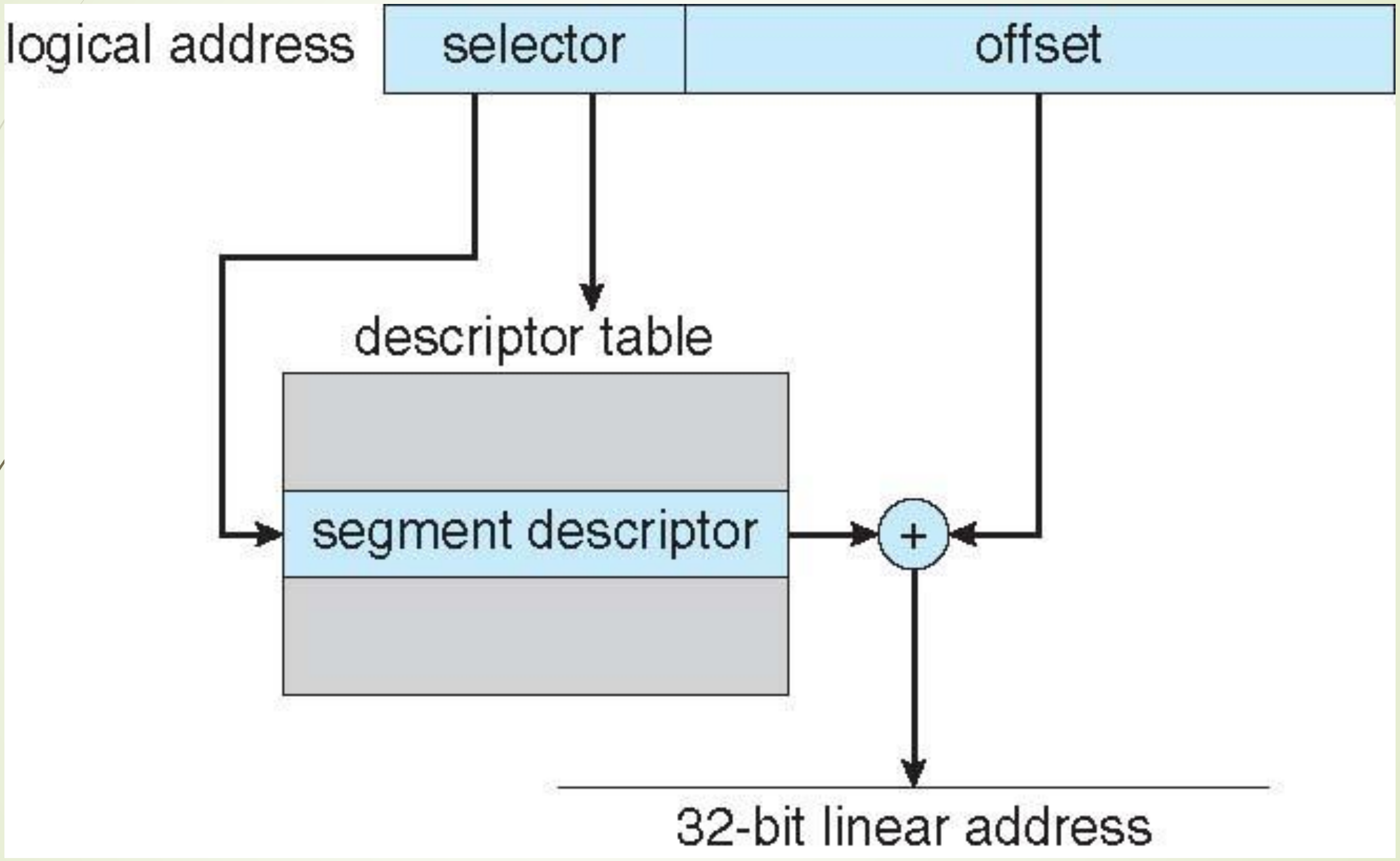
Logical to Physical Address Translation in IA-32





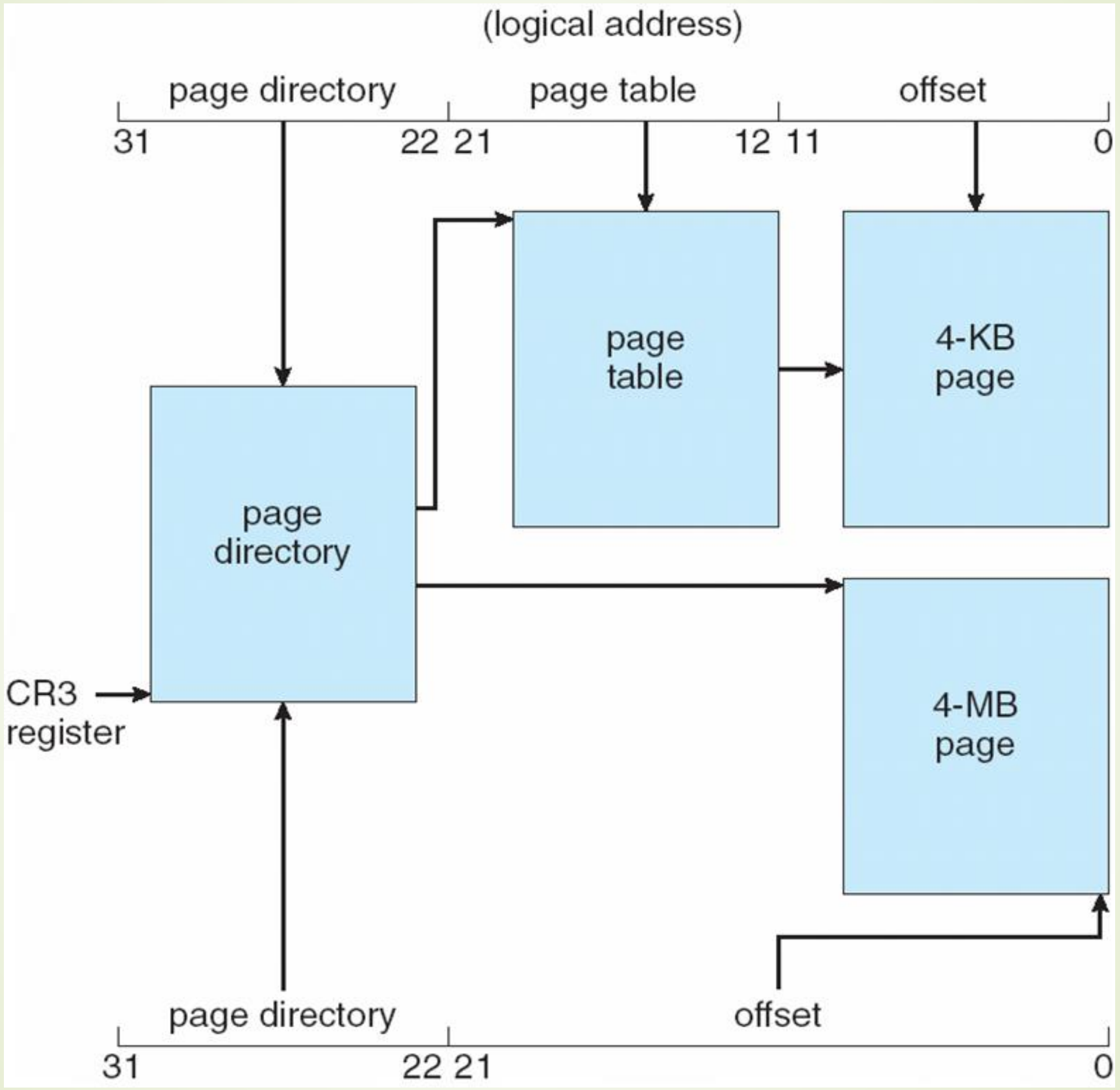
[Source: Operating Systems Concepts, 10th ed.]

Intel IA-32 Segmentation



[Source: Operating Systems Concepts, 10th ed.]

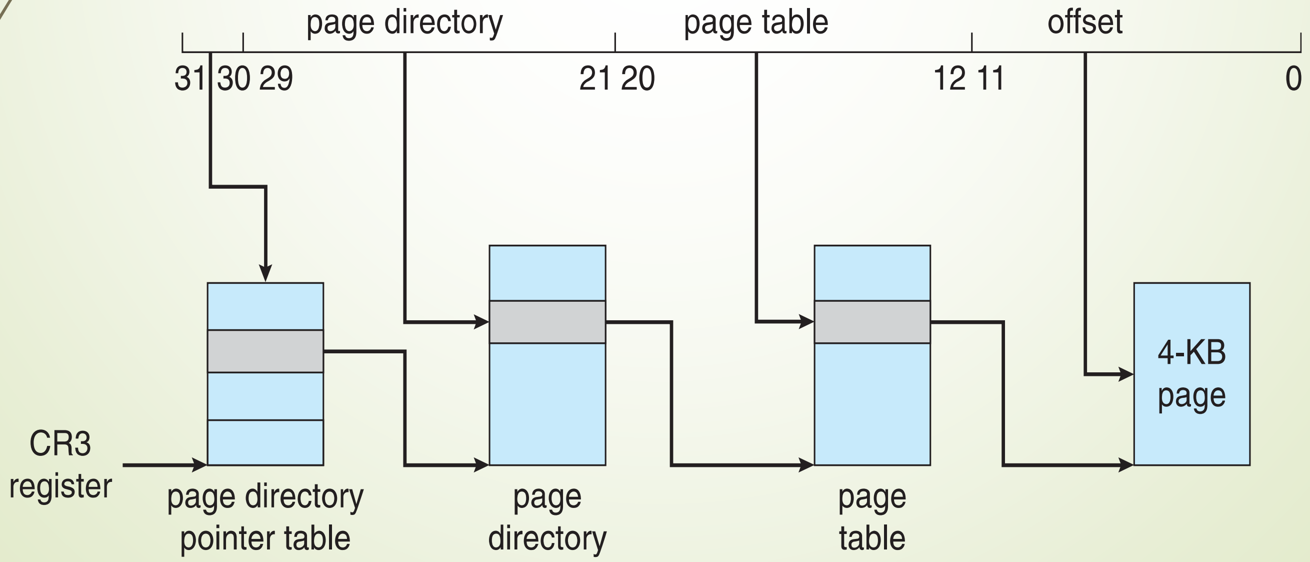
Intel IA-32 Paging Architecture



[Source: Operating Systems Concepts, 10th ed.]

Intel IA-32 Page Address Extensions

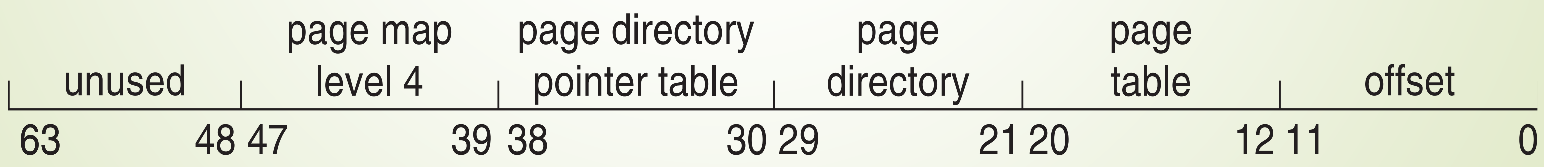
- 32-bit address limits led Intel to create **page address extension** (**PAE**), allowing 32-bit apps access to more than 4GB memory space
  - Paging went to a 3-level scheme
  - Top two bits refer to a **page directory pointer table**
  - Page-directory and page-table entries moved to 64-bits
  - Net effect is increasing address space to 36 bits – 64GB of physical memory



[Source: Operating Systems Concepts, 10th ed.]

Intel x86-64

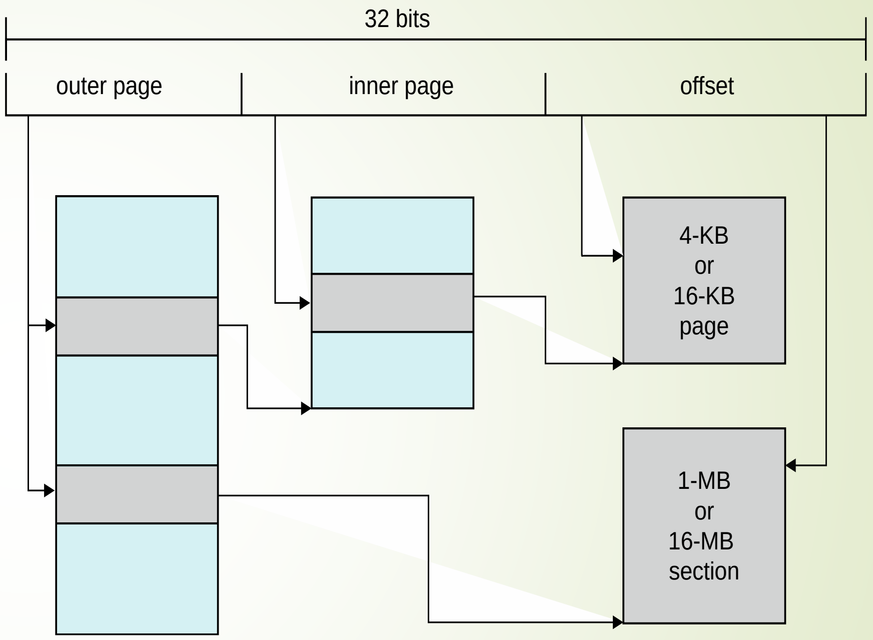
- Current generation Intel x86 architecture
- 64 bits is ginormous (> 16 exabytes)
- In practice only implement 48 bit addressing
  - Page sizes of 4 KB, 2 MB, 1 GB
  - Four levels of paging hierarchy
- Can also use PAE so virtual addresses are 48 bits and physical addresses are 52 bits



[Source: Operating Systems Concepts, 10th ed.]

Example: ARM Architecture

- Dominant mobile platform chip (Apple iOS and Google Android devices for example)
  - Modern, energy efficient, 32-bit CPU
  - 4 KB and 16 KB pages
  - 1 MB and 16 MB pages (termed **sections**)
- One-level paging for sections, two-level for smaller pages
- Two levels of TLBs
  - Outer level has two micro TLBs (one data, one instruction)
  - Inner is single main TLB
  - First inner is checked, on miss outers are checked, and on miss page table walk performed by CPU



[Source: Operating Systems Concepts, 10th ed.]

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 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 -> 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 program runs faster

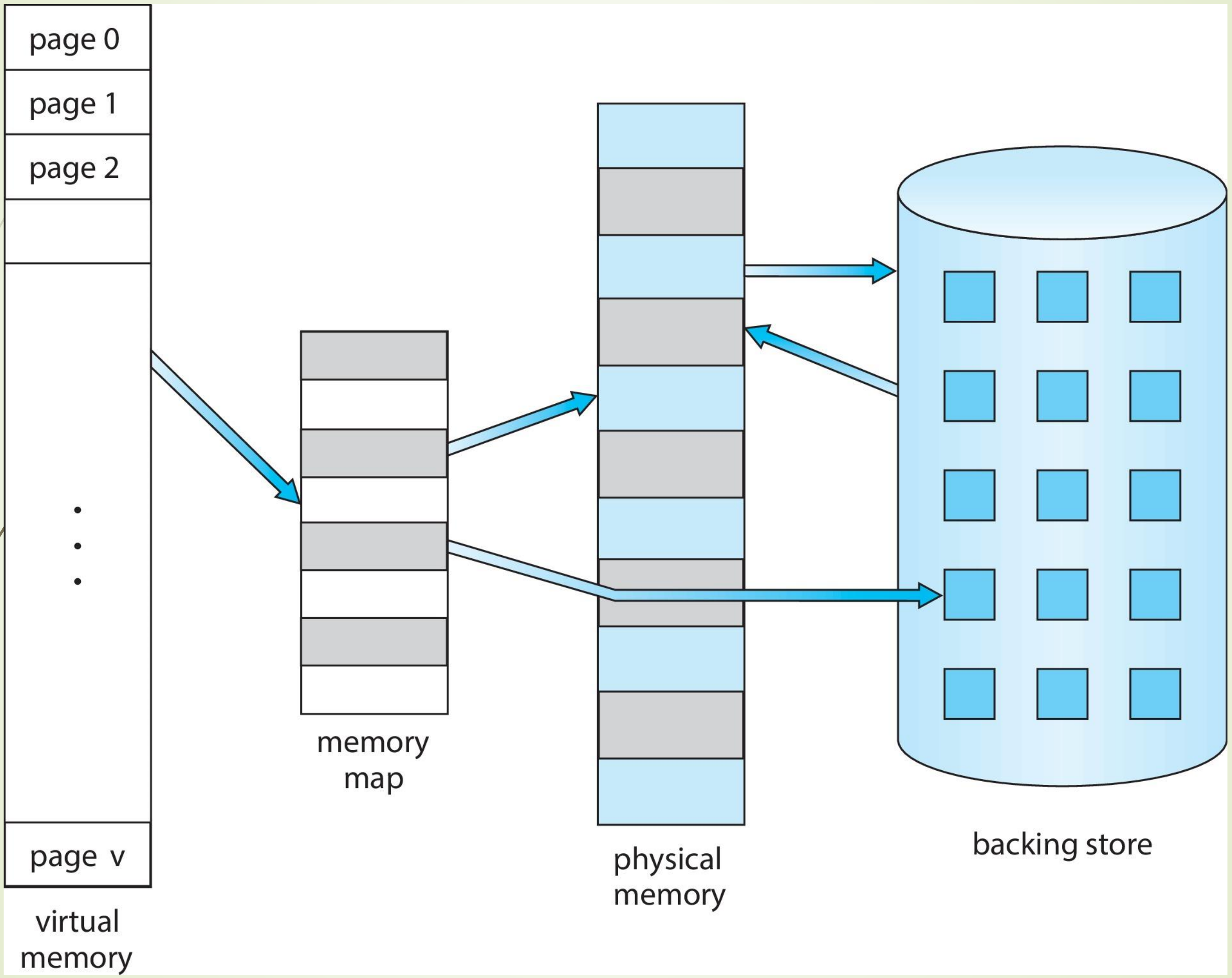
Virtual memory

- **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
  - More programs running concurrently
  - Allows for more efficient process creation
  - 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

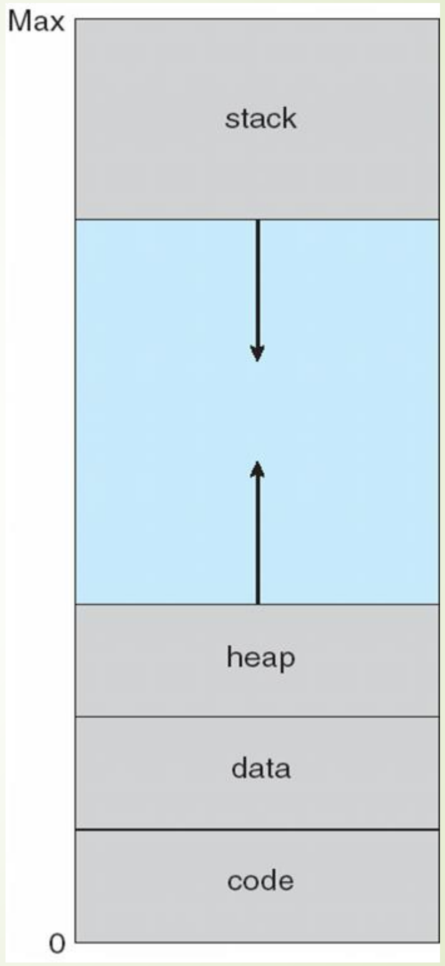
Virtual Memory That is Larger Than Physical Memory



[Source: Operating Systems Concepts, 10th ed.]

Virtual-address Space

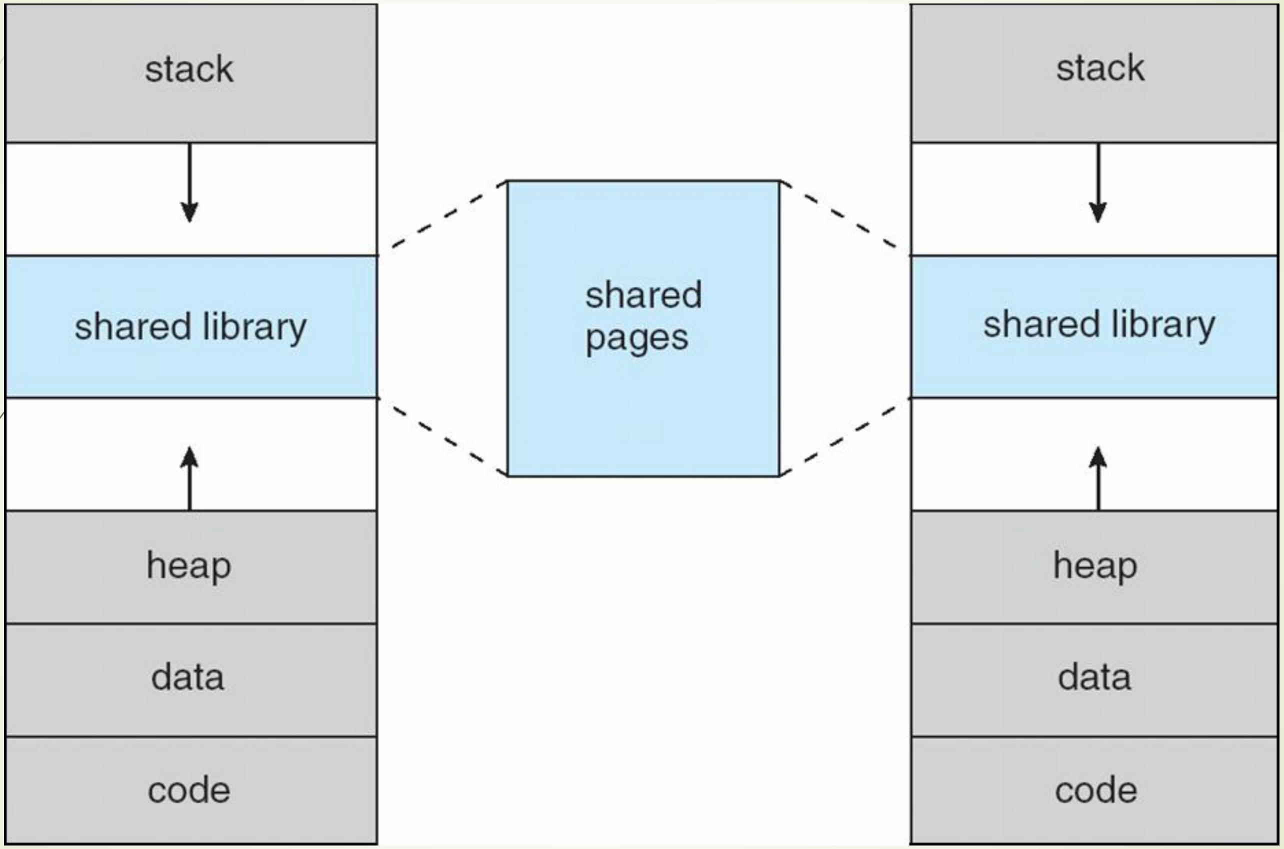
- Usually design logical address space for stack to grow “down” from Max address, while heap grows “up”
  - Maximizes address space use
  - Unused address space between the two is hole
    - No physical memory needed until heap or stack grows to a given new page



[Source: Operating Systems Concepts, 10th ed.]

- Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc.
- System libraries shared via mapping into virtual address space
- Shared memory by mapping pages read-write into virtual address space
- Pages can be shared during `fork()`, speeding process creation

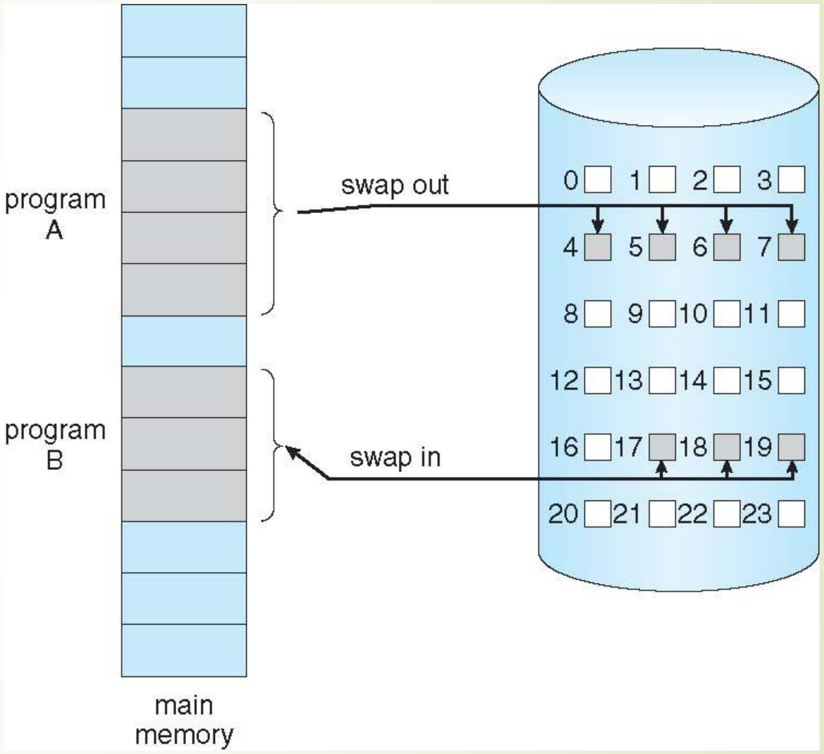
Shared Library Using Virtual Memory



[Source: Operating Systems Concepts, 10th ed.]

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)



[Source: Operating Systems Concepts, 10th ed.]

Demand Paging

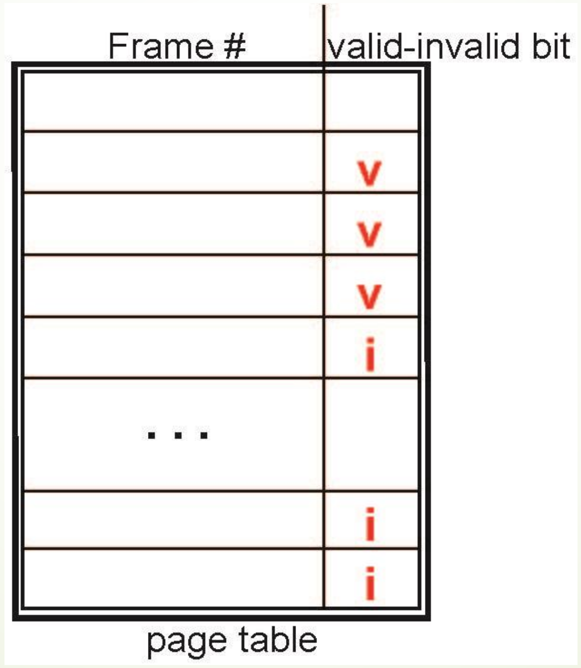
- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory
- **Lazy swapper** – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**

Basic Concepts

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

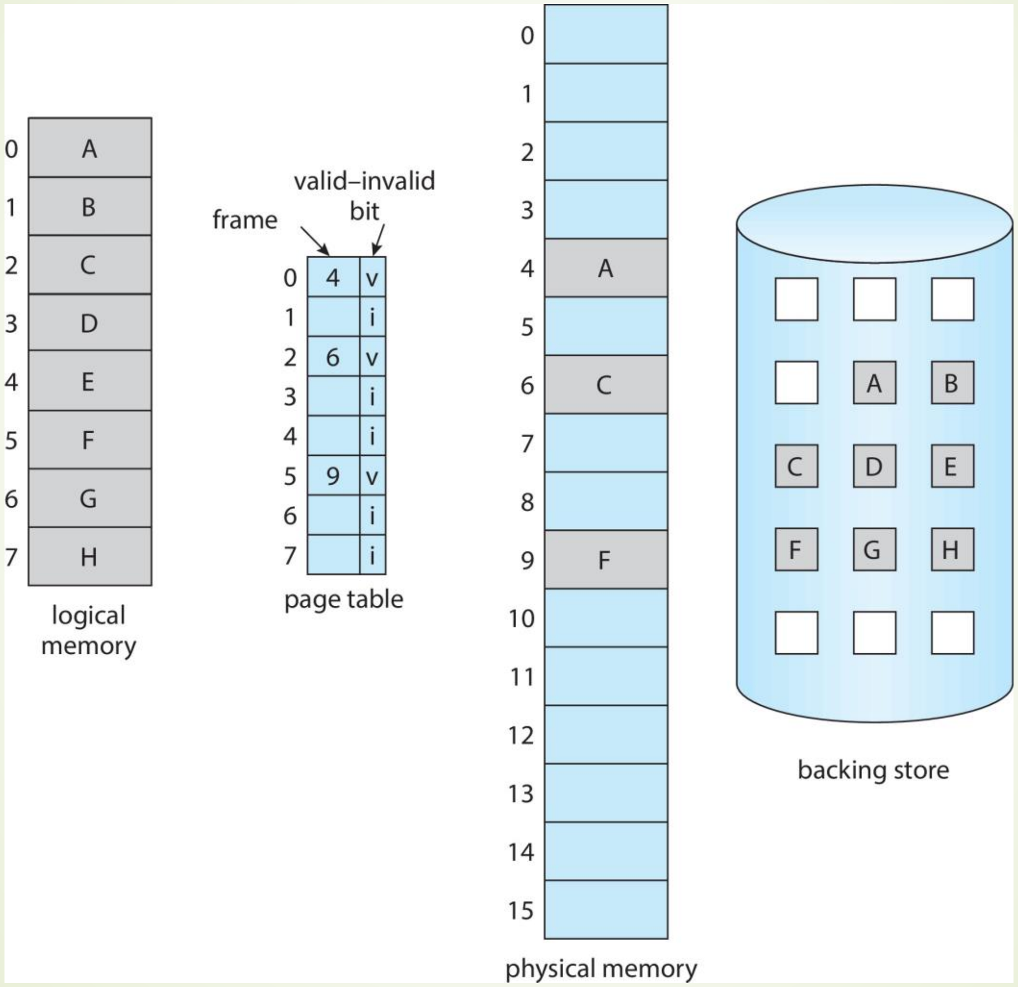
- With each page table entry a valid–invalid bit is associated (**v** ⇒ in-memory – **memory resident**, **i** ⇒ not-in-memory)
  - Initially valid–invalid bit is set to **i** on all entries
  - Example of a page table snapshot:



[Source: Operating Systems Concepts, 10th ed.]

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

Page Table When Some Pages are not in Main Memory

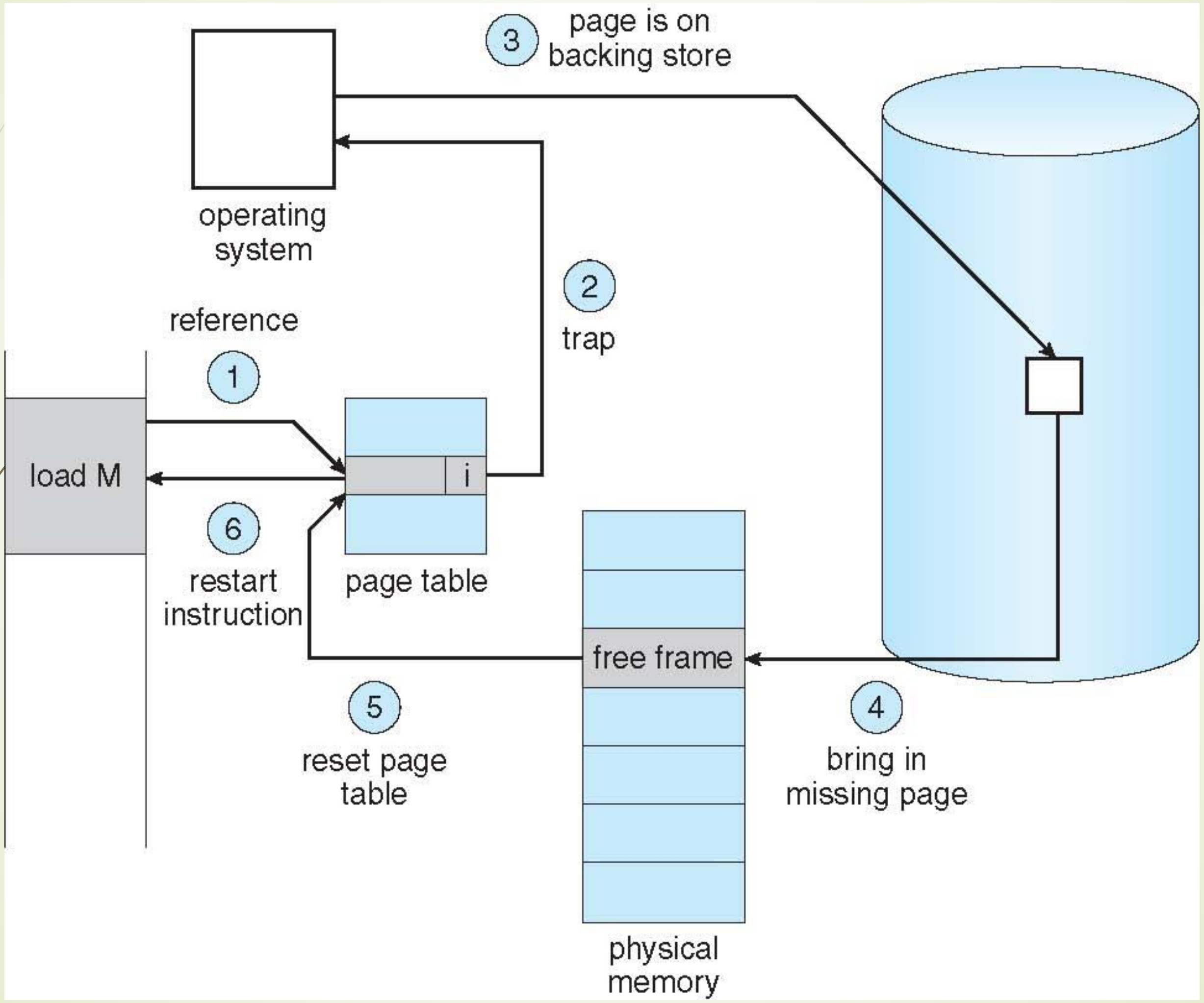


[Source: Operating Systems Concepts, 10th ed.]

Steps in Handling Page Fault

1. If there is a reference to a page, first reference to that page will trap to OS
   - Page fault
2. OS 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.)



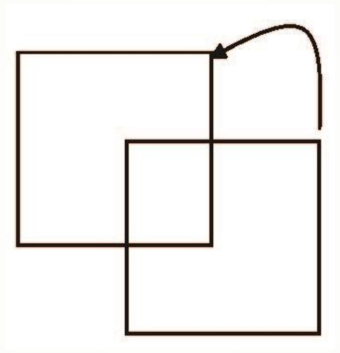
[Source: Operating Systems Concepts, 10th ed.]

Aspects of Demand Paging

- Extreme case – start process with *no* pages in memory
  - OS sets program counter 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 2 numbers from memory and stores result back to memory
  - Pain decreased because of **locality of reference**
- Hardware support needed for demand paging
  - Page table with valid / invalid bit
  - Secondary memory (swap device with **swap space**)
  - Instruction restart

Instruction Restart

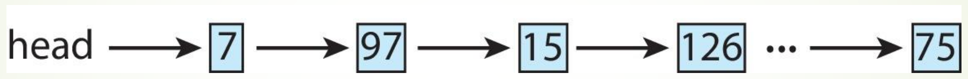
- Consider an instruction that could access several different locations
  - Block move



  - Auto increment/decrement location
  - Restart the whole operation?
    - What if source and destination overlap?

Free-Frame List

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



- OS typically allocates free frames using a technique known as **zero-fill-on-demand** -- the content of the frames zeroed-out before being allocated
- When a system starts up, all available memory is placed on the free-frame list

Stages in Demand Paging – Worst Case

1. Trap to the OS
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:
    a) Wait in a queue for this device until the read request is serviced
    b) Wait for the device seek and/or latency time
    c) Begin the transfer of the page to a free frame

Stages in Demand Paging (Cont.)

6. While waiting, allocate the CPU to some other user
7. Receive an interrupt from the disk I/O subsystem (I/O completed)
8. Save the registers and process state for the other user
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

- 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

  $+ \; p$ x (page fault overhead

  + swap page out

  + swap page in)

Demand Paging Example

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

  = (1 – p x 200 + p x 8,000,000

  = 200 + p x 7,999,800
- If one access out of 1,000 causes a page fault, then

  EAT = 8.2 microseconds

  This is a slowdown by a factor of 40!!
- If want performance degradation < 10 percent
  - 220 > 200 + 7,999,800 x p
    20 > 7,999,800 x p
  - p < .0000025
  - < one page fault in every 400,000 memory accesses

Demand Paging Optimizations

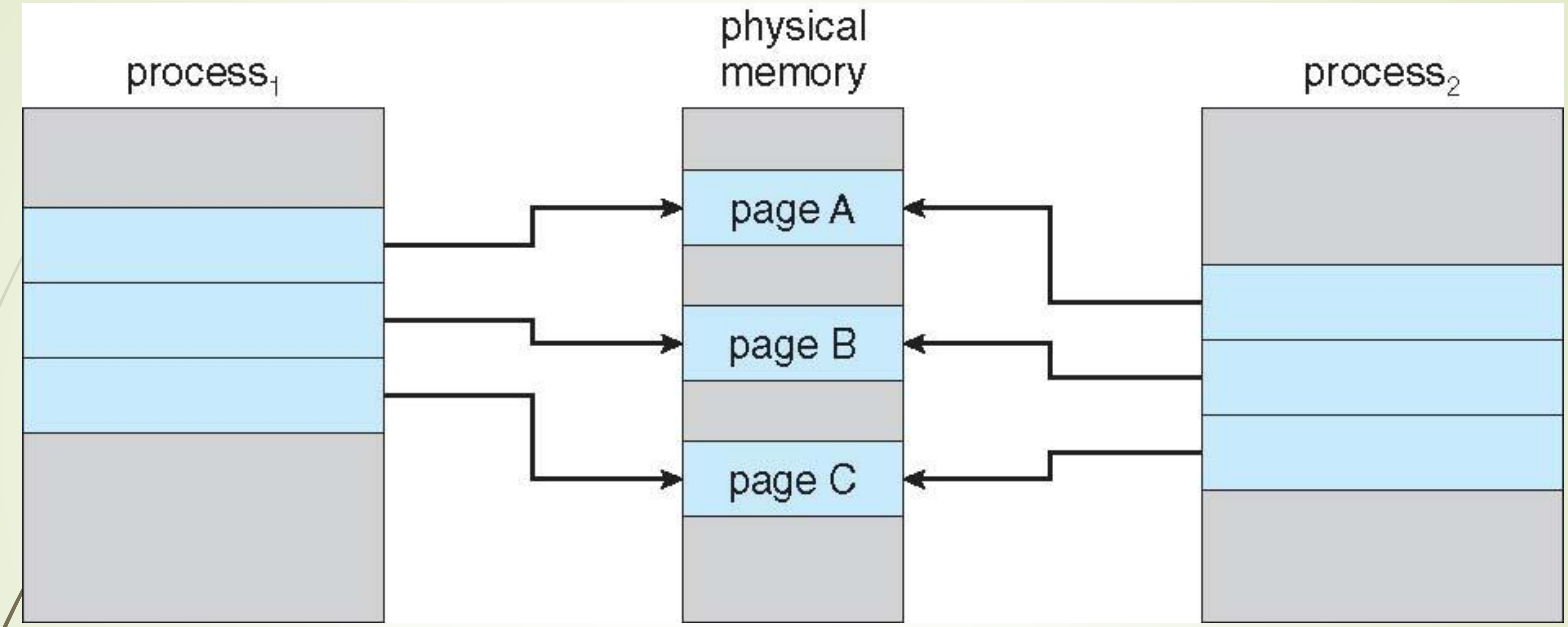
- Swap space I/O faster than file system I/O even if on the same device
  - Swap allocated in larger chunks, less management needed than file system
- Copy entire process image to swap space at process load time
  - Then page in and out of swap space
  - Used in older BSD Unix

- 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

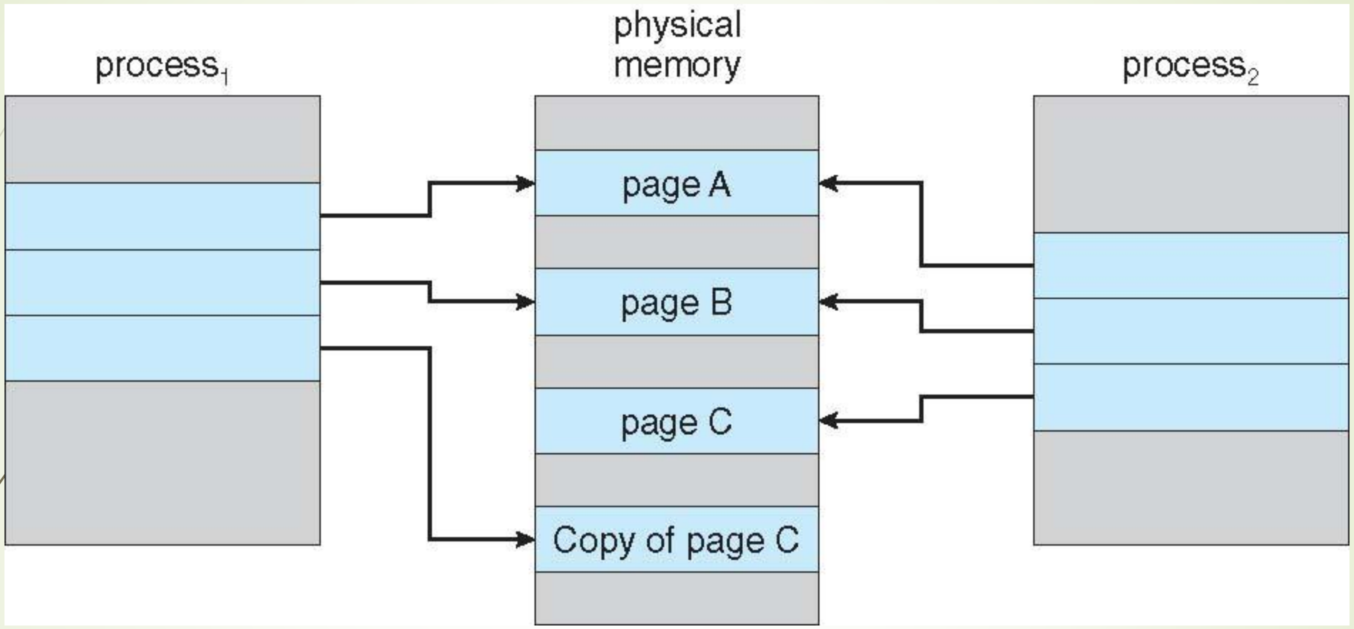
- **Copy-on-Write** (COW) allows both parent and child processes to initially ***share*** the same pages in memory
  - If either process modifies a shared page, only then is the page copied
- COW allows more efficient process creation as only modified pages are copied

Before Process 1 Modifies Page C



[Source: Operating Systems Concepts, 10th ed.]

After Process 1 Modifies Page C



[Source: Operating Systems Concepts, 10th ed.]

- 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

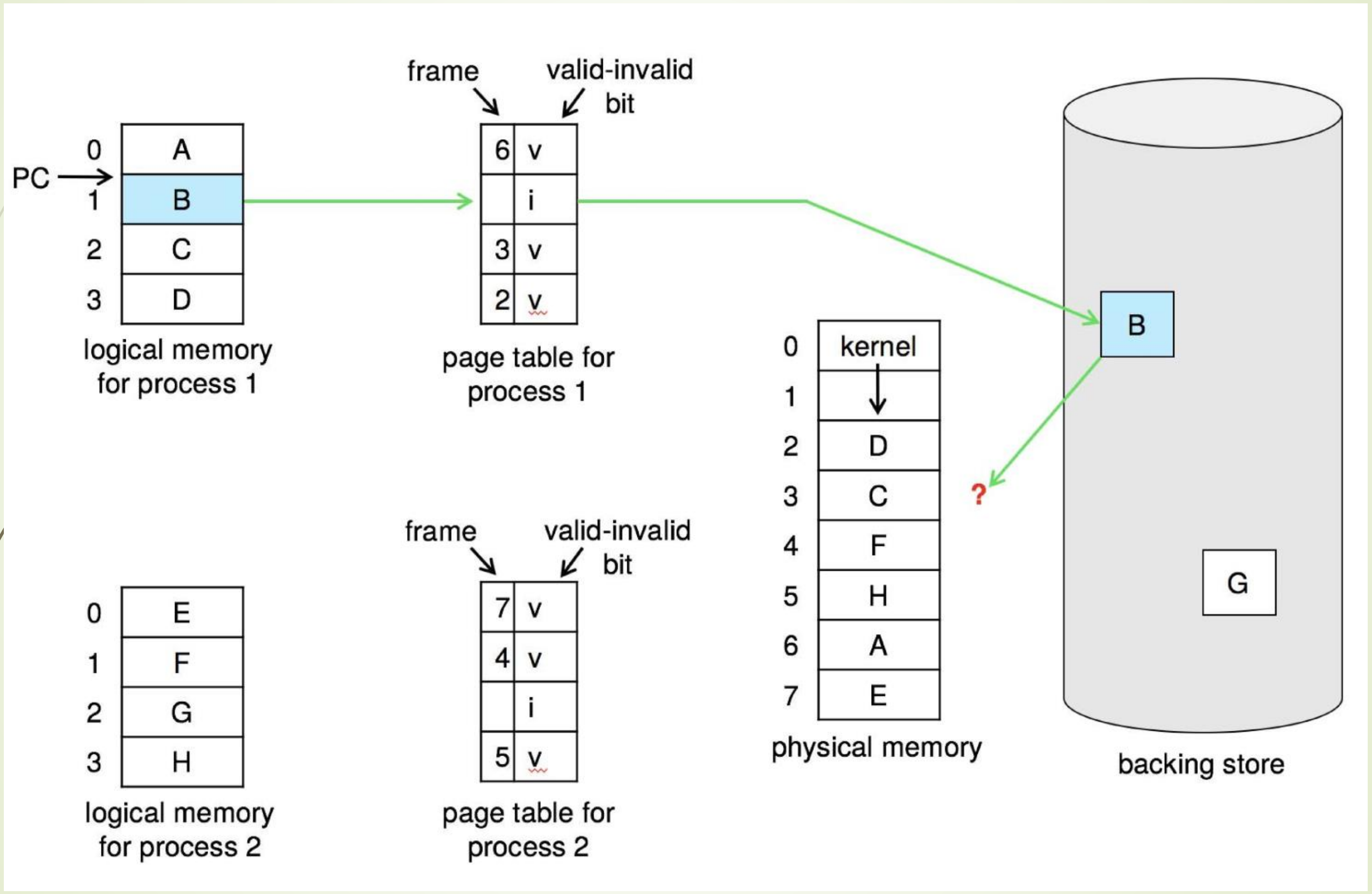
What Happens if There is no Free Frame?

- Pages used up by process
  - 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



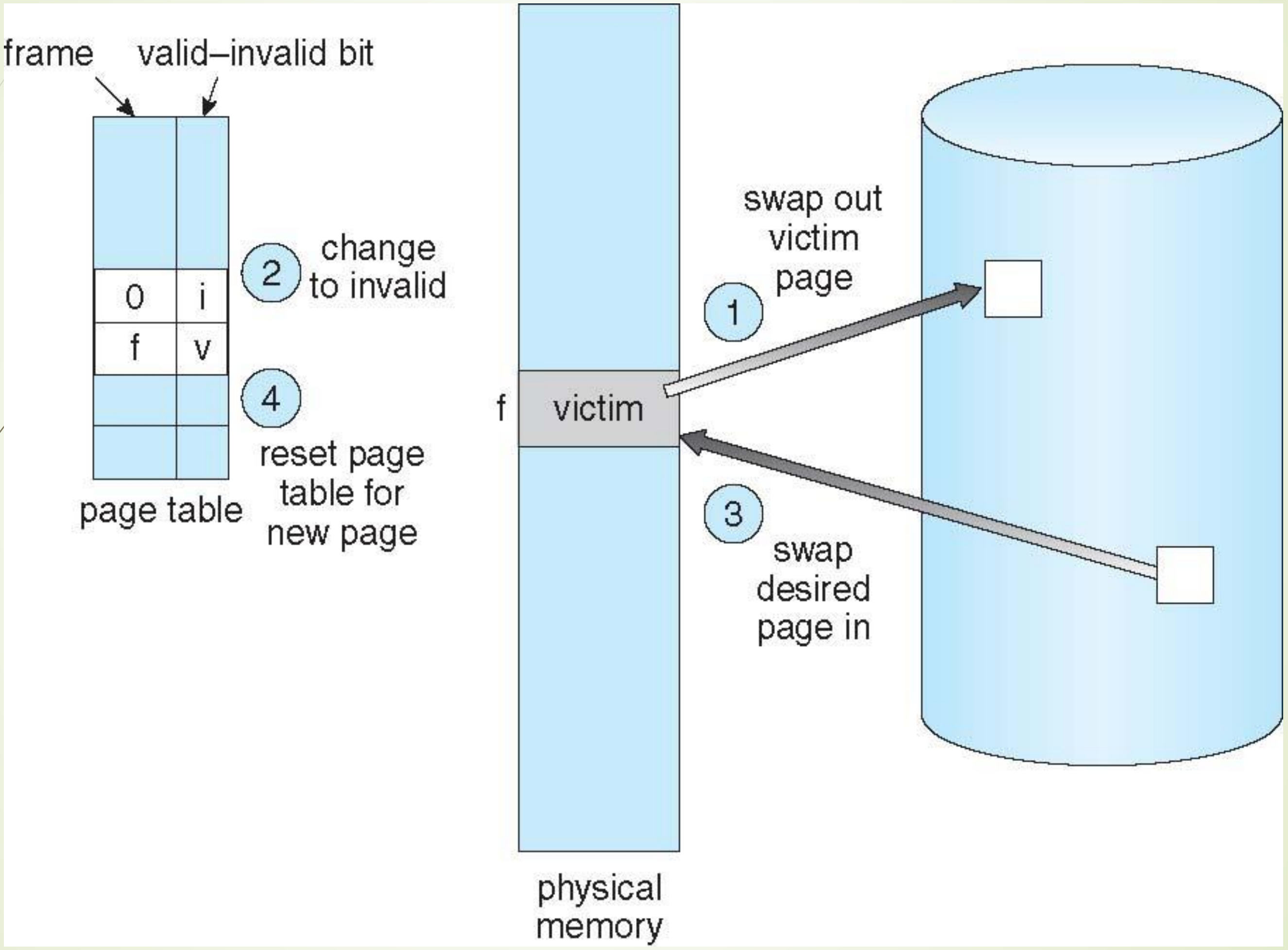
[Source: Operating Systems Concepts, 10th ed.]

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



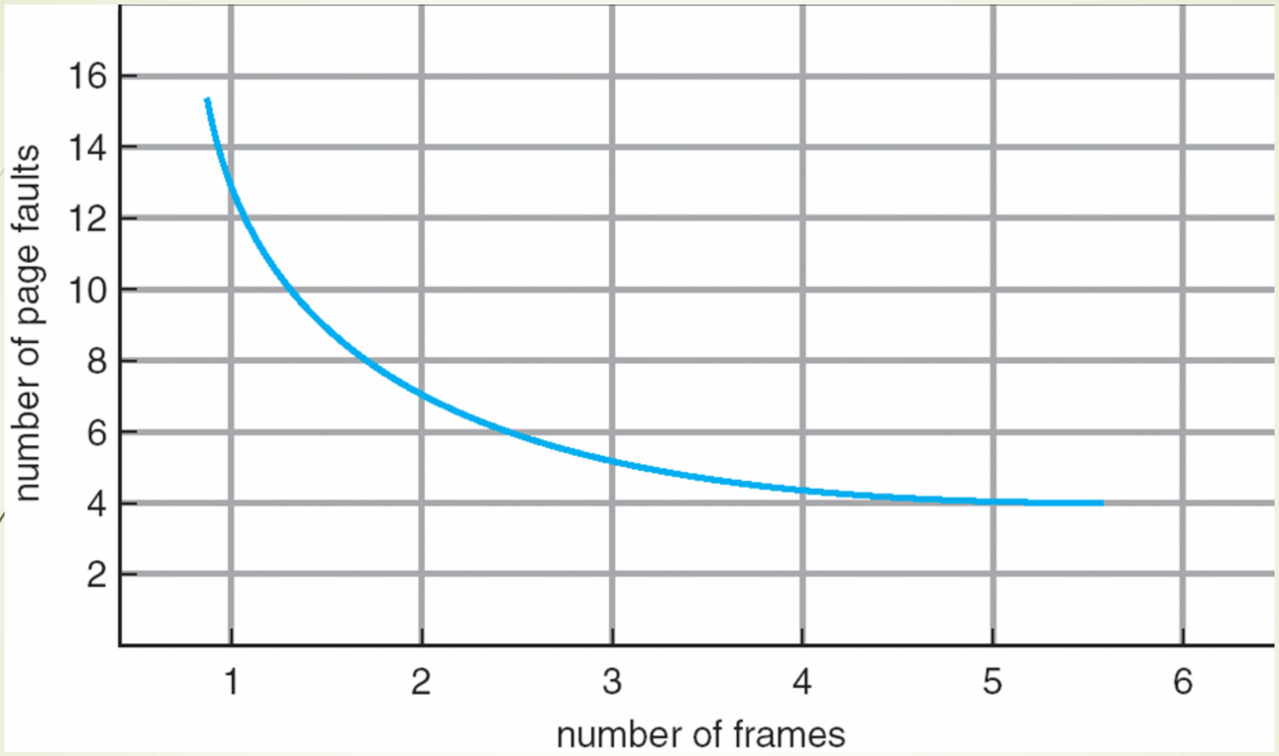
[Source: Operating Systems Concepts, 10th ed.]

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

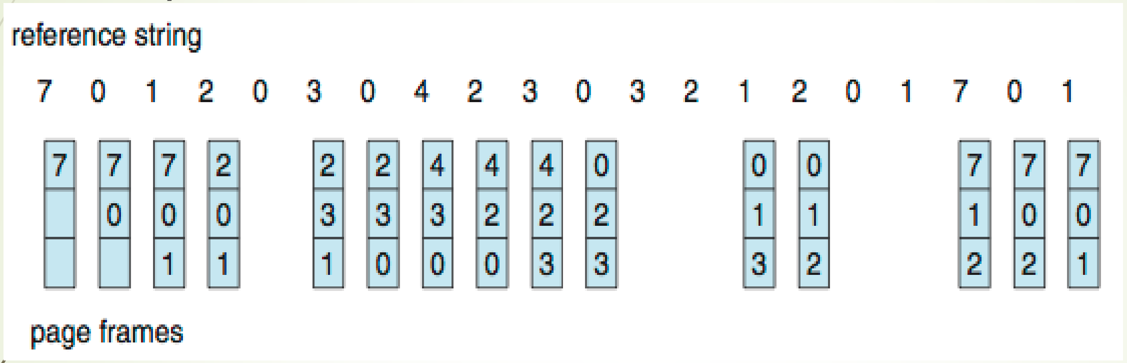
Graph of Page Faults Versus the Number of Frames



[Source: Operating Systems Concepts, 10th ed.]

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)



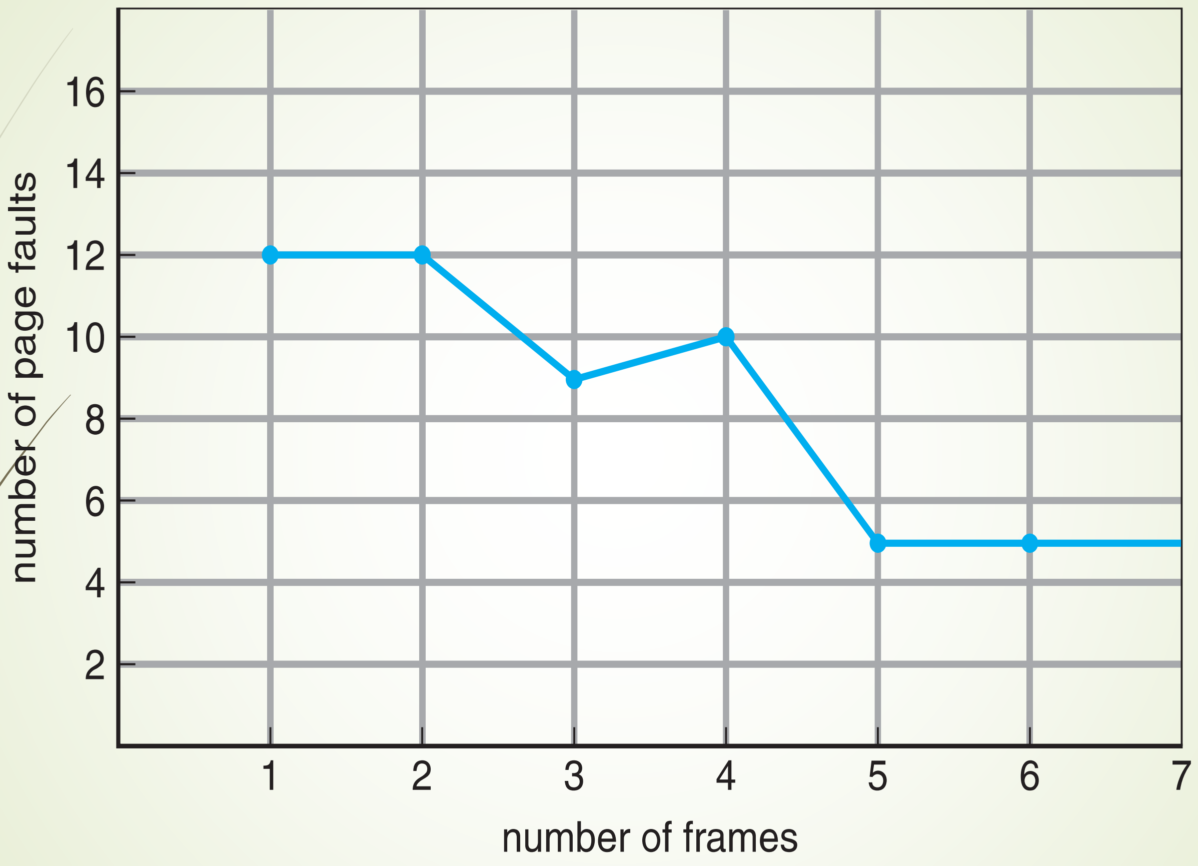
7	7	7	2	2	2	4	4	4	0	0	0	7	7	7
	0	0	0	3	3	3	2	2	2	1	1	1	0	0
		1	1	1	0	0	0	3	3	3	2	2	2	1

15 page faults

[Source: Operating Systems Concepts, 10th ed.]

- Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5
  - Adding more frames can cause more page faults!
    - **Belady's Anomaly**
- How to track ages of pages?
  - Just use a FIFO queue

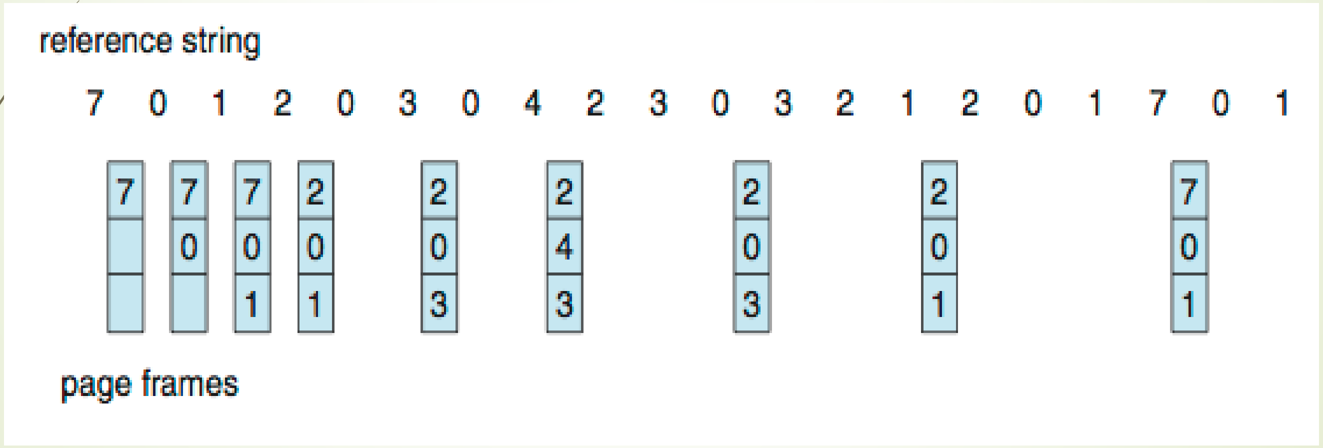
FIFO Illustrating Belady's Anomaly



[Source: Operating Systems Concepts, 10th ed.]

Optimal Algorithm

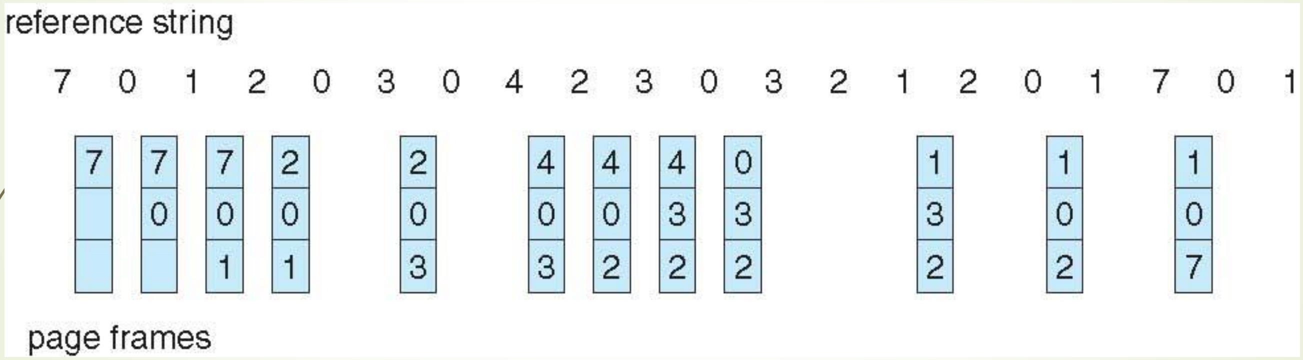
- Replace page that will not be used for longest period of time
  - 9 is optimal for the example
- How do you know this?
  - Can't read the future
  - Used for measuring how well your algorithm performs



[Source: Operating Systems Concepts, 10th ed.]

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?

[Source: Operating Systems Concepts, 10th ed.]

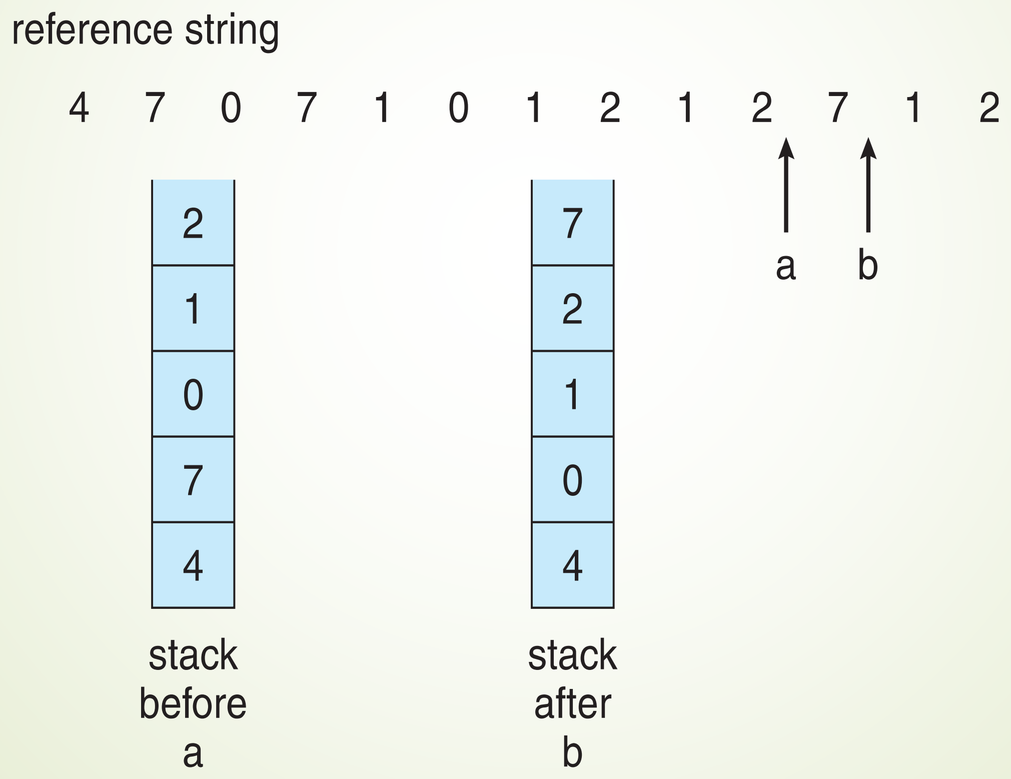
LRU Algorithm (Cont.)

- Counter implementation
  - Every page entry has a counter
    - every time page is referenced, copy the clock into the counter
  - When a page needs to be changed, find smallest value among the counters
    - 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
  - No search for replacement
    - But each update more expensive

LRU Algorithm (Cont.)

LRU and OPT are cases of **stack algorithms** that don't have Belady's Anomaly

- Use a stack to record Most Recent page references



[Source: Operating Systems Concepts, 10th ed.]

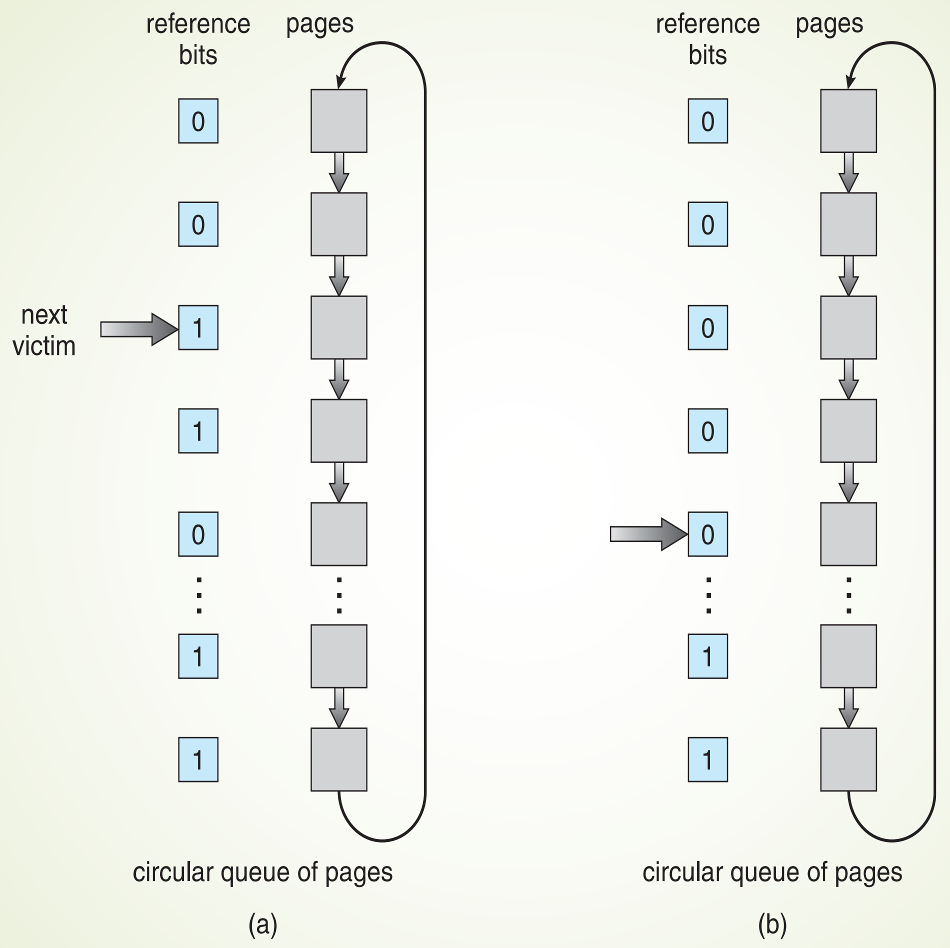
LRU Approximation Algorithms

- LRU needs special hardware and still slow
- **Reference bit**
  - Associated bit for each page, 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

LRU Approximation Algorithms (cont.)

- **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 Algorithm



[Source: Operating Systems Concepts, 10th ed.]

Enhanced Second-Chance Algorithm

- Improve algorithm by using reference bit and modify bits (if available)
- For ordered pair (reference, modify):
  - (0, 0) neither recently used nor modified – best page to replace
  - (0, 1) not recently used but modified – not quite as good, must write out before replacement
  - (1, 0) recently used but clean – probably will be used again soon
  - (1, 1) recently used and modified – probably will be used again soon and need to write out before replacement
- When page replacement needed, use the clock scheme but use the four classes to replace page in lowest non-empty class
  - Might need to search circular queue several times

Counting Algorithms

- Keep a counter of the number of references that have been made to each page
- **Least Frequently Used** (**LFU**) **Algorithm**
  - Replaces page with smallest count
- **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
- Not common
  - Implementation is expensive

Applications and Page Replacement

- In all of these algorithms, OS guesses 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
- If given direct access to the disk, OS getting out of the way of the applications
  - **Raw disk** mode
- Bypasses buffering, locking, etc.

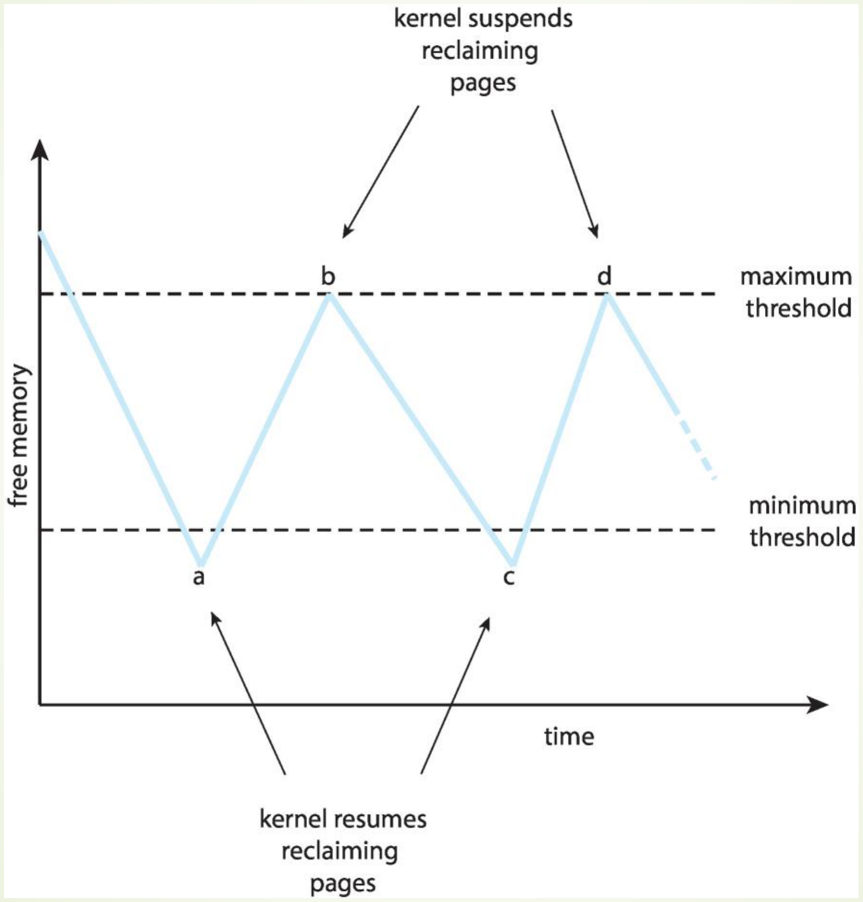
Global vs. Local Allocation

- **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another
  - But then process execution time can vary greatly
  - But greater throughput, so it's more common
- **Local replacement** – each process selects from only its own set of allocated frames
  - More consistent per-process performance
  - But possibly under-utilized memory

Reclaiming Pages

- A strategy to implement global page-replacement policy
- All memory requests are satisfied from the free-frame list
  - Rather than waiting for the list to drop to zero before we begin selecting pages for replacement
- Page replacement is triggered when the list falls below a certain threshold
- This strategy attempts to ensure there is always sufficient free memory to satisfy new requests

Reclaiming Pages Example



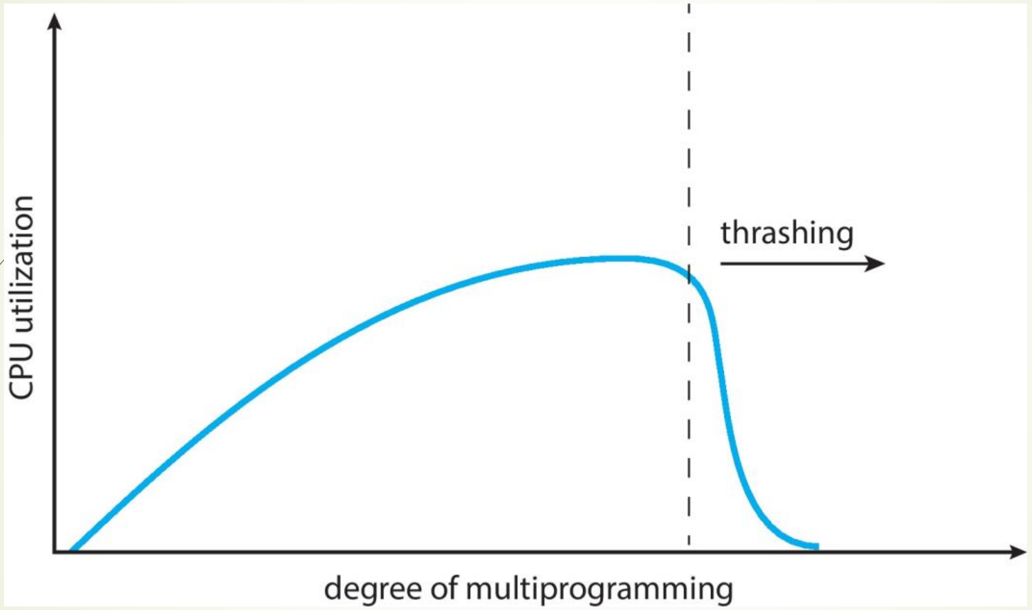
[Source: Operating Systems Concepts, 10th ed.]

Thrashing

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

Thrashing (Cont.)

- **Thrashing**: A process is busy swapping pages in and out



[Source: Operating Systems Concepts, 10th ed.]

Demand Paging and Thrashing

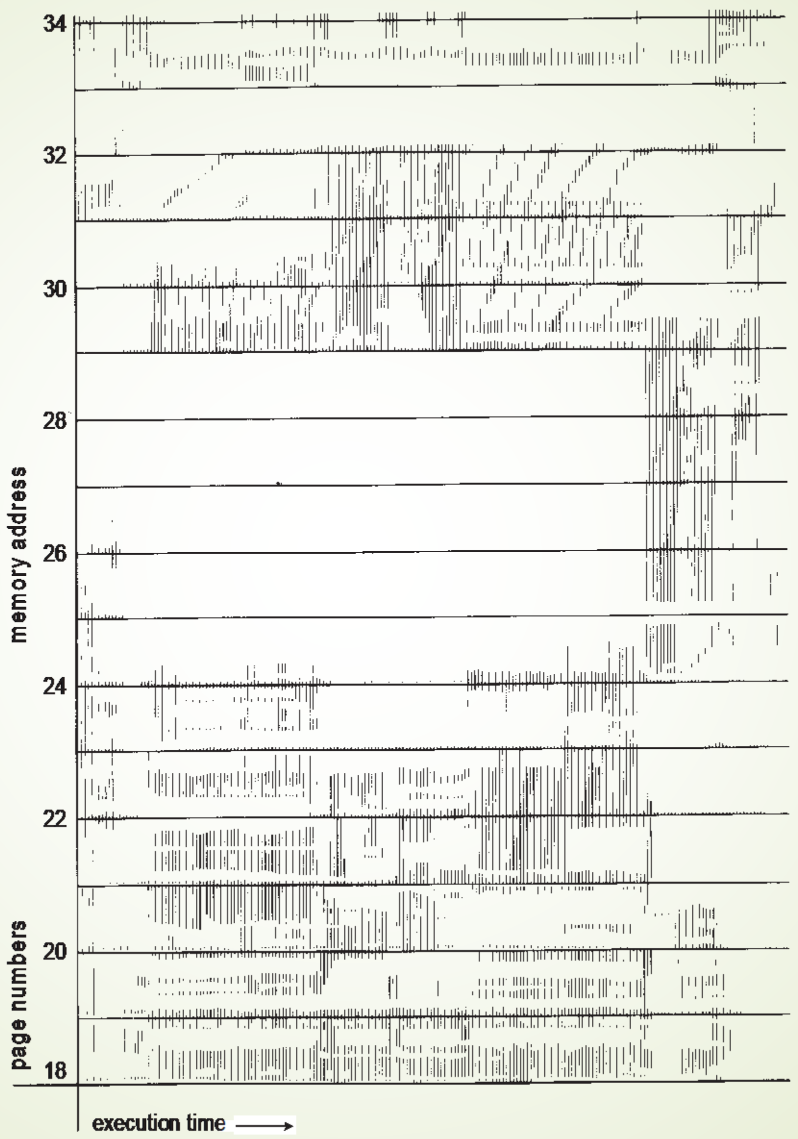
- Why does demand paging work?

  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



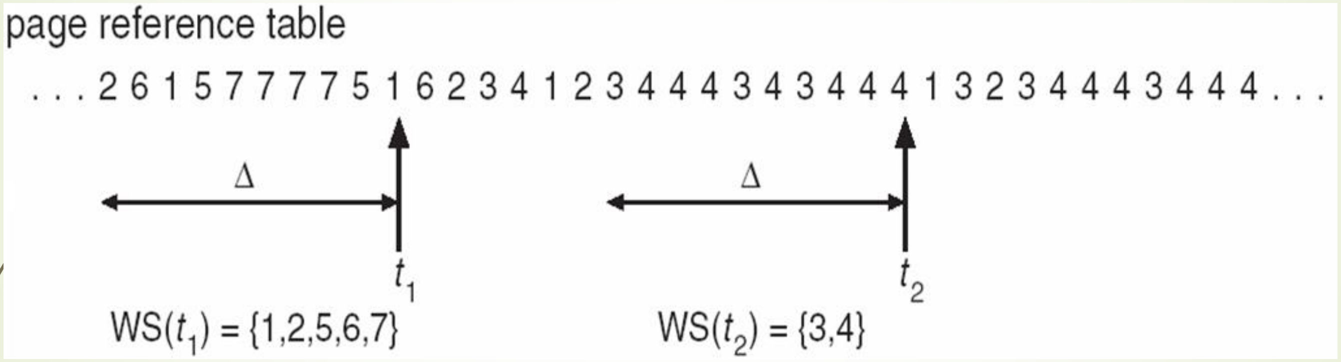
[Source: Operating Systems Concepts, 10th ed.]

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

Working-Set Model (Cont.)

- if $D > m \Rightarrow$ Thrashing
- Policy if D > m, then suspend or swap out one of the processes

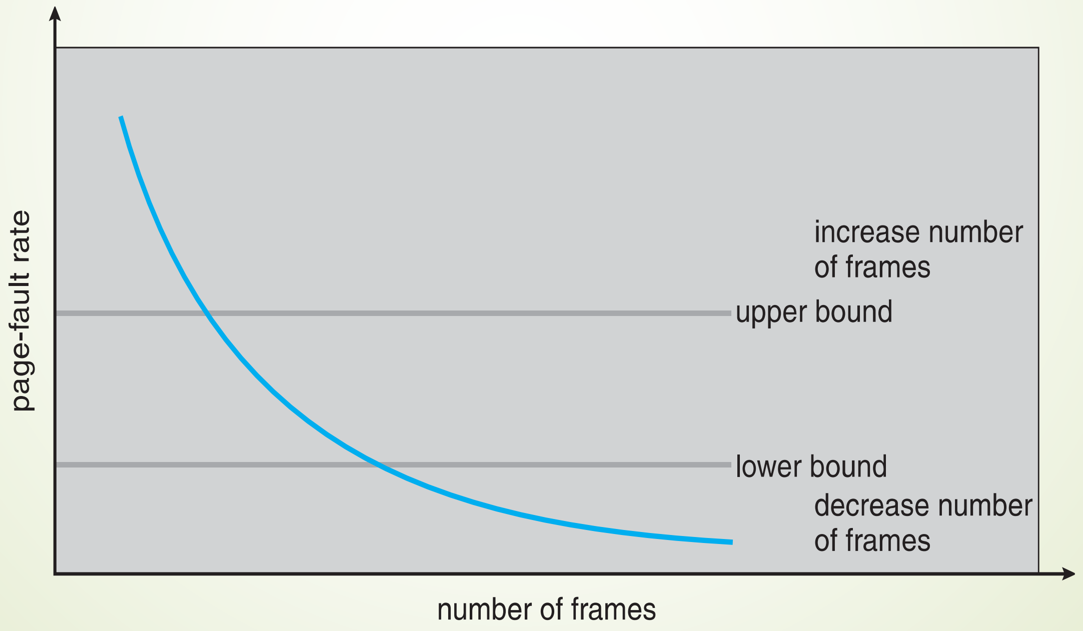


Keeping Track of the Working Set

- 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

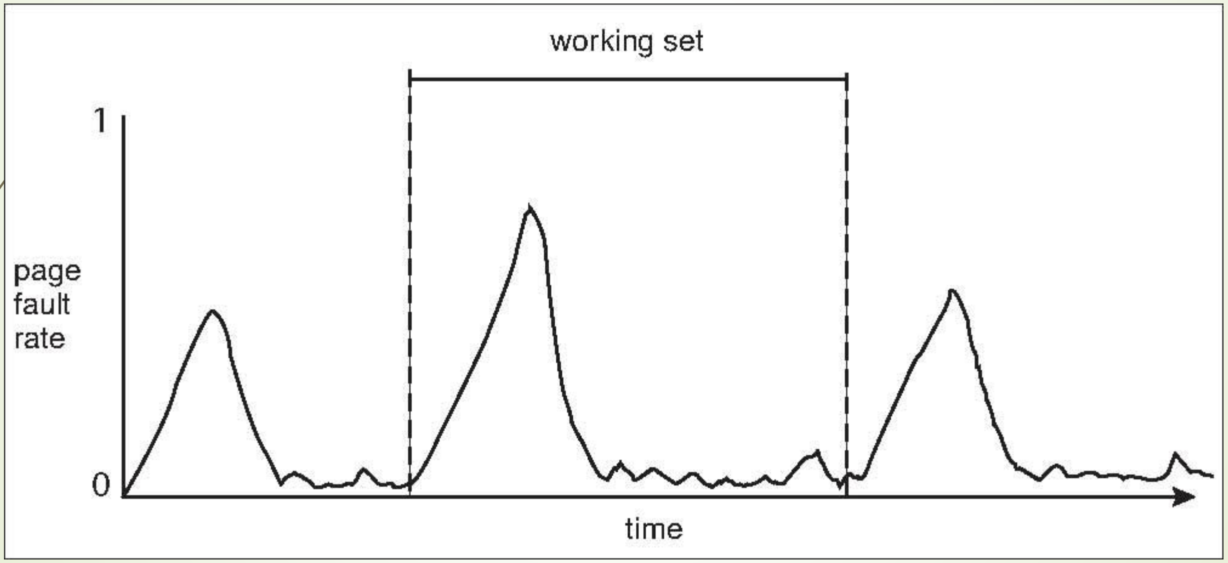
- More direct approach than WSS
- Establish "acceptable" **page-fault frequency** (**PFF**) rate and use local replacement policy
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame



[Source: Operating Systems Concepts, 10th ed.]

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



[Source: Operating Systems Concepts, 10th ed.]

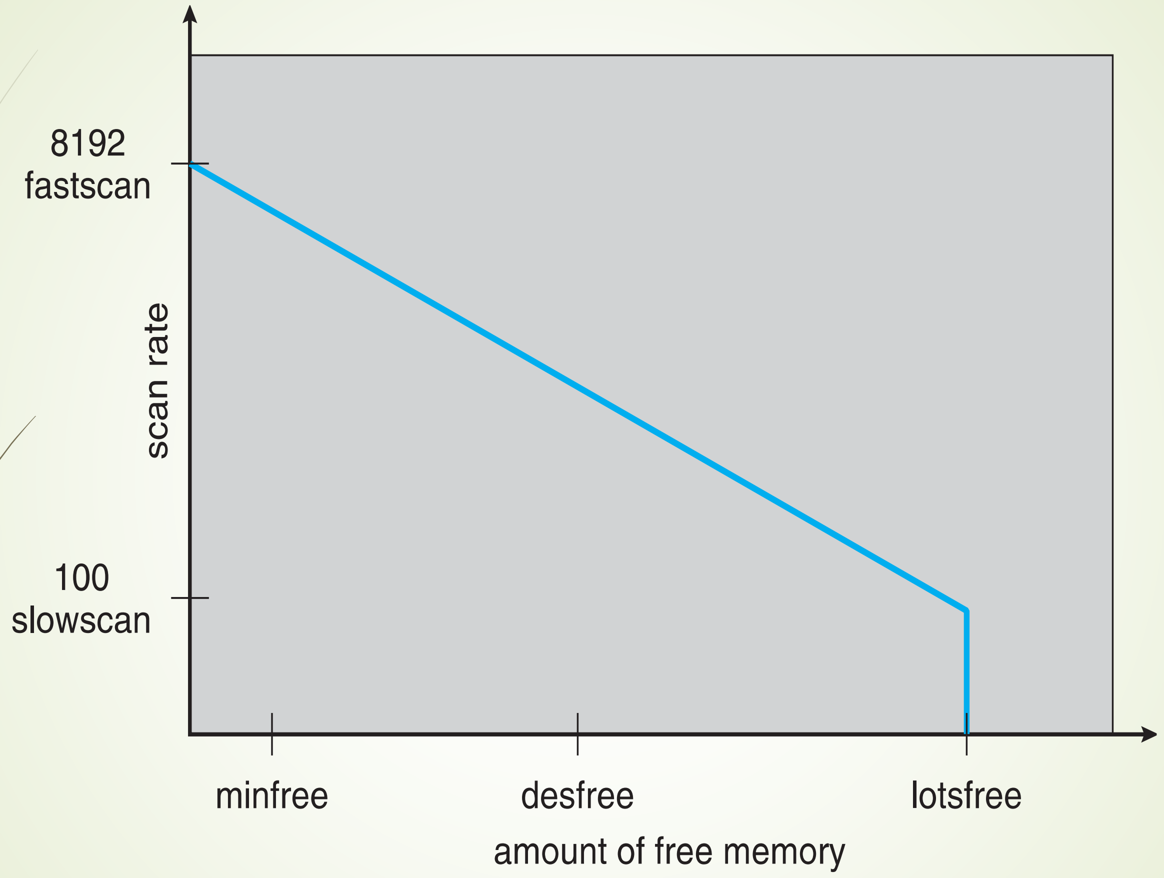
Operating System Examples

- Solaris

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



[Source: Operating Systems Concepts, 10th ed.]

End of Module 2