CS 5323 – OS II

Lecture 9 – Main Memory Management

Logistics

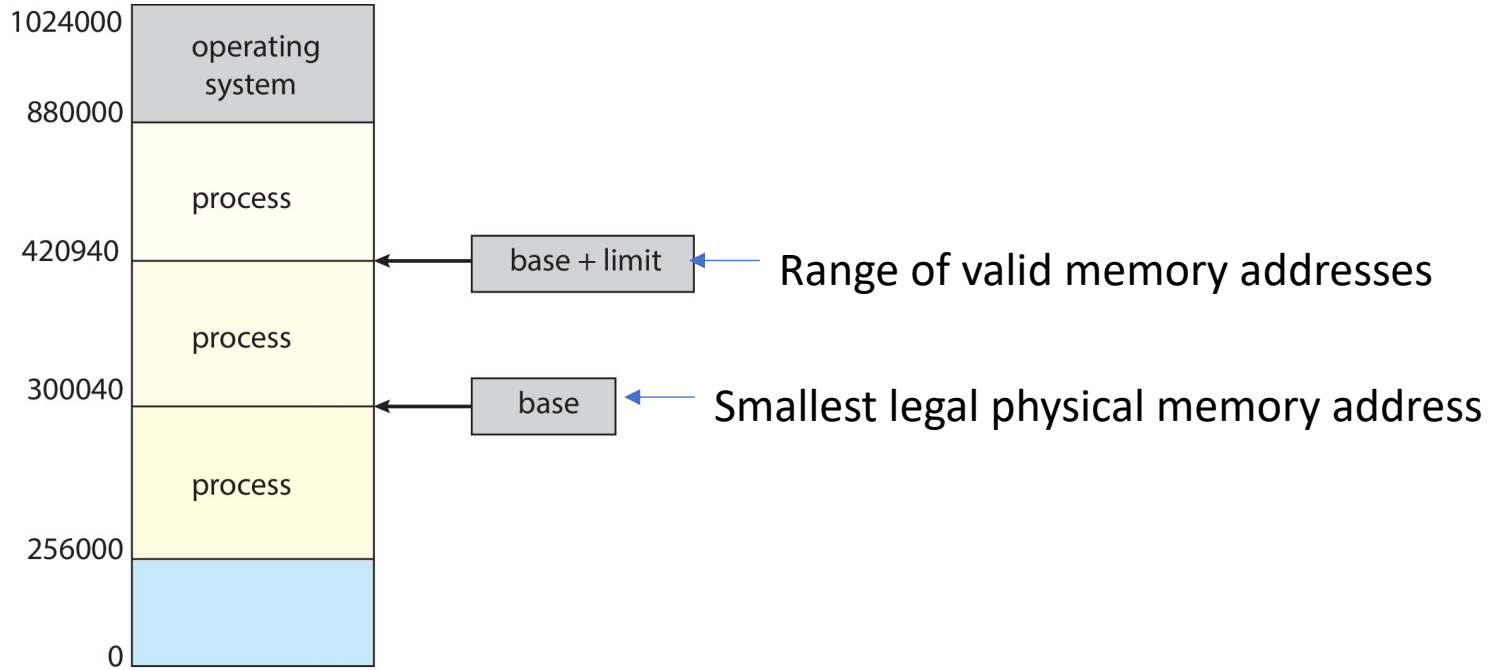
- Quiz 4 will be posted this week and is due: Monday 03/07/2022 11:59 pm

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.

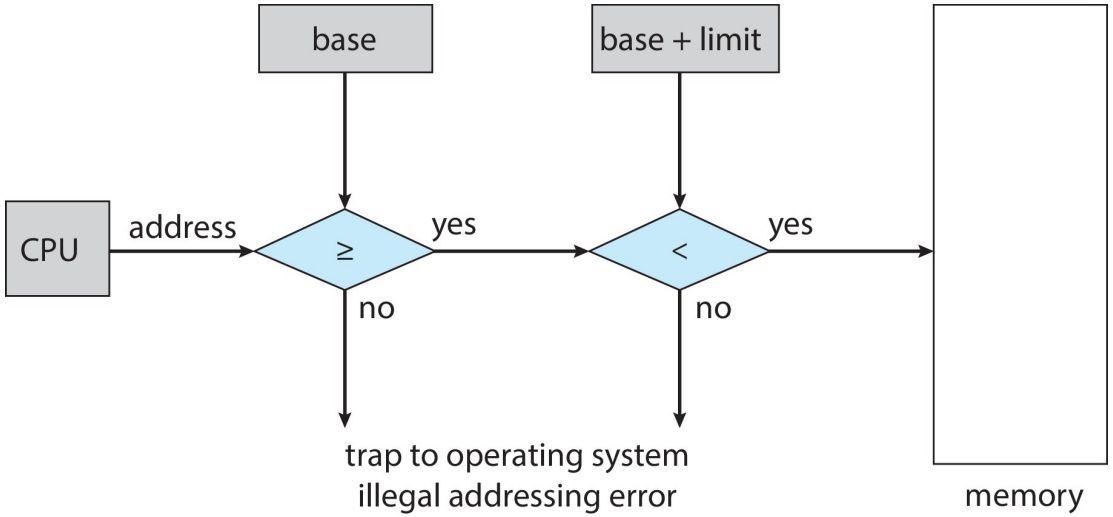
Protection

- Need to censure that a process can access only access those addresses in it address space.
- We can provide this protection by using a pair of **base** and **limit registers** define the logical address space of a process



Hardware Address Protection

- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user



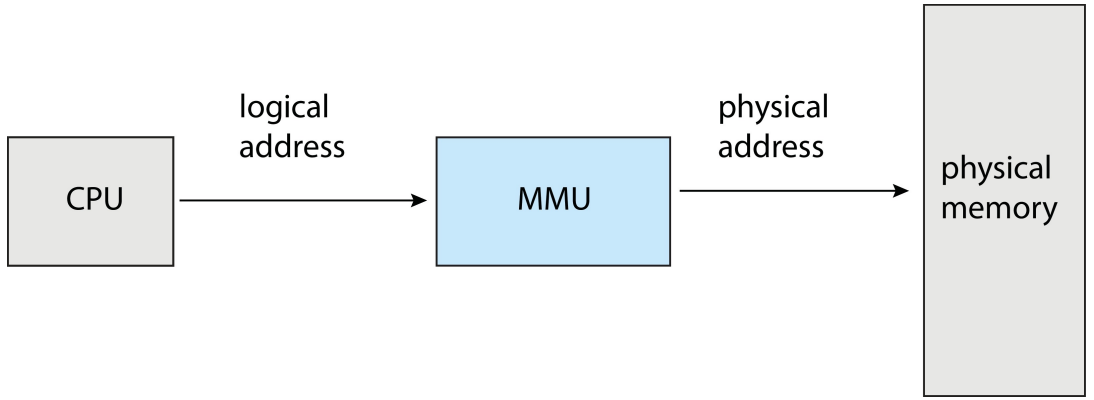
- the instructions to loading the base and limit registers are **privileged**

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; also referred to as **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; logical (virtual) and physical addresses 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 at run time maps virtual to physical address



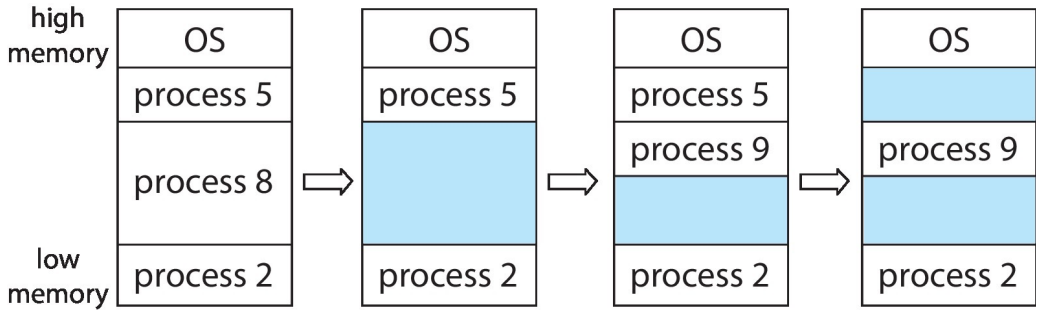
- Many methods possible, covered in the rest of this chapter

Contiguous Allocation

- Main memory must support both OS and user processes
- Limited resource, must allocate efficiently
- Contiguous allocation is one early method
- Main memory usually into two **partitions**:
  - Resident operating system, 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 to the section containing next process

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.)

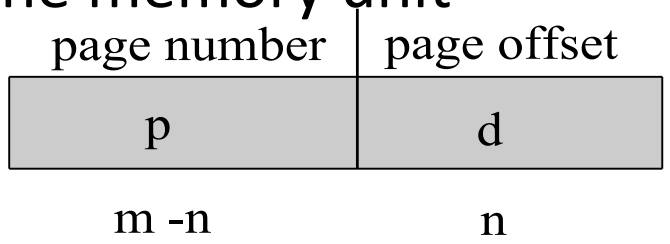
- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time
  - I/O problem
    - Latch job in memory while it is involved in I/O
    - Do I/O 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

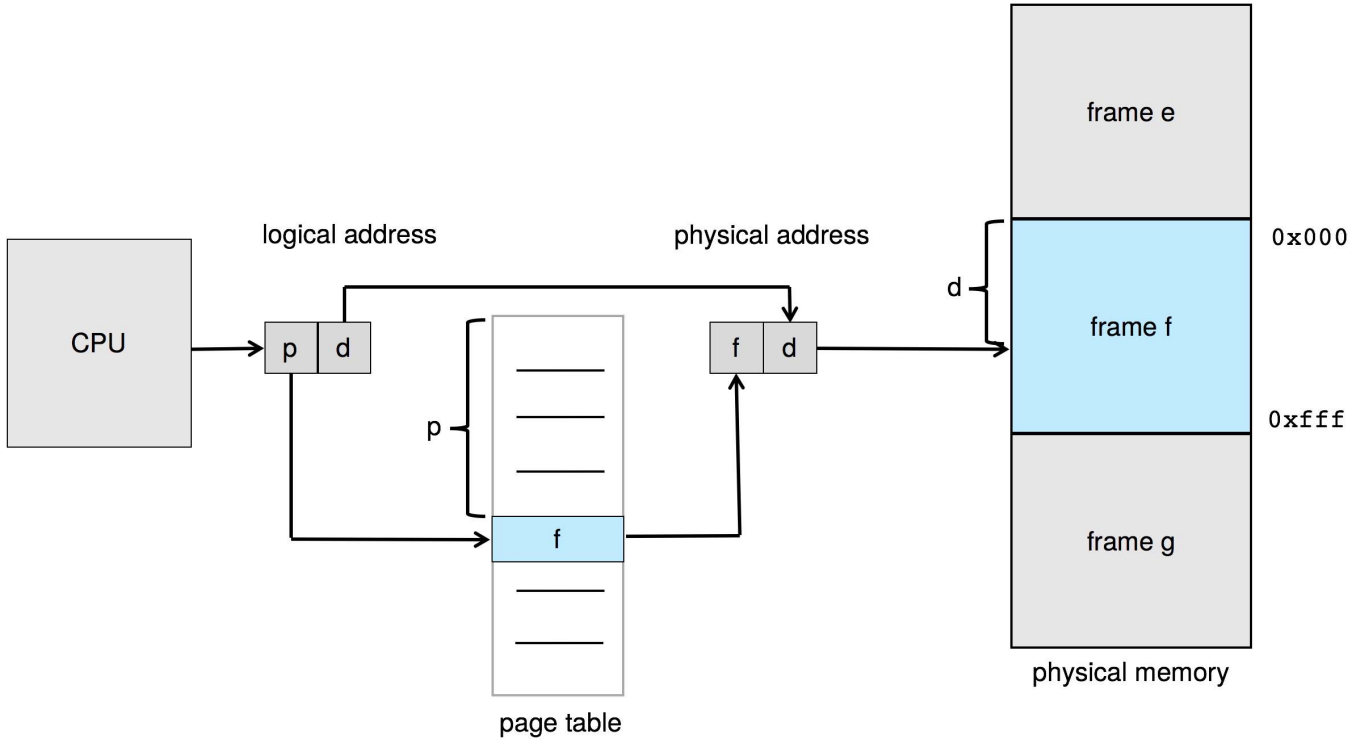
Address Translation Scheme

- 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

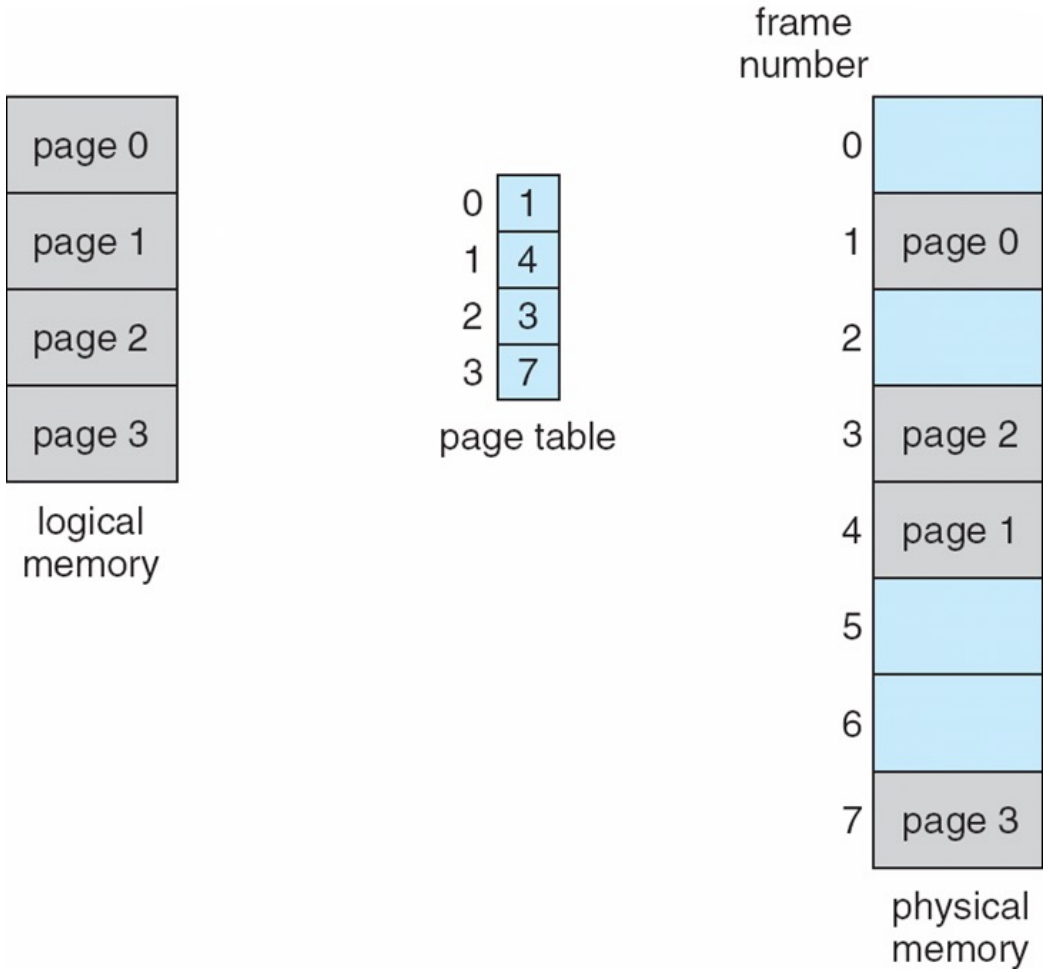


  - If the logical address space is 2m and the page size is 2 n
    - The binary representation of the logical address has m bits, such that
      - The m – n leftmost bits designate the page number p
        - p is index into the page table
      - The rightmost n bits designate the page offset d
   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

Paging Hardware

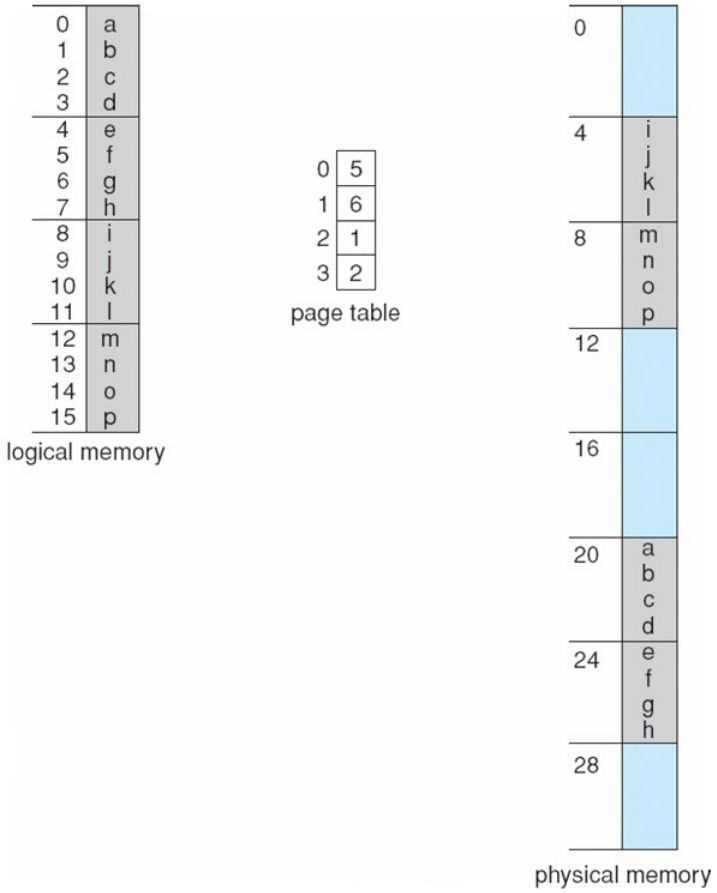


Paging Model of Logical and Physical Memory



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)

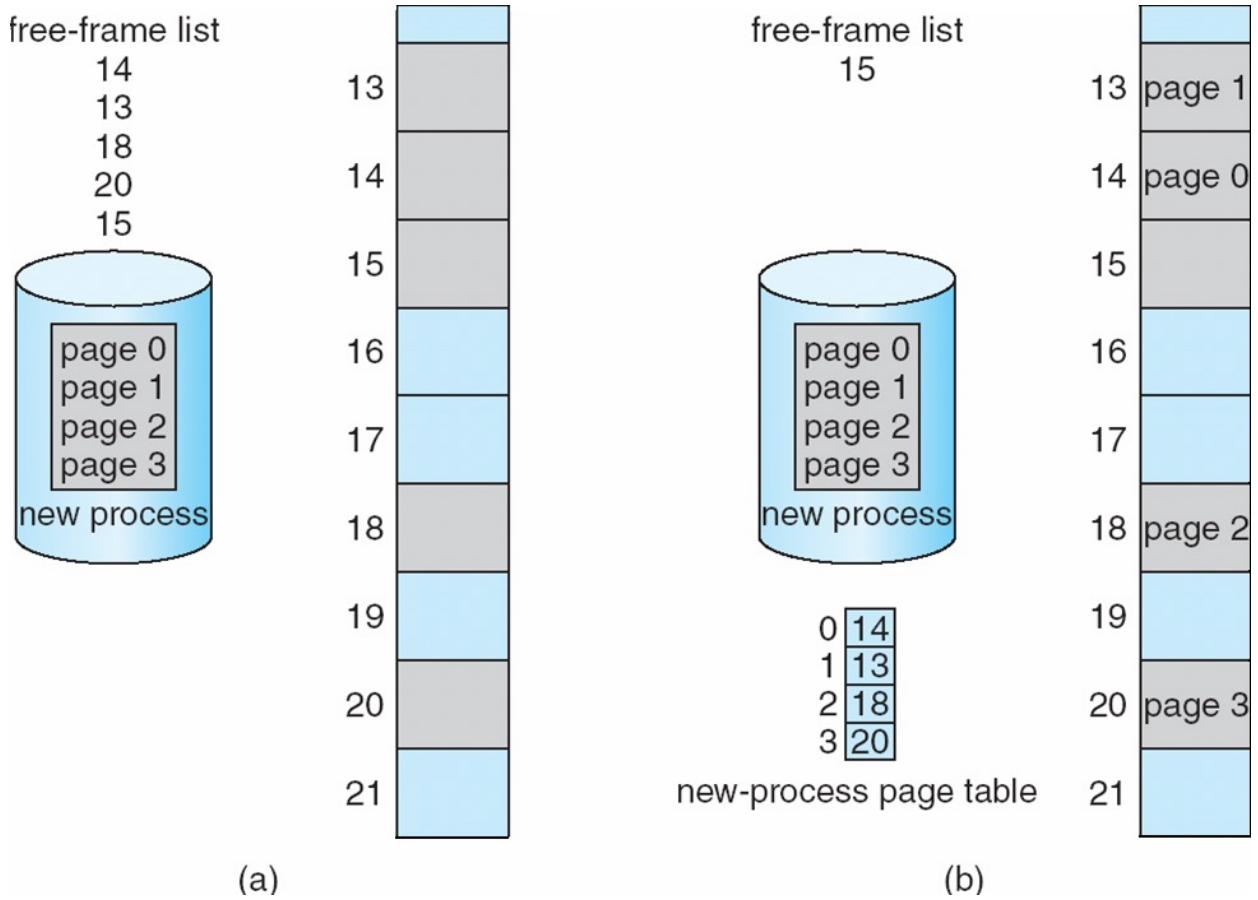


# of pages = total bytes/page size

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



Before allocation

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 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

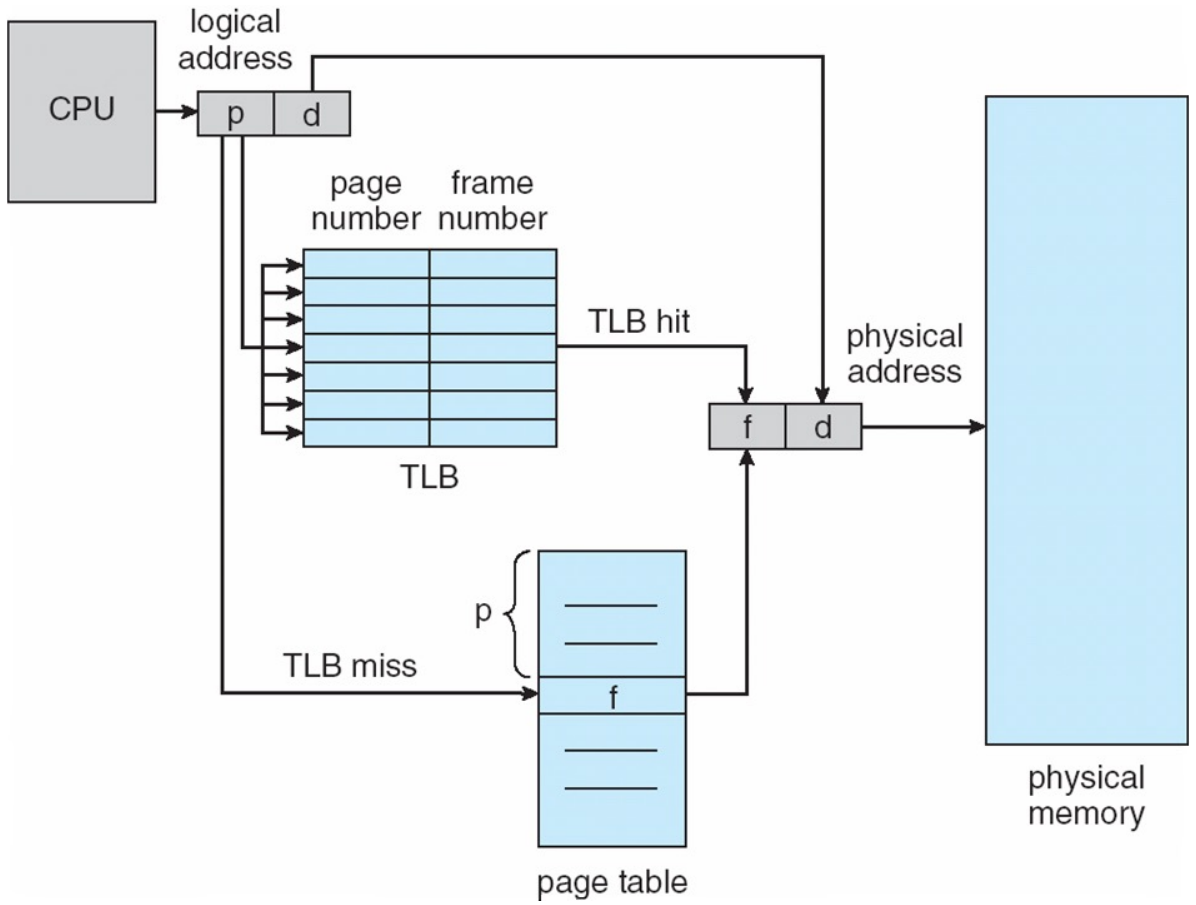
Hardware

- Associative memory – parallel search

Page #	Frame #

- Address translation (p, d)
  - If p is in associative register, get frame # out
  - Otherwise get frame # from page table in memory

Paging Hardware With TLB



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 = hit_rate x TLB Access Time + (1 – hit_rate) * miss_penalty

  Here, 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%,
  - Then, EAT = 0.99 x 10 + 0.01 x 20 = 10.1ns, implying only 1% slowdown in access time.

Measures impact of TLB by considering memory access time to be 1. Not actual EAT!

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`

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

- Consider, a = 80%, e = 20ns, m = 100ns,
  - `EAT = 0.80 x (100 + 20) + (1 – 0.80)x(2*100 + 20) = 140ns` (40% slowdown)
- Consider, a = 99%, e = 20ns, m = 100ns, what is the EAT?

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`

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

- Consider, a = 80%, e = 20ns, m = 100ns,
  - `EAT = 0.80 x (100 + 20) + (1 – 0.80)x(2*100 + 20) = 140ns` (40% slowdown)
- Consider, a = 99%, e = 20ns, m = 100ns,
  - `EAT = 0.99 x (100 + 20) + (1 – 0.99)x(2*100 + 20) = 121ns` (21% slowdown)

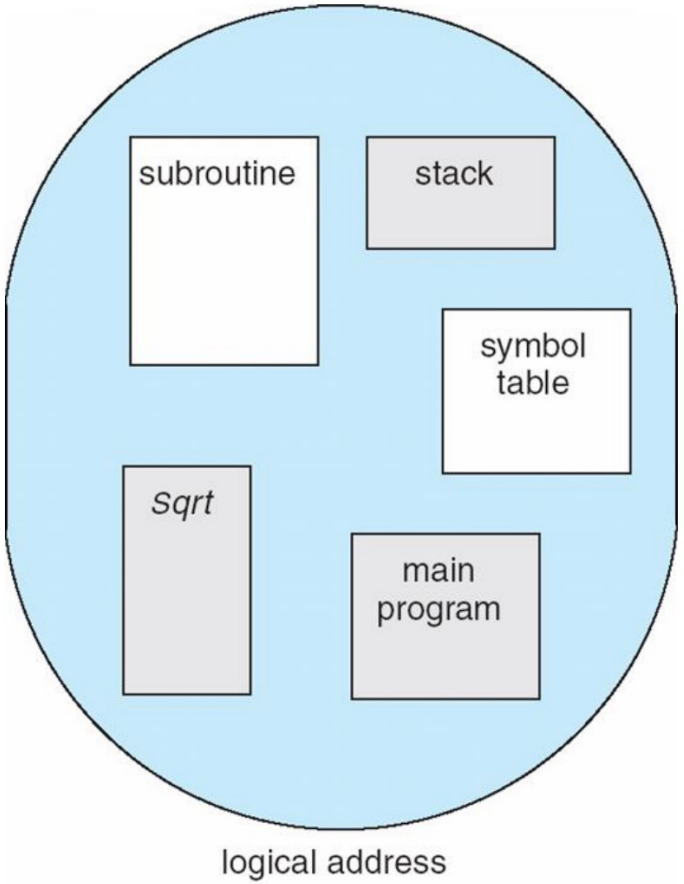
Memory Protection

- 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

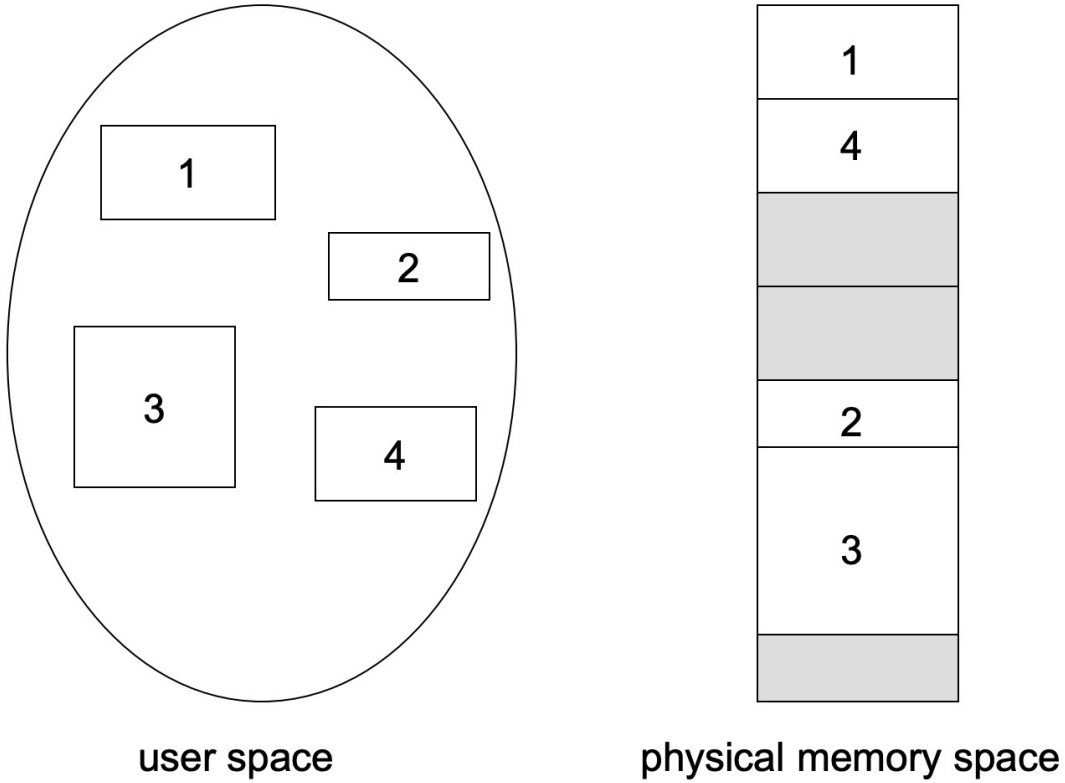
Segmentation

- 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

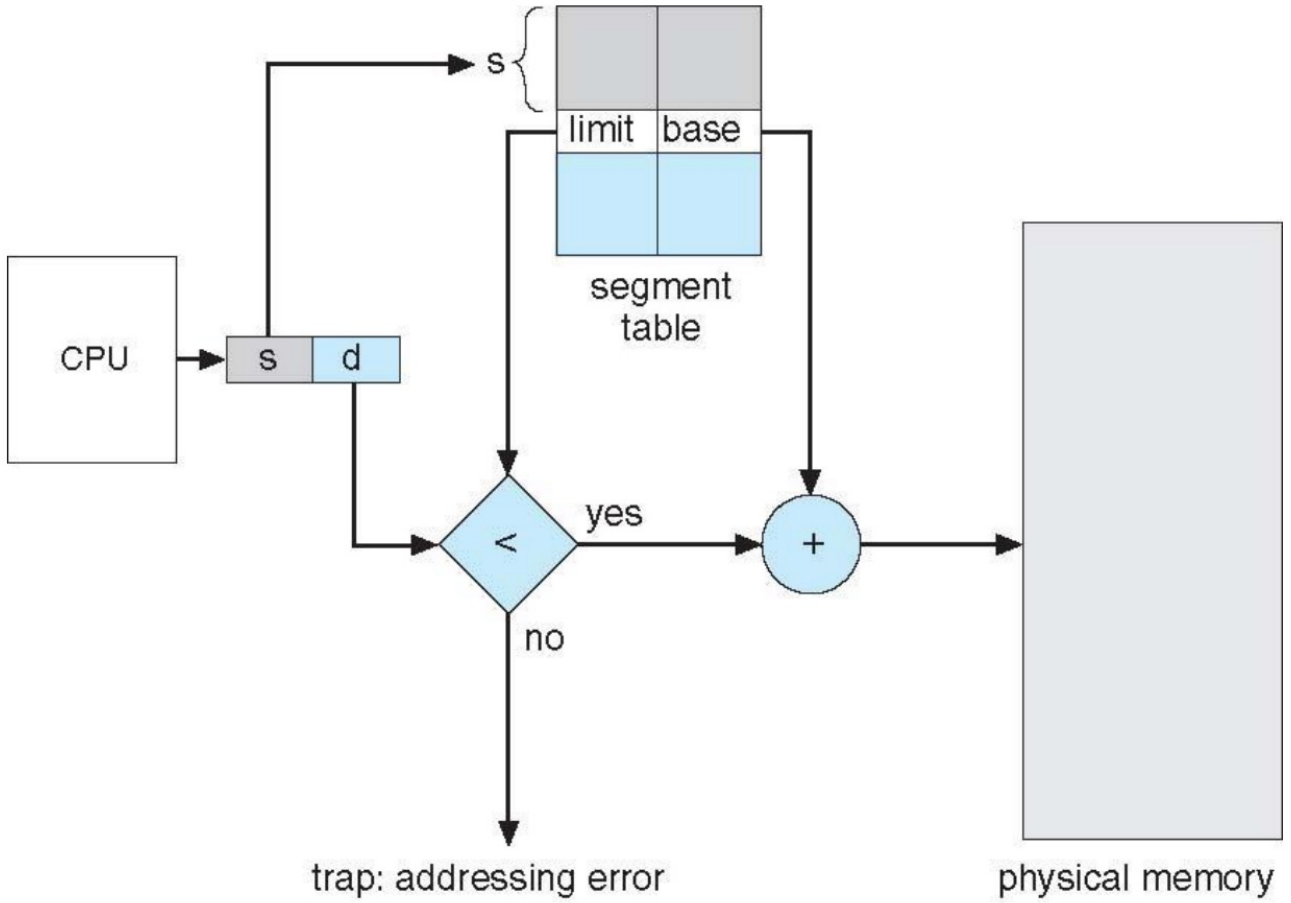
- Logical address consists of a two tuple:

  <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 < STLR

Segmentation Architecture

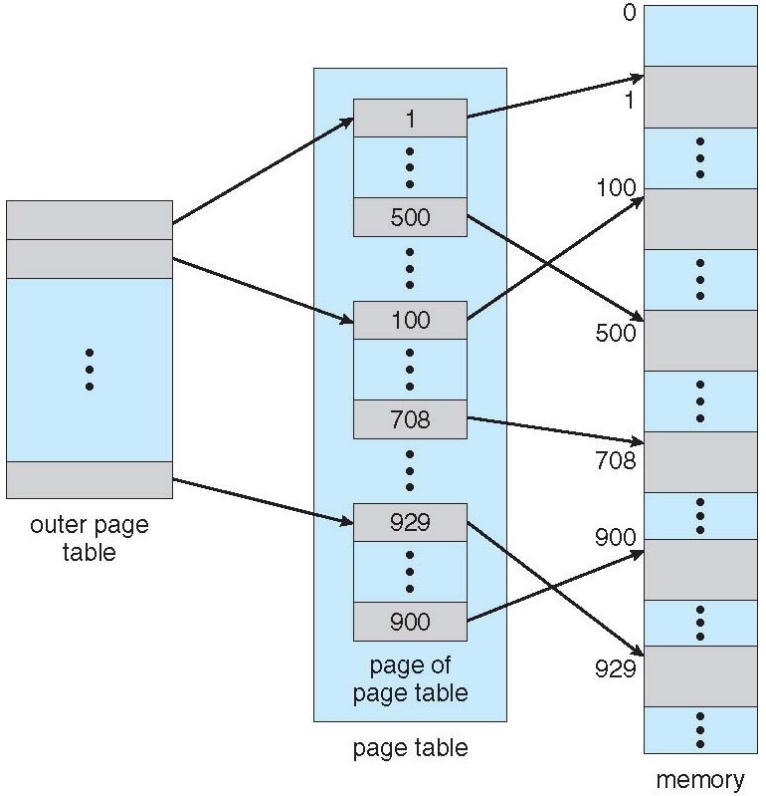


Structure of the Page Table

- 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

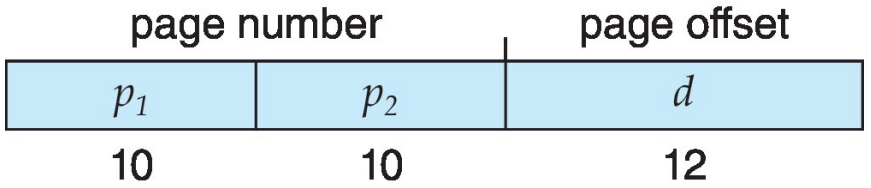
Hierarchical Page Tables

- 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



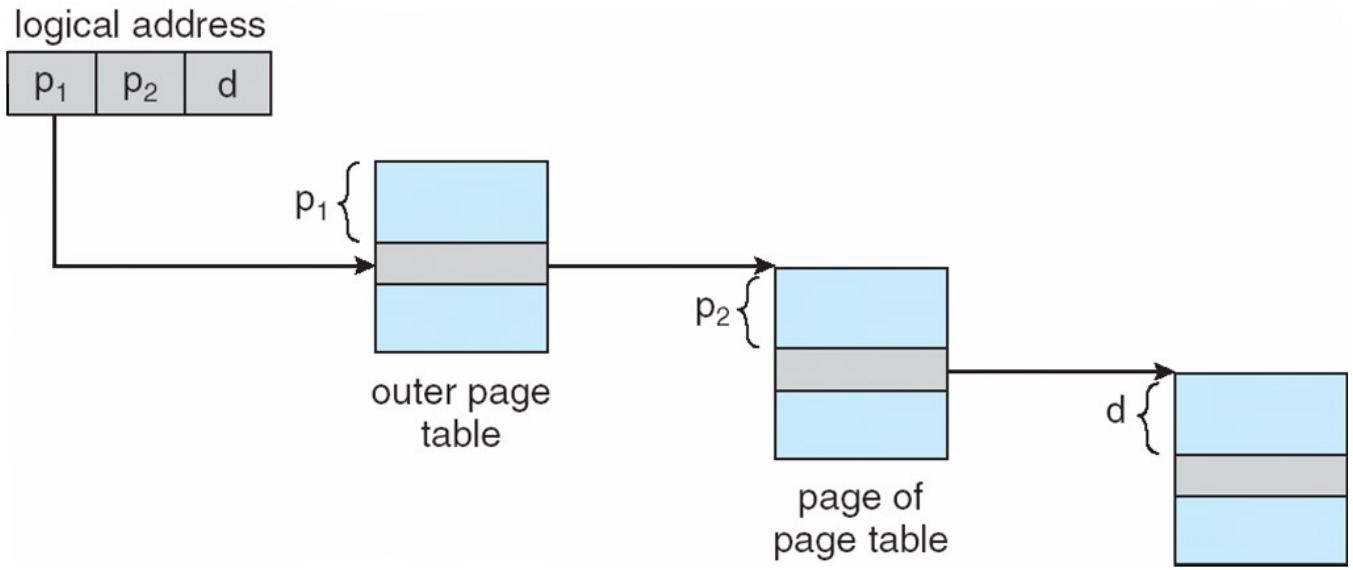
Two-Level Paging Example

- 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:



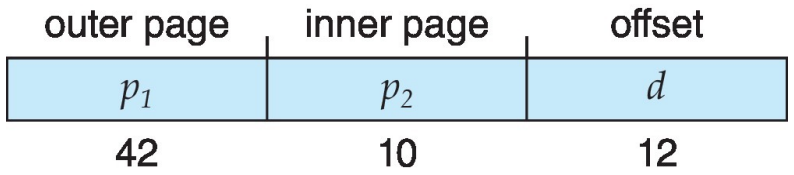
- where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table
- Known as **forward-mapped page table**

Address-Translation Scheme



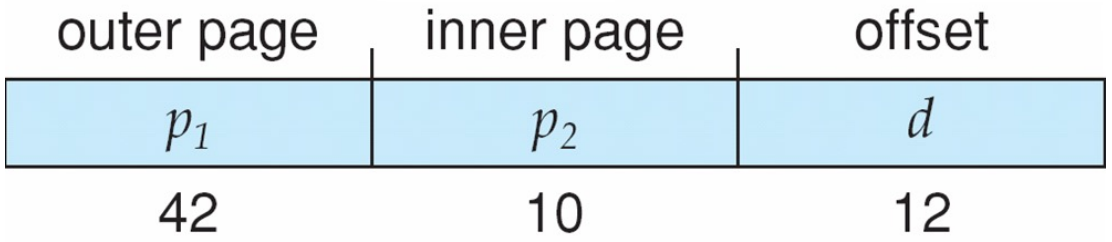
64-bit Logical Address Space

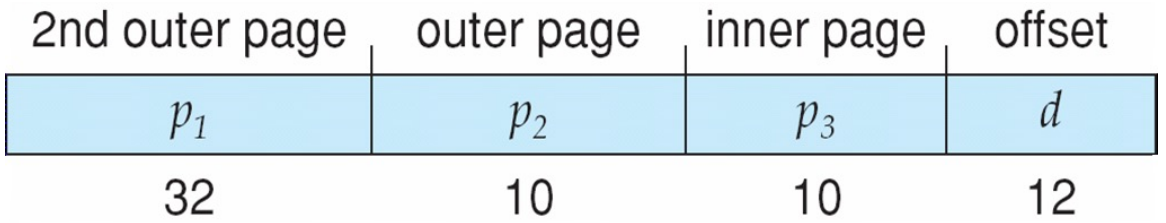
- Even two-level paging scheme not sufficient
  - 64-bit addressing = 2^{64} of physical address space to store a page table
- If page size is 4 KB (2^{12})
  - Then page table has 2^{52} entries; number of pages
  - If two level scheme, inner page tables could be 2^{10} 4-byte entries
  - Address would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



  - Outer page table has 2^{42} entries or 2^{44} bytes
  - One solution is to add a 2^{nd} outer page table
  - But in the following example the 2^{nd} 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

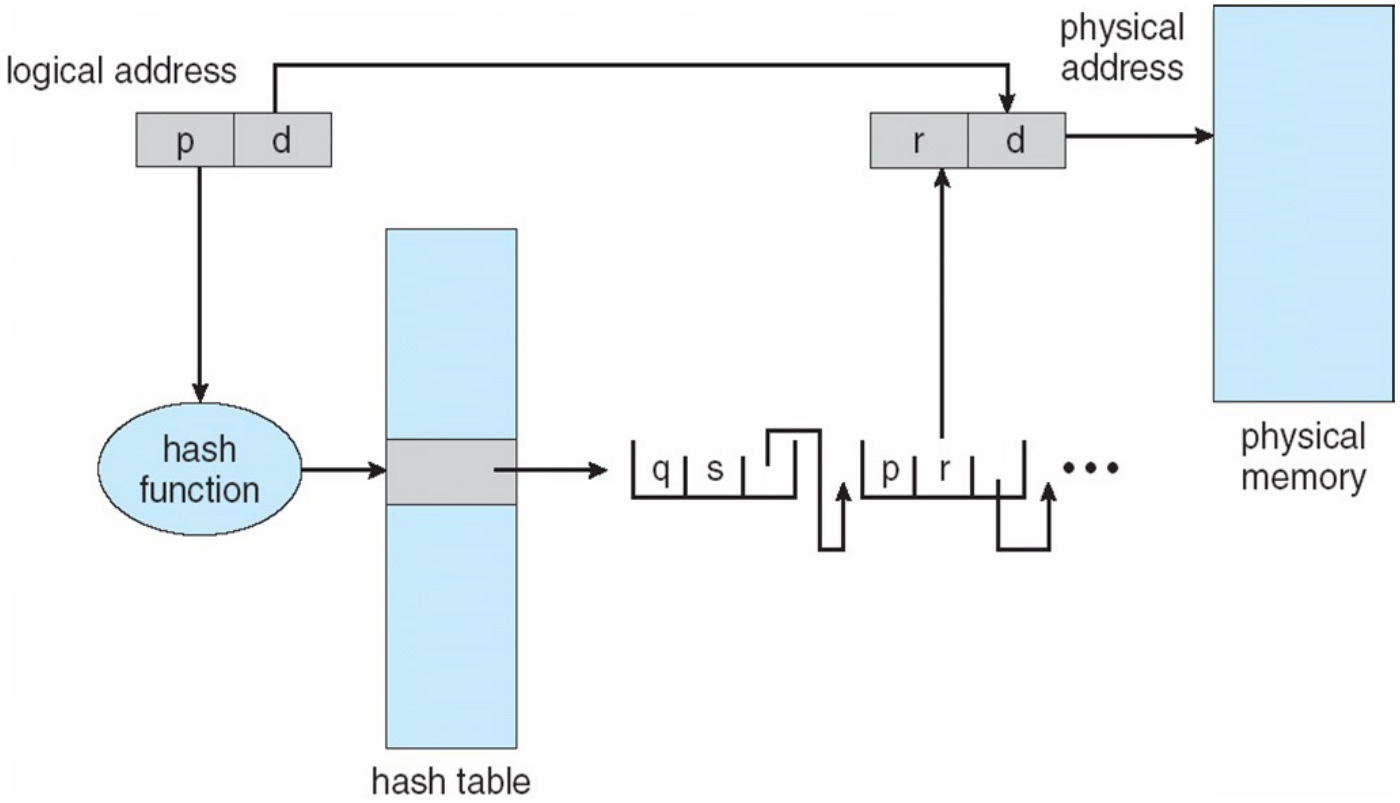




Hashed Page Tables

- 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)

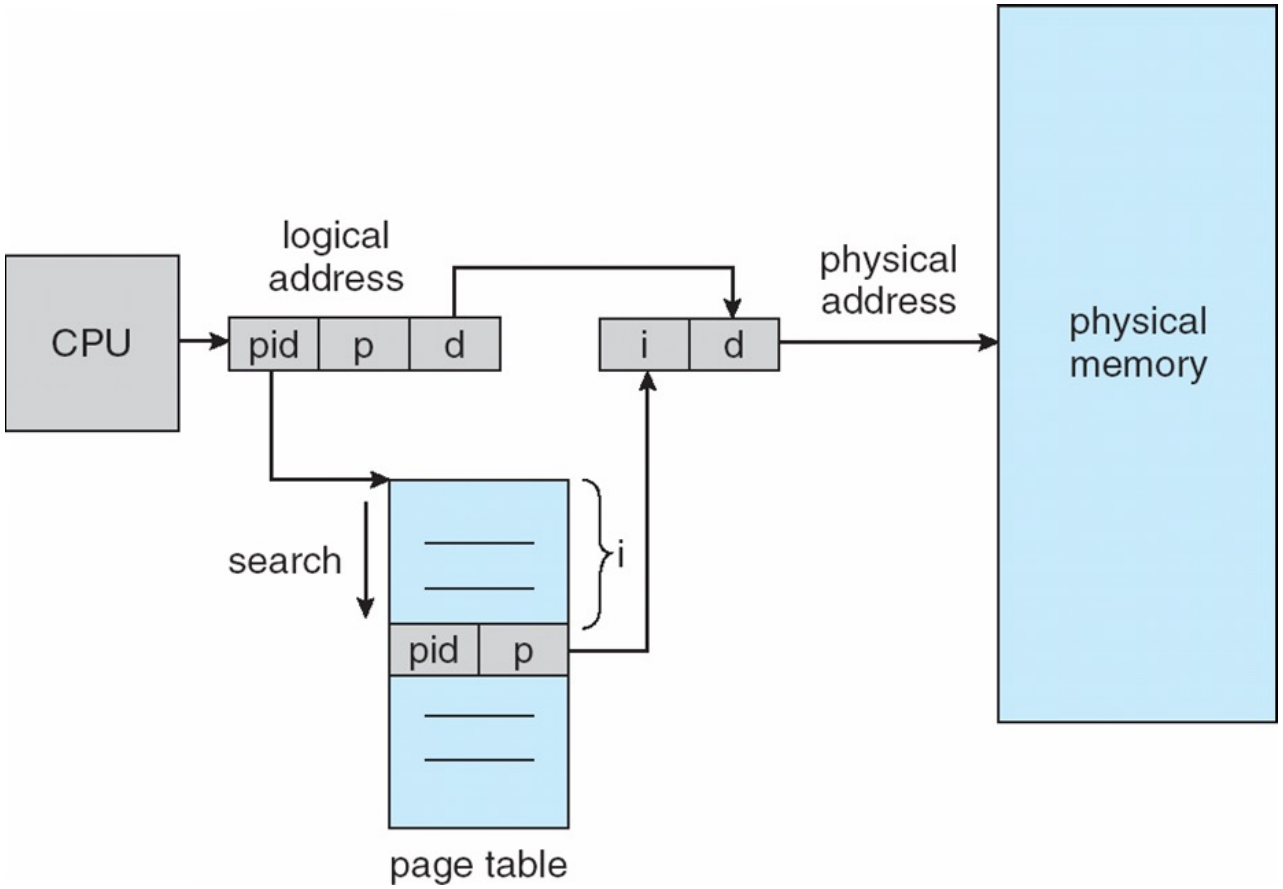
Hashed Page Table



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

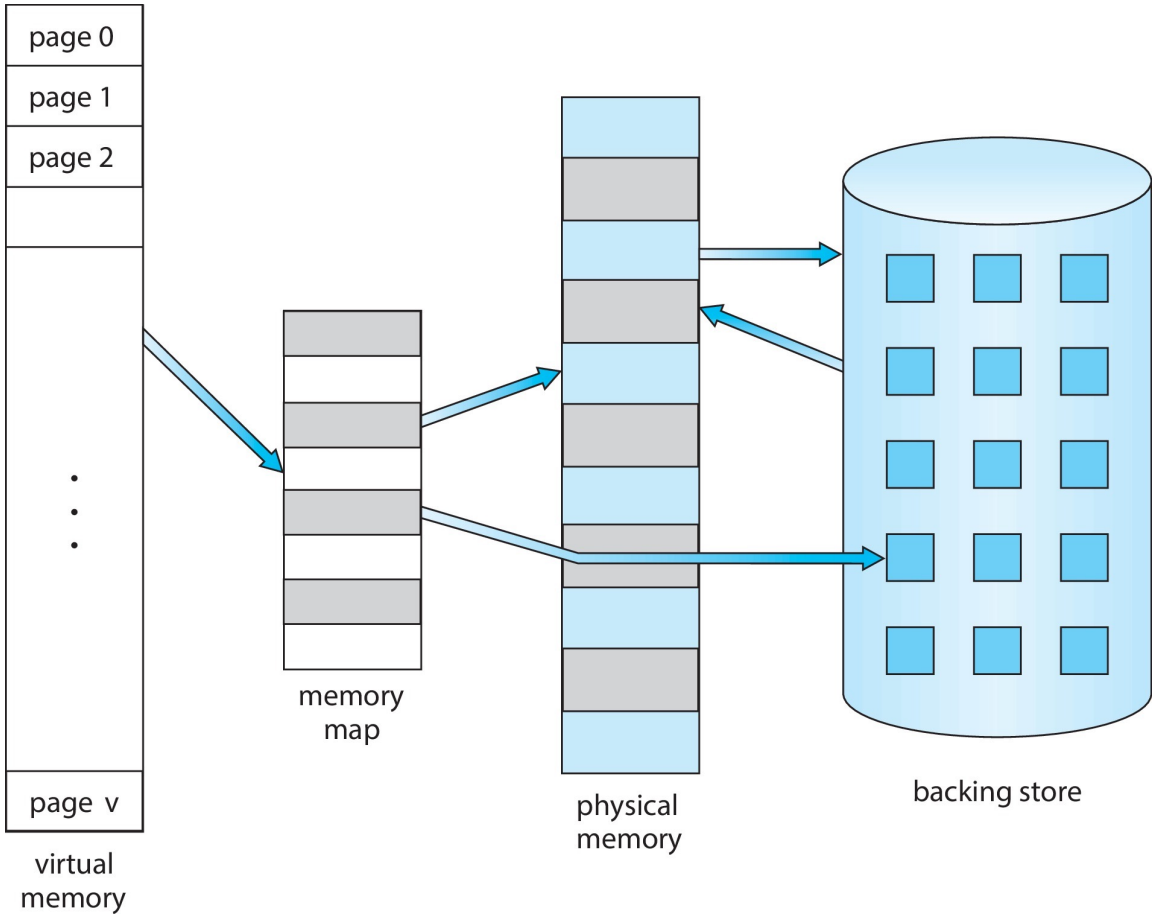
Virtual memory

- **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.)

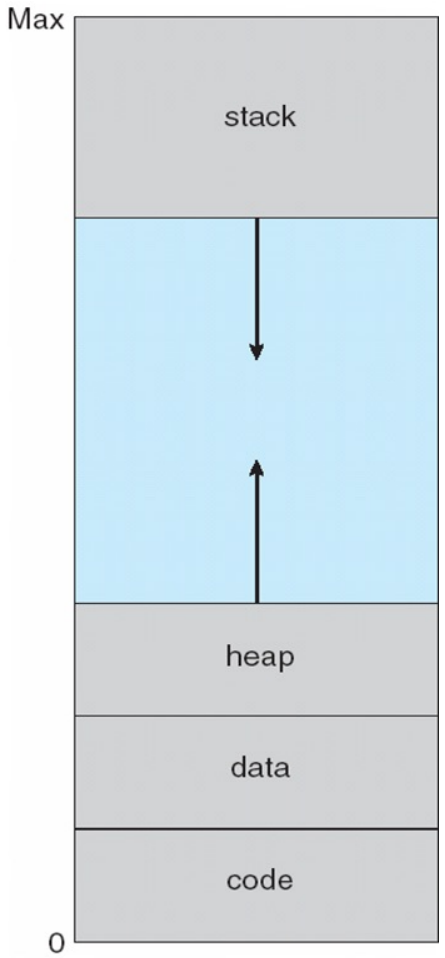
- **Virtual address space** – logical view of how process is stored in memory
  - Usually start at address 0, contiguous addresses until end of space
  - Meanwhile, physical memory organized in page frames
  - MMU must map logical to physical
- Virtual memory can be implemented via:
  - Demand paging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

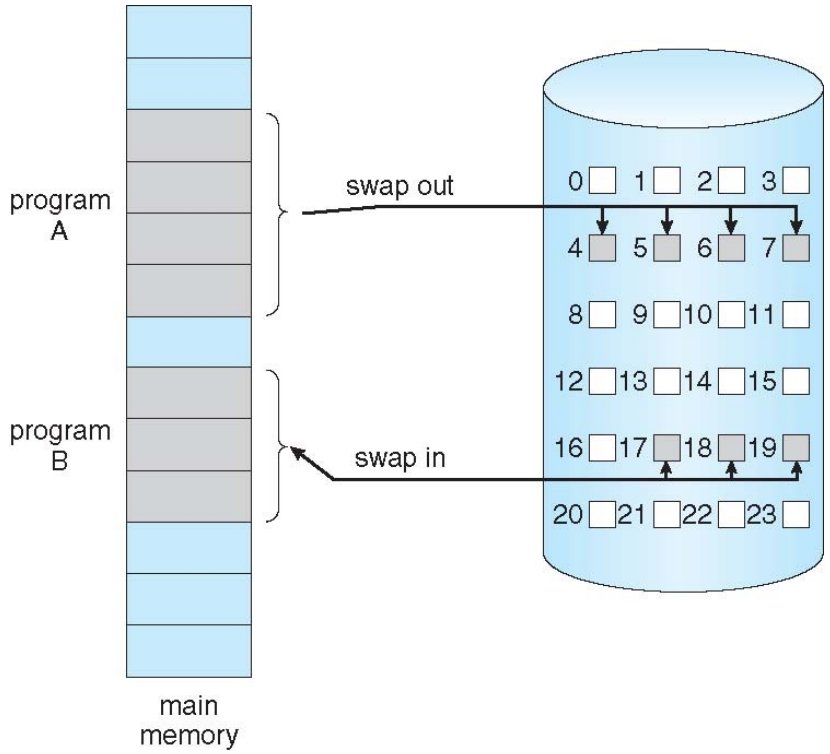


Demand Paging

- 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 ⇒ 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**

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)

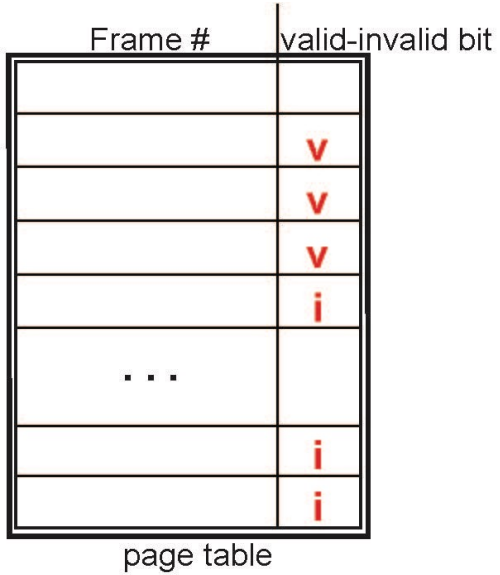


Basic Concepts

- 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

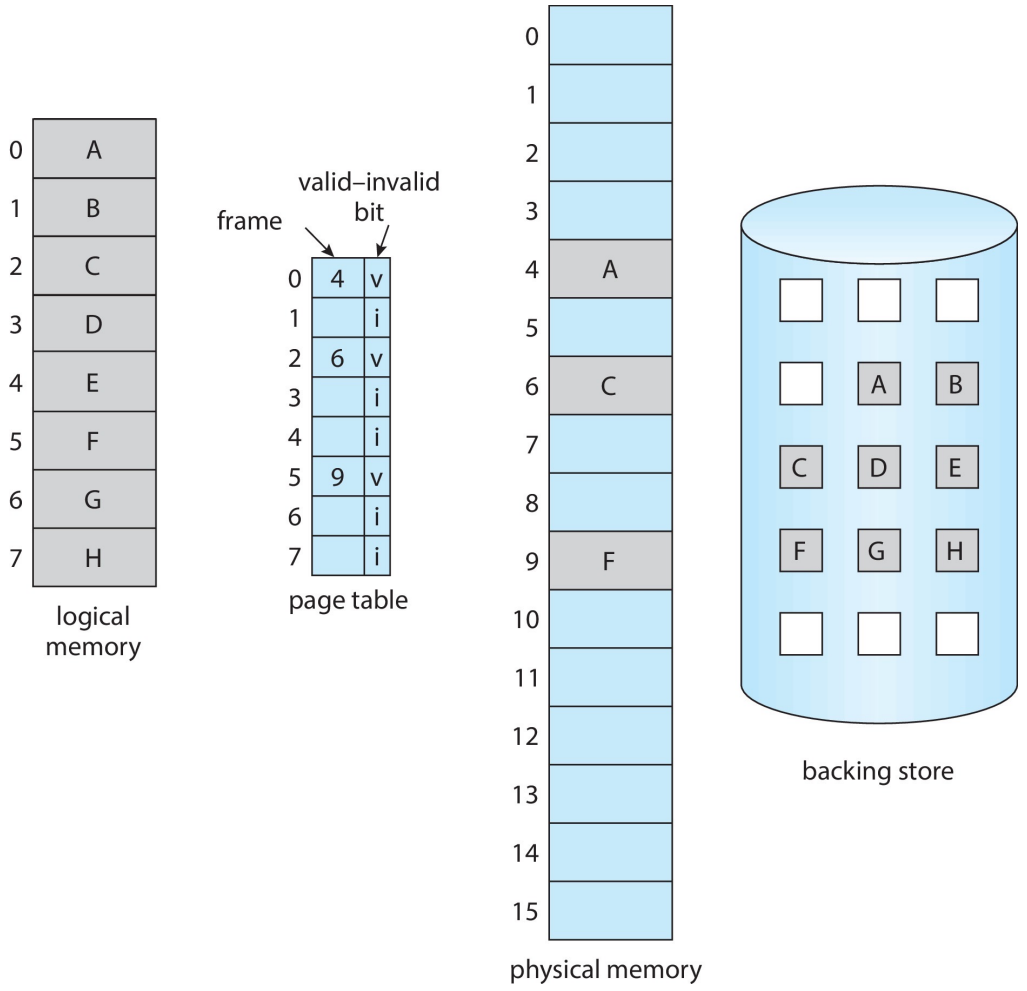
Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated (**v** ⇒ 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** ⇒ page fault

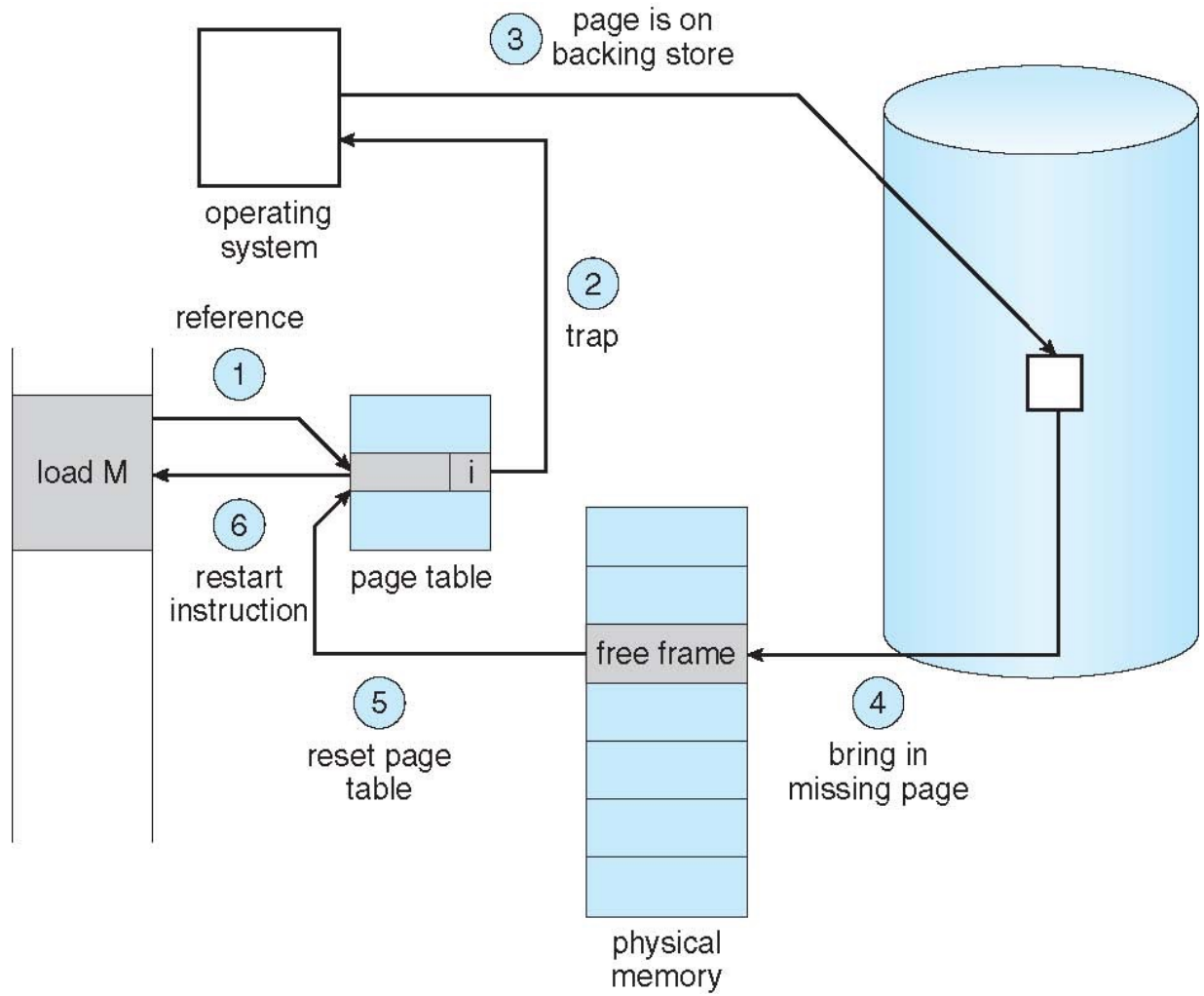
Page Table When Some Pages Are Not in Main Memory



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 ⇒ 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)

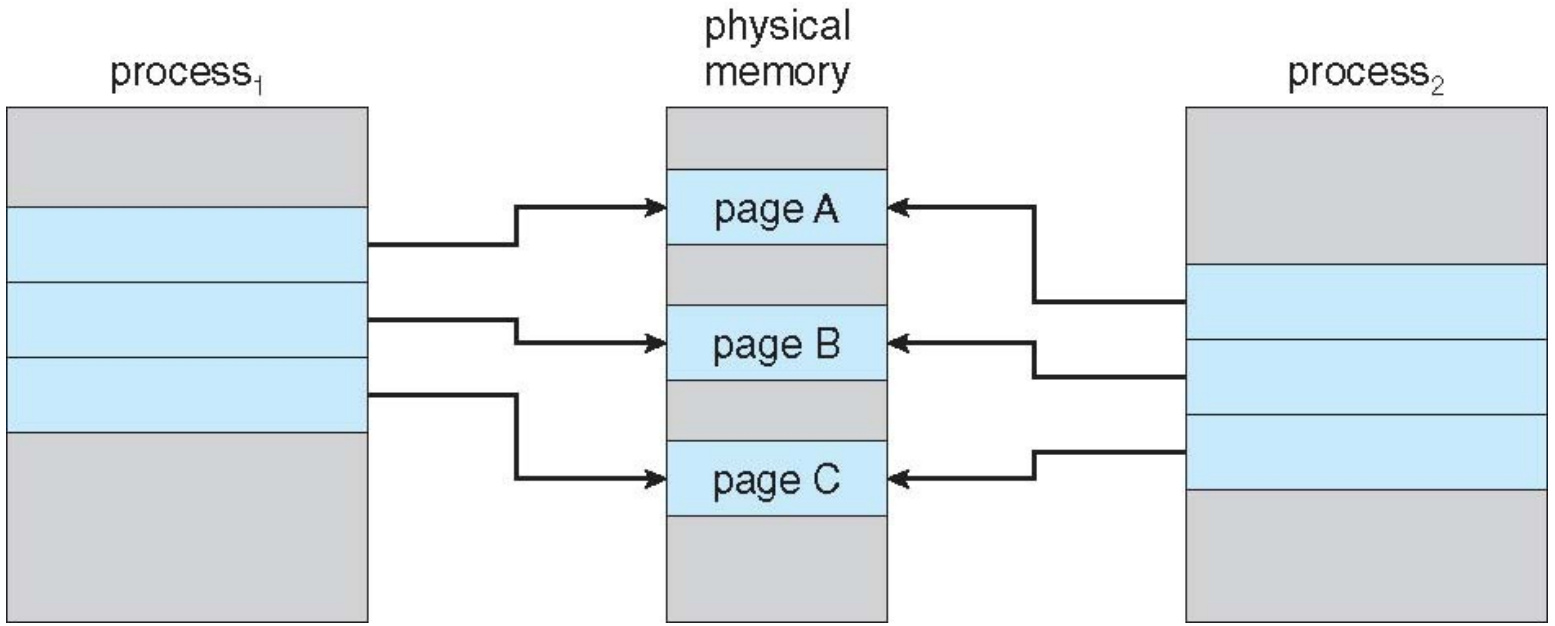
Demand Paging Optimizations

- 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)

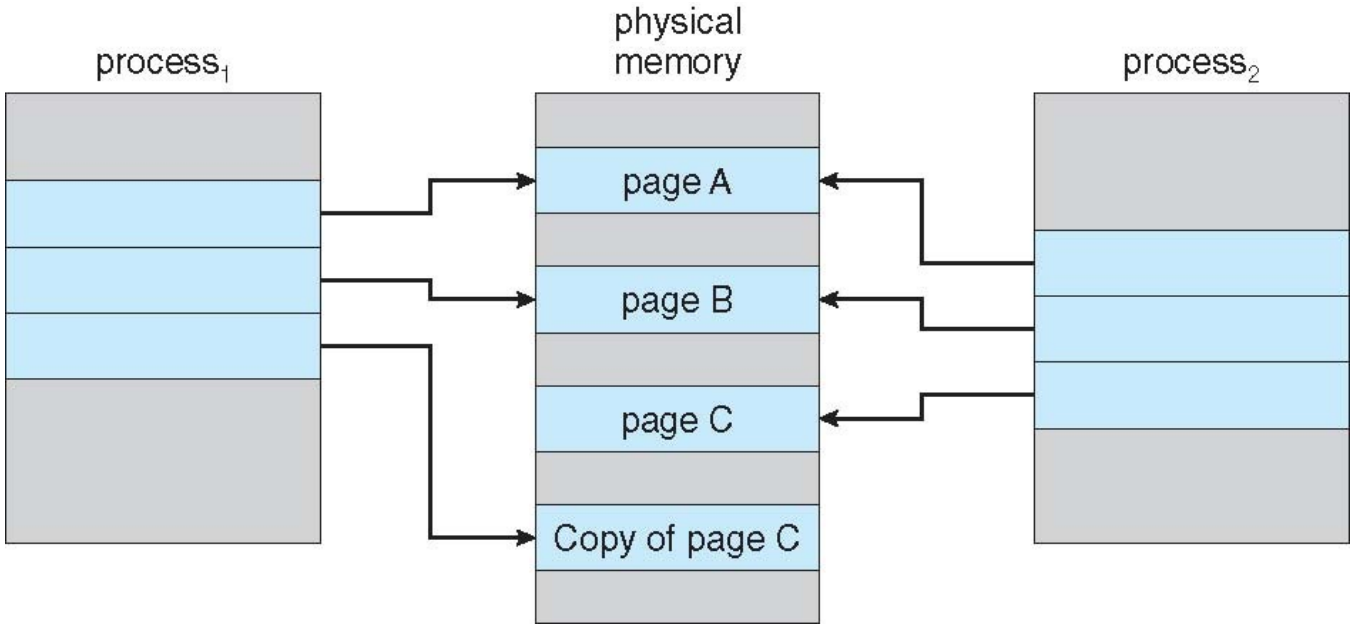
Copy-on-Write

- **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