TDIU11: Operating Systems

III. Memory Management

- Memory allocation, fragmentation, paging, segmentation

SGG9: chap. 8
SGG10: chap. 9

Ahmed Rezine, Linköping University

Background

- A program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are the only storage CPU can access directly
- Memory unit only sees a stream of addresses + read requests, or address + data and write requests

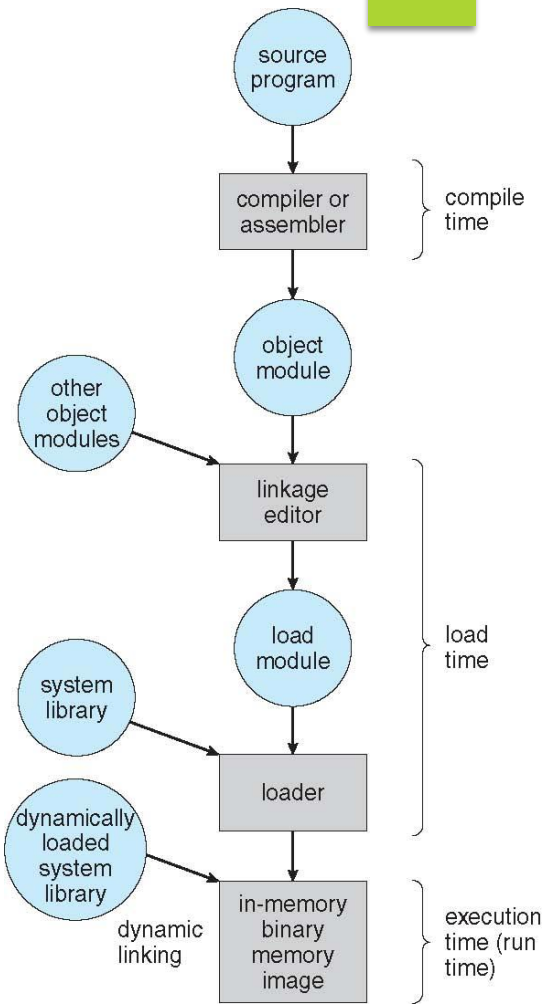
Background (cont.)

- Register access 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

Binding of Instructions and Data to Memory

Address binding of instructions and data to memory addresses:

- **Compile time**: If memory location known a priori; must recompile code if starting location changes
- **Load time**: Compiler must generate **relocatable code** if memory location. Final binding happens at load time. If starting address changes: need only reload the user code.
- **Execution time**: Used by most OS. Binding delayed until run time if the process can be moved during its execution from one memory segment to another

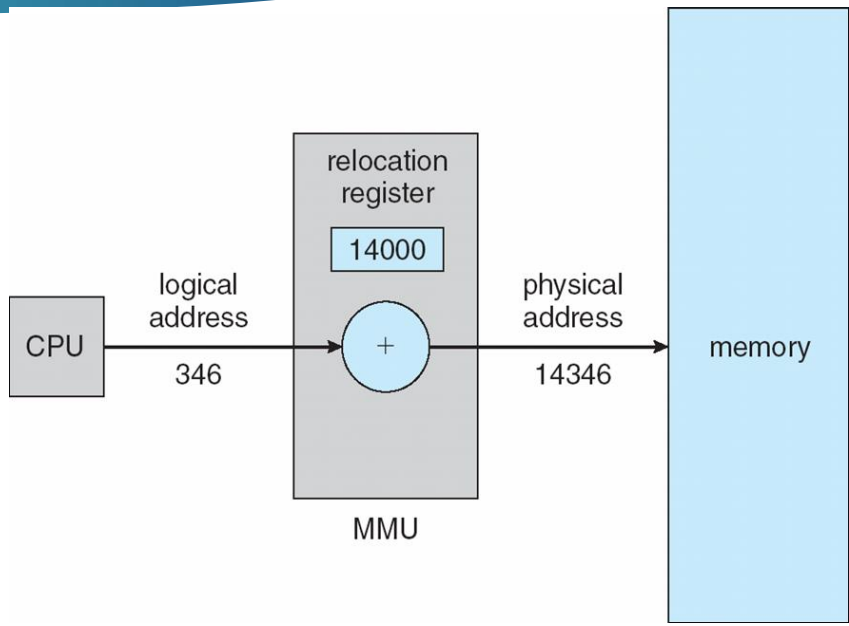


Logical vs. Physical Address Space

- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
  - **Logical (or virtual) address** – generated by the CPU
  - **Physical address** – address seen by the memory unit
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; but not in execution-time schemes

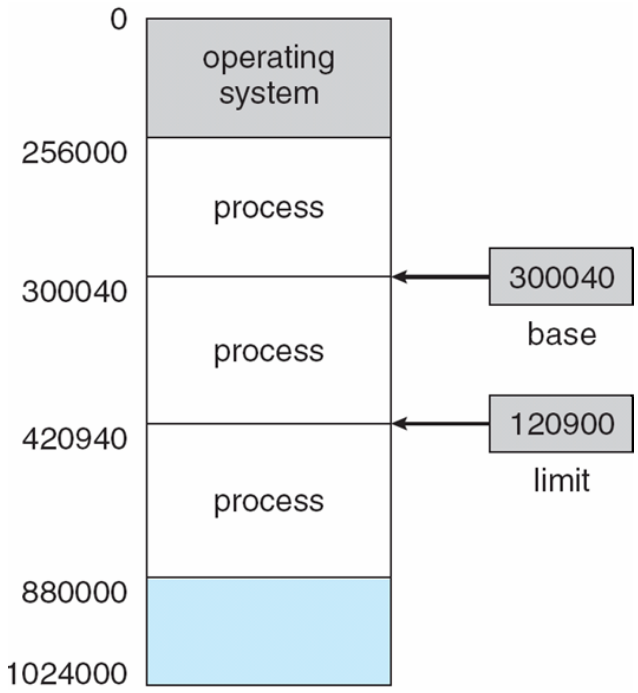
Memory-Management Unit (MMU)

- **Hardware device that at run time maps virtual to physical address**
- A very simple example: just add a base address specific to each process.
- The user program deals with *logical* addresses; it never sees the *real* physical addresses



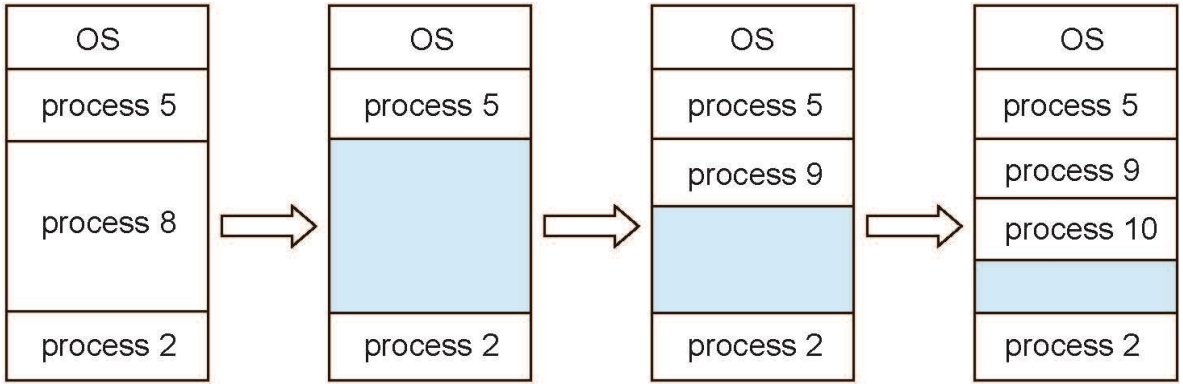
Contiguous Allocation

- Main memory must support both OS and user processes
- 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



Multiple-partition allocation

- Degree of multiprogramming limited by number of partitions
- **Variable-partition** sizes for efficiency (sized to 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



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

Simulations suggest first-fit and best-fit better than worst-fit in terms of speed and storage utilization.

Fragmentation

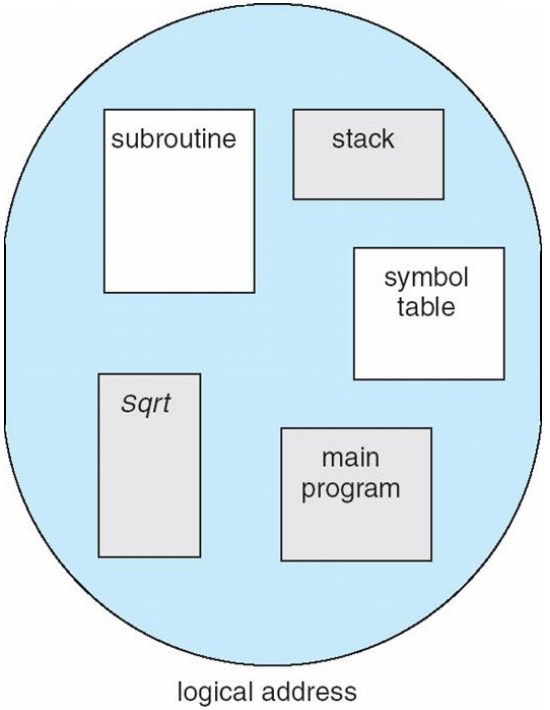
External Fragmentation – total memory space exists to satisfy a request, but it is not contiguous

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

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 at execution time
- I/O problem
  - 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

Segmentation



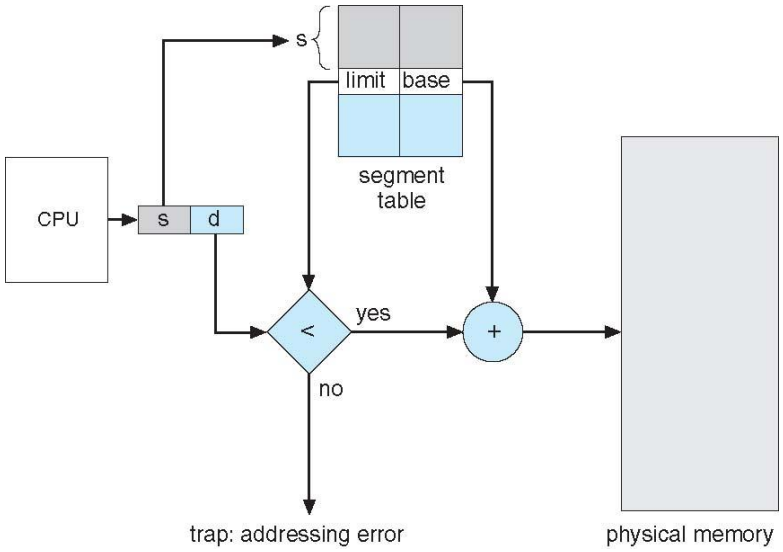
- Supports user view of memory
- A program is a collection of logical units such as:
  - main program, procedures and methods, Objects, local variables and global variables, Stack, symbol table, arrays

Segmentation Architecture

- Logical address consists of a pair: <segment-number, offset>,
- **Segment table** – each table entry has:
  - **base** –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)** number of segments used by a program;

Segmentation Architecture (Cont.)

- Protection: associate to each segment table entry:
  - validation bit = 0 $\Rightarrow$ illegal segment
  - read/write/execute privileges
- Protection bits associated to segments;
- Code sharing occurs at segment level
- Segments vary in length, memory allocation is a dynamic storage-allocation problem



Paging

- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available. Unlike segmentation, paging:
  - Avoids external fragmentation
  - Avoids problem of varying sized memory chunks
- Divide physical memory into fixed-sized blocks called **frames**
  - Size is power of 2, e.g., 512 bytes 16 Mbytes
- Divide logical memory into blocks of same size called **pages**

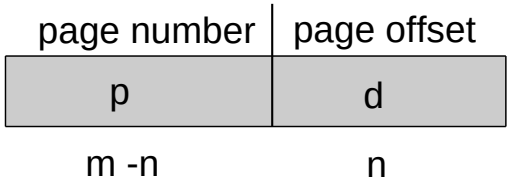
Paging (cont.)

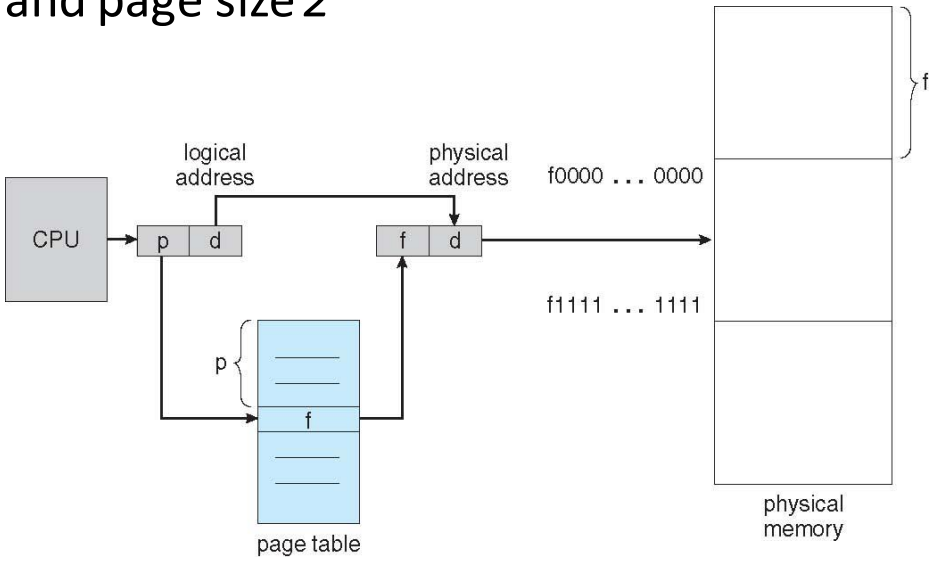
- Divide physical memory into fixed-sized blocks called **frames**
- Divide logical memory into blocks of same size called **pages**
- Keep track of all free frames
- To run a program of size ***N*** pages, need to find ***N*** free frames and load program
- Set up a **page table** to translate logical to physical addresses

- Still have Internal fragmentation

Address Translation Scheme

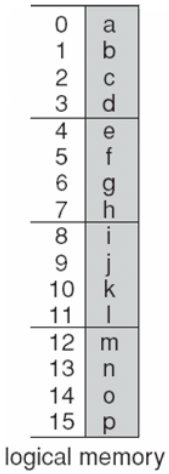
- Address generated by CPU is divided into:
  - **Page number** (***p***) – used as an index into a **page table** which contains base address of each page in physical memory
  - **Page offset** (***d***) – combined with base address to define the physical memory address that is sent to the memory unit
- For given logical address space 2^m and page size 2^n



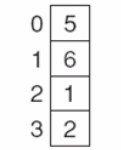


Tiny Paging Example

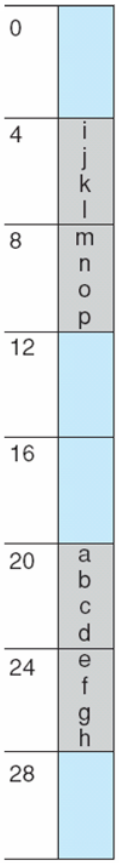
- Page size $2^n = 2^2 = 4$ bytes
- Logical address space:
  - $2^m = 2^4 = 16$ bytes
- 32-byte physical memory (8 frames)
- Logical address 0 is page 0 offset 0
  - Physical 5x4+0=20
- Logical 5 is page 1 offset 1
  - Physical 6x4+1=25



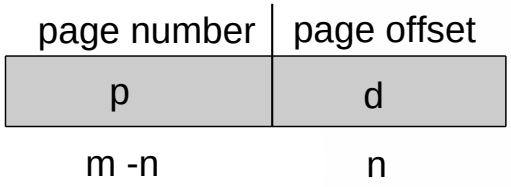
logical memory



page table



physical memory



Paging (Cont.)

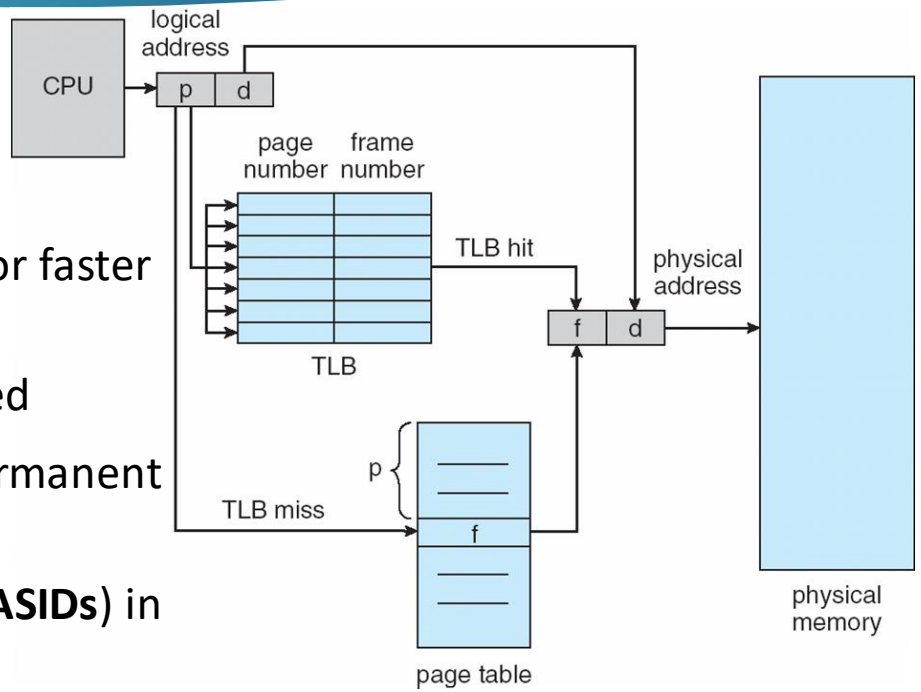
- Calculating internal fragmentation
  - Worst case fragmentation (internal) = 1 frame – 1 byte
  - On average fragmentation = 1 / 2 frame size
  - So small frame sizes desirable?
  - But each page table entry takes memory to track
  - Page sizes growing over time
- Process view and physical memory now very different
- By implementation process can only access its own memory

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 using a special fast-lookup hardware associative cache called **translation look-aside buffers** (**TLBs**)

Implementation of Page Table (Cont.)

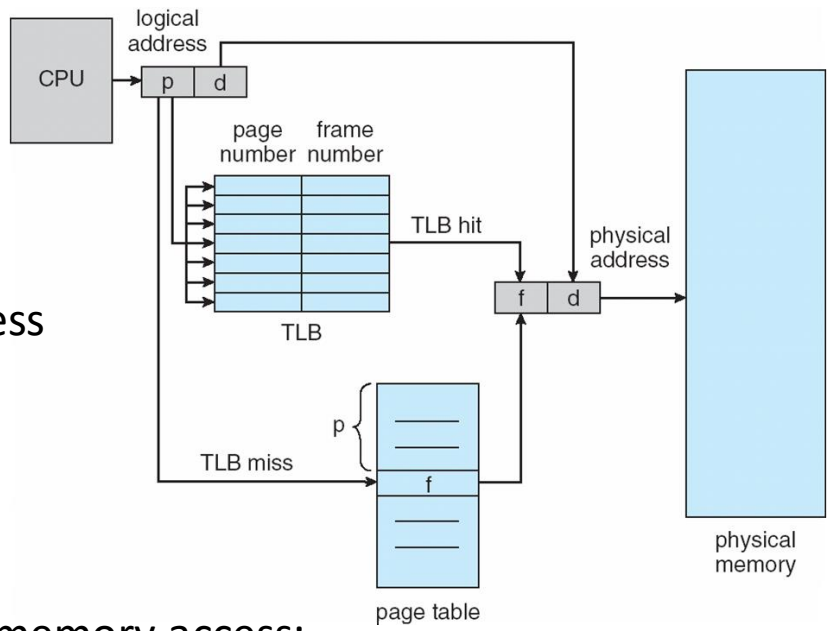
- TLBs typically small (64 to 1,024 entries)
- On a TLB miss, value is loaded into the TLB for faster access next time
  - Replacement policies must be considered
  - Some entries can be **wired down** for permanent fast access
- Some TLBs store **address-space identifiers** (**ASIDs**) in each TLB entry: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



Effective Access Time

- Associative Lookup = ε time unit. Can be < 10% of memory access time

- Hit ratio = α. Percentage of times a page number is found in the associative registers



- If α = hit ration, ε = TLB search, t = memory access
- **Effective Access Time** (**EAT**):

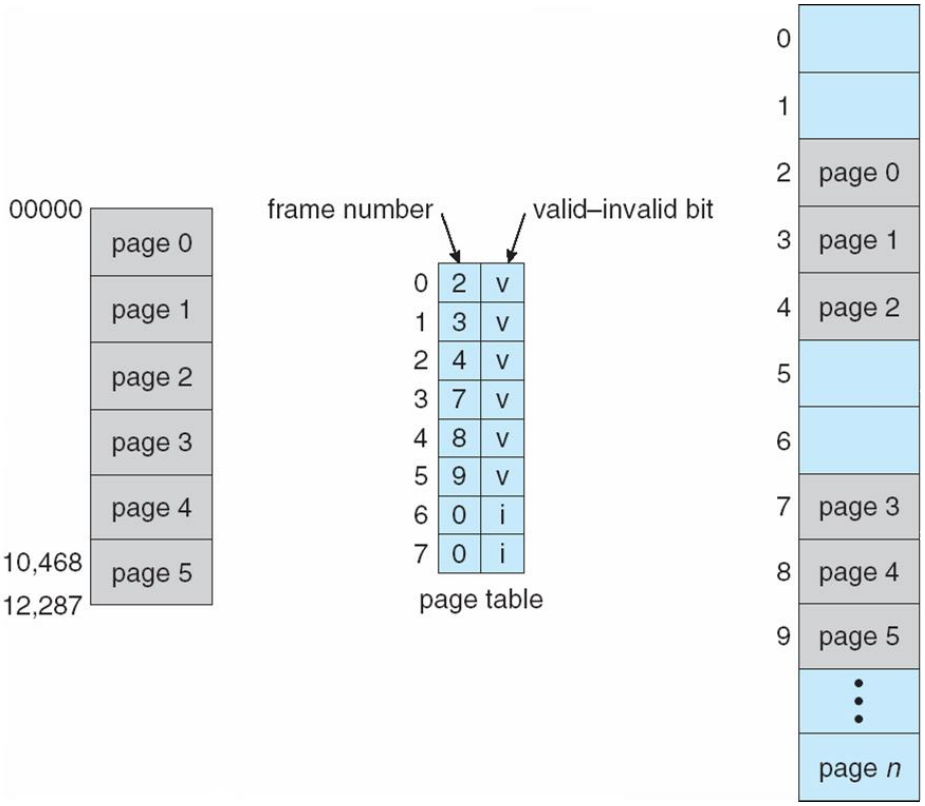
  $EAT = (t + \varepsilon)\,\alpha + (2t + \varepsilon)(1 - \alpha)$

  $= 2t + \varepsilon - \alpha\, t$

- If α = 80%, discarding ε = TLB search, 100ns for memory access:
  - EAT = 2 x 100 - 0.80 x 100 = 120ns
- Consider more realistic hit ratio -> α = 99%, 100ns for memory access.
  - EAT = 0.99 x 100 + 0.01 x 200 = 101ns

Memory Protection

- Associate protection bits to each frame to indicate if read-only or read-write access is allowed. More bits to indicate page execute-only, …
- **Valid-invalid** bit attached to each entry in the page table:
  - "valid": associated page is in the process' logical address space
  - "invalid": 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



Structure of the Page Table

Memory structures for paging can get huge using straight-forward methods

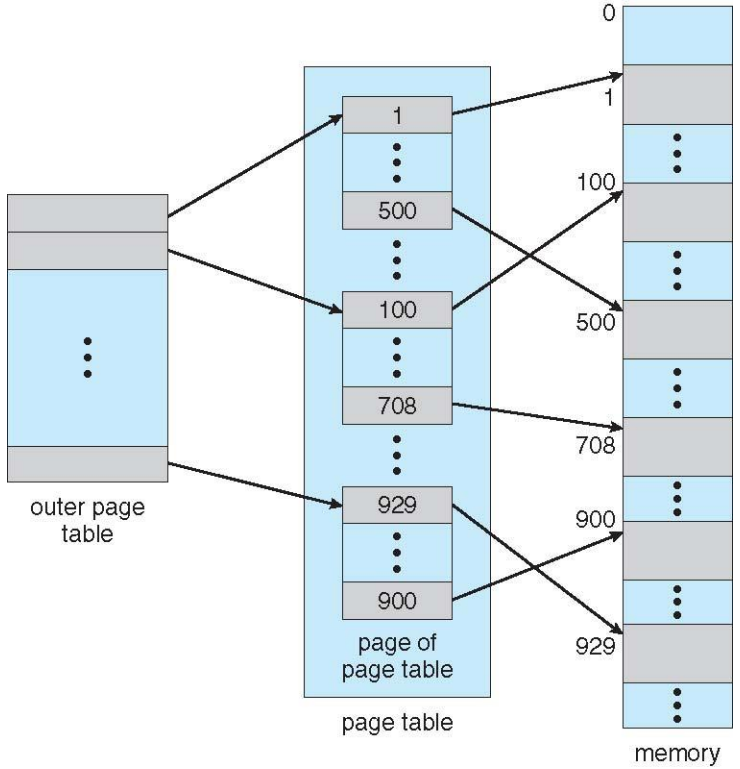
- Consider a 32-bit logical address space
- Page size of 4 KB (2^{12})
- Page table would have 1 million entries (2^{32} / 2^{12})
- If each entry is 4 bytes; we get 4 MB of memory for page table alone
  - That amount of memory used to cost a lot
  - Don't want to allocate that contiguously in main memory (always try to fit a page table in a page)

Solutions:

- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

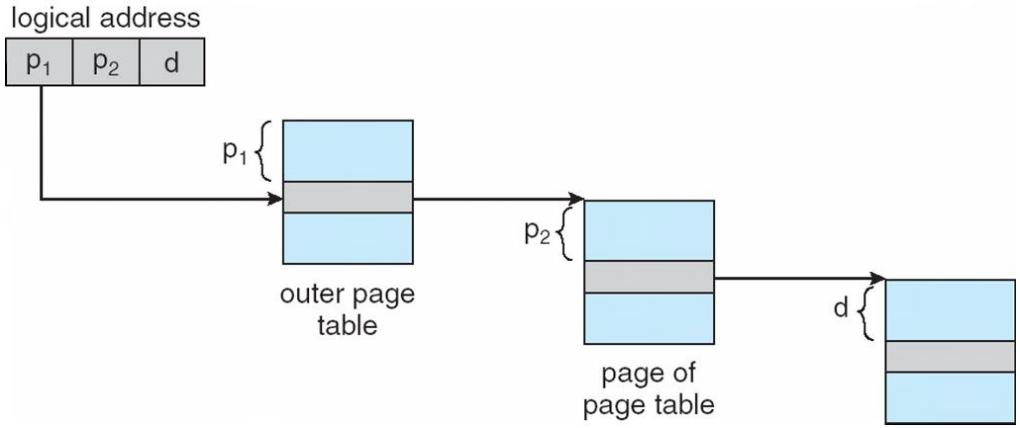
Hierarchical Page Tables

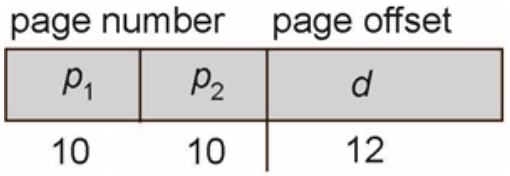
- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
- We then page the page table



Two-Level Paging Example

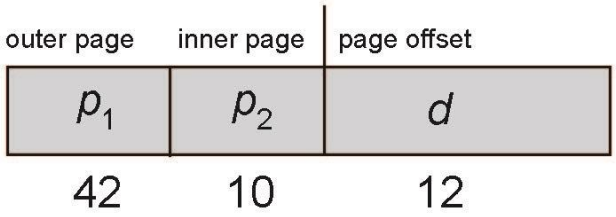
- A logical address (on 32-bit machine with 4KiB page size) has:
  - a page number consisting of 20 bits
  - a page offset consisting of 12 bits
- Since the page table is paged, the page number is further divided into p_1 and p_2.
- Where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table (**forward-mapped page table)**





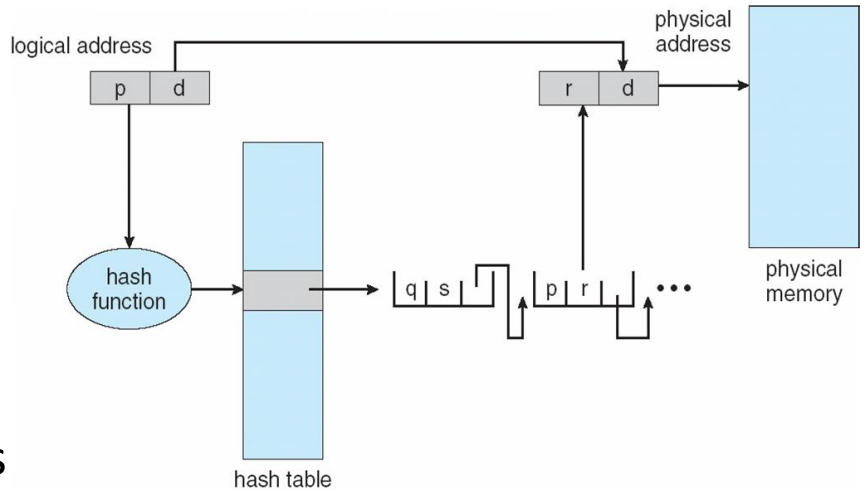
64-bit Logical Address Space

- If page size is 4 KiB (2^{12})
  - Then page table has 2^{52} entries
  - If two level scheme, inner page tables could be 2^{10} 4-byte entries
  - 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
- Even two-level paging scheme not enough



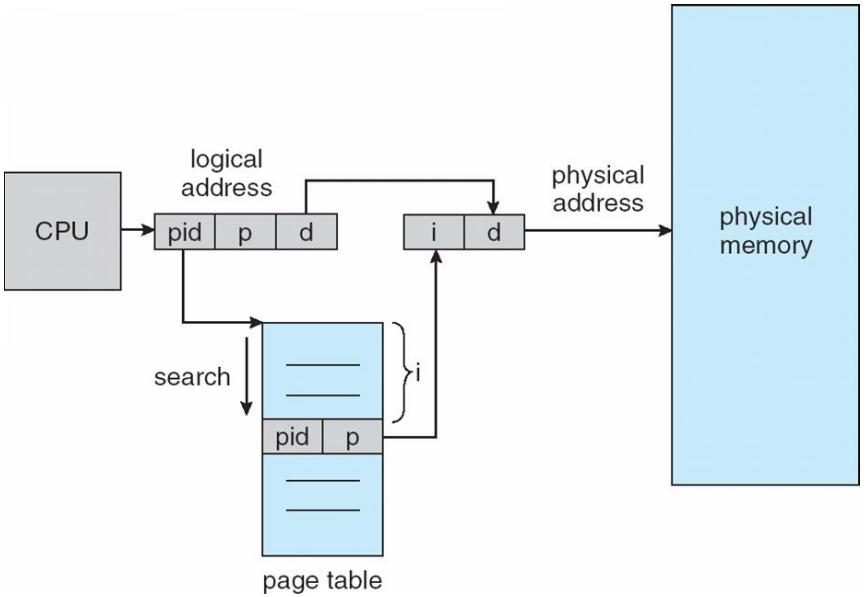
Hashed Page Tables

- Common in address spaces > 32 bits
- The virtual page number is hashed into a page table containing 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



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
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
- shared memory more difficult to implement

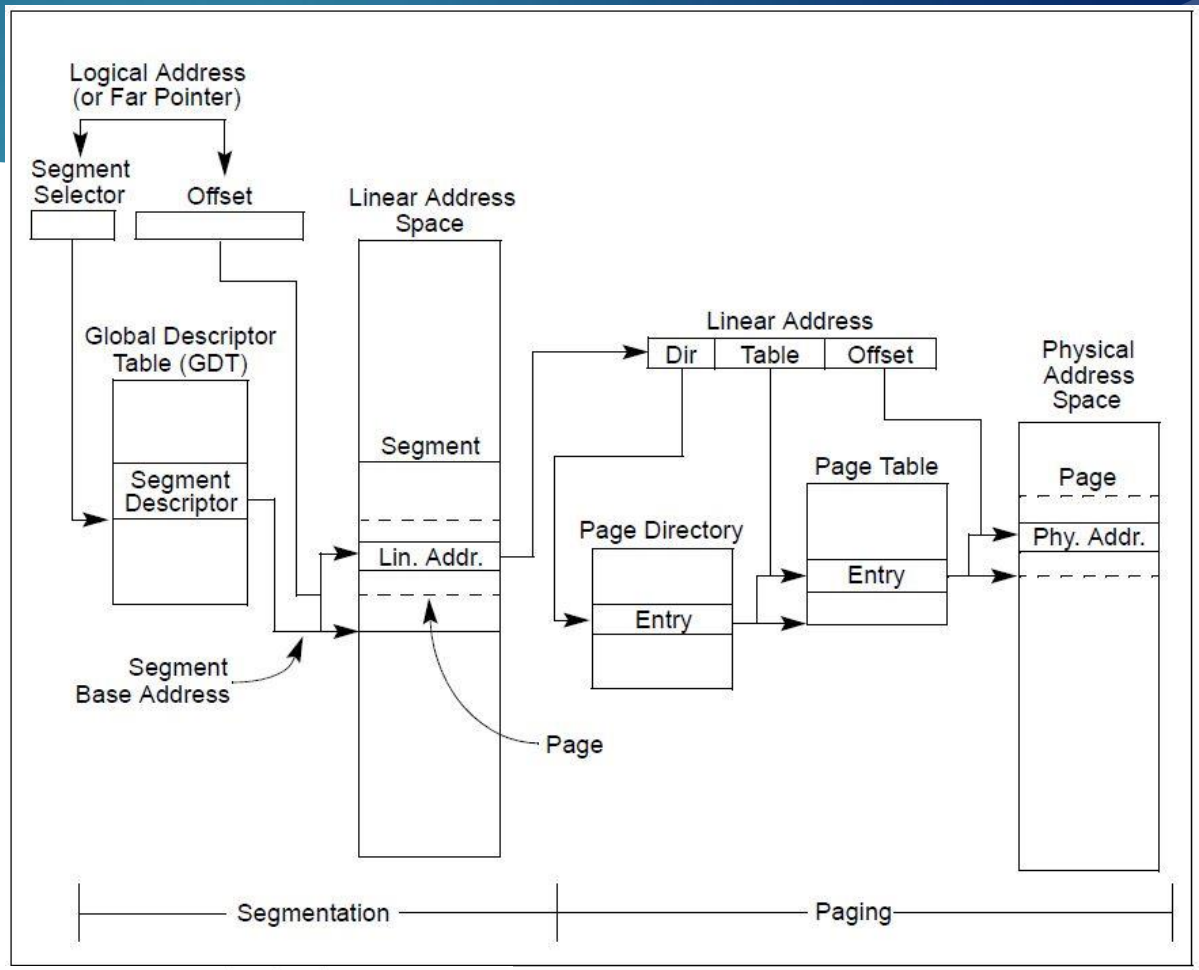


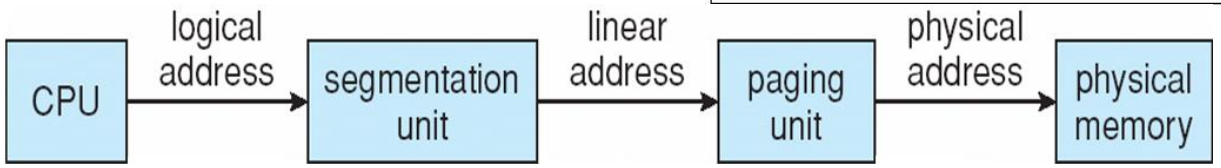
Structure of the Page Table

- Memory structures for paging can get huge using straight-forward methods
- Solutions:
  - Hierarchical Paging
  - Hashed Page Tables
  - Inverted Page Tables

Combining Segmentation and Paging (IA32)

From Intel® 64 and IA-32 Architectures Software Developer's Manual



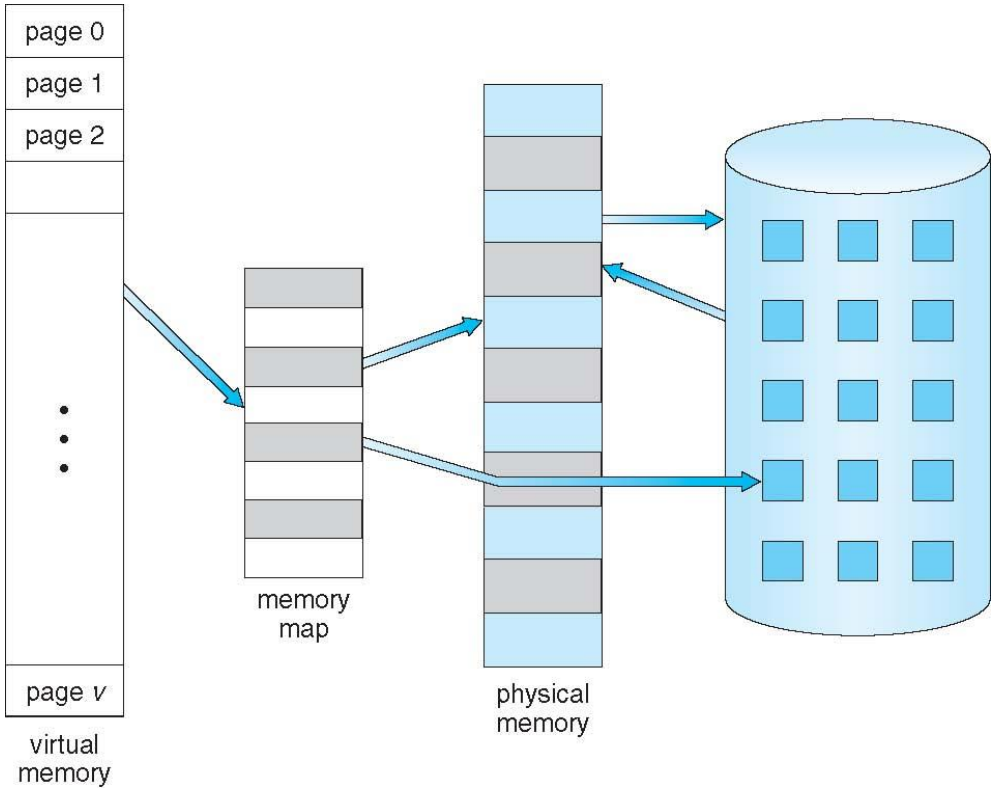


Virtual Memory: Background

- Entire program rarely used: error code, unusual routines, large data structures
- Entire program code not needed at same time. Execute partially-loaded 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, no increase in response time or turnaround time
- Less I/O needed to load or swap programs into memory. Each user program runs faster

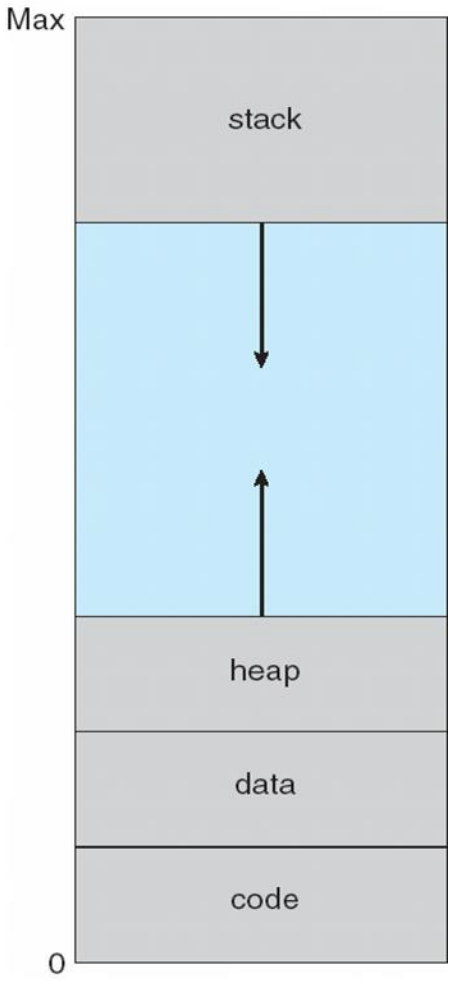
Virtual memory can be implemented via: demand paging or demand segmentation

Virtual Memory That is Larger than Physical Memory



Virtual-address Space

- Usually, stack starts at Max logical address and grows “down” while heap grows “up”.
- 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



Shared Library Using Virtual Memory