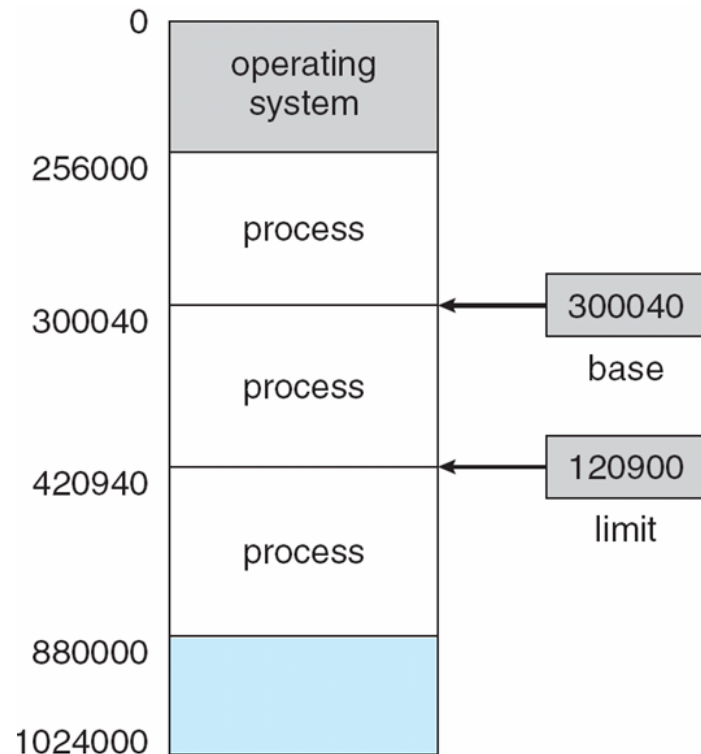


Background

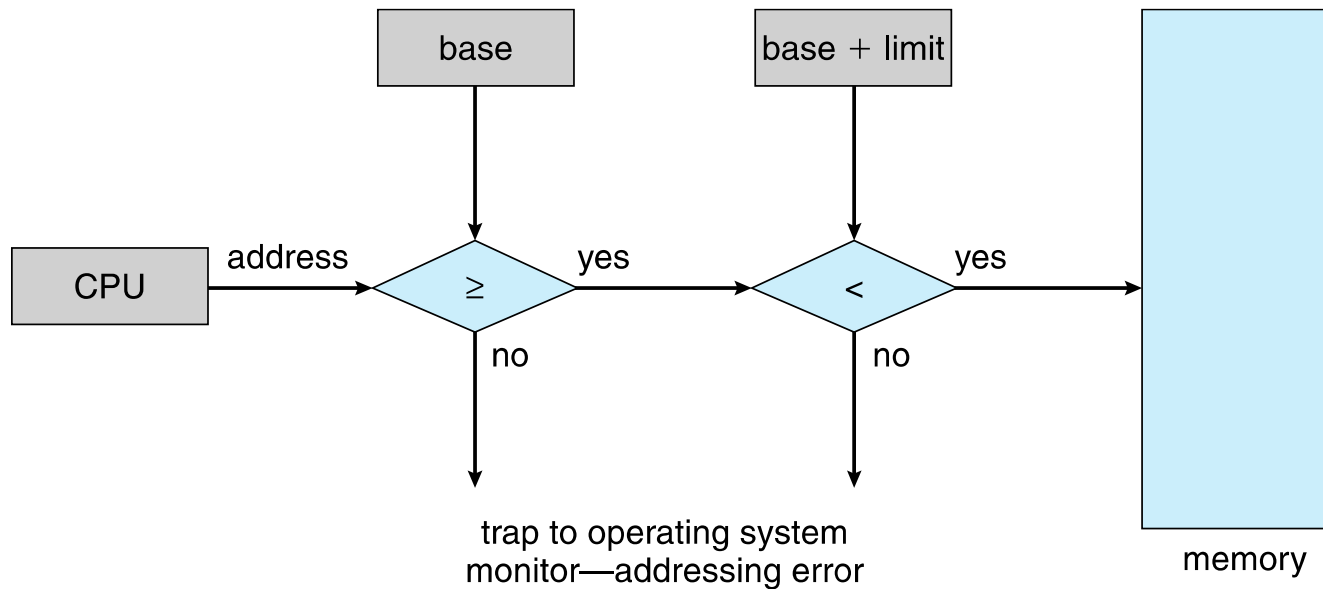
- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Memory unit only sees a stream of addresses + read requests, or address + data and write requests
- Register access in one CPU clock (or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

Base and Limit Registers

- A pair of **base (initial Physical Location)** and **limit registers (Length of Process)** define the logical address space
- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user



Hardware Address Protection



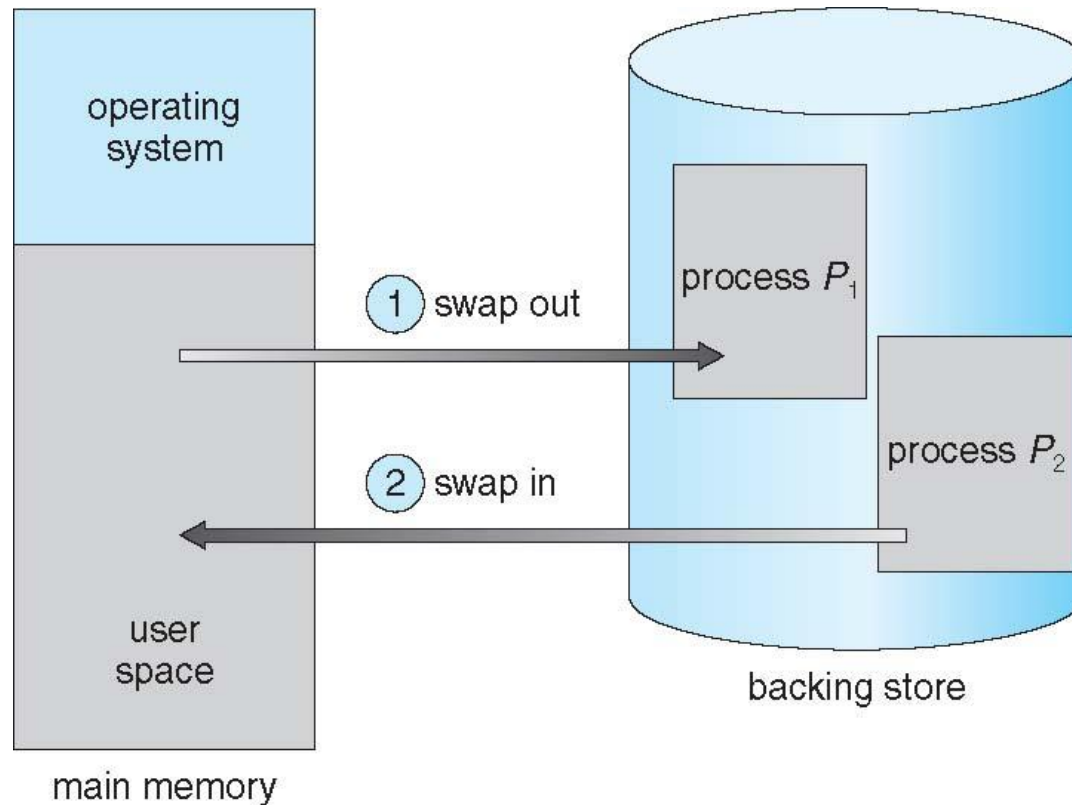
Logical vs. Physical Address Space

- 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

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

Schematic View of Swapping



Context Switch Time including Swapping

- If next processes 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 2 sec
 - 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()`

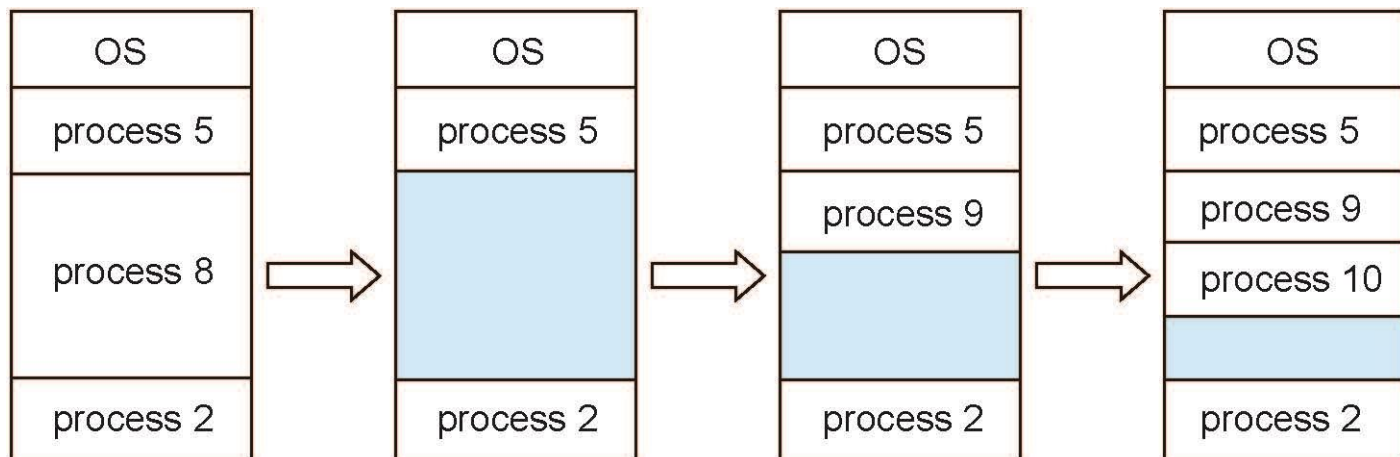
Contiguous Allocation

- 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

Multiple-partition allocation

- 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

- Given five memory partitions of 100 KB, 500 KB, 200 KB, 300 KB, and 600 KB (ill order), how would the first-fit, best-fit, and worst-fit algorithms place processes of 212 KB, 417 KB, 112 KB, and 426 KB (in order)? Which algorithm makes the most efficient use of memory?

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**

Process	Process Size	Status	Available Memory Size
			40KB

1

WWW.T4Tutorials.com

Process	Process Size	Status	Available Memory Size
P1	15KB	Used	15KB
			25KB

2

Process	Process Size	Status	Available Memory Size
P1	15KB	Used	15KB
P2	5KB	Used	5KB
			20 KB

3

Process	Process Size	Status	Available Memory Size
P1	15KB	Used	15KB
P2	5KB	Used	5KB
P3	20KB	Used	20KB

4

Process	Process Size	Status	Available Memory Size
P1 Terminated 15 KB is free Now			15KB
P2	5KB	Used	5KB
P3	20KB	Used	20KB

5

Process	Process Size	Status	Available Memory Size
P1 Terminated 15 KB is free Now			15KB
P2	5KB	Used	5KB
P3 Terminated 20 KB is free Now			20KB

6

Note here that we dynamically allocate the memory to each process to exactly create the fragment of process size but still our problem is not solved.
For example if we want to load a process of 35 KB then it can't be loaded because 35 KB is free but this 35 KB is not contiguous(not located together).

External fragmentation

Figure: external fragmentation

Process	Process Size	Status	Available Memory Size
P1	10 KB	Used 10KB	15KB
Free 5KB			
P2	15KB	Used 15KB	15KB
P3	5KB	Used 5KB	15KB
Free 10KB			

Internal Fragmentation 5KB wasted

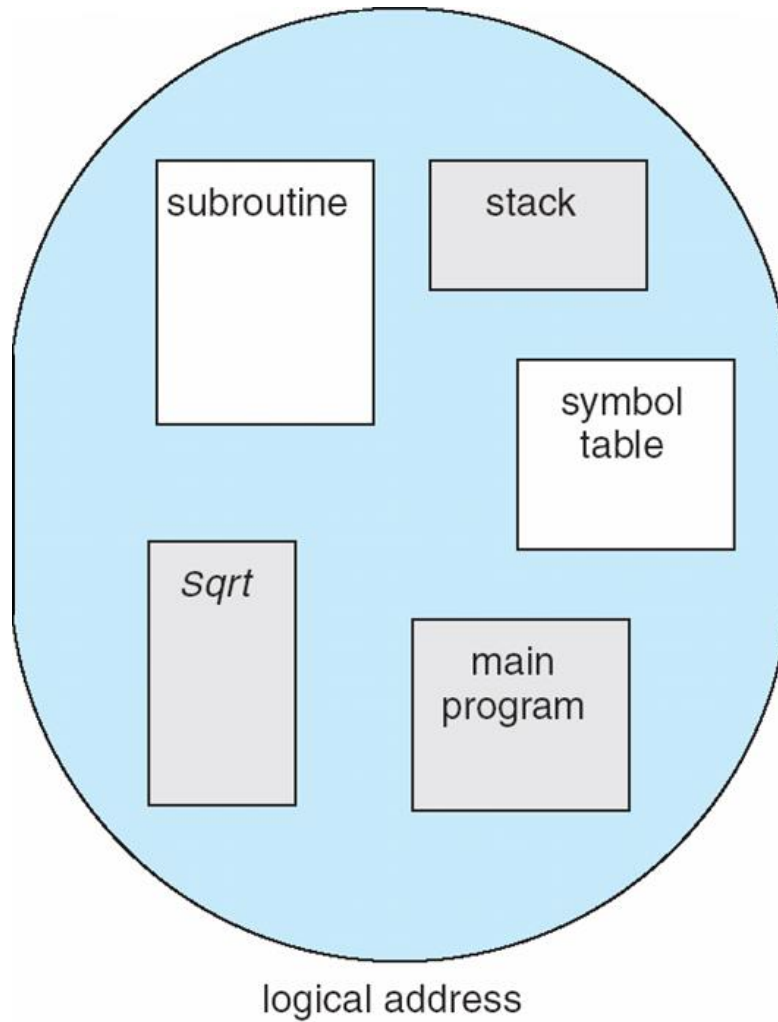
Internal Fragmentation 10KB wasted

Figure: internal fragmentation memory management operating systems os

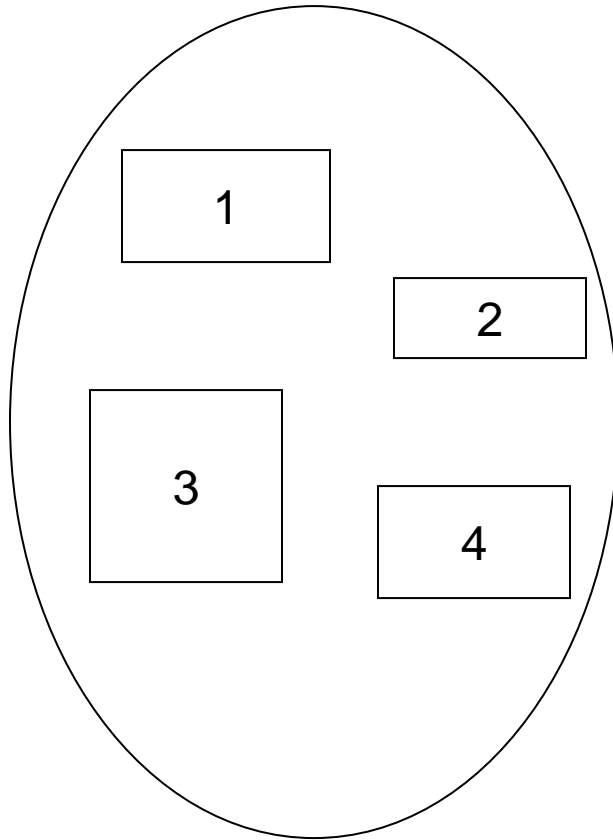
Segmentation

- 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, global variables
 - common block
 - stack
 - symbol table
 - arrays

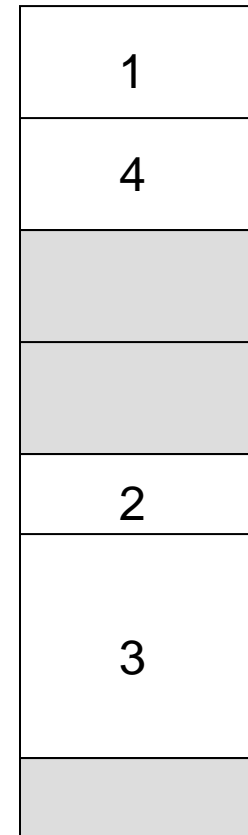
User's View of a Program



Logical View of Segmentation

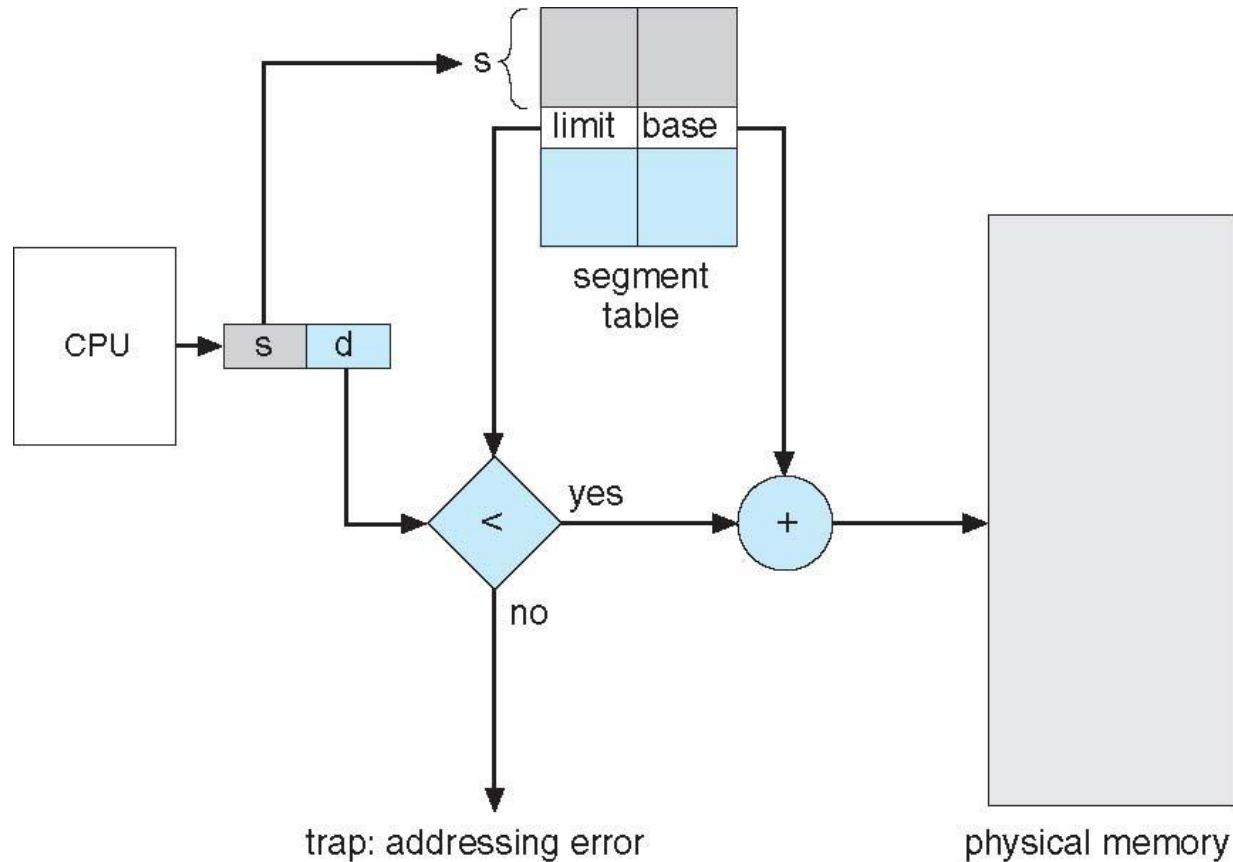


user space



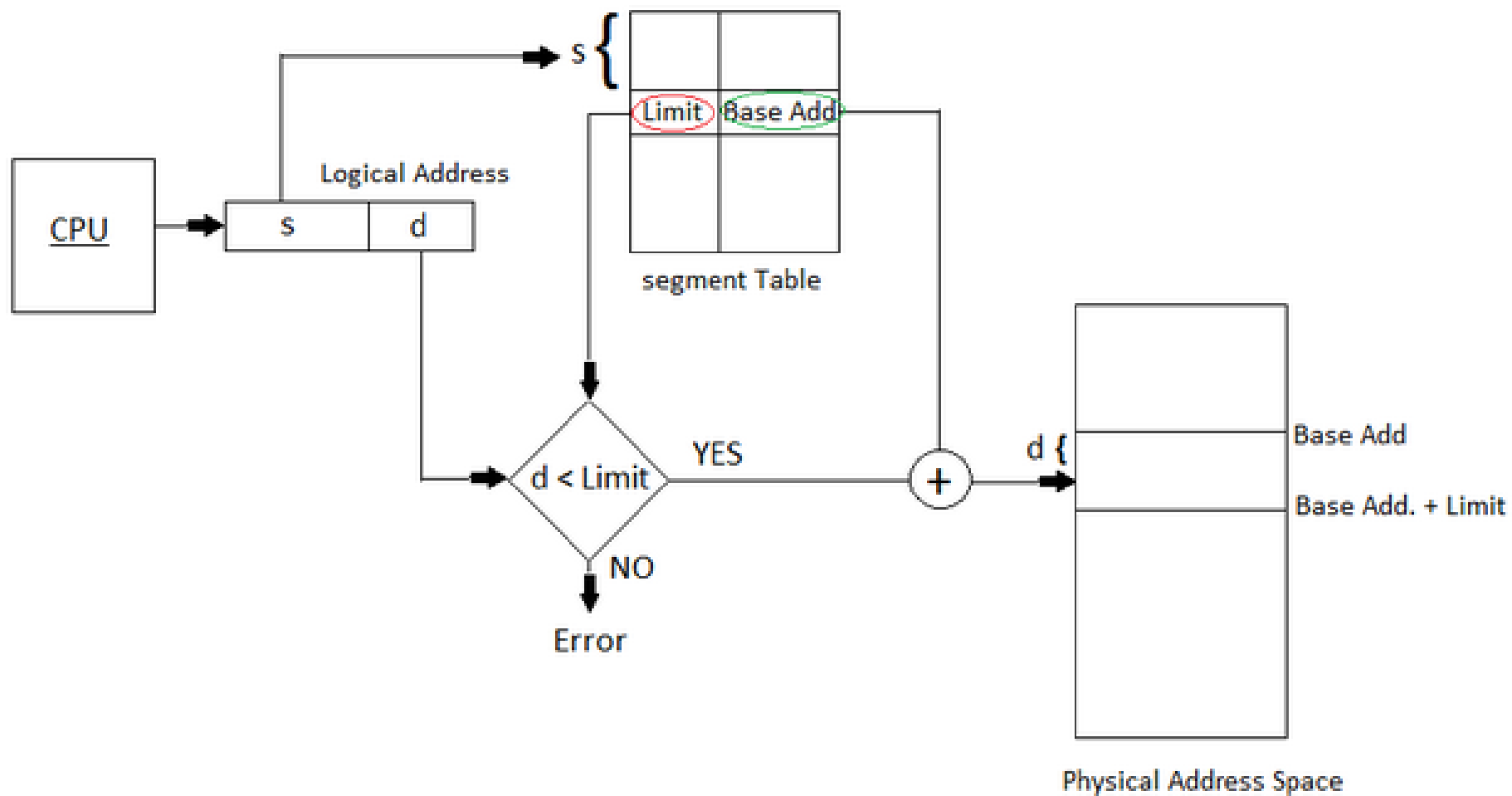
physical memory space

Segmentation Hardware

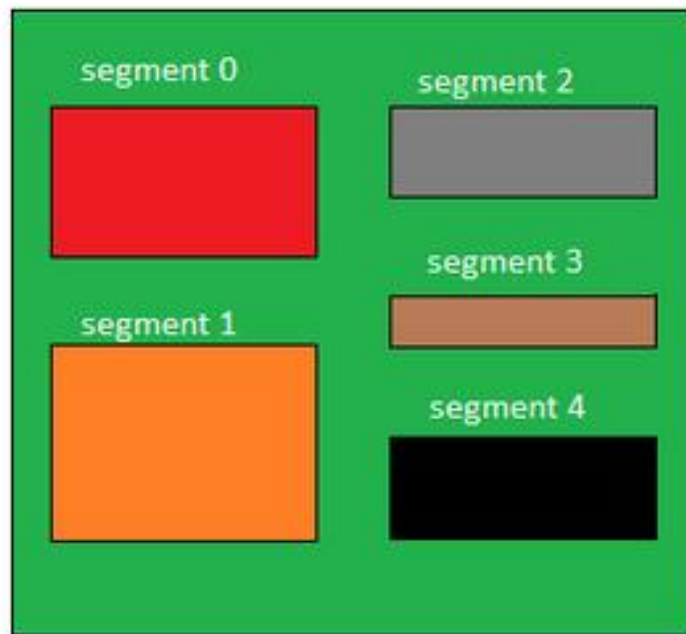


- **Segment number (s):** Number of bits required to represent the segment.

- **Segment offset (d):** Number of bits required to represent the size of the segment.



Logical View of Segmentation

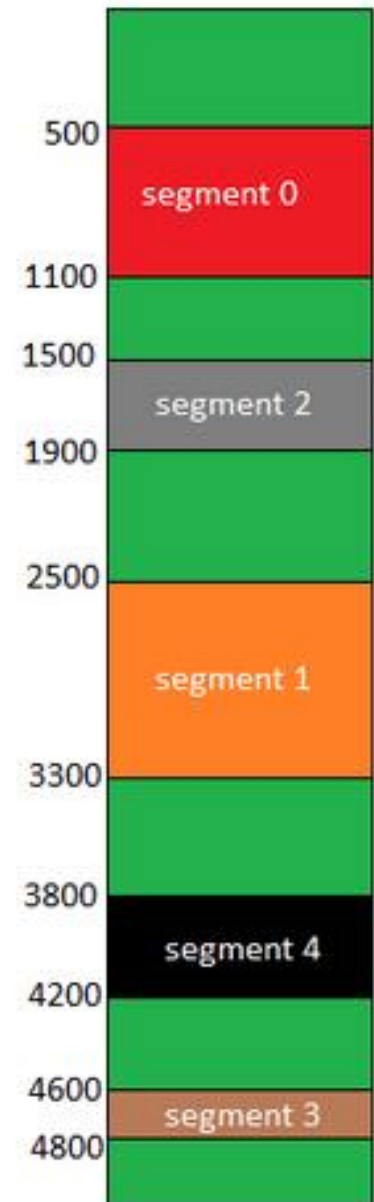


Logical Address Space

Segment Number

	base address	Limit
0	500	600
1	2500	800
2	1500	400
3	4600	200
4	3800	400

Segment Table



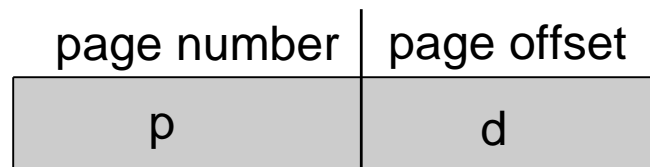
Physical Address Space

Paging

- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
 - 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
- 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 offset** (d) – combined with base address to define the physical memory address that is sent to the memory unit

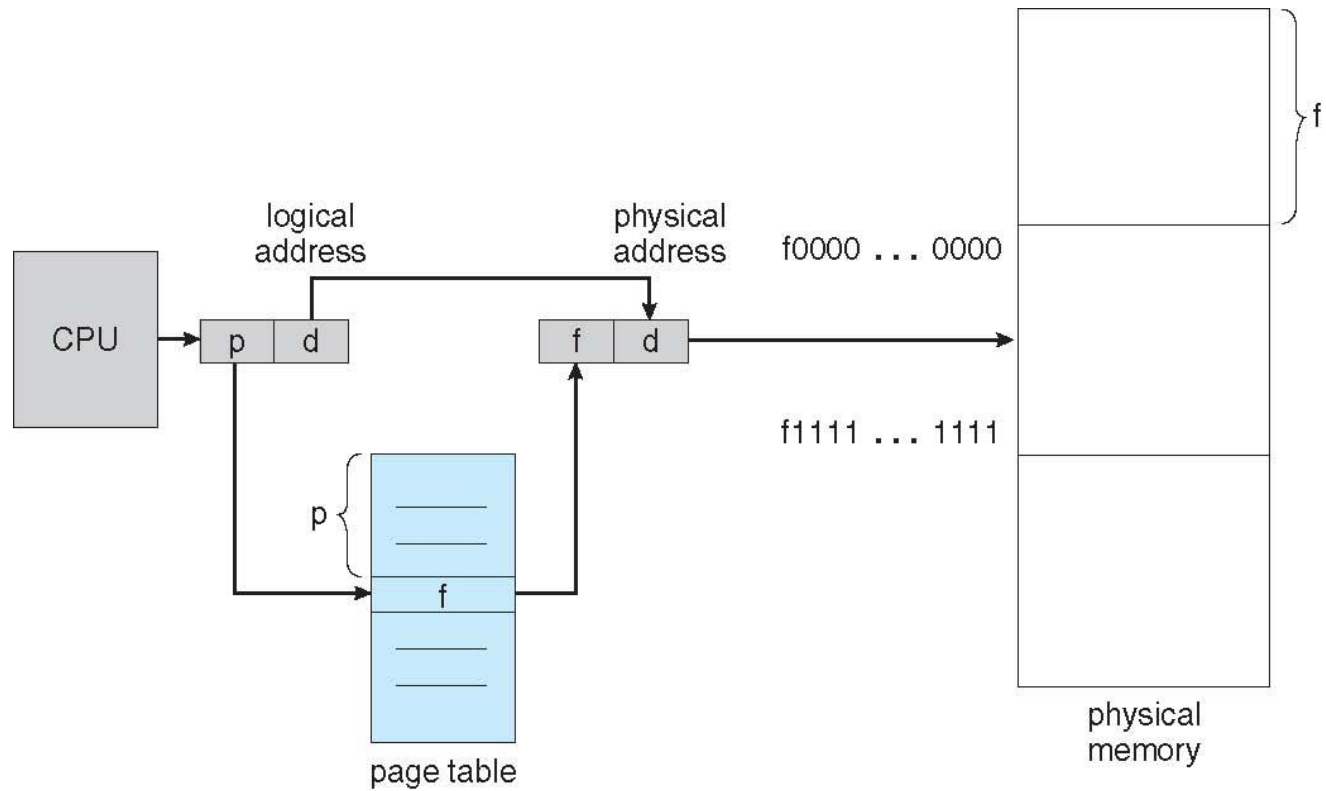


$m - n$

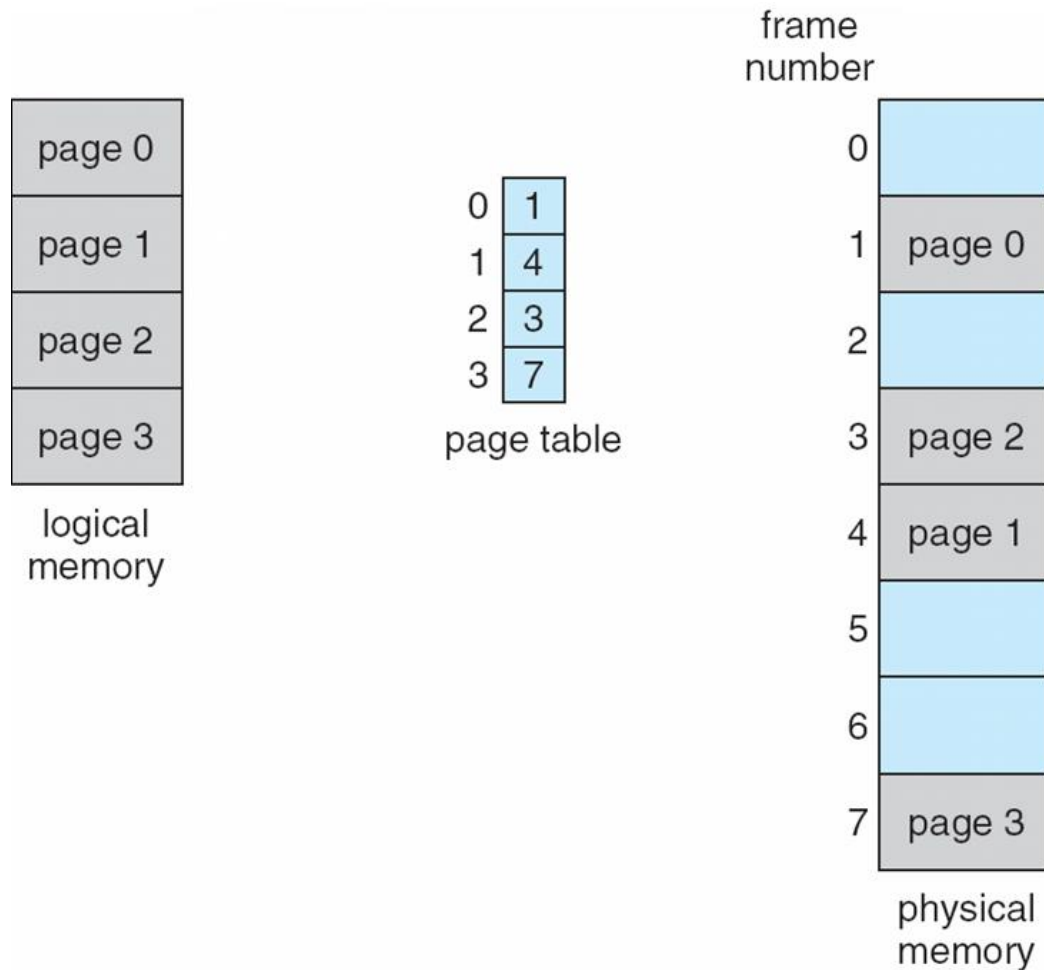
n

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

Paging Hardware



Paging Model of Logical and Physical 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 by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers (TLBs)**

Implementation of Page Table (Cont.)

- 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

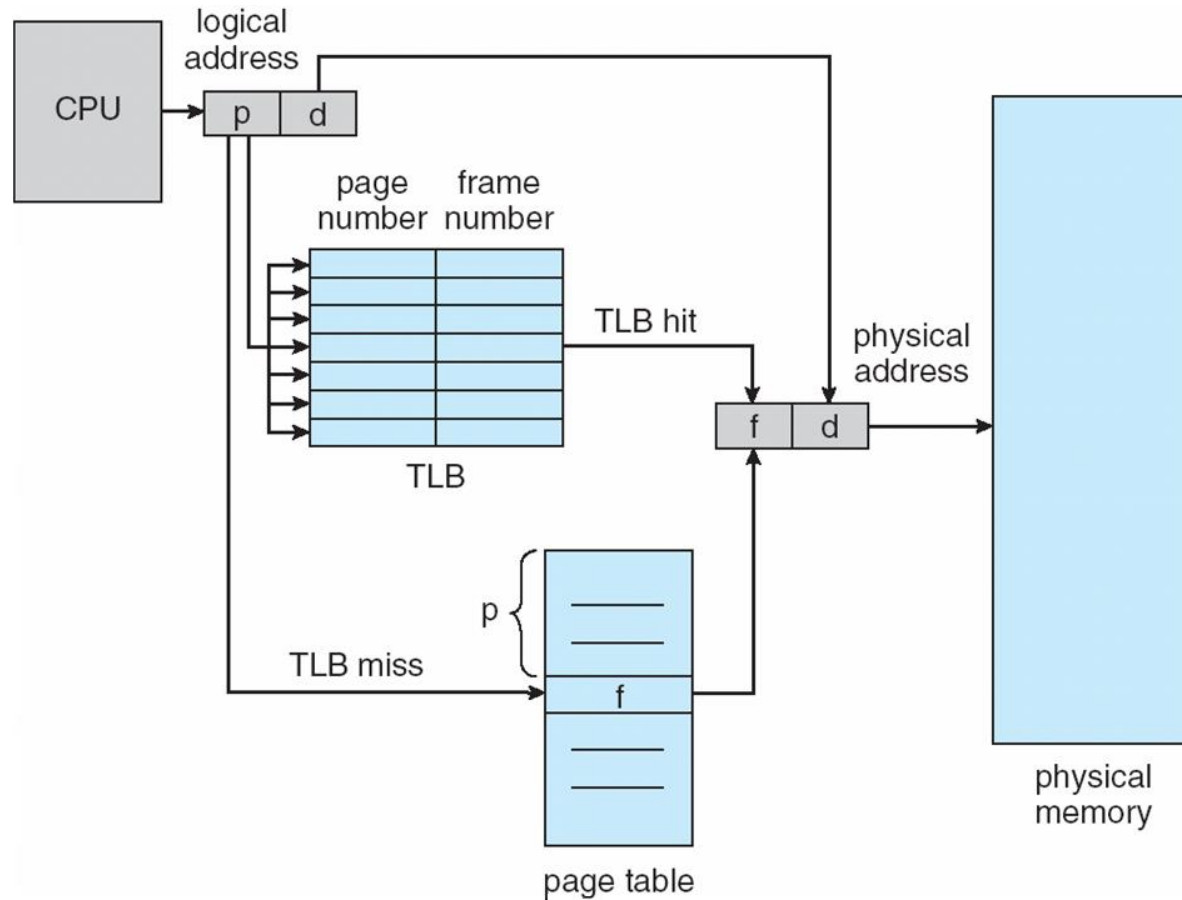
Associative Memory

- 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

- Associative Lookup = ε time unit
 - Can be < 10% of memory access time
- Hit ratio = α
 - Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers

- **Effective Access Time (EAT)**

$$\text{EAT} = \text{hit} [\text{TLB} + \text{MM}] + (1 - \text{hit}) [\text{TLB} + \text{MM} + \text{MM}]$$

- Consider $\alpha = 90\%$, $\varepsilon = 50\mu\text{s}$ for TLB search, $100\mu\text{s}$ for memory access
 - $\text{EAT} = 0.90[50+100] + 0.10[50+100+100] = 160\mu\text{s}$
- Consider more realistic hit ratio -> $\alpha = 99\%$, $\varepsilon = 50\mu\text{s}$ for TLB search, $100\mu\text{s}$ for memory access
 - $\text{EAT} = 0.99 [50+100] + 0.01[50+100+100] = 151\mu\text{s}$

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