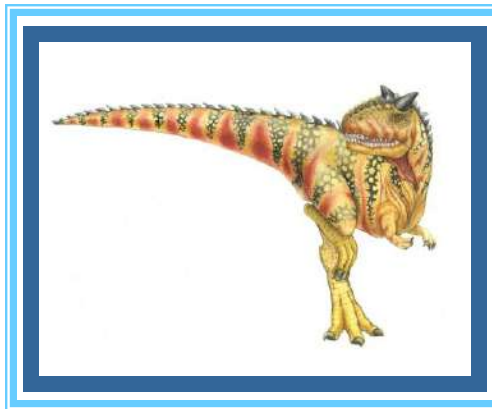


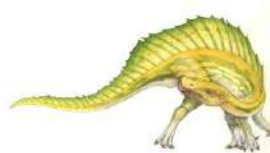
Main Memory





Chapter 9: Memory Management

- Background
- Contiguous Memory Allocation
- Paging
- Structure of the Page Table
- Swapping
- Example: The Intel 32 and 64-bit Architectures
- Example: ARMv8 Architecture





Background

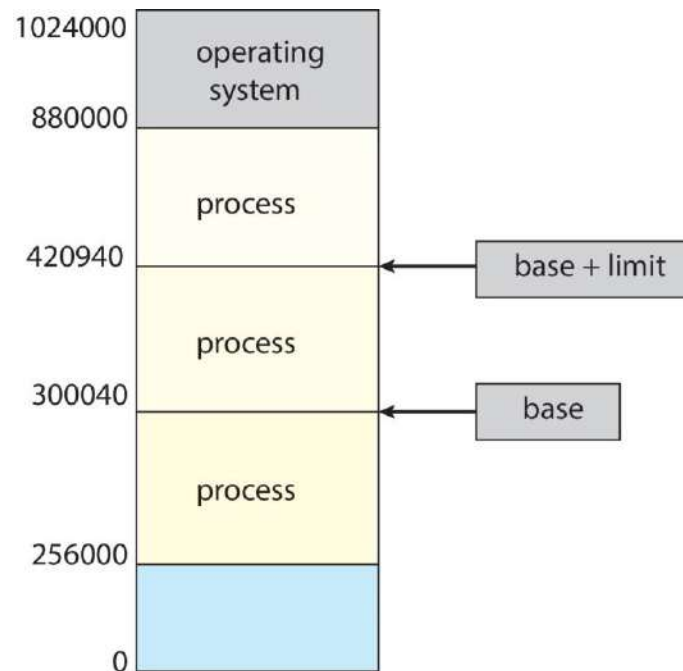
- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Memory unit only sees a stream of:
 - addresses + read requests, or
 - address + data and write requests
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation





Protection

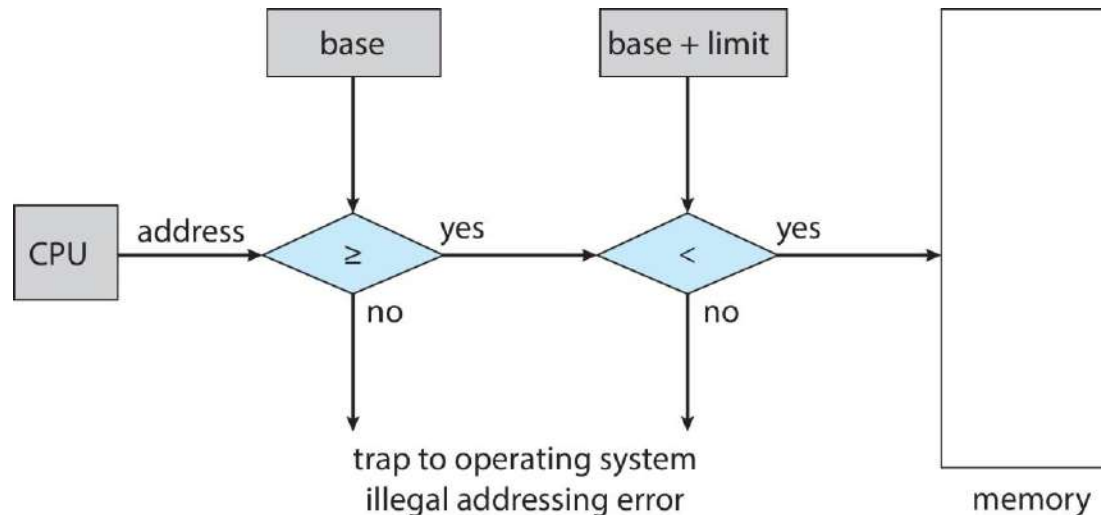
- Need to ensure that a process can access only those addresses in its address space.
- We can provide this protection by using a pair of **base** and **limit registers** define the logical address space of a process





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





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
- 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





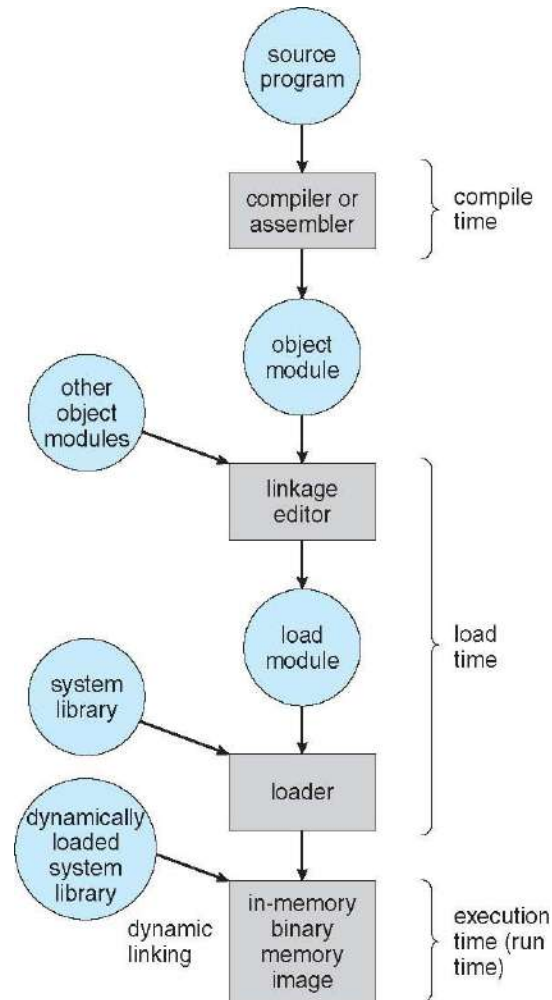
Binding of Instructions and Data to Memory

- 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 from one memory segment to another
 - ▶ Need hardware support for address maps (e.g., base and limit registers)





Multistep Processing of a User Program





Logical vs. Physical Address Space

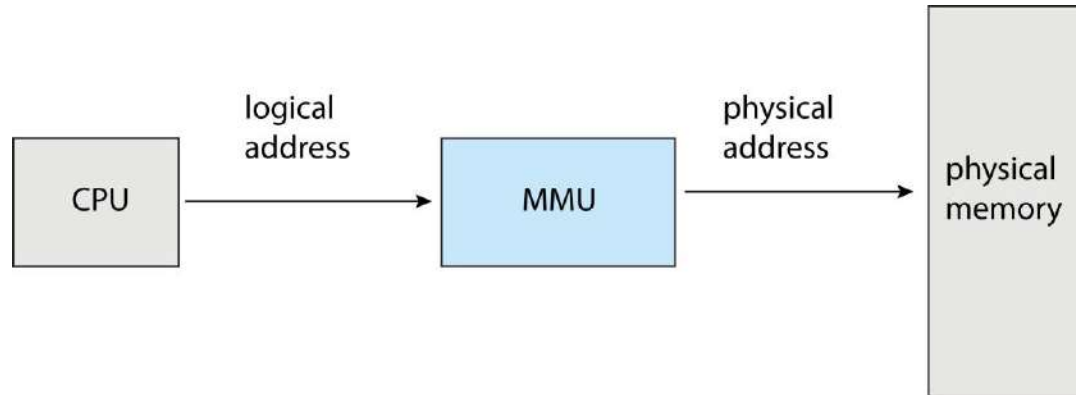
- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
 - **Logical address** – generated by the CPU; also referred to as **virtual address**
 - **Physical address** – address seen by the memory unit
- Logical and physical 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



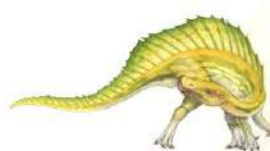


Memory-Management Unit (MMU)

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



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





Memory-Management Unit (Cont.)

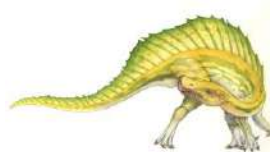
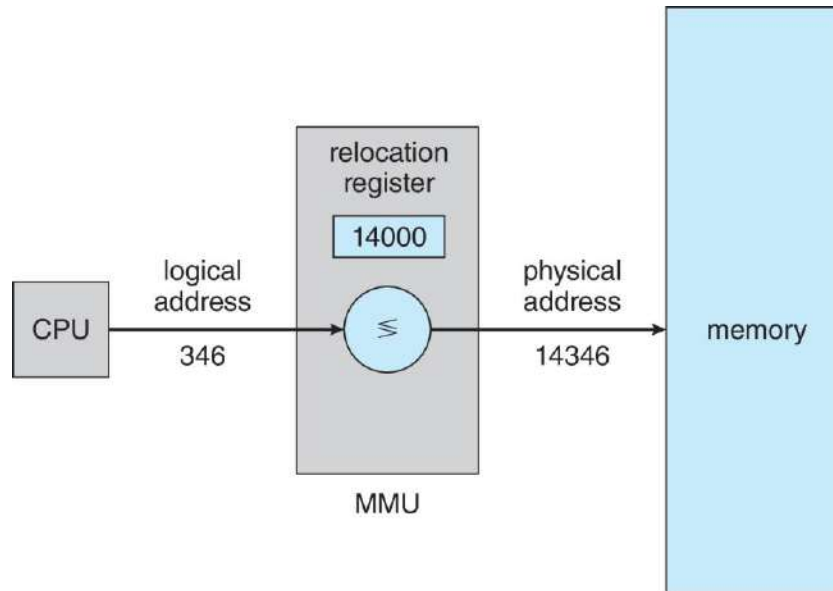
- Consider simple scheme. which is a generalization of the base-register scheme.
- The base register now called **relocation register**
- The value in the relocation register is added to every address generated by a user process at the time it is sent to memory
- The user program deals with *logical* 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

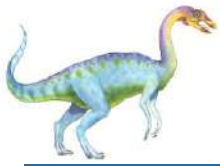




Memory-Management Unit (Cont.)

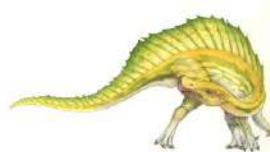
- Consider simple scheme. which is a generalization of the base-register scheme.
- The base register now called **relocation register**
- The value in the relocation register is added to every address generated by a user process at the time it is sent to memory





Dynamic Loading

- The entire program does need to be in memory to execute
- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded
- All routines kept on disk in relocatable load format
- Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required
 - Implemented through program design
 - OS can help by providing libraries to implement dynamic loading





Dynamic Linking

- **Static linking** – system libraries and program code combined by the loader into the binary program image
- Dynamic linking –linking postponed until execution time
- Small piece of code, **stub**, used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and executes the routine
- Operating system checks if routine is in processes' memory address
 - If not in address space, add to address space
- Dynamic linking is particularly useful for libraries
- System also known as **shared libraries**
- Consider applicability to patching system libraries
 - Versioning may be needed





Contiguous Allocation

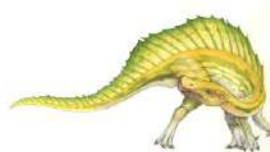
- Main memory must support both OS and user processes
- Limited resource, must allocate efficiently
- Contiguous allocation is one early method
- Main memory usually into two **partitions**:
 - Resident operating system, usually held in low memory with interrupt vector
 - User processes then held in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





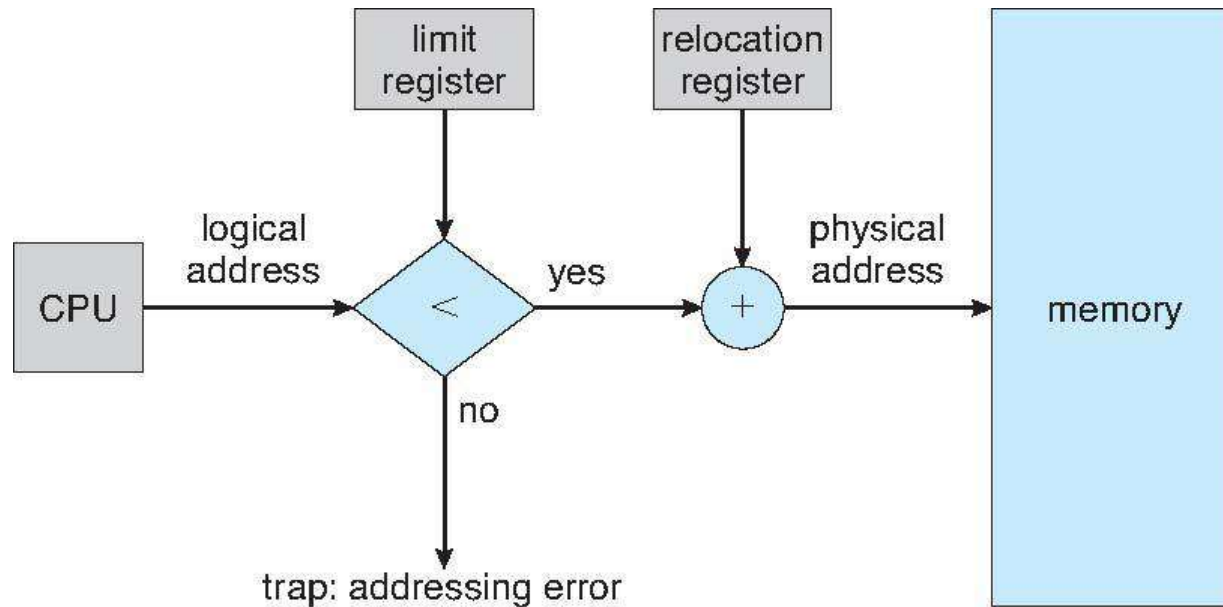
Contiguous Allocation (Cont.)

- Relocation registers used to protect user processes from each other, and from changing operating-system 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





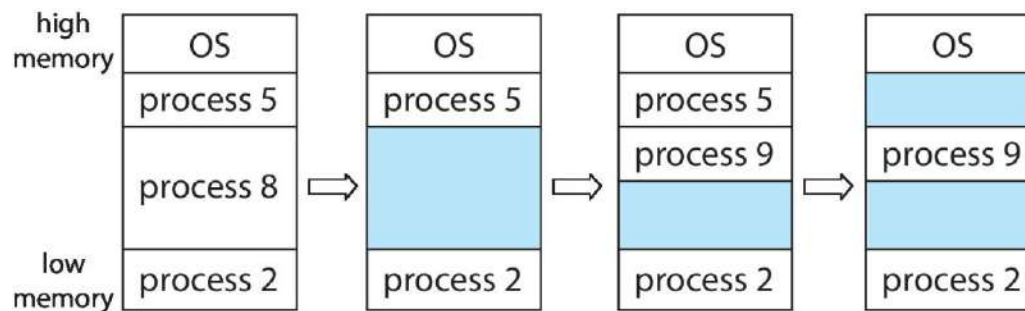
Hardware Support for Relocation and Limit Registers





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)



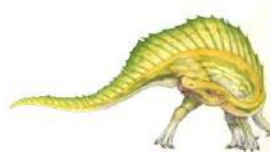


Dynamic Storage-Allocation Problem

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





Example: Given memory partitions of 100K, 500K, 200K, 300K, and 600K (in order), how would each of the First-fit, Best-fit, and Worst-fit algorithms place processes of 212K, 417K, 112K, and 426K (in order)? Which algorithm makes the most efficient use of memory?

Answer:

First-fit:

212K is put in 500K partition
417K is put in 600K partition
112K is put in 288K partition (new partition $288K = 500K - 212K$)
426K must wait

Best-fit:

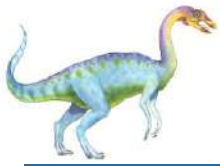
212K is put in 300K partition
417K is put in 500K partition
112K is put in 200K partition
426K is put in 600K partition

Worst-fit:

212K is put in 600K partition
417K is put in 500K partition
112K is put in 388K partition ($600K - 212K$)
426K must wait

In this example, Best-fit turns out to be the best.





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
- The solutions of external fragmentation are such as
 - Compaction, Segmentation and Paging





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
 - ▶ Handle 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





Paging

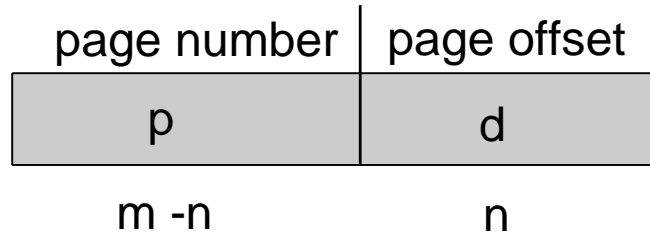
- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
 - Avoids external fragmentation
 - Avoids problem of varying sized memory chunks
- Divide physical memory into fixed-sized blocks called **frames**
 - Size is power of 2, 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



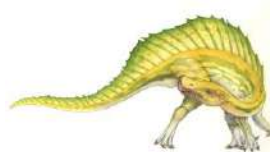


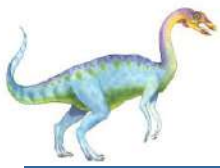
Address Translation Scheme

- 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

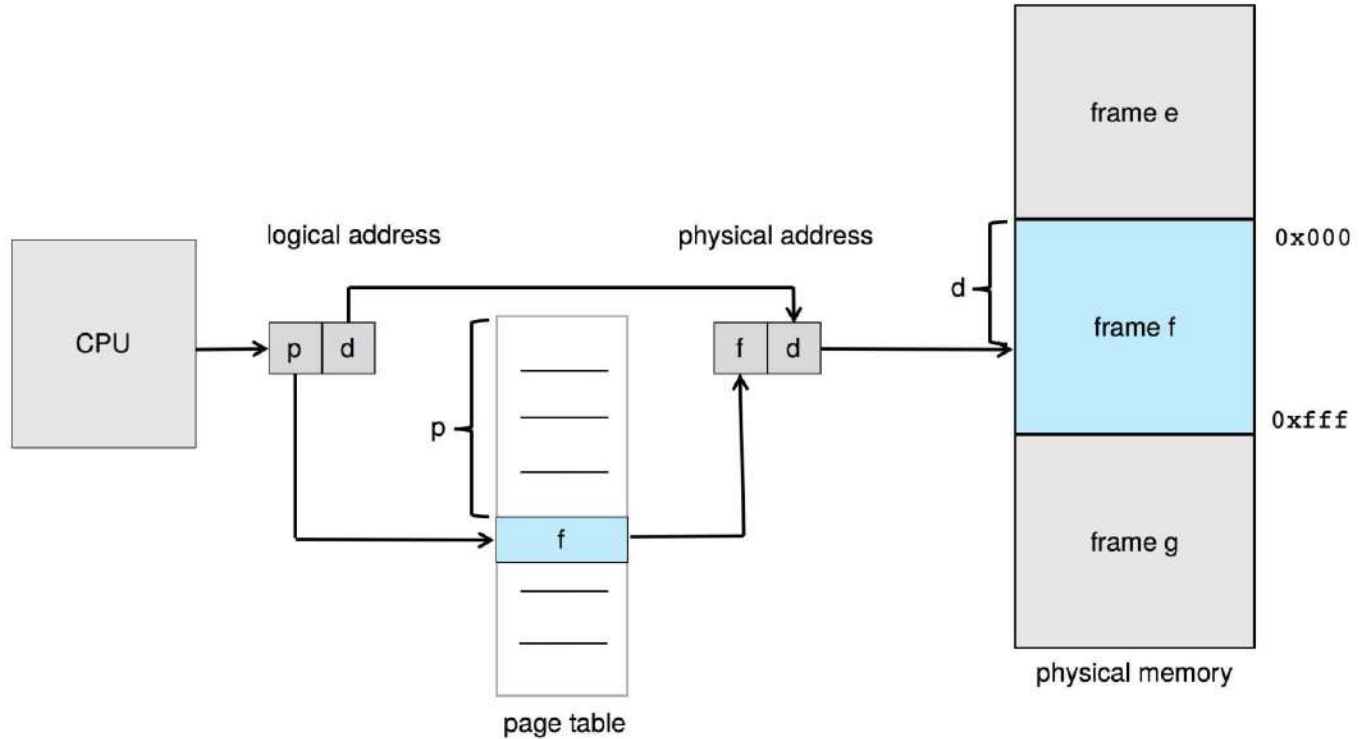


- For given logical address space 2^m and page size 2^n



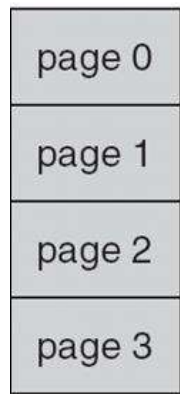


Paging Hardware





Paging Model of Logical and Physical Memory

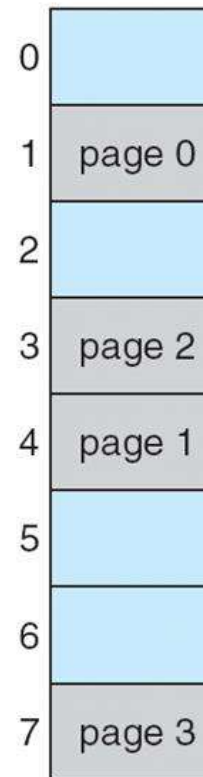


logical
memory

| | |
|---|---|
| 0 | 1 |
| 1 | 4 |
| 2 | 3 |
| 3 | 7 |

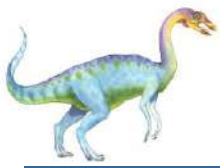
page table

frame
number



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)

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

logical memory

| | |
|---|---|
| 0 | 5 |
| 1 | 6 |
| 2 | 1 |
| 3 | 2 |

page table

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

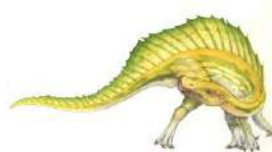
physical memory

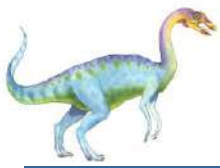




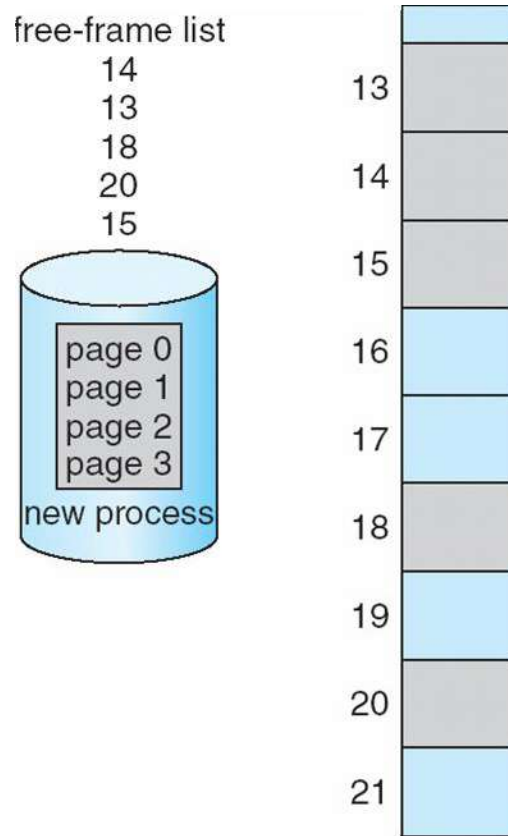
Paging -- Calculating internal fragmentation

- Page size = 2,048 bytes
- Process size = 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



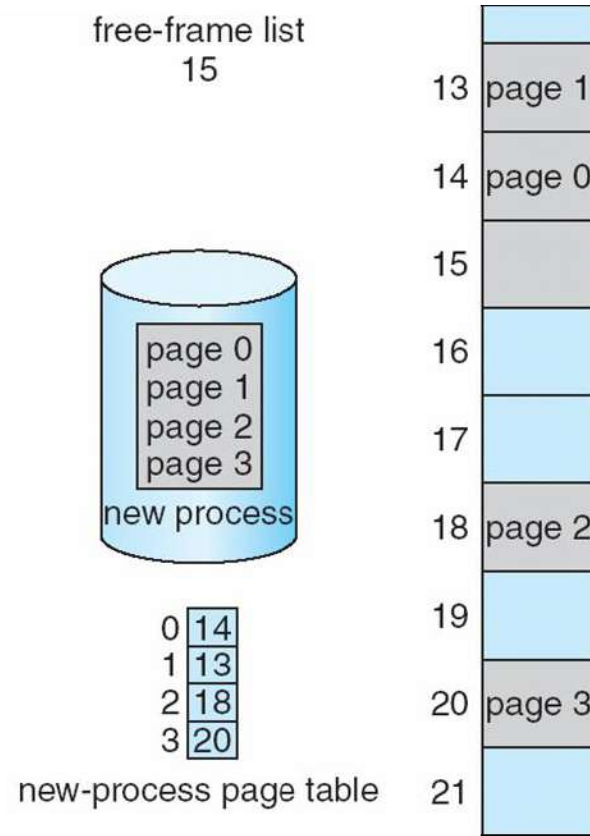


Free Frames



(a)

Before allocation



(b)

After allocation





Implementation of Page Table

- Page table is kept in main memory
 - **Page-table base register (PTBR)** points to the page table
 - **Page-table length register (PTLR)** indicates size of the page table
- 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**).





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





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

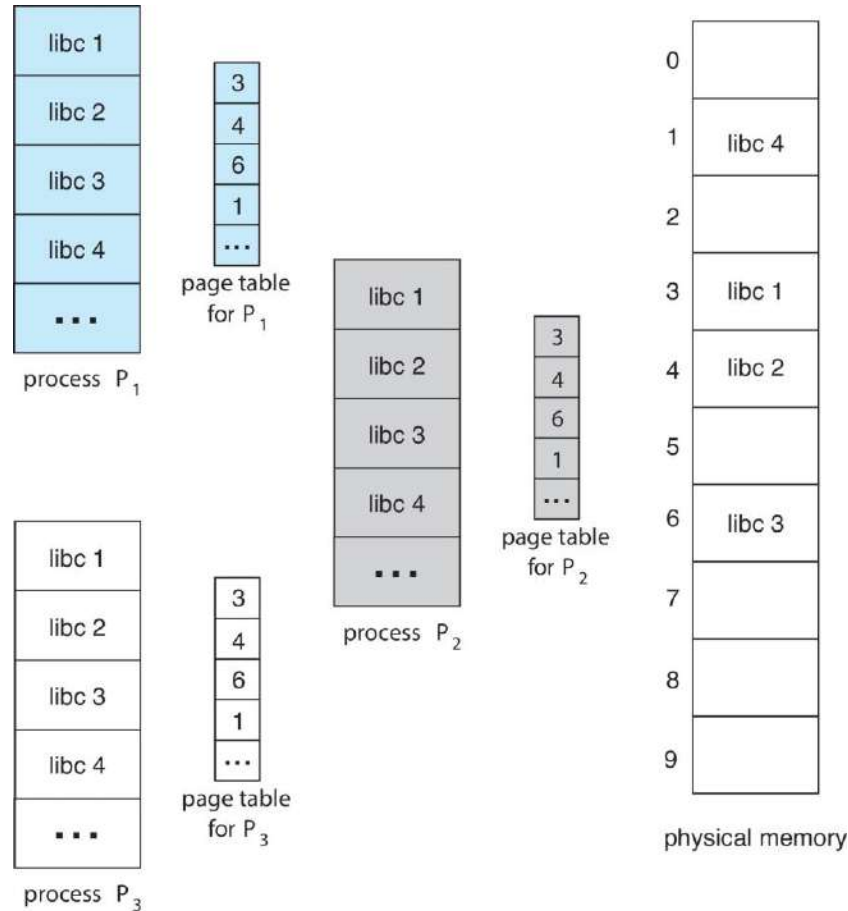
■ Private code and data

- 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





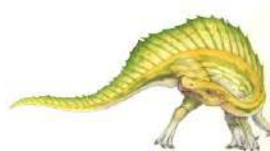
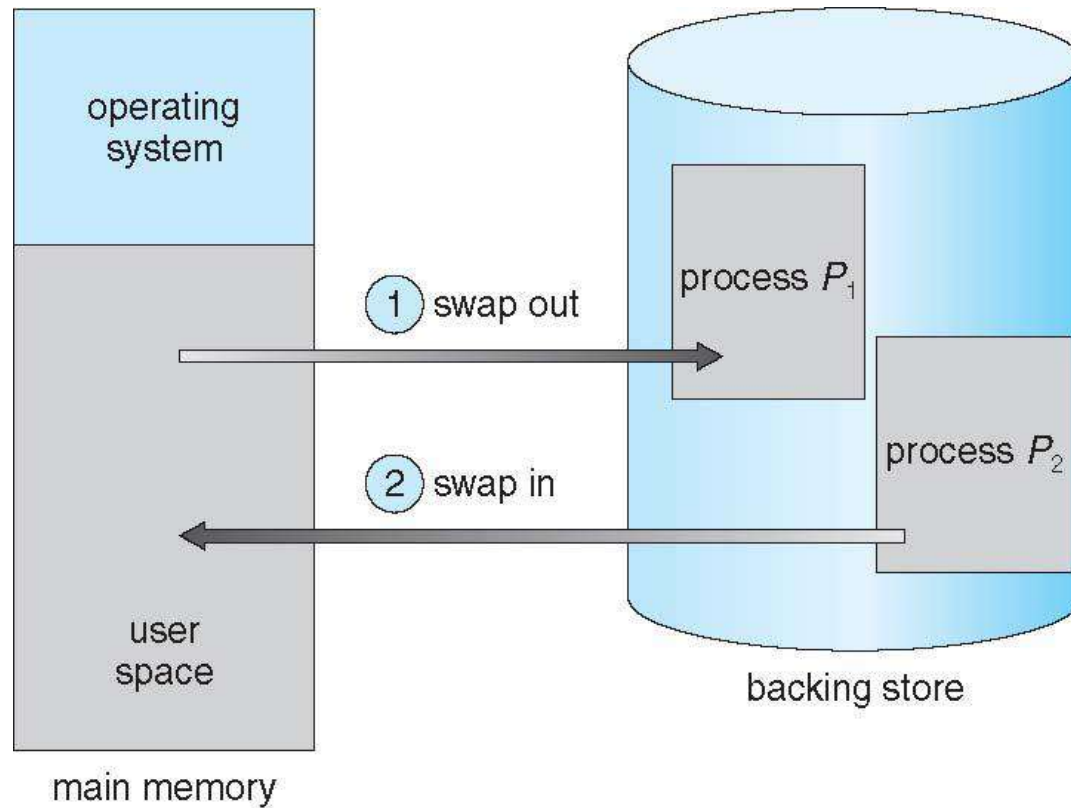
Swapping

- 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





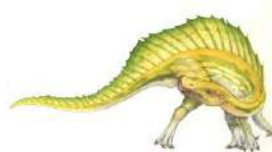
Schematic View of Swapping





Context Switch Time including Swapping

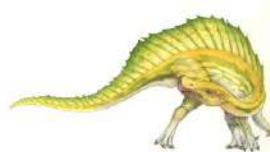
- If next processes to be put on CPU is not in memory, need to swap out a process and swap in target process
- Context switch time can then be very high
- 100MB process swapping to hard disk with transfer rate of 50MB/sec
 - Swap out time of 2000 ms
 - Plus swap in of same sized process
 - Total context switch swapping component time of 4000ms (4 seconds)





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 operating systems
 - But modified version common
 - ▶ Swap only when free memory extremely low

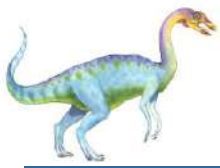




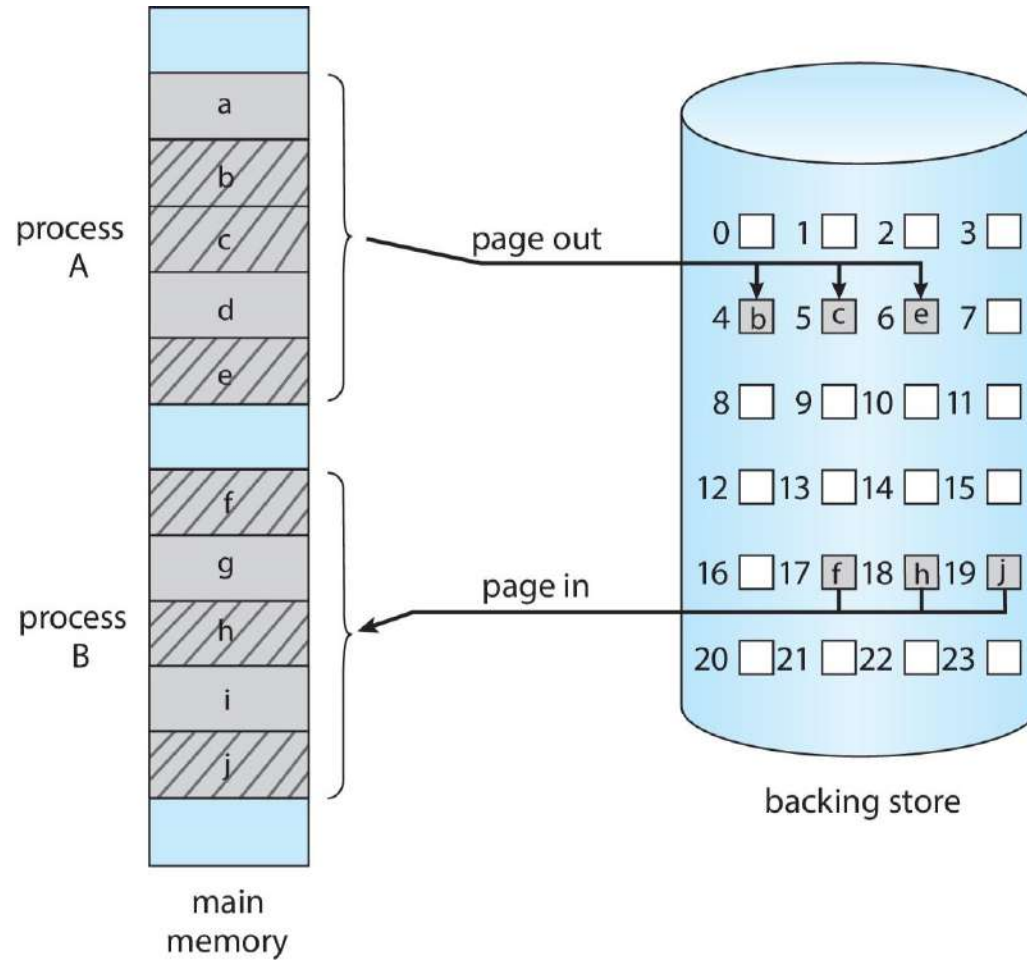
Swapping on Mobile Systems

- 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

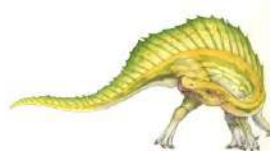




Segmentation

- **Segmentation** is a memory-management scheme that supports this programmer view of memory. A logical address space is a collection of segments.
- Each segment has a name and a length.
- The addresses specify both the segment name and the offset within the segment.
- The programmer therefore specifies each address by two quantities:
 - a segment name and an offset

<segment-number, offset>.



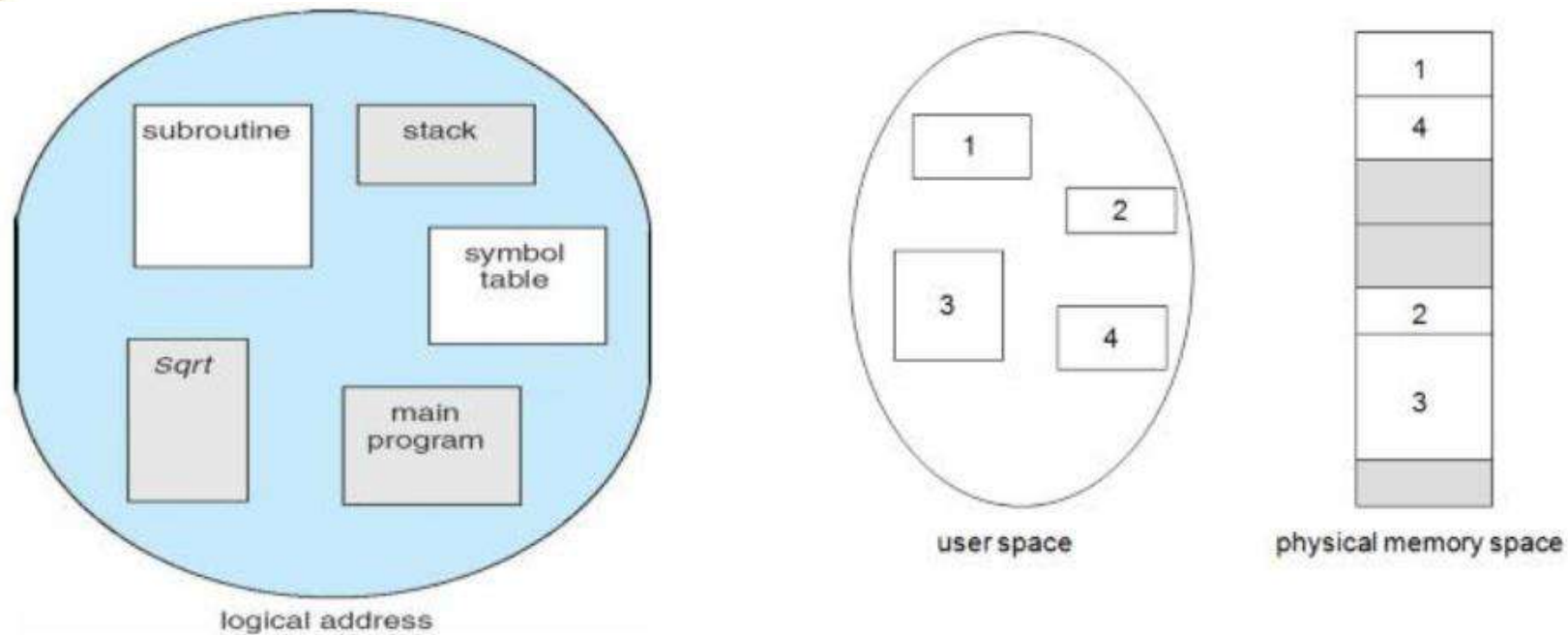


Figure 7: Programmer's view of a program and logical view of segmentation





Segmentation Hardware

- Each entry in the segment table has a **segment base** and a **segment limit**.
 - The **segment base** contains the starting physical address where the segment resides in memory,
 - the **segment limit** specifies the length of the segment

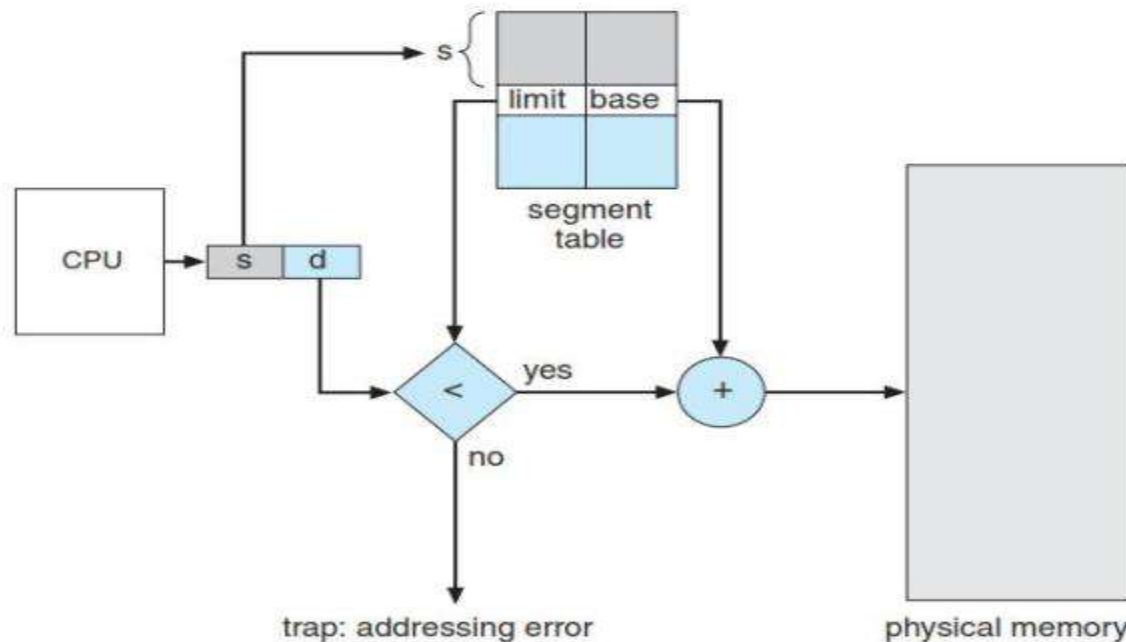


Figure 8: Segmentation hardware





Example

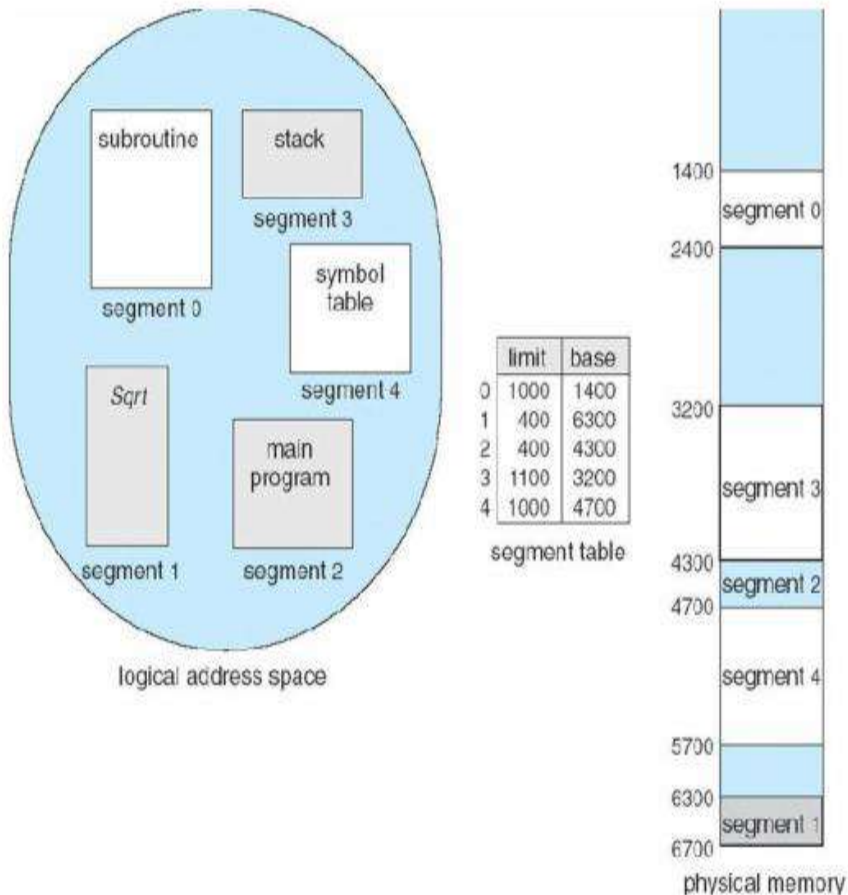


Figure 9: Example of segmentation.

Examples:

- A reference to byte 53 of segment 2 is mapped onto location $4300 + 53 = 4353$.
- A reference to segment 3, byte 852, is mapped to $3200 + 852 = 4052$.

What about a reference to byte 1222 of segment 0 ?

