NPTEL ONLINE CERTIFICATION COURSES

Operating System Fundamentals

Santanu Chattopadhyay
Electronics and Electrical Communication Engg.

Memory Management

CONCEPTS COVERED

Concepts Covered:

- Background
- Swapping
- Contiguous Memory Allocation
- Segmentation
- Paging
- Structure of the Page Table
- Example Architectures

Objectives

- To provide a detailed description of various ways of organizing memory hardware
- To discuss various memory-management techniques, including paging and segmentation
- To provide a detailed description of memory management policies in different architectures

Background

- A program must be brought (from disk) into memory and placed within a process for it to be run
- A program can be written in machine language, assembly language, or high-level language.
- Main memory and registers are the only storage entities that a CPU can access directly
- The CPU fetches instructions from main memory according to the value of the program counter.
- Typical instruction execution cycle – fetch instruction from memory, decode the instruction, operand fetch, possible storage of result in memory.

Background (Cont.)

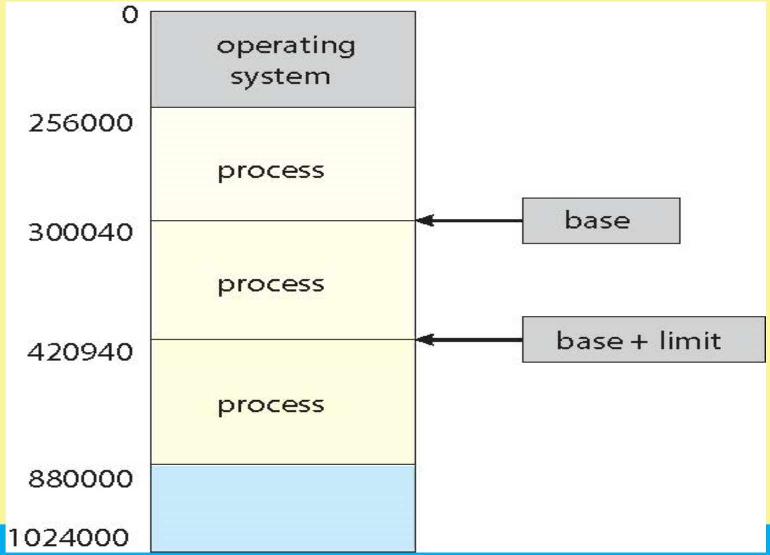
- Memory unit only sees a stream of one of the following:
  - address + read requests (e.g., load memory location 20010 into register number 8)
  - address + data and write requests (e.g., store content of register 6 into memory location 1090)
- Memory unit does not know how these addresses were generated
- Register access can be done in one CPU clock (or less)

Background (Cont.)

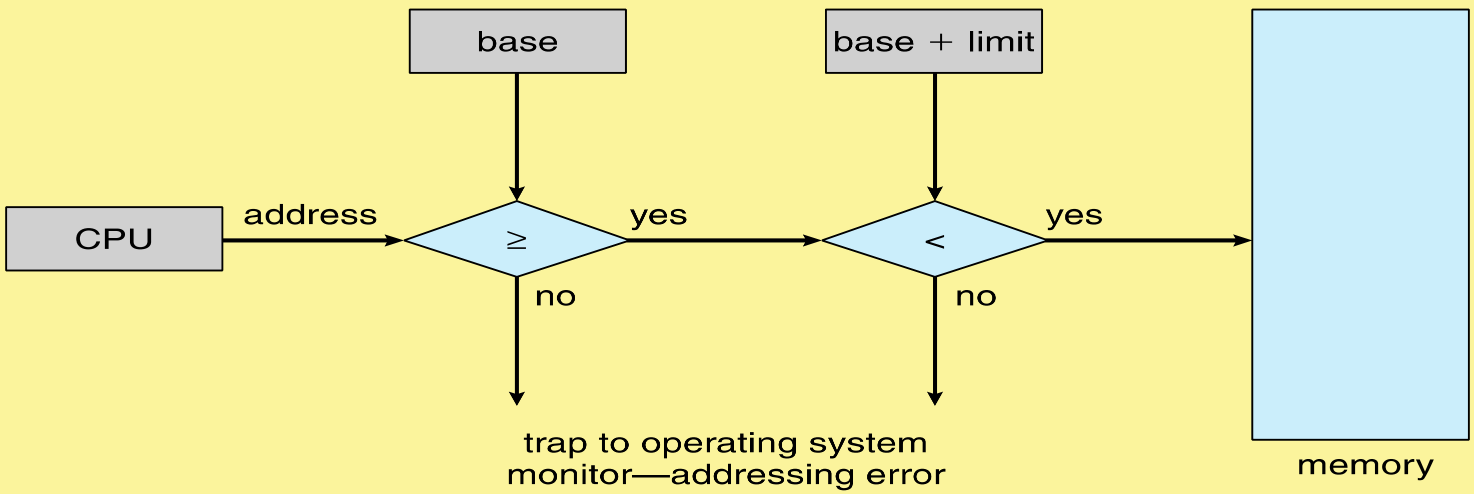
- Completing a memory access may take many cycles of the CPU clock. In such a case the processor needs to **stall** since it does not have the data required to complete the instruction it is executing.
- **Cache** sits between main memory and CPU registers to deal with the “stall” issue.
- Protection of memory is required to ensure correct operation:
  - User process cannot access OS memory
  - One user process cannot access the memory of another user process.

Memory Protection

- A **base register** (holding the smallest legal physical address of a program in memory) and a **limit register** (specifies the size of the program) define the boundary of a program in memory.
- CPU must check that every memory access generated in user mode is between the base and base+limit for that user



Hardware Address Protection



Address Binding

- A program residing on the disk needs to be brought into memory in order to execute. Such a program is usually stored as a binary executable file and is kept in an **input queue.**
- In general, we do not know a priori where the program is going to reside in memory. Therefore, it is convenient to assume that the first physical address of a program always starts at location 0000.
- Without some hardware or software support, program must be loaded into address 0000
- It is impractical to have first physical address of user process to always start at location 0000.
- Most (all) computer systems provide hardware and/or software support for memory management

Address Binding (Cont.)

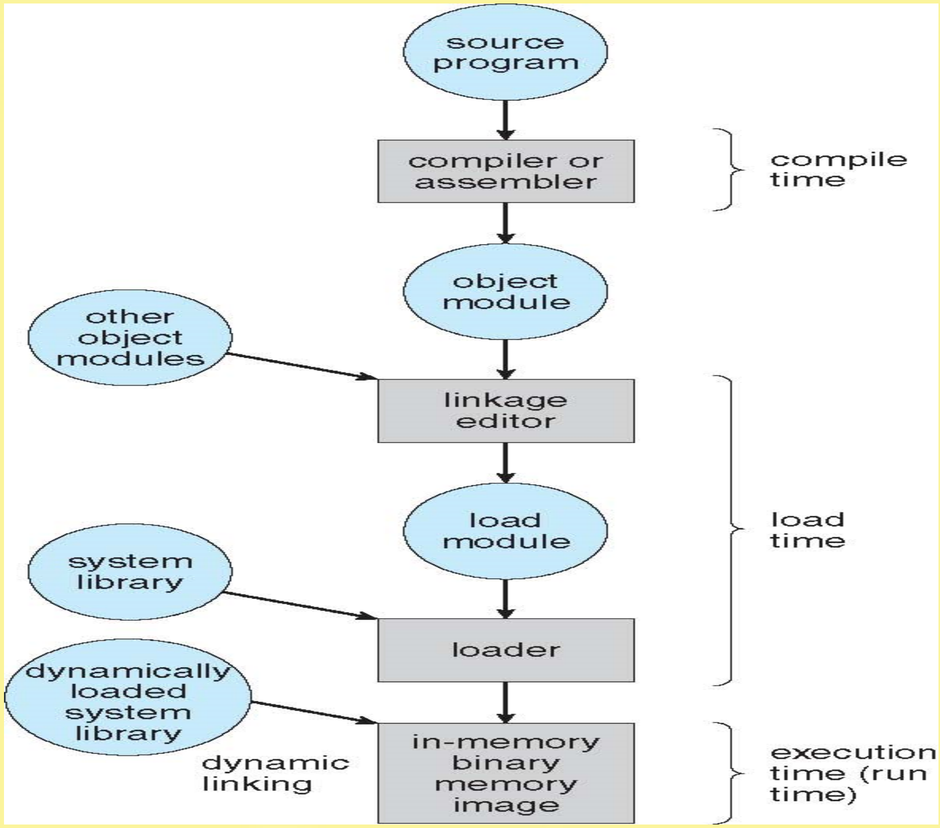
- In general, addresses are represented in different ways at different stages of a program's life
  - Addresses in the source program are generally symbolic
    - For example, variable "count"
  - A compiler typically **bind**s these symbolic addresses to relocatable addresses
    - For example, "14 bytes from beginning of this module"
  - Linker or loader will bind relocatable addresses to absolute (physical) addresses
    - For example, 74014
  - Each binding maps one address space to another address space

Binding of Instructions and Data to Memory

Address binding of instructions and data to memory addresses can happen at three different points in time:

- **Compile time**: If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes.
- **Load time**: If memory location is not known at compile time and no hardware support is available, **relocatable code** must be generated (in software).
- **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



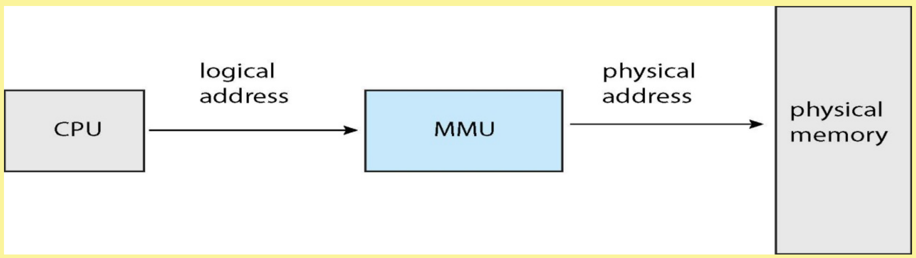
Logical vs. Physical Address Space

- 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.
  - **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. In that case the logical address is referred to as **virtual address.**

  We use Logical address and virtual address interchangeably
- **Logical address space** is the set of all logical addresses generated by a program
- **Physical address space** is the set of all physical addresses corresponding to a given logical address space.

Memory-Management Unit (MMU)

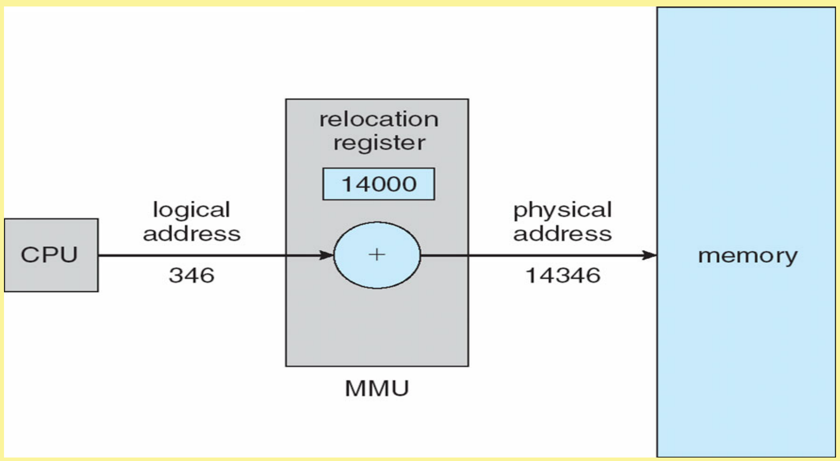
- Hardware device that at run time maps virtual addresses to physical address



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

Dynamic relocation using a relocation register

- To start, consider simple scheme where the value in the base register is added to every address generated by a user process at the time it is sent to memory
  - Base register now called **relocation register**
  - MS-DOS on Intel 80x86 used 4 relocation registers



Dynamic Loading

- Until now we assumed that the entire program and data has to be in main memory to execute
- **Dynamic loading** allows a routine (module) to be loaded into memory only when it is called (used)
- Results in better memory-space utilization; an 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 (e.g., exception handling)
- No special support from the operating system is required
  - It is the responsibility of the users to design their programs to take advantage of such a method
  - OS can help by providing libraries to implement dynamic loading

Dynamic Linking

- **Dynamically linked libraries** – system libraries that are linked to user programs when the programs are run.
  - Similar to dynamic loading. But, linking rather than loading is 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**

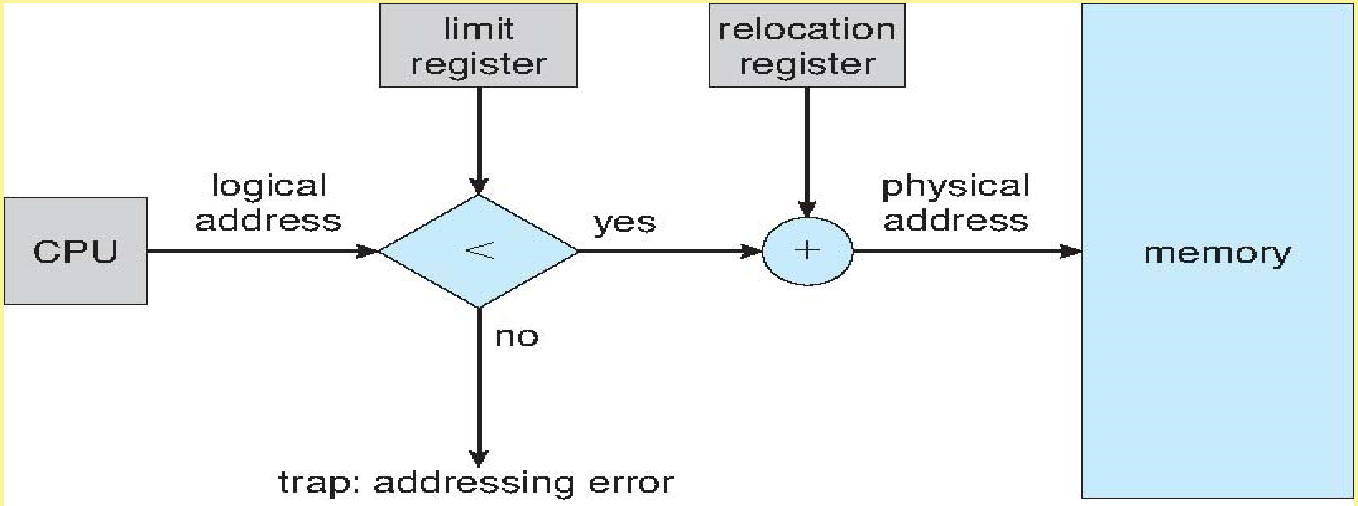
Contiguous Allocation

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

Contiguous Allocation (Cont.)

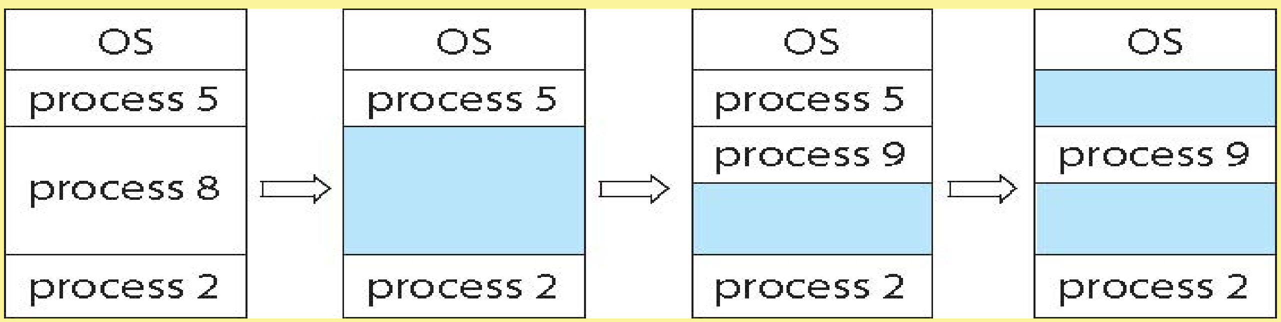
- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - Base register contains value of smallest physical address
  - Limit register contains range of logical addresses – each logical address must be less than the limit register
  - MMU maps logical address *dynamically*
  - Can then allow actions such as kernel code being **transient** – comes and goes as needed. Thus, kernel can change size dynamically.

Hardware Support for Relocation and Limit Registers



Multiple-partition allocation

- **Variable-partition** -- 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 (holes)



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 the list is ordered by size.
    - Produces the smallest leftover hole
  - **Worst-fit**: Allocate the ***largest*** hole; must also search entire list, unless the list is ordered by size
    - Produces the largest leftover hole
- First-fit and best-fit are 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 and therefore cannot be used.
  - First fit analysis reveals that given *N* allocated blocks, another 0.5 *N* blocks will be lost to fragmentation
    - 1/3 of memory may be unusable -> **50-percent rule**
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory.
  - Can happen if there is hole of size 15,000 bytes and a process needs 14,900 bytes; Keeping a hole of size 100 bytes is not worth the effort so the process is allocated 15,000 bytes.
  - The size difference of 100 bytes 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, and is done at execution time
- I/O problem -- cannot perform compaction while I/O is in progress involving memory that is being compacted.
  - Latch job in memory while it is involved in I/O
  - Do I/O only into OS buffers

Non-contiguous Allocation

- Partition the a program into a number of small units, each of which can reside in a different part of the memory.
- Need hardware support.
- Various methods to do the partitions:
  - Segmentation.
  - Paging
  - paged segmentation.

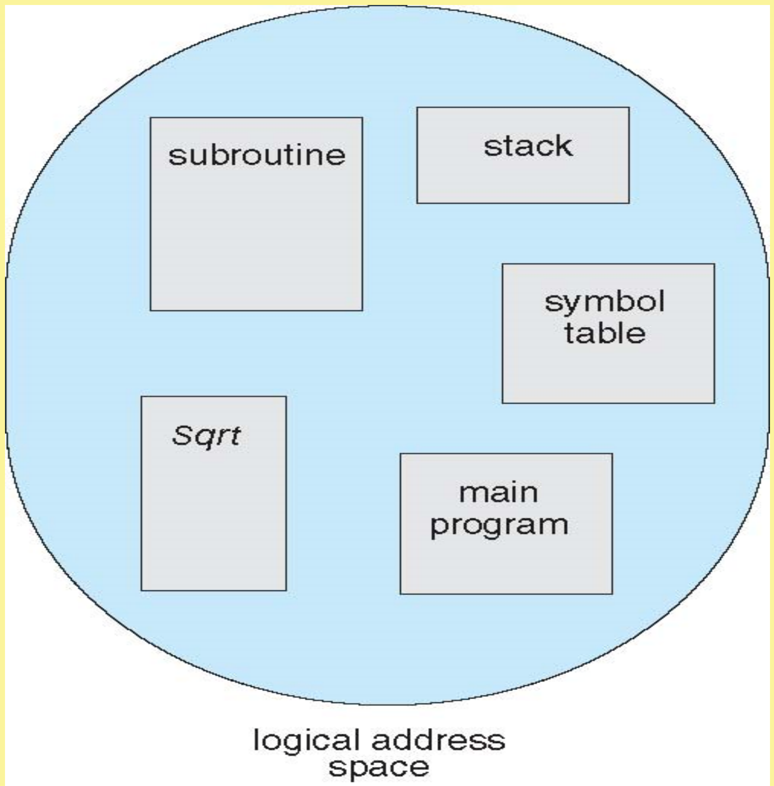
Segmentation

- Memory-management scheme that supports user's view of memory
- A program is a collection of segments -- a logical unit such as:

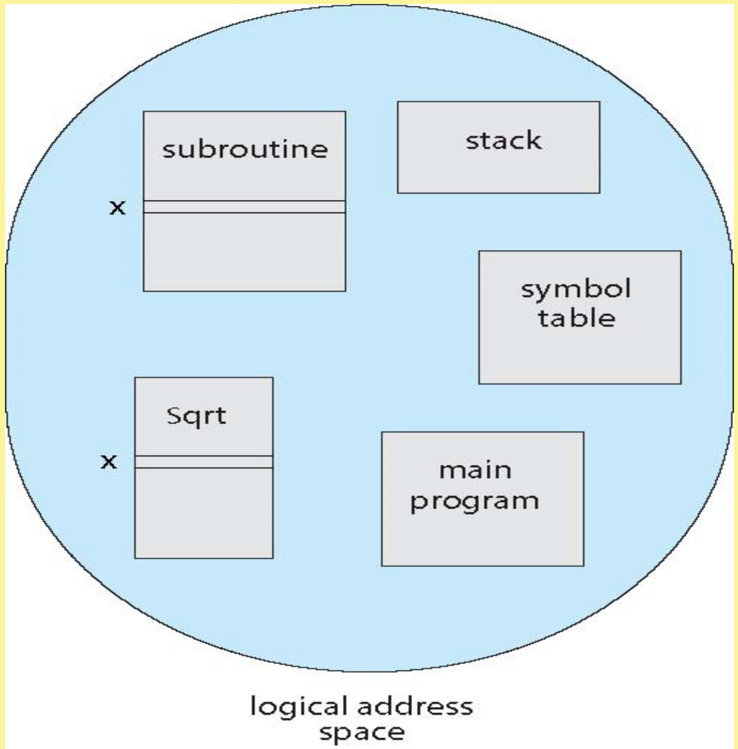
  main program, procedure, function, method,
  object, local variables, global variables,
  common block, stack, symbol table, arrays

- Each segment can do reside in different parts of memory.
- Way to circumvent the contiguous allocation requirement.

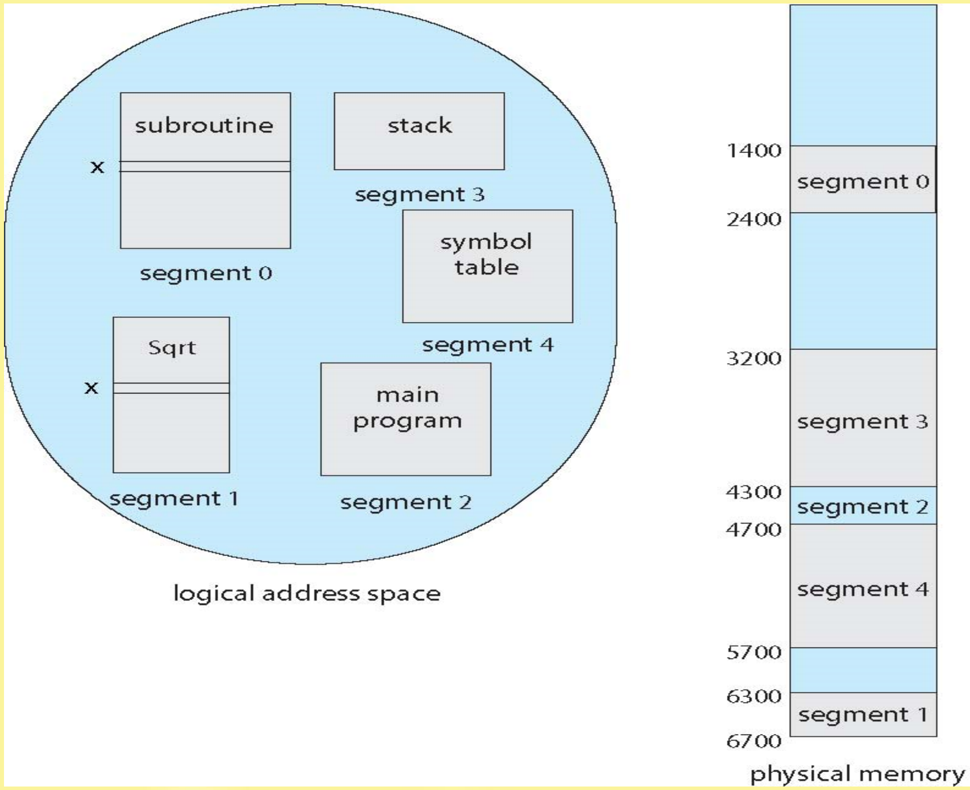
User's View of a Program



Two Dimensional Addresses



Logical and Physical Memory



Segmentation Architecture

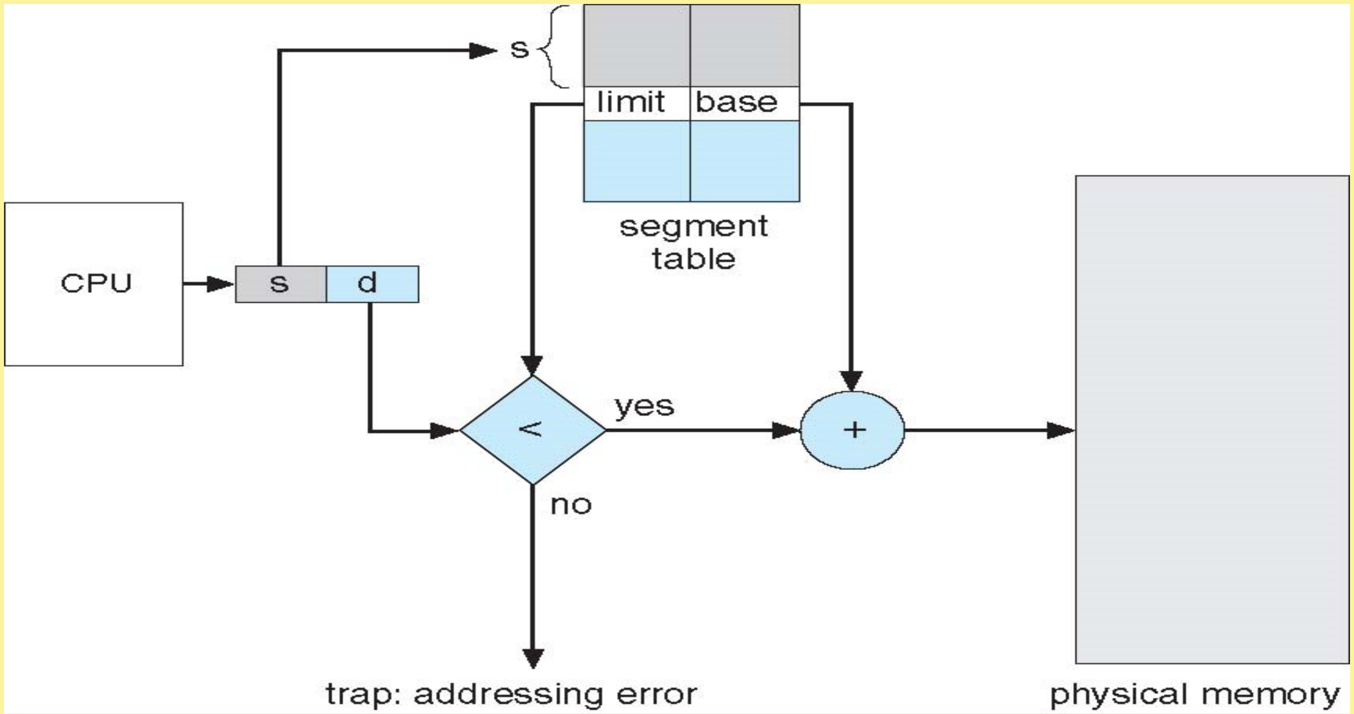
- Logical address consists of a two tuple:

  $\langle$segment-number, offset$\rangle$

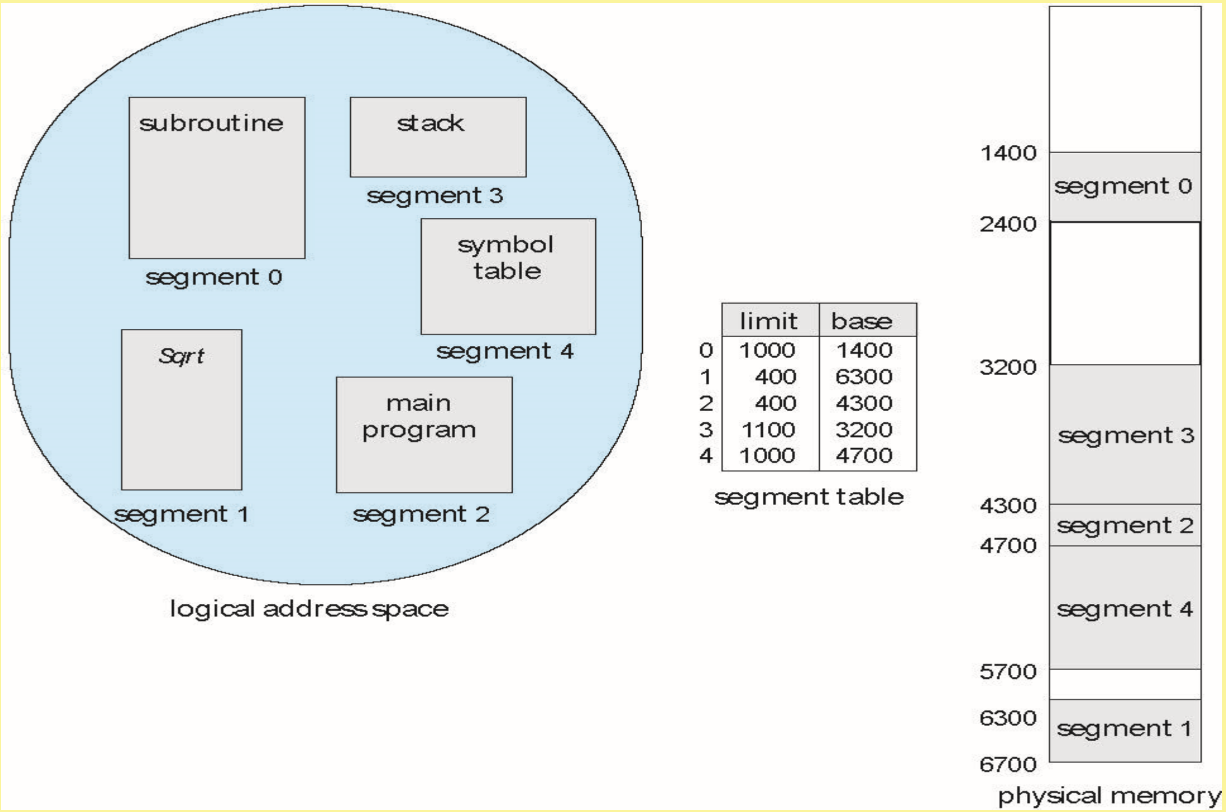
- Need to map a two-dimensional logical addresses to a one-dimensional physical address. Done via **Segment table**:
  - **base** – contains the starting physical address where a segments reside in memory
  - **limit** – specifies the length of the segment
- Segment table is kept in memory
  - **Segment-table 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 Hardware



Example of Segmentation



Paging

- Physical address space of a process can be non-contiguous.
- Process is divided into fixed-size blocks, each of which may reside in a different part of physical memory.
- Divide physical memory into fixed-sized blocks called **frames**
  - Size of a frame is power of 2 between 512 bytes and 16 Mbytes
- Divide logical memory into blocks of same size as frames called **pages**
- Backing store (dedicated disk), where the program is permanently residing, is also split into storage units (called **blocks**), which are the same size as the frame and pages.
- Physical memory allocated whenever the latter is available
  - Avoids external fragmentation
  - Still have Internal fragmentation

Paging (Cont.)

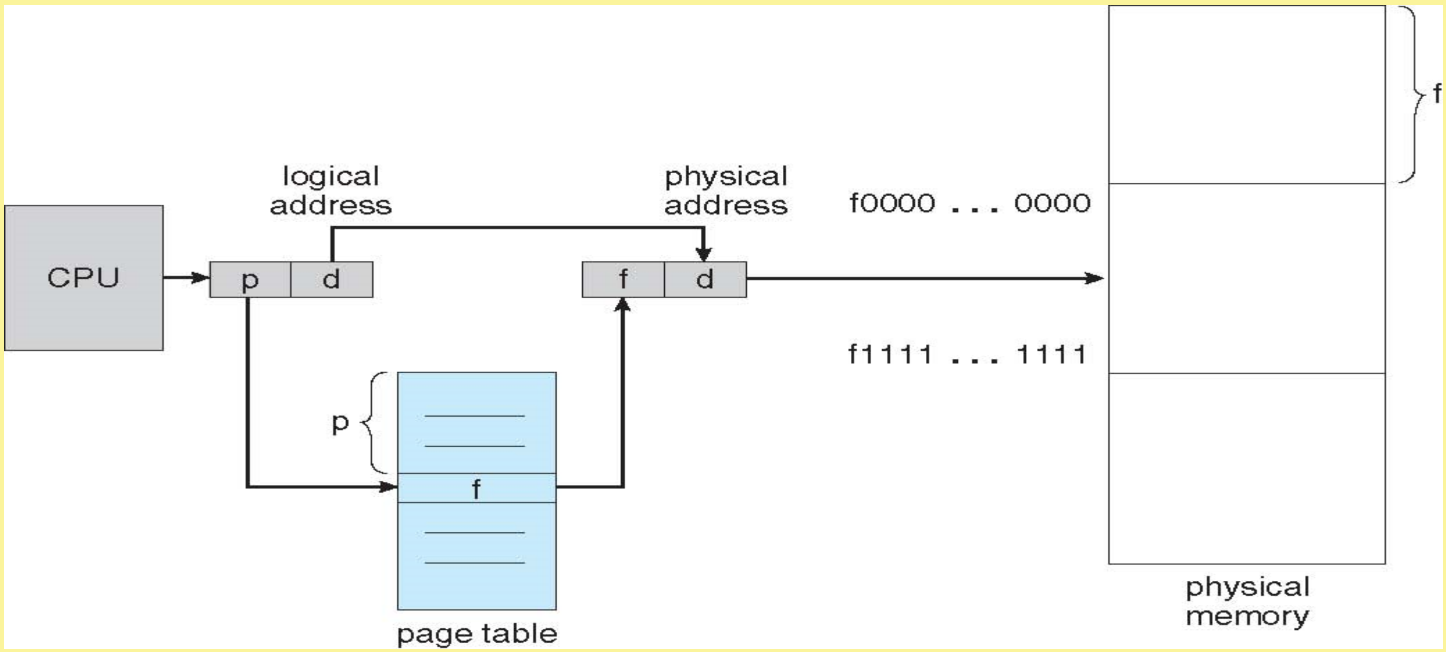
- Keep track of all free frames
- To run a program of size ***N*** pages, need to find ***N*** free frames and load program from backing store.
- Set up a **page table** to translate logical to physical addresses
- Page table is kept in memory.
  - **Page-table base register** (**PTBR**) points to the page table
  - **Page-table length register** (**PTLR**) indicates size of the page table
- Still have Internal fragmentation

Address Translation Scheme

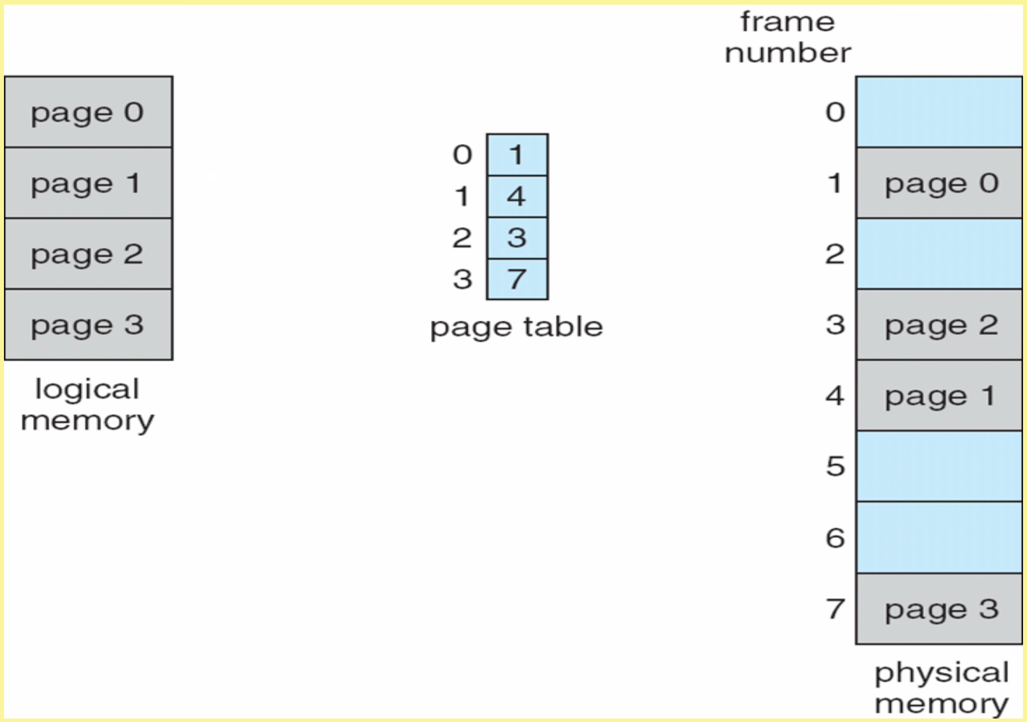
- Assume the logical address space is $2^{m.}$ *(*How is *m* determined?)
- Assume page size is *2^n*
- Address generated by CPU is divided into:
  - **Page number** (***p***) – used as an index into a **page table** which contains base address of each page in physical memory. Size of ***p*** is "m – n"
  - **Page offset** (***d***) – combined with base address to define the physical memory address that is sent to the memory unit. Size of ***d*** is "n".

page number	page offset
p	d
$m - n$	n

Paging Hardware

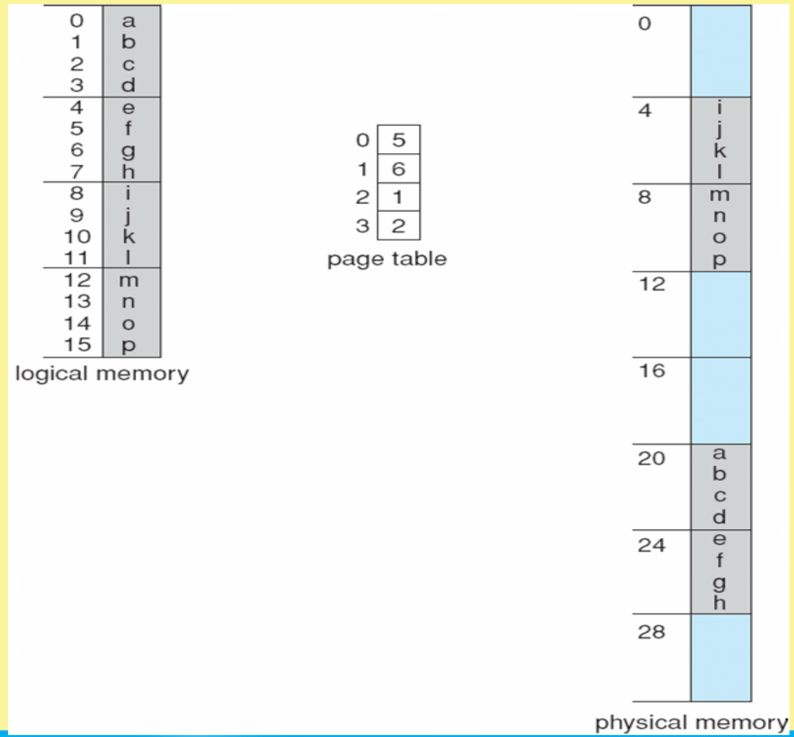


Paging Model of Logical and Physical Memory



Paging Example

Assume m = 4 and n = 2 and 32-byte memory and 4-byte pages



Internal Fragmentation

- 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
- By implementation process can only access its own memory