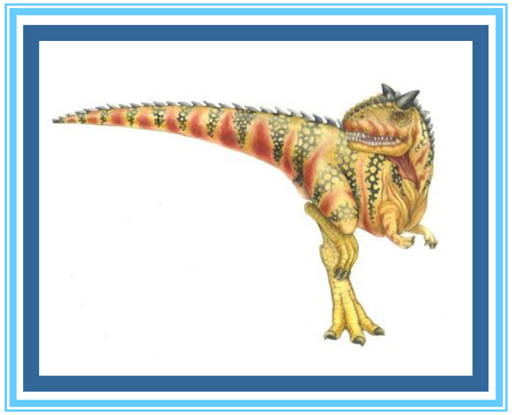
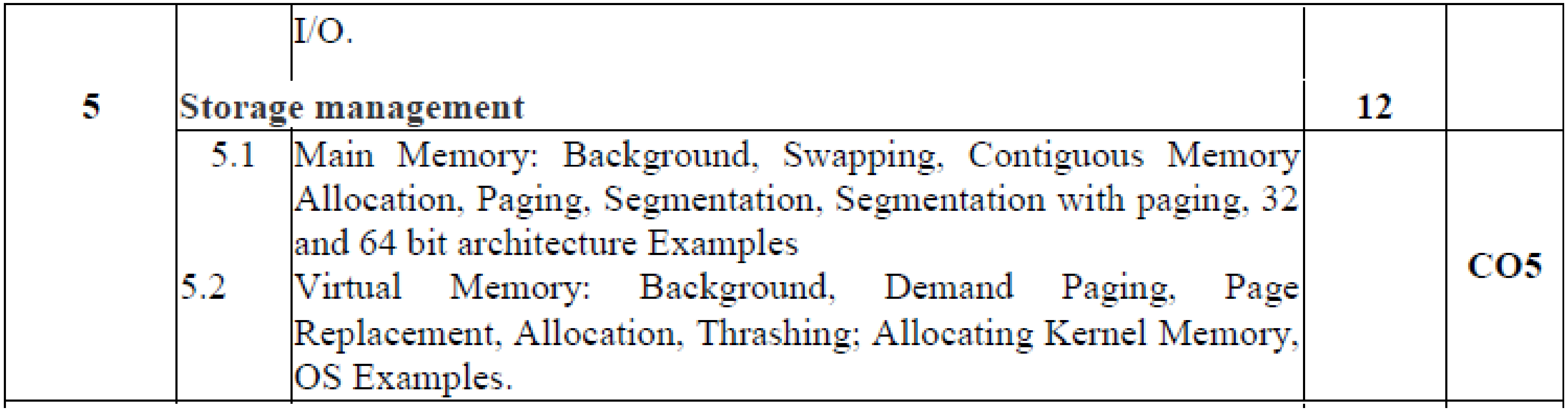
Storage Management
Module 5.1





		I/O.		
5	Storage management		12	
	5.1 5.2	Main Memory: Background, Swapping, Contiguous Memory Allocation, Paging, Segmentation, Segmentation with paging, 32 and 64 bit architecture Examples Virtual Memory: Background, Demand Paging, Page Replacement, Allocation, Thrashing; Allocating Kernel Memory, OS Examples.		CO5

How do you differentiate?

- Memory
- Main memory
- Physical Memory
- RAM
- Primary memory

Basics

- Memory is central to operation of computing systems
- Memory = a large array of words/bytes
- Each byte or word has own address
- Memory contains the program to be executed and data, both
- Program is executed line by line with Instruction Fetch, Instruction Decode, Operand Fetch, Execute cycles
- Program counter contains address of memory location to be executed next

Background

- Memory consists of a large array of words or bytes, each with its own address.
- The CPU fetches instructions from memory according to the value of the program counter. These instructions may cause additional loading from and storing to specific memory addresses.

Background

- A typical instruction-execution cycle, for example,

1) first fetches an instruction from memory.
2) The instruction is then decoded and may cause operands to be fetched from memory.
3) After the instruction has been executed on the operands, results may be stored back in memory.

Memory unit only sees a stream of addresses + read requests, or address + data and write requests

Memory : hardware

- CPU can directly access only main memory and processor registers
- The instructions take main memory addresses as program arguments and not disk addresses
- **Hence- Data and instructions executing them, have to be in MM**
- **Issues and concerns?**
  - **Speed of access : solution cache memory**
  - **Protect program data : solution base and limit registers**
  - **Size of memory : solution swapping, virtual memory**
  - **Many more ones….**

Basic Hardware

- Register access in one CPU clock (or less)
- Main memory access can take many cycles, causing a **stall**
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

Basic Hardware

- **Register access in one CPU clock (or less)**
- Main memory access can take many cycles, causing a **stall**
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

- Registers that are built into the CPU are generally accessible within one cycle of the CPU clock.

Basic Hardware

- Register access in one CPU clock (or less)
- **Main memory access can take many cycles, causing a stall**
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

- Main memory is accessed via a transaction on the memory bus.
- Completing a memory access may take many cycles of the CPU clock.
- In such cases, the processor normally needs to stall, since it does not have the data required to complete the instruction that it is executing

Speed of access

- One of the Speed judgement parameters: number of CPU clock cycles needed to perform memory operation
- CPU is a faster than memory
- CPU registers are accessible in one CPU clock cycle
- But MM is accessed via a transaction on memory bus
- This access takes many CPU clocks to complete
- Result? : CPU has program but not the data to complete instruction execution i.e. **CPU stalling**

Basic Hardware

- Register access in one CPU clock (or less)
- Main memory access can take many cycles, causing a stall
- **Cache sits between main memory and CPU registers**
- Protection of memory required to ensure correct operation

- The remedy is to add fast memory between the CPU and main memory called cache.

Speed of access

- As the data are needed always for instruction execution, frequent memory stalls are bottleneck
- Solution : add a faster memory between CPU and main memory i.e. **cache memory**
- Cache memory: a buffer to accommodate the speed difference

Basic Hardware

- Register access in one CPU clock (or less)
- Main memory access can take many cycles, causing a stall
- Cache **sits between main memory and CPU registers**
- Protection of memory required to ensure correct operation

- Apart from relative speed of accessing physical memory, we also must ensure correct operation.
- To protect the operating system from access by user processes
- To protect user processes from one another. This protection must be provided by the hardware.

Access Protect

- OS must be protected from unauthorized access by user process
- **User processes must be protected from each other**
- **Each process has range of legal addresses to access**
- Base register and limit register ensure process can access only addresses within the legal range

Image courtesy: Galvin

Basic Hardware

- We first need to make sure that each process has a separate memory space.
- **Determine the range of legal addresses that the process may access**
- **To ensure that the process can access only these legal addresses.**
- **We can provide this protection by using two registers, usually a base and a limit.**

Base and Limit Registers

- Base Register-Holds the smallest legal physical memory address
- Limit Register- specifies the size of the range.

Base and Limit Registers

- For example,
- if the base register holds 300040
- and the limit register is 120900,
- then the program can legally access all addresses from 30 0040 through 420939 (inclusive).

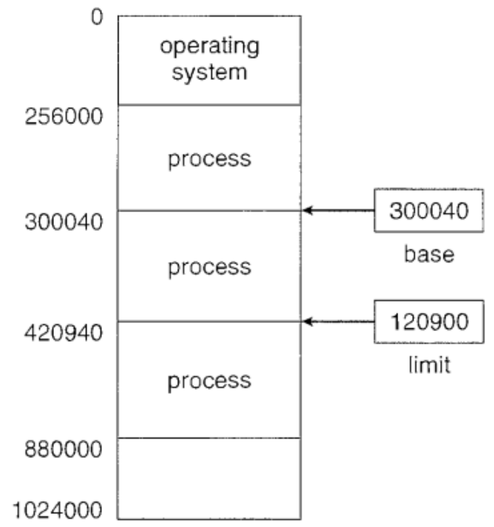


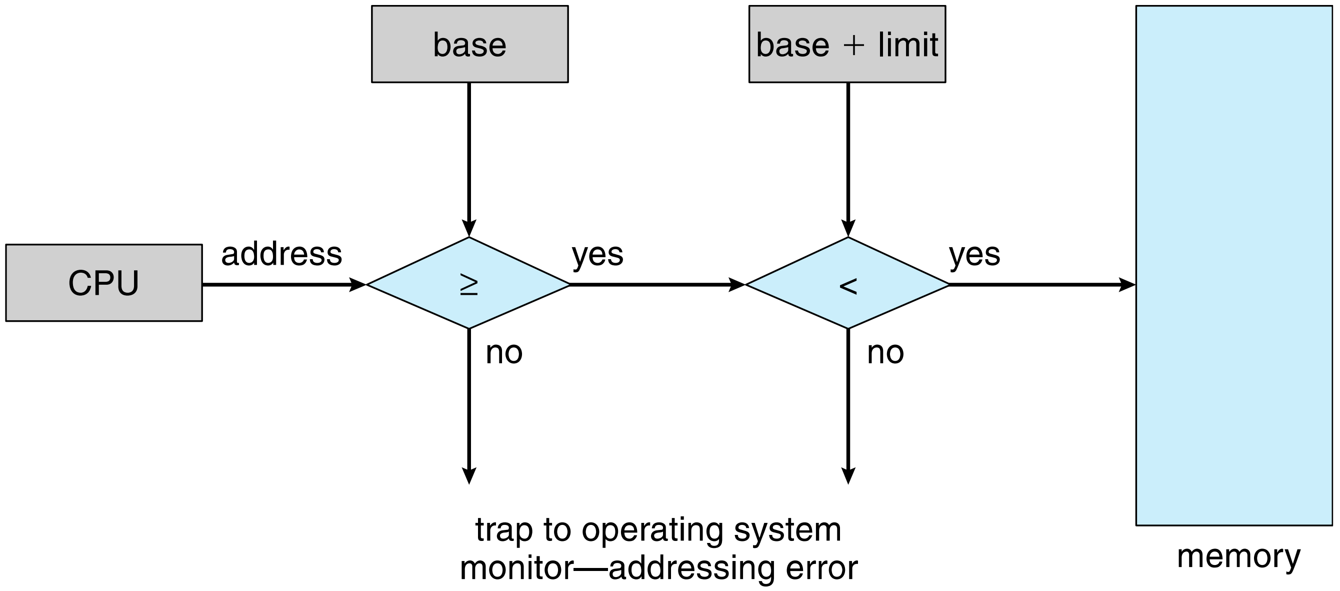
Figure 8.1 A base and a limit register define a logical address space.

Base and Limit Registers

- Protection of memory space is accomplished by having the CPU hardware compare *every* address generated in user mode with the registers.
- Any attempt by a program executing in user mode to access operating-system memory or other users' memory
- **Results in a trap to the operating system, which treats the attempt as a fatal error**
- **This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the operating system or other users.**

Base and Limit Registers

- 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

- **The base and limit registers can be loaded only by the operating system**, which uses a special privileged instruction.
- Since privileged instructions can be executed only in kernel mode,
- since only the operating system executes in kernel mode,
- only the operating system can load the base and limit registers.
- **This scheme allows the operating system to change the value of the registers but prevents user programs from changing the registers' contents.**

Hardware Address Protection

- The operating system, executing in kernel mode, is given unrestricted access to both operating system memory and users' memory.
- This provision allows the operating system
  - to load users' programs into users' memory,
  - to dump out those programs in case of errors

Address Binding

- Usually, a program resides on a disk as a binary executable file.
- To be executed, the program must be brought into memory and placed within a process.
- Programs on disk, ready to be brought into memory to execute form an **input queue**
- **The normal procedure is to select one of the processes in the input queue and to load that process into memory.**
- As the process is executed, it accesses instructions and data from memory.
- Eventually, the process terminates, and its memory space is declared available.

Address binding

- User processes reside in any part of main memory
- Computer address space starts at 00000
- But user process need not be loaded at 00000
- The program is processed by various system softwares before being executed such as macro processor, compiler, linker, loader etc..
- User program addresses may be represented in various ways during these executions

Address binding.. contd

- Addresses in source program are symbolic
- Compiler binds symbolic addresses to relocatable addresses
  - E,g. 14 bytes from beginning of this module
- The linker or loaders turn relocatable addresses to absolute addresses
- Each such binding is mapping from one address space to another

Address Binding

- Further, addresses represented in different ways at different stages of a program's life
  - Source code addresses are usually symbolic (Eg-count)
  - A compiler will typically bind these symbolic addresses 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 from one address space to another

Binding of Instructions and Data to Memory

- The binding of instructions and data to memory addresses i.e.
- Address binding of instructions and data to memory addresses can happen at three different stages
  - **Compile time**
  - **Load time**
  - **Execution time**

Binding instructions & data to memory

- Can be done at any step during execution based on available info
- If known at Compile time:
  - generates **absolute code** that resides at given memory.
  - Re-compile if the starting location changes
- If known at Load time but not at compile time:
  - Generates relocatable code
  - Final binding is delayed until load time
  - If starting address changes, only reload the user code to incorporate the changed value
- If known at Execution time
  - If process can be moved from one memory segment to another during execution
  - Binding is delayed until run time
  - Needs special hardware

Binding of Instructions and Data to Memory

- **Compile time**:
  - **If you know at compile time where the process will reside in memory, then absolute code can be generated.**
  - **If memory location known a priori,**
  - For example, if you know that a user process will reside starting at location *R,* then the generated compiler code will start at that location and extend up from there.
- Must recompile code if starting location changes-
  - **If, at some later time, the starting location changes, then it will be necessary to recompile this code.**

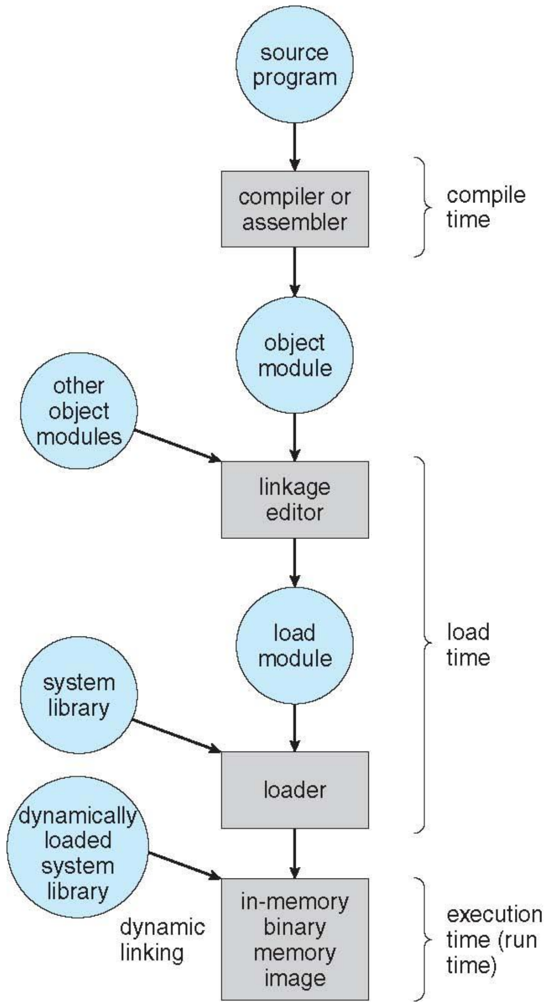
Binding of Instructions and Data to Memory

- **Load time**:
- Must generate **relocatable code** if memory location is not known at compile time
- **If it is not known at compile time where the process will reside in memory, then the compiler must generate relocatable code**
- **In this case, final binding is delayed until load time.**
- If the starting address changes, we need **only reload the user code to incorporate this changed value.**

Binding of Instructions and Data to Memory

- **Execution time**:
- If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time.
- Need hardware support for address maps (e.g., base and limit registers)

Multistep Processing of a User Program



Logical Vs physical address space

- Logical address: An address generated by CPU
- Physical address : an address loaded into memory address register of memory
- The compile time and load time address-binding methods generate identical logical and physical address
- But, execution time binding results in different logical and physical address space
    - Logical address space: set of all addresses generated by a program
    - Physical address space: set of all physical addresses corresponding to all logical addresses

Logical vs. Physical Address Space

- **Logical address** –
  - An address generated by the CPU
  - also referred to as **virtual address**
- **Physical address** –
  - Address seen by the memory unit
  - that is, the one loaded into memory address register of the memory
- **Logical address space-** the set of all logical addresses generated by a program
- **Physical address space-** the set of all physical addresses corresponding to these logical addresses

Logical vs. Physical Address Space

- **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**
- Thus, in the execution-time address-binding scheme, the logical and physical address spaces differ.

Memory-Management Unit (MMU)

- The run-time mapping from virtual to physical addresses is done by a hardware device called the MMU
- Many methods possible for this mapping

MMU

- MMU (memory management unit) maps virtual addresses to physical addresses
- Simple MMU scheme: generalization of base-register scheme
- Base register is called relocation register
- While sending to memory, relocation register value is added to every address generated by user process

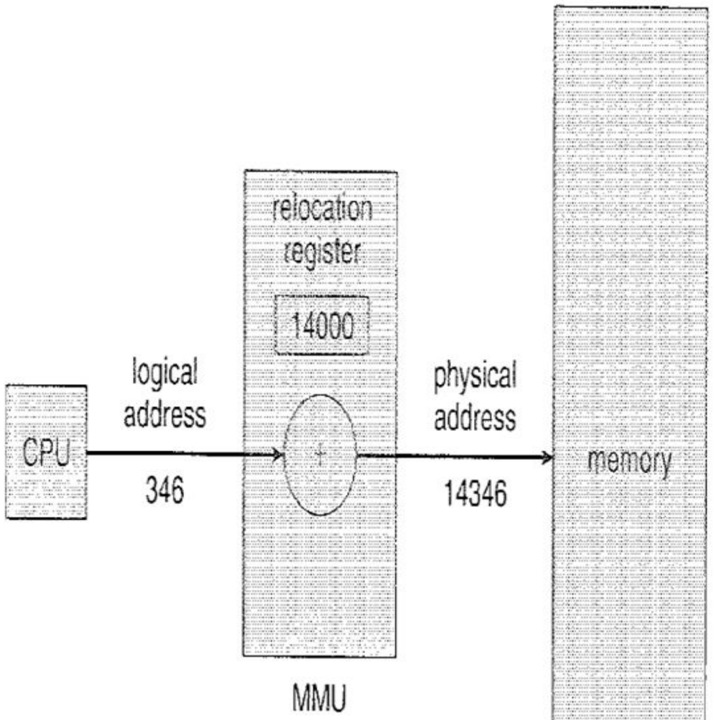
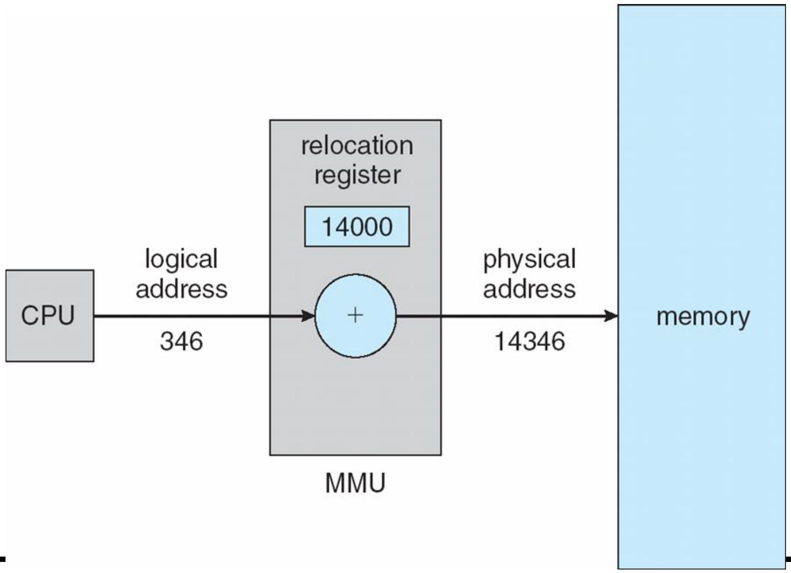


Figure 8.4 Dynamic relocation using a relocation register.

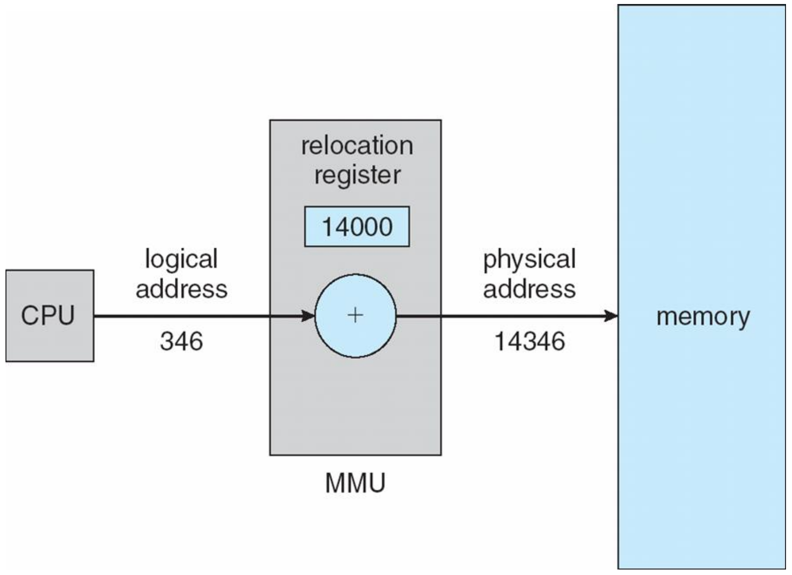
Memory-Management Unit (MMU)

- To start, consider simple scheme –
- 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



Memory-Management Unit (MMU)

- For example,
- if the base is at 14000,
- then an attempt by the user to address location 0 is dynamically relocated to location 14000;
- an access to location 346 is mapped to location 14346.



- The user program never sees the ‘real’ physical address.
- User program deals with logical addresses
- The memory mapping hardware converts logical addresses to physical addresses
- Logical addresses: Range is 0 to max
- Physical addresses: R+0 to R+max where R is base value

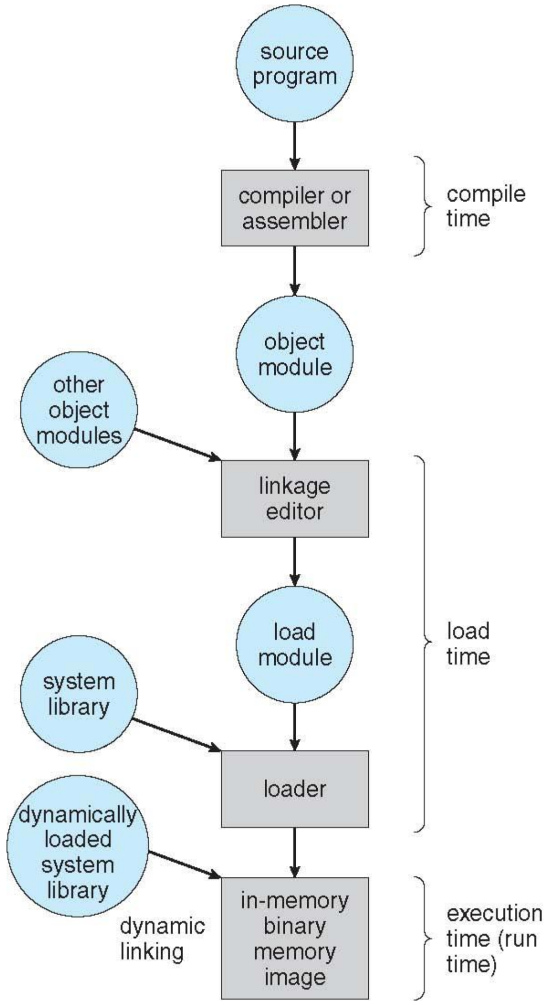
Dynamic Loading

- So far:-
- The entire program and all data of a process to be in physical memory for the process to execute.
- **The size of a process has thus been limited to the size of physical memory.**

Dynamic Loading

- To obtain better memory-space utilization,
- we can use Dynamic Loading

Multistep Processing of a User Program



Dynamic Loading

- Routine is not loaded until it is called
- All routines kept on disk in relocatable load format

1) The main program is loaded into memory and is executed.
2) When a routine needs to call another routine, the calling routine first checks to see whether the other routine has been loaded.
3) If it has not, the relocatable linking loader is called to load the desired routine into memory and to update the program's address tables to reflect this change.
4) Then control is passed to the newly loaded routine.

Dynamic Loading

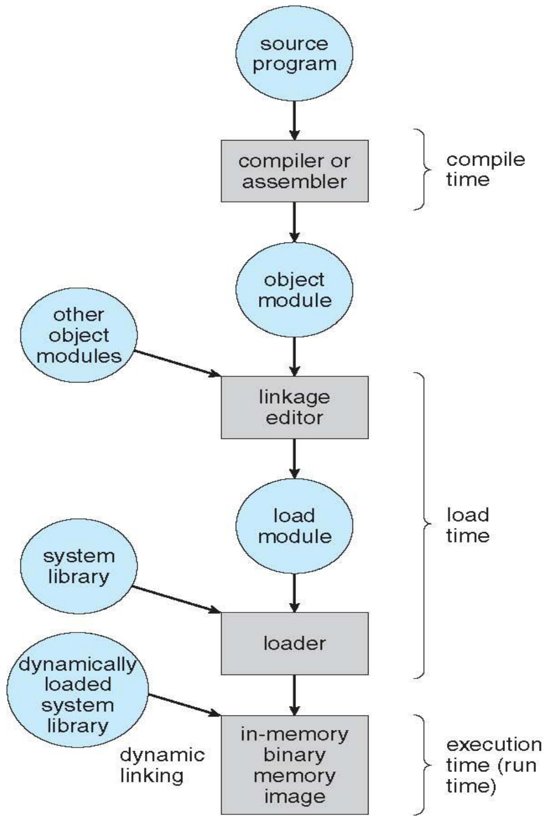
Advantage –

- Unused routine is never loaded.
- Useful when large amounts of code are needed to handle infrequently occurring cases. Eg- Error Routines
- No special support from the operating system is required

- Implemented through program design-
  - It is the responsibility of the users to design their programs to take advantage of such a method.
- Operating systems may help the programmer,
  - by providing library routines to implement dynamic loading.

Dynamic Linking

- **Static linking** –
- System libraries and program code combined by the linker/loader into the binary program image
- Dynamic linking –linking postponed until execution time



Dynamic Linking

- Dynamic linking –
- This feature is usually used with system libraries, such as language subroutine libraries.

- Without this facility, each program on a system must include a copy of its language library (or at least the routines referenced by the program) in the executable image.
- This requirement wastes both disk space and main memory.

Dynamic Linking

- A *stub* is included in the image for each library routine reference.
- The stub is a small piece of code that indicates how to locate the appropriate memory-resident library routine or how to load the library if the routine is not already present.

Dynamic Linking

- When the stub is executed,

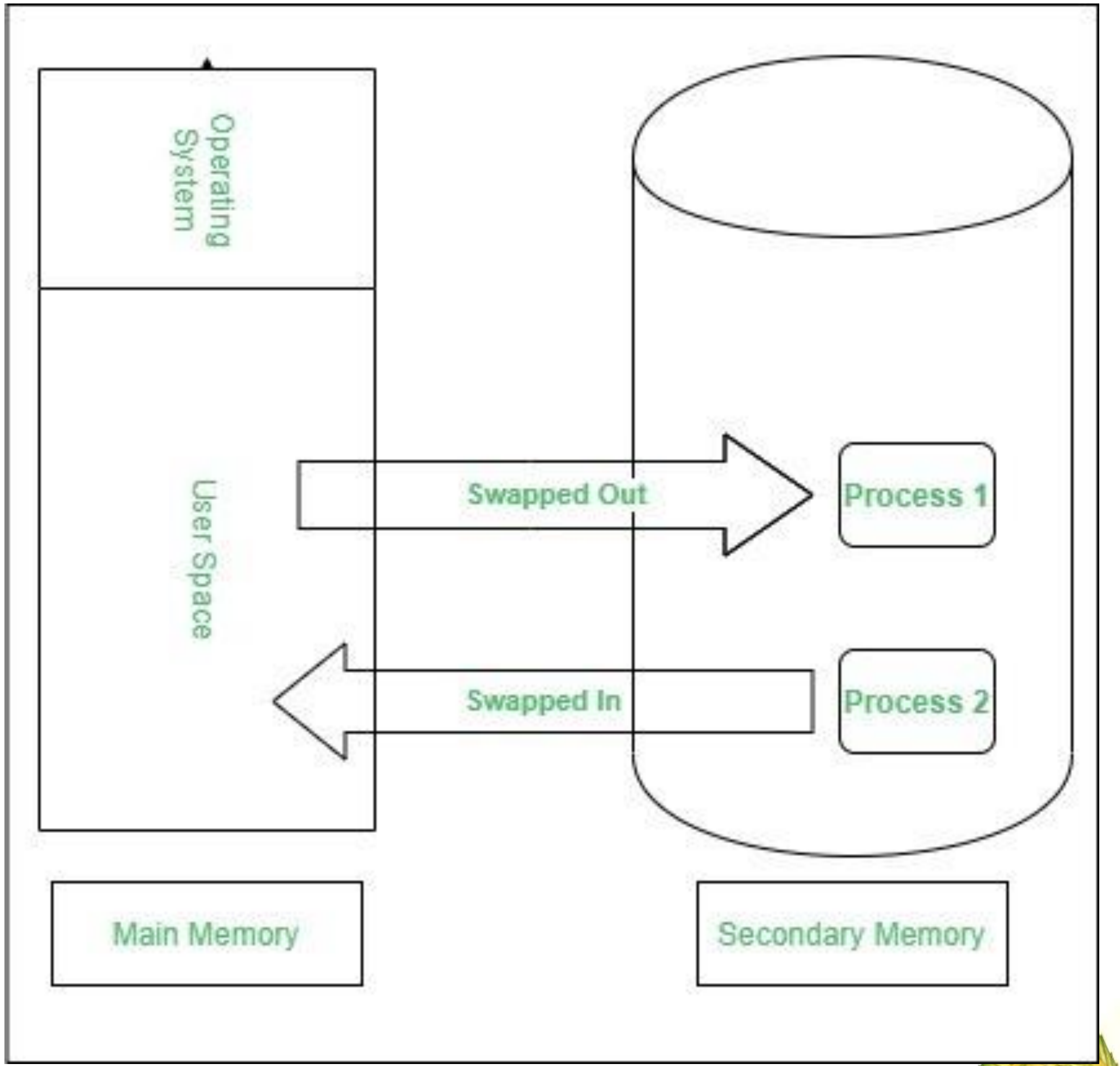
1) It checks to see whether the needed routine is already in memory.
2) If it is not, the program loads the routine into memory.
3) Either way, the stub replaces itself with the address of the routine and executes the routine.
4) Thus, the next time that particular code segment is reached, the library routine is executed directly, incurring no cost for dynamic linking.

Dynamic Linking

- This feature can be extended to library updates (such as bug fixes).
- A library may be replaced by a new version,
- All programs that reference the library will automatically use the new version. Without dynamic linking, all such programs would need to be relinked to gain access to the new library.

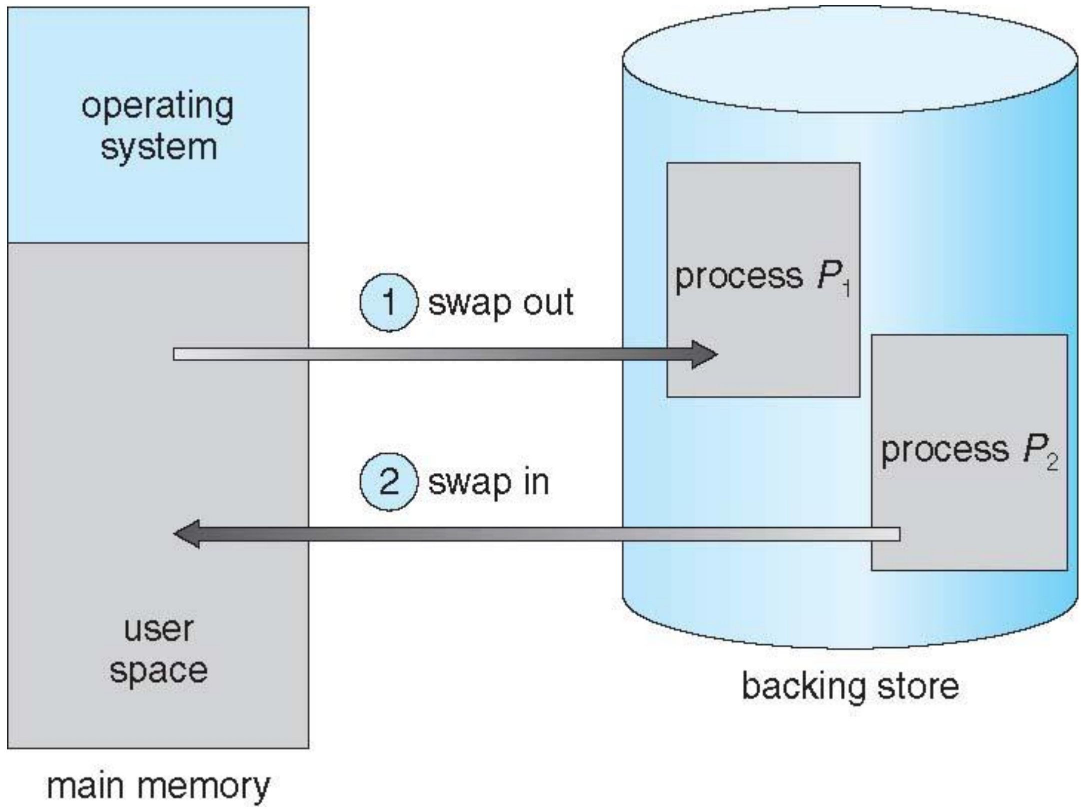
Swapping

- Swapping is a process of swapping a process temporarily to a secondary memory from the main memory



Swapping

- A process can be **swapped** temporarily out of memory to a backing store, and then brought back into memory for continued execution



Swapping

- **Backing store** –
- fast disk large enough to accommodate copies of all memory images for all users;
- must provide direct access to these memory images

Swapping

Q
MMU /Dispatcher

For example,

- Assume a multiprogramming environment with a round-robin CPU-scheduling algorithm.

1) When a quantum expires, the memory manager will start to swap out the process that just finished and to swap another process into the memory space that has been freed
2) In the meantime, the CPU scheduler will allocate a time slice to some other process in memory.
3) When each process finishes its quantum, it will be swapped with another process.
4) Ideally, the memory manager can swap processes fast enough that some processes will be in memory, ready to execute, when the CPU scheduler wants to reschedule the CPU.

Swapping

- **Roll out, roll in** –
- A variant of this swapping policy is used for priority-based scheduling algorithms.

1) If a higher-priority process arrives and wants service,
2) The memory manager can swap out the lower-priority process and then load and execute the higher-priority process.
3) When the higher-priority process finishes, the lower-priority process can be swapped back in and continued.

- This variant of swapping is sometimes called roll out, roll in.

Swapping

- 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

Swapping (Cont.)

- Does the swapped out process need to swap back in to same physical addresses?

Swapping

- Normally, a process that is swapped out will be swapped back into the same memory space it occupied previously.
- The restriction is dictated by the method of address binding.
- If binding is done at assembly/Compile or load time, then the process cannot be easily moved to a different location.
- if execution time binding is being used, then a process can be swapped into a different memory space as the physical addresses are computed at execution time

Dispatcher and swapper

- Swapping requires a backing store.
- The backing store is commonly a fast disk, large enough to accommodate copies of all memory images for all users
- Provides direct access to these memory images.
- **The system maintains a ready queue of all processes whose memory images are in memory and are ready to run.**
- Whenever the CPU scheduler decides to execute a process, it calls the dispatcher.
- The dispatcher checks to see whether the next process in the queue is in memory.
- If it is not, and if there is no free memory region, the swapper swaps out a process currently in memory and swaps in the desired process.
- It then reloads registers and transfers control to the selected process.

- The context-switch time in such a swapping system is fairly high.
- Assume a user process of 10 MB size and the backing store is a standard hard disk with a transfer rate of 40 MB per second. The actual transfer of the 10-MB process to or from main memory takes:

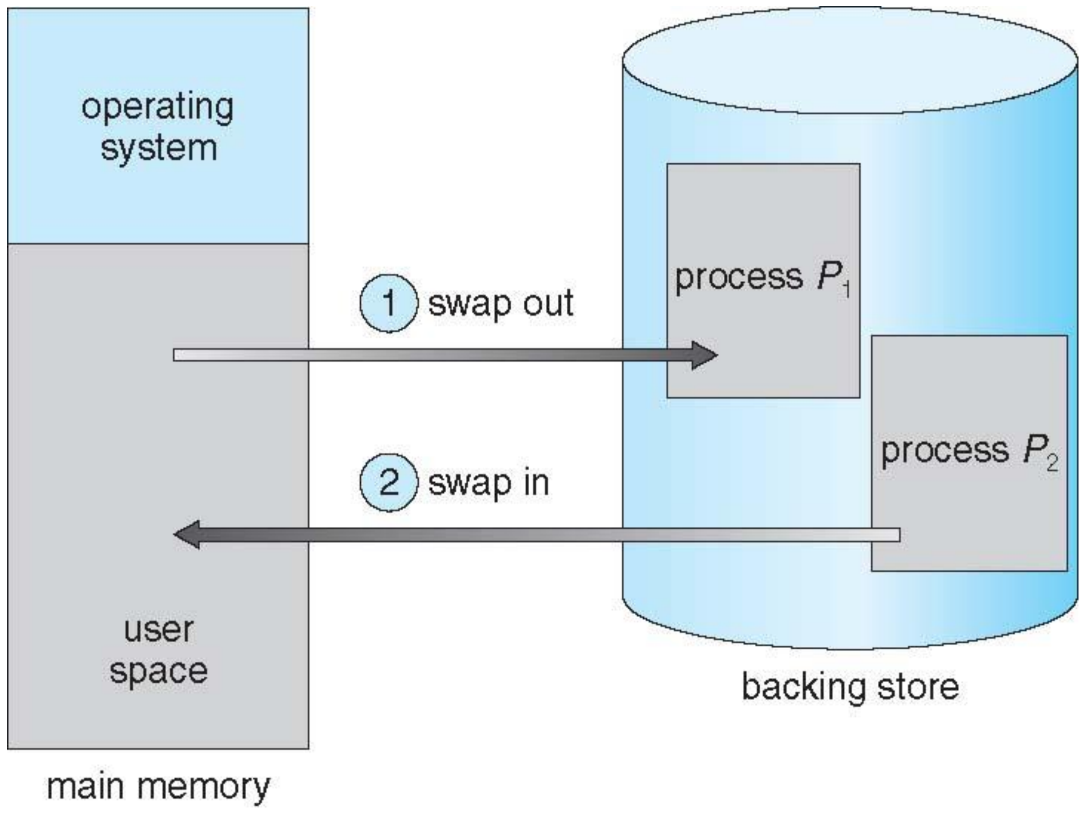
$$\begin{aligned} 10000 \text{ KB}/40000 \text{ KB per second} &= 1/4 \text{ second} \\ &= 250 \text{ milliseconds.} \end{aligned}$$

- Assuming that no head seeks are necessary, and assuming an average latency of 8 milliseconds, the swap time is 258 milliseconds.
- Considering both swap out and swap in, the total swap time = 516 milliseconds.
- For efficient CPU utilization, the execution time for each process should be long relative to the swap time.
- Thus, in a round-robin CPU-scheduling algorithm, for example, the time quantum should be substantially larger than 0.516 seconds.

Swapping (Cont.)

- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
  - Swapping normally disabled
  - Started if more than threshold amount of memory allocated
  - Disabled again once memory demand reduced below threshold

Schematic View of Swapping



Contiguous Allocation

- Main memory must support both OS and user processes
- Limited resource, must allocate efficiently
- Contiguous allocation per process is one early method

Memory allocation to OS and User programs

- The main memory must accommodate both the operating system and the various user processes.
- Hence, OS needs to allocate the parts of the main memory in the most efficient way possible.
- The memory is divided into two partitions:
  - For the resident operating system and
  - For the user processes.
- So the operating system in either low memory or high memory.
- The major factor affecting this decision is the location of the interrupt vector.
- Since the interrupt vector is often in low memory, programmers usually place the operating system in low memory as well.
- User processes then held in high memory

Contiguous Allocation

- In contiguous memory allocation, each process is contained in a single contiguous section of memory.

Memory Mapping and Protection

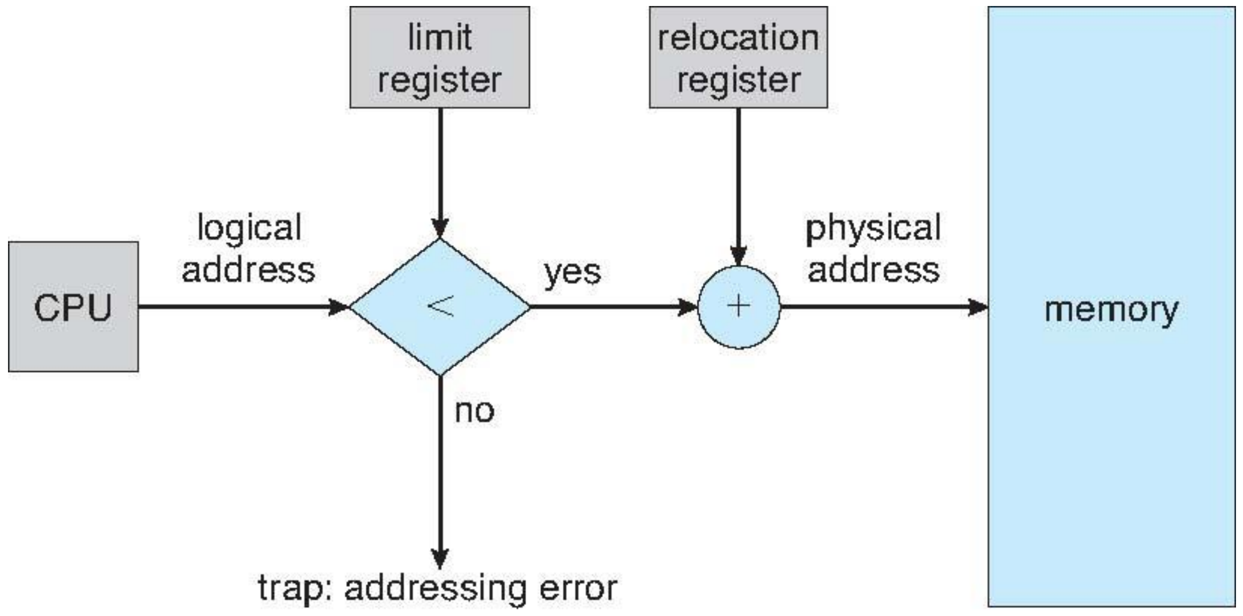
- To protect user processes from each other, and from changing operating-system code and data by using a relocation register together with a limit register.

1) The relocation register contains the value of the smallest physical address
2) The limit register contains the range of logical addresses – each logical address must be less than the limit register

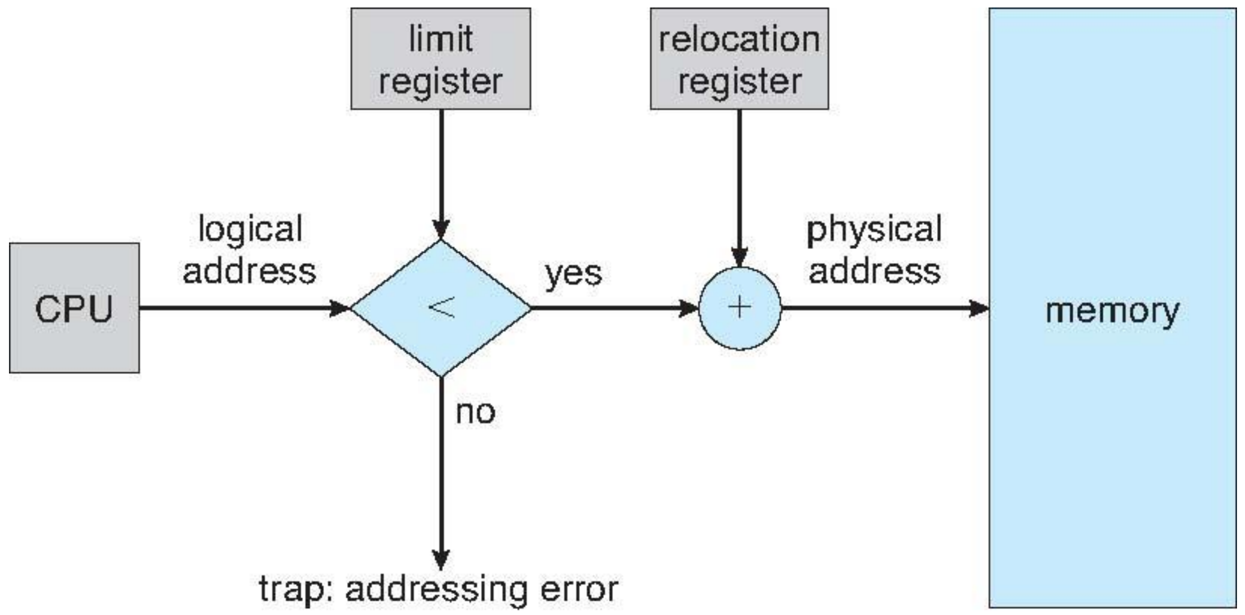
Memory Mapping and Protection

1) **Each logical address must be less than the limit register;**
2) **The MMU maps the logical address *dynamically* by adding the value in the relocation register.**
3) **This mapped address is sent *to* memory**

(for example, relocation= 100040 and limit= 74600).



Hardware Support for Relocation and Limit Registers



Types of Memory Management Techniques

- Contiguous
- Non-Contiguous

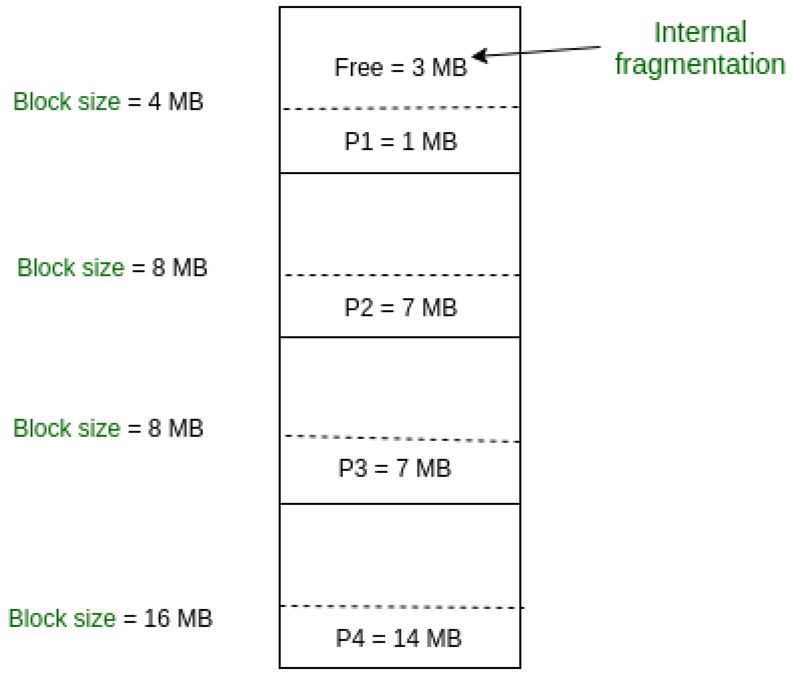
- In Contiguous Technique, executing process must be loaded entirely in the main memory.
- The contiguous Technique can be divided into:
  - Fixed (or static) partitioning
  - Variable (or dynamic) partitioning

Fixed partitioning

- Also known as static partitioning,
- **No of Partitions are fixed**
- **In this partitioning, the number of partitions (non-overlapping) in RAM is fixed but the size of each partition may or may not be the same.**

Fixed partitioning

- Is a memory allocation technique used in operating systems to divide the physical memory into fixed-size partitions or regions, each assigned to a specific process or user.
- **Partitions are made before execution or during system configure.**
- Each partition remains dedicated to a specific process until it terminates or releases the partition.



Fixed size partition

Fixed partitioning

- Disadvantages:
- **It can lead to internal fragmentation, where memory in a partition remains unused.** This can happen when the process's memory requirements are smaller than the partition size, leaving some memory unused

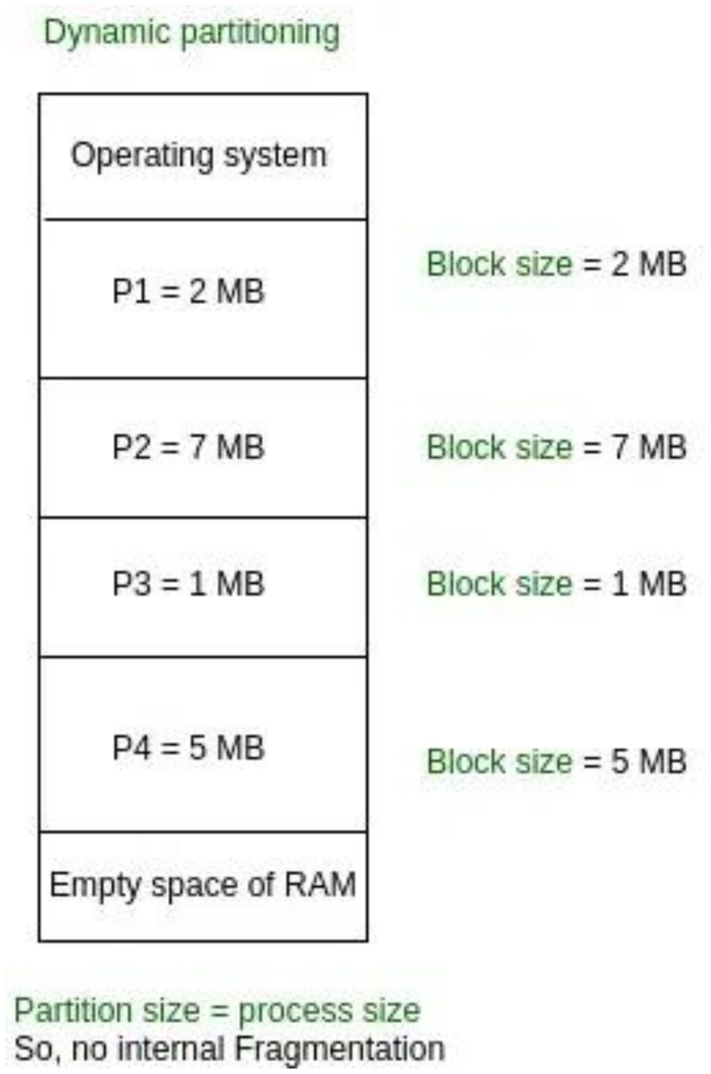
Variable (or Dynamic) Partitioning

- In contrast with fixed partitioning, partitions are not made before the execution or during system configuration.
- **No of partitions are not fixed**

Variable (or Dynamic) Partitioning

- **Features**:
- Initially, RAM is empty and partitions are made during the run-time according to the process's need instead of partitioning during system configuration.
- The size of the partition will be equal to the incoming process.
- The partition size varies according to the need of the process so that internal fragmentation can be avoided to ensure efficient utilization of RAM.
- The number of partitions in RAM is not fixed and depends on the number of incoming processes and the Main Memory's size.

Variable (or Dynamic) Partitioning



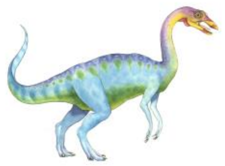
Non-Contiguous Memory Allocation

- In this technique, each process is allocated a series of non-contiguous blocks of memory that can be located anywhere in the physical memory.

- Paging and Segmentation are the two ways that allow a process’s physical address space to be non-contiguous. It has the advantage of reducing memory wastage but it increases the overheads due to address translation. It slows the execution of the memory because time is consumed in address translation.

Memory Allocation-Fixed Size Partitions

- One of the simplest methods for allocating memory is –
- To divide memory into several **fixed-sized partitions.**
- Each partition may contain exactly one process.
- Thus, the degree of multiprogramming is bound by the number of partitions.
- In this Multiple partition method, when a partition is free, a process is selected from the **input queue** and is loaded into the free partition.
- When the process terminates, the partition becomes available for another process.
- This method was originally used by the IBM OS/360 operating system (called MFT); it is no longer in use.



Memory Allocation

- **In the Variable Partition scheme,**
- The operating system keeps a table indicating which parts of memory are available and which are occupied.
- Initially, all memory is available for user processes and is considered one large block of available memory-**a Hole.**
- Eventually as you will see, memory contains a set of holes of various sizes.

“Hole – block of available memory; holes of various size are scattered throughout memory”

Memory Allocation-Fixed Size Partitions

- As processes enter the system, they are put into an input queue.
- At any given time, then, we have a list of available block sizes and an input queue.
- The operating system can order the input queue according to a scheduling algorithm.
  - Process selected from input queue is allocated memory from a hole large enough to accommodate it
  - The operating system can wait until a large enough block is available,
  - or it can skip down the input queue to see whether the smaller memory requirements of some other process can be met.

Memory Allocation-Fixed Size Partitions

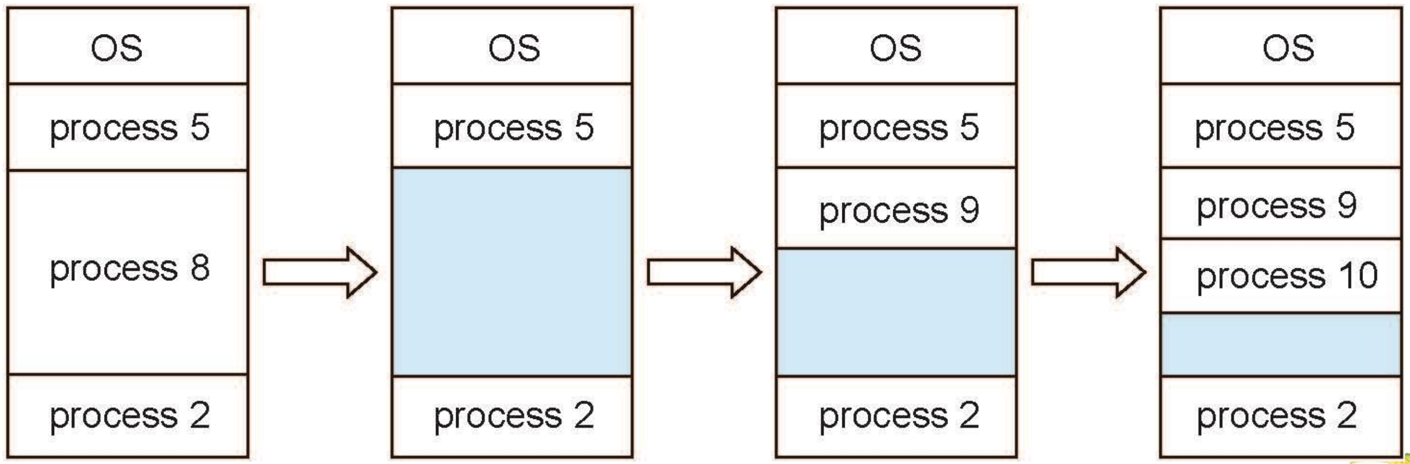
- The memory blocks available comprise a *set* of holes of various sizes scattered throughout memory.
- When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process.
  - If the hole is too large, it is split into two parts.
  - **One part is allocated to the arriving process;**
  - **the other is returned to the set of holes.**

Memory Allocation-Fixed Size Partitions

- When a process terminates, it releases its block of memory, which is then placed back in the set of holes.
  - **If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole.**
  - At this point, the system may need to check whether there are processes waiting for memory and whether this newly freed and recombined memory could satisfy the demands of any of these waiting processes.

Multiple-partition allocation

- **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**
- **Best-fit**
- **Worst-fit**

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
- Searching can start either at the beginning of the set of holes or at the location where the previous first-fit search ended.
- We can stop searching as soon as we find a free hole that is large enough.

Dynamic Storage-Allocation Problem

How to satisfy a request of size ***n*** from a list of free holes?

- **Best-fit**:
- Allocate the ***smallest*** hole that is big enough;
- We must search entire list, unless ordered by size
- This strategy produces the smallest leftover hole

Dynamic Storage-Allocation Problem

How to satisfy a request of size ***n*** from a list of free holes?

- **Worst-fit**:
- Allocate the ***largest*** hole;
- We must also search entire list unless it is sorted by size.
- This strategy produces the largest leftover hole which may be more useful than the smaller leftover hole from a best-fit approach.

Dynamic Storage-Allocation Problem

- Simulations have shown that

1) Both first fit and best fit are better than worst fit in terms of decreasing time and storage utilization.
2) Neither first fit nor best fit is clearly better than the other in terms of storage utilization, but first fit is generally faster.

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

- As processes are loaded and removed from memory, the free memory space is broken into little pieces

Fragmentation

Memory fragmentation can be-

- **External Fragmentation**
- **Internal Fragmentation**

Fragmentation

- **External Fragmentation** –
- Total memory space exists to satisfy a request, but it is not contiguous
- Storage is fragmented into a large number of small holes.
- This fragmentation problem can be severe
- In the worst case,
  - A block of free (or wasted) memory between every two processes.
  - If all these small pieces of memory were in one big free block instead, we might be able to run several more processes.

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
- Unused Memory Internal to a partition

Fragmentation

- **Internal Fragmentation** –
- Consider a multiple-partition allocation scheme with a hole of 18,464 bytes.
- Suppose that the next process requests 18,462 bytes.
- If we allocate exactly the requested block,
- We are left with a hole of 2 bytes.

Fragmentation

- 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
  - If addresses are relocated dynamically, relocation requires only moving the program and data and then changing the base register to reflect the new base address.

Fragmentation (Cont.)

- The simplest compaction algorithm is
  - To move all processes toward one end of memory;
  - all holes move in the other direction,
  - producing one large hole of available memory.
  - This scheme can be expensive.

Fragmentation (Cont.)

- Another possible solution-
  - To permit the logical address space of the processes to be non contiguous,
  - thus allowing a process to be allocated physical memory wherever such memory is available.

Non-Contiguous Memory Allocation

- In this technique, each process is allocated a series of non-contiguous blocks of memory that can be located anywhere in the physical memory.

- Paging and Segmentation are the two ways that allow a process’s physical address space to be non-contiguous. It has the advantage of reducing memory wastage but it increases the overheads due to address translation. It slows the execution of the memory because time is consumed in address translation.

Paging

- Paging is a memory-management scheme that permits the physical address space of a process to be non-contiguous
- 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

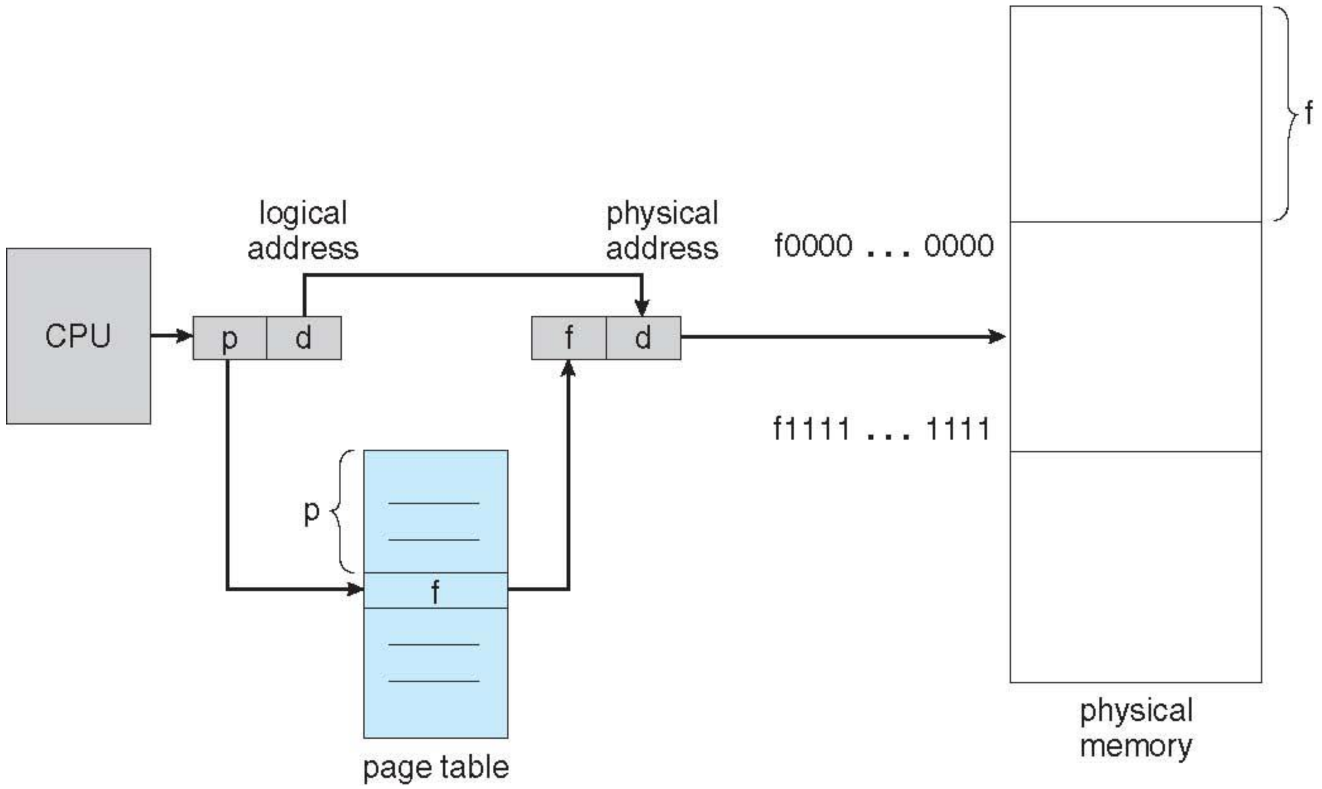
Paging

- 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

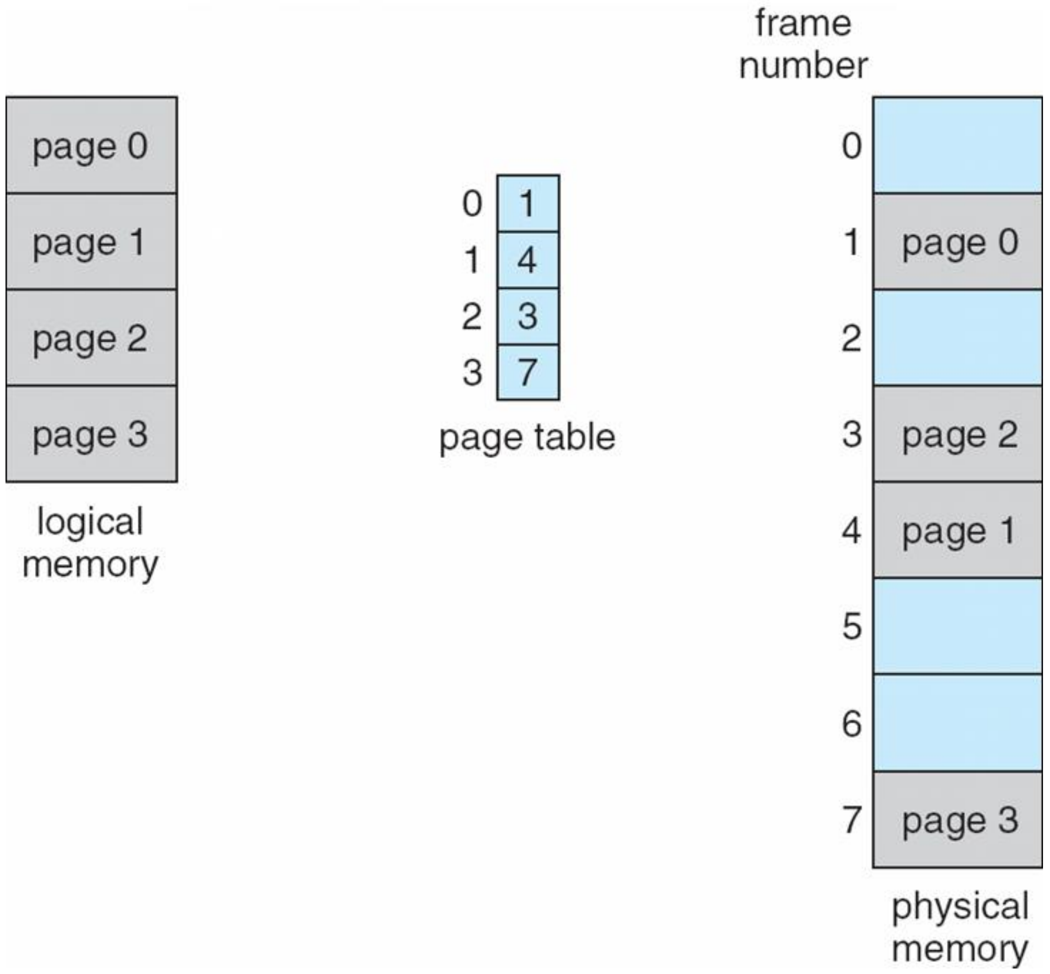
Address Translation Scheme

- Address generated by CPU is divided into:
  - **Page number** (***p***) – used as an index into a **page table**
  - **The 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

Paging Hardware



Paging Model of Logical and Physical Memory



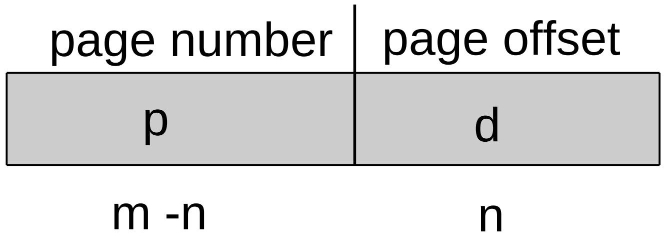
Paging (Cont.)

- The page size (like the frame size) is defined by the hardware.
- The size of a page is typically a power of 2,
- Varying between 512 bytes and 16 MB per page, depending on the computer architecture.
- The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset particularly easy.

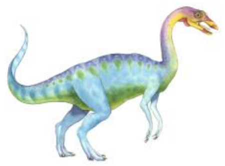
Paging (Cont.)

- If the size of the logical address space = 2^m, and
- a page size = 2^n addressing units (bytes or words)
- then the high-order *m- n* bits of a logical address designate the page number,
- the *n* low-order bits designate the page offset.

- where *p* is an index into the page table and
- *d* is the displacement within the page.

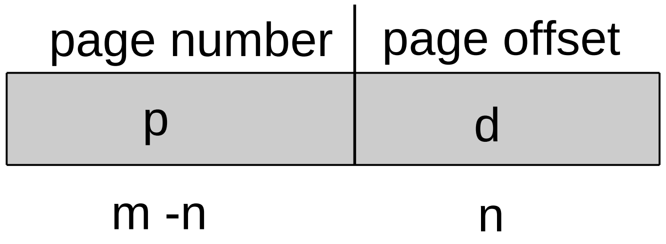


For given logical address space 2^m and page size 2^n



Paging (Cont.)

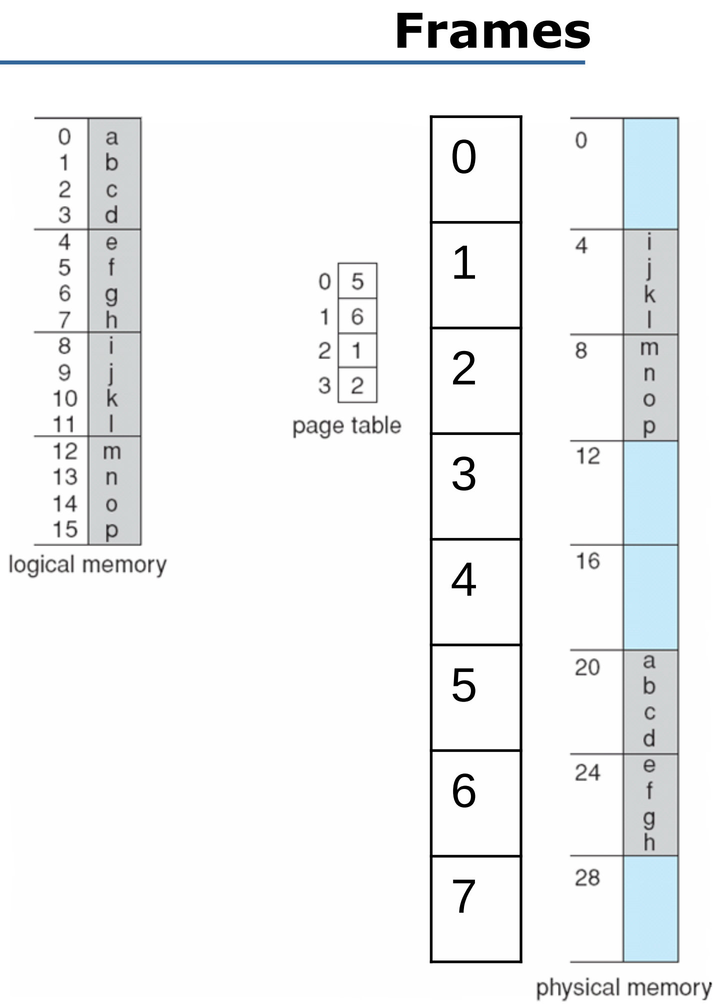
- In the logical address, *n*= 2 and *m* = 4
- If the size of the logical address space = 2^m, = 2^4 and
- So, **a page size = 2^n addressing units (bytes or words) = 2^2 bytes**
- Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages),
- We show how the user's view of memory can be mapped into physical memory



Paging Example

1) Logical address 0 is page 0, offset 0. Indexing into the page table, we find that page 0 is in frame 5.
Thus, logical address 0 maps to physical address 20 [= (5 x 4) + 0].

*Using a page size of 4 bytes



Logical memory:

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

Page table:

0	5
1	6
2	1
3	2

Physical memory:

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

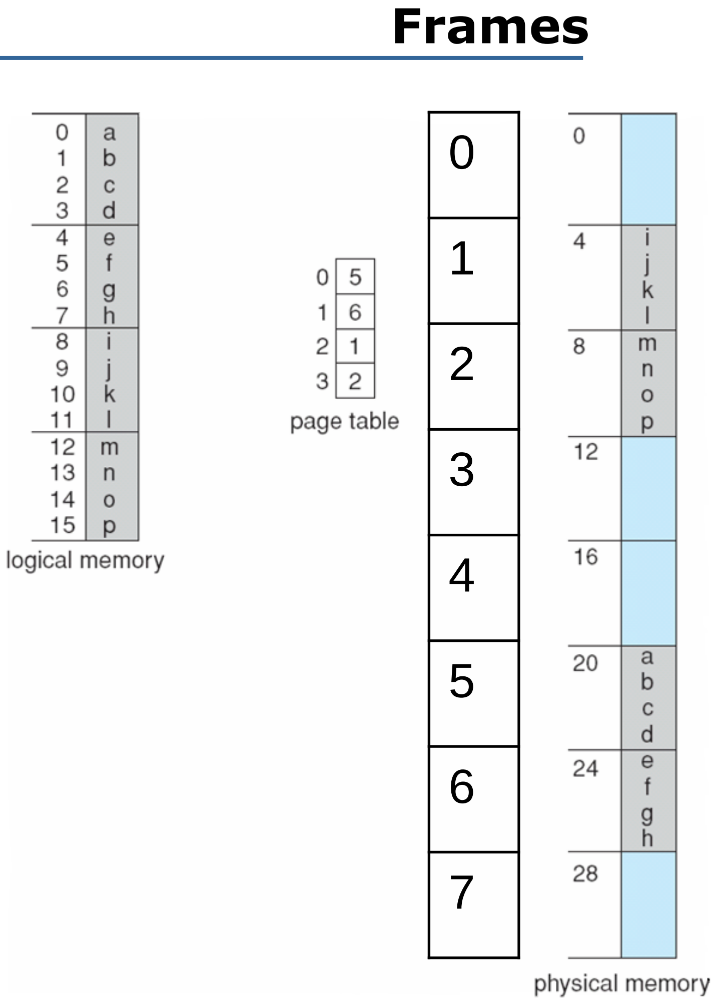
n=2 and *m*=4   32-byte memory and 4-byte pages

Paging Example

2) Logical address 3 (page 0, offset 3), we find that page 0 is in frame 5, maps to physical address 23 [= (5 x 4) + 3].

3) Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6. Thus, logical address 4 maps to physical address 24 [= (6 x 4) + O].

4) Logical address 13 (page 3, offset 1) according to the page table, page 3 is mapped to frame 2. Thus, logical address 4 maps to physical address 9 [= (2 x 4) + 1].



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

Paging (Cont.)

- When we use a paging scheme,
- We have no external fragmentation: *any* free frame can be allocated to a process that needs it.
- We may have some internal fragmentation.

Paging (Cont.)

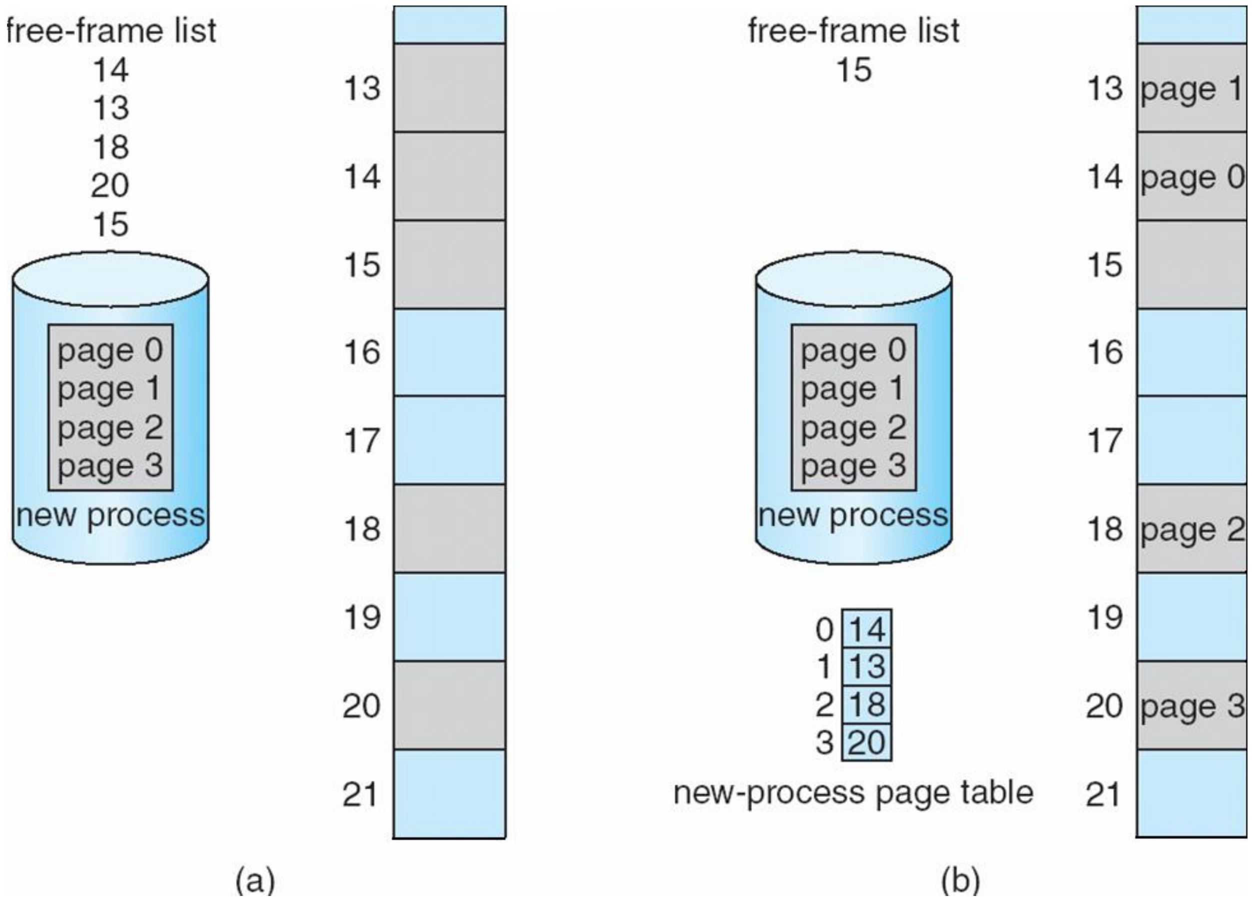
- Calculating internal fragmentation
  - Page size = 2,048 bytes
  - Process size = 72,766 bytes
  - 35 pages + 1,086 bytes
  - Internal fragmentation of 2,048 - 1,086 = 962 bytes

- Worst case fragmentation
  - A process would needs n pages plus 1 byte. It
  - would be allocated n+ 1 frames
  - resulting in internal fragmentation of almost an entire frame.

Paging (Cont.)

- Process view and physical memory now very different
- By implementation process can only access its own memory

Free Frames

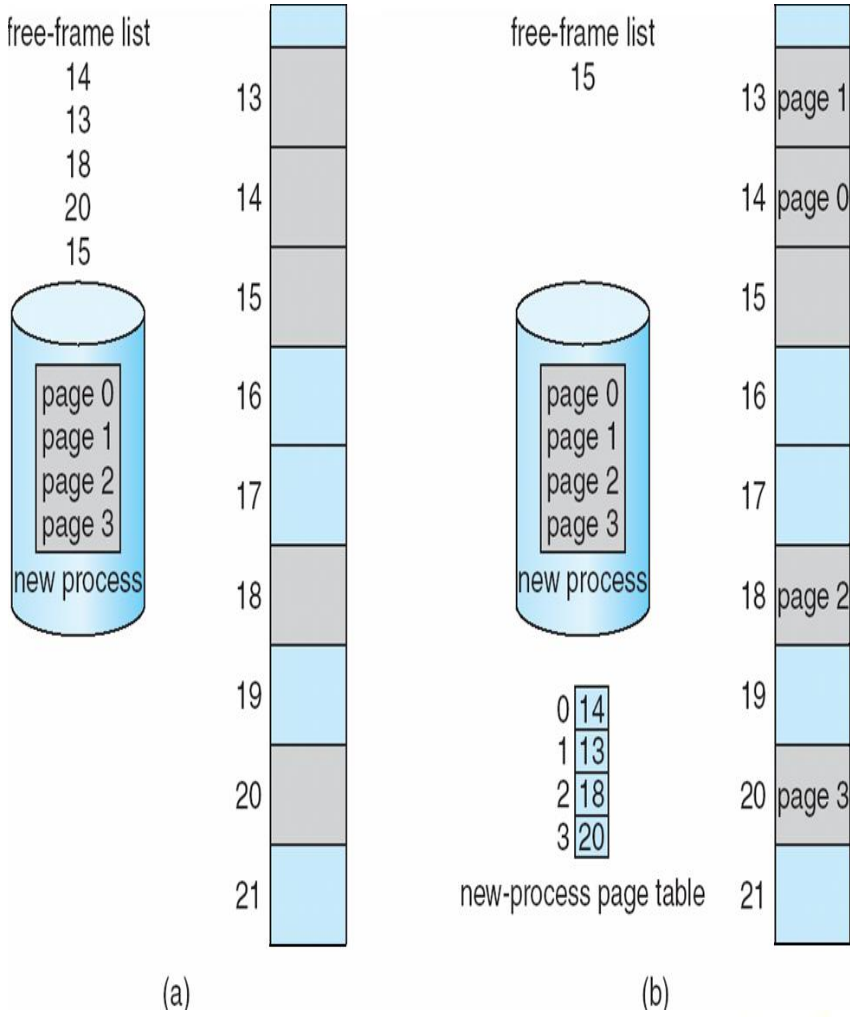


Before allocation

After allocation

Free Frames

- When a process arrives in the system to be executed, its size, expressed in pages, is examined.
- Each page of the process needs one frame.
- Thus, if the process requires *n* pages, at least *n* frames must be available in memory.

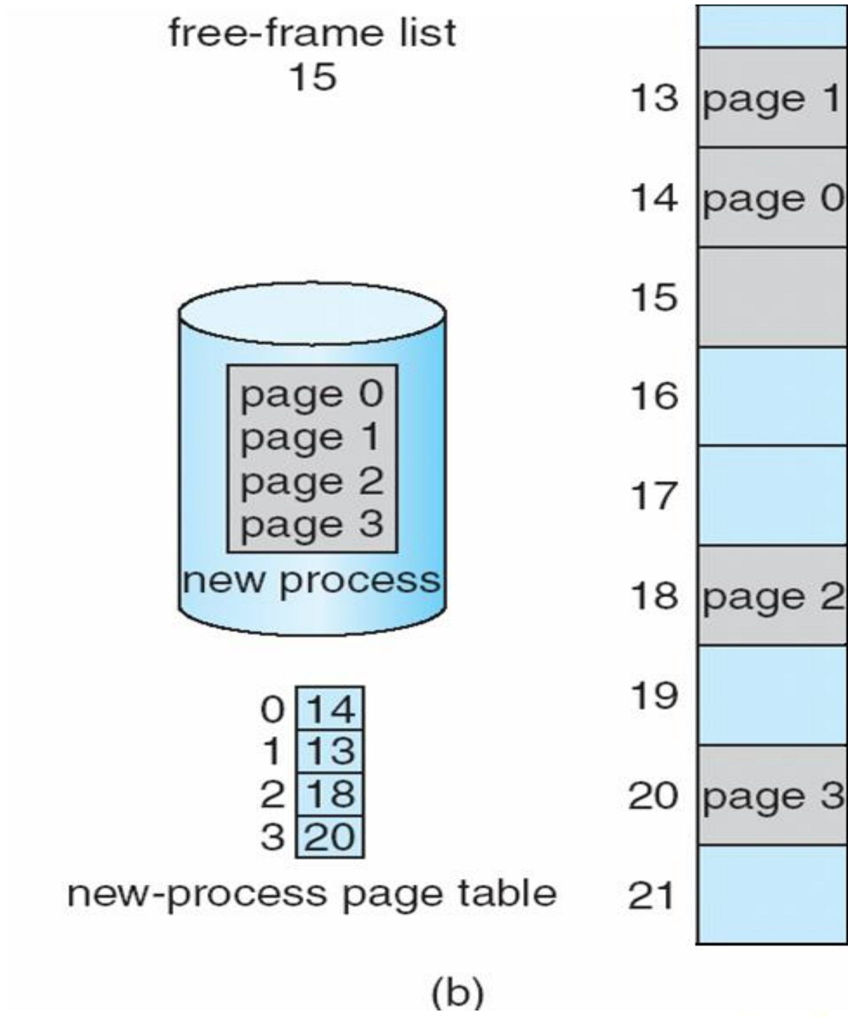


Before allocation

After allocation

Free Frames

- If *n* frames are available, they are allocated to this arriving process.
- The first page of the process is loaded in one of the allocated frames,
- The frame number is put in the page table for this process.
- The next page is loaded into another frame, its frame number is put into the page table, and so on



After allocation

Before allocation

Paging

- An important aspect of paging
- Clear separation between the user's view of memory and the actual physical memory.
- The user program views memory as one single space, containing only this one program.
- In fact, the user program is scattered throughout physical memory, which also holds other programs.

Paging

- The difference between the user's view of memory and the actual physical memory is reconciled by the address-translation hardware.
- The logical addresses are translated into physical addresses.
- This mapping is hidden from the user and is controlled by the operating system.

Hardware Support

- Each operating system has its own methods for storing page tables.
- Most allocate a page table for each process.
- A pointer to the page table is stored with the other register values (like the instruction counter) in the **process control block.**

PCB-MODULE 2

- When the dispatcher is told to start a process, it must reload the user registers and define the correct hardware page-table values from the stored user page table.

Implementation of Page Table

- The hardware implementation of the page table can be done in several ways.

1) **In the simplest case, the page table is implemented as a set of dedicated registers.**

- These registers should be built with very high-speed logic to make the paging-address translation efficient.
- The use of registers for the page table is satisfactory if the page table is reasonably small (for example, 256 entries).
- Most contemporary computers, however, allow the page table to be very large (for example, 1 million entries).
- **For these machines, the use of fast registers to implement the page table is not feasible.**

Implementation of Page Table

2) **Page table is kept in main memory**

- **Page-table base register** (**PTBR**) points to the page table
- Changing page tables requires changing only this one register
- **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**)

What is TLB?

3) Use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers** (**TLBs**)

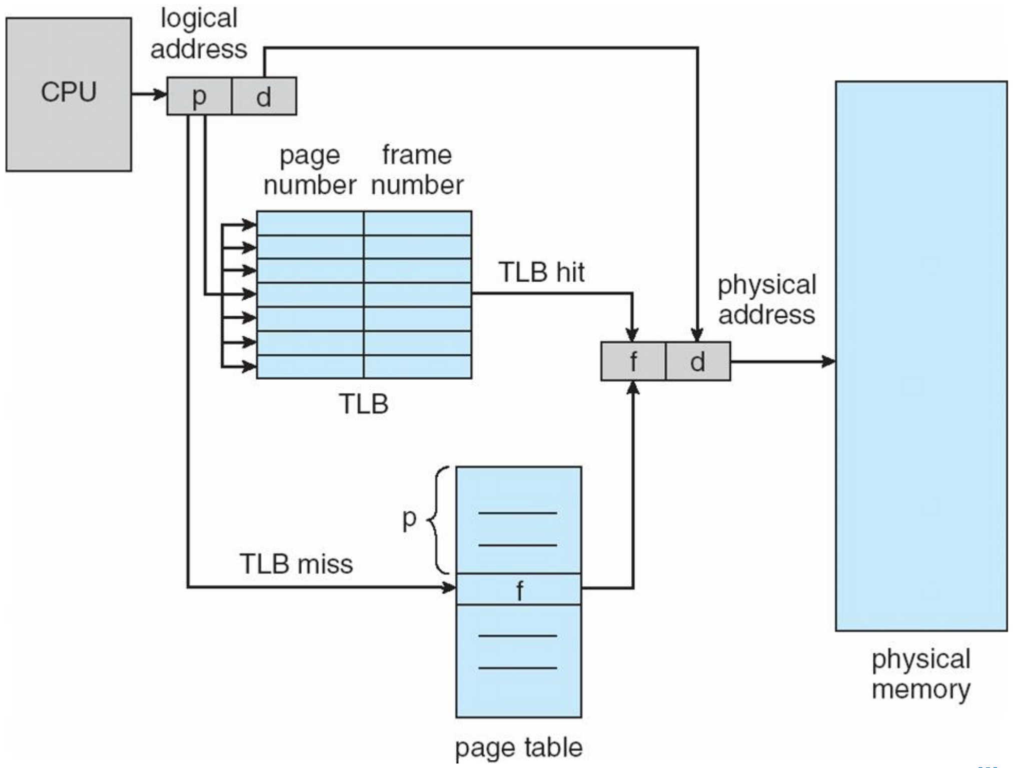
- **The TLB is associative, high-speed memory.**
- Each entry in the TLB consists of two parts:
- A key (or tag) and a value.

What is TLB?

- When the associative memory is presented with an item,
- The item is compared with all keys simultaneously. If the item is found, the corresponding value is returned.
- **The search is fast; the hardware, however, is expensive.**
- Typically, the number of entries in a TLB is small, often numbering between 64 and 1,024.

How does TLB work?

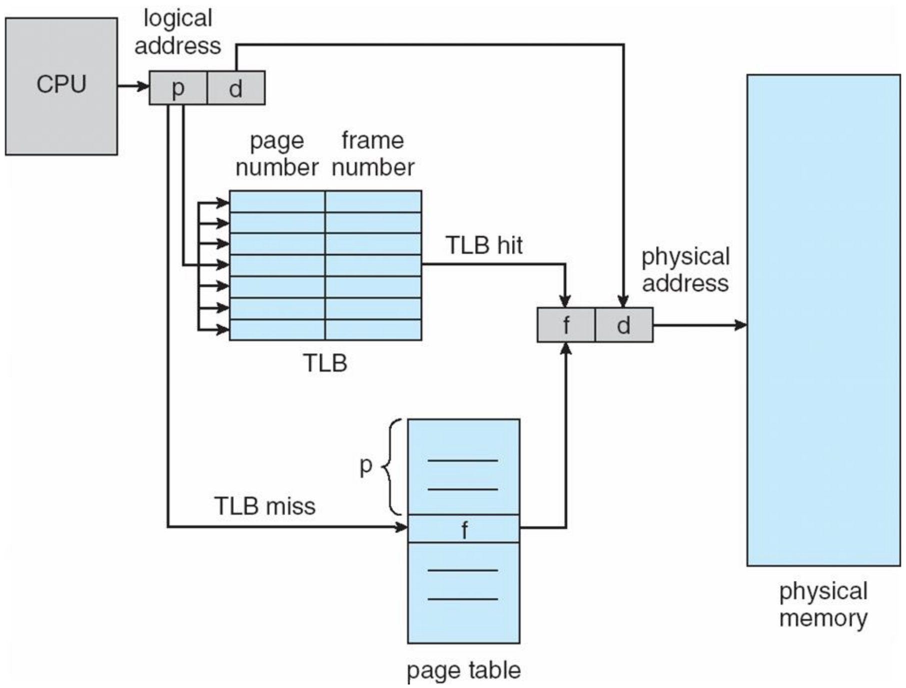
- The TLB contains only a few of the page-table entries.
- When a logical address is generated by the CPU, its page number is presented to the TLB.
- If the page number is found,
- **=TLB Hit,** Its frame number is immediately available and is used to access memory.



How does TLB work?

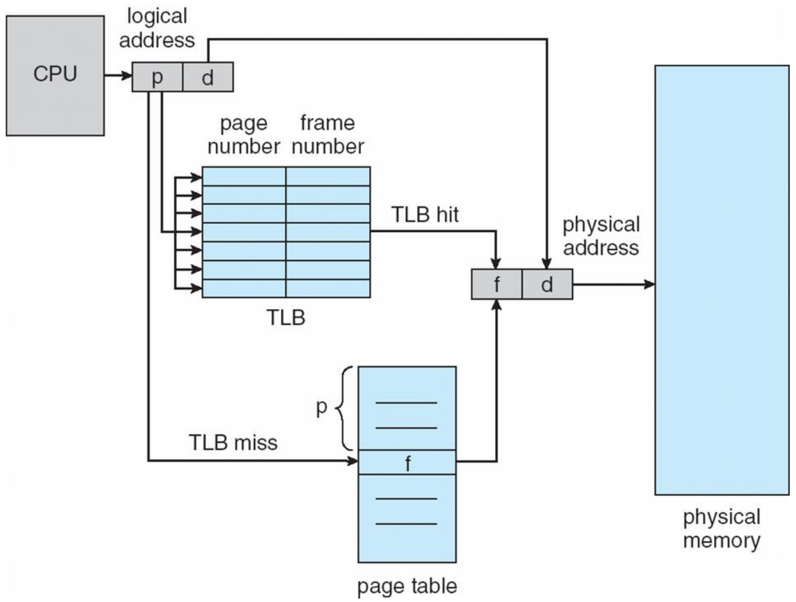
- If the page number is not in the TLB

1) **=TLB miss**,
2) Reference to the **page table** must be made.
3) When the frame number is obtained, we can use it to access memory .
4) In addition, we add the page number and frame number to the TLB, so that they will be found quickly on the next reference.



How does TLB work?

- If the TLB is already full of entries, the operating system must select one for replacement.
- Replacement policies range from least recently used (LRU) to random.

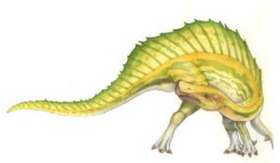


Associative Memory

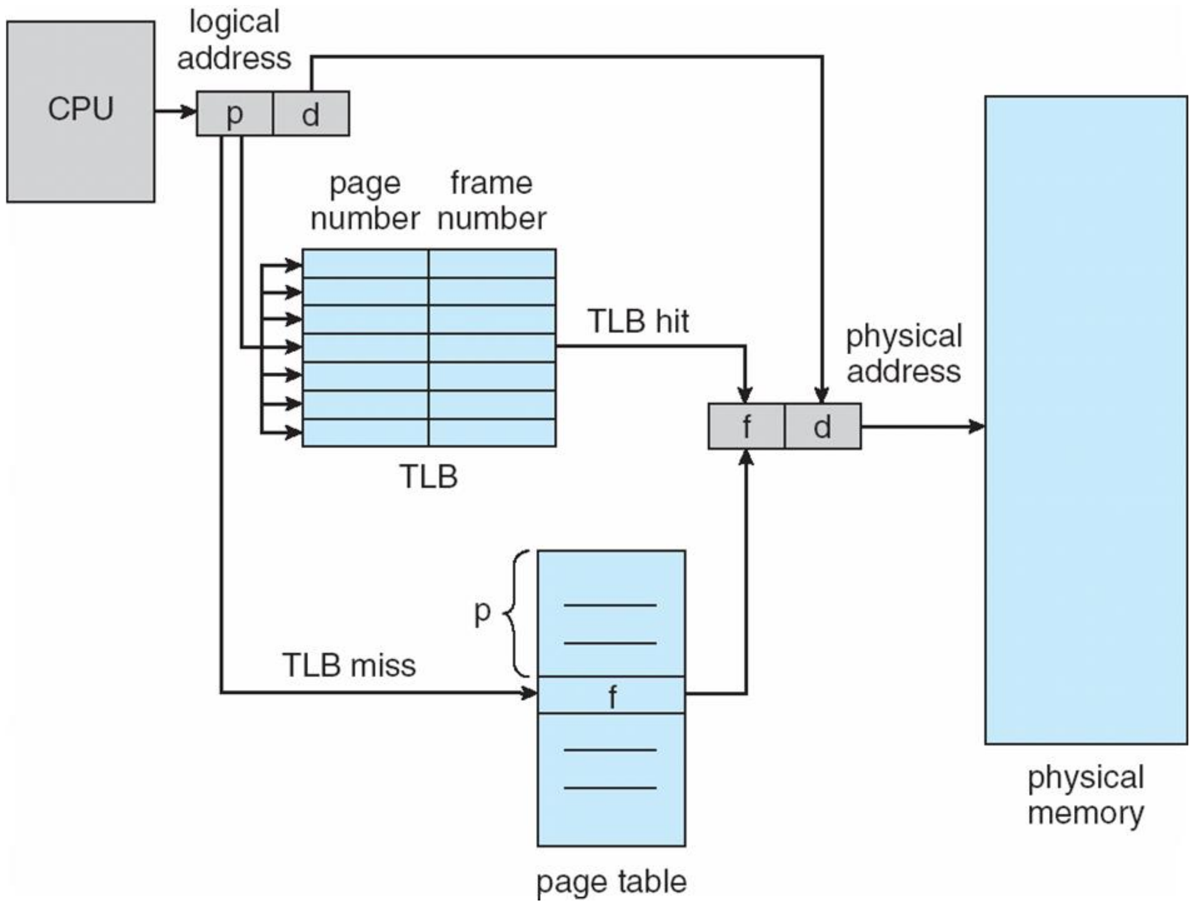
- Associative memory – parallel search

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



Implementation of Page Table (Cont.)

- Some TLBs store **address-space identifiers** (**ASIDs**) in each TLB entry –
- An ASID Uniquely identifies each process and provides address-space protection for that process

1) When the TLB attempts to resolve virtual page numbers, it ensures that the ASID for the currently running process matches the ASID associated with the virtual page.
2) If the ASIDs do not match, the attempt is treated as a TLB miss.

- In addition to providing address-space protection, an ASID allows the TLB to contain entries for several different processes simultaneously.

Effective Access Time

- Hit Ratio=The percentage of times that a particular page number is found in the TLB
- An 80-percent hit ratio, for example, means that we find the desired page number in the TLB 80 percent of the time.
- TLB Hit-
  - If it takes 20 nanoseconds to search the TLB and 100 nanoseconds to access memory, then a mapped-memory access takes 120 nanoseconds
- TLB Miss-
  - If we fail to find the page number in the TLB (20 nanoseconds), then we must first access memory for the page table and frame number (100 nanoseconds) and then access the desired byte in memory (100 nanoseconds), for a total of 220 nanoseconds.

Effective Access Time

- To find the effective memory access time,

Effective access time = 0.80 x 120 + 0.20 x 220

= 140 nanoseconds.

Effective Access Time

- To find the effective memory access time,
- For a 98-percent hit ratio,

Effective access time = 0.98 x 120 + 0.02 x 220

= 122 nanoseconds.

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
- Consider α = 80%, ε = 20ns for TLB search, 100ns for memory access
- **Effective Access Time** (**EAT**)

$$\begin{aligned} EAT &= (1 + \varepsilon)\,\alpha + (2 + \varepsilon)(1 - \alpha) \\ &= 2 + \varepsilon - \alpha \end{aligned}$$

Memory Protection

- Memory protection implemented by associating protection bit with each frame
- Normally, these bits are kept in the page table.
- **Protection bit can indicate if read-only or read-write access is allowed**

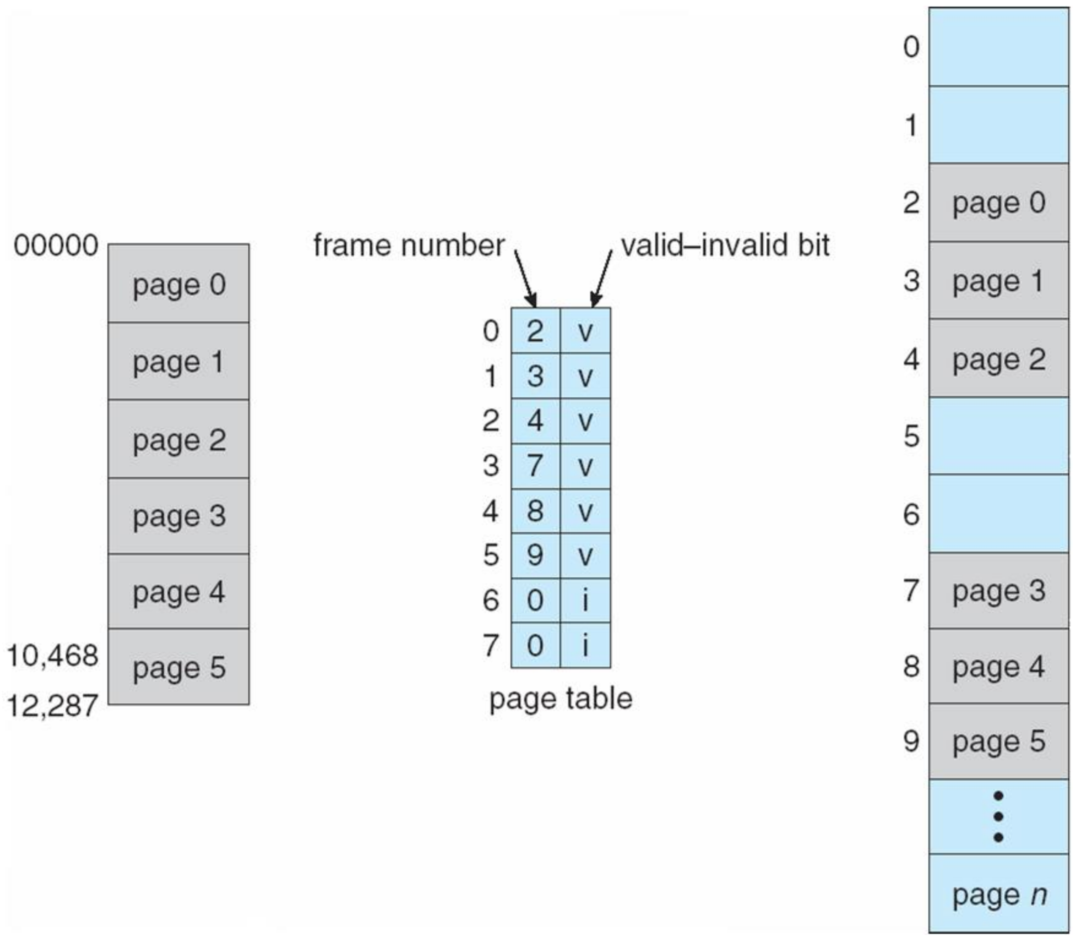
Memory Protection

- Every reference to memory goes through the page table to find the correct frame number.
- At the same time that the physical address is being computed, the protection bits can be checked **to verify that no writes are being made to a read-only page.**
- An attempt to write to a read-only page causes a hardware trap to the operating system (or memory-protection violation).

Memory Protection

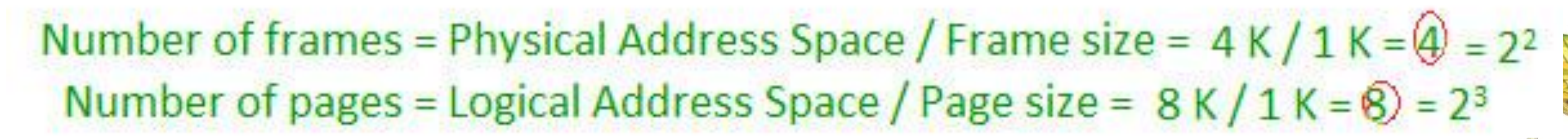
- Can also add more bits to indicate page execute-only, and so on
- **Valid-invalid** bit attached to each entry in the page table:
  - "valid" indicates that the associated page is in the process' logical address space, and is thus a legal page
  - "invalid" indicates that the page is not in the process' logical address space
  - Or use **page-table length register** (**PTLR**)
- Any violations result in a trap to the kernel

Valid (v) or Invalid (i) Bit In A Page Table



Structure of the Page Table

- Memory structures for paging can get huge using straight-forward methods
  - **Consider a 32-bit logical address as on modern computers=Logical Address Space=2^{32}**
  - **Page size of 4 KB (2^{12})**
  - **Page table would have 1 million entries =2^{32} / 2^{12}**
  - If each entry is 4 bytes -> 4 MB of physical address space / memory for page table alone
    - That amount of memory used to cost a lot
    - We would not want to allocate the page table contiguously in main memory



Number of frames = Physical Address Space / Frame size = 4 K / 1 K = $\boxed{4}$ = 2^2

Number of pages = Logical Address Space / Page size = 8 K / 1 K = $\boxed{8}$ = 2^3

Structure of the Page Table

One simple solution *to* this problem is to divide the page table into smaller pieces.

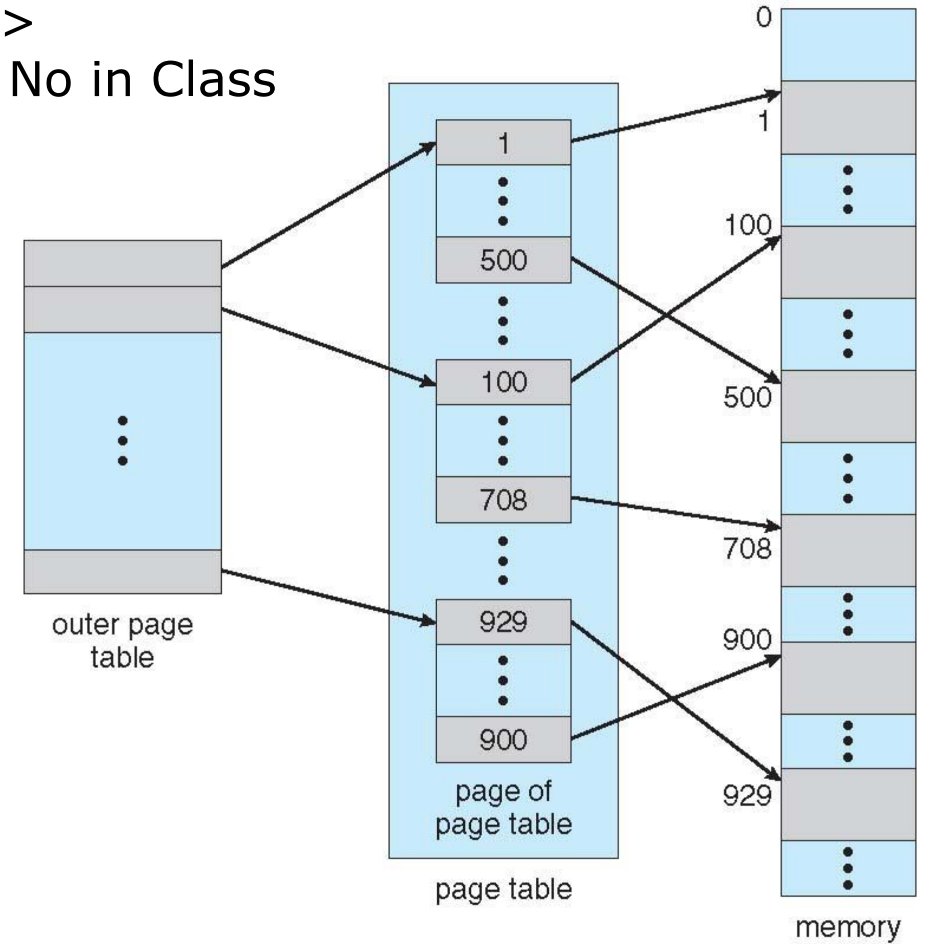
1) Hierarchical Paging
2) Hashed Page Tables
3) Inverted Page Tables

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

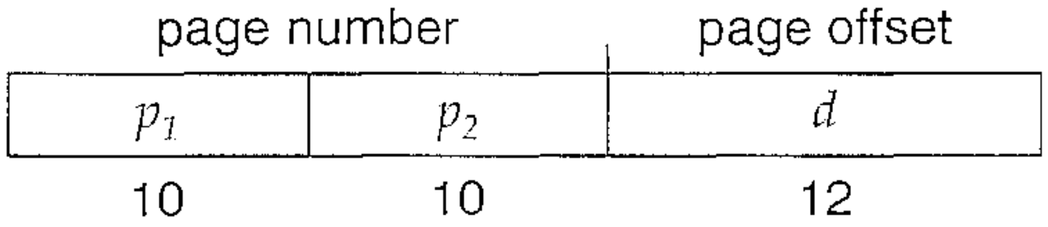
Two-Level Page-Table Scheme

Admin=>Dept=>
ClassNo=>Seat No in Class



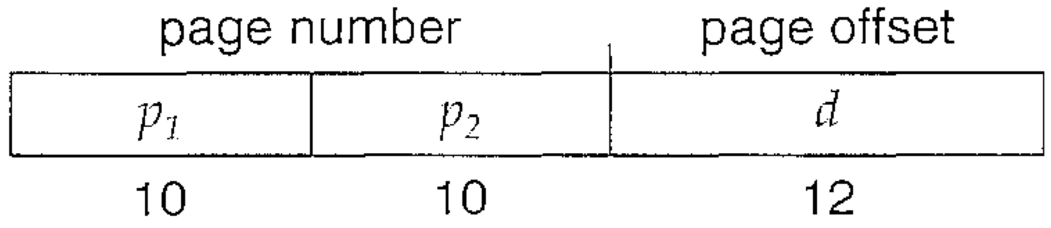
Two-Level Paging Example

- A logical address (on 32-bit machine with 1K page size) is divided into:
  - 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:
  - a 10-bit page number
  - a 10-bit page offset
- Thus, a logical address is as follows:



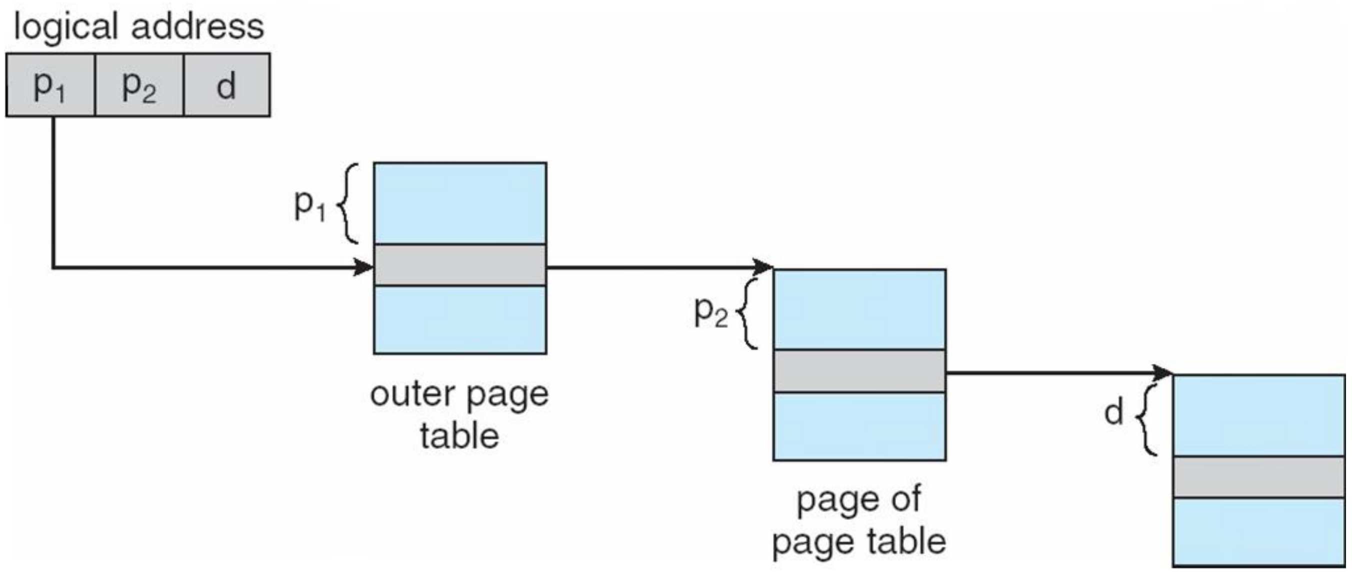
Two-Level Paging Example

- Thus, a logical address is as follows:



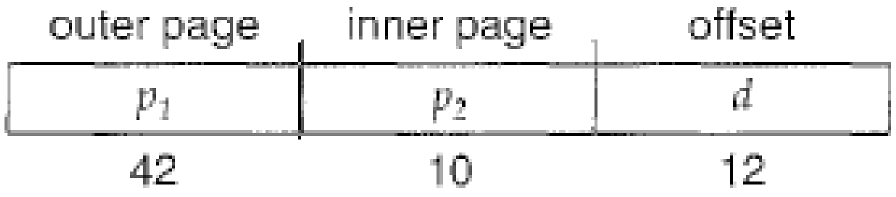
- where
- p_1 is an index into the outer page table,
- p_2 is the displacement within the page of the outer page table
- Because address translation works from the outer page table inward, this scheme is also known as **forward-mapped page table**

Address-Translation Scheme



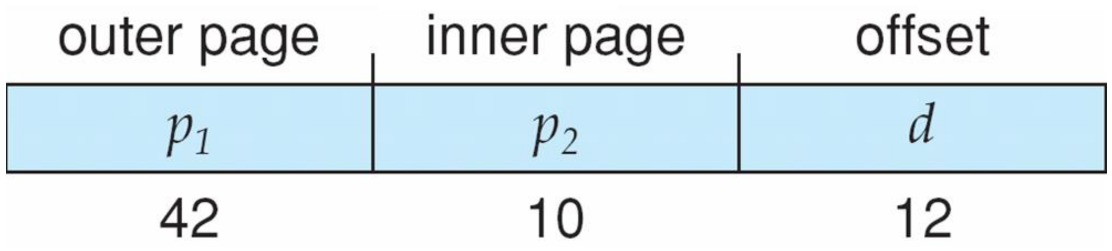
64-bit Logical Address Space

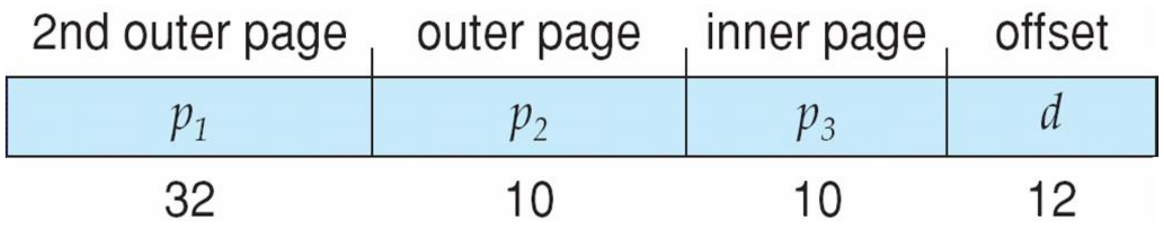
- Even two-level paging scheme not sufficient
- Logical Address Space= 2^{64}
- If page size is 4 KB (2^{12})
  - No of pages=$2^{64}/2^{12}= 2^{52}$
  - Then page table has 2^{52} entries
  - If two level scheme, Address would look like



  - Outer page table has 2^{42} entries or 2^{44} bytes

Three-level Paging Scheme





- One solution is to add a 2nd outer page table
- The outer page table is still 2^{32} bytes in size.
- The next step would be a four-level paging scheme, where the second-level outer page table itself is also paged, and so forth.

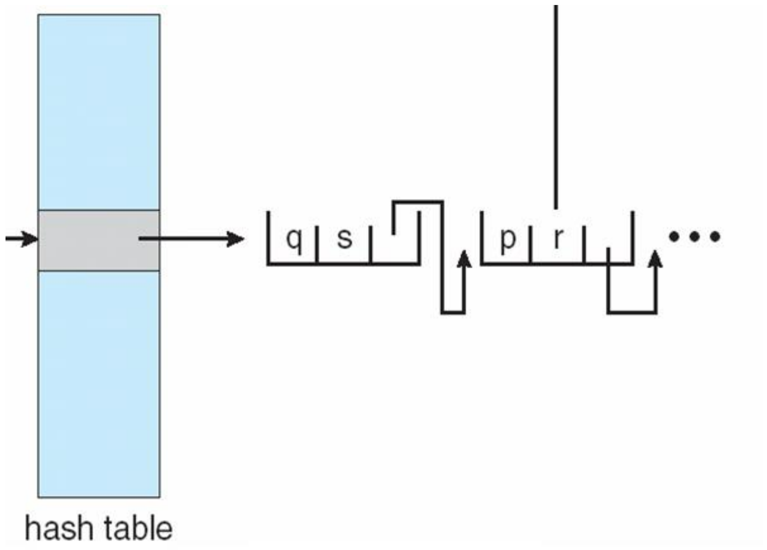
Hashed Page Tables

- Common in address spaces > 32 bits
- The virtual page number is hashed into a page table
  - Each entry in the hash table contains a linked list of elements that hash to the same location (to handle collisions).
  - Each element consists of three fields:

(1) the virtual page number,

(2) the value of the mapped page frame, and

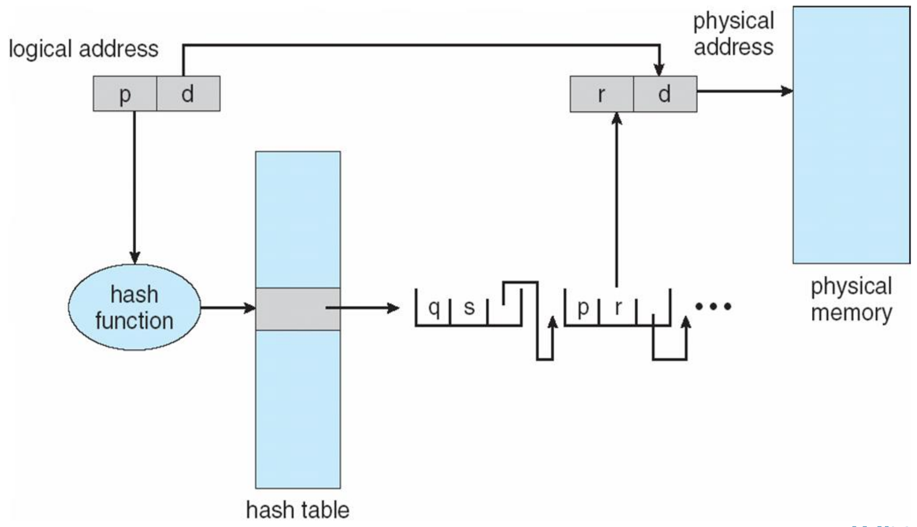
(3) a pointer to the next element in the linked list.



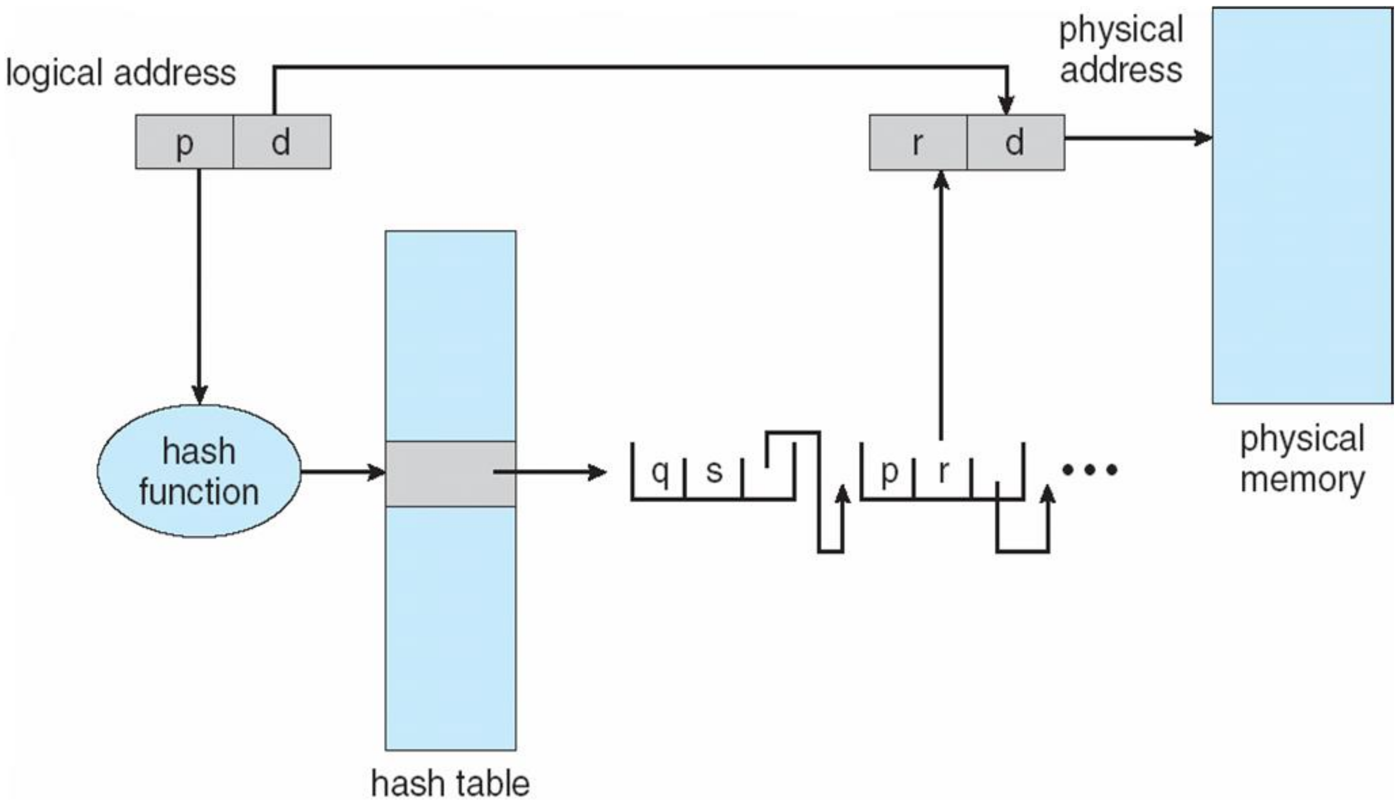
Hashed Page Tables

- The algorithm works as follows:

1) The virtual page number in the virtual address is hashed into the hash table.
2) The virtual page number is compared with field 1 in the first element in the linked list.
3) If there is a match, the corresponding page frame (field 2) is used to form the desired physical address.
4) If there is no match, subsequent entries in the linked list are searched for a matching virtual page number.



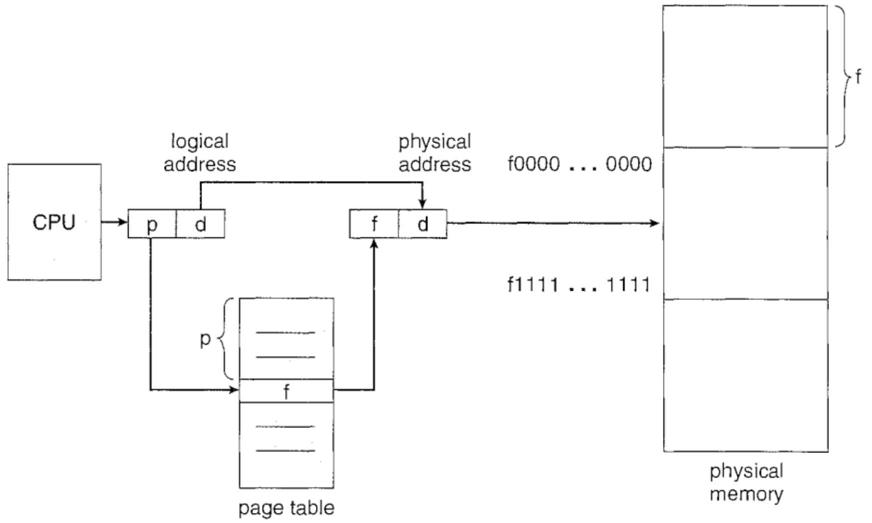
Hashed Page Table

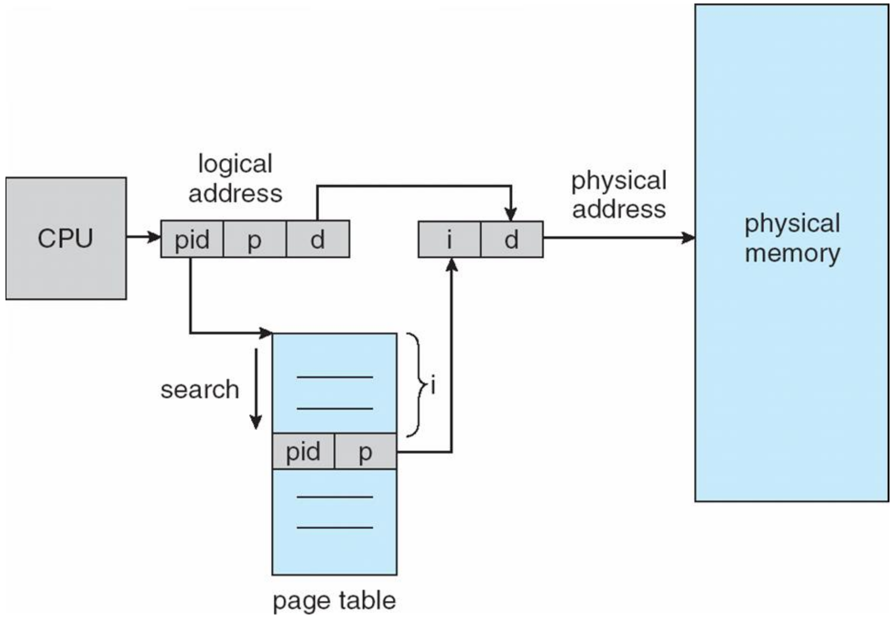


Inverted Page Table

- 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(frame) 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
- **Thus, only one page table is in the system, and it has only one entry for each page(frame) of physical memory.**

Inverted Page Table Architecture



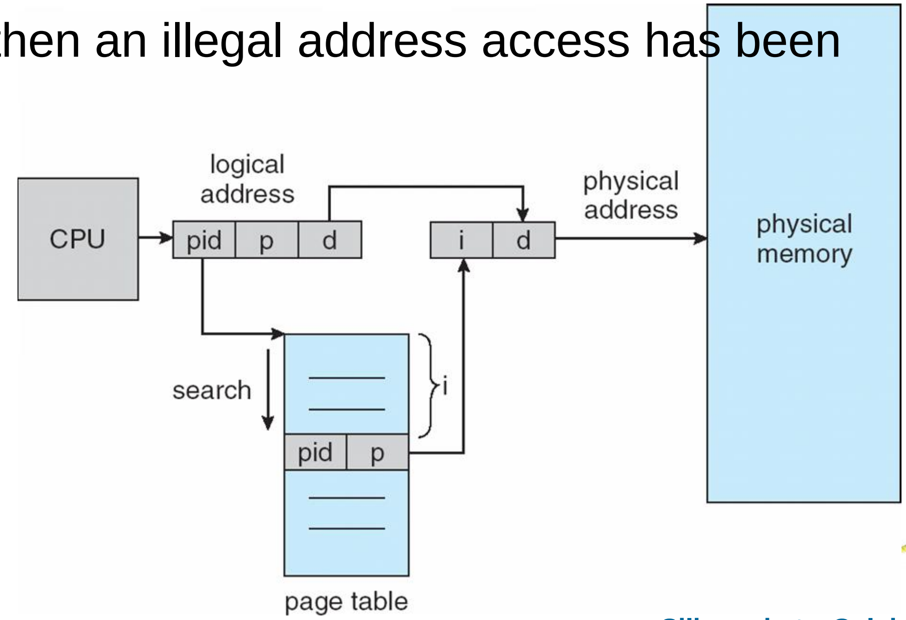


Inverted Page Table Architecture

- Each virtual address in the system consists of a triple:

<process-id, page-number, offset>

1) When a memory reference occurs, part of the virtual address, consisting of <process-id, pagenumber> is presented to the memory subsystem.
2) The inverted page table is then searched for a match.
3) If a match is found-say, at entry i-then the physical address <*i,* offset> is generated.
4) If no match is found, then an illegal address access has been attempted.



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

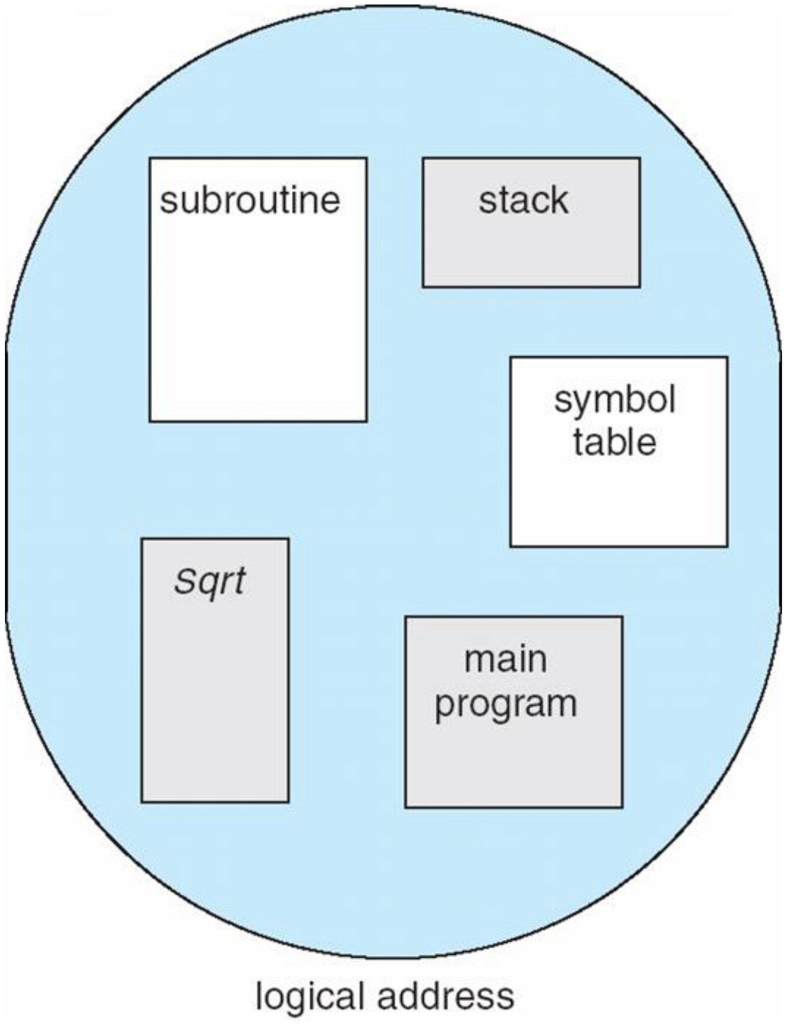
    common block

    stack

    symbol table

    arrays

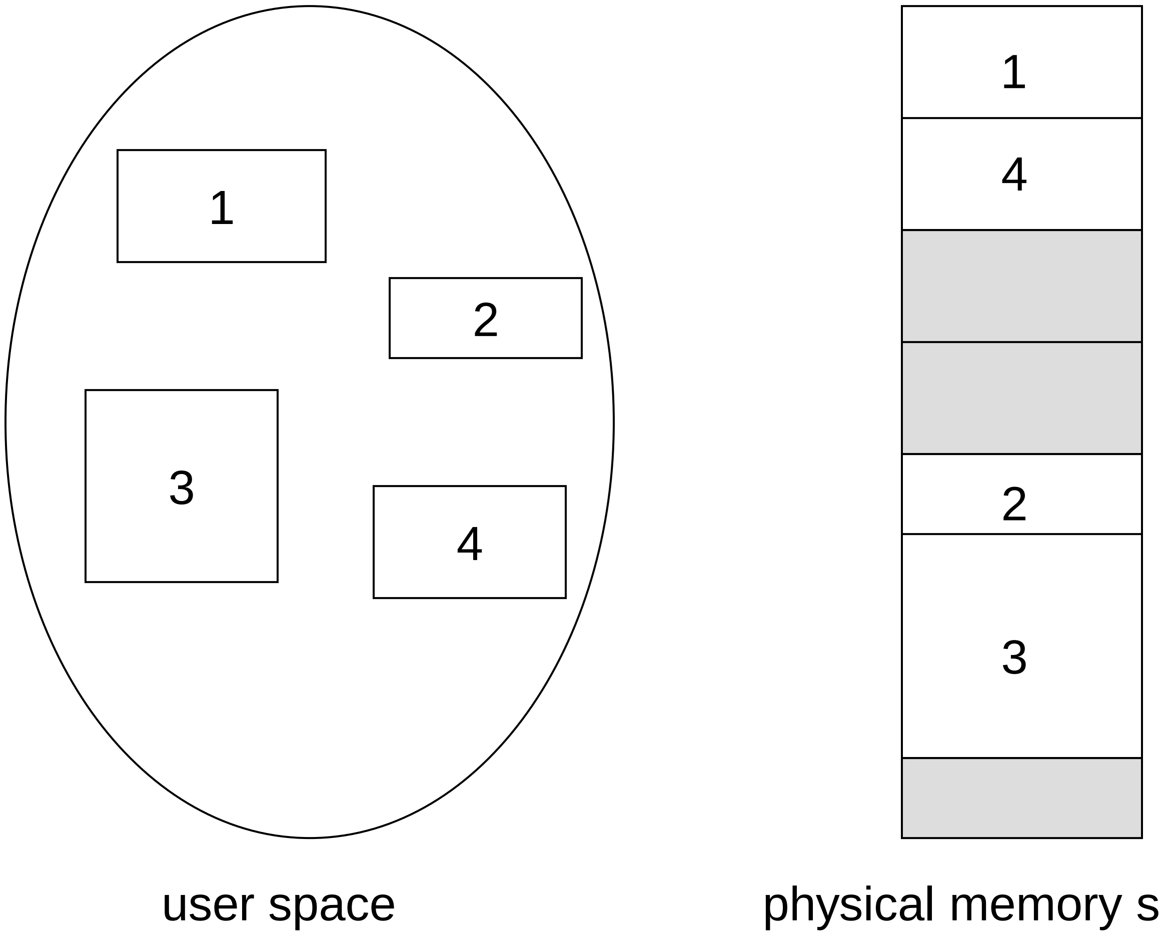
User' s View of a Program



- A logical address space is a collection of segments.
- Each segment has a name and a length.
- The addresses specify both the

1) segment name and
2) the offset within the segment.

Logical View of Segmentation



Segmentation Architecture

- Segments are numbered and are referred to by a segment number, rather than by a segment name.
- **Logical address consists of a two tuple:**

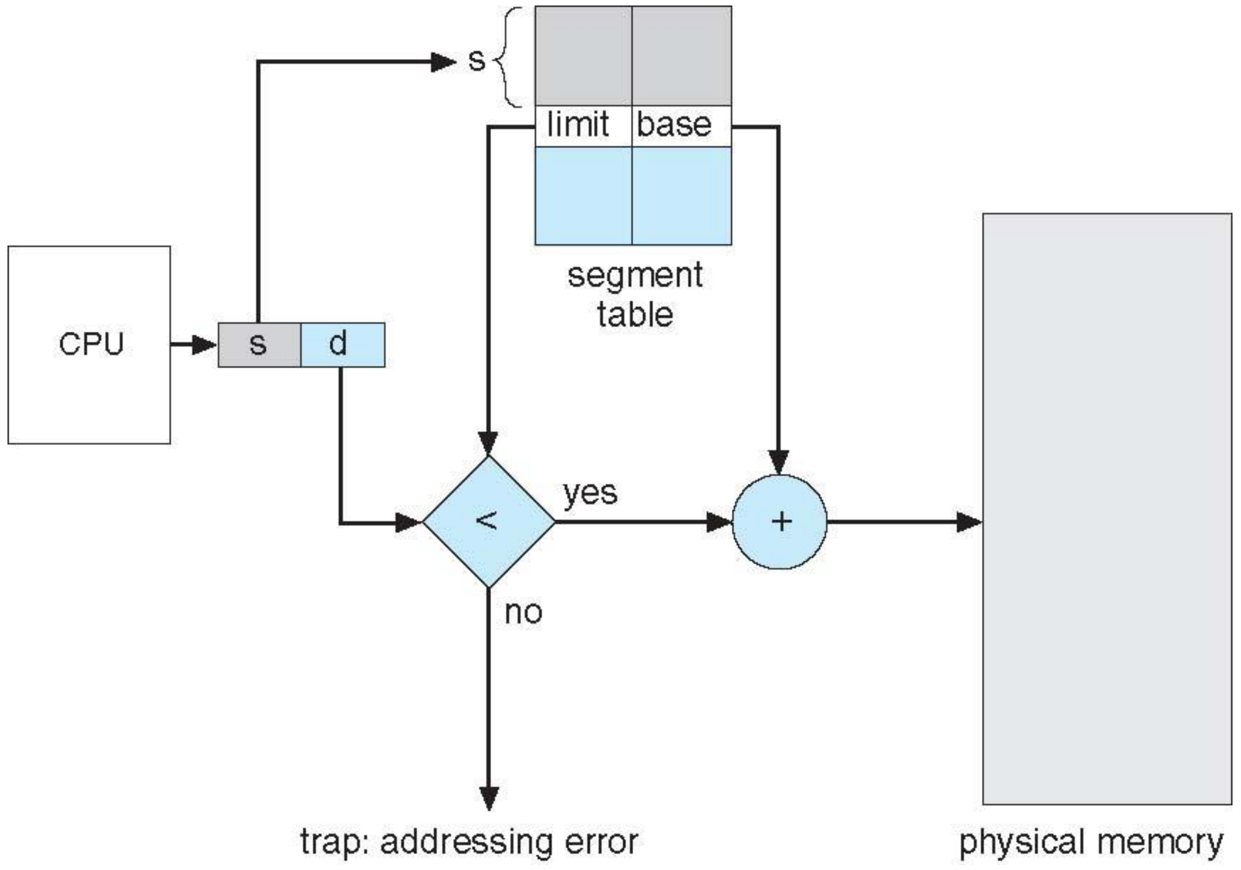
  <segment-number, offset>,

Segmentation Architecture

- Normally, the user program is compiled, and
- The compiler automatically constructs segments reflecting the input program.
- A C compiler might create separate segments for the following:

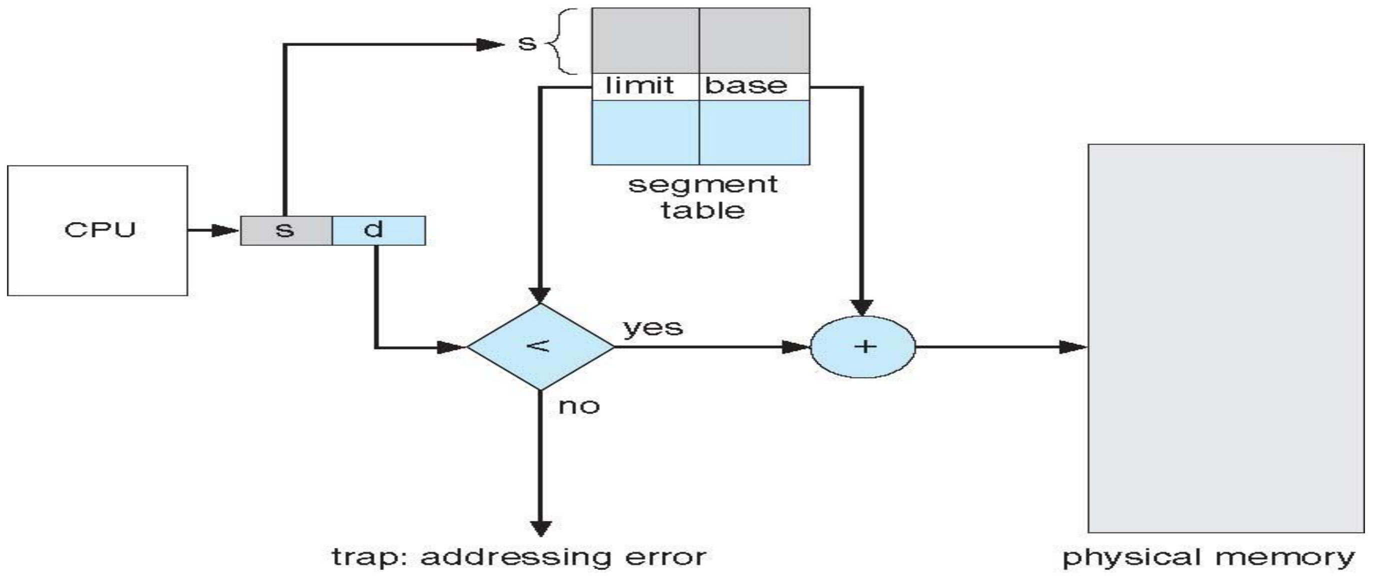
1) **The code**
2) **Global variables**
3) **The heap, from which memory is allocated**
4) **The stacks used by each thread**
5) **The standard C library**

Segmentation Hardware



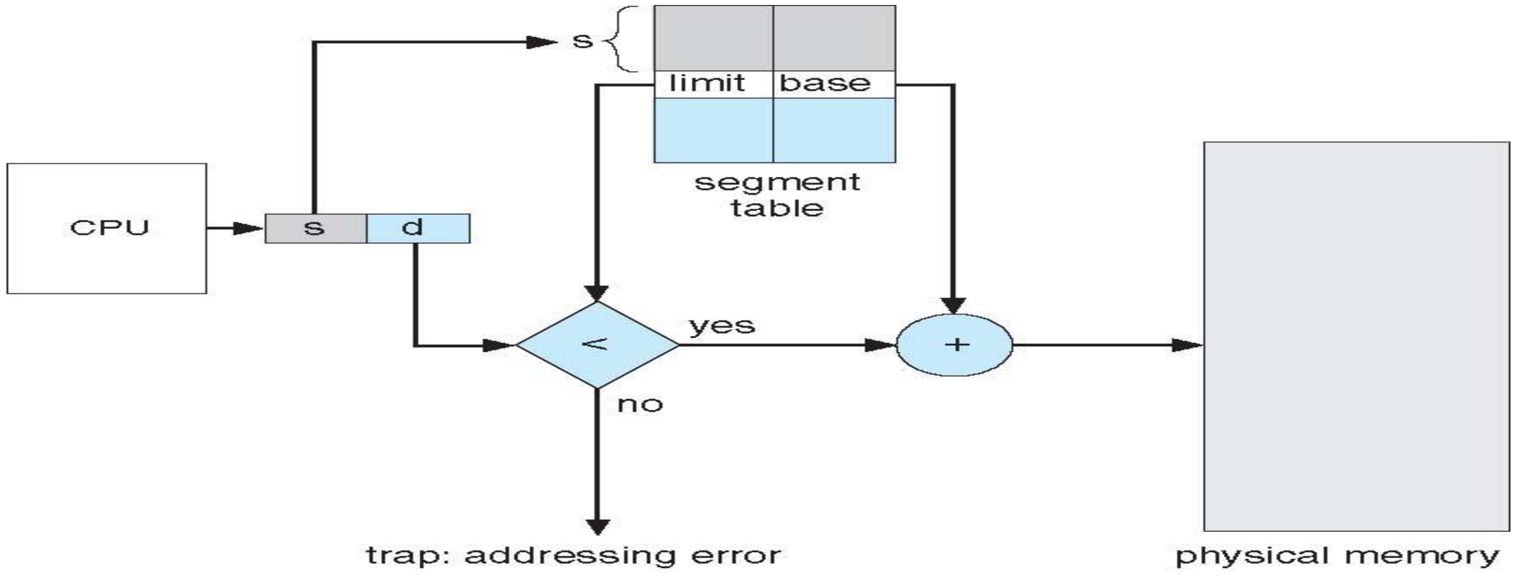
Segmentation Hardware

- **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



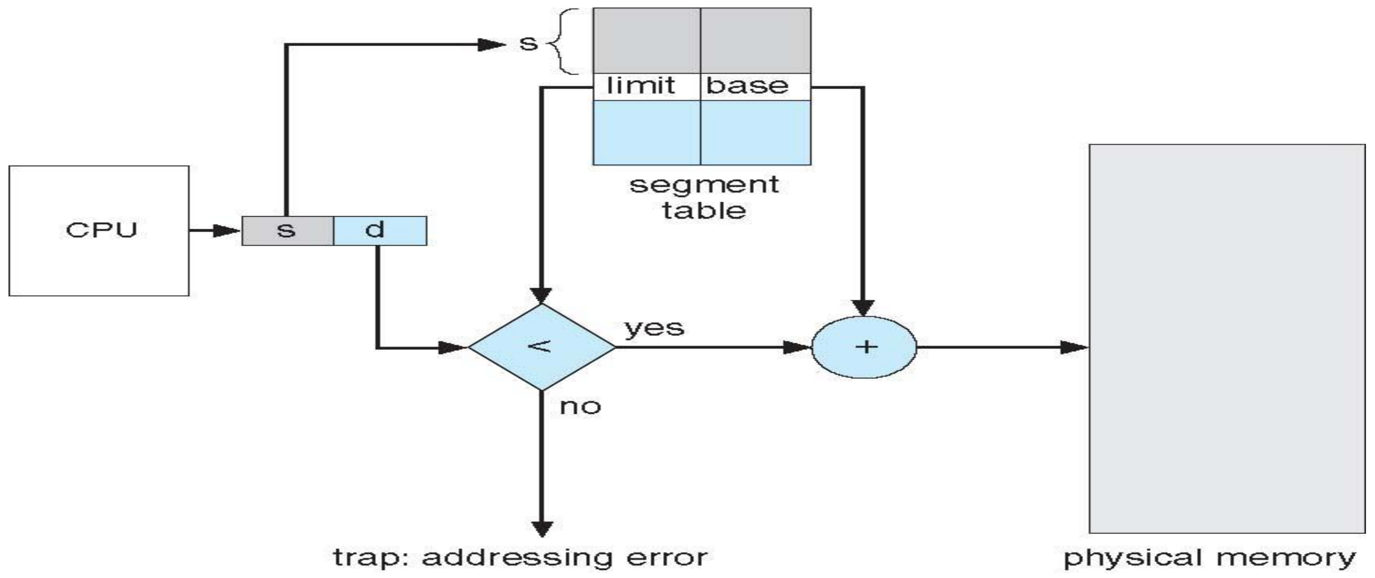
Segmentation Hardware

- The segment number is used as an index to the segment table.
- The offset *d* of the logical address must be between 0 and the segment limit.
- If it is not, we trap to the operating system (logical addressing attempt beyond end of segment).



Segmentation Hardware

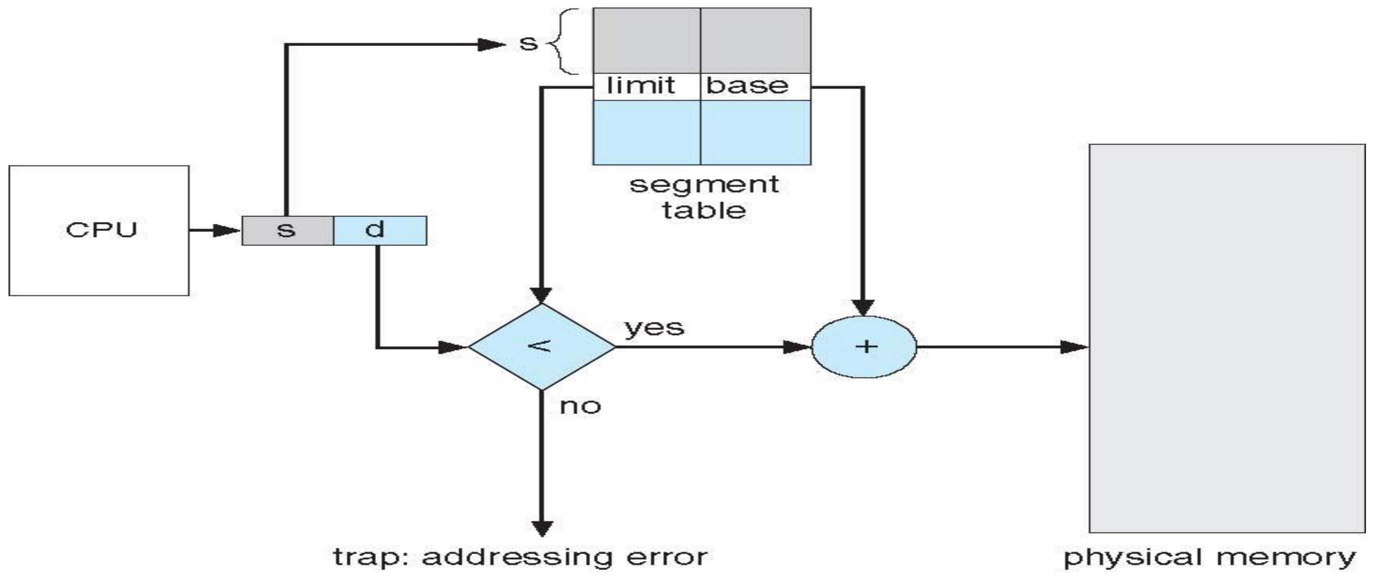
- When an offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte.
- The segment table is thus essentially an array of base-limit register pairs.



Segmentation Hardware

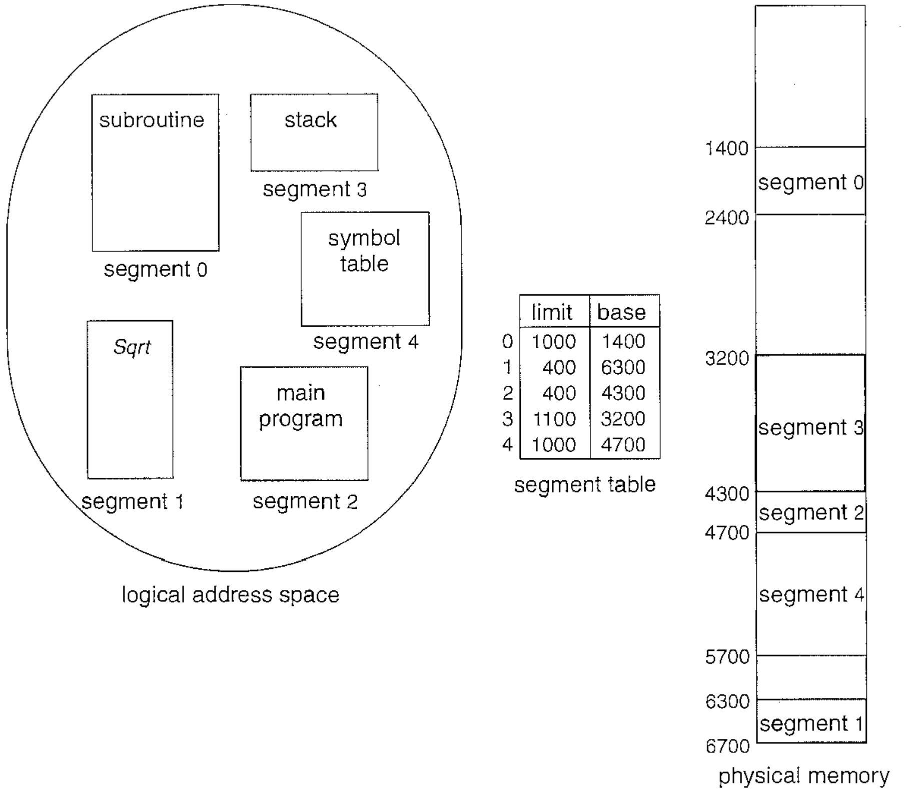
- **Segment-table base register (STBR)** points to the segment table's location in memory
- **Segment-table length register (STLR)** indicates number of segments used by a program;

  segment number ***s*** is legal if ***s*** < **STLR**



Segmentation Example

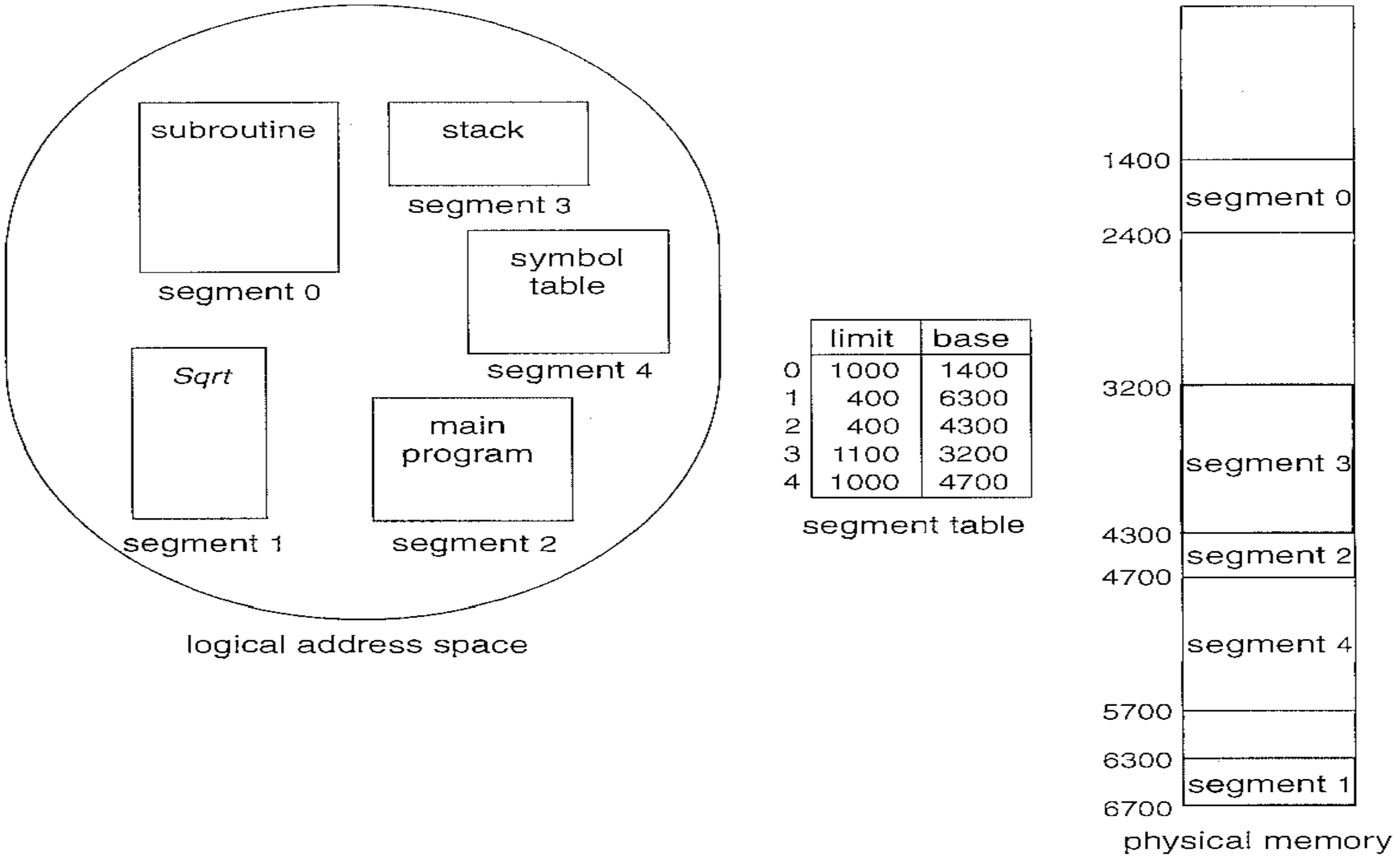
- We have five segments numbered from 0 through 4.
- The segments are stored in physical memory as shown.
- The segment table has a separate entry for each segment, giving the beginning address of the segment in physical memory or base) and the length of that segment (or limit).



	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

Segmentation Example

- Segment 2 is 400 bytes long and begins at location 4300. Thus, a reference to byte 53 of segment 2 is mapped onto location 4300 +53= 4353.

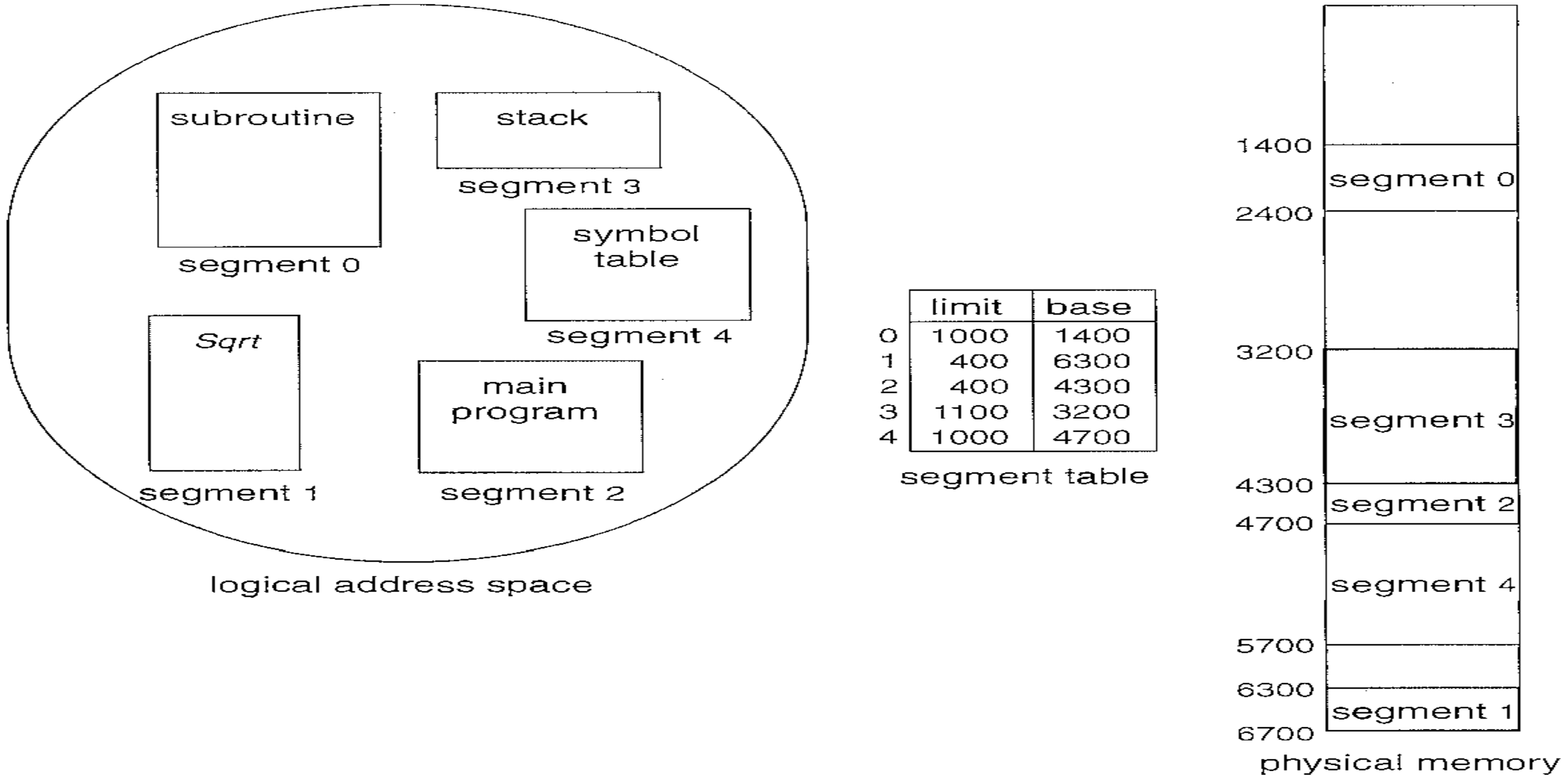


	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

segment table

Segmentation Example

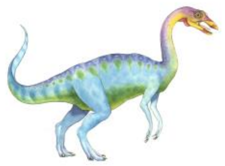
- A reference to segment 3, byte 852, is mapped to 3200 (the base of segment 3) + 852 = 4052.
- A reference to byte 1222 of segment 0 would result in a trap to the operating system, as this segment is only 1OOO bytes long.



	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

Example: The Intel 32 and 64-bit Architectures

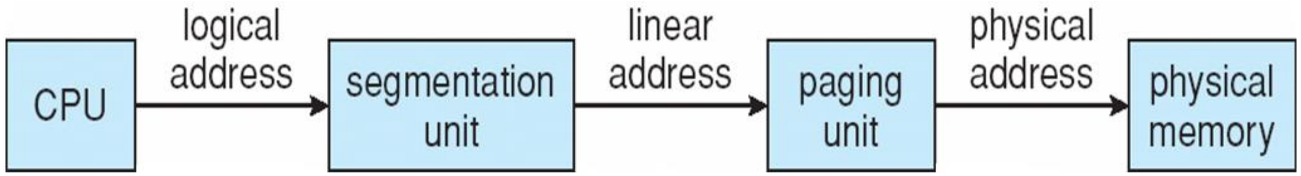
- Dominant industry chips
- Pentium CPUs are 32-bit and called IA-32 architecture
- Current Intel CPUs are 64-bit and called IA-64 architecture
- Many variations in the chips, cover the main ideas here

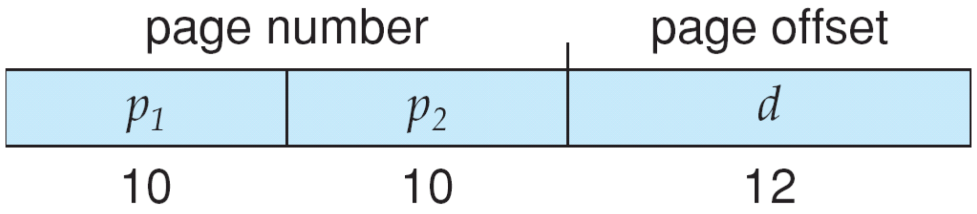


Example: The Intel IA-32 Architecture- Segmentation with Paging

- **Supports both segmentation and segmentation with paging**
  - Each segment can be 4 GB
  - Up to 16 K segments per process
  - Divided into two partitions
    - First partition of up to 8 K segments are private to process (kept in **local descriptor table** (**LDT**))
    - Second partition of up to 8K segments shared among all processes (kept in **global descriptor table** (**GDT**))

Logical to Physical Address Translation in IA-32

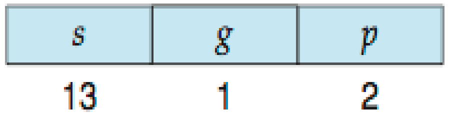




1) The CPU generates logical addresses, which are given to the segmentation unit.
2) The segmentation unit produces a linear address for each logical address.
3) The linear address is then given to the paging unit, which in turn generates the physical address in main memory.
4) Thus, the segmentation and paging units form the equivalent of the memory-management unit (MMU).

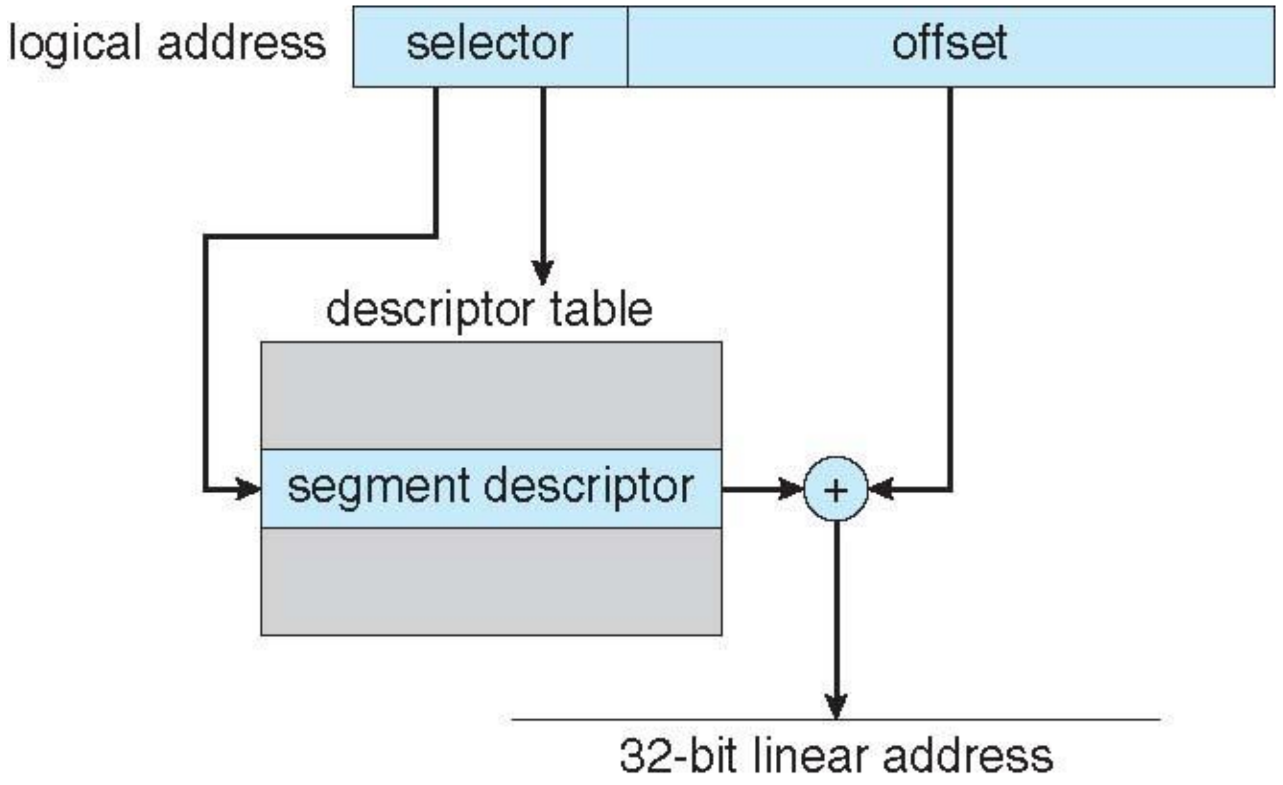
Example: The Intel IA-32 Architecture (Cont.)

- CPU generates logical address
  - Selector given to segmentation unit
    - Which produces linear addresses

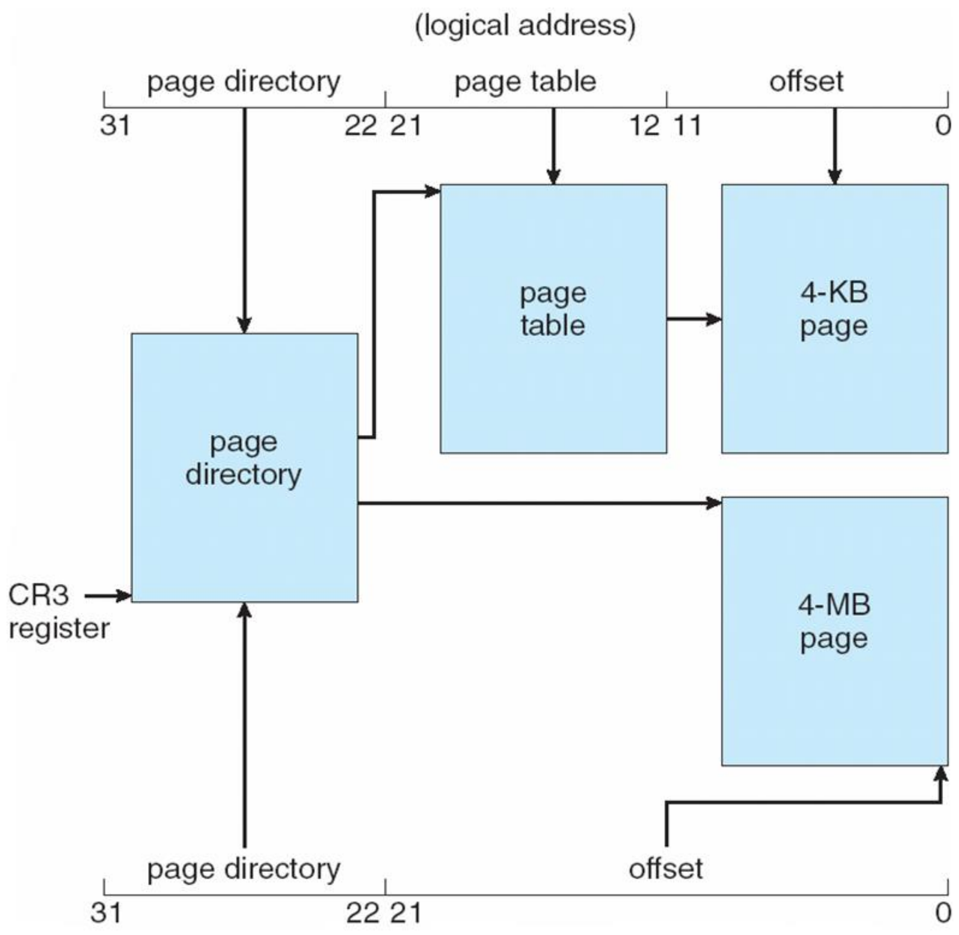


  - Linear address given to paging unit
    - Which generates physical address in main memory
    - Paging units form equivalent of MMU
    - Pages sizes can be 4 KB or 4 MB

Intel IA-32 Segmentation

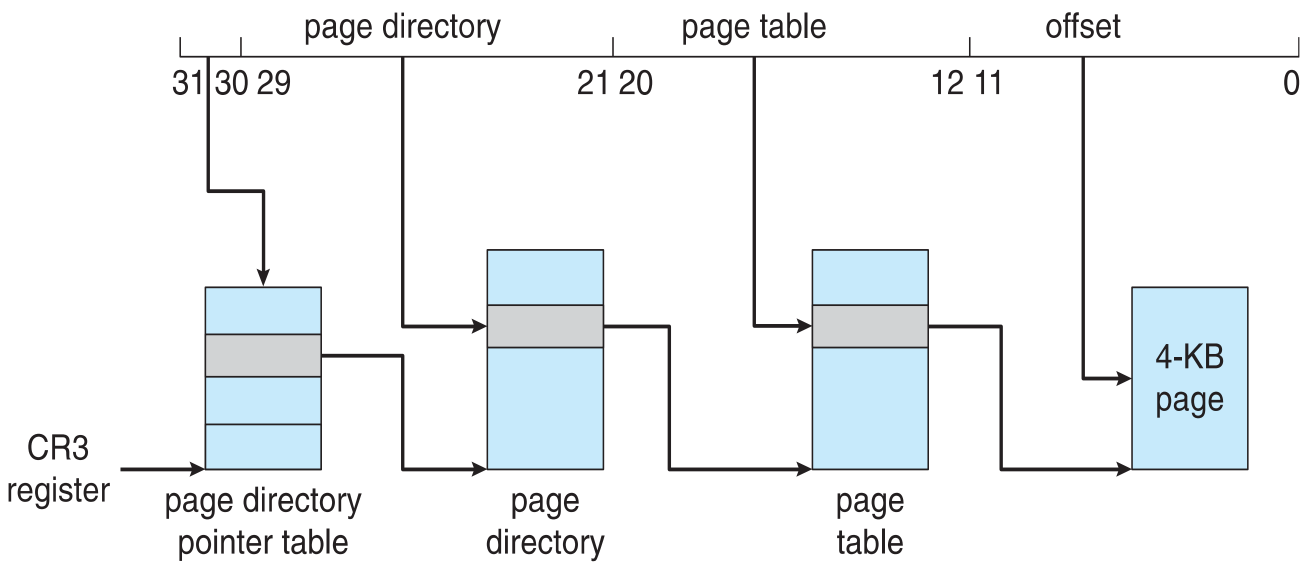


Intel IA-32 Paging Architecture



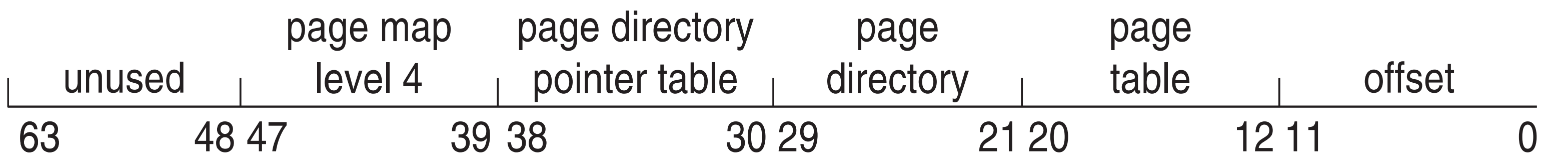
Intel IA-32 Page Address Extensions

- 32-bit address limits led Intel to create **page address extension** (**PAE**), allowing 32-bit apps access to more than 4GB of memory space
  - Paging went to a 3-level scheme
  - Top two bits refer to a **page directory pointer table**
  - Page-directory and page-table entries moved to 64-bits in size
  - Net effect is increasing address space to 36 bits – 64GB of physical memory



Intel x86-64

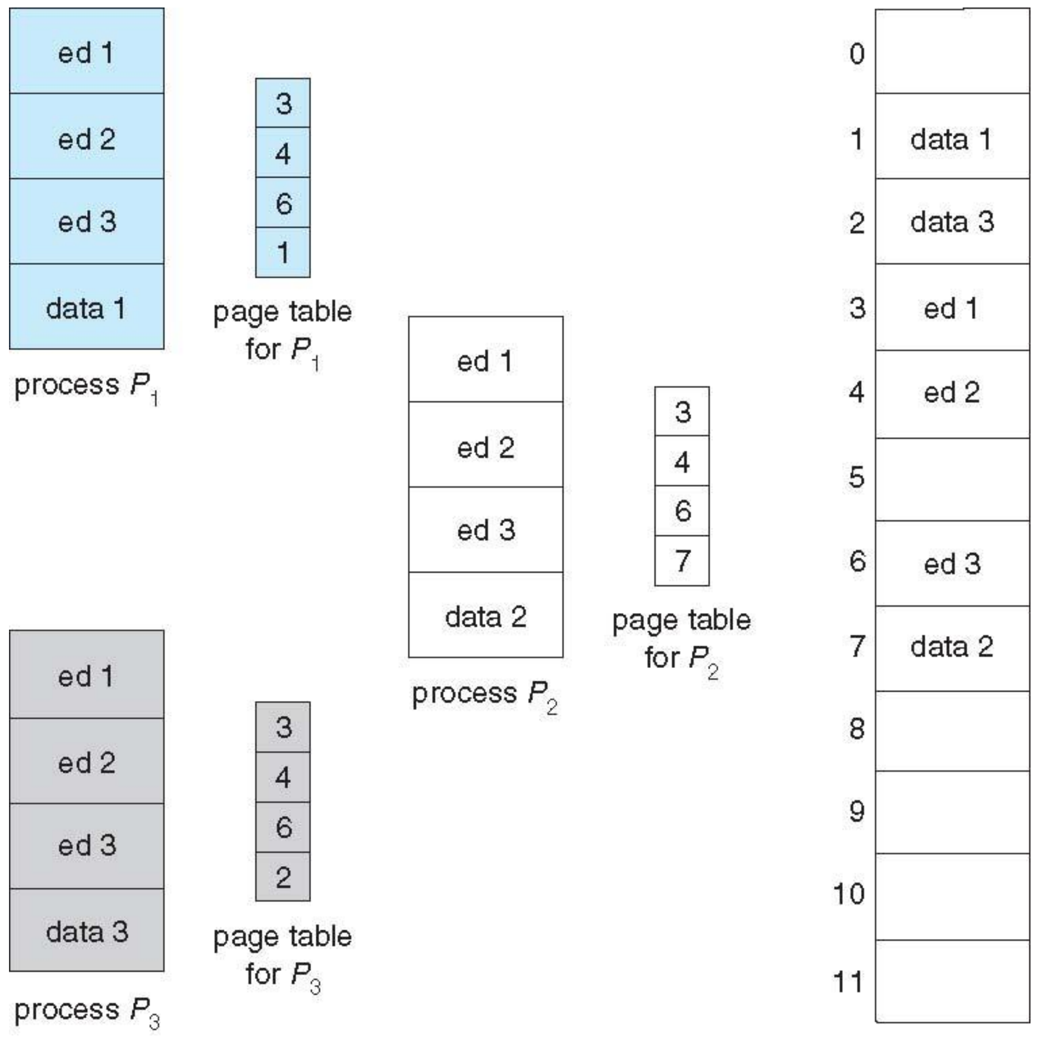
- Current generation Intel x86 architecture
- 64 bits is ginormous (> 16 exabytes)
- In practice only implement 48 bit addressing
  - Page sizes of 4 KB, 2 MB, 1 GB
  - Four levels of paging hierarchy
- Can also use PAE so virtual addresses are 48 bits and physical addresses are 52 bits



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


ed 1
ed 2
ed 3
data 1
process P_1
3
4
6
1
page table for P_1
ed 1
ed 2
ed 3
data 2
process P_2
3
4
6
7
page table for P_2
ed 1
ed 2
ed 3
data 3
process P_3
3
4
6
2
page table for P_3
0
1 data 1
2 data 3
3 ed 1
4 ed 2
5
6 ed 3
7 data 2
8
9
10
11