# Chap 6: Memory Management Strategies

BY PRATYUSA MUKHERJEE, ASSISTANT PROFESSOR (I)

KIIT DEEMED TO BE UNIVERSITY

# Background

- Memory is central to operation of a modern computer system and it consists of a large array of bytes each with its own address.
- CPU fetches instructions from memory as per the value of PC. These instructions may cause additional loading from and storing to specific memory addresses.
- The typical flow of events are: fetch inst. from memory, decode it, fetch operands from memory, execute inst. on operands, store result back in memory.
- The memory unit sees only a stream of addresses, it doesn't know how they are generated or what they are. Thus we are also only interested in the sequence of memory addresses generated by a running program / process.

# Basic Hardware

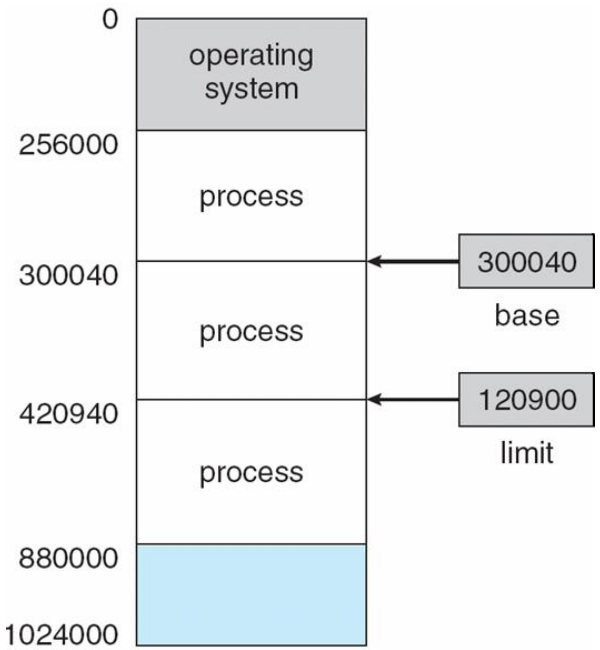
- CPU can directly access only the main memory (MM) and registers.
- Thus any inst. in execution and any data used by this inst. must be in one of these direct access storage devices. If they are not in the memory, they must be moved there before the CPU can operate on them.
- Registers are usually accessed in one CPU clock cycle. But completing a memory access may take many CPU cycles.
- Thus the processor needs to stall as it doesn't have the data required to complete the instruction which it is executing. This becomes intolerable due to the frequency of memory access. So we need a fast memory between CPU and MM - cache mamory.
- To manage a cache built into the CPU, the hardware automatically speeds up memory access without any OS control.

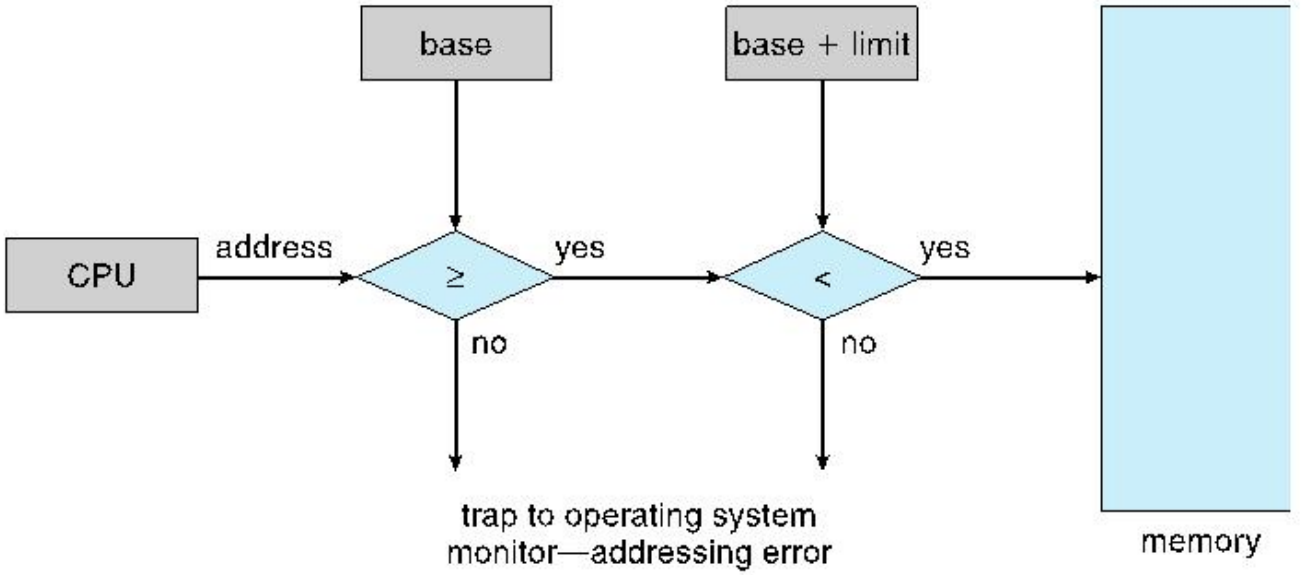
# Point of Concern

- Apart from the relative speed of accessing the physical memory, we must also ensure correct operation. Thus we must protect the OS from access by user processes. On multi-user OS, we must also protect user processes from one another.
- This protection must be provided y the hardware as OS doesn't intervene between CPU and its memory access
- A possible solution is: **make sure each process has a separate memory space**. This will protect the processes from each other and allows multiple processes to be loaded in memory for concurrent execution.
- Thus, we have to determine the range of legal addresses that the process may access and ensure that the process truely accesses these legal addresses only.
- This is done using two registers: **base register and limit register** which define the logical address space.

# 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.**



# Important Points

- Base and limit registers can be loaded only by the operating system only using special privileged instructions that can be executed only in kernel mode.
- This thus prevents the user programs from changing the value of registers.
- An OS executing in kernel mode is given unrestricted access to both OS memory and user's memory. This allows OS to load user's progs into user's memory, dump theseout in case of errors, access and modify parameters of system calls, perform I/O to and from the user memory.

# Address Binding

**Input Queue:**

- Program resides on a disk as a binary executable file. In order to be executed, it is brought into memory and placed within a process.
- Depending on the memory management in use, processes may be moved between between disk and memory during its execution.
- The processes on the disk that are waiting to brought into memory for execution form the **input queue.**
- Thus a process is selected from the input queue and is brought to the memory.As the process executes, it accesses operands and operations from the memory. Eventually, the process terminates and the memory space is declared available/ free.

# Address Binding contd.

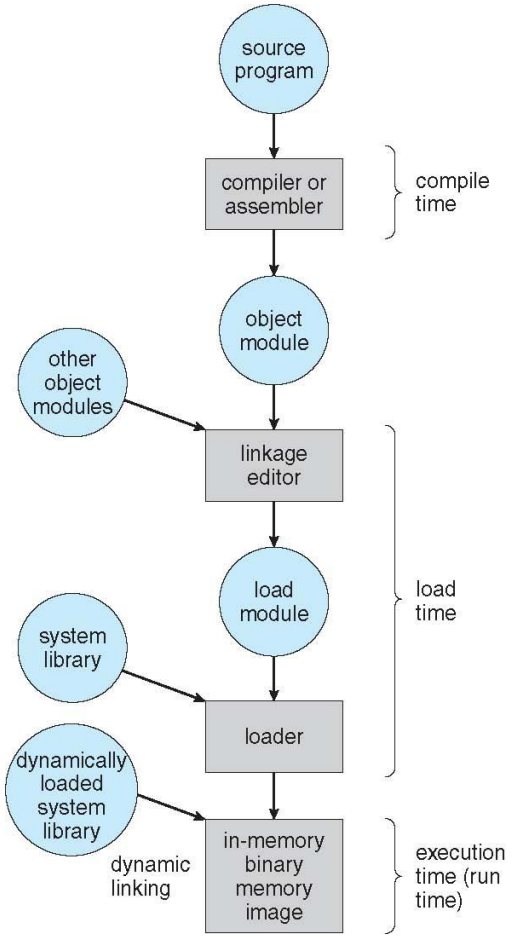
- A user program goes through various steps before being executed and during these steps, the addresses are represented in different ways.
- Address in **source program** are generally **symbolic** ( Eg: Variable count)
- A **compiler** binds these s**ymbolic addresses to relocatable address** (Eg: 14 bytes from the beginning of the module)
- The **linkage editor or loader** in turn binds the **relocatable address to an absolute address** ( Eg: 74014)
- Thus each binding is a mapping from 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. If its known the user process resides at a location R, the generated compiler code will also start at R and extend up from there. Thus we must recompile code if starting location changes
- **Load time**: Must generate **relocatable code** if memory location is not known at compile time. Thus final binding is delayed until load time. If starting address change, we only reload the user code to incorporate changes.
- **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 such 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 i.e. the one loaded into the **memory address register** of the memory

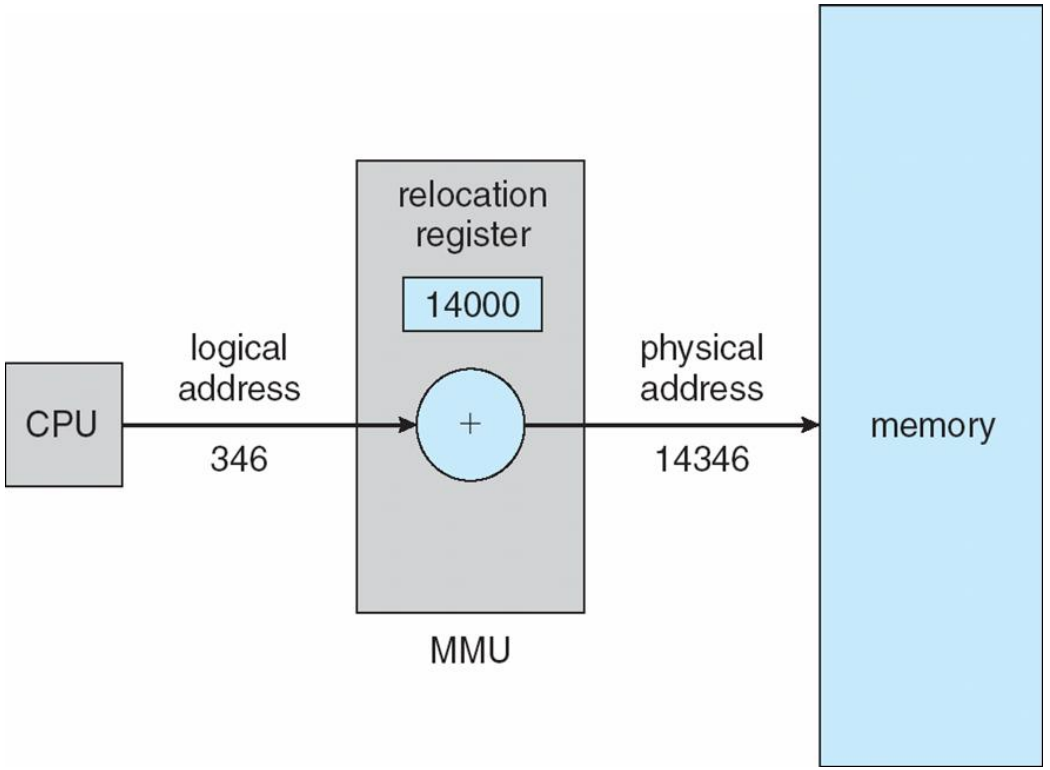
**Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme**

**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)

- It is a hardware device that at run time maps virtual to physical address.
- Consider a simple MMU scheme the is a generalization of the base register scheme.
- The base register is now called the relocation register. The value of the relocation register is added to every address generated by a user process at the time the address is sent to memory.
- Thus the user never sees the real physical address, it only deals with logical addresses.
- The final location of a referenced memory is not determined until the reference is made.



# Dynamic Loading

- So far we saw that it is necessary for entire program and all data must be in physical memory for the process to execute.
- The size of a process is thus limited to the size of physical memory.
- To enable better memory space utilization, we can use dynamic loading.
- With Dynamic loading, a routine is not loaded until it is called and all routines are kept on disk in a relocatable load format.
- Useful when large amounts of code are needed to handle infrequently occurring cases
- This does not need any special support from the OS. Implemented through program design OS can help by providing libraries to implement dynamic loading

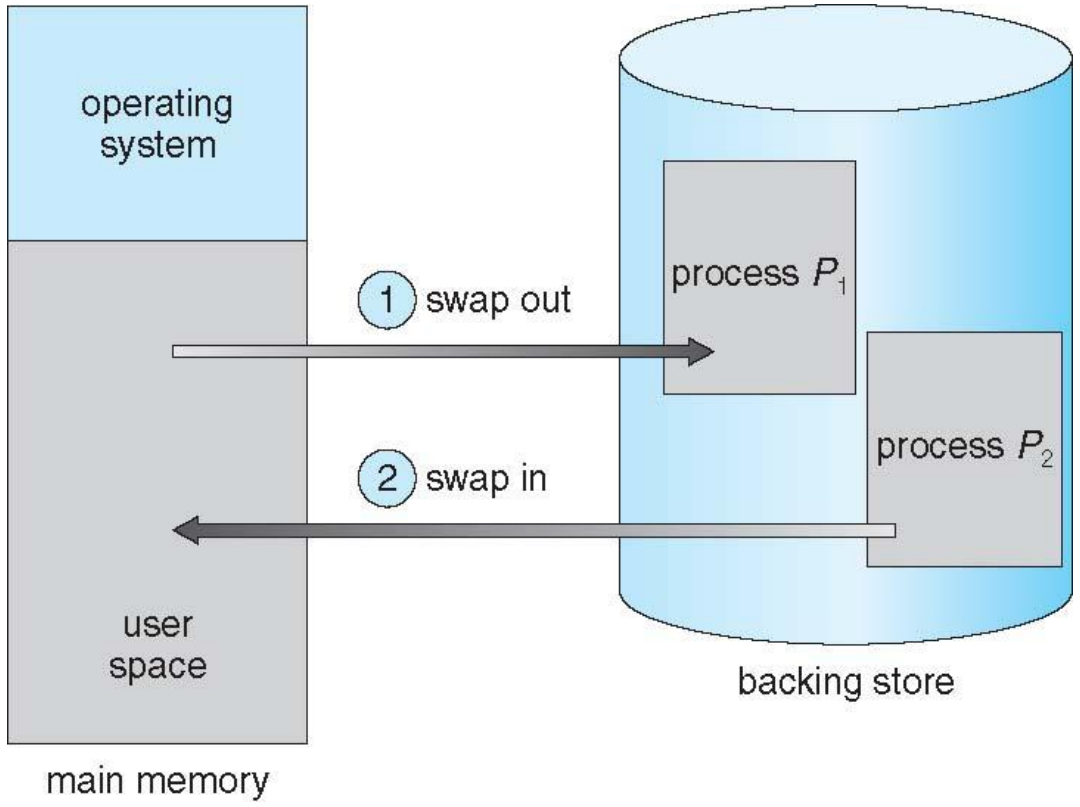
# Dynamic Linking

- **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

**Dynamic Linking vs Dynamic Loading**

# Swapping

- A process must be in memory to be executed. However, it can be **temporarily swapped out** of the main memory to a backing store and the **again brought back** to the memory for continued execution
- Thus swapping makes it possible for the total physical address space of all processes to exceed the real physical memory of the system, thus **increasing the degree of multiprogramming in a system.**



# Standard Swapping

- It involves moving processes between main memory and backing store.
- **Backing store** is a flat disk, large enough to accommodate copies of all memory images for all users and it must provide direct access to these memory images.
- System maintains a **ready queue** consisting of all processes whose memory images are on the backing store or in main memory and are ready to run.
- Whenever the CPU scheduler decides to execute a process, it calls the **dispatcher**. The dispatcher checks whether the next process in the queue is in the momory or not. If it is not, and if there is no free memory region, the dispatcher swaps out a process currently in memmory and swaps in this desired process. It then reloads registers and transfers control to the selected process

# What about context switching while swapping??

- In standard swapping system, the **context switch time is fairly high**. **(Eg discussed)**
- Major part of the swap time is **transfer time.** Total transfer time is directly proportional to amount of memory swapped. **(Eg discussed)**
- It is thus useful to know how much memory a process is actually using and not simply how much it might use. Then we would need to swap only what is actually used, thus reducing the swap time.
- Hence a user must keep the system informed of any changes in memory requirement using 2 systems calls **request memory () and release memory ().**
- **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

# Double Buffering

- If we want to swap a process, we must ensure it is **sitting completely idle**.
- But a process may be **waiting for a pending I/O** ( thus it seems to be idle) when we want to swap that process to free up memory.
- But if the I/O is **asynchronously accessing** the user memory for I/O buffers, then the process cannot be swapped. Otherwise, it will attempt to use memory that might belong to some other process (Eg Explained)
- 2 possible solutions to this problem: **never swap a process with pending I/O or execute I/O operations only in OS buffers**.
- This transfer between OS buffers and process memory then occurs only when the process is swapped in. Thus known as **double buffering**.
- It also leads to **substancial overhead** as we need to copy the data again from kernel memory to user memory before the user process can access it

# Disadvantages of Standard Swapping

- It needs **too much swapping time** and provides **too less execution time** to be a reasonable memory management solution.
- Thus modified versions of swapping are used.
- In one common version, swapping is normally disabled but will start if the amount of free memory falls below a threshold amount. Swapping is again halted when the amount of free memory increases.
- Another version involves swapping portions of processes instead of the entire process to reduce the swap time

# Swapping on Mobile Systems

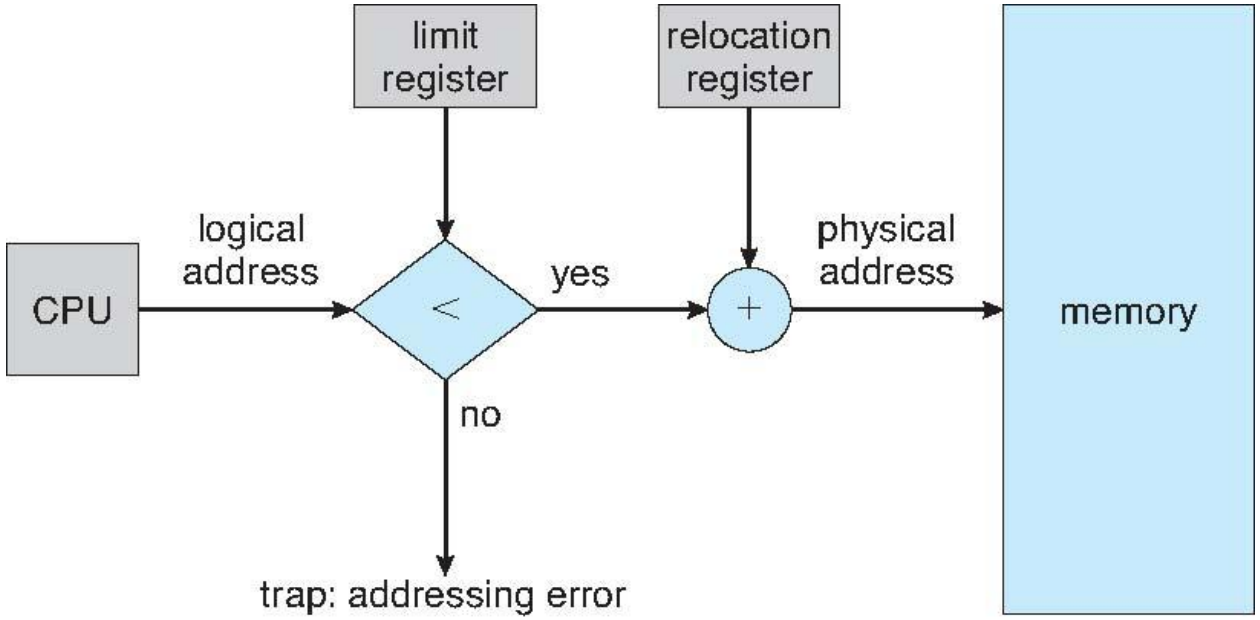
- Mobile systems do not support swapping in any form.
- They use flash memory rather than more spacious hard disks as their persistent storage.
- The resulting space constraint is why mobile OS designers avoid swapping.
- Other reasons include limited number of writes the flash memory can tolerate before it becomes unreliable and the poor throughput between main memory and flash memory.
- When memory falls below a certain threshold, Apple's ioS asks apps to voluntarily relinquish allocated memory. Read only data are removed from the system and later reloaded from flash memory if required. Data that has been modified are never removed. Any app that fails to free up sufficient memory is terminated by the OS. **(What about Android??)**
- Thus mobile developers carefully allocate and release memory to ensure that their apps do not use too much memory or suffer from memory leaks.

# Continuous Memory Allocation (CMA)

- Main Memory must accommodate both the OS and user processes. Thus we must allocate the main memory in the most efficient way possible.
- **The memory is always divided into two partitions: one for the OS and other for the user processes.**
- We can place the OS either in the low memory or high memory. The major factor affecting this decision is location of the interrupt vector. as the interrupt vector is often in the low memory, we also place the OS in low memory **(what do you mean by high memory low memory?)**
- We want several user processes to reside in the memory simultaneously. the question is how to judiciously allocate available memory to them.
- **In CMA, each process is contained in a single section of memory that is contiguous to the sections containg the next process.**

# Memory Protection in CMA

- **In order to prevent a process from accessing memory it does not own, we will use the concept of relocation register plus limit register.**
- **Relocation register contains the value of smallest physical address. Limit register contains the range of logical address.**
- When CPU Scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values as a part of the context switch.
- Every address generated by the CPU is checked against these registers, we can protect both the OS and other user programs and data from being modified by this running process.



# Transient OS code

- Relocation register scheme provides an effective way to allow the OS size to change dynamically.
- The OS contains code and buffer space for device drivers.
- **If a device driver or other OS services are not commonly used, we dont keep this code or data in memory so that we can use this space for other purposes.**
- Such code is called tansient OS code because it comes and goes as needed.
- Thus using this code changes the size of the OS during program execution.

# Memory Allocation in CMA

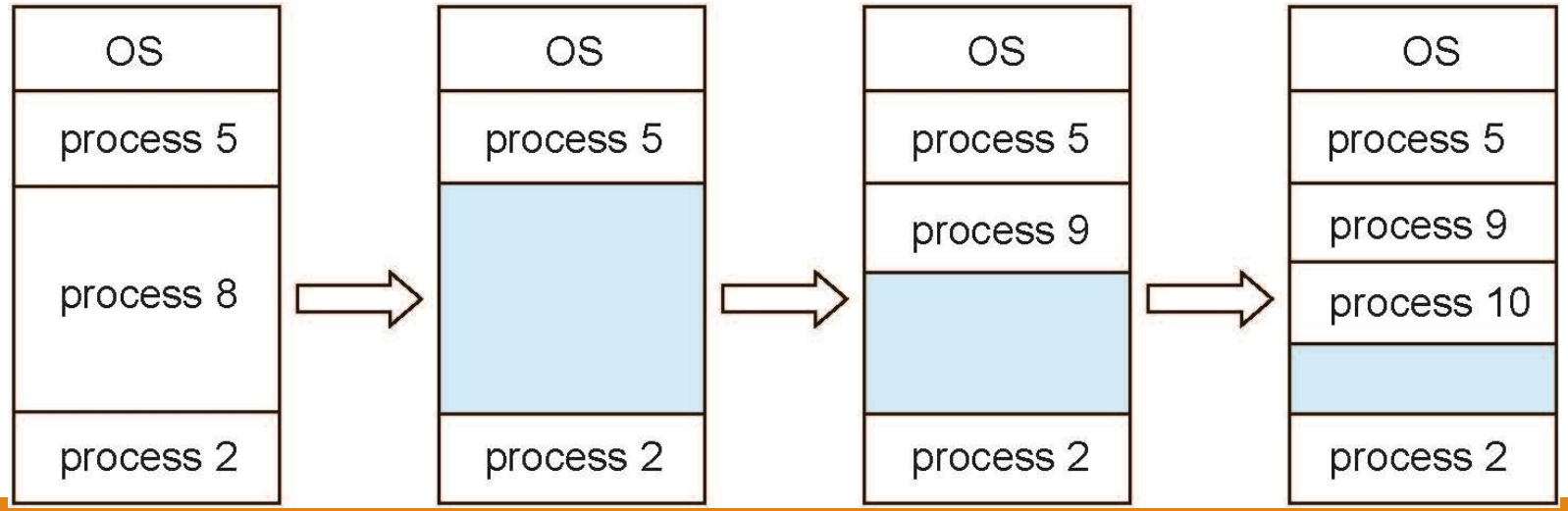
## Fixed Sized Partitioning (MFT):

- Simplest method to allocate memory is to divide memory into **several fixed sized partitions**.
- Each partition containes exactly one process. **Thus degree of multi-programming is bound by number of partitions.**
- **In this multiple partition method, when a partition is free, a process is selected from input queue and is loaded into the free partition. When the process terminates, the partition becomes available for other processes.**

**Disadvantages??**

# Variable Sized Partitioning (MVT):

- The OS keeps a table indicating which parts of the memory are available **(hole)** and which are occupied.
- Initially, entire memory is available and is considered one large block or hole.
- It enhances the efficiency because it is sized to a given process's needs
- Thus 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)
- Hence it gives a dynamic approach to memory allocation.



# 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 decreasing time (speed) and storage utilization.
- First-fit is generally faster.

# External Fragmentation

- **This exists when there is enough total memory space to satisfy a request but the available spaces are not contiguous. Thus it cant be used.**
- Hence we can say that the storage is fragmented into a large number of small holes.
- Both first and best fit suffer from this
- Worst case scenario will be to have a block of free memory between each processes. If all these small pieces were in one big free block instead, we could have run few more processes.
- First fit statistical analysis reveals that given N blocks allocated, 0.5 N blocks will be lost to fragmentation i.e. 1/3 may be unusable. This property is called **50-percent rule**

# Internal Fragmentation

- Imagine a multiple partition allocation scheme with a hole of 18464 bytes. The next process requests 18462 bytes. If we allocate exactly the request block, we are left with a hole of 2 bytes.
- The overhead to keep a track of this hole will be substancially larger than the hole itself.
- Thus it is better to break physical memory into fixed sized blocks and allocate memory in units based on block size.
- **In such case memory allocated to a process may be slightly larger than the request one.**
- **This difference between the two numbers is called internal fragmentation as it is the unused memory internal to a partition.**

# Solution to External fragmentation:

## Compaction:

- Shuffle the memory space so as to place all free memory together in one large block
- This is not possible if rerlocation is static and is done at assembly or load time.
- It is only possible 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 value to reflect the new base address.
- The simplest compaction algorithm involves moving all processes towards one end of memory and all holes move to another direction producing one large hole of available memory. However, this is pretty expensive and cumbersome.

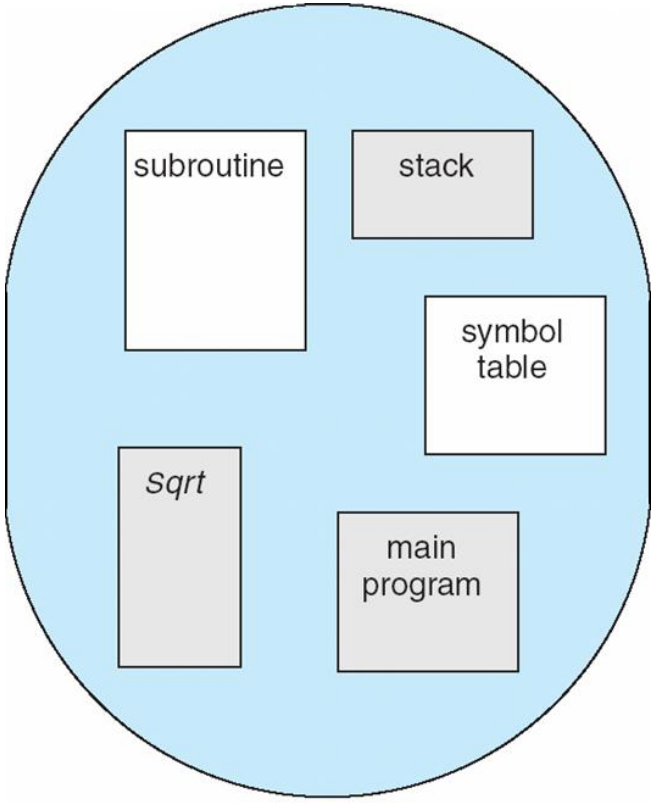
**Permit the logical address space to be noncontiguous thus allowing a process to be allocated physical memory whereever available.** This will require concepts of segmentation and paging

# Segmentation

- The user's or programmer's view of memory is not the same as the actual physical memory.
- So we need a hardware to provide some memory mechanism to map the programmer's view to the actual physical memory. This is provided by segmentation.
- It enables more freedom to manage memory and the programmer also has a more natural programming environment.
- Thus memory is not a liner array of bytes some containing instructions and others data. Instead, it is a collection of vaiable sized segments with no necessary ordering among them.

# An example of Segmentation

- While writing a code to find squareroot of a number, a programmer would think of the following: the main program, subroutine, stack, the math library, sqrt etc.
- **He does not care which address in the memory do these elements occupy. But in reality all of these are stored somewhere in the memory in variable sized segments and the lengthof this segment is intrinsically defined by its purpose.**
- **Elements within a segment are identified by their offset from the beginning of the segment.**
- Thus we can talk about the first statement of the main program, seventh stack entry in the stack, fifth instruction in sqrt and so on.

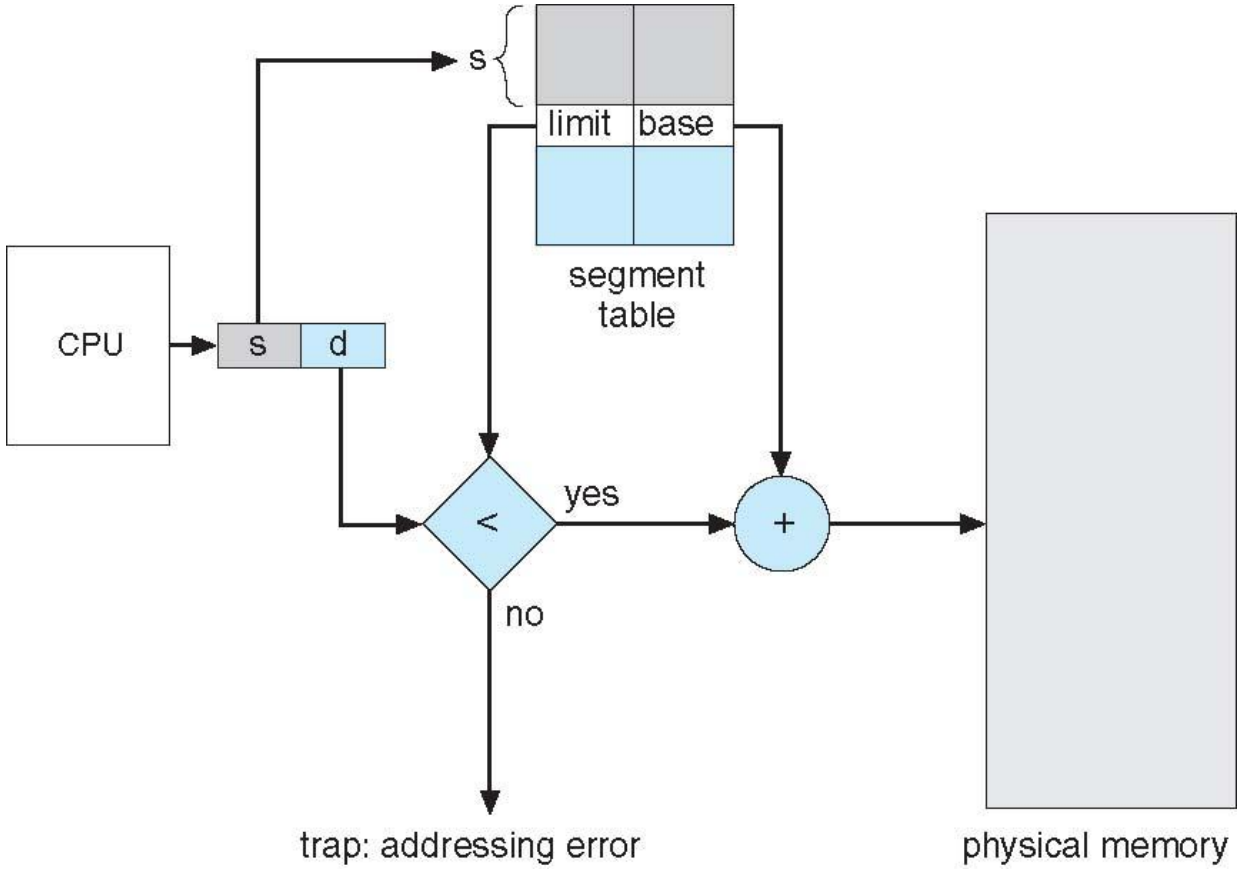


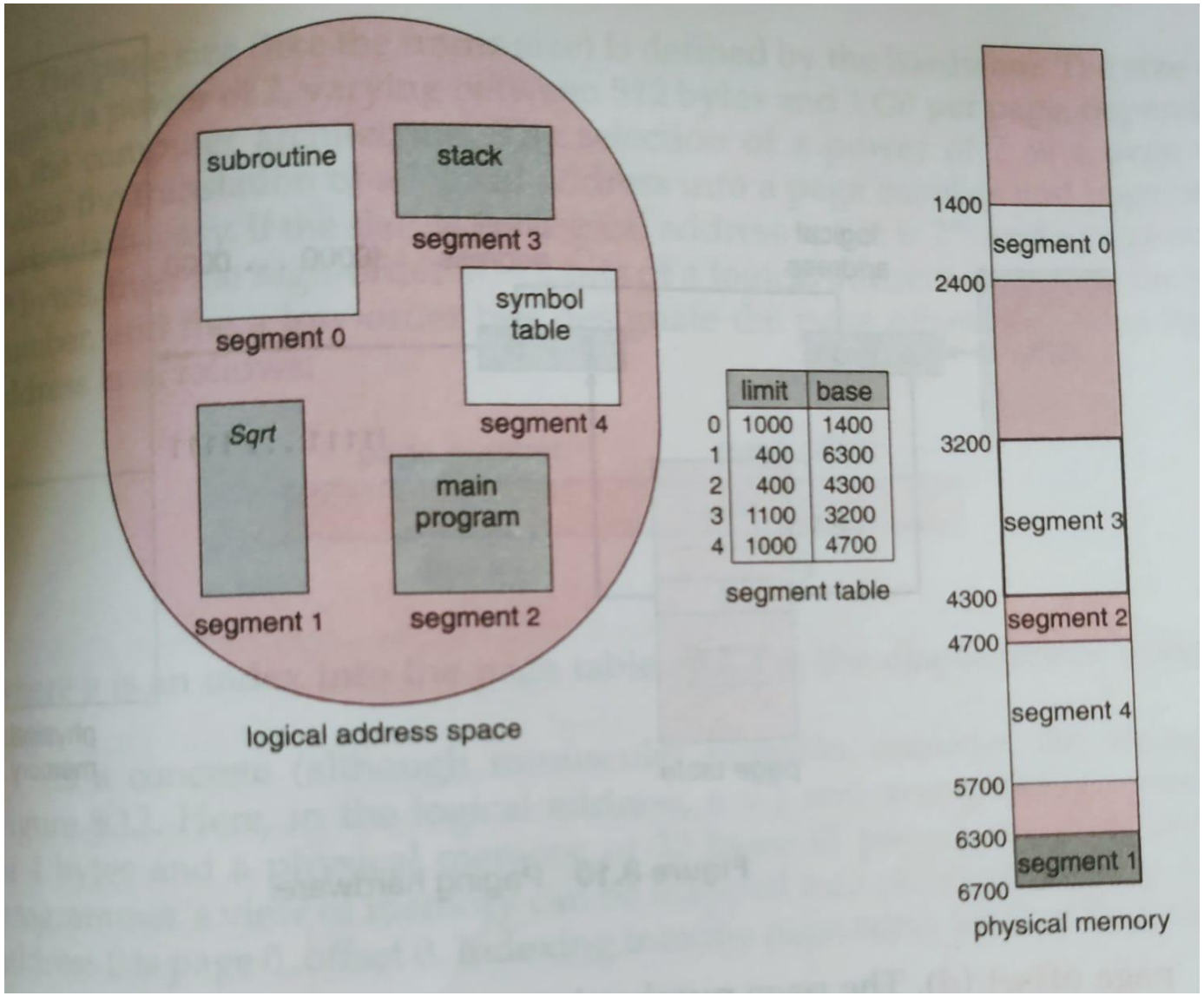
# Segmentation Architecture

- **The logical address space is a collection of segments**. Each segment has a name and a length.
- The address specify both segment name and the offset within the segment.
- Thus programmer in each address specifies 2 quantities : segment name and offset.
- But for simplicity, segments are numbered thus we now say that logical address consists of two tuples: segment number (s) and offset (d).
- **Hence logical address : <segment-number, offset>**
- **Thus your Logical address is a 2D address but the actual physical memory is still a 1D sequence of bytes.**
- **So we need a mapping between 2D programmer defined address into 1D physical address and this is provided by Segmentation Table**.

# Segmentation Hardware

- **Each entry in Segmentation Table has a Segment Base (STBR) and Segment Limit (STBL).**
- STBR has starting physical address where the segment resides in memory and STBL specifies the length of the segment used by the programmer.






- Segment 2 is 400 bytes long and begins at 4300.
- A reference to byte 52 in segment 2 is mapped to 4300 + 52 = 4352.
- A reference to byte 852 in segment 3 will be??
- A reference to byte 1222 of segment 0 will be??

# Paging

- Both segmentation and paging permits physical address space of a process to be non continuous whereas other startegies do not.
- Paging avoids external fragmentation and need for compaction whereas segmentation does not.
- Paging also solves the problem of fitting memory chunks of varying size onto the backing store whereas other strategies do not. This solution is important because when code fragments or data residing in main memory need to be swapped out, sufficient space must be there in backing store. This backing store also suffers from fragmentation problem. But in backing store, access is much slower so compaction is possible.
- Hence considering all its advantages, paging in its various forms are mostly used in OS memory management strategies.
- It is implemented through cooperation between OS and computer hardware.

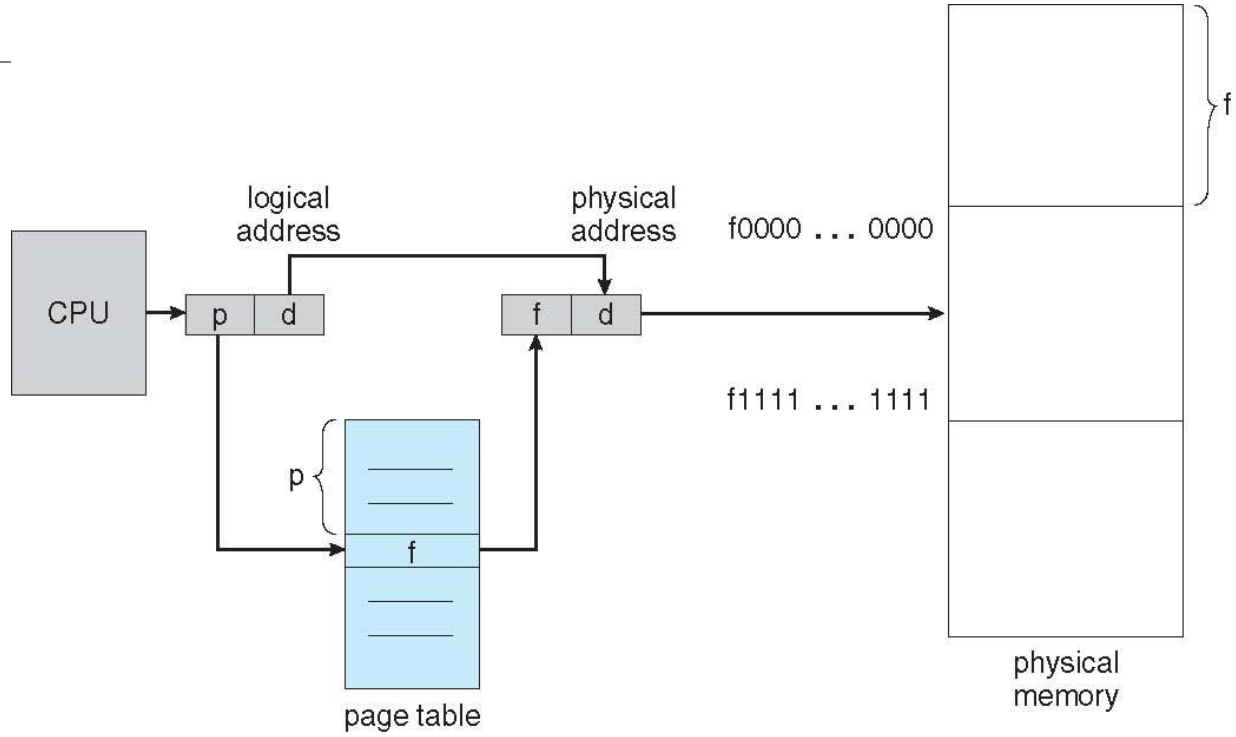
# Basic Method of Paging

- **It breaks physical memory into fixed size blocks called frames** where size is power of 2, between 512 bytes and 16 Mbytes
- **It breaks logical memory into blocks of same sizes called pages**.
- When a process is to be executed, its pages are loaded into any available memory frames from their source (file or backing store).
- **Backing store is divided into fixed size blocks that are of same size as memory frames or clusters of memory frames**
- The task is to keep track of all free frames
- To run a program of size ***N*** pages, we need to find ***N*** free frames and load program.
- Thus now a logical address can be 64 bit even though the system has less than $2^{64}$ bytes of physical memory

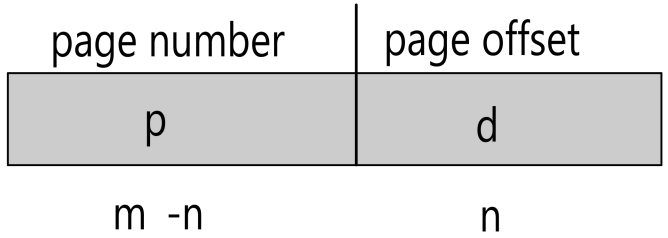
# Paging Hardware

Address generated by CPU (Logical address) is divided into two parts:

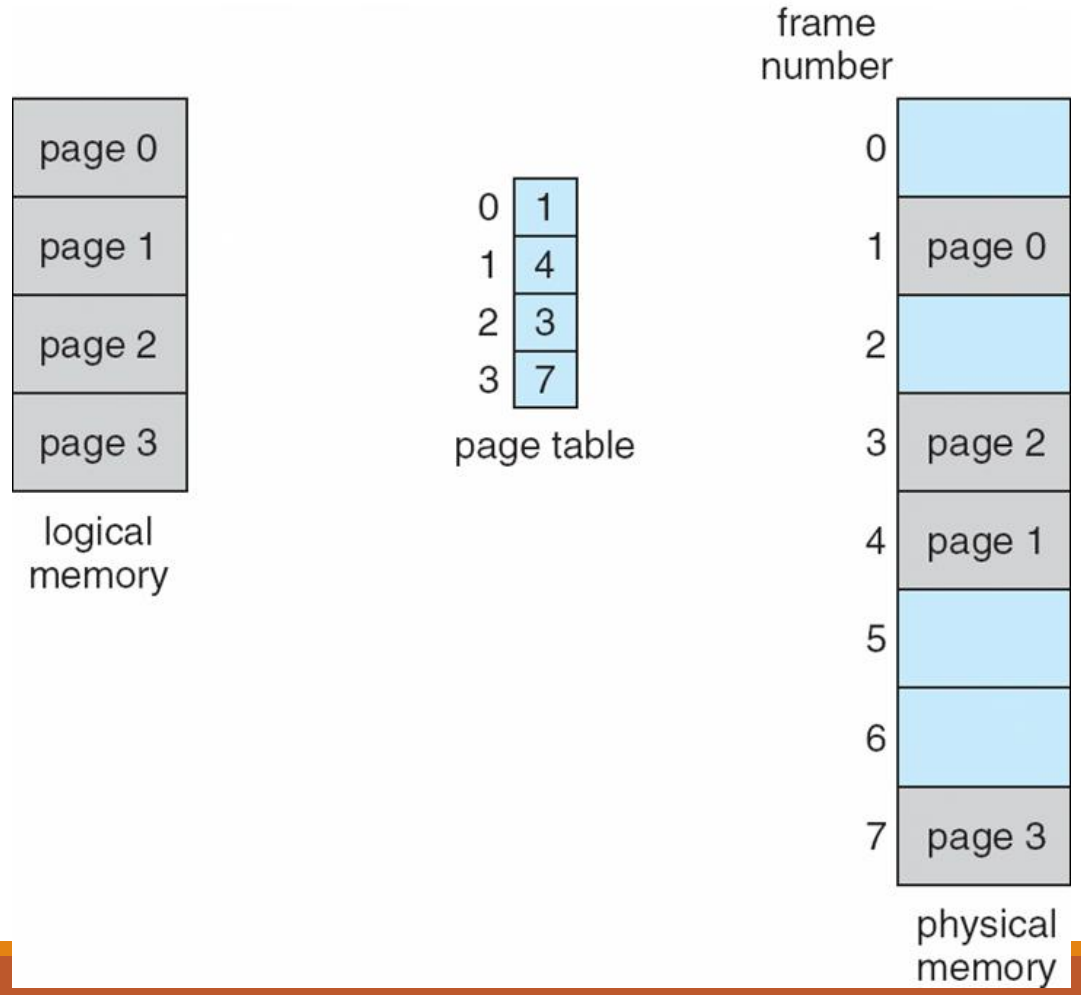
- **Page number** (***p***) – used as an index into a **page table** which contains base address of each page in physical memory (f)
- **Page offset** (***d***) – combined with base address (p) to define the physical memory address that is sent to the memory unit




**If size of logical address space is $2^m$ and page size is $2^n$ bytes, then logical address is as follows :**

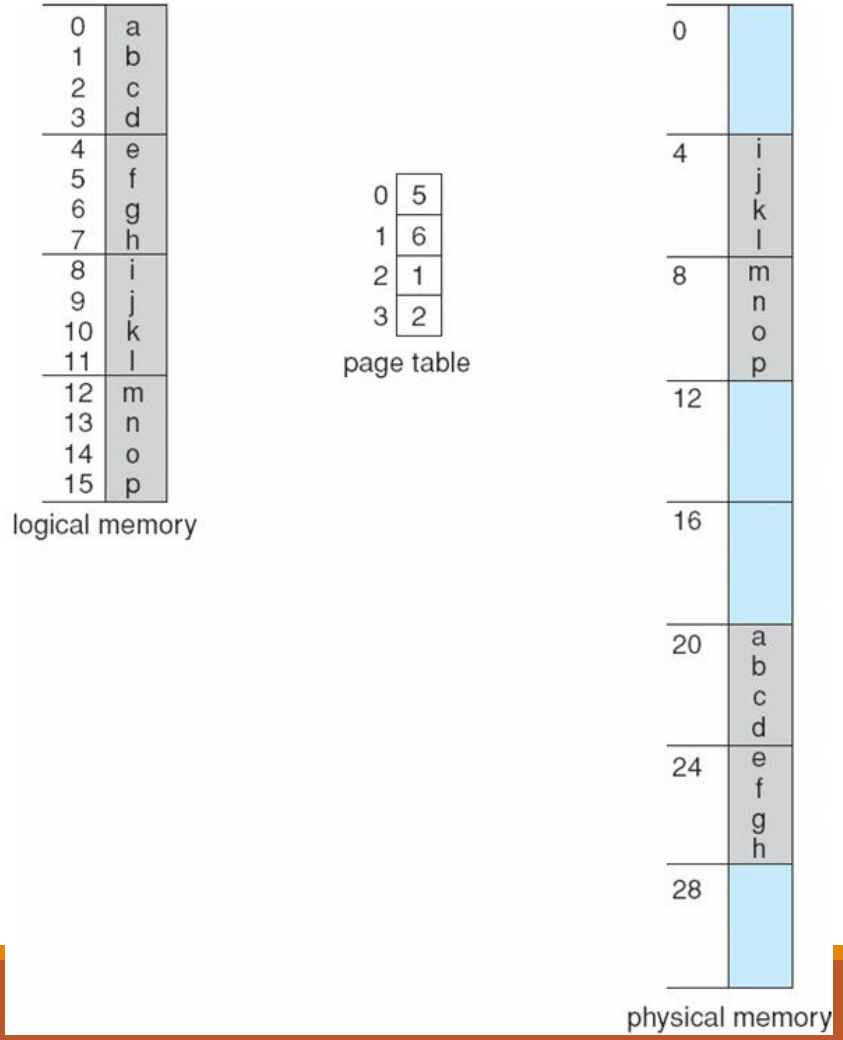


# Paging Model of Logical and Physical Memory



# Paging Example

**In logical address n = 2, m = 4. Page size is 4 bytes and physical memory of 32 bytes (8 pages)**




**Thus every LA is bound by paging hardware to some PA.**

**Using paging is identical to using a table of base register one for each frame.**

# Fragmentations in Paging

- **No external fragmentation** as whereever we have free frames, they are allocated to a needy process.
- **Internal Fragmentation**: If the memory requirement of a process does not coincide with page boundaries, te last allocated frame may not be completely full.

**Example**: If page size is 2048 bytes, process of 72766 bytes wil need _____ pages.

Then how much will be the internal fragmentation_____ bytes.

**Can you guess the worst case situation?**

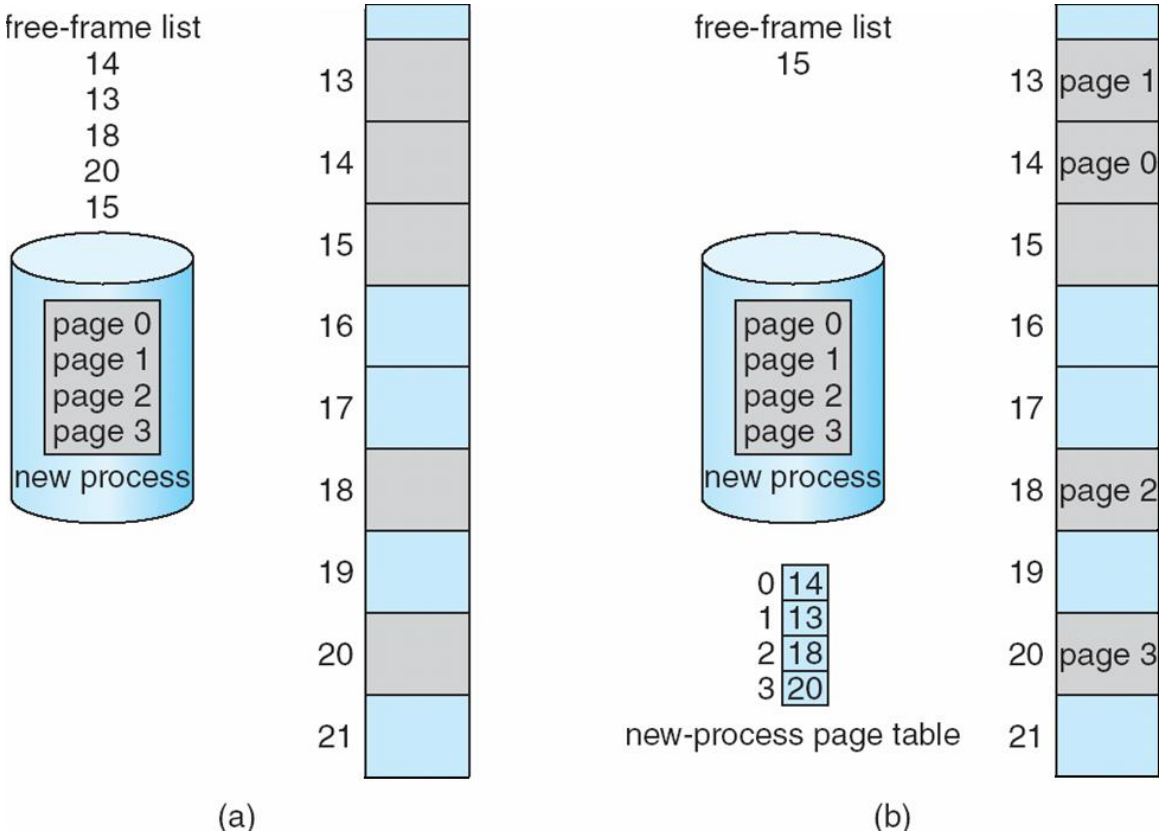
**What is the average Internal Fragmentation experienced?**

**How to suitably decide the page size?**

# Frame Table

- Since OS is managing physical memory, it must be aware of the number of free frames, number of allocated frames, total how many frames are there and so on. To manage this information, we have frame table.
- The frae table has one entry for each physical page frame, indicationg whether it is allocated ot free. If allocated then to which page of which process.

# Free Frames




Before allocation

After allocation

# Implementation of Page Table

- Page Table is kept in main memory and is implemented as a set of dedicated registers.
- A **Page Table base register (PTBR)** points to page table. Changing page tables requires changing ony this one register thus reducing context switching.
- **Page-table length register (PTLR)** indicates size of the page table.

☐**Disadvantage**: In order to access location i, we must first index into page table using PTBR offset by page number for i. Thus it involves one memory access. Now this page table gives the frame number, which is then combined with page offset to produce the actual address. So it again involves a memory access. So we need 2 memory access to reach a particular byte. Hence the activity is slowed by a factor of 2 which is really intolerable.

☐**Solution**: Use a special small fast (high speed lookup hardware cache called Translation Look-aside Buffer (TLB).

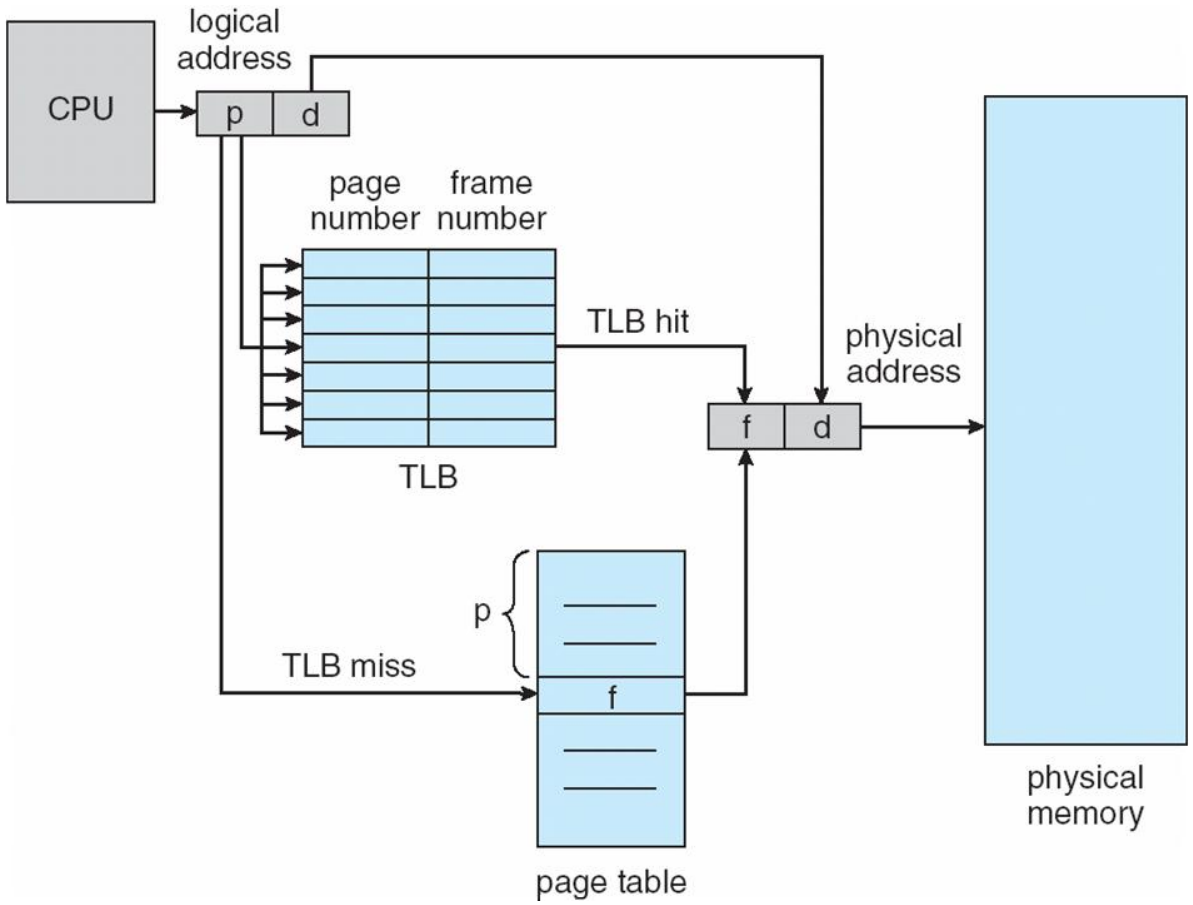
# Paging Hardware With TLB

**How does a TLB work?**

- TLB is an associative high speed memory where each entry has two parts: key (tag) and a value.
- Whenever it gets an item, this item is compared with all keys simultaneously.
- If the item is found, the corresponding value field is returned.
- Thus thesearch is fast.
- For this reason, the size of TLB must be small between 32 and 1024 entries only)

# How is TLB used in Page Tables?

- 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.
- Remember, these steps are executed as a part of the instruction pipeline within the CPU this adding no performance penalty.
- **If the page number is not found (TLB miss)**, a memory access to the page table is made. Finally when the frame number is obtained, it used to acess the memory.
- After a TLB miss, we add the page number and corresponding frame number to the TLB so that they will be found quickly on the next reference.



# Creating the TLB

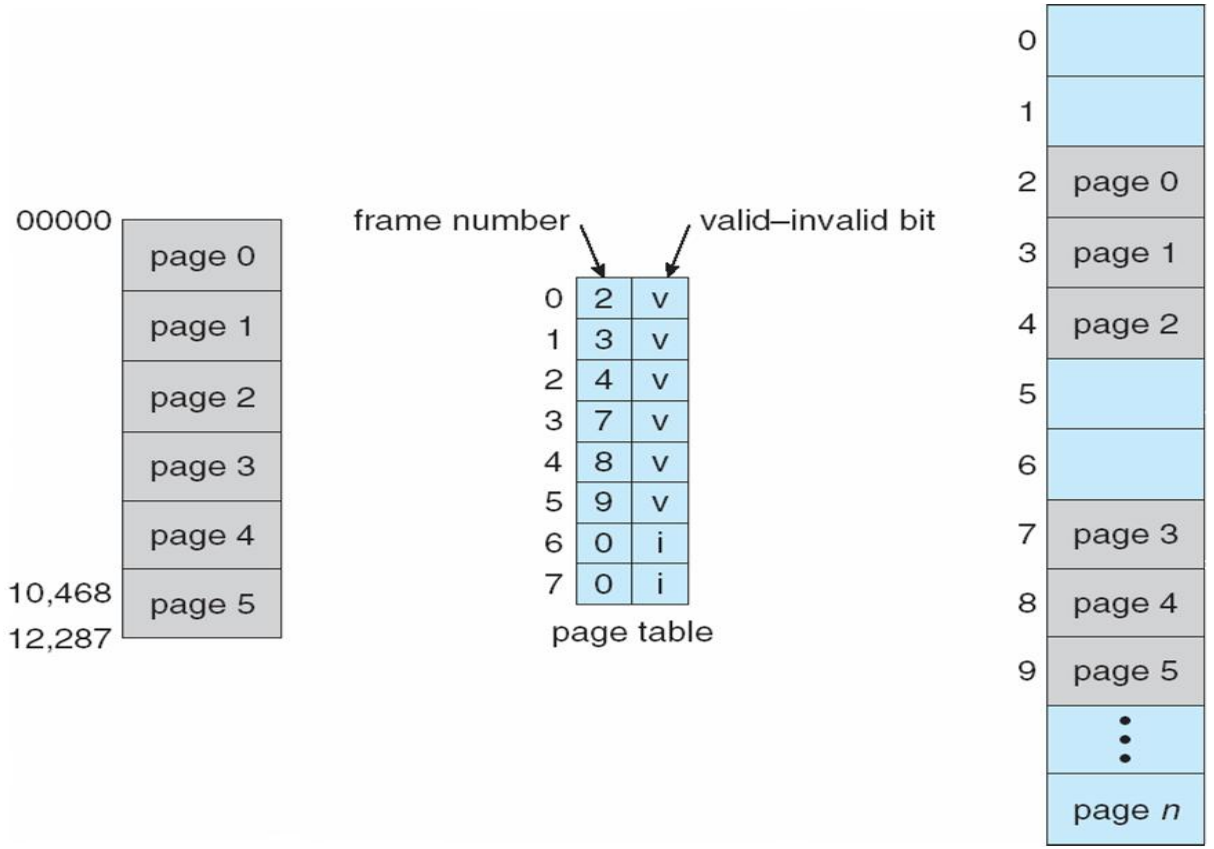
- A TLB accpets only certain entries of the page table. So after a TLB miss when this new entry is to added to the TLB, it is problematic if the TLB is already full.
- In such situation, an existing entry must be selected for **replacement** using policies such as - LRU (least recently used), Round Robin or absolutely random.
- Some CPU allows OS to participate in this replacement, or they handle it themselves.
- Some TLBs make certain entries **wired down** : they cannot be removed from the TLB (Eg: TLB for kernel code)
- Some TLB use **address space identifiers (ASIDs)** : uniquely identifies each process to provide address-space protection for that process.
- From time to time the TLB is also **flushed** to avoid any incorrect or invalid entries.

**Numerical on Hit Ratio discussed on whiteboard**

# Memory Protection in Paging

- Memory protection in a paged environment is accomplished by proection bits associated with each frame. These bits are also kept in page table.
- One protection bit can define a page to be read - write or read - only or execute - only. It ensures that no write is being made to a read-only page otherwise a trap addressing error is invoked.
- **One additional bit is generally attached to each entry in page table : valid - invalid bit.**
- **When this bit is valid** : associated page is in the process's logical address space and is thus legal.
- **When this bit is invalid**: page is not in the process's logical address space and is thus illegal.
- The OS sets this bit for each page to allow or disallow access to the page.

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

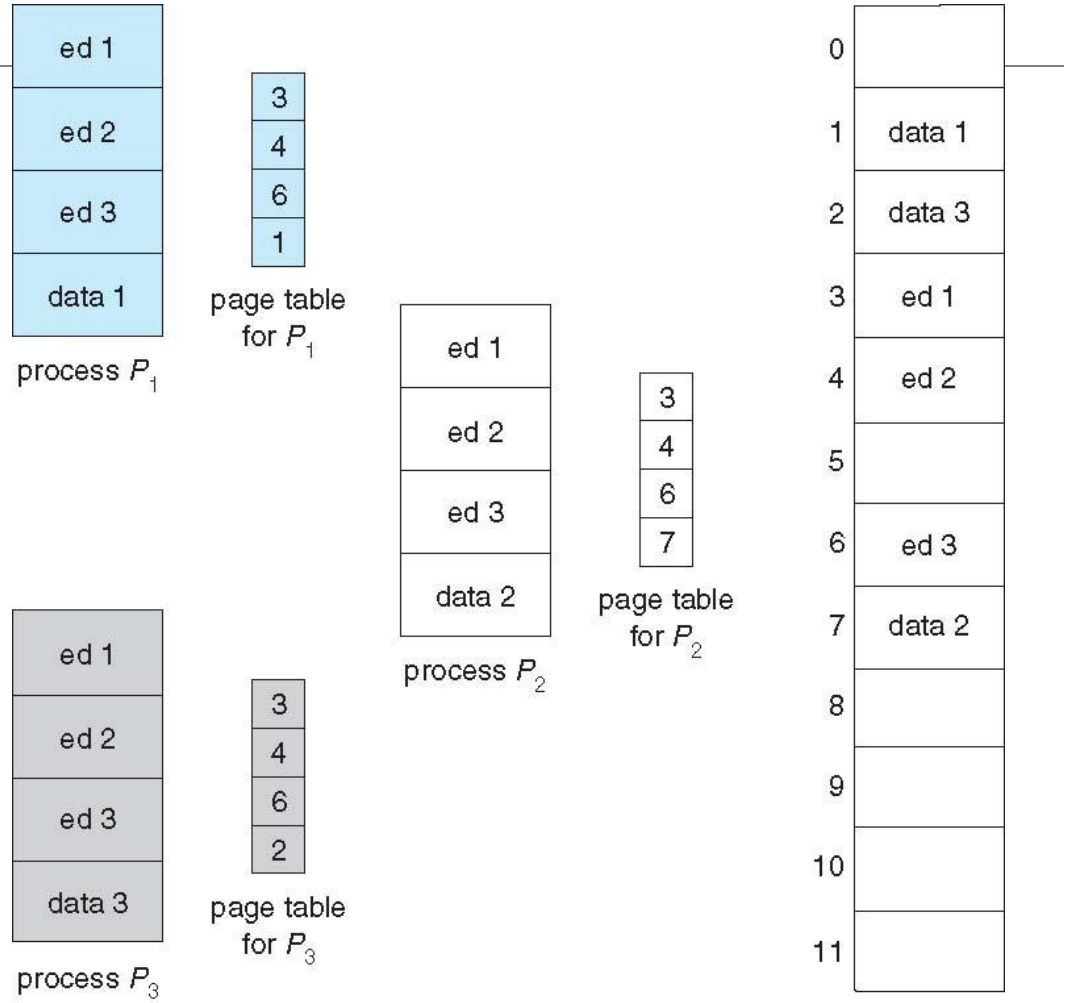



**Disadvantage??**

# Shared Pages

- Advantage of paging is the possibility of sharing common code.
- If the code is **reentrant code (pure code)**, it can be shared.
- Such reentrant code is non-self-modifying code thus it never changes during execution. So multiple processes can execute the same code simultaneously.
- **But even though, processes share the same code, the data required by them is obviously different.**

**Proof related to Advantage discussed in whiteboard**



# Structure of Page Tables

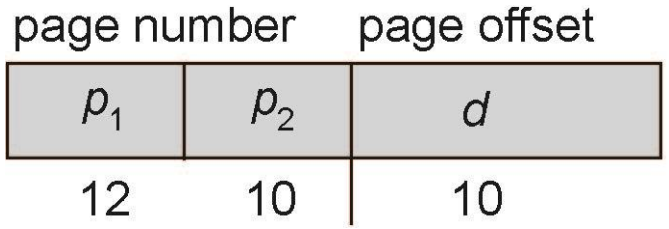
- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

# Hierarchical Paging

- Modern computers support a large logical address space ($2^{32}$ to $2^{64}$).
- Corresponding to this, the page table will also be excessively large.
- To store this huge page table, again a lot of physical memory will be used up.
- So definitely we must not allocate the page table contiguously in main memory.
- The solution is to divide the page tables also into smaller pieces.
- Thus we can have a multi level paging where the page table itself is also paged.

# Two Level Hierarchical Paging

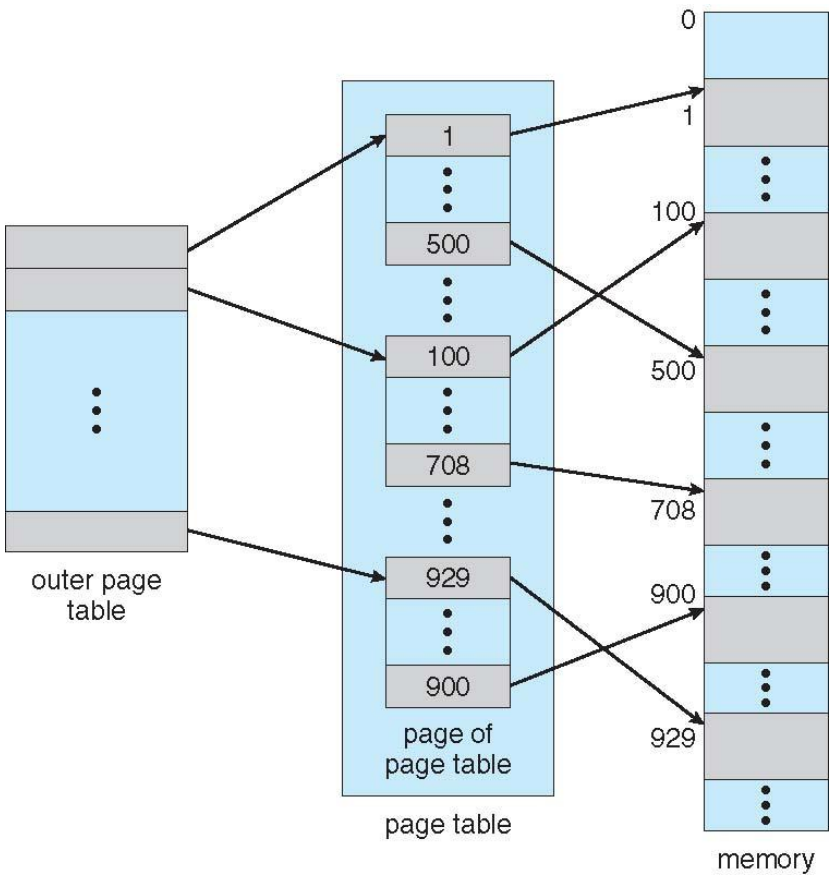
- A logical address (on 32-bit machine with 1K page size) is divided into: a page number consisting of 22 bits and a page offset consisting of 10 bits
- Since the page table is paged, the page number is further divided into: a 12-bit page number and a 10-bit page offset
- Thus, a logical address is as follows:



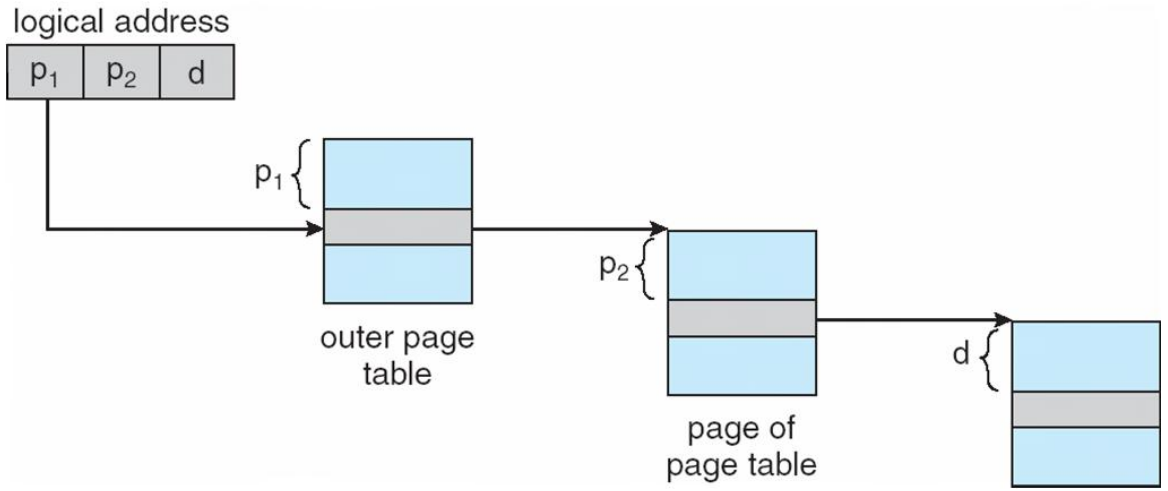

**where p1 is an index into the outer page table, and p2 is the displacement within the page of the inner page table**

**Because address translation works from the outer table inwards towards other tables, we call it Forward mapped page table.**

# Two-Level Page-Table Scheme

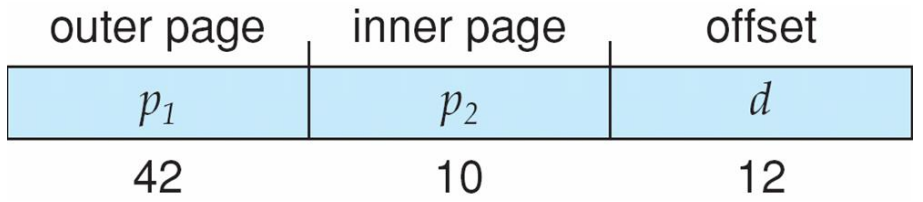



**Corresponding Address Translation Scheme**



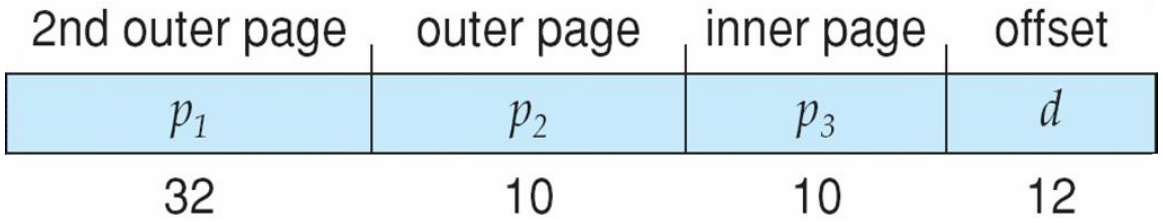
# Example: 64 bit Logical Address space

- Logical Address Space : 64 bit i.e $2^{64}$. If page size is 4KB i.e $2^{12}$. Thus page table consists of $2^{52}$ entries.
- If two level scheme, inner page tables could conveniently be of 1 page i.e contain $2^{10}$ 4-byte entries.
- The address would look like :



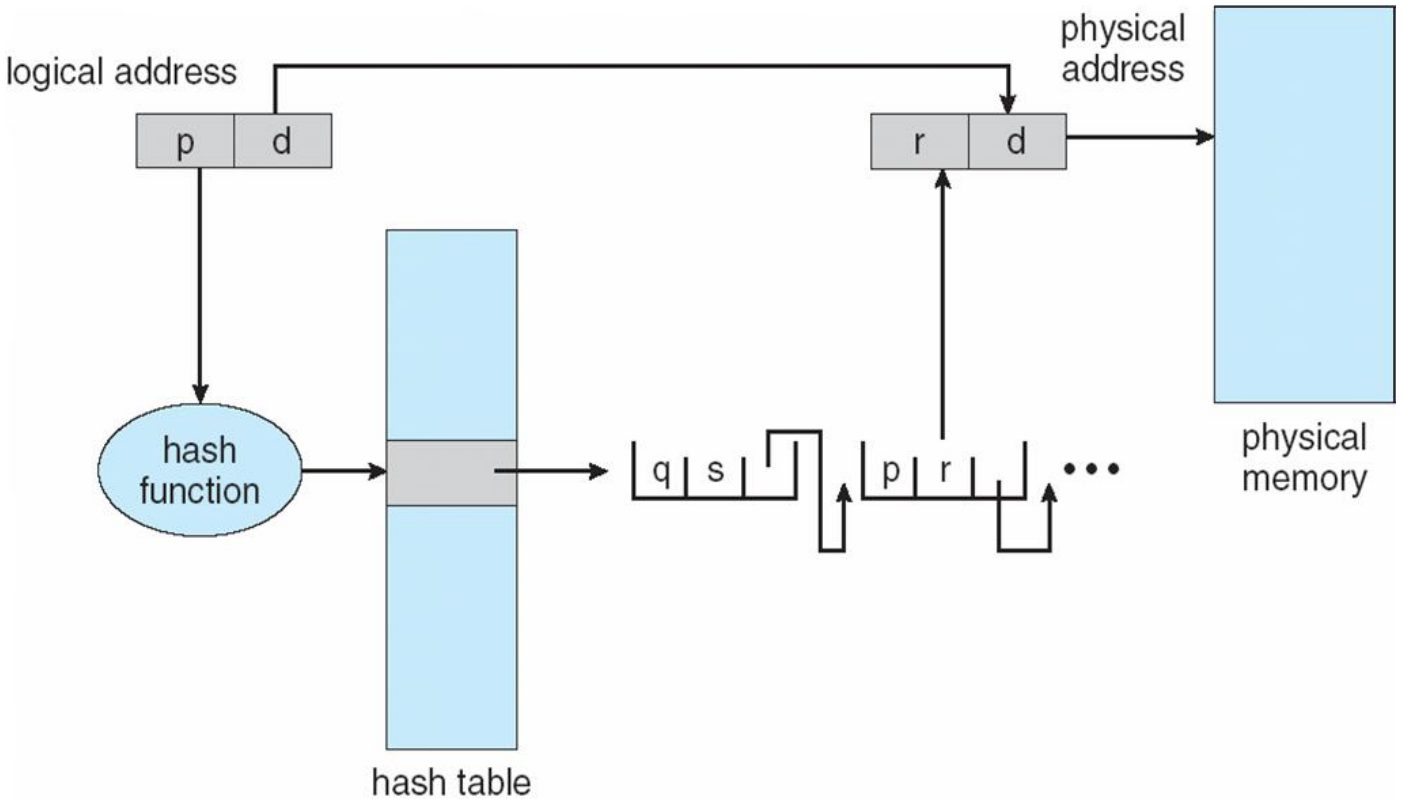

- The outer page table thus has $2^{42}$ entries which is also huge.
- So to prevent such huge outer table, we will divide it into further smaller pieces.Thus giving a three level paging scheme. Thus address would look like:

**Disadvantage of Hierarchical Paging?**



# Hashed Page Tables

- Common approach to handle LAS larger than 32 bits is to use **hashed page table** where the hash value is the vitual page number.
- Each entry in the hash table contains a linked list of elements that hash to the same location to handle collisions.
- **Each element has three fields : the virtual page number, value of mapped page frame, pointer to the next element in the linked list.**




## What are Clustered Page Tables?

- The virtual page number is hashed into the hash table using a hash function.
- This is then compared with field 1 in the first element of the linked list.
- If there is a match, the corresponding page frame is read from the field 2.
- This is then used to form the desired physical address using the offset.
- If there is no match, subsequent entries in the linked list are searched for a matching virtual page number.

# Need of Inverted Page Tables

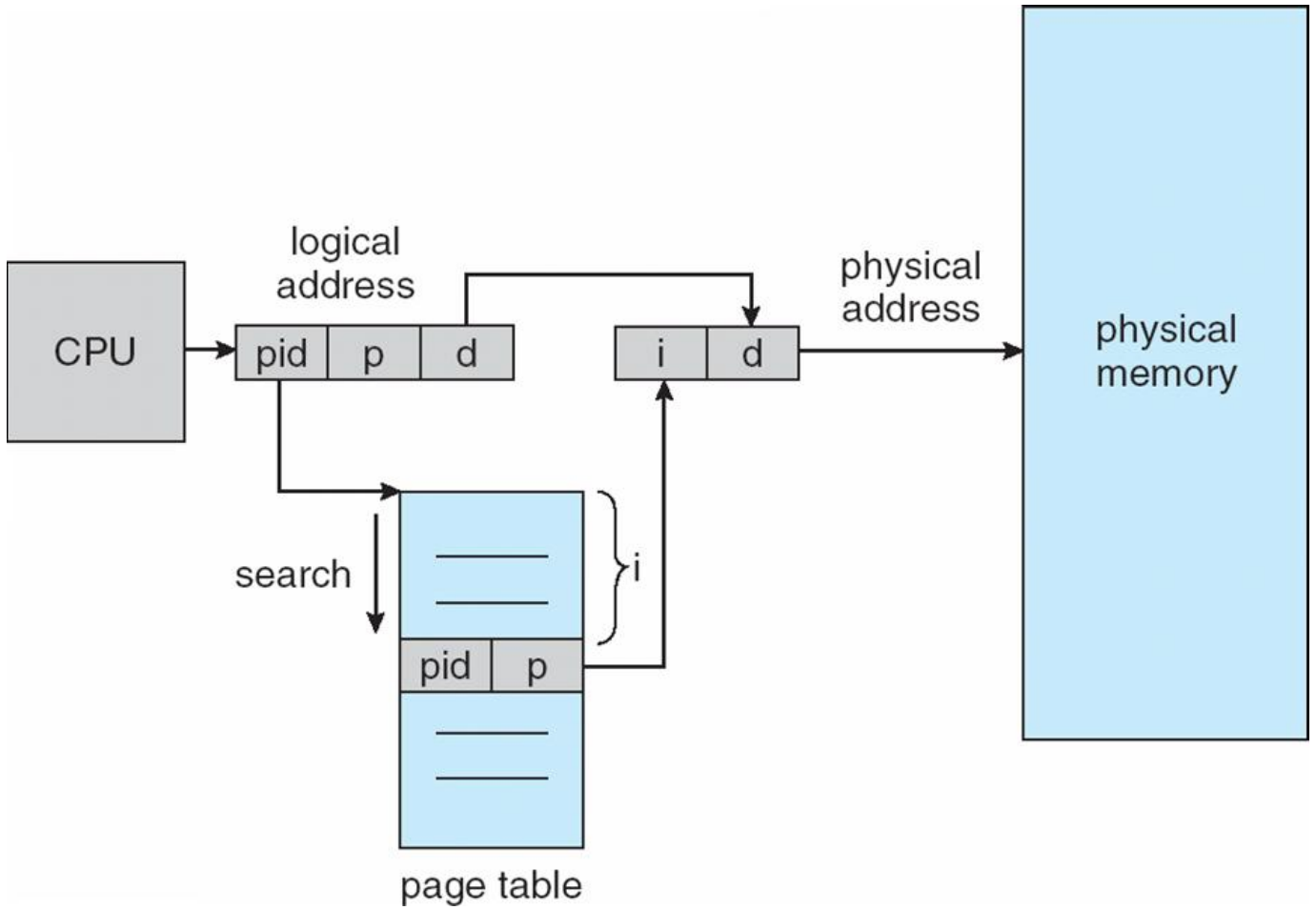
- Each process has an associated page table. This page table has one entry for each page that the process is using.
- This table representation is a natutal one as processes reference pages through the page's virtual address. The OS translates this reference into physical memory address.
- Since the table is sorted by virtual address, the OS can calculate where in the table the associated physical address entry is located and to use that value directly.
- **Drawback is that each page table has millions of entries and thus consume large amount of physical memory just to keep a track of how other portions of physical memory is used.**
- To solve this, we use **inverted page tables**.

# Inverted Page Tables

- An inverted page table has one entry for each real page or frame of physical memory.
- Each entry consists of the virtual address of page stored in that real memory location, with information about the process that owns the page.
- So we have only one page table in the system and it has only one entry for each page of physical memory.

**Advantage of Inverted Page Tables**: Decreases amount of memory to store each page table.
**Disadvantage of Inverted Page Tables**: a) Increases amount of time needed to search the table when page reference occurs. Inverted page tables are sorted by physical address but lookups occur on basis of virtual address so whole table might need to be searched before a match is found **(Solution??)**

b) Difficulty to implement shared memory as multiple virtual addresses are mapped to one physical page. But in inverted page table we have, one virtual page entry for every physical page so one physical page cannot have two shared virtual pages. **(Solution??)**




- **Each virtual address has three fields: <pid, page number and offset>**
- **Each inverted page table entry is a pair: <pid, page number> where pid acts as the address identifier.**
- When a memory reference occurs, portion of the virtual address consisting of pid and page number is presented to the memory subsystem.
- The inverted page table is then searched for a match.
- **If match is found, at suppose entry i, then physical address <i, offset> is generated.**
- If no match found, then illegal address access has been attempted.