UNIT 3
Memory Management

Course Learning Rationale	Emphasize the importance of Memory Management concepts of an Operating system
Course Learning Outcomes	Understand the need of Memory Management functions of an Operating system

Topics

1	**MEMORY MANAGEMENT: Memory Management: Logical Vs Physical address space, Swapping**
	Understanding the basics of Memory management
2	Contiguous Memory allocation – Fixed and Dynamic partition
	Getting to know about Partition memory management and issues: Internal fragmentation and external fragmentation problems
3	Strategies for selecting free holes in Dynamic partition
	Understanding the allocation strategies with examples
4	Paged memory management
	Understanding the Paging technique.PMT hardware mechanism
5	Structure of Page Map Table
	Understanding the components of PMT
6	Example : Intel 32 bit and 64 –bit Architectures
	Understanding the Paging in the Intel architectures
7	Example : ARM Architectures
	Understanding the Paging with respect to ARM
8	Segmented memory management
	Understanding the users view of memory with respect to the primary memory
9	Paged segmentation Technique
	Understanding the combined scheme for efficient management

SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
Deemed to be University u/s 3 of UGC Act, 1956

Basics of Memory Management

Memory : Basics

- Memory is central part for the operation of a modern computer system
- 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 which may need additional loading from and storing to specific memory addresses.
- A typical instruction-execution cycle, first fetches an instruction from memory. The instruction is then decoded and may cause operands to be fetched from memory.
- After the instruction has been executed on the operands, results may be stored back in memory.

Courtesy : Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, Operating systems, 9th ed., John Wiley & Sons, 2013

Memory : Basics (Cont.)

- The memory unit sees only a
  - Stream of memory addresses;
  - It does not know how they are generated (by the instruction counter, indexing, indirection, literal addresses, and so on) or
  - What they are for (instructions or data).
- Hence we can ignore *how* a program generates a memory address.
- We are interested only in the sequence of memory addresses generated by the running program.

Courtesy : Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, Operating systems, 9th ed., John Wiley & Sons, 2013

Basic Hardware

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

Courtesy : Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, Operating systems, 9th ed., John Wiley & Sons, 2013

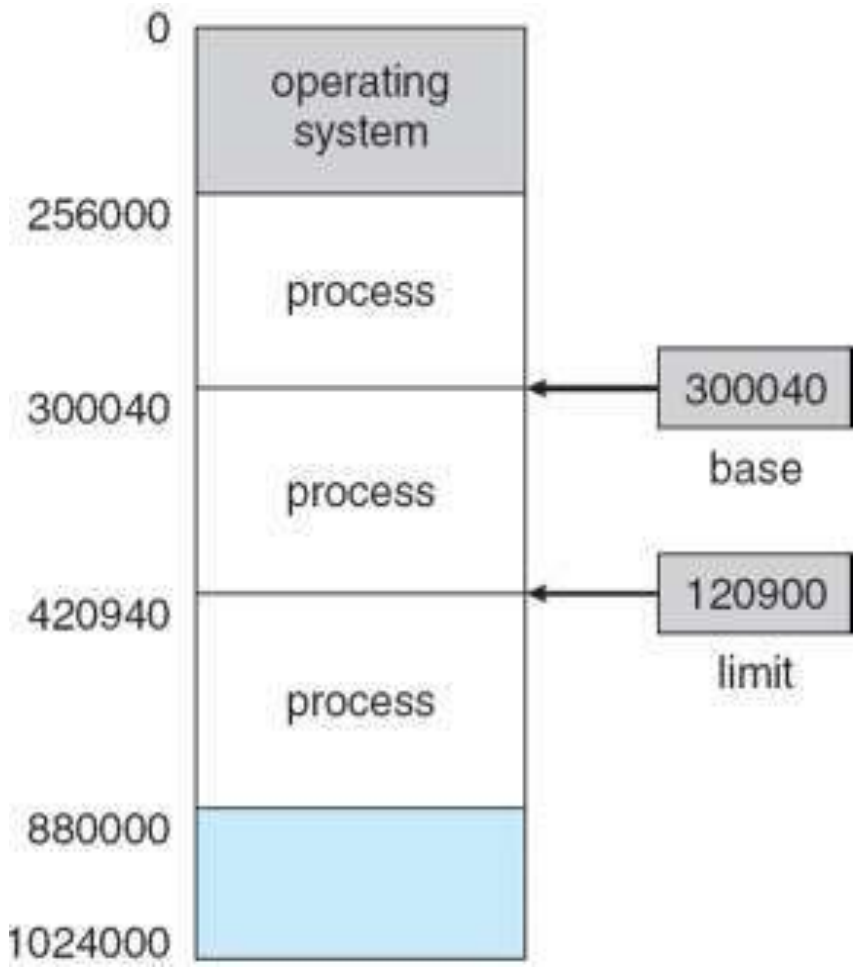
A Base and a Limit Register

- Each process should have a separate memory space.
- To do this, we need the ability to determine the range of legal addresses that the process may access and to ensure that the process can access only these legal addresses.
- Operating system provide this protection by using two registers:
  - a base and
  - a limit
- The **base register** holds the smallest legal physical memory address
- The **limit register** specifies the size of the range

Courtesy : Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, Operating systems, 9th ed., John Wiley & Sons, 2013

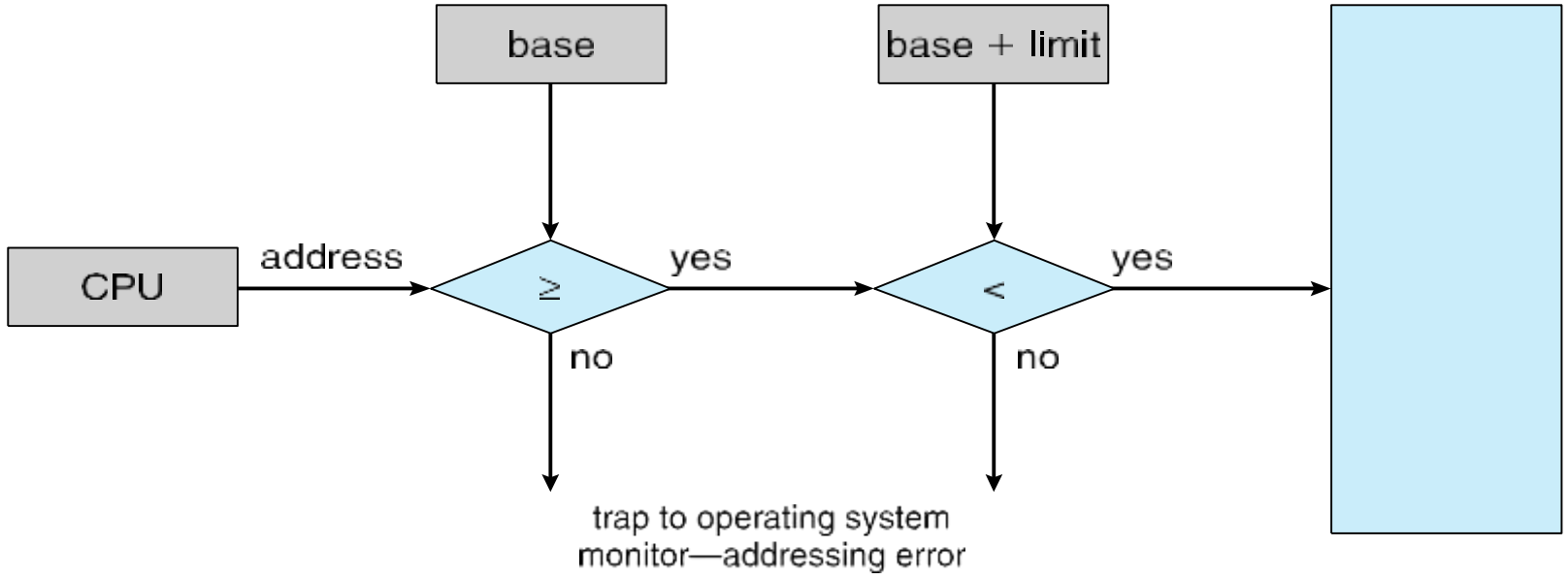
A Base and a Limit Register

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



Courtesy : Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, Operating systems, 9th ed., John Wiley & Sons, 2013

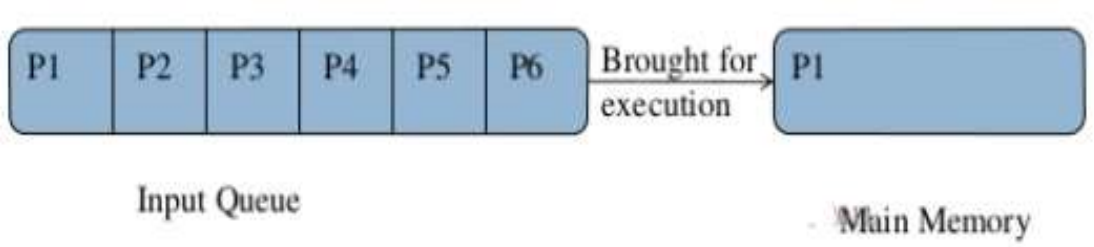
Hardware Address Protection



Courtesy : Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, Operating systems, 9th ed., John Wiley & Sons, 2013

Address Binding

- Usually, a program resides on a disk as a binary executable file.
- The program to be executed must be brought into memory and placed within a process.
- Depending on the memory management in use, the process may be moved between disk and memory during its execution.
- The processes on the disk that are waiting to be brought into memory for execution form the **input queue**.



Courtesy : Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, Operating systems, 9th ed., John Wiley & Sons, 2013

Address Binding

- Addresses may be represented in different ways during the execution of the program
- Addresses in the source program are generally symbolic (such as *count*)
- A compiler will typically **bind** these symbolic addresses to relocatable addresses (such as “14 bytes from the beginning of this module”)
- The linkage editor or loader will in turn bind the relocatable addresses to absolute addresses (such as 74014)
- Each binding is a mapping

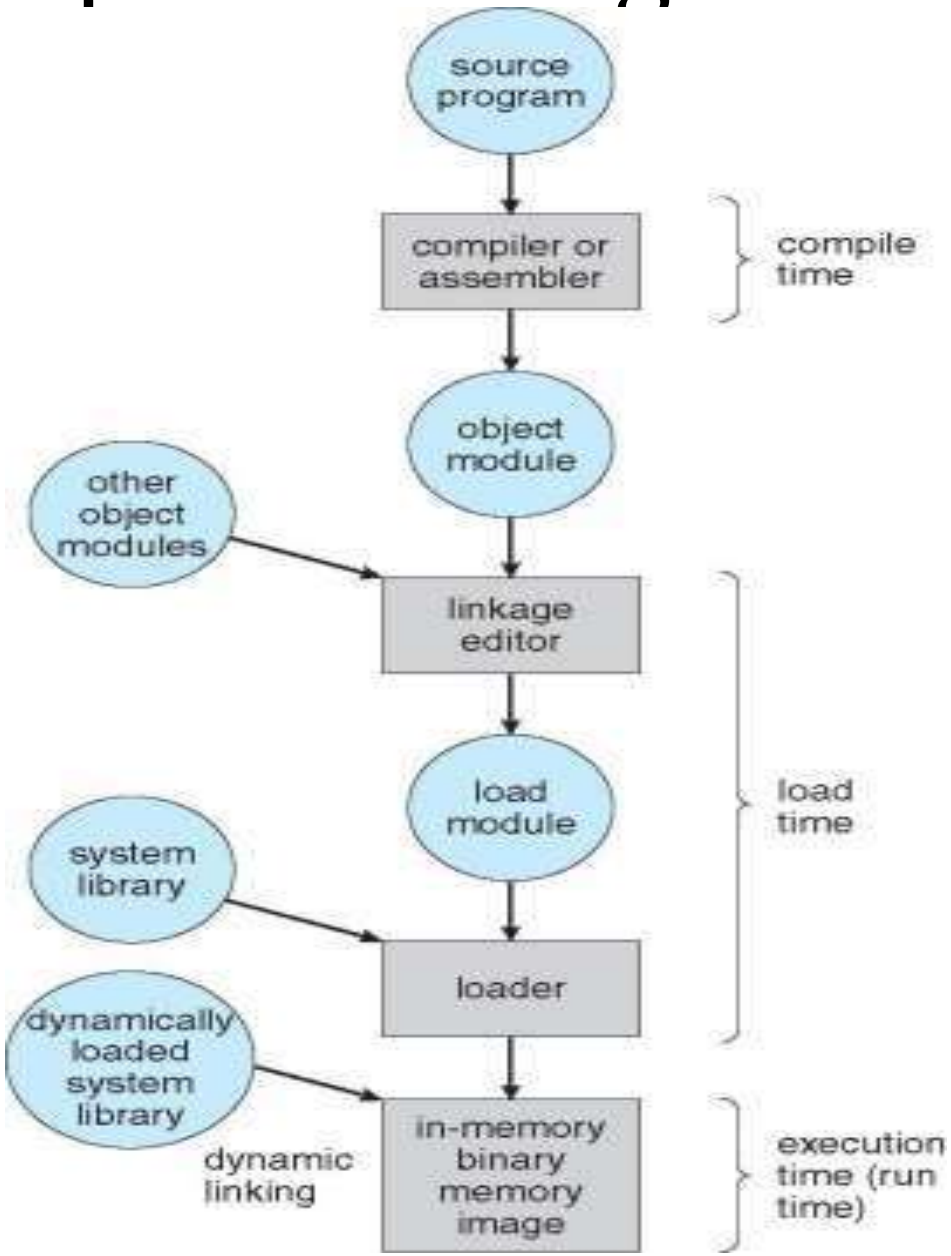
Courtesy : Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, Operating systems, 9th ed., John Wiley & Sons, 2013

Binding of Instructions and Data to Memory

- Address binding of instructions and data to memory addresses can happen at three different stages
  - **Compile time**: If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
  - **Load time**: Must generate **relocatable code** if memory location is not known at compile time
  - **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another
    - Need hardware support for address maps (e.g., base and limit registers)

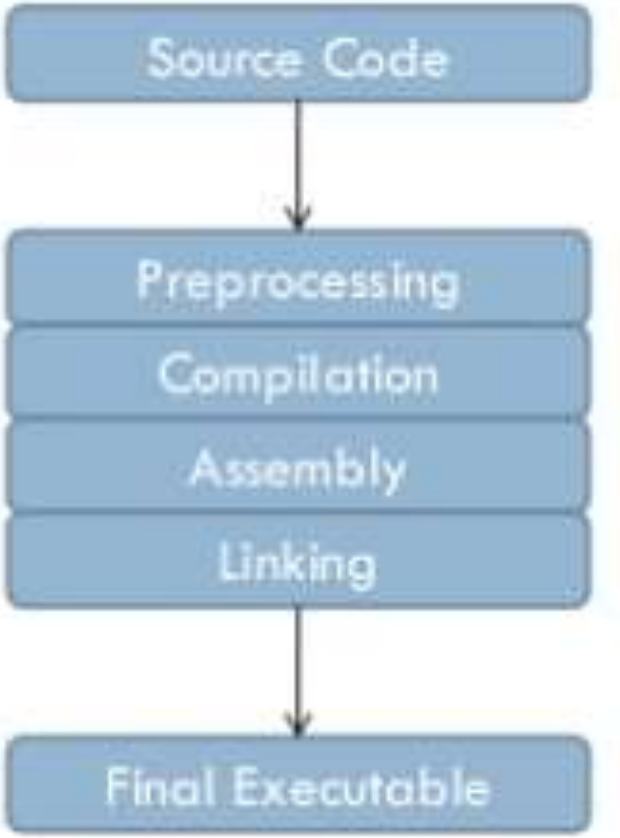
Courtesy : Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, Operating systems, 9th ed., John Wiley & Sons, 2013

Multistep Processing of a User Program

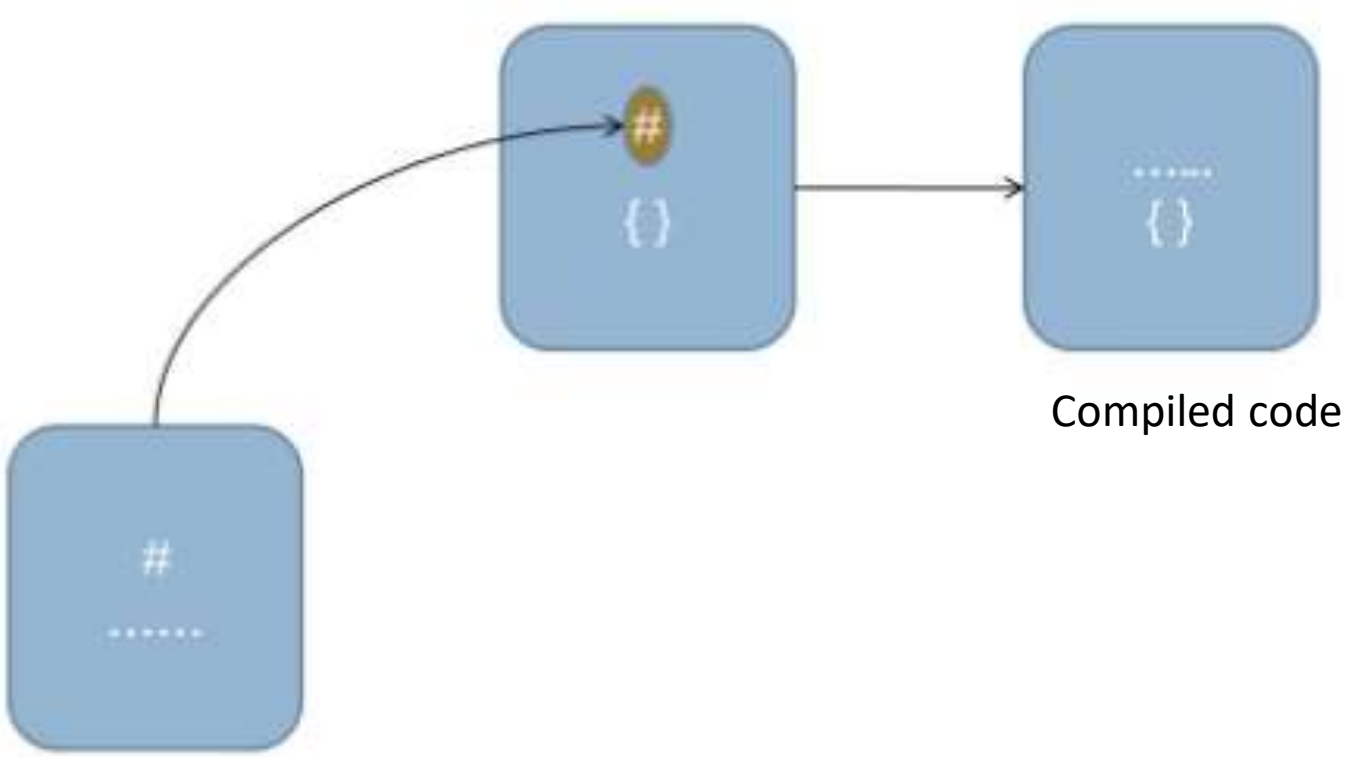


Courtesy : Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, Operating systems, 9th ed., John Wiley & Sons, 2013

Program Execution



Compilation

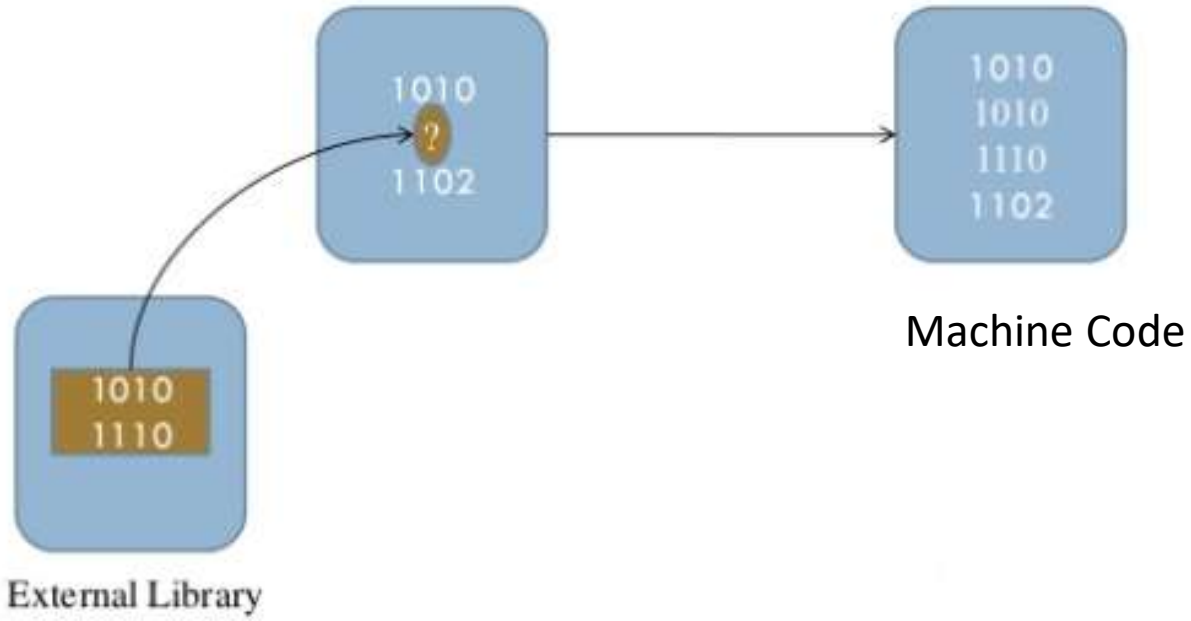


Assembler

Assembly Code

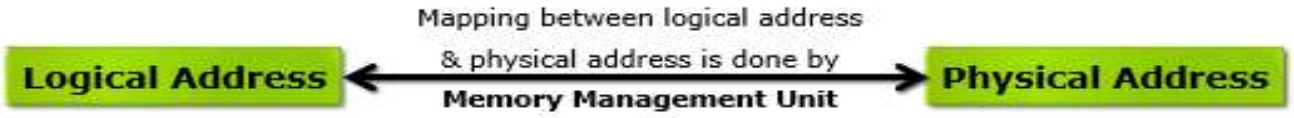
Assembler Leaves the address of external function undefined

Linker/Loader



Logical vs Physical Address Space

- An address generated by the CPU is commonly referred to as a **logical address**
- An address seen by the memory unit—that is, the one loaded into the **memory-address register** of the memory—is commonly referred to as a **physical address**.
- The compile-time and load-time address-binding methods generate identical logical and physical addresses.
- However, the execution-time address binding scheme results in differing logical and physical addresses.



Courtesy : Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, Operating systems, 9th ed., John Wiley & Sons, 2013

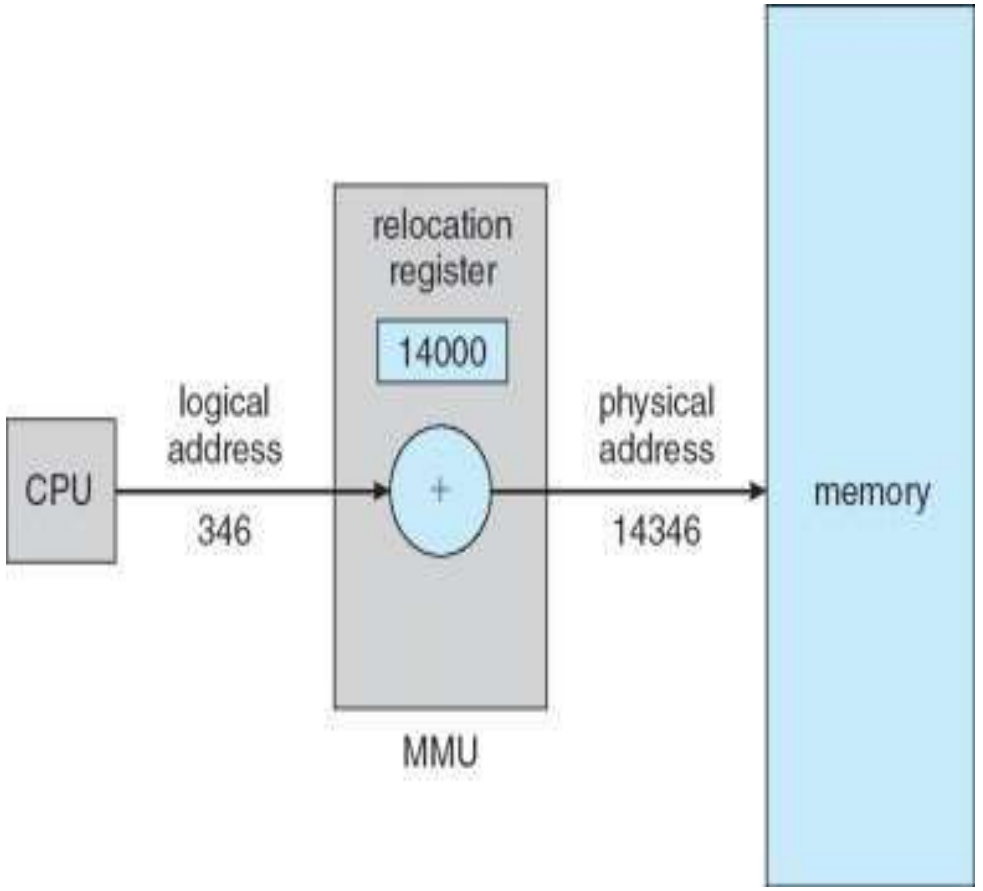
Logical vs Physical Address Space (Cont.)

- In this case, we usually refer to the logical address as a **virtual address**.
- The set of all logical addresses generated by a program is a **logical address space**;
- The set of all physical addresses corresponding to these logical addresses is a **physical address space**.
- The run-time mapping from virtual to physical addresses is done by a hardware device called the **memory-management unit (MMU)**.

Courtesy : Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, Operating systems, 9th ed., John Wiley & Sons, 2013

Dynamic Relocation using a Relocation Register

- The base register is now called a **Relocation Register**.
- The value in the relocation register is *added* to every address generated by a user process at the time the address is sent to memory.
- 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.



Courtesy : Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, Operating systems, 9th ed., John Wiley & Sons, 2013

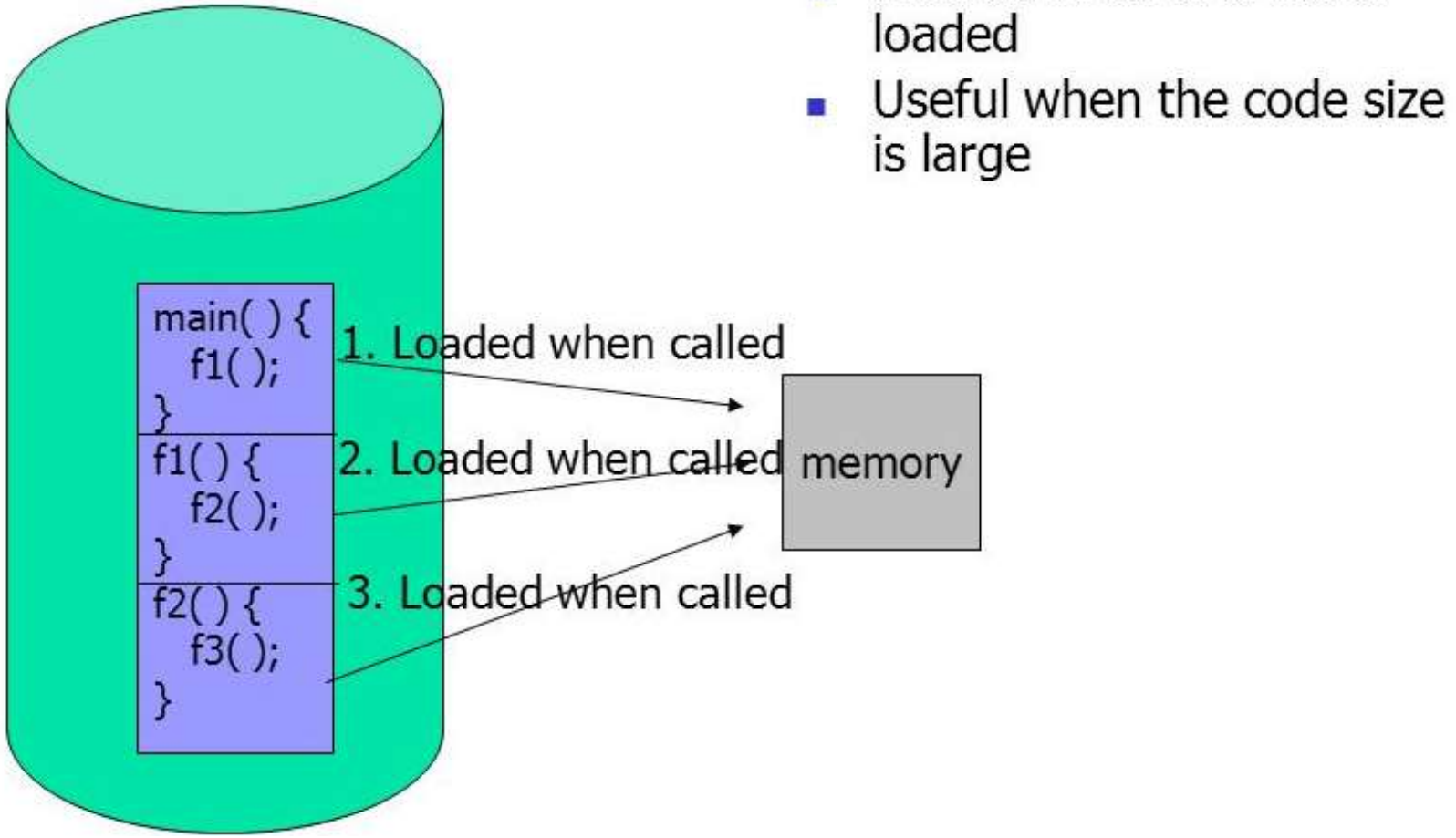
Logical vs Physical Address Space (Cont.)

- The two different types of addresses:
  - logical addresses (in the range 0 to *max*) and
  - physical addresses (in the range $R + 0$ to $R + max$ for a base value R)
- The user generates only logical addresses and thinks that the process runs in locations 0 to *max*.
- The user program generates only logical addresses and thinks that the process runs in locations 0 to *max*.
- However, these logical addresses must be mapped to physical addresses before they are used.

Courtesy : Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, Operating systems, 9th ed., John Wiley & Sons, 2013

Dynamic Loading

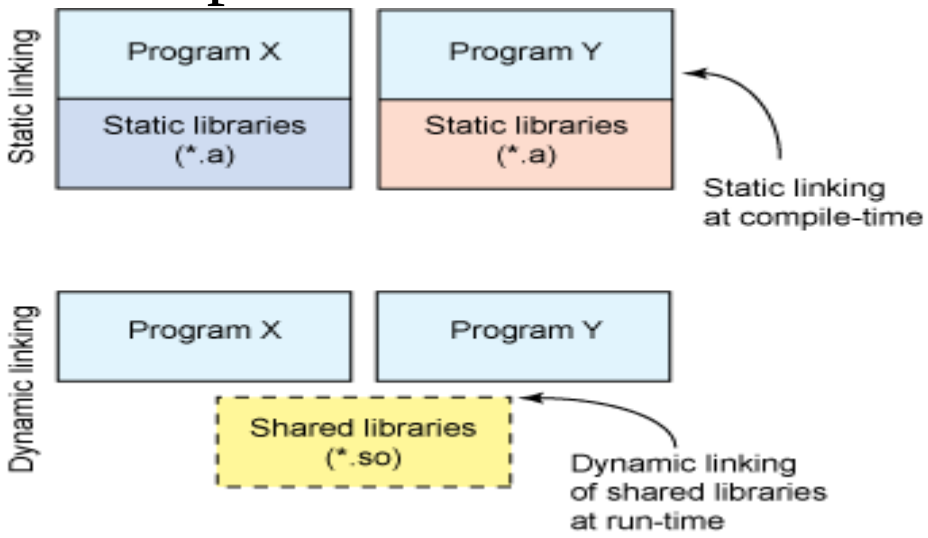
- Unused routine is never loaded
- Useful when the code size is large



Dynamic linking

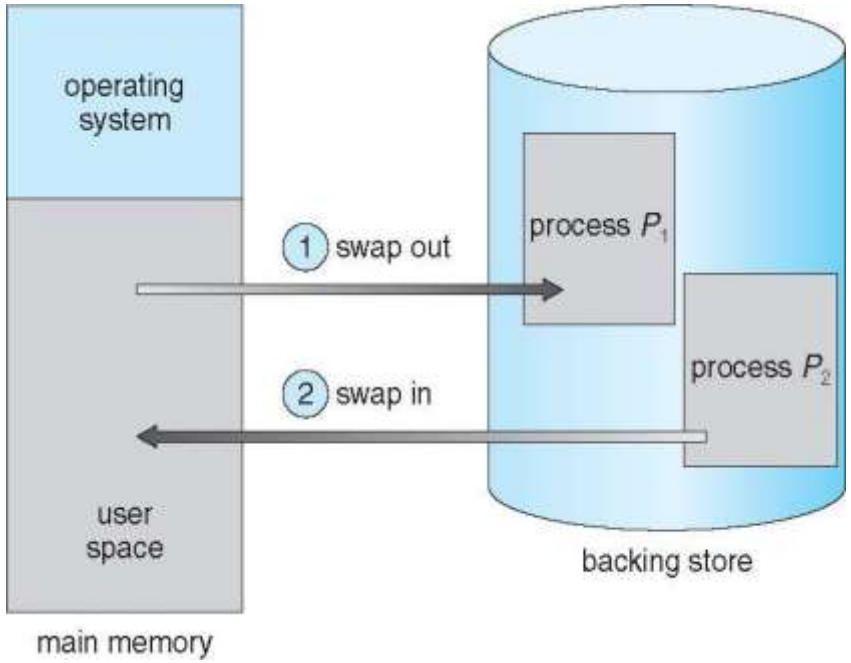
- Static linking
- System libraries and program code seperately
- Some OS support only static linking

- Dynamic linking
- Linking is postponed until execution time
- Wastage of both disk space and main memory is avoided



Swapping

- A process can be **swapped** temporarily out of memory to a backing store, and then brought back into memory for continued execution
  - Total physical memory space of processes can exceed physical memory
  - This increases the degree of multiprogramming in a system
    - **Backing store** – Normally its hard disk



CONTIGUOUS MEMORY ALLOCATION

Contiguous Allocation

- The main memory must accommodate both the operating system and the various user processes.
- The memory is usually divided into two partitions:
  1. one for the resident operating system
  2. one for the user processes.

Contiguous Allocation (Cont.)

- Several user processes want to reside in memory at the same time.
- Then it is to be considered, how to allocate available memory to the processes that are in the input queue waiting to be brought into memory.
- In **contiguous memory allocation**, each process is contained in a single section of memory that is contiguous to the section containing the next process.

Contiguous Allocation (Cont.)

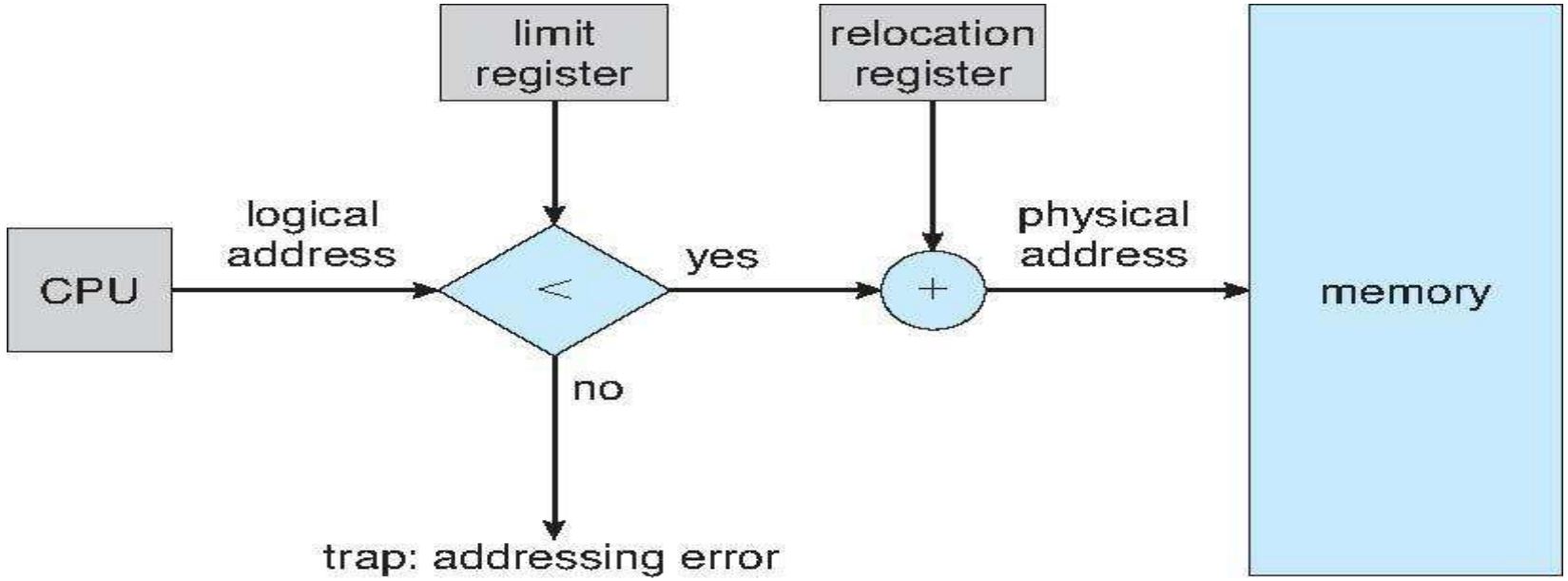
- The relocation register contains the value of the smallest physical address;
- The limit register contains the range of logical addresses (for example, relocation = 100040 and limit = 74600).
- Each logical address must fall within the range specified by the limit register.
- The MMU maps the logical address dynamically by adding the value in the relocation register.

Contiguous Allocation (Cont.)

- This mapped address is sent to memory. When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values as part of the context switch.

- Because every address generated by a CPU is checked against these registers, we can protect both the operating system and the other users programs and data from being modified by this running process.

Hardware Support for Relocation and Limit Registers



Contiguous Allocation (Cont.)

- The relocation-register scheme provides an effective way to allow the operating system's size to change dynamically. This flexibility is desirable in many situations.

- For example, the operating system contains code and buffer space for device drivers.

Contiguous Allocation (Cont.)

- If a device driver (or other operating-system service) is not commonly used, we do not want to keep the code and data in memory, as we might be able to use that space for other purposes.

- Such code is sometimes called **transient** operating-system code; it comes and goes as needed. Thus, using this code changes the size of the operating system during program execution.

Contiguous Allocation Technique

- There are two popular techniques used for contiguous memory allocation

  - **Fixed Partitioning**
  - **Variable Partitioning**

Fixed Partitioning

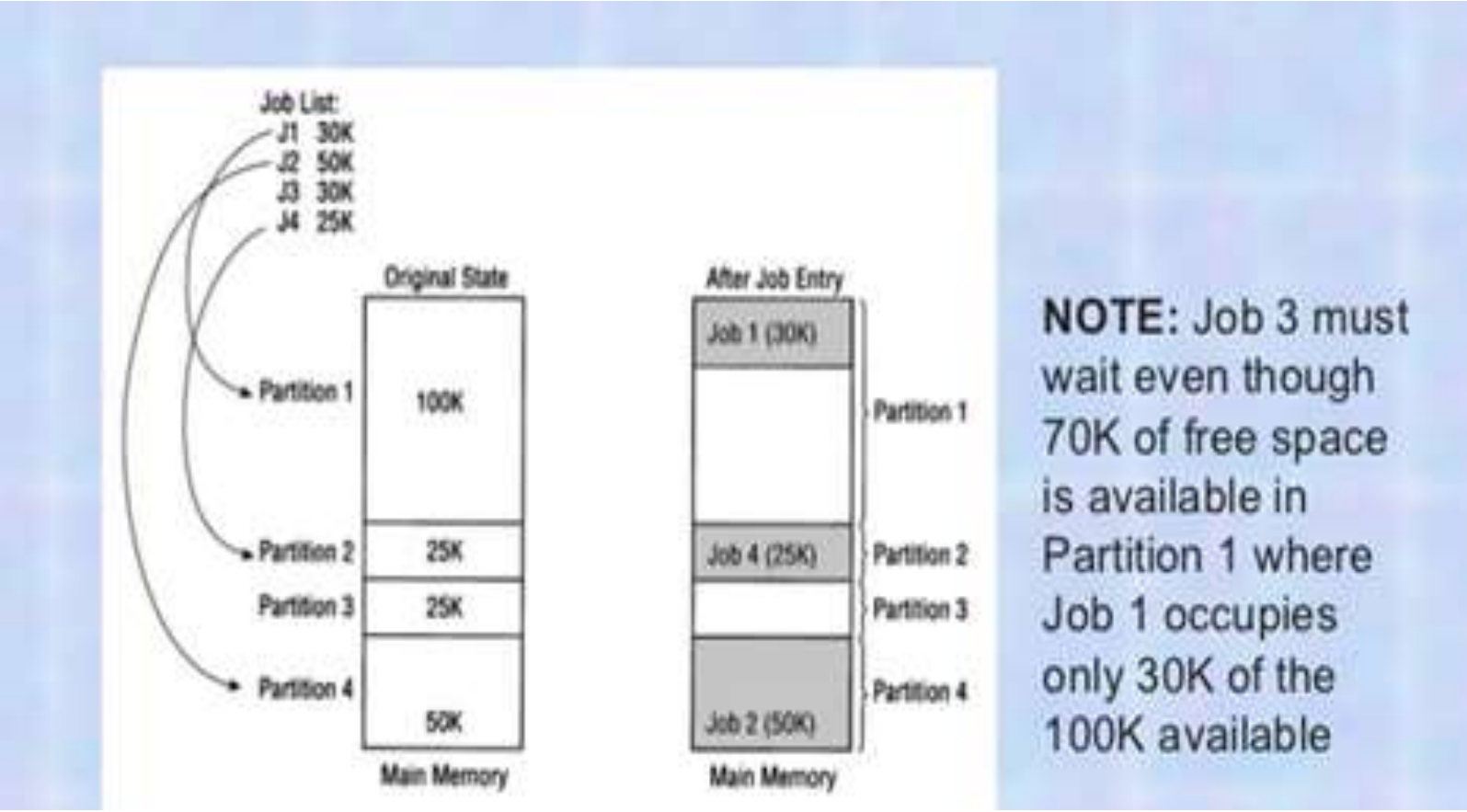
- Static partitioning is a fixed partitioning scheme.
- In this partitioning, **number of partitions** (non-overlapping) in RAM are **fixed** but **size of each partition may or may not be same.**
- As it is **contiguous** allocation, hence no spanning is allowed.
- Here partition are made before execution or during system configure.
- Each partition is allowed to store only one process.
- These partitions are allocated to the processes as they arrive.

Fixed Partitioning (Cont.)

- 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.

Fixed Partitioning (Cont.)

Example



Disadvantages of Fixed Partitioning

- **Limitation on the Size of the Process: –** Sometimes, when the size of the process is larger than the maximum partition size, then we cannot load that process into the memory. So, this is the main disadvantage of the fixed partition.
- **Degree of Multiprogramming is Less: –** We can understand from the degree of multiprogramming that it means at the same time, the maximum number of processes we can load into the memory. In Fixed Partitioning, the size of the partition is fixed, and we cannot vary it according to the process size; therefore, in fixed partitioning, the degree of multiprogramming is less and fixed.
- **Internal Fragmentation**

Let us first discuss about what is fragmentation and then discuss about Internal Fragmentation

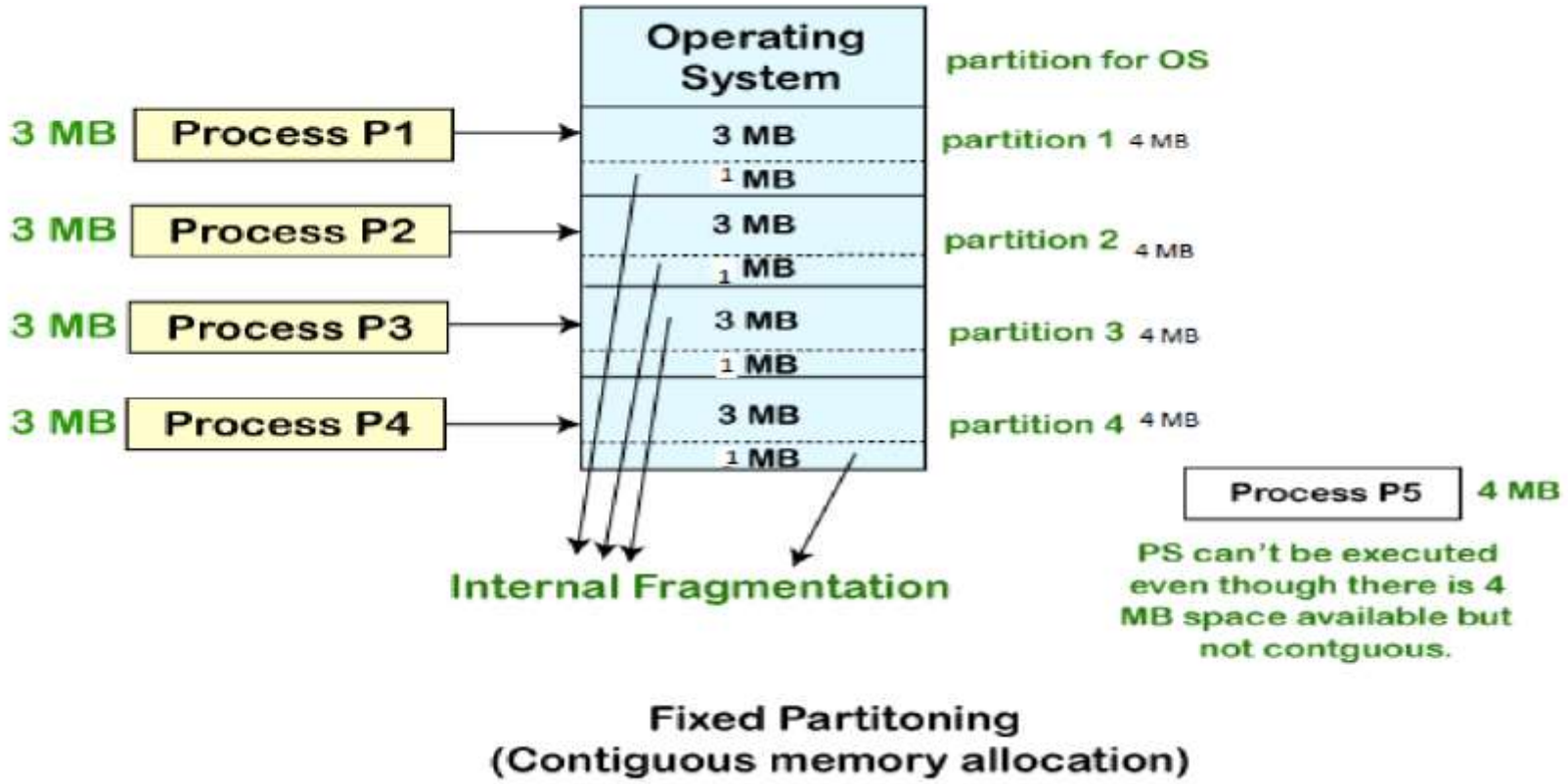
Fragmentation

- Fragmentation is an unwanted problem where the memory blocks cannot be allocated to the processes due to their small size and the blocks remain unused.
- It can also be understood as when the processes are loaded and removed from the memory they create free space or hole in the memory and these small blocks cannot be allocated to new upcoming processes and results in inefficient use of memory.
- Basically, there are two types of fragmentation:
  - Internal Fragmentation
  - External Fragmentation

Internal Fragmentation

- It occurs when the space is left inside the partition after allocating the partition to a process.
- This space is called as internally fragmented space.
- This space can not be allocated to any other process.
- This is because only Fixed (static) partitioning allows to store only one process in each partition.
- Internal Fragmentation occurs only in Fixed (static) partitioning .

Internal Fragmentation (Cont.)



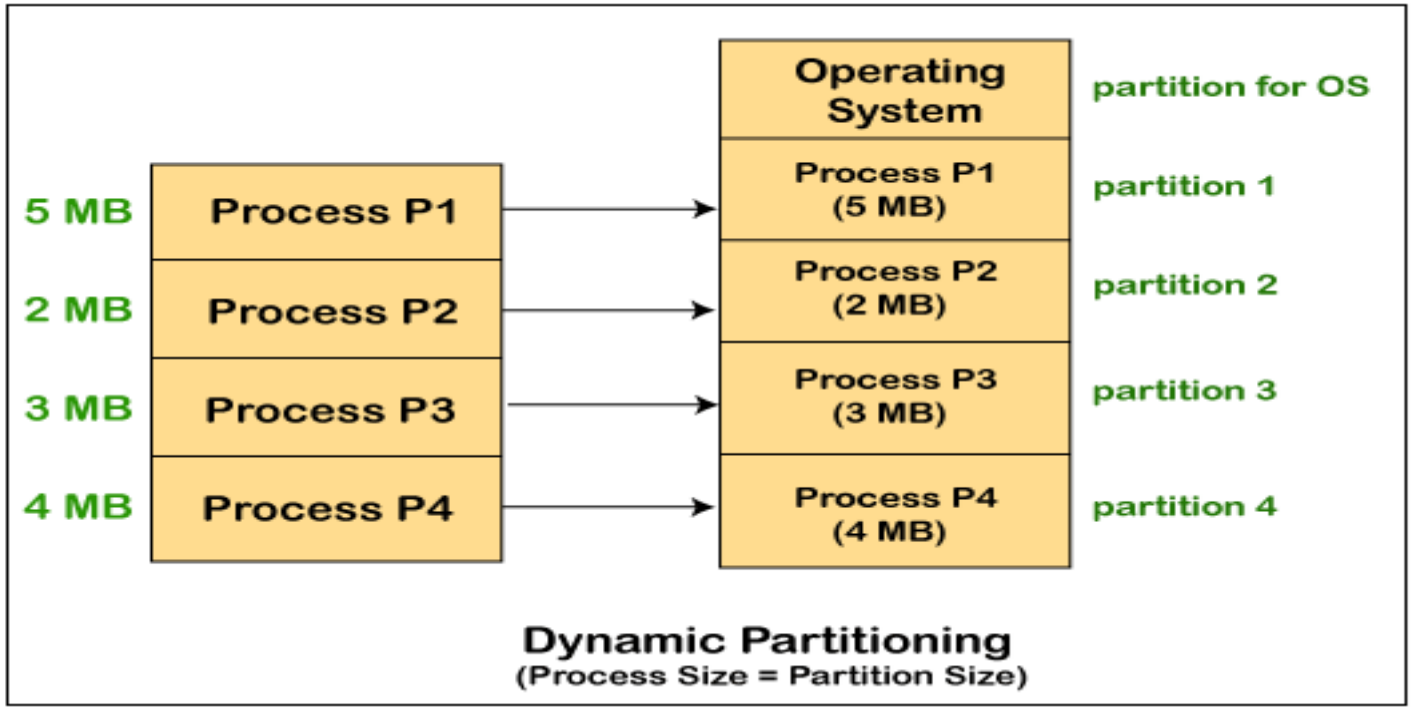
Solution to Internal Fragmentation

- This problem is occurring because we have fixed the sizes of the memory blocks. This problem can be removed if we use dynamic partitioning for allocating space to the process.
- In dynamic partitioning, the process is allocated only that much amount of space which is required by the process. So, there is no internal fragmentation.

Variable - Partitioning

- 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, memory contains a set of holes of various sizes.
- As processes enter the system, they are put into an input queue.

Variable – Partitioning (Cont.)



Variable – Partitioning (Cont.)

- The operating system takes into account the memory requirements of each process and the amount of available memory space in determining which processes are allocated memory.
- When a process is allocated space, it is loaded into memory, and it can then compete for CPU time.
- When a process terminates, it releases its memory, which the operating system may then fill with another process from the input queue.

Variable – Partitioning (Cont.)

- 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. Memory is allocated to processes until, finally, the memory requirements of the next process cannot be satisfied — that is, no available block of memory (or hole) is large enough to hold that process.

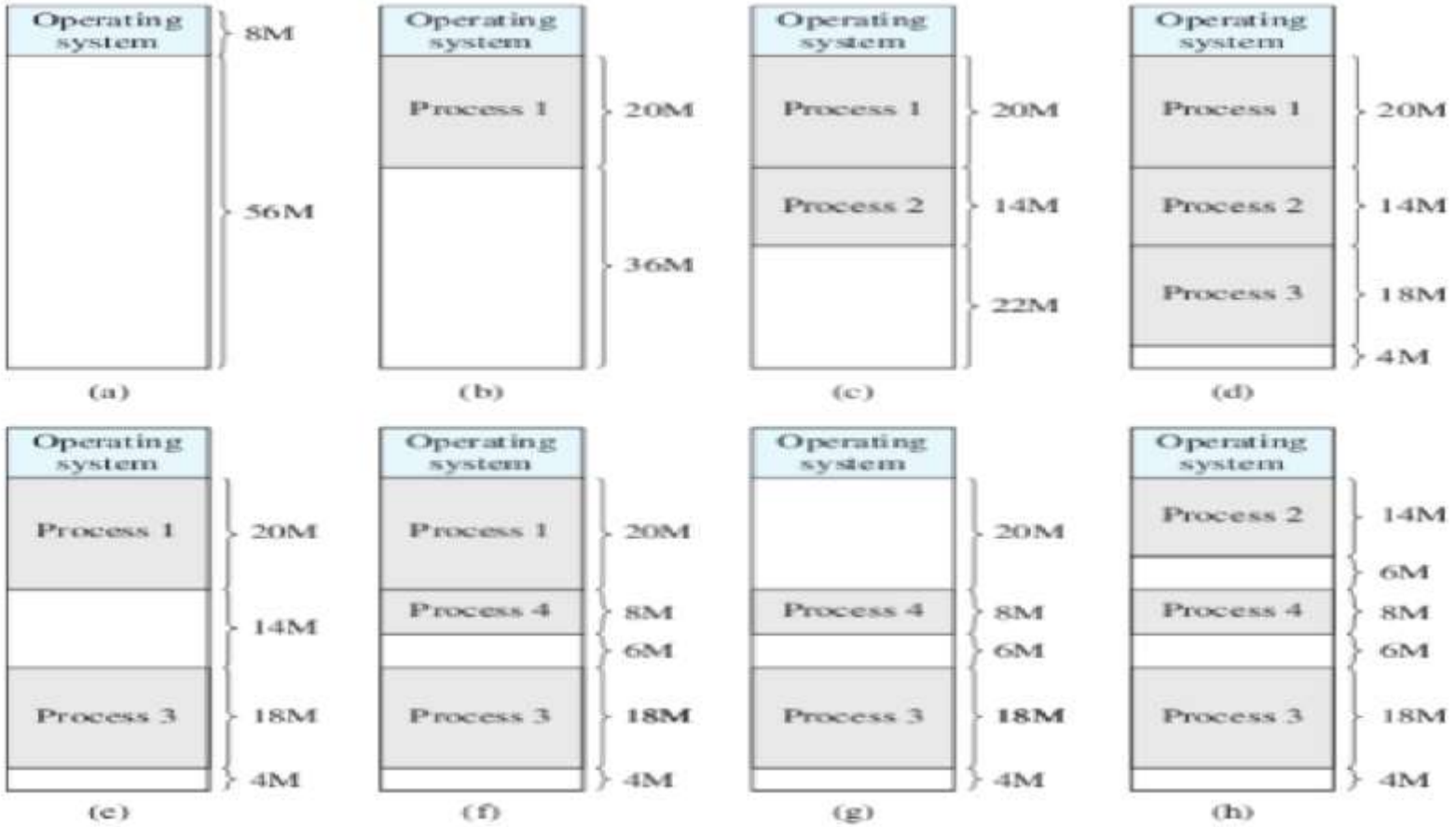
Variable – Partitioning (Cont.)

- The operating system can then 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.
- In general, as mentioned, 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.

Variable – Partitioning (Cont.)

- 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.
- 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.

Variable – Partitioning (Cont.)

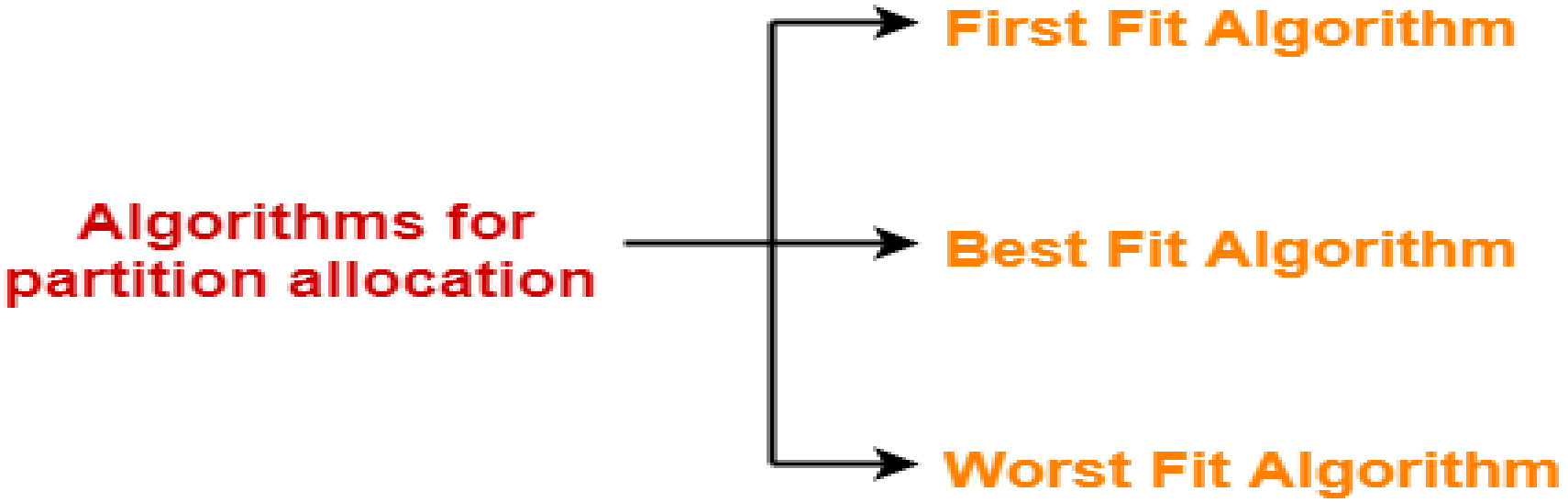


Variable – Partitioning (Cont.)

- This procedure is a particular instance of the general dynamic storage-allocation problem, which concerns how to satisfy a request of size n from a list of free holes.

- There are many solutions to this problem. The first-fit, best-fit, and worst-fit strategies are the ones most commonly used to select a free hole from the set of available holes.

Partition Allocation Algorithm

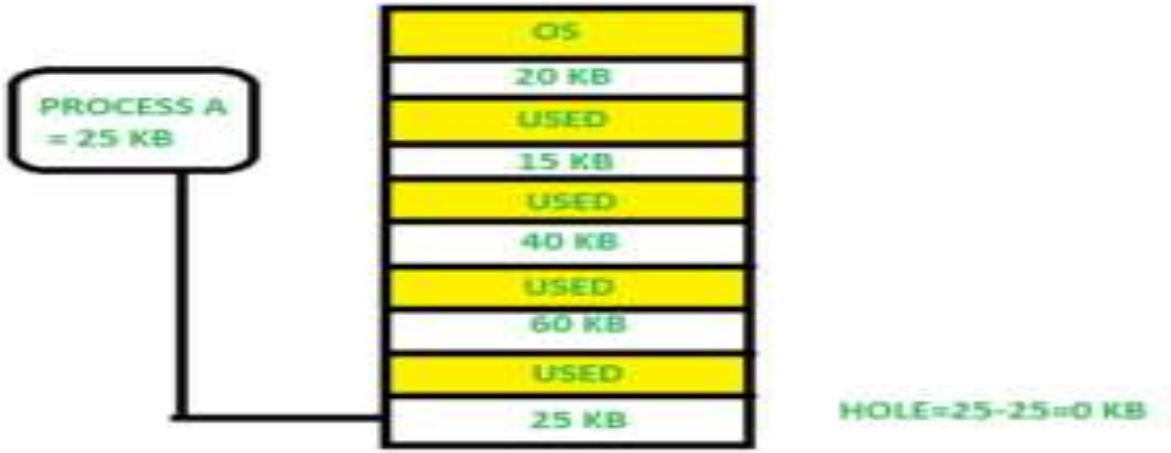


First Fit

- 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.

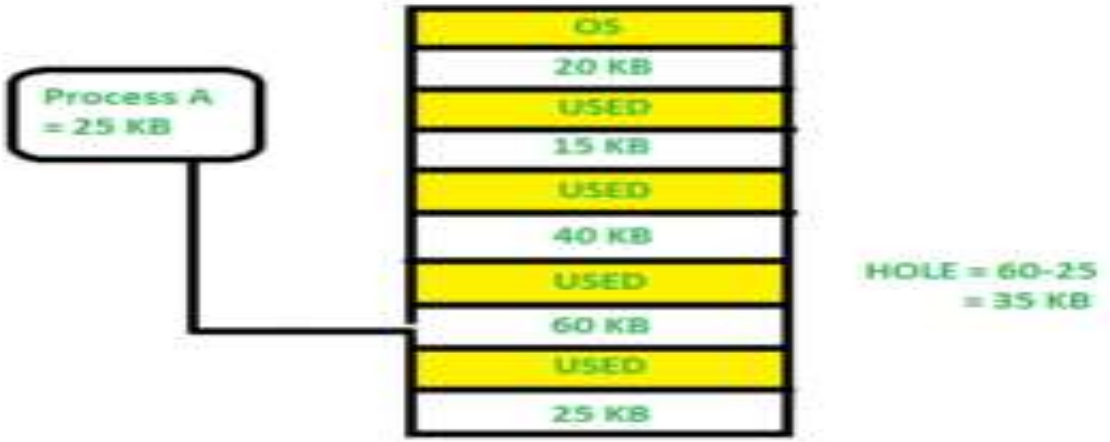
Best Fit

- Allocate the smallest hole that is big enough.
- We must search the entire list, unless the list is ordered by size.
- This strategy produces the smallest leftover hole.



Worst Fit

- Allocate the largest hole.
- Again, we must search the entire list, unless it is sorted by size.
- This strategy produces the largest leftover hole which may be useful for the other process



Problem to Solve

Question

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

Steps to Solve – First Fit

Memory partitions: 100Kb, 500Kb, 200Kb, 300Kb, 600Kb

Processes: 212 Kb, 417 Kb, 112 Kb, and 426 Kb

First Fit

212Kb is put in 500Kb partition

417Kb is put in 600Kb partition

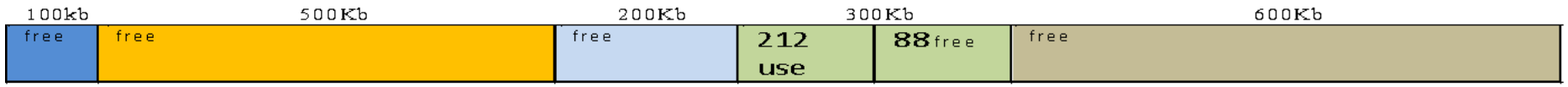
112Kb is put in 288Kb partition (new partition 288Kb = 500Kb - 212Kb)

426Kb must wait

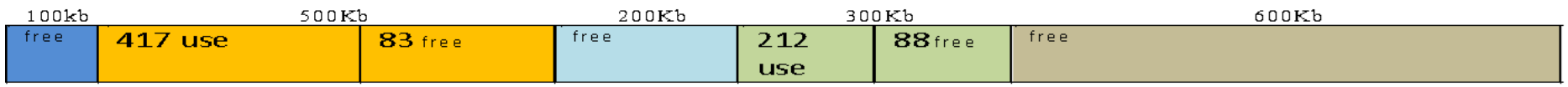
Steps to Solve – Best Fit

Best Fit

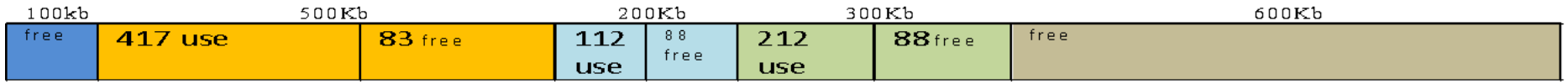
212Kb is put in 300Kb partition



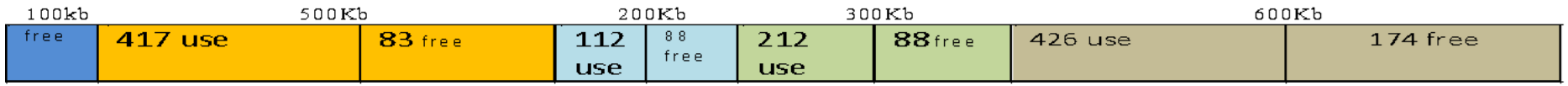
417Kb is put in 500Kb partition



112Kb is put in 200Kb partition



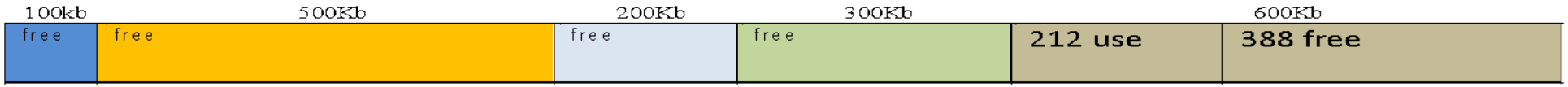
426Kb is put in 600Kb partition



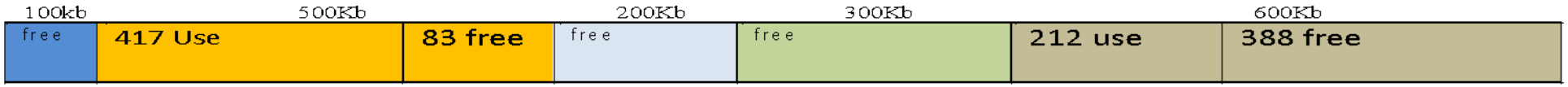
Steps to Solve – Worst Fit

Worst Fit

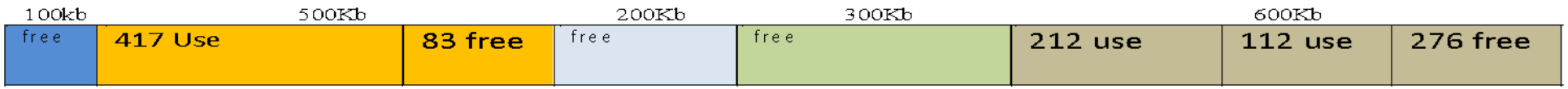
212Kb is put in 600Kb partition



417Kb is put in 500Kb partition



112Kb is put in 388Kb partition



426Kb must wait

Disadvantages of Variable - Partitioning

- Complex Memory Allocation
- External Fragmentation

Complex Memory Allocation

- The task of allocation and deallocation is tough because the size of the partition varies whenever the partition is allocated to the new process. The operating system has to keep track of each of the partitions.
- So, due to the difficulty of allocation and deallocation in the dynamic memory allocation, and every time we have to change the size of the partition; therefore, it is tough for the operating system to handle everything.

External Fragmentation

- It occurs when the total amount of empty space required to store the process is available in the main memory.

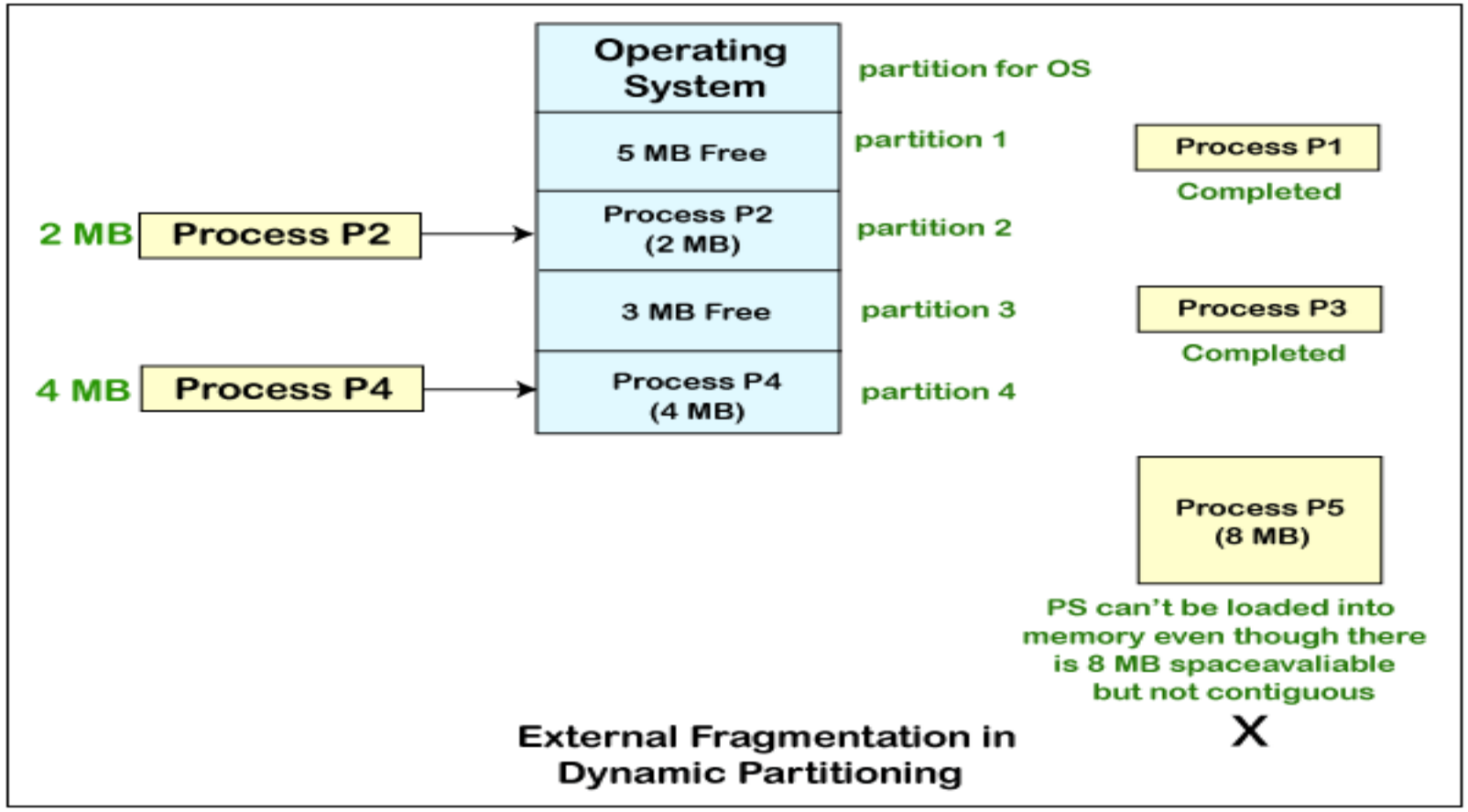
- But because the space is not contiguous, so the process can not be stored.

External Fragmentation (Cont.)

- Consider, three processes P1 (1MB), P2 (3MB), and P3 (1MB), and we want to load the processes in the various partitions of the main memory.
- Now the processes P1 and P3 are completed, and space that is assigned to the process P1 and P3 is freed. Now we have partitions of 1 MB each, which are unused and present in the main memory, but we cannot use this space to load the process of 2 MB in the memory because space is not contiguous.
- The rule says that we can load the process into the memory only when it is contiguously residing in the main memory. So, if we want to avoid external fragmentation, then we have to change this rule.

External Fragmentation (Cont.)

- Example

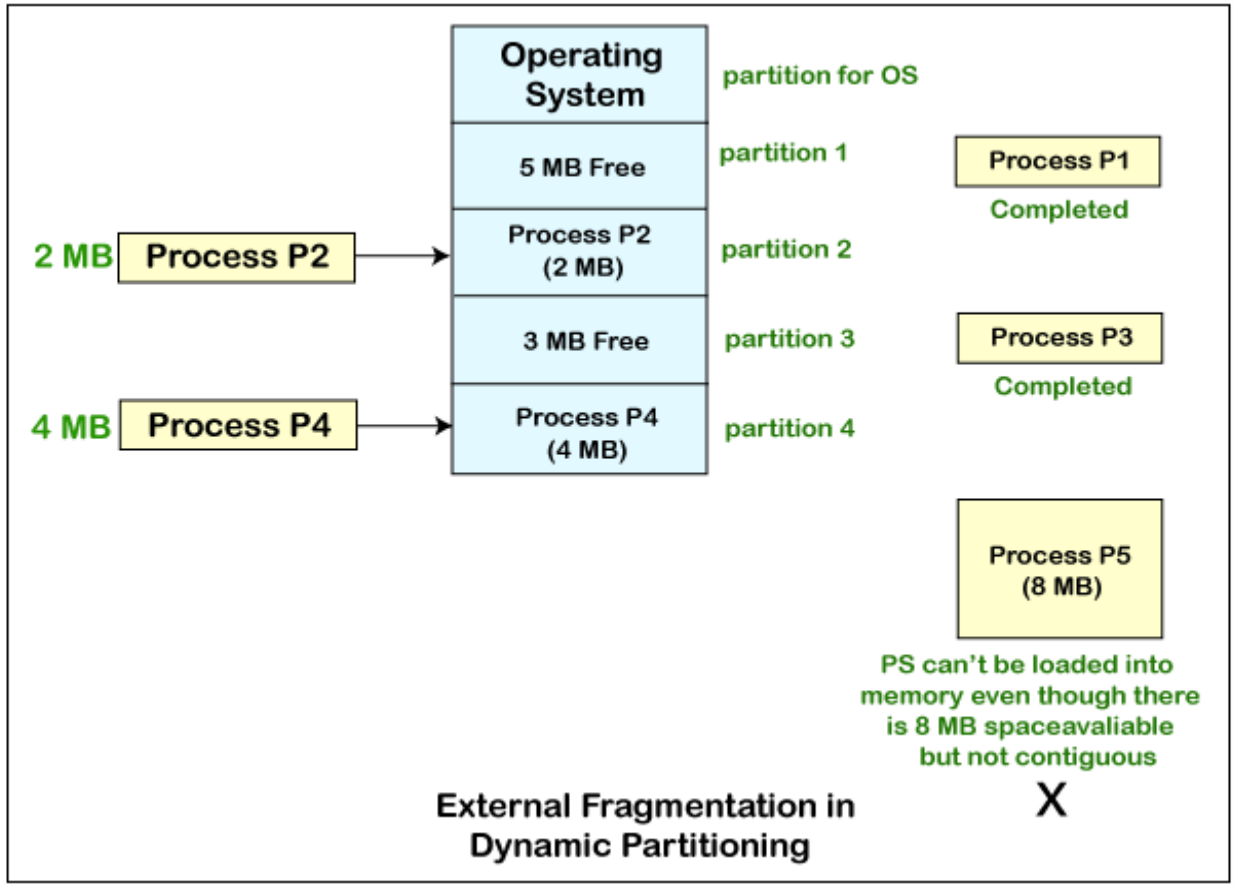


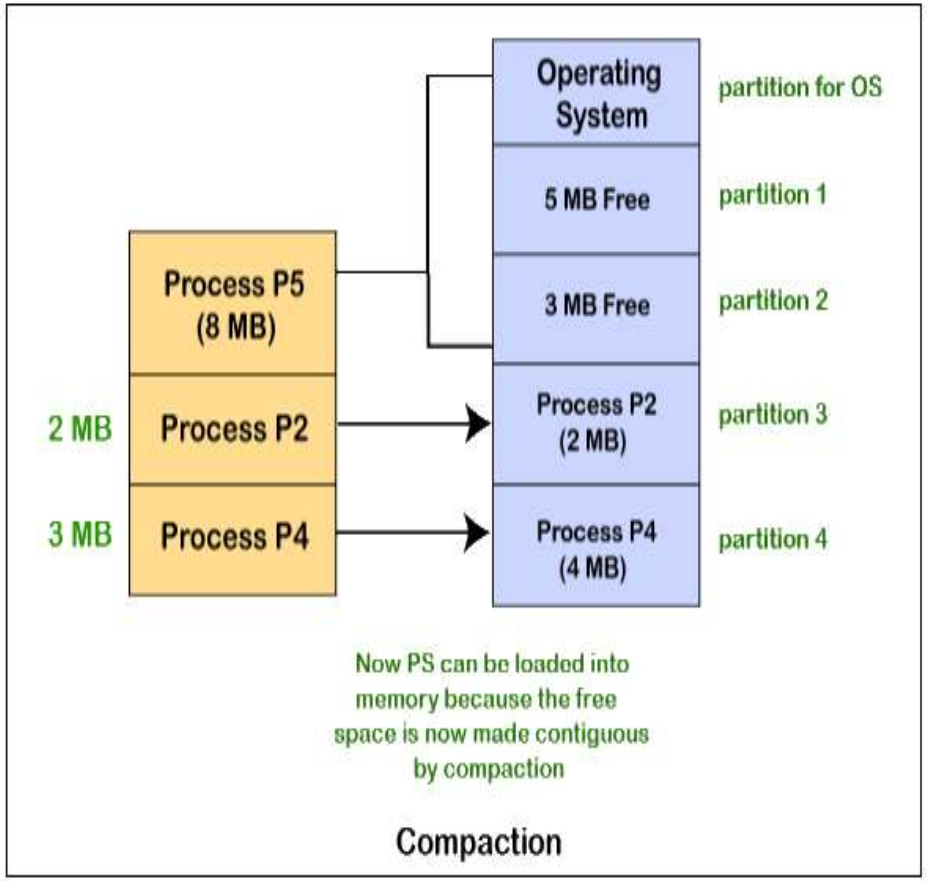
Solution to External Fragmentation

- This problem is occurring because **we are allocating memory continuously** to the processes. So, if we remove this condition external fragmentation can be reduced.

- One of the way to remove external fragmentation is **compaction**. When dynamic partitioning is used for memory allocation then external fragmentation can be reduced by **merging all the free memory** together in one large block.

Solution to External Fragmentation (Cont.)





Solution to External Fragmentation (Cont.)

- This technique is also called **defragmentation.** This larger block of memory is then used for allocating space according to the needs of the new processes.

Problem with Compaction

- Due to compaction, the system efficiency is decreased because we need to move all the free spaces from one place to other.
- In this way, the more amount of time is wasted, and the CPU remains ideal all the time. Instead of that, with the help of compaction, we can avoid external fragmentation, but this will make the system inefficient.

Solution to External Fragmentation (Cont.)

- Another way, **paging & segmentation** (non-contiguous memory allocation techniques) where memory is allocated non-contiguously to the processes.

Fixed vs Variable Partitioning

S.NO.	Fixed partitioning	Variable partitioning
1.	In multi-programming with fixed partitioning the main memory is divided into fixed sized partitions.	In multi-programming with variable partitioning the main memory is not divided into fixed sized partitions.
2.	Only one process can be placed in a partition.	In variable partitioning, the process is allocated a chunk of free memory.
3.	It does not utilize the main memory effectively.	It utilizes the main memory effectively.
4.	There is presence of internal fragmentation and external fragmentation.	There is external fragmentation.
5.	Degree of multi-programming is less.	Degree of multi-programming is higher.
6.	It is more easier to implement.	It is less easier to implement.
7.	There is limitation on size of process.	There is no limitation on size of process.

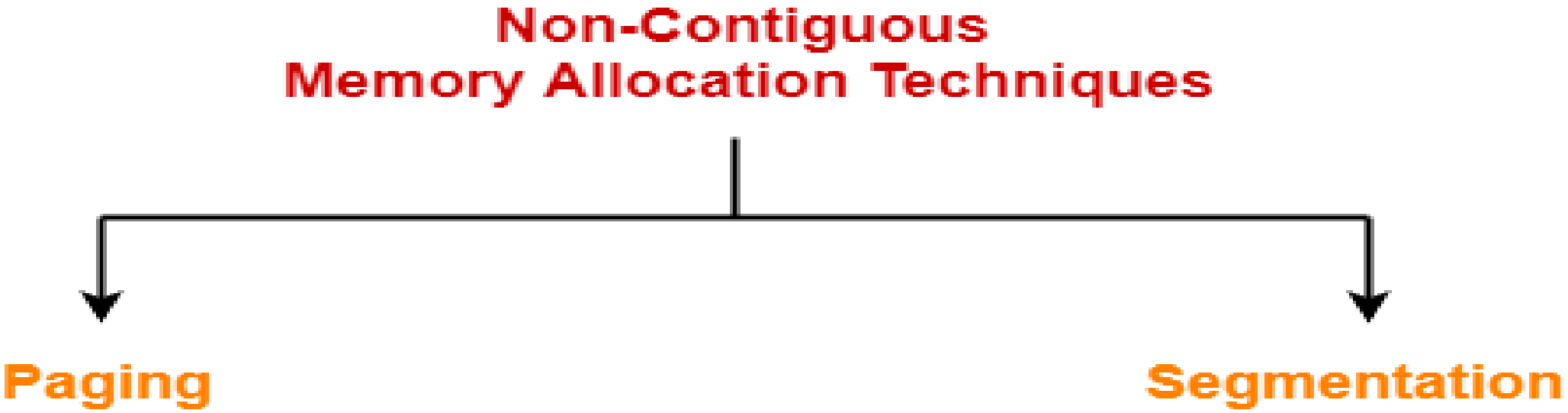
Internal vs External Fragmentation

INTERNAL FRAGMENTATION	EXTERNAL FRAGMENTATION
Memory allocated to a process may be slightly larger than the requested memory. The difference between these two numbers is internal fragmentation.	External fragmentation exists when there is enough total memory space to satisfy a request but available spaces are not contiguous.
First-fit and best-fit memory allocation does not suffer from internal fragmentation.	First-fit and best-fit memory allocation suffers from external fragmentation.
Systems with fixed-sized allocation units, such as the single partitions scheme and paging suffer from internal fragmentation.	Systems with variable-sized allocation units, such as the multiple partitions scheme and segmentation suffer from external fragmentation.

Non Contiguous Allocation

- Non-contiguous memory allocation is a memory allocation technique.
- It allows to store parts of a single process in a non-contiguous fashion.
- Thus, different parts of the same process can be stored at different places in the main memory.

Techniques


Non-Contiguous
Memory Allocation Techniques
Paging
Segmentation

SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
Deemed to be University u/s 3 of UGC Act, 1956

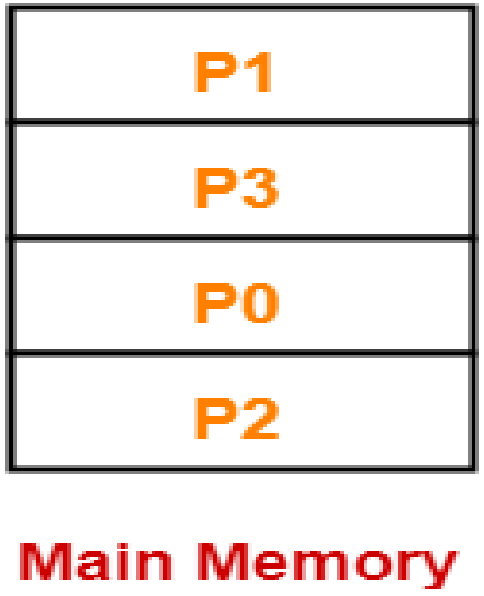
PAGING

Paging

- Paging is a memory management scheme that eliminates the need for contiguous allocation of physical memory.
- Paging permits the physical address space of a process to be non – contiguous.
- The mapping from virtual to physical address is done by the memory management unit (MMU) which is a hardware device and this mapping is known as paging technique.

Example

- Consider a process is divided into 4 pages P_0, P_1, P_2 and P_3.



- In the above example, the process P is divided into 4 sections namely P0, P1, P2, P3.
- All the four sections are mapped in the main memory in a non – contiguous fashion.

Paging

- In the paging mechanism two addresses are involved
- Logical Address

  CPU always generates a logical address.
- Physical Address

  A physical address is needed to access the main memory.

Logical Address and Physical Address

- Logical Address or Virtual Address :

  An address generated by the CPU

- Logical Address Space or Virtual Address Space:

  The set of all logical addresses generated by a program

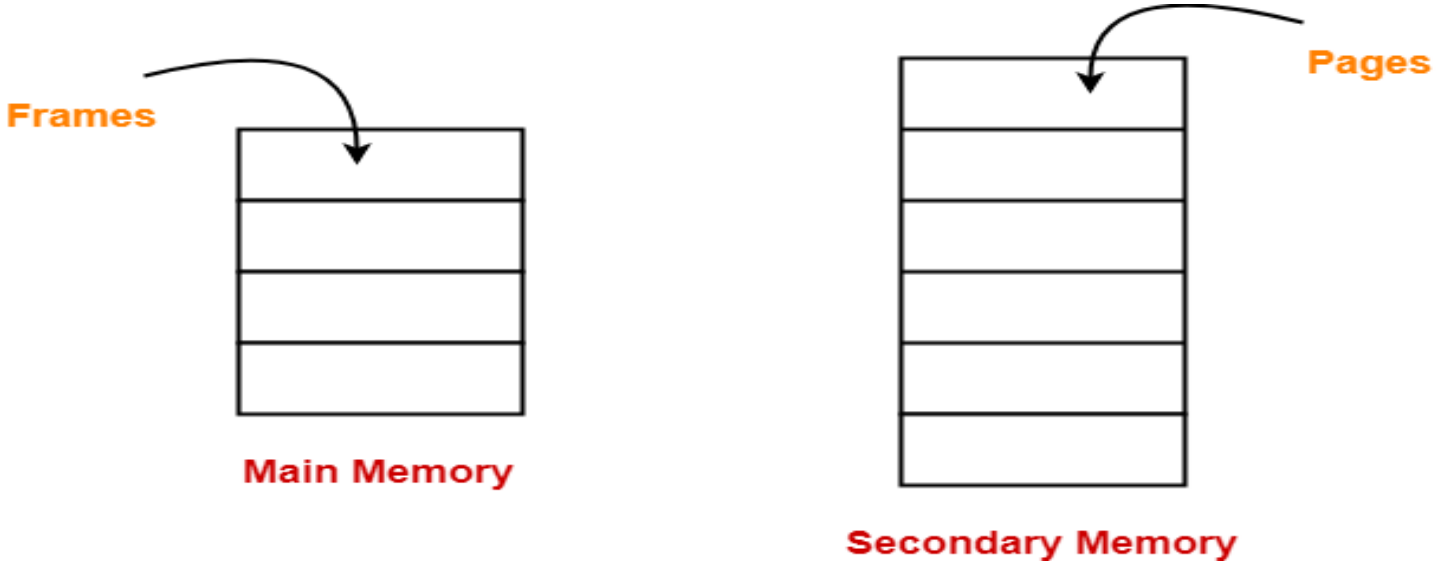
- Physical Address:

  An address actually available on memory unit

- Physical Address Space :

  The set of all physical addresses corresponding to the logical addresses

- The Physical Address Space is conceptually divided into a number of fixed-size blocks, called **frames**.
- The Logical address Space is also divided into fixed-size blocks, called **pages**.
- Page Size = Frame Size



Logical Address

- Address generated by CPU is divided into:
  - **Page number** (***p***) – used as an index into a **page table** which contains base address of each page in physical memory
  - **Page offset** (***d***) – combined with base address to define the physical memory address that is sent to the memory unit
- For given logical addre

Physical Address

- Physical Address is divided into

  Frame number(f): Number of bits required to represent the frame of Physical Address Space or Frame number.

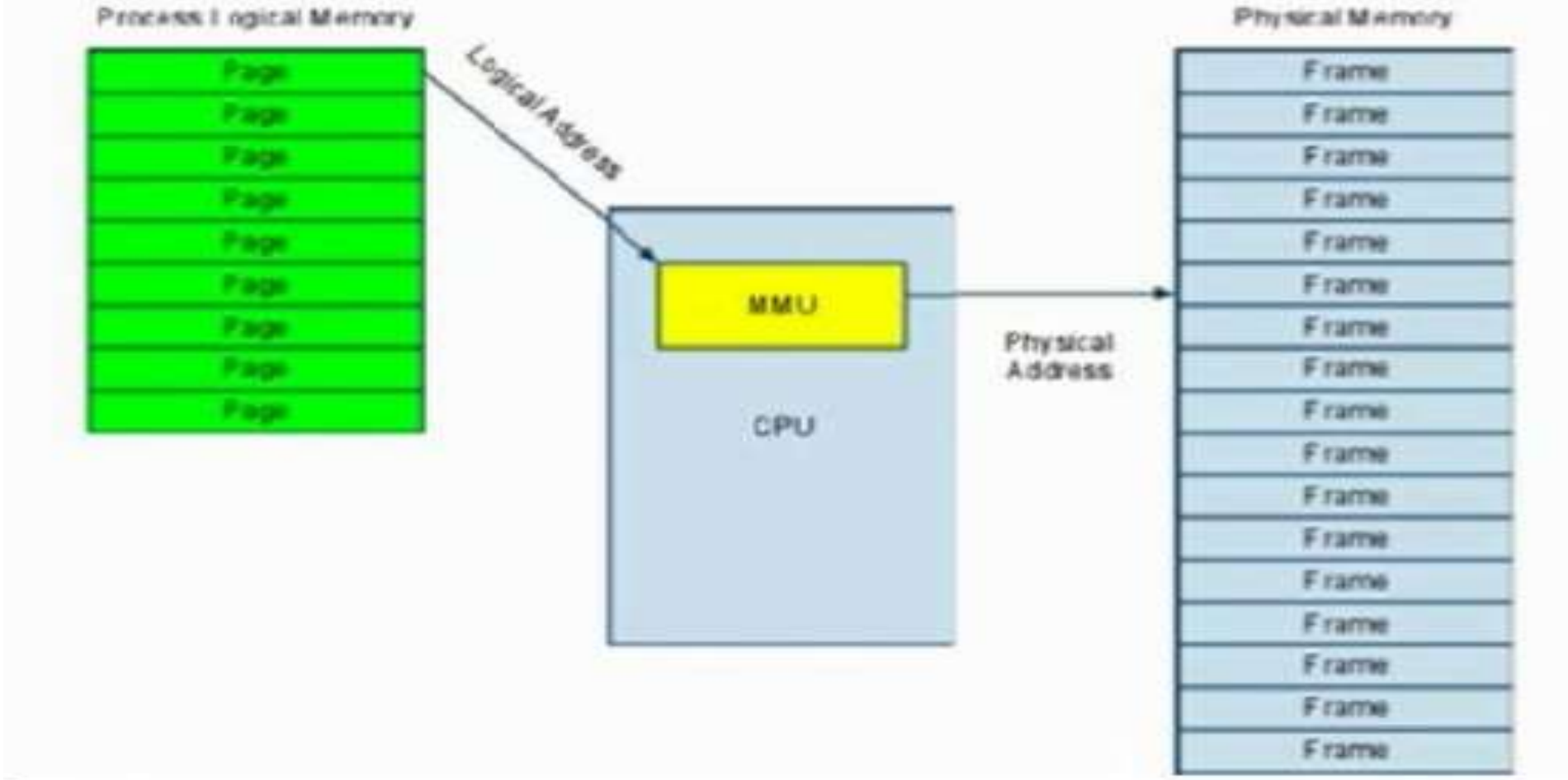
  Frame offset(d): Number of bits required to represent particular word in a frame or frame size of Physical Address Space or word number of a frame or frame offset.

- If Logical Address = 31 bit, then Logical Address Space = 2^{31} words = 2 G words (1 G = 2^{30})
- If Logical Address Space = 128 M words = $2^7 * 2^{20}$ words, then Logical Address = $\log_2 2^{27}$ = 27 bits
- If Physical Address = 22 bit, then Physical Address Space = 2^{22} words = 4 M words (1 M = 2^{20})
- If Physical Address Space = 16 M words = $2^4 * 2^{20}$ words, then Physical Address = $\log_2 2^{24}$ = 24 bits

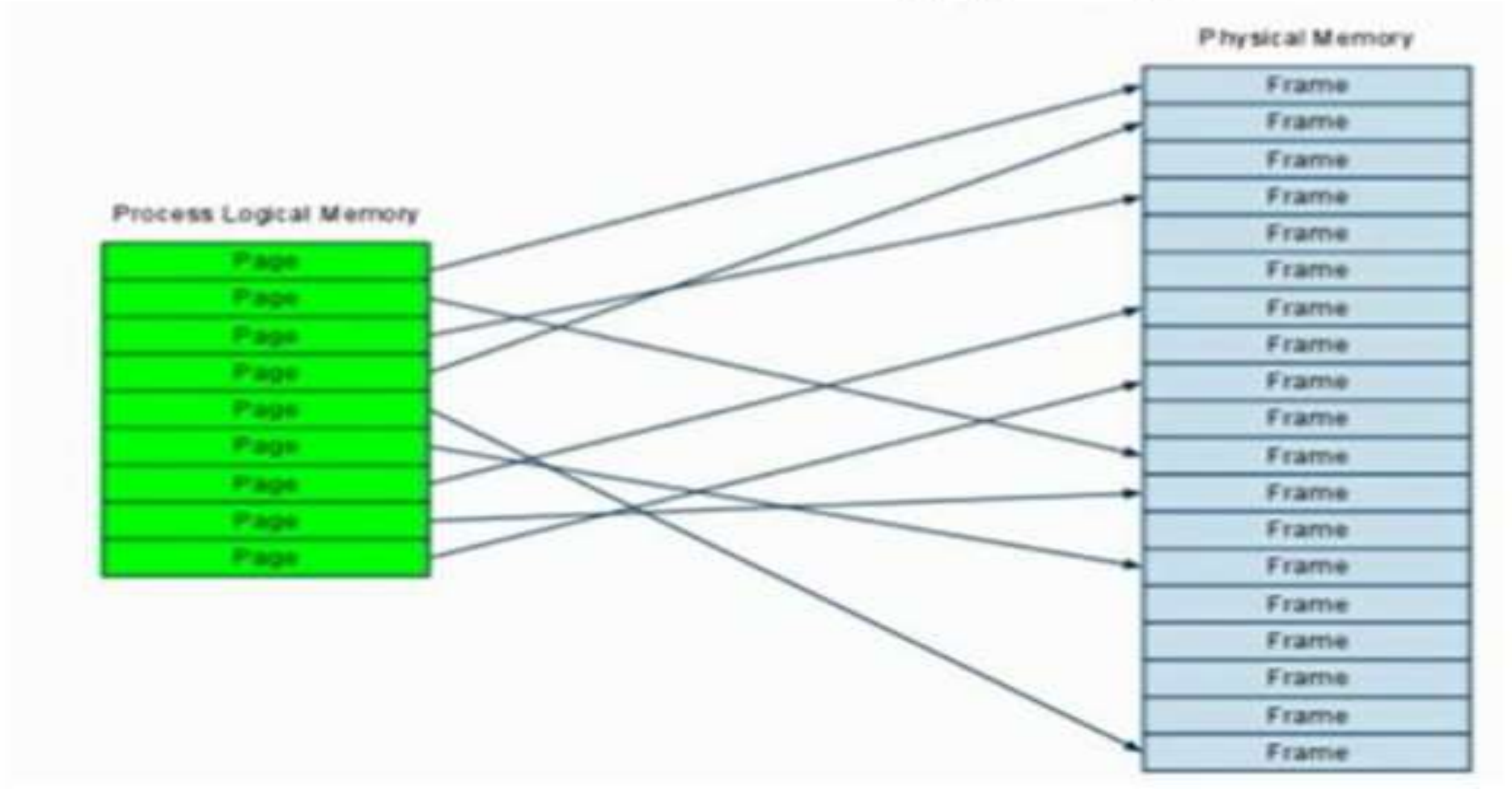
Memory Management

- The Mapping between these two address is done to execute the process using the paging technique.
- The Hardware device that at run time maps virtual address to physical address is done by the Memory Management Unit (MMU)

Page Translation



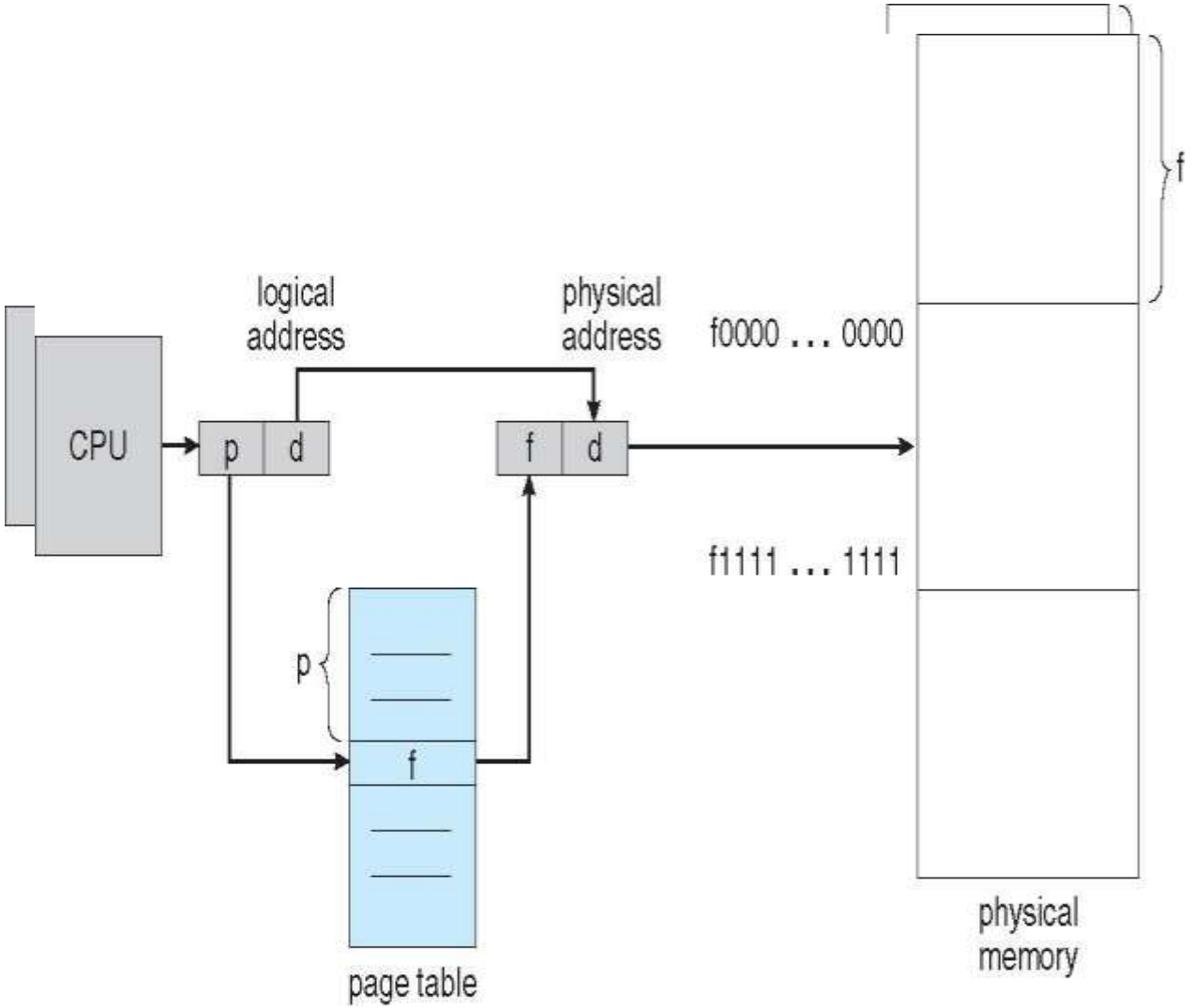
Address Mapping



Paging Hardware

- How does system perform translation? Simplest solution: use a page table.
- Page table is a linear array indexed by virtual page number that gives the physical page frame that contains that page through the look up process.

Paging Hardware



Paging Hardware

LOOK UP PROCESS

Step 1 :Extract page number.

Step 2 : Extract offset.

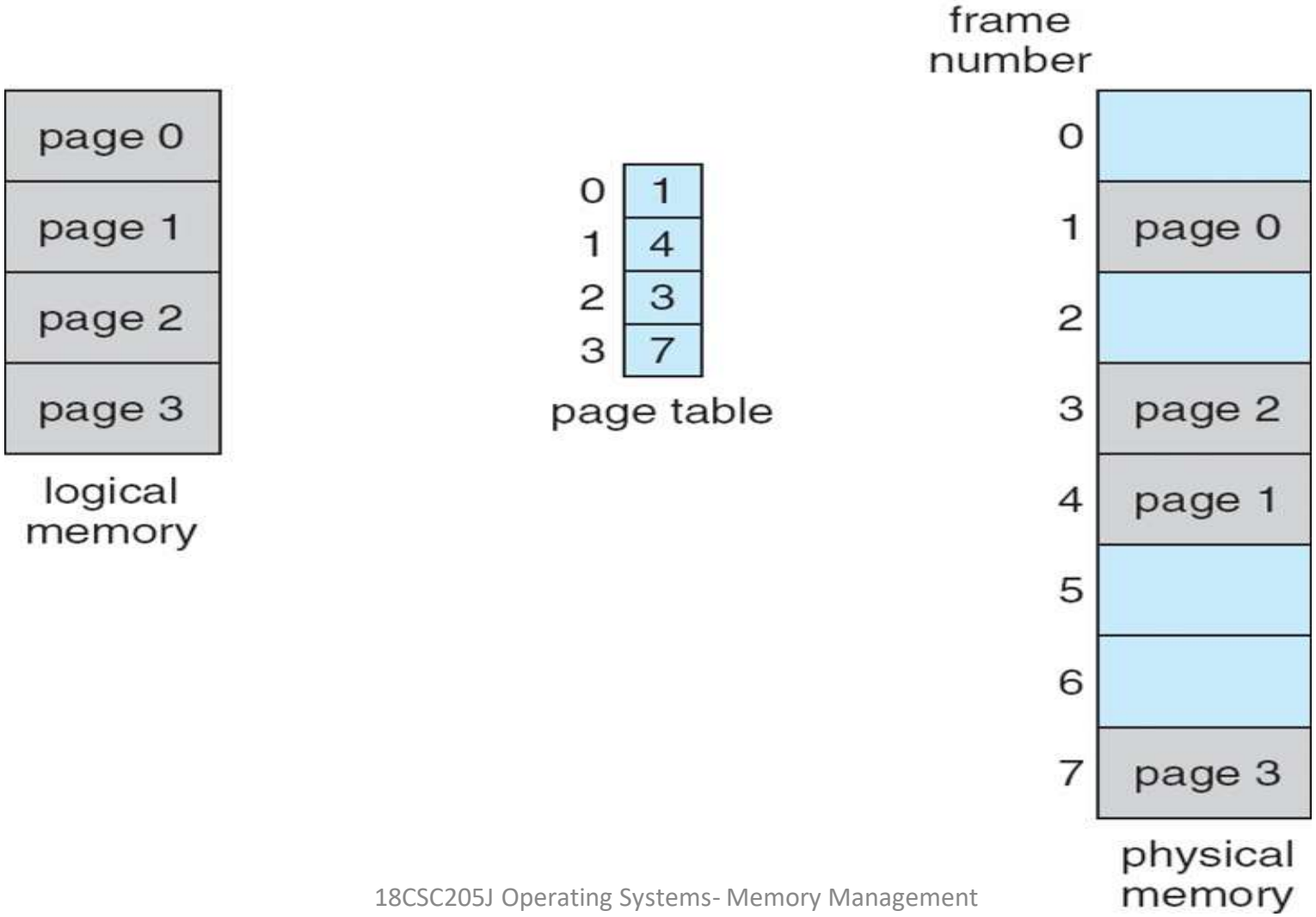
Step 3 : Check that page number is within address space of process.

Step 4 : Look up page number in page table.

Step 5 : Add offset to resulting physical page number

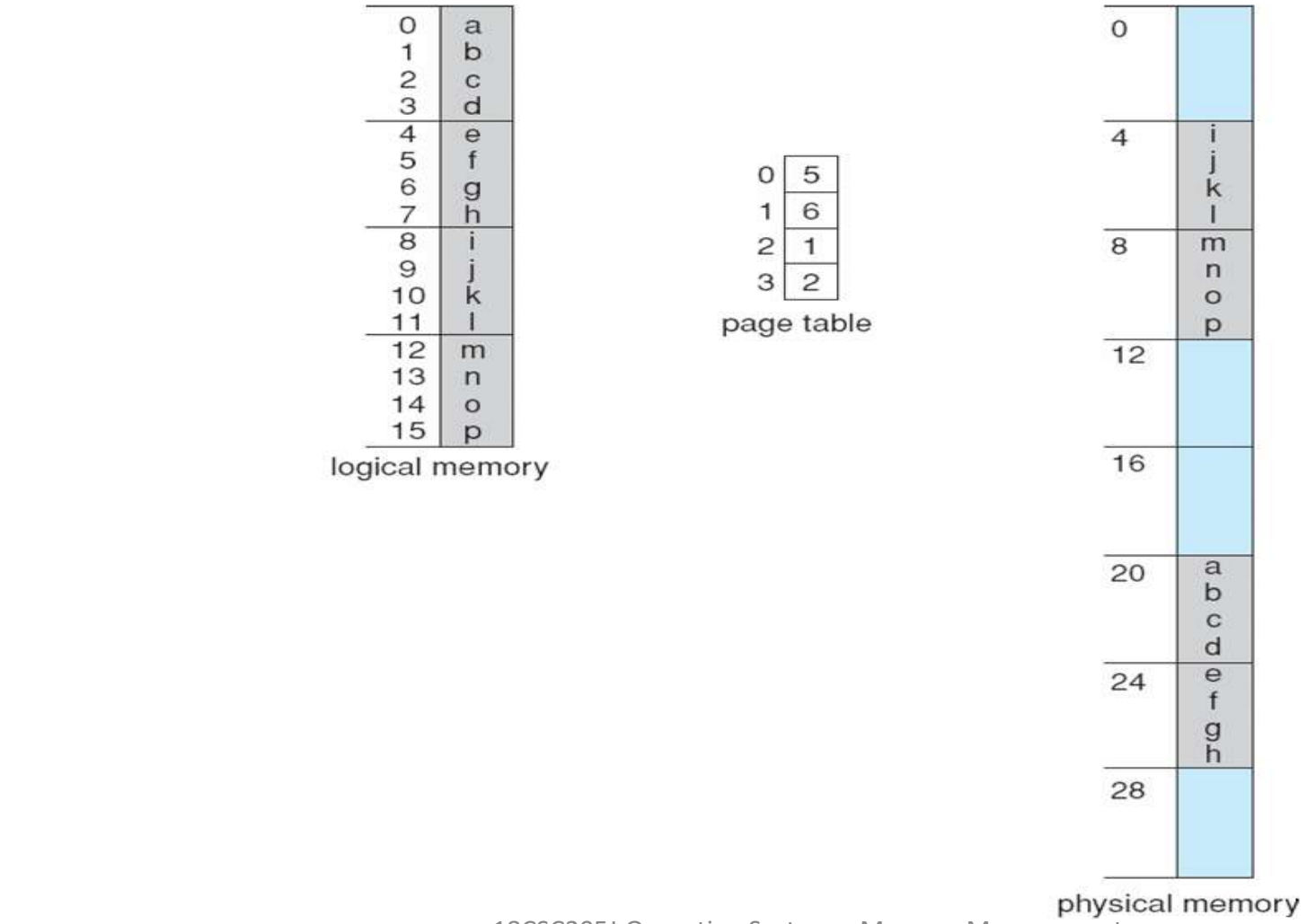
Step 6 : Access memory location.

Paging Model of Logical and Physical Memory



- In the example a process is divided into four pages as page0, page1, page 2 and page3.
- All the four pages are mapped to the frames in the physical memory through the page table.
- The page table maps the page number to its corresponding frame number in the physical memory.
- page0 is mapped to frame1 , page1 is mapped to frame 4, page2 is mapped to frame 3 and page4 is mapped to frame 7.

Paging Example



Free Frame List

Free Frame List

The example

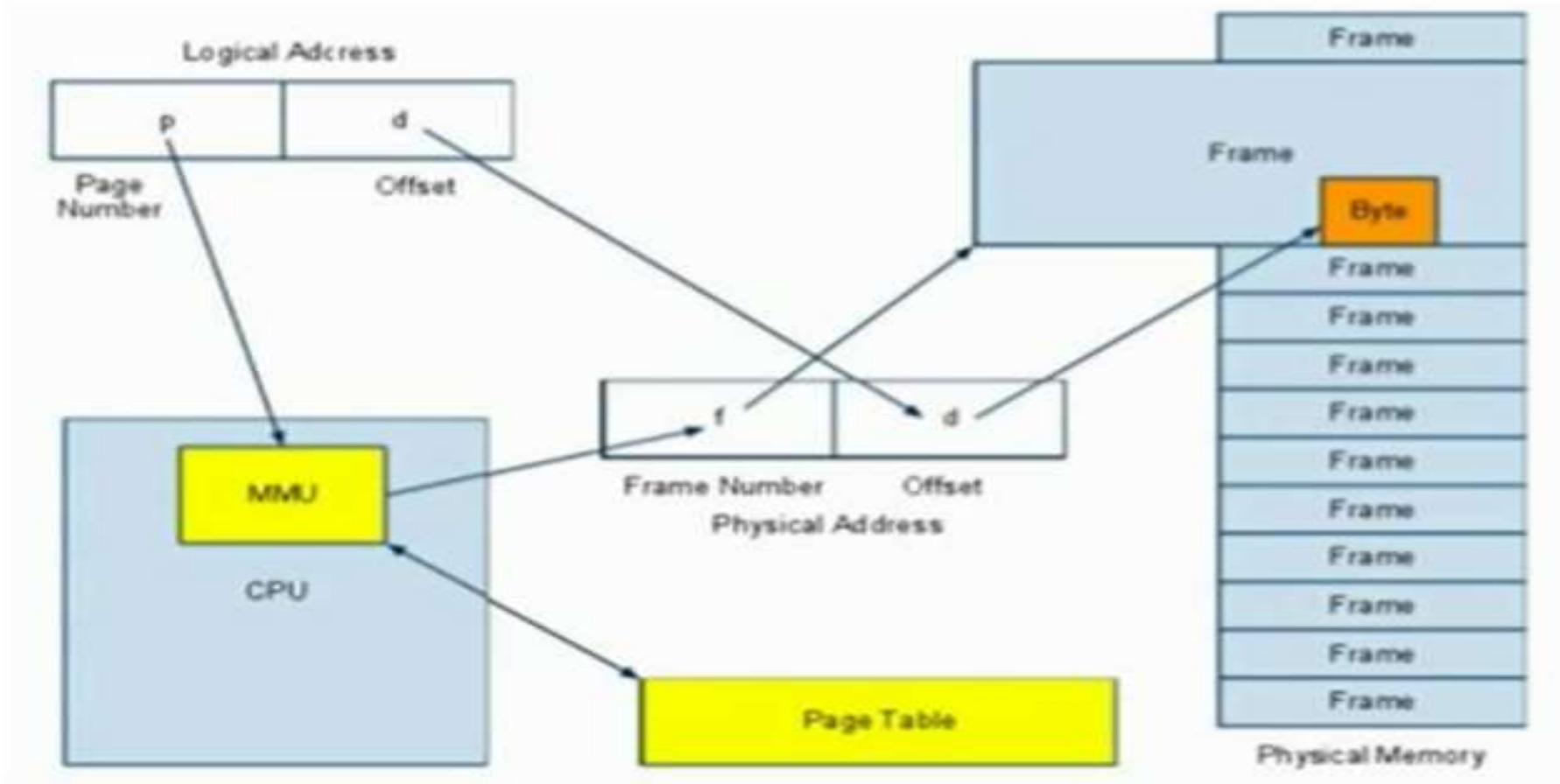
Fig (a) shows a process divided into 4 pages and the list of free frames available in the physical memory.

Fig(b) shows the mapping of these pages to the free frames available and the updated list of free frames available in the physical memory.

Page Table

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

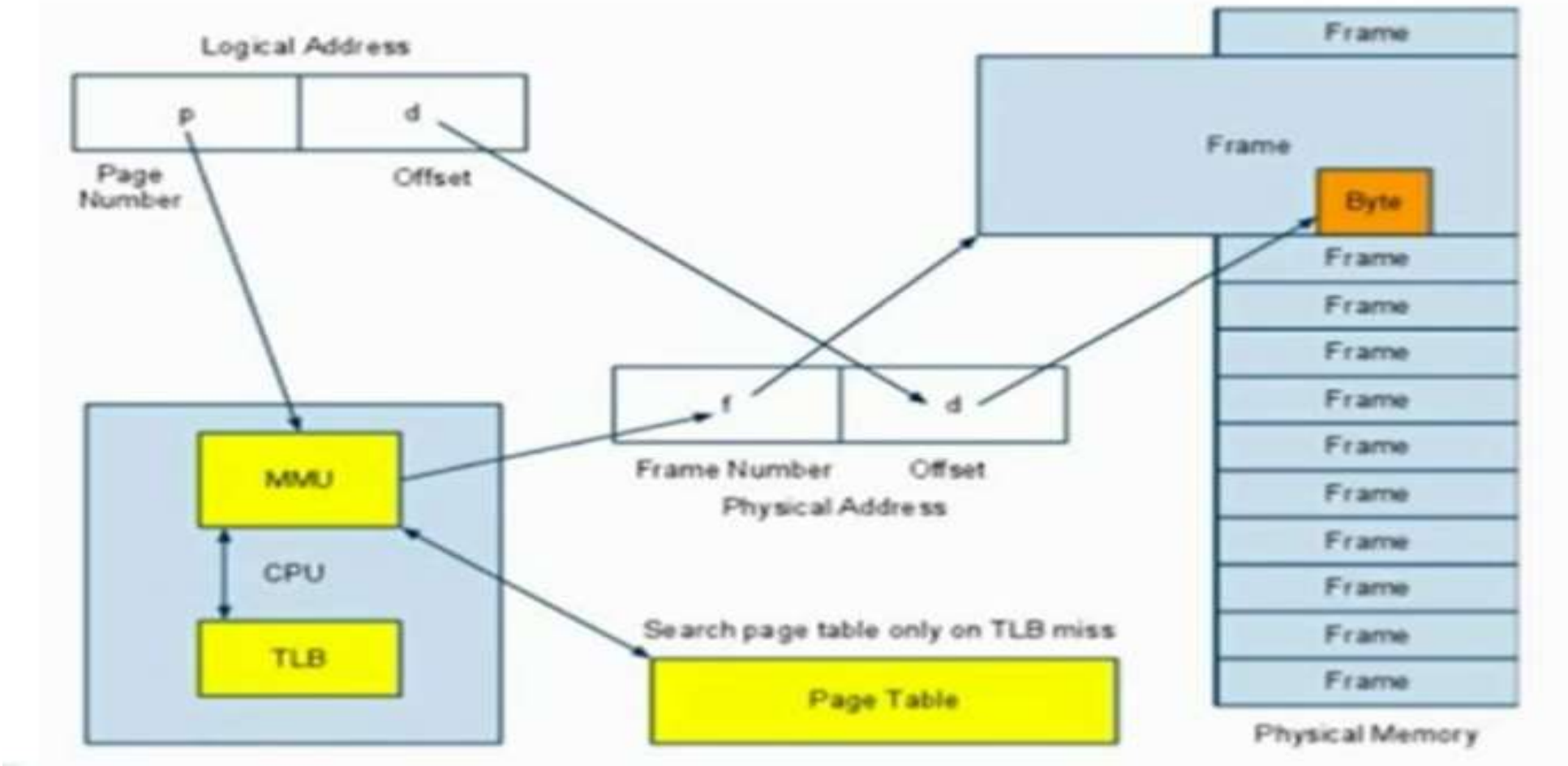
MMU Address Transaction



Problem with paging

- 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)**

TLB–Assisted Transaction



Translation Lookaside Buffer (TLB)

- Speed up the lookup problem with a cache. Store most recent page lookup values in TLB.

Procedure of TLB

Step 1 :Extract page number.

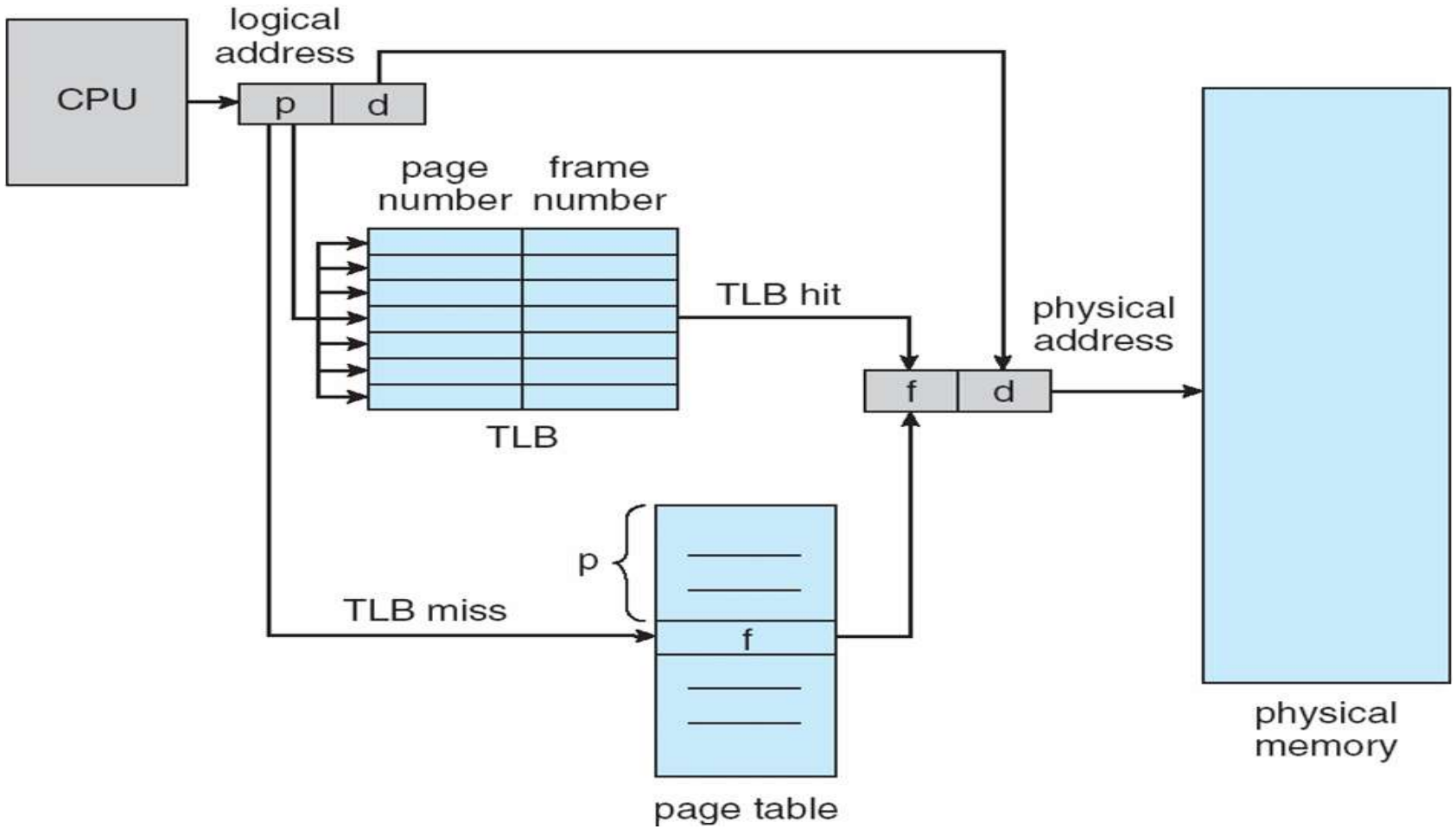
Step 2 : Extract offset.

Step 3 : Look up page number in TLB.

Step 4 : If there, add offset to physical page number and access memory location.

Step 5 : Otherwise, trap to OS. OS performs check, looks up physical page number, and loads translation into TLB. Restarts the instruction.

Paging with TLB



- If the page number is not in the TLB (TLB miss) a memory reference to the page table must be made.
- In addition, we add the page number and frame number into TLB
- If the TLB already full, the OS have to must select one for replacement
- Some TLBs allow entries to be wire down, meaning that they cannot be removed from the TLB, for example kernel codes
- The percentage of times that a particular page number is found in the TLN is called hit ratio

- Fixed size allocation of physical memory in page frames dramatically simplifies allocation algorithm.
- OS can just keep track of free and used pages and allocate free pages when a process needs memory.
- There is no fragmentation of physical memory into smaller and smaller allocatable chunks.

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**)

$$EAT = (1 + \varepsilon)\,\alpha + (2 + \varepsilon)(1 - \alpha)$$

$$= 2 + \varepsilon - \alpha$$

- Consider α = 80%, ε = 20ns for TLB search, 100ns for memory access
  - EAT = 0.80 x 100 + 0.20 x 200 = 120ns

Memory Protection

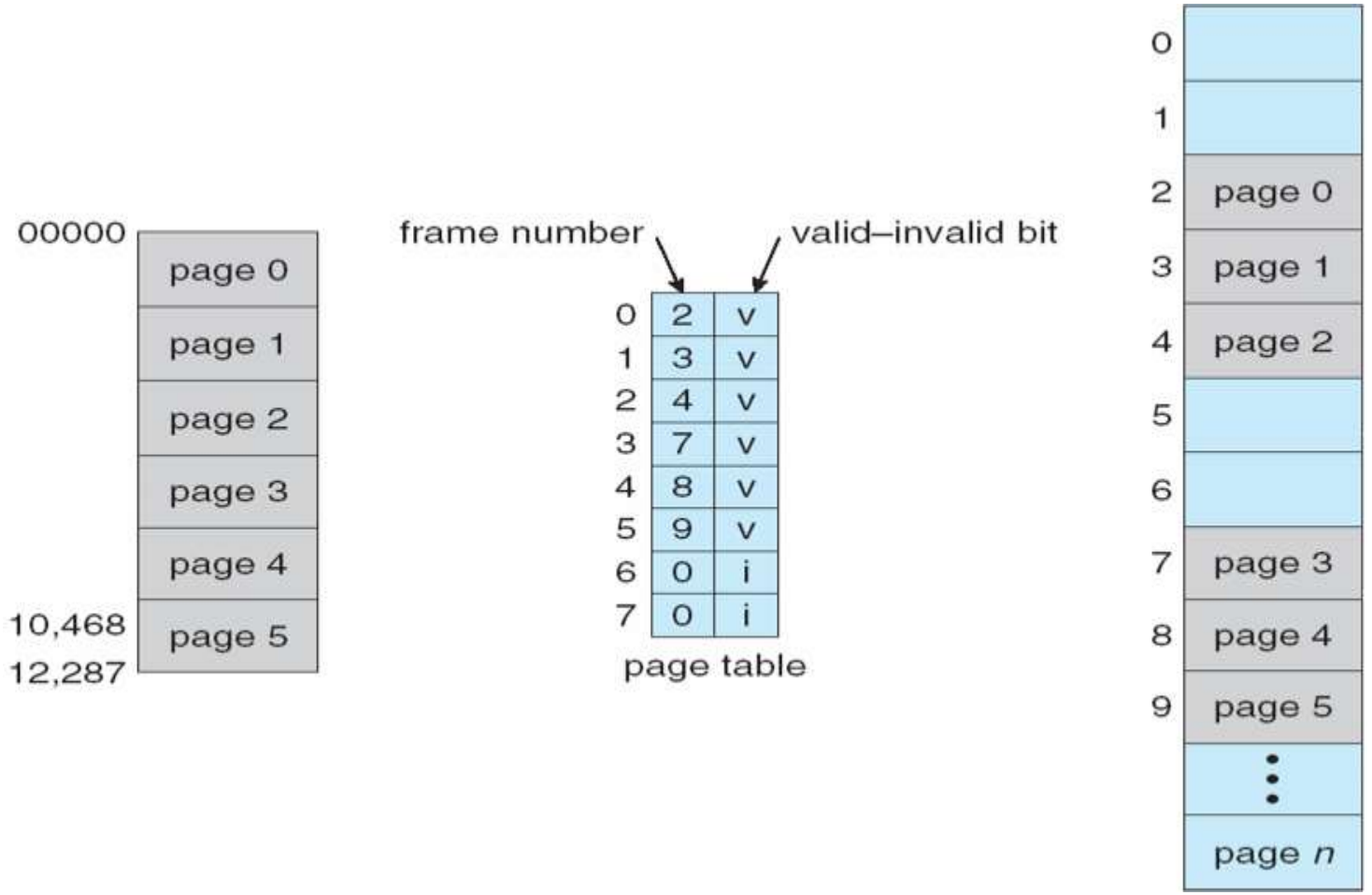
The process has to be protected for the following reasons:

- Preventing one process from reading or writing another process' memory.
- Preventing one process from reading another process' memory.
- Preventing a process from reading or writing some of its own memory.
- Preventing a process from reading some of its own memory.

Memory Protection

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

Valid – Invalid Bit



- Pages that are not loaded into memory are marked as invalid in the page table, using the invalid bit.
- In the above example Page0 to page5 is present in the memory .so , the corresponding valid/Invalid bit is marked as V(Valid).
- The other pages which are not loaded such as page6 and page7 are marked as I(Invalid)

PageFault

- if a page is needed that was not originally loaded up, then a ***page fault trap*** is generated
- Page fault is handled by the following steps:

Step 1: The memory address requested is first checked, to make sure it was a valid memory request.

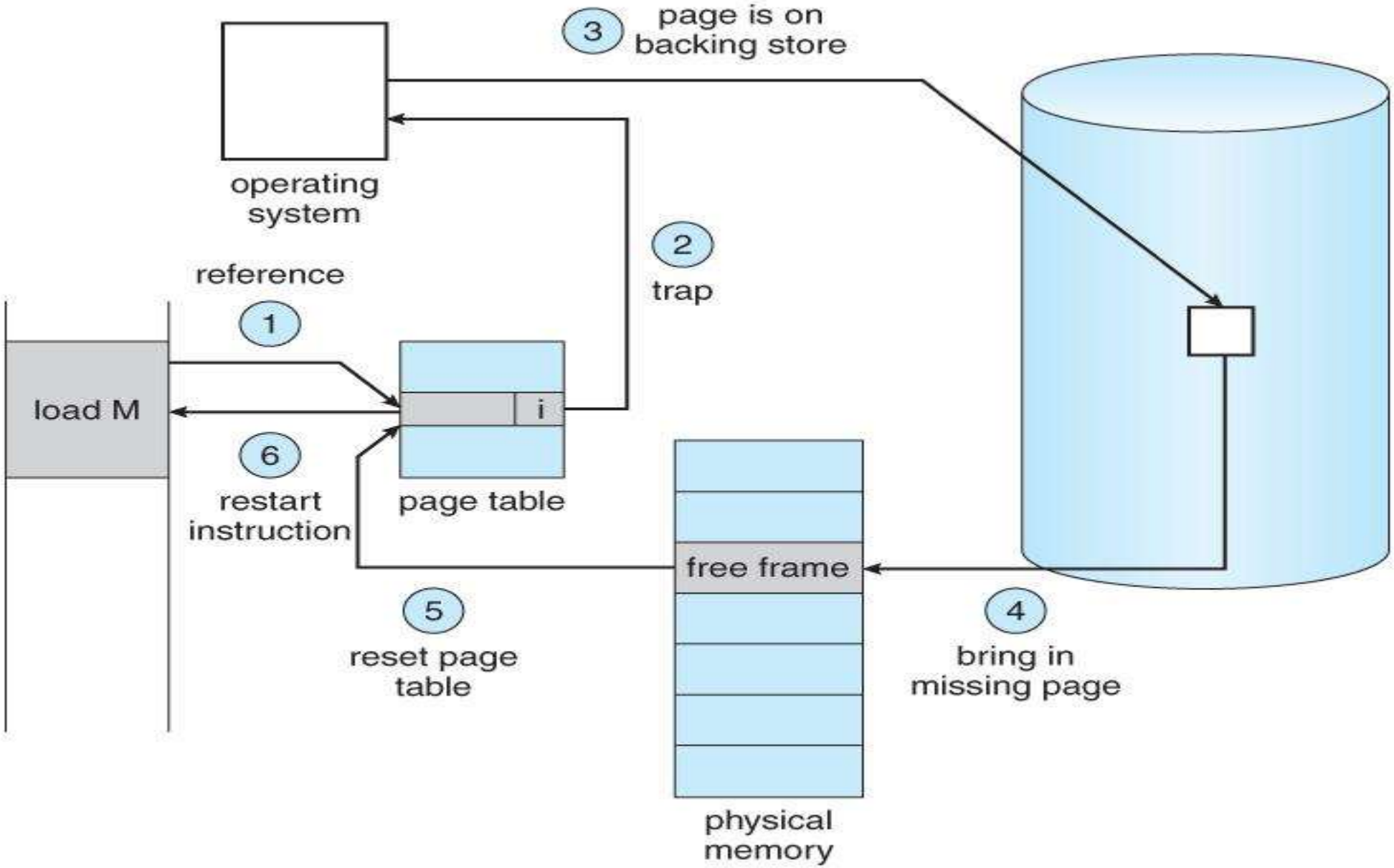
Step 2: If the reference was invalid, the process is terminated. Otherwise, the page must be paged in.

Step 3: A free frame is located, possibly from a free-frame list.

Step 4: A disk operation is scheduled to bring in the necessary page from disk.

Step 5: When the I/O operation is complete, the process's page table is updated with the new frame number, and the invalid bit is changed to indicate that this is now a valid page reference.

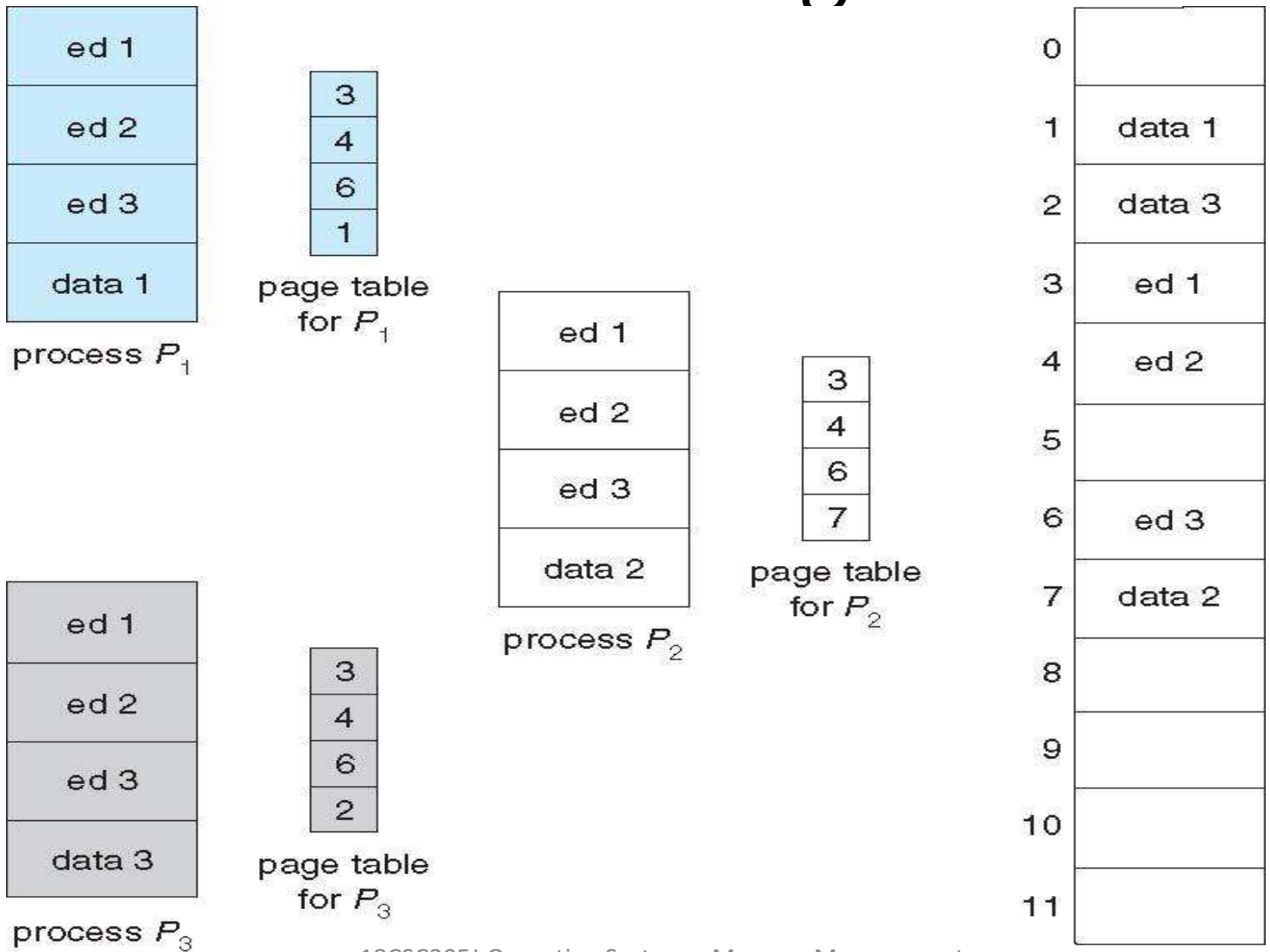
Step 6: The instruction that caused the page fault must now be restarted from the beginning


3
page is on
backing store
operating
system
2
trap
reference
1
load M
i
6
restart
instruction
page table
free frame
5
reset page
table
4
bring in
missing page
physical
memory

Shared Pages

- **Shared code**
  - One copy of read-only (**reentrant**) code shared among processes (i.e., text editors, compilers, window systems)
  - Similar to multiple threads sharing the same process space
  - Also useful for interprocess communication if sharing of read-write pages is allowed
- **Private code and data**
  - Each process keeps a separate copy of the code and data
  - The pages for the private code and data can appear anywhere in the logical address space

Shared Pages



Shared Pages

- In the example figure, three pages ed1,ed2 and ed3 are shred by three processes Process1 , Process2, Process3.
- The page is loaded only once and the corresponding frame number is updated in the page table of the three processes.

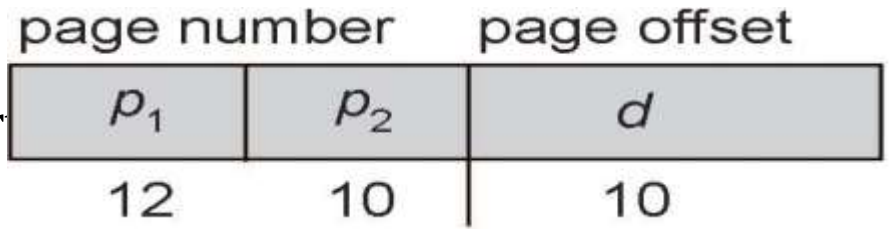
STRUCTURE OF A PAGE TABLE

Structure of the Page Table

- Memory structures for paging can get huge using straight-forward methods
  - Consider a 32-bit logical address space as on modern computers
  - 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
    - Don’t want to allocate that contiguously in main memory
- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

Two-Level Paging Example

- Simple solution to this problem is to divide the page table into smaller pieces We can accomplish this division in several ways
- A logical address (on 32-bit machine with 1K page size) is divided into:
  - a page number consisting of 22 bits
  - a page offset consisting of 10 bits
- Since the page table is paged, the page number is fur
  - a 12-bit page number
  - a 10-bit page offset



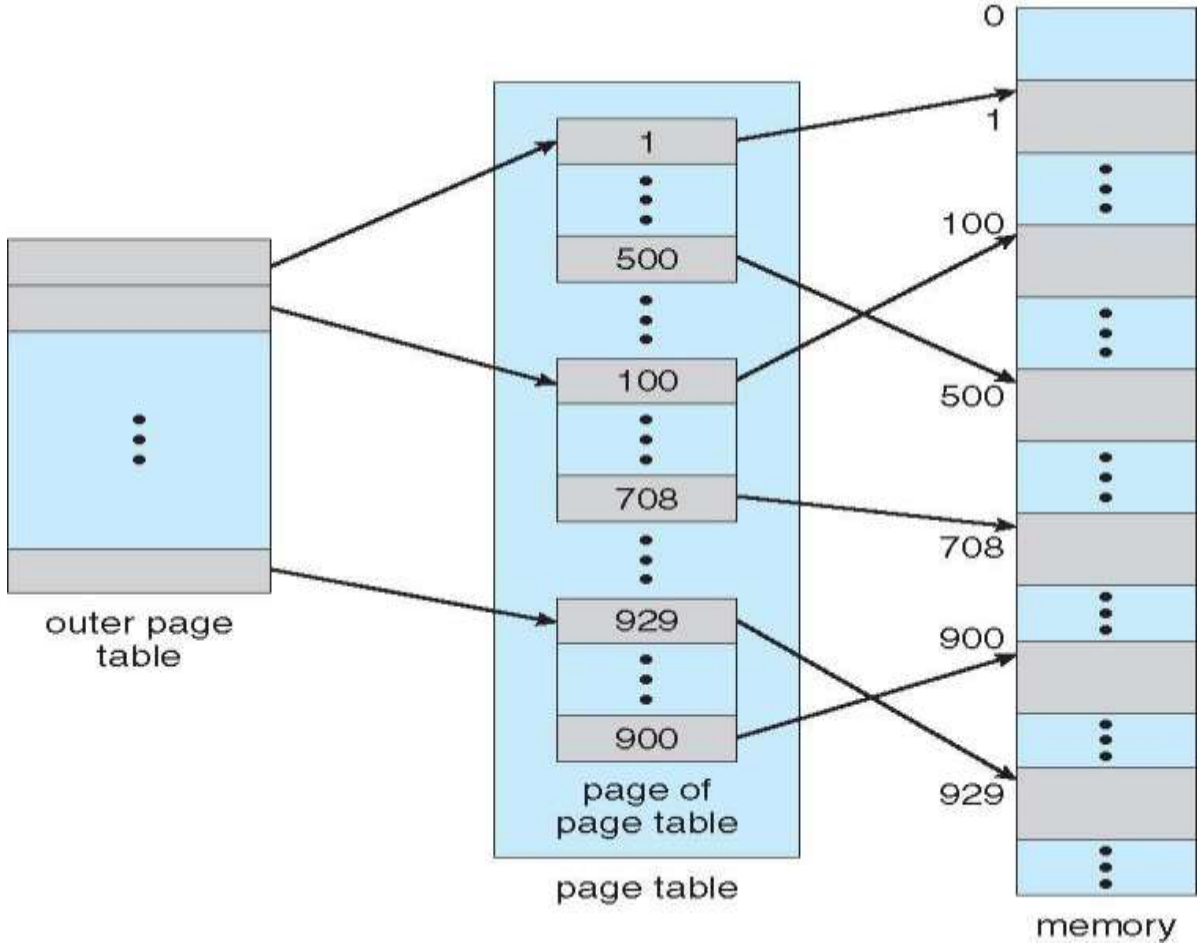
- Thus, a logical address is as follows:

- where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table

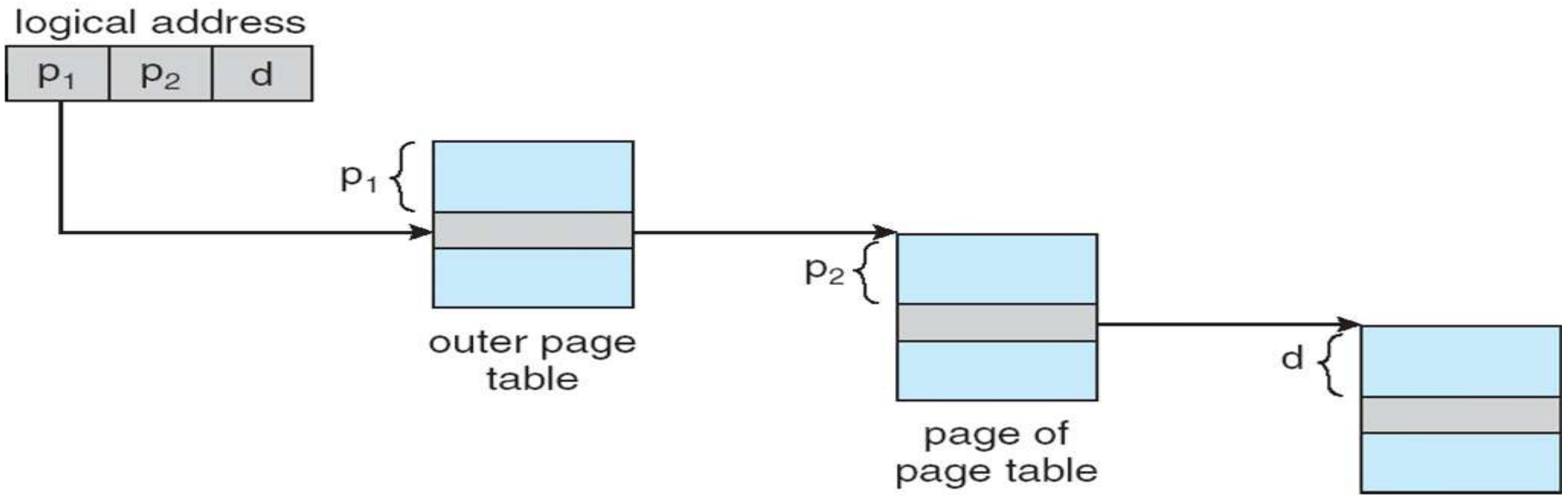
Hierarchical Page Tables

- 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

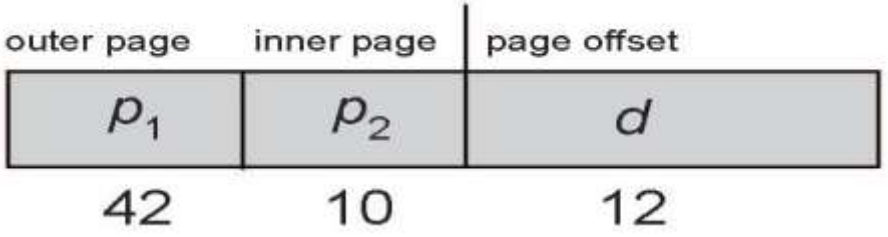


Address-Translation Scheme for two level 32 bit paging architecture



64-bit Logical Address Space

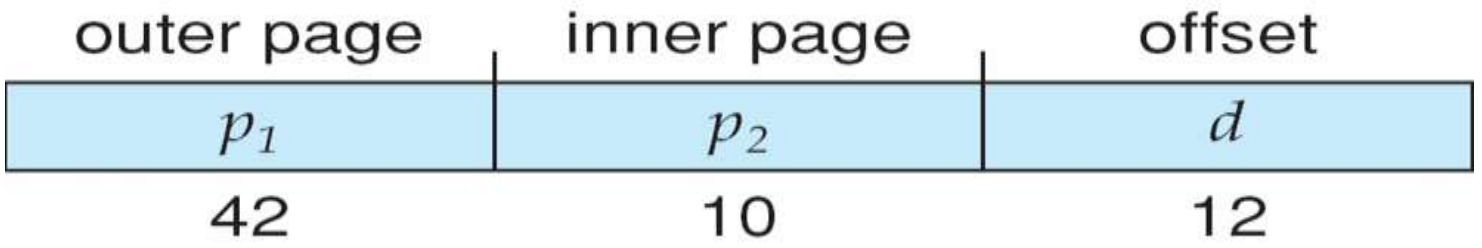
- For a system with a 64-bit logical address space, a two-level paging scheme is no longer appropriate
- If page size is 4 KB (2^{12})
  - Then page table has 2^{52} entries
  - If two level scheme, inner page tables could be 2^{10} 4-byte entries
  - Address would look like

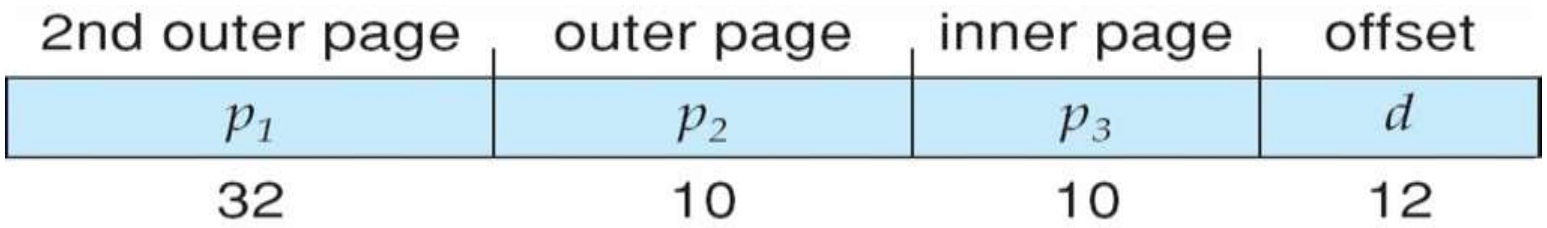


  - Outer page table has 2^{42} entries or 2^{44} bytes
  - One solution is to add a 2^{nd} outer page table
  - But in the following example the 2^{nd} outer page table is still 2^{34} bytes in size
    - And possibly 4 memory access to get to one physical memory location

- We can divide the outer page table in various ways. For example, we can page the outer page table, giving us a three-level paging scheme. Suppose that the outer page table is made up of standard-size pages (210 entries, or 212 bytes). In this case, a 64-bit address space is still daunting
- This keeps growing that is the reason hierarchical page tables are generally considered inappropriate

Three-level Paging Scheme





Hashed Page Tables

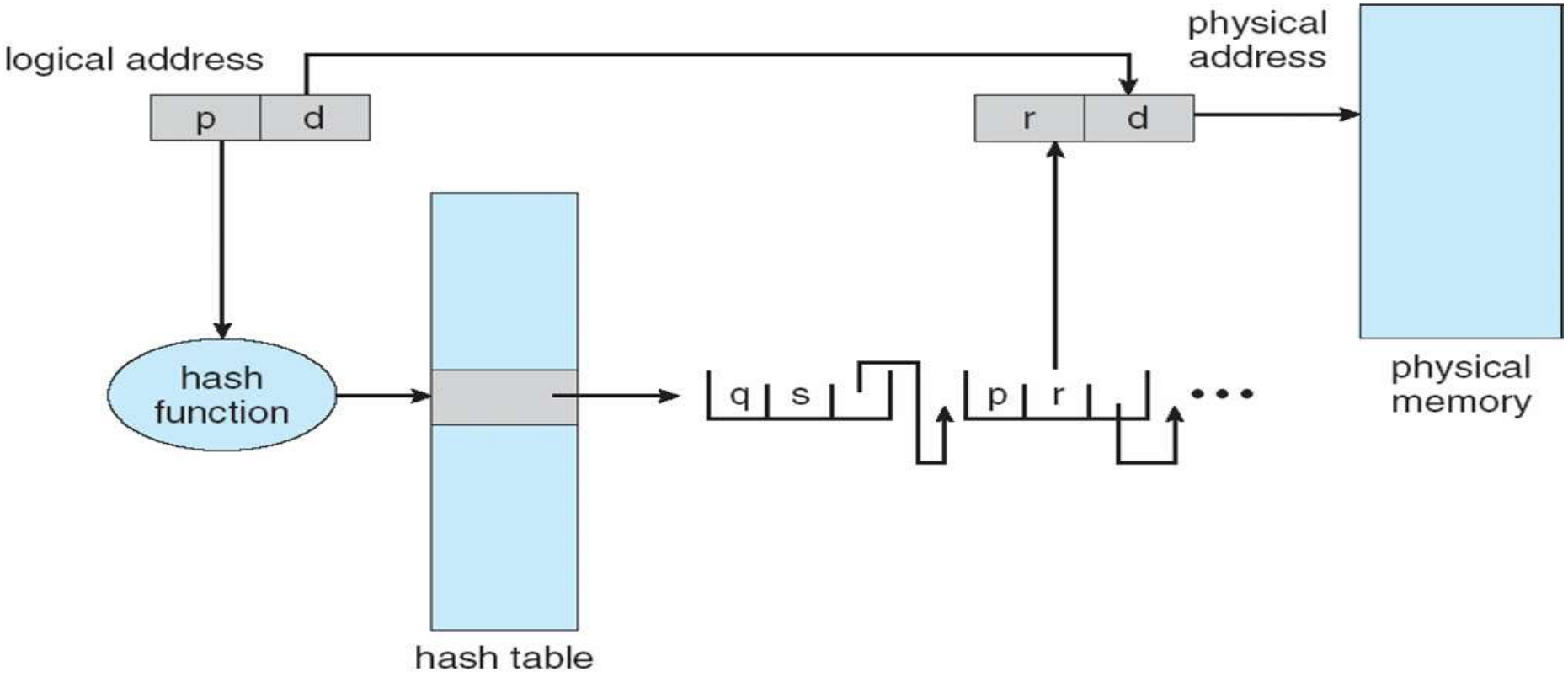
- One approach for handling address spaces larger than 32 bits is to use a hashed page table, with the hash value being the virtual page number
- Common in address spaces > 32 bits
- The virtual page number is hashed into a page table
  - This page table contains a chain of elements hashing to the same location
- Each element contains (1) the virtual page number (2) the value of the mapped page frame (3) a pointer to the next element

Hashed Page Tables (Cont.)

- Virtual page numbers are compared in this chain searching for a match
  - If a match is found, the corresponding physical frame is extracted

- Variation for 64-bit addresses is **clustered page tables**
  - Similar to hashed but each entry refers to several pages (such as 16) rather than 1
  - Especially useful for **sparse** address spaces (where memory references are non-contiguous and scattered)

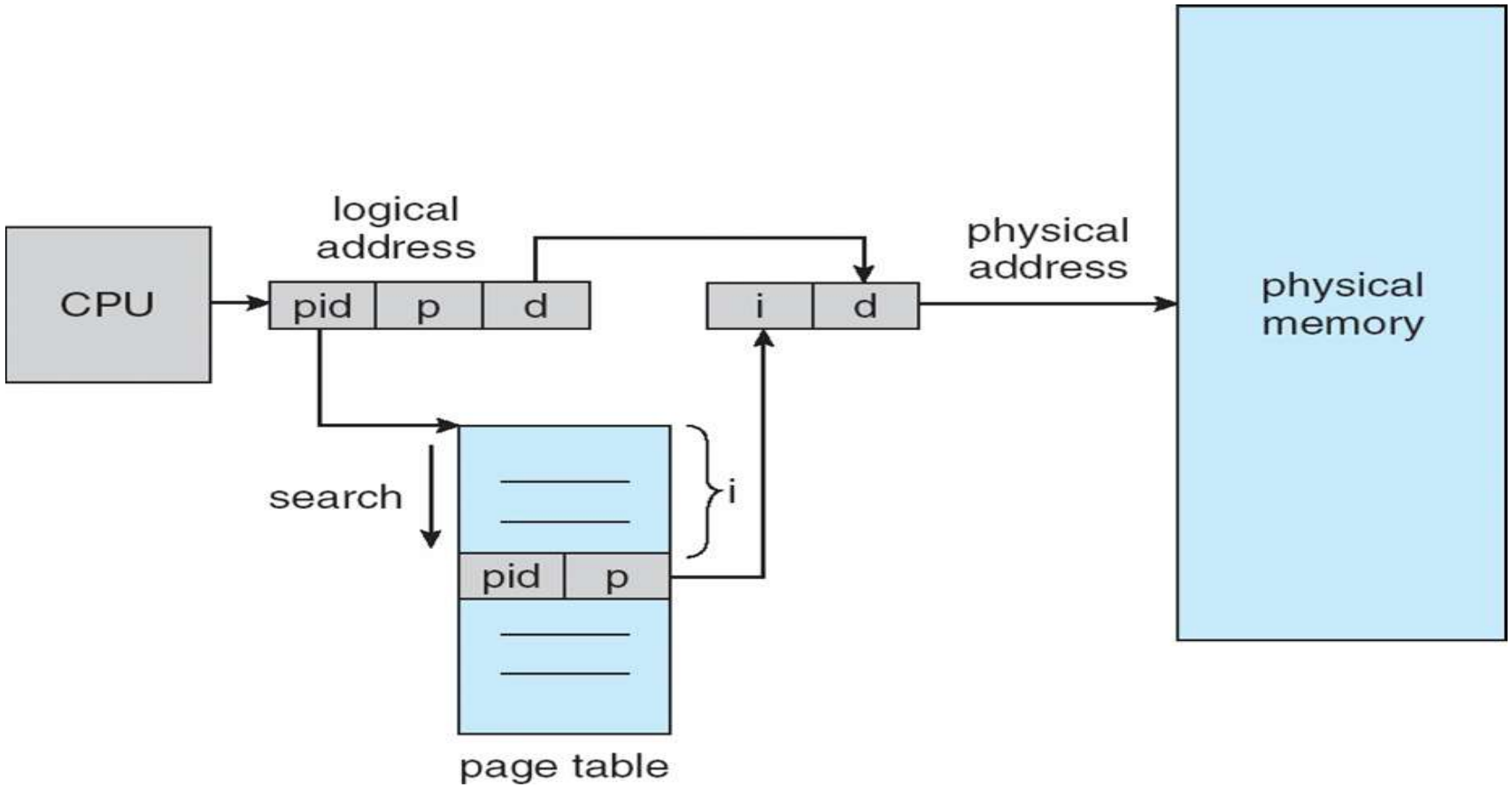
Hashed Page Table



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 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

Inverted Page Table Architecture

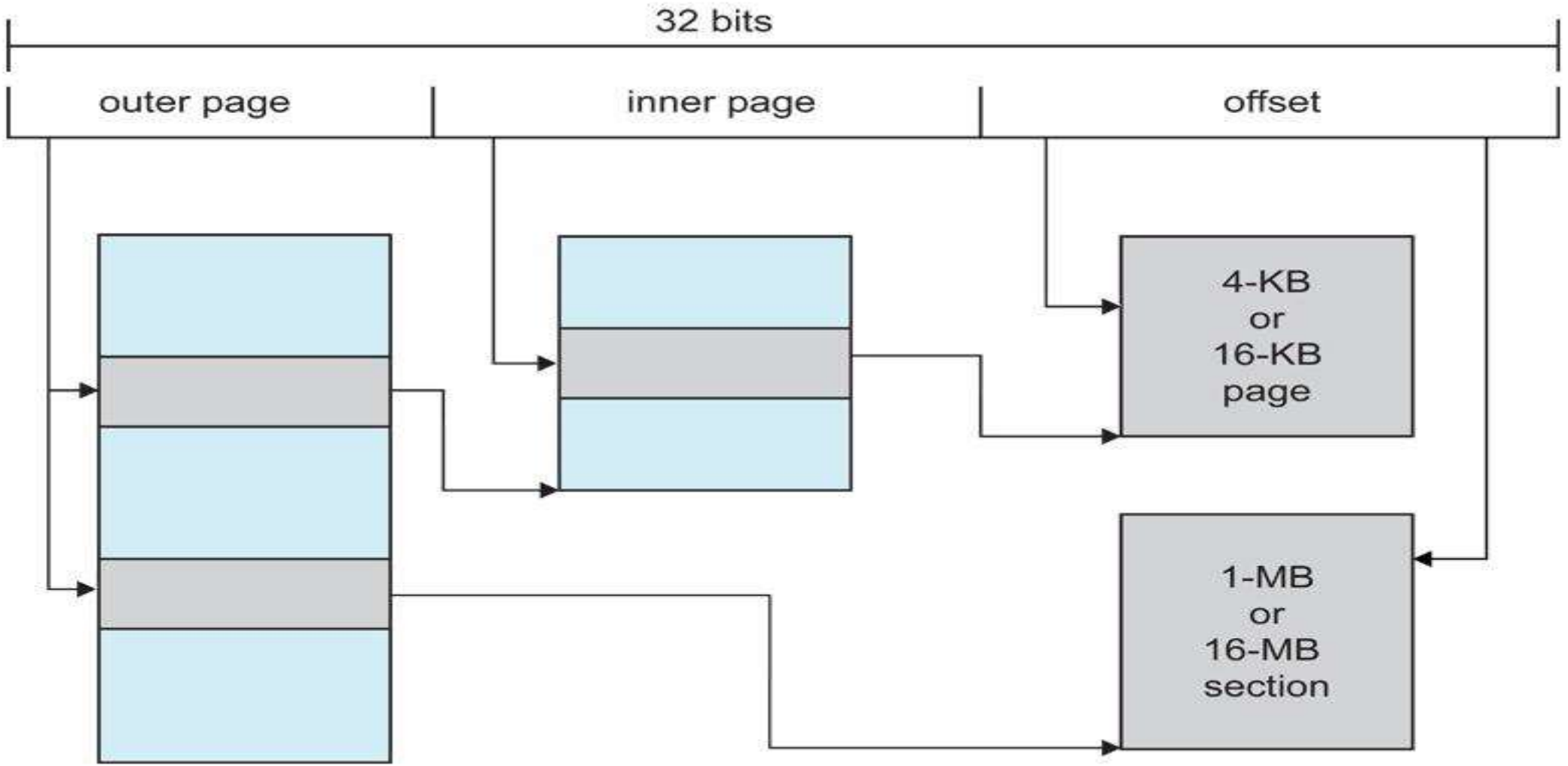


Paging - ARM

- The paging scheme used by ARM processors uses a 2 level page table.
- The first level page table has 16kB size and each entry covers a region of 1MB (sections).
- The second level page tables have 1kB size and each entry covers a region of 4KB (pages).
- The TLB also supports 16MB sections and 64kB pages

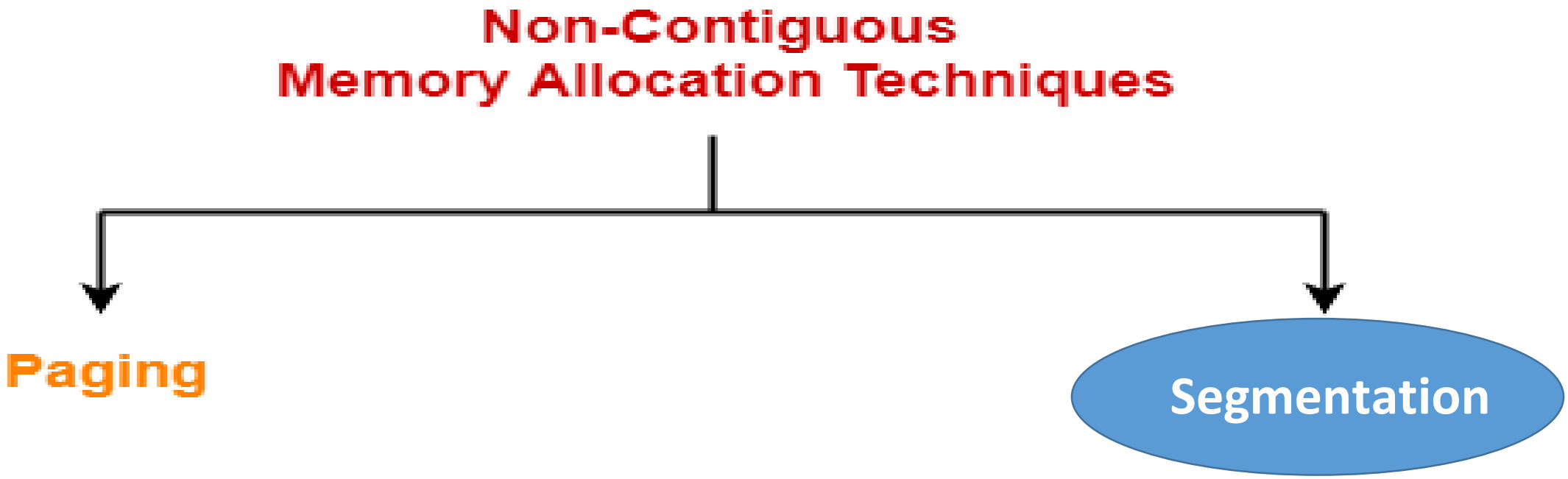
Paging - ARM

- On ARMv4 and ARMv5 pages might also be divided into what the manual calls 'subpages‘
- Example : for a 4KB page, 1KB subpages, and for a 64KB page, 16KB subpages.
- The ARM processor supports 2 independent page tables.
- The first page table can be abbreviated from the full 16kB
- The second page table is always 16KB in size and is used for addresses beyond the end of the first page table.


32 bits
outer page
inner page
offset
4-KB
or
16-KB
page
1-MB
or
16-MB
section

SEGMENTATION

Non-Contiguous Memory Allocation



Segmentation

- Memory-management scheme that supports user view of memory.
- Segmentation overcomes the problem of dealing with memory in terms of its physical properties which is inconvenient to both the operating system and the programmer.
- The system have more freedom to manage memory.
- The programmer would get an experience to work in more natural programming environment.

Program and Segment

- A program is a collection of segments in which segment is a logical unit which consists of the following :
    - main program
    - procedure
    - Function
    - Method
    - Object
    - local variables, global variables
    - common block
    - Stack
    - symbol table
    - arrays

User's View of a Program

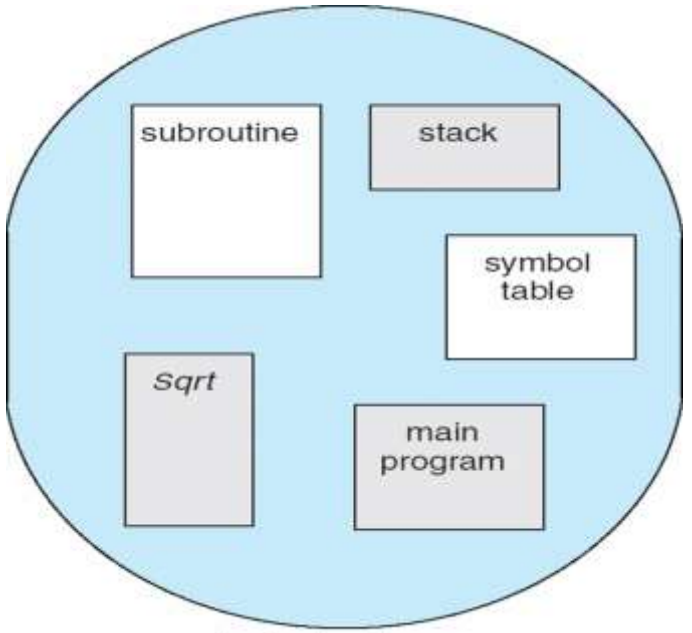
- Most programmers prefer to view memory as a collection of variable-sized segments with no necessary ordering among the segments.

- From programmer point of view , a program will appear as a main program with a set of methods, procedures, or functions. Program also includes the following data structures:
  - Objects
  - Arrays
  - Stacks and
  - Variables

- Modules or data elements is referred by name.

- The programmer talks about "the stack," "the math library," and "the main program" without caring what addresses in memory these elements occupy.

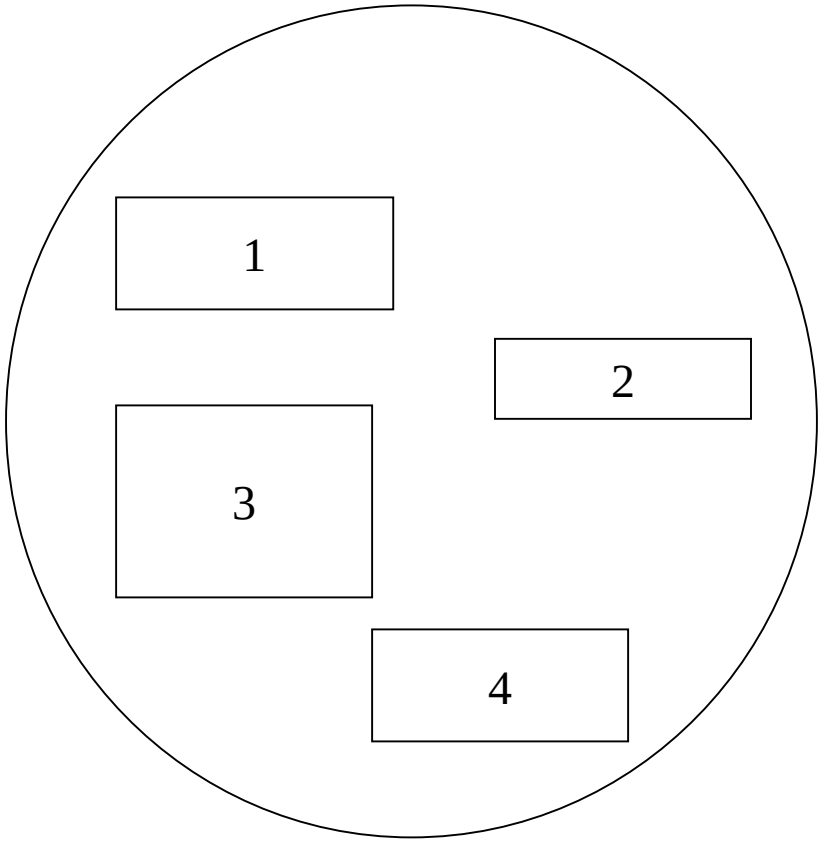
- Programmer is not concerned with whether the stack is stored before or after the Sqrt() function.



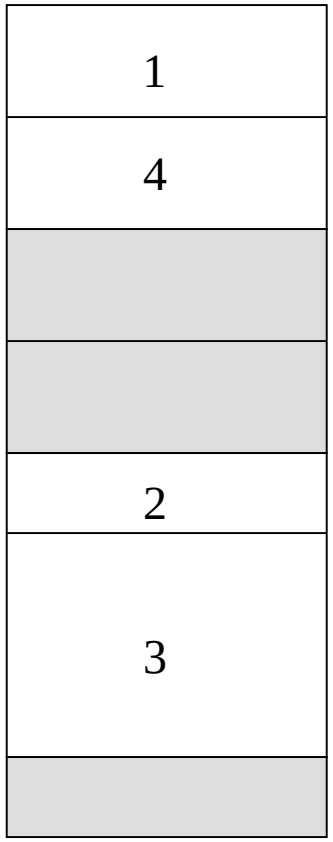
Logical View of Segmentation

- In the figure, both user space and physical memory space are given.
- Segments vary in length, and the length of each is intrinsically defined by its purpose in the program.
- Segments are numbered as 1,2,3 and 4.
- In Physical memory space, Segments are allocated in different parts of the memory
- For example,

  Elements within a segment are identified by their offset from the beginning of the segment: the first statement of the program, the seventh stack frame entry in the stack and the fifth instruction of the Sqrt()



user space



physical memory space

Example – Compilation of C program

- Normally, when a program is compiled, the compiler automatically constructs segments reflecting the input program.

- A “C” compiler might create separate segments for the following:
  - The code
  - Global variables
  - The heap, from which memory is allocated
  - The stacks used by each thread
  - The standard C library

- Libraries that are linked in during compile time might be assigned separate segments.

- The loader would take all these segments and assign them segment numbers.

Segmentation Architecture

- Logical address consists of a two tuple:

  <segment-number, offset>

- A logical address consists of two parts: a segment number, *s*, and an offset into that segment, *d*.
- 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, a trap is sent to the operating system (logical addressing attempt beyond end of segment).
- When an offset *d* is legal, it is added to the segment base to produce the address in physical memory of the desired byte.

Segmentation Architecture (Cont.)

- **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

- **Segment-table base register (STBR)** points to the segment table's location in memory

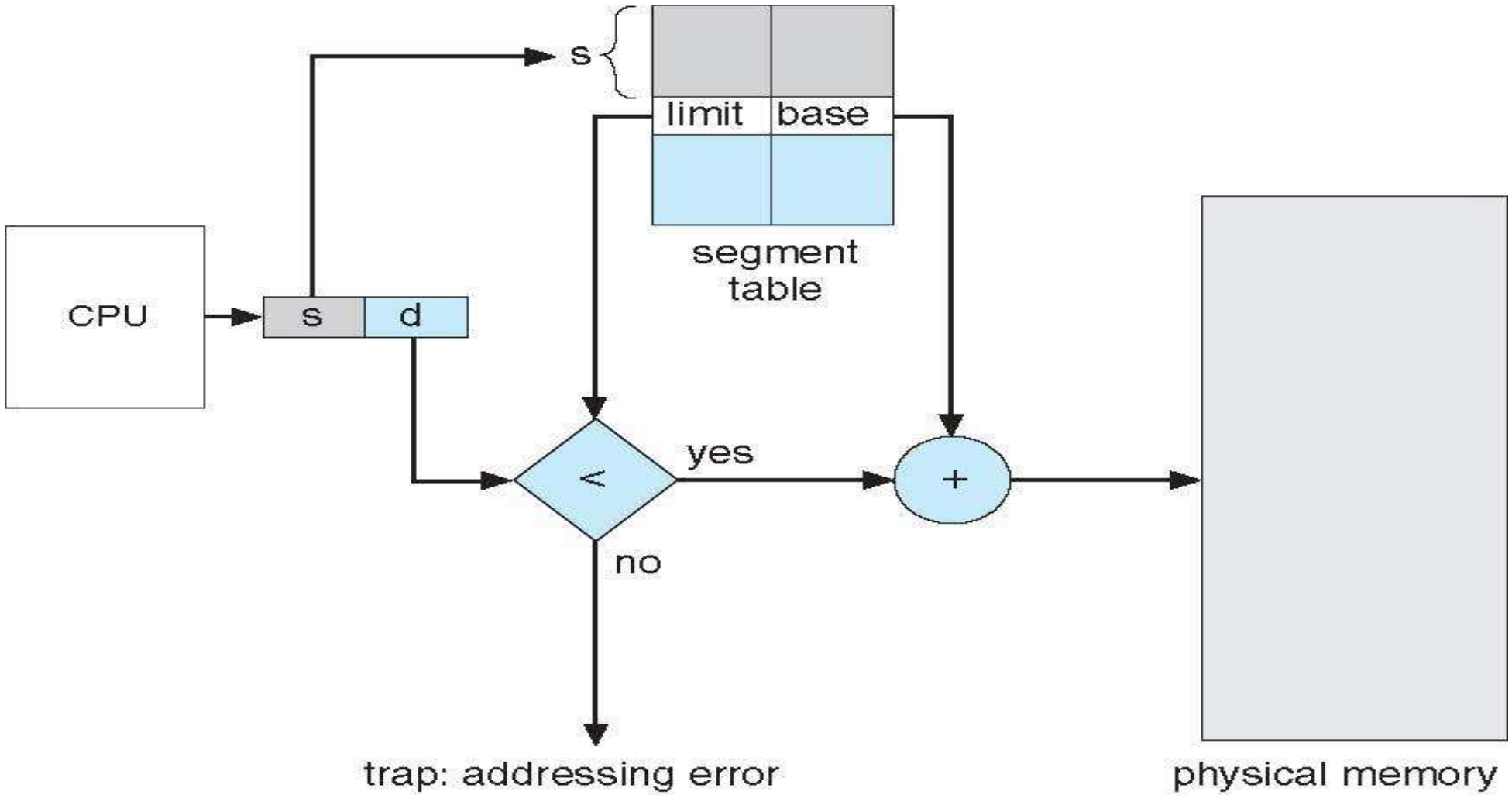
- **Segment-table length register (STLR)** indicates number of segments used by a program;

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

Segmentation Architecture (Cont.)

- Protection
  - With each entry in segment table associate:
    - validation bit = 0 ⇒ illegal segment
    - read/write/execute privileges
- Protection bits associated with segments with code sharing occurs at segment level
- Since segments varies in length, memory allocation is a dynamic

Segmentation Hardware



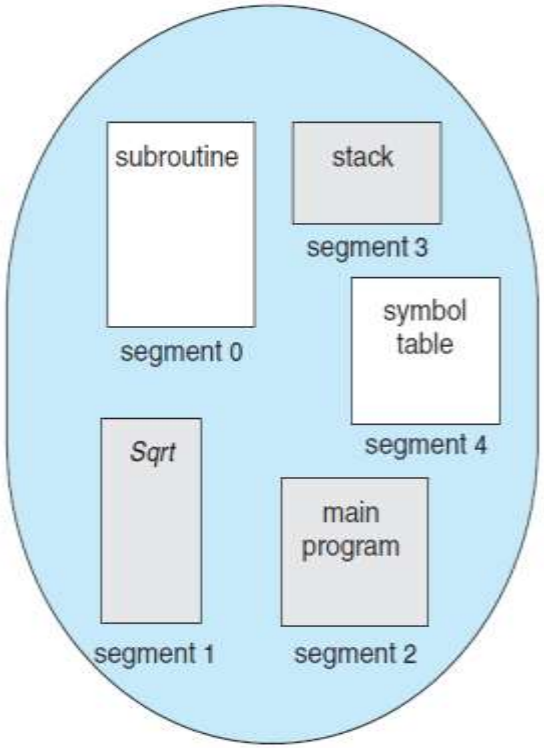
Segmentation Example

- There are Five Segments numbered from 0 through 4.
- The segments are stored in physical memory as shown in the diagram.
- 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).

For example, segment 2 is 400 bytes long and begins at location 4300.
A reference to byte 53 of segment 2 is mapped onto location 4300 + 53 = 4353.

A reference to segment 3, byte 852, is mapped to 3200 (the base of segment 3) + 852 = 4052.

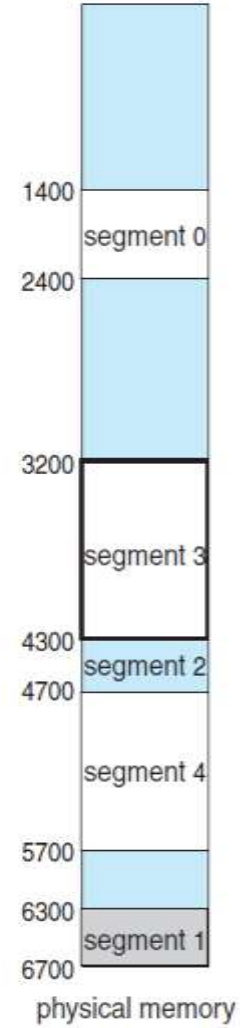
A reference to byte 1222 of segment 0 would result in a **trap** to the operating system, as this segment is only 1,000 bytes long.



logical address space

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

segment table



physical memory

Paged Segmentation

- Paging + Segmentation
- Takes advantage of both paging and segmentation
- Implementation: Treat each segment independently. Each segment has a page table
- Logical address now consists of 3 parts:
  - Segment number
  - Page number
  - Offset

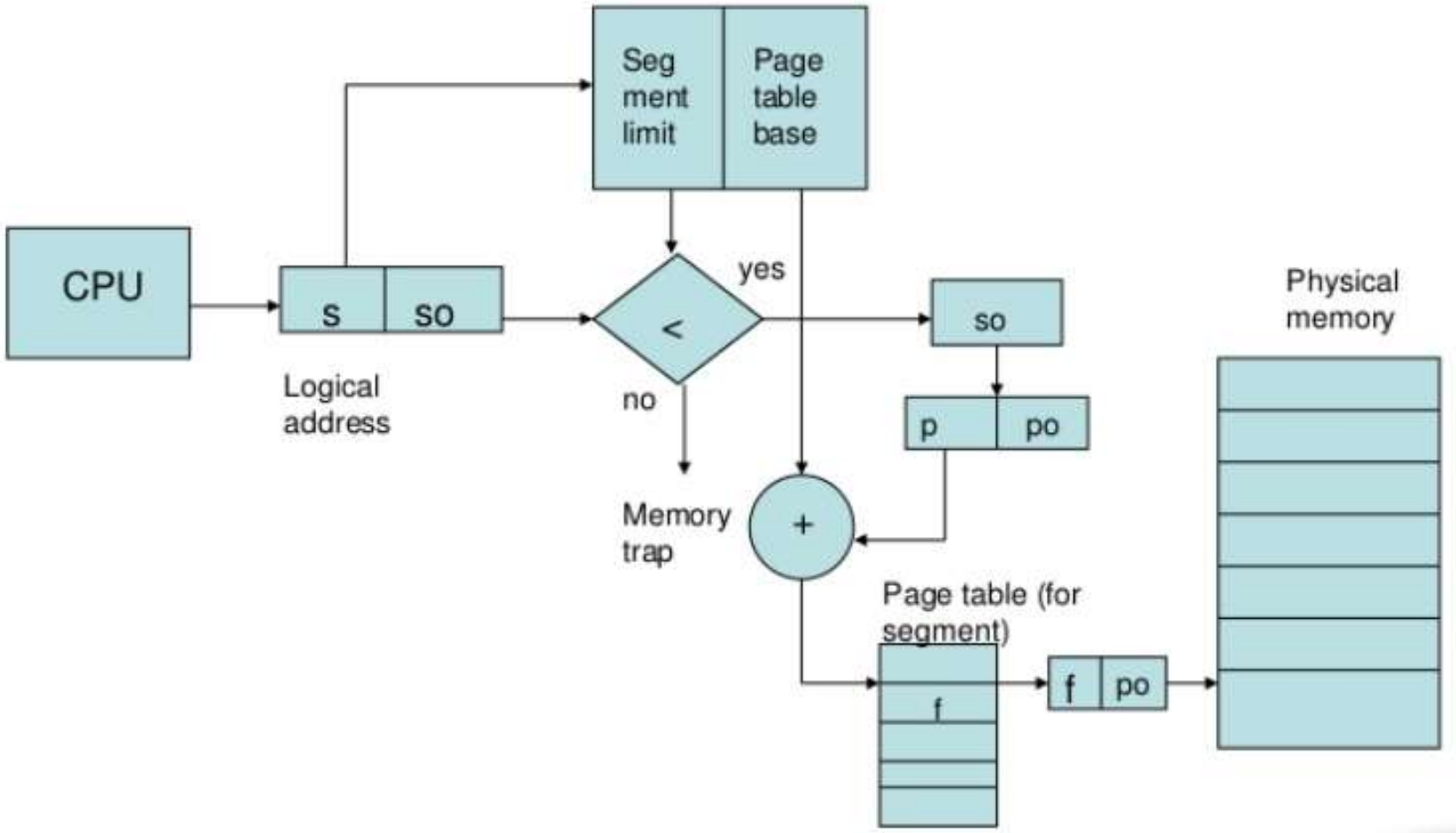
Paged Segmentation Implementation

- Segment table has base address for page table
- Look up <page number, offset> in page table
- Get physical address in return
- Examples: MULTICS and Intel 386

Combined Paging and Segmentation Scheme

- In a combined paging/segmentation system, a user's address is broken up into a number of segments.
- Each segment is broken up into a number of fixed-sized pages which are equal in length to the main memory frame

Address Translation

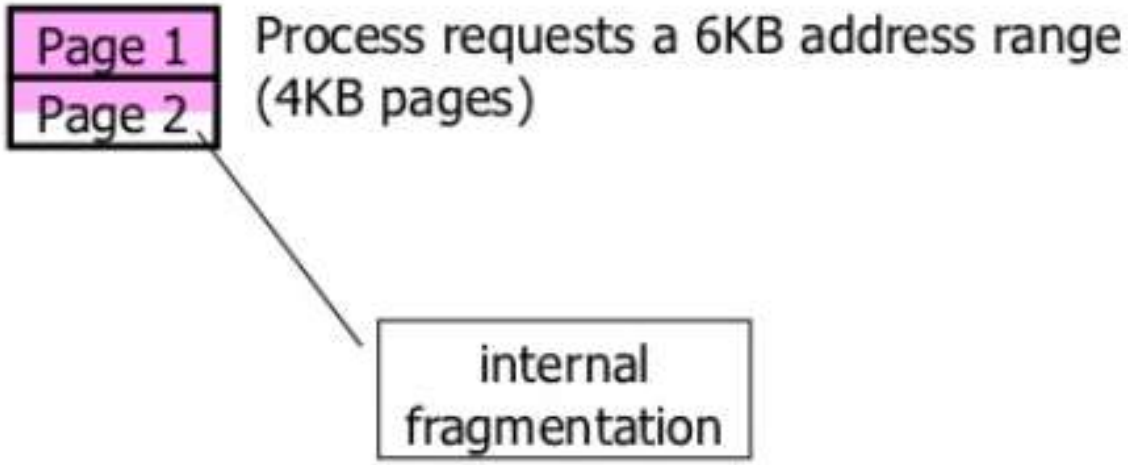


Advantages of Combined Scheme

- Reduces memory usage as opposed to pure paging
  - Page table size limited by segment size
  - Segment table has only one entry per actual segment
- Share individual pages by copying page table entries
- Share whole segments by sharing segment table entries, which is the same as sharing the page table for that segment
- Most advantages of paging still hold
  - Simplifies memory allocation
  - Eliminates external fragmentation
- This system combines the efficiency in paging with protection and sharing capabilities of segmentation

Disadvantage

- Internal fragmentation still exists

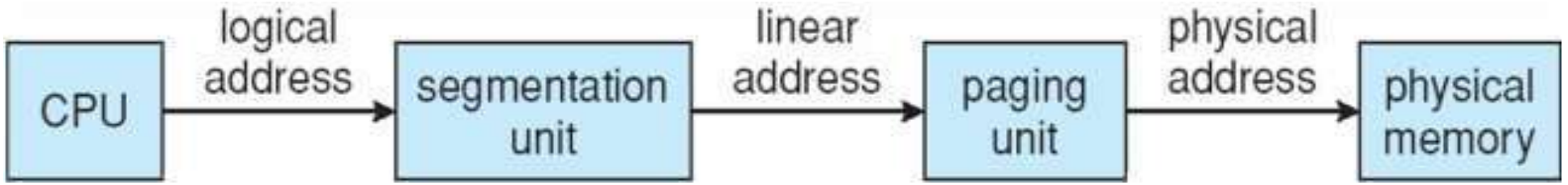


Example: The Intel 32 and 64-bit Architectures

- 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

- Memory management is divided as – Segmentation and Paging
- 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

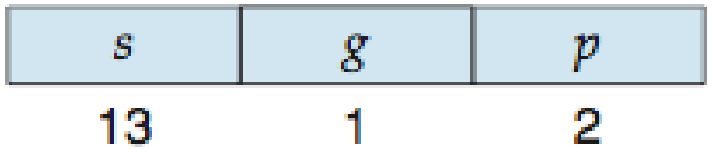


The Intel IA-32 Segmentation

- Supports both segmentation and 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**))

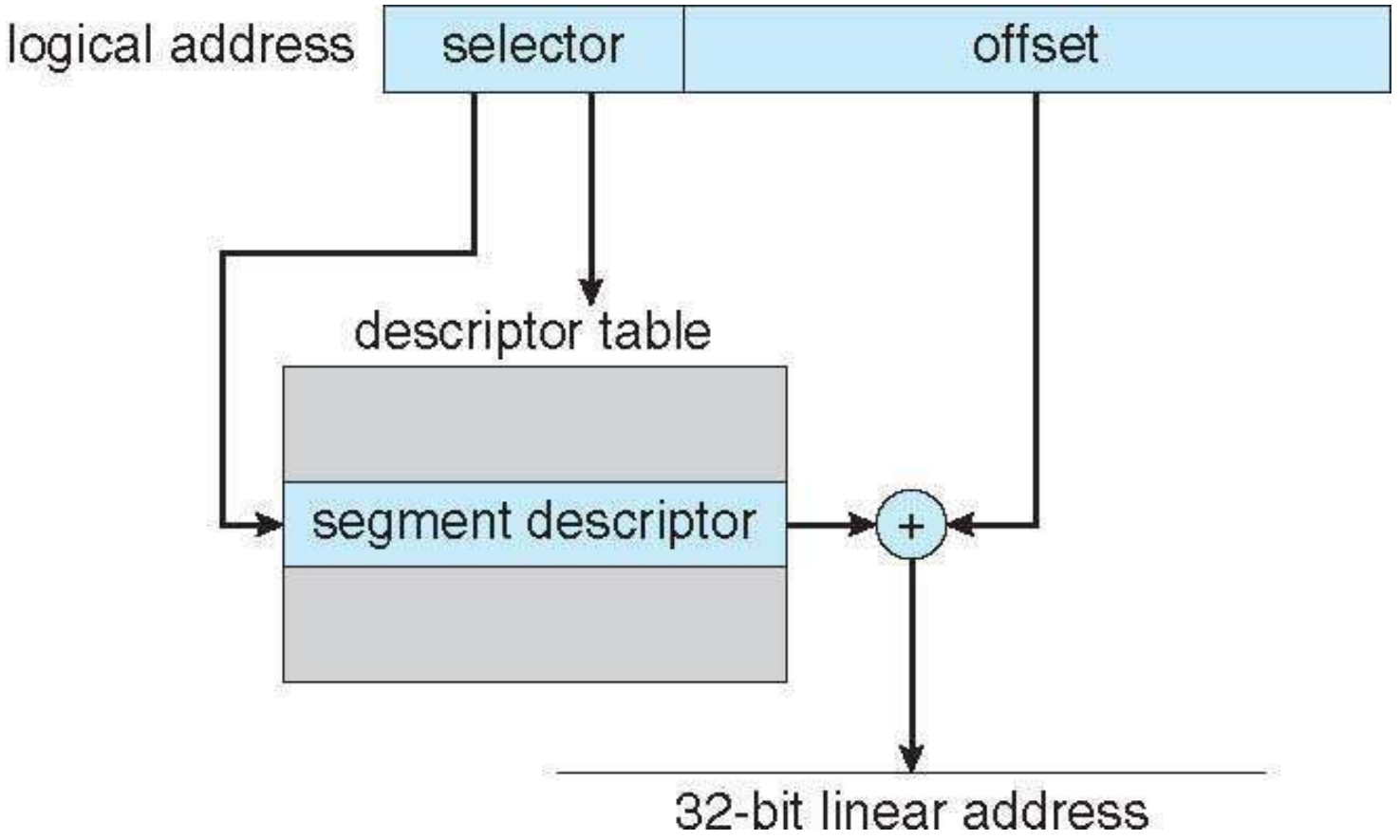
Intel IA-32 Segmentation

- The logical address is a pair (selector, offset), where the selector is a 16-bit number:



- s designates segment number
- *g* indicates if segment is GDT or LDT
- *p* deals with protection
- Offset is a 32-bit number and it specifies the location of the byte within segment

Intel IA-32 Segmentation

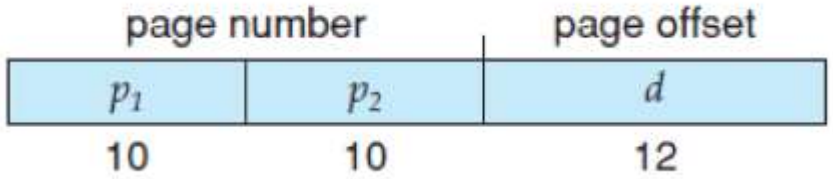


Intel IA-32 Segmentation

- There are six segment registers – six segments can be addressed at a time
- There are six 8-byte microprogram registers to hold LDT or GDT descriptors
- Linear address is 32-bits long
  - Base and limit of segment is used to generate linear address
  - Limit is used to check address validity
  - Invalid address results in a trap
  - Value of base with offset results in 32-bit address

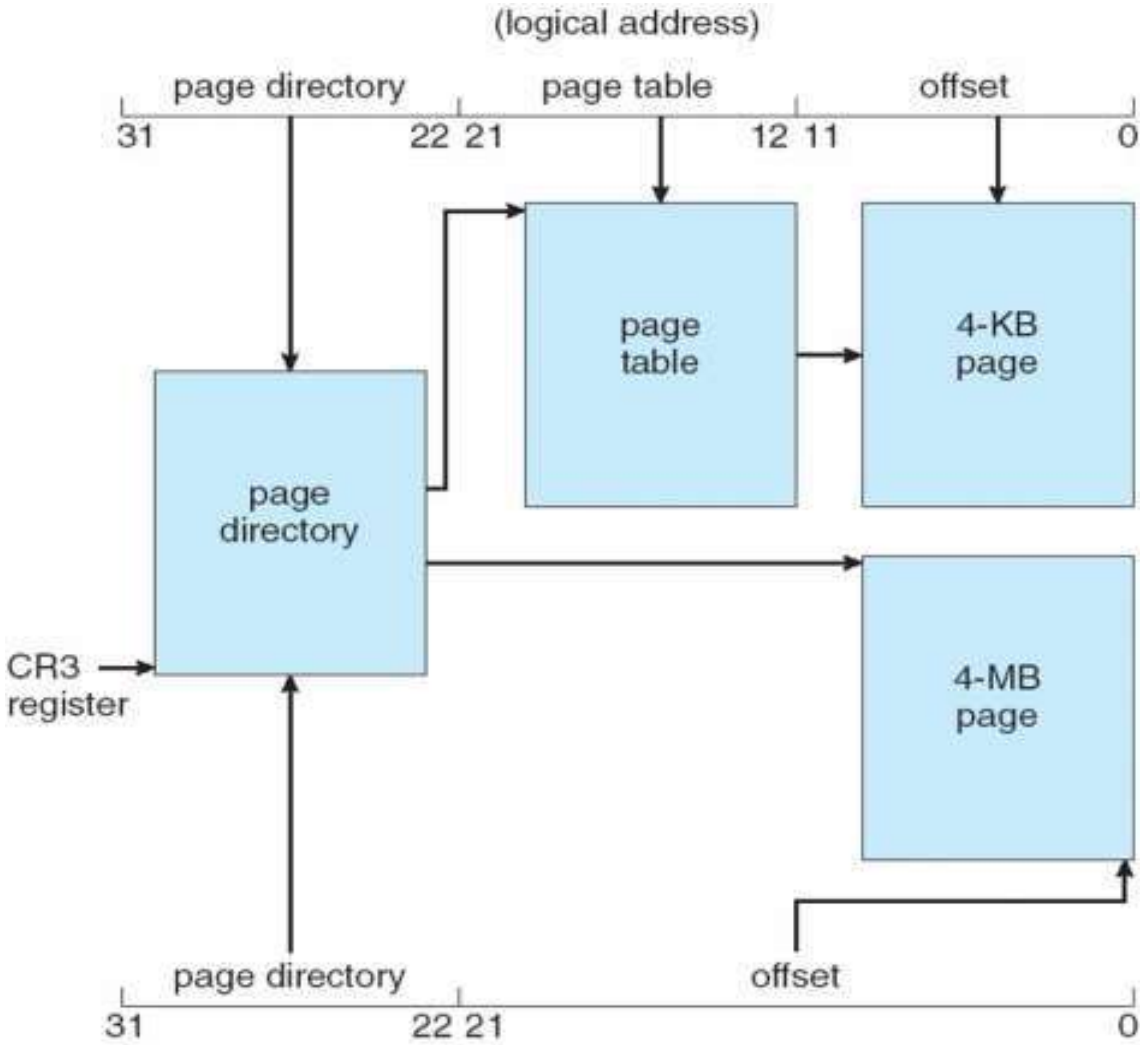
IA-32 Paging

- Pages sizes of 4KB or 4MB is allowed in IA-32
- 4KB pages use two-level paging scheme



- The 10 higher order bits reference the page directory – outermost page table
- The innermost 10 bits refer the inner page table
- Lower order 12 bits refer the offset in the page
- 1 bit in page directory is Page-Size flag
  - If set, indicates 4MB page rather than the default 4KB page

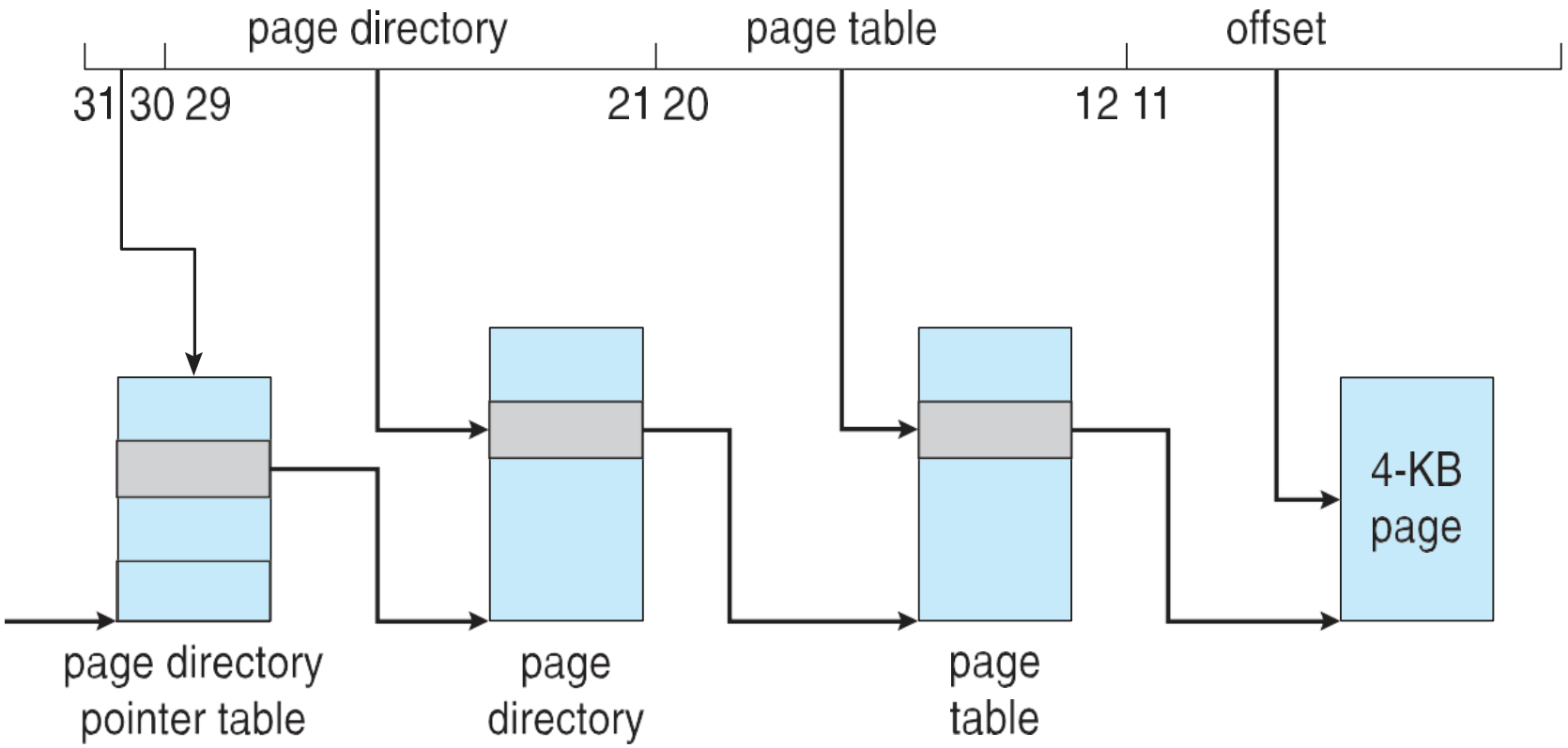
Intel IA-32 Paging Architecture



Intel IA-32 Page Address Extensions

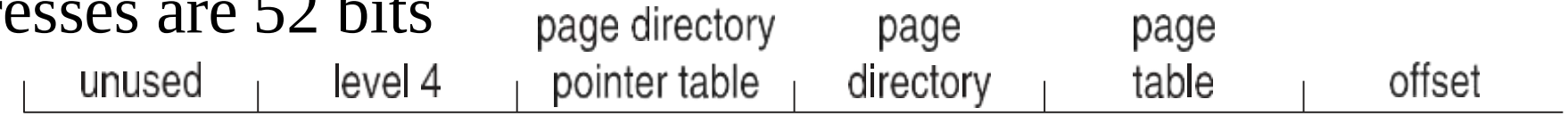
- 32-bit address limits led Intel to create page address extension (PAE), allowing 32-bit apps access to more than 4GB of memory space
- Paging went to a 3-level scheme
- Top two bits refer to a page directory pointer table
- Page-directory and page-table entries moved to 64-bits in size
- Net effect is increasing address space to 36 bits – 64GB of physical memory
- Linux and MAC support PAE
- But, 32-bit versions of Windows desktop operating systems still provide support for only 4 GB of physical memory, even if PAE is enabled.

Intel IA-32 Page Address Extensions



Intel x86-64

- Current generation Intel x86 architecture
- Support for larger physical and logical address spaces
- 64 bit-systems can address 2^{64} of addressable memory (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



Reference

- Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, Operating systems, 9th ed., John Wiley & Sons, 2013