Operating Systems with Linux

(MCA-105)

Unit – 3

by

Dr. Sunil Pratap Singh
(Associate Professor, BVICAM, New Delhi)

2023

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.1

Deadlock

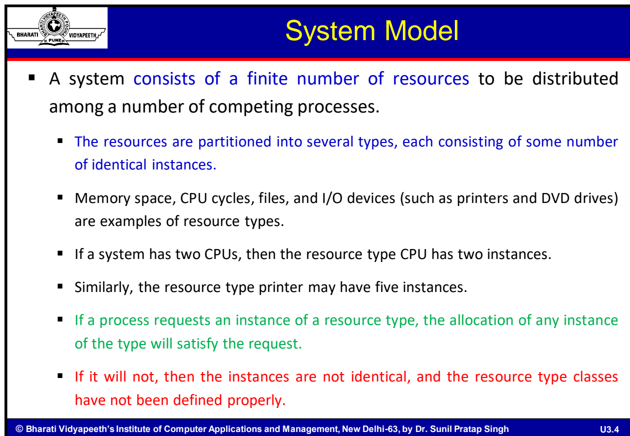
- In a multiprogramming environment, several processes may compete for a finite number of resources.
- A process requests resources; if the resources are not available at that time, the process enters a waiting state.
- Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes.
- This situation is called a deadlock.

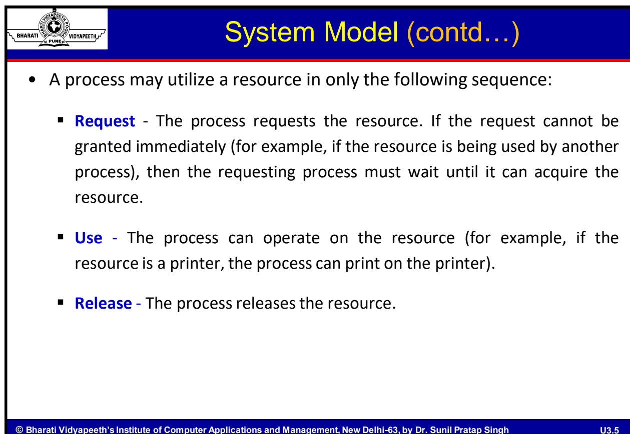
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.2

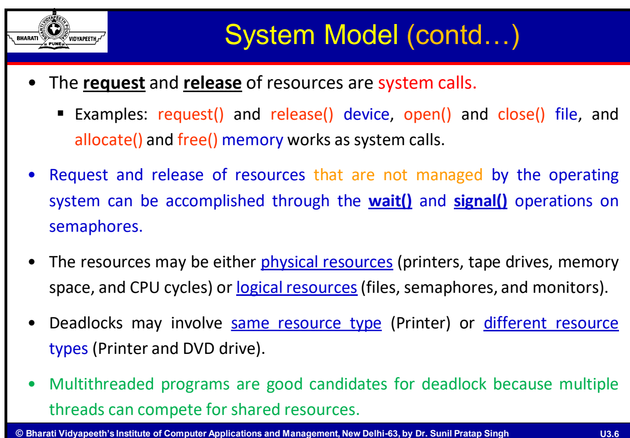
Deadlock (contd…)

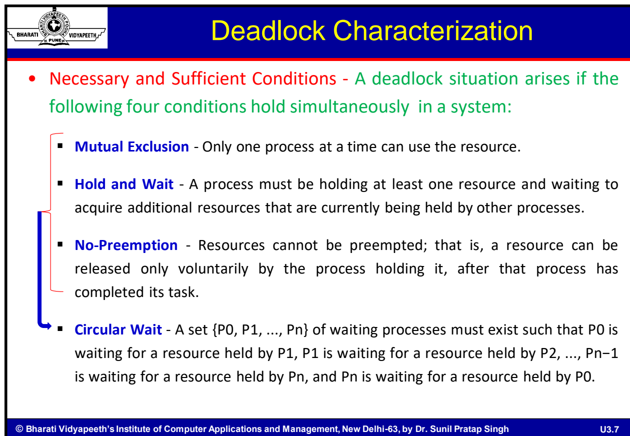
- Deadlock can be defined as the permanent blocking of a set of processes that either compete for system resources or communicate with each other.
- A set of processes is deadlocked when each process in the set is blocked awaiting an event (typically the freeing up of some requested resource) that can only be triggered by another blocked process in the set.

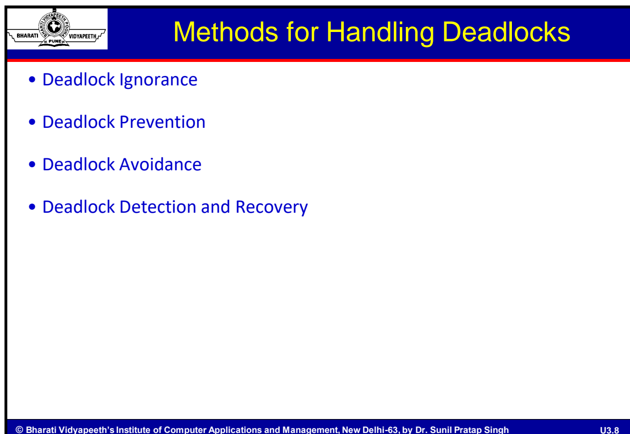
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.3

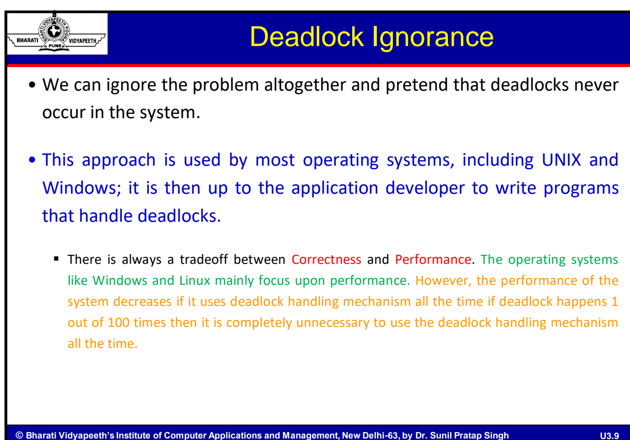

System Model
A system consists of a finite number of resources to be distributed among a number of competing processes.
The resources are partitioned into several types, each consisting of some number of identical instances.
Memory space, CPU cycles, files, and I/O devices (such as printers and DVD drives) are examples of resource types.
If a system has two CPUs, then the resource type CPU has two instances.
Similarly, the resource type printer may have five instances.
If a process requests an instance of a resource type, the allocation of any instance of the type will satisfy the request.
If it will not, then the instances are not identical, and the resource type classes have not been defined properly.
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh
U3.4

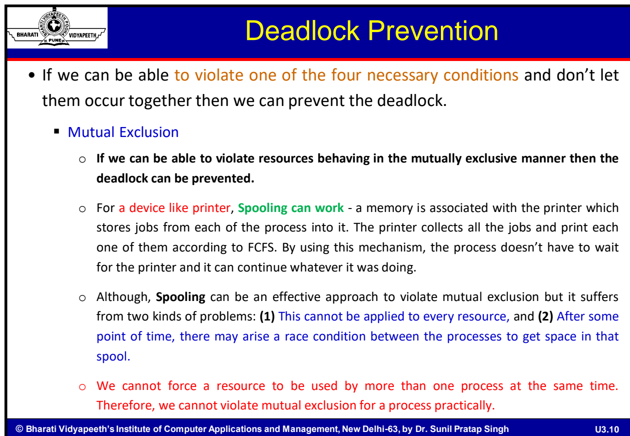

System Model (contd…)
A process may utilize a resource in only the following sequence:
Request - The process requests the resource. If the request cannot be granted immediately (for example, if the resource is being used by another process), then the requesting process must wait until it can acquire the resource.
Use - The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).
Release - The process releases the resource.
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh
U3.5

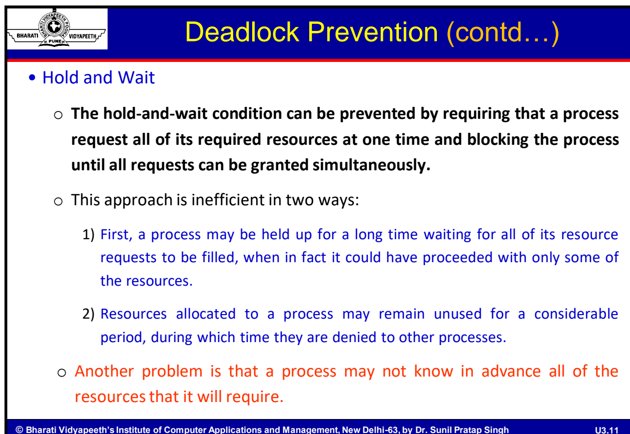

System Model (contd…)
The request and release of resources are system calls.
Examples: request() and release() device, open() and close() file, and allocate() and free() memory works as system calls.
Request and release of resources that are not managed by the operating system can be accomplished through the wait() and signal() operations on semaphores.
The resources may be either physical resources (printers, tape drives, memory space, and CPU cycles) or logical resources (files, semaphores, and monitors).
Deadlocks may involve same resource type (Printer) or different resource types (Printer and DVD drive).
Multithreaded programs are good candidates for deadlock because multiple threads can compete for shared resources.
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh
U3.6

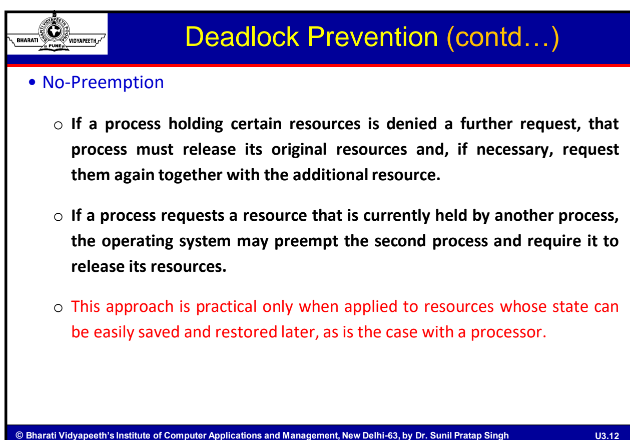

BHARATI VIDYAPEETH PUNE
Deadlock Characterization
• Necessary and Sufficient Conditions - A deadlock situation arises if the following four conditions hold simultaneously in a system:
▪ Mutual Exclusion - Only one process at a time can use the resource.
▪ Hold and Wait - A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.
▪ No-Preemption - Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.
▪ Circular Wait - A set {P0, P1, ..., Pn} of waiting processes must exist such that P0 is waiting for a resource held by P1, P1 is waiting for a resource held by P2, ..., Pn−1 is waiting for a resource held by Pn, and Pn is waiting for a resource held by P0.
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh
U3.7


BHARATI VIDYAPEETH PUNE
Methods for Handling Deadlocks
• Deadlock Ignorance
• Deadlock Prevention
• Deadlock Avoidance
• Deadlock Detection and Recovery
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh
U3.8


BHARATI VIDYAPEETH PUNE
Deadlock Ignorance
• We can ignore the problem altogether and pretend that deadlocks never occur in the system.
• This approach is used by most operating systems, including UNIX and Windows; it is then up to the application developer to write programs that handle deadlocks.
▪ There is always a tradeoff between Correctness and Performance. The operating systems like Windows and Linux mainly focus upon performance. However, the performance of the system decreases if it uses deadlock handling mechanism all the time if deadlock happens 1 out of 100 times then it is completely unnecessary to use the deadlock handling mechanism all the time.
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh
U3.9


Deadlock Prevention
• If we can be able to violate one of the four necessary conditions and don't let them occur together then we can prevent the deadlock.
▪ Mutual Exclusion
○ If we can be able to violate resources behaving in the mutually exclusive manner then the deadlock can be prevented.
○ For a device like printer, Spooling can work - a memory is associated with the printer which stores jobs from each of the process into it. The printer collects all the jobs and print each one of them according to FCFS. By using this mechanism, the process doesn't have to wait for the printer and it can continue whatever it was doing.
○ Although, Spooling can be an effective approach to violate mutual exclusion but it suffers from two kinds of problems: (1) This cannot be applied to every resource, and (2) After some point of time, there may arise a race condition between the processes to get space in that spool.
○ We cannot force a resource to be used by more than one process at the same time. Therefore, we cannot violate mutual exclusion for a process practically.
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh
U3.10


Deadlock Prevention (contd…)
• Hold and Wait
○ The hold-and-wait condition can be prevented by requiring that a process request all of its required resources at one time and blocking the process until all requests can be granted simultaneously.
○ This approach is inefficient in two ways:
1) First, a process may be held up for a long time waiting for all of its resource requests to be filled, when in fact it could have proceeded with only some of the resources.
2) Resources allocated to a process may remain unused for a considerable period, during which time they are denied to other processes.
○ Another problem is that a process may not know in advance all of the resources that it will require.
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh
U3.11


Deadlock Prevention (contd…)
• No-Preemption
○ If a process holding certain resources is denied a further request, that process must release its original resources and, if necessary, request them again together with the additional resource.
○ If a process requests a resource that is currently held by another process, the operating system may preempt the second process and require it to release its resources.
○ This approach is practical only when applied to resources whose state can be easily saved and restored later, as is the case with a processor.
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh
U3.12

Deadlock Prevention (contd…)

- Circular Wait
  - **The circular-wait condition can be prevented by defining a linear ordering of resource types.**
  - If a process has been allocated resources of type R, then it may subsequently request only those resources of types following R in the ordering.
  - This approach may be inefficient, slowing down processes and denying resource access unnecessarily.

Deadlock Avoidance

- Deadlock avoidance allows the three necessary conditions but makes judicious choices to assure that the deadlock point is never reached.
- With deadlock avoidance, a decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to a deadlock.
- Deadlock avoidance thus requires knowledge of future process resource requests.
- **Approaches to deadlock avoidance:**
  - Process Initiation Denial
  - Resource Allocation Denial

Process Initiation Denial

Consider a system with n processes and m different types of resources.

[Resource = $\mathbf{R} = (R_1, R_2, \ldots, R_m)$	total amount of each resource in the system
Available = $\mathbf{V} = (V_1, V_2, \ldots, V_m)$	total amount of each resource not allocated to any process
Claim = $\mathbf{C} = \begin{bmatrix} C_{11} & C_{12} & \ldots & C_{1m} \\ C_{21} & C_{22} & \ldots & C_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ C_{n1} & C_{n2} & \ldots & C_{nm} \end{bmatrix}$	C_{ij} = requirement of process i for resource j
Allocation = $\mathbf{A} = \begin{bmatrix} A_{11} & A_{12} & \ldots & A_{1m} \\ A_{21} & A_{22} & \ldots & A_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ A_{n1} & A_{n2} & \ldots & A_{nm} \end{bmatrix}$	A_{ij} = current allocation to process i of resource j

Process Initiation Denial (contd…)

- The following relationship holds:

1. $R_j = V_j + \sum_{i=1}^{n} A_{ij}$, for all j — All resources are either available or allocated.
2. $C_{ij} \leq R_j$, for all i, j — No process can claim more than the total amount of resources in the system.
3. $A_{ij} \leq C_{ij}$, for all i, j — No process is allocated more resources of any type than the process originally claimed to need.

- With these quantities defined, we can define a deadlock avoidance policy that refuses to start a new process if its resource requirements might lead to deadlock.
- Start a new process P_{n+1} only if $R_j \geq C_{(n+1)j} + \sum_{i=1}^{n} C_{ij}$ for all j

Resource Allocation Denial (Banker's Algorithm)

- Why Banker's algorithm is named so?
  - Banker's algorithm is named so because it is used in banking system to check whether loan can be sanctioned to a person or not.
  - The customers who wish to borrow money corresponds to processes and the money to be borrowed corresponds to resources.
  - Suppose there are **n** number of account holders in a bank and the total sum of their money is **S**. If a person applies for a loan then the bank first subtracts the loan amount from the total money that bank has and if the remaining amount is greater than **S** then only the loan is sanctioned. It is done because if all the account holders comes to withdraw their money then the bank can easily do it.
  - In other words, the bank would never allocate its money in such a way that it can no longer satisfy the needs of all its customers. The bank would try to be in safe state always.

Banker's Algorithm

- Consider a system with a fixed number of processes and a fixed number of resources.
- At any time a process may have zero or more resources allocated to it.
- The state of the system reflects the current allocation of resources to processes.
  - The state consists of the two vectors, Resource and Available, and the two matrices, Claim and Allocation.
  - A state of the system is called safe if the system can allocate all the resources requested by all the processes without entering into deadlock.
  - If the system cannot fulfill the request of all processes then the state of the system is called unsafe.

Banker's Algorithm (contd…)

- Banker's algorithm is a deadlock avoidance and resource allocation algorithm.
- When a process makes a request for a set of resources, assume that the request is granted, update the system state accordingly, and then determine if the result is a safe state.
- If so, grant the request and, if not, block the process until it is safe to grant the request.

Banker's Algorithm – Example 1

Claim matrix C

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Allocation matrix A

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

C – A

	R1	R2	R3
P1	2	2	2
P2	0	0	1
P3	1	0	3
P4	4	2	0

Resource vector R

R1	R2	R3
9	3	6

Available vector V

R1	R2	R3
0	1	1

Initial State

Banker's Algorithm – Example 1 (contd..)

Claim matrix C

	R1	R2	R3
P1	3	2	2
P2	0	0	0
P3	3	1	4
P4	4	2	2

Allocation matrix A

	R1	R2	R3
P1	1	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

C – A

	R1	R2	R3
P1	2	2	2
P2	0	0	0
P3	1	0	3
P4	4	2	0

Resource vector R

R1	R2	R3
9	3	6

Available vector V

R1	R2	R3
6	2	3

P2 runs to completion

Banker's Algorithm – Example 1 (contd..)

Claim matrix C

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	3	1	4
P4	4	2	2

Allocation matrix A

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

C – A

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	1	0	3
P4	4	2	0

Resource vector R

R1	R2	R3
9	3	6

Available vector V

R1	R2	R3
7	2	3

P1 runs to completion

Banker's Algorithm – Example 1 (contd..)

Claim matrix C

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	2

Allocation matrix A

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	2

C – A

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	0

Resource vector R

R1	R2	R3
9	3	6

Available vector V

R1	R2	R3
9	3	4

P3 runs to completion, similarly P4 will also run to completion.

It is clear that the given state is safe state.

Banker's Algorithm – Example 2

Given the initial state, assume that P1 has an additional unit of R1 and R3.
Determine the state of the system.

Claim matrix C

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Allocation matrix A

	R1	R2	R3
P1	1	0	0
P2	5	1	1
P3	2	1	1
P4	0	0	2

C – A

	R1	R2	R3
P1	2	2	2
P2	1	0	2
P3	1	0	3
P4	4	2	0

Resource vector R

R1	R2	R3
9	3	6

Available vector V

R1	R2	R3
1	1	2

Initial State

If we assume that the request is granted, the state will be as shown in next slide.

Banker's Algorithm - Example 2 (contd..)

P1 requests one additional unit of R1 and R3

Claim matrix C

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Allocation matrix A

	R1	R2	R3
P1	2	0	1
P2	5	1	1
P3	2	1	1
P4	0	0	2

C – A

	R1	R2	R3
P1	1	2	1
P2	1	0	2
P3	1	0	3
P4	4	2	0

Resource vector R

R1	R2	R3
9	3	6

Available vector V

R1	R2	R3
0	1	1

Is this a safe state? The answer is no, because each process will need at least one additional unit of R1, and there are none available.

Thus, on the basis of deadlock avoidance, the request by P1 should be denied and P1 should be blocked.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.25

Deadlock Detection and Recovery

- In this approach, the operating system does not apply any mechanism to avoid or prevent the deadlocks.
- Therefore, the system considers that the deadlock will definitely occur.
- In order to get rid of deadlocks, the operating system periodically checks the system for any deadlock.
- In case, it finds any of the deadlock, then, the operating system will recover the system using some recovery techniques.
- The main task of the operating system is detecting the deadlocks. The operating system can detect the deadlocks with the help of Resource Allocation Graph.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.26

Deadlock Detection and Recovery (contd..)

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.27

Resource-Allocation Graph

- Deadlocks can be described more precisely in terms of a directed graph, called a system **Resource-Allocation Graph**:
  - This graph consists of a set of vertices **V** and a set of edges **E**.
  - The set of vertices V is partitioned into two different types of nodes:
    - $P = \{P_1, P_2, ..., P_n\}$, the set consisting of all the active processes in the system.
    - $R = \{R_1, R_2, ..., R_m\}$, the set consisting of all resource types in the system.
  - A directed edge from process P_i to resource type R_j is denoted by $P_i \rightarrow R_j$ **(request edge)**; it signifies that process P_i has requested an instance of resource type R_j and is currently waiting for that resource.
  - A directed edge from resource type R_j to process P_i is denoted by $R_j \rightarrow P_i$ **(assignment edge)**; it signifies that an instance of resource type R_j has been allocated to process P_i.

Resource-Allocation Graph (contd…)

- Each process P_i is represented as a circle and each resource type R_j is represented as a rectangle.
  - Resource type R_j may have more than one instance, each instance is represented as a dot within the rectangle.
  - A request edge points to only the rectangle R_j, whereas an assignment edge must also designate one of the dots in the rectangle.
- When process P_i requests an instance of resource type R_j, a request edge is inserted in the resource-allocation graph.
- When this request can be fulfilled, the request edge is instantaneously transformed to an assignment edge.
- When the process no longer needs access to the resource, it releases the resource; as a result, the assignment edge is deleted.

Resource-Allocation Graph (contd…)

Vertices
Process Vertex
Pi
Resources Vertex
Single Instance
Eg. CPU
Multiple Instance
Eg. Registers

Edges
Assign edge
Pi
Ri
Request edge
Pi
Ri

Resource-Allocation Graph: Example

- P = {P1, P2, P3}
- R = {R1, R2, R3, R4}
- E = {P1 → R1, P2 → R3, R1 → P2, R2 → P2, R2 → P1, R3 → P3}

R_1 R_3 P_1 P_2 P_3 R_2 R_4

Deadlock Detection using RAG

- If the graph contains no cycles, then no process in the system is deadlocked.
- If the graph does contain a cycle, then a deadlock may exist.
- If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred.
- If the cycle involves only a set of resource types, each of which has only a single instance, then a deadlock has occurred.
- Each process involved in the cycle is deadlocked.
- In this case, a cycle in the graph is both a necessary and a sufficient condition for the existence of deadlock.
- If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred. In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.

Deadlock Detection using RAG: Example

R_1 R_3 P_1 P_2 P_3 R_2 R_4

- **Cycle 1:** $P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$
- **Cycle 2:** $P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$
- Processes P_1, P_2, and P_3 are deadlocked.

Deadlock Detection using RAG: Example

- Cycle 1: $P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$
- There is no deadlock.
- The process P_4 may release its instance of resource type R_2. That resource can then be allocated to P_3, breaking the cycle.

In summary, if a resource-allocation graph does not have a cycle, then the system is not in a deadlocked state. If there is a cycle, then the system may or may not be in a deadlocked state.

R_1 P_2 P_3 P_1 R_2 P_4

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh

Deadlock Detection using RAG: Example

P1 P2 P3
R1 R2

P1 P2 P3
R1 R2 R3

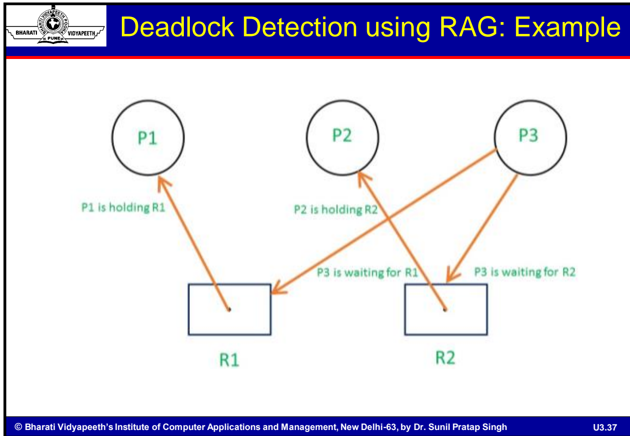
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh

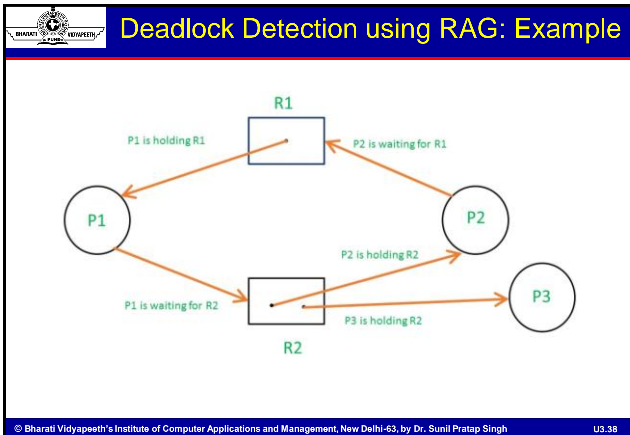
Deadlock Detection using RAG: Example

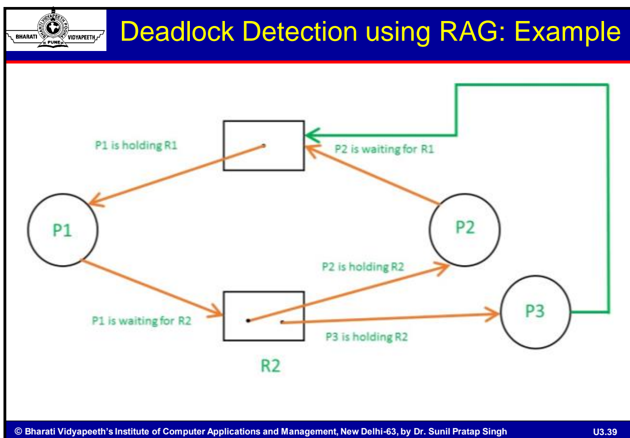
R1
P1 is holding R1
P2 is waiting for R1
P1
P2
P1 is waiting for R2
P2 is holding R2
R2

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh


Deadlock Detection using RAG: Example
P1
P2
P3
P1 is holding R1
P2 is holding R2
P3 is waiting for R1
P3 is waiting for R2
R1
R2
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh
U3.37


Deadlock Detection using RAG: Example
R1
P1 is holding R1
P2 is waiting for R1
P1
P2
P2 is holding R2
P1 is waiting for R2
P3
P3 is holding R2
R2
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh
U3.38


Deadlock Detection using RAG: Example
P1 is holding R1
P2 is waiting for R1
P1
P2
P2 is holding R2
P1 is waiting for R2
P3
P3 is holding R2
R2
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh
U3.39

Deadlock Detection for Several Instance of Resource

- Let, an Allocation Matrix, a Request Matrix Q, and an Available Vector
- The algorithm proceeds by marking processes that are not deadlocked. Initially, all processes are unmarked. Then the following steps are performed:

1. Mark each process that has a row in the Allocation matrix of all zeros.
2. Initialize a temporary vector **W** to equal the Available vector.
3. Find an index i such that process i is currently unmarked and the ith row of **Q** is less than or equal to **W**. That is, $Q_{ik} \le W_k$, for $1 \le k \le m$. If no such row is found, terminate the algorithm.
4. If such a row is found, mark process i and add the corresponding row of the allocation matrix to **W**. That is, set $W_k = W_k + A_{ik}$, for $1 \le k \le m$. Return to step 3.

- A deadlock exists if and only if there are unmarked processes at the end of the algorithm. Each unmarked process is deadlocked.

Deadlock Detection - Example

	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
P3	0	0	0	0	1
P4	1	0	1	0	1

Request matrix Q

	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

Allocation matrix A

R1	R2	R3	R4	R5
2	1	1	2	1

Resource vector

R1	R2	R3	R4	R5
0	0	0	0	1

Available vector

- Mark P4, because P4 has no allocated resources.
- Set W = (0 0 0 0 1).
- The request of process P3 is less than or equal to W, so mark P3 and set W = W + (0 0 0 1 0) = (0 0 0 1 1).
- No other unmarked process has a row in Q that is less than or equal to W. Therefore, terminate the algorithm.
- The algorithm concludes with P1 and P2 unmarked, indicating that these processes are deadlocked.

Deadlock Recovery

- To recover deadlock, the operating system examines either resources or processes.
- For Process:
  - Kill a Process: In this approach, kill the process due to which deadlock occurred. But the selection of the process to kill is a tough task. In this, the operating system mainly kills that process, which does not work more till now.
  - Kill all Process: Kill all the processes is not a suitable approach. We can use this approach when the problem becomes critical. By killing all the processes, the system efficiency will be decreased, and we have to execute all the processes further from the start.

Deadlock Recovery (contd…)

- For Resources:
  - Preempt the Resource: In this, we take the resource from one process to the process that needs it to finish its execution, and after the execution is completed, the process soon releases the resource. In this, the resource selection is difficult, and the snatching of the resource is also difficult.
  - Rollback to a Safe State: To enter into the deadlock, the system goes through several states. In this, the operating system can easily roll back the system to the earlier safe state. To do so, we require to implement checkpoints at every state. At the time when we detect deadlock, then we need to rollback every allocation so that we can enter into the earlier safe state.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh

Background: Clock Cycle

- CPU speed is determined by the clock cycle.
- The clock cycle is the amount of time between two pulses of an oscillator.
- The clock speed is measured in Hz, often either megahertz (MHz) or gigahertz (GHz).
  - For example, a 4 GHz processor performs 4,000,000,000 clock cycles per second.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh

Background: CPU Cycle

- **CPU cycle** is also called **machine cycle**.
- **CPU cycle refers to the** time required for the execution of one simple **processor operation** such as an addition.
  - For an instruction cycle, we fetch an instruction and execute it, at least two CPU cycles are required. At least one CPU cycle is required to fetch instructions, and at least one CPU cycle is required to execute them. Complex instructions require more CPU cycles.
- An instruction cycle may include multiple CPU cycles, and a CPU cycle may include multiple clock cycles.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh

Background: Instruction Cycle

- Instruction cycle refers to the time taken to execute an instruction. It is the basic operational process of a computer. This process is repeated continuously by CPU from boot up to shut down of computer.
- The execution process of instructions is divided into the following steps:
  - **Fetch** - The instruction is fetched from memory address that is stored in PC (Program Counter) and stored in the Instruction Register IR.
  - **Decode** - According to the instructions in the instruction register, decode what kind of operation is to be parsed.
  - **Execute** - Run the corresponding instructions to perform arithmetic and logic operations, data transmission, etc.

Instruction Cycle, Machine Cycle and Clock Cycle

Background: Memory

- Main memory and the registers built into the processor itself are the only storage that the CPU can access directly.
- There are machine instructions that take memory addresses as arguments, but none take disk addresses.
- Therefore, any instructions in execution, and any data being used by the instructions, must be in one of these direct-access storage devices.
- If the data are not in memory, they must be moved there before the CPU can operate on them.
- Registers that are built into the CPU are generally accessible within one clock cycle.
- Most CPUs can decode instructions and perform simple operations on register contents at the rate of one or more operations per clock tick.
  - The same cannot be said of main memory, which is accessed via a transaction on the memory bus.

Background: Memory (contd…)

- Completing a memory access may take many clock cycles.
- In such cases, the processor normally needs to stall, since it does not have the data required to complete the instruction that it is executing.
- This situation is intolerable because of the frequency of memory accesses.
- The remedy is to add fast memory between the CPU and main memory.
- A memory buffer used to accommodate a speed differential, is called cache.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.49

Basic Hardware

- There are several hardware-level approaches to protect the operating system from access by user processes and, in addition, to protect user processes from one another.
- To make sure that each process has a separate memory space, the protection is provided by using two registers: **Base Register** and **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).
  - Protection of memory space is accomplished by having the CPU hardware compare every address generated in user mode with the registers.
  - Any attempt by a program executing in user mode, to access operating-system memory or other users' memory, results in a trap to the operating system, which treats the attempt as **a fatal error** (an error that causes a program to terminate without any warning or saving its state).
    - This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the operating system or other users.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.50

Hardware Address Protection

- The base and limit registers can be loaded only by the operating system, which uses a special privileged instruction.

Base and Limit Register defining a logical address space

Hardware address protection with Base and Limit Registers

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.51

Basic Concepts

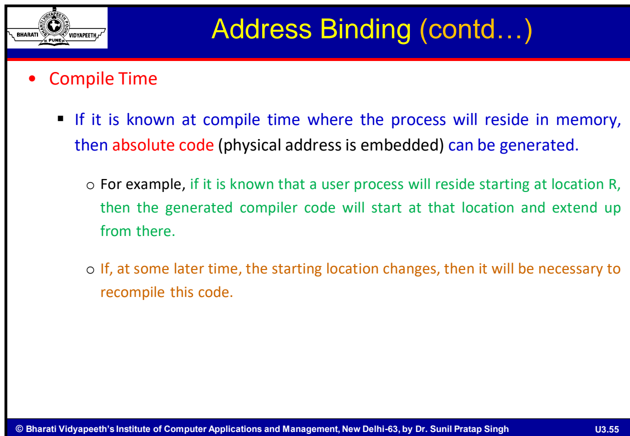
- Address Binding
  - Compile Time Binding
  - Load Time Binding
  - Execution Time Binding
- Logical Address and Physical Address
- Linking
  - Static Linking
  - Dynamic Linking
- Loading
  - Static Loading
  - Dynamic Loading
- Swapping

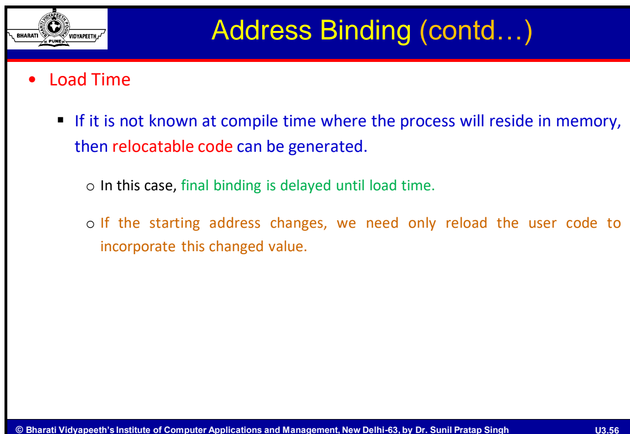
Address Binding

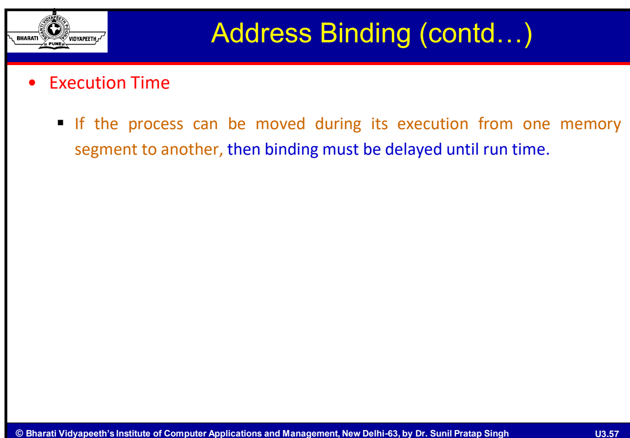
- The process (program) may be moved between disk and memory during its execution.
- The processes on the disk that are waiting to be brought into memory for execution **form** the input queue.
- The normal procedure is to select one of the processes in the input queue and to load that process into memory.
- In most cases, a user program go through several steps (such as compiling, loading, execution) before being executed.
  - Addresses may be represented in different ways during these steps.
  - Addresses in the source program are generally **symbolic** (such as count).

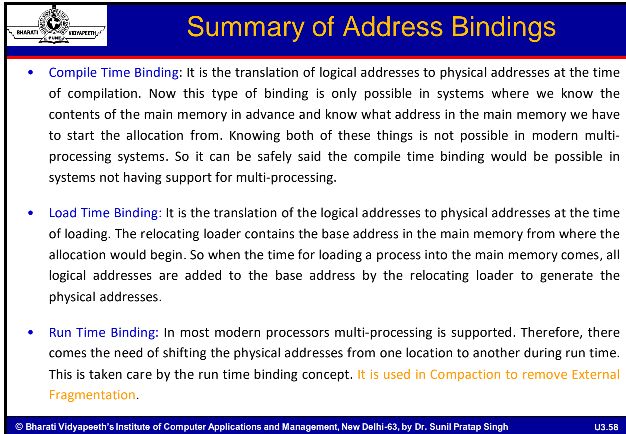
Address Binding (contd…)

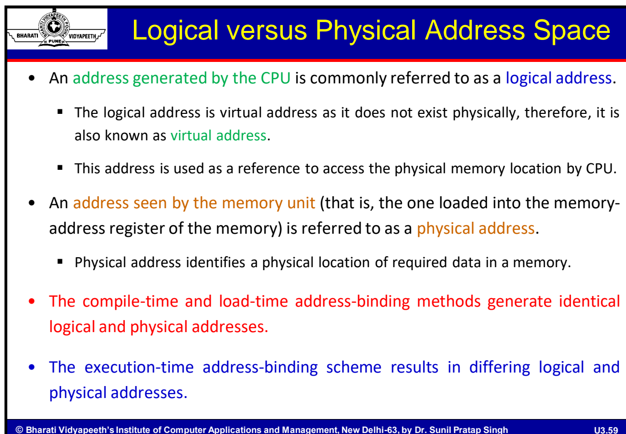
- The **binding** of instructions and data **to memory addresses** can be done at any step along the way:
  - Compile Time
  - Load Time
  - Execution (Run) Time

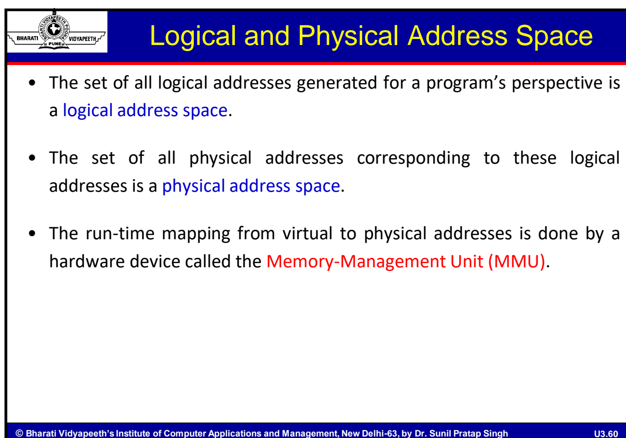

BHARATI VIDYAPEETH PUNE
Address Binding (contd…)
Compile Time
If it is known at compile time where the process will reside in memory, then absolute code (physical address is embedded) can be generated.
For example, if it is known that a user process will reside starting at location R, then the generated compiler code will start at that location and extend up from there.
If, at some later time, the starting location changes, then it will be necessary to recompile this code.
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh
U3.55


BHARATI VIDYAPEETH PUNE
Address Binding (contd…)
Load Time
If it is not known at compile time where the process will reside in memory, then relocatable code can be generated.
In this case, final binding is delayed until load time.
If the starting address changes, we need only reload the user code to incorporate this changed value.
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh
U3.56


BHARATI VIDYAPEETH PUNE
Address Binding (contd…)
Execution Time
If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time.
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh
U3.57


BHARATI VIDYAPEETH PUNE
Summary of Address Bindings
• Compile Time Binding: It is the translation of logical addresses to physical addresses at the time of compilation. Now this type of binding is only possible in systems where we know the contents of the main memory in advance and know what address in the main memory we have to start the allocation from. Knowing both of these things is not possible in modern multi-processing systems. So it can be safely said the compile time binding would be possible in systems not having support for multi-processing.
• Load Time Binding: It is the translation of the logical addresses to physical addresses at the time of loading. The relocating loader contains the base address in the main memory from where the allocation would begin. So when the time for loading a process into the main memory comes, all logical addresses are added to the base address by the relocating loader to generate the physical addresses.
• Run Time Binding: In most modern processors multi-processing is supported. Therefore, there comes the need of shifting the physical addresses from one location to another during run time. This is taken care by the run time binding concept. It is used in Compaction to remove External Fragmentation.
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh
U3.58


BHARATI VIDYAPEETH PUNE
Logical versus Physical Address Space
• An address generated by the CPU is commonly referred to as a logical address.
▪ The logical address is virtual address as it does not exist physically, therefore, it is also known as virtual address.
▪ This address is used as a reference to access the physical memory location by CPU.
• An address seen by the memory unit (that is, the one loaded into the memory-address register of the memory) is referred to as a physical address.
▪ Physical address identifies a physical location of required data in a memory.
• The compile-time and load-time address-binding methods generate identical logical and physical addresses.
• The execution-time address-binding scheme results in differing logical and physical addresses.
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh
U3.59


BHARATI VIDYAPEETH PUNE
Logical and Physical Address Space
• The set of all logical addresses generated for a program's perspective 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).
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh
U3.60

Mapping from Virtual to Physical Addresses

relocation register
14000
CPU
logical address
346
+
physical address
14346
memory
MMU

Example: Simple MMU scheme (a generalization of the base-register scheme)

- CPU generates logical address 346.
- MMU generates relocation register (base register) 14000.
- In memory, the physical address is 14346 (346+14000).

Linking

- Linking intends to generate an executable module of a program by combining the object codes generated by the compiler or assembler.
- Linking is the process of connecting all the modules or the function of a program for program execution.
- The linker, also known as the link editor, takes object modules from the assembler and forms an executable file for the loader.
- Linking is classified into two types, based on the time when it is done:
  - Static Linking
  - Dynamic Linking

Linking (contd…)

- In the static linking, each program binds to its dependent libraries at compile time.
  - With static linking, the user ends up copying functions or routines that are repetitive across various executables.
  - For example: Nearly every program needs printf() function. Thus, a copy of it is present in all executables which wastes space.
- In the case of dynamic linking, programs use shared libraries, and these libraries are linked to the programs at run time.

Static Linking
Program A
Static Libraries (*.a)
Program B
Static Libraries (*.b)
Static Linking at compile -time
Dynamic Linking
Program A
Program B
Shared libraries (*.so)
Dynamic Linking at run-time

Dynamic Linking

- In the dynamic linking approach, the linker does not copy the routines into the executables. It takes note that the program has a dependency on the library.
- With dynamic linking, a stub is included in the image for each library routine reference.
  - The stub is a small piece of code that indicates how to locate the appropriate memory-resident library routine or how to load the library if the routine is not already present.
  - **When the stub is executed, it checks to see whether the needed routine is already in memory.** If it is not, the program loads the routine into memory.
  - Under this scheme, all processes that use a language library execute only one copy of the library code.
  - This feature can be extended to library updates (such as bug fixes). A library may be replaced by a new version, and all programs that reference the library will automatically use the new version.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.64

Loading

- Loading is the process of loading the program from secondary memory to the main memory for execution.
- It is necessary for the entire program and all data of a process to be in physical memory for the process to execute.
- The size of a process has thus been limited to the size of physical memory.
- To obtain better memory-space utilization, we can use dynamic loading.
  - Dynamic loading is the technique through which a computer program, at runtime, load a library into memory, retrieve the variable and function addresses, executes the functions, and unloads the program from memory.
  - It is often used to implement software plugins.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.65

Dynamic Loading

- With dynamic loading, a routine is not loaded until it is called.
- All routines are kept on disk in a relocatable load format.
- The main program is loaded into memory and is executed.
- When a routine needs to call another routine, the calling routine first checks to see whether the other routine has been loaded.
- If it has not, the relocatable linking loader is called to load the desired routine into memory and to update the program's address tables to reflect this change. Then control is passed to the newly loaded routine.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.66

Dynamic Loading (contd…)

- Advantage of Dynamic Loading:
  - An unused routine is never loaded.
  - This method is particularly useful when large amounts of code are needed to handle infrequently occurring cases, such as error routines.
  - In this case, although the total program size may be large, the portion that is used (and hence loaded) may be much smaller.
- Dynamic loading does not require special support from the operating system.
- It is the responsibility of the users to design their programs to take advantage of such a method.
- Operating systems may help the programmer, however, by providing library routines to implement dynamic loading. Loading Examples of Java:

Class.forName (String className); //Dynamic Loading TestClass tc = new TestClass(); //Static Loading

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.67

Swapping

- **A process must be in memory to be executed.**
- A process can be swapped temporarily out of memory to a backing store and then brought back into memory for continued execution.
- **Example**: Assume a multiprogramming environment with a round-robin CPU-scheduling algorithm. When a quantum expires, the memory manager will start to swap out the process that just finished and to swap another process into the memory space that has been freed.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.68

Swapping (contd…)

- A variant of swapping policy is used for priority-based scheduling algorithms.
- If a higher-priority process arrives and wants service, the memory manager can swap out the lower-priority process and then load and execute the higher-priority process.
- When the higher-priority process finishes, the lower-priority process can be swapped back in and continued.
- This variant of swapping is sometimes called roll out, roll in.
- Normally, a process that is swapped out will be swapped back into the same memory space it occupied previously.
- This restriction is dictated by the method of address binding. If binding is done at assembly or load time, then the process cannot be easily moved to a different location. If execution-time binding is being used, then a process can be swapped into a different memory space, because the physical addresses are computed during execution time.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.69

Swapping (contd…)

- Swapping requires a backing store (a fast disk, large enough to accommodate copies of all memory images, 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 memory and are ready to run.
- Whenever the CPU scheduler decides to execute a process, it calls the dispatcher.
- The dispatcher checks to see whether the next process in the queue is in memory.
- If it is not, and if there is no free memory region, the dispatcher swaps out a process currently in memory and swaps in the desired.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.70

Memory Management Techniques

- There are two techniques for memory management:
  - Contiguous Memory Allocation
  - Non-Contiguous Memory Allocation
- In Contiguous Memory Allocation, the process must be loaded entirely in main-memory at contiguous locations.
- In Non-Contiguous Memory Allocation, the process is loaded in several memory blocks at different memory locations in the memory.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.71

Contiguous Memory Allocation

- A single contiguous section/part of memory is allocated to a process or file needing it.

Process

Memory blocks

Contiguous Memory Allocation

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.72

Non-Contiguous Memory Allocation

- The process is loaded in several memory blocks at different memory locations in the memory.

Process

Memory blocks

Noncontiguous Memory Allocation

Contiguous Memory Allocation

- In contiguous memory allocation, each process is contained in a single contiguous block of memory.
- Discuss -
  - Memory Mapping and Protection
  - Partitioning
  - Allocation Policies
  - Performance Parameters
    - Fragmentation
    - Maximum Process Size
    - Degree of Multiprogramming
    - Allocation Policy

Memory Mapping and Protection

- These features can be provided by using a relocation register together with a limit register.
  - 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).

limit register
relocation register
CPU
logical address
<
yes
+
physical address
memory
no
trap: addressing error

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

Memory Allocation

- Dividing memory into several partitions is one of the simplest methods for allocating memory.
  - Each partition may contain exactly one process.
  - 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.
- Partitioning can be done in two ways:
  - Fixed (Static) Partitioning
  - Variable (Dynamic) Partitioning

Memory Allocation Policies

- The memory blocks available comprise a set of holes of various sizes scattered throughout memory. When a process arrives and needs memory, the system can search an appropriate hole with following policies:
  - 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.
  - 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.
  - Next-Fit - It is similar to the first fit but it will search for the first sufficient partition from the last allocation point.

Memory Allocation Policies: First Fit

PROCESS A = 25 KB

OS
20 KB
USED
15 KB
USED
40 KB
USED
60 KB
USED
25 KB

HOLE=40-25=15 KB

Memory Allocation Policies: Best-Fit

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.79

Memory Allocation Policies: Worst-Fit

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.80

Fragmentation

- Internal Fragmentation
  - Memory block allocated is slightly larger than request memory, therefore, some portion of memory is left unused, as it cannot be used by another process.
  - The internal fragmentation can be reduced by effectively assigning the smallest partition but large enough for the process.
- External Fragmentation
  - Total memory space exists to satisfy a request, but it is not contiguous.
  - The external fragmentation may be reduced by compaction (also known as defragmentation) – shuffle memory contents to place all free memory together in one large block.
  - **Compaction is possible only if relocation is dynamic, and is done at execution time.**

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.81

Fragmentation

Compaction

- If processes are relocatable, the used memory blocked may be moved together to make a larger free memory block.

Fixed Partitioning

- In this partitioning, number of partitions in memory 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.
- Sum of internal fragmentation in every block = (4-1)+(8-7)+(8-7)+(16-14) = 3 + 1 + 1 + 2 = 7MB.
- Suppose a process P5 of size 7MB comes. But, this process cannot be accommodated inspite of available free space because of contiguous allocation. Hence, 7MB becomes part of External Fragmentation.

Advantages/Disadvantages of Fixed Partitioning

- Advantages
  - Easy to implement
  - Little OS overhead
- Disadvantages
  - Internal fragmentation
  - External fragmentation
  - Limit process size - Process of size greater than size of partition in Main Memory cannot be accommodated.
  - Limitation on degree of multiprogramming
- Best Allocation Policy
  - Best-Fit

Variable Partitioning

- This partitioning tries to overcome the problems caused by fixed partitioning.
- In this technique, the partition size is not declared initially.
- It is declared at the time of process loading.
- The partition size varies according to the need of the process so that the internal fragmentation can be avoided.
- The size of each partition will be equal to the size of the process.

Size	Process	Memory	Partition
		Operating System	partition for OS
5 MB	Process P1	Process P1 (5 MB)	partition 1
2 MB	Process P2	Process P2 (2 MB)	partition 2
3 MB	Process P3	Process P3 (3 MB)	partition 3
4 MB	Process P4	Process P4 (4 MB)	partition 4

Advantages/Disadvantages of Variable Partitioning

- Advantages
  - No internal fragmentation
  - No limitation on the size of the process
  - Degree of multiprogramming is dynamic
- Disadvantages
  - External fragmentation
- Best Allocation Policy
  - Worst-Fit

Advantages/Disadvantages of Variable Partitioning

- External Fragmentation
  - Suppose, Process P1 (2MB) and process P3 (1MB) completed their execution. Hence two spaces are left, i.e. 2MB and 1MB.
  - Let's suppose process P5 of size 3MB comes. The empty space in memory cannot be allocated as no spanning is allowed in contiguous allocation.
  - The rule says that process must be contiguously present in main memory to get executed.
  - Hence, variable partitioning may result in external fragmentation.

Dynamic partitioning

Memory	Block size
Operating system	
P1 (2 MB) executed, now empty	Block size = 2 MB
P2 = 7 MB	Block size = 7 MB
P3 (1 MB) executed	Block size = 1 MB
P4 = 5 MB	Block size = 5 MB
Empty space of RAM	

Partition size = process size
So, no internal Fragmentation

Example - 1

Consider the following heap in which blank regions are not in use and shaded regions are in use.

50	150	300	350	600

The sequence of requests for blocks of size 300, 25, 125, 50 can be satisfied if we use:

a) either First-fit or Best-fit policy.
b) First-fit but not Best-fit
c) Best-fit but not First-fit
d) None of these

Example - 2

Consider a system in which the memory consists of the following free holes:

Sizes in Memory Order: 15K, 5K, 20K, 4K and 7K

Which hole is taken for successive segment request of 12K, 7K and 15K for Next-Fit. Assuming that the last process was swapped in just before the 5K hole.

a) 15K, 20K, Out of Memory
b) 15K, 7K, 20K
c) 20K, 7K, 15K
d) 15K, 7K, 20K

Paging

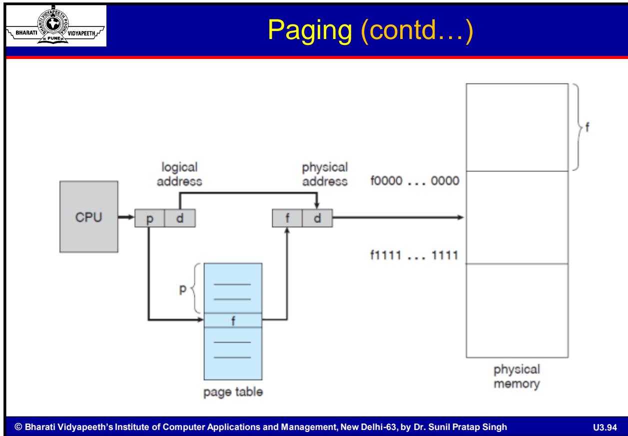
- Paging is a memory-management scheme that permits the physical address space of a process to be non-contiguous.
- Paging avoids external fragmentation and the need for compaction.
- Paging in its various forms is used in most operating systems.
- Implementation of paging involves dividing the process into blocks of the same size called pages which are mapped to same size blocks on physical memory called frames.
  - Physical Memory Blocks --> Frames
  - Logical Memory Blocks --> Pages

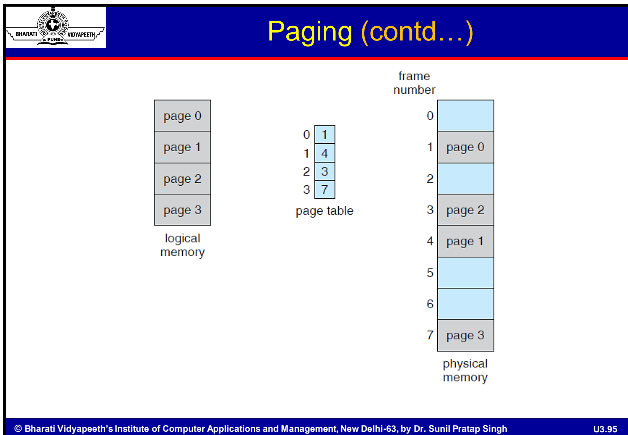
Paging (contd…)

- In every non-contiguous memory management technique, we need to consider the following points:
  - Organization of Logical Address Space (LAS)
  - Organization of Physical Address Space (PAS)
  - Organization of Memory Management Unit (MMU)
  - Translation Algorithm

Paging (contd…)

- **Address generated by CPU is divided into:**
  - Page number (p): Number of bits required to represent the pages in Logical Address Space.
  - Page offset (d): Number of bits required to represent particular word in a page or word number of a page.
- **Physical Address is divided into:**
  - Frame number (f): Number of bits required to represent the frame of Physical Address Space.
  - Frame offset (d): Number of bits required to represent particular word in a frame or word number of a frame.





Paging (contd…)

Let, LAS = 8 KB, PAS = 4 KB, PS (Page Size) = 1KB, 1Word = 1 Byte

Note: LAS = 2^{LA} and PAS = 2^{PA}, LAS & PAS is in words and LA & PA is in bits.

Organization of Logical Address Space (LAS)

- LAS is divided into equal size pages.
- Page size in power of 2 --> (2^k).
- Number of pages (N) = $\frac{\text{LAS}}{\text{PS}} = \frac{8\text{KB}}{1\text{KB}} = 8$
- Page number (p) = $\lceil \log_2 N \rceil$=3 (To represent 8 pages, we need 3 bit number)
- Page offset (To point a particular word in a page) (d) = $\lceil \log_2 \text{PS} \rceil = 10$
- Therefore, LA is divided into **p + d** ---> **3 + 10 = 13**

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.96

Paging (contd…)

Let, LAS = 8 KB, PAS = 4 KB, PS (Page Size) = 1KB, 1Word = 1 Byte

Note: LAS = 2^{LA} and PAS = 2^{PA}, LAS & PAS is in words and LA & PA is in bits.

Organization of Physical Address Space (PAS)

- LAS is divided into equal size frames.
- Frame size = Page size
- Number of frames (M) = $\frac{PAS}{PS} = \frac{4KB}{1KB} = 4$
- Frame number (f) = $\lceil \log_2 M \rceil$=2 (To represent 4 frames, we need 2 bit number)
- Frame offset (To point a particular word in a frame) (d) = $\lceil \log_2 FS \rceil = 10$ bits

  (Page size and frame size are equal)
- Therefore, PA is divided into **f + d --->** 2 + 10 = **12**

Paging (contd…)

Let, LAS = 8 KB, PAS = 4 KB, PS (Page Size) = 1KB, 1Word = 1 Byte

Note: LAS = 2^{LA} and PAS = 2^{PA}, LAS & PAS is in words and LA & PA is in bits.

Organization of Memory Management Unit (MMU)

- In paging, it is **Page Table**.
- Number of entries in Page Table is equal to number of pages in LAS.
- Page table contains frame number and other bits (valid/invalid bit, protection bit, modified bit, etc.)
- **Each process has its page table.**
- Page table also resides in main memory.
- Page Table Size = N * e (entry size)

Paging (contd…)

Number of frames = Physical Address Space / Frame size = 4 K / 1 K = 4 = 2^2

Number of pages = Logical Address Space / Page size = 8 K / 1 K = 8 = 2^3

Logical Address

13 bit

3 10

CPU p d

Physical Address

12 bit

2 10

f d

page number

if want to access page number 3

0 1 2 3 4 5 6 7

$(2)_{10} = (10)_2$

Page Map Table (PMT) or Page table

frame number in binary

frame number

0 1 2 3

0

contains 2^{10} words

2^{10}-1

Physical Memory

Paging: Example - 1

- Consider a system supporting,
  - LA = 32 Bits,
  - PA = 27 Bits,
  - PS = 4KB,
  - Page Table Entry Size (e) = 3 Byte
- What is Page Table Size?

Page Table Size $= N \times e$

$N = \frac{LAS}{Page\ Size} = \frac{2^{LA}}{Page\ Size} = \frac{2^{32}}{2^{12}}$

Page Table Size $= \frac{2^{32}}{2^{12}} \times 3 = 2^{20} \times 3 = 1024 \times 1024 \times 3$ Bytes

= 31,45,728 Bytes → 3 MB

Paging: Example - 2

- Consider a system with 2K Pages, 512 Frames, Page Offset = 9 Bits, and e = 4 Bytes. Determine p, f, LAS, PAS and Page Table Size.

Given, Number of Pages (N) = 2K = 2048 = 2^{11}

Number of Frames (M) = 512

Page Offset (d) = 9 Bits

Each Entry Size of Page Table (e) = 4 Bytes

$p = \lceil log_2 N \rceil = \lceil log_2 2048 \rceil = 11$

$f = \lceil log_2 M \rceil = \lceil log_2 512 \rceil = 9$

$LAS = 2^{LA} = 2^{(p+d)} = 2^{(11+9)} = 2^{20}$

$PAS = 2^{PA} = 2^{(f+d)} = 2^{(9+9)} = 2^{18}$

$Page\ Table\ Size = N \times e = 2^{11} \times 4 = 2048 \times 4$ Bytes $= 8192$ Bytes $= 8$KB

Paging: Example - 3

- Consider a system with **35 Bits LA**, **32 Bits PA**, **4KB Page Size**, and each Table Entry contains **2 Protection Bits**, **1 Valid/Invalid Bit**, and **1 Modified Bit** along with Frame Number. What is Page Table Size?

Size of Each Entry in Table

Bits to Represent **Frame No. (f)**	Protection Bits	Valid/Invalid Bit	Modified Bit

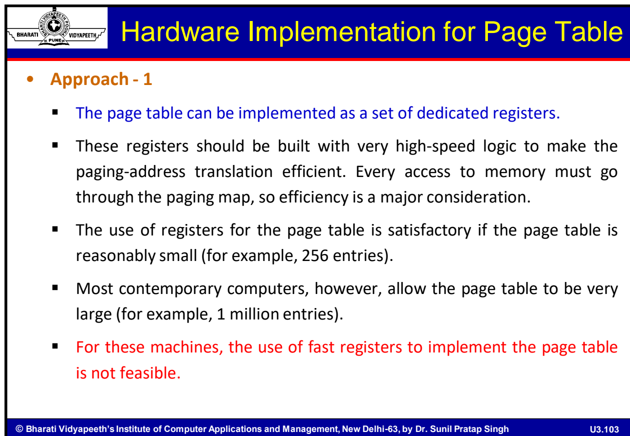
No. of Pages (**N**) $= \frac{LAS}{Page\ Size} = \frac{2^{LA}}{Page\ Size} = \frac{2^{35}}{2^{12}} = 2^{23}$

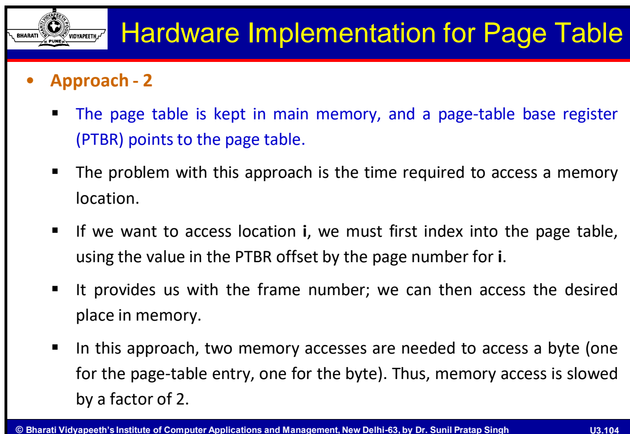
No. of Frames (**M**) $= \frac{PAS}{Frame\ Size} = \frac{2^{PA}}{Frame\ Size} = \frac{2^{32}}{2^{12}} = 2^{20}$

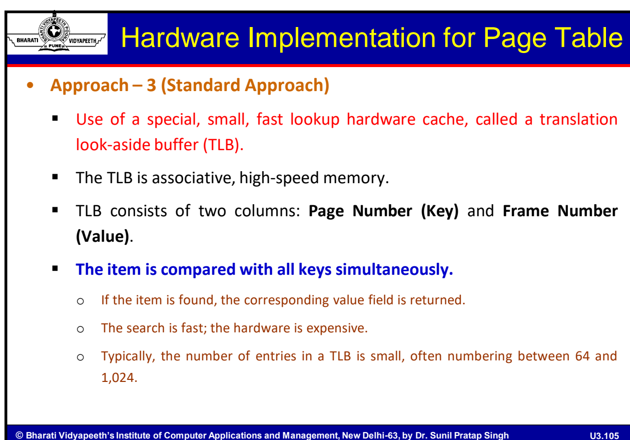
No. of Bits required to Represent Frame No. (**f**) $= \lceil log_2 M \rceil = \lceil log_2 2^{20} \rceil = 20$

Size of Each Entry in Table = 20 + 2 + 1 + 1 = 24 Bits

Page Table Size = $N \times e = 2^{23} \times 3$ Bytes $= 83{,}88{,}608 \times 3$ Bytes = **24 MB**


Hardware Implementation for Page Table
Approach - 1
The page table can be implemented as a set of dedicated registers.
These registers should be built with very high-speed logic to make the paging-address translation efficient. Every access to memory must go through the paging map, so efficiency is a major consideration.
The use of registers for the page table is satisfactory if the page table is reasonably small (for example, 256 entries).
Most contemporary computers, however, allow the page table to be very large (for example, 1 million entries).
For these machines, the use of fast registers to implement the page table is not feasible.
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh
U3.103

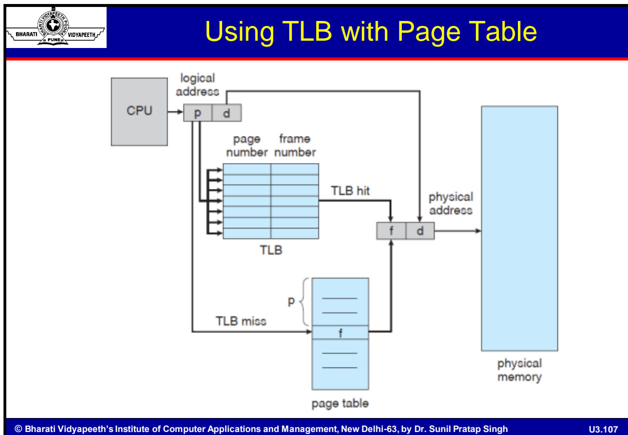

Hardware Implementation for Page Table
Approach - 2
The page table is kept in main memory, and a page-table base register (PTBR) points to the page table.
The problem with this approach is the time required to access a memory location.
If we want to access location i, we must first index into the page table, using the value in the PTBR offset by the page number for i.
It provides us with the frame number; we can then access the desired place in memory.
In this approach, two memory accesses are needed to access a byte (one for the page-table entry, one for the byte). Thus, memory access is slowed by a factor of 2.
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh
U3.104

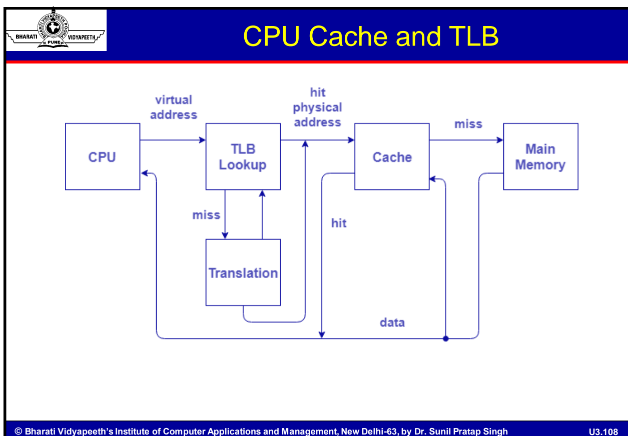

Hardware Implementation for Page Table
Approach – 3 (Standard Approach)
Use of a special, small, fast lookup hardware cache, called a translation look-aside buffer (TLB).
The TLB is associative, high-speed memory.
TLB consists of two columns: Page Number (Key) and Frame Number (Value).
The item is compared with all keys simultaneously.
If the item is found, the corresponding value field is returned.
The search is fast; the hardware is expensive.
Typically, the number of entries in a TLB is small, often numbering between 64 and 1,024.
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh
U3.105

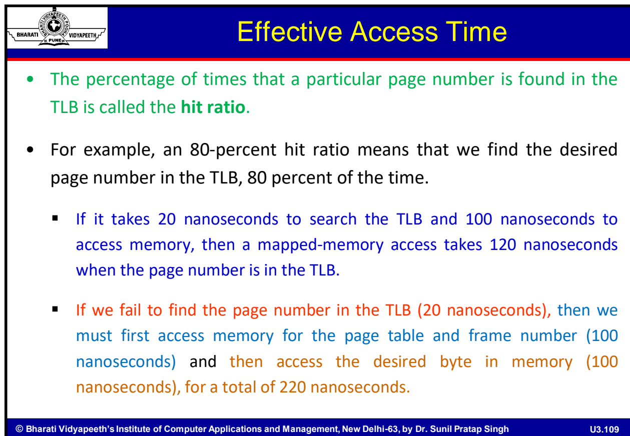
Using TLB with Page Table

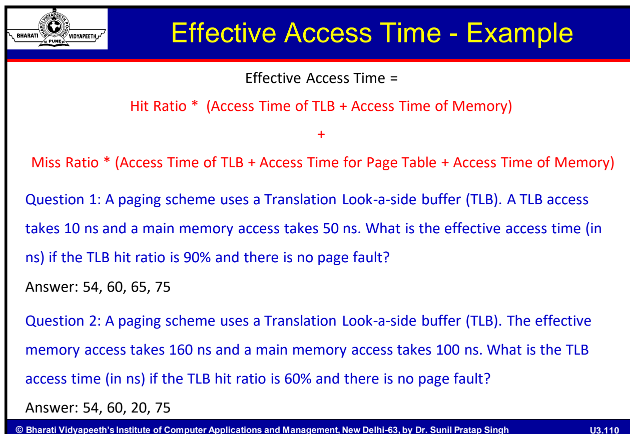
- The TLB contains only a few of the page-table entries.
- When a logical address is generated by the CPU, its page number is presented to the TLB.
  - If the page number is found, its frame number is immediately available and is used to access memory.
  - If the page number is not in the TLB **(known as a TLB miss)**, a memory reference to the page table must be made. In addition, we add the page number and frame number to the TLB, so that they will be found quickly on the next reference.
  - If the TLB is already full of entries, the operating system must select one for replacement. Replacement policies range from least recently used (LRU) to random. Furthermore, some TLBs allow certain entries to be wired down, meaning that they cannot be removed from the TLB. Typically, TLB entries for kernel code are wired down.

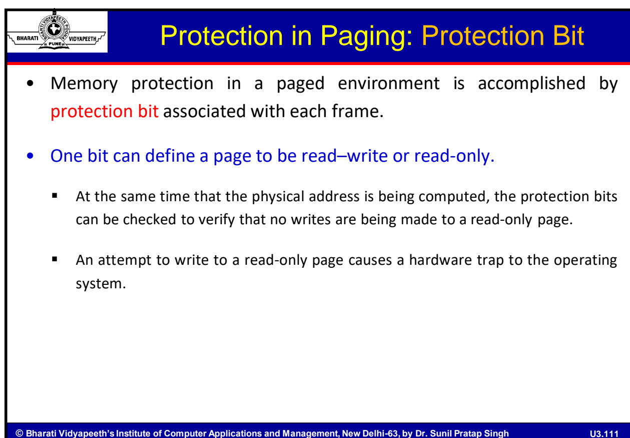
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.106






Effective Access Time
The percentage of times that a particular page number is found in the TLB is called the hit ratio.
For example, an 80-percent hit ratio means that we find the desired page number in the TLB, 80 percent of the time.
If it takes 20 nanoseconds to search the TLB and 100 nanoseconds to access memory, then a mapped-memory access takes 120 nanoseconds when the page number is in the TLB.
If we fail to find the page number in the TLB (20 nanoseconds), then we must first access memory for the page table and frame number (100 nanoseconds) and then access the desired byte in memory (100 nanoseconds), for a total of 220 nanoseconds.
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh
U3.109


Effective Access Time - Example
Effective Access Time =
Hit Ratio * (Access Time of TLB + Access Time of Memory)
+
Miss Ratio * (Access Time of TLB + Access Time for Page Table + Access Time of Memory)
Question 1: A paging scheme uses a Translation Look-a-side buffer (TLB). A TLB access takes 10 ns and a main memory access takes 50 ns. What is the effective access time (in ns) if the TLB hit ratio is 90% and there is no page fault?
Answer: 54, 60, 65, 75
Question 2: A paging scheme uses a Translation Look-a-side buffer (TLB). The effective memory access takes 160 ns and a main memory access takes 100 ns. What is the TLB access time (in ns) if the TLB hit ratio is 60% and there is no page fault?
Answer: 54, 60, 20, 75
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh
U3.110


Protection in Paging: Protection Bit
Memory protection in a paged environment is accomplished by protection bit associated with each frame.
One bit can define a page to be read–write or read-only.
At the same time that the physical address is being computed, the protection bits can be checked to verify that no writes are being made to a read-only page.
An attempt to write to a read-only page causes a hardware trap to the operating system.
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh
U3.111

Protection in Paging: Valid–Invalid Bit

- One additional bit is generally attached to each entry in the page table: a valid–invalid bit.
  - When this bit is set to "valid," the associated page is in the process's logical address space and is thus a legal (or valid) page.
  - When the bit is set to "invalid," the page is not in the process's logical address space.
  - The OS sets this bit for each page to allow or disallow access to the page.

Protection in Paging: Valid–Invalid Bit

Structure of Page Table

- Hierarchical (Multi-Level) Paging
- Hashed Page Tables
- Inverted Page Tables

Hierarchical (Multi-Level) Paging

- Consider a system with a 32-bit logical address space.
  - If the page size in such a system is 4 KB (2^{12}), then a page table may consist of up to 1 million entries $2^{32}/2^{12}$).
  - Assuming that each entry consists of 4 bytes, each process may need up to 4MB of physical address space for the page table alone.
  - If we would not want to allocate the page table contiguously in main memory, the solution is to divide the page table into smaller pieces.
    - One way is to use a two-level paging algorithm, in which the page table itself is also paged.

Hierarchical (Multi-Level) Paging

A two-level page-table scheme

- Consider a system with a 32-bit logical address space and 4KB page size:
  - A logical address is divided into a page number consisting of 20 bits and a page offset consisting of 12 bits. Because we page the page table, the page number is further divided into a 10-bit page number and a 10-bit page offset.
  - Thus, a logical address is as follows:

page number		page offset
p_1	p_2	d
10	10	12

Two-Level 32-Bit Paging Architecture

Address Translation Process

Hierarchical (Multi-Level) Paging

For a system with a 64-bit logical address space, is a two-level paging scheme appropriate?

- To illustrate this point, let us suppose that the page size in such a system is 4 KB (2^{12}).
- In this case, the page table consists of up to 2^{52} entries.
- If we use a two-level paging scheme, then the inner page tables can conveniently be one page long, or contain 2^{10} 4-byte entries.

outer page	inner page	offset
p_1	p_2	d
42	10	12

2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12

The outer page table is still 234 bytes in size.

The next step would be a four-level paging scheme,

Hashed Page Tables

- A common 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.
- Each entry in the hash table contains a linked list of elements that hash to the same location (to handle collisions).

Inverted Page Table

- An inverted page table has one entry for each real page (or frame) of memory.

Shared Pages

- An advantage of paging is the possibility of sharing common code, particularly in a time-sharing environment.
- Consider a system that supports 40 users, each of whom executes a text editor. If the text editor consists of 150 KB of code and 50 KB of data space, we need 8,000 KB to support the 40 users.
- In shared mode, only one copy of the editor need be kept in physical memory.
- Each user's page table maps onto the same physical copy of the editor, but data pages are mapped onto different frames.
- Thus, to support 40 users, we need only one copy of the editor (150 KB), plus 40 copies of the 50 KB of data space per user. The total space required is now 2,150 KB instead of 8,000 KB—a significant savings.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.121

Shared Pages (contd…)

Sharing of Code in a Paging Environment

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.122

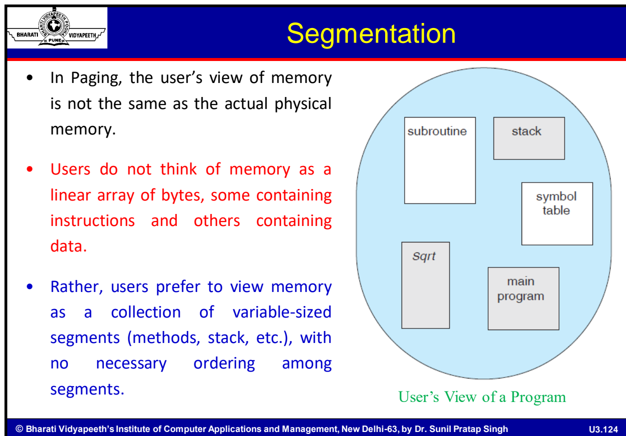
Limitations of Paging

- Internal Fragmentation
  - If the memory requirements of a process do not happen to coincide with page boundaries, the last frame allocated may not be completely full.
  - **Example:** If page size is 2048 bytes, a process of 72,766 bytes will need 35 pages plus 1,086 bytes. It will be allocated 36 frames, resulting in internal fragmentation of 2,048 − 1,086 = 962 bytes.
  - If process size is independent of page size, we expect internal fragmentation to average one-half page per process.
  - This consideration suggests that small page sizes are desirable. However, overhead is involved in each page-table entry, and this overhead is reduced as the size of the pages increases. Also, disk I/O is more efficient when the amount of data being transferred is larger.
- Larger Access Time
- Memory Required for Page Table

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.123

Segmentation

- In Paging, the user's view of memory is not the same as the actual physical memory.
- Users do not think of memory as a linear array of bytes, some containing instructions and others containing data.
- Rather, users prefer to view memory as a collection of variable-sized segments (methods, stack, etc.), with no necessary ordering among segments.



User's View of a Program

Segmentation (contd…)

- Segmentation is a memory-management scheme that supports user's view of memory.
- A logical address space is a collection of segments.
- Each segment has a name and a length.
- The addresses specify both the segment number and the offset within the segment.
  - 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, the fifth instruction of the Sqrt(), and so on.

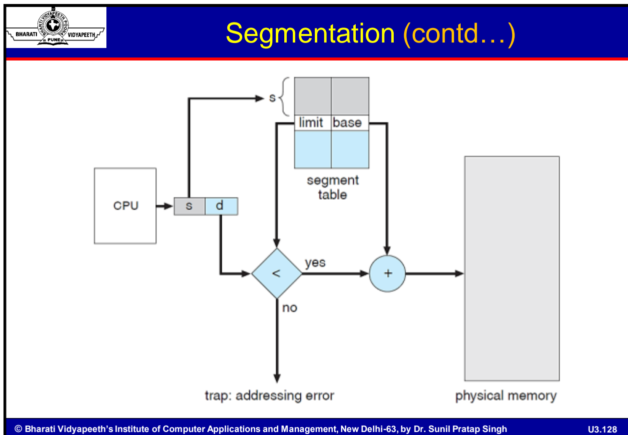
Segmentation (contd…)

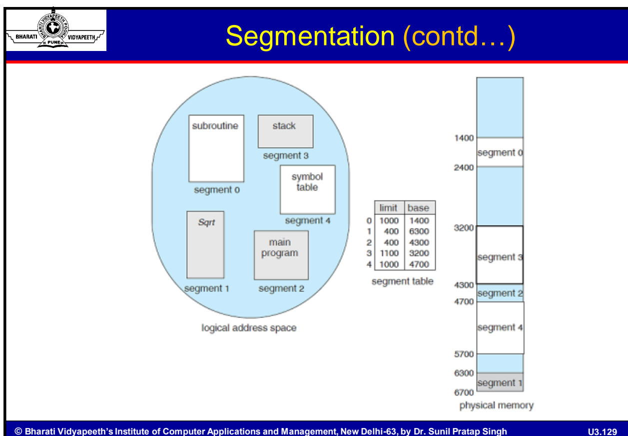
- While compiling, the compiler automatically constructs segments reflecting the input program.
- A 'C' compiler might create separate segments for the following:
  - The code
  - Global variables
  - The heap, from which memory is allocated
  - The stacks used by each thread
  - The standard C library
  - Libraries that are linked in during compile time
- The loader takes all these segments and assign them segment numbers.

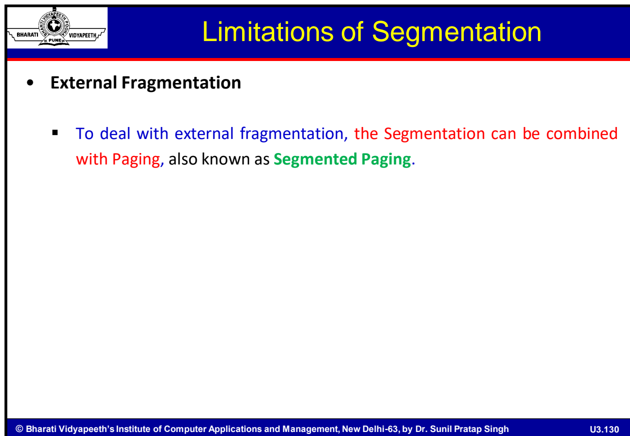
Segmentation (contd…)

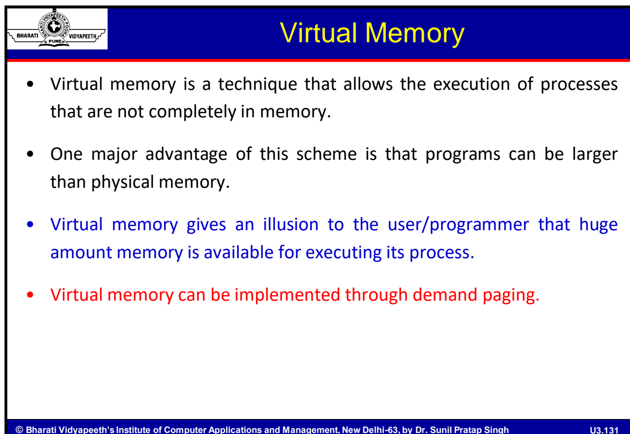
- Each entry in the segment table has a segment base and a segment limit.
  - The segment base contains the starting physical address where the segment resides in memory.
  - The segment limit specifies the length of the segment.
- A logical address consists of two parts:
  - A segment number **(s)**, 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, we trap to the operating system.

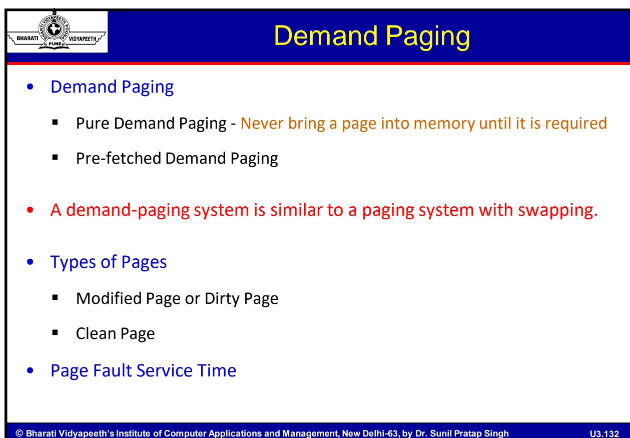
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.127






Limitations of Segmentation
External Fragmentation
To deal with external fragmentation, the Segmentation can be combined with Paging, also known as Segmented Paging.
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh
U3.130


Virtual Memory
Virtual memory is a technique that allows the execution of processes that are not completely in memory.
One major advantage of this scheme is that programs can be larger than physical memory.
Virtual memory gives an illusion to the user/programmer that huge amount memory is available for executing its process.
Virtual memory can be implemented through demand paging.
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh
U3.131

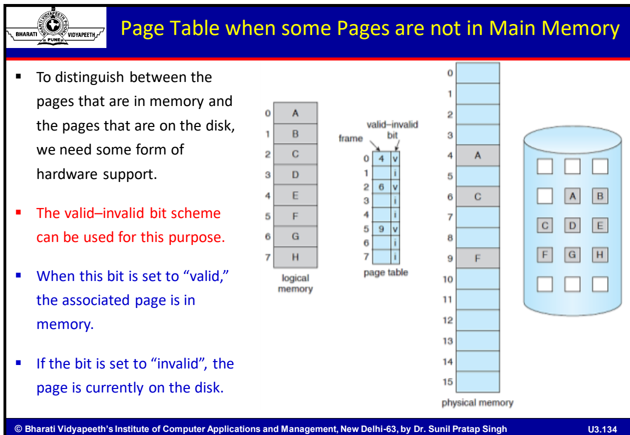

Demand Paging
Demand Paging
Pure Demand Paging - Never bring a page into memory until it is required
Pre-fetched Demand Paging
A demand-paging system is similar to a paging system with swapping.
Types of Pages
Modified Page or Dirty Page
Clean Page
Page Fault Service Time
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh
U3.132

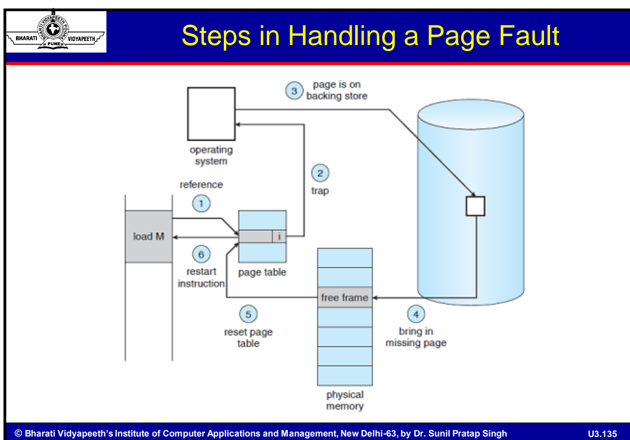
Hardware Support for Demand Paging

- Page Table
- Secondary Memory
  - This memory holds those pages that are not present in main memory.
  - The secondary memory is usually a high-speed disk.
  - It is known as the swap device, and the section of disk used for this purpose is known as swap space.

Note: To implement demand paging, we must develop a frame-allocation algorithm and a page-replacement algorithm.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.133





Performance of Virtual Memory

- Let
  - Main Memory Access Time (MMAT) = M
  - Page Fault Rate (PFR) = P
  - Page Fault Service Time (PFST) = S
  - Effective Memory Access Time (EMAT) = P*S+(1-P)*M
- If MMAT = 200 microsecond, PFR = 25%, PFST = 1 millisecond, then what is EMAT (in Microsecond)?

  EMAT = 0.25*1000 + 0.75*200 = 250 + 150 = 400 Microsecond

Performance of Virtual Memory

- If MMAT (M) = 1 microsecond, PFST (S) = 10 Millisecond, Hit Ratio = 99.99%, the what is EMAT (in Microsecond)?

  If Hit Ratio = 99.99, then P = 0.01%

  Therefore, EMAT = 0.0001*10000 + (1-0.0001)*1 =

  = 1 + 0.9999 = 1.9999 Microsecond

Page Replacement

1) Find the location of the desired page on the disk.
2) Find a free frame:
   a) If there is a free frame, use it.
   b) If there is no free frame, use a page-replacement algorithm to select a victim frame.
   c) Write the victim frame to the disk (If dirty); change the page and frame tables accordingly.
3) Bring the desired page into the (newly) freed frame; change the page and frame tables.
4) Continue the process by restarting the instruction that caused the trap.

Page Replacement (contd…)

frame valid–invalid bit
0 i
f v
page table
2 change to invalid
4 reset page table for new page
f victim
physical memory
1 swap out victim page
3 swap desired page in

Page Replacement Algorithms

- First-Come, First-Out
- Optimal Page Replacement
- Least Recently Used

We can evaluate an algorithm by running it on a particular string of memory references and computing the number of page faults.

- The string of memory references is called a reference string.

Page Replacement Algorithm: First-Come, First-Out

- We can create a FIFO queue to hold all pages in memory. We replace the page at the head of the queue. When a page is brought into memory, we insert it at the tail of the queue.

Q. Consider page reference string 1, 3, 0, 3, 5, 6, 3 with 3 page frames. Find number of page faults.

Page reference: 1, 3, 0, 3, 5, 6, 3

1	3	0	3	5	6	3
		0	0	0	0	3
	3	3	3	3	6	6
1	1	1	1	5	5	5
Miss	Miss	Miss	Hit	Miss	Miss	Miss

Total Page Fault = 6

Page Replacement Algorithm: First-Come, First-Out

Q. Consider page reference string 3, 2, 1, 0, 3, 2, 4, 3, 2, 1, 0, 4 with 3 page frames. Find number of page faults.

Q. Consider page reference string 3, 2, 1, 0, 3, 2, 4, 3, 2, 1, 0, 4 with 4 page frames. Find number of page faults.

Belady's Anomaly – For some page-replacement algorithms, the page-fault rate may increase as the number of allocated frames increases.

Page Replacement Algorithm: Optimal Page Replacement

Replace the page that will not be used for the longest period of time.

- Use of this page-replacement algorithm guarantees the lowest possible page fault rate for a fixed number of frames.
- Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string.
- The optimal algorithm is used mainly for comparison studies.

Page Replacement Algorithm: Optimal Page Replacement

Q. Consider the page references 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, with 4 page frame. Find number of page fault using **optimal page replacement algorithm**.

Page reference 7,0,1,2,0,3,0,4,2,3,0,3,2,3 No. of Page frame - 4

7	0	1	2	0	3	0	4	2	3	0	3	2	3
			2	2	2	2	2	2	2	2	2	2	2
		1	1	1	1	1	4	4	4	4	4	4	4
	0	0	0	0	0	0	0	0	0	0	0	0	0
7	7	7	7	7	3	3	3	3	3	3	3	3	3
Miss	Miss	Miss	Miss	Hit	Miss	Hit	Miss	Hit	Hit	Hit	Hit	Hit	Hit

Total Page Fault = 6

Page Replacement Algorithm: Least-Recently-Used (LRU)

Replace the page that has not been used for the longest period of time.

- LRU replacement associates with each page the time of that page's last use.
- When a page must be replaced, LRU chooses the page that has not been used for the longest period of time.
- This strategy can be considered as the optimal page-replacement algorithm looking backward in time, rather than forward.
- The LRU policy is often used as a page-replacement algorithm and is considered to be good.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.145

Page Replacement Algorithm: LRU Example

Q. Consider the page reference string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2 with 4 page frames. Find number of page faults using least-recently-used algorithm.

Page reference 7,0,1,2,0,3,0,4,2,3,0,3,2,3 No. of Page frame - 4

7	0	1	2	0	3	0	4	2	3	0	3	2	3
			2	2	2	2	2	2	2	2	2	2	2
		1	1	1	1	1	4	4	4	4	4	4	4
	0	0	0	0	0	0	0	0	0	0	0	0	0
7	7	7	7	7	3	3	3	3	3	3	3	3	3
Miss	Miss	Miss	Miss	Hit	Miss	Hit	Miss	Hit	Hit	Hit	Hit	Hit	Hit

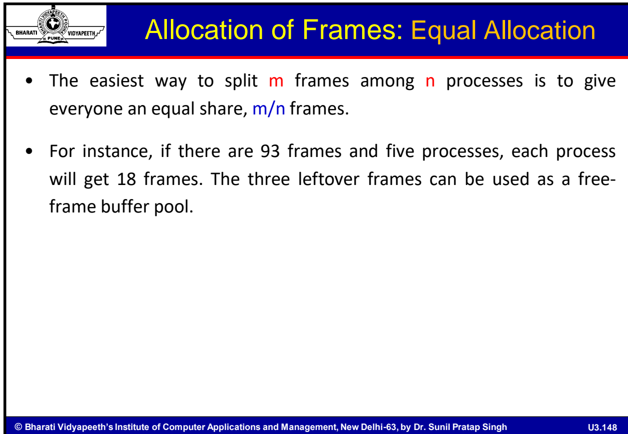
Total Page Fault = 6

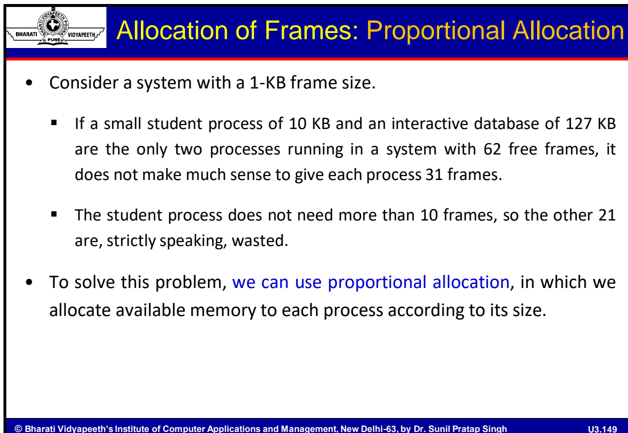
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.146

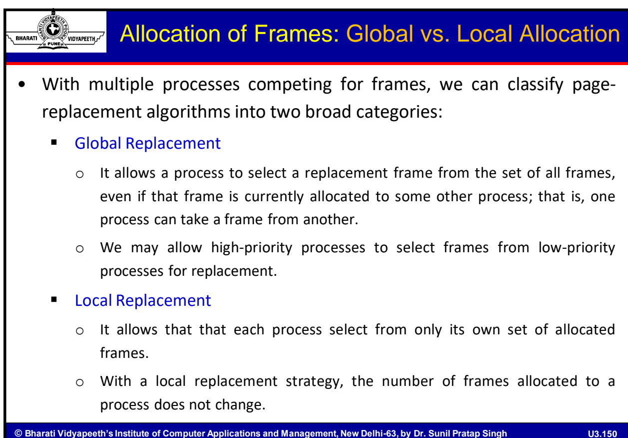
Allocation of Frames

- How do we allocate the fixed amount of free memory among the various processes?
  - The minimum number of frames per process is defined by the architecture.
  - The maximum number is defined by the amount of available physical memory.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.147


Allocation of Frames: Equal Allocation
The easiest way to split m frames among n processes is to give everyone an equal share, m/n frames.
For instance, if there are 93 frames and five processes, each process will get 18 frames. The three leftover frames can be used as a free-frame buffer pool.
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh
U3.148

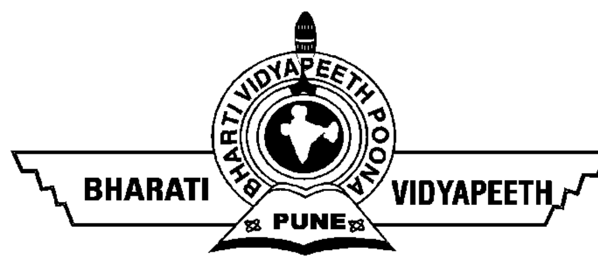

Allocation of Frames: Proportional Allocation
Consider a system with a 1-KB frame size.
If a small student process of 10 KB and an interactive database of 127 KB are the only two processes running in a system with 62 free frames, it does not make much sense to give each process 31 frames.
The student process does not need more than 10 frames, so the other 21 are, strictly speaking, wasted.
To solve this problem, we can use proportional allocation, in which we allocate available memory to each process according to its size.
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh
U3.149


Allocation of Frames: Global vs. Local Allocation
With multiple processes competing for frames, we can classify page-replacement algorithms into two broad categories:
Global Replacement
It allows a process to select a replacement frame from the set of all frames, even if that frame is currently allocated to some other process; that is, one process can take a frame from another.
We may allow high-priority processes to select frames from low-priority processes for replacement.
Local Replacement
It allows that that each process select from only its own set of allocated frames.
With a local replacement strategy, the number of frames allocated to a process does not change.
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh
U3.150

Thrashing

- If the process does not have the number of frames it needs to support pages in active use, it will quickly page-fault.
- At this point, it must replace some page.
- However, since all its pages are in active use, it must replace a page that will be needed again right away.
- Consequently, it quickly faults again, and again, and again, replacing pages that it must bring back in immediately.
- This high paging activity is called **thrashing**.
- A process is thrashing if it is spending more time in paging than executing.

Thrashing (contd…)



Operating Systems with Linux

(MCA-105)

Unit – 3

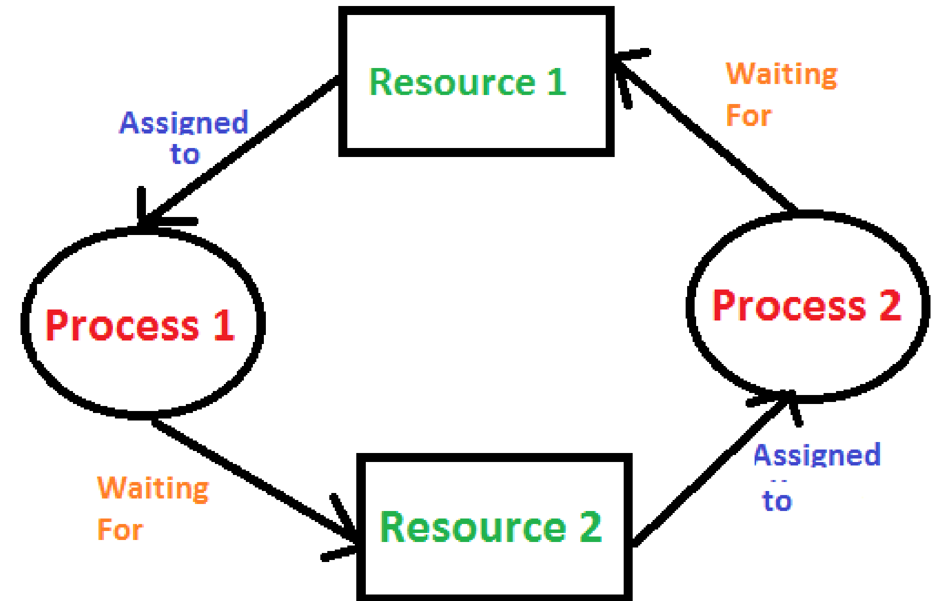
by

Dr. Sunil Pratap Singh
(Associate Professor, BVICAM, New Delhi)

2023

Deadlock

- In a multiprogramming environment, several processes may compete for a finite number of resources.
- A process requests resources; if the resources are not available at that time, the process enters a waiting state.
- Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes.
- This situation is called a deadlock.



Deadlock (contd…)

- Deadlock can be defined as the permanent blocking of a set of processes that either compete for system resources or communicate with each other.

- A set of processes is deadlocked when each process in the set is blocked awaiting an event (typically the freeing up of some requested resource) that can only be triggered by another blocked process in the set.

System Model

- A system consists of a finite number of resources to be distributed among a number of competing processes.
  - The resources are partitioned into several types, each consisting of some number of identical instances.
  - Memory space, CPU cycles, files, and I/O devices (such as printers and DVD drives) are examples of resource types.
  - If a system has two CPUs, then the resource type CPU has two instances.
  - Similarly, the resource type printer may have five instances.
  - If a process requests an instance of a resource type, the allocation of any instance of the type will satisfy the request.
  - If it will not, then the instances are not identical, and the resource type classes have not been defined properly.

System Model (contd…)

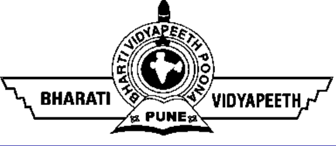
- A process may utilize a resource in only the following sequence:
  - **Request** - The process requests the resource. If the request cannot be granted immediately (for example, if the resource is being used by another process), then the requesting process must wait until it can acquire the resource.
  - **Use** - The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).
  - **Release** - The process releases the resource.

System Model (contd…)

- The **request** and **release** of resources are system calls.
  - Examples: request() and release() device, open() and close() file, and allocate() and free() memory works as system calls.
- Request and release of resources that are not managed by the operating system can be accomplished through the **wait()** and **signal()** operations on semaphores.
- The resources may be either physical resources (printers, tape drives, memory space, and CPU cycles) or logical resources (files, semaphores, and monitors).
- Deadlocks may involve same resource type (Printer) or different resource types (Printer and DVD drive).
- Multithreaded programs are good candidates for deadlock because multiple threads can compete for shared resources.

Deadlock Characterization

- Necessary and Sufficient Conditions - A deadlock situation arises if the following four conditions hold simultaneously in a system:
  - **Mutual Exclusion** - Only one process at a time can use the resource.
  - **Hold and Wait** - A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.
  - **No-Preemption** - Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.
  - **Circular Wait** - A set {P0, P1, ..., Pn} of waiting processes must exist such that P0 is waiting for a resource held by P1, P1 is waiting for a resource held by P2, ..., Pn−1 is waiting for a resource held by Pn, and Pn is waiting for a resource held by P0.



Methods for Handling Deadlocks

- Deadlock Ignorance
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection and Recovery

Deadlock Ignorance

- We can ignore the problem altogether and pretend that deadlocks never occur in the system.

- This approach is used by most operating systems, including UNIX and Windows; it is then up to the application developer to write programs that handle deadlocks.

  - There is always a tradeoff between Correctness and Performance. The operating systems like Windows and Linux mainly focus upon performance. However, the performance of the system decreases if it uses deadlock handling mechanism all the time if deadlock happens 1 out of 100 times then it is completely unnecessary to use the deadlock handling mechanism all the time.

Deadlock Prevention

- If we can be able to violate one of the four necessary conditions and don't let them occur together then we can prevent the deadlock.
  - Mutual Exclusion
    - **If we can be able to violate resources behaving in the mutually exclusive manner then the deadlock can be prevented.**
    - For a device like printer, **Spooling can work** - a memory is associated with the printer which stores jobs from each of the process into it. The printer collects all the jobs and print each one of them according to FCFS. By using this mechanism, the process doesn't have to wait for the printer and it can continue whatever it was doing.
    - Although, **Spooling** can be an effective approach to violate mutual exclusion but it suffers from two kinds of problems: **(1)** This cannot be applied to every resource, and **(2)** After some point of time, there may arise a race condition between the processes to get space in that spool.
    - We cannot force a resource to be used by more than one process at the same time. Therefore, we cannot violate mutual exclusion for a process practically.

Deadlock Prevention (contd…)

- Hold and Wait

  - **The hold-and-wait condition can be prevented by requiring that a process request all of its required resources at one time and blocking the process until all requests can be granted simultaneously.**

  - This approach is inefficient in two ways:

    1) First, a process may be held up for a long time waiting for all of its resource requests to be filled, when in fact it could have proceeded with only some of the resources.

    2) Resources allocated to a process may remain unused for a considerable period, during which time they are denied to other processes.

  - Another problem is that a process may not know in advance all of the resources that it will require.

Deadlock Prevention (contd…)

- No-Preemption
  - **If a process holding certain resources is denied a further request, that process must release its original resources and, if necessary, request them again together with the additional resource.**
  - **If a process requests a resource that is currently held by another process, the operating system may preempt the second process and require it to release its resources.**
  - This approach is practical only when applied to resources whose state can be easily saved and restored later, as is the case with a processor.

Deadlock Prevention (contd…)

- Circular Wait
  - **The circular-wait condition can be prevented by defining a linear ordering of resource types.**
  - If a process has been allocated resources of type R, then it may subsequently request only those resources of types following R in the ordering.
  - This approach may be inefficient, slowing down processes and denying resource access unnecessarily.

Deadlock Avoidance

- Deadlock avoidance allows the three necessary conditions but makes judicious choices to assure that the deadlock point is never reached.
- With deadlock avoidance, a decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to a deadlock.
- Deadlock avoidance thus requires knowledge of future process resource requests.
- **Approaches to deadlock avoidance:**
  - Process Initiation Denial
  - Resource Allocation Denial

Process Initiation Denial

Consider a system with n processes and m different types of resources.

[Resource = $\mathbf{R} = (R_1, R_2, \ldots, R_m)$	total amount of each resource in the system
Available = $\mathbf{V} = (V_1, V_2, \ldots, V_m)$	total amount of each resource not allocated to any process
Claim = $\mathbf{C} = \begin{bmatrix} C_{11} & C_{12} & \ldots & C_{1m} \\ C_{21} & C_{22} & \ldots & C_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ C_{n1} & C_{n2} & \ldots & C_{nm} \end{bmatrix}$	C_{ij} = requirement of process i for resource j
Allocation = $\mathbf{A} = \begin{bmatrix} A_{11} & A_{12} & \ldots & A_{1m} \\ A_{21} & A_{22} & \ldots & A_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ A_{n1} & A_{n2} & \ldots & A_{nm} \end{bmatrix}$	A_{ij} = current allocation to process i of resource j

Process Initiation Denial (contd…)

- The following relationship holds:

1. $R_j = V_j + \sum_{i=1}^{n} A_{ij}$,	for all j	All resources are either available or allocated.
2. $C_{ij} \leq R_j$,	for all i, j	No process can claim more than the total amount of resources in the system.
3. $A_{ij} \leq C_{ij}$,	for all i, j	No process is allocated more resources of any type than the process originally claimed to need.

- With these quantities defined, we can define a deadlock avoidance policy that refuses to start a new process if its resource requirements might lead to deadlock.

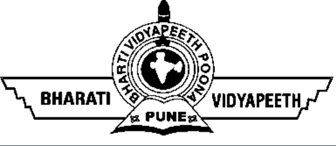
- Start a new process P_{n+1} only if $R_j \geq C_{(n+1)j} + \sum_{i=1}^{n} C_{ij}$ for all j

Resource Allocation Denial (Banker's Algorithm)

- Why Banker's algorithm is named so?
  - Banker's algorithm is named so because it is used in banking system to check whether loan can be sanctioned to a person or not.
  - The customers who wish to borrow money corresponds to processes and the money to be borrowed corresponds to resources.
  - Suppose there are **n** number of account holders in a bank and the total sum of their money is **S**. If a person applies for a loan then the bank first subtracts the loan amount from the total money that bank has and if the remaining amount is greater than **S** then only the loan is sanctioned. It is done because if all the account holders comes to withdraw their money then the bank can easily do it.
  - In other words, the bank would never allocate its money in such a way that it can no longer satisfy the needs of all its customers. The bank would try to be in safe state always.

Banker's Algorithm

- Consider a system with a fixed number of processes and a fixed number of resources.
- At any time a process may have zero or more resources allocated to it.
- The state of the system reflects the current allocation of resources to processes.
  - The state consists of the two vectors, Resource and Available, and the two matrices, Claim and Allocation.
  - A state of the system is called safe if the system can allocate all the resources requested by all the processes without entering into deadlock.
  - If the system cannot fulfill the request of all processes then the state of the system is called unsafe.



Banker's Algorithm (contd…)

- Banker's algorithm is a deadlock avoidance and resource allocation algorithm.
- When a process makes a request for a set of resources, assume that the request is granted, update the system state accordingly, and then determine if the result is a safe state.
- If so, grant the request and, if not, block the process until it is safe to grant the request.

Banker's Algorithm – Example 1

Claim matrix C

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Allocation matrix A

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

C – A

	R1	R2	R3
P1	2	2	2
P2	0	0	1
P3	1	0	3
P4	4	2	0

Resource vector R

R1	R2	R3
9	3	6

Available vector V

R1	R2	R3
0	1	1

Initial State

Banker's Algorithm – Example 1 (contd..)

Claim matrix C

	R1	R2	R3
P1	3	2	2
P2	0	0	0
P3	3	1	4
P4	4	2	2

Allocation matrix A

	R1	R2	R3
P1	1	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

C – A

	R1	R2	R3
P1	2	2	2
P2	0	0	0
P3	1	0	3
P4	4	2	0

Resource vector R

R1	R2	R3
9	3	6

Available vector V

R1	R2	R3
6	2	3

P2 runs to completion

Banker's Algorithm – Example 1 (contd..)

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
7	2	3

Available vector V

P1 runs to completion

Banker's Algorithm – Example 1 (contd..)

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
9	3	4

Available vector V

P3 runs to completion, similarly P4 will also run to completion.

It is clear that the given state is safe state.

Banker's Algorithm – Example 2

Given the initial state, assume that P1 has an additional unit of R1 and R3.
Determine the state of the system.

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	1	0	0
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	2	2	2
P2	1	0	2
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
1	1	2

Available vector V

Initial State

If we assume that the request is granted, the state will be as shown in next slide.

Banker's Algorithm - Example 2 (contd..)

P1 requests one additional unit of R1 and R3

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	2	0	1
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	1	2	1
P2	1	0	2
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
0	1	1

Available vector V

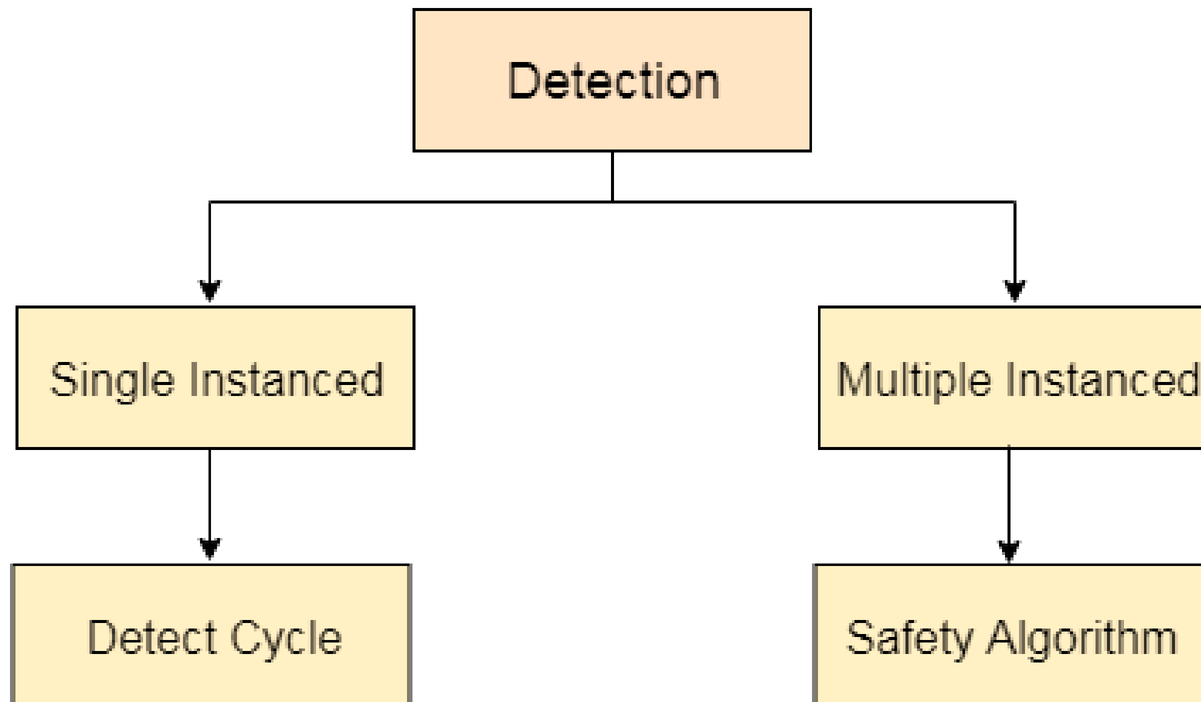
Is this a safe state? The answer is no, because each process will need at least one additional unit of R1, and there are none available.

Thus, on the basis of deadlock avoidance, the request by P1 should be denied and P1 should be blocked.

Deadlock Detection and Recovery

- In this approach, the operating system does not apply any mechanism to avoid or prevent the deadlocks.
- Therefore, the system considers that the deadlock will definitely occur.
- In order to get rid of deadlocks, the operating system periodically checks the system for any deadlock.
- In case, it finds any of the deadlock, then, the operating system will recover the system using some recovery techniques.
- The main task of the operating system is detecting the deadlocks. The operating system can detect the deadlocks with the help of Resource Allocation Graph.

Deadlock Detection and Recovery (contd..)



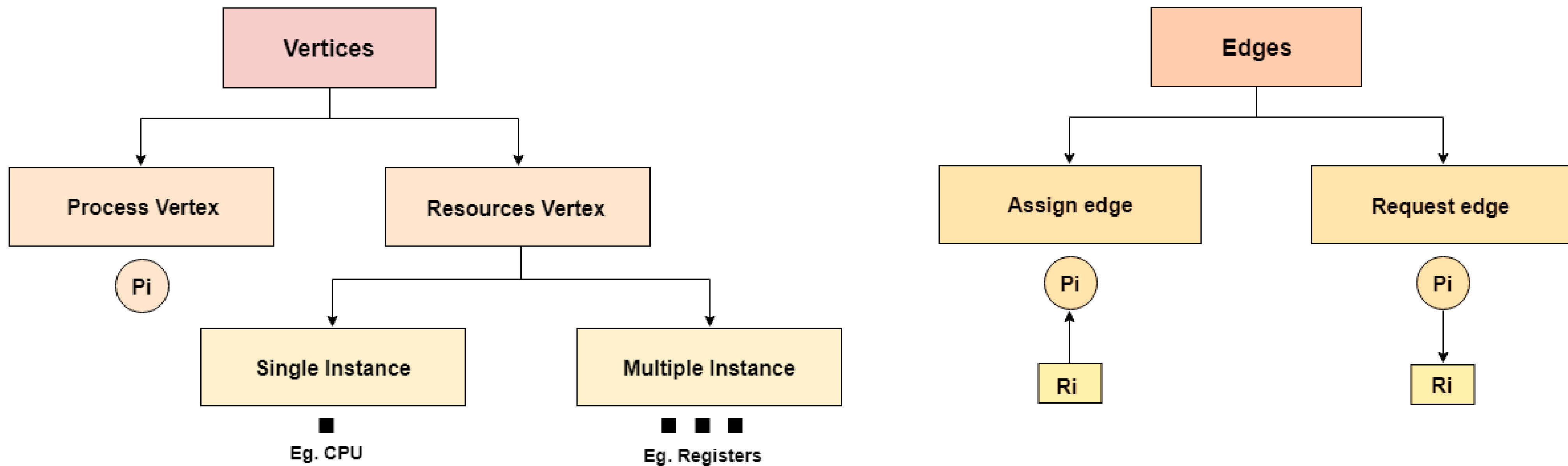
Resource-Allocation Graph

- Deadlocks can be described more precisely in terms of a directed graph, called a system **Resource-Allocation Graph**:
  - This graph consists of a set of vertices **V** and a set of edges **E**.
  - The set of vertices V is partitioned into two different types of nodes:
    - **$P = \{P_1, P_2, ..., P_n\}$**, the set consisting of all the active processes in the system.
    - **$R = \{R_1, R_2, ..., R_m\}$**, the set consisting of all resource types in the system.
  - A directed edge from process **P_i** to resource type **R_j** is denoted by **$P_i \rightarrow R_j$** **(request edge)**; it signifies that process **P_i** has requested an instance of resource type **R_j** and is currently waiting for that resource.
  - A directed edge from resource type **R_j** to process **P_i** is denoted by **$R_j \rightarrow P_i$** **(assignment edge)**; it signifies that an instance of resource type **R_j** has been allocated to process **P_i** .

Resource-Allocation Graph (contd…)

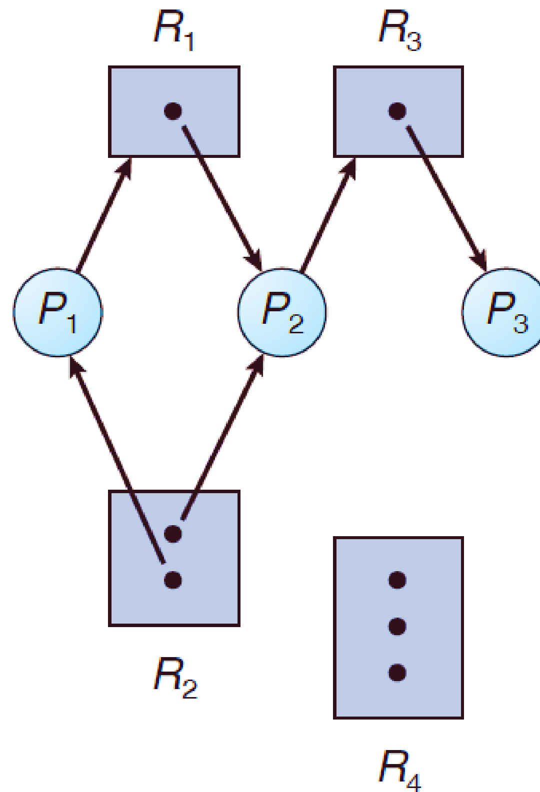
- Each process P_i is represented as a circle and each resource type R_j is represented as a rectangle.
  - Resource type R_j may have more than one instance, each instance is represented as a dot within the rectangle.
  - A request edge points to only the rectangle R_j, whereas an assignment edge must also designate one of the dots in the rectangle.
- When process P_i requests an instance of resource type R_j, a request edge is inserted in the resource-allocation graph.
- When this request can be fulfilled, the request edge is instantaneously transformed to an assignment edge.
- When the process no longer needs access to the resource, it releases the resource; as a result, the assignment edge is deleted.

Resource-Allocation Graph (contd…)



Resource-Allocation Graph: Example

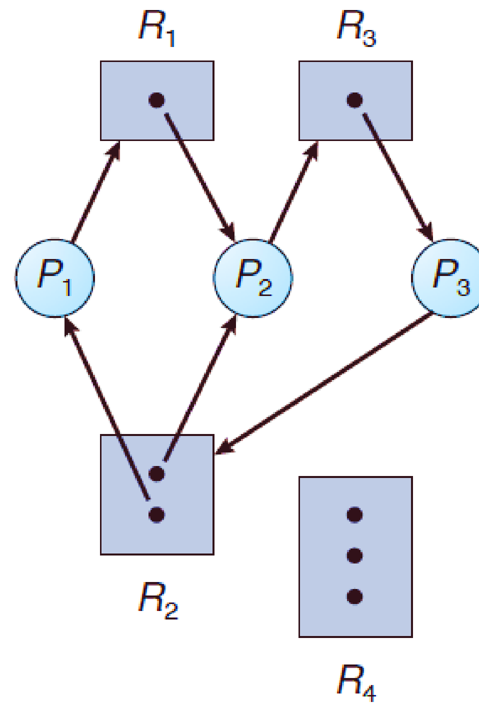
- P = {P1, P2, P3}
- R = {R1, R2, R3, R4}
- E = {P1 → R1, P2 → R3, R1 → P2, R2 → P2, R2 → P1, R3 → P3}



Deadlock Detection using RAG

- If the graph contains no cycles, then no process in the system is deadlocked.
- If the graph does contain a cycle, then a deadlock may exist.
- If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred.
- If the cycle involves only a set of resource types, each of which has only a single instance, then a deadlock has occurred.
- Each process involved in the cycle is deadlocked.
- In this case, a cycle in the graph is both a necessary and a sufficient condition for the existence of deadlock.
- If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred. In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.

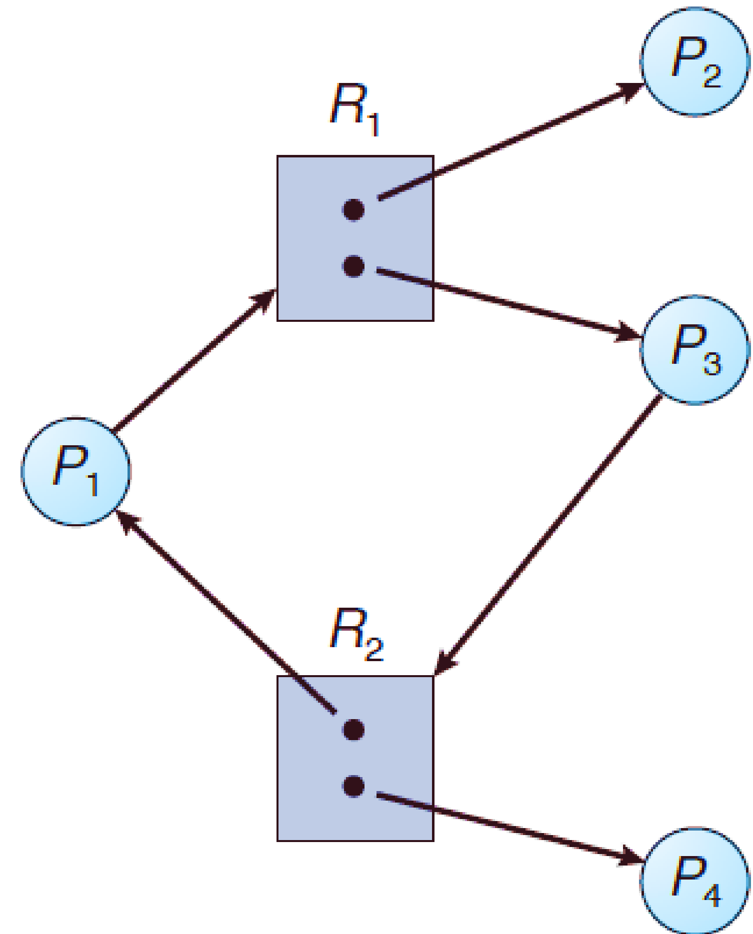
Deadlock Detection using RAG: Example



- **Cycle 1:** $P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$
- **Cycle 2:** $P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$
- Processes P_1, P_2, and P_3 are deadlocked.

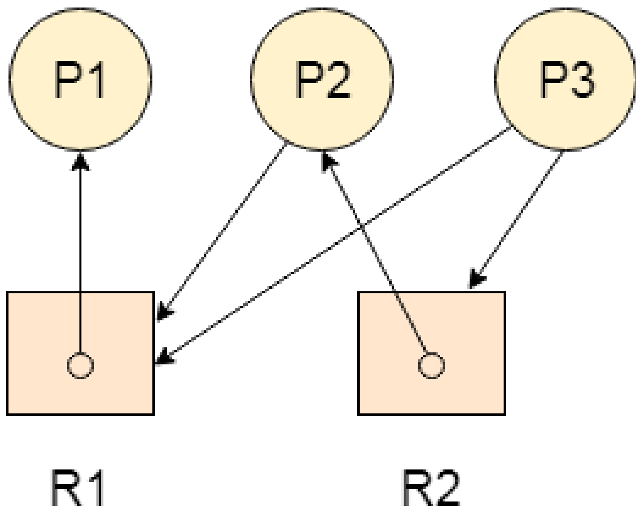
Deadlock Detection using RAG: Example

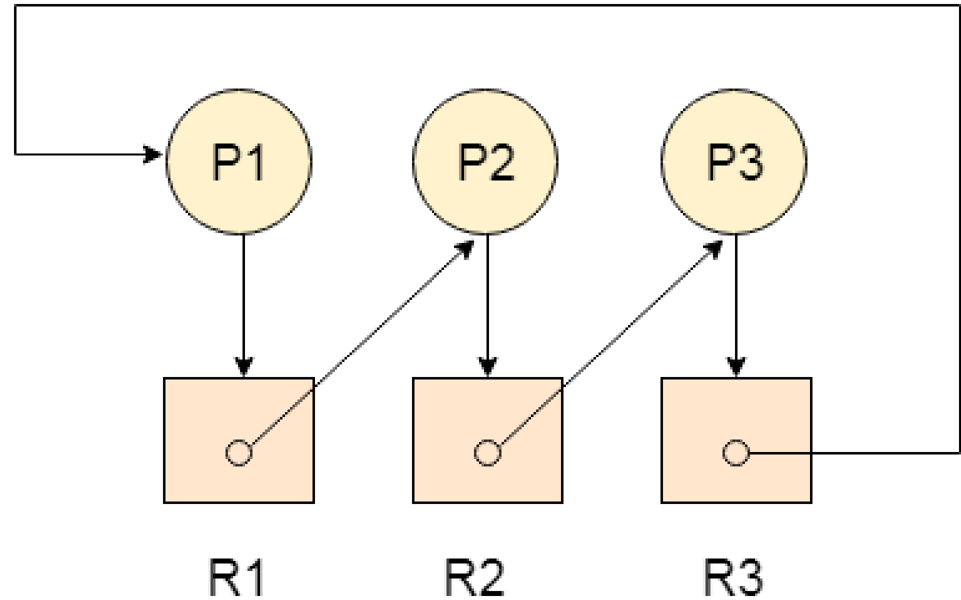
- Cycle 1: $P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$
- There is no deadlock.
- The process P_4 may release its instance of resource type R_2. That resource can then be allocated to P_3, breaking the cycle.



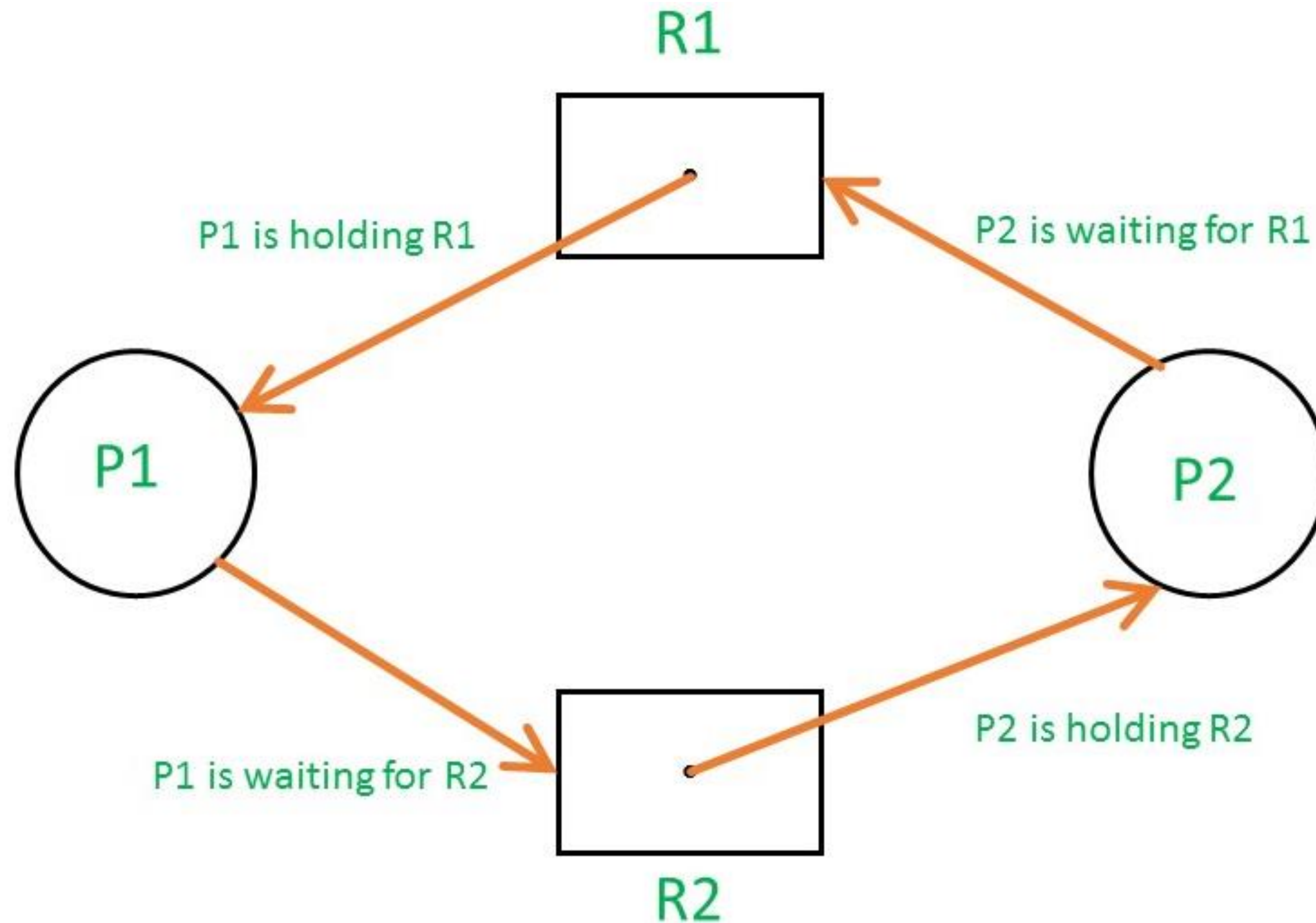
In summary, if a resource-allocation graph does not have a cycle, then the system is not in a deadlocked state. If there is a cycle, then the system may or may not be in a deadlocked state.

Deadlock Detection using RAG: Example

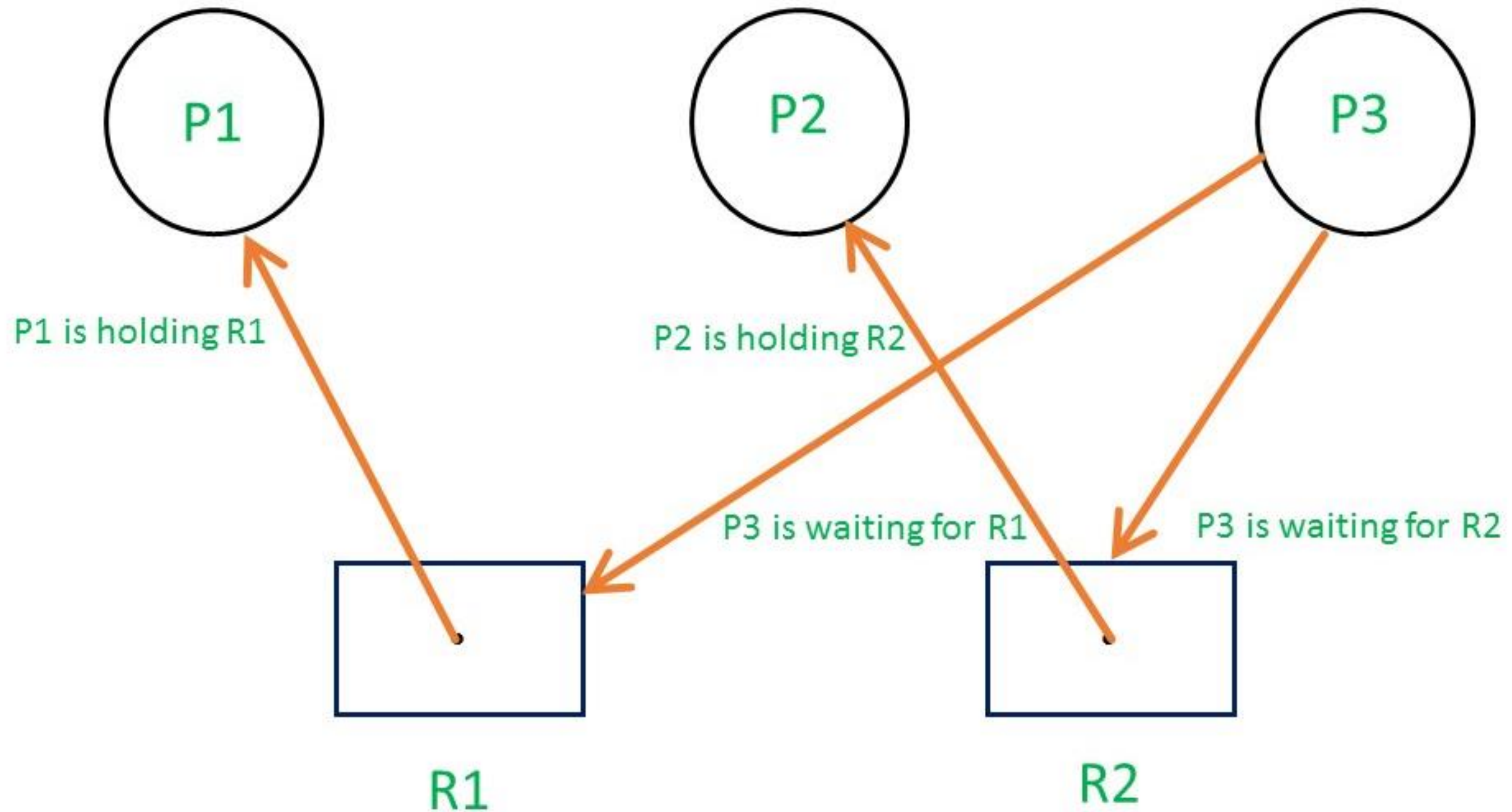




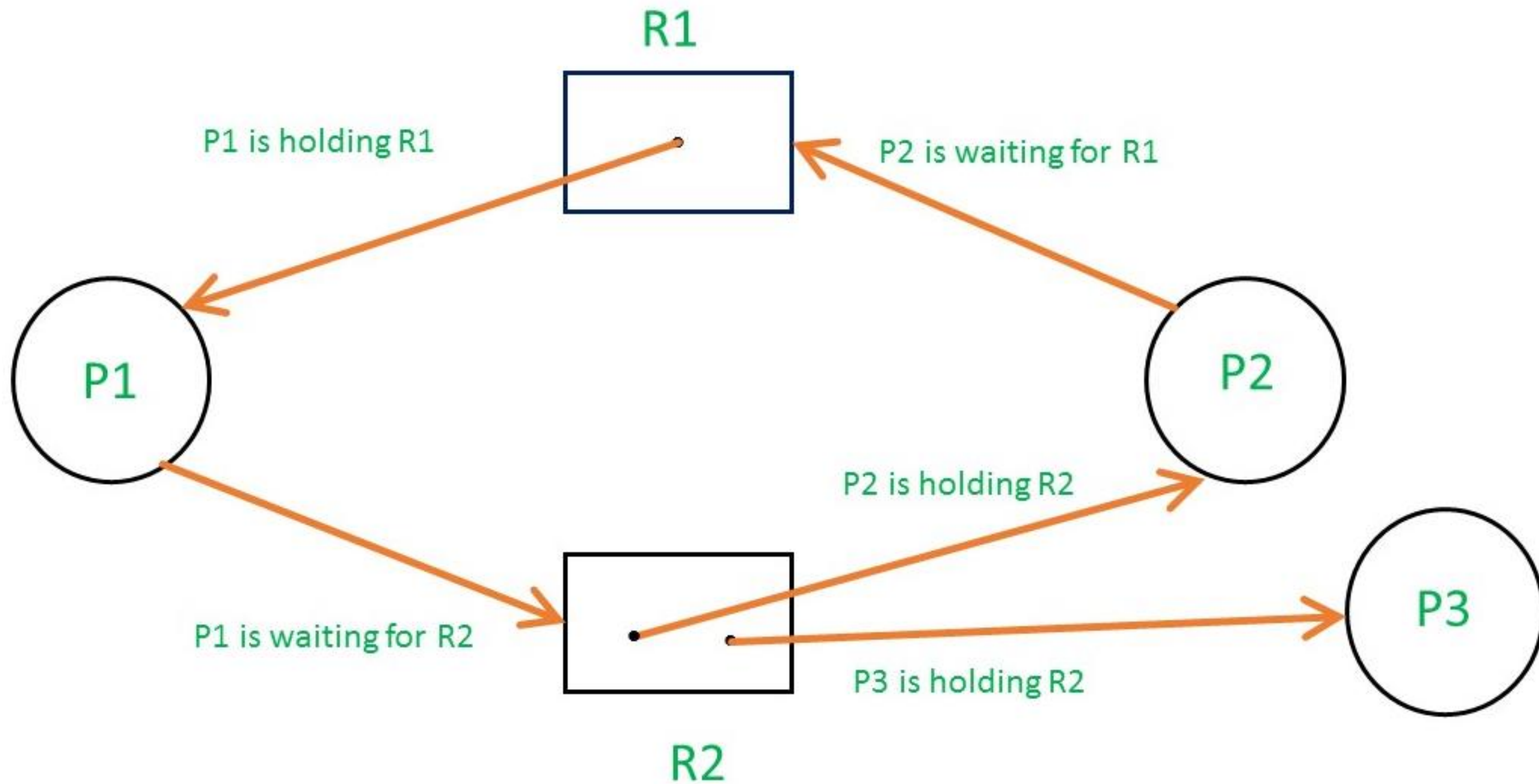
Deadlock Detection using RAG: Example



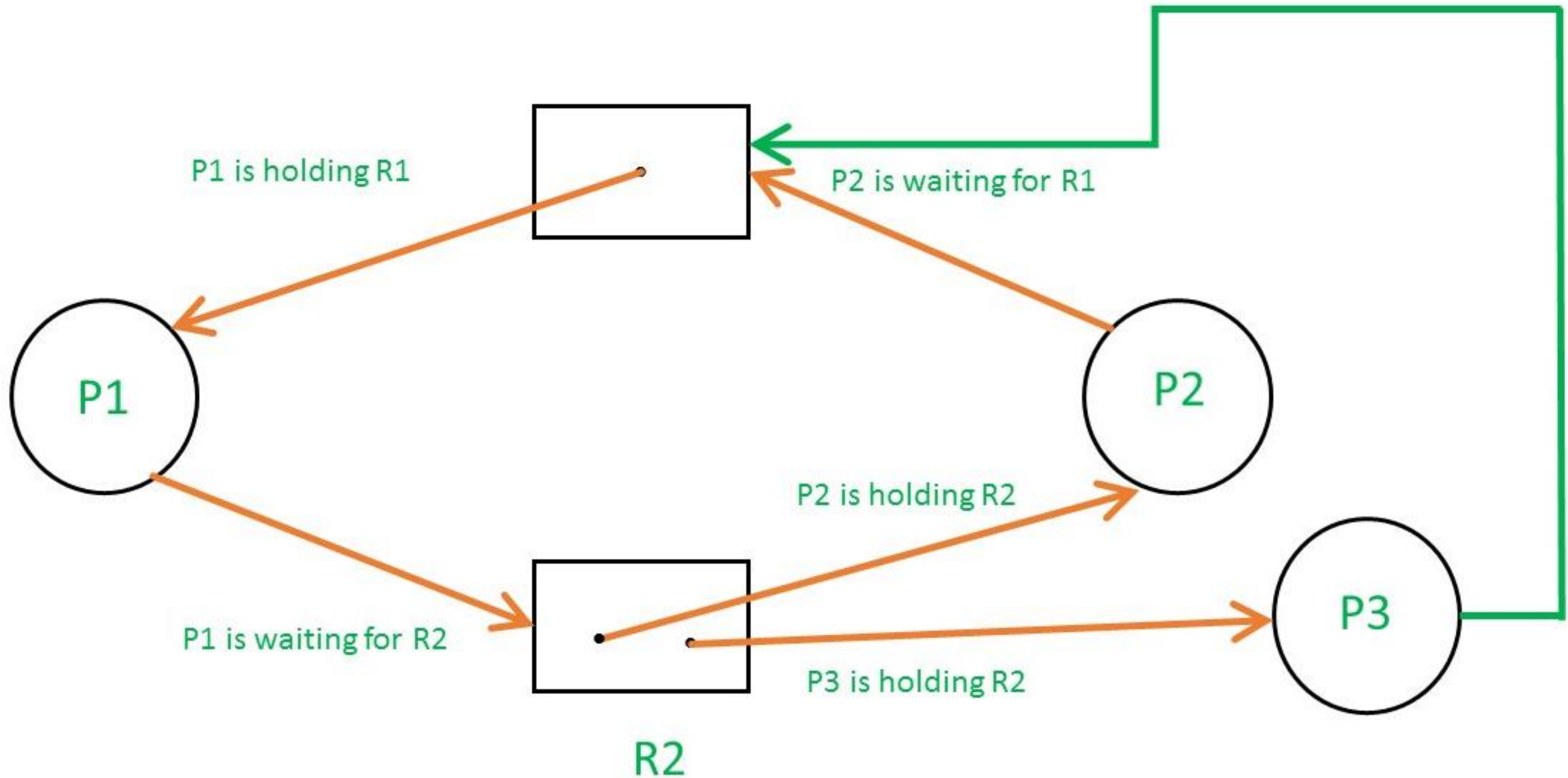
Deadlock Detection using RAG: Example



Deadlock Detection using RAG: Example



Deadlock Detection using RAG: Example



Deadlock Detection for Several Instance of Resource

- Let, an Allocation Matrix, a Request Matrix Q, and an Available Vector
- The algorithm proceeds by marking processes that are not deadlocked. Initially, all processes are unmarked. Then the following steps are performed:

1. Mark each process that has a row in the Allocation matrix of all zeros.
2. Initialize a temporary vector **W** to equal the Available vector.
3. Find an index i such that process i is currently unmarked and the ith row of **Q** is less than or equal to **W**. That is, $Q_{ik} \leq W_k$, for $1 \leq k \leq m$. If no such row is found, terminate the algorithm.
4. If such a row is found, mark process i and add the corresponding row of the allocation matrix to **W**. That is, set $W_k = W_k + A_{ik}$, for $1 \leq k \leq m$. Return to step 3.

- A deadlock exists if and only if there are unmarked processes at the end of the algorithm. Each unmarked process is deadlocked.

BHARATI VIDYAPEETH PUNE

Deadlock Detection - Example

Request matrix Q

	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
P3	0	0	0	0	1
P4	1	0	1	0	1

Allocation matrix A

	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

Resource vector

R1	R2	R3	R4	R5
2	1	1	2	1

Available vector

R1	R2	R3	R4	R5
0	0	0	0	1

- Mark P4, because P4 has no allocated resources.
- Set W = (0 0 0 0 1).
- The request of process P3 is less than or equal to W, so mark P3 and set W = W + (0 0 0 1 0) = (0 0 0 1 1).
- No other unmarked process has a row in Q that is less than or equal to W. Therefore, terminate the algorithm.
- The algorithm concludes with P1 and P2 unmarked, indicating that these processes are deadlocked.

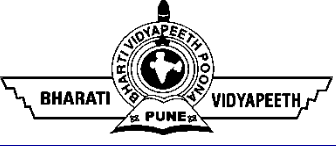
Deadlock Recovery

- To recover deadlock, the operating system examines either resources or processes.

- For Process:
  - Kill a Process: In this approach, kill the process due to which deadlock occurred. But the selection of the process to kill is a tough task. In this, the operating system mainly kills that process, which does not work more till now.
  - Kill all Process: Kill all the processes is not a suitable approach. We can use this approach when the problem becomes critical. By killing all the processes, the system efficiency will be decreased, and we have to execute all the processes further from the start.

Deadlock Recovery (contd…)

- For Resources:
  - Preempt the Resource: In this, we take the resource from one process to the process that needs it to finish its execution, and after the execution is completed, the process soon releases the resource. In this, the resource selection is difficult, and the snatching of the resource is also difficult.
  - Rollback to a Safe State: To enter into the deadlock, the system goes through several states. In this, the operating system can easily roll back the system to the earlier safe state. To do so, we require to implement checkpoints at every state. At the time when we detect deadlock, then we need to rollback every allocation so that we can enter into the earlier safe state.



Background: Clock Cycle

- CPU speed is determined by the clock cycle.
- The clock cycle is the amount of time between two pulses of an oscillator.
- The clock speed is measured in Hz, often either megahertz (MHz) or gigahertz (GHz).
  - For example, a 4 GHz processor performs 4,000,000,000 clock cycles per second.

Background: CPU Cycle

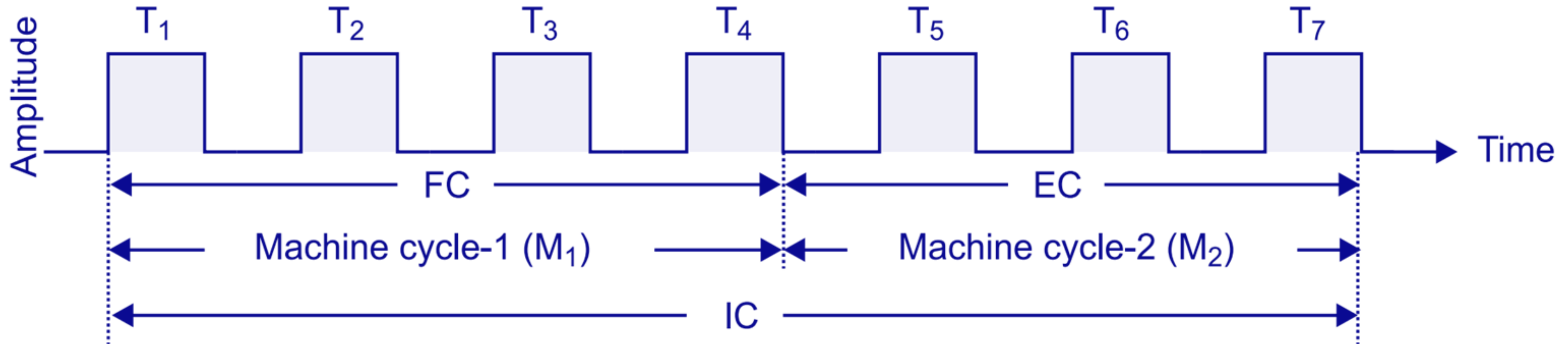
- **CPU cycle** is also called **machine cycle**.
- **CPU cycle refers to the** time required for the execution of one simple **processor operation** such as an addition.
  - For an instruction cycle, we fetch an instruction and execute it, at least two CPU cycles are required. At least one CPU cycle is required to fetch instructions, and at least one CPU cycle is required to execute them. Complex instructions require more CPU cycles.
- An instruction cycle may include multiple CPU cycles, and a CPU cycle may include multiple clock cycles.

Background: Instruction Cycle

- Instruction cycle refers to the time taken to execute an instruction. It is the basic operational process of a computer. This process is repeated continuously by CPU from boot up to shut down of computer.

- The execution process of instructions is divided into the following steps:
  - **Fetch** - The instruction is fetched from memory address that is stored in PC (Program Counter) and stored in the Instruction Register IR.
  - **Decode** - According to the instructions in the instruction register, decode what kind of operation is to be parsed.
  - **Execute** - Run the corresponding instructions to perform arithmetic and logic operations, data transmission, etc.

Instruction Cycle, Machine Cycle and Clock Cycle



Background: Memory

- Main memory and the registers built into the processor itself are the only storage that the CPU can access directly.
- There are machine instructions that take memory addresses as arguments, but none take disk addresses.
- Therefore, any instructions in execution, and any data being used by the instructions, must be in one of these direct-access storage devices.
- If the data are not in memory, they must be moved there before the CPU can operate on them.
- Registers that are built into the CPU are generally accessible within one clock cycle.
- Most CPUs can decode instructions and perform simple operations on register contents at the rate of one or more operations per clock tick.
  - The same cannot be said of main memory, which is accessed via a transaction on the memory bus.

Background: Memory (contd…)

- Completing a memory access may take many clock cycles.
- In such cases, the processor normally needs to stall, since it does not have the data required to complete the instruction that it is executing.
- This situation is intolerable because of the frequency of memory accesses.
- The remedy is to add fast memory between the CPU and main memory.
- A memory buffer used to accommodate a speed differential, is called cache.

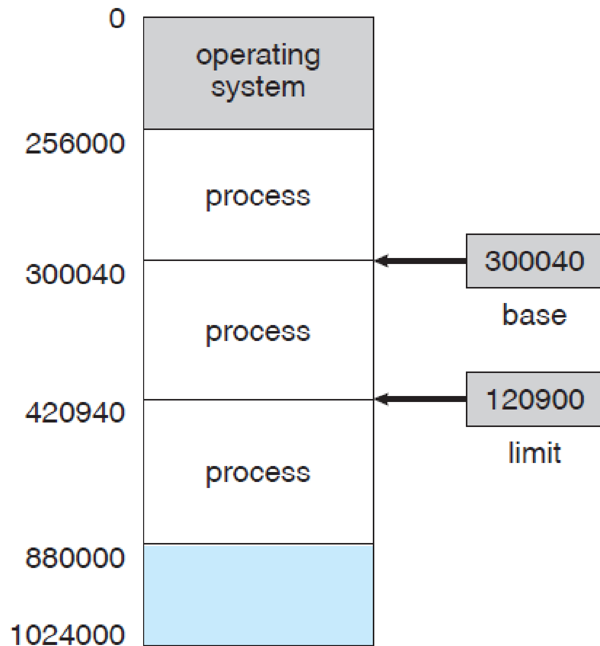
BHARATI VIDYAPEETH PUNE

Basic Hardware

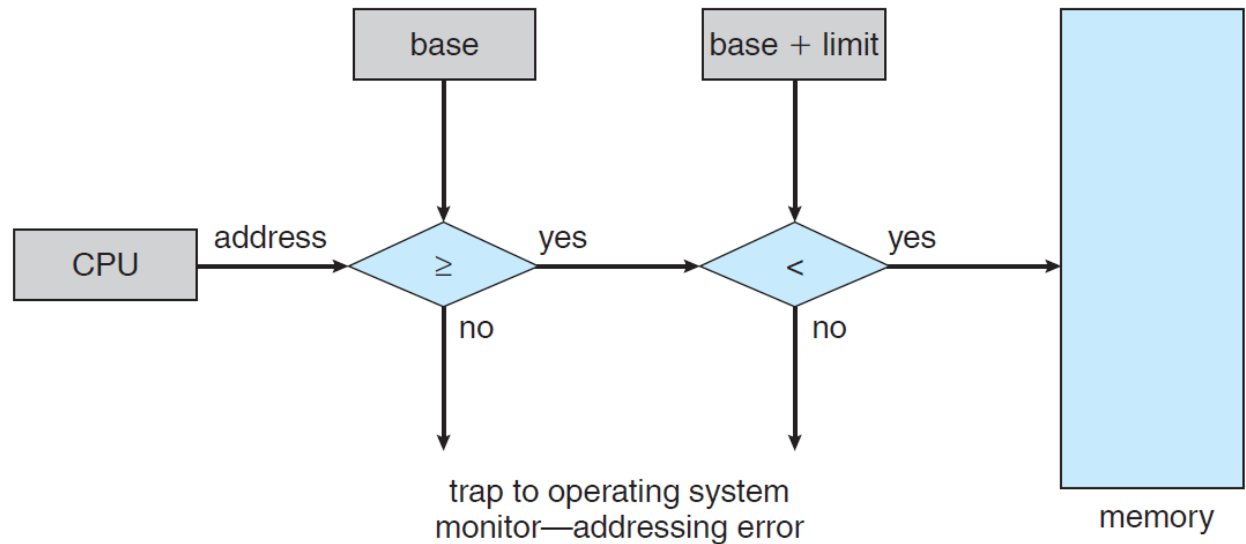
- There are several hardware-level approaches to protect the operating system from access by user processes and, in addition, to protect user processes from one another.
- To make sure that each process has a separate memory space, the protection is provided by using two registers: **Base Register** and **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).
  - Protection of memory space is accomplished by having the CPU hardware compare every address generated in user mode with the registers.
  - Any attempt by a program executing in user mode, to access operating-system memory or other users' memory, results in a trap to the operating system, which treats the attempt as **a fatal error** (an error that causes a program to terminate without any warning or saving its state).
    - This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the operating system or other users.

Hardware Address Protection

- The base and limit registers can be loaded only by the operating system, which uses a special privileged instruction.



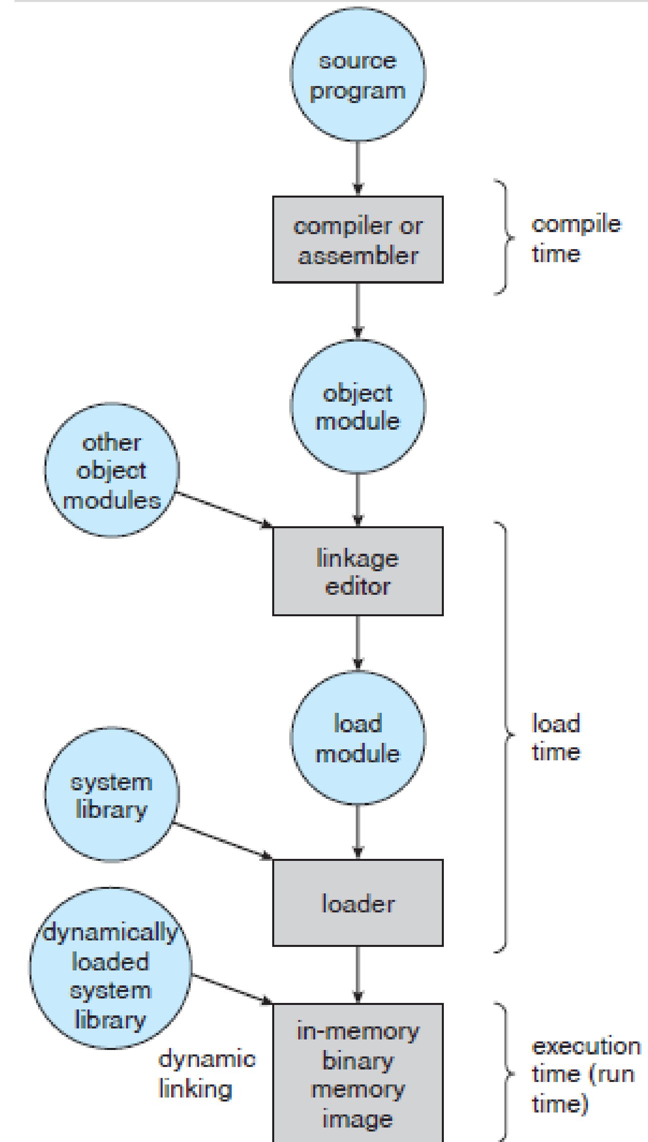
Base and Limit Register defining a logical address space



Hardware address protection with Base and Limit Registers

Basic Concepts

- Address Binding
  - Compile Time Binding
  - Load Time Binding
  - Execution Time Binding
- Logical Address and Physical Address
- Linking
  - Static Linking
  - Dynamic Linking
- Loading
  - Static Loading
  - Dynamic Loading
- Swapping

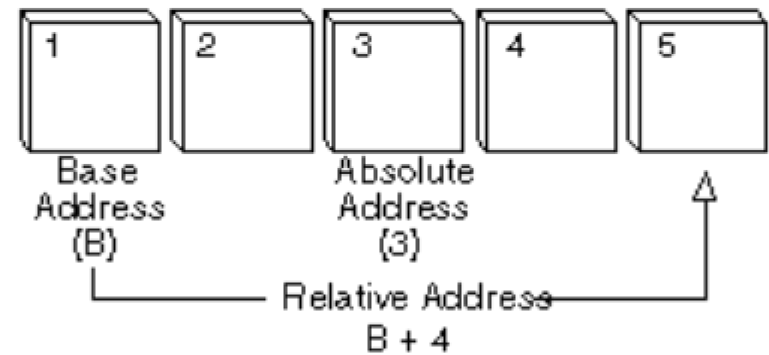


Address Binding

- The process (program) 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.
- The normal procedure is to select one of the processes in the input queue and to load that process into memory.
- In most cases, a user program go through several steps (such as compiling, loading, execution) before being executed.
  - Addresses may be represented in different ways during these steps.
  - Addresses in the source program are generally **symbolic** (such as count).

Address Binding (contd…)

- The **binding** of instructions and data **to memory addresses** can be done at any step along the way:
  - Compile Time
  - Load Time
  - Execution (Run) Time

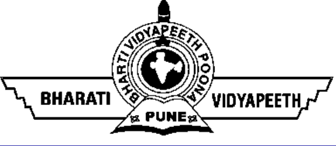


Address Binding (contd…)

- Compile Time
  - If it is known at compile time where the process will reside in memory, then absolute code (physical address is embedded) can be generated.
    - For example, if it is known that a user process will reside starting at location R, then the generated compiler code will start at that location and extend up from there.
    - If, at some later time, the starting location changes, then it will be necessary to recompile this code.

Address Binding (contd…)

- Load Time
  - If it is not known at compile time where the process will reside in memory, then relocatable code can be generated.
    - In this case, final binding is delayed until load time.
    - If the starting address changes, we need only reload the user code to incorporate this changed value.



Address Binding (contd…)

- Execution Time
  - If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time.

Summary of Address Bindings

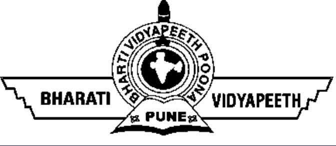
- Compile Time Binding: It is the translation of logical addresses to physical addresses at the time of compilation. Now this type of binding is only possible in systems where we know the contents of the main memory in advance and know what address in the main memory we have to start the allocation from. Knowing both of these things is not possible in modern multi-processing systems. So it can be safely said the compile time binding would be possible in systems not having support for multi-processing.

- Load Time Binding: It is the translation of the logical addresses to physical addresses at the time of loading. The relocating loader contains the base address in the main memory from where the allocation would begin. So when the time for loading a process into the main memory comes, all logical addresses are added to the base address by the relocating loader to generate the physical addresses.

- Run Time Binding: In most modern processors multi-processing is supported. Therefore, there comes the need of shifting the physical addresses from one location to another during run time. This is taken care by the run time binding concept. It is used in Compaction to remove External Fragmentation.

Logical versus Physical Address Space

- An address generated by the CPU is commonly referred to as a logical address.
  - The logical address is virtual address as it does not exist physically, therefore, it is also known as virtual address.
  - This address is used as a reference to access the physical memory location by CPU.
- An address seen by the memory unit (that is, the one loaded into the memory-address register of the memory) is referred to as a physical address.
  - Physical address identifies a physical location of required data in a memory.
- The compile-time and load-time address-binding methods generate identical logical and physical addresses.
- The execution-time address-binding scheme results in differing logical and physical addresses.



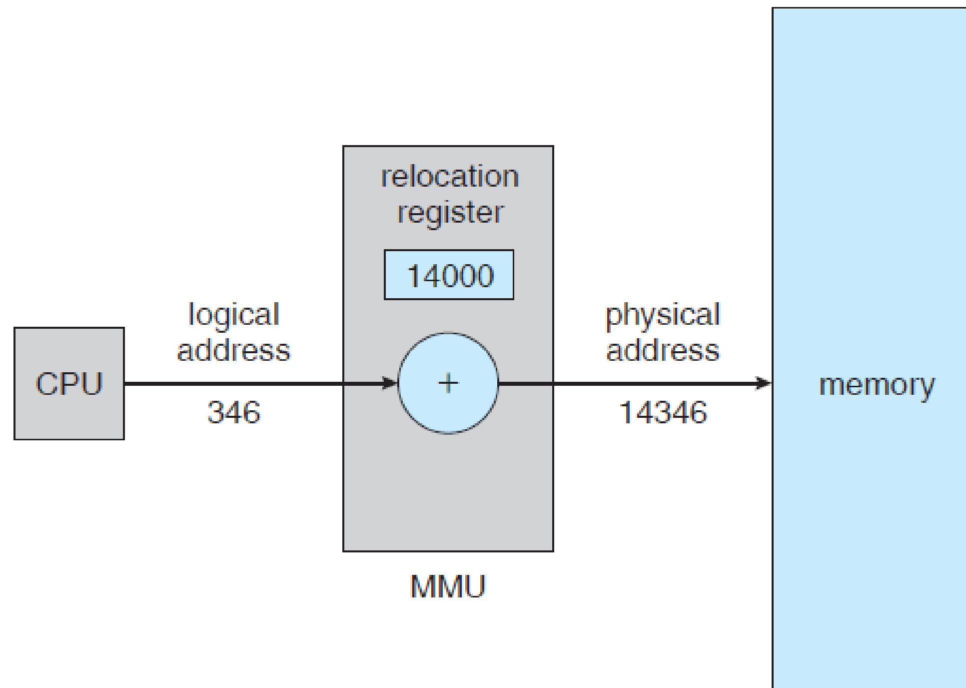
Logical and Physical Address Space

- The set of all logical addresses generated for a program's perspective 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).

Mapping from Virtual to Physical Addresses



Example: Simple MMU scheme (a generalization of the base-register scheme)

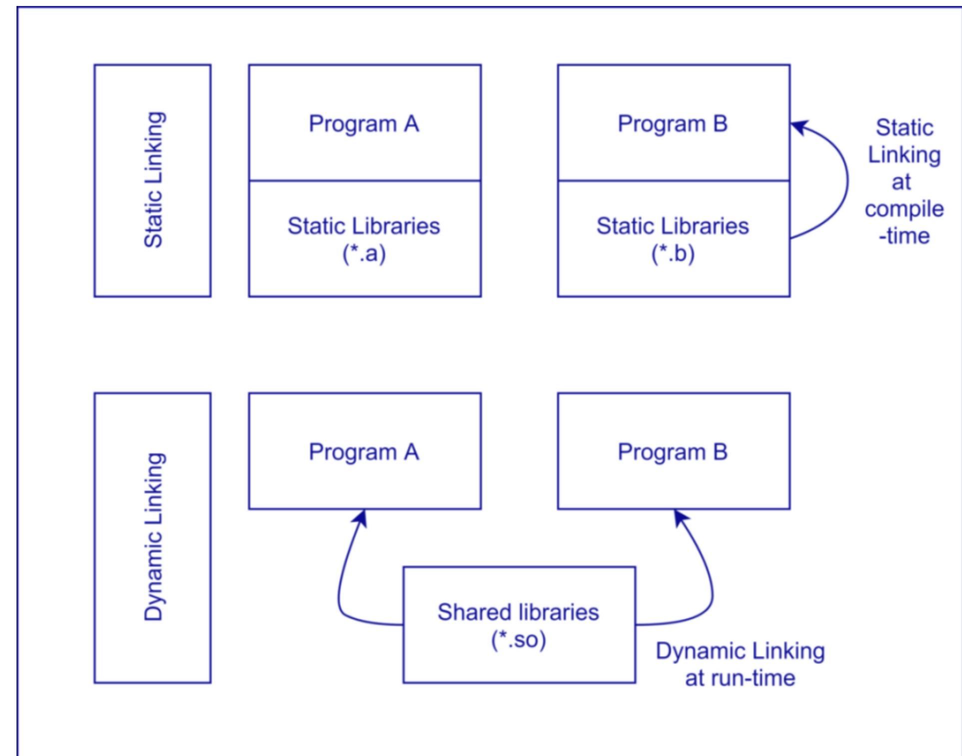
- CPU generates logical address 346.
- MMU generates relocation register (base register) 14000.
- In memory, the physical address is 14346 (346+14000).

Linking

- Linking intends to generate an executable module of a program by combining the object codes generated by the compiler or assembler.
- Linking is the process of connecting all the modules or the function of a program for program execution.
- The linker, also known as the link editor, takes object modules from the assembler and forms an executable file for the loader.
- Linking is classified into two types, based on the time when it is done:
  - Static Linking
  - Dynamic Linking

Linking (contd…)

- In the static linking, each program binds to its dependent libraries at compile time.
  - With static linking, the user ends up copying functions or routines that are repetitive across various executables.
  - For example: Nearly every program needs printf() function. Thus, a copy of it is present in all executables which wastes space.
- In the case of dynamic linking, programs use shared libraries, and these libraries are linked to the programs at run time.



Dynamic Linking

- In the dynamic linking approach, the linker does not copy the routines into the executables. It takes note that the program has a dependency on the library.
- With dynamic linking, a stub is included in the image for each library routine reference.
  - The stub is a small piece of code that indicates how to locate the appropriate memory-resident library routine or how to load the library if the routine is not already present.
  - **When the stub is executed, it checks to see whether the needed routine is already in memory.** If it is not, the program loads the routine into memory.
  - Under this scheme, all processes that use a language library execute only one copy of the library code.
  - This feature can be extended to library updates (such as bug fixes). A library may be replaced by a new version, and all programs that reference the library will automatically use the new version.

Loading

- Loading is the process of loading the program from secondary memory to the main memory for execution.
- It is necessary for the entire program and all data of a process to be in physical memory for the process to execute.
- The size of a process has thus been limited to the size of physical memory.
- To obtain better memory-space utilization, we can use dynamic loading.
  - Dynamic loading is the technique through which a computer program, at runtime, load a library into memory, retrieve the variable and function addresses, executes the functions, and unloads the program from memory.
  - It is often used to implement software plugins.

Dynamic Loading

- With dynamic loading, a routine is not loaded until it is called.
- All routines are kept on disk in a relocatable load format.
- The main program is loaded into memory and is executed.
- When a routine needs to call another routine, the calling routine first checks to see whether the other routine has been loaded.
- If it has not, the relocatable linking loader is called to load the desired routine into memory and to update the program's address tables to reflect this change. Then control is passed to the newly loaded routine.

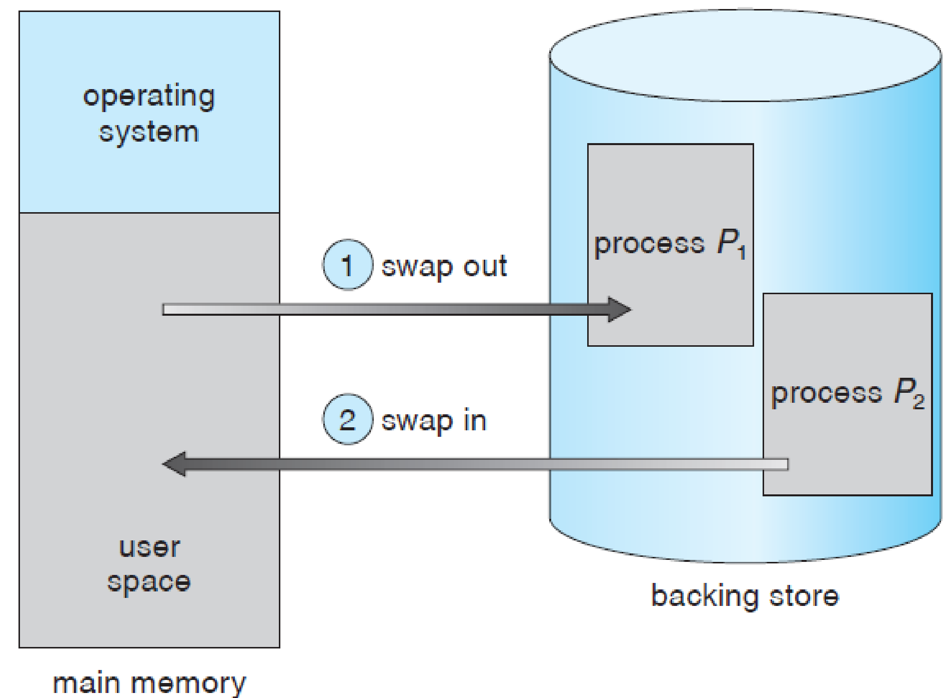
Dynamic Loading (contd…)

- Advantage of Dynamic Loading:
  - An unused routine is never loaded.
  - This method is particularly useful when large amounts of code are needed to handle infrequently occurring cases, such as error routines.
  - In this case, although the total program size may be large, the portion that is used (and hence loaded) may be much smaller.
- Dynamic loading does not require special support from the operating system.
- It is the responsibility of the users to design their programs to take advantage of such a method.
- Operating systems may help the programmer, however, by providing library routines to implement dynamic loading. Loading Examples of Java:

```
Class.forName (String className); //Dynamic Loading        TestClass tc = new TestClass(); //Static Loading
```

Swapping

- **A process must be in memory to be executed.**
- A process can be swapped temporarily out of memory to a backing store and then brought back into memory for continued execution.
- **Example**: Assume a multiprogramming environment with a round-robin CPU-scheduling algorithm. When a quantum expires, the memory manager will start to swap out the process that just finished and to swap another process into the memory space that has been freed.



Swapping (contd…)

- A variant of swapping policy is used for priority-based scheduling algorithms.
- If a higher-priority process arrives and wants service, the memory manager can swap out the lower-priority process and then load and execute the higher-priority process.
- When the higher-priority process finishes, the lower-priority process can be swapped back in and continued.
- This variant of swapping is sometimes called roll out, roll in.
- Normally, a process that is swapped out will be swapped back into the same memory space it occupied previously.
- This restriction is dictated by the method of address binding. If binding is done at assembly or load time, then the process cannot be easily moved to a different location. If execution-time binding is being used, then a process can be swapped into a different memory space, because the physical addresses are computed during execution time.

Swapping (contd…)

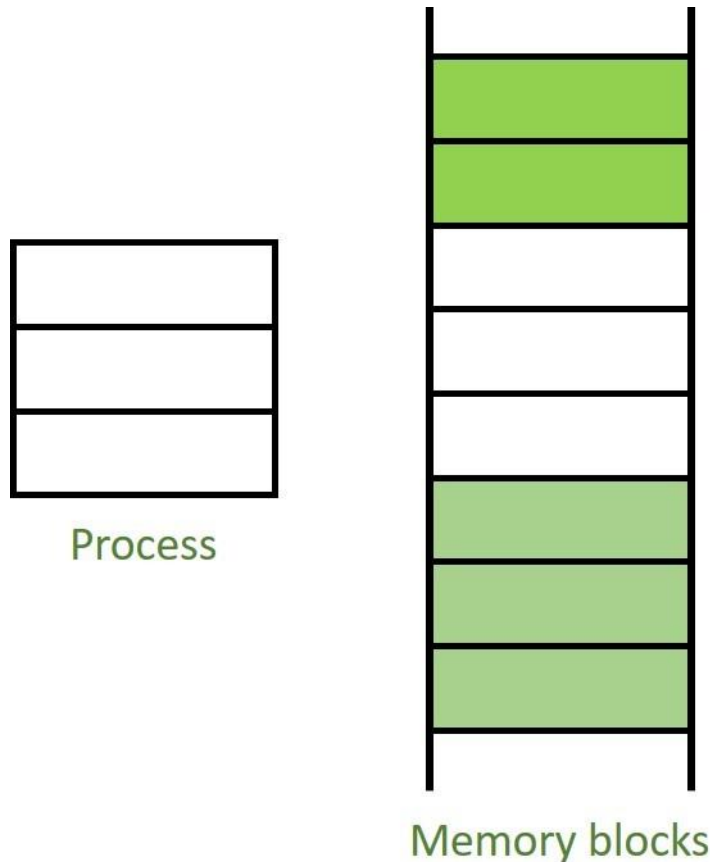
- Swapping requires a backing store (a fast disk, large enough to accommodate copies of all memory images, and it must provide direct access to these memory images).
- System maintains a ready queue consisting of all processes whose memory images are on the backing store or in memory and are ready to run.
- Whenever the CPU scheduler decides to execute a process, it calls the dispatcher.
- The dispatcher checks to see whether the next process in the queue is in memory.
- If it is not, and if there is no free memory region, the dispatcher swaps out a process currently in memory and swaps in the desired.

Memory Management Techniques

- There are two techniques for memory management:
  - Contiguous Memory Allocation
  - Non-Contiguous Memory Allocation
- In Contiguous Memory Allocation, the process must be loaded entirely in main-memory at contiguous locations.
- In Non-Contiguous Memory Allocation, the process is loaded in several memory blocks at different memory locations in the memory.

Contiguous Memory Allocation

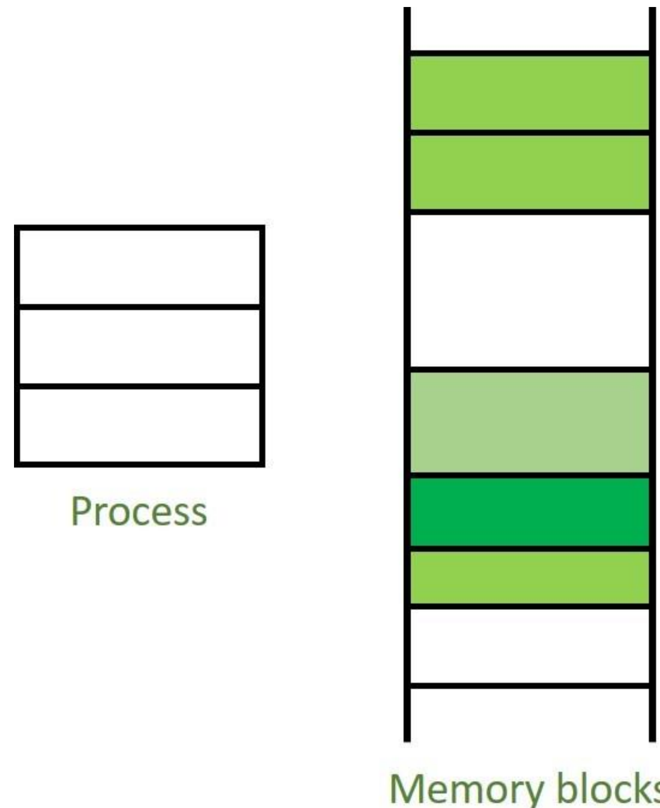
- A single contiguous section/part of memory is allocated to a process or file needing it.



Contiguous Memory Allocation

Non-Contiguous Memory Allocation

- The process is loaded in several memory blocks at different memory locations in the memory.

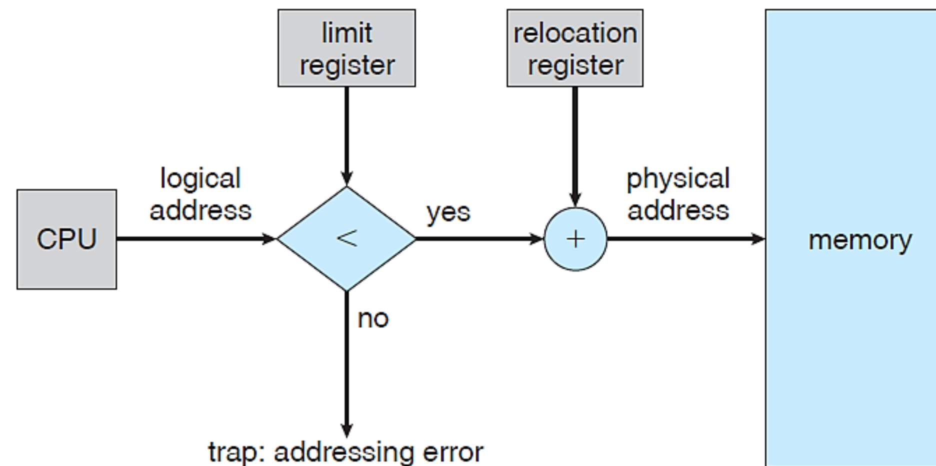


Contiguous Memory Allocation

- In contiguous memory allocation, each process is contained in a single contiguous block of memory.
- Discuss -
  - Memory Mapping and Protection
  - Partitioning
  - Allocation Policies
  - Performance Parameters
    - Fragmentation
    - Maximum Process Size
    - Degree of Multiprogramming
    - Allocation Policy

Memory Mapping and Protection

- These features can be provided by using a relocation register together with a limit register.
  - 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).



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

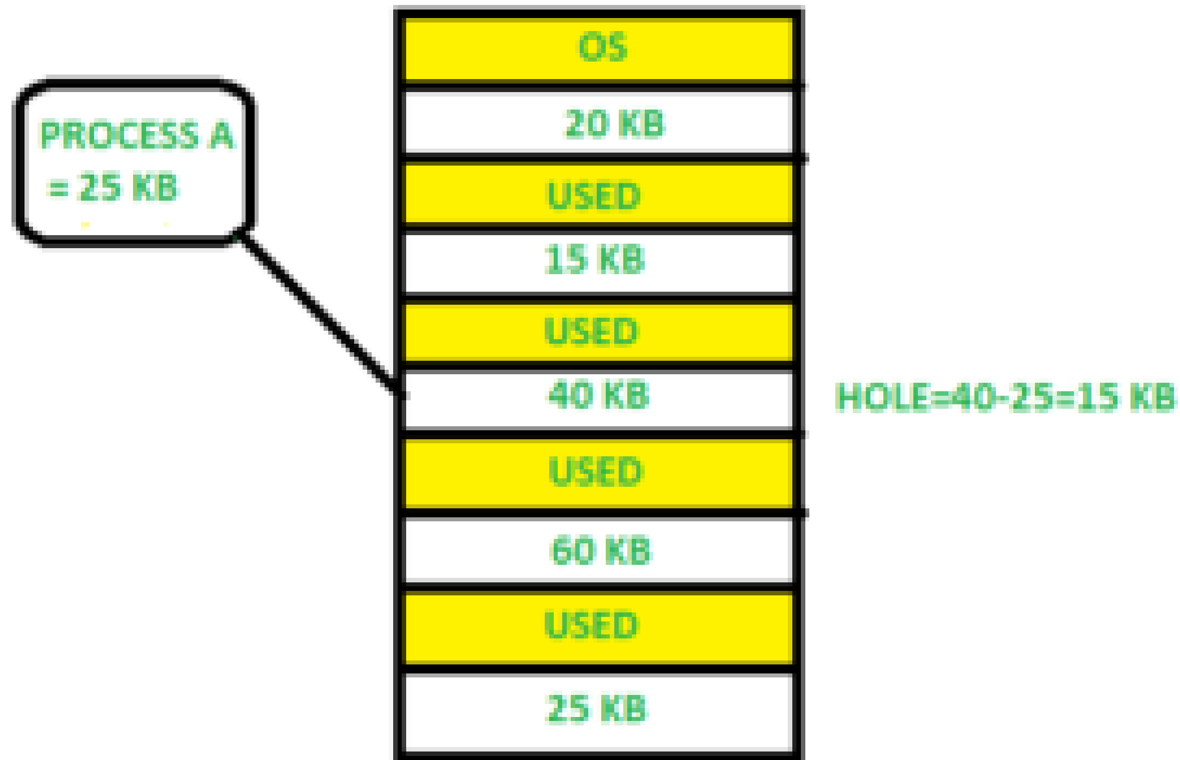
Memory Allocation

- Dividing memory into several partitions is one of the simplest methods for allocating memory.
  - Each partition may contain exactly one process.
  - 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.
- Partitioning can be done in two ways:
  - Fixed (Static) Partitioning
  - Variable (Dynamic) Partitioning

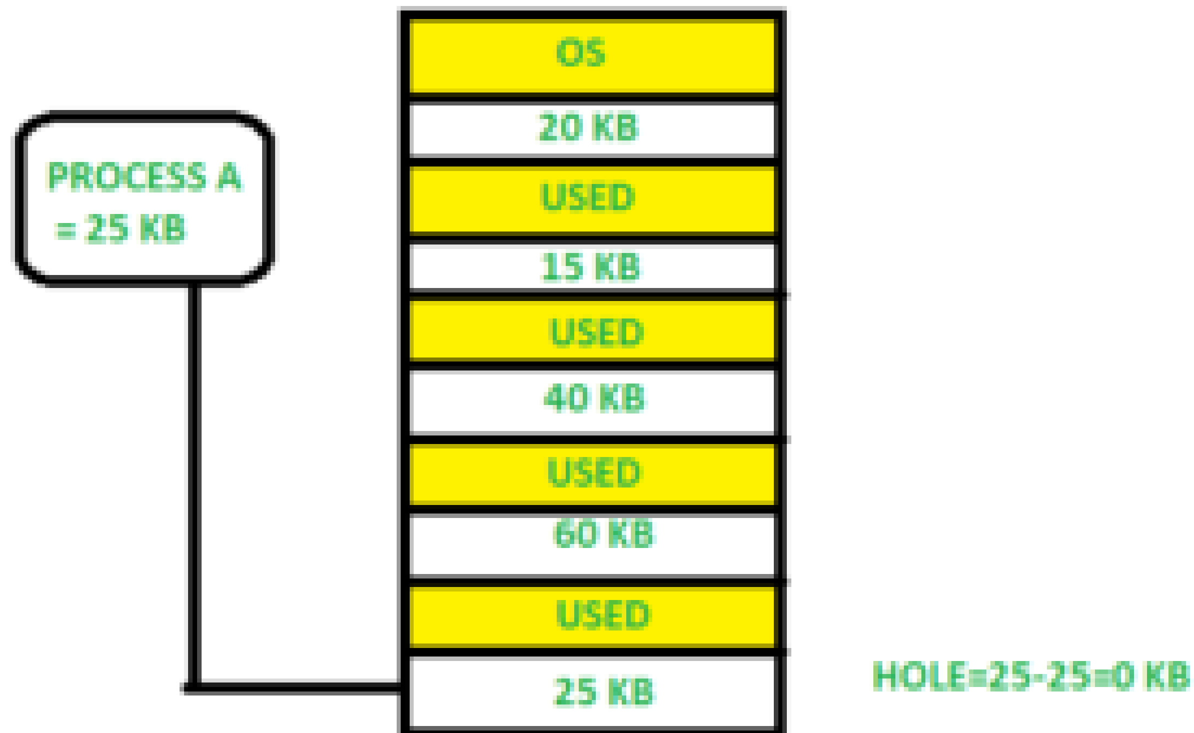
Memory Allocation Policies

- The memory blocks available comprise a set of holes of various sizes scattered throughout memory. When a process arrives and needs memory, the system can search an appropriate hole with following policies:
  - 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.
  - 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.
  - Next-Fit - It is similar to the first fit but it will search for the first sufficient partition from the last allocation point.

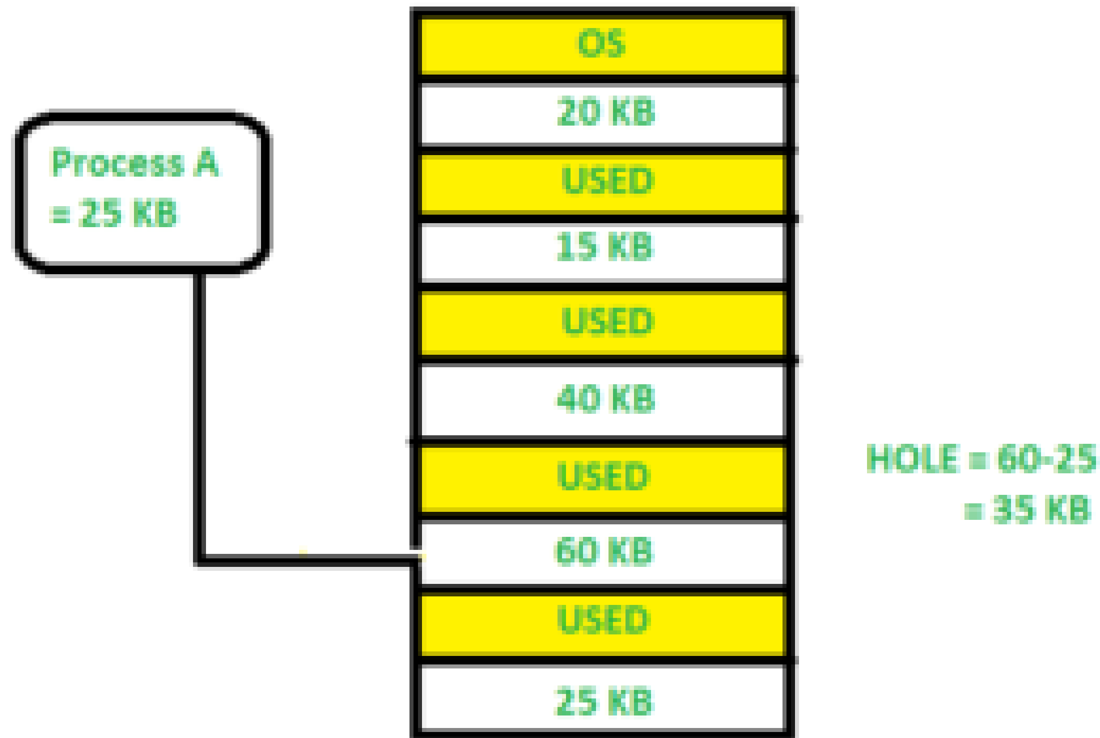
Memory Allocation Policies: First Fit



Memory Allocation Policies: Best-Fit



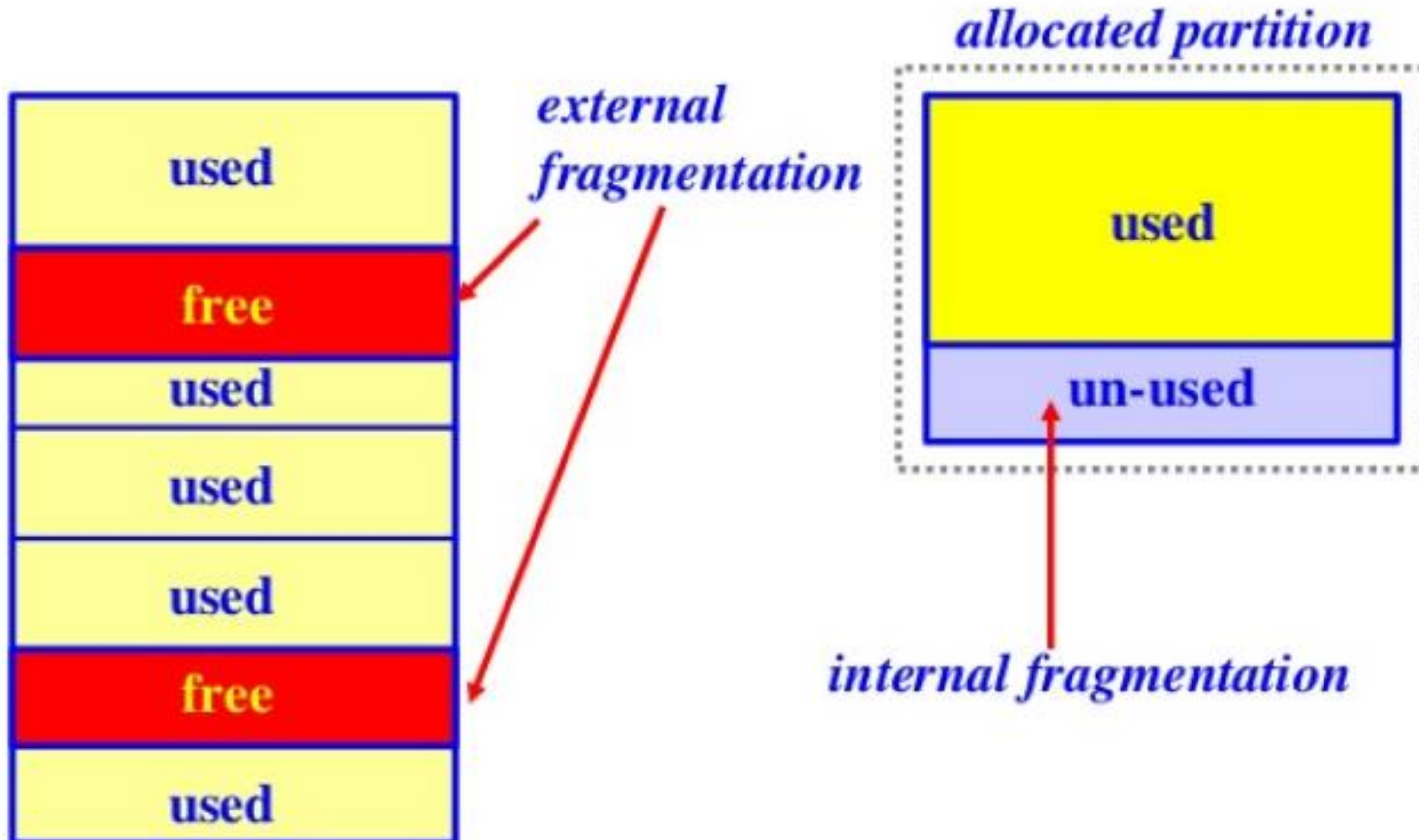
Memory Allocation Policies: Worst-Fit



Fragmentation

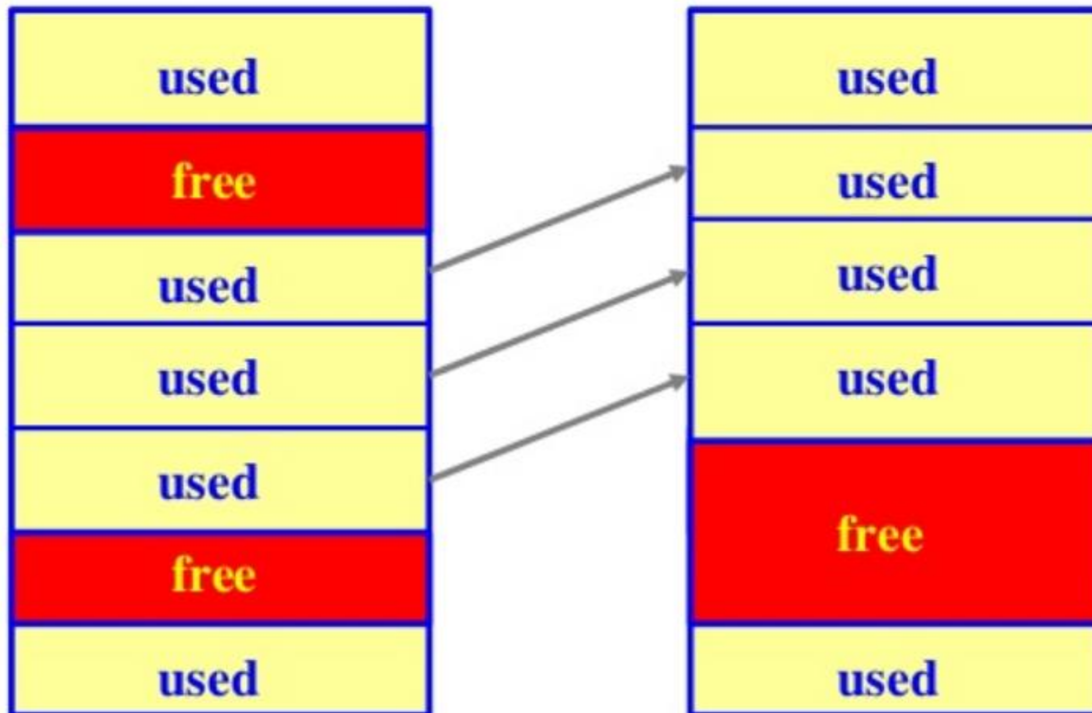
- Internal Fragmentation
  - Memory block allocated is slightly larger than request memory, therefore, some portion of memory is left unused, as it cannot be used by another process.
  - The internal fragmentation can be reduced by effectively assigning the smallest partition but large enough for the process.
- External Fragmentation
  - Total memory space exists to satisfy a request, but it is not contiguous.
  - The external fragmentation may be reduced by compaction (also known as defragmentation) – shuffle memory contents to place all free memory together in one large block.
  - **Compaction is possible only if relocation is dynamic, and is done at execution time.**

Fragmentation



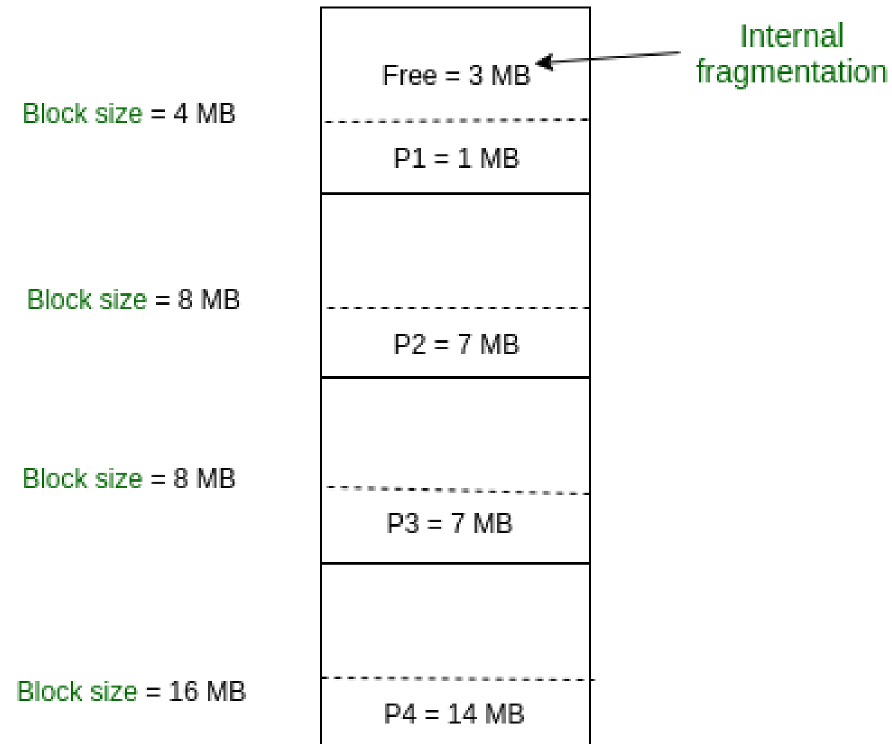
Compaction

- If processes are relocatable, the used memory blocked may be moved together to make a larger free memory block.



Fixed Partitioning

- In this partitioning, number of partitions in memory 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.
- Sum of internal fragmentation in every block = (4-1)+(8-7)+(8-7)+(16-14) = 3 + 1 + 1 + 2 = 7MB.
- Suppose a process P5 of size 7MB comes. But, this process cannot be accommodated inspite of available free space because of contiguous allocation. Hence, 7MB becomes part of External Fragmentation.



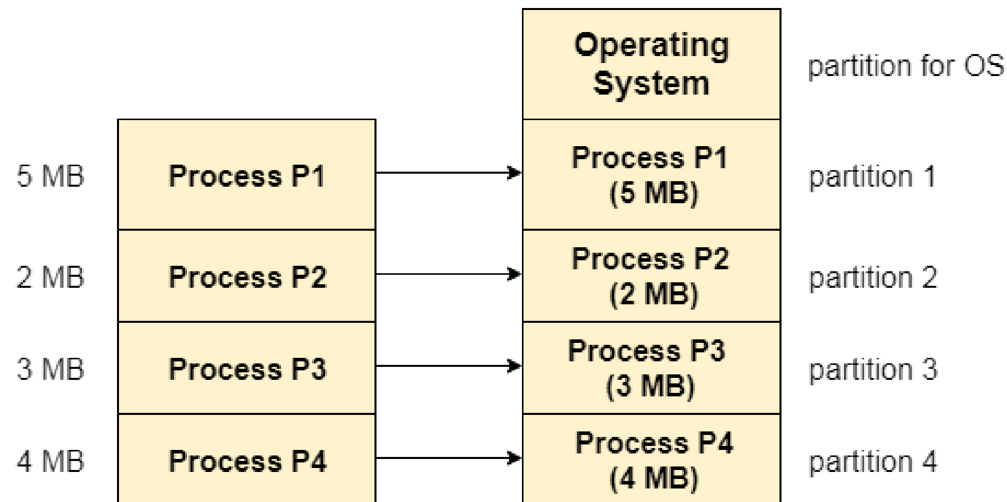
BHARATI VIDYAPEETH PUNE

Advantages/Disadvantages of Fixed Partitioning

- Advantages
  - Easy to implement
  - Little OS overhead
- Disadvantages
  - Internal fragmentation
  - External fragmentation
  - Limit process size - Process of size greater than size of partition in Main Memory cannot be accommodated.
  - Limitation on degree of multiprogramming
- Best Allocation Policy
  - Best-Fit

Variable Partitioning

- This partitioning tries to overcome the problems caused by fixed partitioning.
- In this technique, the partition size is not declared initially.
- It is declared at the time of process loading.
- The partition size varies according to the need of the process so that the internal fragmentation can be avoided.
- The size of each partition will be equal to the size of the process.



Advantages/Disadvantages of Variable Partitioning

- Advantages
  - No internal fragmentation
  - No limitation on the size of the process
  - Degree of multiprogramming is dynamic
- Disadvantages
  - External fragmentation
- Best Allocation Policy
  - Worst-Fit

Advantages/Disadvantages of Variable Partitioning

- External Fragmentation
  - Suppose, Process P1 (2MB) and process P3 (1MB) completed their execution. Hence two spaces are left, i.e. 2MB and 1MB.
  - Let's suppose process P5 of size 3MB comes. The empty space in memory cannot be allocated as no spanning is allowed in contiguous allocation.
  - The rule says that process must be contiguously present in main memory to get executed.
  - Hence, variable partitioning may result in external fragmentation.

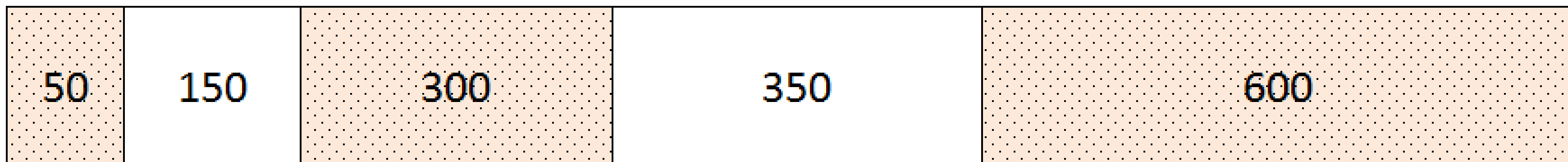
Dynamic partitioning

Memory	Block size
Operating system	
P1 (2 MB) executed, now empty	Block size = 2 MB
P2 = 7 MB	Block size = 7 MB
P3 (1 MB) executed	Block size = 1 MB
P4 = 5 MB	Block size = 5 MB
Empty space of RAM	

Partition size = process size
So, no internal Fragmentation

Example - 1

Consider the following heap in which blank regions are not in use and shaded regions are in use.



The sequence of requests for blocks of size 300, 25, 125, 50 can be satisfied if we use:

a) either First-fit or Best-fit policy.

b) First-fit but not Best-fit

c) Best-fit but not First-fit

d) None of these

Example - 2

Consider a system in which the memory consists of the following free holes:

Sizes in Memory Order: 15K, 5K, 20K, 4K and 7K

Which hole is taken for successive segment request of 12K, 7K and 15K for Next-Fit. Assuming that the last process was swapped in just before the 5K hole.

a) 15K, 20K, Out of Memory

b) 15K, 7K, 20K

c) 20K, 7K, 15K

d) 15K, 7K, 20K

Paging

- Paging is a memory-management scheme that permits the physical address space of a process to be non-contiguous.
- Paging avoids external fragmentation and the need for compaction.
- Paging in its various forms is used in most operating systems.
- Implementation of paging involves dividing the process into blocks of the same size called pages which are mapped to same size blocks on physical memory called frames.
  - Physical Memory Blocks --> Frames
  - Logical Memory Blocks --> Pages

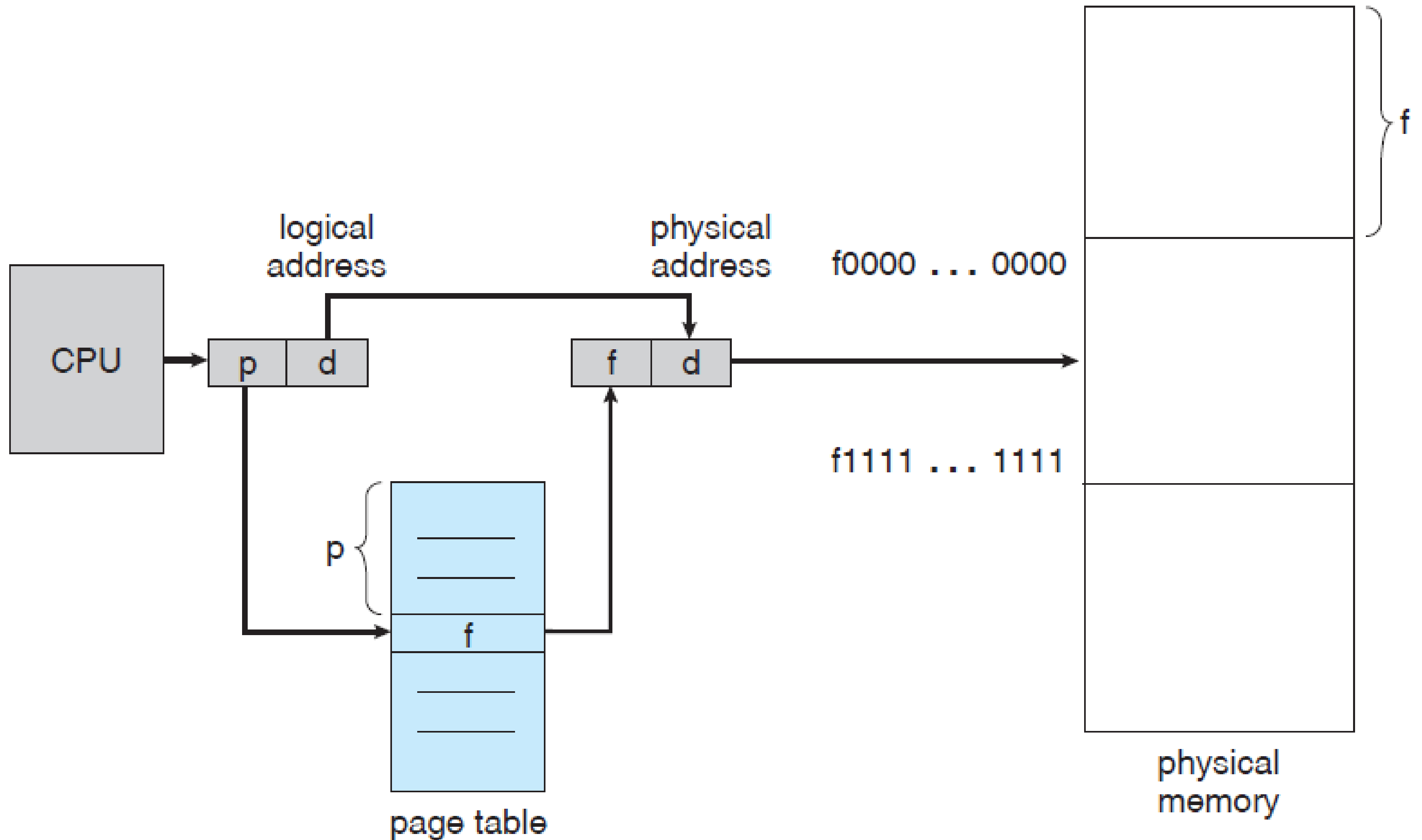
Paging (contd…)

- In every non-contiguous memory management technique, we need to consider the following points:
  - Organization of Logical Address Space (LAS)
  - Organization of Physical Address Space (PAS)
  - Organization of Memory Management Unit (MMU)
  - Translation Algorithm

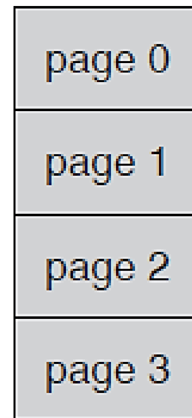
Paging (contd…)

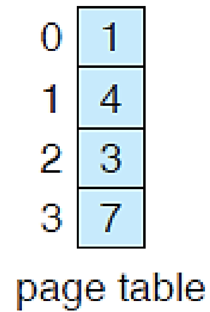
- **Address generated by CPU is divided into:**
  - Page number (p): Number of bits required to represent the pages in Logical Address Space.
  - Page offset (d): Number of bits required to represent particular word in a page or word number of a page.
- **Physical Address is divided into:**
  - Frame number (f): Number of bits required to represent the frame of Physical Address Space.
  - Frame offset (d): Number of bits required to represent particular word in a frame or word number of a frame.

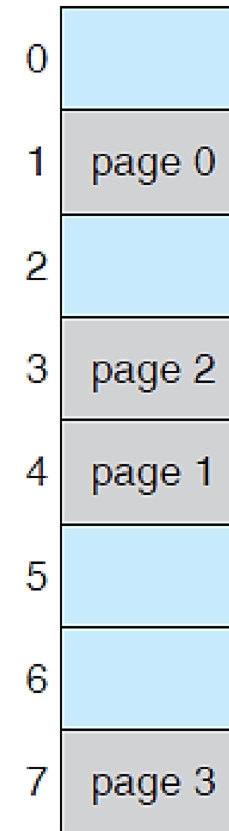
Paging (contd…)



Paging (contd…)







Paging (contd…)

Let, LAS = 8 KB, PAS = 4 KB, PS (Page Size) = 1KB, 1Word = 1 Byte

Note: LAS = 2^{LA} and PAS = 2^{PA}, LAS & PAS is in words and LA & PA is in bits.

Organization of Logical Address Space (LAS)

- LAS is divided into equal size pages.
- Page size in power of 2 --> (2^k).
- Number of pages (N) = $\frac{\text{LAS}}{\text{PS}} = \frac{8\text{KB}}{1\text{KB}} = 8$
- Page number (p) = $\lceil \log_2 N \rceil$=3 (To represent 8 pages, we need 3 bit number)
- Page offset (To point a particular word in a page) (d) = $\lceil \log_2 \text{PS} \rceil = 10$
- Therefore, LA is divided into **p + d** ---> **3 + 10 = 13**

Paging (contd…)

Let, LAS = 8 KB, PAS = 4 KB, PS (Page Size) = 1KB, 1Word = 1 Byte

Note: $LAS = 2^{LA}$ and $PAS = 2^{PA}$, LAS & PAS is in words and LA & PA is in bits.

Organization of Physical Address Space (PAS)

- LAS is divided into equal size frames.
- Frame size = Page size
- Number of frames (M) = $\frac{PAS}{PS} = \frac{4KB}{1KB} = 4$
- Frame number (f) = $\lceil \log_2 M \rceil$=2 (To represent 4 frames, we need 2 bit number)
- Frame offset (To point a particular word in a frame) (d) = $\lceil \log_2 FS \rceil = 10$ bits

  (Page size and frame size are equal)
- Therefore, PA is divided into **f + d ---> 2 + 10 = 12**

Paging (contd…)

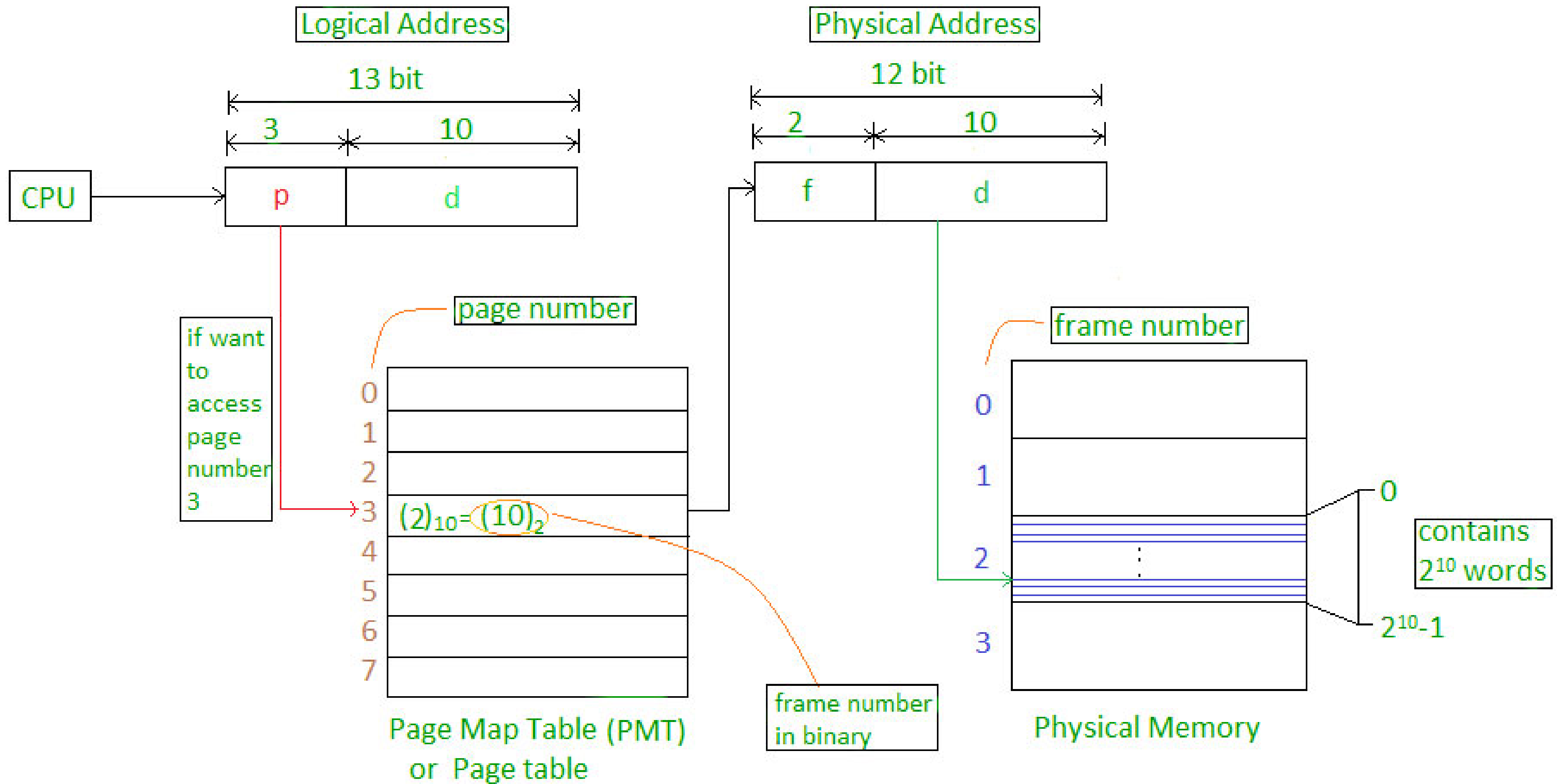
Let, LAS = 8 KB, PAS = 4 KB, PS (Page Size) = 1KB, 1Word = 1 Byte

Note: LAS = 2^{LA} and PAS = 2^{PA}, LAS & PAS is in words and LA & PA is in bits.

Organization of Memory Management Unit (MMU)

- In paging, it is **Page Table**.
- Number of entries in Page Table is equal to number of pages in LAS.
- Page table contains frame number and other bits (valid/invalid bit, protection bit, modified bit, etc.)
- **Each process has its page table.**
- Page table also resides in main memory.
- Page Table Size = N * e (entry size)

Paging (contd…)



Paging: Example - 1

- Consider a system supporting,
  - LA = 32 Bits,
  - PA = 27 Bits,
  - PS = 4KB,
  - Page Table Entry Size (e) = 3 Byte
- What is Page Table Size?

Page Table Size $= N \times e$

$$N = \frac{LAS}{Page\ Size} = \frac{2^{LA}}{Page\ Size} = \frac{2^{32}}{2^{12}}$$

Page Table Size $= \frac{2^{32}}{2^{12}} \times 3 = 2^{20} \times 3 = 1024 \times 1024 \times 3$ Bytes

= 31,45,728 Bytes → 3 MB

Paging: Example - 2

- Consider a system with 2K Pages, 512 Frames, Page Offset = 9 Bits, and e = 4 Bytes. Determine p, f, LAS, PAS and Page Table Size.

Given, Number of Pages (N) = 2K = 2048 = 2^{11}

Number of Frames (M) = 512

Page Offset (d) = 9 Bits

Each Entry Size of Page Table (e) = 4 Bytes

$$p = \lceil log_2 N \rceil = \lceil log_2 2048 \rceil = 11$$

$$f = \lceil log_2 M \rceil = \lceil log_2 512 \rceil = 9$$

$$LAS = 2^{LA} = 2^{(p+d)} = 2^{(11+9)} = 2^{20}$$

$$PAS = 2^{PA} = 2^{(f+d)} = 2^{(9+9)} = 2^{18}$$

$$Page\ Table\ Size = N \times e = 2^{11} \times 4 = 2048 \times 4 \text{ Bytes} = 8192 \text{ Bytes} = 8\text{KB}$$

Paging: Example - 3

- Consider a system with **35 Bits LA**, **32 Bits PA**, **4KB Page Size**, and each Table Entry contains **2 Protection Bits**, **1 Valid/Invalid Bit**, and **1 Modified Bit** along with Frame Number. What is Page Table Size?

Size of Each Entry in Table

Bits to Represent **Frame No. (f)**	Protection Bits	Valid/Invalid Bit	Modified Bit

No. of Pages (**N**) = $\frac{LAS}{Page\ Size}$ = $\frac{2^{LA}}{Page\ Size}$ = $\frac{2^{35}}{2^{12}}$ = 2^{23}

No. of Frames (**M**) = $\frac{PAS}{Frame\ Size}$ = $\frac{2^{PA}}{Frame\ Size}$ = $\frac{2^{32}}{2^{12}}$ = 2^{20}

No. of Bits required to Represent Frame No. (**f**) $= \lceil log_2 M \rceil = \lceil log_2 2^{20} \rceil = 20$

Size of Each Entry in Table = 20 + 2 + 1 + 1 = 24 Bits

Page Table Size = $N \times e = 2^{23} \times 3$ Bytes $= 83{,}88{,}608 \times 3$ Bytes $=$ **24 MB**

Hardware Implementation for Page Table

- **Approach - 1**
  - The page table can be implemented as a set of dedicated registers.
  - These registers should be built with very high-speed logic to make the paging-address translation efficient. Every access to memory must go through the paging map, so efficiency is a major consideration.
  - The use of registers for the page table is satisfactory if the page table is reasonably small (for example, 256 entries).
  - Most contemporary computers, however, allow the page table to be very large (for example, 1 million entries).
  - For these machines, the use of fast registers to implement the page table is not feasible.

Hardware Implementation for Page Table

- **Approach - 2**
  - The page table is kept in main memory, and a page-table base register (PTBR) points to the page table.
  - The problem with this approach is the time required to access a memory location.
  - If we want to access location **i**, we must first index into the page table, using the value in the PTBR offset by the page number for **i**.
  - It provides us with the frame number; we can then access the desired place in memory.
  - In this approach, two memory accesses are needed to access a byte (one for the page-table entry, one for the byte). Thus, memory access is slowed by a factor of 2.

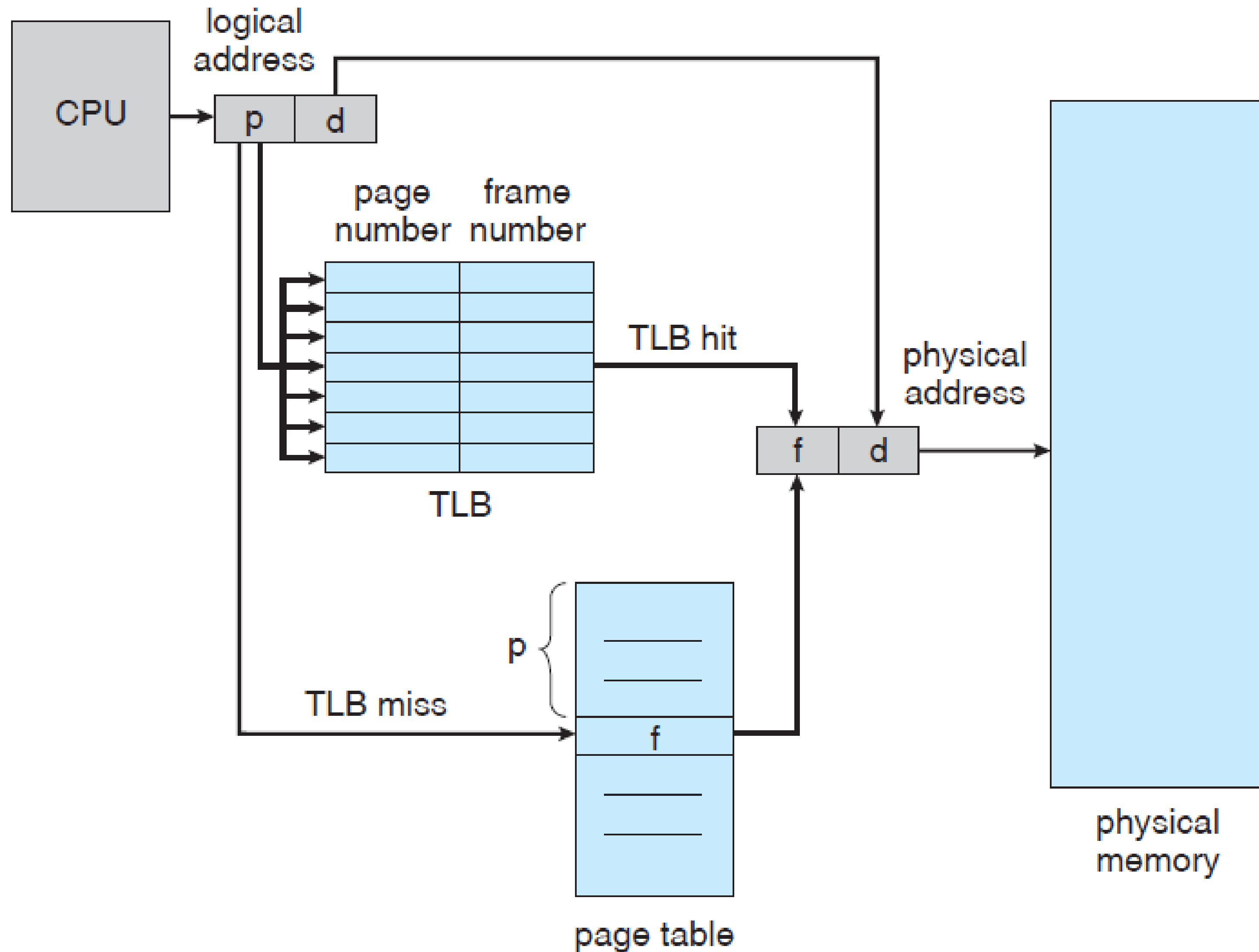
Hardware Implementation for Page Table

- **Approach – 3 (Standard Approach)**
  - Use of a special, small, fast lookup hardware cache, called a translation look-aside buffer (TLB).
  - The TLB is associative, high-speed memory.
  - TLB consists of two columns: **Page Number (Key)** and **Frame Number (Value)**.
  - **The item is compared with all keys simultaneously.**
    - If the item is found, the corresponding value field is returned.
    - The search is fast; the hardware is expensive.
    - Typically, the number of entries in a TLB is small, often numbering between 64 and 1,024.

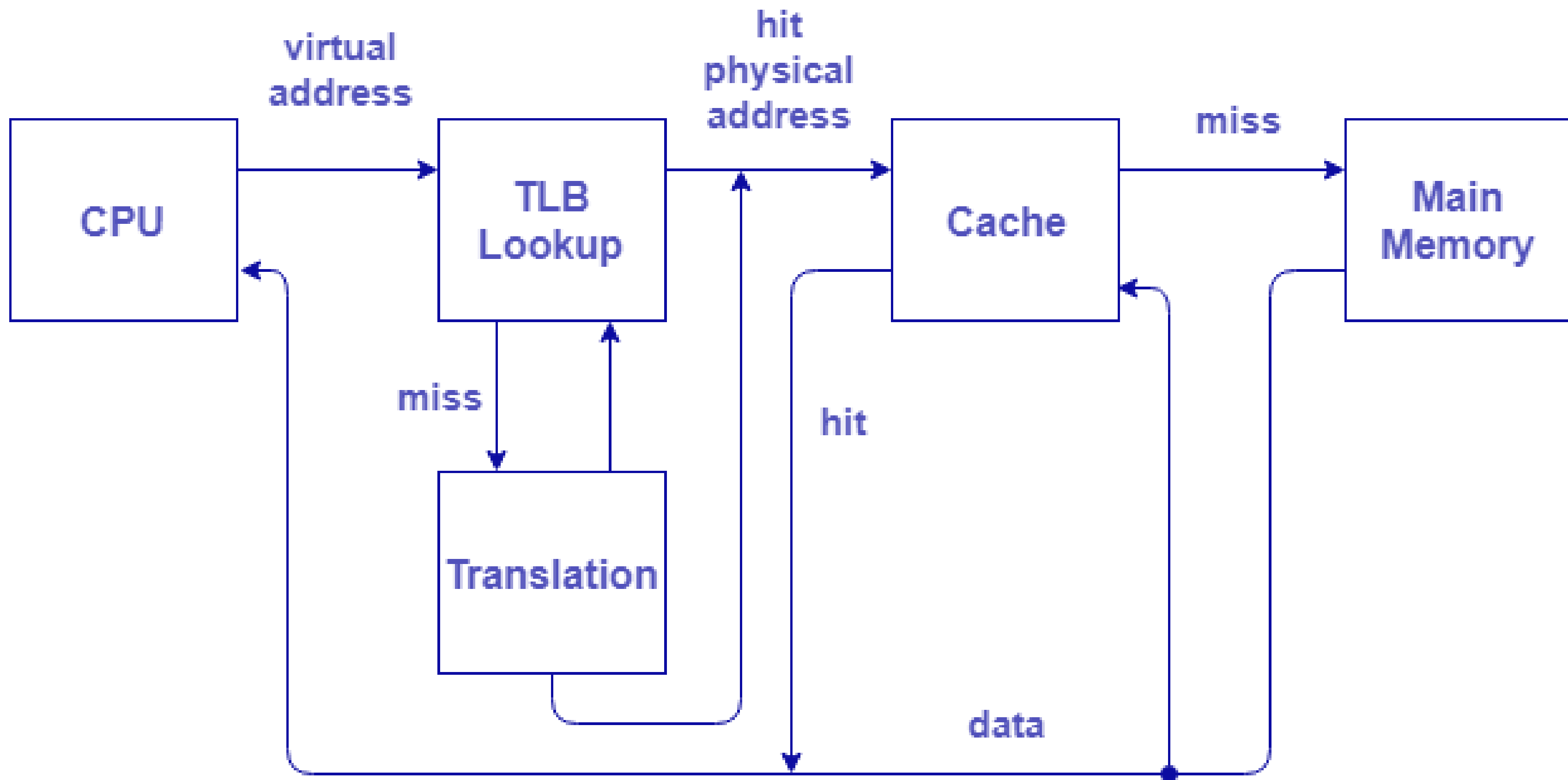
Using TLB with Page Table

- 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, its frame number is immediately available and is used to access memory.
  - If the page number is not in the TLB **(known as a TLB miss)**, a memory reference to the page table must be made. In addition, we add the page number and frame number to the TLB, so that they will be found quickly on the next reference.
  - If the TLB is already full of entries, the operating system must select one for replacement. Replacement policies range from least recently used (LRU) to random. Furthermore, some TLBs allow certain entries to be wired down, meaning that they cannot be removed from the TLB. Typically, TLB entries for kernel code are wired down.

Using TLB with Page Table



CPU Cache and TLB



Effective Access Time

- The percentage of times that a particular page number is found in the TLB is called the **hit ratio**.

- For example, an 80-percent hit ratio means that we find the desired page number in the TLB, 80 percent of the time.

  - If it takes 20 nanoseconds to search the TLB and 100 nanoseconds to access memory, then a mapped-memory access takes 120 nanoseconds when the page number is in the TLB.

  - If we fail to find the page number in the TLB (20 nanoseconds), then we must first access memory for the page table and frame number (100 nanoseconds) and then access the desired byte in memory (100 nanoseconds), for a total of 220 nanoseconds.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh

Effective Access Time - Example

Effective Access Time =

Hit Ratio * (Access Time of TLB + Access Time of Memory)

+

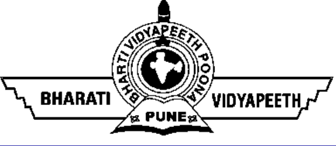
Miss Ratio * (Access Time of TLB + Access Time for Page Table + Access Time of Memory)

Question 1: A paging scheme uses a Translation Look-a-side buffer (TLB). A TLB access takes 10 ns and a main memory access takes 50 ns. What is the effective access time (in ns) if the TLB hit ratio is 90% and there is no page fault?

Answer: 54, 60, 65, 75

Question 2: A paging scheme uses a Translation Look-a-side buffer (TLB). The effective memory access takes 160 ns and a main memory access takes 100 ns. What is the TLB access time (in ns) if the TLB hit ratio is 60% and there is no page fault?

Answer: 54, 60, 20, 75



Protection in Paging: Protection Bit

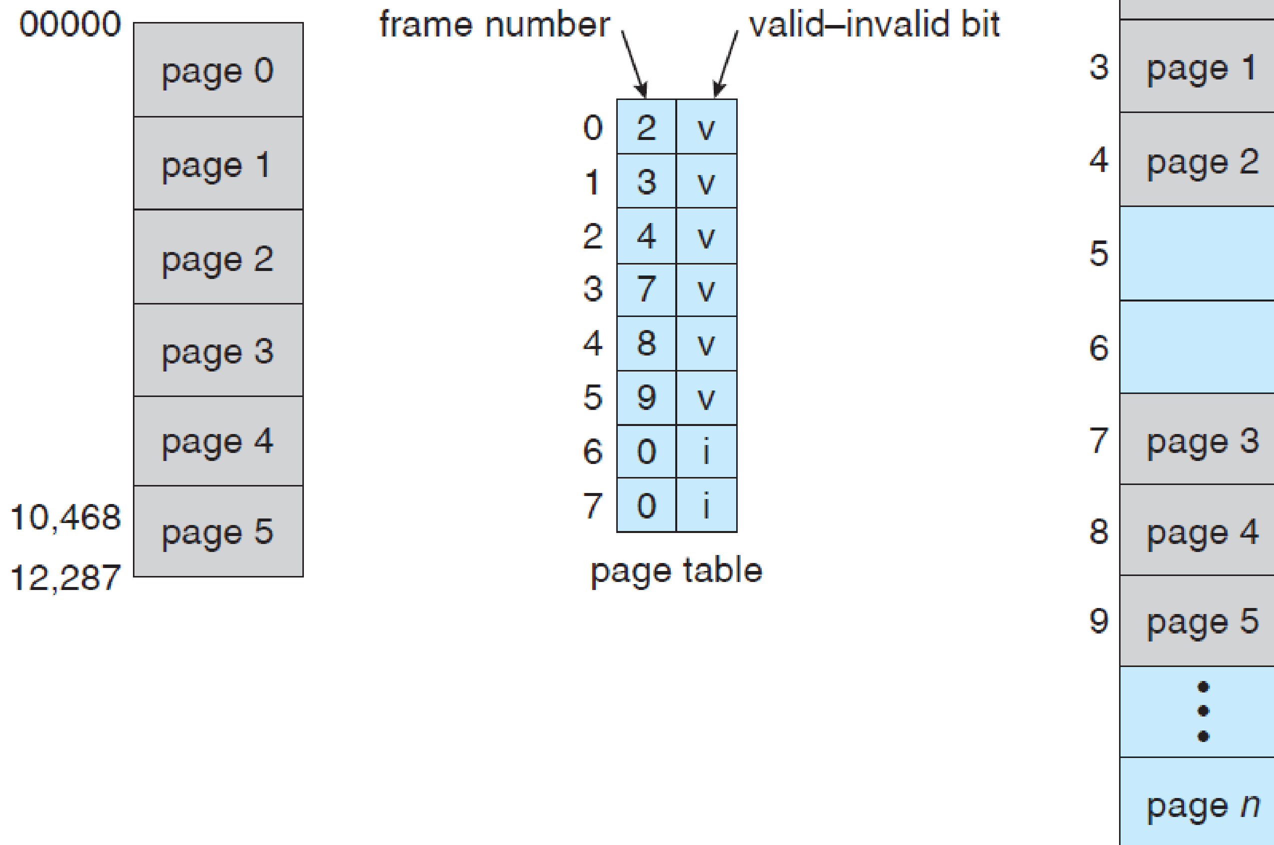
- Memory protection in a paged environment is accomplished by protection bit associated with each frame.

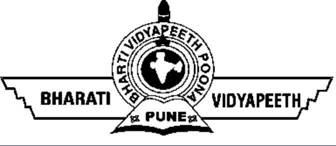
- One bit can define a page to be read–write or read-only.
  - At the same time that the physical address is being computed, the protection bits can be checked to verify that no writes are being made to a read-only page.
  - An attempt to write to a read-only page causes a hardware trap to the operating system.

Protection in Paging: Valid–Invalid Bit

- One additional bit is generally attached to each entry in the page table: a valid–invalid bit.
  - When this bit is set to "valid," the associated page is in the process's logical address space and is thus a legal (or valid) page.
  - When the bit is set to "invalid," the page is not in the process's logical address space.
  - The OS sets this bit for each page to allow or disallow access to the page.

Protection in Paging: Valid–Invalid Bit





Structure of Page Table

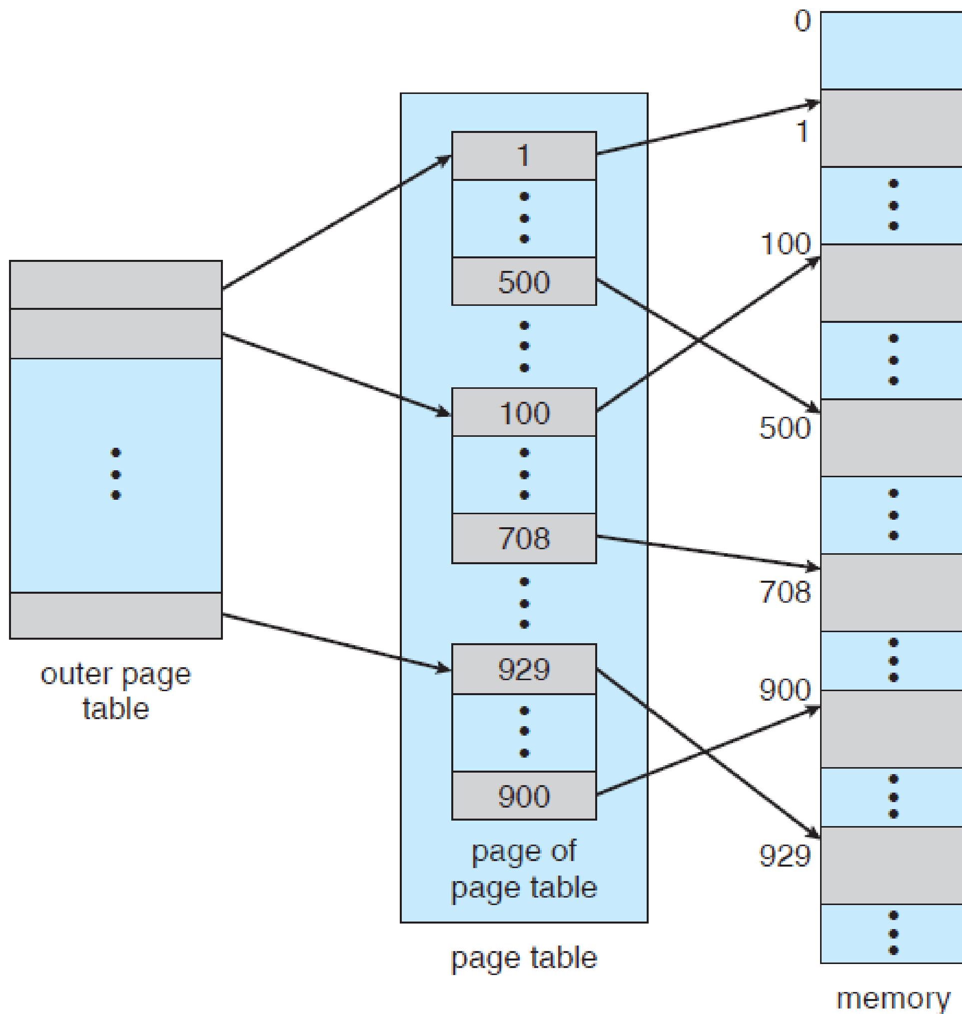
- Hierarchical (Multi-Level) Paging
- Hashed Page Tables
- Inverted Page Tables

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh

Hierarchical (Multi-Level) Paging

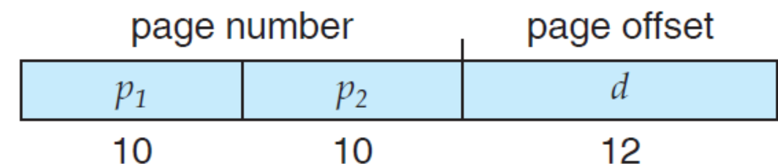
- Consider a system with a 32-bit logical address space.
  - If the page size in such a system is 4 KB (2^{12}), then a page table may consist of up to 1 million entries $2^{32}/2^{12}$).
  - Assuming that each entry consists of 4 bytes, each process may need up to 4MB of physical address space for the page table alone.
  - If we would not want to allocate the page table contiguously in main memory, the solution is to divide the page table into smaller pieces.
    - One way is to use a two-level paging algorithm, in which the page table itself is also paged.

Hierarchical (Multi-Level) Paging



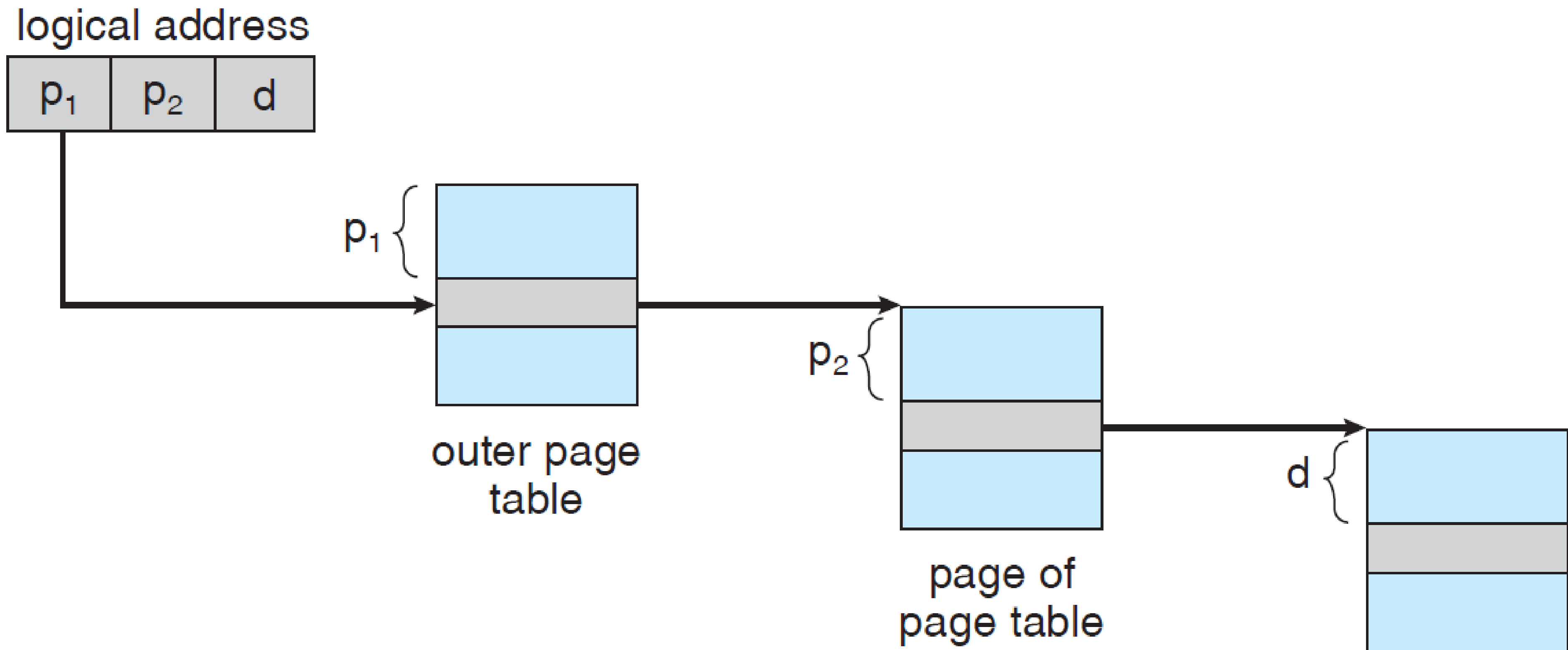
A two-level page-table scheme

- Consider a system with a 32-bit logical address space and 4KB page size:
  - A logical address is divided into a page number consisting of 20 bits and a page offset consisting of 12 bits. Because we page the page table, the page number is further divided into a 10-bit page number and a 10-bit page offset.
  - Thus, a logical address is as follows:



Two-Level 32-Bit Paging Architecture

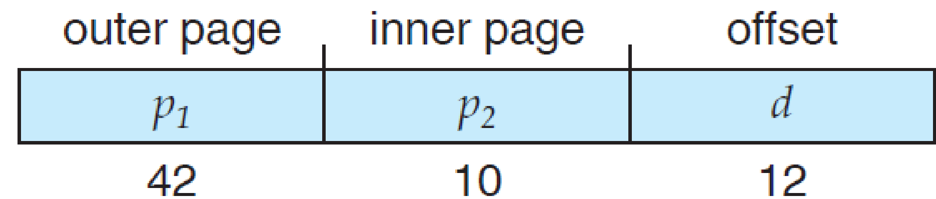
Address Translation Process

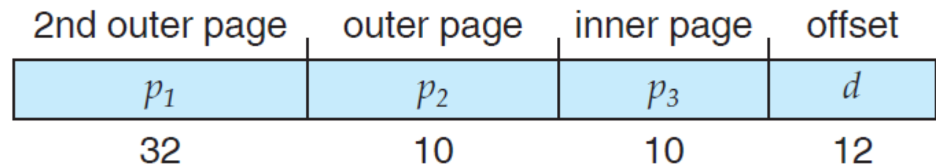


Hierarchical (Multi-Level) Paging

For a system with a 64-bit logical address space, is a two-level paging scheme appropriate?

- To illustrate this point, let us suppose that the page size in such a system is 4 KB (2^{12}).
- In this case, the page table consists of up to 2^{52} entries.
- If we use a two-level paging scheme, then the inner page tables can conveniently be one page long, or contain 2^{10} 4-byte entries.



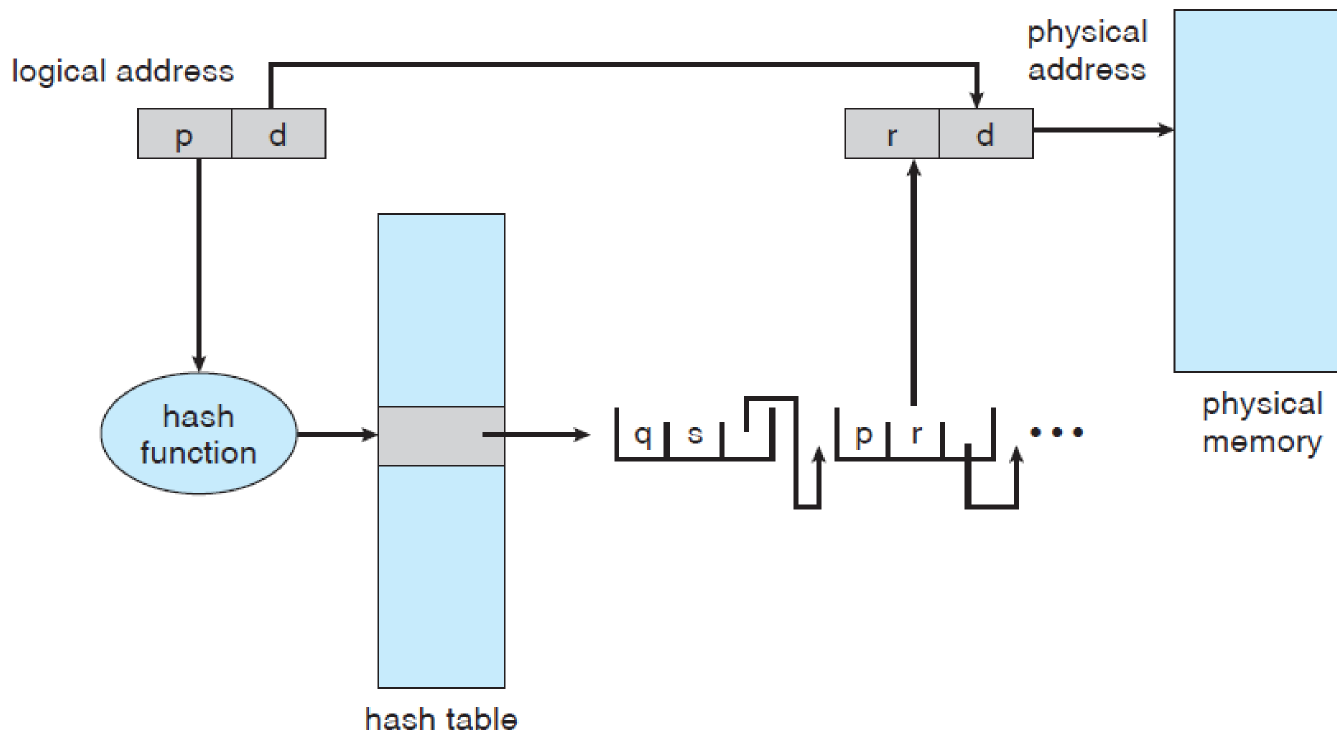


The outer page table is still 234 bytes in size.

The next step would be a four-level paging scheme,

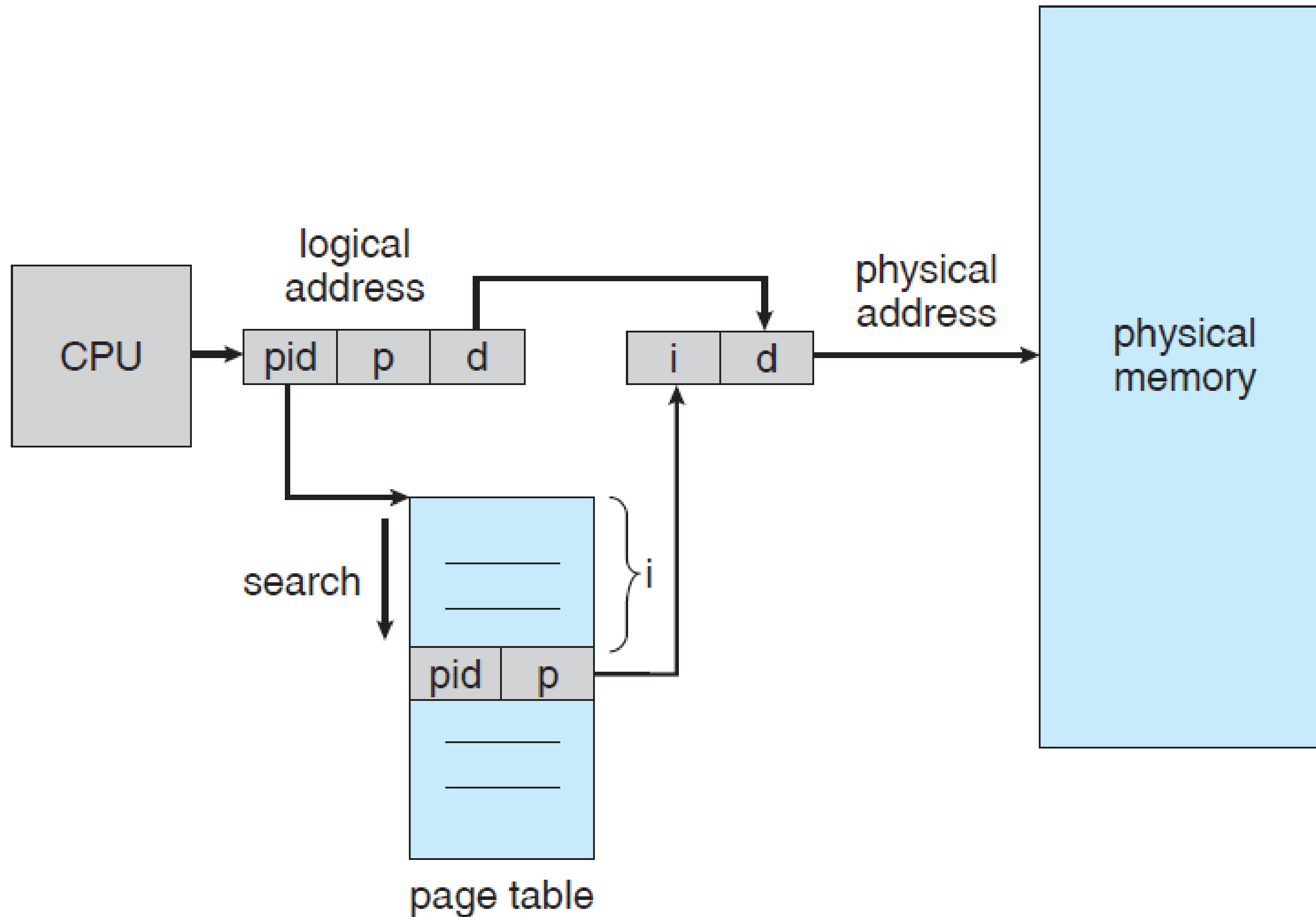
Hashed Page Tables

- A common 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.
- Each entry in the hash table contains a linked list of elements that hash to the same location (to handle collisions).



Inverted Page Table

- An inverted page table has one entry for each real page (or frame) of memory.

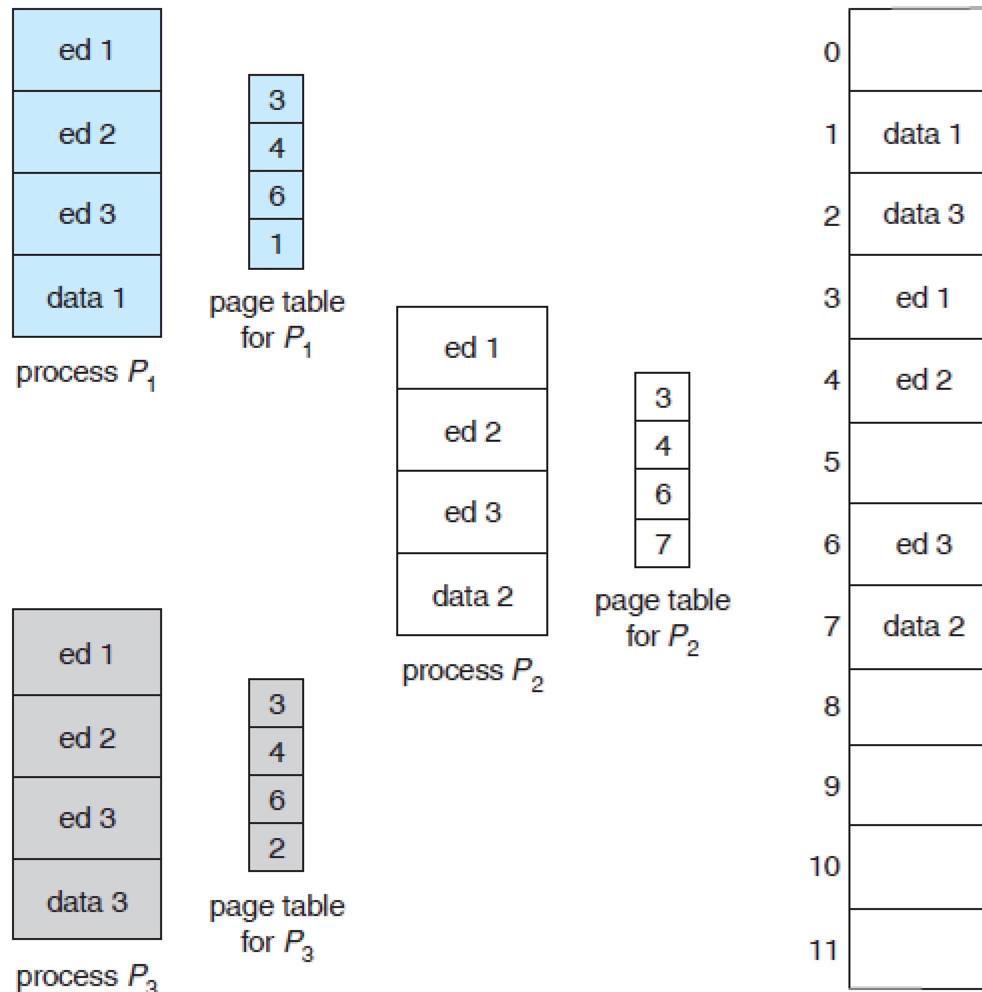


Shared Pages

- An advantage of paging is the possibility of sharing common code, particularly in a time-sharing environment.
- Consider a system that supports 40 users, each of whom executes a text editor. If the text editor consists of 150 KB of code and 50 KB of data space, we need 8,000 KB to support the 40 users.
- In shared mode, only one copy of the editor need be kept in physical memory.
- Each user's page table maps onto the same physical copy of the editor, but data pages are mapped onto different frames.
- Thus, to support 40 users, we need only one copy of the editor (150 KB), plus 40 copies of the 50 KB of data space per user. The total space required is now 2,150 KB instead of 8,000 KB—a significant savings.

Shared Pages (contd…)

Sharing of Code in a Paging Environment

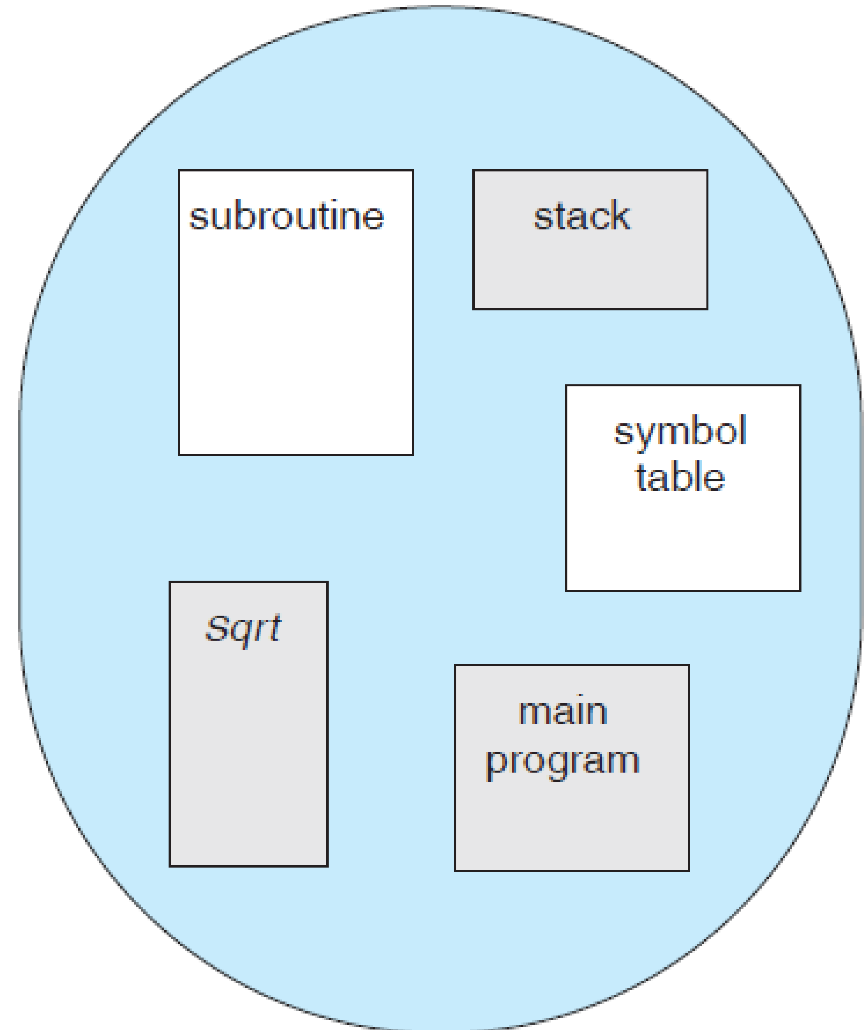


Limitations of Paging

- Internal Fragmentation
  - If the memory requirements of a process do not happen to coincide with page boundaries, the last frame allocated may not be completely full.
  - **Example**: If page size is 2048 bytes, a process of 72,766 bytes will need 35 pages plus 1,086 bytes. It will be allocated 36 frames, resulting in internal fragmentation of 2,048 − 1,086 = 962 bytes.
  - If process size is independent of page size, we expect internal fragmentation to average one-half page per process.
  - This consideration suggests that small page sizes are desirable. However, overhead is involved in each page-table entry, and this overhead is reduced as the size of the pages increases. Also, disk I/O is more efficient when the amount of data being transferred is larger.
- Larger Access Time
- Memory Required for Page Table

Segmentation

- In Paging, the user's view of memory is not the same as the actual physical memory.
- Users do not think of memory as a linear array of bytes, some containing instructions and others containing data.
- Rather, users prefer to view memory as a collection of variable-sized segments (methods, stack, etc.), with no necessary ordering among segments.



User's View of a Program

Segmentation (contd…)

- Segmentation is a memory-management scheme that supports user's view of memory.
- A logical address space is a collection of segments.
- Each segment has a name and a length.
- The addresses specify both the segment number and the offset within the segment.
  - 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, the fifth instruction of the Sqrt(), and so on.

BHARATI VIDYAPEETH PUNE

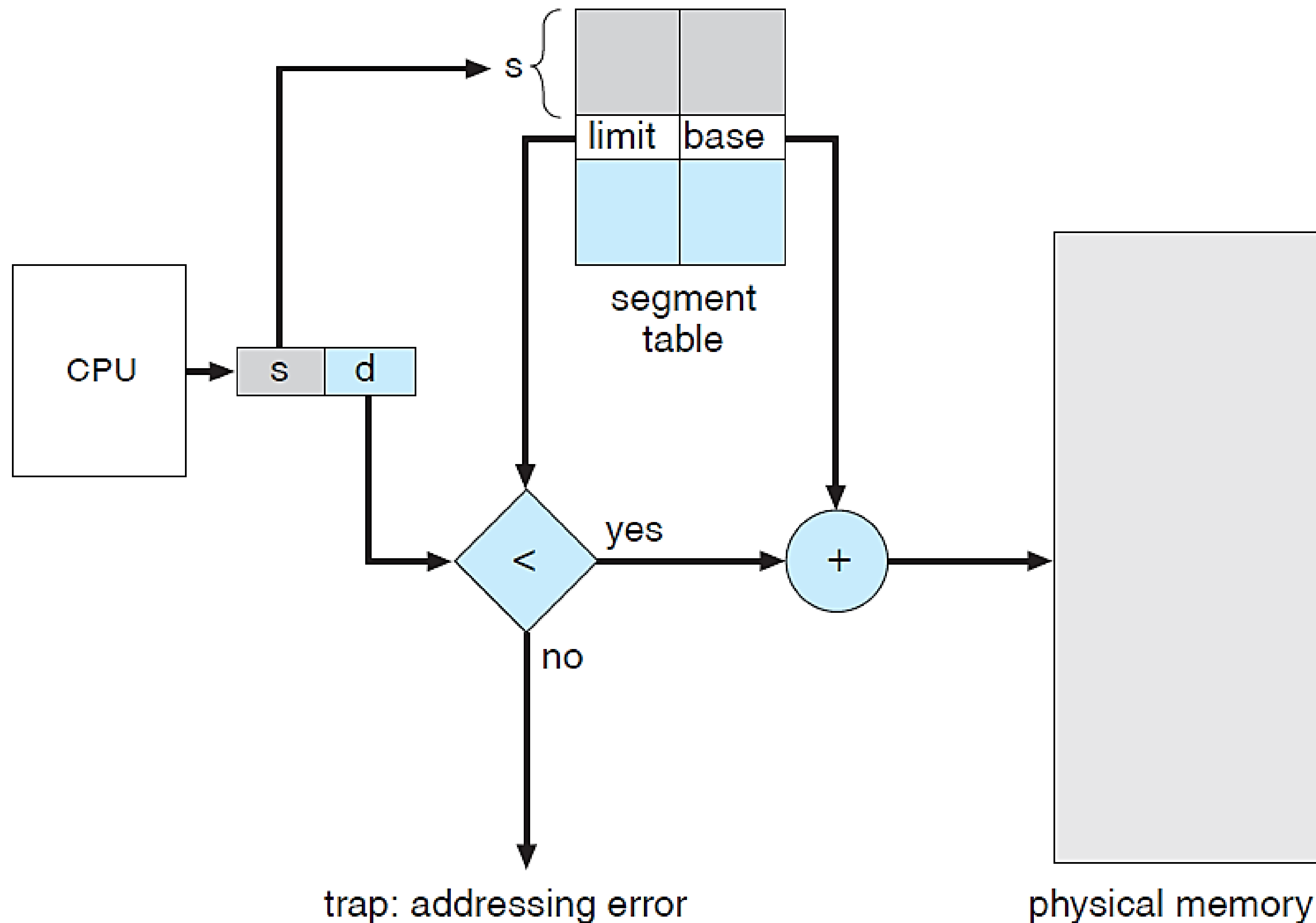
Segmentation (contd…)

- While compiling, 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
- The loader takes all these segments and assign them segment numbers.

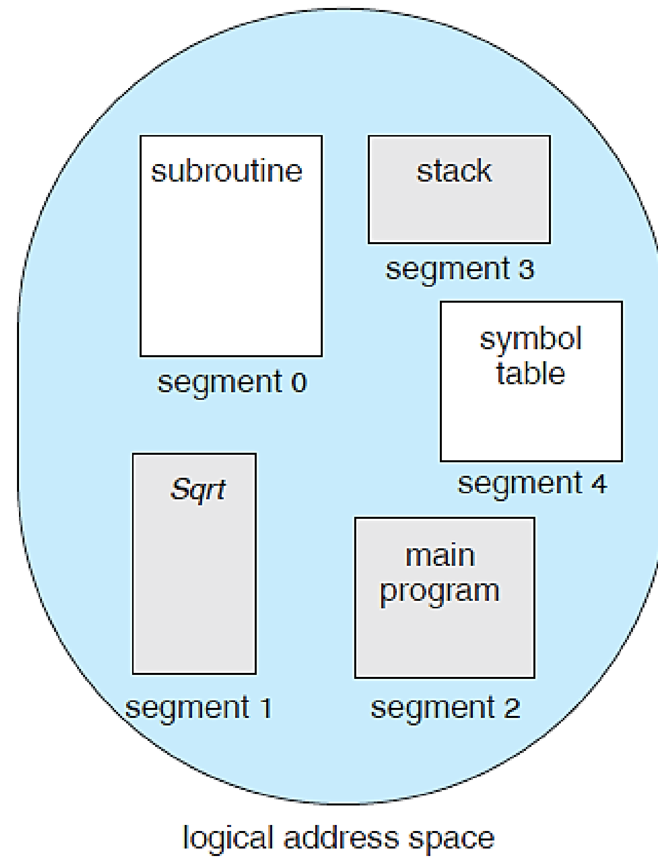
Segmentation (contd…)

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

Segmentation (contd…)

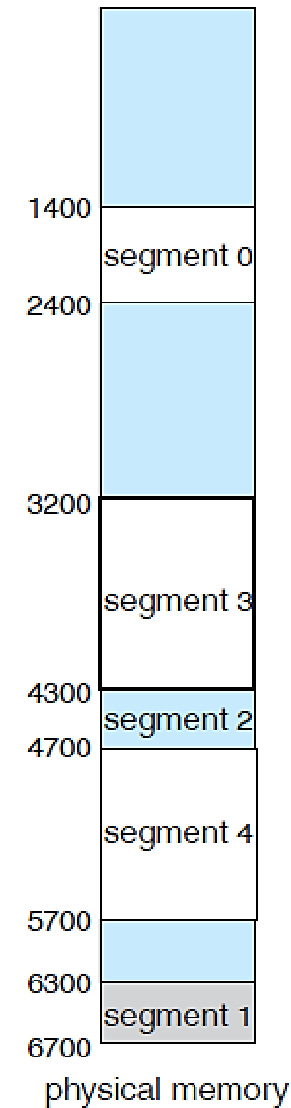


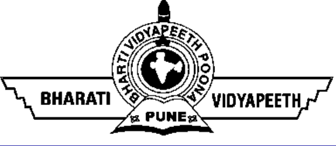
Segmentation (contd…)



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

segment table





Limitations of Segmentation

- **External Fragmentation**
  - To deal with external fragmentation, the Segmentation can be combined with Paging, also known as **Segmented Paging.**

Virtual Memory

- Virtual memory is a technique that allows the execution of processes that are not completely in memory.
- One major advantage of this scheme is that programs can be larger than physical memory.
- Virtual memory gives an illusion to the user/programmer that huge amount memory is available for executing its process.
- Virtual memory can be implemented through demand paging.

Demand Paging

- Demand Paging
  - Pure Demand Paging - Never bring a page into memory until it is required
  - Pre-fetched Demand Paging
- A demand-paging system is similar to a paging system with swapping.
- Types of Pages
  - Modified Page or Dirty Page
  - Clean Page
- Page Fault Service Time

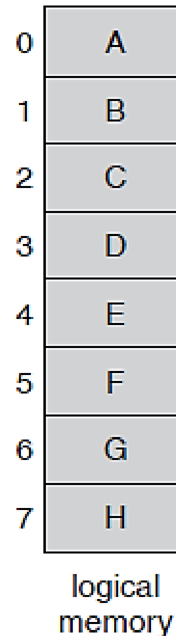
Hardware Support for Demand Paging

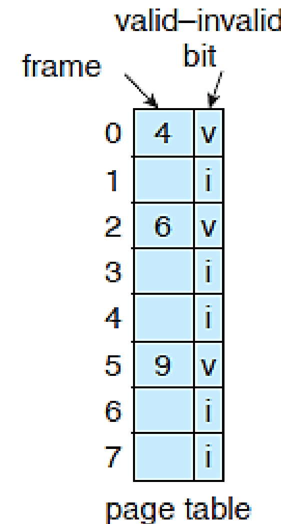
- Page Table
- Secondary Memory
  - This memory holds those pages that are not present in main memory.
  - The secondary memory is usually a high-speed disk.
  - It is known as the swap device, and the section of disk used for this purpose is known as swap space.

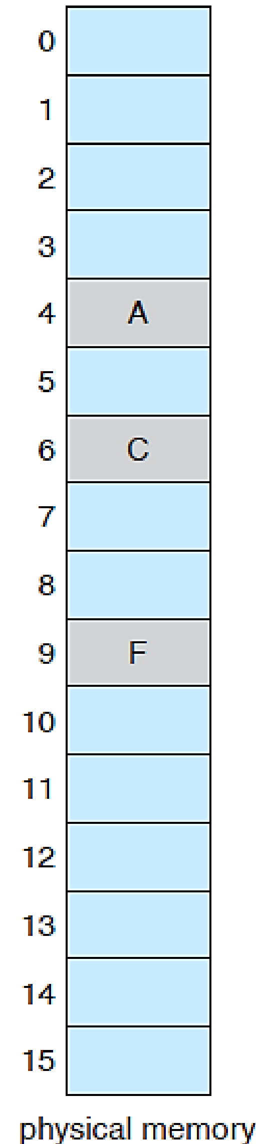
Note: To implement demand paging, we must develop a frame-allocation algorithm and a page-replacement algorithm.

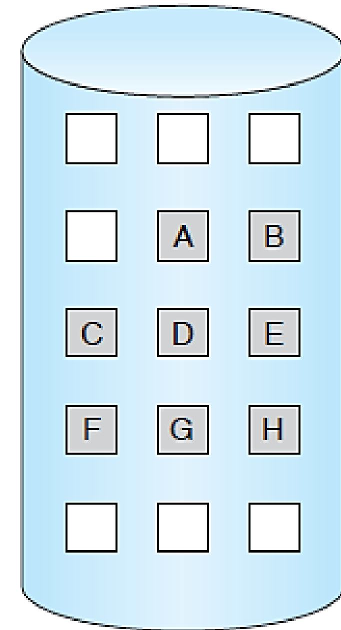
Page Table when some Pages are not in Main Memory

- To distinguish between the pages that are in memory and the pages that are on the disk, we need some form of hardware support.
- The valid–invalid bit scheme can be used for this purpose.
- When this bit is set to "valid," the associated page is in memory.
- If the bit is set to "invalid", the page is currently on the disk.

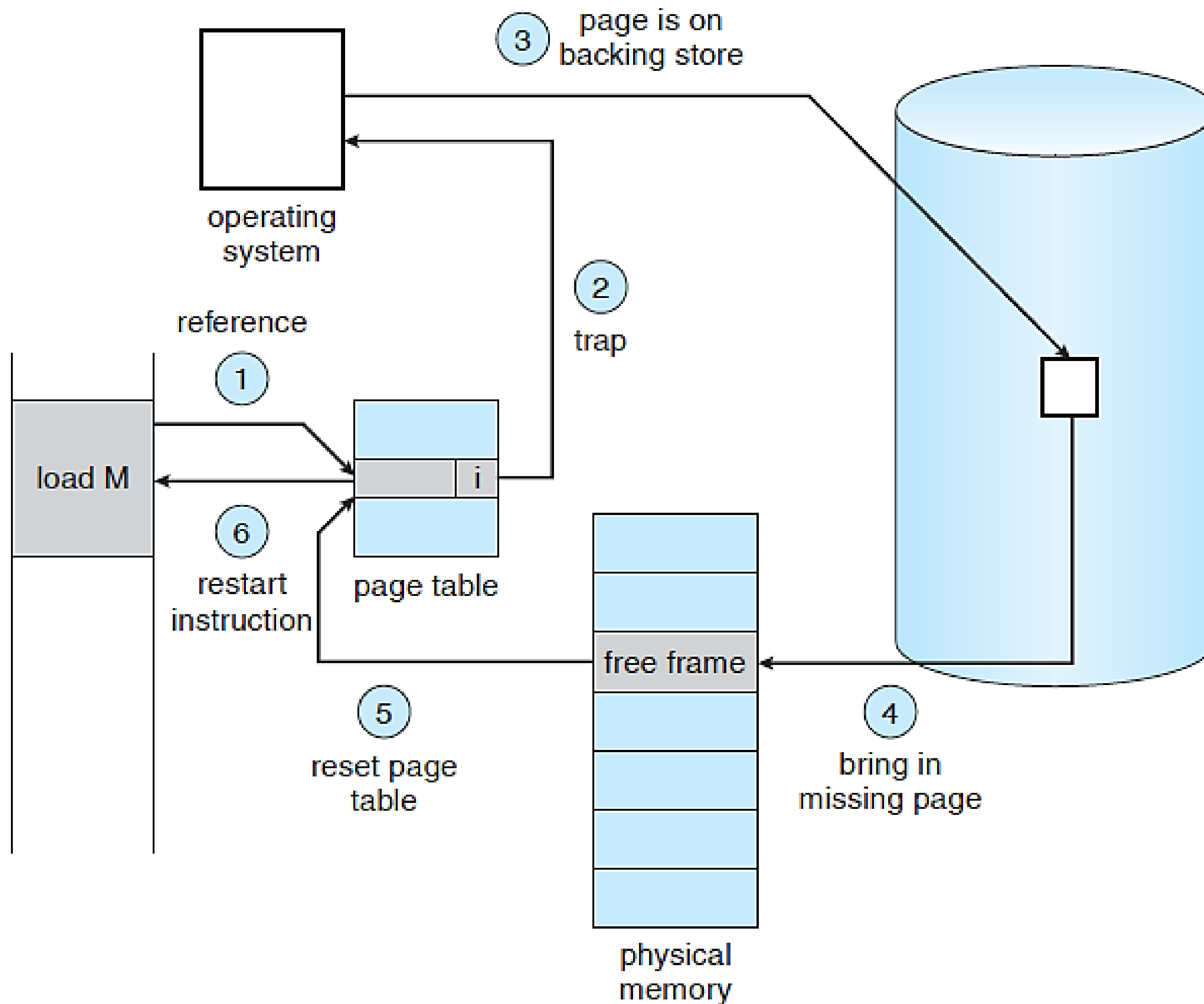








Steps in Handling a Page Fault



Performance of Virtual Memory

- Let
  - Main Memory Access Time (MMAT) = M
  - Page Fault Rate (PFR) = P
  - Page Fault Service Time (PFST) = S
  - Effective Memory Access Time (EMAT) = P*S+(1-P)*M
- If MMAT = 200 microsecond, PFR = 25%, PFST = 1 millisecond, then what is EMAT (in Microsecond)?

EMAT = 0.25*1000 + 0.75*200 = 250 + 150 = 400 Microsecond

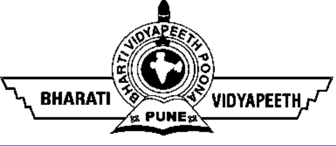
Performance of Virtual Memory

- If MMAT (M) = 1 microsecond, PFST (S) = 10 Millisecond, Hit Ratio = 99.99%, the what is EMAT (in Microsecond)?

If Hit Ratio = 99.99, then P = 0.01%

Therefore, EMAT = 0.0001*10000 + (1-0.0001)*1 =

= 1 + 0.9999 = 1.9999 Microsecond



Page Replacement

1) Find the location of the desired page on the disk.

2) Find a free frame:

   a) If there is a free frame, use it.

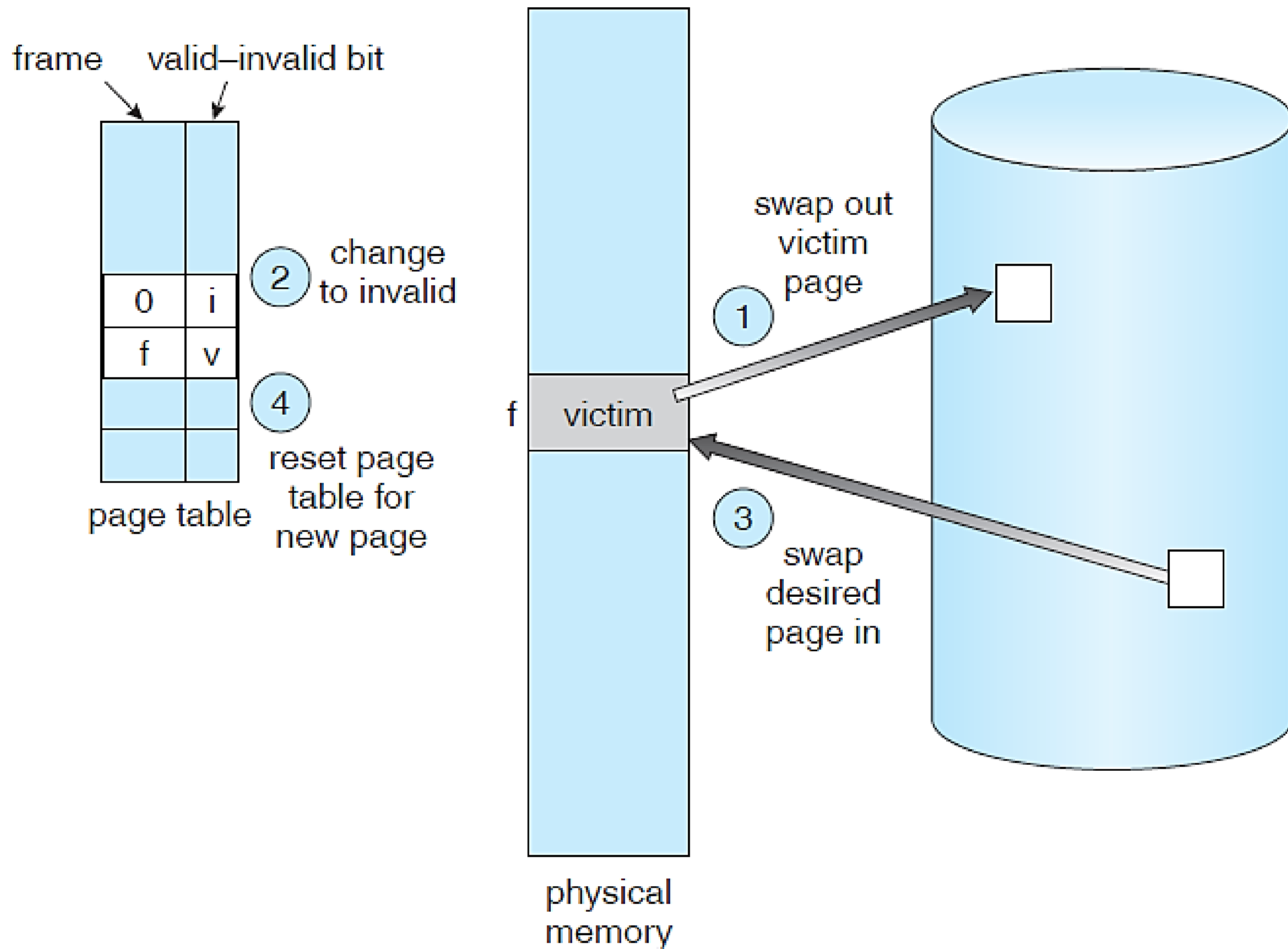
   b) If there is no free frame, use a page-replacement algorithm to select a victim frame.

   c) Write the victim frame to the disk (If dirty); change the page and frame tables accordingly.

3) Bring the desired page into the (newly) freed frame; change the page and frame tables.

4) Continue the process by restarting the instruction that caused the trap.

Page Replacement (contd…)



Page Replacement Algorithms

- First-Come, First-Out
- Optimal Page Replacement
- Least Recently Used

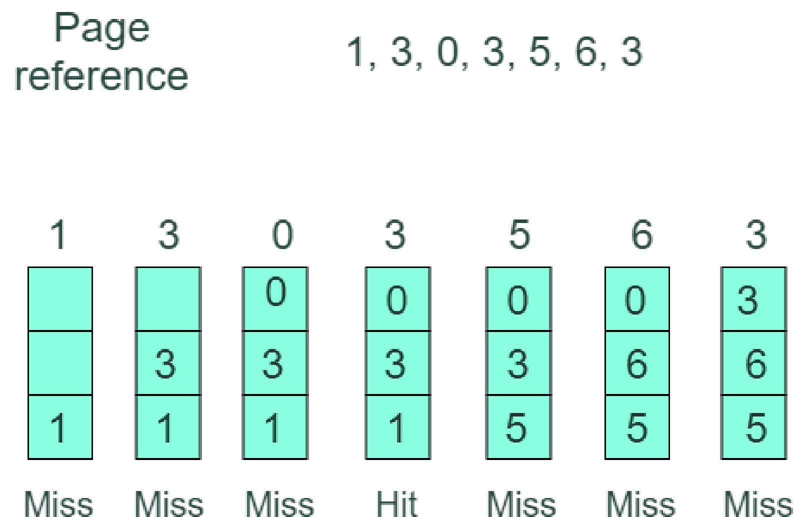
We can evaluate an algorithm by running it on a particular string of memory references and computing the number of page faults.

- The string of memory references is called a reference string.

Page Replacement Algorithm: First-Come, First-Out

- We can create a FIFO queue to hold all pages in memory. We replace the page at the head of the queue. When a page is brought into memory, we insert it at the tail of the queue.

Q. Consider page reference string 1, 3, 0, 3, 5, 6, 3 with 3 page frames. Find number of page faults.



Page Replacement Algorithm: First-Come, First-Out

Q. Consider page reference string 3, 2, 1, 0, 3, 2, 4, 3, 2, 1, 0, 4 with 3 page frames. Find number of page faults.

Q. Consider page reference string 3, 2, 1, 0, 3, 2, 4, 3, 2, 1, 0, 4 with 4 page frames. Find number of page faults.

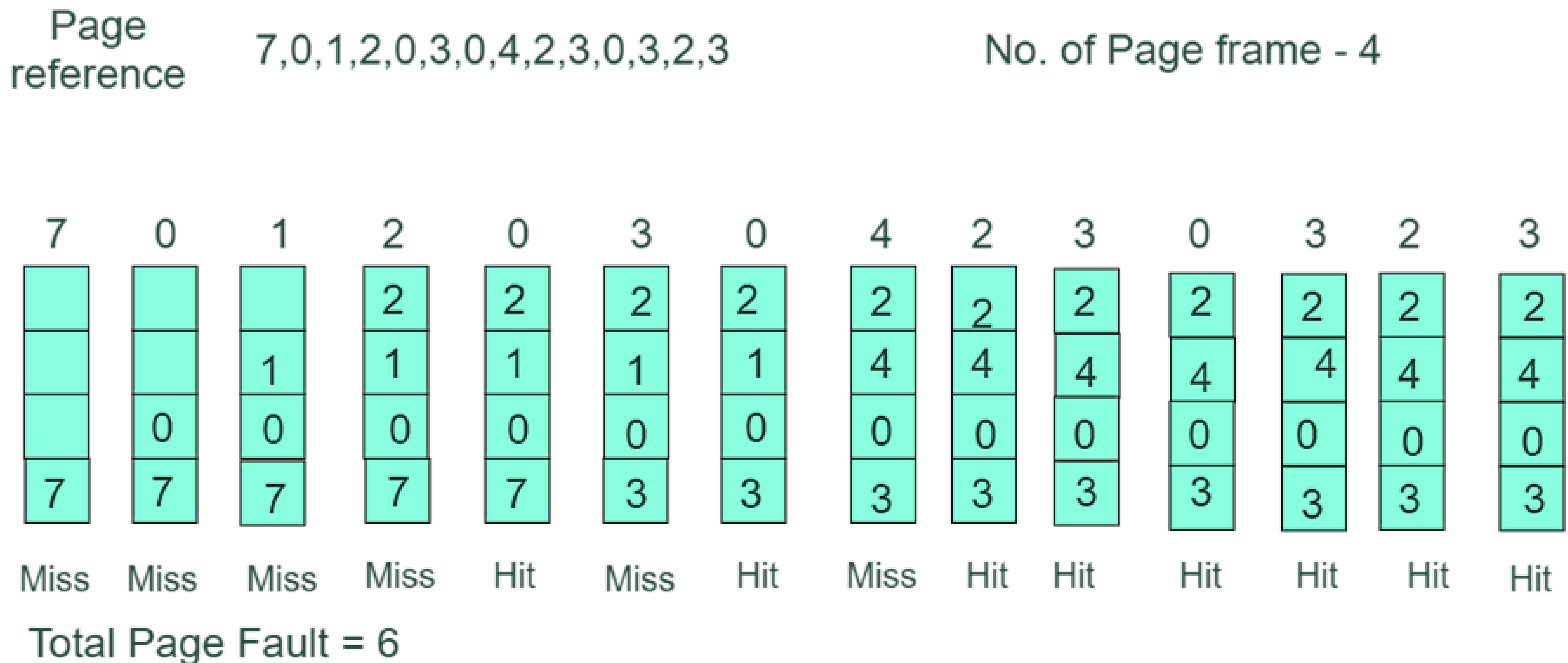
Belady's Anomaly – For some page-replacement algorithms, the page-fault rate may increase as the number of allocated frames increases.

Page Replacement Algorithm: Optimal Page Replacement

Replace the page that will not be used for the longest period of time.

- Use of this page-replacement algorithm guarantees the lowest possible page fault rate for a fixed number of frames.
- Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string.
- The optimal algorithm is used mainly for comparison studies.

Q. Consider the page references 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, with 4 page frame. Find number of page fault using **optimal page replacement algorithm**.



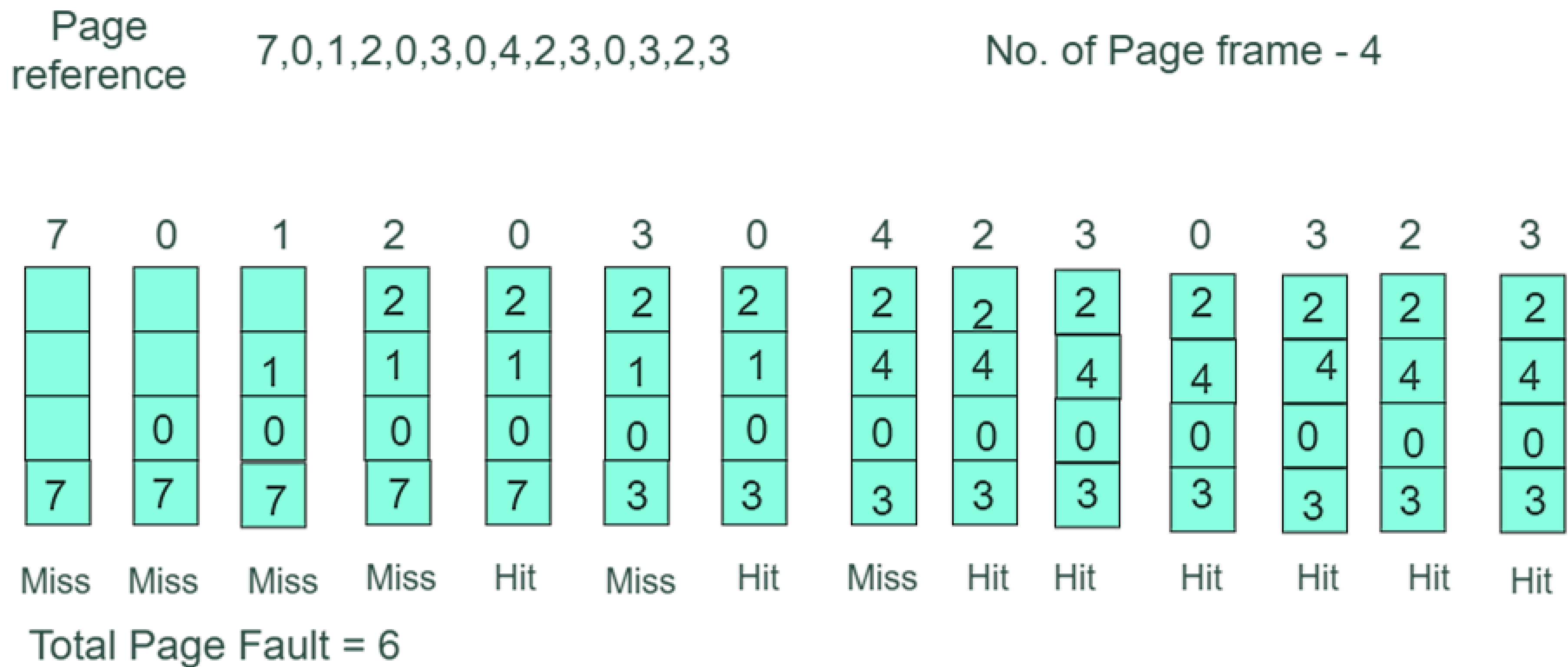
Page Replacement Algorithm: Least-Recently-Used (LRU)

Replace the page that has not been used for the longest period of time.

- LRU replacement associates with each page the time of that page's last use.
- When a page must be replaced, LRU chooses the page that has not been used for the longest period of time.
- This strategy can be considered as the optimal page-replacement algorithm looking backward in time, rather than forward.
- The LRU policy is often used as a page-replacement algorithm and is considered to be good.

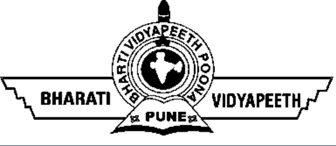
Page Replacement Algorithm: LRU Example

Q. Consider the page reference string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2 with 4 page frames. Find number of page faults using **least-recently-used algorithm**.



Allocation of Frames

- How do we allocate the fixed amount of free memory among the various processes?
  - The minimum number of frames per process is defined by the architecture.
  - The maximum number is defined by the amount of available physical memory.



Allocation of Frames: Equal Allocation

- The easiest way to split m frames among n processes is to give everyone an equal share, m/n frames.
- For instance, if there are 93 frames and five processes, each process will get 18 frames. The three leftover frames can be used as a free-frame buffer pool.

Allocation of Frames: Proportional Allocation

- Consider a system with a 1-KB frame size.
  - If a small student process of 10 KB and an interactive database of 127 KB are the only two processes running in a system with 62 free frames, it does not make much sense to give each process 31 frames.
  - The student process does not need more than 10 frames, so the other 21 are, strictly speaking, wasted.
- To solve this problem, we can use proportional allocation, in which we allocate available memory to each process according to its size.

Allocation of Frames: Global vs. Local Allocation

- With multiple processes competing for frames, we can classify page-replacement algorithms into two broad categories:
  - Global Replacement
    - It allows a process to select a replacement frame from the set of all frames, even if that frame is currently allocated to some other process; that is, one process can take a frame from another.
    - We may allow high-priority processes to select frames from low-priority processes for replacement.
  - Local Replacement
    - It allows that that each process select from only its own set of allocated frames.
    - With a local replacement strategy, the number of frames allocated to a process does not change.

Thrashing

- If the process does not have the number of frames it needs to support pages in active use, it will quickly page-fault.
- At this point, it must replace some page.
- However, since all its pages are in active use, it must replace a page that will be needed again right away.
- Consequently, it quickly faults again, and again, and again, replacing pages that it must bring back in immediately.
- This high paging activity is called **thrashing**.
- A process is thrashing if it is spending more time in paging than executing.

Thrashing (contd…)