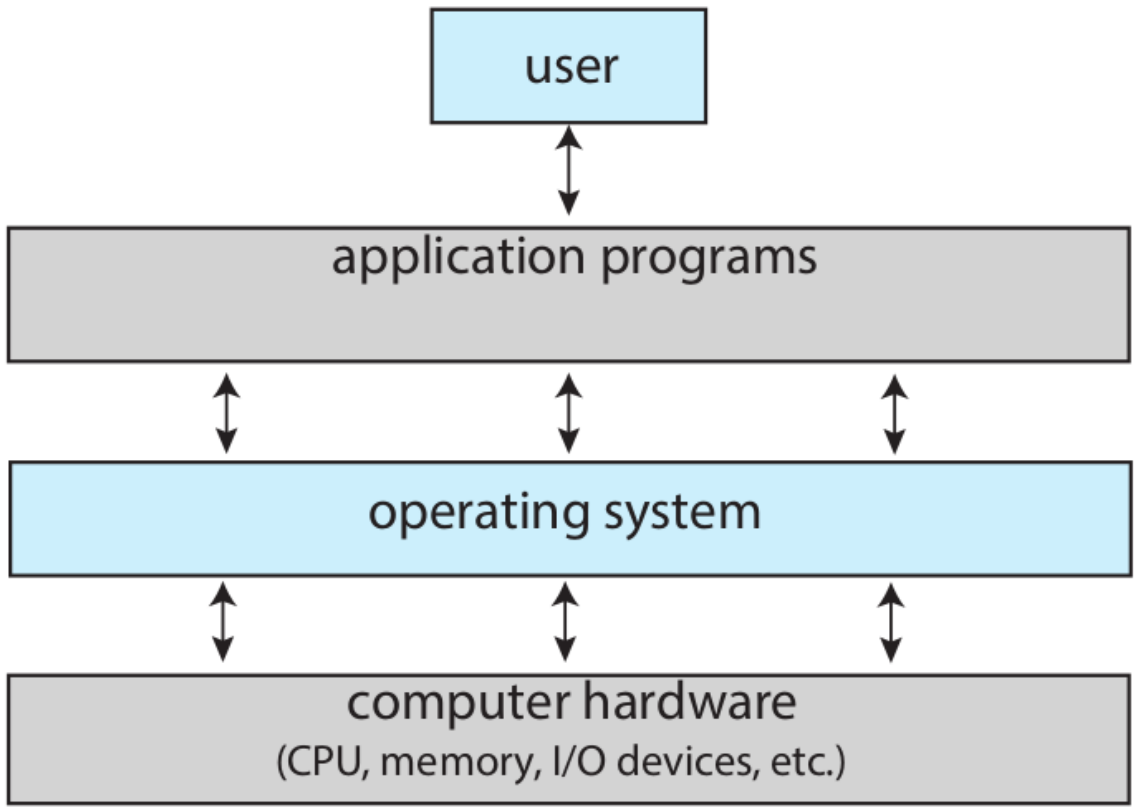
Operating Systems

Vasudevan T V

Module 1
Overview

- An Operating System is a software that acts as an interface between the user of a computer and the computer hardware
- It provides an environment in which a user can execute programs in a convenient and efficient manner.



Functions of an Operating System

- Memory Management
- Process Management
- Device Management
- Storage Management
- Security

Functions of an Operating System

- Memory Management
  - It refers to the management of main memory
  - To execute a program all (or part) of the instructions must be in memory
  - All (or part) of the data that is needed by the program must be in memory
- Memory Management Activities
  - Keeping track of which parts of memory are currently being used and by whom
  - In a multiprogramming environment, the OS decides which process will get memory when and how much
  - Allocating and deallocating memory space as needed

Functions of an Operating System

- Process Management
  - A process is a program in execution
- Process Management Activities
  - Create and delete processes
  - In a multiprogramming environment, the OS decides which process gets the processor, when and for how much time. This is called process scheduling
  - Keeps track of the status of processor and processes.
  - Allocates the processor to a process
  - De-allocates processor when it is no longer required
  - Provide mechanisms for process communication

Functions of an Operating System

- Device Management
  - It refers to the management of I/O devices
- Device Management Activities
  - Keeps track of all devices
  - Decides which process gets the device, when and for how much time
  - Allocates the device to a process
  - De-allocates the device when it is no longer required

Functions of an Operating System

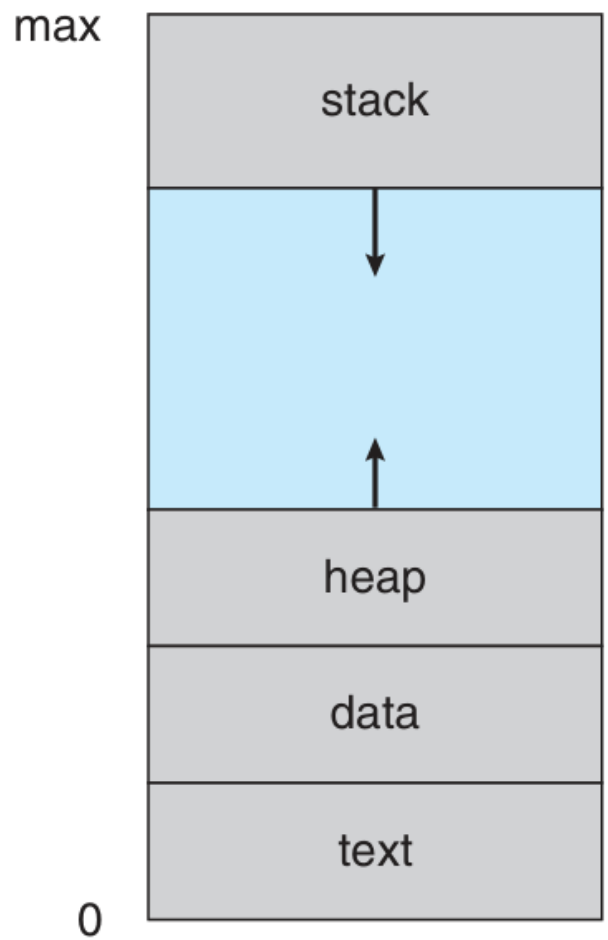
- Storage Management
  - It refers to the management of secondary storage
  - Programs are stored on secondary storage devices until loaded into memory
- Storage Management Activities
  - Mounting and Unmounting
  - Free Space Management
  - Storage Allocation
  - Disk Scheduling
  - Disk Partitioning

Functions of an Operating System

- Protection and Security
  - Protection is a mechanism for controlling access of processes or users to resources defined by the OS (Setting Passwords, Access Privileges)
  - Security is a mechanism for handling external threats to the system (Attack from hackers, viruses)

Process Management: Concept of a Process

- A process is a program in execution
- Memory layout of a process



Concept of a Process

- The memory layout of a process is divided into 4 sections.

1. Text Section
   It contains the executable code.
2. Data Section
   It contains the global variables.
3. Heap Section
   It holds memory that is dynamically allocated during program runtime.
4. Stack Section
   This is used as temporary data storage when invoking functions (such as function parameters, return addresses, and local variables)

- The sizes of text and data sections are fixed while the sizes of heap and stack sections are varying.

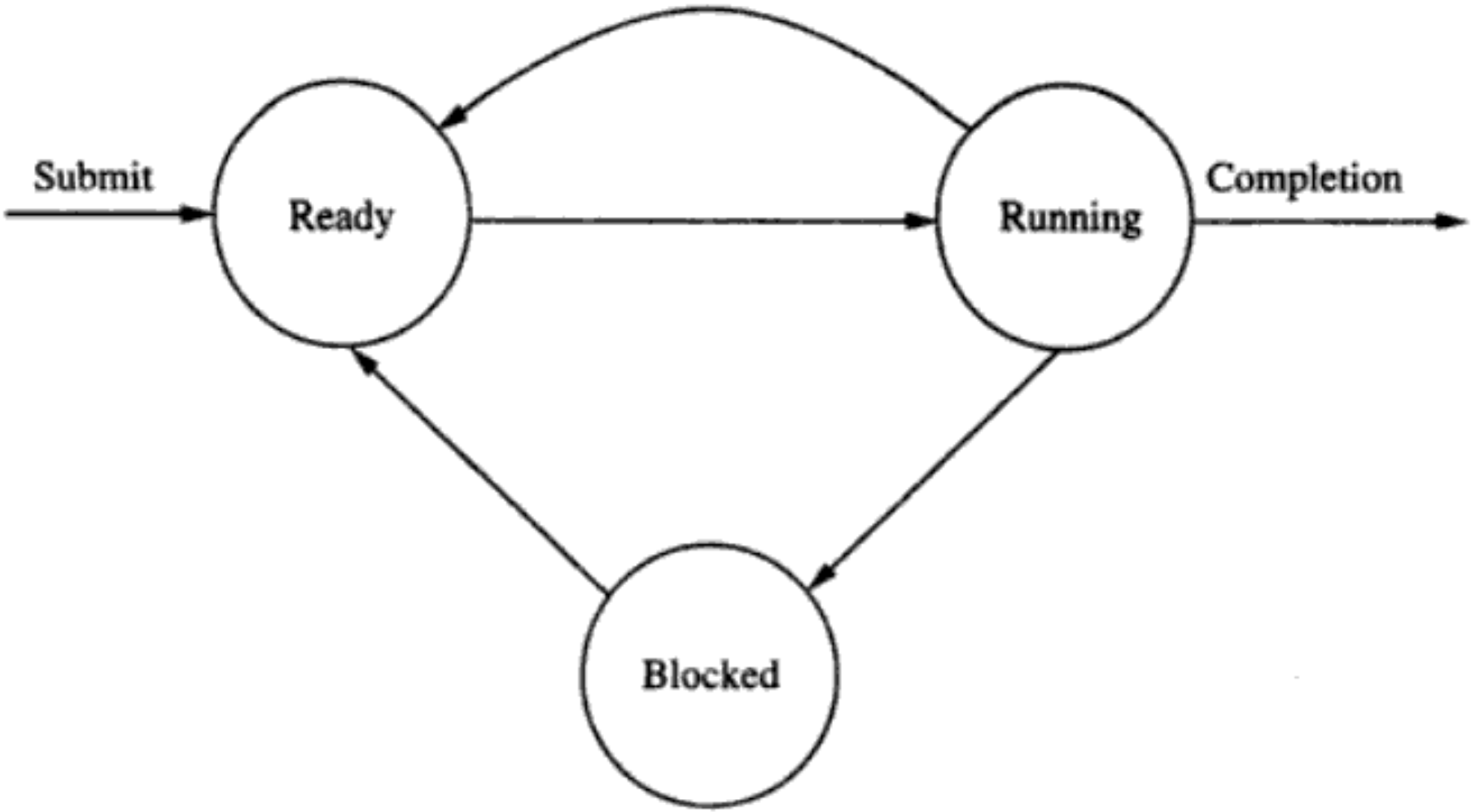
Process States

- A process can be in any of the three basic states

1. Running - Instructions are being executed
2. Ready - The process is waiting for the availability of the processor
3. Blocked (Waiting) - The process is waiting for some event to occur such as I/O completion, Memory Availability etc.

Process States

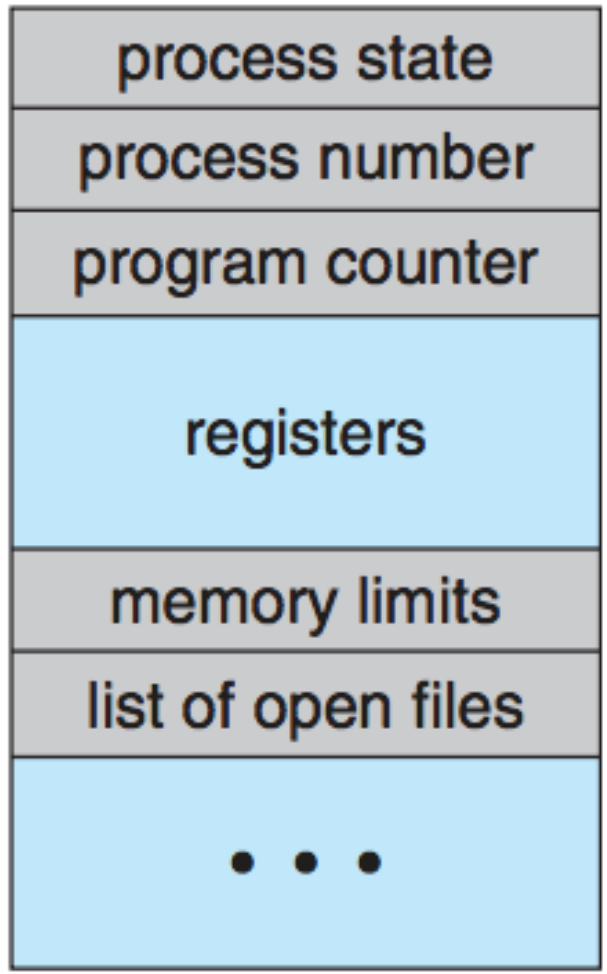
- State Transition Diagram of a Process



Process Control Block

- Each process is represented in the operating system by a process control block (PCB) (task control block)
- It contains several pieces of information associated with a process, including these
- process state
- process number
- program counter - This counter indicates the address of the next instruction to be executed for this process
- CPU Registers - high speed memory in the CPU for storing data
- memory limits - maximum memory allocated for the process
- list of open files

Process Control Block

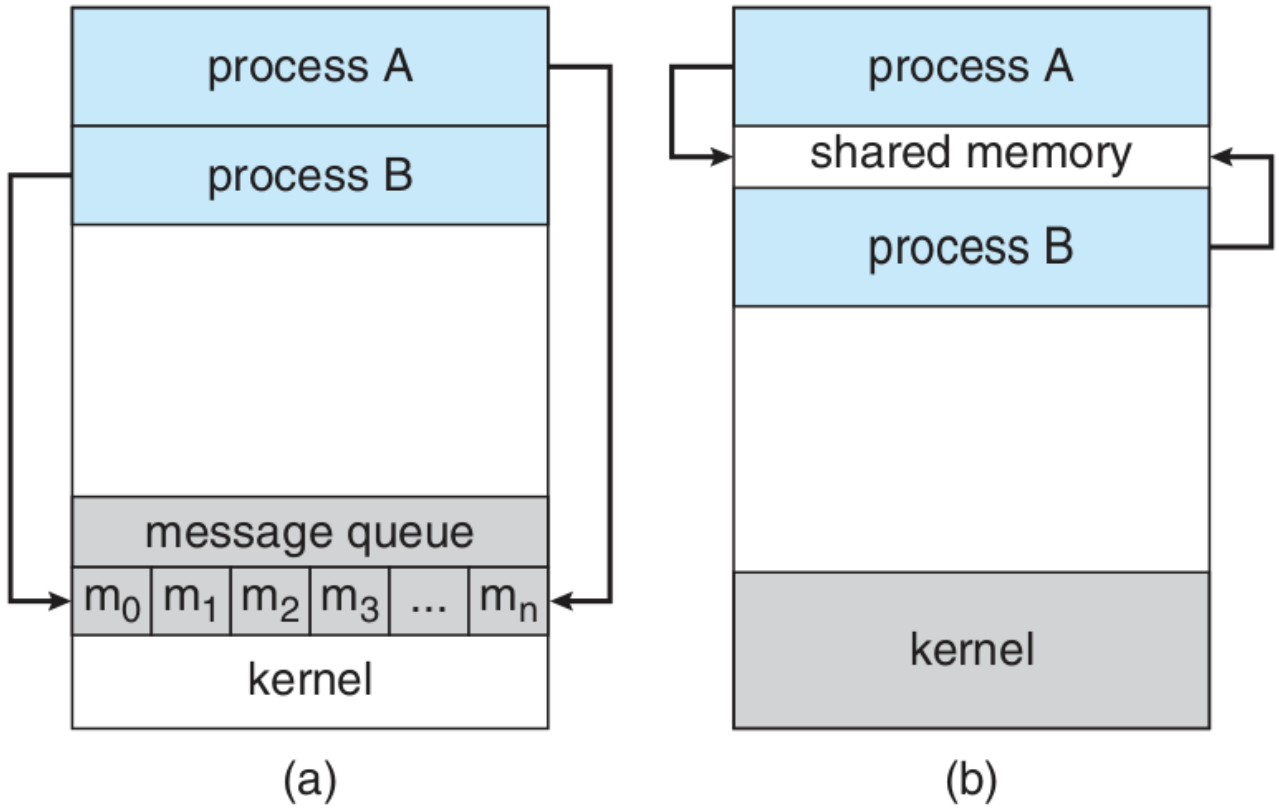


Inter Process Communication

- Processes require an interprocess communication (IPC) mechanism that will allow them to exchange data and information
- There are two fundamental models of interprocess communication: shared memory model and message passing model
- In the shared memory model, a region of memory is shared by the cooperating processes
- Processes can then exchange information by reading and writing data to the shared region
- In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes

Inter Process Communication

- (a) Message Passing Model (b) Shared Memory Model



- Message passing model is useful for exchanging smaller amounts of data
- Shared memory model can be faster than message passing model

Inter Process Communication

- The message passing model uses two communication primitives - SEND and RECEIVE
- These primitives can have four modes - blocking,non blocking, synchronous and asynchronous
- In blocking mode, SEND primitive does not return control to the sender process until message has been sent, or an acknowledgement has been received
- In non blocking mode, SEND primitive returns control to the sender process as soon as the message is copied to kernel buffer, an intermediate buffer used for storing message
- In synchronous mode, SEND primitive is blocked until the execution of the corresponding RECEIVE message (Another SEND cannot be performed before the completion of RECEIVE)
- In asynchronous mode, SEND primitive is not blocked until the execution of the corresponding RECEIVE message (Another SEND can be performed before the completion of RECEIVE)

CPU Scheduling

- In a multiprogramming environment, several processes are to be executed in the CPU
- As processes enter the system, they are put into a job queue, which consists of all processes in the system
- The processes that are ready and waiting to execute are kept on a list called the ready queue
- The processes waiting for a particular I/O device are put in a device queue
- The CPU scheduler selects from among the processes that are ready to execute and allocates the CPU to one of them

CPU Scheduling

- For CPU scheduling, various algorithms are used
- The choice of a particular algorithm may favour one class of processes over another
- The criteria used for judging the best algorithm to be used in a particular situation is given below

1. CPU Utilisation

- It is the fraction of time the CPU remains busy
- Conceptually, CPU utilisation can range from 0 to 100 percent
- In a real system, it ranges from 40 percent (for a lightly loaded system) to 90 percent (for a heavily loaded system)

2. Throughput

- The number of processes that are completed per time unit is called throughput
- For long processes, this rate may be one process per hour
- For short transactions, it may be ten processes per second

CPU Scheduling

3. Turnaround Time

- The interval from the time of submission of a process to the time of completion is the turnaround time

4. Waiting Time

- It is the sum of the periods spent waiting by a process in the ready queue

5. Response Time

- It is the time from the submission of a request until the first response is produced by a process

Scheduling Algorithms

- There are 2 kinds of scheduling algorithms - preemptive and non-preemptive
- A preemptive scheduling algorithm picks a process and lets it run for a maximum of some fixed time
- If it is still running at the end of the time interval, it is suspended and the scheduler picks another process to run (if one is available)
- A non preemptive scheduling algorithm picks a process to run and then just lets it run until it releases the CPU either by terminating or by requesting I/O

Scheduling Algorithms

1. First Come First Served (FCFS) Scheduling Algorithm

- It is the simplest CPU scheduling algorithm
- Here, the process that requests the CPU first, is allocated the CPU first
- It is a non preemptive scheduling algorithm
- Consider the following set of processes that arrive at time 0, with the length of the CPU burst time given in milliseconds
- Burst time is the total time taken by the process for its execution on the CPU.

Process	Burst Time
P1	24
P2	3
P3	3

Scheduling Algorithms

1. First Come First Served (FCFS) Scheduling Algorithm

- Let these processes arrive in the order P_1 , P_2 , P_3
- The execution of these processes in FCFS order is given in the Gantt chart below

- Waiting Time for $P_1 = 0$
- Waiting Time for $P_2 = 24$
- Waiting Time for $P_3 = 27$
- Average Waiting Time $= \frac{0+24+27}{3} = 17$

Scheduling Algorithms

1. First Come First Served (FCFS) Scheduling Algorithm

- Let these processes arrive in the order P_2 , P_3 , P_1
- The execution of these processes in FCFS order is given in the Gantt chart below

- Waiting Time for $P_1 = 6$
- Waiting Time for $P_2 = 0$
- Waiting Time for $P_3 = 3$
- Average Waiting Time $= \frac{6+0+3}{3} = 3$
- Thus we can see that the average waiting time varies substantially, if the CPU burst times vary greatly

Scheduling Algorithms

2. Shortest Job First (SJF) Scheduling Algorithm

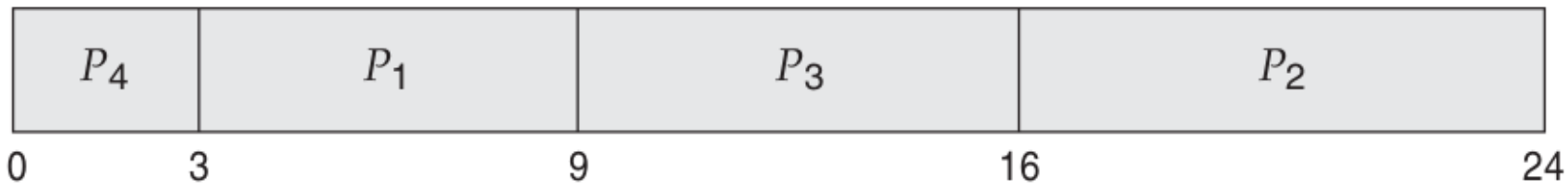
- Here, the process that has the shortest burst time is allocated the CPU first
- The CPU is assigned next to the process having the smallest burst time among the remaining ones
- This is continued till all processes are completed
- If 2 processes are having the same burst time, then FCFS scheduling algorithm is used to break the tie

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

Scheduling Algorithms

2. Shortest Job First (SJF) Scheduling Algorithm

- The execution of these processes using this algorithm is given in the Gantt chart below



- Waiting Time for $P_1 = 3$
- Waiting Time for $P_2 = 16$
- Waiting Time for $P_3 = 9$
- Waiting Time for $P_4 = 0$
- Average Waiting Time $= \frac{3+16+9+0}{4} = 7$ ms
- This is less than the average waiting time in FCFS scheduling which is 10.25 ms

Scheduling Algorithms

2. Shortest Job First (SJF) Scheduling Algorithm

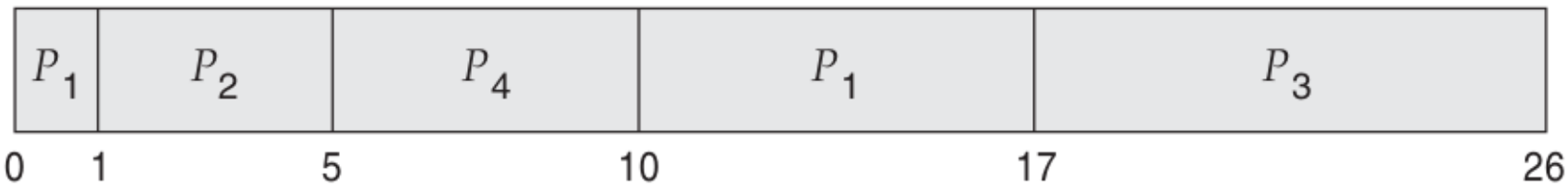
- There is a preemptive version of the SJF scheduling algorithm
- Here the currently executing process is preempted if its remaining time is greater than the burst time of a newly arriving process
- This is called Shortest Remaining Time First algorithm

Process	Arrival Time	Burst Time
P1	0	8
P2	1	4
P3	2	9
P4	3	5

Scheduling Algorithms

2. Shortest Job First (SJF) Scheduling Algorithm

- The execution of these processes using this algorithm is given in the Gantt chart below



- Waiting Time for $P_1 = 10 - 1 = 9$
- Waiting Time for $P_2 = 1 - 1 = 0$
- Waiting Time for $P_3 = 17 - 2 = 15$
- Waiting Time for $P_4 = 5 - 3 = 2$
- Average Waiting Time $= \frac{9+0+15+2}{4} = 6.5$ ms
- This is less than the average waiting time in non preemptive SJF scheduling which is 7.75 ms

Scheduling Algorithms

3. Priority Scheduling Algorithm

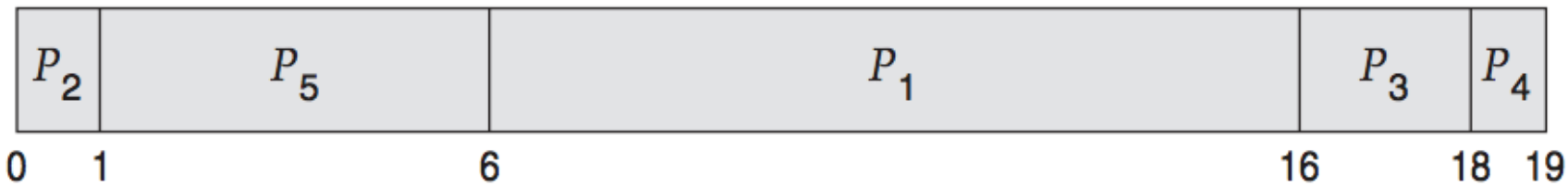
- Here, each process is assigned some priority
- The CPU is assigned first to the process having the highest priority
- The CPU is assigned next to the process having the next higher priority
- This is continued till all processes are completed
- If 2 processes are having the same priority, then FCFS scheduling algorithm is used to break the tie
- Here lower numbers represent higher priority

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

Scheduling Algorithms

3. Priority Scheduling Algorithm

- The execution of these processes using this algorithm is given in the Gantt chart below



- Waiting Time for $P_1 = 6$
- Waiting Time for $P_2 = 0$
- Waiting Time for $P_3 = 16$
- Waiting Time for $P_4 = 18$
- Waiting Time for $P_5 = 1$
- Average Waiting Time $= \frac{6+0+16+18+1}{5} = 8.2$ ms

Scheduling Algorithms

3. Priority Scheduling Algorithm

- A priority scheduling algorithm can leave some low-priority processes waiting indefinitely
- This problem is called indefinite blocking or starvation
- A solution to this problem is to gradually increase the priority of processes that wait in the system for a long time
- This is called aging
- A priority scheduling algorithm can be preemptive also
- A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process

Scheduling Algorithms

4. Round Robin(RR) Scheduling Algorithm

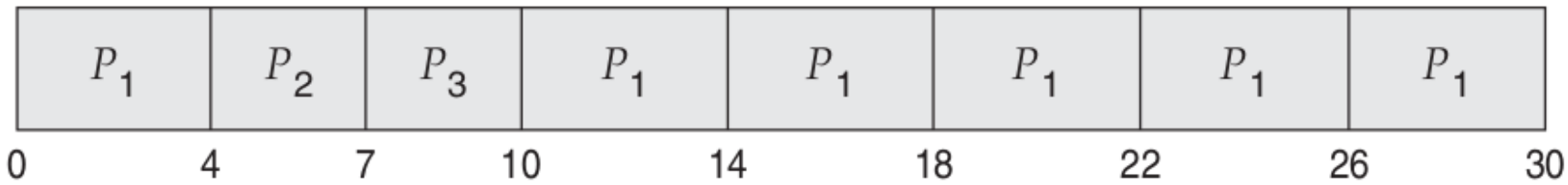
- Here, CPU is allocated to each process for a time interval called a time quantum or time slice
- After that time quantum, the currently running process is preempted and the next process is executed
- This is continued till all the processes are completed
- The process which has arrived first begins its execution first
- If the execution time of a process is less than the time quantum, the CPU is released voluntarily by the process

Process	Burst Time
P1	24
P2	3
P3	3

Scheduling Algorithms

4. Round Robin(RR) Scheduling Algorithm

- Let the time quantum be 4ms
- The execution of these processes using this algorithm is given in the Gantt chart below



- Waiting Time for P_1 = 10 - 4 = 6
- Waiting Time for P_2 = 4
- Waiting Time for P_3 = 7
- Average Waiting Time = $\frac{6+4+7}{3}$ = 5.66 ms

Scheduling Algorithms

4. Round Robin(RR) Scheduling Algorithm

- The performance of the RR algorithm depends heavily on the size of the time quantum
- If the time quantum is extremely large, the RR policy is same as the FCFS policy
- If the time quantum is extremely small (say, 1 millisecond), the RR approach can result in a large number of context switches
- This will slow down the execution of processes

Module 2
Concepts of Threads

- A thread is a basic unit of execution within a process
- A thread performs a single task associated with a process
- A thread is a lightweight process
- A process can have many threads within it
- A web browser might have one thread display images or text while another thread retrieves data from the network
- A word processor may have a thread for displaying graphics, another thread for responding to keystrokes from the user, and a third thread for performing spelling and grammar checking in the background.

The Critical Section Problem and Process Synchronisation

- Concurrent Processes(or Threads) access shared variables at the same time
- A Critical Section is a code segment in a process in which a shared variable is accessed
- In such a situation, the integrity of variables may be violated
- To solve this problem (Critical Section Problem) we have to ensure that processes need to synchronise in such a way that only one process access the shared variables at one time
- Process Synchronisation is the task of coordinating the execution of processes in a way that no two processes can have access to the same shared data and resources
- A resource is any hardware or software associated with a computer.
- We have to ensure that no two processes are executing in their critical sections at the same time

The Critical Section Problem

- Each process must request permission to enter its critical section
- The section of code implementing this request is the entry section
- The critical section may be followed by an exit section
- The remaining code is the remainder section

```
while (true) {

    entry section

        critical section

    exit section

        remainder section

}
```

The Critical Section Problem

- A solution to the critical-section problem must satisfy the following three requirements:

1. Mutual Exclusion
   Only one process can execute in its critical section in a given period of time.
2. Progress
   When no process is executing in its critical section, then any process which wants to enter its critical section must be granted permission without delay.
   If there are two or more processes which want to enter their critical sections, then the selection cannot be postponed indefinitely.
3. Bounded Waiting
   No process can prevent any other from entering its critical section indefinitely. In other words, there exists a bound on the waiting time of a process before entering its critical section.

Semaphores

- A solution to this problem is the usage of semaphores
- A semaphore is an integer variable that, apart from initialisation, is accessed only through two standard atomic operations: wait() and signal() .
- Semaphores were introduced by the Dutch computer scientist E.W.Dijkstra

```
wait(S)
{
     while (S <= 0);  // If S is not positive no
                      // operation is performed
     S = S - 1;  // If S is positive, it is decremented
                 // by 1
}
signal(S)
{
     S = S + 1; // S is incremented by 1
}
```

Semaphores

- There are two types of semaphores

1. counting semaphore
   They have an unrestricted value domain. The semaphore count is the number of available resources. If the resources are added, semaphore count is incremented and if the resources are removed, the count is decremented.
2. binary semaphore
   Their values are restricted to 0 and 1. The wait operation only works when the semaphore is 1 and the signal operation works only when semaphore is 0.

- Binary semaphores are easier to implement than counting semaphores

Semaphore

- Semaphore Usage
- Consider two concurrently running processes: P1 with a statement S1 and P2 with a statement S2.
- Suppose we require that S2 be executed only after S1 has completed
- We can implement this scheme by letting P1 and P2 share a common binary semaphore S, initialised to 0

```
Process P1              Process P2
----------              ----------

..........              ..........
S1 ;                    wait(S);
signal(S);              S2;
..........              ..........
```

- The object used for achieving mutual exclusion is called mutex
- Here binary semaphore S is the mutex

Deadlocks

- In a multi programming 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
- Example
- P_1 is holding a resource R1 and waiting for resource R2
- P_2 is holding a resource R2 and waiting for resource R1

Necessary Conditions for Deadlock

- A deadlock situation can arise if the following four conditions hold simultaneously in a system

1. Mutual Exclusion

- At least one resource must be held in non shareable mode; that is only one process can use this resource at a time

2. Hold and Wait

- A process holds at least one resource and waits to acquire other resources that are currently being held by other processes

3. No Preemption

- Resources cannot be preempted; They are released voluntarily by processes after completing their tasks

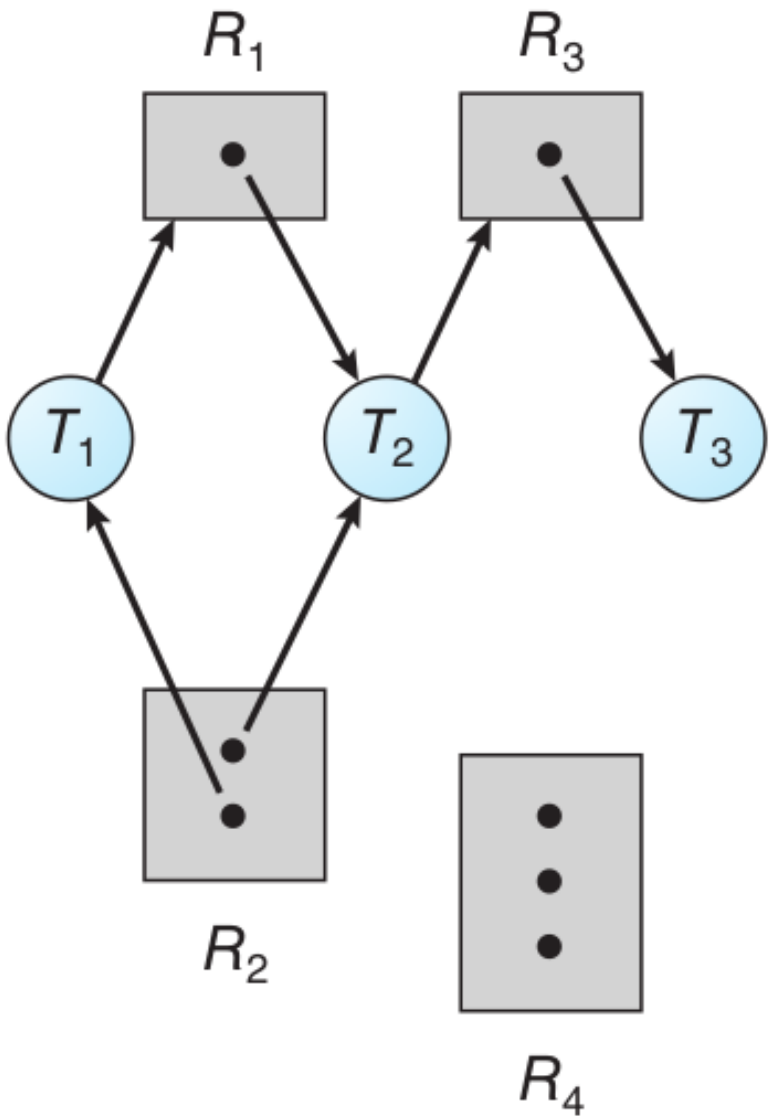
4. Circular Wait

- A set $\{P_0 , P_1 , ..., P_n \}$ of waiting processes must exist such that P_0 is waiting for a resource held by P_1 , P_1 is waiting for a resource held by P_2 , ..., P_{n-1} is waiting for a resource held by P_n , and P_n is waiting for a resource held by P_0

Resource Allocation Graph

- A deadlock situation can be described using a directed graph called a 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
- T $= \{T_1, T_2, ..., T_n\}$, the set consisting of all the active threads in the system
- R $= \{R_1, R_2, ..., R_m\}$, the set consisting of all resource types in the system
- A directed edge $T_i \rightarrow R_j$ is called a request edge
- It signifies that thread T_i has requested an instance of resource type R_j and is currently waiting for that resource
- A directed edge $R_j \rightarrow T_i$ is called an assignment edge
- It signifies that an instance of resource type R_j has been allocated to thread T_i

Resource Allocation Graph



Resource Allocation Graph

- It depicts the following situation
- Sets
- $\mathrm{T} = \{T_1, T_2, T_3\}$
- $\mathrm{R} = \{R_1, R_2, R_3, R_4\}$
- $\mathrm{E} = \{T_1 \rightarrow R_1, T_2 \rightarrow R_3, R_1 \rightarrow T_2, R_2 \rightarrow T_2, R_2 \rightarrow T_1, R_3 \rightarrow T_3\}$
- Resource Instances
- One instance of resource type R_1
- Two instances of resource type R_2
- One instance of resource type R_3
- Three instances of resource type R_4

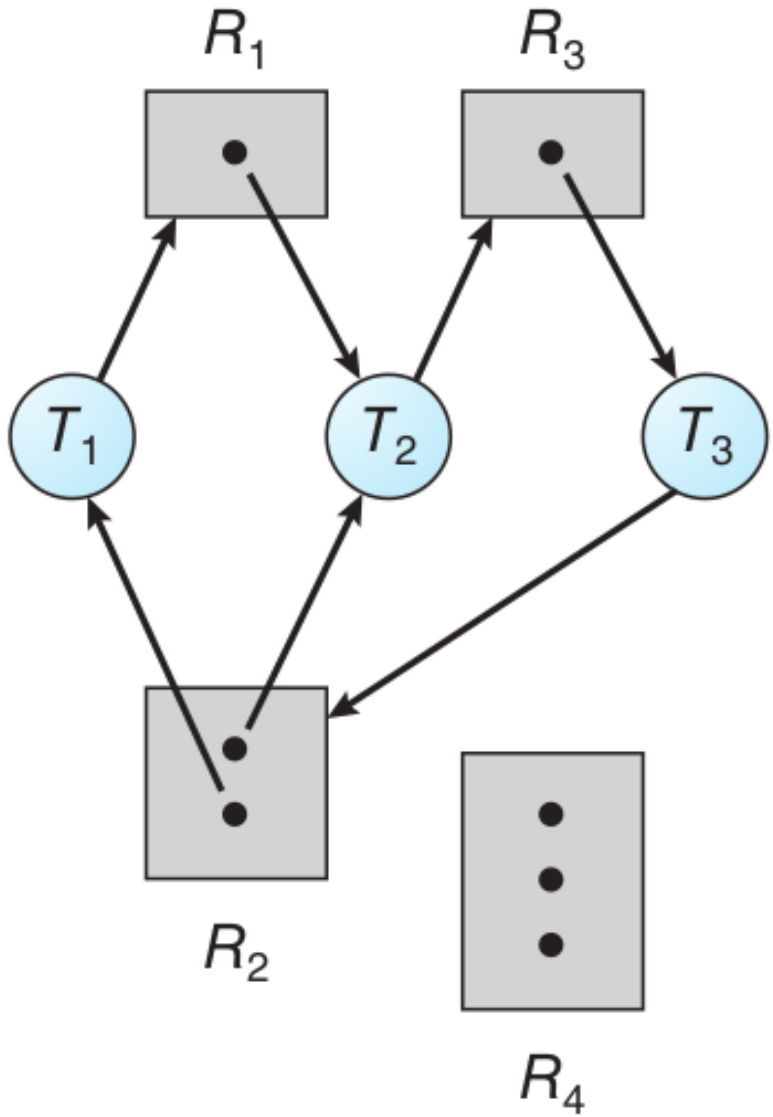
Resource Allocation Graph

- Thread states
- Thread T_1 is holding an instance of resource type R_2 and is waiting for an instance of resource type R_1
- Thread T_2 is holding an instance of R_1 and an instance of R_2 and is waiting for an instance of R_3
- Thread T_3 is holding an instance of R_3

- 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
- If each resource type has only one instance and if there is a cycle, then it is an indication that deadlock has occurred
- If each resource type has multiple instances and if there is a cycle, it may not indicate a deadlock state

Resource Allocation Graph

- with a deadlock

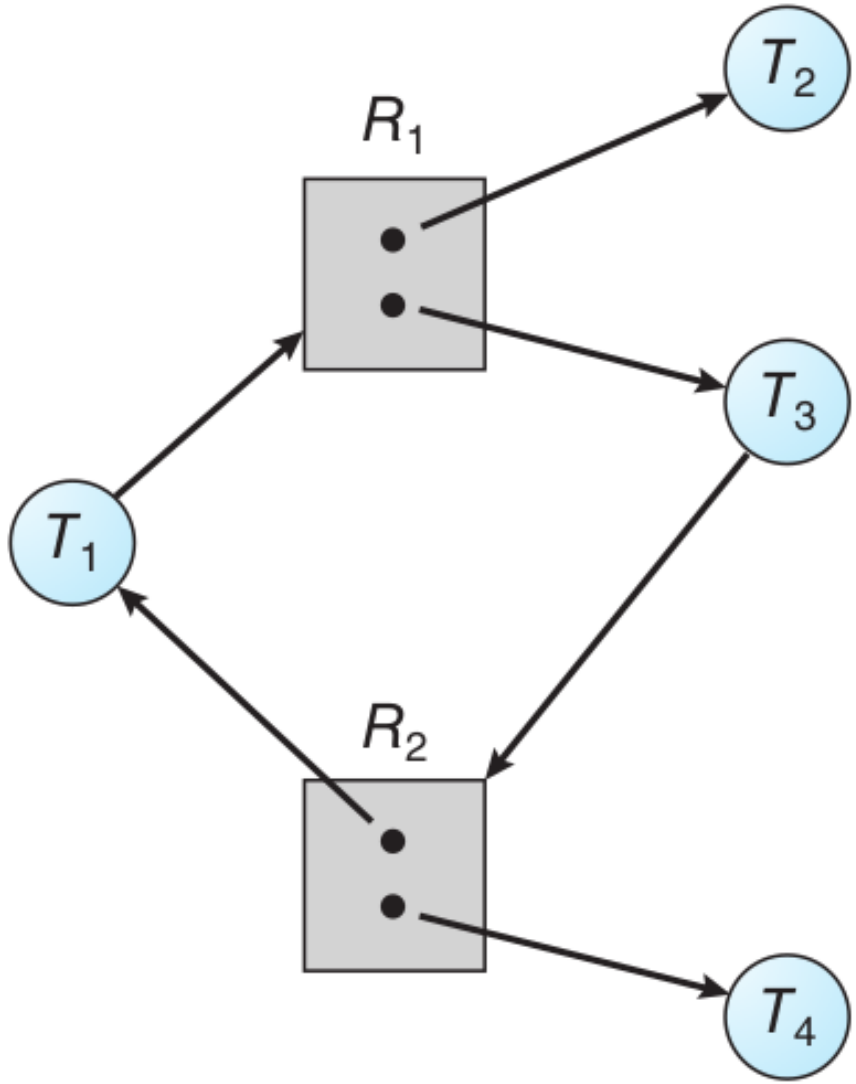


Resource Allocation Graph

- Cycles
- $T_1 \rightarrow R_1 \rightarrow T_2 \rightarrow R_3 \rightarrow T_3 \rightarrow R_2 \rightarrow T_1$
- $T_2 \rightarrow R_3 \rightarrow T_3 \rightarrow R_2 \rightarrow T_2$
- Threads T1 , T2 , and T3 are deadlocked
- Thread T2 is waiting for the resource R3 , which is held by thread T3
- Thread T3 is waiting for either thread T1 or thread T2 to release resource R2
- In addition, thread T1 is waiting for thread T2 to release resource R1 .

Resource Allocation Graph

- With a cycle but no deadlock



Resource Allocation Graph

- Cycle
- $T_1 \rightarrow R_1 \rightarrow T_3 \rightarrow R_2 \rightarrow T_1$
- There is no deadlock
- Thread T4 may release its instance of resource type R2
- That resource can then be allocated to T3 , breaking the cycle

Deadlock Prevention

- There are 4 necessary conditions for a deadlock to occur

1. Mutual Exclusion
2. Hold and Wait
3. No Pre-emption
4. Circular Wait

- If we prevent any of these, then we can prevent deadlock
- We will examine each of these conditions separately

Deadlock Prevention

1. Preventing Mutual Exclusion

- The condition of mutual exclusion says that at least one resource must be held in non shareable mode; that is only one process can use this resource at a time
- If there are only shareable resources like read only files then mutual exclusion condition never occurs
- However, in practical situations there may be non shareable resources
- Hence we cannot prevent mutual exclusion

Deadlock Prevention

2. Preventing Hold and Wait

- In hold and wait condition, a process holds a resource and waits for other resources which are held by other processes
- To prevent this condition we follow 2 protocols
- In the first protocol, before a process begins execution, all the resources needed are allocated
- Hence it need not request any other
- In the second protocol, before a process requests a resource, all the currently held ones are released
- Disadvantages
- Resource utilisation may be low
- There may be starvation problem where some processes may wait indefinitely for resources

Deadlock Prevention

2. Preventing No Pre-emption

- In no pre-emption condition, processes release resources only at the end of execution
- We can use 2 protocols for preventing this
- In the first one, If a process holds a resource and requests for another resource that cannot be immediately allocated to it, then the currently held resources are pre-empted and added to the waiting list of resources
- The process will restart when all the resources including old and new are available

Deadlock Prevention

2. Preventing No Pre-emption

- In the second protocol, if a thread requests some resources, we first check whether they are available
- If they are, we allocate them
- If they are not, we check whether they are allocated to some other process that is waiting for additional resources
- If so, we pre-empt the desired resources from the waiting process and allocate them to the requesting process
- If the resources are neither available nor held by a waiting process, the requesting process must wait

Deadlock Prevention

2. Preventing No Pre-emption

- While it is waiting, some of its resources may be pre-empted, but only if another process requests them
- A process can be restarted only when it is allocated the new resources it is requesting and recovers any resources that were pre-empted while it was waiting
- These protocols are often applied to resources whose state can be easily saved and restored later, such as CPU registers and database transactions
- It cannot generally be applied to such resources as semaphores

Deadlock Prevention

2. Preventing Circular Wait

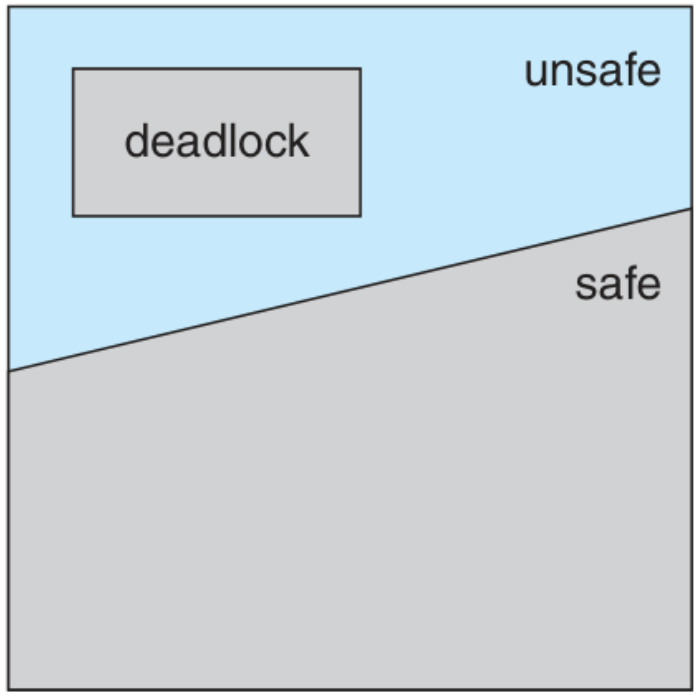
- It can be prevented using the following method
- Let R = $\{R_1, R_2, ..., R_m\}$ be the set of resource types.
- We assign to each resource type a unique integer number $F(R_i)$
- This allows us to compare two resources and to determine whether one precedes another in our ordering
- Each thread can request resources only in an increasing order of enumeration
- That is, a thread can initially request an instance of a resource—say, R_i
- After that, the thread can request an instance of resource R_j if and only if $F(R_j) > F(R_i)$
- A thread requesting an instance of resource R_j must have released any resources R_i such that $F(R_i) \geq F(R_j)$

Deadlock Avoidance

- Deadlock avoidance method allocates resources to processes in such a way that a deadlock is avoided
- Regarding resource allocation, a system can be in a safe state or an unsafe state
- A system is in a safe state, if resources can be allocated to processes in some order and still avoid a deadlock
- This order of allocation of resources is called a safe sequence
- If there is no safe sequence, then the system is in an unsafe state
- An unsafe state may lead to a deadlock
- Not all unsafe states are deadlocks
- A system can go from a safe state to an unsafe state

Deadlock Avoidance

- Safe and Unsafe System States



Deadlock Avoidance

- Consider a system with 12 resources and 3 threads: T_0, T_1, and T_2
- The maximum and current needs (at time t_0) of these threads are given below.

	Maximum Needs	Current Needs
T_0	10	5
T_1	4	2
T_2	9	2

- At time t_0, the system is in a safe state

Deadlock Avoidance

- The sequence $< T_1, T_0, T_2 >$ satisfies the safety condition
- Thread T_1 can immediately be allocated all its resources and then return them
- The system will then have 5 available resources
- Then thread T_0 can get all its resources and return them
- The system will then have 10 available resources
- Finally thread T_2 can get all its resources and return them
- The system will then have all 12 resources available

Deadlock Avoidance

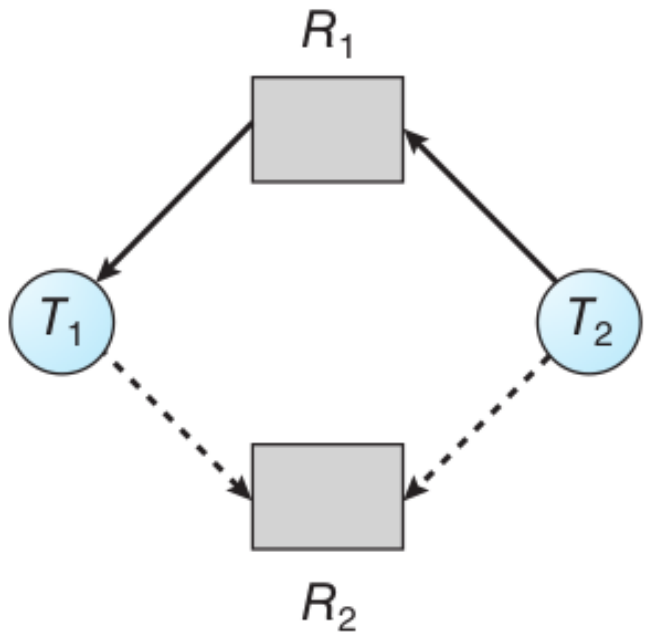
- A system can go from a safe state to an unsafe state
- Suppose that, at time t_1, thread T_2 requests and is allocated one more resource
- The system is no longer in a safe state
- At this point, only thread T_1 can be allocated all its resources
- When it returns them, the system will have only 4 available resources
- Since thread T_0 is allocated five resources but has a maximum of 10, it may request 5 more resources
- If it does so, it will have to wait, because they are unavailable
- Similarly, thread T_2 may request six additional resources and have to wait, resulting in a deadlock
- Our mistake was in granting the request from thread T_2 for one more resource
- If we had made T_2 wait until either of the other finished and released its resources, then we could have avoided the deadlock.

Deadlock Avoidance

- We can use a variant of the resource allocation graph for deadlock avoidance
- In addition to the request and assignment edges already described, we introduce a new type of edge, called a claim edge
- A claim edge $T_i \rightarrow R_j$ indicates that thread T_i may request resource R_j at some time in the future.
- This edge resembles a request edge in direction but is represented in the graph by a dashed line.
- All resources must be claimed in the beginning of execution
- When a thread requests a resource the corresponding claim edge is converted into a request edge
- Similarly, when a resource is released by a thread, the corresponding assignment edge is reconverted into a claim edge

Deadlock Avoidance

- Resource Allocation Graph



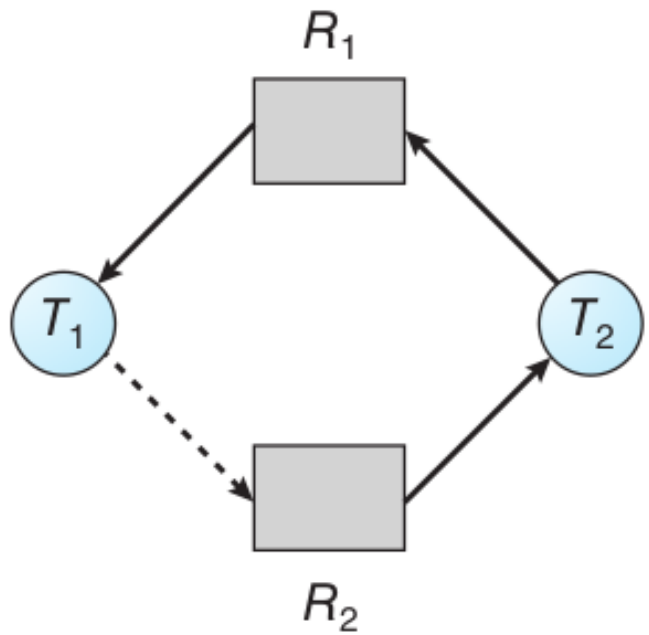
- Here T_1 and T_2 may request for R_2 in future

Deadlock Avoidance

- Resource Allocation Graph Algorithm
- Suppose that thread T_i requests resource R_j
- The request can be granted only if converting the request edge $T_i \rightarrow R_j$ to an assignment edge $R_j \rightarrow T_i$ does not result in the formation of a cycle in the resource-allocation graph
- If no cycle exists, then the allocation of the resource will leave the system in a safe state
- If a cycle is found, then the allocation will put the system in an unsafe state
- In that case, thread T_i will have to wait for its requests to be satisfied

Deadlock Avoidance

- Consider the previous resource allocation graph
- Suppose that T_2 request R_2
- Even though R_2 is currently free, we cannot allocate it to T_2 , since this action will create a cycle in the graph, as given below



- If T_1 requests R_2, then a deadlock will occur

Deadlock Avoidance

- Banker's Algorithm
- Resource allocation graph algorithm can be used for avoiding deadlocks, in the case of single instance resources
- When resources have multiple instances, Banker's algorithm can be used
- This name was chosen, since a bank allocates cash in such a way that the needs of all its customers are satisfied

Deadlock Avoidance

- Banker's Algorithm
- The following data structures are used here
- Available - It is the number of available resources of each type
- Max - The maximum resources of each type, each process might need
- Allocation - The number of resources of each type, allocated to each process
- Need - The remaining resources needed, for each process
- Need = Max - Allocation
- Request - The number of resources of each type, requested by a process

Deadlock Avoidance

- Banker's Algorithm
- There are 2 algorithms within it

1. Safety Algorithm

- This will find out whether the system is in a safe state or not

2. Resource Request Algorithm

- This will find out whether a request for a resource can be safely granted

Deadlock Avoidance

- Banker's Algorithm

1. Safety Algorithm

- Step 1 - For each process, check if it can finish with the available resources
- Step 2 - A process can finish if Need $\leq$ Available
- Step 3 - If a process can finish, calculate the future values of available resources by adding the allocated resources
- Step 4 - If all processes can finish in this way, the system is in a safe state, otherwise it is in an unsafe state

Deadlock Avoidance

- Banker's Algorithm

2. Resource Request Algorithm

- Step 1 - Let a process requests for resources
- Step 2 - If Request $\leq$ Need, go to step 3; Otherwise raise an error condition
- Step 3 - If Request $\leq$ Available, go to step 4; Otherwise the process must wait, since resources are not available
- Step 4 - Now the resources can be allocated to the process; Calculate the future values of the following data structures
  Available = Available - Request
  Allocation = Allocation + Request
  Need = Need - Request
- Step 5 - Check whether the resulting state is safe using the safety algorithm; If it is safe, then the resources can be allocated; otherwise not

Deadlock Avoidance

- Banker's Algorithm - Example
- Consider a system with 5 threads T_0 through T_4 and 3 resource types A, B, and C
- Resource type A has 10 instances, resource type B has 5 instances, and resource type C has 7 instances
- The current state of the system can be represented as given below

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
T0	0	1	0	7	5	3	3	3	2
T1	2	0	0	3	2	2			
T2	3	0	2	9	0	2			
T3	2	1	1	2	2	2			
T4	0	0	2	4	3	3			

Deadlock Avoidance

- Banker's Algorithm - Example
- Need = Max - Allocation

	Need		
	A	B	C
T0	7	4	3
T1	1	2	2
T2	6	0	0
T3	0	1	1
T4	4	3	1

Deadlock Avoidance

- Banker's Algorithm - Example
- Now we will check the safety of the current state using safety algorithm
- T_1 is the first thread for which Need $\leq$ Available
- Hence T_1 can complete first

	Need				Available			Allocation			
	A	B	C		A	B	C	A	B	C	
T0	7	4	3		3	3	2	0	1	0	
T1	1	2	2	*				2	0	0	*
T2	6	0	0					3	0	2	
T3	0	1	1					2	1	1	
T4	4	3	1					0	0	2	

Deadlock Avoidance

- Banker's Algorithm - Example
- Now for T_3 , Need $\leq$ Available
- Hence T_3 can complete next

	Need				Available			Allocation			
	A	B	C		A	B	C	A	B	C	
T0	7	4	3		5	3	2	0	1	0	
T1	1	2	2	*				2	0	0	*
T2	6	0	0					3	0	2	
T3	0	1	1	*				2	1	1	*
T4	4	3	1					0	0	2	

Deadlock Avoidance

- Banker's Algorithm - Example
- Now for T_4 , Need $\leq$ Available
- Hence T_4 can complete next

	Need				Available			Allocation			
	A	B	C		A	B	C	A	B	C	
T0	7	4	3		7	4	3	0	1	0	
T1	1	2	2	*				2	0	0	*
T2	6	0	0					3	0	2	
T3	0	1	1	*				2	1	1	*
T4	4	3	1	*				0	0	2	*

Deadlock Avoidance

- Banker's Algorithm - Example
- Now for T_0 , Need $\leq$ Available
- Hence T_0 can complete next

	Need				Available			Allocation			
	A	B	C		A	B	C	A	B	C	
T0	7	4	3	*	7	4	5	0	1	0	*
T1	1	2	2	*				2	0	0	*
T2	6	0	0					3	0	2	
T3	0	1	1	*				2	1	1	*
T4	4	3	1	*				0	0	2	*

Deadlock Avoidance

- Banker's Algorithm - Example
- Now for T_2 , Need $\leq$ Available
- Hence T_2 can complete next

	Need				Available			Allocation			
	A	B	C		A	B	C	A	B	C	
T0	7	4	3	*	7	5	5	0	1	0	*
T1	1	2	2	*				2	0	0	*
T2	6	0	0	*				3	0	2	*
T3	0	1	1	*				2	1	1	*
T4	4	3	1	*				0	0	2	*

- Thus all the threads can complete in the sequence $< T_1, T_3, T_4, T_0, T_2 >$
- Hence the current system state is safe

Deadlock Avoidance

- Banker's Algorithm - Example
- Suppose now that thread T_1 requests one additional instance of resource type A and two instances of resource type C
- Hence Request = (1,0,2)

	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
T0	0	1	0	7	4	3	3	3	2
T1	2	0	0	1	2	2			
T2	3	0	2	6	0	0			
T3	2	1	1	0	1	1			
T4	0	0	2	4	3	1			

- Request $\leq$ Available
- Hence this request can be granted if the resulting system state is safe

Deadlock Avoidance

- Banker's Algorithm - Example
- After allocation, the new system state will be

	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
T0	0	1	0	7	4	3	2	3	0
T1	3	0	2	0	2	0			
T2	3	0	2	6	0	0			
T3	2	1	1	0	1	1			
T4	0	0	2	4	3	1			

Deadlock Avoidance

- Banker's Algorithm - Example
- Now we will check the safety of the new system state using safety algorithm
- T_1 is the first thread for which Need $\leq$ Available
- Hence T_1 can complete first

	Allocation				Need				Available		
	A	B	C		A	B	C		A	B	C
T0	0	1	0		7	4	3		2	3	0
T1	3	0	2	*	0	2	0	*			
T2	3	0	2		6	0	0				
T3	2	1	1		0	1	1				
T4	0	0	2		4	3	1				

Deadlock Avoidance

- Banker's Algorithm - Example
- Now for T_3, Need $\leq$ Available
- Hence T_3 can complete next

	Allocation				Need				Available		
	A	B	C		A	B	C		A	B	C
T0	0	1	0		7	4	3		5	3	2
T1	3	0	2	*	0	2	0	*			
T2	3	0	2		6	0	0				
T3	2	1	1	*	0	1	1	*			
T4	0	0	2		4	3	1				

Deadlock Avoidance

- Banker's Algorithm - Example
- Now for T_4, Need $\leq$ Available
- Hence T_4 can complete next

	Allocation				Need				Available		
	A	B	C		A	B	C		A	B	C
T0	0	1	0		7	4	3		7	4	3
T1	3	0	2	*	0	2	0	*			
T2	3	0	2		6	0	0				
T3	2	1	1	*	0	1	1	*			
T4	0	0	2	*	4	3	1	*			

Deadlock Avoidance

- Banker's Algorithm - Example
- Now for T_0, Need $\leq$ Available
- Hence T_0 can complete next

	Allocation				Need				Available		
	A	B	C		A	B	C		A	B	C
T0	0	1	0	*	7	4	3	*	7	4	5
T1	3	0	2	*	0	2	0	*			
T2	3	0	2		6	0	0				
T3	2	1	1	*	0	1	1	*			
T4	0	0	2	*	4	3	1	*			

Deadlock Avoidance

- Banker's Algorithm - Example
- Now for T_2, Need $\leq$ Available
- Hence T_2 can complete finally

	Allocation				Need				Available		
	A	B	C		A	B	C		A	B	C
T0	0	1	0	*	7	4	3	*	7	5	5
T1	3	0	2	*	0	2	0	*			
T2	3	0	2	*	6	0	0	*			
T3	2	1	1	*	0	1	1	*			
T4	0	0	2	*	4	3	1	*			

- Thus all the threads can complete in the sequence $< T_1, T_3, T_4, T_0, T_2 >$
- The new system state is safe; Hence we can grant the request of T_1

Deadlock Avoidance

- Banker's Algorithm - Example

	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
T0	0	1	0	7	4	3	2	3	0
T1	3	0	2	0	2	0			
T2	3	0	2	6	0	0			
T3	2	1	1	0	1	1			
T4	0	0	2	4	3	1			

- Suppose now that thread T_4 makes a Request = (3,3,0)
- This request cannot be granted since the resources are not available

Deadlock Avoidance

- Banker's Algorithm - Example

	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
T0	0	1	0	7	4	3	2	3	0
T1	3	0	2	0	2	0			
T2	3	0	2	6	0	0			
T3	2	1	1	0	1	1			
T4	0	0	2	4	3	1			

- Suppose now that thread T_0 makes a Request = (0,2,0)
- Request $\leq$ Available
- Hence this request can be granted if the resulting system state is safe

Deadlock Avoidance

- Banker's Algorithm - Example
- After allocation, the new system state will be

	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
T0	0	3	0	7	2	3	2	1	0
T1	3	0	2	0	2	0			
T2	3	0	2	6	0	0			
T3	2	1	1	0	1	1			
T4	0	0	2	4	3	1			

Deadlock Avoidance

- Banker's Algorithm - Example
- Now we will check the safety of the new system state using safety algorithm
- There are no threads for which Need $\leq$ Available
- The resulting system state is unsafe; Hence the request by T_0 cannot be granted

	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
T0	0	3	0	7	2	3	2	1	0
T1	3	0	2	0	2	0			
T2	3	0	2	6	0	0			
T3	2	1	1	0	1	1			
T4	0	0	2	4	3	1			

Deadlock Detection

- If a system does not use either a deadlock prevention algorithm or a deadlock avoidance algorithm, deadlock will occur
- In that case, the following deadlock detection algorithms are used for identifying a deadlock situation

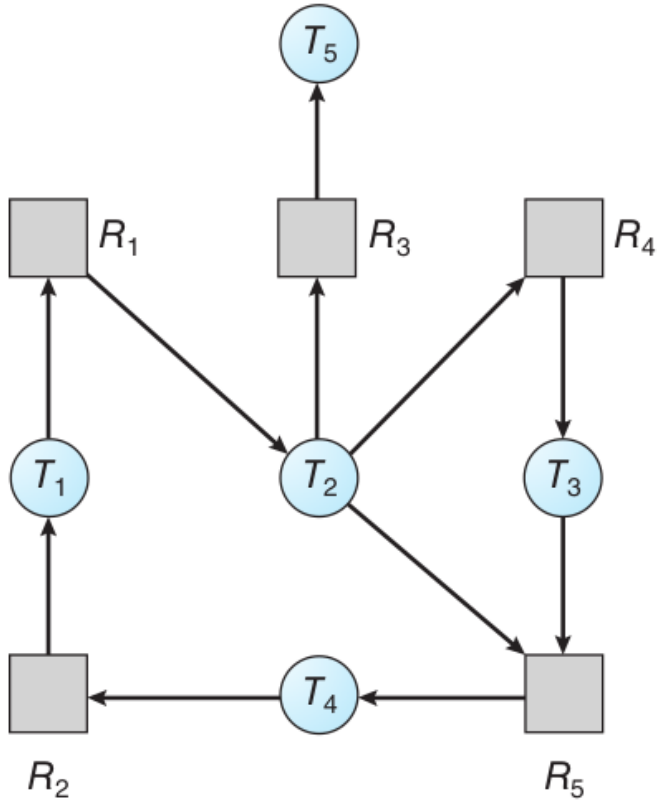
1. Deadlock detection algorithm for single instance resource types
2. Deadlock detection algorithm for multi instance resource types

Deadlock Detection

1. Deadlock detection algorithm for single instance resource types

- Step 1 - Construct a wait-for graph from the resource allocation graph by removing the resource nodes and collapsing the appropriate edges
- In a wait-for graph, there is an edge from T_i to T_j, if T_i is waiting for a resource that has been allocated to T_j
- Step 2 - A deadlock exists in the system if and only if the wait-for graph contains a cycle

Deadlock Detection



- (a) Resource Allocation Graph
- (b) Wait-For Graph
- Since the wait-for graph contains cycles, deadlock is detected here

Deadlock Detection

1. Deadlock detection algorithm for multi instance resource types

- The following data structures are used here
- Available - It is the number of available resources of each type
- Allocation - The number of resources of each type, allocated to each process
- Request - The number of resources of each type, requested by a process
- Finish - It indicates whether a process has finished or not

Deadlock Detection

1. Deadlock detection algorithm for multi instance resource types

- Step 1 - If Allocation $\neq 0$, for a process, then Finish = false, else Finish = true
- Step 2 - If Finish == false and Request $\leq$ Available,for a process, then goto step 3, else goto step 4
- Step 3 - Here the requested resources are allocated to the process, and after its completion
  Available = Available + Allocation
  Finish = true, for that process
  Here we assume that this process may not need any resource later
  If our assumption is incorrect, then deadlock may happen later, which will be detected when we execute this algorithm later
- Step 4 - If Finish == false for some process, then the system is in a deadlocked state

Deadlock Detection

- Deadlock detection algorithm for multi instance resource types - Example
- Consider a system with 5 threads T_0 through T_4 and 3 resource types A, B, and C
- Resource type A has 7 instances, resource type B has 2 instances, and resource type C has 6 instances
- The current state of the system can be represented as given below

	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
T0	0	1	0	0	0	0	0	0	0
T1	2	0	0	2	0	2			
T2	3	0	3	0	0	0			
T3	2	1	1	1	0	0			
T4	0	0	2	0	0	2			

Deadlock Detection

- Deadlock detection algorithm for multi instance resource types - Example
- Here Request $\leq$ Available for T_0
- Hence T_0 can complete first

	Allocation				Request			Available		
	A	B	C		A	B	C	A	B	C
T0	0	0	0	*	0	0	0	0	1	0
T1	2	0	0		2	0	2			
T2	3	0	3		0	0	0			
T3	2	1	1		1	0	0			
T4	0	0	2		0	0	2			

Deadlock Detection

- Deadlock detection algorithm for multi instance resource types - Example
- Now Request $\leq$ Available for T_2
- Hence T_2 can complete next

	Allocation				Request			Available		
	A	B	C		A	B	C	A	B	C
T0	0	0	0	*	0	0	0	3	1	3
T1	2	0	0		2	0	2			
T2	0	0	0	*	0	0	0			
T3	2	1	1		1	0	0			
T4	0	0	2		0	0	2			

Deadlock Detection

- Deadlock detection algorithm for multi instance resource types - Example
- Now Request $\leq$ Available for T_3
- Hence T_3 can complete next

	Allocation				Request			Available		
	A	B	C		A	B	C	A	B	C
T0	0	0	0	*	0	0	0	5	2	4
T1	2	0	0		2	0	2			
T2	0	0	0	*	0	0	0			
T3	0	0	0	*	0	0	0			
T4	0	0	2		0	0	2			

Deadlock Detection

- Deadlock detection algorithm for multi instance resource types - Example
- Now Request $\leq$ Available for T_4
- Hence T_4 can complete next

	Allocation				Request			Available		
	A	B	C		A	B	C	A	B	C
T0	0	0	0	*	0	0	0	5	2	6
T1	2	0	0		2	0	2			
T2	0	0	0	*	0	0	0			
T3	0	0	0	*	0	0	0			
T4	0	0	0	*	0	0	0			

Deadlock Detection

- Deadlock detection algorithm for multi instance resource types - Example
- Now Request $\leq$ Available for T_1
- Hence T_1 can complete finally
- Thus all the threads can complete in the sequence $< T_0,\ T_2,\ T_3,\ T_4,\ T_1 >$

	Allocation				Request			Available		
	A	B	C		A	B	C	A	B	C
T0	0	0	0	*	0	0	0	7	2	6
T1	0	0	0	*	0	0	0			
T2	0	0	0	*	0	0	0			
T3	0	0	0	*	0	0	0			
T4	0	0	0	*	0	0	0			

Deadlock Detection

- Deadlock detection algorithm for multi instance resource types - Example
- Suppose now that thread T_2 makes one additional request for an instance of type C
- The Request matrix is modified as follows
- The current state of the system can be represented as given below

	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
T0	0	1	0	0	0	0	0	0	0
T1	2	0	0	2	0	2			
T2	3	0	3	0	0	1			
T3	2	1	1	1	0	0			
T4	0	0	2	0	0	2			

Deadlock Detection

- Deadlock detection algorithm for multi instance resource types - Example
- Here Request $\leq$ Available for T_0
- Hence T_0 can complete first

	Allocation				Request			Available		
	A	B	C		A	B	C	A	B	C
T0	0	0	0	*	0	0	0	0	1	0
T1	2	0	0		2	0	2			
T2	3	0	3		0	0	1			
T3	2	1	1		1	0	0			
T4	0	0	2		0	0	2			

Deadlock Detection

- Deadlock detection algorithm for multi instance resource types - Example
- Now there are no threads for which Request $\leq$ Available
- Hence deadlock exists, consisting of threads T_1, T_2, T_3 and T_4

	Allocation				Request			Available		
	A	B	C		A	B	C	A	B	C
T0	0	0	0	*	0	0	0	0	1	0
T1	2	0	0		2	0	2			
T2	3	0	3		0	0	1			
T3	2	1	1		1	0	0			
T4	0	0	2		0	0	2			

Deadlock Recovery

- Recovery from deadlock is done using either of the below given methods

1. Process and Thread Termination

- Here we terminate a process or a thread using the below given methods

(a) Abort all deadlocked processes

- This will be at a great expense, since we have to again run these processes, which involves recomputation

(b) Abort one process at a time until the deadlock cycle is eliminated

- After each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked

Deadlock Recovery

2. Resource Preemption

- We successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken
- The following issues need to be addressed here

(a) Selecting a victim

- We need to decide which resources from which processes are to be preempted
- It depends on the number of resources held by a process as well as the amount of time a process has so far consumed

(b) Rollback

- When resources are preempted, processes need to be rolled back
- We can perform the rollback either fully or partially

(c) Starvation

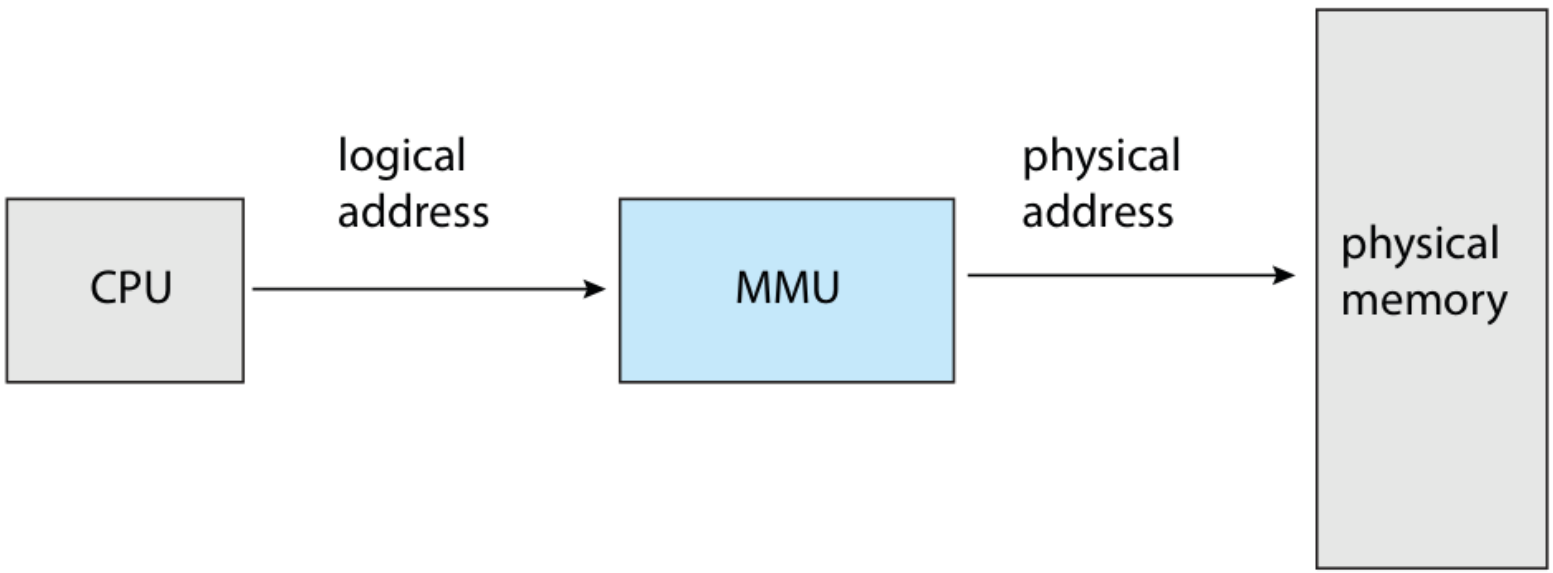
- We need to ensure that resources will not always be preempted from the same process

Module 3
Memory Management:
Main Memory - Logical and Physical Address Space

- It refers to the management of main memory, where data and programs are stored
- In operating systems, physical and logical addresses are used to access memory
- A physical address is the actual address in main memory where data and programs are stored
- An address generated by the CPU during program execution is called a logical address (virtual address)
- The set of all logical addresses generated by a program is a logical address space
- The set of all physical addresses corresponding to these logical addresses is a physical address space
- The run-time mapping from logical to physical addresses is done by a hardware device called the memory-management unit (MMU)

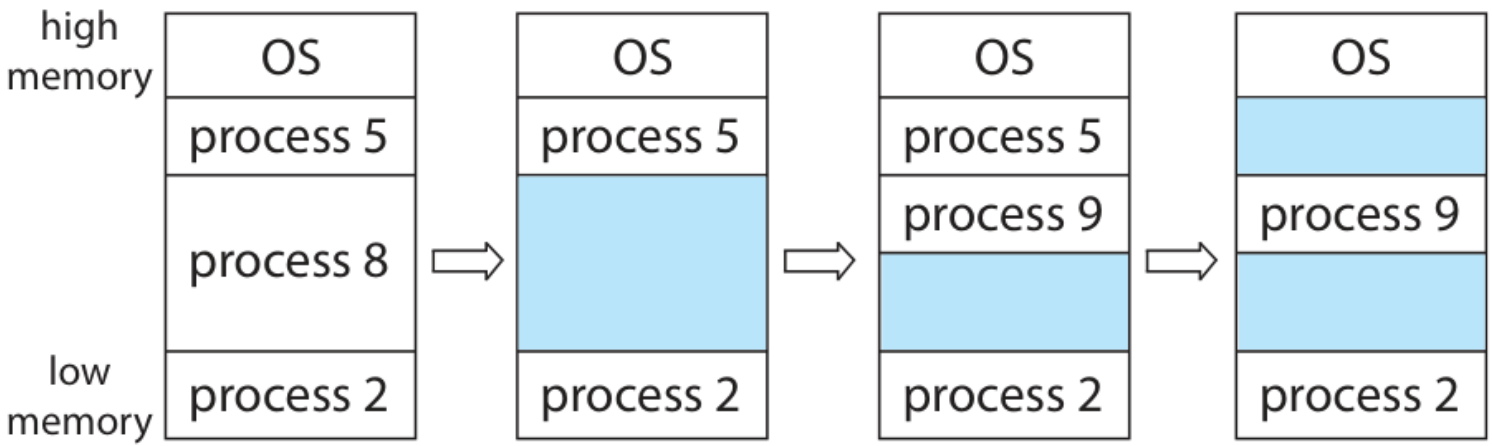
Logical and Physical Address Space

- Memory Management Unit



Memory Allocation

- Each process is allocated memory depending on its requirement
- Thus memory will be divided into variable size partitions, where each partition is allocated to exactly one process
- This memory allocation scheme is called variable partition scheme



- Here an unallocated memory block is called a hole

Memory Allocation

- We can use the following strategies for allocating memory to processes

1. First fit

- Allocate the first hole that is big enough for a process

2. Best fit

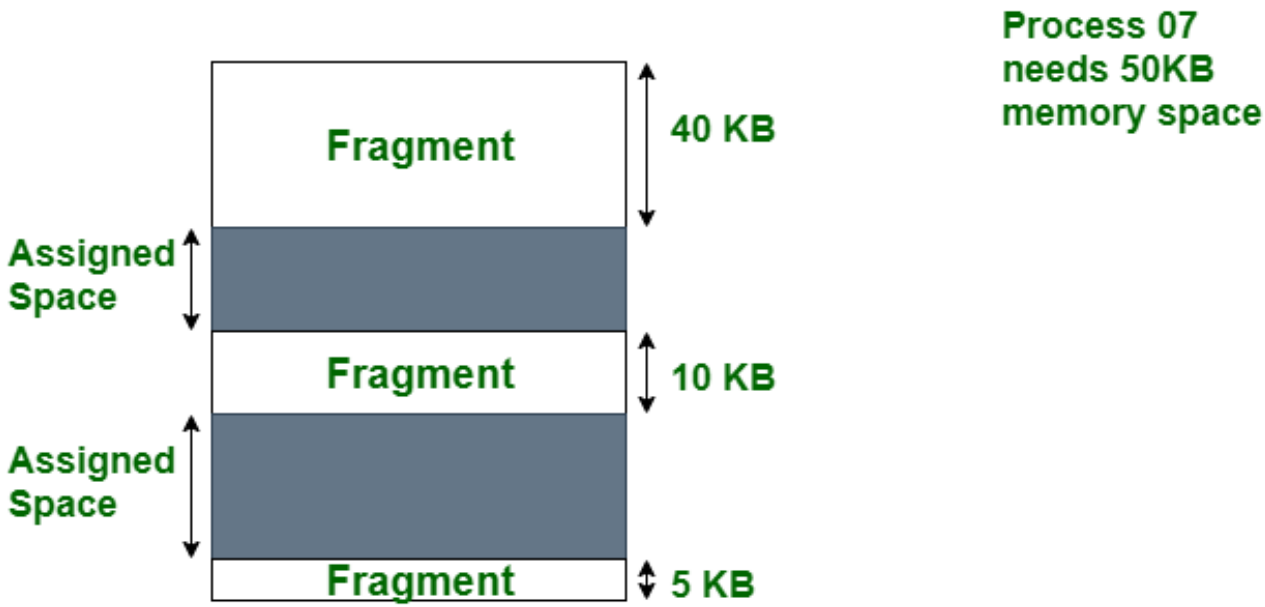
- Allocate the smallest hole that is big enough for a process

3. Worst fit

- Allocate the largest hole for a process
- First fit and best fit methods are better than worst fit method regarding memory utilisation
- First fit method is faster than best fit method

Fragmentation

- Both the first-fit and best-fit strategies for memory allocation suffer from external fragmentation
- External fragmentation exists when there is enough total memory space to satisfy a request but the available spaces are not contiguous: storage is fragmented into a large number of small holes.



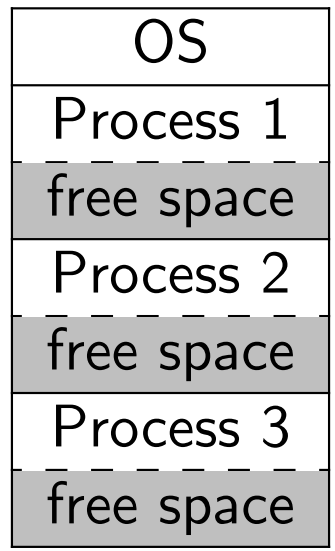
- Even though 55 KB memory is free, it cannot be allocated, since is not contiguous

Fragmentation

- One solution to the problem of external fragmentation is compaction
- Here the free memory blocks are combined into a single large memory block
- However, this is not always possible
- Another solution is to permit the address space of processes to be noncontiguous

Fragmentation

- Another type of fragmentation is internal fragmentation, which occurs in fixed size partitioning scheme
- Here memory is divided into fixed size blocks and each process is allocated one or more blocks, depending upon its requirement

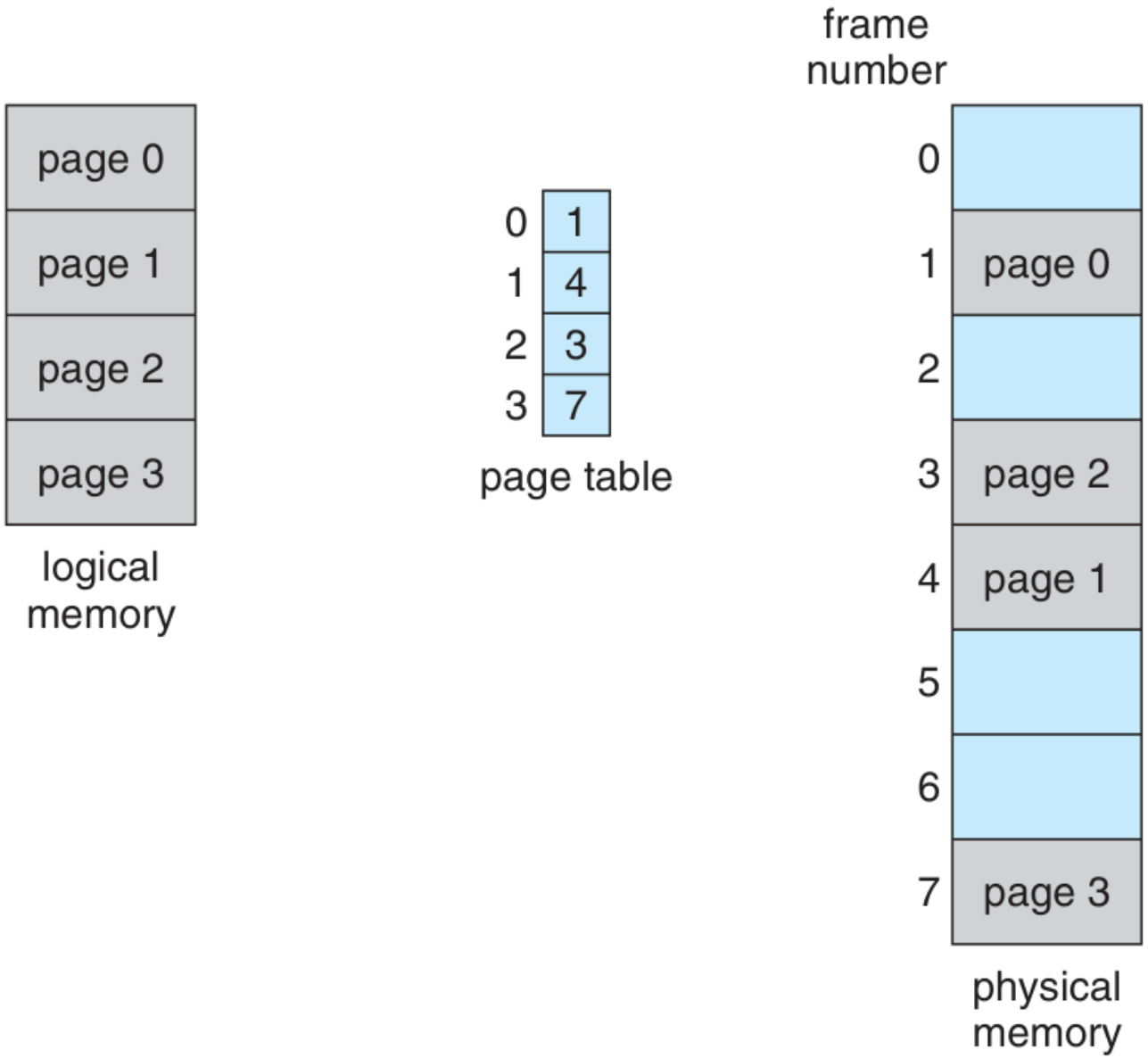


- Here the memory allocated to a process may be larger than its requirement
- Hence there are unused memory within each block; This is called internal fragmentation

Paging

- Paging is a memory management scheme that permits a process's physical address space to be non-contiguous
- It avoids external fragmentation
- Here physical memory is divided into fixed-sized blocks called frames and logical memory is divided into blocks of the same size called pages
- When a process is to be executed, its pages are loaded into any available memory frames

Paging


page 0
page 1
page 2
page 3
logical memory
0 1
1 4
2 3
3 7
page table
frame number
0
1 page 0
2
3 page 2
4 page 1
5
6
7 page 3
physical memory

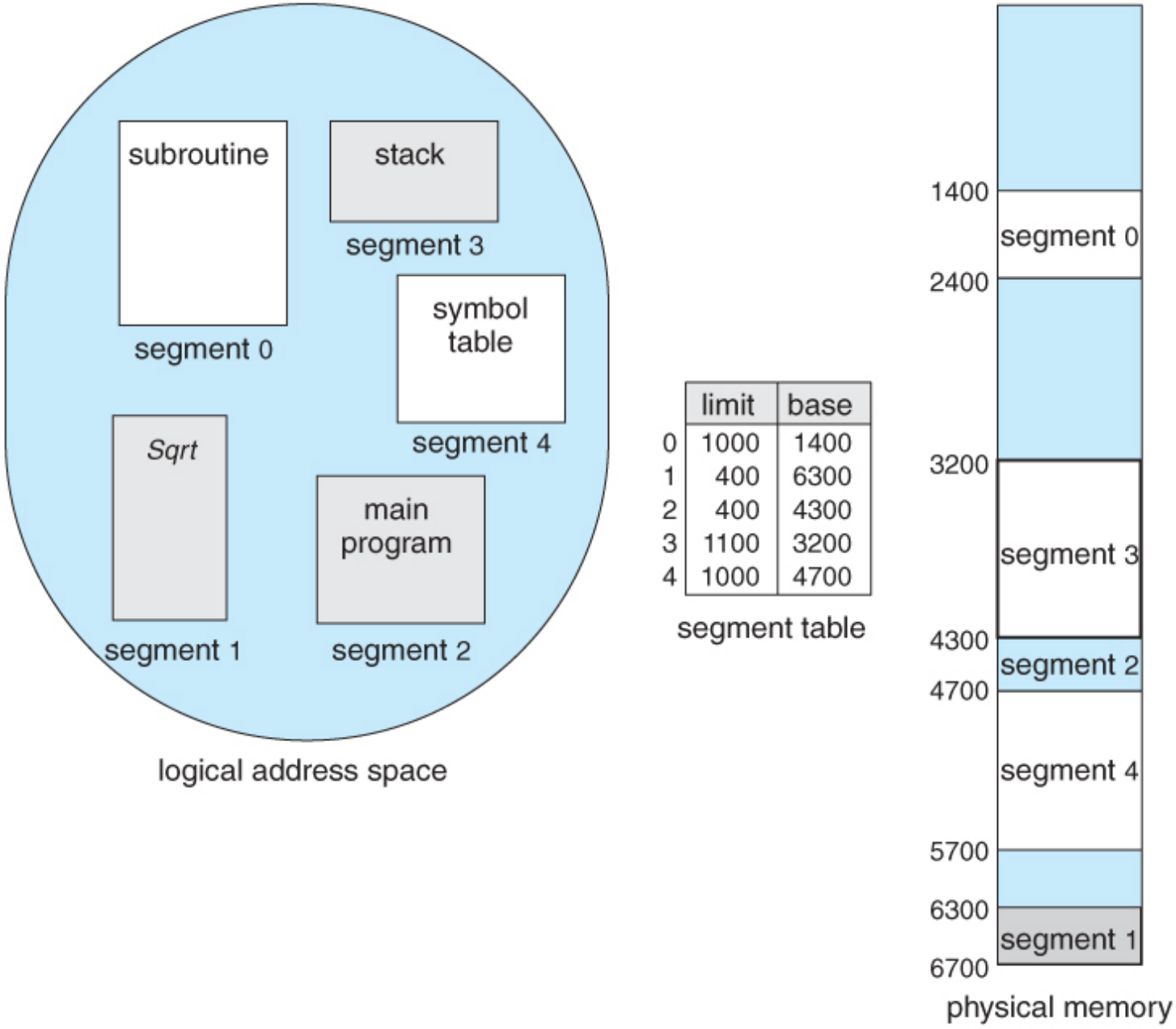
Paging

- The operating system maintains a data structure called page table, for mapping logical addresses to physical addresses
- A page table contains the following information
- page number
- A number that identifies the page being referenced
- frame number
- A number that identifies the frame being referenced
- status bits
- They provide additional information about the page

Segmentation

- Segmentation is a memory management scheme in which logical memory is divided into blocks of varying size called segments
- When a process is to be executed, its segments are loaded into any available memory
- Segmentation matches the user's logical view of a program containing functions, arrays, modules etc. each with varying size
- Segmentation eliminates internal fragmentation
- Segmentation may lead to external fragmentation

Segmentation



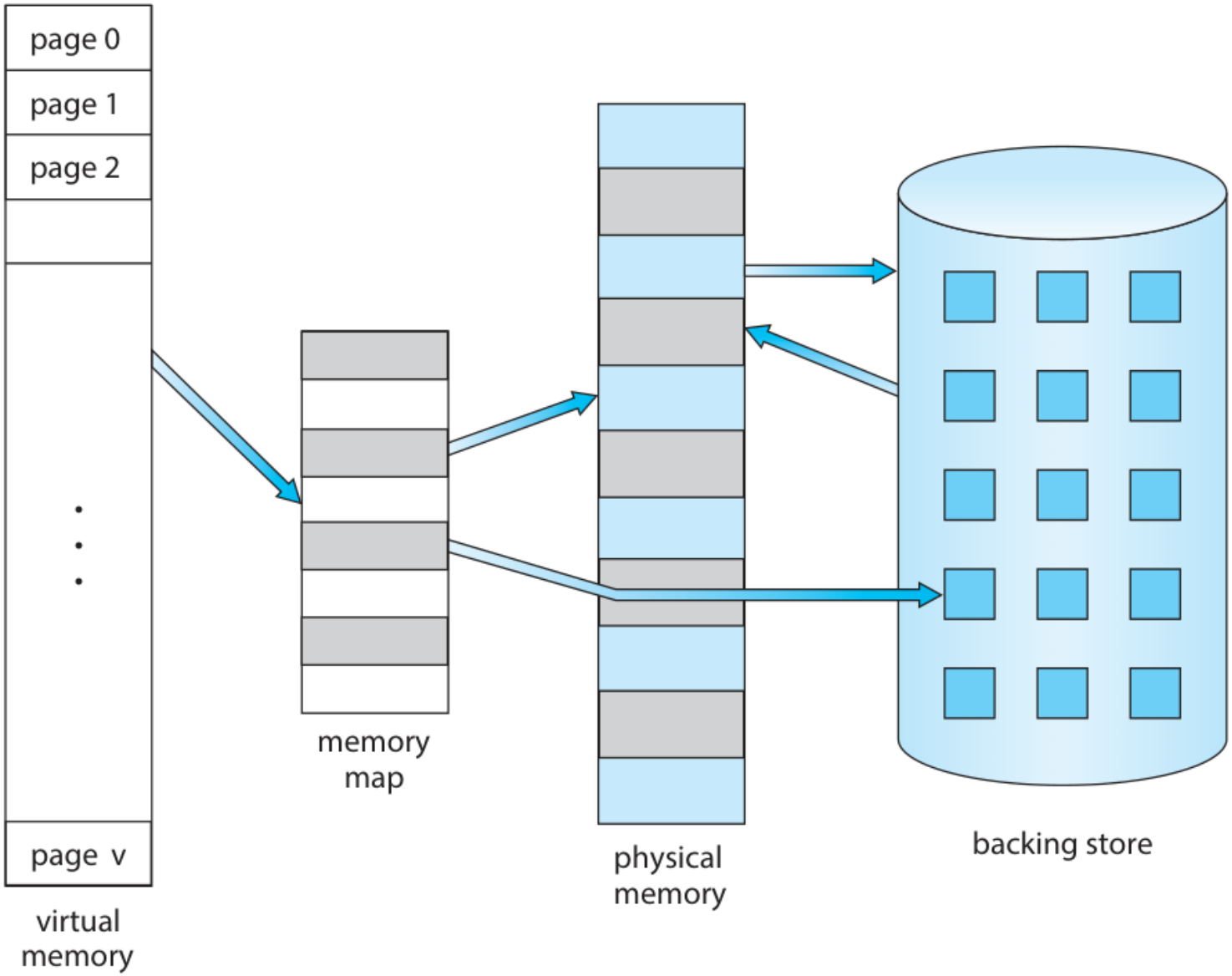
Segmentation

- The operating system maintains a data structure called segment table, for mapping logical addresses to physical addresses
- The main contents in a segment table are as follows
- segment number
- A number that identifies the segment being referenced
- base address
- The starting physical address in memory where the segment is loaded
- limit
- The length of the segment, indicating how much memory is allocated for it

Virtual Memory

- Virtual memory is a technique that allows the execution of processes that are not completely in memory.
- Only the portions of programs that are currently executing need to be in main memory
- The remaining portions are in secondary memory and are loaded into main memory only when needed
- One major advantage of this scheme is that programs can be larger than physical memory

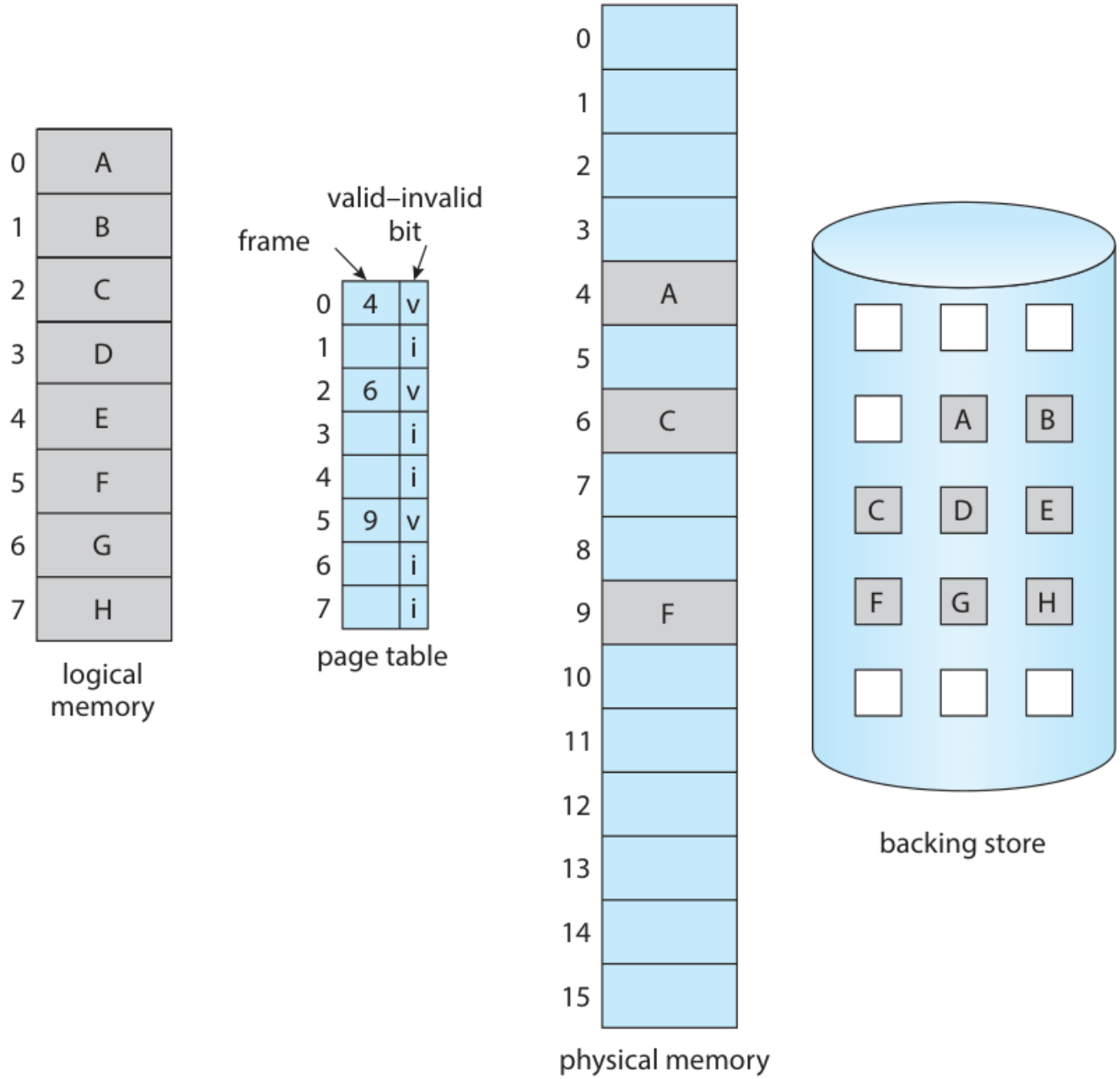
Virtual Memory



Demand Paging

- Demand Paging is a technique used in virtual memory systems
- In this technique pages are loaded from secondary memory to main memory, when demanded during program execution
- The page table used in demand paging contains one additional entry called valid-invalid bit
- If a page is in main memory, this bit is set to v(valid)
- If a page is not in main memory, this bit is set to i(invalid)

Demand Paging

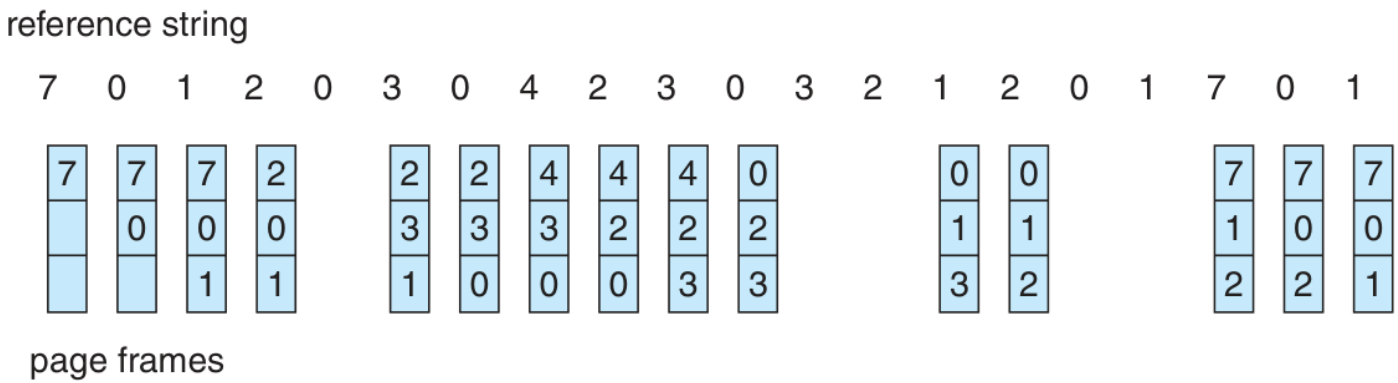


Page Replacement Algorithms

- When a process tries to access a page that is not brought into main memory (a page with invalid bit), a page fault occurs
- When a page fault occurs, the operating system must bring the desired page from secondary storage into main memory
- The technique used here is called page replacement
- In this approach, if a frame is free the page is brought there
- Otherwise one of the existing frames is replaced by the desired page
- We will be using either FIFO, Optimal or LRU page replacement algorithms for this

FIFO Page Replacement Algorithm

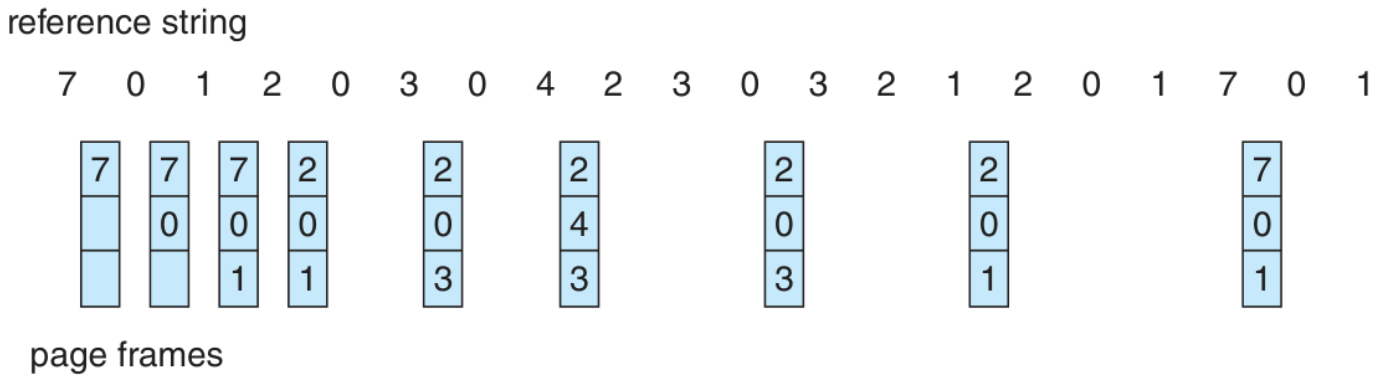
- In this algorithm, the desired page replaces the oldest page,i.e. the page which arrived in memory at the earliest period of time



- We 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
- Here the number of page faults is 15

Optimal Page Replacement Algorithm

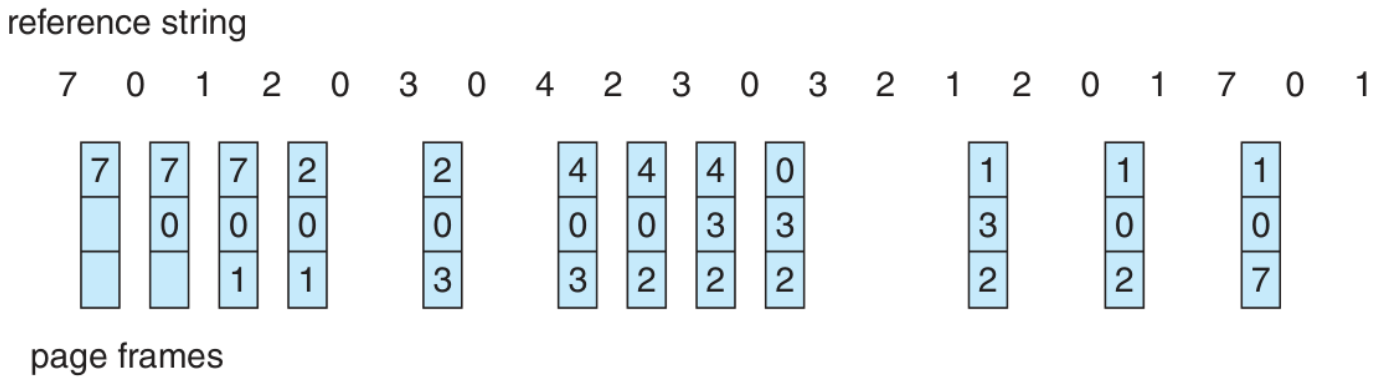
- In this algorithm, the desired page replaces the page that will not be used for the longest period of time.



- Here the number of page faults is 9
- This algorithm guarantees the lowest possible page-fault rate for a fixed number of frames.
- But this algorithm is difficult to implement, because it requires future knowledge of the reference string.

LRU Page Replacement Algorithm

- In LRU (Least Recently Used) page replacement algorithm, the desired page replaces the page that has not be used for the longest period of time.



- Here the number of page faults is 12
- Its performance is better than FIFO page replacement algorithm
- But this algorithm is also difficult to implement, because it requires past knowledge of the reference string.

Module 4
File System

- The file system consists of two distinct parts

1. A collection of files, each storing related data
2. A directory structure, which organises and provides information about all the files in the system

- File systems live on secondary storage devices

File Concept

- A file is a named collection of related information that is recorded on secondary storage
- Depending on the information stored in a file, there are different types of files
- A text file is a sequence of printable characters organised into lines
- It does not contain any special formatting such as bold text, italic text etc.
- A source file is a file containing a program written in a programming language
- An executable file is a file containing a program capable of being executed by a computer

File Attributes

- File attributes are different kinds of metadata stored about files

1. Name
- The name of a file
2. Identifier
- A unique tag, usually a number, used for identifying a file
3. Type
- The type of a file such as .txt, .mp3, .mp4 etc.
4. Location
- The location of a file on a device
5. Size
- The current size of a file
6. Protection
- Access-control information which determines who can do reading,writing, executing, and so on.
7. Timestamps
- Various timestamps like time of creation, last modification time, last access time etc.

File Operations

- The following operations are performed on a file

1. Create

- We need to find space for the file in the file system
- An entry for the file is to be made in a directory

2. Open

- All operations except create and delete require a file open first

3. Write

- The system must keep a write pointer to the location in the file where the next write is to take place

File Operations

- The following operations are performed on a file

4. Read

- The system must keep a read pointer to the location in the file where the next read is to take place

5. Reposition(Seek)

- The system also keeps a current file position pointer to the location in the file where the current operation is performed
- The current file position pointer can be repositioned to a new location

6. Delete

- All the contents of a file are erased and the file space is released

7. Truncate

- All the contents of a file are erased, but its attributes are kept

File Types

- Common file types

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, perl, asm	source code in various languages
batch	bat, sh	commands to the command interpreter
markup	xml, html, tex	textual data, documents
word processor	xml, rtf, docx	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	gif, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	rar, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, mp3, mp4, avi	binary file containing audio or A/V information

Text Books

1. A. Silberchatz et.al., "Operating System Concepts", 9th Edition Wiley (2015)