

UNIT3



Concurrent Process: Introduction

- ❖ What are the consequences and difficulty of the co-existence of multiple processes within one system?
- ❖ What are the effects of processes competing for resources?
- ❖ How are the need of processes which want to communicate with each other?



Introduction



- ❖ **Multiprogramming:** Management of multiple processes within a set of processor system
- ❖ **Multiprocessing:** Management of multiple processes within a multiprocessor
- ❖ **Distributed Processing:** Management of multiple processes executing on multiple distributed computer system



Key Terms related to Concurrency



atomic operation	A sequence of one or more statements that appears to be indivisible; that is, no other process can see an intermediate state or interrupt the operation.
critical section	A section of code within a process that requires access to shared resources and that must not be executed while another process is in a corresponding section of code.
deadlock	A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something.
livelock	A situation in which two or more processes continuously change their states in response to changes in the other process(es) without doing any useful work.
mutual exclusion	The requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources.
race condition	A situation in which multiple threads or processes read and write a shared data item and the final result depends on the relative timing of their execution.
starvation	A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen.



What is Concurrency?

❖ What is Concurrency?

- Communication among Processes
- Sharing of and competing for Resources
- Synchronization of Activities of Multiple Processes
- Allocation of Processor Time to Processes



Race Conditions

- ❖ A Race Condition occurs, if two or more processes/threads access and manipulate the same data concurrently, and the outcome of the execution depends on the particular order in which the access takes place.
- ❖ Synchronization is needed to prevent race conditions from happening.



Principles of concurrency

❖ Problems encountered during concurrency

- Sharing of global Resources is Troubled with Risk
- Difficulty in managing the allocation of Resources optimally
- Difficulty in locating a programming error because results are typically not Deterministic and Reproducible

OS Concerns



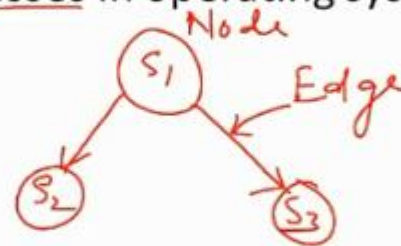
❖ What design and management issues are raised by concurrency?

- OS should keep track of active processes
- OS should Allocate and De-allocate resources to active processes
- Processor Time/Memory/Files/I-O Devices
- OS should protect against the interference by other processes
- Result of a process should be independent of the speed of execution relative to other concurrent process (Process Interaction)

Precedence Graph

❖ **Precedence Graph** is a directed acyclic graph which is used to show the execution level of several processes in operating system.

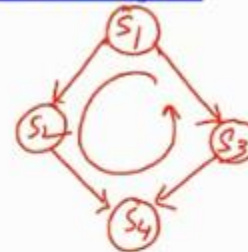
❖ It consists of nodes and edges.



❖ Nodes represent the processes and the edges represent the flow of execution.

Properties of Precedence Graph

- ❖ It is a directed graph.
- ❖ It is an acyclic graph.
- ❖ Nodes of graph correspond to individual statements of program code.
- ❖ Edge between two nodes represents the execution order.
- ❖ A directed edge from node A to node B shows that statement A executes first and then Statement B executes.



Example-1

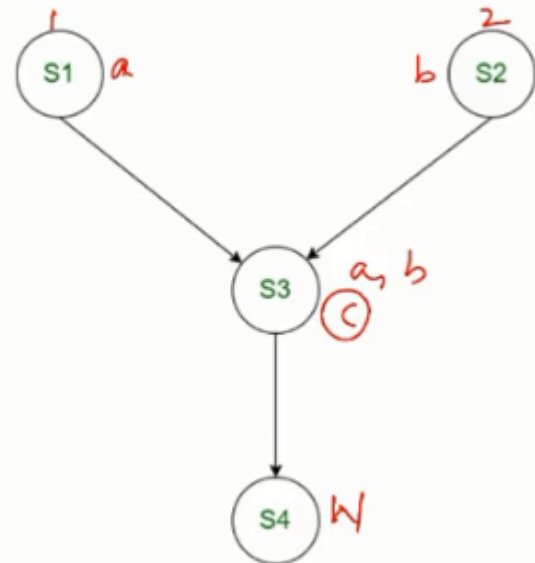
S1 : a = x + y;

S2 : b = z + 1;

S3 : c = a - b;

S4 : w = c + 1;

If above code is executed concurrently,
the following precedence relations exist:

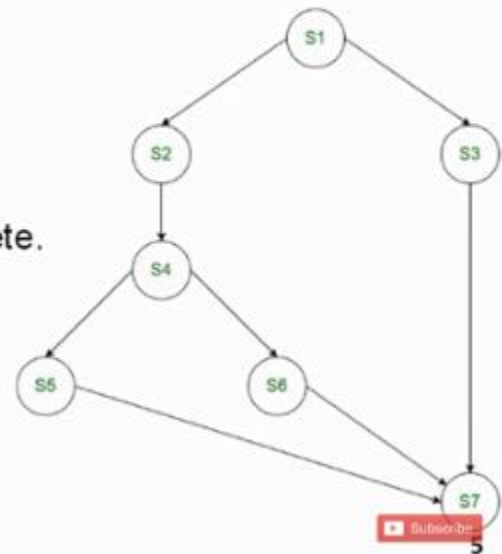


Example-2

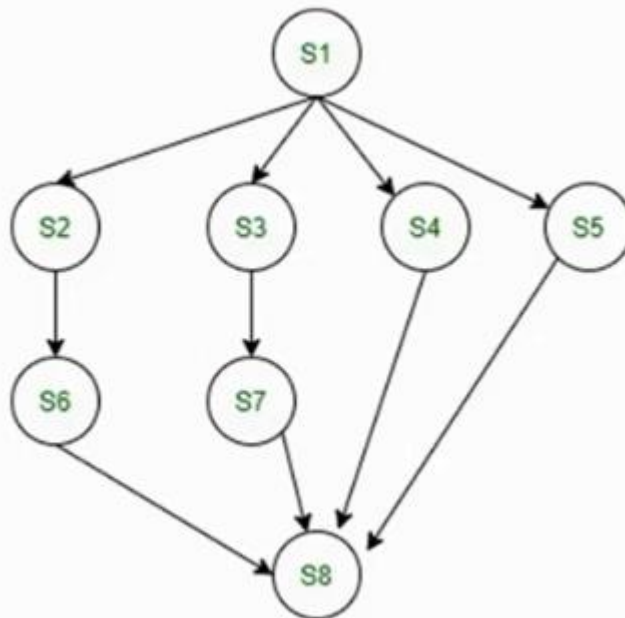


❖ Consider the following precedence relations of a program:

- S2 and S3 can be executed after S1 completes.
- S4 can be executed after S2 completes.
- S5 and S6 can be executed after S4 completes.
- S7 can be executed after S5, S6 and S3 complete.



Concurrent Program for Precedence Graph



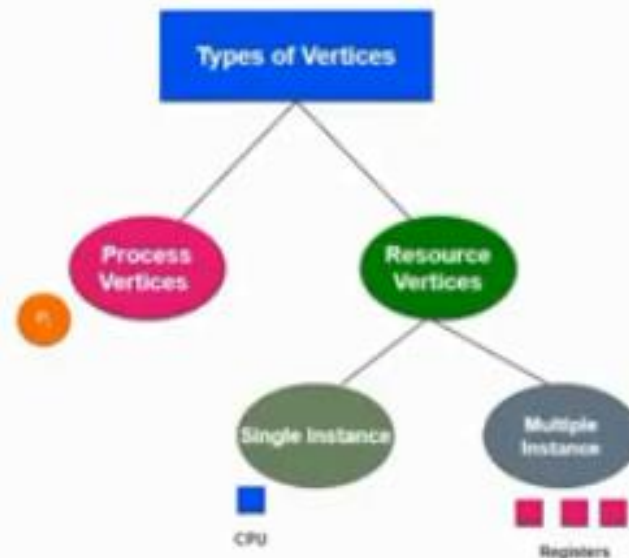
```
begin
  S1:
    begin
      begin
        S2;
        S6;
      end
      begin
        S3;
        S7;
      end
      S4;
      S5;
    end
    S8;
  end
end
```

Resource Allocation Graph: Introduction

- ❖ The resource allocation graph is the pictorial representation of the state of a system.
- ❖ The resource allocation graph is the complete information about all the processes which are **holding** some resources or **waiting** for some resources.
- ❖ It also contains the information about all the instances of all the resources whether they are available or being used by the processes.
- ❖ In this graph, the process is represented by a **Circle** while the Resource is represented by a **Rectangle**.

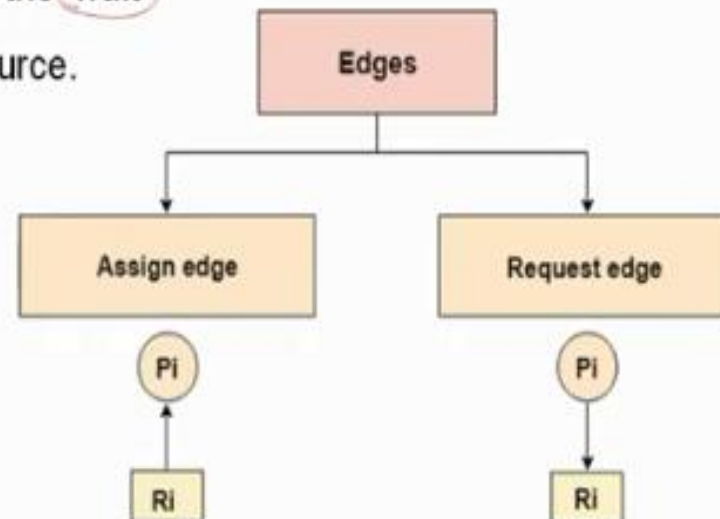
Vertices

- ❖ Vertices are mainly of two types, Resource and process.
- ❖ Each of them will be represented by a different shape.
- ❖ Circle represents process while rectangle represents resource.
- ❖ A resource can have more than one instance. Each instance will be represented by a dot inside the rectangle.



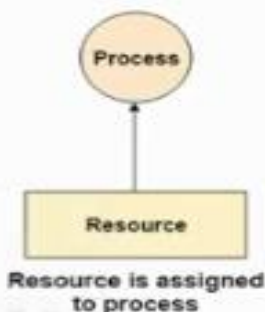
Edges

- ❖ Edges are also of two types, one represents assignment and other represents the wait of a process for a resource.

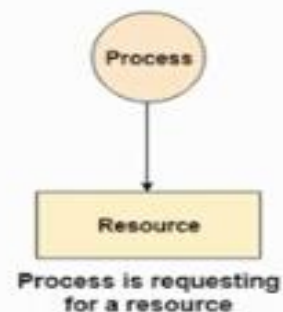


Resource Allocation and Request

- ❖ A resource is shown as assigned to a process if the tail of the arrow is attached to an instance to the resource and the head is attached to a process.

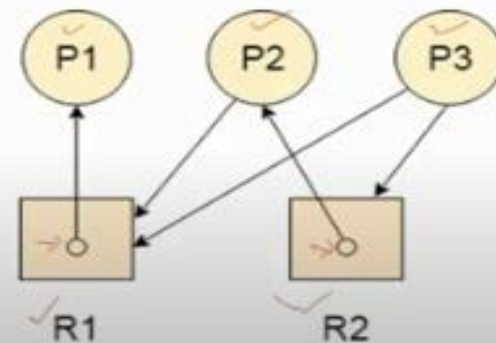


- ❖ A process is shown as waiting for a resource if the tail of an arrow is attached to the process while the head is pointing towards the resource.



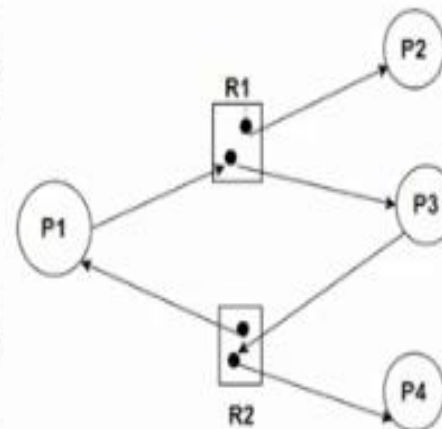
Single Instance RAG Example

- ❖ Consider 3 processes **P1**, **P2**, and **P3**, and two types of resources **R1** and **R2**.
- ❖ The resources are having 1 instance each.
- ❖ According to the graph, **R1** is being used by **P1**, **P2** is holding **R2** and waiting for **R1**, **P3** is waiting for **R1** as well as **R2**.
- ❖ The graph is deadlock free since no cycle is being formed in the graph.



Multiple Instance RAG Example

- ❖ Suppose there are four processes P1, P2, P3, P4 and there are two instances of resource R1 and two instances of resource R2.
- ❖ One instance of R2 is assigned to process P1 and another instance of R2 is assigned to process P4, Process P1 is waiting for resource R1.
- ❖ One instance of R1 is assigned to Process P2 while another instance of R2 is assigned to process P3, Process P3 is waiting for resource R2.



Deadlock Characterization: Introduction



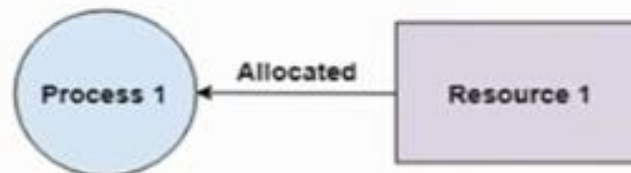
- ❖ Deadlock characterization describes the distinctive features that are the cause of deadlock occurrence.
- ❖ Deadlock is a condition in the multiprogramming environment where the executing processes get stuck in the middle of execution waiting for the resources that have been held by the other waiting processes thereby preventing the execution of the processes.

Conditions for Deadlock

- ❖ This Lecture discuss the characteristics that are essential for the occurrence of deadlock.
- ❖ The four conditions that must sustain at the same time to occur a deadlock are:
 1. Mutual Exclusion
 2. Hold and Wait
 3. No Preemption
 4. Circular Wait
- ❖ Deadlock Characterization
 - Deadlock Prerequisites
 - Systems Resource Allocation Graph

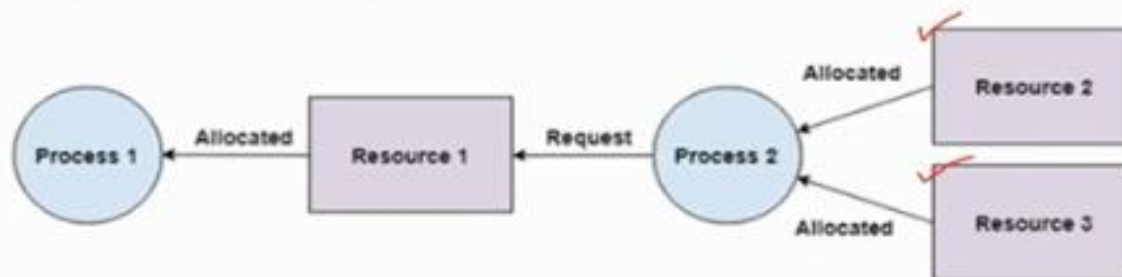
Deadlock Prerequisites: 1. Mutual Exclusion

- ❖ There should be a resource that can only be held by one process at a time.
- ❖ In the diagram, there is a single instance of Resource 1 and it is held by Process 1 only.



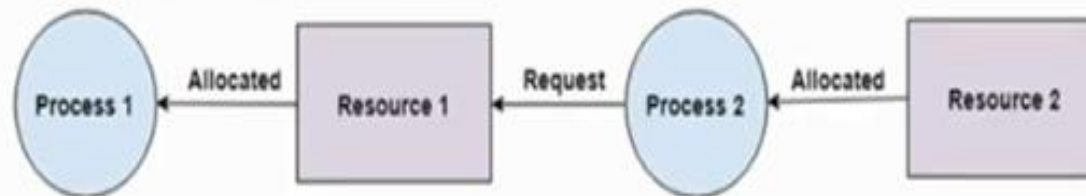
Deadlock Prerequisites: 2. Hold and Wait

- ❖ A process can hold multiple resources and still request more resources from other processes which are holding them.
- ❖ In the diagram given below, Process 2 holds Resource 2 and Resource 3 and is requesting the Resource 1 which is held by Process 1.



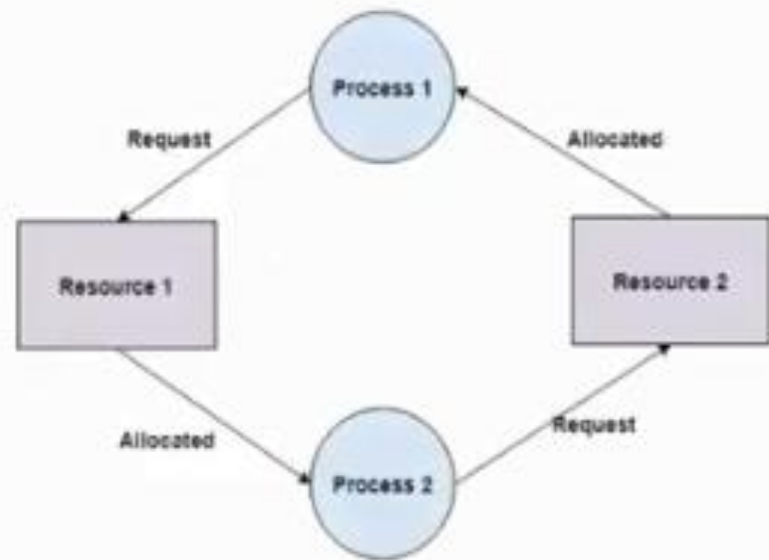
Deadlock Prerequisites: 3. No Preemption

- ❖ A resource cannot be preempted from a process by force.
- ❖ A process can only release a resource voluntarily.
- ❖ In the diagram below, Process 2 cannot preempt Resource 1 from Process 1.
- ❖ It will only be released when Process 1 surrenders it voluntarily after its execution is complete.



Deadlock Prerequisites: 4. Circular Wait

- ❖ A process is waiting for the resource held by the second process, which is waiting for the resource held by the third process and so on, till the last process is waiting for a resource held by the first process.
- ❖ This forms a circular chain.
- ❖ **For Example:** Process 1 is allocated Resource 2 and it is requesting Resource 1.



System Resource Allocation Graph

- ❖ The system reallocation graph is a directed graph that briefs you about the deadlock more precisely.
 - ❖ Like every graph, it also has a set of vertices and a set of edges.
 - ❖ Further, the set of vertices can be classified into two types of nodes P and R.
 - ❖ Where P is the set of vertices indicating the set of active processes and R is the set of vertices indicating all types of resources in the system.
 - ❖ When a process requests for a resource it denoted by the request edge in the resource-allocation graph.
 - ❖ The request edge is a directed edge from the requesting process P_i to requested resource R_j i.e. $P_i \rightarrow R_j$.
-

DEADLOCKS

Strategy

HOW TO HANDLE DEADLOCKS – GENERAL STRATEGIES

There are three methods:

Ignore Deadlocks:



Most Operating systems do this!!

Ensure deadlock **never** occurs using either

Prevention Prevent any one of the 4 conditions from happening.

Avoidance Allow all deadlock conditions, but calculate cycles about to happen and stop dangerous operations..

Allow deadlock to happen. This requires using both:

Detection Know a deadlock has occurred.

Recovery Regain the resources.

DEADLOCKS

Deadlock Prevention

Do not allow one of the four conditions to occur.

Mutual exclusion:

- a) Automatically holds for printers and other non-sharables.
- b) Shared entities (read only files) don't need mutual exclusion (and aren't susceptible to deadlock.)
- c) Prevention not possible, since some devices are intrinsically non-sharable.

Hold and wait:

- a) Collect all resources before execution.
- b) A particular resource can only be requested when no others are being held. A sequence of resources is always collected beginning with the same one.
- c) Utilization is low, starvation possible.

DEADLOCKS

Deadlock Prevention

Do not allow one of the four conditions to occur.

No preemption:

- a) Release any resource already being held if the process can't get an additional resource.
- b) Allow preemption - if a needed resource is held by another process, which is also waiting on some resource, steal it. Otherwise wait.

Circular wait:

- a) Number resources and only request in ascending order.
- b) EACH of these prevention techniques may cause a decrease in utilization and/or resources. For this reason, prevention isn't necessarily the best technique.
- c) Prevention is generally the easiest to implement.

DEADLOCKS

Deadlock Avoidance

If we have prior knowledge of how resources will be requested, it's possible to determine if we are entering an "unsafe" state.

Possible states are:

Deadlock No forward progress can be made.

Unsafe state A state that **may** allow deadlock.

Safe state A state is safe if a sequence of processes exist such that there are enough resources for the first to finish, and as each finishes and releases its resources there are enough for the next to finish.

The rule is simple: If a request allocation would cause an unsafe state, do not honor that request.

NOTE: All deadlocks are unsafe, but all unsafes are NOT deadlocks.

Banker's Algorithm: Introduction

- ❖ Banker's algorithm is a deadlock avoidance algorithm.
- ❖ It is named so because this algorithm is used in banking systems to determine whether a loan can be granted or not.
- ❖ Consider there are n account holders in a bank and the sum of the money in all of their accounts is S .
 - Every time a loan has to be granted by the bank, it subtracts the **loan amount** from the **total money** the bank has.
 - ✓ Then it checks if that difference is greater than S .
 - It is done because, only then, the bank would have enough money even if all the n account holders draw all their money at once.
- ❖ Banker's algorithm works in a similar way in computers.
- ❖ Whenever a new process is created, it must specify the maximum instances of each resource type that it needs, exactly.

Data Structures used to implement the Banker's Algorithm

🔗 Let us assume that there are n processes and m resource types.

1. Available

- It is an array of length m .
- It represents the number of available resources of each type.
- If $Available[j] = k$, then there are k instances available, of resource type R_j .

2. Max

- It is an $n \times m$ matrix which represents the maximum number of instances of each resource that a process can request.
- If $Max[i][j] = k$, then the process P_i can request at most k instances of resource type R_j .

Data Structures used to implement the Banker's Algorithm

③ 3. Allocation



- It is an $n \times m$ matrix which represents the number of resources of each type currently allocated to each process.
- If $Allocation[i][j] = k$, then process P_i is currently allocated k instances of resource type R_j

4. Need

- It is a two-dimensional array.
- It is an $n \times m$ matrix which indicates the remaining resource needs of each process.
- If $Need[i][j] = k$, then process P_i may need k more instances of resource type R_j to complete its task.

Banker's Algorithm Requirements



❖ When working with a banker's algorithm, it requests to know about three things:

- How much each process can request for each resource in the system. It is denoted by the **[MAX]** request.
- How much each process is currently holding each resource in a system. It is denoted by the **[ALLOCATED]** resource.
- It represents the number of each resource currently available in the system. It is denoted by the **[AVAILABLE]** resource.

Numerical Example: Banker's Algorithm

- ❖ Consider a system that contains five processes P1, P2, P3, P4, P5 and the three resource types A, B and C. Following are the resources types: A has 10, B has 5 and the resource type C has 7 instances.
- ❖ Answer the following questions using the banker's algorithm:
- What is the reference of the need matrix?
 - Determine if the system is safe or not.
 - What will happen if the resource request (1, 0, 2) for process P1 can the system accept this request immediately?

Process	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P1	0	1	0	7	5	3	3	3	2
P2	2	0	0	3	2	2			
P3	3	0	2	9	0	2			
P4	2	1	1	2	2	2			
P5	0	0	2	4	3	3			

Example

Process	Allocation			Max			Available			Need		
	A	B	C	A	B	C	A	B	C	A	B	C
P1	0	1	0	7	5	3	3	3	2	7	4	3
P2	2	0	0	3	2	2				1	2	2
P3	3	0	2	9	0	2				6	0	0
P4	2	1	1	2	2	2				0	1	1
P5	0	0	2	4	3	3				4	3	1

Need [i] = Max [i] - Allocation [i]

Need for P3: $(9, 0, 2) - (3, 0, 2) = 6, 0, 0$

Need for P1: $(7, 5, 3) - (0, 1, 0) = 7, 4, 3$

Need for P4: $(2, 2, 2) - (2, 1, 1) = 0, 1, 1$

Need for P2: $(3, 2, 2) - (2, 0, 0) = 1, 2, 2$

Need for P5: $(4, 3, 3) - (0, 0, 2) = 4, 3, 1$

Example

Process	Allocation			Max			Available			Need		
	A	B	C	A	B	C	A	B	C	A	B	C
P1	0	1	0	7	5	3	3	3	2	7	4	3
P2	2	0	0	3	2	2				1	2	2
P3	3	0	2	9	0	2				6	0	0
P4	2	1	1	2	2	2				0	1	1
P5	0	0	2	4	3	3				4	3	1

Step 1: For Process P1:

$\text{Need} \leq \text{Available}$

$7, 4, 3 \leq 3, 3, 2$ condition is **False**.

So, we examine another process, P2.

Example

Process	Allocation			Max			Available			Need		
	A	B	C	A	B	C	A	B	C	A	B	C
✗ P1	0	1	0	7	5	3	3	3	2	7	4	3
✓ P2	2	0	0	3	2	2	5	3	2	1	2	2
P3	3	0	2	9	0	2				6	0	0
P4	2	1	1	2	2	2				0	1	1
P5	0	0	2	4	3	3				4	3	1

Step 2: For Process P2:
 Need \leq Available
 1, 2, 2 \leq 3, 3, 2 condition **True**
 New Available = Available + Allocation
 (3, 3, 2) + (2, 0, 0) = 5, 3, 2
 Similarly, we examine another process P3.

Example



Process	Allocation			Max			Available			Need		
	A	B	C	A	B	C	A	B	C	A	B	C
P1	0	1	0	7	5	3	3	3	2	7	4	3
P2	2	0	0	3	2	2	5	3	2	1	2	2
P3	3	0	2	9	0	2				6	0	0
P4	2	1	1	2	2	2				0	1	1
P5	0	0	2	4	3	3				4	3	1

Step 3: For Process P3:

P3 Need \leq Available

6, 0, 0 \leq 5, 3, 2 condition is **False**.

Similarly, we examine another process, P4.

Example

Process	Allocation			Max			Available			Need		
	A	B	C	A	B	C	A	B	C	A	B	C
P1	0	1	0	7	5	3	3	3	2	7	4	3
✓ P2	2	0	0	3	2	2	5	3	2	1	2	2
P3	3	0	2	9	0	2	7	4	3	6	0	0
✓ P4	2	1	1	2	2	2				0	1	1
P5	0	0	2	4	3	3				4	3	1

Step 4:

For Process P4:

P4 Need \leq Available

0, 1, 1 \leq 5, 3, 2 condition is **True**

New Available = Available + Allocation

5, 3, 2 + 2, 1, 1 = 7, 4, 3

Similarly, we examine another process P5.

Example



Process	Allocation			Max			Available			Need		
	A	B	C	A	B	C	A	B	C	A	B	C
P1	0	1	0	7	5	3	3	3	2	7	4	3
P2	2	0	0	3	2	2	5	3	2	1	2	2
P3	3	0	2	9	0	2	7	4	3	6	0	0
P4	2	1	1	2	2	2	7	4	5	0	1	1
P5	0	0	2	4	3	3				4	3	1

Step 5:

For Process P5:

P5 Need \leq Available

4, 3, 1 \leq 7, 4, 3 condition is **True**

New Available = Available + Allocation

7, 4, 3 + 0, 0, 2 = 7, 4, 5

Now, we again examine each type of resource request for processes P1 and P3.

Example

Process	Allocation			Max			Available			Need		
	A	B	C	A	B	C	A	B	C	A	B	C
P1	0	1	0	7	5	3	3	3	2	7	4	3
P2	2	0	0	3	2	2	5	3	2	1	2	2
P3	3	0	2	9	0	2	7	4	3	6	0	0
P4	2	1	1	2	2	2	7	4	5	0	1	1
P5	0	0	2	4	3	3	7	5	5	4	3	1

Step 6:

For Process P1:

P1 Need \leq Available

7, 4, 3 \leq 7, 4, 5 condition is **True**

New Available = Available + Allocation

7, 4, 5 + 0, 1, 0 = 7, 5, 5

So, we examine another process ~~P2~~ P3

Example

Process	Allocation			Max			Available			Need		
	A	B	C	A	B	C	A	B	C	A	B	C
P1	0	1	0	7	5	3	3	3	2	7	4	3
P2	2	0	0	3	2	2	5	3	2	1	2	2
P3	3	0	2	9	0	2	7	4	3	6	0	0
P4	2	1	1	2	2	2	7	4	5	0	1	1
P5	0	0	2	4	3	3	7	5	5	4	3	1

Step 7:

For Process P3:

P3 Need \leq Available

6, 0, 0 \leq 7, 5, 5 condition is **True**

New Available = Available + Allocation

7, 5, 5 + 3, 0, 2 = 10, 5, 7

Hence, we execute the banker's algorithm to find the safe state and the safe sequence like P2, P4, P5, P1 and P3.

Answer-3

❖ What will happen if the resource request (1, 0, 2) for process P1 can the system accept this request immediately?

❖ Ans.:

For granting the Request (1, 0, 2), first we have to check that **Request** \leq **Available**, that is (1, 0, 2) \leq (3, 3, 2), since the condition is true. So the process P1 gets the request immediately.

Critical Section: Introduction

- ❖ When more than one processes access a same code segment that segment is known as **Critical Section**.
- ❖ Critical section contains shared variables or resources which are needed to be synchronized to maintain consistency of data variable.
- ❖ The critical section is a code segment where the shared variables can be accessed.
- ❖ Critical Section is the part of a program which tries to access shared resources. That resource may be any resource in a computer like a memory location, Data structure, CPU or any IO device.
- ❖ The critical section cannot be executed by more than one process at the same time; operating system faces the difficulties in allowing and disallowing the processes from entering the critical section.
- ❖ The critical section problem is used to design a set of protocols which can ensure that the Race condition among the processes will never arise.



Solution to the Critical Section Problem

❖ The critical section problem needs a solution to synchronize the different processes.

❖ The solution to the critical section problem must satisfy the following conditions:

1. Mutual Exclusion
 2. Progress
 3. Bounded Waiting
 4. Architectural Neutrality
-

1. Mutual Exclusion

- ❖  Mutual exclusion implies that only one process can be inside the critical section at any time.
- ❖  If any other processes require the critical section, they must wait until it is free.

2. Progress

- ❖ Progress means that if a process is not using the critical section, then it should not stop any other process from accessing it.
- ❖ In other words, any process can enter a critical section if it is free.

3. Bounded Waiting

- ❖ Bounded waiting means that each process must have a limited waiting time.
- ❖ It should not wait endlessly to access the critical section.

4. Architectural Neutrality

- ❖ Our mechanism must be architectural natural.
- ❖ It means that if our solution is working fine on one architecture then it should also run on the other ones as well.

Semaphore: Introduction

- ❖ Semaphore was proposed by Dijkstra in 1965 which is a very significant technique to manage concurrent processes by using a simple integer value, which is known as a semaphore.
- ❖ Semaphore is simply a variable that is non-negative and shared between threads.
- ❖ This variable is used to solve the critical section problem and to achieve process synchronization in the multiprocessing environment.
- ❖ A semaphore is a signaling mechanism, and a thread that is waiting on a semaphore can be signaled by another thread.
- ❖ It uses two atomic operations for the process synchronization
 1. Wait
 2. Signal

Wait and Signal Operations



❖ Wait

- The wait operation decrements the value of its argument S , if it is positive.
- If S is negative or zero, then no operation is performed.

```
wait(S)
{
    while(S <= 0) ;
    S--;
}
```

❖ Signal

- The signal operation increments the value of its argument S .

```
signal(S)
{
    S++;
}
```

Characteristic of Semaphore



- ❖ It is a mechanism that can be used to provide synchronization of tasks.
- ❖ It is a low-level synchronization mechanism.
- ❖ Semaphore will always hold a non-negative integer value.
- ❖ Semaphore can be implemented using test operations and interrupts, which should be executed using file descriptors.
- ❖ Semaphore cannot be implemented in the user mode because race condition may always arise when two or more processes try to access the variable simultaneously.
- ❖ It always needs support from the operating system to be implemented.

Types of Semaphore

❖ Counting Semaphores



- These are integer value semaphores and have an unrestricted value domain.
- These semaphores are used to coordinate the resource access, where the semaphore count is the number of available resources.
- If the resources are added, semaphore count automatically incremented and if the resources are removed, the count is decremented.

❖ Binary Semaphores

- The binary semaphores are like counting semaphores but their value is restricted to 0 and 1.
- The wait operation only works when the semaphore is 1 and the signal operation succeeds when semaphore is 0.
- It is sometimes easier to implement binary semaphores than counting semaphores.

Advantages of Semaphores



- ❖ Semaphores allow only one process into the critical section. They follow the mutual exclusion principle strictly and are much more efficient than some other methods of synchronization.
- ❖ There is no resource wastage because of busy waiting in semaphores as processor time is not wasted unnecessarily to check if a condition is fulfilled to allow a process to access the critical section.
- ❖ Semaphores are implemented in the machine independent code of the microkernel. So they are machine independent.

Disadvantages of Semaphores



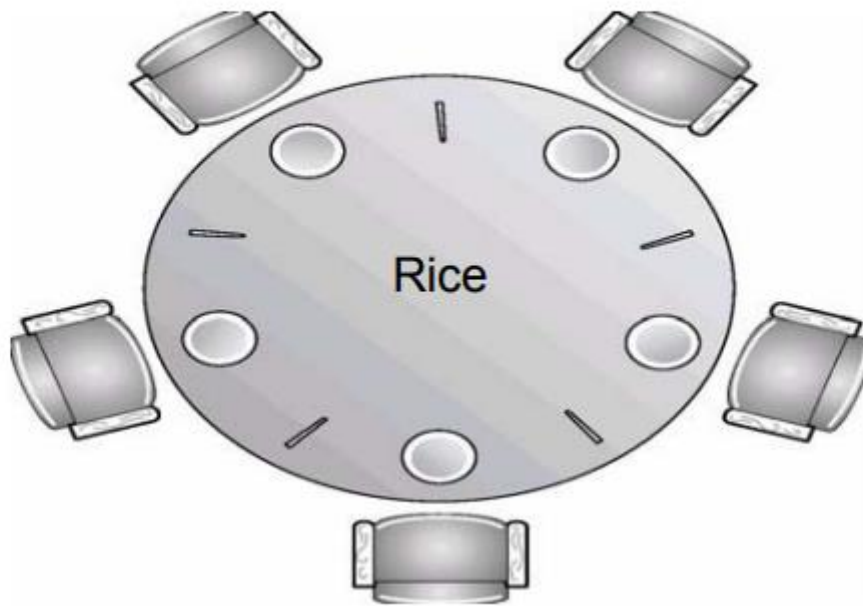
- ❖ Semaphores are complicated so the wait and signal operations must be implemented in the correct order to prevent deadlocks.
- ❖ Semaphores are impractical for last scale use as their use leads to loss of modularity. This happens because the wait and signal operations prevent the creation of a structured layout for the system.
- ❖ Semaphores may lead to a priority inversion where low priority processes may access the critical section first and high priority processes later.

Classical Problem Of Synchronization

Contents Of Discussion

- The Dining-Philosophers Problem
- The Reader Writers Problem
- The Bounded Buffer Problem

Dining-Philosophers Problem:



Dining-Philosophers Problem

- *Table, a bowl of rice in center, five chairs, and five chopsticks, also in each chair there is a philosopher.*
- *A philosopher may pick up only one chopstick at a time.*
- *When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks.*
- *When she is finished eating, she puts down both of her chopsticks and starts thinking.*
- *From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (chopsticks between her left and right neighbors).*

The Structure of philosopher i

```
do{
    wait(chopstick[i]);
    wait(chopstick[(i+1)%5]);
    ....
    //eat
    ....
    signal(chopstick[i]);
    signal(chopstick[(i+1)%5]);
    ....
    //think
    ....
}while(TRUE);
```

Dining-Philosophers Problem (Cont.)

Solution:

- Present each chopstick by a semaphore.
- A philosopher tries to grab the chopstick by executing a wait operation on that semaphore.
- She releases her chopsticks by executing the signal operation on the appropriate semaphores.
- This solution guarantees that no two neighbors are eating simultaneously, but it could cause a deadlock.
- Suppose all five philosophers become hungry simultaneously, and each grabs her left chopstick.
- When each philosopher tries to grab her right chopstick, she will cause the possibility that the philosophers will starve to death.

Dining-Philosophers Problem (Cont.)

- We present a solution to the dining-philosophers problem that ensure freedom from deadlock.
 - Allow at most 4 philosophers to be sitting simultaneously at the table.
 - Allow a philosopher to pick up her chopstick only if both chopsticks are available.
 - An odd philosopher picks up first her left chopstick and then her right chopstick, whereas, an even philosopher picks up her right chopstick first and then her left chopstick.
- **Note:** A deadlock-free solution does not necessarily eliminate the possibility of starvation.

A semaphore solution:

```
// represent each chopstick with a semaphore
```

```
Semaphore chopstick[] = new Semaphore[5]; // all = 1 initially
```

```
process philosopher_i {
```

```
    while (true) {
```

```
        // pick up left chopstick
```

```
        chopstick[i].acquire();
```

Cont...

```
// pick up right chopstick
    chopstick[(i+1) % 5].acquire();

// eat

// put down left chopstick
    chopstick[i].release();

// put down right chopstick
    chopstick[(i+1) % 5].release();

// think
```

Cont...

This solution guarantees no two neighboring philosophers eat simultaneously, but has the possibility of creating a deadlock

Uses of semaphores

- protecting access to a critical section (e.g., db in the R/W problem)
- as a mutex lock (e.g., to protect the shared variables used in solving a problem such as readerCount above)
- to protect a relationship (e.g., empty and full as used in the P/C problem)
- to support atomicity (e.g., you pick up either both chopsticks at the same time or neither, you cannot pick up just one)

Application

- *The dining philosophers problem represents a situation that can occur in large communities of processes that share a sizeable pool of resources.*

Bounded Buffer Problem

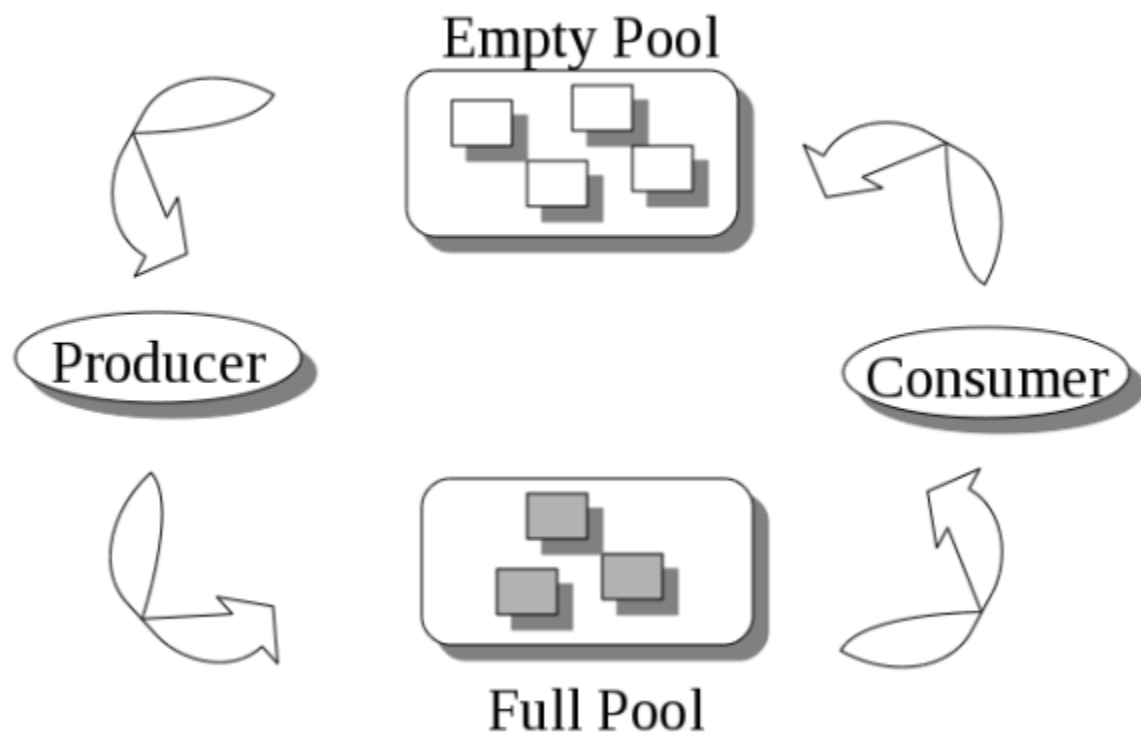
- *also called the Producers and Consumers problem.*
- *A finite supply of containers is available. Producers take an empty container and fill it with a product. Consumers take a full container, consume the product and leave an empty container.*
- *The main complexity of this problem is that we must maintain the count for both the number of empty and full containers that are available.*
- *Producers produce a product and consumers consume the product, but both use of one of the containers each time.*

The Bounded Buffer Problem

Consider:

- a buffer which can store n items
- a producer process which creates the items (1 at a time)
- a consumer process which processes them (1 at a time)
- A producer cannot produce unless there is an empty buffer slot to fill.
- A consumer cannot consume unless there is at least one produced item.

Cont..



Bounded-Buffer Problem Process

Shared data

semaphore full, empty, mutex;

Initialisation:

full = 0, empty = n, mutex = 1

Producer Process

```
do {  
  ...  
  produce an item in nextp  
  ...  
  wait(empty);  
  wait(mutex);...  
  add nextp to buffer...  
  signal(mutex);  
  signal(full);  
} while (1);
```

Consumer Process

```
do {  
  wait(full)  
  wait(mutex);  
  ...  
  remove an item from buffer to nextc  
  ...  
  signal(mutex);  
  signal(empty);  
  ...  
  consume the item in nextc  
  ...  
} while (1);
```


Application

- *A pipe or other finite queue (buffer), is an example of the bounded buffer problem.*

Reader Writer Problem

- *If one notebook exists where writers may write information to, only one writer may write at a time. Confusion may arise if a reader is trying read at the same as a writer is writing. Since readers only look at the data, but do not modify the data, we can allow more than one reader to read at the same time.*
- *The main complexity with this problems stems from allowing more than one reader to access the data at the same time.*

Reader Writer Problem



The Readers-Writers Problem

- A data item such as a file is shared among several processes.
- Each process is classified as either a reader or writer.
- Multiple readers may access the file simultaneously.
- A writer must have exclusive access (i.e., cannot share with either a reader or another writer).
- A solution gives priority to either readers or writers.

Cont....

- readers' priority: no reader is kept waiting unless a writer has already obtained permission to access the database
- writers' priority: if a writer is waiting to access the database, no new readers can start reading
- A solution to either version may cause starvation
- in the readers' priority version, writers may starve
- in the writers' priority version, readers may starve
- A semaphore solution to the readers' priority version (without addressing starvation):

Reader Writer Problem

Shared data

semaphore mutex, wrt;

Initially

mutex = 1, wrt = 1, readcount = 0

wait(wrt);

...

writing is performed

...

signal(wrt);

Readers-Writers Problem

Reader Process

```
wait(mutex);  
readcount++;  
if (readcount == 1)  
    wait(rt);  
signal(mutex);  
...  
reading is performed  
...  
wait(mutex);  
readcount--;  
if (readcount == 0)  
    signal(wrt);  
signal(mutex);
```

A semaphore solution to the readers priority version :

```
Semaphore mutex = 1;  
Semaphore db = 1;  
int readerCount = 0;
```

Cont....

```
process writer {
```

```
    db.acquire();
```

```
    // write
```

```
    db.release();
```

```
}
```

Conti...

```
process reader {  
    // protecting readerCount  
    mutex.acquire();  
    ++readerCount;  
    if (readerCount == 1)  
        db.acquire();  
    mutex.release();  
    // read  
    // protecting readerCount  
    mutex.acquire();  
    --readerCount;  
    if (readerCount == 0)  
        db.release;  
    mutex.release();  
}
```

Application

A message distribution system is an example of Readers and Writers. We must keep track of how many messages are in the queue

