Unit 3

Process Synchronization and Deadlocks

Concurrency

Concurrency

- Concurrency refers to the execution of multiple instructions at the same time.
- The threads can interact with one another via shared memory or message passing.
- Concurrency results in resource sharing.

Due to **Resource Sharing** between the processes there comes a need for the processes to **communicate** with each other.

Inter-Process Communication

Inter-process communication (IPC)

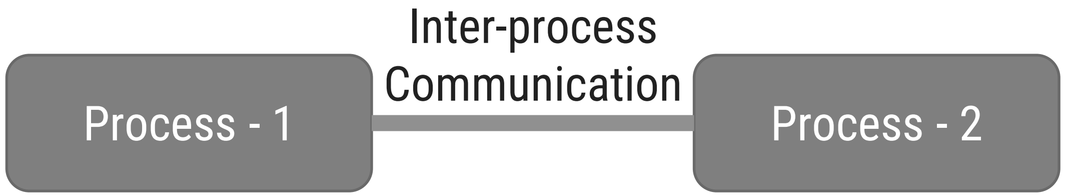
Inter-process communication is the **mechanism provided by the operating system that allows processes to communicate with each other**.

This communication could involve a process letting another process know that some event has occurred or transferring of data from one process to another.

Processes in a system can be independent or cooperating.

- Independent process cannot affect or be affected by the execution of another process.
- Cooperating process can affect or be affected by the execution of another process.

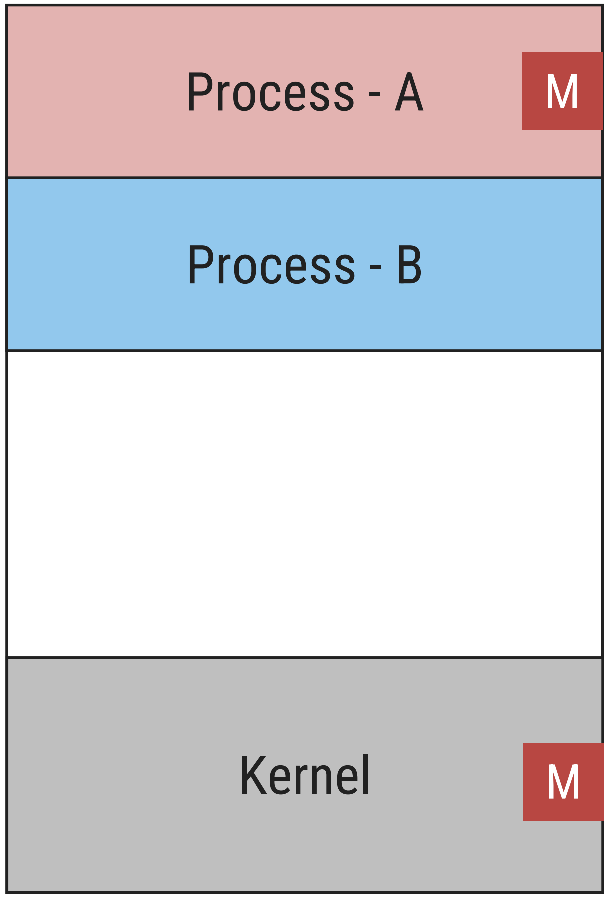
Cooperating processes need inter process communication mechanisms.



Models for Inter-process communication (IPC)

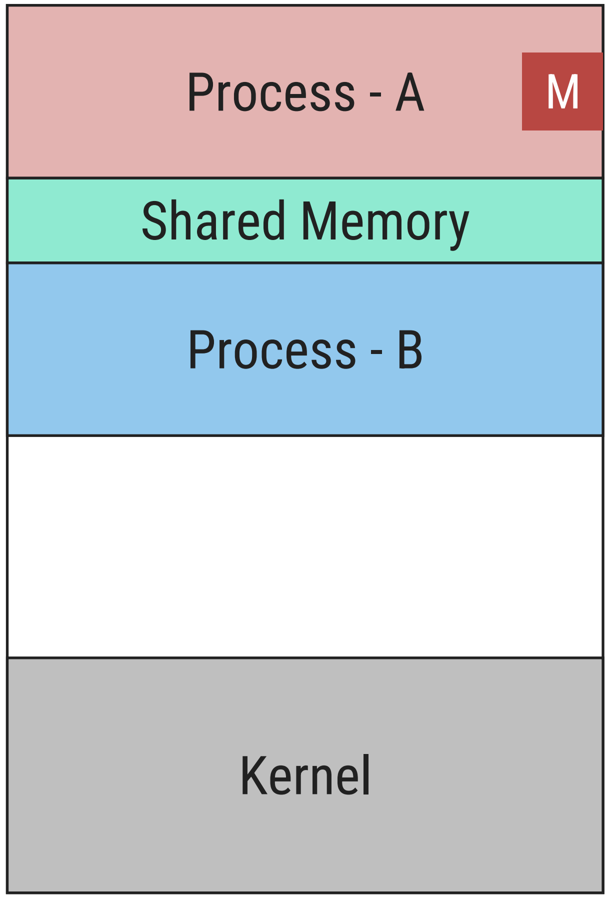
Message Passing

- Process A send the message to Kernel and then Kernel send that message to Process B



Shared Memory

- Process A put the message into Shared Memory and then Process B read that message from Shared Memory



Inter-process communication (IPC)

Reasons of process cooperation

- **Information sharing**: Several processes may need to access the same data (such as stored in a file.)
- **Computation speed-up**: A task can often be run faster if it is broken into subtasks and distributed among different processes.
- **Modularity**: It may be easier to organize a complex task into separate subtasks, then have different processes or threads running each subtask.
- **Convenience**: An individual user can run several programs at the same time, to perform some task.

Issues of process cooperation

- Data corruption, deadlocks, increased complexity
- Requires processes to synchronize their processing

Race Condition

The situation where **several processes access and manipulate shared data concurrently**. The **final value of the shared data depends upon which process finishes last**.

A race condition is an **undesirable situation** that **occurs when a device or system attempts to perform two or more operations at the same time**.

But, because of the nature of the device or system, the **operations must be done in the proper sequence** to be done correctly.

To prevent race conditions, **concurrent processes must be synchronized**.

Critical Section

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 problem is **to design a protocol that the processes can use to cooperate.**

Solving Critical-Section Problem

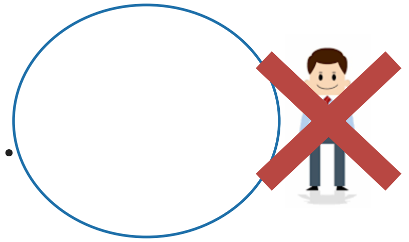
Any good solution to the problem must satisfy following three conditions:

Mutual Exclusion

- No two processes may be simultaneously inside the same critical section.

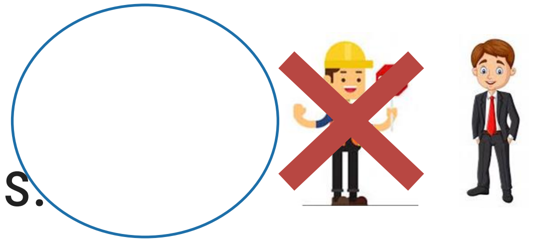
Bounded Waiting

- No process should have to wait forever to enter a critical section.



Progress

- No process running outside its critical region may block other processes.



Mutual Exclusion

Mutual Exclusion

Mutual Exclusion: Way of making sure that if one process is using a shared variable or file; the other process will be excluded (stopped) from doing the same thing.

Requirements of Mutual Exclusion

Any facility or capability that is to provide support for mutual exclusion should meet the following requirements:

1. Mutual exclusion must be enforced: Only one process at a time is allowed into its critical section, among all processes that have critical sections for the same resource or shared object.
2. A process that halts in its noncritical section must do so without interfering with other processes.
3. It must not be possible for a process requiring access to a critical section to be delayed indefinitely: no deadlock or starvation.
4. When no process is in a critical section, any process that requests entry to its critical section must be permitted to enter without delay.
5. No assumptions are made about relative process speeds or number of processors.
6. A process remains inside its critical section for a finite time only.

Mutual Exclusion :Hardware Support

1. Disabling Interrupts

- A process will continue to run until it invokes an OS service or until it is interrupted. Therefore, to guarantee mutual exclusion, it is sufficient to prevent a process from being interrupted.
- Because the critical section cannot be interrupted, mutual exclusion is guaranteed.

Mutual Exclusion :Hardware Support

2. **Test and Set the Lock(TSL)**

- TS instruction performs two actions: First, it tests the value of a memory byte (for lock variable) to check whether it is zero or one.
- If it is zero, then it performs the second operation, that is, sets the byte to one.
- The initial value of a lock is zero. Now, to make these two operations indivisible, the processor executing these instructions locks the memory bus so that no other process may request to access the memory at this time.
- In this way, the indivisible operations are implemented using special machine instructions
- Compare-and-swap and Exchange are some other examples of hardware-supported indivisible instructions that have been implemented in systems.

Semaphore

Section - 4

Semaphore

- A semaphore is a **variable that provides an abstraction for controlling the access of a shared resource** by multiple processes in a parallel programming environment.
- There are 2 types of semaphores:
  - **Binary semaphores** :-
    - Binary semaphores can take only 2 values (0/1).
    - Binary semaphores have 2 methods associated with it (up, down / lock, unlock).
    - They are used to acquire locks.
  - **Counting semaphores** :-
    - Counting semaphore can have possible values more than two.

Operations on Semaphore

Wait(): a process performs a wait operation to **tell the semaphore that it wants exclusive access to the shared resource**.

- If the value of semaphore is one, then the semaphore value is changed to zero and allows the process to continue its execution immediately.

Signal(): a process performs a signal operation to **inform the semaphore that it is finished using the shared resource**.

- If there are processes suspended on the semaphore, the semaphore value is changed to one.

The Producer Consumer Problem

The Producer Consumer Problem

It is **multi-process synchronization** problem.

It is also known as **bounded buffer problem**.

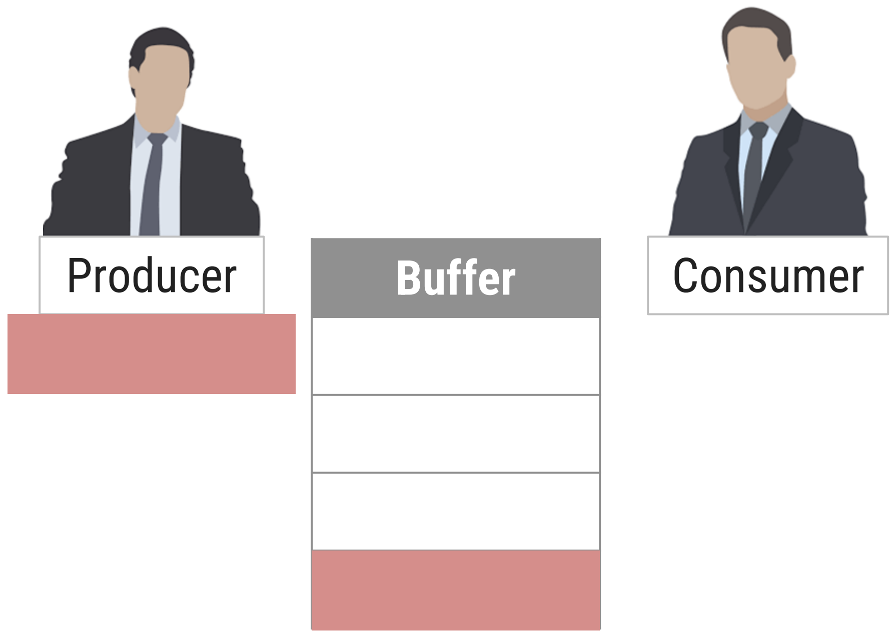
This problem describes two processes producer and consumer, who share common, fixed size buffer.

Producer process

- Produce some information and put it into buffer

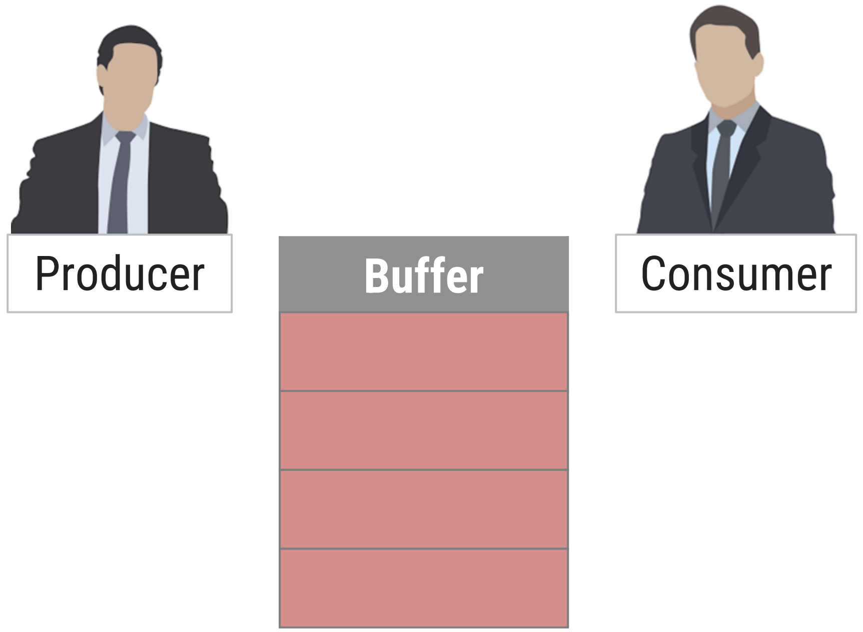
Consumer process

- Consume this information (remove it from the buffer)



What Producer Consumer problem is?

The Problem: ensure that the producer can’t add data into full buffer and consumer can’t remove data from an empty buffer



Producer Consumer problem Solution

- **The solution for the producer is to go to sleep if the buffer is full.** The next time the consumer removes an item from the buffer, it wakes up the producer who starts to fill the buffer again.

- **In the same way, the consumer goes to sleep if it finds the buffer to be empty**. The next time the producer puts data into the buffer, it wakes up the sleeping consumer. The solution can be reached by means of inter-process communication, typically using semaphores. An inadequate solution could result in a deadlock where both processes are waiting to be awakened.

Producer Consumer problem’s Solution with Semaphore

The producer process after producing the item waits on the **empty semaphore** because it needs to check whether there is an empty slot in the buffer.

If there is a slot, the semaphore allows it to go further.

Once it passes this condition, it waits on the **Buffer_access semaphore** to check whether any other process is accessing it.

If any other process is already accessing it, then the semaphore will not allow it.

After getting the permission, the process starts accessing and storing the item in the buffer.

After storing**, it signals the Buffer_access so that any other process in wait** can access the buffer.

Moreover, it also signals the full semaphore to indicate that the buffer now contains one item.

In this way, the producer stores the item in the buffer and also updates the status of the buffer. Similarly, the algorithm of the consumer can be understood.

Deadlock

Deadlock

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

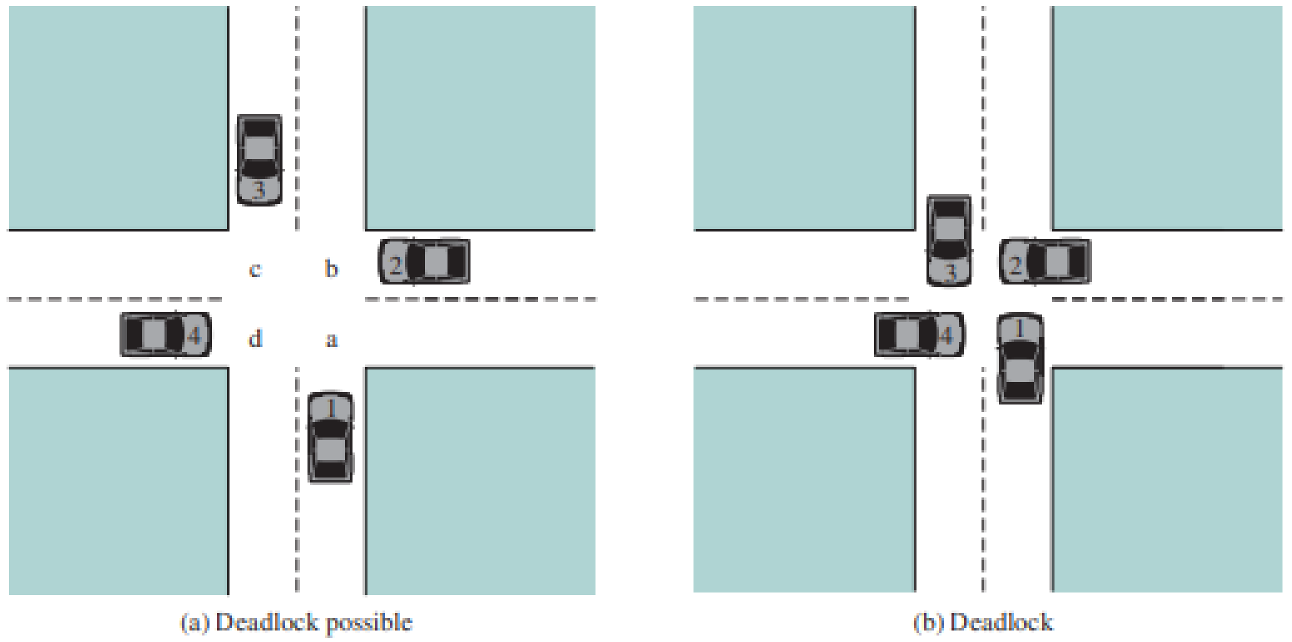


Figure 6.1 Illustration of Deadlock

Resources

Consumable Resources

- A consumable resource is one that can be created (produced) and destroyed (consumed). Typically, there is no limit on the number of consumable resources of a particular type.
- An unblocked producing process may create any number of such resources.
- When a resource is acquired by a consuming process, the resource ceases to exist.
- Examples of consumable resources are interrupts, signals, messages, and information in I/O buffers.

Reusable Resources

- A reusable resource is one that can be safely used by only one process at a time and is not depleted by that use.
- Processes obtain resource units that they later release for reuse by other processes.
- Examples of reusable resources include processors; I/O channels; main and secondary memory; devices; and data structures such as files, databases, and semaphores.

Conditions for Deadlock

Four conditions must be present for a deadlock to be possible:

- **Mutual exclusion**- Only one process may use a resource at a time. No process may access a resource unit that has been allocated to another process.
- **Hold and wait**- A process may hold allocated resources while awaiting assignment of other resources.
- **No preemption**- No resource can be forcibly removed from a process holding it.
- **Circular wait**- A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain

Resource Allocation Graphs

A useful tool in characterizing the allocation of resources to processes is the resource allocation graph, introduced.

The resource allocation graph is a directed graph that depicts a state of the system of resources and processes, with each process and each resource represented by a node.

A graph edge directed from a process to a resource indicates a resource that has been requested by the process but not yet granted.

Within a resource node, a dot is shown for each instance of that resource.

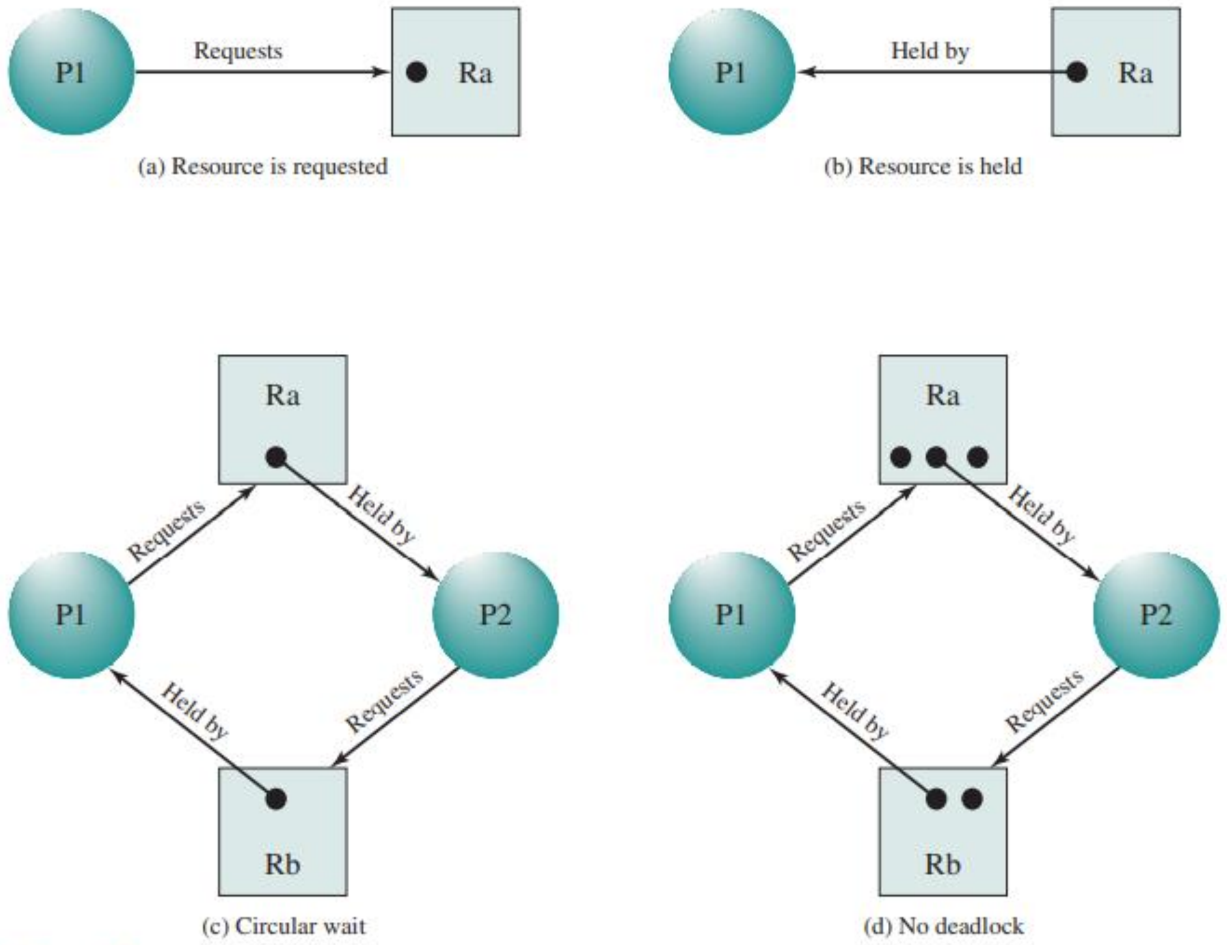


Figure 6.5 Examples of Resource Allocation Graphs

Deadlock Prevention

If at least one of these conditions cannot hold, we can prevent the occurrence of a deadlock.

- **Mutual exclusion**- To prevent deadlock, mutual exclusion of shareable resources must be prevented.
- **Hold and wait**- To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it does not hold any other resources.
  - Each process to request and be allocated all its resources before it begins execution.
  - Allows a process to request resources only when it has none. A process may request some resources and use them. Before it can request any additional resources, it must release all the resources that it is currently allocated.

Deadlock Prevention

No preemption- To ensure that **no preemption** does not hold, we can use the following protocol.

1. If a process is holding some resources and requests another resource that cannot be immediately allocated to it, then all resources the process is currently holding are preempted (released).

2. If a process 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 preempt the desired resources from the waiting process and allocate them to the requesting process.

3. If the resources are neither available nor held by a waiting process, the requesting process must wait. While it is waiting, some of its resources may be preempted, 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 preempted while it was waiting.

Deadlock Prevention

Circular Wait - The circular wait can be prevented independently through resource-ranking or ordering.

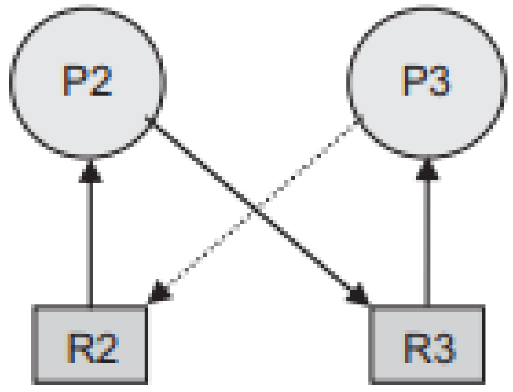
- Resource ordering is a method, in which every resource type in the system is given a unique integer number as an identification.
- Whenever a process requests a resource, it is checked whether it holds any other resource. If it does, IDs of both resource types are compared.
- If the ID of a requested resource is greater than the ID of the resource it holds, **the request is valid.**
- Otherwise, it is rejected.
- This protocol implies that if a process wants to request a resource type with lower ID as compared to the ID of resource type it holds, it must release all its resources.
- This specific ordering of requests will not allow the circular wait condition to arise.

Deadlock Avoidance

Deadlock can be avoided using two algorithms-

1. Resource Allocation Graph Algorithm

- To avoid a deadlock in a system, where every resource type has a single instance of resource, the RAG can be used again, but along with a new edge, known as claim edge.
- The claim edge is the same as request edge drawn from a process to a resource instance, but this does not mean that the request has been incorporated in the system. It is drawn in dotted lines.
- The RAG can be used in such a way that when a process requests for a resource, a corresponding claim edge is drawn, and the graph is checked before converting it to a request edge.
- That is, a process request will not be entertained until the cycle check has been done. After the cycle check, if it is confirmed that there will be no circular wait, the claim edge is converted to a request edge.
- Otherwise, it will be rejected. In this way, the deadlock is avoided.



Deadlock Avoidance

2. Banker's Algorithm-

Let *n* = number of processes, and *m* = number of resources types.

Available: Vector of length *m*. If available [*j*] = *k*, there are *k* instances of resource type R_j available.

Max*: n x m* matrix. If *Max* [*i,j*] = *k*, then process P_i may request at most *k* instances of resource type R_j.

Allocation: *n* x *m* matrix. If Allocation[*i,j*] = *k* then P_i is currently allocated *k* instances of $R_{j.}$

Need: *n* x *m* matrix. If *Need*[*i,j*] = *k*, then P_i may need *k* more instances of R_j to complete its task.

***Need* [*i,j*] = *Max*[*i,j*] – *Allocation* [*i,j*].**

Deadlock Avoidance

a. Safety Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively.

   Initialize:

   Work = Available

   Finish [*i*] = *false* for *i* = 0, 1, …, *n*- 1.

2. Find an *i* such that both:

```
(a) Finish [i] = false          // process i not yet done
(b) $Need_i \leq$ Work           //its need can be satisfied
If no such i exists, go to step 4.
```

3. *Work* = *Work* + *Allocation*$_i$ *Finish*[*i*] = *true*   *// run it and reclaim*
   go to step 2.   *// process i completes*

4. If *Finish* [*i*] == true for all *i*, then the system is in a safe state.

Deadlock Avoidance

- **Example 1-** Consider a system with 12 magnetic tape drives. Snapshot at time T0 is:

Process	Maximum needs	Allocated
P0	10	5
P1	4	2
P2	9	2

Available
3

Find need matrix and safe sequence.

If Available vector is not given then, Available Resources= Total Resources - Allocated Resources

Deadlock Avoidance

Need Matrix

Process	Maximum needs	Allocated
P0	10	5
P1	4	2
P2	9	2

Need = Max- Allocation
5
2
7

Deadlock Avoidance

- Consider a system with 12 magnetic tape drives. Snapshot at time T0 is:

Process	Maximum needs	Allocated
P0	10	5
P1	4	2
P2	9	2

Available/ work
3

Need = Max-Allocation
5
2
7

- Safe sequence < P1 P0 P2>

1 P0 Need< = Work NO
2 P1 Need<= Work YES
Work = Work+ Allocation
= 3+2 =5
3.P2 Need <= Work NO
4. P0 Need<= Work YES
Work = Work + Allocation
= 5+5=10
5. P2 Need<= Work YES
Work = Work + Allocation
rating System =10+2=12

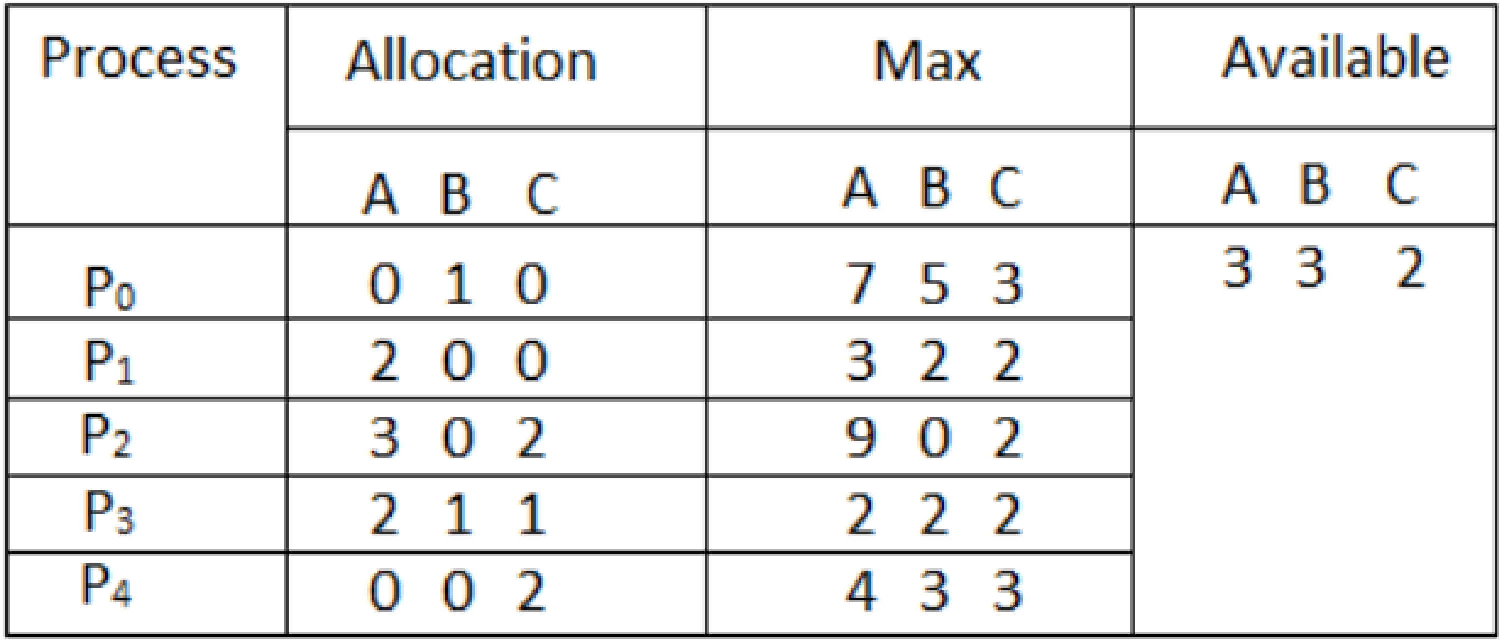
Deadlock Avoidance

Points to Remember-

- Available Resources = Total Resources - Allocated Resources
- Work = Available
- Need = Max – Allocated
- Check Need<=Work
- Work = Work + Allocated

Deadlock Avoidance

Example 2- Consider a system. Snapshot at time T0 is:

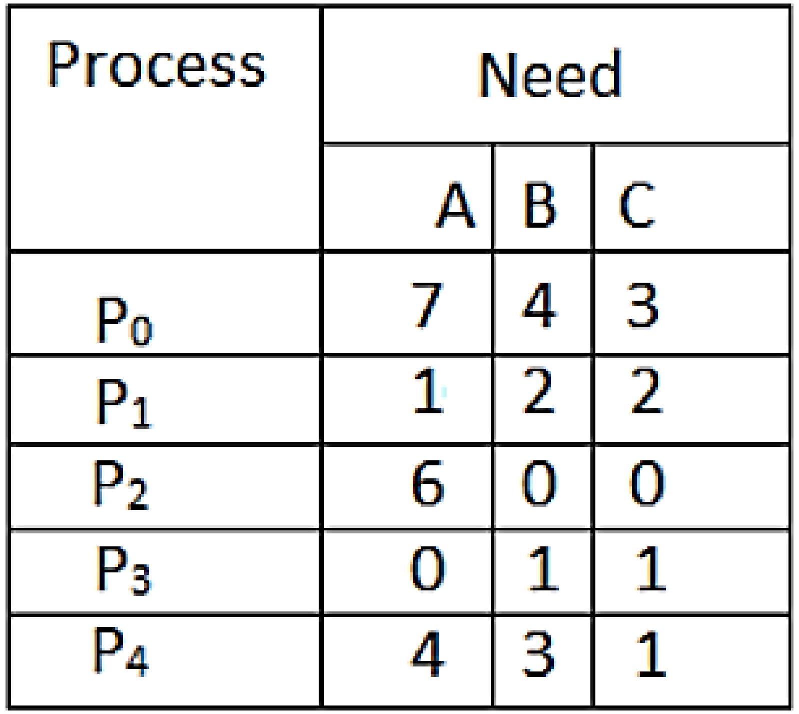


Process	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	3	3	2
P_1	2	0	0	3	2	2			
P_2	3	0	2	9	0	2			
P_3	2	1	1	2	2	2			
P_4	0	0	2	4	3	3			

Deadlock Avoidance

Need [i, j] = Max [i, j] – Allocation [i, j]

So, the content of Need Matrix is:



Process	Need		
	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

Step 1 of Safety Algo

Work = Available

Work =

3	3	2

Finish =

0	1	2	3	4
false	false	false	false	false

For i = 0 ✗ Step 2

$Need_0$ = 7, 4, 3

Finish [0] is false and $\overset{7,4,3}{Need_0} > \overset{3,3,2}{Work}$

So P_0 must wait

But Need ≤ Work

For i = 1 ✓ Step 2

$Need_1$ = 1, 2, 2

Finish [1] is false and $\overset{1,2,2}{Need_1} < \overset{3,3,2}{Work}$

So P_1 must be kept in safe sequence

Step 3

$Work = \overset{3,\,3,\,2}{Work} + \overset{2,\,0,\,0}{Allocation_1}$

Work =

A	B	C
5	3	2

Finish =

0	1	2	3	4
false	true	false	false	false

For i = 2 ✗ Step 2

$Need_2$ = 6 , 0, 0

Finish [2] is false and $\overset{6,\,0,\,0}{Need_2} > \overset{5,3,2}{Work}$

So P_2 must wait

For i=3 ✓ Step 2

$Need_3$ = 0, 1, 1

Finish [3] = false and $\overset{0,\,1,\,1}{Need_3} < \overset{5,\,3,\,2}{Work}$

So P_3 must be kept in safe sequence

Step 3

$Work = \overset{5,\,3,\,2}{Work} + \overset{2,\,1,\,1}{Allocation_3}$

Work =

A	B	C
7	4	3

Finish =

0	1	2	3	4
false	true	false	true	false

For i = 4 ✓ Step 2

$Need_4$= 4, 3, 1

Finish [4] = false and $\overset{4,\,3,\,1}{Need_4} < \overset{7,\,4,\,3}{Work}$

So P_4 must be kept in safe sequence

Step 3

$Work = \overset{7,\,4,\,3}{Work} + \overset{0,\,0,\,2}{Allocation_4}$

Work =

A	B	C
7	4	5

Finish =

0	1	2	3	4
false	true	false	true	true

For i = 0 ✓ Step 2

$Need_0$ = 7, 4, 3

Finish [0] is false and $\overset{7,\,4,\,3}{Need} < \overset{7,\,4,\,5}{Work}$

So P_0 must be kept in safe sequence

Step 3

$Work = \overset{7,\,4,\,5}{Work} + \overset{0,\,1,\,0}{Allocation_0}$

Work =

A	B	C
7	5	5

Finish =

0	1	2	3	4
true	true	false	true	true

For i = 2 ✓ Step 2

$Need_2$ = 6 , 0, 0

Finish [2] is false and $\overset{6,\,0,\,0}{Need_2} < \overset{7,\,5,\,5}{Work}$

So P_2 must be kept in safe sequence

Step 3

$Work = \overset{7,\,5,\,5}{Work} + \overset{3,\,0,\,2}{Allocation_2}$

Work =

A	B	C
10	5	7

Finish =

0	1	2	3	4
true	true	true	true	true

Step 4

Finish [i] = true for $0 \le i \le n$

Hence the system is in Safe state

The safe sequence is P_1, P_3 , P_4 , P_0, P_2

Deadlock Avoidance

Example 2

	MAX			ALLOCATION		
PROCESS	R1	R2	R3	R1	R2	R3
P1	5	6	3	2	1	0
P2	8	5	6	3	2	3
P3	4	8	2	3	0	2
P4	7	4	3	3	2	0
P5	4	3	3	1	0	1

Total Resources		
R1	R2	R3
15	8	8

Q1. Find out available resources
Q2. Calculate need matrix.
Q2. Determine whether system is in safe state or no?

Deadlock Avoidance

Example 2

- Available = total – allocation

  = 3 3 2

SAFE SEQUENCE:
< P5 P4 P1 P2 P3 >

Need matrix

R1	R2	R3
3	5	3
5	3	3
1	9	0
4	2	3
3	3	2

Deadlock Avoidance

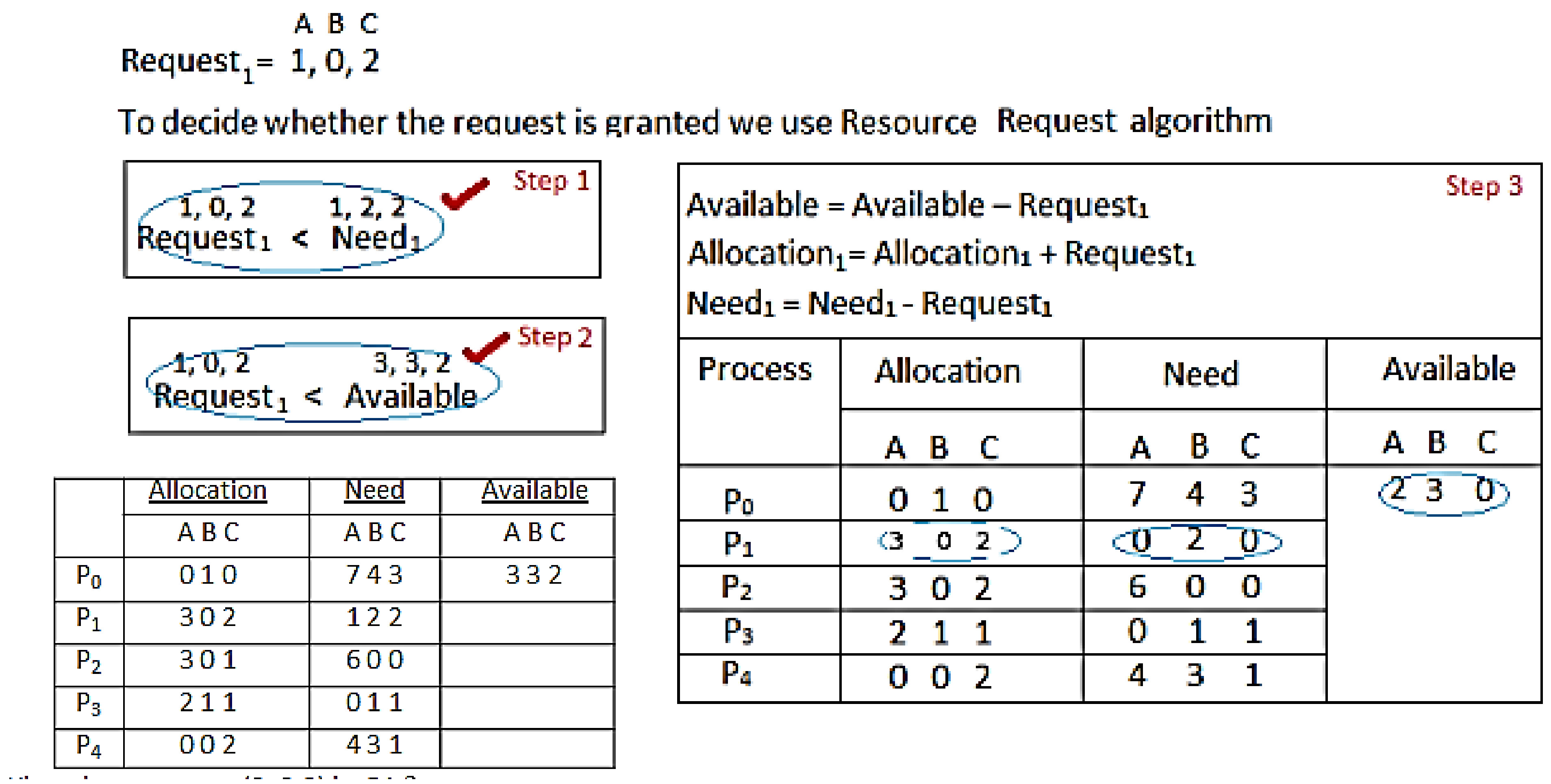
What will happen if process P_1 requests one additional instance of resource type A and two instances of resource type C?

b. Resource-Request Algorithm

Let Request_i be the request array for process P_i. Request_i [j] = k means process P_i wants k instances of resource type R_j. When a request for resources is made by process P_i, the following actions are taken:

```
1) If Request_i <= Need_i
Goto step (2) ; otherwise, raise an error condition, since the process has exceeded its
maximum claim.
2) If Request_i <= Available
Goto step (3); otherwise, P_i must wait, since the resources are not available.
3) Have the system pretend to have allocated the requested resources to process Pi by
modifying the state as
follows:
Available = Available – Requesti
Allocation_i = Allocation_i + Request_i
Need_i = Need_i– Request_i
```

Deadlock Avoidance



$$\text{Request}_1 = \begin{array}{ccc} A & B & C \\ 1, & 0, & 2 \end{array}$$

To decide whether the request is granted we use Resource Request algorithm

Step 1 ✓

$\overset{1,0,2}{\text{Request}_1} < \overset{1,2,2}{\text{Need}_1}$

Step 2 ✓

$\overset{1,0,2}{\text{Request}_1} < \overset{3,3,2}{\text{Available}}$

	Allocation	Need	Available
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	3 3 2
P_1	3 0 2	1 2 2	
P_2	3 0 1	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

Step 3

$\text{Available} = \text{Available} - \text{Request}_1$

$\text{Allocation}_1 = \text{Allocation}_1 + \text{Request}_1$

$\text{Need}_1 = \text{Need}_1 - \text{Request}_1$

Process	Allocation	Need	Available
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

We must determine whether this new system state is safe. To do so, we again execute Safety algorithm on the above data structures.

Thank You