CHAPTER 3

PROCESS COORDINATION

- ***Basic Concepts of Inter-process Communication and Synchronization***
- ***Race Condition***
- ***Critical Region and Problem***
- ***Peterson's Solution***
- ***Synchronization Hardware and Semaphores***
- ***Classic Problems of Synchronization***
- ***Message Passing***
- ***Introduction to Deadlocks***
- ***System Model***
- ***Deadlock Characterization***
- ***Deadlock Detection and Recovery***
- ***Deadlock Prevention***
- ***Deadlock Avoidance***

➢ Basic Concepts of Inter-process Communication and Synchronization

In modern computer systems, the concept of a process (a program in execution) is used to perform multiple tasks at the same time. And since multiple processes running on a single or multiple computers at the same time are connected to a network can exchange the data/information through a process called inter-process communication (IPC).

Operating systems provide communication between processes so that information can be exchanged between processes is called inter-process communication (IPC). It allows programmers to coordinate actions between multiple processes running on the system at the same time. There are two types of processes in the system independent process and the Cooperative process.

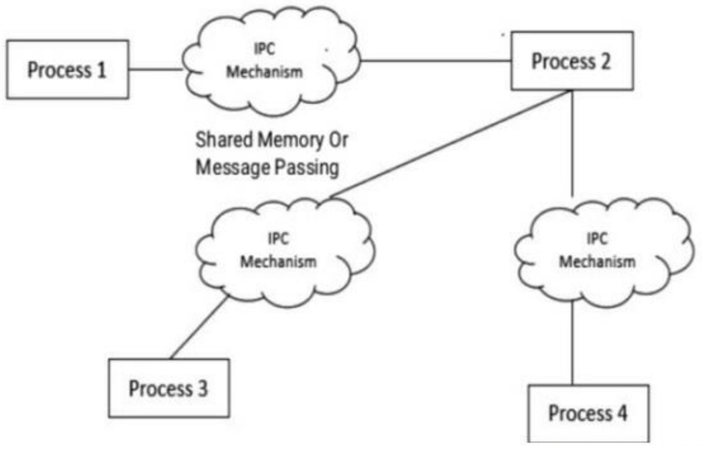


Fig.: Overview of Inter process Communication

From the above diagram, Inter-Process Communication (IPC) mechanism allows the data to be exchanged between processes. It uses either shared memory or a memory mapping mechanism. This allows resources and data to be shared between processes without interference. Processes running at the same time in the OS can be either co-operating processes or independent processes.

- **Independent process:** An independent process is a process that does not affect or is not affected by other processes running on the system. In simple words, any process that doesn't share data with other processes.
- **Cooperative process:** It is when it affects or can be affected by other processes running on the system. A process that communicates with other processes can exchange the data. Inter-process communication comes under cooperating process because it can provide information sharing, computational speed, modularity, and ease of data access.

The Different types of Inter Process Communication are:

1. **Shared Memory:** In this method, the multiple processes can communicate with each other simultaneously by accessing or sharing the memory. The multiple processes can exchange the data using read/write to the shared memory because the OS creates a common memory in RAM for sharing. It requires protection by the synchronization of access of all multiple processes in the communication.

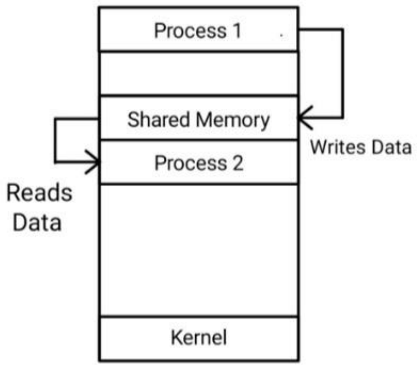


Fig.: Shared memory type in IPC

The information generated by process 1 about a particular resource and stored the record in shared memory. If process 2 wants to use this information, it checks the stored record in the shared memory, takes the information generated by process 1, and processes it accordingly. Thus, processes 1 and 2 can utilize shared memory to retrieve information such as records from other processes and transfer certain information to other processes.

2. **Message Passing:** In message passing, the communication between the processes is performed through the communication link. The operations performed by the processes are, sending the message and receiving the message. Since the message might be variable or fixed type.

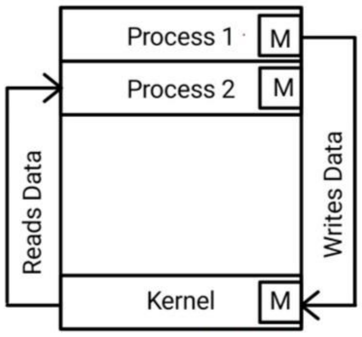


Fig.: Message Passing type in IPC

process 1 and process 2 communicated together by establishing the communication link as shown in the figure above. After that, they exchange the message through send and receive operations. The message is transferred in the First In First Out (FIFO) style. The standard message exchanged by the processes contains 2 parts. Such as

Header: It contains the type of message, source and destination identities, length of the message, and control information like priority, sequence number, and low disk space actions.

Body: It contains the original message that is to be exchanged.

The communication link can be established in different types. They are,

1. Direct communication.
2. Indirect communication.
3. Synchronous message passing.
4. Asynchronous message passing.
5. Buffering.

3. **Pipes:** It is a one-way communication or half-duplex communication mode in IPC. It is the most widely used IPC for communication between the two processes. If one process writes/sends the data, then the other process will receive or read it. Pipes are created using a pipe system called.

5. **Message Queues:** This type of IPC provides communication between single or multiple processes in full-duplex mode. It is a linked messages list stored in a kernel of an operating system. And the stored messages are recognized and identified by the message queue identifier.

   The messages are coordinated by the Application Program Interface(API). Several messages can read and write data to and from the message queue. A message queue is where messages are stored or queued until the recipient retrieves them. It is used in IPS and all operating systems.

6. **FIFO:** It is First-In-First-Out inter-process communication. It is a kind of mutual communication between two independent processes. It can also be called a full-duplex. That is, one process can interact with another process and vice versa. It is similar to the pipes.

8. **Direct Communication:** This type of IPC creates or establishes a connection or communication link between two communication processes. However, each pair of communication processes can have only one connection. It is a unidirectional and bidirectional communication process. The communication between the processes is either symmetric addressing type or asymmetric addressing type.

9. **Indirect Communication:** This type of IPC can exist or be created when processes share the same mailbox and each pair of those processes share multiple communication channels or links. These shared links can be one-way or two-way. When process 1 sends the data to the mailbox, then process 2 at the receiver retrieves the data from the mailbox. One to one, one to many, many to one and many to many communication links can be formed through a common mailbox.

➢ Race Condition

When more than one process is executing the same code or accessing the same memory or any shared variable in that condition there is a possibility that the output or the value of the shared variable is wrong so for that all the processes doing the race to say that my output is correct this condition known as a race condition.

Several processes access and process the manipulations over the same data concurrently, then the outcome depends on the particular order in which the access takes place. A race condition is a situation that may occur inside a critical section. This happens when the result of multiple thread execution in the critical section differs according to the order in which the threads execute.

Race conditions in critical sections can be avoided if the critical section is treated as an atomic instruction. Also, proper thread synchronization using locks or atomic variables can prevent race conditions.

➢ Critical Region and Problem

A critical section is a code segment that can be accessed by only one process at a time. The critical section contains shared variables that need to be synchronized to maintain the consistency of data variables. So the critical section problem means designing a way for cooperative processes to access shared resources without creating data inconsistencies.

```
do {
    entry section
        critical section
    exit section
        remainder section
  } while (TRUE);
```

Any solution to the critical section problem must satisfy three requirements:

1. **Mutual Exclusion**: If a process is executing in its critical section, then no other process is allowed to execute in the critical section.

2. **Progress**: If no process is executing in the critical section and other processes are waiting outside the critical section, then only those processes that are not executing in their remainder section can participate in deciding which will enter in the critical section next, and the selection can't be postponed indefinitely.

3. **Bounded Waiting**: A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

➢ PETERSON'S SOLUTION

Peterson's Solution is a classical software-based solution to the critical section problem. In Peterson's solution, we have two shared variables:

i. Boolean flag[i]: Initialized to FALSE, initially no one is interested in entering the critical section
ii. int turn: The process whose turn is to enter the critical section.

```
do {

    flag[i] = TRUE ;
    turn = j ;
    while (flag[j]  &&  turn == j) ;

        critial section

    flag[i] = FALSE ;

        remainder section

  }  while (TRUE) ;
```

Peterson's Solution preserves all three conditions:

- Mutual Exclusion is assured as only one process can access the critical section at any time.
- Progress is also assured, as a process outside the critical section does not block other processes from entering the critical section.
- Bounded Waiting is preserved as every process gets a fair chance.

Disadvantages of Peterson's solution:

- It involves busy waiting.(In the Peterson's solution, the code statement- "while(flag[j] && turn == j);" is responsible for this. Busy waiting is not favoured because it wastes CPU cycles that could be used to perform other tasks.)
- It is limited to 2 processes.
- Peterson's solution cannot be used in modern CPU architectures.

➢ Synchronization Hardware

Hardware Locks are used to solve the problem of `process synchronization. The process synchronization problem occurs when more than one process tries to access the same resource or variable. If more than one process tries to update a variable at the same time then a data inconsistency problem can occur. This process synchronization is also called synchronization hardware in the operating system.

There are 3 algorithms to for hardware synchronization:

1. **Test and set:**

   In the above algorithm the TestAndSet() function takes a Boolean value and returns the same value. TestAndSet() function sets the lock variable to true.

   In test and set algorithm the incoming process trying to enter the critical section does not wait in a queue so any process may get the chance to enter the critical section as soon as the process finds the lock variable to be false. It may be possible that a particular process never gets the chance to enter the critical section and that process waits indefinitely.

2. **Swap:**

   Swap function uses two Boolean variables lock and key. Both lock and key variables are initially initialized to the false. Swap algorithm is the same as lock and set algorithm. The Swap algorithm uses a temporary variable to set the lock to true when a process enters the critical section of the program.

3. **Unlock and Lock**

   Unlock and lock algorithm uses the TestAndSet method to control the value of lock. Unlock and lock algorithm uses a variable waiting[i] for each process. All the processes are maintained in a ready queue before entering into the critical section. The processes are added to the queue with respect to their process number. The queue is the circular queue.

   In Unlock and lock algorithm the lock is not set to false as one process comes out of the critical section. In other algorithms like swap and Test and set the lock was being set to false as the process comes out of the critical section so that any other process can enter the critical section.

➢ SEMAPHORES

Semaphores are a synchronization mechanism used to coordinate the activities of multiple processes in a computer system. They are used to enforce mutual exclusion, avoid race conditions and implement synchronization between processes.

Semaphores provide two operations: wait (P) and signal (V). The wait operation decrements the value of the semaphore, and the signal operation increments the value of the semaphore. When the value of the semaphore is zero, any process that performs a wait operation will be blocked until another process performs a signal operation.

Semaphores are used to implement critical sections, which are regions of code that must be executed by only one process at a time. By using semaphores, processes can coordinate access to shared resources, such as shared memory or I/O devices.

Types of Semaphores: There are two main types of semaphores

6. **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.

8. **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.
- 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.
- Semaphores may lead to a priority inversion where low priority processes may access the critical section first and high priority processes later.

➢ Classic Problems of Synchronization

There are number of classical problems of synchronization as examples of a large class of concurrency-control problems. The solutions to the problems, we use semaphores for synchronization, since that is the traditional way to present such solutions. However, actual implementations of these solutions could use mutex locks in place of binary semaphores.

These problems are used for testing nearly every newly proposed synchronization scheme. The following problems of synchronization are considered as classical problems:

1. **Bounded-buffer (or Producer-Consumer) Problem:**
   Bounded Buffer problem is also called producer consumer problem. This problem is generalized in terms of the Producer-Consumer problem. Solution to this problem is, creating two counting semaphores "full" and "empty" to keep track of the current number of full and empty buffers respectively. Producers produce a product and consumers consume the product, but both use of one of the containers each time.

2. **Dining-Philosophers Problem:**
   The Dining Philosopher Problem states that K philosophers seated around a circular table with one chopstick between each pair of philosophers. There is one chopstick between each philosopher. A philosopher may eat if he can pickup the two chopsticks adjacent to him. One chopstick may be picked up by any one of its adjacent followers but not both. This problem involves the allocation of limited resources to a group of processes in a deadlock-free and starvation-free manner.

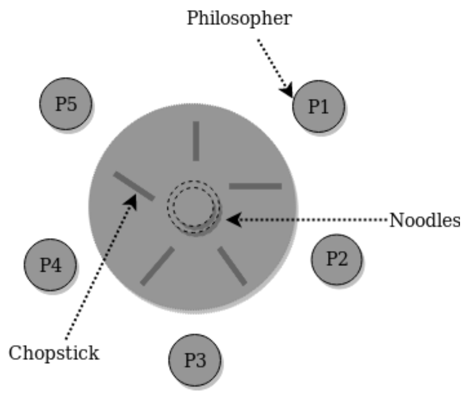


Fig.: Overview of Dining Philosophers Problem

3. **Readers and Writers Problem:**
   Suppose that a database is to be shared among several concurrent processes. Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database. We distinguish between these two types of processes by referring to the former as readers and to the latter as writers.

Precisely in OS we call this situation as the readers-writers problem. Problem parameters:

- One set of data is shared among a number of processes.
- Once a writer is ready, it performs its write. Only one writer may write at a time.
- If a process is writing, no other process can read it.
- If at least one reader is reading, no other process can write.
- Readers may not write and only read.

4. **Sleeping Barber Problem:**

Barber shop with one barber, one barber chair and N chairs to wait in. When no customers the barber goes to sleep in barber chair and must be woken when a customer comes in. When barber is cutting hair new customers take empty seats to wait, or leave if no vacancy.

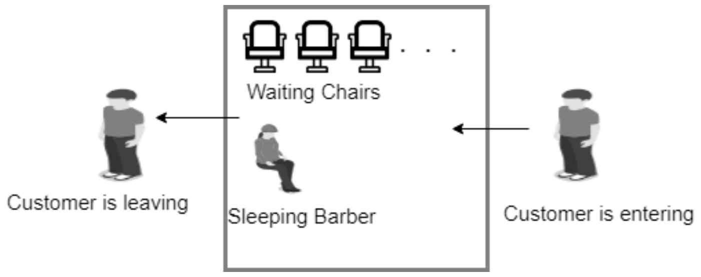


Fig.: Overview of sleeping barber problem

9. INTRODUCTION TO DEADLOCKS

A Deadlock is a situation where each of the computer process waits for a resource which is being assigned to some another process. In this situation, none of the process gets executed since the resource it needs, is held by some other process which is also waiting for some other resource to be released.

Let us assume that there are three processes P1 and P2. There are two different resources R1, R2. R1 is assigned to P1, R2 is assigned to P2. After some time, P1 demands for R2 which is being used by P2. P1 halts its execution since it can't complete without R2. P2 also demands for R1 which is being used by P1. P2 also stops its execution because it can't continue without R2.

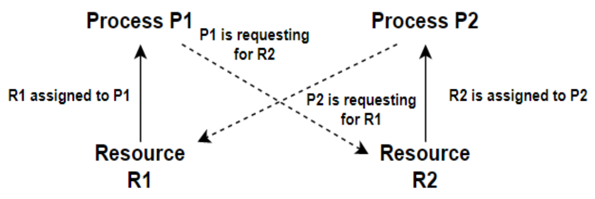


Fig.: Overview of deadlock

In this scenario, a cycle is being formed among the three processes. None of the process is progressing and they are all waiting. The computer becomes unresponsive since all the processes got blocked.

10. System Model

For the purposes of deadlock, a system can be modelled as a collection of limited resources that can be divided into different categories and allocated to a variety of processes, each with different requirements. Memory, printers, CPUs, open files, tape drives, CD-ROMs, and other resources are examples of resource categories.

By definition, all resources within a category are equivalent, and any of the resources within that category can equally satisfy a request from that category. If this is not the case, then that category must be subdivided further. Some categories may only have one resource.

The kernel keeps track of which resources are free and which are allocated, to which process they are allocated, and a queue of processes waiting for this resource to become available for all kernel-managed resources. Mutexes or wait() and signal() calls can be used to control application-managed resources i.e. binary or counting semaphores. When every process in a set is waiting for a resource that is currently assigned to another process in the set, the set is said to be deadlocked.

Operations: In normal operation, a process must request a resource before using it and release it when finished, as shown below.

1. **Request** – If the request cannot be granted immediately, the process must wait until the resource(s) required to become available. The system, for example, uses the functions open(), malloc(), new(), and request ().

2. **Use** – The process makes use of the resource, such as printing to a printer or reading from a file.

3. **Release** – The process relinquishes the resource, allowing it to be used by other processes.

11. Deadlock Characterization

There are two characteristics of deadlock from which we can identify the deadlock is appearing or not.

12. Necessary conditions for Deadlocks

1. **Mutual Exclusion:** A resource can only be shared in mutually exclusive manner. It implies, if two process cannot use the same resource at the same time.

2. **Hold and Wait:** A process waits for some resources while holding another resource at the same time.

3. **No pre-emption**: The process which once scheduled will be executed till the completion. No other process can be scheduled by the scheduler meanwhile.

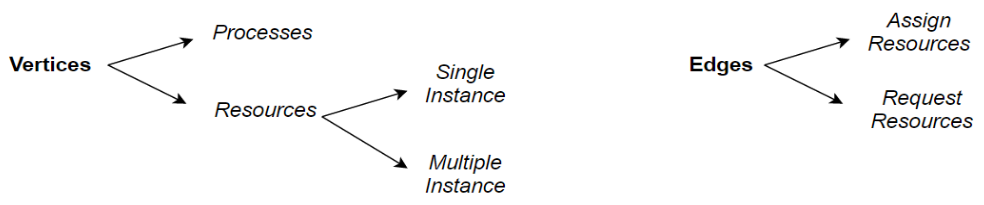
4. **Circular Wait:** All the processes must be waiting for the resources in a cyclic manner so that the last process is waiting for the resource which is being held by the first process.

13. Resource Allocation Graph

The resource allocation graph is the pictorial representation of the state of a system. As its name suggests, the resource allocation graph is the complete information about all the processes which are holding some resources or waiting for some resources.

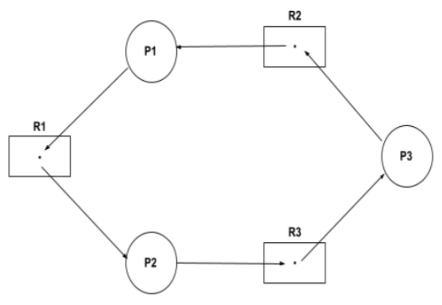
It also contains the information about all the instances of all the resources whether they are available or being used by the processes. In Resource allocation graph, the process is represented by a Circle while the Resource is represented by a rectangle.

The following hierarchy specifies the vertices and edges types.



The RAG represents by two types of matrices which are Allocation matrix and Request matrix, The allocation matrix set the value 1 for assigned resources and request matrix set the value 1 for request resources.

Example 1: Consider the following example of RAG of single instance.



Allocation Matrix:

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

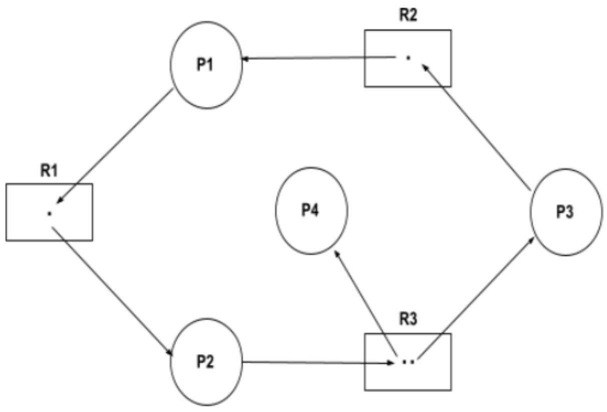
Resource Matrix:

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

Here, from the matrix above, the available instance of R1 is 0, R2 is 0 and R3 is 0, and each of the processes is waiting for any one resource. So, none of the processes will get executed and there will be a deadlock.

Note: If there is a cycle in a single instance RAG then the processes will be in a deadlock.

Example 2: Consider the following example of RAG of multiple instances.



Allocation Matrix:

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

Resource Matrix:

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

As R2 is allocated to P1, write 1 in the Allocation Matrix corresponding to P1 and R1 and others 0 and similarly for all others. P1 is waiting for R1 so, write 1 in the Request Matrix corresponding to P1 and R1 and others 0 and similarly for all others.

To check the deadlock, we have to perform the following steps:

14. First, find the currently available instances of each resource.
15. Check for each process which can be executed using the allocated + available resource.
16. Add the allocated resource of the executable process to the available resources and terminate it.
17. Repeat the 2nd and 3rd steps until the execution of each process.
18. If at any step, none of the processes can be executed then there is a deadlock in the system.

Using the above algorithm, we will get that there is no deadlock in the above-given example, and their sequence of execution can be P4 → P2 → P1 → P3.

Note: Unlike Singles Instances RAG, a cycle in a Multiple Instances RAG does not guarantee that the processes are in deadlock.

19. Deadlock Prevention

Deadlock prevention is eliminating one of the necessary conditions of deadlock so that only safe requests are made to OS and the possibility of deadlock is excluded before making requests. The operating system can grant all requests safely. Here OS does not need to do any additional tasks as it does in deadlock avoidance by running an algorithm on requests checking for the possibility of deadlock.

Deadlock prevention techniques refer to violating any one of the four necessary conditions.

1. Mutual Exclusion

Mutual section from the resource point of view is the fact that a resource can never be used by more than one process simultaneously which is fair enough but that is the main reason behind the deadlock. If a resource could have been used by more than one process at the same time then the process would have never been waiting for any resource. However, if we can be able to violate resources behaving in the mutually exclusive manner then the deadlock can be prevented.

2. Hold and Wait

Hold and wait condition lies when a process holds a resource and waiting for some other resource to complete its task. Deadlock occurs because there can be more than one process which are holding one resource and waiting for other in the cyclic order. However, we have to find out some mechanism by which a process either doesn't hold any resource or doesn't wait. That means, a process must be assigned all the necessary resources before the execution starts. A process must not wait for any resource once the execution has been started.

!(Hold and wait) = !hold or !wait (negation of hold and wait is, either you don't hold or you don't wait)

This can be implemented practically if a process declares all the resources initially. However, this sounds very practical but can't be done in the computer system because a process can't determine necessary resources initially. The problem with the approach is:

20. Practically not possible.
21. Possibility of getting starved will be increases due to the fact that some process may hold a resource for a very long time.

3. No Pre-emption

Deadlock arises due to the fact that a process can't be stopped once it starts. However, if we take the resource away from the process which is causing deadlock then we can prevent deadlock. This is not a good approach at all since if we take a resource away which is being used by the process then all the work which it has done till now can become inconsistent.

4. Circular Wait

To violate circular wait, we can assign a priority number to each of the resource. A process can't request for a lesser priority resource. This ensures that not a single process can request a resource which is being utilized by some other process and no cycle will be formed.

22. Deadlock Avoidance

The general idea behind deadlock avoidance is to prevent deadlocks from ever happening, by preventing at least one of the mentioned conditions. This requires more information about each process, and tends to lead to low device utilization.

In some algorithms the scheduler only needs to know the maximum number of each resource that a process might potentially use. In more complex algorithms the scheduler can also take advantage of the schedule of exactly what resources may be needed in what order.

When a scheduler sees that starting a process or granting resource requests may lead to future deadlocks, then that process is just not started or the request is not granted. A resource allocation state is defined by the number of available and allocated resources, and the maximum requirements of all processes in the system.

1. Safe State

A state is safe if the system can allocate all resources requested by all processes without entering a deadlock state.

A state is safe if there exists a safe sequence of processes { P0, P1, P2, ..., PN } such that all of the resource requests for Pi can be granted using the resources currently allocated to Pi and all processes Pj where j < i.

If a safe sequence does not exist, then the system is in an unsafe state, which may lead to deadlock

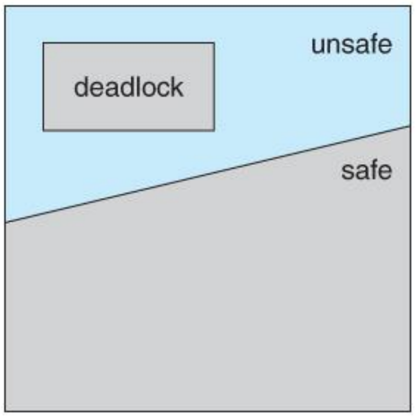


Fig: Safe, unsafe, and deadlocked state spaces.

2. Resource Allocation Graph Algorithm

If resource categories have only single instances of their resources, then deadlock states can be detected by cycles in the resource-allocation graphs.

In this case, unsafe states can be recognized and avoided by augmenting the resource-allocation graph with claim edges, noted by dashed lines, which point from a process to a resource that it may request in the future.

In order for this technique to work, all claim edges must be added to the graph for any particular process before that process is allowed to request any resources**.**

This approach works by denying requests that would produce cycles in the resource-allocation graph, taking claim edges into effect.

Example: Consider for example what happens when process P2 requests resource R2:

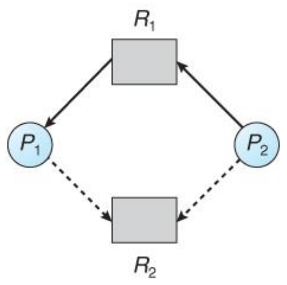


Fig: Resource allocation graph for deadlock avoidance

The resulting resource-allocation graph would have a cycle in it, and so the request cannot be granted.

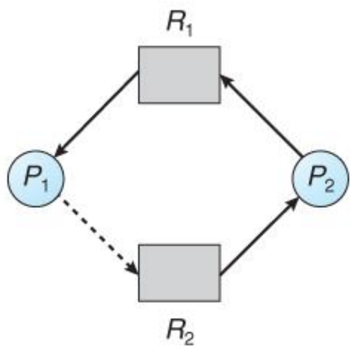


Fig: An unsafe state in a resource allocation graph

3. Bankers's Algorithm

For resource categories that contain more than one instance the resource-allocation graph method does not work, and more complex methods must be chosen.

The Banker's Algorithm gets its name because it is a method that bankers could use to assure that when they lend out resources, they will still be able to satisfy all their clients.

When a process starts up, it must state in advance the maximum allocation of resources it may request, up to the amount available on the system. When a request is made, the scheduler determines whether granting the request would leave the system in a safe state. If not, then the process must wait until the request can be granted safely. The banker's algorithm relies on several key data structures:

- n is the number of processes and m is the number of resource categories.
- Available[m] indicates how many resources are currently available of each type.
- Max[n][m] indicates the maximum demand of each process of each resource.
- Allocation[n][m] indicates the number of each resource category allocated to each process.
- Need[n][m] indicates the remaining resources needed of each type for each process.
- Need[i], can be treated as a vector corresponding to the needs of process i, and similarly for Allocation and Max.
- A vector X is considered to be <= a vector Y if X[i] <= Y[i] for all i.

4. Safety Algorithm

In order to apply the Banker's algorithm, we first need an algorithm for determining whether or not a particular state is safe.

This algorithm determines if the current state of a system is safe, according to the following steps:

1. Let Work and Finish be vectors of length m and n respectively.
   - Work is a working copy of the available resources, which will be modified during the analysis.
   - Finish is a vector of booleans indicating whether a particular process can finish. (or has finished so far in the analysis.)
   - Initialize Work to Available, and Finish to false for all elements.

2. Find an i such that both (A) Finish[i] == false, and (B) Need[i] < Work. This process has not finished, but could with the given available working set. If no such i exists, go to step 4.

3. Set Work = Work + Allocation[i], and set Finish[i] to true. This corresponds to process i finishing up and releasing its resources back into the work pool. Then loop back to step 2.
4. If finish[i] == true for all i, then the state is a safe state, because a safe sequence has been found.

➢ DEADLOCK DETECTION AND RECOVERY

If deadlocks are not avoided, then another approach is to detect when they have occurred and recover somehow. In addition to the performance hit of constantly checking for deadlocks, a policy / algorithm must be in place for recovering from deadlocks, and there is potential for lost work when processes must be aborted or have their resources pre-empted.

1. **Deadlock Detection If resources have a single instance** – In this case for Deadlock detection, we can run an algorithm to check for the cycle in the Resource Allocation Graph. The presence of a cycle in the graph is a sufficient condition for deadlock.

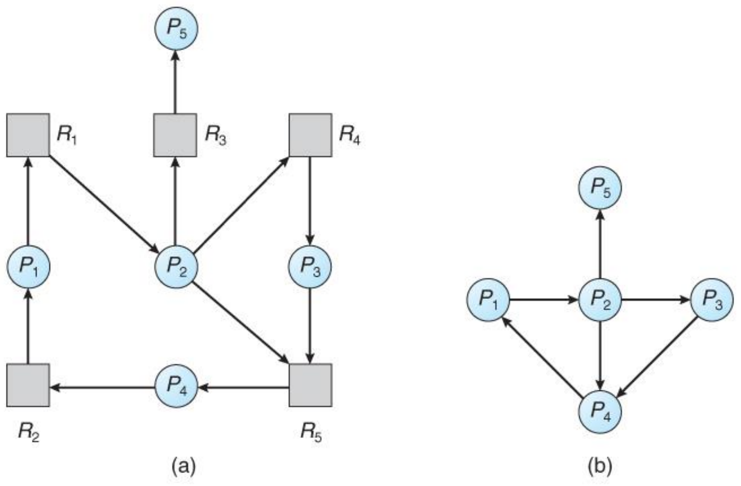


Figure- (a) Resource allocation graph. (b) Corresponding wait-for graph

2. **Deadlock Detection If there are multiple instances of resources-** Detection of the cycle is necessary but not sufficient condition for deadlock detection, in this case, the system may or may not be in deadlock varies according to different situations.

➢ Deadlock Recovery:

There are three basic approaches to recovery from deadlock:

- Inform the system operator, and allow him/her to take manual intervention.
- Terminate one or more processes involved in the deadlock
- Pre-empt resources.

1. **Terminate the process** – Killing all the processes involved in the deadlock. Killing process one by one. After killing each process check for deadlock again keep

repeating the process till the system recovers from deadlock. Killing all the processes one by one helps a system to break circular wait condition.

2. Resource Pre-emption – When pre-empting resources to relieve deadlock, there are three important issues to be addressed:

- **Selecting a victim** - Deciding which resources to pre-empt from which processes involves many of the same decision criteria outlined above.

- **Rollback** - Ideally one would like to roll back a pre-empted process to a safe state prior to the point at which that resource was originally allocated to the process. Unfortunately, it can be difficult or impossible to determine what such a safe state is, and so the only safe rollback is to roll back all the way back to the beginning.

- **Starvation** - How do you guarantee that a process won't starve because its resources are constantly being pre-empted? One option would be to use a priority system, and increase the priority of a process every time its resources get pre-empted. Eventually it should get a high enough priority that it won't get pre-empted any more.

- **Difference between Deadlock and Starvation**

Sr. No.	Deadlock	Starvation
1.	All processes keep waiting for each other to complete and none get executed	High priority processes keep executing and low priority processes are blocked
2.	Resources are blocked by the processes	Resources are continuously utilized by high priority processes
3.	Necessary conditions Mutual Exclusion, Hold and Wait, No pre-emption, Circular Wait	Priorities are assigned to the processes
4.	Also known as Circular wait	Also known as lived lock
5.	It can be prevented by avoiding the necessary conditions for deadlock	It can be prevented by Aging