Scheme of Evaluation

Internal Assessment Test 2 – April 2024

Sub:	**Operating System Concepts**							**Sub Code:**	**22MCA12**
Date:	08-04-24	Duration:	90 mins	Max Marks:	50	**Sem:**	I	**Branch:**	**MCA**

Q.NO	**Description**	**Marks Distribution**	**Max Marks**
1	**What do you mean by Critical Section problem? Illustrate Peterson's solution for a critical section problem.** • Definition of Critical section Problem • Illustrate the Peterson's solution	**4** **6**	**10**
2	**What are Semaphores? Explain the process of implementation of a Semaphore with an example.** • Definition of Semaphores • Implementation of semaphore with an example	**3** **7**	**10**
3	**Explain the readers-writers problem and give a solution using semaphores.** • Explanation of readers-writers problem using semaphores	**10**	**10**
4	**Explain the producer-consumer problem and give a solution using semaphores.** • Explanation of producer-consumer problem using semaphores	**10**	**10**
5	**What are monitors? Explain in detail with syntax.** • Definition of monitors • Explanation of monitors with syntax	**2** **8**	**10**
6	**What is a deadlock? What are the necessary conditions for a deadlock to occur?** • Definition of Deadlock • List of necessary conditions for a deadlock to occur	**3** **7**	**10**
7	**What is a thread? Explain the various types of threads.** • Definition of thread • Types of thread with detail notes	**2** **8**	**10**

8	**With neat diagrams explain Resource Allocation graph.** • Explanation of RAG with diagarms	**10**	**10**
9	**Write and explain Banker's algorithm with an example.** • Explanation of Banker's Algorithm. • Give an example of it	**5** **5**	**10**
10	**What are the various approaches used in Deadlock prevention.** • Explanation of deadlock prevention • Detail notes on various approaches used in deadlock prevention	**3** **7**	**10**

Internal Assessment Test 2 – April 2024

Operating System Concepts							Sub Code:	22MCA12
08/04/2024	Duration:	90 min’s	Max Marks:	50	Sem:	I	Branch:	MCA

PART I

1. **What do you mean by Critical Section problem? Illustrate Peterson’s solution for a critical section problem.**

Critical section problem

A critical section is a code segment that accesses shared variables and has to be executed as an atomic action. It means that in a group of co-operating processes, at a given point of time, only one process must be executing its critical section. If other processes also want to execute its critical section, it must wait until the first one finishes.

Critical section:

- It is the part of the program where shared resources are accessed by various processes.
- It is the place where shred variable, resources are placed.

Solution to Critical section Problem

A solution to the critical section problems must satisfy the following three conditions:

1. Mutual exclusion
2. Progress
3. Bounded waiting
4. No assumption related to H/W speed

1. Mutual exclusion

Out of a group of co-operating processes, only one process can be in its critical section at a given point of time.

2. Progress:

If no process is in its critical section and if one or more process wants to execute in critical section than one of these process must be allowed to get into its critical section.

3. Bounded waiting:

After a process makes a request for getting into its critical section, there is a limit for how many other processes can get into their critical section, before this process’s request is granted. So after the limit time is reached, system must grant the process permission to get into its critical section.

4. No assumption related to H/W speed

Peterson’s Solution

- Good algorithmic description of solving the problem

  Two process solution
- Assume that the load and store machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:

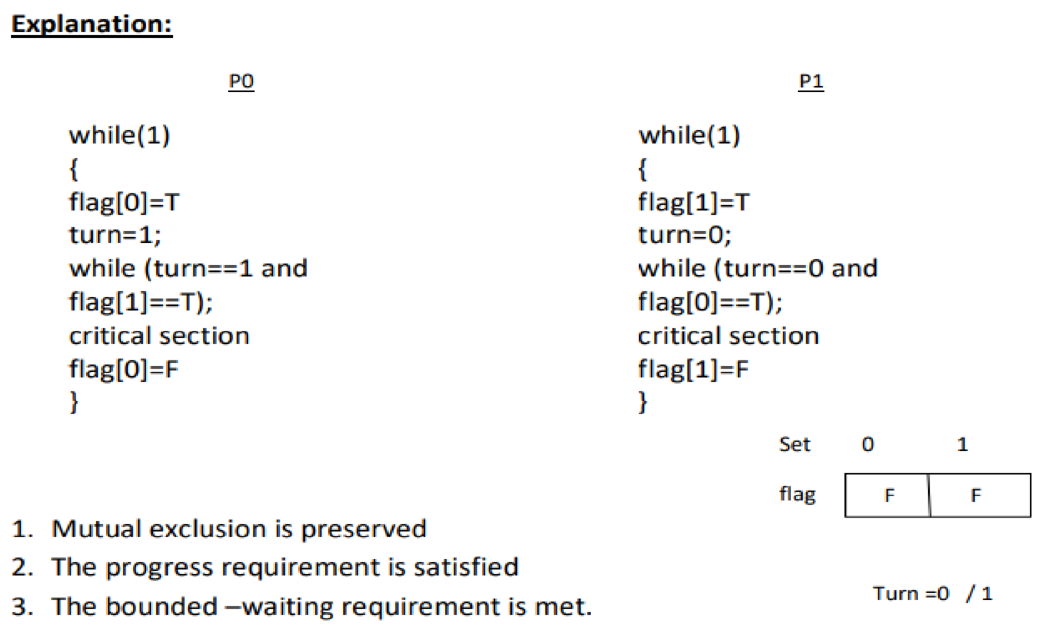
  int turn;

Boolean flag[2]

- ➢ The variable turn indicates whose turn it is to enter the critical section
- ➢ The flag array is used to indicate if a process is ready to enter the critical section.
- ➢ flag[i] = true implies that process Pi is ready!

Algorithm for Process Pi

```
do {
flag[i] = true;
turn = j;
while (flag[j] && turn = = j);

critical section

flag[i] = false;

remainder section

} while (true);
```



Explanation:

P0

```
while(1)
{
flag[0]=T
turn=1;
while (turn==1 and
flag[1]==T);
critical section
flag[0]=F
}
```

P1

```
while(1)
{
flag[1]=T
turn=0;
while (turn==0 and
flag[0]==T);
critical section
flag[1]=F
}
```

Set	0	1
flag	F	F

Turn =0 / 1

1. Mutual exclusion is preserved
2. The progress requirement is satisfied
3. The bounded –waiting requirement is met.

2. **What are Semaphores? Explain the process of implementation of a Semaphore with an example.**

Semaphores

- ➢ Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- ➢ Semaphore S – integer variable
- ➢ Can only be accessed via two indivisible (atomic) operations
  - wait() and signal()
  - Originally called P() and V()
- ➢ Definition of the wait() operation

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

- ➢ Definition of the signal() operation

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

- ➢ Counting semaphore – integer value can range over an unrestricted domain
- ➢ Binary semaphore – integer value can range only between 0 and 1
  - Same as a mutex lock
- ➢ Can solve various synchronization problems
- ➢ Consider P1 and P2 that require S1 to happen before S2

Create a semaphore "synch" initialized to 0

```
P1:
S1;
signal(synch);
P2:
wait(synch);
S2;
```

- ➢ Can implement a counting semaphore S as a binary semaphore

Semaphore Implementation

- ➢ Must guarantee that no two processes can execute the wait() and signal() on the same semaphore at the same time.
- ➢ Thus, the implementation becomes the critical section problem where the wait and signal code are placed in the critical section
  - Could now have busy waiting in critical section implementation
  - But implementation code is short
  - Little busy waiting if critical section rarely occupied

- ➢ Note that applications may spend lots of time in critical sections and therefore this is not a good solution
- ➢

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
    - value (of type integer)
    - pointer to next record in the list

Two operations:

- block – place the process invoking the operation on the appropriate waiting queue
- wakeup – remove one of processes in the waiting queue and place it in the ready queue

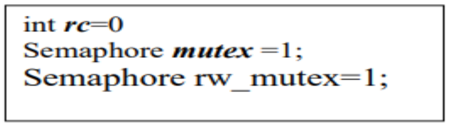
```
 typedef struct{
int value;
struct process *list;
} semaphore;

wait(semaphore *S) {
S->value--;
if (S->value < 0) {
add this process to S->list;
block();
}
}
signal(semaphore *S) {
S->value++;
if (S->value <= 0) {
remove a process P from S->list;
wakeup(P);
}
}
```

PART II

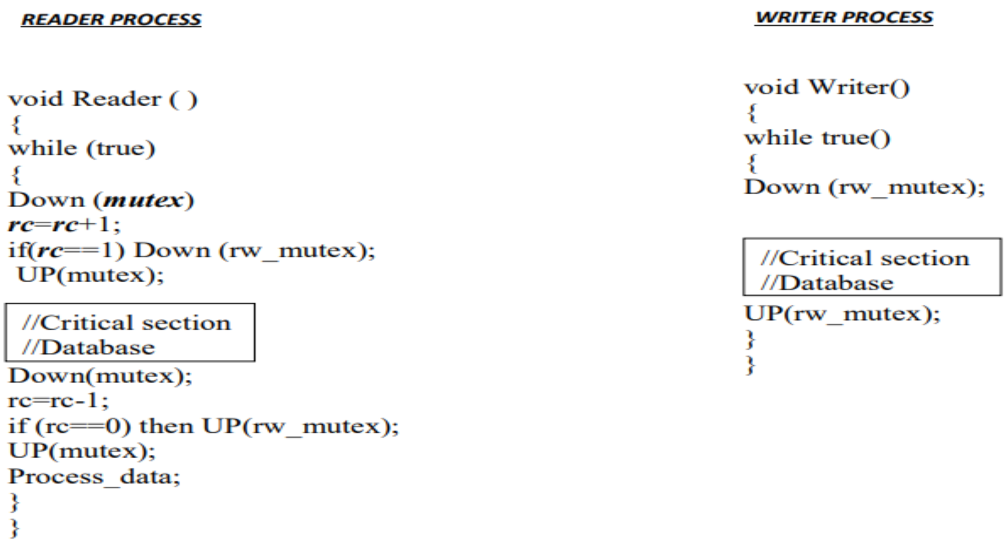
3. **Explain the readers-writers problem and give a solution using semaphores.**
   Definition:
   There is a data containing some files ,records etc that is shared among the number of concurrent processes. The processes that reads the data from that common shared data area are called reader processes and processes that perform write operation(writing new data value or updating or modifying the data value) on the data stored in common shared data area are called writer processes. The various conditions that need to take care in Reader-writer case are:
    - Any number of reader processes can simultaneously read the data from common shared data area but only one writer at a time may write to that common shared data area.
    - If any of the writer process is writing to common shared data area, then no reader processes are allowed to read it till the writer process has finished.
    - If there is at least one reader reading the common data area, no writer processes are allowed to that common data area

- The reader-writer problem solution using semaphores consists of two binarysemaphores-mutex and rw_mutex and one integer variable NumberOfReaders(rc).

- The semaphore rw_mutex is shared by the all the processes and the semaphore mutex and the integer variable NumberOfReaders(rc) is shared by reader processes only.
- Here, variable NumberOfReaders(rc) keep track of how many reader processes are reading the common shared data at a time, and mutex provide mutual exclusion among reader processes when variable NumberOfReaders(rc) is incremented or decremented
- The semaphore rw_mutex which is common to both readers and writers processes ensures that when one writer process is using the common data area, no other reader or writer processes can access that common data area



```
int rc=0
Semaphore mutex =1;
Semaphore rw_mutex=1;
```

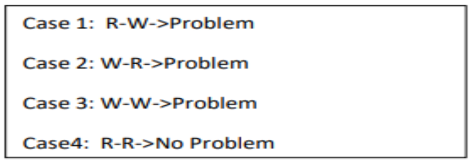


READER PROCESS

```
void Reader ( )
{
while (true)
{
Down (mutex)
rc=rc+1;
if(rc==1) Down (rw_mutex);
 UP(mutex);
  //Critical section
  //Database
Down(mutex);
rc=rc-1;
if (rc==0) then UP(rw_mutex);
UP(mutex);
Process_data;
}
}
```

WRITER PROCESS

```
void Writer()
{
while true()
{
Down (rw_mutex);
  //Critical section
  //Database
UP(rw_mutex);
}
}
```



Case 1: R-W->Problem

Case 2: W-R->Problem

Case 3: W-W->Problem

Case4: R-R->No Problem

4. Explain the producer-consumer problem and give a solution using semaphores.

Concurrent access to shared data may result in data inconsistency (change in behavior)
Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
Suppose that we wanted to provide a solution to the “producer-consumer” problem that fills all the buffers.

We can do so by having an integer variable "count" that keeps track of the number of full buffers.
Initially, count is set to 0.
It is incremented by the producer after it produces a new buffer.
It is decremented by the consumer after it consumes a buffer.

```
Producer
while (true) {
/* produce an item and put in next Produced */
while (count == BUFFER_SIZE)
; // do nothing
buffer [in] = next Produced;
in = (in + 1) % BUFFER_SIZE;
count++;

}
Consumer
while (true) {

while (count == 0)
; // do nothing
next Consumed = buffer[out];
out = (out + 1) % BUFFER_SIZE;
count--;
/* consume the item in next Consumed

}
```

PART III

5. What are monitors? Explain in detail with syntax.

- A feature of programming languages called monitors helps control access to shared data. The Monitor is a collection of shared actions, data structures, and synchronization between parallel procedure calls. A monitor is therefore also referred to as a synchronization tool. Some of the languages that support the usage of monitors include Java, C#, Visual Basic, Ada, and concurrent Euclid. Although they can call the monitor's procedures, processes running outside of the monitor are unable to access its internal variables.
- For example, the Java programming language provides synchronization mechanisms like the wait() and notify() constructs.

- Monitor in os has a simple syntax similar to how we define a class, it is as follows:

```
Monitor monitorName{
variables_declaration;
condition_variables;
procedure p1{ ... };
procedure p2{ ... };
...
procedure pn{ ... };

{
   initializing_code;
}
}
```

In an operating system, a monitor is only a class that includes variable_declarations, condition_variables, different procedures (functions), and an initializing_code block for synchronizing processes.

Characteristics of Monitors in OS

Monitors in operating systems possess several key characteristics that make them valuable tools for managing concurrent access to shared resources. Here are the main characteristics of monitors:

- **Mutual Exclusion:** Monitors ensure mutual exclusion, which means only one process or thread can be inside the monitor at any given time. This property prevents concurrent processes from accessing shared resources simultaneously and eliminates the risk of data corruption or inconsistent results due to race conditions.
- **Encapsulation:** Monitors encapsulate both the shared resource and the procedures that operate on it. By bundling the resource and the relevant procedures together, monitors provide a clean and organized approach to managing concurrent access. This encapsulation simplifies the design and maintenance of concurrent programs, as the necessary synchronization logic is localized within the monitor.
- **Synchronization Primitives:** Monitors often support synchronization primitives, such as condition variables. Condition variables enable threads within the monitor to wait for specific conditions to become true or to signal other threads when certain conditions are met. These primitives allow for efficient coordination among threads and help avoid busy-waiting, which can waste CPU cycles.
- **Blocking Mechanism:** When a process or thread attempts to enter a monitor that is already in use, it is blocked and put in a queue (entry queue) until the monitor becomes available. This blocking mechanism avoids busy-waiting and allows other processes to proceed while waiting for their turn to access the monitor.
- **Local Data:** Each thread that enters a monitor has its own local data or stack, which means the variables declared within a monitor procedure are unique to each thread's execution. This feature prevents interference between threads and ensures that data accessed within the monitor remains consistent for each thread.
- **Priority Inheritance:** In some advanced implementations of monitors, a priority inheritance mechanism can be used to prevent priority inversion. When a higher-priority thread is waiting for a lower-priority thread to release a resource inside the monitor, the lower-priority thread's priority may be temporarily elevated to avoid unnecessary delays caused by priority inversion scenarios.
- **High-Level Abstraction:** Monitors provide a higher-level abstraction for concurrency management compared to low-level synchronization mechanisms like semaphores or

spinlocks. This abstraction reduces the complexity of concurrent programming and makes it easier to write correct and maintainable code.

Components of Monitor in an operating system

In an operating system, a monitor is a synchronization construct that helps manage concurrent access to shared resources by multiple processes or threads. A monitor typically consists of the following main components:

- **Shared Resource:**
  The shared resource is the data or resource that multiple processes or threads need to access in a mutually exclusive manner. Examples of shared resources can include critical sections of code, global variables, or any data structure that needs to be accessed atomically.
- **Entry Queue:**
  The entry queue is a data structure that holds the processes or threads that are waiting to enter the monitor and access the shared resource. When a process or thread tries to enter the monitor while it is already being used by another process, it is placed in this queue, and its execution is temporarily suspended until the monitor becomes available.
- **Entry Procedures (or Monitor Procedures):**
  Entry procedures are special procedures that provide access to the shared resource and enforce mutual exclusion. When a process or thread wants to access the shared resource, it must call one of these entry procedures. The monitor's implementation ensures that only one process or thread can execute an entry procedure at a time, thus achieving mutual exclusion.
- **Condition Variables:**
  Condition variables enable communication and synchronization between processes or threads within the monitor. They allow threads to wait until a specific condition is satisfied or to signal other threads when certain conditions become true. Condition variables are crucial for avoiding busy-waiting, which can be inefficient and wasteful of system resources.
- **Local Data (or Local Variables):**
  Each process or thread that enters the monitor has its own set of local data or local variables. These variables are unique to each thread's execution and are not shared between threads. Local data allows each thread to work independently within the monitor without interfering with other threads' data.
- **Condition Variables**
  The condition variables of the monitor can be subjected to two different types of operations:
  1. Wait
  2. Signal

  Consider a condition variable (y) is declared in the monitor:

  y.wait(): The activity/process that applies the wait operation on a condition variable will be suspended, and the suspended process is located in the condition variable's block queue.

  y.signal(): If an activity/process applies the signal action on the condition variable, then one of the blocked activity/processes in the monitor is given a chance to execute.

Advantages of Monitor in OS

- Compared to semaphore-based solutions, monitors have the advantage of making concurrent or parallel programming simpler and less error-prone.
- It helps in operating system process synchronization.
- Monitors have mutual exclusion built in.
- Semaphores are more difficult to set up than monitors.
- Semaphores can lead to timing errors, which monitors may be able to fix.

Disadvantages of Monitor in OS

- Monitors must be implemented with the programming language.
- Monitor puts more work on the compiler.
- The monitor needs to be aware of the features offered by the operating system for managing critical steps in the parallel processes.

6. **What is a deadlock? What are the necessary conditions for a deadlock to occur?**

➢ Every process needs some resources to complete its execution. However, the resource is granted in a sequential order.

1. The process requests for some resource.
2. OS grant the resource if it is available otherwise let the process waits.
3. The process uses it and release on the completion.

➢ 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, P2 and P3. There are three different resources R1, R2 and R3. R1 is assigned to P1, R2 is assigned to P2 and R3 is assigned to P3.

➢ After some time, P1 demands for R1 which is being used by P2. P1 halts its execution since it can't complete without R2. P2 also demands for R3 which is being used by P3. P2 also stops its execution because it can't continue without R3. P3 also demands for R1 which is being used by P1 therefore P3 also stops its execution.

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 preemption

➢ 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. 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.

SYSTEM MODEL

A deadlock occurs when a set of processes is stalled because each process is holding a resource and waiting for another process to acquire another resource. In the diagram below, for example, Process 1 is holding Resource 1 while Process 2 acquires Resource 2, and Process 2 is waiting for Resource 1.

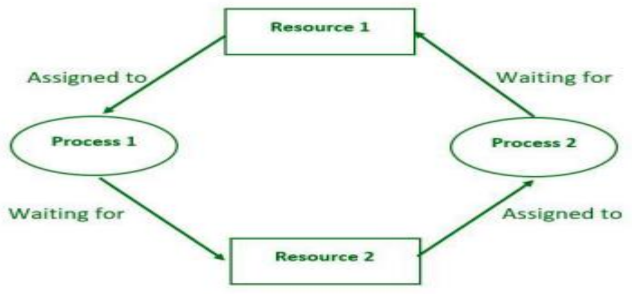


Figure: Deadlock in Operating system

System Model :

- For the purposes of deadlock discussion, a system can be modeled 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 (i.e. if there is some difference between the resources within a category), then that category must be subdivided further. For example, the term "printers" may need to be subdivided into "laser printers" and "color inkjet printers."
- 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.
Necessary Conditions
There are four conditions that must be met in order to achieve deadlock as follows.
1. Mutual Exclusion –
At least one resource must be kept in a non-shareable state; if another process requests it, it must wait for it to be released.

2. Hold and Wait –
A process must hold at least one resource while also waiting for at least one resource that another process is currently holding.

3. No preemption –
Once a process holds a resource (i.e. after its request is granted), that resource cannot be taken away from that process until the process voluntarily releases it.

4. Circular Wait –
There must be a set of processes P0, P1, P2,..., PN such that every P[I] is waiting for P[(I + 1) percent (N + 1)]. (It is important to note that this condition implies the hold-and-wait condition, but dealing with the four conditions is easier if they are considered separately).

PART IV

7. **What is a thread? Explain the various types of threads.**
A thread is a single sequence stream within a process. Threads are also called lightweight processes as they possess some of the properties of processes. Each thread belongs to exactly one process. In an operating system that supports multithreading, the process can consist of many threads. But threads can be effective only if the CPU is more than 1 otherwise two threads have to context switch for that single CPU.

- Threads run in parallel improving the application performance. Each such thread has its own CPU state and stack, but they share the address space of the process and the environment.
- Threads can share common data so they do not need to use inter-process communication. Like the processes, threads also have states like ready, executing, blocked, etc.
- Priority can be assigned to the threads just like the process, and the highest priority thread is scheduled first.
- Each thread has its own Thread Control Block (TCB). Like the process, a context switch occurs for the thread, and register contents are saved in (TCB). As threads share the same address space and resources, synchronization is also required for the various activities of the thread.

Components of Threads

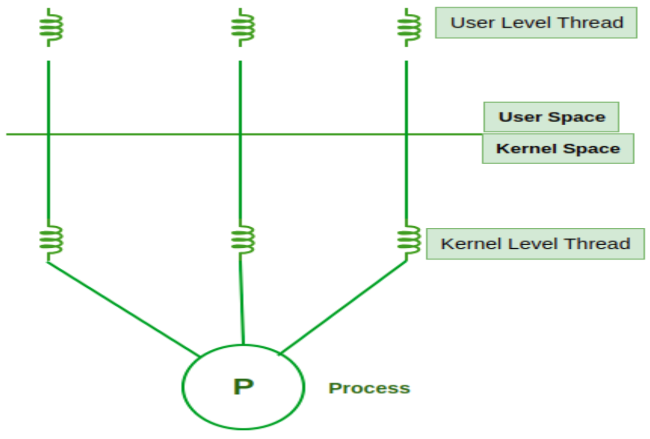
These are the basic components of the Operating System.

- Stack Space
- Register Set
- Program Counter

Types of Thread in Operating System

Threads are of two types. These are described below.

- User Level Thread
- Kernel Level Thread



Threads

1. User Level Threads

User Level Thread is a type of thread that is not created using system calls. The kernel has no work in the management of user-level threads. User-level threads can be easily implemented by the user. In case when user-level threads are single-handed processes, kernel-level thread manages them. Let's look at the advantages and disadvantages of User-Level Thread.

Advantages of User-Level Threads

- Implementation of the User-Level Thread is easier than Kernel Level Thread.
- Context Switch Time is less in User Level Thread.
- User-Level Thread is more efficient than Kernel-Level Thread.
- Because of the presence of only Program Counter, Register Set, and Stack Space, it has a simple representation.

Disadvantages of User-Level Threads

- There is a lack of coordination between Thread and Kernel.
- Inc case of a page fault, the whole process can be blocked.

2. Kernel Level Threads

A kernel Level Thread is a type of thread that can recognize the Operating system easily. Kernel Level Threads has its own thread table where it keeps track of the system. The operating System Kernel helps in managing threads. Kernel Threads have somehow longer context switching time. Kernel helps in the management of threads.

Advantages of Kernel-Level Threads

- It has up-to-date information on all threads.
- Applications that block frequency are to be handled by the Kernel-Level Threads.
- Whenever any process requires more time to process, Kernel-Level Thread provides more time to it.

Disadvantages of Kernel-Level threads

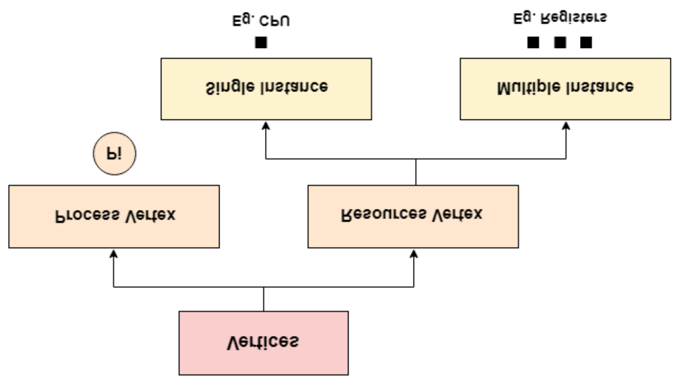
- Kernel-Level Thread is slower than User-Level Thread.
- Implementation of this type of thread is a little more complex than a user-level thread.

8. With neat diagrams explain 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.

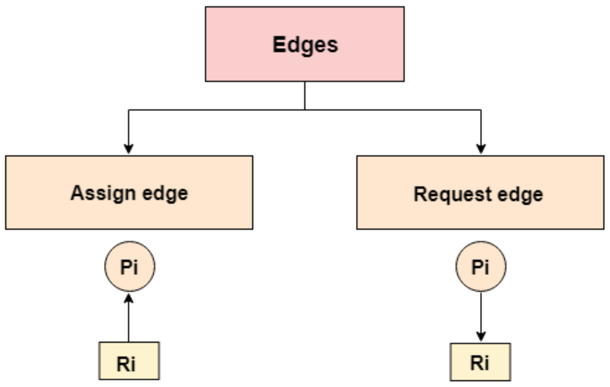
It also contains the information about all the instances of all the resources whether they are available or being used by the processes.

In Resource allocation graph, the process is represented by a Circle while the Resource is represented by a rectangle. Let's see the types of vertices and edges in detail.



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.

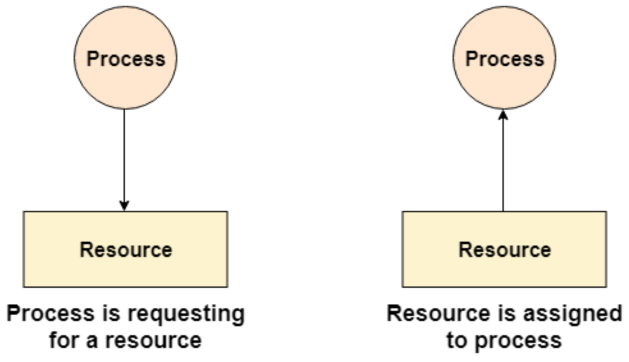
A resource can have more than one instance. Each instance will be represented by a dot inside the rectangle.



Edges in RAG are also of two types, one represents assignment and other represents the wait of a process for a resource. The above image shows each of them.

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.

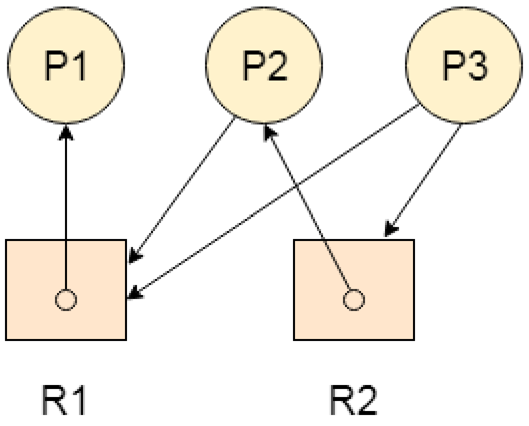
A process is shown as waiting for a resource if the tail of an arrow is attached to the process while the head is pointing towards the resource.


Process
Resource
Process is requesting for a resource
Process
Resource
Resource is assigned to process

Example
Let's 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.

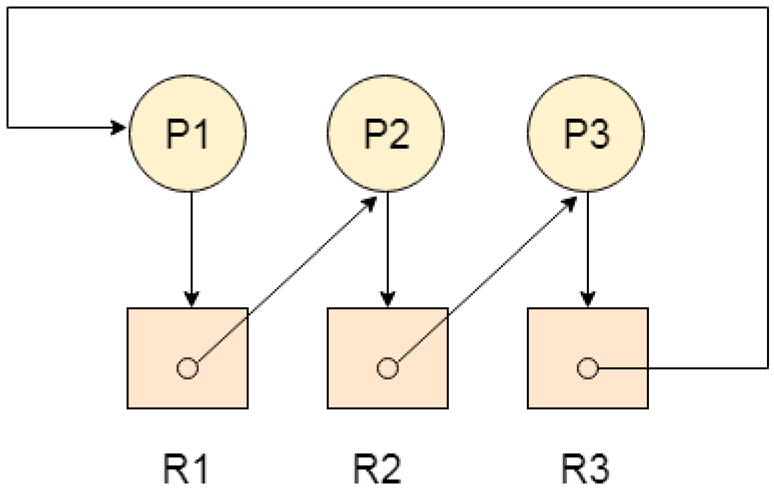


Deadlock Detection using RAG

If a cycle is being formed in a Resource allocation graph where all the resources have the single instance then the system is deadlocked.

In Case of Resource allocation graph with multi-instanced resource types, Cycle is a necessary condition of deadlock but not the sufficient condition.

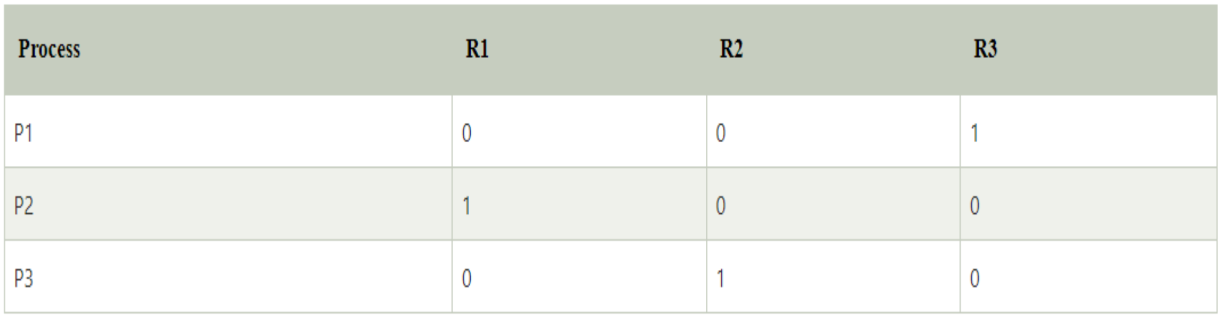
The following example contains three processes P1, P2, P3 and three resources R2, R2, R3. All the resources are having single instances each.



If we analyze the graph then we can find out that there is a cycle formed in the graph since the system is satisfying all the four conditions of deadlock.

Allocation Matrix

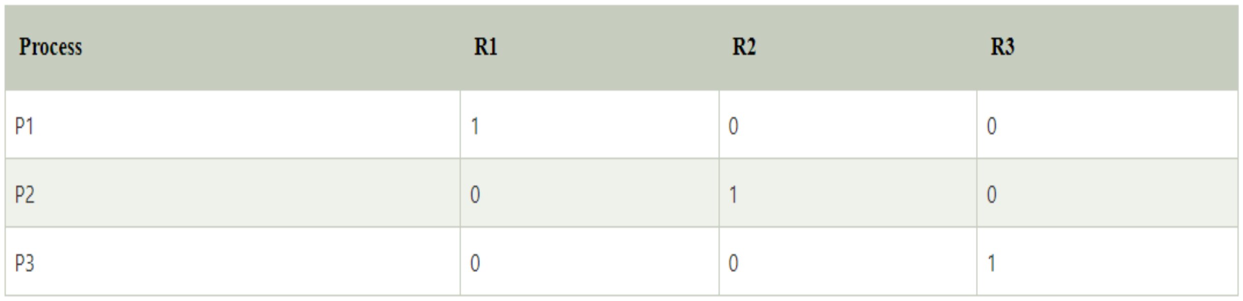
Allocation matrix can be formed by using the Resource allocation graph of a system. In Allocation matrix, an entry will be made for each of the resource assigned. For Example, in the following matrix, en entry is being made in front of P1 and below R3 since R3 is assigned to P1.



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

Request Matrix

In request matrix, an entry will be made for each of the resource requested. As in the following example, P1 needs R1 therefore an entry is being made in front of P1 and below R1.



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

Avial = (0,0,0)

Neither we are having any resource available in the system nor a process going to release. Each of the process needs at least single resource to complete therefore they will continuously be holding each one of them.We cannot fulfill the demand of at least one process using the available resources therefore the system is deadlocked as determined earlier when we detected a cycle in the graph.

PART V

9. Write and explain Banker’s algorithm with an example.

The banker’s algorithm is a resource allocation and deadlock avoidance algorithm
that tests for safety by simulating the allocation for predetermined maximum possible amounts of all resources, then makes an “s-state” check to test for possible activities, before deciding whether allocation should be allowed to continue.

Why Banker’s algorithm is name do so?

Banker’s algorithm is named so because it is used in banking system to check whether loan can be sanctioned to a person or not. Suppose there are n number of account holders in a bank and the total sum of their money is S. If a person applies for a loan then the bank first subtracts the loan amount from the total money that bank has and if the remaining amount is greater than S then only the loan is sanctioned. It is done because if all the account holders comes to withdraw their money then the bank can easily do it.

In other words, the bank would never allocate its money in such a way that it can no longer satisfy the needs of all its customers. The bank would try to be in safe state always. Following Data structures are used to implement the Banker’s Algorithm: Let ‘n’ be the number of processes in the system and ‘m’ be the number of resources types.

Available :
□ It is a 1-d array of size ‘m’ indicating the number of available resources of
each type.
□ Available[j] = k means there are ‘k’ instances of resource type Rj
Max :
□ It is a 2-d array of size ‘n x m’ that defines the maximum demand of each
process in a system.
□ Max[i] [j] = k means process Pi may request at most ‘k’ instances of
resource type Rj.

Allocation :
□ It is a 2-d array of size ‘n x m’ that defines the number of resources of each
type currently allocated to each process.
□ Allocation[i][j] = k means process Pi is currently allocated ‘k’ instances of
resource type Rj
Need :
□ It is a 2-d array of size ‘nxm’ that indicates the remaining resource need of
each process.
□ Need [i] [j] = k means process Pi currently need ‘k’ instances of resource
type Rj for its execution.
□ Need [i][j] = Max [i][j] – Allocation [i][j]

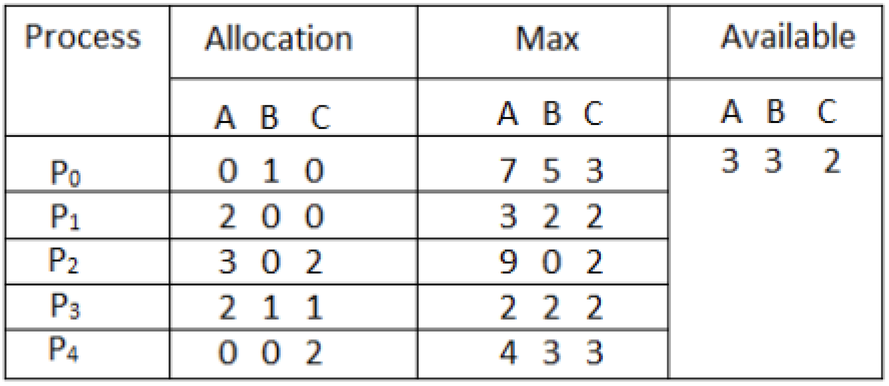
Allocationi specifies the resources currently allocated to process Pi and Needi specifies the additional resources that process Pi may still request to complete its task.
Banker’s algorithm consists of Safety algorithm and Resource request algorithm

ALGORITHM:

1) Let Work and Finish be vectors of length ‘m’ and ‘n’ respectively.
Initialize: Work = Available
Finish[i] = false; for i=0, 1, 2, 3....n-1;
2) Find an index i such that both
a) Finish[i] = false
b) Needi <= Work
if no such i exists goto step (4)

3) Work = Work + Allocation[i]
Finish[i] = true
goto step (2)
4) if Finish [i] = =true for all i
then the system is in a safe state
Where m= number of resource types
n=number of process in the system

Example:
Considering a system with five processes P0 through P4 and three resources of type A, B, C. Resource type A has 10 instances, B has 5 instances and type C has 7 instances. Suppose at time t0 following snapshot of the system has been taken:



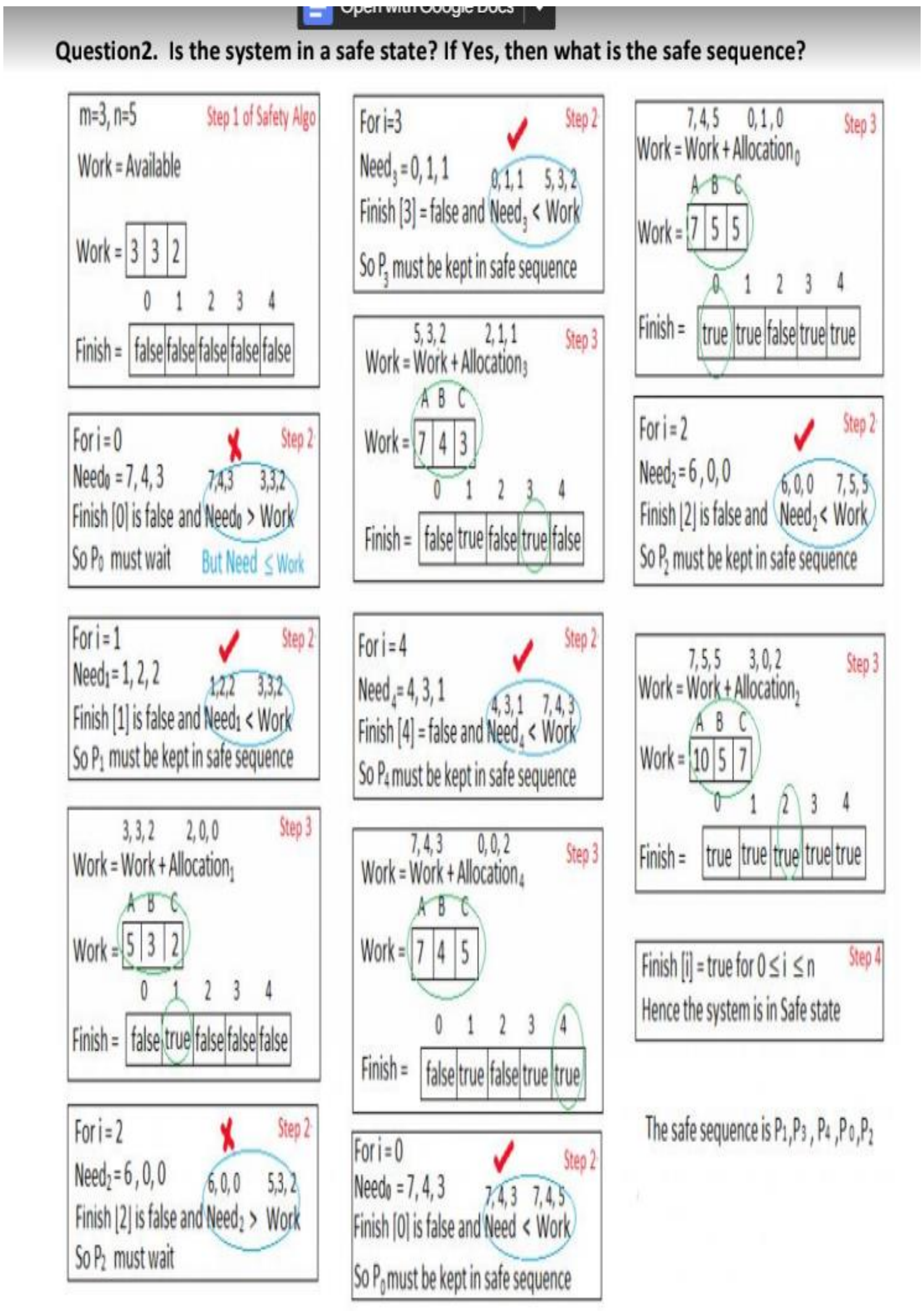
Process	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Question1. What will be the content of the Need matrix?

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



Question2. Is the system in a safe state? If Yes, then what is the safe sequence?

m=3, n=5 — Step 1 of Safety Algo

Work = Available

Work = | 3 | 3 | 2 |

	0	1	2	3	4
Finish =	false	false	false	false	false

For i = 0 ✗ — Step 2

$Need_0$ = 7, 4, 3 — 7,4,3 3,3,2

Finish [0] is false and $Need_0$ > Work

So P_0 must wait — But Need ≤ Work

For i = 1 ✓ — Step 2

$Need_1$ = 1, 2, 2 — 1,2,2 3,3,2

Finish [1] is false and $Need_1$ < Work

So P_1 must be kept in safe sequence

Step 3

3, 3, 2 — 2, 0, 0

Work = Work + $Allocation_1$

Work = | 5 | 3 | 2 | (A B C)

	0	1	2	3	4
Finish =	false	true	false	false	false

For i = 2 ✗ — Step 2

$Need_2$ = 6 , 0, 0 — 6,0,0 5,3,2

Finish [2] is false and $Need_2$ > Work

So P_2 must wait

For i=3 ✓ — Step 2

$Need_3$ = 0, 1, 1 — 0,1,1 5,3,2

Finish [3] = false and $Need_3$ < Work

So P_3 must be kept in safe sequence

Step 3

5, 3, 2 — 2, 1, 1

Work = Work + $Allocation_3$

Work = | 7 | 4 | 3 | (A B C)

	0	1	2	3	4
Finish =	false	true	false	true	false

For i = 4 ✓ — Step 2

$Need_4$ = 4, 3, 1 — 4,3,1 7,4,3

Finish [4] = false and $Need_4$ < Work

So P_4 must be kept in safe sequence

Step 3

7, 4, 3 — 0, 0, 2

Work = Work + $Allocation_4$

Work = | 7 | 4 | 5 | (A B C)

	0	1	2	3	4
Finish =	false	true	false	true	true

For i = 0 ✓ — Step 2

$Need_0$ = 7, 4, 3 — 7,4,3 7,4,5

Finish [0] is false and Need < Work

So P_0 must be kept in safe sequence

Step 3

7, 4, 5 — 0, 1, 0

Work = Work + $Allocation_0$

Work = | 7 | 5 | 5 | (A B C)

	0	1	2	3	4
Finish =	true	true	false	true	true

For i = 2 ✓ — Step 2

$Need_2$ = 6 , 0, 0 — 6,0,0 7,5,5

Finish [2] is false and $Need_2$ < Work

So P_2 must be kept in safe sequence

Step 3

7, 5, 5 — 3, 0, 2

Work = Work + $Allocation_2$

Work = | 10 | 5 | 7 | (A B C)

	0	1	2	3	4
Finish =	true	true	true	true	true

Step 4

Finish [i] = true for 0 ≤ i ≤ n

Hence the system is in Safe state

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

10. What are the various approaches used in Deadlock prevention.

Deadlocks can be prevented by preventing at least one of the four required conditions:

Mutual Exclusion

Shared resources such as read-only files do not lead to deadlocks. Unfortunately some resources, such as printers and tape drives, require exclusive access by a single process.

Hold and Wait

To prevent this condition processes must be prevented from holding one or more resources while simultaneously waiting for one or more others.

There are several possibilities for this:

i. Require that all processes request all resources at one time. This can be wasteful of system resources if a process needs one resource early in its execution and doesn't need some other resource until much later.

ii. Require that processes holding resources must release them before requesting new resources, and then re-acquire the released resources along with the new ones in a single new request. This can be a problem if a process has partially completed an operation using a resource and then fails to get it re-allocated after releasing it.

iii. Either of the methods described above can lead to starvation if a process requires one or more popular resources.

No Preemption

Preemption of process resource allocations can prevent this condition of deadlocks, when it is possible.

i. One approach is that if a process is forced to wait when requesting a new resource, then all other resources previously held by this process are implicitly released, (preempted), forcing this process to reacquire the old resources along with the new resources in a single request, similar to the previous discussion.

ii. Another approach is that when a resource is requested and not available, then the system looks to see what other processes currently have those resources and are themselves blocked waiting for some other resource. If such a process is found, then some of their resources may get preempted and added to the list of resources for which the process is waiting.

iii. Either of these approaches may be applicable for resources whose states are easily saved and restored, such as registers and memory, but are generally not applicable to other devices such as printers and tape drives

Circular Wait

i. One way to avoid circular wait is to number all resources, and to require that processes request resources only in strictly increasing (or decreasing) order.
ii. In other words, in order to request resource Rj, a process must first release all Ri such that i >= j.
iii. One big challenge in this scheme is determining the relative ordering of the different resources