Internal Assessment Test II – April 2023

Operating System Concepts								Sub Code:	22MCA12
24/04/23	Duration:	90 min's	Max Marks:	50	Sem:	I	Branch:	MCA	

Scheme

1. What do you mean by Critical Section problem? Illustrate Peterson's solution for a critical section problem.
CS problem-3 marks
CS structure-2 marks
Algorithm-5 marks
OR
2.What are Semaphores? Explain the process of implementation of a Semaphore with an example.
Semaphore definition-2 marks
Synatax-3 marks
Example-3 marls
Implementation-2 marks
3.Explain the readers-writers problem and give a solution using semaphores.
Problem-2 marks
Algorithm- 8 marks
OR
4.Explain the producer-consumer problem and give a solution using semaphores.
Problem-2 marks
Algorithm- 8 marks

5.What are monitors? Explain in detail with syntax.
Definition- 2 marks
Syntax-3 marks
Diagram-3 marks
Implementation- 2marks
OR
6.What is a deadlock? What are the necessary conditions for a deadlock to occur?
Definition-2 marks
Each-4 marks

7.Explain how resource allocation graph can be used find deadlocks with examples.
RAG- 4 marks
Detection methods-6 marks

OR
8.With neat diagrams explain Resource Allocation graph.

Notations-4 marks
Explanation with ex-6 marks

9. Write and explain Banker's algorithm with an example.
Algorithm- 7 marks
Ex-3 marks

OR

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

Prevention approach- 2marks
Each- 2 marks

SOLUTION

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

Critical-Section Problem

• **Critical-section** is a segment-of-code in which a process may be
→ changing common variables
→ updating a table or
→ writing a file.
• Each process has a critical-section in which the shared-data is accessed.
• General structure of a typical process has following (Figure 2.12):

1) Entry-section

● Requests permission to enter the critical-section.

2) Critical-section

● Mutually exclusive in time i.e. no other process can execute in its critical-section.

3) Exit-section

● Follows the critical-section.

4) Remainder-section

Figure 2.12 General structure of a typical process

• Problem statement:
―Ensure that when one process is executing in its critical-section, no other process is to be allowed to execute in its critical-section‖.

• A solution to the problem must satisfy the following 3 requirements:

1) Mutual Exclusion:

● No more than one process can be in critical-section at a given time.

2) Progress:

● When no process is in the critical section, any process that requests entry into the critical section must be permitted without any delay..

3) Bounded Waiting (No starvation):

● There is an upper bound on the number of times a process enters the critical section, while another is waiting.

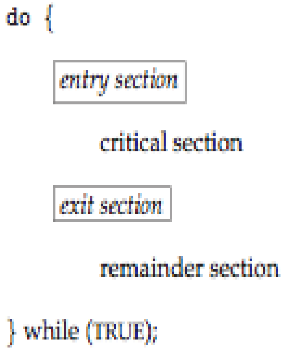
- Two approaches used to handle critical-sections:

1) Preemptive Kernels

- Allows a process to be preempted while it is running in kernel-mode.
- More suitable for real-time proframming

2) Non-preemptive Kernels

- Does not allow a process running in kernel-mode to be preempted as it is free from race conditions on kernel data structures, as only one process is active in the kernel at a time.



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

Figure 6.1 General structure of a typical process P_i.

Peterson's Solution

******Detailed understanding: https://nptel.ac.in/courses/106106144/26******

- This is a classic **software-based solution** to the critical-section problem.
- This is limited to 2 processes.
- The 2 processes alternate execution between

→ critical-sections and

→ remainder-sections.

```
do {
        flag[i] = TRUE;
        turn = j;
        while (flag[j] && turn == j);
                critical section
        flag[i] = FALSE;
                remainder section
   } while (TRUE);
```

- The 2 processes (say i & j)share two globally defined variables:

'turn' – indicates whose turn it is to enter its critical-section.

(i.e., if turn==i, then process Pi is allowed to execute in its critical-section).

'flag' – indicates if a process is ready to enter its critical-section.

(i.e. if flag[i]=true, then Pi is ready to enter its critical-section).

- The following code shows the structure of ***process Pi*** in Peterson's solution:

UNLOCK LOCK

- To enter the critical-section,

→ firstly, process Pi sets flag[i] to be true and
→ then sets turn to the value j.

- If both processes try to enter at the same time, turn will be set to both i and j at roughly the same time.
- The final value of turn determines which of the 2 processes is allowed to enter its critical-section first.
- To prove that this solution is correct, we show that:

1) Mutual-exclusion is preserved:

- Observation1: Pi enters the CS only if flag[j]== false or turn ==i.
- Observation2: If both processes can be executing in their CSs at the same time, then flag[i]==flag[j]==true.

These two observations imply that Pi and Pj could not have successfully executed their *while* statements at about the same time, since the value of turn can be either i or j but cannot be both.

Hence, the process which sets ‘turn’ first will execute and Mutual Exclusion is preserved.

2) The progress requirement & The bounded-waiting requirement is met:

- The process which executes while statement first (say Pi), doesn’t change the value of turn. So other process (Say Pj) will enter the CS (Progress) after at most one entry (Bounded Waiting)

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

Semaphores

******Detailed understanding:* ***https://nptel.ac.in/courses/106106144/30********

- A semaphore is a synchronization-tool.
- It used to control access to shared-variables so that only one process may at any point in time change the value of the shared-variable.
- A semaphore(S) is an integer-variable that is accessed only through 2 atomic-operations:

1) wait() and
2) signal().

- wait() is termed P ("to test or decrement”) signal() is termed V ("to increment").

Definition of wait(): Definition of signal():

- When one process modifies the semaphore-value, no other process can simultaneously modify that same semaphore-value.

Semaphore Usage:

a) Binary Semaphore

- The value of a semaphore can range only between 0 and 1.
- On some systems, binary semaphores are known as **mutex locks**, as they are locks that provide mutual-exclusion.
- Used for two processes

b) Counting Semaphore:

- The value of a semaphore can ranges over an unrestricted domain
- Used for multiple processes

```
wait (s) {
         while s <= 0  // no-op
           s = s -1
     }
signal (s) {
       s = s+1
    }
```

```
Semaphore mutex;    //  initialized to 1
do {
    wait (mutex);
       // Critical Section
    signal (mutex);
             // remainder section
    } while (TRUE);
```

Examples of Semaphore Usage:

1) Solution for Critical-section Problem using Binary Semaphores

- Binary semaphores can be used to solve the critical-section problem for multiple processes.
- The ‗n' processes share a semaphore mutex initialized to 1

Mutual-exclusion implementation with semaphores

2) Use of Counting Semaphores

- Counting semaphores can be used to control access to a given resource consisting of a finite number o£ instances.
- The semaphore is initialized to the number of resources available.
- Each process that wishes to use a resource performs a wait() operation on the semaphore (thereby decrementing the count).
- When a process releases a resource, it performs a signal() operation (incrementing the count).
- When the count for the semaphore goes to 0, all resources are being used.
- After that, processes that wish to use a resource will block until the count becomes greater than 0.

3) Solving Synchronization Problems

- Semaphores can also be used to solve synchronization problems.
- For example, consider 2 concurrently running-processes:
- P1 with a statement S1 and P2 with a with a statement S2
- Suppose we require that S2 be executed only after S1 has completed.

• We can implement this scheme readily

→ by letting P1 and P2 share a common semaphore synch initialized to 0, and

→ by inserting following statements in process P1:

S1;
Signal(synch);

→ by inserting following statements in process P2:

Wait(synch);
S2;

● Because synch is initialized to 0, P2 will execute S2 only after P1 has invoked signal (synch), which is after statement S1 has been executed.

3. Explain the readers-writers problem and give a solution using semaphores.

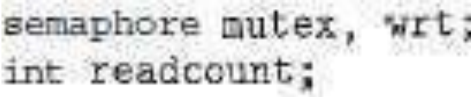
1) The Readers-Writers Problem

- A data set is shared among a number of concurrent processes.
- **Readers** are processes which want to only read the database (DB).
  Writers are processes which want to update (i.e. to read & write) the DB.
- Problem:
  - Obviously, if 2 readers can access the shared-DB simultaneously without anyproblems.
  - However, if a writer & other process (either a reader or a writer) access the shared-DB simultaneously, problems may arise.

Solution:

- The writers must have exclusive access to the shared-DB while writing to the DB.

● **Shared-data**



```
semaphore mutex, wrt;
int readcount;
```

where,

¤ mutex is used to ensure mutual-exclusion when the variable readcount is updated.

¤ wrt is common to both reader and writer processes.

wrt is used as a mutual-exclusion semaphore for the writers.

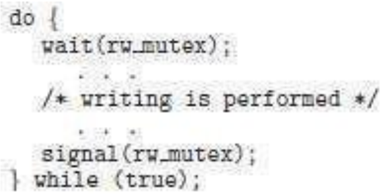
wrt is also used by the first/last reader that enters/exits the critical-section.

¤ readcount counts no. of processes currently reading the object.

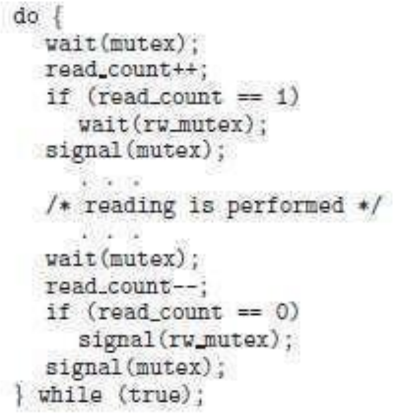
Initialization

mutex = 1, wrt = 1, readcount = 0

Writer Process: **Reader Process:**



```
do {
   wait(rw_mutex);
     . . .
   /* writing is performed */
     . . .
   signal(rw_mutex);
} while (true);
```



```
do {
   wait(mutex);
   read_count++;
   if (read_count == 1)
      wait(rw_mutex);
   signal(mutex);
     . . .
   /* reading is performed */
     . . .
   wait(mutex);
   read_count--;
   if (read_count == 0)
      signal(rw_mutex);
   signal(mutex);
} while (true);
```

- The readers-writers problem and its solutions are used to provide **reader-writer locks** on some systems.
- The mode of lock needs to be specified:

1) read mode

- When a process wishes to read shared-data, it requests the lock in read mode.

2) write mode

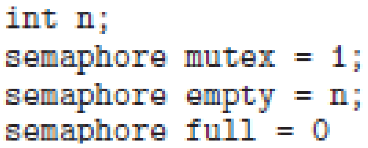
- When a process wishes to modify shared-data, it requests the lock in write mode.
- Multiple processes are permitted to concurrently acquire a lock in read mode, but only one process may acquire the lock for writing.
- These locks are most useful in the following situations:

1) In applications where it is easy to identify
   → which processes only read shared-data and
   → which threads only write shared-data.
2) In applications that have more readers than writers.

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

The Bounded-Buffer Problem

- The bounded-buffer problem is related to the producer consumer problem.
- There is a pool of n buffers, each capable of holding one item.

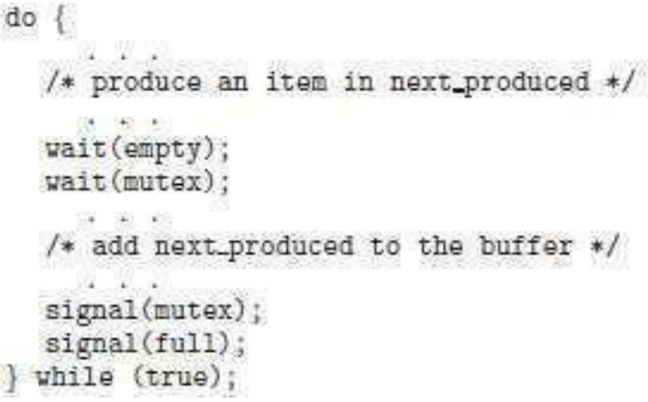


```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0
```

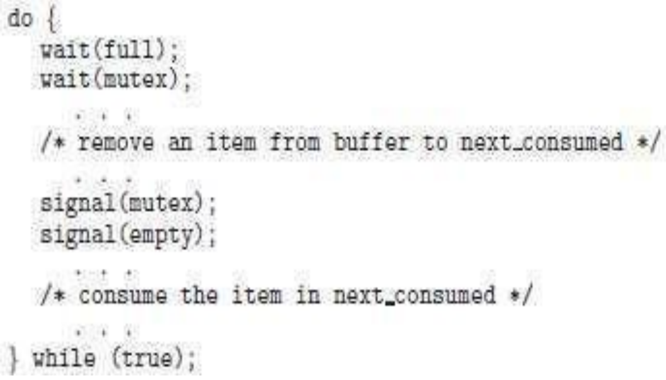
- **Shared-data**

where,
¤ mutex provides mutual-exclusion for accesses to the buffer-pool.
¤ empty counts the number of empty buffers.
¤ full counts the number of full buffers.

- The symmetry between the producer and the consumer.
¤ The producer produces full buffers for the consumer.
¤ The consumer produces empty buffers for the producer.



```
do {
      . . .
   /* produce an item in next_produced */
      . . .
   wait(empty);
   wait(mutex);
      . . .
   /* add next_produced to the buffer */
      . . .
   signal(mutex);
   signal(full);
} while (true);
```



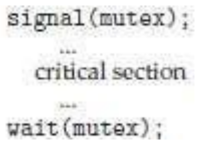
```
do {
   wait(full);
   wait(mutex);
      . . .
   /* remove an item from buffer to next_consumed */
      . . .
   signal(mutex);
   signal(empty);
      . . .
   /* consume the item in next_consumed */
      . . .
} while (true);
```

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

- **Monitor** is a high-level synchronization construct.
- It provides a convenient and effective mechanism for process synchronization.

Need for Monitors

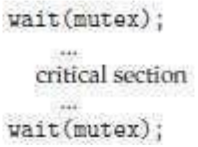
- When programmers use semaphores incorrectly, following types of errors may occur:
  1) Suppose that a process interchanges the order in which the wait() and signal() operationson



```
signal(mutex);
   ...
  critical section
   ...
wait(mutex);
```

the semaphore —mutex‖ are executed, resulting in the following execution:

- In this situation, several processes may be executing in their critical-sections simultaneously, violating the mutual-exclusion requirement.



```
wait(mutex);
   ...
  critical section
   ...
wait(mutex);
```

2) Suppose that a process replaces signal(mutex) with wait(mutex). That is, it executes

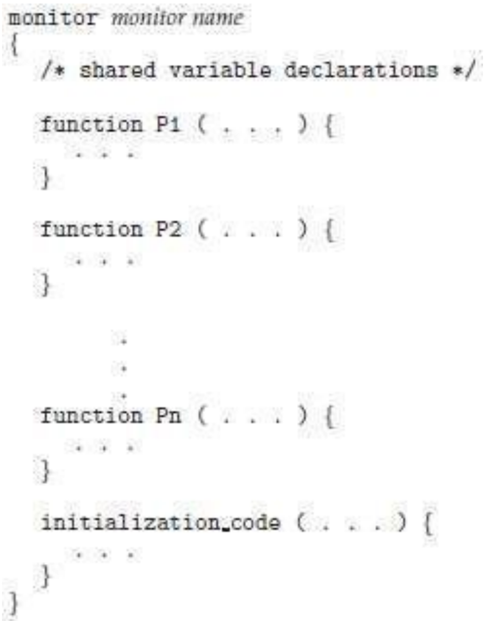
- In this case, a deadlock will occur.

3) Suppose that a process omits the wait(mutex), or the signal(mutex), or both.

- In this case, either mutual-exclusion is violated or a deadlock will occur.

Monitors Usage

- A **monitor type** presents a set of programmer-defined operations that are providedto ensure mutual-exclusion within the monitor.
- It also contains (Figure 2.23):
  - → declaration of variables
  - → bodies of procedures(or functions).
- A procedure defined within a monitor can access only those variables declared locally within the monitor and its formal-parameters.



```
monitor monitor name
{
  /* shared variable declarations */

  function P1 ( . . . ) {
     . . .
  }

  function P2 ( . . . ) {
     . . .
  }
          .
          .
          .
  function Pn ( . . . ) {
     . . .
  }

  initialization_code ( . . . ) {
     . . .
  }
}
```

Similarly, the local-variables of a monitor can be accessed by only the local-procedures.

Figure 2.23 Syntax of a monitor

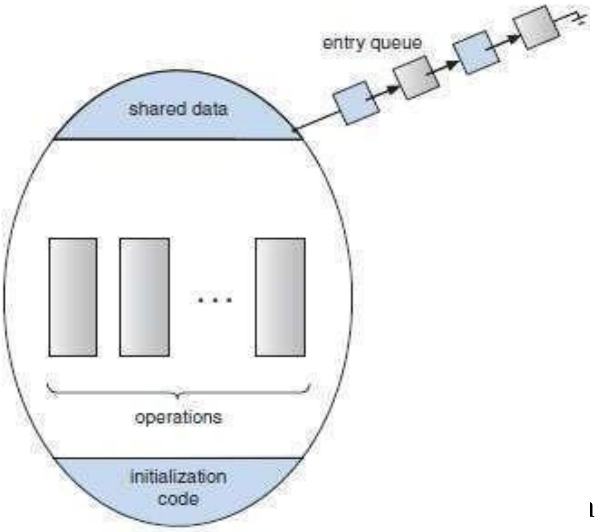
- Only one process at a time is active within the monitor (Figure 2.24).

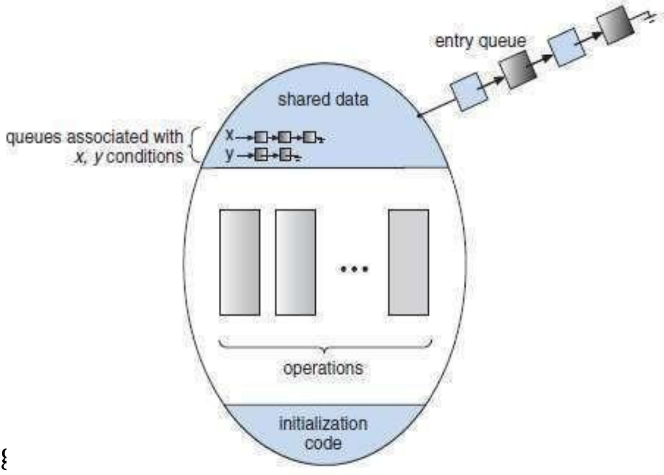
```
condition x, y;
```

- To allow a process to wait within the monitor, a condition variable must be declared, as
- Condition variable can only be used with the following 2 operations (Figure 2.25):

1) **x.signal()**
   - This operation resumes exactly one suspended process. If no process is suspended, then the signal operation has no effect.

2) **x.wait()**
   - The process invoking this operation is suspended





until another process invokesx.signal().

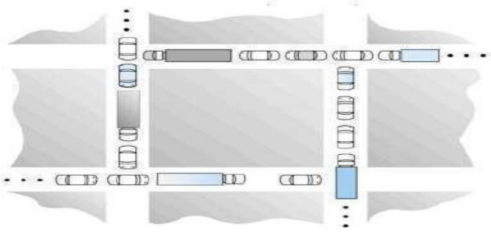
Figure 2.24 Schematic view of a monitor Figure 2.25 Monitor with condition variables

- Suppose when the x.signal() operation is invoked by a process P, there exists a suspended process Q associated with condition x.
- Both processes can conceptually continue with their execution. Two possibilities exist:
  1) **Signal and wait**
     - P either waits until Q leaves the monitor or waits for another condition.
  2) **Signal and continue**
     - Q either waits until P leaves the monitor or waits for another condition.

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

Deadlocks

- Deadlock is a situation where a set of processes are blocked because each process is
  → holding a resource and
  → waiting for another resource held by some other process.
- Real life example:
  When 2 trains are coming toward each other on same track and there is only one track,none of the trains can move once they are in front of each other.
- Similar situation occurs in operating systems when there are two or more processes hold some resources and wait for resources held by other(s).



- Here is an example of a situation where deadlock can occur (Figure 3.1).

Figure 3.1 Deadlock Situation

Deadlock Characterization

- In a deadlock, processes never finish executing, and system resources are tied up,preventing other jobs from starting.

1) **Necessary Conditions**

- There are four conditions that are necessary to achieve deadlock:

i) **Mutual Exclusion**

At least one resource must be held in a non-sharable mode.

i.e., If one process holds a non-sharable resource and if any other process requests this resource, then the requesting-process must wait for the resource to be released.

ii) **Hold and Wait**

□A process must be simultaneously
→ holding at least one resource and
→ waiting to acquire additional resources held by the other process.

iii) **No Preemption**

Resources cannot be preempted.
A resource can be released voluntarily by the process holding it.

iv) **Circular Wait**

A set of processes { P0, P1, P2, . . ., PN } mustexist
P0 is waiting for a resource that is held by P1, P1 is waiting for a resource that is heldby P2 ….and PN is waiting for a resource held by P0.

7. Explain how resource allocation graph can be used find deadlocks with examples.

Deadlock Detection

- If a system does not use deadlock-prevention or deadlock-avoidance algorithmthen a deadlock may occur.
- In this environment, the system must provide
  1) An algorithm to examine the system-state to determine whether a deadlock has

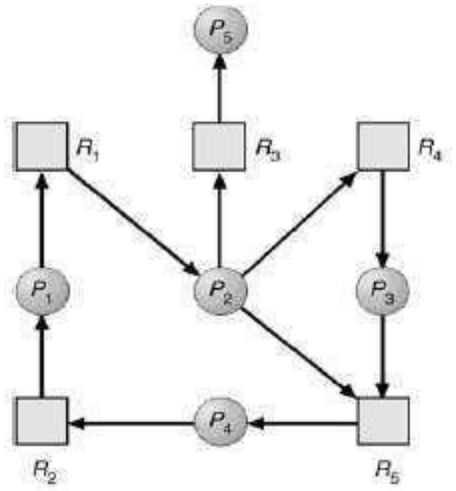
     occurred.
  2) An algorithm to recover from the deadlock.

1) Single Instance of Each Resource Type

- If all the resources have only a single instance, then deadlock detection-algorithm can be defined using a wait-for-graph.
- The wait-for-graph is applicable to only a single instance of a resource type.
- A wait-for-graph (WAG) is a variation of the resource-allocation-graph.
- The wait-for-graph can be obtained from the resource-allocation-graph by
  → removing the resource nodes and
  → collapsing the appropriate edges.

- An edge from P_i to P_j implies that process P_i is waiting for process P_j to release a resource that P_i needs.
- An edge $P_i \rightarrow P_j$ exists if and only if the corresponding graph contains two edges
  1) $P_i \rightarrow R_q$ and
  2) $R_q \rightarrow P_j$.
- For example:

  Consider resource-allocation-graph shown in Figure 3.6 Corresponding wait-for-graph is shown in Figure 3.7.



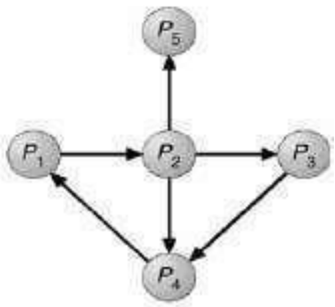


Figure 3.6 Resource-allocation-graph Figure 3.7 Corresponding wait-for-graph.

- A deadlock exists in the system if and only if the wait-for-graph contains a cycle.
- To detect deadlocks, the system needs to

  → maintain the wait-for-graph and

  → periodically execute an algorithm that searches for a cycle in the graph.

2) Several Instances of a Resource Type

- The wait-for-graph is applicable to only a single instance of a resource type.
- Problem: However, the wait-for-graph is not applicable to a multiple instance of a resource type.
- Solution: The following detection-algorithm can be used for a multiple instance of a resource type.
- Assumptions:

  Let 'n' be the number of processes in the system Let 'm' be the number of resources types.

- Following data structures are used to implement this algorithm.
  1) **Available [m]**
     i. This vector indicates the no. of available resources of each type.
     ii. If Available[j]=k, then k instances of resource type Rj is available.
  2) **Allocation [n][m]**
     - This matrix indicates no. of resources currently allocated to each process.
     - If Allocation[i,j]=k, then Pi is currently allocated k instances of Rj.
  3) **Request [n][m]**
     i. This matrix indicates the current request of each process.
     ii. If Request [i, j] = k, then process Pi is requesting k more instances of resource type Rj.

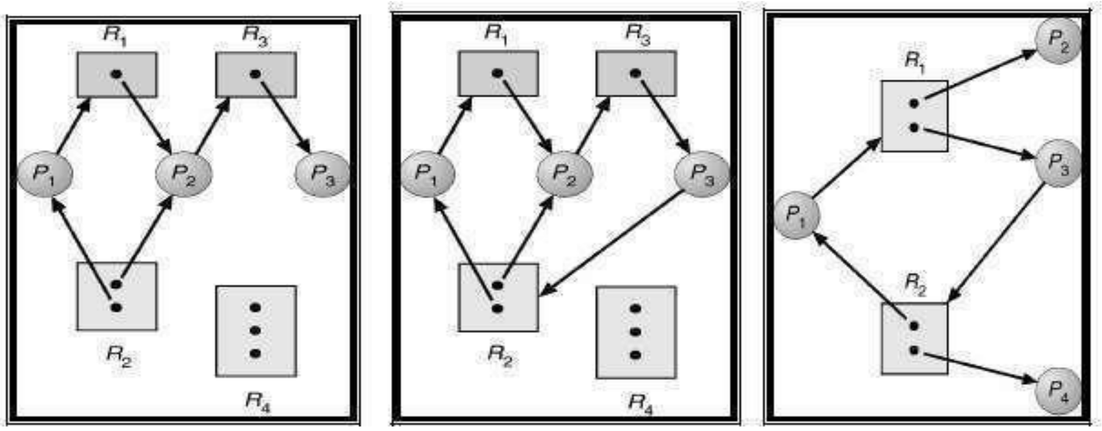
This algorithm requires an order mxn^2 operations to detect whether the system is in a deadlocked state

8. With neat diagrams explain Resource Allocation graph.

Resource-Allocation-Graph

- The resource-allocation-graph (RAG) is a directed graph that can be used to describethe deadlock situation.
- RAG consists of a
  → set of vertices (V) and
  → set of edges (E).
- V is divided into two types of nodes
  ii) P={P1,P2… Pn} i.e., set consisting of all active processes in the system.
  iii) R={R1,R2… Rn} i.e., set consisting of all resource types in the system.
- E is divided into two types of edges:
  1) **Request Edge**
  □A directed-edge $P_i \rightarrow R_j$ is called a request edge.
  □$P_i \rightarrow R_j$ indicates that process P_i has requested a resource R_j.
  2) **Assignment Edge**
  □A directed-edge $R_j \rightarrow P_i$ is called an assignment edge.
  □$R_j \rightarrow P_i$ indicates that a resource R_j has been allocated to process P_i.
- Suppose that process Pi requests resource Rj.
  Here, the request for Rj from Pi can be granted only if the converting request-edge to assignment-edge do not form a cycle in the resource-allocation graph.

- Pictorially,
  - → We represent each process P_i as a **circle**.
  - → We represent each resource-type R_j as a **rectangle**.
- As shown in below figures, the RAG illustrates the following 3 situation (Figure 3.3):
  1) RAG with a deadlock
  2) RAG with a cycle and deadlock
  3) RAG with a cycle but no deadlock



(1) **Resource allocation Graph (2) with a deadlock (3) with cycle but no deadlock**

Figure 3.3 Resource allocation graphs

1) If a graph contains no cycles, then the system is not deadlocked.
2) If the graph contains a cycle then a deadlock may exist.

Therefore, a cycle means deadlock is possible, but not necessarily present.

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

1) Banker's Algorithm

- 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. (A banker won't loan out a little money to start building a house unless they are assuredthat they will later be able to loan out the rest of the money to finish the house.)
- 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 canbe granted

safely.

- If the system in a safe state,
  the resources are allocated;
  else
  the process must wait until some other process releases enough resources.
- Assumptions:
  Let n = number of processes in the system Let m = number of resources types.
- Following data structures are used to implement the banker's algorithm.
  i. **Available [m]**
    - This vector indicates the no. of available resources of each type.
    - If Available[j] =k, then k instances of resource type Rj is available.
  ii. **Max [n][m]**
    - This matrix indicates the maximum demand of each process of each resource.
    - If Max[i,j]=k, then process Pi may request at most k instances of resourcetype Rj.
  iii. **Allocation [n][m]**
    - This matrix indicates no. of resources currently allocated to each process.
    - If Allocation[i,j]=k, then Pi is currently allocated k instances of Rj.
  iv. **Need [n][m]**
    - This matrix indicates the remaining resources need of each process.
    - If Need [i,j]=k, then Pi may need k more instances of resource Rj to completeits task.
    - So, Need[i,j] = Max[i,j] - Allocation[i]
- The Banker's algorithm has two parts:
  - **Safety Algorithm**
  - **Resource – Request Algorithm**

i. Safety Algorithm

- This algorithm is used for finding out whether a system is in safe state or not.
- Assumptions:
  Work is a working copy of the available resources, which will be modified during the analysis. Finish is a vector of boolean values indicating whether a particular process can

finish.

Request **= request vector for process P_i. If $Request_i[j] = k$ then process P_i wants k instances of resource type R_j**

1. If $Request_i \leq Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim

2. If $Request_i \leq Available$, go to step 3. Otherwise P_i must wait, since resources are not available

3. <u>Pretend to allocate</u> requested resources to P_i by modifying the state as follows:

Available = Available – Request;
$Allocation_i = Allocation_i + Request_i$;
$Need_i = Need_i – Request_i$;

- *If safe ⇒ the resources are allocated to Pi*
- *If unsafe ⇒ Pi must wait, and the old resource-allocation state is restored*

Safety Algorithm:

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*
(b) *$Need_i \leq Work$*
If no such *i* exists, go to step 4
3. Work = Work + $Allocation_i$
Finish*[*i*] = *true
go to step 2
4. If *Finish* [*i*] == true for all *i*, then the system is in a safe state

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

<u>Deadlock-Prevention:</u>

- Deadlocks can be eliminated by preventing at least one of the four required conditions:
  1) Mutual exclusion
  2) Hold-and-wait
  3) No preemption
  4) Circular-wait.

1) **Mutual Exclusion**

- This condition must hold for non-sharable resources.
- For example:

  A printer cannot be simultaneously shared by several processes.
- On the other hand, shared resources do not lead to deadlocks.
- For example:

  Simultaneous access can be granted for read-only file.
- A process never waits for accessing a sharable resource.

□In general, we cannot prevent deadlocks by denying the mutual-exclusion condition because some resources are non-sharable by default.

2) **Hold and Wait**

- To prevent this condition, ensure that – Whenever a process requests a resource, itdoes not hold any other resources.
- There are several solutions to this problem.
- For example:

  Consider a process that

  → copies the data from a tape drive to the disk

  → sorts the file and

  → then prints the results to a printer.

  Protocol-1

  - Each process must be allocated with all of its resources before it beginsexecution.
  - All the resources (tape drive, disk files and printer) are allocated to the process at the beginning.

  Protocol-2

  □ A process must request a resource only when the process has none.

  □ Initially, the process is allocated with tape drive and disk file.

  □ The process performs the required operation and releases both tape drive and diskfile.

  □ Then, the process is again allocated with disk file and the printer

  □ Again, the process performs the required operation & releases both disk file and theprinter.

- **Disadvantages** of above 2 methods:

  i. Resource utilization may be low, since resources may be allocated but unusedfor a long period.

  ii. Starvation is possible.

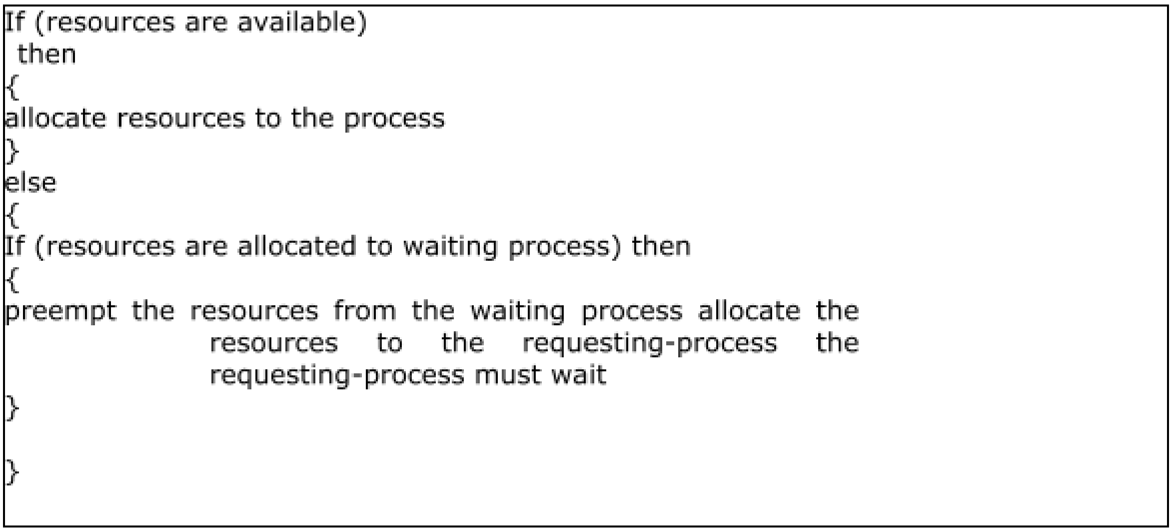
3) **No Preemption:**

- To prevent this condition: the resources must be preempted.
- There are several solutions to this problem.

Protocol-1

- If a process is holding some resources and requests another resource that cannot be immediately allocated to it, then all resources currently being held are preempted.
- The preempted resources are added to the list of resources for which the processis waiting.
- The process will be restarted only when it regains the old resources and the new resources that it is requesting.

Protocol-2



```
If (resources are available)
 then
{
allocate resources to the process
}
else
{
If (resources are allocated to waiting process) then
{
preempt the resources from the waiting process allocate the
                resources to the requesting-process the
                requesting-process must wait
}

}
```

- When a process request resources, we check whether they are available or not.

These 2 protocols may be applicable for resources, whose states are easily saved and restored, such as registers and memory.

4) Circular-Wait

- Deadlock can be prevented by using the following 2 protocol:

  Protocol-1

  - Assign numbers all resources.
  - Require the processes to request resources only in increasing/decreasing order.

  Protocol-2

  - Require that whenever a process requests a resource, it has releasedresources with a lower number.
- One big challenge in this scheme is determining the

relative ordering of the differentresources.

Side effects of preventing deadlocks:

- Low device utilization
- Reduced system throughput