###### Unit IV

**Part-A(Process Synchronization)**

1. The critical section problem.

2. Peterson’s Solution

3. Synchronization Hardware

4. Semaphores

5. Classic problems of synchronization.

6. Monitors

**Part-B(Principles of Deadlocks)**

1. System Model

2. Deadlock Characterization

3. Deadlock Handling

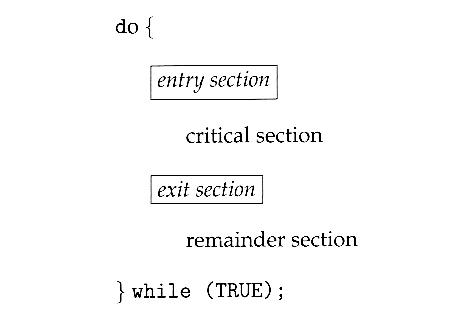
4. Deadlock Prevention, Detection and Avoidance

5. Recovery & Starvation

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

### **1. The Critical-Section Problem**

1. If a system consist of n processes {Po, P 1 , ... , Pn}. Each process has a segment of code,
2. called a critical section., where the process may change common variables, updating a
3. table, writing a file, and so on.
4. When one process is executing in its critical section, no other process is to be allowed to
5. execute in its critical section. That is, no two processes are executing in their critical
6. sections at the same time.
7. The critical-section problem is to design a protocol that the processes can use to
8. cooperate. Each process must request permission to enter its critical Section.
9. The section of code implementing this request is the Entry-Section.
10. The critical section may be followed by an Exit-Section and the remaining code is
11. Remainder-Section.

  
General structure of a typical process Pi

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

**1. Mutual Exclusion-** Only one process at a time can be executing in their critical section.

**2. Progress -** If no process is currently executing in their critical section, and one or more processes want to execute their critical section, then only the processes not in their remainder sections can participate.

**3. Bounded Waiting -** There exists a bound, or limit, 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.

### **2. Peterson's Solution**

Peterson's Solution is a classic software-based solution to the critical section problem.

Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections.

The processes are numbered Pi and Pj .

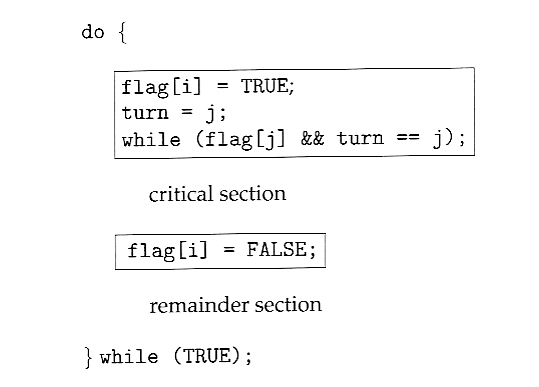
Peterson's solution requires the two processes to share two data items:

int turn;

boolean flag[2];

The variable turn indicates whose turn it is to enter its critical section. That is, if turn == i, then process Pi is allowed to execute in its critical section. The flag array is used to indicate if a process is ready to enter its critical section.

For example, if flag [i] is true, this value indicates that Pi is ready to enter its critical section.

  
The structure of process Pi in Peterson's solution.

To prove that this solution is correct. We need to show that,

a. Mutual exclusion is preserved.

b. The progress requirement is satisfied.

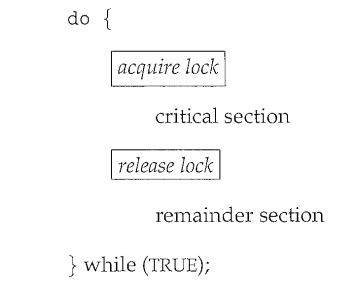
c. The bounded-waiting requirement is met.

**3. Synchronization Hardware**

It is a Hardware based solution for synchronization.

The soloution to critical section problem is done by using Locks.

a process must acquire a lock before entering a critical section and releases the lock when it exits the critical section.



**4. Semaphores**

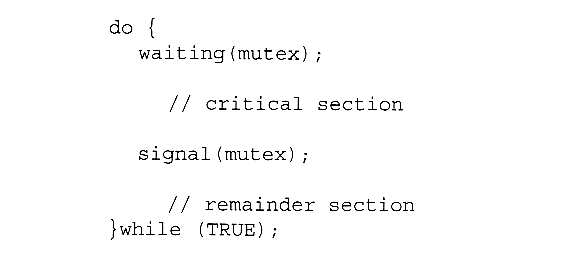
A Semaphore is a synchronization primitive which uses mutexes. These mutexes are integer variables that are used to solve the critical section problem by using two atomic operations, wait and signal that are used for process synchronization.

**Implementation :**

To overcome the problem of busy waiting, when a process is in its critical section and any other process that tries to enter into its Critical section has to loop continuously. It has to acquire a lock which is called spin lock.

when a process is waiting in a semaphore S, it should be restarted when some other process executes a signal() and that process will be in ready queue for execution.

Mutual exclusion implementation for semaphores is given below,



The definitions of wait and signal are as follows −

Wait :

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

wait(S)

{

while (S<=0); //No Operation

S--;

}

Signal :

The signal operation increments the value of its argument S.

signal(S)

{

S++;

}

## **Types of Semaphores :**

There are two main types of semaphores i.e. counting semaphores and binary semaphores.

Counting Semaphores :

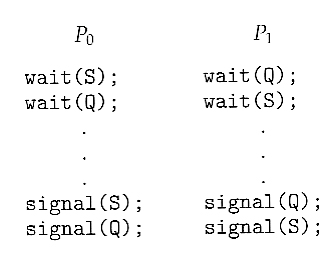
These are integer value semaphores and have an unrestricted value domain. These semaphores are used to coordinate the resource access, where the semaphore count is the number of available resources. If the resources are added, semaphore count automatically incremented and if the resources are removed, the count is decremented. ie., whenever the counting semaphore is greater than zero, then a process can enter a critical section and use one of the resources. When the counter gets to zero ( or negative), then the process blocks until another process frees up a resource and increments the counting semaphore with a signal call.

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.

#### **Deadlocks and Starvation using semaphores:**

One important problem that can arise when using semaphores to block processes waiting for a limited resource is the problem of *deadlocks*, which occur when multiple processes are blocked, each waiting for a resource that can only be freed by one of the other ( blocked ) processes, as mentioned below,



Another problem to consider is that of *starvation*, in which one or more processes gets blocked forever, and never get a chance to take their turn in the critical section. For example, in the semaphores above, we did not specify the algorithms for adding processes to the waiting queue in the semaphore in the wait() call, or selecting one to be removed from the queue in the signal( ) call. If the method chosen is a FIFO queue, then every process will eventually get their turn, but if a LIFO queue is implemented instead, then the first process to start waiting could starve.

**5. Classic problems of Synchronization**

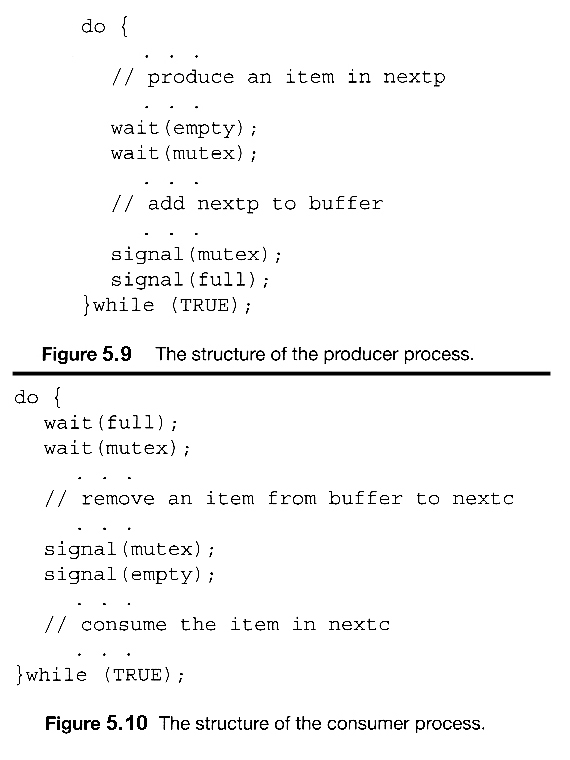
The following classic problems are used to test Synchronization algorithm.

#### **1. The Bounded-Buffer Problem**

This is a generalization of the producer-consumer problem wherein access is controlled to a shared group of buffers of a limited size.

In this solution, the two counting semaphores "full" and "empty" keep track of the current number of full and empty buffers respectively ( and initialized to 0 and N respectively. )

The binary semaphore mutex controls access to the critical section. The producer and consumer processes are nearly identical - One can think of the producer as producing full buffers, and the consumer producing empty buffers.



#### **2. The Readers-Writers Problem**

In the readers-writers problem there are some processes ( termed readers ) who only read the shared data, and never change it, and there are other processes ( termed writers ) who may change the data in addition to or instead of reading it.

There is no limit to how many readers can access the data simultaneously, but when a writer accesses the data, it needs exclusive access.

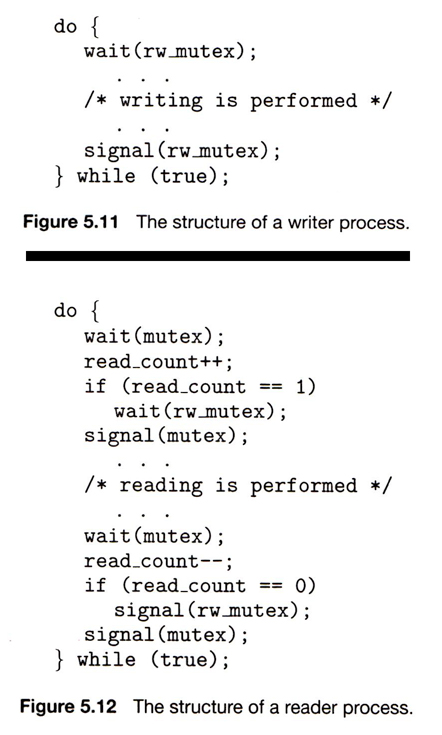
There are several variations to the readers-writers problem, like *first* readers-writers problem which gives priority to readers and The *second* readers-writers problem which gives priority to the writers.

The following code is an example of the first readers-writers problem, and involves an important counter and two binary semaphores:

readcount : It is used by the reader processes, to count the number of readers currently accessing the data.

Mutex : It is a semaphore used only by the readers for controlled access to readcount.

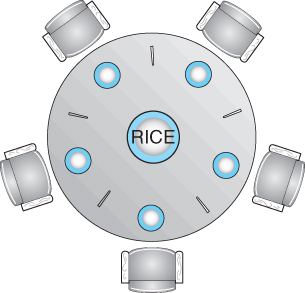
rw\_mutex : is a semaphore used to block and release the writers.



#### **3. The Dining-Philosophers Problem**

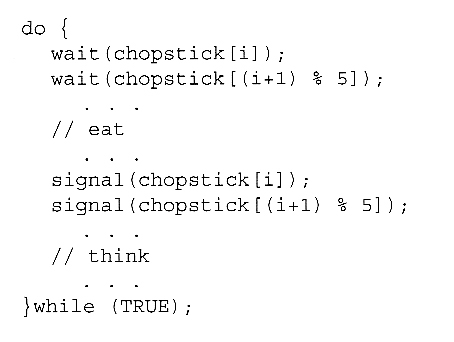
The dining philosophers problem is a classic synchronization problem involving the allocation of limited resources among a group of processes in a deadlock-free and starvation-free manner:

* + Consider five philosophers sitting around a table, in which there are five chopsticks evenly distributed and an endless bowl of rice in the center. (There is exactly one chopstick between each pair of dining philosophers.)
  + These philosophers spend their lives alternating between two activities: eating and thinking.
  + When it is time for a philosopher to eat, it must first acquire two chopsticks - one from their left and one from their right.
  + When a philosopher thinks, it puts down both chopsticks in their original locations.

  
The situation of the dining philosophers

One possible solution, as shown in the following code section, is to use a set of five semaphores ( chopsticks[ 5 ] ), and to have each hungry philosopher first wait on their left chopstick ( chopsticks[ i ] ), and then wait on their right chopstick ( chopsticks[ ( i + 1 ) % 5 ] )

But suppose that all five philosophers get hungry at the same time, and each starts by picking up their left chopstick. They then look for their right chopstick, but because it is unavailable, they wait for it, forever, and eventually all the philosophers starve due to the resulting deadlock.

  
The structure of philosopher i.

Some potential solutions to the problem include:

* + Only allow four philosophers to dine at the same time. ( Limited simultaneous processes. )
  + Allow philosophers to pick up chopsticks only when both are available, in a critical section.
  + Use an asymmetric solution, in which odd philosophers pick up their left chopstick first and even philosophers pick up their right chopstick first.

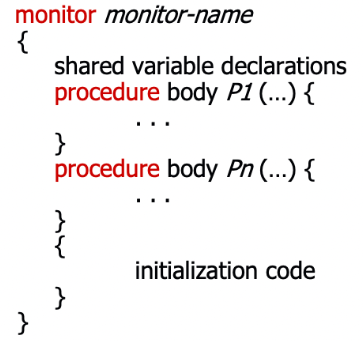
**6. Monitors**

Monitors are a synchronization construct that were created to overcome the problems caused by semaphores such as timing errors.

Monitors are abstract data types and contain shared data variables and procedures. The shared data variables cannot be directly accessed by a process and procedures are required to allow a single process to access the shared data variables at a time.

Only one process can be active in a monitor at a time. Other processes that need to access the shared variables in a monitor have to line up in a queue and are only provided access when the previous process release the shared variables.

Syntax :



**Part-B(Principles of Deadlocks)**

1. System Model

2. Deadlock Characterization

3. Deadlock Handling

4. Deadlock Prevention, Avoidance & Detection

5. Recovery & Starvation

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

### **1. System Model**

* A system can be modeled as a collection of limited resources, which can be partitioned into different categories, to be allocated to a number of processes with each having different needs.
* Resource categories may include memory, printers, CPUs, open files, tape drives, CD-ROMS, etc.
* All the resources within a category are equivalent, and a request of this category can be equally satisfied by any one of the resources in that category. If this is not the case, then that category needs to be further divided into separate categories.
* In normal operation a process must request a resource before using it, and release it when it is done, in the following sequence:

**1. Request -** If the request cannot be immediately granted, then the process must wait until the resource(s) it needs become available.

Eg: System calls (open(), malloc(), new(), and request())

**2. Use** - The process uses the resource,

E.g. prints to the printer or reads from the file.

**3. Release -** The process releases the resource. so that it becomes available for other processes.

Eg:- System calls (close(), free(), delete(), and release())

* A set of processes is deadlocked when every process in the set is waiting for a resource that is currently allocated to another process in the set.

**2. Deadlock Characterization**

In a deadlock, the process never finish its execution where system resources are tied and will not let the other process to start.

The various characterization of deadlocks are,

#### **a) Necessary Conditions**

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

**1. Mutual Exclusion -** At least one resource must be held in a non-sharable mode; If any other process requests this resource, then that process must wait for the resource to be released.

**2. Hold and Wait -** A process must be simultaneously holding at least one resource and waiting for at least one resource that is currently being held by some other process.

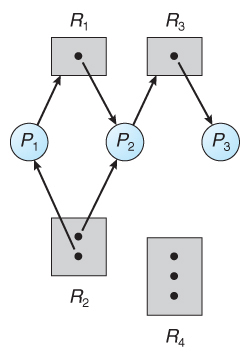
**3. No preemption -** Once a process is holding a resource ( i.e. once its request has been granted ), then that resource cannot be taken away from that process until the process voluntarily releases it.

**4. Circular Wait -** A set of processes { P0, P1, P2, . . ., PN } must exist such that every P[ i ] is waiting for P[ ( i + 1 ) % ( N + 1 ) ].

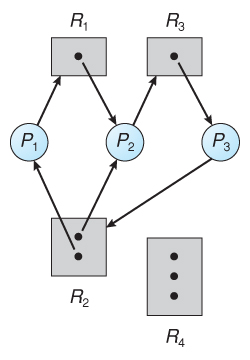
#### **b) Resource-Allocation Graph**

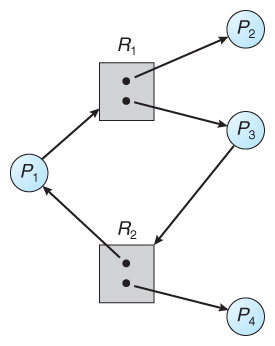
* Resource-Allocation Graphs, having the following properties:
  + A set of resource categories, { R1, R2, R3, . . ., RN }, which appear as square nodes on the graph. Dots inside the resource nodes indicate specific instances of the resource. ( E.g. two dots might represent two laser printers. )
  + A set of processes, { P1, P2, P3, . . ., PN }
  + Request Edges -A set of directed arcs from Pi to Rj, indicating that process Pi has requested Rj, and is currently waiting for that resource to become available.
  + Assignment Edges -A set of directed arcs from Rj to Pi indicating that resource Rj has been allocated to process Pi, and that Pi is currently holding resource Rj.
  + A request edge can be converted into an assignment edge by reversing the direction of the arc when the request is granted.

Example:

  
Resource allocation graph

* If a resource-allocation graph contains no cycles, then the system is not deadlocked.
* If a resource-allocation graph does contain cycles AND each resource category contains only a single instance, then a deadlock exists.
* If a resource category contains more than one instance, then the presence of a cycle in the resource-allocation graph indicates the *possibility* of a deadlock, but does not guarantee one.

  
Resource allocation graph with a deadlock

  
Resource allocation graph with a cycle but no deadlock

**3. Deadlock Handling**

A deadlock can be handled in 3 ways, they are

a. Deadlock Prevention

b. Deadlock Avoidance and

c. Deadlock Detection & Recovery

**4. Deadlock Prevention, Avoidance and Detection**

### **Deadlock Prevention :**

Deadlock prevention is concerned about how the requests are to be made.

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

#### a) 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.

#### b) 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.

#### c) No Preemption

* Preemption of process resource allocations can prevent this condition of deadlocks, when it is possible.
* 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 re-acquire the old resources along with the new resources in a single request, similar to the previous discussion.

#### d) Circular Wait

* 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.
* In other words, in order to request resource Rj, a process must first release all Ri such that i >= j.

### **Deadlock Avoidance :**

Deadlock Avoidance is an alternate method to deadlock prevention where we need to know more information about how resources are to be requested.

A deadlock avoidance algorithm dynamically examines the resource allocation state.

A state is safe if the system allocates resources to each process in some order without deadlock.

A system is in safe state if there exists a safe sequence.

A deadlock avoidance procedures involves in

a. Safe state

b. Resource Allocation Graph and

c. Banker's Algorithm

**Banker's Algorithm**

**Procedure :**

When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number of resources in the system.

When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated else, the process must wait until some other process releases enough resources.

**Data structures:**

Suppose n is the number of processes in the system and m is the number of resource types.

**Available :** A vector of length m indicates the number of available resources of each type. If Available[j] equals k, then k instances of resource type Rj are available.

**Max :** An n x m matrix defines the maximum demand of each process. If Max[i] [j] equals k, then process Pi may request at most k instances of resource type Rj.

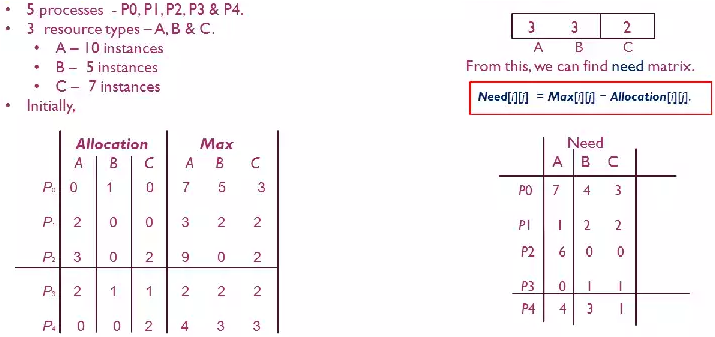
**Allocation :** An n x m matrix defines the number of resources of each type currently allocated to each process. If Allocation[i][j] equals k, then process Pi is currently allocated k instances of resource type Rj.

**Need :** An n x m matrix indicates the remaining resource need of each process. If Need[i][j] equals k, then process Pi may need k more instances of resource type Rj to complete its task.

Note : Need[i][j]==Max[i][j] - Allocation [i][j].

These data structures vary over time in both size and value.

**Example :**

Safe Sequence : <P1,P3,P4,P0,P2>

**5. Recovery & Starvation**

**Recovery :**

When a detection algorithm determines that a deadlock exists, several alternatives are available.

One possibility is to inform the operator that a deadlock has occurred and to let the operator deal with the deadlock manually.

Another possibility is to let the system recover from the deadlock automatically.

There are two options for breaking a deadlock,

One is simply to abort one or more processes to break the circular wait and the other is to preempt some resources from one or more of the deadlocked processes.

**a. Process Termination**

To eliminate deadlocks by aborting a process, we use one of two methods.

1. Abort all deadlocked processes.

2. Abort one process at a time until the deadlock cycle is eliminated.

**b. Resource Preemption**

To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken.

If preemption is required to deal with deadlocks, then three issues need are

**1. Selecting a victim :**

It involves in which resources and which processes are to be preempted. As in process

termination, we must determine the order of preemption to minimize cost which may

include the number of resources and the amount of time as parameters.

2**. Rollback :**

If we preempt a resource from a process, it cannot continue with its normal execution.

It is missing some needed resource. We must roll back the process to some safe state and

restart it from that state.

**3. Starvation :**

In a system where victim selection is based primarily on cost factors, it may happen that

the same process is always picked as a victim. As a result, this process never completes its

designated task which leads to a starvation situation. We must ensure that a process can

be picked as a victim" only a (small) finite number of times.

The most common solution is to include the number of rollbacks in the cost factor.