Solution to Critical Section Problem :

(1.) **Mutual Exclusion -**

If one process is executing in its critical section, then no other process can be executing in their critical sections.

(2.) **Progress -**

If some processes wish to enter their critical section then the selection of the processes can not be postponed indefinitely.

(3.) **Bounded waiting -**

A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter that request is granted.

(3.) **Bounded waiting -**

A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter that request is granted

Two Process Solutions

→ Algorithms are restricted to only tw

until false.

→ Ensures Mutual Exclusion but does not satisfy progress requirement.

Algorithm 2 :

where flag [j] = true indicates that process j

ready to enter its critical section.

```
repeat
    flag [i] = true;
    while flag [j] do no-op;

        critical section

    flag [i] = false;
                remainder section
until false;
```

structure of process P_i

→ Mutual Exclusion is satisfied.

→ Progress requirement is still not satisfied.

Algorithm 3:

```
repeat

    flag [i] = true;
    turn = j;
    while ( flag [j] and turn ==j) do no_op;
            /* Asserts that it is other process turn to
               enter its critical section */

    critical section

    flag [i] = false;

    remainder section

until false;
```

→ Mutual Exclusion is preserved.

→ Progress requirement is satisfied.

→ Bounded waiting requirement is met.

Synchronization Hardware

→ Two hardware instructions :

- Test and set
- Swap

are used to solve critical section problem.

→ These instructions are atomic ⇒ non- interruptible.

TestAndSet Instruction :

→ Tests and sets a memory address.

→ Code is

```
boolean TestAndSet ( boolean * target)
 {
   boolean rv = * target;
   * target = TRUE;
     return rv;
 }
```

Solution using TestAndset

→ Shared boolean variable lock is used, which is set to false initially.

```
do
{
  while (TestAndSet (&lock))      // Entry section
            ;     // do nothing
    critical section
    lock = FALSE;                 // Exit section
    remainder section
} while (TRUE);
```

→ Mutual Exclusion & Progress requirement is satisfied.

→ Bounded waiting may not be satisfied.

Swap Instruction:

→ swaps content of two words atomically.

```
void swap ( boolean *a, boolean *b)
  {
     boolean temp = *a;
                *a = *b;
                *b = temp;
  }
```

Solution using Swap:

→ Shared boolean variable lock is set to FALSE.

→ Each process has a local boolean variable Key.

```
do {
      Key = TRUE;
      while (Key == TRUE)                // Entry section
          swap ( &lock, &Key);
          critical section
          lock = FALSE;                  // Exit section
          remainder section
   } while (TRUE);
```

```
    j = (i+1) mod n;
   while((j ≠ i) and (not waiting[j]))              // Exit section
           j=(j+1) mod n;
   if (j==i)    lock = false;
           else  waiting[j] = FALSE;

     remainder  section
}
 while (TRUE);
```

→ Mutual Exclusion is met.

→ Progress requirement is met –

as each process after critical section either sets lock to false or waiting [j] to false.

→ Bounded waiting is also met.

- → Any process waiting to enter critical section will do so within (n-1) turns.

→ Mutual Exclusion & progress requirement is satisfied.

→ Bounded waiting is not satisfied.

Algorithm satisfying all conditions :

→ Shared data(s) are : lock → a boolean variable.
waiting [n] → a boolean array.

All values are initialized to FALSE.

→ A local variable key is used.

```
int j;

do
 {
    waiting [i] = TRUE;
      key = TRUE;
    while (waiting [i] and key)
          key = TestAndSet (lock);
    waiting [i] = false;
```

// ~~exit~~ Entry section

```
    critical section
```

Semaphores :

→ An integer variable denoted by S.

→ Accessed by two atomic operations:

- Wait (S)

```
{
    while S ≤ 0 ;
            S--;
}
```

- signal (S)

```
{
    S++;
}
```

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

Usage :

(1.) To deal with the n-process critical section problem

- Each process share a semaphore - mutex (~~Mutat~~ Mutual Exclusion) initialized to 1.

- Each process P_i is organized as:

```
do
 {
   wait( mutex);

   critical section

   signal ( mutex);
       remainder section
 } while (TRUE);
```

(2.) Can be used for other synchronization problem.

- Two processes P_1 with a statement S1
  &
  P_2 with a statement S2.

  we require S2 to be executed only after S1.

  A semaphore S, initialized to 0 is shared by two processes as follows:

```
  ....
  ....
  S1;              }
  signal (S);      }  P1
  ....             }
  ....             }


  ....
  wait (S);        }  P2
  S2;              }
  ....             }
```

Semaphore Implementation

→ Main disadvantage of se Mutual exclusive solution using semaphore is - busy waiting.

→ Such semaphore is also called - spinlock - as process spins while waiting for the lock.

→ To avoid busy waiting -

Each semaphore is associated with a waiting queue, as well as a value associated with it.

→ Two operations:

Block - places the process invoking the operation on the waiting queue.

Wakeup - remove one of the process in the waiting queue & place it in the ready queue.

→ Semaphore operations wait (S) & signal (S) are now changed.

```
wait(s):
        s.value = s.value - 1;
        if (s.value < 0)
          {
             add this process into the waiting
             queue of the semaphore;
             block;
          }

signal(s):
        s.value = s.value + 1;
        if (s.value <= 0)
          {
             remove a process P from waiting
             queue of the semaphore;
             wakeup;
          }
```

→ classical definition of semaphore with busy waiting can has only zero or positive value.

→ Above implementation may have -ive values as well.

→ If value is -ive, magnitude gives no. of processes in the waiting queue.

Deadlock & starvation

Deadlock: When two or more processes are waiting indefinitely for an event that can be caused by one of the waiting processes.

P_0	P_1
wait (S);	wait (Q);
wait (Q);	wait (S);
⋮	⋮
signal (S);	signal (Q);
signal (Q);	signal (S);

Two processes P_0 & P_1, each accessing semaphores S and Q, which are set the value of 1.

→ P_0 & P_1 are in deadlock state.

Starvation:
- A situation when process wait indefinitely for accessing critical section.
- May occur if we add & remove processes from the list in semaphore using LIFO order.

Binary Semaphore :

→ semaphore described so far is commonly known as Counting semaphore.

→ Their values can range over an unrestricted domain.

→ Binary semaphore can has integer values in the range 0 to 1 only.

→ Simpler to implement.

Deadlock

→ ~~If~~ A set of processes is in deadlock state if every process in the set is waiting for an event (or resource) that can be caused by only another process in the set.

→ Under normal mode of operation, each process utilise a resource in only following order -

a.) Request - resource is requested. If resource can not be granted immediately, process must wait until it can acquire the resource.

b.) Use - Process operate on the resource.

c.) Release - Process releases the resource after use.

Deadlock characterization

→ For a deadlock to exist, four conditions hold simultaneously.

1.) Mutual Exclusion : At least one resource must be held in non sharable mode.

2.) Hold & Wait : Must exist a process that is holding at least one resource and is waiting to acquire additional resources that are currently held by other processes.

3.) No Preemption : Resources can not be preempt i.e. it can be released volun- ~~tearly~~ tarily by the process once it has completed its task.

4.) Circular wait : There must exist a set $\{P_0, \cdots P_n\}$ of waiting processes such that P_0 is waiting for a resource held by P_1, P_1 is waiting for $P_2 \ldots$ P_n is waiting for P_0.

Resource Allocation Graph

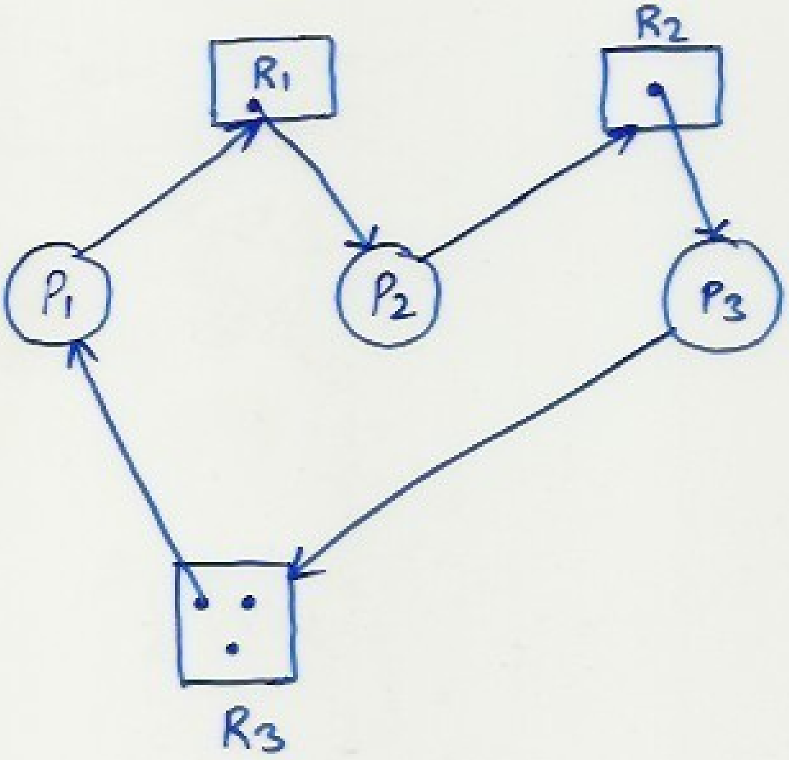
→ A set of vertices V and a set of edges E.

→ V is partitioned into two types-

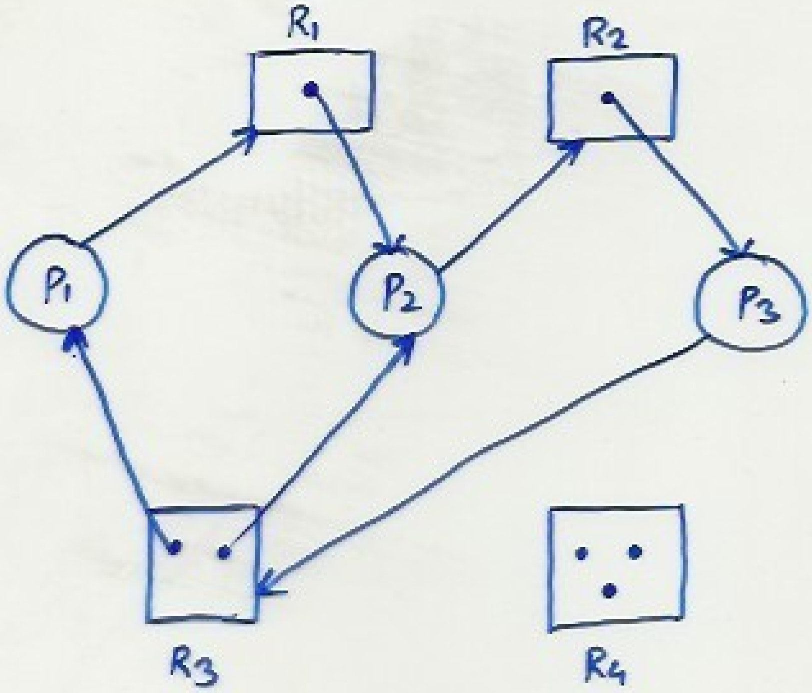
- P = {$P_1, P_2, \ldots P_n$}, the set consisting of all the processes in the system.
- R = {$R_1, \ldots\ldots R_m$}, the set consisting of all resource types in the system.

→ Request edge → a directed edge $P_i \rightarrow R_j$

→ Assignment edge → a directed edge $R_j \rightarrow P_k$



Resource Allocation Graph with Deadlock



Processes {P1, P2 & P3} are in deadlock stat.

→ If graph contains no cycle, ⇒ no deadlock.

→ If graph contains a cycle ⇒

- If only one instance per resource, then deadlock.
- If several instances per resource type, then possibility of deadlock.