PROCESS SYNCHRONIZATION

→ Concurrent access to shared data may result in data inconsistency. Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

- Process Synchronization means sharing system resources by processes in such a way that concurrent access to shared data is handled thereby minimizing the chance of inconsistent data. Maintaining data consistency demands mechanisms to ensure synchronized execution of co-operating processes. Process Synchronization was introduced to handle problems that arise while multiple process executions.

ex:- Instructions of co-operating processes can be interleaved arbitrarily. Hence the order of instructions is irrelevant.

Consider two co-operating processes (sharing a variable 'counter') and having the following set of instructions in each of them

Process 1. (Count++)
register1 = counter
register1 = register1 + 1
counter = register1

Process 2 (Count--)
register2 = counter
register2 = register2 - 1
counter = register2

where register1 & register2 are local CPU registers

Process 1

```
main()
{
int x = 5
x = x+1
var = x;
printf("%d", var)
}
```

Process 2

```
main()
{
int y = 10
y = y-2
var = y;
printf("%d", var)
}
```

Critical section

Eg

- The concurrent execution of process 1 and process 2 is equivalent to a sequential execution where lower level statements are interleaved in some arbitrary order. One such interleaving is:

T0:	Process 1	main()	
T1:	Process 2	main()	
T2:	Process 1	int x = 5;	
T3:	Process 2	int y = 10;	
T4:	Process 1	x = x+1;	(x = 6)
T5:	Process 2	y = y-2;	(y = 8)
T6:	Process 1	var = x;	(var = 6)
T7:	Process 2	var = y;	(var = 8)
T8:	Process 1	Printf(var)	o/p = 8
T9:	Process 2	Printf(var)	o/p = 8

Notice that we have arrived at the incorrect state "output 8-8" when infact output should be "output 6-8".

Pizza problem

We would arrive at this incorrect state because we allowed both processes to manipulate the variable counter concurrently. A situation like this, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the ~~process~~ access takes place, is called a race condition.

Race condition :→ if the final output of of the execution of processes depends ~~on~~ on the execution of the processes.

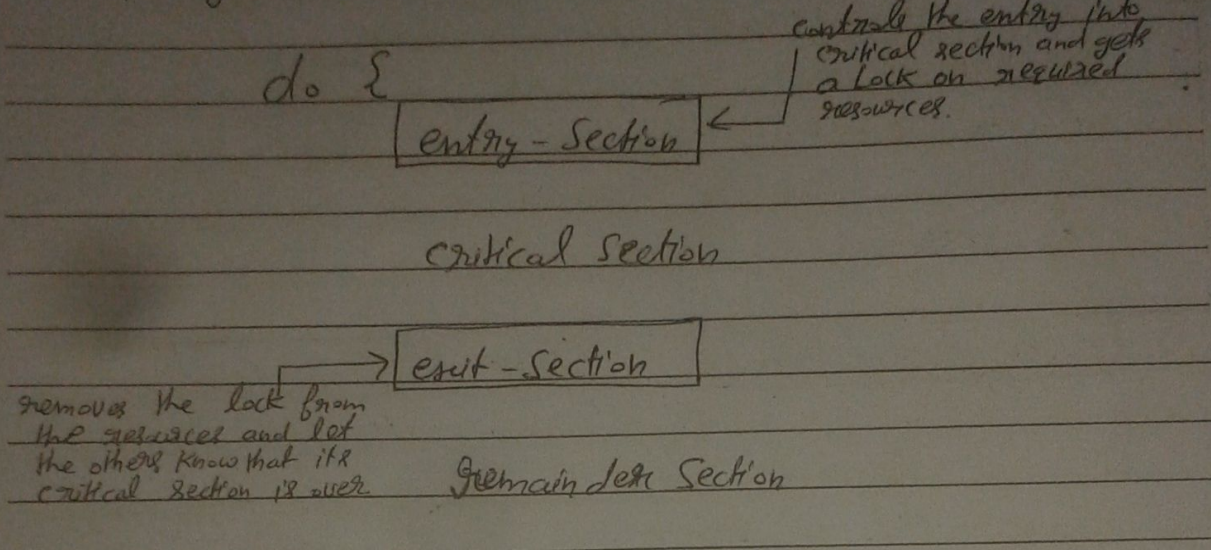
Critical SECTION Problem :→

Critical Section :→ Each process has a segment of code, called a critical section, in which the process may be changing common variables (shared variables), updating a table, writing a file and so on.

When one process is executing in its critical section, no other process is to be allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time. The critical section problem is to design a protocol that the processes

Can use to cooperate.

→ the general structure of a process is,



General Structure of a Process.

→ Each process must request permission to enter its critical section. The section of code implementing this request is the entry Section. The critical section may be followed by an exit section. The remaining code is the remainder section.

Progress:- if critical section is empty and some process wants to go to its critical section, then it should be allowed

→ A solution to the critical-section problem must satisfy the following three requirements:

① Mutual Exclusion:- if Process P_i is executing in its critical section, then no other processes can be executing in their critical sections.

② Progress :- if no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in the decision on which will enter its critical section next, and this selection can not be postponed indefinitely

- if no process is in its critical section, and if one or more threads want to execute their critical section then any one of these threads must be allowed to get into its critical section.

(i.e. processes can not be blocked forever waiting to get into their critical section)

→ waiting should be limited

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.

- No process should have to wait forever to enter

into critical section

(if bounded waiting is not satisfied then it is possible for the starvation).

[B.W is satisfied, when the process requested first will to critical section first]

⇒ There are various solution to the critical section problems as following:-

Solutions

Software Solutions	Hardware Solutions	O.S. and Compiler Solutions
• Dekker's Algo } for 2 process • Peterson's Algo } • Bakery Algo ↓ for n processes	Here special Instructions supported by Hardware are included in Entry or Exit Section eg:- • TestAndSet • Exchange or Swap ↓ both for n no. of process	• Semaphore • monitor

Peterson's Algorithm :-

→ It's a 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 P_0 and P_1. When presenting P_i, we use P_j to denote the other process; that is, j equals $1-i$.

→ Peterson's solution requires two shared data items :-

- int Turn - Indicates whose turn it is to enter into the critical section. If Turn == i, then process P_i is allowed to execute in its critical section.
- boolean flag[2];
  The flag array is used to indicate if a process is ready to enter its critical section or indicates when a process wants to ~~go into~~ enter into their critical section. For example, if flag[i] is True, this value indicates that P_i is ready to enter its critical section. Initially false.

→ In the structure of process in Peterson's algorithm, the entry and exit sections are enclosed in boxes.

- In the entry section, process i first

raises a flag indicating a desire (or ready) to enter the critical section.
- Then Turn is set to j to allow the other process to enter their critical section if process j so desires.
- The while loop is a busy loop, which makes process i wait as long as process j has the Turn and wants to enter the critical.
- Process i lowers the flag[i] in the exit section, allowing process j to continue if it has been waiting.

```
do {
    flag[i] = true;
    Turn = j;                          {do skip}
    while (flag[j] && Turn == j);

    → critical section

    flag[i] = false;

    remainder section

} while (true)
```

The structure of process Pi in Peterson's Solution

→ This solution is correct as having following properties:

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

To prove property 1, we note that each P_i enters its critical section only if either flag[j] == false or turn == i. Also note that, if both processes can be executing in their critical sections at the same time, then flag[0] == flag[1] == true. These two observations imply that P_0 and P_1 could not have successfully executed their while statements at about the same time, since the value of turn can be either 0 or 1 but cannot be both. Hence one of the processes —say— P_j must have successfully executed the while statement, whereas P_i had to execute at least one additional statement ("turn == j"). However, since, at that time, flag[j] == true, and turn == j, and this condition will persist as long as P_j is in its critical section, the result follows: Mutual exclusion is preserved.

To prove properties 2 & 3, we note that a process P_i can be prevented from entering the critical section only if it is stuck in the while loop with the condition flag[j] == true and turn == j; this loop is the only one possible. If P_j is not ready to enter the critical section, then flag[j] == false and P_i can enter its critical section. If P_j has set flag[j] to true and is also executing

busy waiting :- when the process is waiting in loop (keep repeating in loop).

Turn == j, if Turn == i, then P_i will enter the critical section. if Turn == j, then P_j will enter the critical section. However, once P_j exits its critical section, it will reset flag[j] to false, allowing P_i to enter its critical section. if P_j resets to flag[j] to True, it must also set Turn to i. Thus, since P_i doesn't change the value of the ~~value~~ variable Turn while executing the while statement, P_i will enter the critical section (progress) after at most one entry by P_j (bounded waiting).

- Peterson's Algorithm has the following characteristics:-
  - A simple algorithm that can be run by two processes to ensure mutual exclusion for one resource (say one variable or data structure)
  - Does not require any special hardware
  - It uses busy waiting (a spinlock)

Note:- if more than two processes exists in the system, An algorithm for n process process is known as Bakery Algorithm. This algorithm extends the Two process Dekker's Algorithm to n process solution for Bakery algorithm. //

BAKERY Algorithm

- A better solution for more than one processes
It is based on scheduling algorithm commonly used in ~~Bakeries~~ bakeries, Ice Cream stores and the other locations where order must be made out of chaos.
- On entering the store, each customer receives a number. The customer with the lowest number is served next.

- In this algorithm, every process receives a number before entering its critical section. The process with the lowest number enters the critical section next. In case, two Processes P_i and P_j receive the same number and $i < j$, then P_i is served next (ie process names are unique and totally ordered). when exiting from its critical section, the process sets its number to 0. This algorithm uses two shared data structures:
- Choosing : array [0 .. n-1] of Boolean values - Initialized to false
- Number : array [0 .. n-1] of integer values - Initialized to 0.

→ The structure of process is as follows -

Notation:-
(Number[j], j) < (Number[i], i) implies that the above relation is True if Number[j] < Number[i] or if Number[j] = Number[i] and j < i.

```
repeat
choosing[i] = true;
number[i] = max(number[0] ... number[i-1]) + 1;
choosing[i] = false;
for (j=0; j<n; j++)
{ while (choosing[j]);
while ((number[j]!=0) && (number[j],j) < (number[i],i));
}
< critical-section of Pi; >
number[i] = 0;
< Remainder-section of Pi; >
until false;
```

Structure of process P_i used in Bakery Algo:

Mutual Exclusion:- if P_i is in the critical section, it implies that any other process P_k will have a number $k \neq 0$ and number[k] will be greater than number[i] and thus at this time only P_i can execute its critical section. if P_k tries to enter its critical section, it will be looping in the second while statement as its number is greater than number of P_i.

Progress:- It is there becoz a process after exiting from its critical section, makes its number 0 and it needs to take a new number (which is obviously going to be larger as compared to the

existing number to contest with other processes trying to enter their critical section. Therefore every process will be fairly dealt with by being given an opportunity to enter its critical section after a finite time CRS done in bakery by assigning tokens to its customers

Hardware Solutions

→ Many systems provide special Hardware Instructions that allow us either to test and modify the content of a word or to swap the contents of two words atomically

→ The Test-and-Set instruction is executed atomically

```
boolean TestAndSet(boolean *Target)
{
    boolean rv = *Target;
    *Target = True;
    return rv;
}
```

Test-and-set() Instruction.

- if two TestAndSet() instructions are executed simultaneously (each on diff CPU), they will be executed sequentially

⇒ In contrast to TestAndSet() instruction, there is Swap() Instruction that operates on the contents of two words. It is also executed atomically.

```
Void Swap(Boolean *a, Boolean *b)
{
    Boolean Temp = *a;
    *a = *b;
    *b = Temp;
}
```

⇒ Swap() Instruction

⇒ Test-and-Set() Instruction can be implemented mutual exclusion by declaring a Boolean variable 'lock', initialized to false.

```
do
{
while (TestAndSetLock(&lock))
    ; // do nothing

    // critical section

lock = False;
    // remainder section

} while (True);
```

Mutual exclusion Implementation with TestAndSet()

① TestAndSet(int *x)
{ if (*x == 0)
{ *x = 1;
return false;
}
return True;
}

int bolt = 0;

{ bolt = 0; process allowed to C.S.
bolt = 1; " not allowed.

P_0	P_1	P_2
do		
while(TestAndSet(&bolt))	do	do
{	{while(TestAndSet(&bolt)	{while(TestAndSet(&bolt))
do nothing	{do nothing}	{do nothing
}	}	}
<C.S.>	<C.S.>	<C.S.>
bolt = 0;	bolt = 0;	bolt = 0;
<R.S.>	<R.S.>	<R.S.>
}while(1);	}while(1);	}while(1);

→ M.E and progress is satisfied here but B. waiting is not satisfied here bcoz after finishing P_0 C.S bolt = 0 and other process are waiting and now if P_0 wants to go so, It can go in C.S bcoz bolt = 0;

=> The Swap() Instruction implements mutual exclusion by using global Boolean variable lock is declared and is initialized to false. In addition, each process has a local Boolean variable key, ~~The structure~~

```
do{
    key = True;
    while (key == True)
        swap(&lock, &key)
        // critical section
    lock = false;
        // remainder section

}while(True);
```

Mutual-exclusion implementation with Swap().

=> These algorithms satisfy the mutual-exclusion requirement. They do not satisfy the bounded-waiting requirement.

①

③ Swap or Exchange

```
Swap(int &x, int &y)
{ int Temp;
   Temp = x;
   x = y;
   y = Temp;
   return y;
}
```

here int bolt = 0;

K = 1;

P_0	P_1 (K = 1)	P_2 (K = 1)
do {	do	do
int K = 1;	int K = 1;	int K = 1;
{while (Swap (bolt, K))}	while (Swap(bolt, K))	{while (Swap(bolt, K))
{ do nothing }	{ do nothing }	{ do nothing }
<C.S.>	<C.S.>	<C.S.>
bolt = 0;	bolt = 0;	bolt = 0;
<R.S.>	<R.S.>	<R.S.>
} while (1);	} while (1);	} while (1);
		//

Semaphores

Semaphore :→ Semaphore is a global shared variable that, apart from initialization, is accessed only through two standard atomic operations : wait() and signal()

(i) wait (semaphore S) written as P(S)
(ii) signal (semaphore S) written as V(S)

Classical definition of wait and signal :-

wait()

```
int S=1
wait (S)
{
 while (S<=0)
 {
   do nothing
 }
 S--;
}
```

signal()

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

Note:- wait is Entry section
& signal is Exit section

→ Mutual exclusion implementation with semaphores is as:-

```
do {
    waiting (mutex);
    <critical-section>
    Signal (mutex);
    <remainder section>
} while (True);
```

Mutual exclusion implementation with semaphores-

→ It satisfies mutual exclusion and progress but bounded-waiting is not satisfied

- Busy-waiting :→ while a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code. Busy waiting wastes CPU cycles that some other process might be able to use productively. This type of semaphore is also called Spinlock.
- Spinlock are useful in multiprocessor systems. Advantage of a spinlock is that no context switch is required when a process

must wait on a lock and content may take considerable time.

- To overcome the need for busy waiting we can modify the definition of the wait and signal semaphore operations. When a process executes the wait operation and finds that the semaphore value is not positive it must wait. However rather than busy waiting the process can block itself. The block operation places a process into a waiting queue associated with the semaphore and the state of the process is switched to the waiting state. Then, control is transferred to the CPU scheduler which selects another process to execute.
A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal operation. The process is restarted by a wakeup operation which changes the process from the waiting state to the ready state. The process is then placed in the ready queue.

- Spin lock solution is structure implementation

⇒ They are of Two Types of Semaphores
(i) Counting - Semaphore
(ii) Binary - Semaphore

① Counting Semaphore is

here we define a semaphore as 'C' struct

```
Typedef struct {
    int value;
    struct Process *List;
} Semaphore;
```

S.value = 1;

Each semaphore has an integer value and a list of processes list. when a process must wait on a semaphore, it is added to the list of processes. A signal() operation removes one process from the list of waiting processes and awakens that process.

- The Semaphore operation can be defined as:

wait()

```
wait (Semaphore S)
{
    S.value --;
    if (S.value < 0)
    { Block the process and
      add this process to
      the list of Semaphores S
    }
}
```

Signal()

```
Signal (Semaphore S)
{
    S.value ++;
    if (S.value <= 0)
    { pick one process from
      block queue of Semaphore
      and Insert in ready
      queue (wakeup (P));
    }
}
```

-if the semaphore value is negative, its magnitude is the number of processes waiting on that semaphore

P_0	P_1	P_2
do	do	do
{ wait(S)	{ wait(S)	{ wait(S)
<C.S>	<C.S>	<C.S>
signal(S)	signal(S)	signal(S)
<R.S>	<R.S>	<R.S>
}while(1)	}while(1)	}while(1)

-here is bounded waiting

FIFO queue only satisfy the bounded waiting

⇒ if S.value is initially 0, then deadlock will be there & all processes are blocked

S.value initially 1 → 0 → -1 → -2

⇒ Deadlock & starvation:-

- The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes. when such a state is reached, these processes are said to be deadlocked

eg: Set S and Q to value 1:

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

- Suppose that P_0 executes wait(S) and P_1 executes wait(Q), when P_0 executes wait(Q), it must wait until P_1 executes signal(Q). Similarly when P_1 executes wait(S)

it must wait until P_0 execute signal()
Since these signal operations cannot be executed P_0 & P_1 deadlocked

- Starvation :- a situation where processes wait indefinitely within the semaphore
  - Indefinite blocking may occur, if we add and remove processes from the list associated with semaphore in LIFO order.

(2) Binary Semaphore :-

→ A Binary semaphore is a semaphore with an integer value that can range only between 0 and 1.

→
```
Struct BinarySemaphore
{
    Boolean value;
    struct process *L;
} BS;
```

→ The wait() & signal() operation can be redefined as follows:-

```
Wait (Binary Semaphore BS)
{
 if (BS.value == 1)
  {
   BS.value = 0;
  }
 else
 {
  Block the process &
  Insert the process in
  block queue of
  Semaphore
 }
}
```

```
Signal (Binary Semaphore BS)
{
 if (Blocked queue of
   Semphore not empty)
 {
  pick one process from
  block queue and Insert
  in ready queue
 }
 else
 {
   BS.value = 1;
 }
}
```

Eg:

P_0	P_1	P_2
do	do	do
{ wait (S)	{ wait (S)	{ wait (S)
<C.S>	<C.S.>	<C.S.>
Signal (S)	Signal (S)	Signal (S)
<R.S>	<R.S.>	<R.S.>
} while (1);	} while (1);	} while (1);

Initially B.S value : [1]

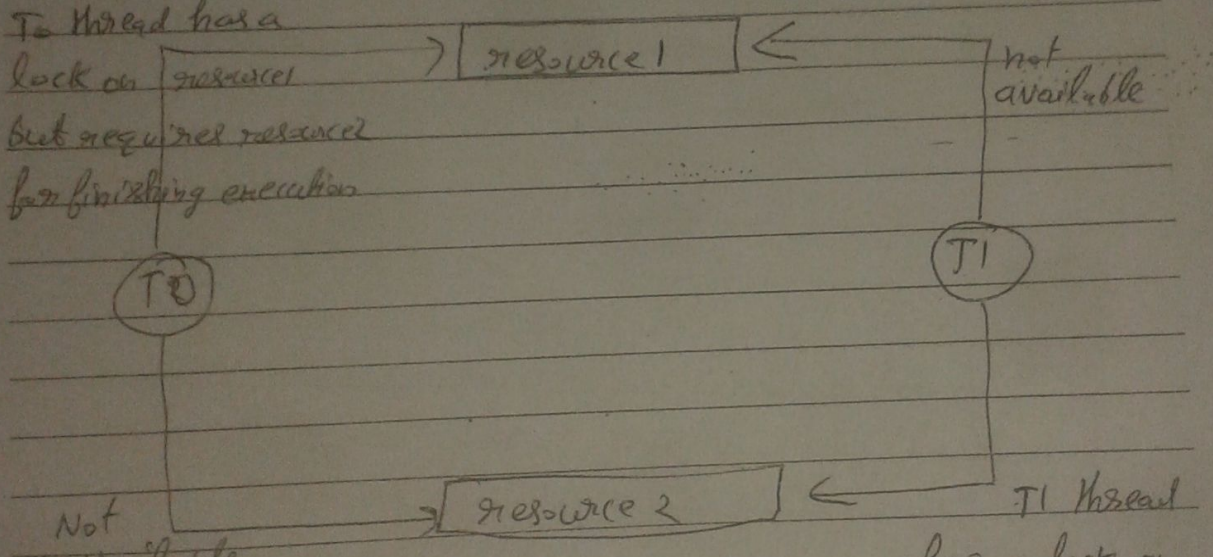
[0] --- Only [1] process should be allowed to go to C.S.

Deadlock

Deadlock :→ A set of processes is in a deadlock state when every process in the set is waiting for an event that can be caused only by another process(es) in the set.

or

Deadlock is the state of the system where the two or more processes are waiting on some event to happen, which never happens, then these processes are said to be involved in the deadlock.



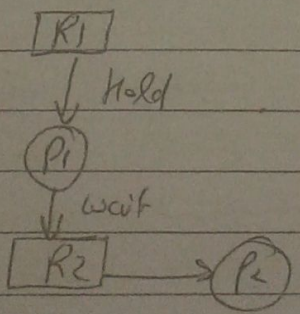
Deadlock characterization :→

⇒ Necessary Conditions for deadlock :→

→ A deadlock situation can arise if the following four conditions hold simultaneously in a system:-

1. **Mutual Exclusion:-**
Resources cannot be shared. only one process at a time can use the resource. if another process requests that resource, the requesting process must be delayed until the resource has been released.

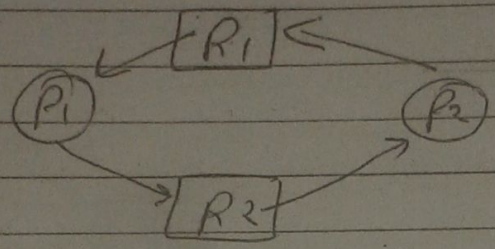
2. **Hold and wait :-**
A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.



3. **No Preemption :-**
Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.

4. Circular wait :-

A set $\{P_0, P_1, P_2, \ldots P_n\}$ of waiting processes must exist such that P_0 is waiting for a resource held by P_1, P_1 is waiting for a resource held by P_2, ... P_{n-1} is waiting for a resource held by P_n and P_n is waiting for a resource held by P_1.



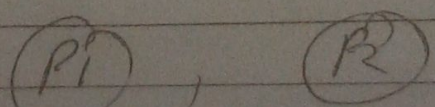
→ if all the above four conditions are existing simultaneously in the system, then definately there exist a deadlock

- all the above four conditions are not Truely independent because the circular wait condition implies the hold-wait condition

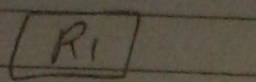
⇒ Resource - Allocation Graph :→

- A Tool for recognizing deadlock is "Resource - Allocation Graph" (RAG)

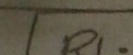
- Pictorially, we represent each process P_i as a circle



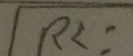
- Each Resource Type R_i is represented as a Rectangle



Since Resource-Type R_i may have more than one instance (like files, I/O devices (printers & DVD) or more than one CPU), we represent each such instance as a dot within the rectangle



- Single instance of resource R1



- two instances of resource R2

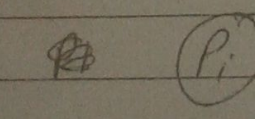
$$RAG = G(V, E)$$

vertices include processes, resources

Edges includes Allocation, Requesting

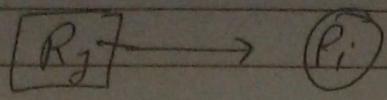
Its edges are of Two Types:-

(a) Request edge :- A directed edge from process P_i to resource Type R_j is denoted by $P_i \rightarrow R_j$; It signifies that process P_i has requested an instance of resource Type R_j and is currently waiting for that resource

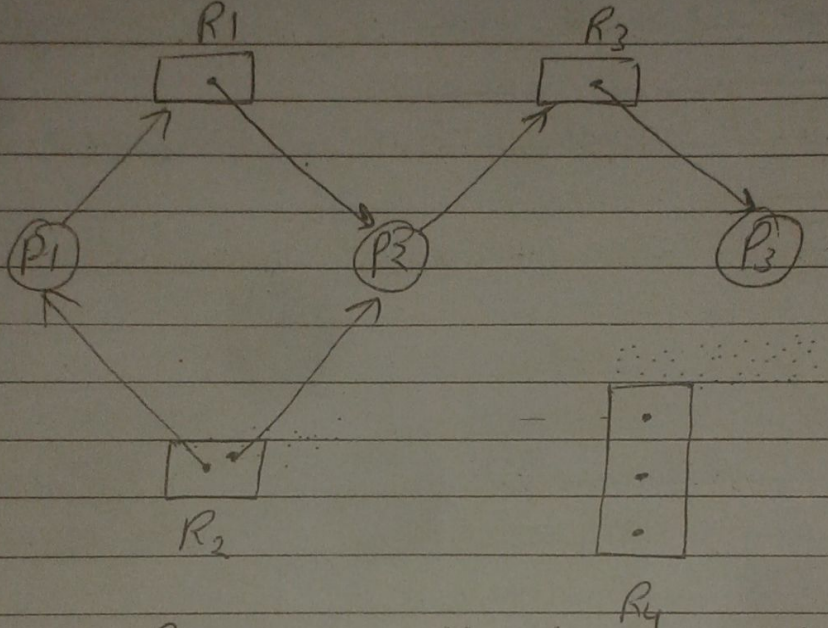


(b) Assignment edge :- A directed edge from resource R_j to process P_i is denoted by $R_j \rightarrow P_i$; it signifies that

an instance of resource type R_i has been allocated to process P_i.



Eg:-



Resource- allocation graph

process states:-

- Process P1 is holding an instance of resource type R_2 and is waiting for an instance of resource type R_1.
- Process P_2 is holding an instance of R1 and an instance of R_2 and is waiting for an instance of R_3.
- Process P_3 is holding an instance of R3.