6/2/2020.

UNIT - 3

PROCESS SYNCHRONISATION.

Co operating process :

producer code

```
while (true)
{
    while (counter == Buffer_size);
    Buffer[in] = next_produced ;
    in = (in+1) % Buffer_size ;
    counter++ ;
}
```

Consumer code :

```
while (true)
{
    while (counter == 0);
    next_Consumed = Buffer[out] ;
    out = (out+1) % Buffer_size ;
    counter-- ;
}
```

Race condition:

A situation where several processes access and manipulate the same data con-currently and the outcome of the execution depends on the particular order in which the access takes place is called race condition

Critical Section:

Each process has a segment of code called critical section in which the process may be changing common variables, updating a table, writing a file and so on

Critical section problem:

General structure:

```
do {
        Entry section
            Critical section;
        Exit Section
            Remainder section;
} while (true);
```

1 Critical section must specify three requirements as stated below:

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 classes that are not executing in their remainder section can participate in the decision on which will enter next.

3. Bounded waiting: There exists a bound or limit on the no. of times that other processes are allowed to enter their critical sections after a process makes a request.

7/2/2020 :

* Approaches

1. Premptive kernels. - good for co-operating process
2. Non-Premptive Kernels.

* Peterson's solution.

%

Process structure P_i : int turn. boolean flag;

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

        Critical section

    flag[i] = FALSE ;

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

Synchronisation Hardware :

```
do {
        Acquire LOCK

            Critical condition ;

        Release LOCK

        Remainder section ;
    } while(TRUE);
```

8/2/2020 :

Semaphores :

```
wait(s)
{
    while(s<=0);
    s--;
}

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

Two types :

i. Binary - counts only bet^n 0 and 1 - generally used for implementing process

ii. Counting - used for resources/instances attached to the system

* Mutual Exclusion implementation using Binary Semaphore.

```
do {
    waiting(mutex);
        Critical Section;
    signal(mutex);
        remainder section;
} while(TRUE);
```

A semaphore 's' is an integer variable that apart from initialisation ~~whose~~ is accessed only through 2 standard atomic operations wait and signal

wait is represented as P → to test

signal " " V → increment.

* Classical problems of Synchronisation

1. Bounded Buffer

Structure of producer

```
do {
    wait (empty);
    wait (mutex);
    ========
    signal (mutex);
    signal (full);
} while (TRUE);
```

Structure of Consumer.

```
do {
    wait (full);
    wait (mutex);
    ========
    signal (mutex);
    signal (empty);
} while (TRUE);
```

- In this problems there are n buffers each capable of holding 1 item.
- The mutex semaphore provides mutual exclusion for accesses to the buffer ~~stream~~ full and is initialized to the value 1.
- The empty and full semaphores count the number of empty and full buffers, empty is initialized to the value n and full is initialized to the value 0
- The code for producer and consumer are as given above.
- It can be interpreted as the producer producing full buffers for the consumers or as the consumer producing empty buffers for the producer

2. Readers-Writers Problem.

6M OR 8M

Structure of writer :

```
do{
        wait (wrt);
        ------
        ------
        signal (wrt);
}while (TRUE);
```

Structure of Reader

```
do{
   wait (mutex);
   read_count++;
   if (read_count == 0)
        wait (wrt);
   signal (mutex);
   ------
   ------
   wait (mutex);
   read_count--;
   if (read_count == 0)
        signal (wrt);
   signal (mutex);
}while (TRUE);
```

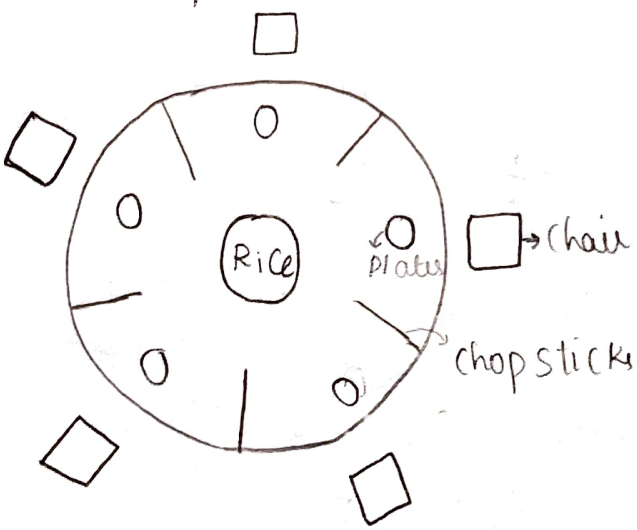
- A database is to be shared among several concurrent processes.
- Some processes may want to read data and others may want to update the data
- These processes can be referred to as readers and writers.
- If 2 readers access the shared data at the same time there is no adverse effects but if a writer and some other thread (reader / writer) access the data at the same time the data may get affected
- To overcome this problem it is required that the writers have exclusive access to the shared database
- This is called as readers writers problem.
- This problem has several variations like

1. It requires that no reader will be kept waiting unless a writer has already obtained permission to use the shared data

2. It requires that once a writer is ready writer performs the write operation as soon as possible i.e. if a writer is waiting to access the data no new readers may start reading

- A mutex and wrt are initialized to 1 read_count is initialized to 0, the semapho wrt is common to both reader and writer processes. The semaphore mutex is used to ensure mutual exclusion

3. Dining Philosophers Problem:



- 5 philosophers spend their lives thinking and eating.
- They share a circular dining table surrounded by 5 chairs, 5 plates and 5 chopsticks

- The centre is a bowl of rice
- When a philosopher thinks he does not interact with others. From time to time they get hungry and try to pick up the 2 chopsticks that are close to them (left and right chopstick)
- While eating if other (neighbour) philosophers feels hungry then he cannot eat because only 1 chopstick is left out.

```
do{
    wait(chopstick[i]);
    wait(chopstick[(i+1)%5]);
    ---------
     // eat
    ---------
    signal(chopstick[i]);
    signal(chopstick[(i+1)%5]);
    ---------
    // think
}while(TRUE);
```

- A simple solution is to represent each chopstick with a semaphore
- A philosopher tries to grab a chopstick by executing a wait operation on that semaphore
- Then releases chopsticks by executing the signal operation on the appropriate semaphore
- All chopsticks are initialized to one, they are declared as semaphore chopstick[5];

5-2-2020

* Deadlock

  Request

  Use

  Release

  4 Conditions:

1. Mutual Exclusion
2. Hold and wait
3. No preemption
4. Circular wait

Deadlock:

A waiting process is not able to change its state because of the resources held by other processes then it goes into a state called deadlock.

* A deadlock situation can arise if the following 4 condition hold simultaneously in a system.

1. Mutual Exclusion:

Atleast 1 resource must be held in a non-shaxable mode i.e. only one process at a time can use the resource.

2. Hold and wait

A process must be holding atleast one resource and waiting to acquire additional resources that are currently being held by other processes.

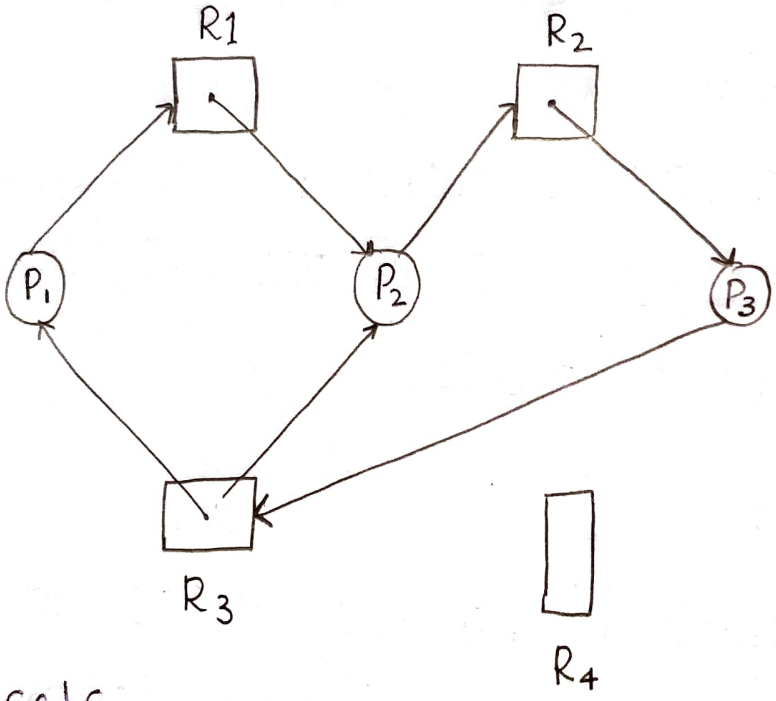
3. No preemption:

Resources cannot be preempted i.e. a resource can be released only voluntarily by the process holding it.

4 Circular wait

A set of waiting processes exist such as P_0 is waiting for a resource held by P_1. P_1 is waiting for a resource held by P_2 and so on. P_n is waiting for a resource held by P_0. This creates circular dependency

* Resource Allocation Graph:



3 sets

$P = \{P_1, P_2, P_3\}$

$R = \{R_1, R_2, R_3, R_4\}$

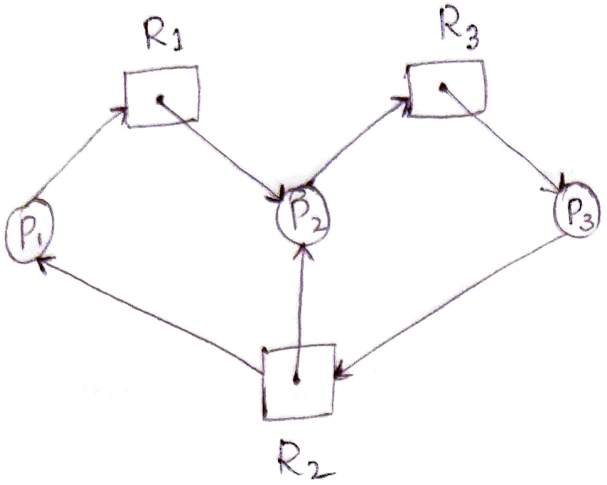
$E = \{ P_1 \to R_1, P_2 \to R_2, R_1 \to P_2, R_2 \to P_3, R_3 \to P_1, R_3, P_2 \}$

$P_i \to R_j$.

$R_i \to P_j$.

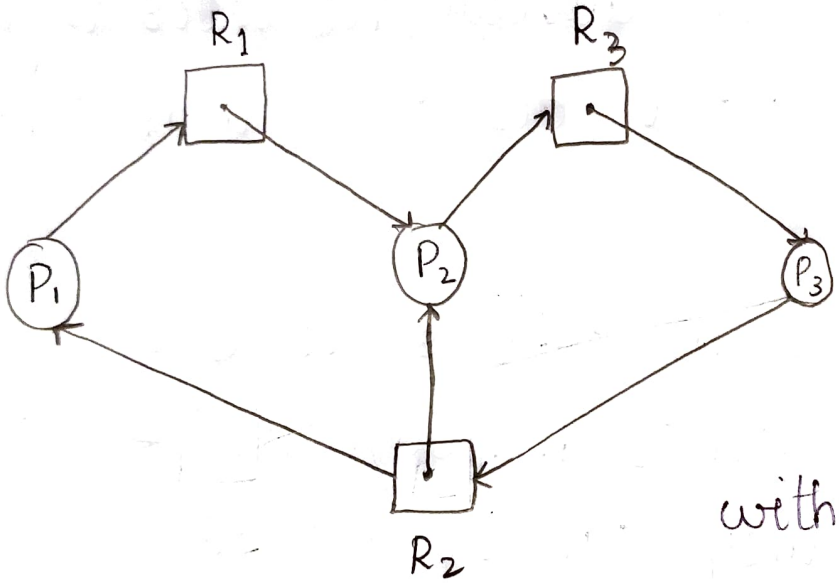
Cycle: $P_1 \to R_1 \to P_2 \to R_2 \to P_3 \to R_3 \to P_1$

$P_2 \to R_2 \to P_3 \to R_3 \to P_2$



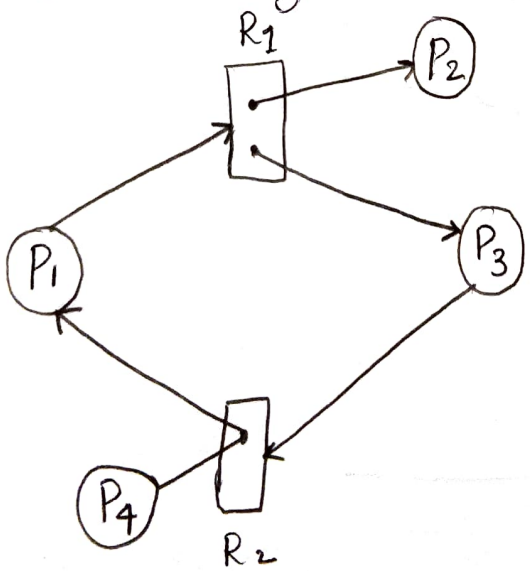
cycles : $P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$



without deadlock

cycles : no cycles



cycle :

$P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

without deadlock.

3-3-2020

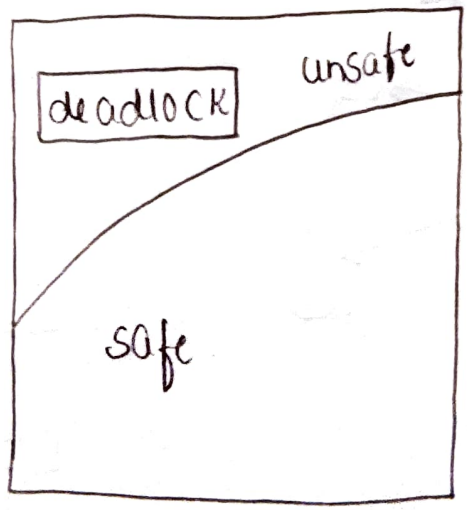
Methods to handle Dead lock

1. Protocol - prevent / avoid
2. Allow to enter into deadlocks, detect, overcome
3. Ignore and pretend like deadlock never occurs

Safe State:

If a system can allocate resources to each process and still avoid deadlock then it is in safe state.

A system is in safe state only if there exists a safe sequence i.e. sequence of processes in which order they can be executed



Magnetic tape drive : 12

Eg:

	Max. Need	Current Need.
P_0	10	5
P_1	4	2
P_2	9	2.

Rem : $12 - 9 = 3$

$P_1 = (2 + 2) = 4$

Available = 5

$P_0 = (5 + 5) = 10$

Available = 10

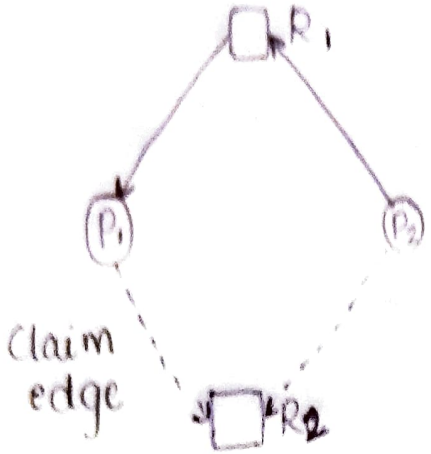
$P_2 = (2 + 7) = 9$

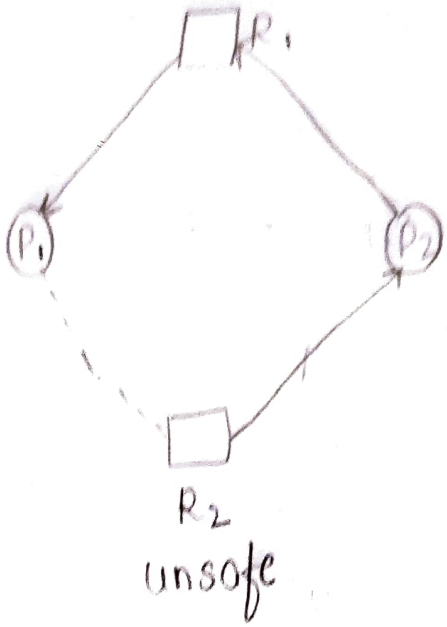
Available : 12.

$\therefore \langle P_1, P_0, P_2 \rangle$ - Safe sequence

5. 3.2020

* Resource Allocation Graph Algorithm





unsafe

Bankers Algorithm:

1

	Allocation			Max			Available			Need		
	A	B	C	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	3	3	2	7	4	3
P_1	2	0	0	3	2	2	5	3	2	1	2	2
P_2	3	0	2	9	0	2	7	4	3	6	0	0
P_3	2	1	1	2	2	2	7	4	5	0	1	1
P_4	0	0	2	4	3	3	7	5	5	4	3	1

Need ≤ work (Available) ⇒ work = work + alloc

P_0 748 ≤ 332 ×

P_1 122 ≤ 332 ✓

work = 332 + 200

= 532

P_2 : $600 \leq 532$ ×

P_3 : $011 \leq 532$ ✓

Work = 532 + 211

= 743

P_4 : $431 \leq 743$ ✓

Work = 743 + 002

= 745

P_0 - $743 \leq 745$ ✓

Work = 745 + 010

= 755

P_2 - $600 \leq 755$ ✓

- Work = 755 + 302

(A B C)

= 1057

$< P_1, P_3, P_4, P_0, P_2 >$ safe sequence.

6-3-2020

2 Find the safe sequence for a system in a above example in which P_1 requests 1 additional instance of resource type A and 2 instances of resource type C.

	Allocation A B C	Max A B C	Available A B C	Need A B C
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	3 0 2	3 2 2	6 3 4	0 2 0
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

P_0 : 743 <= 332 ×

P_1 : 020 <= 332 ✓

332 + 302 = 634

P_2 600 <= 634 ✓

634 + 302 = 936

P_3 011 <= 936 ✓

936 + 211 = 1147

P_4 : 4 3 1 <= 1 1 4 7 ✓

: 1147 + 002 = 1149

P_0 : 743 <= 1149

1149 + 010

1159

Safe Sequence : < P_1 P_2 P_3 P_4 P_0 >

3

	Allocation	Max	Available	Need
	A B C D	A B C D	A B C D	A B C D
P_0	0 0 1 2	0 0 1 2	1 5 2 0	0 0 0 0
P_1	1 0 0 0	1 7 5 0		0 7 5 0
P_2	1 3 5 4	2 3 5 6		1 0 0 2
P_3	0 6 3 2	0 6 5 2		0 0 2 0
P_4	0 0 1 4	0 6 5 6		0 6 4 2
	2 9 10 12			
	1 5 2 0			
	3 14 12 12			

P_0 : 0000 <= 1520 ✓

1520 + 0012

: 1532

P_1 - 0 7 5 0 <= 1 5 3 2 ✓

1532 + 1000 = 2532

P_2 - 1002 <= 2532 ✓

2532 + 1354 = 3886

P_3 - 0020 <= 3886 ✓

3886 + 0632

= 3 14 12 8

P_4 0642 <= 3 14 12 8

3 14 12 8 + 0 0 1 4

3 14 12 12

< P_0 P_1 P_2 P_3 P_4 >

The system is in safe state

If the request from process P1 arrives for (0 4 2 0) can the request be granted for the above system

Bankers Algorithm

The Bankers algorithm has n no. of processes in the system and m no. of resource types the following DS are needed

1. Available : It is an array of length m indicates the no. of available resources of each type i.e. if available[j] = K there are K instances of resource type R_j.

2. Max : This is an n x n matrix which holds max. demand of each process if max[i][j] = K i.e. the process P_i may request atmost K instances of resource type R_j.

3. Allocation : This is an n x n matrix which has the no. of resources of each type currently allocated to each process

4. Need : This is an n x m matrix which indicates the remaining resource need of each process

Safety Algorithm

Step 1 : Initialize .

Work = Available

Finish = false

for i = 0 to n - 1

Step 2 : Find an i such that

a Finish[i] = False,

b. Need <= work

if no such i exists go to 4

Step 3 : work = work + allocation

finish[i] = TRUE ;

go to step 2.

Step 4 : if Finish[i] = TRUE then

System is is Safe state

2. Resource Request Algo.

1 If Request i <= Need i goto 2

2 If Request i <= Available goto 3.

(Pi must wait)

3. Available = Available - Request i.

Allocation i = Allocation i + Request i

Need i = Need i - Request i