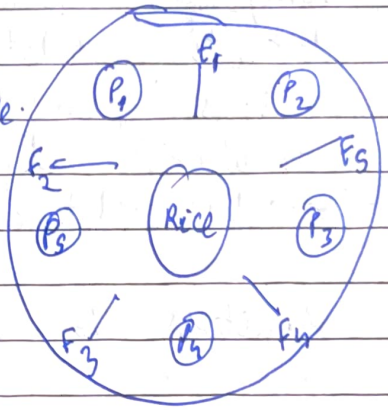
Unit 5.3

Classical Problems of Synchronization :

(1) Dining Philosophers Problem :



- There are 5 philosophers sitting around table with bowl of rice.
- There are 5 chopsticks, one b/w each pair of philosophers.
- To eat philosophers need 2 chopsticks - one on their left & right.
- Each philosopher alternates between think() & eat() operation.
- The challenge is to coordinate their actions so that no two adjacent philosophers try to pick up the same chopstick at same time causing a deadlock.

Solution :

```
# define N 5
# define LEFT (i+N-2)%N // number of i's left neighbour
# define RIGHT (i+1)%N
# define THINKING 0
# define HUNGRY 1 // Philosopher trying to get forks
# define EATING 2
typedef int semaphore;
int state[N];
semaphore mutex = 1;
semaphore s[N];
```

P_1	S_1	S_2
P_2	S_2	S_3
P_3	S_3	S_4
P_4	S_4	S_5
P_5	S_1	S_5

```
void Philosopher (int i)
{
  while (TRUE) {
      think ( );
      take_forks (i);   // acquire 2 forks or block
      eat ( );
      Put_forks (i);    // Put both forks back to table
    }
}

void take_fork (int i)
{
  down (& mutex)        // enter critical region
  state [i] = HUNGRY
  test (i);             // try to acquire 2 forks
  up (mutex);           // exit critical region
  down (& s [i]);       // block if forks were not acquired
}

void Put_forks (int i)
{
  down (& mutex);  enter // enter critical region
  State [i] = ~~HUNGRY~~; THINKING;
  test ( LEFT);
  test (RIGHT);
  up (& mutex);  // exit critical region
}
void test (i) {{
 if ( state [i] == HUNGRY && state [LEFT] != EATING && state [RIGHT] != EATING)
{
    State [i] = EATING;
    up (& s [i]);
}
}
```

Same database
R - W ⇒ Problem
W - R ⇒ Problem
W - W ⇒ Problem
R - R ⇒ No Problem.

Page No.
Date

(2) Readers Writers Problems:

- There is shared data resource that can read concurrently but must be accessed exclusive when writing.
- There are multiple readers that only read data.
- There are multiple writer processes that can update/modify the data.
- The challenge is to coordinate the readers & writers to allow maximum parallelism but prevent data corruptions.
- Readers increment a readCount when starting, decrement on end.
- Writers acquire a write lock, blocking if any readers or writers active.

Pseudocode (Solution)

```
typedef int semaphore;
Semaphore mutex=1;
Semaphore db=1;
int rc=0;

void reader(void)
{
  while(TRUE){
      down(&mutex);        //get exclusive access
      rc=rc+1;             //one more readers now
      if(rc==1) down(&db);
      up(&mutex);          // release exclusive access
      read_data_base();    //access the data
      down(&mutex);
      rc=rc-1;             //one readers less
      if(rc==0) up(&db);
```

Four Cases :-
i) Read First - Writer → Problem
ii) Write First - Reader → Problem
iii) Writer - Writer → Problem
iv) Read - Read → No Problem

```
    up (&mutex);
    use_data_read();        //non critical region (exit)
  }
}

Void Writer(void)
{
  while(TRUE){
    think_up_data();        // non-critical region
    down(&db);              // get exclusive access
    write_up_data_base();   // update the data
    up(&db);                // release
  }
}
```

(3) Producer-Consumer Problems

- There are multiple producer threads/processes generating data items.
- There are multiple consumer threads/processes consuming data items.
- The producers & consumers share a fixed size buffer used to store and pass data items.

- Challenge is to synchronize the producers & consumers to ensure:
  i) Producers cannot add items to a full buffer.
  ii) Consumers cannot remove items from an empty buffer.
- Counter tracks available slots & items in the buffer.

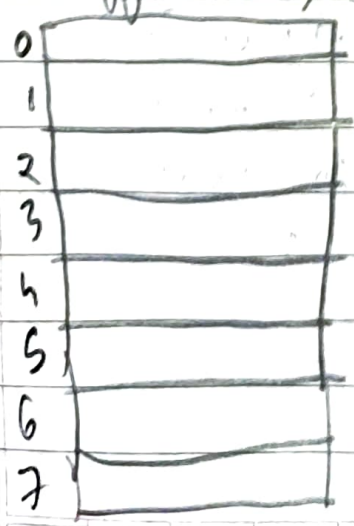
```
int count = 0;

void producer (void)
{
    int itemp;
    while (TRUE)
    {
        Produce_item (item p);
        while (count == n);
        Buffer [in] = item p;
        in = (in + 1) mod n;
        count = count + 1 ;
    }
}
```

Load R_p, m[count]
INCR R_p,
Store m[count], R_p

n = 8

Buffer [0, ..., n - 1]



count

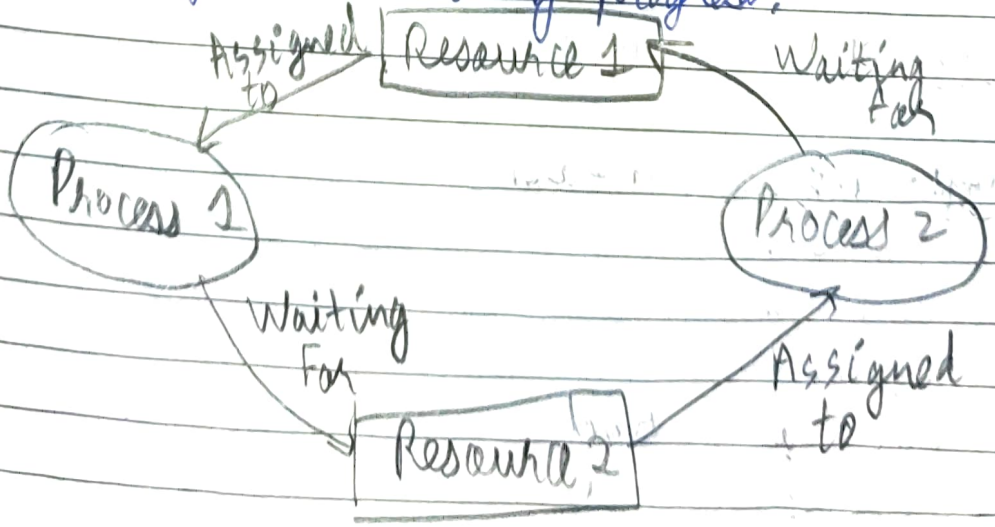
```
void Consumer (void)
{
    int item c;
    while (TRUE)
    {
        while (count == 0);
        item c = Buffer (out);
        Out = (Out + 1) mod n;
        Count = count - 1;
        Process_item (item);
    }
}
```

```
Load R_c, m[count]
DECR R_c
Store m[count], R_c
```

Unit 3, 4

Deadlock's

Deadlock refers to a situation where two or more processes are blocked waiting for each other to release resources, resulting in none of them making progress.



Page No.
Date

Deadlock Vs Starvation

Deadlock	Starvation
• It is a process situation where no process gets blocked & no process proceeds.	• Starvation is a situation where the low priority process got blocked and high priority processes proceed.
• Deadlock is an infinite waiting.	• Starvation is a long waiting but not infinite.
• The requested resource is blocked by the other process.	• The requested resource is continuously be used by the higher priority processes.
• Every deadlock is always starving.	• Every starvation need not be deadlock.

◇ Principles of Deadlock:

(i) **Mutual Exclusion**: Each resource must be either currently assigned to exactly one process or available.

(ii) **Hold and Wait**: A process must be holding at least one resource and waiting to acquire additional resources held by other processes.

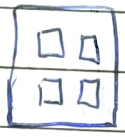
(iii) **No-Preemption**: Resources cannot be forcibly taken away from process they must be ~~removed~~ released voluntarily by the process holding them.

(iv) **Circular Wait**: A set of waiting process must exist such that each process in the set is waiting for a resource held by next process in set; creating circular chain of processes waiting for resources.

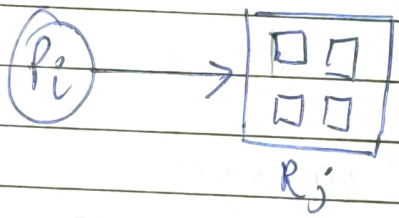
V) Resource Allocation Graphs:

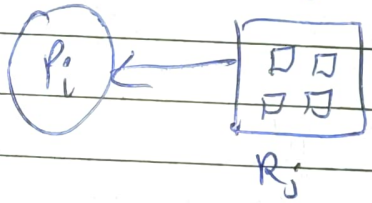
A graph model known as RAG can be used to represent the relationships b/w processes & resources. It helps in visualizing the potential for deadlocks.

Process ◯

Resource type (4 instances) 

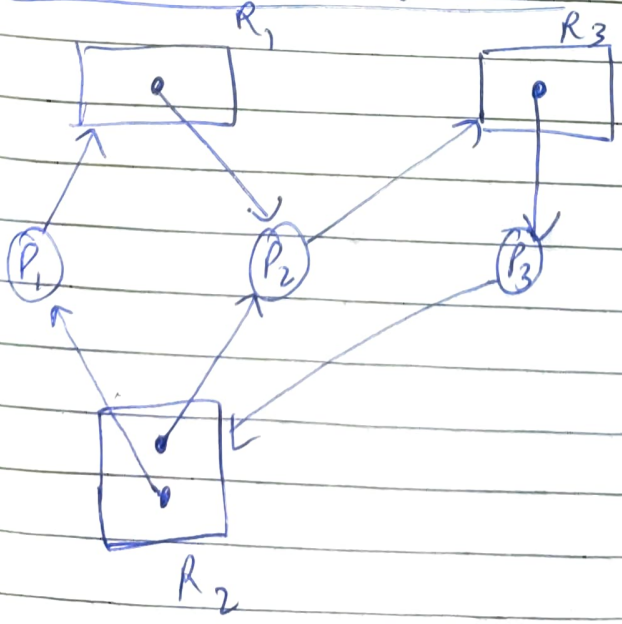
P_i requests instance of R_j



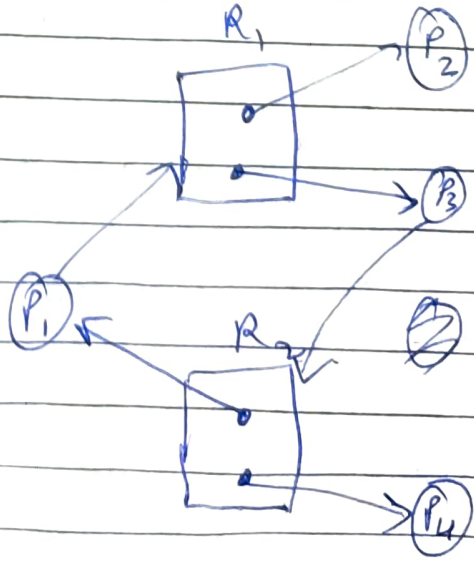


P_i holding an instance of R_j

RAG with a Deadlock



RAG with a cycle but no deadlock



Page No.
Date

Basic facts

- If graph contains no cycles ⇒ no deadlock
- If graph contains a cycles ⇒
  - If only one instance per resource type ⇒ deadlock
  - If several " ⇒ may or may not

◇ Deadlock Prevention

Ways to Prevent Deadlock are:-

(i) Mutual Exclusion: not required for shareable resources (eg: read only file); must hold for non-shareable resources

(ii) Hold & Wait: must guarantee that whenever a process requests a resource it does not hold any other resources.
- The process should be required with resources allocated before it begins execution.
- Low resource utilization starvation possible.

(iii) No-Preemption:
- If a process that is holding some resources requests another resource that can not be held immediately allocated to it, then all resources currently being held are released.

(iv) Circular Wait: To avoid circular wait, we can assign a priority number to each of resource. A process can't request for less priority resource.

Deadlock Avoidance:

- The most useful model requires each model process declare the maximum number of resources of each type they need.
- The deadlock avoidance algo. dynamically examines the resource allocation state to ensure that there can be never a circular wait condition.

The resource allocation state of a system can be defined by instances of available & allocated resources, and the maximum instance of the resources demanded by process.

Safe State:

When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state.

Basic facts -

If a system is in safe state → No deadlock

" unsafe state ⇒ possibility of deadlock.

Avoidance → Ensure that a system will never enter an unsafe state.

Avoidance Algorithms

Single instance of a resource type – Resource Allocation Graph

Multiple" " – Bankers's Algo.

* Bankers Algorithm (Deadlock Avoidance / Deadlock Detection)

Total A = 10, B = 5, C = 7

A → CPU B → Memory C → Printer

A, B, C → Resource types

Process	Allocation (A B C)	Max needed (A B C)	Available (A B C)	Remaining Need (Max need − Allocation) (A B C)	
P_1	0 1 0	7 5 3	3 3 2	7 4 3	(P_0) ✓
P_2	2 0 0	3 2 2	5 3 2	1 2 2	(P_2) ✓
P_3	3 0 2	9 0 2	7 4 3	6 0 0	(P_3)
P_4	2 1 1	4 2 2	7 4 5	2 1 1	(P_4) ✓
P_5	0 0 2	5 3 3	7 5 5	5 3 1	(P_5) ✓
	7 2 5		10 5 7		

Available = Total − Allocation.

Safe Sequence

$P_2 \rightarrow P_4 \rightarrow P_5 \rightarrow P_1 \rightarrow P_3$

No deadlock will occur in this

Current Availability = 10 − 7 = 3

= 5 − 2 = 3

= 7 − 5 = 2

All Remaining need ≤ Available (Process executes)

	A	B	C
Now Current Availability	3	3	2
P_2 Executes & Releases	+2 5	+0 3	+0 2

A → 12 D → 10
B → 12
C → 8

* Banker's (Resource Request Algorithm)

Process	Allocation				Max				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
P_0	2	0	0	1	4	2	1	2	3	3	2	1
P_1	3	1	2	1	5	2	5	2	5	3	2	2
P_2	2	1	0	3	2	3	1	6	6	6	3	4
P_3	1	3	1	2	1	4	2	4	7	10	6	6
P_4	1	4	3	2	3	6	6	5	10	12	8	7
									12	12	8	10

going in sequence

Need Matrix :-

Process	Remaining Need				
	A	B	C	D	
P_0	2	2	1	1	✓
P_1	2	1	3	1	✓
P_2	0	2	1	3	✓
P_3	0	1	1	2	✓
P_4	2	2	3	3	✓

Safe sequence: $P_0 \rightarrow P_3 \rightarrow P_4 \rightarrow P_1 \rightarrow P_2$

Q. if request from P_1 arrives for (1, 1, 0, 0) can request be immediately granted?

Ans We will update the Allocation vector for P_1 process from
[3 1 2 1] → [4 2 2 1] &
Available vector.
[3 3 2 1] → [2 2 2 1] &
Need vector

[2 1 3 1] → [1 0 3 1].

Safe sequence after this request is = P_0 P_3 P_4 P_1 P_2

- **Banker's Algo:**

i) Multiple instances
ii) Each process must a priori claim maximum use.
iii) When a process requests a resources it ~~must return them in a finite~~ may have to wait.
iv) When a process gets all its resources it must return them in a finite amount of time.

Data Structure for Banker's Algo :-

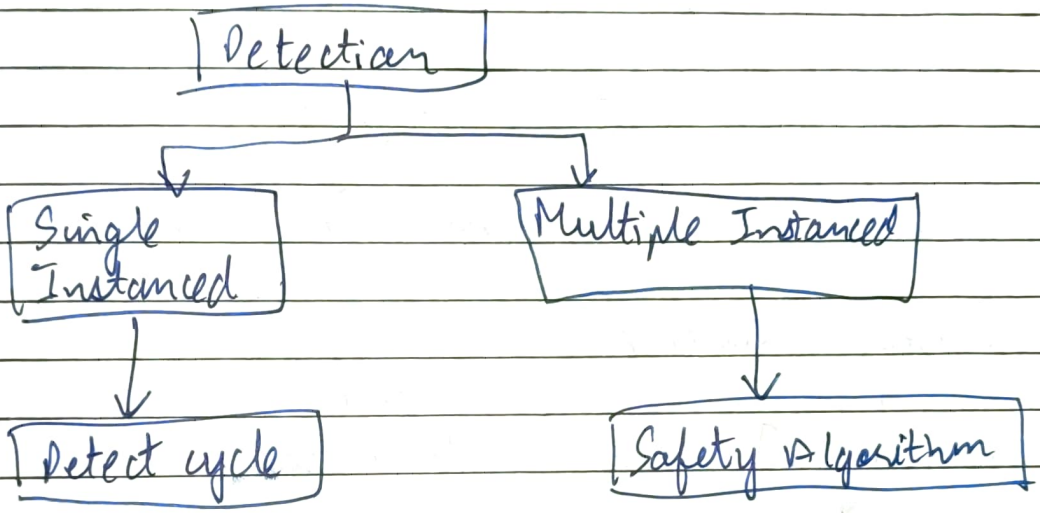
let n = no of processes
m = no of resource types

- Available : Vector of length m. If available [j] = k there are k instances of resource type R_j available.
- Max : n x m matrix
- Allocation : n x m matrix, If Allocation [i, j] = k then P_i is currently allocated k instances
- Need : n x m matrix. Need [i, j] = k

Need [i, j] = Max [i, j] - Allocation [i, j]

◊ Deadlock Detection

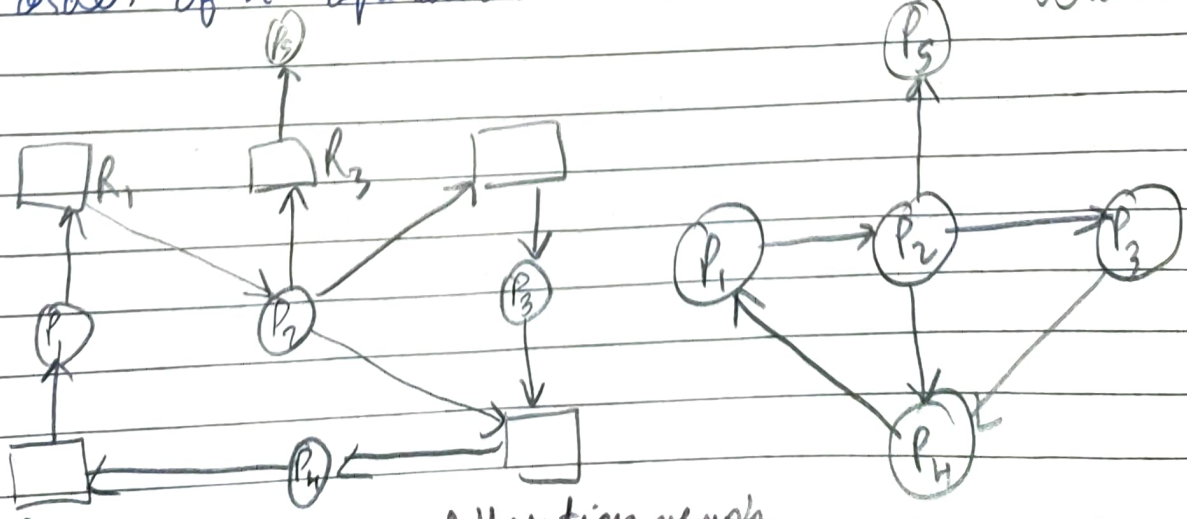
- Deadlock detection involves determining if a deadlock has occured by analyzing state of the system.
- It allows deadlock to occur, then detects it and initiates recovery.



If there is cycle there is a deadlock.

* Single Instance: Maintain wait for graph
  Nodes are processes
  $P_i \rightarrow P_j$ if P_i is waiting for P_j.

An algo to detect a cycle in a graph requires an order of n^2 operation.



(i) Available : A vector of length m

(ii) Allocation: n x m matrix

(iii) Request: An n x m matrix
Request [i] [j] = k, the process P_i is requesting k more instances

* Detection Algorithm : (safety algo)

(i) Let work (m) & Finish (n) be vectors.

(a) Work = Available
(b) For i = 1, 2, ..., n if Allocation$_i$ ≠ 0 then
Finish [i] = false, otherwise Finish [i] = true

(ii) Find an index i such that both:

(a) Finish [i] == false
(b) Request$_i$ ≤ Work

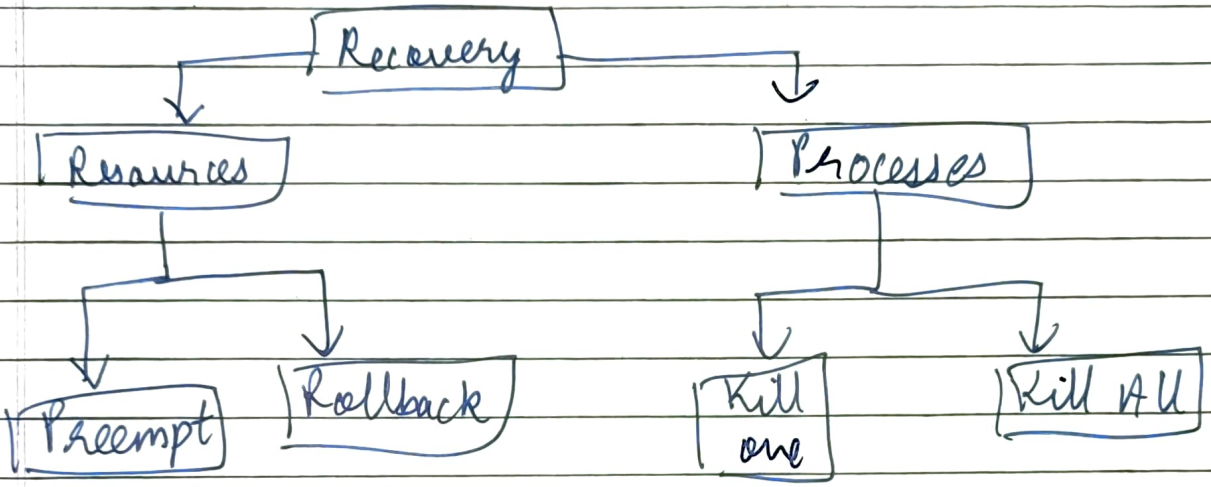
(iii) Work = Work + Allocation$_i$
Finish [i] = true

(iv) If Finish [i] == false, for some i, then the system is in deadlocked state.

Algorithm complexity : $O(m \times n^2)$ to detect whether system is in deadlock state.

Deadlock Recovery:

Abort all deadlocked processes.



Recovery:

Preempt - We can snatch one resource from the owner (process) and give it to another process with expectation that it will complete the execution and will release soonest.

Rollback - System passes through various states to get into deadlock state. The OS can rollback the system to previous safe state.

Processes:

Kill a process - The OS kills a process which has done least amount of work until now.

Kill all process - This is not suggestible approach but can be implemented if the problem is serious.