# Module - 3

## (Inter Process Communication)

### Two types of (IPC)

* **Independent Processes** -

They cannot affect or be affected by other processes executing in the system.

* **Cooperating Processes** - (sharing data with other processes

They can affect or be affected by other processes executing in the system.

Q What is Interprocess Communication? (main fun^n → Data transfer, shared data, Event notify, resource share → process control)

It is a Mechanism provided by operating system that allows processes to communicate with each other. This communication could involve a process letting another process know that some event has occured or the transferring of data from one process to another.

* **Synchronization in Inter Process Communication**

- **Semaphore**: A semaphore is a variable that controls the access to a common resource by multiple processes.
  There are two types of semaphores : one is binary semaphores and counting semaphore

- **Mutual Exclusion**: It basically requires only one process thread can enter the critical section at a time. This is useful for synchroni... at...

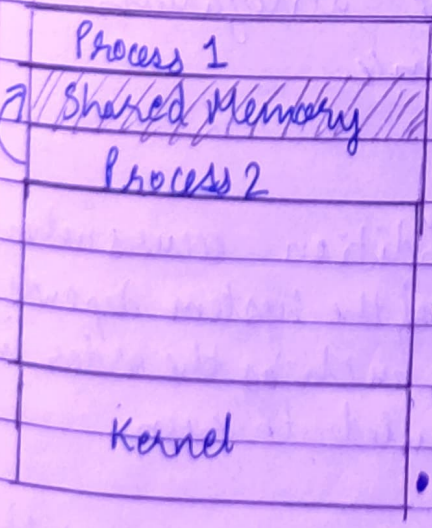
- **Barrier**: A barrier does not allow individual processes to proceed until all the processes reach it. Many parallel languages and collective routines impose barriers.

- **Spinlock**: This is a type of lock. The processes trying to acquire this lock wait in a loop while checking if the lock is available or not.
  This is known as busy waiting because the process is not doing any useful operation even though it is active.

## * Approaches to Inter Process Communication

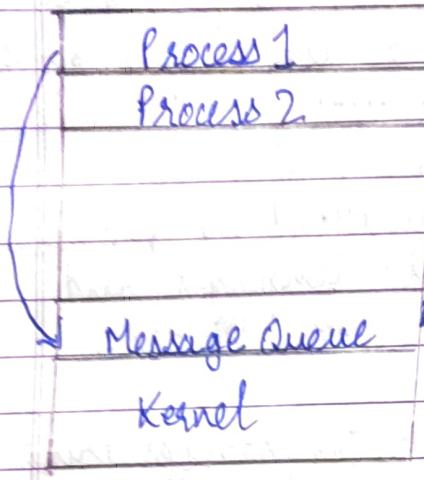
- Pipes
- Message Queues
- Shared Memory
- Message Passing
- Direct Communication
- Indirect Communication
- FIF...

### (1) Shared Memory




- It is method which involves use of shared memory region, where multiple processes can share same data in memory.
- Each process can read & write to the shared memory region, allowing them to exchange data and coordinate their actions...
- It gives high performance as it eliminates the need to copy data b/w address spaces and allows processes to access the same data in memory.

(2) **Message Queue**




- It involves use of shared queue, where processes can add messages & retrieve message for communicati
- The queue acts as buffer for the mes
- In this IPC, each message has priority associated with it, and messages are retrieved from the queue in order of their ~~property~~ priority.
- This allows process to prioritize delivery of important mes
- It provides flexible & scalable method of communicati b/w all processes as messages can be sent & received asynchronously.

---

Unit 5.1

**Concurrency:**

◇ **Issues with concurrency** ~~malfunctions~~

Concurrency in O.S refers to ability of multiple processes to execute in overlapping time periods allowing for better resource utilization & improved system performance.
Some common issues are:

* **Race Condition**: A race condition occurs when the behaviour of the system depends on relative timing of events, such as the order in which threads are scheduled to run.

issue: If multiple threads or processes access shared

* Atomicity - Operations within critical section must be atomic meaning they are executed as a single, indivisible unit.

data concurrently without proper synchronization it can lead to unpredictable & incorrect results.

* Deadlock : A situation in which two or more process are unable to proceed because each is waiting for one of the others to do something

(Necessary conditions: Mutual Exclusion, Non-preemption & Circular wait)

issue : Deadlocks can significantly impact system performance and lead to complete system freeze if not properly handled.

* Starvation : It is a problem that occurs when higher priority processes keep executing & lower priority processes get blocked for indefinite time.

(Prevented by aging)

issues : In heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU time.

* Critical Section : A section of code within process that requires access to shared resources and that must not be executed while another process is in a corresponding section of code.
[It contains shared variables or resources that need to be synchronized to maintain consistency]

* Mutual Exclusion : The requirement when one process is in critical section that accesses shared resources, no other process may be in critical section that accesses any of those shared resources, no other processes may be in critical section that accesses any of those shared resources.

## ◊ Principles of Concurrency:

* **Process Isolation** - Each process should have its own memory space and resources to prevent interference between processes. This isolation is ~~necessary~~ critical to maintain system stability.

* **Synchronization** - Synchronization mechanisms like Locks, semaphores & mutexes are used to coordinate access to shared resources and ensure data consistency.

* **Deadlock Avoidance** - OSs implement algorithms to detect and avoid deadlock situations where processes are stuck waiting for resources indefinitely.

* **Fairness** - The OS should allocate CPU time fairly among processes to prevent any single process from monopolizing system resources

### Problems

| | |
|---|---|
| • Sharing global resources | Sharing of global resources is difficult. |
| • Optimal allocation of resources | It is difficult for O.S. to manage the allocation of resources optimally |
| • Locating programming errors | It is difficult to locate these errors as reports are usually not reproducible |
| • Locking the channel | It may be inefficient for the operating system to simply lock the channel and prevents its use by other resources. |

◇ Critical Section :

It must satisfy conditions
- Mutual exclusion
- Progress
- Bounded waitings.

Pseudocode

```
do {
    entry section

        critical section

    exit section

        remainder section
} while (TRUE);
```

When more than one process try to access same code segment that segment is known as the critical section. The critical section contains shared variables or resources that need to be synchronized to maintain the consistency of data variables.

To manage critical sections operating systems typically provide synchronization mechanisms that help enforce mutual exclusion.
Here are some common synchronization mechanisms used to implement critical sections:

* Mutex (Mutual Exclusion Lock)
* Semaphore
* Spinblock
* Condition Variable.

◇ **Race condition:** (It can be avoided by implementing Mutual Exclusion techniques such as locks, semaphores or monitors)

It occurs when multiple processes or thread read & write data items on relative timing of events and the final results depends on order of execution.

Example

* common race condition example is banking application where multiple users attempt to withdraw funds from the same amount simultaneously.
If not properly synchronized the balance may be incorrect due to simultaneous withdrawals.

---

◇ **Pipes & Types of Pipes:**

A pipe is a mechanism for IPC allowing communication between two processes. It provides a way for one process to send data to another process. (Used in Producer-consumer model)

There are two types:-

* **Unnamed Pipes:**

i) An unnamed pipe, also known as ordinary pipe, is a simple process form of inter-process communication that allows data to be transferred between two related processes.

ii) It is unidirectional meaning data flows in one directio
iii) Unnamed pipes are created using 'pipe()' system call.
int pipe (int *filedes)

filedes[0] = read operation
filedes[1] = write opr

* Named Pipe (FIFO):

i) A named pipe, also known as FIFO (First In, First Out), is a more flexible form of inter process.
ii) Named pipes can be used for communication b/w unrelated processes
iii) Named pipes are created using the 'mkfifo()' System call
iv) They provide file like interface, allowing processes to read & write data as if it were a regular file.

In typical producer - consumer scenario using pipes:

- The producer process writes data to the pipe.
- The consumer process reads data from the pipe.

---

Unit 5.2

## Mutual Exclusion:

Mutual Exclusion is a property of process synchronization that states no two processes can exist in the critical section at any given point of time.

### Requirements of Mutual Exclusion

i) At any time only one process is allowed to enter its critical section.
ii) The solution is implemented purely in software.
iii) A process remains inside its critical section for a bounded time only.

iv) A process cannot prevent any other process from entering into a critical section.
v) A process must not be indefinitely postponed from entering its critical section.
vi) No assumptions are made about relative speeds of processes or numbers of CPUs.
vii) No deadlock or starvation.

◇ Hardware & Software Approaches :

* Hardware Support : (It provides low-level mechanisms to enforce exclusive access to shared resources)

• Interrupt disabling • Special machine instruction

(i) Interrupt disabling -

Disabling interrupts on a processor prevents the processor from responding to external interrupts during execution of critical section.

disadvantages - • The efficiency of execution could be noticeably degraded.
• this approach will not work in a multiprocessor architecture.

(ii) Compare & Swap (Instruction) -

• It is also called as "compare and exchange instruction"
• A compare is made b/w a memory value & test value
• If the values are same a swap occurs.
• It is carried out atomically.

Instruction:

```
int compare_&_swap(int *word, int testval, int newval)
{
  int oldval;
  oldval = *word;
  if (oldval == testval)
  {
     *word = newval;
  }
  return oldval;
}
```

(iii) Special Test & Set (Instruction)

- The test and set instruction is a hardware supported operation that atomically sets a memory location to a specific value and returns its previous value.

```
Test_and_Set (lock)
{
original_value = *lock;   // read the current val
  *lock = 1;              // read set lock to 1
return original_value;    // Return original val
}
```

(iv) Bit-manipulation Instructions

Advantages of these Special machine instructions

- The Applicable to any number of processes on either a single processor or multiple processors sharing main memory.
- Simple & easy to verify.

Disadvantages:

- Busy-waiting is employed
- Starvation
- Deadlock.

* Software Support

Software approaches for mutual exclusion involving use of programming constructs, algorithms & synchronization primitives to ensure that only one process or thread can access a critical section of a code at any given time.

i) Locks / Mutexes
ii) Semaphores
iii) Spinlocks
iv) Read-write locks
v) Peterson's Algorithm -

It allows two processes to share a single resource without conflict using only shared memory for communication.

Algo

```
shared variables:
    int turn;          // indicates whose turn it is to enter critical section
    boolean flag[2];   // flags for indicating interest of each process

Process P0:
    flag[0] = true;    // express interest in entering critical section
    turn = 1;          // set the turn to other process
    while (flag[1] && turn == 1);
    {
        // Critical section
    }
    flag[0] = false;   // reset the interest flag to indicate
                       leaving the critical section

Process P1:
    flag[1] = true;    // express interest in entering critical condition
    turn = 0;          // set the turn to the other process
    while (flag[0] && turn == 0)
    {
        // Critical section
    }
    flag[1] = false;   // reset the interest flag to indicate
                       leaving the critical section
```

A process enters critical section only if the other process is not interested ('flag[other] == false').

After leaving the critical section, a process resets its 'flag' to 'false'.

## ◇ OS/Programming Language Support:

### (i) Semaphores:

In OS semaphores are a synchronization primitive used to control access to shared resources by multiple processes or threads.
It is an Integer variable & value of semaphore can be modified only by two functions wait() & signal() apart from initialization.

There are two categories

* **Counting Semaphore: (0 to ∞)**

In Counting semaphore, whenever process wants to access resources it performs wait() operation on semaphore & decrements value by 1.
When it releases resource, it performs signal() operation on the semaphore & increments the value by 1.

When it goes 0 which means all the resources of semaphores are occupied by processes.

* **Binary Semaphore: (0 & 1 only)**

In Binary semaphore, the value ranges b/w 0 & 1.
It is similar to mutex lock, but mutex is locking mechanism while this is signalling mechanism.

In binary semaphore if a process wants to access the resource it performs wait() operation on the semaphore and decrements value from 1 to 0.

Page No.
Date

## Advantages:

- Avoiding Race conditions.
- Coordination b/w processes.
- Flexible & robust to manage shared resources.
- Used to implement critical section in a program.
- used in mutual exclusion.

Nomenclature :- P(), ~~sem~~ wait(), down() → Entry section
V(), up(), signal() → Exit section.

Pseudocode

```
wait ( Semaphore S) → Entry
{
  S.value = S.value - 1
  if (S.value < 0)
  {
    Put // Put Current process in suspended list
  }
  else {
    // let Process enter critical section
  }
}
signal ( Semaphore S) → Exit
{
  S.value = S.value + 1;
  if (S.value ≥ 0)
  {
    // select process from wait list
    wakeup()
  }
}
```

# Semaphore Vs Mutex

| Basic Compare | Semaphore | Mutex |
|---|---|---|
| Basic | Signalling mechanism. It is used to signal other processes. | It determine locking of shared resource and if any process locks the resource, then no other processes is permitted to access it for read/write operations. |
| Value | Semaphore is an integer variable, binary. | Mutex is an object. |
| Operation: | It is modified using wait() & signal() operation. | Mutex object is locked or unlocked by process requesting or release of resource. |
| Function: | Semaphores allow multiple program threads to access a finite instance of resources simultaneously. | ~~Mutex object lock is released only by process~~ Mutex allow multiple threads to access a single resource single instance but not simultaneously. |
| Ownership: | Semaphore value can be changed by any process acquiring or releasing the resource. | Mutex object lock is released only by the process that has acquired lock on it. |

(ii) Monitors

Monitors are higher level synchronization construct that simplifies process synchronization by providing high-level abstraction for data access & synchronisation.
Monitors implemented in programming languages (Java, Modula-2) and provides mutual exclusion, condition variables & data encapsulation in single construct

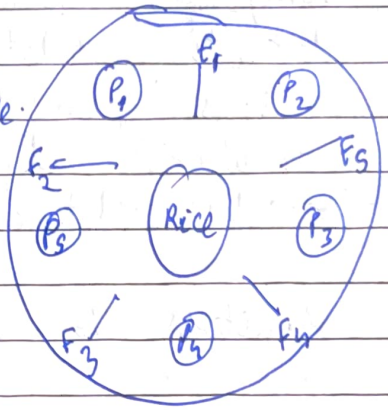
Monitors Vs Semaphore

| Basic Compare | Semaphore | Monitor |
|---|---|---|
| Definition | An integer value for controlling access. | An object/module that encapsulates shared data and operations |
| Signaling | No explicit signal/wait | Signal/wait using condition variables |
| Queue | External queue needed | Built in ready queue |
| Implementation: | Atomic operations like P() and V(). | enter/leave monitor, condition variables. |
| Performance | Lower | Higher |
| Structure | No relationships b/w data and operations. | Data & operations encapsulated together. |
| Blocking | Busy wait or sleep/wakeup. | sleep/wakeup using condition variables. |

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.

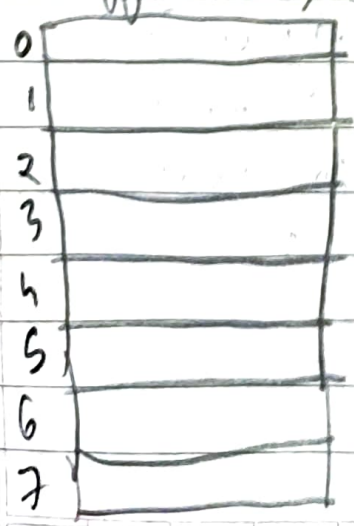
```
int count = 0;

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

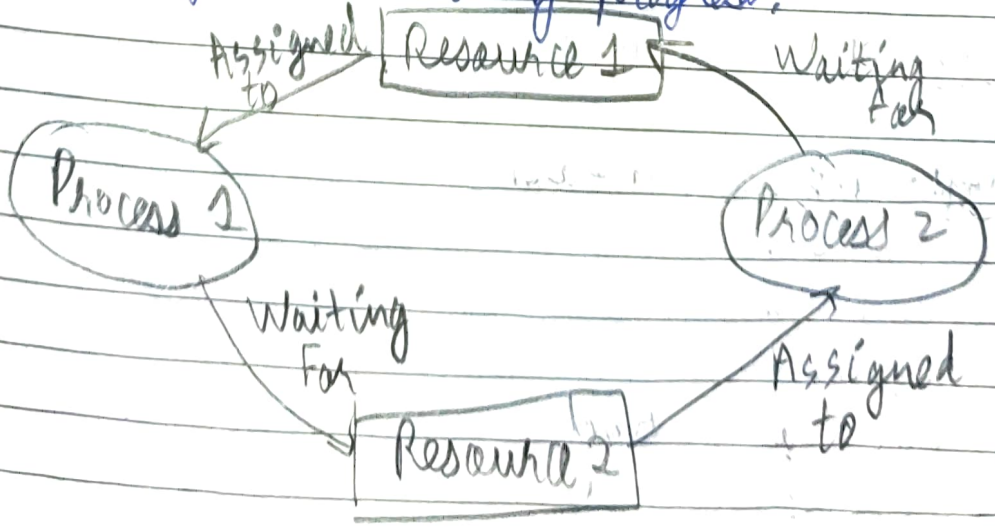
```
void Consumer (void)
{
    int item c;
    while (TRUE)
    {
        while (count == 0);
        item c = Buffer (out);
        Out = (Out + 1) mod n;
        Count = count - 1;
        Process_item (item);
    }
}
```

```
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.

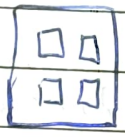
(iii) **No-Preemption**: Resources cannot be forcibly taken away from process they must be ~~removed~~ released voluntarily by the process holding them.

(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.

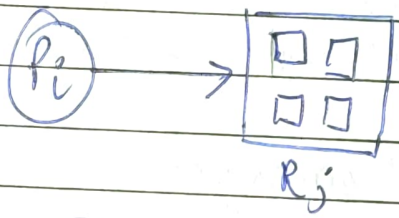
## V) Resource Allocation Graphs:

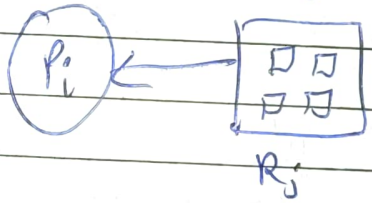
A graph model known as RAG can be used to represent the relationships b/w processes & resources. It helps in visualizing the potential for deadlocks.

Process ◯

Resource type (4 instances) 

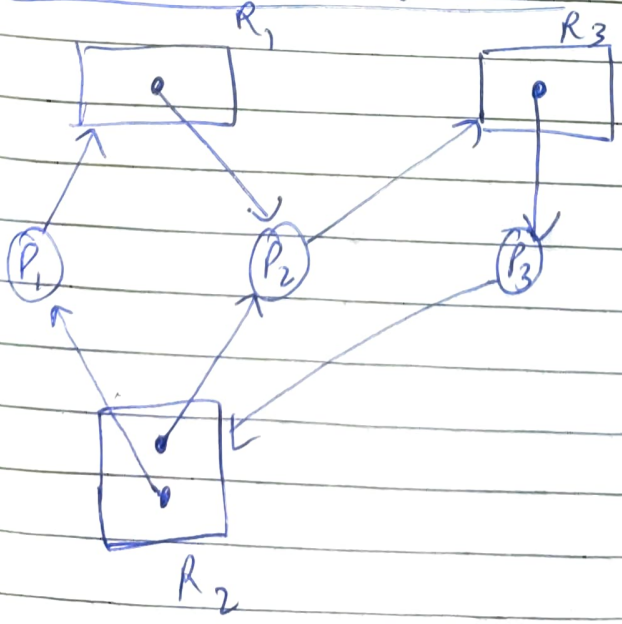
$P_i$ requests instance of $R_j$



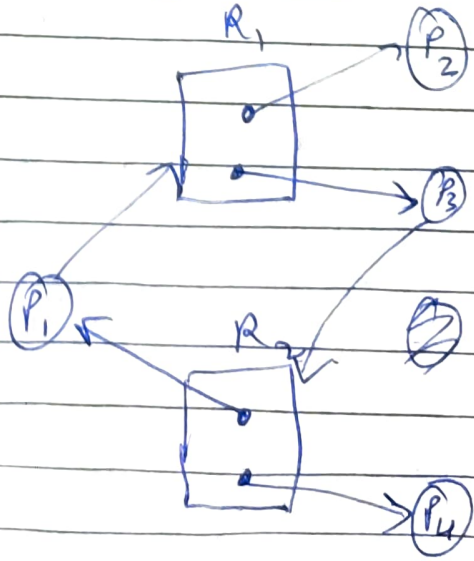




$P_i$ holding an instance of $R_j$

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 :-

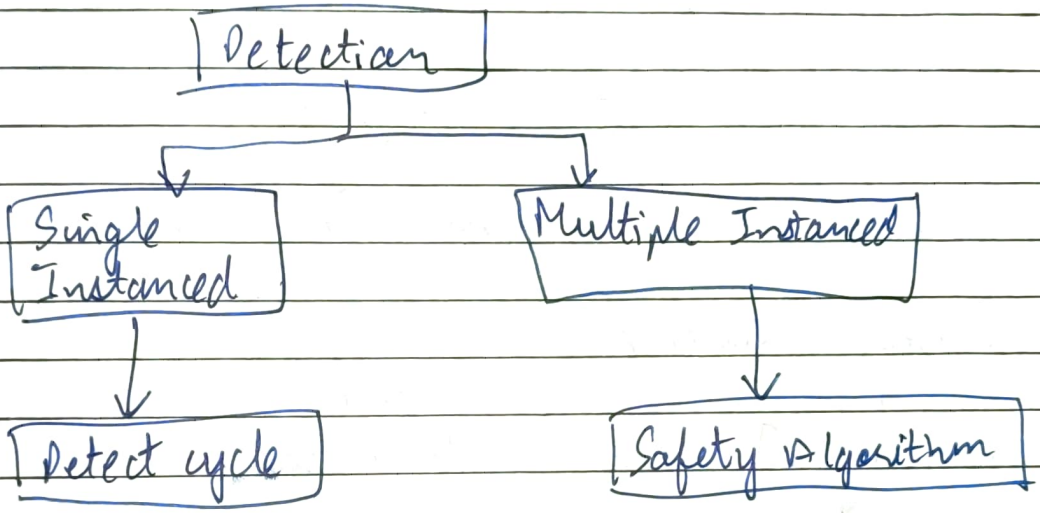
let n = no of processes
m = no of resource types

- 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

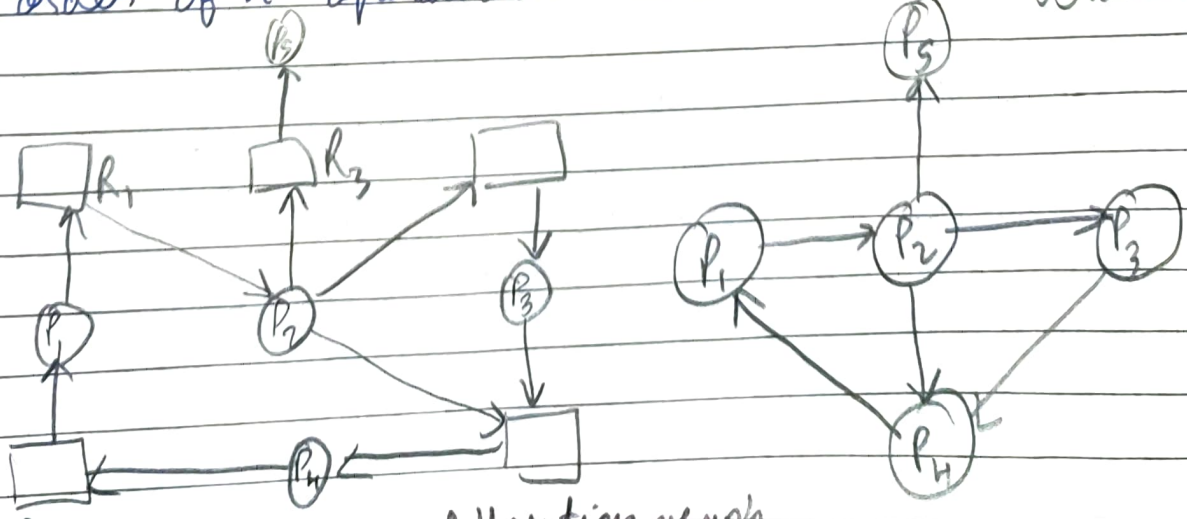
- Deadlock detection involves determining if a deadlock has occured by analyzing state of the system.
- It allows deadlock to occur, then detects it and initiates recovery.




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

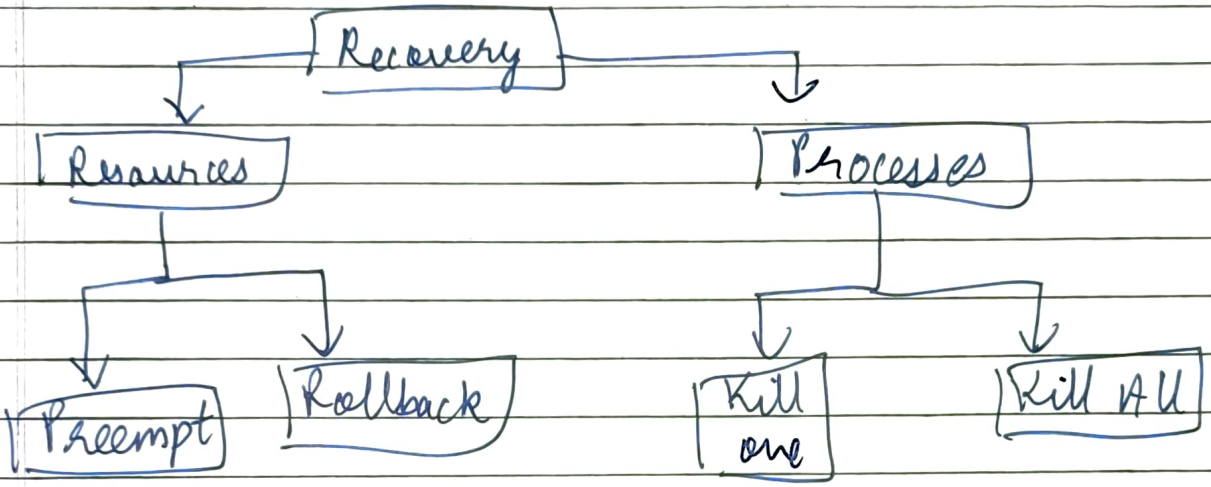
(iii) Work = Work + Allocation$_i$
Finish [i] = true

(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.