Unit-2

Process Synchronization

Critical Section → Critical section is a portion of program text where the shared resources or shared variables will be placed.

Consider a system consisting of n processes $\{P_0, P_1, P_2 \ldots P_{n-1}\}$. Each process has a segment of code, called a critical section, in which the process may be changing common variables, updating a table, writing a file and so on.

The important feature of the system is that, 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 other processes is executing in their critical section at the same time.

The critical section problem is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section. The section of code implementing this request ~~permission~~ is the entry section. The critical section may be followed by exit section. The remaining

code is the remainder section.

The general structure of a typical process P_i is shown below

```
do
{
    entry section
        critical section
    exit section
        remainder section
} while (true);
```

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 section.

(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 deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.

↑ max. bound on each process to wait before it will get a chance to enter CS after that

(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.

Race Condition → When order of execution can change then it is the result is called race condition.

Process Synchronization

What is the need of process synchronization

→ As we know we use multiprogramming operating system so many processes are reside in main memory. Suppose we have two resources R_1, R_2 both are sharable. It doesn't mean that resources are shared it can use in sharable mode.

Printer is a shared resource and diff" process can use printer at different instance of time.
If all the process will use printer will use at the same time. It may lead inconsistency.

So we must have some ordering scheme which will decide which process use resource at which instance of time.

Date / /
Page

Solution for synchronization

→ S/w support
- ① Lock variable
- ② Peterson solution
- ③ Strict alternation or Decker Algorithm

→ H/w support
- TSL instruction set
- ② Disabling interrupts

→ OS support
- ① Semaphores → Binary
  - Counting

→ Compiler support
- Monitors

shared = 5

P_1	P_2
X = shared	Y = shared
X++;	Y--;
sleep(1)	sleep(1);
shared = X;	shared = Y;
⑥	④

Race Condition

Q→ How many diff^n values you can get for B if the below two process in any order.

```
P1()                    P2()
{                       {
① C = B - 1             ③ D = 2 * B;
② B = 2 * C             ④ B = D - 1;
}                       }
```

B is a shared variable with initial value 2

a) 3 b) 2 c) 5 d) 4

P1 ① ②	P2 ③ ④				
C = 1	3 4 1 2	1 3 2 4	3 1 2 4	3 1 4 2	1 3 4 2
B = 2*1 = 2	D = 2*2 = 4	C = 1	D = 2*2 = 4	D = 4	C = 1
D = 4	B = 4-1 = 3	D = 2*2 = 4	C = 2-1 = 1	C = 1	D = 4
B = 3	C = 3-1 = 2	B = 2*1 = 2	B = 2	B = 3	B = 3
	B = 2*2 = 4	B = 4-1 = 3	B = 4-1 = 3	B = 2	B = 2

Using Turn Variable

P_0	P_0	P_1
entry section	while(1)	while(1)
	{	{
critical sec^n	~~while~~ (turn != 0);	while (turn != 1);
exit section	Critical section	critical section;
remainder section	turn = 1;	turn = 0;
	remainder section	remainder section;
	}	}

Initially turn = 0

P_0: while(0 == 1) false → critical section; then preempt

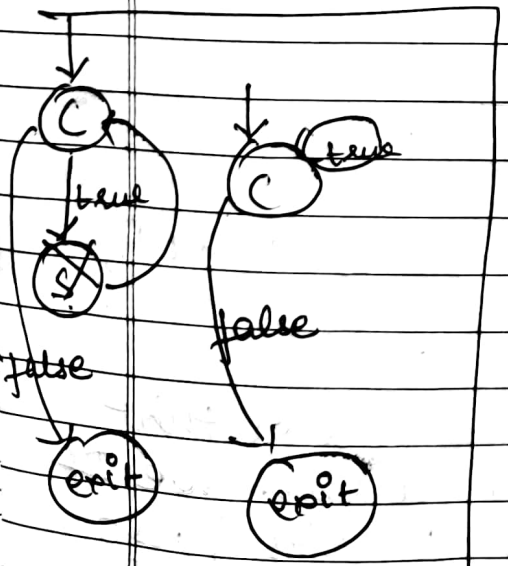
P_1: while(0 == 0) true (so it execute again & again)

(1) (2) (3)

P_0: turn = 1; remainder section; }

then we check P_0 is able to go execute again then check (1 == 1) true so it will not go again in CS. then chances is for P_1

P_1: while(1 == 0) false → CS. turn = 0;



(1) Mutual Exclusion satisfy

(2) & Progress, ~~we want~~ only those processes compete which actually want to enter in CS. But here Charge is given to $P_0 \rightarrow P_1$.

But if P_1 is doesn't want to go in CS, coe don't ask from P_1 then here no process is there.

Date / Page

Using Flag Variable

P_0	P_1
while (1)	while (1)
{	{
flag [0] = T	flag [1] = T
while (flag [1]);	while (flag [0]);
critical section;	critical section;
flag [0] = F	flag [1] = F
}	}

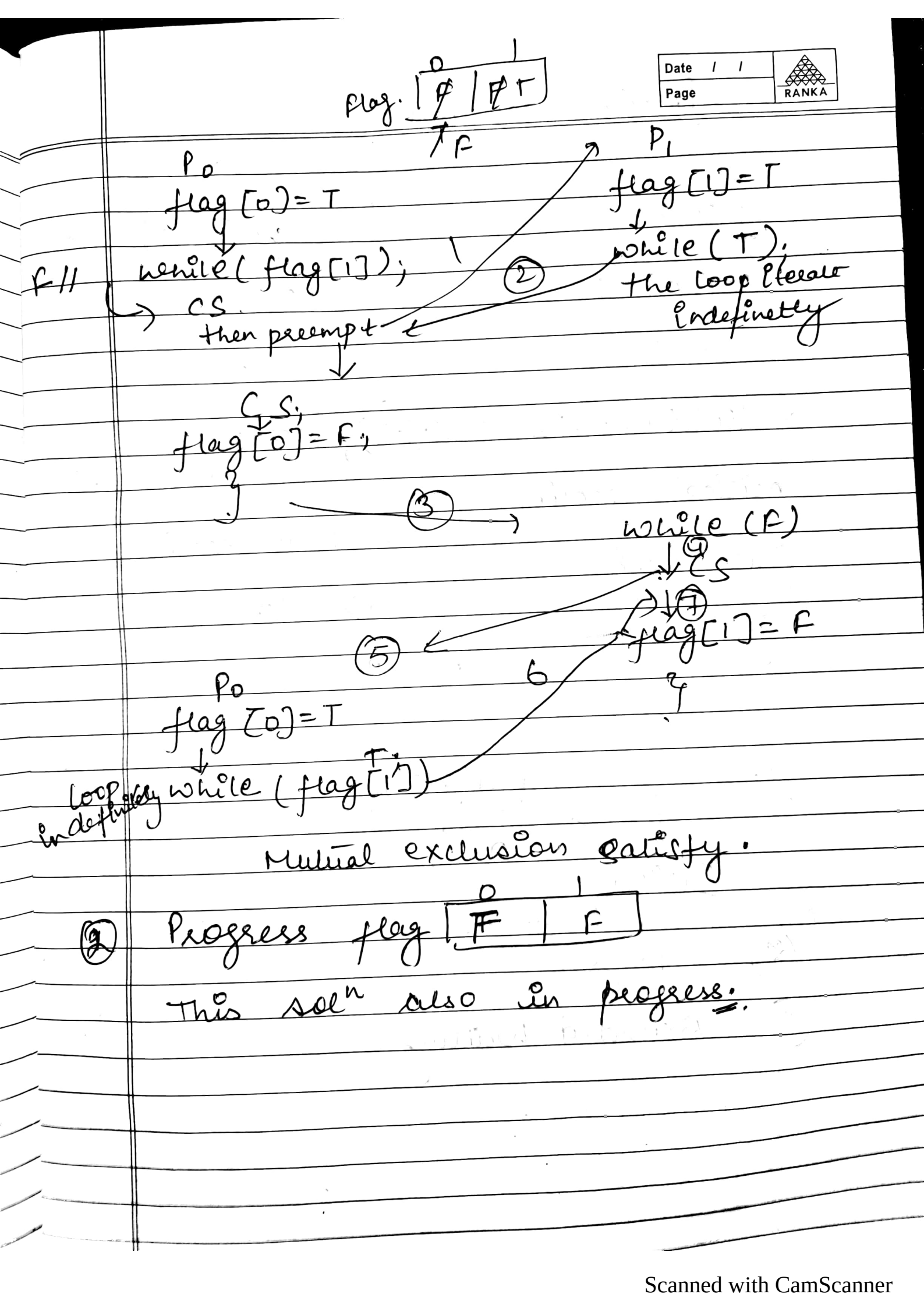
Initially flag

0	1
F	F

when flag [0] = T it means that P_0 wants to enter in CS.

while (flag [1]); // we check that flag [1] is true or false in array. If it is true it means process P_1 also want to enter into CS. In this case, process P_0 doesn't enter into cs because while became true and loop iterate endefinitely.

Date / /
Page



Mutual exclusion satisfy.

(2) Progress flag [F | F]

This soln also in progress.

Peterson Solution

P_0

```
while(1)
{
flag[0] = T
turn = 1
while (turn == 1 && flag[1] = T);
critical section
flag[0] = F
}
```

P_1

```
while(1)
{
flag[1] = T
turn = 0
while (turn == 0 &&
       flag[0] == T);
critical section;
flag[1] = F
}
```

turn = 0/1

flag: [F | F]
0 1

Satisfy

1. Mutual Exclusion
2. Progress
3. Bounded Waiting

Consider the code used by the processes P and Q for accessing their CS. The initial values for shared Boolean variables S and T are false —

Code of P	Code of Q
while (S == T);	while (S != T);
CS	CS
S = not(T);	S = T;

a) Code will violate the Mutual exclusion
b) Process P can go into CS multiple times w/o the single entry of Q into CS
c) Process Q can go into the CS after exactly one entry by Process P into its CS
d) None of these.

Appⁿ of semaphores i) to decide the order of procedure execution of procedures.
ii) Resource Mgt iii) solve critical section problem

Semaphores (it gives n process solution)

A semaphore is an integer variable that apart from initialization is accessed only through two standard operations:

int S

it is atomic operation which reduces value by 1

i) wait(S)

```
wait(S)
{
while(S<=0);
  S=S-1;
}
```

ii) signal(S)

```
signal(S)
{
S=S+1;
}
```

Whenever we solve CS problem then we always initialize S=1 and after that we can't directly access it. We can access through standard Operations —

i) wait() or P() ii) signal() or V() or up

```
P1
do
{
entry section
  critical section
exit section
remainder section
} while(T)
```

```
P1
do
{
wait(S);
// critical section
signal(S);
remainder section
} while(T)
```

or

Semaphore is an integer variable which is used in mutual exclusive manner by various concurrent cooperative processes in order to achieve synchronization.

Semaphores are of two types

i) Counting (-∞ to ∞)

ii) Binary (mutex locks) (0 to 1)

(1) Binary Semaphore - This is also known as mutex lock. It can have only two values -> 0 & 1. Its value is initialized to 1. It is used to implement the solution of critical section problem with multiple processes.

(2) Counting Semaphore — Its value can range over an unrestricted domain. It is used to control access to a resource that has multiple instances.

Counting Semaphore

```
Down (Semaphore S)
  {

S.value = S.value - 1
if (S.value < 0)
  {
if
put process (PCB)
in suspend list, sleep()
  }

else
  return;
  }

  up (Semaphore S)
    {
S.value = S.value + 1
if (S.value <= 0)
  {
Select a process
  from suspend list && wake up();
  }

}
```

Date / /
Page

RANKA

Binary Semaphore

```
Down ( Semaphore S )
  {
if (S.value == 1)
  {
S.value = 0;
  }
else
  {
Block this process and place in
  suspend list, sleep();
  }
}

up ( Semaphore S )
  {
if ( suspend list is empty )
  {
S.value = 1
  }
else
  {
Select a process from suspend list
  and wakeup();
  }
}
```

Classical Synchronization Problem

Reader-writer problem

The reader-writer problem relates to an object such as a file that is shared between multiple processes. Some of these processes are readers i.e. they only want to read the data from the object and some of the processes are writers i.e. they want to write into the object.

The reader-writer problem is used to manage synchronization so that there are no problems with the object data. However if two writers or a reader and writer access the object at the same time, there may be problems.

To solve this situation, a writer should get exclusive access to an object i.e when a writer is accessing the object, no reader or writer may access it. However, multiple readers can access the object at the same time.

This can be implemented using binary semaphores. The code for the reader and writer process in the reader-writer problem are given as follows —

For writer

```
do
{
wait(wrt)
write opn
signal(wrt)
}
```

For reader

```
wait(mutex)
readcount++
if(readcount==1)
wait(wrt)
signal(mutex)
read operation
wait(mutex)
readcount--
if(readcount==0)
signal(wrt)
signal(mutex)
```

Initially mutex = 1 readcount = 0
wrt = 1

If a writer wants to access the object, wait operation is performed on wrt. After that no other writer can access the object. When a writer is done writing into the object, signal operation is performed on wrt.

In the above code, mutex semaphore ensures mutual exclusion and wrt handle the writing mechanism and is common between reader & writer process. The variable rc denote the number of readers accessing the object. As soon as rc becomes 1, wait operation is used on wrt. This means that a writer cannot access the object anymore. After the read opn is done, rc is --. When rc becomes zero, signal is used on wrt. So writer can access the object now

opn is used on wrt.

Dining Philosophers Problem

The DPP states that there are 5 philosophers sharing a circular table and they eat and think alternatively. There is a bowl of rice for each of the philosopher and 5 chopsticks. A philospher needs both their right and left chopstick to eat. A hungry philospher may not eat if there are both chopsticks are not available otherwise a philospher puts down their chopstick and begin thinking again.

Solution of Dining Philospher Problem

A solution of the dining philospher problem is to use a semaphore to represent the chopstick. A chopstick can be picked up by executing a wait operation on the semaphore and released by executing a signal semaphore.

The structure of the chopstick is shown below —

```
Semaphore chopstick [5];
```

Initially the elements of the chopstick are initialized to 1 as the chopsticks are on the table and not picked up by a philospher.

```
do
{
wait ( chopstick[i] );
wait ( chopstick[(i+1)%05] );
  eating;
signal ( chopstick[i] );
signal ( chopstick[(i+1)%05] );
  thinking;
} while(1);
```

In the above structure, first wait operation is performed on chopstick[i] and chopstick[(i+1)%05]. This means that the philosphes i has picked up the chopsticks on his sides. Then the eating function is performed.

After that, signal operation is performed on chopstick[i] and chopstick[(i+1)%05]. This means that the philosphes i has eaten and put down the chopsticks on his sides. Then the philosphes goes back to thinking.

Difficulty with the solution

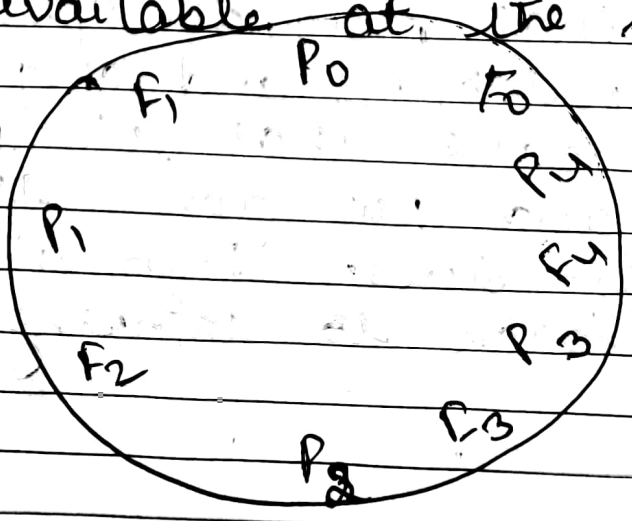
The above solution makes sure that no two neighbouring philosphers can eat at the same time. But this solution can lead to a deadlock. This may happen if all the philosphers pick their left chopstick simultaneously. Then none of them can eat and deadlock occur.

Some of the ways to avoid deadlock are as follows:-

→ There should be atmost four philosphers on the table.

→ An even philosphers should pick the right chopstick and then left chopstick while an odd philosphers should pick the left chopstick and then the right chopstick.

→ A philosphers should only be allowed to pick their chopstick if both are available at the same time.



Non-preemptive scheduling → It does not interrupt the process running CPU in middle of the execution. Instead, it waits till process complete its CPU burst time & then it can allocate the CPU to another process.

CPU scheduling

Arrival time — The time at which process enter the ready queue or state.

Burst time :— Time required by a process to get execute on CPU.

Completion time :— The time at which process complete its execution.

Turnaround time — The interval from the time of submission of a process to the time of completion.
TAT = Completion time − arrival time

Waiting time → TAT − Burst Time

Response time → The time at which a process get CPU first time.

1) FCFS (First-Come, First Serve Scheduling)

With this scheme, the process that requests the CPU first is allocated the CPU first. The implementation of FCFS policy is easily managed with a FIFO queue.

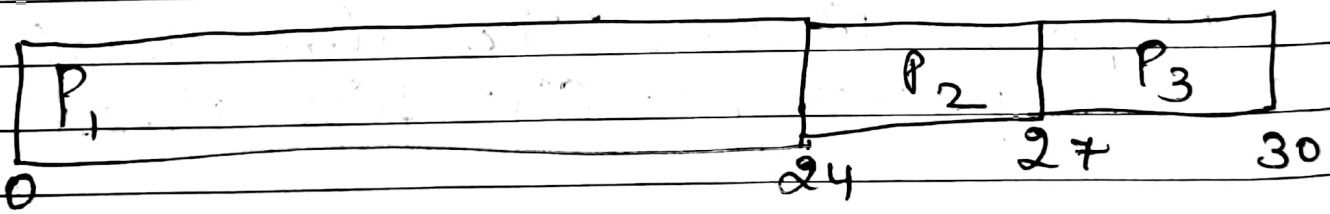
→ Gantt chart, which is a bar chart that illustrates a particular schedule

including the start and finish times of each of the particular processes.

→ FCFS scheduling algorithm is non preemptive. Once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O.

Process	Burst time
P_1	24
P_2	3
P_3	3

Calculate average waiting time of process.



Turnaround time = CT − AT

" P_1 = 24 − 0 = 24

" P_2 = 27 − 0 = 27.

" P_3 = 30 − 0 = 30

Waiting time P_1 = TAT − BT

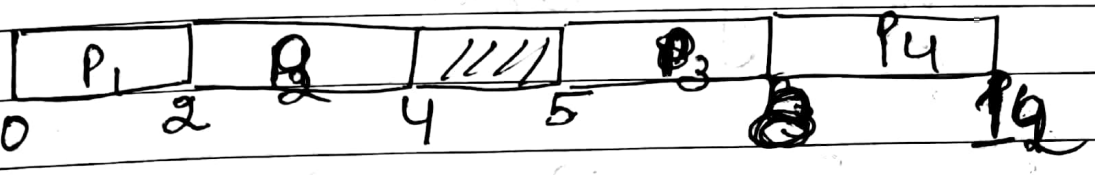
= 24 − 24 = 0

P_2 = 27 − 3 = 24

P_3 = 30 − 3 = 27

Avg waiting time = (0+24+27)/3 = 17 msec.

	AT	BT	CT	TAT = CT - AT	WT = TAT - BT
P_1	0	2	2	2	0
P_2	1	2	4	3	1
P_3	5	3	8	3	0
P_4	6	4	12	6	2



Avg TAT = (2+3+3+6)/4 = 14/4 = 3.5 msec.

Avg WT = (0+1+0+2)/4 = 3/4 = 0.75 ms

SJF (Shortest Job First Scheduling)

In the SJF, if the CPU is available, it is assigned to the process that has the smallest next CPU burst. If the next CPU bursts of two processes are same, FCFS scheduling is used to break the tie.

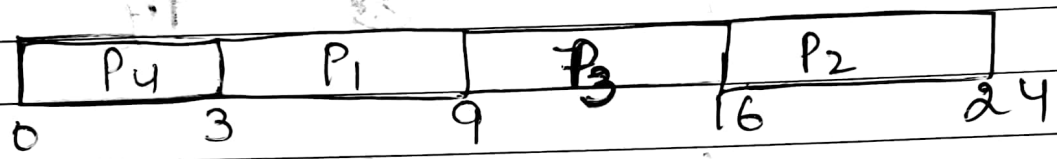
Note that more appropriate term of this scheduling method would be the shortest next CPU burst algorithm.

SJF

Q-

Process	Burst Time	CT	TAT	WT = TAT - BT
P_1	6	9	9	3
P_2	8	24	24	16
P_3	7	16	16	9
P_4	3	3	3	0

Calculate average turnaround time & waiting time.



Avg waiting time = (0+9+16+3)/4 = 7 msec.

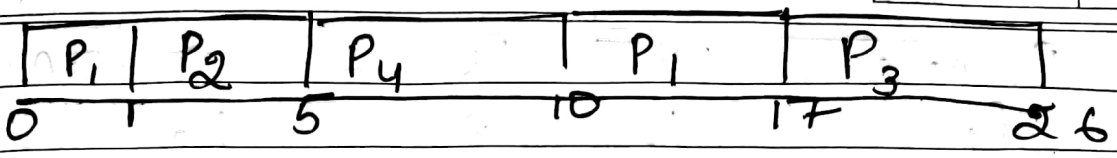
Avg TAT = (9+24+19)/4 = 52/4 = 13 msec.

SJF algorithm may be preemptive and non-preemptive as well.

Preemptive SJF scheduling is sometimes called shortest-remaining time-first scheduling.

Eg→

Process	AT	BT	CT	TAT = CT - AT	WT = TAT - BT
P_1	0	8	17	17	9
P_2	1	4	5	4	0
P_3	2	9	26	24	15
P_4	3	5	10	7	2



Avg TAT = (17+4+24+7)/4 = 52/4 = 13 msec

Avg WT = (9+0+15+2)/4 = 26/4
= 6.5 msec.

Priority Scheduling

SJF - algorithm is a special case of the general priority scheduling algorithm.

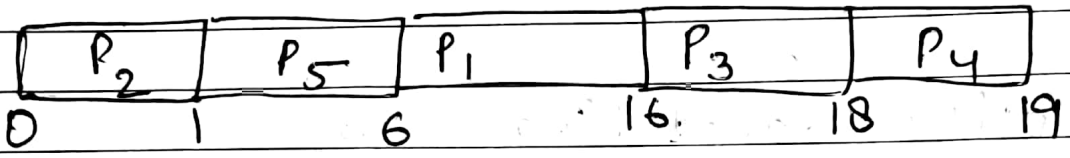
→ A priority is associated with each process, and the CPU is allocated to the process with the highest priority.

→ Equal priority processes are scheduled in FCFS order.

→ An SJF algorithm is simply a priority algorithm where the priority (P) is the inverse of the next CPU burst. The larger the CPU burst, the lower the priority and vice versa.

Non-Preemptive Priority Scheduling (smallest no. highest priority)

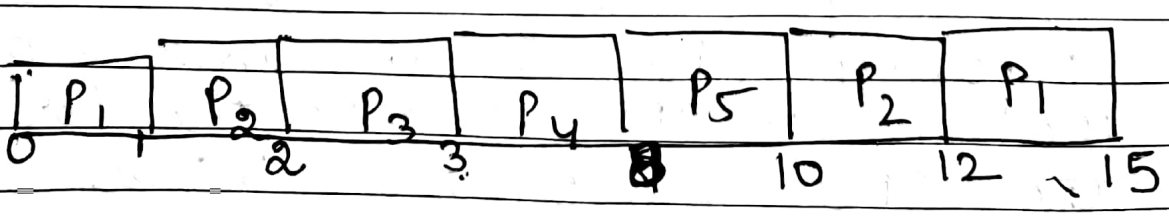
Process	BT	Priority	CT	TAT	WT = TAT - BT
P_1	10	3	16	16	6
P_2	1	1	1	1	0
P_3	2	4	18	18	16
P_4	1	5	19	19	18
P_5	5	2	6	6	1



Avg WT = (6 + 0 + 16 + 18 + 1) / 5
= 41/5 = 8.2 msec.

Preemptive Priority Scheduling (largest no. highest priority)

	AT	BT	Priority	CT	TAT = CT - AT	WT = TAT - BT
P_1	0	~~10~~ 3	2	15	15	11
P_2	1	~~8~~ ~~1~~ 0	3	12	11	8
P_3	2	~~2~~ 0	4	3	1	0
P_4	3	~~1~~ 0	5	8	5	0
P_5	4	~~5~~ 0	5	10	6	4



Avg TAT = (15 + 11 + 1 + 5 + 6) / 5 = 38/5 = 7.6 msec

Avg WT = (11 + 8 + 4) / 5 = 23/5 = 4.6 msec

Problem in Priority Scheduling

Starvation → A process that is ready to run but waiting for the CPU can be considered blocked. A priority scheduling algorithm can leave some low priority processes waiting indefinitely.

A solution to a problem is aging.

Gradually increasing the priority of a process waiting for long.

Round-Robin Scheduling

It is designed especially for time-sharing systems. It is similar to FCFS scheduling but preemption is added to enable the system to switch between processes.

A small unit of time, called a time quantum or time slice, is defined.

A time quantum is generally from 10 to 100 msec in length.

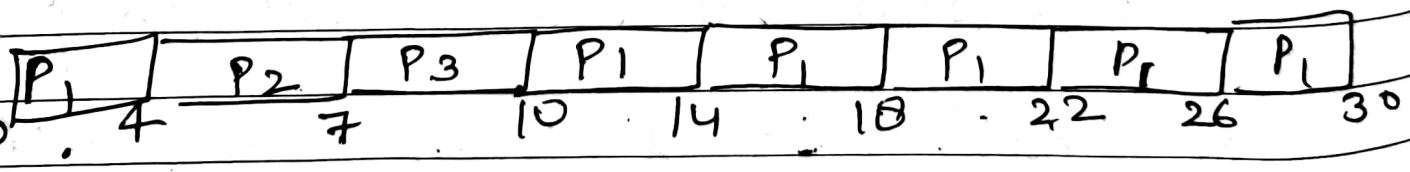
The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of upto 1 time quantum.

To implement RR scheduling, we keep the ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum and dispatches the process.

Process	Burst Time	CT	TAT = CT - AT	WT = TAT - BT
P_1	~~24~~ 20	30	30	6
P_2	~~3~~ 0	7	7	4
P_3	~~3~~ 0	10	10	7

time quantum = 4 msec

P_1



$$\text{Avg WT} = \frac{(6+4+7)}{3} = \frac{17}{3} = 5.66 \text{ msec}$$

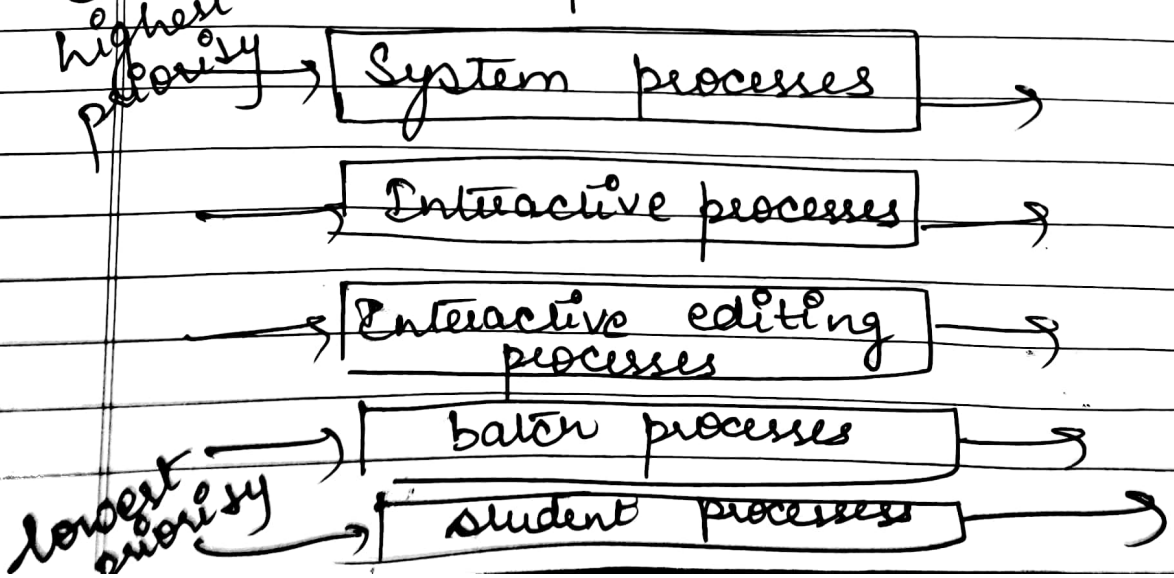
Multilevel Queue & Multilevel Feedback Queue Scheduling

A multilevel queue scheduling algorithm partitions the ready queue into several seperate queues. The processes are permanently assigned to one queue generally based on some property of the process. For ex→ seperate queues might be used for foreground and background processes.

The foreground queue might be scheduled by an RR algorithm while the background queue is scheduled by an FCFS algorithm.

Multilevel queue scheduling algorithm with five queues—

1. System processes
2. Interactive processes
3. Interactive editing processes
4. Batch processes
5. Student processes



Each queue has absolute priority over lower priority queues. No process in the batch queue for example — could run unless the queues for system processes, interactive processes and interactive editing processes were all empty. If an interactive editing process entered the ready queue while a batch process was running, the batch process would be preempted.

Another possibility is to time-slice among the queues. Here, each queue gets a certain portion of the CPU time, which it can then schedule among its various processes. For ex, in foreground-background queue ex, the foreground queue can be given 80 percent of the CPU time for RR scheduling among its processes, whereas the background queue receives 20 percent of the CPU to give to its processes on a FCFS basis.

Multilevel Feedback Queue Scheduling

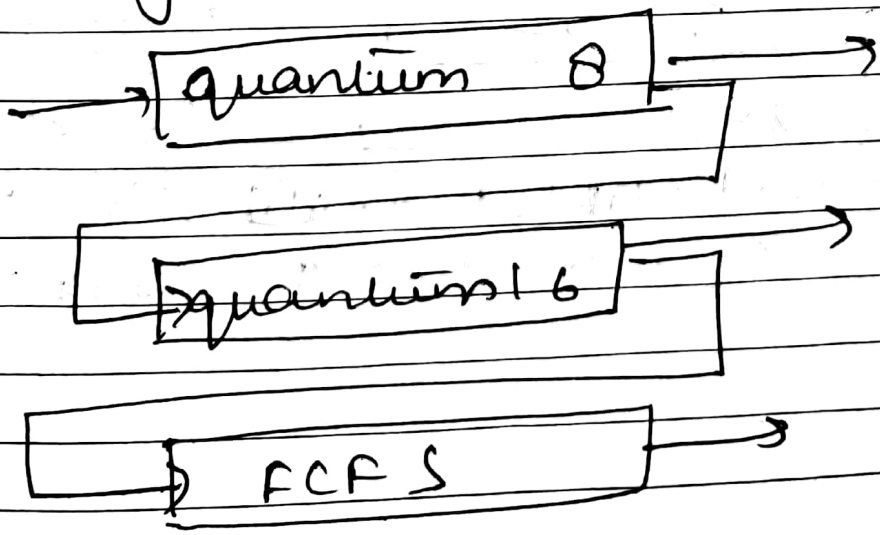
When the multilevel queue scheduling algo is used, processes are permanently assigned to a queue when they enter the system. If there are seperate queues for foreground and background processes do not change their foreground and background nature.

In contrast, multilevel feedback queue scheduling algorithm, in contrast, allows a process to move b/w queues. The idea is to seperate processes according to the characteristics of their CPU bursts. If a process uses too much CPU time, it will be moved to a lower priority queue. This schemes leaves I/O bound and interactive processes in the higher priority queues. In addition, a process that waits too long in a lower-priority queue may be moved to a higher priority queue. This form of aging prevents starvation.

For ex → Consider a multilevel feedback queue scheduler with three queues numbered from 0 to 2. The scheduler first executes all processes in queue 0. Only when queue 0 is empty will it execute processes in queue 1.

Similarly, processes in queue 2 will only be executed if queues 0 and 1 are empty. A process that arrives for queue 1 will preempt a process in queue 2. A process in queue 1 will in turn be preempted by a process arriving for queue 0.

A process entering the ready queue is put in queue 0. A process in queue 0 is given a time quantum of 8 msec. If it doesn't finish within this time, it is moved to the tail of the queue. If queue 0 is empty, the process at the head of queue 1 is given a quantum of 16 msec. If it doesn't complete, it is preempted and is put into queue 2. Processes in queue 2 are run on an FCFS basis but are run only when queues 0 & 1 are empty.



Rate Monotic Scheduling

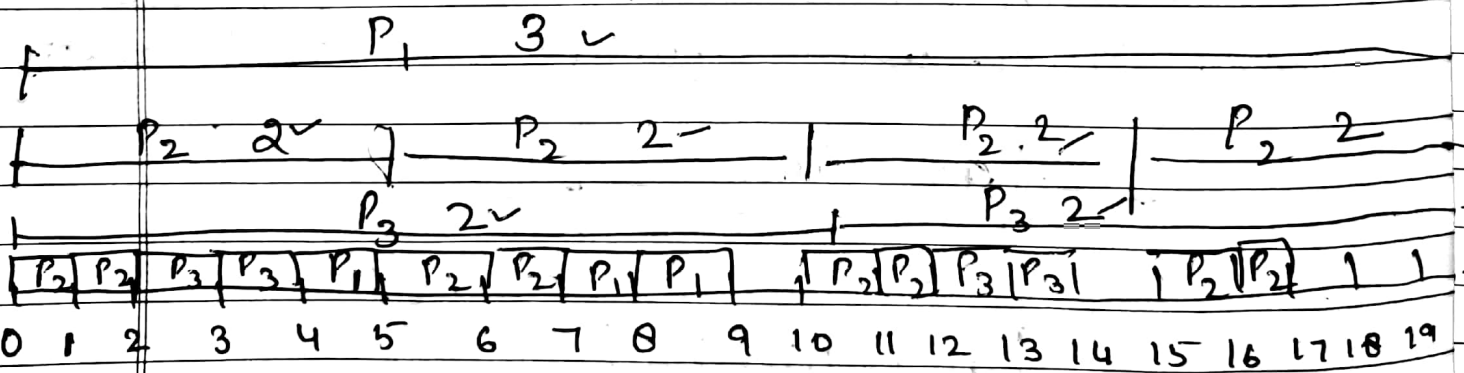
Process	Capacity	Period
P_1	3	20
P_2	2	5
P_3	2	10

LCM (20, 5, 10) = 20

Priority is decided based on periods

$P_2 > P_3 > P_1$

(period is low then it has highest priority).

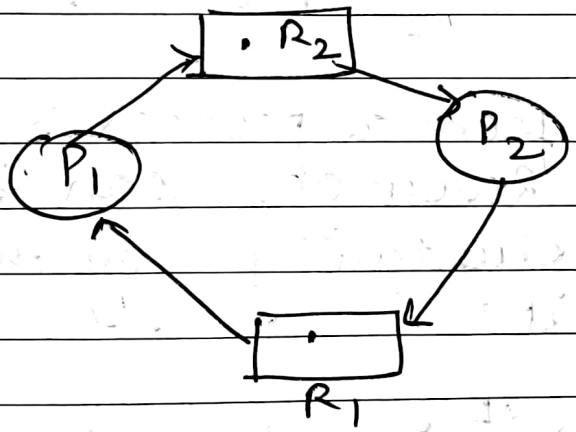


Date / /
Page
RANKA

Deadlock

In a multiprogramming system, a number of process compete for limited number of resources and if a resource is not available at that instance then process enters into a waiting state.

If a process unable to change its waiting state indefinetly because the resource is requested by it are held by another waiting process, then it is said to be in deadlock.



System model

Every process will request for the resource
→ if entertained, then process will use the resource.
→ Process must release the resource after use.

Date / /
Page
RANKA

Necessary condition Of Deadlock

(1) **Mutual exclusion**: At least One resource type in the system which can be used in non-Sharable mode i.e. mutual exclusion (one at a time / one by one) eg printer.

(2) **Hold & wait**:→ A process is currently holding atleast one resource and requesting addition resources which are being held by other processes.

(3) **No-preemption**→ Resource cannot be preempted from a process by another process. Resources can be released only voluntarily by the process holding it.

(4) **Circular wait**:→ Each process must be waiting for a resource which is being held by another process which in turn is waiting for the first process to release the resource.

Date / /
Page

RANKA

Deadlock handling methods

(1) Prevention → means design such a system which violate atleast one of four necessary conditions of deadlock and ensure independence from deadlock.

(2) Avoidance: System maintains a set of data using which it takes a decision whether to entertain a new request or not, to be in safe state.

(3) Detection and recovery → Here we wait until deadlock occurs and once we detect it & we recover from it.

(4) Ignorance or ostrich algo → we ignore the problem as if it doesn't exist.

Deadlock prevention method :-

For a deadlock, each of the four necessary conditions must hold. By ensuring that atleast one of these conditions cannot hold, we can prevent the occurrence of a deadlock.

(1) Mutual Exclusion → Mutual exclusion condition must hold for non-sharable resources. For eg → a printer cannot be simultaneously shared by several processes. Sharable resources, in contrast, do not require mutually exclusive access and thus cannot be involved in a deadlock.
A process never needs to wait for a sharable resources.
We cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources are Intrinsically non-Sharable.

(2) Hold & Wait :— Process is allowed to start execution if and only if it has acquired all the resources.
(less efficient, not implementable, easy, deadlock independence).

Donot hold → Process will acquire only desired resources, but before making any fresh request it must release

all the resources that is currently held.

Wait timeouts → We place a maximum time upto which a process can wait after which process must release all the holding resource & exit.

(3) No- Preemption →

Forceful Preemption → We allow a process to forcefully preempt the resource holding by other processes.

→ This method may be used by high priority process or system process.

→ Processes which are in waiting state must be selected as a victim processes in the running state.

(4) Circular Wait :-

Circular wait can be eliminated by first giving a natural number of every resource $f: N \to R$

→ allow every process to either only in increasing or decreasing order of the resource number.

Date / /
Page

→ If a process require a lesser number (in case of increasing order) than it must first release all the resource larger than required number.

Resource Allocation Graph

Deadlock can be described more precisely in terms of a directed graph called system resource allocation graph.

This graph consists of a set of vertices V and a set of edges E. This set of vertices V is partitioned into two different types of nodes : $P = \{P_1, P_2 - - P_n\}$, the set consisting of all the active processes in the system, and $R = \{R_1, R_2 - R_n\}$ the set consisting of all resource types in the system.

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 the resource.

A directed edge from resource type R_j to process P_i is denoted by $R_j \rightarrow P_i$; it signifies that an instance of resource type R_j has been allocated to process P_i.

A directed edge $P_i \rightarrow R_j$ is called a request edge and a directed edge $R_j \rightarrow P_i$ is called an assignment edge.

Pictorially, we represent each process P_i as a circle and each resource Type R_j as a rectangle. Since resource type R_j may have more than one instance, we represent each such instance as a dot within the rectangle.

When process P_i requests an instance of resource type R_j, a request edge is inserted in the RAG. When this request can be fulfilled, the request edge is instantaneously transformed to an assignment edge. When the process no longer needs access to the resource, it releases the resource, as a result, the assignment edge is deleted.

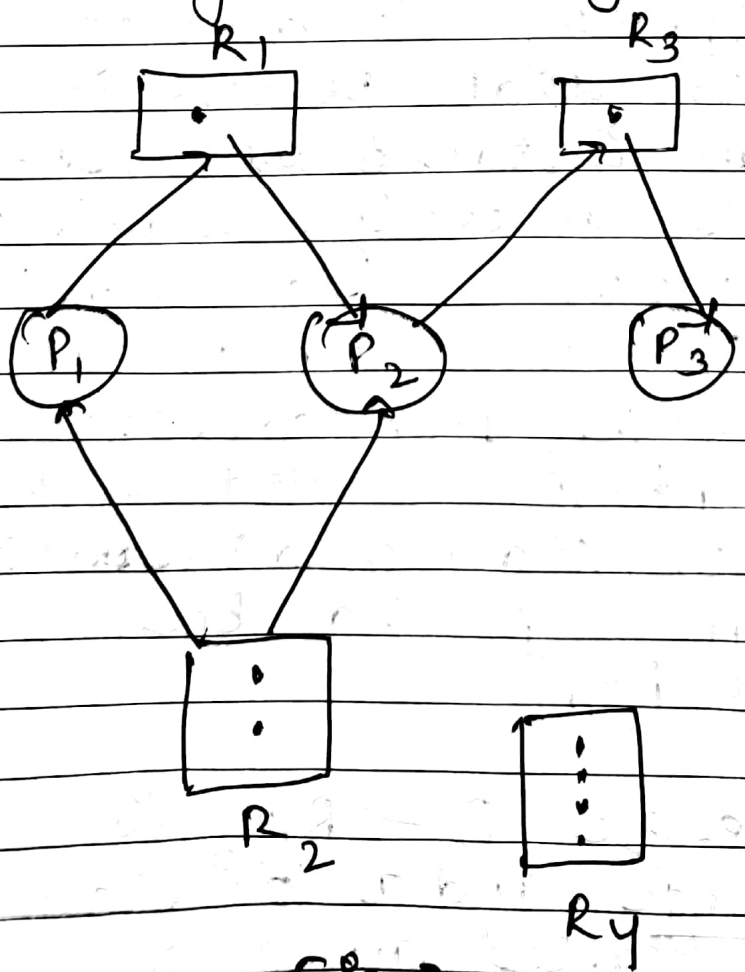


Fig 2

Given the definition of a RAG, it can be shown that, if the graph contains no cycles, then no process in the system is deadlocked.

If the graph does contain a cycle, then a deadlock may exist.

If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred.

If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred.

To illustrate this concept, we return to RAG depicted in previous example.
Fig 2. Suppose that process P_3 requests an instance of resource type R_2.
Since no resource instance is currently available, a request edge $P_3 \rightarrow R_2$ is added to the graph.
At this point, two minimal ~~cost~~ cycles exist in the system:

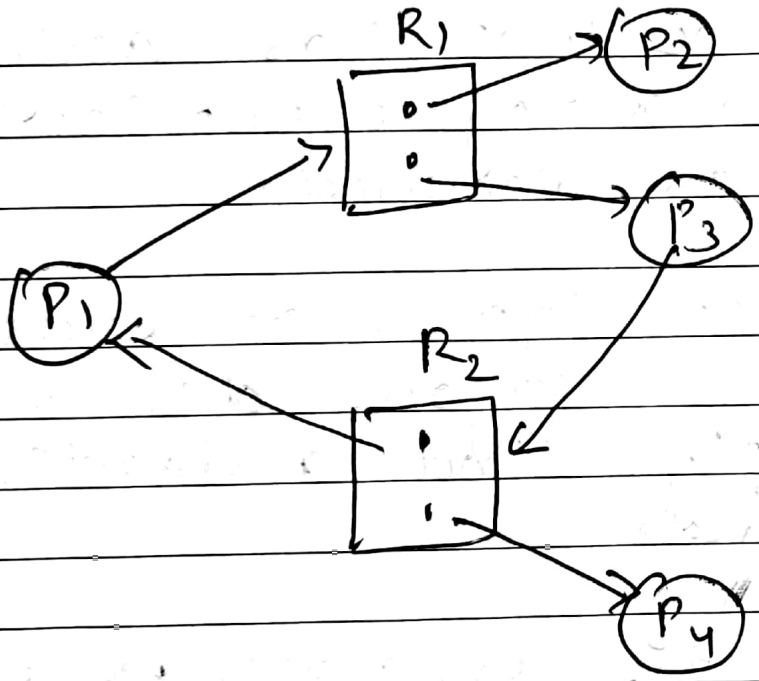
$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

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

Processes P_1, P_2, P_3 are deadlocked. Process P_2 is waiting for the resource R_3, which is held by process P_3. Process P_3

is waiting for either process P_1 or process P_2 to release resource R_2. In addition, Process P_1 is waiting for process P_2 to release resource R_1.

Now consider the RAG. In this ex. we also have a cycle.



$P_1 \rightarrow R_1 \rightarrow R_3 \rightarrow R_2 \rightarrow P_1$

However, there is no deadlock. Observe that process P_4 may release its instance of resource type R_2. That resource can then be allocated to P_3, breaking the cycle.

Deadlock Avoidance

→ Requires that the system has some additional a priori information available.

→ Simplest and most useful model requires that each process declare the maximum number of resources of each type it may need.

→ The deadlock avoidance algorithm dynamically examines the resource allocation state to ensure that there can never be a circular wait condition.

→ Resource allocation state is defined by the number of available and allocated resources and the maximum demands of the processes.

Safe State

→ A state is safe if the system can allocate resources to each process in some order and still avoid a deadlock.

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

→ System is in safe state if there exists a safe sequence of all processes.

Safe state

Example: A system with 12 units of a resource & three processes

P1: max need = 10 allocated = 5
P2: max need = 4 allocated = 2
P3: max need = 9 allocated = 2

Total resources are = 12
Total allocated resource = 5+2+2 = 9
Total available resources = Total resources − Total allocated resources.
12 − 9 = 3

	Max. Need	Allocated	Need or Requirement
P_1	10	5	5
P_2	4	2	4−2=2
P_3	9	2	9−2=7

Total available resources = 3

By 3 available resources, we cannot fulfill the requirement of P_1 but we can fulfill the requirement of P_2. So we give available resources to P_2. P_2 will finish its work. and release the allocated resources + requirement resources.

2 + 2 = 4

So, now total available resources are

4 + 1 = 5

Now we fulfill the requirement of P_1. Then P_1 will release all the

resources.

Now Available resources are 10.

Now we will fulfill the req^n of P_3

So safe sequence = $< P_2, P_1, P_3 >$.

Avoidance Algorithm

For a single instance of a resource type, use a resource-allocation graph.

For a multiple instances of a resource type, use a banker's algorithm.

Data Structures for the Banker's Algo

Let n = number 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. If Max[i, j] = k, the process P_i may request atmost k instances of resource type R_j.

Allocation: n x m matrix. If Allocation[i, j] = k then P_i is currently allocated k instances of R_j.

Need : n x m matrix. If Need [i,j] = k then P_i may need k more instances of R_j to complete its task.

Need [i, j] = Max [i, j] − Allocation [i, j].

Ex. of Banker's Algo.

5 processes P_0 through P_4
3 resource types A (10 instances)
B (5 instances)
C (7 instances)

Snapshot at time T_0

	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		1 2 2
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

Need = Max − Allocation

We cannot fullfil req^n of P_0 (7, 4, 3) and available resource is (3, 3, 2).
So we move forward
Here we fulfill the req^n of P_1 (1, 2, 2) and available resource is (3, 3, 2)
So we fulfil the req^n of P_1
Now Available = (3, 3, 2) − (1, 2, 2) = (2, 1, 0)
and after completing its execution P_1 release all the allocated resources

Now Available = (3,2,2) + (2,1,0)
= (5,3,2)

Now P_2 X

~~P_3~~ we fullfil the requirement of P_3.
Now Available = (5,3,2) − (0,1,1)
= (5,2,1)

Now P_3 release its resources (2,2,2)
Now Available = (5,2,1) + (2,2,2)
= (7,4,3)

Now we fullfil the requirement of P_4.
Now Available = (7,4,3) − (4,3,1)
= (3,1,2).
Now P_4 release its resources (4,3,3)
Now Available = (3,1,2) + (4,3,3)
= (7,4,5)

Now we will be able to satisfy the need of P_0 (7,4,3)
Now Available = (7,4,5) − (7,4,3)
= (0,0,2).
Now P_0 is release its resources (7,5,3)
Now Available = (0,0,2) + (7,5,3)
= (7,5,5)

we already fulfill the req^n of P_1.
Now we fulfill the req^n of P_2 (6,0,0)
Now Available = (7,5,5) − (6,0,0)
= (1,5,5)
Now P_2 release its resources: (1,5,5) + (6,0,0) = (7,5,5)

Now Sequence is = $\langle P_1, P_3, P_4, P_0, P_2 \rangle$

Deadlock Detection and Recovery

→ Allow system to enter deadlock state

→ Detection Algorithm

→ Recovery scheme

Single Instance of each resource type

Requires the creation and maintenance of a wait-for graph.

It is a variant of resource-allocation graph.

The graph is obtained by removing the resource nodes from a resource-allocation graph and collapsing the appropriate edges.

Consequently, all nodes are processes $P_i \rightarrow P_j$ if P_i is waiting for P_j.

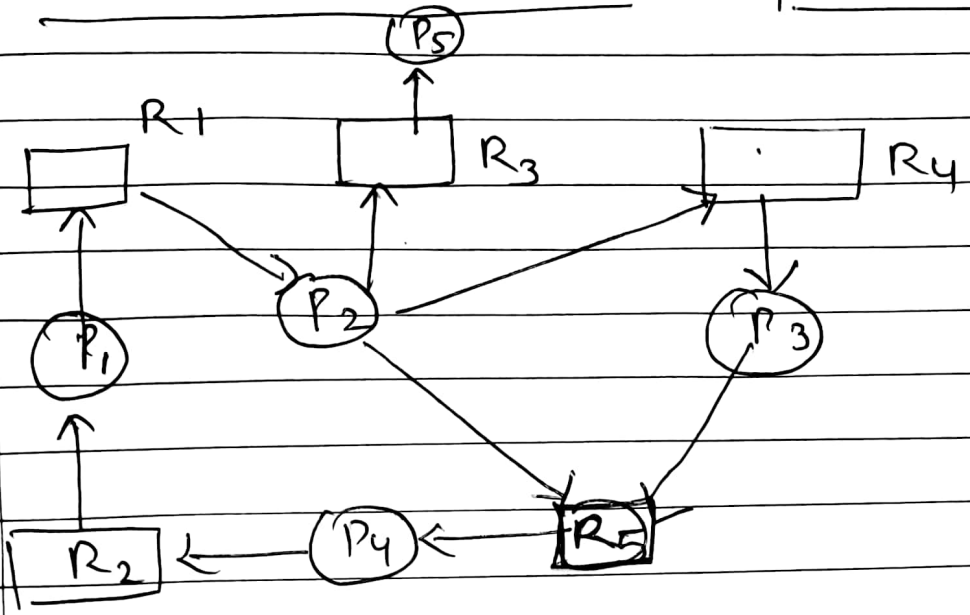
Periodically invoke a algorithm that searches for a cycle in the graph.

→ If there is a cycle, there exists a deadlock.

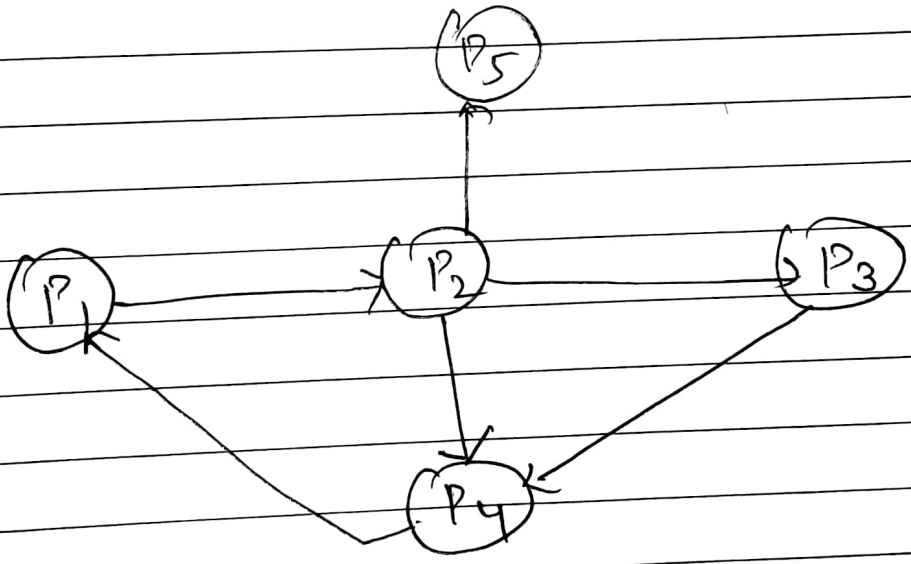
Date / /
Page

RANKA

Resource Allocation Graph & wait for graph



Resource Allocation Graph



wait - for graph