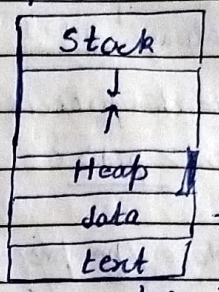
Unit-3 (Process)

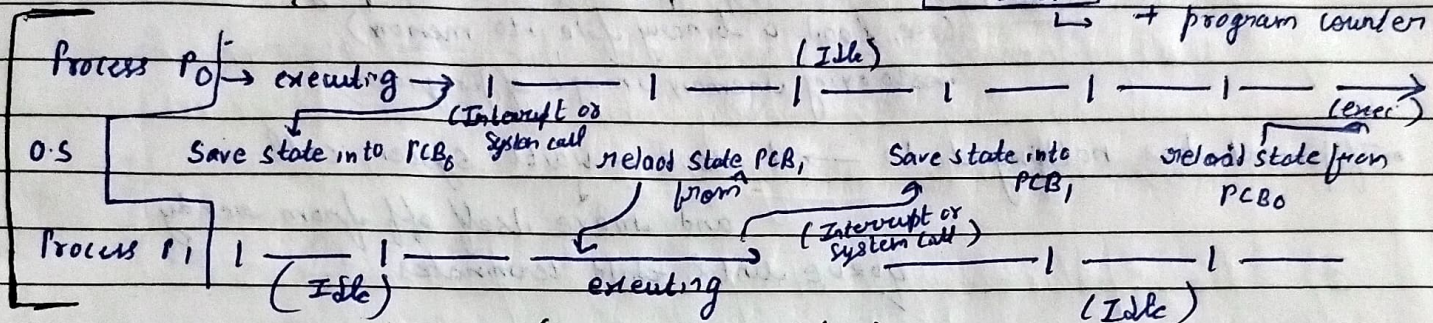
Date _____

- **Process** :- A program in execution
  - (active entity)
  - (passive entity)

* Structure of process in memory

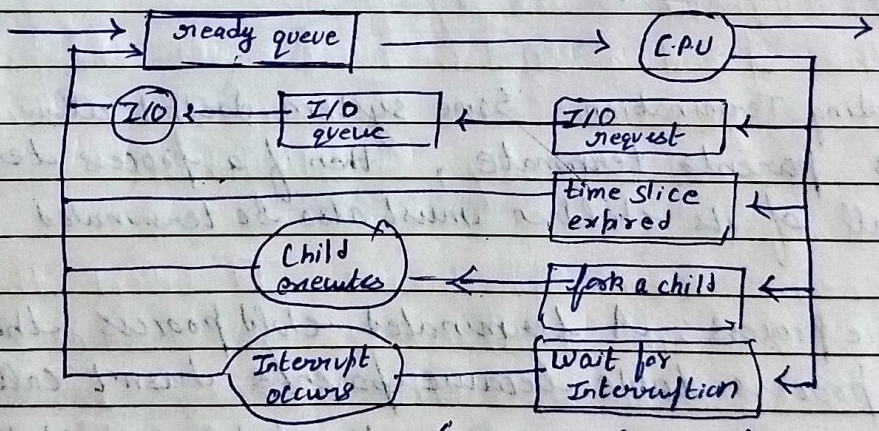


- L-Process State →
- L-PCB → {Doubly}



- **ready queue**, generally stored as linked-list
  - ready queue contains pointer to first and final PCB's in list
    - header
  - each PCB also include pointer field that points to next PCB in ready queue

- Queueing Diagram :



(Queueing Diagram)

- Process are generally of two types
  - I/O bound process → {doing more I/O}
  - C.P.U process → {doing more computation}

So, L.T.S always select mix of I/O and C.P.U process (Long term sch.)

- Also, a (Middle Term Scheduler) → It do **swapping** {:- sometimes its advantageous to remove process from ready queue}

It reduces the degree of multi-programming.

Spiral

Date

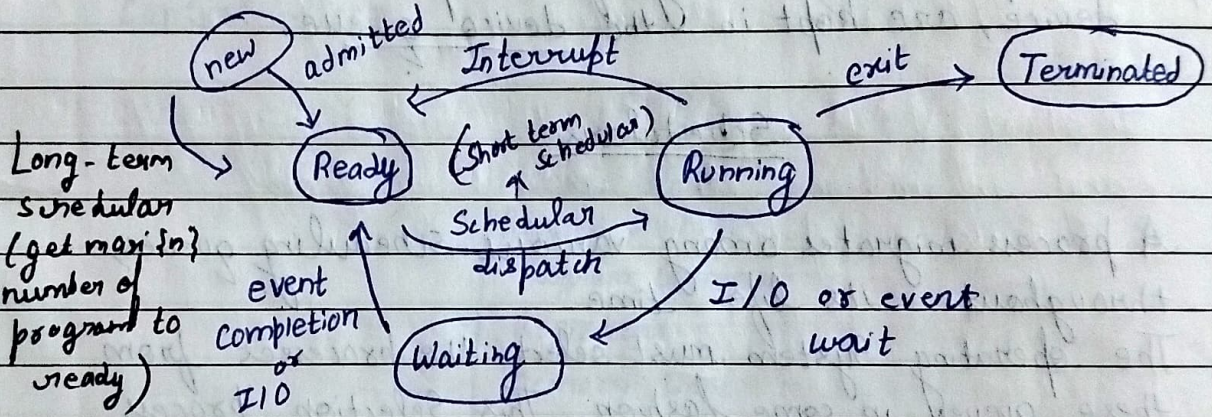
Process Management

- A program in execution is known as the process.

Process State :

As a process executes, it changes state. A process may be in one of the following stages :

1) New
2) Running
3) Waiting
4) Ready
5) Terminated



(Process - State Diagram)

(a) New → The process is being created.
(b) Running → Instructions are being executed.
(c) Waiting → The process is waiting for some event to occur, (such as I/O completion)
(d) Ready → The process is waiting to be assign to the processor
(e) Terminated → The process has finished the execution.

Spiral

Date

Scheduling Queues :

1.) As processes enter the system, they are put into a Job queue, which consists of all processes in system.

2.) The processes that are residing in the main memory and are ready and waiting to execute, are kept in the ready queue.

3.) The processes waiting for a particular I/O device, are kept in the device queue.

Schedulars

A process migrates among various scheduling queues, throughout its life time.
The operating System must select the processes from these queues in some fashion. This Selection process is carried out by a schedular.

1.) Long Term Schedular: It select the processes from a mass-storage device (disk) and loads them into the memory for execution.
It controls the degree of multi-programming

2.) Short term or C.P.U Schedular: It selects from among the processes that are ready to execute and allocates the C.P.U to one of them.

Spiral

Date _____

- **Context Switching**: Switching the C.P.U to another process requires, saving the state of the current process and re-storing the state of a different process. This task is known as a context switch.

Task control Block

- **Process - Control Block** : Each process is represented by a P.C.B.

Imp

: It is also called the task control block.

P.C.B

Process State
Process number
Program Counter
Registers
Memory Limits
List of open-files

: It contains many pices of information, associated with the specific process

1.) Process state: The state may be new, ready, running, waiting and so on.

2.) Program Counter: It indicates the ~~adjust~~ address of the next instruction, to be executed for this process.

3.) C.P.U registers: The registers vary in number and type depending on the computer architecture, they include accumulators, index registers, stack pointers etc.

Spiral

Date

4) C.P.U Scheduling Information: This info includes scheduling parameters for a process

5) Memory management Info: This info includes the value of base and limit registers and the page tables, segment tables etc.

6) Accounting Info: This info includes the amount of C.P.U and time lapsed

7) I/O status info: This info includes the list of I/O devices allocated to the process

In short, P.C.B serves as the repository for any information that may vary from process to process

---

You visit bank at 11:00 am → Arrival Time (Time which process enter ready queue)

Wait in queve to get cash withdraw = 45 min {11:45} − Waiting Time (TAT − BT)

finally get a chance to withdraw cash = 15 min to withdraw cash {11:45 − 12:00} → Burst time (Time taken by C.P.U to get execute that process) duration

ok, after that come out of Bank = 12 Noon → Completion time (Time at which process complete execution)

The complete time of process = Turn around time = 60 min

(Take R.H.S in gantt chart)

. Response Time = The time at which a process get C.P.U first Time − Arrival Time

(Take L.H.S in gantt chart)

I enter in bank 11 am, but first time I get served by clerk at 12:45 am ∴ 45 min

Spiral

Types of C.P.U Scheduling algorithm

Date 17-08-24

Premptive

1) Shortest remaining Time first (SRTF)
2) Round Robin (RR)
3) Priority Scheduling
4) Longest remaining Time first (LRTF)

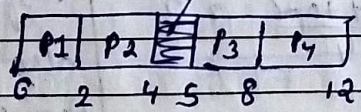
Non-premptive

1) First come first serve (FCFS)
2) Shortest Job first (SJF)

1) F.C.F.S → Criteria = Arrival Time
→ Mode = Non-premptive

Process No	Arrival Time	Burst Time	Completion Time (T)	Turn around Time = CT - AT
P1	0	2	2	2
P2	1	2	4	3
P3	5	3	8	3
P4	6	4	12	6

Gantt chart



Waiting Time = TAT - BT	Response Time = C.P.U first Time - A.T
0	0
1	1
0	0
2	2

- Turn around Time : It is the time the process remain in the system

avg. TAT = 0+2+3+6/4 = 3.5
avg W.T = 0+1+0+2/4 = 0.75
avg R.T = 0+1+0+2/4 = 0.75

In non-premptive approach, response Time is always equal to waiting Time

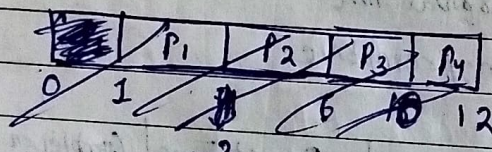
Date

Give priority to process which takes less time? → i.e. Burst Time

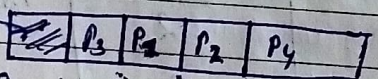
2.) Shortest Job first :
- Criteria : Burst Time
- Mode : Non-premptive

Process No	Arrival Time	Burst Time	Completion Time	Turn around Time	W.T	R.T
P1	1	3	6	5	2	2
P2	2	4	10	8	4	4
P3	1	2	3	2	0	0
P4	4	4	14	10	6	6

Gantt chart



C.P.U Idol



(P1, P3) (P1, P2) (P2, P4) — { Same burst time, so, check arrival time } → P2 has less A.T

avg TAT = $\frac{5+6+2+10}{4}$ = 6.25

avg W.T = $\frac{2+4+0+6}{4}$ = 3

avg RT = 3

- If A.T also same, then come to process I.D, less I.D, first execute!

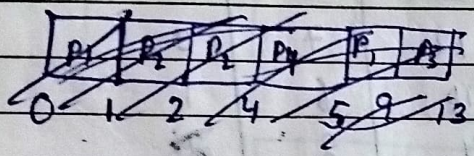
Spiral

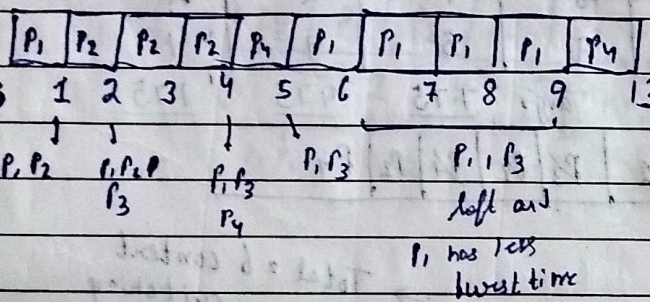
* Always first check Arrival time,
↳ then Jump to Criteria { If Multiple process collect together } Date _____

3) SRJF → Criteria : Burst Time → { Basically → Shortest Job first + Preemptive }
Mode : Preemptive

Take → original Burst time

Process no	Arrival Time	Burst Time	C.T	T.AT	W.T	RT
P_1	0	~~5~~ ~~4~~ 3	9	9	4	0 {0-0}
P_2	1	~~3~~ 2 ~~1~~ 0	4	3	0	0 {1-1}
P_3	2	4	13	11	7	7 {9-2}
P_4	4	1	5	1	0	0 {4-4}



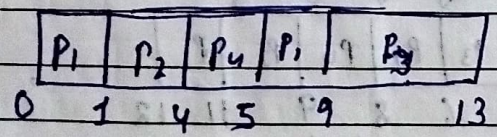


∴ avg Tat = $6\frac{24}{4} = 6$

avg W.T = $\frac{11}{4} = 2.75$

avg RT = $\frac{7}{4} = 1.75$

Overall :

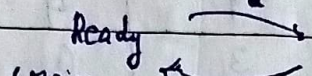


∴ The time at which all process executed = 13

Date

- Round Robin → Criteria : Time quantum
  → Mode : Premptive

* Time quantum :- Particular time till which process executes

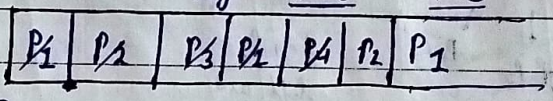


If T.Q = 2, so it execute all process at till 2.

* Sequence of processes in ready queue is most important.

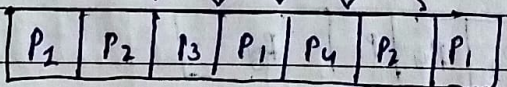
Process No	Arrival Time	Burst Time	C.T	T.A.T	W.T	R.T
P_1	0	~~8~~ ~~7~~ 1	12	12	7	0
P_2	1	~~4~~ ~~3~~ 0	11	10	6	1
P_3	2	~~2~~ 0	6	4	2	2
P_4	4	~~1~~	9	5	4	4
			avg:	7.75	4.75	1.75

Given, TQ = 2 Ready queue



Total = 6 context switching

Running queue (Gantt chart)



∵ P_4 takes only 1 s

time →

Here we do Context switching (Save running and load new process) } process contest tie, PCB (Process control Block)

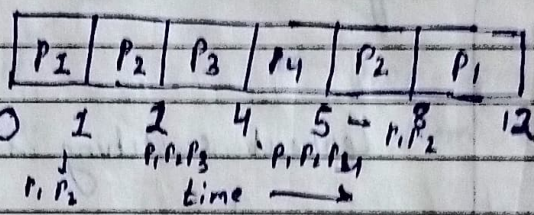
i.e, In future we resume that process, not re-start.

- after putting every process in ready queue, check → New process ready → If Yes, then Spiral put in ready queue
  → Running process remain → If Yes then also put in ready queue.

Date

- Pre-emptive Priority Scheduling → following → Priority
  → Horde → ~~Never~~ pre-emptive
  * Higher the number → Higher the priority

Priority	Process	A.T	B.T	C.T	TAT	W.T	R.T
10	P_1	0	84	12	12	7	0
20	P_2	1	43	8	7	3	0
30	P_3	2	20	4	2	0	0
40	P_4	4	10	5	1	0	0
				avg	5.5	2.5	



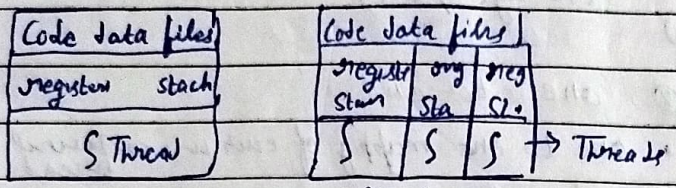
Spiral

Continuation of unit-3

Date

(from 4.1 . . . .)

- **Threads:** → basic unit of CPU utilization



eg Word processor → multiple thread
1.) Displaying graphs
2) Grammar / Spelling check
3) Keystroke from user ... many more tasks

- **Benefits of multithreaded programming:**

(a) Responsiveness: ensure application remain interactive
(b) Resource sharing: Simplies and speeds up Inter thread communication
(c) Economy: process are costly, so thread reduce cost overhead
(d) Scalability: → utilize multicore architecture more efficiently by parallel execution

- **Multicore programming**: multiple cores on a single chip, each core appearing as a separate processor, facilitates multi-threading enhances concurrency and ||ism.
{multiple task can progress Ind., but may or may not run simultaneously}
{Task execute simultaneously}

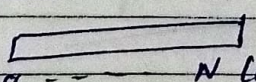
So, ~~con~~ concurrency without parallelism (possible ✓)
Inverse (not ✗)

- **Programming challenges in multicore:**

(a) Identifying task: find ind. and concurrent task that can run in parallel.
(b) Balancing workload: ensure task have similar workloads to avoid Idling.
(c) Data splitting: Divide data for Ind. processing {avoid conflicts}.
(d) Data dependency: Synchronize task when they share or depend on data.
(e) Testing and debugging: || programs are harder to debug due to unpredictable execution path

- **Data vs Task ||ism** subset of same

→ • Distribution of data across multiple cores, performing same operation on each core.

eg 

splitting half and adding both

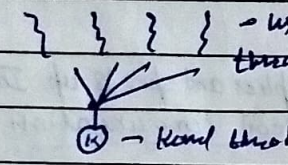
• Dist of task (threads) across multiple cores, performing a unique operation
{Diff Thread may operate on same or diff data}

e.g. Son threads but unique Spiral stats operation.

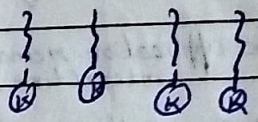
Date

- Support of threads, may provide as User or Kernel threads
  (Manage in user space without O.S support) (manage by O.S)

① Many-to-one
- Many user threads to 1 Kernel thread
- Blocking system call by 1 user thread, block entire process {No True Illism}



② one-to-one
- one to one mapping of each user and kernel thread
- No blocking issue
- High no of thread creation overhead and performance issue



③ Many-to-one
- Many user thread to smaller / equal no of Kernel threads
- Unlimited user thread + ll execution.

also two-level Model : to bind user thread to kernel thread

- POSIX Thread (P thread) an API in Unix, Mac, Linux to create, manage and synchronize threads.

~~Problem and Solution for multicore process~~
Synchronization

- Co-operating process : Process that can affect or be effected by other processes executing in the system.

- Situation in which several processes access and manipulate same data concurrently, and ~~order~~ outcome of execution depends on the particular order in which access takes place → called race condition.

eg Variable X = 10
first X = X + 10 and X = X * 2 → = 40
or X = X * 2, X = X + 10 → = 30 (order of execution)

a = 10
P1 → P1 = Counter++ (all process 3 things: • Read value of counter • modify value of counter • Write modified value back to counter)
P2 → P2 = Counter--
P1 do → 11
P2 do → 9
final = 9
expected = 10 {∵ +, - cancel out}

Spinal
- Non-preemptive Kernel

Date

- **Critical Section problem** and **Peterson Solution**

↳ each process has a segment of code called **critical section** in which the process ~~exchange~~ change common variables, update table, read file and so on....

- **Peterson's Soln**: Software based approach to solve critical-section problem for two processes. It ensures all 3 requirment to solve critical section problem.

- Two shared variables are used:

(a) flag[i] = Indicates if a process wants to enter critical section
(true means the process want to enter)
false means " " " " exit)

(b) turn : Decides which process gets the priority to enter the critical section

→ Structure of process P_i in Peterson soln.

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

    critical Section

    flag[i] = false;

    remainder Section

} while (true);
```

(entry Section)

{ Give priority to P_j } (wait until P_j finishes)

(exit section)

Why it works?

(a) Mutual exclusion: Only 1 process can enter the critical section at a time.

(b) Progress: If no process is in critical section, waiting processes can proceed { also ensure it's not also in remainder section } executing in

(c) Bounded : No process waits forever, waiting each gets a chance after the other finishes

Spiral

Critical Section Explain in unit-5

Date

• C.P.U Scheduling

. Any process undergoes various C.P.U – I/O burst cycles

• Scheduling decision is place under these 4 circumstances :

→ When a process switches from running ⟶ Waiting state. (1)
→ " " " running ⟶ ready state (2)
→ " " " waiting → ready state (3)
→ " terminates. (4)

1, 4 → Non-premptive or Co-operating Scheduling scheme (No choice, to select which process)
2, 3 → premptive Scheduling scheme (Choice given)

- Dispatcher → It's a module that gives control of C.P.U to process selected by S.T.S (short term)
• (Dispatcher latency) → Time taken by it to stop one process and start another running.

• Scheduling criteria

(a) C.P.U Utilization (Maximize it)
(b) Throughput → Number of process completes per unit time (Maximize)
(c) TAT →, (d) W.T (Time process spend waiting in ready queue) (e) Response Time ⟶ (Minimize)

→ for Interattive System, we want to min Variance than to min avg response time. (i.e, Inconsistency)

(a) FCFS →. More waiting time, as its non-premptive.

Convoy effect: Assume 1 → CPU Bound process, Many → I/O bound process
(C.B) (I/O)

When C.B takes C.P.U, all I/O finish execution, now move in ready queue and wait for C.P.U, making I/O device sits Idle.
Now, then C.B finish C.P.U burst and moves to I/O, then all I/O quickly finish C.P.U burst, again wait in I/O device queue., making C.P.U Idle.
. . . . ⟶ and cycle repeats creating convoy effect, where all other process wait for 1 big process to get off C.P.U.

Spiral

Date

(b) S.J.F : • It's optimal as it reduces avg waiting time, but harder to implement, as we not know in advance, the length of next CPU burst

• Methods like exponential averaging used to predict next CPU burst, not so efficient, also introduce overhead.

(c) Priority Scheduling : Here problem of Starvation (Indefinite blocking) occurs, in which low priority process may never get CPU if the Higher priority process keeps arriving.

Soln → aging (gradually increase priority of a process waiting for a long time)

• Round Robin → Here time quantum should choose wisely,

• Larger Time quantum → RR behaves like FCFS (few context switch)

• Smaller " " → Causes many context switch, reducing CPU efficiency

rule of thumb → 80% of CPU burst should be shorter than time quantum to balance efficiency and responsiveness

Spiral

Date

• Deadlock

- In a multi-programming environment, several processes may compete for a finite number of resources.
- If the resource requesting by the process, is not available at that time, then the process enters a waiting stage.
- Sometimes, a waiting process is never able to get the requested resource, because that resource is held by other waiting processes. This situation is called a deadlock.

→ Deadlock is a state of system, in which a set of processes are waiting for each other for in-definite Time.

- A process may utilize a resource in the following sequence:

(a) Request : The process request the resource. If the request can't be granted immediately {for e.g. If the resource is used by another process}, then the requesting process must wait, until it can acquire that resource.

(b) Use : The process can operate on the resource.

(c) Release : The process releases the resource.

Date

• Conditions for a deadlock to occur:

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

(a) Mutual exclusion: Only one process at a time can use the a resource. If another process request that resource, the requesting process must be delayed until the resource has been released.

(b) Hold and ~~Weight~~ Wait: A process must be holding atleast one resource, and requesting for other resources that are currently being held by other processes

(c) No preemption: Resources can't be ~~preemption~~ preempted (i.e, a resource can be released, voluntary by the process holding it, after that process has completed its task)

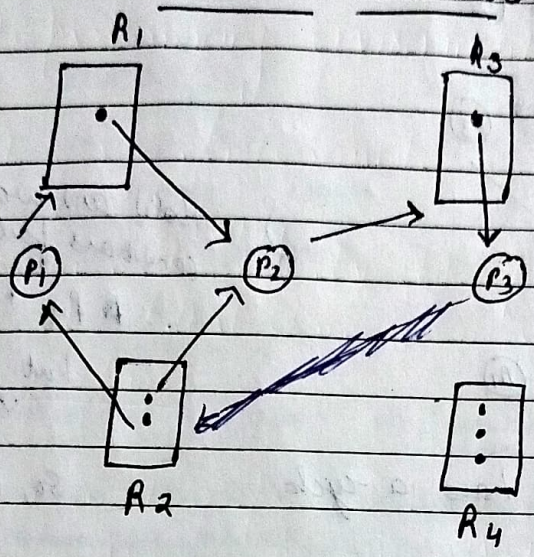
(d) Circular Wait: A set of waiting processes, must exist such that p_0 is waiting for a resource held by 'p_1', 'p_1' is waiting for a resource held by 'p_2', ..., and so on. {$p_0, p_1 \ldots p_n$}

↳ It is important to note, that all 4 conditions must hold true, for a deadlock to occur. (above)

Spiral

Date

• Resource Allocation Graph



P = { P1, P2, P3, ... }

R = { R1, R2, R3, R4 }

Set of edges

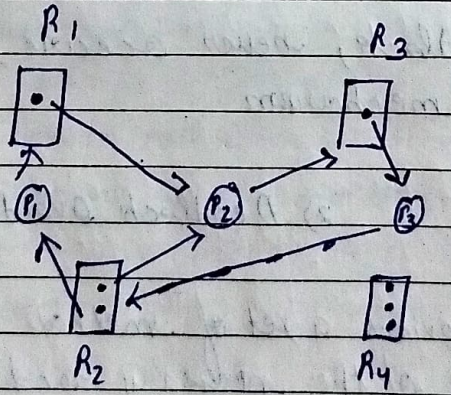
E = { request edge,
P1 → R1,
R2 → P2
assignment edge }

fig 1: There is no cycle,
∴ processes P1, P2 and P3 will never be in a deadlock state.

• → Instance
□ → Resource
○ → Process

from the resource allocation graph, it can be shown that, if the graph contain no cycle, then no process in the system is deadlock.

However if a graph contains a cycle, then a deadlock may exist.



In fig 2: There are two cycles formed

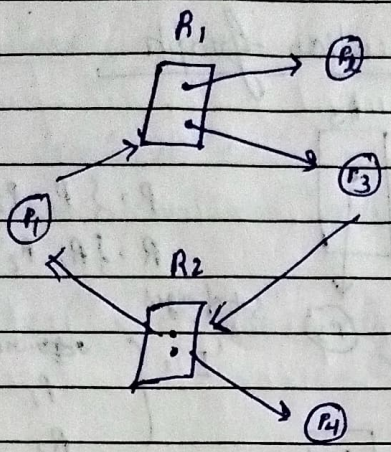
→ { P1 → R1 → P2 → R3 → P3 → R2 → P1 }

→ { P2 → R3 → P3 → R2 → P2 }

P1, P2, P3 are in a deadlock state.

Spiral

Date



{ ∵ Hold and wait conditions fails
∴ P4 has assignment edge but doesn't request edge
So, cycle occurs but no deadlock

Fig 3: In fig 3: we have a cycle:

Cycle:

$\{P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1\}$

However, there is no deadlock. This is because P_4 may release an instance of resource ~~type~~ R_2, which is then allocated to P_3, and Hence breaking the cycle.

Methods for handling Deadlock

To ensure that deadlock, never occurs, the system uses two mechanism:

1.) Deadlock prevention 2) Deadlock avoidance

- Deadlock prevention: It provides a set of methods that ensures, atleast one of the necessary conditions can't hold

- avoidance → It requires that O.S, be given additional inf. in advance about which resources, a process will request and use during its lifestyle.

Spiral

Date

- In linux, → init process (root parent process)
  always pid = 1 ∀ process

- fork() → create child process (pid = 0)
  parent's pid > 0

exec() → replace the process's memory space with new program
(i.e, load a binary file into memory)
(destroying memory image of program)

- If parent has nothing to do → it issue → wait() system call
  and move itself off from ready queue until child terminates

- When a parent terminate execution of one of its children?

(i) Child has exceeded usage of some of its resources that it has been allocated.
(ii) task assign to child is no longer required.
(iii) parent is exiting, and O.S does not allow a child to continue, if its parent's terminate

- Cascading Termination: Some system doesn't allow child to exist if its parents terminate, then if a process terminates then all of its children must also be terminated

- Zombie process: A terminated child process that remains in process table because parent hasn't called wait()
  as soon as wait() invoked 0, exit status is ~~stored~~ finish

- Orphan process: A process whose parent has terminated before it finishes execution.
  - In Linux/unix init process adopt them which ensures resource clean up ~~de~~ calling wait().
    by

Spiral