Operating Systems

Unit - 2

Process Management

*Process - a program in execution, process execution must progress in a sequential fashion.

It has different parts:

(i) program code = text section

(ii) program counter

(iii) stack w/ temporary data like fn. parameters, return addresses, local variables

(iv) data section = global variables

(v) heap = dynamically allotted memory during runtime

* Program vs. Process

→ program is a passive entity, process is active

→ program becomes a process when an executable file is loaded into the memory.

* Process State

As a process executes, it changes state:

(i) new : the process is being created

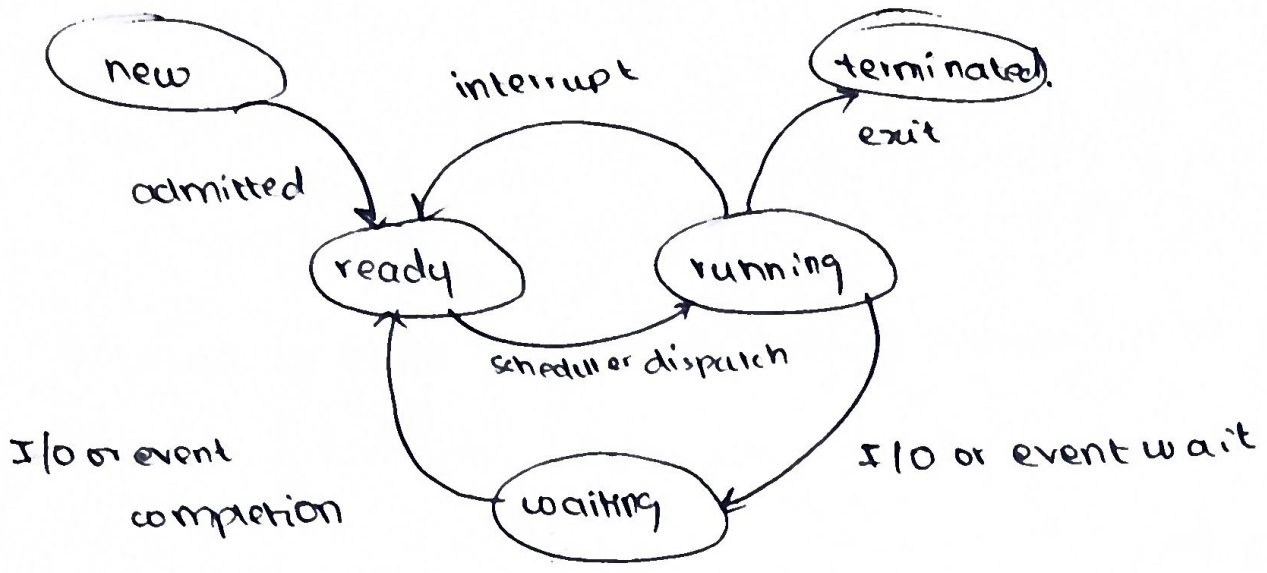
(ii) running : instructions are being executed

(iii) waiting : process is waiting for some instruction to occur

(iv) ready : the process is waiting to be assigned to a processor

(v) terminated : the process has finished execution

Process State Diagram



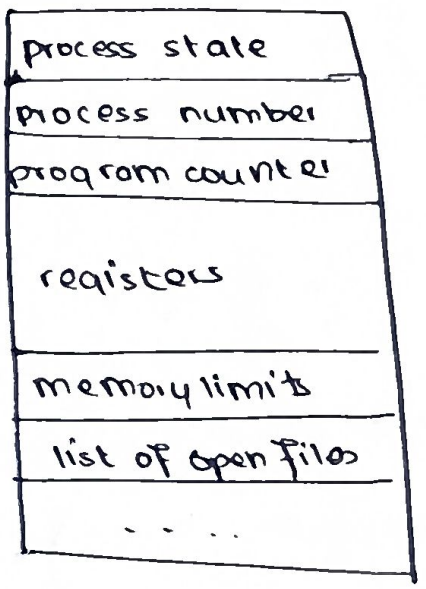
* Process Control Block

→ has information associated with each process, also called the task control block

(i) process state - running, waiting etc.

(ii) program counter - location of the next executable instruction

(iii) CPU registers



(iv) CPU scheduling information - priorities, scheduling queue pointers

(v) memory management information

(vi) accounting information - CPU used, clock time elapsed since start,

(vii) I/O status information - I/O devices alloted to process, list of open files.

* Process Scheduling

→ The process scheduler selects the next among the available processes for execution.

→ It maintains scheduling queues of processes.

It has:

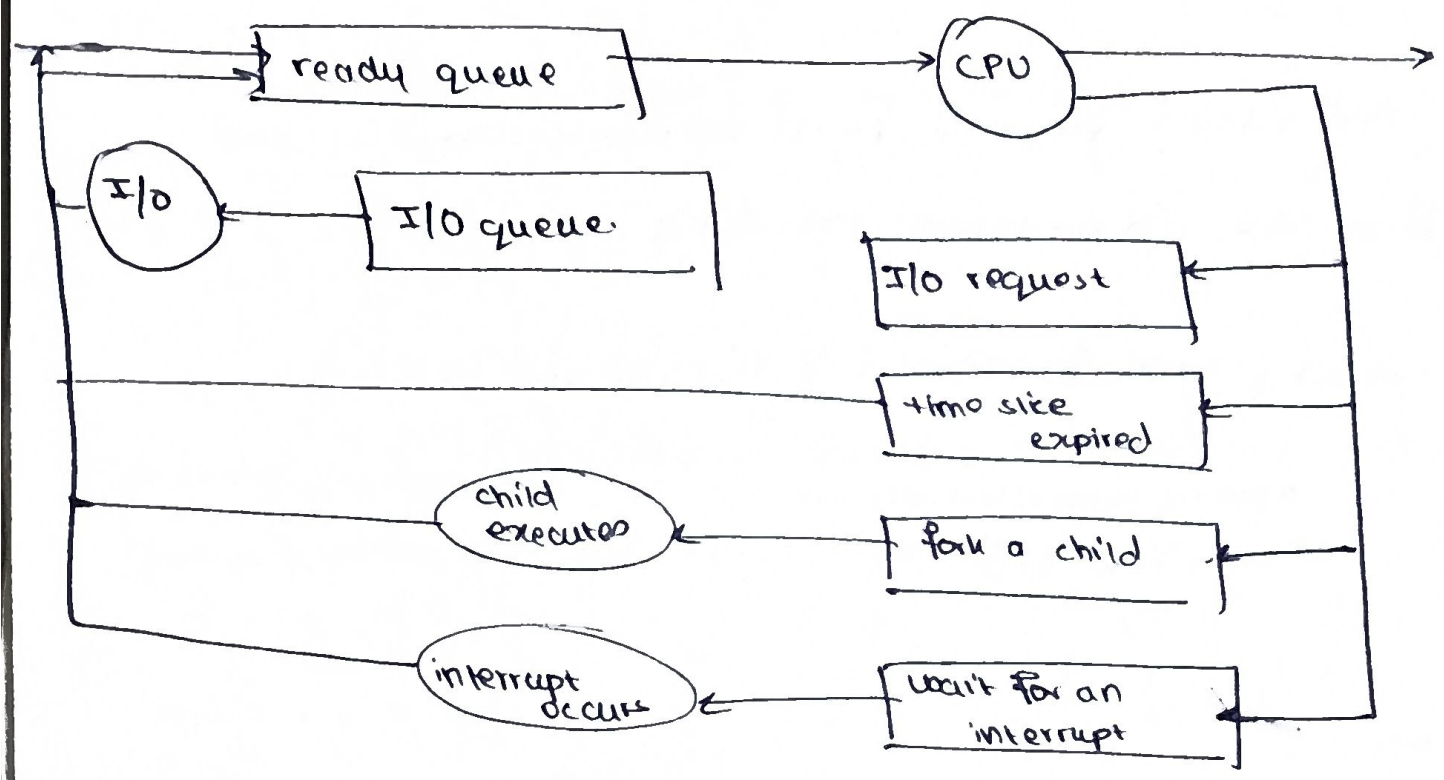
(i) Job queue - set of all processes in the system

(ii) Ready Queue - set of all processes residing in main memory, ready and waiting to execute

(iii) Device queues - set of processes waiting for an I/O device

Processes migrate among the various queues.

Queueing Diagram



* Types of Schedulers

1. Short Term Scheduler / CPU Scheduler

→ selects which process to be allocated next and allocates CPU

→ invoked frequently, ∴ must be fast

2. Long Term Scheduler / Job Scheduler

→ selects which process should be brought into the ready queue

→ invoked infrequently, ∴ may be slow

→ controls the degree of programming

3. Medium Term Scheduler

→ can be added if the degree of multiple programming needs to decrease

* Types of Processes

- I/O bound process: spends more time doing I/O than computations many short CPU bursts

- CPU bound process - spends more time doing computations, few long CPU bursts.

* Swapping : to remove process from memory, store on disk, bring back in from disk to continue executing

* Multitasking : running both foreground & background processes

↙ usually controlled via user interface

↘ in memory, running but not on the display

* Context Switching :

→ When the CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a context switch.

→ Context of a process is represented in the PCB

* Operations on Processes

(i) process creation

(ii) process termination

(iii) process communication & inter process communication

(iv) process synchronization

(v) process scheduling

A. Process Creation

→ Parent process creates child processes, which in turn, create other processes, forming a tree of processes.

→ Each process is identified and managed via a p-id. (process identifier)

Resource Sharing Options

→ parent & child share all resources

→ children share a subset of parent's resources

→ parent and child share no resources

Execution Options

→ parent and children execute concurrently

→ parent waits until child terminates.

→ In unix, fork() → creates new process

exec() → used after fork() to replace memory space of new program.

B. Process Termination

→ Process executes the last statement, and then asks the OS to delete it using exit()

→ Returns status data from child to parent (via wait())

→ The process' resources are deallocated

abnormal termination: using abort()
can be when:

(i) child has exceeded allocated resources

(ii) task assigned to child no longer required

(iii) parent is exiting, and OS does not allow child to continue if parent terminates.

* Cascading Termination - if parent terminates, all children, grandchildren also terminate

* Orphan Processes - parent process waits for termination of child using the wait() system call. If parent is terminated without involving wait, the process becomes an orphan.

C. Interprocess Communication

→ Processes in a system may be independent or cooperating

Independent: process not affected by the execution of another process

Cooperating: can be affected by the execution of another process

* Multiprocess Architecture - Chrome Browser

Chrome browser runs 3 types of processes

(i) browser - w/ UI, and network

(ii) renderer - renders webpages w/ HTML, JS

(iii) plug-in - process for each type of plug-in

Advantages of process cooperation

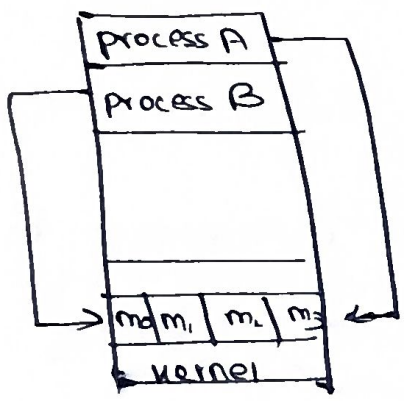
(i) information sharing
(ii) computation speed-up
(iii) modularity
(iv) convenience

→ Cooperating processes need interprocess communication (IPC).

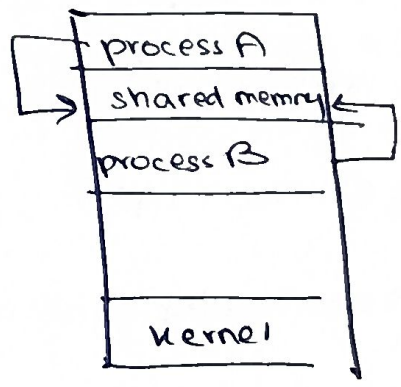
There are 2 models of IPC.

(i) shared memory
(ii) message passing

message passing



shared memory



Shared memory

→ an area of memory shared among the processes that wish to communicate

→ one major issue is that mechanisms that allow user processes to synchronize their actions, when they access shared memory must be provided

Message Passing — processes communicate with each other w/o resorting to shared variables.

→ There are 2 operations :

(i) send (message)
(ii) receive (message)

→ A communication link must be established between the Process. It can be a

(i) physical link

→ shared memory

→ hardware bus

→ network

(ii) Logical link

→ direct or indirect

→ synchronous & asynchronous

→ automatic or explicit buffering

Direct Communication

→ processes must name each other explicitly

send (P, message) - send a message to process P

receive (Q, message) - receive a message from process Q

→ links are automatically stablished, between exactly one pair of communicating processes

→ links may be unidirectional, but is usual bidirectional

Indirect Communication

→ messages are directed and received from mailboxes

→ Each mailbox has a unique id

→ Processes can communicate only if they share a mailbox

→ Each pair of processes may share several communication links

→ links maybe unidirectional or bidierectional

Steps

- create a new mailbox(port)
- send and receive messages through mailbox
- destroy mailbox

fns:

send (A, msg) → send message to mailbox A

receive (A, msg) → receive message from mailbox A

Synchronization

A. **Blocking** = synchronous

blocking send - sender is blocked until the message is received

blocking receive - receiver is blocked until a message is available.

B. **Non-Blocking** - asynchronous

non-blocking send - sender sends msg. & continues

non-blocking receive - receiver receives a valid message or a null msg.

★ → If both send & receive are blocking ⇒ rendezvous

Buffering

→ a queue of messages attached to the link

→ implemented as:

(i) zero capacity - no messages are queued on link, sender waits for receiver (rendezvous)

(ii) bounded capacity - finite length of n msgs, sender must wait if link is full

(iii) unbounded capacity - infinite length, sender never waits

* Communication in Client-Server Systems

A. Sockets

→ an endpoint for communication

→ is the concatenation of IP address & port

eg. 161.125.19.8 : 1625
host : port

→ communication happens between pairs of sockets

→ all ports below 1024 are well known, used for standard services.

B. Remote Procedure Calls

→ abstracts procedure calls between processes

stubs = client-side proxy for the actual process on the server

→ client-side stub locates server & marshalls params

→ server-side stub receives message, unpacks marshalled parameters & performs the procedure

→ Data represented in big-endian / little-endian format

C. Pipes

→ acts as a conduit allowing 2 processes to communicate

* Ordinary pipes : cannot be accessed from outside the process that created it. Usually, parent creates pipe & to communicate w/ child process,

Named Pipes → comm. in standard producer-consumer style

→ producer writes to one end (write-end of pipe)

→ consumer reads from the other end (read-end of pipe)

→ Ordinary pipes are therefore unidirectional

* Named Pipes → more powerful than ordinary pipes

→ bidirectional comm.

→ no-parent child relationship necessary between communicating processes

→ several processes can used the named pipe for comm.

*CPU Scheduling

*CPU - I/O Burst Cycle

→ process execution consists of a cycle of CPU execution & I/O wait

→ CPU burst followed by I/O burst

load store
add store
read from file
} CPU burst

wait for I/O
} I/O burst

store increment index
write to file
} CPU burst

wait for I/O
} I/O burst

load store
add store
read from file
} CPU burst

* Preemptive and Non-Preemptive Scheduling

Preemptive

→ used when a process switches from running state to ready state or from waiting state to ready state

→ resources are allocated for a limited amount of time to the process and then taken away

→ process is placed back in ready queue if the process still has CPU burst time remaining

eg. RR, SRTF, Priority (preemptive)

Non - Preemptive

→ used when a process terminates, or when a process switches from running to waiting state.

→ Once resources are allocated, the process holds the CPU till it gets terminated, or until it reaches a waiting state.

→ Non- preemptive scheduling does not interrupt a process running CPU in the middle of the execution. Instead, it waits until the process completes its CPU burst time & then allocates CPU to another process.

→ eg. FCFS, SJF, Priority (non-preemptive version)

* Dispatcher

→ this module gives control of the CPU to the process selected by the short-term scheduler. This involves:

(i) switching context

(ii) switching to user mode

(iii) jumping to the proper location in the user program to restart that program.

* Dispatch Latency - time taken by the dispatcher to stop one process and start running another.

* Scheduling Criteria

(i) CPU utilization - keep CPU as busy as possible MAX

(ii) Throughput - number of processes that complete their execution per time unit. MAX

(iii) Turnaround Time - amount of time to execute a particular process MIN

(iv) Waiting time - amount of time a process has been waiting in the ready queue. MIN

(v) Response - amount of time it takes from when a request was submitted, until the first response is produced MIN

* First Come First Serve (FCFS)

→ Processes are executed in the order in which they arrive.

→ non-preemptive scheduling algorithm

eg ①

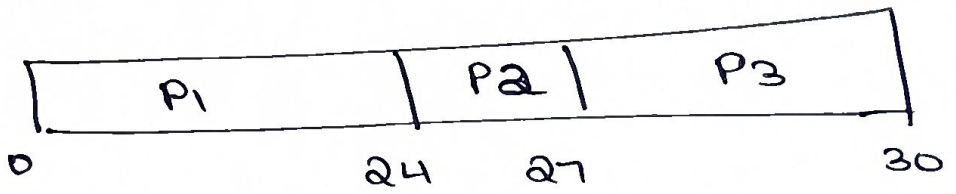
Process	Burst Time
P_1	24
P_2	3
P_3	3

arrive in the order P_1, P_2, P_3

ST+BT ST-AT ET-AT / WT+BT ST-AT

PID	Arrival Time	Burst Time	ST	ET ↑	WT ↑	TAT ↑	RT →
P_1	0	24	0	24	0	24	0
P_2	0	3	24	27	24	27	24
P_3	0	3	27	30	27	30	27
					17	27	17

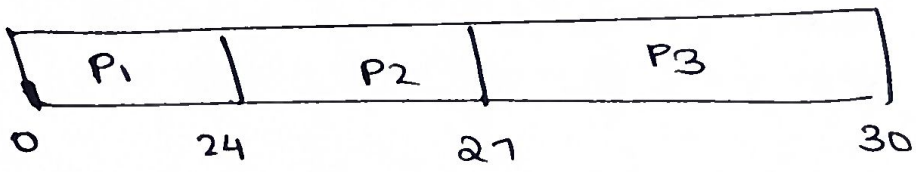
Gantt Chart



②

PID	AT	BT
P_1	0	24
P_2	10	3
P_3	15	3

Gantt Chart

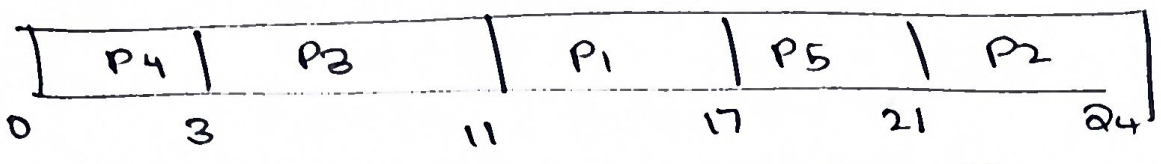


PID	AT	BT	ST	ET	WT	TAT	RT
P_1	0	24	0	24	0	24	0
P_2	10	3	24	27	14	17	14
P_3	15	3	27	30	12	15	12
					8.66	18.66	8.66

(3)

PID	AT	BT
P1	2	6
P2	5	3
P3	1	8
P4	0	3
P5	4	4

Gantt chart:

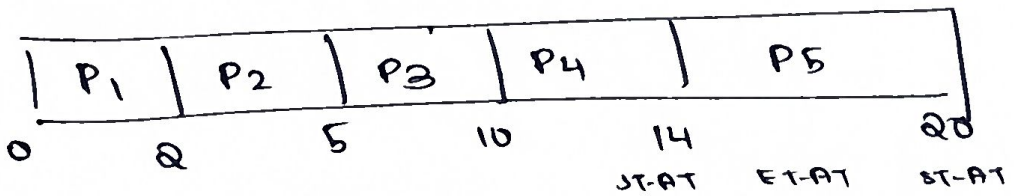


PID	AT	BT	ST	ET	WT (ST-AT)	TAT (ET-AT)	RT (ST-AT)
P4	0	3	0	3	0	3	0
P3	1	8	3	11	2	10	2
P1	2	6	11	17	9	15	9
P5	4	4	17	21	13	17	13
P2	5	3	21	24	16	19	16
					8	12.8	8

(4)

PID	AT	BT
P1	0	2
P2	1	3
P3	2	5
P4	3	4
P5	4	6

Gantt Chart



PID	AT	BT	ST	ET	WT (ST-AT)	TAT (ET-AT)	RT (ST-AT)
P1	0	2	0	2	0	2	0
P2	1	3	2	5	1	4	1
P3	2	5	5	10	3	8	3
P4	3	4	10	14	7	11	7
P5	4	6	14	20	10	16	10
					4.2	8.2	4.2

* Convoy Effect - short process then long processes

* Shortest Job First

→ associate each process to the length of the next CPU burst

→ These lengths are used to schedule the process with the shortest time

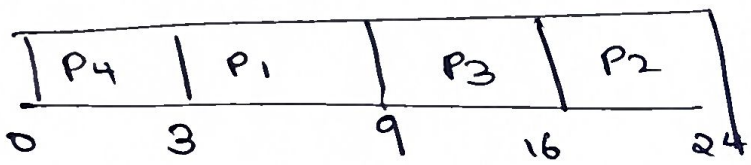
Disadvantage – will not know length of next CPU request

A. Non-Preemptive SJF (NPSJF)

(1)

PID	AT	BT
P1	0	6
P2	0	8
P3	0	7
P4	0	3

Gantt Chart

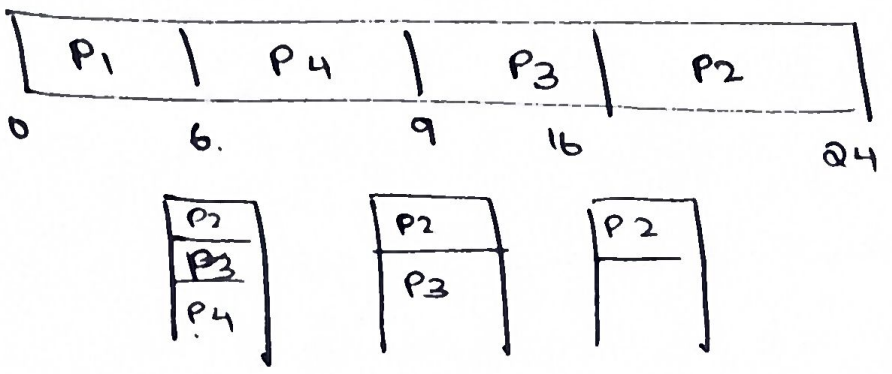


PID	AT	BT	ST	ET	ST-AT WT	ET-AT TAT	ST-AT RT
P4	0	3	0	3	0	3	0
P1	0	6	3	9	3	9	3
P3	0	7	9	16	9	16	9
P2	0	8	16	24	16	24	16
					7	13	7

(2)

PID	AT	BT
P1	0	6
P2	2	8
P3	4	7
P4	5	3

Gantt Chart

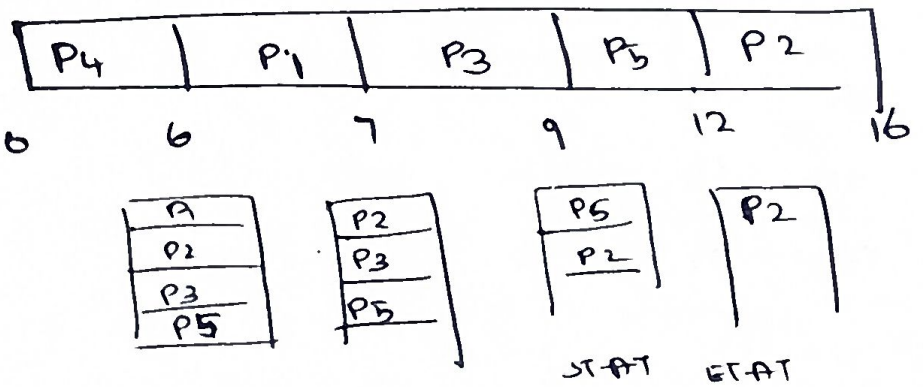


PID	AT	BT	ST	ET	ST-AT WT	ET-AT TAT	ST-AT RT
P1	0	6	0	6	0	6	0
P4	5	3	6	9	1	4	1
P3	4	7	9	16	5	12	5
P2	2	8	16	24	14	22	14
					5	11	5

(3)

PID	AT	BT	
P1	3	1	—
P2	1	4	
P3	4	2	—
P4	0	6	—
P5	2	3	

Gantt Chart



ST-AT ET-AT

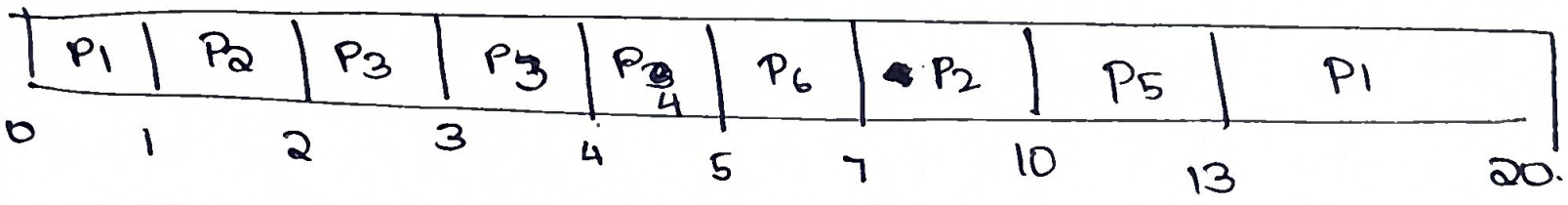
PID	AT	BT	ST	ET	WT	TAT	RT
P4	0	6	0	6	0	6	0
P1	3	1	6	7	3	4	3
P3	4	2	7	9	3	5	3
P5	2	3	9	12	7	10	7
P2	1	4	12	16	11	15	11
					4.8	8	4.8

B. Preemptive SJF | Shortest Remaining Time First

①

PID	AT	BT
P1	0	8
P2	1	4
P3	2	2
P4	3	1
P5	4	3
P6	5	2

Gantt Chart



TAT - BT

ET - AT

ST - AT

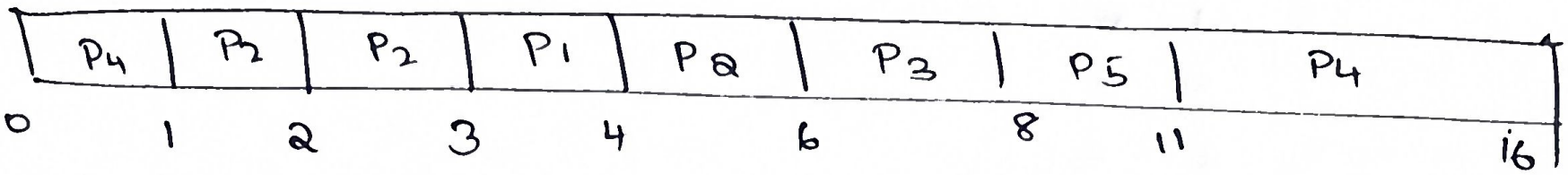
PID	AT	BT	ST	ET	WT	TAT	RT
P1	0	8	0	20	12	20	0
P2	1	4	1	10	5	9	0
P3	2	2	2	4	0	2	0
P4	3	1	4	5	1	2	1
P5	4	3	10	13	6	9	6
P6	5	2	5	7	0	2	0
					4	7.3	1.3

②

PID	AT	BT
P1	3	1
P2	1	4
P3	4	2
P4	0	6
P5	2	3

Gantt @hart

	AT	BT
P_1	3	~~x~~ 0
P_2	1	~~4~~ ~~3~~ 2 0
P_3	4	2 0
P_4	0	6 5
P_5	2	3 0

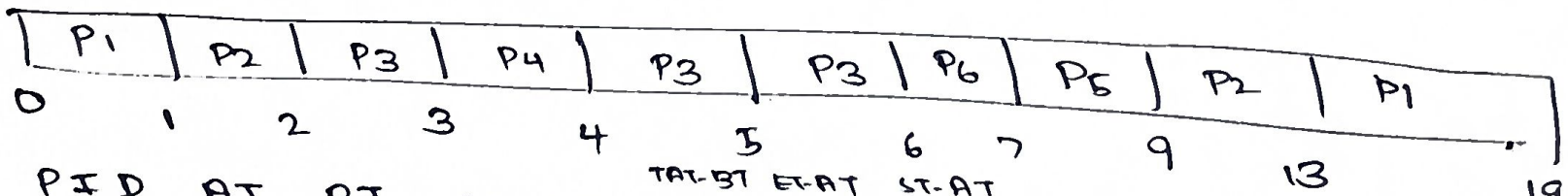


PID	AT	BT	ST	ET	TAT-BT WT	ET-AT TAT	ST-AT RT
P_1	3	1	3	4	0	1	0
P_2	1	4	1	6	1	5	0
P_3	4	2	6	8	2	4	2
P_4	0	6	11	16	10	16	11
P_5	2	3	8	11	6	9	6
					3.8	7	3.8

(3) PID AT BT

P_1	0	~~7~~	6
P_2	1	~~5~~	4
P_3	2	~~3~~ ~~2~~ 1	0
P_4	3	X	0
P_5	4	2	0
P_6	5	X	0

Gantt @hart

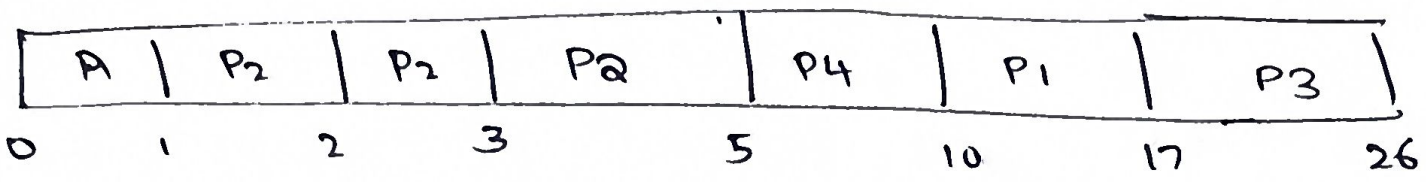


PID	AT	BT	ST	ET	TAT-BT WT	ET-AT TAT	ST-AT RT
P_1	0	7	0	19	12	19	0
P_2	1	5	1	13	7	12	0
P_3	2	3	2	6	1	4	0
P_4	3	1	3	4	0	1	0
P_5	4	2	7	9	3	5	3
P_6	5	1	6	7	1	2	1
			0		4	7.17	0.66

(4)

PID	AT	BT
P1	0	8
P2	1	4
P3	2	9
P4	3	5

Gantt Chart



TAT-BT ET-AT ST-AT

PID	AT	BT	ST	ET	WT	TAT	RT
P1	0	8	0	17	9	17	0
P2	1	4	1	5	0	4	0
P3	2	9	17	26	16	24	15
P4	3	5	5	10	2	7	2
					6.5	13	4.25

* Priority Scheduling

→ A priority number is associated w/ each process

→ CPU is allocated to the process w/ the highest priority (smallest integer = highest priority)

Problems: Starvation – low priority processes may never execute

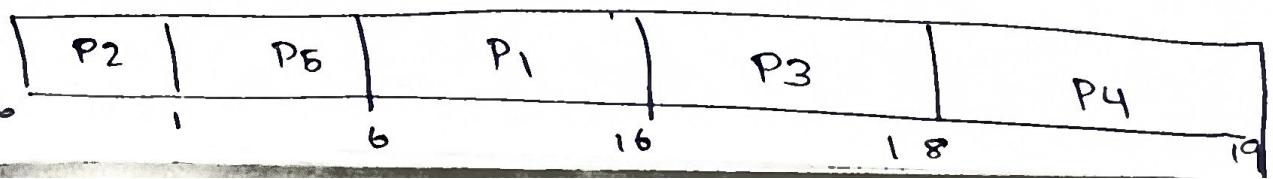
A. Non-Preemptive Priority

Solution : Aging – increase priority as time progresses

(1)

PID	BT	P
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

Gantt Chart



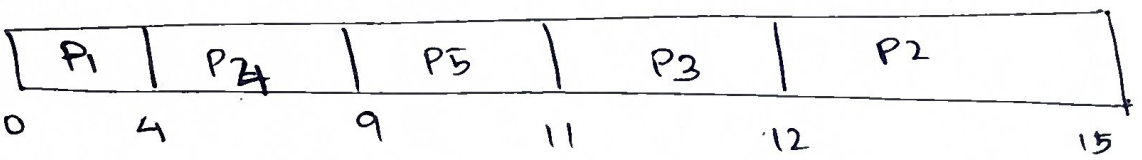
PID	AT	BT	ST	ET	ST-AT WT	ET-AT TAT	ST-AT RT
P_2	0	1	0	1	0	1	0
P_5	0	5	1	6	1	6	1
P_1	0	10	6	16	6	16	6
P_3	0	2	16	18	16	18	16
P_4	0	1	18	19	18	19	18
					8.2	12	8.2

(2)

PID	AT	BT	Priority	
P_1	0	4	2	-
P_2	1	3	3	
P_3	2	1	4.	
P_4	3	5	5	-
P_5	4	2	5.	-

(assume that higher no = higher priority)

Gantt Chart

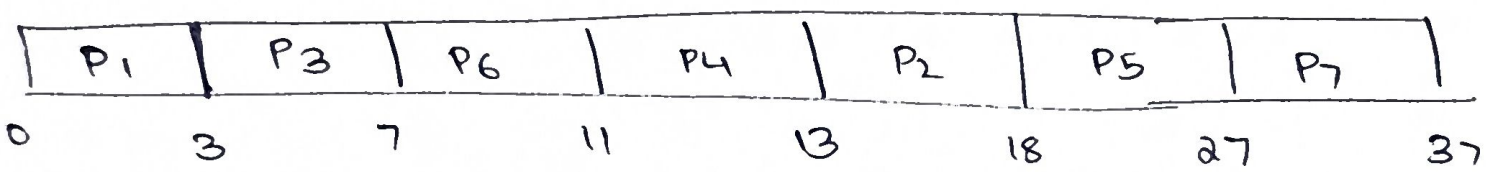


PID	AT	BT	ST	ET	ST-AT WT	ET-AT TAT	ST-AT RT
P_1	0	4	0	4	0	4	0
P_2	1	3	12	15	11	14	11
P_3	2	1	11	12	9	10	9
P_4	3	5	4	9	1	6	1
P_5	4	2	9	11	5	7	5
					5.2	8.2	5.2

(3) PID AT BT Priority (small no => larger priority)

P_1	0	3	2
P_2	2	5	6
P_3	1	4	3
P_4	4	2	5
P_5	6	9	7
P_6	5	4	4
P_7	7	10	10

Gantt Chart



PID	AT	BT	~~Priority~~	ST	ET	WT	TAT	RT
P_1	0	3		0	3	0	3	0
P_2	2	5		13	18	11	16	11
P_3	1	4		3	7	2	6	2
P_4	4	2		11	13	7	9	7
P_5	6	9		18	27	12	21	12
P_6	5	4		7	11	2	6	2
P_7	7	10		27	37	20	30	20
						7.7	13	7.7

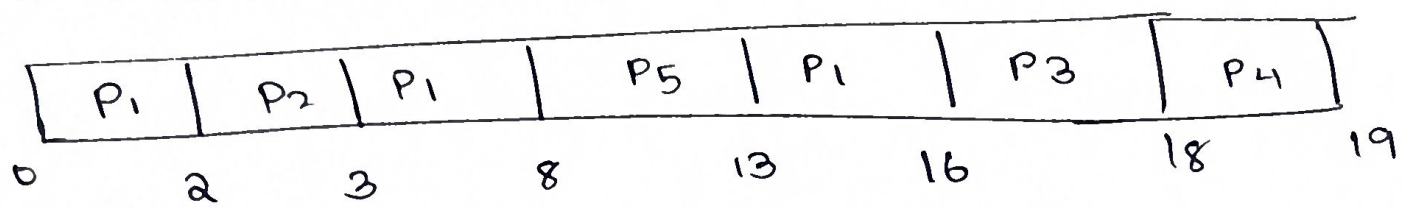
B. Preemptive Priority Scheduling

(1)

PID	AT	BT	Priority
P_1	0	10	3
P_2	2	1	1
P_3	3	2	4
P_4	8	1	5
P_5	8	5	2

PID	AT	BT	Priority
P1	0	10 ~~8 3~~	3
P2	2	~~1~~ 0	1
P3	3	2 ~~0~~	4
P4	~~4~~	1	5
P5	8	~~5~~ 0	2

Gantt Chart



BT AT-BT ET-AT ST-AT

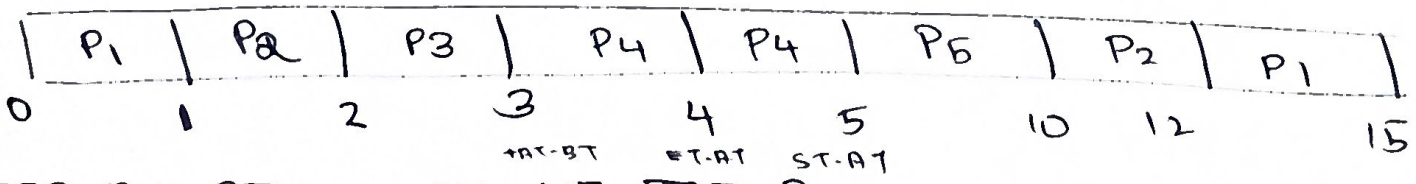
PID	AT	BT	ST	ET	WT	TAT	RT
P1	0	10	0	16	6	16	0
P2	2	1	2	3	0	1	0
P3	3	2	16	18	13	15	13
P4	8	1	18	19	10	11	10
P5	8	5	8	13	0	.5	0
					5.8	9.6	4.6

(2)

PID	AT	BT	Priority
P1	0	~~4 3~~	2
P2	1	~~3~~ 2	3
P3	2	~~1~~ 0	4
P4	3	~~2~~ 1 0	5
P5	4	5	5

(Higher no = higher priority)

Gant Chart



TAT-BT ET-AT ST-AT

PID	AT	BT	ST	ET	WT	TAT	RT
P1	0	4	0	15	11	15	0
P2	1	3	1	12	8	10	0
P3	2	1	2	3	0	1	0
P4	3	2	3	5	0	2	0
P5	4	5	5	10	1	6	1
					4	7	0.2

WT = TAT - BT

TAT = ET - AT

RT = ST - AT

* Round Robin Scheduling

→ Each process gets a small unit of CPU time (time quantum q)

→ After this time has elapsed, the process is preempted & added to the end of the ready queue, i.e time interrupts every quantum to schedule the next process

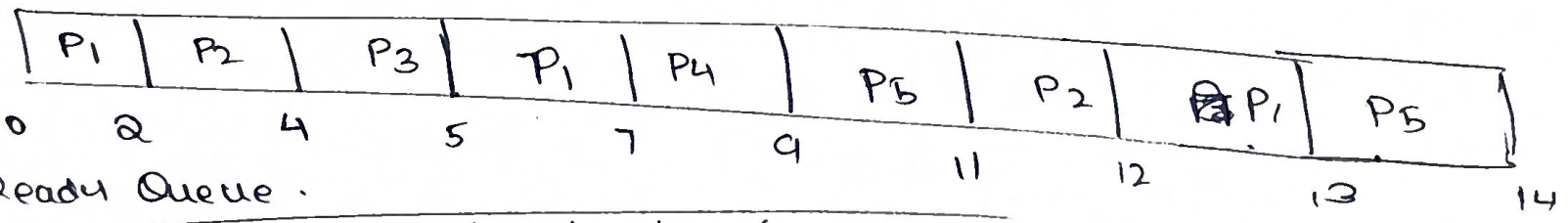
→ With a decreasing value of time quantum, the no. of context switches increases, the response time decreases ⇒ chances of starvation decreases

(1)

PID	AT	BT
P1	0	5 3 1 0
P2	1	3 1 0
P3	2	1 0
P4	3	2 0
P5	4	3 1

q = 2

Gantt Chart



Ready Queue.

P1 P2 P3 P1 P4 P5 P2 P1 P5

PID	AT	BT	ST	ET	WT (TAT-BT)	TAT (ET-AT)	RT (ST-AT)
P1	0	5	0	13	8	13	0
P2	1	3	2	12	8	11	1
P3	2	1	4	5	2	3	2
P4	3	2	7	9	4	6	4
P5	4	3	9	14	7	10	5
					5.8	8.6	2.6

(2)

PID	AT	BT	
P1	0	4	4 0 —
P2	1	5	5 3 1 0 —
P3	2	2	2 0 —
P4	3	1	1 0 —
P5	4	6	6 4 2
P6	6	3	3 1 0

q = 2

Gantt Chart

P1	P2	P3	P1	P4	P5	P2	P6	P5	P2	P6	P5

0 2 4 6 8 9 11 13 15 17 18 19 21

Ready Queue

P1 P2 P3 P1 P4 P5 P2 P6 P5 P1 P6 P5

PID	AT	BT	ST	ET	WT	TAT (ET-AT)	RT
P1	0	4	0	8	4	8	0
P2	1	5	2	18	12	17	1
P3	2	2	4	6	2	4	2
P4	3	1	8	9	5	6	5
P5	4	6	9	21	13	17	5
P6	6	3	13	19	10	13	7
					9.2	10.83	3.3

check

(3)

PID	AT	BT	
P1	5	5 1 0	—
P2	4	6 3 0	—
P3	3	7 4 1 0	—
P4	1	9 6 3 0	—
P5	2	2 0	—
P6	6	3 0	—

q = 3

Gantt Chart

	P4	P5	P3	P2	P4	P1	P6	P3	P2	P4	P1	P3

0 1 4 6 9 12 15 18 21 24 27 30 32 33

Ready Queue

P4 P5 P3 P2 P4 P1 P6 P3 P2 P4 P1 P3

PID	AT	BT	ST	ET	WT	TAT	RT
P_1	5	5	15	32	22	27	10
P_2	4	6	9	27	17	23	5
P_3	3	7	6	33	23	30	3
P_4	1	9	12	30	20	29	11
P_5	2	2	4	6	2	4	2
P_6	6	3	18	21	12	15	12
					16	21.33	7.16

* Multilevel Queue

→ Ready queue is partitioned into 2 separate queues

(i) foreground (interactive)

(ii) background (batch)

→ Each process is permanently given a queue.

→ Each queue has its own scheduling algorithm

(i) foreground - RR

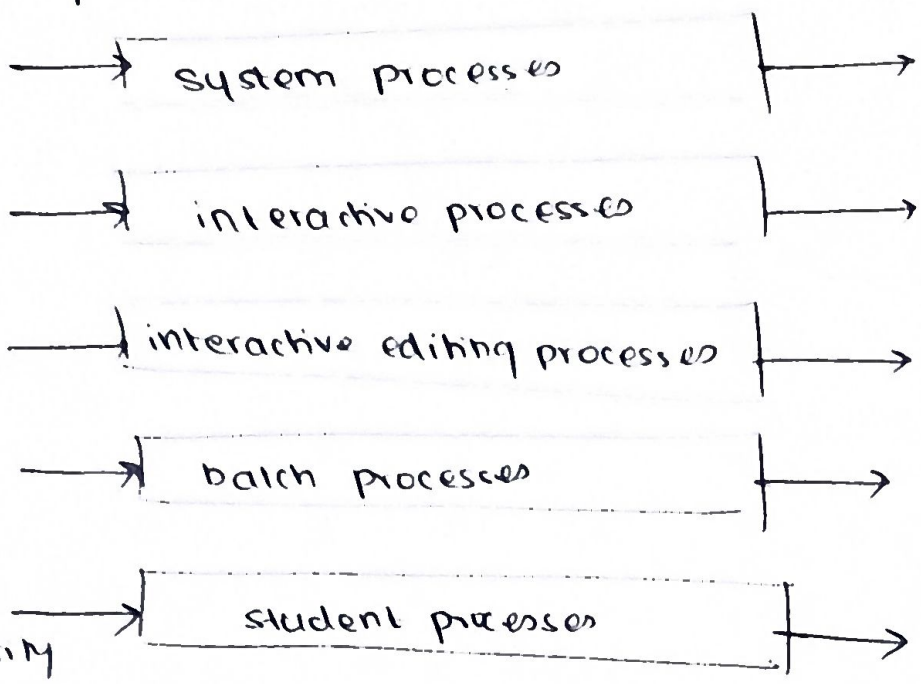
(ii) background - FCFS

→ Scheduling must be done between the queues as well:

A. Fixed Priority Scheduling - serve all from foreground, then background

B. Timeslice - each queue gets a certain amount of time which it can schedule among its processes. eg 80% FG RR & 20% BG FCFS.

Highest Priority



* Multilevel Feedback Queue

→ A process can move between the various queues- aging can be implemented in this way.

→ A multilevel queue is defined by the following parameters:

(i) no. of queues

(ii) scheduling algorithms for each queue

(iii) method used to determine when to upgrade a process

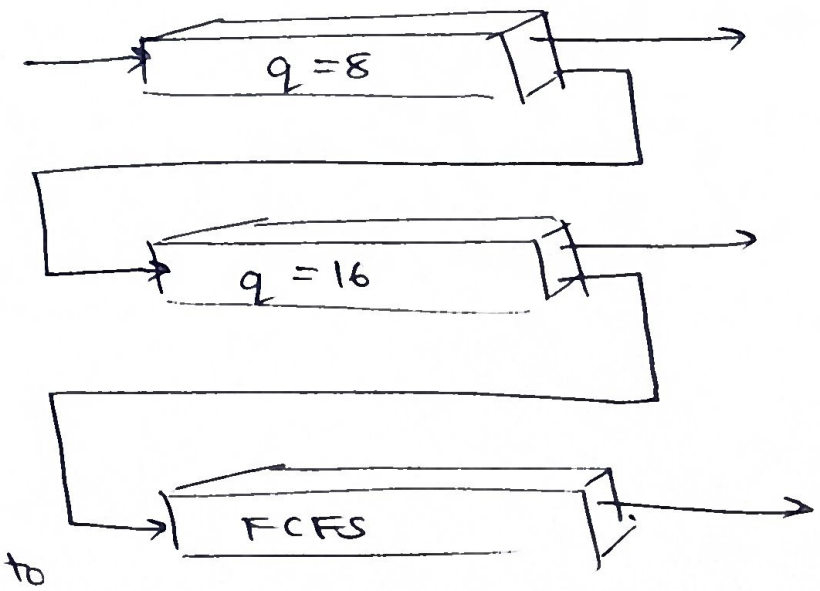
(iv) when to demote a process

(v) to determine which queue a process will enter when it needs service.

Eg. Q_0 = RR, with $q = 8$ ms

Q_1 = RR, with $q = 16$ ms

Q_2 = FCFS



Scheduling : → A new job enters Q_0, which is served ~~FCFS~~ RR, gets 8 ms.

→ If not done, moves into Q_1, gets 16 ms

→ If still not done, preempt & move to Q_2.

* Real Time CPU Scheduling

a. Soft Real-Time Systems - no guarantee as to when a critical real time process will be scheduled.

b. Hard Real Time Systems - Task must be serviced by its deadline

→ Two types of latencies affect performance

(i) interrupt latency - time from arrival of interrupt to start of routine that services interrupt

(i) dispatch latency - time taken for schedule to take current process off CPU and switch to another.

conflicts in dispatch latency

A. preemption of any process running in kernel mode

B. release resources of low priority resources needed by high priority processes

* Priority-Based Scheduling

→ Real time schedulers must support preemptive, priority-based scheduling (but this only guarantees soft real time)

→ With hard real-times, deadlines also must be met.

→ Processes now have characteristics like

(i) period - p

(ii) processing time - t

(iii) deadline - d

$$0 \leq t \leq d \leq p$$

rate of periodic task = $1/p$

* Virtualization and Scheduling?

→ virtualization software schedules multiple guests onto CPU(s).

→ Each guest does their own scheduling. This can

(i) result in poor response time

(ii) affect time of day clocks in guests

(iii) undo good scheduling algo. efforts of guests

* Rate Monotonic Scheduling

→ priority assigned on the basis of the inverse of period

(shorter period ⇒ higher priority)

→ CPU Utilization = $\sum B_i / p_i$

→ worst case CPU utilization = $n(2^{1/n} - 1)$.

Eg1 $P_1 = 50$ $P_2 = 100$

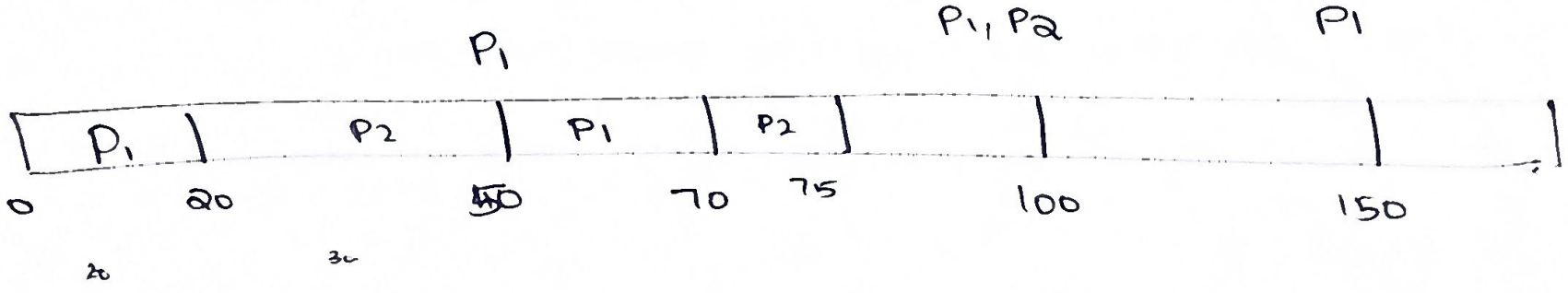
$t_1 = 20$ $t_2 = 35$

Sol

CPU utilization = $\left(\frac{20}{50}\right) + \left(\frac{35}{100}\right)$

$= 75/100 = 0.75$

P_1 has higher priority



20 35

eg 2 $p_1 = 50$ $p_2 = 80$

$t_1 = 25$ $t_2 = 35$

CPU Utilization: $\left(\frac{25}{50}\right) + \left(\frac{35}{80}\right)$

$= 0.5 + 0.4375$

$= 0.9375$

Worst case utilization : $n(2^{1/n} - 1)$

$= 2(2^{0.5} - 1)$

$= 0.8284$

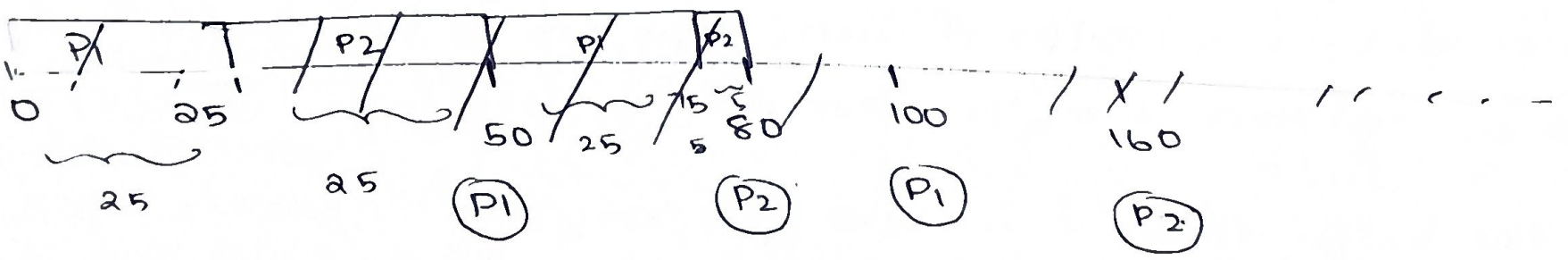
∴ cannot schedule to meet deadlines

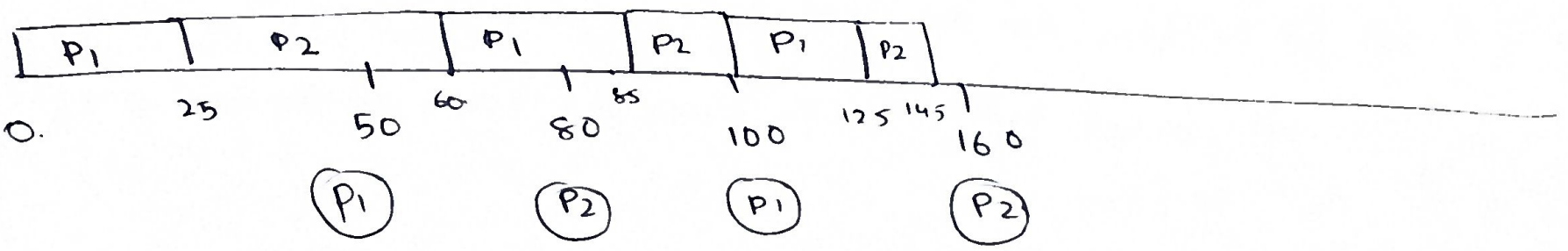
* Earliest Deadline First (EDF)

→ priorities are assigned on the basis of deadlines, the earlier the deadline, the higher the priority.

Example $p_1 = 50$ $p_2 = 80$

$t_1 = 25$ $t_2 = 35$





→ Unlike the rate monotonic algorithm, EDF does not require that processes be periodic, or must require a constant amt. of CPU time per burst.

→ The only requirement: process must announce its deadline to the scheduler, when it becomes runnable.

→ Theoretically optimal, but in practice, very difficult to achieve this level of CPU utilization, due to the cost of context switching between processes & interrupt handling.

* Process Synchronization

A. Producer-Consumer Problem

→ a common paradigm for cooperating processes

→ A producer produces info, that is consumed by a consumer process

Solution

→ Use a shared memory

→ To allow producer & consumer processes to run concurrently, there must be a buffer of items that can be filled by the producer, and emptied by the consumer.

→ The buffer resides in a region of memory that is shared by both the producer & the consumer.

→ 2 kinds of buffers can be used:

(i) unbounded buffer - no practical limit on the size of the buffer

(ii) bounded buffer - assumes a fixed buffer size.

Bounded Buffer Solution

→ The shared buffer is implemented as a circular queue, w| 2 pointers in and out.

→ The variable in points to the next free position in the buffa
out points to the first full position in the buffer

→ buffer empty ⇒ in == out

full ⇒ ((in+1) % buffer-size) == out.

Code

```
#define BUFFER_SIZE 10
typedef struct {
    .
    .
} item;
item buffer [buffer-size];
int in=0;
int out =0;
```

Producer Process

```
while (true) {
        while ((in+1) % buffer-size) == out);

        buffer [in] = next-produced;
        in = (in+1) % BUFFER-SIZE;
}
```

Consumer Access

```
item next_consumed;
while (true) {
    while (in == out) ;
    next_consumed = buffer[out]
    out = (out+1) % BUFFER_SIZE
```

Disadvantage: can store at most BUFFER_SIZE-1 items in buffer.

→ This can be remedied using an integer variable counter, set to 0.

→ counter is incremented every time we add a new item & decremented every time we remove one item.

Code

Producer

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

Consumer

```
while (true) {
    while (counter == 0);
    next_consumed = buffer[out];
    out = (out+1) % BUFFER_SIZE
    counter--;
}
```

* Race Conditions

→ Although the producer and consume routines are correct separately, they may not function correctly, when executed concurrently.

eg. If count = 5, if the ~~con~~ processes concurrently execute, counter++ & counter -- would result in the value of the counter variable becoming 4, 5 or 6.

This is because the internal implementation is as follows:

```
reg1 = counter
reg1 = reg1 + 1
counter = reg1
```

(OR)

```
reg2 = counter
reg2 = reg2 - 1
counter = reg2
```

→ These statements may get interleaved in an arbitrary manner.

→ A situation where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access take place, is called a race condition.

→ This means that only variable at a time should be allowed to manipulate the counter variable.

* Critical Selection Problem

→ Consider a system of n processes $\{P_0, P_1, P_2 \cdots P_{n-1}\}$

→ Each process has a critical section segment of code.

→ The may be changing common vars, updating table etc.

→ When 1 process is in the critical section, no other may be in it

critical section.

General Structure

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

Algorithms

Dekker's Algorithm 1 - Assigning turns

For Pi

```
do {
  while(turn != i);
        critical section
  turn = j
        remainder section
} while(true)
```

For Pj

```
do {
  while (turn != j);
        critical section
  turn = i
        remainder section
} while(true)
```

Test Cases

1. Pi enters CS → works, sets turn to j
2. Pj enters CS → works, provided Pi entered before
3. Pi wants to enter, then Pj wants to enter ⟷ Pi executes first then Pj
4. Pi & Pj want to enter at the same time → boils down to (3)
5. Pi enters, then Pi wants to enter again → does not work, since after exec. of Pi, it is Pj's turn = strict alternation

Failures : Mutual exclusion holds
Progress fails - strict alternation
Bounded wait - holds.

* Solution to the Critical Section Problem

1. Mutual Exclusion - If process P_i is executing in its critical section, then no other process can be executing in their critical sections.

2. Progress - The entering of another process into the critical section should not indefinitely be postponed, i.e one process does not block another process in entering its critical section

3. Bounded Waiting - A bound must exist on the no. of times other processes are allowed to enter their critical sections after another process has made a request to enter its critical section, before that request is granted.

* Dekker's Algorithm 2 - Usage of a flag variable

→ Flag is set to true, when a process expresses its intent to enter

→ Sets flag to false after <CS> is complete

→ initially, flag[i] & flag[j] = false

Process for P_i

```
do {
    flag[i] = true
    while (flag[j] == true);
    <critical section>
    flag[i] = false;
    <remainder section>
} while (True)
```

Process for P_j

```
do {
    flag[j] = true
    while (flag[i] == true);
    <critical section>
    flag[j] = false
    <remainder section>
} while (True);
```

Test Cases

(i) Pi WTE → works

(ii) Pj WTE → works

(iii) Pi WTE, then Pj WTE → works

(iv) Pi & Pj WTE at the same time → fail

(Both will express their intent to get in, set their flags to true. To each process, it would appear as if the other is in the critical section), and neither ends up entering)

(v) Pi WTE, Pi WTE → works

∴ Satisfies mutual exclusion, but not progress - goes into a deadlock when both processes want to enter at the same time.

* Peterson's Solution

→ The 2 processes share 2 variables

```
int turn;
Boolean flag[2]
```

turn → whose turn it is to enter the critical section

flag → indicates whether a process is ready to enter the critical section.

For Pi

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

        <CS>
    flag[i] = false

        <remainder>
} while (true);
```

For Pj

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

        <CS>
    flag[j] = false

        <remainder>
} while (true);
```

→ All test cases hold good

→ Mutual exclusion, progress & bounded-wait are all satisfied

* Synchronization - Hardware

→ hardware support for implementing critical section code

→ critical regions protected using locks

Solution to Critical Section Problem using Locks

```
do {
    | acquire lock |
            critical section
    | release lock |
            remainder section
} while (True);
```

① Test and Set Instructions

```
boolean test_and_set (boolean *lock) {
    boolean oldvalue = *lock;
    *lock = True;
    return oldvalue;
}
```

Shared boolean variable lock.

initialized to False

```
do {
    while ( test_and_set ( &lock) == True );
            <cs>
    lock = false;
            <rs>
} while (true);
```

(2) Compare & Swap Instructions

0.0.1

```
int compare_and_swap (int * lock, int expected_false, int
                                          new_value_true) {
    int temp = * lock;
    if ( * lock == expected_false)
        * lock = new_value_true;
    return temp;
```

Shared integer lock assigned to 0.

```
do {
    while (compare_and_swap( & lock , 0,1) != 0) ;
                    < CS >
    lock = 0;
                    < RS >
} while (true);
```

* Mutex locks

→ designed by OS designers to solve the critical section problem.

→ mutex locks protect critical regions & thus prevent race conditions.

→ A process acquires a lock before entering a CS, it releases the lock when it exits.

```
acquire() {
    while ( !available );
    available = false;
}
```

```
release {
    available = true;
}
```

```
do {
        acquire lock
            critical section
        release lock
            remainder section
} while (true);
```

Note: This type is lock is called a spinlock because the process continually waits (spins), while waiting for the lock to become available.

* Semaphores

→ A semaphore is a variable that is non-negative and shared between threads, used to solve the critical section problems.

→ A semaphore S is an integer variable, that apart from initialization can be accessed only through 2 standard atomic operations wait() & signal().

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

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

* Types of semaphores

A. Counting Semaphore

→ value can range over an unrestricted domain

→ used to control access to a given resource (which has a finite no. of instances

→ initial value of semaphore = no. of resources available

→ when a resource is used wait() decrements S

→ when the count is 0, all the resources are beingused

B. Binary Semaphores

→ value can range only between 0 & 1.

→ similar to mutex locks

* Semaphore Implementation

w/o busy wait

→ maintain a queue of processes that have been put to sleep.

```
typedef struct {
    int value;
    struct process *list;
} semaphore

wait(semaphore *s) {
    s->value--;
    if (s->value < 0) {
        add this process to s->list;
        block()
    }
}

signal(semaphore *s) {
    s->value++;
    if (s->value <= 0) {
        remove a process P from s->list
        wakeup(P);
    }
}
```

* Disadvantages and Starvation & Deadlock

→ Implementation of a semaphore w/ a waiting queue may result in a situation where 2 or more processes are waiting indefinitely for an event, that can be caused only by the waiting process

→ deadlock condition

P_0	P_1
wait(S)	wait(Q)
wait(Q)	wait(S)

→ P_0 runs wait(S)

→ P_1 runs wait(Q)

Wlen P_0 executes wait(Q), it must wait until P_1 executes a signal(Q).

Wlen when P_1 executes wait(S), it must wait until P_0 executes signal(S).

→ This is a deadlocked condition.

Note: There may also be indefinite blocking or starvation, where processes wait indefinitely within the semaphore.

→ This may happen if we remove processes from the list in a FIFO order.

* Bounded Buffer Problem using Mutex locks

3 semaphores are used:

(i) m (mutex) => a binary semaphore used to acquire & release the lock.

(ii) empty - a counting semaphore, maintains no. of empty slots

(iii) full - a counting semaphore (initially 0), keeps track of no of full slots.

Producer

```
do {
    wait (empty);    // wait until empty > 0, then decrement empty
    wait (mutex)     // acquire lock
    // add data to buffer
    signal (mutex)   // release lock
    signal (full)    // increment full
} while(True)
```

Consumer

```
do {
    wait (full)      // wait until full > 0, then decrement full
    wait (mutex)     // acquire lock
    // remove data from buffer
    signal (mutex)   // release lock
    signal (empty)   // increment empty
} while (True)
```

* Deadlock Characterization

Deadlocks can occur if these condns occur simultaneously

(i) mutual exclusion - only one process at a time can use a resource

(ii) hold & wait - a process holding at least one resource is waiting to acquire additional resources held by other processes

(iii) no preemption - a resource can be released only voluntarily by the process holding it, after that process has completed its tasks.

(iv) circular wait - there exists a set $\{P_0, P_1, P_2 \ldots P_n\}$ of waiting processes such that P_0 is waiting for a resource held by P_1, P_1 is waiting for a resource held by P_2...

* Request Allocation Graph

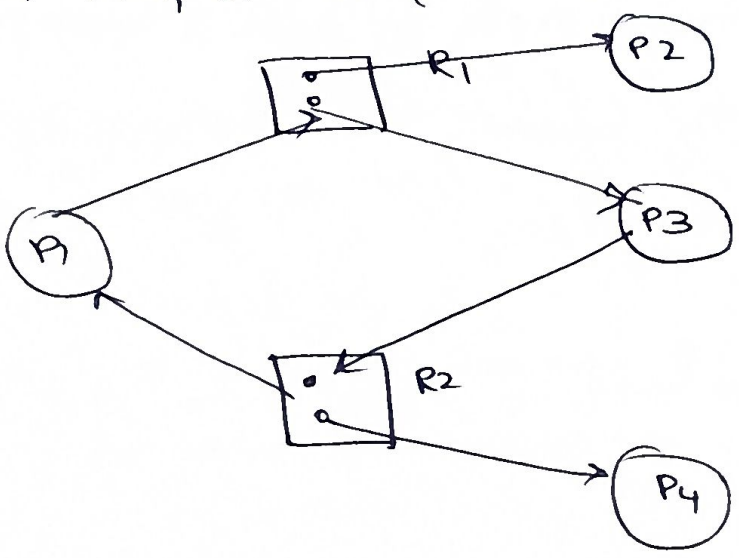
request edge : $P_i \longrightarrow R_j$

assignment edge : $R_j \longrightarrow P_i$

* Finding deadlocks

no cycles ⇒ no deadlock

cycles ⇒ may or may not have deadlock



has a cycle, but no deadlock

* Methods for handling deadlocks

① Deadlock Prevention

A. Mutual Exclusion

→ at least one resource must be non-shareable

→ shareable resources, do not require mutually exclusive access & cannot be involved in a deadlock.

eg. read-only files = shareable resource

B. Hold & Wait

→ Whenever a process requests for a resource, it should not hold any other resources.

Two options:

(i) process should request & be allocated its resource before it begins execution.

(ii) Should only request resources when it has none

C. No Preemption

→ If a process is holding some resources and requests another ~~request~~ resource, that cannot immediately be alloted to it, then all resources that the process is holding must be preempted.

D. Circular Wait

→ ensure a total ordering of resource types & require that each process requests resources in increasing order of enumeration.

② Deadlock Avoidance

→ requires that each process declare the maximum no. of resources it may need.

→ dynamically examines the resource allocation state, to ensure that there can never be a circular wait condition

* Safe State

→ The system is in safe state if there exists a sequence $<P_1, P_2 \ldots P_n>$ of all processes in the system, such that for each P_i, the resources that P_i can still request can be satisfied by the currently available resources + all that held by P_j

i.e.

→ if P_i's resources are not immediately available, then P_i can wait until all of P_j has finished.

→ Once P_j finishes, P_i can obtain the needed resources, execute, return allocated resources & terminate.

* Avoidance Algorithms

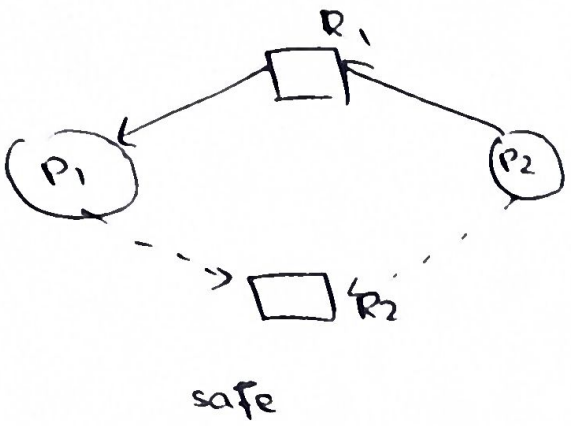
A. Resource Allocation graph (single resource, single instance)

→ use claim edge $P_i \rightarrow R_j$ indicates that the process P_i may request resource R_j

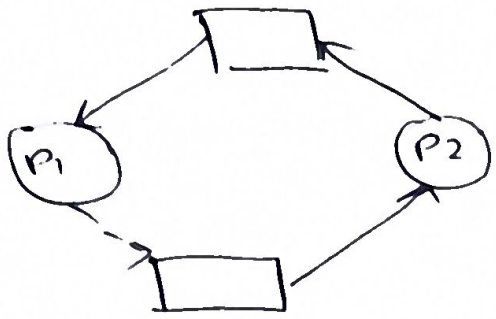
→ rep. by a dashed line

→ claim edge converted to request edge, when a process requests a resource.

→ Resource must be claimed a prior



safe



→ A request can be granted only if it does not result in the formation of a cycle.

B. Banker's Algorithm (multiple instances)

Data Structures

(i) Available matrix

(ii) Allocation matrix (how many instances are currently allocated)

(iii) Need matrix (how many more instances are needed to complete the task)

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

Process	Allocation	Max Need	Available	Need.	
	A B C	A B C	A B C	A B C	
P_1	0 1 0	7 5 3	3 3 2.	7 4 3	P_1
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		2 1 1	P_4 —
P_5	0 0 2	5 3 3		5 3 1	P_5 —
	7 2 5				

Total

P_2 → P_4 → P_5 → P_1 → P_3

Available = - Σ Allocation ~~Total~~

(10, 5, 7) − (7, 2, 5) = (3, 3, 2)

Need = Max Need − Allocation

Check if available ≥ need for each.

⇒ P_1 not satisfied

⇒ P_2 satisfied ⇒ add w/ allocation

⇒ P_3 ✗

⇒ P_4 ✓ ⇒ add w/ allocation

⇒ P_5 ✓

Q (2) Assume that there are 5 processes & 4 types of resources. At To, the system state is

A = 3 B = 17 C = 16 D = 12

Process	Allocation				Max Need.			
	A	B	C	D	A	B	C	D.
P_0	0	1	1	0	0	2	1	0
P_1	1	2	3	1	1	6	5	2
P_2	1	3	6	5	2	3	6	6
P_3	0	6	3	2	0	6	5	2
P_4	0	0	1	4	0	6	5	6

Use the safety algorithm to check if the system is safe or not.

Ans.

Process	Allocation				Max Need				Available				Need			
	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
P_0	0	1	1	0	0	2	1	0	1	5	2	0	0	1	0	0 ✓
P_1	1	2	3	1	1	6	5	2	1	6	3	0	0	4	2	1 ✓
P_2	1	3	6	5	2	3	6	6	1	12	6	2	1	0	0	1 ✓
P_3	0	6	3	2	0	6	5	2	1	12	7	6	0	0	2	0 ✓
P_4	0	0	1	4	0	6	5	6	2	14	10	7	0	6	4	2 ✓
	2	12	14	12					3	17	16	12				

A. available = Total - Σallocation

= (3, 17, 16, 12) - (2, 12, 14, 12)

= (1, 5, 2, 0)

B. Need matrix = max need - allocation

check if availability satisfies need.

P_0 - ✓ add alloc

P_1 - X

P_2 - X

P_3 - ✓ add alloc

P_4 - ✓ add alloc

P_1 - ✓ add alloc

P_2 - ✓ add alloc

safestate $< P_0, P_3, P_4, P_1, P_2 >$

Queue

P_0 ↓ P_3 ↓ P_4 ↓ P_1 ↓ P_2

a. P_1 requests $(2,1,1,0)$ ⇒ cannot satisfy

b. P_1 requests $(0,2,1,0)$ ⇒ can satisfy,

(add to corresponding allocation)

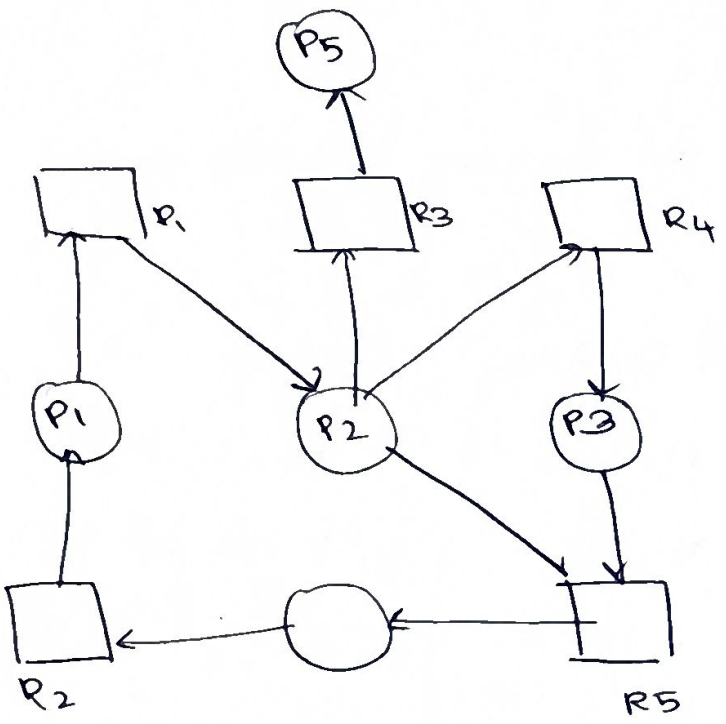
proceed w/ safety algorithm

(3) Deadlock Detection

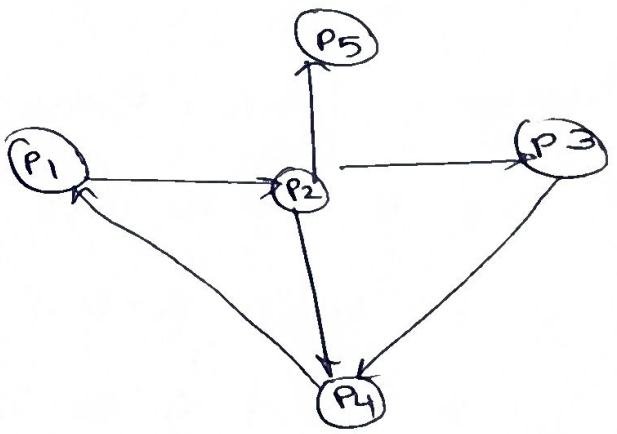
→ allows system to enter deadlock state

single instance of each type ⇒ make a wait-for graph

check for cycle in wait-for graph.



remove resources, connect processes



deadlock in P_1, P_2, P_3, P_4

For multiple instances, use the detection algo- same as banker's

(4) Deadlock Recovery

A. Process Termination

→ abort all deadlocked processes

→ abort one process at a time until the deadlock cycle is eliminated

order to abort

(i) priority

(ii) how long process has computed, how much longer left

(iii) resources used

(iv) resources needed to complete

B. Resource Preemption

1. Select victim - choose resource / process to be preempted based on minimum cost.

2. Rollback - return to some safe state, restart process from that state.

3. Starvation - same process may always be picked as victim

soln: include a no. of rollbacks in the cost factor.