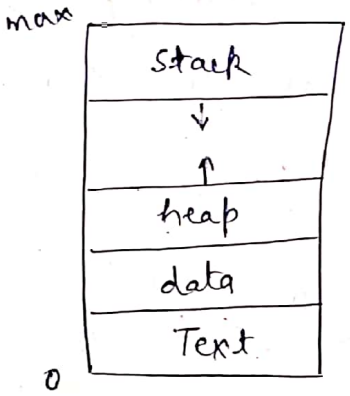
UNIT - III

The Process :-

- A process is more than the program code, which is known as the text section.
- It includes the current activity represented by the program counter.



Process in memory

- Also includes process stack, which contains temporary data (functions parameters, local variables etc).
- A data section which contains global variables.
- ~~Also includes data section, which contains~~
- Also includes heap, which is memory that is dynamically allocated during process run time.

Process state :-

- As a process executes, it changes state and state is defined in part by the current activity of a process.

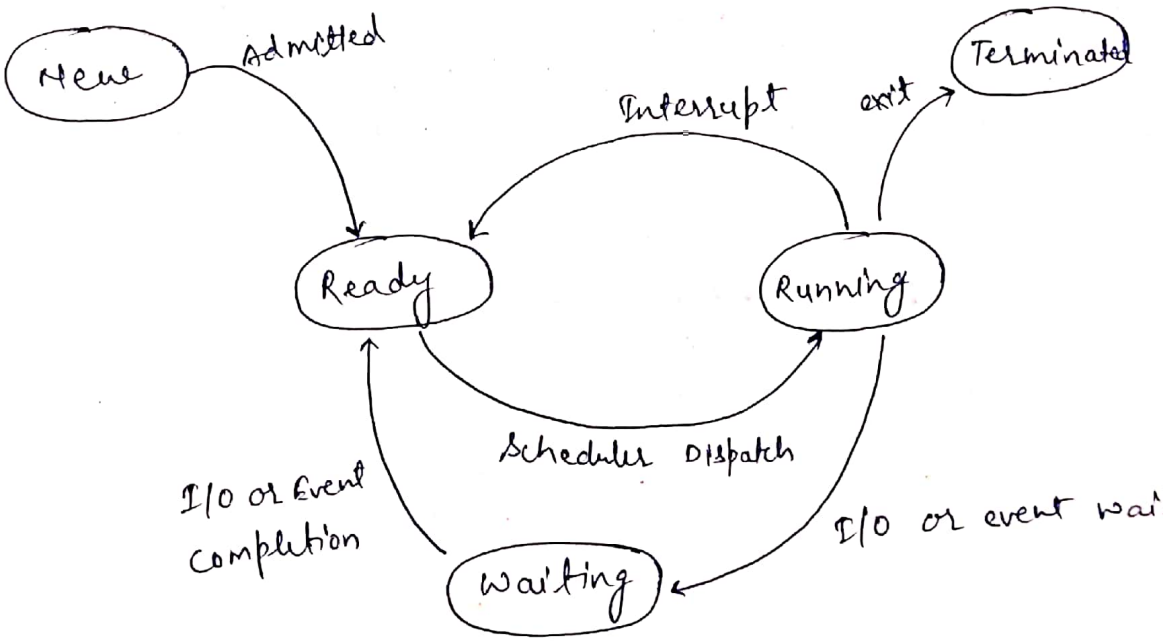
(i) **New** :- The process is being created.

(ii) **Running** - Instructions are being executed.

(iii) **Waiting** - The process is waiting for some event to occur (such as an I/O completion or reception of a signal).

(iv) **Ready** The process is waiting to be assigned to a processor.

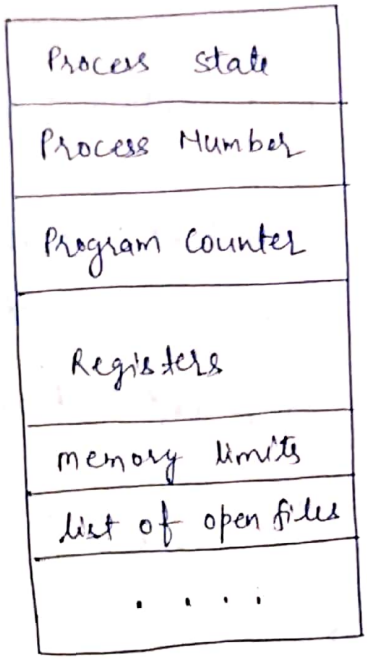
(v) **Terminated** The process has finished execution.



Process Transition Diagram

Process Control Block

- Each process is represented in the operating system by a process control block (PCB) or task control block.



- Process State - The state may be new, ready, running waiting, halted and so on.
- Program Counter - The counter indicates the address of the next instruction to be executed for this process.
- CPU Registers - Includes accumulators, index registers, stack pointers, and general purpose registers plus any condition-code information.
  - Along with the program counter, this state information

must be saved when an interrupt occurs, to allow the process to be continued correctly afterward.

- **CPU scheduling information :→**
  - This information includes a process priority, pointers to scheduling queues and any other scheduling parameters.
- **Memory-management information :→**
  - This information may include such information as the value of the base and limit registers, the page tables or the segment tables, depending on the memory system used by the operating system.
- **Accounting information :→**
  - Includes the amount of CPU and real time used, time limits, account numbers, job or process numbers and so on.
- **I/O status information :→**
  - Includes the list of I/O devices allocated to the process, a list of open files and so on.

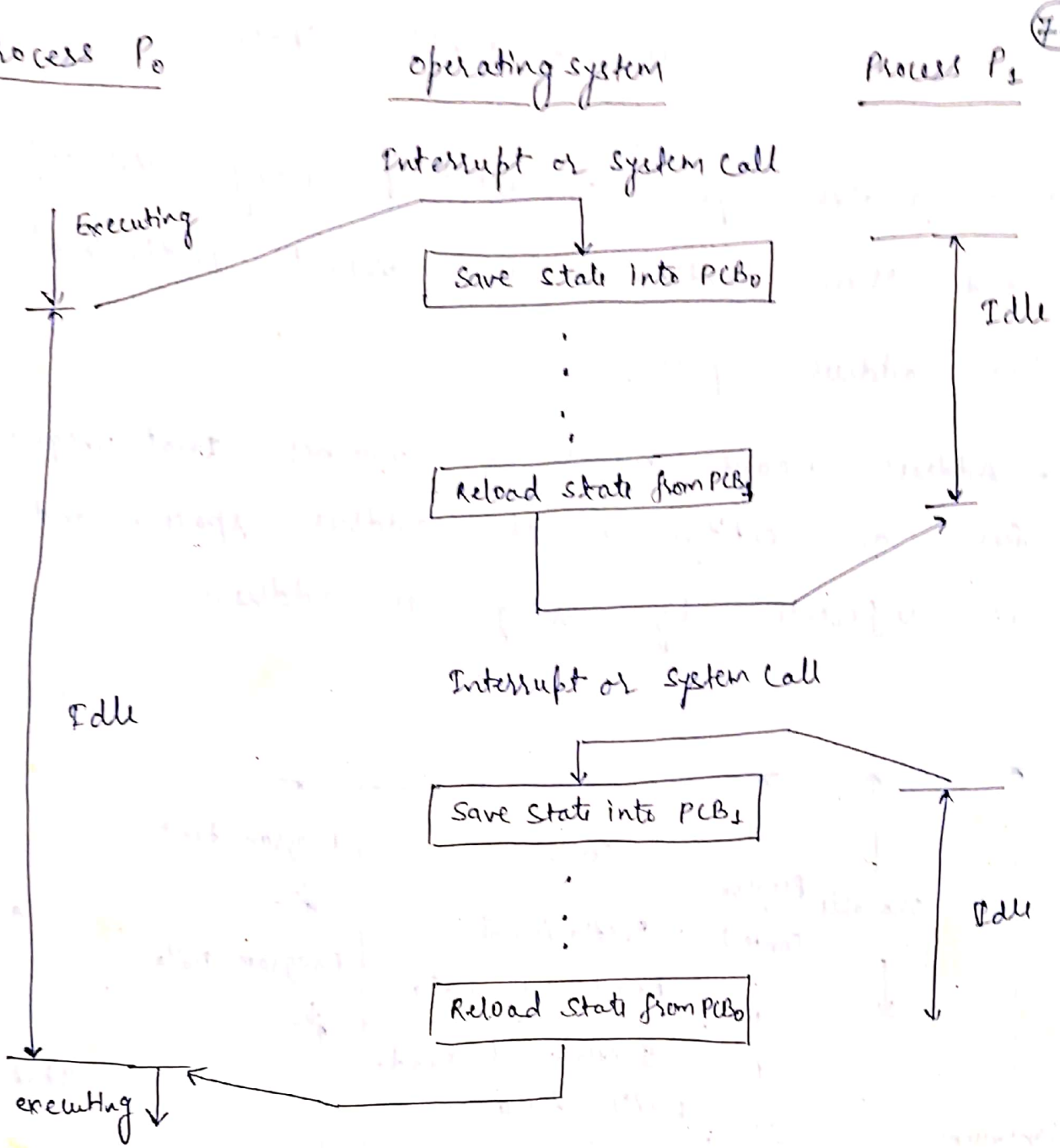
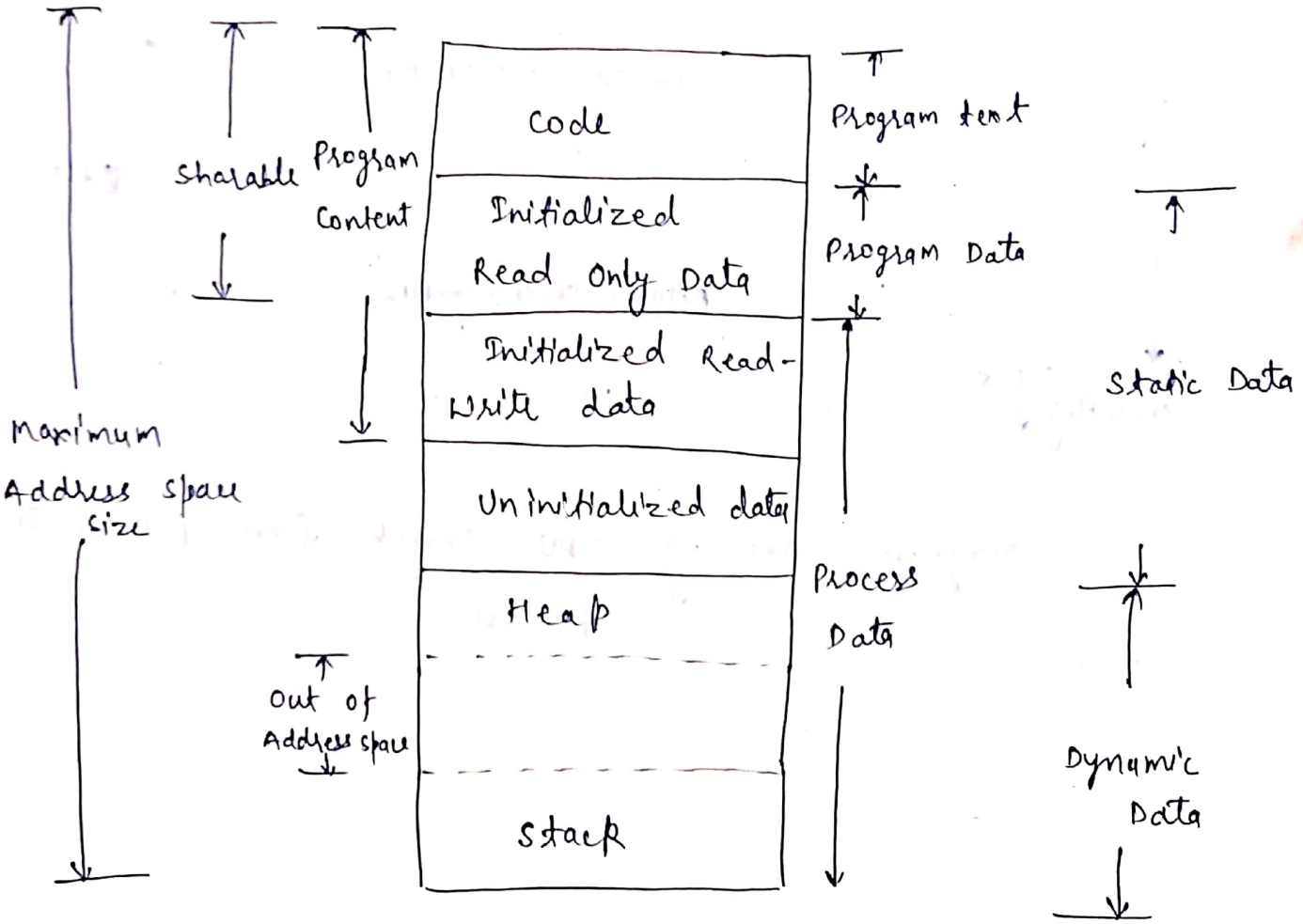


Diagram showing CPU switch from process to process

Process Address Space

- A address space consists of many different entities and these entities have distinct individual addresses in address space.
- Address means names or number that uniquely identifies an entity in the address space. and can only be referenced by using its address.



Logical Address Space of a process

- An address space is typically partioned into a number of blocks of variable sizes.
- In UNIX systems a private address space is composed of four blocks
  (a) Code (b) Data (c) Stack (d) Heap

- Code Block :→
  - Also called the text section and contains machine instructions that the CPU can execute.
  - It is composed of codes from an executable image and many libraries.
  - Code sharing is automatically setup by the operating system.
  - The process has its own copy of the block text section.
  - Writes to the block are considered illegal and are trapped by the operating system to forcefully terminate the process.

- **Data block :→**
  - Contains the program's own static data.
  - Divided into three sub-blocks ; (a) The 'initialized read-only' ; (b) Initialized read-write ; (c) Uninitialized
  - (a) It is initialized by the program : its value is determined at the program compilation time and cannot be changed.
    - shared by multiple processes.
  - (b) data is initialized by the program.
  - (c) data is not kept as a part of the compiled program and hence the process/program does not initialized it.
  - Uninitialized program data are initialized to zeros when the process address space is setup.
- **Stack block :-**
  - The dynamic data of a process is partioned into two different blocks : Stack and Heap.
  - A stack is a data structure in which entries

are added to and removed from one end, which is called the stack top.

- Push and Pop operations are implemented to access a stack and expand and shrink the process address space respectively.
- Every process has a private stack, which is used to store local variables and parameter values.
- It is not the program's static data and is required only at runtime.

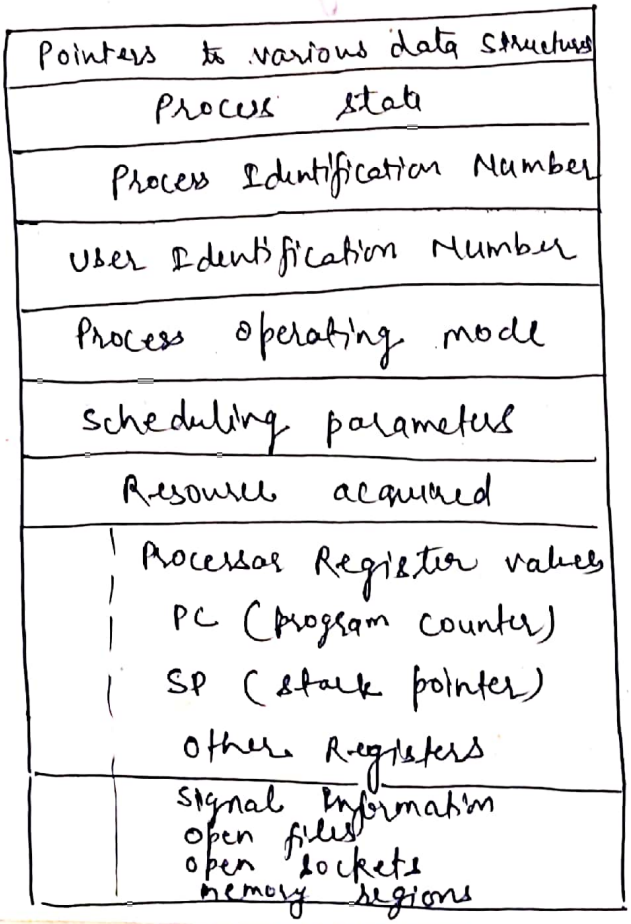
- **Heap Block :→**
  - Is used to allocate more data entities dynamically to the process address space.
  - These are not the static data of the program and are required only at runtime.
- The allocation and deallocation of space to heap expands and shrinks the process address space.

Process Identification Information

- In a multiprocess system, to interact with a process we need to identify the particular process.
- A process is uniquely identified by its pid - the process identifier.
- It is normally a positive integer number that uniquely identifies a single process from all processes currently active in the system.
- The operating system assigns a unique pid value to a process when it creates the process and this value remains unchanged until it destroys the process.
- Once the system allocates a process descriptor for a newly created process, its location does not generally change.
- Consequently, the address of a descriptor may be used to identify a process.
- Because of the limitation of physical resources (especially the main memory) we may need to recycle a limited number of descriptors for new processes.

so within a short span of time, many different processes may use the same descriptor one after another which might make it difficult to distinguish a terminated process from a new process.

- Thus pid values are kept independent of process descriptor addresses and values are stored in process descriptors.
- Given a pid value, we need to find the corresponding process.
- We need a map between active pid values and the corresponding process descriptors.
- The operating system implements this map from pid values to process descriptors.



Structure of a process descriptor

User Identifier :→

- Each process is owned by a legal user.
- The identity of the owner is stored in a process attribute, named user identifier (uid).

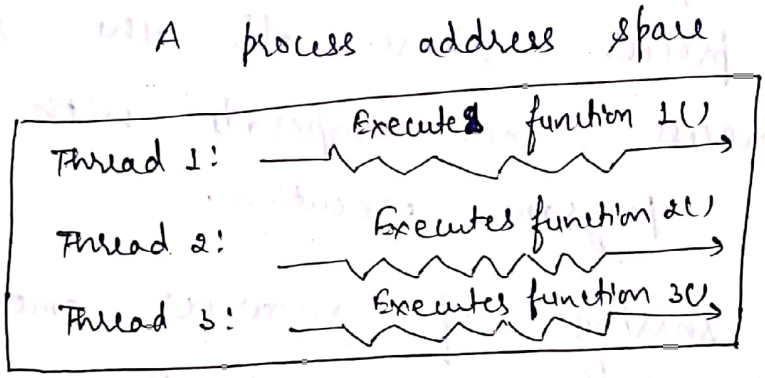
Process Group Identifier :→

- A process group is a collection of processes that are engaged in performing a common task.
- The group is uniquely identified by what we call a process group identifier (pgid).
- Processes in a group may have the same pgid value but will have different pid values.
- The pgid is primarily used for interprocess communication.

Threads and their Management

Thread and Process Relationship : →

- A thread is an independent strand in a process, and the process is said to own that thread and the thread is contained on the process.



- Thread separate the notion of program execution from the rest of the traditional definition of a process.
- A single thread executes a program concurrently with other threads in the same process.
- A thread always runs within a process. A process may have many threads, but a thread ~~has~~ always has one container process.
- A traditional process is one that has a single thread.

- A thread termination does not necessarily imply that the containel process execution is over.

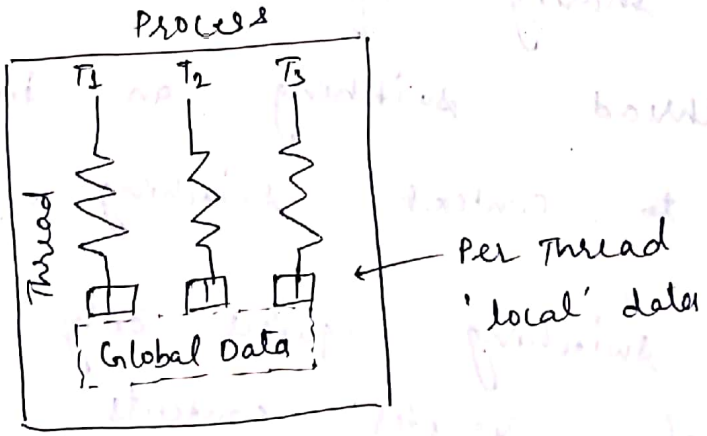
Benefits of Threads

- Process threading is a convenient for exploiting concurrency within the same process.
- One may use threads when a program has many independent tasks that can be executed concurrently.
- Threads acquire resources in the content of the container process, share all the resources allocated to the process, and cooperate with one another in a single program execution.
- Extensive sharing of resources among threads makes thread creation / destruction an inexpensive operation compared to process creation / destruction.
- Creation of a process involves setting up a new address space.
- A thread can creat another thread and they are considered sibling threads.
- There is no parent – child relationship among threads.

- All threads share the same address space and all resources (open files, memory regions) acquired by the process.
- Each thread needs some private space with in the process and has its own copy of PC, SP and other general purpose & CPU registers.
- Each thread has its own state information, and different threads in a process can be in different states at the same time.

Thread Synchronization

- The term thread gets its significance only when the system is multithreaded.



- In multithreaded system, threads within a process can independently perform different tasks at the same time.

- Access to the global data and resources may need to be synchronized.
- Application programs themselves, and not the operating system, provide this synchronization.

Thread Management

- When the need arises the CPU is switched among sibling threads.
- This is called thread switching, in contrast to context switching where the CPU is switched between (threads of) two different processes.
- Extensive sharing of resources among sibling threads makes thread switching an inexpensive operation compared to context switching.
- Thread switching requires only switching between subsets of register contents, including PC and SP registers.
- Thread switching can be implemented in user space or kernel space or both.

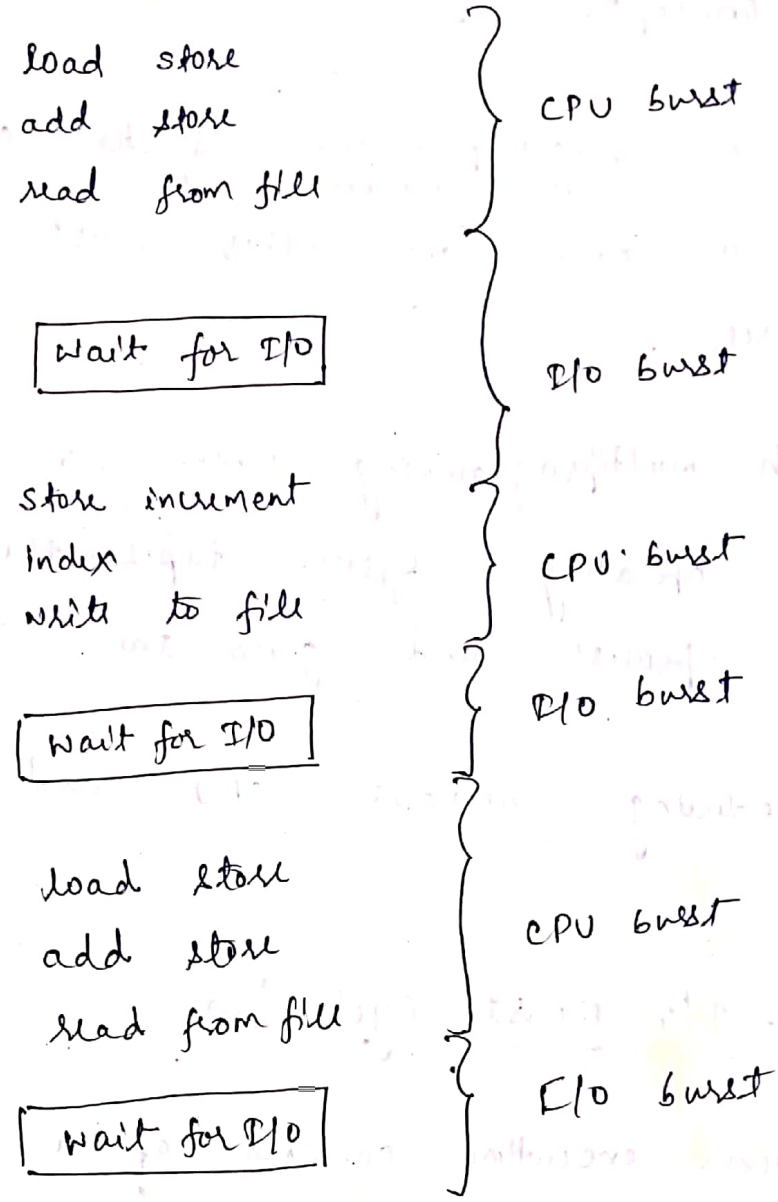
CPU Scheduling

Basic concepts :-

- In a single processor (process) system only one process can run at a time, any others must wait until the CPU is free.
- With multiprogramming when one process has to wait the operating system takes the CPU away from that process and gives the CPU to another process.
- scheduling increases CPU and resources utilization.

CPU - I/O Burst Cycle :→

- Process execution consists of a cycle of CPU execution and I/O wait.
- Process execution begins with a CPU burst that is followed by an I/O burst.
- Final CPU burst ends with a system request to terminate execution.



CPU Scheduler

Short Term Scheduler :-

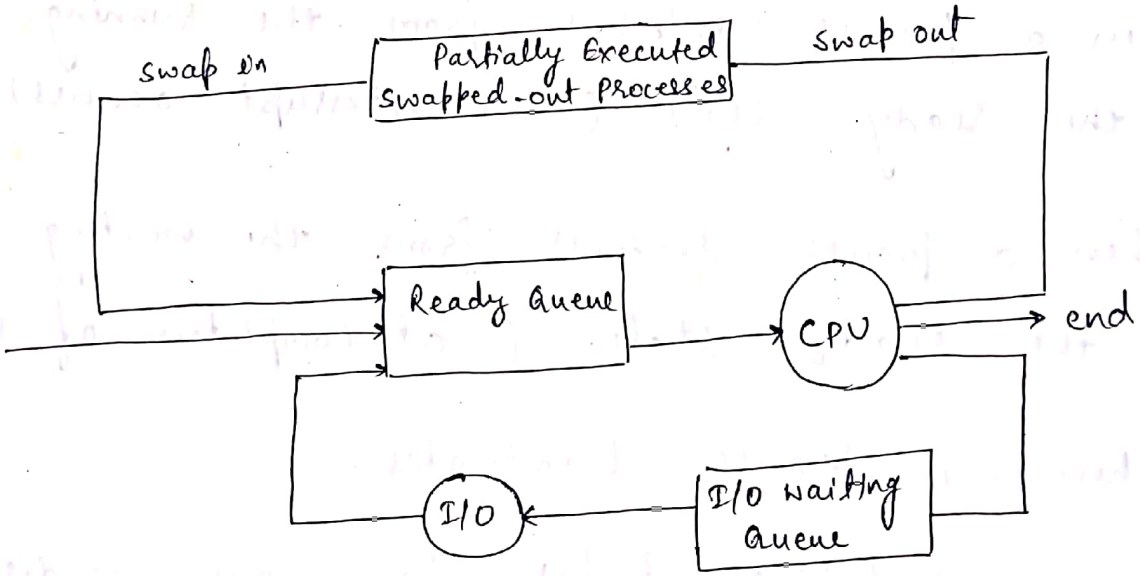
- This scheduler selects a process from the processes in memory that are ready to execute and allocates the CPU to that process.

Long term scheduler :→

- Long term scheduler control the degree of multi:- processing.
- ~~Decides how many processes and which process executions will be deferred.~~
- Decides how many processes and which process will be brought into the main memory for execution and which process executions will be deferred.

Medium term scheduler :→

- Medium term scheduler can be advantageous to remove processes from memory and thus reduce the degree of multiprogramming.



Medium term scheduling to the queuing diagram

- Later the process can be reintroduced into memory, and its execution can be continued where it left off.
- This scheme is called swapping. The process is swapped out and is later swapped in by the medium term scheduler.

Preemptive scheduling

CPU scheduling decisions may take place under the following four circumstances.

(i) When a process switches from the running state to the waiting state (I/O request)

(ii) When a process switches from the running state to the ready state (an interrupt occurs)

(iii) When a process switches from the waiting state to the ready state (at completion of I/O).

(iv) When a process terminates.

- When scheduling takes place only under circumstances 1 and 4, it is nonpreemptive or cooperative.
- Otherwise it is preemptive.

Scheduling Criteria

(i) CPU utilization :→

- We want to keep the CPU as busy as possible.
- CPU utilization can range from 0 to 100 percent.

(ii) Throughput :→

- One measure of work is the number of processes that are completed per time unit, called throughput.
- For long processes, this rate may be one process per hour.
- For short transactions, it may be 10 processes per second.

(iii) Turnaround time :→

- The interval from the time of submission of a process to the time of completion is the turn around time.
- Turn around time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.

(iv) Waiting time :→ • The CPU algorithm affects only the amount of time that a process

spends waiting in the ready queue.

- Waiting time is the sum of the periods spent waiting in the ready queue.

(V) Response time :→

- In an interactive system, turnaround time may not be the best criterion.
- A process can produce some output fairly early and can continue computing new results while previous results are being output to the user.
- Another measure is the time from the submission of a request until the first response is produced.
- This is called response time, is the time it takes to start responding, not the time it takes to output the response.

Dispatcher

- Dispatcher gives control of the CPU to the process selected by the short term scheduler.
- Time it takes for the dispatcher to stop one process and start another running is called as dispatch latency.

Scheduling Algorithms

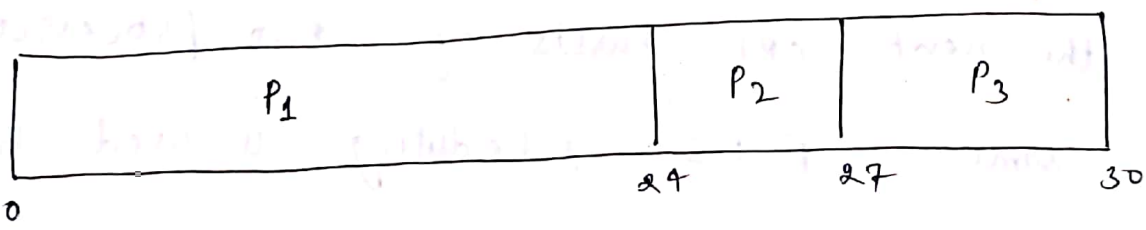
1) First come, First served scheduling :→

- With this scheme, the process that requests the CPU first is allocated the CPU first.
- When a process enters the ready queue, its PCB is linked onto the tail of the queue.
- When the CPU is free, it is allocated to the process at the head of the queue.
- The running process is then removed from the queue.

Q.

Process	Burst Time (milliseconds)
P_1	24
P_2	03
P_3	03

arriving order is P_1, P_2, P_3. And processes arrive at time 0.



Gantt chart

waiting time of process $P_1 = 0$

" " " " $P_2 = 24$

$P_3 = 27$

average waiting time = $(0+24+27)/3 = 17$ milliseconds.

- The average waiting time under an FCFS is generally not minimal.
- There is a convoy effect as all the other processes wait for the one big process to get off the CPU.
- This effect results in lower CPU and device utilization.
- The FCFS scheduling algorithm is nonpreemptive.

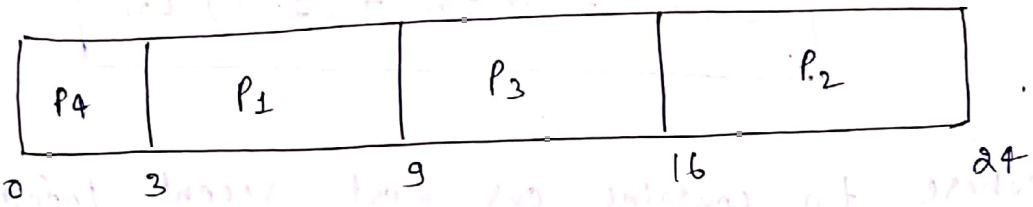
(ii) Shortest - Job - First scheduling :→

- This algorithm associates with each process the length of the process's next CPU burst.
- When 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 the same, FCFS scheduling is used to break the tie.

- Scheduling depends on the length of the next CPU burst of a process, rather than its total length.

Q:

Process	Burst time
P_1	6
P_2	8
P_3	7
P_4	3



Waiting time of $P_1 = 3$

" " " $P_3 = 9$

" " " $P_2 = 16$

average waiting time = $(3+9+16)/4 = 7$ millisec

- moving a short process before a long one decreases average waiting time.
- The real difficulty with the SJF algo. is knowing the length of the next CPU request.
- A prediction formula may be used to predict the amount of time for which CPU may be required by a process.

- Let t_n be the length of nth CPU burst and let τ_{n+1} be our predicted value for the next CPU burst.

- Defining a parameter α that controls the relative weight of recent and past history in our prediction such that $0 \leq \alpha \leq 1$ then a formula called exponential average is defined as -

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$$

where t_n contains our most recent information and τ_n stores the past history on prediction

- If $\alpha = 0$, then $\tau_{n+1} = \tau_n$ ⇒ recent history has no effect (current conditions are assumed to be transient)

- If $\alpha = 1$, then $\tau_{n+1} = t_n$ ⇒ Only the most recent CPU burst matters

- If $\alpha = 1/2$, then recent and past history are equally weighted.

- The SJF algorithm can be either preemptive or non preemptive.

- When a new process arrives at the ready queue while a previous process process is still executing. The next CPU burst of the newly arrived process may be shorter than what is left of the currently executing process.

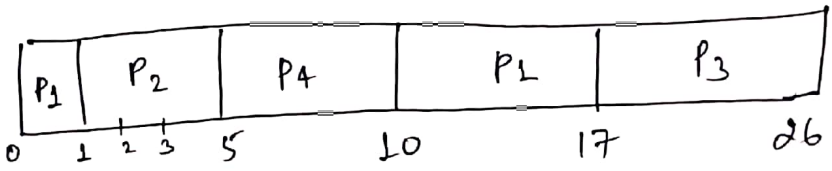
- A preemptive SJF algo. will preempt the currently executing process. It is called as <u>shortest remaining time first scheduling</u>.

- Whereas a nonpreemptive SJF algo. will allow the currently running process to finish its CPU burst.

Q:

Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

<u>Preemptive SJF</u>.



Waiting time for process P_1 = 0 + ((0-1) = 9 ms

" " " " P_2 = 0

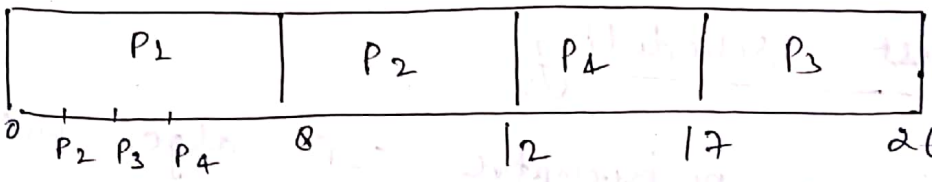
" " " " P_3 = 17 - 2 = 15 ms

" " " " P_4 = 5 - 3 = 2 ms

Average waiting time = $\frac{9+0+15+2}{4}$ ms

= 6.5 ms

Non preemptive SJF :-



waiting time for process P_1 = 0 ms

" " " " P_2 = 8 - 1 = 7 ms

" " " " P_3 = 17 - 2 = 15 ms

" " " " P_4 = 12 - 3 = 9 ms

Average waiting time = $\frac{0+7+15+9}{4}$ = ms

= $\frac{31}{4}$ = 7.75 ms.

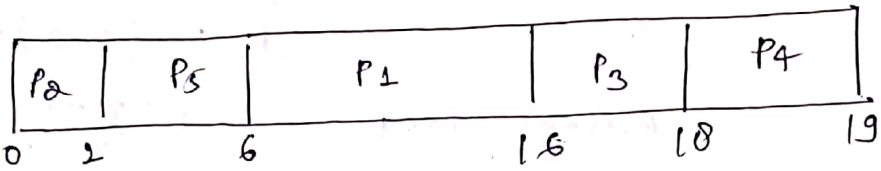
(iii) Priority Scheduling :→

- 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 algo. where the priority (p) is the inverse of the (predicted) next CPU burst.
- The larger the CPU burst, the lower the priority and vice versa.
- Priorities can be defined either internally or externally.
- Internally defined priorities use some measurable quantity to compute the priority.
- For eg. time limits, memory requirements, the no. of open files, ratio of average I/O burst to average CPU burst.
- External priorities are set by criteria outside the operating system such as the importance of the process, the type and amount of funds being paid for computer use, the department sponsoring the work.

- Priority scheduling can be either preemptive or non preemptive.
- A preemptive priority scheduling algo will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process.
- A non preemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.
- A major problem with priority scheduling algorithms is indefinite blocking or starvation.
- A priority scheduling algorithm can leave some low priority processes waiting indefinitely.
- Solution to this problem is aging.
- Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time.
- For eg. If priority of a waiting range from 127 (low) to 0 (high), we could increase the priority of a waiting process by 1 every 15 minutes.
- It would take no more than 32 hours for a priority 127 process to age to a priority 0 process.

Q:

Process	Burst time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2



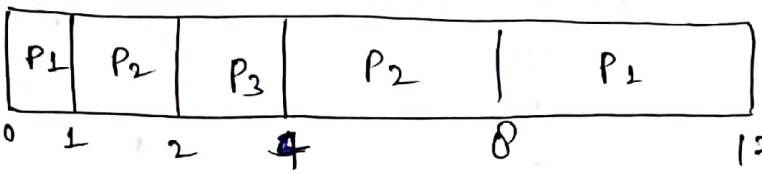
waiting time for process P_1 = 6 ms
P_2 = 0 "
P_3 = 16 "
P_4 = 18 "
P_5 = 1 "

Avg. waiting time = $\frac{6+0+16+18+1}{5}$ = 8.2 ms

Preemptive Priority Scheduling

Process	Burst time	Priority	Arrival Time
P_1	10	3	0
P_2	5	2	1
P_3	2	1	2

- Initially process P_1 is in the system.
- After 1 ms at $t=1$ new process P_2 with priority 2 arrives, its priority is higher than the priority of P_1. so P_1 is preempted.
- Again at $t=2$, P_3 (priority = 1) arrives so P_2 is preempted.
- P_3 is executed until its execution is completed since P_3 has got highest priority.



waiting time for process P_1 = 0 + (0-1) ms = 7 ms

P_2 = (4-2) ms = 2 ms

P_3 = (2-2) ms = 0 ms

Average w.t. = $\frac{7+2+0}{3}$ ms = 3 ms.

~~Round Robing~~ Round Robin Scheduling :→

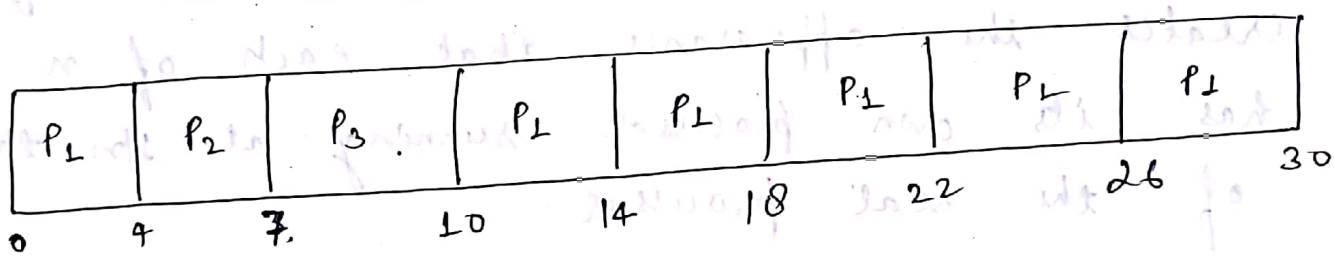
- Designed especially for time sharing systems.
- similar to FCFS scheduling, but preemption is added to switch between processes.
- A small unit of time, called a time quantum or time slice is defined. and ranges generally from 10 to 100 milliseconds.
- The ready queue is treated as a circular queue and scheduler allocate CPU to each process for a time interval of up to 1 time quantum.
- Ready queue will be FIFO queue and new processes are added to the tail of the ready queue.

- If a process has CPU burst of less than 1 time quantum, the process itself will release the CPU.
- otherwise time will go off and will cause an interrupt to the O.S..
- A context switch will be executed and the process will be put at the tail of the ready queue.
- Preemptive scheduling algo. and implement context switching.

Q:

Process	Burst Time
P_1	24
P_2	3
P_3	3

Time quantum = 4ms



waiting time for process $P_1 = 3+3 = 6$

$P_2 = 4$

$P_3 = 7$

avg. waiting time = $\frac{6+4+7}{3} = \frac{17}{3} = 5.66$ ms.

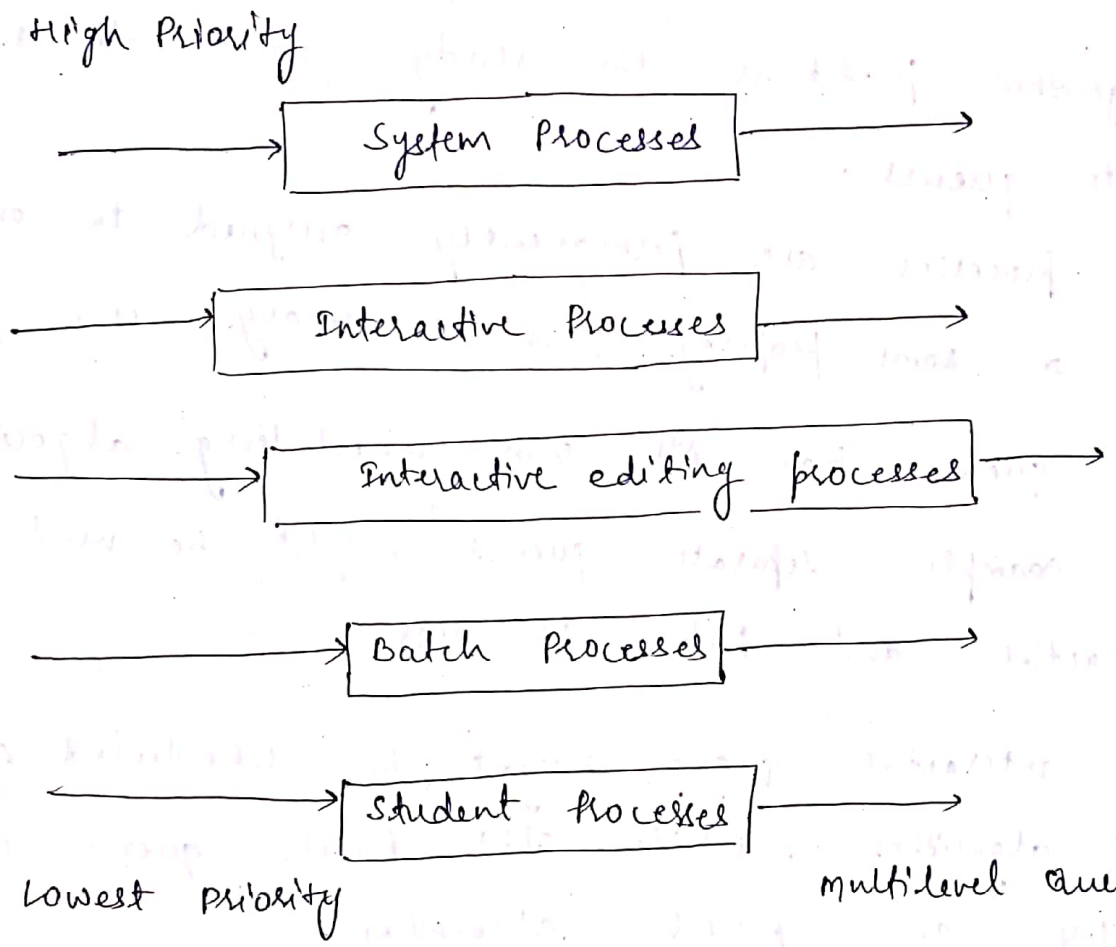
- If there are n processes in the ready queue and the time quantum is q, then each process gets 1/n of the CPU time in chunks of at most q time units.
- Each process must wait no longer than (n-1)×q time units until its next time quantum.
- The performance of the RR algo. depends heavily on the size of the time quantum
- If the time quantum is extremly large, the RR policy is the same as the FCFS.
- If the time quantum is extremely small the RR approach is called processor sharing and creates the appearance that each of n process has its own processor running at 1/n the speed of the real processor.
- Turnaround time also depends on the size of the time quantum.
- The average turnaround time increases for a smaller time quantum since more context switches are required
- If the time quantum is too large, RR degenerates to FCFS policy.

- If the time quantum of three processes (10 unit time) is 1 then avg. turnaround time is = 29.
- If time quantum is 10, the avg. turnaround time drops to 20.

Multilevel Queue Scheduling

- Algorithm partitions the ready queue into several separate queues.
- The processes are permanently assigned to one queue based on some property such as memory, size, priority etc.
- Each queue has its own scheduling algorithm.
- For example separate queues might be used for interactive and batch processes.
- The interactive queue might be scheduled by an RR algorithm, while the batch queue is scheduled by an FCFS algorithm.
- Scheduling among the queues is commonly implemented as fixed priority preemptive scheduling.
- Each queue has absolute priority over lower priority queues.

- For example no process in the batch queue 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.



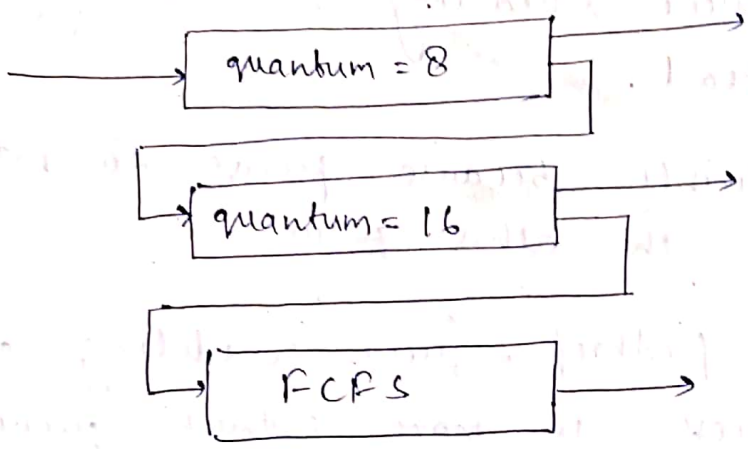
Multilevel Queue Scheduling

- Another possibility is to time slice among the queues.
- Each queue gets a certain portion of the CPU time which it can then schedule among its various processes.

Multilevel Feedback Queue Scheduling

- In multilevel queue scheduling has advantage of low scheduling overhead.
- But it is inflexible because process do not move from one queue to the other queue.
- The multilevel feedback-queue scheduling algorithm allows a process to move between queues.
- The idea is to separate 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.
- A process that wait too long in a lower priority queue may be moved to a higher-priority queue.
- This form of aging prevents starvation.
- For example consider a multilevel feedback-queue scheduler with three queues numbered from 0 to 2.
- A process entering the ready queue is put on queue 0 and process in queue 0 is given a time quantum of 8 ms.

- If it does not finish within this time, it is moved to the tail of queue 1.



Multilevel feedback queues

- If queue 0 is empty the process at the head of queue 1 is given a quantum of 16 ms.
- If it does not 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 and 1 are empty.
- This scheduling algorithm gives highest priority to any process with a CPU burst of 8 ms or less.
- Processes that need more than 8 but less than 24 ms are also served quickly, although with lower priority than shorter processes.

long processes automatically sink to queue 2 and are served in FCFS order with any CPU cycles left over from queues 0 and 1.

(26) A multilevel feedback-queue scheduler is defined by the following parameters.

- The number of queues
- The scheduling algorithm for each queue
- The method used to determine when to upgrade a process to a higher priority queue.
- The method used to determine when to demote a process to a lower priority queue.
- The method used to determine which queue a process will enter when that process needs service.

Multiprocessor Scheduling

- In one approach all scheduling decisions, I/O processing and other system activities handled by a single processor - the master server.
- The other processors execute only user code.
- Called as Asymmetric Multiprocessing and is simple because only one processor accesses the system data structures, reducing the need for data sharing.

- In symmetric multiprocessing (SMP), each processor is self scheduling.
- All processes may be in a common ready queue, or each processor may have its own private queue of ready processes.
- Scheduling proceeds by having the scheduler for each processor examine the ready queue and select a process to execute.
- Virtually all modern operating systems support SMP, including XP, 2000, Solaris, Linux etc.
- If multiple processors trying to access and update a common data structure, the scheduler must be programmed carefully.
- We must ensure that two processors do not choose the same process and that processes are not lost from the queue.

Issues concerning SMP System :→

(1) Processor Affinity :→

- In cache memory, the data most recently accessed by the process populates the cache for the processor.

- As a result successive memory accesses by the process are often satisfied in cache memory.
- If the process migrates to another processor, the contents of cache memory must be invalidated for the processor being migrated from, and the cache for the processor being migrated to must be repopulated.
- That is why most SMP systems try to avoid migration of processes.
- And SMP system try to keep a process running on the same processor.
- This is known as processor affinity meaning that a process has an affinity for the processor on which it is currently running.
- When an operating system has a policy of attempting to keep a process running on the same processor but not guaranteed and process can migrate between processors, this situation is called as <u>soft affinity</u>.
- Some systems such as Linux also provide system calls that support hard affinity, thereby allowing a process to specify that it is not to migrate to other processors.

(11) Load Balancing :→

- It is important to keep the workload balanced among all processors to fully utilize the benefits of having more than one processor.
- There are two general approaches to load balancing (a) push migration (b) pull migration.
- With push migration, a specific task periodically checks the load on each processor.
- If it finds an imbalance - evenly distributes the load by moving (or pushing) processes from overloaded to idle or less busy processors.
- Pull migration occurs when an idle processor pulls a waiting task from a busy processor.
- Push and pull migration need not be mutually exclusive and can be implemented in parallel.
- Processor affinity concept can not be implemented here.

Dead Locks

- In a multiprogramming environment, several processes may complete for a finite number of resources.
- A process requests resources and if the resources are not available at that time, the process waits.
- If the resources it has requested are held by other waiting process, then this situation is called as dead lock.

System Model

- A system consists of a finite number of resources to be distributed among a number of completing processes.
- The resources are partitioned into several types, each consisting of some number of identical instances.
- Memory space, CPU cycles, files and I/O devices (printers and DVD drives etc) are examples of resource types.
- If a system has two CPUs then the resource type CPU has two instances.
- Similarly the resource type printer may have five instances

- If a process requests an instance of a resource type the allocation of any instance of the type will satisfy the request.
- A process must request a resource before using it and must release the resource after using it.

A process may utilize a resource in only the following sequence.

(I) Request :- If the request can not be granted immediately then the requesting process must wait until it can acquire the resource.

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

(III) Release :- The process releases the resources.

Dead lock Characterization

Necessary conditions for dead lock :→

(i) Mutual Exclusion :→

- At least one resource must be held in a non sharable mode.
- Mean only one process at a time can use the resource.

- If another process requests that resource, the requesting process must be delayed until the resource has been released.

(ii) Hold and wait :→

- A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.

(iii) NO Preemption :→

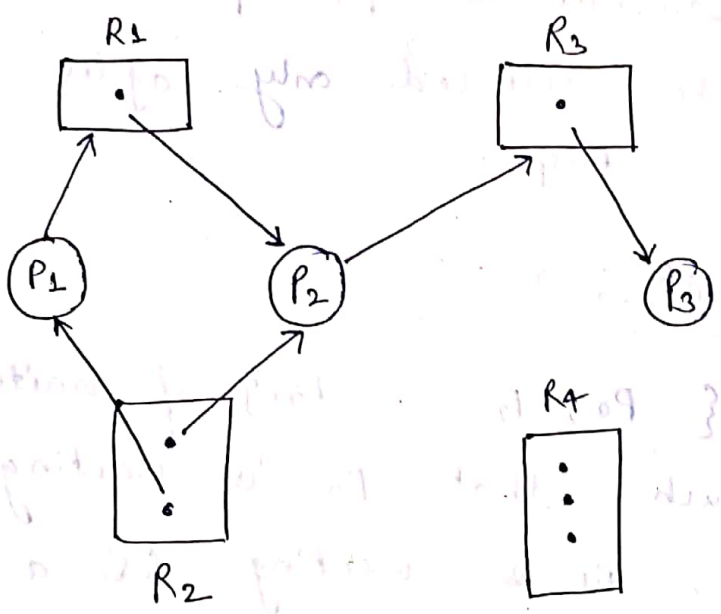
- Resources can not be preempted that means a resource can be released only after process has completed its task.

(iv) Circular wait :→

- A set $\{P_0, P_2, \ldots, P_n\}$ 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 $P_2, \ldots, P_{n-1}$ is waiting for resource held by P_0.

Resource Allocation Graph

- Deadlocks can be described in terms of a directed graph called a system resource - allocation graph.
- This graph consists of a set of vertices V and set of edges E.
- The set of vertices V is partitioned into two different types of node, set of processes and set of resources.



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

- A directed edge $P_i \rightarrow R_j$ is called a request edge and $R_j \rightarrow P_i$ is called an assignment edge.
- When process requests a resource, a request edge is inserted in the graph and when this request is fulfilled, the request edge is instantaneously transformed to an assignment edge.
- When process releases the resources, the assignment edge is deleted.

Figure depicts the following situation.

- The sets P, R and E:

$P = \{P_1, P_2, P_3\}$; $R = \{R_1, R_2, R_3, R_4\}$

$E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$

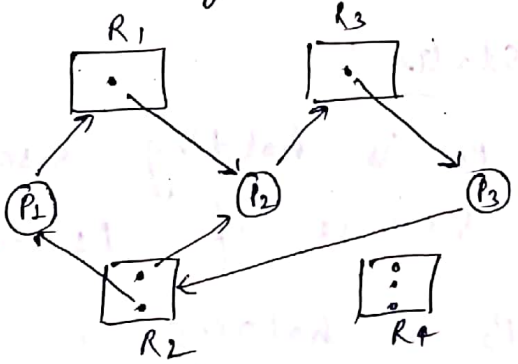
- Resource Instances:
  - One instance of resource type R_1
  - Two " " " " R_2
  - One " " " " R_3
  - Three " " " " R_4
- Process States:
  - Process P_1 is holding resource of R_2 and is waiting for resource of R_1.
  - Process P_2 is holding an instance of R_1 and an instance of R_2 and waiting for an instance of R_3

- Process P_3 is holding an instance of R_3.

By the definition, if 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 occured.
- If cycle involves only a set of resource types, each of which has only a single instance, each process involved in the cycle is deadlocked.
- If each resource type has several instances, then a cycle does not necessarily imply that a dead lock has occured.

In figure, suppose P_3 requests an instance of resource type R_2. since no resource instance is available, a request edge $P_3 \rightarrow R_2$ is added to the graph.



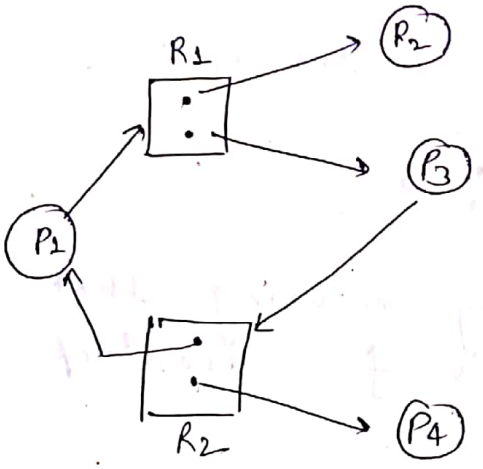
∴ Two minimal cycles exist in the system.

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

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

Processes P_1, P_2 and P_3 are deadlocked.

- P_2 is waiting for R_3 which is held by P_3.
- P_3 is waiting for either P_1 or P_2 to release resource R_2.
- P_1 is waiting for process P_2 to release resource R_1.



In above graph there is cycle but there is no dead lock.

$P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

Because P_4 may release its instance of resource type R_2. And that resource can then be allocated to P_3 breaking the cycle.

Methods for Handling Deadlocks

We can deal with the deadlock problem in one of three ways

(i) We can use a protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlock state.

(ii) We can allow the system to enter a deadlock state, detect it and recover.

(iii) We can ignore the problem altogether and pretend that deadlocks never occur in the system.

Deadlock ~~Avoidance~~ Prevention :→

- In this scheme, by ensuring that at least one of the four conditions for deadlock can not hold

(a) Mutual Exclusion :—

- The mutual exclusion condition must hold for non sharable resource.
- For example a printer can not be simultaneously shared by several processes.
- sharable resources do not require mutually exclusive access and thus cannot be involved in a deadlock.
- For example read only files can be open by several processes at the same time.

(b) No preemption :→

- To ensure that this condition should not happen we follow following steps -
  - If a process requests some resources, we check whether they are allocated to some other process that is waiting for additional resources.
- If so, we preempt the desired resources from the waiting process and allocate them to the requesting process.
- If the resources are neither available nor held by a waiting process, the requesting process must wait
- while it is waiting, some of its resources may be preempted but only if another process requests them.

(c) Hold and wait :→

- To ensure that the hold and wait condition never occur in the system, we must guarantee that whenever a process requests a resource, it does not hold any other resources.
- one protocol that can be used requires each process to request and be allocated all its resources before it begins execution.
- Second protocol allows a process to request resources only when it has none.

- A process may request some resources and use them.
- Before it can request any additional resources, it must release all the resources that it is currently allocated.
- Disadvantages is starvation and low utilization of resources.

(d) Circular wait :→

- To ensure that this condition never holds is to impose a total ordering of all resource types and to require that each process requests resources in an increasing order of enumeration.
- Let $R = \{R_1, R_2, R_3 \ldots R_m\}$ be the set of resource types.
- We assign to each resource type a unique integer number.
- Each process can request resources only in an increasing order of enumeration. Means a process can initially request any number of instances of a resource type R_i.
- After that the process can request instances of resource type R_j if and only if $F(R_j) > F(R_i)$
- If several instances of the same resource type are needed, a single ~~proces~~ request for all of them must be issued.

- Let the set of processes involved in the circular wait be $\{P_0, P_1, \ldots P_n\}$, where P_i is waiting for a resource R_i, which is held by process P_{i+1}.
- Since process P_{i+1} is holding resource R_i while requesting resource R_{i+1} we must have $F(R_i) < F(R_{i+1})$ for all i.
- But this condition means that $F(R_0) < F(R_1) < \ldots F(R_n) < F(R_0)$.
- By transitivity $F(R_0) < F(R_0)$, which is impossible.
- Therefore there can not be circular wait.

~~Deadlock Avoidance~~

* We define a one to one function $F: R \to N$, where N is the set of natural no.

For eg

F(tape drive) = 1

F(disk drive) = 5

F(printer) = 12

Deadlock Avoidance :→

- Requires additional information about how resources are to be requested.
- For ex. in a system with one printer and one tape drive the system might need to know that process P will request first the tape drive and then the printer.

- Whereas process Q will request first printer and then the tape drive.
- With this releasing resource information and acquired resources information system can avoid deadlock.
- The simplest and useful algorithm requires that each process declare the maximum number of resources of each type that it may need.
- A deadlock avoidance algorithm dynamically examines the resource allocation state to ensure that a circular wait condition can never exist.
- The resource allocation state is defined by the number of available and allocated resources and the maximum demands of the processes.

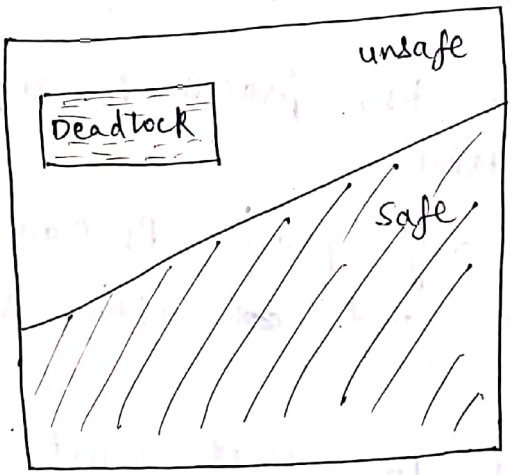
There are two deadlock avoidance algorithms

(i) Safe state :→

- A state is safe if the system can allocate resource to each process (max.) in some order and still avoid a deadlock.
- Alternatively a system is on a safe state only if there exists a safe sequence.
- A sequence of processes $<P_1, P_2, \cdots P_n>$ is a safe sequence for the current allocation state if, for each P_i, the resource requests that P_i can still make can be satisfied by the currently available

resources plus the resources held by all P_j with $j < i$.

- In this situation if the resources that P_i needs are not immediately available then P_i can wait until all P_j have finished.
- When they have finished, P_i can obtain all of its needed resources, complete its task and return its allocated resources.
- When P_i terminates, P_{i+1} can obtain its needed resources and so on.
- If no such sequence exists then the system state is said to be unsafe.
- A safe state is not a deadlock state.
- A deadlocked state is an unsafe state.
- Not all unsafe states are deadlocks.



For example consider a system with 12 magnetic tape drives

	Max Needs	Need	Current allocation
P_0	10	5	5
P_1	4	2	2
P_2	9	7	2

At time t_0, the system is in a safe state. The sequence $<P_1, P_0, P_2>$ satisfies the safety condition.

- Process P_1 can immediately be allocated all its tape drives and return them (the system will have 5 available tape drives now).
- Then process P_0 can get all its tape drives and return them (now 10 are available)
- Finally process P_2 can get all its tape drives and return them (Now 12 tape are available).

A system can go from a safe state to an unsafe state.

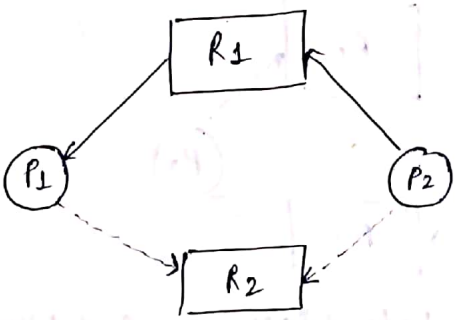
- Suppose at time t_1, process P_2 requests and is allocated one more tape drive.
- At this time only process P_1 can be allocated all its tapes drives and after return total available will be 4.
- Therefore process P_0 must wait because only 4 is available and requirement is 05.

Idea is simple whenever a process requests a resource that is currently available, the system must decide whether the resource can be allocated immediately or process must wait.

- Disadvantage is that requested resource may be available but process have to wait. i.e lower resource utilization.

(ii) Resource - Allocation Graph Algorithm

- Similar to resource allocation graph
- New type of edge is introduced called a claim edge. A claim edge $P_i \rightarrow R_j$ indicates that process P_i may request resource R_j at some time in future.
- It is represented in the graph by a dashed line.
- When process P_i requests resource R_j, the claim edge $P_i \rightarrow R_j$ is converted to a request edge.



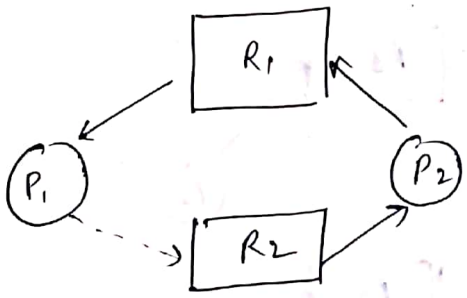
Resource allocation graph for deadlock avoidance

- When a resource R_j is released by P_i the assignment edge $R_j \rightarrow P_i$ is reconverted to a claim edge $P_i \rightarrow R_j$.

- Before process P_i starts executing, all its claim edges must already appear in the resource-allocation graph.
- We can relax this condition by allowing a claim edge $P_i \rightarrow R_j$ to be added to the graph only if all the edges associated with process P_i are claim edges.
- If P_i requests resource R_j then it can only be granted if there is no cycle in the graph on converting request edge into assignment edge.
- If no cycle exists then the allocation of the resource will leave the system in a safe state. Otherwise system will be in unsafe state.

For example - According to previous figure suppose that P_2 request R_2.

- Although R_2 is currently free, we cannot allocate it to it since this action will create a cycle in the graph.



An unsafe state in a resource allocation graph

Now if P_1 request R_2 and P_2 request R_1 then a deadlock will occur. Hence system is in unsafe state.

(iii) Banker's Algorithm

- The resource allocation graph algorithm is not applicable to a resource allocation system with multiple instances of each resource type.
- Banker's algo is less effective than the resource allocation scheme but can be applicable to multiple instances of resource type.
- Name was chosen because bank ensures that it never allocated its available cash in such a way that it could no longer satisfy the needs of all its customers.
- When a new process enters the system it must declare maximum number of instances of each resource type that it may need.
- When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state or not.
- Let n be the number of processes in the system and m be the number of resource types. We need the following data structures –

(a) **Available** :– A vector of length m indicates the number of available resources of each type. If Available [j] equals k, there are k instances of resource type R_j available.

(b) Max :- An n×m matrix defines the maximum demand of each process.

• If Max[i][j] equals k, then process Pi may request at most k instances of resource type Rj.

(c) Allocation :- An n×m matrix defines the number of resources of each type currently allocated to each process.

• If allocation [i][j] equals k, then process Pi is currently allocated k instances of resource type Rj.

(d) Need :→ An n×m matrix indicates the remaining resource need of each process.

• If Need [i][j] equals k, then process Pi may need k more instances of resource type Rj to complete its task.

• Note that Need [i][j] equals max[i][j] - Allocation [i][j]

(a) Safety Algorithm :-

• This algorithm is used to find out whether a system is in a safe state or not.

(1) Let work and Finish be vector of length m and n, respectively.

- Initialize work = Available and Finish[i] = false for i=0, 1, ..., n-1.

(2) Find an i such that both
(i) Finish[i] == false
(ii) Needi ≤ work

If no such i exists, go to step 4.

3. Work = Work + Allocation$_i$

Finish[i] = true

goto step 2

4. If Finish[i] == true for all i, then the system is in a safe state.

Algorithm may require an order of $m \times n^2$ operations to determine whether a state is safe.

(b) Resource Request algorithm :—

- Algorithm which determines if requests can be safely granted.
- Let Request$_i$ be the request vector for process P_i. If Request$_i$[j] == k, then process P_i wants k instances of resource type R_j.
- When a request for resources is made by process P_i the following actions are taken -

1. If Request$_i$ ≤ Need$_i$ go to step 2. otherwise raise an error condition, since the process has exceeded its maximum claim.

2. If Request$_i$ ≤ Available go to step 3. Otherwise P_i must wait, since the resources are not available.

3. Have the system pretend to have allocated the requested resources to process P_i by modifying the state as

follows -

Available = Available − Request$_i$;

Allocation$_i$ = Allocation$_i$ + Request$_i$;

Need$_i$ = Need$_i$ − Request$_i$;

If the resulting resource-allocation state is safe the transaction is completed and process P_i is allocated its resources. However if the new state is unsafe then P_i must wait for request$_i$ and the old resource allocation state is restored.

Example of Banker's Algorithm

Consider there are five processes P_0 --- P_4 and three resource type A, B and C. A has 10 instance, B has 5 instances and C has 7 instances.

Suppose that at time T_0, the following snapshot of the system has been taken -

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	3	3	2
P_1	2	0	0	3	2	2			
P_2	3	0	2	9	0	2			
P_3	2	1	1	2	2	2			
P_4	0	0	2	4	3	3			

The content of the matrix Need is defined to be Max - Allocation and is as follows -

	Need A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

- System is currently in a safe state. The sequence $\langle P_1, P_3, P_4, P_2, P_0 \rangle$ satisfies the safety criteria.
- Suppose that now process P_1 requests one additional instance of resource type A and two instances of resource type C, so Request$_1$ = (1,0,2).
- To decide whether this request can be immediately granted, we first check that Request$_1 \leq$ Available - That is that $(1,0,2) \leq (3,3,2)$ which is true.
- We then pretend that this request has been fulfilled and we arrive at the following new state:

	Allocation A B C	Need A B C	Available A B C
			2 3 0
P_0	0 1 0	7 4 3	
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

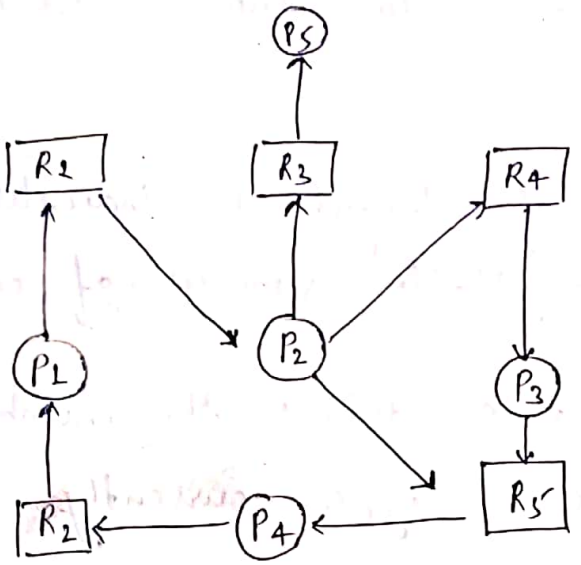
- We must determine whether this new system state is safe.
- To do so, we execute our safety algo. and find that the sequence $< P_1, P_3, P_4, P_0, P_2 >$ satisfies the safety requirement.
- So we can immediately grant the request of process P_1.
- Also in this state a request for (3, 3, 0) by P_4 cannot be granted, since the resources are not available.
- Furthermore, a request for (0, 2, 0) by P_0 cannot be granted, even though the resources are available, since the resulting state is unsafe.

Deadlock Detection

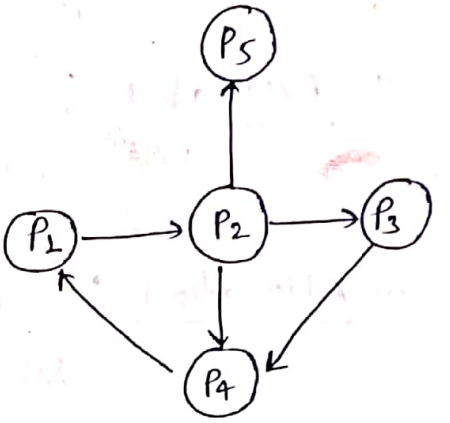
- If a system does not employ either a deadlock prevention or a deadlock avoidance algorithm then a deadlock situation may occur.
- In this environment the system must provide.
- An algorithm that examines the state of the system to determine whether a deadlock has occurred.
- An algorithm to recover from the deadlock

(a) Single instance of Each Resource Type :→

- If all resources have only a single instance then we use resource allocation graph to determine a deadlock and it is called as wait-for graph.



Resource allocation graph



Wait for graph

- In wait for graph, edge from Pi to Pj implies that process Pi is waiting for process Pj to release a resource
- An edge Pi → Pj exists in a wait for graph if and only if the corresponding resource allocation graph contain two edges Pi → Rq and Rq → Pj for some resource Rq.
- Deadlock exists if and only if the wait for graph Contain cycle.
- System needs to maintain the wait for graph and periodically invoke an algo. that searches for a cycle.
- To detect cycle in graph requires an order of n^2 operations

n = no. of vertices

(b) Several instances of a Resource type :→

- The wait for graph scheme is not applicable to a resource allocation system with multiple instances of each resource
- This algorithm employs several time varying data structures that are similar to those used in the banker's algorithm.

(a) Available :- A vector of length m indicates the number of available resources of each type.

(b) Allocation :- An n×m matrix defines the number of resources of each type currently allocated to each process.

(c) Request :- An n×m matrix indicates the current request of each process.

- If request [i][j] equals k, then process Pi is requesting k more instances of resource type Rj.

1. Let work and finish be vectors of length m and n respectively. Initialize work = Available. For i=0, 1, n-1, if allocation i ≠ 0, then
2. Find an index i such that both
   Finish[i] == false
   Request i ≤ work

   If no such i exists go to step 4.

3. Work = Work + Allocation$_i$
   Finish[i] = true
   Goto step 2

4. If finish[i] == false, for some i, $0 \leq i < n$, then the system is in a deadlocked state. Moreover if finish[i] == false, then process P_i is deadlocked.

This algorithm requires an order of $m \times n^2$ operations to detect whether the system is in a deadlocked state.

Example — There are 4 processes and three resource types A, B and C.

- A has 7 instances, B has 2 instances, C has 6 instances.

At time To, we have the following resource - allocation state:

	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	0	0	0	0	0	0
P_1	2	0	0	2	0	2			
P_2	3	0	3	0	0	0			
P_3	2	1	1	1	0	0			
P_4	0	0	2	0	0	2			

The system is not in a deadlocked state. If we execute our algo. we will find that the $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ sequence results in finish[i] == true for all i.

Suppose now that process P_2 makes one additional request for an instance of type C.

	Request		
	A	B	C
P_0	0	0	0
P_1	2	0	2
P_2	0	0	1
P_3	1	0	0
P_4	0	0	2

We claim that the system is now deadlocked. Resources held by process P_0, the number of available resources is not sufficient to fulfill the requests of the other processes. Thus a deadlock exists, consisting of processes P_1, P_2, P_3 and P_4.

(c) Detection - Algorithm Usage

- Usage of algorithm depends on two factors:
  - How often is a deadlock likely to occur?
  - How many processes will be affected by deadlock when it happens?
- If deadlock occur frequently then the detection algorithm should be invoked frequently.
- Deadlocks occur only when some process makes a request that can not be granted immediately.
- Disadvantage is that if it is invoked for every resource request this will incur a considerable overhead in computation time.

Recovery from Deadlock

There are two options for breaking a deadlock.

- One is simply to abort one or more processes to break the circular wait.
- The other is to prempt some resources from one or more of the deadlocked processes.

(a) **Process Termination** –

- To eliminate deadlocks by aborting a process, we use one of two methods.

(i) **Abort all deadlocked processes** :–

- This method clearly will break the deadlock cycle but at great expense.
- The deadlocked processes may have computed for a long time. and after abortion of the process we have to recomputed later.

(ii) **Abort one process at a time until the deadlock cycle is eliminated** :→

- This method incurs considerable overhead, since after each process is aborted, a deadlock detection algo. must be invoked to determine whether processes are still deadlocked.

- We should abort those processes whose termination will incur the minimum cost.

Many factors may affect which process is chosen including -

(I) What is the priority of the process is

(II) How long the process has computed and how much longer the process will compute before completing its designated task.

(III) How many and what type of resources the process has used means whether the resources are simple to preempt.

(IV) How many more resources the process needs in order to complete

(V) How many processes will need to be terminated

(VI) Whether the process is interactive or batch.

(b) Resource Preemption

If preemption is required to deal with deadlocks then three issues need to be addressed.

(i) Selecting a victim :→

- We must determine the order of preemption to minimize cost.

Includes no. of ~~process~~ resources a deadlocked process is holding and amount of time the process has thus far consumed during its execution.

(ii) Roll back :—

- If we preempt a resource from a process, we must roll back the process to some safe state and restart it from that state.
- It is difficult to determine what a safe state is the simplest solution is a total roll back! Abort the process and then restart it.
- This method requires the system to keep more information about the state of all running processes.

(iii) Starvation :→

- How can we guarantee that resources will not always be preempted from the same process.
- must ensure that a process can be picked as a victim only a small finite number of times.
- Common solution is to include the number of roll backs in the cost factor.