

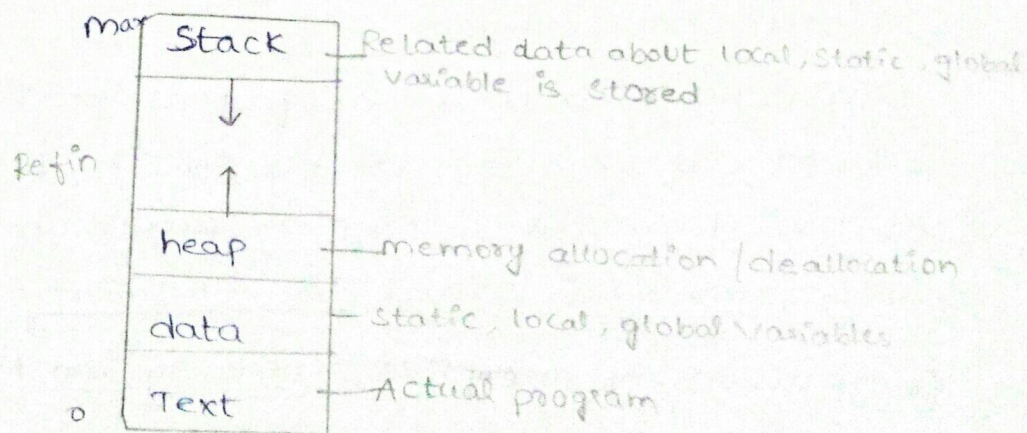
PROCESSES AND CPU Scheduling

process:- A process is defined as a program in execution
(or) A program under execution (or) A unit of work (or)
an instance of a running program.

Difference between program and process

Basic for comparison	Program	Process
Basic	program is a set of instruction	When a program is executed, it is known as process
Nature	passive	Active
Life span	longer	Limited
Required Resources	program is stored on disk in some file and does not require any other resources	process holds resources such as CPU, memory address, disk, I/O etc

process in memory:-



9/11/18

cpu scheduling:- CPU scheduling is a technique in operating system which handles the removal (or) selection of processes from ready queue.

* CPU scheduling decisions may take place under the following four conditions.

1. When a process switches from running state to waiting state.
2. When a process switches from running state to ready state.
3. When a process switches from waiting state to ready state.

4. When a process terminates

* When scheduling takes place under the conditions 1 and 4 scheduling scheme is "non-preemptive" (or) "co-operative" else it is called "preemptive"

* under non preemptive scheduling once the CPU has been allocated to a process the process keeps the CPU until it releases the CPU either terminating (or) by switching to the waiting state.

* preemptive scheduling ~~includes~~ cost associative with access to shared data.

Dispatcher:- The Dispatcher is the module that gives control of the CPU to the process selected by the short term scheduler. This function involves

1. Switching Context
2. Switching to user mode
3. Jumping to the proper location in the user program to restart that program.

Dispatcher should be as fast as possible since it is invoked during every process switch. The time it takes for the dispatcher to stop one process and start another process running is called "dispatch latency".

Comparison of preemptive scheduling and Non preemptive scheduling-

Definition	processor can be preempted to execute a different process in the middle of execution of any current process.	once processor starts to execute a process it must finish it before executing the other. it cannot be paused in middle.
Interrupt	process can be interrupted in between	process can not be interrupted till it terminates or switches to waiting state.
CPU utilization	cpu utilization is more compared to non preemptive scheduling	cpu utilization is less compared to non preemptive scheduling
Time	waiting time & response time is less	waiting time & response time is more
Starvation	If a high priority process frequently arrives in the ready queue	

Nonpreemptive Scheduling Algorithms:-

"Gantt chart:- A graphical representation of execution of a process flow upto the termination.

1. First come first serve scheduling Algorithm (FCFS)
2. Shortest Job first (SJF)
3. priority Scheduling Algorithm (PSA)
4. Round Robin Scheduling Algorithm (RRSA)

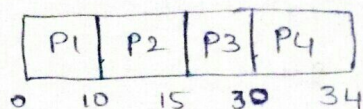
first come first serve scheduling Algorithm (FCFS):-

eg:-

Process	Bursttime
P1	10
P2	5
P3	15
P4	4

find the average waiting time and the turnaround time of the above process.

Gantt chart:-



Waiting time:- $0 + 10 + 15 + 30 = 55$

$$\text{Average waiting time} = \frac{55}{4} = 13.75$$

Turn around time:- $10 + 15 + 30 + 34 = 89$

$$\text{Average Turn around time} = 89/4 = 22.25$$

eg:- 2)

Process	Burst time
P1	5
P2	15
P3	2
P4	20
P5	7

3)

Process	Burst
P1	24
P2	3
P3	3

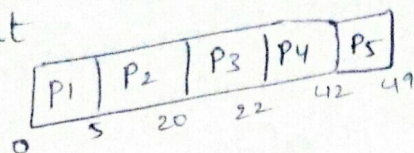
4)

Process	Bursttime
P1	10
P2	3
P3	4
P4	7
P5	-

5)

Process	Bursttime
P1	7
P2	6
P3	4
P4	1

2) Gantt chart



$$\text{Waiting time} = 0 + 5 + 20 + 22 + 42 = 89$$

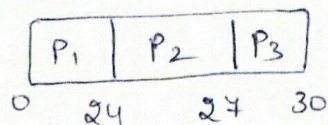
$$\text{Average waiting time} = 89/5 = 17.8$$

$$\text{Turn around time} = 5 + 20 + 22 + 42 + 49 = 138$$

Average turn around time

$$= \frac{138}{5} = 27.6$$

3) Gantt chart



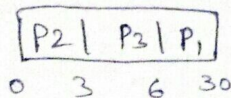
$$\text{Waiting time:- } 0 + 24 + 27 = 51$$

$$\text{Average waiting time:- } \frac{51}{3} = 17$$

$$\text{Turn around time:- } 24 + 27 + 30 = 81$$

$$\text{Avg Turn around time} = \frac{81}{3} = 27$$

3) In other way



$$\text{W.T} = 0 + 3 + 6 = 9$$

$$\text{A.W.T} = \frac{9}{3} = 3$$

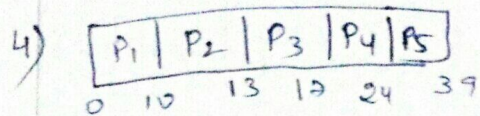
$$\text{Turn around time} = 3 + 6 + 30 = 39$$

$$\text{Avg Turn around time} = \frac{39}{3} = 13$$

Drawback:-

1. Problems with FCFS are non preemptive scheduling (whatever is waiting in queue that has to be picked by CPU no concept of priority).
2. Improper process scheduling (depending upon pattern we are getting different values).
3. Resource utilization in parallel is not possible that which leads to "convey effect" and hence it is proved that poor resource utilization.

Convey Effect:- In which whole operating system slows down due to some few process.

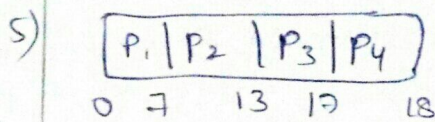


Waiting time = $0 + 10 + 13 + 17 + 24 = 64$

Average waiting time = $\frac{64}{5} = 12.8$

Turn around time = $10 + 13 + 17 + 24 + 39 = 103$

Average Turn around time = $\frac{103}{5} = 20.6$



Waiting time = $0 + 7 + 13 + 17 = 37$

Average waiting time = $\frac{37}{4} = 9.25$

Turn around time = $7 + 13 + 17 + 18 = 55$

Average Turn around time = $\frac{55}{4} = 13.75$

SJFC (Shortest Job First):- It is the best approach to minimize the waiting time.

* It selects the smallest execution time to execute next.

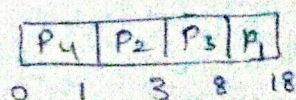
eg:- 1)

Process	Burst time
P ₁	10
P ₂	2
P ₃	5
P ₄	1

2)

Process	Burst time
P ₁	12
P ₂	10
P ₃	5
P ₄	6

Grantt chat



Waiting time = 12

Average waiting time = 3

Turn around time = 30

Average Turn around time = 7.25

3)

Process	Burst time
P ₁	6
P ₂	8
P ₃	7
P ₄	3

Drawback:-

1. Total execution time of a process must be known, before execution while it is not possible to perfectly predict execution time.

Priority Scheduling Algorithm:-

1. A priority number (integer) is associated with each process.
2. The CPU is allotted to the process with highest priority.
3. SJF is priority scheduling algorithm i.e. Reverse order smallest one first.

Drawback:- The problem with priority scheduling is starvation.

Starvation:- Suppose when low priority process may never execute in the CPU.

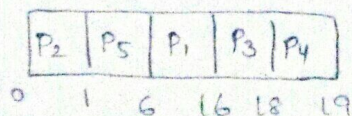
Solution for starvation aging.

Aging means as time progress increases giving the priority to process.

eg:-

Process	Burst time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

Gantt Chart:-



Waiting time = $0 + 1 + 6 + 16 + 18 + 19 = 41$

Average waiting time = $\frac{41}{5} = 8.2$

Turn around time = $1 + 6 + 16 + 18 + 19 = 60$

Average turn around time = $\frac{60}{5} = 12$

2)

Process	Bursttime	priority
P1	5	3
P2	7	2
P3	2	4
P4	4	1

3)

Process	Bursttime	priority
P1	6	4
P2	7	1
P3	10	3
P4	3	5
P5	2	2

4)

Process	Bursttime	priority
P1	12	5
P2	16	4
P3	8	1
P4	4	2
P5	1	3

Shortest Job first (SJF):-

2)

P1	12
P2	10
P3	5
P4	6

3)

P1	6
P2	8
P3	7
P4	3

Gantt chart:-

P3	P4	P2	P1
----	----	----	----

0 5 11 21 33

Waiting time = $0 + 5 + 11 + 21 = 37$

Average waiting time = $\frac{37}{4} = 9.25$

Turn around time $P1=12, P2=10, P3=5, P4=6$

$12 + 10 + 5 + 6 = 33$

Average Turn around time = $\frac{33}{4} = 8.25$

4)

P1	24
P2	3
P3	3

Gantt chart -

P3	P2	P1
----	----	----

0 3 6 30

W.T = $0 + 3 + 6 = 9$

AWT = $9/3 = 3$

TAT = $3 + 6 + 30 = 39$

ATAT = $39/3 = 13$

Gantt chart:-

P4	P1	P3	P2
----	----	----	----

0 3 9 16 24

Waiting time = $0 + 3 + 9 + 16 = 28$

Average waiting time = $\frac{28}{4} = 7$

Turn around time = $3 + 9 + 16 + 24 = 52$

Average Turn around time = $\frac{52}{4} = 13$

5)

P1	5
P2	16
P3	3
P4	15
P5	1
P6	15
P7	4

Gantt chart:-

P5	P3	P7	P1	P2	P4	P6
----	----	----	----	----	----	----

0 1 4 8 13 18 33

W.T = $0 + 1 + 4 + 8 + 13 + 18 + 33 = 87$

AWT = $87/7 = 12.42$

TAT = $1 + 4 + 16 + 13 + 23 + 38 + 53 = 140$

ATAT = $140/7 = 20$

Priority scheduling Algorithm:-

2)

P1	5	3
P2	7	2
P3	2	4
P4	4	1

Gantt chart:-

P_4	P_2	P_1	P_3	
0	4	11	16	18

$$W.T = 0 + 4 + 11 + 16 = 31$$

$$AWT = \frac{31}{4} = 7.75$$

$$TAT = 4 + 11 + 16 + 18 = 49$$

$$ATAT = \frac{49}{4} = 12.25$$

3)

P1	6	4
P2	7	1
P3	10	3
P4	3	5
P5	2	2

Gantt chart:-

P2	P5	P3	P1	P4	
0	7	9	19	25	27

$$W.T = 0 + 7 + 9 + 19 + 25 = 60$$

$$AWT = \frac{60}{5} = 12$$

$$TAT = 7 + 9 + 19 + 25 + 27 = 87$$

$$ATAT = \frac{87}{5} = 17.4$$

4)

P1	12	5
P2	16	4
P3	8	1
P4	4	2
P5	1	3

$$W.T = 0 + 8 + 12 + 13 + 29 = 62$$

$$TAT = 8 + 12 + 13 + 29 + 41 = 103$$

$$AWT = \frac{62}{5} = 12.4$$

$$ATAT = \frac{103}{5} = 20.6$$

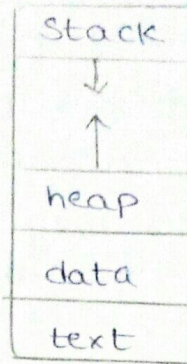
Gantt Chart:-

P3	P4	P5	P2	P1	
0	8	12	13	29	41

PROCESS:- A process is a program in execution. A process is more than a program code sometimes known as text section. It also includes the current activity as represented by the value of the program counter and the contents of the processor registers. A process generally also includes the process stack which contains temporary data and a data section which contains global variables. A process may also include a heap which is memory that is dynamically allocated during process runtime.

Program itself is not a process, a program is a passive entity such as a file containing a list of instructions stored on disk (called an executable file) whereas process is an active entity with a program counter specifying the next instruction to execute and a set of associated resources. A program becomes a process when an executable file is loaded into memory. Although two processes may be associated with the same program, they are considered two separate execution sequences.

max



A process in memory

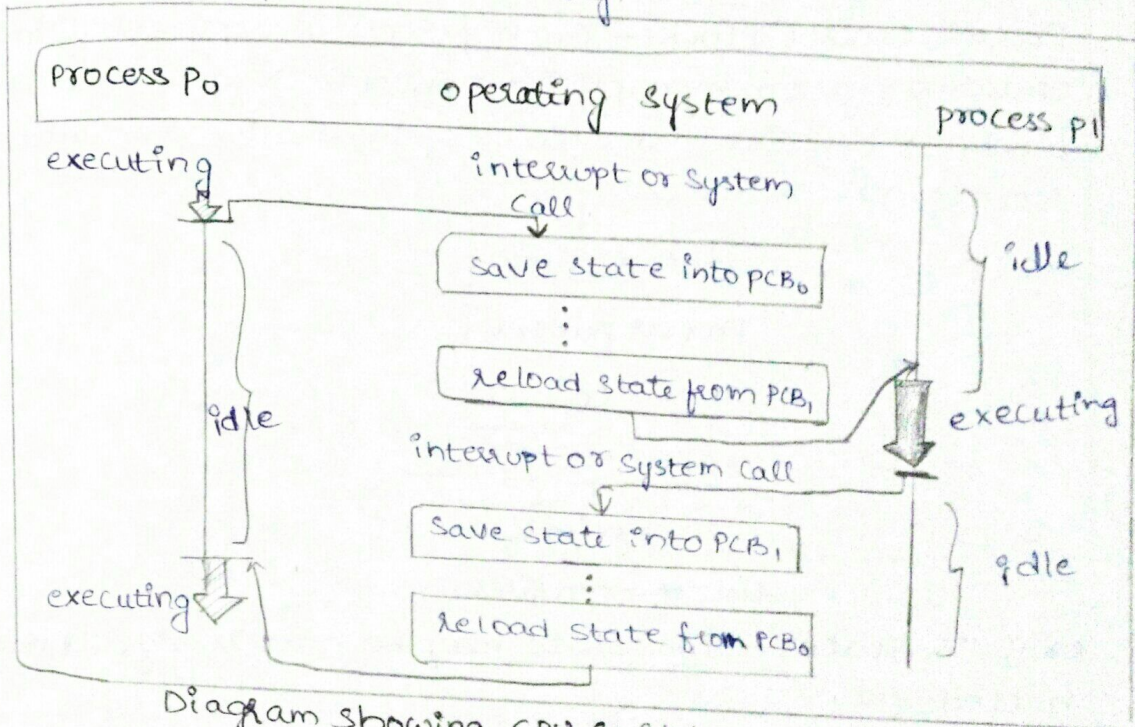


Diagram showing CPU Switch from process to process

Process state:— As a process executes, it change state. The state of a process is defined in part by the current activity of that process. Each process may be in one of the following states

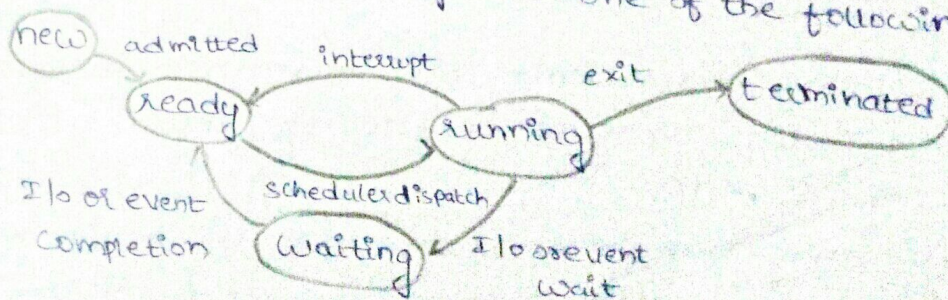


Diagram of process state

- * process may be in one of 5 states, as shown in figure 3.2 below
- New — The process is in the stage of being created.
- Ready — The process has all the resources available that it needs to run, but the CPU is not currently working on this process instruction.

Running:— The CPU is working on this process instruction.

Waiting:— The process cannot run at the moment, because it is waiting for some resource to become available or for some event to occur. For example, the process may be waiting for keyboard input, disk access request, inter-process messages, a timer to go off, or a child process to finish.

Terminated:— The process has completed.

Process Control Block:— Each process is represented in the operating system by a process control block also called Task Control block. It contains many pieces of information associated with a specific process including.

Process state
Process number
Program Counter
Registers
Memory limits
List of open files

Process state:— The state may be new, ready, running, waiting, halted etc.

Program Counter:— The counter indicates the address of the next instruction to be executed for this process.

CPU registers:— The registers vary in number and type depending on the computer architecture.

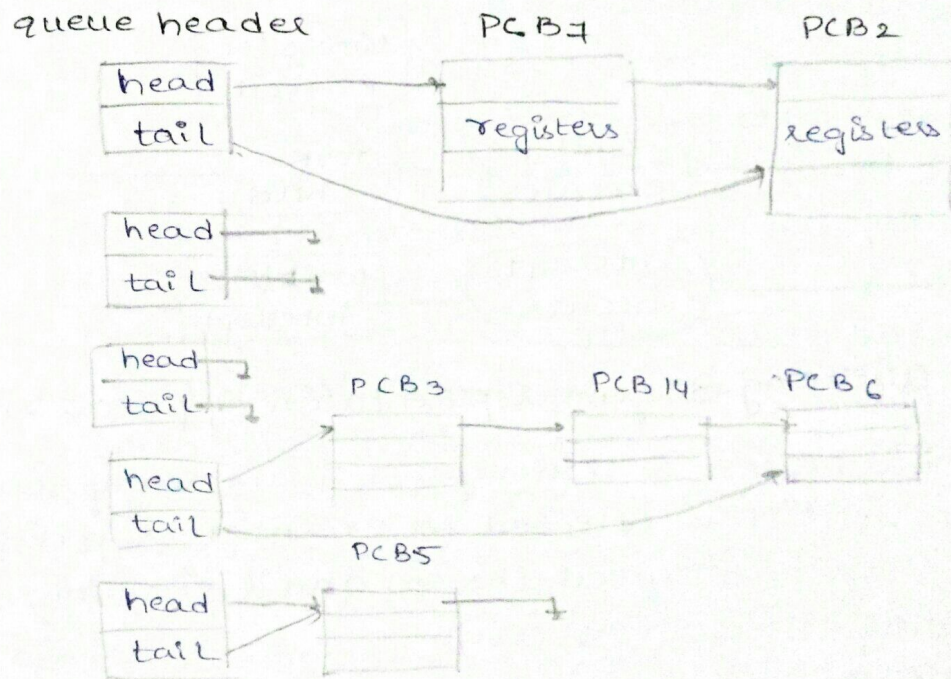
CPU scheduling information:— This information includes a process priority, pointer to scheduling queues, and other scheduling parameters.

Memory management information:— This information may include such information as the value of base and limit registers etc.

Accounting information:— This information includes the amount of CPU and real time used, time limits etc.

I/O status information:— This information includes the list of I/O devices allocated to the process etc.

process scheduling:— The objective of multiprogramming is to have some process running at all times to maximize cpu utilization. The objective of time sharing is to switch the cpu among processes so frequently that users can interact with each program while it is running. To meet these objectives, the process scheduler selects an available process for program execution on the cpu. For program a single processes system, there will never be more than one running process.

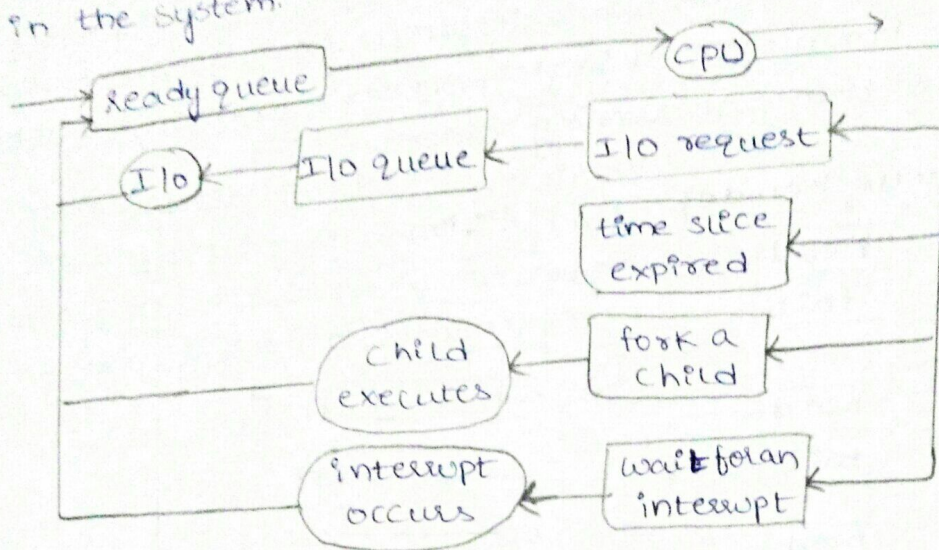


Scheduling queues:— As processes enter the system, they are put inside the job queue, which consists of all processes in the system. The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the ready queue. This queue is stored as a linked list. A ready queue header contains pointer to the first and final PCBs in the list. Each PCB includes a pointer field that points to the next PCB in the ready queue.

When a process is allocated the cpu it executes for a while and eventually quits, is interrupted or waits for the occurrence of a particular event such as the completion of an I/O request. The list of processes waiting for a particular I/O device is called a device queue. Each device has its own device queue.

A common representation for process scheduling is

queuing diagram. Each rectangular box represents a queue. Two types of queues are present: ready queue and a set of device queues. Circles represent the resources that serve the queues and the arrows indicate the flow of processes in the system.



Queuing-diagram representation of process scheduling

A new process is initially put in the ready queue. It waits there until it is selected for execution or is dispatched once the process is allocated the CPU and is executing, one of the following events might occur.

- The process could issue an I/O request and then be placed in the I/O queue.
- The process could create a new sub process and wait for the sub process termination.
- The process could be removed forcibly from the CPU as a result of an interrupt and be put back in the ready queue.

Scheduler: - A program migrated among the various scheduling queues throughout its life time. The OS must select for scheduling purposes, processes from these queues and this selection is carried out by scheduler.

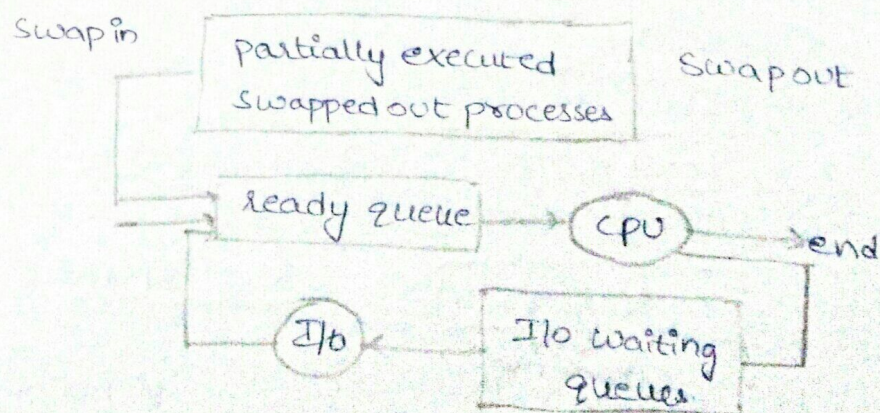
In a batch system, more processes are submitted than can be executed immediately. These processes are spooled to a mass storage device (disk) where they are kept for later execution. The long term scheduler or job scheduler selects processes from this pool and loads them into memory for

execution. The Short term scheduler or cpu scheduler selects from among the processes that are ready to execute and allocates the cpu to one of them.

The distinction between these two lies in the frequency of execution. The long term scheduler controls the degree of multi programming (the number of processes in memory).

most processes can be described as either I/O bound or cpu bound. An I/O bound process is one that spends more of its time doing I/O than it spends doing computations. A cpu bound process generates I/O requests infrequently using more of its time doing computations. long term scheduler must select a good mix of I/O bound and cpu bound processes. A system with best performance will have a combination of cpu bound and I/O bound processes.

Some operating systems such as time sharing systems may introduce additional/intermediate level of scheduling. Idea behind medium term scheduler is that sometimes it can be advantageous to remove processes from memory and thus reduce the degree of multi programming. 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.

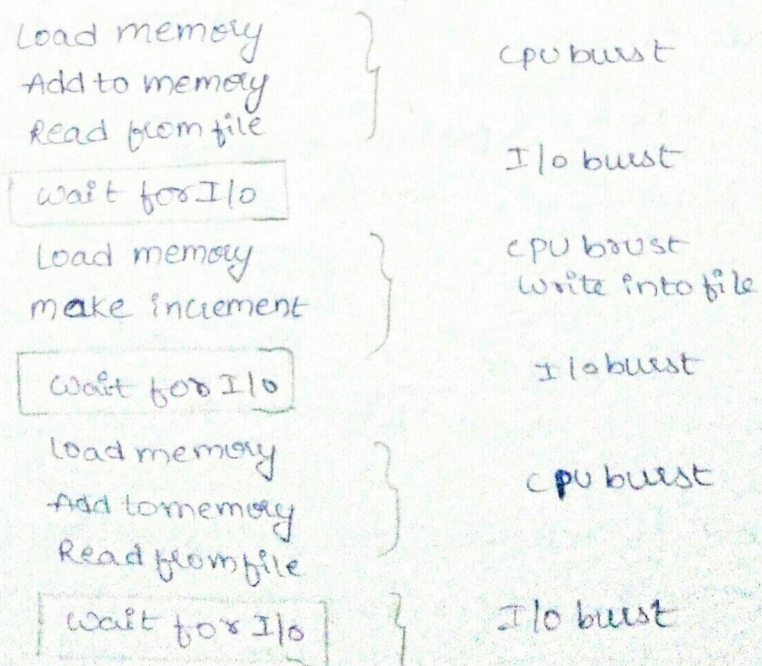


Context switch:— Interrupts cause the OS to change a cpu from its current task and to run a kernel routine. When an interrupt occurs, the system needs to save the current context of the process currently running on the CPU so that it can restore that context when its processing is done, essentially suspending the process and

then resuming it. The Context is represented in the PCB of the process, it includes the value of CPU registers, process state and memory management information. We perform a state save of the current state of the CPU and then state restore to resume operations.

Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process. This task is known as Context Switch. Kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run. Context switch time is pure overhead. Its speed varies from machine to machine depending on the memory speed, the number of registers that must be copied and the existence of special instructions. Context switch times are highly dependent on hardware support.

CPU-I/O burst cycle:- process execution consists of a cycle of CPU execution and I/O wait. Processes alternate between these two states. process execution begins with CPU burst followed by an I/O burst and so on. The final CPU burst ends with a system request to terminate execution.



An I/O bound program has many short CPU bursts
A CPU bound program might have a few long CPU bursts

CPU Scheduling:— whenever the cpu becomes idle, the operating system must select one of the process & the ready queue to be executed. The selection process is carried out by the short term scheduler or cpu scheduler. The scheduler selects a process from the processes in memory that are ready to execute and allocates the cpu to that process.

The ready queue is not necessarily a first in first out queue. A ready queue can be implemented as a fifo queue, a priority queue, a tree or an unordered linked list. All the processes in the ready queue are linked up waiting for a chance to run on the cpu. The records in the queue are the process control blocks of the processes.

Scheduling Criteria:— Different cpu scheduling algorithms have different properties. Criteria for comparing cpu scheduling algorithms.

- a) CPU utilization:— keeps the cpu as busy as possible.
- b) Throughput:— one measure of work of cpu is the number of processes that are completed per time unit called throughput.
- c) Turn around time:— The interval from the time of submission of a process to the time of completion is the turn around time. It 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.
- d) Waiting time:— It is the sum of the periods spent waiting in the ready queue.
- e) Response time:— The time it takes for the process to start responding but is not the time it takes to output the response.

It is desirable to maximize cpu utilization and throughput and to minimize turn around time, waiting time and response time.

Scheduling algorithm:—

First come first serve:— This is the simplest cpu scheduling algorithm. The process that requests the cpu first is allocated the cpu first. The implementation of fcfs is managed with a fifo queue. 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. The average waiting time under FCFS is often quite long. FCFS scheduling algorithm is non preemptive.

once the CPU has been allocated to a process, that process keeps CPU until it releases the CPU either by termination or by requesting I/O. It is simple, fair, but poor performance. Average queuing time may be long.

Shortest Job First:- This algorithm associates with each process the length of the process 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.

SJF algorithm is optimal in that it gives the minimum average waiting time for a given set of processes. The real difficulty with SJF is knowing the length of the next CPU request. SJF scheduling is used frequently in long term scheduling. It cannot be implemented at the level of short term scheduling as there is no way to know the length of the next CPU burst.

The SJF algorithm can be either preemptive or non preemptive. This choice arises when a new process arrives at the ready queue while a previous 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 will preempt the currently executing process whereas non preemptive SJF algorithm will allow the currently executing process to finish its CPU burst. Preemptive SJF is sometimes called shortest remaining time first scheduling.

Priority:- The SJF is a special case of the general priority scheduling algorithm. A priority is associated with each process and CPU is allocated to the priority process with highest priority. Equal priority processes are scheduled in FCFS order. An SJF algorithm is simply a priority algorithm where the priority is the inverse of the predicted next CPU burst. The larger the CPU burst, the lower the priority. priority

are generally indicated by some fixed range of numbers usually 0 to 7.

Priorities can be defined either internally or externally. Internally defined priorities use some measurable quantity to compute the priority of a process. External priorities are set by criteria outside the operating system.

Priority scheduling can be either preemptive or non-preemptive. When a process arrives at the ready queue, its priority is compared with the priority of the currently running process. A preemptive priority scheduling algorithm 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 put the new process at the head of the ready queue.

The major problem with priority scheduling algorithm is indefinite blocking or starvation. A process that is ready to run but waiting for the CPU can be considered blocked. A priority scheduling algorithm can leave some low priority process waiting indefinitely. A solution to the problem of indefinite blockage of low priority processes is aging. Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time.

Round Robin:- The round robin scheduling algorithm is designed for time sharing systems. It is 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. A time quantum is generally from 10 to 100 milliseconds. The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue allocating the CPU to each process for a time interval of up to 1 time quantum. To implement RR scheduling, ready queue is implemented as a FIFO queue. New processes are added to the tail of ready queue. The CPU scheduler picks the first process from the ready queue, sets the timer to interrupt after 1 time quantum.

and dispatches the process.

The process may either have a CPU burst of less than 1 time quantum. Then, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. Else if the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed and the process will be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue.

In the RR Scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row. If a process CPU burst exceeds 1 time quantum, that process is preempted and is put back in the ready queue. Thus this algorithm is preemptive.

The performance of RR algorithm depends heavily on the size of time quantum. If the time quantum is extremely large, the RR policy is same as the FCFS policy. If the time quantum is extremely small, the RR approach is called processor sharing.

eg:-

eg:-

Process Id	Burst time
P1	3 → 1 → 0
P2	6 → 4 → 2
P3	4 → 2 → 0
P4	2 → 0

Time quantum = 2

turn around $P_1 = 3 + 6 = 9$

$P_2 = 6 + 9 = 15$

$P_3 = 4 + 9 = 13$

$P_4 = 2 + 6 = 8$

eg:- Gantt Chart

P1	P2	P3	P4	P1	P2	P3	P2	
0	2	4	6	8	9	11	13	15

Waiting time: $0 + 2 + 4 + 6 + 8 + 10$

$P_1 = 0 + (8 - 2) = 6$

$P_2 = 2 + (9 - 4) + (13 - 11) = 9$

$P_3 = 4 + (11 - 6) = 9$

$P_4 = 6$

waiting time = $6 + 9 + 9 + 6 = 30$

$$\text{Average waiting time} = \frac{30}{4} = 7.5$$

$$\text{Turn around time} = \text{Burst time} + \text{Waiting time}$$

$$= 9 + 15 + 13 + 8 = 45$$

$$\text{Average Turn around time} = \frac{45}{4} = 11.25$$

Process Id	Burst time	Completion time	Waiting time	Turn around time
P1	3	9	6	9
P2	6	15	9	15
P3	4	13	9	13
P4	2	8	6	8

2)

Process	Burst time	Time quantum = 5
P1	16	$\rightarrow 11 \rightarrow 6 \rightarrow 1$
P2	10	$\rightarrow 5 \rightarrow 0$
P3	5	$\rightarrow 0$

Sol:- Grantl chatt.

P ₁	P ₂	P ₃	P ₁	P ₂	P ₁	P ₁	
0	5	10	15	20	25	30	31

$$\text{Waiting time } P_1 = 0 + (15-5) + (25-20) + (30-30) = 15$$

$$P_2 = 5 + (20-10) = 15$$

$$P_3 = 10$$

$$\text{Average waiting time} = \frac{40}{3} = 13.33$$

$$\text{Turn around time} = \text{Burst time} + \text{Waiting time}$$

$$P_1 = 16 + 15 = 31$$

$$P_2 = 10 + 15 = 25$$

$$P_3 = 5 + 10 = 15$$

$$\text{Average Turn around time} = \frac{71}{3} = 23.6$$

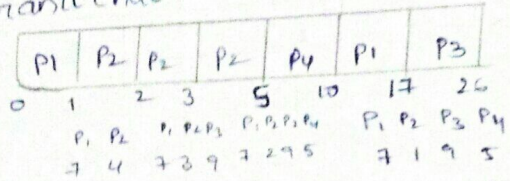
Process Id	Burst time	Completion time	Waiting time	Turn around time
P1	16	31	15	31
P2	10	25	15	25
P3	5	15	10	15

Shortest Remaining Time First (SRTF):-

eg:-

Process	Burst time	Arrival time
P1	8	0
P2	4	1
P3	9	2
P4	5	3

Sol:- Gantt chart



Process Id	Burst time	Arrival time	Completion time	TAT (CT-AI)	WT (TAT-BT)
P1	8	0	17	17	9
P2	4	1	5	4	0
P3	9	2	26	24	15
P4	5	3	10	7	2

$$TAT = 17 + 4 + 24 + 7 = 52$$

$$WT = 9 + 0 + 15 + 2 = 26$$

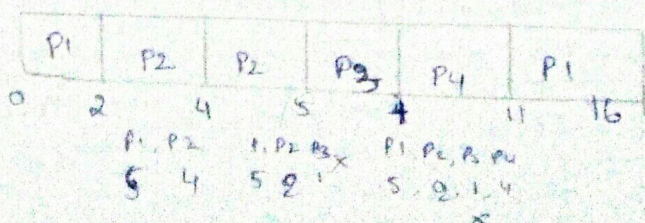
$$ATAT = \frac{52}{4} = 13$$

$$AWT = \frac{26}{4} = 6.5$$

2)

P Id	BT	AT	CT	TAT	WT
P1	7	0	16	16	9
P2	4	2	7	5	1
P3	1	4	5	1	0
P4	4	5	11	6	2

Sol:- Gantt chart:-



$$TAT = 16 + 5 + 1 + 6 = 28$$

$$WT = 9 + 1 + 0 + 2 = 12$$

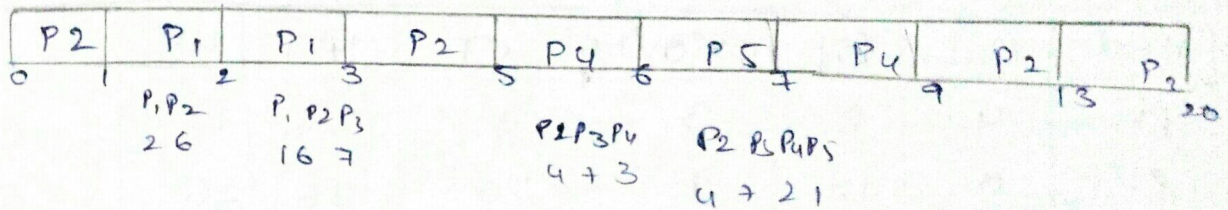
$$ATAT = \frac{28}{4} = 7 \text{ ms}$$

$$AWT = \frac{12}{4} = 3 \text{ ms}$$

3) PID

	AT	BT	CT	TAT	WT
P1	1	2	3	2	0
P2	0	7	13	13	6
P3	2	7	20	18	11
P4	5	3	9	4	1
P5	6	1	7	1	0

Gantt chart



$$ATAT = 18 + 2 + 13 + 5 = \frac{38}{5} = 7.6$$

$$AWT = \frac{0 + 6 + 11 + 1}{5} = \frac{18}{5} = 3.6$$

24/7/18

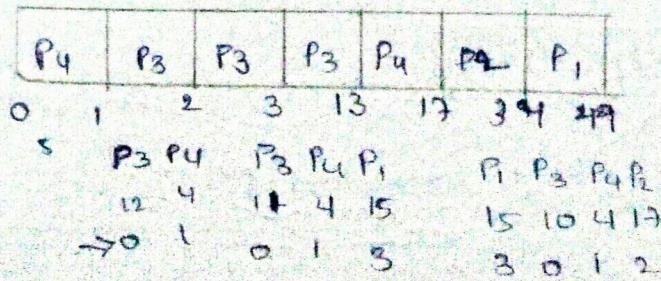
Preemptive priority CPU scheduling algorithm:-

eg: if

Process	AT	BT	priority	CT	TAT	WT
P1	2	15	3	49	49	32
P2	3	17	2	34	31	14
P3	1	12	0	13	12	0
P4	0	5	1	17	17	12

107 58

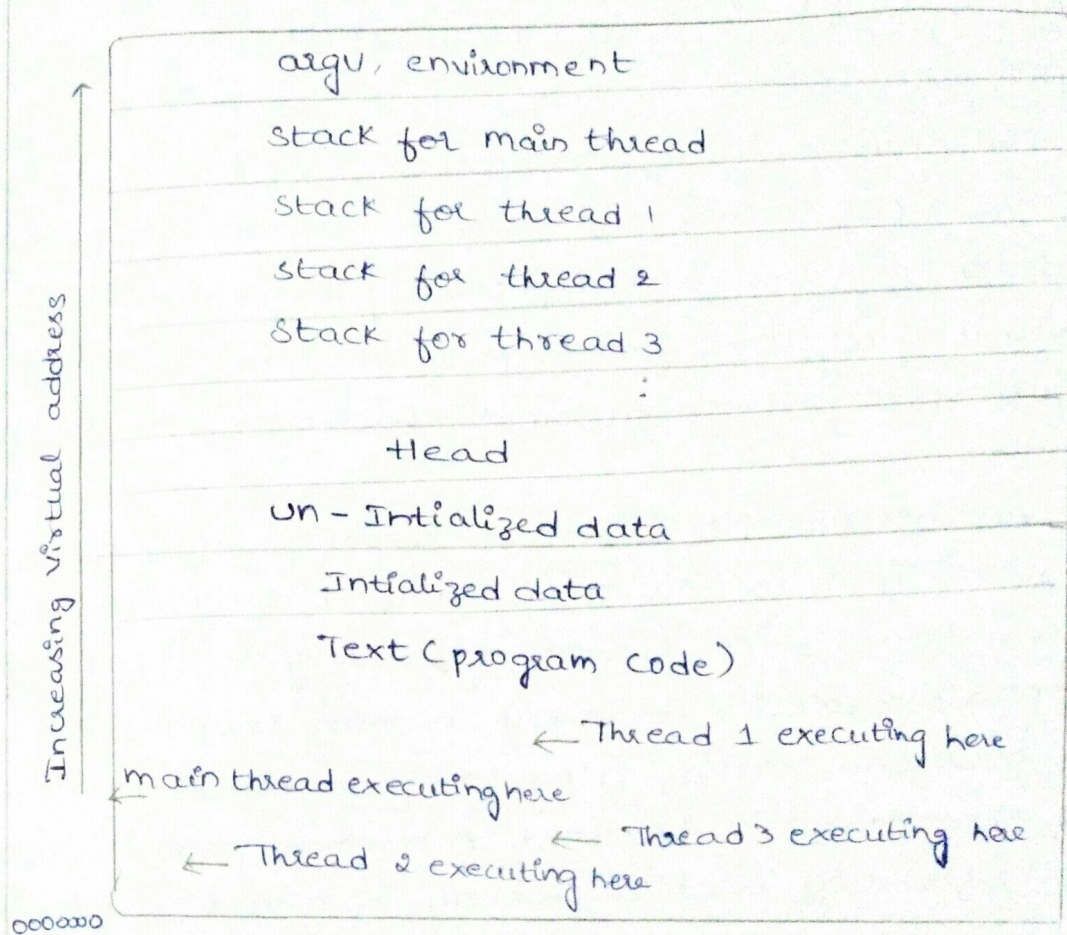
Sol:- Gantt chart:-



$$ATAT = \frac{107}{4} = 26.75 \text{ ms}$$

$$AWT = \frac{58}{4} = 14.5 \text{ ms}$$

Threads in memory (in segments as parts the memory is allocated)



Thread:-

1. A thread is a basic unit of CPU utilization, consisting of a program counter, a stack and a set of registers (and a thread ID).
2. Traditional (heavy weight) processes have a single thread of control. There is a one program counter, and one sequence of instructions that can be carried out at any given time.
3. As shown in figure below multi threaded applications have multiple threads within a single process, each having their own program counter, stack and set of registers, but sharing common code, data and certain structures such as open files.

Motivation:-

1. Threads are very useful in modern programming whenever a process has multiple tasks to perform.

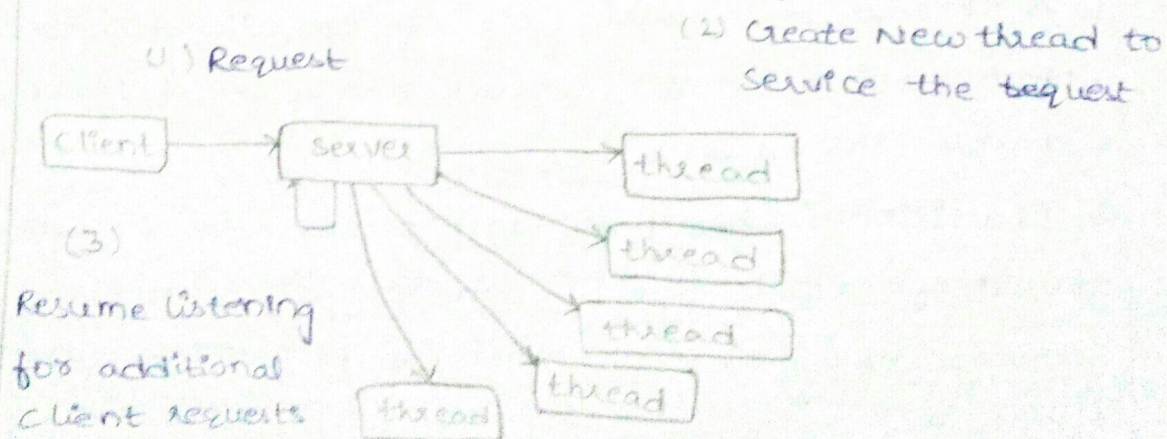
of the others

2. This is particularly true when one of the tasks may block, and it is desired to allow the other tasks to proceed without blocking.

3. for example in a word processor, a background thread may check spelling and grammar while a foreground thread processes user input (key strokes) while yet a third thread loads images from the hard drive, and a fourth does periodic automatic backups of the file being edited.

4. Another example is a web server - multiple threads allow for multiple requests to be satisfied simultaneously without having to service requests sequentially or to fork off separate processes for every incoming request (The latter is how this sort of thing was done before the concept of threads was developed. A daemon would listen at a port, fork off a child for every incoming request to be processed, and then go back to listening to the port).

Multithreaded Server Architecture:-



Benefits:- There are four major categories of benefits to multi-threading.

1. Responsiveness:- one thread may provide rapid response while other threads are blocked or slowed down intensive calculations.

2. Resource Sharing:- By default threads share common code, data and other resources, which allows multiple tasks to

be performed simultaneously in single address space.

3. Economy:— Creating and managing threads (and context switch between them) is much faster than performing the same tasks for processes.

4. Scalability:— utilization of multiprocessor architecture a single threaded process can only run on one cpu, no matter how many may be available where as the execution of a multi-threaded Application may be split amongst available processors.

Multithreading models:—

1. There are two types of threads to be managed in a modern system: user threads and kernel threads.
2. user threads are supported above the kernel, without kernel support. These are the threads that application programmers would put into their programs.
3. kernel threads are supported within the kernel of the operating system itself. All modern OSes support kernel level threads, allowing the kernel to perform multiple simultaneous tasks or to service multiple kernel system calls simultaneously.

In a specific implementation, the user threads must be mapped to kernel threads, using one of the following strategies.

1. Many to one model:—

1. In the many to one model, many user level threads are all mapped onto a single kernel thread.
2. Thread management is handled by the thread library in user space, which is very efficient.
3. However, if a blocking system call is made, then the entire process blocks, even if the other user threads would otherwise be able to continue.
4. Because a single kernel thread can operate only on a single cpu, the many to one model does not allow individual processes to be split across multiple cpus.
5. Green threads for Solaris and GNU portable threads implement the many to one model. In the past, the few

System continue to do so today.

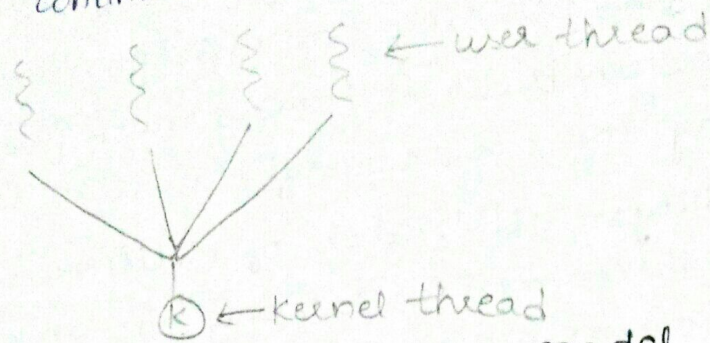


figure - many to one model.

2. one to one model:-

- * The one to one model creates a separate kernel thread to handle each user thread.
- * one to one model overcomes the problems listed above involving blocking system calls and the splitting of processes across multiple CPUs.
- * However the overhead of managing the one to one model is more significant, involving more overhead and slowing down the system.
- * most implementations of this model place a limit on how many threads can be created.
- * Linux and Windows from 95 to XP implement the one to one model for threads.

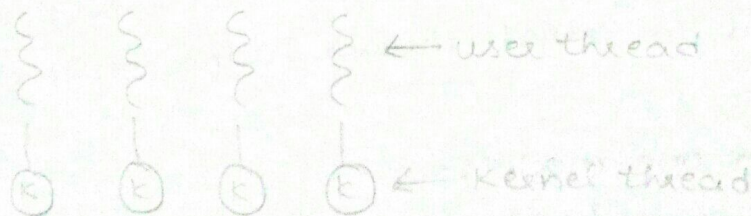


figure: one to one model

3. many to many model:-

- * The many to many model multiplexes any number of user threads onto an equal or smaller number of kernel threads.
- * Combining the best features of the one to one and many to one models.
- * User have no restrictions on the number of threads created.
- * Blocking kernel system calls do not block the entire process.
- * processes can be split across multiple processors.

* Individual processes may be allocated variable numbers of kernel threads depending on the number of CPUs present and other factors.

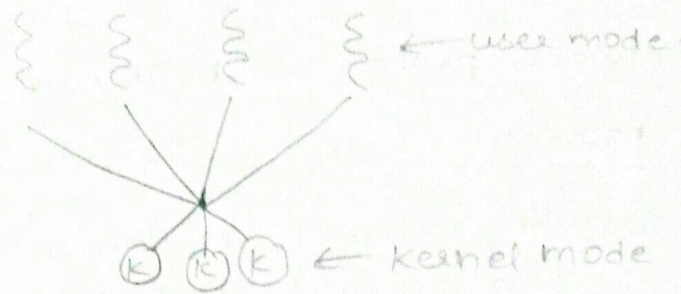
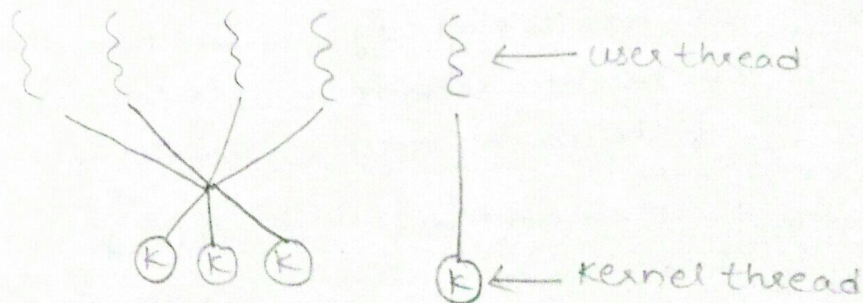


figure - many to many model.



Two level model.

Thread Libraries:-

1. Thread libraries provide programmer with an API for creating and managing threads.
2. Thread libraries may be implemented either in user space or in kernel space. The former involves API functions implemented solely within user space, with no kernel support. The latter involves system calls and requires a kernel with thread library support.
3. There are three main thread libraries in use today.
 - (i) posix threads:- May be provided as either a user or kernel library as an extension to the posix standard. It will be available on OS Linux, Solaris etc
 - (ii) win 32 threads:- provided as a kernel-level library on Windows systems.
 - (iii) Java threads:- Since Java generally runs on a Java virtual machine, the implementation of threads is based

upon whatever os and hardware the JVM is running on
i.e either p threads or win 32 threads depending on the
systems

Comparison between process and Thread.

	process	Thread
Definition	An executing Instance of a program is called a process.	A thread is a subset of the process.
process	It has its own copy of the data segment of the parent process.	It has direct access to the data segment of its process.
Communication	processes must use inter-process communication to communicate with sibling processes.	Threads can directly communicate with other threads of its process.
overheads	processes have considerable overhead	Threads have almost no overhead.
Creation	New processes require duplication of the parent	New threads are easily created.
Control	processes can only exercise control over child processes	Threads can exercise considerable control over threads of the same process.
Changes	Any change in the parent process does not affect child processes	Any change in the main thread may affect the behavior of the other threads of the process.
Memory	Run in separate memory spaces	Run in shared memory spaces.
file descriptors	most file descriptors are not shared	It shares file descriptors

file system	There is no sharing of file system context	It shares file system context.
signal	It does not share signal handling.	It shares signal handling.
Controlled by	process is controlled by the operating system	Threads are controlled by programmer in a program.
Dependence	processes are independent	Threads are dependent

Thread Scheduling:- One distinction between user level and kernel level threads lies in how they are scheduled. On systems implementing the many to one and many to many models, the thread library schedules user-level threads to run on an available lwp. a scheme known as process contention scope (PCS).

Since competition for the CPU takes place among threads belonging to the same process. When we say the thread library schedules user threads onto available lwps, we do not mean that the thread is actually running on a CPU; this would require the operating system to schedule the kernel thread onto a physical CPU. To decide which kernel thread to schedule onto a CPU, the kernel uses system contention scope (SCS).

Competition for the CPU with SCS scheduling takes place among all threads in the system. Systems using the one to one model (such as windows, xp, Solaris, linux) schedule threads using only SCS.

Typically PCS is done according to priority. The scheduler selects the runnable threads with the highest priority to run. user-level thread priorities are set by the programmer and are not adjusted by the thread library although some thread libraries may allow the programmer to change the priority of a thread.

It is important to note that PCs will typically ^{process} the thread currently running in favour of a higher ^{priority} thread; however, there is no guarantee of time slicing among threads of equal priority.

Multilevel queue scheduling:-

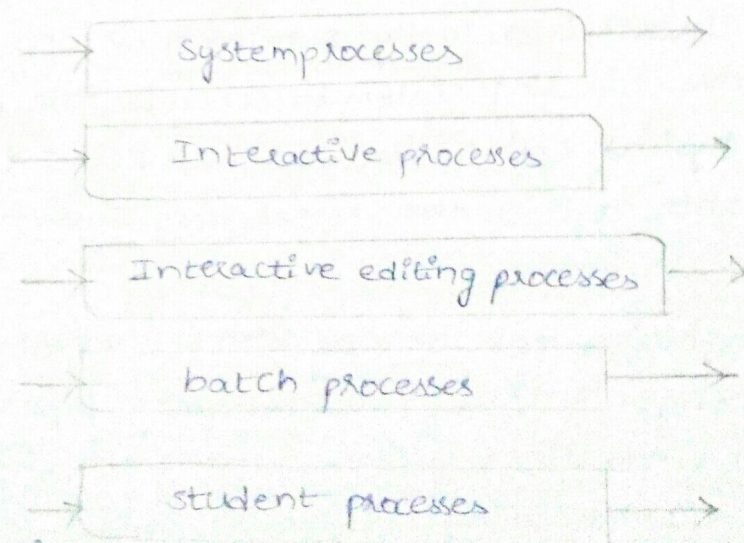
* When processes can be readily categorized, then multiple separate queues can be established, each implementing whatever scheduling algorithm is most appropriate for that type of job, and/or with different parametric adjustments.

* scheduling also must be done between queues, that is scheduling one queue to get time relative to other queues.

Two common options are strict priority (no job in a lower priority queue runs until ^{all} high priority queues are empty) and round robin (each queue gets a time slice in turn, possibly of different sizes).

* Note that under this ^{algorithm} job cannot switch from queue to queue once they are assigned a queue, that is either queue until they finish.

Highest priority



lowest priority

Figure: multilevel queue scheduling

Multilevel feedback queue scheduling:-

* Multilevel feedback queue scheduling is similar to ^{ordinary} multilevel queue scheduling described above, except jobs may be moved from one queue to another for a variety of reasons.

- * If the characteristics of job change between cpu intensive and io intensive then it may be appropriate to switch a job from one queue to another.
- * Aging can also be incorporated so that job that has waited for a long time can get bumped up into a higher priority queue for a while.
- * multilevel feed back queue scheduling is the most flexible, because it can be tuned for any situation. But it is also the most complex to implement because of all the adjustable parameters some of the parameters which defines one of these system include.
 - * The number of queues
 - * The scheduling algorithm for each queue.
 - * The methods used to upgrade or demote processes from one queue to another (which may be different).
 - * The method used to determine which queue a process enters initially.

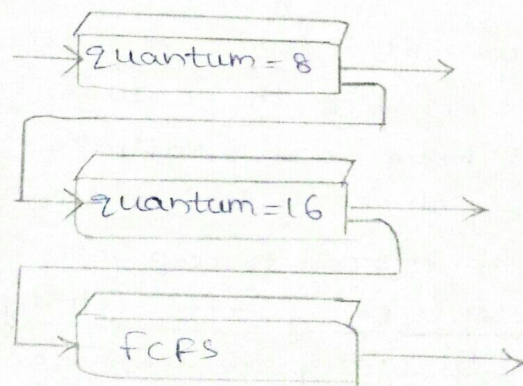


figure: multilevel feedback queue scheduling.

Multiprocessor Scheduling:-

- * When multiple processor are available then the scheduling gets more complicated because now there is more than one cpu which must be kept busy and its effective use at all time.
- * loading sharing revolves around balancing the load between multiple processors
- * multiprocessor system may be heterogeneous (different kind of cpus) or homogeneous (all the same kind of cpu). even in the latter case there may be special scheduling constraints, such as devices which are connected via a private bus to

only one of the cpus. This book will restrict its discussion to homogenous systems

Approaches to multi-processor scheduling :-

- * one approach to multi processor scheduling is asymmetric multiprocessing in which one processor is the master, controlling all activities and running all kernel code, while the other runs only user code. This approach is relatively simple as there is no need to share critical system data.
- * Another approach is symmetric multiprocessing smp, where each processor schedules its own jobs either from a common ready queue or from separate ready queue for each processor.
- * virtually all modern oses support smp, including xp, win 2000, solaris, linux, and mac osx.

processor Affinity :-

- * processor contain cache memory which speed up repeated accesses to the same memory locations.
- * If process were to switch from one processor to another each time it got a time slice, the data in the cache (for that process) would have to be invalidated and reloaded from main memory, thereby obviating the benefit of the cache.
- * Therefore smp system attempt to keep processes on the same processor. via processor affinity, soft affinity occurs when the system attempts to keep processes on the same processor but makes no guarantees. linux and some other oses support hard affinity in which a process specifies that it is not to be moved between processors.
- * main memory architecture can also affect process affinity if particular cpus have faster access to memory on the same chip or board than to other memory located elsewhere. (Non-uniform memory Access, Numa) as shown below, if assigned memory storage in "local" fast access areas.

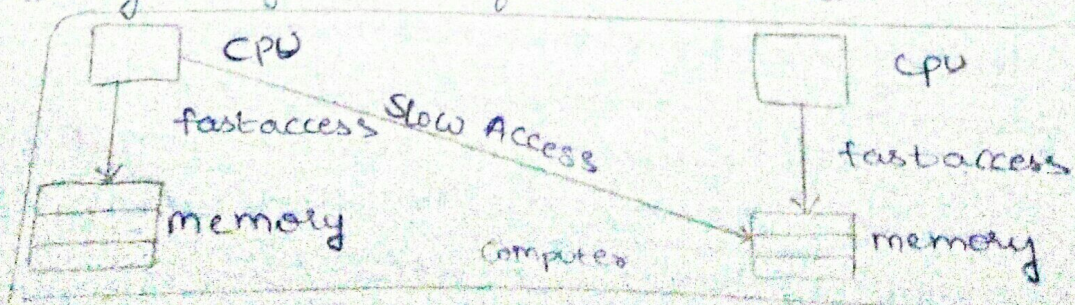


figure NUMA and cpu scheduling

Load balancing:-

- * obviously an important goal in a multiprocessor system is to balance the load between processors so that one processor won't be sitting idle while another is overloaded.
- * System using a common ready queue are naturally self balancing and don't need any special handling. most systems, however, maintain separate ready queue for each processor.
- * Balancing can be achieved through either push migration or pull migration.
- * push migration involves a separate process that runs periodically (eg every 200 milliseconds) and moves processes from heavily loaded processor onto less loaded ones.
- * pull migration involves idle processor taking processes from the ready queues of other processors.
- * push and pull migration are not mutually exclusive.
- * Note that moving processes from processor to processor to achieve load balancing works against the principle of processor affinity and if not carefully managed the savings gained by balancing the system can be lost in rebuilding cache. one option is to only allow migration when imbalance surpasses a given threshold.

Real time operating scheduling:-

The real time operating system used for a real-time application means for those applications where data processing should be done in the fixed and small quantum of time. It is different from general purpose computer where time concept is not considered as much crucial as in real time operating system. RTOS is a time-sharing system based on clock interrupts.

- * RTOS are controlling traffic signal; Nuclear reactors control scientific experiments, medical imaging systems, industrial system, fuel injection system, home appliance are some applications of Real time operating system.

Real time operating systems are very fast and quick respondent systems. These systems are used in an environment where a large number of events (generally external) must be accepted and processed in a short time. Real time processing requires quick

transaction and characterized by supplying immediate response. for example, a measurement from a petroleum refinery indicating that temperature is getting too high and might demand for immediate attention to avoid an explosion.

* In real time operating system there is a little swapping of programs between primary and secondary memory. most of the time, processes remain in primary memory in order to provide quick response. therefore memory management in real time system is less demanding compared to other systems.

few more examples of real-time processing are:

1. Airlines reservation system.
2. Air traffic control system.
3. systems that provide immediate updating.
4. systems that provide up to the minute information on stock prices.
5. Defense application systems like as RADAR.

Real time operating systems mostly use the preemptive priority scheduling. These support more than one scheduling policy and often allow the user to set parameters associated with such policies, such as the time slice in Round Robin scheduling where each task in the task queue is scheduled up to a maximum time, set by the time-slice parameter in a round robin manner. Hundred of the priority levels are commonly available for scheduling. Some specific tasks can also be indicated to be non preemptive.

Types of Real-Time operating system:—

1. Soft real time operating system:— A process might not be executed in given deadline. It can be crossed it then executed next, without harming the system. Example are a digital camera, mobile phones etc.
2. Hard Real time operating system:— A process should be executed in given deadline. The deadline should not be crossed. preemption time for Hard Real-time operating system is almost less than few microseconds.
Examples are Air bag control in cars, anti lock brake, engine control

Process Coordination:-

Critical section is part of program where shared resources are accessed by various process.

Race Condition:-

It says when order of execution can change result.
eg:- producer consumer.

The critical section problem solution is divided into two types according to use of process involved.

- i) two process solution ii) multi process solution.

Critical solution problem must follow some conditions to give a solution.

- * Mutual exclusion (no other process can be entered in critical section)
- * progress (process which wish to execute will enter initial solution)
- * Bounded waiting.

Race condition

eg:- $P_1()$

```
{  
①  $C = B - 1$   
②  $B = 2 * C$   
}
```

$P_2()$

```
{  
③  $D = 2 * B$   
④  $B = D - 1$   
}
```

1234 $\Rightarrow$
 $C = 1$
 $B = 2$
 $D = 2 * 2 = 4$
 $B = 4 - 1 = 3$

3412 $\Rightarrow$
 $D = 2 * 2 = 4$
 $B = 4 - 1 = 3$
 $C = 3 - 1 = 2$
 $B = 2 * 2 = 4$

1342 $\Rightarrow$
 $C = 2 - 1 = 1$
 $D = 2 * 2 = 4$
 $B = 4 - 1 = 3$
 $B = 2 * 1 = 2$

3124 $\Rightarrow$
 $D = 2 * 2 = 4$
 $C = 2 - 1 = 1$
 $B = 2 * 1 = 2$
 $B = 4 - 1 = 3$

1324 $\Rightarrow$
 $C = 2 - 1 = 1$
 $D = 2 * 2 = 4$
 $B = 2 * 1 = 2$
 $B = 4 - 1 = 3$

3142 $\Rightarrow$
 $D = 4$
 $C = 1$
 $B = 3$
 $B = 2$

To avoid the race condition process should be synchronized

General structure of process P_i :

```
Pi do
{
    entry section
    critical section
    Exit section
    Remainder section
} while (true);
```

Types of critical

2- process solution:

vs mp

Peterson's Solution:

code for entry section.

P_0 <pre>while(1) { while (turn != 0); critical section turn = 1; Remainder section; }</pre>	P_1 <pre>while(1) { while (turn != 1); critical section turn = 0; Remainder section; }</pre>
--	--

It satisfies mutual exclusion.

Progress:- strict interleaving.

P_1 is finished P_0 get chances

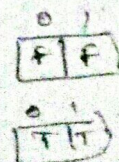
It not satisfies progress.

Case (ii)

P_0 <pre>while(1) { flag[0] = T; while (flag[1]); // CS flag[0] = F; }</pre>	P_1 <pre>while(1) { flag[1] = T; while (flag[0]); // CS flag[1] = F; }</pre>
--	--

Which is
interested
should
enter

False



It satisfies mutual exclusion but not progress

Case (iii):-

P₀
 while (1)
 { flag[0] = T;
 turn = 1;
 // while (turn == 1 && flag[1] = T);
 CS ✓ T && F = F
 flag[0] = F;
 }

Context switch

P₁
 while (1) turn = 0
 { flag[1] = T;
 turn = 0;
 while (turn == 0 && flag[0] = T);
 CS ✗ T && T = T
 flag[1] = F;
 }

Shared variables are turn and flag.

∴ It satisfies the mutual exclusion, progress and bounded waiting.

(After some time it gives chance to other process)

Perterson's solution gives solution to critical section by two process solution.

N Variable solution:-

Semaphores (S): It is a integer variable apart from initialization is accessed only through two standard automatic operation i.e wait-function, wait and signal()

It doesn't want to maintain busy waiting

All modifications through the integer value of the semaphore is wait () & signal (). operations must be executed individually.

* Semaphore is a general synchronization tool it can solved as (i) counting semaphore (ii) Binary semaphore

Wait ():-

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

Signal ():-

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

automatic function

P_i

```
{  
  entry section — wait(S) n variable can be achieved  
  critical section  
  exit section  
  remainder section — Signal(S)  
}
```

* Any semaphore we initialize as $s = 1$

* 1) S is used as a solution for critical section problem

* 2) Semaphore is used to decide the order of execution among the process.

* 3) Resource management i.e. any resource with multiple copies for that S has to manage proper management

* There exists two types of semaphores

1. Binary semaphore :- It can range only between 0 and 1. We use binary semaphore to deal with critical section problem for multiple process.

2. Counting semaphore :- It can range over an unrestricted domain. Counting semaphores can be used to control access given resources consisting of finite number of instances.

Note:- Semaphore is initialized to a number of resources available.

Semaphore are used to solve critical section problem

(Binary semaphore):-

We can satisfy mutual exclusion, progress and bounded waiting by using Binary semaphores as follow.

Step 1:- Initialize $s = 0$ for solving critical section problem

```
do  
{  
  Semaphore = 1  
  wait(S)  
  critical section  
  signal(S)  
} while(1)
```


Implementation:- Consider P_1 and initially $s=1$

while ($1 \leq 0$); false P_2
 $s--$; $1--$ $0 \leq 0$;
 $s=0$;

P_1 in critical section

While $0 \leq 0$ P_2 will be in the loop

while $0 \leq 0$ P_3 will be in the loop

while $0 \leq 0$ P_4 will be in the loop

for P_1 after critical section it gives the incremental value by using signal. i.e $s=1$

So P_1 is executed and remaining are made to execute further with value $s=1$.

Therefore mutual exclusion is satisfied as only one process is present in critical section.

Progress:- Therefore we achieve progress as more processes can be executed according to their interest.

Therefore Bounded waiting is satisfied as it gives signal to other process (As P_1 will not stay in the critical section) when it is coming out i.e s value is incremented by 1 and giving the signal to enter into critical section.

Deciding the order of execution:-

Eg:- P_1, P_2, P_3 must be execute in an order suppose P_2, P_1, P_3 it can be achieved as follows

P_1	P_2	P_3
wait(s_1) ↓ signal(s_1)	signal(s_1)	wait(s_2)

$P_2 \rightarrow P_1 \rightarrow P_3$ here
 s_1, s_2 are semaphore variables

After getting the safe sequence order given by operating system the process gets executed so that safe sequence order in deciding the order of execution.

In above example we initialized $s_1=1$ and $s_2=1$

managing the Resource (or) Resource management:- (Countable Semaphore),

Wait(s)

{
Critical section
}

Signal(s)

Remainder section.

order of process execution we
use Countable Semaphore

eg:- 5 process and 5 printers

Wait(s)

{
while (S <= 0)
Critical section
}

Here S = 5

P₁ P₂ P₂ P₃ P₄ P₅

All the five processors enter into
CS until S = 0 then the

When another process P₆ wants to enter into critical section as S = 0 it will not enter but any one of the entered process completes its process the S value is incremented so that process can be executed.

classical problem of synchronization:-

1. bounded buffer Problem [producer-consumer problem]
2. Reader/writer problem
3. dining philosophers problem

Process Synchronization

Definition:-

A race condition is an undesirable situation that occurs when a device or system attempts to perform two or more operations at the same time, but because of the nature of the device or system the operations must be done in the proper sequence to be done correctly.

Cooperating process:- A process is independent if it cannot affect other process or be affected by it. Any process that does not share data with other is independent. otherwise the process is cooperating, cooperation is done to provide information sharing, computational speedups, modularity and convenience.

process synchronization means sharing system resources by processes in a such a way that concurrent access to shared data is handled ^{these} by minimizing the chance of inconsistent data. maintaining data consistency demands mechanisms to ensure synchronized execution of cooperating processes.

A critical section is a code segment that accesses shared variables and has to be executed as an atomic action. It means that in a group of cooperating processes at a given point of time, only one process must be executing its critical section.

(or)

In concurrent programming, concurrent accesses to shared resources can lead to unexpected or erroneous behavior. So parts of the program where the shared resources is accessed are protected. This protected section is the critical section or critical region. It cannot be executed by more than one process at a time. Typically, the critical section accesses a shared resource, such as a data structure, a peripheral device, or a network connection, that would not operate correctly in the context of multiple concurrent accesses.

Peterson's algorithm (or Peterson's solution) is a concurrent programming algorithm for mutual exclusion that allows two or more processes to share a single use resource without conflict, using only shared memory for communication. It was formulated by Gray L. Peterson in 1981. While Peterson's original formulation worked with only two processes, the algorithm can be generalized for more than two.

Semaphore vs Mutex: In the case of mutex only the thread that locked or acquired the mutex can unlock it. In the case of a semaphore, a thread waiting on a semaphore can be signaled by a different thread. Some operating systems support using mutex and semaphores between threads. Typically usage is created in shared memory.

A semaphore is an integer variable, which can be accessed only through two operations: wait() and signal().

All the processes can share the same mutex semaphore that is initialized to 1. Then, a process has to wait until the lock becomes 0. Then the process can make the mutex semaphore 1 and start its critical section.

~~Consumer~~ process: -

```
while (true)
```

```
{ while (counter == 0); /* do nothing */
```

```
  next consumed = buffer[out];
```

```
  out = (out + 1) % Buffer-size;
```

```
  counter --;
```

```
  /* Consume the item in next consumed */
```

```
}
```

Producer process:

```
while (true)
```

```
{ /* produce an item in next produced */
```

```
  while (counter == Buffer-size); /* do nothing */
```

```
  buffer[in] = next produced;
```

```
  in = (in + 1) % Buffer-size;
```

```
  counter ++;
```

```
}
```

first producer
next consumer

* Unfortunately we have now introduced a new problem, because both the producer and consumer are adjusting the value of the variable counter, which can lead to a condition known as race condition. In this condition a piece of code may or maynot work correctly, depending on which of two simultaneous processes executes first and more importantly if one of the processes gets interrupted such that the other process runs between important steps of the first process (Bank balance example discussed in class)

* The particular problem above comes from the producer executing "counter++" at the same time the consumer is executing "counter--". If one process gets part way through making the update and then the other process butts in, the value of counter can get left in an incorrect state.

* But, you might say, "Each of those are single instructions. How can they get interrupted halfway through?" The answer is that although they are single instructions in C++, they are

actually three steps each at the hardware level.

1. fetch counter from memory into a register
2. increment or decrement the register
3. store the new value of counter back to memory.

If the instructions from the two processes get interleaved, there could be serious problem, such as illustrated by the following.

Producer:

register 1 = counter
register 1 = register 1 + 1
counter = register 1

Consumer:

register 2 = counter
register 2 = register 2 - 1
counter = register 2

Interleaving:

T0:	Producer	execute	register 1 = counter [register 1 = 5]
T1:	producer	execute	register 1 = register 1 + 1 [register 1 = 6]
T2:	consumer	execute	register 2 = counter [register 2 = 5]
T3:	consumer	execute	register 2 = register 2 - 1 [register 2 = 4]
T4:	producer	execute	counter = register 1 [counter = 6]
T5:	consumer	execute	counter = register 2 [counter = 4]

* Exercise: What would be the resulting value of counter if the order of statements T4 and T5 were reversed? (What should the value of counter be after one producer and one consumer, assuming the original value was 5?)

* Note that race conditions are notoriously difficult to identify and debug, because by their very nature they only occur on rare occasions and only when the timing is just exactly right (or wrong! :-)) Race conditions are also very difficult to reproduce.

* obviously the solution is to only allow one process at a time to manipulate the value "counter". This is a very common occurrence among cooperating processes, so let's look at some ways in which this is done, as well as some classic problems in this area.

The critical-section problems:-

* The producer-consumer problem described above is a specific example of a more general situation known as the critical section problem. The general idea is that in a number of

cooperating processes, each has a critical section of code, with the following conditions and terminologies:

- * only one process in the group can be allowed to execute in their critical section at any one time. If one process is already executing their critical section and another process wishes to do so, then the second process must be made to wait until the first process has completed their critical section work.
- * The code preceding the critical section and which controls access to the critical section is termed the entry section. It acts like a carefully controlled locking door.
- * The code following the critical section and which controls access to the critical section, is termed the exit section. It generally releases the lock on someone else's door, or at least lets the world know that they are no longer in their critical section.
- * The rest of the code not included in either the critical section or the entry or exit sections is termed the remainder section.

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

Figure 5.1 - General structure of a typical process P:

- * A solution to the critical section problem must satisfy the following three conditions:
 1. Mutual exclusion - only one process at a time can be executing in their critical section.
 2. Progress - If no process is currently executing in their critical section, and one or more processes want to execute their critical section then only the processes not in their remainder sections can participate in the decision, and the decision cannot be postponed indefinitely. (i.e. processes cannot be blocked forever waiting to get into their critical sections).
 3. Bounded waiting - There exists a limit as to how many other processes can get into their critical sections after a process requests entry into their critical section and before that request is granted (i.e. a process requesting entry into their critical section and before that request is granted, i.e. a process requesting entry

into critical section will get a turn eventually and there is a limit as to how many other processes get to go first.
* We assume that all processes proceed at a non zero speed but no assumptions can be made regarding the relative speed of one process versus another.

* Kernel processes can also be subject to race conditions, which can be especially problematic when updating commonly shared kernel data structures such as open file tables or virtual memory management. Accordingly kernels can take on one of two forms.

* Non preemptive kernels do not allow processes to be interrupted while in kernel mode. This eliminates the possibility of kernel mode race conditions, but requires kernel mode operations to complete very quickly, and can be problematic for real-time systems, because timing cannot be guaranteed.

* preemptive kernels allow for real-time operations but must be carefully written to avoid race conditions. This can be especially tricky on SMP systems, in which multiple kernel processes may be running simultaneously on different processors.

Non preemptive kernels include Windows XP, 2000, traditional UNIX, and Linux prior to 2.6, preemptive kernels include Linux 2.6 and later, and some commercial UNIXes such as Solaris and IRIX.

Peterson's solution:-

* Peterson's solution is a classic software-based solution to the critical section problem. It is unfortunately not guaranteed to work on modern hardware, due to vagaries of load and store operations, but it illustrates a number of important concepts.

* Peterson's solution is based on two processes P_0 and P_1 , which alternate between their critical sections and non-critical sections. For convenience of discussion "this" process is P_i and "other" process is P_j (i.e. $j = 1 - i$).

* Peterson's solution requires two shared data items.
int turn:- indicates whose turn it is to enter into the critical section. If $turn = i$, then process i is allowed into their critical section.

Boolean flag[]:- indicates when a process wants to enter into their critical section. When process i wants to enter their critical section, it sets $flag[i]$ to true.

* In the following diagram, the entry and exit sections are enclosed in boxes.

- i) In the entry section, process i first raises a flag indicating a desire to enter the critical section.
- ii) Then $turn$ is set to j to allow the other process to enter their critical section if process j so desires.
- iii) The while loop is a busy loop (notice the semicolon at the end), which makes process i wait as long as process j has the $turn$ and wants to enter the critical section.
- iv) process i lowers the $flag[i]$ in the exit section, allowing process j to continue if it has been waiting.

```
do
{
    flag[i] = True;
    turn = j;
    while (flag[j] && turn == j);
    Critical section
    flag[i] = false;
    remainder section
} while (true);
```

figure 5.2 - The structure of process P_i in peterson's solution.

* To prove that the solution is correct, we must examine the three conditions listed above:

1. mutual exclusion:- If one process is executing their critical section when the other wishes to do so, the second process will become blocked by the flag of the first process. If both processes attempt to enter at the same time, the last process to execute " $turn=j$ " will be blocked.
2. progress:- Each process can only be blocked at the while $(flag[j] == true)$, AND it is the other process $turn$ to use the critical section $(turn == j)$. If both of those conditions are true, then the other process j will be allowed to enter the critical section, and upon exiting the critical section, will set $flag[j]$ to false, releasing process i . The shared variable $turn$ assures that only one process at a time can be blocked, and the flag variable allow one process to release the other when exiting their critical section.
3. Bounded waiting:- As each process enters their entry section, they set the $turn$ variable to be the other processes $turn$. Since no process ever sets it back to their own $turn$, this ensures

that each process will have to let the other process go first at most one time before it becomes their turn again.

- * Note that the instruction "turn = j" is atomic, that is it is a single machine instruction which cannot be interrupted.

Semaphores:-

- * A more robust alternative to simple mutexes is to use semaphores, which are integer variables for which only two (atomic) operations are defined, the wait and signal operations, as shown in the following figure.
- * Note that not only must the variable-changing steps (s- and s++) be indivisible, it is also necessary that for the wait operation when the test proves false that there be no interruption before s get decremented. It is okay, however, for the busy loop to be interrupted when the test is true, which prevents the system from hanging forever.

wait:

```
wait(s)
{
  while s <= 0;
  s--;
```

Signal

```
Signal(s)
{
  s++;
```

Semaphore usage:-

- * In practice, semaphores can take on one of two forms: Binary Semaphores can take on one of two values 0 or 1. They can be used to solve the critical section problem as described above, and can be used as mutexes on systems that do not provide a separate mutex mechanism. The use of mutexes for this purpose is shown in the below figure.

mutual exclusion implementation with semaphores:-

```
do
{
  waiting (mutex); // Critical section
  signal (mutex); // remainder section
} while (true);
```

Counting semaphores can take on any integer value, and are usually used to count the number remaining of some limited resource. The counter is initialized to the number of such resources available in the system, and whenever the counting semaphore is greater than zero, then a process can

enter a critical section and use one of the resources. When the counter gets to zero (or negative in some implementations) then the process block until another process frees up a resource and increment the counting semaphore with a signal call. (The binary semaphore can be seen as just a special case where the number of resources initially available is just one).

Semaphore can also be used to synchronize certain operations between processes. For example, suppose it is important that process P₁ executes statement S₁ before process P₂ executes statement S₂.

- * First we create a semaphore named `synchron` that is shared by two processes, and initialize it to zero

- * Then in process P₁ we insert the code:

```
S1;  
signal(synch);
```

- * and in process P₂ we insert the code:

```
wait(synch);  
S2;
```

- * Because `synch` was initialized to 0, process P₂ will block on the wait until after P₁ executes the call to `signal`.

Semaphore Implementation:-

- * The big problem with semaphores as described above is the busy loop in the wait call, which consumes CPU cycles without doing any useful work. This type of lock is known as spinlock, because the lock just sits there and spins while it waits. While this is generally a bad thing, it does have the advantage of not invoking context switches and so it is sometimes used in multi-processing systems when the wait time is expected to be short - one thread spins on one processor while another completes their critical section on another processor.

An alternative approach is to block a process when it is forced to wait for an available semaphore and swap it out of the CPU. In this implementation each semaphore needs to maintain a list of processes that are blocked waiting for it, so that one of the processes can be woken up and swapped back in when the semaphore becomes available. (Whether it gets swapped back into the CPU immediately or whether it needs to hang out in the ready queue for a while is a scheduling problem).

The new definition of a semaphore and the corresponding wait and signal operations are shown as follow

Semaphore structure:-

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

wait operation:-

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

signal operation:-

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

Note that in this implementation the value of the semaphore can actually become negative in which case its magnitude is the number of processes waiting for that semaphore. This is a result of decrementing the counter before checking its value.

Key to the success of semaphores is that the wait and signal operations be atomic, that is no other process can execute a wait or signal on the same semaphore at the same time. (Other processes could be allowed to do other things, including working with other semaphores, they just can't have access to this semaphore) on single processors this can be implemented by disabling interrupts during the execution of wait and signal; multiprocessor systems have to use more complex methods, including the use of spinlocking.

classical problems of synchronization:-

1. Bounded buffer problem:- [producer - consumer problem]

Procedure:-

do

{

 Producer()

 wait(E)

 wait(cs)

 append()

 Signal(S)

 Signal(f) $0+1=1$

 while(true); $S=S+1$

exit
section

$S = X \emptyset 1$

$E = X 4$ } 5

$F = \emptyset 1$

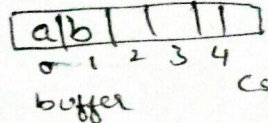
$S = X \emptyset 1$

$E = X 3$ } 5

$F = 2$

append the value to the buffer

↓



It checks whethere buffer is overflow (x) underflow

Note:- Avoiding the Race condition

Consumer:- checking the underflow condition

do

{

 wait(F)

 wait(cs)

 take()

 Signal(S)

 Signal(E)

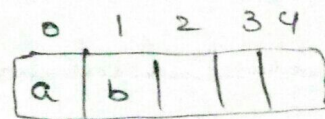
} while(true);

$S = X 0$

$F = X 1$

$E = 4$

$E + F = \text{buffer size}$



traps
maintained

2. Readers/writers problem:- To avoid the conflicts of read-write, write-read, write-write we are writing the code

Any document can be read by many people and the write operation should not be performed.

mutex:- mutual exclusion to solve the critical section problem. we are using the semaphore variable

Writer:-

wait(wrt)

// writer

Signal(wrt)

Number of
persons can read the
particular document

Last
reader lock
should be released.

Reader:- 0 + 1 + 2

do

{

wait(mutex); // lock on
the reader
function

ReadCount++;

if (ReadCount == 1)
The write should stop to avoid the
conflict

wait(wrt);

Signal(mutex);

// reading;

wait(mutex); // 0

readCount--;

if (readCount == 0);

Signal(wrt); // 1

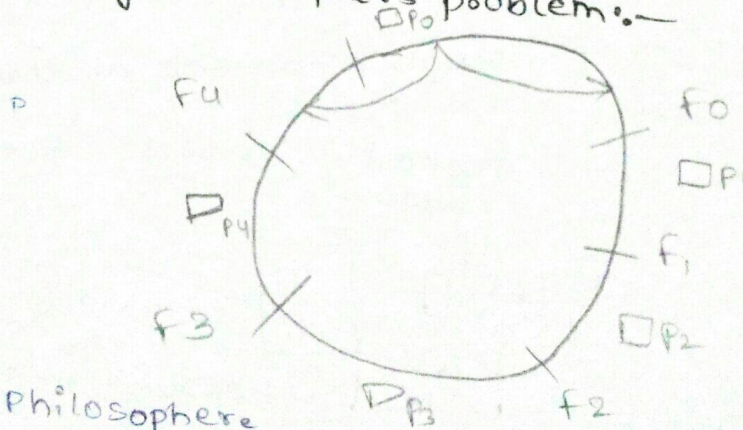
Signal(mutex); // 1

while (true);

}

28/8/18

Dining philosophers problem:-



Philosophers

Rice - critical section

starvation arises over here

while (true)

{

wait(chopstick[i]);

wait(chopstick[i+1] + 5);

// eat


```

    signal (chopstick[i]);
    signal (chopstick[i+1] / 5);
}
}

```

Five philosophers are sitting around the circular table. Dining table has 25 chop sticks and a bowl of rice in the middle.

philosophers either can be eat or think.

When a philosopher wants to eat he uses a chop stick.

When a philosopher wants to think he keeps down both chop sticks.

problem is no philosopher can know when other wants to eat or think.

The Actual problem is if all are interested eat at the same time. Every one will pick one chopstick and sit so there will be a deadlock situation because of this no philosopher can eat at that position.

Paging:-

