- The state of the thread can be changed any no. of times.

* Scheduling policy:-

- 3 diff. sched algorithms -
  i). FIFO ii). Round Robin iii). Others.
  - Real time - Real time. - Non Real time

- Total 140 priorities available - 0 to 139.
- 0-99 - Real Time Priority
- 100-139 - Non real time priority.

- Round Robin & FIFO are implemented using the same data structure.
- Round Robin with 0 time quantum is FIFO.

- In Round Robin a process joins at the back of the queue and in FIFO a process joins at the front of the queue.

- So in FIFO the same process keeps on executing until the process is executed completely.

- A non-real time thread will always have a non-real time algorithm so no need to change the policy.

- These algorithms are only for processes in the same priority no.
- Default is others.

— For real time threads/processes the priority value need also be set — b/w 0 and 99.

— All these are done using attributes functions

25/09/19 Nice Value — -20 → +19

— Nice value + 120 → Gives the priority no. for Non Real time proc (From 100 to 139).

— For real time process priority value need to be set by user but for non-real time proc the priority is calculated automatically using nice value.

— Each priority no. maintains a separate linked list of process. and

— Each level can use diff. priority algo.

- CPU Protection — Timer.
  — Time slice expiry.

— In FIFO any algo. time slice expiry is a decision point.

— In FIFO, the processes join at the start of the queue after time slice expiry.

- pthread_attr_setschedparam — set the priority value for a process.
  — struct is used.
  — Pass the addr./pointer of the structure.

— Default value: Priority is 0.

• pthread_attr_getschedparam — Get the priority value of a process.

* Inherit sched — By default new threads will have to given per scheduling policy by the user.

— The new threads can also inherit the policy from the parent thread.

* Scope of a thread — In linux there is only one scope - system scope.

• pthread scope system — All the threads compete with all other threads of the complete system, because each thread is a process.

• pthread_scope_process — CPU is given a particular process and any thread of from that process can executed acc. to scheduling policy.

— Windows.

— Threads compete within it's own process only because each thread does not make a process.

* SetSched Param —

— Attributes of any thread can be changed after creation but not with the attribute functions.

- pthread_setschedparam — Changes the scheduling and priority of a particular thread by giving tid as an arg.

— It is not necessary to call the function from the thread which node to be changed. Can be called from any function.

* Self & Equal :-
- pthread_self — Returns the tid of calling function. thread.

- pthread_equal — Check weat whether two thread identifiers refer to the same thread.

* Detach — Change the thread to detach after a thread is created as joinable.

— A detach thread cannot be made joinable after creation.

- pthread_detach(pthread_t th) —

— works only if no thread has made this thread joinable.

— If another thread has joined this thread then detach will not work.

* Exit — pthread_exit(void *retval).

— Return value of thread is a void *.

— So, so many variables can be stored in

heap and the add. of heap can be passed.

- stack gets cleared after thread is exited so don't pass add. of stack.
- Heap is not cleared by returning from a function. It needs to cleared by the user.

• Finalisation function – If a thread is stuck some where then this function clears thread from all those places.

- Joinable thread has to give the it's return value/structure to other thread. Till then it's not removed.
- Detached thread doesn't need to do so.

* Join – pthread_join( )

- Blocks the thread until some other thread (passed tid by argument) has finished execution.

• void ** thread_return – Add. which stores add. of some other variable.

- The thread which dies passes the add. of it's return value to the calling thread.

Page No.: study time
Date : / /

4/10/19

* Threading Issues :-

- The scope of thread is always till it's parent process only and not above that level.

- If exec() is executed in a thread then the process which created thread will also (In linux) - Because the kill signal goes to the process group id. within the

- Usually all the thread will get killed if any one thread executes exec() because group leader will gets killed.

* Linux Threads -

- clone() - system call.
- do_fork() - Function in kernel.

- File handlers are shared b/w all the threads so no. of files opened will not be increased by 1 on creating a new thread.

* Implicit Threading -

- Thread Pool - Some threads already created and ready to be used to speed up system
- Parallel Lang. - OpenMP, Verilog.

- After the use of thread is completed it is kept back in the pool

ways the it's
not above that

thread then
read will die.
ill signal
id. which to

get killed if
) because

el.

ll the thread
not be incre-
thread.

y created
k up after

it is kept

* Signal Handling —

- One signal handler for the process and all it's threads but diff. signal mask for all threads.
- User defined signals can be masked but not critical signals.
- Signal is given to the process and not the threads.

• TASK_KILL — Uninterruptible blocked state with can be stopped with only SIGKILL signal.

- 90% of the time pthread_exit() will release all the data structures of thread. Only 10% of time the threads are joinable.

* Thread Cancellation —

• Asynch — Thread dies immediately.
• Deferred — Thread dies after reaching a (Synch.) certain cancellation pt. even though the signal was present before.

- Pthread libraries support deferred cancellation.
- pthread_join() is a cancellation point.
- Deferred is preferred over asynch.

9/10/19

CPU Scheduling :-

* Long Time Scheduler - Admit prog. from new to ready
  - Improves multiprogramming.
  - Few or like Linux does not have long time scheduling.

→ Now jobs available ready for execution.
- But less resource available for each job because each process takes some resource to be ready.
- very low.

* Short term Scheduler - From ready to running (Dispatcher)
- 4 Major decision pt - i). Time slice expiry
  ii). Process completion
  iii). Process moves to waiting state from running.
  iv). Higher priority job arrival. - Switches from waiting/ready to running

- (ii) & (iii) are voluntary context switch. - Non Preemptive
- (i) & (iv) are involuntary context switch. - Preemptive

- A non-preemptive scheduler will have only (ii) & (iii) as decision pt.
- A preemptive scheduler all of the above are decision pt.

- Most frequently run scheduler.

- Scheduler is also a process so it has highest priority of 0.

* Middle Term Scheduler :- From system to swap/suspend state, or vice versa
  - Reduces multiprogramming.
  - Increases resources available for each process.
  - Usually by adding a new job the CPU utilization increases.
  - But if by adding a new job the CPU utilization increases then a process needs to be suspended and mid-term scheduler is called.

* Background Process - Usually only CPU process or very less I/O operation.
  - I/O Bound Process - More time in I/O than in CPU for computation.
  - Most systems are designed for I/O bound process.
  - Responsiveness is needed i.e. process should be given enough time.
  - Preemptive scheduling takes more time than non-preemptive scheduling because to check for whenever a new process joins ready state it has to be checked for priority.

* Dispatch latency - Time taken to switch a process.

11/10/19

- saving the data of currently running process and loading the new process.
- That's Non productive work done in dispatcher latency.
- That's why quantum time should not be too low - atleast 1000 times that of disp. -tch latency.

* Response Time - Time taken b/w the submission of a req./ process until the first response/output come.
- Not till the completion of the job.

* Turn Around Time - Total time taken to complete the job after its arrival/in the submission.
- when scheduling as shortest process first, the response time decrease.

- User Oriented - less response time needed.
  - More responsive.
- System - Oriented - Effective and efficient utilization of the processor.
- Server machines are designed for 24x7 operation

* Waiting Time - waiting in ready queue and not in waiting state.

11/10/19

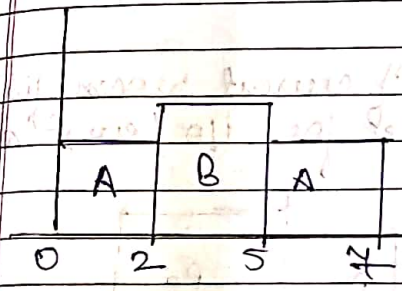
★ CPU Scheduling :-

★ CPU Utilization -

Prod. time - Executing proj.
unprod. time - Management work.
Idle Time - Can be used to improve performance like power management.

	Arrival	Exec.
A -	0	4
B -	2	3

↓ Higher Priority



- Turn Ar. Time (A) = 7-0 = 7
  " " " (B) = 5-2 = 3.

- Normalised Turn. Ar. Time = Turn Ar. Time / Execution Time

- Norm Turn Ar. time (A) = 7/4 ≈ 1.75
  " " " " (B) = 3/3 = 1.

- Best Norm turn ar. time is 1.
  Norm turn ar. time ≥ 1.

- Waiting time (A) = 3
  " " (B) = 0

- Response time (A) = 0
  " " (B) = 0.

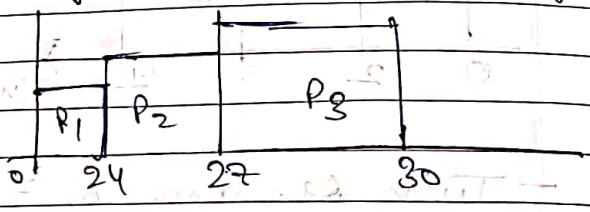
study time
Page No.:
Date : / /

- Waiting Time in Ready queue = Turn Ar. Time − Exec. time − I/O time

* First-Come-First-Serve :-

- Simplest Algo.
- Uses simple queue.
- By default non-preemptive algo. i.e. time slice expiry or higher priority procss is not a decision pt.
- Process changes only if current procss finishes the execution or goes for i/o / event / signal.

Eg :- • $P_1 = 24$
$P_2 = 3$
$P_3 = 3$.

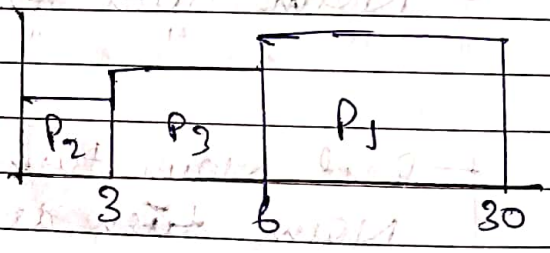


CPU Exec time →
Ready Queue Arrival time →

waiting time $P_1 = 0$
" " $P_2 = 24$
" " $P_3 = 27$

Avg. waiting time $= \frac{24+27}{3} = 17$

• $P_2 \rightarrow P_3 \rightarrow P_1$



waiting time $P_2 = 0$
" " $P_3 = 3$
" " $P_1 = 6$

Avg waiting time $= (3+6)/3 = 3$

study time
Page No.:
Date : / /

= Exec. time
= I/O time

algo. i.e. time
process is not a

process finishes
/ event / signal.

30

P_1

30

study time
Page No.:
Date : / /

shorter

- Convoy effect — Small processes have to wait for long time due to longer processes coming before it.
- Priority is not considered at all.
- I/O bound process suffer because i/o processes are more in no. and shorter duration whereas CPU processes are longer but and less in no.

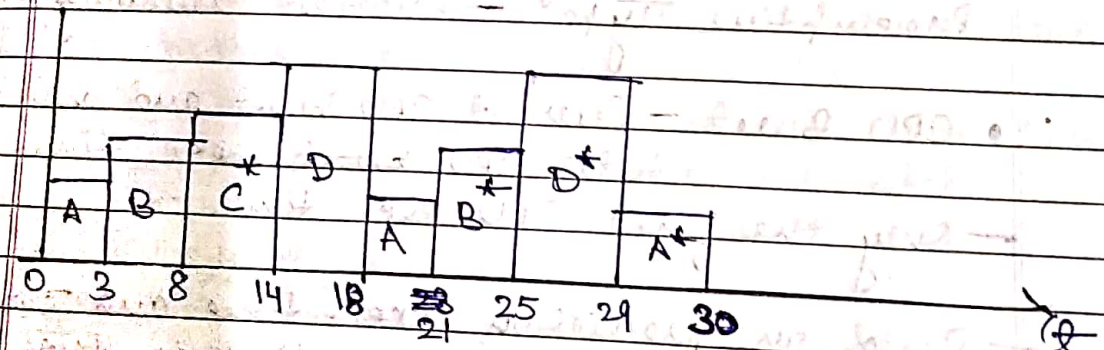
Eg:-

	Arrival time	Exec time (CPU) (total)
A —	0	7 - 3C + 5I/O + 3C + 5I/O + 1C
B —	2	9 - 5C + 2I/O + 4C
C —	4	6 - 6C
D —	6	8 - 4C + 1I/O + 4C

CPU Exec time → 3C 5C 6C 4C 3C 5C 4C 1C

Ready Queue → A | B | C | D | A | B | D | A

Arrival time → 0 2 4 6 8 10 19 26



- Response Time (A) = 0
  " " (B) = 3 - 2 = 1
  " " (C) = 8 - 4 = 4
  " " (D) = 14 - 6 = 8

Page No.: study time
Date : / /

14/10/19

- Turnaround time (A) = 30-0 = 30
" " (B) = 25-2 = 23
" " (C) = 14-4 = 10
" " (D) = 29-6 = 23

- Waitime Time (A) = 30-7-10 = 13
" " (B) = 23-9-2 = 12
" " (C) = 10-6 = 4
" " (D) = 23-8-1 = 14

- Norm. TAT (A) = 30/7
" " (B) = 23/9
(C) = 10/6
(D) = 23/8

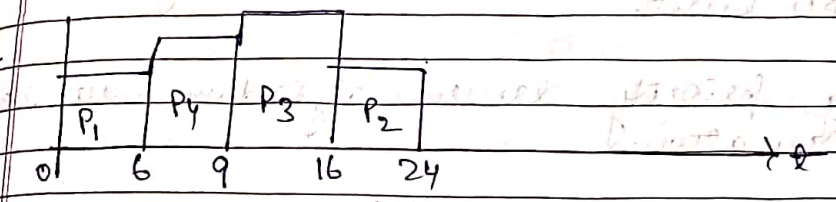
* Shortest Job First :-

- Non-Preemptive type - Shortest job first.
Preemptive Type - Shortest remaining time first.
• CPU Burst - Current CPU burst and not the total CPU exec. time.
- Only the first CPU exe. time.
- Based on prediction, exec. time and I/O time cannot be known before the complete exe. of prog.
- Gives min. avg waiting time.
- Picks the process which has shortest next CPU burst.

study time
Page No.:
Date : / /

14/10/19

• Starvation – If jobs with smaller CPU time keeps on coming so the process with higher CPU times does not gets a chance or it starves.



	Arrival time	Burst time
P_1 –	0	6
P_2 –	1	8
P_3 –	2	7
P_4 –	3	3

Avg. waiting time = (0+15+7+3)/4 = 6.25

– For preemptive type (shortest job first), the decision pt. is only termination of current process or process goes to I/O.
– Even if a shorter job comes in blw, then also the current process will not stop.

* CPU Burst :-

$$T_{n+1} = \alpha t_n + (1-\alpha) T_n \quad ; \quad 0 \le \alpha \le 1.$$

t_n – Actual length of n^{th} CPU burst
T_n – Predicted value of last " "
T_{n+1} – " " " next " "

– Generally $\alpha = 1/2$

- If $\alpha = 0$, next CPU burst prediction depends only on the previous history.
- If $\alpha = 1$, prediction depends only on the last CPU burst.

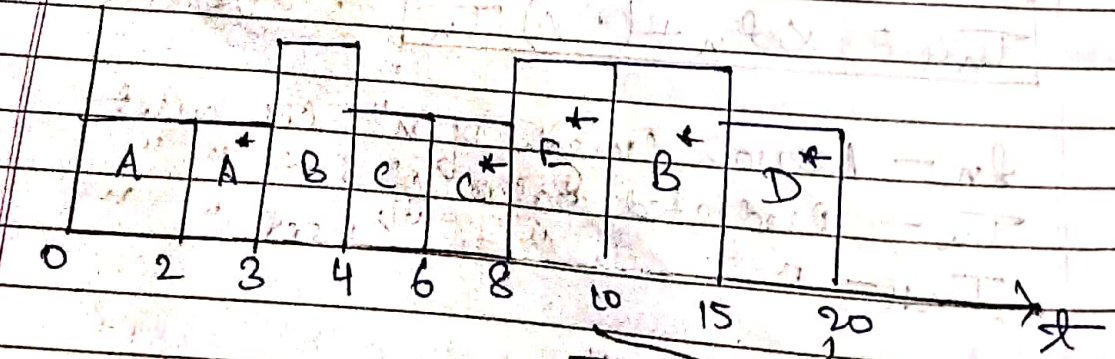
- Any priority driven algorithm will result in starvation.
- FIFO & Round Robin are not priority driven algo, they are queue based algo. — No starvation.

CPU BRS. — time
Ready queue —
Arrival time

* Shortest Remaining Time First :—

— Preemptive time
— Decision At. — Termination, I/O, Arrival of new process.

Process	Arr. Time	Ex. Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2



Tie break b/w B & D, we choose B because it had already executed for some time before.

prediction depends
tory.
only on the last

um will result in

priority driven
lgo. — No starv-

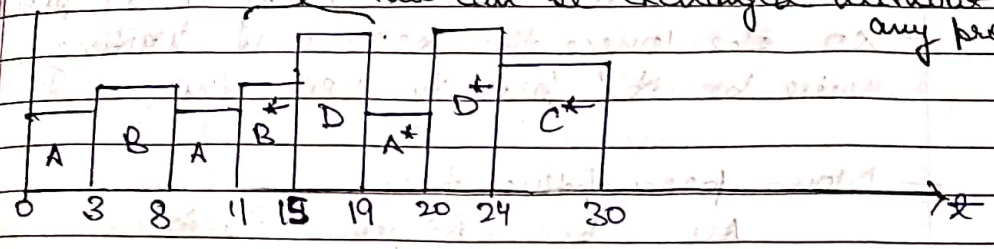
TD, Arrival of D

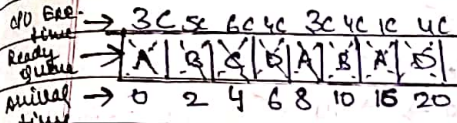
D, so we
it had already
me before.

Eg:

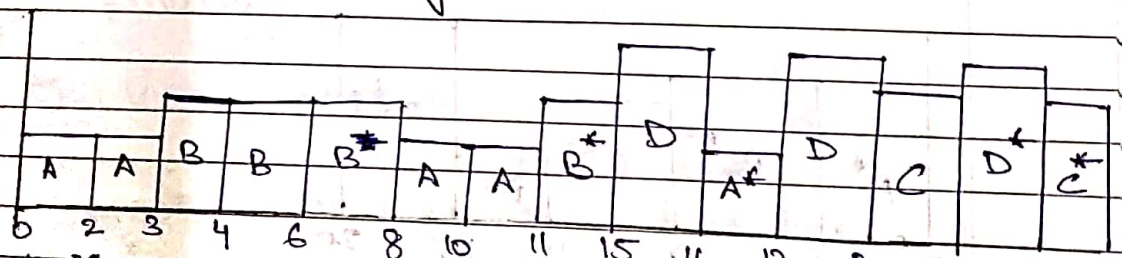
	Arr.	Exe.
A	0	7 — 3C+5I+3C+5I+1C
B	2	9 — 5C+2I+4C
C	4	6 — 6C
D	6	8 — 4C+1I+4C

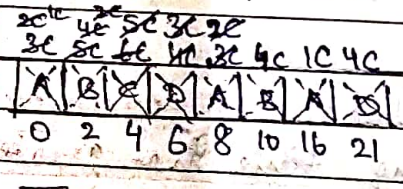
— Shortest Job First (SJF) —





— Shortest Remaining Time First (SRTF).





— The no. of decision pts. in SRTF is greater than in SJF but avg. waiting time is less.

— At each decision pt there will be context switching because scheduler is called even if the same process is continuing operation.

Page No.: study time
Date : / /

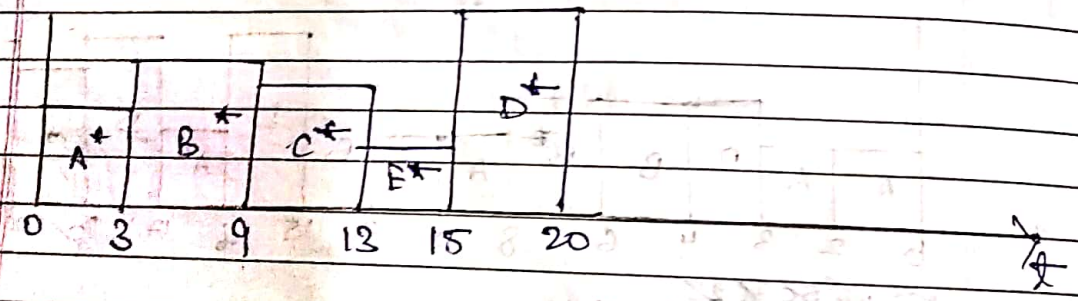
* Highest Response Ratio Next (HRRN) :-

$$\text{Response Ratio} = \frac{\text{Time Spent waiting} + \text{Expected service Time}}{\text{Expected service Time.}}$$

- Ratio increases with waiting of the process.
- so the longer the process is waiting, higher will be it's priority next time.

- Non-preemptive type.

	Arr.	Exe.	Response Ratio
A	0	3	At t=0 ✓ RR=1
B	2	6	At t=3 ✓ RR = 1+6/6
C	4	4	At t=9 ⇒ 5+4/4 ✓
D	6	5	⇒ 3+5/5 At t=13 7+5/5
E	8	2	⇒ 1+2/2 5+2/2 ✓



- E waited less than D because if execution time is less than than waiting time then greater impact on priority.
- so this algo. gives priority to process with smaller CPU burst in blw I/O.
- When a process goes to I/o and comes back

15/10/19

(CHRRN) :-

+ Expected Service Time
d service Time.

of the process.
waiting, higher
me. - 0

io

~~A1 2-13~~
7+5/5
5+2/2 ✓

2/2

cution time is
greater impact

with small

comes back

thus waiting time is calculated from the time of it's I/O arrival and not from the start.

10/19 Priority Scheduling :-

- Starvation in both preemptive and non-preemptive.
- Ageing - As the waiting time increases, the priority of the process increases.
- Solution to starvation.
- LFU :- Freq - Count incremented each time the prog. is used.
- Multiple priority queue - each queue representing 1 priority.
- In Linux - Priority = 140 queue total,
  - 0 to 139 — lowest
  - Highest Priority Priority
- If a queue no. n doesn't have any process then it checks in (n+1)th queue.

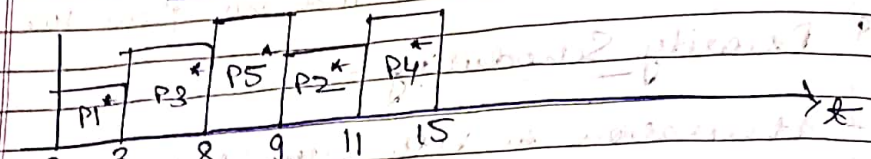
Eg :-

	Arr.	Ex.	Priority
P1	0	3	5
P2	2	2	3
P3	3	5	2
P4	4	4	4
P5	6	1	1

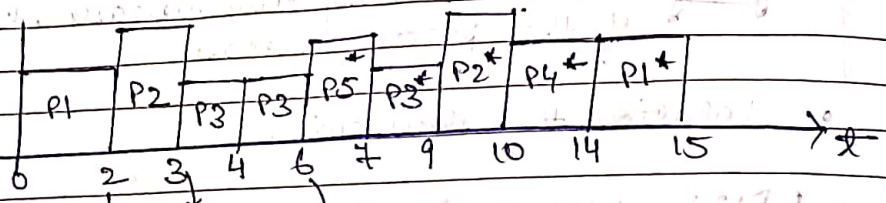
- Lower the no higher the priority.

study time
Page No.:
Date : / /

- Non-Preemptive - No preemption, only completion



- Preemptive :-

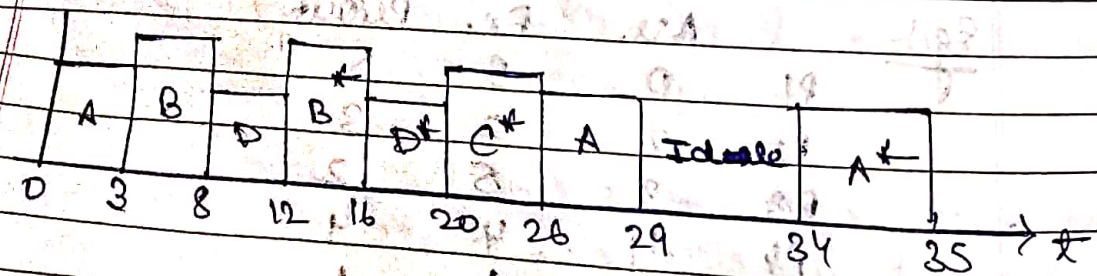


Eg:-

	Arr.	Ex.	Priority
A	0	7 = 3C+5I+3C+5I+1C	3
B	2	9 = 5C+2I+4C	1
C	4	6 = 6C	2
D	6	8 = 4C+1I+4C	0

Exe time → 3C 5I 6C 4C 3C 4C 4C 1C
Process queue → A B C D A B D A
Arrival time → 0 2 4 6 8 10 13 34

- Non Preemptive



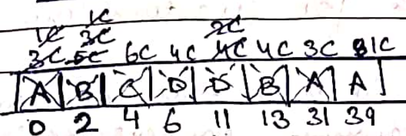
CPU Utilization = $\frac{30}{35} = \frac{6}{7}$

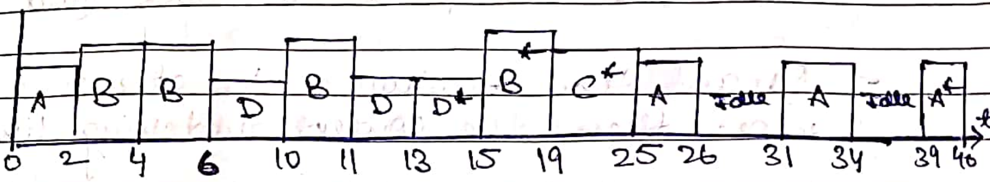
Page No.:
Date: / /

…n, only completion

Priority
3
1
2
0

— Preemptive —





CPU utilization = ~~40~~ $\frac{30}{40}$ = 0.75

- No. of decision pts. in ~~no~~ preemptive is always greater than in non-preemptive.

Round Robin (RR):-

- Usually 10–100 ms.
- No starvation.

— For n processes in ready queue and time quantum q. Max. response time for any process is then (n−1)q time units.

- At every decision pt (i.e. process completion or time slice expiry or I/O) the scheduler is called to check for a change in process (even if there is only 1 process/job in the queue).
- RR gaurantees CPU protection because after every quantum time the CPU calls the scheduler which is a kernel process.

study time
Page No.:
Date : / /

- CPU protection – timer – RR uses the time to check for quantum time expiry.

– If quantum time is more, no. of decision pt. is less so less context switch and more less non-productive work (and vice versa)

– Quant time should be atleast 1000 times more than the context switching time.

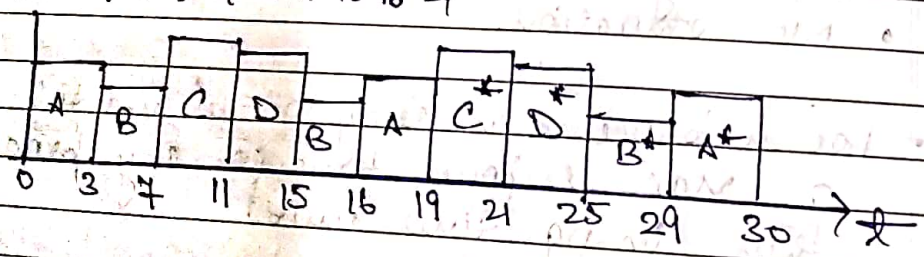
Eg	Arr.	Ex.	SJ
A	0	7 = 3C+5I+2I+1C	
B	2	9 = 5C+2I+4C	
C	4	6 = 6C	
D	6	8 = 4C+1I+4C	

Quantum time = 4.

3C 5C 6C 4C 3C 1C 3C 2C 4C 4C 1C

A B C D A B A C D B A

0 2 4 6 8 7 8 11 16 18 24



– Preemptive schedule – Not with arrival but with time slice expiry.

– Decision pt. are completion/termination, I/O or time slice expiry (No new process arrival).

– Disadvantage – If a process is going out for

I/O or event) before time slice expiry (like A) then it loses its remaining time and has to join back the queue.

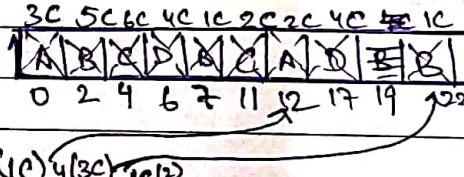
Virtual Round Robin (VRR) :-

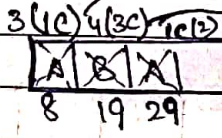
- **Auxillary Queue** — To overcome the above problem.

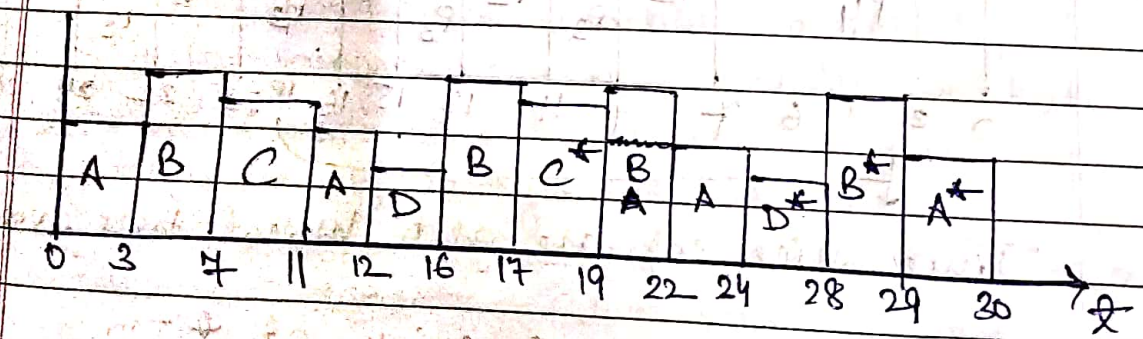
- when a process (like A) comes back, it joins aux. queue and not main queue with remaining time quantum (and not the complete quan. time).

- Aux. queue has higher priority than main queue, so if there is any process in aux. queue then it will be executed first.

Eg:- Same Question with aux. queue.







- Gives advantage to I/O bound processes or user interactive processes.
- Job/process arrival is not a decision pt.

Page No.: study time
Date: / /

16/10/19

* Priority scheduling with Round Robin :-

- Used in linux.
- Same priority :- round robin.

Eg:-

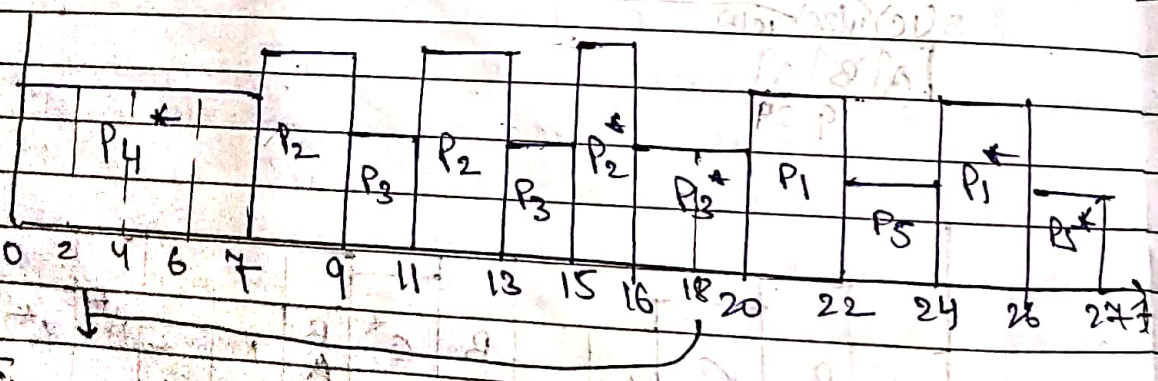
	Burst time	Priority
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3

Quantum time = 2 sec.

Priority 1 Qu → P_4 (7C) — 0

Priority 2 → P_2 (5C) P_3 (8C) P_2 (3C) P_3 (6C) P_2 (1C) P_3 (4C)
0 0 9 11 13 15

Priority 3 → P_1 (4C) P_5 (3C) P_1 (2C) P_5 (1C)
0 0 22 24



Time slice exp. are also present here as decision pt.

Eg:- P_0 with priority 0 arrives at $t = 12$.

- If it is preemptive scheduling then arrival

d Robin :-

P1 P5

26 27

s decision pt.

arrival is

a decision pt so CPU is given to P_5 because it has highest priority.

- In linux, if a process is preempted in b/w (before time slice expiry) then that process (here P_2) will join at the front of its priority queue with only the remaining quantum time and not the fresh time.

- In this case decision pt is all the 4 cases -
  - Time slice expiry due to RR of each queue.
  - Rest three decision pt. due to priority scheduling scheme (i.e. I/O, arrival of new process, departure/termination of current process).

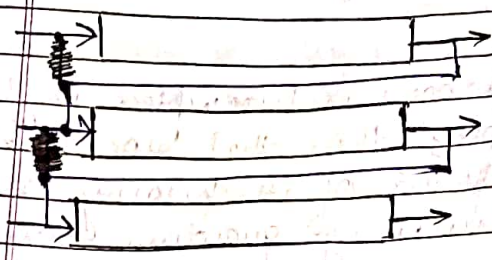
- If each queue is following FIFO and not RR, then time slice expiry is not a decision pt., but the rest three decision pt. are present due to priority scheduling.
  - FIFO = RR with ∞ time slice.

★ Multilevel Queue :-

- More than 1 queue available for diff. types of process - 1 for foreground and other for background.
- Starvation of background process.
  - To overcome this - time sharing b/w the foregnd, backgnd ~~processes~~ queue.
- Sys Process - Scheduler, ~~Priority~~ dispatcher.

Page No.: study time
Date : / /

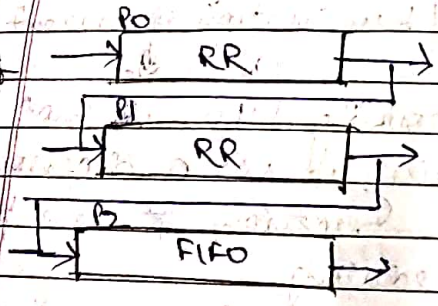
Multi level Feedback Scheduler :-



- Process can join in any of the levels.
- Process can be promoted / demoted to any of levels (i.e. both increasing the priority and decreasing as well)

- Scheduling alg. can be diff. for diff. queue (i.e. one can have RR, other FIFO, etc.).

Eg :-



- After 8 time unit, processes demote from priority 0 to priority 1 queue and then after 16 time unit it is moved to third priority level.

- So for interactive / I/O bound process (they usually have smaller CPU time) they are always in P0 queue because they have small CPU so they complete execution before 8 unit (usually).

Priority Calculation:-

$$CPU_j(i) = \frac{CPU_j(i-1)}{2}$$

$$GCPU_k(i) = \frac{GCPU_k(i-1)}{2}$$

$$P_j(i) = Base_j + \frac{CPU_j(i)}{2} + \frac{GCPU_k(i)}{4 \times W_k}$$

interval i

CPU_j – Measure of processor utilization by process j through

$GCPU_j$ – " " " " " of group k " "

$P_j(i)$ – Priority of process j at the beg. of interval i; lower value means higher priority.

$Base_j$ – Base priority of process j.

W_k – weighting value assigned to group k.

- Group count – CPU may be grouped segregated in various groups.
  - Any process running on a CPU increases it's process count as well as Group count.
  - Group count is common for all process of a group.
- $0 < W_k \leq 1$ and $\sum_k W_k = 1$
- Base priority of process remains same.

Eg: Process A – Group 1 – $W_1 = 0.5$; $W_2 = 0.5$
Process B – Group 2
" C – Group 2

– If B is executing then group count of C also increases and vice versa.

time	PA			PB			PC		
	Base	CPU	GCPU	Base	CPU	GCPU	Base	CPU	GCPU
	60	0	0	60	0	0	60	0	0
	Commit cycle of PA here								
		60	60		0	0		0	0
	90	30	30	60	0	0	60	0	0
								1	1
		30	30		60	60		0	60
	74	15	15	90	30	30	75	0	30

* Fair Share Scheduling :–

– If a group has more processes than some other group, then first group has lesser priority than the second group.

18/10/19 Eg: PA – Group 1 – $W_1 = 0.8$, $W_2 = 0.2$ *
PB – " 2 $Base_1 = 30$
PC – " 3 $Base_2 = 40$
$Base_3 = 50$.

Time Quantum A = 50
" " B = 40
" " C = 30

Base
30
✓
49
39
✓
5
✓
A

PA			PB			PC		
Base	CPU	GCPU	Base	CPU	GCPU	Base	CPU	GCPU
30	0	0	40	0	0	50	0	0
✓	↓	↓		↓	↓		↓	↓
	50	50		0	0		0	0
49	25	25	40	0	0	50	0	0
	↓	↓	✓	↓	↓		↓	↓
	25	25		40	40		0	40
39	12	12	75	20	20	75	0	20
✓	↓	↓		↓	↓		↓	↓
	62	62		20	20		0	20
54	31	31	57	10	10	62	0	10
✓								

★ Traditional UNIX Scheduling :-

$$CPU_j(i) = \frac{CPU_j(i-1)}{2}$$

$$P_j(i) = Base_j + \frac{CPU(i)}{2} + nice_j$$

- Nice value = −20 to 20
  — lower being higher priority.

★ Linux Scheduling :-

- Interactive Performance — Process with higher I/O should get more chance.
- Good wakeup Perf — New process should be admitted with not much of a delay.

study time
Page No.:
Date: / /

- Fairness – No starvation – Aur. queue concept (Pending queue).
  - Exception – Real time process
    - All real time processes are critical process so they will not wait for fairness and execute as soon as possible.
    - Conventional / user created process follow fairness.

- Real Time Process – Priority – 0 – 99 (100)
- Conventional / user – " – 100 – 139. (40).

* SMP – Symmetric multiple Processor.

i). Freq. Sym – same operating freq. of diff. processor.
Eg – Intel atom.

ii). ISA Sym – same ISA of diff. process
- A job can be transferred from one core / CPU to other core/CPU if they supp use same ISA.

- SMP Efficiency – If any core / CPU is idle then processes / load can be should be shifted to that core/CPU from a heavily loaded core / CPU.

- SMP affinity – Processes can set what all cores it can execute – Hard affine.

* Soft
– most
* Syste
• wi
• re
• sc
Eg
* Lin
– Sc
Sc
Sc
* Sc
–

Aux. queue concept.
Pending queue).
ess
are critical
not wait for
soon as possible.

process follows

0 - 99 (100)
100 - 139. (40).

ocesses.

freq. of diff-

diff. process

m one core/
support use

is idle then
should
from a

t. what all
- Hard
affine.

• Soft affine - Process can shift from one core to other.

- Most processes are soft affine.

* System Calls :-

• nice() - Check nice value.
• renice() - Change nice value.
• sched_yield() - Leave the CPU by itself.
- Nobody is forcing that process.
- Join the ready queue.

Eg :- Producer - consumer problem.

* Linux Schedulers -

- SCHED_RR } Real-time
SCHED_FIFO }
SCHED_OTHERS → Non-real / Conventional.

* SCHED_FIFO - When a process moves to ready queue it will join at the start of the queue.

- So if there are no real-time process with higher priority than the current process is available, then the same process will continue until it finishes (or higher priority job comes). There will be decision pt. but same process will continue in CPU.

- If a higher priority process (Real-time) joins then CPU is given to this new process.

- Joining of non-real time process does not matter because they are always lower priority than real time process.

- Linux 2.4 — Non-real time
  Linux 2.6 — Real-time
  - waiting time in admitting new process is decreased

* SCHED_RR - when a process moves to ready queue then it will join at the back of the queue.

- Within same priority number/level, some processes may be FIFO and other may be RR.
- The only diff is the place where they join in ready queue, other things remain same

21/10/19 Conventional Scheduling - Preemptive priority

- Two diff types - i) Static Priority
  ii) Dynamic P.
- Short time sched. is based on dynamic priority.
- Dynamic priority is based on static priority
- Static " " " " nice value
- Nice value is entered by user - ~~-20 to +19~~
  -20 to +19.

- Static priority is always b/w 100 to 139 irrespective of whether it is real time or conventional process because it is calculated from nice value.

- Static Pr. No. = 120 + Nice value
  (100 - 139) (-20 - +19)

- Quantum time of a process is based on the static priority no. of that process - higher priority (or less no) will get more quan. time.

- A new process will inherit the nice value/static priority from parent which can be changed later using nice() sys. call.

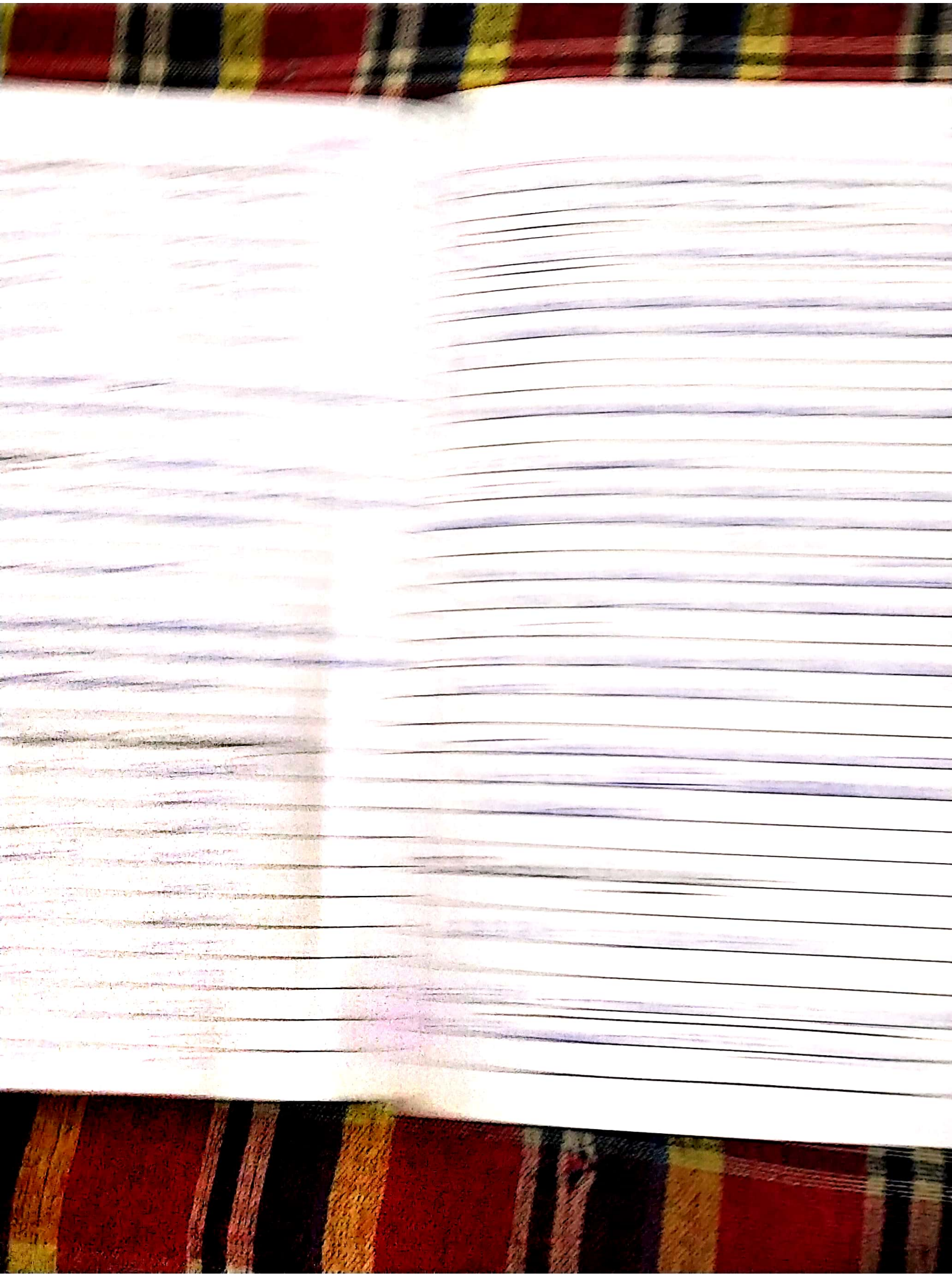
→ Static. Pr. No. = MAX_RT_PRIO + (Nice) + 20.
(100)

- -ve nice value are for system process or more imp. processes.
- +ve nice value are usually user level process.

- For priority 120 and above -
  Quan. time = (140 - St. Pr.) * 5 ms.

- For priority below 120 -
  Quan. time = (140 - St. Pr.) * 20 ms.

Scanned by CamScanner



- Dy. Pr should always lie in the range of 100 to 139, so —

$$\text{Dy. Pr.} = \max(100, \min(\text{st.pr} - \text{bonus} + 5, 139))$$

- when the process starts for first time, bonus is taken as a default value which usually is 0.
- Lower dynamic priority no. has higher priority.
- $0 \leq \text{Bonus} \leq 10$.
- If bonus is less than 5 then dy. priority no. ~~reduces~~ increases from static priority no. and vice versa.
- Avg. sleep time reduces when process is running in CPU and increases when process is in I/O.
- Avg. sleep time does not change when process is in ready queue.
- Interactive process may get some advantage but batch process does not get any advantage.

* Interactive Process —

$$\text{D.P} <= \frac{3}{4}\text{S.P} + 28$$

$$\Rightarrow \text{S.P} - \text{bonus} + 5 <= \frac{3}{4}\text{SP} + 28$$

$$\Rightarrow \boxed{\text{Bonus} - 5 >= \frac{\text{SP}}{4} - 28}$$

- Since avg. sleep time changes after each CPU execution so bonus value, dynamic priority and thus quantum time may change for next execution.

- $\boxed{\frac{S.P.}{4} - 28 = \text{Interactive delta}}$

$\Rightarrow -3 \leq I.\delta \leq 6.$

∴ A process is called interactive process based on both its avg. sleep time (Bonus value) and it's static priority no.

- A process with ~~sta~~ lesser static priority can be int. process with less bonus value.

Eg:- • SP = 100.

$\text{Bonus} - 5 \geq = \frac{100}{4} - 28$

$\text{Bonus} - 5 \geq = -3 \Rightarrow \text{Bonus} \geq = 2$

∴ Avg sleep time should be greater than 200ms.

- SP = 120.

$\text{Bonus} - 5 \geq = \frac{120}{4} - 28$

$\text{Bonus} - 5 \geq = 30 - 28$

$\text{Bonus} \geq = 7$

∴ Avg sleep time $\geq = 700$ ms.

- SP = 139.

$\text{Bonus} - 5 \geq = \frac{139}{4} - 28$

$\text{Bonus} - 5 \geq = 34.75 - 28$

$\text{Bonus} \geq = 6.75 + 5 = 11.75$

— Not possible.

∴ Process with SP = 139 can never become int.

21/10/19

each CPU execution and thus quantum.

Page No.:
Date: / /

eeta

eive process based (Bonus value)

tic priority can us value.

=2

ter than 200ms.

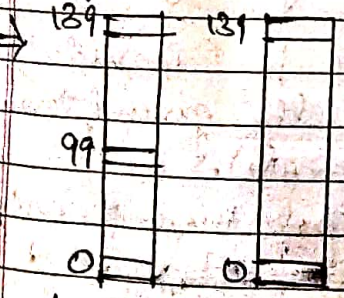
ver become

Page No.:
Date: / /

Interactive Process —

- To avoid starvation in linux, there are two queue — Active Queue & Expired Queue.
- Any process after completing it's allocated quantime time will go to expired queue at the same priority level, acc. to it's next priority value (Because dy. bo. may/may not change).
- The process with highest priority will be selected from active queue to execute next (Not from expired).
- So a process will get only one chance to execute in 1 cycle.
- Exception — If a process is interactive process then it may join the active queue back instead of expired queue.

11/10/19 No real process will ever expire i.e. it will never go to expired queue and will always finish from active queue. New real process will join active queue.



- After the active queue is becomes empty the expired queue will become active queue and active queue becomes expired queue.
- Only the pointer needs to be exchanged.

- Interactive process will not join the active queue again if -
  - The oldest expired process has already waited for long time.
  - An expired process has higher static priority (lower no.) than the interactive process.
- Other than these two condition the int. process will join back the active queue always.
  - Every batch process (CPU bound usually) will join the exp. queue after time slice expire.
  - Any new process or I/O / suspended process will always join the active queue.
    Every time slice expired process will go to expired.

★ Real time process scheduling - Preemptive priority
- Time quantum is always decided from static priority for both types of process.
- Real time process dynamic pr. is not calculated from static pr. but directly given (Parameter passed in SET_SCHED_PARAM).
  - So dynamic priority of real time priority does not change.
  - If during a running process a new process with higher priority comes then the current

process has to [illegible] and give CPU to new process. [illegible] the execution of new process, the [illegible] will continue execution but only for the remaining time quantum.

- [illegible] the process of 1 priority level finishes execution [illegible] lower priority [illegible].

- [illegible] is [illegible] to real-time processes also but there is no mechanism to stop that [illegible] process.

- [illegible] yield() – sys. call to give up CPU by a running process voluntarily.

  – The process will execute [illegible] next time with the remaining time quantum only.

- Any [illegible] from outside (i.e. new admission or [illegible] suspended state) will join at the [illegible] of [illegible] queue.

  – The process which is already in the CPU ([illegible] running) will have option to join [illegible] (fully or [illegible]).

- At every [illegible] the dynamic priority of [illegible] changes but quantum time is changed only when the present quantum time is [illegible].

• Queues – Each queue is a circular doubly linked list.
– Each head is a dummy node which has the no. of nodes attached to it (no. of process in that priority).

• Bitmap – The lowest priority queue at which a process is available is set 1, other bits are 0.

– O(1) scheduler.

• recalc_task_prio() – updates the dyna. pr. and avg sleep time for the conventional process coming back from I/O.

• Direct Sched. – Current process blocked or finishing execution then scheduler is called.
– Non preemptive decision pts. & are direct calling to sched.

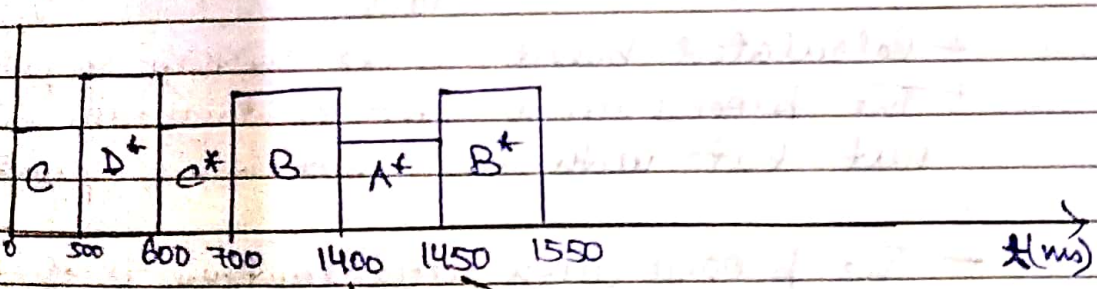
• Lazy Invocation – By setting the TIF_NEED_RESCHED flag to 1.
– When a process currently executing switches from kernel space to user space.

– Preemptive dec. pt. are lazy invocation.
(Time slice expiry: done by scheduler_tick() function, higher priority process came at ready queue: done by try_to_wake_up() function).

- When a sched_setscheduler() system call is issued — A process may get higher priority than the currently running process.

Eg:- Process	Nice	Ex. time (ms)	Sleep time (ms)	Real time Priority	Sched Policy
A	8	50	0	–	SCHED_OTHER
B	-15	800	0	–	SCHED_OTHER
C	-5	600	–	30	SCHED_RR
D	0	100	–	30	SCHED_FIFO

Process	S.P.	Q.t (ms)	DP.
A	128	60	133
B	105	700	110
C	115	500	30
D	120	100	30



- Assuming A, B, C, D came in ready queue in that order.

B moves to exp. queue - It can not become ind. process because avg. sleep time is 0 and CPU ex. is taking place so avg. sleep time remains 0.

Now the active queue becomes empty so expired queue becomes active and B executes because it is the only available process there.

Page No.: study time
Date : / /

22/10/19

★ Completely Fair Scheduler :-

- Complete Fair Share - For CPU sched.
- Complete Fair Queue - For I/O (disk) sched.
  - Red ~~Blog tree~~ Black tree is used instead of queue.
  - Picks the process that has used the least amt of time.
  - If process which spends a lot of it's time sleeping then it's spent time value will be low so it will be selected for execution with low CPU time.
  - Order of complexity is $\log(N)$ for insertion / deletion etc.
  - Min. allocated time - 1 msec.
- Virtual Runtime - Time provided to a given task.
  - Calculated based on the history of process.
  - The process with least runtime will be executed next but with the least time quantum.
  - The process with least runtime is at the left of tree.
  - Complexity of picking the next process to run is $O(1)$ because while adding / deleting a node the left most node will be saved in the cache.
  - Self Balancing - No path will be twice as long as any other node so max complexity can be $2\log(N)$.

- Abstract arg. is used to initialise a new task.
- Group sched. is used to decide virtual run time.

* Time slice —

$$\text{timeslice} = \text{sched\_period} * \left(\frac{\text{task's weight}}{\text{total weight of tasks in run queue}} \right)$$

- Task's wt depends on the task's nice value.
- Min wt = 15 corresponding to nice value 19.
- Max. wt = 88761 corresponding to nice value -20.
- Weight is the granularity of that nice value.
- Time slice becomes smaller as the load increases.
- Granularity of a particular nice value is not a fixed value but a range of value out of which 1 is given to a process dep. on the type of process.

* Multi Processor Scheduling —

- Single cycle data path — IF, ID, EX, MEMR, MEMW will all happen in 1 clock cycle.
  - Cycle width is big.
- Multi cycle data path — Each process will happen in 1 cycle each.
  - Cycle width is small.

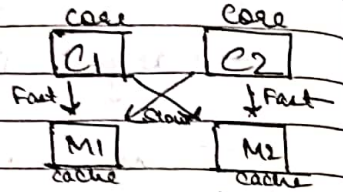
- Each cycle uses the o/p of previous cycle so buffer/latch is needed to store that o/p.

- Pipelining - Multiple inst. can be execute parallely at diff stage (one at IF stage, other at ID, other at EX and so on at the same time.

- Intel i7 20 stage pipelined.

- Super Scalar - Multiple pipeline.

- NUMA - Non uniform memory access.



- Core 1 can access M2 & core 2 can access M1 but it may do so with slower speed than if the cores accessed their corresponding cache memory.

* Symmetric Multiprocessing (SMP) - Multi core MP.

- Two mechanism - i). 1 Global queue for all the cores ii). Diff. queues for diff. cores.

- Global Queue - Not possible.

- If a process running in core 1 goes to I/O and comes back, it should come back at in core 1 only because all cache data are there in cache of core 1. If it comes at core 2 then data has to be moved from cache 1 to cache 2.

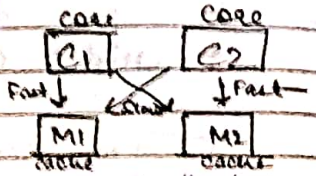
o/p of previous cycle
et to store that o/p.

can use execute
stage (one at a
ue at EX and so

d.

eline for

y access.



accessed their

)— Multicore

HP.

queue for all
diff. cores.

se I goes to I/o
come back at
cache data
. If it a comes
be moved from

- Linux uses local queue for core.

- cache I to each

- Local Queue -

- If multiple core req. a job at the same time then it may happen that multiple cores may pick the same job. To avoid this if one core is accessing the queue to pick up the & process them the queue has to locked so that other cores cannot use it at that time.

- Local Queue - load sharing is req. explicitly.
- In global queue load sharing is not req. (Done automatically).

+ Memory Stall Cycle - If a process needs a data which is not in L1 cache then it has to wait in CPU till data comes from L2 cache. If again L2 cache miss then it has to fetch from primary memory and process has to wait in CPU.
- Till that time CPU will execute 'nop' so no productive work is done that time. (Reg_write = 0, Mem_write = 0).

- So getting data from I/O is better because that process gets blocked and other process can execute in CPU.

+ Multi Threading - Each core has > 1 h/w thread.
- Hyperthreading in intel.
- Diff. from single thread / os threads.

- If in a core 1 thread needs data from mem. then core switches to other thread for execution until the data is fetched for 1st thread.

[illegible]

- The local [illegible] of a process [illegible] to the [illegible], there [illegible] all [illegible] same core because [illegible] of that core [illegible] not [illegible].

- Processor affinity – although maintained by [illegible] core process to [illegible] which core/processor [illegible] is executed in.

- Hard affinity – Migration not allowed.
  – Remain only 1 core.

- Soft affinity – Migration allowed.
  – Can execute on other [illegible] for sake of load balancing is required.

- Affinity of a process can be set by user.

  Eg: Pass 1 & Pass 2 during compilation in the same core / [illegible].

- NUMA – [illegible] in Intel.
  – Any core can access the cache of any core but the time taken for a core to access other core's cache will be more than if it access it's own cache.

→ So if a core wants data which is not present in its cache but present in some other [illegible]

23/10/19

- In local queue bee core if a process goes to I/O, then it will come back to the same core because all it's data is in the cache of that core. So cache flush does not happen.

* Processor Affinity – Bitmap maintained by each process to decide which core/processor can it execute in.

- Hard affinity – Migration not allowed.
  – Execution only 1 core.
- Soft affinity – Migration allowed.
  – Can execute in other pe core also if load balancing is required.

– Affinity of a process can be set by user.

→ Eg:- Pass 1 & Pass 2 during compilation in the same core / µP.

* NUMA :- Quick path in Intel.
  – Any core can access the cache of any core but the time taken for a core to access other core's cache will be more than if it access it's own cache.

– So if a core needs data which is not present in it's cache but present in some one else's

study time
Page No.:
Date: / /

if a process goes
value back to the
it's data is in the
cache flush does

b maintained by
ocess to decide
n it execute in.

not allowed.
y 1 core.
llowed.
e in other pe core
balancing is

et by user.
compilation in the

s the cache of
time taken for
re's cache will
n it's own

not
ch is present
some one else's

study time
Page No.:
Date: / /

cache then it that core can take that data directly without copying that data to it's own cache first which speeds up performance.

* NORMA — Non remote memory access.
— Core cannot access cache of other core.

- Multicore — shared memory.
- Distributed sys. — No shareble memory.

* Load Migration —

- Pull Migration — If a core is free, it can take the job of any other busy core to execute.

- Push Migration — If a core is very busy, it can push it's job to other core less busy cores of the system.

- Octa Core — Two groups of 4 core.
— Try balancing within the the group first (b/w 4 cores of that grp).
— If groups are unbalanced then transfer job from a group to other

- In linux, push migration algo. will run every 200 ms.
— Pull mig. will happen automatically when a core becomes free/idle.

* Windows
- Push migration algo. uses past history of the CPU to determine which is busy and what is free/idle.

* Windows CPU Sched. :-

- Priority: 0 - Mem. Management
  1-15 - Variable
  16-31 - Real time.

- One queue for each priority level.
- Higher no. - higher priority.
- High priority to int. process - mouse/window.
- Variable class :- HIGH_PRIORITY_CLASS
  ABOVE_NORMAL
  NORMAL
  BELOW_NORMAL
  LOWEST
  IDLE

- Three classes present in both Real-time Range and Variable Range.
- From windows 7 - Two levels of thread sch.
  - User level thread scheduling
  - Kernel level thread scheduling.