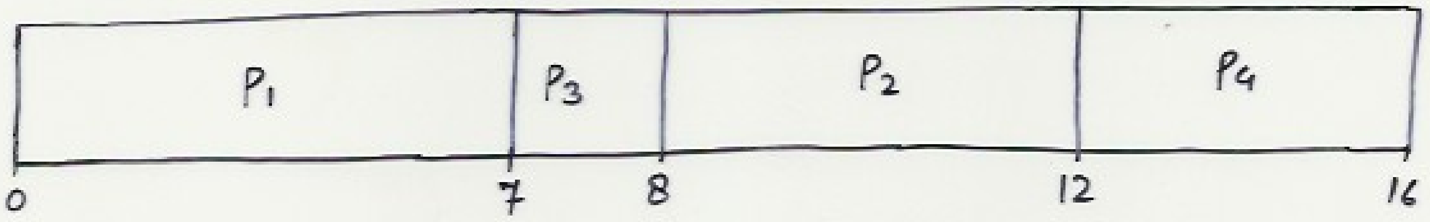
Non - preemptive SJF :

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

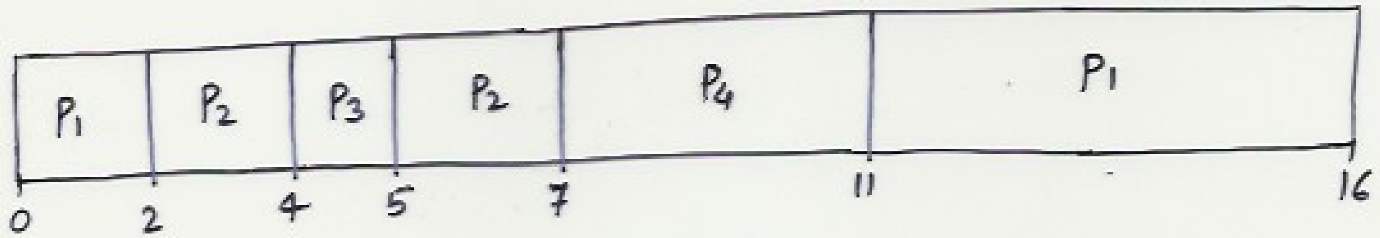
SJF (non preemptive)



average waiting time = $(0+6+3+7)/4$

$= 4$

Preemptive SJF :



average waiting time : $(9+1+0+2)/4$

$= 3$

→ Drawback -

- Length of the next CPU burst is required.

→ One approach is to approximate SJF, where next CPU burst is estimated.

→ t_n : length of the nth CPU burst.

τ_n : predicted length of the nth CPU burst.

τ_{n+1} : " " " " $(n+1)^{th}$ " " .

α, $0 \leq \alpha \leq 1$.

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

Exponential Averaging :

- if $\alpha = 0$,

  from eqn (1), $\tau_{n+1} = \tau_n$

  i.e. Recent history does not count.

- if $\alpha = 1$, $\tau_{n+1} = t_n$

  i.e. only actual last CPU burst counts.

- Eqn (1) can be :

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\alpha t_{n-1} + \cdots\cdots (1-\alpha)^j \alpha t_{n-j} + \cdots$$
$$\cdots\cdots\cdots\cdots (1-\alpha)^{n+1} \tau_0$$

- Since α & $(1-\alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor.

Priority Scheduling

→ A priority no. (integer) is associated with each process.

→ A CPU is allocated to the process with the highest priority. (smaller integer = highest priority).

→ Scheduling can be
  - preemptive
  - non preemptive.

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

Arrival time of all processes is 0.

Using priority scheduling

P_2	P_5	P_1	P_3	P_4

0   1   6   16   18   19

average waiting time = 8.2

→ SJF is a priority scheduling where priority is based on the predicted next CPU time.

→ Drawback -

- * Algorithm leads to <u>starvation</u>.
- * where a low priority process may never execute.

→ Solution to starvation is <u>aging</u>.

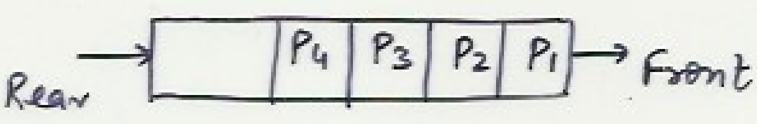
→ As time progresses, increase the priority of the process.

Round Robin (RR) Scheduling

- → Each process gets a small unit of CPU time, called <u>time quantum</u>.
- → After time quantum, the process is preempted & added to the end of the ready queue.
- → If there are n processes in the ready queue & the time quantum is q, no process waits more than $(n-1)q$ time units.
- → Example of RR with time quantum = 20

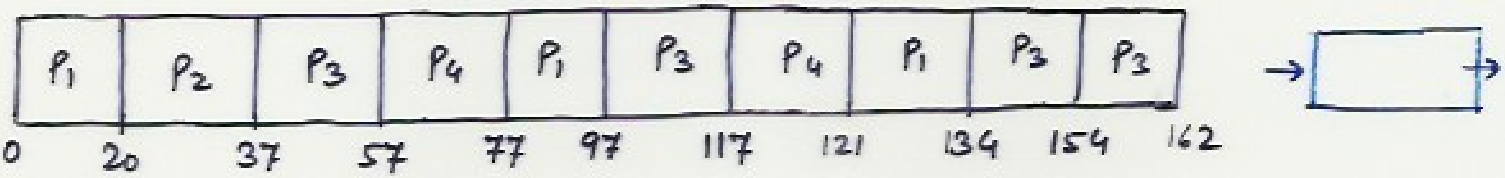
Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

Initially ready queue



~~After~~ During 1st time quantum

~~After~~ During 2nd time quantum



average waiting time: $(81 + 20 + 94 + 97)/4$

$= 73$

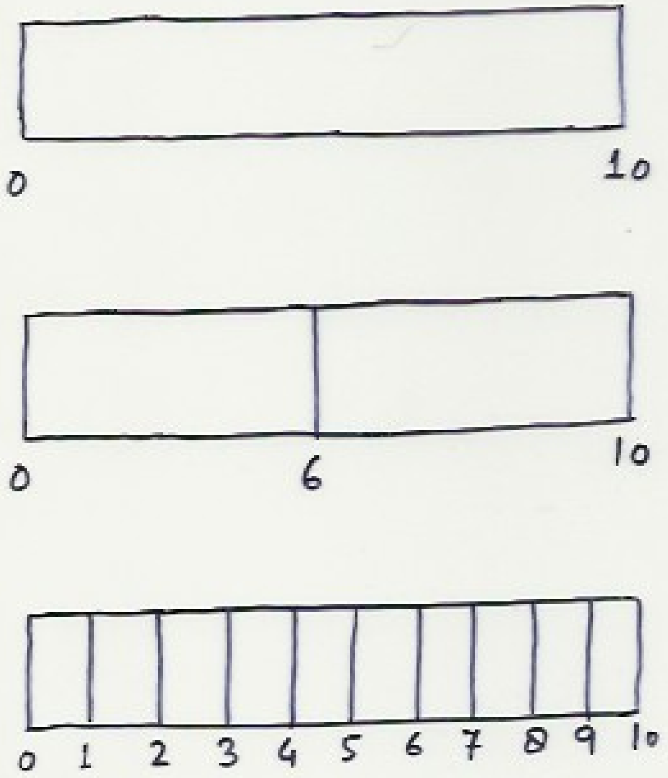
→ Typically higher avg. waiting time than SJF, but better response.

Performance with time Quantum :

- Performance of RR scheduling depends on time quantum size.
- If time quantum is large -

  scheduling degenerates to FCFS.
- If time quantum is small -
  - * more context switching.
  - * Overhead is too high.

Process Time = 10



Quantum	No. of context switch
12	0
6	1
1	9

→ A rule of thumb is that 80% of the CPU bursts should be smaller than time quantum.

Multilevel Queue Scheduling:

→ Ready queue is partitioned into separate queues:

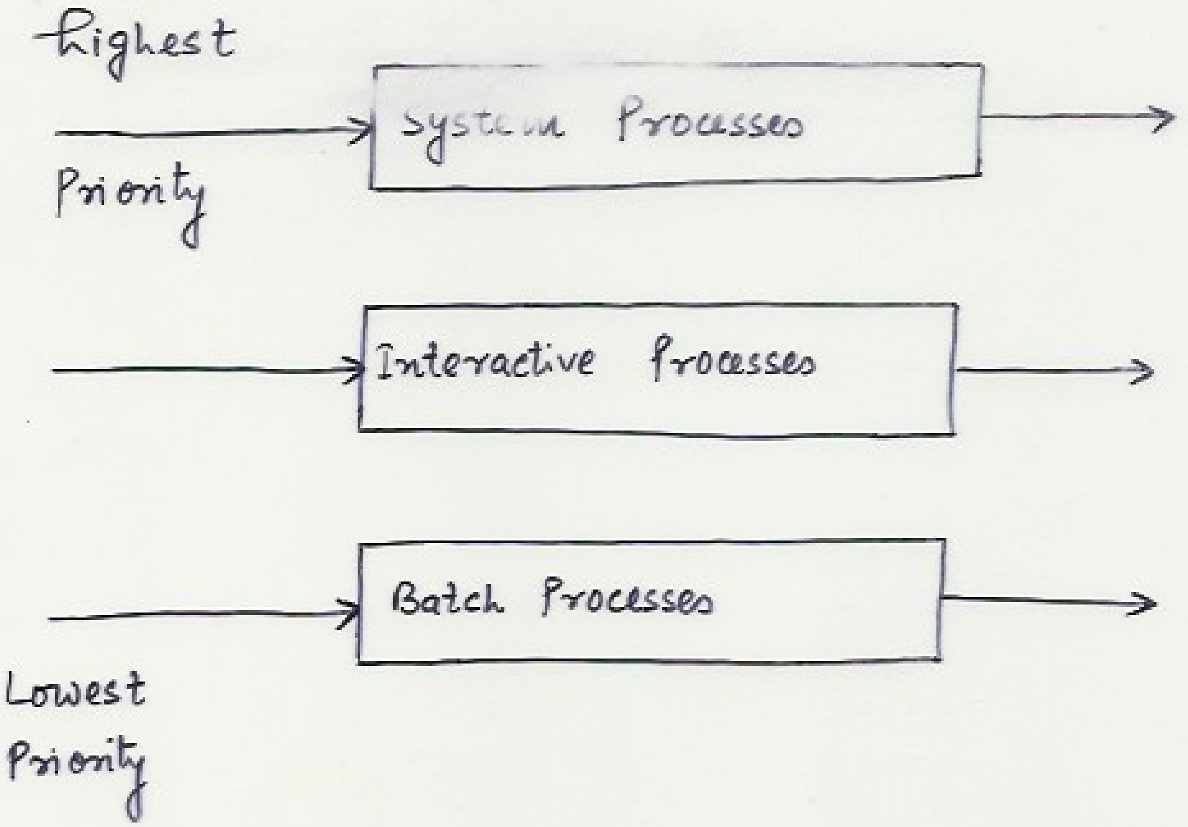
- foreground (interactive)
- background (batch)

→ Each queue has its own scheduling algorithm:

- foreground (RR)
- background (FCFS)

→ Scheduling must be done between the queues.

- Fixed Priority Scheduling - serve all processes from foreground then from background.
  - Possibility of starvation.
- Use Time Slice - Each queue is given a certain amount of CPU time which it can schedule amongst its processes.
  - 80% may be given to foreground (RR)
  - 20% " " " " background (FCFS)



Multilevel Feedback Queue:

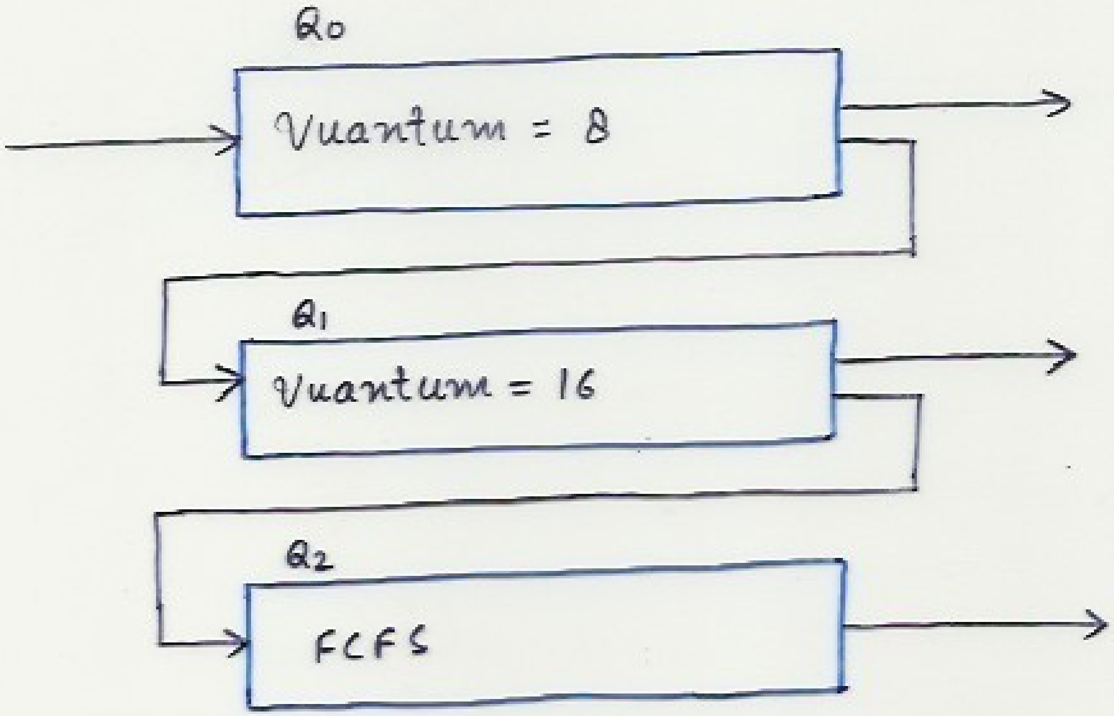
→ A process can move between various queues.

→ Aging can be implemented this way.

→ Multilevel feedback scheduler is defined by the following parameters:

- Number of queues.
- Scheduling algorithm for each queue.
- Method used to upgrade a process.
- Method used to degrade a process.

- Method used to determine which queue a process will enter when that process needs service.



→ Three queues:

- Q_0 - RR with time quantum = 8 msec.
- Q_1 - " " " " " 16 "
- Q_2 - FCFS

→ Scheduling -

- A new job enters queue Q_0 & served as FCFS.
- When it gains CPU, recieves 8 msec. of CPU.
- If it does not finish in 8 msec, job moves to queue Q_1.

- At Q_1, job is served FCFS and recieves 16 additional msec. If it still does not complete, it is preempted & moved to queue Q_2.

→ It is most general scheduling but also most complex

Multiprocessor Scheduling

→ So far, uniprocessor scheduling concept is used.

→ Multiple-processor scheduling is more complex.

→ Assumption is homogeneous processors within multiprocessor.

→ Load sharing can be done.

- Separate queue for each processor is provided.
- drawback is - One queue may be full, while other is empty.
- A common queue may be used.

→ Using common queue - two scheduling approaches may be used.

- In 1st approach - each processor is self scheduling.
- Drawback is - a process may be selected by two or more processes.
- Assymetric multiprocessing can be used to avoid it.
  - A processor is responsible for selecting a process from common queue.
  - Other processors execute these processes.

Process Synchronization

→ Concurrent access to shared data may result in data inconsistency.

→ Maintaining data consistency requires mechanism to ensure the orderly execution of cooperating processes.

→ Let us consider Producer- Consumer processes once again.

- A variable count may be used to access shared buffer by Producer/ Consumer processes.
- Initially count = 0.
- It is incremented by producer, after it produces a new item.
- It is decremented by consumer, after it consumes a buffer.

Producer :

/* produces an item & put in nextproduct */

```
while (true)
   {
      while ( count == BUFFER_SIZE)
                ; /* do nothing */

      buffer [in] = nextProduct;
      in = (in +1) % BUFFER_SIZE;
      count ++;
   }
```

Consumer:

```
while (1)
   {
      while ( count == 0)
            ; /* do nothing */
      nextConsumed = buffer [out]; /* Consume
                                      in nextconsu
      out = (out +1) % BUFFER_SIZE;
      count--;
   }
```

Race Condition :

→ Though producer & consumer process code are correct separately but they may not function correctly when executed concurrently.

→ Count ++ can be implemented in machine language as:

```
register1 = count;
register1 = register1 + 1;
count = register1;
```

→ count-- can be implemented in machine language as:

```
register2 = count;
register2 = register2 - 1;
count = register2;
```

→ let us assume count = 5, initially.

→ count ++ & count-- may be interleaved as follows:

T0 : producer execute register1 = count { register1 = 5 }

T1 : producer execute register1 = register1 + 1 { register1 = 6 }

T2 : consumer execute register2 = count { register2 = 5 }

T3: consumer execute register2 = register2 - 1 { register2 = 4 }

T4: producer execute count = register1 { count = 6 }

T5: consumer execute count = register2 { count = 4 }

→ Incorrect value of count = 4.

→ Count = 6, if we reverse T4 & T5, which is again an incorrect value.

→ A situation like this -

"where several processes access & manipulate the same data concurrently, and the outcome of the execution depends on the particular order in which access takes place, is called a Race Condition."

Critical Section Problem

→ A system consisting of n processes $\{P_0, P_1, \cdots P_n\}$

→ Each process has a segment of code, called a critical section.

→ Important feature is -

"When one process is execuding in its critical section, no other process is to be allowed to execute in its critical section".

→ Each process must request permission to enter its critical section, in its entry section of the code.

→ critical section may be followed by an exit section.

Remaining code is the remaining section.

```
repeat

        [Entry section]

        critical section

        [Exit section]

        remainder section

until false.
```

General structure of process Pi

Shortest job First Scheduling

→ Associate with each process the length of its next CPU burst.

→ Use these lengths to schedule the process with the shortest time.

→ Two schems -

- Nonpreemptive - Once CPU given to the process, it can not be preempted, until completes its CPU bursts.
- Preemptive - If a new process arrives with CPU burst length less than remaining time of current executing process, 1st one is preempted.

  - This scheme is known as Shortest Remaining Time First (SRTF).

→ SJF is optimal - gives minimum avg. waiting time for a given set of processes.