SCHEDULING AND SYNCHRONISATION.

- Context switching policies:
  - Scheduling: (issue of context switching among larger no. of threads)
  - Running ready wait

- I/O wait
  - Blocking system calls

Scheduling problem: Should we transition from one process to another which process from the ready queue should be select.

Example:

```
read()  run->wait
sleep()  run->ready
```
In a pre-emptive system, yield() gets called from the timer interrupt.

* Scheduling Mechanisms:
  (1) **Strict Priority:** Each task in the system is assigned a unique priority. The highest priority process/running task is chosen to be executed.

\[\text{run} \rightarrow \text{wait} \quad \text{run} \rightarrow \text{wait}\]

\[\text{ready} \quad \text{ready}\]

- This kind of scheduling mechanism is really useful in real-time systems.
  - Examples: Assigning priorities based on frequency
    - video streaming etc.

- However, it is not a general purpose solution for simple scheduling between processes.
(2) **Round Robin**: Switch among different processes based on time quantum.

**Processor Scheduling**: All tasks in a queue are worked on simultaneously with each process getting an equal share of the processor.

(round robin → proc. sharing as time slice → 0)

---

**Diagram**:

```
wait [t1] [CPU] wait [t2]
```

```
\[ t_1 \quad \text{inter-arrival} \quad t_2 \]
```

---

**Criteria**:

- **Response Time**: Amount of time from when an I/O request occurred till when it is completed.

  *e.g.*: Web server serving requests.
(ii) Throughput: Scheduling the CPU in such a way that you are able to serve maximum number of processes thus reducing the time you spend on context switching.

(iii) Fairness: Different processes should receive fair amount of CPU.

e.g. Avoiding starvation.

* FCFS Vs SJF (shortest job first)

FCFS → 5 3 1 5 + 8 + 1 = 7

SJF → 1 3 5 1 + 4 + 9 = 4 2/3

Thus, usage of SJF we can get a better average response time.
Multi-level Feedback Queue (MFF)

Traditional scheduling used in UNIX.

Set of round robin queues by priority.

If a process from a particular queue is slow, it is sent to the queue at the next level.

Thus, newer processes can be given a larger time slice. This will avoid frequent switching.

LINUX 2.4 kernels used this scheduler.

Instead of MFF, everytime priority changes, the list/queues will have to be re-ordered or at every instant scheduling decision would have to be used.

This is feasible for comparatively smaller systems.

Thus, LINUX switched to Order-1 Schedules.
ORDER-1 Scheduling:

- Active
- Expired
- Time quantum
- Expired

pret: hi - 0.0.0
- 0.0.0

- Fixed no. of priorities
- Each job gets a time-slice proportional to its priority.
- Time quantum < priority
- Each job remains in the 'active list' until its quantum expires.
- After its time quantum expires, it is moved to 'expired list'.
- Expired tasks become active when there are no more tasks in the active list.
VIRTUAL MACHINE SCHEDULING:

- Switch back and forth between 2 VM's which gives each one fair amount of time, is efficient and responsive.

3) Strict timeslice
   Y N N

   (Consider sending TCP packets between VM1 - VM2 round trip time for packet increase)

2) Yield to other VM on wait
   N Y N

   This is efficient because we reduce wasted CPU.

   VM1  VM2
   ↓  ↓
   ↓  ↓
   ↑  ↑
   ↑  ↑  ↑

   (wait time is reduced.)

   Based on a fixed time slice.

× Vulnerabilities in scheduler: stealing time slices.