Scheduling Notes

Metrics Calculation

Throughput

N = jobs completed

T = time to complete all jobs

$$\frac{N}{T}$$

Avg. Turnaround Time

C = individual job completion time

N = jobs completed

$$\frac{\sum C}{N}$$

Avg. Wait Time

C = individual job wait time

N = jobs completed

$$\frac{\sum C}{N}$$

Avg. Response Time

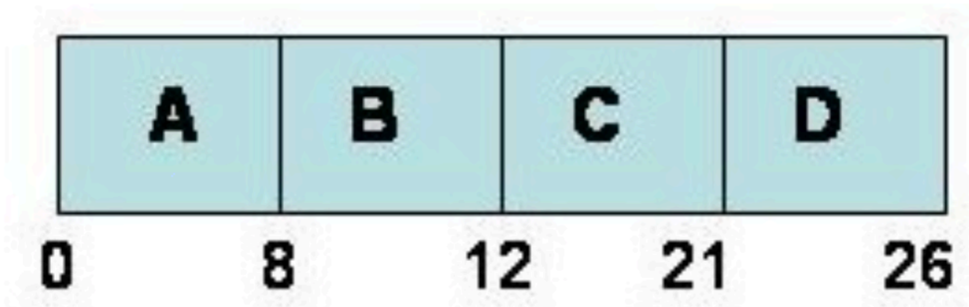
C = individual job response time == first time process begins execution

N = jobs completed

$$\frac{\sum C}{N}$$

FCFS

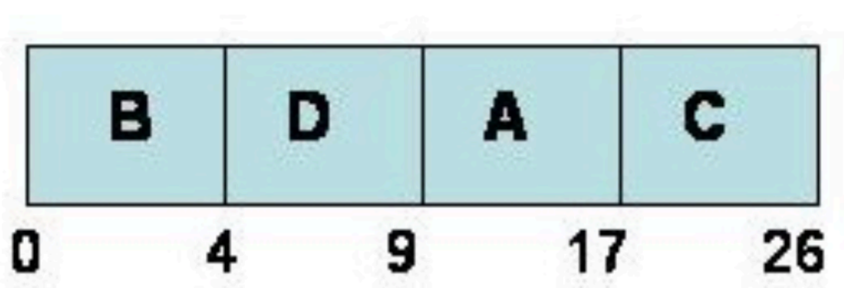
Task	Time Units
A	8
B	4
C	9
D	5



Metric	FCFS
Turn around time	(8+12+21+26)/4 = 16.5 sec/process
Waiting	(0+8+12+21)/4 = 10.25 sec/process
Throughput	4/(26) = .15 process/sec
Response	(0+8+12+21)/4 = 10.25 sec/process

SJF

Task	Time Units
A	8
B	4
C	9
D	5

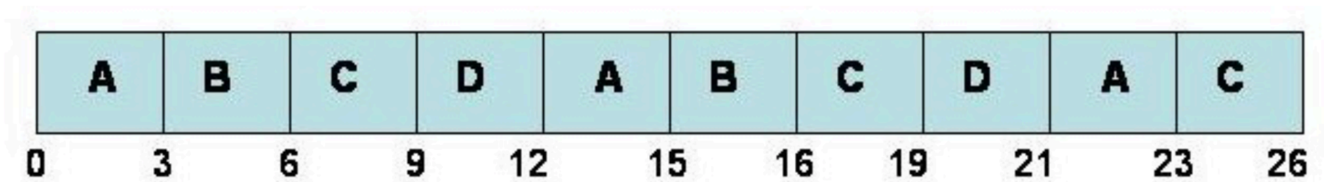


Metric	FCFS
Turn around time	(4+9+17+26)/4 = 14 sec/process
Waiting	(0+4+9+17)/4 = 7.5 sec/process
Throughput	4/(26) = .15 process/sec
Response	(0+4+9+17)/4 = 7.5 sec/process

RR

Task	Time Units
A	8
B	4
C	9
D	5

Time Quanta (time slice) == 3



Metric	FCFS
Turn around time	(23+16+26+21)/4 = 21.5 sec/process
Waiting	((21-3-3)+(15-3)+(23-3-3)+(19-3))/4 = 15 sec/process
Throughput	4/(26) = .15 process/sec
Response	(0+3+6+9)/4 = 4.5 sec/process

Note for round robin wait time we have to subtract the execution time. The easiest way to do this is find the final time slice and subtract t at that time from the number of time slices execute * the time quanta. So you could rewrite the equation

$$\frac{\sum t\_at\_last\_slice - \#\_of\_execution\_slices * time\_slice}{\#\_of\_processes}$$

$$\frac{\sum(21 - 2 * 3) + (15 - 1 * 3) + (23 - 2 * 3) + (19 - 1 * 3)}{4}$$

Timeslice

CPU Bound Tasks: *prefer longer timeslices*

- Limits context switching overheads
- Keeps CPU utilization and throughput high

I/O Bound Tasks: *prefer shorter timeslices*

- I/O bound tasks can issue I/O ops earlier
- Keeps CPU and device utilization high
- Better user-perceived performance

Linux O(1) Scheduler

1. O(1) == constant time to select/add task to the run queue regardless of task count

2. Preemptive, priority-based

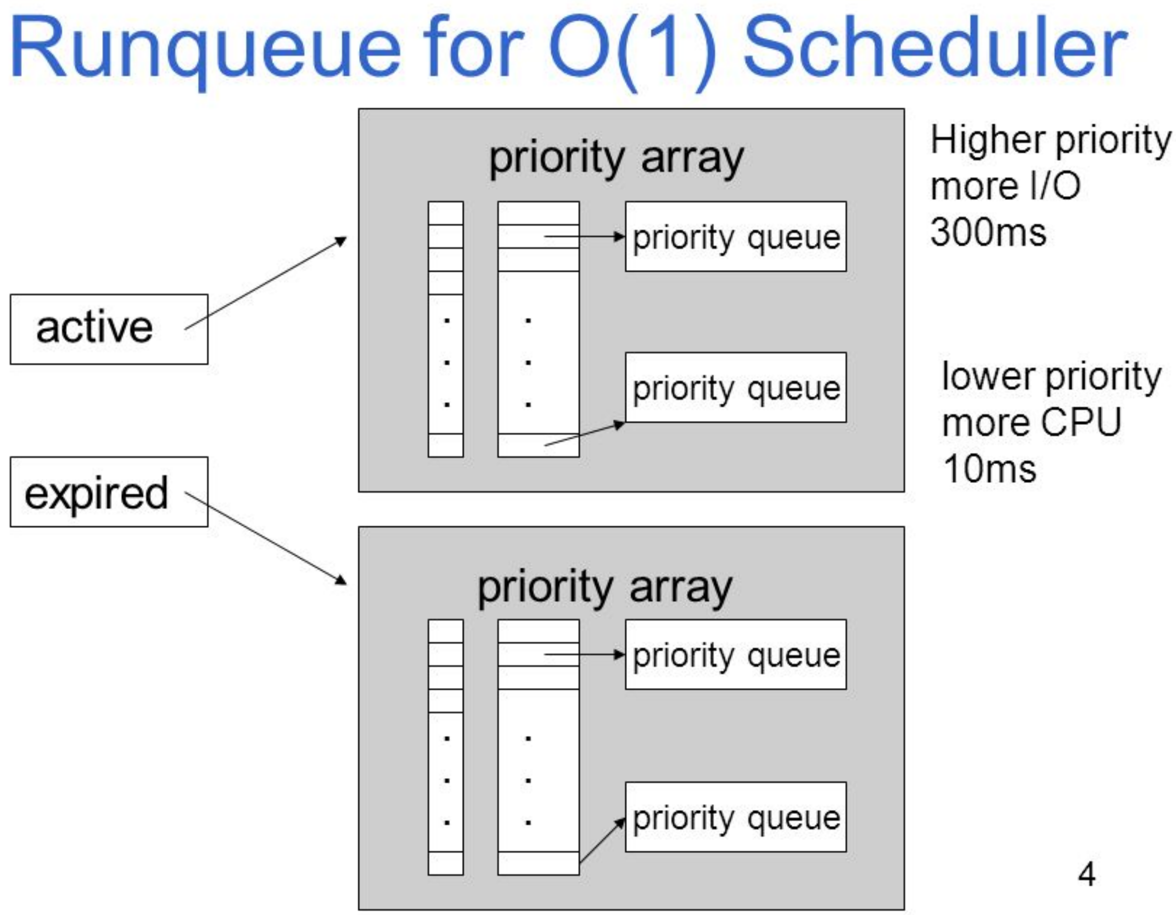
- realtime (0-99)
- timesharing (100-139)

3. User Processes

- default == 120
- nice value(-20 to 19)
  - system call to span 100-139

4. Borrows two traits from MLFQ, though these are implemented differently than a pure MLFQ

- *Timeslice:* the time quantum is associated with the priority level
  - depends on priority
  - smallest for low priority
  - highest for high priority
- *Feedback:* maintains knowledge of the "history" of the tasks
  - sleep time: waiting/idling
  - longer sleep means interactive task- priority **-5** (boost)
  - smaller sleep mean CPU intensive - priority **+5** (lowered)
  - these will increment and decrement between 100-139



5. Runqueue (ready queue) - There are **2 arrays** of tasks

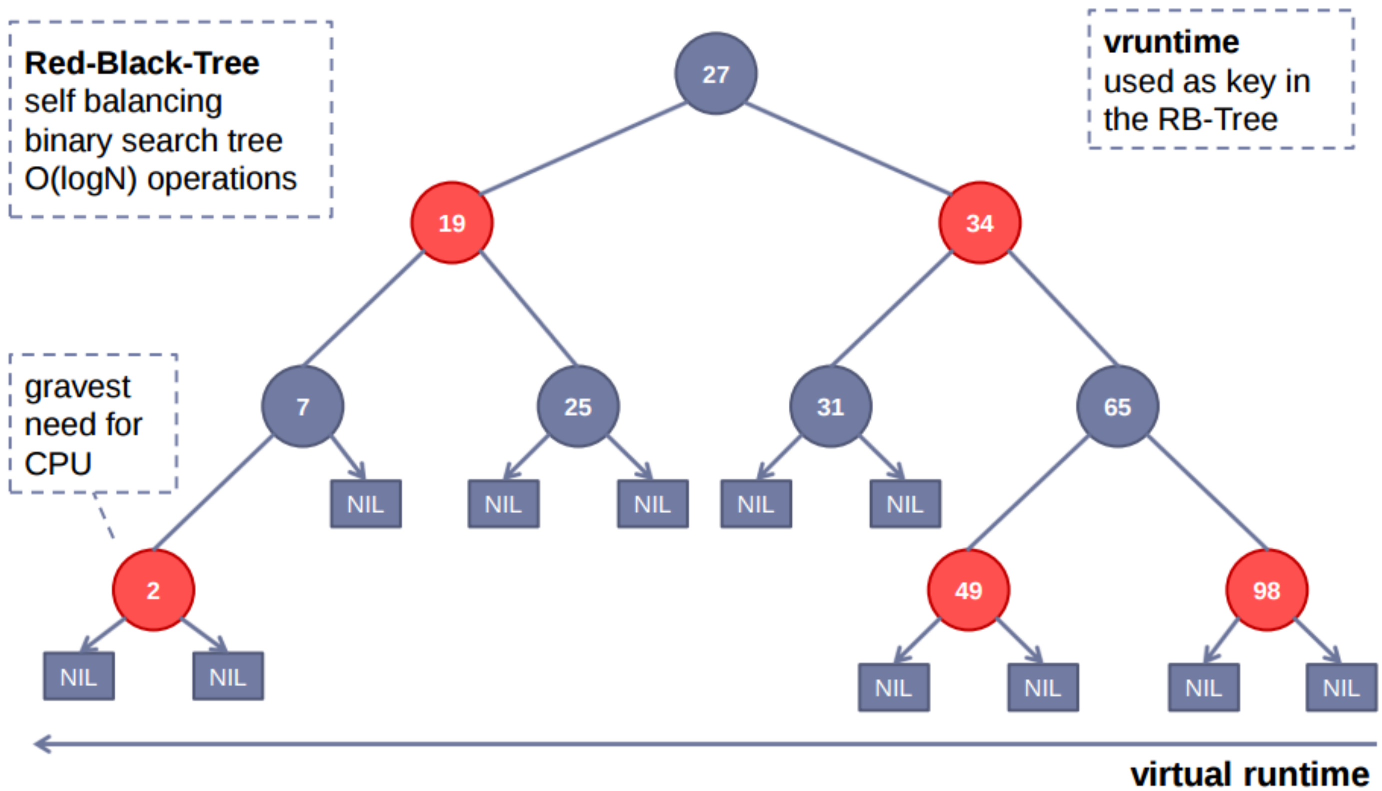
- Active:
  - used to pick the next task to run
  - constant time to add/select
    - uses a bitmap to access array
    - constant time to add and pull from queue at array index
- Expired:
  - inactive list: meaning task will not be scheduled as long as there is something to run in the active array
  - when no more tasks are in active array the pointers to the expired and active arrays swap
    - This helps ensure fairness, though it does not guarantee fairness
  - High to Low timeslice values associate with priority
    - The lower priority task are given low values which makes them more likely to expire
    - The higher priority task are given high timeslice values which makes them likely to stay in the active array

6. Performance
  - poor performance of interactive task
    - when an interactive process is placed on the expired queue (timeslice expired), it will not be

given the CPU until **ALL** task on the active queue are removed
        - This causes "jitter" in performance
    - Does not guarantee fairness
        - starvation could occur (or poor performance) if there is always a process on the active queue

---

Linux CFS Scheduler (now default)



1. CFS was developed to address the issues with O(1) scheduler

2. Runqueue == Red-Black Tree

- Self balancing
- ordered by "vruntime"
- vruntime == time spent on the cpu
    - tracked in a nanosecond granularity

3. Ordering

- The left most child is the task that has spent the least amount of time on the CPU
- The right most child is the task that has consumed more vruntime or CPU

4. CFS scheduling algorithm

- always picks the left most node next
- periodically adjust vruntime
    - will compare currently running task vruntime to the LMN in the tree
    - if smaller, continue running

    - if larger, preempt and place in the tree
- vruntime progress rate depends on priority of niceness
    - vruntime progresses faster for low-priority
        - they will lose the cpu faster
    - vruntime progress slower for high-priority task
        - they will keep the CPU longer
- uses one data structure (red black tree) for all priority levels

5. Performance

- select task => O(1) time
- add task => O(log N)
    - this is acceptable at current system loads, though it may be replaced as systems continue to support more task