# Lecture 12: Process Scheduling Policies

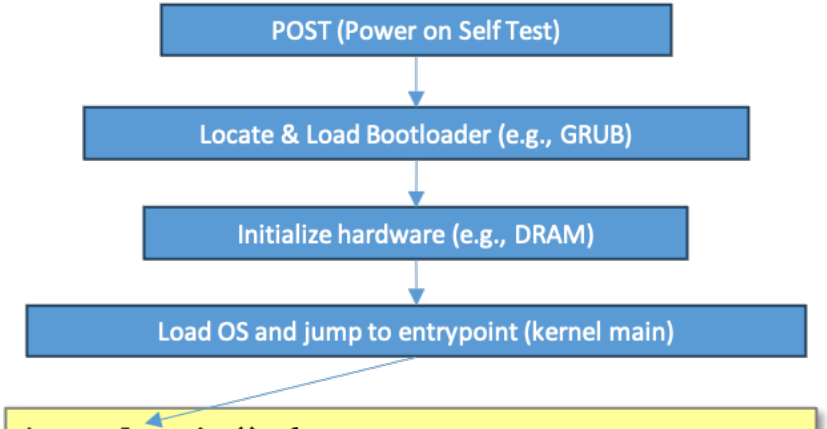
Vivek Kumar

Computer Science and Engineering

IIIT Delhi

vivekk@iiitd.ac.in

# Last Lecture




```
kernel_main() {
    setup();
    init_all_cpus();
    start_init_process(); // on CPU-0
    for(int cpu=1; cpu<numCPUs; cpu++) {
        launch(cpu, scheduler);
    }
    scheduler(); // on CPU-0
}
```

```
void scheduler() {
    while(true) {
        lock(process_table);
        foreach(Process p: scheduling_algorithm(process_table)) {
            if(p->state != READY) {
                continue;
            }
            p->state = RUNNING;
            unlock(process_table);
            swtch(scheduler_process, p);
            // p is done for now..
            lock(process_table);
        }
        unlock(process_table);
    }
}
```

| Property | FCFS | SJF |
|---|---|---|
| Optimize Average Completion Time | ✗ | ✓ |
| Prevent Starvation | ✓ | ✗ |
| Prevent Convoy Effect | ✗ | ✗ |
| Psychic Skills **NOT** Needed | ✓ | ✗ |

# Today's Class

- **Process scheduling policies**
  - **Shortest Time to Completion First**
  - Round Robin
  - Multi-level feedback scheduling
  - Linux Completely Fair Scheduler (CFS)

Acknowledgements: Lecture content derived from the CS162 course offered at University of California, Berkeley

# Shortest Time to Completion First (STCF)

Introduce the notion of preemption

A running task can be de-scheduled before completion.

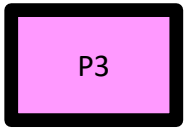
**STCF**

Schedule the task with the least amount of time **left**

# Shortest Time to Completion First (STCF)

Schedule the task with the least amount of time left

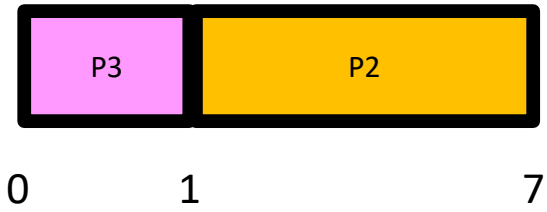
| Process | Burst Time (left) | Arrival Time |
|---|---|---|
| $P_1$ | 3 | *10* |
| $P_2$ | 6 | *1* |
| $P_3$ | 24 | *0* |
| $P_4$ | 16 | *18* |



# Shortest Time to Completion First (STCF)

Schedule the task with the least amount of time left

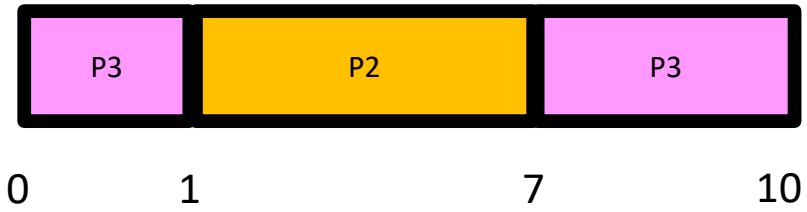
| Process | Burst Time (left) | Arrival Time |
|---|---|---|
| $P_1$ | 3 | *10* |
| $P_2$ | 6 | *1* |
| $P_3$ | 23 | *0* |
| $P_4$ | 16 | *18* |



# Shortest Time to Completion First (STCF)

Schedule the task with the least amount of time left

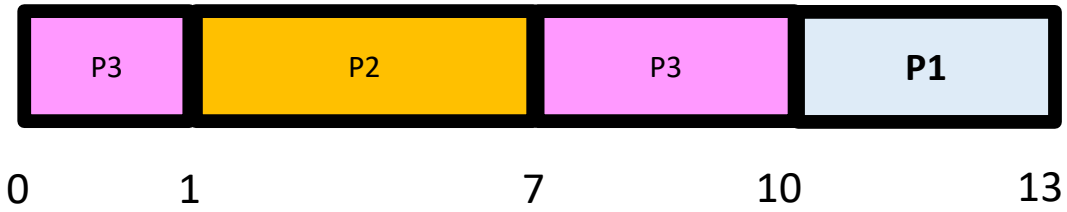
| Process | Burst Time (left) | Arrival Time |
|---|---|---|
| $P_1$ | 3 | *10* |
| $P_2$ | 0 | *1* |
| $P_3$ | 23 | *0* |
| $P_4$ | 16 | *20* |



# Shortest Time to Completion First (STCF)

Schedule the task with the least amount of time left

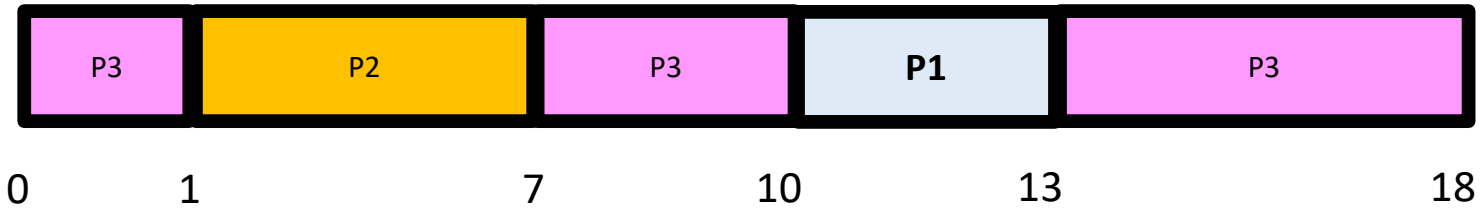
| Process | Burst Time (left) | Arrival Time |
|---|---|---|
| $P_1$ | 3 | *10* |
| $P_2$ | 0 | *1* |
| $P_3$ | 20 | *0* |
| $P_4$ | 16 | *18* |



# Shortest Time to Completion First (STCF)

Schedule the task with the least amount of time left

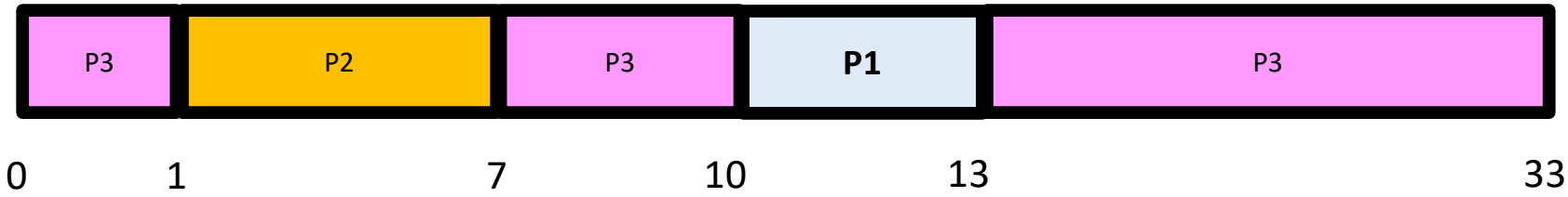
| Process | Burst Time (left) | Arrival Time |
|---|---|---|
| $P_1$ | 0 | *10* |
| $P_2$ | 0 | *1* |
| $P_3$ | 15 | *0* |
| $P_4$ | 16 | *18* |



# Shortest Time to Completion First (STCF)

Schedule the task with the least amount of time left

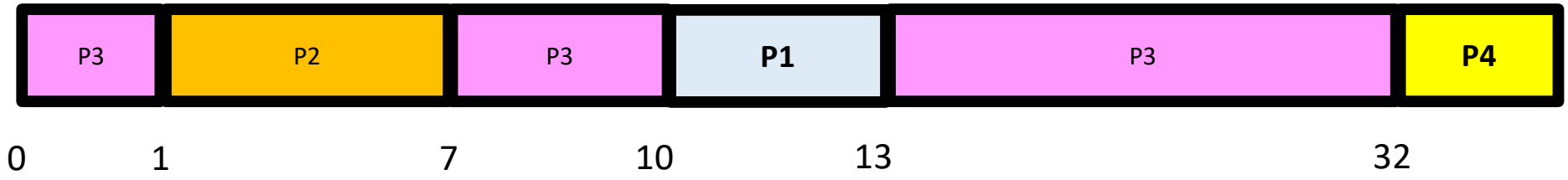
| Process | Burst Time (left) | Arrival Time |
|---|---|---|
| $P_1$ | 0 | *10* |
| $P_2$ | 0 | *1* |
| $P_3$ | 0 | *0* |
| $P_4$ | 15 | *18* |



# Shortest Time to Completion First (STCF)

Schedule the task with the least amount of time left

| Process | Burst Time (left) | Arrival Time |
|---|---|---|
| $P_1$ | 0 | *10* |
| $P_2$ | 0 | *1* |
| $P_3$ | 0 | *0* |
| $P_4$ | 15 | *18* |



# Are we done?

Can STCF lead to starvation?

Yes

Any scheduling policy that always favours a fixed property for scheduling leads starvation

No change!

# Are we done?

Is STCF subject to the convoy effect?

No!

STCF is a preemptible policy

# Today's Class

- **Process scheduling policies**
  - Shortest Time to Completion First
  - **Round Robin**
  - Multi-level feedback scheduling
  - Linux Completely Fair Scheduler (CFS)

Acknowledgements: Lecture content derived from the CS162 course offered at University of California, Berkeley

# Round-Robin Scheduling

RR runs a job for a **time slice**

(a **scheduling quantum**)

Once time slice over,

Switch to next job in ready queue.

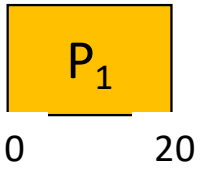
=> Called time-slicing

# RR with Time Quantum = 20

| Process | Burst Time |
|---|---|
| $P_1$ | 53 |
| $P_2$ | 8 |
| $P_3$ | 68 |
| $P_4$ | 24 |

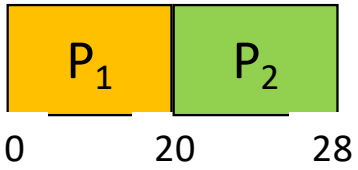
# RR with Time Quantum = 20

| Process | Burst Time |
|---|---|
| $P_1$ | 53 => 33 |
| $P_2$ | 8 |
| $P_3$ | 68 |
| $P_4$ | 24 |



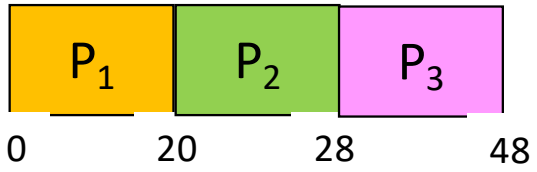
# RR with Time Quantum = 20

| Process | Burst Time |
|---|---|
| $P_1$ | 33 |
| $P_2$ | 8 => 0 |
| $P_3$ | 68 |
| $P_4$ | 24 |



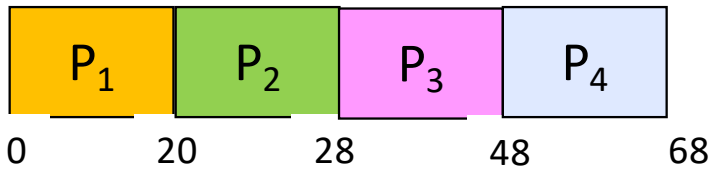
# RR with Time Quantum = 20

| Process | Burst Time |
|---|---|
| $P_1$ | 33 |
| $P_2$ | 0 |
| $P_3$ | 68 => 48 |
| $P_4$ | 24 |



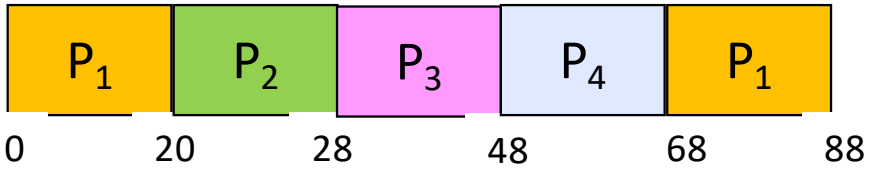
# RR with Time Quantum = 20

| Process | Burst Time |
|---|---|
| $P_1$ | 33 |
| $P_2$ | 0 |
| $P_3$ | 48 |
| $P_4$ | 24 => 4 |



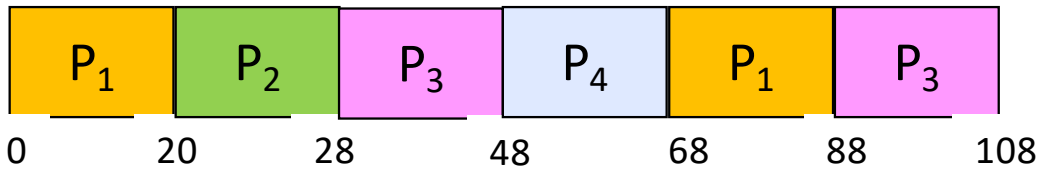
# RR with Time Quantum = 20

| Process | Burst Time |
|---|---|
| $P_1$ | 33 => 13 |
| $P_2$ | 0 |
| $P_3$ | 48 |
| $P_4$ | 4 |



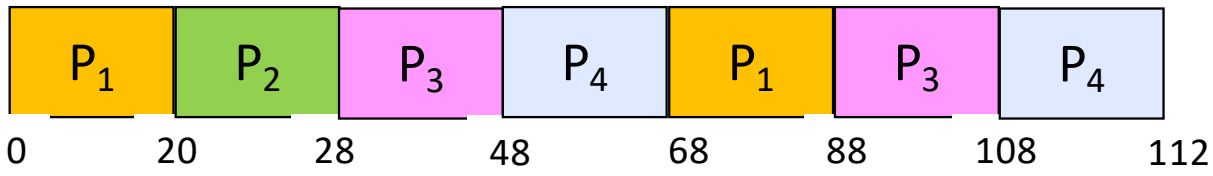
# RR with Time Quantum = 20

| Process | Burst Time |
|---|---|
| $P_1$ | 13 |
| $P_2$ | 0 |
| $P_3$ | 48 => 28 |
| $P_4$ | 4 |



# RR with Time Quantum = 20

| Process | Burst Time |
|---|---|
| $P_1$ | 13 |
| $P_2$ | 0 |
| $P_3$ | 28 |
| $P_4$ | 4 => 0 |



# RR with Time Quantum = 20

## Waiting time

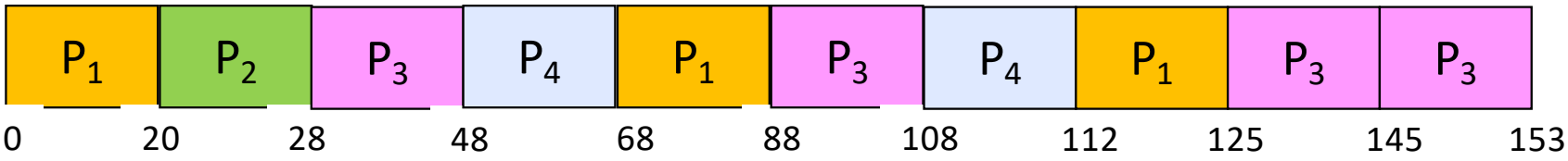
- $P_1$= 0 + (68-20)+(112-88)=72
- $P_2$=(20-0)=20
- $P_3$=(28-0)+(88-48)+(125-108)+0=85
- $P_4$=(48-0)+(108-68)=88

## Average waiting time

$$(\frac{72+20+85+88}{4} = 66.25)$$

## Average completion time

$$(\frac{125+28+153+112}{4} = 104.25)$$



# RR Quantum

- Assume there is no context switching overhead
- What happens to average completion time when we *decrease Q*?

# Switching is not free!

Small scheduling quantas lead to

frequent context switches

- Context switch overhead

- Trash cache-state

*q* must be large with respect to context switch,

otherwise overhead is too high

# Are we done?

Can RR lead to starvation?

No

# Are we done?

Can RR suffer from convoy effect?

No

Only run a time-slice at a time

# Summary

| Property | FCFS | SJF | STCF | RR |
|---|---|---|---|---|
| Optimize Average **Completion** Time | ✗ | ✓ | ✓ | ✗ |
| Optimize Average **Waiting** Time | ✗ | ✗ | ✗ | ✓ |
| Prevent Starvation | ✓ | ✗ | ✗ | ✓ |
| Prevent Convoy Effect | ✗ | ✗ | ✓ | ✓ |
| Psychic Skills **NOT** Needed | ✓ | ✗ | ✗ | ✓ |

High in case of long running processes

Doesn't favor any fixed property

Due to time slicing

# Today's Class

- **Process scheduling policies**
  - Shortest Time to Completion First
  - Round Robin
  - **Multi-level feedback scheduling**
  - Linux Completely Fair Scheduler (CFS)

Acknowledgements: Lecture content derived from the CS162 course offered at University of California, Berkeley

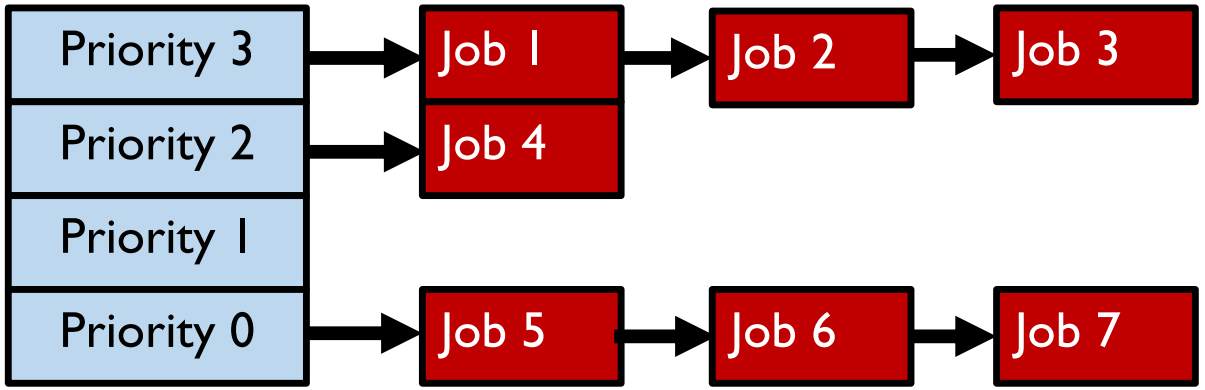
# What we Want?

It cannot be achieved without having psychic skills

Would it be true for IO/interactive jobs having short CPU bursts?

| Property | STCF | RR |
|---|---|---|
| Optimize Average **Completion** Time | ✓ | ✗ |
| Optimize Average **Waiting** Time | ✗ | ✓ |
| Prevent Starvation | ✗ | ✓ |
| Prevent Convoy Effect | ✓ | ✓ |
| Psychic Skills **NOT** Needed | ✗ | ✓ |

# Approximation of CPU Bursts using Priority Scheduling




- Priority scheduling always run the **ready** process (or job) with *highest priority*
- Systems may try to set priorities according to some **policy goal**
  - Example: Give IO/interactive jobs higher priority than long calculation
- How to achieve fairness and avoid starvation?
  - Elevate priority of threads that don't get CPU time

# Multi-level Feedback Queue

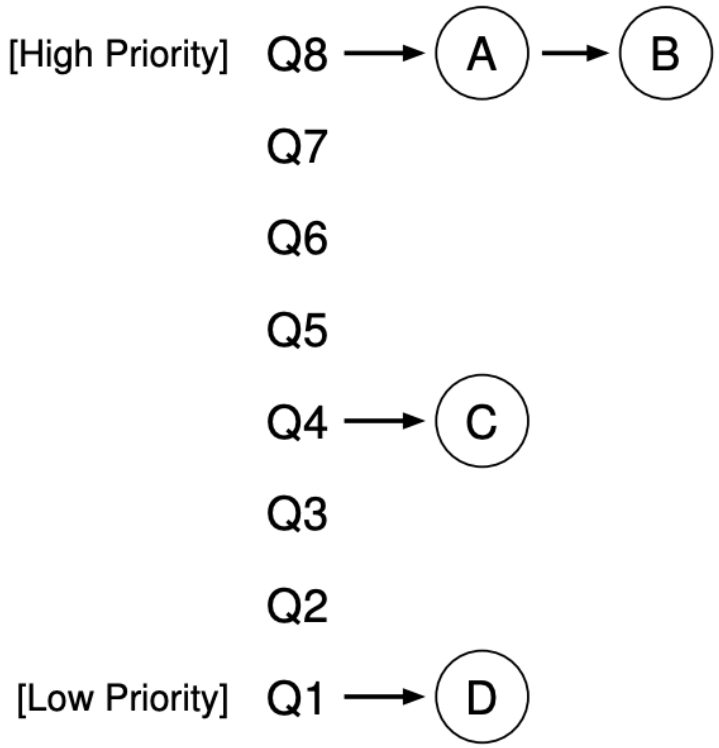



Figure 8.1: **MLFQ Example**

- Create distinct **queues for ready processes**, each assigned a different **priority level**
- All processes belong to any one queue at any given time
- Processes can move between queues
- Use priorities to decide from which queue process should be picked next
- Individual queues run RR with increasing time slice

# MLFQ (Naïve Version)

## Rule 1

If Priority(A) > Priority(B) (different queues)

A runs (B doesn't).

## Rule 2

If Priority(A) = Priority(B), A & B run in RR.

## Key question:

**How do you set the priorities?**

Vary the priority of a job based on its *observed behavior*

# Learning behavior

## Rule 3

When a job enters the system, it is placed at the highest priority (the topmost queue).

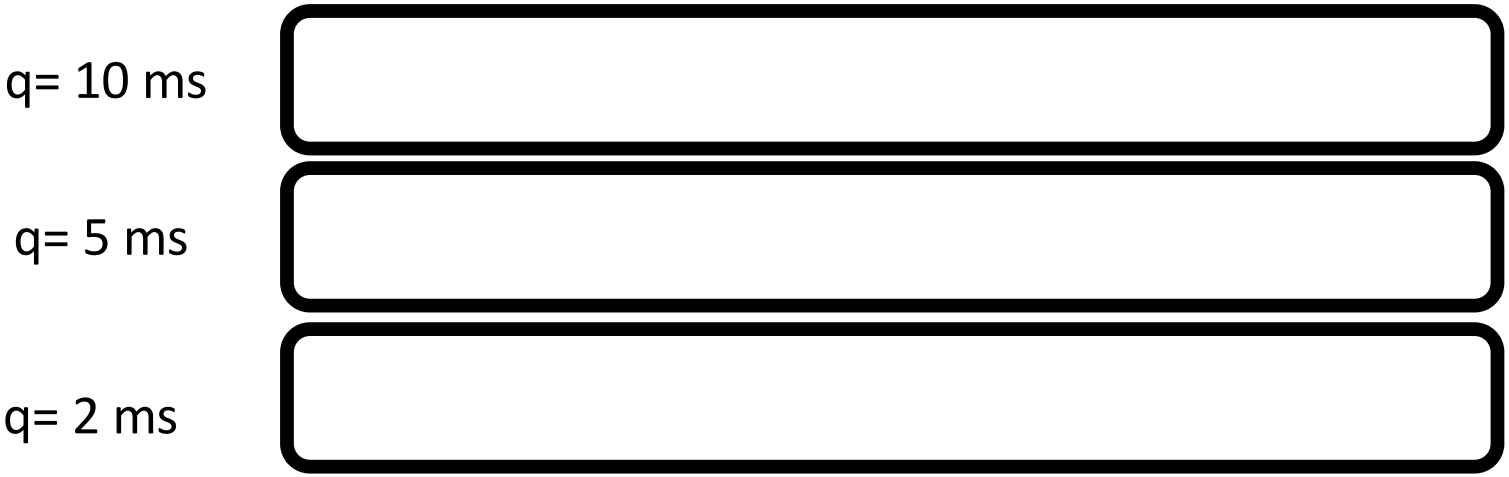
## Rule 4a

If a job uses up an entire time slice while running, its priority is *reduced* (i.e., it moves down one queue).

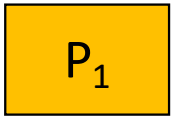
# Learning behavior

$P_1$

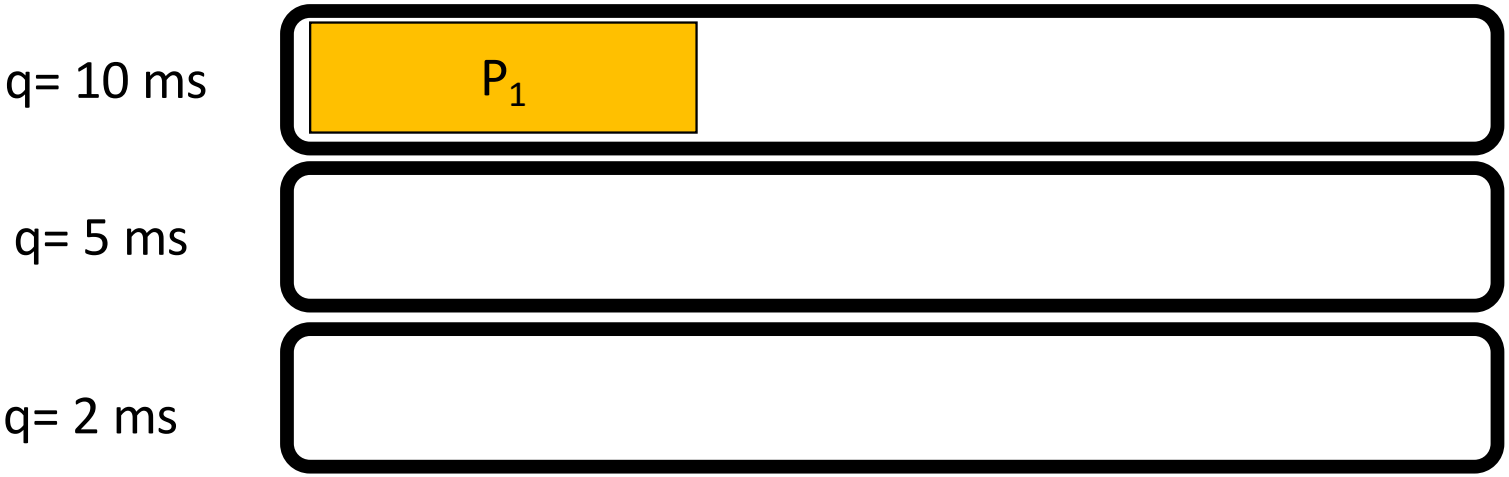
Long running CPU-bound



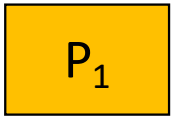
# Learning behavior



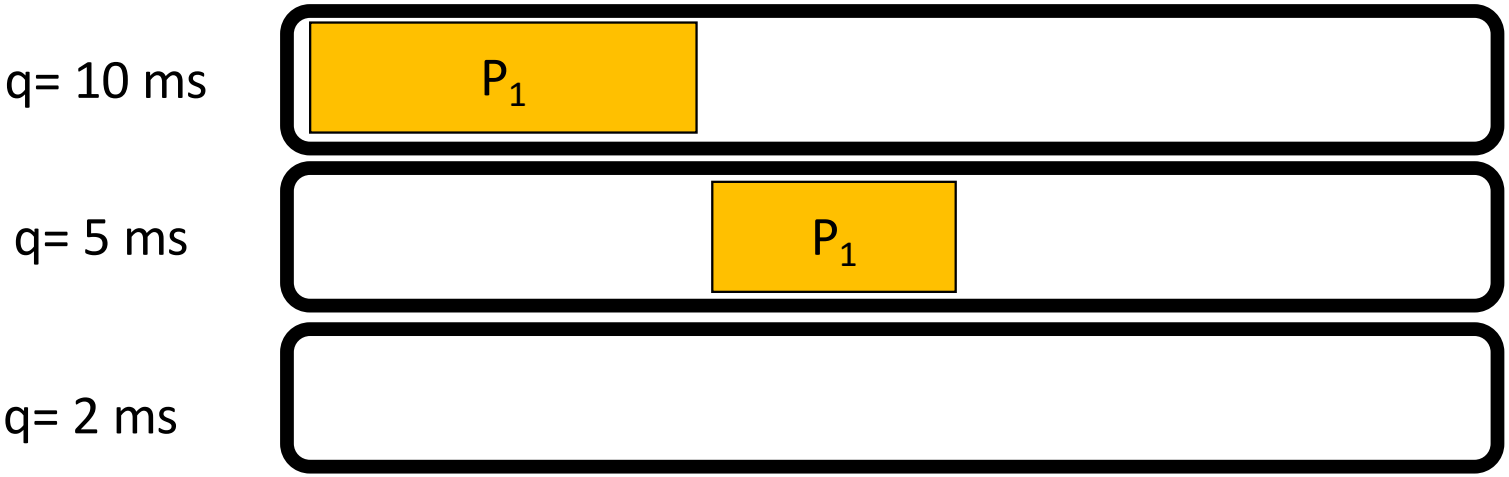

Long running CPU-bound



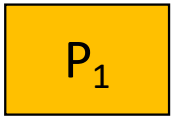
# Learning behavior



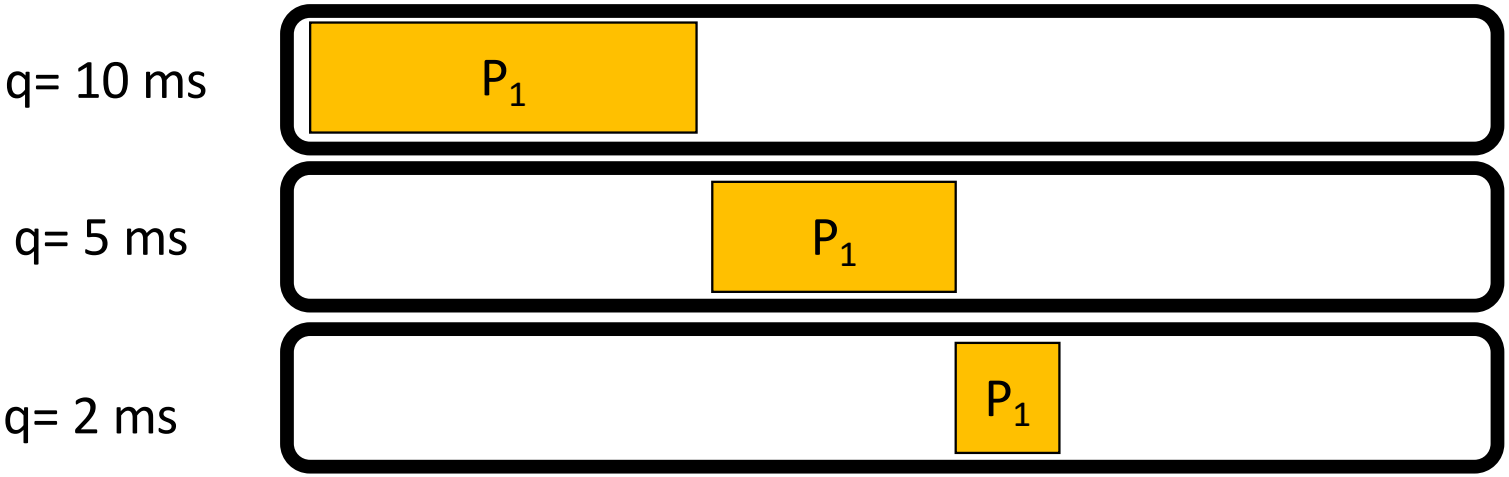

Long running CPU-bound



# Learning behavior




Long running CPU-bound



# Learning behavior

**Rule 3**

When a job enters the system, it is placed at the highest priority (the topmost queue).

**Rule 4a**

If a job uses up an entire time slice while running, its priority is *reduced* (i.e., it moves down one queue).

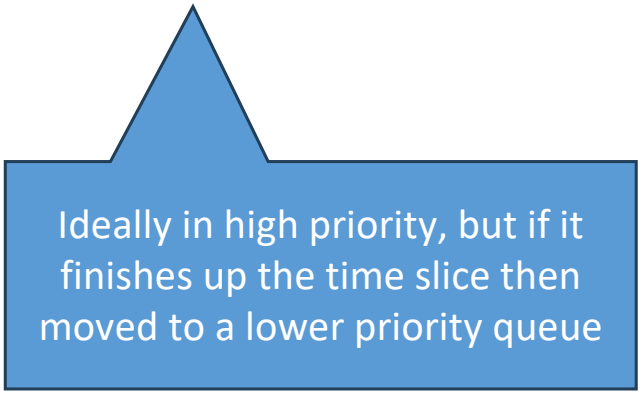
**Rule 4b**

If a job gives up the CPU before the time slice is up, it stays at the *same* priority level.

# Learning behavior

**Where do IO-bound/interactive jobs end up?**

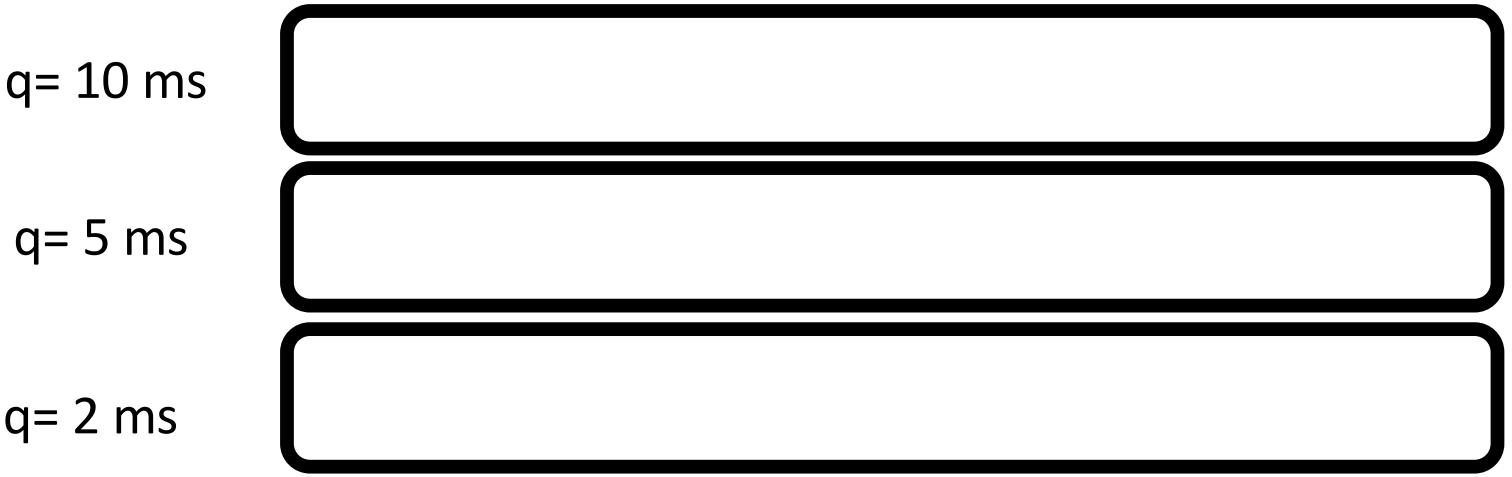
a) High Priority Queue b) Low Priority Queue



# Learning behavior

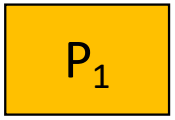
$P_1$

Long running CPU-bound

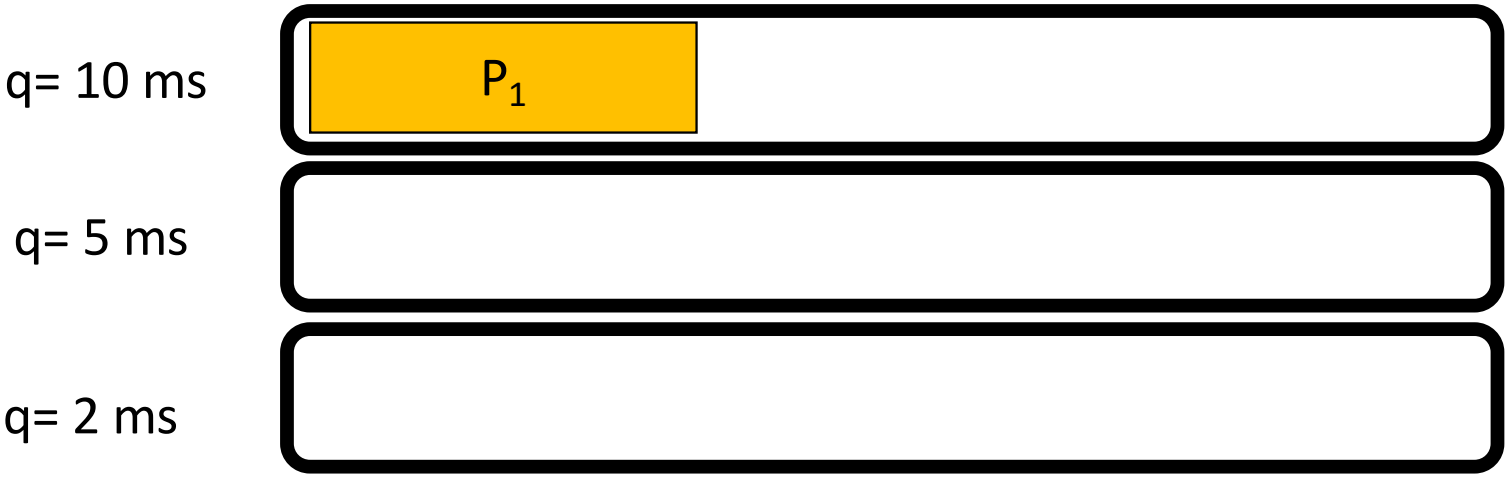



Schedule

# Learning behavior

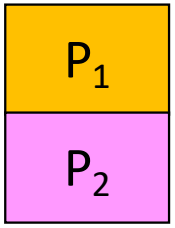



Long running CPU-bound



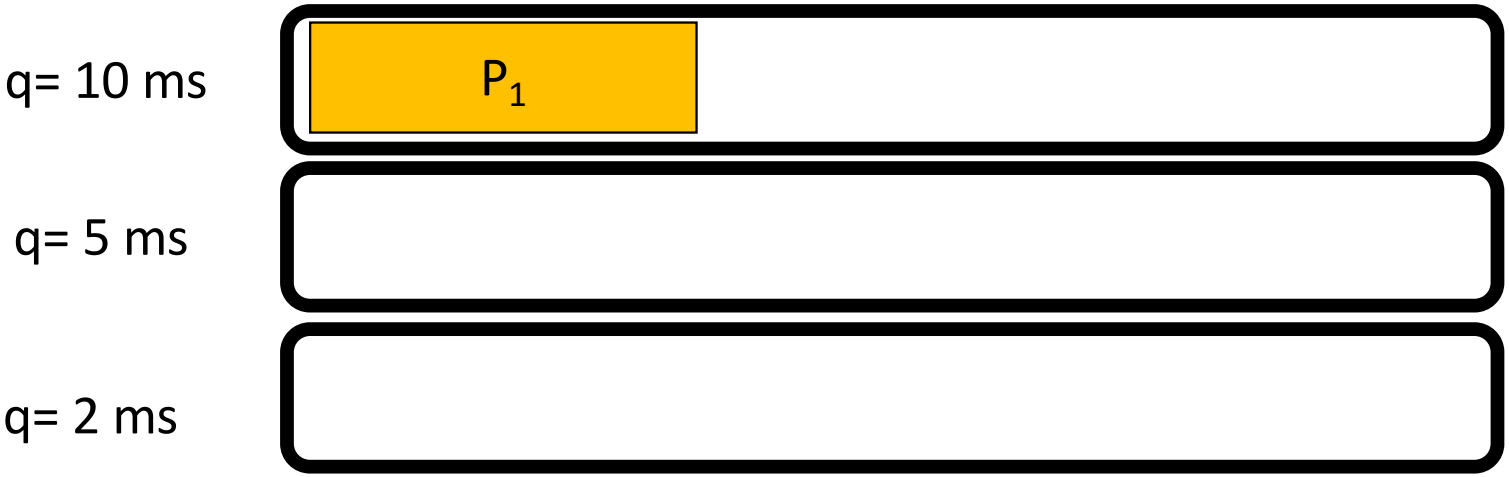

Schedule

# Learning behavior



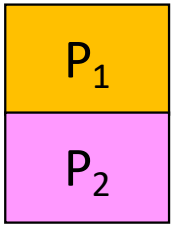

Long running CPU-bound

Computes for 1 ms and then IO for 1 ms




Schedule

# Learning behavior

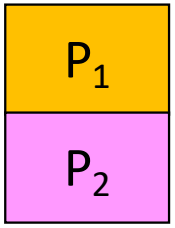



Long running CPU-bound

Computes for 1 ms and then IO for 1 ms

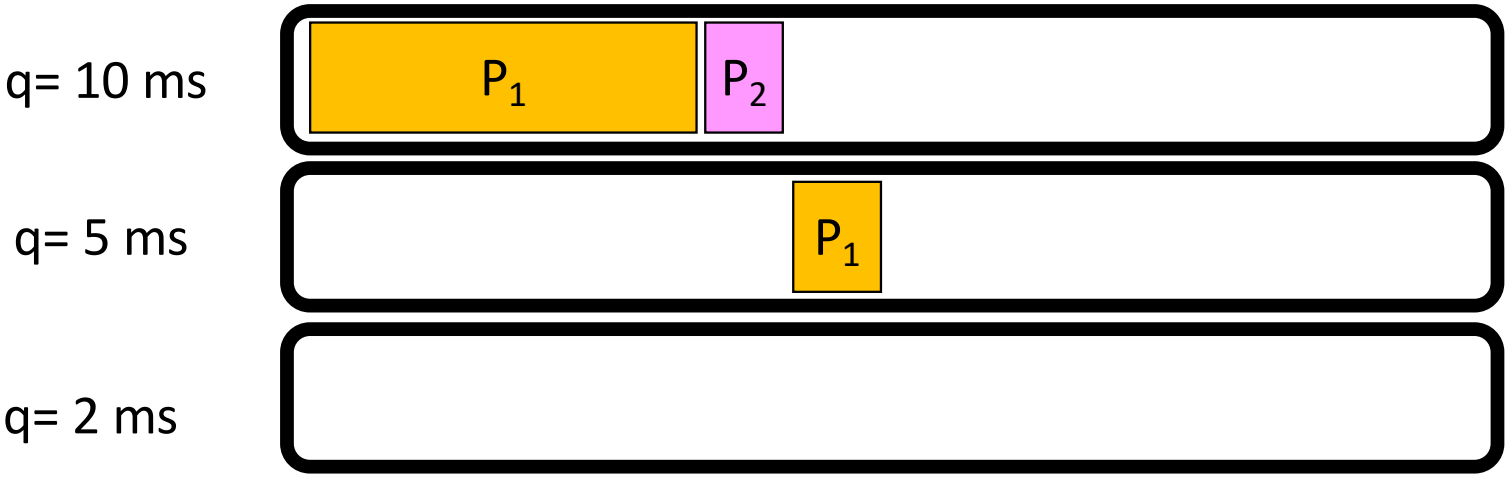

Schedule

# Learning behavior



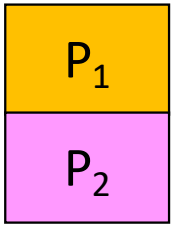

Long running CPU-bound

Computes for 1 ms and then IO for 1 ms



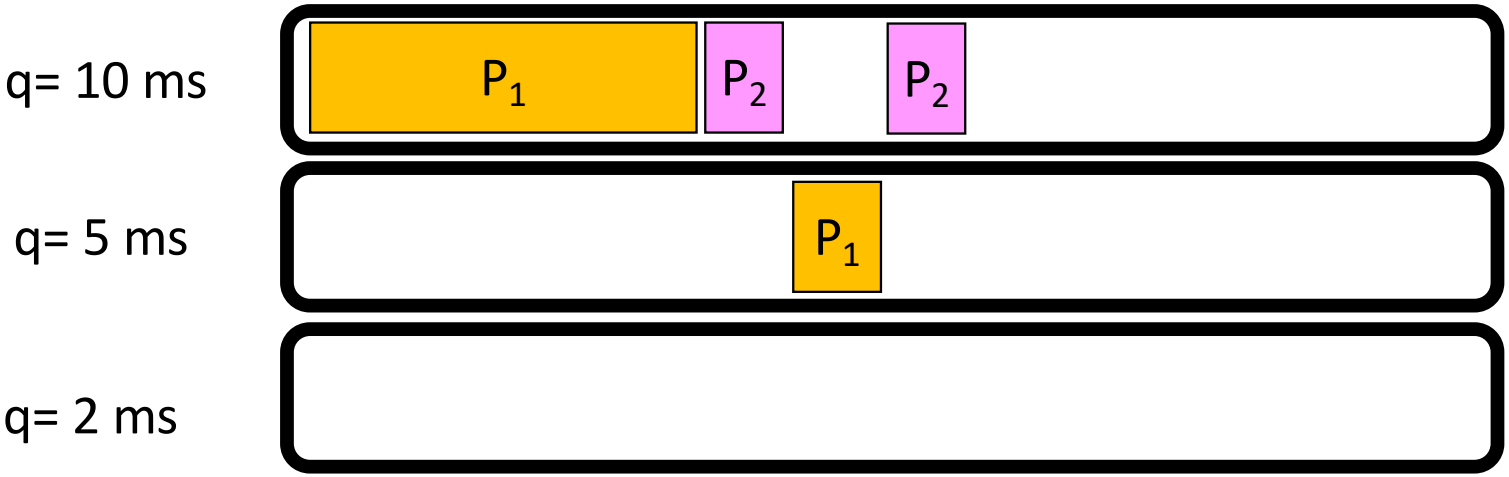

Schedule

# Learning behavior



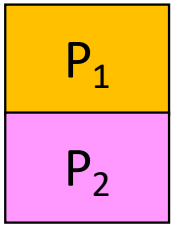

Long running CPU-bound

Computes for 1 ms and then IO for 1 ms



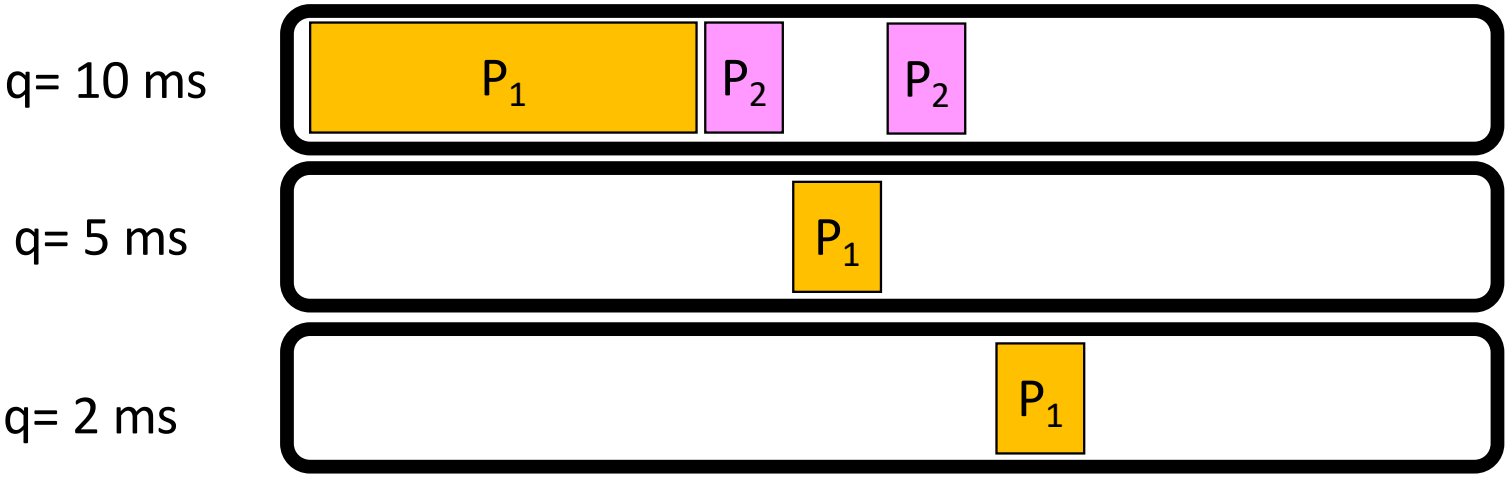

Schedule

# Learning behavior



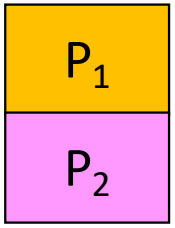

Long running CPU-bound

Computes for 1 ms and then IO for 1 ms



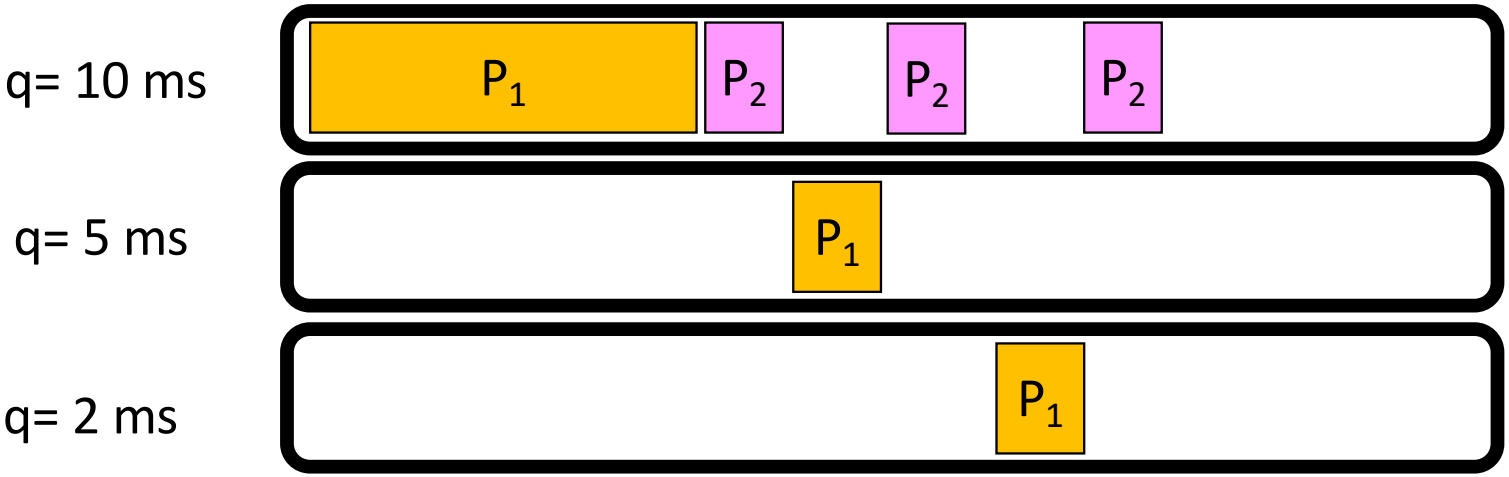

Schedule

# Learning behavior



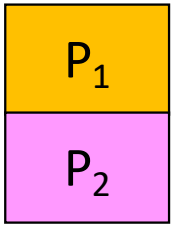

Long running CPU-bound

Computes for 1 ms and then IO for 1 ms



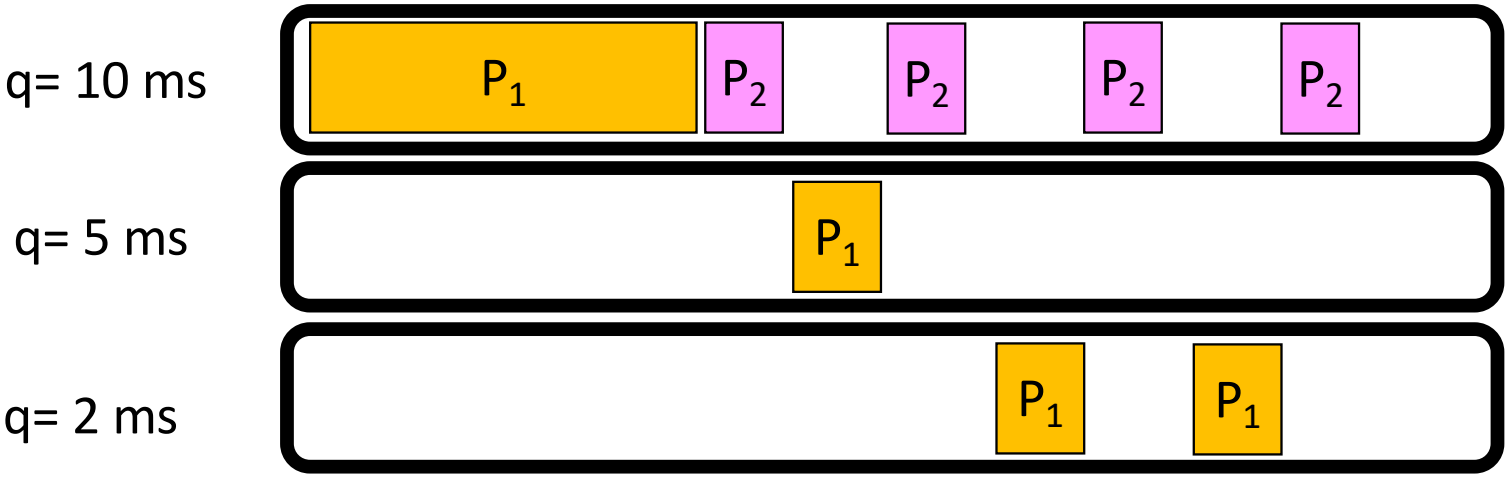

Schedule

# Learning behavior




Long running CPU-bound

Computes for 1 ms and then IO for 1 ms




Schedule

# MLFQ (Naïve Version): Pseudocode

```
foreach(queue : PrioritizedReadyQueues) { // RR
    foreach(job : queue) { // RR
        int timeslice = queue.getSlice();
        // Execute job for timeslice associated with this queue
        // If job used full timeslice then push job in
        // ready queue with one lower priority level
    }
}
```

# Are we done?

Can MLFQ can be gamed?

Intentionally insert IO request just before time quanta to stay on queue.

MLFQ is subject to starvation?

Yes!

# Are we done?

MLQF can be gamed:

Intentionally insert IO request just before time quanta to stay on queue.

Rule 4

Once a job uses up its time allotment at a given levels (regardless of how many times gave up CPU), reduce priority

MLQF is subject to starvation:

Systematically prioritize higher-priority queues

Rule 5

After some time period S, move all jobs in system to the topmost queue.

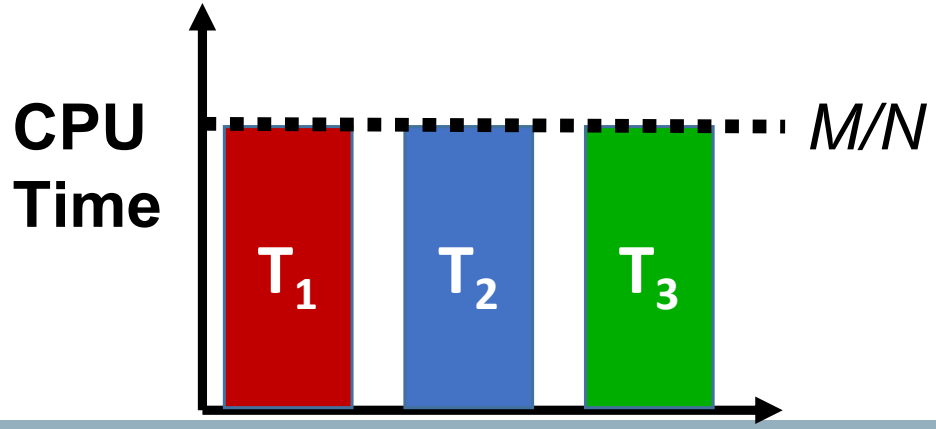
# Today's Class

- **Process scheduling policies**
  - Shortest Time to Completion First
  - Round Robin
  - Multi-level feedback scheduling
  - **Linux Completely Fair Scheduler (CFS)**

Acknowledgements: Lecture content derived from the CS162 course offered at University of California, Berkeley

# Linux Completely Fair Scheduler

- Goal: Each process gets an equal share of CPU
- Attempts to divide the CPU time fairly (equally) among all the processes
  - If the latency to resume a scheduler is M units of time, N processes will get M/N time slice of CPU



# Linux CFS

- **Constraint 1**: *Scheduling Latency*

- Target Latency: 20ms, 4 Processes
  - Each process gets 5ms time slice
- Target Latency: 20 ms, 200 Processes
  - Each process gets 0.1ms time slice
  - Recall Round-Robin: Huge context switching overhead

# Linux CFS

- **Constraint 2**: *Minimum Granularity for scheduling latency*

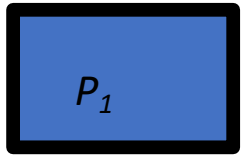
- Target Latency 20ms, Minimum Granularity 1ms, 100 processes
  - Each process gets 1ms time slice

# Linux CFS

- Which process to pick from the ready queue?
- **Constraint 3**: *Pick the process from the ready queue with minimum vruntime (helps the IO/interactive jobs)*
  - Uses virtual runtime (vruntime) variable in PCB to keep track of time a process has executed on the CPU, and it is updated at every **context switch**
    - vruntime += t * weight based on process priority

# Linux CFS

| Process | Burst Time |
|---|---|
| $P_1$ | 5 |
| $P_2$ | 10 |
| $P_3$ | 16 |
| $P_4$ | 16 |

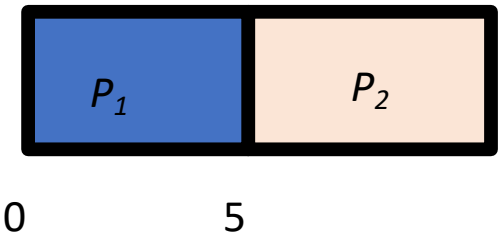



0

- Scheduling latency is **20ms**, minimum granularity is 1ms, and weight=1
- Time slice in beginning = 20/4 = 5ms

# Linux CFS

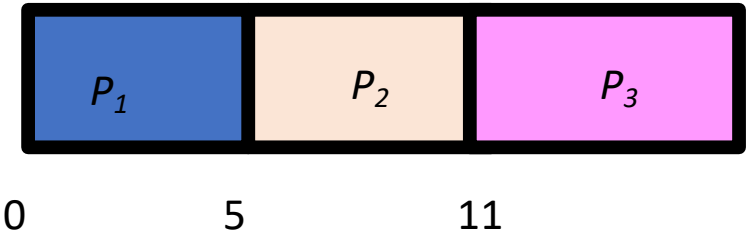
| Process | Burst Time |
|---|---|
| $P_1$ | **0** |
| $P_2$ | 10 |
| $P_3$ | 16 |
| $P_4$ | 16 |




- Scheduling latency is **20ms**, minimum granularity is 1ms, and weight=1
- Time slice in beginning = 20/4 = 5ms
- Time slice after P1 terminates = 20/3 ~ 6ms

# Linux CFS

| Process | Burst Time |
|---|---|
| $P_1$ | 0 |
| $P_2$ | **4** |
| $P_3$ | 16 |
| $P_4$ | 16 |

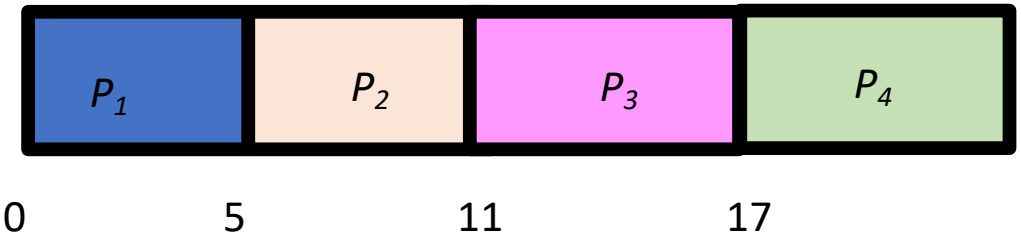



- Scheduling latency is **20ms**, minimum granularity is 1ms, and weight=1
- Time slice in beginning = 20/4 = 5ms
- Time slice after P1 terminates = 20/3 ~ 6ms

# Linux CFS

| Process | Burst Time |
|---|---|
| $P_1$ | 0 |
| $P_2$ | 4 |
| $P_3$ | **10** |
| $P_4$ | 16 |

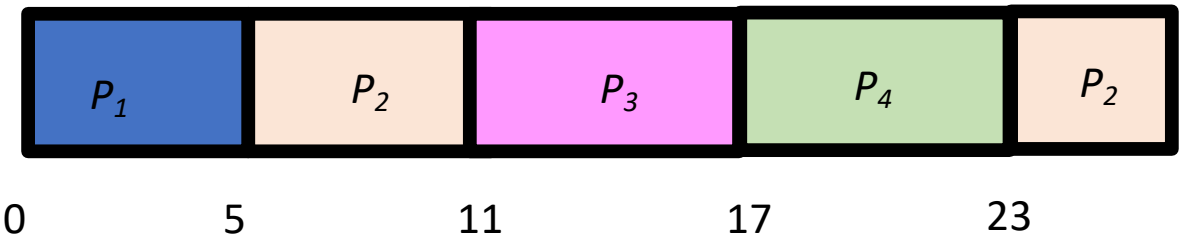
- Scheduling latency is **20ms**, minimum granularity is 1ms, and weight=1
- Time slice in beginning = 20/4 = 5ms
- Time slice after P1 terminates = 20/3 ~ 6ms



# Linux CFS

| Process | Burst Time |
|---|---|
| $P_1$ | 0 |
| $P_2$ | 4 |
| $P_3$ | 10 |
| $P_4$ | **10** |

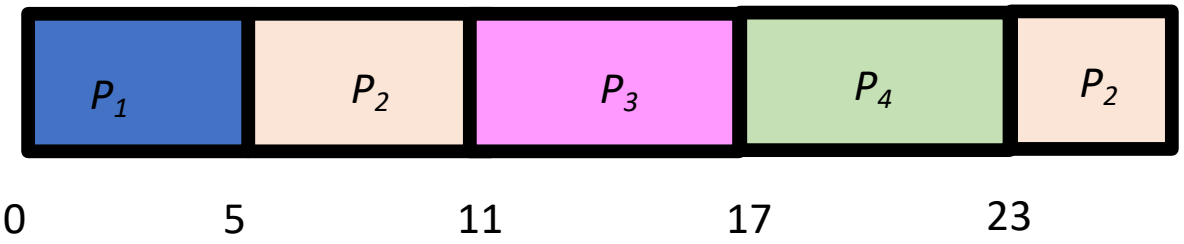
- Scheduling latency is **20ms**, minimum granularity is 1ms, and weight=1
- Time slice in beginning = 20/4 = 5ms
- Time slice after P1 terminates = 20/3 ~ 6ms



# Linux CFS

| Process | Burst Time |
|---|---|
| $P_1$ | 0 |
| $P_2$ | **0** |
| $P_3$ | 10 |
| $P_4$ | 10 |

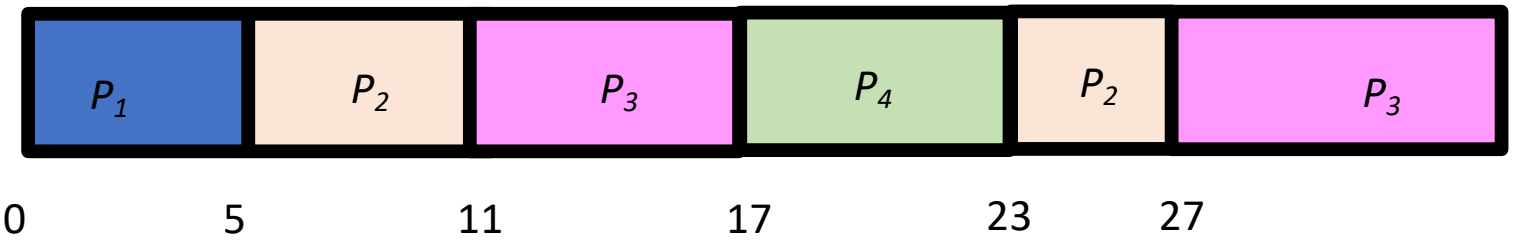
- Scheduling latency is **20ms**, minimum granularity is 1ms, and weight=1
- Time slice in beginning = 20/4 = 5ms
- Time slice after P1 terminates = 20/3 ~ 6ms
- Time slice after P2 terminates = 20/2 = 10ms



# Linux CFS

| Process | Burst Time |
|---|---|
| $P_1$ | 0 |
| $P_2$ | 0 |
| $P_3$ | **0** |
| $P_4$ | 10 |

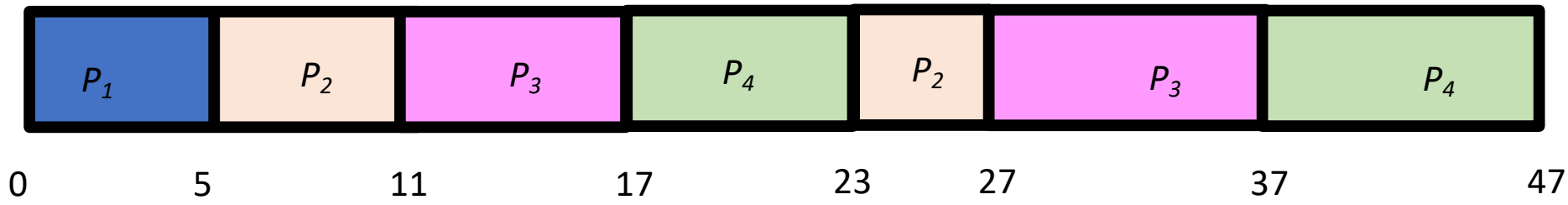
- Scheduling latency is **20ms**, minimum granularity is 1ms, and weight=1
- Time slice in beginning = 20/4 = 5ms
- Time slice after P1 terminates = 20/3 ~ 6ms
- Time slice after P2 terminates = 20/2 = 10ms



# Linux CFS

| Process | Burst Time |
|---|---|
| $P_1$ | 0 |
| $P_2$ | 0 |
| $P_3$ | 0 |
| $P_4$ | 0 |

- Scheduling latency is **20ms**, minimum granularity is 1ms, and weight=1
- Time slice in beginning = 20/4 = 5ms
- Time slice after P1 terminates = 20/3 ~ 6ms
- Time slice after P2 terminates = 20/2 = 10ms



# Next Lecture

- Dynamic memory allocation