CS162
Operating Systems and Systems Programming
Lecture 11

Scheduling
Core Concepts and Classic Policies

Professor Natacha Crooks
https://cs162.org/

Slides based on prior slide decks from David Culler, Ion Stoica, John Kubiatowicz, Alison Norman and Lorenzo Alvisi

Recall: Scheduling Policy Goals/Criteria

Minimise Latency

Maximise Throughput

While remaining fair and starvation-free

Recall: Useful metrics

Waiting time for P

Total Time spent waiting for CPU

Average waiting time

Average of all processes’ wait time

Response Time for P

Time to when process gets first scheduled

Completion time

Waiting time + Run time

Average completion time

Average of all processes' completion time

Recall: Important Performance Metrics

Fairness

Equality in the performance perceived by one task

Starvation

The lack of progress for one task, due to resources being allocated to different tasks

Recall: Assumptions

Threads are independent!

One thread = One User

Unrealistic but simplify the problem so it can be solved

Only look at work-conserving scheduler
=> Never leave processor idle if work to do

Recall: FCFS/FIFO Summary

<u>The good</u>

Simple
Low Overhead
No Starvation*

<u>The bad</u>

Sensitive to arrival order
(poor predictability)

<u>The ugly</u>

Convoy Effect.
Bad for Interactive Tasks

Recall: SJF Summary

The good

Optimal Average Completion Time when jobs arrive simultaneously

The bad

Still subject to convoy effect

The ugly

Can lead to starvation!

Requires knowing duration of job

Recall: STCF Summary

The good

Optimal Average Completion Time Always

The bad

The ugly

Can lead to starvation!

Requires knowing duration of job

Recall: Taking a step back

Property	FCFS	SJF	STCF
Optimise Average Completion Time		✓	✓
Prevent Starvation			
Prevent Convoy Effect	✓ *		
Psychic Skills Not Needed			✓

✓

Goals for Today

- Round-robin scheduling (continued)
- What is MLFQ and how is it used today?
- What does Linux do?

Round-Robin Scheduling

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

(a scheduling quantum)

Once time slice over,

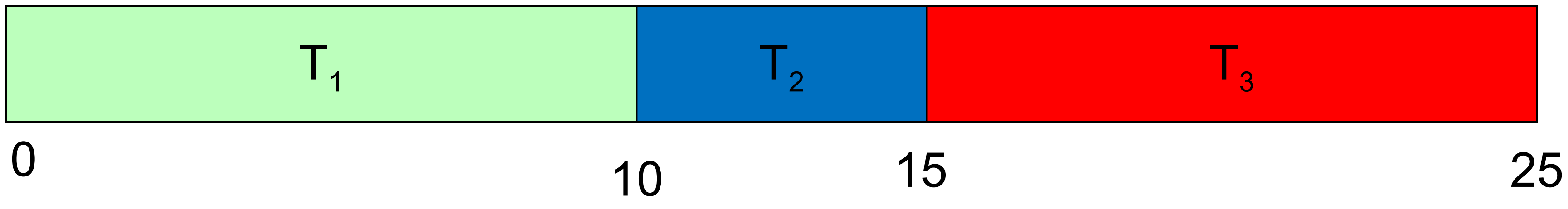
Switch to next job in ready queue.

=> Called time-slicing

Decrease Completion Time

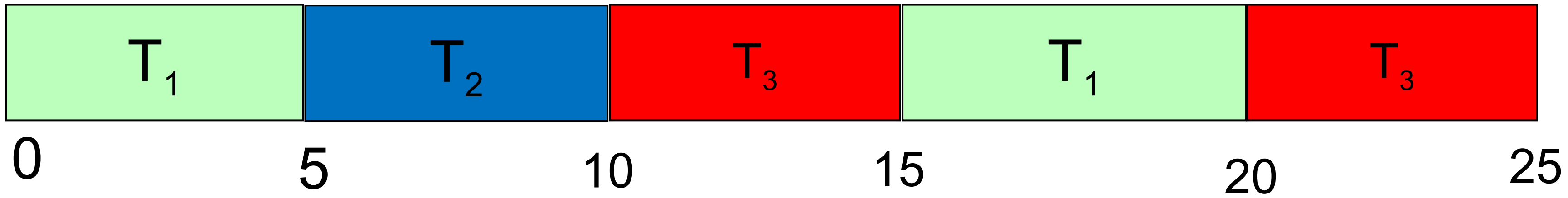
- T_1: Burst Length 10 T_3: Burst Length 10
- T_2: Burst Length 5

$Q = 10$



Average Completion Time = (10 + 15 + 25)/3 = 16.7

$Q = 5$



Average Completion Time = (20 + 10 + 25)/3 = 18.3

Switching is not free!

Small scheduling quantas lead to
frequent context switches
- Mode 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

No process waits more than $(n\text{-}1)q$ time units

Are we done?

Can RR suffer from convoy effect?

No

Only run a time-slice at a time

RR Summary

The good

Bounded response time

The bad

Completion time can be high
(stretches out long jobs)

The ugly

Overhead of context switching

Taking a step back

Property	FCFS	SJF	STCF	RR
Optimise Average Completion Time		✓	✓	
Optimise Average Response Time				✓
Prevent Starvation	✓			✓
Prevent Convoy Effect			✓	✓
Psychic Skills Not Needed	✓			✓

FCFS and Round Robin Showdown

Assuming zero-cost context-switching time,
is RR always better than FCFS?

10 jobs, each take 100s of CPU time
RR scheduler quantum of 1s
All jobs start at the same time

Job #	FIFO	RR
1	100	991
2	200	992
…	…	…
9	900	999
10	1000	1000

Earlier Example with Different Time Quantum

Best FCFS:

P_2 [8]	P_4 [24]	P_1 [53]	P_3 [68]

0 8 32 85 153

Quantum	P1	P2	P3	P4	Average
Best FCFS	85	8	16	32	69.5
Q=1	137	30	153	81	100.5
Q=5	135	28	153	82	99.5
Q=8	133	16	153	80	99,5
Q=10	135	18	153	92	104.5
Q=20	125	28	153	112	104.5
Worst FCFS	121	153	68	145	121.75

RR Summary

The good

Bounded wait time

The bad

Completion time can be high (stretches out long jobs)

The ugly

Overhead of context switching

Recall: Workload Assumptions

A workload is a set of tasks for some system to perform, including how long tasks last and when they arrive

Compute-Bound

Tasks that primarily perform compute

Fully utilise CPU

IO Bound

Mostly wait for IO, limited compute

Often in the Blocked state

RR & IO

RR performs poorly when running mix of
IO and Compute tasks

IO tasks need to run "immediately" for a short duration of time (low waiting time).

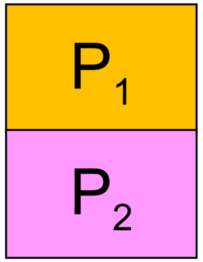
P_1 Computes for 1 ms. Uses disk for 10 ms

P_2 Computes for 50 ms.

RR & IO

Time Quanta:

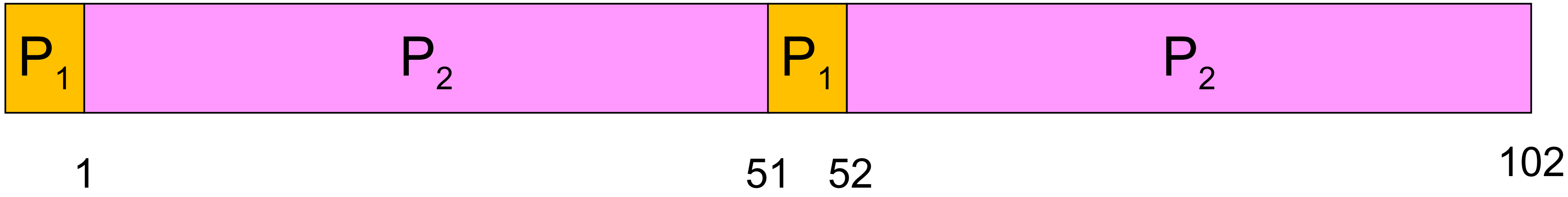
50 ms



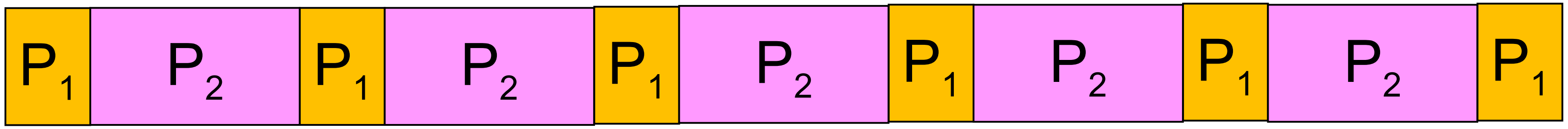
Computes for 1 ms. Uses disk for 10 ms

Computes for 50 ms.

Current Schedule



Optimal Schedule



What we want

1) Minimise average waiting time for IO/interactive tasks
(tasks with short CPU bursts)

2) Miminise average completion time

3) Maximise throughput
(includes minimizing context switches)

4) Remain fair/starvation-free

A side note: priorities

Some jobs are more important than others

Should be scheduled first.
Should get a larger share of the CPU

Assign each job with a priority

A side note: priorities

nice(2) — Linux manual page

NAME | SYNOPSIS | DESCRIPTION | RETURN VALUE | ERRORS | CONFORMING TO | NOTES | SEE ALSO | COLOPHON

Search online pages

```
NICE(2)                 Linux Programmer's Manual                NICE(2)
```

NAME top

```
       nice - change process priority
```

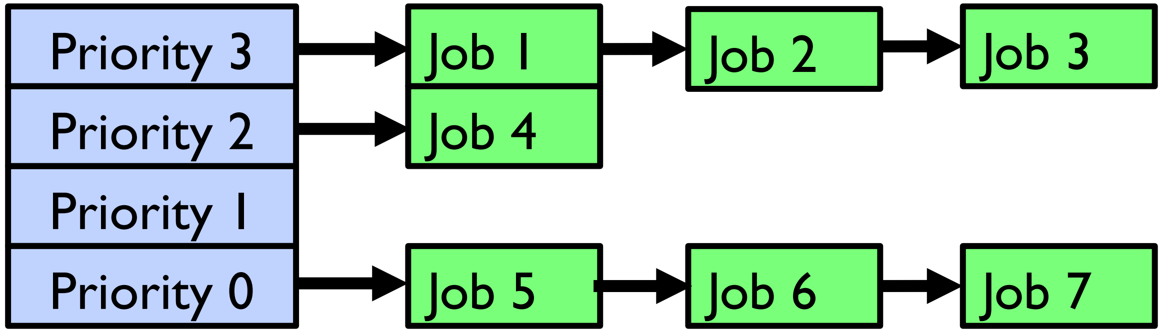
SYNOPSIS top

```
       #include <unistd.h>

       int nice(int inc);

   Feature Test Macro Requirements for glibc (see
   feature_test_macros(7)):

       nice():
           _XOPEN_SOURCE
               || /* Since glibc 2.19: */ _DEFAULT_SOURCE
               || /* Glibc <= 2.19: */ _BSD_SOURCE || _SVID_SOURCE
```

DESCRIPTION top

```
       nice() adds inc to the nice value for the calling thread.  (A
       higher nice value means a lower priority.)
```

Strict Priority Scheduling



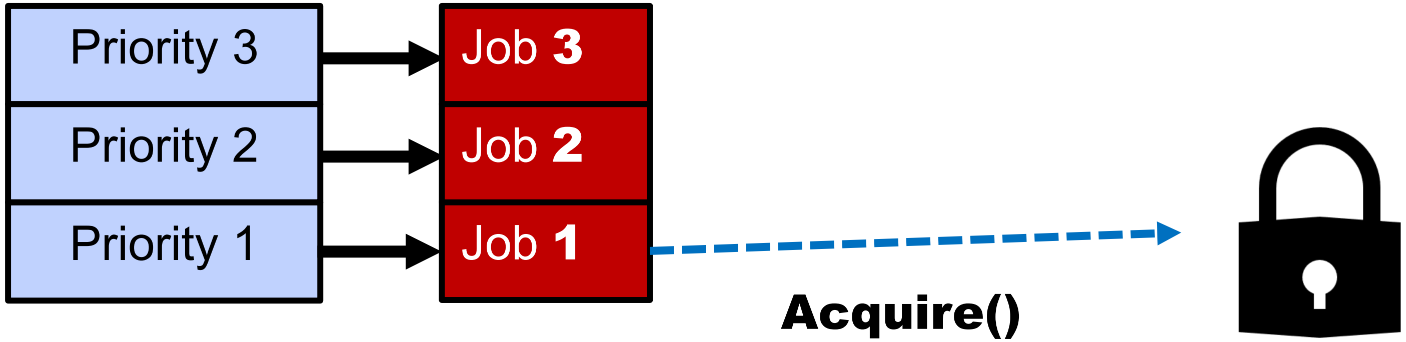
Split jobs by priority into n different queues.

Always process highest-priority queue if not empty. Process each queue round-robin.

Does this lead to starvation?

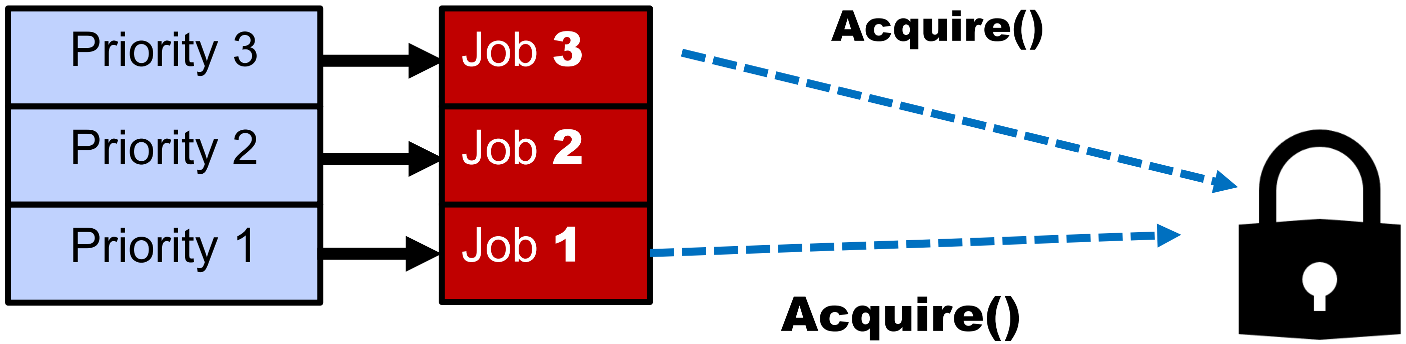
Priority Inversion

A high–priority thread can become starved by waiting on a low priority thread to release a resource that the high priority thread needs to make progress



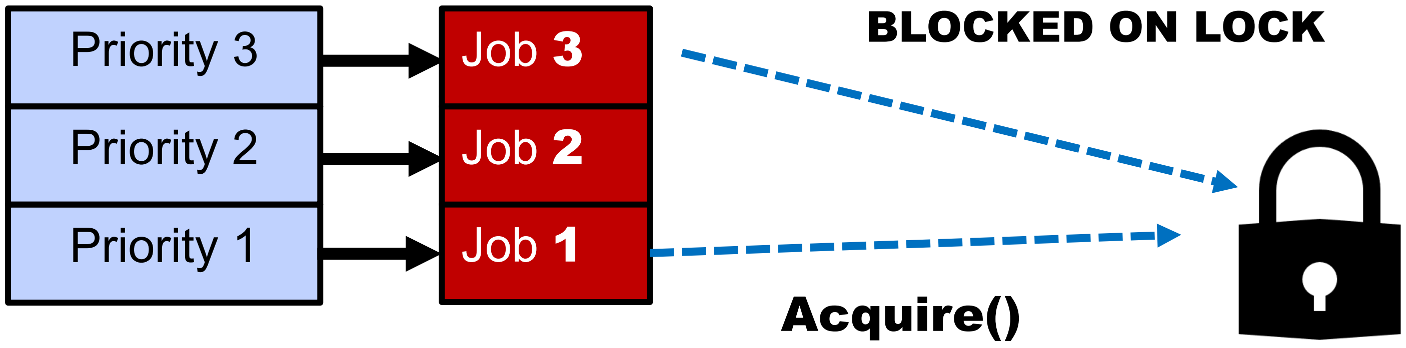
Priority Inversion

A high–priority thread can become starved by waiting on a low priority thread to release a resource that the high priority thread needs to make progress



Priority Inversion

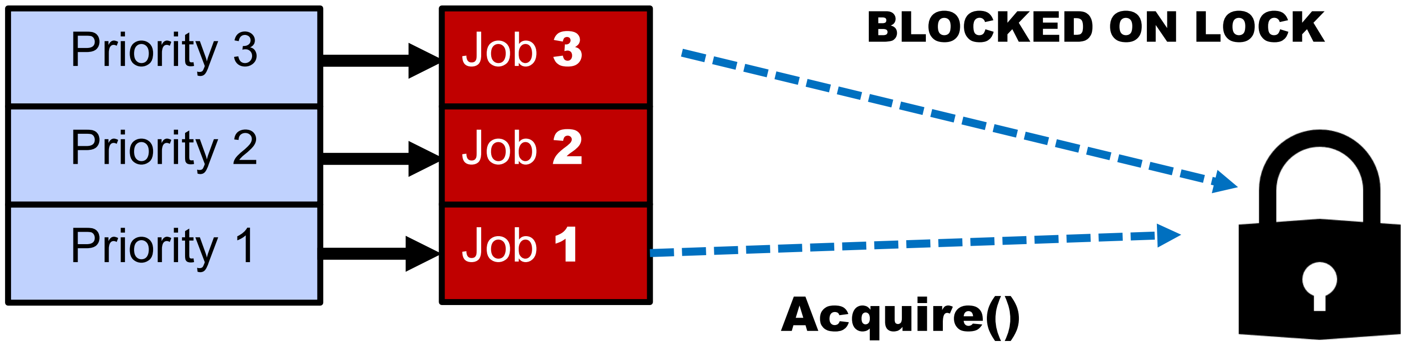
A high–priority thread can become starved by waiting on a low priority thread to release a resource that the high priority thread needs to make progress



Schedule Job 2 instead.

Priority Inversion

A high–priority thread can become starved by waiting on a low priority thread to release a resource that the high priority thread needs to make progress



Keeps scheduling Job 2 over Job 1, Job 3 never runs!

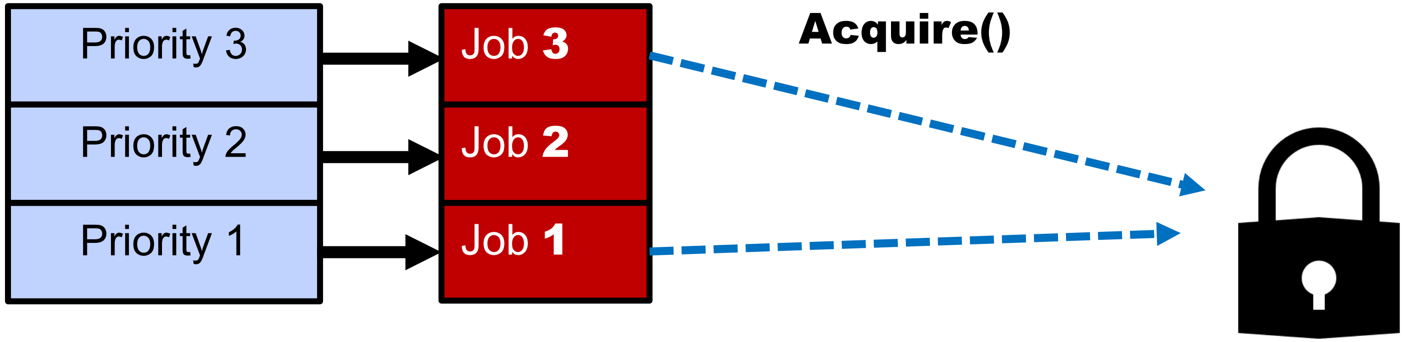
Priority Inversion

Where high priority task is blocked waiting
on low priority task

Low priority one ***must*** run for high priority to make progress

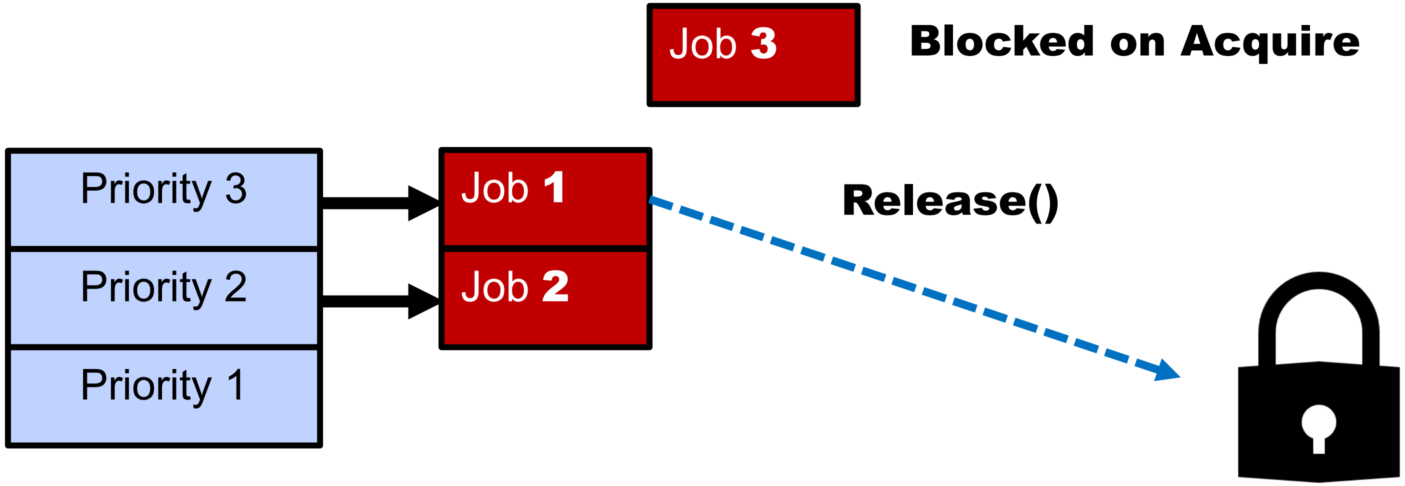
Medium priority task can starve a high priority one

One Solution: Priority Donation/Inheritance



Job 3 temporarily grants Job 1 its “high priority” to run on its behalf

One Solution: Priority Donation/Inheritance



Job 3 temporarily grants Job 1 its “high priority” to run on its behalf

Case Study: Martian Pathfinder Rover

July 4, 1997 – Pathfinder lands on Mars
- First US Mars landing since Vikings in 1976; first rover

And then…a few days into mission…:
- System would reboot randomly, losing valuable time and progress

Problem? Priority Inversion!
- Low priority task grabs mutex trying to communicate with high priority task:
- Realtime watchdog detected lack of forward progress and invoked reset to safe state

Recall: What we want

1) Minimise average waiting time for IO/interactive tasks
(tasks with short CPU bursts)

2) Miminise average completion time

3) Maximise throughput
(includes minimizing context switches)

4) Remain fair/starvation-free

Recall: STCF

Schedule jobs in order of shortest completion time

Requires knowledge of job completion time

⇩

Approximate duration of CPU burst; encode it in priorities

Subject to Starvation

⇩

Dynamically adapt priorities

Introducing the Multi-level Feedback Queue

Create distinct **queues for ready jobs**, each assigned a different **priority level**.

All jobs belong to one queue at a time. Jobs can move between queues.

MLFQ uses priorities to decide from which queue it should pick next job.

Individual queues run RR with increasing time quantas

MLFQ (V 1.0)

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 behaviour*

Use the *history* of the job to predict its *future* behaviour

Learning behaviour

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 behaviour

Where do IO-bound/interactive jobs end up?

a) Top Queue b) Bottom Queue

MLQF emulates STCF: short jobs given higher priorities than long jobs.

First assumes all jobs are short. If jobs finish < time quanta, assume IO-bound, otherwise CPU bound

Learning behaviour

P_1 Computes for 1 ms. Uses disk for 10 ms

P_2 Computes for 50 ms.

q= 2 ms

q= 10 ms

q= 100 ms

Schedule

Learning behaviour

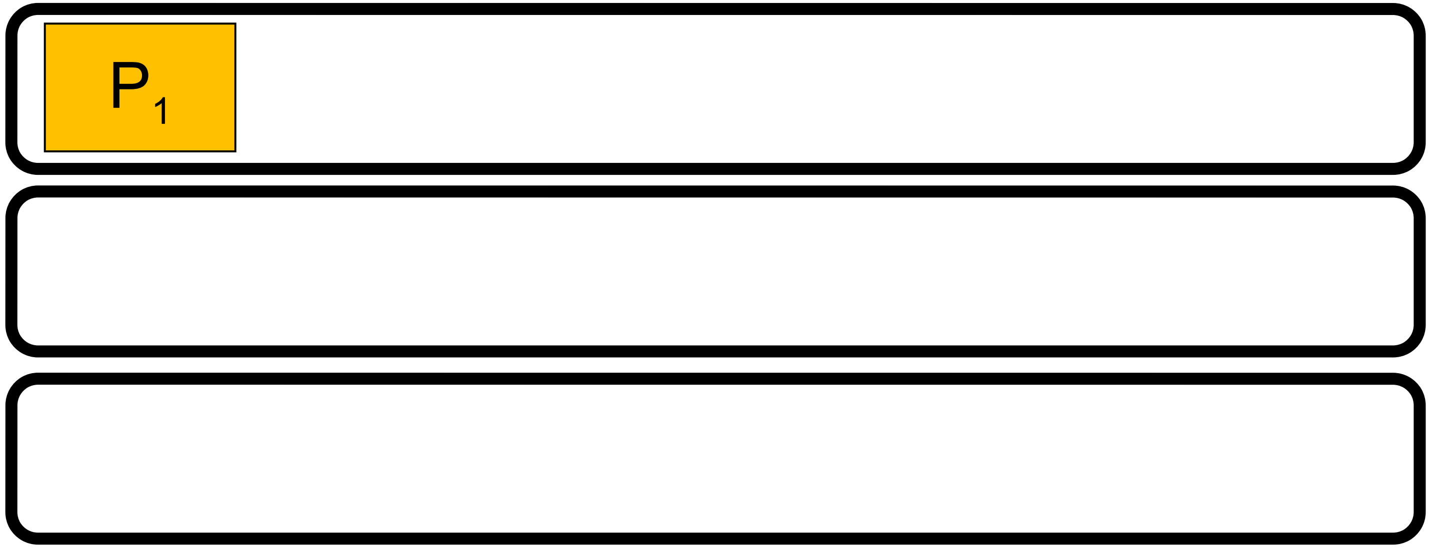
P_1 Computes for 1 ms. Uses disk for 10 ms

P_2 Computes for 50 ms.

q= 2 ms

q= 10 ms

q= 100 ms



Schedule

Learning behaviour

Process	Behaviour
P_1	Computes for 1 ms. Uses disk for 10 ms
P_2	Computes for 50 ms.

q= 2 ms

q= 10 ms

q= 100 ms

Schedule P_1

Learning behaviour

P_1 Computes for 1 ms. Uses disk for 10 ms

P_2 Computes for 50 ms.

q= 2 ms P_1

q= 10 ms

q= 100 ms

Schedule P_1

Learning behaviour

P_1 Computes for 1 ms. Uses disk for 10 ms

P_2 Computes for 50 ms.

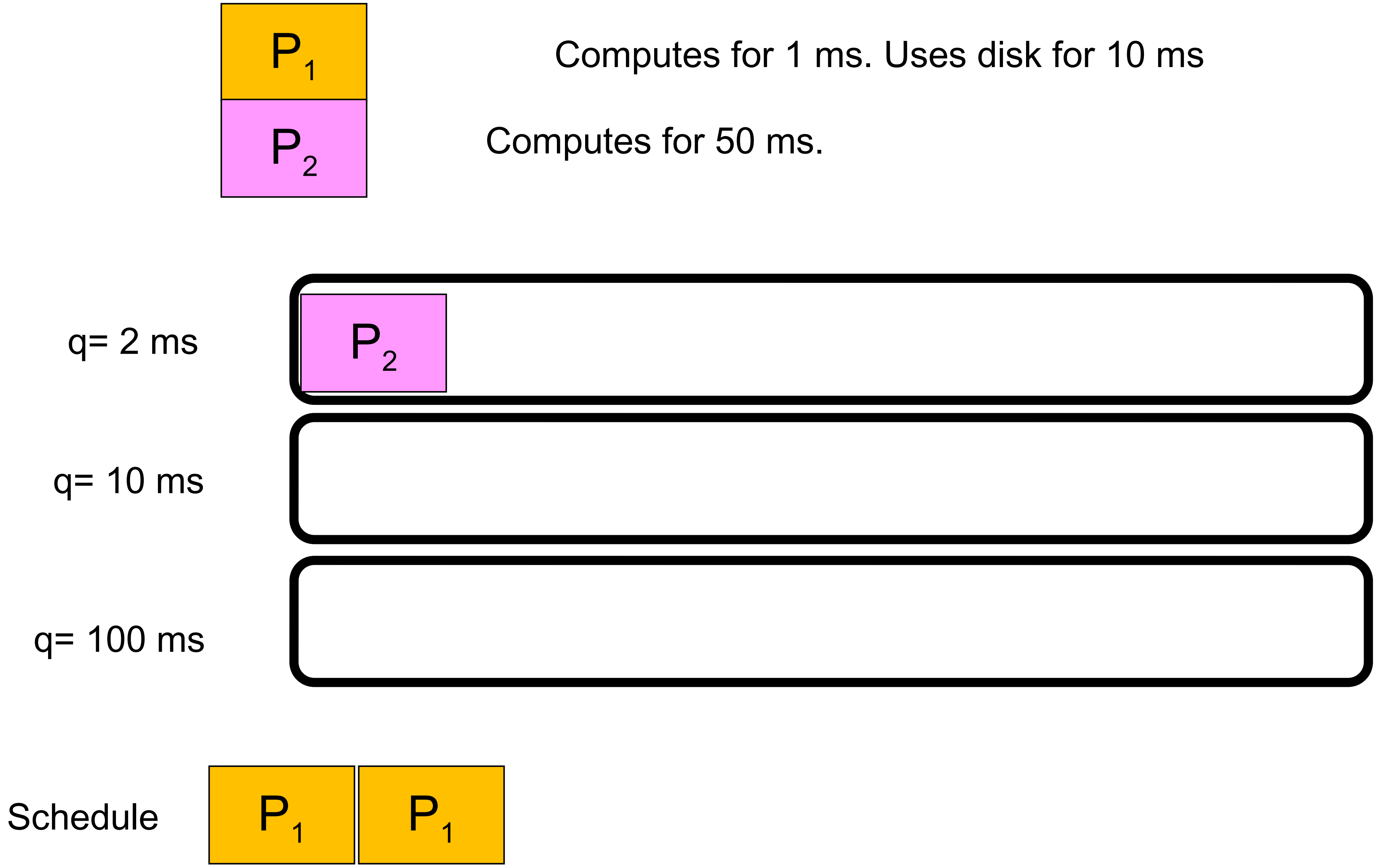
q= 2 ms P_1 P_2

q= 10 ms

q= 100 ms

Schedule P_1

Learning behaviour



Learning behaviour

P_1 Computes for 1 ms. Uses disk for 10 ms

P_2 Computes for 50 ms.

q= 2 ms

q= 10 ms

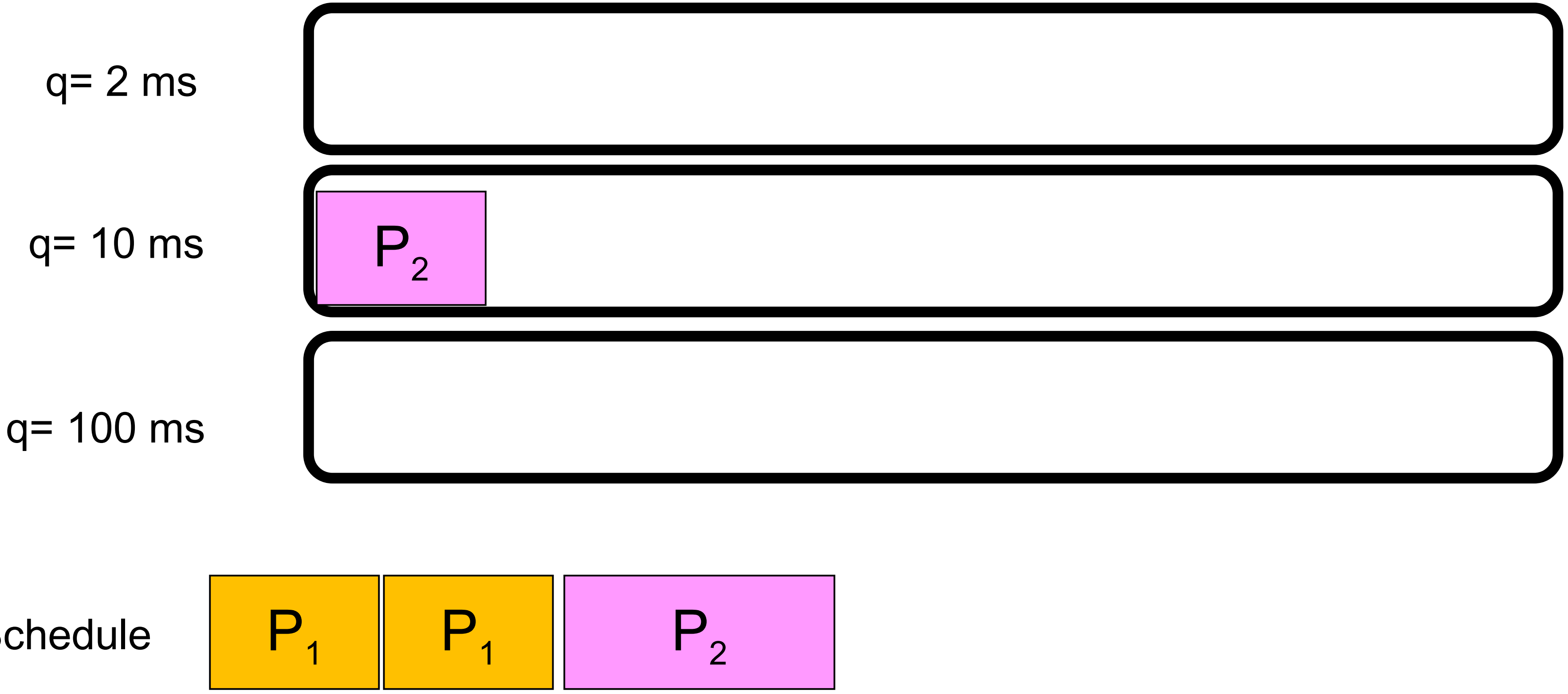
q= 100 ms

Schedule P_1 P_1 P_2

Learning behaviour

P_1 Computes for 1 ms. Uses disk for 10 ms

P_2 Computes for 50 ms.



Learning behaviour

P_1 Computes for 1 ms. Uses disk for 10 ms

P_2 Computes for 50 ms.

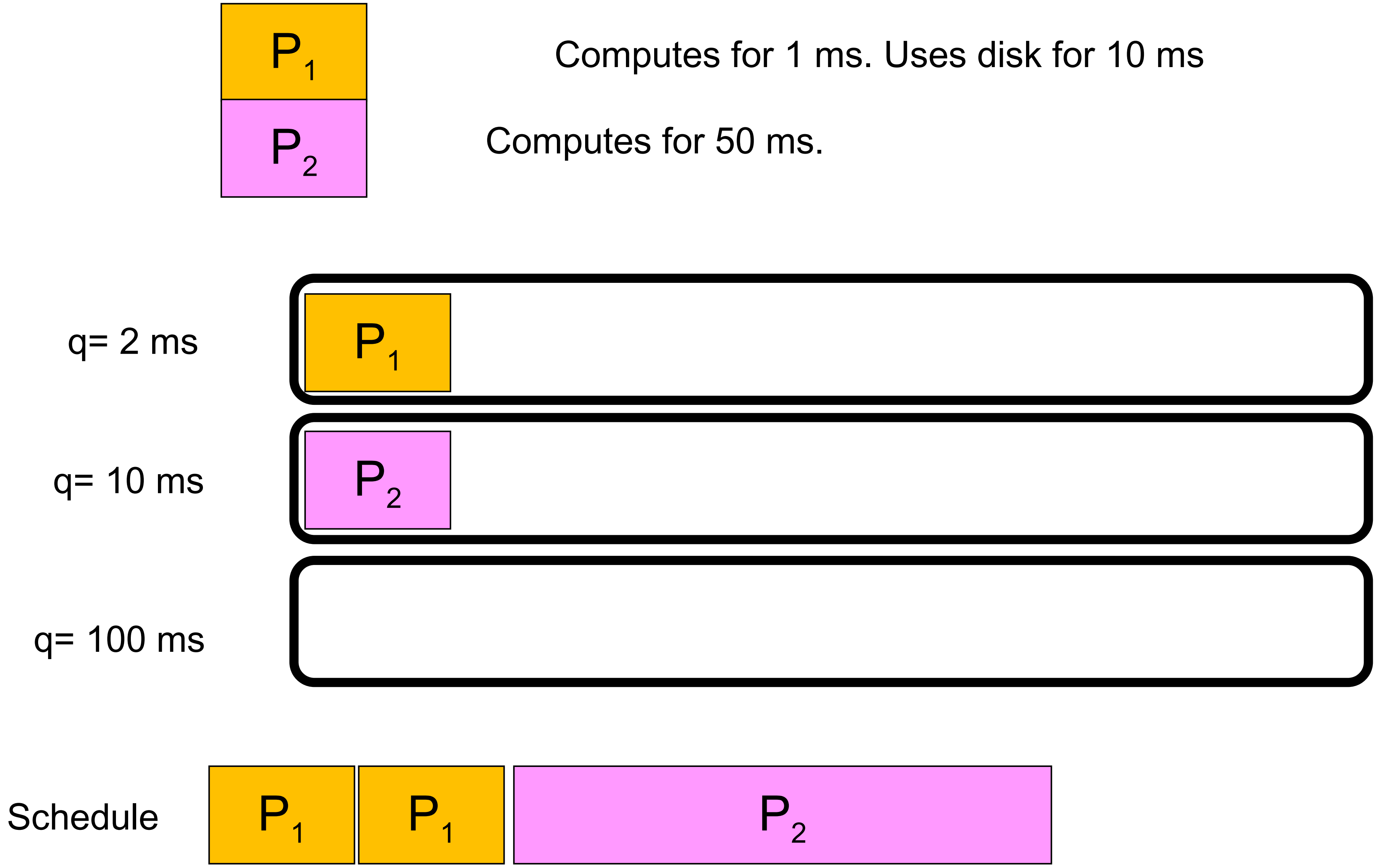
q= 2 ms

q= 10 ms

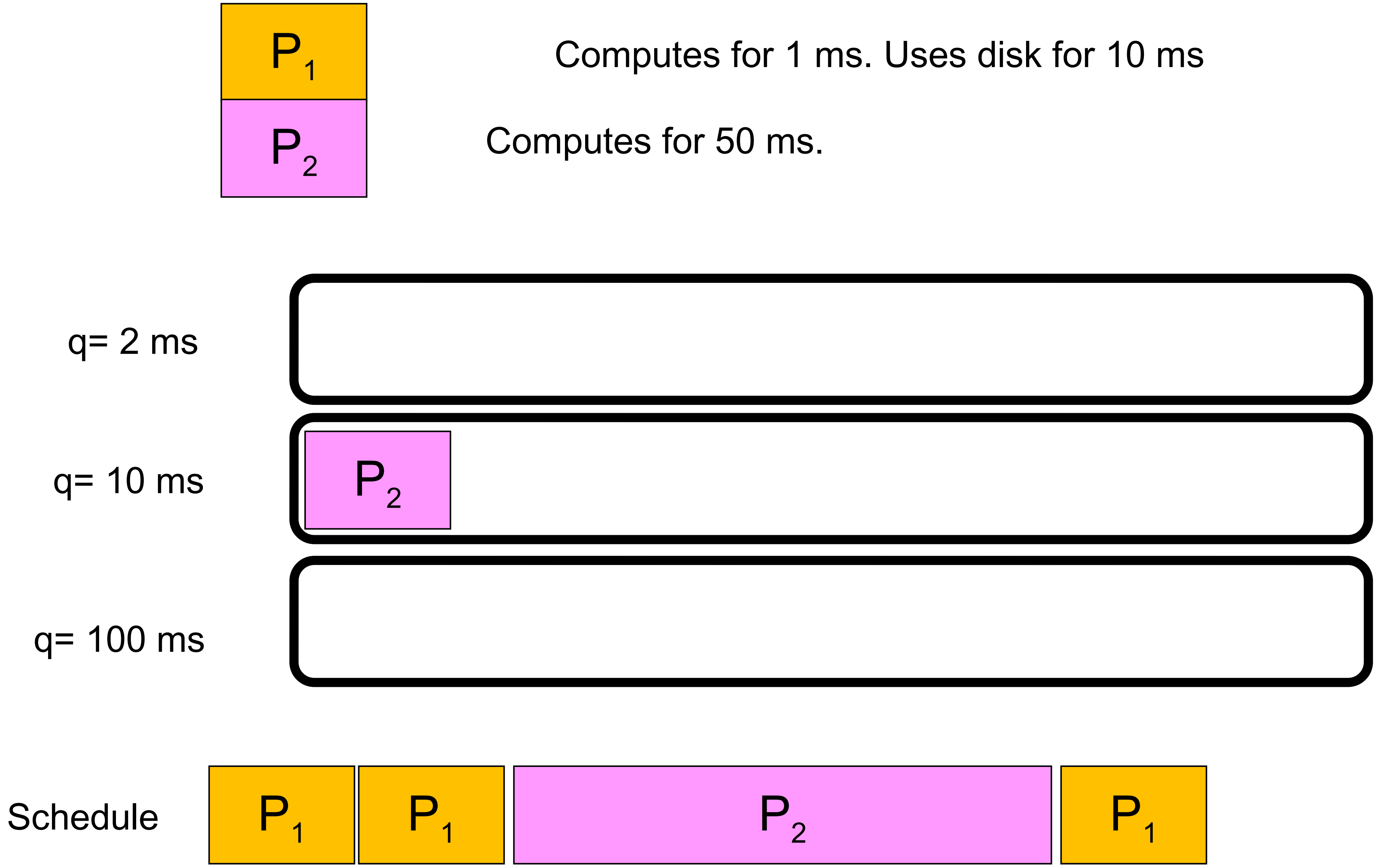
q= 100 ms

Schedule P_1 P_1 P_2

Learning behaviour



Learning behaviour



Learning behaviour

P_1	Computes for 1 ms. Uses disk for 10 ms
P_2	Computes for 50 ms.

q= 2 ms

q= 10 ms

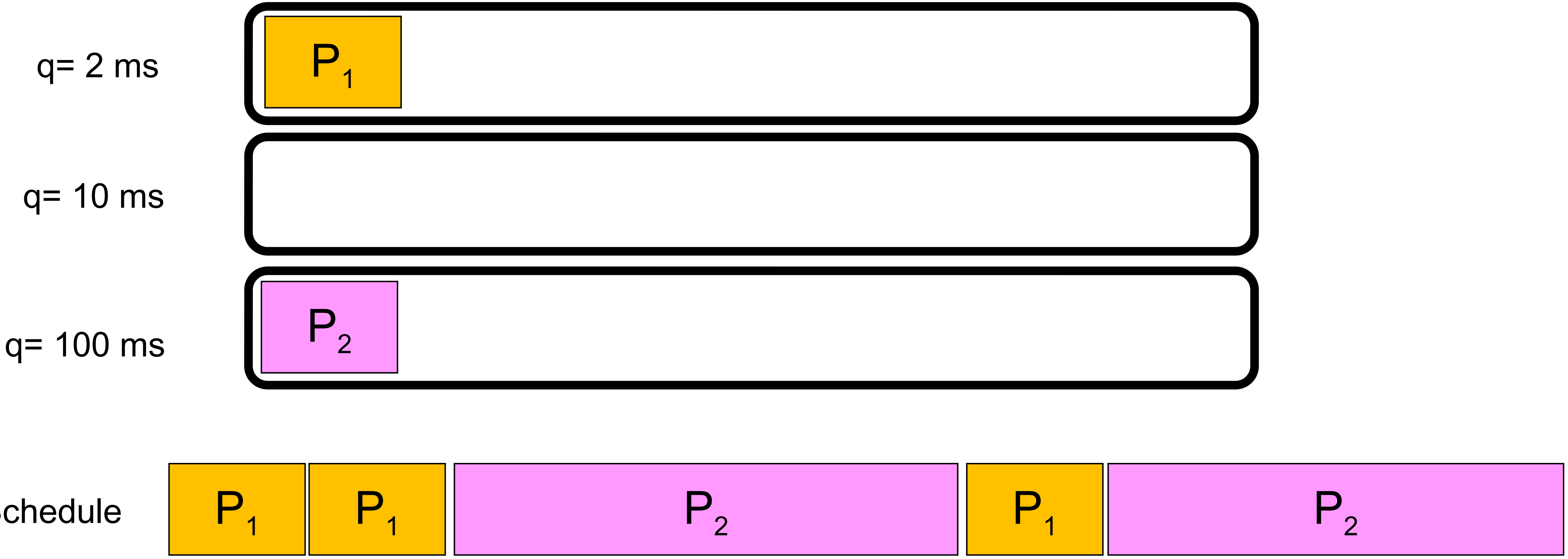
q= 100 ms

Schedule	P_1	P_1	P_2	P_1	P_2

Learning behaviour

P_1 Computes for 1 ms. Uses disk for 10 ms

P_2 Computes for 50 ms.



Are we done?

MLQF can be gamed:

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

The “Othello” strategy

MLQF is subject to starvation:

Systematically prioritise higher-priority queues

Are we done?

MLQF can be gamed:

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

The "Othello" strategy

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 prioritise higher-priority queues

Rule 5

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

MLFQ

Rule 1

If Priority(A) > Priority(B), A runs (B doesn’t).

Rule 2

If Priority(A) = Priority(B), A & B run RR using quantum of queue.

Rule 3

A new job is placed in the topmost queue.

Rule 4

Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced.

Rule 5

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

Many many different variants of MLQF

Change how prevent starvation

Change constants

Change scheduling policies within each queue

Most modern schedulers are variants of MLQF queues

History of Schedulers in Linux

O(n) scheduler

Linux 2.4 to Linux 2.6

O(1) scheduler

Linux 2.6 to 2.6.22

CFS scheduler

Linux 2.6.23 onwards

Case Study: Linux O(n) Scheduler

At every context switch:

- Scan full list of processes in the ready queue
- Compute relevant priorities
- Select the best process to run

Scalability issues:

- Context switch cost increases as number of processes increase
- Single queue even in multicore systems

Case Study: Linux O(1) Scheduler

Next process to run is chosen in constant time

Priority-based scheduler with 140 different priorities

Real-time/kernel tasks assigned priorities 0 to 99 (0 is highest priority)

User tasks (interactive/batch) assigned priorities 100 to 139 (100 is highest priority)

Case Study: O(1) Scheduler – User tasks

Per priority-level, each CPU has **two ready queues**

An active queue, for processes which have not used up their time quanta

An expired queue, for processes who have

Timeslices/priorities/interactivity credits all computed when jobs finishes timeslice

Timeslice depends on priority

User tasks – Priority Adjustment

User-task priority adjusted ±5 based on heuristics

» p->sleep_avg = sleep_time – run_time

» Higher sleep_avg ⇒ more I/O bound the task, more reward (and vice versa)

Interactive Credit

» Earned when a task sleeps for a “long” time

» Spend when a task runs for a “long” time

» IC is used to provide hysteresis to avoid changing interactivity for temporary changes in behavior

However, “interactive tasks” get special dispensation

» To try to maintain interactivity

» Placed back into active queue, unless some other task has been starved for too long…

O(1) Scheduler – Real tasks

Real-Time Tasks always preempt non-RT tasks

No dynamic adjustment of priorities

Scheduling schemes:

- » SCHED_FIFO: preempts other tasks, no timeslice limit
- » SCHED_RR: preempts normal tasks, RR scheduling amongst tasks of same priority

An aside: Real-Time Scheduling

Goal

Predictability of Performance!

We need to predict with confidence worst case response times for systems!

Real-time is about enforcing predictability,
and does not equal fast computing.