Scheduling

Science

Computer Science

CS 450: Operating Systems

Michael Lee <lee@iit.edu>

ILLINOIS TECH | College of Computing

Agenda

- Overview
  - Not just one scheduler
- Scheduling metrics
  - “Interactive” jobs and responsiveness
- Scheduling policies
  - FCFS, SJF, PSJF, RR, HPRN
  - MLQ, MLFQ

§ Overview

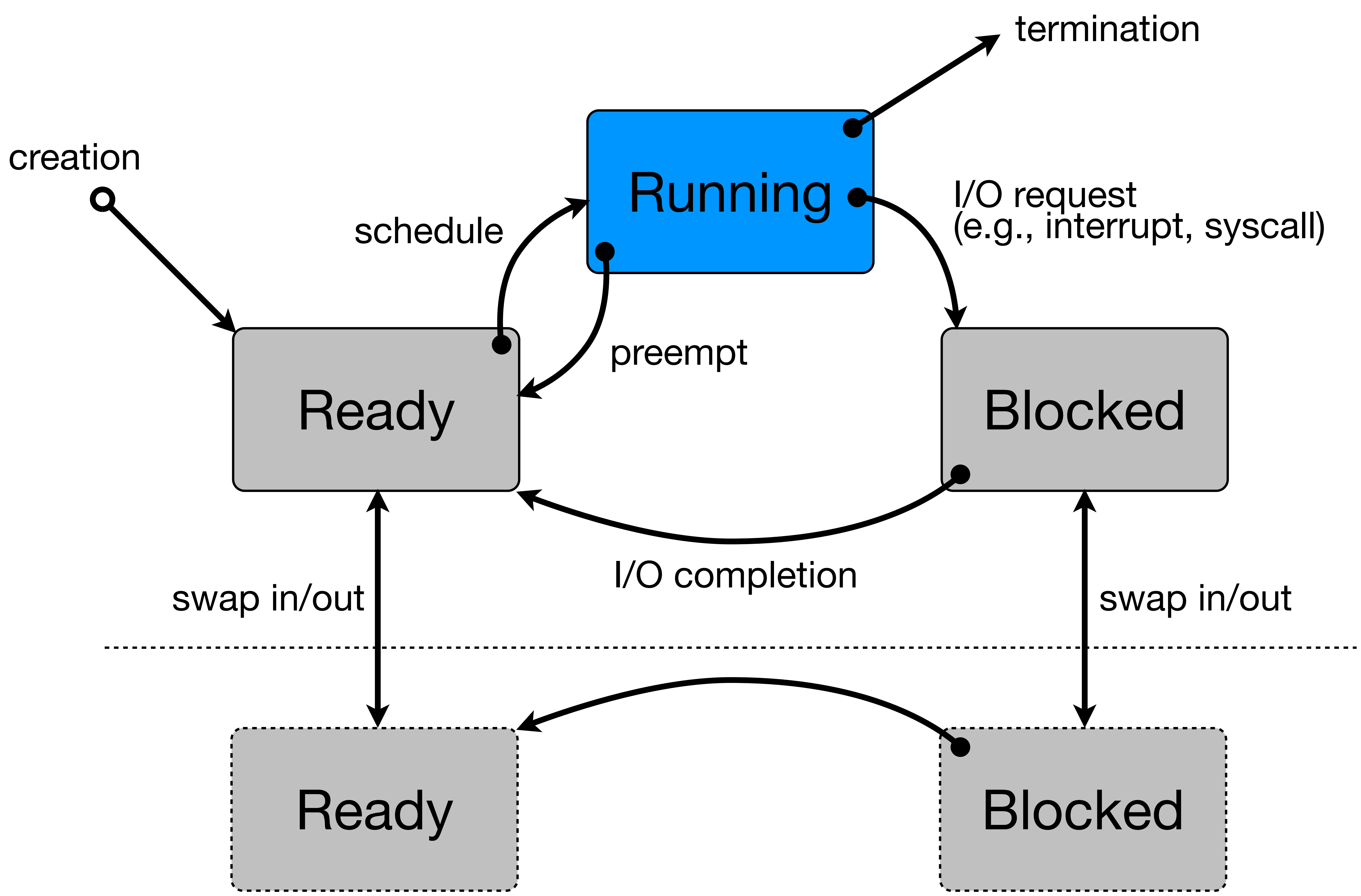
Definition

- Scheduling: **policies** & **mechanisms** used to allocate **limited resources** to some set of **entities**
- Initial focus: **resource** & **entities = CPU** & **processes** (aka **jobs**)
  - other possibilities:
    - **resources**: memory, I/O bus/devices
    - **entities**: threads, users, groups
    - schedulers for the above may exist in an OS (and must play nice with each other)!

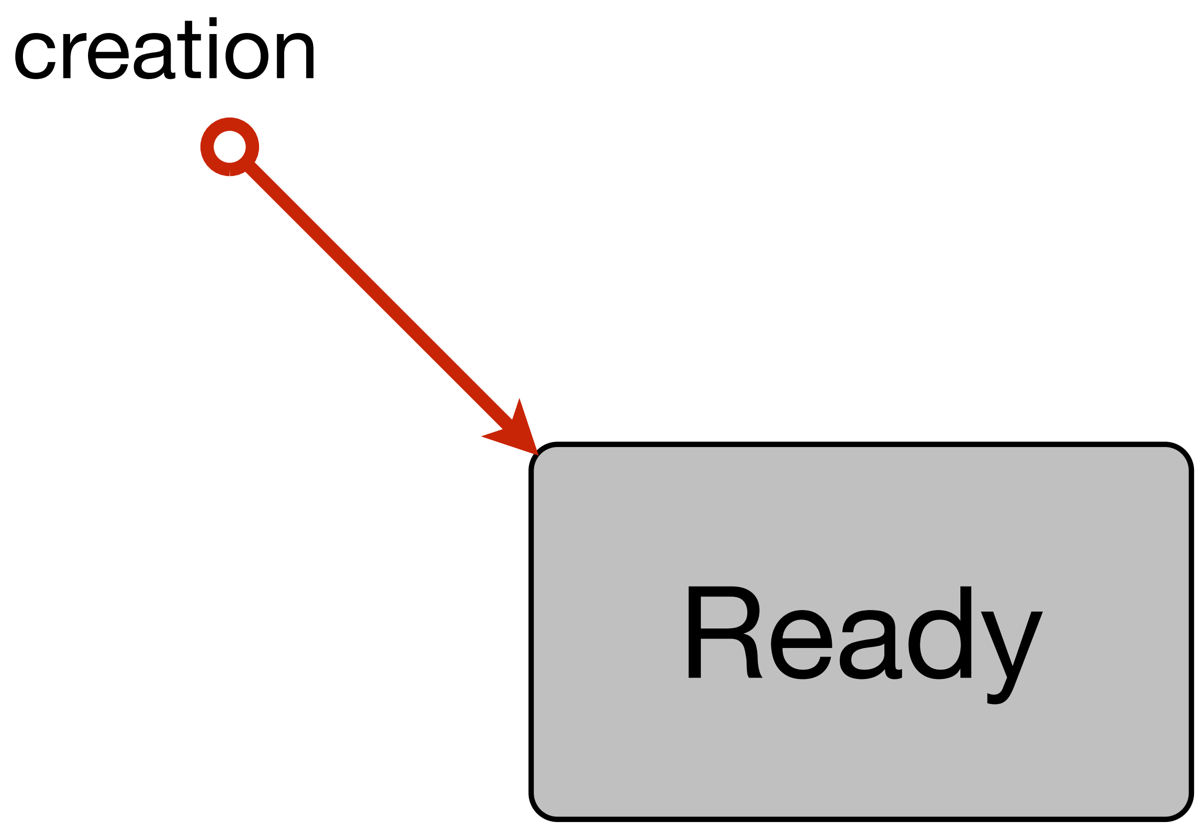
Policy vs. Mechanism

- high-level “what”
- scheduling disciplines
  - e.g., FCFS, SJF, RR, etc.
- driven by a variety of potentially conflicting goals
  - e.g., performance and fairness

- low-level “how”
- combination of HW/SW
  - e.g., clock interrupt, high precision timer, PCB
- scattered throughout kernel codebase

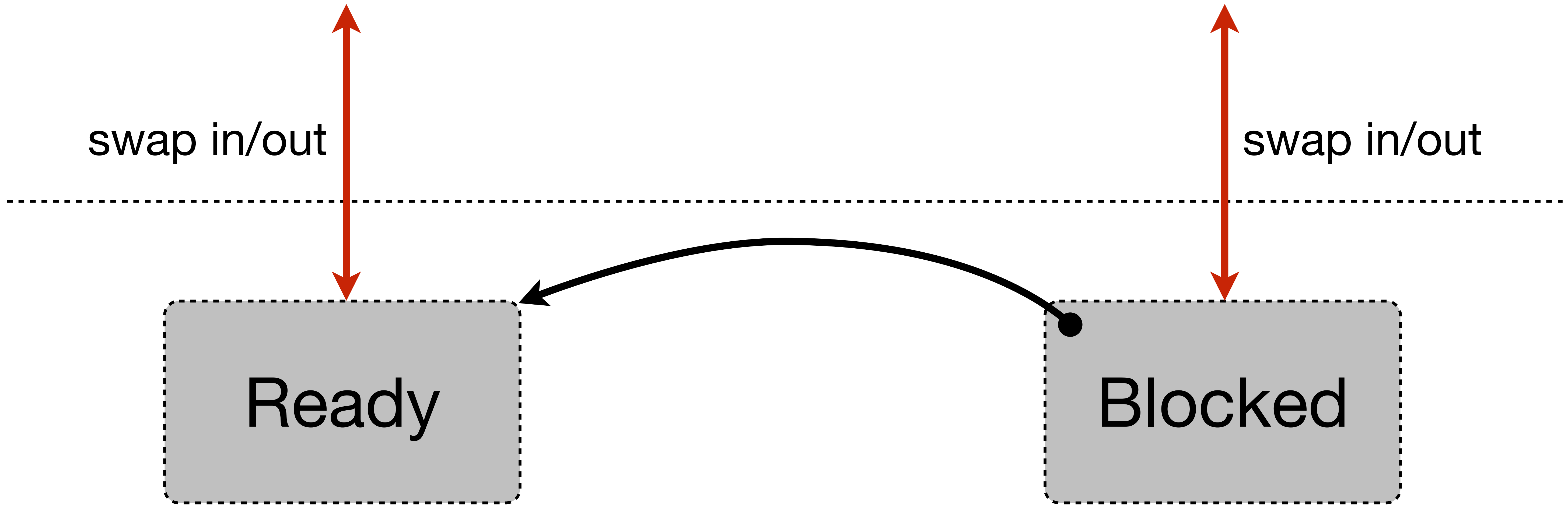


Schedulers are concerned with transitions between **process states**



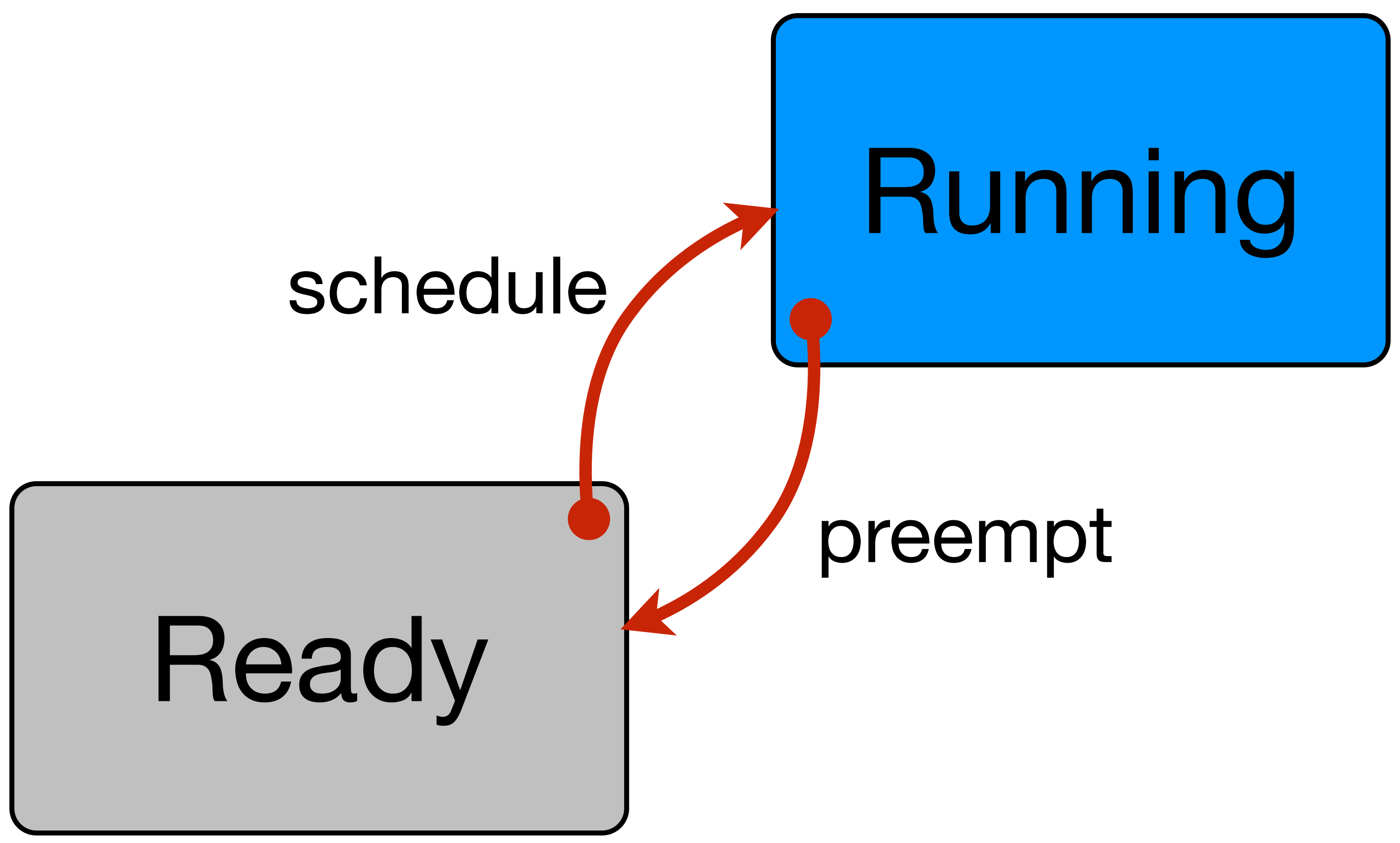
Domain of the “long-term” scheduler

- choose which jobs are admitted to the system
  - may control mix of jobs (e.g., I/O vs. CPU bound)
- not common in general-purpose, time-shared OSes



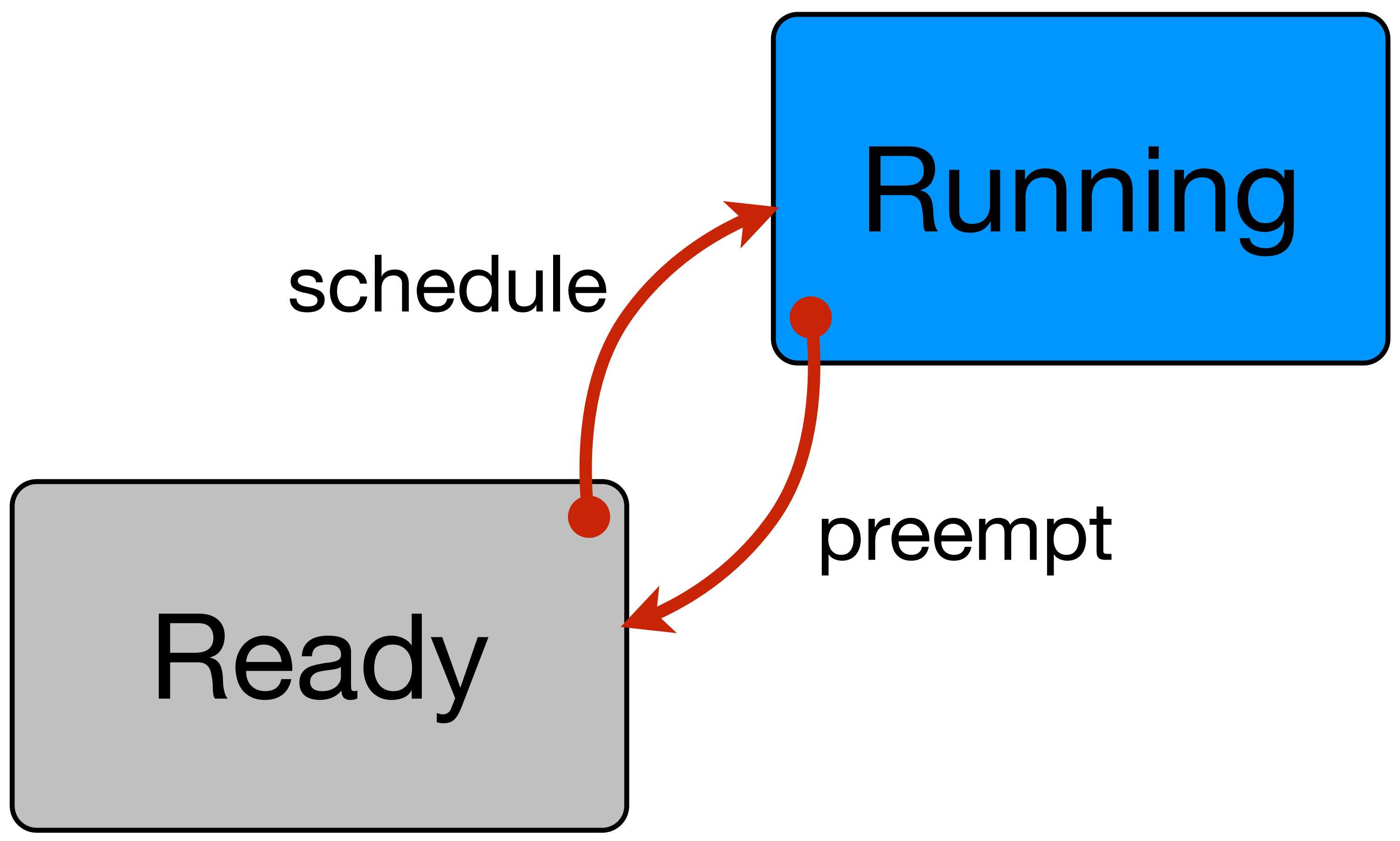
Domain of the "medium-term" scheduler

- swaps processes out to disk to make room for others
- active when there is insufficient memory
- runs much less frequently (slower!) than CPU scheduler



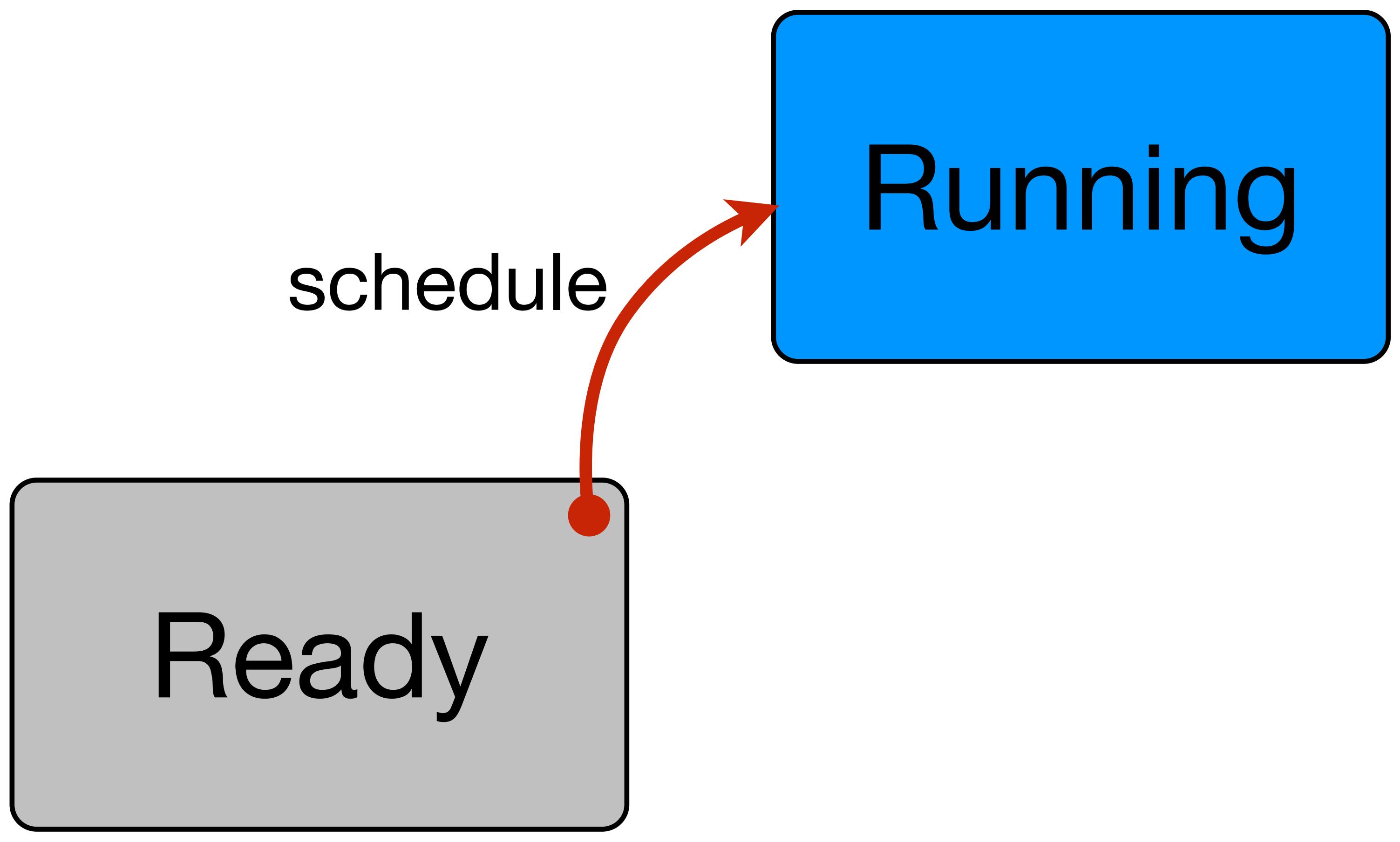
Domain of the “short-term” scheduler, i.e., the CPU scheduler

- chooses between in-memory, ready processes to run on CPU
- invoked to carry out scheduling policies after interrupts/traps



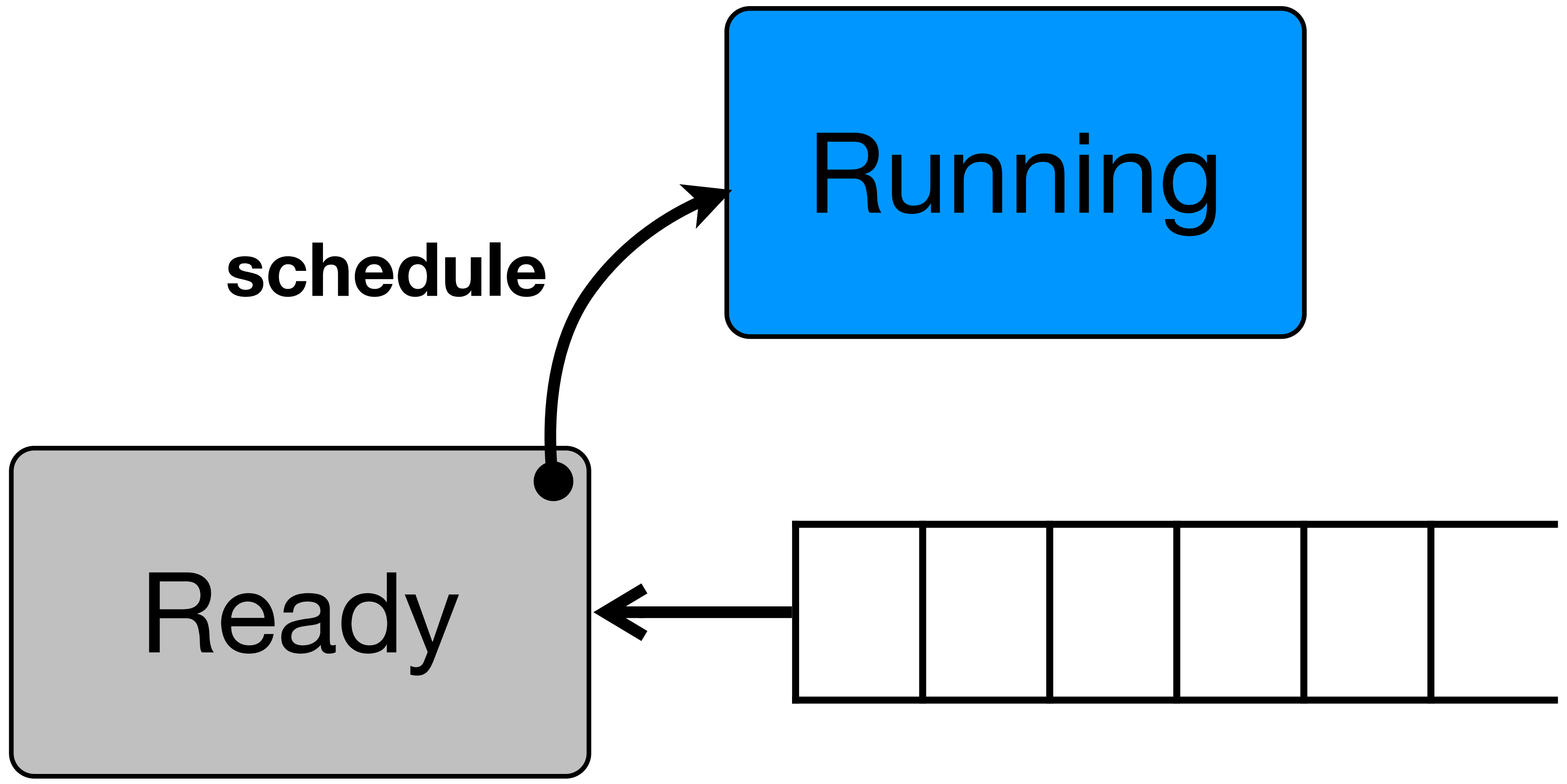
preemptive scheduling

relies on clock interrupt
(to regain control of CPU)



non-preemptive scheduling

once a job starts, it continues
until it terminates/blocks



- convenient to envision a **ready queue** (though not necessarily FIFO!)
- the **scheduling policy** decides which job to select from the set of ready (runnable) jobs to run next

High-level policy considerations

- Preemptive vs. Non-preemptive
- Information available for making informed decisions
  - Depends on lower-level mechanisms available
- Scheduling goals
  - Based on optimizing/tuning **scheduling metrics**

§ Scheduling Metrics

Some scheduling metrics

- Turnaround time
- Wait time
- Response time
- Throughput
- Utilization

Turnaround time

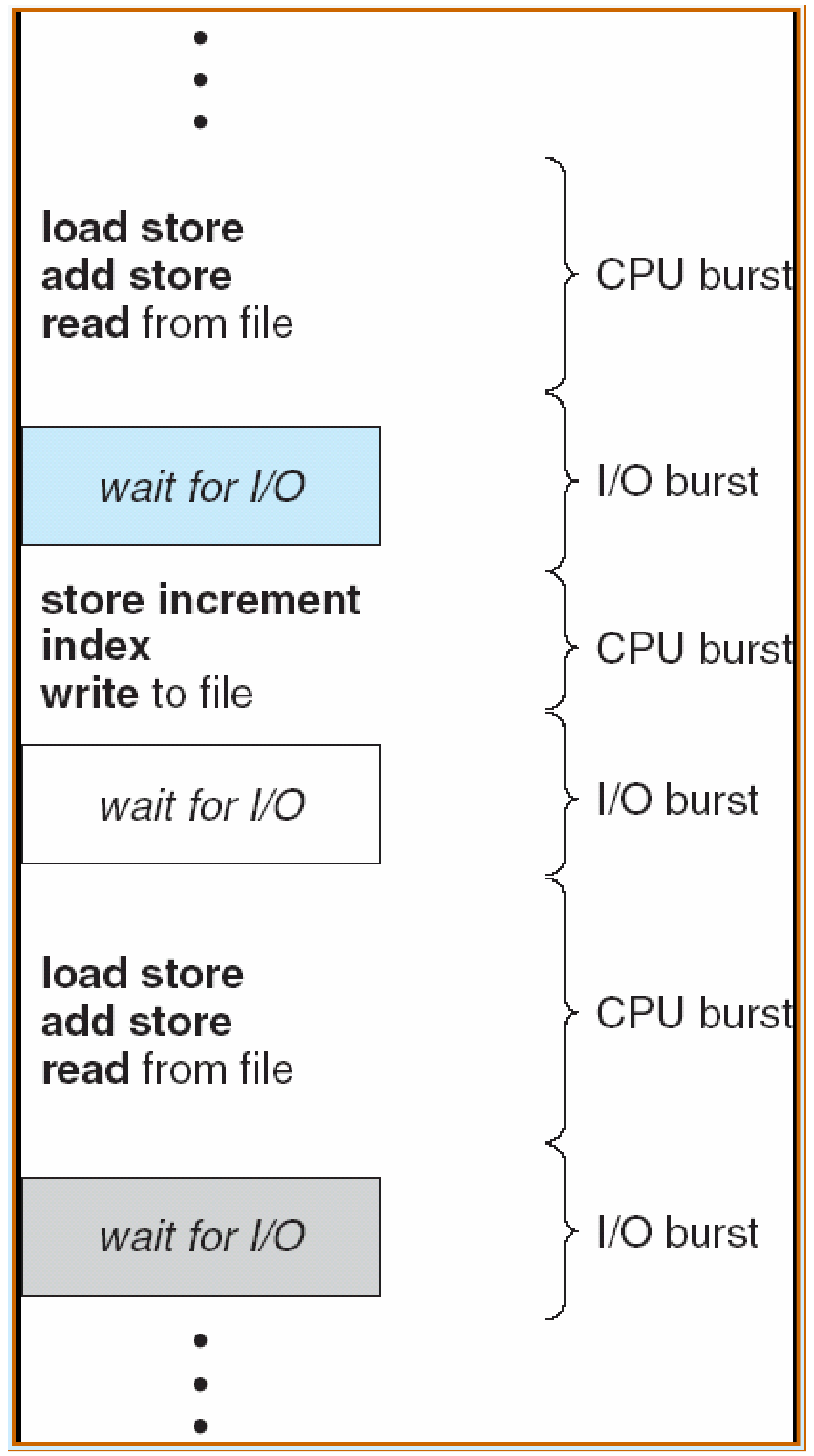
- $T_{turnaround} = T_{completion} - T_{creation}$
  - i.e., total time to complete job
- Useful metric for a CPU-bound process — how much time is required to carry out a lengthy computation?
- Not generally a great yardstick for evaluating a scheduler!
  - What if job is I/O-bound?
  - What if job never "completes"?

Wait time

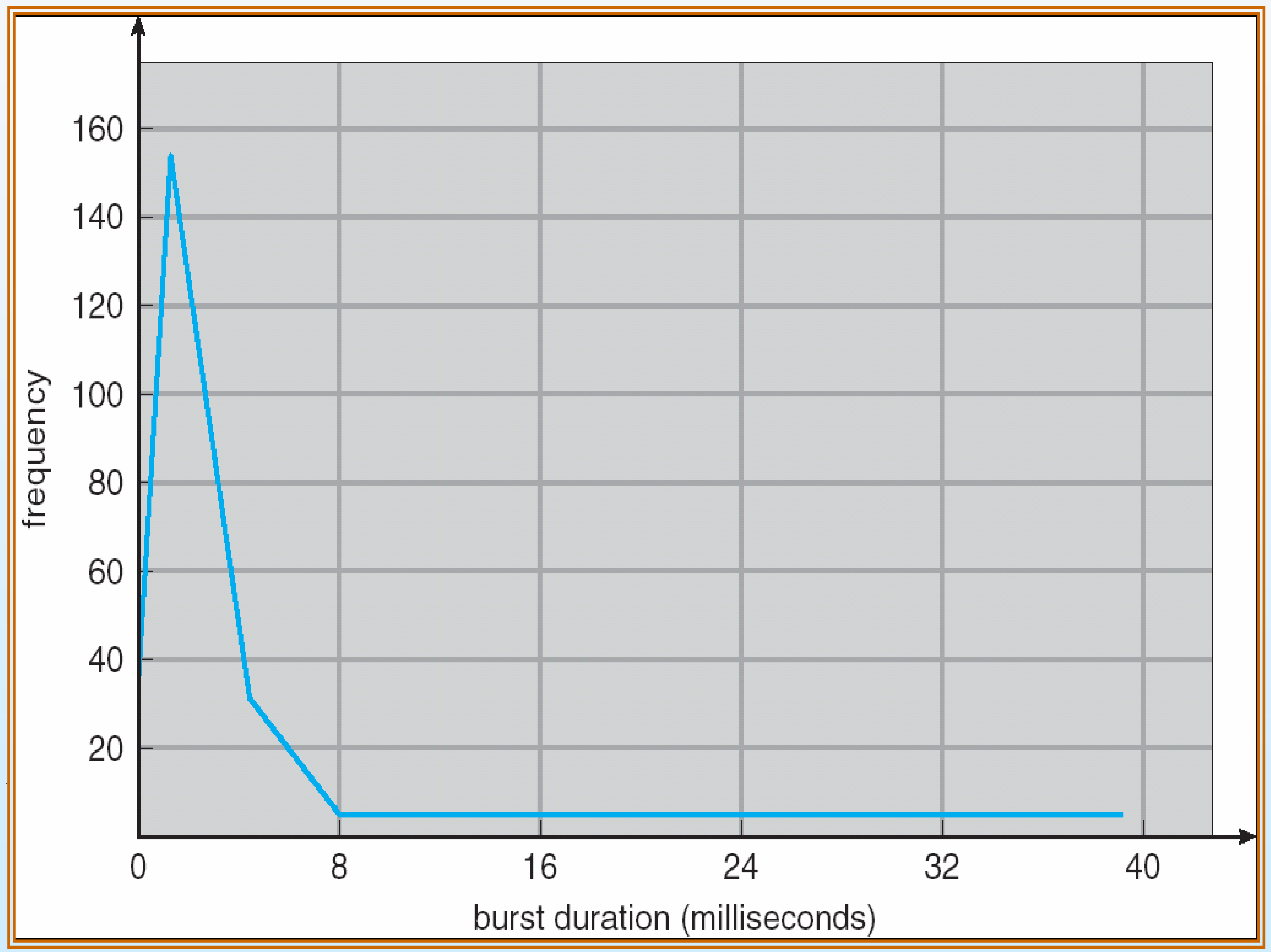
- Time spent in ready queue
  - i.e., *runnable*, but not actually running
    - CPU is busy doing other things
    - this is not an ideal state for a process!
- Minimizing wait time is a possible goal for a scheduling policy

Interactive processes

- Turnaround & Wait time may be measured over the *entire course* of a job
- Not a very relevant metric for interactive processes! (why?)
  - Interactive jobs have “bursty” execution — alternate between bursts of CPU and I/O activity
    - May never terminate! (e.g., consider browser, email client, etc.)
- Can compute turnaround/wait times on a *per-burst* basis
  - i.e., how long does a burst (of CPU activity) need to complete/wait before getting to the next I/O burst?



“bursty” execution



burst length histogram

“Responsiveness"

- For interactive jobs, improving **responsiveness** is arguably more important than optimizing total turnaround/wait times
  - How to quantify this?
- Response time: $T_{response} = T_{firstrun} - T_{arrival}$
  - i.e., how soon is a job given a chance to run after becoming ready?
  - What’s wrong with this? (consider requirements for “interaction”)
    - How might we improve this metric?

Throughout & Utilization

- **Aggregate** metrics
- **Throughput**: # of completed jobs or bursts per unit time
  - e.g., 5 processes / minute, 25 CPU bursts / second
- **Utilization**: percentage of time CPU is busy running jobs
  - Context switch time counts against utilization!
  - CPU can be idle if there are no active jobs or if all jobs are blocked

“Fairness”

- What does it mean?
- How to measure/quantify it?
- Is it useful?
- How to enforce it?
- Prioritizing fairness may lower performance — which is more important?

§ Scheduling Policies

First come first served (FCFS)

Process	***Arrival Time***	***Burst Time***
P_1	0	24
P_2	0	3
P_3	0	3

“Gantt chart”

Wait times: $P_1 = 0$, $P_2 = 24$, $P_3 = 27$

Average: $(0 + 24 + 27) / 3 = 17$

Convoy Effect

First come first served (FCFS)

Process	*Arrival Time*	*Burst Time*
P_3	0	3
P_2	0	3
P_1	0	24

(better for everyone) →

Wait times: $P_1 = 6$, $P_2 = 3$, $P_3 = 0$

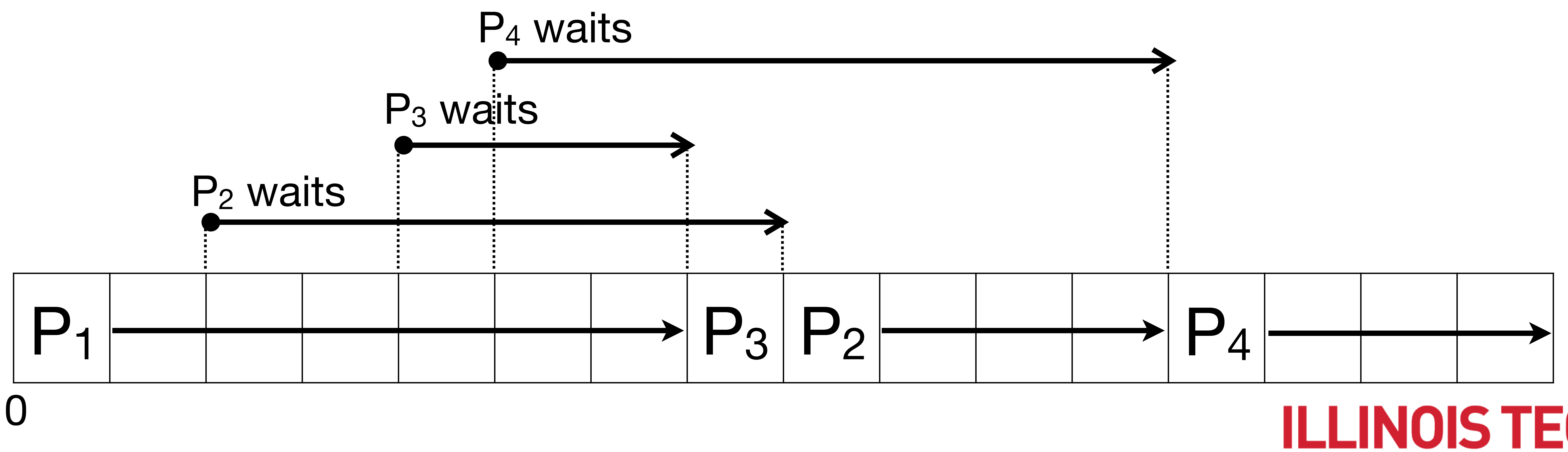
Average: $(6 + 3 + 0) / 3 = 3$

Shortest Job First (SJF)

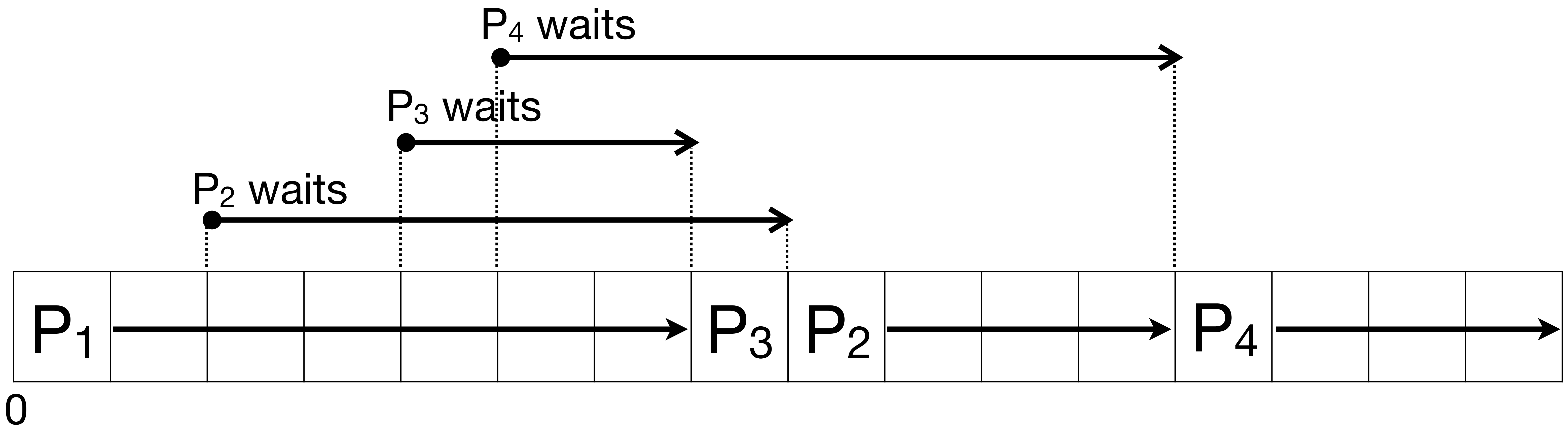
- “Obvious” improvement to FCFS
- What metric(s) are we improving?
- Still a non-preemptive policy — i.e., once a job starts executing a CPU burst, it runs until it blocks (or completes)

Shortest Job First (SJF)

Process	***Arrival Time***	***Burst Time***
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4



Shortest Job First (SJF)



Wait times: $P_1 = 0$, $P_2 = 6$, $P_3 = 3$, $P_4=7$

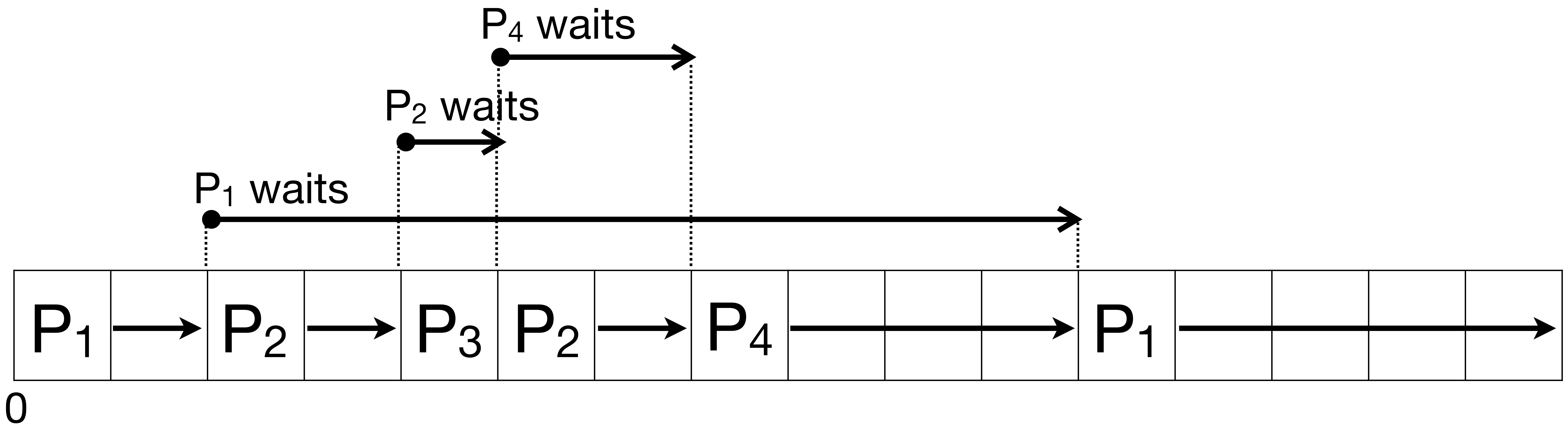
Average: $(0 + 6 + 3 + 7) / 4 = 4$

(can we do better?)

Preemptive SJF (PSJF)

- aka “Shortest Time-to-Completion First” (STCF)
- aka “Shortest Remaining-Time First” (SRTF)
- May preempt running job to schedule a different (ready) job

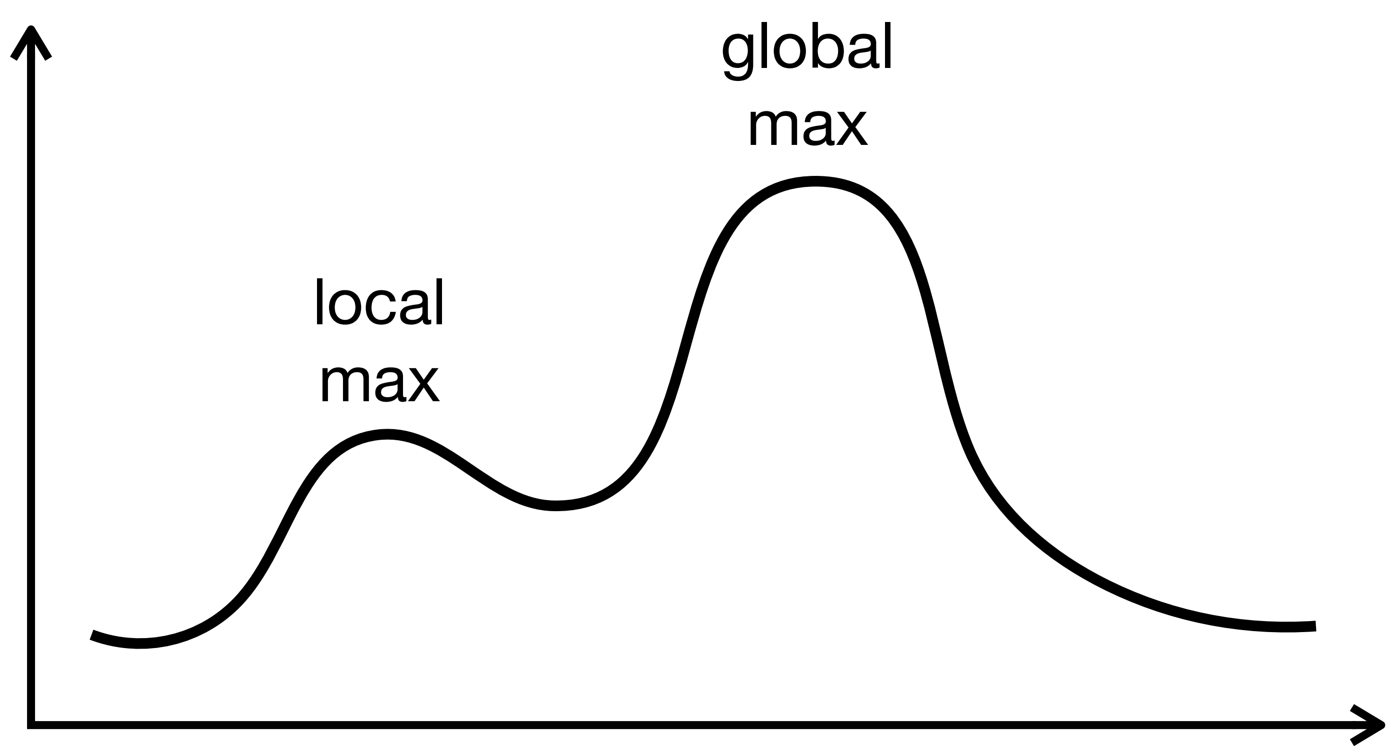
Process	***Arrival Time***	***Burst Time***
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4



Wait times: $P_1 = 9$, $P_2 = 1$, $P_3 = 0$, $P_4 = 2$

Average: $(9 + 1 + 0 + 2) / 4 = 3$ (vs SJF @ 4)

Greedy algorithms



- SJF/PSJF are **greedy** algorithms
  - i.e., they select the best choice *at the moment* ("local maximum")
- Greedy algorithms don't always produce globally maximal results
  - e.g., naive hill-climbing algorithm (only take a step if it brings me to higher ground) doesn't always find the tallest peak!
- Are SJF/PSJF optimal?

Optimal?

- Consider 4 jobs with burst lengths t_0, t_1, t_2, t_3 that just became ready
  - What is the average wait time if scheduled in the order given?

    $= (3 \cdot t_0 + 2 \cdot t_1 + t_2) / 4$

    - Weighted average — clearly minimized by running shortest jobs first!
- SJF/PSJF are **provably optimal** with respect to average wait time
  - But at what cost?
    - **Potential CPU starvation!** (e.g., longer jobs keep getting put off)

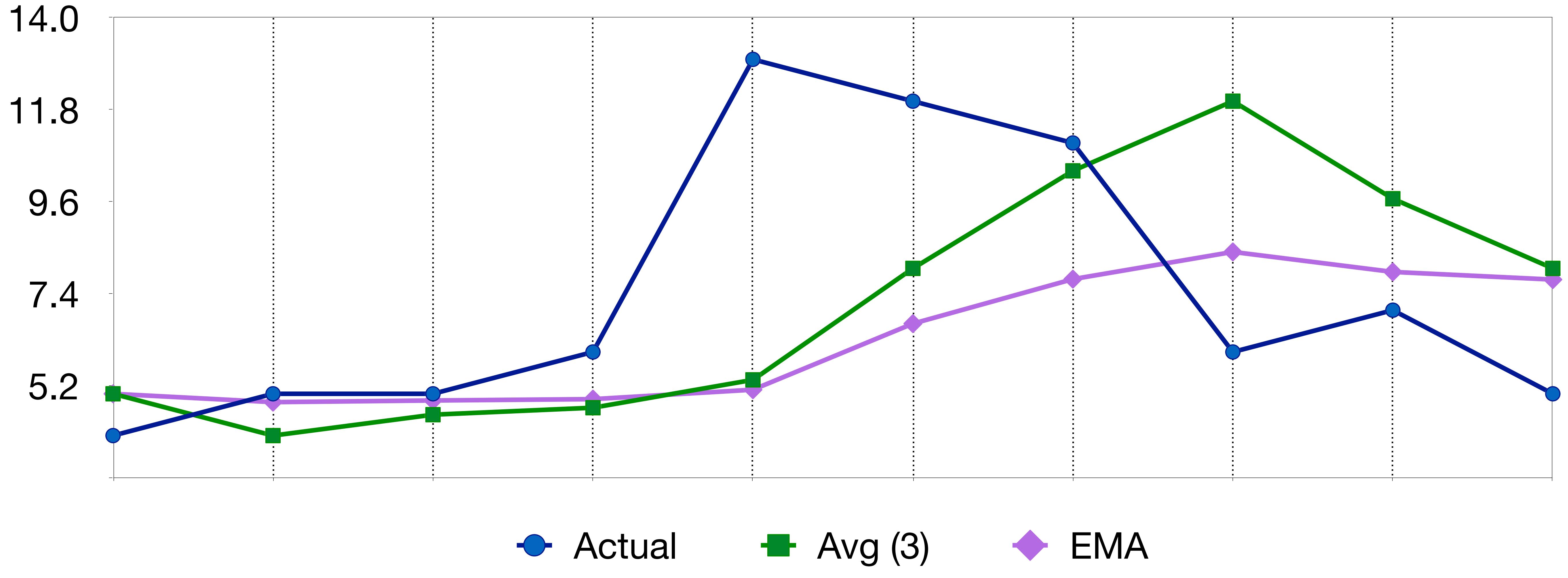
A snag: no Oracle

- We’ve been assuming that job/burst lengths are known in advance
- May be possible in rare circumstances (e.g., repeated jobs, job profiling), but unlikely in practice
- Common approach: predict future burst lengths based on past behavior
  - Simple moving average (sliding window of past values)
  - Exponentially weighted moving average (EMA)

Exponential Moving Average (EMA)

- Observed: $\rho_{n\text{-}1}$
- Estimated: $\sigma_{n\text{-}1}$
- Weight (a): $0 \leq a \leq 1$
- Predicted: $\sigma_n = a \cdot \rho_{n\text{-}1} + (1 - a) \cdot \sigma_{n\text{-}1}$
- i.e., bigger a = more weight given to observed data

Actual	Avg (3)	Error	EMA (α=0.2)	Error
4	5.00	1.00	5.00	1.00
5	4.00	1.00	4.80	0.20
5	4.50	0.50	4.84	0.16
6	4.67	1.33	4.87	1.13
13	5.33	7.67	5.10	7.90
12	8.00	4.00	6.68	5.32
11	10.33	0.67	7.74	3.26
6	12.00	6.00	8.39	2.39
7	9.67	2.67	7.92	0.92
5	8.00	3.00	7.73	2.73
	Avg err:	2.78	Avg err:	2.50

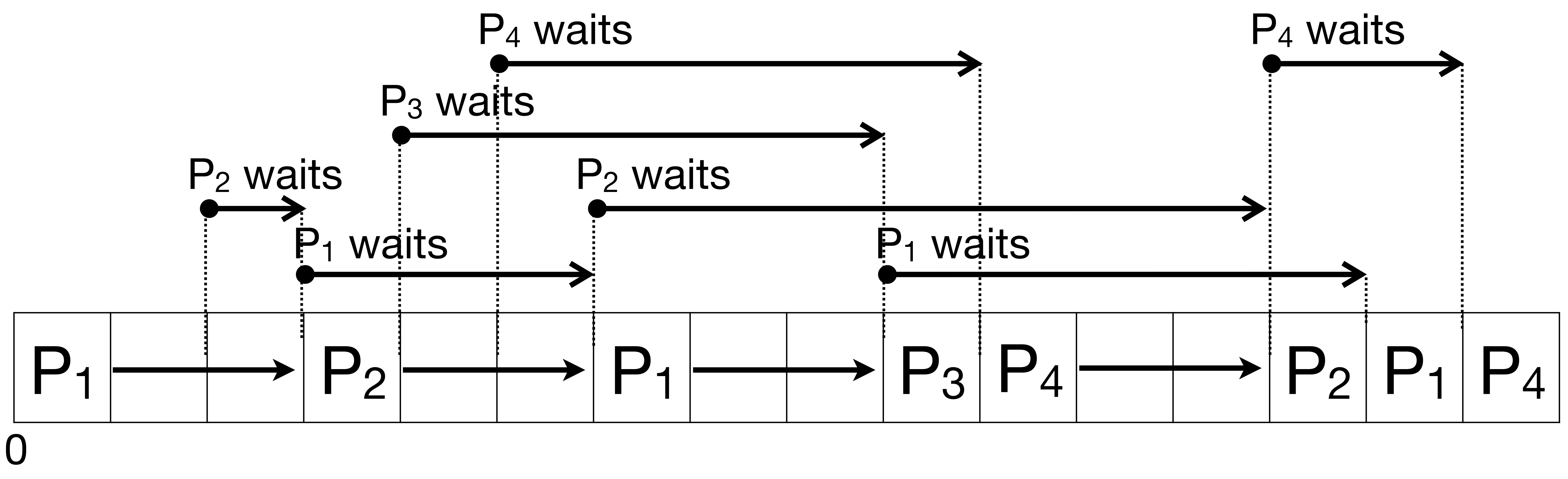


Round Robin (RR)

- The “fairest” of them all
- Uses a FIFO queue:
  - Each job runs for a maximum fixed time quantum q
  - If unfinished, re-enter queue at the tail end
- Given time quantum q and n jobs:
  - max wait time (per cycle) = $q \cdot (n - 1)$
  - each job receives $1/n$ timeshare

Process	*Arrival Time*	*Burst Time*
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

RR *q=3*



Wait times: $P_1 = 8$, $P_2 = 8$, $P_3 = 5$, $P_4 = 7$

Average: $(8 + 8 + 5 + 7) / 4 = 7$

Process	*Arrival Time*	*Burst Time*
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

	Avg. Turnaround	**Avg. Wait Time**	
RR *q*=7	8.75	4.75	(FCFS)
RR *q*=4	9	5	
RR *q*=3	11	7	
RR *q*=1	9.75	5.75	

Process	*Arrival Time*	*Burst Time*
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

	Throughput	**Utilization**
RR *q*=7	0.25	1.0
RR *q*=4	0.25	1.0
RR *q*=3	0.25	1.0
RR *q*=1	0.25	1.0

Another snag: context switch time

- CST = interrupt + context switch + scheduler
  - ~1 μs in Linux on recent hardware
- Each time we preempt a job we introduce systemic overhead (i.e., costs not incurred by the job itself) and reduce utilization
- Longer quantum times help **amortize** the cost of CSTs
- Just measuring CST oversimplifies the cost of context switches
  - E.g., *cache perturbation* significantly affects execution efficiency

Process	***Arrival Time***	***Burst Time***
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

(CST=1)	**Avg. Turnaround**	**Avg. Wait Time**
RR *q*=7	10.25	6.25
RR *q*=4	11.5	7.25
RR *q*=3	16.25	11.25
RR *q*=1	20.25	13.25

Process	*Arrival Time*	*Burst Time*
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

(CST=1)	**Throughput**	**Utilization**
RR *q*=7	0.2	0.8
RR *q*=4	0.19	0.762
RR *q*=3	0.167	0.667
RR *q*=1	0.125	0.5

Tuning *q*

- Generally, try to choose q to help tune system responsiveness
- May use different predictors:
  - Predetermined burst-length threshold (for interactive jobs)
  - Median of EMAs
  - Process profiling
- RR prevents starvation and allows both CPU-hungry and interactive jobs to share resources fairly
  - But potentially at the cost of poor turnaround/wait times!

Which is which?

Simulation: SJF / PSJF / RR q=10 / RR q=20

processes: uniform bursts ≤ 20, CST = 1.0

Table Data

									Entries		Average Time	
Name	Key	Time	Processes	Finished	CPU Utilization	Throughput	CST	LA	CPU	I/O	CPU	I/O
secret_1	ALG 1	10550.00	100	100	.947867	.009479	550.00	91.36	1375	770	7.27	50.20
secret_2	ALG 2	10511.66	100	100	.951324	.009513	348.00	59.74	870	770	11.49	50.20
secret_3	ALG 3	10376.90	100	100	.963679	.009637	348.00	88.01	870	770	11.49	50.20
secret_4	ALG 4	10588.08	100	100	.944459	.009445	440.80	59.72	1102	770	9.07	50.20

		Turnaround Time				Waiting Time			
Name	Key	Average	Minimum	Maximum	SD	Average	Minimum	Maximum	SD
secret_1	ALG 1	10124.63	8887.82	10549.80	405.48	9637.08	8435.62	10046.80	3.72
secret_2	ALG 2	6765.84	1956.80	10511.46	2342.38	6279.30	1455.20	10045.31	23.57
secret_3	ALG 3	9619.54	7277.89	10376.70	712.98	9133.00	6926.89	9774.70	6.65
secret_4	ALG 4	6809.22	1967.20	10587.88	2370.05	6322.22	1465.60	10121.12	23.85

Done

Priority Schedulers

- Can implement more fine-grained scheduling policies by introducing a system of arbitrary priorities, gathered/computed by the scheduler
  - Process with maximum priority is scheduled
- SJF/PSFJ are priority schedulers! (priority = 1 / predicted burst length)
- Starvation due to priority scheduling may be combatted by **aging**
  - But there may be other insidious issues!

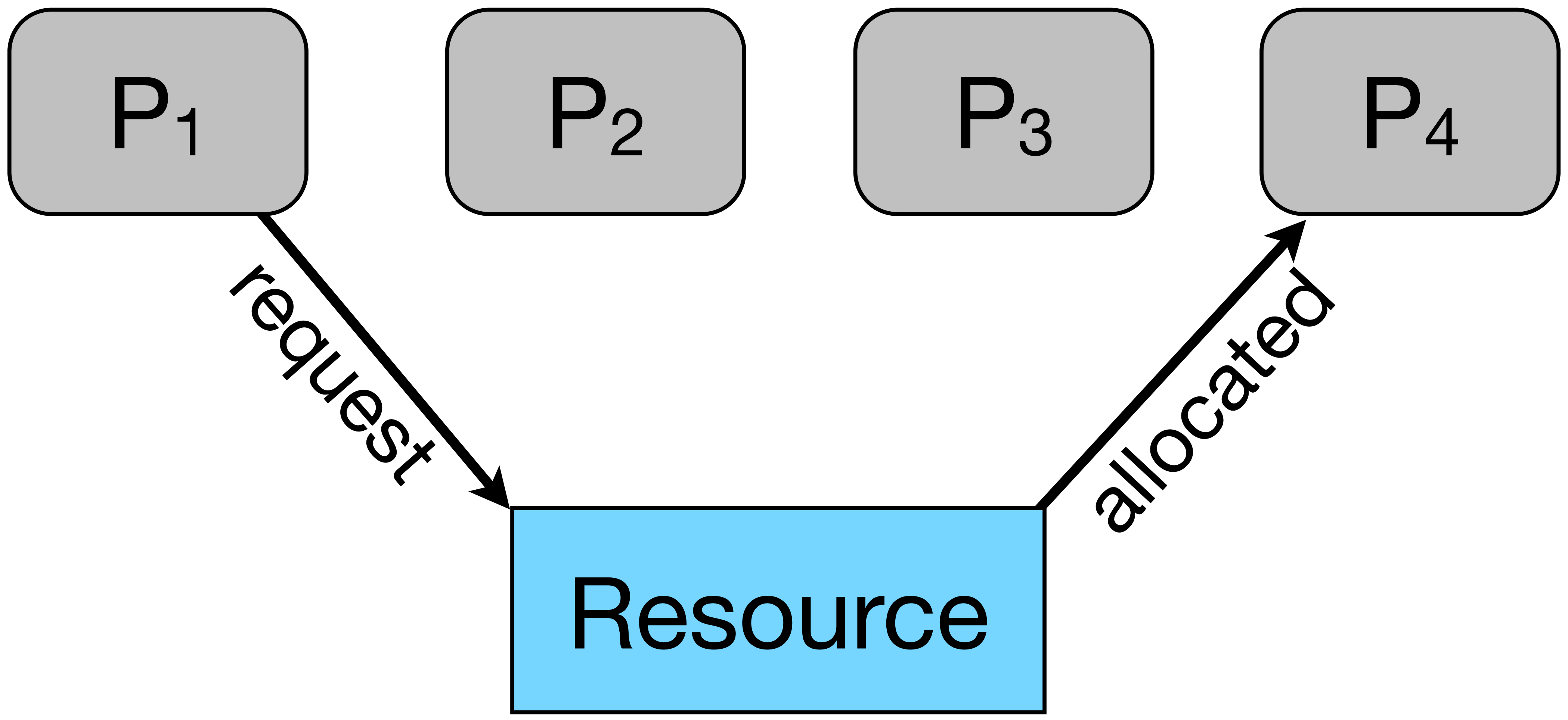
Highest Penalty Ratio Next (HPRN)

- Example of a priority scheduler that implements aging
- Two statistics maintained by scheduler for each job:
  1. "wall clock" age, **t**
  2. total CPU execution time, **e**
- Priority is the "penalty ratio" = **t / e**
  - ∞ when job is first ready, decreases as job receives CPU time
- In practice would incur too many context switches!
  - Can institute a minimum execution quantum (is this RR?)

E.g., another possible problem due to priority scheduling: **priority inversion**

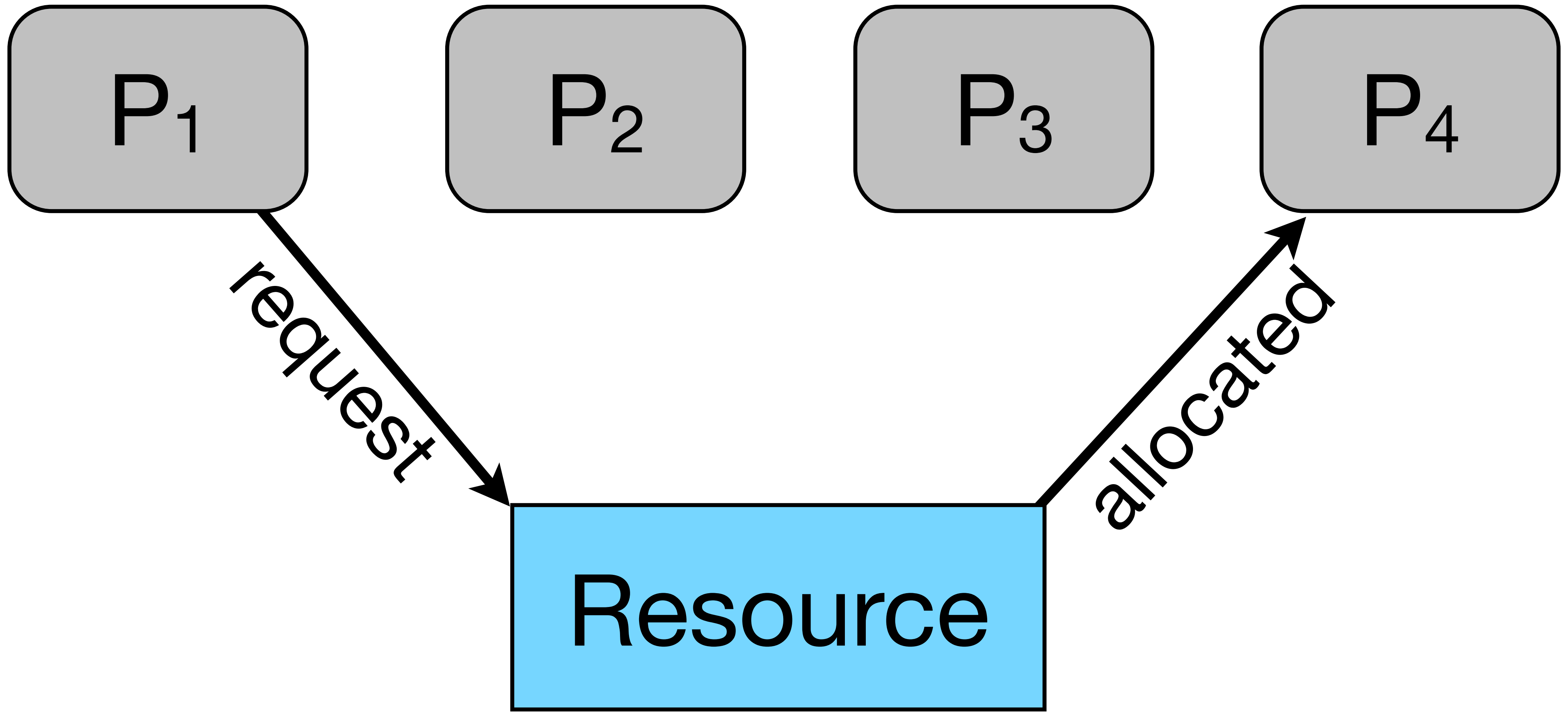
Process	Priority	State
P_1	High	Ready
P_2	Mid	Ready
P_3	Mid	Ready
P_4	Low	Ready

Process	Priority	State
P_1	High	Running
P_2	Mid	Ready
P_3	Mid	Ready
P_4	Low	Ready



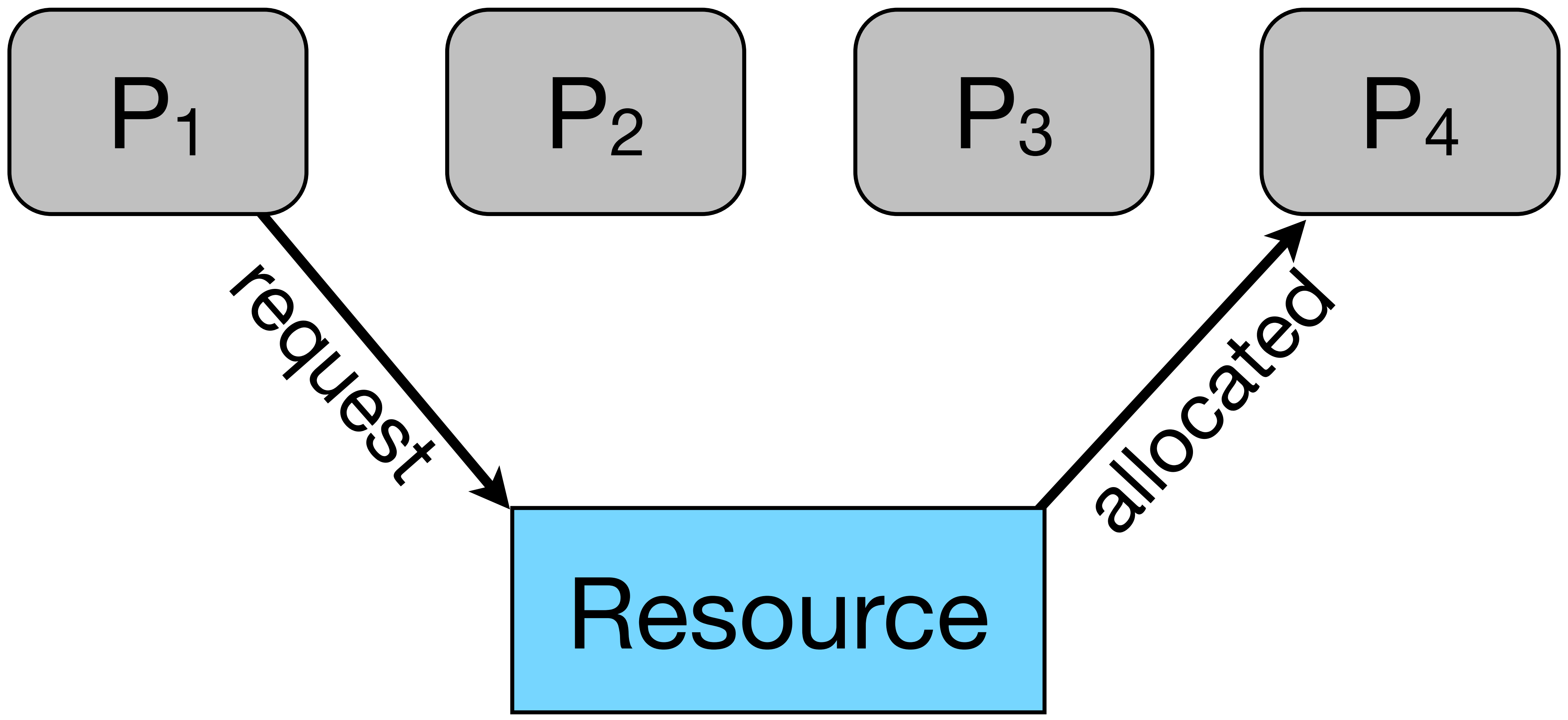
(*mutually exclusive* allocation)

Process	Priority	State
P_1	High	**Blocked**
P_2	Mid	Ready
P_3	Mid	Ready
P_4	Low	Ready



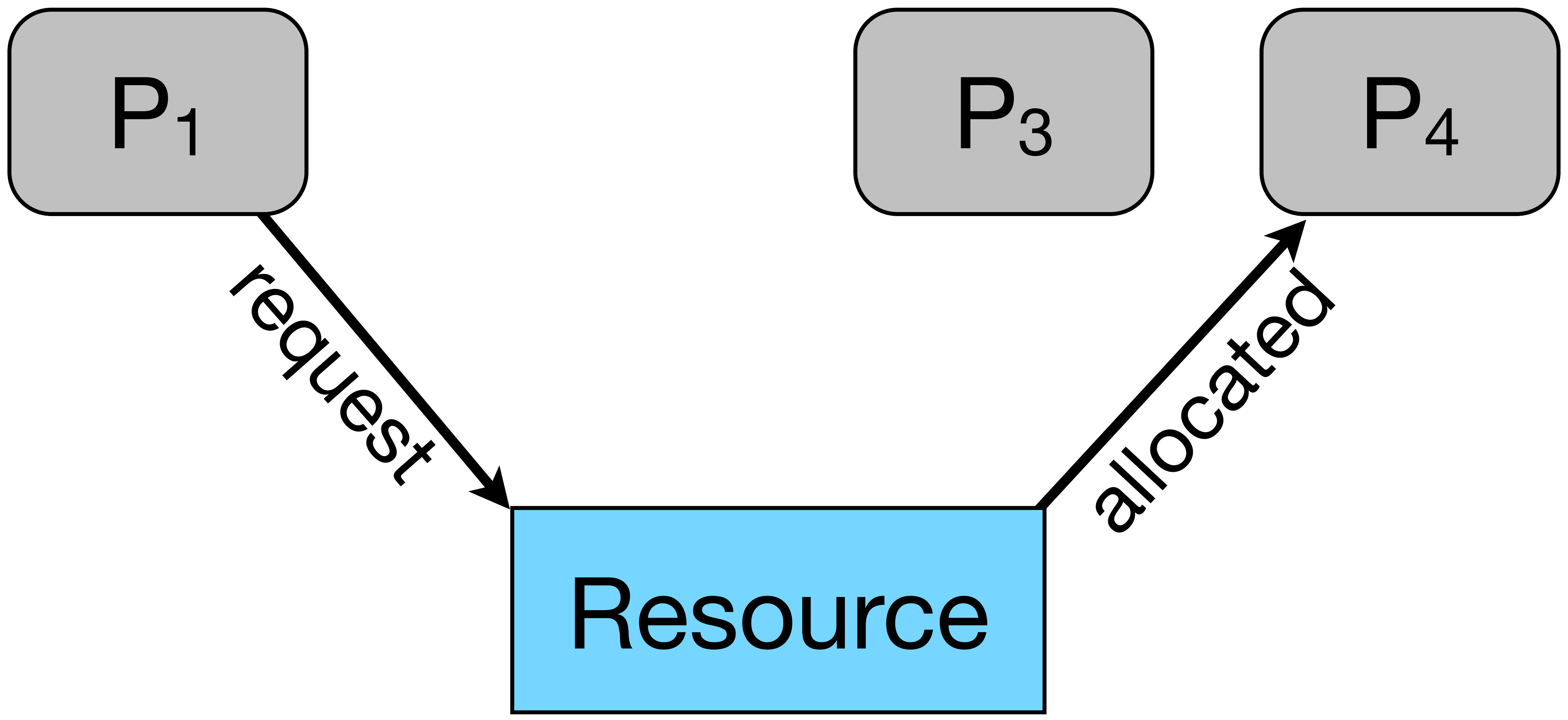
(*mutually exclusive* allocation)

Process	Priority	State
P_1	High	Blocked
P_2	Mid	Running
P_3	Mid	Ready
P_4	Low	Ready



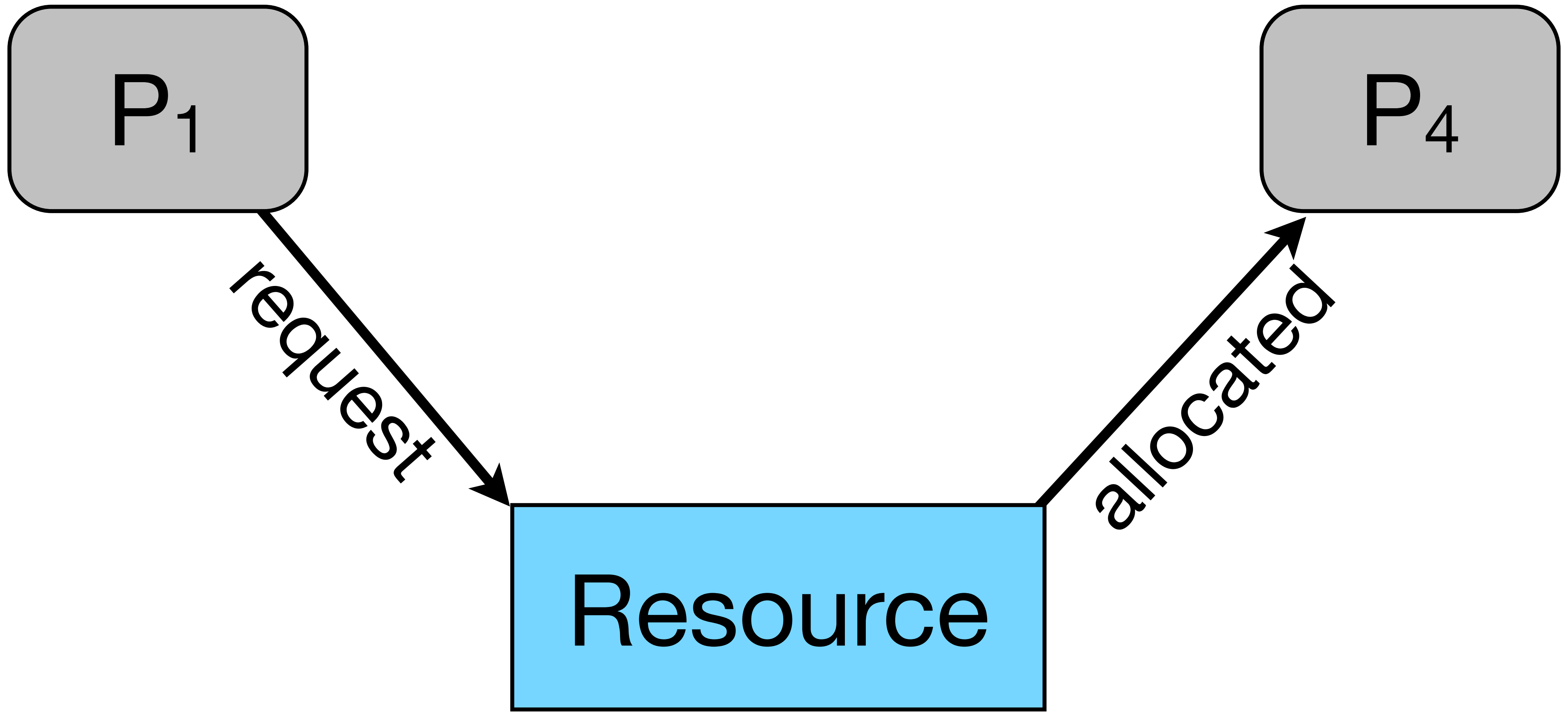
(*mutually exclusive* allocation)

Process	Priority	State
P_1	High	Blocked
~~P_2~~	~~Mid~~	~~Done~~
P_3	Mid	Running
P_4	Low	Ready



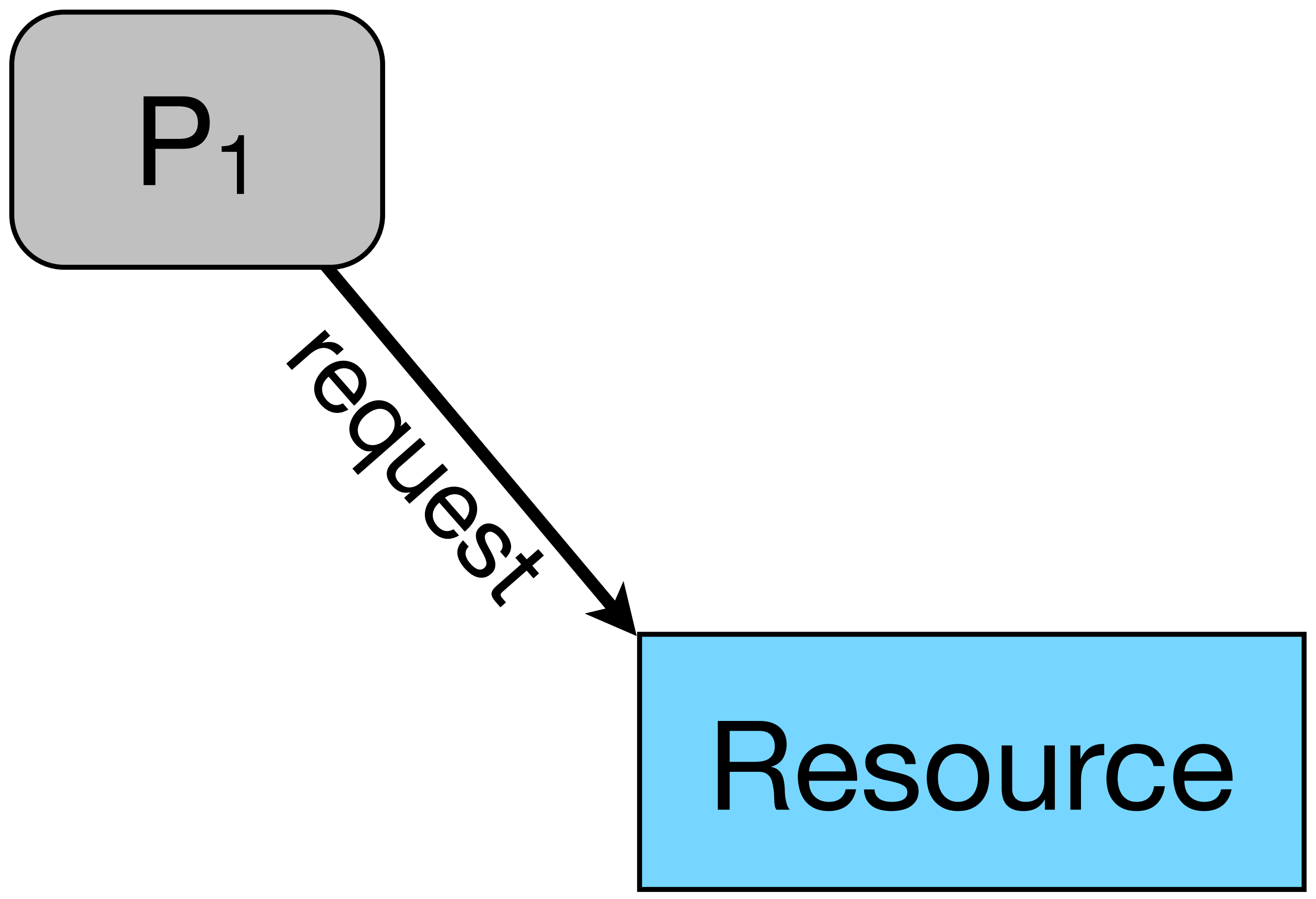
(*mutually exclusive* allocation)

Process	Priority	State
P_1	High	Blocked
~~P_2~~	~~Mid~~	~~Done~~
~~P_3~~	~~Mid~~	~~Done~~
P_4	Low	Running



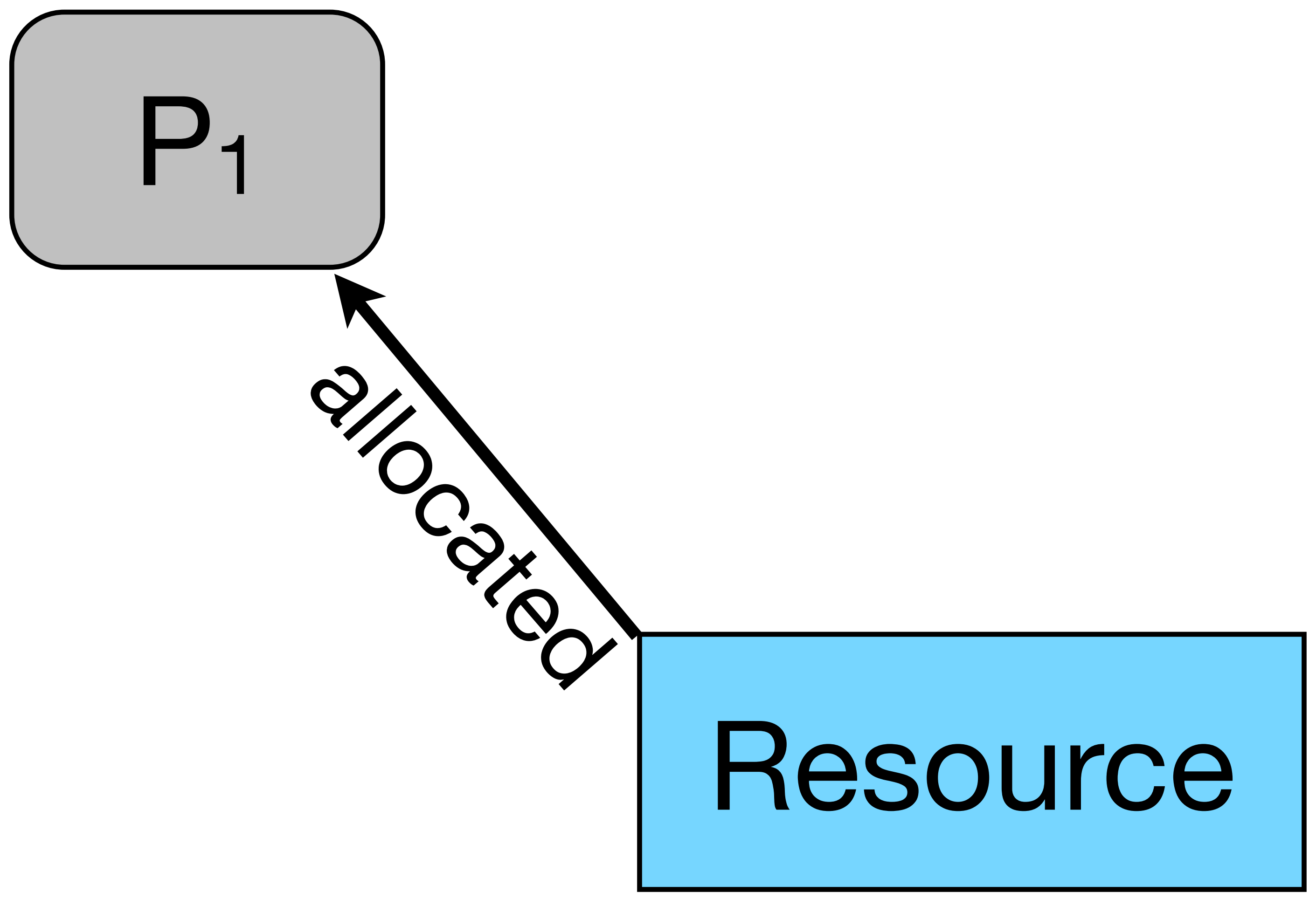
(*mutually exclusive* allocation)

Process	Priority	State
P_1	High	Blocked
~~P_2~~	~~Mid~~	~~Done~~
~~P_3~~	~~Mid~~	~~Done~~
~~P_4~~	~~Low~~	~~Done~~



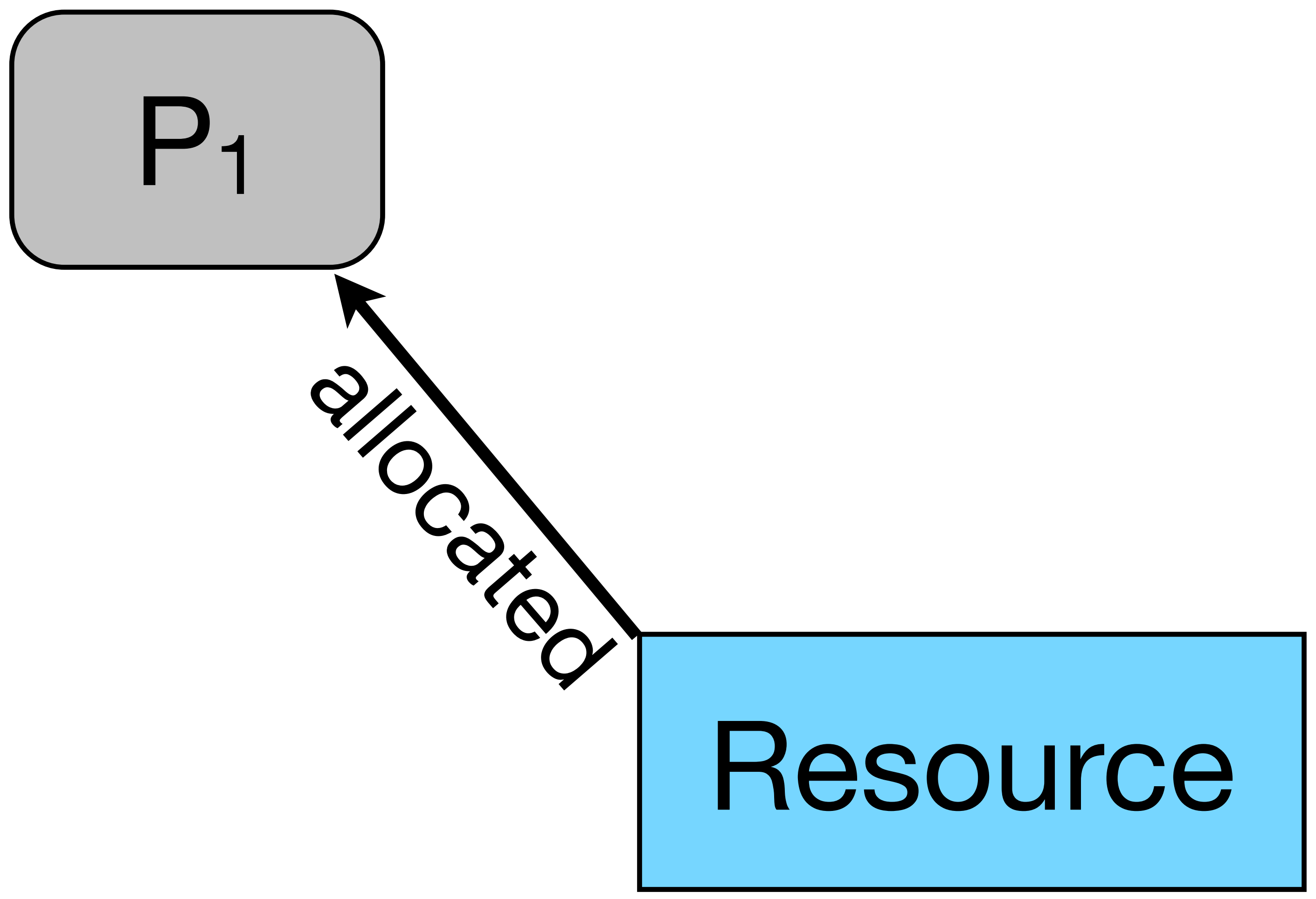
(*mutually exclusive* allocation)

Process	Priority	State
P_1	High	Ready
~~P_2~~	~~Mid~~	~~Done~~
~~P_3~~	~~Mid~~	~~Done~~
~~P_4~~	~~Low~~	~~Done~~



(*mutually exclusive* allocation)

Process	Priority	State	
P_1	High	Running	(Finally!)
~~P_2~~	~~Mid~~	~~Done~~	
~~P_3~~	~~Mid~~	~~Done~~	
~~P_4~~	~~Low~~	~~Done~~	



(*mutually exclusive* allocation)

e.g., NASA Pathfinder (1996-1997)

- Real-time OS (vxWorks) developed a recurring system failure/reset after robot was deployed to surface of Mars
- Culprit: unanticipated flood of meteorological data coupled with low priority of the data gathering job (**ASI/MET**)
  - **ASI/MET** held onto a resource needed by the high priority data distribution job (**bc_dist**), but **ASI/MET** was superseded by medium-priority jobs

e.g., NASA Pathfinder (1996-1997)

- Scheduler determined that **bc_dist** couldn’t complete by the hard deadline set by the RTOS
  - Declared error and performed system reset to “fix” scheduling!
- Reproduced in lab on Earth after 18 hours of simulation

e.g., NASA Pathfinder (1996-1997)

- Fix: *priority inheritance*
  - Lower priority job inherits the priority of the job waiting for its resource
  - i.e., run **ASI/MET** at the priority of **bc_dist** until resource released
- Engineers remote-patched robot from Earth, enabling priority inheritance for the in-demand resource in vxWorks OS (why wasn't it enabled before?)
  - Hailed as an operational success!

“Our before launch testing was limited to the “best case” high data rates and science activities… We did not expect nor test the “better than we could have ever imagined” case.”

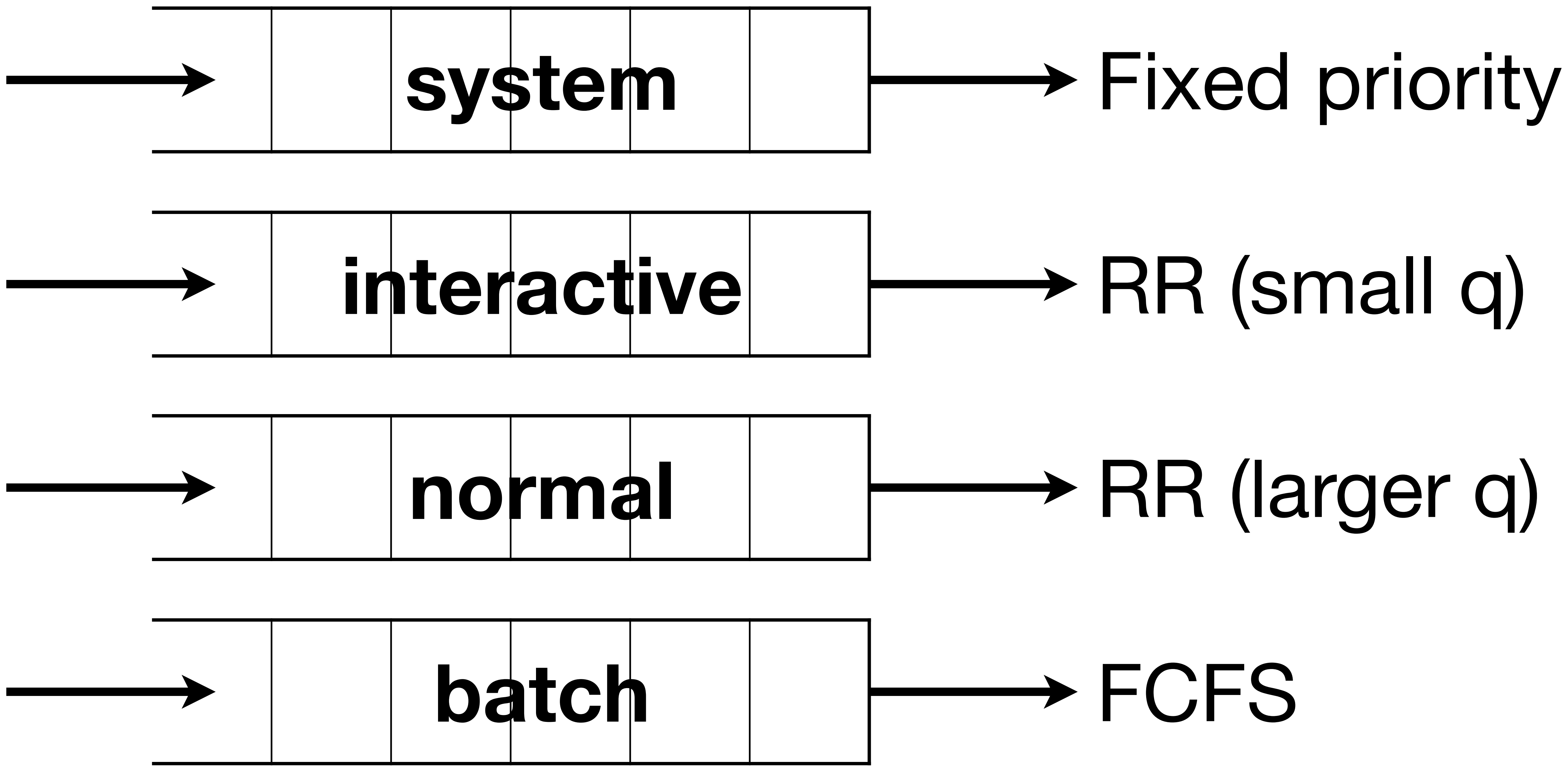
- Glenn Reeves
  Software team lead

Scheduling is rocket science!

- Jobs are unpredictable, and interactions between jobs even more so
- Priority-based scheduling is useful, as it may help us optimize different scheduling metrics. But there are potential downsides:
  - Starvation and Priority inversion
  - Not all jobs require the same sort of optimization!
    - E.g., CPU-bound vs. interactive jobs
- Would like a mechanism that allows us to optimize for different metrics across separate groups of processes

Multi-Level Queue

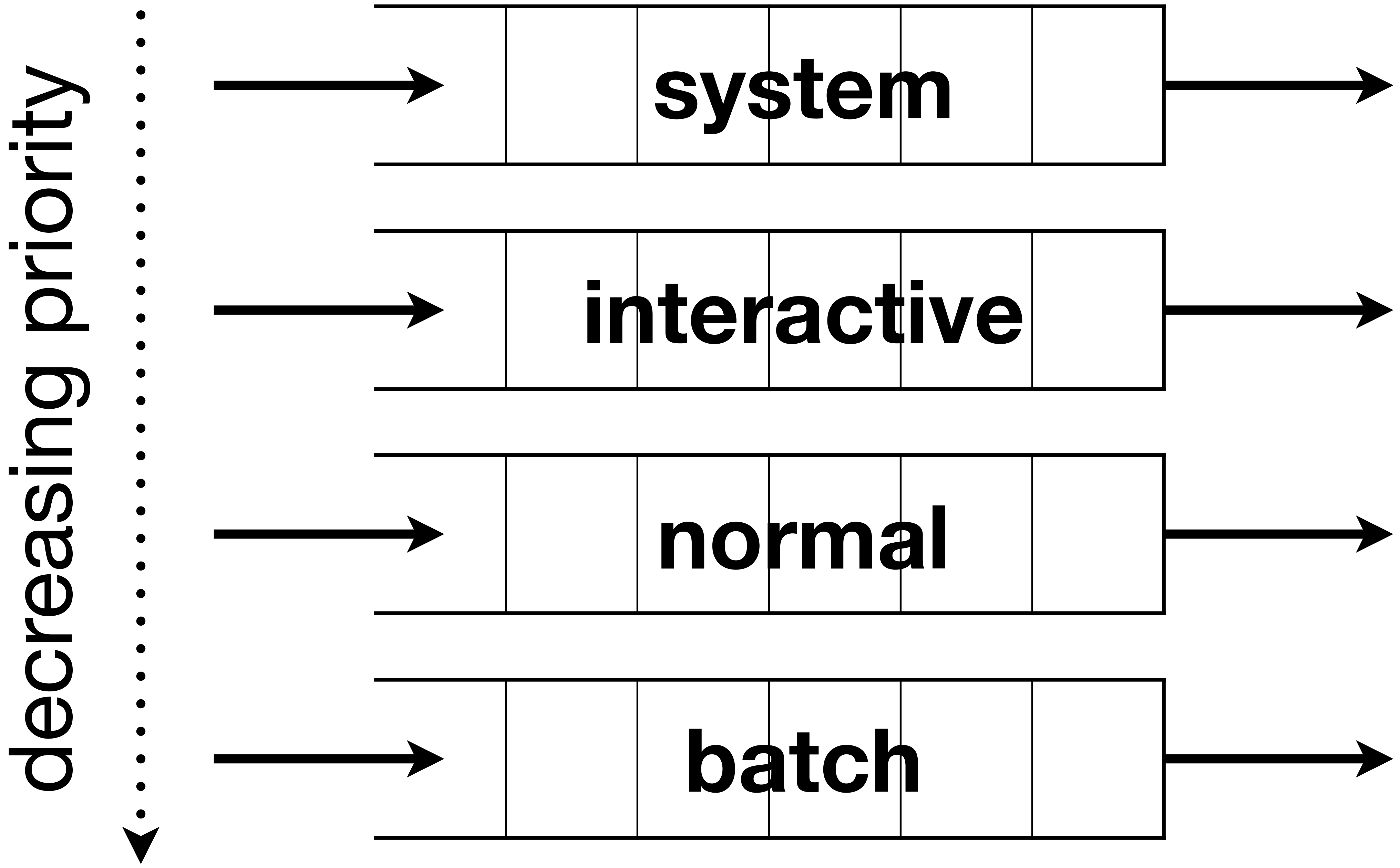
- Idea: disjoint ready queues, with separate scheduling policies
- E.g.,



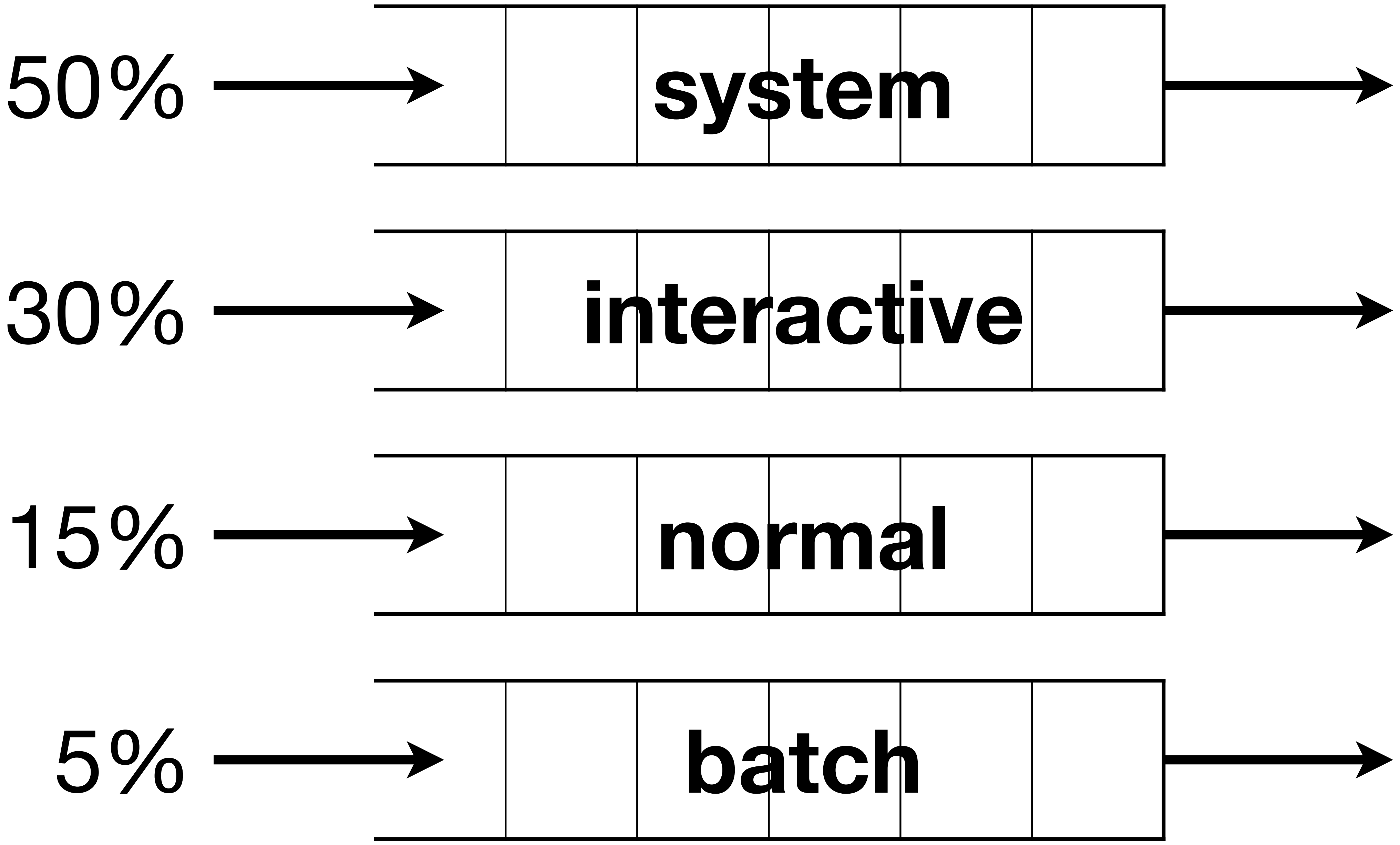
Multi-Level Queue

- Requires a *queue arbitration* policy, i.e., which queue to select jobs from?

- Approach 1: select jobs from top, non-empty queue



- Approach 2: allocate macro time slices to each queue



Multi-Level Queue

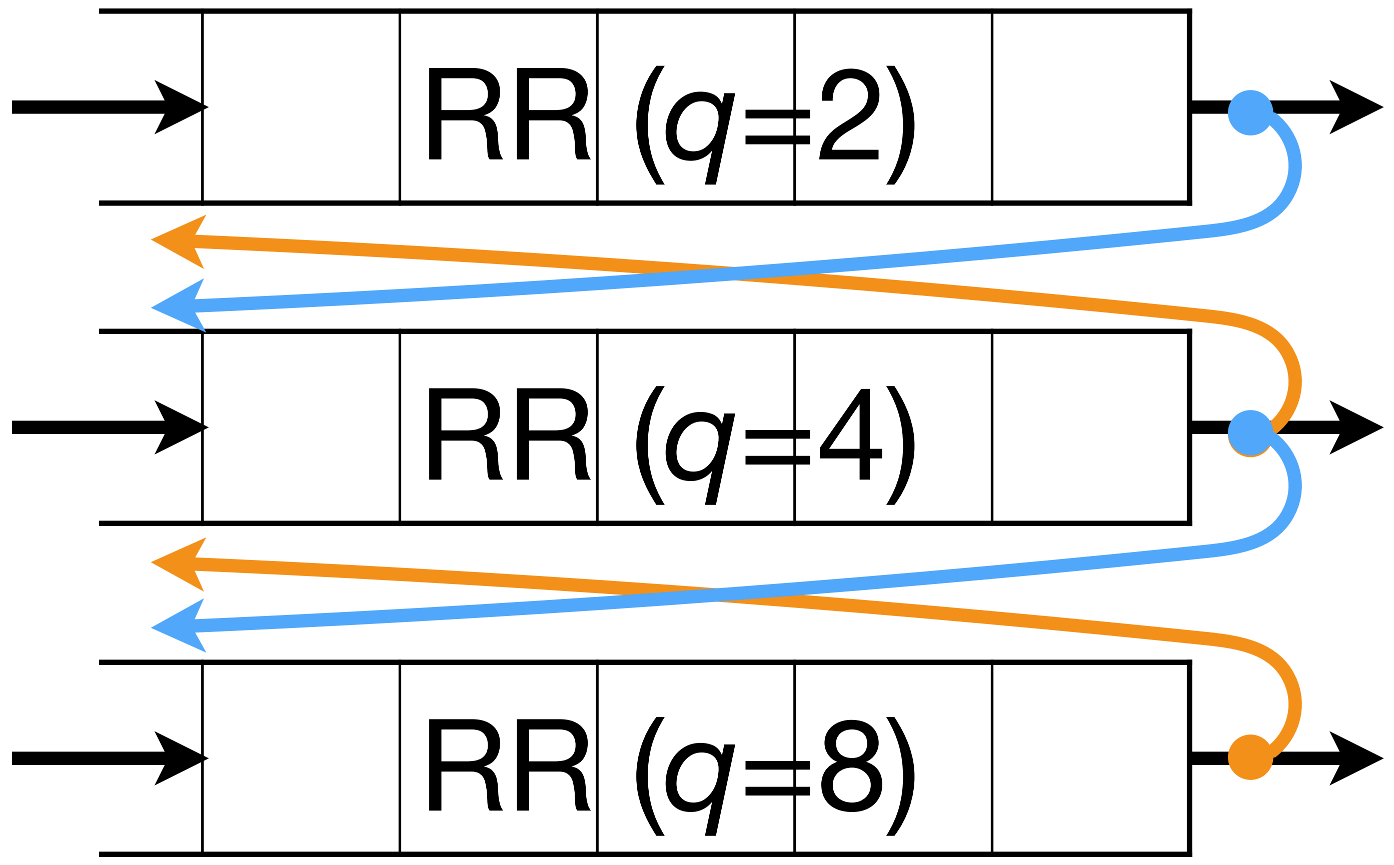
- Which jobs go in which queues?
- Can be self-declared/assigned
  - e.g., UNIX “nice” value
  - Can jobs be trusted?
- Jobs can be *profiled* based on initial burst(s)
  - e.g., short, periodic CPU bursts → classify as interactive job
  - May be gamed by programmers looking for better treatment

Shifting requirements?

- More important issue: what if job requirements change *dynamically*?
- E.g., photo editor: tool selection (interactive) → apply filter (CPU-bound) → simple edits (interactive) → apply compression (CPU-bound) …
- Scheduler should respond to changes in job requirements by applying appropriate policies
  - While maximizing responsiveness and efficiency where possible!

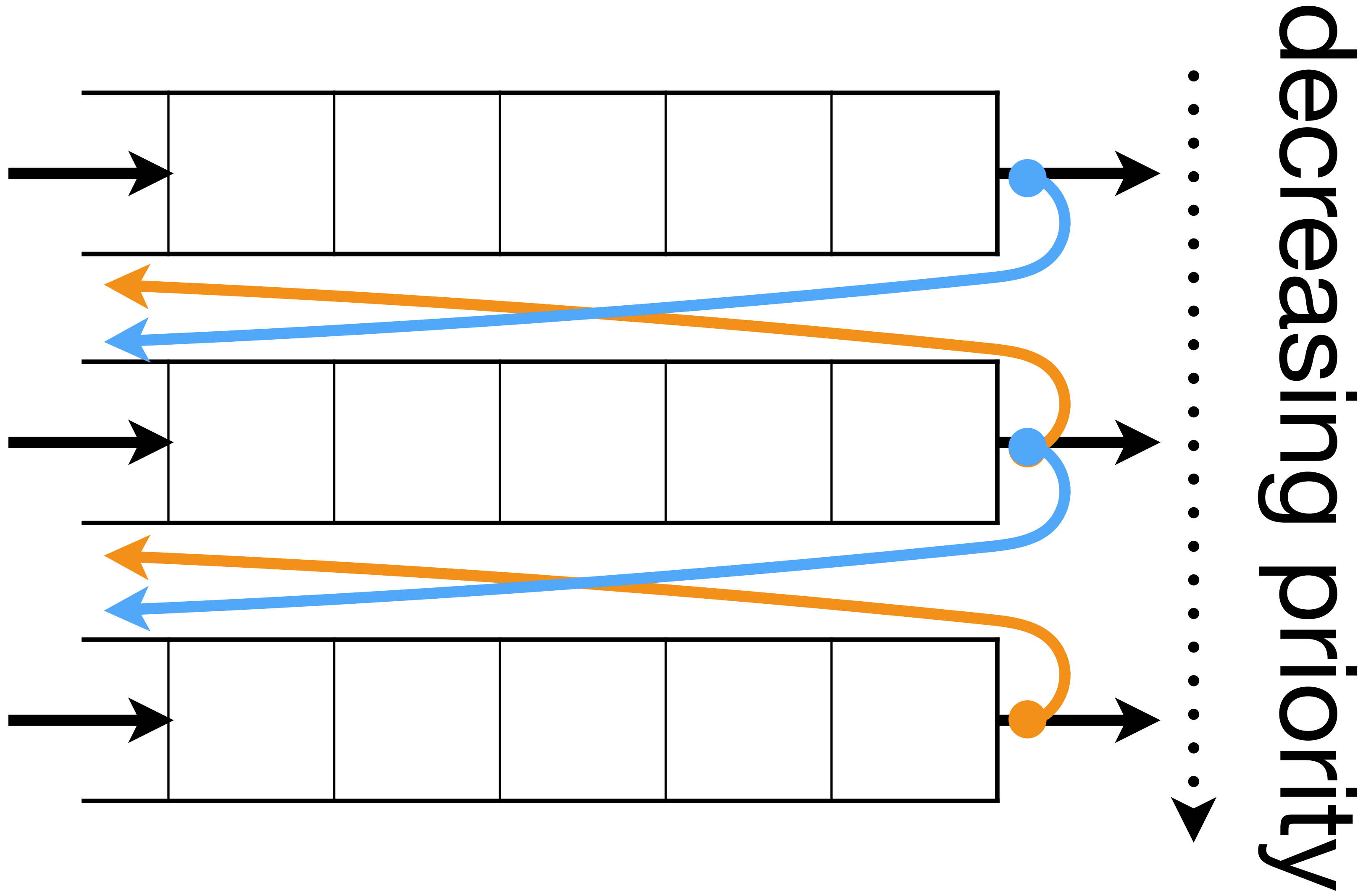
Multi-Level Feedback Queue (MLFQ)

- Supports movement between queues after initial assignment
  - Based on dynamic job characteristics (mostly discerned from burst lengths relative to allocated quanta)
  - e.g., 3 RR queues with different *q*

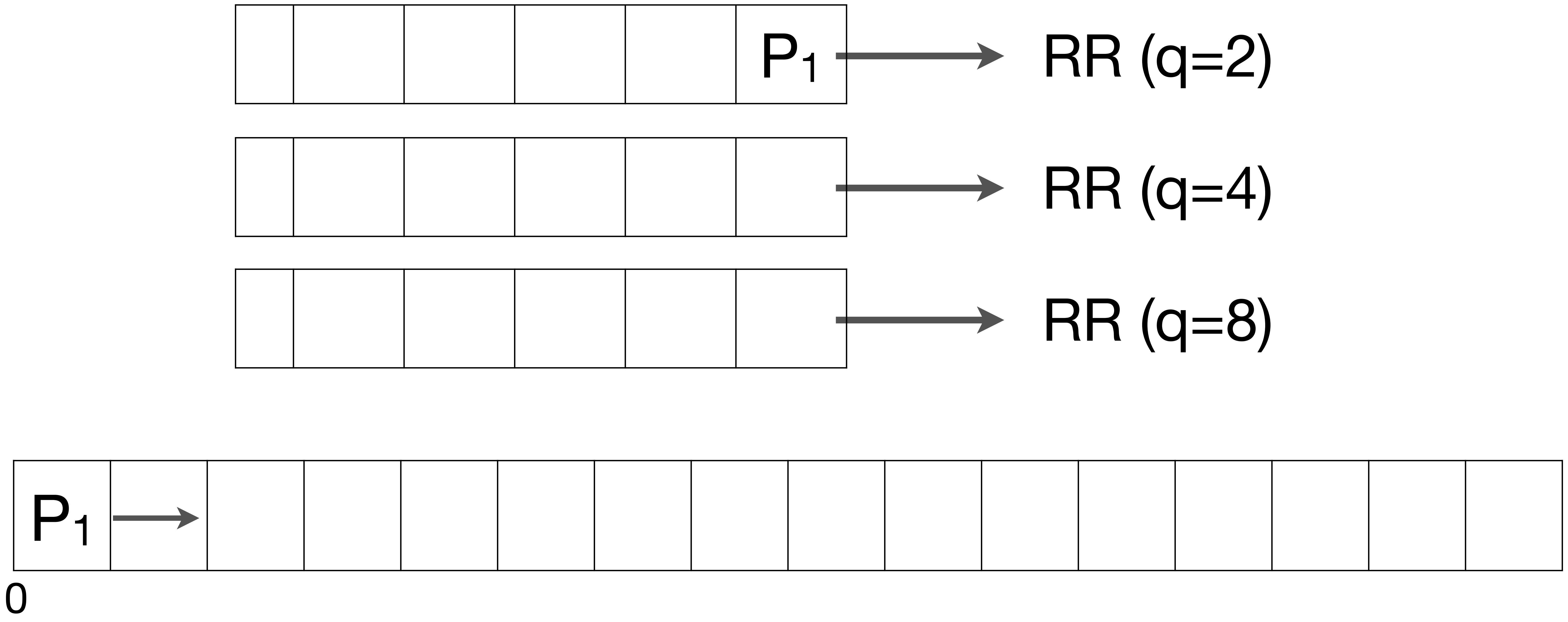


Multi-Level Feedback Queue (MLFQ)

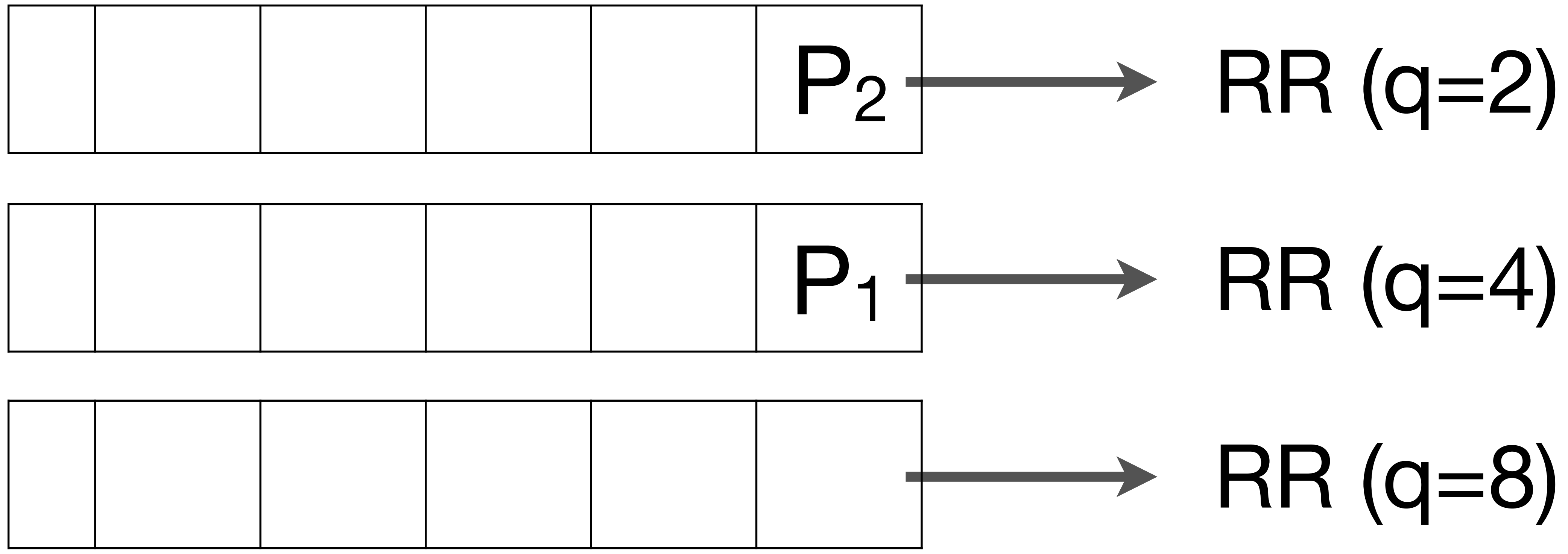
- Rules:
  - Only select from highest non-empty queue
  - Within a queue, schedule using RR
  - New jobs enter into highest priority queue
  - If job uses entire quantum, move down (deprioritize)

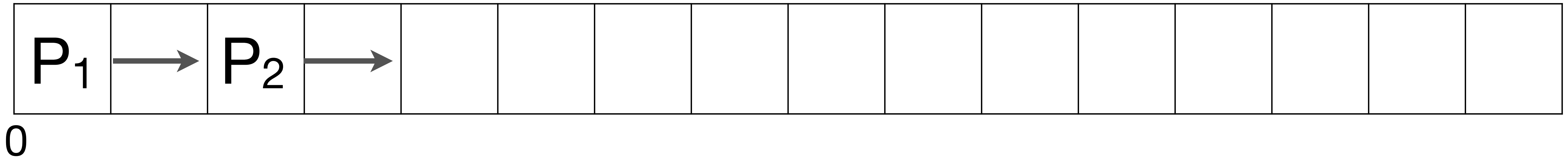


Process	***Arrival Time***	***Burst Time***
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

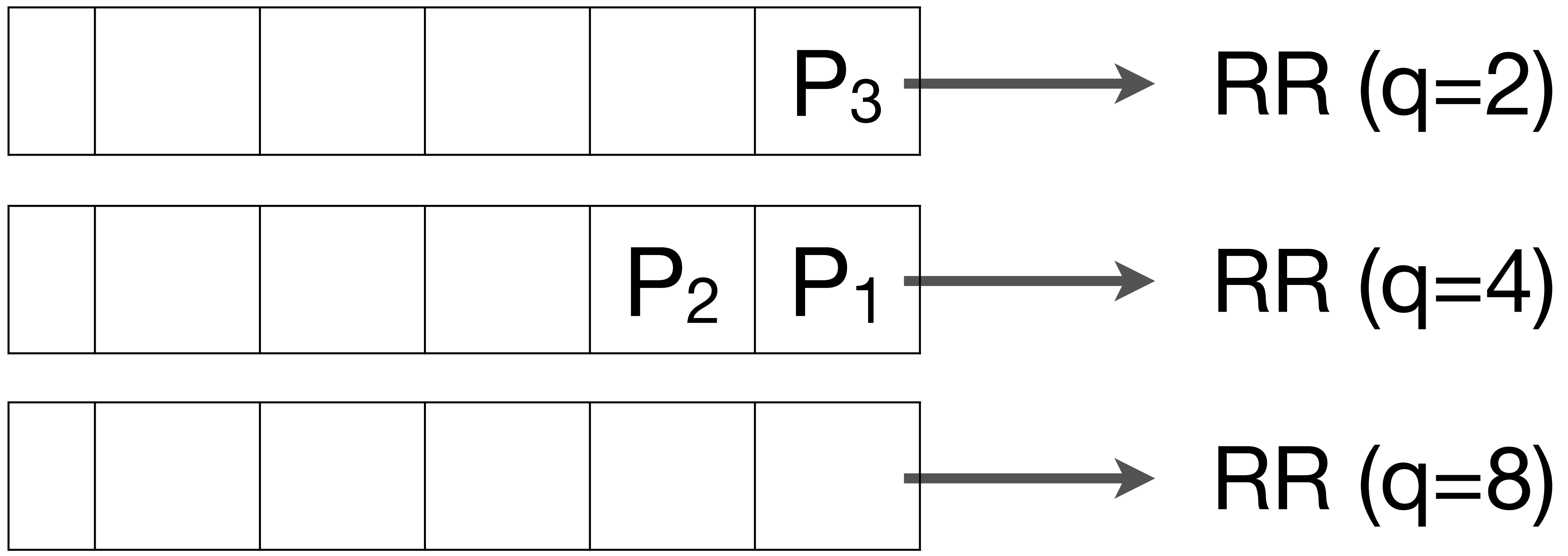


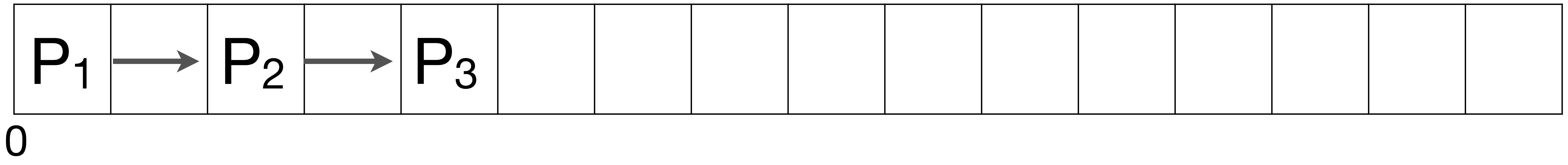
Process	*Arrival Time*	*Burst Time*
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4



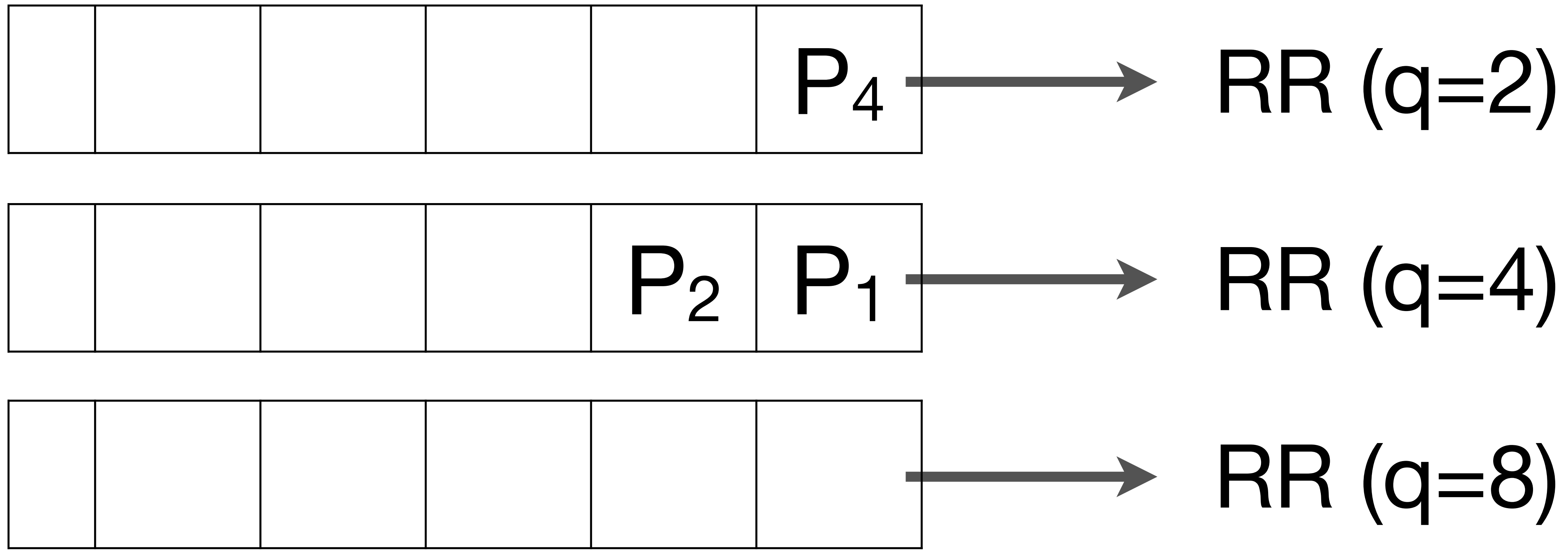


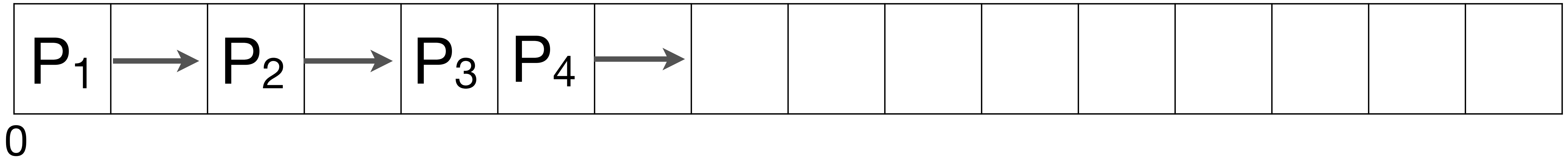
Process	***Arrival Time***	***Burst Time***
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4



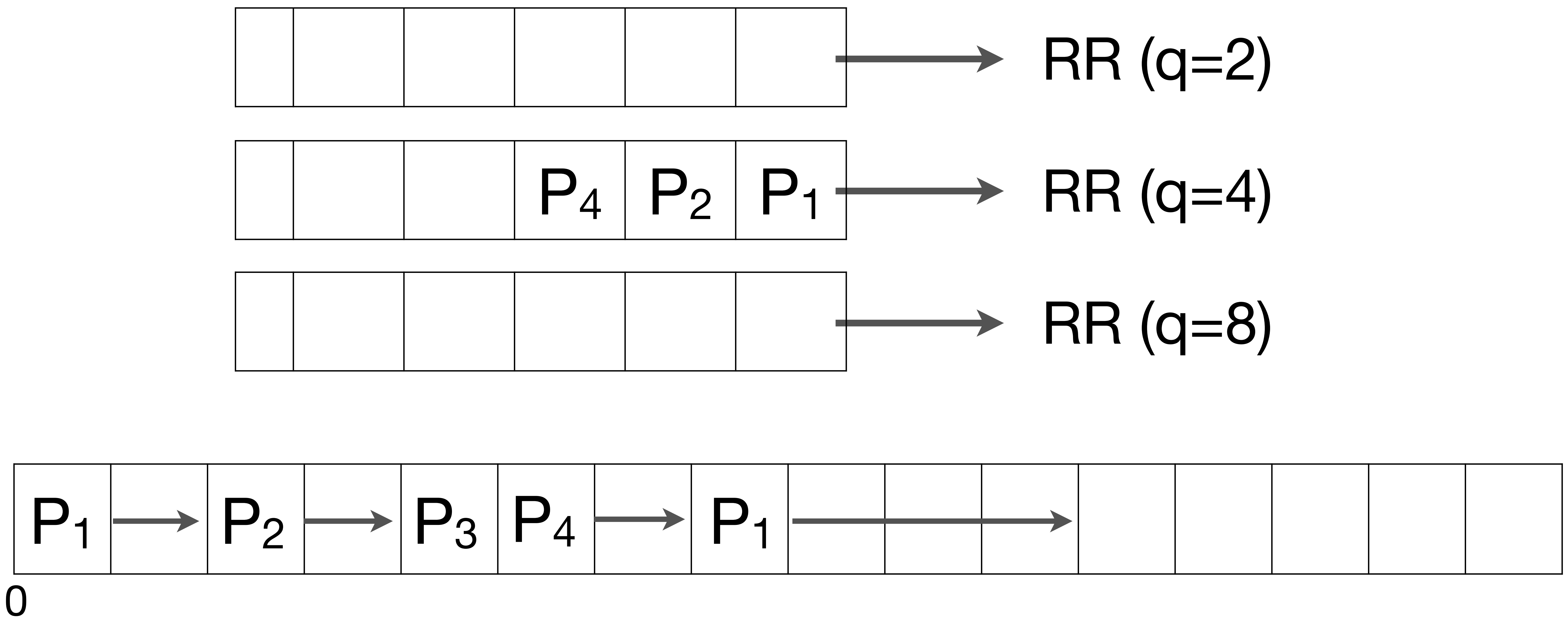


Process	***Arrival Time***	***Burst Time***
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

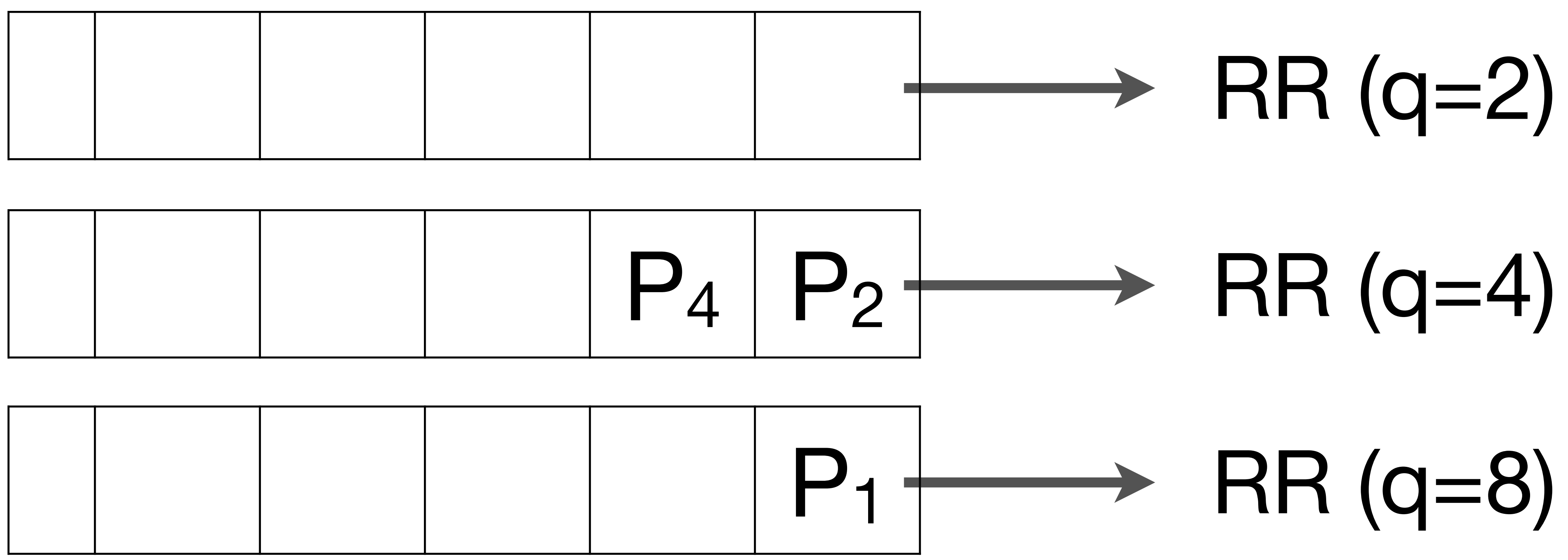


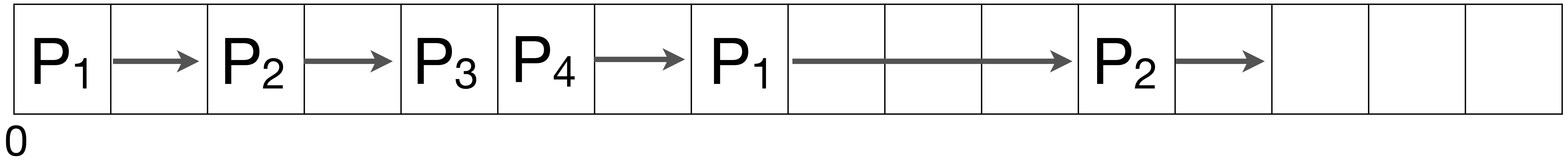


Process	*Arrival Time*	*Burst Time*
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

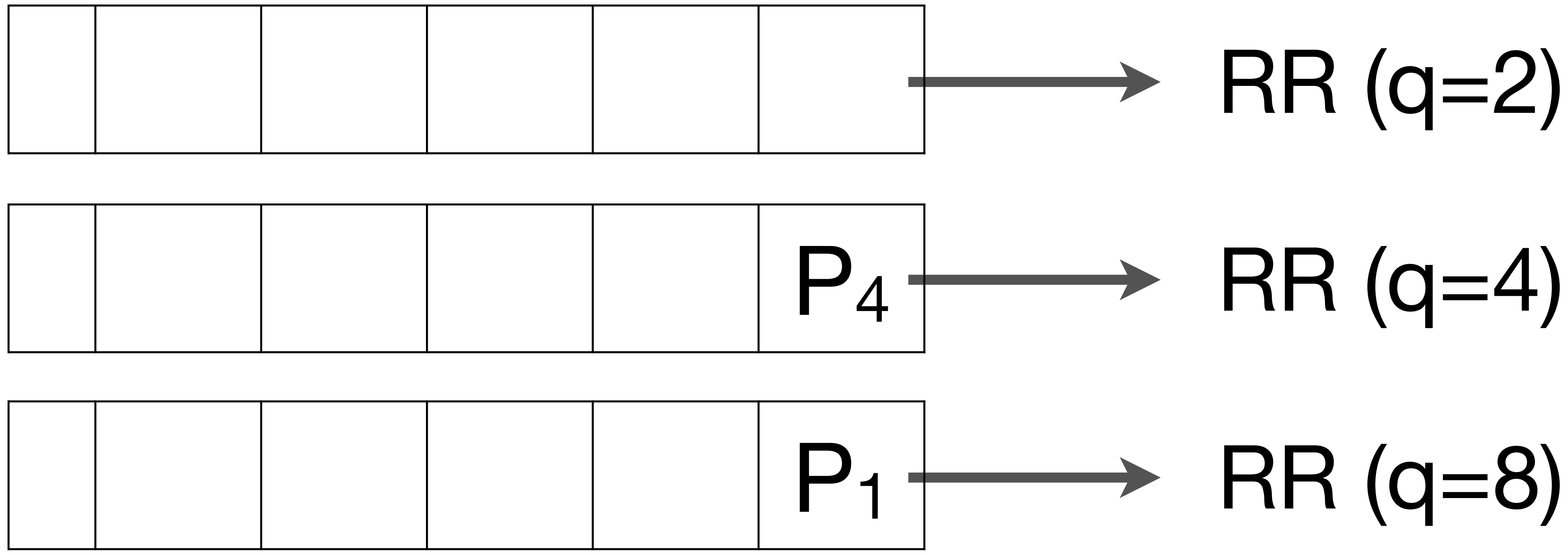


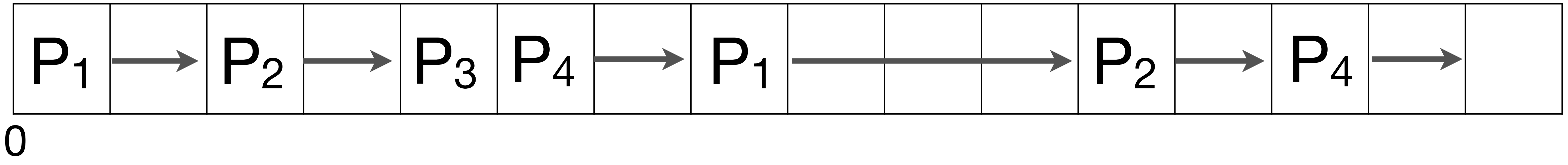
Process	*Arrival Time*	*Burst Time*
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4



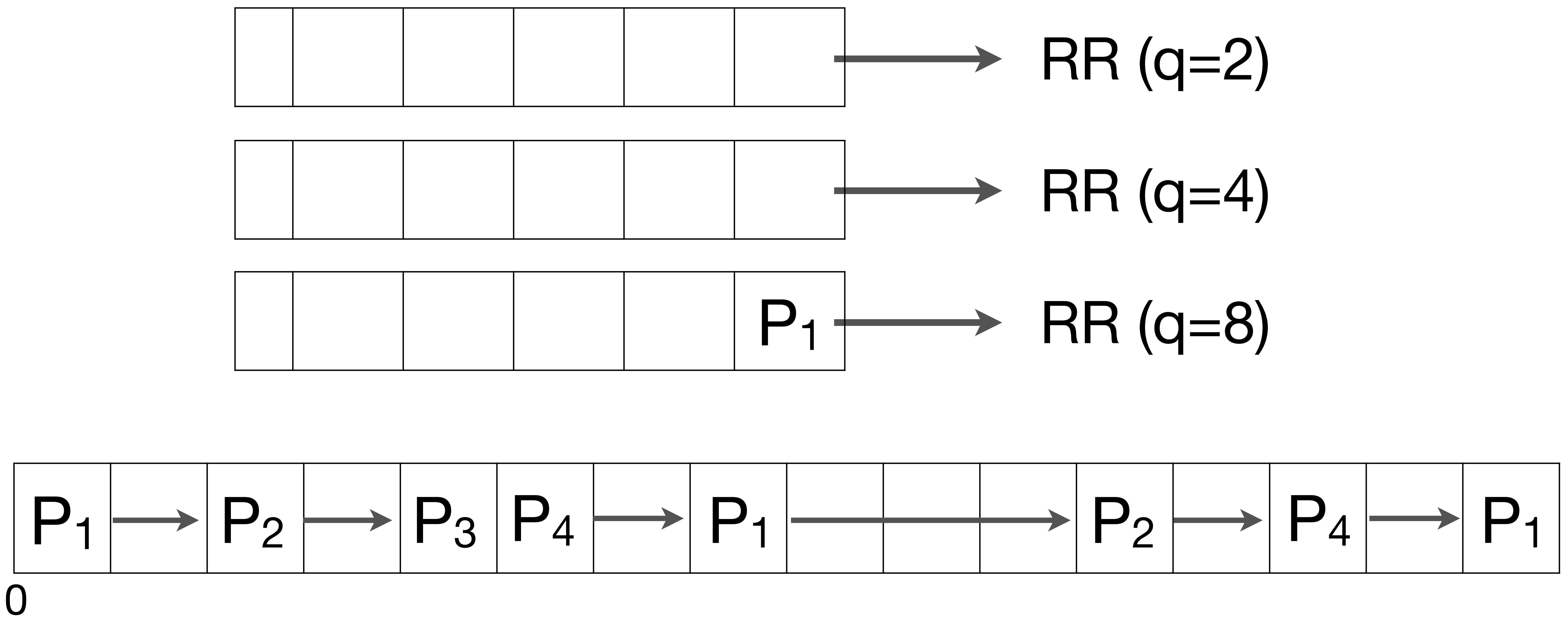


Process	*Arrival Time*	*Burst Time*
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4





Process	*Arrival Time*	*Burst Time*
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

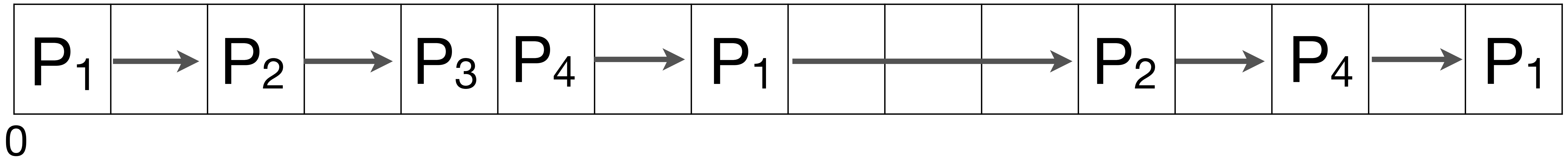


Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

Wait times: $P_1 = 9$, $P_2 = 7$, $P_3 = 0$, $P_4 = 6$

Average: $(9 + 7 + 0 + 6) / 4 = 5.5$

(vs 7 for RR, q=3)

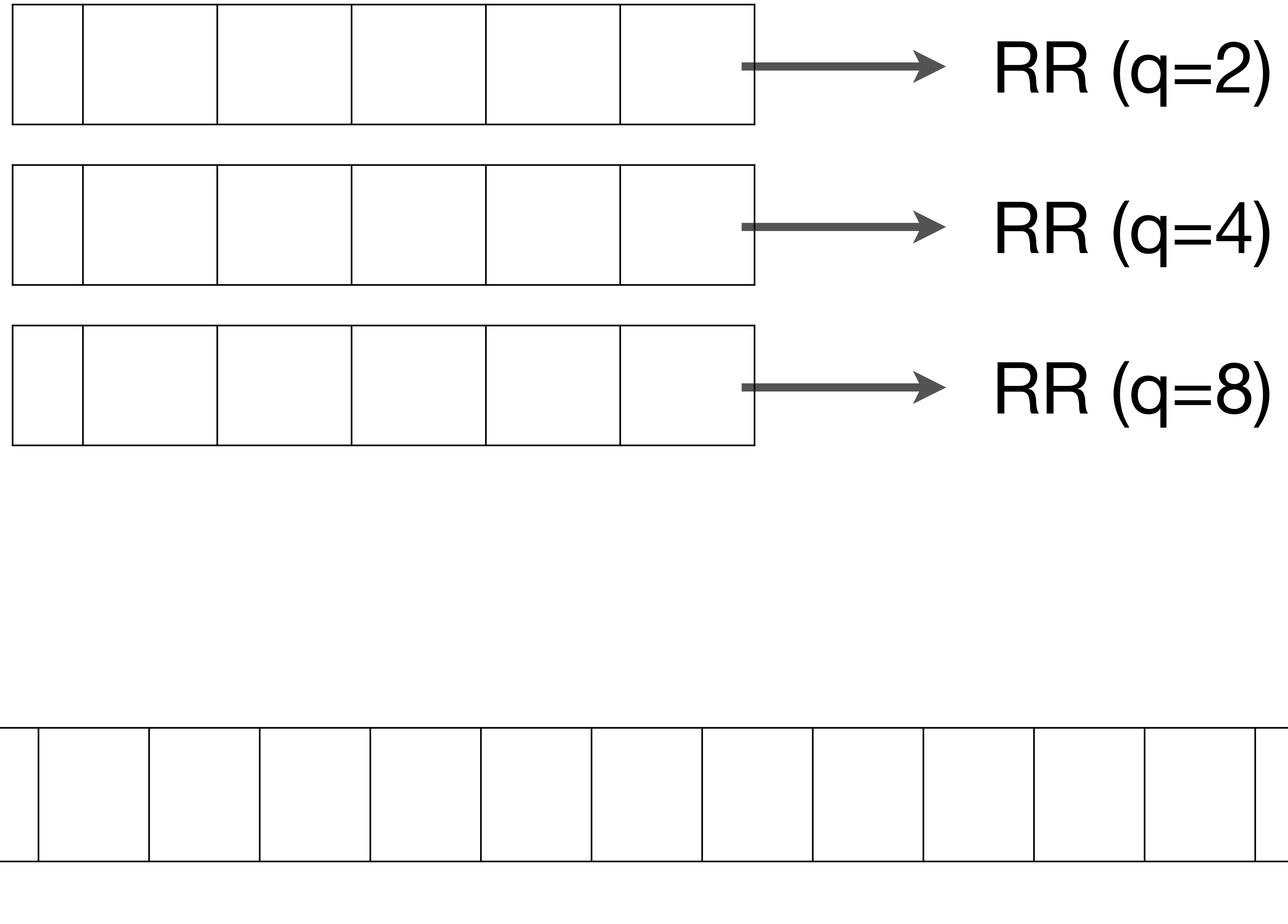


Other rules?

- These rules may be gamed
  - e.g., job may keep relinquishing CPU to retain priority
- May keep track of total time allotment for a job in a given queue and move down when exhausted
- When to move back up?
  - Book suggests moving all jobs to top queue periodically
  - Alternative: move up if job completes burst in less than a quantum

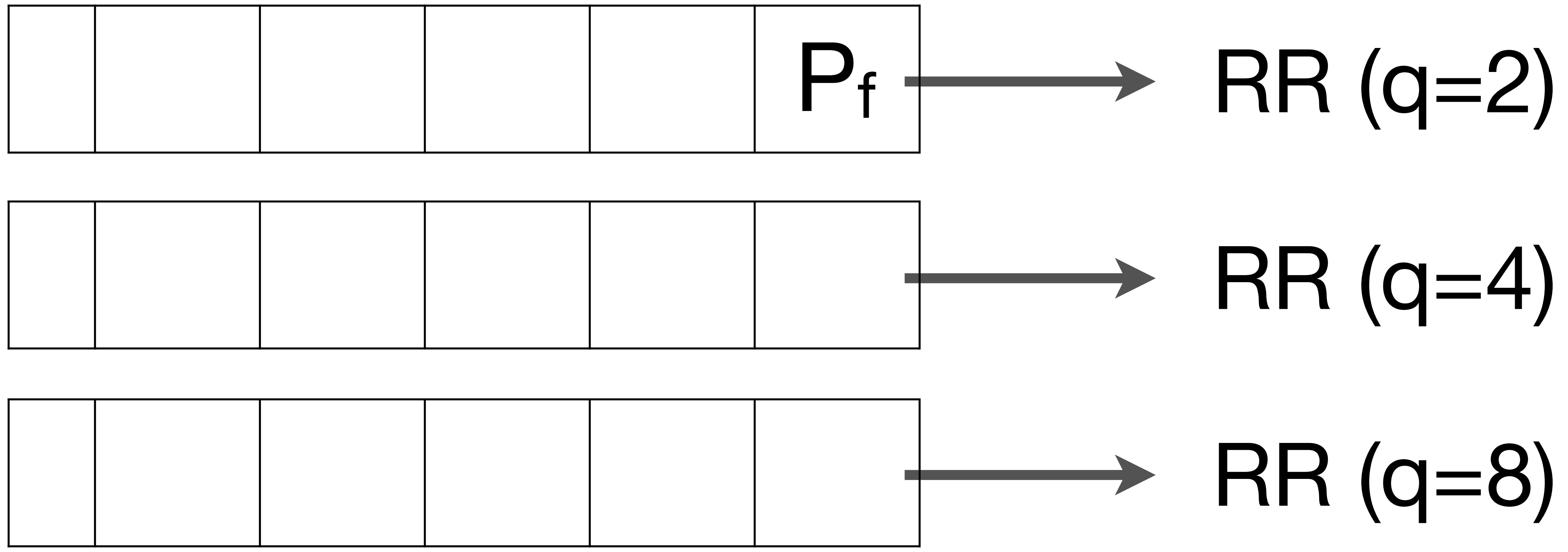
e.g., P_{flaky} arrives at t=0

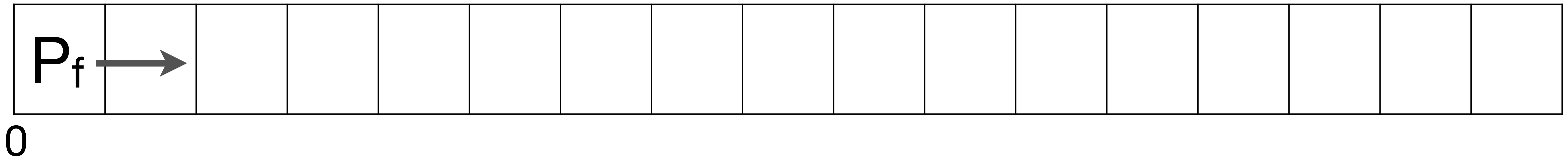
CPU burst lengths = 7, 4, 1, 5 (I/O between)



e.g., P_{flaky} arrives at t=0

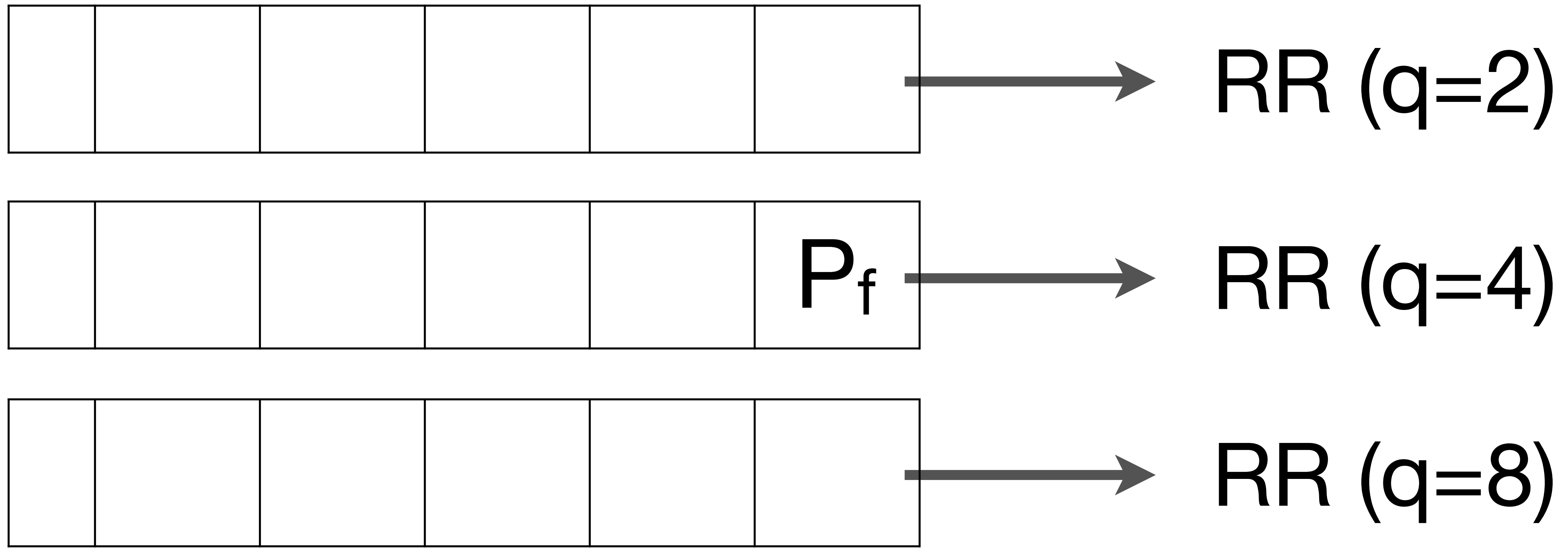
CPU burst lengths = 7, 4, 1, 5 (I/O between)

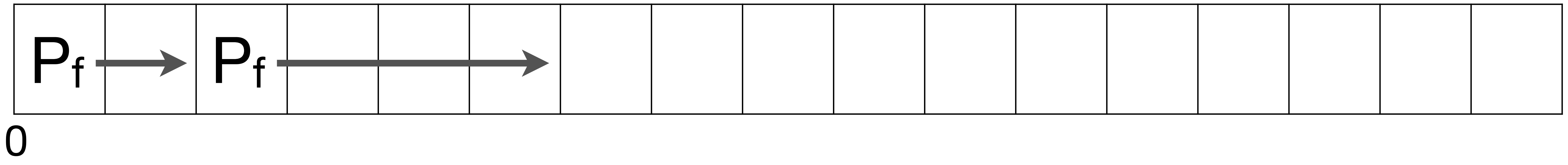




e.g., P_{flaky} arrives at t=0

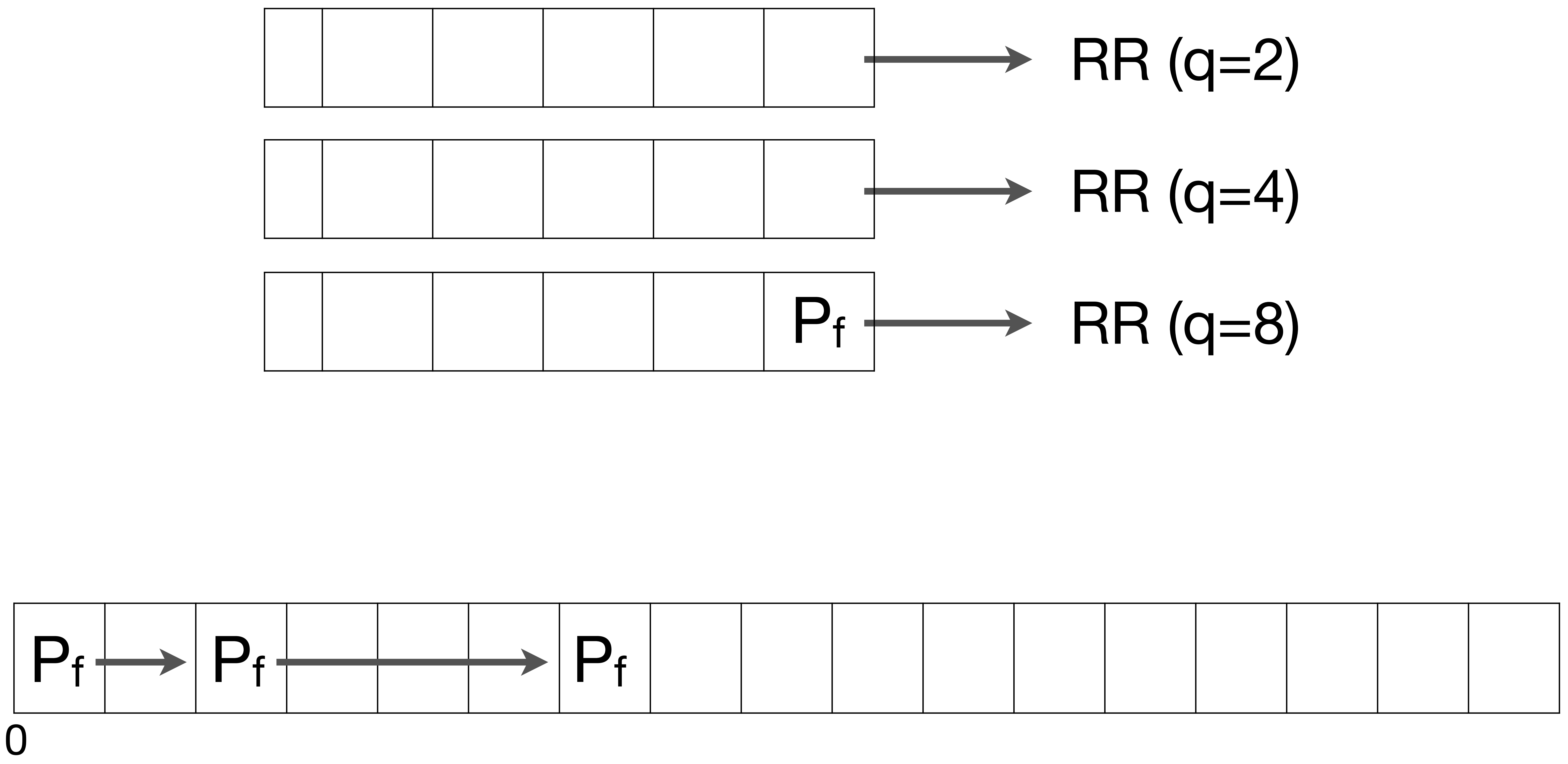
CPU burst lengths = 7, 4, 1, 5 (I/O between)





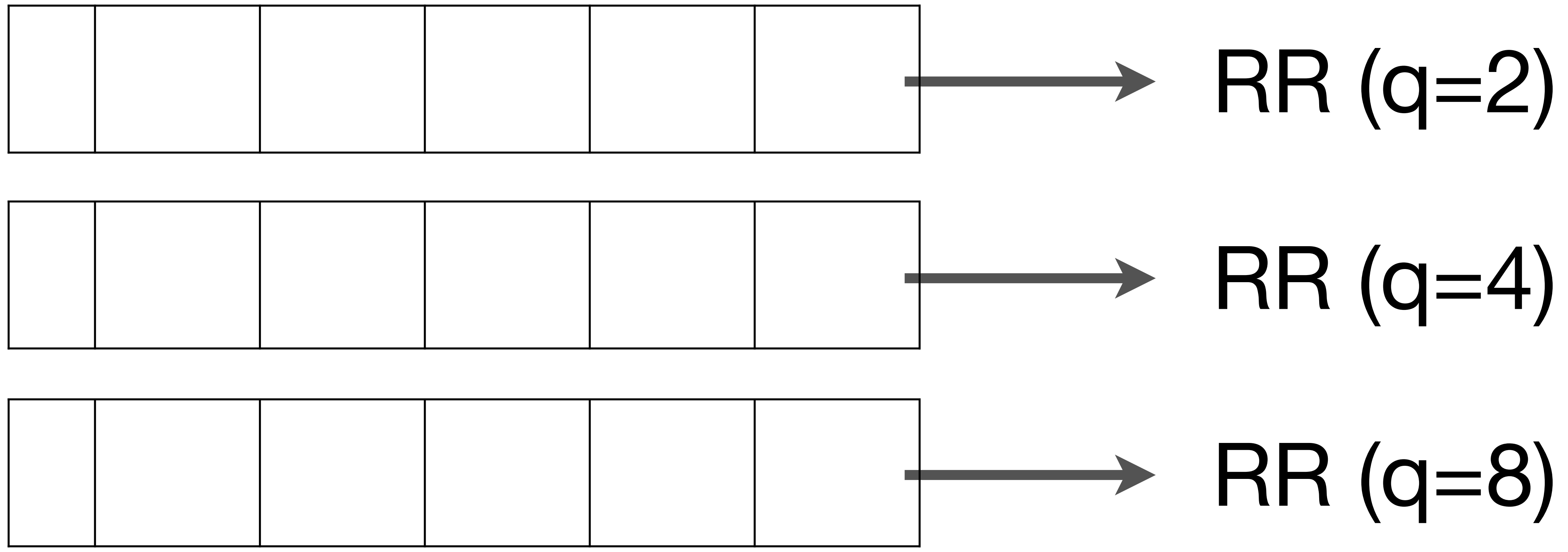
e.g., P_{flaky} arrives at t=0

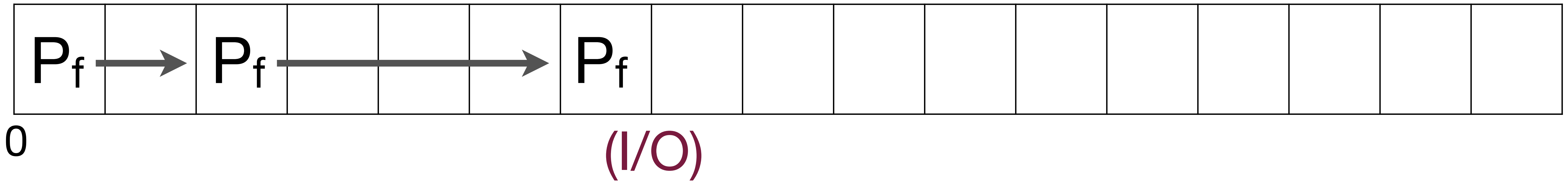
CPU burst lengths = 7, 4, 1, 5 (I/O between)



e.g., P_{flaky} arrives at t=0

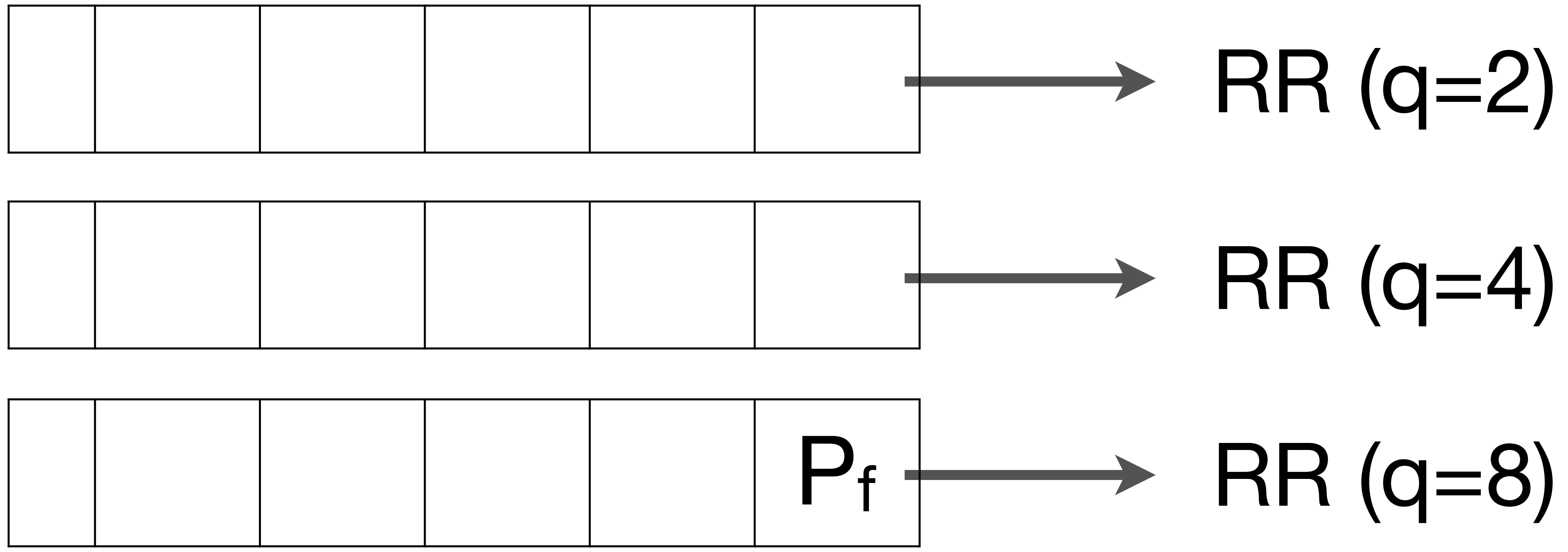
CPU burst lengths = 7, 4, 1, 5 (I/O between)

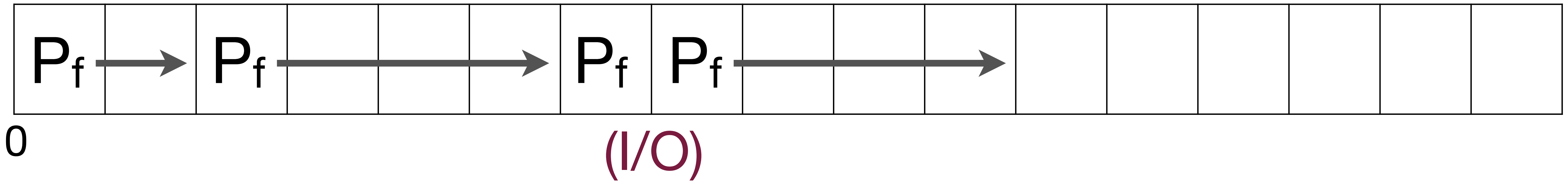




e.g., P_{flaky} arrives at t=0

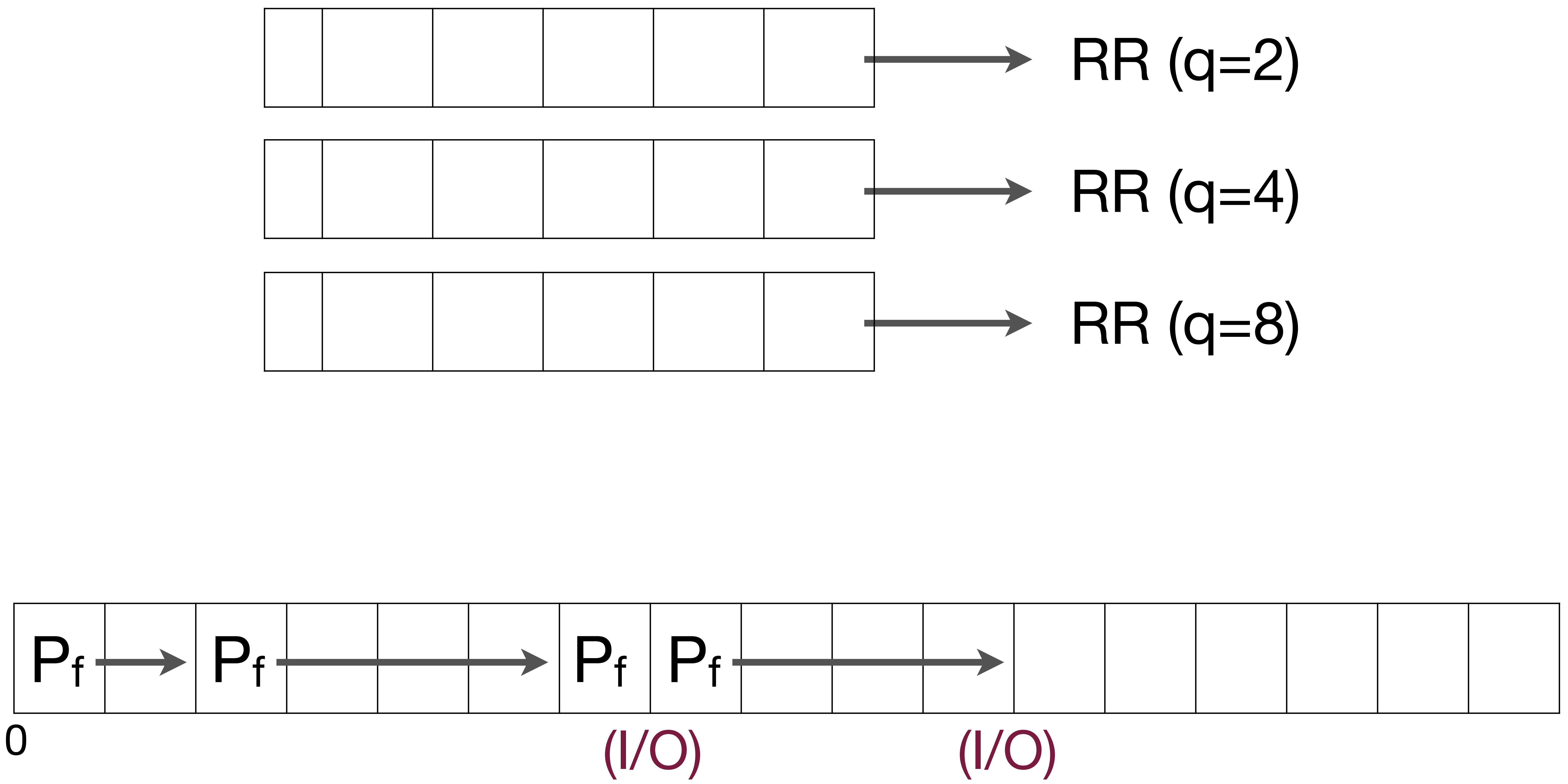
CPU burst lengths = 7, 4, 1, 5 (I/O between)





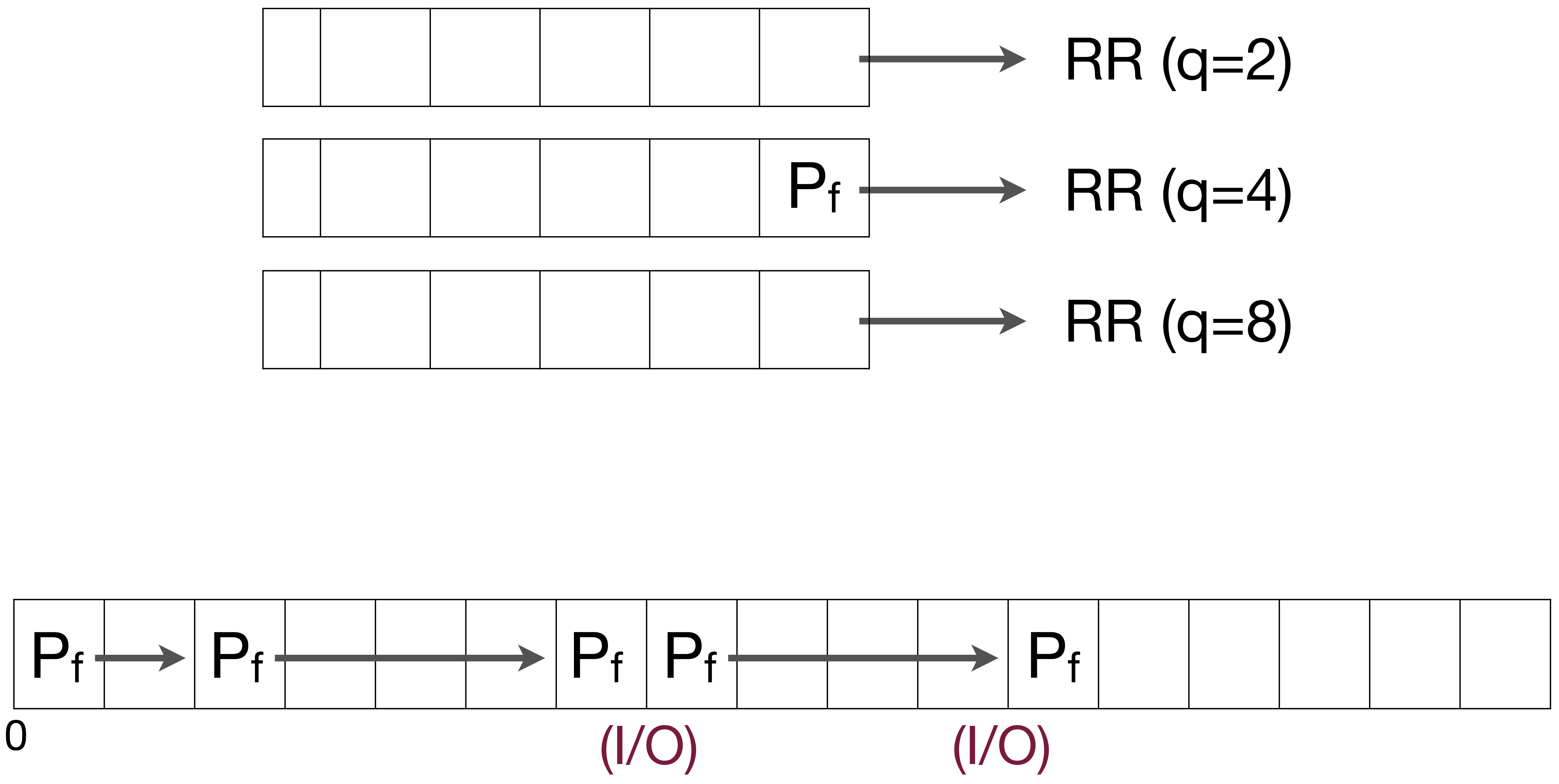
e.g., P_{flaky} arrives at t=0

CPU burst lengths = 7, 4, 1, 5 (I/O between)



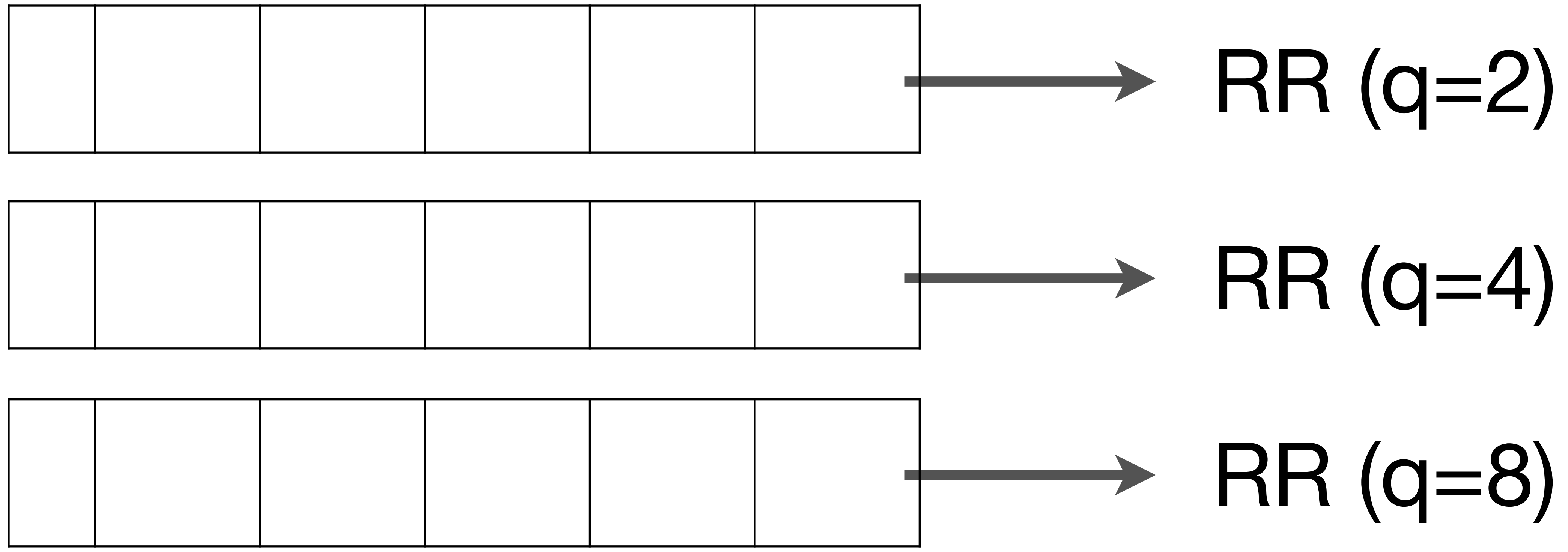
e.g., P_{flaky} arrives at t=0

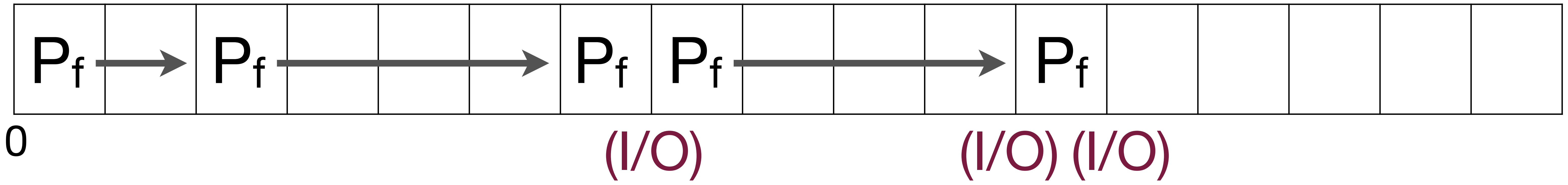
CPU burst lengths = 7, 4, 1, 5 (I/O between)



e.g., P_{flaky} arrives at t=0

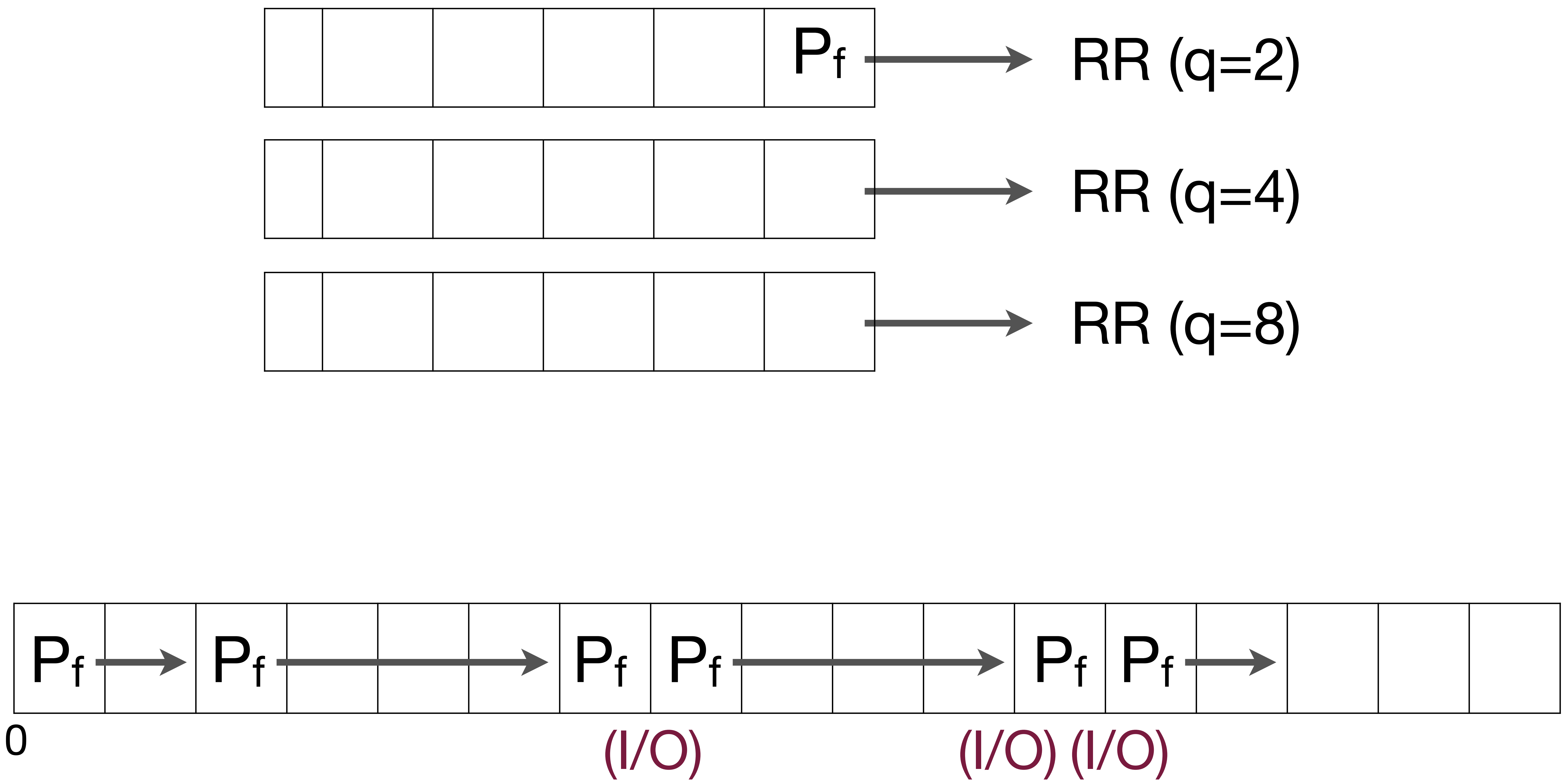
CPU burst lengths = 7, 4, 1, 5 (I/O between)





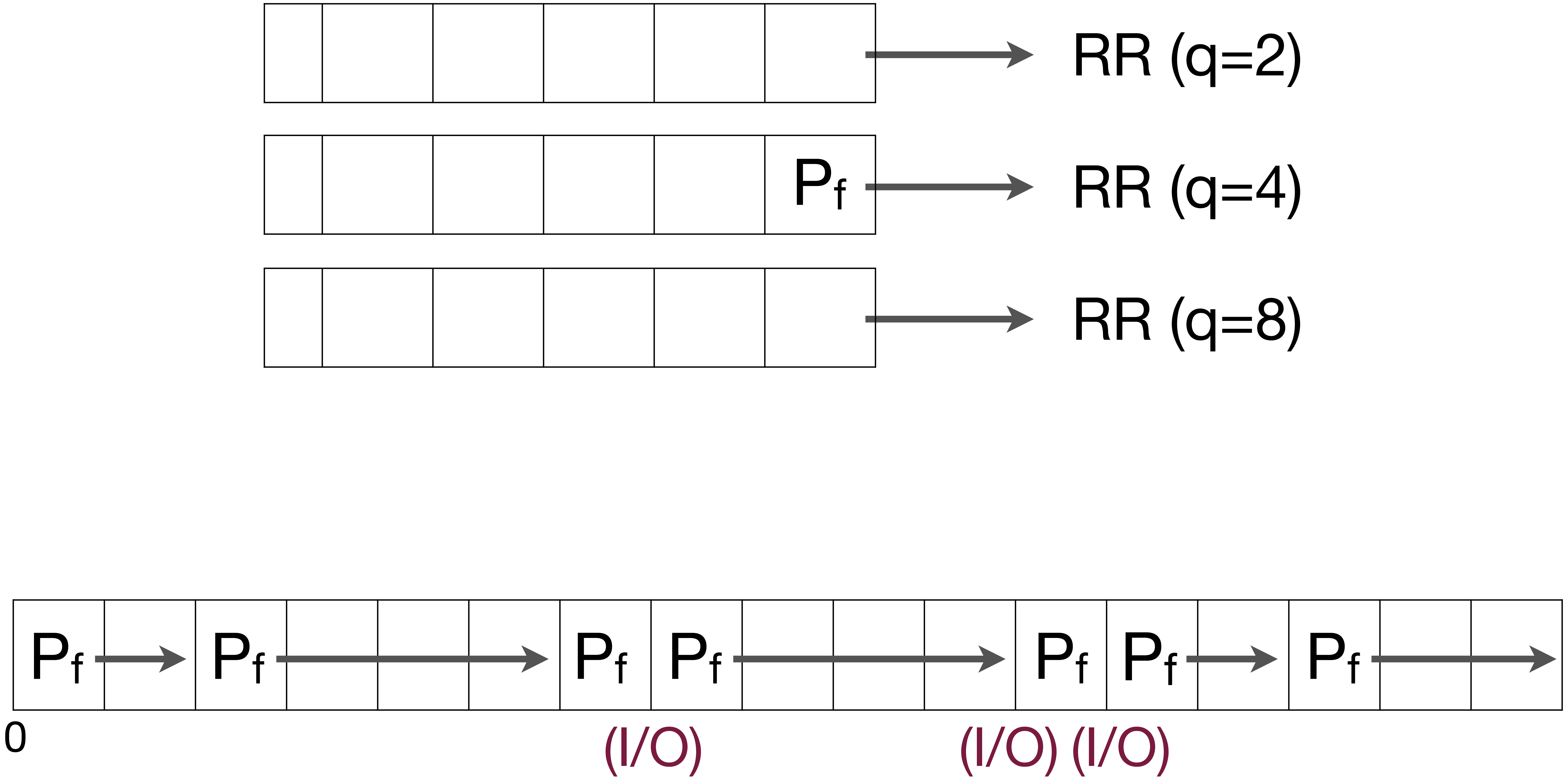
e.g., P_{flaky} arrives at t=0

CPU burst lengths = 7, 4, 1, 5 (I/O between)



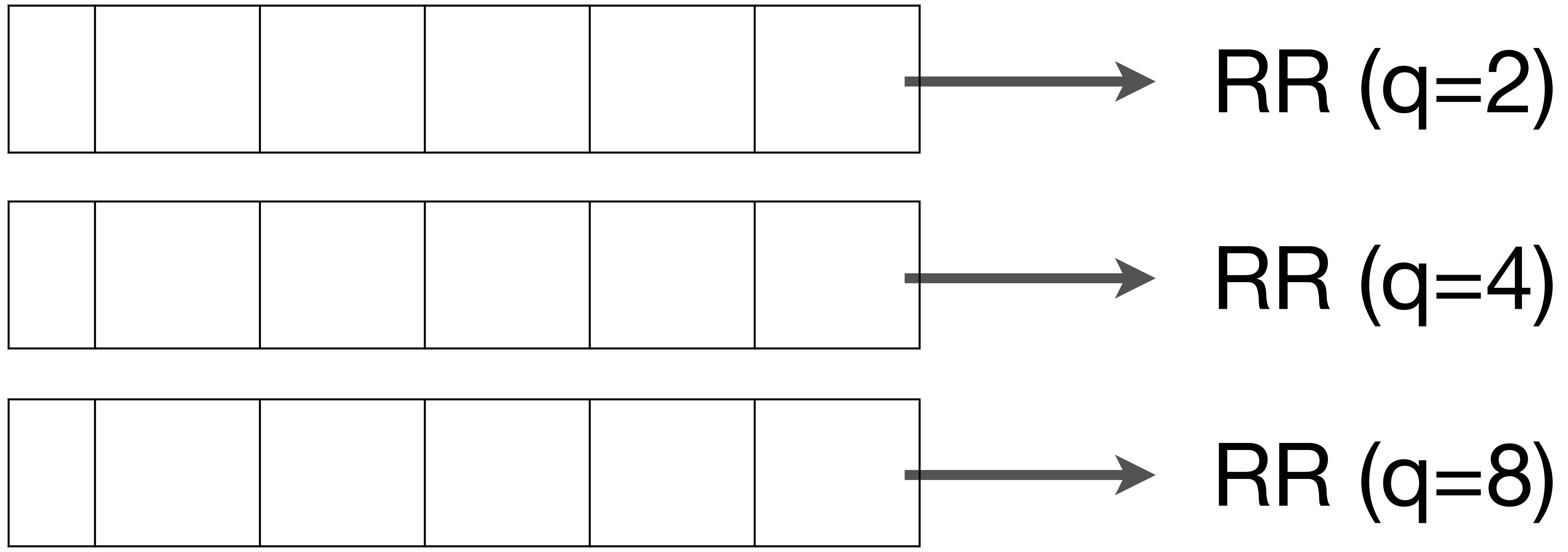
e.g., P_{flaky} arrives at t=0

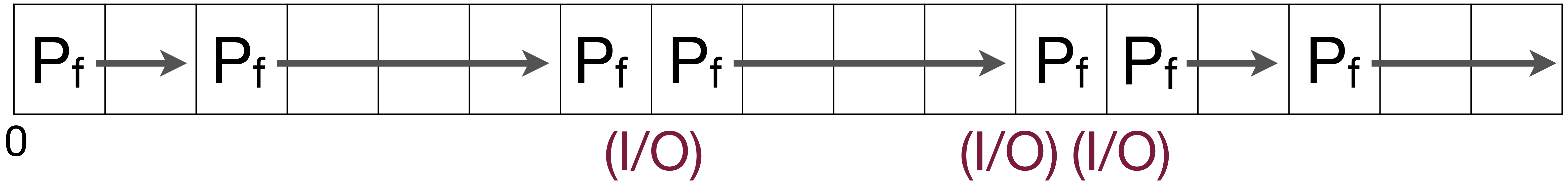
CPU burst lengths = 7, 4, 1, 5 (I/O between)



e.g., P_{flaky} arrives at t=0

CPU burst lengths = 7, 4, 1, 5 (I/O between)





MLFQ summary

- Many parameters may be needed to fine-tune an MLFQ scheduler
  - Behavior may be driven by a combination of heuristics and mathematical/algorithmic optimization
  - Hard to avoid the use of “magic numbers” that work for specific systems and workloads
- MLFQ helps dynamically identify and balance interactive and CPU-bound jobs — a popular choice in modern operating systems!