Operating Systems

Sistemas Operativos

CPU Scheduling

University of Minho

2024-2025

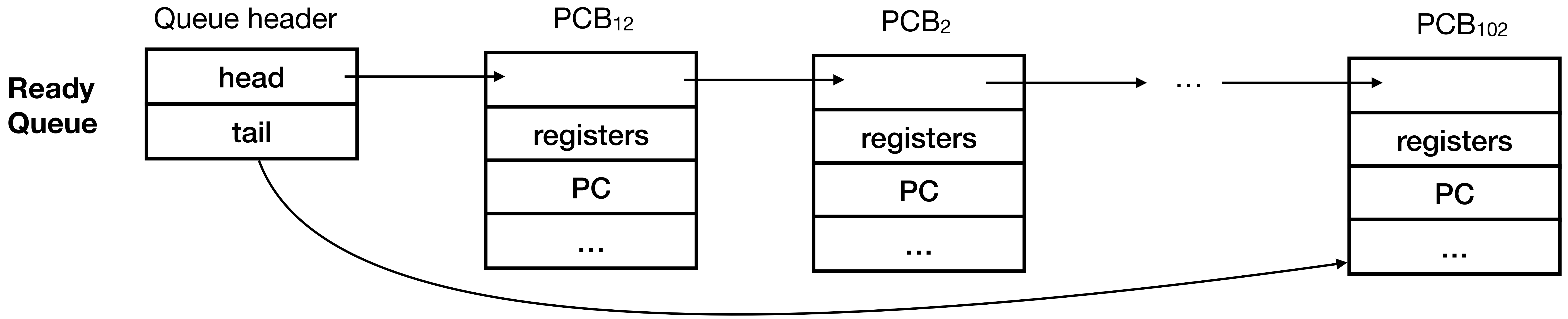
What will we learn?

CPU Management

- Although a computer has a limited number of CPUs, users get the impression that several programs (a lot more than the available CPUs) are running simultaneously
  - How does the OS provide this illusion?
  - How can users use the OS APIs to start and stop programs?
  - How does the OS choose what programs will be running, be switched, …?
- Let us move to the **policies**!

Scheduler

- Number of processes ready to run is significantly higher than the number of available CPU cores
- The PCBs of *Ready* processes (check *Processes slides*!) are kept in a **Ready queue**
- The **OS scheduler** is responsible for choosing the next process to execute
  - By now, you should understand the mechanisms used by the OS to run another process
  - But what about the **policies**? What process should be scheduled next?



Workload Assumptions

Let's start simple…

- **Workload:** set of running processes
  (*Processes* are also referred as **jobs** in the literature)

- Let's define some assumptions to reason about scheduling policies
  1. All processes run for the **same amount of time**[1], and the **time is known**
  2. All processes **arrive** in the system **at the same time**
  3. Once started, each process **runs until completion**
  4. Processes **only use the CPU** (i.e., no I/O is performed)

- Most of these assumptions are unrealistic…
  …we will refined these as we move along

[1]The amount of time that a process is using the CPU is also referred to as the process's **CPU Burst**

Metrics

How can we tell if algorithms are good?

- Let's define a first metric to compare scheduling algorithms
  - **Turnaround time:** the time the process takes to complete after arriving in the system

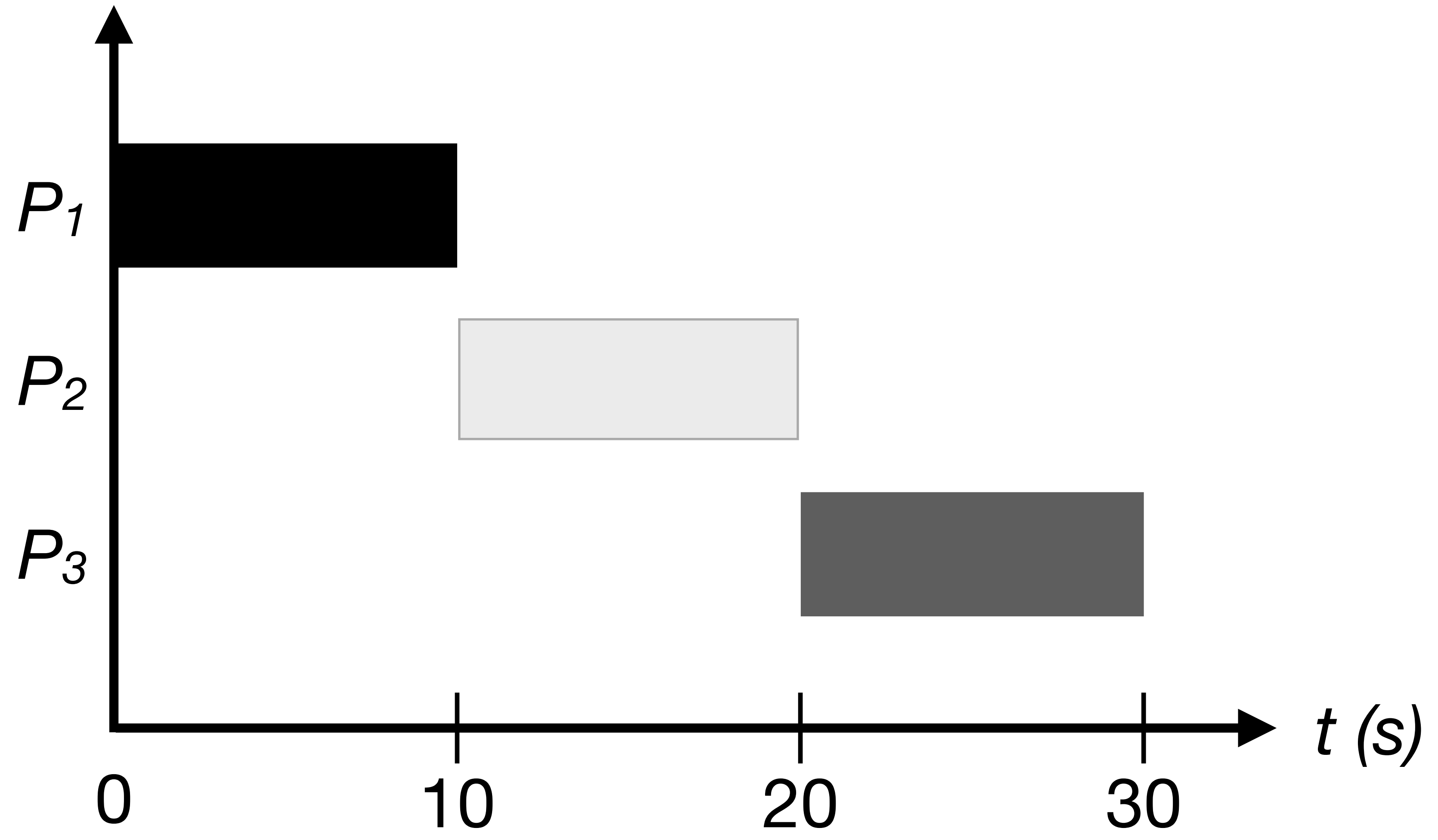
$$T_{turnaround} = T_{completion} - T_{arrival}$$

- The turnaround time is a **performance** metric.
- There are other interesting non-performance metrics, for example
  - **CPU utilization,** as the OS wants to keep the CPU busy running useful instructions!
  - **Fairness**, i.e., if all processes get a chance to execute
  - …

FCFS (aka FIFO)

First come, First Served (aka First in, First out)

- **Selection criteria:** Schedule the process that arrived first (earlier)
- **Example:** Processes P_1, P_2 *and* P_3, each taking 10 seconds to execute
  - **Order of arrival:** $P_1 \rightarrow P_2 \rightarrow P_3$
    (we assume that all arrive at instant $t = 0$, with a very small delay among them)
  - **Average turnaround time:** $\frac{10 + 20 + 30}{3} = 20\ seconds$



FCFS (aka FIFO)

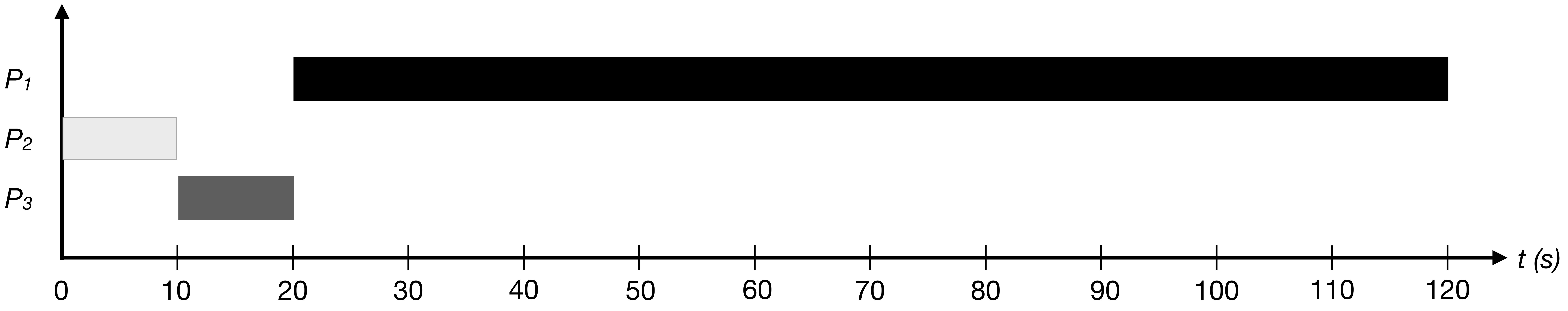
Processes with distinct runtimes

- Lets relax *Assumption 1: all processes run for the same amount of time*
- **Example:** P_1 runs for 100 seconds, while P_2 *and* P_3 run for 10 seconds
- **Questions** (note that our goal is *minimizing the average turnaround time*)
  - Can you think of a workload where FCFS performs well?
  - Can you think of a workload where FCFS performs poorly?

FCFS (aka FIFO)

Good scenario

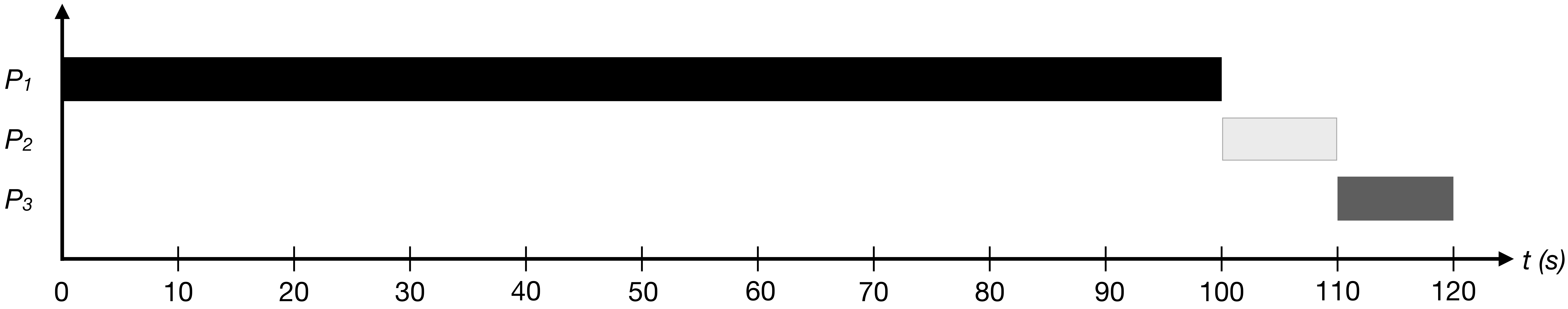
- **Example:** P_1 runs for 100 seconds, while P_2 *and* P_3 run for 10 seconds
  - **Order of arrival:** $P_2 \rightarrow P_3 \rightarrow P_1$
    (all arrived at $t = 0$, with a very small delay among them)
  - **Average turnaround time:** $\frac{10 + 20 + 120}{3} = 50\ seconds$



FCFS (aka FIFO)

Bad scenario

- **Example:** P_1 runs for 100 seconds, while P_2 *and* P_3 run for 10 seconds
  - **Order of arrival:** $P_1 \rightarrow P_2 \rightarrow P_3$
    (all arrived at $t = 0$, with a very small delay among them)
  - **Average turnaround time:** $\frac{100 + 110 + 120}{3} = 110\ seconds$



FCFS (aka FIFO)

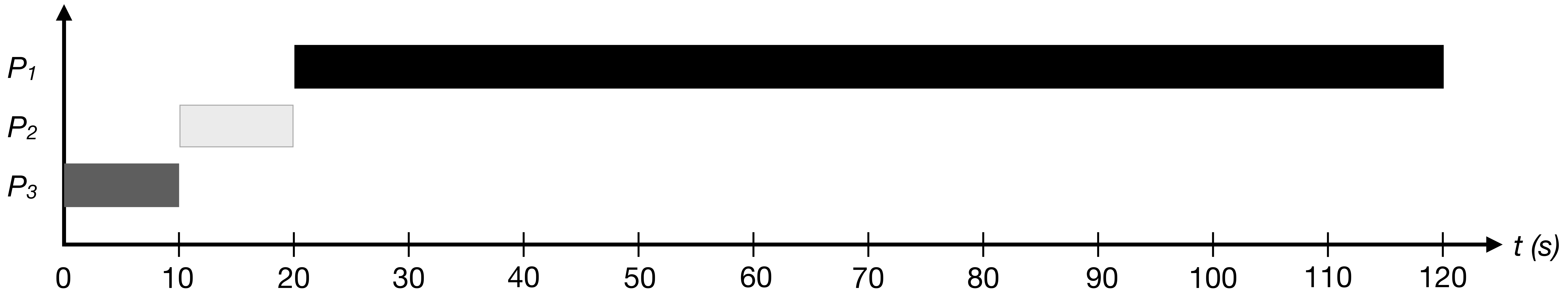
Convoy effect

- **Convoy effect:** short consumers get queued behind a heavyweight one
  - Imagine a supermarket queue, and you got stuck behind a customer with 3 full shopping carts…
- **Question:** Can you think of a better policy that reduces turnaround time?

SJF

Shortest Job First

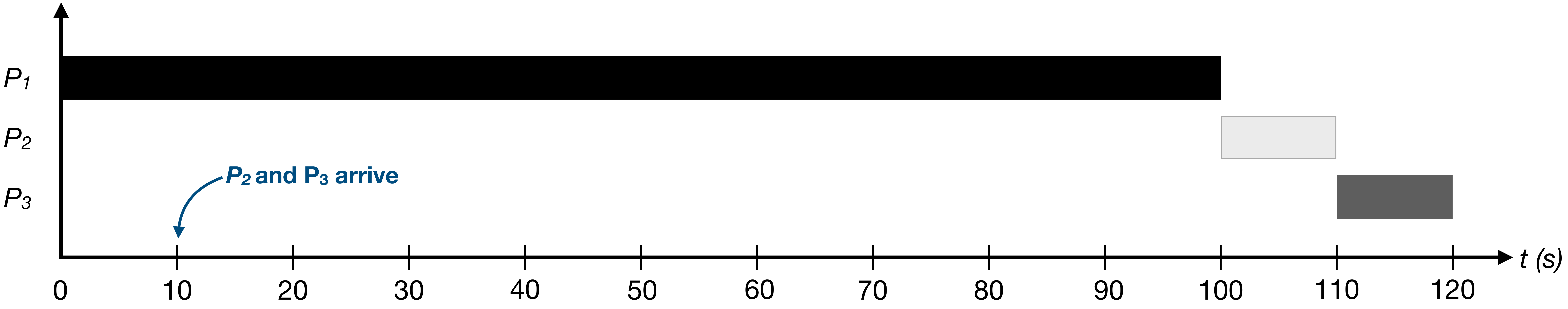
- **Selection criteria:** Run the process with the shortest runtime, then the next shortest, …
- **Example:** P_1 executes for 100 seconds, while P_2 *and* P_3 execute for 10 seconds
  - **Order of arrival:** $P_1 \rightarrow P_3 \rightarrow P_2$ (all arrive at $t = 0$, with a very slight delay among them) Considering this example with SJF, the order of process arrival is irrelevant, right?
  - **Average turnaround time:** $\frac{10 + 20 + 120}{3} = 50\ seconds$ (~2x reduction over FCFS!)



SJF

Processes arriving at different times

- Lets relax *Assumption 2 - all processes arrive in the system at the same time*
- **Example:** P_1 executes for 100 seconds, while P_2 *and* P_3 execute for 10 seconds
  - **Order of arrival:** P_1 arrives at instant $t = 0$, while P_2 *and* P_3 arrive at instant $t = 10$
  - **Average turnaround time:** $\frac{100 + (110 - 10) + (120 - 10)}{3} = 103.33\ seconds$ **(convoy effect!)**



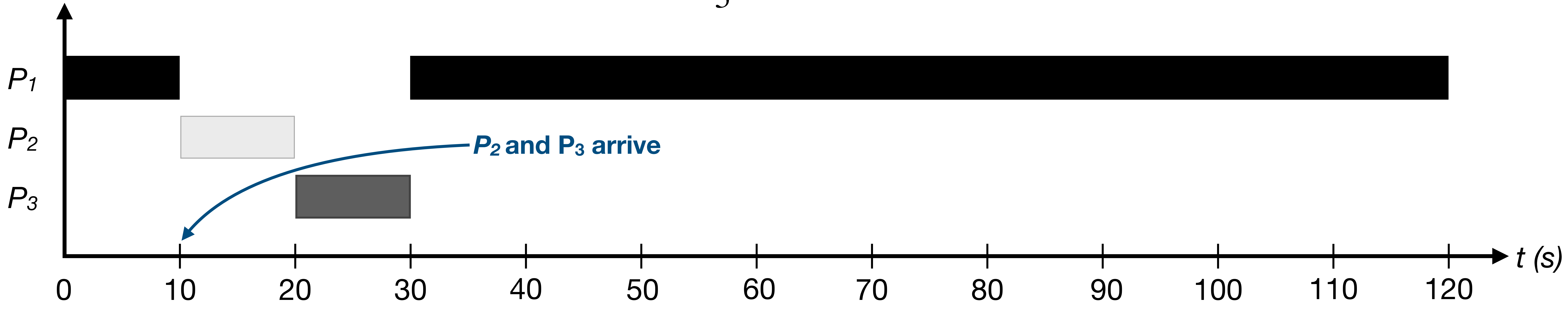
Preemption

- By definition SJF is a **non-preemptive** scheduler,
  - This means that processes run until they yield the CPU
  - To alleviate the previous convoy effect, the scheduler must **preempt** P_1 and run instead another process
- Combine the **timer interrupt** and **context switching** mechanisms!
  - Check the *Scheduling Mechanisms slides*!
- Let's relax *Assumption 3 - Once started, processes runs until completion*

STCF (aka SRT)

Shortest Time-to-Completion First (aka Shortest Remaining Time)

- **Selection criteria:** Anytime a new process enters the system, schedule the process with the least (shortest) runtime left
- **Example:** Process P_1 executes for 100 seconds, while P_2 *and* P_3 execute for 10 seconds
  - **Order of arrival:** P_1 arrives at instant $t = 0$, while P_2 *and* P_3 arrive at instant t = 10
  - STCF must preempt P_1 when P_2 *and* P_3 arrive
  - **Average turnaround time:** $\dfrac{(120 - 0) + (20 - 10) + (30 - 10)}{3} = 50\ seconds$

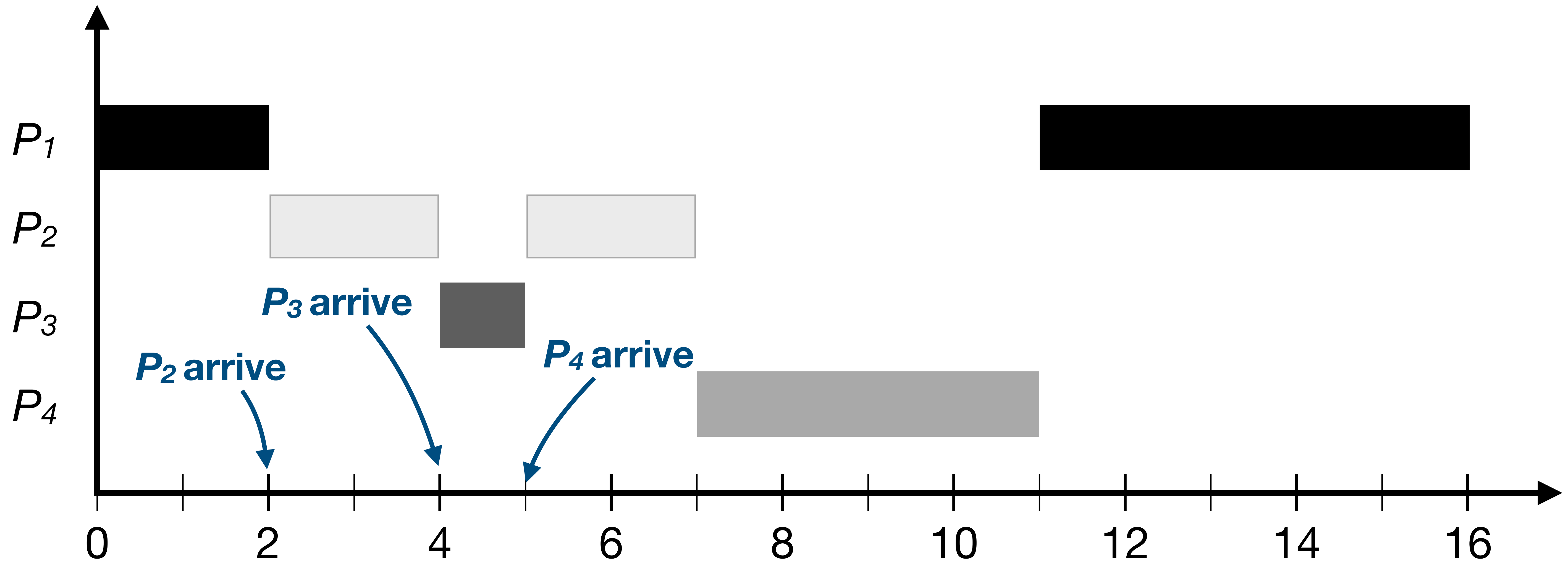


STCF (aka SRT)

Another example

- Assuming the arrival and execution times at the table below
  - **Average turnaround time:** $\frac{(16-0)+(7-2)+(5-4)+(11-5)}{4} = 7.75\ seconds$

Process	**Arrival**	**Execution Time** *(CPU Burst)*
1	0	7
2	2	4
3	4	1
4	5	4



Interactive Systems

Response Time

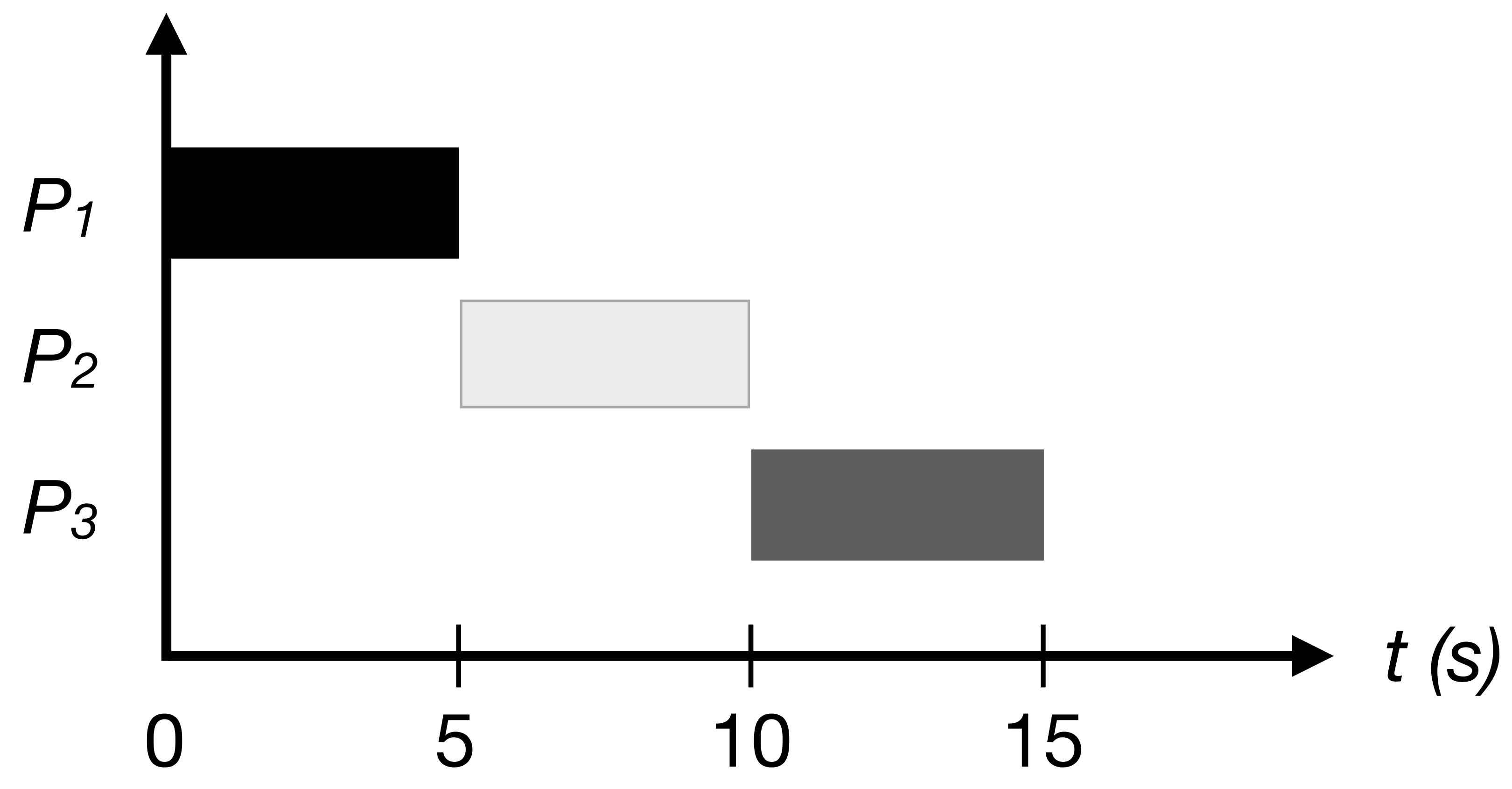
- With our current assumptions and metric, STCF is a good solution!
  - Useful for **batch** systems!
- With **time-shared** computers, users started asking for interactive performance
  - And, thus, a new metric arises: **Response Time**
    (i.e., time from when the process arrives in the system to the first time it is scheduled)

$$T_{response} = T_{firstrun} - T_{arrival}$$

Interactive Systems

Example with SJF

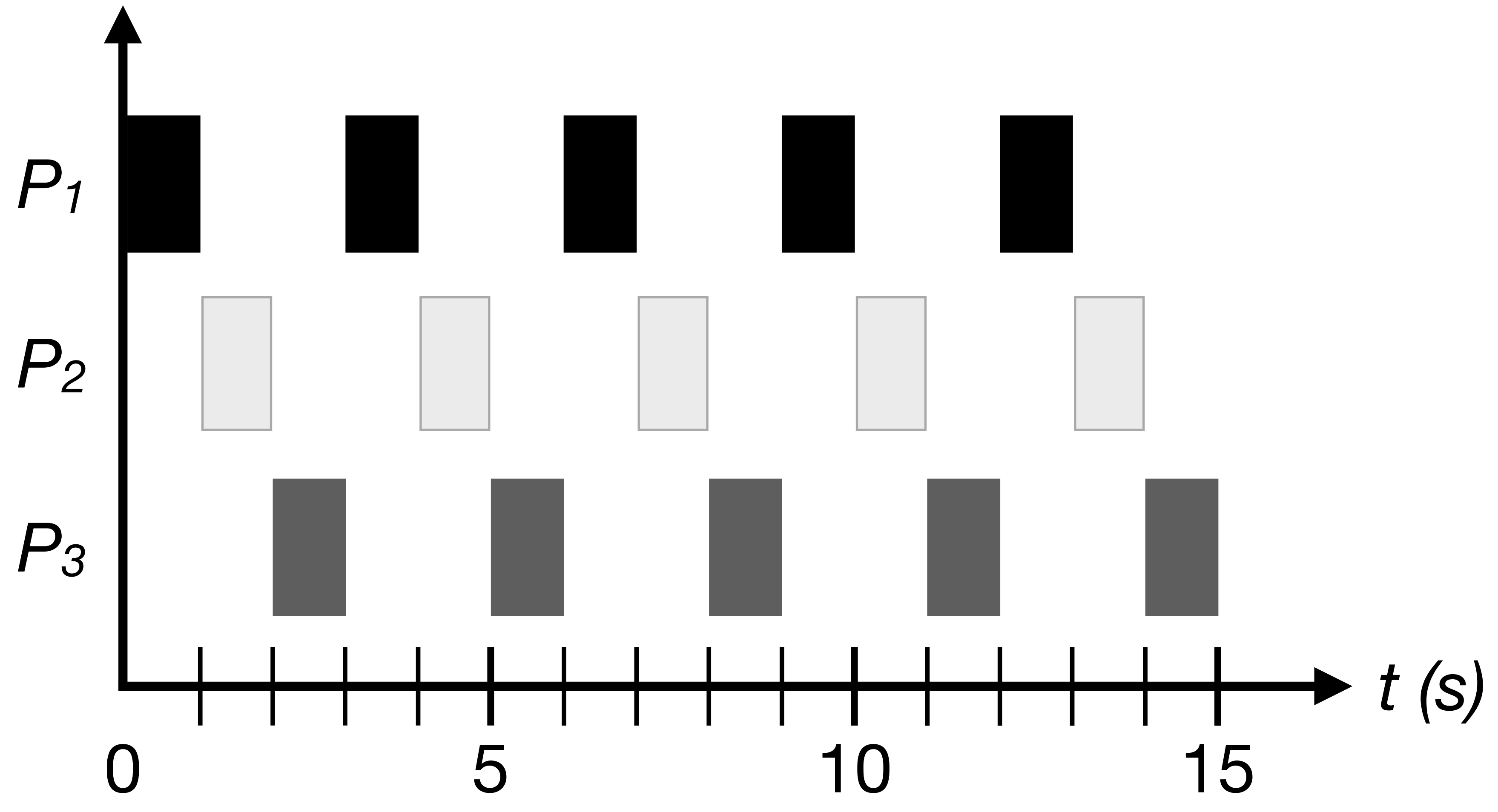
- **Example:** Processes P_1, P_2 *and* P_3, each taking 5 seconds to execute
  - **Order of arrival:** $P_1 \rightarrow P_2 \rightarrow P_3$ (all arrive at $t = 0$, with a very small delay)
  - **Average response time:** $\frac{0 + 5 + 10}{3} = 5\ seconds$



RR

Round Robin (aka time-slicing)

- **Selection criteria:** Run a process for a given time slice (**scheduling quantum**), then switch to the next process in queue (queued processes are served in a FCFS fashion)
- **Example:** Processes P_1, P_2 *and* P_3, each taking 5 seconds to execute
  - **Order of arrival:** $P_1 \rightarrow P_2 \rightarrow P_3$ (all arrived at instant 0, with a very small delay)
  - **Scheduling quantum:** 1 second
  - **Average response time:** $\frac{0+1+2}{3} = 1 \ second$



RR

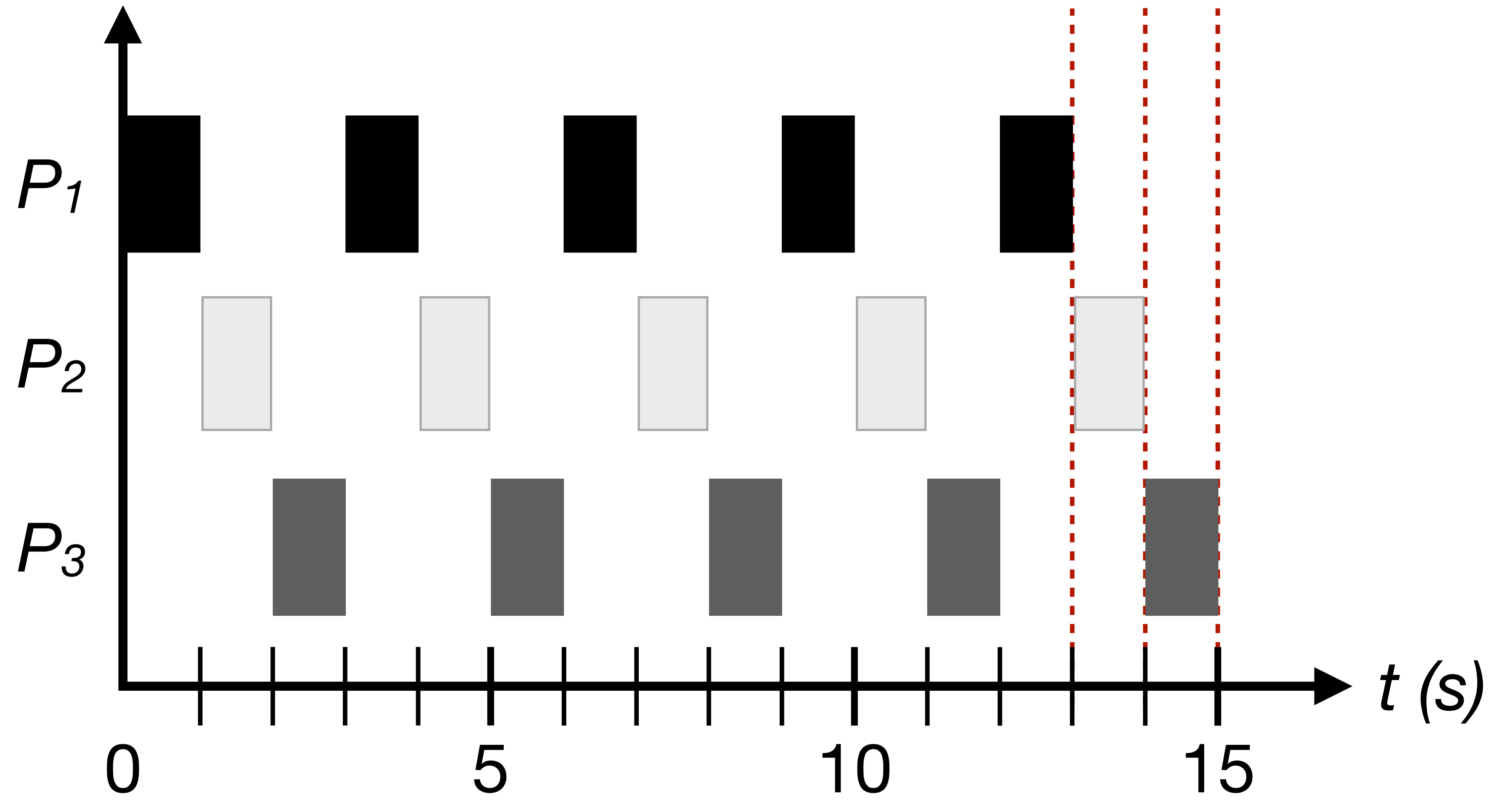
Considerations

- The **scheduling quantum** is critical for RR's efficiency
  - Short values - better response time
  - Large values - similar to a FCFS policy
- However, a very small scheduling quantum can be problematic
  - The cost (time spent) of context switching can dominate the overall performance (more time spent on switching processes than executing processes)
  - The duration of the scheduling quantum must **amortize** the cost of context switching
- Examples (the values are not supposed to be representative of today's systems):
  - *scheduling quantum = 10 ms* and *context-switching = 1 ms* - 10% wasted time
  - *scheduling quantum = 100 ms* and *context-switching = 1 ms* - 1% wasted time

RR

Turnaround time

- **Example:** Processes P_1, P_2 *and* P_3, each taking 5 seconds to execute
  - **Order of arrival:** $P_1 \rightarrow P_2 \rightarrow P_3$ (all arrived at $t = 0$, with a very small delay)
  - **Scheduling quantum:** 1 second
  - **Average turnaround time:** $\dfrac{13 + 14 + 15}{3} = 14\ seconds$



RR

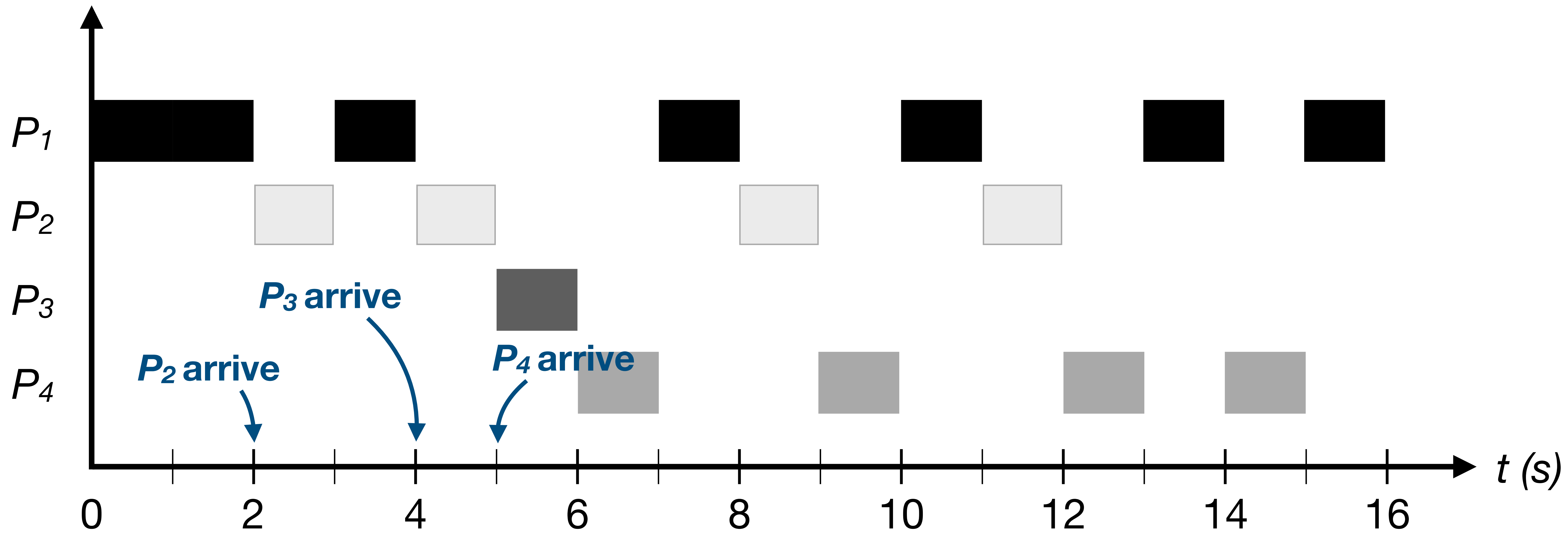
Another example

Homework:
Calculate response and turnaround times for this example with other scheduling algorithms!

◉ Assuming the following arrival and execution times

- **Average response time:** $\frac{(0) + (2-2) + (5-4) + (7-6)}{4} = 0.5 \ seconds$
- **Average turnaround time:** $\frac{(16-0) + (12-2) + (6-4) + (15-5)}{4} = 9.5 \ seconds$

Process	Arrival	Execution Time *(CPU Burst)*
1	0	7
2	2	4
3	4	1
4	5	4



Summary

FCFS, SJF, STCF, RR

- FCFS (FIFO)
  - Simple and easy to implement!
  - Turnaround time is very sensitive to the processes order of arrival
- SJF and STCF
  - Both improve turnaround time. STCF is better when processes arrive at distinct times
  - Both require knowing runtime in advance, and are bad when considering response time
- RR
  - A fair scheduler, good for improving response time (important for interactive systems)
  - One of the worst algorithms for turnaround time (fair algorithms are bad for this metric)
- Tradeoff between optimizing turnaround time and response time

Accounting for I/O

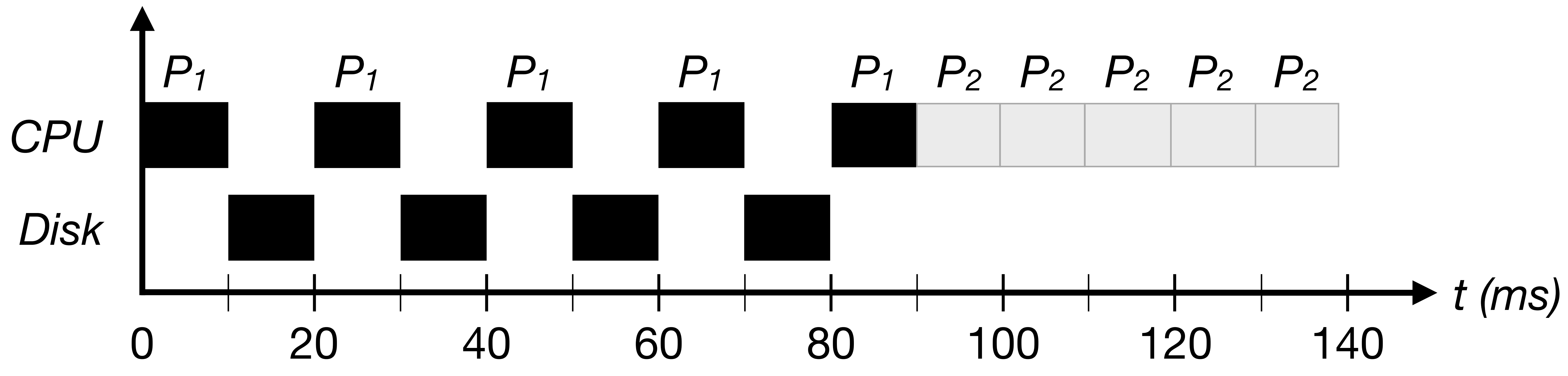
Scheduler decisions

- Lets relax *Assumption 4 - processes only use the CPU (i.e., no I/O is done)*
- Scheduler decisions
  - If a process blocks for I/O (trap) should another process be scheduled?
  - When I/O is done (interrupt operation) should the corresponding process be resumed immediately?

Accounting for I/O

Naive approach - without overlap

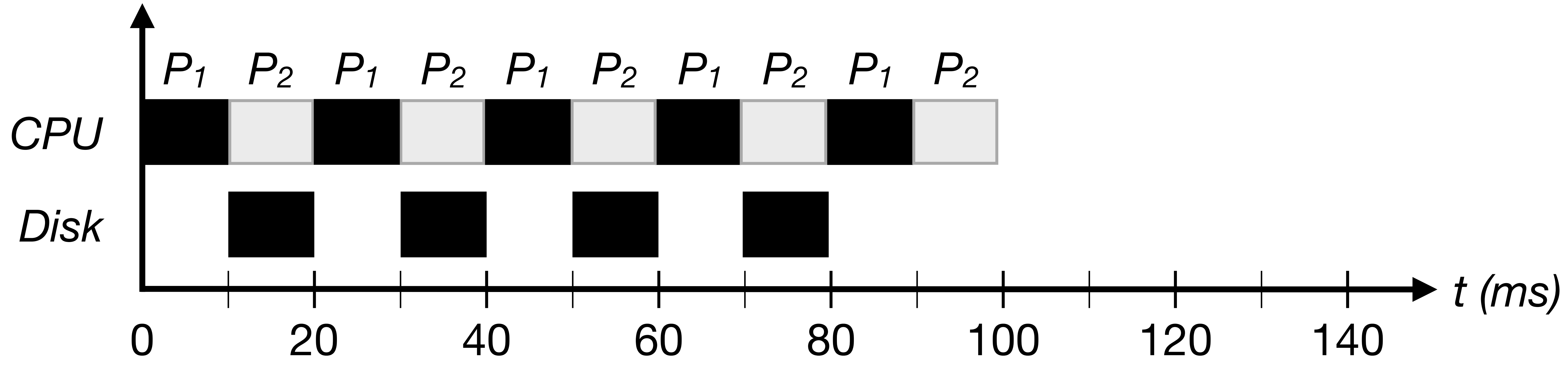
- **Example:** Processes P_1 *and* P_2 *execute for 50 ms*
  - P_1 executes I/O operations every 10ms, P_2 only uses CPU
- The scheduler runs P_1 and then P_2



Accounting for I/O

Resourse overlap

- Let's assume an algorithm based on STCF, but
  - Every 10 ms time slice from P_1 is treated as an independent process
  - Interactive processes (blocking for I/O) run frequently
  - CPU-intensive processes run while interactive ones block for I/O (overlap)
  - Better turnaround and response times (**Homework:** calculate them!)



MLFQ (Multi-Level Feedback Queue)

No more Oracle

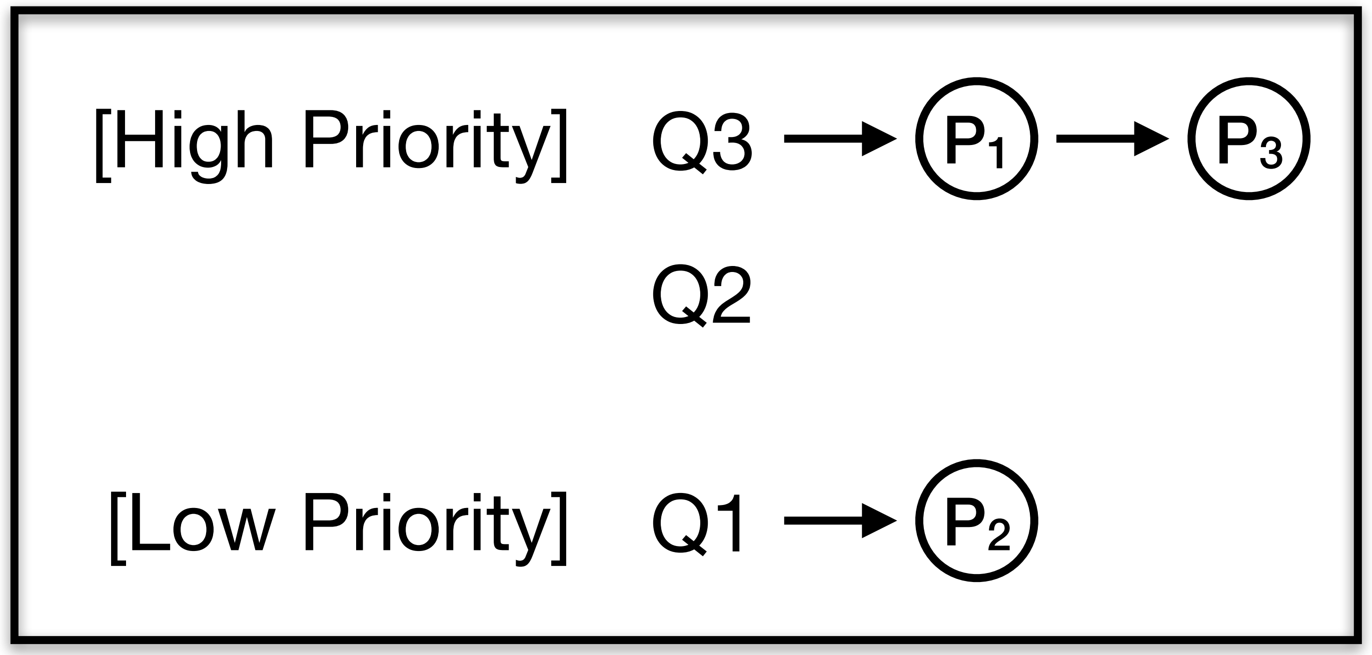
- Lets relax our last *Main Assumption - the runtime of processes is known*
- OS goals
  - Optimize turnaround time, like in SJF or STCF
    (but without knowing the runtime of processes…)
  - Make the system feel responsive to interactive users, like in RR
    (good response time)
- Solution
  - The scheduler must combine multiple scheduling policies
  - The scheduler must learn from the past to make better decisions in the future!

Note: This work led to the award of a Turing Award to its author Fernando J. Corbató

MLFQ

Basic rules

- Multiple queues, ordered according to their **priority**
- Priorities are used to choose what process runs next
  - **Rule 1:** If Priority(P_1) > Priority(P_2), P_1 runs (P_2 does not)
  - **Rule 2:** If Priority(P_1) = Priority(P_3), P_1 & P_3 run in RR

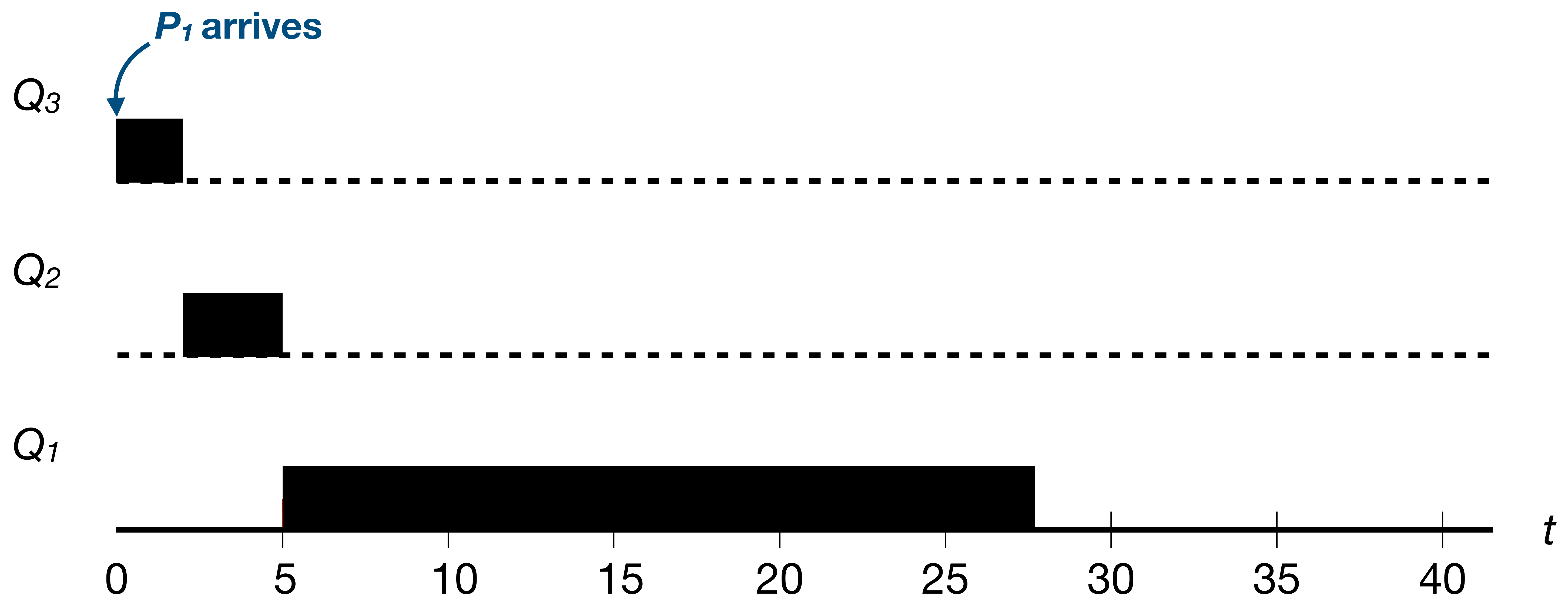


- How are priorities assigned to processes and, do these change over time?
  - Compensate well behaved processes that relinquish CPU frequently
  - Demote processes that are CPU intensive
  - **Allotment**: the amount of time a process can spend at a given priority level
    - The time slice for each queue (priority level) may change
      (e.g., longer time slices for low priority queues)

MLFQ

Queue movement

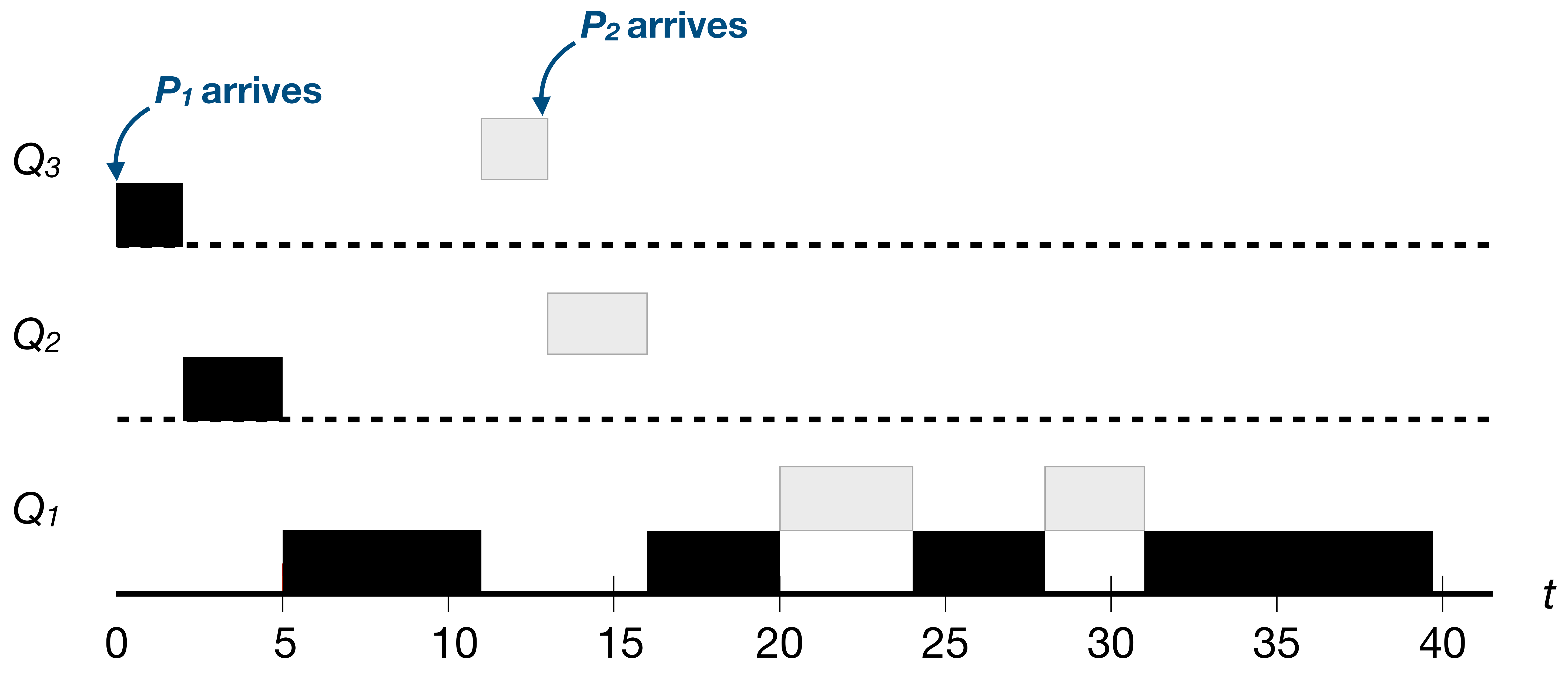
- **Rule 3:** When a process enters the system it has the highest priority (top queue)
- **Rule 4:** Once a process uses up its time allotment at a given level, its priority is reduced (i.e., moves down one queue)



MLFQ

Preemption of low priority processes

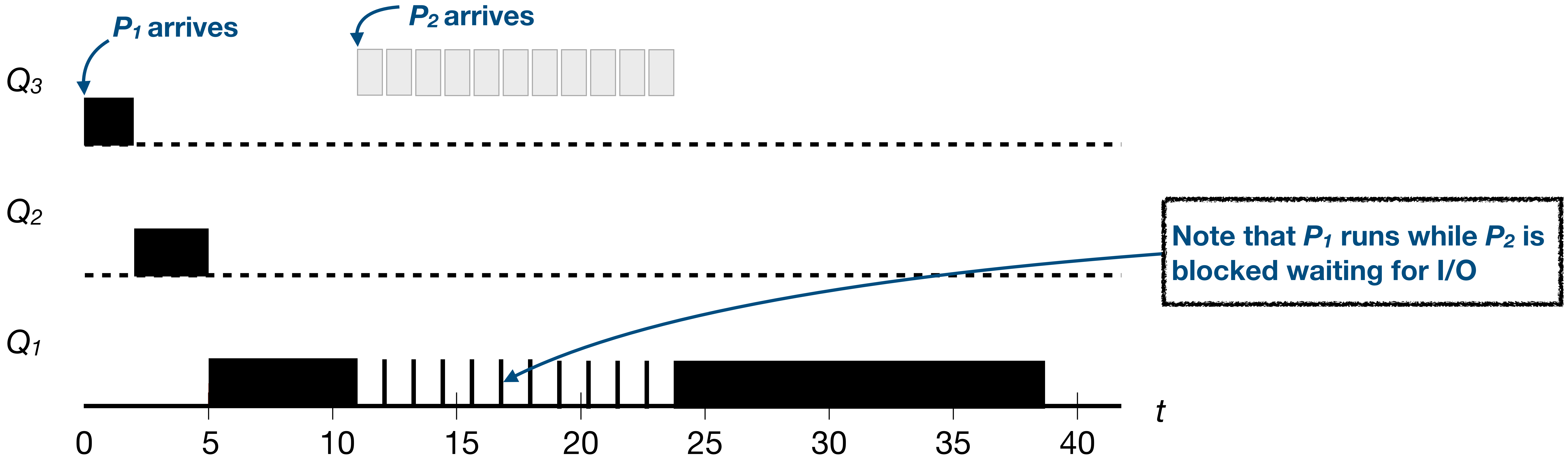
- When new processes, with higher priority arrive, low priority processes executing are preempted
- P_1 is preempted (does not run) until P_2 has the same priority (both are at Q_1)



MLFQ

Gaming

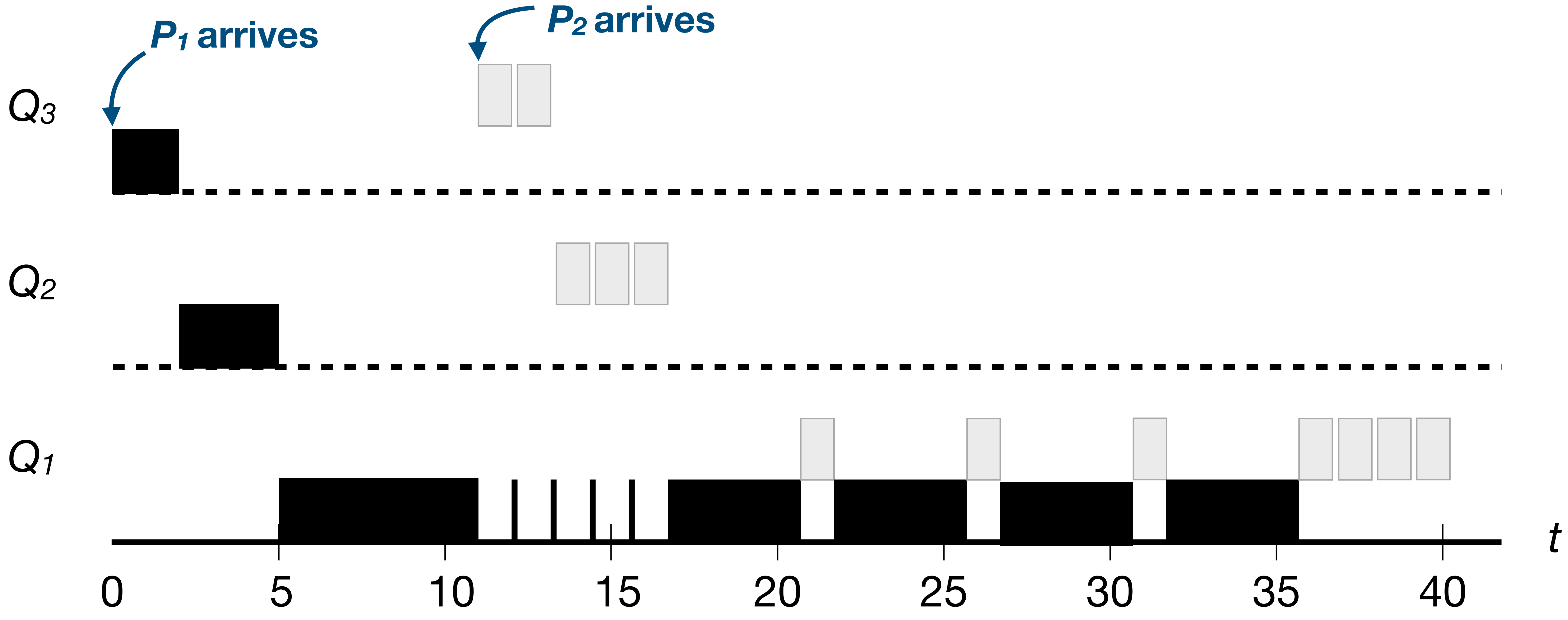
- Interactive processes (e.g., P_2) may relinquish the CPU (block for I/O) before their allotment expires. Should their priority be kept?
  - Bad users could trick (*game*) the OS and starve other processes by writing program that repeatedly use 99% of the allotment and then yield the CPU



MLFQ

Better Accounting

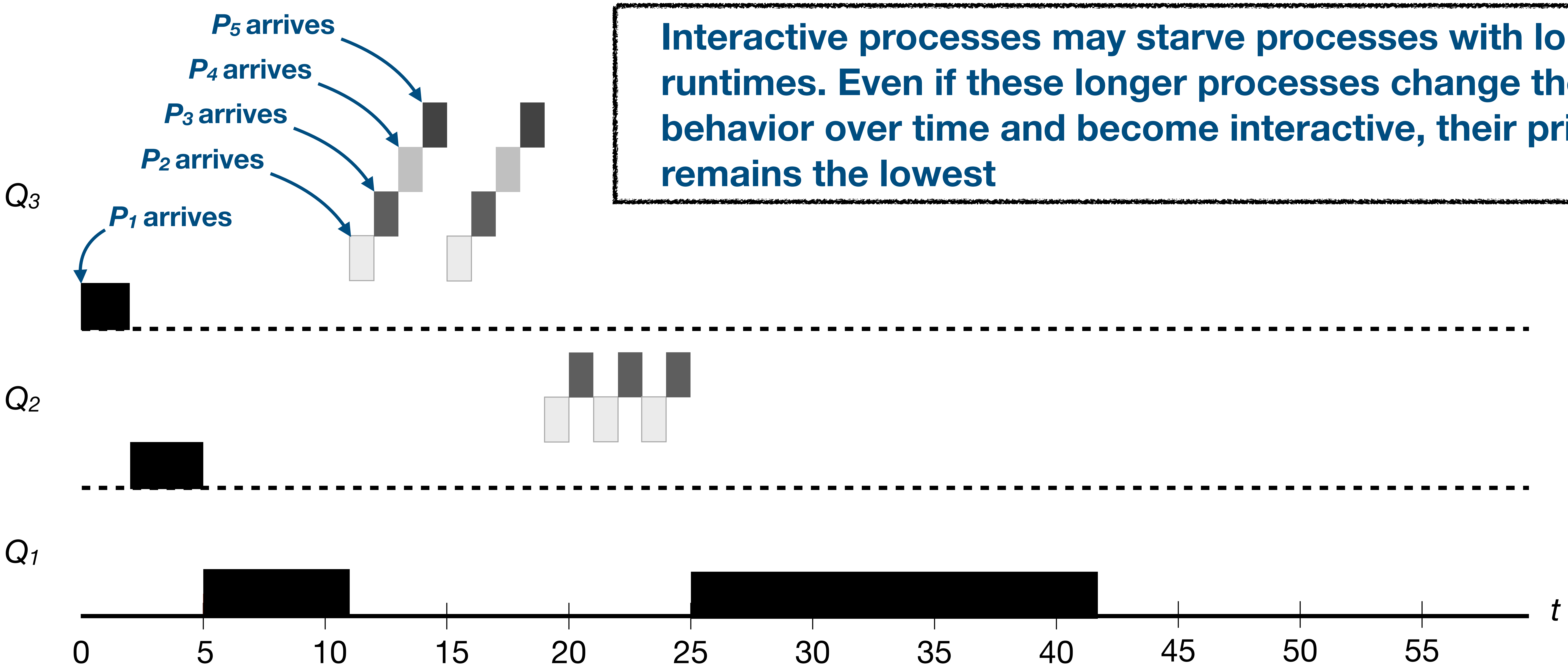
- **Updated Rule 4:** Once a process uses up its time allotment at a given level **(regardless of how many times it has given up the CPU)**, its priority is reduced (i.e., moves down one queue)
  - Thus, the allotment time must consider the sum of all CPU bursts of a process



MLFQ

Starvation

- **Starvation** of processes with longer runtimes may still exist

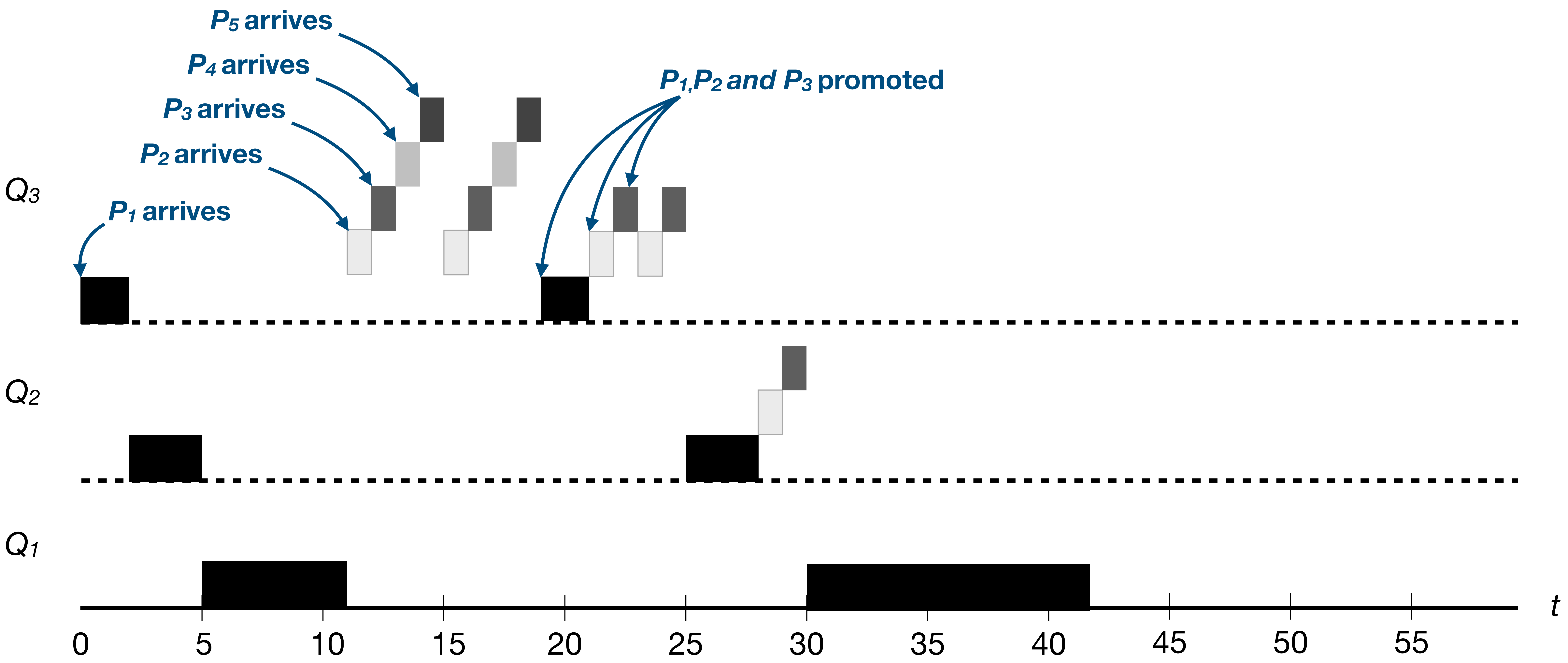


Interactive processes may starve processes with longer runtimes. Even if these longer processes change their behavior over time and become interactive, their priority remains the lowest

MLFQ

Priority Boost (aging mechanism)

- **Rule 5:** After a time period S, move all the jobs to the topmost queue



MLFQ
Summary

- Short duration processes run with high priority (approximates STCF)
  - Good for turnaround time
- High priority processes are frequently switched (approximates RR)
  - Good for response time
- Challenge: several tuning knobs in MLFQ
    - How many queues?
    - How long should the allotment time per queue be?
    - When should the priority boost be called?
  - No easy answer
    - One must know the OS and workloads running there

More Information

- **Chapters 7 and 8** - Remzi H. Arpaci-Dusseau, Andrea C. Arpaci-Dusseau. **Operating Systems: Three Easy Pieces.** Arpaci-Dusseau Books, 2018.
- Avi Silberschatz, Peter Baer Galvin, Greg Gagne. **Operating System Concepts (10. ed)**. John Wiley & Sons, 2018.

Proportional-Sharing

Interesting topics - not covered in detail in this course

- Proportional-share (aka fair-share) schedulers
  - **Goal:** give each job a fair percentage of CPU (**fairness** metric)
  - Interesting algorithms
    - Lottery Scheduling
    - Stride Scheduling
    - Completely Fair Scheduler (CFS)
- **CFS** is the most widely-used fair scheduler
  - Used by default in several Linux distributions
  - Goals
    - Be fair among the multiple processes ready to run
    - Be very efficient on choosing the next process to run

CFS

A very brief overview

- **Virtual runtime (*vruntime*):** counts the CPU time used by a given process
  - Incremented every time a process uses the CPU
- **Selection criteria:** choose the process with the lowest *vruntime*
- *vruntime* may be incremented differently for specific processes
  - **Example:** time increases slower for high priority processes
  - Priority may be chosen by users and system administrators (**nice** level of a process)
- A **red-black tree** structure is used to index processes by their *vruntime* and to efficiently choose the next to run

CFS

A very brief overview

- **Scheduling latency target (sched_latency):** Within the *sched_latency* time interval each process should have a fair chance to use the CPU
  - **Example:** Assuming 4 processes and *sched_latency = 48 ms*, each process is given a time slice of 12 ms
- The time-slice adjusts dynamically when processes arrive/leave the system
  - **Let's go back to the previous example:** if 2 processes finish executing, the time slice for each of the two remaining processes is increased to 24 ms
- A minimum time slice must be set when a large number of processes are waiting to be scheduled (**min_granularity**)
  - **Question:** Do you know why?

Advanced topics

Not covered in this course

- Multiprocessor scheduling
  - Modern computers can have several CPUs…
  - New challenges:
    - Synchronization
    - Cache Affinity
    - …

- Want to know more about these topics?

  - **Chapters 9 and 10** - Remzi H. Arpaci-Dusseau, Andrea C. Arpaci-Dusseau. **Operating Systems: Three Easy Pieces.** Arpaci-Dusseau Books, 2018.

Questions?