University of Pittsburgh

Introduction to Operating Systems
CS 1550

Spring 2023

Sherif Khattab

ksm73@pitt.edu

(Some slides are from **Silberschatz, Galvin and Gagne ©2013)**

Announcements

- Upcoming deadlines
  - Homework 6 is due **this Friday**
  - Quiz 1 and Lab 2 due on Tuesday 2/28 at 11:59 pm
  - Project 2 is due Friday 3/17 at 11:59 pm
- Midterm exam on Thursday 3/2
  - In-person, on paper, closed book
  - Study guide, old exam, and practice Midterm on Canvas
- Lost points because autograder or simple mistake?
  - please reach out to Grader TA over Piazza
- Navigating the Panopto Videos
  - Video contents
  - Search in captions

Previous lecture …

- Sleepy Barbers solution using Condition Variables

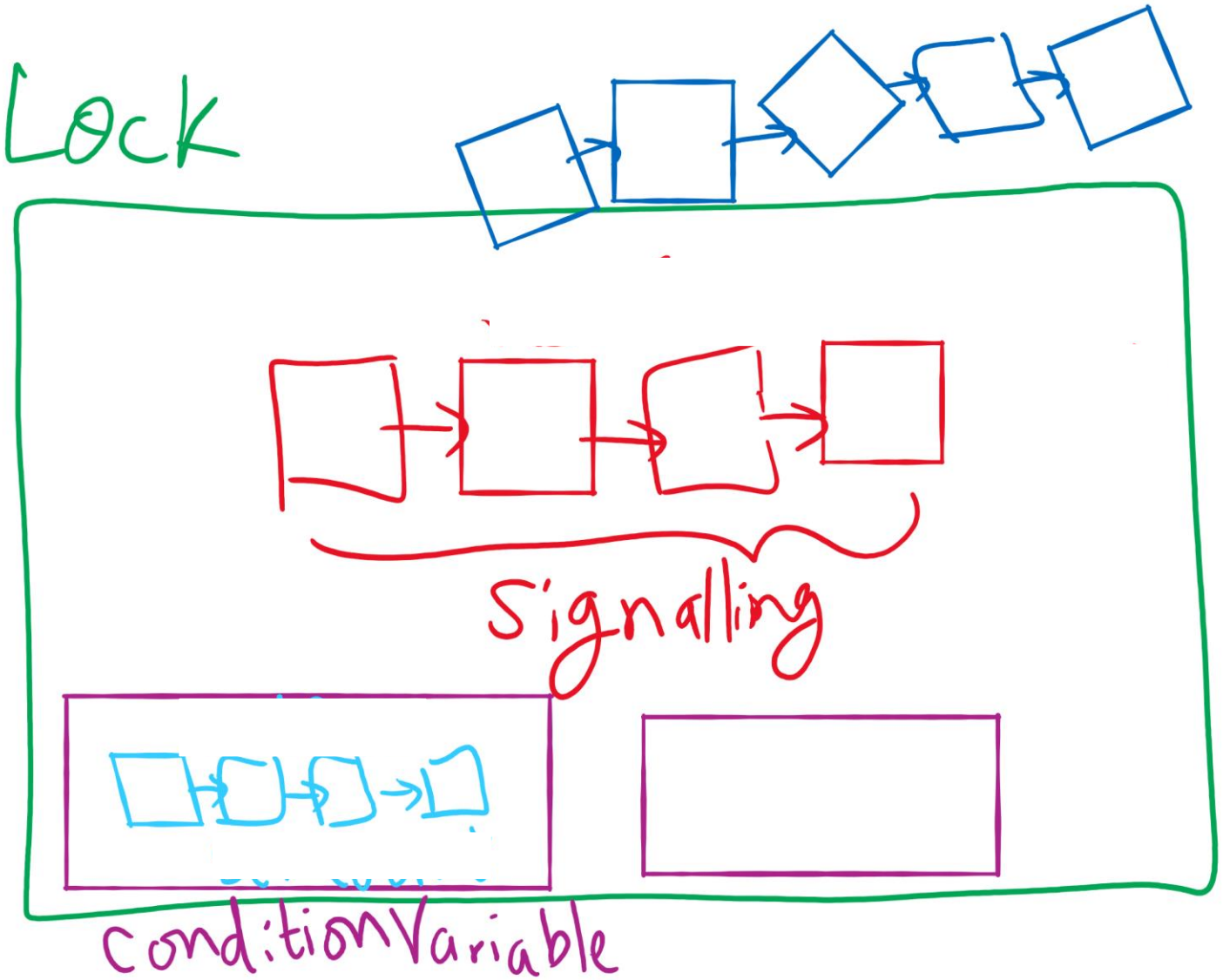
Today …

- How to implement condition variables
- Reflections on using semaphores and condition variables
- CPU Scheduling

User-level implementation of Condition Variables

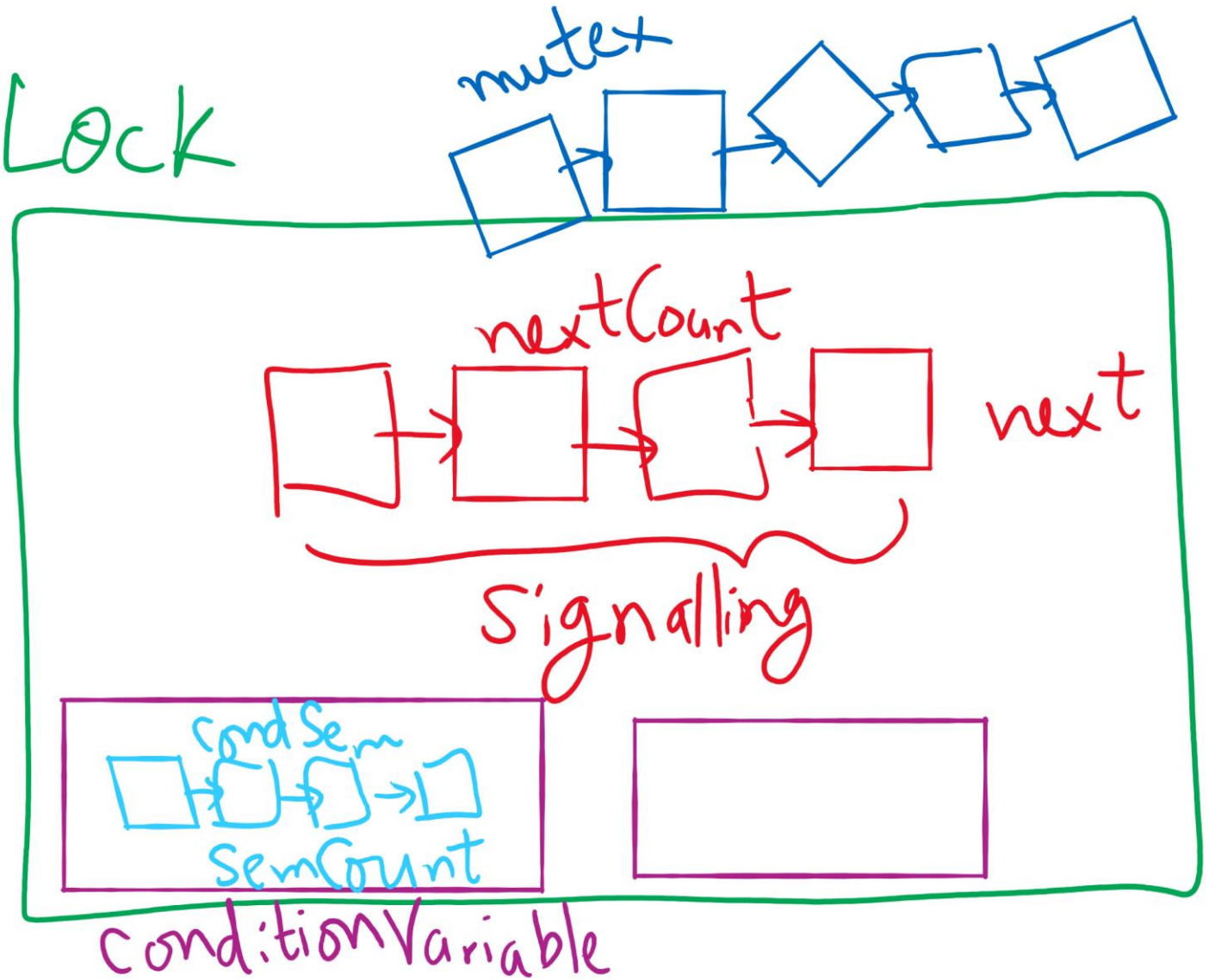
- Why?
  - Some operating systems don't have condition variables
  - Another exercise on solving synchronization problems with semaphores
- What are the waiting situations?
  - waiting to acquire the lock
  - waiting on the condition variable
  - waiting after signaling on a condition variable
    - This is Hoare semantics
      - signaling process waits
    - Compare to Mesa semantics
      - signaled process waits

Lock and Condition Variable Implementation



Lock and Condition Variable Implementation

- Let's have a semaphore for each waiting situation



User-level implementation of Condition Variables

A Lock with two waiting queues

```
struct Lock {
  Semaphore mutex(1);
   Semaphore next(0);
  int nextCount = 0;
}
Acquire(){
  mutex.down();
}
```

```
Release(){
   if(nextCount > 0){
      next.up();
      nextCount--;
  } else mutex.up();
}
```

Condition Variable

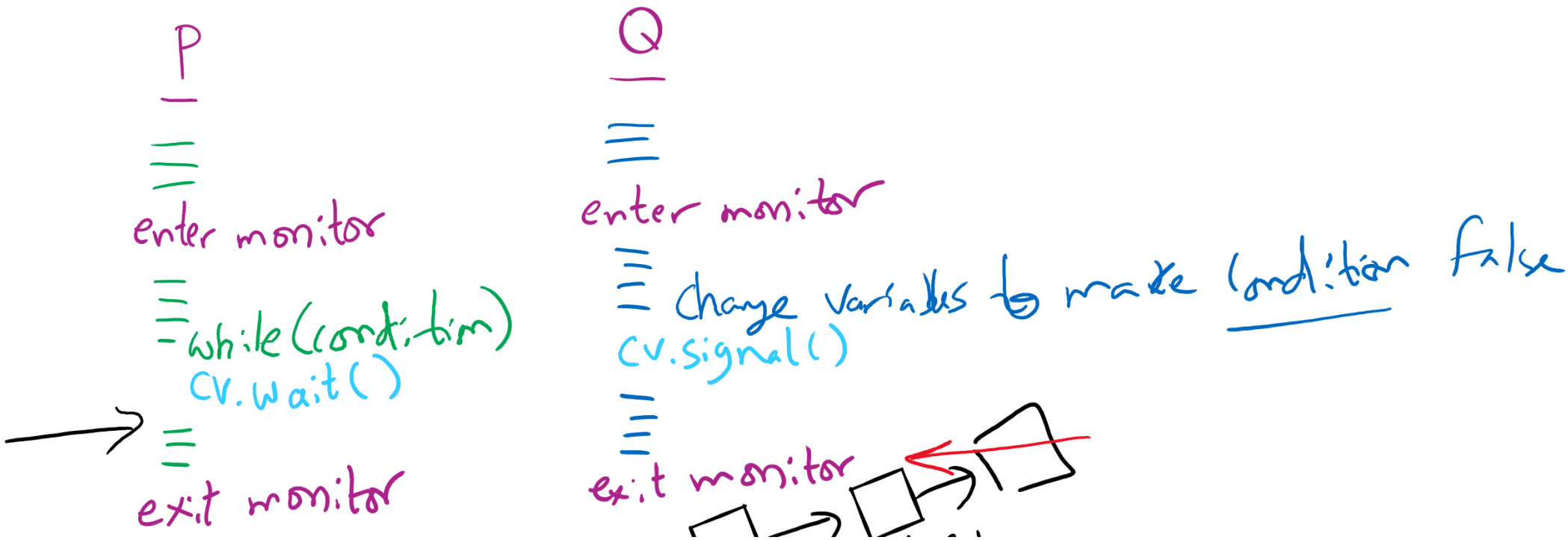
```
struct ConditionVariable {
    Semaphore condSem(0);
    int semCount = 0;
    Lock *lk;
}
```

```
Wait(){
  if(lk->nextCount > 0)
     lk->next.up();
     lk->nextCount--;
  else lk->mutex.up();
  semCount++;
  condSem.down();
  semCount--;
}
```

```
Signal(){
  if(semCount > 0){
    condSem.up()
    lk->nextCount++
    lk->next.down();
    lk->nextCount—
  }
}
```

Let's trace our solution!

- Note: Monitor is another name for Lock



Reflections on semaphore usage

- Semaphores can be used as
  - Resource counters
  - Waiting spaces
  - For mutual exclusion

Reflections on Condition Variables

- Define a class and put all shared variables inside the class
- Include a mutex and a condition variable in the class
- For each public method of the class
  - Start by locking the mutex lock
  - If need to wait, use a while loop and wait on the condition variable
  - Before **broadcasting** on the condition variable, make sure to change the waiting condition

Final Remarks on Process Synchronization

- Many other synchronization mechanisms
  - Message passing
  - Barriers
  - Futex
  - Re-entrant locks
  - AtomicInteger, AtomicX

Problem of the Day: CPU Scheduling

How does the ***short-term scheduler*** select the next process to run?

CPU Scheduling

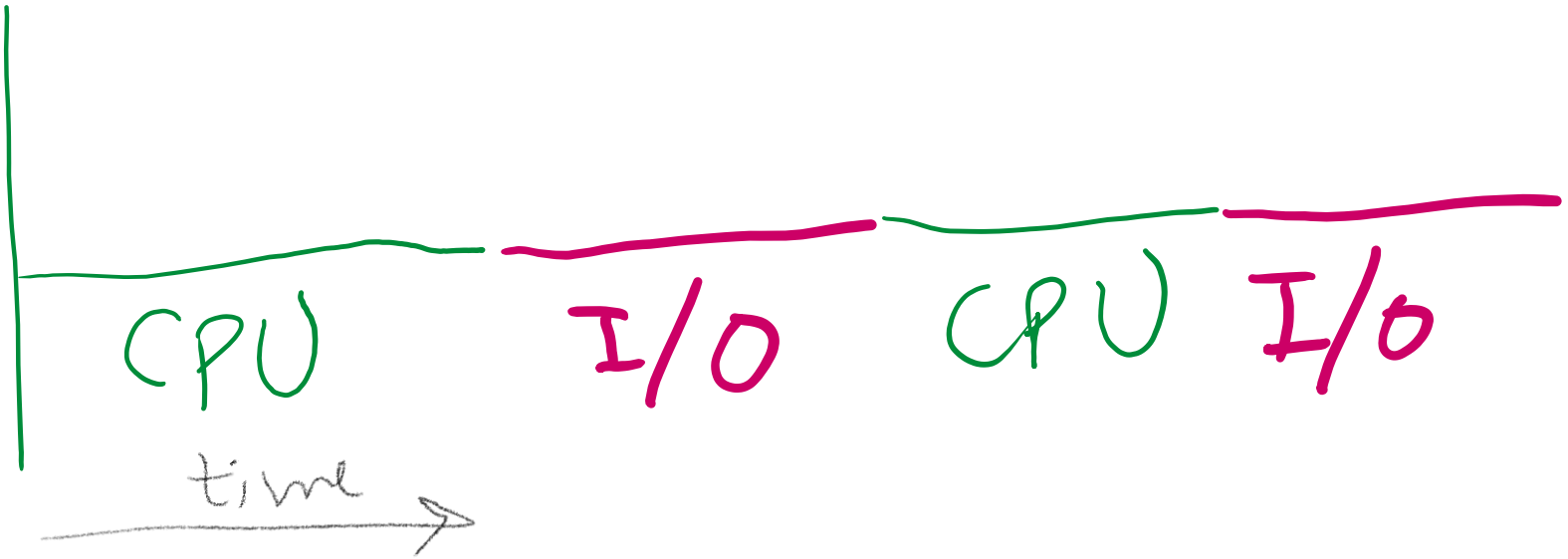
- Scheduling the processor among all ready processes
- User-oriented criteria
  - Response Time: Elapsed time between the submission of a request and the receipt of a response
  - Turnaround Time: Elapsed time between the submission of a process to its completion
- System-oriented criteria
  - Processor utilization
  - Throughput: number of process completed per unit time
  - Fairness

Short-Term Scheduler Dispatcher

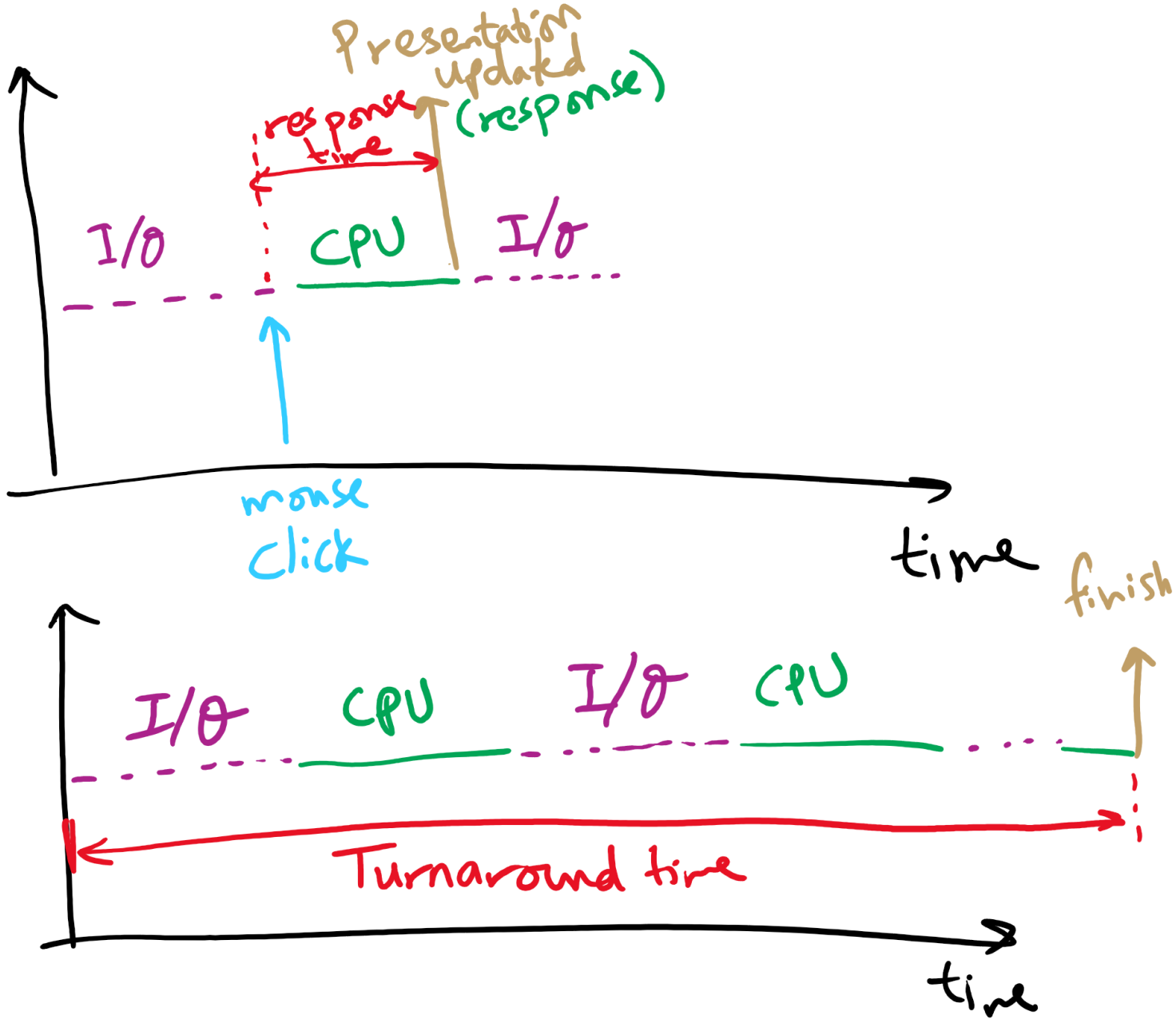
- The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler
- The functions of the dispatcher include:
  - Switching context
  - Switching to user mode
  - Jumping to the location in the user program to restart execution
- The dispatch latency must be minimal

The CPU-I/O Cycle

- Processes require alternate use of processor and I/O in a repetitive fashion
- Each cycle consist of a CPU burst followed by an I/O burst
  - A process terminates on a CPU burst
- CPU-bound processes have longer CPU bursts than I/O-bound processes



Response time vs. Turnaround time



Scheduling Algorithms

- First-Come, First-Served Scheduling
- Shortest-Job-First Scheduling
  - Also referred to as Shortest Process Next
- Priority Scheduling
- Round-Robin Scheduling
- Multilevel Queue Scheduling
- Multilevel Feedback Queue Scheduling

Characterization of Scheduling Policies

- The selection function determines which ready process is selected next for execution
- The decision mode specifies the instants in time the selection function is exercised
  - Nonpreemptive
    - Once a process is in the running state, it will continue until it terminates or blocks for an I/O
  - Preemptive
    - Currently running process may be interrupted and moved to the Ready state by the OS
    - Prevents one process from monopolizing the processor

Process Mix Example

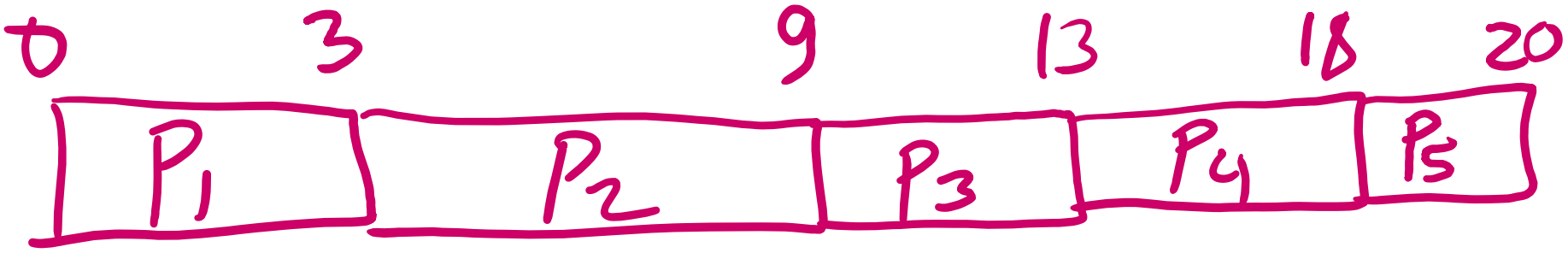
Process	Arrival Time	Service Time
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2

Service time = total processor time needed in one (CPU-I/O) cycle
Jobs with long service time are CPU-bound jobs and are referred to as "long jobs"

First Come First Served (FCFS)

- Selection function: the process that has been waiting the longest in the ready queue (hence, FCFS)
- Decision mode: non-preemptive
  - a process runs until it blocks for an I/O

Process	Arrival Time	Service Time
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2



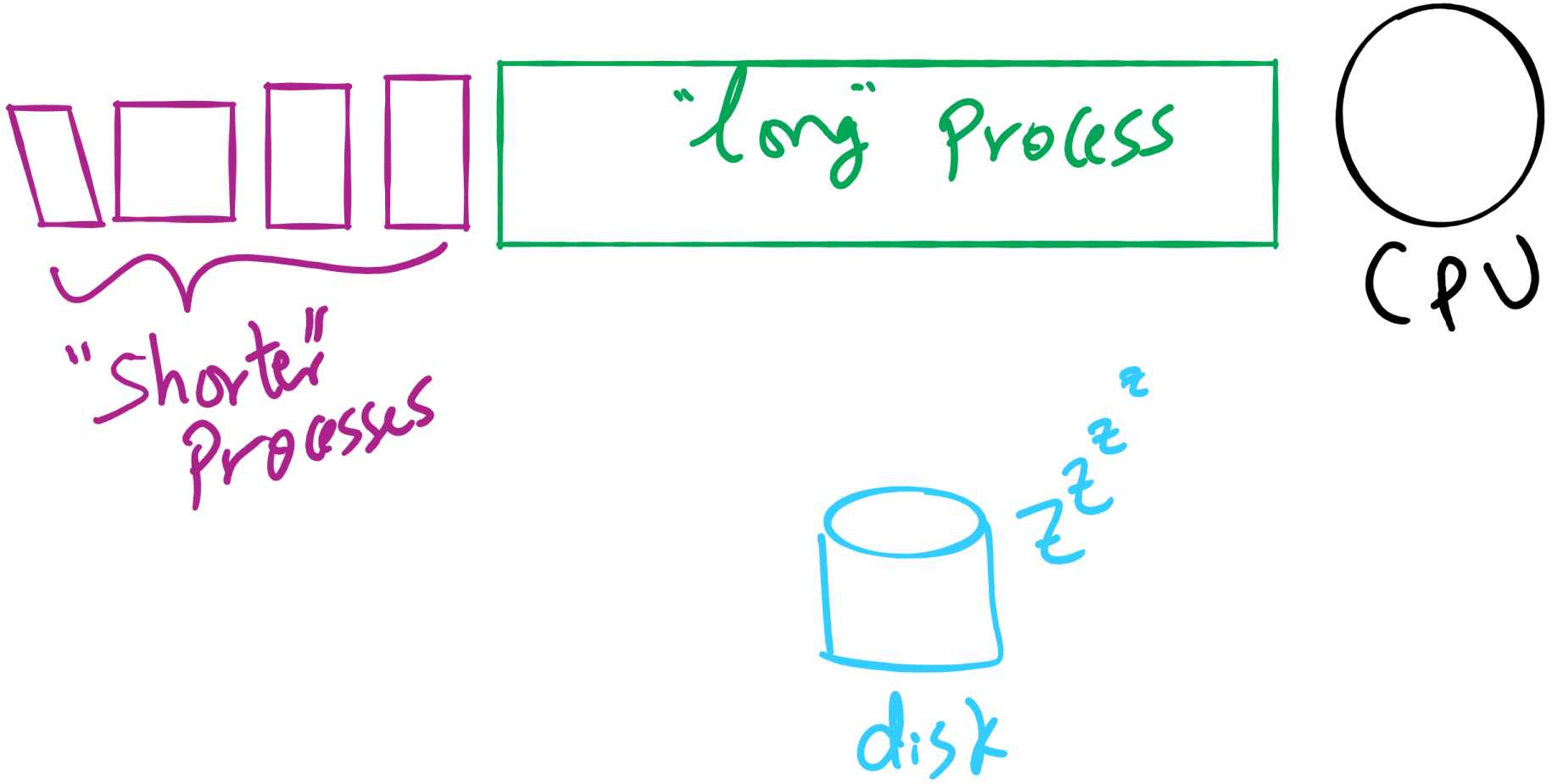
Average Response Time

$$\text{Average Response Time} = \frac{(P_1 - 0) + (P_2 - 2) + (P_3 - 4) + (P_4 - 6) + (P_5 - 8)}{5}$$

FCFS drawbacks

- Favours CPU-bound processes
  - CPU-bound processes monopolize the processor
  - I/O-bound processes have to wait until completion of CPU-bound process
    - I/O-bound processes may have to wait even after their I/Os are completed (poor device utilization)
    - Convoy effect
  - Better I/O device utilization could be achieved if I/O bound processes had higher priority

Convoy Effect



Shortest Job First (Shortest Process Next)

- Selection function: the process with the shortest expected CPU burst time
  - I/O-bound processes will be selected first
- Decision mode: non-preemptive
- The required processing time, i.e., the CPU burst time, must be estimated for each process

Process	Arrival Time	Service Time
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2

SJF / SPN Critique

- Possibility of starvation for longer processes
- Lack of preemption is not suitable in a time sharing environment
- SJF/SPN implicitly incorporates priorities
  - Shortest jobs are given preference
  - CPU bound processes have lower priority, but a process doing no I/O could still monopolize the CPU if it is the first to enter the system

Priorities

- Implemented by having multiple ready queues to represent each level of priority
- Scheduler selects the process of a higher priority over one of lower priority
- Lower-priority may suffer starvation
- To alleviate starvation allow dynamic priorities
  - The priority of a process changes based on its age or execution history

Round-Robin

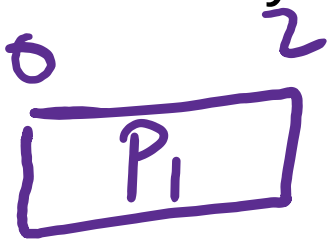
- Selection function: same as FCFS
- Decision mode: pre-emptive
  - a process is allowed to run until the time slice period (quantum, typically from 10 to 100 ms) has expired
  - a clock interrupt occurs and the running process is put on the ready queue

Process	Arrival Time	Service Time
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2

Round-Robin

- Selection function: same as FCFS
- Decision mode: pre-emptive
  - a process is allowed to run until the time slice period (quantum, typically from 10 to 100 ms) has expired
  - a clock interrupt occurs and the running process is put on the ready queue

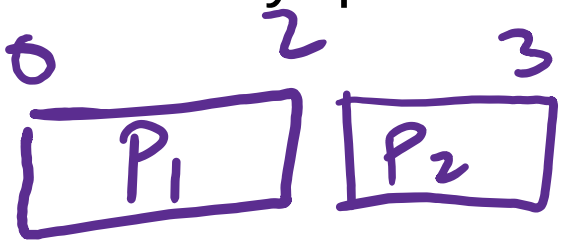
Process	Arrival Time	Service Time
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2



Round-Robin

- Selection function: same as FCFS
- Decision mode: pre-emptive
  - a process is allowed to run until the time slice period (quantum, typically from 10 to 100 ms) has expired
  - a clock interrupt occurs and the running process is put on the ready queue

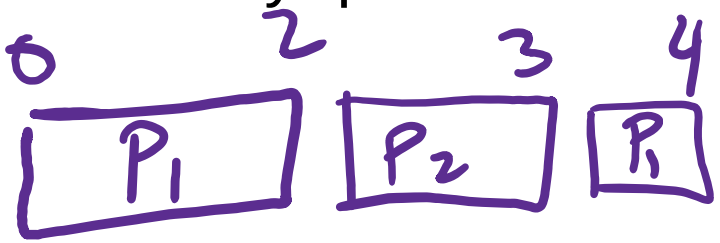
Process	Arrival Time	Service Time
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2



Round-Robin

- Selection function: same as FCFS
- Decision mode: pre-emptive
  - a process is allowed to run until the time slice period (quantum, typically from 10 to 100 ms) has expired
  - a clock interrupt occurs and the running process is put on the ready queue

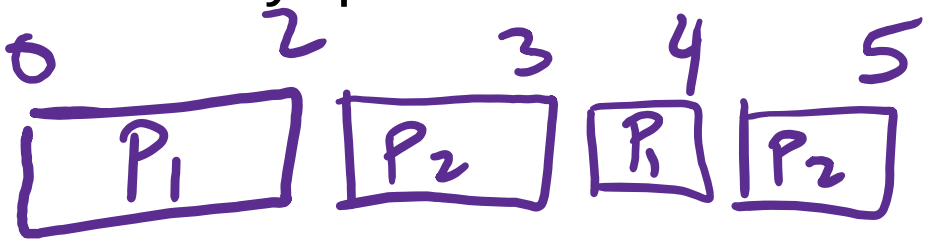
Process	Arrival Time	Service Time
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2



Round-Robin

- Selection function: same as FCFS
- Decision mode: pre-emptive
  - a process is allowed to run until the time slice period (quantum, typically from 10 to 100 ms) has expired
  - a clock interrupt occurs and the running process is put on the ready queue

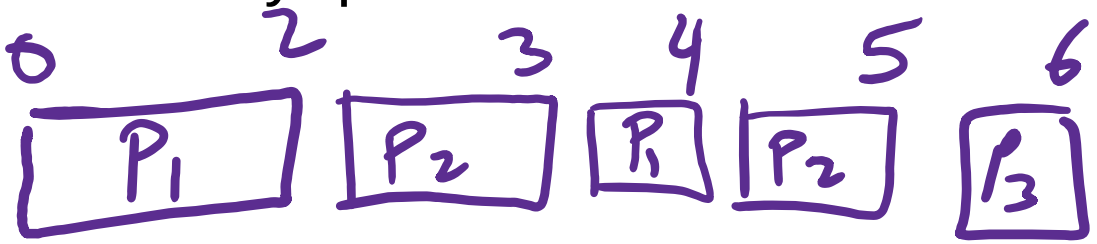
Process	Arrival Time	Service Time
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2



Round-Robin

- Selection function: same as FCFS
- Decision mode: pre-emptive
  - a process is allowed to run until the time slice period (quantum, typically from 10 to 100 ms) has expired
  - a clock interrupt occurs and the running process is put on the ready queue

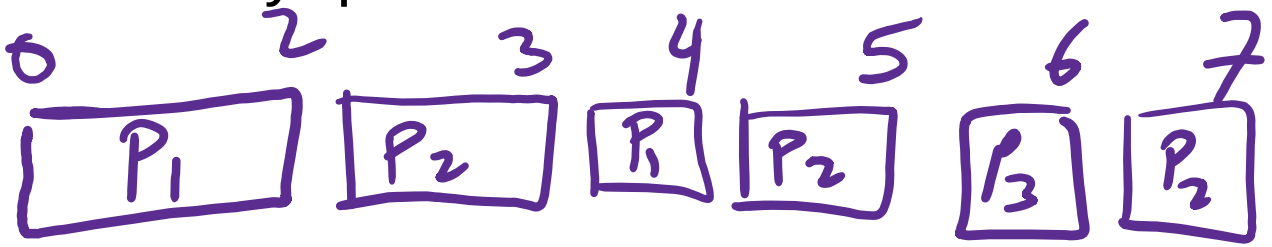
Process	Arrival Time	Service Time
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2



Round-Robin

- Selection function: same as FCFS
- Decision mode: pre-emptive
  - a process is allowed to run until the time slice period (quantum, typically from 10 to 100 ms) has expired
  - a clock interrupt occurs and the running process is put on the ready queue

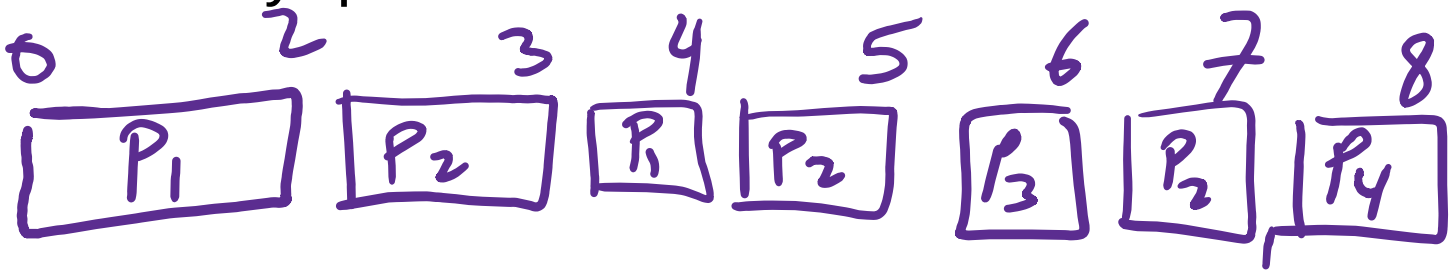
Process	Arrival Time	Service Time
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2



Round-Robin

- Selection function: same as FCFS
- Decision mode: pre-emptive
  - a process is allowed to run until the time slice period (quantum, typically from 10 to 100 ms) has expired
  - a clock interrupt occurs and the running process is put on the ready queue

Process	Arrival Time	Service Time
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2



Round-Robin

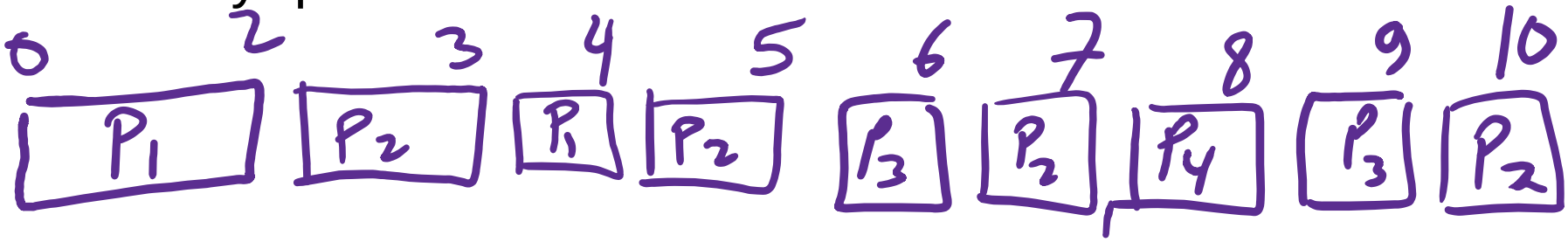
- Selection function: same as FCFS
- Decision mode: pre-emptive
  - a process is allowed to run until the time slice period (quantum, typically from 10 to 100 ms) has expired
  - a clock interrupt occurs and the running process is put on the ready queue

Process	Arrival Time	Service Time
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2

Round-Robin

- Selection function: same as FCFS
- Decision mode: pre-emptive
  - a process is allowed to run until the time slice period (quantum, typically from 10 to 100 ms) has expired
  - a clock interrupt occurs and the running process is put on the ready queue

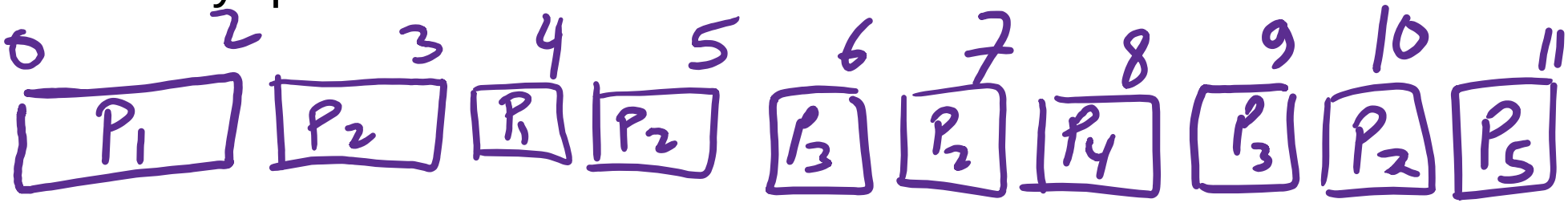
Process	Arrival Time	Service Time
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2



Round-Robin

- Selection function: same as FCFS
- Decision mode: pre-emptive
  - a process is allowed to run until the time slice period (quantum, typically from 10 to 100 ms) has expired
  - a clock interrupt occurs and the running process is put on the ready queue

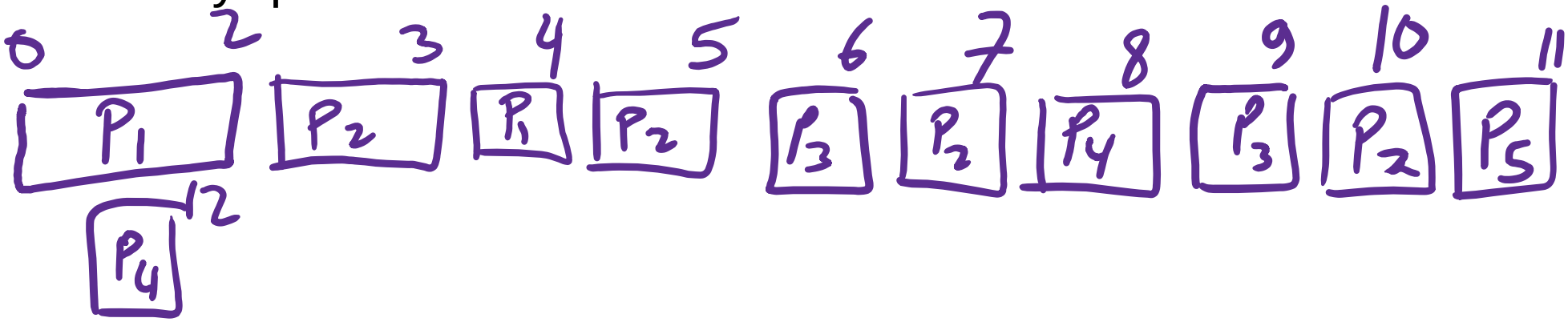
Process	Arrival Time	Service Time
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2



Round-Robin

- Selection function: same as FCFS
- Decision mode: pre-emptive
  - a process is allowed to run until the time slice period (quantum, typically from 10 to 100 ms) has expired
  - a clock interrupt occurs and the running process is put on the ready queue

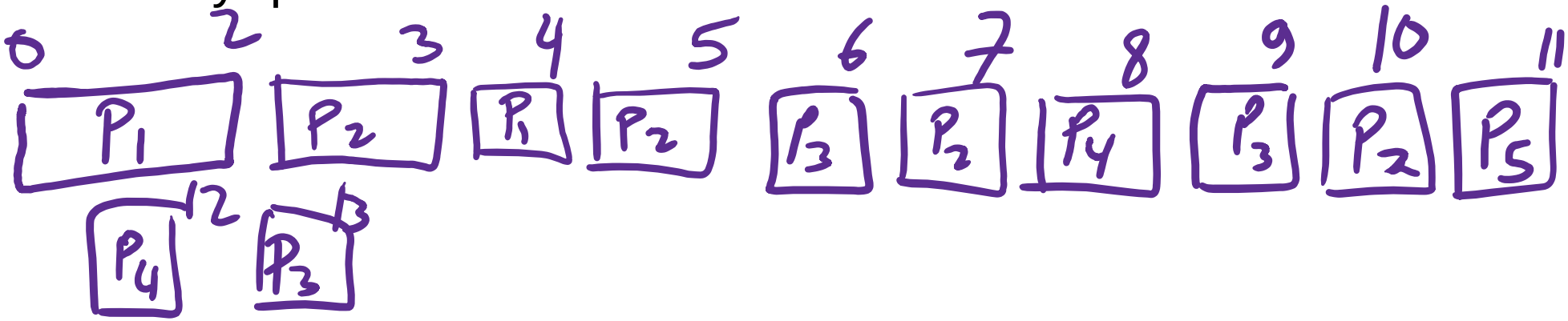
Process	Arrival Time	Service Time
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2



Round-Robin

- Selection function: same as FCFS
- Decision mode: pre-emptive
  - a process is allowed to run until the time slice period (quantum, typically from 10 to 100 ms) has expired
  - a clock interrupt occurs and the running process is put on the ready queue

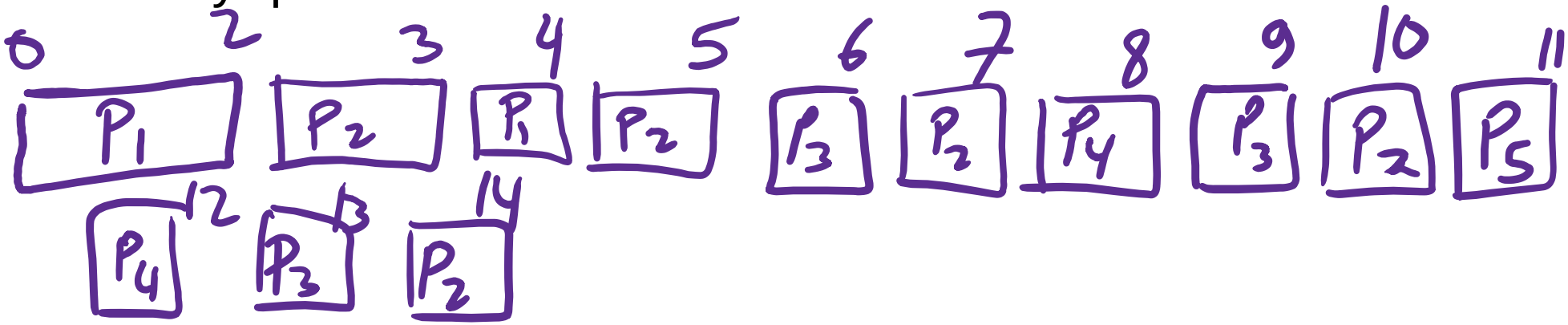
Process	Arrival Time	Service Time
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2



Round-Robin

- Selection function: same as FCFS
- Decision mode: pre-emptive
  - a process is allowed to run until the time slice period (quantum, typically from 10 to 100 ms) has expired
  - a clock interrupt occurs and the running process is put on the ready queue

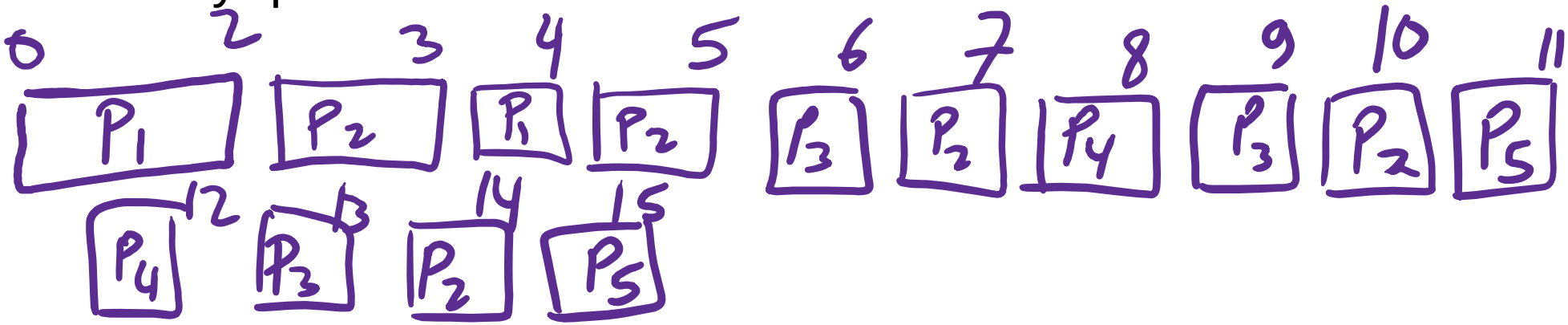
Process	Arrival Time	Service Time
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2



Round-Robin

- Selection function: same as FCFS
- Decision mode: pre-emptive
  - a process is allowed to run until the time slice period (quantum, typically from 10 to 100 ms) has expired
  - a clock interrupt occurs and the running process is put on the ready queue

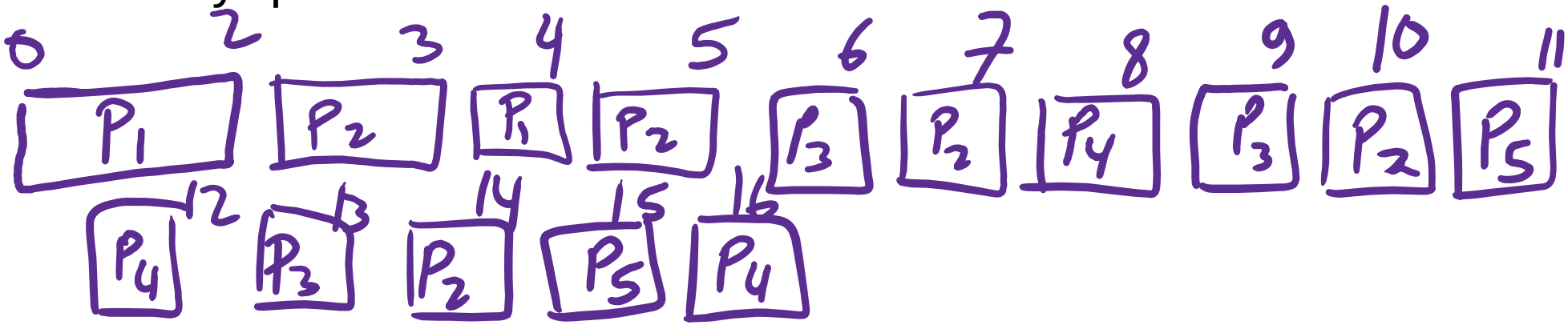
Process	Arrival Time	Service Time
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2



Round-Robin

- Selection function: same as FCFS
- Decision mode: pre-emptive
  - a process is allowed to run until the time slice period (quantum, typically from 10 to 100 ms) has expired
  - a clock interrupt occurs and the running process is put on the ready queue

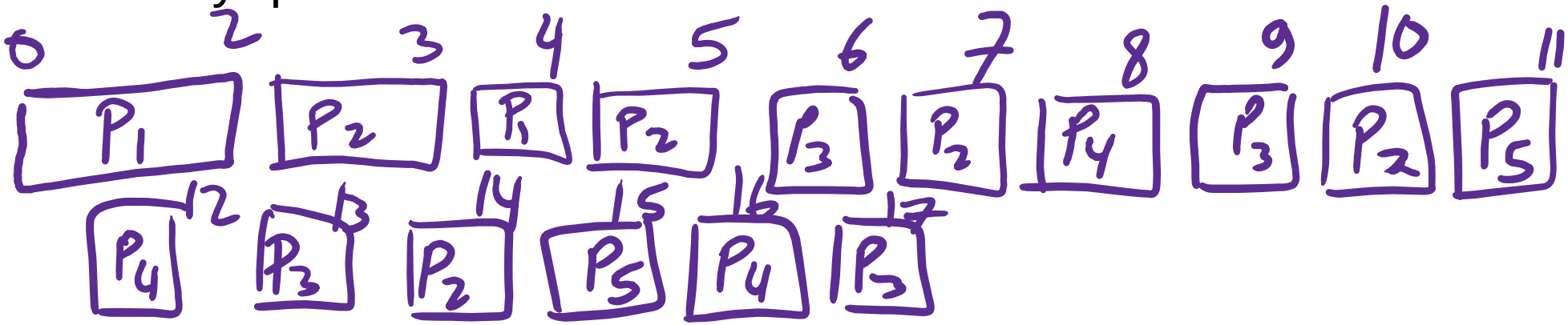
Process	Arrival Time	Service Time
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2



Round-Robin

- Selection function: same as FCFS
- Decision mode: pre-emptive
  - a process is allowed to run until the time slice period (quantum, typically from 10 to 100 ms) has expired
  - a clock interrupt occurs and the running process is put on the ready queue

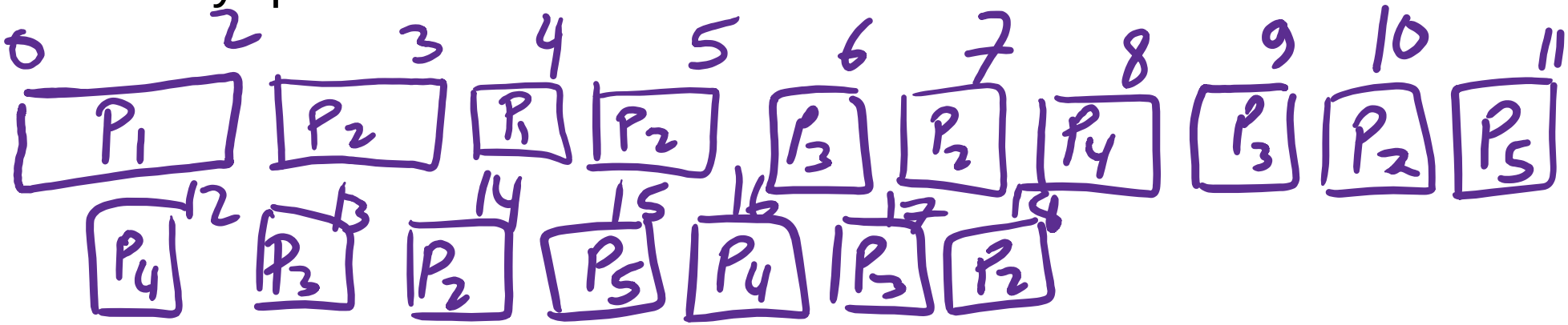
Process	Arrival Time	Service Time
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2



Round-Robin

- Selection function: same as FCFS
- Decision mode: pre-emptive
  - a process is allowed to run until the time slice period (quantum, typically from 10 to 100 ms) has expired
  - a clock interrupt occurs and the running process is put on the ready queue

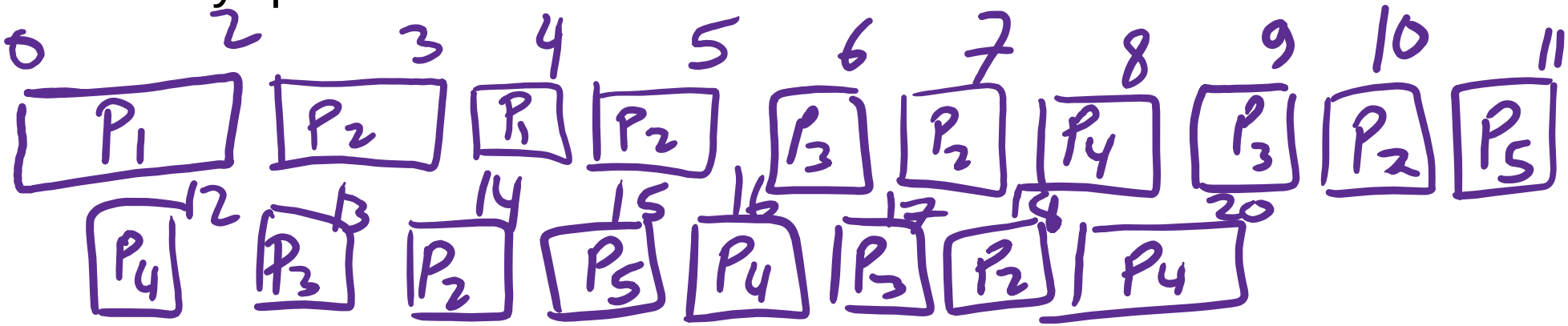
Process	Arrival Time	Service Time
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2



Round-Robin

- Selection function: same as FCFS
- Decision mode: pre-emptive
  - a process is allowed to run until the time slice period (quantum, typically from 10 to 100 ms) has expired
  - a clock interrupt occurs and the running process is put on the ready queue

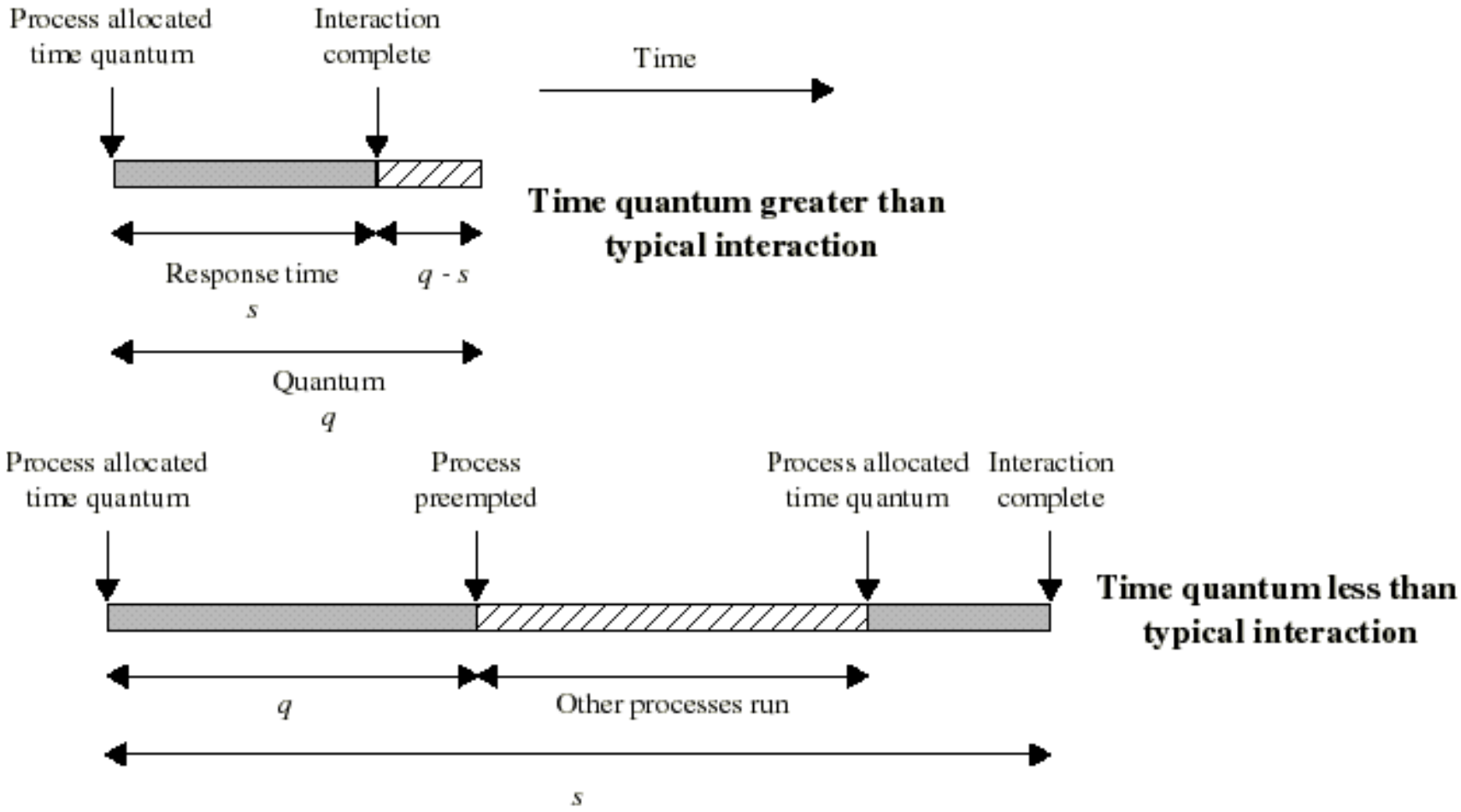
Process	Arrival Time	Service Time
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2



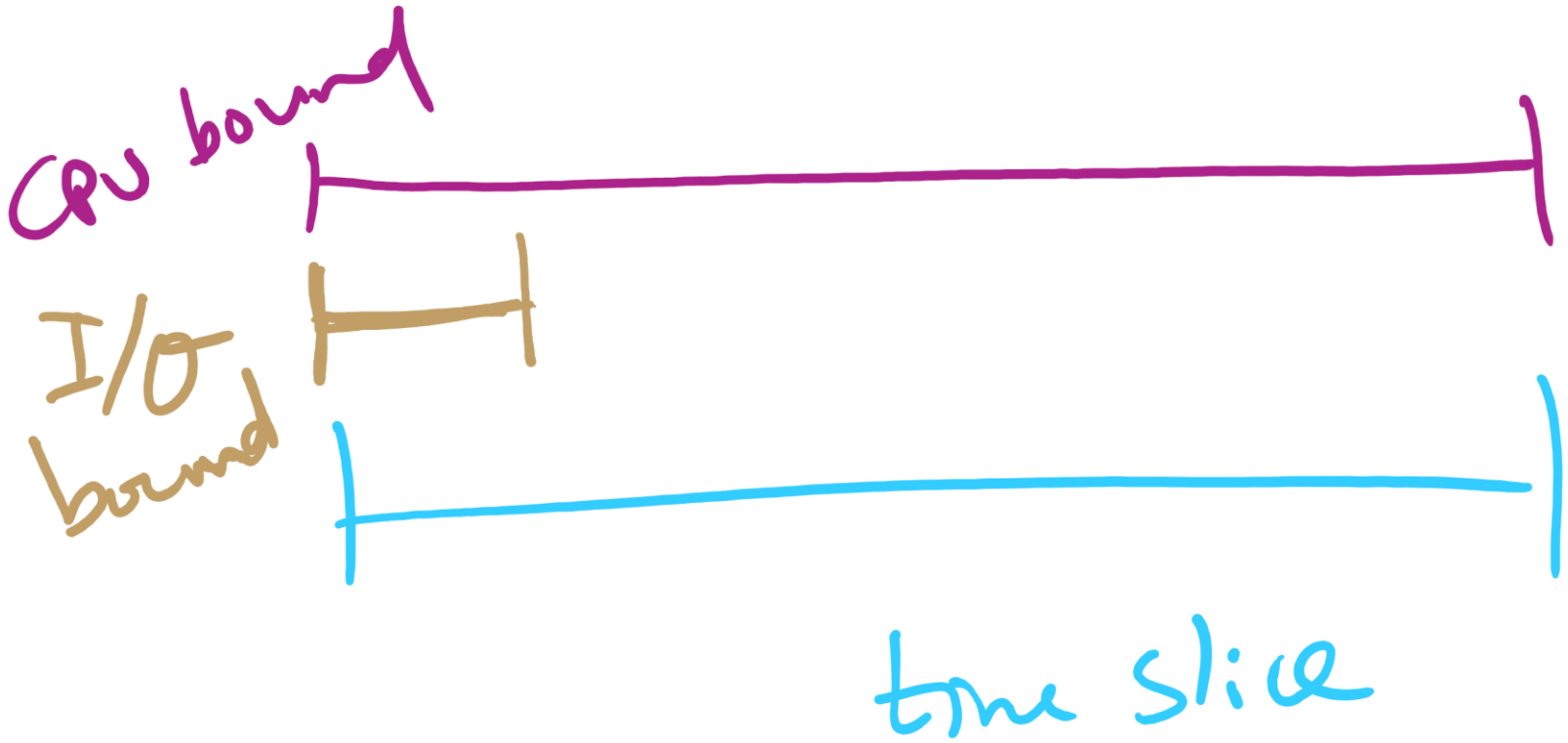
RR Time Quantum

- Quantum must be substantially larger than the time required to handle the clock interrupt and dispatching
- Quantum should be larger then the typical interaction
  - but not much larger, to avoid penalizing I/O bound processes

RR Time Quantum



Quantum Length



Round Robin: critique

- Still favors CPU-bound processes
  - An I/O bound process uses the CPU for a time less than the time quantum before it is blocked waiting for an I/O
  - A CPU-bound process runs for all its time slice and is put back into the ready queue
    - May unfairly get in front of blocked processes

Multilevel Feedback Scheduling

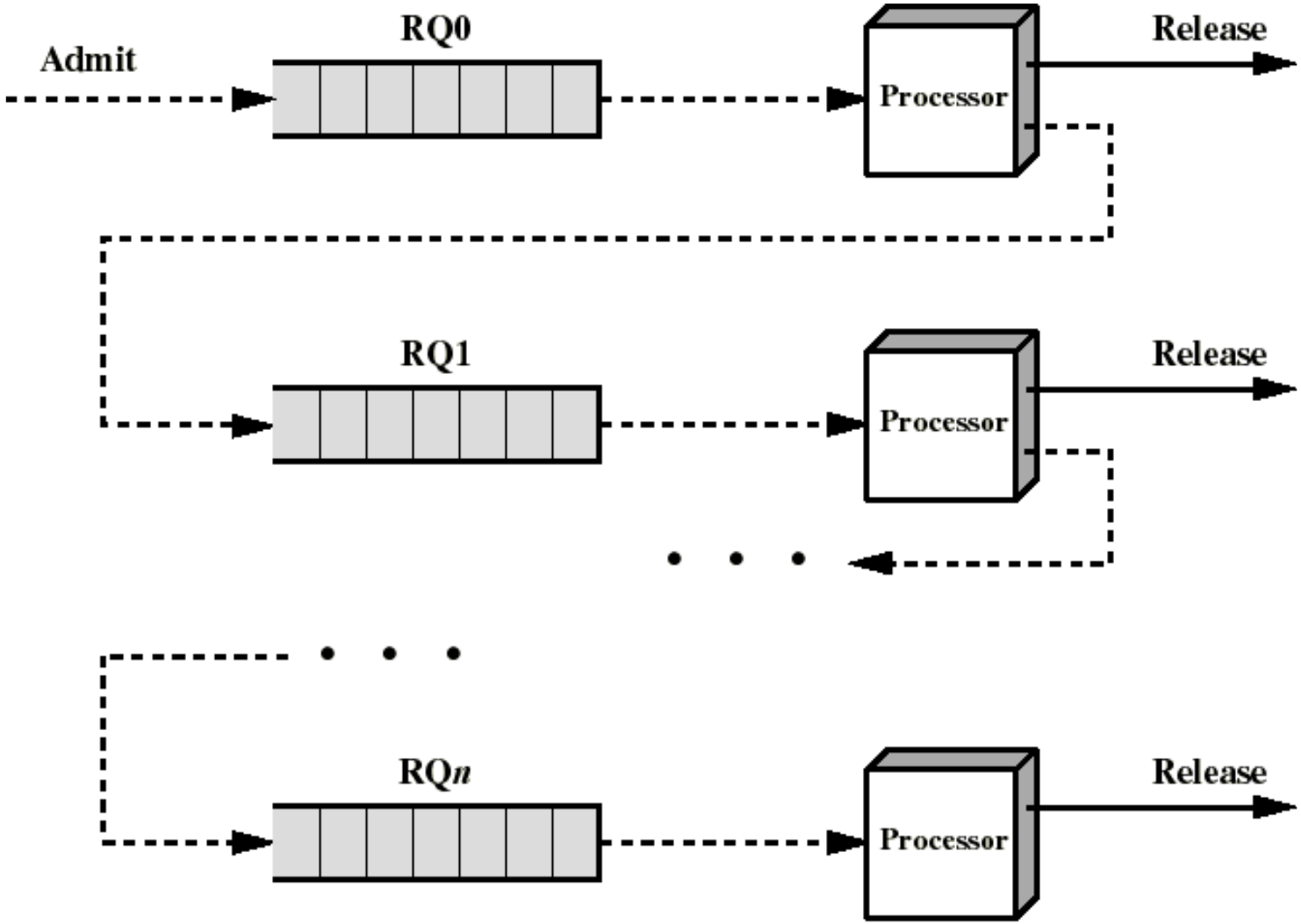
- Preemptive scheduling with dynamic priorities
- N ready to execute queues with decreasing priorities:
  - $P(RQ_0) > P(RQ_1) > ... > P(RQ_{N-1})$
- Dispatcher selects a process for execution from RQ_i only if RQ_{i-1} to RQ_0 are empty

Multilevel Feedback Scheduling

- New process are placed in RQ_0
- After the first quantum, they are moved to RQ_1, and to RQ_2 after the second quantum, … and to RQ_{N-1} after the Nth quantum
- I/O-bound processes remain in higher priority queues.
  - CPU-bound jobs drift downward.
  - Hence, long jobs may starve

Multiple Feedback Queues

Different RQs may have different quantum values



Time Quantum for feedback Scheduling

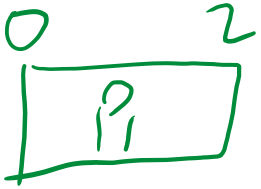
- With a fixed quantum time, the turn-around time of longer processes can be high
- To alleviate this problem, the time quantum can be increased based on the depth of the queue
  - Time quantum of $RQ_i = 2^{i-1}$
- May still cause longer processes to suffer starvation.
  - Possible fix is to promote a process to higher queue after some time

Process	Arrival Time	Service Time
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2

Time Quantum for feedback Scheduling

- With a fixed quantum time, the turn-around time of longer processes can be high
- To alleviate this problem, the time quantum can be increased based on the depth of the queue
  - Time quantum of $RQ_i = 2^{i-1}$
- May still cause longer processes to suffer starvation.
  - Possible fix is to promote a process to higher queue after some time

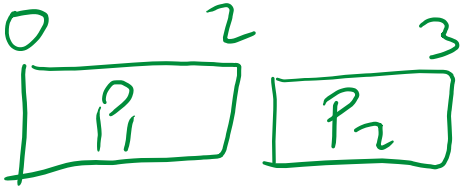
Process	Arrival Time	Service Time
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2



Time Quantum for feedback Scheduling

- With a fixed quantum time, the turn-around time of longer processes can be high
- To alleviate this problem, the time quantum can be increased based on the depth of the queue
  - Time quantum of $RQ_i = 2^{i-1}$
- May still cause longer processes to suffer starvation.
  - Possible fix is to promote a process to higher queue after some time

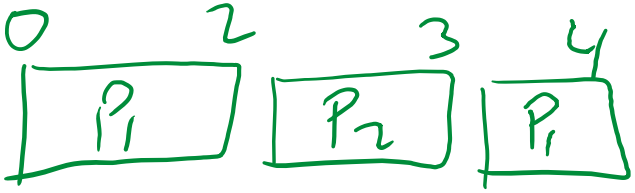
Process	Arrival Time	Service Time
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2



Time Quantum for feedback Scheduling

- With a fixed quantum time, the turn-around time of longer processes can be high
- To alleviate this problem, the time quantum can be increased based on the depth of the queue
  - Time quantum of $RQ_i = 2^{i-1}$
- May still cause longer processes to suffer starvation.
  - Possible fix is to promote a process to higher queue after some time

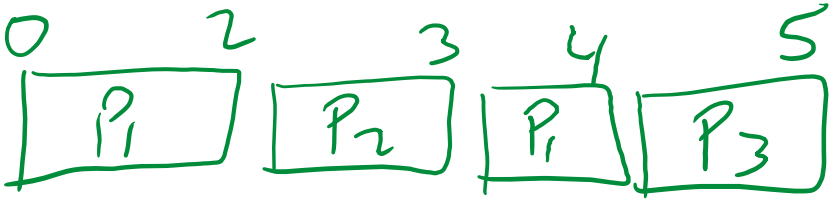
Process	Arrival Time	Service Time
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2



Time Quantum for feedback Scheduling

- With a fixed quantum time, the turn-around time of longer processes can be high
- To alleviate this problem, the time quantum can be increased based on the depth of the queue
  - Time quantum of $RQ_i = 2^{i-1}$
- May still cause longer processes to suffer starvation.
  - Possible fix is to promote a process to higher queue after some time

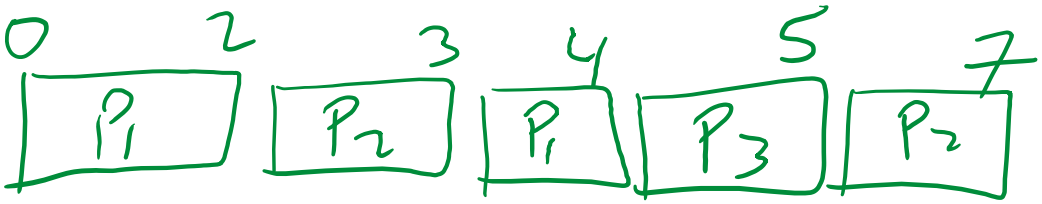
Process	Arrival Time	Service Time
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2



Time Quantum for feedback Scheduling

- With a fixed quantum time, the turn-around time of longer processes can be high
- To alleviate this problem, the time quantum can be increased based on the depth of the queue
  - Time quantum of $RQ_i = 2^{i-1}$
- May still cause longer processes to suffer starvation.
  - Possible fix is to promote a process to higher queue after some time

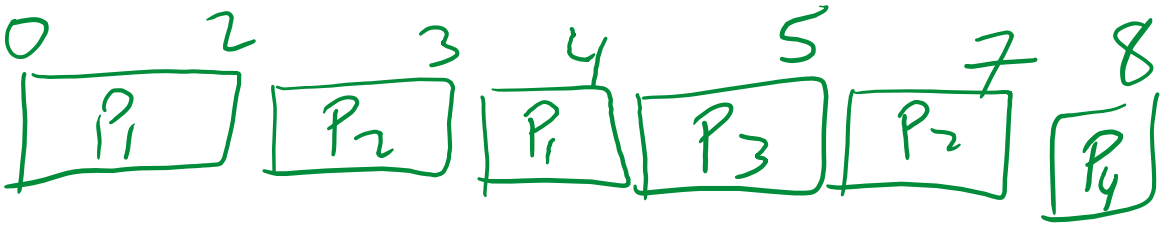
Process	Arrival Time	Service Time
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2



Time Quantum for feedback Scheduling

- With a fixed quantum time, the turn-around time of longer processes can be high
- To alleviate this problem, the time quantum can be increased based on the depth of the queue
  - Time quantum of $RQ_i = 2^{i-1}$
- May still cause longer processes to suffer starvation.
  - Possible fix is to promote a process to higher queue after some time

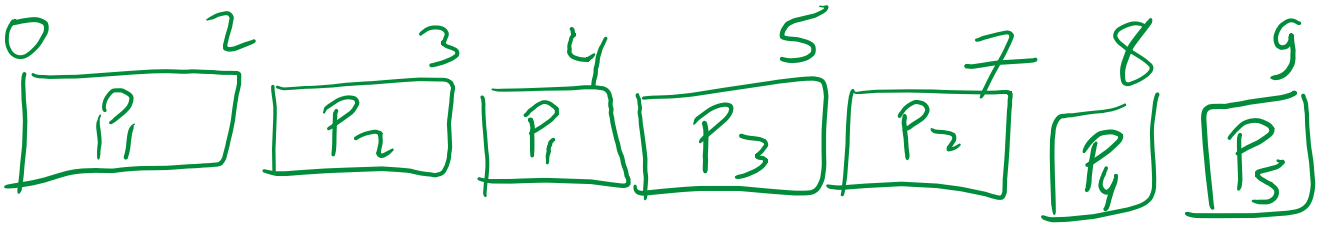
Process	Arrival Time	Service Time
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2



Time Quantum for feedback Scheduling

- With a fixed quantum time, the turn-around time of longer processes can be high
- To alleviate this problem, the time quantum can be increased based on the depth of the queue
  - Time quantum of $RQ_i = 2^{i-1}$
- May still cause longer processes to suffer starvation.
  - Possible fix is to promote a process to higher queue after some time

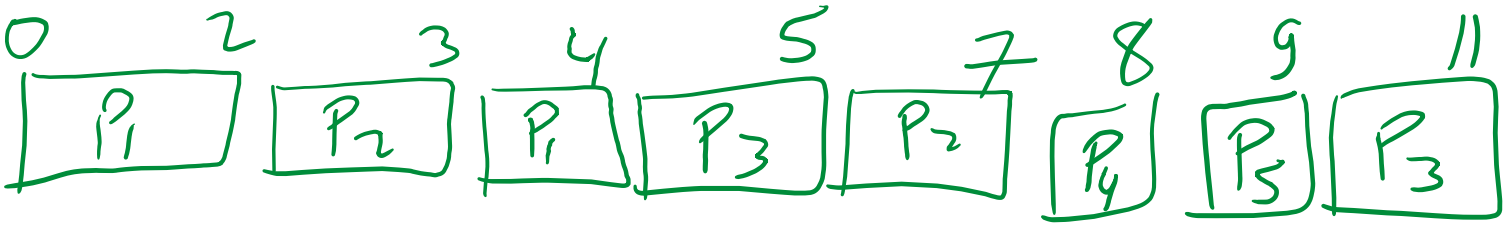
Process	Arrival Time	Service Time
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2



Time Quantum for feedback Scheduling

- With a fixed quantum time, the turn-around time of longer processes can be high
- To alleviate this problem, the time quantum can be increased based on the depth of the queue
  - Time quantum of $RQ_i = 2^{i-1}$
- May still cause longer processes to suffer starvation.
  - Possible fix is to promote a process to higher queue after some time

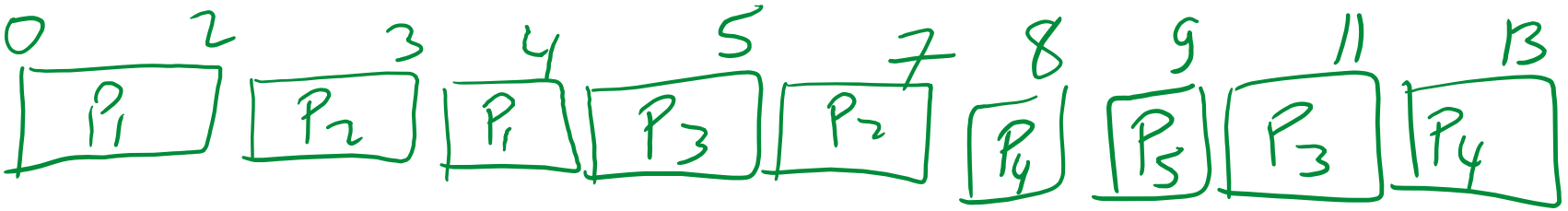
Process	Arrival Time	Service Time
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2



Time Quantum for feedback Scheduling

- With a fixed quantum time, the turn-around time of longer processes can be high
- To alleviate this problem, the time quantum can be increased based on the depth of the queue
  - Time quantum of $RQ_i = 2^{i-1}$
- May still cause longer processes to suffer starvation.
  - Possible fix is to promote a process to higher queue after some time

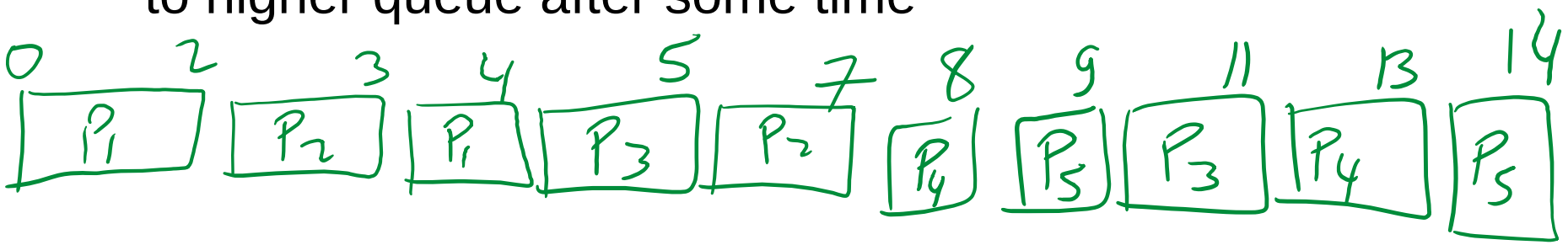
Process	Arrival Time	Service Time
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2



Time Quantum for feedback Scheduling

- With a fixed quantum time, the turn-around time of longer processes can be high
- To alleviate this problem, the time quantum can be increased based on the depth of the queue
  - Time quantum of $RQ_i = 2^{i-1}$
- May still cause longer processes to suffer starvation.
  - Possible fix is to promote a process to higher queue after some time

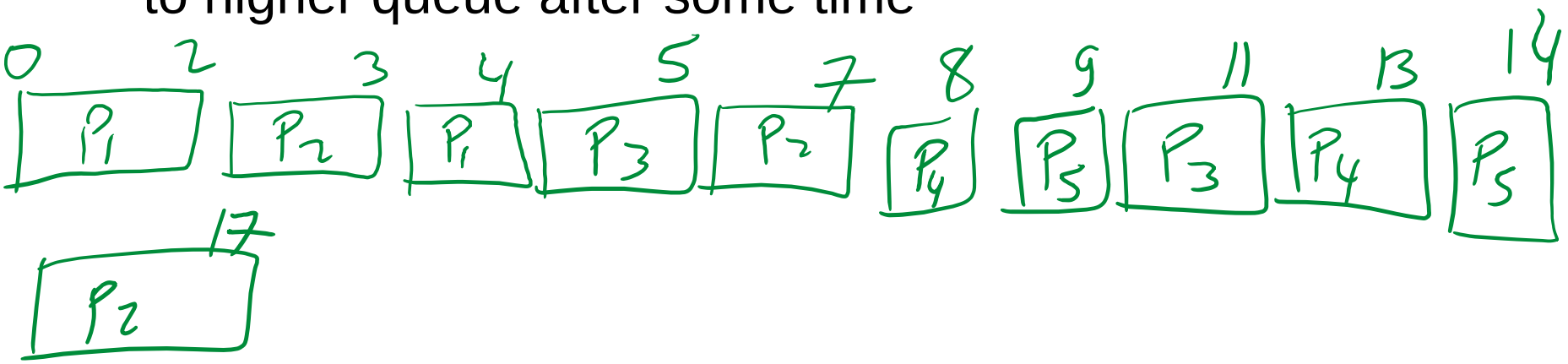
Process	Arrival Time	Service Time
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2



Time Quantum for feedback Scheduling

- With a fixed quantum time, the turn-around time of longer processes can be high
- To alleviate this problem, the time quantum can be increased based on the depth of the queue
  - Time quantum of $RQ_i = 2^{i-1}$
- May still cause longer processes to suffer starvation.
  - Possible fix is to promote a process to higher queue after some time

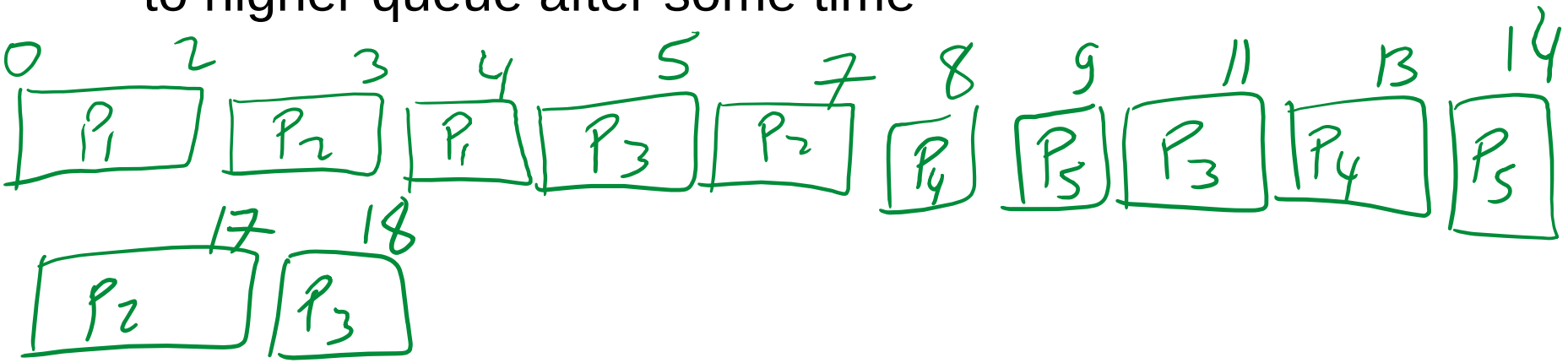
Process	Arrival Time	Service Time
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2



Time Quantum for feedback Scheduling

- With a fixed quantum time, the turn-around time of longer processes can be high
- To alleviate this problem, the time quantum can be increased based on the depth of the queue
  - Time quantum of $RQ_i = 2^{i-1}$
- May still cause longer processes to suffer starvation.
  - Possible fix is to promote a process to higher queue after some time

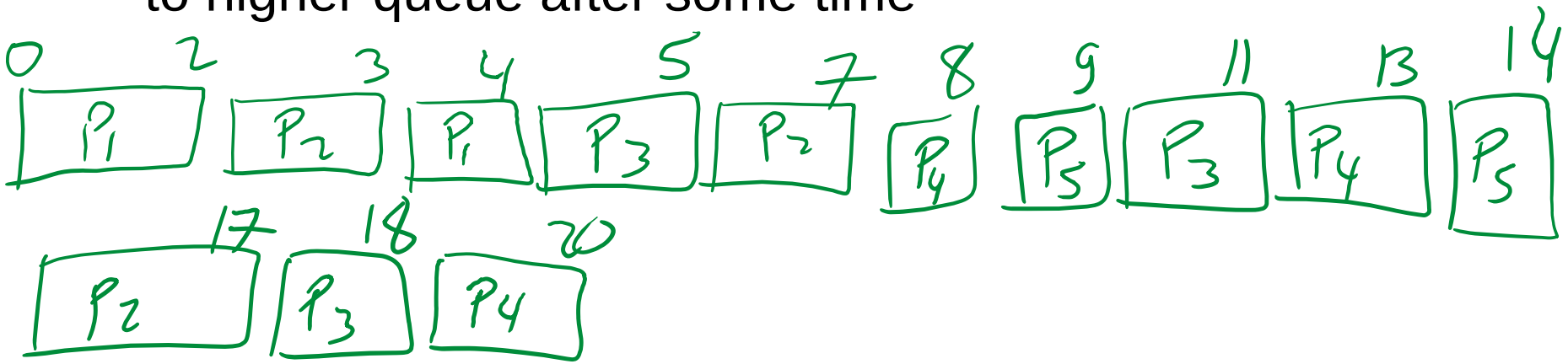
Process	Arrival Time	Service Time
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2



Time Quantum for feedback Scheduling

- With a fixed quantum time, the turn-around time of longer processes can be high
- To alleviate this problem, the time quantum can be increased based on the depth of the queue
  - Time quantum of $RQ_i = 2^{i-1}$
- May still cause longer processes to suffer starvation.
  - Possible fix is to promote a process to higher queue after some time

Process	Arrival Time	Service Time
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2



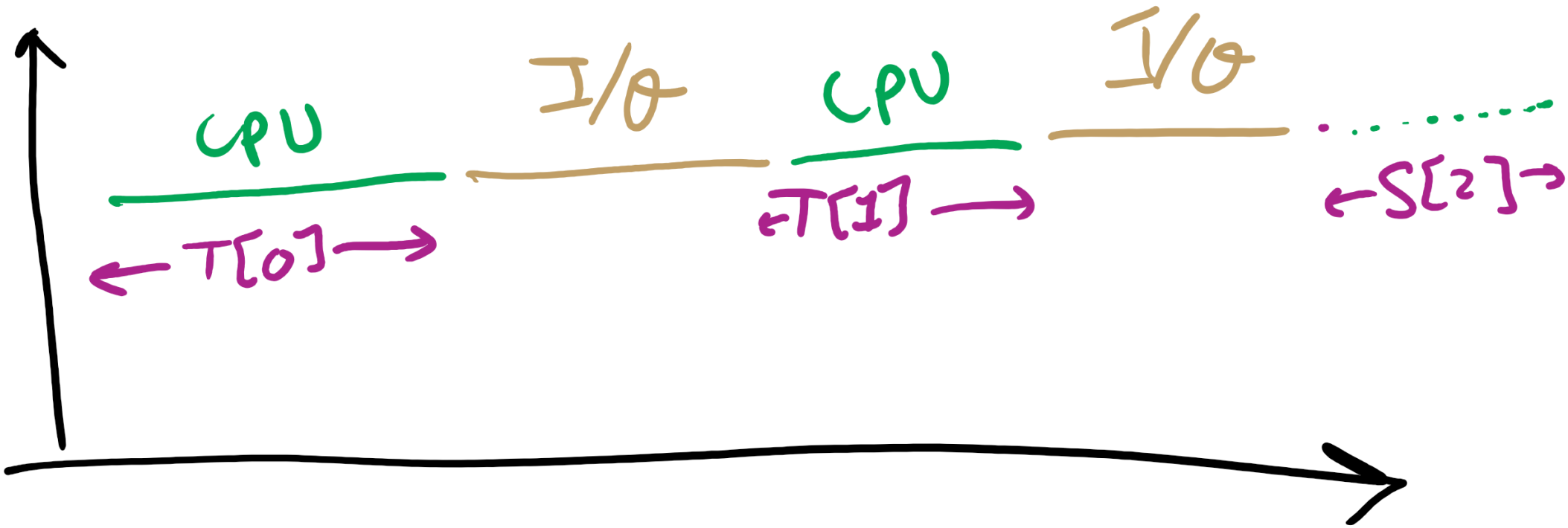
Algorithm Comparison

- Which one is the best?
- The answer depends on many factors:
  - the system workload (extremely variable)
  - hardware support for the dispatcher
  - relative importance of performance criteria (response time, CPU utilization, throughput...)
  - The evaluation method used (each has its limitations...)

Back to SJF: CPU Burst Estimation

- Let T[i] be the execution time for the ith instance of this process: the actual duration of the ith CPU burst of this process
- Let S[i] be the predicted value for the ith CPU burst of this process. The simplest choice is:
  - $S[n+1] = (1/n)(T[1]+\ldots+T[n]) = (1/n) \sum_{i=1 \text{ to } n} T[i]$
- This can be more efficiently calculated as:
  - $S[n+1] = (1/n)\ T[n] + ((n-1)/n)\ S[n]$
- This estimate, however, results in equal weight for each instance

CPU Burst Estimation



Estimating the required CPU burst

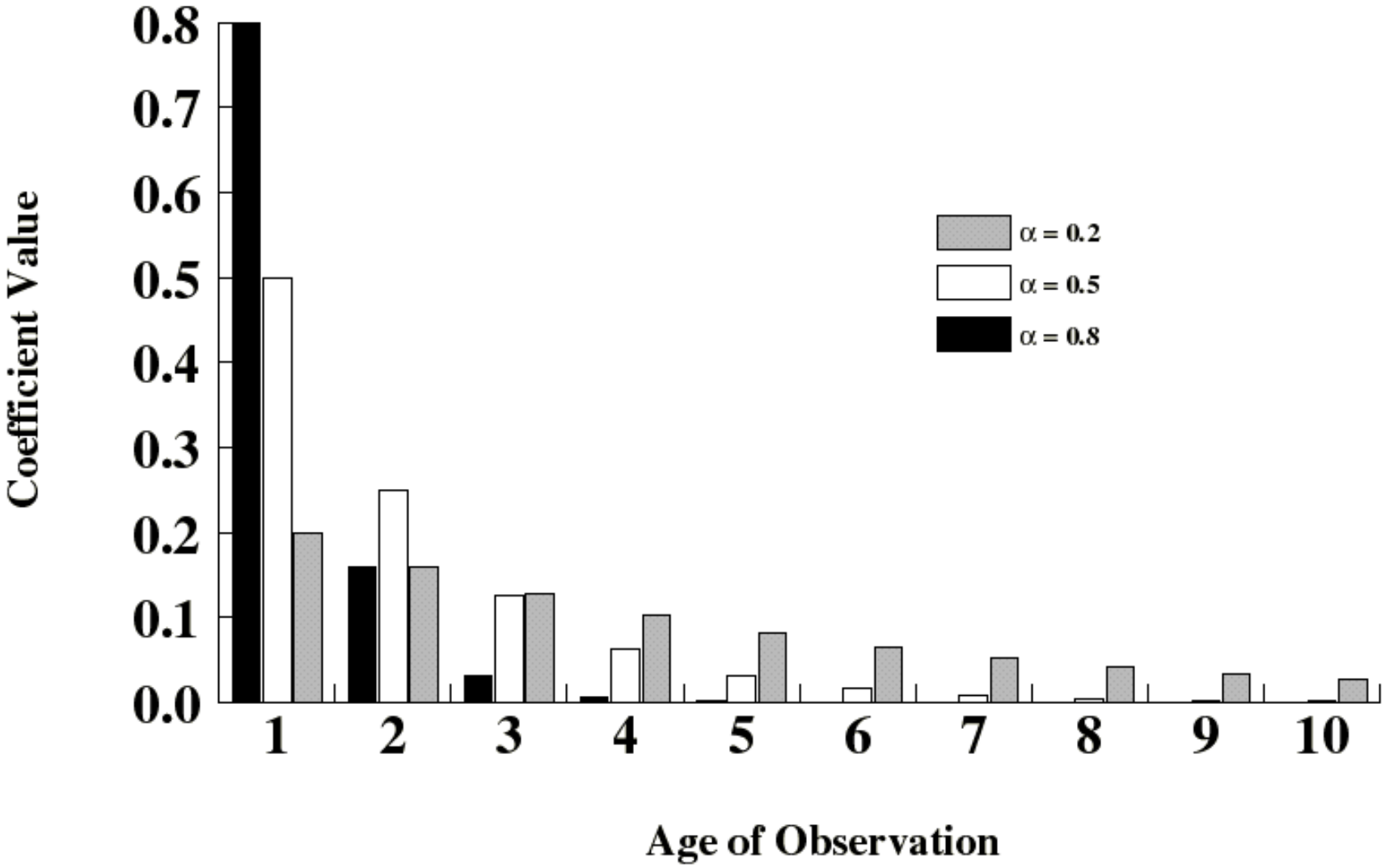
- Recent instances are more likely to better reflect future behavior
- A common technique to factor the above observation into the estimate is to use exponential averaging :

  - $S[n+1] = \alpha\, T[n] + (1- \alpha)\, S[n]$; $0 < \alpha < 1$

CPU burst Estimate (Exponential Average)

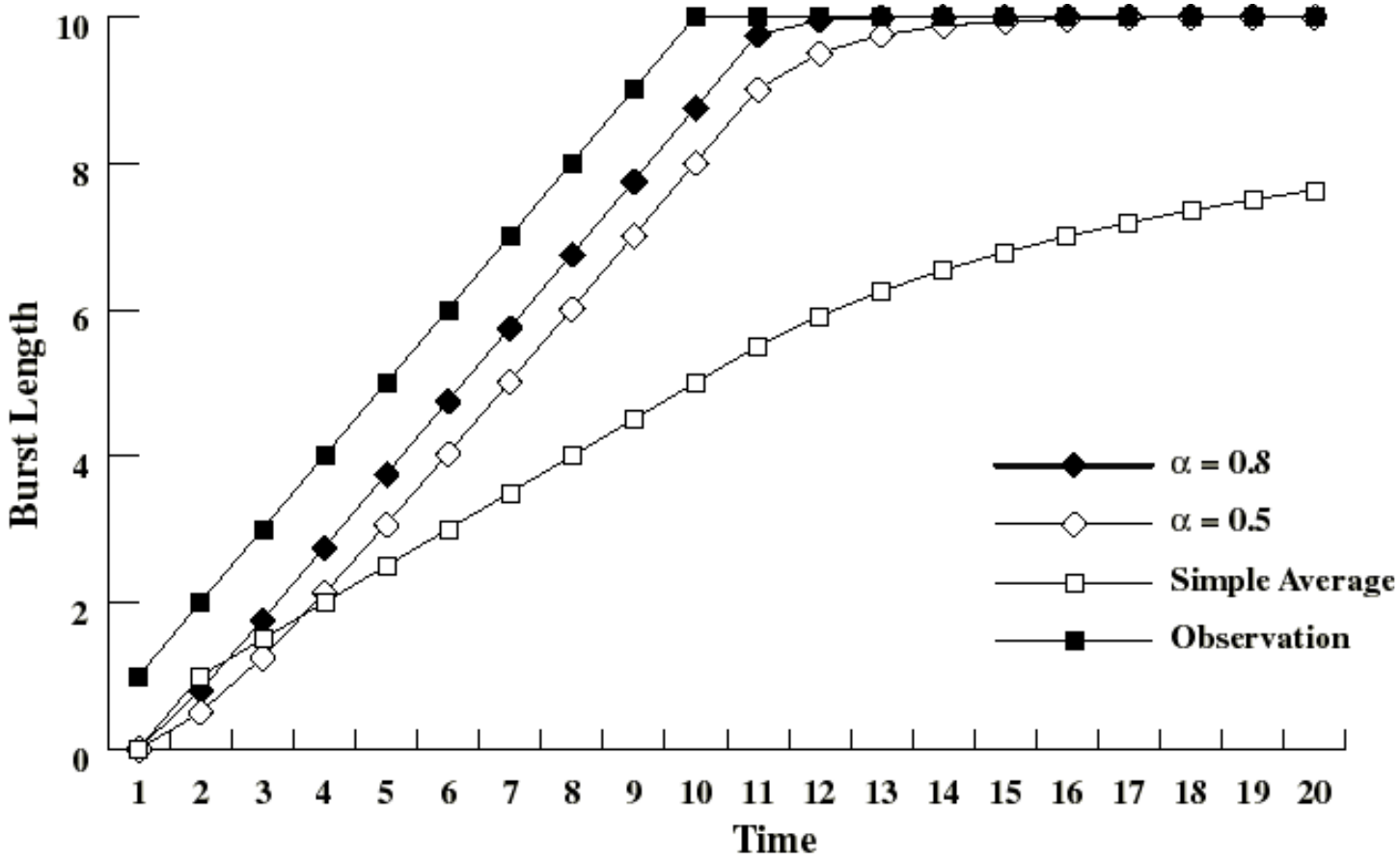
- Recent instances have higher weights, whenever $\alpha > 1/n$
- Expanding the estimated value shows that the weights of past instances decrease exponentially
  - $S[n+1] = \alpha\, T[n] + (1-\alpha)\,\alpha\, T[n-1] + \ldots (1-\alpha)^{i}\,\alpha\, T[n-i] +$
  - $\ldots + (1-\alpha)^{n} S[1]$
  - The predicted value of 1st instance, S[1], is usually set to 0 to give priority to new processes

Exponentially Decreasing Coefficients

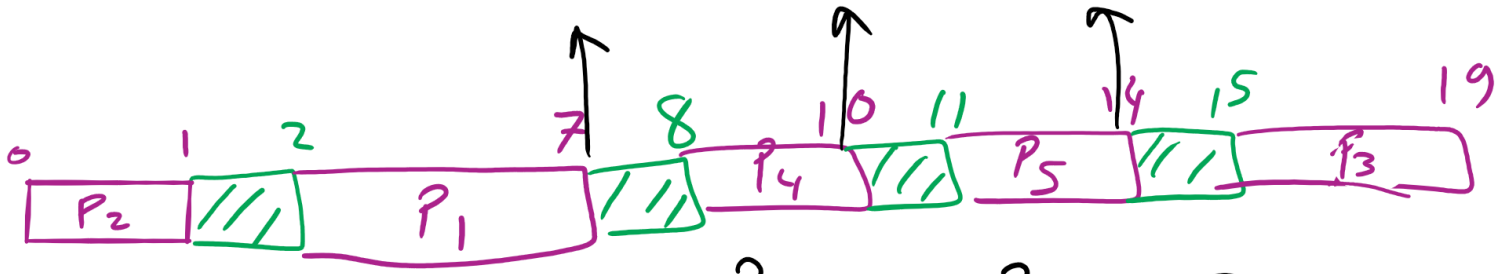


Exponentially Decreasing Coefficients

- S[1] = 0 to give high priority to new processes
- Exponential averaging tracks changes in process behavior much faster than simple averaging



FCFS Problem in HW7



$$A.R.T. = \frac{P_1 \quad P_2 \quad P_3 \quad P_4 \quad P_5}{(7-2)+(1-0)+(19-8)+(10-4)+(14-7)}{5}$$

$$= \frac{5+1+11+6+7}{5} = 6ms$$

$$CPU\ Util = \frac{19-4}{19} = \frac{15}{19}$$

CPU Burst Estimation

$$S[n+1] = \alpha T[n] + (1-\alpha) S[n]$$

$$0 < \alpha < 1$$