Operating System Concepts

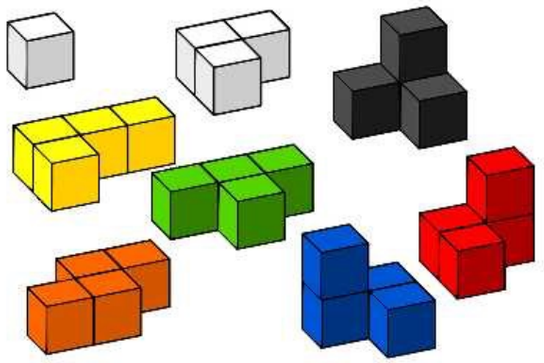
Lecture 12: CPU Scheduling

Omid Ardakanian
oardakan@ualberta.ca
University of Alberta

MWF 12:00-12:50 VVC 2 215

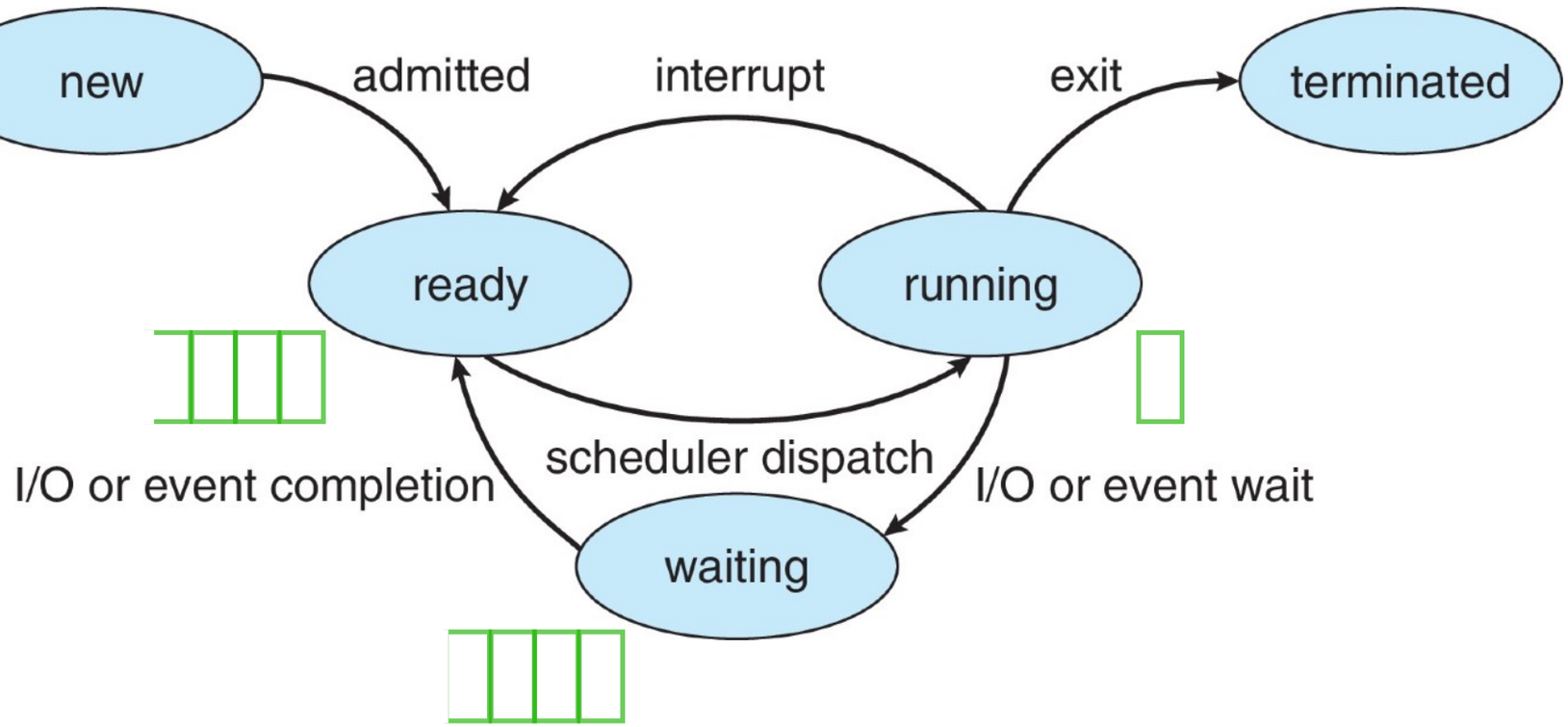
Today's class

- Scheduling goals
  - switching CPU among processes can make computers more productive
- Scheduling algorithms
  - FCFS: First-Come, First-Served
  - RR: Round Robin
  - SJF: Shortest Job First



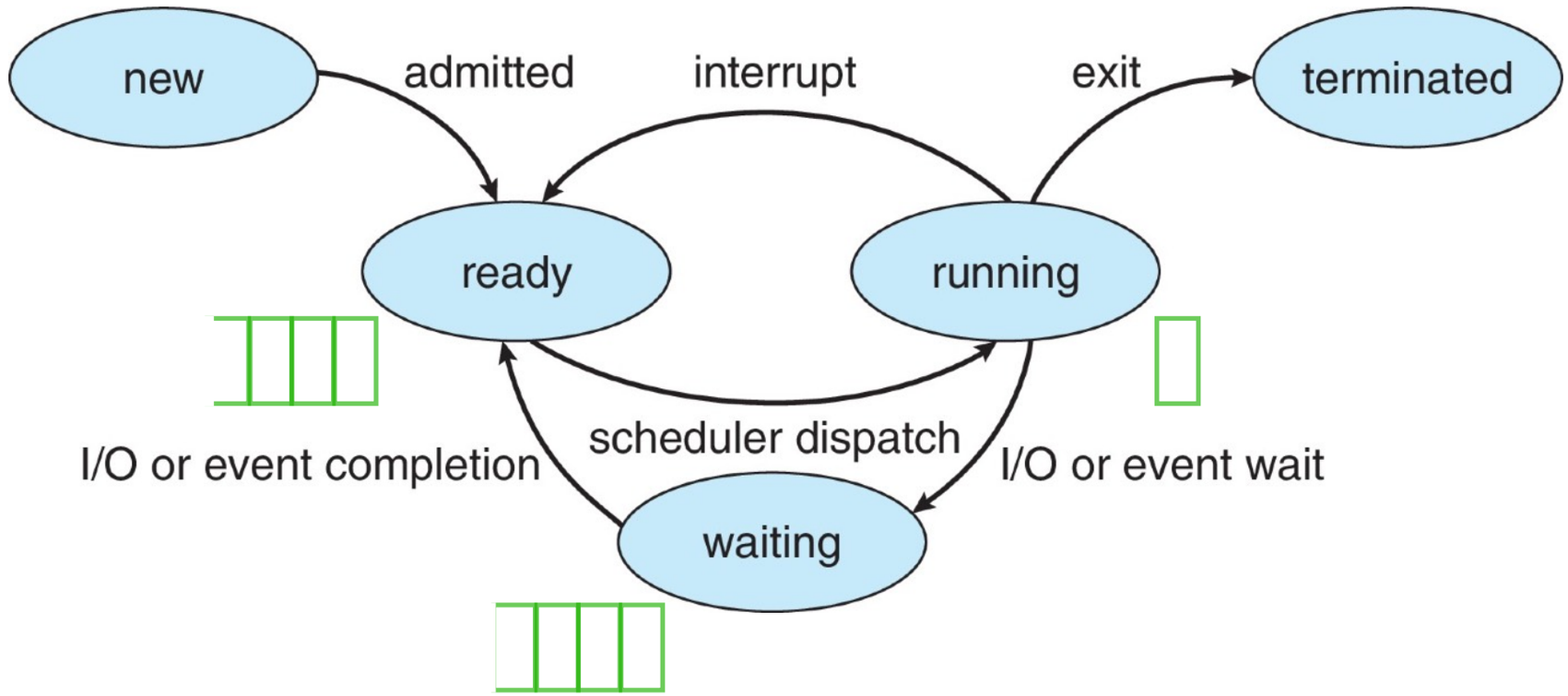
Scheduling processes

Process/thread scheduling is the basis of multiprogrammed systems
Definition: which process/thread to remove from ready queue?



Scheduling processes

Process/thread scheduling is the basis of multiprogrammed systems
Definition: which process/thread to remove from ready queue?

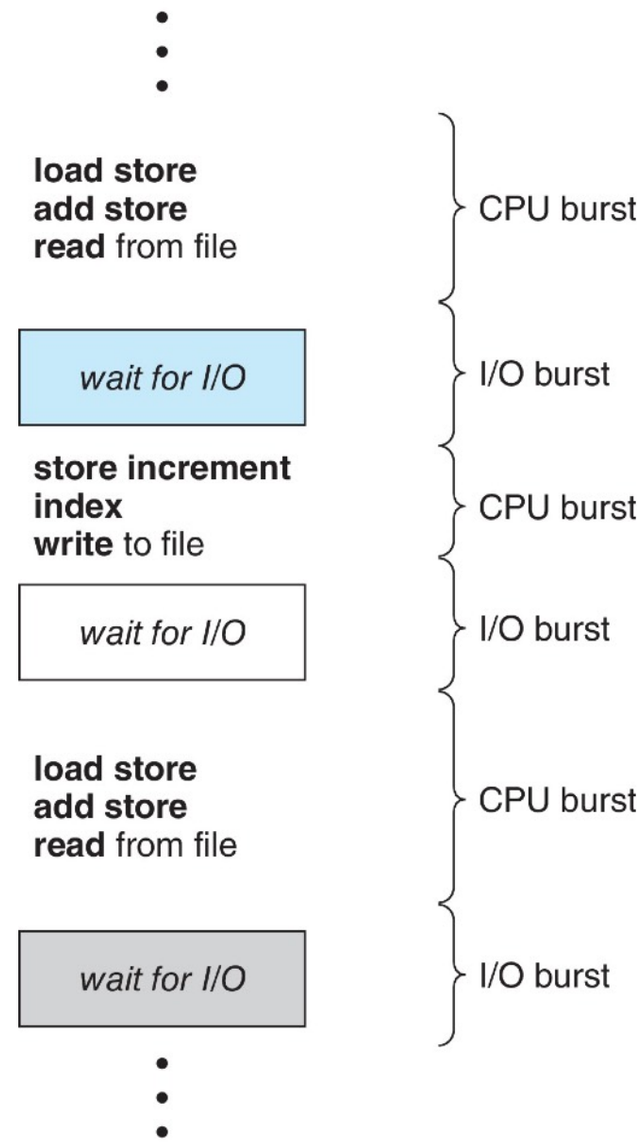


scheduling is a difficult problem in general because of

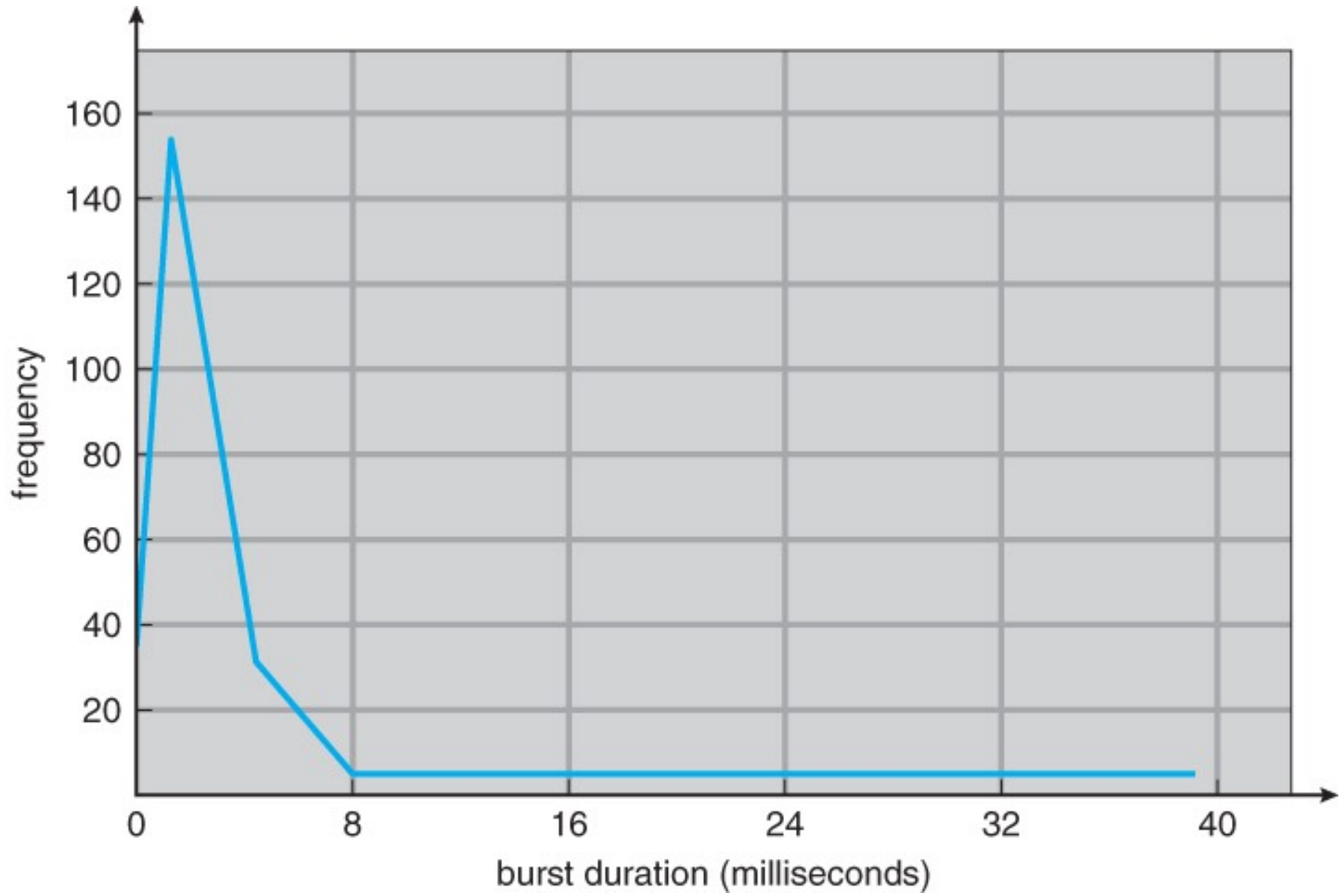
1. competing objectives (keeping users happy and the system fully utilized)
   - users care about getting their job done quickly
   - system cares about overall efficiency, energy use, and …
2. huge variation in workload

Programs alternate between bursts of CPU and I/O activities

- Process execution consists of a cycle of CPU execution and I/O wait
  - CPU burst followed by I/O burst
  - CPU burst distribution is our main concern
- three types of processes
  - CPU bound (spends more time doing computation)
  - I/O bound (spends more time doing I/O)
  - balanced
- if all processes are I/O bound, the ready queue will almost always be empty and if all of them are CPU bound then the I/O waiting queue will almost always be empty
  - the best performance is achieved when we have a good mix of I/O and CPU bound processes



CPU bursts pattern for a typical process



Definition

- workload
  - a set of tasks for a system to run, each task has a specific arrival time and a burst length (and a deadline in real-time systems)
- performance metrics
  - criteria considered for comparing scheduling policies (e.g., throughput, response time)
- scheduling policy
  - decision about the order of executing tasks for a given workload
  - different policies have difference performance criteria
- overhead
  - the extra work done by the scheduler (e.g., context switching)
- fairness
  - fraction of resources (CPU cycles) provided to each task
  - max-min fairness: maximize the minimum allocation given to a task

Scheduling

- long-term scheduling (job scheduling)
  - how does the OS determine the degree of multiprogramming, i.e., the number of jobs executing at once in the primary memory?
  - admitting jobs and loading them into memory for execution to provide a balanced mix of jobs
- short-term scheduling (CPU scheduling)
  - how does the OS select a process from the ready queue to execute?
- off-line scheduling computes a schedule given the entire set of tasks (assuming the knowledge of arrival times and burst lengths)
- on-line scheduling makes decisions as tasks arrive
  - which one yields better performance?

Scheduling opportunities

- the kernel runs the scheduler at least when
  - a process switches from running to waiting
  - an interrupt occurs (e.g., timer expires)
  - a process is created or terminated
- non-preemptive system
  - the scheduler must wait for one of these events
- preemptive system
  - the scheduler can interrupt a running process

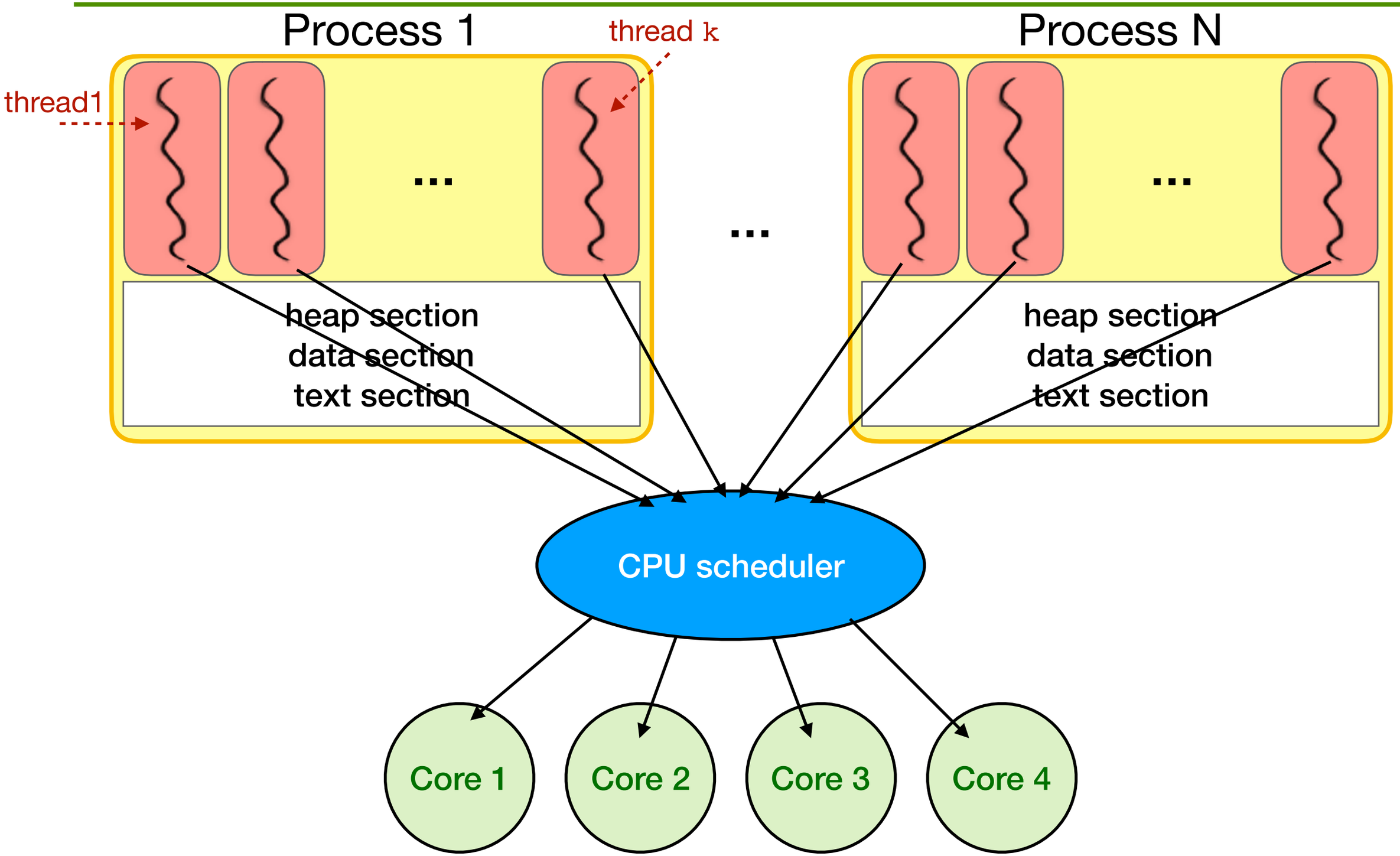
Performance criteria for scheduling

- **CPU Utilization** (higher is better)
  - percentage of time CPU is busy
- **Throughput** (higher is better)
  - number of processes completing in a unit of time (e.g., in one second)
  - overhead reduces the throughput
- **Waiting time** (lower is better)
  - total amount of time that process is in ready queue
- **Turnaround time** (lower is better)
  - length of time it takes to run process from initialization to termination, including all waiting time
- **Response time** (lower is better)
  - what user sees from keypress to character on screen
  - time between when process is ready to run and its next I/O request (or completion time)

How to choose a scheduling policy?

- choose a CPU scheduler that optimizes all criteria simultaneously
  - maximize CPU utilization and throughput; minimize turnaround time, waiting time, and response time (**not usually possible**)
- choose a scheduling algorithm that implements a particular policy
  - minimize average response time - provide output to user as quickly as possible and process their input as soon as it is received
  - minimize variance of response time - in interactive systems, predictability may be more important than a low average with a high variance
  - maximize throughput - has two components:
    - minimize overhead (OS overhead, context switching)
    - efficient use of system resources (CPU, I/O devices)
  - minimize waiting time - give each process the same amount of time on the processor (fair?). This might actually increase the average response time

Process versus thread scheduling



Assumptions

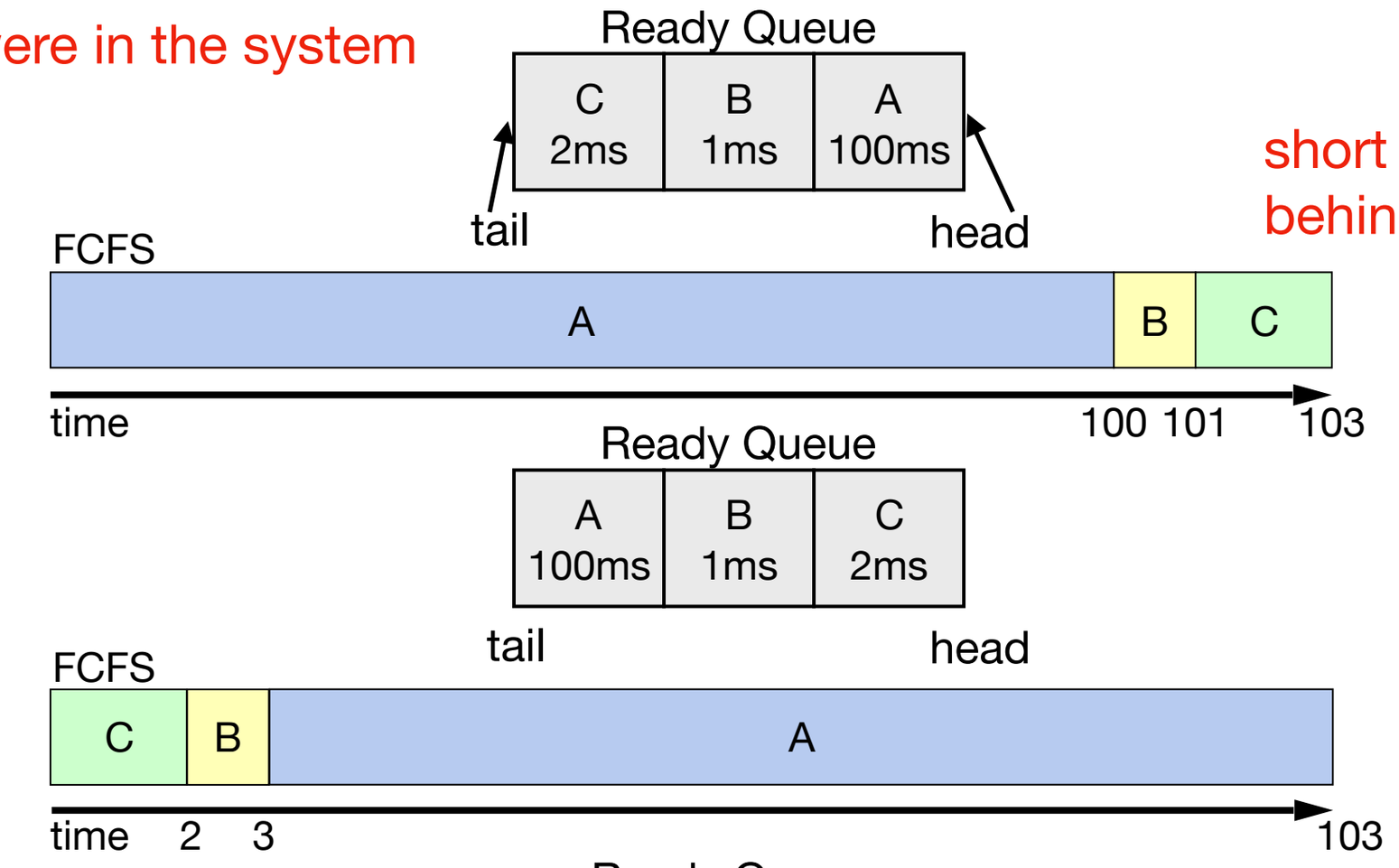
- we make a couple of simplifying assumptions today
  - one process per user
  - one thread per process
  - independent processes
  - one processing core only
- researchers developed these algorithms in the 70's when these assumptions were more realistic
  - relaxing some of these assumptions is still an open problem

FCFS scheduling

- FCFS: First-Come-First-Served (or FIFO)
  - the scheduler executes jobs **to completion** in the order they arrive
  - in early FCFS schedulers, the job did not relinquish the CPU even when it was doing I/O
  - we will assume a FCFS scheduler that runs when processes are blocked on I/O, but that is non-preemptive, i.e., the job keeps the CPU until it blocks (say on an I/O device)

FCFS example

A, B, C were in the system at time 0



short processes stuck behind long processes

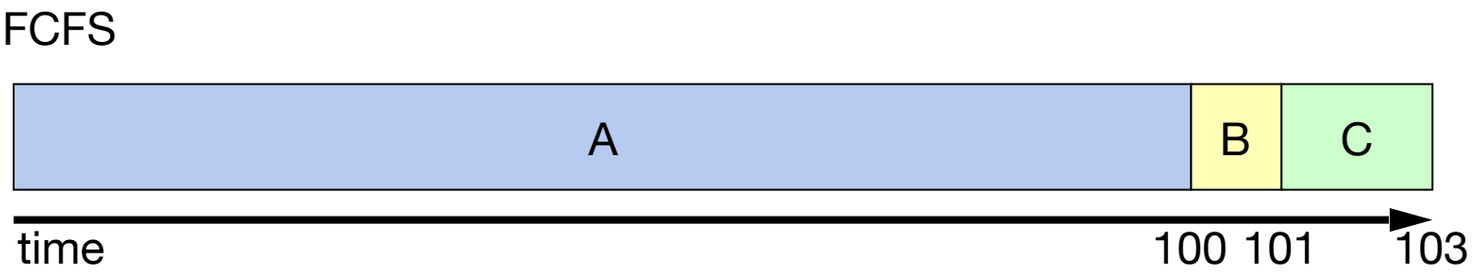
Avg. turnaround time: 101.3
Avg. waiting time: 67

Avg. turnaround time: 36
Avg. waiting time: 1.6

Avg. turnaround time: 20
Avg. waiting time: 10

FCFS example

Process	Burst length	Arrival time
A	100	0
B	1	10
C	2	50



Avg. turnaround time:
[100+(101-10)+(103-50)]/3 = 81.3

Avg. waiting time:
[0+(100-10)+(101-50)]/3= 47

On what workloads is FCFS particularly bad?

- Advantage: simplicity and low overhead

On what workloads is FCFS particularly bad?

- Advantage: simplicity and low overhead
- Disadvantage
  - average wait time is highly variable as short jobs may wait behind long jobs
    - if tasks are variable in size, FCFS can have very poor average response time
    - If tasks are equal in size, FCFS is optimal in terms of average response time
  - not fair
  - may lead to poor overlap of I/O and CPU since CPU-bound processes will force I/O bound processes to wait for the CPU, leaving the I/O devices idle

Round Robin scheduling

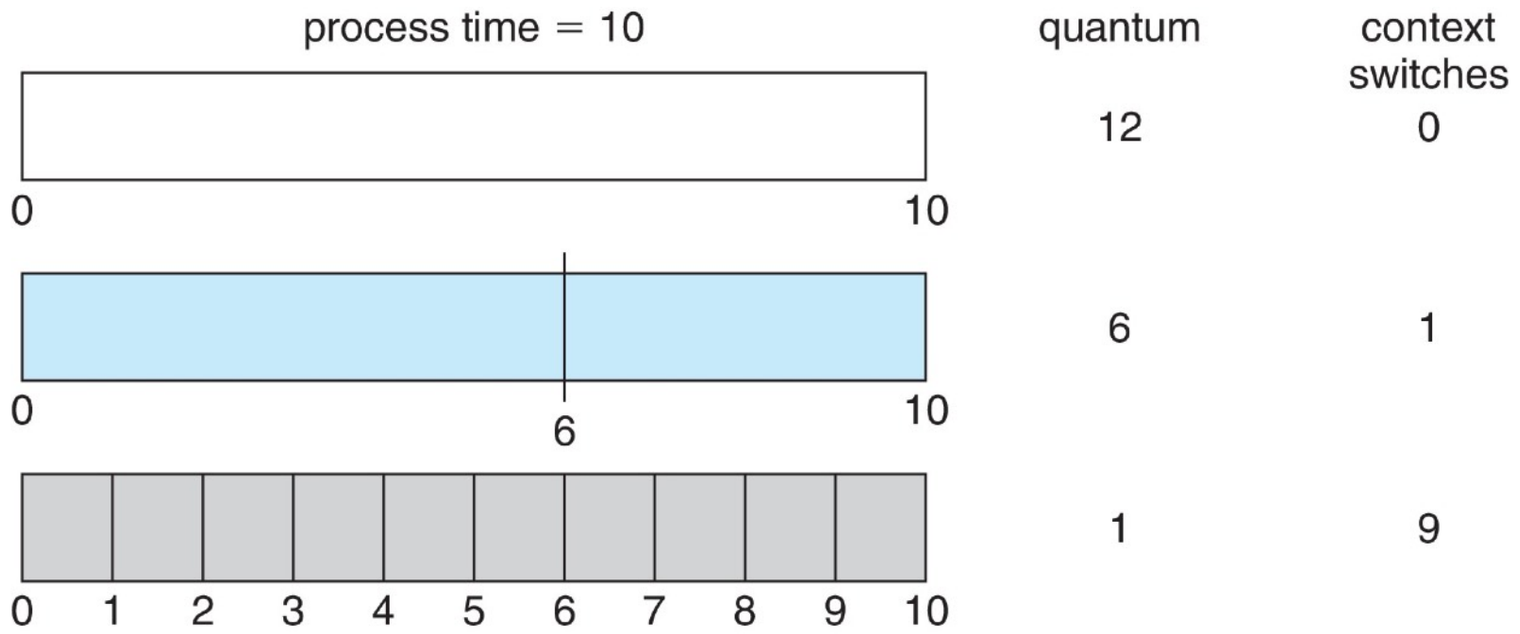
- each task gets resource for a fixed period of time (time quantum)
  - if it does not complete, it goes back in line (at the end of the ready queue)
  - how to implement? add a timer and use a preemptive policy

Round Robin scheduling

- each task gets resource for a fixed period of time (time quantum)
  - if it does not complete, it goes back in line (at the end of the ready queue)
  - how to implement? add a timer and use a preemptive policy
- with quantum length Q *ms*, process waits at most (N-1)*Q *ms* to run again if there is a total of N processes

Round Robin scheduling

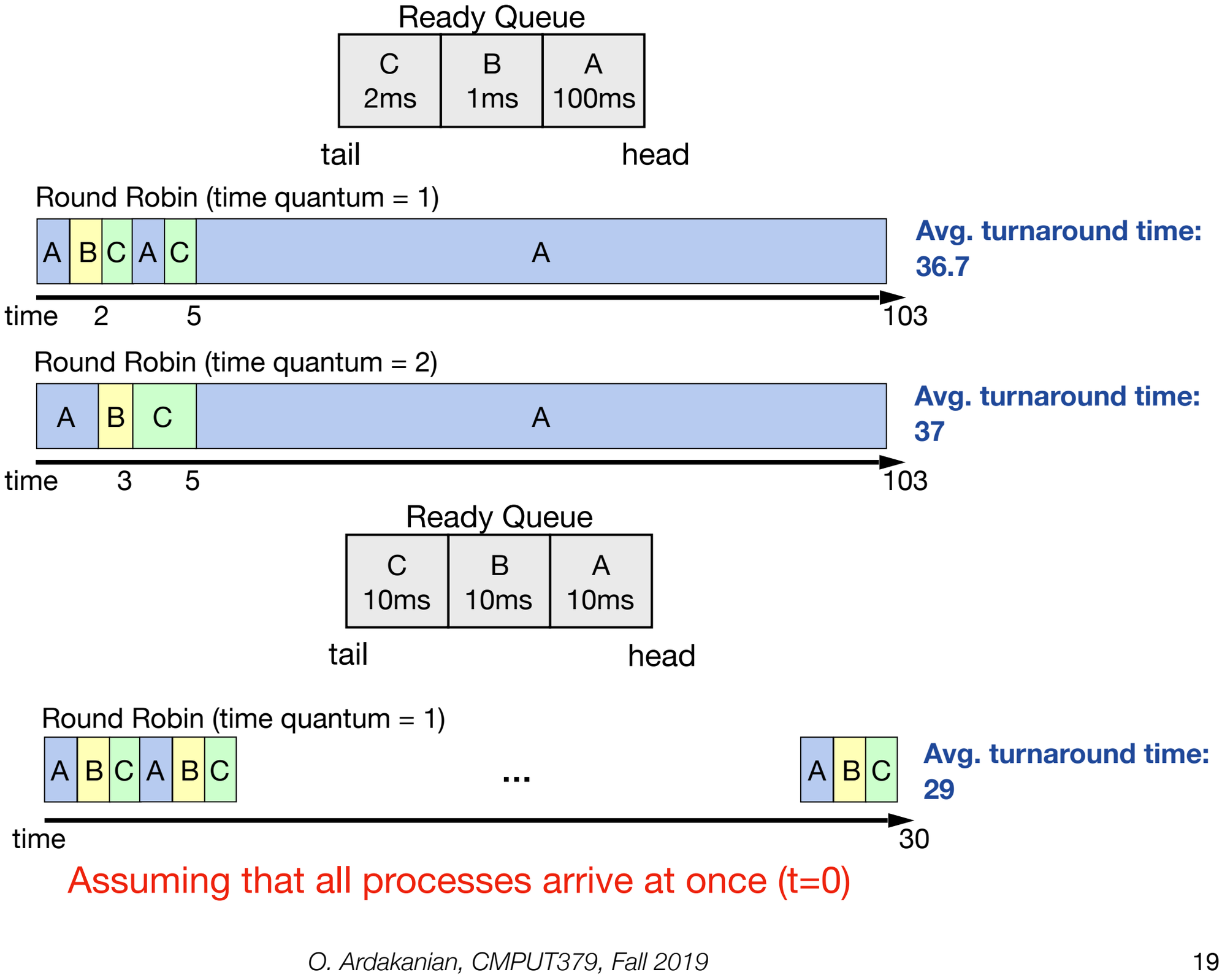
- each task gets resource for a fixed period of time (time quantum)
  - if it does not complete, it goes back in line (at the end of the ready queue)
  - how to implement? add a timer and use a preemptive policy
- with quantum length Q *ms*, process waits at most (N-1)*Q *ms* to run again if there is a total of N processes
- choosing the time quantum (Q) is key
  - if too long - waiting time suffers, degenerates to FCFS if processes are never preempted
  - if too short - throughput suffers because too much time is spent context switching (high overhead)
  - balance these tradeoffs by selecting a time slice where context switching is roughly 1% of the time slice
  - today: typical time slice is 10-100ms, context switch time is 0.1-1ms



Round Robin scheduling

- variants of round robin scheduling are used in most time-sharing systems
- Advantage: round robin is fair; each job gets an equal shot at the CPU
- Disadvantage: average waiting time can be bad if tasks are equal in size

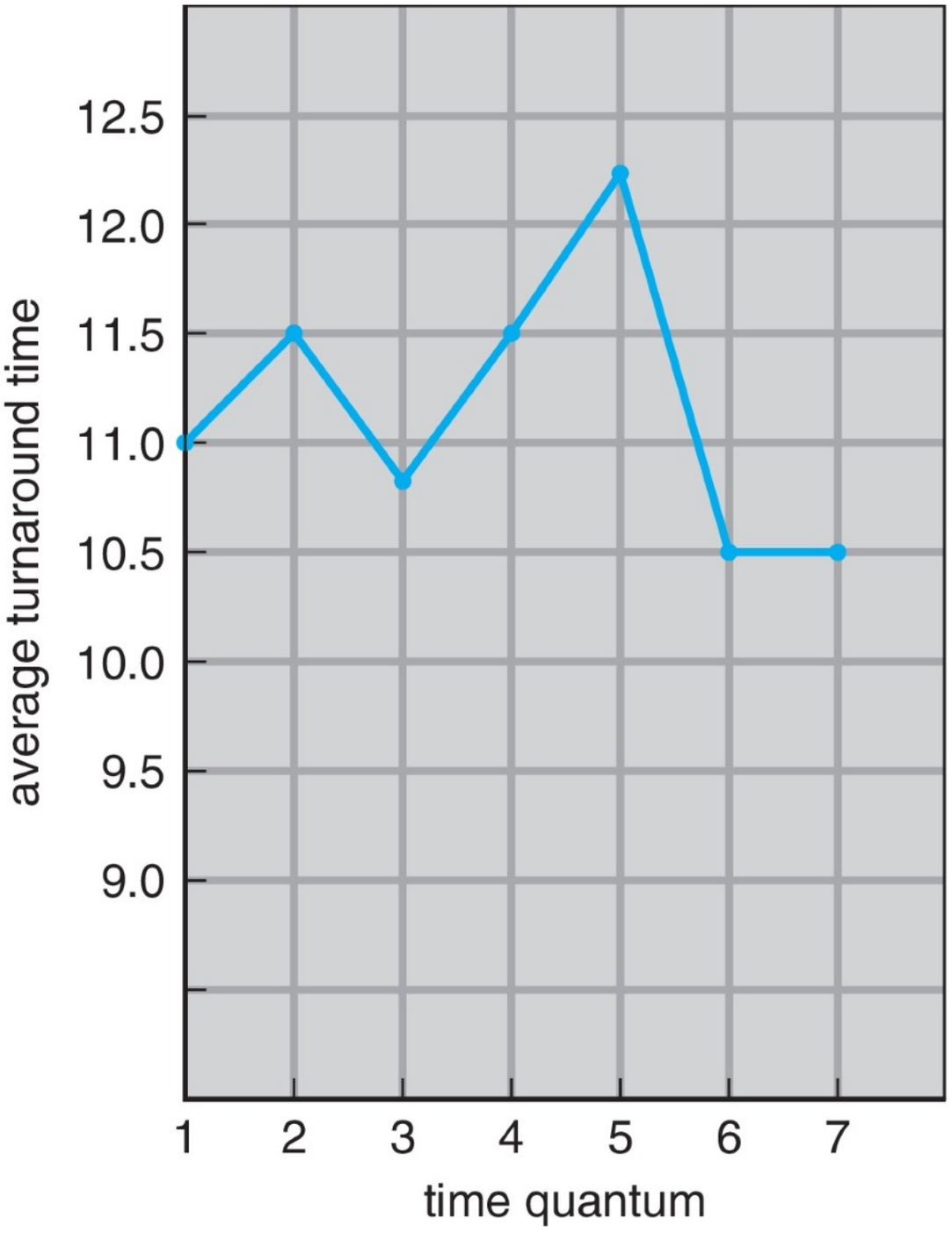
RR example



Assuming that all processes arrive at once (t=0)

Setting the time quantum

- suppose there is no context switching overhead and all process arrive at t=0



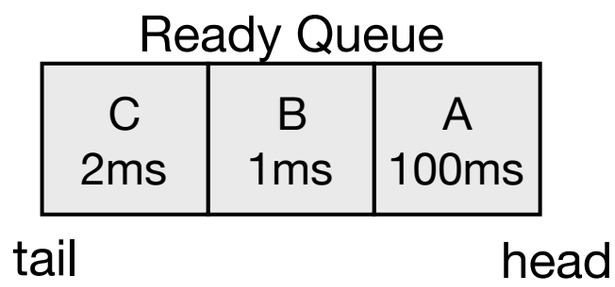
process	time
P_1	6
P_2	3
P_3	1
P_4	7

increasing the time quantum can have different effects on the average turnaround time

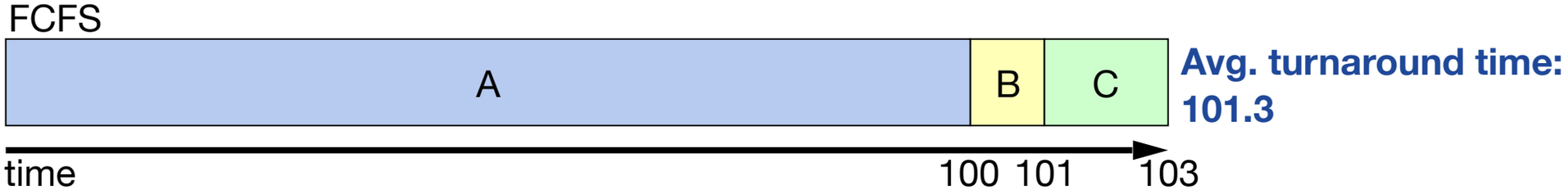
SJF scheduling

- schedule the job that has the least (expected) amount of work (CPU time) to do until its next I/O request or termination
  - much less sensitive to the arrival order!
- if tasks are variable in size, Round Robin approximates SJF
- Advantages
  - provably optimal with respect to minimizing the average waiting time
  - works for preemptive and non-preemptive schedulers
    - Preemptive SJF is called Shortest Remaining Time First (SRTF)
- Disadvantages
  - impossible to predict the amount of CPU time a job needs
  - with SRTF, long running CPU bound jobs can starve (if new short jobs keep arriving)

Scenario A

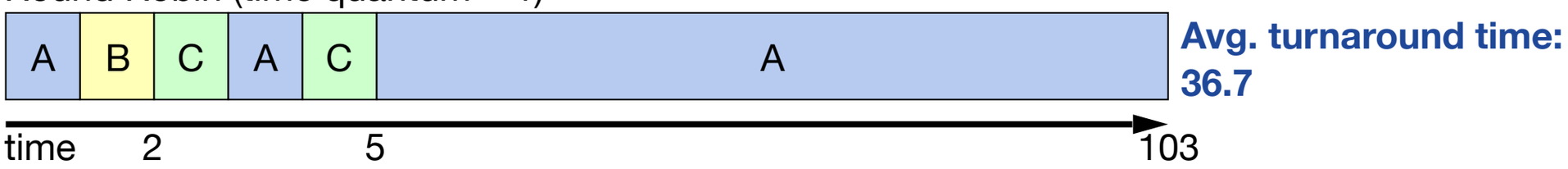


FCFS



Avg. turnaround time: 101.3

Round Robin (time quantum = 1)

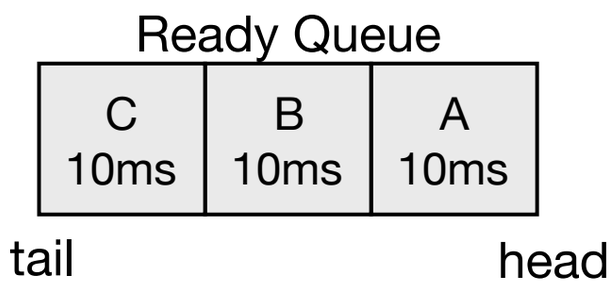


Avg. turnaround time: 36.7

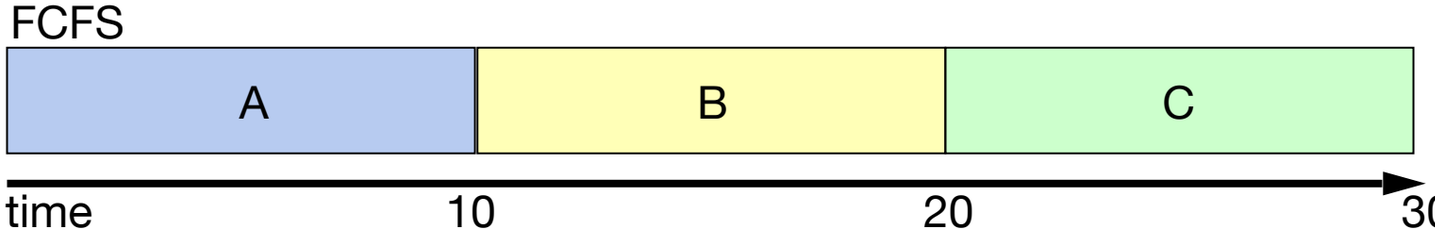
SJF

Avg. turnaround time: 35.7

Scenario B

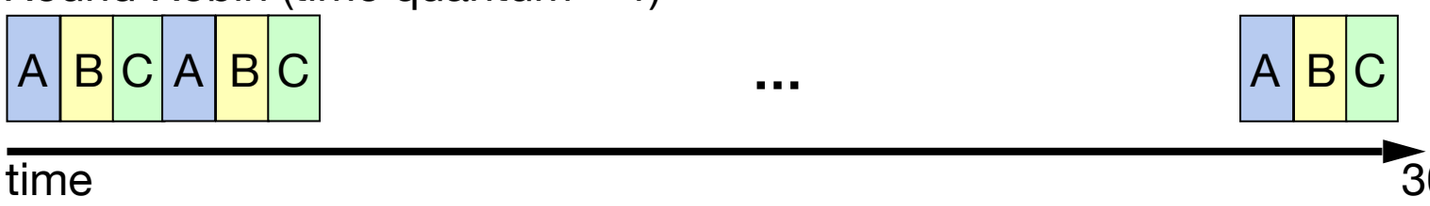


FCFS



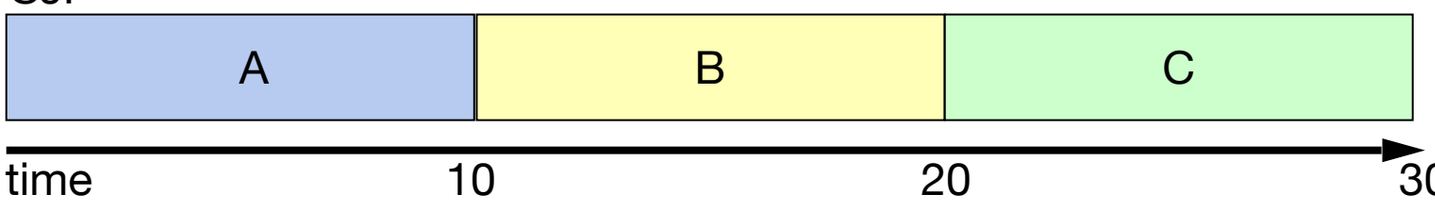
Avg. turnaround time: (10+20+30)/3=20

Round Robin (time quantum = 1)



Avg. turnaround time: 29

SJF

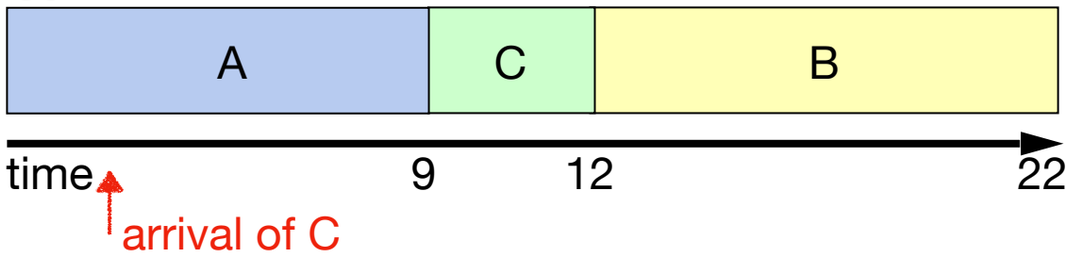


Avg. turnaround time: 20

Comparing SRTF and SJF

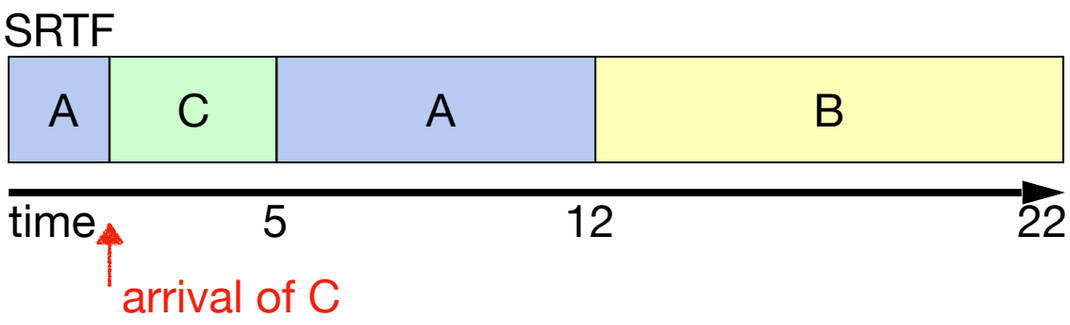
Process	Burst length	Arrival time
A	9	0
B	10	0
C	3	2

SJF



Avg. turnaround time:
[9+(12-2)+22]/3=13.6

SRTF



Avg. turnaround time:
[(5-2)+12+22]/3=12.3