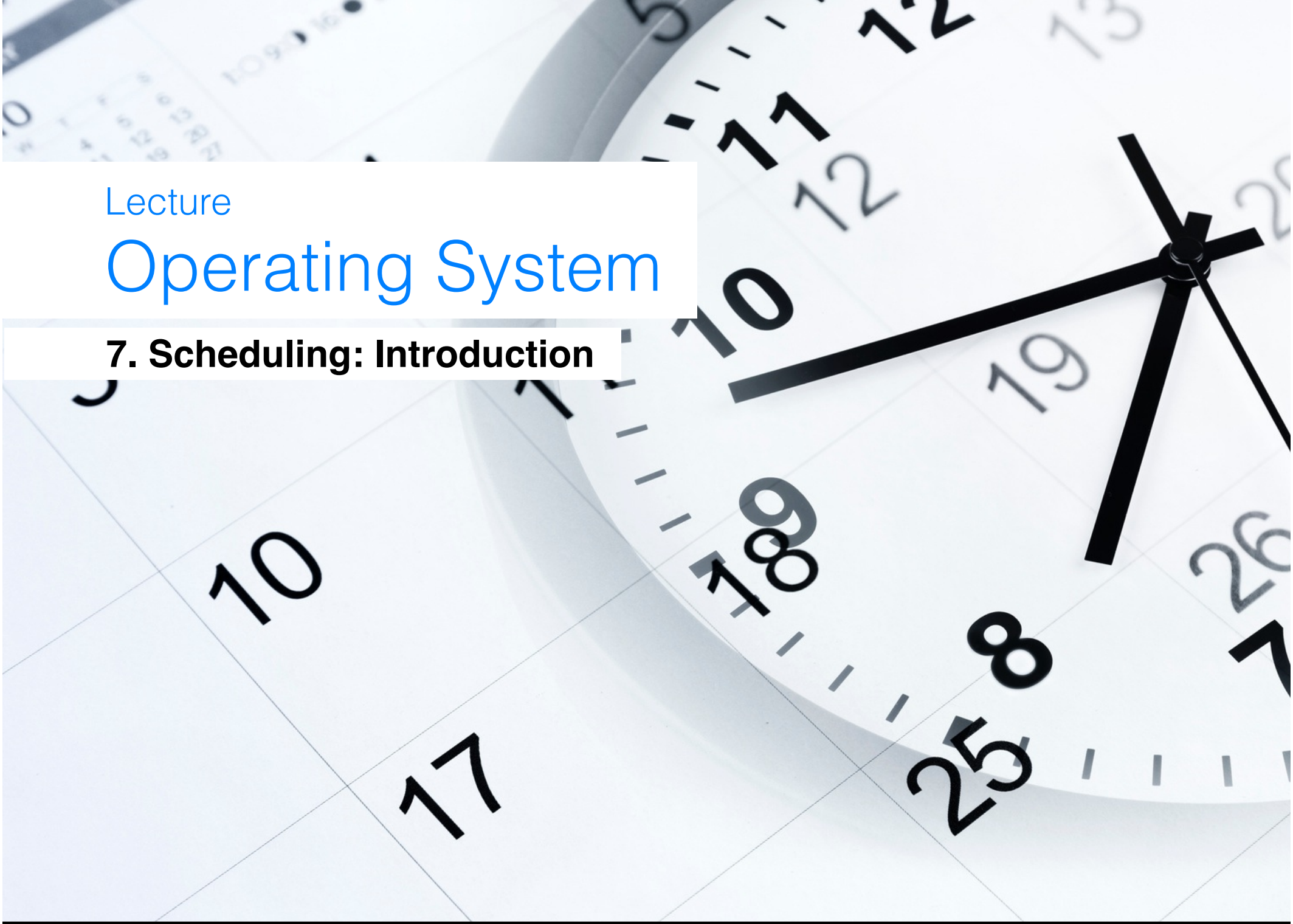


Lecture

# Operating System

## 7. Scheduling: Introduction

Professor Dr. Michael Mächtel

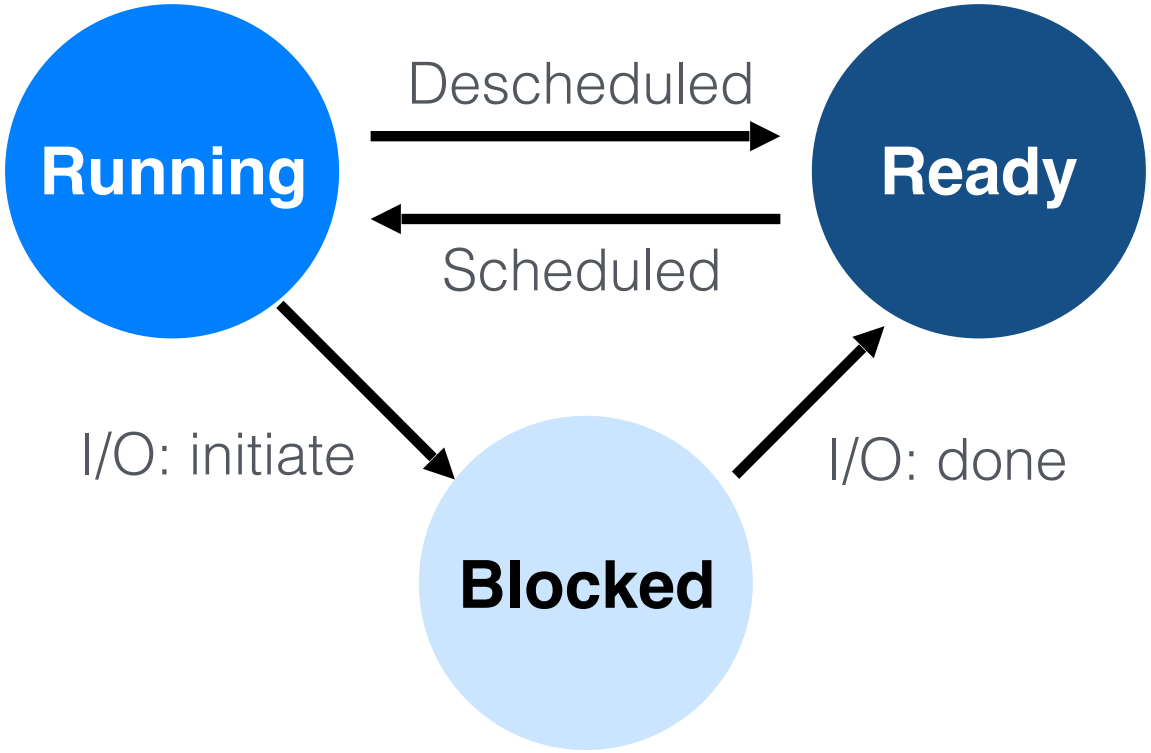
# 7. Scheduling: Introduction

1. **Scheduler: Policy to determine which process gets CPU when**
2. **Policies Examples**

# 7. Scheduling: Introduction

1. **Scheduler: Policy to determine which process gets CPU when**
2. **Policies Examples**

# State Transitions




How to transition? ("mechanism")

When to transition? ("policy")

# Vocabulary

- **Workload**: set of job descriptions (arrival time, run_time)
  - Job: View as current CPU burst of a process
  - Process alternates between CPU and I/O
    process moves between ready and blocked queues
- **Scheduler**: logic that decides which ready job to run
- **Metric**: measurement of scheduling quality

# Workload Assumptions

1. Each job runs for the **same amount of time**.
2. All jobs **arrive** at the same time.
3. All jobs only use the **CPU** (i.e., they perform no I/O).
4. The **run-time** of each job is known.

# Scheduling Metrics

- Performance metric: **Turnaround time**
  - The time at which the **job completes** minus the time at which the **job arrived** in the system.

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

- Another metric is **fairness**.
  - Performance and fairness are often at odds in scheduling.

# More Metrics

- **Minimize Response Time**
  - Schedule interactive jobs promptly so users see output quickly

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

- **Minimize Waiting Time**
  - Do not want to spend much time in Ready queue
- **Maximize Throughput**
  - Want many jobs to complete per unit of time
- **Maximize Resource Utilization**
  - Keep expensive devices busy
- **Minimize Overhead**
  - Reduce number of context switches

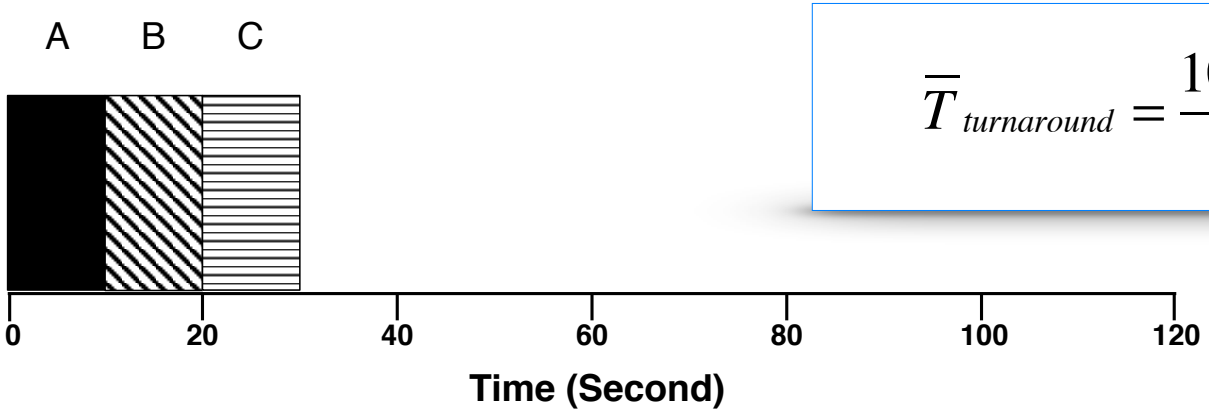
# 7. Scheduling: Introduction

1. Scheduler: Policy to determine which process gets CPU when
2. **Policies Examples**

# First In, First Out (FIFO)

- First Come, First Served (FCFS)
  - Very simple and easy to implement
  - **Non-preemptive** scheduler
- Example:
  - A arrived just before B which arrived just before C.
  - Each job runs for 10 seconds.

| Job | $T_{arrival}$ | $T_{runtime}$ |
|---|---|---|
| A | 0 | 10 |
| B | 0 | 10 |
| C | 0 | 10 |




$$\overline{T}_{turnaround} = \frac{10 + 20 + 30}{3} = 20\,\text{sec}$$

# Workload Assumptions

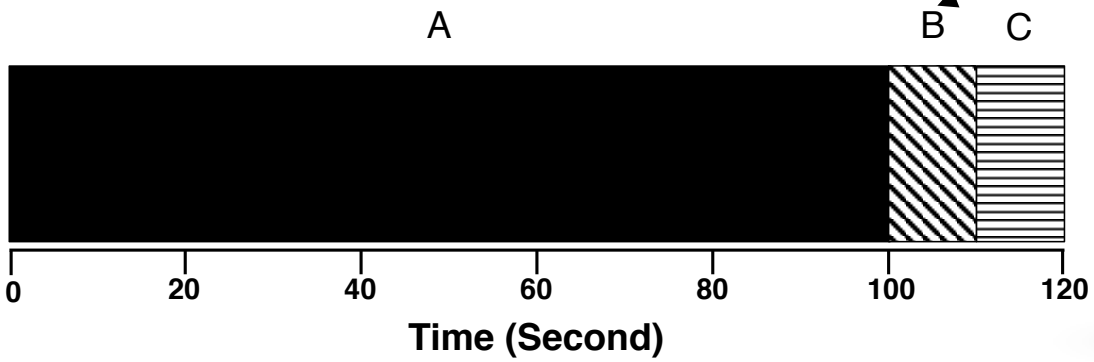
1. ~~Each job runs for the **same amount of time**~~.
2. All jobs **arrive** at the same time.
3. All jobs only use the **CPU** (i.e., they perform no I/O).
4. The **run-time** of each job is known.

# Why FIFO is not that great?

- Each job no longer runs for the same amount of time.
- Example:
  - A arrived just before B which arrived just before C.
  - A runs for 100 seconds, B and C run for 10 each.

| Job | $T_{arrival}$ | $T_{runtime}$ |
|---|---|---|
| A | 0 | 100 |
| B | 0 | 10 |
| C | 0 | 10 |

Convoy effect




$$\overline{T}_{turnaround} = \frac{100+110+120}{3} = 110\,\text{sec}$$

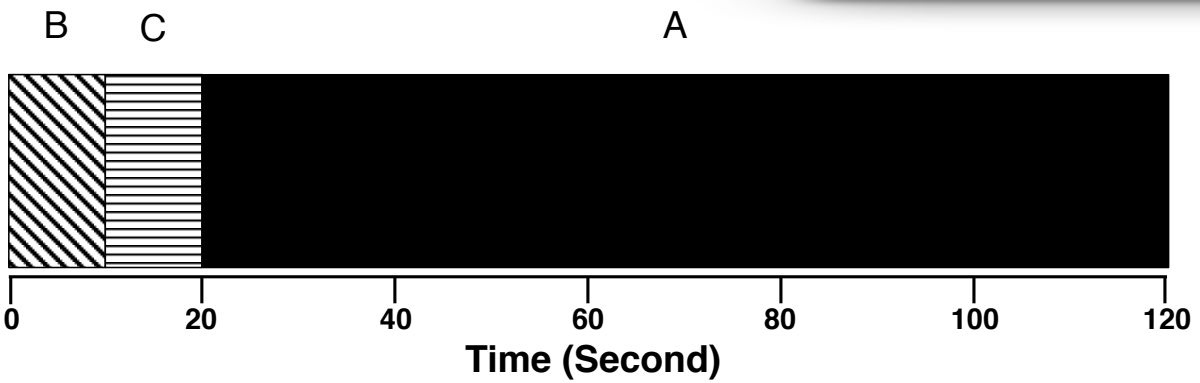
# Convoy Effect: How to solve?

- Problem with Previous Scheduler:
  - FIFO: Turnaround time can suffer when short jobs must wait for long jobs
- New scheduler:
  - SJF (Shortest Job First)
    - Choose job with smallest run_time

# Shortest Job First (SJF)

- Run the shortest job first, then the next shortest, and so on
  - **Non-preemptive** scheduler
- Example:
  - A arrived just before B which arrived just before C.
  - A runs for 100 seconds, B and C run for 10 each.

$$\overline{T}_{turnaround} = \frac{10 + 20 + 120}{3} = 50\,\text{sec}$$



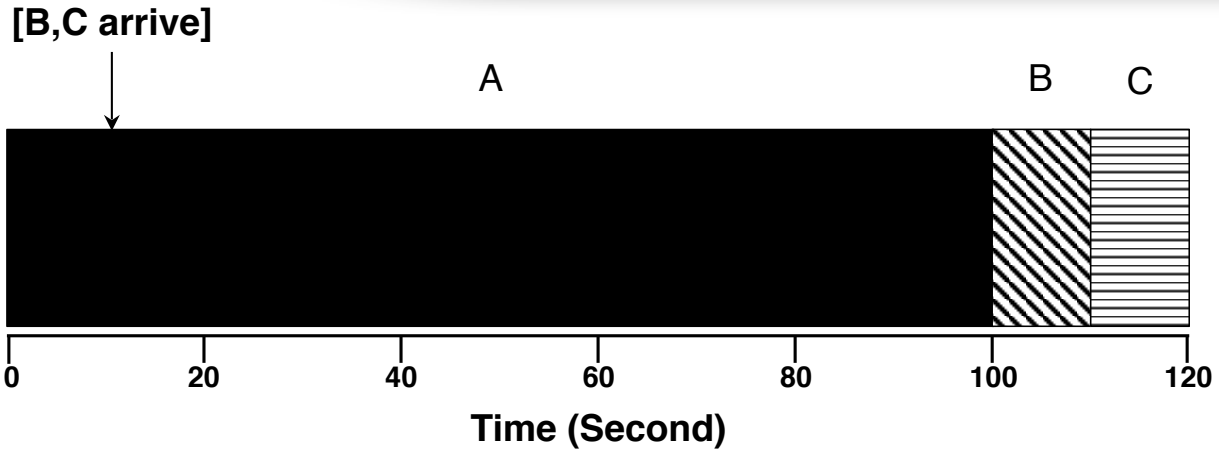
# Workload Assumptions

1. ~~Each job runs for the **same amount of time**.~~
2. ~~All jobs **arrive** at the same time.~~
3. All jobs only use the **CPU** (i.e., they perform no I/O).
4. The **run-time** of each job is known.

# SJF with Late Arrivals from B and C

- Jobs can arrive at any time.
- Example:
  - A arrives at t=0 and needs to run for 100 seconds.
  - B and C arrive at t=10 and each need to run for 10 seconds

$$\overline{T}_{turnaround} = \frac{100 + (110 - 10) + (120 - 10)}{3} = 103,33\,\text{sec}$$




| Job | $T_{arrival}$ | $T_{runtime}$ |
|---|---|---|
| A | 0 | 100 |
| B | 10 | 10 |
| C | 10 | 10 |

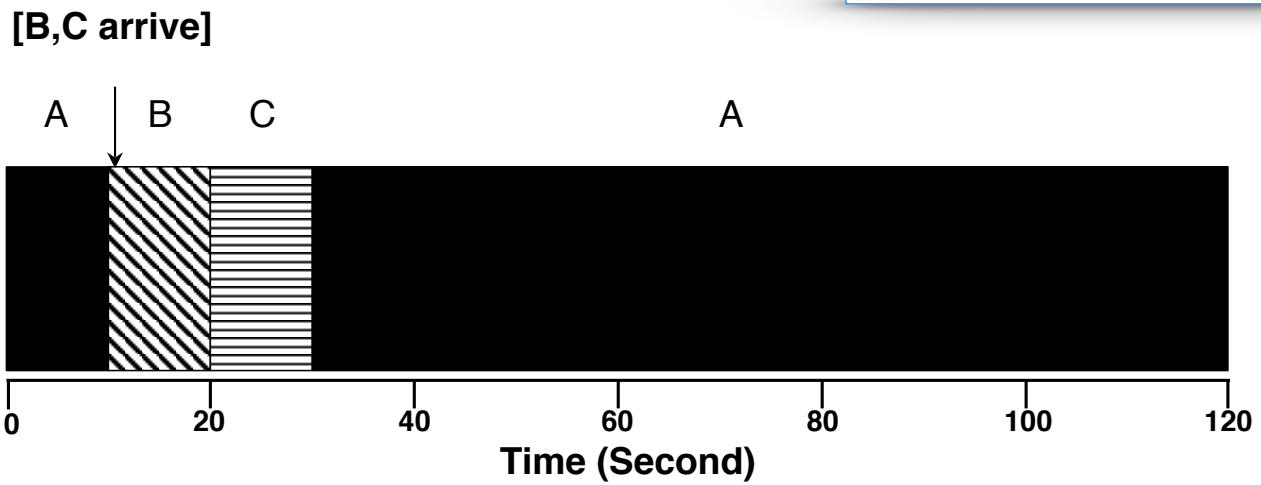
# Shortest Time-to-Completion First (STCF)

- Add **preemption** to SJF
  - Also knows as Preemptive Shortest Job First (PSJF)
  - or as Shortest Remaining Time (SRT)
- A new job enters the system:
  - Determine of the remaining jobs and new job
  - Schedule the job which has the less time left

# Shortest Time-to-Completion First (STCF)

- Example:
  - A arrives at t=0 and needs to run for 100 seconds.
  - B and C arrive at t=10 and each need to run for 10 seconds

$$\overline{T}_{turnaround} = \frac{10 + 20 + 120}{3} = 50\,\text{sec}$$




| Job | $T_{arrival}$ | $T_{runtime}$ |
|---|---|---|
| A | 0 | 100 |
| B | 10 | 10 |
| C | 10 | 10 |

# New scheduling metric: Response time

- The time from when the job arrives to the first time it is scheduled.
  - STCF and related disciplines are not particularly good for response time.

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

How can we build a scheduler that is **sensitive to response time**?

# Round Robin (RR) Scheduling

- Time slicing Scheduling
  - Run a job for a **time slice** and then switch to the next job in the **run queue** until the jobs are finished.
    - Time slice is sometimes called a scheduling quantum.
  - It repeatedly does so until the jobs are finished.
  - The length of a time slice must be a multiple of the **timer-interrupt period**.

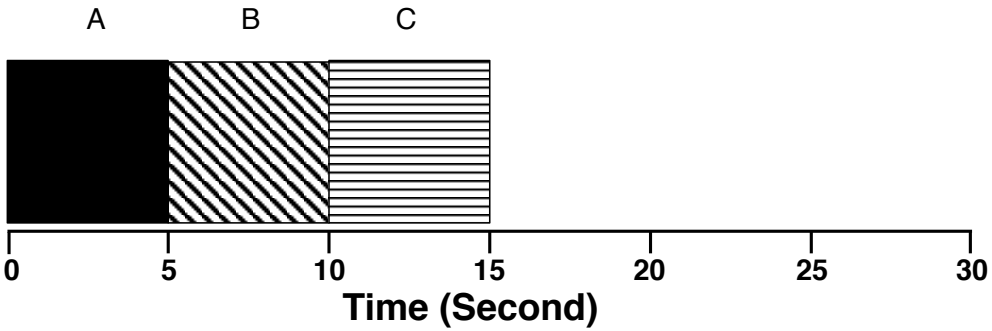
RR is fair, but performs poorly on metrics such as turnaround time

# RR Scheduling Example

- A, B and C arrive at the same time.
- They each wish to run for 5 seconds.

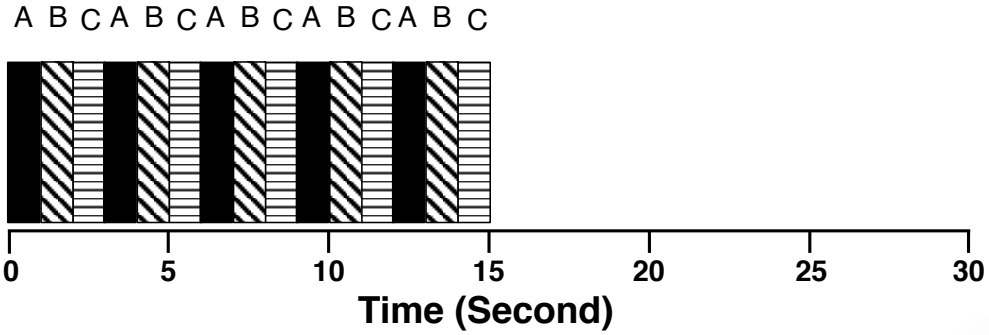
| Job | $T_{arrival}$ | $T_{runtime}$ |
|---|---|---|
| A | 0 | 5 |
| B | 0 | 5 |
| C | 0 | 5 |

**SJF (Bad for Response Time)**




$$\overline{T}_{response} = \frac{0+5+10}{3} = 5\,\text{sec}$$

**RR with a time-slice of 1sec (Good for Response Time)**



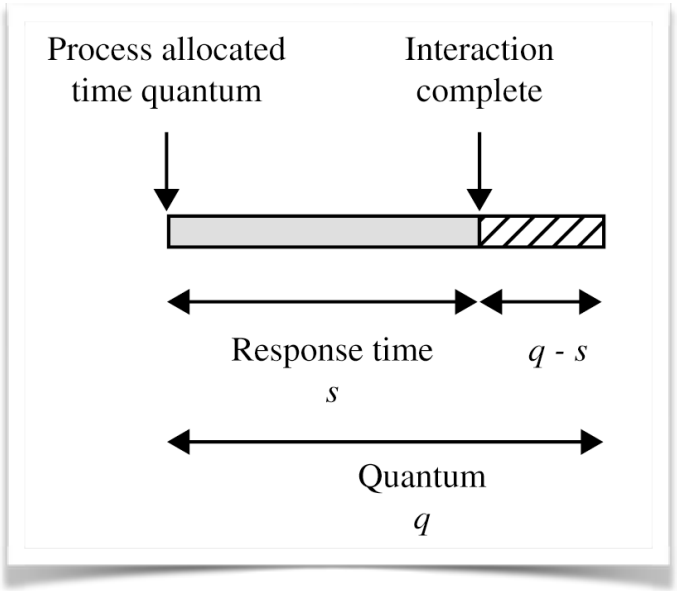

$$\overline{T}_{response} = \frac{0+1+2}{3} = 1\,\text{sec}$$

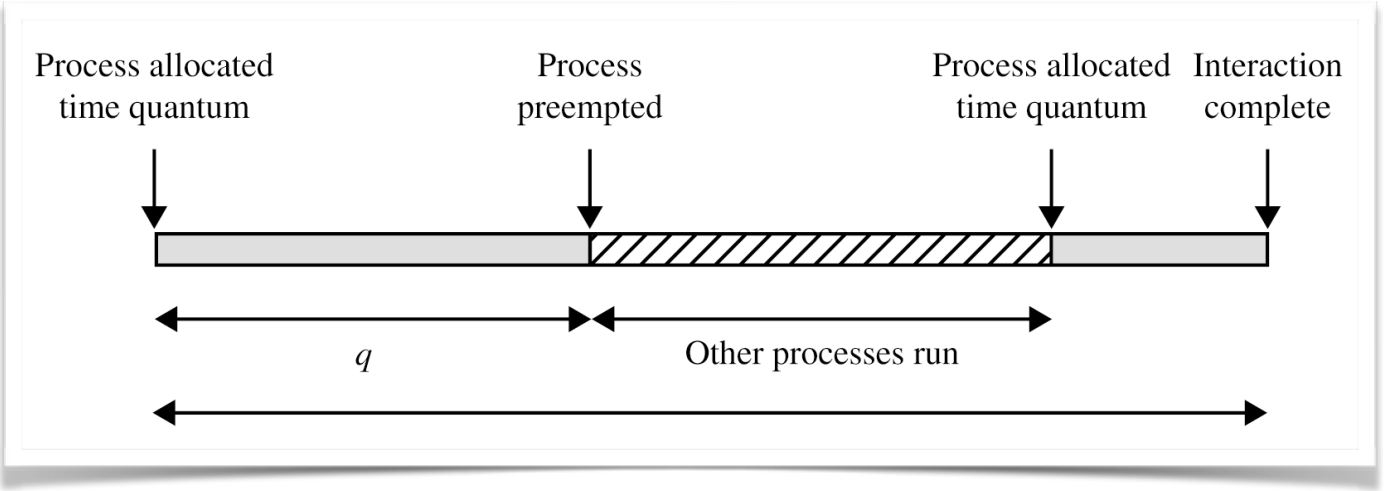
# The length of time slice is critical

- The **shorter** time slice
  - Better response time
  - The cost of context switching will dominate overall performance.
- The **longer** time slice
  - Amortize the cost of switching
  - Worse response time

# The length of time slice is critical

- Deciding on the **length** of the **time slice**
  - presents a trade-off to a system designer.
  - should be compared to typical ‚interaction'.



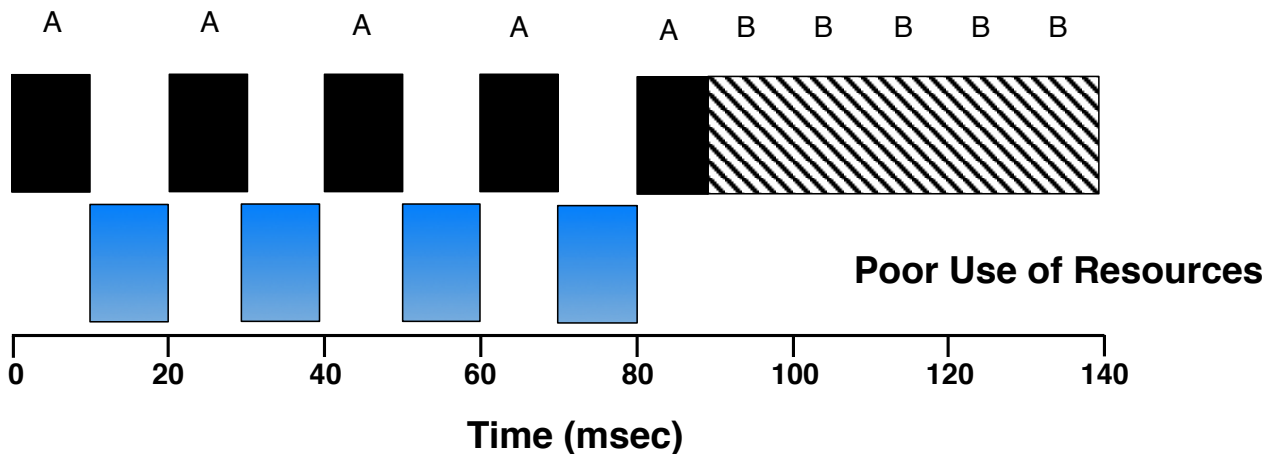
# Workload Assumptions

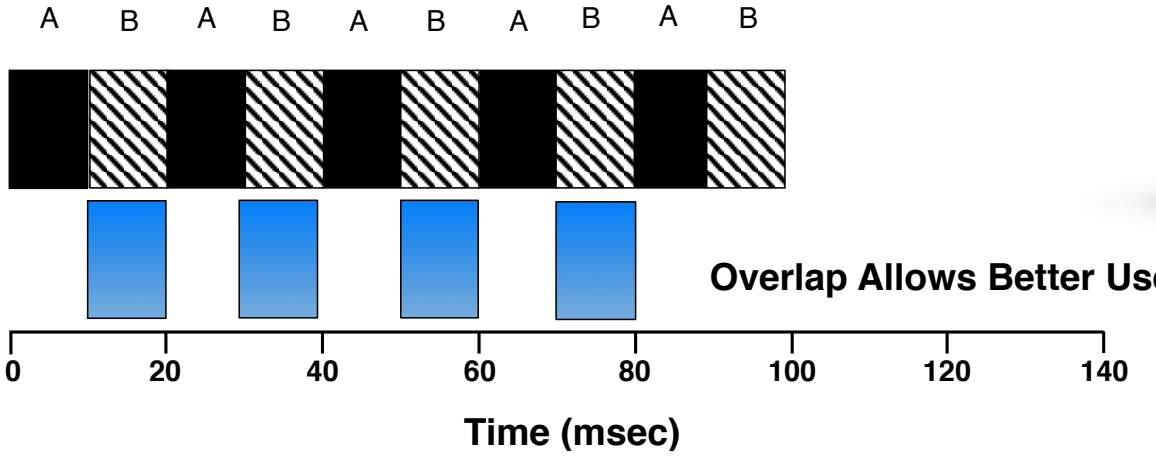
1. ~~Each job runs for the **same amount of time**.~~
2. ~~All jobs **arrive** at the same time.~~
3. ~~All jobs only use the **CPU** (i.e., they perform no I/O).~~
4. The **run-time** of each job is known.

# Incorporating I/O

- All programs perform I/O
- Example:
  - A and B need 50ms of CPU time each.
  - **A** runs for 10ms and then issues an **I/O** request
    - I/Os each take 10ms
  - **B** simply uses the CPU for 50ms and performs **no I/O**
  - The scheduler runs A first, then B after

# Incorporating I/O (Cont.)



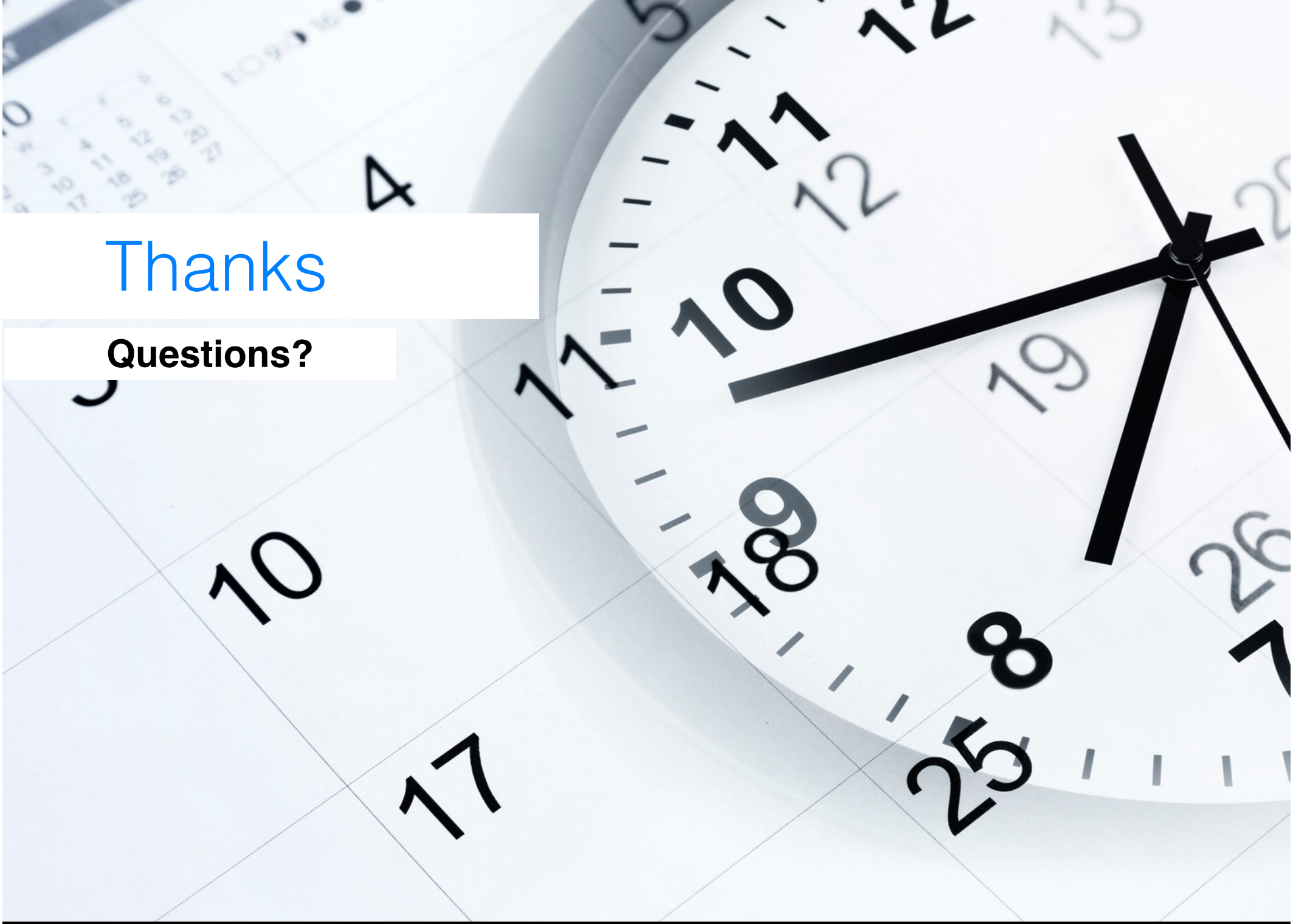

Maximize the CPU utilization

# Incorporating I/O (Cont.)

- When a job **initiates** an I/O request.
  - The job is blocked waiting for I/O completion.
  - The scheduler should schedule another job on the CPU.
- When the I/O **completes**
  - An interrupt is raised.
  - The OS moves the process from blocked back to the ready state.

# Preemptive / Non-Preemptive

- Non-Preemptive schedulers:
  - Only schedule new job when previous job voluntarily relinquishes CPU (performs I/O or exits)
  - FIFO and SJF are non-preemptive
- Preemptive schedulers
  - Potentially schedule different job at any point by taking CPU away from running job
  - STCF and RR are preemptive
    - STCF: Always run job that will complete the quickest
    - RR: Schedule after Time Slice



# Thanks

**Questions?**