Operating Systems
Process/Thread Scheduling

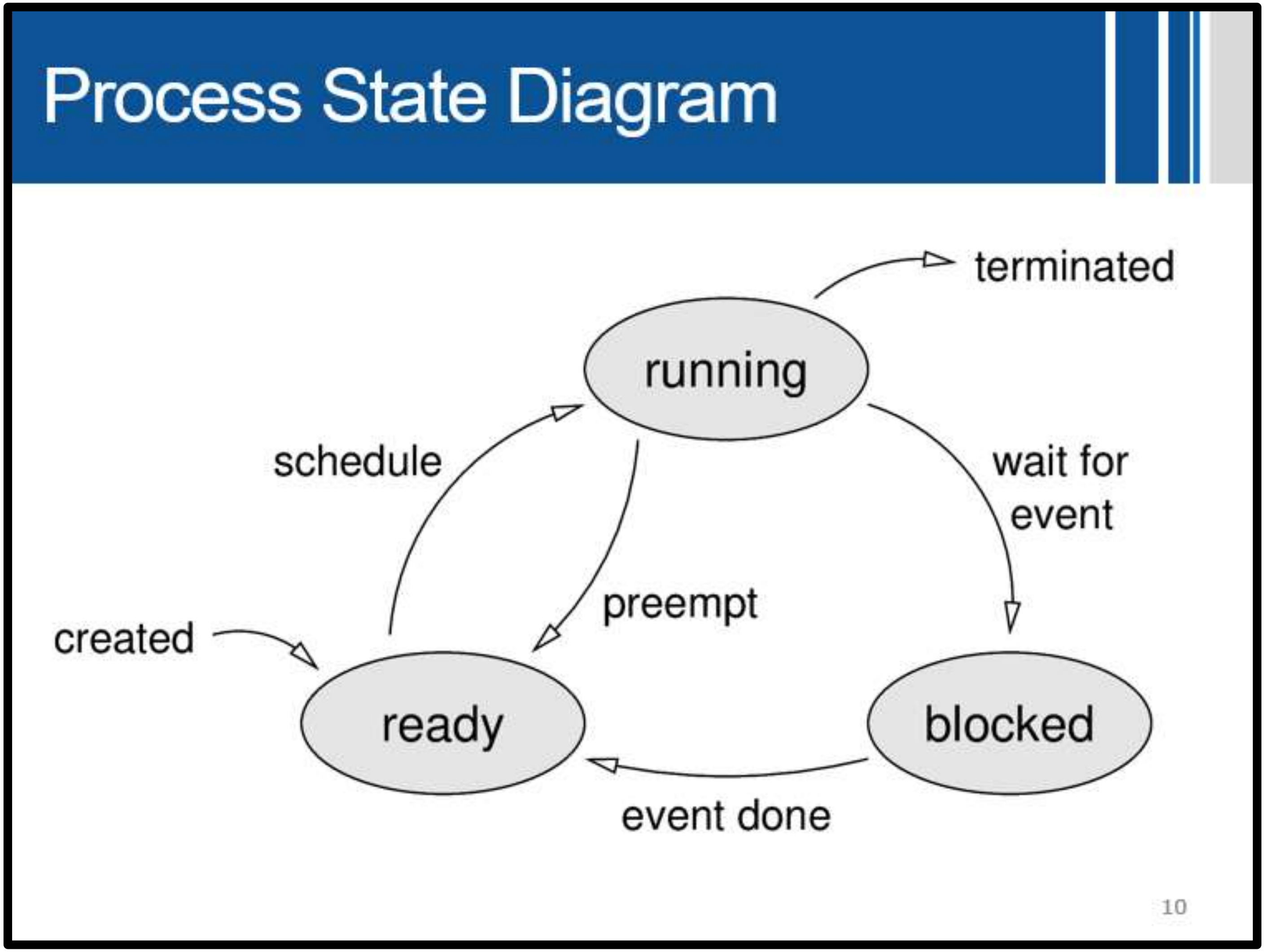
David Hay
Dror Feitelson

Schedulers are Everywhere

Every time a resource has more than one user, we need a scheduler to decide who is served next

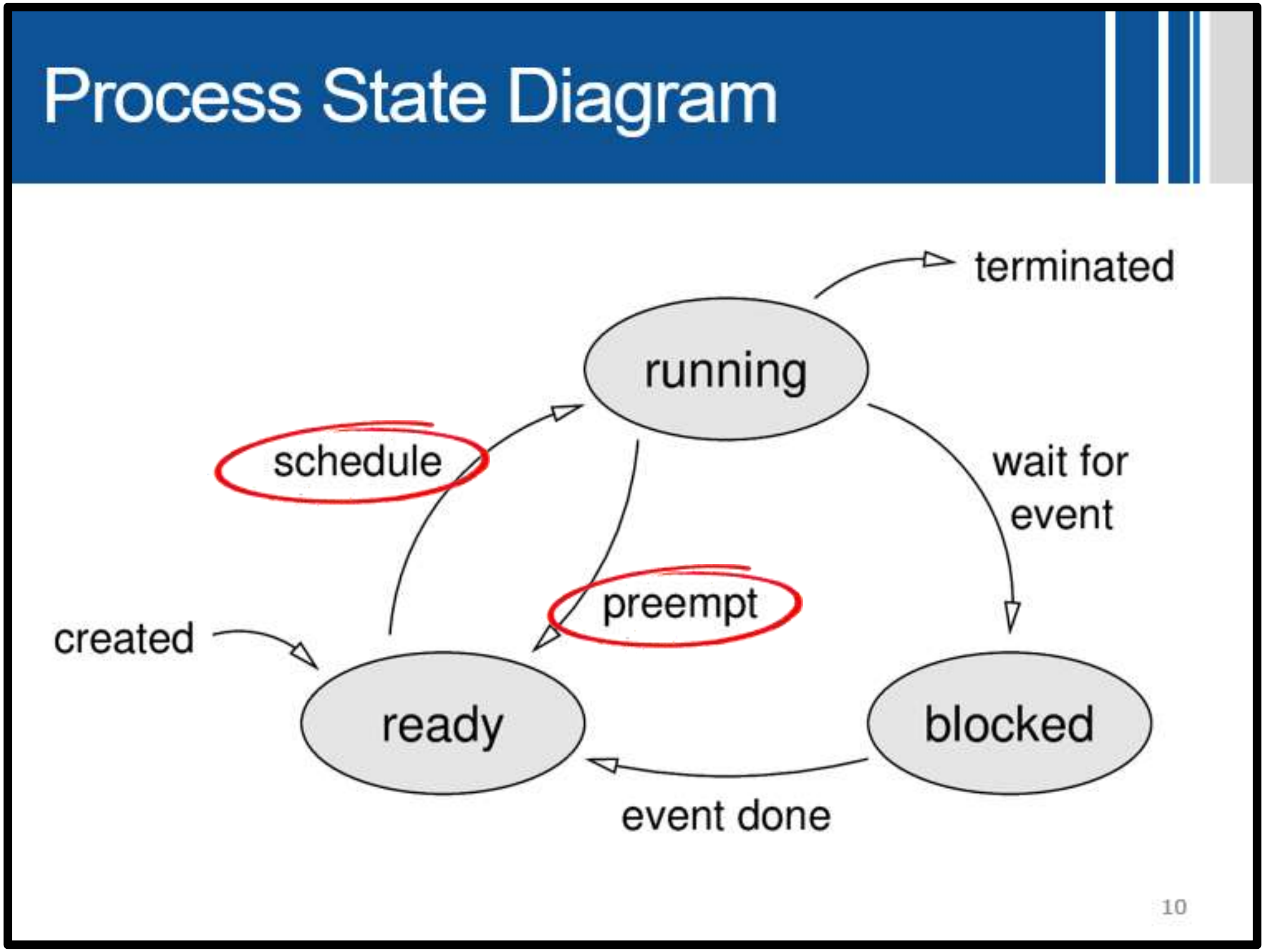
- **CPU Scheduler** (short-term scheduler)
- Mid/long-term scheduler (loading into memory)
- Scheduling jobs in the print queue
- Scheduling requests to I/O devices (e.g. disk scheduler)
- Packet Schedulers in computer networks
- Scheduling requests in a web-server
- Scheduling in a supermarket/post office

Recall…

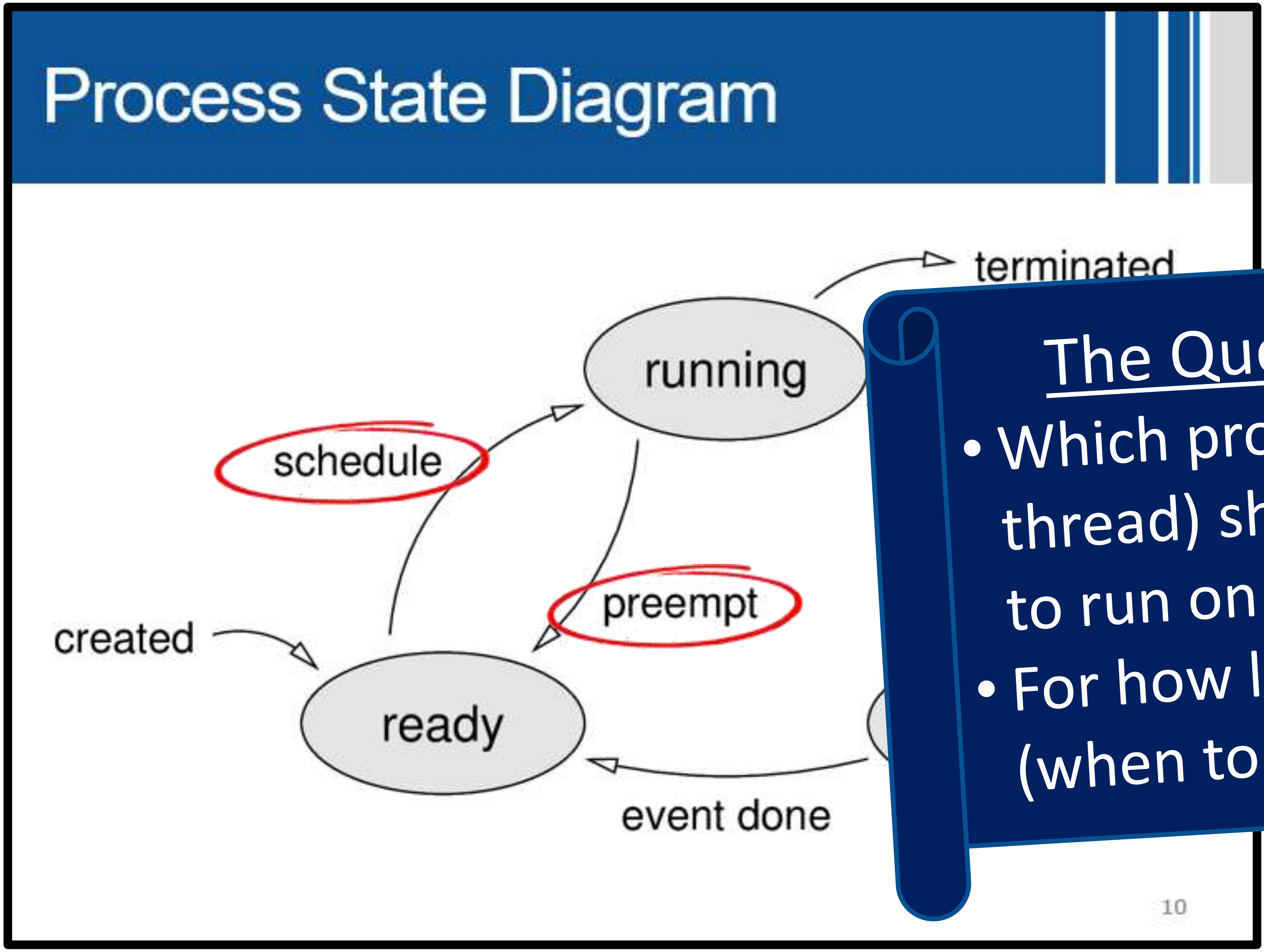


Process State Diagram
terminated
running
schedule
wait for
event
preempt
created
ready
blocked
event done
10

Recall…


Process State Diagram
terminated
running
schedule
wait for
event
preempt
created
ready
blocked
event done
10

Recall...



The Question:

- Which process (or thread) should get to run on the CPU?
- For how long? (when to preempt?)

A Scheduler and a Dispatcher

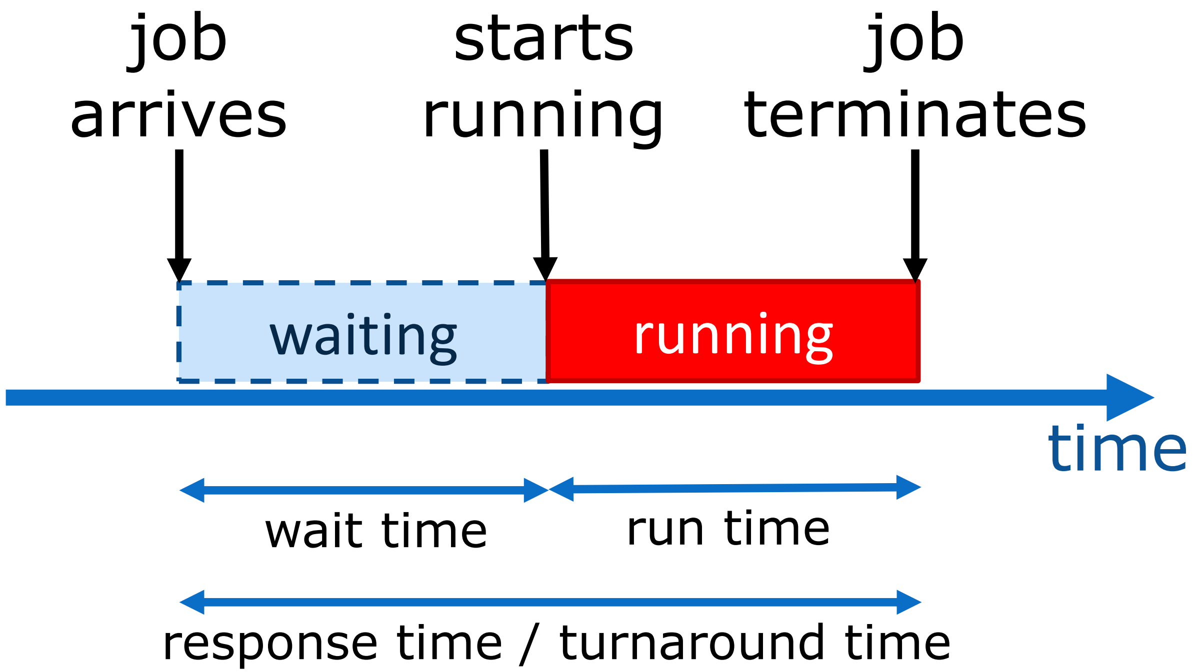
- **CPU Scheduler:** Decide which process should run on the CPU (and sometimes for how long)
  - "Short-term scheduling": tens/hundreds of times a second
- **Dispatcher:** The module responsible for executing the CPU scheduler decisions:
  - Context switch
  - Switching back to user mode

A Scheduler and a Dispatcher

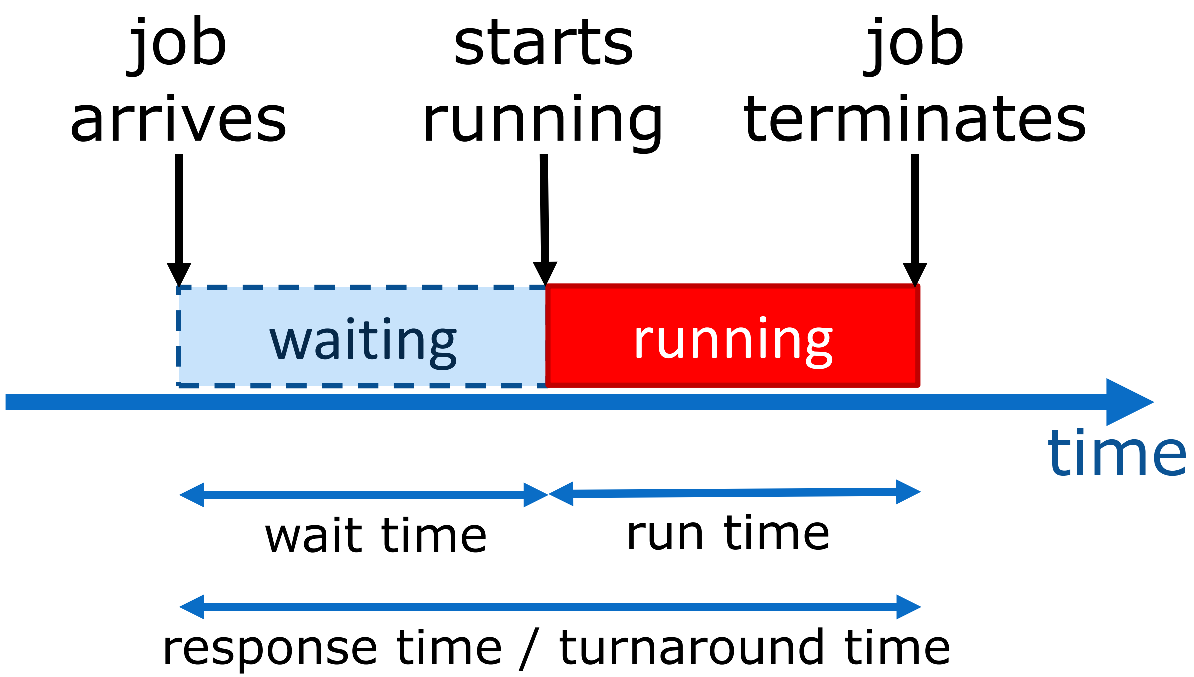
- **CPU Scheduler:** Decide which process should run on the CPU (and sometimes for how long)
  - "Short-term scheduling": tens/hundreds of times a second
- **Dispatcher:** The module responsible for executing the CPU scheduler decisions:
  - Context switch
  - Switching back to user mode

Ex2

Job Timeline

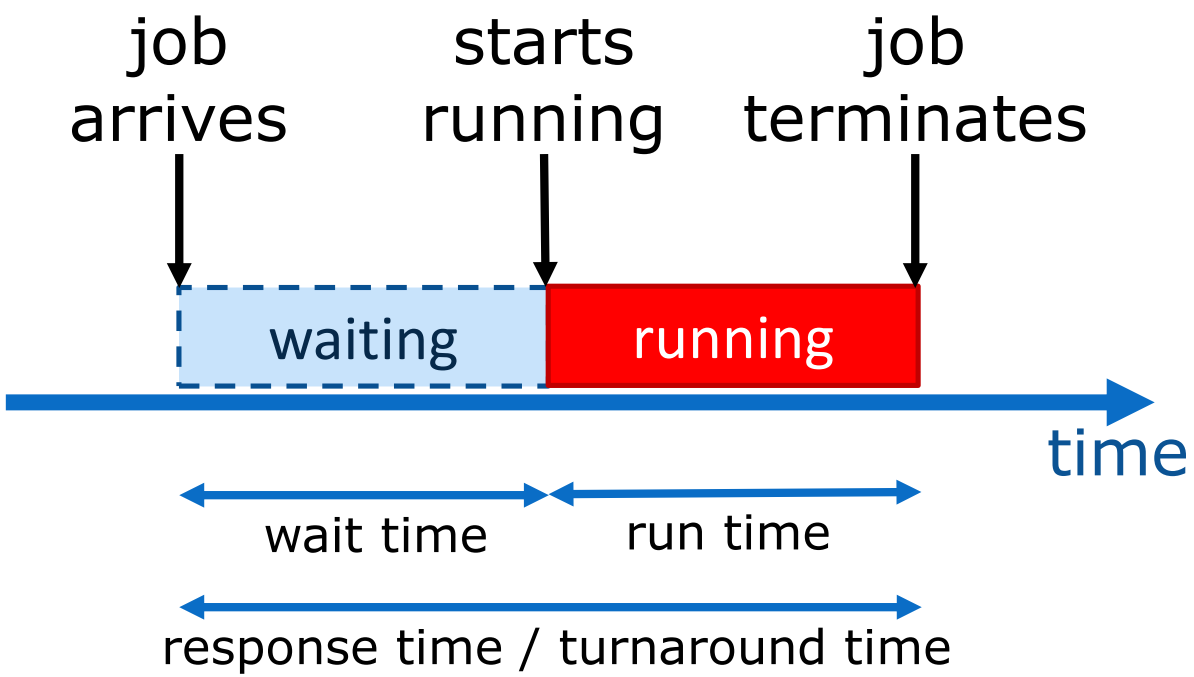


Job Timeline



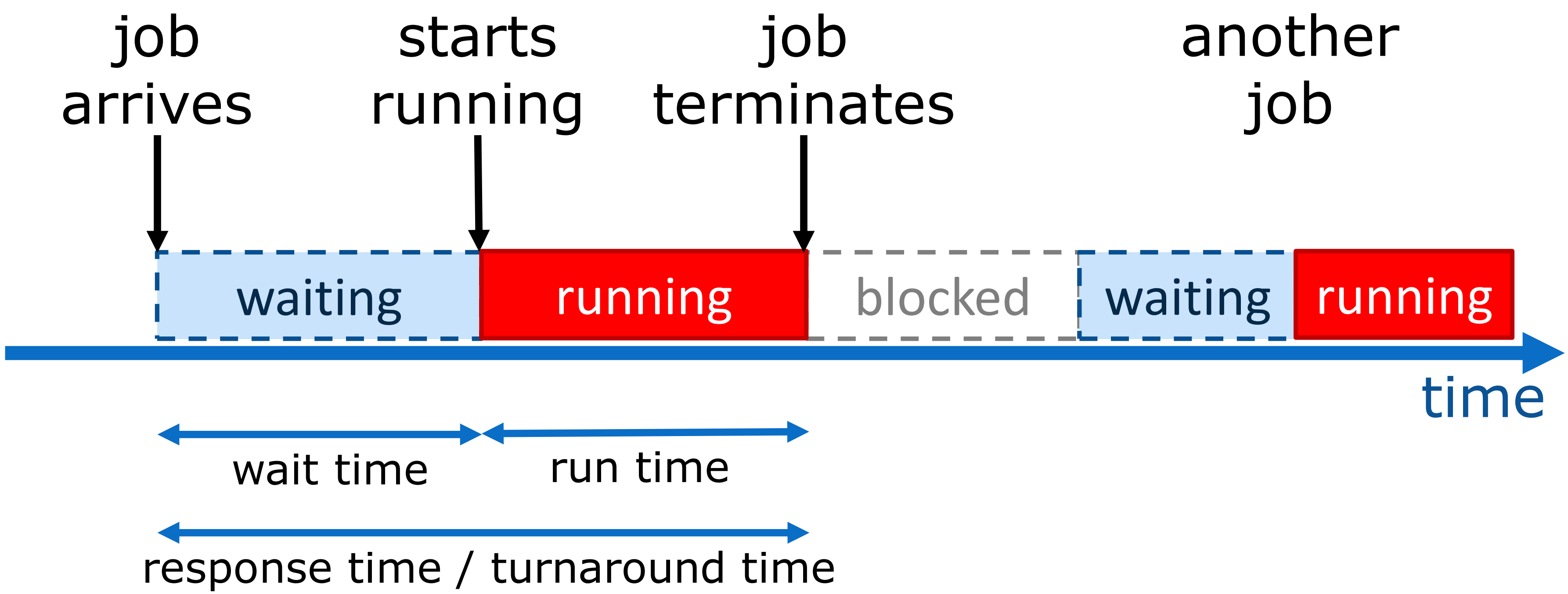
- A process in the "ready" state is waiting for the CPU

Job Timeline



- A process in the “ready” state is waiting for the CPU
- A process in the “running” state is using the CPU

Job Timeline



- Interactive (reactive) programs have multiple CPU bursts – one after each event
- Or perform I/O operations separated by CPU bursts
- We (usually) treat them as independent jobs

Scheduling Framework

A scheduler is an algorithm

Scheduling Framework

A scheduler is an algorithm

- What is the input?

Scheduling Framework

A scheduler is an algorithm

- What is the input?

The jobs to schedule

Scheduling Framework

A scheduler is an algorithm

- What is the input?

The jobs to schedule

- What is the output?

Scheduling Framework

A scheduler is an algorithm

- What is the input?
- What is the output?

The jobs to schedule

A decision which job to run now

Scheduling Framework

A scheduler is an algorithm

- What is the input?
- What is the output?
- What is the objective?

The jobs to schedule

A decision which job to run now

Scheduling Framework

A scheduler is an algorithm

- What is the input?
- What is the output?
- What is the objective?

The jobs to schedule

A decision which job to run now

“Good performance”

Scheduling Framework

A scheduler is an algorithm

- What is the input?
- What is the output?
- What is the objective?
- What are the available actions?

The jobs to schedule

A decision which job to run now

“Good performance”

Scheduling Framework

A scheduler is an algorithm

- What is the input?
- What is the output?
- What is the objective?
- What are the available actions?

The jobs to schedule

A decision which job to run now

“Good performance”

Can we preempt?

Many Different Objectives

Many Different Objectives

- **Low Response Time**: Total time it takes to complete a job (wait time + run time)
  – What users care about
  – Not everything is in the scheduler's control…
- **Low Wait Time**
  – This *is* what the scheduler controls
- **Low Slowdown:** how much slower the system appears to be

(response time) / (run time)

Many Different Objectives

- **High Throughput:**

  (# completed jobs) / (time unit)

  - If the system is stable, this is equal to (# started jobs) / (time unit)
- **High CPU Utilization**: Fraction of time the CPU is busy
  - Not counting overhead

Many Different Objectives

- **High Throughput:**
  (# completed jobs) / (time unit)
  - If the system is stable, this is equal to (# started jobs) / (time unit)
- **High CPU Utilization**: Fraction of time the CPU is busy
  - Not counting overhead
- Actually determined by the arrival process!
- Both limited when system is overloaded

Other Objectives

- **Fairness:**
  - Give users their **fair share**
    - What if they don't need their fair share?
  - Avoid job **starvation**
    - Is this a problem in a stable system?
  - Support user/job **priorities**
    - Who sets the priorities?
- We'll typically assume equal shares and priorities

The Objectives Might Contradict Each Other

- To optimize throughput: run jobs to completion to reduce overhead (time wasted due to context switches)

- To optimize response time: schedule each new job as soon as possible, even if this leads to overhead due to context switching

The Objectives Might Contradict Each Other

- To optimize throughput: run jobs to completion to reduce overhead (time wasted due to context switches)

What are the right objectives for me?

- To optim ... each new job ... his leads to overhead due to context switching

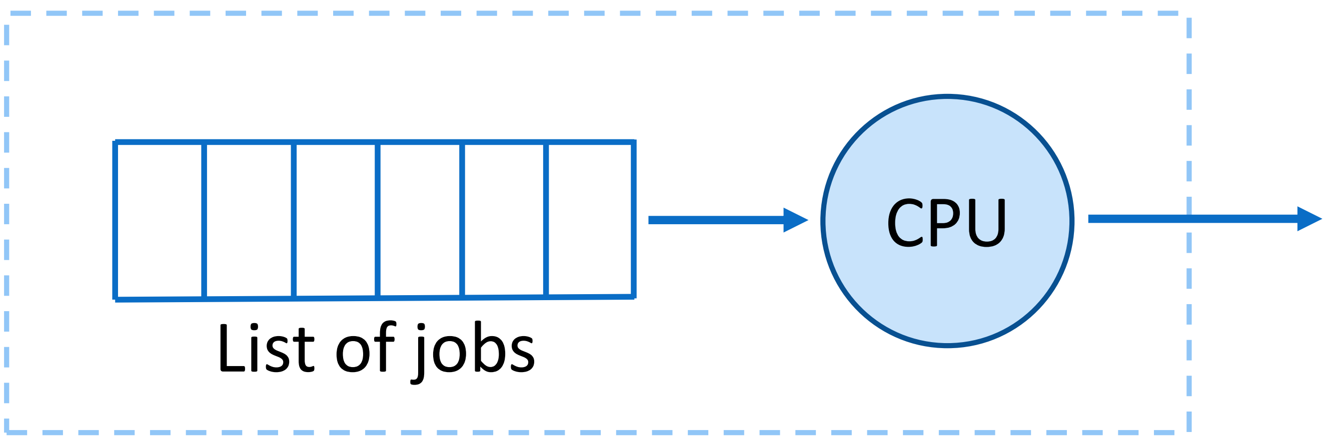
OFF-LINE ALGORITHMS

On-Line vs. Off-Line

- Off-line: get all the input at the outset
  - What you are used to
- On-line: get the input piecemeal
  - At every instant you just get part of the input
  - Need to produce partial/tentative output (make partial tentative decisions) based on your current knowledge of the input
  - May change/regret it later

System Model

- All jobs are given in advance
  - No additional arrivals
- Job runtimes are known in advance



Baseline Scheduler: First Come First Serve (FCFS)

Input:

Job	Runtime
P1	17
P2	2
P3	3

Baseline Scheduler:
First Come First Serve (FCFS)

Schedule:

Job	Runtime	Wait time	Response time
P1	17	0	17
P2	2	17	19
P3	3	19	22

Baseline Scheduler: First Come First Serve (FCFS)

Schedule:

Job	Runtime	Wait time	Response time
P1	17	0	17
P2	2	17	19
P3	3	19	22

Average waiting time: (0+17+19)/3=12
Average response time: (17+19+22)/3=19.33
Throughput: 3 / 22 = 0.136

Baseline Scheduler: First Come First Serve (FCFS)

Schedule:

Job	Runtime	Wait time	Response time
P1	17	0	17
P2	2	17	19

FCFS suffers from a "convoy effect": short jobs get stuck behind long ones

3

0 22

Average waiting time: (0+17+19)/3=12
Average response time: (17+19+22)/3=19.33
Throughput: 3 / 22 = 0.136

How Can We Improve?

How Can We Improve?

Put the short job before the long job!

How Can We Improve?

Put the short job before the long job!

How Can We Improve?

Put the short job before the long job!

Average wait time was: (0+7)/2=3.5
New average wait time: (0+4)/2=2

Average response time was: (7+11)/2=9
New average response time: (4+11)/2=7.5

Throughput was: 2 / 11 = 0.182
New throughput: 2 / 11 = 0.182

How Can We Improve?

Put

NOTE:

This applies **every time** a short job comes after a longer one

Av

New ave

Average response time was: (7+11)/2=9

New average response time: (4+11)/2=7.5

Throughput was: 2 / 11 = 0.182

New throughput: 2 / 11 = 0.182

Improved Scheduler: Shortest Job First (SJF)

Schedule:

Job	Runtime	Wait time	Response time
P2	2	0	2
P3	3	2	5
P1	17	5	22

Improved Scheduler: Shortest Job First (SJF)

Schedule:

Job	Runtime	Wait time	Response time
P2	2	0	2
P3	3	2	5
P1	17	5	22

Average waiting time: (0+2+5)/3=2.33 (was 12)
Average response time: (2+5+22)/3=9.67 (was 19.3)
Throughput: 3 / 22 = 0.136 (same)

SJF Notes

- SJF works in an **offline** setting
- SJF aims to optimize the **average waiting time**

SJF Notes

- SJF works in an **offline** setting
- SJF aims to optimize the **average waiting time**

- **Can we do better?**

SJF Notes

- SJF works in an **offline** setting
- SJF aims to optimize the **average waiting time**

- **Can we do better?** **No. SJF is optimal.**

SJF Notes

- SJF works in an **offline** setting
- SJF aims to optimize the **average waiting time**

- **Can we do better?** **No. SJF is optimal.**
- **Can preemption help?**

SJF Notes

- SJF works in an **offline** setting
- SJF aims to optimize the **average waiting time**

- **Can we do better?** **No. SJF is optimal.**
- **Can preemption help?** **No.**

SJF Notes

- SJF works in an **offline** setting
- SJF aims to optimize the **average waiting time**

- **Can we do better? No. SJF is optimal.**
- **Can preemption help? No.**
- **What about average response time?**

SJF Notes

- SJF works in an **offline** setting
- SJF aims to optimize the **average waiting time**

- **Can we do better?** **No. SJF is optimal.**
- **Can preemption help?** **No.**
- **What about average response time?**

  Also optimal.

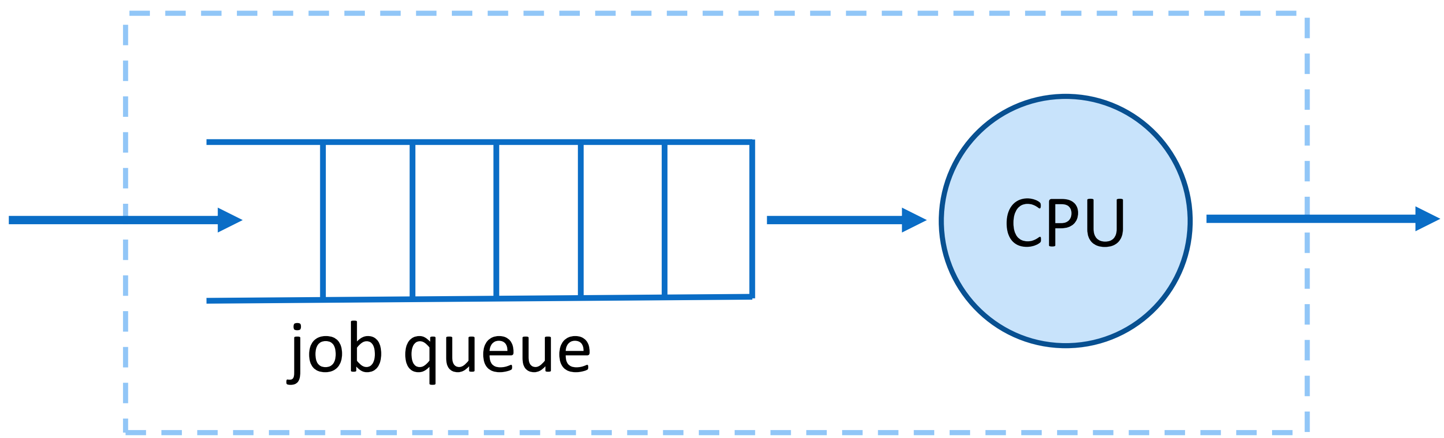
Optimality Proof

1. Assume a schedule S has the optimal (lowest) average wait time
2. If S is *not* SJF, there are a pair of jobs such that $p_i.runtime > p_{i+1}.runtime$
3. Switching these jobs reduces their contribution to the average, without changing anything else
4. Contradiction to assumption that S is optimal

ON-LINE ALGORITHMS

System Model I (simplified)

- Jobs arrive at unknown times new
  - Open system model
- Job runtimes are known in advance
- No preemption



Baseline Scheduler: First Come First Serve (FCFS)

Input:

Job	arrival	Runtime
P1	0	7
P2	2	4
P3	3	2
P4	6	3

Baseline Scheduler: First Come First Serve (FCFS)

Schedule:

Job	Arrival	Runtime
P1	0	7
P2	2	4
P3	3	2
P4	6	3

Wait	Start	End	Response

t=0

Baseline Scheduler:
First Come First Serve (FCFS)

Schedule:

Job	Arrival	Runtime
P1	0	7
P2	2	4
P3	3	2
P4	6	3

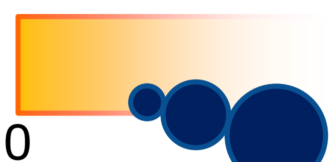
Wait	Start	End	Response
0	0		

t=0

Baseline Scheduler:
First Come First Serve (FCFS)

Schedule:

Job	Arrival	Runtime	Wait	Start	End	Response
P1	0	7	0	0		
P2	2	4				
P3	3	2				
P4	6	3				



t=0

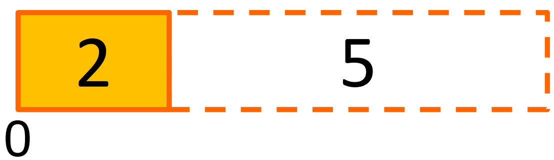
Scheduler will be called whenever "something happens"

1. Job arrival
2. Job termination

Baseline Scheduler: First Come First Serve (FCFS)

Schedule:

Job	Arrival	Runtime	Wait	Start	End	Response
P1	0	7	0	0		
P2	2	4				
P3	3	2				
P4	6	3				



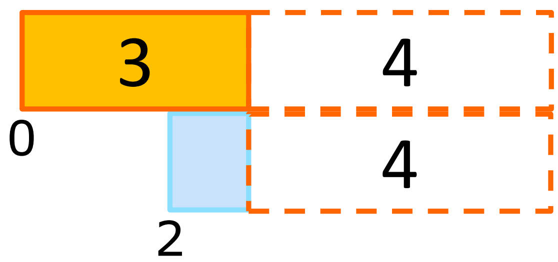
t=2

Baseline Scheduler: First Come First Serve (FCFS)

Schedule:

Job	Arrival	Runtime
P1	0	7
P2	2	4
P3	3	2
P4	6	3

Wait	Start	End	Response
0	0		
1			



t=3

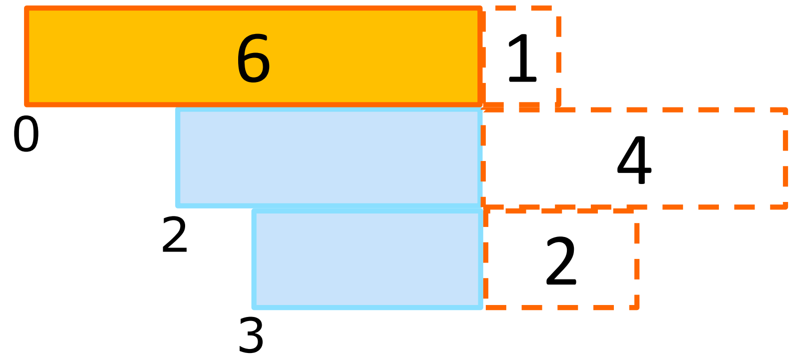
Baseline Scheduler: First Come First Serve (FCFS)

Schedule:

Job	Arrival	Runtime
P1	0	7
P2	2	4
P3	3	2
P4	6	3

Wait	Start	End	Response
0	0		
4			
3			

t=6

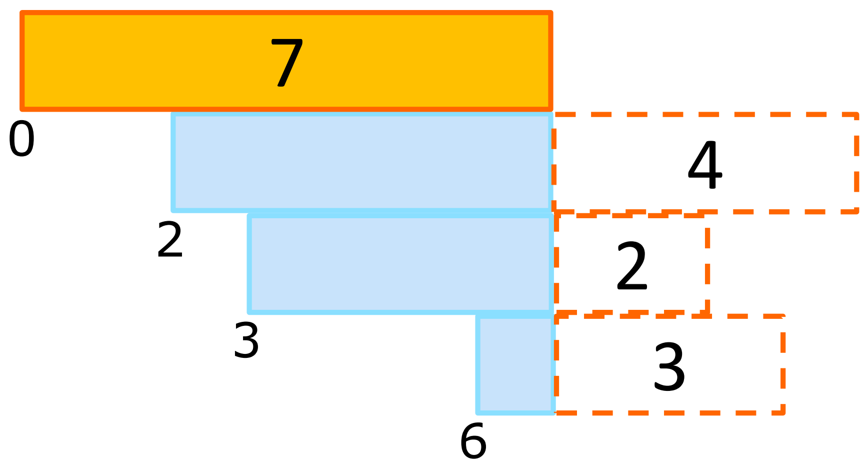


Baseline Scheduler: First Come First Serve (FCFS)

Schedule:

Job	Arrival	Runtime	Wait	Start	End	Response
P1	0	7	0	0	7	7
P2	2	4	5	7		
P3	3	2	4			
P4	6	3	1			

t=7

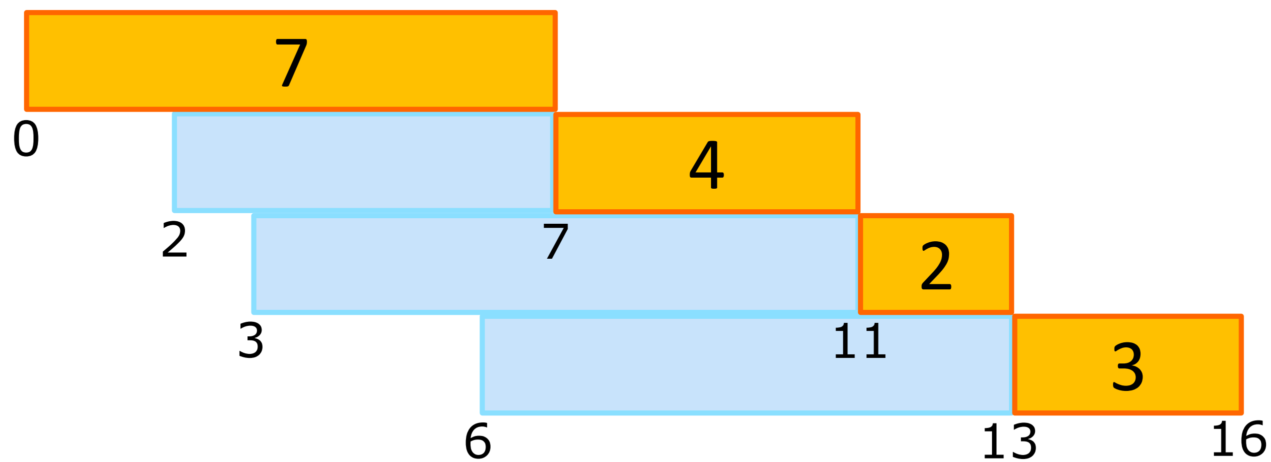


Baseline Scheduler: First Come First Serve (FCFS)

Schedule:

Job	Arrival	Runtime	Wait	Start	End	Response
P1	0	7	0	0	7	7
P2	2	4	5	7	11	9
P3	3	2	8	11	13	10
P4	6	3	7	13	16	10

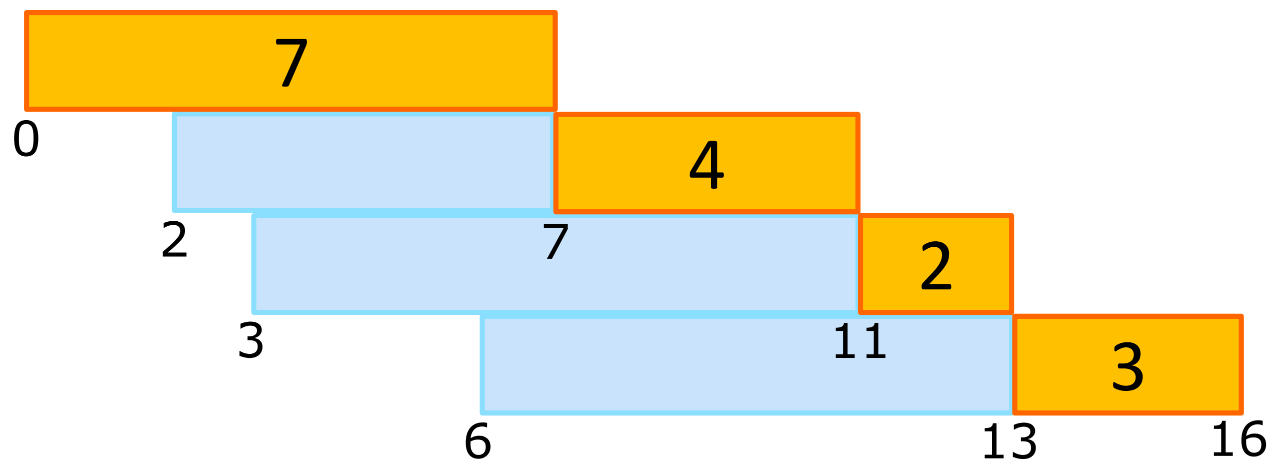
t=16



Baseline Scheduler: First Come First Serve (FCFS)

Schedule:

Job	Arrival	Runtime	Wait	Start	End	Response
P1	0	7	0	0	7	7
P2	2	4	5	7	11	9
P3	3	2	8	11	13	10
P4	6	3	7	13	16	10

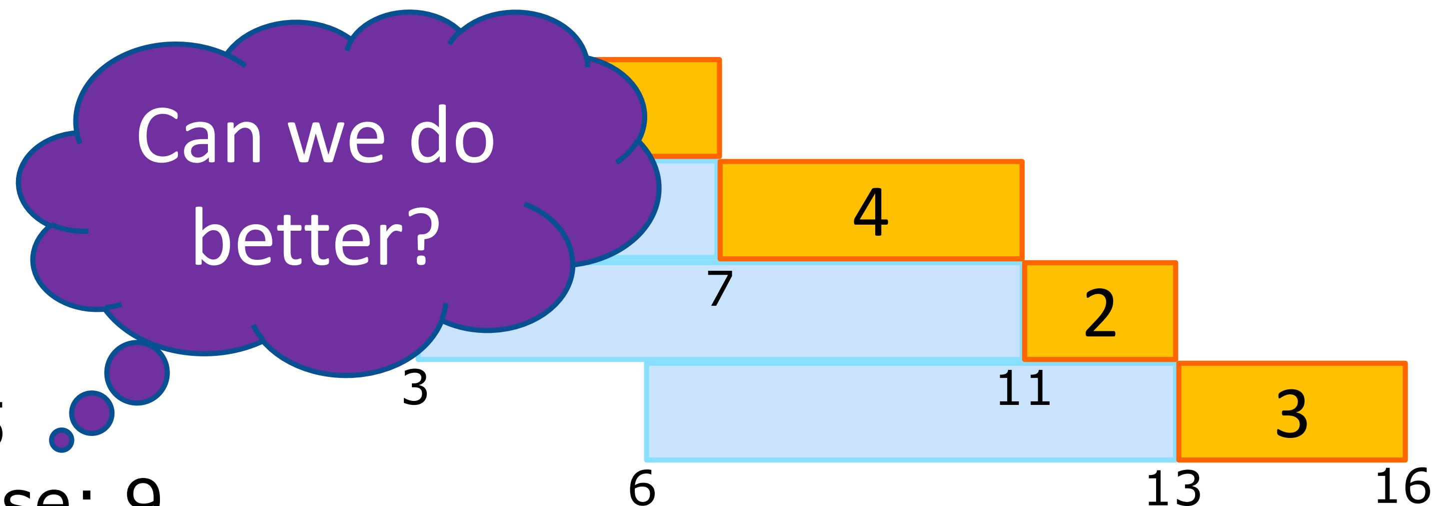


Avg. wait: 5
Avg. response: 9
Throughput: 0.25

Baseline Scheduler: First Come First Serve (FCFS)

Schedule:

Job	Arrival	Runtime	Wait	Start	End	Response
P1	0	7	0	0	7	7
P2	2	4	5	7	11	9
P3	3	2	8	11	13	10
P4	6	3	7	13	16	10



Avg. wait: 5
Avg. response: 9
Throughput: 0.25

Improved Scheduler: Shortest Job First (SJF)

Schedule:

Job	Arrival	Runtime
P1	0	7
P2	2	4
P3	3	2
P4	6	3

Wait	Start	End	Response

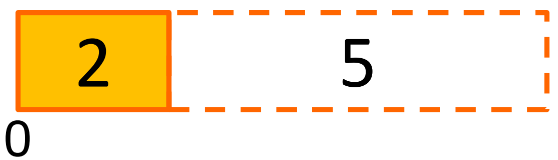
t=0

Improved Scheduler: Shortest Job First (SJF)

Schedule:

Job	Arrival	Runtime
P1	0	7
P2	2	4
P3	3	2
P4	6	3

Wait	Start	End	Response
0	0		

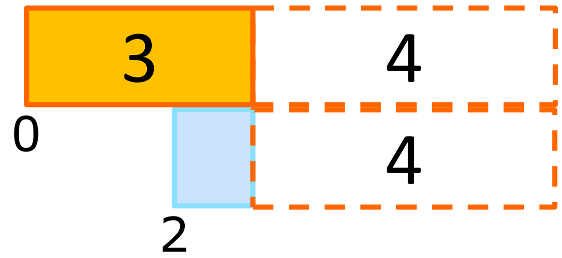


t=2

Improved Scheduler: Shortest Job First (SJF)

Schedule:

Job	Arrival	Runtime	Wait	Start	End	Response
P1	0	7	0	0		
P2	2	4	1			
P3	3	2				
P4	6	3				



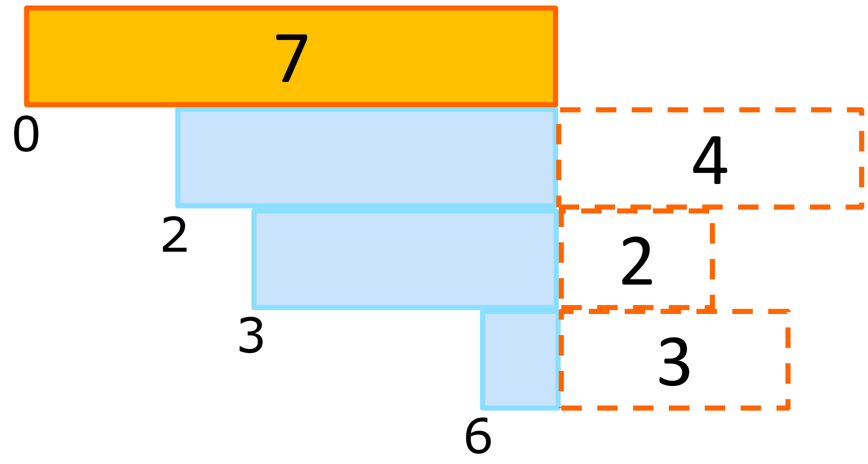
t=3

Improved Scheduler: Shortest Job First (SJF)

Schedule:

Job	Arrival	Runtime	Wait	Start	End	Response
P1	0	7	0	0	7	7
P2	2	4	5			
P3	3	2	4			
P4	6	3	1			

t=7

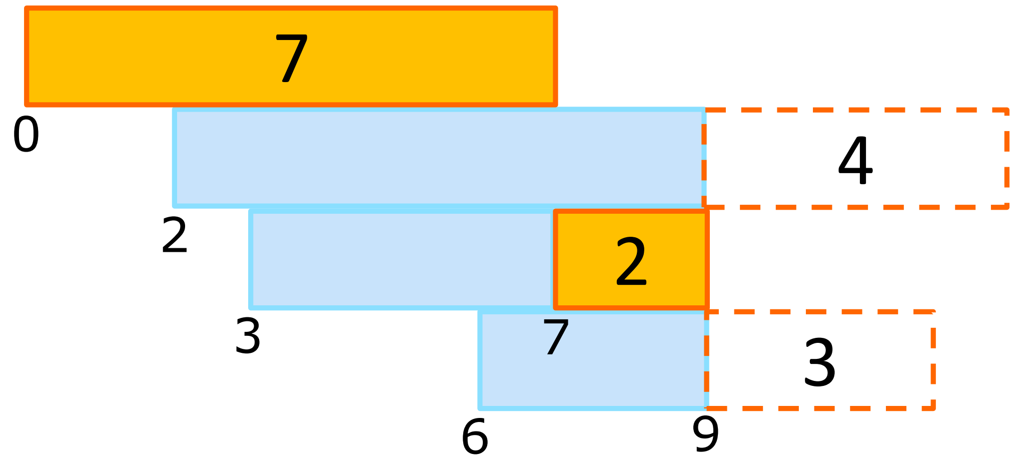


Improved Scheduler: Shortest Job First (SJF)

Schedule:

Job	Arrival	Runtime	Wait	Start	End	Response
P1	0	7	0	0	7	7
P2	2	4	7			
P3	3	2	4	7	9	6
P4	6	3	3			

t=9

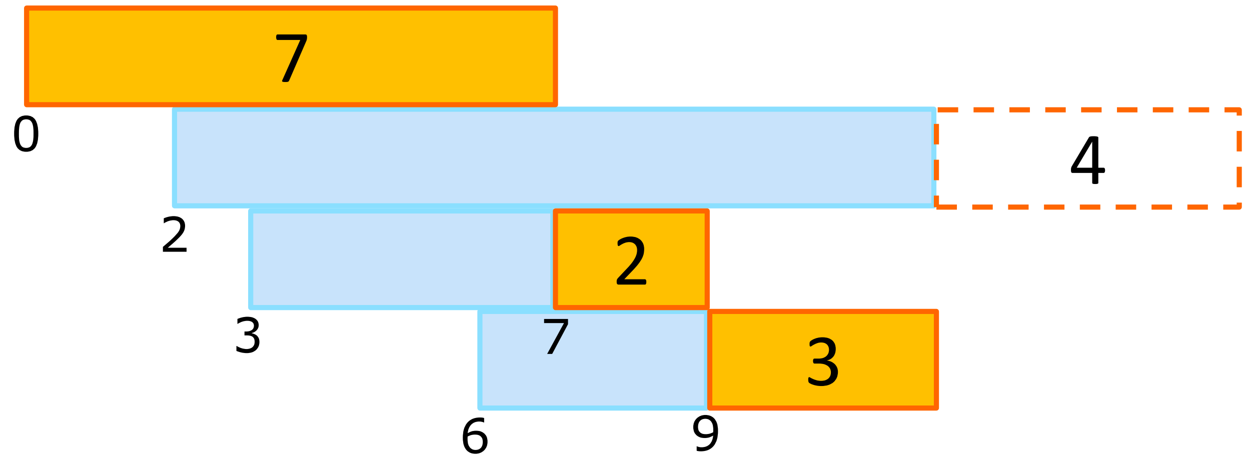


Improved Scheduler: Shortest Job First (SJF)

Schedule:

Job	Arrival	Runtime	Wait	Start	End	Response
P1	0	7	0	0	7	7
P2	2	4	10			
P3	3	2	4	7	9	6
P4	6	3	3	9	12	6

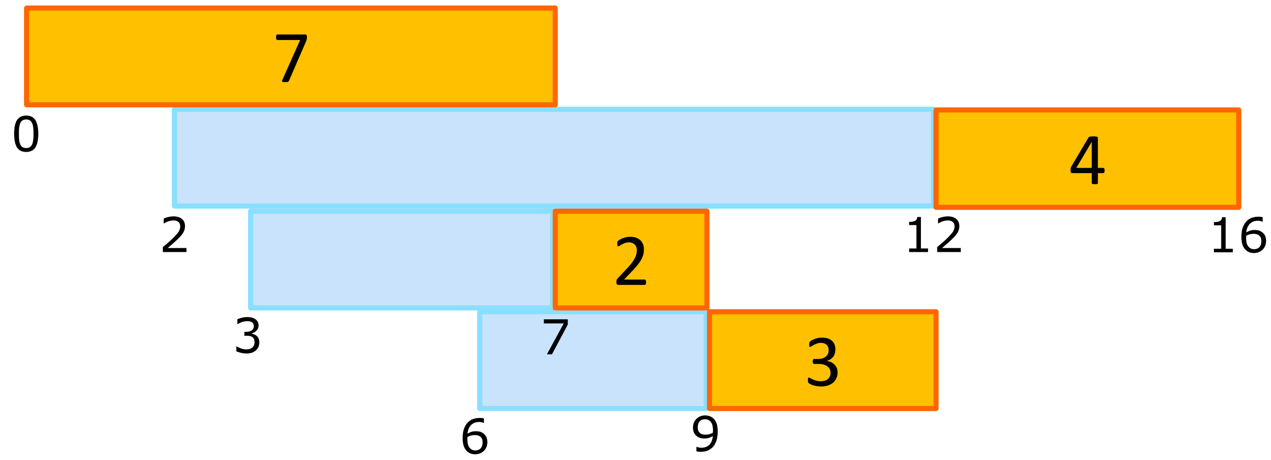
t=12



Improved Scheduler: Shortest Job First (SJF)

Schedule:

Job	Arrival	Runtime	Wait	Start	End	Response
P1	0	7	0	0	7	7
P2	2	4	10	12	16	14
P3	3	2	4	7	9	6
P4	6	3	3	9	12	6

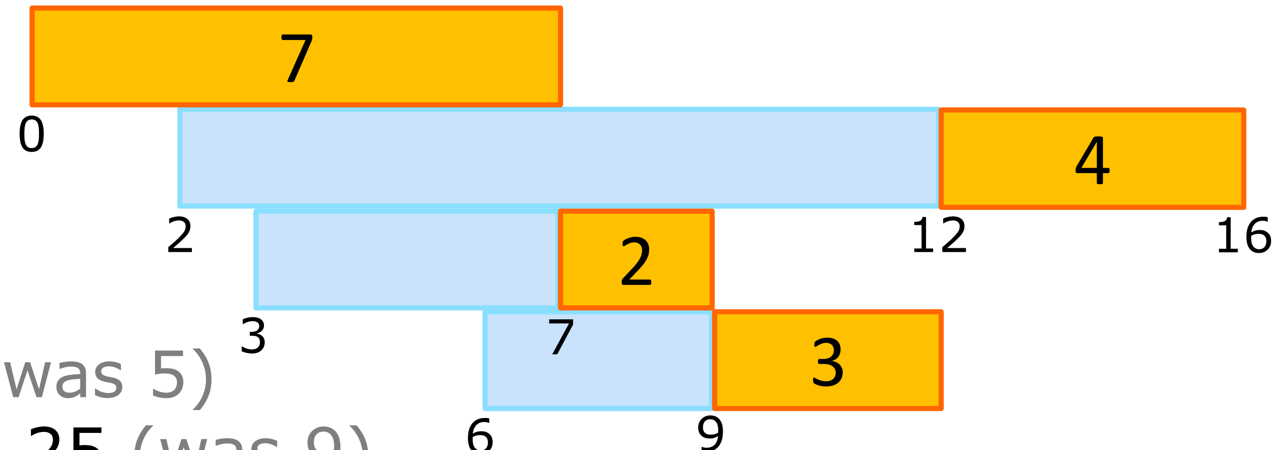


Improved Scheduler: Shortest Job First (SJF)

Schedule:

Job	Arrival	Runtime
P1	0	7
P2	2	4
P3	3	2
P4	6	3

Wait	Start	End	Response
0	0	7	7
10	12	16	14
4	7	9	6
3	9	12	6



Avg. wait: 4.25 (was 5)
Avg. response: 8.25 (was 9)
Throughput: 0.25 (same)

Priority Scheduling

- Many scheduling algorithms can be interpreted as **priority scheduling** algorithms:
  1. Assign each job a priority
  2. Schedule the highest priority job

Priority Scheduling

- Many scheduling algorithms can be interpreted as **priority scheduling** algorithms:
  1. Assign each job a priority
  2. Schedule the highest priority job
- In FCFS:

Priority Scheduling

- Many scheduling algorithms can be interpreted as **priority scheduling** algorithms:
  1. Assign each job a priority
  2. Schedule the highest priority job
- In FCFS: priority = time since arrival

Priority Scheduling

- Many scheduling algorithms can be interpreted as **priority scheduling** algorithms:
  1. Assign each job a priority
  2. Schedule the highest priority job
- In FCFS: priority = time since arrival
- In SJF:

Priority Scheduling

- Many scheduling algorithms can be interpreted as **priority scheduling** algorithms:
    1. Assign each job a priority
    2. Schedule the highest priority job
- In FCFS: priority = time since arrival
- In SJF: priority = runtime (inverted)

SJF had a Problem

- An on-line algorithm does not know the future
- When P1 arrived it was scheduled
- But then the scheduler was stuck till P1 terminated
- Even when shorter jobs became available!

The Solution

- Use preemption!
- Revert earlier decisions that turn out to be sub-optimal

The Solution

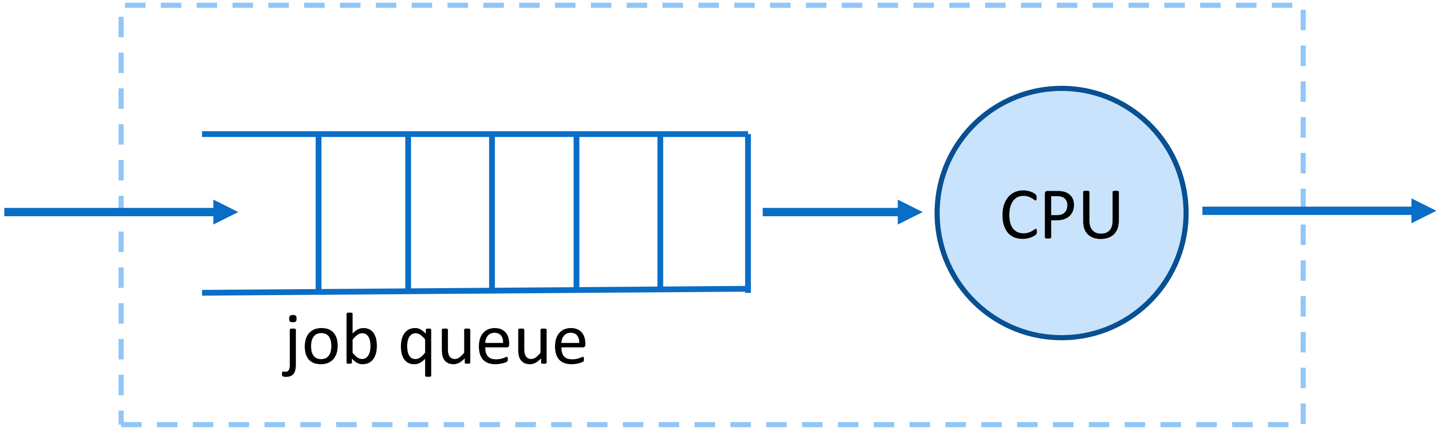
- Use preemption!
- Revert earlier decisions that turn out to be sub-optimal
- *Compensate for not knowing the future*

The Solution

- Use preemption!
- Revert earlier decisions that turn out to be sub-optimal
- *Compensate for not knowing the future*
- The cost: context switching overhead

System Model II (simplified)

- Jobs arrive at unknown times
  - Open system model
- Job runtimes are known in advance
- We can use preemption 



Preemptive SJF: Shortest Remaining Processing Time (SRPT)

Schedule:

Job	Arrival	Runtime
P1	0	7
P2	2	4
P3	3	2
P4	6	3

Wait	Start	End	Response

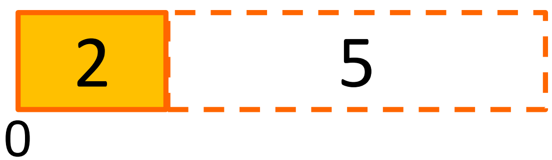
t=0

Preemptive SJF: Shortest Remaining Processing Time (SRPT)

Schedule:

Job	Arrival	Runtime
P1	0	7
P2	2	4
P3	3	2
P4	6	3

Wait	Start	End	Response
0	0		

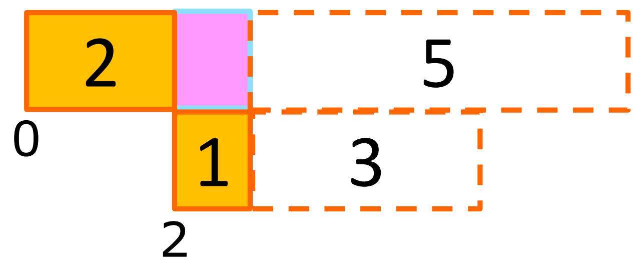


t=2

Preemptive SJF: Shortest Remaining Processing Time (SRPT)

Schedule:

Job	Arrival	Runtime	Wait	Start	End	Response
P1	0	7	(1)	0		
P2	2	4	0	2		
P3	3	2				
P4	6	3				



t=3

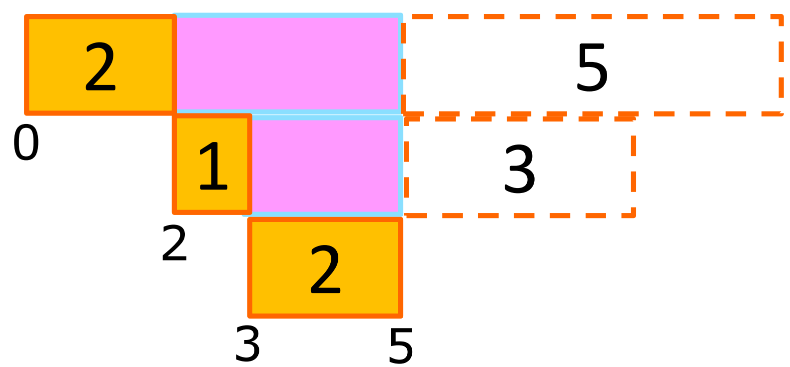
Preemptive SJF: Shortest Remaining Processing Time (SRPT)

Schedule:

Job	Arrival	Runtime
P1	0	7
P2	2	4
P3	3	2
P4	6	3

Wait	Start	End	Response
(3)	0		
(2)	2		
0	3	5	2

t=5



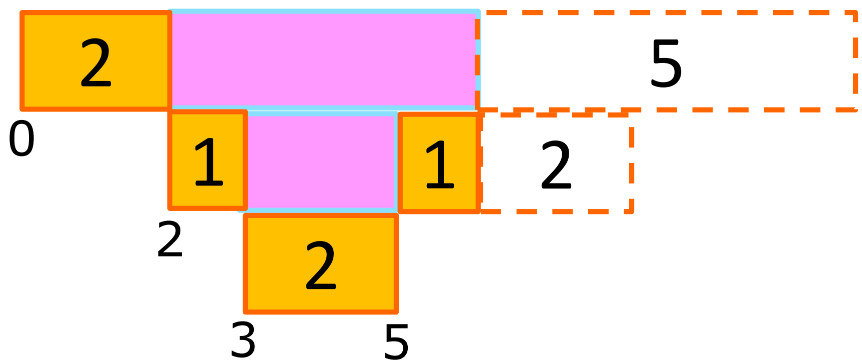
Preemptive SJF: Shortest Remaining Processing Time (SRPT)

Schedule:

Job	Arrival	Runtime
P1	0	7
P2	2	4
P3	3	2
P4	6	3

Wait	Start	End	Response
(4)	0		
(2)	2		
0	3	5	2

t=6

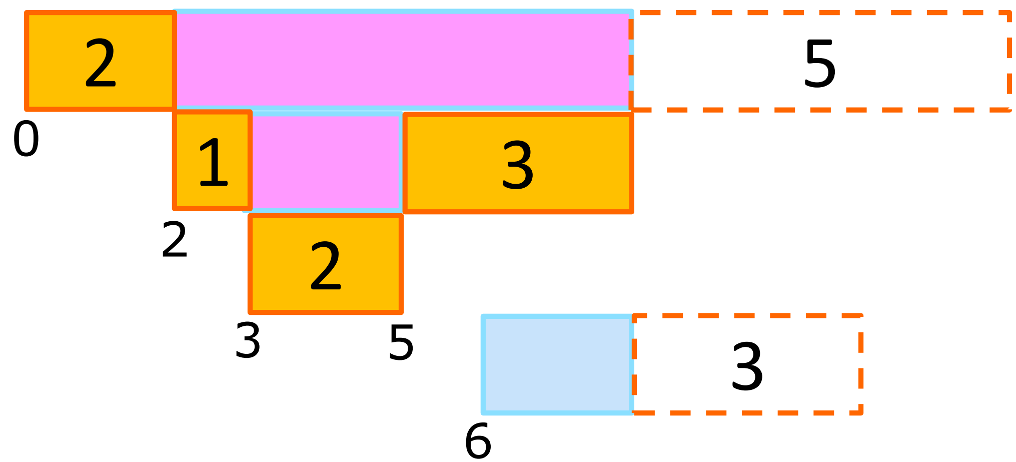


Preemptive SJF: Shortest Remaining Processing Time (SRPT)

Schedule:

Job	Arrival	Runtime	Wait	Start	End	Response
P1	0	7	(6)	0		
P2	2	4	(2)	2	8	6
P3	3	2	0	3	5	2
P4	6	3	2			

t=8

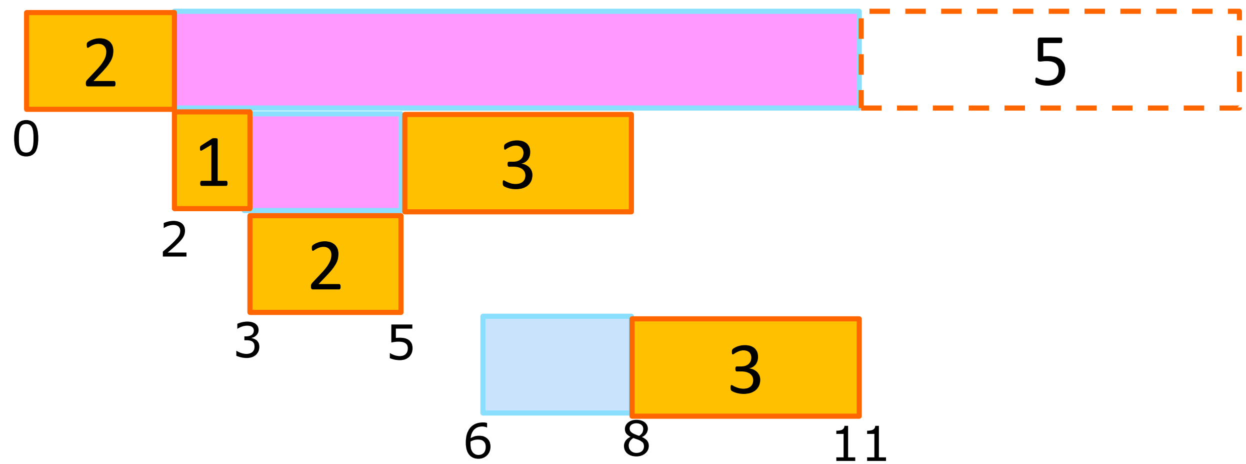


Preemptive SJF: Shortest Remaining Processing Time (SRPT)

Schedule:

Job	Arrival	Runtime	Wait	Start	End	Response
P1	0	7	(9)	0		
P2	2	4	(2)	2	8	6
P3	3	2	0	3	5	2
P4	6	3	2	8	11	5

t=11

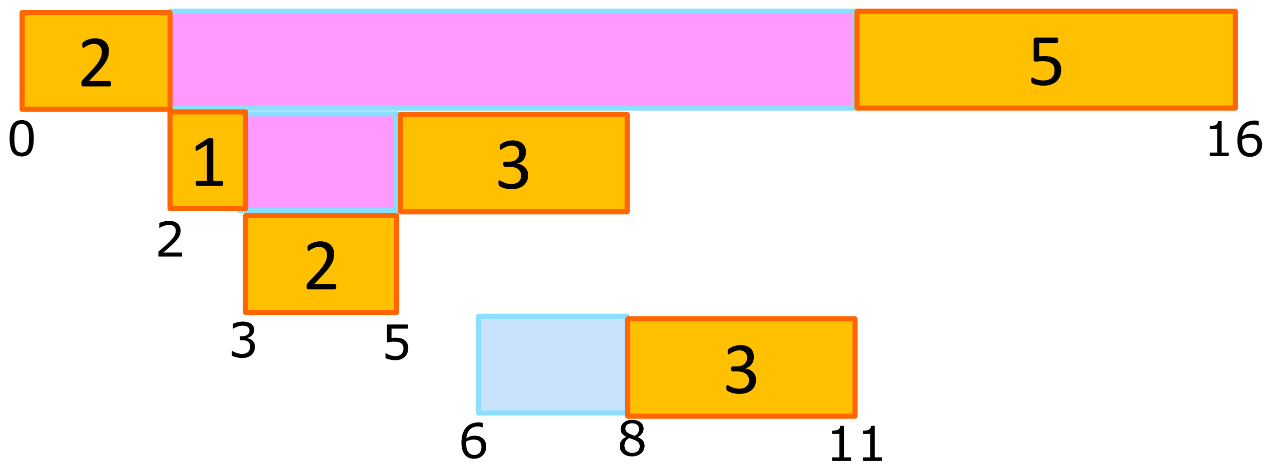


Preemptive SJF: Shortest Remaining Processing Time (SRPT)

Schedule:

Job	Arrival	Runtime	Wait	Start	End	Response
P1	0	7	(9)	0	16	16
P2	2	4	(2)	2	8	6
P3	3	2	0	3	5	2
P4	6	3	2	8	11	5

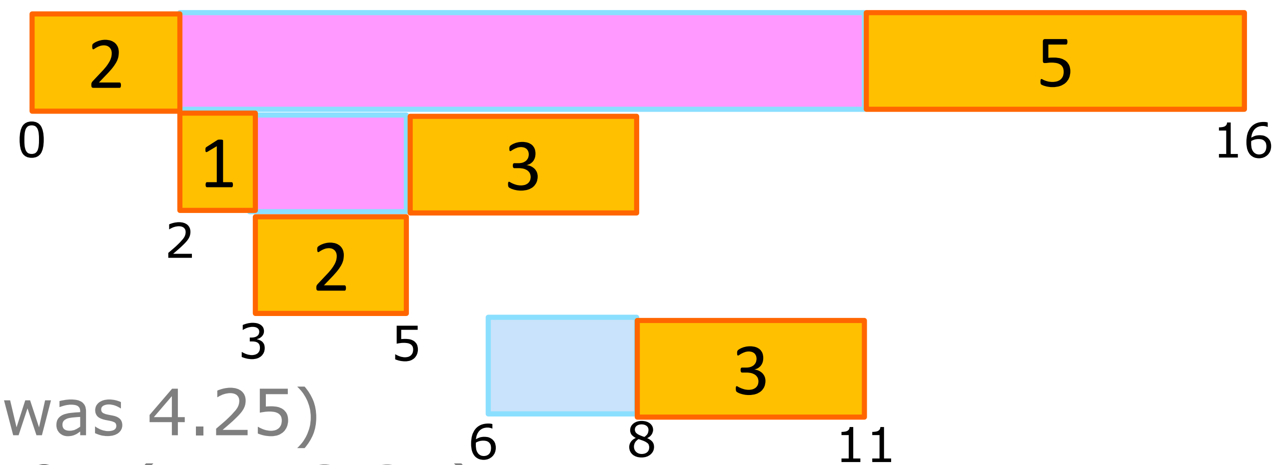
t=16



Preemptive SJF: Shortest Remaining Processing Time (SRPT)

Schedule:

Job	Arrival	Runtime	Wait	Start	End	Response
P1	0	7	(9)	0	16	16
P2	2	4	(2)	2	8	6
P3	3	2	0	3	5	2
P4	6	3	2	8	11	5

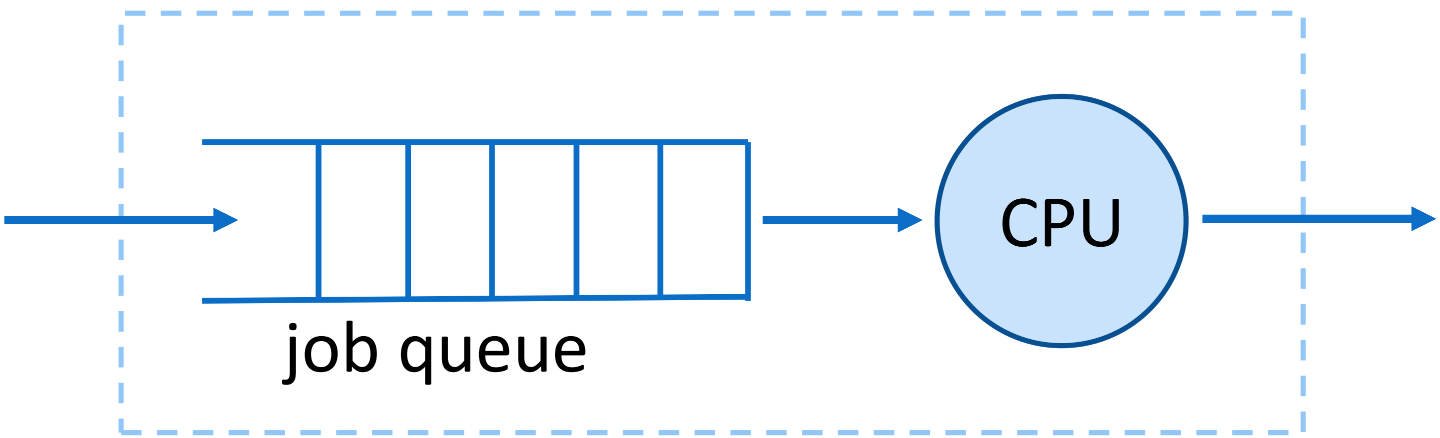


Avg. wait: 3.25 (was 4.25)
Avg. response: 7.25 (was 8.25)
Throughput: 0.25 (same)

But what if we don’t know job runtimes in advance?

System Model III (realistic)

- Jobs arrive at unknown times
  - Open system model
- Job runtimes **not** known in advance new
- We can use preemption



The Problem

- We don’t know the future
- When a new job arrives, we don’t know whether it will be short or long
- So we don’t know whether to preempt and let it run
- Short jobs can get stuck after long jobs

The Problem

- We don’t know the future
- When a new job arrives, we don’t know whether it will be short or long
- So we don’t know whether to preempt and let it run
- Short jobs can get stuck after long jobs

Possible Solution: Estimate

- Assume job has multiple CPU bursts
- Use exponentially weighted average of previous bursts to estimate length of next burst

  1. t_n = actual length of the n^{th} CPU burst
  2. τ_{n+1} = predicted value for the next CPU burst
  3. α = weight factor, $0 \leq \alpha \leq 1$
  4. Define: $\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\, \tau_n \quad [\tau_1 = 0]$

  Result: $\tau_{n+1} = \sum_{i=1}^{n} (1-\alpha)^{n-i} \alpha \boldsymbol{t}_i$

- Might not give good results
- Lots of overhead

Alternative Partial Solution

- Assume jobs can run together!
- Then the short jobs won't get stuck

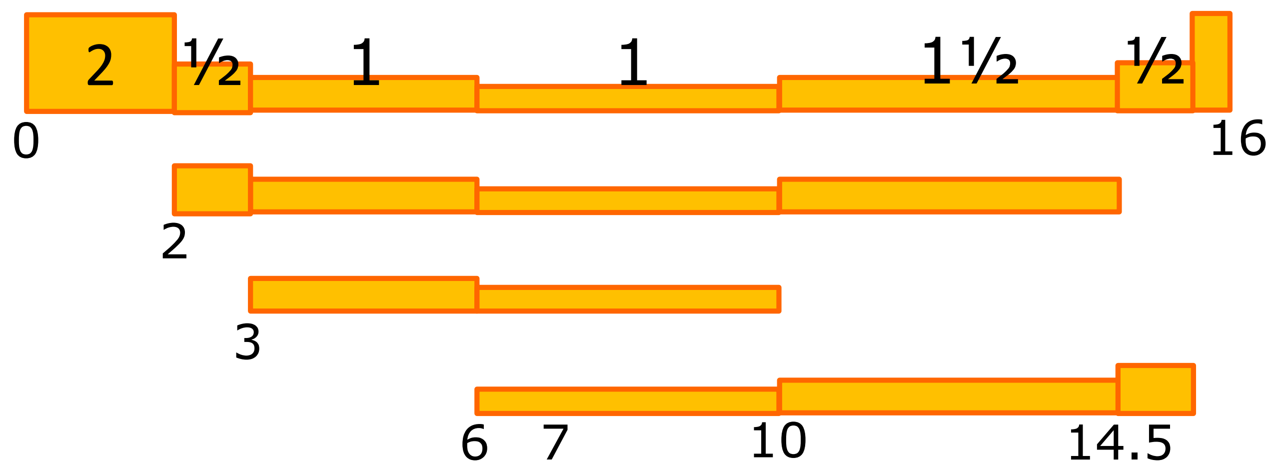
This is called "processor sharing"

When *k* jobs are present, they all advance at a rate of *1/k*

Processor Sharing (PS)

Schedule:

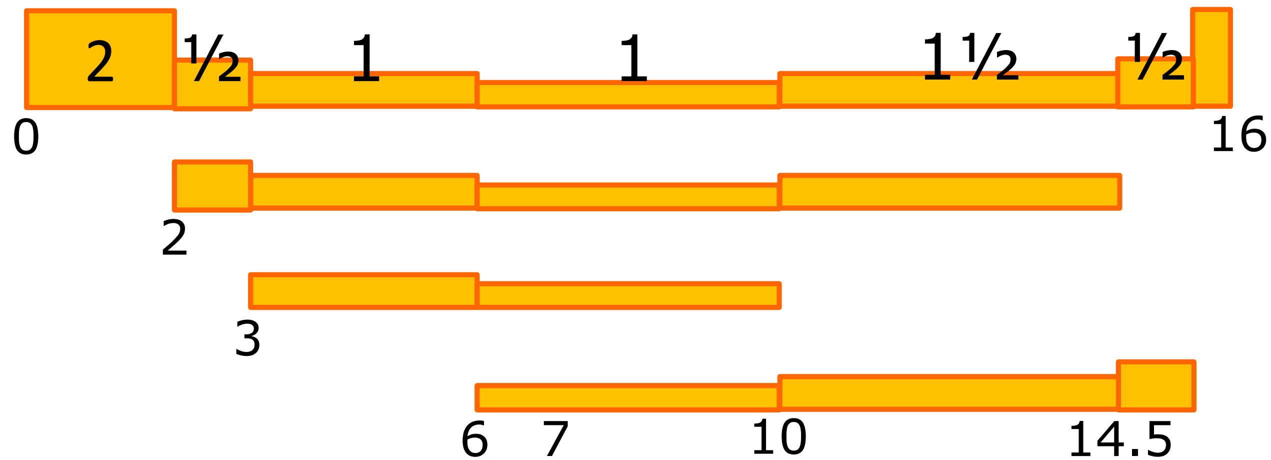
Job	Arrival	Runtime	Wait	Start	End	Response
P1	0	7	0	0	16	16
P2	2	4	0	2	14.5	12.5
P3	3	2	0	3	10	7
P4	6	3	0	6	15.5	9.5



Processor Sharing (PS)

Schedule:

Job	Arrival	Runtime	Wait	Start	End	Response
P1	0	7	0	0	16	16
P2	2	4	0	2	14.5	12.5
P3	3	2	0	3	10	7
P4	6	3	0	6	15.5	9.5



Avg. wait: undef
Avg. response: 11.25 (SRPT was 7.25)
Throughput: 0.25

Processor Sharing (PS)

- Good: short jobs don't get stuck
- Bad: everyone runs at slower rate
- So is it really beneficial?

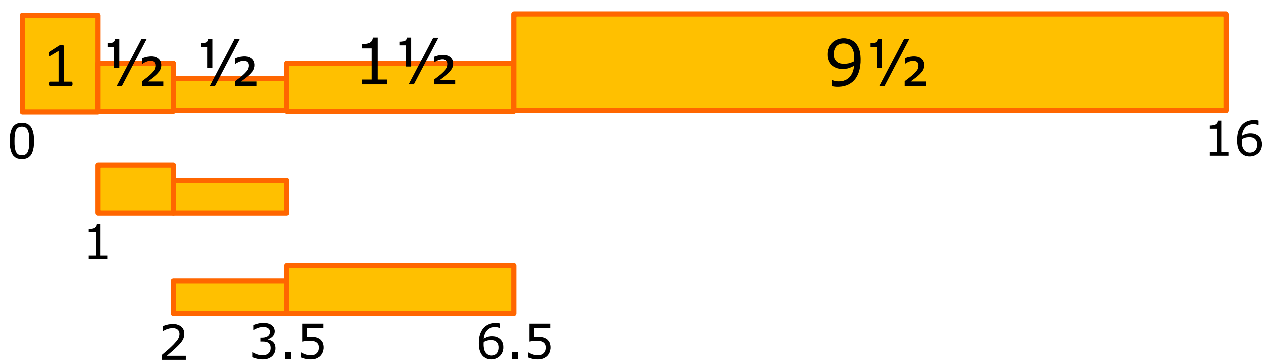
Processor Sharing (PS)

- Good: short jobs don’t get stuck
- Bad: everyone runs at slower rate
- So is it really beneficial?
- Depends on *workload statistics*
- Specifically on distribution of job lengths

Good Case

Schedule:

Job	Arrival	Runtime	Wait	Start	End	Response
P1	0	13	0	0	16	16
P2	1	1	0	1	3.5	2.5
P3	2	2	0	2	6.5	4.5

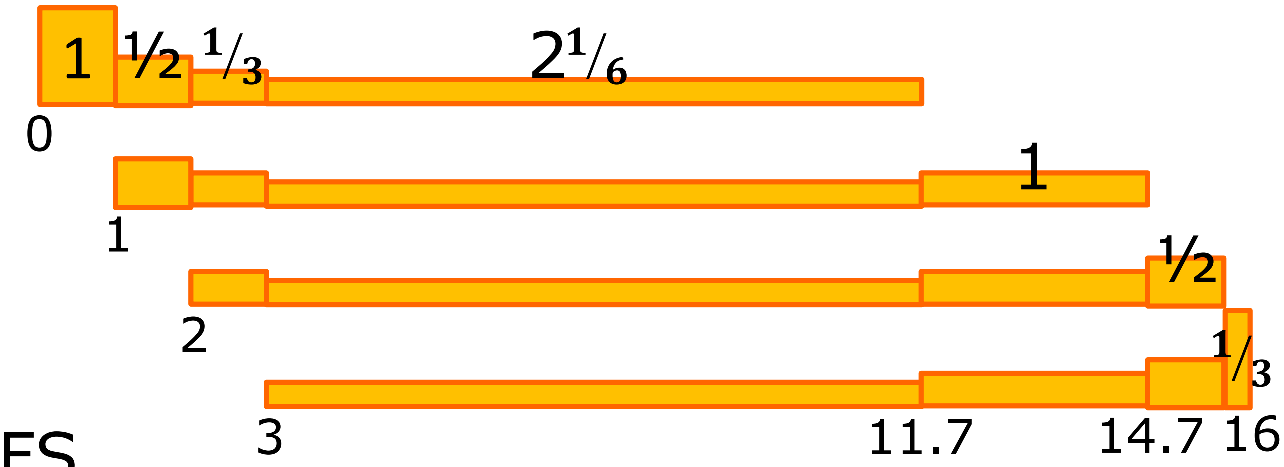


Approximates SRPT!

Bad Case

Schedule:

Job	Arrival	Runtime	Wait	Start	End	Response
P1	0	4	0	0	11.7	11.7
P2	1	4	0	1	14.7	13.7
P3	2	4	0	2	15.7	13.2
P4	3	4	0	3	16	13



Worse than FCFS…

Theory

- Processor sharing is good for **skewed** distribution
  - Many small values
  - Few very large values
- Specifically when **CV > 1**
- Coefficient of variation definition:

$$\mathrm{CV} = \frac{\mathrm{std\_dev}}{\mathrm{mean}}$$

Data

- Most processes are indeed very short
- Some are very long
- The distribution has a Pareto tail

$$\Pr(r > t) = 1/t$$

- Caveat: there is little data, and different systems are probably very different
- So indeed processor sharing should be good

Data

- Most processes are indeed very short
- Some are very long
- The distribution has a Pareto tail

$$\Pr(r > t) = 1/t$$

- Caveat: there is little data, and different systems are probably very different
- So indeed processor sharing should be good
- Too bad it can't be done…

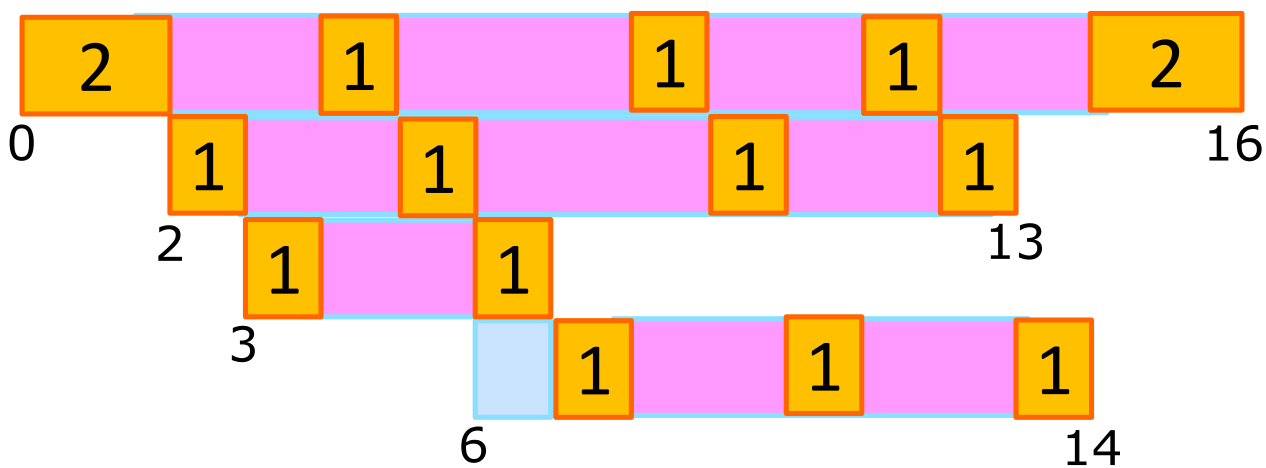
Round-Robin (RR) Scheduling

- Approximation of processor sharing
- Run each process for a certain **time quantum**
  - Predefined short time
  - E.g. around 10-100 ms
- Then preempt and move to back of queue

Round-Robin (RR) Scheduling

Schedule:

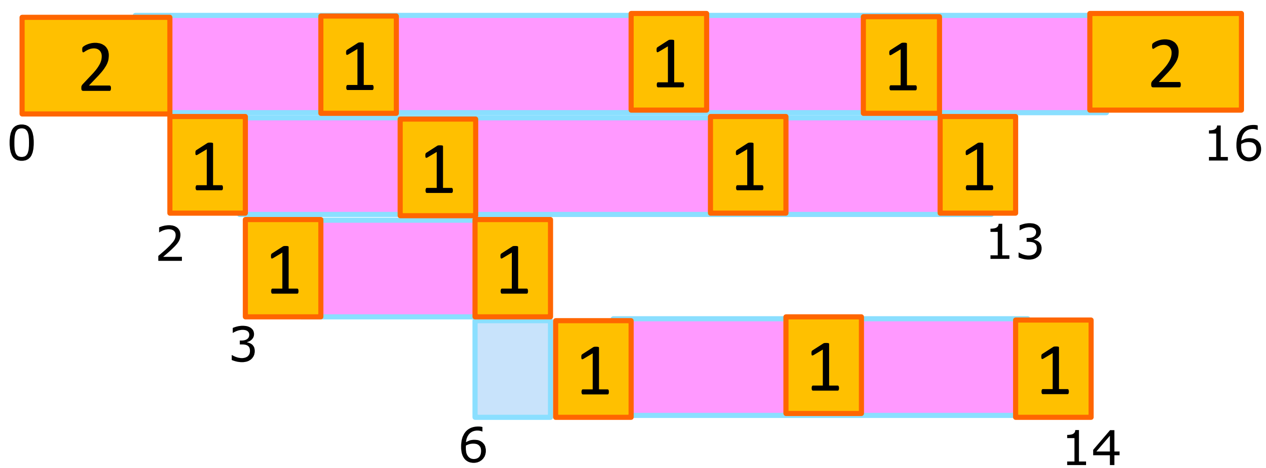
Job	Arrival	Runtime	Wait	Start	End	Response
P1	0	7	(9)	0	16	16
P2	2	4	(7)	2	13	11
P3	3	2	(2)	6	7	4
P4	6	3	(5)	7	14	7



Round-Robin (RR) Scheduling

Schedule:

Job	Arrival	Runtime	Wait	Start	End	Response
P1	0	7	(9)	0	16	16
P2	2	4	(7)	2	13	11
P3	3	2	(2)	6	7	4
P4	6	3	(5)	7	14	7



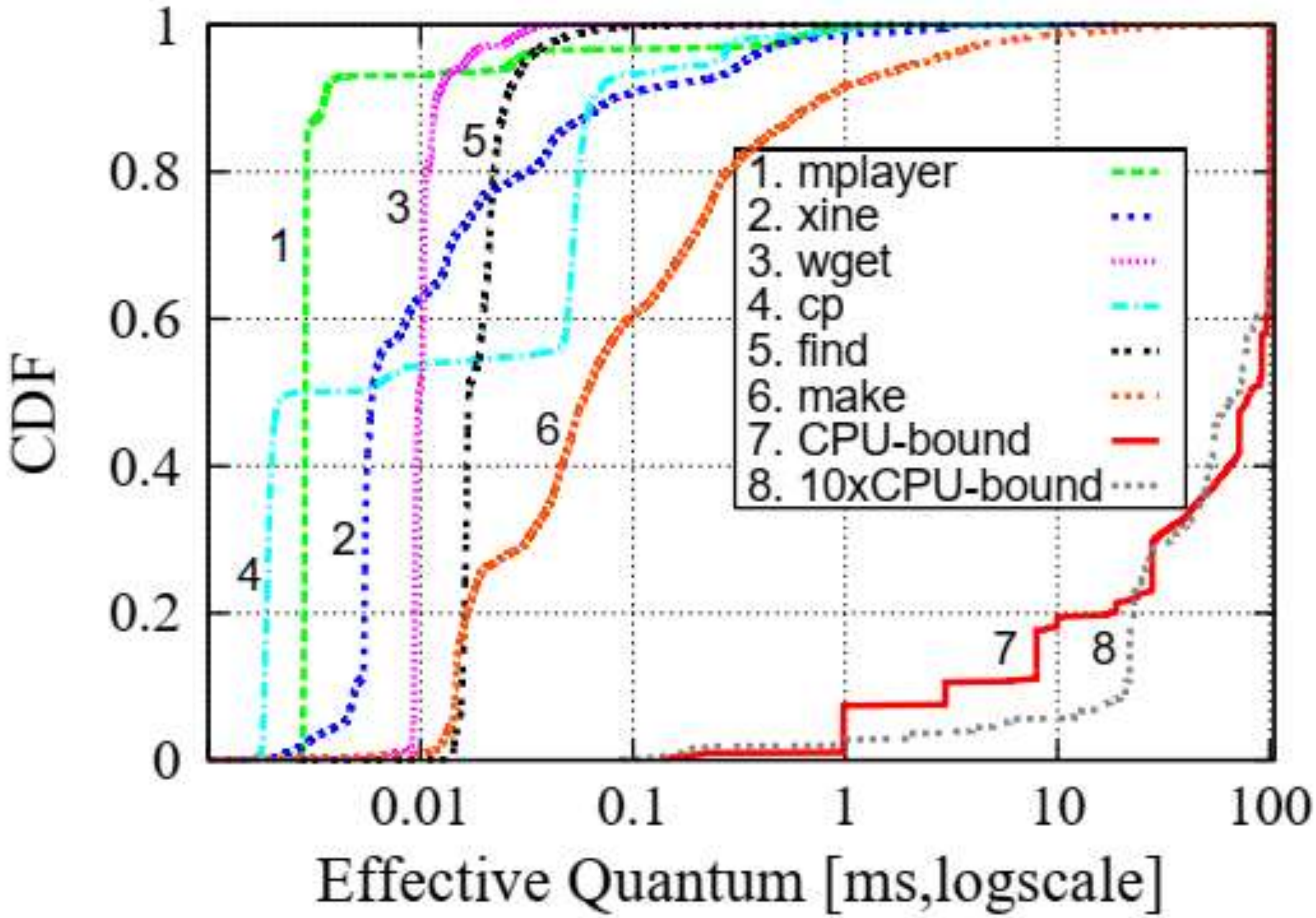
Avg. wait: 5.75
Avg. response: 9.5 (SRPT was 7.25)
Throughput: 0.25

RR Time Quantum

- With n jobs and quantum q no job waits more than $(n-1)q$
  - Shorter q → shorter wait
- If context switch takes c time, overhead will be $c/(q+c)$
  - Shorter q → more overhead
- If q is long, bursts will finish before end of quantum
  - Leads to behavior like FCFS
  - May be OK: only want to break ***long*** bursts

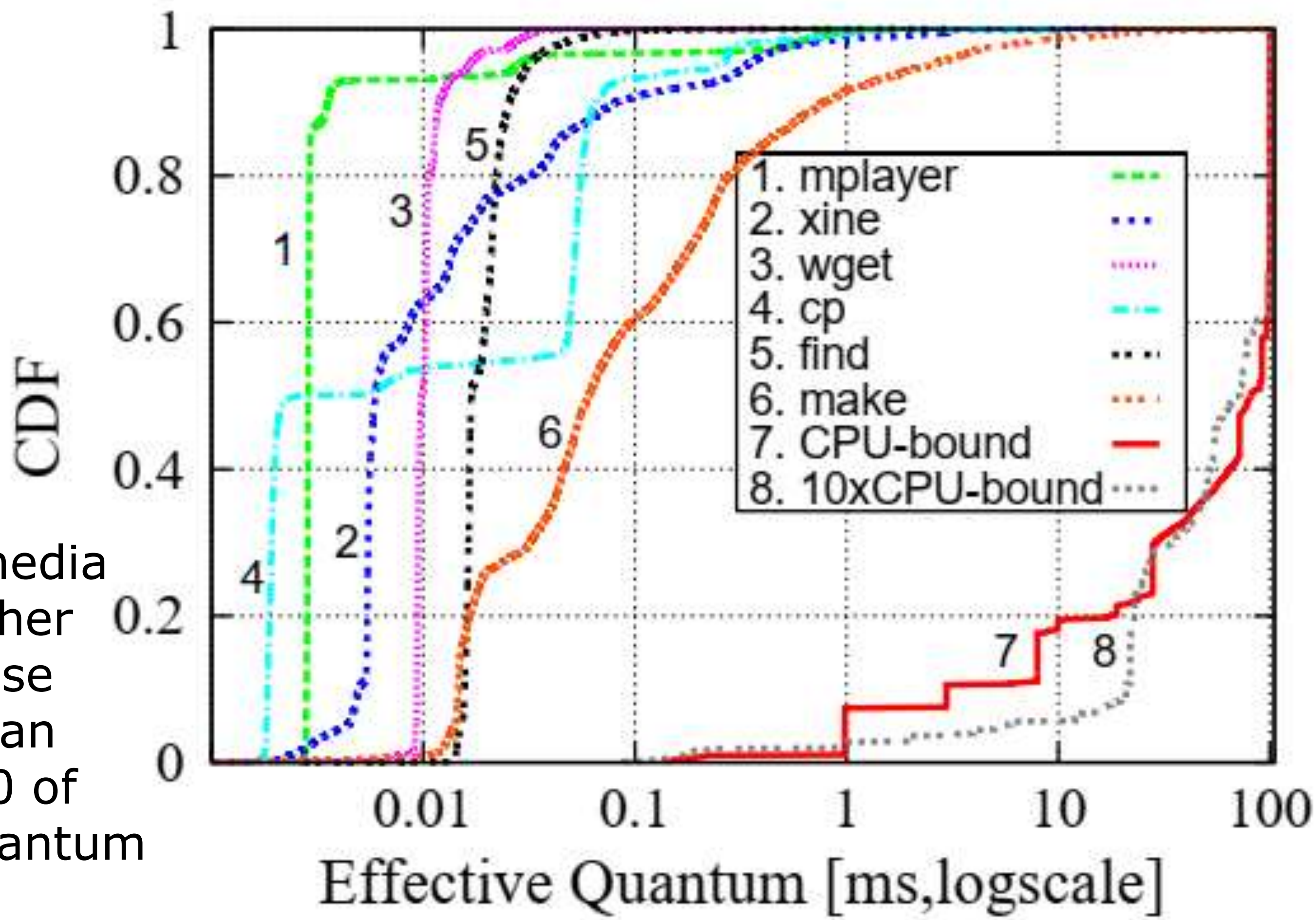
Data on Effective Quantum

Distribution of actual running time for different apps
(till finished burst, external interrupt, or end of quantum)



Data on Effective Quantum

Distribution of actual running time for different apps (till finished burst, external interrupt, or end of quantum)

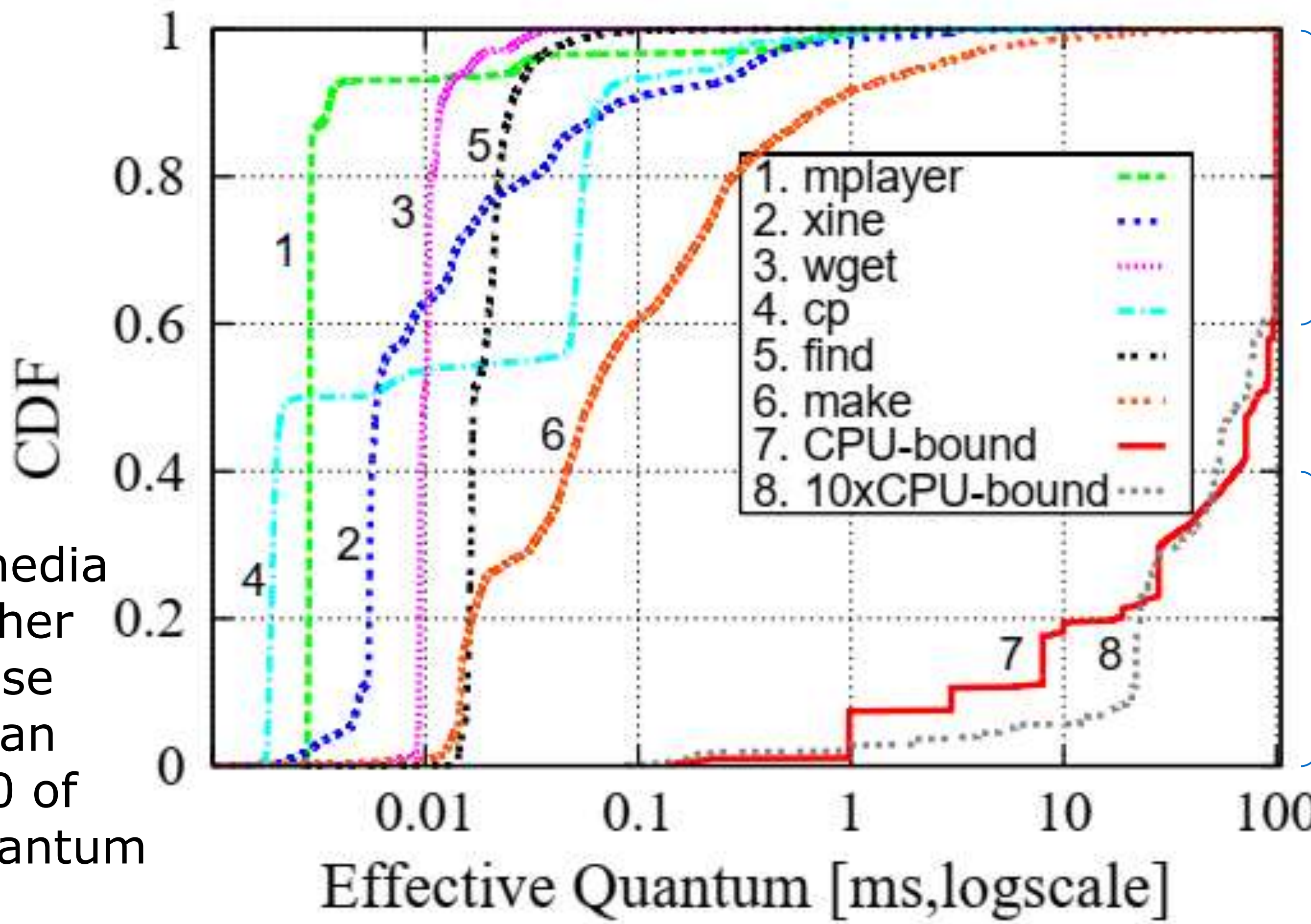


multimedia and other apps use less than 1/1000 of the quantum

Data on Effective Quantum

Distribution of actual running time for different apps (till finished burst, external interrupt, or end of quantum)

multimedia and other apps use less than 1/1000 of the quantum



~40% of CPU-bound jobs use full quantum

~40% are interrupted and use less than half the quantum

RR Implementation

- The scheduler selects a process to run and gives it the CPU

RR Implementation

- The scheduler selects a process to run and gives it the CPU
  - The OS loses control of the system
  - Possibly until it performs a system call

RR Implementation

- The scheduler selects a process to run and gives it the CPU
  - The OS loses control of the system
  - Possibly until it performs a system call
- So how can we enforce a limited time quantum?

RR Implementation

- The scheduler selects a process to run and gives it the CPU
  - The OS loses control of the system
  - Possibly until it performs a system call
- So how can we enforce a limited time quantum?
- Answer: **Hardware support through periodic clock interrupts**
  - The only way for the OS to re-gain control over the CPU
  - Once the OS is in control, it can perform a context switch

RR Notes

- RR works in an **online** setting
- RR uses preemption to cope with **lack of knowledge**
  - Will additional jobs arrive?
  - How long will jobs run?
- RR gives **uniform treatment** to all jobs

RR Notes

- RR works in an **online** setting
- RR uses preemption to cope with **lack of knowledge**
  - Will additional jobs arrive?
  - How long will jobs run?
- RR gives **uniform treatment** to all jobs

- **Can we do better?**