

Scheduling

Johan Montelius

KTH

2019

Problem:

Problem:

We have a set of processes: they all want to execute immediately and they do not want to be interrupted.

Problem:

We have a set of processes: they all want to execute immediately and they do not want to be interrupted.

Solution:

Problem:

We have a set of processes: they all want to execute immediately and they do not want to be interrupted.

Solution:

Let's keep some waiting and let's interrupt them.

Problem:

We have a set of processes: they all want to execute immediately and they do not want to be interrupted.

Solution:

Let's keep some waiting and let's interrupt them.

Question:

Problem:

We have a set of processes: they all want to execute immediately and they do not want to be interrupted.

Solution:

Let's keep some waiting and let's interrupt them.

Question:

- What metrics are important?

Problem:

We have a set of processes: they all want to execute immediately and they do not want to be interrupted.

Solution:

Let's keep some waiting and let's interrupt them.

Question:

- What metrics are important?
- Does it matter in what order we schedule processes?

Problem:

We have a set of processes: they all want to execute immediately and they do not want to be interrupted.

Solution:

Let's keep some waiting and let's interrupt them.

Question:

- What metrics are important?
- Does it matter in what order we schedule processes?
- Are there optimal solutions?

The unrealistic assumption ...

Assume we have a set of *jobs*.

The unrealistic assumption ...

Assume we have a set of *jobs*.

- Each job takes an equal amount of time.

The unrealistic assumption ...

Assume we have a set of *jobs*.

- Each job takes an equal amount of time.
- All jobs *arrive* at the same time.

The unrealistic assumption ...

Assume we have a set of *jobs*.

- Each job takes an equal amount of time.
- All jobs *arrive* at the same time.
- A job will run to completion.

The unrealistic assumption ...

Assume we have a set of *jobs*.

- Each job takes an equal amount of time.
- All jobs *arrive* at the same time.
- A job will run to completion.
- The jobs only use the CPU (no I/O etc).

The unrealistic assumption ...

Assume we have a set of *jobs*.

- Each job takes an equal amount of time.
- All jobs *arrive* at the same time.
- A job will run to completion.
- The jobs only use the CPU (no I/O etc).
- The run-time of each job is known.

The unrealistic assumption ...

Assume we have a set of *jobs*.

- Each job takes an equal amount of time.
- All jobs *arrive* at the same time.
- A job will run to completion.
- The jobs only use the CPU (no I/O etc).
- The run-time of each job is known.

The unrealistic assumption ...

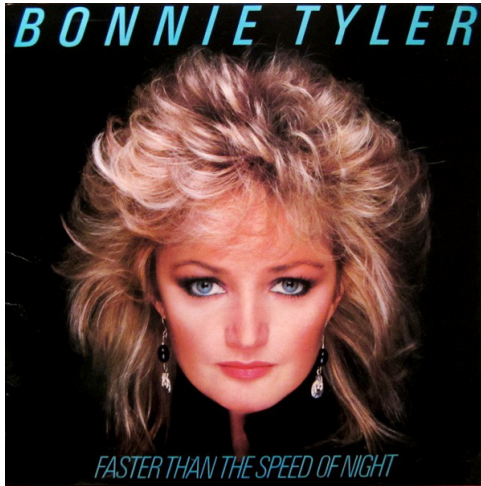
Assume we have a set of *jobs*.

- Each job takes an equal amount of time.
- All jobs *arrive* at the same time.
- A job will run to completion.
- The jobs only use the CPU (no I/O etc).
- The run-time of each job is known.

This is unrealistic - we will relax these requirements.

...every now and then I get a little bit lonely

...every now and then I get a little bit lonely



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

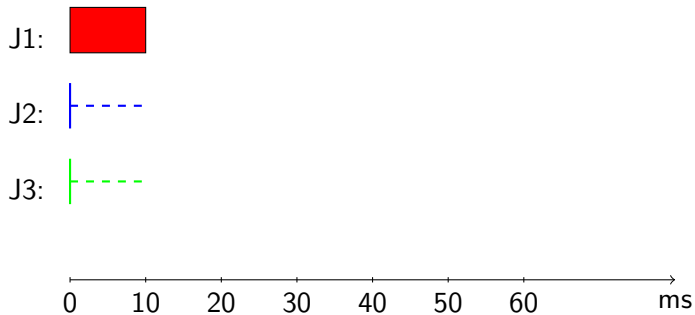
How long time does it take to complete the job?

First Come First Serve (FCFS)

Assume we have three tasks, all *arrive* at time 0 and take 10 ms to execute.

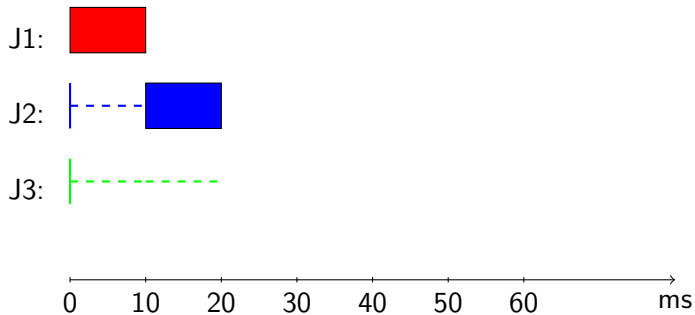
First Come First Serve (FCFS)

Assume we have three tasks, all *arrive* at time 0 and take 10 ms to execute.



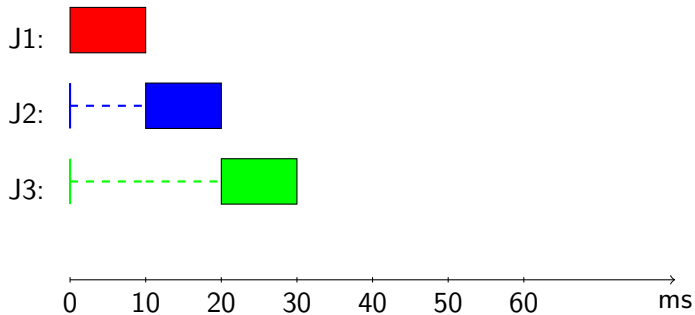
First Come First Serve (FCFS)

Assume we have three tasks, all *arrive* at time 0 and take 10 ms to execute.



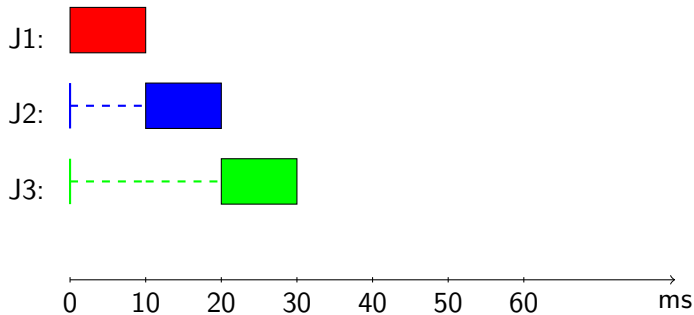
First Come First Serve (FCFS)

Assume we have three tasks, all *arrive* at time 0 and take 10 ms to execute.



First Come First Serve (FCFS)

Assume we have three tasks, all *arrive* at time 0 and take 10 ms to execute.



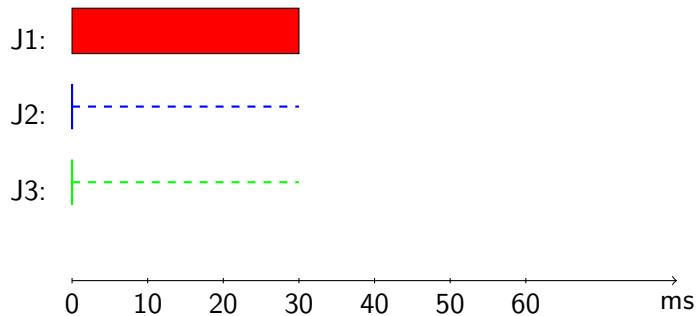
What is the average $T_{\text{turnaround}}$?

Not so good...

Assume one task takes 30 ms to execute.

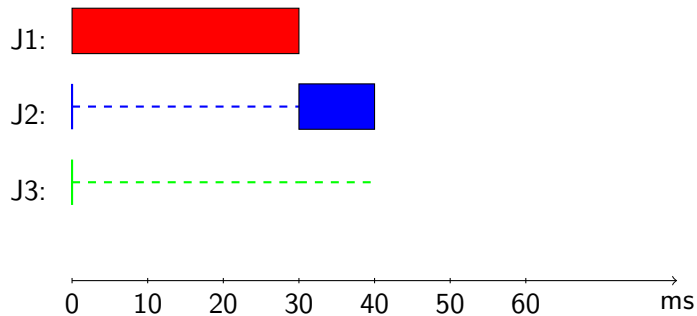
Not so good...

Assume one task takes 30 ms to execute.



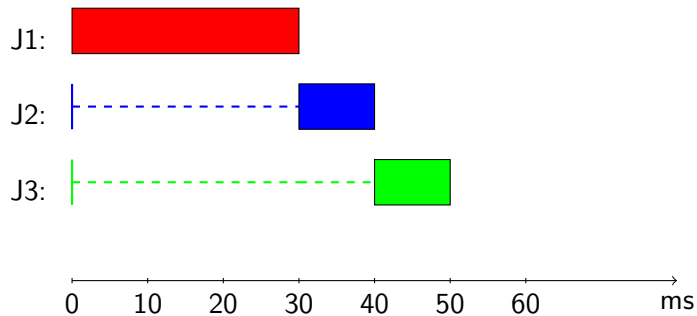
Not so good...

Assume one task takes 30 ms to execute.



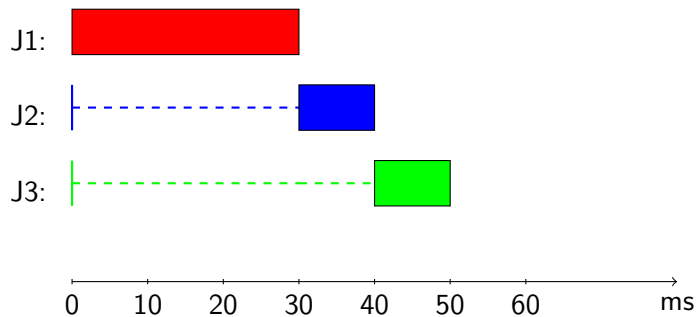
Not so good...

Assume one task takes 30 ms to execute.



Not so good...

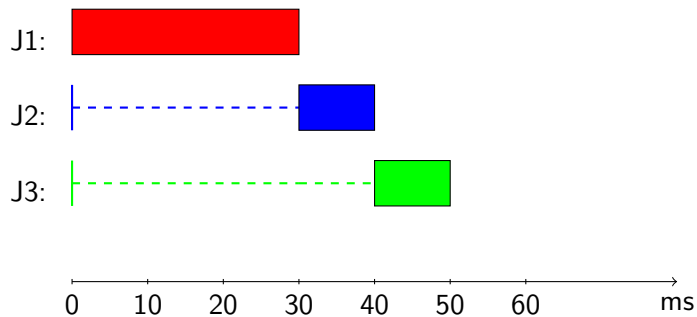
Assume one task takes 30 ms to execute.



What is the average $T_{\text{turnaround}}$?

Not so good...

Assume one task takes 30 ms to execute.



What is the average $T_{\text{turnaround}}$?

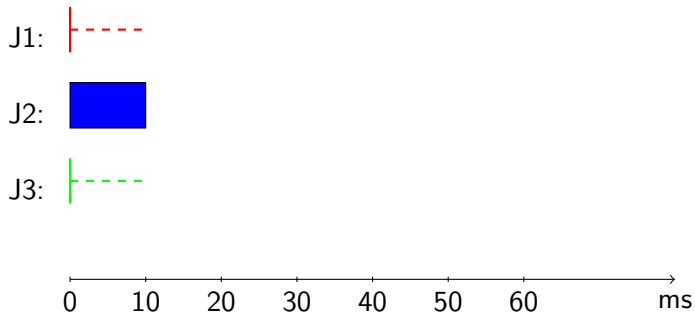
Can we do better?

Shortest Job First (SJF)

Always schedule the shortest job.

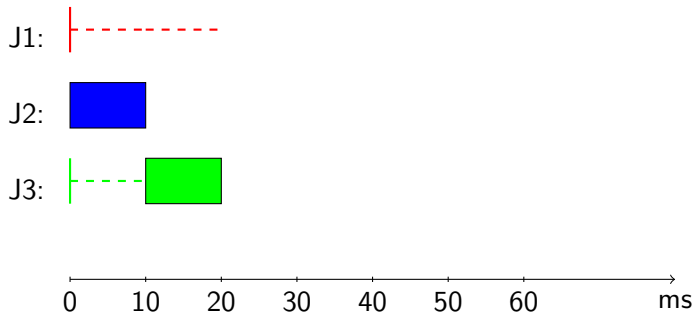
Shortest Job First (SJF)

Always schedule the shortest job.



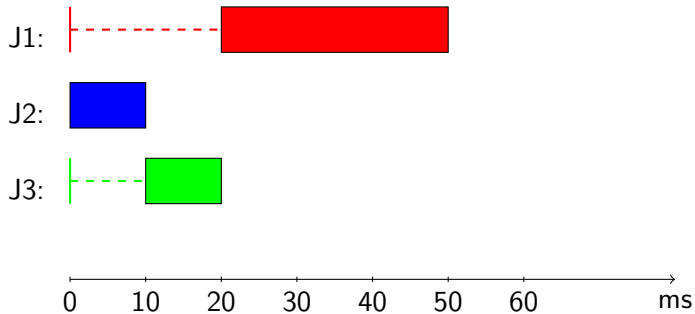
Shortest Job First (SJF)

Always schedule the shortest job.



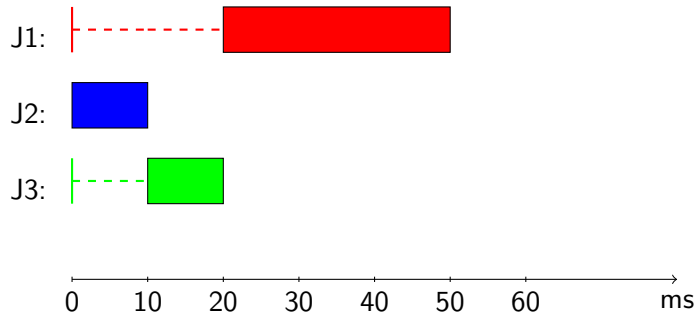
Shortest Job First (SJF)

Always schedule the shortest job.



Shortest Job First (SJF)

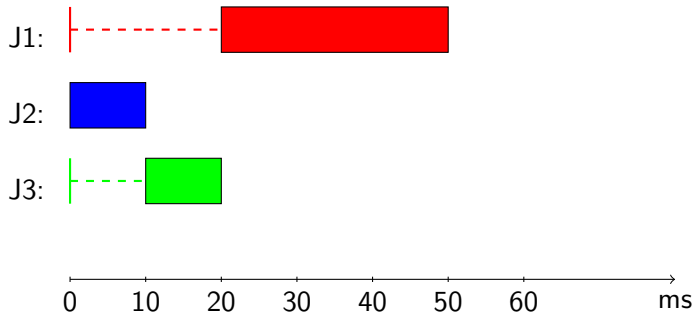
Always schedule the shortest job.



What is the average $T_{\text{turnaround}}$?

Shortest Job First (SJF)

Always schedule the shortest job.



What is the average $T_{\text{turnaround}}$?

Problem solved!

What if jobs arrive later?

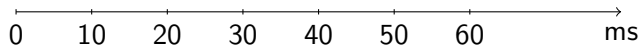
What if jobs arrive later?

Assume we have three tasks, one arrive at time 0 and takes 30 ms to execute. Two arrive at time 10 and take 10 ms each.

J1:

J2:

J3:



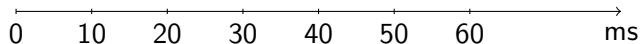
What if jobs arrive later?

Assume we have three tasks, one arrive at time 0 and takes 30 ms to execute. Two arrive at time 10 and take 10 ms each.

J1: 

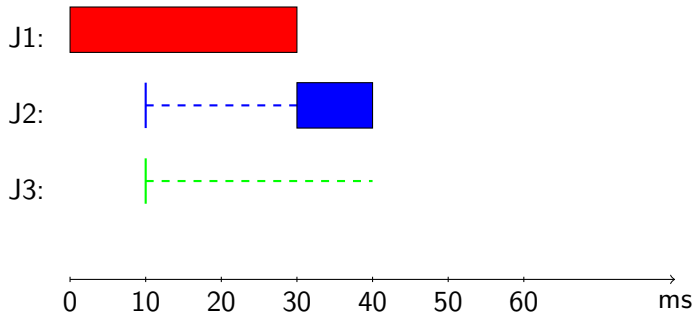
J2:

J3:



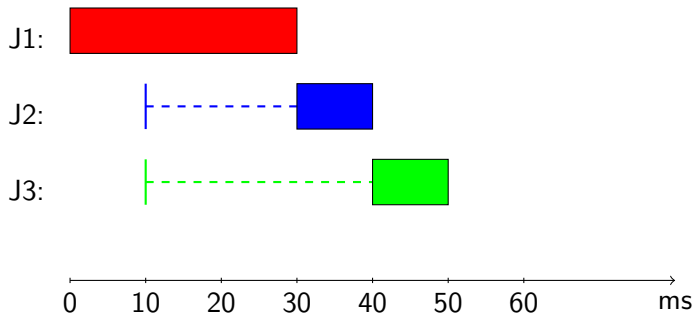
What if jobs arrive later?

Assume we have three tasks, one arrive at time 0 and takes 30 ms to execute. Two arrive at time 10 and take 10 ms each.



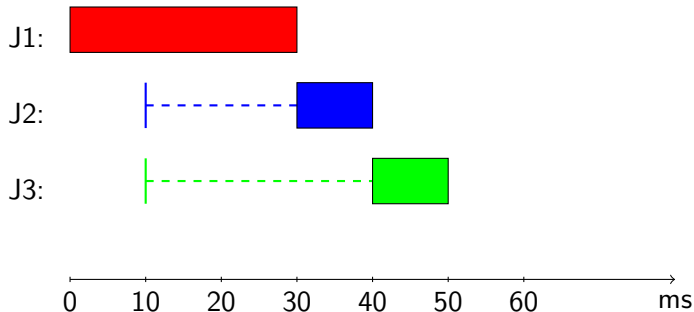
What if jobs arrive later?

Assume we have three tasks, one arrive at time 0 and takes 30 ms to execute. Two arrive at time 10 and take 10 ms each.



What if jobs arrive later?

Assume we have three tasks, one arrive at time 0 and takes 30 ms to execute. Two arrive at time 10 and take 10 ms each.



We need to preempt the execution of a job.

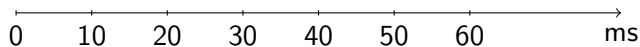
Shortest Time-to-Completion First (STCF)

Let's always schedule the task that has the shortest time left to completion.

J1:

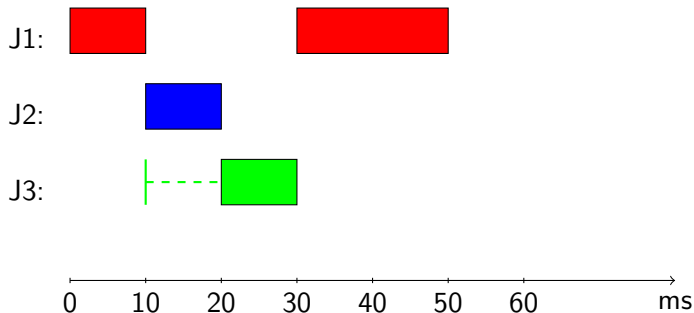
J2:

J3:



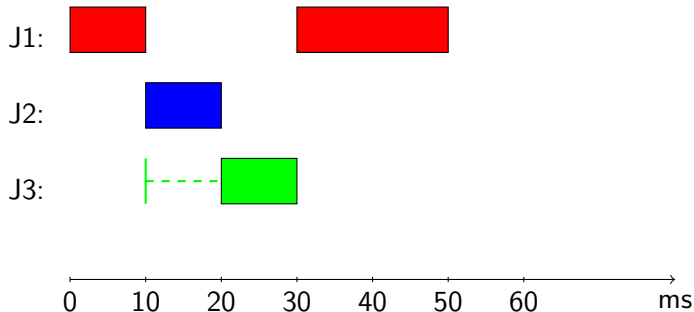
Shortest Time-to-Completion First (STCF)

Let's always schedule the task that has the shortest time left to completion.



Shortest Time-to-Completion First (STCF)

Let's always schedule the task that has the shortest time left to completion.



The policy is also known as Preemptive Shortest Job First (PSJF)

If we actually know the total execution time of each job as they arrive, then

If we actually know the total execution time of each job as they arrive, then

Shortest Time-to-Completion First is an optimal policy.

If we actually know the total execution time of each job as they arrive, then

Shortest Time-to-Completion First is an optimal policy.

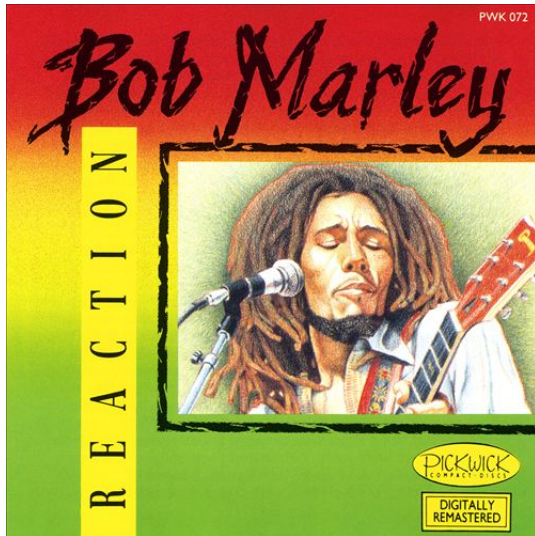
The problem is that we do not know the total execution time beforehand.

If we actually know the total execution time of each job as they arrive, then

Shortest Time-to-Completion First is an optimal policy.

The problem is that we do not know the total execution time beforehand.

There might be more important metrics than turnaround time.



In an interactive environment we might want to minimize *response time*.

In an interactive environment we might want to minimize *response time*.

$$T_{\text{response}} = T_{\text{first scheduled}} - T_{\text{arrival}}$$

In an interactive environment we might want to minimize *response time*.

$$T_{\text{response}} = T_{\text{first scheduled}} - T_{\text{arrival}}$$

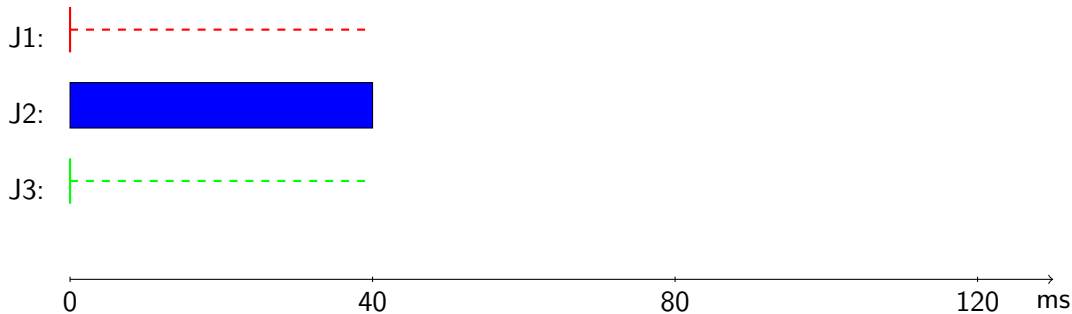
The response might not be completed unless the job completes but it's an ok metrics.

Try Shortest Job First

Assume we have three jobs that all arrive at time 0 and all take 40 ms to complete.

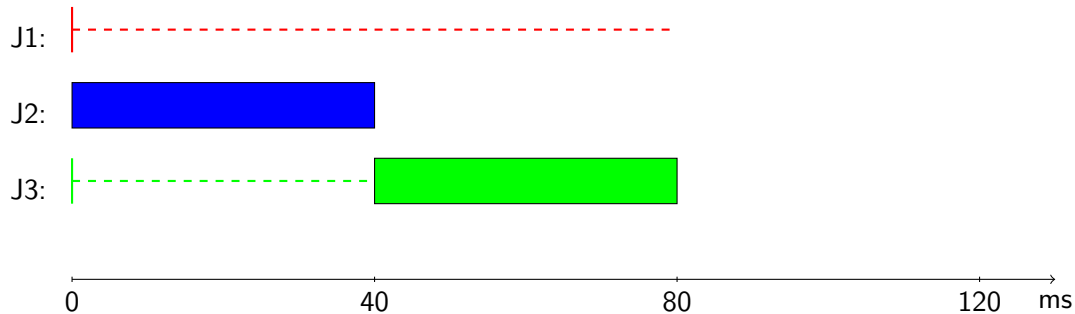
Try Shortest Job First

Assume we have three jobs that all arrive at time 0 and all take 40 ms to complete.



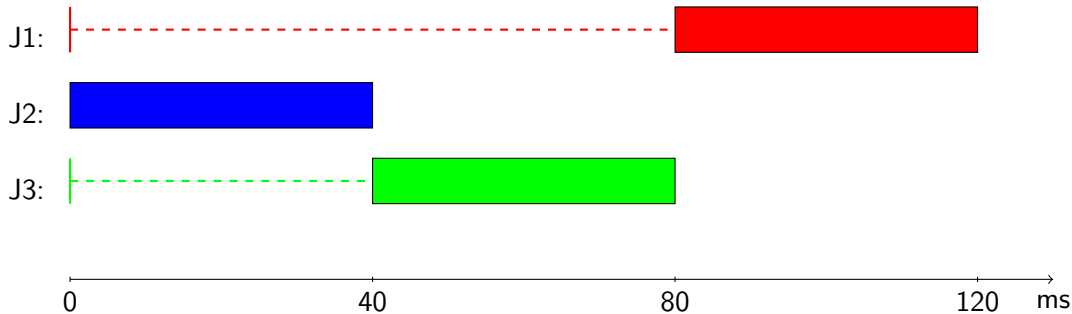
Try Shortest Job First

Assume we have three jobs that all arrive at time 0 and all take 40 ms to complete.



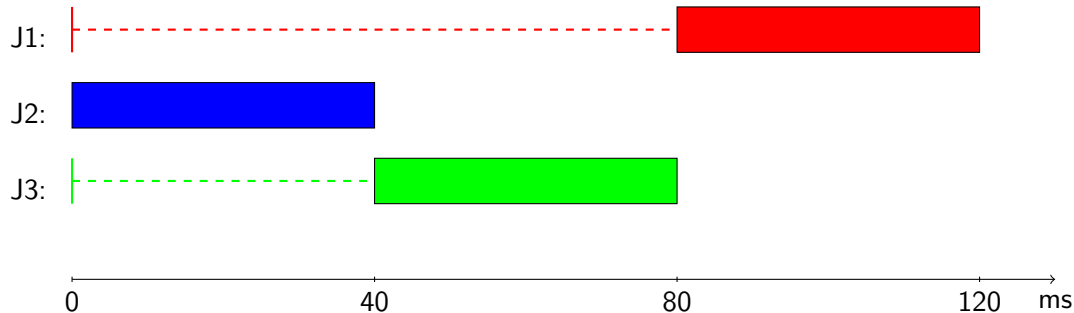
Try Shortest Job First

Assume we have three jobs that all arrive at time 0 and all take 40 ms to complete.



Try Shortest Job First

Assume we have three jobs that all arrive at time 0 and all take 40 ms to complete.



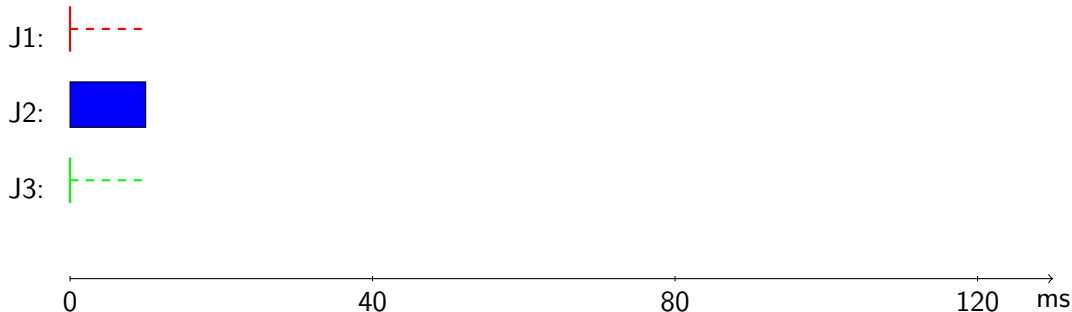
What is the average response time?

Round-robin

Preempt a job in order to improve response time, give each job a time-slice of 10 ms.

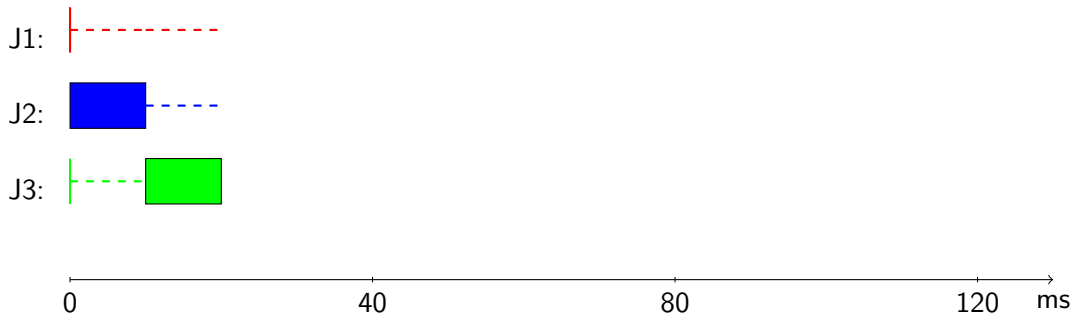
Round-robin

Preempt a job in order to improve response time, give each job a time-slice of 10 ms.



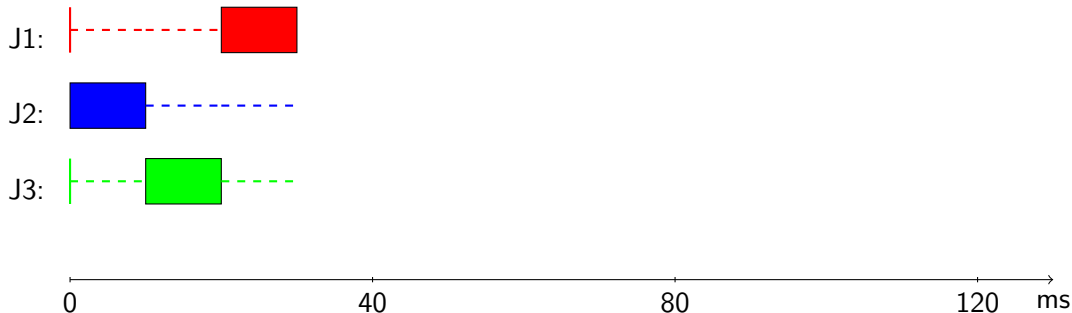
Round-robin

Preempt a job in order to improve response time, give each job a time-slice of 10 ms.



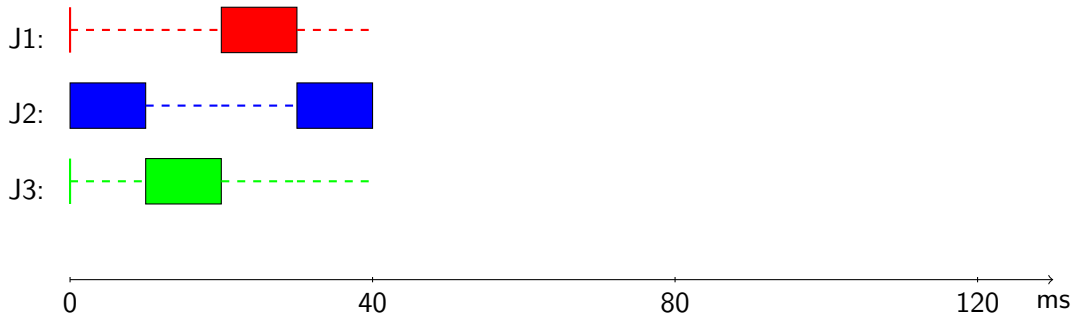
Round-robin

Preempt a job in order to improve response time, give each job a time-slice of 10 ms.



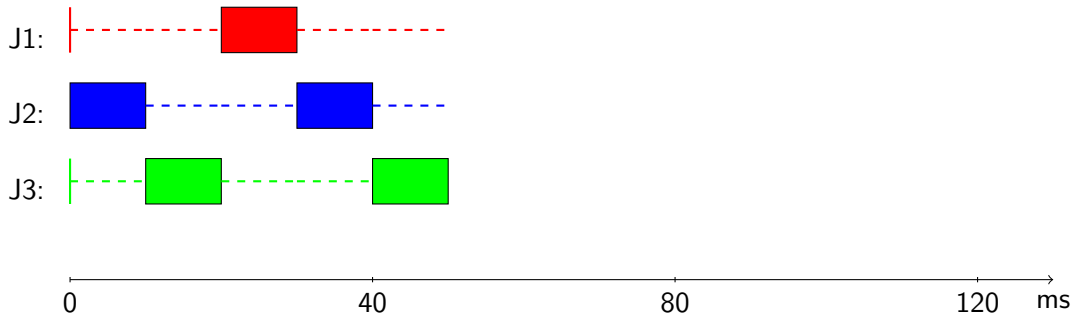
Round-robin

Preempt a job in order to improve response time, give each job a time-slice of 10 ms.



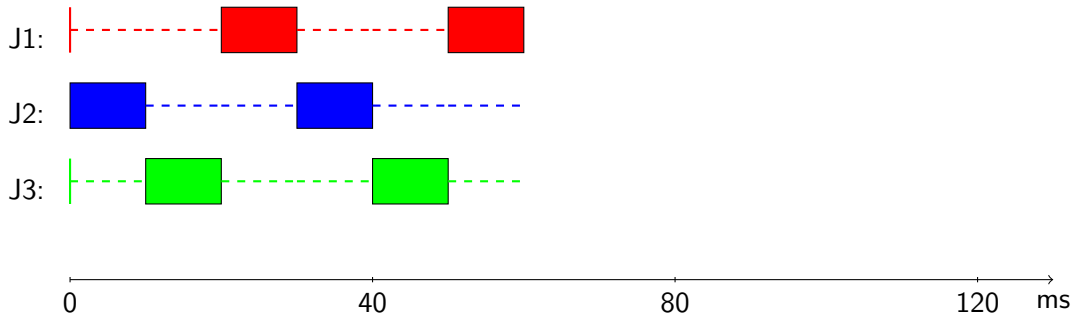
Round-robin

Preempt a job in order to improve response time, give each job a time-slice of 10 ms.



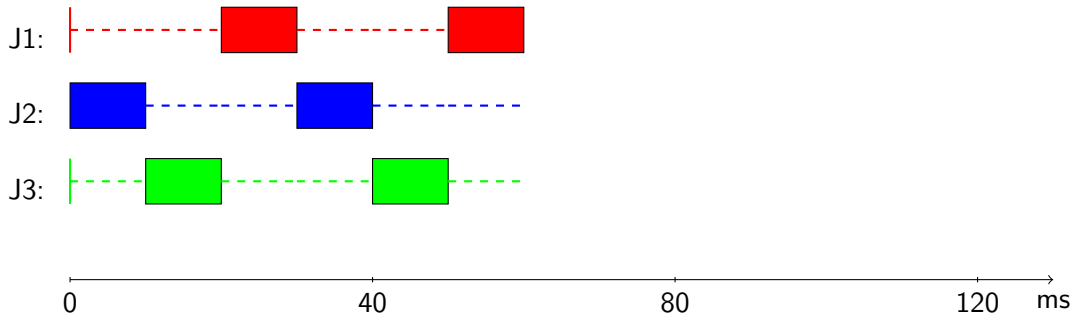
Round-robin

Preempt a job in order to improve response time, give each job a time-slice of 10 ms.



Round-robin

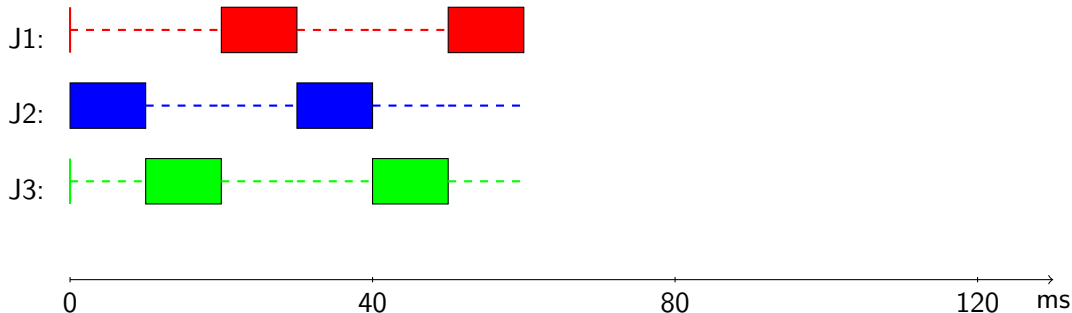
Preempt a job in order to improve response time, give each job a time-slice of 10 ms.



What is the average response time?

Round-robin

Preempt a job in order to improve response time, give each job a time-slice of 10 ms.

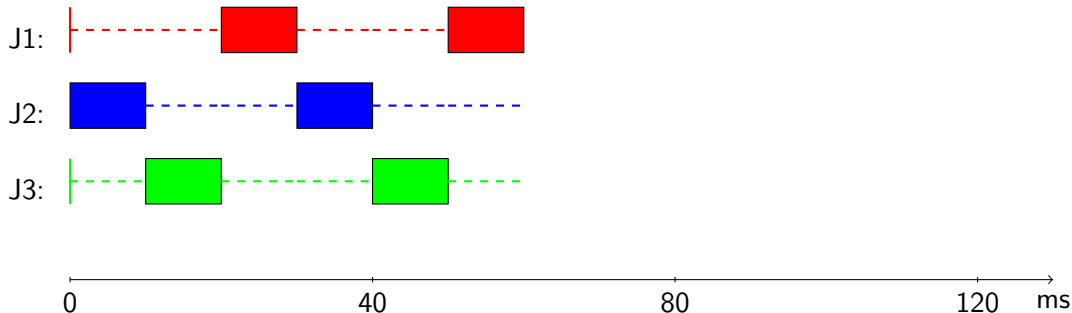


What is the average response time?

What is the average turnaround time?

Round-robin

Preempt a job in order to improve response time, give each job a time-slice of 10 ms.



What is the average response time?

What is the average turnaround time?

How to choose the time-slice?

You can't

6103 062

062 **sad day** **the rolling stones**
you can't always get
what you want

DECCA

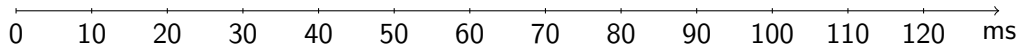


processes do I/O

Assume we have two processes, each take 40 ms of CPU time but one will do I/O-operations every 10 ms.

J1:

J2:

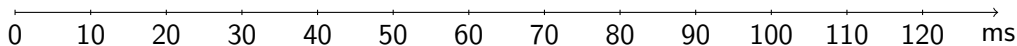


processes do I/O

Assume we have two processes, each take 40 ms of CPU time but one will do I/O-operations every 10 ms.

J1: 

J2: 



processes do I/O

Assume we have two processes, each take 40 ms of CPU time but one will do I/O-operations every 10 ms.

J1:  I/O

J2: 

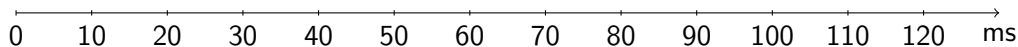


processes do I/O

Assume we have two processes, each take 40 ms of CPU time but one will do I/O-operations every 10 ms.



J1: 

J2: 



processes do I/O

Assume we have two processes, each take 40 ms of CPU time but one will do I/O-operations every 10 ms.

J1:  I/O  I/O

J2:  

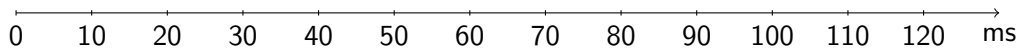


processes do I/O

Assume we have two processes, each take 40 ms of CPU time but one will do I/O-operations every 10 ms.

J1: 

J2: 



processes do I/O

Assume we have two processes, each take 40 ms of CPU time but one will do I/O-operations every 10 ms.

J1:  I/O  I/O  I/O

J2:  

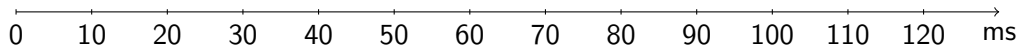


processes do I/O

Assume we have two processes, each take 40 ms of CPU time but one will do I/O-operations every 10 ms.

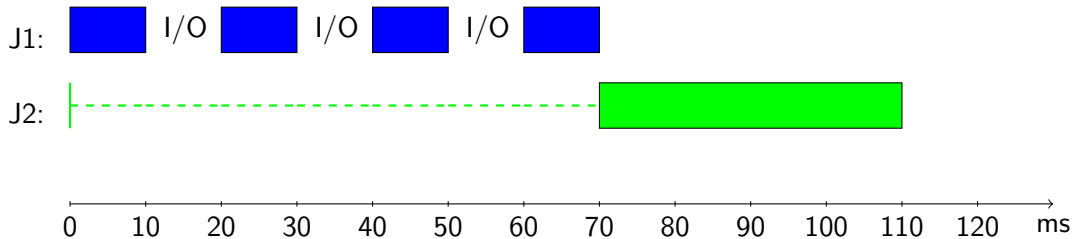
J1:  J1's execution timeline consists of four 10ms CPU blocks (blue rectangles) and three 10ms I/O gaps (labeled 'I/O'). The blocks occur from 0-10ms, 20-30ms, 40-50ms, and 60-70ms. The I/O gaps occur from 10-20ms, 30-40ms, and 50-60ms.

J2:  J2's execution timeline is represented by a solid green vertical line at 0ms and a dashed green horizontal line extending to 70ms, indicating a single 70ms CPU block.



processes do I/O

Assume we have two processes, each take 40 ms of CPU time but one will do I/O-operations every 10 ms.

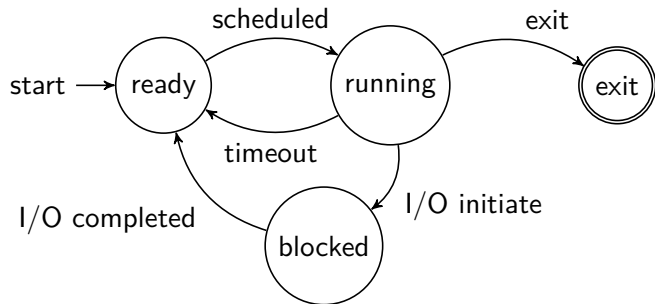


deschedule when initiate I/O

An I/O-operation will take time to complete and we (the CPU) could do some useful work while a process is waiting.

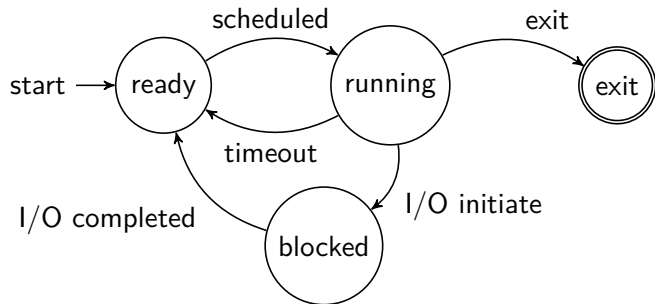
deschedule when initiate I/O

An I/O-operation will take time to complete and we (the CPU) could do some useful work while a process is waiting.



deschedule when initiate I/O

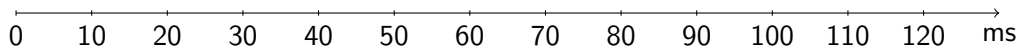
An I/O-operation will take time to complete and we (the CPU) could do some useful work while a process is waiting.



A process is descheduled if it is preempted or if it initiates a I/O-operation.

J1:

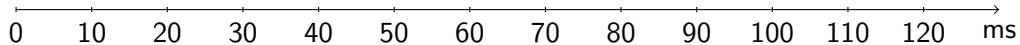
J2:



much better

J1: 

J2: 






J1:  I/O

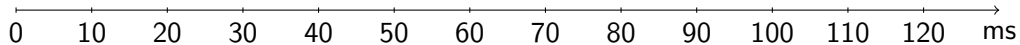
J2: 





much better

J1:  I/O 

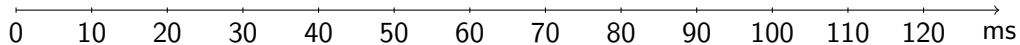
J2:   



much better



J1:  I/O  I/O

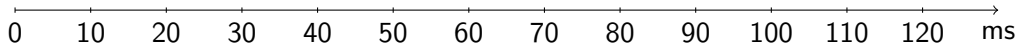
J2:  



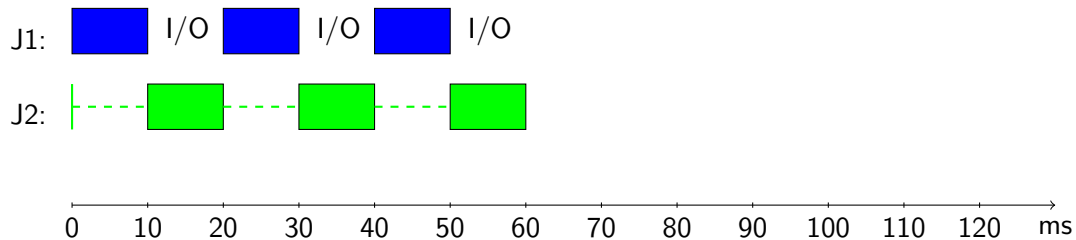
much better

J1:  I/O  I/O 

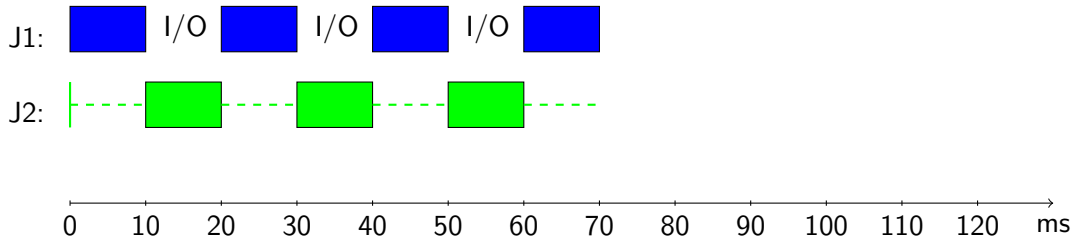
J2:  



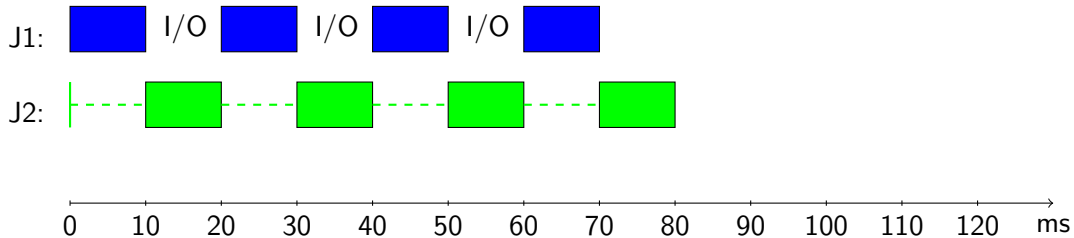
much better



much better



much better



Ideal world:

- Each job takes an equal amount of time.

Ideal world:

- Each job takes an equal amount of time.
- All jobs *arrive* at the same time.

Ideal world:

- Each job takes an equal amount of time.
- All jobs *arrive* at the same time.
- A job will run to completion.

Ideal world:

- Each job takes an equal amount of time.
- All jobs *arrive* at the same time.
- A job will run to completion.
- The jobs only use the CPU (no I/O etc).

Ideal world:

- Each job takes an equal amount of time.
- All jobs *arrive* at the same time.
- A job will run to completion.
- The jobs only use the CPU (no I/O etc).
- The run-time of each job is known.

Ideal world:

- Each job takes an equal amount of time.
- All jobs *arrive* at the same time.
- A job will run to completion.
- The jobs only use the CPU (no I/O etc).
- The run-time of each job is known.

Ideal world:

- Each job takes an equal amount of time.
- All jobs *arrive* at the same time.
- A job will run to completion.
- The jobs only use the CPU (no I/O etc).
- The run-time of each job is known.

Real world:

the challenge

Ideal world:

- Each job takes an equal amount of time.
- All jobs *arrive* at the same time.
- A job will run to completion.
- The jobs only use the CPU (no I/O etc).
- The run-time of each job is known.

Real world:

- Jobs take different amount of time.

the challenge

Ideal world:

- Each job takes an equal amount of time.
- All jobs *arrive* at the same time.
- A job will run to completion.
- The jobs only use the CPU (no I/O etc).
- The run-time of each job is known.

Real world:

- Jobs take different amount of time.
- Jobs arrive at different time.

the challenge

Ideal world:

- Each job takes an equal amount of time.
- All jobs *arrive* at the same time.
- A job will run to completion.
- The jobs only use the CPU (no I/O etc).
- The run-time of each job is known.

Real world:

- Jobs take different amount of time.
- Jobs arrive at different time.
- We can preempt job.

the challenge

Ideal world:

- Each job takes an equal amount of time.
- All jobs *arrive* at the same time.
- A job will run to completion.
- The jobs only use the CPU (no I/O etc).
- The run-time of each job is known.

Real world:

- Jobs take different amount of time.
- Jobs arrive at different time.
- We can preempt job.
- Jobs do use I/O.

the challenge

Ideal world:

- Each job takes an equal amount of time.
- All jobs *arrive* at the same time.
- A job will run to completion.
- The jobs only use the CPU (no I/O etc).
- The run-time of each job is known.

Real world:

- Jobs take different amount of time.
- Jobs arrive at different time.
- We can preempt job.
- Jobs do use I/O.
- Run-time is not know.

the challenge

Ideal world:

- Each job takes an equal amount of time.
- All jobs *arrive* at the same time.
- A job will run to completion.
- The jobs only use the CPU (no I/O etc).
- The run-time of each job is known.

Real world:

- Jobs take different amount of time.
- Jobs arrive at different time.
- We can preempt job.
- Jobs do use I/O.
- Run-time is not know.
- What do we do?

the challenge

Ideal world:

- Each job takes an equal amount of time.
- All jobs *arrive* at the same time.
- A job will run to completion.
- The jobs only use the CPU (no I/O etc).
- The run-time of each job is known.

Real world:

- Jobs take different amount of time.
- Jobs arrive at different time.
- We can preempt job.
- Jobs do use I/O.
- Run-time is not know.
- What do we do?

Can we design scheduling policies that give us good turn-around time and short response time?

Multi-level Feedback Queue (MLFQ)

Goals:

Multi-level Feedback Queue (MLFQ)

Goals:

- Good turnaround time - scheduled jobs so that jobs with short time to completion are not delayed too much.

Multi-level Feedback Queue (MLFQ)

Goals:

- Good turnaround time - scheduled jobs so that jobs with short time to completion are not delayed too much.
- Improve responsiveness of interactive jobs - schedule *interactive processes* more often.

Idea:

Multi-level Feedback Queue (MLFQ)

Goals:

- Good turnaround time - scheduled jobs so that jobs with short time to completion are not delayed too much.
- Improve responsiveness of interactive jobs - schedule *interactive processes* more often.

Idea:

- Multiple levels of priority - interactive jobs have higher priority.

Multi-level Feedback Queue (MLFQ)

Goals:

- Good turnaround time - scheduled jobs so that jobs with short time to completion are not delayed too much.
- Improve responsiveness of interactive jobs - schedule *interactive processes* more often.

Idea:

- Multiple levels of priority - interactive jobs have higher priority.
- Each level uses round-robin to give processes an equal share.

Multi-level Feedback Queue (MLFQ)

Goals:

- Good turnaround time - scheduled jobs so that jobs with short time to completion are not delayed too much.
- Improve responsiveness of interactive jobs - schedule *interactive processes* more often.

Idea:

- Multiple levels of priority - interactive jobs have higher priority.
- Each level uses round-robin to give processes an equal share.
- Processes can be moved to a higher or lower level depending on their behavior.

Multi-level Feedback Queue (MLFQ)

Goals:

- Good turnaround time - scheduled jobs so that jobs with short time to completion are not delayed too much.
- Improve responsiveness of interactive jobs - schedule *interactive processes* more often.

Idea:

- Multiple levels of priority - interactive jobs have higher priority.
- Each level uses round-robin to give processes an equal share.
- Processes can be moved to a higher or lower level depending on their behavior.

How do we identify interactive processes and how do we make sure that they have high priority?

Rules of the game: MLFQ

Basic rules:

- Rule 1: if $\text{Priority}(A) > \text{Priority}(B)$ then A is scheduled for execution.

Basic rules:

- Rule 1: if $\text{Priority}(A) > \text{Priority}(B)$ then A is scheduled for execution.
- Rule 2: if $\text{Priority}(A) = \text{Priority}(B)$ then A and B are scheduled in round-robin.

Basic rules:

- Rule 1: if $\text{Priority}(A) > \text{Priority}(B)$ then A is scheduled for execution.
- Rule 2: if $\text{Priority}(A) = \text{Priority}(B)$ then A and B are scheduled in round-robin.
- Rule 3: when a new job is created it starts with the highest priority.

Basic rules:

- Rule 1: if $\text{Priority}(A) > \text{Priority}(B)$ then A is scheduled for execution.
- Rule 2: if $\text{Priority}(A) = \text{Priority}(B)$ then A and B are scheduled in round-robin.
- Rule 3: when a new job is created it starts with the highest priority.

Basic rules:

- Rule 1: if $\text{Priority}(A) > \text{Priority}(B)$ then A is scheduled for execution.
- Rule 2: if $\text{Priority}(A) = \text{Priority}(B)$ then A and B are scheduled in round-robin.
- Rule 3: when a new job is created it starts with the highest priority.

Change priority (let's try this)

- Rule 4a: a job that has to be preempted (time-slice consumed) is moved to a lower priority.

Basic rules:

- Rule 1: if $\text{Priority}(A) > \text{Priority}(B)$ then A is scheduled for execution.
- Rule 2: if $\text{Priority}(A) = \text{Priority}(B)$ then A and B are scheduled in round-robin.
- Rule 3: when a new job is created it starts with the highest priority.

Change priority (let's try this)

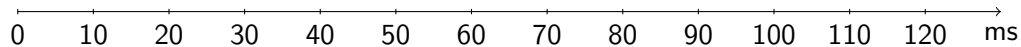
- Rule 4a: a job that has to be preempted (time-slice consumed) is moved to a lower priority.
- Rule 4b: a job that initiates a I/O-operation (or yields) remains on the same level.

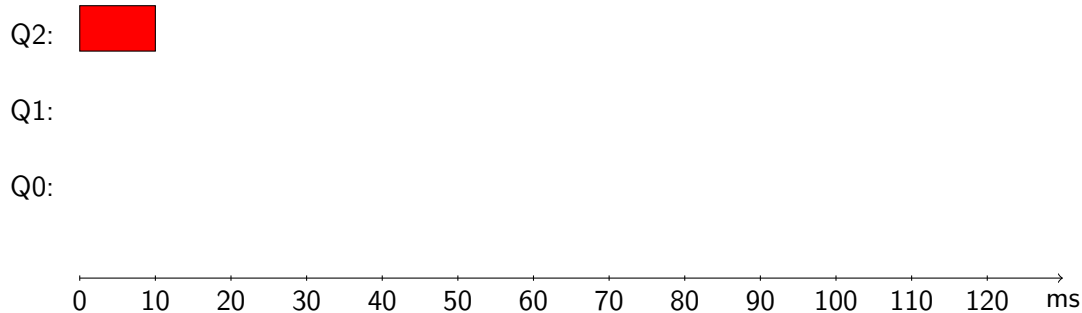
fine, no problem ...

Q2:

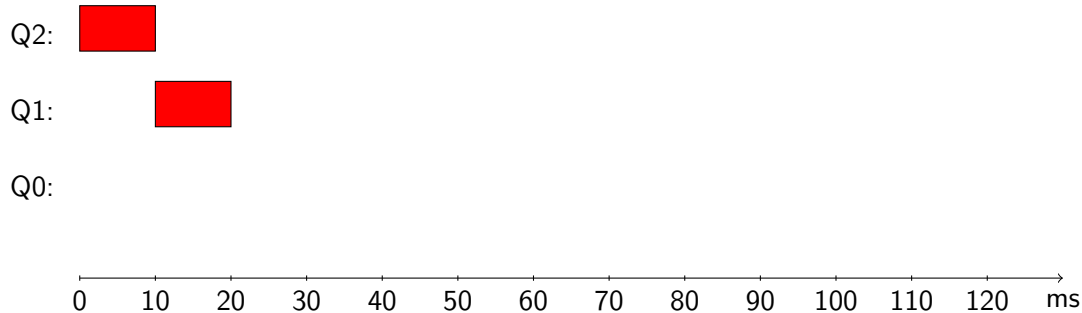
Q1:

Q0:

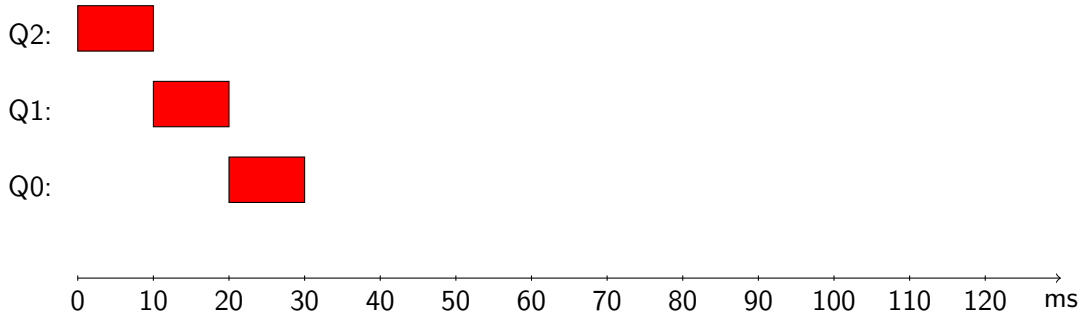




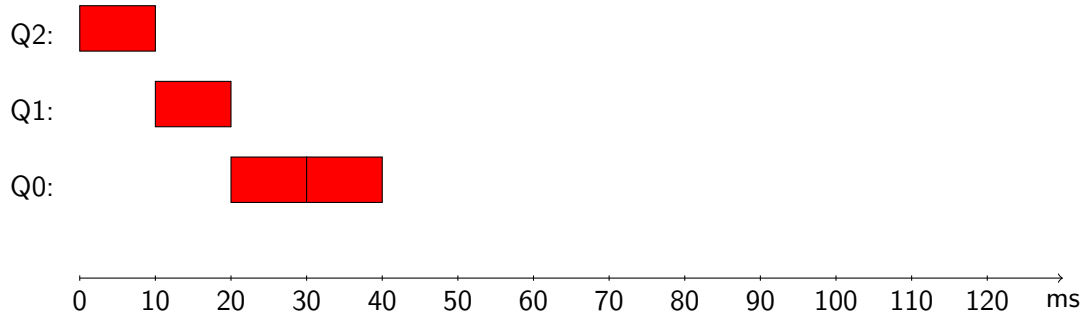
fine, no problem ...



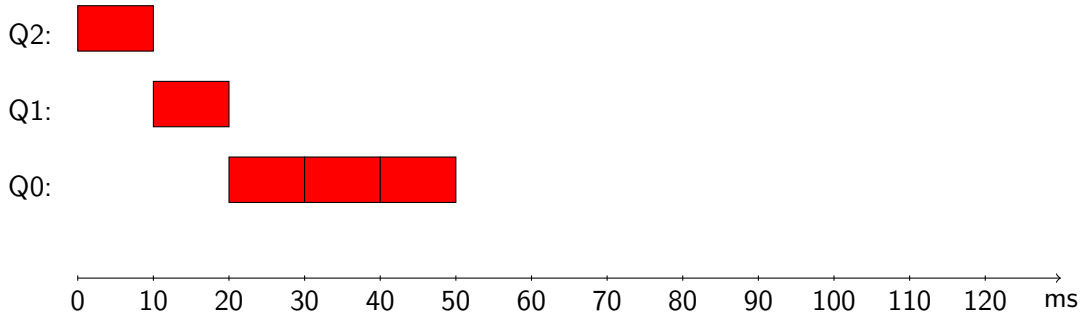
fine, no problem ...



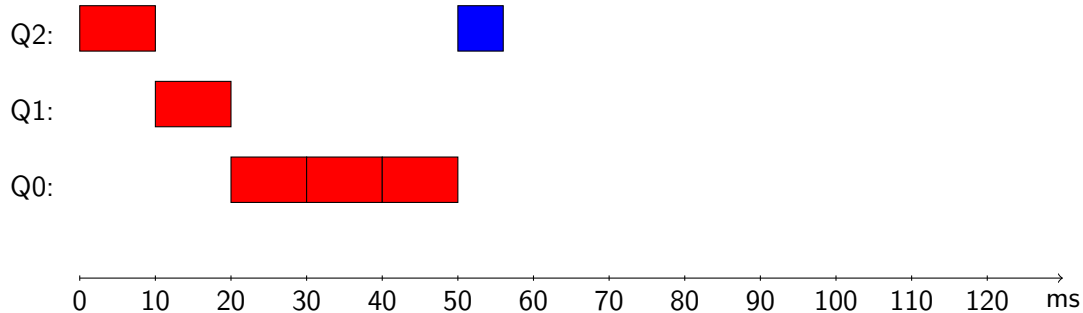
fine, no problem ...



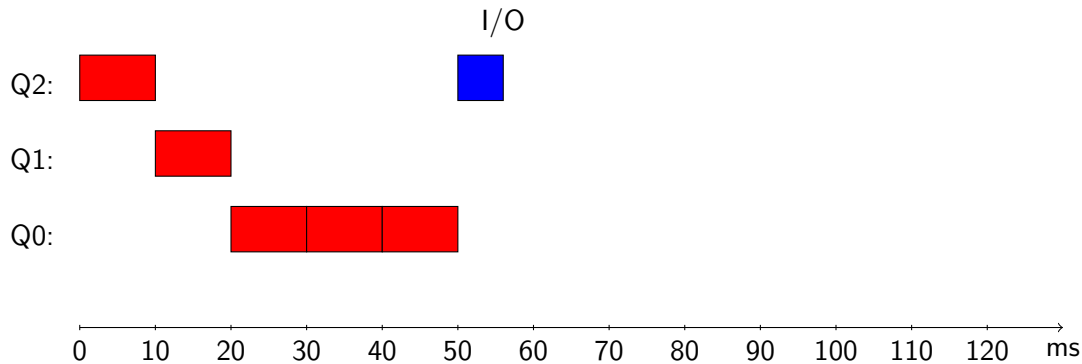
fine, no problem ...



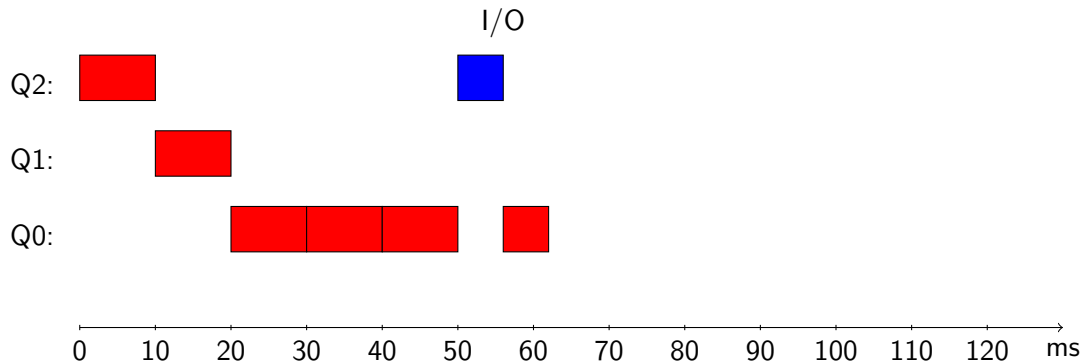
fine, no problem ...



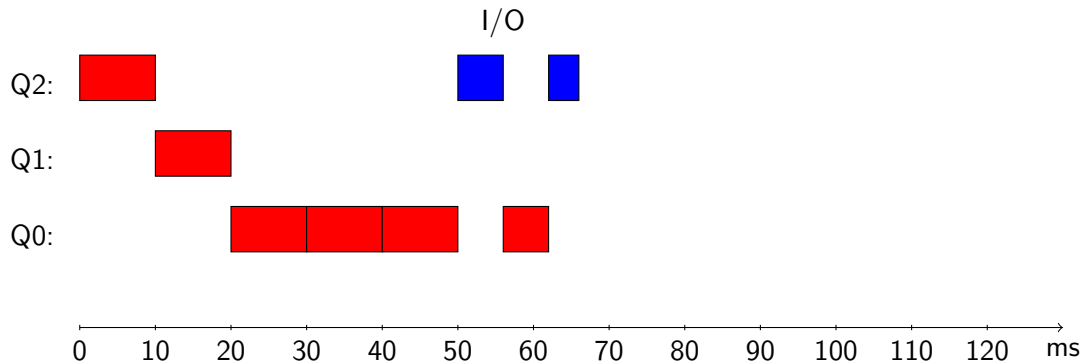
fine, no problem ...



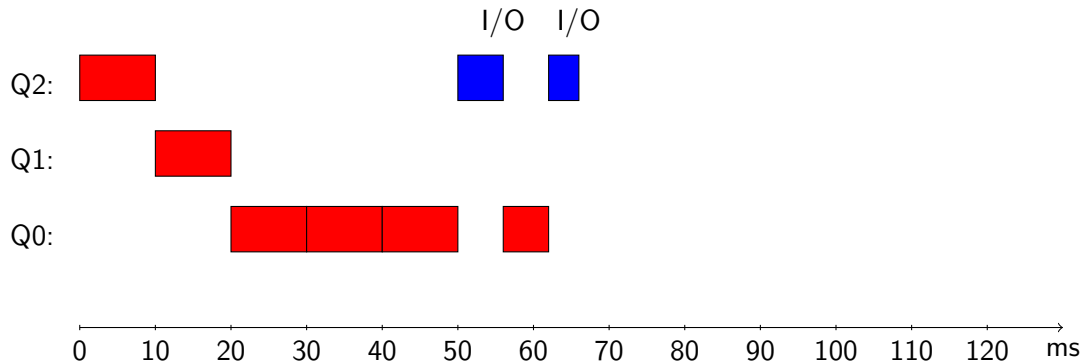
fine, no problem ...



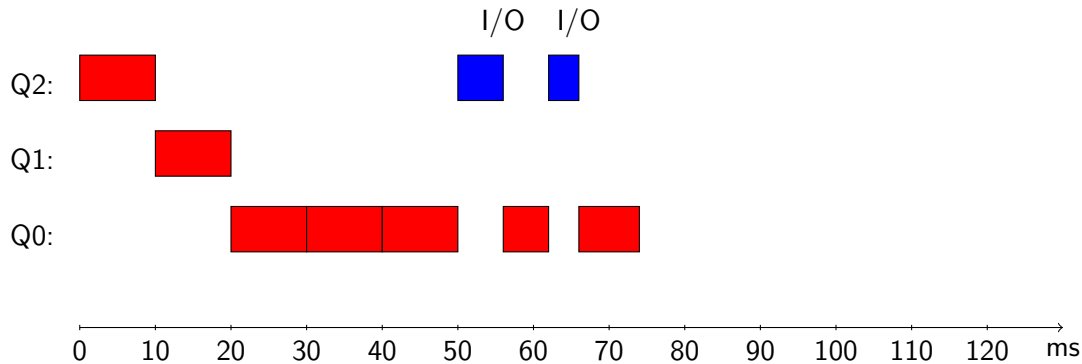
fine, no problem ...



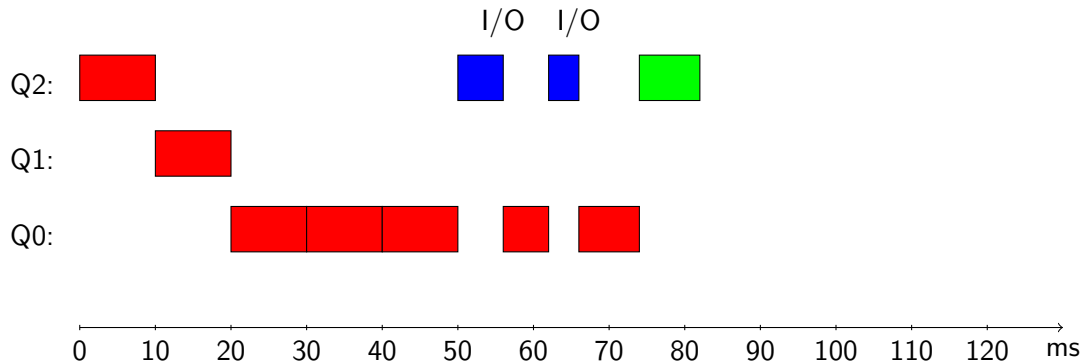
fine, no problem ...



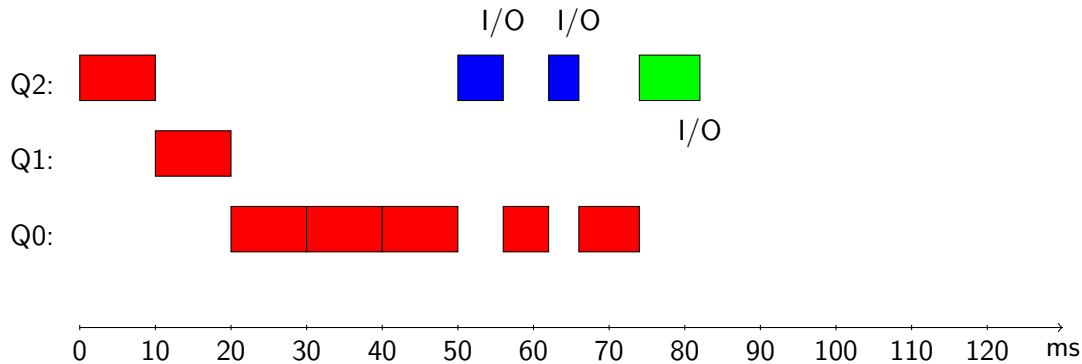
fine, no problem ...



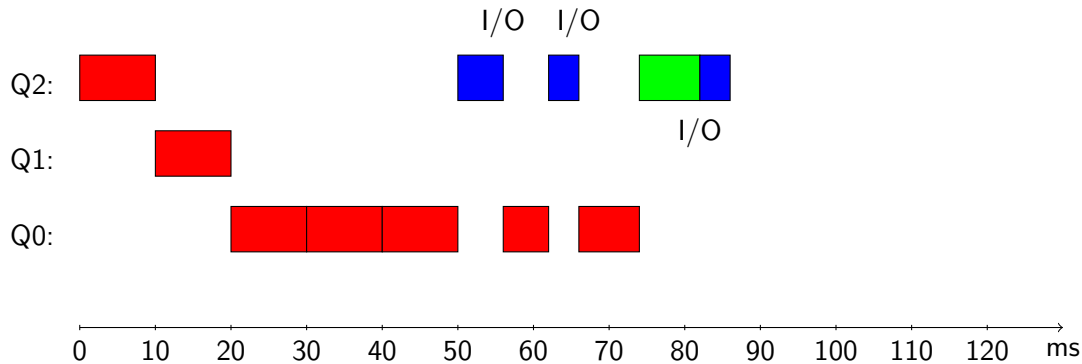
fine, no problem ...



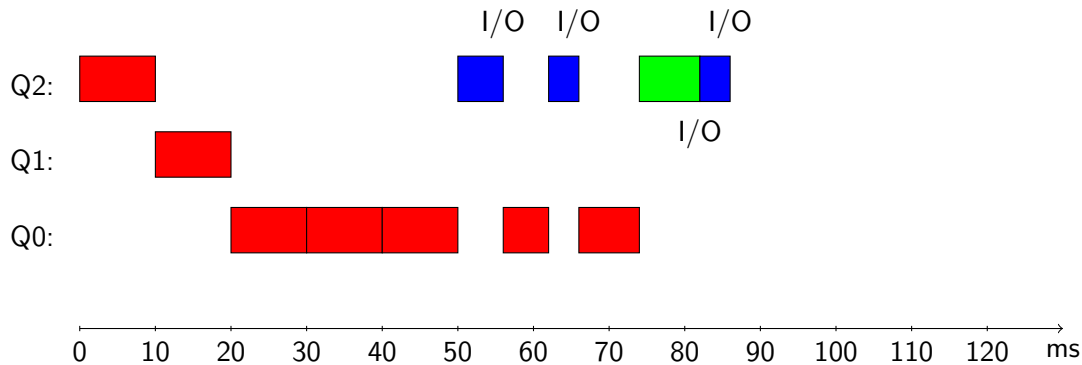
fine, no problem ...



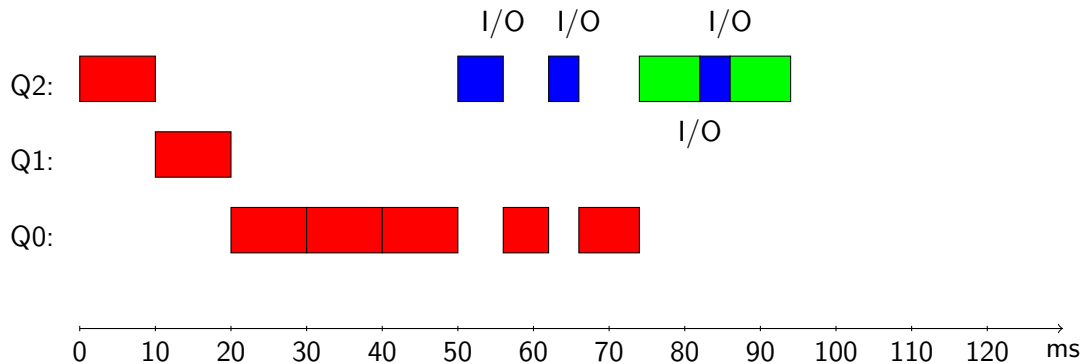
fine, no problem ...



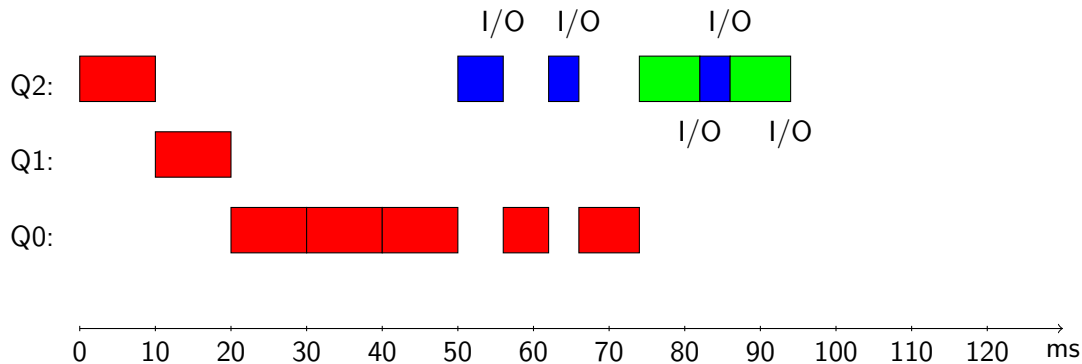
fine, no problem ...



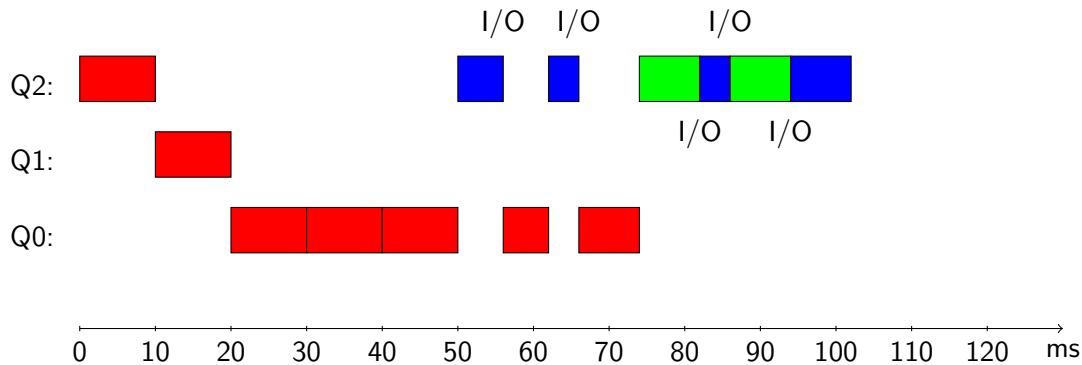
fine, no problem ...



fine, no problem ...



fine, no problem ...



- Rule 5: after some time, move a job to the highest priority.

- Rule 5: after some time, move a job to the highest priority.

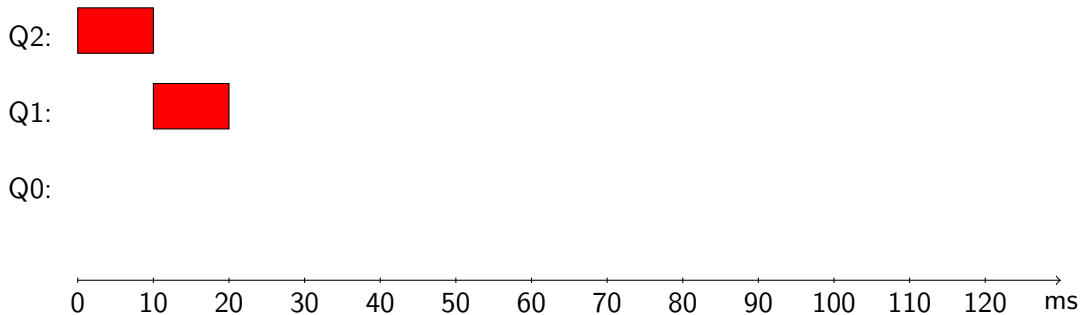
Q2: 

Q1:

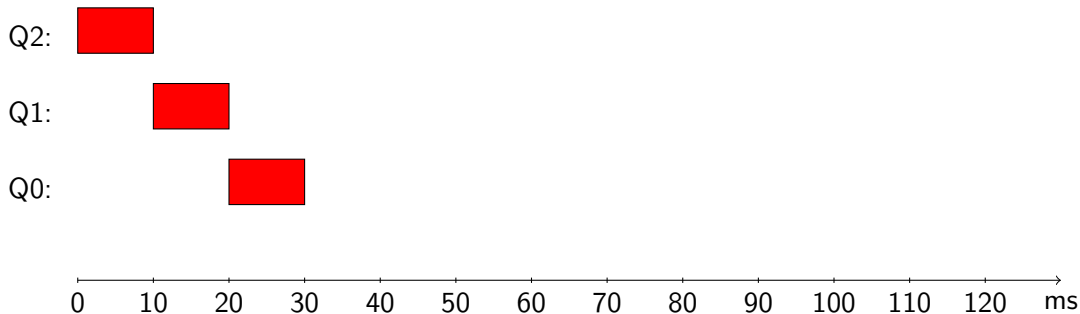
Q0:



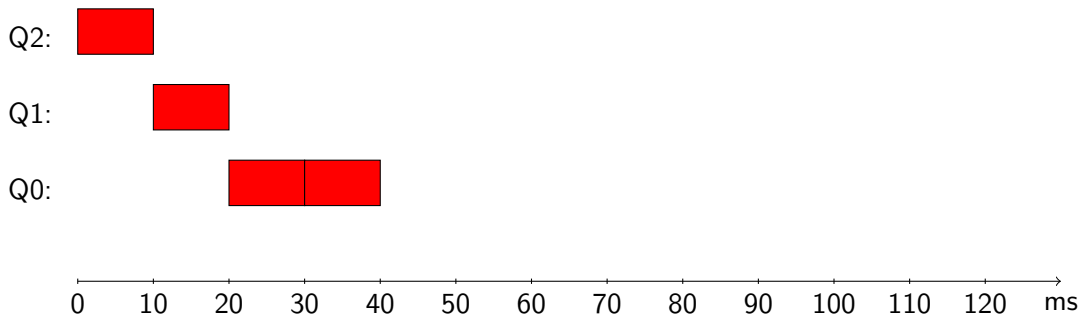
- Rule 5: after some time, move a job to the highest priority.



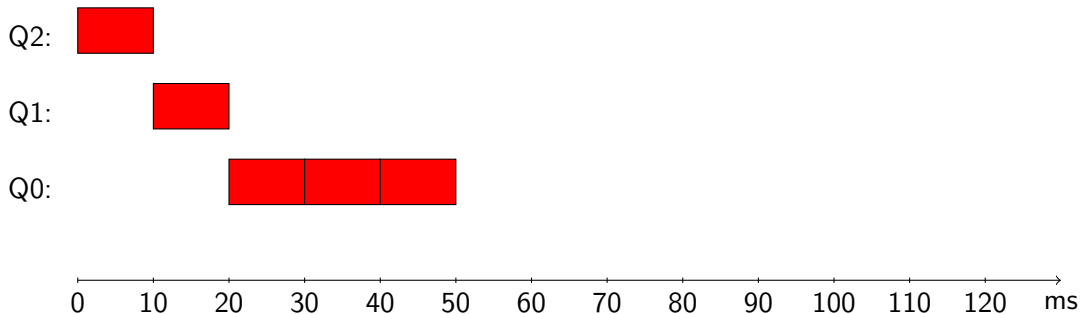
- Rule 5: after some time, move a job to the highest priority.



- Rule 5: after some time, move a job to the highest priority.

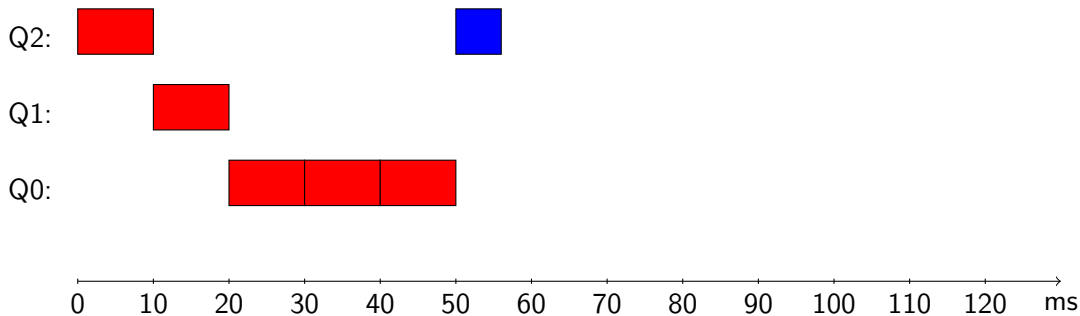


- Rule 5: after some time, move a job to the highest priority.

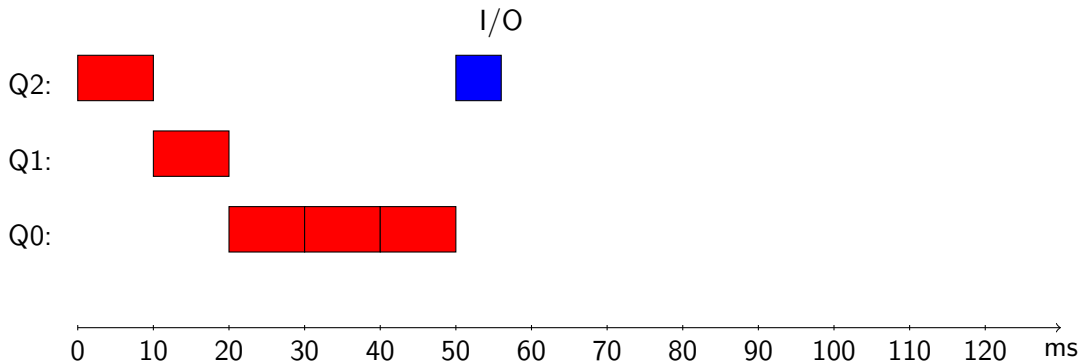


boost a job

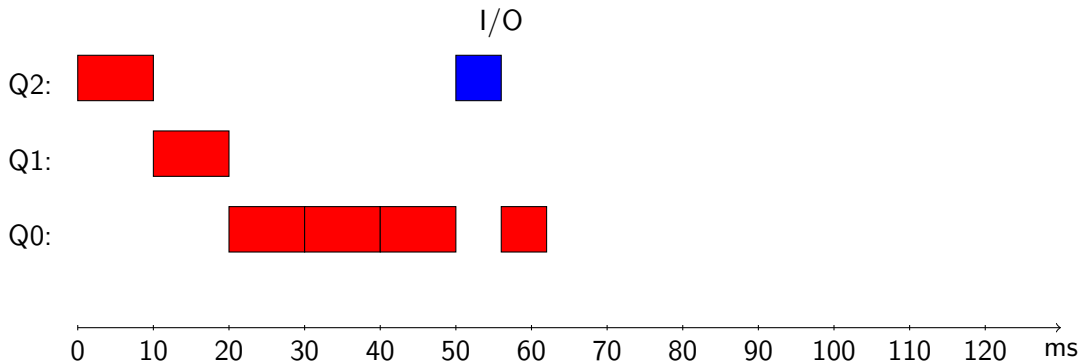
- Rule 5: after some time, move a job to the highest priority.



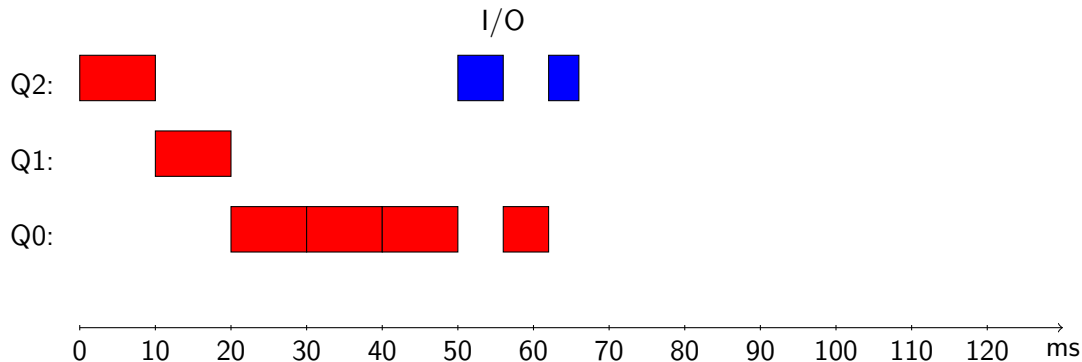
- Rule 5: after some time, move a job to the highest priority.



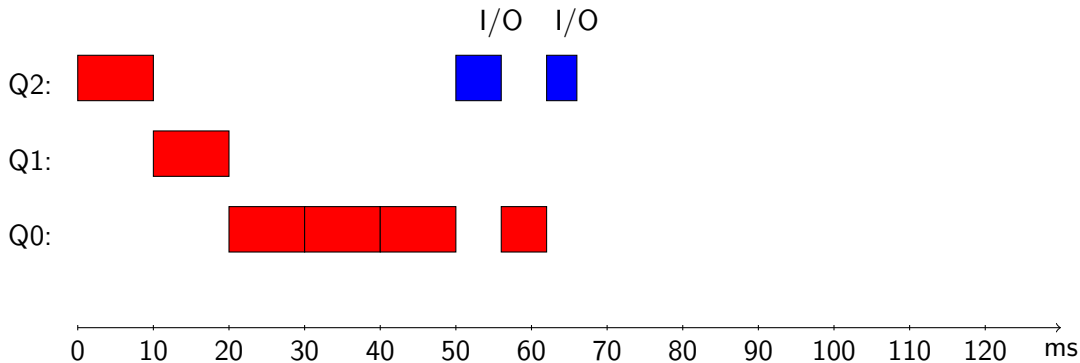
- Rule 5: after some time, move a job to the highest priority.



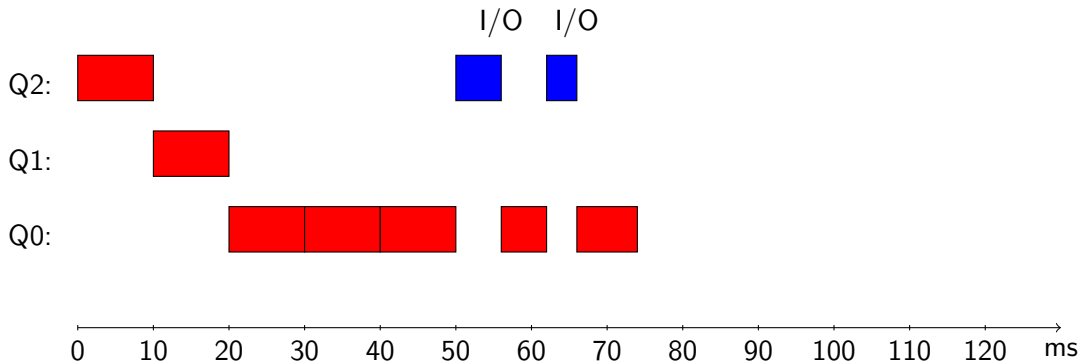
- Rule 5: after some time, move a job to the highest priority.



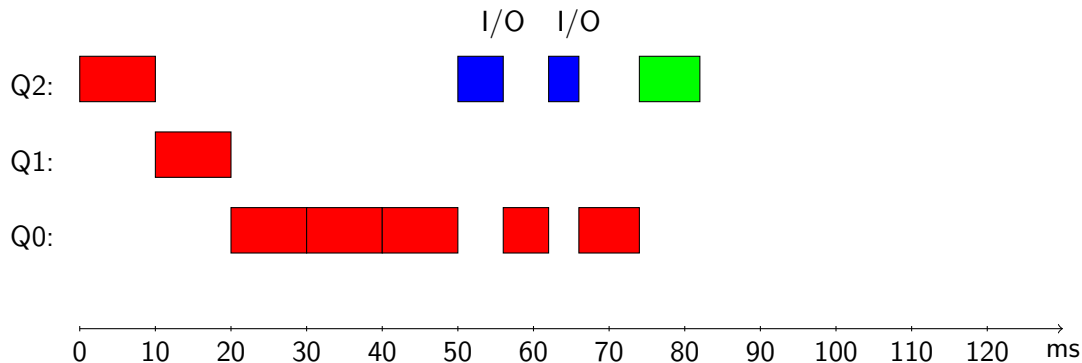
- Rule 5: after some time, move a job to the highest priority.



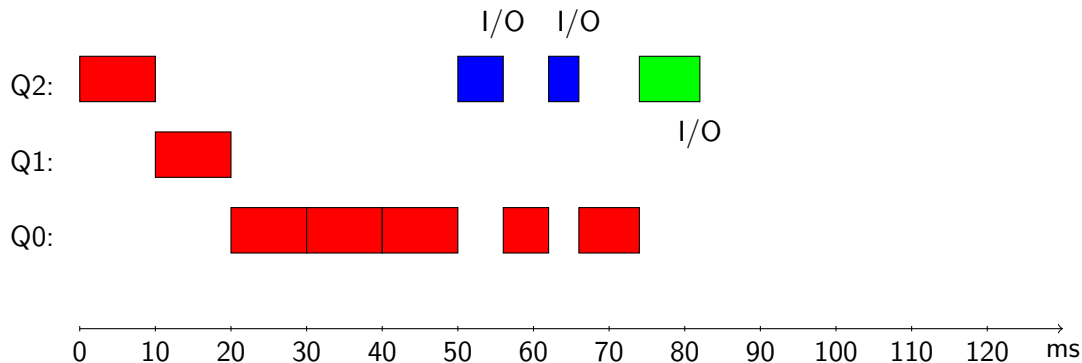
- Rule 5: after some time, move a job to the highest priority.



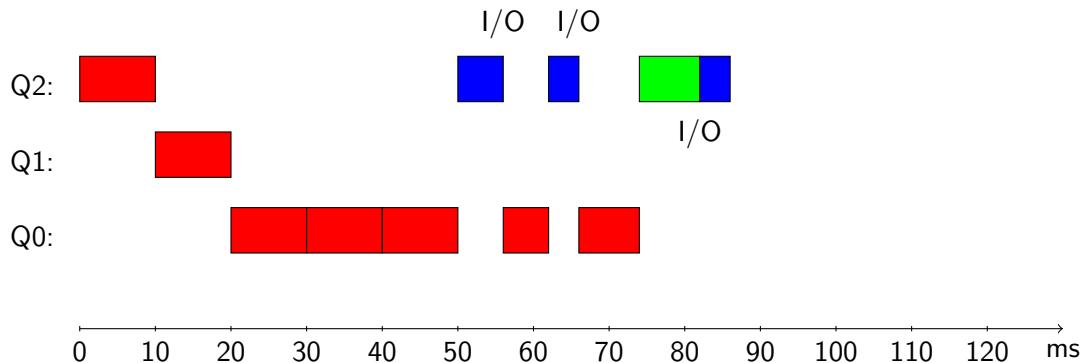
- Rule 5: after some time, move a job to the highest priority.



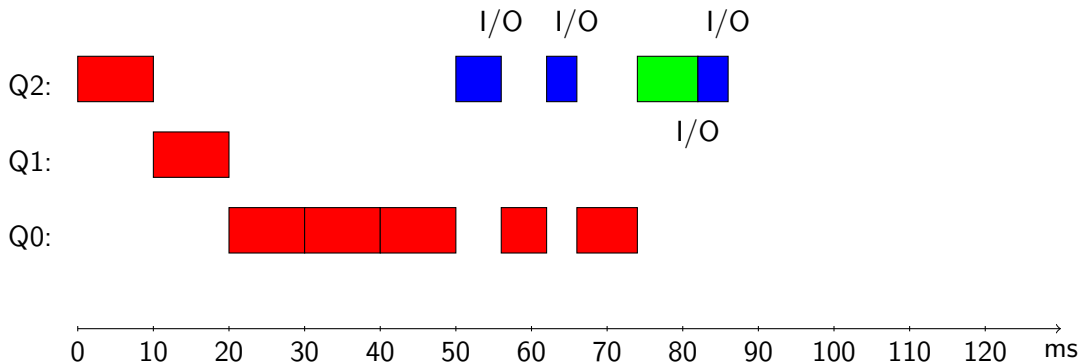
- Rule 5: after some time, move a job to the highest priority.



- Rule 5: after some time, move a job to the highest priority.

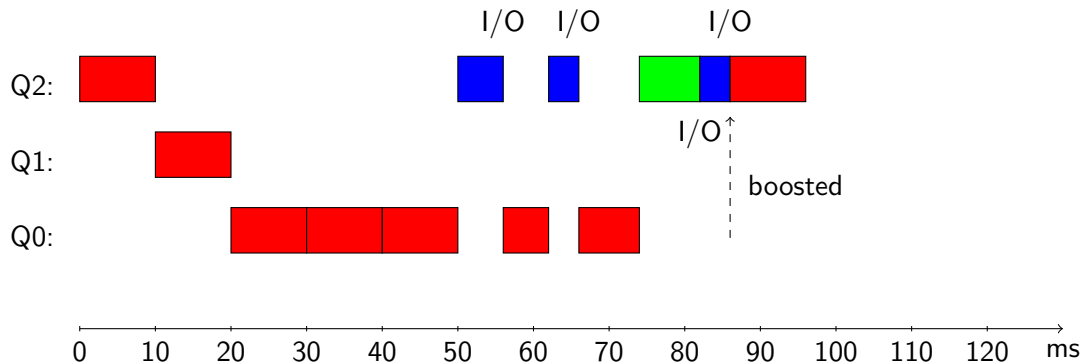


- Rule 5: after some time, move a job to the highest priority.

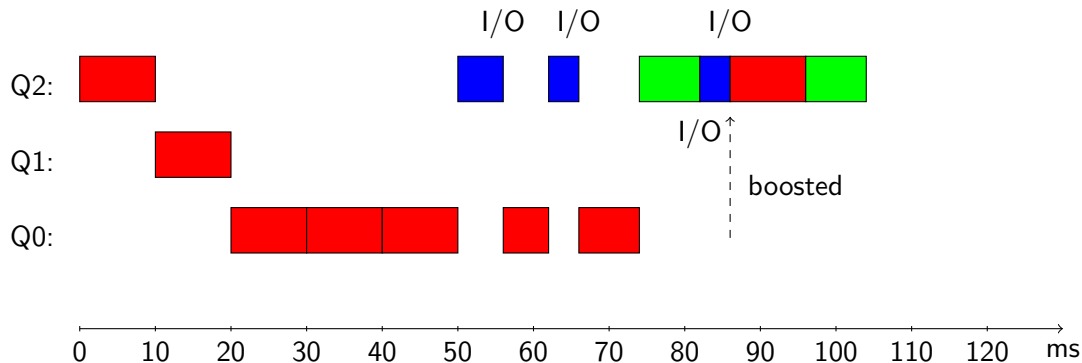


boost a job

- Rule 5: after some time, move a job to the highest priority.

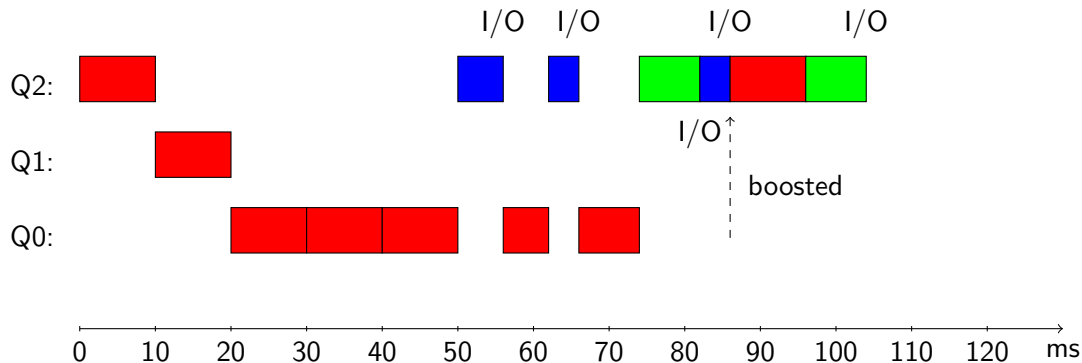


- Rule 5: after some time, move a job to the highest priority.

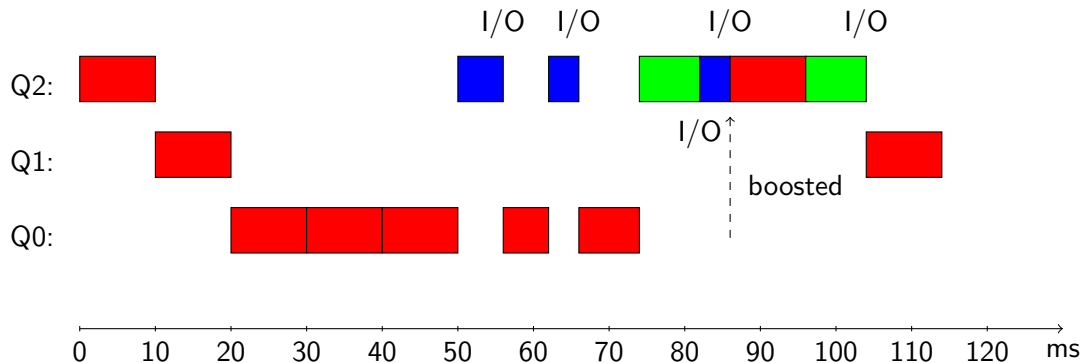


boost a job

- Rule 5: after some time, move a job to the highest priority.



- Rule 5: after some time, move a job to the highest priority.



trick the scheduler

If the scheduler was constructed given the rules 1-5, how would you write your program?

If the scheduler was constructed given the rules 1-5, how would you write your program?

- Rule 1: if $\text{Priority}(A) > \text{Priority}(B)$ then A is scheduled for execution.

If the scheduler was constructed given the rules 1-5, how would you write your program?

- Rule 1: if $\text{Priority}(A) > \text{Priority}(B)$ then A is scheduled for execution.
- Rule 2: if $\text{Priority}(A) = \text{Priority}(B)$ then A and B are scheduled in round-robin.

If the scheduler was constructed given the rules 1-5, how would you write your program?

- Rule 1: if $\text{Priority}(A) > \text{Priority}(B)$ then A is scheduled for execution.
- Rule 2: if $\text{Priority}(A) = \text{Priority}(B)$ then A and B are scheduled in round-robin.
- Rule 3: when a new job is created it starts with the highest priority.

If the scheduler was constructed given the rules 1-5, how would you write your program?

- Rule 1: if $\text{Priority}(A) > \text{Priority}(B)$ then A is scheduled for execution.
- Rule 2: if $\text{Priority}(A) = \text{Priority}(B)$ then A and B are scheduled in round-robin.
- Rule 3: when a new job is created it starts with the highest priority.
- Rule 4a: a job that has to be preempted (time-slice consumed) is moved to a lower priority.

If the scheduler was constructed given the rules 1-5, how would you write your program?

- Rule 1: if $\text{Priority}(A) > \text{Priority}(B)$ then A is scheduled for execution.
- Rule 2: if $\text{Priority}(A) = \text{Priority}(B)$ then A and B are scheduled in round-robin.
- Rule 3: when a new job is created it starts with the highest priority.
- Rule 4a: a job that has to be preempted (time-slice consumed) is moved to a lower priority.
- Rule 4b: a job that initiates a I/O-operation (or yields) remains on the same level.
- Rule 5: after some time, move a job to the highest priority.

A job is given a *allotted time*, to consume at each priority level.

A job is given a *allotted time*, to consume at each priority level.

- Rule 4: a job that has consumed its allotted time is moved to a lower priority.

A job is given a *allotted time*, to consume at each priority level.

- Rule 4: a job that has consumed its allotted time is moved to a lower priority.
- Rule 5: after some time, move all jobs to the highest priority.

Setting the parameters:

Setting the parameters:

- How long is a time slice?
- How many queues should there be?
- How long time should a allotted time be in a specified queue?
- How often should a job be boosted to the highest priority?

Change the perspective

What if:

Change the perspective

What if:

- we stop focusing on *turnaround time* and *reaction* and

Change the perspective

What if:

- we stop focusing on *turnaround time* and *reaction* and
- start treating every job in a fair manner.

Change the perspective

What if:

- we stop focusing on *turnaround time* and *reaction* and
- start treating every job in a fair manner.

Give each job fair share.

Justice for all



Proportional share

Let's have a lottery:

Proportional share

Let's have a lottery:



and the winner is

We divide the tickets among the jobs: A - 35 tickets, B - 15 tickets and C - 50 tickets.

and the winner is

We divide the tickets among the jobs: A - 35 tickets, B - 15 tickets and C - 50 tickets.

The scheduler selects a winning ticket by random.

and the winner is

We divide the tickets among the jobs: A - 35 tickets, B - 15 tickets and C - 50 tickets.

The scheduler selects a winning ticket by random.

And the winner is: 23, 56, 13, 73, 8, 82, 17, 34,

and the winner is

We divide the tickets among the jobs: A - 35 tickets, B - 15 tickets and C - 50 tickets.

The scheduler selects a winning ticket by random.

And the winner is: 23, 56, 13, 73, 8, 82, 17, 34,



- A new job can be given a set of tickets as long as we keep track of how many tickets we have.

- A new job can be given a set of tickets as long as we keep track of how many tickets we have.
- We can give a *user* a set of tickets and allow the user to distribute them among its jobs.

- A new job can be given a set of tickets as long as we keep track of how many tickets we have.
- We can give a *user* a set of tickets and allow the user to distribute them among its jobs.
- Each user can have its local tickets and then have a local lottery.

- A new job can be given a set of tickets as long as we keep track of how many tickets we have.
- We can give a *user* a set of tickets and allow the user to distribute them among its jobs.
- Each user can have its local tickets and then have a local lottery.
- We could allow each user to create new tickets, i.e. inflation, if we trust each other.

How to implement?

Stand in line

- Each job is given a number that represents the number of tickets it owns.
- All jobs are lined up in a row.
- Pick a random number from zero to the total number of tickets.
- Walk down the line and select the winner.



How does this work?

Why random?

A deterministic approach: stride scheduling

- Each job is given a *stride value*, the higher the stride the lower the priority.

A deterministic approach: stride scheduling

- Each job is given a *stride value*, the higher the stride the lower the priority.
- Each job keeps a *pass value* initially set to 0.

A deterministic approach: stride scheduling

- Each job is given a *stride value*, the higher the stride the lower the priority.
- Each job keeps a *pass value* initially set to 0.
- In each round the job with *the lowest pass value is selected* and ...

A deterministic approach: stride scheduling

- Each job is given a *stride value*, the higher the stride the lower the priority.
- Each job keeps a *pass value* initially set to 0.
- In each round the job with *the lowest pass value is selected* and ...
- ... the pass value is incremented by its stride value.

A deterministic approach: stride scheduling

- Each job is given a *stride value*, the higher the stride the lower the priority.
- Each job keeps a *pass value* initially set to 0.
- In each round the job with *the lowest pass value is selected* and ...
- ... the pass value is incremented by its stride value.

A low stride value will make it more likely to be scheduled soon again.

In real time scheduling we introduce a new requirement: things should be completed within a given time period.

In real time scheduling we introduce a new requirement: things should be completed within a given time period.

- Hard : all deadlines should be met, missing a deadline is a failure.

In real time scheduling we introduce a new requirement: things should be completed within a given time period.

- Hard : all deadlines should be met, missing a deadline is a failure.
- Soft : deadlines could be missed but the application should be notified and be able to take actions.

In real time scheduling we introduce a new requirement: things should be completed within a given time period.

- Hard : all deadlines should be met, missing a deadline is a failure.
- Soft : deadlines could be missed but the application should be notified and be able to take actions.

We often have real-time requirements that are simply met since we happen to have the available resources.

In hard real-time systems, *tasks* are known beforehand and described by a triplet $\langle e, d, p \rangle$

In hard real-time systems, *tasks* are known beforehand and described by a triplet $\langle e, d, p \rangle$

- e : the worst case execution time for the task.

In hard real-time systems, *tasks* are known beforehand and described by a triplet $\langle e, d, p \rangle$

- e : the worst case execution time for the task.
- d : the deadline, when in the future do we need to finish.

In hard real-time systems, *tasks* are known beforehand and described by a triplet $\langle e, d, p \rangle$

- e : the worst case execution time for the task.
- d : the deadline, when in the future do we need to finish.
- p : the period, how often should the task be scheduled.

Real-time scheduling

In hard real-time systems, *tasks* are known beforehand and described by a triplet $\langle e, d, p \rangle$

- e : the worst case execution time for the task.
- d : the deadline, when in the future do we need to finish.
- p : the period, how often should the task be scheduled.



Real-time scheduling

In hard real-time systems, *tasks* are known beforehand and described by a triplet $\langle e, d, p \rangle$

- e : the worst case execution time for the task.
- d : the deadline, when in the future do we need to finish.
- p : the period, how often should the task be scheduled.



Real-time scheduling

In hard real-time systems, *tasks* are known beforehand and described by a triplet $\langle e, d, p \rangle$

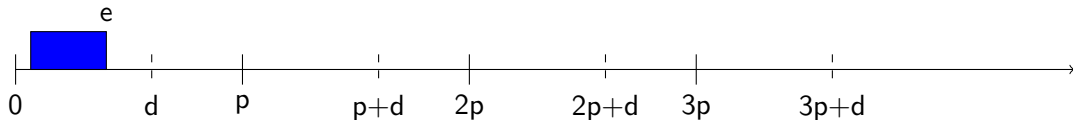
- e : the worst case execution time for the task.
- d : the deadline, when in the future do we need to finish.
- p : the period, how often should the task be scheduled.



Real-time scheduling

In hard real-time systems, *tasks* are known beforehand and described by a triplet $\langle e, d, p \rangle$

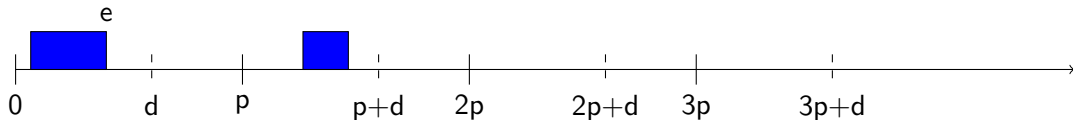
- e : the worst case execution time for the task.
- d : the deadline, when in the future do we need to finish.
- p : the period, how often should the task be scheduled.



Real-time scheduling

In hard real-time systems, *tasks* are known beforehand and described by a triplet $\langle e, d, p \rangle$

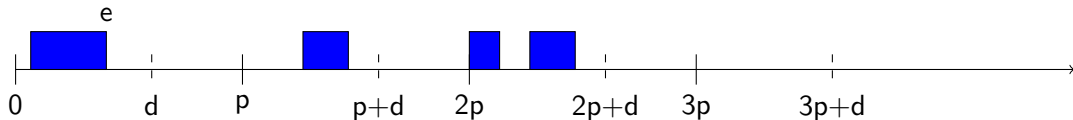
- e : the worst case execution time for the task.
- d : the deadline, when in the future do we need to finish.
- p : the period, how often should the task be scheduled.



Real-time scheduling

In hard real-time systems, *tasks* are known beforehand and described by a triplet $\langle e, d, p \rangle$

- e : the worst case execution time for the task.
- d : the deadline, when in the future do we need to finish.
- p : the period, how often should the task be scheduled.



$d < p$: constrained, $d = p$ default, $d > p$ several out-standing

Given a set of tasks: T1: $\langle 10, 30, 40 \rangle$, T2: $\langle 20, 60, 100 \rangle$, T3: $\langle 60, 200, 200 \rangle$: , find the scheduling.

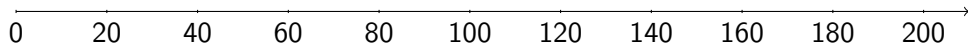
Real-time Scheduling

Given a set of tasks: T1: $\langle 10, 30, 40 \rangle$, T2: $\langle 20, 60, 100 \rangle$, T3: $\langle 60, 200, 200 \rangle$: , find the scheduling.

T1:

T2:

T3:



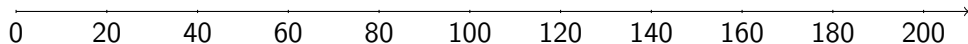
Real-time Scheduling

Given a set of tasks: T1: $\langle 10, 30, 40 \rangle$, T2: $\langle 20, 60, 100 \rangle$, T3: $\langle 60, 200, 200 \rangle$: , find the scheduling.

T1: 

T2:

T3:



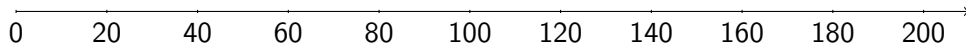
Real-time Scheduling

Given a set of tasks: T1: $\langle 10, 30, 40 \rangle$, T2: $\langle 20, 60, 100 \rangle$, T3: $\langle 60, 200, 200 \rangle$: , find the scheduling.

T1: 

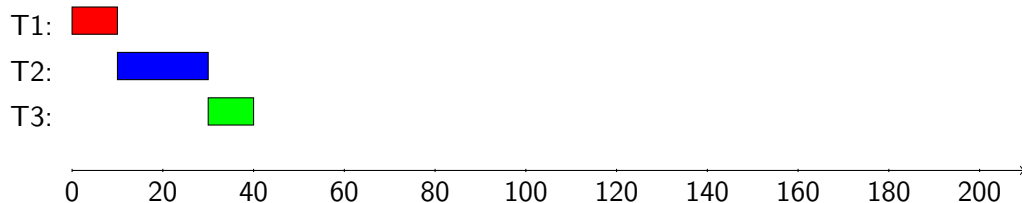
T2: 

T3:



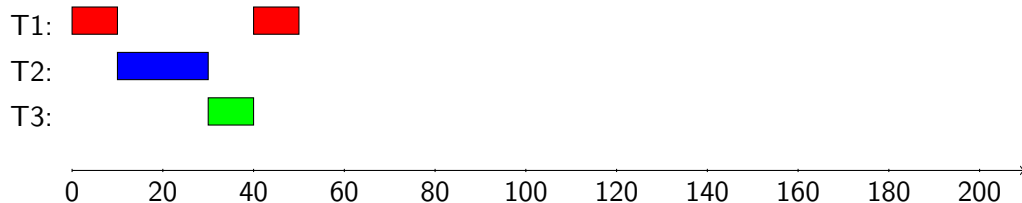
Real-time Scheduling

Given a set of tasks: T1: $\langle 10, 30, 40 \rangle$, T2: $\langle 20, 60, 100 \rangle$, T3: $\langle 60, 200, 200 \rangle$: , find the scheduling.



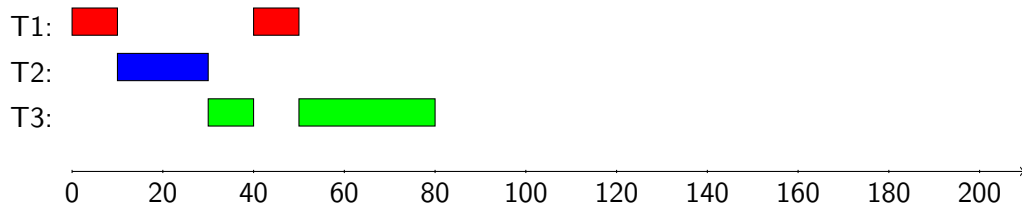
Real-time Scheduling

Given a set of tasks: T1: $\langle 10, 30, 40 \rangle$, T2: $\langle 20, 60, 100 \rangle$, T3: $\langle 60, 200, 200 \rangle$: , find the scheduling.



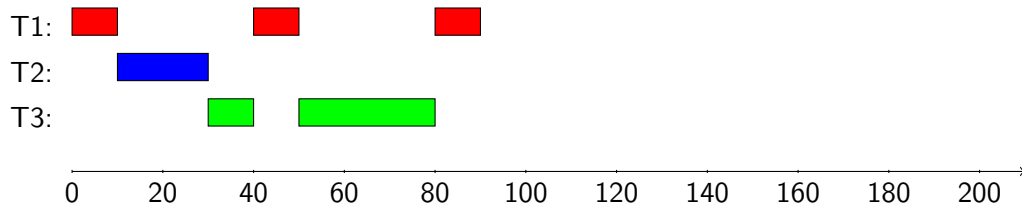
Real-time Scheduling

Given a set of tasks: T1: $\langle 10, 30, 40 \rangle$, T2: $\langle 20, 60, 100 \rangle$, T3: $\langle 60, 200, 200 \rangle$: , find the scheduling.



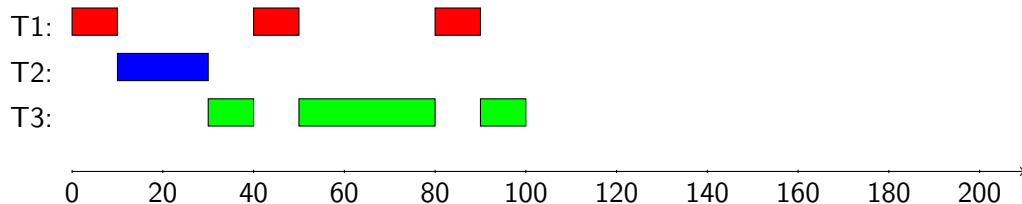
Real-time Scheduling

Given a set of tasks: T1: $\langle 10, 30, 40 \rangle$, T2: $\langle 20, 60, 100 \rangle$, T3: $\langle 60, 200, 200 \rangle$: , find the scheduling.



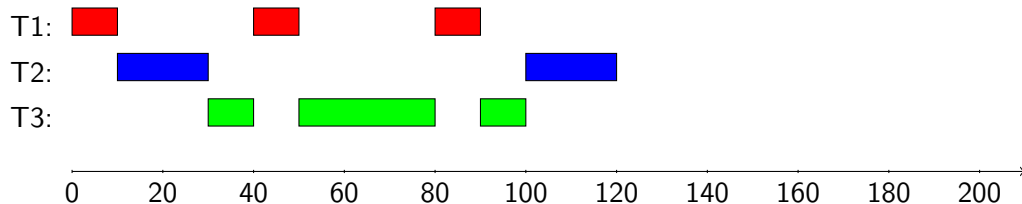
Real-time Scheduling

Given a set of tasks: T1: $\langle 10, 30, 40 \rangle$, T2: $\langle 20, 60, 100 \rangle$, T3: $\langle 60, 200, 200 \rangle$: , find the scheduling.



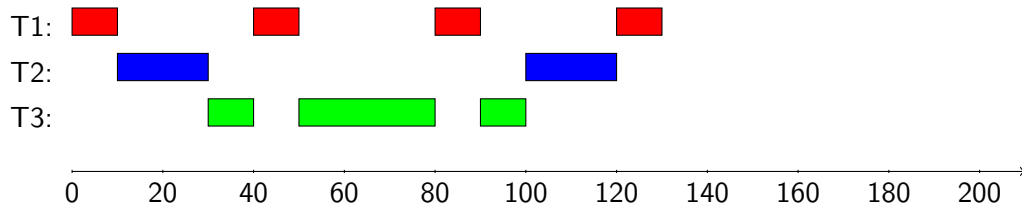
Real-time Scheduling

Given a set of tasks: T1: $\langle 10, 30, 40 \rangle$, T2: $\langle 20, 60, 100 \rangle$, T3: $\langle 60, 200, 200 \rangle$: , find the scheduling.



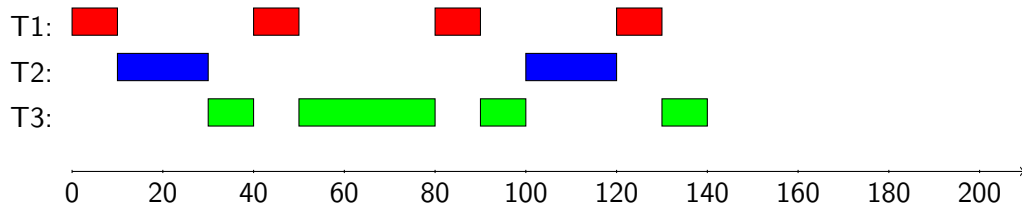
Real-time Scheduling

Given a set of tasks: T1: $\langle 10, 30, 40 \rangle$, T2: $\langle 20, 60, 100 \rangle$, T3: $\langle 60, 200, 200 \rangle$: , find the scheduling.



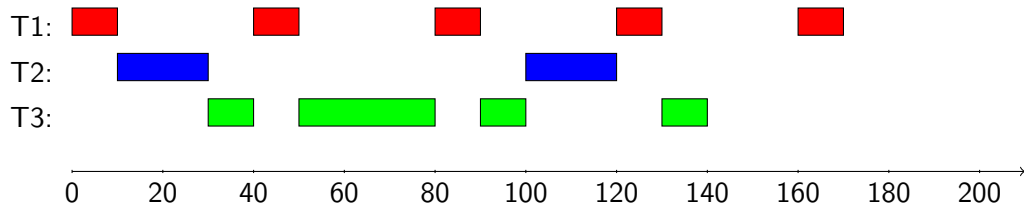
Real-time Scheduling

Given a set of tasks: T1: $\langle 10, 30, 40 \rangle$, T2: $\langle 20, 60, 100 \rangle$, T3: $\langle 60, 200, 200 \rangle$: , find the scheduling.



Real-time Scheduling

Given a set of tasks: T1: $\langle 10, 30, 40 \rangle$, T2: $\langle 20, 60, 100 \rangle$, T3: $\langle 60, 200, 200 \rangle$: , find the scheduling.



Given that $p = d$ i.e. a task must be completed within its period.

- Rate Monotonic Scheduling (RMS):
 - Schedule the available task with the shortest period
 - Always works if utilization is $< 69\%$ (actually less than $n * (2^{1/n} - 1)$), where n is the number of processes) could work for higher loads.
 - Simpler to reason about, easy to implement.

Given that $p = d$ i.e. a task must be completed within its period.

- Rate Monotonic Scheduling (RMS):
 - Schedule the available task with the shortest period
 - Always works if utilization is $< 69\%$ (actually less than $n * (2^{1/n} - 1)$), where n is the number of processes) could work for higher loads.
 - Simpler to reason about, easy to implement.
- Earliest Deadline First (EDF):
 - Schedule based on the deadline, more freedom to choose tasks.
 - Always works if utilization is $< 100\%$.
 - Used by Linux in the real-time extension (not in the regular system)

Given that $p = d$ i.e. a task must be completed within its period.

- Rate Monotonic Scheduling (RMS):
 - Schedule the available task with the shortest period
 - Always works if utilization is $< 69\%$ (actually less than $n * (2^{1/n} - 1)$), where n is the number of processes) could work for higher loads.
 - Simpler to reason about, easy to implement.
- Earliest Deadline First (EDF):
 - Schedule based on the deadline, more freedom to choose tasks.
 - Always works if utilization is $< 100\%$.
 - Used by Linux in the real-time extension (not in the regular system)

Rate Monotonic Scheduling (RMS)

Assume we have tasks: T1: $\langle 10, 40, 40 \rangle$, T2: $\langle 20, 60, 60 \rangle$, T3: $\langle 30, 80, 80 \rangle$.

Rate Monotonic Scheduling (RMS)

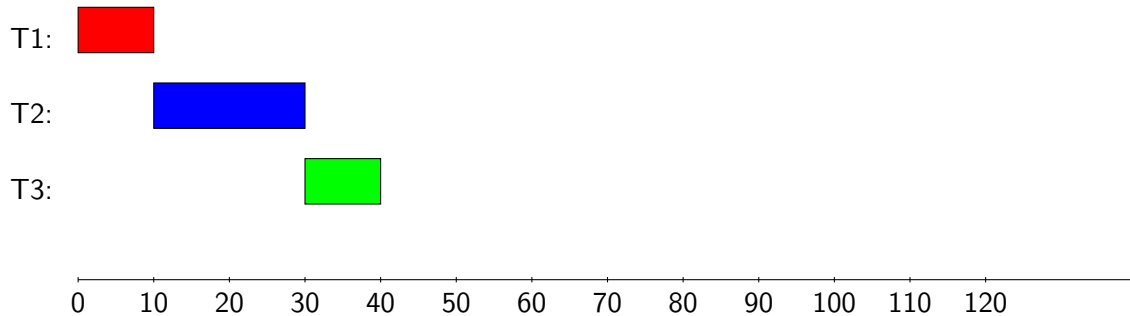
Assume we have tasks: T1: $\langle 10, 40, 40 \rangle$, T2: $\langle 20, 60, 60 \rangle$, T3: $\langle 30, 80, 80 \rangle$.

$$10/40 + 20/60 + 30/80 = 6/24 + 8/24 + 9/24 = 23/24$$

Rate Monotonic Scheduling (RMS)

Assume we have tasks: T1: $\langle 10, 40, 40 \rangle$, T2: $\langle 20, 60, 60 \rangle$, T3: $\langle 30, 80, 80 \rangle$.

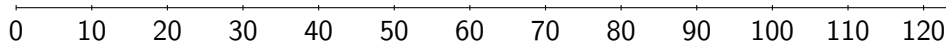
$$10/40 + 20/60 + 30/80 = 6/24 + 8/24 + 9/24 = 23/24$$



Rate Monotonic Scheduling (RMS)

Assume we have tasks: T1: $\langle 10, 40, 40 \rangle$, T2: $\langle 20, 60, 60 \rangle$, T3: $\langle 30, 80, 80 \rangle$.

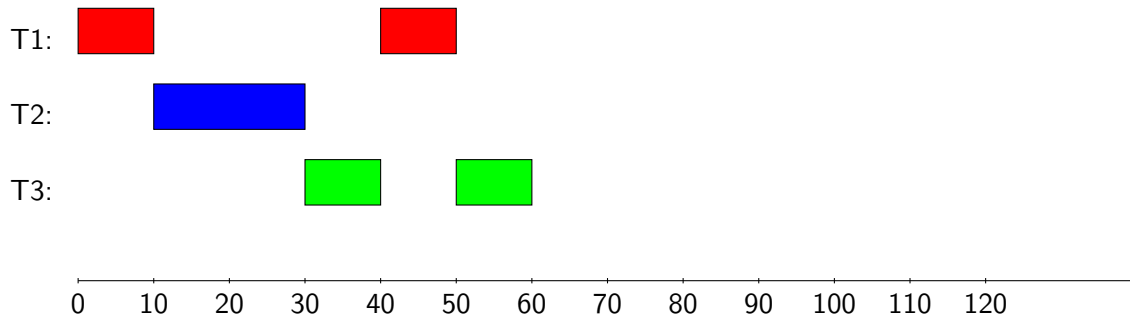
$$10/40 + 20/60 + 30/80 = 6/24 + 8/24 + 9/24 = 23/24$$



Rate Monotonic Scheduling (RMS)

Assume we have tasks: T1: $\langle 10, 40, 40 \rangle$, T2: $\langle 20, 60, 60 \rangle$, T3: $\langle 30, 80, 80 \rangle$.

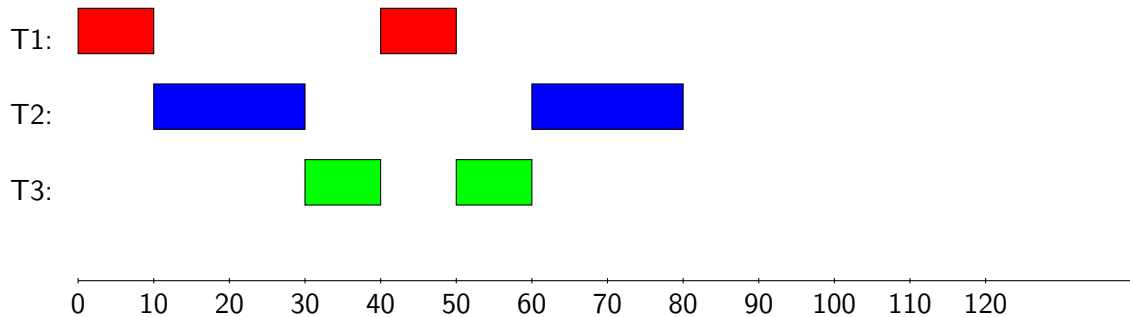
$$10/40 + 20/60 + 30/80 = 6/24 + 8/24 + 9/24 = 23/24$$



Rate Monotonic Scheduling (RMS)

Assume we have tasks: T1: $\langle 10, 40, 40 \rangle$, T2: $\langle 20, 60, 60 \rangle$, T3: $\langle 30, 80, 80 \rangle$.

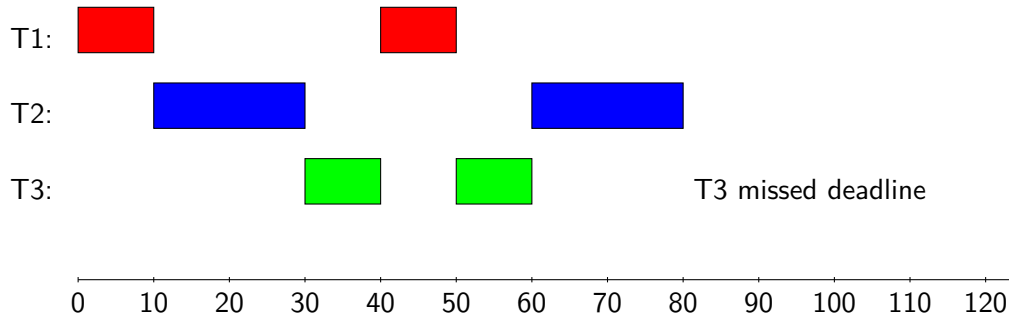
$$10/40 + 20/60 + 30/80 = 6/24 + 8/24 + 9/24 = 23/24$$



Rate Monotonic Scheduling (RMS)

Assume we have tasks: T1: $\langle 10, 40, 40 \rangle$, T2: $\langle 20, 60, 60 \rangle$, T3: $\langle 30, 80, 80 \rangle$.

$$10/40 + 20/60 + 30/80 = 6/24 + 8/24 + 9/24 = 23/24$$



Earliest Deadline First (EDF)

Assume we have tasks: T1: $\langle 10, 40, 40 \rangle$, T2: $\langle 20, 60, 60 \rangle$, T3: $\langle 30, 80, 80 \rangle$.

Earliest Deadline First (EDF)

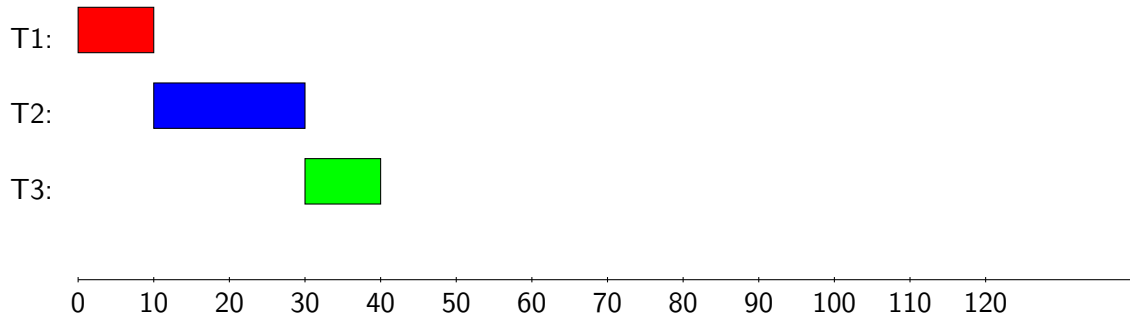
Assume we have tasks: T1: $\langle 10, 40, 40 \rangle$, T2: $\langle 20, 60, 60 \rangle$, T3: $\langle 30, 80, 80 \rangle$.

$$10/40 + 20/60 + 30/80 = 6/24 + 8/24 + 9/24 = 23/24$$

Earliest Deadline First (EDF)

Assume we have tasks: T1: $\langle 10, 40, 40 \rangle$, T2: $\langle 20, 60, 60 \rangle$, T3: $\langle 30, 80, 80 \rangle$.

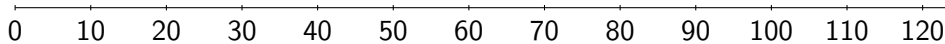
$$10/40 + 20/60 + 30/80 = 6/24 + 8/24 + 9/24 = 23/24$$



Earliest Deadline First (EDF)

Assume we have tasks: T1: $\langle 10, 40, 40 \rangle$, T2: $\langle 20, 60, 60 \rangle$, T3: $\langle 30, 80, 80 \rangle$.

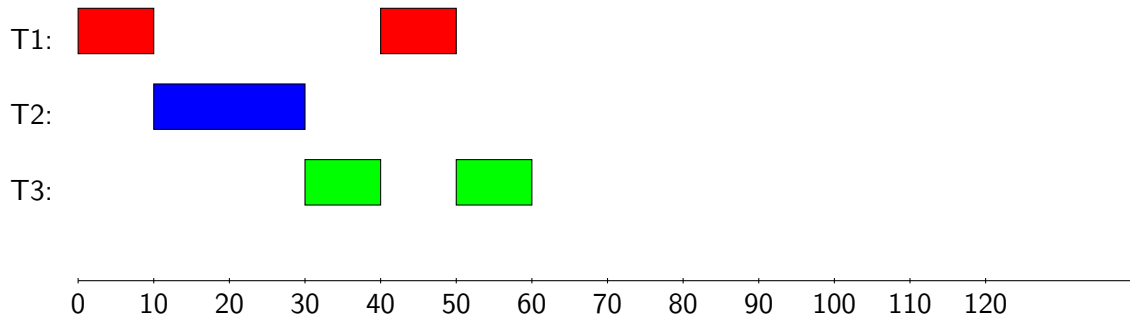
$$10/40 + 20/60 + 30/80 = 6/24 + 8/24 + 9/24 = 23/24$$



Earliest Deadline First (EDF)

Assume we have tasks: T1: $\langle 10, 40, 40 \rangle$, T2: $\langle 20, 60, 60 \rangle$, T3: $\langle 30, 80, 80 \rangle$.

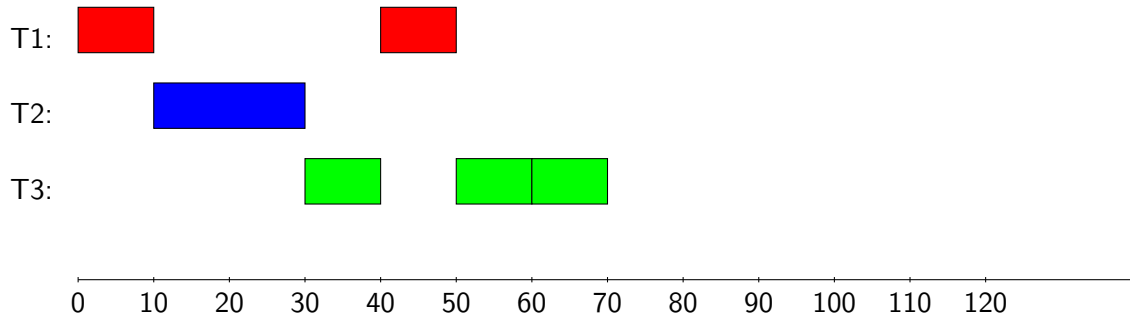
$$10/40 + 20/60 + 30/80 = 6/24 + 8/24 + 9/24 = 23/24$$



Earliest Deadline First (EDF)

Assume we have tasks: T1: $\langle 10, 40, 40 \rangle$, T2: $\langle 20, 60, 60 \rangle$, T3: $\langle 30, 80, 80 \rangle$.

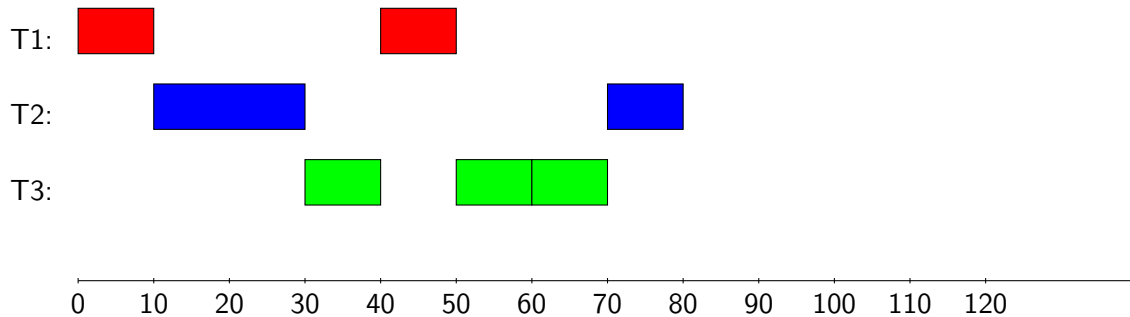
$$10/40 + 20/60 + 30/80 = 6/24 + 8/24 + 9/24 = 23/24$$



Earliest Deadline First (EDF)

Assume we have tasks: T1: $\langle 10, 40, 40 \rangle$, T2: $\langle 20, 60, 60 \rangle$, T3: $\langle 30, 80, 80 \rangle$.

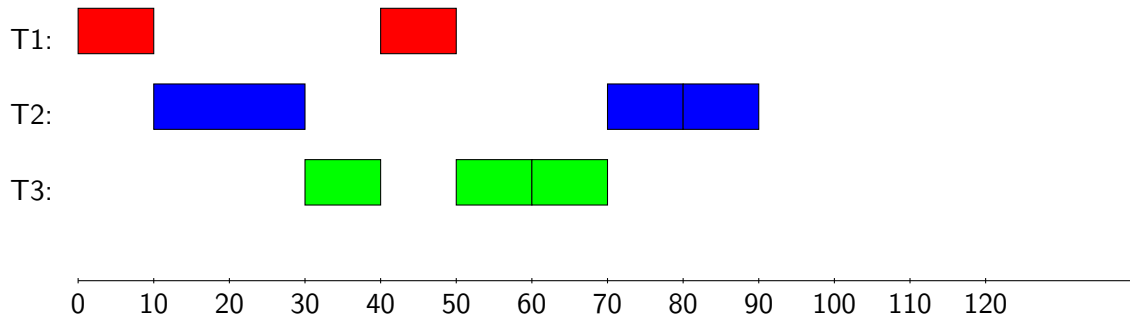
$$10/40 + 20/60 + 30/80 = 6/24 + 8/24 + 9/24 = 23/24$$



Earliest Deadline First (EDF)

Assume we have tasks: T1: $\langle 10, 40, 40 \rangle$, T2: $\langle 20, 60, 60 \rangle$, T3: $\langle 30, 80, 80 \rangle$.

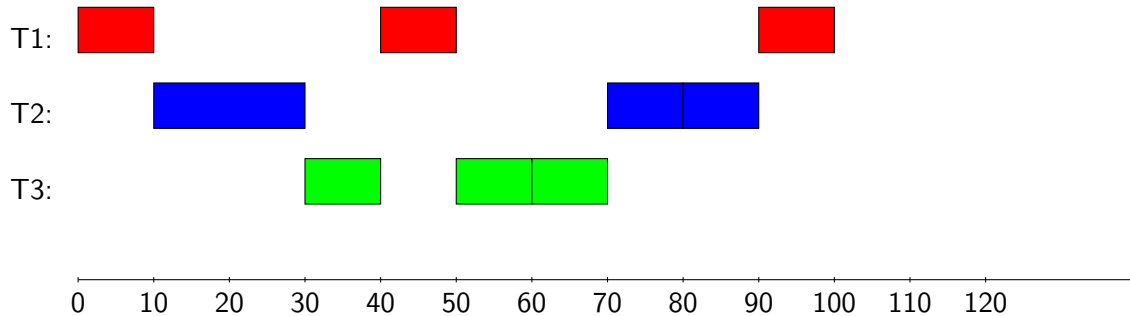
$$10/40 + 20/60 + 30/80 = 6/24 + 8/24 + 9/24 = 23/24$$



Earliest Deadline First (EDF)

Assume we have tasks: T1: $\langle 10, 40, 40 \rangle$, T2: $\langle 20, 60, 60 \rangle$, T3: $\langle 30, 80, 80 \rangle$.

$$10/40 + 20/60 + 30/80 = 6/24 + 8/24 + 9/24 = 23/24$$



With what accuracy can we determine worst case execution time?

With what accuracy can we determine worst case execution time?

Should we be conservative or take a chance?

With what accuracy can we determine worst case execution time?

Should we be conservative or take a chance?

Can we handle a dynamic set of tasks?

With what accuracy can we determine worst case execution time?

Should we be conservative or take a chance?

Can we handle a dynamic set of tasks?

What happens when we have critical resources that are protected by locks?

Scheduling for a multi-core architecture more problematic (or rather more problematic to achieve high utilization).

Why?

How is scheduling managed in a Linux system?

How is scheduling managed in a Linux system?

- $O(n)$ scheduler: the original scheduler, did not scale well.

How is scheduling managed in a Linux system?

- $O(n)$ scheduler: the original scheduler, did not scale well.
- $O(1)$ scheduler: multi-level feedback queues, dynamic priority, used up to version 2.6

How is scheduling managed in a Linux system?

- $O(n)$ scheduler: the original scheduler, did not scale well.
- $O(1)$ scheduler: multi-level feedback queues, dynamic priority, used up to version 2.6
- CFS: the *completely fair scheduler*, $O(\lg(n))$, default today.

How is scheduling managed in a Linux system?

- $O(n)$ scheduler: the original scheduler, did not scale well.
- $O(1)$ scheduler: multi-level feedback queues, dynamic priority, used up to version 2.6
- CFS: the *completely fair scheduler*, $O(\lg(n))$, default today.
- BF scheduler: no I will not tell you what it stands for.

The Completely Fair Scheduler

- Similar to stride scheduler but uses a red-black tree to order processes.

The Completely Fair Scheduler

- Similar to stride scheduler but uses a red-black tree to order processes.
- Will keep processes on the same core if it thinks it's a good choice.

The Completely Fair Scheduler

- Similar to stride scheduler but uses a red-black tree to order processes.
- Will keep processes on the same core if it thinks it's a good choice.
- Scheduling classes:

The Completely Fair Scheduler

- Similar to stride scheduler but uses a red-black tree to order processes.
- Will keep processes on the same core if it thinks it's a good choice.
- Scheduling classes:
 - SCHED_FIFO, SCHED_RR: high priority classes (often called real-time processes)

The Completely Fair Scheduler

- Similar to stride scheduler but uses a red-black tree to order processes.
- Will keep processes on the same core if it thinks it's a good choice.
- Scheduling classes:
 - SCHED_FIFO, SCHED_RR: high priority classes (often called real-time processes)
 - SCHED_NORMAL: all the regular interactive processes

The Completely Fair Scheduler

- Similar to stride scheduler but uses a red-black tree to order processes.
- Will keep processes on the same core if it thinks it's a good choice.
- Scheduling classes:
 - SCHED_FIFO, SCHED_RR: high priority classes (often called real-time processes)
 - SCHED_NORMAL: all the regular interactive processes
 - SCHED_BATCH: processes that only run if there are no interactive processes available.

The Completely Fair Scheduler

- Similar to stride scheduler but uses a red-black tree to order processes.
- Will keep processes on the same core if it thinks it's a good choice.
- Scheduling classes:
 - `SCHED_FIFO`, `SCHED_RR`: high priority classes (often called real-time processes)
 - `SCHED_NORMAL`: all the regular interactive processes
 - `SCHED_BATCH`: processes that only run if there are no interactive processes available.
 - `SCHED_IDLE`: if we've got nothing else to do.

The Completely Fair Scheduler

- Similar to stride scheduler but uses a red-black tree to order processes.
- Will keep processes on the same core if it thinks it's a good choice.
- Scheduling classes:
 - `SCHED_FIFO`, `SCHED_RR`: high priority classes (often called real-time processes)
 - `SCHED_NORMAL`: all the regular interactive processes
 - `SCHED_BATCH`: processes that only run if there are no interactive processes available.
 - `SCHED_IDLE`: if we've got nothing else to do.

Summary Scheduling

- Bonnie Tyler: Turnaround, every now and then ...

Summary Scheduling

- Bonnie Tyler: Turnaround, every now and then ...
- Bob Marley: Talking 'bout reaction

Summary Scheduling

- Bonnie Tyler: Turnaround, every now and then ...
- Bob Marley: Talking 'bout reaction
- Rolling Stones: You can't always get what you want.

Summary Scheduling

- Bonnie Tyler: Turnaround, every now and then ...
- Bob Marley: Talking 'bout reaction
- Rolling Stones: You can't always get what you want.
- Metallica: Justice for all.

Summary Scheduling

- Bonnie Tyler: Turnaround, every now and then ...
- Bob Marley: Talking 'bout reaction
- Rolling Stones: You can't always get what you want.
- Metallica: Justice for all.
- Leif “Loket” Olsson: a lottery might work ok

Summary Scheduling

- Bonnie Tyler: Turnaround, every now and then ...
- Bob Marley: Talking 'bout reaction
- Rolling Stones: You can't always get what you want.
- Metallica: Justice for all.
- Leif “Loket” Olsson: a lottery might work ok
- Real-time scheduling: if we actually know the maximum execution time, the deadline and the period.

Summary Scheduling

- Bonnie Tyler: Turnaround, every now and then ...
- Bob Marley: Talking 'bout reaction
- Rolling Stones: You can't always get what you want.
- Metallica: Justice for all.
- Leif “Loket” Olsson: a lottery might work ok
- Real-time scheduling: if we actually know the maximum execution time, the deadline and the period.
- Multi-core schedulers: you have to think twice before selecting a process.

Summary Scheduling

- Bonnie Tyler: Turnaround, every now and then ...
- Bob Marley: Talking 'bout reaction
- Rolling Stones: You can't always get what you want.
- Metallica: Justice for all.
- Leif “Loket” Olsson: a lottery might work ok
- Real-time scheduling: if we actually know the maximum execution time, the deadline and the period.
- Multi-core schedulers: you have to think twice before selecting a process.
- Linux: Completely Fair Scheduler, schedules in $O(\lg(n))$ time, similar to stride scheduling.