Batch and Gang Scheduling

Arnaud Legrand, CNRS, University of Grenoble

LIG laboratory, arnaud.legrand@imag.fr

March 25, 2009

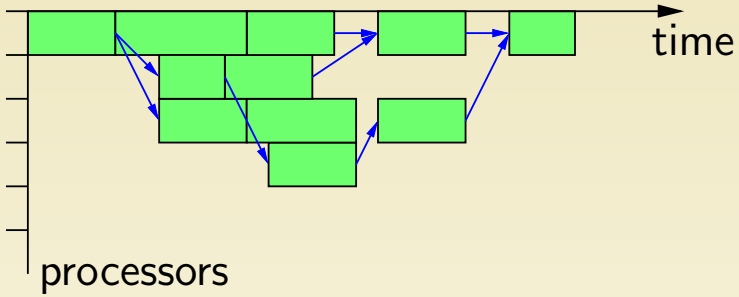
Outline

Outline

Outline

Need for Batch Scheduling

- Parallel Tasks from Scientific Computations (simulation, medical)

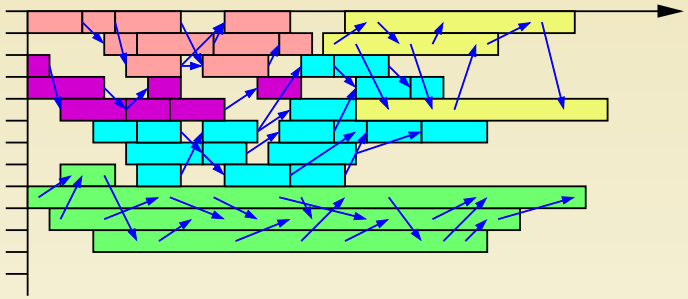


Need for Batch Scheduling

- Parallel Tasks from Scientific Computations (simulation, medical)

Need for Batch Scheduling

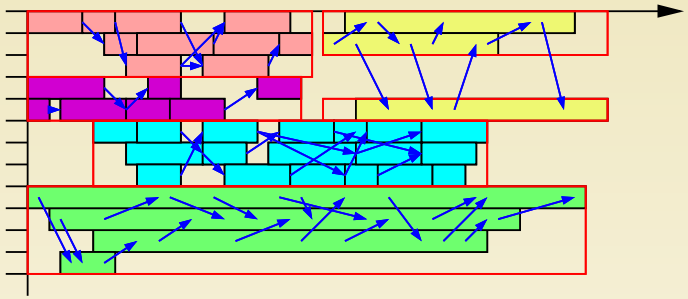
- Parallel Tasks from Scientific Computations (simulation, medical)



- When one purchases a cluster, typically many users want to use it.
  - One cannot let them step on each other's toes
  - Every user wants to be on a dedicated machine
  - Applications are written assuming some amount of RAM, some notion that all processors go at the same speed, etc.

Need for Batch Scheduling

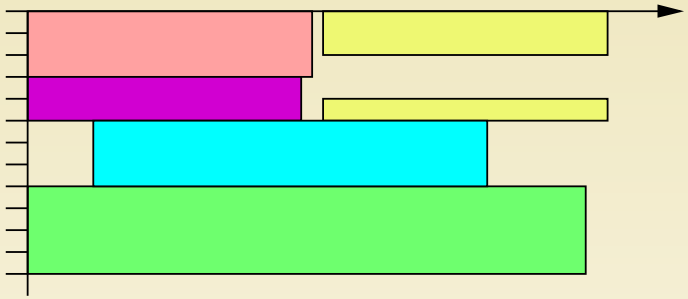
- Parallel Tasks from Scientific Computations (simulation, medical)



- When one purchases a cluster, typically many users want to use it.
  - One cannot let them step on each other's toes
  - Every user wants to be on a dedicated machine
  - Applications are written assuming some amount of RAM, some notion that all processors go at the same speed, etc.

Need for Batch Scheduling

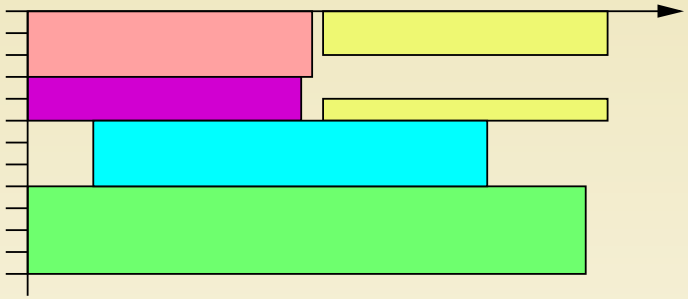
- Parallel Tasks from Scientific Computations (simulation, medical)



- When one purchases a cluster, typically many users want to use it.
  - One cannot let them step on each other's toes
  - Every user wants to be on a dedicated machine
  - Applications are written assuming some amount of RAM, some notion that all processors go at the same speed, etc.

Need for Batch Scheduling

- Parallel Tasks from Scientific Computations (simulation, medical)



- When one purchases a cluster, typically many users want to use it.
  - One cannot let them step on each other's toes
  - Every user wants to be on a dedicated machine
  - Applications are written assuming some amount of RAM, some notion that all processors go at the same speed, etc.

  The Job Scheduler is the entity that prevents them from stepping on each other's toes

  The Job Scheduler gives out nodes to applications

Batch Scheduling

Each job is defined as a Number of nodes (q_i) and a Time (p_i):

I want 6 nodes for 1h

Typically users are "charged" against an "allocation": e.g. "*You only get 100 CPU hours per week*".
A batch scheduler is a central middleware to manage resources (e.g. processors) of parallel machines:

- accept jobs (computing tasks) submitted by users
- decide when and where jobs are executed
- start jobs execution

They take into account:

- unavailability of some nodes
- users jobs mutual exclusion
- specific needs for jobs (memory, network, ...)

While trying to :

- maximize resources usage
- be fair among users

Batch Scheduling

Typical wanted features:

- Interactive mode
- Batch mode
- Parallel jobs support
- Multi-queues with priorities
- Admission policies (limit on usage, notions of user groups, power users)
- Resources matching
- File staging
- Jobs dependences
- Backfilling
- Reservations
- Best effort jobs
- Environment reconfiguration

There are many existing batch schedulers : LSF, PBS/Torque, Maui scheduler, Sun Grid Engine, EASY, OAR, ...

These are complex systems with many config options !

Main Batch Schedulers Features

	OpenPBS	SGE	Maui Scheduler (+ OpenPBS)	OAR
Interactive mode	×	×	×	×
Batch mode	×	×	×	×
Parallel jobs support	×	×	×	×
Multi-queues with priorities	×	×	×	×
Resources matching	×	×	×	×
Admission policies	×	×	×	×
File staging	×	×	×	
Jobs dependences	×	×	×	
Backfilling			×	×
Reservations			×	×
Best effort jobs				×
Environment reconfiguration				×
Fair sharing			×	×

Outline

List Scheduling

When simple problems are hard, we should try to find good approximation heuristics. A ϱ-approximation is an algorithm whose output is never more than a factor ϱ times the optimum solution.
Natural idea: using greedy strategy like trying to allocate the most possible task at a given time-step. However at some point we may face a choice (when there is more ready tasks than available processors).

List Scheduling

When simple problems are hard, we should try to find good approximation heuristics. A ϱ-approximation is an algorithm whose output is never more than a factor ϱ times the optimum solution.
Natural idea: using greedy strategy like trying to allocate the most possible task at a given time-step. However at some point we may face a choice (when there is more ready tasks than available processors).
Any strategy that does not let on purpose a processor idle is efficient [Coffman76]. Such a schedule is called list-schedule.

Theorem 1: Coffman.

Let $G = (V, E, w)$ be a DAG of sequential tasks, p the number of processors, and σ_p a list-schedule of G on p processors.

$$C_{\max}(\sigma_p) \leqslant \left(2 - \frac{1}{p}\right) C^*_{\max}(p) \ .$$

List Scheduling

When simple problems are hard, we should try to find good approximation heuristics. A ϱ-approximation is an algorithm whose output is never more than a factor ϱ times the optimum solution.
Natural idea: using greedy strategy like trying to allocate the most possible task at a given time-step. However at some point we may face a choice (when there is more ready tasks than available processors).
Any strategy that does not let on purpose a processor idle is efficient [Coffman76]. Such a schedule is called list-schedule.

Theorem 1: Coffman.

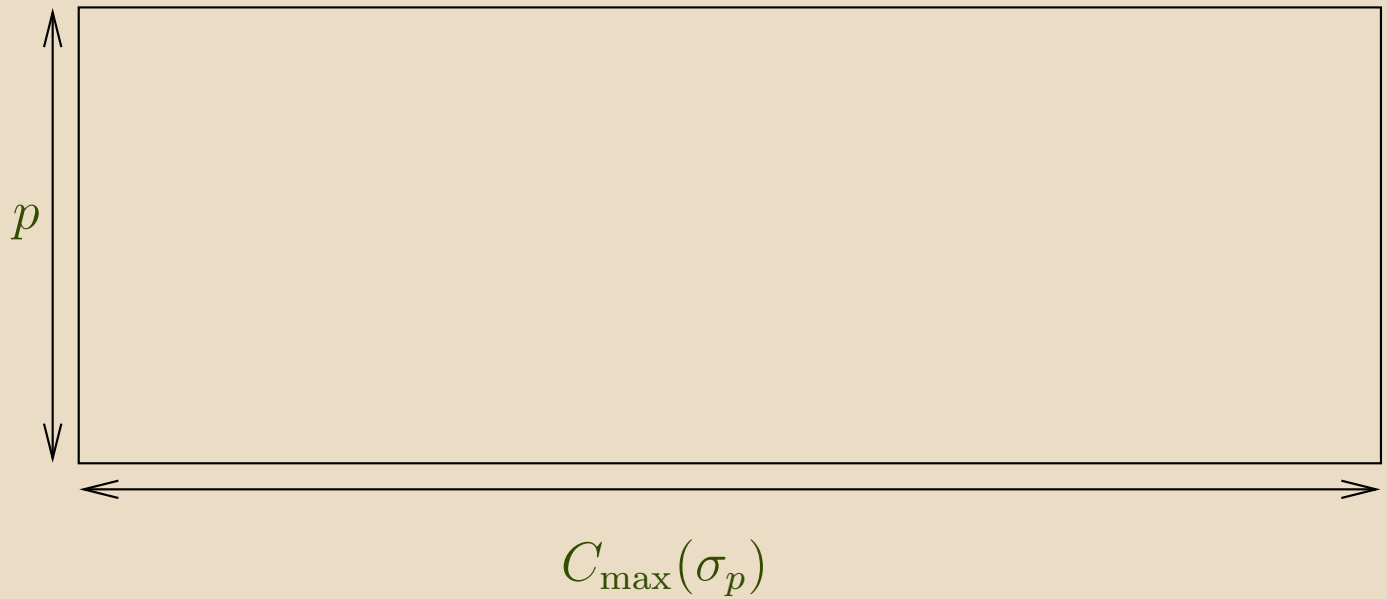
Let $G = (V, E, w)$ be a DAG of sequential tasks, p the number of processors, and σ_p a list-schedule of G on p processors.

$$C_{\max}(\sigma_p) \leqslant \left(2 - \frac{1}{p}\right) C^*_{\max}(p) \ .$$

Most of the time, list-heuristics are based on the critical path.

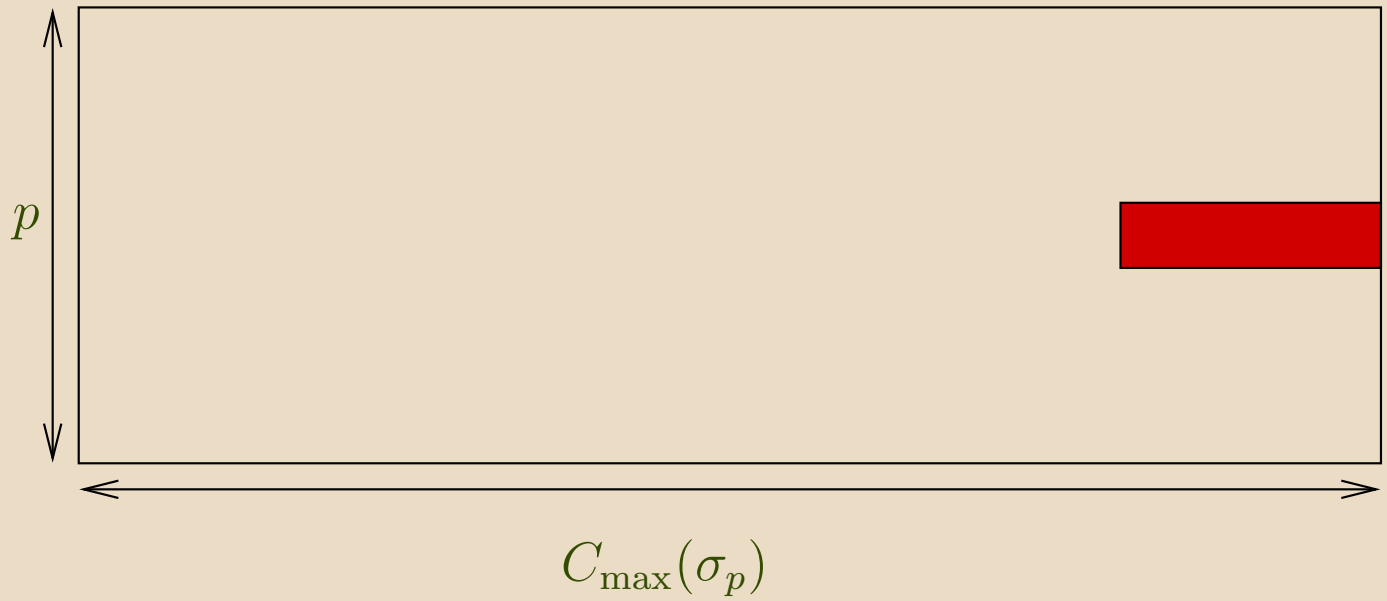
List Scheduling: proving the Coffman result

Proof.



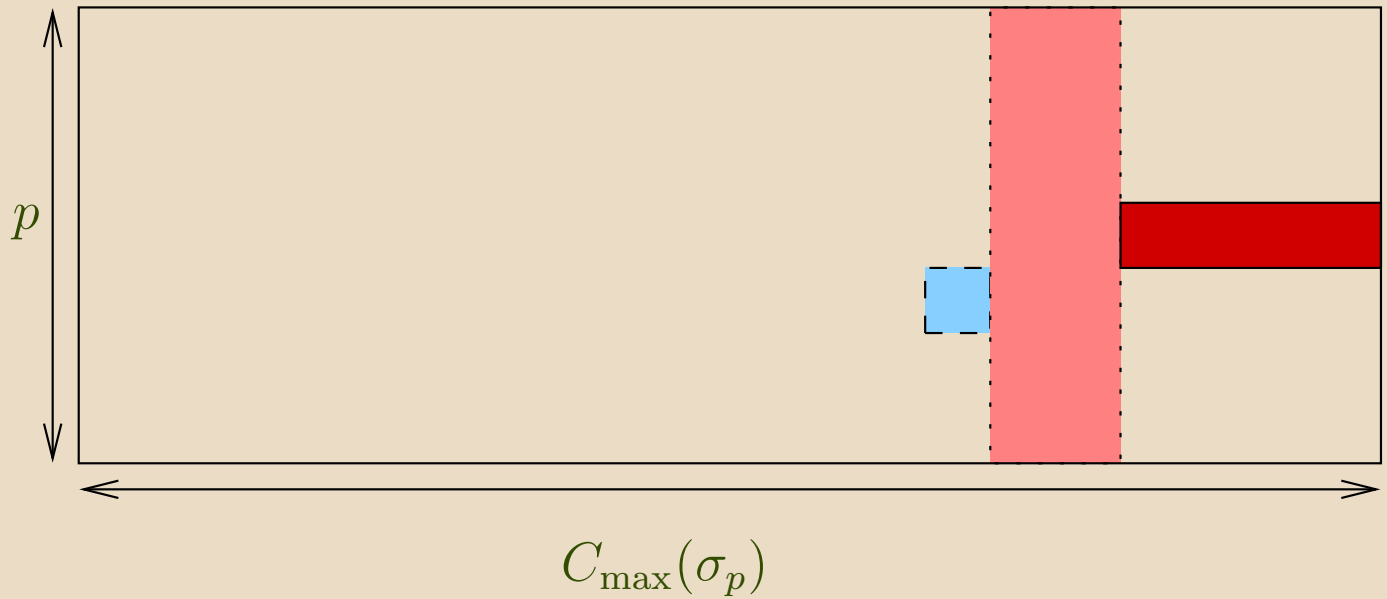
List Scheduling: proving the Coffman result

Proof.



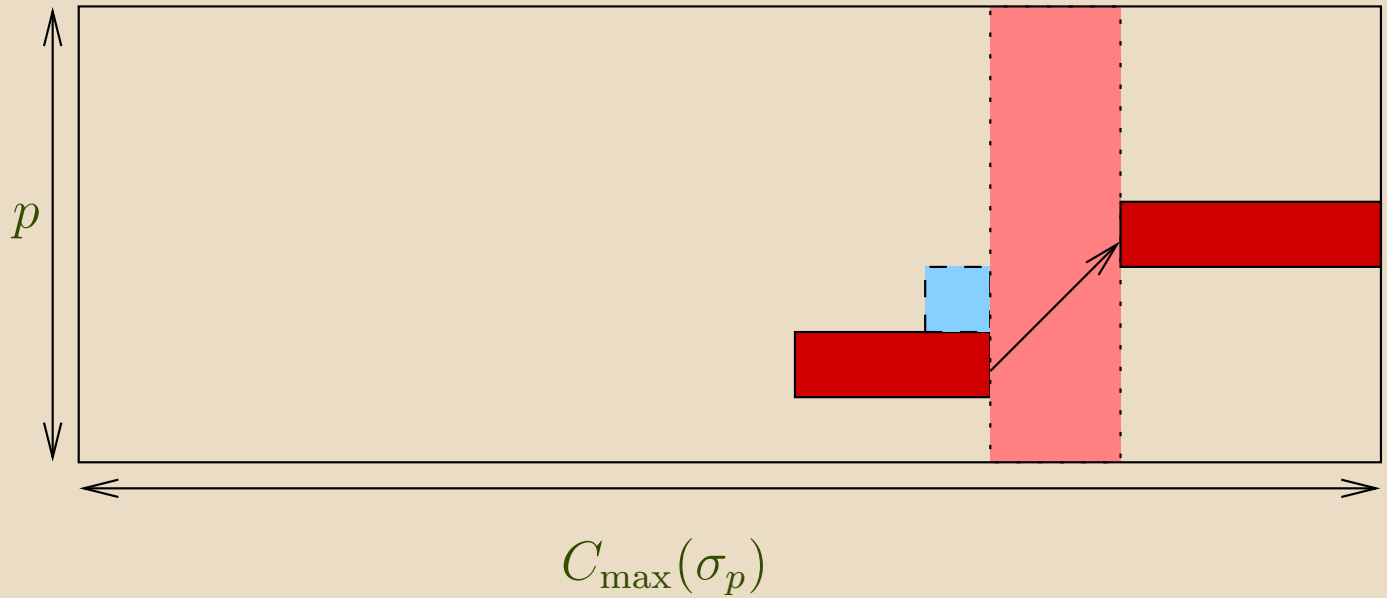
List Scheduling: proving the Coffman result

Proof.



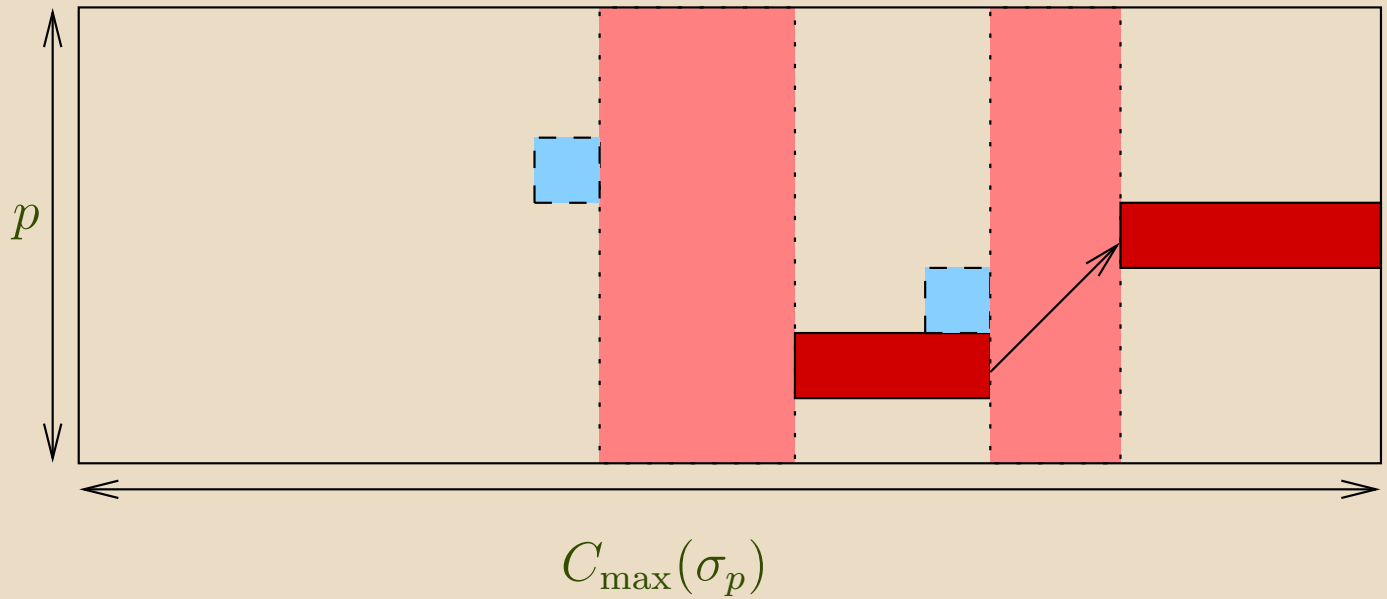
List Scheduling: proving the Coffman result

Proof.



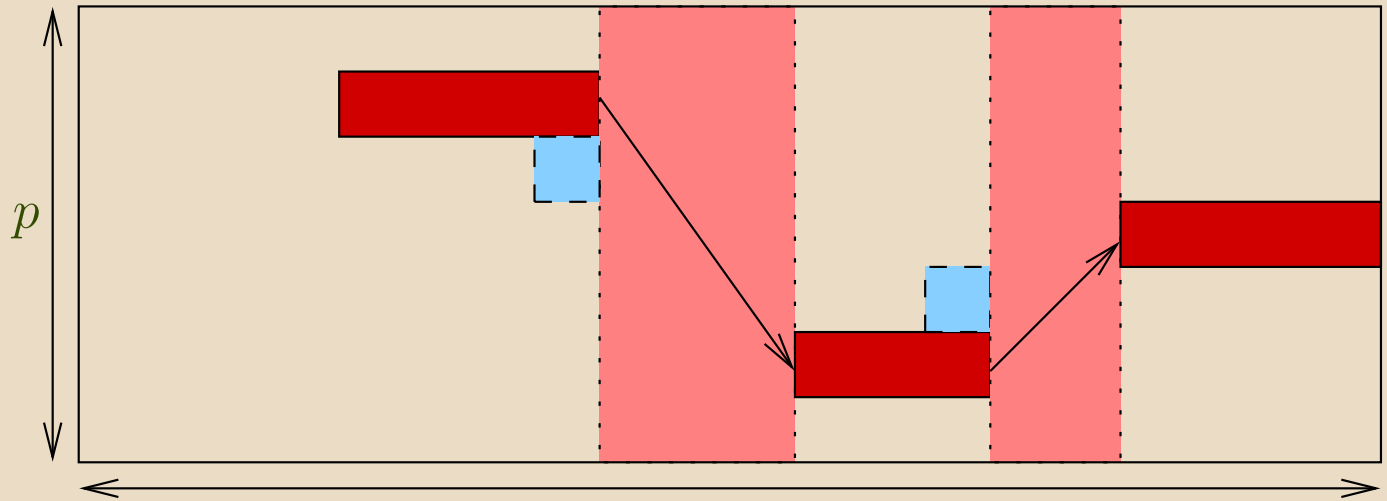
List Scheduling: proving the Coffman result

Proof.



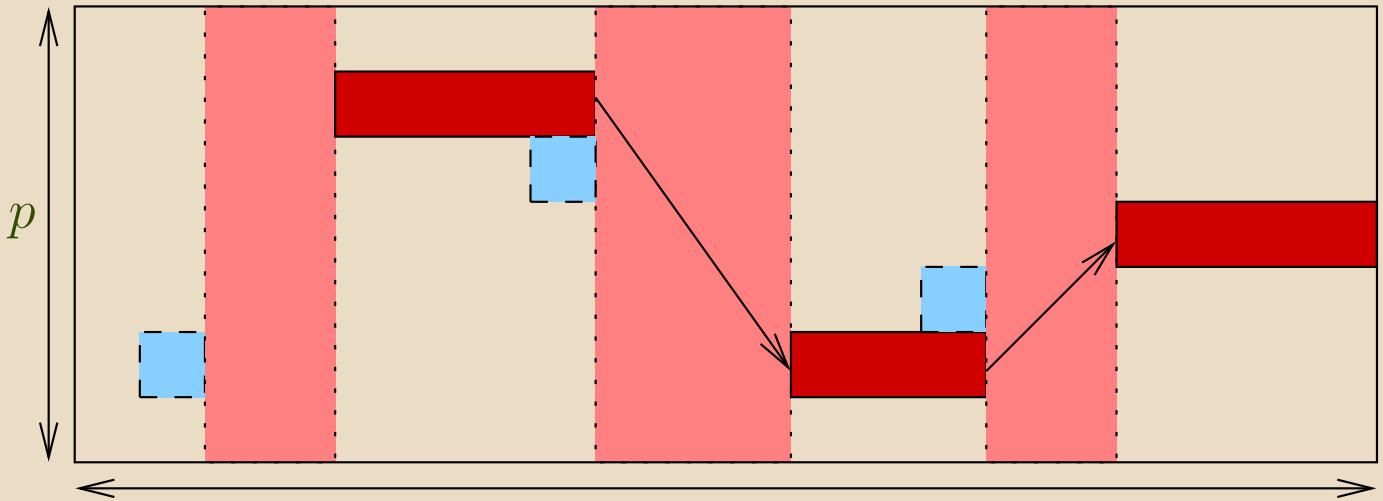
List Scheduling: proving the Coffman result

Proof.



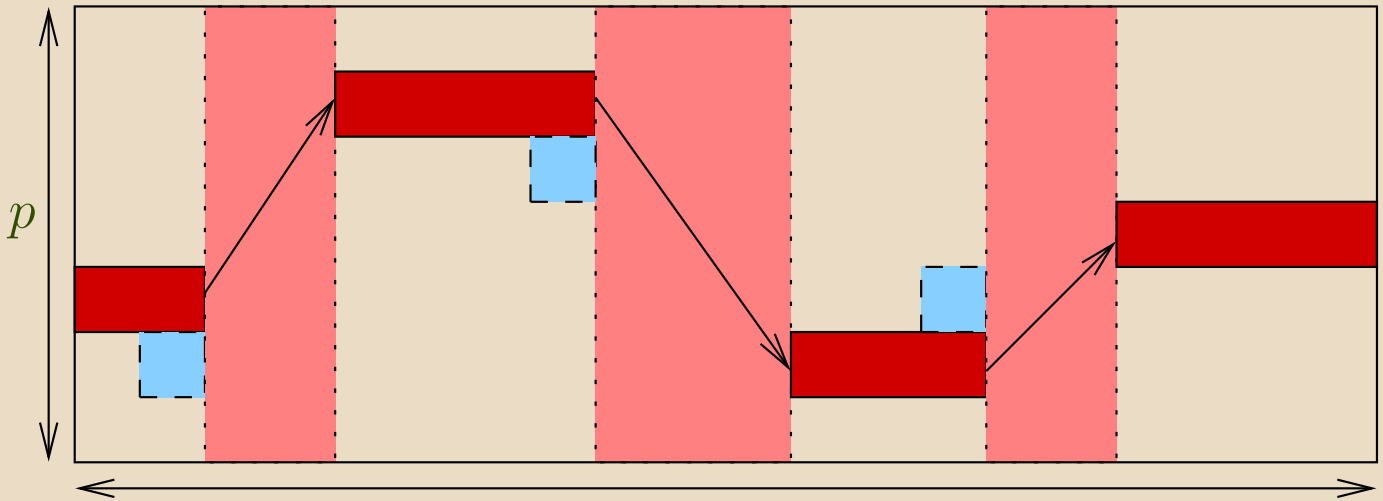
List Scheduling: proving the Coffman result

Proof.



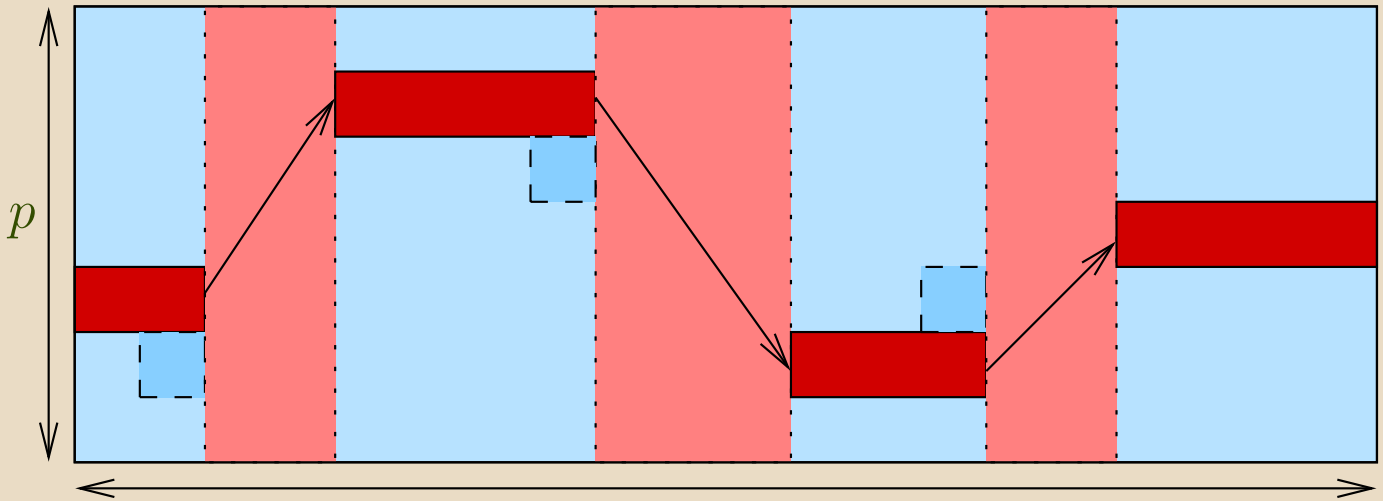
List Scheduling: proving the Coffman result

Proof.



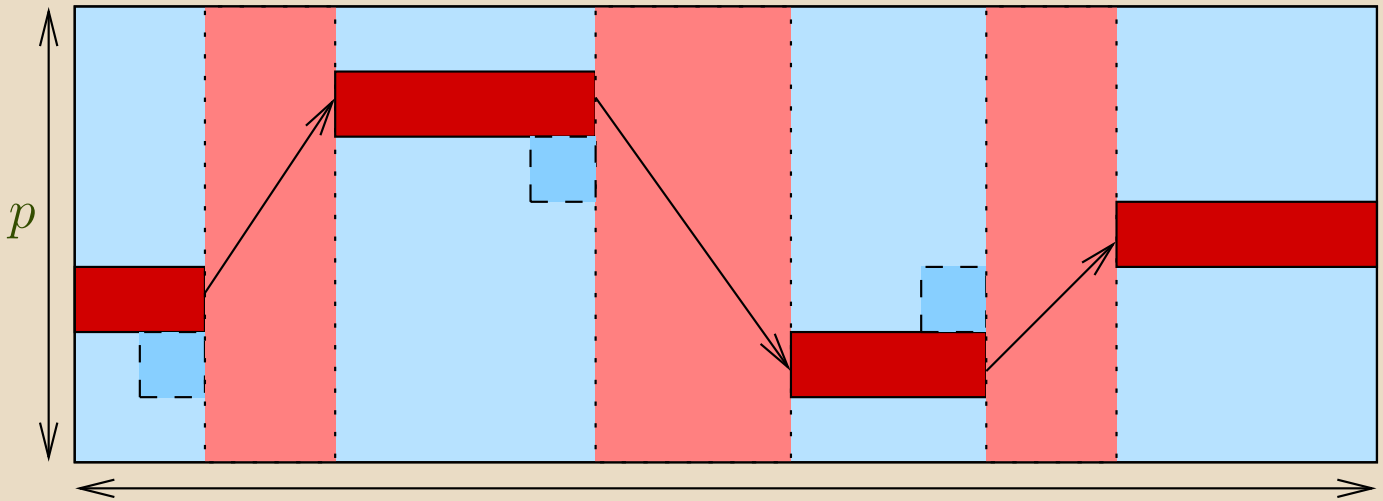
List Scheduling: proving the Coffman result

Proof.



List Scheduling: proving the Coffman result

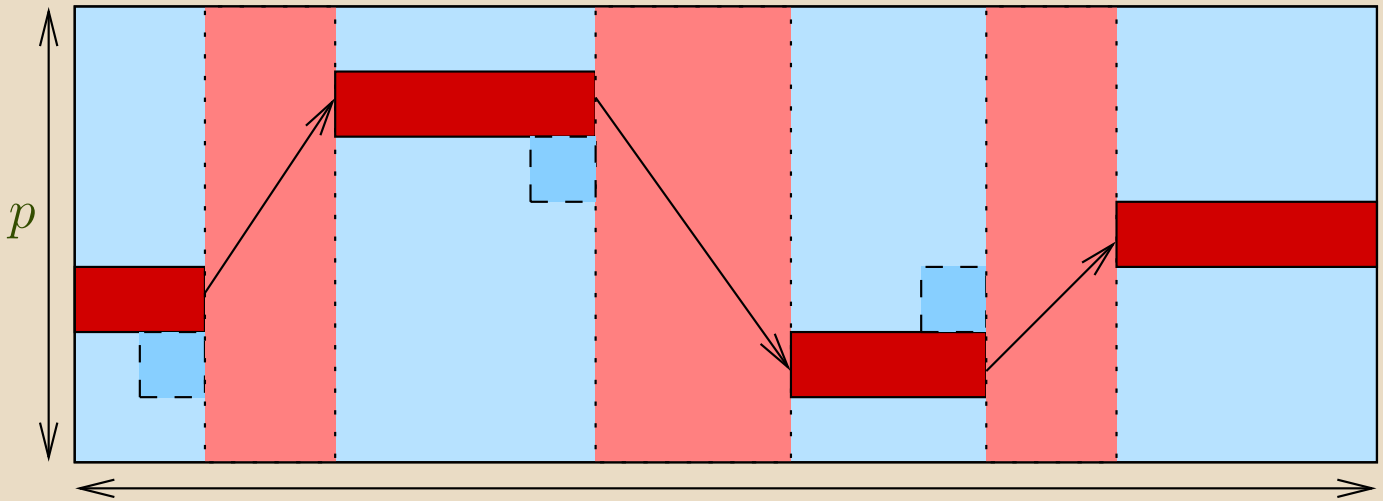
Proof.



Therefore, $Idle \leqslant (p-1).w(\Phi)$ for some Φ

List Scheduling: proving the Coffman result

Proof.



Therefore, $Idle \leqslant (p-1).w(\Phi)$ for some Φ

Hence,

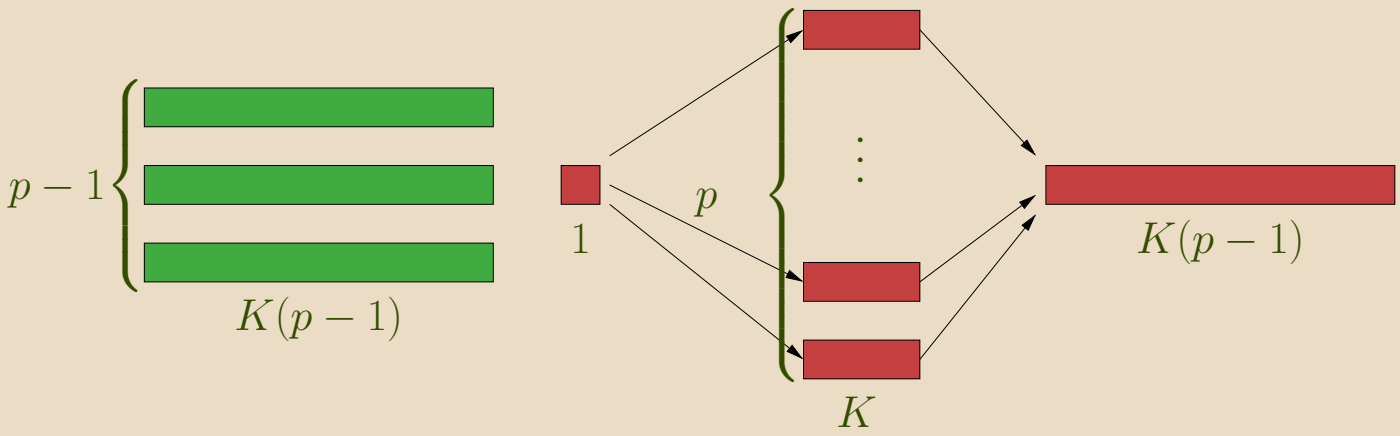
$$p.C_{\max}(\sigma_p) = Idle + Seq \leqslant (p-1)w(\Phi) + Seq$$
$$\leqslant (p-1)C^*_{\max}(p) + p.C^*_{\max}(p) = (2p-1)C^*_{\max}(p)$$

□

List Scheduling: proving the Coffman result

One can actually prove that this bound cannot be improved.

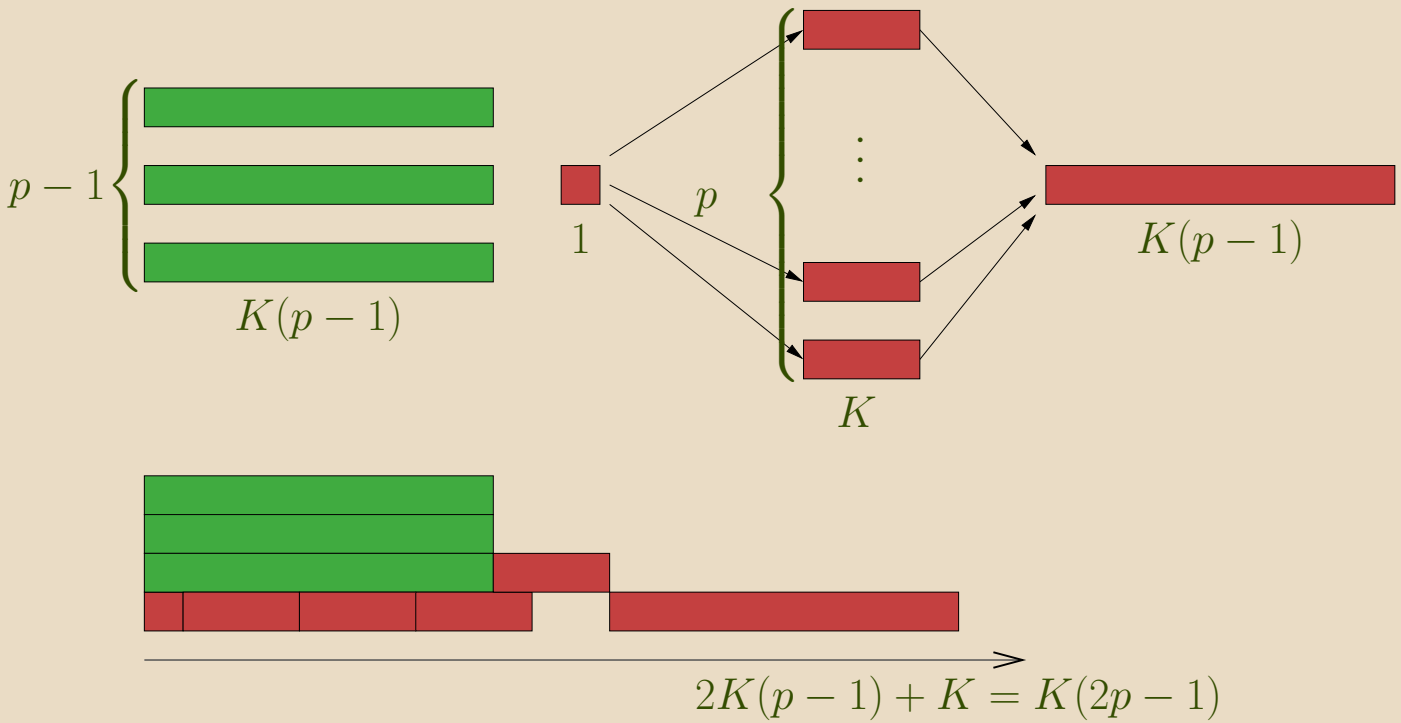
Proof.



List Scheduling: proving the Coffman result

One can actually prove that this bound cannot be improved.

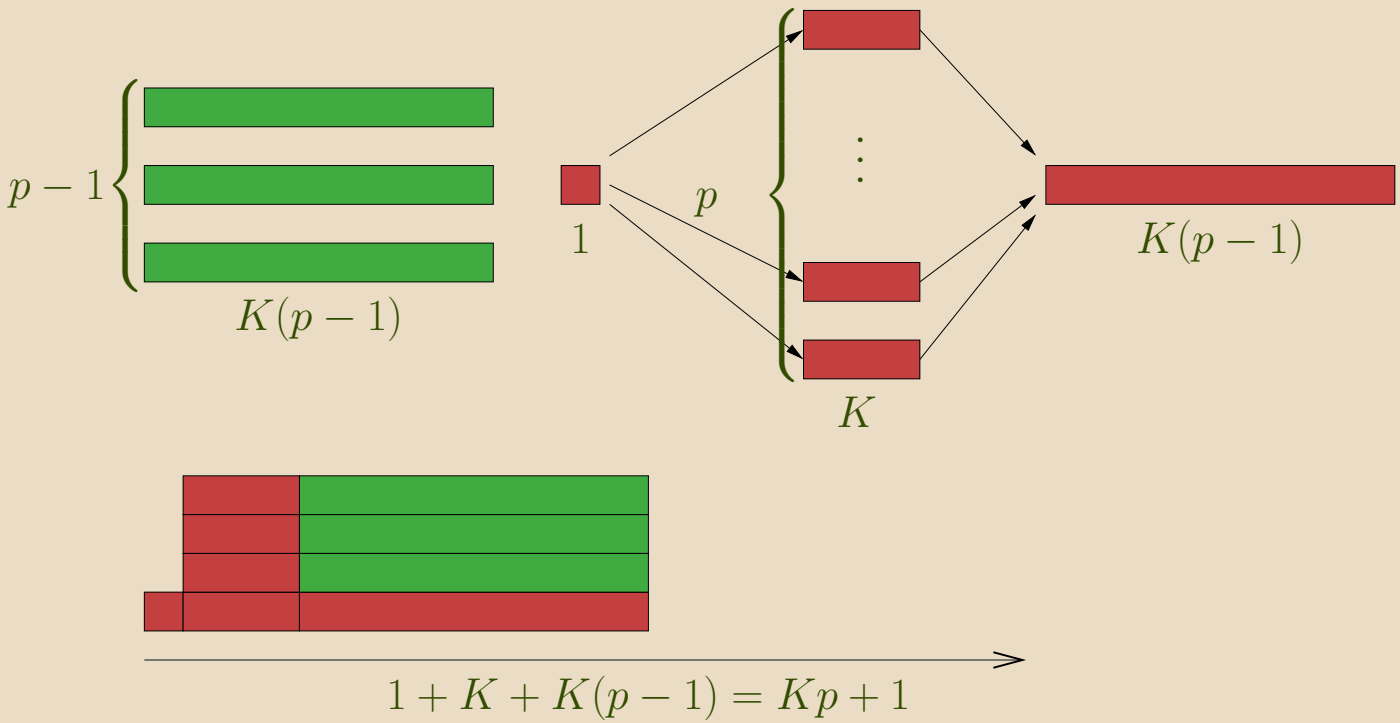
Proof.



List Scheduling: proving the Coffman result

One can actually prove that this bound cannot be improved.

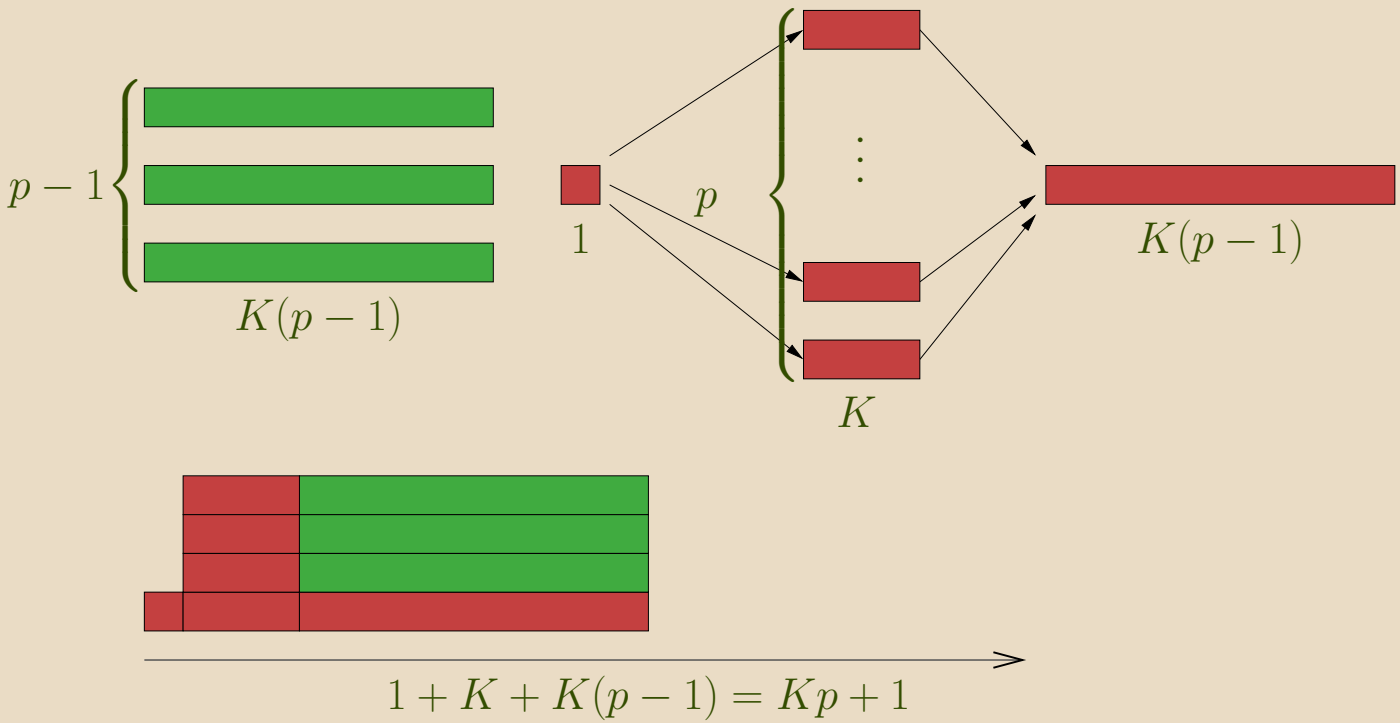
Proof.



List Scheduling: proving the Coffman result

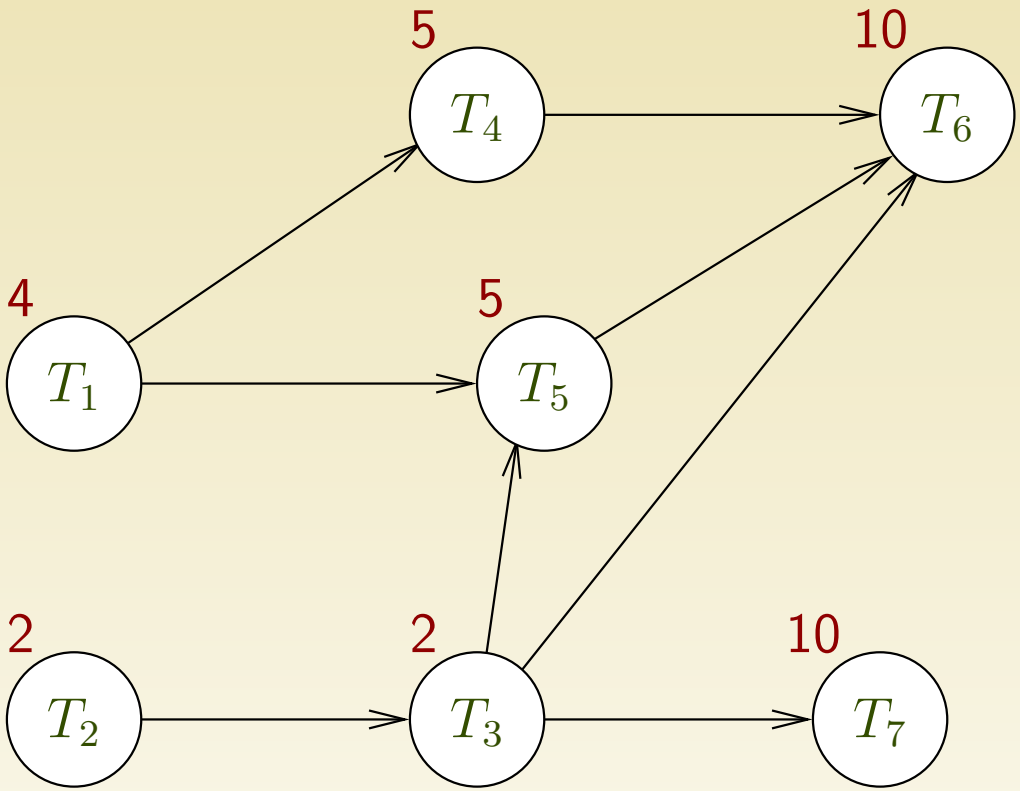
One can actually prove that this bound cannot be improved.

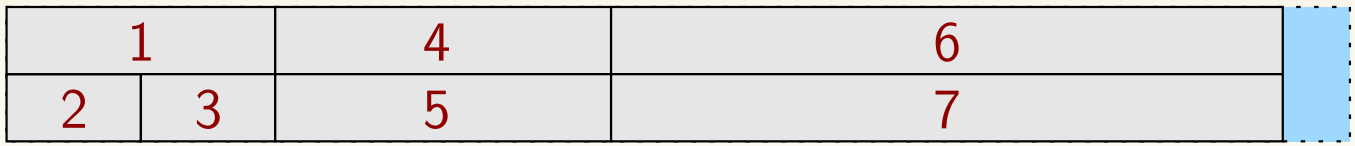
Proof.



$$\varrho \geqslant \frac{K(2p-1)}{Kp+1} \xrightarrow[K\to\infty]{} \frac{2p-1}{p}$$

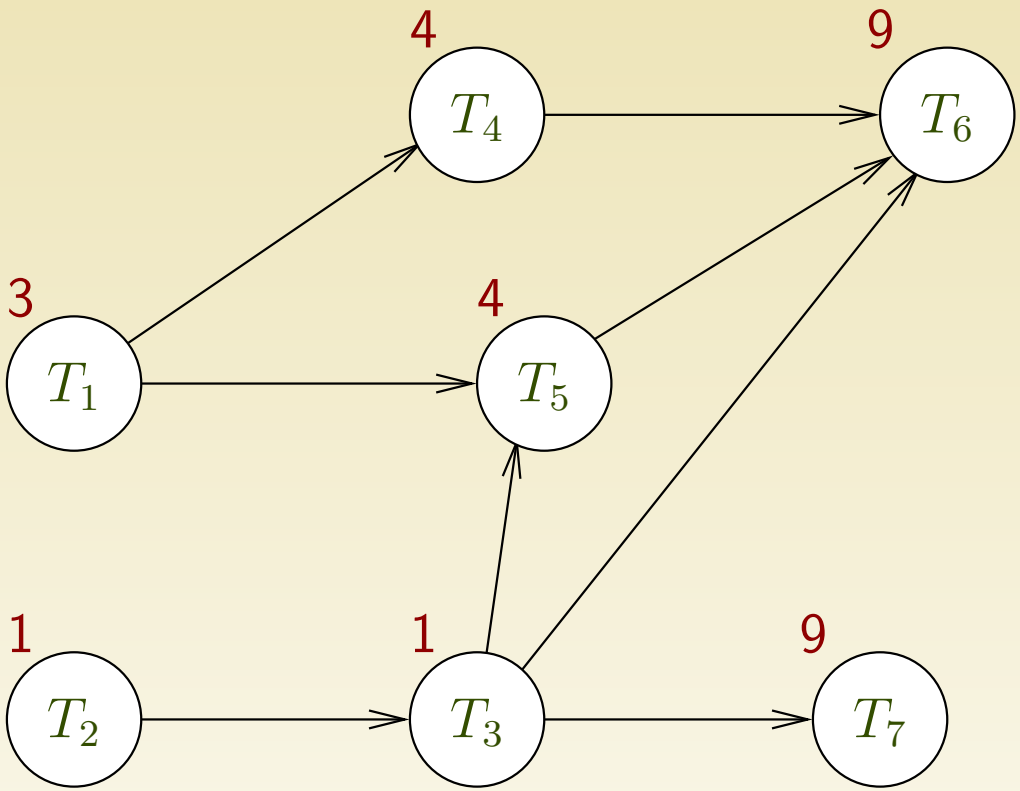
List scheduling Anomalies

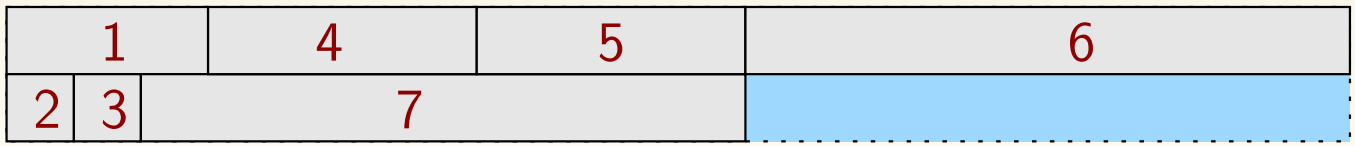




$$MS = 19$$

List scheduling Anomalies





$$MS = 20$$

List Scheduling for Parallel Rigid Tasks

Let us assume we have n independent rigid jobs $J_1 = (p_1, q_1), \ldots, J_n = (p_n, q_n)$ and m machines.
Let us denote by T^* the optimal makespan for this instance.

List Scheduling for Parallel Rigid Tasks

Let us assume we have n independent rigid jobs $J_1 = (p_1, q_1), \ldots, J_n = (p_n, q_n)$ and m machines.
Let us denote by T^* the optimal makespan for this instance.
Let us consider a list schedule of makespan T. Let us denote by $q(t)$ the number of active processors at time t.
We have $\forall t_1, t_2 \in [0, T] : t_1 \leqslant t_2 - T^* \Rightarrow q(t_1) + q(t_2) > m$ (otherwise, the tasks running at time t_2 could have been run at time t_1).

List Scheduling for Parallel Rigid Tasks

Let us assume we have n independent rigid jobs $J_1 = (p_1, q_1), \ldots, J_n = (p_n, q_n)$ and m machines.
Let us denote by T^* the optimal makespan for this instance.
Let us consider a list schedule of makespan T. Let us denote by $q(t)$ the number of active processors at time t.
We have $\forall t_1, t_2 \in [0, T] : t_1 \leqslant t_2 - T^* \Rightarrow q(t_1) + q(t_2) > m$ (otherwise, the tasks running at time t_2 could have been run at time t_1).
Let us assume that $T > 2T^*$. Then we have:

$$mT^* \geqslant \sum_i q_i p_i = \int_0^T q(t) = \int_0^{2T^*} q(t) + \int_{2T^*}^T q(t)$$

$$\geqslant \underbrace{\int_0^{T^*} q(t) + q(t+T^*)}_{>mT^*} + \underbrace{\int_{2T^*}^T q(t)}_{\geqslant 0}, \text{which is absurd.}$$

List Scheduling for Parallel Rigid Tasks

Let us assume we have n independent rigid jobs $J_1 = (p_1, q_1), \ldots, J_n = (p_n, q_n)$ and m machines.
Let us denote by T^* the optimal makespan for this instance.
Let us consider a list schedule of makespan T. Let us denote by $q(t)$ the number of active processors at time t.
We have $\forall t_1, t_2 \in [0, T] : t_1 \leqslant t_2 - T^* \Rightarrow q(t_1) + q(t_2) > m$ (otherwise, the tasks running at time t_2 could have been run at time t_1).
Let us assume that $T > 2T^*$. Then we have:

$$mT^* \geqslant \sum_i q_i p_i = \int_0^T q(t) = \int_0^{2T^*} q(t) + \int_{2T^*}^T q(t)$$

Theorem 2.

List-scheduling has an approximation factor of 2 for minimizing the Cmax of Parallel Rigid Tasks.

Going Online

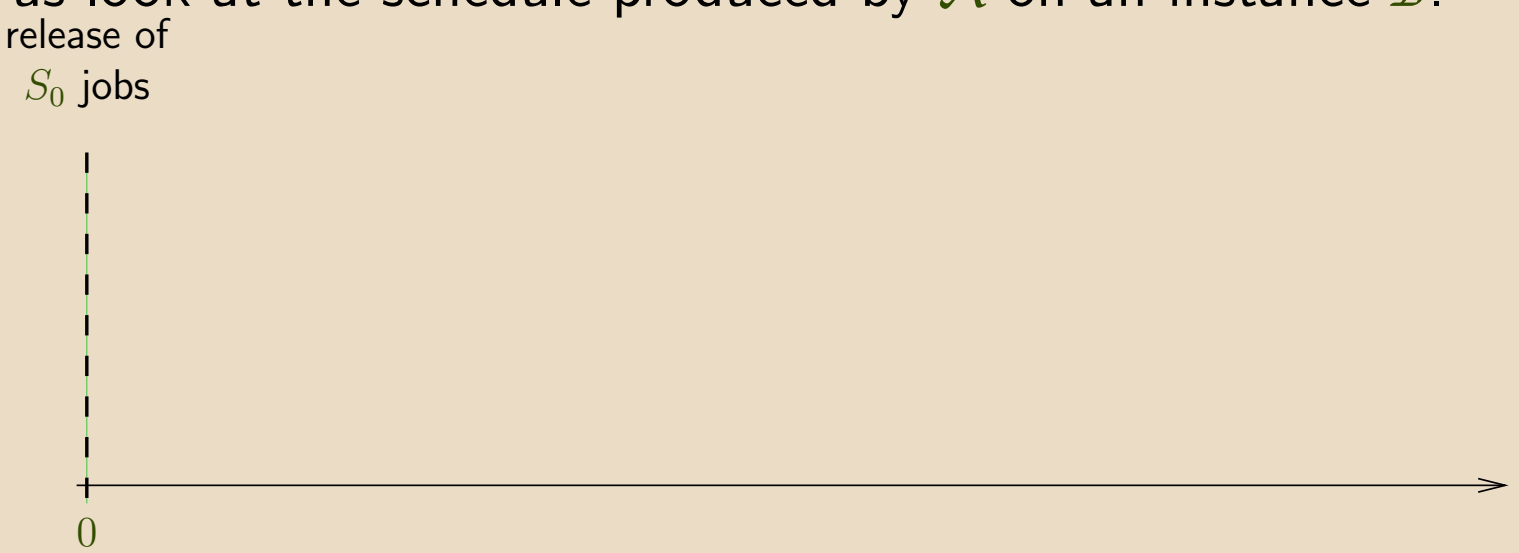
How can we use the previous result when going online?

Theorem 3: [Shmoys91].

Let $\mathcal{A}$ be a polynomial-time ϱ-approximation for $\langle P|size_j|C_{\max}\rangle$. Based on $\mathcal{A}$, we can build a 2ϱ-competitive polynomial-time online clairvoyant algorithm for $\langle P|size_j, r_j|C_{\max}\rangle$.

Proof.

Let us look at the schedule produced by $\mathcal{A}$ on an instance $\mathcal{I}$.



Going Online

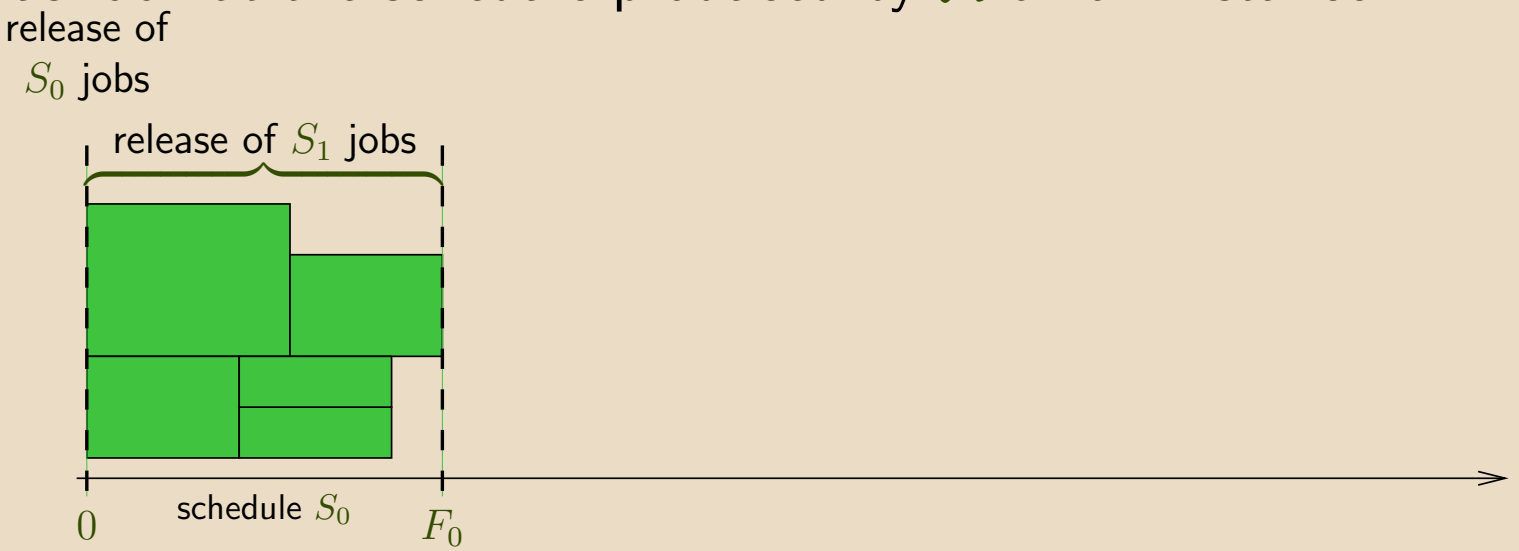
How can we use the previous result when going online?

Theorem 3: **[Shmoys91]**.

Let $\mathcal{A}$ be a polynomial-time ϱ-approximation for $\langle P|size_j|C_{\max}\rangle$. Based on $\mathcal{A}$, we can build a 2ϱ-competitive polynomial-time online clairvoyant algorithm for $\langle P|size_j, r_j|C_{\max}\rangle$.

Proof.

Let us look at the schedule produced by $\mathcal{A}$ on an instance $\mathcal{I}$.



Going Online

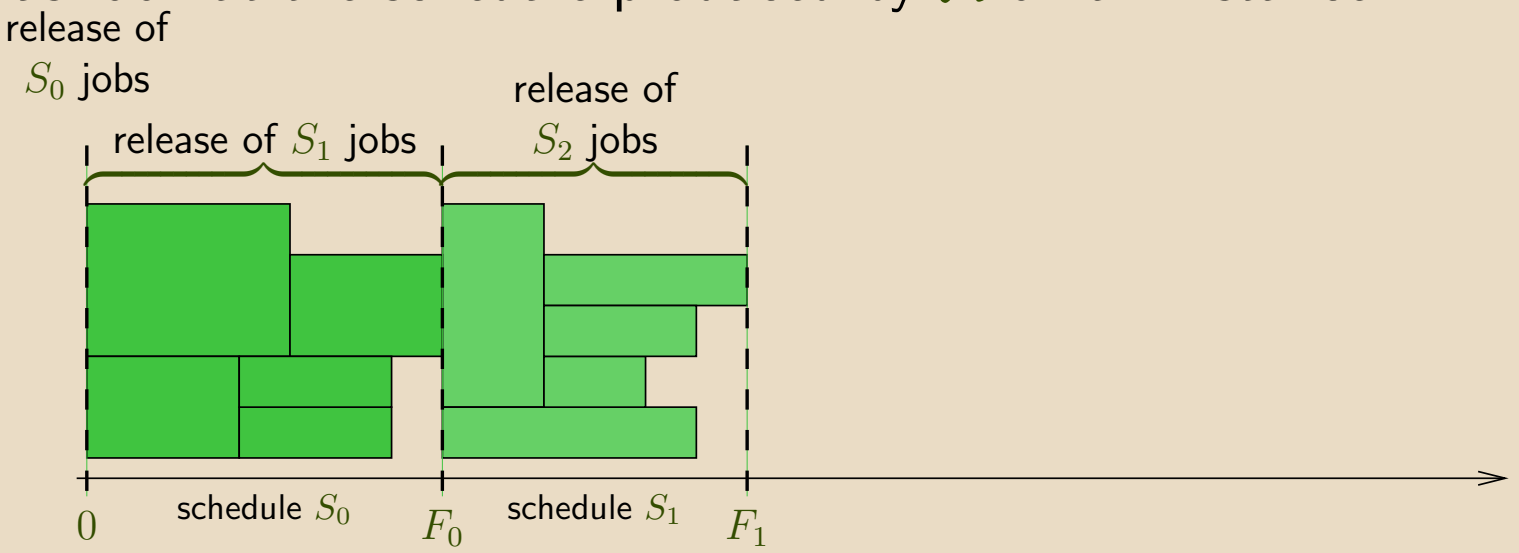
How can we use the previous result when going online?

Theorem 3: **[Shmoys91]**.

Let $\mathcal{A}$ be a polynomial-time ϱ-approximation for $\langle P|size_j|C_{\max}\rangle$. Based on $\mathcal{A}$, we can build a 2ϱ-competitive polynomial-time online clairvoyant algorithm for $\langle P|size_j, r_j|C_{\max}\rangle$.

Proof.

Let us look at the schedule produced by $\mathcal{A}$ on an instance $\mathcal{I}$.



Going Online

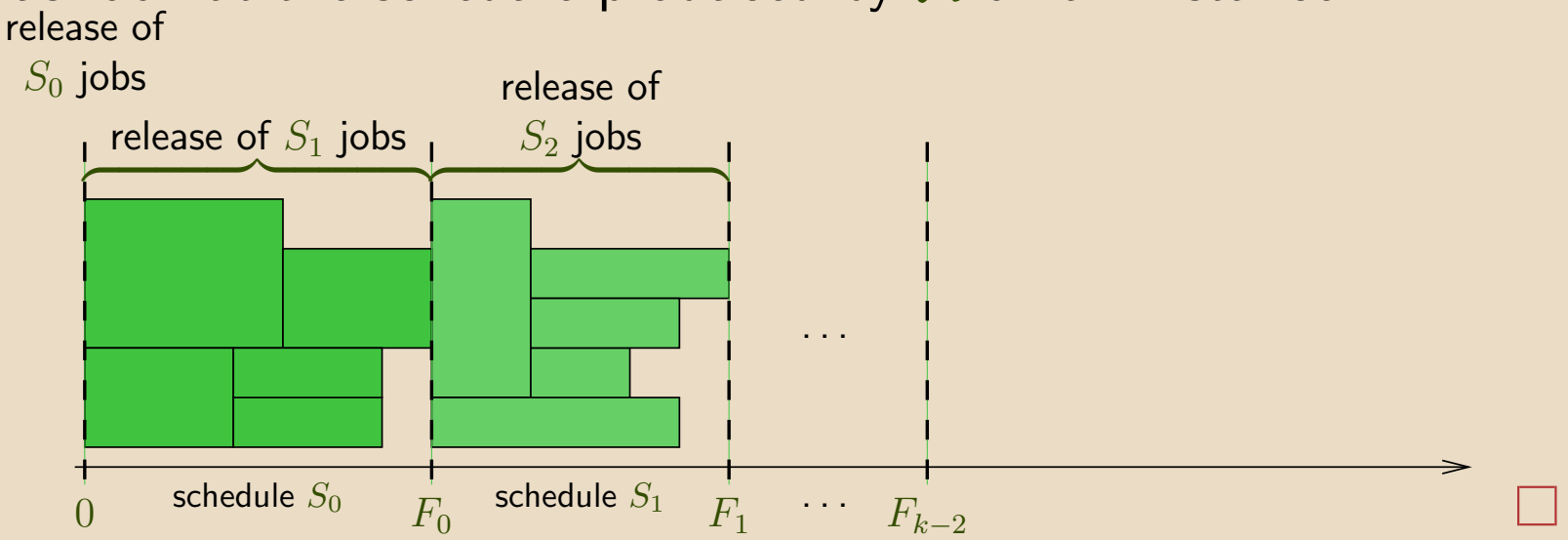
How can we use the previous result when going online?

Theorem 3: **[Shmoys91]**.

Let $\mathcal{A}$ be a polynomial-time ϱ-approximation for $\langle P|size_j|C_{\max}\rangle$. Based on $\mathcal{A}$, we can build a 2ϱ-competitive polynomial-time online clairvoyant algorithm for $\langle P|size_j, r_j|C_{\max}\rangle$.

Proof.

Let us look at the schedule produced by $\mathcal{A}$ on an instance $\mathcal{I}$.



Going Online

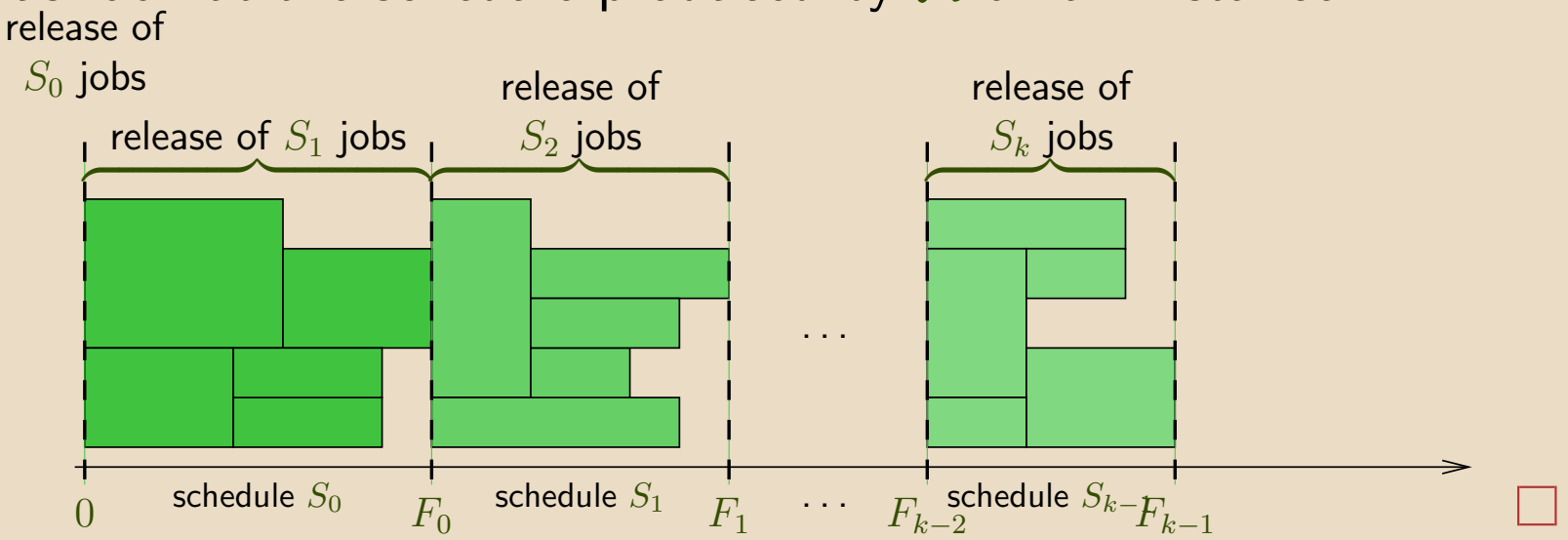
How can we use the previous result when going online?

Theorem 3: **[Shmoys91]**.

Let $\mathcal{A}$ be a polynomial-time ϱ-approximation for $\langle P|size_j|C_{\max}\rangle$. Based on $\mathcal{A}$, we can build a 2ϱ-competitive polynomial-time online clairvoyant algorithm for $\langle P|size_j, r_j|C_{\max}\rangle$.

Proof.

Let us look at the schedule produced by $\mathcal{A}$ on an instance $\mathcal{I}$.



Going Online

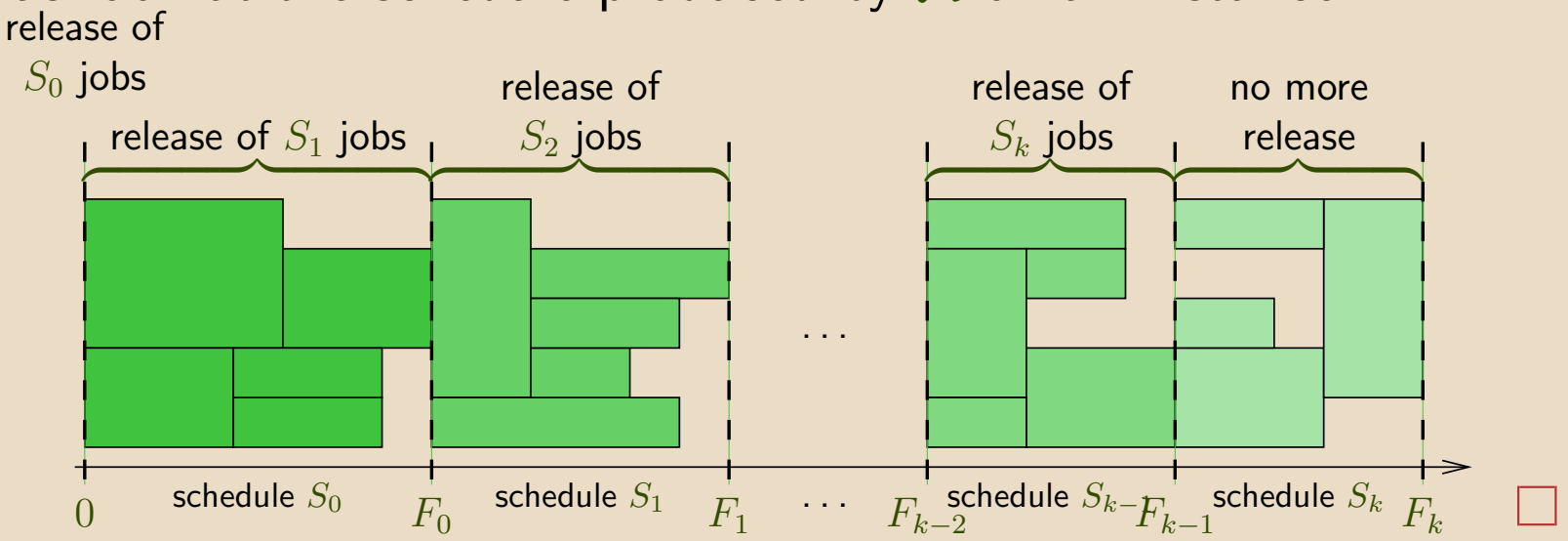
How can we use the previous result when going online?

Theorem 3: [Shmoys91].

Let $\mathcal{A}$ be a polynomial-time ϱ-approximation for $\langle P|size_j|C_{\max}\rangle$. Based on $\mathcal{A}$, we can build a 2ϱ-competitive polynomial-time online clairvoyant algorithm for $\langle P|size_j, r_j|C_{\max}\rangle$.

Proof.

Let us look at the schedule produced by $\mathcal{A}$ on an instance $\mathcal{I}$.



Going Online

How can we use the previous result when going online?

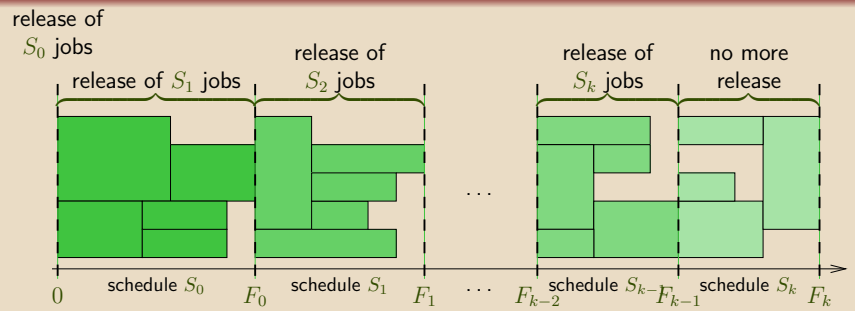
Theorem 3: **[Shmoys91]**.

Let $\mathcal{A}$ be a polynomial-time ϱ-approximation for $\langle P|size_j|C_{\max}\rangle$. Based on $\mathcal{A}$, we can build a 2ϱ-competitive polynomial-time online clairvoyant algorithm for $\langle P|size_j, r_j|C_{\max}\rangle$.

Proof.

Consider $\mathcal{I}'$ where S_k jobs are released at time F_{k-2}. We have:

$$C^*_{\max}(\mathcal{I}') \leqslant C^*_{\max}(\mathcal{I}).$$



Going Online

How can we use the previous result when going online?

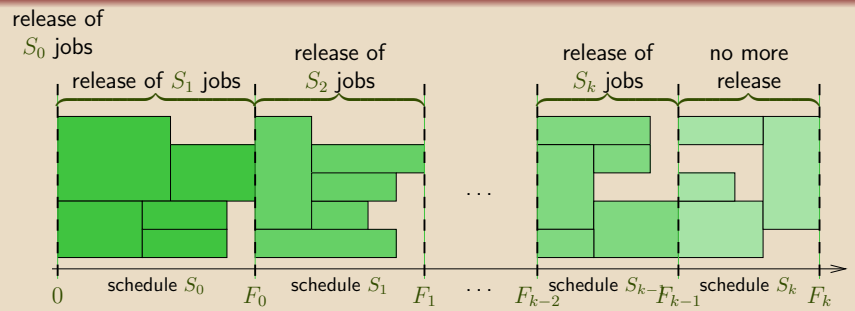
Theorem 3: **[Shmoys91]**.

Let $\mathcal{A}$ be a polynomial-time ϱ-approximation for $\langle P|size_j|C_{\max}\rangle$. Based on $\mathcal{A}$, we can build a 2ϱ-competitive polynomial-time online clairvoyant algorithm for $\langle P|size_j, r_j|C_{\max}\rangle$.

Proof.

Consider $\mathcal{I}'$ where S_k jobs are released at time F_{k-2}. We have:

$$C^*_{\max}(\mathcal{I}') \leqslant C^*_{\max}(\mathcal{I}).$$



- $F_{k-2} + F_k - F_{k-1} \leqslant \varrho C^*_{\max}(\mathcal{I}')$

Going Online

How can we use the previous result when going online?

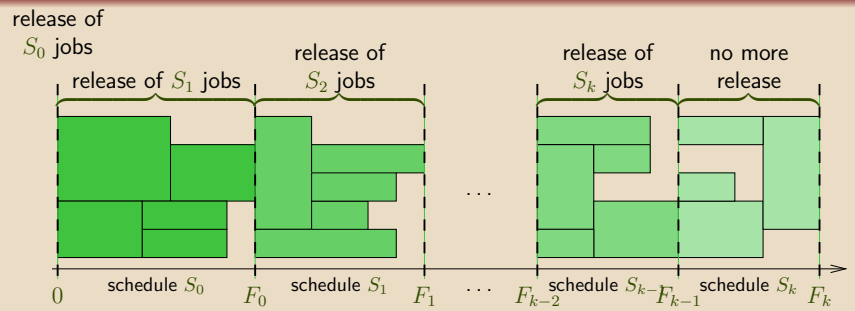
Theorem 3: **[Shmoys91]**.

Let $\mathcal{A}$ be a polynomial-time ϱ-approximation for $\langle P|size_j|C_{\max}\rangle$. Based on $\mathcal{A}$, we can build a 2ϱ-competitive polynomial-time online clairvoyant algorithm for $\langle P|size_j, r_j|C_{\max}\rangle$.

Proof.

Consider $\mathcal{I}'$ where S_k jobs are released at time F_{k-2}. We have:

$$C^*_{\max}(\mathcal{I}') \leqslant C^*_{\max}(\mathcal{I}).$$



- $F_{k-2} + F_k - F_{k-1} \leqslant \varrho C^*_{\max}(\mathcal{I}')$
- $F_{k-1} - F_{k-2} \leqslant \varrho C^*_{\max}(\mathcal{I}')$

Going Online

How can we use the previous result when going online?

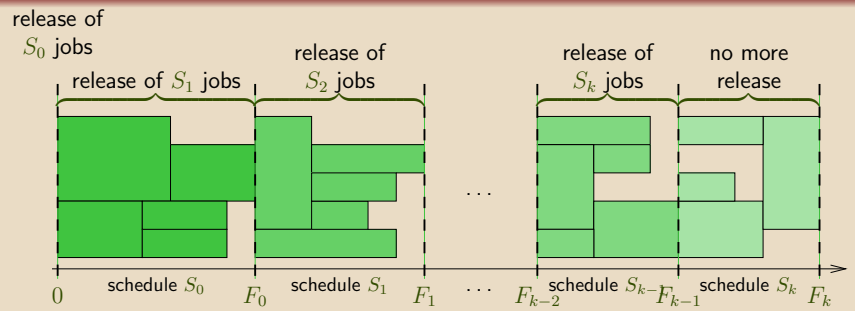
Theorem 3: **[Shmoys91]**.

Let $\mathcal{A}$ be a polynomial-time ϱ-approximation for $\langle P|size_j|C_{\max}\rangle$. Based on $\mathcal{A}$, we can build a 2ϱ-competitive polynomial-time online clairvoyant algorithm for $\langle P|size_j, r_j|C_{\max}\rangle$.

Proof.

Consider $\mathcal{I}'$ where S_k jobs are released at time F_{k-2}. We have:

$$C^*_{\max}(\mathcal{I}') \leqslant C^*_{\max}(\mathcal{I}).$$



- $F_{k-2} + F_k - F_{k-1} \leqslant \varrho C^*_{\max}(\mathcal{I}')$
- $F_{k-1} - F_{k-2} \leqslant \varrho C^*_{\max}(\mathcal{I}')$

Hence $F_k \leqslant 2\varrho C^*_{\max}(\mathcal{I}') \leqslant 2\varrho C^*_{\max}(\mathcal{I})$

Going Online

How can we use the previous result when going online?

Theorem 3: [Shmoys91].

Let $\mathcal{A}$ be a polynomial-time ϱ-approximation for $\langle P|size_j|C_{\max}\rangle$. Based on $\mathcal{A}$, we can build a 2ϱ-competitive polynomial-time online clairvoyant algorithm for $\langle P|size_j, r_j|C_{\max}\rangle$.

- There is a PTAS for $\langle Q||C_{\max}\rangle$. Hence, there is an $(2+\varepsilon)$-competitive online clairvoyant algorithm for $\langle Q|r_j|C_{\max}\rangle$.

Going Online

How can we use the previous result when going online?

Theorem 3: [Shmoys91].

Let $\mathcal{A}$ be a polynomial-time ϱ-approximation for $\langle P|size_j|C_{\max}\rangle$. Based on $\mathcal{A}$, we can build a 2ϱ-competitive polynomial-time online clairvoyant algorithm for $\langle P|size_j, r_j|C_{\max}\rangle$.

- There is a PTAS for $\langle Q||C_{\max}\rangle$. Hence, there is an $(2+\varepsilon)$-competitive online clairvoyant algorithm for $\langle Q|r_j|C_{\max}\rangle$.
- There is a 2 approximation $\langle Q||C_{\max}\rangle$. Hence, there is an 4-competitive online clairvoyant algorithm for $\langle Q|r_j|C_{\max}\rangle$.

Going Online

How can we use the previous result when going online?

Theorem 3: [Shmoys91].

Let $\mathcal{A}$ be a polynomial-time ϱ-approximation for $\langle P|size_j|C_{\max}\rangle$. Based on $\mathcal{A}$, we can build a 2ϱ-competitive polynomial-time online clairvoyant algorithm for $\langle P|size_j, r_j|C_{\max}\rangle$.

- There is a PTAS for $\langle Q||C_{\max}\rangle$. Hence, there is an $(2+\varepsilon)$-competitive online clairvoyant algorithm for $\langle Q|r_j|C_{\max}\rangle$.
- There is a 2 approximation $\langle Q||C_{\max}\rangle$. Hence, there is an 4-competitive online clairvoyant algorithm for $\langle Q|r_j|C_{\max}\rangle$.
- There is a 2 approximation $\langle P|size_j|C_{\max}\rangle$. Hence, there is an 4-competitive online clairvoyant algorithm for $\langle Q|size_j|C_{\max}\rangle$.

Going Online

How can we use the previous result when going online?

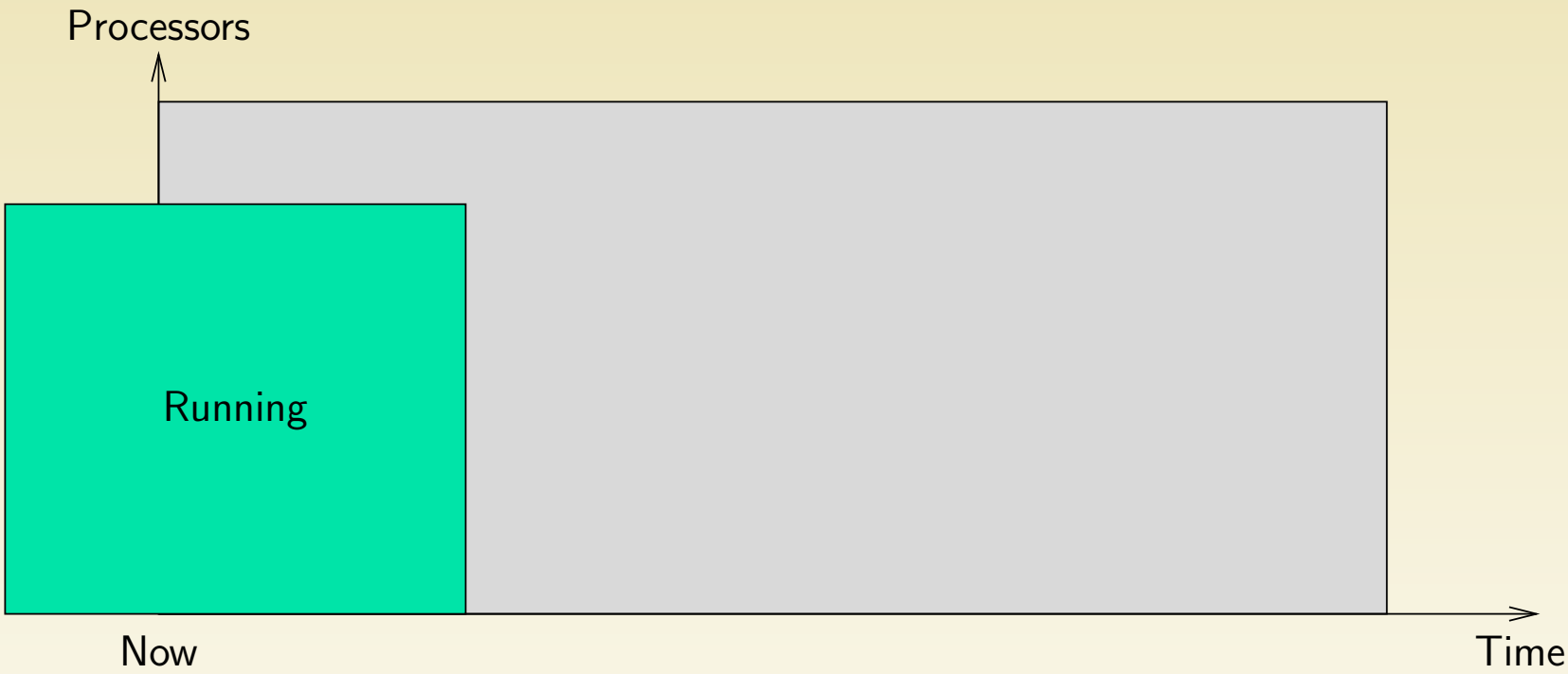
Theorem 3: [Shmoys91].

Let $\mathcal{A}$ be a polynomial-time ϱ-approximation for $\langle P|size_j|C_{\max}\rangle$. Based on $\mathcal{A}$, we can build a 2ϱ-competitive polynomial-time online clairvoyant algorithm for $\langle P|size_j, r_j|C_{\max}\rangle$.

- There is a PTAS for $\langle Q||C_{\max}\rangle$. Hence, there is an $(2+\varepsilon)$-competitive online clairvoyant algorithm for $\langle Q|r_j|C_{\max}\rangle$.
- There is a 2 approximation $\langle Q||C_{\max}\rangle$. Hence, there is an 4-competitive online clairvoyant algorithm for $\langle Q|r_j|C_{\max}\rangle$.
- There is a 2 approximation $\langle P|size_j|C_{\max}\rangle$. Hence, there is an 4-competitive online clairvoyant algorithm for $\langle Q|size_j|C_{\max}\rangle$.
- Actually, by doing a slightly finer analysis, on can show that the list-scheduling algorithm is a $(2-1/m)$-competitive non-clairvoyant algorithm for $\langle P|r_j|C_{\max}\rangle$.

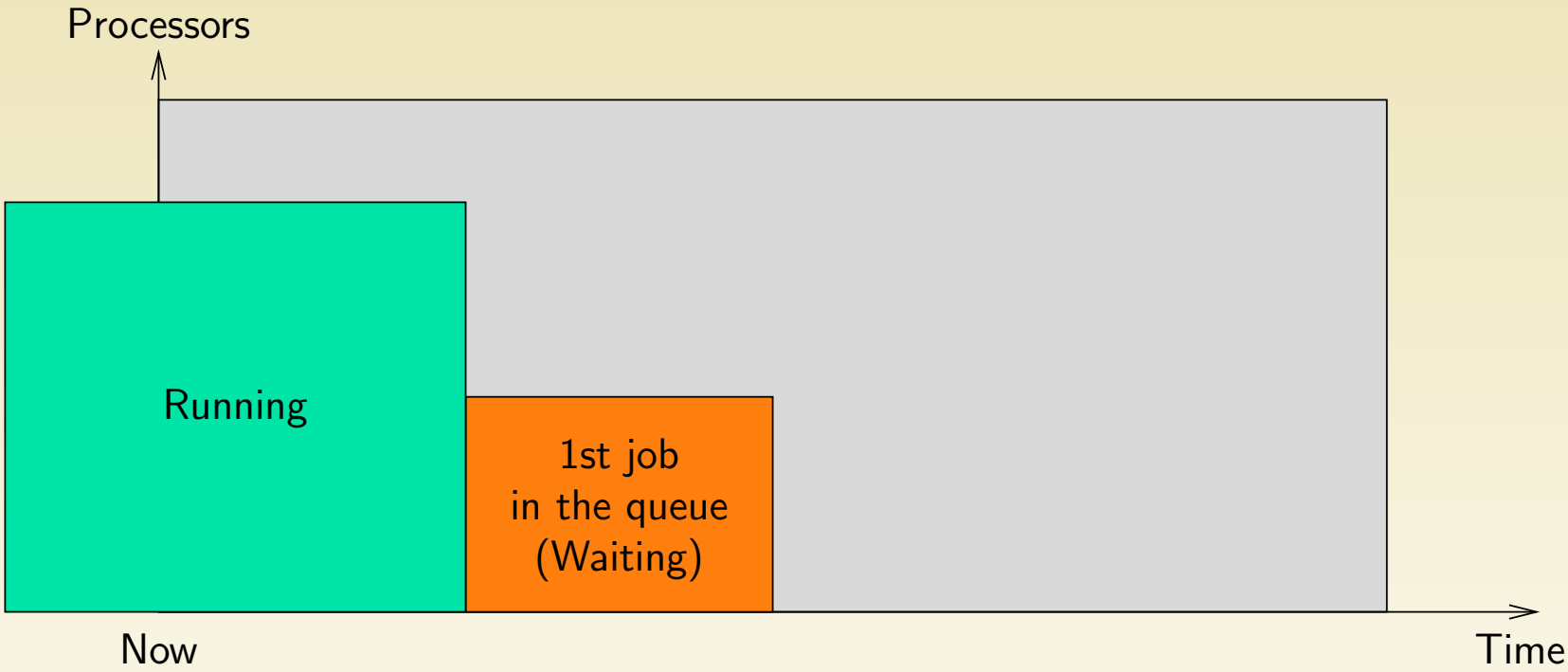
Outline

General Principle



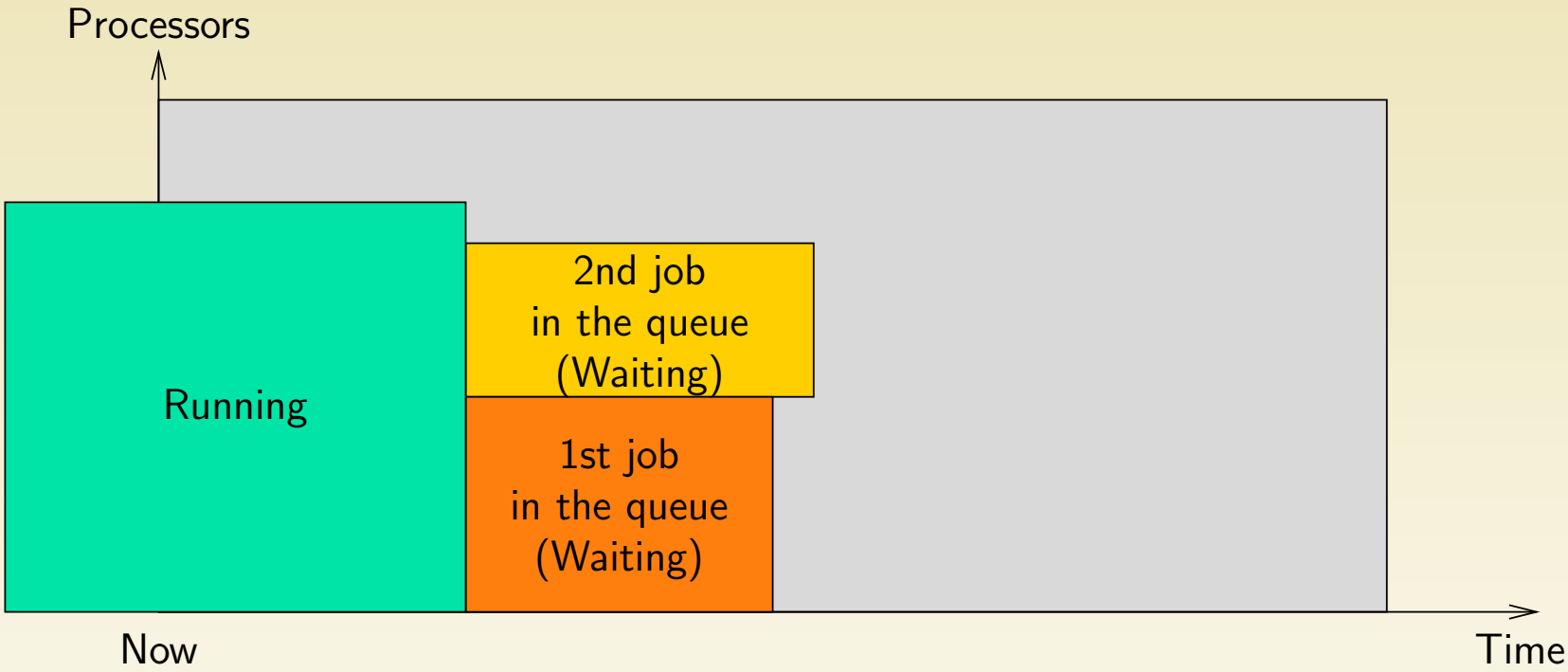
- Jobs arrive one after the other and are scheduled at arrival.

General Principle



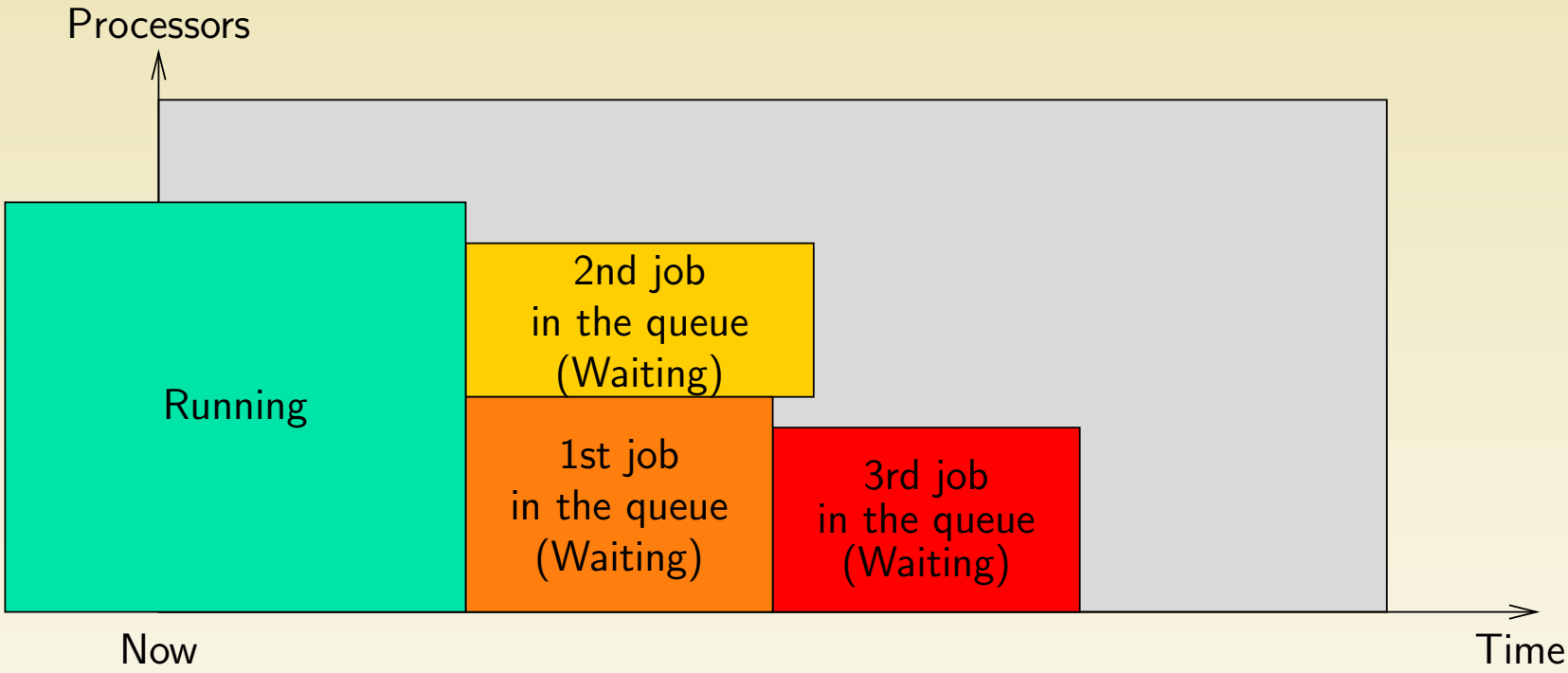
- Jobs arrive one after the other and are scheduled at arrival.

General Principle



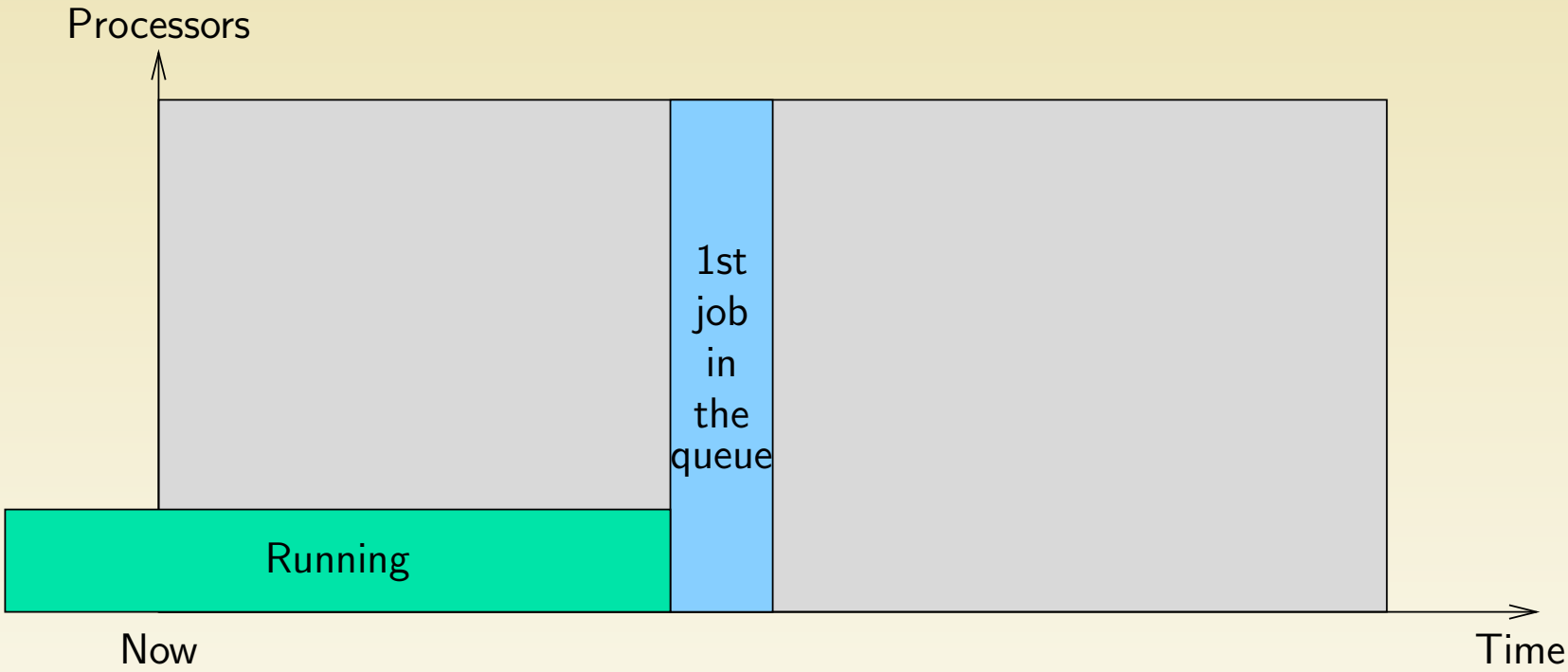
- Jobs arrive one after the other and are scheduled at arrival.

General Principle



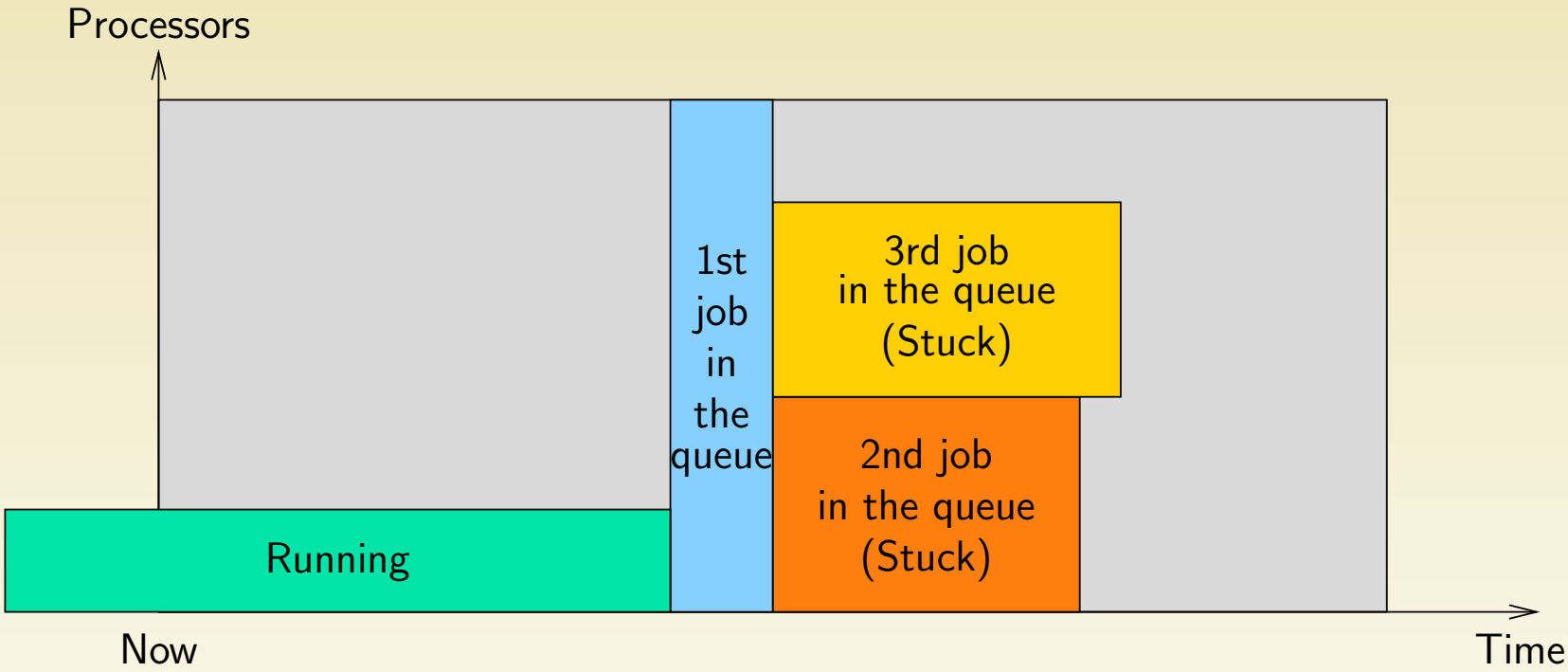
- Jobs arrive one after the other and are scheduled at arrival.

First Come First Served



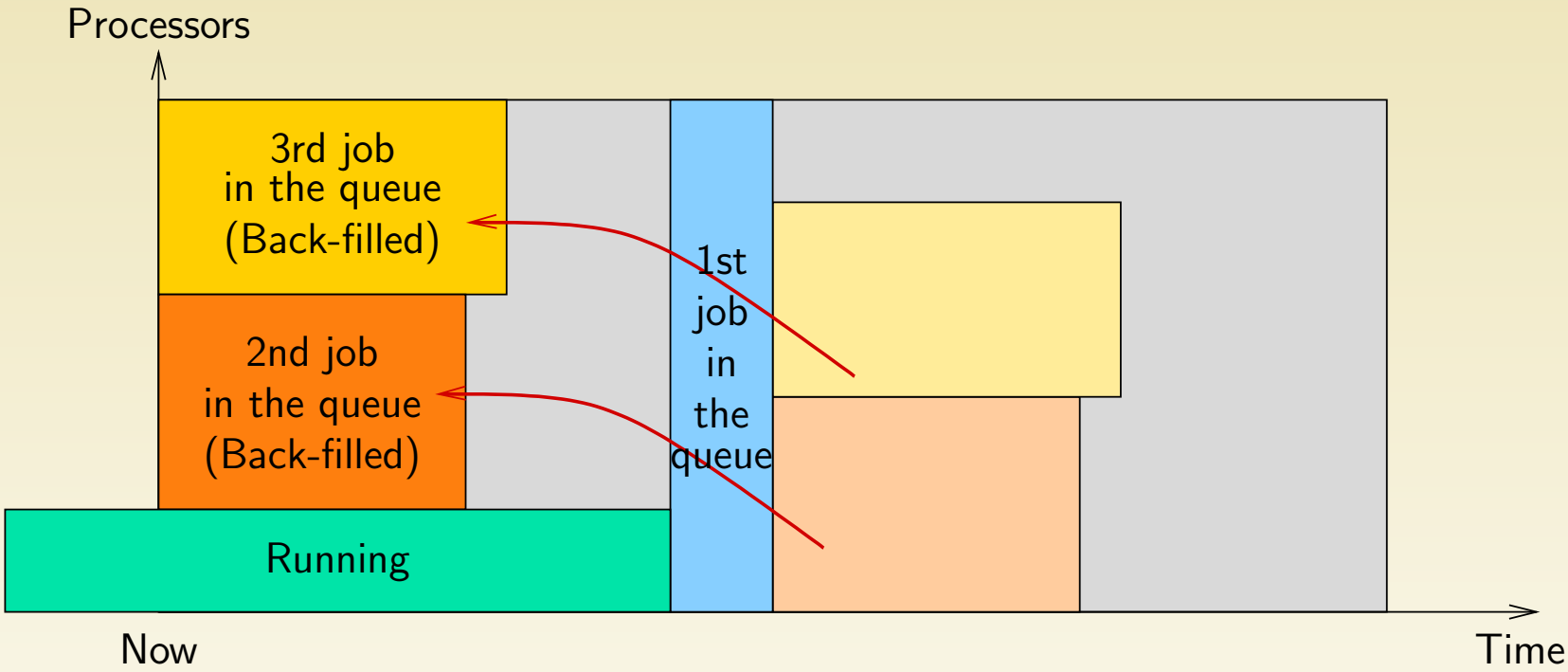
- FCFS = simplest scheduling option
- Fragmentation $\rightsquigarrow$ need for backfilling

First Come First Served



- FCFS = simplest scheduling option
- Fragmentation $\rightsquigarrow$ need for backfilling

First Come First Served



- FCFS = simplest scheduling option
- Fragmentation $\rightsquigarrow$ need for backfilling

Backfilling: Question

- Which job(s) should be picked for promotion through the queue?
- Many heuristics are possible
- Two have been studied in detail
  - EASY
  - Conservative Back Filling (CBF)
- In practice EASY (or variants of it) is used, while CBF is not.
- Although, OAR, a recently proposed batch scheduler implements CBF.

Outline

EASY Backfilling

Extensible Argonne Scheduling System

Maintain only one *reservation*, for the first job in the queue.

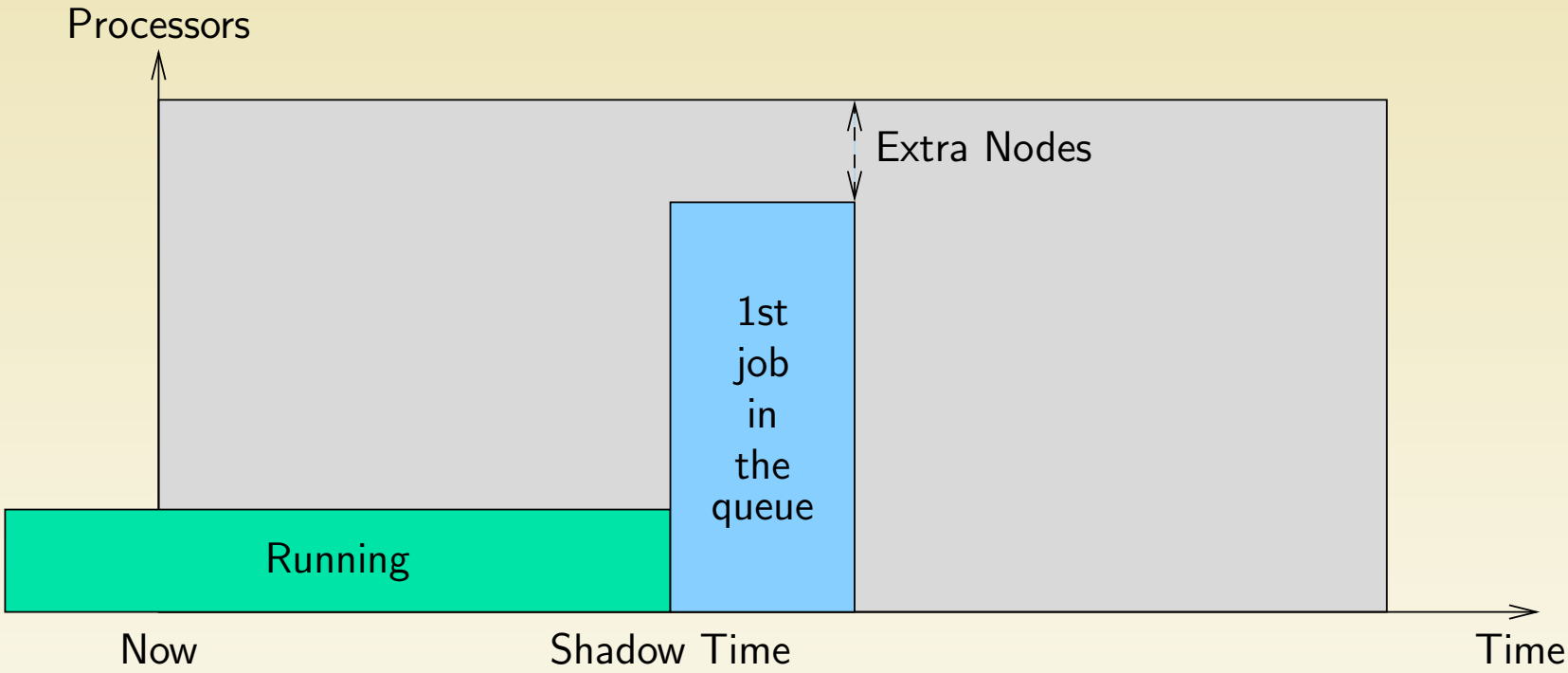
Definitions:

Shadow time time at which the first job in the queue starts execution

Extra nodes number of nodes idle when the first job in the queue starts execution

1. Go through the queue in order starting with the 2nd job.
2. Backfill a job if it will terminate by the shadow time, or it needs less than the extra nodes.

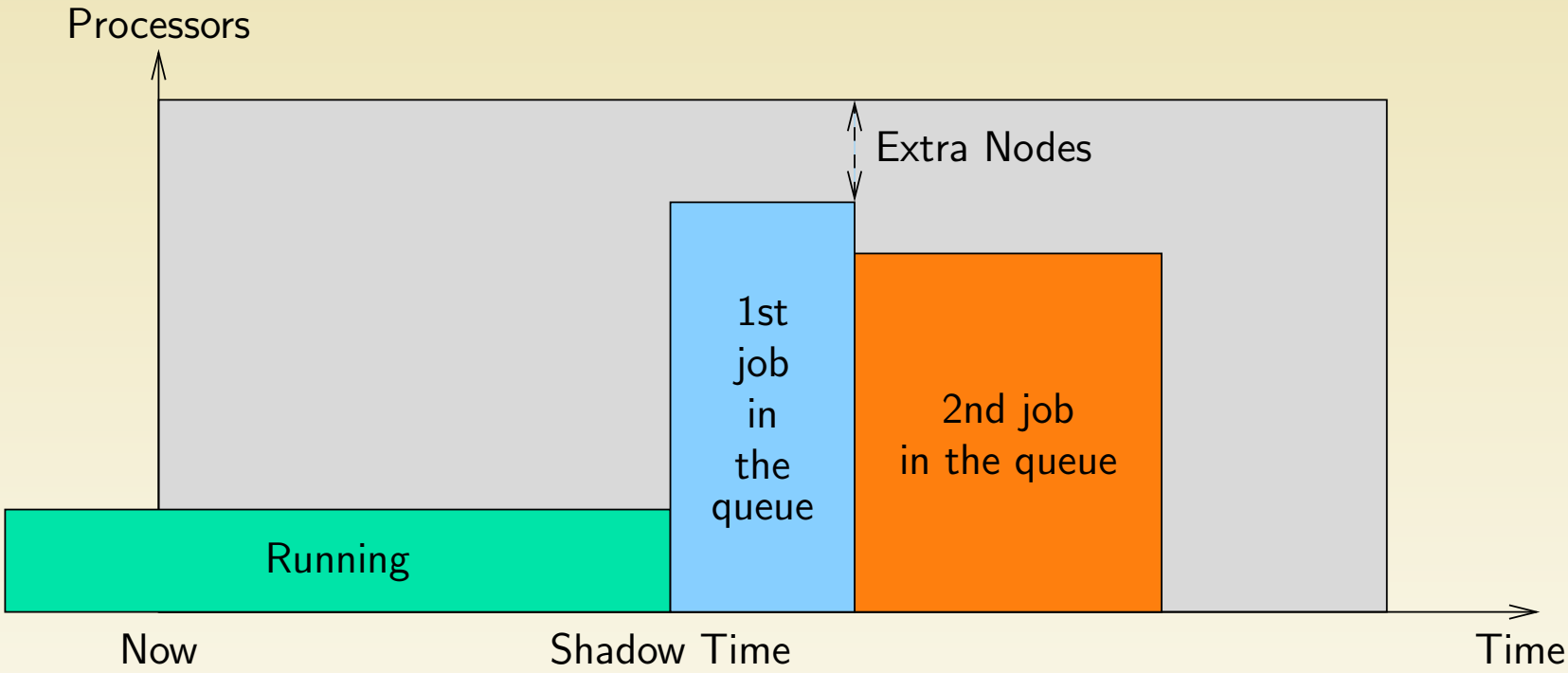
EASY



Property:

- The first job in the queue will never be delayed by backfilled jobs

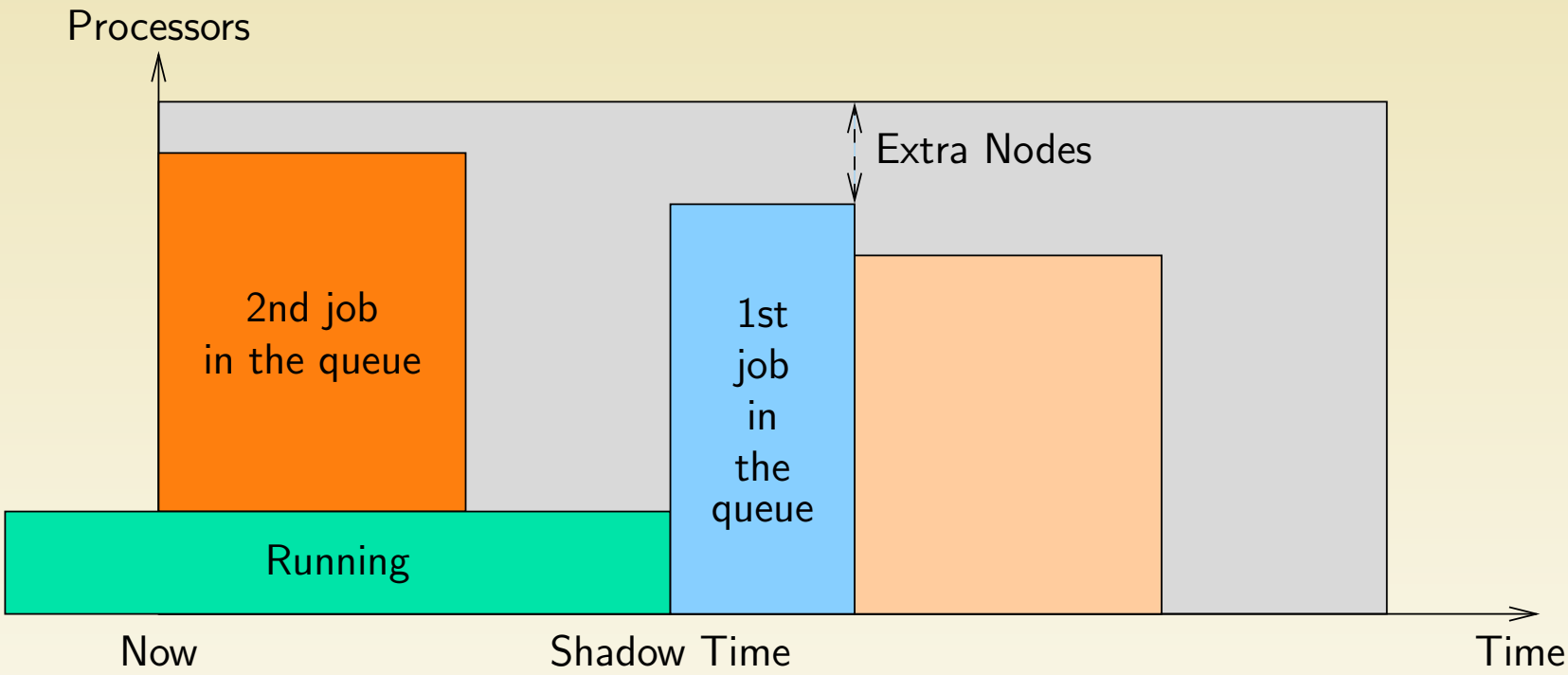
EASY



Property:

- The first job in the queue will never be delayed by backfilled jobs

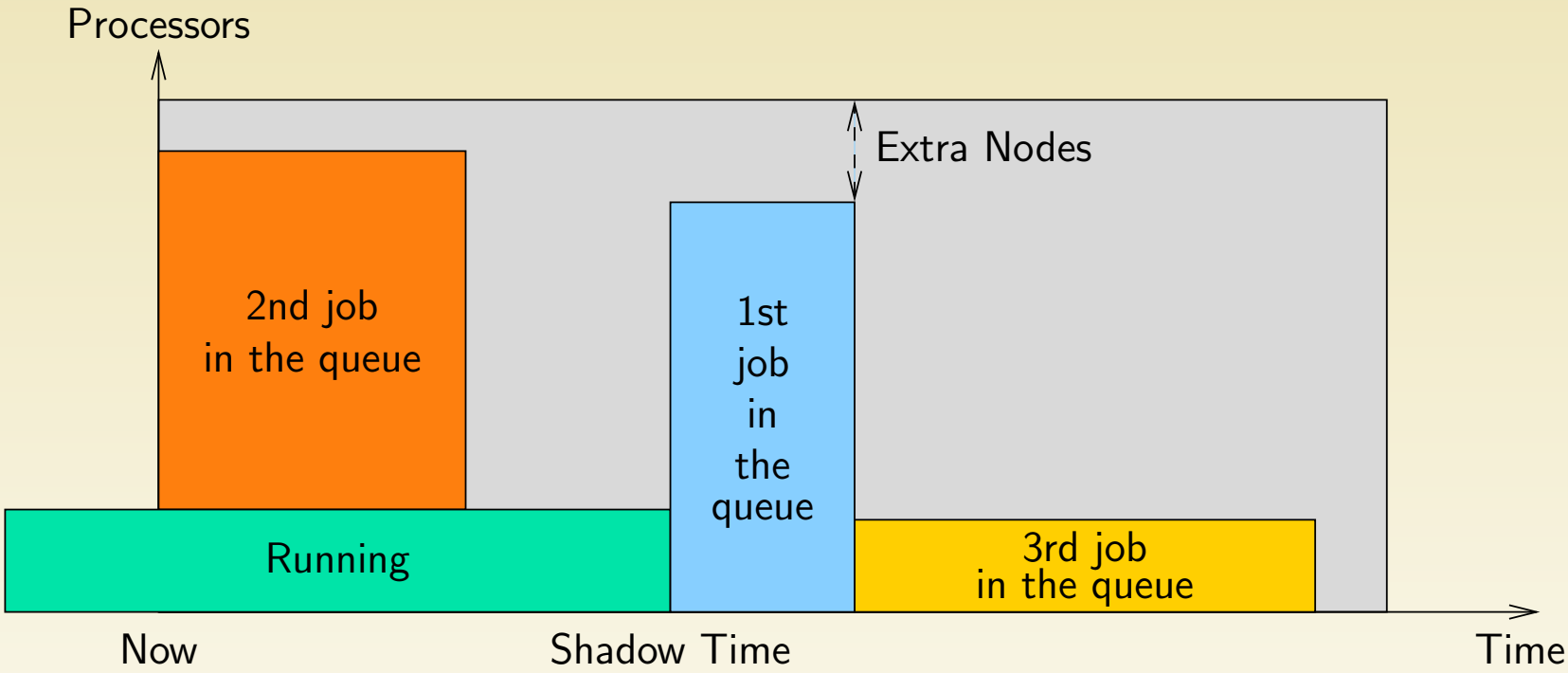
EASY



Property:

- The first job in the queue will never be delayed by backfilled jobs

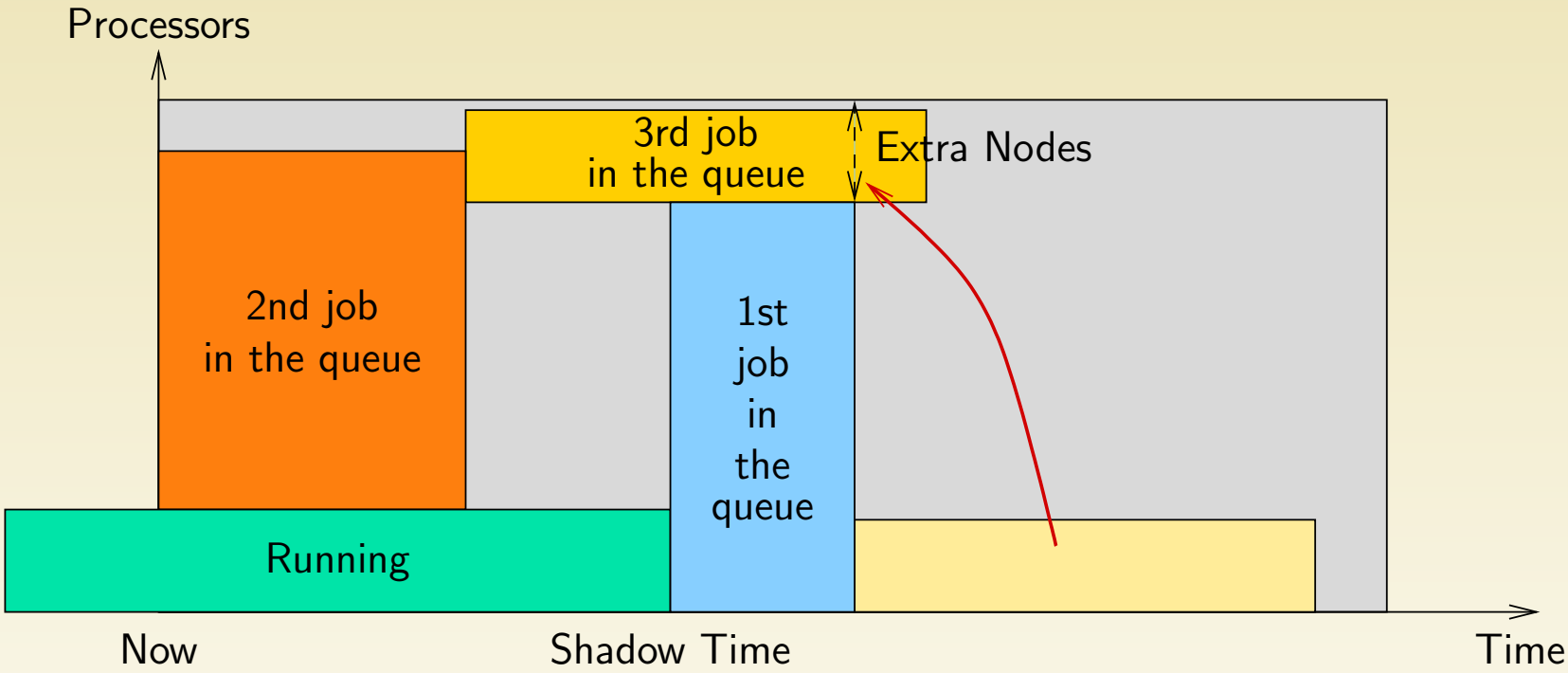
EASY



Property:

- The first job in the queue will never be delayed by backfilled jobs

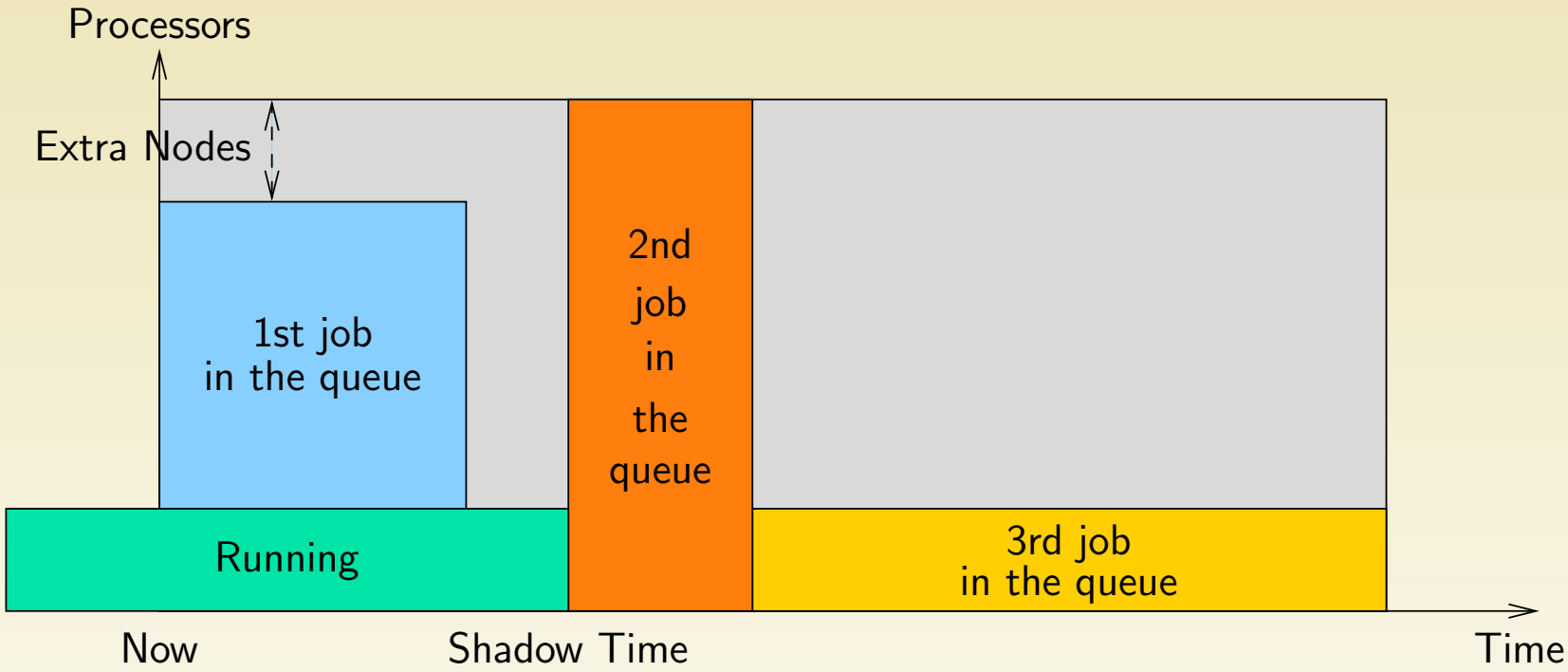
EASY



Property:

- The first job in the queue will never be delayed by backfilled jobs

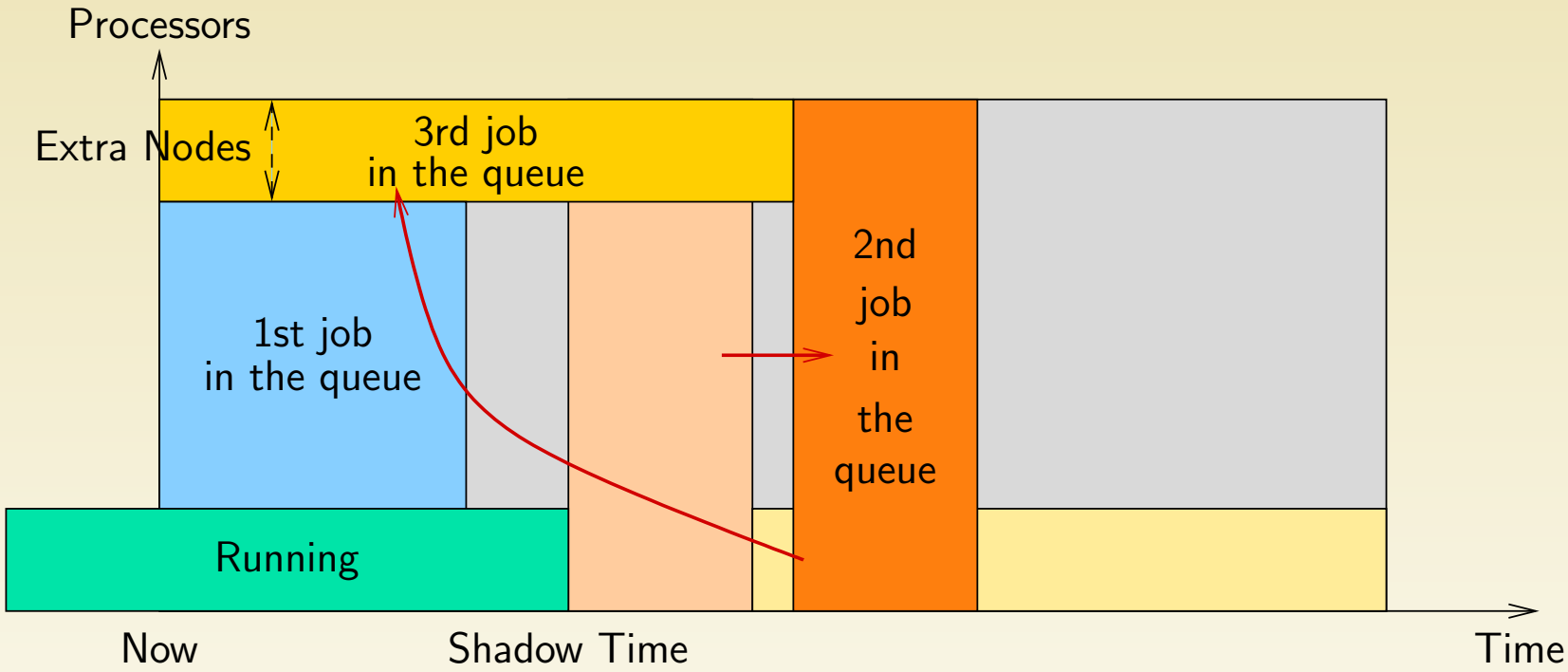
EASY



Property:

- The first job in the queue will never be delayed by backfilled jobs
- BUT, other jobs may be delayed infinitely!

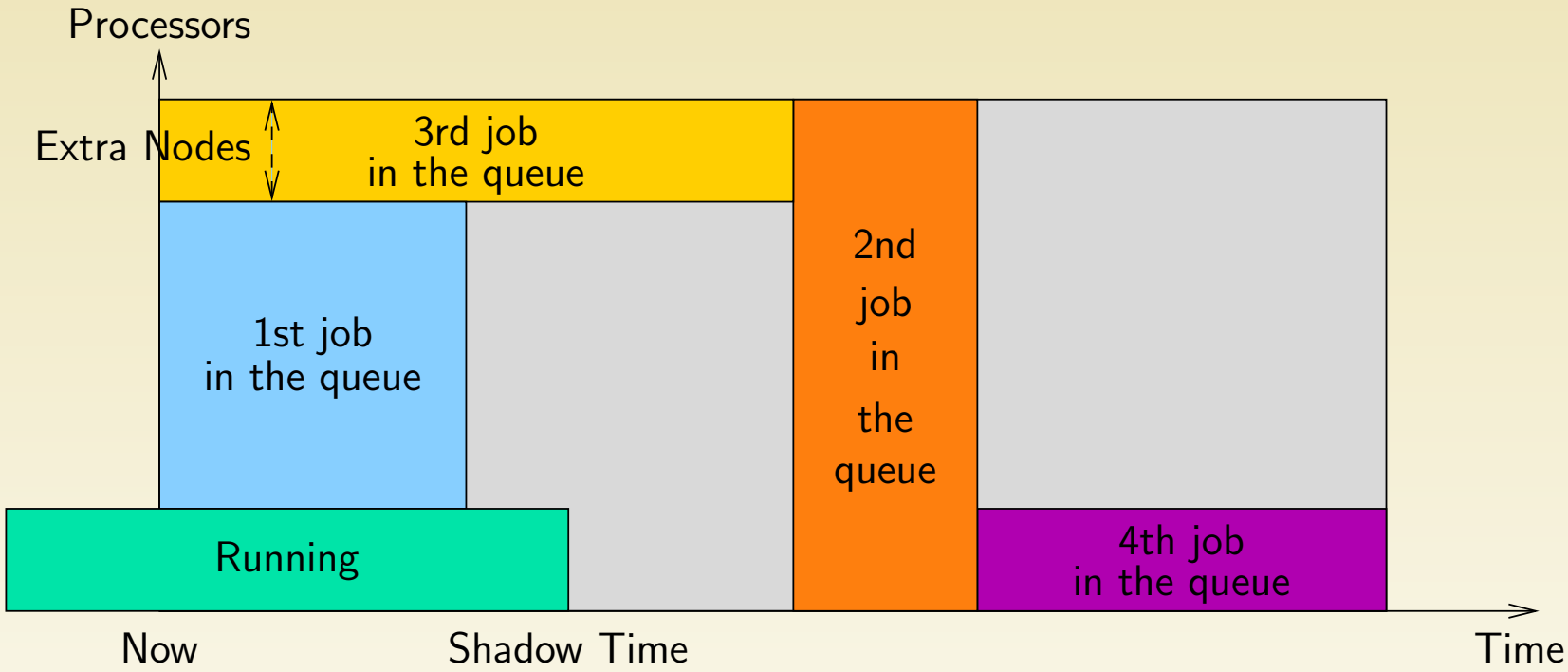
EASY



Property:

- The first job in the queue will never be delayed by backfilled jobs
- BUT, other jobs may be delayed infinitely!

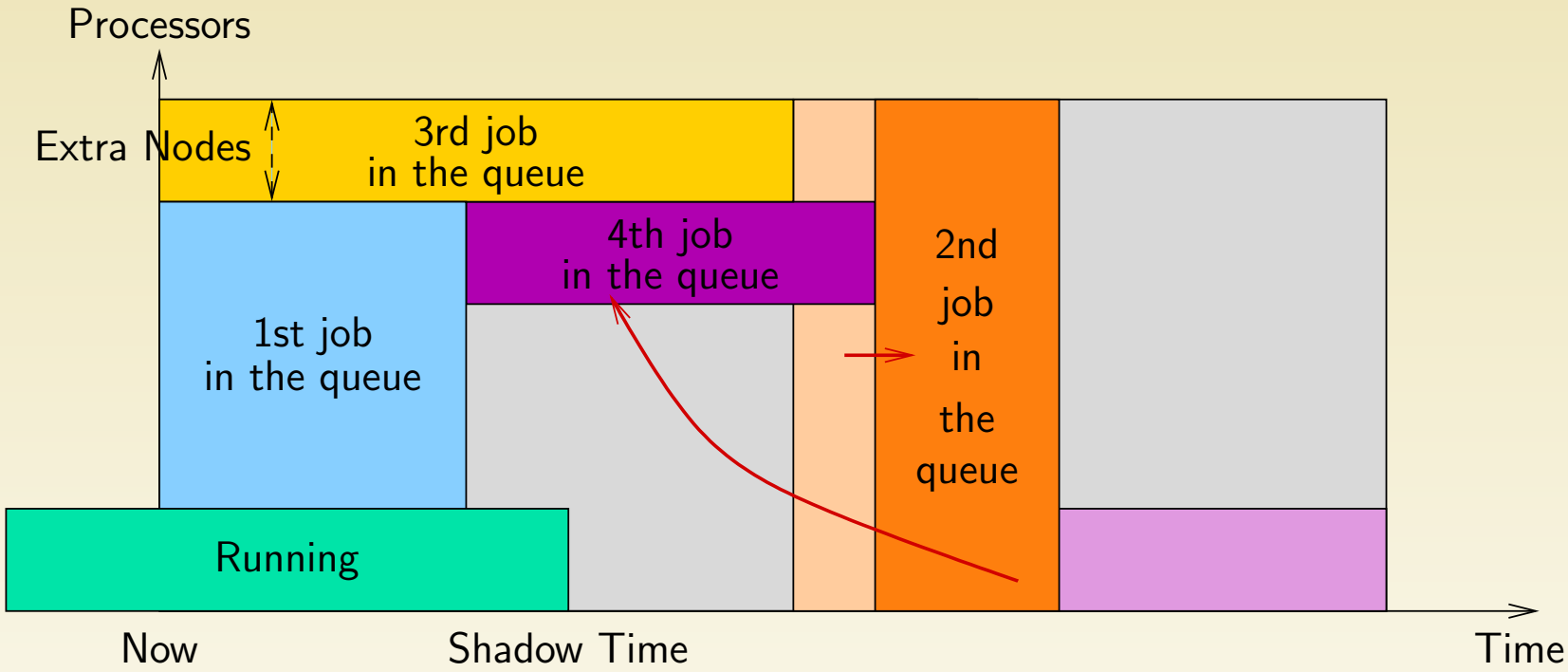
EASY



Property:

- The first job in the queue will never be delayed by backfilled jobs
- BUT, other jobs may be delayed infinitely!

EASY



Property:

- The first job in the queue will never be delayed by backfilled jobs
- BUT, other jobs may be delayed infinitely!

EASY Properties

Unbounded Delay.
- The first job in the queue will never be delayed by backfilled jobs
- BUT, other jobs may be delayed infinitely!

No Starvation.
- Delay of first job is bounded by runtime of current jobs
- When the first job finishes, the second job becomes the first job in the queue
- Once it is the first job, it cannot be delayed further

Other approach.
- Conservative Backfilling. *EVERY* job has a *reservation*. A job may be backfilled only if it does not delay any other job ahead of it in the queue.
- Fixes the unbounded delay problem that EASY has. More complicated to implement (The algorithm must find holes in the schedule) though.
- EASY favors small long jobs and harms large short jobs.

Outline

When Does Backfilling Happen?

Possibly when

- A new job arrives

When Does Backfilling Happen?

Possibly when

- A new job arrives
- The first job in the queue starts

When Does Backfilling Happen?

Possibly when

- A new job arrives
- The first job in the queue starts
- When a job finishes early

When Does Backfilling Happen?

Possibly when

- A new job arrives
- The first job in the queue starts
- When a job finishes early

Users provide job runtime estimates (Jobs are killed if they go over).

When Does Backfilling Happen?

Possibly when

- A new job arrives
- The first job in the queue starts
- When a job finishes early

Users provide job runtime estimates (Jobs are killed if they go over).
Trade-off:

- provide a conservative estimate: you goes through the queue faster (may be backfilled)
- provide a loose estimate: your job will not be killed

When Does Backfilling Happen?

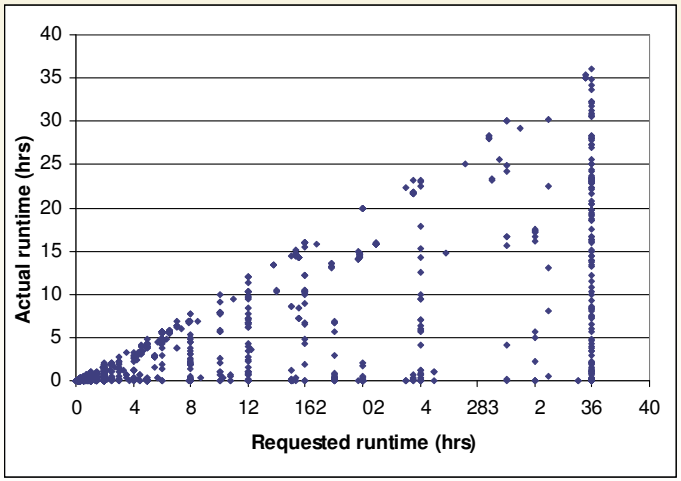
Possibly when

- A new job arrives
- The first job in the queue starts
- When a job finishes early

Users provide job runtime estimates (Jobs are killed if they go over).
Trade-off:

- provide a conservative estimate: you goes through the queue faster (may be backfilled)
- provide a loose estimate: your job will not be killed

Are estimates accurate?



How Good is the Schedule ?

All of this is great, but how do we know what a "good" schedule is? FCFS, EASY, CFB, Random?

What we need are metrics to quantify how good a schedule is. It has to be an aggregate metric over all jobs

How Good is the Schedule ?

All of this is great, but how do we know what a "good" schedule is? FCFS, EASY, CFB, Random?

What we need are metrics to quantify how good a schedule is. It has to be an aggregate metric over all jobs

1. Turn-around time or flow (Wait time + Run time).
   Job 1 needs 1h of compute time and waits 1s
   Job 2 needs 1s of compute time and waits 1h
   Clearly Job 1 is really happy, and Job 2 is not happy at all

How Good is the Schedule ?

All of this is great, but how do we know what a "good" schedule is? FCFS, EASY, CFB, Random?

What we need are metrics to quantify how good a schedule is. It has to be an aggregate metric over all jobs

1. Turn-around time or flow (Wait time + Run time).
   Job 1 needs 1h of compute time and waits 1s
   Job 2 needs 1s of compute time and waits 1h
   Clearly Job 1 is really happy, and Job 2 is not happy at all
2. Wait time (equivalent to "user happiness")
   Job 1 asks for 1 nodes and waits 1 h
   Job 2 asks for 512 nodes and waits 1h
   Again, Job 1 is unhappy while Job 2 is probably sort of happy.
   We need a metric that represents happiness for small, large, short, long jobs.

How Good is the Schedule ?

All of this is great, but how do we know what a "good" schedule is? FCFS, EASY, CFB, Random?

What we need are metrics to quantify how good a schedule is. It has to be an aggregate metric over all jobs

1. Turn-around time or flow (Wait time + Run time).
   Job 1 needs 1h of compute time and waits 1s
   Job 2 needs 1s of compute time and waits 1h
   Clearly Job 1 is really happy, and Job 2 is not happy at all
2. Wait time (equivalent to "user happiness")
   Job 1 asks for 1 nodes and waits 1 h
   Job 2 asks for 512 nodes and waits 1h
   Again, Job 1 is unhappy while Job 2 is probably sort of happy.
   We need a metric that represents happiness for small, large, short, long jobs.
3. Slowdown or Stretch (turn-around time divided by turn- around time if alone in the system)
   Doesn't really take care of the small/large problem. Could think of some scaling, but unclear !

Now What ?

Now we have a few metrics we can consider
We can run simulations of the scheduling algorithms, and see how they fare.
We need to test these algorithms in representative scenarios
Supercomputer/cluster traces. Collect the following for long periods of time:

- Time of submission
- How many nodes asked
- How much time asked
- How much time was actually used
- How much time spent in the queue

Uses of the traces:

1. Drive simulations
2. Come up with models of user behaviors

Sample Results

A type of experiments that people have done: replace user estimate by f times the actual run time
Possible to improve performance by multiplying user estimates by 2!

	EASY	CBF
Mean Slowdown		
KTH	-4.8%	-23.0%
CTC	-7.9%	-18.0%
SDSC	+4.6%	-14.2%
Mean Response time		
KTH	-3.3%	-7.0%
CTC	-0.9%	-1.6%
SDSC	-1.6%	-10.9%

Message

- These are all heuristics.
- They are not specifically designed to optimize the metrics we have designed.
- It is difficult to truly understand the reasons for the results.
- But one can derive some empirical wisdom.
- One of the reasons why one is stuck with possibly obscure heuristics is that we're dealing with an *on-line* problem: We don't know what happens next.
- We cannot wait for all jobs to be submitted to make a decision. But we can wait for a while, accumulate jobs, and schedule them together.

Summary

Batch Schedulers are what we're stuck with at the moment.
They are often hated by users.

- I submit to the queue asking for 10 nodes for 1 hour.
- I wait for two days.
- My code finally starts, but doesn't finish within 1 hour and gets killed!!

Summary

Batch Schedulers are what we're stuck with at the moment.
They are often hated by users.

- I submit to the queue asking for 10 nodes for 1 hour.
- I wait for two days.
- My code finally starts, but doesn't finish within 1 hour and gets killed!!

A lot of research, a few things happening "in the field".
When you go to a company that has clusters (like most of them), they typically have a job scheduler, so it's good to have some idea of what it is.

Summary

Batch Schedulers are what we're stuck with at the moment.
They are often hated by users.

- I submit to the queue asking for 10 nodes for 1 hour.
- I wait for two days.
- My code finally starts, but doesn't finish within 1 hour and gets killed!!

A lot of research, a few things happening "in the field".
When you go to a company that has clusters (like most of them), they typically have a job scheduler, so it's good to have some idea of what it is.
A completely different approach is gang scheduling, which we discuss next.

Outline

Outline

Gang Scheduling: Basis

- All processes belonging to a job run at the same time (the term gang denotes all processors within a job).
- Each process runs alone on each processor.
- BUT: there is rapid coordinated context switching.
- It is possible to suspend/preempt jobs arbitrarily

Gang Scheduling: Basis

- All processes belonging to a job run at the same time (the term gang denotes all processors within a job).
- Each process runs alone on each processor.
- BUT: there is rapid coordinated context switching.
- It is possible to suspend/preempt jobs arbitrarily $\leadsto$ May allow more flexibility to optimize some metrics.

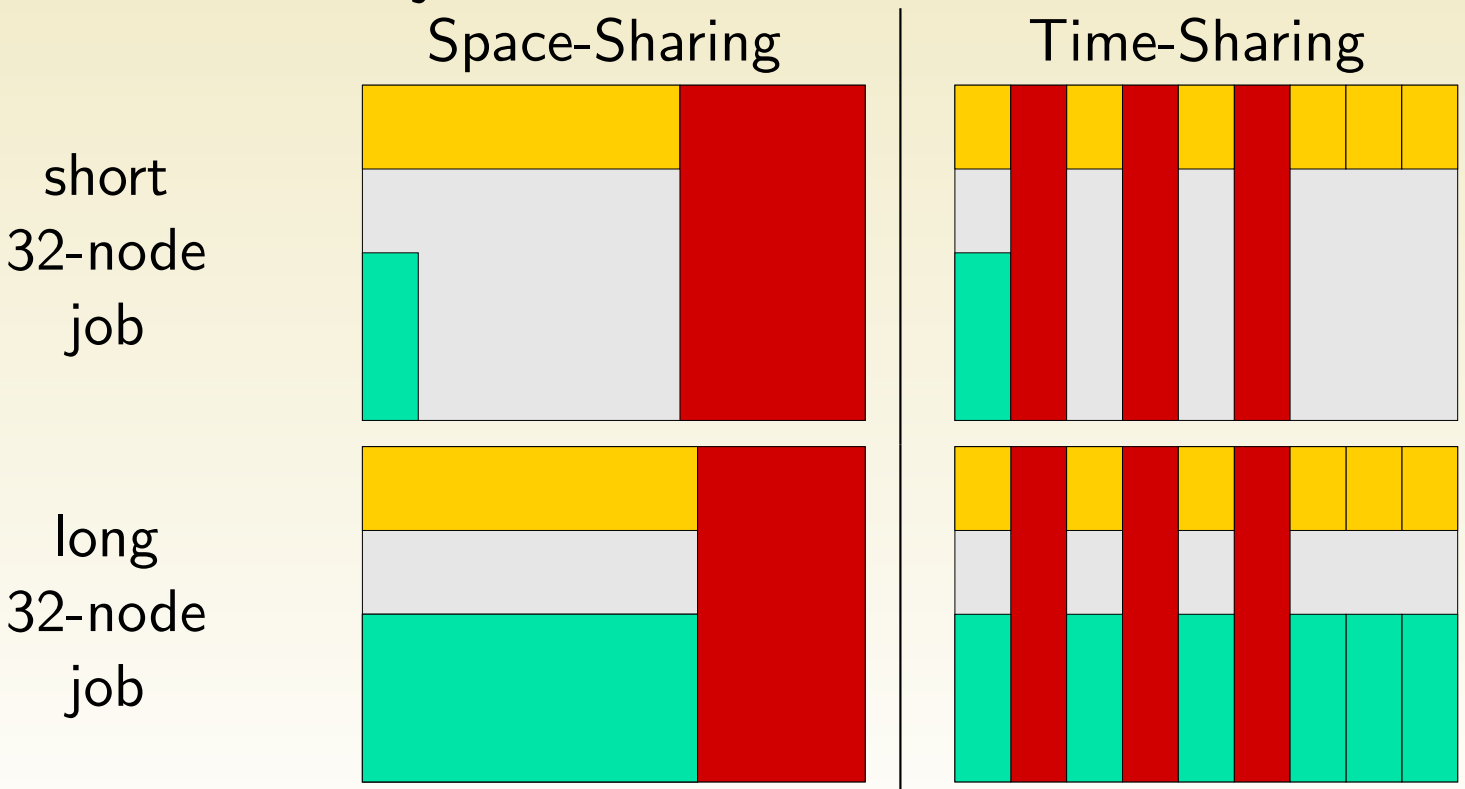
Gang Scheduling: Basis

- All processes belonging to a job run at the same time (the term gang denotes all processors within a job).
- Each process runs alone on each processor.
- BUT: there is rapid coordinated context switching.
- It is possible to suspend/preempt jobs arbitrarily $\leadsto$ May allow more flexibility to optimize some metrics.
- If processing times are not known in advance (or grossly erroneous), preemption can help short jobs that would be "stuck" behind a long job.
- Should improve machine utilization.

Gang Scheduling: an Example

- A 128 node cluster.
- A running 64-node job.
- A 32-node job and a 128-node job are queued.

Should the 32-node job be started ?



More uniform slowdown, better resource usage.

Outline

Gang Scheduling: Drawbacks

- Overhead for context switching (trade-off between overhead and fine grain).

Gang Scheduling: Drawbacks

- Overhead for context switching (trade-off between overhead and fine grain).
- Overhead for coordinating context switching across multiple processors.

Gang Scheduling: Drawbacks

- Overhead for context switching (trade-off between overhead and fine grain).
- Overhead for coordinating context switching across multiple processors.
- Reduced cache efficiency(Frequent cache flushing).

Gang Scheduling: Drawbacks

- Overhead for context switching (trade-off between overhead and fine grain).
- Overhead for coordinating context switching across multiple processors.
- Reduced cache efficiency(Frequent cache flushing).
- RAM Pressure (more jobs must fit in memory, swapping to disk causes unacceptable overhead).

Gang Scheduling: Drawbacks

- Overhead for context switching (trade-off between overhead and fine grain).
- Overhead for coordinating context switching across multiple processors.
- Reduced cache efficiency(Frequent cache flushing).
- RAM Pressure (more jobs must fit in memory, swapping to disk causes unacceptable overhead).
- Typically not used in production HPC systems (batch scheduling is preferred).

Gang Scheduling: Drawbacks

- Overhead for context switching (trade-off between overhead and fine grain).
- Overhead for coordinating context switching across multiple processors.
- Reduced cache efficiency(Frequent cache flushing).
- RAM Pressure (more jobs must fit in memory, swapping to disk causes unacceptable overhead).
- Typically not used in production HPC systems (batch scheduling is preferred).
- Some implementations (MOSIX, Kerighed).

Outline

Batch Scheduling it is then

So it seems we're stuck with batch scheduling.
Why don't we like Batch Scheduling?

Batch Scheduling it is then

So it seems we're stuck with batch scheduling.
Why don't we like Batch Scheduling? Because queue waiting times are difficult to predict.

- depends on the status of the queue
- depends on the scheduling algorithm used
- depends on all sorts of configuration parameters set by system administrator
- depends on future job completions!
- etc.

So I submit my job and then it's in limbo somewhere, which is eminently annoying to most users.

Batch Scheduling it is then

So it seems we're stuck with batch scheduling.
Why don't we like Batch Scheduling? Because queue waiting times are difficult to predict.

- depends on the status of the queue
- depends on the scheduling algorithm used
- depends on all sorts of configuration parameters set by system administrator
- depends on future job completions!
- etc.

So I submit my job and then it's in limbo somewhere, which is eminently annoying to most users.
That is why there is more and more demand for reservation support.
Users build (badly?) the schedule by themselves.

Outline

Batch Scheduling and Grids

Grids result from the collaboration of many Universities/Computing Centers.
Everyone runs its own Batch Scheduler that cannot be bypassed.
How to decide where we should submit our jobs?

Batch Scheduling and Grids

Grids result from the collaboration of many Universities/Computing Centers.
Everyone runs its own Batch Scheduler that cannot be bypassed.
How to decide where we should submit our jobs?
When in doubt, a brute-force approach is to:

- Do multiple submissions for different numbers of nodes
- Cancel all submissions but the first one that comes back
- Or possibly make some ad-hoc call regarding whether to keep a potentially poor request in the hope of getting a better one through shortly after.

Batch Scheduling and Grids

Grids result from the collaboration of many Universities/Computing Centers.
Everyone runs its own Batch Scheduler that cannot be bypassed.
How to decide where we should submit our jobs?
When in doubt, a brute-force approach is to:

- Do multiple submissions for different numbers of nodes
- Cancel all submissions but the first one that comes back
- Or possibly make some ad-hoc call regarding whether to keep a potentially poor request in the hope of getting a better one through shortly after.

What happens if everybody does this?

Batch Scheduling and Grids

Grids result from the collaboration of many Universities/Computing Centers.
Everyone runs its own Batch Scheduler that cannot be bypassed.
How to decide where we should submit our jobs?
When in doubt, a brute-force approach is to:

- Do multiple submissions for different numbers of nodes
- Cancel all submissions but the first one that comes back
- Or possibly make some ad-hoc call regarding whether to keep a potentially poor request in the hope of getting a better one through shortly after.

What happens if everybody does this?

Other issues:

- File Staging ?
- Load Balancing between sites ?

Sequential Job Scheduling for Grids

A set unrelated processors $P_1, \ldots, P_n$ and a set of sequential jobs $J_1, \ldots, J_n$ (processing time $p_{i,j}$).
Let's try a few natural scheduling strategies. We denote by a_i the time at which P_i is available (at the beginning $a_i = 0$ for all P_i):

Sequential Job Scheduling for Grids

A set unrelated processors $P_1, \ldots, P_n$ and a set of sequential jobs $J_1, \ldots, J_n$ (processing time $p_{i,j}$).
Let's try a few natural scheduling strategies. We denote by a_i the time at which P_i is available (at the beginning $a_i = 0$ for all P_i):

Min-Min Compute the minimum completion time $C_j = a_i + p_{i,j}$ of each J_j and choose the one with the smallest C_j. Update the corresponding a_i (its best host) accordingly ($a_i \leftarrow a_i + p_{i,j}$).

Sequential Job Scheduling for Grids

A set unrelated processors $P_1, \ldots, P_n$ and a set of sequential jobs $J_1, \ldots, J_n$ (processing time $p_{i,j}$).
Let's try a few natural scheduling strategies. We denote by a_i the time at which P_i is available (at the beginning $a_i = 0$ for all P_i):

Min-Min Compute the minimum completion time $C_j = a_i + p_{i,j}$ of each J_j and choose the one with the smallest C_j. Update the corresponding a_i (its best host) accordingly ($a_i \leftarrow a_i + p_{i,j}$).

Max-Min Choose J_j with the largest C_j and update the corresponding a_i (its best host) accordingly.

Sequential Job Scheduling for Grids

A set unrelated processors $P_1, \ldots, P_n$ and a set of sequential jobs $J_1, \ldots, J_n$ (processing time $p_{i,j}$).
Let's try a few natural scheduling strategies. We denote by a_i the time at which P_i is available (at the beginning $a_i = 0$ for all P_i):

Min-Min Compute the minimum completion time $C_j = a_i + p_{i,j}$ of each J_j and choose the one with the smallest C_j. Update the corresponding a_i (its best host) accordingly ($a_i \leftarrow a_i + p_{i,j}$).

Max-Min Choose J_j with the largest C_j and update the corresponding a_i (its best host) accordingly.

Sufferage S_j is the difference between the best completion time of J_j and its second best completion time. Choose the job with the largest sufferage and schedule it on its best processor.

Sequential Job Scheduling for Grids

A set unrelated processors $P_1, \ldots, P_n$ and a set of sequential jobs $J_1, \ldots, J_n$ (processing time $p_{i,j}$).
Let's try a few natural scheduling strategies. We denote by a_i the time at which P_i is available (at the beginning $a_i = 0$ for all P_i):

Min-Min Compute the minimum completion time $C_j = a_i + p_{i,j}$ of each J_j and choose the one with the smallest C_j. Update the corresponding a_i (its best host) accordingly ($a_i \leftarrow a_i + p_{i,j}$).

Max-Min Choose J_j with the largest C_j and update the corresponding a_i (its best host) accordingly.

Sufferage S_j is the difference between the best completion time of J_j and its second best completion time. Choose the job with the largest sufferage and schedule it on its best processor.

Problem: How do you get an estimate of $p_{i,j}$?

So Where are we ?

- Batch schedulers are complex pieces of software that are used in practice.
- A lot of experience on how they work and how to use them.
- But ultimately everybody knows they are an imperfect solution.
- Many view the lack of theoretical foundations as a big problem.
- Some just don't care...

> *Fools ignore complexity. Pragmatists suffer it. Some can avoid it. Geniuses remove it.*
>
> *– "Epigrams in Programming", by Alan J. Perlis of Yale University.*

Bibliography

E. G. Coffman.
Computer and job-shop scheduling theory.
John Wiley & Sons, 1976.

D.B. Shmoys, J. Wein, and D.P. Williamson.
Scheduling parallel machines on-line.
Symposium on Foundations of Computer Science, 0:131–140, 1991.