CPU Scheduling

Arnaud Legrand, CNRS, University of Grenoble

LIG laboratory, arnaud.legrand@imag.fr

November 9, 2009

Outline

Outline

Matrix Product: Sequential Version

Remember last week's introductory example.

```
{ To compute C ← C + A × B }
for i = 1 to n do
    for j = 1 to n do
        for k = 1 to n do
            C_{i,j} ← C_{i,j} + A_{i,k} × B_{k,j}
```

$$B = \begin{array}{|c|c|} \hline B_{1,1} & B_{1,2} \\ \hline B_{2,1} & B_{2,2} \\ \hline \end{array}$$

$$A = \begin{array}{|c|c|} \hline A_{1,1} & A_{1,2} \\ \hline A_{2,1} & A_{2,2} \\ \hline \end{array} \qquad C = \begin{array}{|c|c|} \hline C_{1,1} & C_{1,2} \\ \hline C_{2,1} & C_{2,2} \\ \hline \end{array}$$

Matrix Product: Sequential Version

Remember last week's introductory example.

2 $C_{1,1} \leftarrow C_{1,1} + A_{1,1} \times B_{1,1}$

4 $C_{1,1} \leftarrow C_{1,1} + A_{1,2} \times B_{2,1}$

6 $C_{1,2} \leftarrow C_{1,2} + A_{1,1} \times B_{1,2}$

8 $C_{1,2} \leftarrow C_{1,2} + A_{1,2} \times B_{2,2}$

9 $\ldots$

$B_{1,1}$	$B_{1,2}$
$B_{2,1}$	$B_{2,2}$

CPU	2	4	6	8

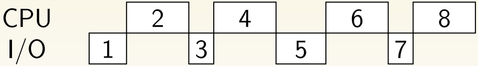
$A_{1,1}$	$A_{1,2}$
$A_{2,1}$	$A_{2,2}$

$C_{1,1}$	$C_{1,2}$
$C_{2,1}$	$C_{2,2}$

Remember last week's introductory example.

1. Load $C_{1,1}$, $A_{1,1}$, $B_{1,1}$
2. $C_{1,1} \leftarrow C_{1,1} + A_{1,1} \times B_{1,1}$
3. Unload $A_{1,1}$, $B_{1,1}$. Load $A_{1,2}$, $B_{2,1}$
4. $C_{1,1} \leftarrow C_{1,1} + A_{1,2} \times B_{2,1}$
5. Unload $C_{1,1}$, $A_{1,2}$, $B_{2,1}$. Load $C_{1,2}$, $A_{1,1}$, $B_{1,2}$
6. $C_{1,2} \leftarrow C_{1,2} + A_{1,1} \times B_{1,2}$
7. Unload $A_{1,1}$, $B_{1,2}$
8. $C_{1,2} \leftarrow C_{1,2} + A_{1,2} \times B_{2,2}$
9. ...

$B_{1,1}$	$B_{1,2}$
$B_{2,1}$	$B_{2,2}$



$A_{1,1}$	$A_{1,2}$
$A_{2,1}$	$A_{2,2}$

$C_{1,1}$	$C_{1,2}$
$C_{2,1}$	$C_{2,2}$

Matrix Product: Sequential Version

Remember last week's introductory example.

Load $C_{1,1}$, $A_{1,1}$, $B_{1,1}$
$C_{1,1} \leftarrow C_{1,1} + A_{1,1} \times B_{1,1}$
Unload $A_{1,1}$, $B_{1,1}$. Load $A_{1,2}$, $B_{2,1}$
$C_{1,1} \leftarrow C_{1,1} + A_{1,2} \times B_{2,1}$
Unload $C_{1,1}$, $A_{1,2}$, $B_{2,1}$. Load $C_{1,2}$, $A_{1,1}$, $B_{1,2}$
$C_{1,2} \leftarrow C_{1,2} + A_{1,1} \times B_{1,2}$
Unload $A_{1,1}$, $B_{1,2}$
$C_{1,2} \leftarrow C_{1,2} + A_{1,2} \times B_{2,2}$

$B_{1,1}$	$B_{1,2}$

Sequential Programs

Sequential programs are generally a succession of CPU burst and I/O burst.

- A CPU-bound program has long CPU-burst.
- An I/O-bound program has short CPU-burst.

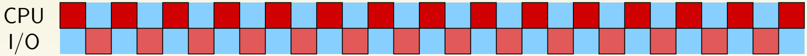
CPU scheduling

- One objective of multi-programming is to maximize CPU utilization i.e. to have process running at all time.
- Scheduling of this kind is a fundamental OS function and has to be *fast* (otherwise you will waste useful CPU cycles) and *fair* (somehow).
- By cleverly mixing I/O bound and CPU bound process, we *could* achieve an "optimal" resource utilization.

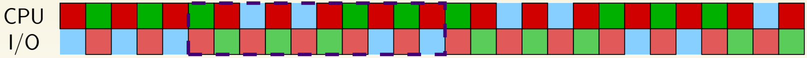
- One objective of multi-programming is to maximize CPU utilization i.e. to have process running at all time.
- Scheduling of this kind is a fundamental OS function and has to be *fast* (otherwise you will waste useful CPU cycles) and *fair* (somehow).
- By cleverly mixing I/O bound and CPU bound process, we *could* achieve an "optimal" resource utilization.

CPU scheduling

- One objective of multi-programming is to maximize CPU utilization i.e. to have process running at all time.
- Scheduling of this kind is a fundamental OS function and has to be *fast* (otherwise you will waste useful CPU cycles) and *fair* (somehow).
- By cleverly mixing I/O bound and CPU bound process, we *could* achieve an “optimal” resource utilization.



- One objective of multi-programming is to maximize CPU utilization i.e. to have process running at all time.
- Scheduling of this kind is a fundamental OS function and has to be *fast* (otherwise you will waste useful CPU cycles) and *fair* (somehow).
- By cleverly mixing I/O bound and CPU bound process, we *could* achieve an “optimal” resource utilization.



- One objective of multi-programming is to maximize CPU utilization i.e. to have process running at all time.
- Scheduling of this kind is a fundamental OS function and has to be *fast* (otherwise you will waste useful CPU cycles) and *fair* (somehow).
- By cleverly mixing I/O bound and CPU bound process, we *could* achieve an "optimal" resource utilization.

Let us forget about I/O for this talk

Let us consider that all our tasks or process are only CPU-bound.
Can you propose a scheduling problem for this setting using the Graham notation?

Let us forget about I/O for this talk

Let us consider that all our tasks or process are only CPU-bound. Can you propose a scheduling problem for this setting using the Graham notation?

- Sequential jobs
- A single CPU
- Short-Term scheduler
- Preemption is allowed
- Online: jobs (tasks/process/requests) arrive one after the other in the system

$$\langle 1 | r_j, \mathit{pmtn} | \ldots \rangle$$

Preemption has a cost but we will ignore it. Does the CPU utilization metric still makes sense ?

Let us forget about I/O for this talk

Let us consider that all our tasks or process are only CPU-bound. Can you propose a scheduling problem for this setting using the Graham notation?

- Sequential jobs
- A single CPU
- Short-Term scheduler
- Preemption is allowed
- Online: jobs (tasks/process/requests) arrive one after the other in the system

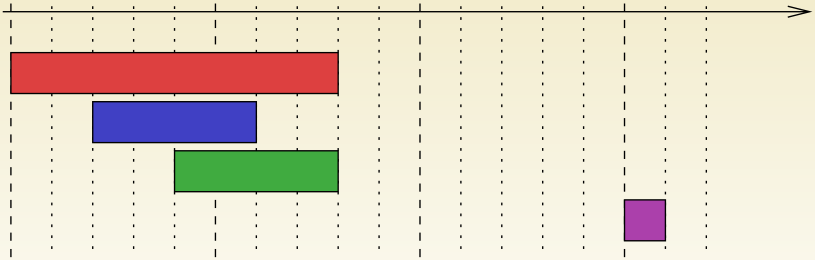
$$\langle 1 | r_j, \mathit{pmtn} | \ldots \rangle$$

Preemption has a cost but we will ignore it. Does the CPU utilization metric still makes sense ?
In the remainder of this talk, we will study this problem for different performance criteria, try to get some intuition and design the optimal strategy.

Outline

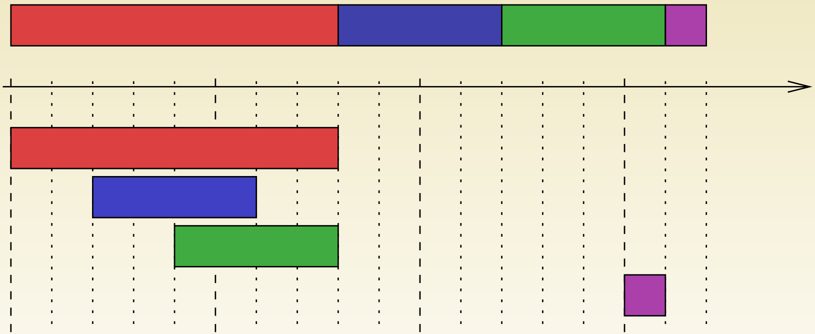
Let's play with a small example

We wish to find a schedule (possibly using preemption) that has the smallest possible max flow $(\max_i C_i - r_i)$.



Let's play with a small example

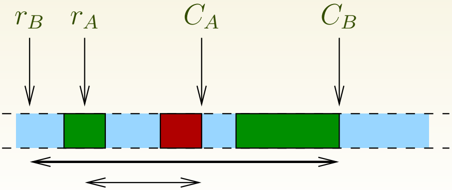
We wish to find a schedule (possibly using preemption) that has the smallest possible max flow ($\max_i C_i - r_i = 12$).



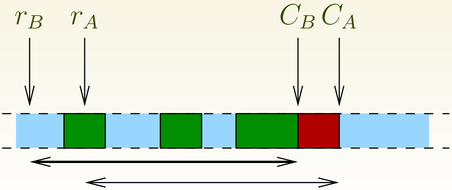
First-Come First-Served seems to be optimal.

FCFS is optimal: sketch of the proof

- Let us consider an optimal schedule σ. Let us assume that there are two jobs A and B that are not scheduled according to the FCFS policy, i.e. $r_B < r_A$ and $C_A < C_B$.

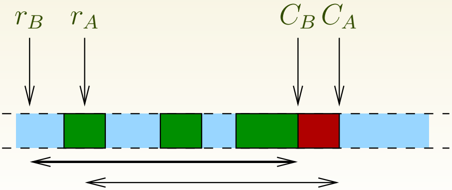


- Let us consider an optimal schedule σ. Let us assume that there are two jobs A and B that are not scheduled according to the FCFS policy, i.e. $r_B < r_A$ and $C_A < C_B$.
- By scheduling B before A, we do not increase $\max(C_A - r_A, C_B - r_B)$.



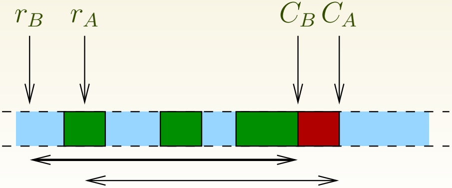
FCFS is optimal: sketch of the proof

- Let us consider an optimal schedule σ. Let us assume that there are two jobs A and B that are not scheduled according to the FCFS policy, i.e. $r_B < r_A$ and $C_A < C_B$.
- By scheduling B before A, we do not increase $\max(C_A - r_A, C_B - r_B)$.
- Therefore, by scheduling A and B according to the FCFS policy, we get a new schedule σ' that is still optimal.



FCFS is optimal: sketch of the proof

- Let us consider an optimal schedule σ. Let us assume that there are two jobs A and B that are not scheduled according to the FCFS policy, i.e. $r_B < r_A$ and $C_A < C_B$.
- By scheduling B before A, we do not increase $\max(C_A - r_A, C_B - r_B)$.
- Therefore, by scheduling A and B according to the FCFS policy, we get a new schedule σ' that is still optimal.
- By proceeding similarly for all pairs of jobs, we prove that FCFS is optimal. $\square$



FCFS is optimal: sketch of the proof

- Let us consider an optimal schedule σ. Let us assume that there are two jobs A and B that are not scheduled according to the FCFS policy, i.e. $r_B < r_A$ and $C_A < C_B$.
- By scheduling B before A, we do not increase $\max(C_A - r_A, C_B - r_B)$.
- Therefore, by scheduling A and B according to the FCFS policy, we get a new schedule σ' that is still optimal.
- By proceeding similarly for all pairs of jobs, we prove that FCFS is optimal. □

We do not even need to preempt jobs! Note that when you have *more than one processor*, things are more complicated:

Bad News NP-complete with no preemption.

Good news Polynomial algorithm with preemption but it is much more complicated than FCFS.

Interesting properties

- The FCFS scheduling policy is non-clairvoyant, easy to implement, and does not use preemption.

Interesting properties

- The FCFS scheduling policy is non-clairvoyant, easy to implement, and does not use preemption.
- The FCFS policy is optimal for minimizing $\max F_i$. It minimizes the response time!

Interesting properties

- The FCFS scheduling policy is non-clairvoyant, easy to implement, and does not use preemption.
- The FCFS policy is optimal for minimizing $\max F_i$. It minimizes the response time!
- Still, nobody would say it is a "reactive" scheduling algorithm.

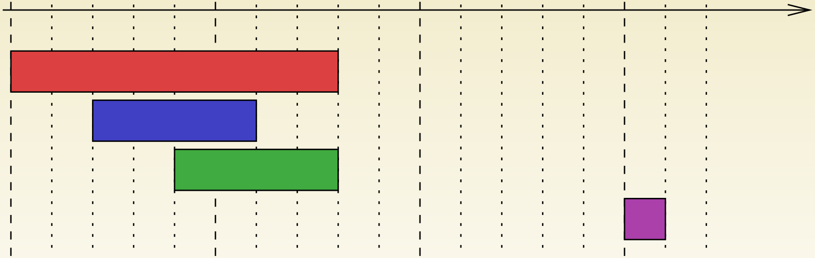
Interesting properties

- The FCFS scheduling policy is non-clairvoyant, easy to implement, and does not use preemption.
- The FCFS policy is optimal for minimizing $\max F_i$. It minimizes the response time!
- Still, nobody would say it is a "reactive" scheduling algorithm.
- Maybe focusing on the worst case (i.e. $\max$) is a bad idea...
  We would accept to sacrifice some jobs to get something more "reactive".

Outline

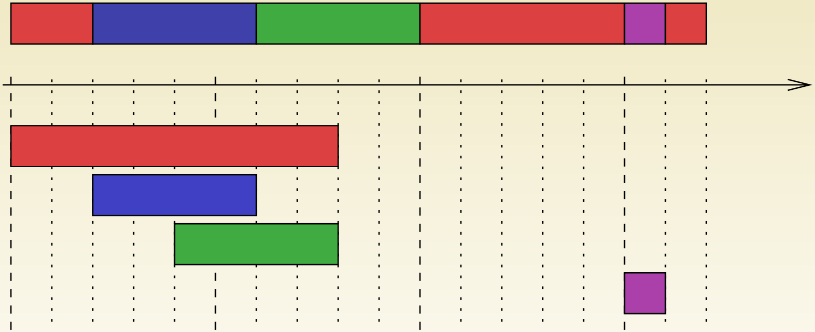
Let's play with a small example

We wish to find a schedule (possibly using preemption) that has the smallest possible sum flow ($\sum_i C_i - r_i$).



Let's play with a small example

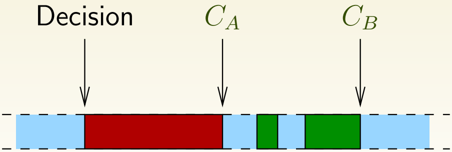
We wish to find a schedule (possibly using preemption) that has the smallest possible sum flow ($\sum_i C_i - r_i = 28$).



Shortest Remaining Processing Timer first seems to be optimal.

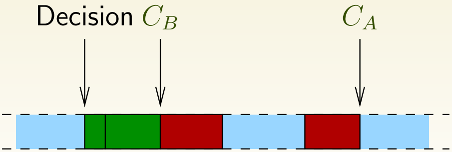
SRPT is optimal: sketch of the proof

- Let us consider an optimal schedule σ. Let us assume that there are two jobs A and B that are not scheduled according to the SRPT policy, i.e. $C_A < C_B$ and at some point there were more work to finish A than to finish B.



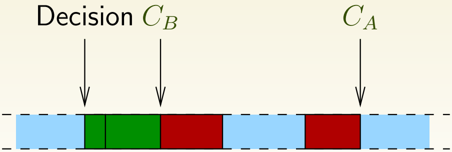
SRPT is optimal: sketch of the proof

- Let us consider an optimal schedule σ. Let us assume that there are two jobs A and B that are not scheduled according to the SRPT policy, i.e. $C_A < C_B$ and at some point there were more work to finish A than to finish B.
- By scheduling B before A, we strictly decrease $C_A + C_B$ and thus we strictly decrease the total flow.



SRPT is optimal: sketch of the proof

- Let us consider an optimal schedule σ. Let us assume that there are two jobs A and B that are not scheduled according to the SRPT policy, i.e. $C_A < C_B$ and at some point there were more work to finish A than to finish B.
- By scheduling B before A, we strictly decrease $C_A + C_B$ and thus we strictly decrease the total flow.
- Therefore, the original schedule was not optimal! The only optimal schedule is thus SRPT. $\square$



SRPT is optimal: sketch of the proof

- Let us consider an optimal schedule σ. Let us assume that there are two jobs A and B that are not scheduled according to the SRPT policy, i.e. $C_A < C_B$ and at some point there were more work to finish A than to finish B.
- By scheduling B before A, we strictly decrease $C_A + C_B$ and thus we strictly decrease the total flow.
- Therefore, the original schedule was not optimal! The only optimal schedule is thus SRPT. $\square$

Here, preemption is required!

Bad News NP-complete for multiple processors or with no preemption.

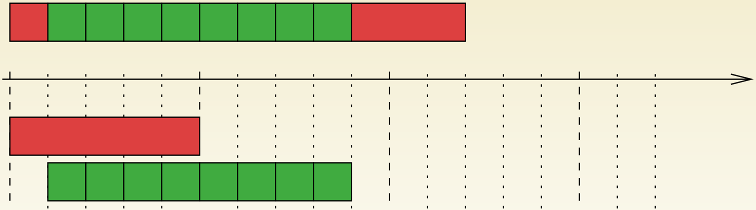
Good News Algorithm with logarithmic competitive ratio on multiple processors exists.

Comments

- Scheduling small jobs first is good for “reactivity” but it requires to know the size of the jobs (i.e. clairvoyant).
- Scheduling small jobs first is good for the average response time but some jobs may be left behind...

Comments

- Scheduling small jobs first is good for "reactivity" but it requires to know the size of the jobs (i.e. clairvoyant).
- Scheduling small jobs first is good for the average response time but large jobs may be left behind...



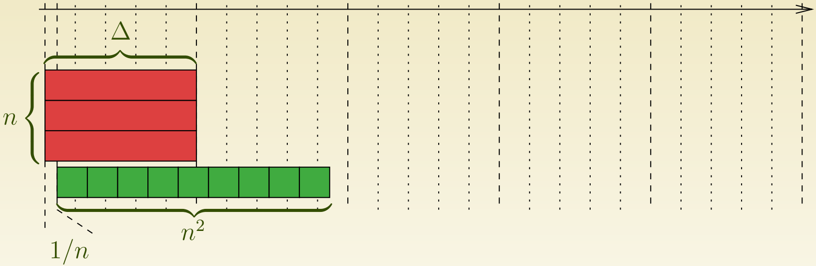
- Do you know an algorithm where job cannot starve?

FCFS is Δ-competitive for $\sum F_i$

Δ: ratio of the sizes of the largest and smallest job.
Let's prove FCFS is at most Δ-competitive.

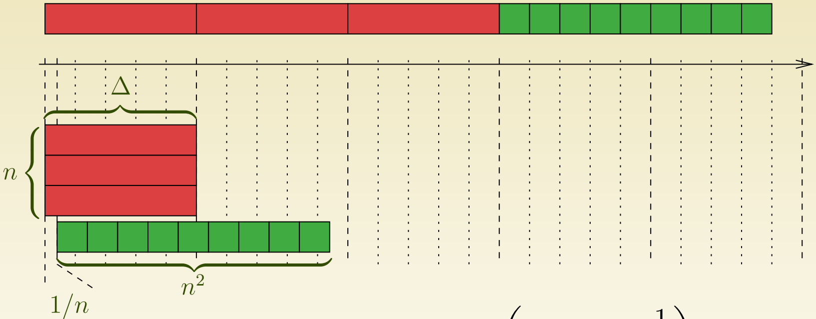
FCFS is Δ-competitive for $\sum F_i$

Δ: ratio of the sizes of the largest and smallest job.
Let's prove FCFS is at most Δ-competitive.



FCFS is Δ-competitive for $\sum F_i$

Δ: ratio of the sizes of the largest and smallest job.
Let's prove FCFS is at most Δ-competitive.

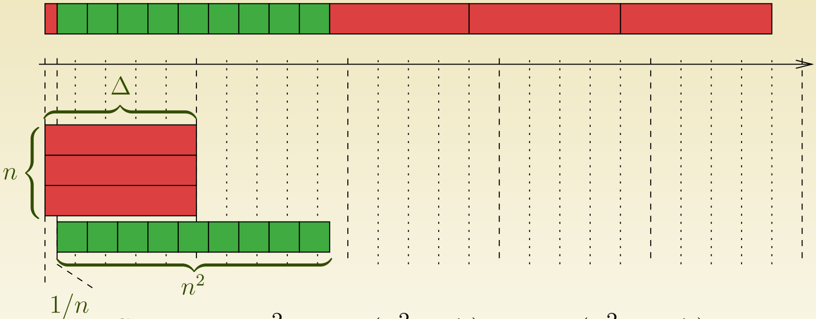


$$SF_{FCFS} = \Delta + \ldots + n\Delta + n^2\left(1 + n\Delta - \frac{1}{n}\right)$$

$$= \frac{2n^3\Delta + n^2(2+\Delta) + n(\Delta - 2)}{2}$$

FCFS is Δ-competitive for $\sum F_i$

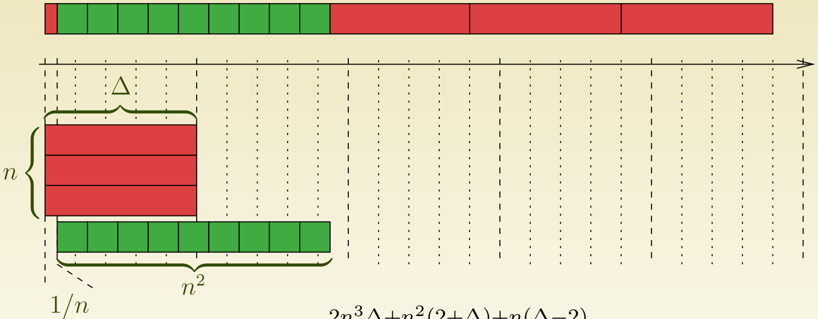
Δ: ratio of the sizes of the largest and smallest job.
Let's prove FCFS is at most Δ-competitive.



$$SF_{SRPT} = n^2 \times 1 + (n^2 + \Delta) + \ldots + (n^2 + n\Delta)$$
$$= n^3 + n^2 + \frac{n(n+1)}{2}\Delta$$

FCFS is Δ-competitive for $\sum F_i$

Δ: ratio of the sizes of the largest and smallest job.
Let's prove FCFS is at most Δ-competitive.

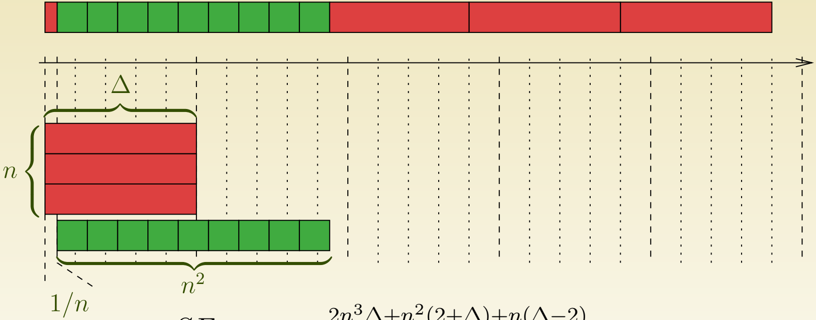


$$\varrho_{FCFS}(\Delta) \geqslant \frac{SF_{FCFS}}{SF_{SRPT}} = \frac{\frac{2n^3\Delta+n^2(2+\Delta)+n(\Delta-2)}{2}}{n^3+n^2+\frac{n(n+1)}{2}\Delta}$$

$$= \frac{2n^3\Delta+n^2(2+\Delta)+n(\Delta-2)}{2n^3+2n^2+n(n+1)\Delta} \xrightarrow[n\to+\infty]{} \Delta$$

FCFS is Δ-competitive for $\sum F_i$

Δ: ratio of the sizes of the largest and smallest job.
Let's prove FCFS is at most Δ-competitive.



$$\varrho_{FCFS}(\Delta) \geqslant \frac{SF_{FCFS}}{SF_{SRPT}} = \frac{\frac{2n^3\Delta + n^2(2+\Delta)+n(\Delta-2)}{2}}{n^3 + n^2 + \frac{n(n+1)}{2}\Delta}$$

$$= \frac{2n^3\Delta + n^2(2+\Delta) + n(\Delta-2)}{2n^3 + 2n^2 + n(n+1)\Delta} \xrightarrow[n\to+\infty]{} \Delta$$

FCFS is at exactly Δ-competitive. ▸ Proof details.

Optimizing the average response time with no starvation?

Theorem 1.

Consider any online algorithm whose competitive ratio for average flow minimization satisfies $\varrho(\Delta) < \Delta$.

There exists for this algorithm a sequence of jobs leading to starvation, and for which the maximum flow can be as far as we want from the optimal maximum flow. ▸ Proof details

The starvation issue is inherent to the optimization of the average response time.

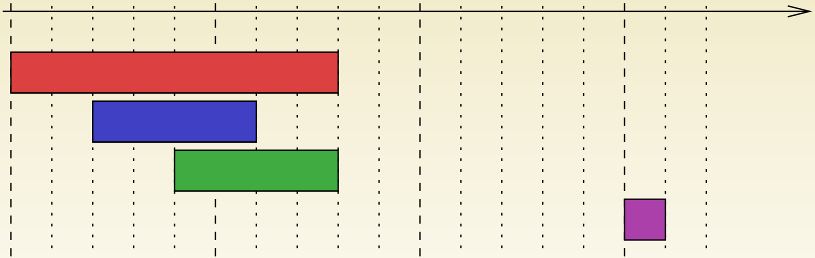
Still, we would like something "reactive" and we like the idea that short jobs have a higher priority.

It probably means that the "response time" is not the right metric.

Outline

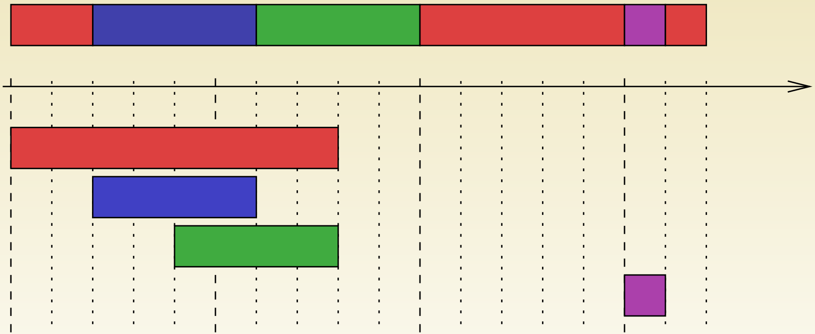
Let's play with a small example

We wish to find a schedule (possibly using preemption) that has the smallest possible sum stretch ($\sum_i \frac{C_i - r_i}{p_i}$).



Let's play with a small example

We wish to find a schedule (possibly using preemption) that has the smallest possible sum stretch ($\sum_i \frac{C_i - r_i}{p_i} = \frac{45}{8} \approx 5.625$).

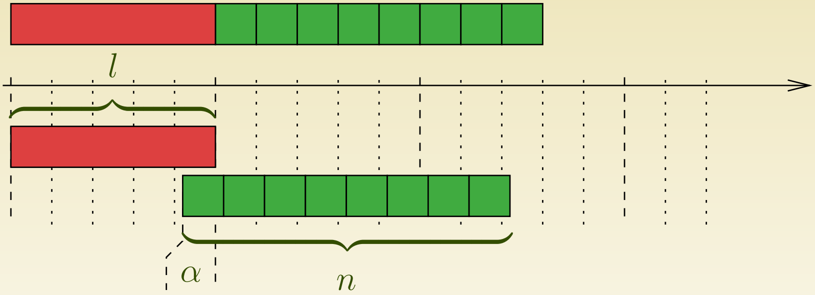


Shortest Remaining Processing Timer first seems to be optimal again.

SRPT is 2-competitive

Let's prove SRPT is at most 2-competitive.

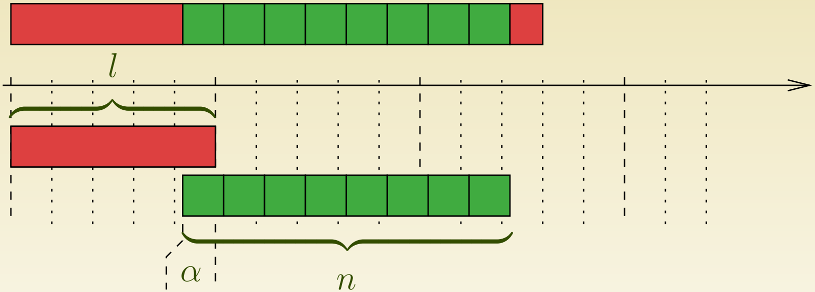
Let's prove SRPT is at most 2-competitive.



$$SS_{SRPT} = 1 + n(1 + \alpha)$$

SRPT is 2-competitive

Let's prove SRPT is at most 2-competitive.

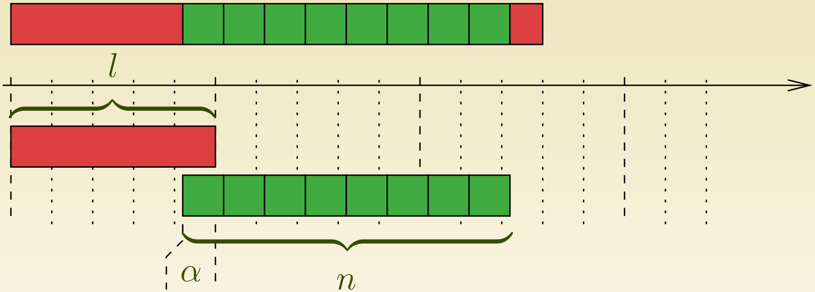


$$SS_{SRPT} = 1 + n(1 + \alpha)$$

$$SS_{opt} = \frac{l+n}{l} + n = 1 + n\left(1 + \frac{1}{l}\right)$$

SRPT is 2-competitive

Let's prove SRPT is at most 2-competitive.



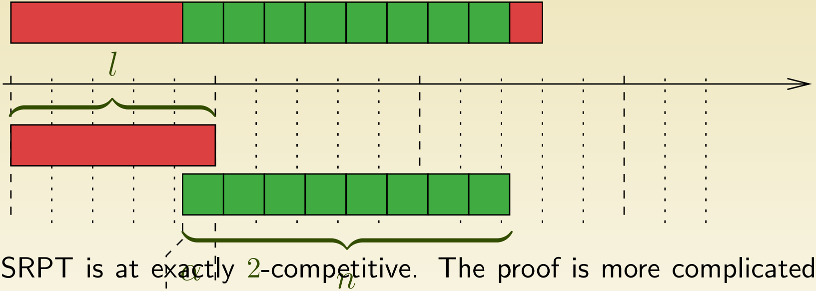
$$SS_{SRPT} = 1 + n(1+\alpha)$$

$$SS_{opt} = \frac{l+n}{l} + n = 1 + n\left(1 + \frac{1}{l}\right)$$

$$\varrho_{SRPT}(n) \geqslant \frac{1+n(1+\alpha)}{1+n(1+1/l)} \xrightarrow[n\to+\infty]{} \frac{1+\alpha}{1+1/l} \xrightarrow[\substack{l\to+\infty \\ \alpha\to 1^-}]{} 2$$

SRPT is 2-competitive

Let's prove SRPT is at most 2-competitive.



SRPT is at exactly 2-competitive. The proof is more complicated though.

Comments

- One might want to adapt SRPT to this new criteria $\rightsquigarrow$ SWRPT: always schedule the task with the smallest (remaining processing time)$\times$(processing time).
- SWRPT is not optimal either. It is exactly 2-competitive. However one can prove that it is optimal when there are only two distinct job sizes!
- Actually, the complexity (P vs. NP-complete, the offline setting) of this problem is still open.
- In the online setting SRPT and SWRPT are the algorithms with the best competitive ratio but again they both may lead to starvation.
- Do you know an algorithm where job cannot starve?

Optimizing the average stretch with no starvation?

One can easily prove that FCFS is Δ^2 competitive.

Theorem 2.

Consider any online algorithm whose competitive ratio for average stretch minimization satisfies $\varrho(\Delta) < \Delta^2$.

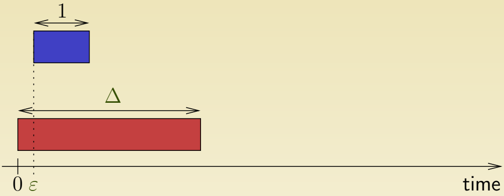
There exists for this algorithm a sequence of jobs leading to starvation, and for which the maximum stretch can be as far as we want from the optimal stretch flow. Proof details

Again the starvation issue is inherent to the optimization of the average stretch.

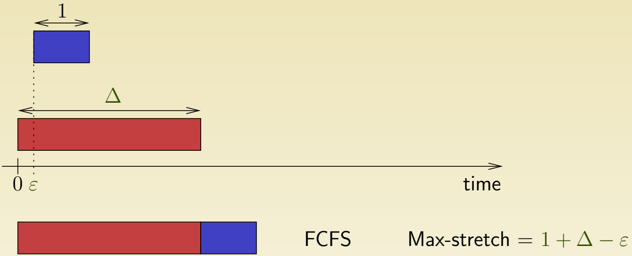
Outline

FCFS competitiveness

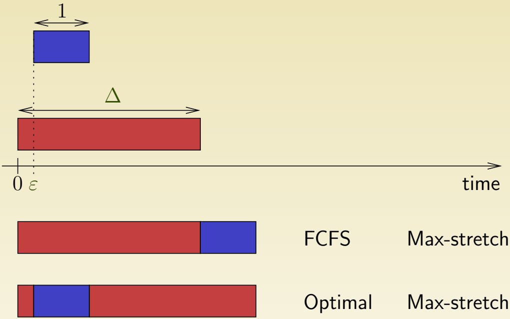
FCFS competitiveness



FCFS competitiveness

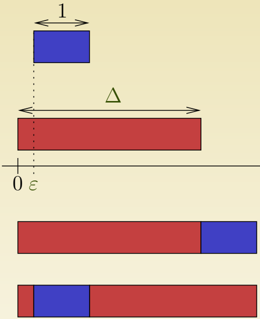


FCFS competitiveness



FCFS	Max-stretch $= 1 + \Delta - \varepsilon$	
Optimal	Max-stretch $= \frac{1+\Delta}{\Delta}$	

FCFS competitiveness

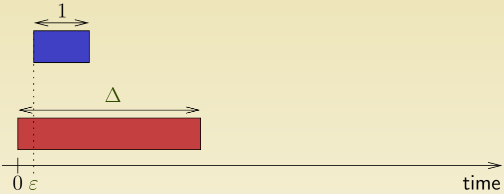


FCFS Max-stretch $= 1 + \Delta - \varepsilon$

Optimal Max-stretch $= \frac{1+\Delta}{\Delta}$

Competitive ratio: $\frac{1+\Delta-\varepsilon}{\frac{1+\Delta}{\Delta}} = \Delta \frac{1+\Delta-\varepsilon}{1+\Delta} = \Delta - \varepsilon \frac{\Delta}{1+\Delta} \geqslant \Delta - \varepsilon.$

FCFS competitiveness



FCFS Max-stretch $= 1 + \Delta - \varepsilon$

Optimal Max-stretch $= \frac{1+\Delta}{\Delta}$

Competitive ratio: $\frac{1+\Delta-\varepsilon}{\frac{1+\Delta}{\Delta}} = \Delta \frac{1+\Delta-\varepsilon}{1+\Delta} = \Delta - \varepsilon \frac{\Delta}{1+\Delta} \geqslant \Delta - \varepsilon.$

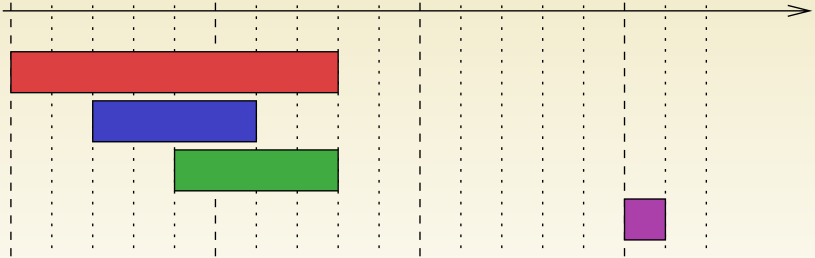
Theorem 3.

FCFS is Δ competitive for maximum stretch minimization.

▸ Proof details

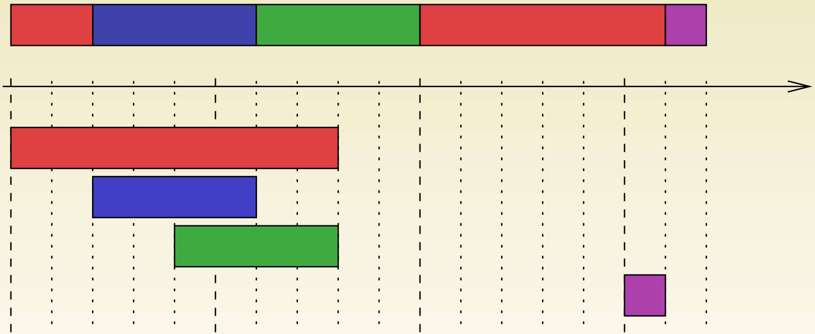
Let's play with a small example

We wish to find a schedule (possibly using preemption) that has the smallest possible max stretch ($\max_i \frac{C_i - r_i}{p_i}$).



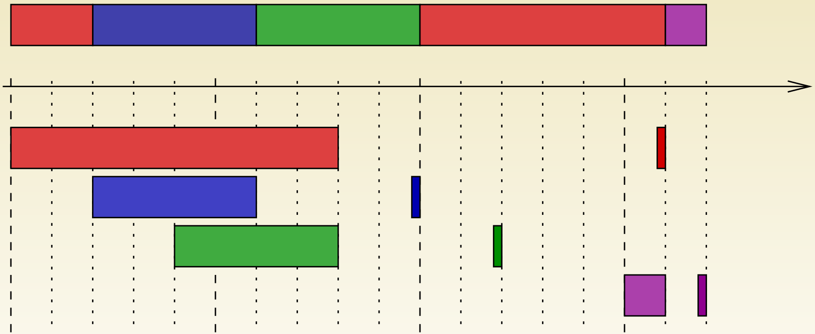
Let's play with a small example

We wish to find a schedule (possibly using preemption) that has the smallest possible max stretch ($\max_i \frac{C_i - r_i}{p_i} = 2$).



Let's play with a small example

We wish to find a schedule (possibly using preemption) that has the smallest possible max stretch ($\max_i \frac{C_i - r_i}{p_i}$).



Ensuring a given max-stretch defines deadlines. . .

Deadline scheduling

- Assume that each job is given a deadline d_i.
- There is a very simple way to know whether it is possible to respect these deadlines:

  always schedule the job with the earliest deadline first (EDF).
- Assume that we want to know whether it is possible to achieve a given max-stretch $\mathcal{S}$. Then we have

$$\frac{C_i - r_i}{p_i} \leqslant \mathcal{S} \Leftrightarrow C_i \leqslant r_i + \mathcal{S}.p_i, \text{ hence } d_i = r_i + \mathcal{S}.p_i.$$

- By doing a dichotomy on $\mathcal{S}$, we have a polynomial (offline) algorithm to minimize the max-stretch.

Going Online

- The previous scheduling algorithm is simple but it requires to have an estimate of the max-stretch, i.e. if you know what you aim at, then you know how to do it.

Going Online

- The previous scheduling algorithm is simple but it requires to have an estimate of the max-stretch, i.e. if you know what you aim at, then you know how to do it.
- Bender, Chahrabarti, and Muthukrishnan (1998).
  Each time a job arrives:
  - Compute the off-line max-stretch $\mathcal{S}$.
  - Jobs are scheduled *earliest deadline first* with the deadlines defined by $\sqrt{\Delta} \times \mathcal{S}$.

  This online algorithm is $\sqrt{\Delta}$-competitive.

- The previous scheduling algorithm is simple but it requires to have an estimate of the max-stretch, i.e. if you know what you aim at, then you know how to do it.
- Bender, Chahrabarti, and Muthukrishnan (1998).
  Each time a job arrives:
  - Compute the off-line max-stretch $\mathcal{S}$.
  - Jobs are scheduled *earliest deadline first* with the deadlines defined by $\sqrt{\Delta} \times \mathcal{S}$.

  This online algorithm is $\sqrt{\Delta}$-competitive.
- Bender, Muthukrishnan, and Rajaraman (2002)
  For each job J_j, we define a pseudo-stretch $\widehat{\mathcal{S}}_j(t)$:

$$\widehat{\mathcal{S}}_j(t) = \begin{cases} \frac{t-r_j}{\sqrt{\Delta}} & \text{if } 1 \leqslant p_j \leqslant \sqrt{\Delta}, \\ \frac{t-r_j}{\Delta} & \text{if } \sqrt{\Delta} < p_j \leqslant \Delta. \end{cases}$$

  The jobs are scheduled by non increasing pseudo-stretch.
  This online algorithm is also $\sqrt{\Delta}$-competitive and is much faster than the previous one.

Bound on the competitive ratio

Actually, it is hard to have a better guarantee of this type.

Theorem 4.

On one processor, any online scheduling algorithm with preemption minimizing the max-stretch has a competitive ratio greater than $\frac{1}{2}\Delta^{\sqrt{2}-1}$, if the system receives at least jobs of three different sizes, and if Δ is the ratio between the size of the largest and the smallest job.

Bound on the competitive ratio

Actually, it is hard to have a better guarantee of this type.

Theorem 4.

On one processor, any online scheduling algorithm with preemption minimizing the max-stretch has a competitive ratio greater than $\frac{1}{2}\Delta^{\sqrt{2}-1}$, if the system receives at least jobs of three different sizes, and if Δ is the ratio between the size of the largest and the smallest job.

Proof principle: by contradiction we assume that there exists an algorithm and we build a sequence of jobs and a scenario to make the algorithm fail. Proof details

Conclusion

Minimizing the average stretch

- Off-line case: looks difficult.
- Online case: rather easy.

Minimizing the max-stretch

- Off-line case: in polynomial time.
- Online: very difficult.

and in practice ?

A non guaranteed heuristic

The Bender98 has two drawbacks:

- It never forgets anything: it computes the optimal offline max stretch from the very beginning $\rightsquigarrow$ slower and slower.
- It focus on optimizing the max-stretch and does not do anything to optimize the second max-stretch, the third max-stretch, and so on.

A non guaranteed heuristic

The Bender98 has two drawbacks:

- It never forgets anything: it computes the optimal offline max stretch from the very beginning $\rightsquigarrow$ slower and slower.
- It focus on optimizing the max-stretch and does not do anything to optimize the second max-stretch, the third max-stretch, and so on.

We have proposed the following heuristic instead.
Each time a job arrives:

1. Preempt the running job (if any).
2. Compute the best achievable max-stretch, $\mathcal{S}$, taking into account the already taken decisions.
3. With the deadlines and time intervals defined by the max-stretch $\mathcal{S}$, solve a Pseudo-approximation of a rational relaxation of sum-stretch (a linear program).

No guarantee !

Simulation results (on one processor)

	Max-stretch			Sum-stretch		
	Mean	SD	Max	Mean	SD	Max
Offline	1.0000	0.0000	1.0000	1.0413	0.0593	1.6735
SWRPT	1.1316	0.2071	3.1643	1.0001	0.0009	1.0398
SRPT	1.1242	0.2003	3.0753	1.0139	0.0212	1.2576
Bender98	1.1200	0.1766	2.5428	1.0194	0.0279	1.4466
Bender03	3.5422	2.4870	21.4819	2.9872	1.9599	15.0019
Heuristic	1.0016	0.0149	1.6344	1.0549	0.0893	1.8134
MCT	8.7762	9.1900	80.7465	6.8979	7.7409	88.2449
RAND	11.3059	11.1981	125.3726	5.8227	6.3942	68.0009

Aggregate statistics for a single machine on various "realistic" workloads.

Conclusion

- The theory claims that optimizing max-stretch is easy in the off-line setting and hard in the online setting.
- The theory claims that optimizing sum-stretch is hard in the off-line setting and rather easy in the online setting.
- The theory claims that optimizing both sum-stretch and max-stretch is impossible.
  More precisely, it is not possible to have worst-case guarantees.

- In practice, optimizing max-stretch online is not that hard and also gives very good results for the average stretch.
- Having good worst-case guarantees does not prevent to perform bad on the average (Bender03).

- Sum-Stretch does not seem a pertinent metric.
- Trying to optimize “recursively” the max-stretch is a good idea and “simple” algorithms can do that.

Outline

Fair Sharing

All previous algorithms (Online, SRPT, SWRPT, . . .) need to know the processing time of the jobs.

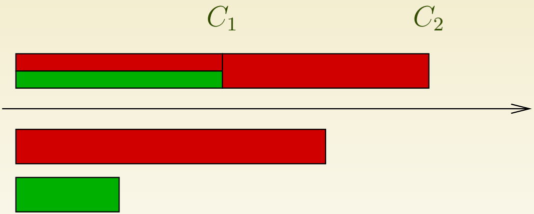
What kind of algorithm could we come up with in a non-clairvoyant setting ?

Fair Sharing

All previous algorithms (Online, SRPT, SWRPT, . . .) need to know the processing time of the jobs.

What kind of algorithm could we come up with in a non-clairvoyant setting ?

Fair Sharing At each time-step, fairly share the resource between the jobs .

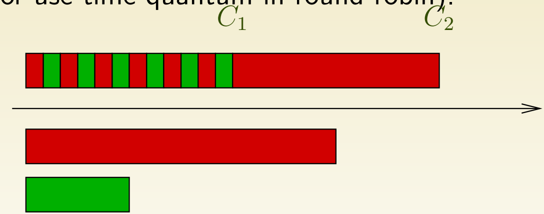


Fair Sharing

All previous algorithms (Online, SRPT, SWRPT, ...) need to know the processing time of the jobs.

What kind of algorithm could we come up with in a non-clairvoyant setting ?

Fair Sharing At each time-step, fairly share the resource between the jobs (or use time quantum in round-robin).

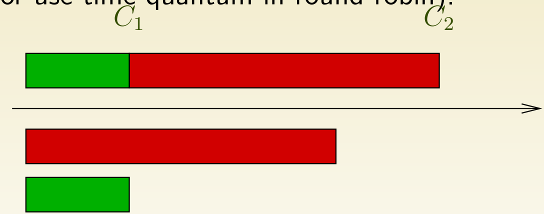


Fair Sharing

All previous algorithms (Online, SRPT, SWRPT, ...) need to know the processing time of the jobs.

What kind of algorithm could we come up with in a non-clairvoyant setting ?

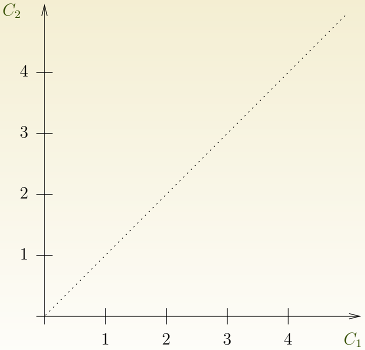
Fair Sharing At each time-step, fairly share the resource between the jobs (or use time quantum in round-robin).

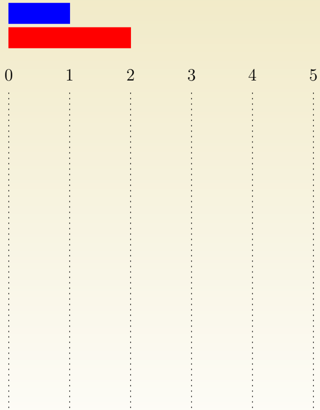


It is not really efficient though...
In the previous algorithms, we have never produced a schedule with $\dots A \dots B \dots A \dots B \dots$. Intuitively alternating jobs is not a good idea.

Pareto optimality

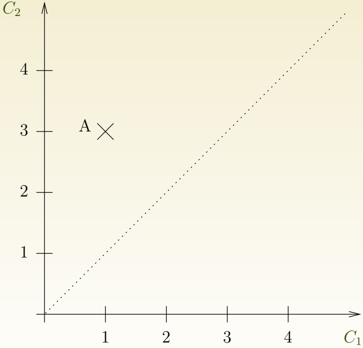
The cost (here the completion time) of a user can not be improved without degrading the cost of another user.

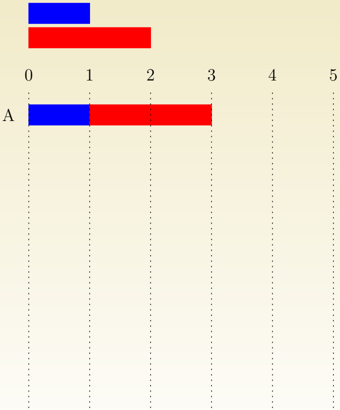




Pareto optimality

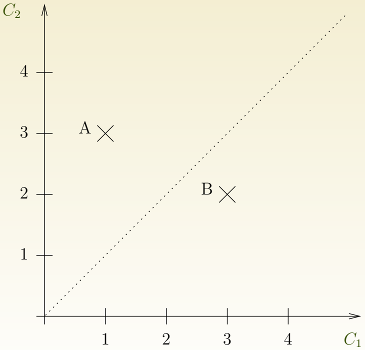
The cost (here the completion time) of a user can not be improved without degrading the cost of another user.

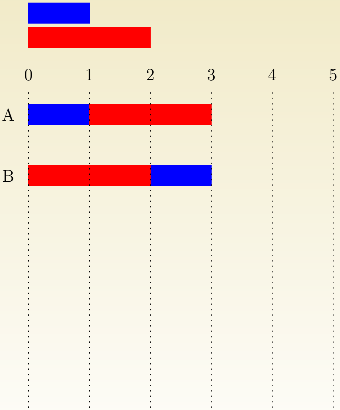




Pareto optimality

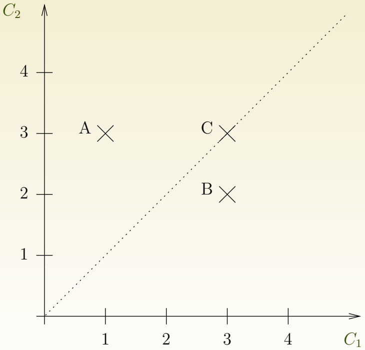
The cost (here the completion time) of a user can not be improved without degrading the cost of another user.

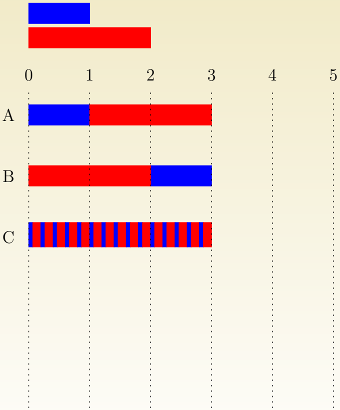




Pareto optimality

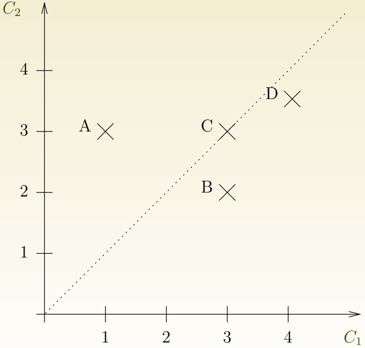
The cost (here the completion time) of a user can not be improved without degrading the cost of another user.

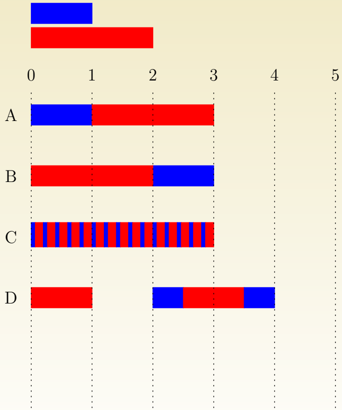




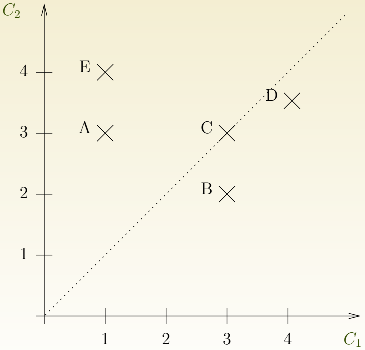
Pareto optimality

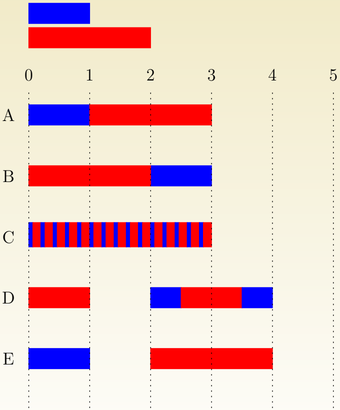
The cost (here the completion time) of a user can not be improved without degrading the cost of another user.





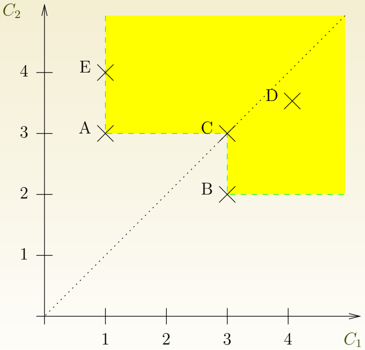
The cost (here the completion time) of a user can not be improved without degrading the cost of another user.

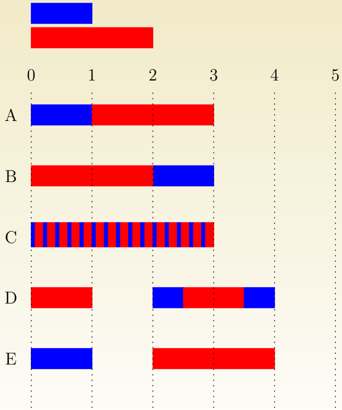




Pareto optimality

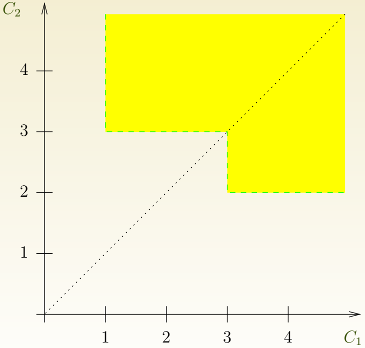
The cost (here the completion time) of a user can not be improved without degrading the cost of another user.



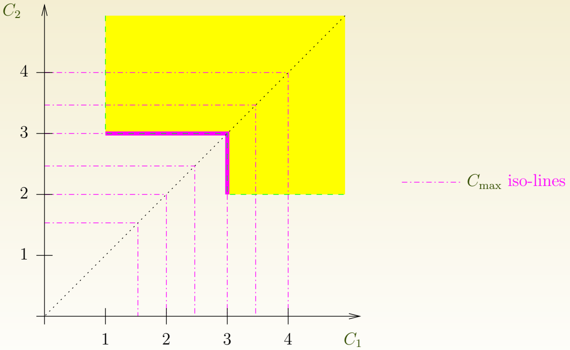


Pareto optimality

The cost (here the completion time) of a user can not be improved without degrading the cost of another user.

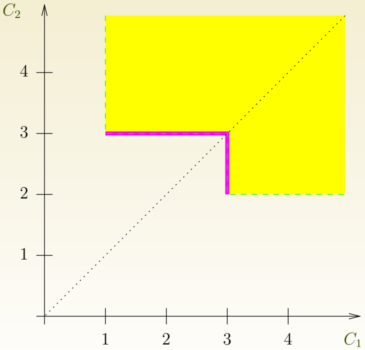


The cost (here the completion time) of a user can not be improved without degrading the cost of another user.

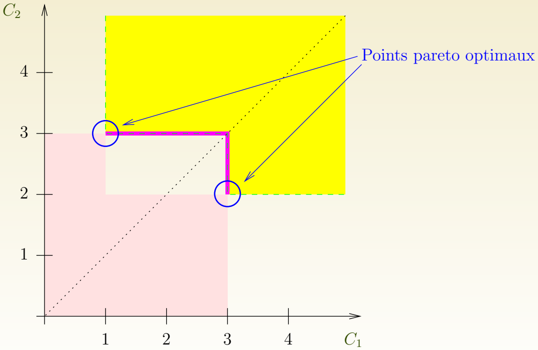


Pareto optimality

The cost (here the completion time) of a user can not be improved without degrading the cost of another user.

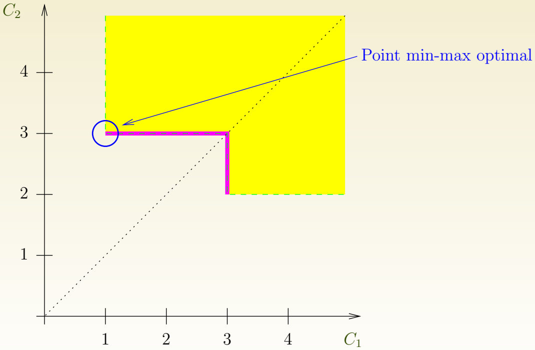


The cost (here the completion time) of a user can not be improved without degrading the cost of another user.

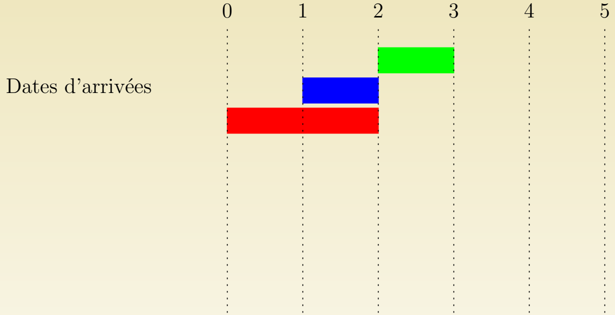


Pareto optimality

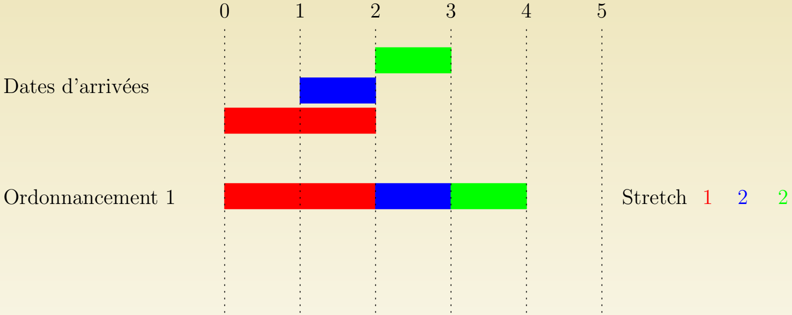
The cost (here the completion time) of a user can not be improved without degrading the cost of another user.



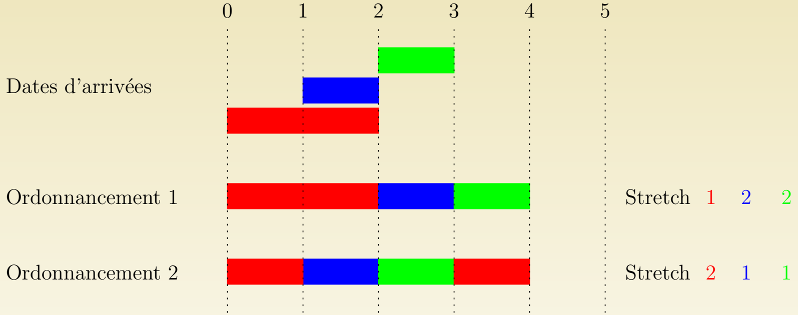
The Pareto max-stretch (1)



The Pareto max-stretch (1)



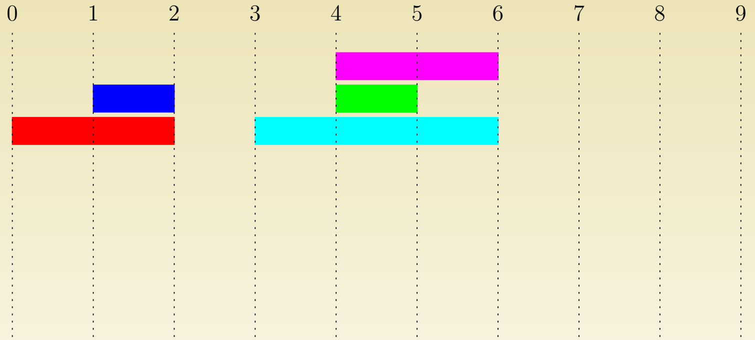
The Pareto max-stretch (1)



Schedule 1 and Schedule 2 are Pareto optimal

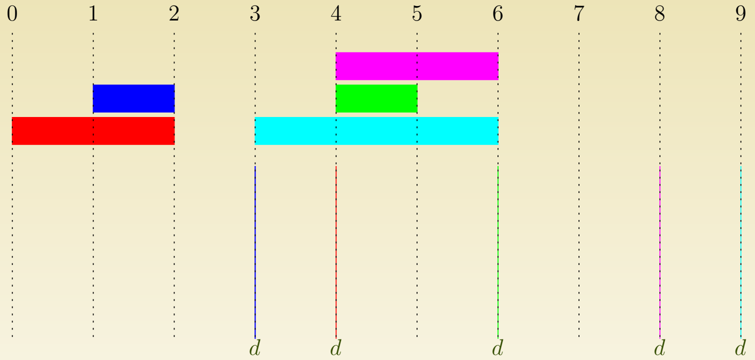
Schedule 2 is the min-max solution

The Pareto max-stretch (2)



Computation of the optimal max-stretch: 2.

The Pareto max-stretch (2)

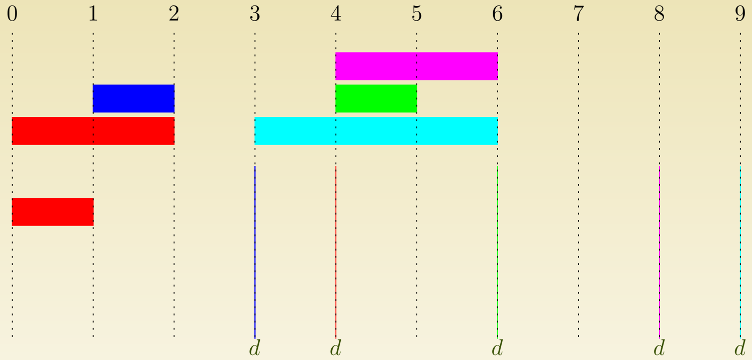


Computation of the optimal max-stretch: 2.

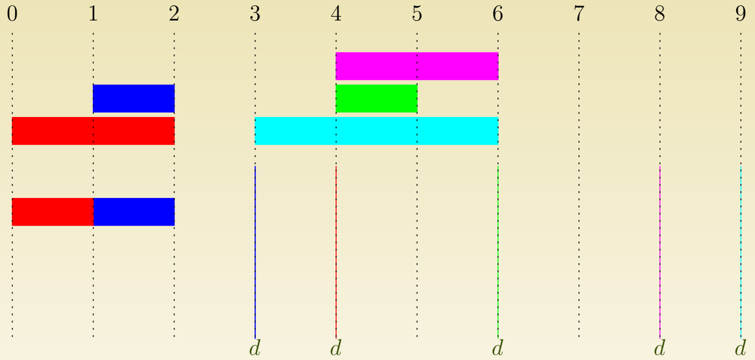
Defining a deadline per job.

Jobs are scheduled *Earliest deadline first*.

The Pareto max-stretch (2)

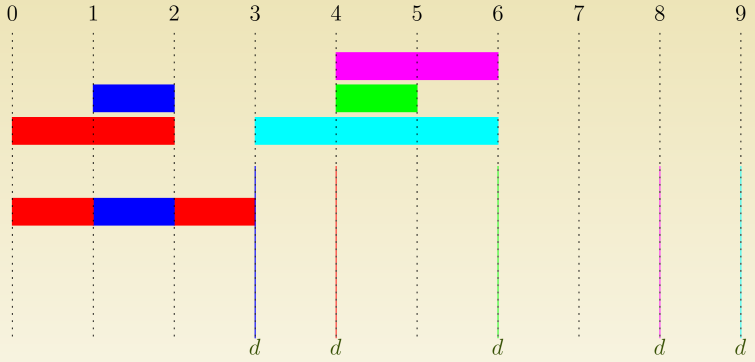


Jobs are scheduled *Earliest deadline first*.



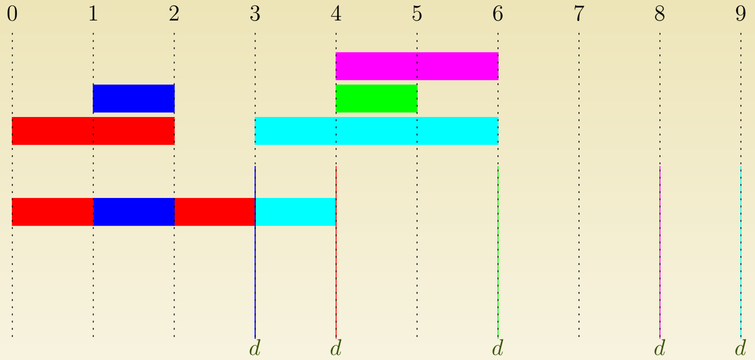
Jobs are scheduled *Earliest deadline first*.

The Pareto max-stretch (2)



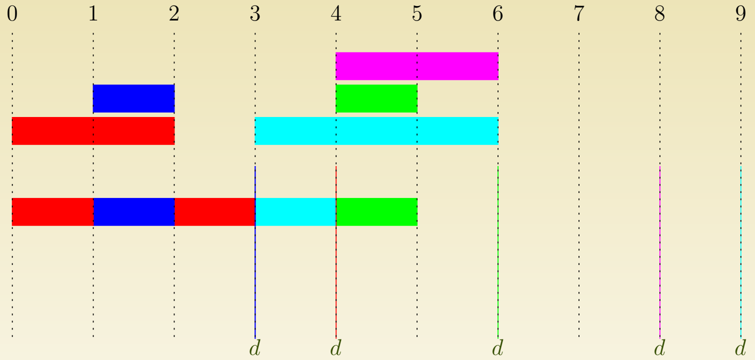
Jobs are scheduled *Earliest deadline first*.

The Pareto max-stretch (2)



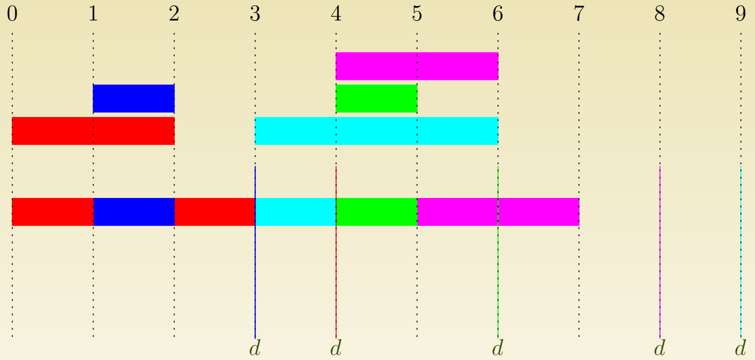
Jobs are scheduled *Earliest deadline first*.

The Pareto max-stretch (2)



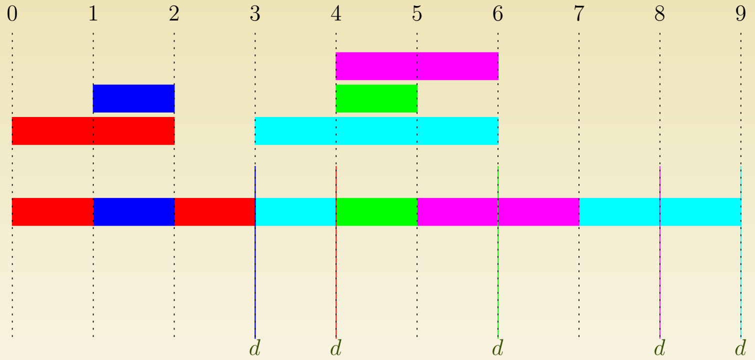
Jobs are scheduled *Earliest deadline first*.

The Pareto max-stretch (2)



Jobs are scheduled *Earliest deadline first*.

The Pareto max-stretch (2)

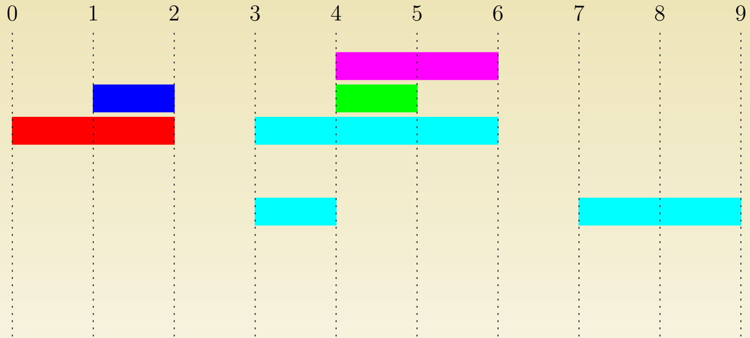


Jobs are scheduled *Earliest deadline first*.

If completion time = deadline, whatever the schedule, the stretch of this job is equal to the maximum stretch.

We set the jobs that cannot be optimized, and we call recursively the process.

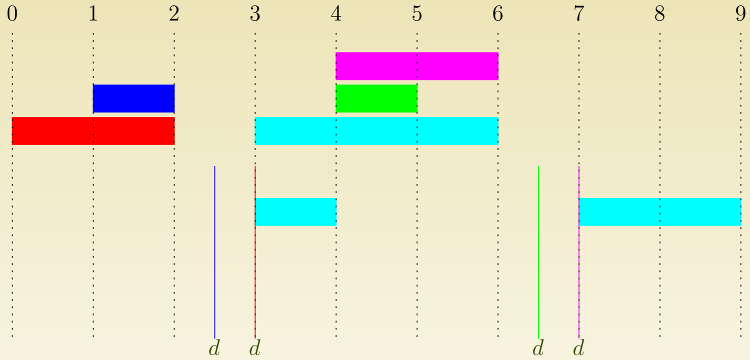
The Pareto max-stretch (2)



We set the jobs that cannot be optimized, and we call recursively the process.

Max-stretch of remaining jobs : 1,5.

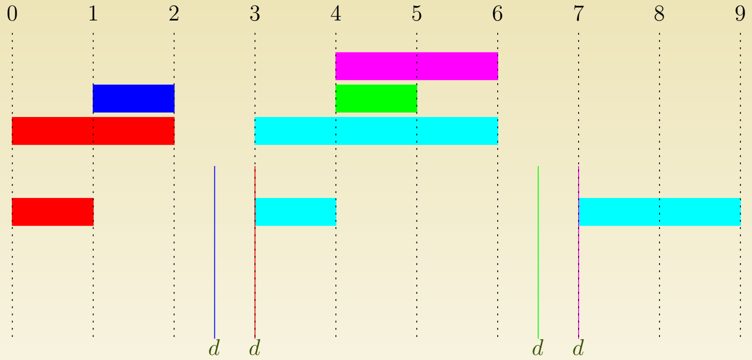
The Pareto max-stretch (2)



We set the jobs that cannot be optimized, and we call recursively the process.

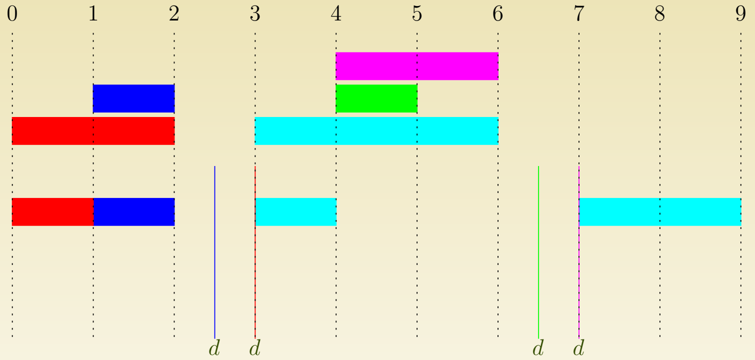
Max-stretch of remaining jobs : 1,5.

The Pareto max-stretch (2)



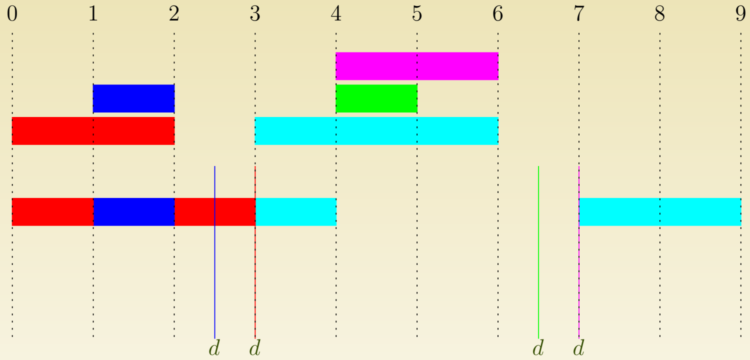
Jobs are scheduled *Earliest deadline first*.

The Pareto max-stretch (2)



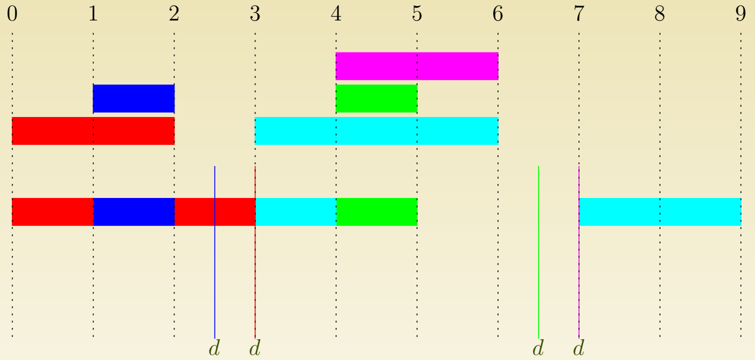
Jobs are scheduled *Earliest deadline first*.

The Pareto max-stretch (2)



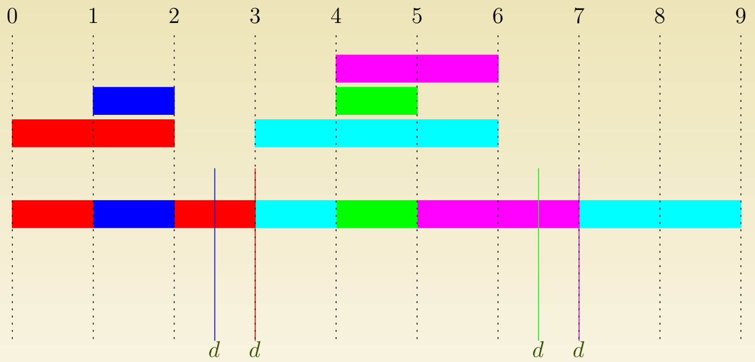
Jobs are scheduled *Earliest deadline first*.

The Pareto max-stretch (2)



Jobs are scheduled *Earliest deadline first*.

The Pareto max-stretch (2)

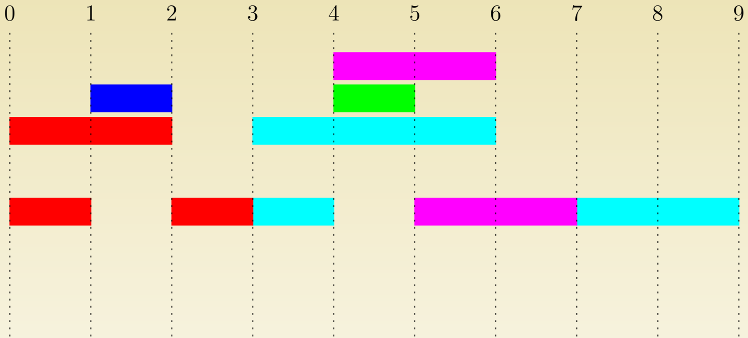


Jobs are scheduled *Earliest deadline first*.

If completion time = deadline, whatever the schedule, the stretch of this job is equal to the maximum stretch.

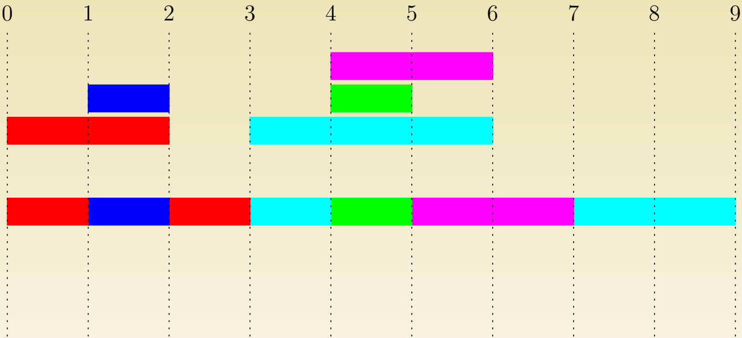
We set the jobs that cannot be optimized, and we call recursively the process.

The Pareto max-stretch (2)



We set the jobs that cannot be optimized, and we call recursively the process.

The Pareto max-stretch (2)

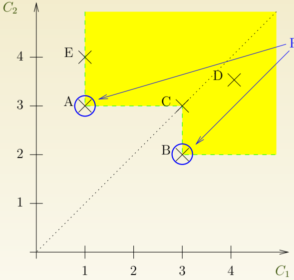


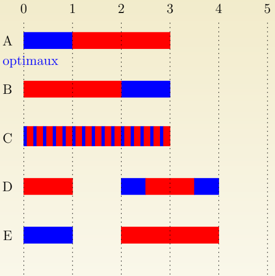
Fair Sharing

The fair sharing approach produces very Pareto-inefficient solutions.

Being fair does not mean "giving the same to everyone".

When looking at $\mathbb{E}[C_i]$ instead of C_i, we get a different cost set.



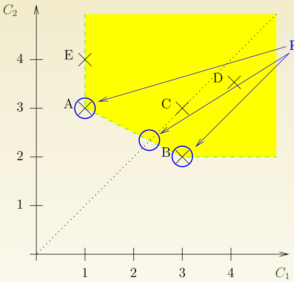


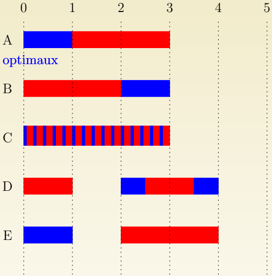
Fair Sharing

The fair sharing approach produces very Pareto-inefficient solutions.

Being fair does not mean "giving the same to everyone".

When looking at $\mathbb{E}[C_i]$ instead of C_i, we get a different cost set.



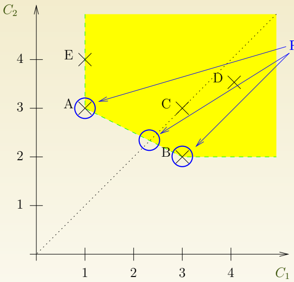


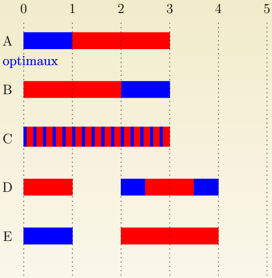
Fair Sharing

The fair sharing approach produces very Pareto-inefficient solutions.

Being fair does not mean "giving the same to everyone".

When looking at $\mathbb{E}[C_i]$ instead of C_i, we get a different cost set.





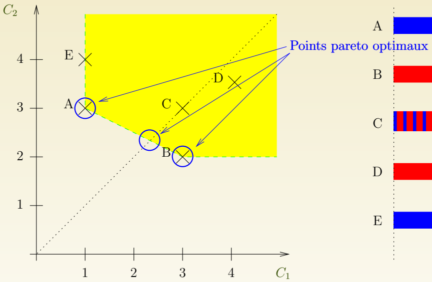
It is better to stick to a choice and to update the balance next time.

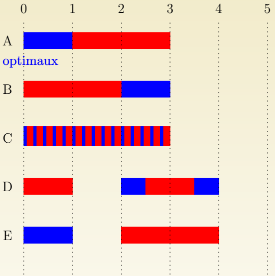
Fair Sharing

The fair sharing approach produces very Pareto-inefficient solutions.

Being fair does not mean "giving the same to everyone".

When looking at $\mathbb{E}[C_i]$ instead of C_i, we get a different cost set.





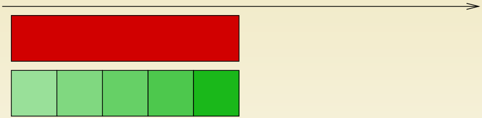
It is better to stick to a choice and to update the balance next time.
Preemption should be used to react to unexpected events, not to try to be "fair".

Fair Sharing

Could you think of a very simple example where Fair Sharing behaves really bad?

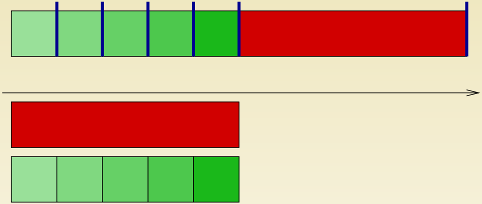
Fair Sharing

Could you think of a very simple example where Fair Sharing behaves really bad?



Fair Sharing

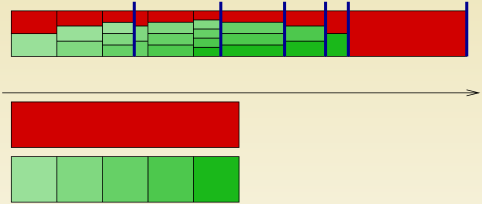
Could you think of a very simple example where Fair Sharing behaves really bad?



$$MS_{opt} = 2 \qquad SS_{opt} = \Delta + 2$$

Fair Sharing

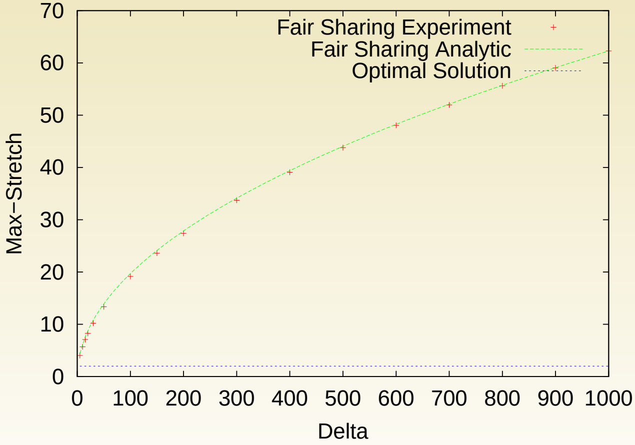
Could you think of a very simple example where Fair Sharing behaves really bad?



$MS_{opt} \approx 1.97\sqrt{\Delta}$ $\qquad$ $SS_{opt} = 1.54\Delta^{1.484}$

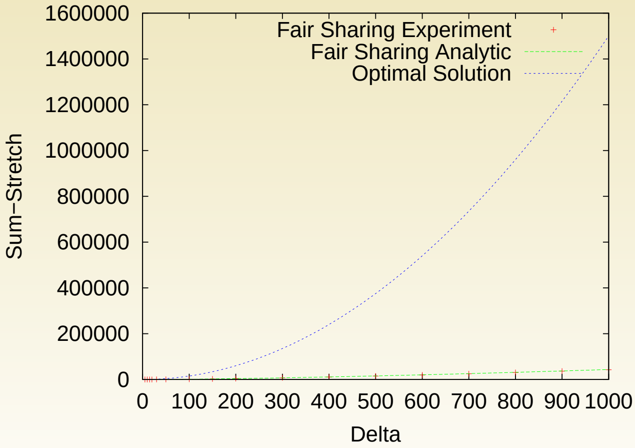
Fair Sharing

Could you think of a very simple example where Fair Sharing behaves really bad?



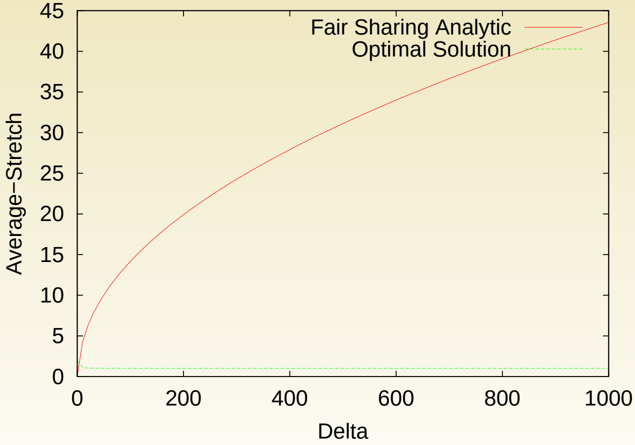
Fair Sharing

Could you think of a very simple example where Fair Sharing behaves really bad?



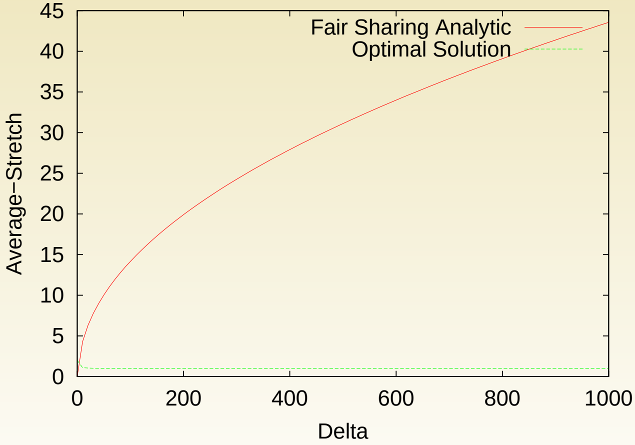
Fair Sharing

Could you think of a very simple example where Fair Sharing behaves really bad?



Fair Sharing

Could you think of a very simple example where Fair Sharing behaves really bad?



Preemption should be used to react to unexpected events, not to try to be “fair”.

Multi-level Queue

The ready queue is partitioned into separate queues:

- foreground queue (interactive or small jobs)
- background queue (batch jobs)

Each queue may have its own scheduling algorithm (e.g., Fair Sharing for the foreground queue and FCFS for the background queue).

Scheduling must be done between the queues:

- Fixed priority scheduling (i.e. foreground first, then background) $\rightsquigarrow$ potential starvation
- Time slice: each queue gets a certain amount of CPU which it can distribute among its processes (e.g., 80% for the foreground queue).

In a non-clairvoyant setting, it may be hard to know in which queue should go a process.

Multi-Level Feedback

Three (or more queues):

- Queue 0: Fair-Sharing with small granularity (e.g., round-robin with time quantum 8 miliseconds).
- Queue 1: Fair-Sharing with larger granularity (e.g., round-robin with time quantum 16 milliseconds)
- Queue 2: FCFS

Multi-Level Feedback

Three (or more queues):

- Queue 0: Fair-Sharing with small granularity (e.g., round-robin with time quantum 8 miliseconds).
- Queue 1: Fair-Sharing with larger granularity (e.g., round-robin with time quantum 16 milliseconds)
- Queue 2: FCFS

Scheduling:

- When a job enters the system, it goes into Queue 0.
- If it does not finish within a few 8ms time quantum, it is moved to Queue 1.
- If it still does not finish within a few 16ms time quantum, it is moved to Queue 2.

Multi-Level Feedback

Three (or more queues):

- Queue 0: Fair-Sharing with small granularity (e.g., round-robin with time quantum 8 miliseconds).
- Queue 1: Fair-Sharing with larger granularity (e.g., round-robin with time quantum 16 milliseconds)
- Queue 2: FCFS

Scheduling:

- When a job enters the system, it goes into Queue 0.
- If it does not finish within a few 8ms time quantum, it is moved to Queue 1.
- If it still does not finish within a few 16ms time quantum, it is moved to Queue 2.

The rationale behind this algorithm is to try to detect small jobs and do some kind of SRPT without knowing the processing time. Larger jobs are served FCFS so there is no starvation.

Multi-Level Feedback

Three (or more queues):

- Queue 0: Fair-Sharing with small granularity (e.g., round-robin with time quantum 8 miliseconds).
- Queue 1: Fair-Sharing with larger granularity (e.g., round-robin with time quantum 16 milliseconds)
- Queue 2: FCFS

Scheduling:

- When a job enters the system, it goes into Queue 0.
- If it does not finish within a few 8ms time quantum, it is moved to Queue 1.
- If it still does not finish within a few 16ms time quantum, it is moved to Queue 2.

The rationale behind this algorithm is to try to detect small jobs and do some kind of SRPT without knowing the processing time. Larger jobs are served FCFS so there is no starvation.

MLF has reactivity, no starvation, and... no guarantee of any kind.

Outline

Conclusion

We have presented and studied many scheduling problems with a "simple" tool: an adversary.

Trying to find bad situations and to trick your algorithms is the best way to understand how to improve them and whether some parameters are important or not.

Theoretical analysis and results help you formalize and understand important scheduling issues:

- What it a relevant objective?
- Can I have a guarantee on how my algorithm behaves in the worst case?
- Is there potential starvation?
- Are common thoughts (like "I'm fair because I give the same to everyone") really true?

FCFS is Δ-competitive for $\sum F_i$

Let's prove FCFS is at least Δ-competitive.

- $\mathcal{F}^{\Theta}(\mathcal{I})$ will denote the sum-flow achieved by the schedule Θ on instance $\mathcal{I}$.

FCFS is Δ-competitive for $\sum F_i$

Let's prove FCFS is at least Δ-competitive.

- $\mathcal{F}^{\Theta}(\mathcal{I})$ will denote the sum-flow achieved by the schedule Θ on instance $\mathcal{I}$.
- $\mathcal{F}^{*}(\mathcal{I})$ will denote the optimal sum-flow for instance $\mathcal{I}$.

Let's prove FCFS is at least Δ-competitive.

- $\mathcal{F}^{\Theta}(\mathcal{I})$ will denote the sum-flow achieved by the schedule Θ on instance $\mathcal{I}$.
- $\mathcal{F}^*(\mathcal{I})$ will denote the optimal sum-flow for instance $\mathcal{I}$.
- We show by recurrence on n that for any instance $\mathcal{I} = \{J_1 = (r_1, p_1), ..., J_n = (r_n, p_n)\}$: $\mathcal{F}^{\mathrm{FCFS}}(\mathcal{I}) \leqslant \Delta \mathcal{F}^*(\mathcal{I})$.

FCFS is Δ-competitive for $\sum F_i$

Let's prove FCFS is at least Δ-competitive.

- $\mathcal{F}^{\Theta}(\mathcal{I})$ will denote the sum-flow achieved by the schedule Θ on instance $\mathcal{I}$.
- $\mathcal{F}^{*}(\mathcal{I})$ will denote the optimal sum-flow for instance $\mathcal{I}$.
- We show by recurrence on n that for any instance $\mathcal{I} = \{J_1 = (r_1, p_1), ..., J_n = (r_n, p_n)\}$: $\mathcal{F}^{\mathrm{FCFS}}(\mathcal{I}) \leqslant \Delta \mathcal{F}^{*}(\mathcal{I})$.
- This property obviously holds for $n = 1$. Let us assume that it has been proved for n and prove that it holds true for n +1.

FCFS is Δ-competitive for $\sum F_i$

- Let us consider $\mathcal{I} = \{J_1 = (r_1, p_1), ..., J_{n+1} = (r_{n+1}, p_{n+1})\}$ an instance with $n+1$ jobs (and w.l.o.g $\min_j p_j = 1$).

FCFS is Δ-competitive for $\sum F_i$

- Let us consider $\mathcal{I} = \{J_1 = (r_1, p_1), ..., J_{n+1} = (r_{n+1}, p_{n+1})\}$ an instance with $n+1$ jobs (and w.l.o.g $\min_j p_j = 1$).
- We may only consider priority list based schedules. Thus let Θ denote the optimal priority list for $\mathcal{I}$.

FCFS is Δ-competitive for $\sum F_i$

- Let us consider $\mathcal{I} = \{J_1 = (r_1, p_1), ..., J_{n+1} = (r_{n+1}, p_{n+1})\}$ an instance with $n+1$ jobs (and w.l.o.g $\min_j p_j = 1$).
- We may only consider priority list based schedules. Thus let Θ denote the optimal priority list for $\mathcal{I}$.
- We denote by A_1 the set of jobs that have a lower priority than J_{n+1} and A_2 the set of jobs that have a higher priority than J_{n+1}.

FCFS is Δ-competitive for $\sum F_i$

- Let us consider $\mathcal{I} = \{J_1 = (r_1, p_1), ..., J_{n+1} = (r_{n+1}, p_{n+1})\}$ an instance with $n+1$ jobs (and w.l.o.g $\min_j p_j = 1$).
- We may only consider priority list based schedules. Thus let Θ denote the optimal priority list for $\mathcal{I}$.
- We denote by A_1 the set of jobs that have a lower priority than J_{n+1} and A_2 the set of jobs that have a higher priority than J_{n+1}.
- $\varrho^{\Theta}(J_k)$ denotes the remaining processing time of J_k at time r_{n+1} under scheduling Θ.

FCFS is Δ-competitive for $\sum F_i$

- Let us consider $\mathcal{I} = \{J_1 = (r_1, p_1), ..., J_{n+1} = (r_{n+1}, p_{n+1})\}$ an instance with $n+1$ jobs (and w.l.o.g $\min_j p_j = 1$).
- We may only consider priority list based schedules. Thus let Θ denote the optimal priority list for $\mathcal{I}$.
- We denote by A_1 the set of jobs that have a lower priority than J_{n+1} and A_2 the set of jobs that have a higher priority than J_{n+1}.
- $\varrho^{\Theta}(J_k)$ denotes the remaining processing time of J_k at time r_{n+1} under scheduling Θ.
- Thus we have:

$$\mathcal{F}^{\Theta}(J_1, \ldots, J_{n+1}) = \mathcal{F}^{\Theta}(J_1, \ldots, J_n) + \underbrace{p_{n+1} + \sum_{k \in A_1} \varrho^{\Theta}(k)}_{\text{The flow of } J_{n+1}} + \underbrace{\sum_{k \in A_2} p_{n+1}}_{\text{The cost incurred by } J_{n+1}}$$

We also have:

$$\mathcal{F}^{\text{FCFS}}(J_1,\dots,J_{n+1}) = \mathcal{F}^{\text{FCFS}}(J_1,\dots,J_n) + \underbrace{p_{n+1} + \sum_{k\leqslant n} \varrho^{\text{FCFS}}(k)}_{\text{The flow of } J_{n+1}}$$

$$\leqslant \Delta\mathcal{F}^*(J_1,\dots,J_n) + p_{n+1} + \sum_{k\leqslant n} \varrho^{\text{FCFS}}(k) \quad \text{(by recurrence}$$

$$\leqslant \Delta\mathcal{F}^{\Theta}(J_1,\dots,J_n) + p_{n+1} + \sum_{k\leqslant n} \varrho^{\text{FCFS}}(k)$$

$$= \Delta\mathcal{F}^{\Theta}(J_1,\dots,J_n) + p_{n+1} + \sum_{k\leqslant n} \varrho^{\Theta}(k)$$

Indeed, for a priority-based scheduling, at any given time step, the remaining processing time of jobs is independent of the priorities.

Therefore, we have:

$$\mathcal{F}^{\mathrm{FCFS}}(J_1,\dots,J_{n+1}) \leqslant \Delta\mathcal{F}^{\Theta}(J_1,\dots,J_n) + p_{n+1} + \sum_{k\in A_1} \varrho^{\Theta}(k) + \sum_{k\in A_2} \varrho^{\Theta}(k)$$

As we have $\varrho^{\Theta}(k) \leqslant \Delta \leqslant \Delta p_{n+1}$, we get

$$\begin{aligned}\mathcal{F}^{\mathrm{FCFS}}(J_1,\dots,J_{n+1}) &\leqslant \Delta\mathcal{F}^{\Theta}(J_1,\dots,J_n) + p_{n+1} + \sum_{k\in A_1} \varrho^{\Theta}(k) + \sum_{k\in A_2} \Delta p_{n+1}\\ &\leqslant \Delta\mathcal{F}^{\Theta}(J_1,\dots,J_n) + \Delta p_{n+1} + \Delta\sum_{k\in A_1} \varrho^{\Theta}(k) +\\ &\quad \Delta \sum_{k\in A_2} p_{n+1}\\ &\leqslant \Delta\mathcal{F}^{\Theta}(J_1,\dots,J_{n+1}) = \Delta\mathcal{F}^{*}(J_1,\dots,J_{n+1}) \qquad \square\end{aligned}$$

Max-Stretch and Sum-Stretch are incompatible : demonstration (1)

- By contradiction, $\exists \Delta > 1$, $\exists \varepsilon > 0$, $\exists$ algorithm $\mathcal{A}$ s.t.

$$\varrho_{\mathcal{A}}(\Delta) < \Delta^2 - \varepsilon.$$

Max-Stretch and Sum-Stretch are incompatible : demonstration (1)

- By contradiction, $\exists \Delta > 1$, $\exists \varepsilon > 0$, $\exists$ algorithm $\mathcal{A}$ s.t.

$$\varrho_{\mathcal{A}}(\Delta) < \Delta^2 - \varepsilon.$$

- Adversary:

Max-Stretch and Sum-Stretch are incompatible : demonstration (1)

- By contradiction, $\exists \Delta > 1$, $\exists \varepsilon > 0$, $\exists$ algorithm $\mathcal{A}$ s.t.

$$\varrho_{\mathcal{A}}(\Delta) < \Delta^2 - \varepsilon.$$

- Adversary:
  - Let $\alpha \in \mathbb{N}^*$ s.t. $\frac{1+\alpha\Delta}{1+\frac{\alpha}{\Delta}} > \Delta^2 - \frac{\varepsilon}{2}$

Max-Stretch and Sum-Stretch are incompatible : demonstration (1)

- By contradiction, $\exists \Delta > 1$, $\exists \varepsilon > 0$, $\exists$ algorithm $\mathcal{A}$ s.t.

$$\varrho_{\mathcal{A}}(\Delta) < \Delta^2 - \varepsilon.$$

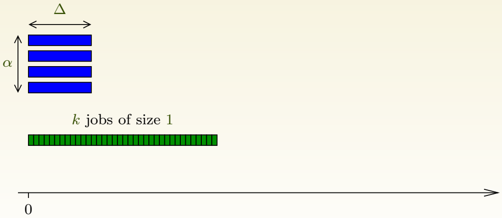
- Adversary:
  - Let $\alpha \in \mathbb{N}^*$ s.t. $\frac{1+\alpha\Delta}{1+\frac{\alpha}{\Delta}} > \Delta^2 - \frac{\varepsilon}{2}$
  - At date 0 arrives α jobs of size Δ, J_1, ..., J_α.

Max-Stretch and Sum-Stretch are incompatible : demonstration (1)

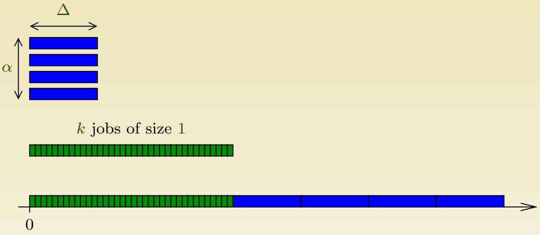
- By contradiction, $\exists \Delta > 1$, $\exists \varepsilon > 0$, $\exists$ algorithm $\mathcal{A}$ s.t.

$$\varrho_{\mathcal{A}}(\Delta) < \Delta^2 - \varepsilon.$$

- Adversary:
  - Let $\alpha \in \mathbb{N}^*$ s.t. $\frac{1+\alpha\Delta}{1+\frac{\alpha}{\Delta}} > \Delta^2 - \frac{\varepsilon}{2}$
  - At date 0 arrives α jobs of size Δ, J_1, ..., J_α.
  - $k \in \mathbb{N}^*$. $\forall t \in [0, k-1]$, job $J_{\alpha+t+1}$ of size 1 arrives at time t.



Max-Stretch and Sum-Stretch are incompatible : demonstration (2)



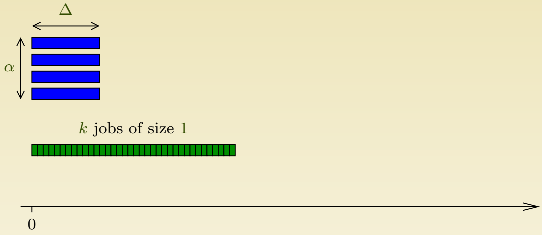
A possible schedule: each of the k unit jobs at its release date, and then the α Δ-units jobs.

$$\text{sum-stretch} = k \times 1 + \frac{k+\Delta}{\Delta} + \ldots + \frac{k+\alpha\Delta}{\Delta} = \frac{\alpha(\alpha+1)}{2} + k\left(1+\frac{\alpha}{\Delta}\right).$$

$$\text{max-stretch} = \alpha + \frac{k}{\Delta}.$$

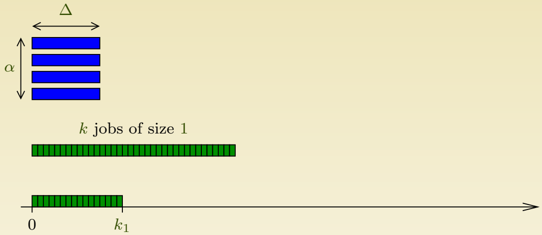
May not be optimal (just an upper-bound) and induces starvation.

Max-Stretch and Sum-Stretch are incompatible : demonstration (2)



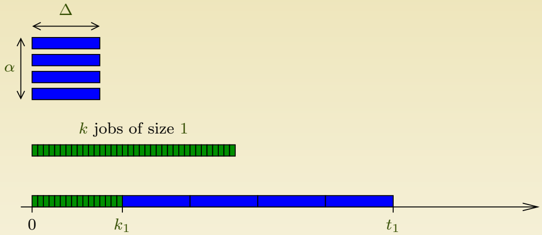
Otherwise, at a date $t_1 < k + \alpha\Delta$ the Δ size jobs are all completed.

Max-Stretch and Sum-Stretch are incompatible : demonstration (2)



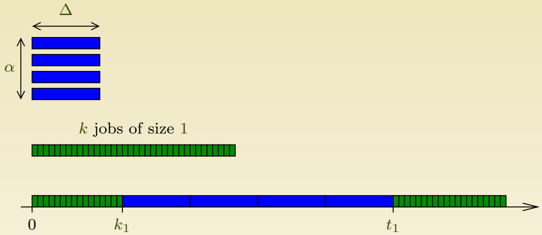
Otherwise, at a date $t_1 < k + \alpha\Delta$ the Δ size jobs are all completed.
k_1 unit size jobs were completed before t_1.

Max-Stretch and Sum-Stretch are incompatible : demonstration (2)



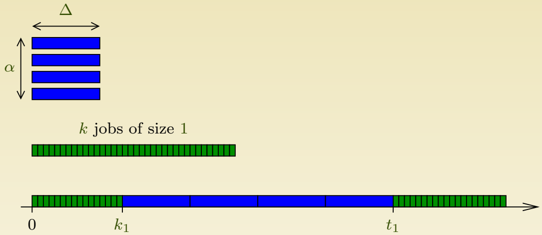
Otherwise, at a date $t_1 < k + \alpha\Delta$ the Δ size jobs are all completed.
k_1 unit size jobs were completed before t_1.

Max-Stretch and Sum-Stretch are incompatible : demonstration (2)



Otherwise, at a date $t_1 < k + \alpha\Delta$ the Δ size jobs are all completed.
k_1 unit size jobs were completed before t_1.

Max-Stretch and Sum-Stretch are incompatible : demonstration (2)



Otherwise, at a date $t_1 < k + \alpha\Delta$ the Δ size jobs are all completed.
k_1 unit size jobs were completed before t_1.
Best achievable sum-stretch:

$$k_1 \times 1 + \frac{k_1 + \Delta}{\Delta} + \ldots + \frac{k_1 + \alpha\Delta}{\Delta} + (k - k_1)(1 + \alpha\Delta) =$$
$$\left(\frac{\alpha(\alpha+1)}{2} + \frac{\alpha k_1}{\Delta}\right) + k_1 + (k - k_1)(1 + \alpha\Delta).$$

Max-Stretch and Sum-Stretch are incompatible : demonstration (3)

Hypothesis: $\mathcal{A}$ is $\varrho_{\mathcal{A}}(\Delta)$-competitive

$$\left(\frac{\alpha(\alpha+1)}{2}+\frac{\alpha k_1}{\Delta}\right)+k_1+(k-k_1)(1+\alpha\Delta) \leqslant \varrho_{\mathcal{A}}(\Delta)\left(\frac{\alpha(\alpha+1)}{2}+k\left(1+\frac{\alpha}{\Delta}\right)\right) \quad \Leftrightarrow$$

$$-\alpha\Delta k_1+\frac{\alpha(\alpha+1)}{2}(1-\varrho_{\mathcal{A}}(\Delta))+\frac{\alpha k_1}{\Delta} \leqslant k\left(\varrho_{\mathcal{A}}(\Delta)\left(1+\frac{\alpha}{\Delta}\right)-(1+\alpha\Delta)\right).$$

Must hold for any k, thus:

$$\left(\varrho_{\mathcal{A}}(\Delta)\left(1+\frac{\alpha}{\Delta}\right)-(1+\alpha\Delta)\right)\geqslant 0 \Rightarrow \Delta^2-\varepsilon>\frac{1+\alpha\Delta}{1+\frac{\alpha}{\Delta}}.$$

FCFS competitiveness: at worst Δ competitive

- An instance J_1, ..., J_n.
  Θ^*: an optimal schedule for max-stretch.
  C_j: completion time of J_j under FCFS (C_j^* under Θ^*).
  $\mathcal{S}_j$: stretch of J_j under FCFS ($\mathcal{S}_j^*$ under Θ^*).

FCFS competitiveness: at worst Δ competitive

- An instance J_1, ..., J_n.
  Θ^*: an optimal schedule for max-stretch.
  C_j: completion time of J_j under FCFS (C_j^* under Θ^*).
  $\mathcal{S}_j$: stretch of J_j under FCFS ($\mathcal{S}_j^*$ under Θ^*).
- Any job J_l s.t. $\mathcal{S}_l > \mathcal{S}_l^*$.

FCFS competitiveness: at worst Δ competitive

- An instance J_1, ..., J_n.
  Θ^*: an optimal schedule for max-stretch.
  C_j: completion time of J_j under FCFS (C_j^* under Θ^*).
  $\mathcal{S}_j$: stretch of J_j under FCFS ($\mathcal{S}_j^*$ under Θ^*).
- Any job J_l s.t. $\mathcal{S}_l > \mathcal{S}_l^*$.
  t last time before C_l s.t. the processor was idle under FCFS.
  t is the release date r_i of some job J_i.

FCFS competitiveness: at worst Δ competitive

- An instance J_1, ..., J_n.
  Θ^*: an optimal schedule for max-stretch.
  C_j: completion time of J_j under FCFS (C_j^* under Θ^*).
  $\mathcal{S}_j$: stretch of J_j under FCFS ($\mathcal{S}_j^*$ under Θ^*).
- Any job J_l s.t. $\mathcal{S}_l > \mathcal{S}_l^*$.
  t last time before C_l s.t. the processor was idle under FCFS.
  t is the release date r_i of some job J_i.
  During the time interval $[r_i, C_l]$, FCFS exactly executes J_i, J_{i+1}, ..., J_{l-1}, J_l.

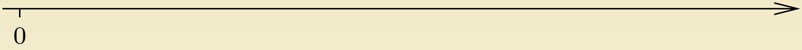
FCFS competitiveness: at worst Δ competitive

- An instance J_1, ..., J_n.
  Θ^*: an optimal schedule for max-stretch.
  C_j: completion time of J_j under FCFS (C_j^* under Θ^*).
  $\mathcal{S}_j$: stretch of J_j under FCFS ($\mathcal{S}_j^*$ under Θ^*).
- Any job J_l s.t. $\mathcal{S}_l > \mathcal{S}_l^*$.
  t last time before C_l s.t. the processor was idle under FCFS.
  t is the release date r_i of some job J_i.
  During the time interval $[r_i, C_l]$, FCFS exactly executes J_i, J_{i+1}, ..., J_{l-1}, J_l.
  As $C_l^* < C_l$, there is a job J_k, $i \leqslant k \leqslant l-1$ s.t. $C_k^* \geqslant C_l$.
  Then:

$$\max_j \mathcal{S}_j^* \geqslant \mathcal{S}_k^* = \frac{C_k^* - r_k}{p_k} \geqslant \frac{C_l - r_l}{p_k} = \frac{C_l - r_l}{p_l}\frac{p_l}{p_k} \geqslant \mathcal{S}_l \times \frac{1}{\Delta}$$

$$\forall l, \mathcal{S}_l > \mathcal{S}_l^* \quad \Rightarrow \quad \mathcal{S}^* \geqslant \mathcal{S}_l \times \frac{1}{\Delta}.$$

The adversary

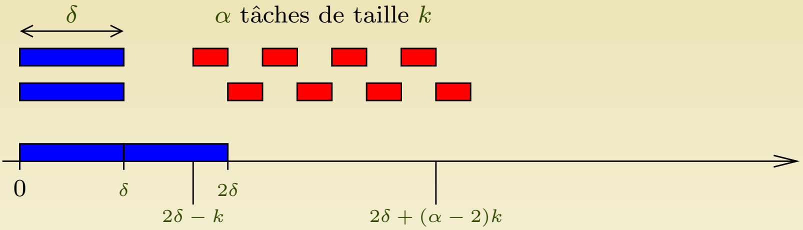


The adversary

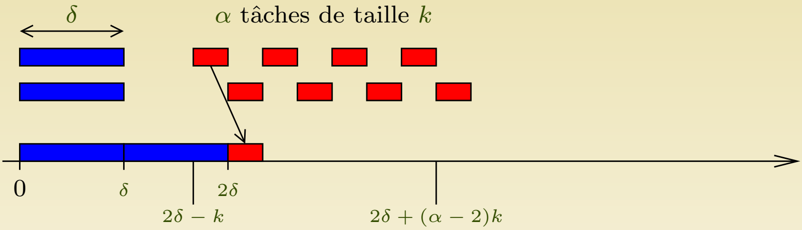
The adversary

Achievable stretch: $\dfrac{2\delta - 0}{\delta} = 2$.

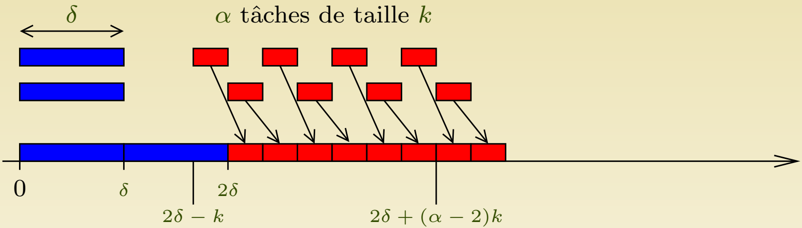
The adversary



The job T_{2+j} arrives at time $2\delta + (j-2)k$.



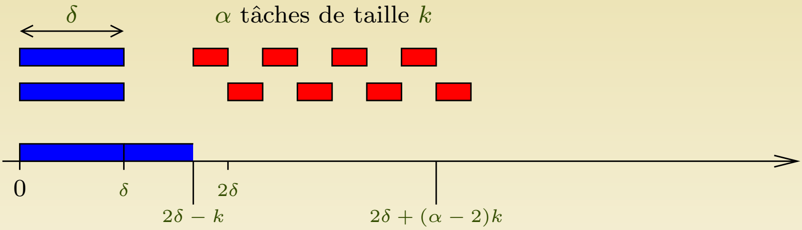
The job T_{2+j} arrives at time $2\delta + (j-2)k$.



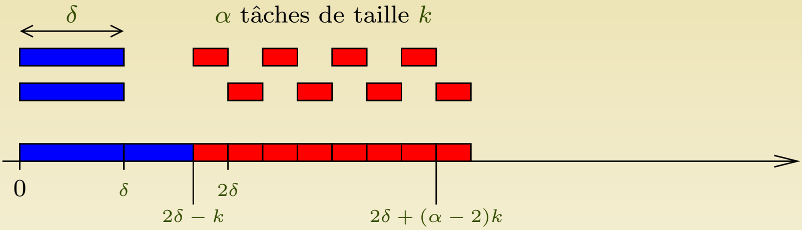
The job T_{2+j} arrives at time $2\delta + (j-2)k$.

Achievable stretch: $$\frac{(2\delta + jk) - (2\delta + (j-2)k)}{k} = 2.$$

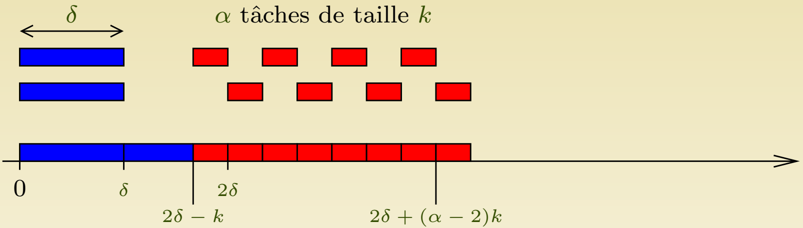
The adversary



In practice: we do not know what happens after $2\delta - k$.



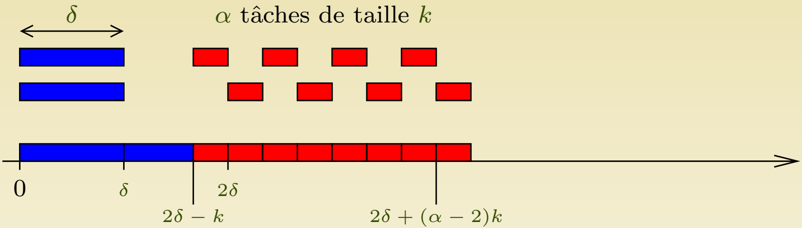
We want to forbid this case (each size-k job being executed at its release date.



We want to forbid this case (each size-k job being executed at its release date.

The algorithm being $\frac{1}{2}\Delta^{\sqrt{2}-1}$-competitive, T_1 and T_2 must be completed at the latest at time: $2 \cdot \frac{1}{2}\Delta^{\sqrt{2}-1} \cdot \delta = 2 \cdot \frac{1}{2}\left(\frac{\delta}{k}\right)^{\sqrt{2}-1} \cdot \delta$

The adversary

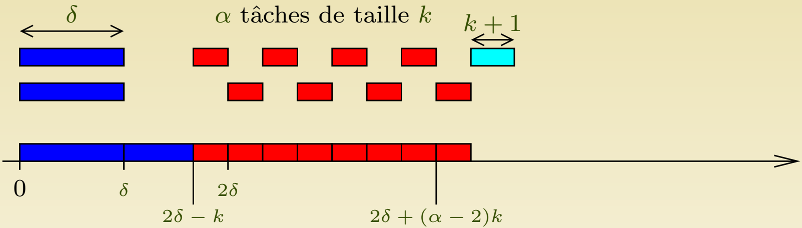


We want to forbid this case (each size-k job being executed at its release date.

The algorithm being $\frac{1}{2}\Delta^{\sqrt{2}-1}$-competitive, T_1 and T_2 must be completed at the latest at time: $2 \cdot \frac{1}{2}\Delta^{\sqrt{2}-1} \cdot \delta = 2 \cdot \frac{1}{2}\left(\frac{\delta}{k}\right)^{\sqrt{2}-1} \cdot \delta$

We let $\alpha = \lceil 1 + k - \frac{2\delta}{k} \rceil$ and then $2\delta + (\alpha - 1)k \geqslant 2 \cdot \frac{1}{2}\left(\frac{\delta}{k}\right)^{\sqrt{2}-1} \cdot \delta$.

The adversary

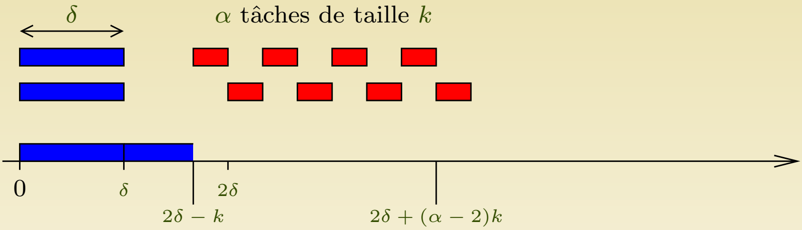


We want to forbid this case (each size-k job being executed at its release date.

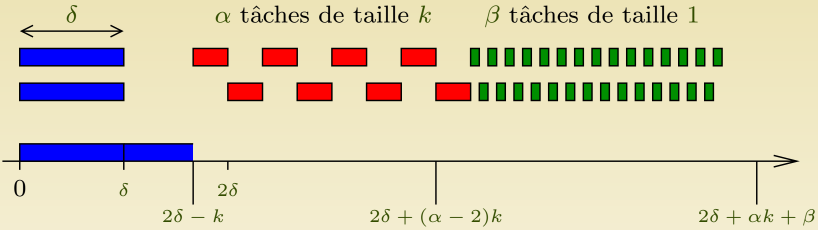
The algorithm being $\frac{1}{2}\Delta^{\sqrt{2}-1}$-competitive, T_1 and T_2 must be completed at the latest at time: $2 \cdot \frac{1}{2}\Delta^{\sqrt{2}-1} \cdot \delta = 2 \cdot \frac{1}{2}\left(\frac{\delta}{k}\right)^{\sqrt{2}-1} \cdot \delta$

We let $\alpha = \lceil 1 + k - \frac{2\delta}{k} \rceil$ and then $2\delta + (\alpha - 1)k \geqslant 2 \cdot \frac{1}{2}\left(\frac{\delta}{k}\right)^{\sqrt{2}-1} \cdot \delta$.

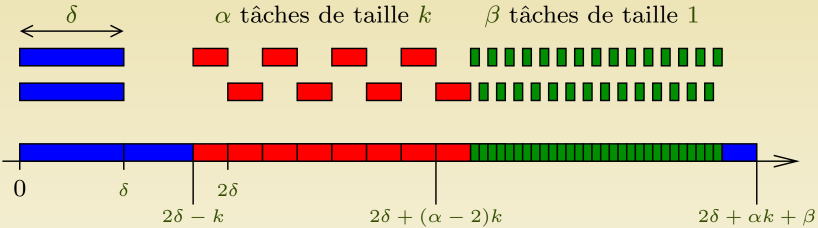
The adversary



The adversary



The job $T_{2+\alpha+j}$ arrives at time $2\delta + (\alpha - 1)k + (j - 1)$.

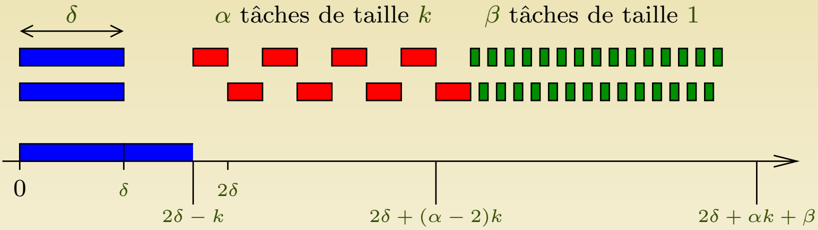


Achievable stretch (off-line)

Stretch of each job of size k or 1 : 1.

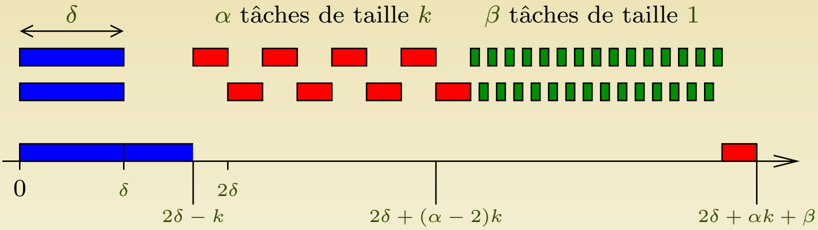
Stretch of T_1 or T_2: $\dfrac{2\delta + \alpha k + \beta}{\delta}$

Optimal stretch $\leqslant \dfrac{2\delta + \alpha k + \beta}{\delta}$



Achievable stretch (online)

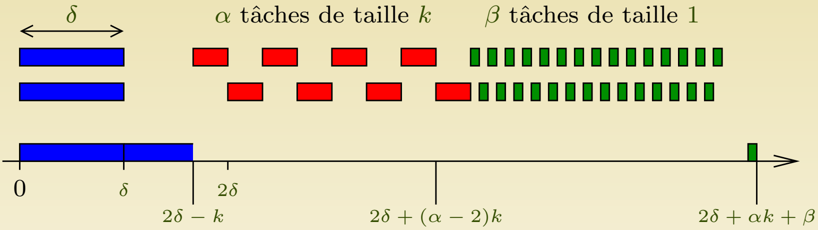
The adversary



Achievable stretch (online)

The last completed job is of size k.

$$\text{Stretch} \geqslant \frac{(2\delta + \alpha k + \beta) - (2\delta + (\alpha - 2)k)}{k} = 2 + \frac{\beta}{k}.$$

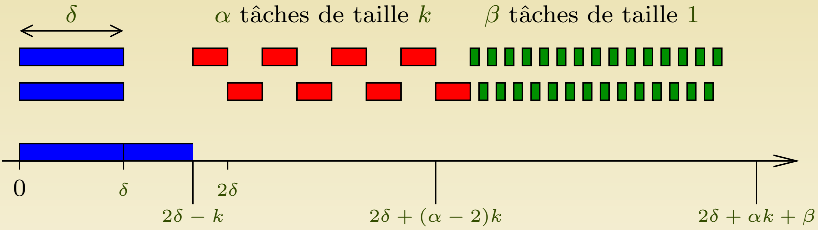


Achievable stretch (online)

The last completed job is of size 1.

$$\text{Stretch} \geqslant \frac{(2\delta + \alpha k + \beta) - (2\delta + (\alpha - 1)k + (\beta - 1))}{1} = k + 1.$$

The adversary



Achievable stretch (online)

Stretch $\geqslant \min\left\{2 + \frac{\beta}{k}, k+1\right\}$

We let: $\beta = \lceil k(k-1) \rceil$

Then: stretch $\geqslant k + 1$.

The adversary: summing things up

$$\alpha = \left\lceil 1 + k - \frac{2\delta}{k} \right\rceil$$

$$\beta = \lceil k(k-1) \rceil$$

Optimal stretch $\leqslant \dfrac{2\delta + \alpha k + \beta}{\delta}$

Achieved stretch $\geqslant k + 1$.

The adversary: summing things up

$$\alpha = \left\lceil 1 + k - \frac{2\delta}{k} \right\rceil$$

$$\beta = \lceil k(k-1) \rceil$$

Optimal stretch $\leqslant \dfrac{2\delta + \alpha k + \beta}{\delta}$

Achieved stretch $\geqslant k + 1$.

We let $k = \delta^{2-\sqrt{2}}$

The adversary: summing things up

$$\alpha = \left\lceil 1 + k - \frac{2\delta}{k} \right\rceil$$

$$\beta = \lceil k(k-1) \rceil$$

Optimal stretch $\leqslant \dfrac{2\delta + \alpha k + \beta}{\delta}$

Achieved stretch $\geqslant k + 1$.

We let $k = \delta^{2-\sqrt{2}}$

Therefore $k + 1 > \left(\dfrac{1}{2}\delta^{\sqrt{2}-1}\right)\left(\dfrac{2\delta + \alpha k + \beta}{\delta}\right)$