INFO-F404, Real-Time Multiprocessor Scheduling

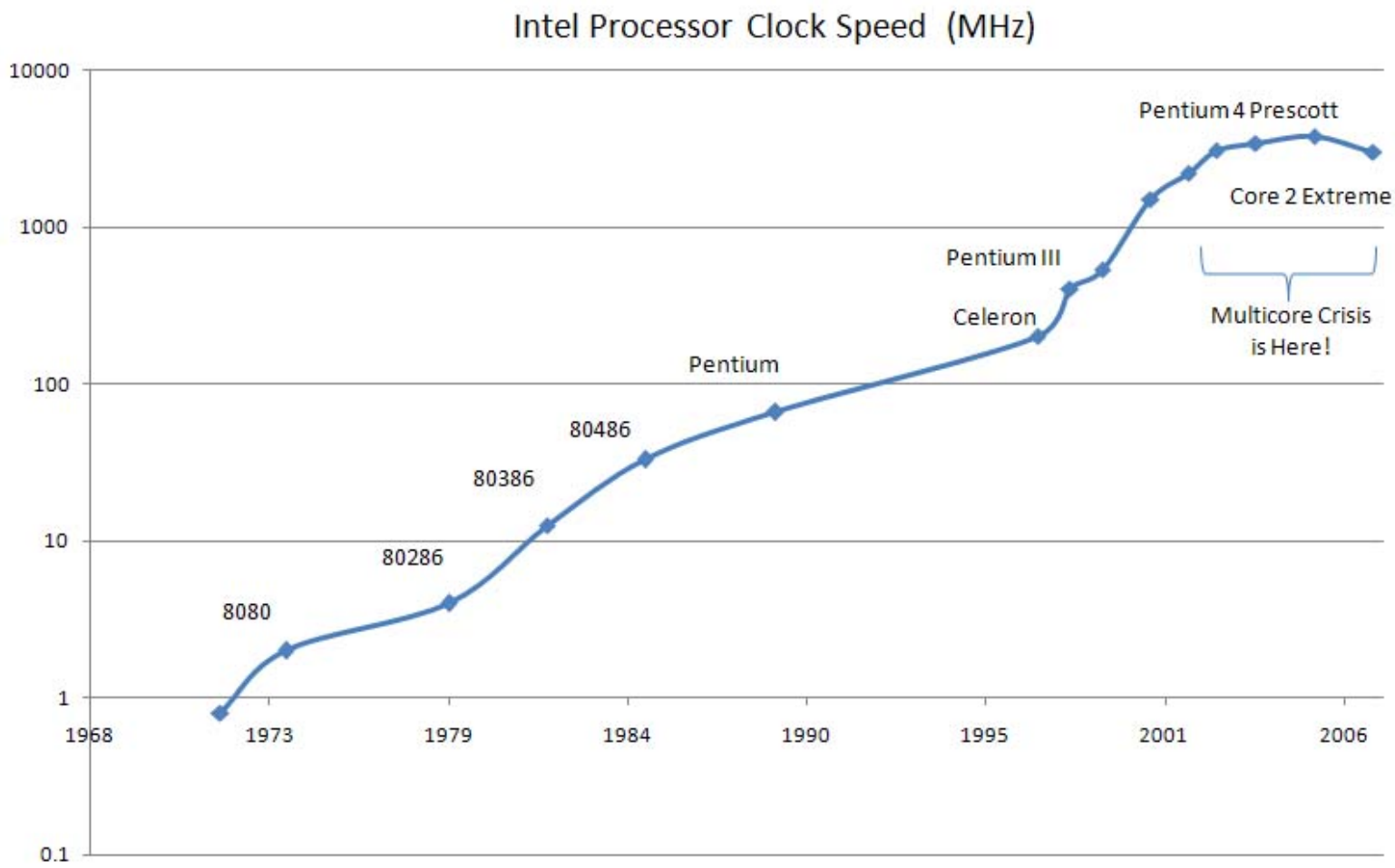
Joël Goossens

1. Introduction

Moore's law

- **Moore's law** refers to an observation made by Gordon Moore in 1965. He noticed that the number of transistors per square inch on integrated circuits had **doubled every year**, with a constant cost, since their invention.
- Moore's law predicts that this trend will continue into the foreseeable future.

Moore's law (cont.)



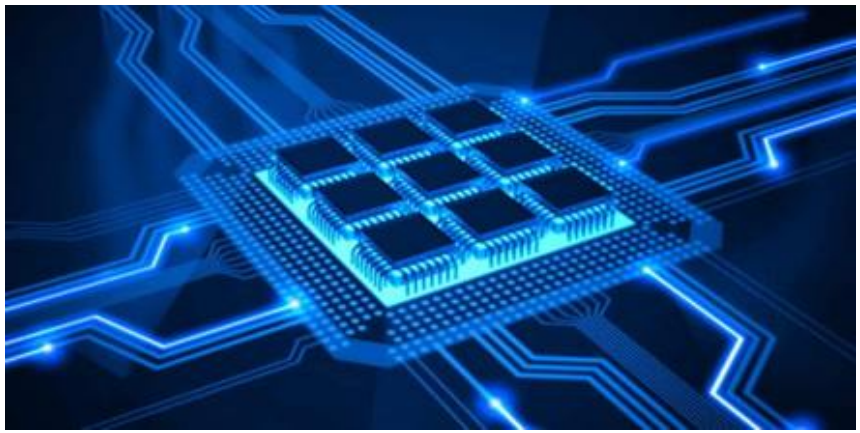
Moore's law (cont.)

> "***The chips are down for Moore's law***. *The semiconductor industry will soon abandon its pursuit of Moore's law. Now things could get a lot more interesting.*"
>
> — M. Mitchell WALDROP
> Nature **530**, 144–147 (February 2016).

- We reached limits
  - Physical limits: atomic scale, peak temperature.
  - Economical limits: too expensive.

Moore's law (cont.)

- The answer of the electronic industry, to provide more and more computing power, is the **parallel and multicore/multiprocessor architectures**.



Assumptions

- We consider platforms composed of **several** processors/cores.
- We consider strongly coupled systems, and which share a common time reference, and a common shared memory.

Limited Parallelism

- A multiprocessor platform is built according to a parallel architecture but this parallelism is **limited**:
  - At any given moment, each processor is idle or is executing **one** single job;
  - At any given moment, a job is either:
    - waiting to be executed;
    - it is being executed on **one** single processor;
    - or it has been completed.

Taxonomy of Multiprocessor Platforms

- We can distinguish between at least three kinds of multiprocessor platforms:
  - **Identical parallel machines.** Platforms on which all the processors are identical, in the sense that they have the same computing power.
  - **Uniform parallel machines.** By contrast, each processor in a uniform parallel machine is characterized by its own computing capacity. A job that is executed on a processor π_i with computing capacity s_i for t time units will be completed after $s_i \times t$ units of execution.
  - **Unrelated parallel machines.** In unrelated parallel machines, there is an execution rate $s_{i,j}$ associated with each job-processor pair. A job J_i that is executed on a processor π_j for t time units will be completed after $s_{i,j} \times t$ units of execution.
- Notice that Identical $\subset$ Uniform $\subset$ Unrelated.

Taxonomy of Multiprocessor Platforms (cont.)

- Identical platforms are homogeneous architectures
- Uniform and Unrelated ones are heterogeneous
- In this chapter, we consider the particular case of **identical** multiprocessors composed of m processors: $\pi_1, \pi_2, \ldots, \pi_m$.
- $m \leq n$ is the interesting case!

Young research topic! (Master & PhD Thesis)

- The scheduling theory for real-time uniprocessor platforms is well developed and has been covered extensively in the last 45+ years.
- On the other hand, real-time **multiprocessor** scheduling theory is much more recent and relatively few results are known so far. Especially heterogeneous multi-core and many-core (multi-clusters) architectures.

Taxonomy of Multiprocessor Schedulers

- We can distinguish between at least two kinds of scheduling techniques
  - **Partitioned Scheduling.** Task partitioning involves statically dividing the set of tasks among the processor cores. Each task is assigned to a specific processor core, and the scheduling of tasks within each core follows a **local** scheduling policy, such as earliest deadline first (EDF). We have m local queues. The scheduling decisions are made **independently** for each core
  - **Global Scheduling.** Global scheduling does not statically assign tasks to specific processor cores. Instead, the scheduling decisions are made dynamically and **globally** based on the current state of the system, including the job priorities, and availability of processors. We have unique global queue.

Illustrating partitioning

	C	T	D
τ_1	1	4	4
τ_2	3	5	5
τ_3	7	20	20

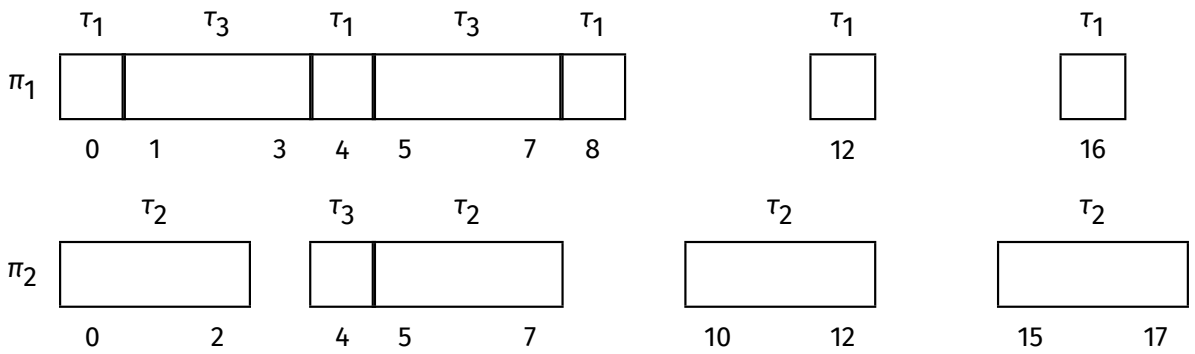
$U(\tau) = 1.2$

- We can schedule τ_1 and τ_2 on a first processor
- and τ_3 on a second processor.

Illustrating Global DM

	C_i	T_i	D_i
τ_1	1	4	4
τ_2	3	5	5
τ_3	7	20	20

$\tau_1 > \tau_2 > \tau_3$



- Notice that τ_3 migrates from π_1 to π_2 and from π_2 to π_1 for each job!
- **Exercise:** schedule the same task set according to global EDF

Partitioned and Global Scheduling are Incomparable

- Partitioned scheduling (which forbids task migration and thus job migration) **is not a special case** of global scheduling (which allows, but does not require, migration).
- We will illustrate incomparability for the general class of FJP schedulers.
- There are systems that global algorithms can schedule but which partitioned algorithms fail to schedule (Lemma 74).
- Somewhat counter-intuitively, there are systems which partitioned FJP algorithms can schedule, but which cannot be scheduled by any global FJP algorithm (Lemma 76).

Partitioned and Global Scheduling are Incomparable (cont.)

Lemma (Lem. 74)

There are task systems that are scheduled using global FJP algorithms that partitioned FJP algorithms cannot schedule.

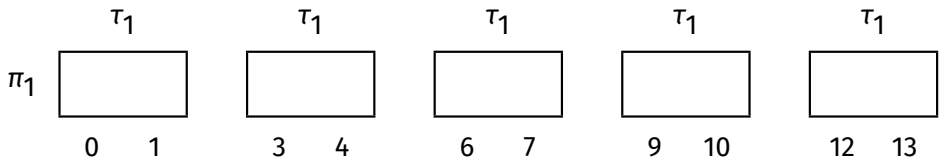
Proof. Here's an example task set scheduled on 2 processors

	C_i	T_i	D_i
τ_1	2	3	2
τ_2	3	4	3
τ_3	5	12	12

Obviously all partitions fail: $U(\tau_i) + U(\tau_j) > 1 \qquad \forall i \neq j.$

Partitioned and Global Scheduling are Incomparable (cont.)

We can schedule the tasks with a global FJP algorithm which assigns the lowest priority to τ_3's jobs. Since there are two processors, τ_1 and τ_2 meet all deadlines. For τ_3, over any 12 contiguous slots, τ_1 may execute during at most 8 slots.



If τ_1 executes for fewer than 8 slots, τ_3 is schedulable. If τ_1 executes for exactly 8 slots, τ_1's jobs must be arriving 3 time-units apart. Based upon τ_2's parameters, we know it is impossible that τ_2's jobs execute in parallel with τ_1's jobs. Consequently, we have at least one extra slot for τ_3 on the remaining processor.

Partitioned and Global Scheduling are Incomparable (cont.)

Lemma (Lem. 76)

There are task systems that are scheduled using partitioned FJP algorithms that global FJP algorithms cannot schedule.

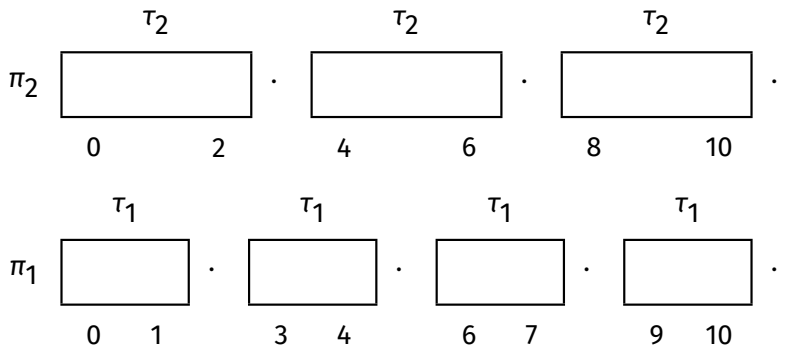
Proof. Here's an example task set scheduled on 2 processors

	C_i	T_i	D_i
τ_1	2	3	2
τ_2	3	4	3
τ_3	4	12	12
τ_4	3	12	12

This system may be partitioned: τ_1 and τ_3 on one processor and the remaining tasks on the remaining processor, each processor being scheduled using EDF.

Partitioned and Global Scheduling are Incomparable (cont.)

To show that no global FJP priority assignment can meet all deadlines, consider the synchronous arrival over $[0, 12)$. For feasibility's sake, we need to first serve all jobs of τ_1 and τ_2 ($C_i = D_i$). Whichever of τ_3 or τ_4's job has lower priority ends up missing its deadline while one processor goes idle over $[11, 12)$.



2. Partitioning

Partitioned Scheduling – A Short Introduction

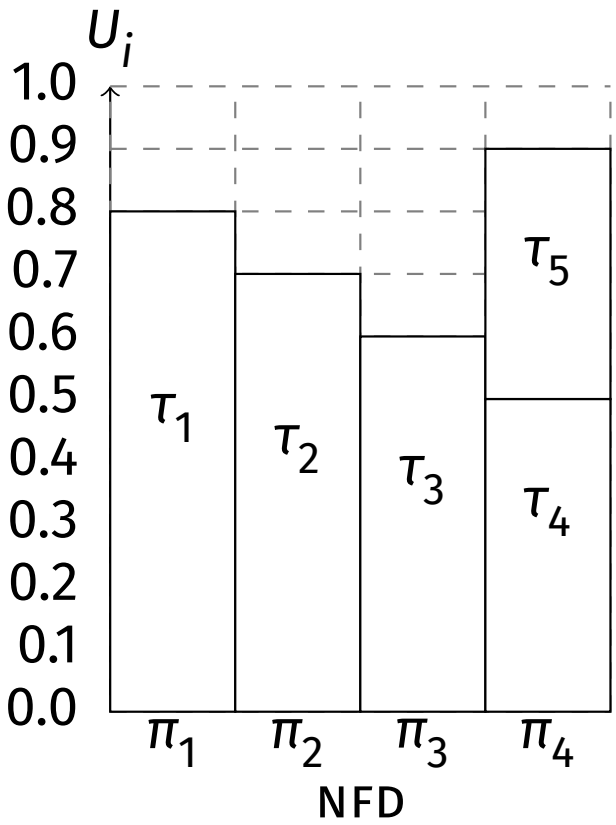
- Remark: in this section concerning partitioned scheduling, we consider implicit-deadline tasks.
- The partitioning problem is actually known as the **Bin-Packing** problem: k objects have to fit in a finite number of bins of capacity $\mathbb{C}$ in a way that minimizes the number of bins used.
- In our case, objects correspond to tasks and bins correspond to processors. The bin size is the largest system utilisation possible for the scheduler, e.g. $\ln 2$ under RM, 1 under EDF.
- The problem is NP-Complete. Many heuristics have been developed, e.g. next-fit and best-fit. These are fast, but often yield suboptimal solutions.

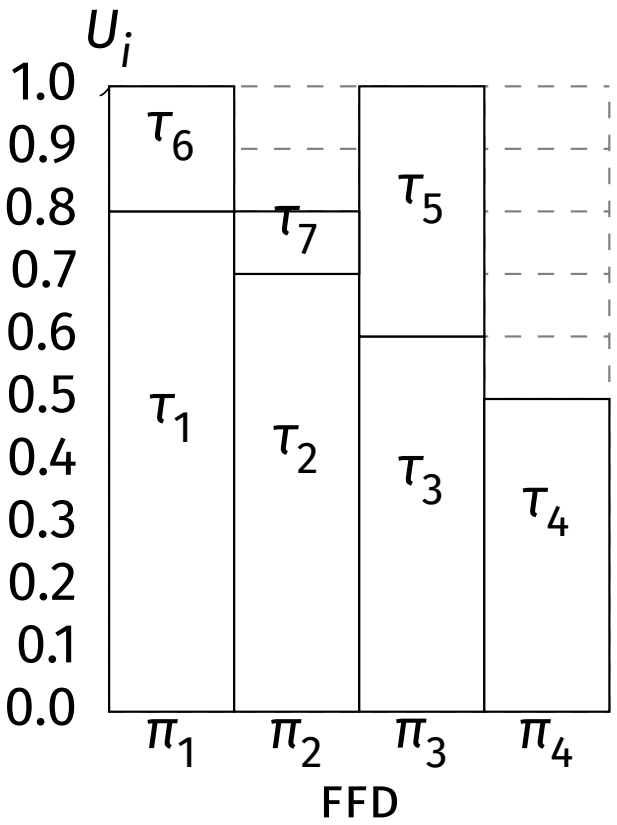
Heuristics

Greedy algorithm (build up a solution piece by piece).

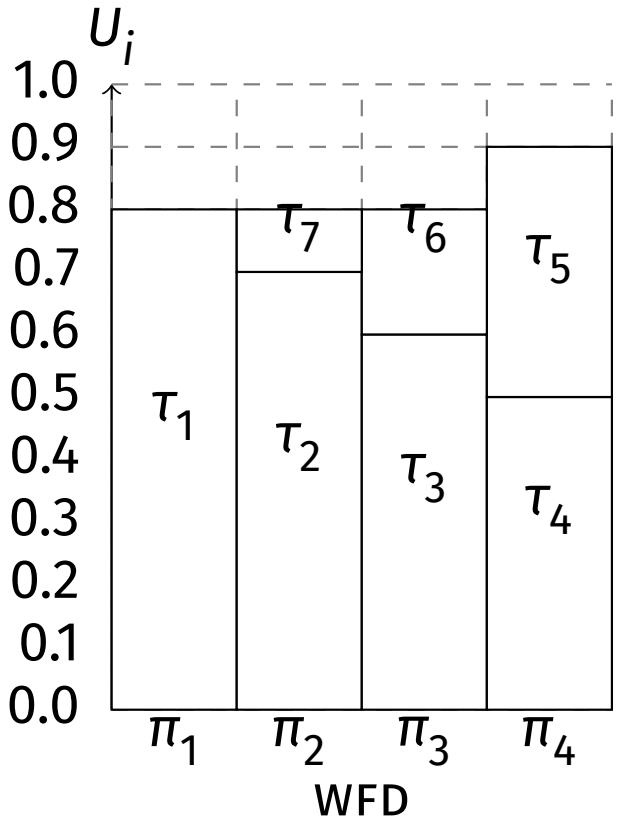
- It is possible to pre-sort the tasks by increasing or decreasing system utilisation.
- A rule to decide where to assign the current task:
  - *First-Fit*
  - *Best-Fit*
  - *Worst-Fit*
  - *Next-Fit*

Heuristics (cont.)





Heuristics (cont.)



Partitioned Rate Monotonic

- For RM, the processor size could be the bound $U(\tau) \leqslant n_\pi(2^{1/n_\pi} - 1)$ where n_π is the number of tasks on processor π ($n_\pi = \#\tau$).
- Notice that the bound is not tight enough to provide good partitions. In the worst case, we can lose about 50% of the platform capacity.

Partitioned Rate Monotonic (cont.)

Example ([2])

Consider $m + 1$ tasks with $T_i = 1$ and $C_i = \sqrt{2} - 1 + \epsilon$. Assume the platform size is m. There must be a processor π_ℓ which schedules two tasks. Upon that processor, the total utilization is $2 \cdot (\sqrt{2} - 1 + \epsilon)$, which is larger than $2 \cdot (\sqrt{2} - 1)$. Consequently, no partition can be found with the chosen bound.
If we consider the limit case (i.e. $\epsilon \to 0$ and $m \to \infty$), we cannot guarantee schedulability above a utilization of $\sqrt{2} - 1$, i.e. approximately 41%.

Partitioned EDF: FFDU

- For EDF, the bound is better (in fact, optimal) since we know that $U(\tau) \leqslant 1$ is a necessary and sufficient condition for uniprocessor schedulability.
- Lopez et al. proposed the First-Fit-Decreasing-Utilization (FFDU) algorithm. The technique considers tasks by decreasing utilization and uses first-fit as heuristic for the bin-packing problem. They proved the following schedulability condition:

Theorem (Th. 78)

A sporadic implicit-deadline task set τ is schedulable using FFDU *(using* EDF *as local scheduler) if*

$$U(\tau) \leqslant (m+1)/2 \qquad \textit{and} \qquad U_{\max} \leqslant 1$$

where $U_{\max} \stackrel{\text{def}}{=} \max\{U(\tau_i) \mid i = 1, \ldots, n\}$.

3. Global Scheduling

No online optimal scheduler exists — set of jobs

A first very important, but unfortunately negative result is that no online optimal scheduler exists.

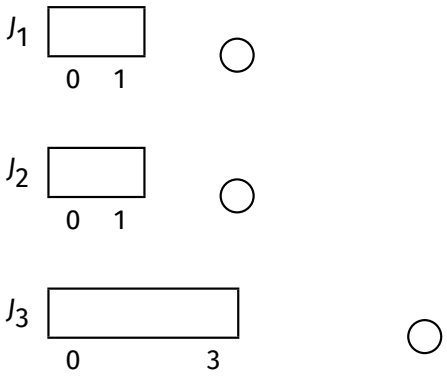
Definition (Online Schedulers)

Online schedulers take their scheduling decisions at **runtime**, based on the characteristics of the jobs already released and without any knowledge of **when** future jobs will be released.

Theorem (Th. 80)

For any $m > 1$, no online and optimal scheduling algorithm exists.

Illustrating the proof

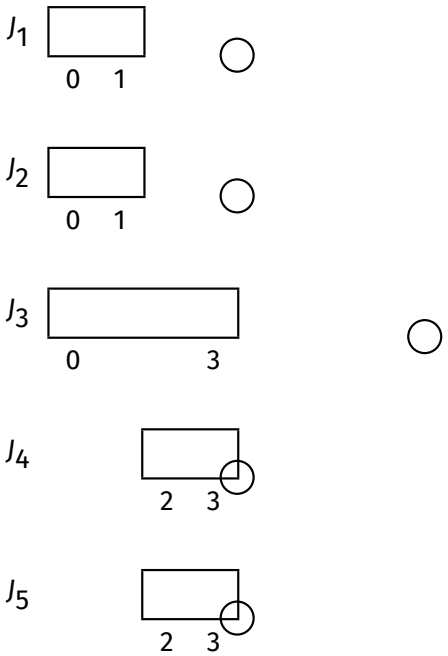


$(J_1 = (0, 2, 4))$

$(J_2 = (0, 2, 4))$

$(J_3 = (0, 4, 8))$

Illustrating the proof



$(J_1 = (0, 2, 4))$

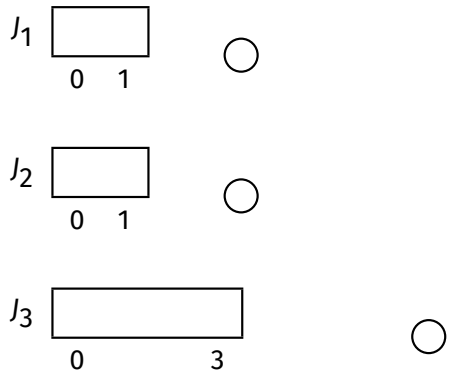
$(J_2 = (0, 2, 4))$

$(J_3 = (0, 4, 8))$

$(J_4 = (2, 2, 4))$

$(J_5 = (2, 2, 4))$

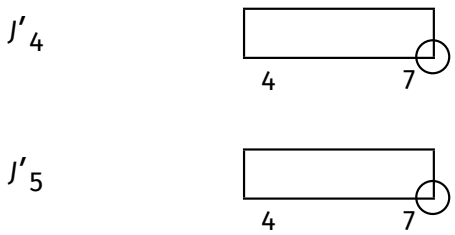
Illustrating the proof



$(J_1 = (0, 2, 4))$

$(J_2 = (0, 2, 4))$

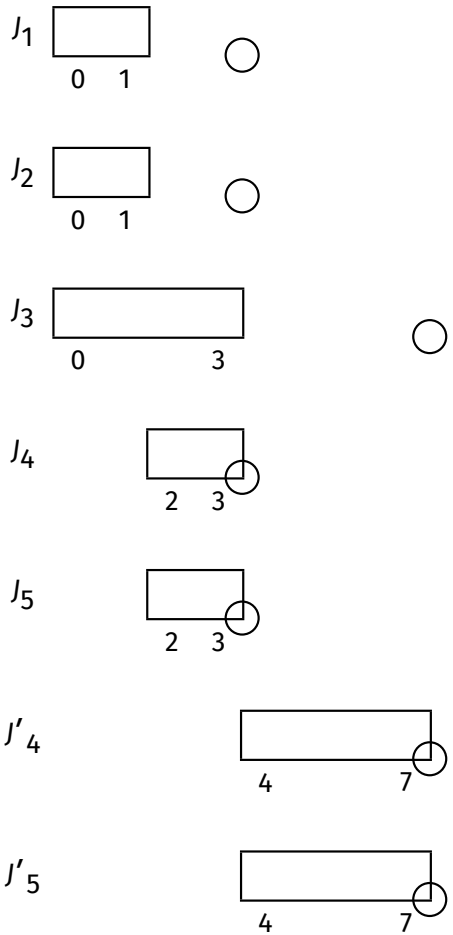
$(J_3 = (0, 4, 8))$



$(J'_4 = (4, 4, 8))$

$(J'_5 = (4, 4, 8))$

Illustrating the proof



$(J_1 = (0, 2, 4))$

$(J_2 = (0, 2, 4))$

$(J_3 = (0, 4, 8))$

$(J_4 = (2, 2, 4))$

$(J_5 = (2, 2, 4))$

$(J'_4 = (4, 4, 8))$

$(J'_5 = (4, 4, 8))$

Scheduling Anomalies — Definition

Definition (Anomaly)

A scheduling anomaly occurs when a change of the system parameters that intuitively seems positive in fact jeopardises the system's schedulability.

Informally, a schedulable system can become unschedulable even though the system was schedulable before a seemingly helpful change.

Definition (Intuitively positive change)

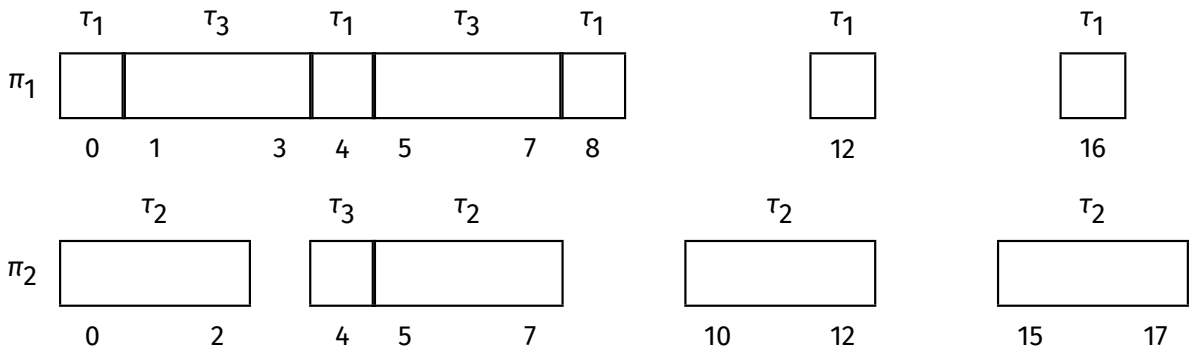
Any change which decreases the utilization factor of tasks: an increase of the period, a decrease of the computation requirement or, equivalently, an increase of the processor speeds.

Scheduling Anomalies – Example

- Multiprocessor schedulers are subject to scheduling anomalies
- For instance, for global FTP scheduling, we can find systems that are schedulable (for a given FTP-priority assignment), but where increasing the period of a task will cause a deadline miss!

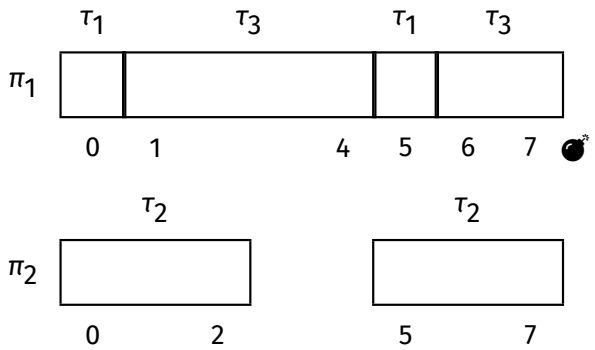
Scheduling Anomalies – Example (cont.)

This is the case of system $\tau_1 = (T_1 = 4, D_1 = 2, C_1 = 1), \tau_2 = (T_2 = 5, D_2 = 3, C_2 = 3), \tau_3 = (T_3 = 20, D_3 = 8, C_3 = 7)$
The system is FTP-schedulable on 2 processors using global DM ($\tau_1 > \tau_2 > \tau_3$)



Scheduling Anomalies – Example (cont.)

Unfortunately, if we increase the period of τ_1 from 4 to 5, the same scheduler (global DM) will cause τ_3 to miss its deadline at instant 8.



Consequences of Scheduling Anomalies

- In order to check the schedulability of **sporadic** tasks, there is no interest (at least it is not sufficient) to check the synchronous periodic sub-case.
- That question is in fact an **open question**:

We have no idea what the critical instant is in the context of multiprocessor scheduling

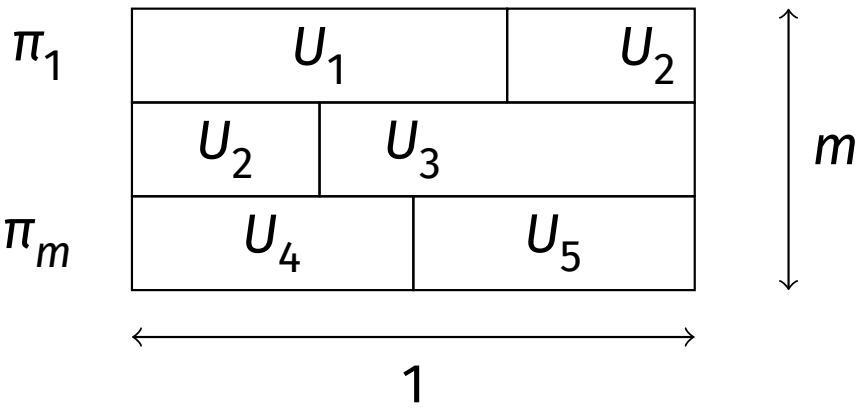
Periodic Implicit-Deadline Systems — Necessary and Sufficient Condition

Theorem

Any Periodic Implicit-Deadline System is feasible iff

$$U(\tau) \leqslant m \qquad \textit{and} \qquad U_{\max}(\tau) \leqslant 1$$

Proof sketch:



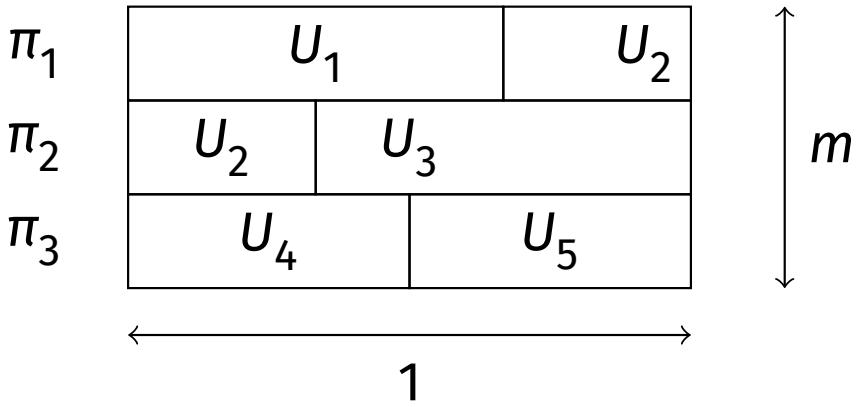
Introduction to Fairness

- In an **ideal** fair schedule, each task receives the processor for an **exact** duration of $U(\tau_i) \times t$ on the interval $[0, t)$.
- This implies that all the deadlines are met, but might be practically **impossible** to implement: this might require an large number of preemptions too many preemptions that are moreover **within** the time slots, which is not possible in a discrete-time situation.

PFAIR Scheduling

- **optimal** scheduler (PFAIR), for multiprocessors in the particular case of periodic and implicit-deadline task sets
- We consider synchronous implicit-deadline periodic tasks and we consider the time as discrete since the timeline is divided into quanta (or time slices).
- The scheduler definition requires the formalization of the schedule: a function $S : \tau \times \mathbb{N} \mapsto \{0, 1\}$, where τ is a periodic task set.
- $S(\tau_i, t) = 1$ means that τ_i is scheduled in the slice $[t, t + 1)$ and $S(\tau_i, t) = 0$ means otherwise.
- In an ideal fair (continuous) schedule, each task receives the processor for an exact duration of $U(\tau_i) \times t$ on the interval $[0, t)$ (which implies that all deadlines are met).

PFAIR Scheduling (cont.)



Lag

- The idea behind PFAIR is to mimic the ideal fair schedule as closely as possible. At any instant, the difference between the actual schedule and the ideal fair schedule is formalized using the notion of **lag**. The lag of task τ_i at time t, denoted $\text{lag}(\tau_i, t)$ is defined as follows:

$$\text{lag}(\tau_i, t) \stackrel{\text{def}}{=} U(\tau_i) \cdot t - \sum_{\ell=0}^{t-1} S(\tau_i, \ell). \quad (1)$$

PFAIR Schedules

Definition (Def. 88, PFAIR Schedule)

A schedule has the PFAIR property iff

$$-1 < \text{lag}(\tau_i, t) < 1 \qquad \forall \tau_i \in \tau, \forall t \in \mathbb{Z}. \tag{2}$$

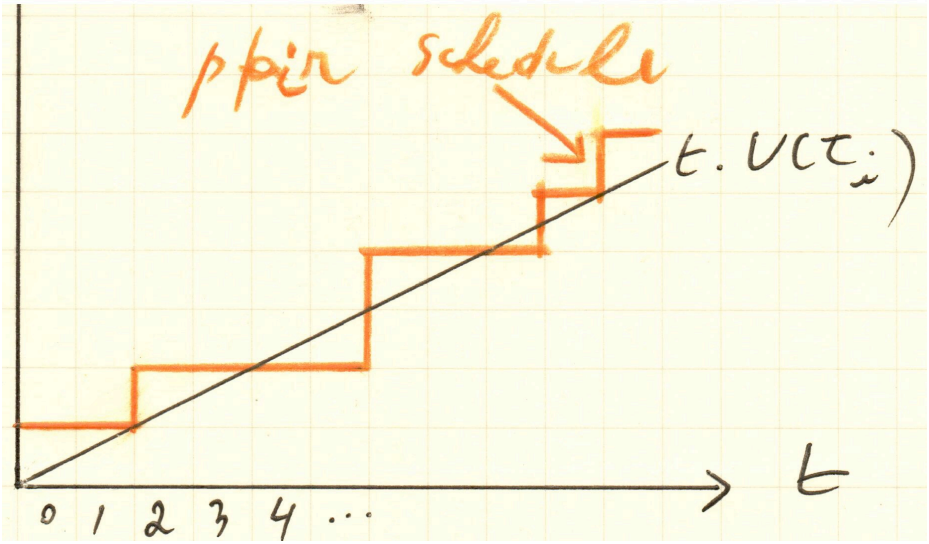
- Informally speaking, Equation 2 means that at any time τ_i has to receive either

$$\lfloor U(\tau_i) \cdot t \rfloor$$

or,

$$\lceil U(\tau_i) \cdot t \rceil$$

PFAIR Schedules (cont.)



PFAIR Schedules (cont.)

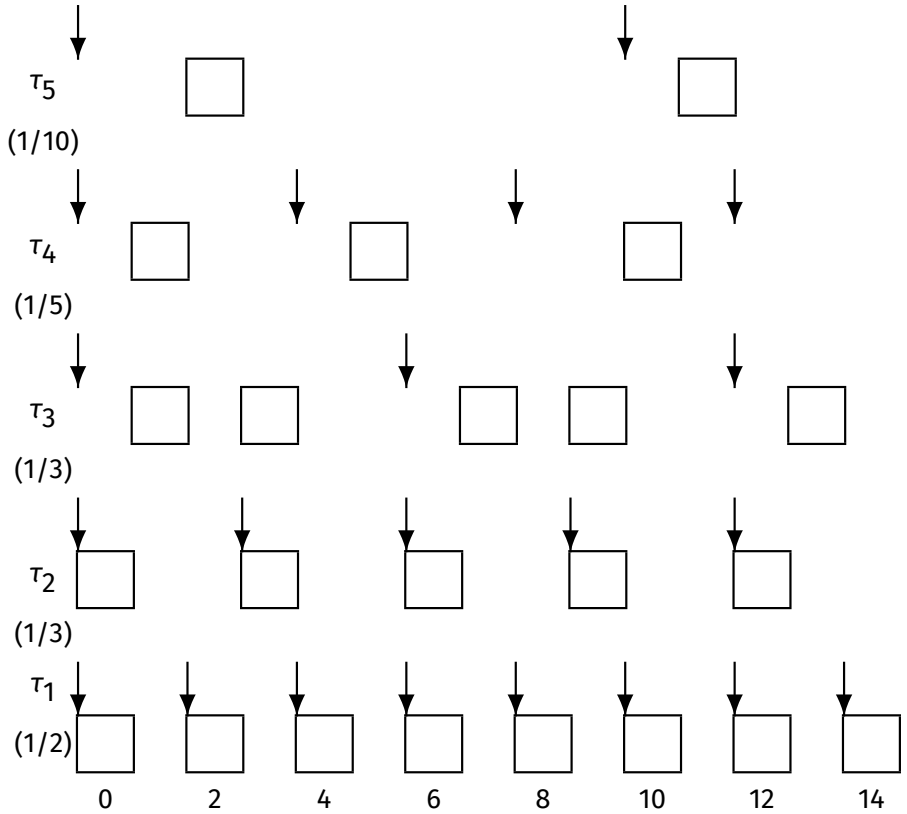
Theorem (Th. 89)

PFAIRness implies that all deadlines are met.

Proof Sketch.

- Consider a deadline: $t = k \cdot T_i$
- $U(\tau_i) \cdot t = \frac{C_i}{T_i} \cdot k \cdot T_i = k \cdot C_i \in \mathbb{N}$

PFAIR Example



PFAIR Optimality

Theorem (Th. 90)

Let τ be a periodic synchronous implicit-deadline system. A PFAIR schedule exists for τ on m processors iff

$$U(\tau) \leqslant m \qquad \text{and} \qquad U_{\max} \leqslant 1. \tag{3}$$

PFAIR Optimality (cont.)

- In other words, PFAIR schedules any schedulable system. Thus, PFAIR is **optimal**.
- Notice this does not contradict Theorem 80 (non-existence of online optimal schedulers) since we only consider periodic tasks in this case (as such, we know the release times of all future jobs).

PFAIR Schedulers

Concerning algorithms to generate PFAIR schedules, we can report the following works: PF [3], PD [4] and PD^2 [1].

Global EDF and Sporadic Implicit-Deadline Systems

Theorem (Th. 91)

Any Sporadic Implicit-Deadline System is schedulable using global EDF *on m processors if*

$$U(\tau) \leqslant m - (m-1)U_{\max}$$

Notice that this test is very pessimistic if $U_{\max} \approx 1$. We will introduce the EDF$^{(k)}$ algorithm which addresses this drawback.

EDF$^{(k)}$ Scheduling Algorithm

- In order to simplify notations, we assume $U(\tau_1) \geqslant U(\tau_2) \geqslant \cdots \geqslant U(\tau_n)$ in the following.
- We also introduce the notation $\tau^{(i)}$ to denote the set of the $(n - i + 1)$ tasks with the lowest utilization factors of τ:

$$\tau^{(i)} \stackrel{\text{def}}{=} \{\tau_i, \tau_{i+1}, \ldots, \tau_n\}$$

- We can derive the following from Theorem 91:

$$m \geqslant \left\lceil \frac{U(\tau) - U(\tau_1)}{1 - U(\tau_1)} \right\rceil$$

- We can adapt EDF slightly to require a smaller number of processors than $\left\lceil \frac{U(\tau)-U(\tau_1)}{1-U(\tau_1)} \right\rceil$.

EDF$^{(k)}$ Scheduling Algorithm (cont.)

Definition (EDF$^{(k)}$)

- For all $i < k$, the jobs of τ_i receive the maximal priority (which can be done by modifying their absolute deadlines to $-\infty$).
- For all $i \geqslant k$, the jobs of τ_i are assigned priorities according to EDF.

EDF$^{(k)}$ Scheduling Algorithm (cont.)

- In other words, EDF$^{(k)}$ gives the highest priority to the jobs of the $k-1$ first tasks of τ and schedules the other tasks according to EDF.
- Notice that "pure" EDF corresponds to EDF$^{(1)}$.

EDF$^{(k)}$ Scheduling Algorithm (cont.)

Theorem (Th. 93)

A sporadic implicit-deadline system is schedulable on m processors using EDF$^{(k)}$ *if*

$$m = (k-1) + \left\lceil \frac{U(\tau^{(k+1)})}{1 - U(\tau_k)} \right\rceil \tag{4}$$

- Informally, EDF$^{(k)}$ requires a processor (π_1) for the jobs of τ_1, π_2 for the jobs of τ_2, ..., π_{k-1} for the jobs of τ_{k-1} and uses EDF for the jobs of the others tasks $\tau_k, \ldots, \tau_n$ on additional processors $\left(\pi_k, \ldots, \pi_{k-1+\left\lceil \frac{U(\tau^{(k+1)})}{1-U(\tau_k)} \right\rceil}\right)$

EDF$^{(k)}$ Scheduling Algorithm (cont.)

Corollary (Cor. 94)

A sporadic implicit-deadline system τ is schedulable on $m_{\min}$ processors using EDF$^{(m_{\min})}$ *with*

$$m_{\min}(\tau) \stackrel{\text{def}}{=} \min_{k=1}^{n} \left\{ (k-1) + \left\lceil \frac{U(\tau^{(k+1)})}{1 - U(\tau_k)} \right\rceil \right\} \tag{5}$$

Let $k_{\min}(\tau)$ be the k which minimizes Equation 5:

$$m_{\min}(\tau) = (k_{\min}(\tau) - 1) + \left\lceil \frac{U(\tau^{(k_{\min}(\tau)+1)})}{1 - U(\tau_{k_{\min}})} \right\rceil$$

EDF$^{(k)}$ – Example

Consider the system τ composed of 5 tasks:

$$\tau = \{(9, 10), (14, 19), (1, 3), (2, 7), (1, 5)\};$$

we have: $U(\tau_1) = 0.9$, $U(\tau_2) = 14/19 \approx 0.737$, $U(\tau_3) = 1/3$, $U(\tau_4) = 2/7 \approx 0.286$, et $U(\tau_5) = 0.2$; $U(\tau) \approx 2.457$.
We can see that Equation 5 is minimized for $k = 3$; thus, $k_{\min}(\tau) = 3$ and $m_{\min}(\tau)$ is

$$\begin{aligned} & (3-1) + \left\lceil \frac{0.286 + 0.2}{1 - 0.334} \right\rceil \\ = \quad & 2 + \left\lceil \frac{0.486}{0.667} \right\rceil \\ = \quad & 3 \end{aligned}$$

EDF$^{(k)}$ – Example (cont.)

Consequently, τ can be scheduled with EDF$^{(3)}$ on 3 processors.

On the other hand, Theorem 91 cannot guarantee schedulability below $\left\lceil \frac{U(\tau)-U(\tau_1)}{1-U(\tau_1)} \right\rceil \approx \lceil 1.557/0.1 \rceil = 16$ processors using "pure" EDF.

Bibliography

[1] Anderson, J., and Srinivasan, A.
Early-release fair scheduling.
In *12th Euromicro Conference on Real-Time Systems* (2000), pp. 35–43.

[2] Andersson, B.
Static-priority scheduling on multiprocessors.
PhD thesis, Chalmers University of Technology, 2003.

[3] Baruah, S., Cohen, N., Plaxton, C. G., and Varvel, D.
Proportionate progress: A notion of fairness in resource allocation.
Algorithmica 15 (1996), 600–625.

[4] Baruah, S., Gehrke, J., and Plaxton, C. G.
Fast scheduling of periodic tasks on multiple resources.
In *9th International Parallel Processing* (1995), pp. 280–288.

Bibliography (cont.)