# INFO-F404, Operating Systems II

## Multiprocessor real-time scheduling
## Academic year 2013–2014

Joël Goossens

Université Libre de Bruxelles

# Multiprocessor Scheduling

## C) Multiprocessor Scheduling

# Context

- In this chapter, we consider platforms composed of **several** processors/cores. Complex real-time applications require powerful, costly and power-consuming uniprocessor platforms or — for approximately the same computing power — cheaper and power-efficient multiprocessor platforms. The reasons for this paradigm shift (uni- to multiprocessor) is due to the cost and difficulty nowadays to build integrated circuits with a high density and at a high working frequency.
- In this framework, we consider strongly coupled systems, i.e. we assume a common time reference and a common shared memory.

# Limited Parallelism

- The multiprocessor platform is obviously a parallel architecture but the parallelism considered here is **limited**:
  - at each instant, each processor executes one task at most
  - at each instant, each task is executed on one processor at most, i.e. we forbid task and job parallelism.

# Taxonomy of Multiprocessor Platforms I

- 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 $\pi_i$ 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 $\pi_j$ 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 II

- 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: $\pi_1, \pi_2, \ldots, \pi_m$.

# 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 40 years.
- On the other hand, real-time **multiprocessor** scheduling theory is much more recent and relatively few results are known so far.

# Taxonomy of Multiprocessor Schedulers I

- We can distinguish between at least two kinds of scheduling techniques
  - **Partitioned Scheduling.** This approach aims to find a strategy to partition tasks on processors. Once a task is assigned to a processor, it cannot migrate. A local (uniprocessor) scheduler is used on each processor. Thus, once the distribution of tasks is done, a **uniprocessor** scheduling feasibility condition is used to decide the schedulability of the system.
  - **Global Scheduling.** This approach aims to find a global strategy to schedule jobs. Jobs are allowed to migrate between processors during their lifetime, i.e. start their execution on one processor and later resume on another processor.

# Illustrating partitioning

| | $C$ | $T$ | $D$ |
|---|---|---|---|
| $\tau_1$ | 1 | 4 | 4 |
| $\tau_2$ | 3 | 5 | 5 |
| $\tau_3$ | 7 | 20 | 20 |

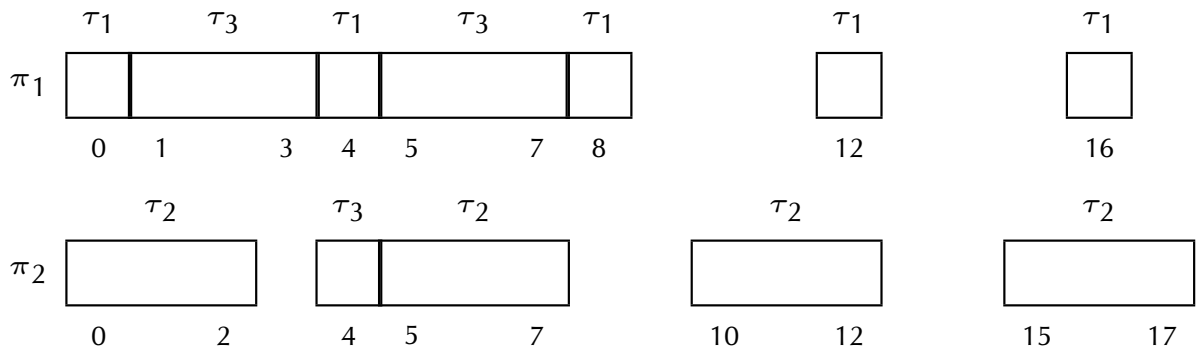
$$U(\tau) = 1.2$$

- Nous pouvons ordonnancer $\tau_1$ et $\tau_2$ sur un seul processeur
- et $\tau_3$ sur un second processeur.

# Illustrating Global DM

| | $C_i$ | $T_i$ | $D_i$ |
|---|---|---|---|
| $\tau_1$ | 1 | 4 | 4 |
| $\tau_2$ | 3 | 5 | 5 |
| $\tau_3$ | 7 | 20 | 20 |

$\tau_1 \succ \tau_2 \succ \tau_3$




- Notice that $\tau_3$ migrates from $\pi_1$ to $\pi_2$ and from $\pi_2$ to $\pi_1$ for each job!
- **Exercise:** schedule the same task set according to global EDF

# Partitioned and Global Scheduling are Incomparable I

- 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 1).
- Somewhat counter-intuitively, there are systems which partitioned FJP algorithms can schedule, but which cannot be scheduled by any global FJP algorithm (Lemma 2).

# Partitioned and Global Scheduling are Incomparable II

## Lemma 1 ([Baruah, 2007])

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

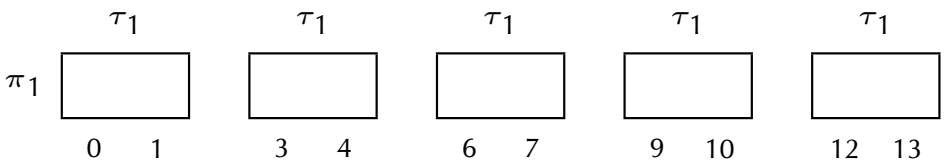
*Proof.* Here's an example task set scheduled on 2 processors

| | $C_i$ | $T_i$ | $D_i$ |
|---|---|---|---|
| $\tau_1$ | 2 | 3 | 2 |
| $\tau_2$ | 3 | 4 | 3 |
| $\tau_3$ | 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 III

We can schedule the tasks with a global FJP algorithm which assigns the lowest priority to $\tau_3$'s jobs. Since there are two processors, $\tau_1$ and $\tau_2$ meet all deadlines. For $\tau_3$, over any 12 contiguous slots, $\tau_1$ may execute during at most 8 slots.




If $\tau_1$ executes for fewer than 8 slots, $\tau_3$ is schedulable. If $\tau_1$ executes for exactly 8 slots, $\tau_1$'s jobs must be arriving 3 time-units apart. Based upon $\tau_2$'s parameters, we know it is impossible that $\tau_2$'s jobs execute in parallel with $\tau_1$'s jobs. Consequently, we have at least one extra slot for $\tau_3$ on the remaining processor.

# Partitioned and Global Scheduling are Incomparable IV

## Lemma 2 ([Baruah, 2007])

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

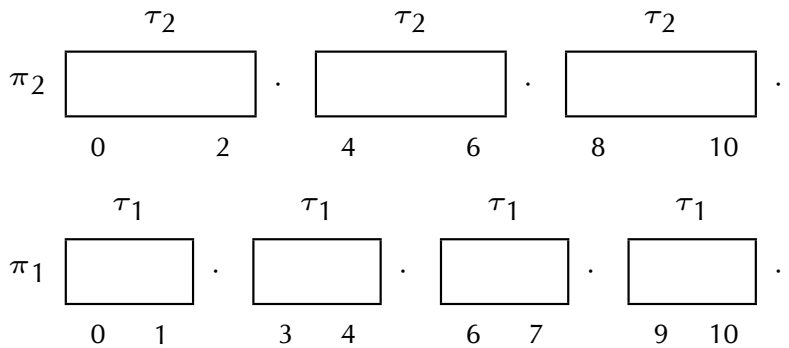
*Proof.* Here's an example task set scheduled on 2 processors

| | $C_i$ | $T_i$ | $D_i$ |
|---|---|---|---|
| $\tau_1$ | 2 | 3 | 2 |
| $\tau_2$ | 3 | 4 | 3 |
| $\tau_3$ | 4 | 12 | 12 |
| $\tau_4$ | 3 | 12 | 12 |

This system may be partitioned: $\tau_1$ and $\tau_3$ on one processor and the remaining tasks on the remaining processor, each processor being scheduled using EDF.

# Partitioned and Global Scheduling are Incomparable V

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 $\tau_1$ and $\tau_2$ ($C_i = D_i$). Whichever of $\tau_3$ or $\tau_4$'s job has lower priority ends up missing its deadline while one processor goes idle over $[11, 12)$.



# Density

A generalization of the concept of utilization is known as **density**:

## Definition 3 (Density $\lambda$)

A task $\tau_i$'s **density** $\lambda_i$ is defined as follows: $C_i / \min(D_i, T_i)$. The **total density** $\lambda_{\text{sum}}(\tau)$ and the **maximal density** $\lambda_{\text{max}}(\tau)$ are defined as follows:

$$\lambda_{\text{sum}}(\tau) \stackrel{\text{def}}{=} (\sum_{\tau_i \in \tau} \lambda_i) \qquad \lambda_{\text{max}}(\tau) \stackrel{\text{def}}{=} (\max_{\tau_i \in \tau} \lambda_i)$$

# Necessary Conditions

For the scheduling of sporadic tasks upon $m$ identical processors, the following conditions are necessary:

$$\lambda_{\text{sum}}(\tau) \leqslant m \qquad \text{and} \qquad \lambda_{\text{max}}(\tau) \leqslant 1$$

# Partitioned Scheduling – A Short Introduction

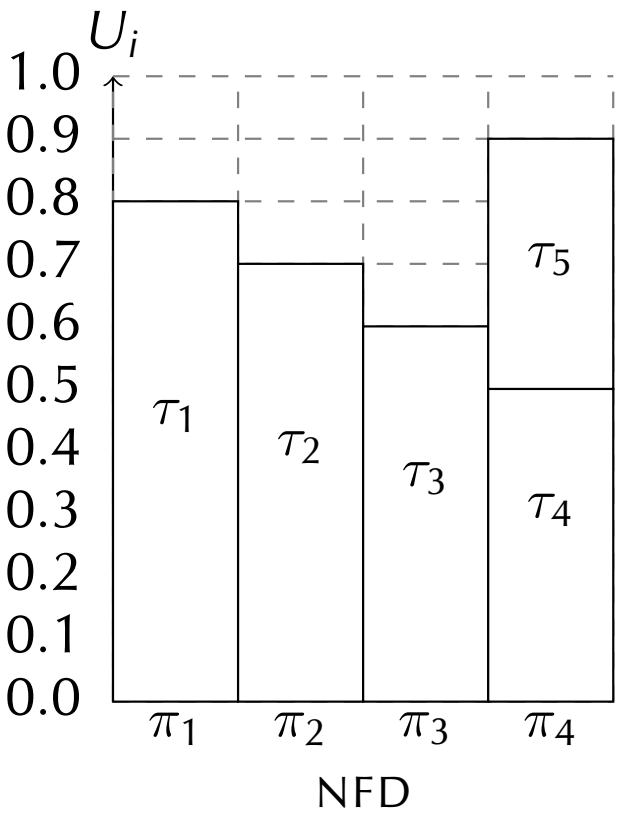
- Remark: in this section concerning partitioned scheduling, we consider implicit-deadline tasks.
- The partitioning problem is actually known as the **Bin-Packing** problem: $k$ objects of different sizes must be packed into a finite number of bins of capacity $B$ in a way that minimizes the number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

# Heuristics I

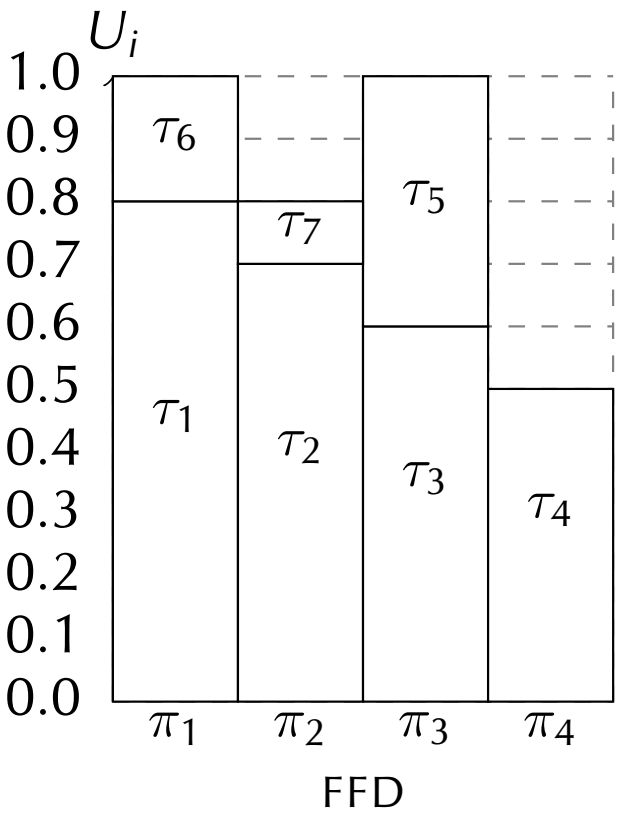
Greedy algorithm (build up a solution piece by piece).

- optional pre-processing which sorts the tasks
  - Decreasing utilization
  - Increasing utilization
- A rule to decide where to assign the current task:
  - *First-Fit*
  - *Best-Fit*
  - *Worst-Fit*
  - *Next-Fit*

# Heuristics II

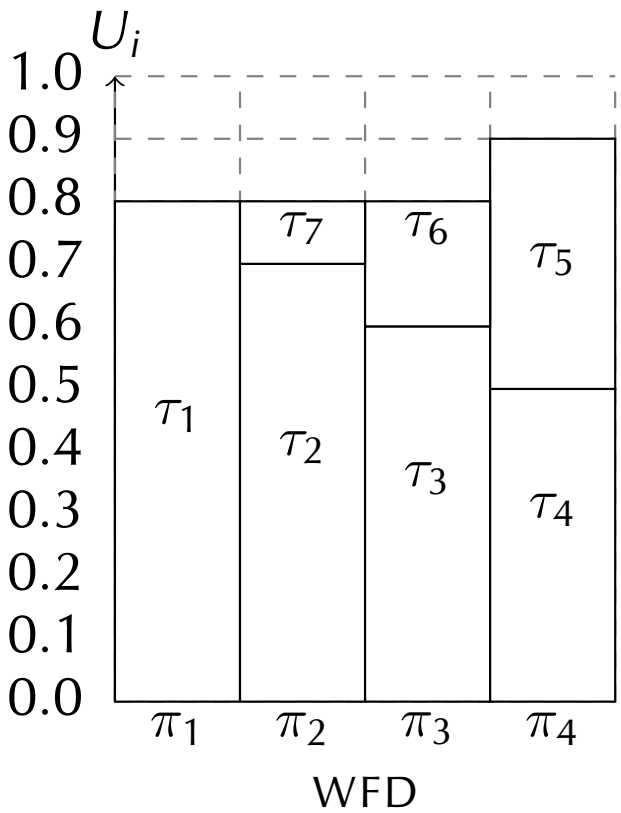



NFD




FFD

# Heuristics III



# Partitioned Rate Monotonic I

- For RM, the processor size could be the bound $U(\tau) < n_\pi(2^{1/n_\pi} - 1)$ where $n_\pi$ is the number of tasks on processor $\pi$ ($n_\pi = \#\tau$).
- 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 II

## Example 4 ([Andersson, 2003])

Consider $m + 1$ tasks with $T_i = 1$ and $C_i = \sqrt{2} - 1 + \varepsilon$. Assume the platform size is $m$. There must be a processor $\pi_\ell$ which schedules two tasks. Upon that processor, the total utilization is $2 \cdot (\sqrt{2} - 1 + \varepsilon)$, 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. $\varepsilon \to 0$ and $m \to \infty$), we cannot guarantee schedulability above a utilization of $\sqrt{2} - 1$, i.e. approximately 41%.

# Partitioned EDF: FFDU I

- 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 5

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

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

# No online optimal scheduler exists — set of jobs I

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

## Definition 6 (Online Schedulers)

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

## Theorem 7 ([Hong and Leung, 1988])

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

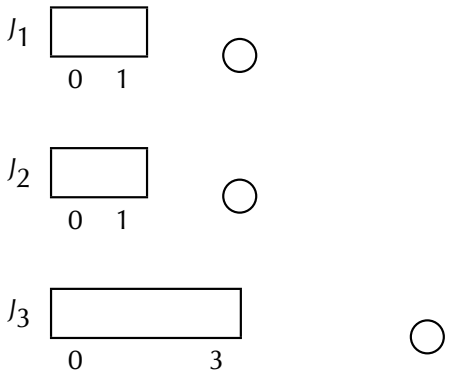
# No online optimal scheduler exists — set of jobs II

*Proof.* We consider the case where $m = 2$ (the arguments used here can be extended to any $m > 2$). The proof is made by contradiction. At instant 0, three jobs $J_1, J_2$ and $J_3$ are released with $d_1 = d_2 = 4$, $d_3 = 8$, $e_1 = e_2 = 2$ and $e_3 = 4$. The algorithm schedules the jobs on two processors starting at instant 0. We can distinguish between two kinds of online schedulers:
*Case 1:* $J_3$ executes during the interval $[0, 2)$. In that case, at least one of the other jobs ($J_1$ or $J_2$) will not complete at instant 2. WLOG, suppose this is the case of $J_2$. Now, assume that two new jobs ($J_4$ and $J_5$) are released at instant 2 with $d_4 = d_5 = 4$ and $e_4 = e_5 = 2$. Obviously, the online schedule fails to meet deadlines even though the system (the 5 jobs) is feasible. The scheduler should not have executed $J_3$ in the interval $[0, 2)$. Unfortunately, that decision would not have been optimal either (see Case 2).

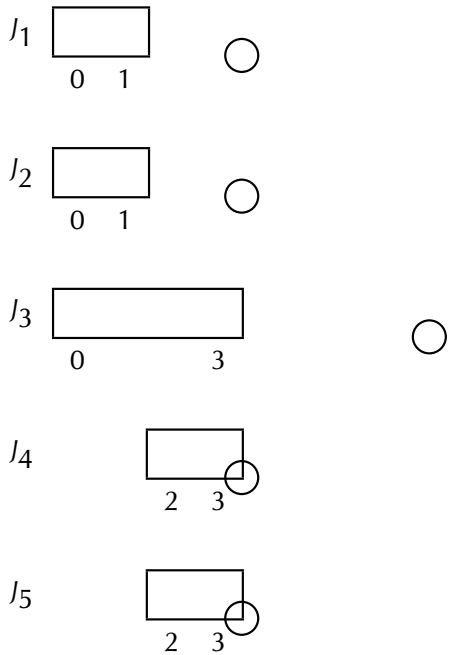
# No online optimal scheduler exists — set of jobs III

*Case 2:* $J_3$ does not progress at all in the interval $[0, 2)$. In that case, at instant 4, the remaining execution requirement is at least two time units. Now, consider that two new jobs ($J'_4$ and $J'_5$) are released at instant 4 with $d'_4 = d'_5 = 8$ and $e'_4 = e'_5 = 4$. Once again, the online schedule fails to meet the deadlines even though the system (the 5 jobs) is feasible. The scheduler should have executed $J_3$ during $[0, 2)$. ■

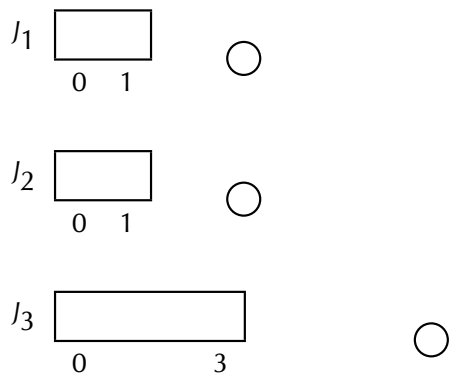
# Illustrating the proof

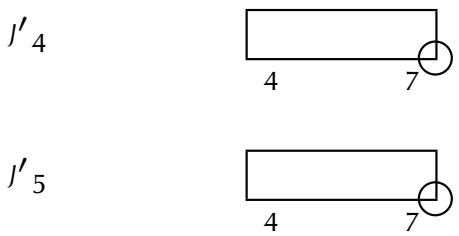


# Illustrating the proof

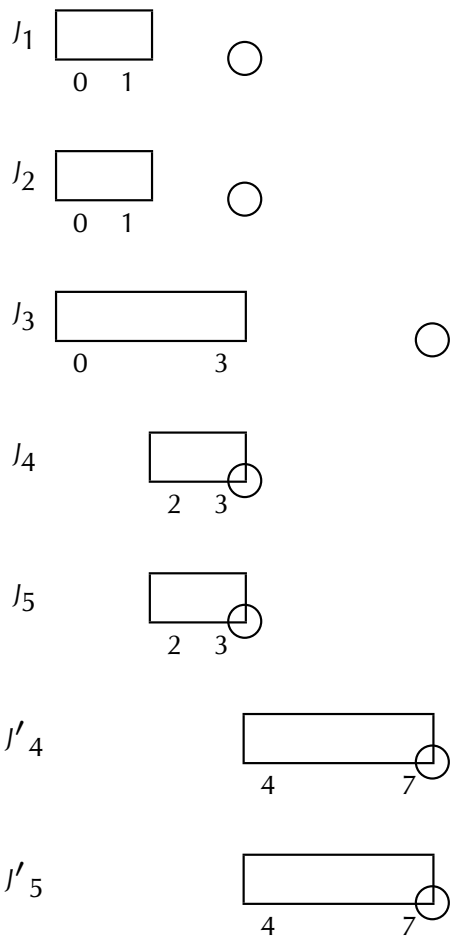


# Illustrating the proof



# Illustrating the proof



# No online optimal scheduler exists — set of sporadic tasks I

- Hong's counter-example concerns the scheduling of arbitrary jobs and cannot be extended to show the non-existence of optimal online multiprocessor scheduling algorithms for **sporadic** task systems.
- In [Fisher et al., 2010], we have shown that the following sporadic task system is feasible, but an optimal online scheduler for it requires **clairvoyance**.

# No online optimal scheduler exists — set of sporadic tasks II

| | $C_i$ | $D_i$ | $T_i$ |
|---|---|---|---|
| $\tau_1$ | 2 | 2 | 5 |
| $\tau_2$ | 1 | 1 | 5 |
| $\tau_3$ | 1 | 2 | 6 |
| $\tau_4$ | 2 | 4 | 100 |
| $\tau_5$ | 2 | 6 | 100 |
| $\tau_6$ | 4 | 8 | 100 |

- The challenge here is to show the feasibility of the task set.

# Scheduling Anomalies — Definition

## Definition 8 (Anomaly)

A scheduling anomaly occurs when an intuitively positive change (Definition 9) of the system parameters (platform, tasks or jobs) may jeopardize the system's schedulability.

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

## Definition 9 (Intuitively positive change)

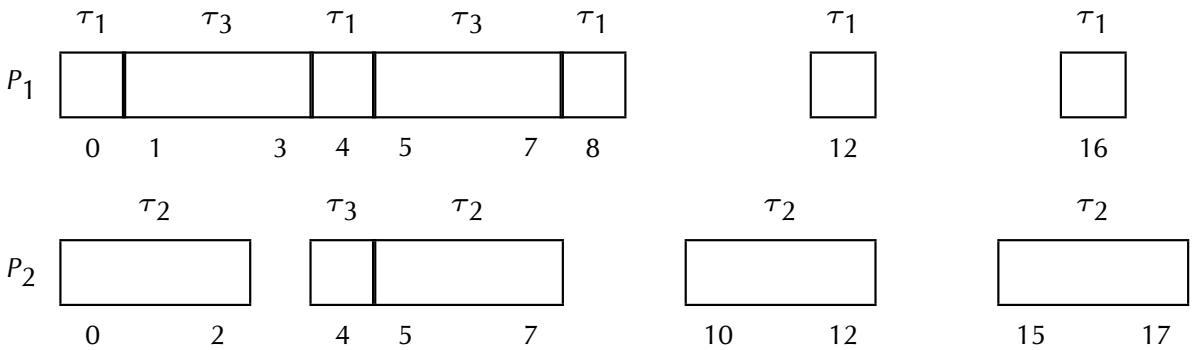
Any change which decreases the utilization factor of tasks: an increase of the period, a decrease of the computation requirement or, equivalently, an increase of the processor speeds.

# Scheduling Anomalies – Example I

- 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!

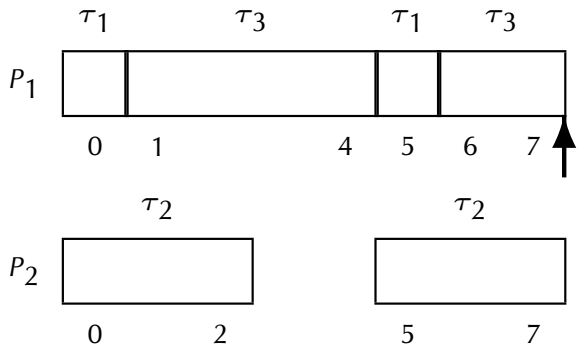
# Scheduling Anomalies – Example II

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
$(\tau_1 \succ \tau_2 \succ \tau_3)$



# Scheduling Anomalies – Example III

Unfortunately, if we increase the period of $\tau_1$ from 4 to 5, the same scheduler (global DM) will cause $\tau_3$ 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

# Sustainability I

## Definition 10 (Sustainability [Baruah and Burns, 2006])

A schedulability test for a scheduling policy is **sustainable** if any system deemed schedulable by the schedulability test remains schedulable when the parameters of one or more individual jobs are changed in any, some, or all of the following ways: (i) decreased execution requirements; (ii) later arrival times; and (iii) larger relative deadlines.

In the framework of this chapter, we are focusing on sustainable schedulers with respect to the execution requirement parameters (also known as predictability).

# Sustainability II

## Definition 11 (Sustainable with Respect to the Execution Requirements)

Let's consider the sets of jobs $J$ and $J'$ which differ only with regards to their execution times: the jobs in $J$ have execution times less than or equal to the execution times of the corresponding jobs in $J'$. A scheduling algorithm $\mathcal{A}$ is **sustainable with respect to the execution requirements** if, when applied independently on $J$ and $J'$, a job in $J$ finishes execution before or at the same time as the corresponding job in $J'$.

- The sustainability property (with respect to the execution requirements) is important in the design of real-time systems since it allows the designers to focus on the worst-case execution times (WCET).

# Sustainability III

- Fortunately, FJP schedulers are sustainable (with respect to the execution requirements)

## Theorem 12 ([Cucu-Grosjean and Goossens, 2010])

*FJP schedulers are sustainable (with respect to the execution requirements) on unrelated platforms.*

# Periodic Implicit-Deadline Systems — Necessary and Sufficient Condition I

## Theorem 13

*Any Periodic Implicit-Deadline System is feasible iff*

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

The argument used for feasibility under those conditions is to consider the "shared processor"schedule adapted to the multiprocessor context. It can be obtained by partitioning the timeline into infinitesimal slots and by scheduling a task $\tau_i$ for a fraction proportional to $U(\tau_i)$ in each slot.

# Introduction to PFAIR I

- We will introduce the general notion of fairness that is the cornerstone of the class of PFAIR schedulers.
- Fairness requires us to schedule each task at a quasi-constant rate.
- For periodic tasks, each task $\tau_i$ progresses approximately at a rate of $U(\tau_i)$, at least if we consider large intervals. In small intervals, the rate can vary a bit (we will formalize this aspect).
- For the particular case of synchronous implicit-deadline periodic tasks, PFAIR is an **optimal** strategy.

# PFAIR Scheduling

- We consider synchronous implicit-deadline periodic tasks and we assume a quantum-based scheduler. The period of time for which a process is allowed to run in a preemptive multitasking system is generally called the time slice, or quantum. The scheduler is called once every time slice to choose the next processes to run.
- The scheduler definition requires the formalization of the schedule: a function $\mathcal{S} : \tau \times \mathbb{Z} \mapsto \{0, 1\}$, where $\tau$ is a periodic task set.
- $\mathcal{S}(\tau_i, t) = 1$ means that $\tau_i$ is scheduled in the slice $[t, t + 1)$ and $\mathcal{S}(\tau_i, t) = 0$ means otherwise.
- In an ideal **fair** schedule, each task $\tau_i$ receives $U(\tau_i) \times t$ processor units during the interval $[0, t)$ (which implies that all deadlines are met). Notice that such a schedule is not necessarily possible using a quantum-based (i.e. discrete) timeline.

# 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 $\tau_i$ at time $t$, denoted $\text{lag}(\tau_i, t)$ is defined as follows:

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

# PFAIR Schedules I

## Definition 14 (PFAIR Schedule)

A schedule has the PFAIR property iff

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

- Informally speaking, Equation 2 requires that the difference with the ideal fair schedule is not larger than the unity at any time, or equivalently that $\tau_i$ must receive $\lfloor U(\tau_i) \cdot t \rfloor$ or $\lceil U(\tau_i) \cdot t \rceil$ in the interval $[0, t)$.

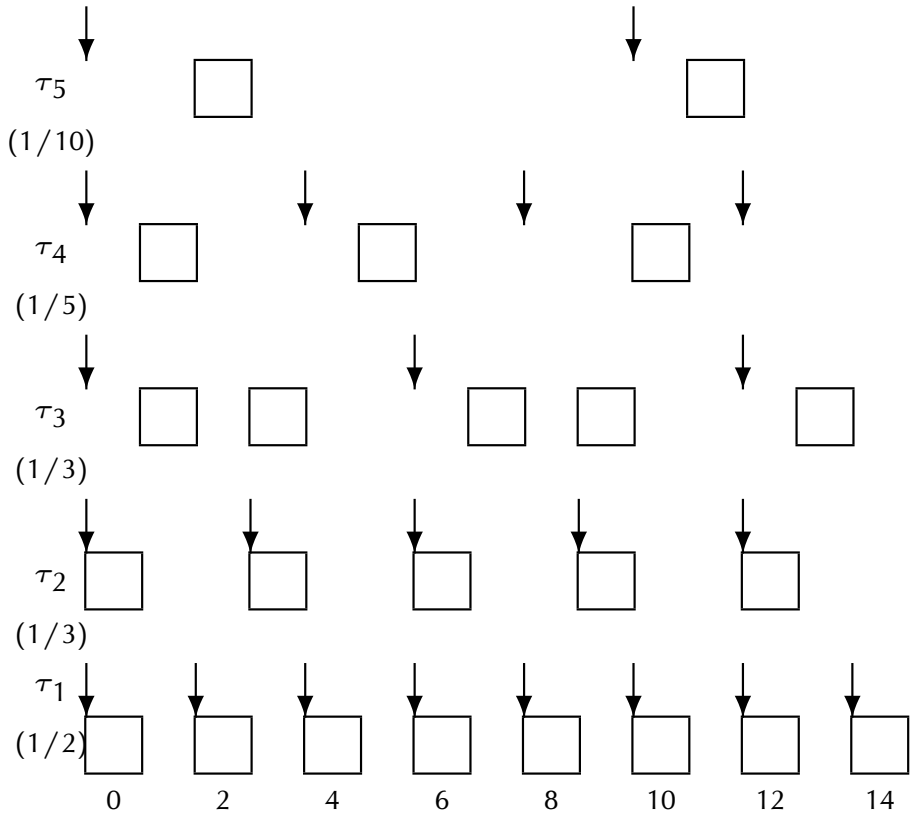
# PFAIR Schedules II

## Theorem 15

PFAIR*ness implies that all deadlines are met.*

*Proof.* Each periodic task $\tau_i$ must receive $C_i$ processor units in each interval $[\ell \cdot T_i, (\ell + 1) \cdot T_i)$, with $\ell \in \mathbb{N}$. At time $t = \ell \cdot T_i$, $U(\tau_i) \cdot t = (C_i/T_i) \cdot \ell \cdot T_i = \ell \cdot C_i$, a natural number. From Equation 2, we have that at time $t = \ell \cdot T_i$, the allocation of PFAIR corresponds to the ideal fair schedule. Since all deadlines are met in the ideal fair schedule, it is also the case for PFAIR. ∎

# PFAIR Example



# PFAIR Optimality I

## Theorem 16 ([Baruah et al., 1996])

*Let $\tau$ be a periodic synchronous implicit-deadline system. A* PFAIR *schedule exists for $\tau$ on $m$ processors iff*

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

# PFAIR Optimality II

- In other words, PFAIR schedules any schedulable system. Thus, PFAIR is **optimal**.
- Notice this does not contradict Theorem 7 (non-existence of online optimal schedulers) (Slide 25) 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 [Baruah et al., 1996], PD [Baruah et al., 1995] and PD$^2$ [Anderson and Srinivasan, 2000].

# Sporadic Arbitrary-Deadline Systems

- First, it is easy to see that Theorem 13 (Slide 42) can be adapted by replacing the utilization with the density to yield a **sufficient** test.

## Theorem 17

*The following condition is sufficient for feasibility of any sporadic arbitrary-deadline system:*

$$\lambda_{\text{sum}}(\tau) \leqslant m \qquad \textit{and} \qquad \lambda_{\text{max}}(\tau) \leqslant 1$$

# Global EDF and Sporadic Implicit-Deadline Systems

## Theorem 18 ([Srinivasan and Baruah, 2002])

*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 I

- 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 $\tau$:

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

- We can derive the following from Theorem 18:

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

- 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 II

## Definition 19 (EDF$^{(k)}$)

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

# EDF$^{(k)}$ Scheduling Algorithm III

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

# EDF$^{(k)}$ Scheduling Algorithm IV

## Theorem 20 ([Goossens et al., 2003])

*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 ($\pi_1$) for the jobs of $\tau_1$, $\pi_2$ for the jobs of $\tau_2, \ldots, \pi_{k-1}$ for the jobs of $\tau_{k-1}$ and uses EDF for the jobs of the others tasks $\tau_k, \ldots, \tau_n$ on additional processors $(\pi_k, \ldots, \pi_{k-1+\left\lceil \frac{U(\tau^{(k+1)})}{1-U(\tau_k)} \right\rceil})$

# EDF$^{(k)}$ Scheduling Algorithm V

## Corollary 21

*A sporadic implicit-deadline system $\tau$ 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 I

Consider the system $\tau$ 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 II

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

On the other hand, Theorem 18 cannot guarantee schedulability below $\left\lceil \frac{U(\tau)-U(\tau_1)}{1-U(\tau_1)} \right\rceil \approx \lceil 1.557/0.1 \rceil = \mathbf{16}$ processors using "pure" EDF.

# References I

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

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

[Baruah, 2007] Baruah, S. (2007).
Techniques for multiprocessor global schedulability analysis.
In *Real-Time Systems Symposium*, pages 119–128. IEEE Computer Society.

# References II

[Baruah and Burns, 2006] Baruah, S. and Burns, A. (2006).
Sustainable scheduling analysis.
In *27th IEEE International Real-Time Systems Symposium*, pages 159–168. IEEE Computer Society.

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

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

# References III

[Cucu-Grosjean and Goossens, 2010] Cucu-Grosjean, L. and Goossens, J. (2010).
Predictability of fixed-job priority schedulers on heterogeneous multiprocessor real-time systems.
*Information Processing Letters*, 110:399–402.

[Fisher et al., 2010] Fisher, N., Goossens, J., and Baruah, S. (2010).
Optimal online multiprocessor scheduling of sporadic real-time tasks is impossible.
*Real-Time Systems: The International Journal of Time-Critical Computing*, 45(1–2):26–71.

# References IV

[Goossens et al., 2003] Goossens, J., Funk, S., and Baruah, S. (2003).
Priority-driven scheduling of periodic task systems on multiprocessors.
*Real-Time Systems: The International Journal of Time-Critical Computing*, 25:187–205.

[Hong and Leung, 1988] Hong, K. and Leung, J. (1988).
On-line scheduling of real-time tasks.
In *Real-Time Systems Symposium*.

[Srinivasan and Baruah, 2002] Srinivasan, A. and Baruah, S. K. (2002).
Deadline-based scheduling of periodic task systems on multiprocessors.
*Information Processing Letters*, 84:93–98.