Real-time task scheduling on multiprocessors

Real-Time Industrial Systems

Marcello Cinque

Roadmap

- Real-time Multiprocessing: rationale
- Definitions and assumptions
- Partitioned scheduling
- Global scheduling

- References:
  - S. Baruah, M. Bertogna, G. Buttazzo. “Multiprocessor Scheduling for Real-Time Systems”, Springer, 2015
    - Chapters 1 to 9

The Multicore chip revolution

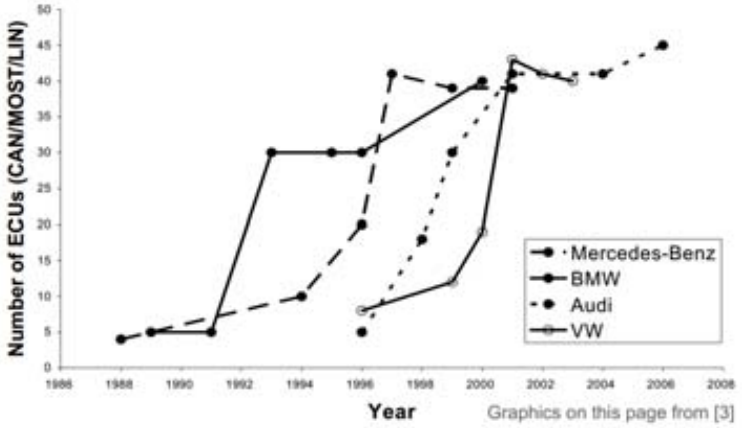
- On May 17th, 2004, Intel, the world's largest chip maker, canceled the development of the Tejas processor, Intel's successor to the Pentium4-style Prescott processor, due to extremely high power consumption
- After more than three decades of single core processors, the company decided two switch to multi core processors, following the competitor AMD.
- The retirement of the Pentium brand was marked by the official release of the first wave of Intel Core Duo processors, on July 27, 2006.
- What were the reasons of this choice?

Moving to multicore

- A consequence to Moore's law
  - the density of transistors on a chip doubles every 24 months
- More (and small) transistors do not necessarily mean higher working frequency, which is limited by physical and thermal constraints
  - If processor performance would have improved by increasing the clock frequency, the chip temperature would have reached levels beyond the capability of current cooling systems
- The solution followed by all major hardware producers to keep the Moore's law alive exploited a higher number of slower logic gates, building parallel devices made with denser chips that work at lower clock speeds

Multicore and real-time

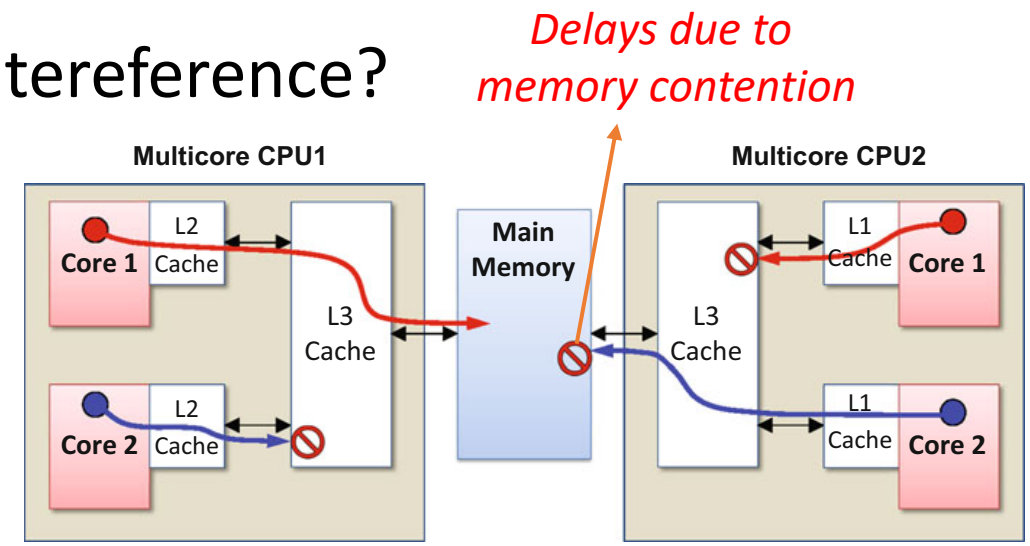
- The increasing use multicores in industrial real-time systems is pushed by several stakeholders
  - Chip vendors, delivering more and more powerful and cheap devices at scale
  - Industries and integrators, aiming to consolidate more workload and applications with different requirements and constraints on a smaller number of devices (e.g., 10 multiprocessors instead of 100 ECUs in modern cars)
    - Simplify deployment and evolution
    - Reduce overall power consumption
    - Reduce TCO
- However, the efficient exploitation of multicore platforms poses a number of new problems for real-time systems



Electronic Control Unit inside the cars

Multicore and real-time issues

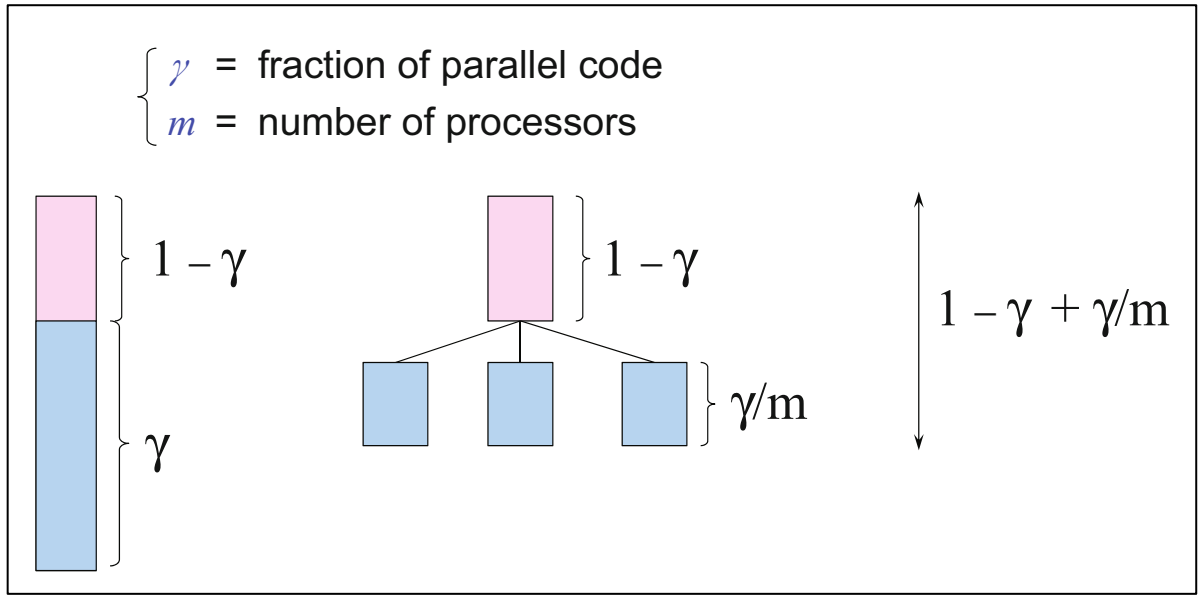
- How to split the executable code into parallel segments that can be executed simultaneously?
- How to allocate such segments to the different cores, assuring all temporal constraints?
- And considering the new sources of intereference?
  - An experimental study carried out at Lockheed Martin on an 8-core platform shows that **the WCET of a task can increase up to 6 times** when the same code is executed on the same platform when an increasing number of cores are active



Multicore and real-time issues

- The performance of an application can vary significantly depending on how tasks are allocated and scheduled on the various cores
  - How to allocate and schedule concurrent tasks in a multicore platform?
  - How to analyze real-time applications to guarantee timing constraints, taking into account communication delays and interference?
  - How to optimize resources (e.g., minimizing the number of active cores under a set of constraints)? (energy-aware scheduling)
  - How to reduce interference?
  - How to simplify software portability?
- Too many problems! :)
- We will mainly focus on multicore CPU scheduling, and on algorithms capable to provide latency bounds and feasibility tests

And, anyway, bear in mind the Amdhal law!



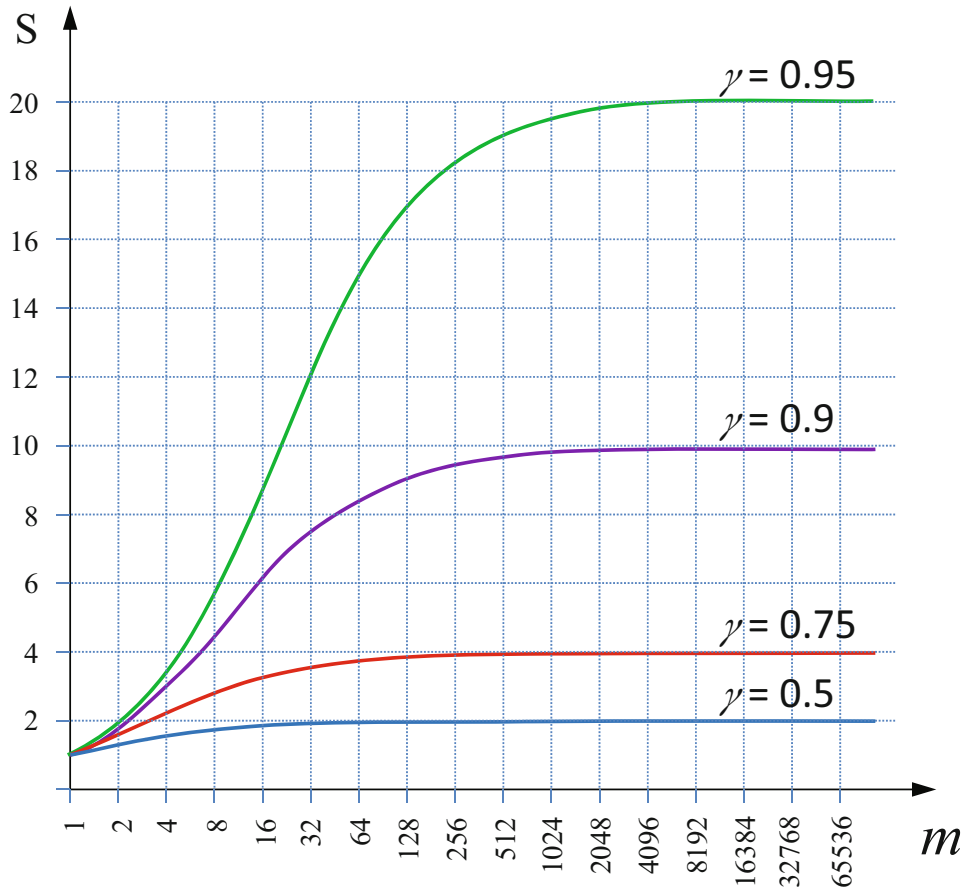
Speedup:

$$S(m,\gamma) = \frac{R_1}{R_m} = \frac{1}{1-\gamma+\gamma/m}$$

- A program in which only 50 percent of the code can be parallelized, even when running on a platform with 100 cores (m = 100), the theoretical speedup with respect to a single core is only S = 2!

Theoretical limits of parallel execution

$$S_{m\to\infty}(\gamma) = \frac{1}{1-\gamma}$$



- Even if the majority of code can be parallelized, there will always be (even implicit) sources of competition that require sequential execution
  - I/O operations
  - Distribution of input data
  - Collection of results
  - Memory and bus conflicts
  - Communication costs
  - …

Sporadic task models taxonomy

Nella letteratura il Modello LL è considerato come uno Sporadic Model, ovvero intendendo il periodo come un Delta minimo prima dell'arrivo del prossimo task, minimum inter-arrival time

- The Liu and Layland (LL) task model
  - A Liu and Layland task denoted τ_i is represented by a *worst-case execution requirement* (WCET) C_i and a *period* (or *inter-arrival separation*) T_i: $\tau_i = (C_i, T_i)$
  - The first job may arrive at any instant, and the arrival times of any two successive jobs are at least T_i time units apart
- The Three-parameter sporadic task model
  - Adds the notion of relative deadline D_i to each task
  - Depending on the relationship between D_i and T_i, the following classes are defined:
    - ***implicit-deadline*** sporadic task system: the relative deadline of each task in the set τ is equal to the task's period: $D_i = T_i$ for all $\tau_i \in \tau$ (exactly the LL system model)
    - ***constrained-deadline*** sporadic task system, the relative deadline of each task in the set τ is no larger than the task's period: $D_i \leq T_i$ for all $\tau_i \in \tau$
    - ***arbitrary-deadline*** sporadic task system do not need to have their relative deadlines satisfy any constraint with regards to their periods

Utilization bound on m-processors:

- Given a scheduling algorithm A for scheduling implicit-deadline sporadic task systems and a platform consisting of m unit-speed processors, the ***utilization bound*** of algorithm A on the m-processor platform is defined to be the largest number U such that all task systems with utilization ≤ U (and with each task having utilization ≤ 1) is successfully scheduled by A to meet all deadlines on the m-processor platform

- Where, in general, the utilization of a task τ_i is defined as $u_i = C_i/T_i$ and the total and maximum utilization of a task system τ are defined as:

$$U_{\text{sum}}(\tau) \stackrel{\text{def}}{=} \sum_{\tau_i \in \tau} u_i; \qquad U_{\text{max}}(\tau) \stackrel{\text{def}}{=} \max_{\tau_i \in \tau} (u_i)$$

Multiprocessor platforms

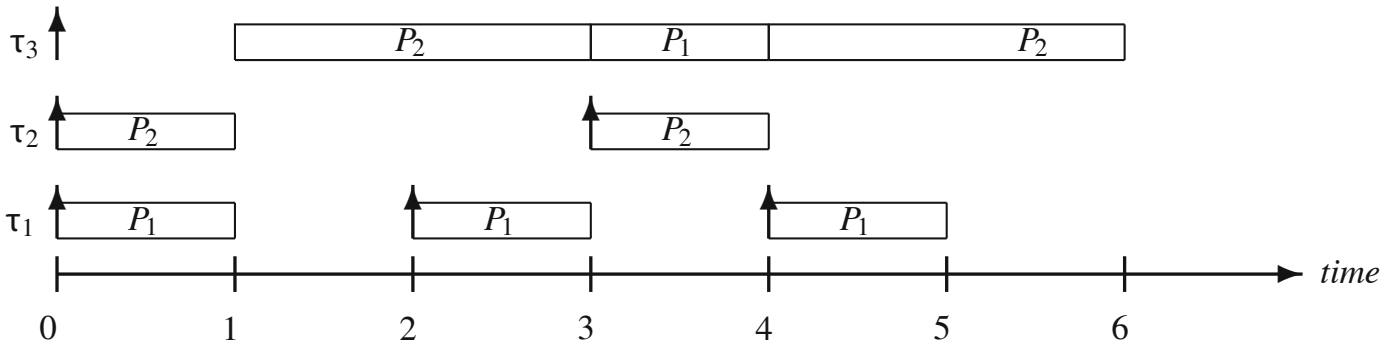
- Symmetric multiprocessing (SMP)
  - Classical and most diffused multiprocessors where all processors have the same computing capabilities
    - In the following, we will focus on SMP
- Asymmetric multiprocessing (AMP)
  - Increasing trend in industry to adopt chips with heterogenous (unrelated) cores, including traditional multicore CPUs, real-time processing units (RPUs), graphical processing units (GPUs), reconfigurable hardware (FPGAs), etc.
    - Will treat them later in the course

Our focus and assumptions

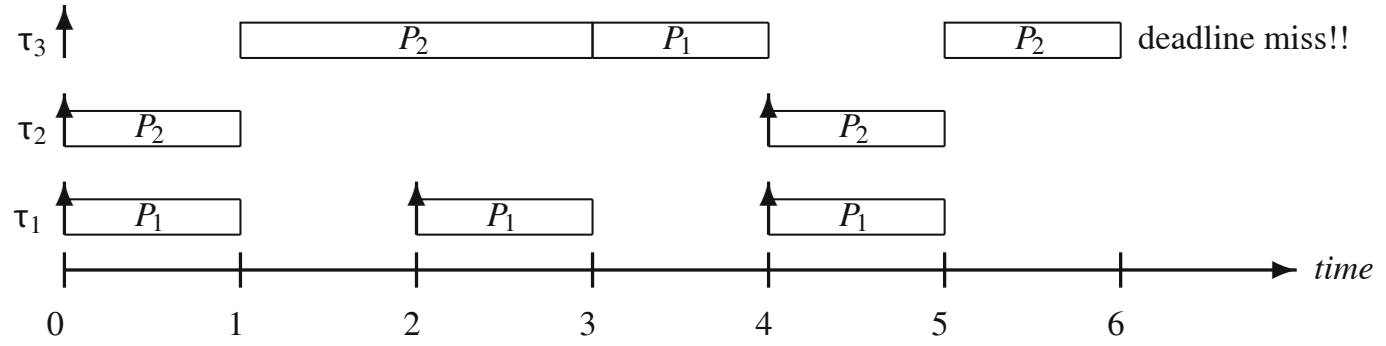
- Focus on CPU scheduling for SMP platforms
- considering independent implicit-deadline tasks (LL model)
  - In general, sporadic task models allow to achieve problems for which there exist tractable solutions (polynomial runtime)
    - Violating these models results in scheduling analysis problems that are highly intractable (nondeterministic polynomial time (NP)-hard)
  - And they are also baked by practical application
- and ignoring preemption, communication and migration costs
  - Priority-driven algorithms possess the pleasing property that the number of preemptions and migrations they generate is strictly less than the number of jobs they schedule
  - Hence, one can account for the overhead cost of preemptions and migrations by simply inflating the WCET parameters by the worst-case cost of one preemption and one migration

Why sporadic and not periodic?

Three periodic tasks $\tau_1 = (1, 1, 2), \tau_2 = (1, 1, 3), \tau_3 = (5, 6, 6)$
on two processors P_1 and P_2



The same example with sporadic tasks and the second job of τ_2 arriving at 4 instead of 3



- On single-processors, the periodic correspondent behavior of a sporadic task system is also the worst case
- This is not the case anymore in multiprocessors!

In pratica a differenza del caso del singol processor i task periodici rappresentavo il worst case rispetto a quello sporadici.

In sistemi SMP quello che è fesaibile con task periodici non è detto che lo sia anche con task sporadici, come nell'esempio di fianco.
Sporadi task are the Worst Case in SMP

Classification of scheduling algorithms by priority assignments

- ***Fixed task priority (FTP)*** scheduling: each sporadic task is assigned a unique priority, and each job inherits the priority of the task that generated it (examples: RM and DM) job= single instance of an aperiodic task
- ***Fixed job priority (FJP)*** scheduling: different jobs of the same task may be assigned different priorities. However, the priority of a job, once assigned, cannot change (example: EDF)
- ***Dynamic priority (DP)*** scheduling: there are no restrictions on the manner in which priorities are assigned to jobs—the priority of a job may change arbitrarily often between its release time and its completion (example: pfair)

Classification of scheduling algorithms with respect to task-to-processor assignment

- ***Partitioned*** scheduling: individual tasks are restricted to executing only upon a single processor; once assigned they are not migrated
  - Good for processor affinity but bad for optimality

- ***Global*** scheduling: task may execute upon different processors at different points in time
  - Achieves better global perfomance, but requires task migrations
  - Requires the simplifying assumption that interprocessor communication incurs no cost or delay. This is a significant short-coming of much of the existing body of research into multiprocessor scheduling theory today

What's next?

- Partitioned scheduling of LL tasks
  - With FJP and FTP algorithms

FJP= Fixed Job Priority
FTP= Fixed Task Priority

- Global scheduling of LL tasks
  - With DP, FJP and FTP algorithms

Partitioned scheduling of LL tasks

- It is generally known that finding an optimal task partitioning is equivalent to the *bin-packing problem*, and so it is highly intractable: nondeterministic polynomial time (NP)-hard in the strong sense
- Approximation algorithms are used, having polynomial runtime and for which utilization bounds can be found
- Most polynomial algorithms have a common structure:
  - They specify an order to consider tasks and processors
  - They try to “fit” tasks to processors, in the established order

Partitioned EDF scheduling
(partitioned FJP)

- With EDF, a LL task “fits” a processor if the task’s utilization does not exceed the processor capacity minus the sum of the utilizations of all tasks previously allocated to the processor
- The algorithm declares:
  - **success** if all tasks are successfully allocated
  - otherwise, it declares **failure**
- A *reasonable allocation (RA)* algorithm is defined as one that fails to allocate a task to a multiprocessor platform only when the task does not fit into any processor upon the platform

Processor allocation heuristics

- ***First-fit (FF)***: the processors are considered ordered in some manner and the task is assigned to the first processor on which it fits

- ***Worst-fit (WF)***: the task is assigned to the processor with the maximum remaining capacity

- ***Best-fit (BF)***: the is assigned to the processor with the minimum remaining capacity *exceeding its own utilization* (i.e., on which it fits)

Task ordering heuristics

- ***Decreasing (D)***: the tasks are considered in nonincreasing order of their utilizations

- ***Increasing (I)***: the tasks are considered in nondecreasing order of their utilizations

- ***Unordered (ε)***: the tasks are considered in arbitrary order

Heuristics combinations

$$\{\mathbf{FF}, \mathbf{BF}, \mathbf{WF}\} \times \{\mathbf{D}, \mathbf{I}, \varepsilon\}$$

- Algorithms are referred with two or three letter acronyms
  - FFD: first fit decreasing
  - WF: worst fit with arbitrary order

Combinando le tecniche di allocazione e le euristiche di ordinamento

Lower Utilization bound

- Given a task set with n tasks to be scheduled on m processors, let α be an upper bound on the per-task utilization and $\beta = \lfloor 1/\alpha \rfloor$
- Any reasonable algorithm has a utilization bound no smaller than:

$$m - (m-1)\alpha$$

- This can be demonstrated considering that, for any task with utilization u_i that cannot be assigned to any processor, it must be:

$$m(1-u_i) + u_i = m - (m-1)u_i \geq m - (m-1)\alpha$$

Not feasible

Upper utilization bound

- No allocation algorithm, reasonable or not, can have a utilization bound larger than:

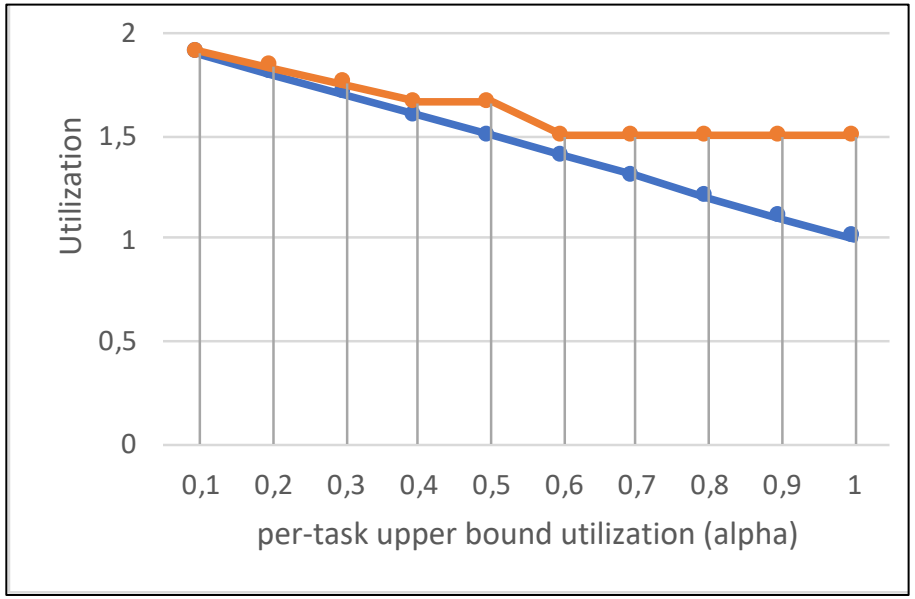
$$\frac{\beta m + 1}{\beta + 1}$$

Beta ci da un approssimazione del Load del sistema:
- Se B<1 significa che tutti i stask hanno un individual utilizzation (Aplah) che è minore di 0.5 quindi abbiamo dei light tasks
-Se B=1 significa che Alpha>0.5 quindi abbiamo dei tasks pesanti

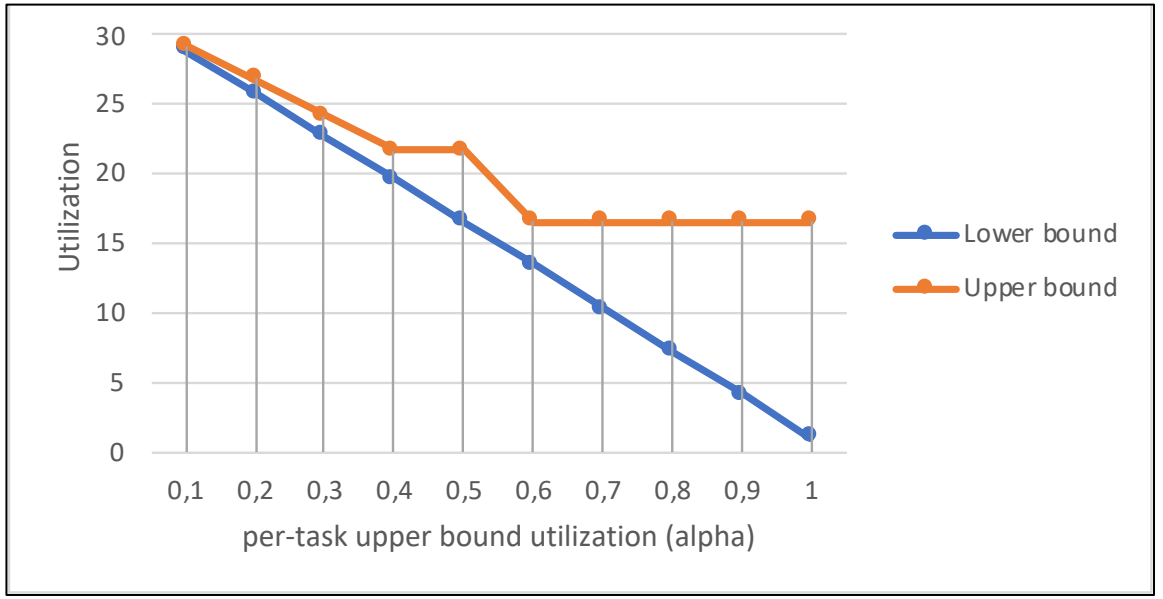
- For arbitrary α (in any case smaller than 1), it can be shown that:
  - WF and WFI have a utilization bound of 1, regardless of *m*
  - The others (FF, FFD, FFI, BF, BFI, BFD, WFD) have a bound $(m + 1)/2$
- Bounds are useful for design considerations, but, given the simplicity of allocation algorithms, they are actually tried out directly

Bounds examples

On a dual core



On a 32 core



- As intuition suggests, task sets with heavy tasks in terms of utilization reduce the overall efficacy of use of the multiprocessor

Partitioned Rate Monotonic Scheduling
(partitioned FTP)

- Tasks are allocated to processors and then scheduled with RM
- Several bounds achieved extending the classical LL uniprocessor bound
  - A simple bound (Oh and Baker): Any implicit-deadline sporadic task system is schedulable with FFD allocation and RM local scheduling on m processors if:

$$U < m(\sqrt{2}-1)$$

Rad2 -1 stend for 40% of utilization

  - A refined bound (Diaz and Garcia): Any implicit-deadline sporadic task system τ of n tasks is schedulable with FFD allocation and RM local scheduling on m processors if:

$$U_{sum}(\tau) < (m-1)\left(\sqrt{2}-1\right) + (n-m+1)(2^{\frac{1}{n-m+1}}-1)$$

Partitioned Scheduling bounds

	Rate Monotonic		EDF
# processors	Oh and Baker bound	Diaz and Garcia bound (n = 10)	(beta = 1)
2	0,83	1,13	1,5
4	1,66	1,97	2,5
8	3,31	3,68	4,5
16	6,63	6,86	8,5
32	13,25	13,52	16,5
64	26,51	26,78	32,5

Global scheduling of LL Tasks

- With global scheduling, migration is allowed.
- Theoretically, Horn established that
  - *"Any implicit-deadline sporadic task system τ satisfying*

$$U_{sum}(\tau) \leq m \text{ and } U_{\max}(\tau) \leq 1$$

is feasible upon a platform of m unit-capacity processors"

but HOW?

- This can be achieved by "processor sharing" approaches, in which jobs can execute at any time instant on different processors, to receive the required u_i fraction of processor capacity each period
- A powerful and efficient implementation of this principle is *pfair scheduling*, that performs optimal global scheduling of LL tasks

Unfeasibility

- A consequence of the Horn theorem is that any task system for which

$$U_{\text{sum}}(\tau) > m \text{ or } U_{\text{max}}(\tau) > 1$$

cannot be feasible on a platform with m unit-capacity processors

Pfair scheduling
(global DP)

- Tasks are explicitly required to make progress at *steady rates*
- The temporal axis is divided in fixed slots, each with a conventional *unit* duration
- Slots are integer and numbered (0, 1, 2, …)
- Task jobs are broken in smaller *subjobs*, each with execution time 1
- Each subjob must execute in a window of time, which end is called *pseudo-deadline*
- The allocation of subjobs is done according to a “fair share” to guarantee “proportionate progress” of all the tasks:
  - A job with utilization u_i released at time t_0, in a given interval $[t_0, t)$ receives exactly $\lfloor (t - t_0)u_i \rfloor \; or \; \lceil (t - t_0)u_i \rceil$ units of execution, for any integer t

like paging the CPU

Esempio: Se ho Utilization ui=0.5 ed ho bisogno di 7 slot, per ogni t ricevo 3 o 4 slot

Pfairness

- Formally, a schedule is *pfair* if and only if, for any job of the task τ_i, released at t_0 and with utilization u_i and any time interval $[t_0, t)$, with $t \in \mathrm{N} \wedge t < T_i$, the difference (lag) between $(t - t_0)u_i$ and the number of slots allocated to τ_i in $[t_0, t)$ is:

$$-1 < lag < 1$$

- In other terms, *pfairness* requires that the absolute value of the difference between the expected allocation and the actual allocation to every task always be strictly less than 1
  - A task never gets an entire slot ahead or behind
  - By construction, pfariness also guarantees execution within deadlines
- It can be demonstrated that for any task set satisfying the Horn theorem, a pfair schedule exists

Pfair scheduling algorithm

- Definitions:
  - A task is *contending* for the slot t if its subjob can be allocated to slot t without violating pfairness
  - A task is *urgent* in the slot t when, it its subjob is not allocated to slot t it will violate pfaireness
  - The *pseudo-deadline* is the latest slot by which the next subjob needs to be allocated to not violate pfairness
- The pfair algorithm schedules, during each time slot, the (up to) m contending tasks with greater priority, where the priority is evaluated according to urgency and pseudo-deadlines

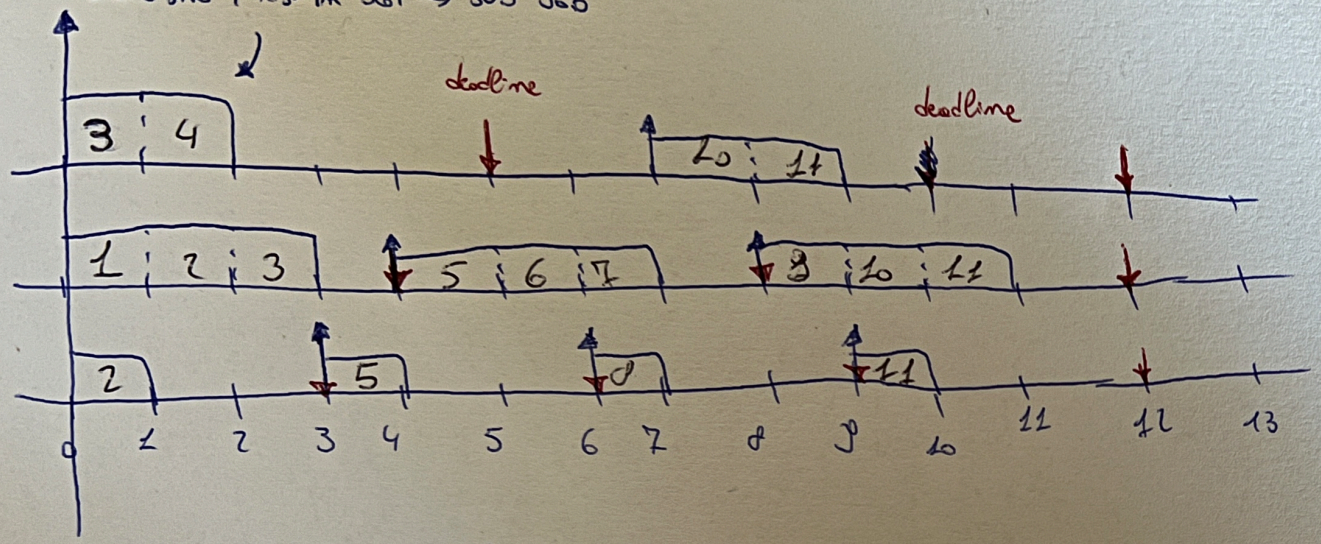
Pfair: considerations

- Pfair is a DP (dynamic priority) scheduling because each job, during its execution, can have different priorities (depending on the pseudo-deadlines of its subjobs)
- Albeit being optimal, breaking tasks into unit-time subjobs can generate a high number of preemptions, context-switches, and migrations
  - For some applications, the introduced overhead can be an issue

ESEMPIO PFAIR

$\tau_1 \Rightarrow C_1 = 2 \quad T_1 = 5$ Supponiamo arrivo a 0 e a 7 $\quad \tau_2 \Rightarrow C_2 = 3 \quad T_3 = 4 \quad \tau_3 \Rightarrow C_3 = 1 \quad T_3 = 3$

dividiamo i task in Slot → sub Job



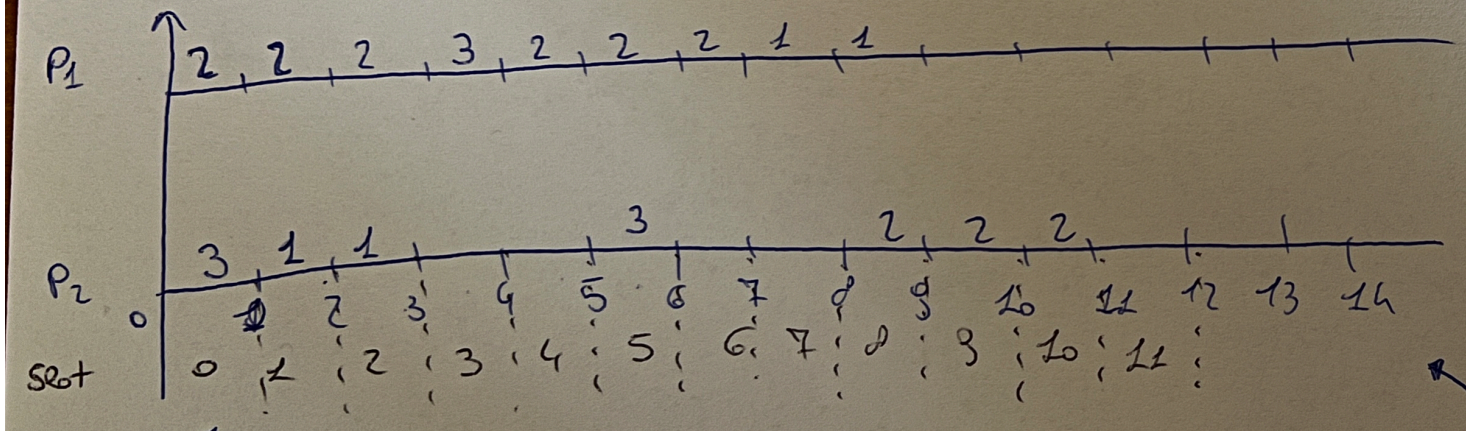
$U = \frac{2}{5} + \frac{3}{4} + \frac{1}{3} \simeq 1,48$ allora non fattibile su un Singol Core

τ_1 diviso in 2 sub-Job

τ_2 diviso in 3 sub-Job

τ_3 diviso in 1 sub-Job

Vediamo PFair su due processori (dividiamo in Tanne Slot unitari per semplificazione



quali Task τ_i assegniamo ad ogni Slot sui due processori

Dobbiamo assegnare ad ogni Sub-Job una Sub-Deadline che deve essere l'ultimo Slot da assegnare al sub-Job per non violare la sua sub-Deadline

Esempio: Sub-Job 2 di τ_1 → deadline 4
Sub-Job 2 di τ_1 → deadline 5
quindi: il sub Job 2 deve essere schedulato entro 3 mentre sub Job 2 entro 4

Sia τ_1, τ_2 che τ_3 passano da un processore ad un altro almeno una volta

Global EDF scheduling
(global FJP)

- Jobs are scheduled on the first available processor, ordered according to deadlines
- Preemptions (and, if necessary, migrations) can occur, but they are strictly less than the number of jobs to be scheduled

- … however, this simple extension of EDF to multiprocessors has a very poor utilization bound …

Dhall effect

- Consider a set of m+1 LL with tasks to be scheduled on a m-processor
- Task parameters are:
  - $C_i = 1$ and $T_i = x$ for $i = 1..m$
  - $C_{m+1} = T_{m+1} = x+1$ 100% utilization per l' m+1 esimo task
- The utilization of the system is $m/x + 1$, which, for any m, tends to 1 as x increases
- If jobs are to be released simultaneously, the first m tasks would be scheduled first, causing the $(m+1)$-th task to miss its deadline

Dhall effect

The utilization bound of global EDF is arbitrarily close to 1, regardless of the number of processors

Global EDF: a general bound

- A general bound for global EDF can be established if the maximum utilization of the task system is known, specifically:

$$U_{sum}(\tau) \leq m - (m-1)U_{max}(\tau)$$

- Note that, for $U_{max}(\tau) \rightarrow 1$, this just confirms the Dhall effect
- In other terms, global EDF is not optimal. However, a total utilization smaller than m is enough to guarantee that non real-time tasks are not starved and that real-time tasks have a bounded latency (even if not sufficient to meet all deadlines, unless the condition above is verified)

A note about SCHED_DEADLINE on multiprocessors

- SCHED_DEADLINE is a global EDF scheduler, so all the considerations on EDF (including the Dhall effect) apply to SCHED_DEADLINE

- If necessary, partitioned SCHED_DEADLINE can be obtained by setting affinities to tasks (with `cpuset` on Linux), that is, pinning the execution of tasks on specific CPUs

Circumventing the Dhall effect

- It can be demonstrated that any implicit-deadline sporadic task system τ such that

$$U_{sum}(\tau) \leq (m+1)/2$$
$$U_{max}(\tau) \leq 1/2$$

  is correctly scheduled to meet all deadlines on m processors by EDF

- In other terms, if there are no “heavy” tasks (with utilization above 50%), a better bound can be established for EDF → $(m+1)/2$
- And if we have heavy tasks?

fpEDF Fixed Priority EDF

- Tries to pragmatically manage task systems with “heavy” tasks

- The idea is to assign a fixed high priority to the first m-1 heavy tasks first (so that they can be assigned to m-1 processors in the worst case) and then schedule the rest with EDF
  - Practically, such tasks receive a $-\infty$ deadline and get scheduled on the first m-1 processors

fpEDF Algorithm

Implicit-deadline sporadic task system $\tau = \{\tau_1, \tau_2, \ldots \tau_n\}$ to be scheduled on m processors
(It is assumed that $u_i \geq u_{i+1}$ for all i, $1 \leq i < n$) In questo caso allora umax=u1

for $i = 1$ to $(m-1)$ do
    if $(u_i > \frac{1}{2})$
        then τ_i's jobs are assigned highest priority (ties broken arbitrarily)
        else break
the remaining tasks' jobs are assigned priorities according to EDF

fpEDF bounds

- It can be demonstrated that fpEDF on a m-processor has a utilization bound no smaller than

$$\frac{m+1}{2}$$

- As done with partitioned scheduling, better bounds can be established if $U_{max}(\tau)$ is known
- In particular, fpEDF correctly schedules on a m-processor any implicit-deadline sporadic task system τ satisfying:

$$U_{sum}(\tau) \leq \max\left(m - (m-1)U_{max}(\tau), \frac{m}{2} + U_{max}(\tau)\right)$$

Global Rate Monotonic Scheduling
(global FTP)

- Tasks are assigned a fixed priority according to the RM rule, and are then scheduled on the first available processor
- It can be shown that
  - RM is not optimal on multiprocessors
  - It suffers the Dhall effect, as seen for EDF
- Better bounds can be established (circumnavigating the Dhall effect)
  - Assuming a “light” workload (RM-light)
  - Or allocating “heavy” tasks first, similar to pfEDF (RM-US)

RM-light

- An implicit-deadline sporadic task system τ is said to be RM-light(m) if:

$$U_{max}(\tau) \leq m/(3m-2) \text{ and}$$
$$U_{sum}(\tau) \leq m^2(3m-2)$$

- Any RM-light(m) implicit-deadline sporadic task system is feasible on a m-processor platform

RM-US(ξ)

Implicit-deadline sporadic task system $\tau = \{\tau_1, \tau_2, \ldots \tau_n\}$ to be scheduled on m processors
(It is assumed that $u_i \geq u_{i+1}$ for all i, $1 \leq i < n$)

for $i = 1$ to $(m-1)$ do
  if $(u_i > \xi)$
    then τ_i is assigned highest priority
    else break
the remaining tasks are assigned priorities according to RM

Psi parameter

- Algorithm RM-US($m^2(3m-2)$) has utilization bound no smaller than $m^2(3m-2)$ on a m unit-speed processor

Tutto quello visto in questo gruppo di slide non tiene conto della contesa della Memoria (cache e memoria secondaria) e semplifica considerando migrazione istantanee tra un Processore ed un altro