

Real Time Scheduling

Jean-Luc Scharbarg - ENSEEIHT - Dpt. SN

September 2024

Main goals of the module

- A general overview of how real-time applications can be executed
 - ▶ Real-time scheduling
 - ★ Why is it different from non real-time scheduling?
 - ★ What are the features that have to be considered?
 - ★ What are the real-time scheduling algorithms?
 - ★ How can we be sure that it works?
 - ★ ...
 - ▶ Real-time operating system
 - ★ Illustration on OSEK

A definition of real-time

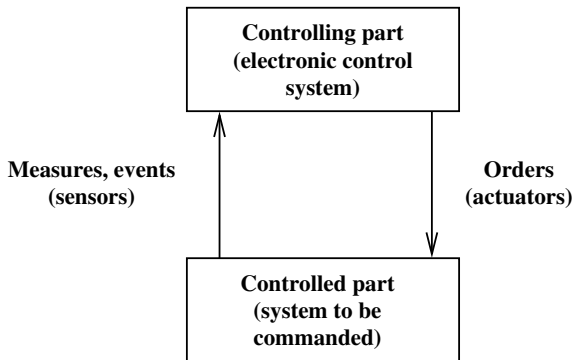
A real-time system is able, first to read all incoming data before they become useless, second to give an appropriate timely reaction

- Temporal correctness : correct result after the deadline = bad (useless) result
- The delay highly depends on the application
- \Rightarrow real-time is not real fast
- Two main classes of real-time systems :
 - ▶ Hard real-time : a missed deadline can have catastrophic consequences
 \Rightarrow strict timing constraints
 - ▶ Soft real-time : some (not frequent) missed deadlines can be tolerated
(\Rightarrow often a predefined rate of allowed missed deadlines)
- Mandatory to guarantee the functional behavior as well as the temporal behavior of the system

Application domains

- Factory automation
 - ▶ Control/command of a production line
 - ▶ Control process
 - ▶ ...
- Embedded systems
 - ▶ Avionics (flight control, ...)
 - ▶ Automotive
 - ▶ Space
 - ▶ Robotics
 - ▶ Home appliance
 - ▶ Phones
 - ▶ ...

Architecture overview



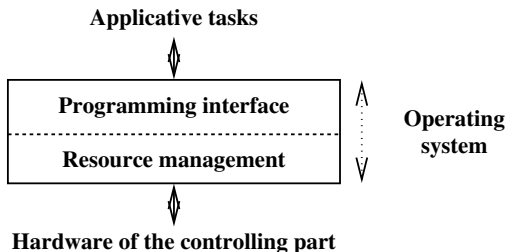
- Inputs and outputs are very important
- Hardware architecture of the controlling part : programmable logic controller (Schneider Electric, Siemens), ad hoc circuit, FPGA, micro-controller, micro-processor
- The controlling part can be centralised or distributed \Rightarrow industrial networks, fieldbuses (CAN, FIP, ...)

Processing overview

```
For each event of the controlled part do  
    Read the data from the controlled part  
    Compute the new system state  
    Send the orders associated to this state  
EndFor
```

- Synchronous behavior (time driven) : data are read at predefined instants \Rightarrow periodic tasks
- Asynchronous behavior (event driven) : immediate answer to events \Rightarrow non periodic tasks
- Hybrid behavior : mix of periodic and aperiodic tasks

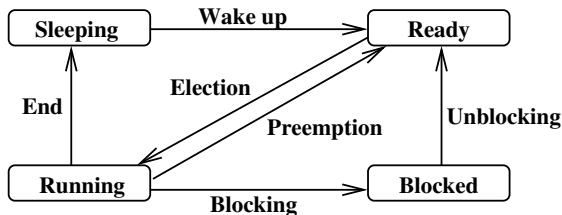
Software architecture



- More or less elaborated operating system, from nothing to a multitask operating system with graphical facilities
- Multitask operating system:
 - ▶ Scheduler to manage processor sharing
 - ▶ Control access mechanisms in order to share the other resources
 - ▶ Synchronisation and communication mechanisms in order to manage global consistency

Scheduler

- State diagram of a task



- Scheduler execution when an interrupt occurs:
 - ▶ The context of running task is saved and this task moves to ready state
 - ▶ One ready task is elected
 - ▶ The context of this elected task is restored
- Scheduling algorithm: activation time of the scheduler + election policy

Operating system

- General purpose operating system:
 - ▶ Goals:
 - ★ optimisation of resource utilisation
 - ★ minimisation of average response time
 - ▶ Features:
 - ★ round robin scheduler
 - ★ complex memory hierarchy
 - ▶ Examples : Linux, Solaris, Windows xx
- Real-time operating system:
 - ▶ Goals:
 - ★ predictable delays
 - ★ reliability
 - ▶ Features:
 - ★ priority based preemptive scheduling
 - ★ limited and controlled memory hierarchy
 - ★ execution of system primitives in bounded time
 - ▶ Examples: Tornado, LynxOS, QNX, RTAI, OSEK, TinyOS

Implementation of a real time system

- Cross-development \Rightarrow two machines:
 - ▶ host machine:
 - ★ application development and debug
 - ★ code generation for the target machine \Rightarrow cross-compilation
 - ★ this code is loaded on the target machine
 - ★ no host machine at run-time
 - ▶ target machine:
 - ★ application execution
- Native development \Rightarrow one single machine:
 - ▶ application development and debug
 - ▶ application execution
 - ▶ Parts of the operating system can be removed at run-time

A real-time application in this module

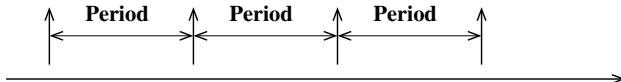
- A set of tasks (pieces of codes)
 - ▶ Can be fully independant
 - ▶ Can share resources, e.g. variables
 - ▶ Can communicate, synchronise, have precedence constraints
- We are only interested in the temporal correctness, not the fonctionnal one
- Each task is modelled by temporal values
- The set of tasks can be executed on a single core or on a multi-core or a multi-processor

Real-time tasks

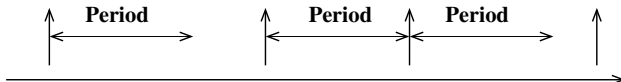
- A task: a recurrent execution of a job.

- Request can be

- ▶ Periodic: constant duration between consecutive releases



- ▶ Sporadic: variable duration between consecutive releases, with a known minimum

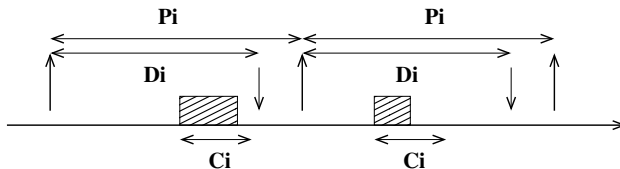


- ▶ Aperiodic: variable duration between consecutive releases.



Non concrete real-time task

- Defined by the following features
 - ▶ A Worst-Case Execution Time (WCET): C_i
 - ★ Safe upper bound on the execution time of one job of the task
 - ▶ A release period: P_i
 - ★ (Minimum) duration between consecutive job releases of the task
 - ▶ A relative deadline D_i
 - ★ Maximum allowed response time for each job of the task

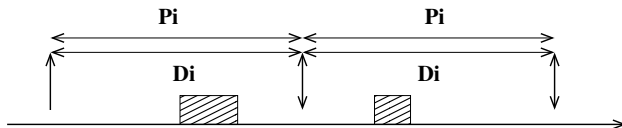


- The first release time of the task is unknown \Rightarrow non concrete task

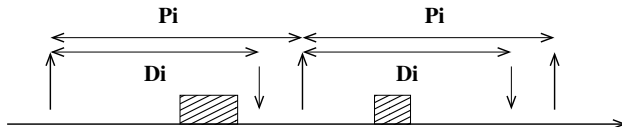
Non concrete real-time task

- Relationship between deadline and period

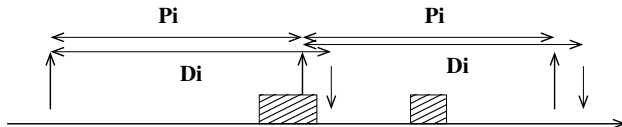
- ▶ Implicit deadline ($D_i = P_i$): the simplest case



- ▶ Constrained deadline ($D_i \leq P_i$): the intermediate case

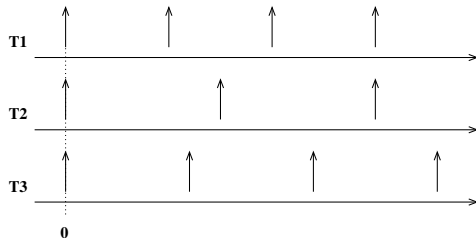


- ▶ Arbitrary deadline (D_i can be greater than P_i): the most complex case

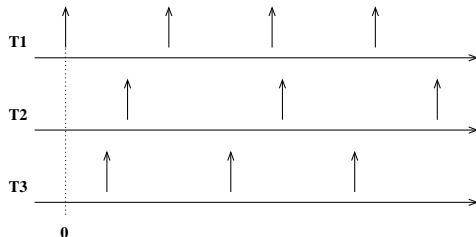


Concrete real-time task

- The first release time r_i of the task is known
- Synchronous system: all the tasks have the same first release time (0)



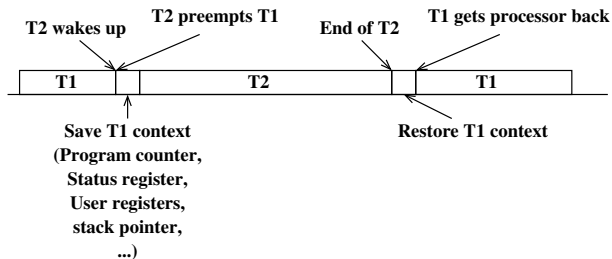
- Asynchronous system: tasks have different release times



Preemptive vs non preemptive tasks

- What is a preemption?

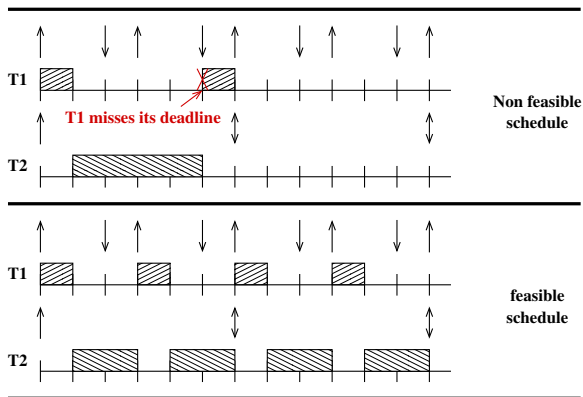
- ▶ the execution of a task is suspended at a point in its execution and will be resumed later at the same point
- ▶ Pros
 - ★ An urgent task can preempt a non-urgent time consuming one
- ▶ Cons
 - ★ context switch overhead



- ▶ Sometimes limited preemption: only possible at given point in the task

Real-time Scheduling

- Multitask system \Rightarrow tasks have to be scheduled
- Real time constraints (especially deadlines) have to be respected.
- Feasible schedule: the worst-case response time of every job of every task is not greater than the task's relative deadline
- Ex: T_1 (r_1 : 0, C_1 : 1, D_1 : 2, P_1 : 3), T_2 (r_2 : 0, C_2 : 4, D_2 : 6, P_2 : 6)

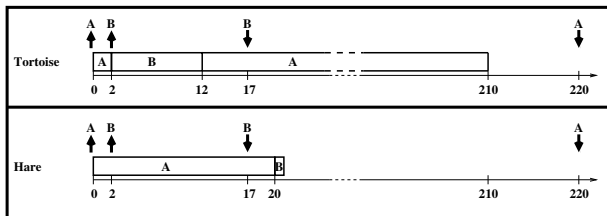


The tortoise and the hare (G. Le Lann)

- Two processors $Pr1$ (hare) et $Pr2$ (tortoise)
 - $Pr1$ 10 times faster than $Pr2$
 - $Pr1$: non preemptive scheduling *First Come First Served*
 - $Pr2$: preemptive scheduling *Earliest Deadline First*
- Two tasks

	C^{hare}	$C^{tortoise}$	Deadline	Release
A	20	200	220	0
B	1	10	15	2

- B respects its deadline on $P2$ but not on $P1$



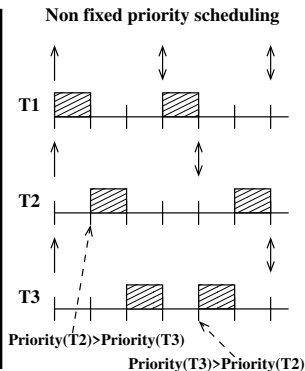
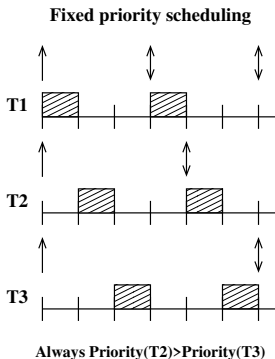
- Choosing the right scheduling policy is essential

Real-Time Scheduling

- Many work has been devoted to real-time scheduling problems
 - ▶ Scheduling policies
 - ▶ Feasibility tests
 - ▶ Complex task models
 - ▶ ...
- Feasibility problem is NP-hard, except for basic cases \Rightarrow
 - ▶ Exact test at exponential cost
 - ▶ Pessimistic test at polynomial or pseudo-polynomial cost
- Scheduling algorithms
 - ▶ On-line scheduling policies (priority driven)
 - ★ Knowledge of the active jobs only
 - ★ Simple scheduling policy in order to choose the highest priority ready job at specific instants (ends of job execution and/or job releases and/or ...)
 - ▶ Off-line scheduling policies (time driven)
 - ★ Knowledge of the past and the future
 - ★ off-line creation of a periodic feasible schedule, using meta-heuristics, branch and bound, ...

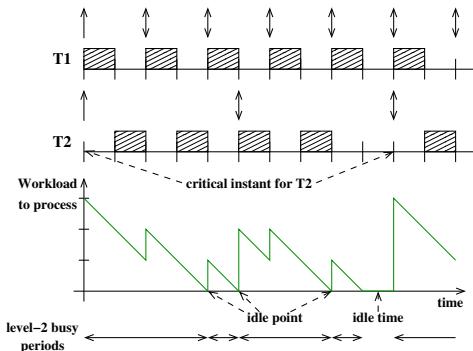
On-line fixed priority scheduling

- We mainly address on-line scheduling in this module
 - ▶ At any time, each ready job is assigned the priority of its task
- Task priorities are fixed off-line and don't change dynamically
- Ex: T_1 (r_1 : 0, C_1 : 1, D_1 : 3, P_1 : 3), T_2 (r_2 : 0, C_2 : 1, D_2 : 4, P_2 : 4), T_3 (r_3 : 0, C_3 : 2, D_3 : 6, P_3 : 6)



Critical instant and busy period

- We assume that tasks are ordered by priority level
 - ▶ Priority (T_1) > Priority (T_2) < Priority (T_3) > ...
- Critical instant for a task T_i : T_i is released simultaneously with all the higher priority tasks
- Level-i busy period: time period where the CPU is kept busy by tasks whose priority is higher or equal to priority of T_i
- Ex: T_1 (r_1 : 0, C_1 : 1, D_1 : 2, P_1 : 2), T_2 (r_2 : 0, C_2 : 2, D_2 : 5, P_2 : 5)



Results based on critical instant and busy period (1)

- A first theorem: the worst-case response time for a task T_i occurs during the longest level- i busy period
- A second theorem: the longest level- i busy period is initiated by the critical instant for task T_i
- Benefit of these two theorems: the worst-case is known for non concrete and synchronous task systems
 - ▶ build the first level- i busy period starting from the critical instant for T_i
 - ▶ claim the worst response time of T_i jobs in this busy period as the worst-case response time for T_i
- A busy period always ends iff the processor is not overloaded (its utilization ratio U doesn't exceed 1):

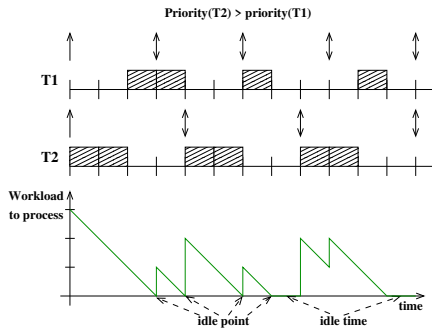
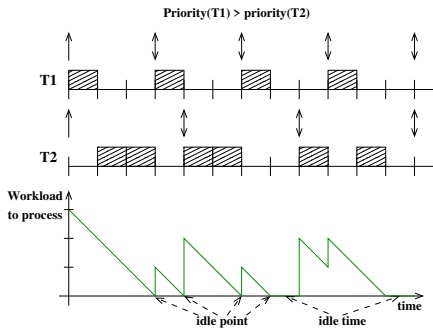
$$U = \sum_{i=1}^n \frac{C_i}{P_i} \leq 1$$

- ▶ Synchronous task system \Rightarrow idle time during the least common multiple of the periods ($LCM_{1 \leq i \leq n} T_i$)

$$LCM_{1 \leq i \leq n} T_i \times (1 - U)$$

Results based on critical instant and busy period (2)

- Idle times and instants are the same for any conservative scheduling algorithm



- Problem: building the level- i busy period time unit per time unit is exponential in time
 - When periods are co-prime numbers, we have

$$LCM_{1 \leq i \leq n} T_i = T_1 \times \dots \times T_n$$

Results based on critical instant and busy period (3)

- Specific case when $D_i \leq P_i$ and preemptive scheduling
 - ▶ If there are two jobs of T_i in a level- i busy period, then T_i misses its deadline \Rightarrow pseudo-polynomial test
- Find the smallest fixed point $R_i^{(*)}$ of the equation:

$$\begin{aligned}R_i^{(0)} &= C_i \\R_i^{(n+1)} &= C_i + \sum_{j \in hp(i)} \lceil \frac{R_i^{(n)}}{P_j} \rceil \times C_j\end{aligned}$$

- Ex: T_1 (r_1 : 0, C_1 : 1, D_1 : 2, P_1 : 2), T_2 (r_2 : 0, C_2 : 2, D_2 : 5, P_2 : 5)

$$\begin{aligned}R_2^{(0)} &= C_2 &= 2 \\R_2^{(1)} &= C_2 + \lceil \frac{R_2^{(0)}}{P_1} \rceil \times C_1 &= 2 + \lceil \frac{2}{2} \rceil \times 1 &= 3 \\R_2^{(2)} &= C_2 + \lceil \frac{R_2^{(1)}}{P_1} \rceil \times C_1 &= 2 + \lceil \frac{3}{2} \rceil \times 1 &= 4 \\R_2^{(3)} &= C_2 + \lceil \frac{R_2^{(2)}}{P_1} \rceil \times C_1 &= 2 + \lceil \frac{4}{2} \rceil \times 1 &= 4\end{aligned}$$

Results based on critical instant and busy period (4)

- What if $D_i > P_i$?

- ▶ $R_i^{(*)} \leq P_i \Rightarrow$ no difference with $D_i \leq P_i$
- ▶ $R_i^{(*)} > P_i \Rightarrow$ a second job of T_i has been released and the busy period is not over
- ▶ The response time of this job has to be computed
- ▶ General case: k jobs have to be tested, until a job finishes execution before the release of the following job
- ▶ Process: starting with $k = 1$, compute the latest possible ending time $R_i^{(*)}(k)$ of k^{th} job of T_i until $R_i^{(*)}(k) \leq k \times P_i$

$$R_i^{(0)}(k) = k \times C_i$$

$$R_i^{(n+1)}(k) = k \times C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^{(n)}(k)}{T_j} \right\rceil \times C_j$$

Ex: $T_1 (C_1:26, D_1 = P_1:70), T_2 (C_2:62, D_2:118, P_2:100)$

$$T_1: R_1^{(0)}(1) = C_1 = 26$$

$$T_1: R_1^{(1)}(1) = C_1 = 26 = R_1^{(*)}(1)$$

$$T_2: R_2^{(0)}(1) = C_2 = 62$$

$$T_2: R_2^{(1)}(1) = C_2 + \lceil \frac{R_2^{(0)}(1)}{T_1} \rceil \times C_1 = 62 + \lceil \frac{62}{70} \rceil \times 26 = 88$$

$$T_2: R_2^{(2)}(1) = C_2 + \lceil \frac{R_2^{(1)}(1)}{T_1} \rceil \times C_1 = 62 + \lceil \frac{88}{70} \rceil \times 26 = 114$$

$$T_2: R_2^{(3)}(1) = C_2 + \lceil \frac{R_2^{(2)}(1)}{T_1} \rceil \times C_1 = 62 + \lceil \frac{114}{70} \rceil \times 26 = 114 = R_2^{(*)}(1)$$

$$T_2: R_2^{(0)}(2) = 2 \times C_2 = 124$$

...

$$T_2: R_2^{(2)}(2) = C_2 + \lceil \frac{R_2^{(1)}(2)}{T_1} \rceil \times C_1 = 124 + \lceil \frac{176}{70} \rceil \times 26 = 202$$

$$T_2: R_2^{(3)}(2) = C_2 + \lceil \frac{R_2^{(2)}(2)}{T_1} \rceil \times C_1 = 124 + \lceil \frac{202}{70} \rceil \times 26 = 202 = R_2^{(*)}(2)$$

\Rightarrow Response time of T_2 second job: 102

Ex: $T_1 (C_1:26, D_1 = P_1:70), T_2 (C_2:62, D_2:118, P_2:100)$

- $R_2^{(*)}(3) = 316$
 \Rightarrow Response time of T_2 second job: 116
- $R_2^{(*)}(4) = 404$
 \Rightarrow Response time of T_2 second job: 104
- $R_2^{(*)}(5) = 518$
 \Rightarrow Response time of T_2 second job: 118
- $R_2^{(*)}(6) = 606$
 \Rightarrow Response time of T_2 second job: 106
- $R_2^{(*)}(7) = 694$
 \Rightarrow Response time of T_2 second job: 94

Rate Monotonic

- Basic algorithm for periodic tasks with implicit deadlines
- The smaller the period of a task, the higher its priority
- Optimal algorithm within the class of fixed priority scheduling algorithms for independent periodic tasks with implicit deadlines
- Sufficient test for the schedulability of a configuration including n tasks:

$$U = \sum_{i=1}^n \frac{C_i}{P_i} \leq n \times (2^{\frac{1}{n}} - 1)$$

n	1	2	3	5	10	grand
bound	1	0.83	0.78	0.74	0.72	0.69

- Necessary test for the schedulability of a configuration including n tasks:

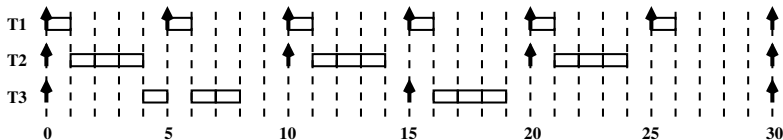
$$U \leq 1$$

Example with Rate Monotonic

- Task features

	WCET	Deadline	Period
T_1	1	5	5
T_2	3	10	10
T_3	3	15	15

- Scheduling sequence in the case of simultaneous releases



Rate Monotonic: worst-case response time computation

- Task features

	WCET	Deadline	Period
T_1	1	5	5
T_2	3	10	10
T_3	3	15	15

- Response time analysis

	T_1	T_2	T_3
t_0	C_1 = 1	$C_1 + C_2$ = 4	$C_1 + C_2 + C_3$ = 7
t_1	$C_1 \left\lceil \frac{t_0}{P_1} \right\rceil$ = 1	$C_1 \left\lceil \frac{t_0}{P_1} \right\rceil + C_2 \left\lceil \frac{t_0}{P_2} \right\rceil$ = 4	$C_1 \left\lceil \frac{t_0}{P_1} \right\rceil + C_2 \left\lceil \frac{t_0}{P_2} \right\rceil + C_3 \left\lceil \frac{t_0}{P_3} \right\rceil$ = 8
t_2			$C_1 \left\lceil \frac{t_1}{P_1} \right\rceil + C_2 \left\lceil \frac{t_1}{P_2} \right\rceil + C_3 \left\lceil \frac{t_1}{P_3} \right\rceil$ = 8
	OK	OK	OK

Rate Monotonic: Exercice

- Task features

	WCET	Deadline	Period
T_1	2	5	5
T_2	4	10	10
T_3	3	18	18

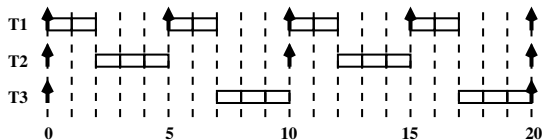
- Response time analysis for T_3 ?

A specific case: harmonic periods

- The sufficient and necessary condition becomes $U \leq 1$
- Task features

	WCET	Deadline	Period
T_1	2	5	5
T_2	3	10	10
T_3	6	20	20

- Scheduling sequence in the synchronous case

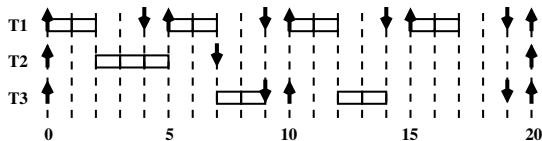


Deadline Monotonic

- In the case of constrained or arbitrary deadlines
- The smaller the deadline of a task, the higher its priority
- Example with the following task features

	WCET	Deadline	Period
T_1	2	4	5
T_2	3	7	20
T_3	2	9	10

- Scheduling case in the synchronous case

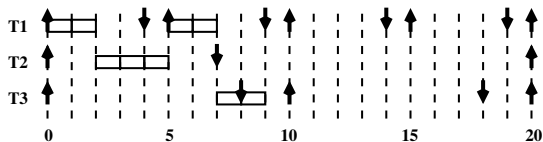


Deadline Monotonic: another example

- Task features

	WCET	Deadline	Period
T_1	2	4	5
T_2	3	7	20
T_3	2	8	10

- Scheduling sequence in the synchronous case

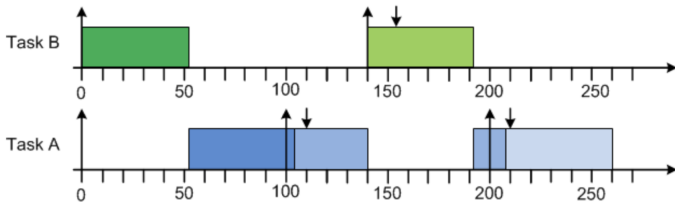


Non optimality of Deadline monotonic

- Deadline Monotonic is not an optimal fixed priority assignment
- Missed deadline if A has the highest priority

■ Tasks with arbitrary deadlines

Task	Execution Time	Deadline	Period
A	52	110	100
B	52	154	140



Dynamic priority algorithms

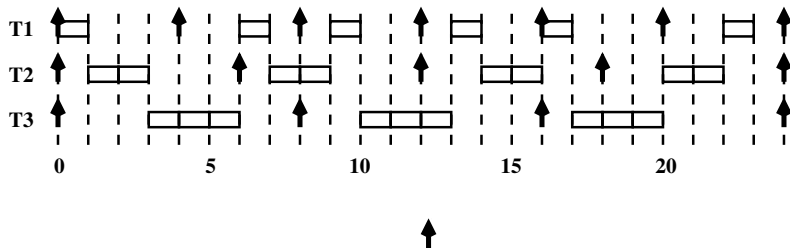
- The priority of a given task is not always the same
- Two classes of solutions
 - ▶ The priority of a job is fixed \Rightarrow job-level fixed priority scheduling
 - ★ Earliest Deadline first
 - ▶ The priority of a job may vary with time \Rightarrow job-level dynamic priority scheduling
 - ★ Least-Laxity First
 - ★ p-fair
 - ▶ Focus on Earliest Deadline First
 - ★ The most important (and analysed) dynamic priority algorithm
 - ★ The priority of a job is inversely proportional to its absolute deadline
 - ★ Two jobs with the same absolute deadline \Rightarrow random choice (doesn't matter)

Earliest Deadline First: example

- Task features

	WCET	Deadline	Period
T_1	1	4	4
T_2	2	6	6
T_3	3	8	8

- Scheduling sequence in the synchronous case



Earliest Deadline First vs Fixed Priority

- Some advantages of EDF
 - ▶ EDF can schedule all task sets that can be scheduled by Fixed Priority, but not vice versa
 - ▶ There is no need to define priorities in EDF
 - ★ Assuming Fixed Priority, basic priority assignment (e.g. Deadline Monotonic) are not optimal
 - ▶ EDF often has less preemptions
- Some disadvantages of EDF
 - ▶ EDF is not provided in many commercial RTOS
 - ▶ More implementation overhead with EDF
 - ▶ EDF is less controllable
 - ★ Reduce the response time of a task with Fixed Priority \Rightarrow increase its priority, nothing can be done with EDF

Schedulability analysis with EDF

- Task set with implicit deadlines for all the tasks
 - ▶ Necessary and sufficient condition

$$U = \sum_{i=1}^N \frac{C_i}{T_i} \leq 1$$

- ▶ \Rightarrow In this situation, EDF is obviously optimal
- Task sets with constrained or arbitrary deadlines
 - ▶ EDF is still optimal
 - ▶ The schedulability analysis becomes more complex
 - ▶ Based on the demand function (df)
 - ★ The demand function for a task T_i is a function of an interval $[t_1, t_2]$ that gives the amount of T_i computation that must be completed in $[t_1, t_2]$ for T_i to be schedulable

$$df_i(t_1, t_2) = \sum_{a_{ij} \geq t_1, d_{ij} \leq t_2} C_{ij}$$

- ★ For the entire task set

$$df(t_1, t_2) = \sum_{i=1}^N df_i(t_1, t_2)$$

Schedulability analysis with EDF

- For a set of synchronous periodic tasks, the worst case demand is found for intervals starting at 0

$$\forall t_1, t_2 > t_1 \quad df(t_1, t_2) \leq df(0, t_2 - t_1)$$

- Demand bound function

$$dbf(L) = \max_t (df(t, t + L) = df(0, L))$$

- Impossible to compute the dbf for all possible values of L
- For a set of synchronous periodic tasks, only consider one hyperperiod H and only consider task deadlines within H
 - ▶ A synchronous periodic task set TS is schedulable by EDF iff

$$\forall L \in dead(TS) \quad dbf(L) \leq L$$

Where $dead(TS)$ is the set of deadlines in $[0, H]$

- The synchronous case is a worst-case of any asynchronous one

Schedulability analysis with EDF

- Example with the following task features

	WCET	Deadline	Period
T_1	1	4	4
T_2	2	6	6
T_3	3	8	8

- $H = 24$
- deadline set: 4, 6, 8, 12, 16, 18, 20, 24
- $dbf(4) = 1$
- $dbf(6) = 3$
- $dbf(8) = 7$
- $dbf(12) = 10$
- $dbf(16) = 14$
- $dbf(18) = 16$
- $dbf(20) = 17$
- $dbf(24) = 23$

Earliest Deadline First: exercise

- Task features

	WCET	Deadline	Period
T_1	2	4	5
T_2	3	7	20
T_3	2	8	10

- Is it schedulable with EDF?

Rate Monotonic / Earliest Deadline First: Mixed solution

- Trade-off between Fixed Priority simplicity and EDF better schedulability
- Rate Monotonic for tasks with shortest periods
- Other tasks with dynamic lower priorities
- Task features

	WCET	Deasline	Period
T_1	1	4	4
T_2	1	5	5
T_3	2	6	6
T_4	2	10	10

Adding aperiodic tasks

- Up to now, only tasks with a known minimum duration between any two consecutive occurrences (the period)
- What if this minimum duration is unknown (i.e. aperiodic tasks)?
- Two cases have to be considered
 - ▶ Aperiodic jobs don't have deadlines
 - ★ Try to minimize their response time while ensuring that periodic tasks never miss deadlines
 - ▶ Aperiodic jobs do have deadlines
 - ★ Acceptance test: can it be executed before its deadline without jeopardising periodic jobs and already accepted aperiodic ones?
 - ★ If not, one solution is to reject the aperiodic job
 - ★ Another solution is to kill the least important job
 - ★ Approaches are classically based on Earliest Deadline First
- Presentation of some simple scheduling algorithms for the case without deadline

Background execution of aperiodic jobs without deadlines

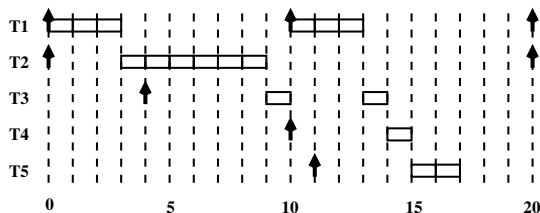
- The simplest solution
- Periodic tasks are scheduled using one of the previously presented algorithms (rate monotonic, deadline monotonic, earliest deadline first, ...)
- Aperiodic jobs are scheduled when there are no pending periodic ones
- Aperiodic jobs are scheduled following a "First Come First Served" policy
- Straightforward implementation
 - ▶ Periodic tasks are assigned priorities, based on the scheduling policy
 - ▶ All aperiodic jobs are assigned the same lowest priority
- This solution makes nothing to minimise the response time of aperiodic jobs

An example of the background execution approach

- Task features

	(first) occurrence	WCET	Deadline	Period
T_1	0	3	10	10
T_2	0	6	20	20
T_3	4	2		
T_4	10	1		
T_5	11	2		

- Task scheduling, assuming rate monotonic



Polling server for aperiodic jobs without deadlines

- Server: periodic task with a budget for the execution of pending aperiodic jobs
- Short period for the server \Rightarrow high priority task (consider for instance rate monotonic)
- Every time the server gets the processor

Budget $\leftarrow C_s$;

Next_Activation $\leftarrow t_{\text{current}} + P_s$;

While Pending aperiodic jobs **and** Budget > 0 **do**

 Execute one time unit of oldest pending aperiodic job;

 Budget \leftarrow Budget - 1;

EndWhile;

Budget $\leftarrow 0$;

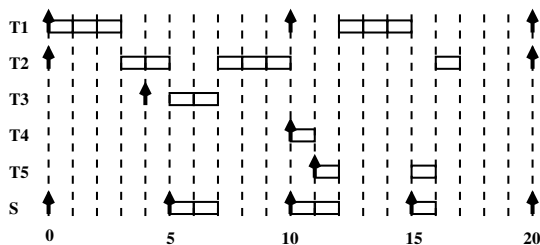
- Aperiodic request after the server dropped its budget \Rightarrow it has to wait for the next server activation

An example of the polling server approach

- Task features

	(first) occurrence	WCET	Deadline	Period
T_1	0	3	10	10
T_2	0	6	20	20
S	0	2	5	5
T_3	4	2		
T_4	10	1		
T_5	11	2		

- Task scheduling assuming rate monotonic



Deferable server for aperiodic jobs without deadlines

- The budget is not reset to 0 when there are no (more) pending aperiodic jobs
- Every time the server is activated

Budget \leftarrow Cs;

Next_Activation \leftarrow t_current + Ps;

While pending aperiodic jobs **and** Budget > 0 **do**

 Execute one time unit of oldest pending aperiodic job;

 Budget \leftarrow Budget - 1;

EndWhile;

- Aperiodic request while server is not executing, but has some remaining budget

While pending aperiodic jobs **and** Budget > 0 **do**

 Execute one time unit of new pending aperiodic job;

 Budget \leftarrow Budget - 1;

EndWhile;

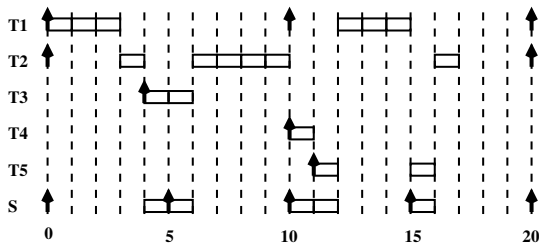
- Can lead to missed deadlines for periodic jobs!!!

An example of the deferrable server approach

- Task features

	(first) occurrence	WCET	Deadline	Period
T_1	0	3	10	10
T_2	0	6	20	20
S	0	2	5	5
T_3	4	2		
T_4	10	1		
T_5	11	2		

- Task scheduling assuming rate monotonic

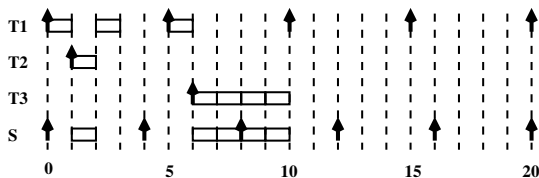


Another example of the deferrable server approach

- Task features

	(first) occurrence	WCET	Deadline	Period
T_1	0	2	5	5
S	0	2	4	4
T_2	1	1		
T_3	6	4		

- Task scheduling assuming rate monotonic



Sporadic server for aperiodic jobs without deadlines

- No budget reset, the period starts when the server consumes budget
- Every time the server is activated

Budget \leftarrow Cs;

If pending aperiodic jobs **then**

Next_Activation \leftarrow t_current + Ps;

While pending aperiodic jobs **and** Budget > 0 **do**

Execute one time unit of oldest pending aperiodic job;

Budget \leftarrow Budget - 1;

EndWhile;

EndIf;

- Aperiodic request while server is not executing, but has some remaining budget

If Budget = Cs **Then** Next_Activation \leftarrow t_current + Ps;

While pending aperiodic jobs **and** Budget > 0 **do**

Execute one time unit of oldest pending aperiodic job;

Budget \leftarrow Budget - 1;

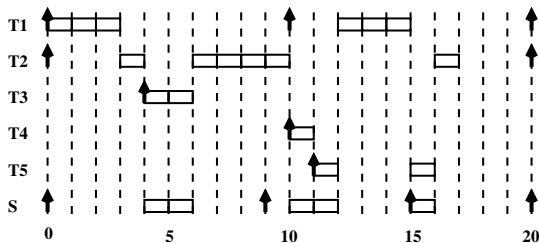
EndWhile;

An example of the sporadic server approach

- Task features

	(first) occurrence	WCET	Deadline	Period
T_1	0	3	10	10
T_2	0	6	20	20
S	0	2	5	5
T_3	4	2		
T_4	10	1		
T_5	11	2		

- Task scheduling assuming rate monotonic

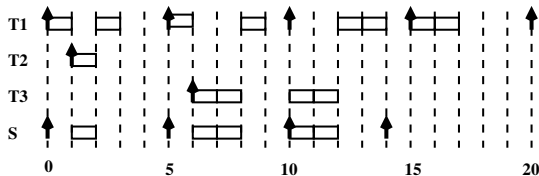


Another example of the sporadic server approach

- Task features

	(first) occurrence	WCET	Délai critique	Période
T_1	0	2	5	5
S	0	2	4	4
T_2	1	1		
T_3	6	4		

- Task scheduling assuming rate monotonic



Exercise

- Task features, periodic tasks scheduled with rate monotonic

	(first) occurrence	WCET	D	P
T_1	0	1	6	6
T_2	0	4	12	12
T_3	0	6	24	24
T_4	1	2		
T_5	11	1		
T_6	13	2		

- Response time of T_4 , T_5 and T_6 with the background approach
- Same question with a polling server with budget 1 and period 4
- Same question with a deferrable server with budget 1 and period 4
- Same question with a sporadic server with budget 1 and period 4

Precedence constraints with rate monotonic

- Precedence constraints between periodic tasks \Rightarrow defines an execution order
- one precedence constraint: n^{th} job of task T_i has to execute before n^{th} job of task T_j

$$T_i \rightarrow T_j$$

- Precedence constraints of a configuration are modelled by a precedence graph
- Tasks sharing precedence constraints typically have the same period
- Rate monotonic solution \Rightarrow precedence constraints are integrated in task features offline

- ▶ Activation dates and deadlines are modified offline for each task T_j

$$\begin{aligned} r_j^* &= \max(r_j, r_i^*) && \text{for every } T_i \rightarrow T_j \\ D_j^* &= D_j - (r_j^* - r_j) \end{aligned}$$

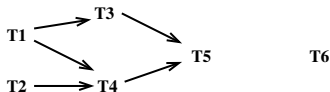
- ▶ Different priorities for tasks with the same period: T_i has a higher priority than T_j if $T_i \rightarrow T_j$

Example with precedence constraints and rate monotonic

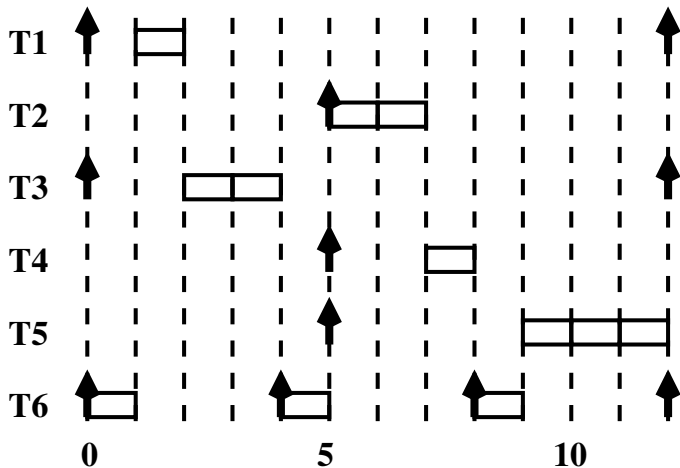
- Task features

	first occurrence		WCET	Deadline		Period
	r	r^*		D	D^*	
T_1	0	0	1	12	12	12
T_2	5	5	2	7	7	12
T_3	0	0	2	12	12	12
T_4	0	5	1	12	7	12
T_5	0	5	3	12	7	12
T_6	0	0	1	4	4	4

- Precedence constraints



Example with precedence constraints and rate monotonic



Precedence constraints with Earliest Deadline First

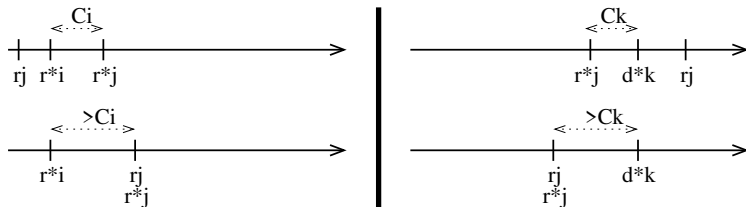
- Similar principle as with rate monotonic \Rightarrow off-line integration of constraints in task features

- Modification of the activation date of T_j

$$r_j^* = \max(r_j, \max(r_i^* + C_i)) \quad \text{for all } T_i \rightarrow T_j$$

- Modification of the deadline of T_j

$$d_j^* = \min(d_j, \min(d_k^* - C_k)) \quad \text{for all } T_j \rightarrow T_k$$

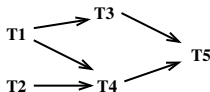


Example with precedence constraints and EDF

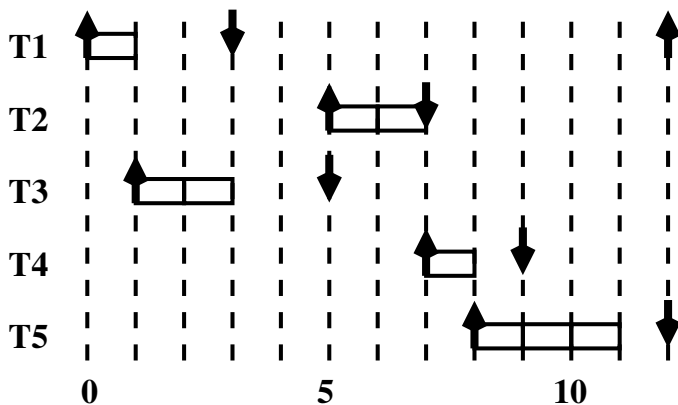
- Task features

	first occurrence		WCET	Deadline		Period
	r	r^*		d	d^*	
T_1	0	0	1	5	3	12
T_2	5	5	2	7	7	12
T_3	0	1	2	5	5	12
T_4	0	7	1	10	9	12
T_5	0	8	3	12	12	12

- Precedence constraints



Example with precedence constraints and EDF



Dealing with shared resources

- Tasks often have to access to shared resources (i.e. shared variables) in mutual exclusion
- Critical sections are used to implement the access to these shared resources.
- A given task T_i cannot enter a critical section if the corresponding resource is not free
- Two potential problems
 - ▶ The deadlock: non cycle-free resource demands at a given time
 - ▶ Priority inversion: a task T_i waits for a resource earned by a low priority task T_j
- A deadlock should never occur in a critical real-time system
 - ▶ One solution: implement tasks such that they a deadlock can never occur (e.g. always take resources in the same order)
 - ▶ Another solution: provide a resource allocation mechanism that prevents deadlock (i.e. gives a resource only if it cannot lead to a deadlock)
- Priority inversion cannot be avoided, but should be upper bounded
 - ▶ Thanks to the resource allocation mechanism

Example of a deadlock

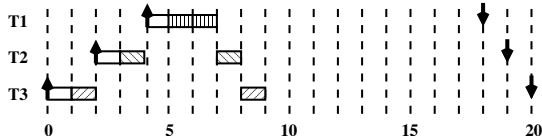
- Task features

	First release	WCET	Deadline	Period
T_1	4	5	14	20
T_2	2	5	17	20
T_3	0	5	20	20

Resource utilisation

	R_3	R_3	$R_1 R_3$	
	R_2	R_2	$R_2 R_3$	
	R_1	R_1	$R_1 R_2$	

- Scheduling result



- Deadlock at $t = 9$

Example of a priority inversion

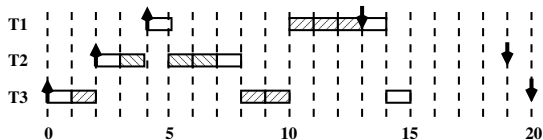
- Task features

	First release	WCET	Deadline	Period
T_1	4	5	9	20
T_2	2	5	17	20
T_3	0	5	20	20

Resource utilisation

	R_1	R_1	R_1	
	R_2	R_2	R_2	
	R_1	R_1	R_1	

- Scheduling result



- Priority inversion between $t = 5$ and $t = 10$

The "super priority" approach

- The simplest solution to prevent deadlock and upper bound priority inversion
- A task asks for a resource \Rightarrow it gets it immediately
- Every task which owns a resource gets the highest priority in the system \Rightarrow no preemption within a critical section \Rightarrow at most one task in critical section at any time
- A deadlock cannot occur
- the duration of a priority inversion is upper bounded by the longest critical section
- No need to know resource utilisation by tasks beforehand
- Doesn't take into account which resource is used \Rightarrow can generate additional priority inversions

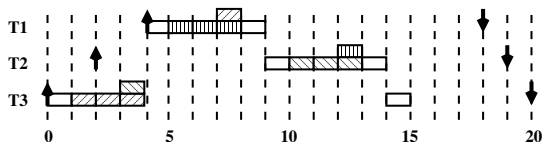
A first example of the super priority approach

- Task features

	First release	WCET	Deadline	Period
T_1	4	5	14	20
T_2	2	5	17	20
T_3	0	5	20	20

Resource utilisation				
	R_3	R_3	$R_1 R_3$	
	R_2	R_2	$R_2 R_3$	
	R_1	R_1	$R_1 R_2$	

- Scheduling result



- No more deadlock

A second example of the super priority approach

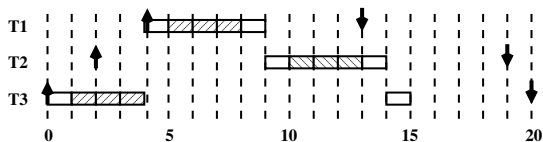
- Task features

	First release	WCET	Deadline	Period
T_1	4	5	9	20
T_2	2	5	17	20
T_3	0	5	20	20

Resource utilisation

	R_1	R_1	R_1	
	R_2	R_2	R_2	
	R_1	R_1	R_1	

- Scheduling result



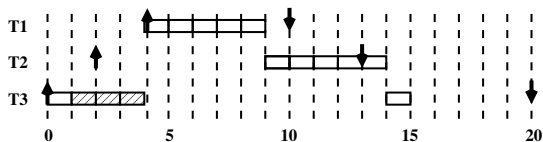
A third example of the super priority approach

- Task features

	First release	WCET	Deadline	Period
T_1	4	5	9	20
T_2	2	5	17	20
T_3	0	5	20	20

Resource utilisation				
	R_1	R_1	R_1	

- Scheduling result



The priority inheritance protocol

- Doesn't prevent deadlock, but upper bound priority inversion duration
- When a task T_i asks for an unavailable resource, the task T_j which owns the resource inherits the priority of blocked task T_i
- This rule is transitively applied
- No need to know resource utilisation by tasks beforehand

An example of the priority inheritance protocol

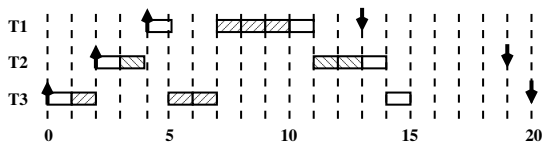
- Task features

	First release	WCET	Deadline	Period
T_1	4	5	9	20
T_2	2	5	17	20
T_3	0	5	20	20

Resource utilisation

	R_1	R_1	R_1	
	R_2	R_2	R_2	
	R_1	R_1	R_1	

- Scheduling result



The “stack-based protocol”

- Based on the ceiling priority notion
 - ▶ The ceiling priority of a resource is the highest priority among the tasks which use this resource
 - ▶ Task requests in terms of resources have to be known beforehand
- A task owning no resource executes at its initial priority
- A task owning resources executes at the highest ceiling priority of these resources
- Every task asking for a resource gets it immediately
- It prevents deadlocks
- It upper bounds priority inversion duration

A first example of the “stack-based protocol”

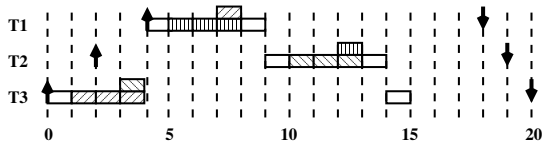
- Task features

	First release	WCET	Deadline	Period
T_1	4	5	14	20
T_2	2	5	17	20
T_3	0	5	20	20

Resource utilisation

	R_3	R_3	$R_1 R_3$	
	R_2	R_2	$R_2 R_3$	
	R_1	R_1	$R_1 R_2$	

- Scheduling result



A second example of the “stack-based protocol”

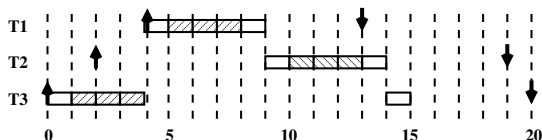
- Task features

	First release	WCET	Deadline	Period
T_1	4	5	9	20
T_2	2	5	17	20
T_3	0	5	20	20

Resource utilisation

	R_1	R_1	R_1	
	R_2	R_2	R_2	
	R_1	R_1	R_1	

- Scheduling result



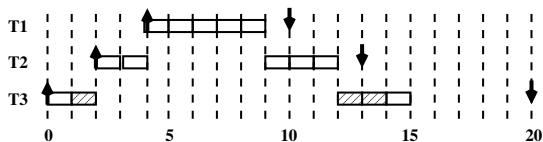
A third example of the “stack-based protocol”

- Task features

	First release	WCET	Deadline	Period
T_1	4	5	9	20
T_2	2	5	17	20
T_3	0	5	20	20

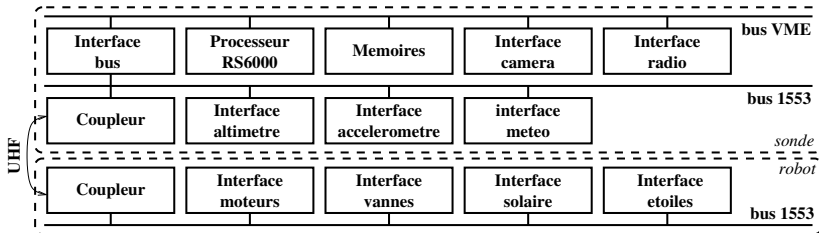
Resource utilisation				
	R_1	R_1	R_1	

- Scheduling sequence



A real example: the Pathfinder mission

- NASA mission (Jet Propulsion Laboratory) in order to characterise Mars environment
- Launched: 4 december, 1996
 - ▶ Lander: 264 Kg
 - ▶ Rover 11.5 Kg with six wheels, powered by solar panels and lithium batteries



- Landing on Mars: 4 juillet, 1997
- Some days without any problem, then the embedded computer resets several times \Rightarrow lots of data lost
- Remote detection and correction of the problem

A real example: the Pathfinder mission

- Multitask software using VxWorks RTOS (Wind River System)
- More than 25 periodic and aperiodic tasks
 - ▶ Mode management (fly, approach, landing, exploration)
 - ▶ Surface pointing
 - ▶ Fault analysis
 - ▶ Weather forecast management
 - ▶ 1553 bus control
 - ▶ Star analysis
 - ▶ ...
- Different phases in the mission (fly, approach, landing, exploration) \Rightarrow a subset of the tasks are executing in each phase
- The mode management task puts each task in the right state (asleep, awake)
- An internal real-time clock for the activation of periodic tasks
- Problem occurs during the execution phase

A real example: the Pathfinder mission

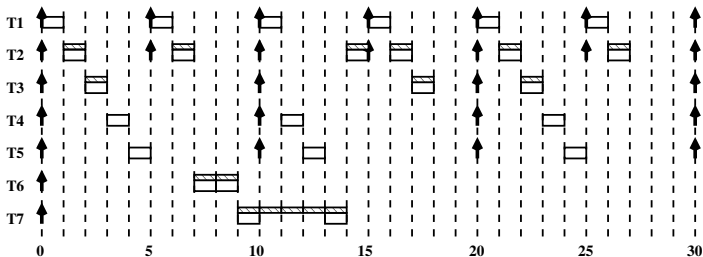
- Task features for the exploration phase

Task name	WCET	Period	Priority
Bus_scheduling (T1)	25 μs	125 μs	1
Data_distribution (T2)	25 μs	125 μs	2
Guiding (T3)	25 μs	250 μs	3
Radio (T4)	25 μs	250 μs	4
Camera (T5)	25 μs	250 μs	5
Measures (T6)	50 μs	5000 μs	6
Weather (T7)	50; 75 μs	5000 μs	7

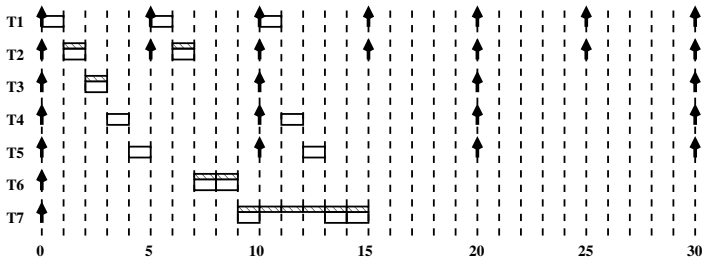
- One time unit: 25 μs
- T2, T3, T6 and T7 share a resource *data_buffer*
- Each task asks for the resource at the beginning of its execution and gives it back at the end
- The inheritance priority protocol is not used

A real example: the Pathfinder mission

- Worst-case when the execution time for Weather task is $50 \mu s \Rightarrow$ ok

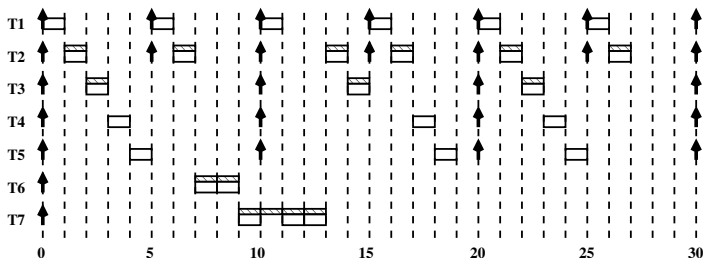


- Worst-case when the execution time for Weather task is $75 \mu s \Rightarrow$ T2 misses a deadline



A real example: the Pathfinder mission

- Problem: too long priority inversion (between $t = 11$ et $t = 15$)
- Solution: Utilisation of the priority inheritance protocol
- Worst-case when the execution time for Weather task is $75 \mu s \Rightarrow$ ok

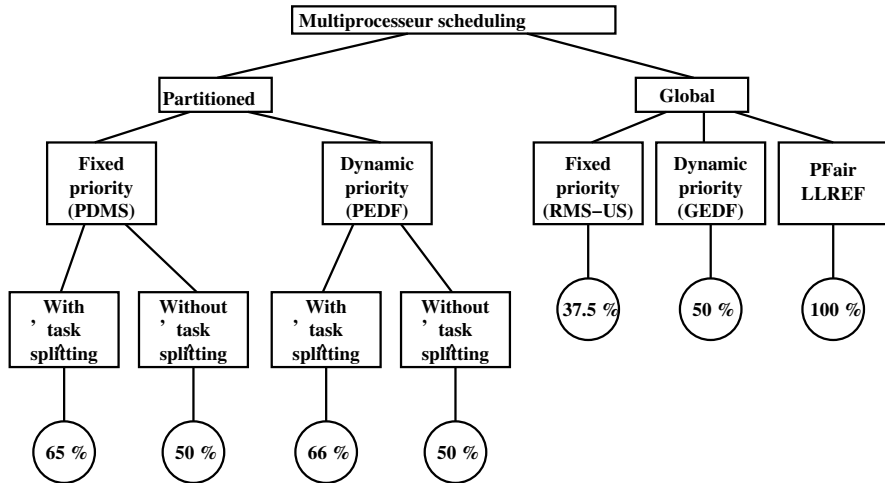


- Correction of the problem by uploading the modified code

Multiprocessor scheduling

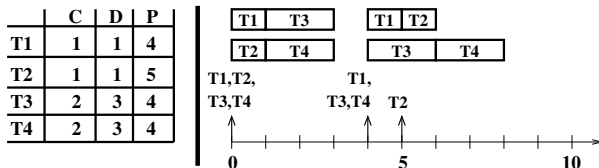
- Problem definition
 - ▶ Scheduling of a set of n periodic tasks on m processors
 - ▶ Each processor executes at most one task at any instant
 - ▶ Each task is executed by at most one processor at any instant
- Different criteria to classify the solutions
 - ▶ To which extent task migrations are allowed
 - ★ Partitioned scheduling No migrations: tasks statically distributed on processors
 - ★ Global scheduling
 - Task level migration: no migration of a job during its execution
 - Job level migration
 - ▶ Priorities assigned to tasks
 - ★ Task level fixed priorities (Rate Monotonic, ...)
 - ★ Job level fixed priorities (Earliest Deadline First, ...)
 - ★ Fully dynamic priorities (least laxity, PFAIR, DP-FAIR, ...)

Overview of the different solutions



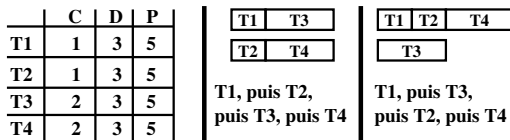
Some scheduling anomalies

- The synchronous case doesn't always lead to the worst-case response time



⇒ The second job of T4 misses its deadline

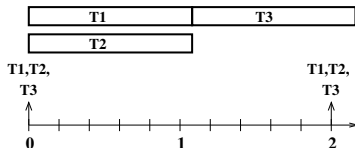
- The response time of a task might depend on the priority order of higher priority tasks



⇒ T4 misses its deadline in the second case

Partitioned scheduling

- Tasks are statically distributed on the available processors
 - ▶ Optimal allocation: non polynomial problem (*bin packing*)
- Each processor executes independently its allocated task set
 - ▶ Benefit from the monoprocessor scheduling theory
- The bound on the utilisation factor for each processor can be quite low
 - ▶ $m + 1$ periodic tasks with implicit deadlines
 - ★ Worst-case execution time of one job: $1 + \epsilon$
 - ★ Period of a task: 2
 - ▶ not schedulable on m processors with a partitioned approach \Rightarrow on such a configuration we cannot exceed $U = \frac{m+1}{2}$ on m processors



Distribution of the tasks on the processors

- Divide the task set into m subsets such that each subset is schedulable on one processor
- One possible algorithm: Rate Monotonic First Fit
 - ▶ Tasks are sorted by increasing periods
 - ▶ Each task is allocated to the first processor which remains schedulable with this additional task
- Example of task distribution with RM first fit

(P_i, C_i)	U_i	(P_i, C_i)	U_i	(P_i, C_i)	U_i
(2, 1)	0.500	(4.5, 0.1)	0.022	(8, 1)	0.125
(2.5, 0.1)	0.040	(5, 1)	0.200	(8.5, 0.1)	0.012
(3, 1)	0.333	(6, 1)	0.167	(9, 1)	0.111
(4, 1)	0.250	(7, 1)	0.143		

$P_1 : (2, 1), (2.5, 0.1), (4.5, 0.1), (6, 1), (8.5, 0.1)$

$P_2 : (3, 1), (4, 1), (7, 1)$

$P_3 : (5, 1), (8, 1), (9, 1)$

Task splitting to improve the partitioning

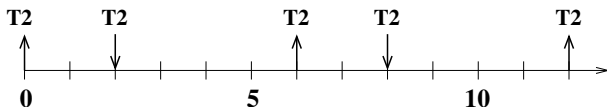
- Comes to allow some task migrations
- Example: PDMS-HPTS-DS algorithm
 - ▶ Deadline Monotonic scheduling on each processor
 - ▶ Tasks are allocated by decreasing load
 - ▶ As soon as the task set allocated on the current processor is no more schedulable, a portion of the highest priority task is removed such that the task set is just schedulable
 - ▶ This portion is allocated to the next processor
- At most $m - 1$ splitted tasks for m processors
- The two portions of the splitted task must not execute simultaneously
 - ▶ The release time of the second portion must not be before the latest possible end of the first portion
 - ▶ Splitting the highest priority task minimises this latest possible end

Task splitting to improve the partitioning

- Example with five tasks on two processors

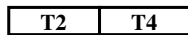
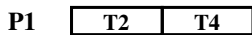
	(P_i, D_i, C_i)	U_i	CH_i		(P_i, D_i, C_i)	U_i	CH_i
T_1	(4, 3, 1)	0.25	0.33	T_2	(6, 2, 2)	0.33	1
T_3	(4, 4, 1)	0.25	0.25	T_4	(6, 4, 2)	0.33	0.5
T_5	(6, 5, 1)	0.16	0.2				

- Allocation of the task with the highest load, i.e. T_2 on processor $P1$:
OK

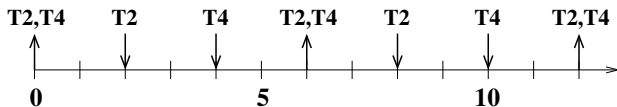


Task splitting to improve the partitioning

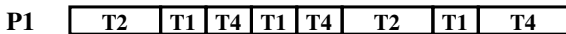
- Allocation of the remaining task with the highest load, i.e. T_4 on processor $P1$: OK



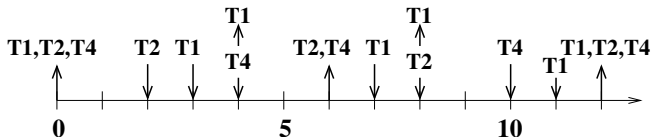
P2



- Allocation of the remaining task with the highest load, i.e. T_1 on processor $P1$: KO (T_4 at 4 and 10)

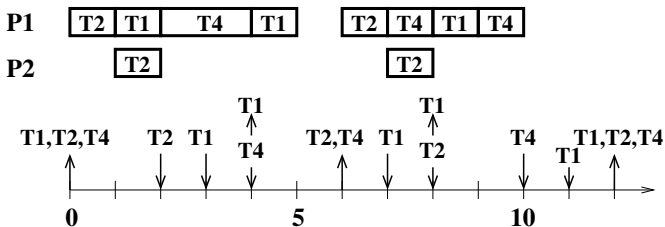


P2

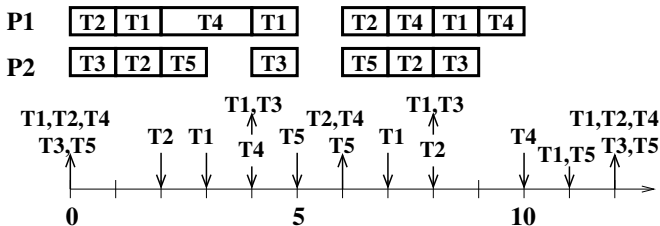


Task splitting to improve the partitioning

- Splitting of the highest priority task (T_2) such that the subset allocated to $P1$ becomes schedulable

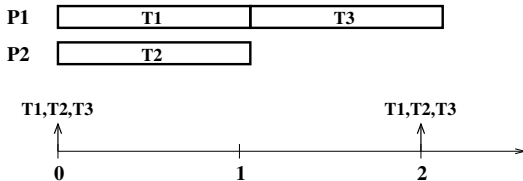


- Allocation of the two remaining tasks (T_3 and T_5) on processor $P2$

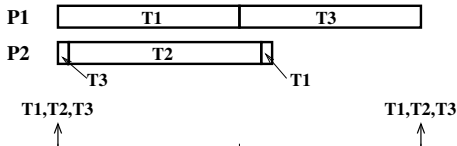


Global approaches without job migrations

- At any time, the m pending tasks with the highest priority are executed
- A job starts execution on a processor and never migrates
- Leads to a load balancing between processors
- Cannot schedule on m processors $m + 1$ periodic tasks avec a WCET of $1 + \epsilon$ and a period of 2
 - Example: three tasks on two processors



- Schedulable configuration if job migration is allowed migration des occurrences



Global approaches with job migrations

- A job can migrate on a different processor during its execution
- An ideal algorithm for tasks with implicit deadlines
 - ▶ At any time, every task gets the percentage of the processor equal to its utilisation factor (fluid model)
 - ▶ Example: a task T_i with $C_i = 8$ and $P_i = 11$ always gets $\frac{8}{11} = 72.73\%$ of the processor
 - ▶ Cannot be implemented like that
- An approximation: Pfair (Proportionate-fair)
 - ▶ The processor is allocated by discrete time units
 - ▶ The difference with the fluid model must never exceed one time unit

$$-1 < \text{difference}(T_i, t) < 1 \text{ for } t \in N^*$$

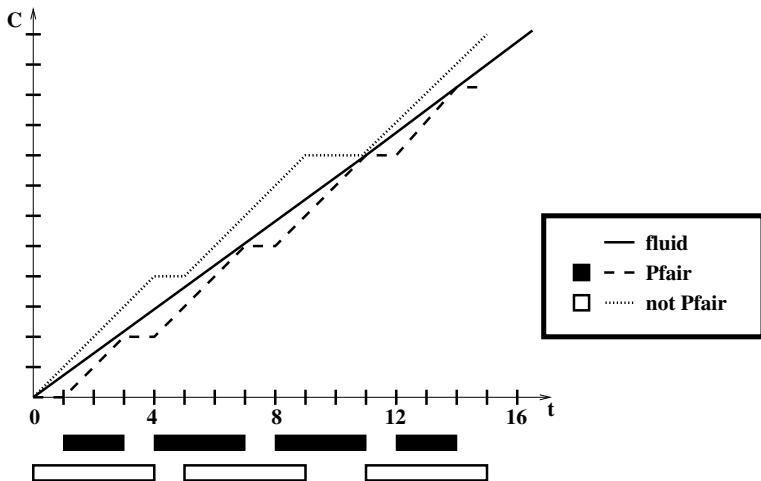
with

$$\text{difference}(T_i, t) = U_i \times t - \sum_{x=0}^{t-1} S(T_i, x)$$

et

$$S(T_i, x) = \begin{cases} 1 & \text{if } T_i \text{ is executing during time unit } x \\ 0 & \text{otherwise} \end{cases}$$

Illustration of Pfair for $C_i = 8$ and $P_i = 11$



- Optimal algorithm
- Typically leads to a very high number of preemptions and migrations
⇒ very large overhead

Can a greedy algorithm be optimal?

- Modeling of the urgency of a task by a single value
 - ▶ *Earliest deadline first*
 - ▶ *Least laxity first*
- Fails for schedulable task configurations
- Example: *least laxity first* on two processors processeurs

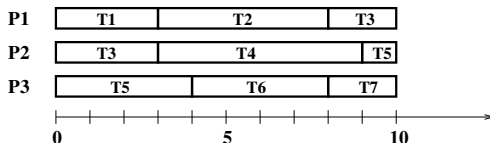
	C	P
T_1	9	10
T_2	9	10
T_3	8	40

- Utilisation factor of 2 for the configuration \Rightarrow no idle time on any processor for any valid scheduling sequence
- One idle time at $t = 9$ with *least laxity first* \Rightarrow fails
- No known greedy algorithm successfully schedules this configuration

Specific case: all the tasks have the same deadline

- Each task gets a slot equal to its WCET
- For m processors and a duration d_{dead} until the deadline
 - ▶ Building of a sequence with length $m \times d_{dead}$ with a slot of length its WCET for each task
 - ▶ Splitting of the sequence in m parts with the same length
- An example with three processors and seven tasks

	C	P		C	P
T_1	3	10	T_5	5	10
T_2	5	10	T_6	4	10
T_3	5	10	T_7	2	10
T_4	6	10			



DP-Fair : Deadline Partitionning

- Segmentation of the time, based on task deadlines
- In each portion, each task gets a slot corresponding to its utilisation factor
- The algorithm of the previous slide is applied in each portion
- Much less preemptions and migrations than with Pfair
- Example with three tasks on two processors

	C	P
T_1	9	10
T_2	9	10
T_3	8	40

