Real-Time Scheduling Analysis

Frank Singhoff

Bureau C-203

University of Brest, France

Lab-STICC UMR CNRS 6285

singhoff@univ-brest.fr

Summary

What is real-time scheduling theory (1)

- **Many real-time systems are built with operating systems providing multitasking facilities, in order to:**
  - Ease the design of complex systems (one function = one task).
  - Increase efficiency (I/O operations, multi-processor architecture).
  - Increase re-usability.

But, multitasking makes the predictability analysis difficult to do : we must take the task scheduling into account in order to check temporal constraints $\Longrightarrow$ schedulability analysis.

What is real-time scheduling theory (2)

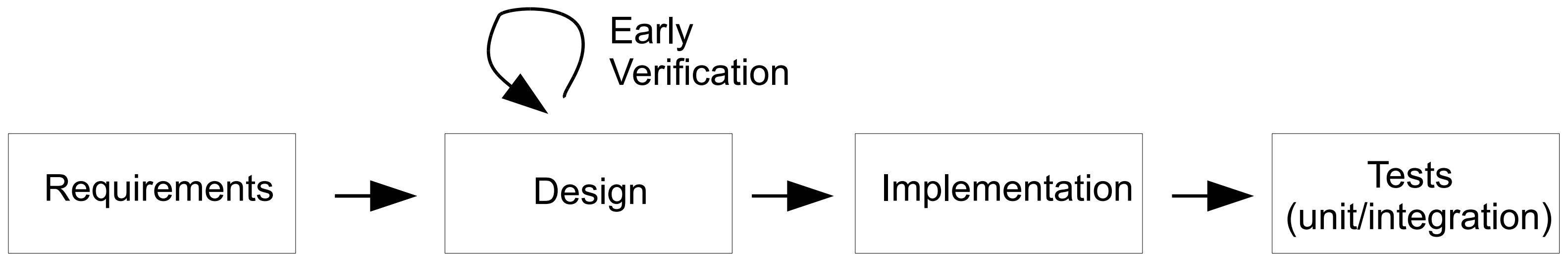
- **Example of a software embedded into a car:**

  1. Tasks are released several times and have a job to do for each release.
  2. Each task completes its current job before it has been released for the next one.
  3. A task displays every 100 milliseconds the current speed of the car.
  4. A task reads a speed sensor every 250 milliseconds.
  5. A task performs an engine monitoring algorithm every 500 milliseconds.

$\Longrightarrow$ How can we check that tasks will be run at the proper rate? Do they meet their timing requirements? Do we have enough processor resource?

$\Longrightarrow$ If the system is complex (e.g. large number of tasks), the designer must be helped to perform such an analysis.

What is real-time scheduling theory (3)



- The real-time scheduling theory provides:

1. **Algorithms to share a processor** (or any resources) by a set of tasks (or any resource users) according to some timing requirements $\Longrightarrow$ take urgency of the tasks into account.
2. **Analytical methods,** called **feasibility tests** or **schedulability tests**, which allow a system designer to early analyze/"compute" the system behavior before implementation/execution time.

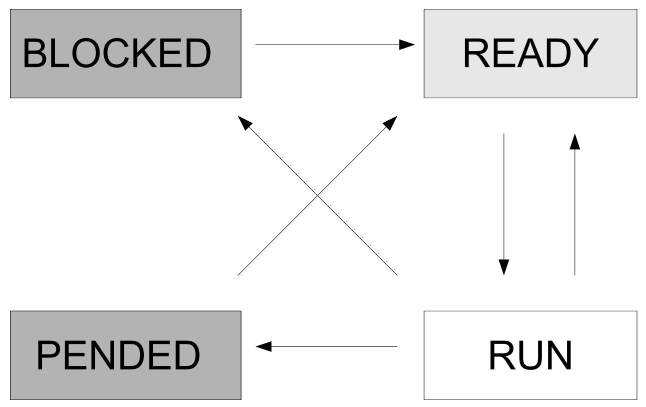
Scheduling algorithms (1)

- **Different kinds of real-time schedulers:**
  - **On-line/off-line scheduler:** the scheduling is computed before or at execution time?
  - **Static/dynamic priority scheduler:** priorities may change at execution time?
  - **Preemptive or non preemptive scheduler:** can we stop a task during its execution ?
    1. Preemptive schedulers are more efficient than non preemptive schedulers (e.g. missed deadlines).
    2. Non preemptive schedulers ease the sharing of resources.
    3. Overhead due to context switches.

Scheduling algorithms (2)

- **Different kinds of real-time schedulers:**
  - **Feasibility tests (schedulability tests) exist or not:** can we prove that tasks will meet deadlines before execution time?
  - **Complexity:** can we apply feasibility tests on large systems (e.g. large number of tasks)?
  - **Suitability:** can we implement the chosen scheduler in a real-time operating system?

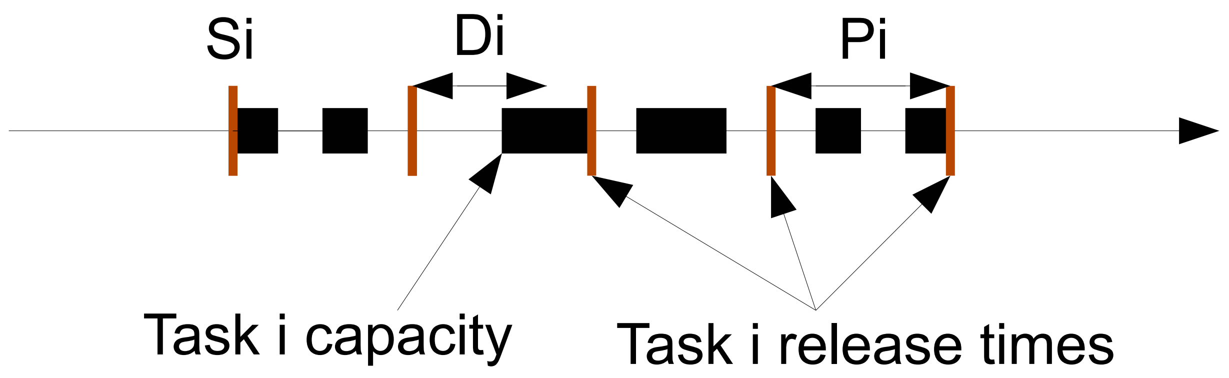
Models of task (1)



- **Task**: sequence of statements + data + execution context (processor and MMU). Usual states of a task.

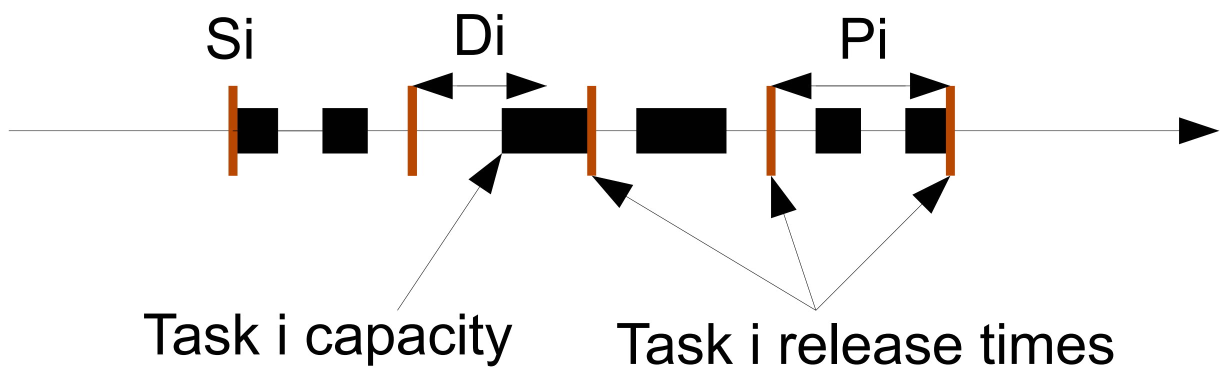
- **Usual task types**:
  - Urgent or/and critical tasks.
  - Independent tasks or dependent tasks.
  - Periodic and sporadic tasks (critical tasks). Aperiodic tasks (non critical tasks).

Models of task (2)



- **Usual parameters of a periodic task i:**
  - Period: P_i (duration between two periodic release times). A task starts a job for each release time.
  - Static deadline to meet: D_i, timing constraint relative to the period/job.
  - First task release time (first job): S_i.
  - Worst case execution time of each job: C_i (or capacity).
  - Priority: allows the scheduler to choose the task to run.

Models of task (3)



- **Assumptions for this lecture (synchronous periodic task with deadlines on requests) [LIU 73]:**

1. Tasks are periodic.
2. Tasks are independent.
3. Tasks have deadline on request, e.g. $\forall i : P_i = D_i$: a task must end its current job before its next release time.
4. Tasks are synchronous, e.g. $\forall i : S_i = 0 \implies$ called critical instant (worst case on processor demand).

Summary

Usual real-time schedulers

1. **Fixed priority scheduler:** Rate Monotonic priority assignment (sometimes called Rate Monotonic Scheduling or Rate Monotonic Analysis, RM, RMS, RMA).
2. **Dynamic priority scheduler:** Earliest Deadline First (or EDF).

Fixed priority scheduling (1)

- **Assumptions/properties of fixed priority scheduling :**
  - Scheduling based on fixed priority $\Longrightarrow$ static and critical applications.
  - Priorities are assigned at design time (off-line).
  - Efficient and simple feasibility tests.
  - Scheduler easy to implement into real-time operating systems.
- **Assumptions/properties of Rate Monotonic assignment:**
  - Optimal assignment in the case of fixed priority scheduling.
  - Periodic tasks only.

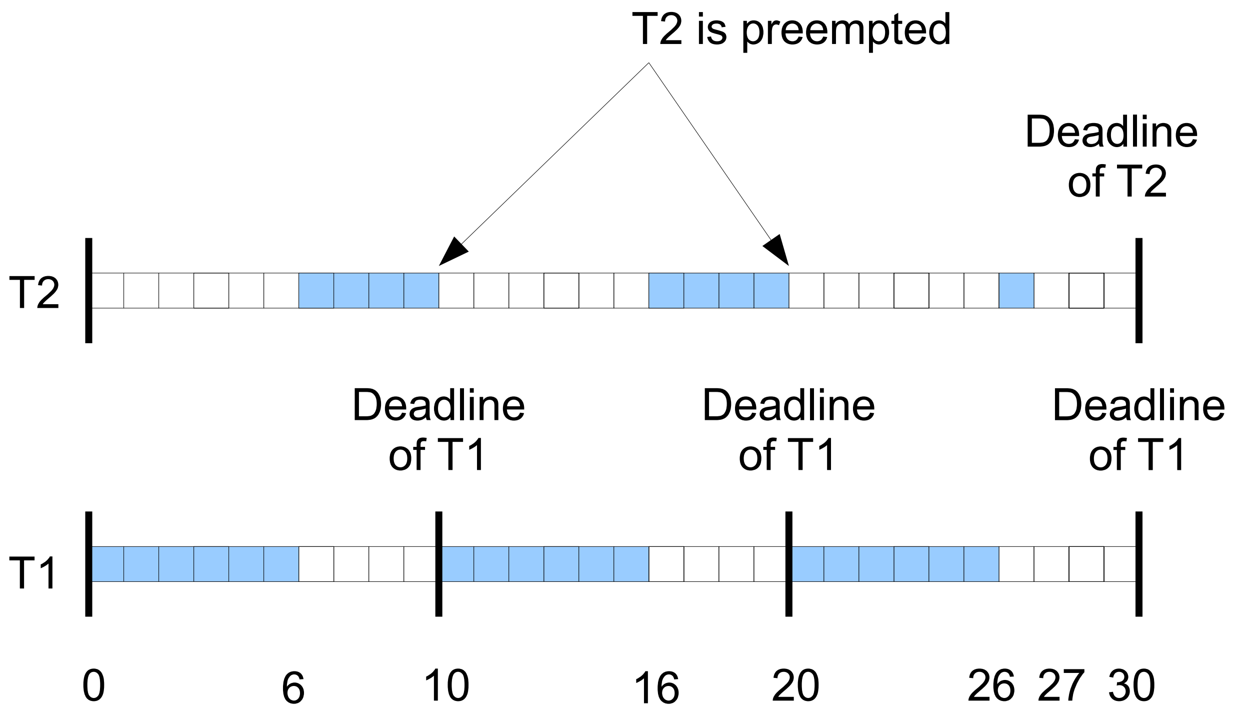
Fixed priority scheduling (2)

- **How does it work:**

1. **"Rate monotonic" task priority assignment:** the highest priority tasks have the smallest periods. Priorities are assigned off-line (e.g. at design time, before execution).
2. **Fixed priority scheduling:** at any time, run the ready task which has the highest priority level.

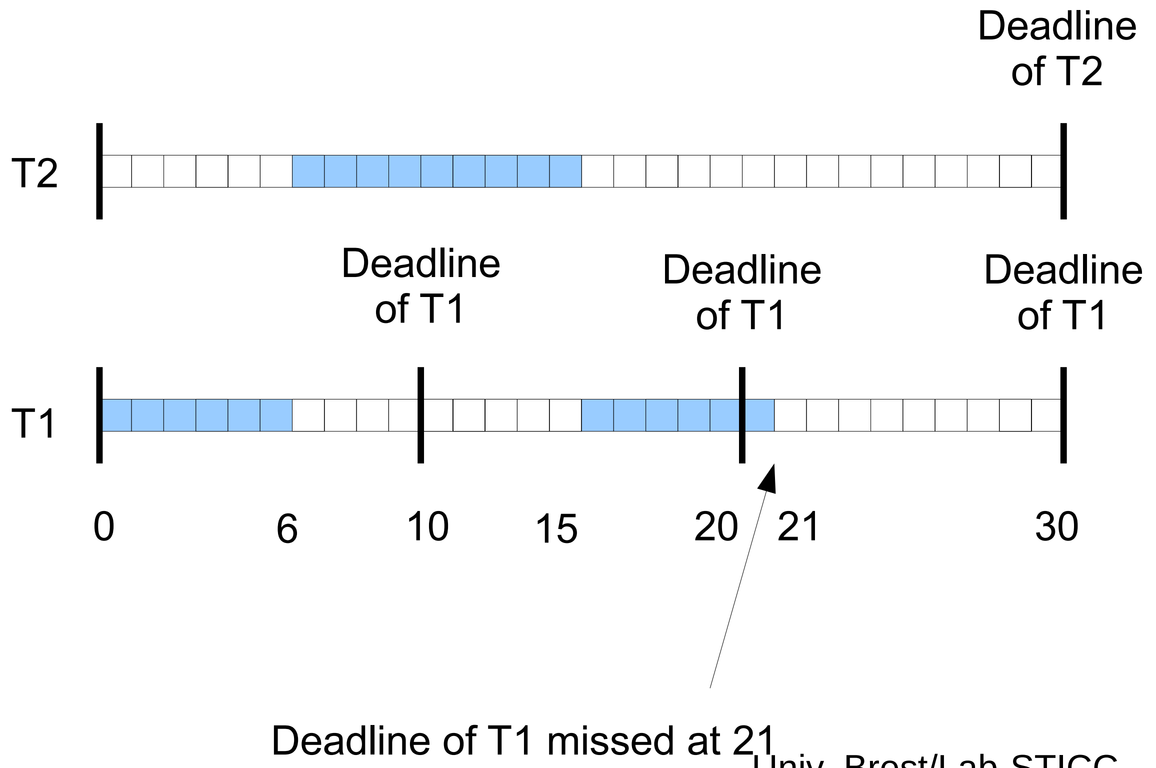
Fixed priority scheduling (3)

- Rate Monotonic assignment and preemptive scheduling:
  - Assuming VxWorks priority levels (high=0 ; low=255)
  - T1 : C1=6, P1=10, Prio1=0
  - T2 : C2=9, P2=30, Prio2=1



Fixed priority scheduling (4)

- Rate Monotonic assignment and non preemptive scheduling:
  - Assuming VxWorks priority levels (high=0 ; low=255)
  - T1 : C1=6, P1=10, Prio1=0
  - T2 : C2=9, P2=30, Prio2=1



Fixed priority scheduling (5)

- **Feasibility/Schedulability tests:**

1. **Run simulations on hyperperiod** = $[0, LCM(P_i)]$. Sufficient and necessary (exact result). Any priority assignment and preemptive/non preemptive scheduling.

2. **Processor utilization factor test**:

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

   Rate Monotonic assignment and preemptive scheduling. Sufficient but not necessary. Does not compute an exact result.

3. **Task worst case response time, noted** r_i $\Longrightarrow$ delay between task release time and task end time. Can compute an exact result. Any priority assignment and preemptive scheduling.

Fixed priority scheduling (6)

- **Compute r_i, the task worst case response time:**
  - Assumptions: preemptive scheduling, synchronous periodic tasks.
  - task i response time $=$ task i capacity + delay the task i has to wait due to higher priority task j. Or:

$$r_i = C_i + \sum_{j \in hp(i)} WaitingTime_j$$

or:

$$r_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{r_i}{P_j} \right\rceil C_j$$

  - $hp(i)$ is the set of tasks which have a higher priority than task i. $\lceil x \rceil$ returns the smallest integer not smaller than x.

Fixed priority scheduling (7)

- **To compute task response time**: compute w_i^k with:

$$w_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{P_j} \right\rceil C_j$$

- Start with $w_i^0 = C_i$.
- Compute w_i^1, w_i^2, w_i^3, ... w_i^k upto:
    - If $w_i^k > P_i$. No task response time can be computed for task i. Deadlines will be missed !
    - If $w_i^k = w_i^{k-1}$. w_i^k is the task i response time. Deadlines will be met.

Fixed priority scheduling (8)

- **Example:** T1 (P1=7, C1=3), T2 (P2=12, C2=2), T3 (P3=20, C3=5)

  - $w_1^0 = C1 = 3 \Longrightarrow r_1 = 3$
  - $w_2^0 = C2 = 2$
  - $w_2^1 = C2 + \left\lceil \frac{w_2^0}{P1} \right\rceil C1 = 2 + \left\lceil \frac{2}{7} \right\rceil 3 = 5$
  - $w_2^2 = C2 + \left\lceil \frac{w_2^1}{P1} \right\rceil C1 = 2 + \left\lceil \frac{5}{7} \right\rceil 3 = 5 \Longrightarrow r_2 = 5$
  - $w_3^0 = C3 = 5$
  - $w_3^1 = C3 + \left\lceil \frac{w_3^0}{P1} \right\rceil C1 + \left\lceil \frac{w_3^0}{P2} \right\rceil C2 = 5 + \left\lceil \frac{5}{7} \right\rceil 3 + \left\lceil \frac{5}{12} \right\rceil 2 = 10$
  - $w_3^2 = 5 + \left\lceil \frac{10}{7} \right\rceil 3 + \left\lceil \frac{10}{12} \right\rceil 2 = 13$
  - $w_3^3 = 5 + \left\lceil \frac{13}{7} \right\rceil 3 + \left\lceil \frac{13}{12} \right\rceil 2 = 15$
  - $w_3^4 = 5 + \left\lceil \frac{15}{7} \right\rceil 3 + \left\lceil \frac{15}{12} \right\rceil 2 = 18$
  - $w_3^5 = 5 + \left\lceil \frac{18}{7} \right\rceil 3 + \left\lceil \frac{18}{12} \right\rceil 2 = 18 \Longrightarrow r_3 = 18$

Fixed priority scheduling (9)

- **Analysis of the car with embedded software example:**

1. $Tdisplay$: task which displays speed. P=100, C=20.
2. $Tspeed$: task which reads speed sensor. P=250, C=50.
3. $Tengine$: task which runs an engine monitoring program. P=500, C=150.

- **Processor utilization test:**

$U = \sum_{i=1}^{n} \frac{C_i}{P_i} = 20/100 + 150/500 + 50/250 = 0.7$

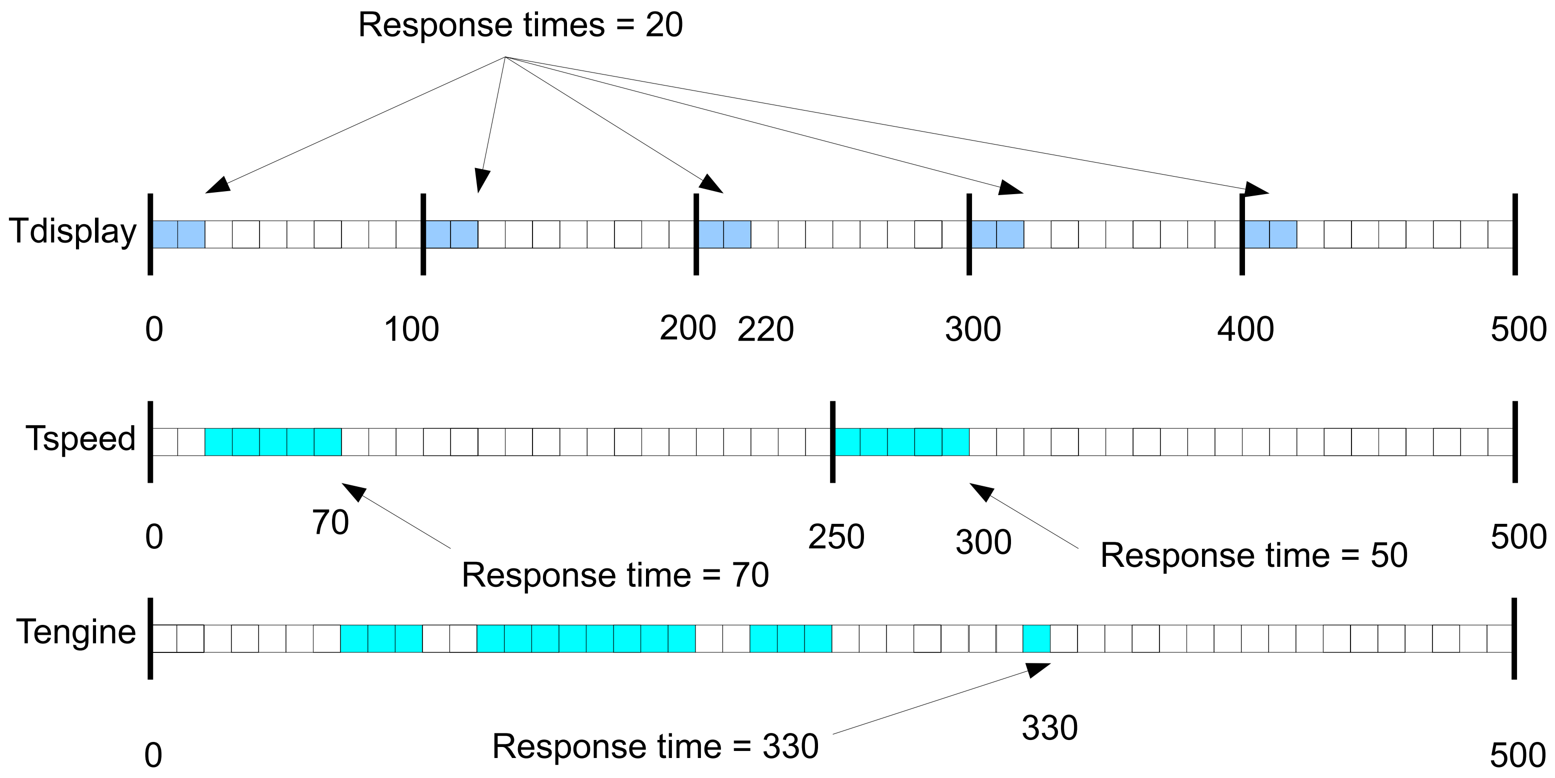
$Bound = n(2^{\frac{1}{n}} - 1) = 3(2^{\frac{1}{3}} - 1) = 0.779$

$U \leq Bound \implies$ deadlines will be met.

- **Task response time:** $r_{Tengine} = 330$, $r_{Tdisplay} = 20$, $r_{Tspeed} = 70$.

Fixed priority scheduling (11)

- **... and check on the computed scheduling**



- Run simulations on **scheduling period** = $[0, LCM(P_i)]$ = $[0, 500]$.

Earliest Deadline First (1)

- **Assumptions and properties:**
  - Dynamic priority scheduler $\Longrightarrow$ suitable for dynamic real-time systems.
  - Is able to schedule both aperiodic and periodic tasks.
  - Optimal scheduler: can reach 100 % of the cpu usage.
  - But difficult to implement into a real-time operating system.
  - Becomes unpredictable if the processor is over-loaded $\Longrightarrow$ not suitable for hard-critical real-time systems.

Earliest Deadline First (2)

- **How does it work:**

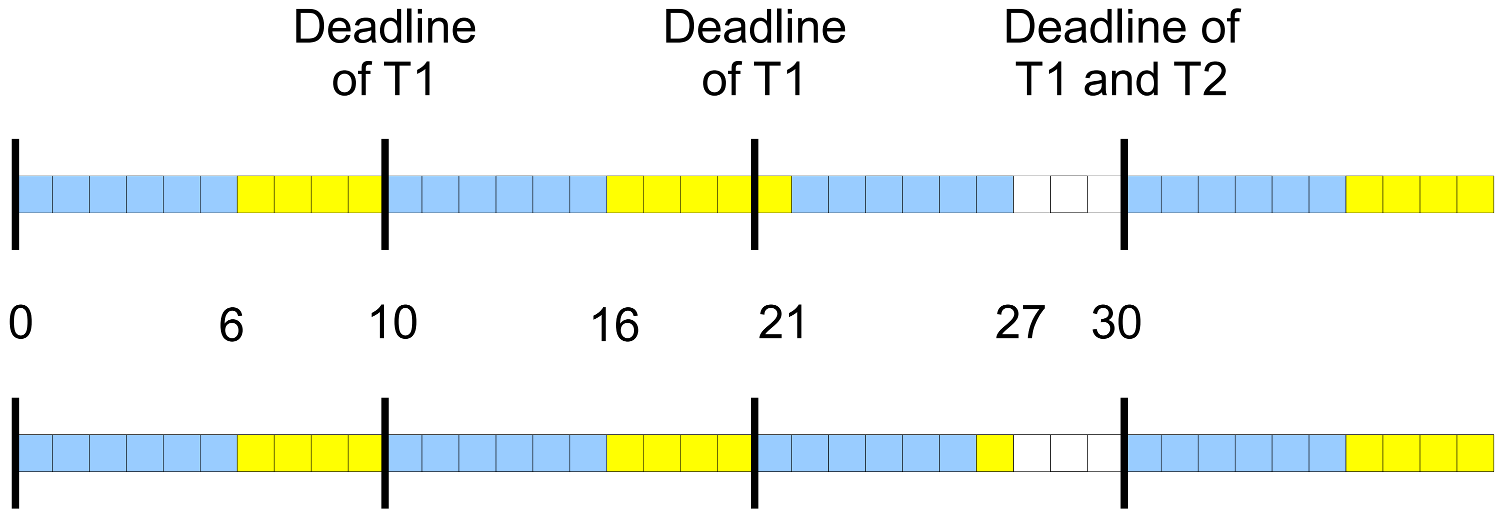
1. **Compute task priorities (called "dynamic deadlines")** $\Longrightarrow D_i(t)$ is the priority/dynamic deadline of task i at time t:
   - Aperiodic task : $D_i(t) = D_i + S_i$.
   - Periodic task: $D_i(t) = k + D_i$, where k is the task release time before t.

2. **Select the task:** at any time, run the ready task which has the shortest dynamic deadline.

Earliest Deadline First (3)

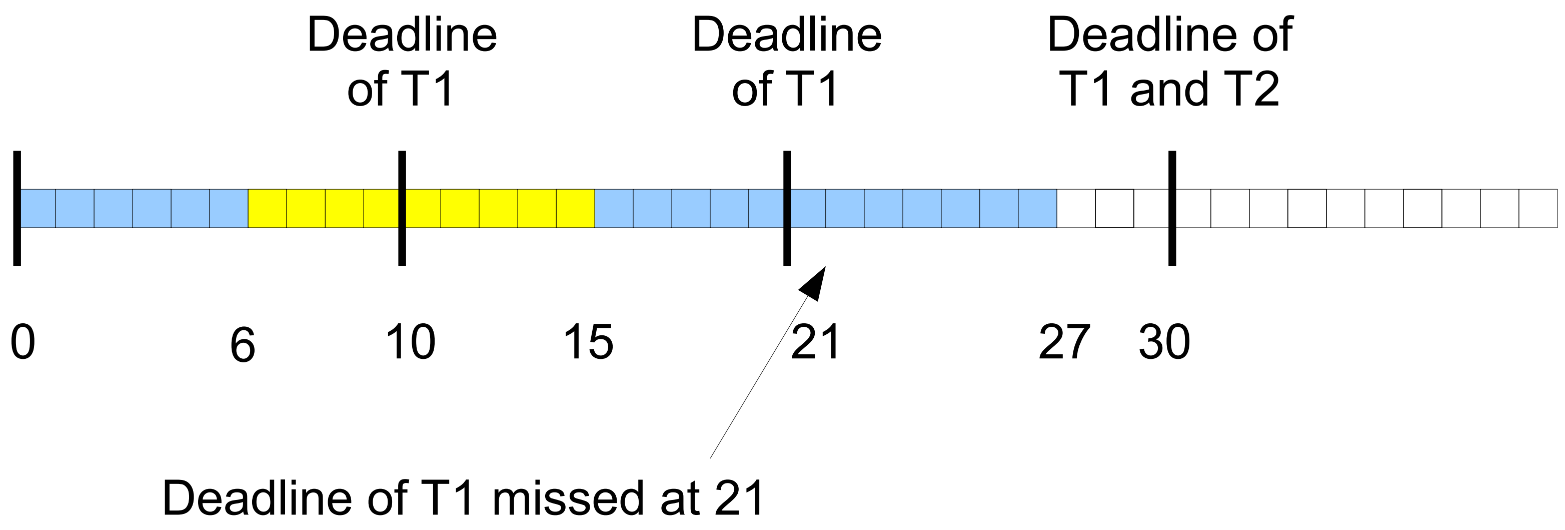
- Preemptive case: (T1/blue, T2/yellow, C1=6; P1=10; C2=9; P2=30)

t	$D_1(t)$	$D_2(t)$
0..9	$k + D_1 = 0 + 10 = 10$	$k + D_2 = 0 + 30 = 30$
10..19	$k + D_1 = 10 + 10 = 20$	30
20..29	$k + D_1 = 20 + 10 = 30$	30



Earliest Deadline First (4)

- Non preemptive case:



Earliest Deadline First (5)

- **Feasibility tests/schedulability tests:**

1. Run simulations on **hyperperiod** = $[0, LCM(P_i)]$. Sufficient and necessary (exact result).

2. **Processor utilization factor test** (e.g. preemptive case):

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

   Sufficient and necessary. Difficult to use in real life applications. Compute an exact result.

3. **Task response time:** a bit more complex to compute (dynamic scheduler) !

EDF vs RM: summary [BUT 03]

- Aperiodic task = non critical task.
- Periodic task = critical task.

	FP/RM	EDF
Applications	critical, static	dynamic, less critical
RTOS implementation	easy	more difficult
Tasks	Periodic only	Aperiodic and periodic
Efficiency	upto 69 %	upto 100 %
Predictability	high	less than FP/RM if $U > 1$

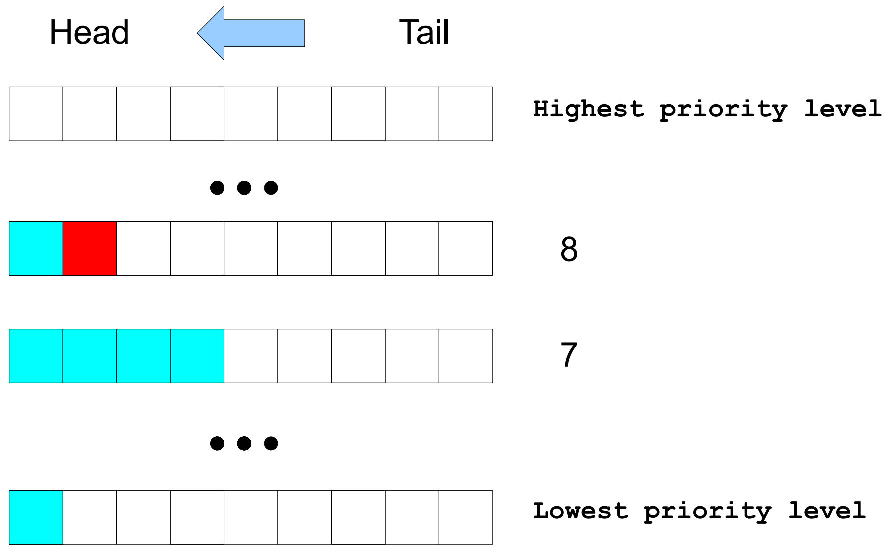
- Warning: the real picture is more complicated: aperiodic tasks scheduling with FP/RM, U=1 with FP/RM, ...

POSIX 1003.1b standard (1)

- POSIX 1003.1b [GAL 95] = POSIX extensions to implement real-time application.

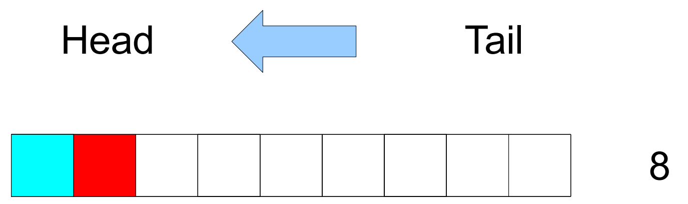
- **Scheduling model:**
  - Preemptive fixed priority scheduling. At least 32 priority levels.
  - Several scheduling policies to cope with same priority tasks.
  - Two-levels scheduling.

POSIX 1003.1b standard (2)



- **Two-levels scheduling:**
  1. Choose the queue which has the highest priority level with at least one ready task.
  2. Choose a task from the queue selected in (1) according to a **policy**.

POSIX 1003.1b standard (3)



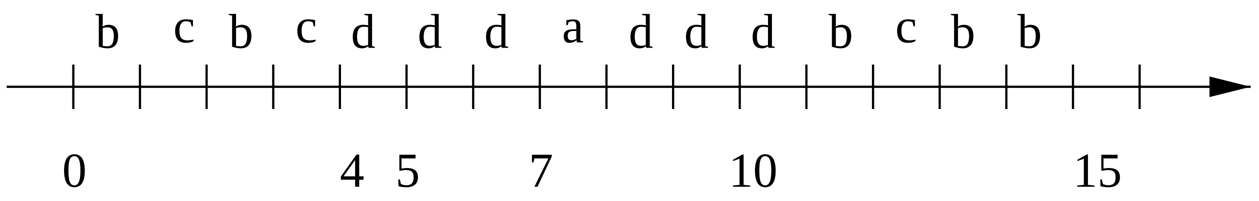
- **POSIX policies:**

1. $SCHED\_FIFO$: similar to the $FIFO\_Within\_Priorities$. Ready tasks of a given priority level get the processor according to their order in the queue.
2. $SCHED\_RR$: $SCHED\_FIFO$ with a time quantum. A time quantum is a maximum duration that a task can run on the processor before preemption by an other task of the same queue. When the quantum is exhausted, the preempted task is moved to the tail of the queue.
3. $SCHED\_OTHER$: implementation defined (usually implements a time sharing scheduler).

POSIX 1003.1b standard (4)

- Example:

Tasks	C_i	S_i	Priority	Policy
a	1	7	1	FIFO
b	5	0	4	RR
c	3	0	4	RR
d	6	4	2	FIFO



- Quantum SCHED_RR = 1 unit of time
- Higher priority level: 1

POSIX 1003.1b standard (5)

- POSIX policies:

```
#define SCHED_OTHER     0
#define SCHED_FIFO      1
#define SCHED_RR        2
```

- Read system properties/capabilities:

```
int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);
int sched_rr_get_interval(pid_t pid,
       struct  timespec *tp);
```

- Release processor:

```
int sched_yield(void);
```

POSIX 1003.1b standard (6)

- Scheduling parameters:

```
struct sched_param
  {
    int sched_priority;
    ...
  };
```

- Read or update scheduling policy:

```
int sched_setscheduler(pid_t pid, int policy,
        const struct sched_param *p);
int sched_getscheduler(pid_t pid);
```

- Read or update scheduling parameter:

```
int  sched_getparam(pid_t  pid,
        struct sched_param *p);
int   sched_setparam(pid_t   pid,
        const struct sched_param *p);
```

POSIX 1003.1b standard (6)

- Example of slide number 15

```
struct sched_param parm;
int res=-1;
...
/* Task T1 ; P1=10 */
parm.sched_priority=15;
res=sched_setscheduler(pid_T1,SCHED_FIFO,&parm);
if(res<0)
     perror("sched_setscheduler task T1");

/* Task T2 ; P2=30 */
parm.sched_priority=10;
res=sched_setscheduler(pid_T2,SCHED_FIFO,&parm);
if(res<0)
     perror("sched_setscheduler task T2");
```

Summary

Shared resources (1)

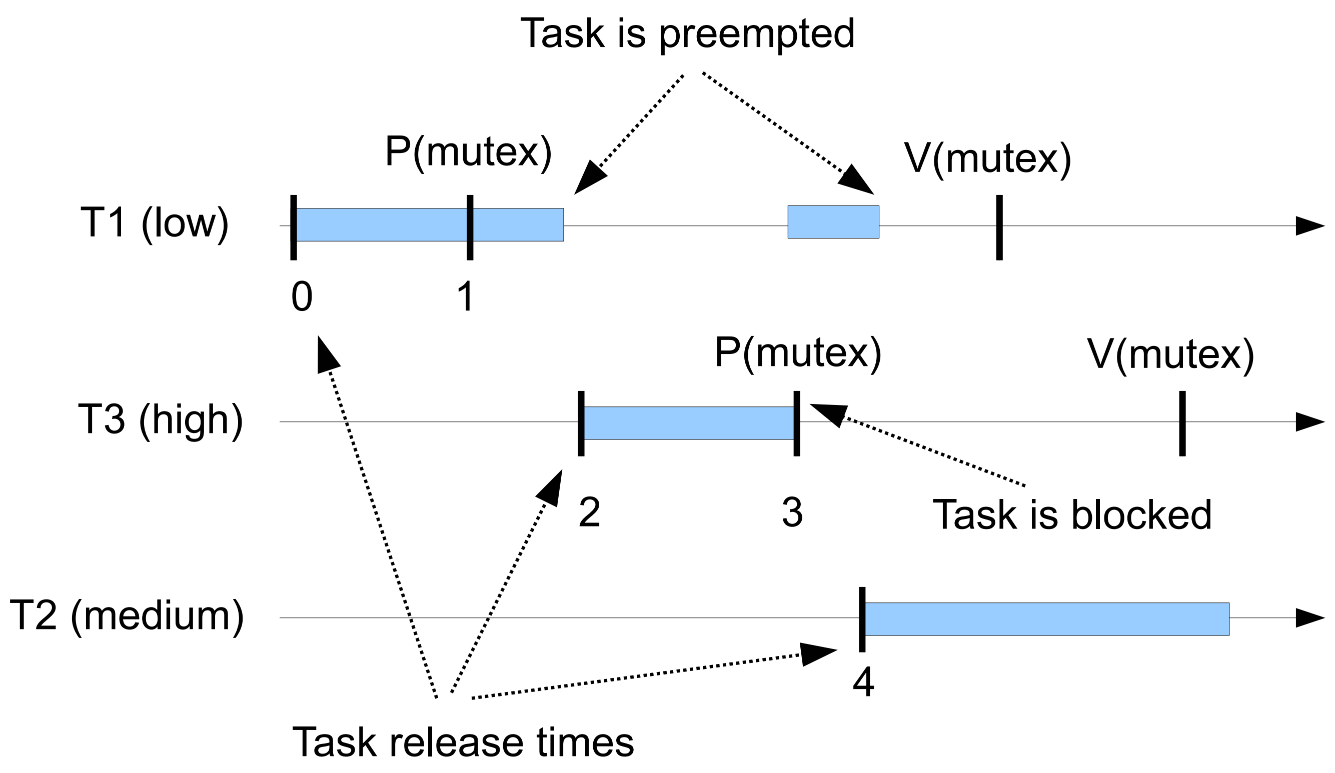
- **In real-time scheduling theory, a shared resource is modeled by a semaphore:**

```
Task i                          Task j
P(sem)                          P(sem)
access_shared_resource(k)       access_shared_resource(k)
V(sem)                          V(sem)
```

```
P(s)                                        | V(s)
     {                                      |      {
     s.cpt=s.cpt−1;                         |      s.cpt=s.cpt+1;
     if (s.cpt<0)                           |      if (s.cpt<=0)
          {                                 |           {
          add_task_to_the_queue(s.file);    |           remove_task_from_the_queue(s.file)
          task_is_blocked();                |           task_is_released();
          }                                 |           }
     }                                      |      }
```

- Semaphore = counter + a FIFO queue.
- A semaphore may be used to model a critical section.
- We use special semaphores in real-time scheduling systems.

Shared resources (2)



- **What is Priority inversion:** a low priority task blocks a high priority task ... allowing a medium priority task to held the processor $\Longrightarrow$ a non critical task runs before a critical task !
- B_i = bound on the shared resource waiting time.

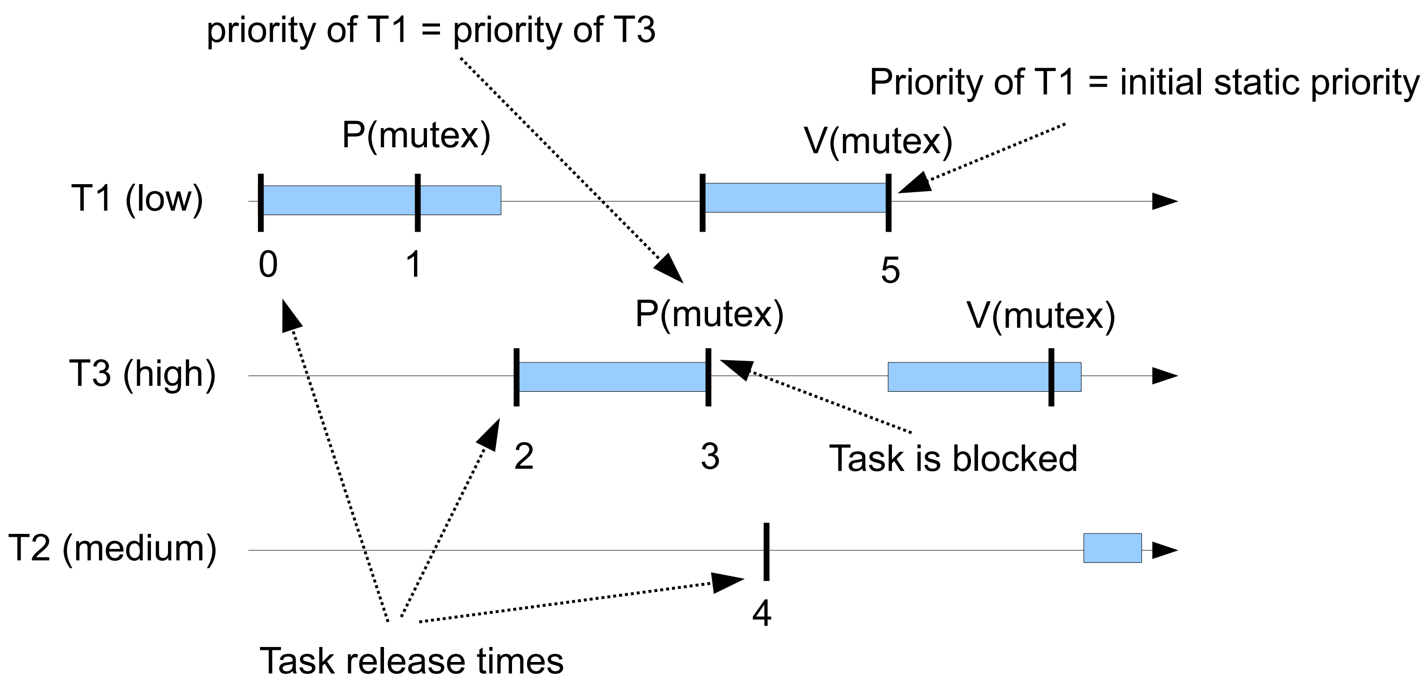
Shared resources (3)

- How to reduce priority inversion?
- How long a task must wait for the access to a shared resource? How to compute B_i?

$\Longrightarrow$ **To reduce priority inversion, we use priority inheritance.**

• Priority inheritance protocols provide a specific implementation of $P()$ and $V()$ of semaphores.

Shared resources (4)

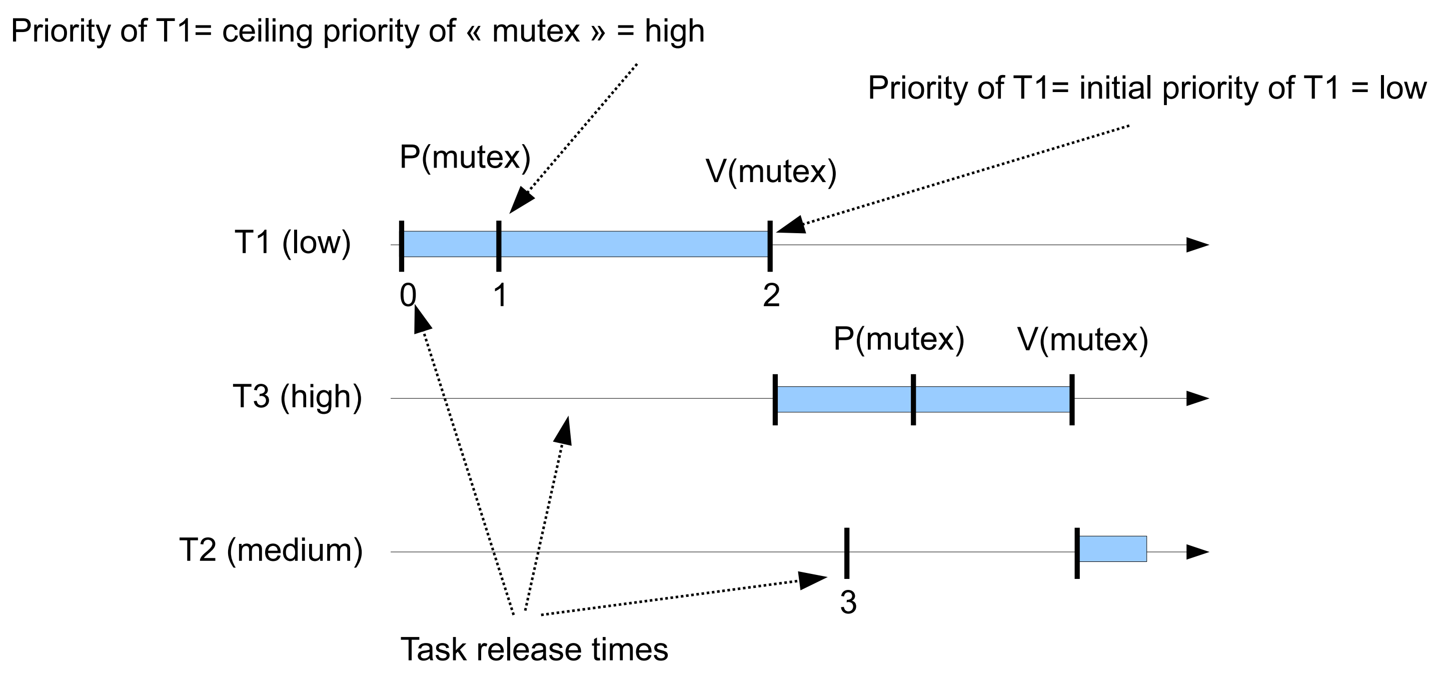


- **Priority Inheritance Protocol or PIP:**
  - A task which blocks a high priority task due to a critical section, sees its priority to be increased to the priority level of the blocked task.
  - $B_i =$ sum of the critical sections of the tasks which have a priority smaller than i.

Shared resources (5)

- PIP can not be used with more than one shared resource due to deadlock.
- PCP (Priority Ceiling Protocol) [SHA 90] is used instead.
- Implemented in most of real-time operating systems (e.g. VxWorks).
- Several implementations of PCP exists: OPCP, ICPP, ...

Shared resources (6)


Priority of T1= ceiling priority of « mutex » = high
Priority of T1= initial priority of T1 = low
P(mutex)
V(mutex)
T1 (low)
0
1
2
P(mutex)
V(mutex)
T3 (high)
T2 (medium)
3
Task release times

- **ICPP (Immediate Ceiling Priority Protocol):**
  - Ceiling priority of a resource = maximum static priority of the tasks which use it.
  - Dynamic task priority = maximum of its own static priority and the ceiling priorities of any resources it has locked.

Shared resources (7)

- **How to take into account the blocking time B_i with the processor utilization factor test:**

  - Preemptive RM feasibility test:

$$\forall\, i, 1 \leq i \leq n : \sum_{k=1}^{i-1} \frac{C_k}{P_k} + \frac{C_i + B_i}{P_i} \leq i(2^{\frac{1}{i}} - 1)$$

  - Preemptive EDF feasibility test:

$$\forall\, i, 1 \leq i \leq n : \sum_{k=1}^{i-1} \frac{C_k}{P_k} + \frac{C_i + B_i}{P_i} \leq 1$$

Shared resources (8)

- **How to take into account the blocking time B_i with the worst case response time r_i of the task i (example of a preemptive RM scheduler):**

$$r_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{r_i}{P_j} \right\rceil C_j$$

with $hp(i)$ the set of tasks which has a lowest priority than task i.

- Which can be computed by:

$$w_i^{n+1} = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{P_j} \right\rceil C_j$$

Summary

Conclusion

1. Classical schedulers (Rate Monotonic, Earliest Deadline First) and task models (periodic task).
2. Feasibility tests and early verification: at design step, on architecture models. Can be made automatically.
3. Most of real-time operating systems provide fixed priority scheduling.
4. POSIX standard.
5. Shared resources and inheritance protocols.

Summary

References

[BUT 03] G. Buttazzo. « Rate monotonic vs. EDF: Judgment day ». *n Proc. 3rd ACM International Conference on Embedded Software, Philadephia, USA*, October 2003.

[GAL 95] B. O. Gallmeister. *POSIX 4 : Programming for the Real World* . O’Reilly and Associates, January 1995.

[LIU 73] C. L. Liu and J. W. Layland. « Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environnment ». *Journal of the Association for Computing Machinery*, 20(1):46–61, January 1973.

References

[SHA 90] L. Sha, R. Rajkumar, and J.P. Lehoczky. « Priority Inheritance Protocols : An Approach to real-time Synchronization ». *IEEE Transactions on computers*, 39(9):1175–1185, 1990.