Fixed Priority Scheduling

Real-Time Operative Systems Course

Paulo Pedreiras, DETI/UA/IT

October 17, 2022

Last lecture

- The concept of temporal complexity
- Definition of schedule and scheduling algorithm
- Some basic scheduling techniques (EDD, EDF, BB)
- The static cyclic scheduling technique

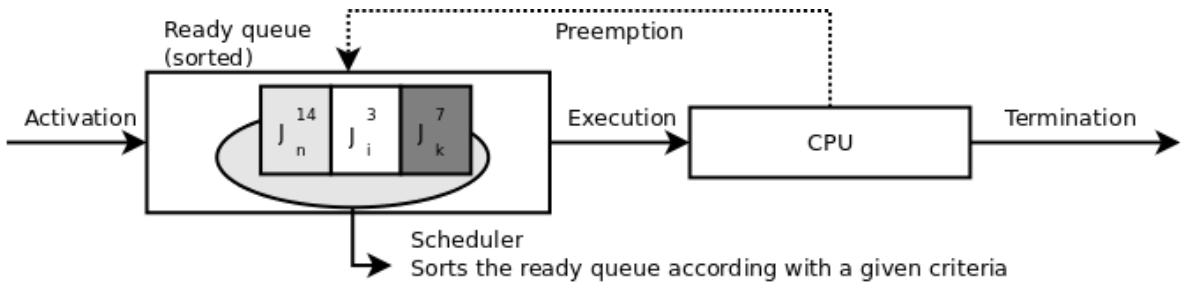
Agenda for today

Fixed-priority online scheduling

- Rate-Monotonic scheduling
- Deadline-Monotonic and arbitrary priorities
- Analysis:
  - The CPU utilization bound
  - Worst-Case Response-Time analysis

Online scheduling with fixed priorities

- The schedule is built while the system is operating normally (**online**) and is based on a **static criterium** (priority)
- The ready queue is **sorted by decreasing priorities** . Executes first the task with highest priority.
- If the system is preemptive, whenever a task job arrives to the ready queue, if it has higher priority than the one currently executing, it starts executing while the latter one is moved to the ready queue
- Complexity: $O(n)$



Online scheduling with fixed priorities

Pros (with respect to Static Cyclic Scheduling)

- Scales
- Changes on the task set are immediately taken into account by the scheduler
- Sporadic tasks are easily accommodated
- Deterministic behavior on overloads
  - Tasks are affected by priority level (lower priority are the first ones)

Cons (with respect to Static Cyclic Scheduling)

- More complex implementation (w.r.t. static cyclic scheduling)
- Higher execution overhead (scheduler + dispatcher)
- On overloads (e.g. due to SW errors or unpredicted events) an higher priority tasks may block the execution of lower priority ones

Online scheduling with fixed priorities

Rules for priority assignment to tasks

- Inversely proportional to period (RM – Rate Monotonic)
  - Optimal among fixed priority scheduling criteria if D=T
- Inversely proportional to deadline (DM – Deadline Monotonic)
  - Optimal if $D \leq T$
- Proportional to the task importance
  - Typically **reduces the schedulability** – not optimal
  - But very common is industry cases – e.g. automotive CAN

Online scheduling with fixed priorities

Schedulability tests

- As the schedule is built online it is fundamental to know a priori if a given task set is schedulable (i.e., its temporal requirements are met)
- There are two basic types of schedulability tests:
  - Based on CPU utilization rate
  - Based on response time

RM Scheduling

Schedulability tests for RM based on task utilization

- Valid with preemption, n independent tasks, D=T
- Liu&Layland's (1973), **Least Upper Bound (LUB)**

$$U(n) = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n \cdot (2^{\frac{1}{n}} - 1) \Rightarrow \text{One execution per period guaranteed}$$

- Bini&Buttazzo&Buttazzo's (2001), **Hyperbolic Bound**

$$\prod_{i=1}^{n} (\frac{C_i}{T_i} + 1) \leq 2 \Rightarrow \text{One execution per period guaranteed}$$

RM Scheduling

Interpretation of the Liu&Layland test

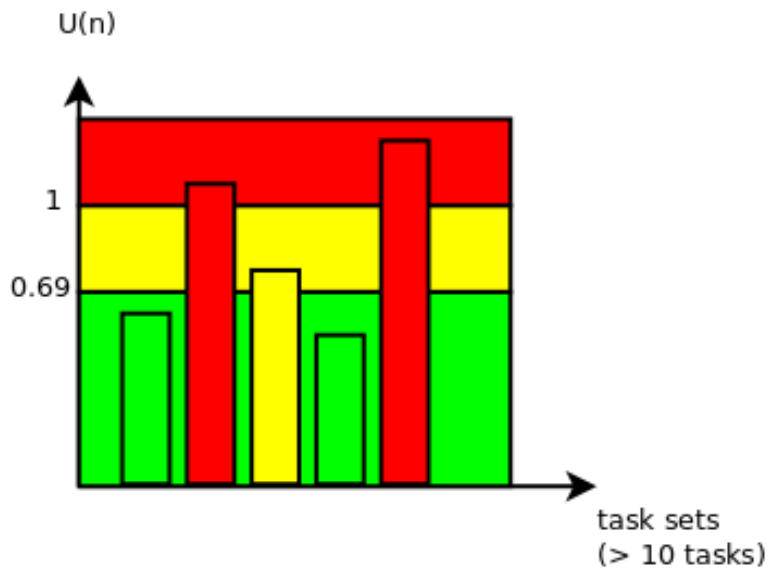
$U(n) > 1 \Rightarrow$ task set not schedulable (overload) – **necessary condition**

$U(n) \leq U_{lub} \Rightarrow$ task set is schedulable – **sufficient condition**

$1 \geq U(n) \geq U_{lub} \Rightarrow$ the test is **indeterminate**

Some U_{lub} values

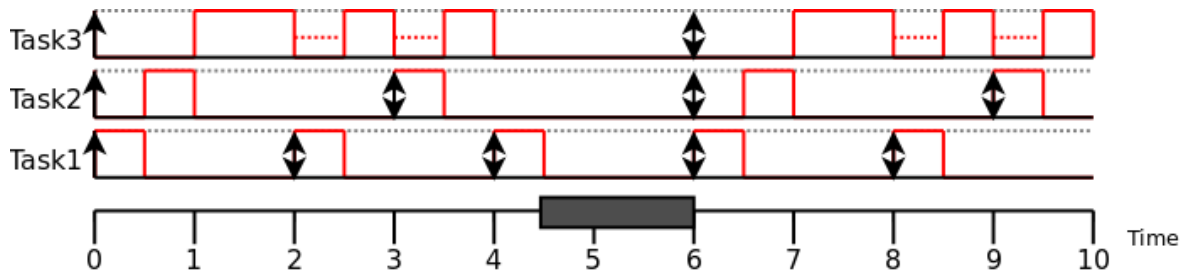
- $U_{lub}(1) = 1$
- $U_{lub}(2) = 0.83$
- $U_{lub}(3) = 0.78$
- ...
- $U_{lub}(n) \rightarrow ln(2) \approx 0.69$



RM Scheduling – example 1

Task properties

τ	C	T
1	0.5	2
2	0.5	3
3	2	6



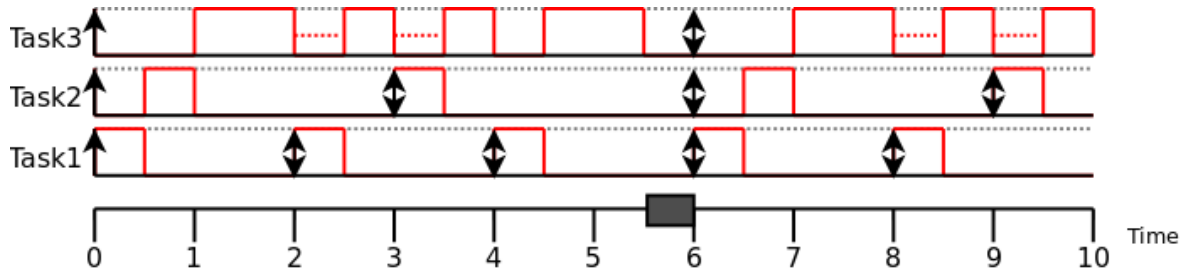
- $U = \frac{0.5}{2} + \frac{0.5}{3} + \frac{2}{6} \approx 0.75$
- $U_{lub}(3) = 0.78$. As $0.75 < 0.78$ one execution per period is guaranteed

RM Scheduling – example 2

Task properties

Same task set, but C_3 increases from 2 to 3 tu.

τ	C	T
1	0.5	2
2	0.5	3
3	3	6

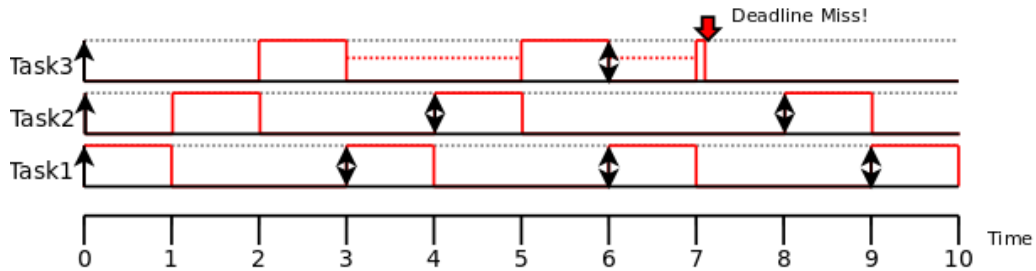


- $U = \frac{0.5}{2} + \frac{0.5}{3} + \frac{3}{6} \approx 0.92$
- $U_{lub}(3) = 0.78$. As $0.92 > 0.78$ one execution per period is **NOT guaranteed**
- But the task set is schedulable (see Gantt chart)

RM Scheduling – example 3

Task set properties

τ	C	T
1	1	3
2	1	4
3	2.1	6

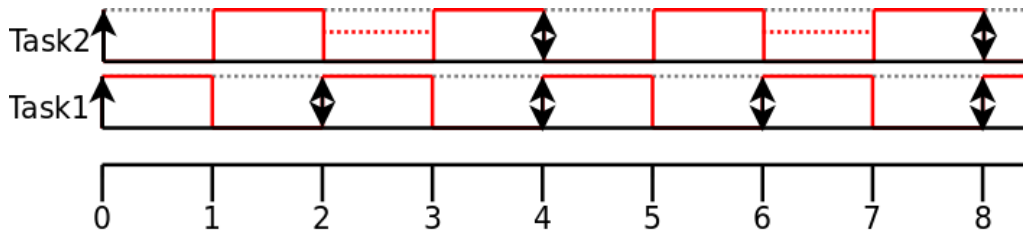


- $U = \frac{1}{3} + \frac{1}{4} + \frac{2.1}{6} \approx 0.93$
- $U_{lub}(3) = 0.78$. As $0.93 > 0.78$ one execution per period is **NOT guaranteed**
- And the task set indeed is **not schedulable** (see Gantt chart)

RM Scheduling – Harmonic Periods

Particular case: if the task **periods are harmonic** then the task set is schedlable iif $U(n) \leq 1$

- E.g. $\Gamma = \{(1,2);(2,4)\}$



Deadline Monotonic Scheduling DM)

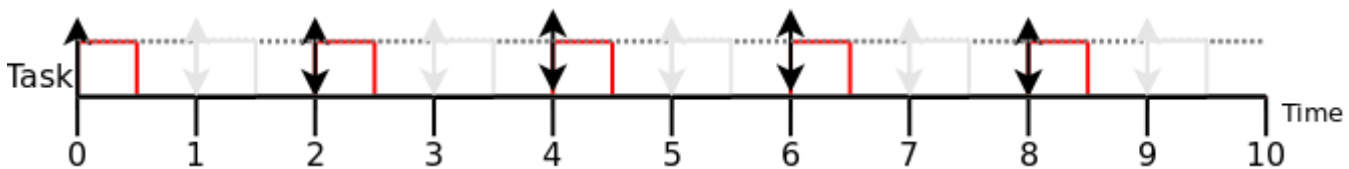
Schedulability tests for DM

- In some cases tasks may have large periods (low frequency) but require a short response time.
- In these cases we assign a deadline shorter than the period, and the scheduling criteria is the deadline.
- It is possible to use utilization-based tests.
  - The adaptation is simple, but the test is **very pessimistic** .

Utilization-based test

$$\sum_{i=1}^{n}\left(\frac{C_i}{D_i}\right) \leq n\left(2^{\frac{1}{n}} - 1\right)$$

E.g. $\{C = 0.5, T = 2, D = 1\}$ (Assumed utilization is doubled)



Response-time analysis

- For arbitrary fixed priorities, including RM, DM and others, the **Response Time Analysis** is an **exact** test (i.e., a necessary and sufficient condition) in the following conditions:
  - full preemption, synchronous release, independent tasks and $D \leq T$
- Worst-case response time (WCRT, Rwc, R,...) = maximum time interval between arrival and finish instants.
  - $Rwc_i = max_k(f_{i,k} - a_{i,k})$

Schedulability test based on the WCRT

$Rwc_i < D_i, \forall i \Leftrightarrow$ task set is schedulable

Response-time analysis

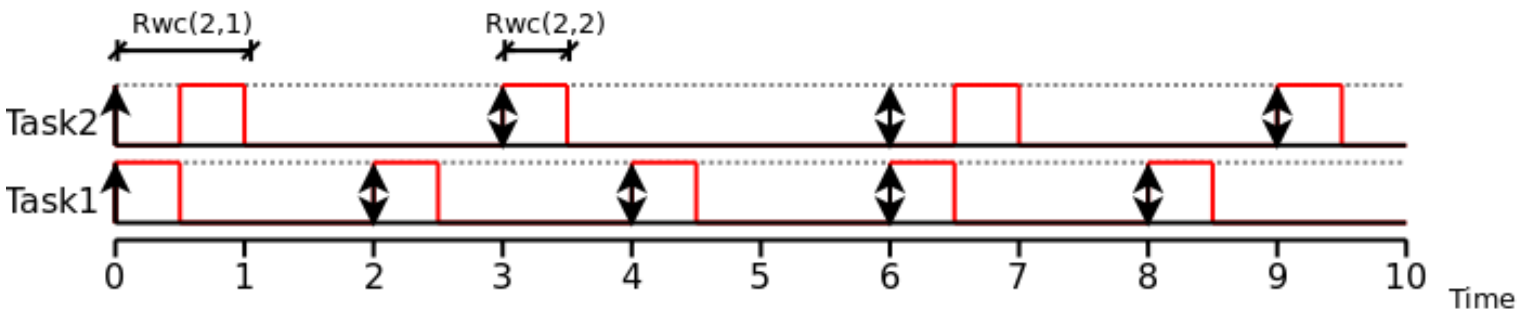
- The WCRT of a given task occurs when the task is activated at the same time as all other high-priority tasks (critical instant)
- I_i – interference caused by the execution of higher priority tasks

Computing Rwc_i

$$\forall i, Rwc_i = I_i + C_i$$

$$I_i = \sum_{k \in hp(i)} \lceil \frac{Rwc_i}{T_k} \rceil \cdot C_k$$

$\lceil \frac{Rwc_i}{T_k} \rceil$ represents the number of activations of hp task "k"



Response-time analysis

The equation is solved iteratively. Stop conditions are:

- A **deadline is violated**
  - $Rwc_i(m) > D_i$
- **Convergence** (two successive iterations yield the same result)
  - $Rwc_i(m+1) = Rwc_i(m)$

Algorithm

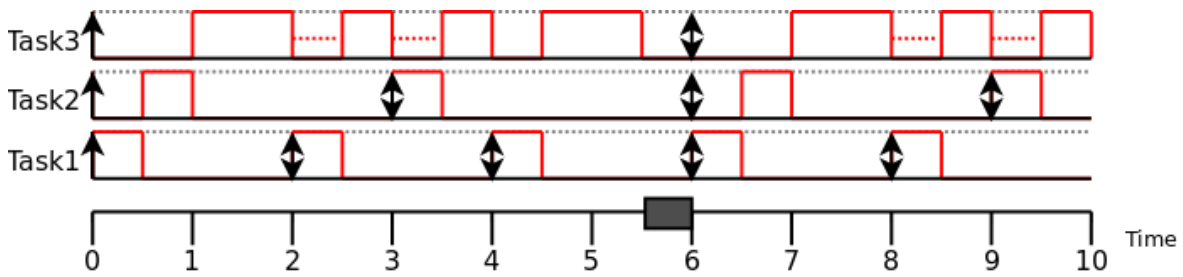
1. $Rwc_i(0) = \sum_{k \in hp(i)} (C_k) + C_i$
2. $Rwc_i(1) = \sum_{k \in hp(i)} \lceil \frac{Rwc_i(0)}{T_k} \rceil \cdot C_k + C_i$
3. ...
4. $Rwc_i(m) = \sum_{k \in hp(i)} \lceil \frac{Rwc_i(m-1)}{T_k} \rceil \cdot C_k + C_i$

Repeat Step 3 until convergence or deadline violation

Response-time analysis

Example:

τ	C	T
1	0.5	2
2	0.5	3
3	3	6

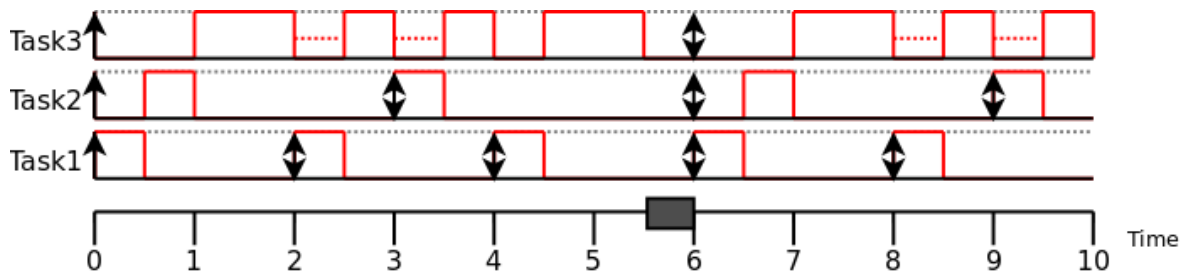


- $Rwc_1 = ?$

Response-time analysis

Example:

τ	C	T
1	0.5	2
2	0.5	3
3	3	6

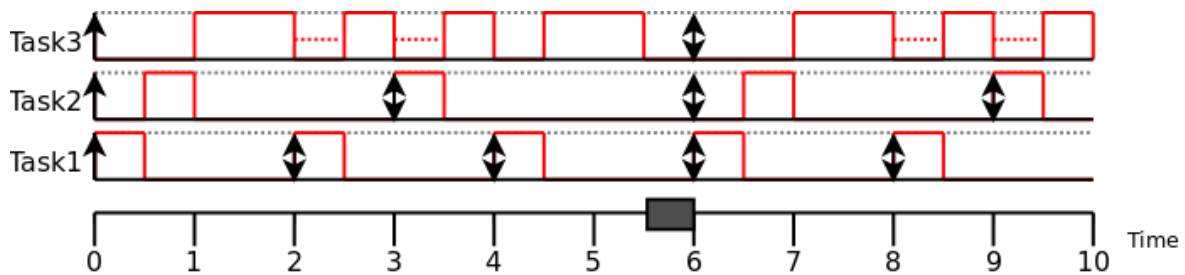


- $Rwc_1 = ?$
  - $Rwc_1(0) = C_1 = 0.5tu$
- $Rwc_2 = ?$

Response-time analysis

Example:

τ	C	T
1	0.5	2
2	0.5	3
3	3	6



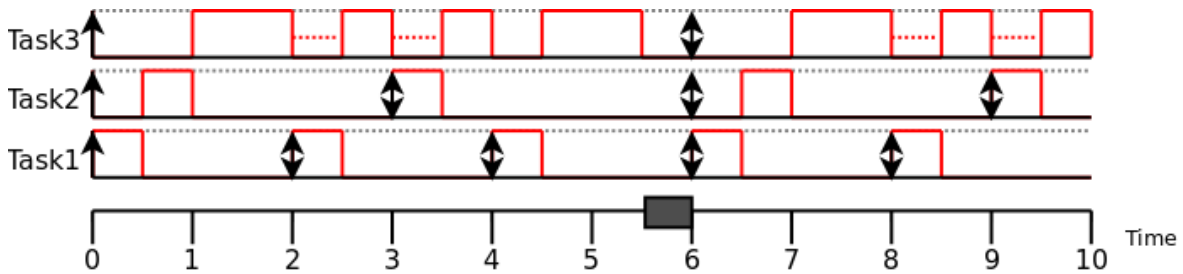
- $Rwc_1 = ?$
  - $Rwc_1(0) = C_1 = 0.5tu$
- $Rwc_2 = ?$
  - $Rwc_2(0) = C_1 + C_2 = 0.5 + 0.5 = 1tu$
  - $Rwc_2(1) = \sum_{k \in \{1\}} \lceil \frac{Rwc_i(0)}{T_k} \rceil \cdot C_k + C_2 = \lceil \frac{1}{2} \rceil \cdot 0.5 + 0.5 = 1tu$
  - **Converged** , thus $Rwc_2 = 1tu$

Response-time analysis

Example:

τ	C	T
1	0.5	2
2	0.5	3
3	3	6

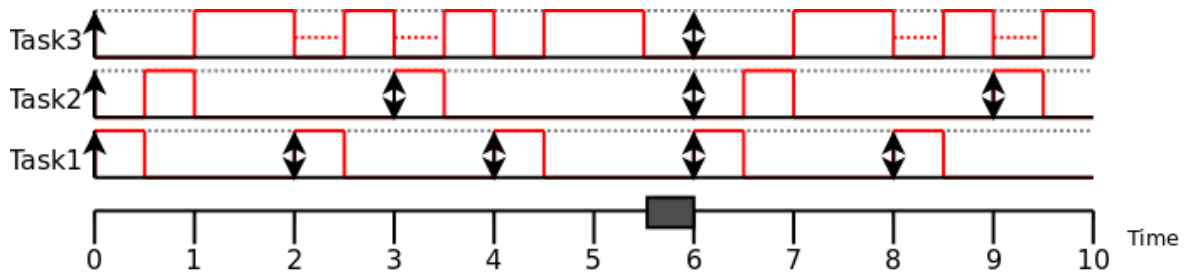
- Rwc_3=?



Response-time analysis

Example:

τ	C	T
1	0.5	2
2	0.5	3
3	3	6

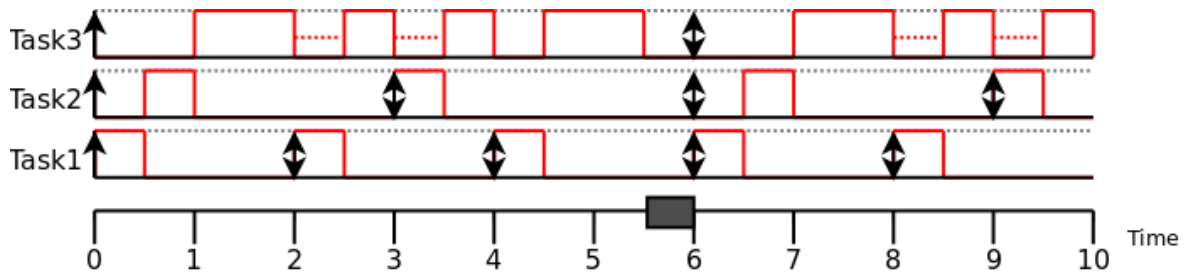


- Rwc_3=?
  - $Rwc_3(0)=C_1+C_2+C_3=4tu$

Response-time analysis

Example:

τ	C	T
1	0.5	2
2	0.5	3
3	3	6

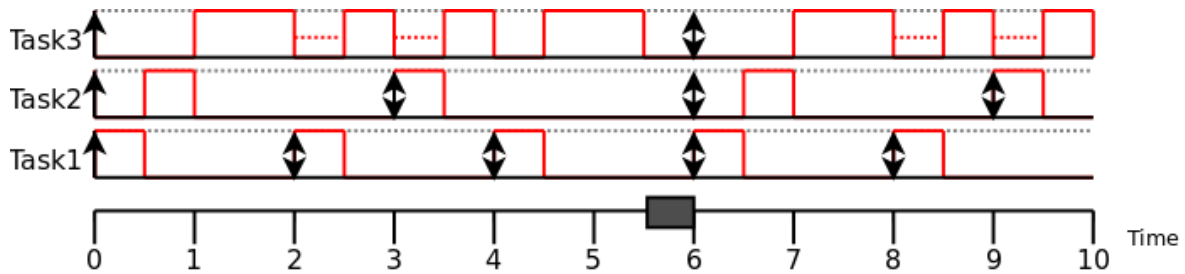


- Rwc_3=?
  - $Rwc_3(0)=C_1+C_2+C_3=4tu$
  - $Rwc_3(1)=\sum_{k\in\{1,2\}}\lceil\frac{Rwc_i(0)}{T_k}\rceil\cdot C_k+C_3=\lceil\frac{4}{2}\rceil\cdot 0.5+\lceil\frac{4}{3}\rceil\cdot 0.5+3=5tu$

Response-time analysis

Example:

τ	C	T
1	0.5	2
2	0.5	3
3	3	6

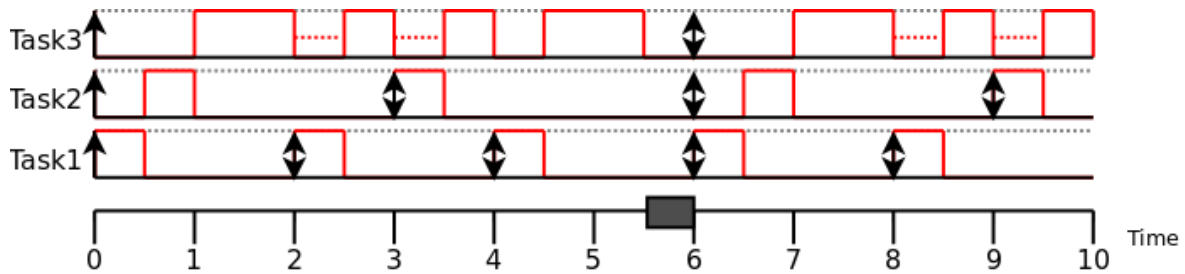


- Rwc_3=?
  - $Rwc_3(0)=C_1+C_2+C_3=4tu$
  - $Rwc_3(1)=\sum_{k\in\{1,2\}}\lceil\frac{Rwc_i(0)}{T_k}\rceil\cdot C_k+C_3=\lceil\frac{4}{2}\rceil\cdot 0.5+\lceil\frac{4}{3}\rceil\cdot 0.5+3=5tu$
  - $Rwc_3(2)=\sum_{k\in\{1,2\}}\lceil\frac{Rwc_i(1)}{T_k}\rceil\cdot C_k+C_3=\lceil\frac{5}{2}\rceil\cdot 0.5+\lceil\frac{5}{3}\rceil\cdot 0.5+3=5.5tu$

Response-time analysis

Example:

τ	C	T
1	0.5	2
2	0.5	3
3	3	6



- Rwc_3=?
  - $Rwc_3(0)=C_1+C_2+C_3=4tu$
  - $Rwc_3(1)=\sum_{k\in\{1,2\}}\lceil\frac{Rwc_i(0)}{T_k}\rceil\cdot C_k+C_3=\lceil\frac{4}{2}\rceil\cdot 0.5+\lceil\frac{4}{3}\rceil\cdot 0.5+3=5tu$
  - $Rwc_3(2)=\sum_{k\in\{1,2\}}\lceil\frac{Rwc_i(1)}{T_k}\rceil\cdot C_k+C_3=\lceil\frac{5}{2}\rceil\cdot 0.5+\lceil\frac{5}{3}\rceil\cdot 0.5+3=5.5tu$
  - $Rwc_3(3)=\sum_{k\in\{1,2\}}\lceil\frac{Rwc_i(2)}{T_k}\rceil\cdot C_k+C_3=\lceil\frac{5.5}{2}\rceil\cdot 0.5+\lceil\frac{5.5}{3}\rceil\cdot 0.5+3=5.5tu$
  - **Converged** , thus $Rwc_3 = 5.5tu$

As $Rwc(i) \leq D_i \forall i$, the task set is schedulable!

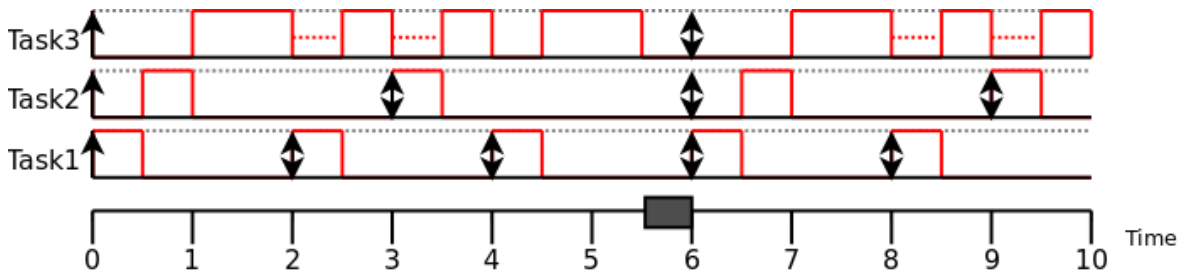
Restrictions to the schedulability tests previously presented

- The previous schedulability tests **must be modified** in the following cases:
  - Non-preemption
  - Tasks not independent
    - Share mutually exclusive resources
    - Have precedence constrains
  - It is also necessary to take into account the overhead of the kernel, because the scheduler, dispatcher and interrupts consume CPU time

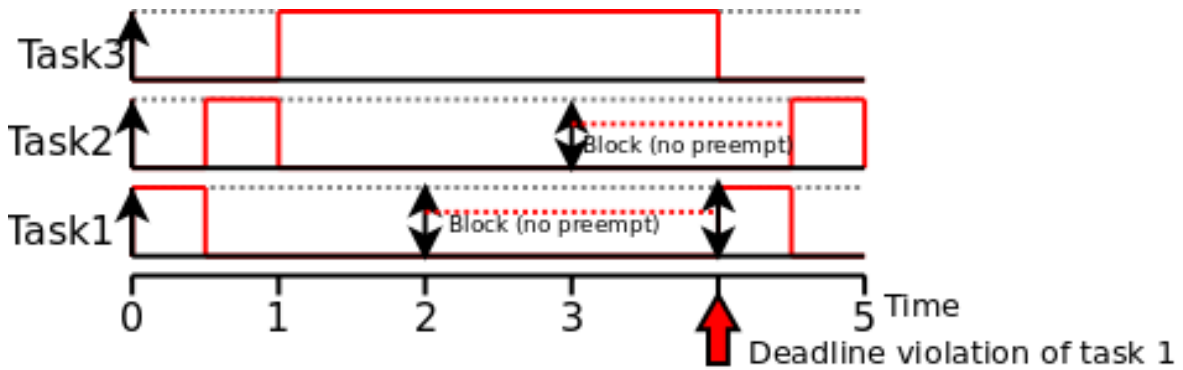
Impact of non-preemption

Example:

τ	C	T
1	0.5	2
2	0.5	3
3	3	6



With preemption



Without preemption

Summary

- On-line scheduling with fixed-priorities
- The Rate Monotonic scheduling policy – schedulability analysis based on utilization
- The Deadline Monotonic and arbitrary deadlines scheduling policies
- Response-time analysis

Scheduling Basics

Real-Time Operative Systems Course

Paulo Pedreiras, DETI/UA/IT

October 10, 2022

Last lecture

- Computation models and Real-Time Kernels and Operating Systems
- Computation models:
  - Tasks with specific temporal constraints;
  - The Event- and Time-Trigger paradigms
- Real-Time OSs and kernels:
  - General architecture
  - Task states
  - Basic components

Agenda for today

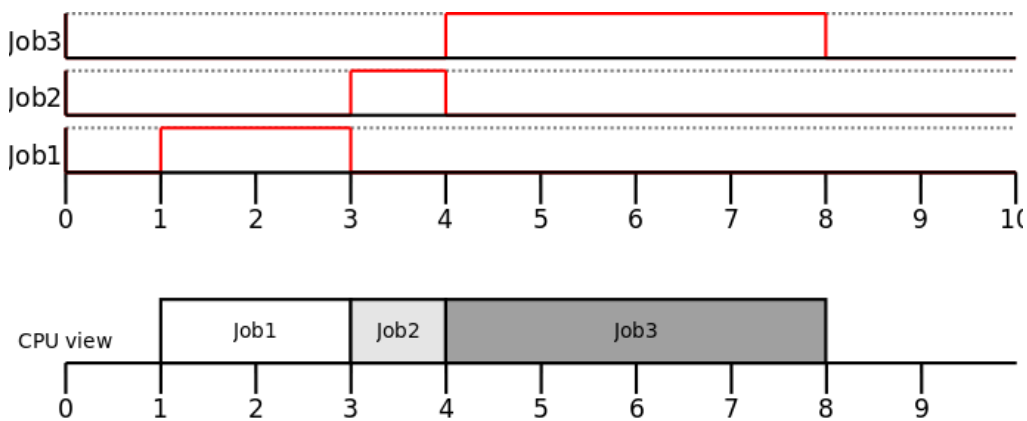
Scheduling basics

- Task scheduling - basic concepts and taxonomy
- Basic scheduling techniques
- Static cyclic scheduling

Scheduling Definition

Task scheduling (also applies to messages, with due adaptations)

- Sequence of task executions (jobs) in one or more processors
- Application of $\mathbb{R}^+$ (time) in $\mathbb{N}_0$ (task set), assigning to each time instant "t" a task/job "i" that executes in that time instant
  $$\sigma(t) : \mathbb{R}^+ \rightarrow \mathbb{N}_0$$
  $i = \sigma(t), t \in \mathbb{R}^+$, i=0 means that the processor is idle
- $\sigma(t)$ is a step function, which can be expressed e.g. as a Gantt graph



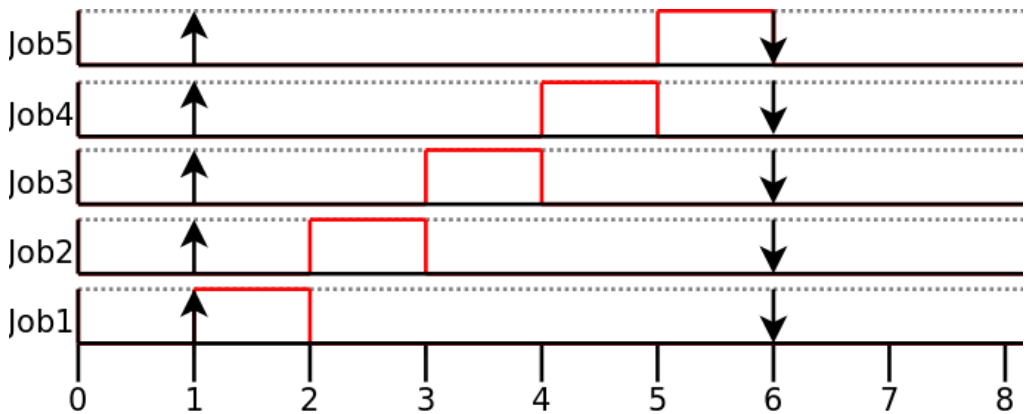
Scheduling Definition

- A schedule is called **feasible** if it fulfills all the task requirements
  - temporal, non-preemption, shared resources, precedences, ...
- A task set is called **schedulable** if there is at least one feasible schedule for that task set

The scheduling problem: easy to formulate, but hard to solve!

- Given:
  - A task set
  - Requirements of the tasks (or cost function)
- Find a time attribution of processor(s) to tasks so that:
  - Tasks are completely executed, and
  - Meet they requirements (or minimize the cost function)

E.g. $J = J_i(C_i = 1, a_i = 1, D_i = 5), i = \{1..5\}$



Scheduling problem

Exercise:

- Build a Gantt diagram for the execution of the following periodic tasks, admitting $D_i = T_i$ and no preemption.
  - $\tau = \{(1, 5)(6, 10)\}$
- Is the execution order important? Why?

Scheduling algorithms

- A **scheduling algorithm** is a method for solving the scheduling problem.
  - Note: don't confuse scheduling algorithm (the process/method) with schedule (the result)
- Classification of scheduling algorithms:
  - Preemptive vs non-preemptive
  - Static vs dynamic (priorities)
  - Off-line vs on-line
  - Optimal vs sub-optimal
  - With strict guarantees vs best effort

A short note on temporal complexity

- Measurement of the **growth** of the execution time of an algorithm as a function of the problem size (e.g. the number of elements of a vector, the number of tasks of a real-time system)
- Expressed via the $O()$ operator (big O notation)
- $O()$ arithmetic, n=problem dimension, k=constant
  - $O(k) = O(1)$
  - $O(kn) = O(n)$
  - $O(k_1 \cdot n^m + k_2 \cdot n^{m-1} + ... + k_{m+1}) = O(n^m)$

Compl. $= O(N)$

```
for (k=0;k<N;k++)
    a[k]=0;
```

Compl. $= O(N^2)$

```
for (k=0;k<N-1;k++)
    for (m=k;m<N;m++)
        if a[k]<a[m]
            swap(a[k],a[m]);
```

Compl. $= O(N^N)$

Computation of the permutations of a set $A = a_i, i = 1..N$

Basic algorithms

EDD - Earliest Due Date (Jackson, 1955)

- Single instance tasks fired synchronously:$J = Ji(Ci, (ai = 0,)Di), i = 1..n$
- Executing the tasks by non-decreasing deadlines minimizes the maximum lateness ($L_{max}(J) = max_i(f_i - d_i)$)
- Complexity: O(n.log(n))

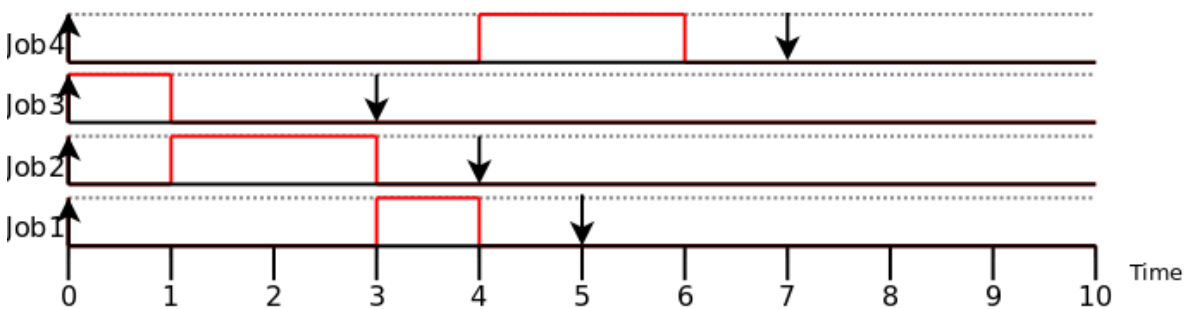
E.g. $J = \{J_1(1, 5), J_2(2, 4), J_3(1, 3), J_4(2, 7)\}$
Determine the maximum lateness!

Basic algorithms

EDD - Earliest Due Date (Jackson, 1955)

- Single instance tasks fired synchronously: $J = Ji(Ci, (ai = 0,)Di), i = 1..n$
- Executing the tasks by non-decreasing deadlines minimizes the maximum lateness ($L_{max}(J) = max_i(f_i - d_i)$)
- Complexity: O(n.log(n))

E.g. $J = \{J_1(1,5), J_2(2,4), J_3(1,3), J_4(2,7)\}$
Determine the maximum lateness!



Lmax,EDD(J) = -1

Basic algorithms

EDF - Earliest Deadline First (Liu and Layland, 1973; Horn, 1974)

- Single instance or periodic, asynchronous arrivals, preemptive: $J = J_i(C_i, a_i, D_i), i = 1..n$
- Always executing the task with shorter absolute deadline minimizes the maximum latency $L_{max}(J) = max_i(f_i - d_i)$
- Complexity: $O(n.log(n))$, **Optimal** among all scheduling algorithms of this class

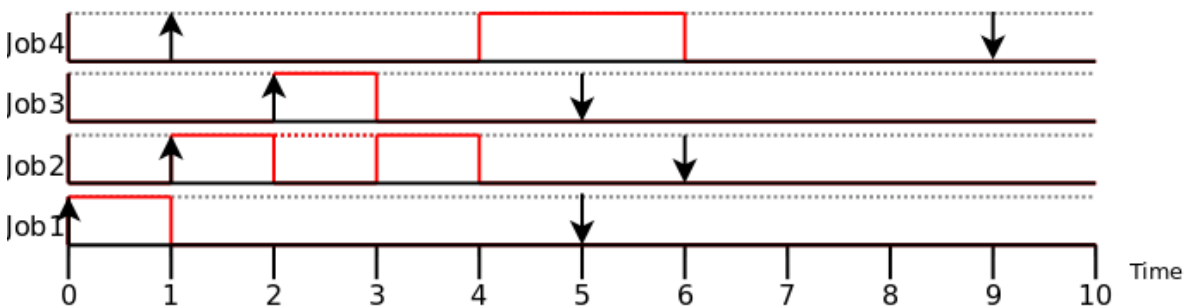
E.g. $J = \{J_1(1, 0, 5), J_2(2, 1, 5), J_3(1, 2, 3), J_4(2, 1, 8)\}$
Determine the maximum lateness!

Basic algorithms

EDF - Earliest Deadline First (Liu and Layland, 1973; Horn, 1974)

- Single instance or periodic, asynchronous arrivals, preemptive: $J = J_i(C_i, a_i, D_i), i = 1..n$
- Always executing the task with shorter absolute deadline minimizes the maximum latency $L_{max}(J) = max_i(f_i - d_i)$
- Complexity: $O(n.log(n))$, **Optimal** among all scheduling algorithms of this class

E.g. $J = \{J_1(1, 0, 5), J_2(2, 1, 5), J_3(1, 2, 3), J_4(2, 1, 8)\}$
Determine the maximum lateness!



Lmax,EDD(J) = -2

Basic algorithms

BB – Branch and Bound (Bratley, 1971)

- Single instance or periodic tasks, asynchronous arrivals, non-preemptive: $J = J_i(C_i, a_i, D_i), i = 1..n$
- Based on building an exhaustive search in the permutation tree space, finding all possible execution sequences:
- Complexity: O(n!)

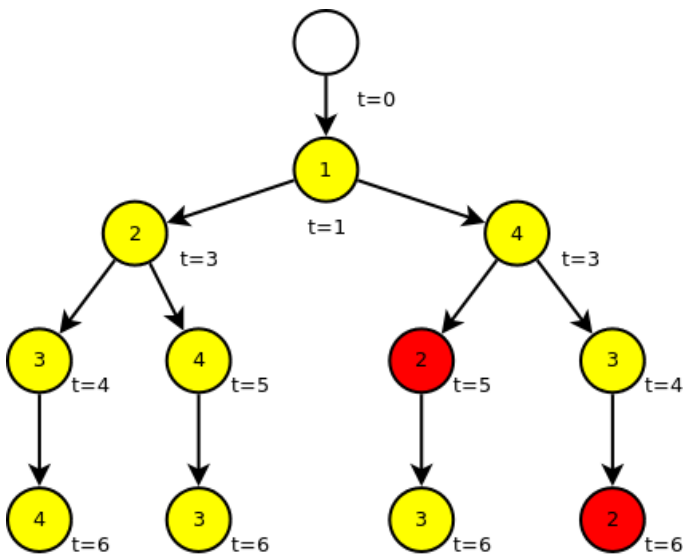
E.g. $J = \{J_1(1, 0, 5), J_2(2, 1, 3), J_3(1, 2, 4), J_4(2, 1, 7)\}$

Basic algorithms

BB – Branch and Bound (Bratley, 1971)

- Single instance or periodic tasks, asynchronous arrivals, non-preemptive: $J = J_i(C_i, a_i, D_i), i = 1..n$
- Based on building an exhaustive search in the permutation tree space, finding all possible execution sequences:
- Complexity: O(n!)

E.g. $J = \{J_1(1, 0, 5), J_2(2, 1, 3), J_3(1, 2, 4), J_4(2, 1, 7)\}$



Periodic task scheduling

Periodic tasks

- The release/activation instants are known **a priori**
- $\Gamma = \{\tau_i(C_i, \phi_i, T_i, D_i)\}, i = \{1...n\}$
- $a_{i,k} = \phi_i + (k-1) \cdot T_i, k = 1, 2, ...$

Thus, in this case the schedule can be built:

Online Tasks to execute are selected as they are released and finish, during normal system operation (addressed in next class)

Offline The task execution order is computed before the system enters in normal operation and stored in a table, which is used at runtime time to execute the tasks (static cyclic scheduling).

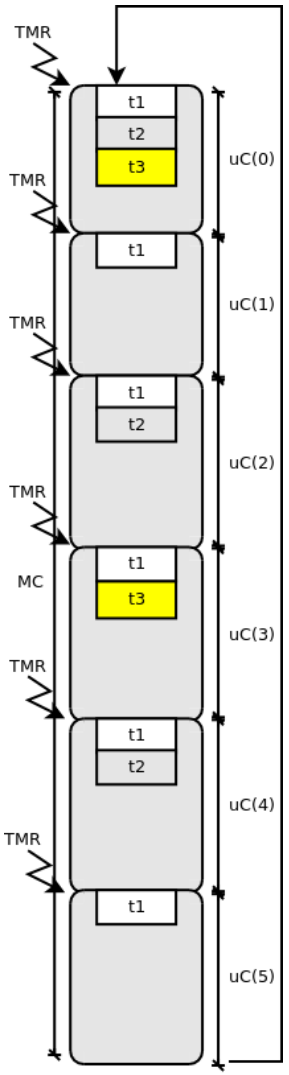
Static cyclic scheduling

- The table is organized in micro-cycles (μC) with a fixed duration. This way it is possible to release tasks periodically
- The micro-cycles are triggered by a Timer
- Scanning the whole table repeatedly generates a periodic pattern, called macro-cycle (MC)

$$\Gamma = \{\tau_i(C_i, \phi_i, T_i, D_i)\}, i = \{1...n\}$$
$$\mu C = GCD(T_i); MC = mCM(T_i)$$

Example:

- $\phi_i = 0$
- $T_1 = 5ms$; $T_2 = 10ms$; $T_3 = 15ms$



Static cyclic scheduling

Pros

- Very simple implementation (timer+table)
- Execution overhead very low (simple dispatcher)
- Permits complex optimizations (e.g. jitter reduction, check precedence constraints)

Cons

- Doesn't scale (changes on the tasks may incur in massive changes on the table. In particular the table size may be prohibitively high)
- Sensitive to overloads, which may cause the "domino effect", i.e., sequence of consecutive tasks failing its deadlines due to a bad-behaving task.

Static cyclic scheduling - Algorithm

How to build the table:

- Compute the micro and macro cycles (μC and MC)
- Express the periods and phases of the tasks as an integer number of micro-cycles
- Compute the cycles where tasks are activated
- Using a suitable scheduling algorithm, determine the execution order of the ready tasks
- Check if all tasks scheduled for a give micro-cycle fit inside the cycle. Otherwise some of them have to be postponed for the following cycle(s)
- It may be necessary to break a task in several parts, so that that each one of them fits inside the respective micro-cycle

Summary

- The concept of temporal complexity
- Definition of schedule and scheduling algorithm
- Some basic scheduling techniques (EDD, EDF, BB)
- The static cyclic scheduling technique

Homework

Consider the following message set (syntax (C,T), with D=T):

$$\Gamma = \{(1,5);(2,10);(3,10);(4,20)\}$$

- Compute the utilization of each task and the global utilization
- Compute the micro and macrocycle
- Build the scheduling table

Computation Models and Introduction to RTOS Architecture and Services

Real-Time Operative Systems Course

Paulo Pedreiras, DETI/UA/IT

September 21, 2022

Last lecture

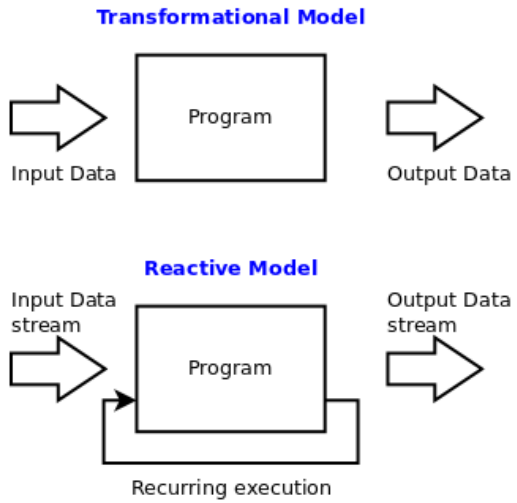
- Notion of real-time and real-time system
- Antagonism between real-time and best effort
- Aspects to consider: execution time, response-time and regularity of periodic events
- Requirements of RTS: functional, temporal and dependability
- Basic nomenclature
- Constraints: soft, firm and hard, and hard real time vs soft real time
- The importance of considering the worst-case scenario

Agenda for today

- Real Time model
- Additional task attributes
- Task states and execution
- Generic architecture of a RTOS
- RTOS examples

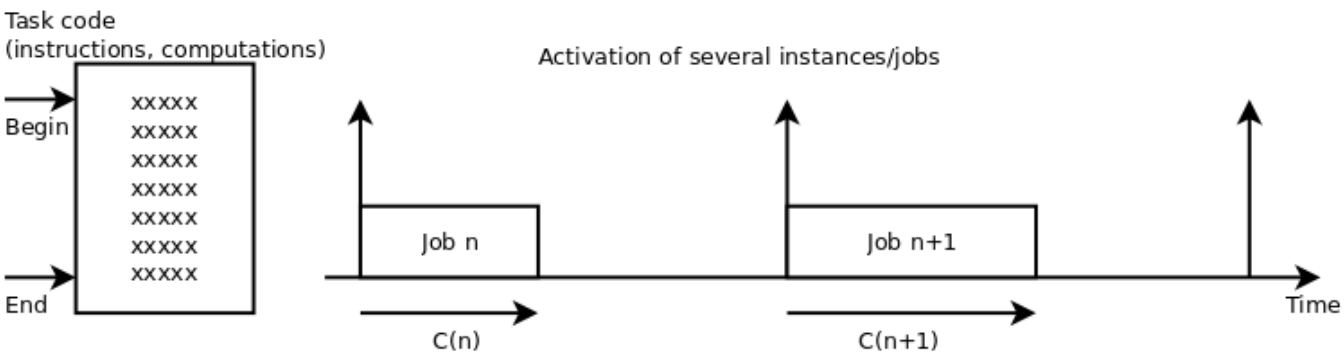
Real-time model

- **Transformational model**
  - According to which a program begins and ends, turning data into results or output data.
- **Reactive model**
  - According to which a program may execute indefinitely a sequence of instructions, for example operating on a data stream.
- **Real-time model**
  - Reactive model in which the program has to keep synchronized with the input data stream, which thus imposes time constraints to the program.



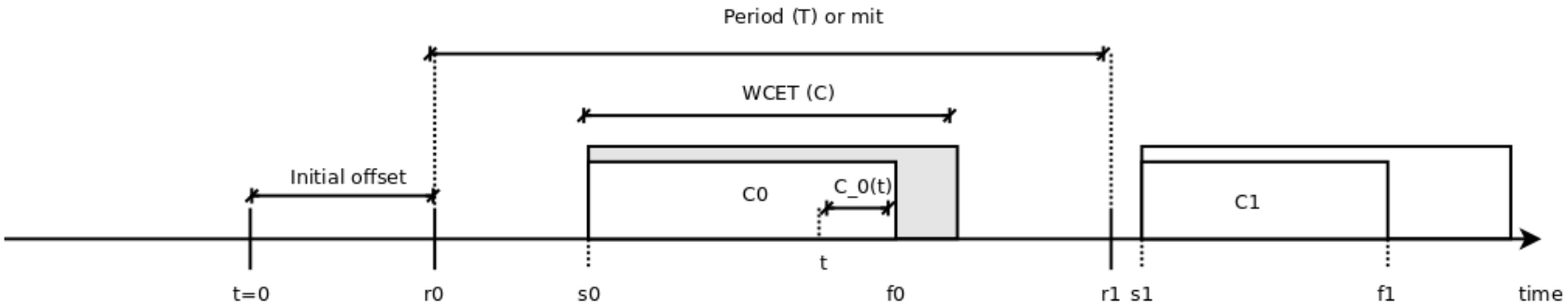
Processes, threads, tasks, activities and jobs

- **Definition of task (process, thread, activity)**
  - Sequence of activations (instances or jobs), each consisting of a set of instructions that, in the absence of other activities, is performed by the CPU without interruption.



Real-time model

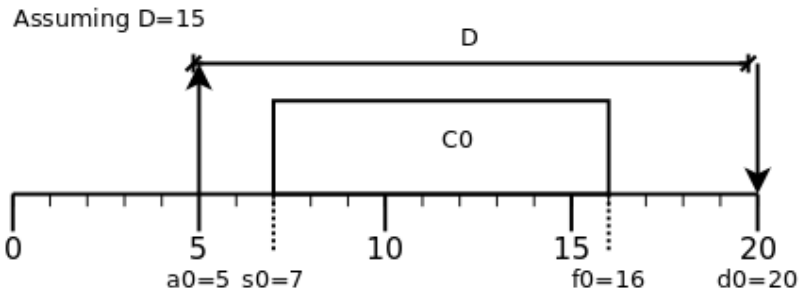
Temporal attributes



- Worst-Case Execution Time ($WCET$)
- Period (T) (if periodic, *mit* if sporadic)
- Relative phase or initial offset (ϕ)
- Release time (r_i)
- Finish/completion time (f_i)
- Residual execution time ($c_n(t)$): maximum remaining execution time required by job n at instant t

Deadline types

Deadline is the most common temporal requirements



D Relative deadline. Time measured from activation.

d_i Absolute deadline. Absolute time instant, regarding the i^{th} activation, in which the task must finish

$$d_i = r_i + D$$

- There are other types of task requirements: window, synchronization, distance, ...

Task temporal characterization

Example of task temporal characterization (there are other forms)

- **Periodic:**
  - $\Gamma = \tau_i(C_i, \phi_i, T_i, D_i)$
  - Examples: $\tau_1 = (1, 0, 4, 4), \tau_2 = (2, 1, 10, 8)$
- **Sporadic:**
  - Similar to periodic, but mit_i replaces T_i and ϕ_i usually is not used (tough it could be used to specify a minimum delay until the first activation)
  - $\Gamma = \tau_i(C_i, mit_i, D_i)$
  - Examples: $\tau_1 = (1, 4, 4), \tau_2 = (2, 10, 8)$

Temporal control

Triggering of activities

- **By time: Time-Triggered**
  - The execution of an activity (function) is triggered via a control signal based on the progression of time (e.g., through a periodic timer interrupt).
- **By events: Event-Triggered**
  - The execution of activities (functions) is triggered through an asynchronous control signal based on a change of system state (e.g., through an external interrupt).

Temporal control

Event-triggered systems

- Systems controlled by the occurrence of events on the environment
- Typical of sporadic conditions monitoring on the system state (e.g., alarm verification or asynchronous requests).
- Utilization rate of the the computing system resources (e.g. CPU) is variable, depending on the frequency of occurrence of events.
  - Ill-defined worst case situation. Implies either:
    - use of probability arguments, or
    - imposition of limitations on the maximum rate of events

Temporal control

Time-triggered systems

- Systems triggered by the progression of time
- Typical e.g. in control applications of cyber-physical systems
- There is a common time reference (allows establishing phase relations)
- CPU utilization is constant, even when there are no changes in the system state.
- Worst case situation is well-defined

Temporal control

Example

- For the following task sets compute the maximum response time that each task may experiment:
  - TT: $\Gamma = \tau_i(C_i = 1, \phi_i = i, T_i = 5, D_i = 5), i = 1...5$
  - ET: $\Gamma = \tau_i(C_i = 1, T_i = 5, D_i = 5), i = 1...5$
- Compute the average and maximum CPU utilization rate for both cases.
  - Admit that, on average, ET tasks are activated every 100 time units
  - Note: the CPU utilization is given by: $U_i = \sum_{i=1}^{N} \frac{C_i}{T_i}$

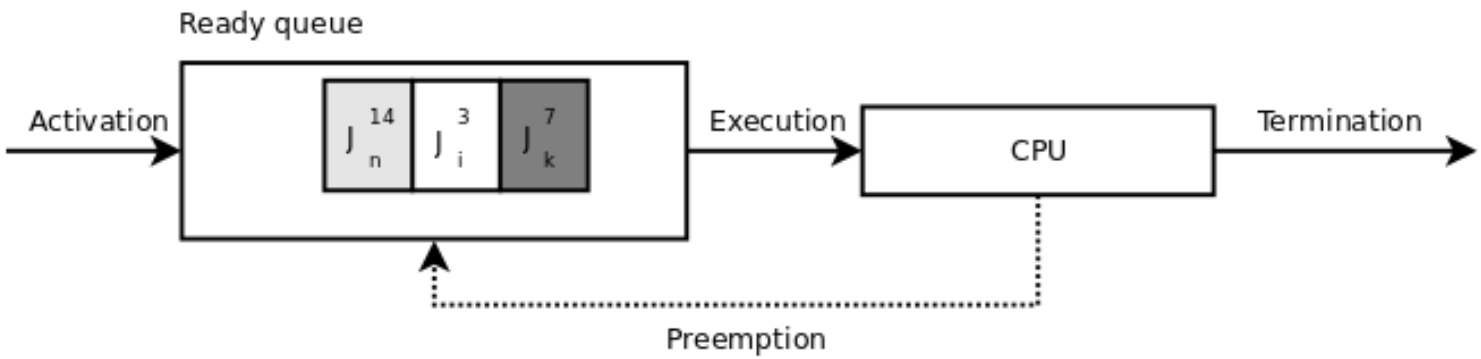
Task states and execution

- Task creation
  - Association between executable code (e.g. a "C" language function) to a private variable space (private stack) and a management structure - task control block (TCB)
- Task execution
  - Concurrent execution of the task's code, using the respective private variable space, under control of the RTOS kernel. The RTOS kernel is responsible for activating each one of the task's jobs, when:
    - A period has elapsed (periodic)
    - An associated external event has occurred (sporadic)

Task states

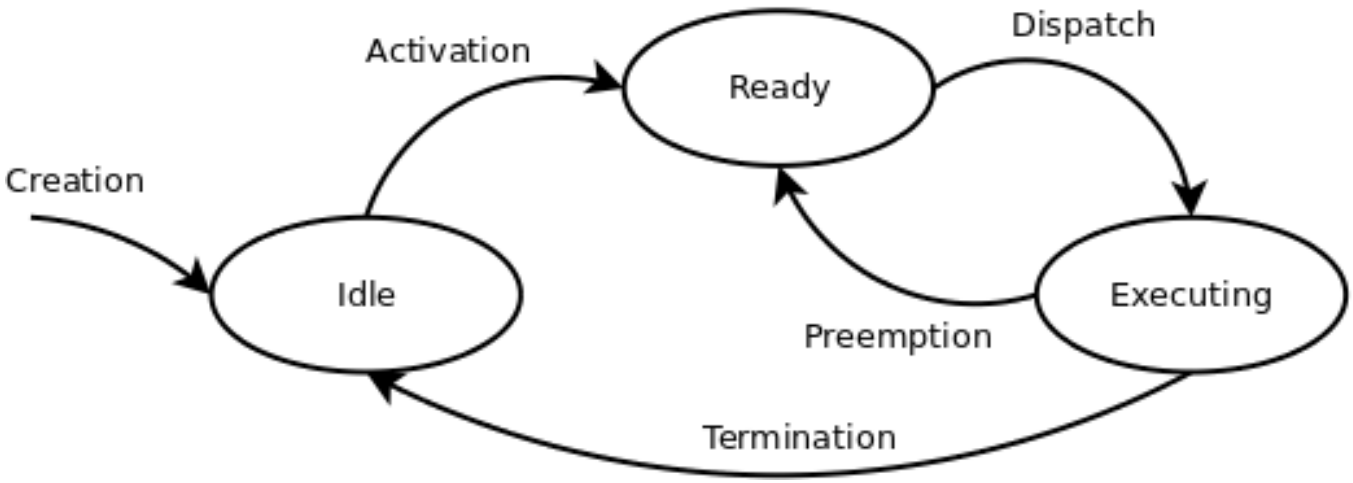
Execution of task instances (jobs)

- After being activated, task's jobs wait in a queue (the ready queue) for its time to execute (i.e., for the CPU)
- The ready queue is sorted by a given criterion (scheduling criterion). In real-time systems, most of the times this criterion is not the arrival order!



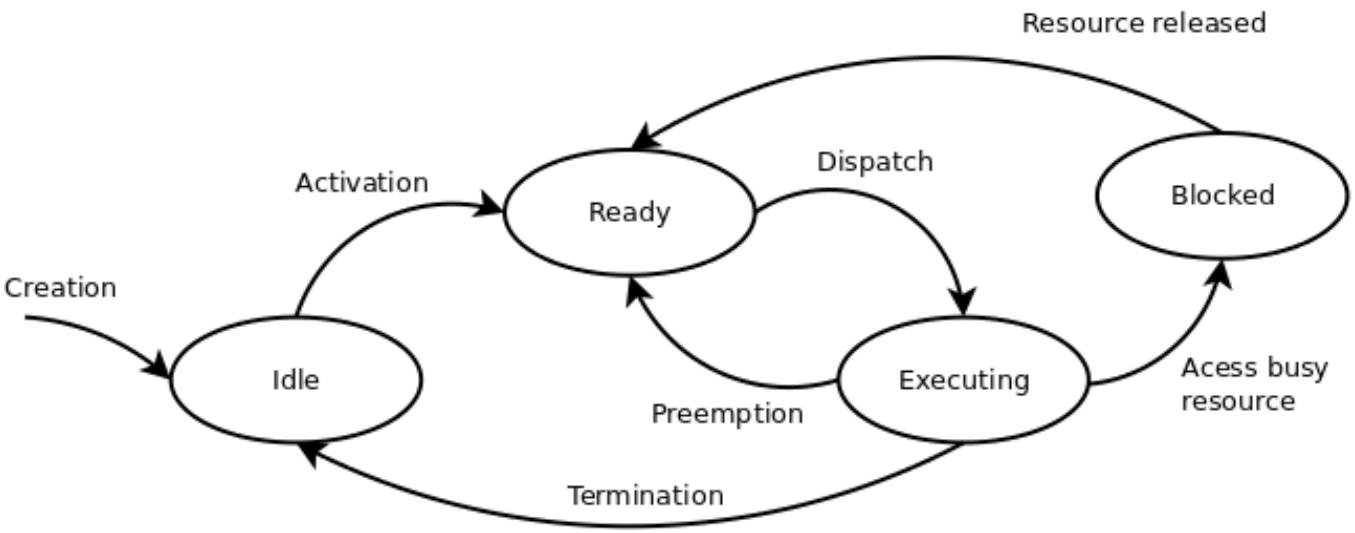
Task states

- Task instances may be waiting for execution (ready) or executing. After completion of each instance, tasks stay in the idle state, waiting for its next activation.
- Thus, the basic set of dynamic states is: **idle** , **ready** and **execution** .



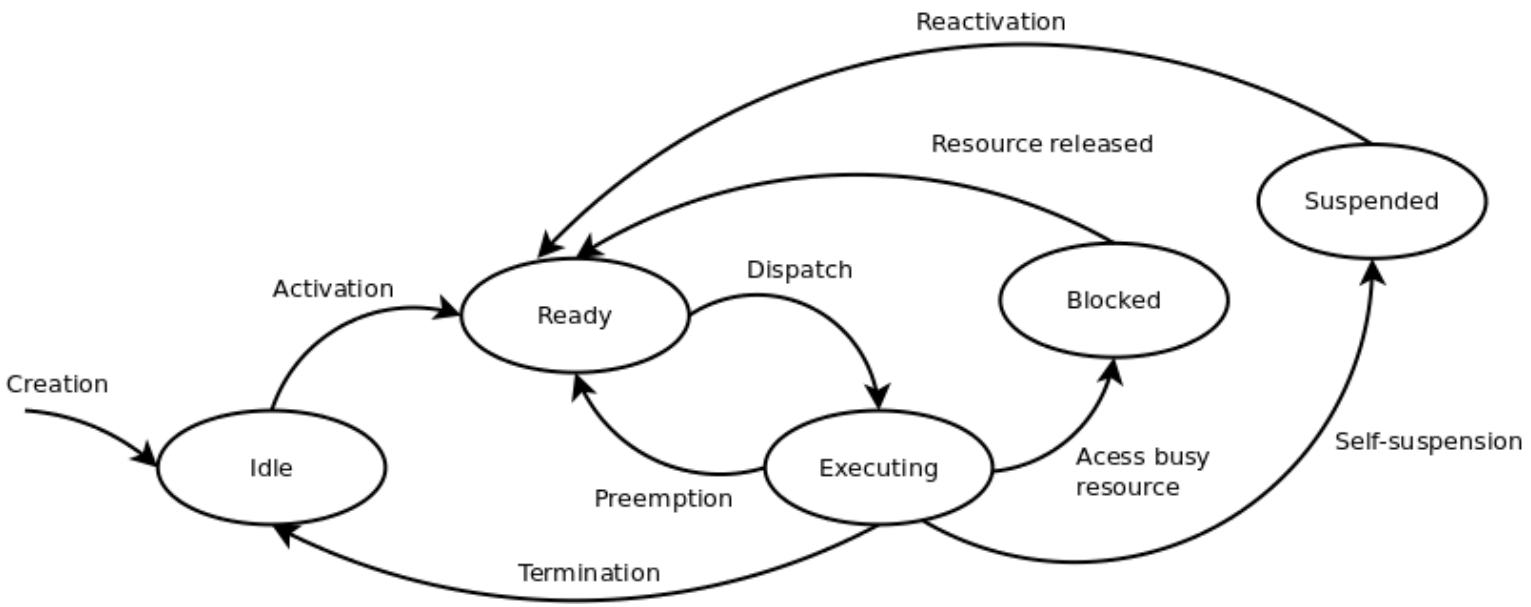
Task states

- Other states: **blocked**
  - Whenever an executing task tries to use a shared resource (e.g. a memory buffer) that is already being used in exclusive mode, the task cannot continue executing.
    In this case it is moved to the **blocked** state. It remains in this state until the moment in which the resource is released. When that happens the task goes to the ready state.



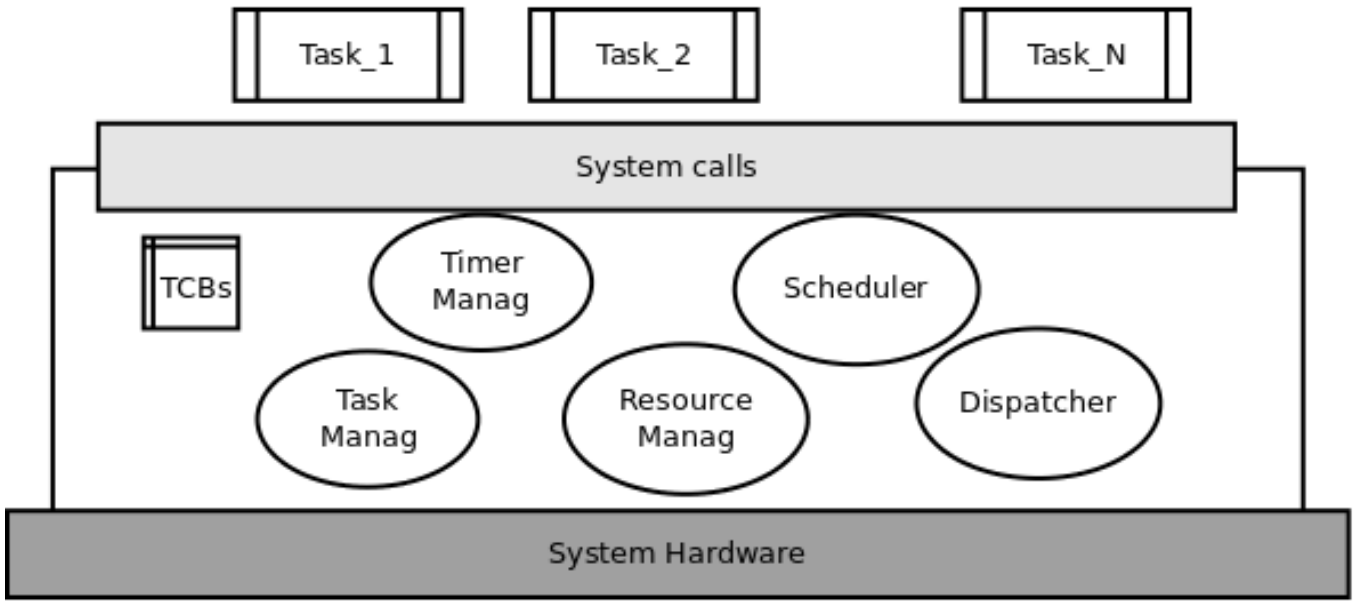
Task states

- Other states: self suspension (**sleep**)
  - In certain applications tasks need to suspend its execution for a given amount of time (e.g. waiting a certain amount of time for a packet acknowledgment), before completing its execution. In that case tasks move to the **suspended** state.



Internal Architecture of a Real-Time OS/Kernel

- Basic services
  - Task management (create, delete, initial activation, state)
  - Time management (activation, policing, measurement of time intervals)
  - Task scheduling (decide what jobs to execute in every instant)
  - Task dispatching (putting jobs in execution)
  - Resource management (mutexes, semaphores, etc.)



Management structures

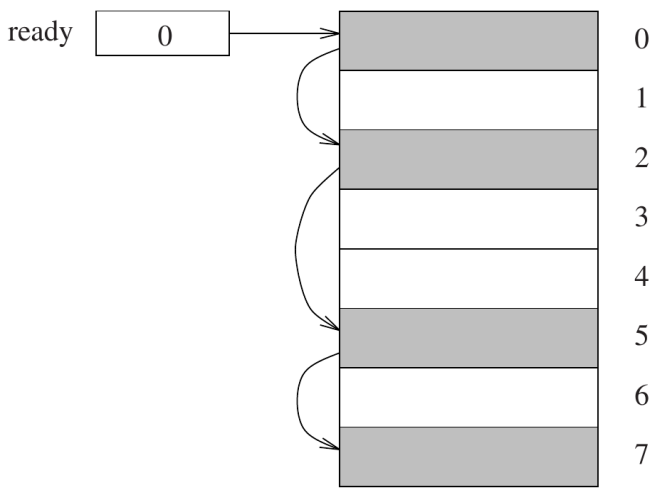
The **TCB** (task control block)

- This is a fundamental structure of a kernel. It stores all the relevant information about tasks, which is then used by the kernel to manage their execution.
- Common data (not exhaustive)
  - Task identifier
  - Pointer to the code to be executed
  - Pointer to the private stack (for context saving, local variables, ...)
  - Periodic activation attributes (task type (periodic/sporadic), period, initial phase, etc)
  - Criticality (hard, soft, non real-time)
  - Other attributes (deadline, priority)
  - Dynamic execution state and other variables for activation control, e.g. SW timers, absolute deadline, ...

Management structures

TCB structure

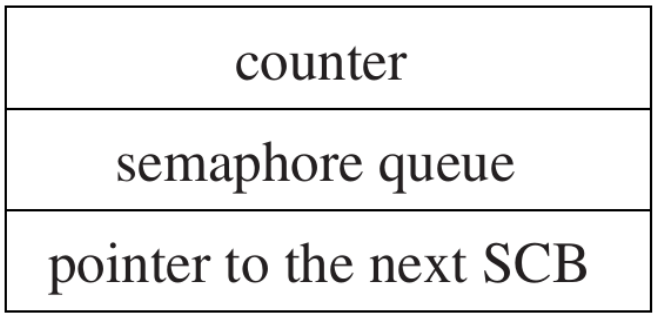
- TCBs are often defined in a static array, but are normally structured as linked lists to facilitate operations and searches over the task set.
- E.g., the ready queue (list of ready tasks sorted by a given criteria) is maintained as a linked list. These linked lists may be implemented e.g. through indexes. Multiple lists may (and usually do) coexist!



Management structures

SCB structure

- Similarly, the information concerning a semaphore is stored in a **Semaphore Control Block (SCB)** , which contains at least the following three fields:
  - A counter, which represents the value of the semaphore
  - A queue, for enqueueing the tasks blocked on the semaphore
  - A pointer to the next SCB, to form a list



Time management functions

- Time management is another critical activity on kernels. It is required to:
  - Activate periodic tasks
  - Check if temporal constraints are met (e.g. deadline violations)
  - Measure time intervals (e.g. self-suspension)
- It is based on a system timer. There are two common modes:
  - **Periodic tick** : generates periodic interrupts (system ticks). The respective ISR handles the time management. All temporal attributes (e.g. period, deadline, waiting times) must be integer multiples of the clock tick.
  - **Single-shot/tickless** : the timer is configured for generating interrupts only when there are periodic task activations or other similar events (e.g. the termination of a task self-suspension interval).

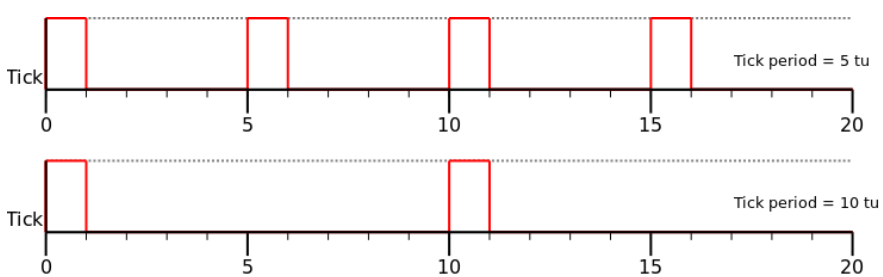
Time management functions

Tick-based systems

- The tick defines the system's temporal resolution.
  - Smaller ticks corresponds to better resolutions
  - E.g. if tick=10 ms and task periods are: T1=20ms, T2=1290ms, ~~T3=25ms~~
- The tick handler is code that is executed periodically, thus it consumes CPU time ($U_{tick} = \frac{C_{tick}}{T_{tick}}$).
- A bigger tick lowers the overhead; compromise!
  - $tick = GCD(T_i), i = 1..N$
  - E.g. T1=20 ms, T2=1290 ms, T3=25 ms, then GCD(20,1290,25)=5 ms

.

  - Works but U_{tick} doubles!



Time management functions

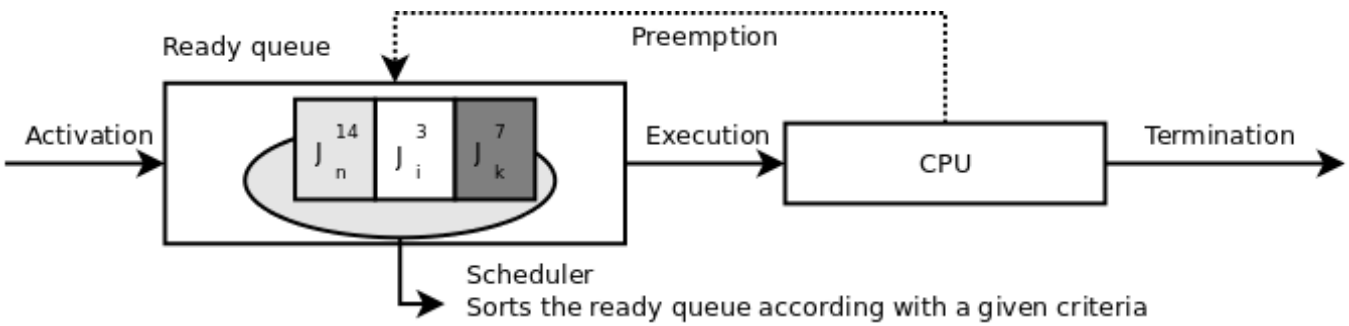
Measurement of time intervals

- In tick-based systems, the kernel keeps a variable that counts the number of ticks since the system boot.
  - In FreeRTOS the system call xTaskGetTickCount() returns the number of ticks since the scheduler was initiated (via vTaskStartScheduler()).
  - The constant portTICK_RATE_MS can be used to calculate the (real) time from the tick rate with the resolution of one tick period
  - Better resolutions require e.g. the use of HW timers.
- Be careful with overflows
- E.g. in Pentium CPUs, with a 1GHz clock, the TSC wraps around after 486 years !!!

Management functions - Scheduler

Scheduler

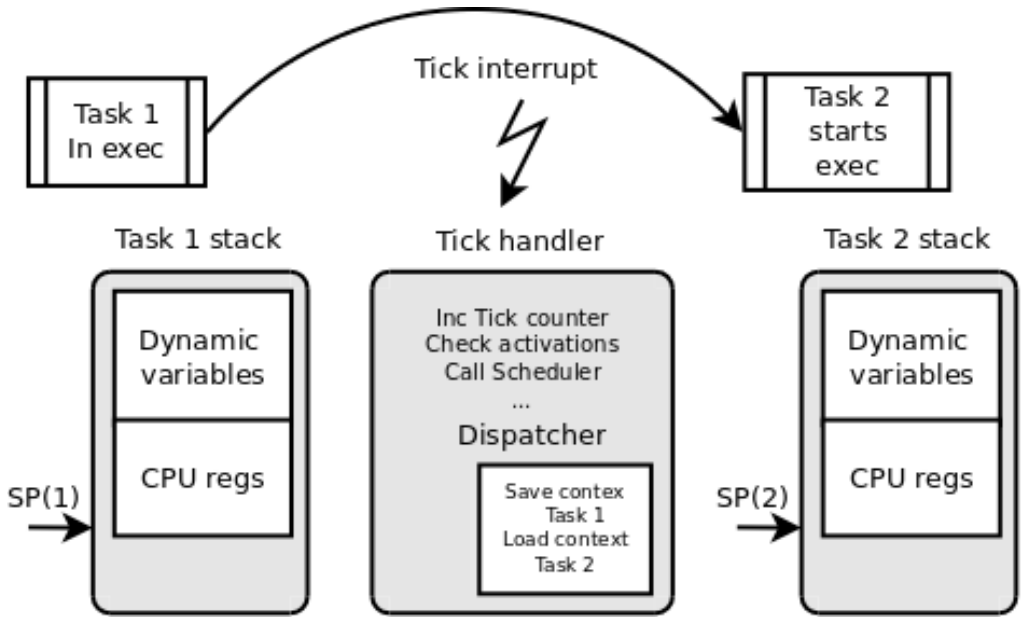
- The scheduler selects which task to execute among the (eventually) several ready tasks
- In real-time systems must be based on a deterministic criteria, which must allow computing an upper bound for the time that a given task may have to wait on the ready queue.



Management functions - Dispatcher

Dispatch

- Puts in execution the task selected by the scheduler
- For preemptive systems it may be needed to preempt (suspend the execution) of a running task. In these cases the dispatch mechanism must also manipulate the stack.



FreeRTOS

- Many CPU architectures (8 to 32 bit)
- Tick and tickless operation
- Task code is cyclic
- Scheduler is part of the kernel
- Allows preemption control
- IPC: queues, buffers
- Synchronization: semaphores, mutex, ...
- Monolithic application (kernel + application code in a single executable file)


```
void vTask1( void *pvParameters )
{
const char *pcTaskName = "Task 1 is running\r\n";
volatile unsigned long ul;

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( pcTaskName );

        /* Delay for a period. */
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
        {
            /* This loop is just a very crude delay implementation.  There is
            nothing to do in here.  Later examples will replace this crude
            loop with a proper delay/sleep function. */
        }
    }
}

int main( void )
{
    /* Create one of the two tasks.  Note that a real application should check
    the return value of the xTaskCreate() call to ensure the task was created
    successfully. */
    xTaskCreate(    vTask1,  /* Pointer to the function that implements the task. */
                    "Task 1",/* Text name for the task.  This is to facilitate
                             debugging only. */
                    1000,    /* Stack depth - most small microcontrollers will use
                             much less stack than this. */
                    NULL,    /* We are not using the task parameter. */
                    1,       /* This task will run at priority 1. */
                    NULL );  /* We are not going to use the task handle. */

    /* Create the other task in exactly the same way and at the same priority. */
    xTaskCreate( vTask2, "Task 2", 1000, NULL, 1, NULL );

    /* Start the scheduler so the tasks start executing. */
    vTaskStartScheduler();
```

Xenomai: Real-Time Framework for Linux

Xenomai, https://xenomai.org/

- Allow the use of Linux for Real-Time applications
- Dynamically loadable modules
- Tasks may execute on kernel or user space
- POSIX (partially compat.)
- Cyclic tasks
- Support to several IPC mechanisms, both between RT and NRT tasks (pipe, queue, buffer, ...)
- Several "skins"

```
// Main
int main(int argc, char *argv[]) {
    .... // Init code
    /* Create RT task */
    err=rt_task_create(\&task_a_desc, "Task a", ...);
    rt_task_start(\&task_a_desc, \&task_a, 0);
    ....
    /* wait for termination signal */
    wait_for_ctrl_c();
    return 0;
}

--------------------------------

    // A task
    void task_a(void *cookie) {
        /* Set task as periodic */
        err=rt_task_set_periodic(NULL, TM_NOW, ...);
        for(;;) { // Forever
            err=rt_task_wait_period(\&overruns);
            .... // Task work
        }
        return;
    }
```

SHaRK: Soft and Hard Real Time Kernel

Academic kernel, http://shark.sssup.it/

- Research kernel, main objective is flexibility in terms of scheduling and shared resource management policies
- POSIX (partially compat.)
- For x86 (i386 with MMU or above) architectures
- Cyclic tasks
- Several IPC methods
- Concept of Task Model(HRT, SRT, NRT, per, aper) and Scheduling Module
- Policing, admission control
- Monolithic application

```
 main( )
{
    create_system(... );
    /* For each task */
    create_task (...);
    config_system();
    release_system( );
    while(1)
    {
          /* background */
    }
}
-----------------------
    task_n( )
    {
        task_init();
        while(1) {
            /* Task code */
       .....
        }
    }
```

Summary

- Computational models (real-time model)
- Implementation of tasks using multitasking kernels
- Logic and temporal control
- Event-triggered and time-triggered tasks
  - States and transition diagram
- The generic architecture of a RT kernel
- The basic components of a RT kernel, its structure and functionalities
- RTOS examples

Basic Concepts on RTS

Real-Time Operative Systems Course

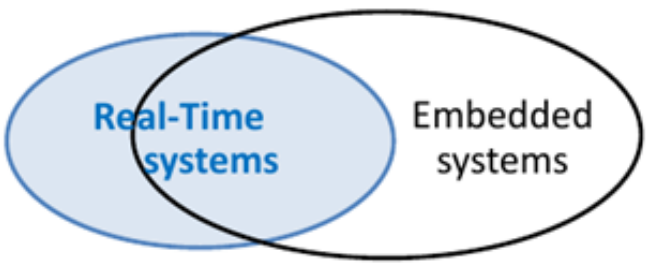
Paulo Pedreiras, DETI/UA/IT

September 21, 2022

Embedded vs Real-Time systems

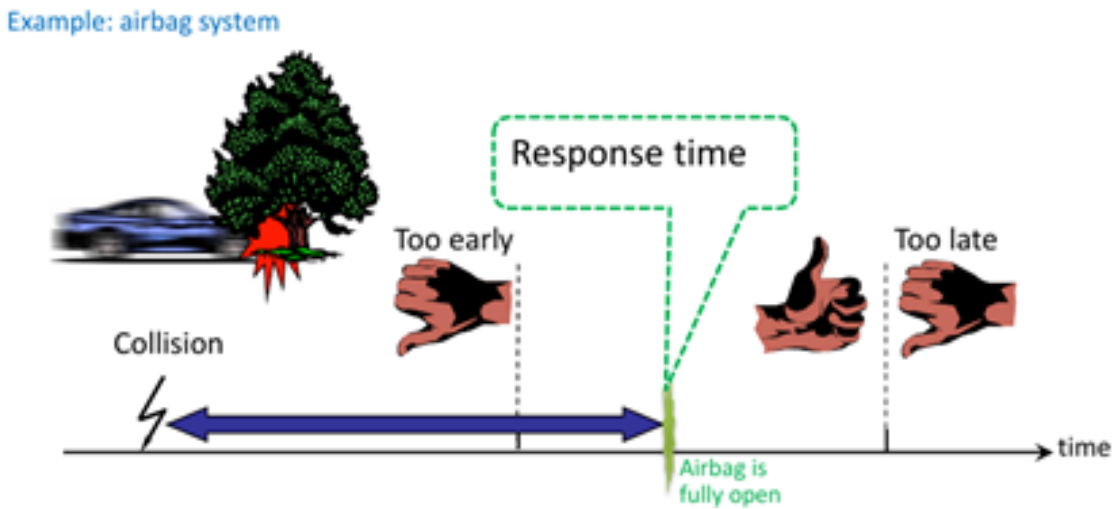
Real-Time and Embedded Systems are often confused. **But they are not the same ...**

- Real-time And Embedded:
  - Nuclear reactor control, Flight control, Mobile phone
- Real-time, but not embedded:
  - Stock trading system, Video streaming/processing
- Embedded, but not real-time:
  - Home temperature control, Washing machine, refrigerator, etc.



A few definitions related with "Real-Time"

- There is a wide variety of definitions related to Real-Time, systems dealing with Real-Time and the services that they provide.
- All of these definitions have in common the fact that express the **dependency of a computer system on the time, as it exists in a particular physical process** .



Definitions related with "Real-Time"

- Real-Time Service or Function
  - Which must be performed or provided within finite time intervals imposed by a physical process
- Real-Time System
  - One who contains at least one feature of real-time or at least providing a service of real-time
- Real-Time Science
  - Branch of computer science that studies the introduction of Real-Time in computational systems.

Real-Time Computation

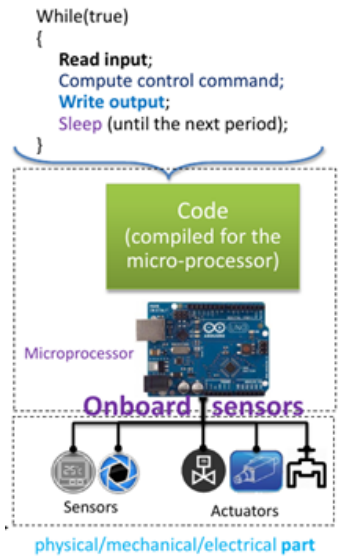
- The computation results must be
  - Logically correct
  - Delivered on time

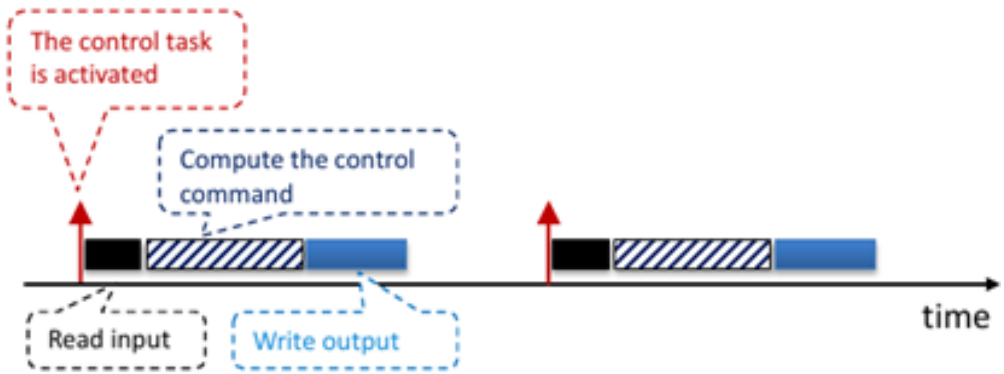
(Stankovic, 1988)

Timeliness  Logic Correction

Real-Time systems - implementation

- Simple case, with an infinite loop:

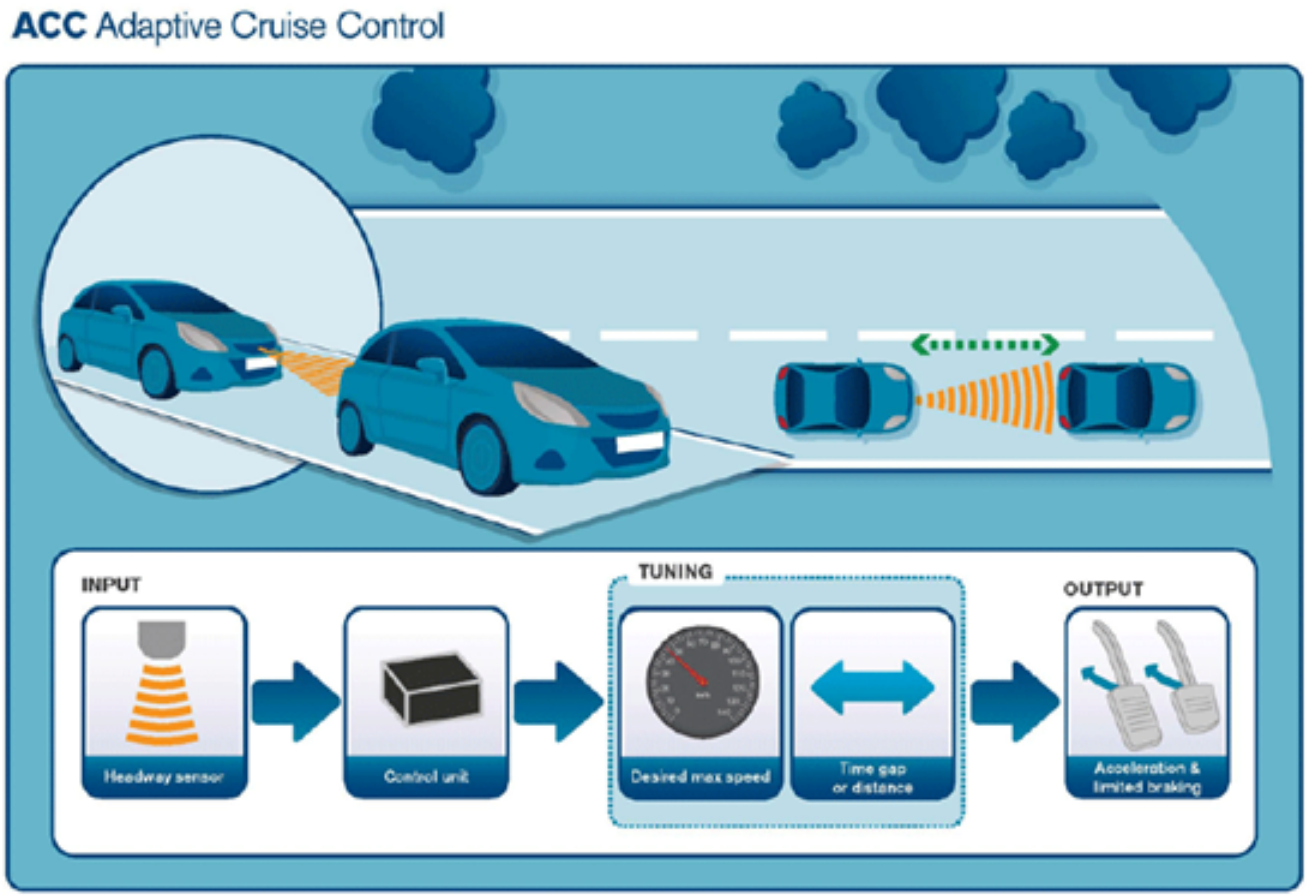




- Looks simple and predictable, right?

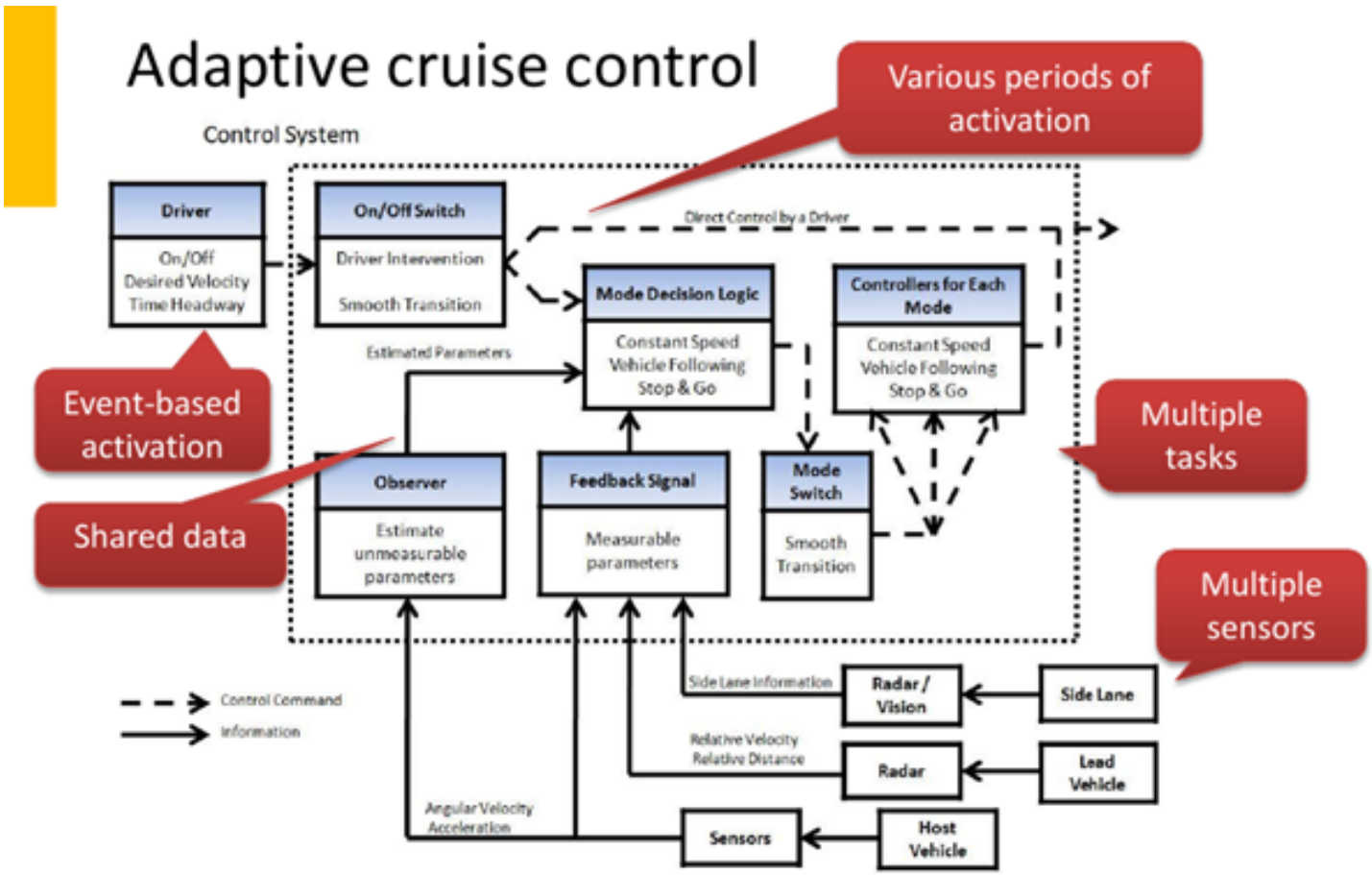
Real-Time systems - implementation

- But it can get **much more** complicated ...



Real-Time systems - implementation

- Does is still look simple and predictable?



https://codingcommunity.de/jaguar-cruise-control-diagram.html

Objective of the study of RTS

- Generally, when a computer system monitors the state of a given physical process and, if necessary, acts on it in time, then it is a real-time system.
- All living beings are real-time in relation to their natural habitats, which are its "real-time system".
- On the other hand, when we build (programmable) machines to interact with physical processes, we need to use Operative Systems, Programming Techniques and Analysis Methods that allow us to have confidence in its ability to carry out **correct and timely actions** .

Objective of the study of RTS

- The main objective of the study of Real-Time Systems is thus the development of techniques for
  - Design,
  - Analysis, and
  - Verification

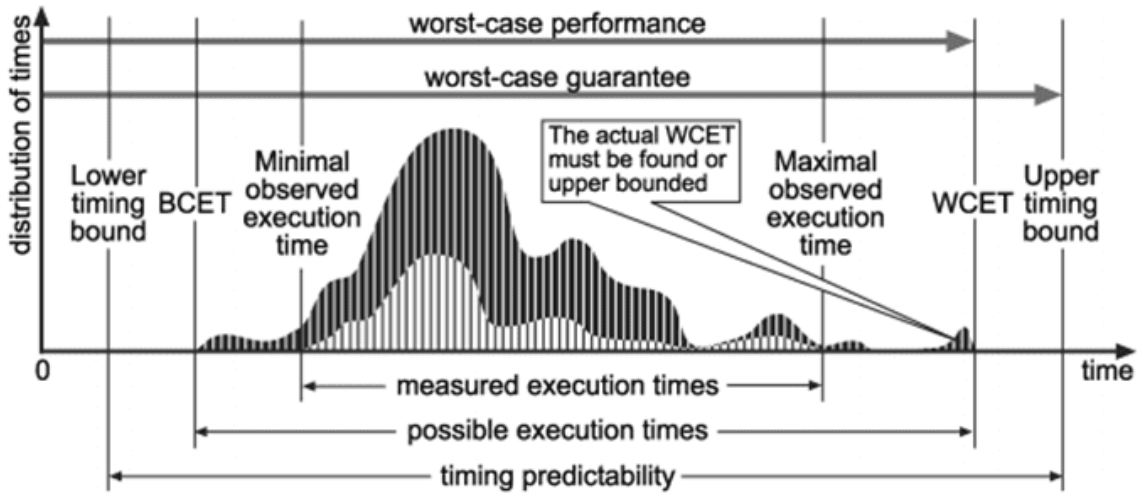
  that allow to assure that a given (computer, in our case) system has an appropriate timing behavior, satisfying the requirements imposed by the dynamics of the system with which it interacts

Objective of the study of RTS

- Regarding the computational activities of RTS, the main aspects to consider are:
  - Execution time
  - Response time
  - Regularity of periodic events

Objective of the study of RTS

Despite essential, these aspects are far from trivial:



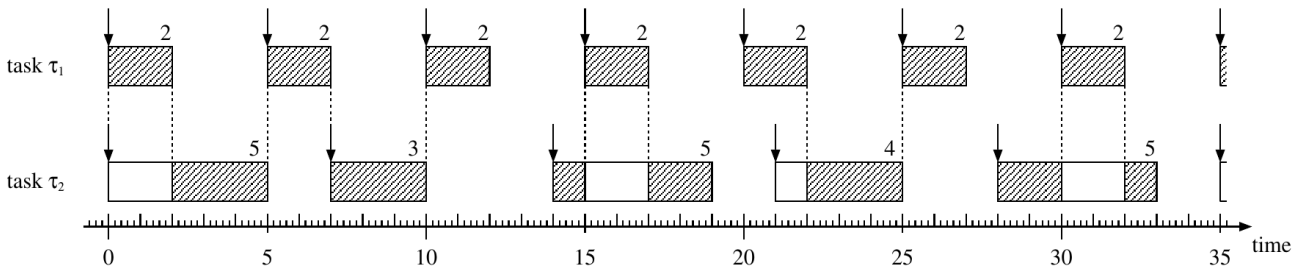
Analysing Extreme Value Theory Efficiency In Probabilistic Wcet Estimation With Block Maxima, ISSN:2448-0959

Reasons for this behaviour include:

- Execution time
  - Code structure (language, conditional execution, cycles)
  - DMA, cache, pipeline
  - Operative System or kernel (system calls)

Objective of the study of RTS

- Response time and regularity
  - Interrupts
  - Multi-tasking
  - Access to shared resources (buses, communication ports, ...)



Impact of interference on the Worst-case Response Time

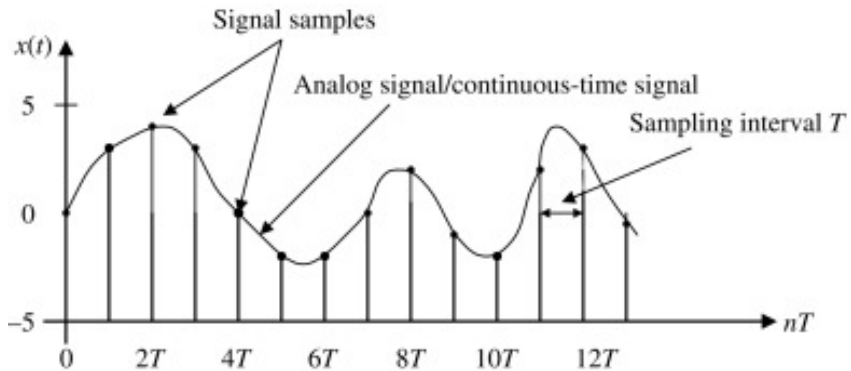
"Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption revisited", Reinder J. Bril et al

Requirements of Real-Time Systems

- The requirements commonly imposed to real-time systems are of three types:
  - Functional
  - **Temporal** (our main focus)
  - Dependability

Functional requirements

- Data gathering
  - Sampling of system variables (real-time entities), both analog and discrete
- Supervise and Control
  - Direct access to sensors and actuators
- Interaction with the operator
  - System status information, logs, support to correct system operation, warnings, ...

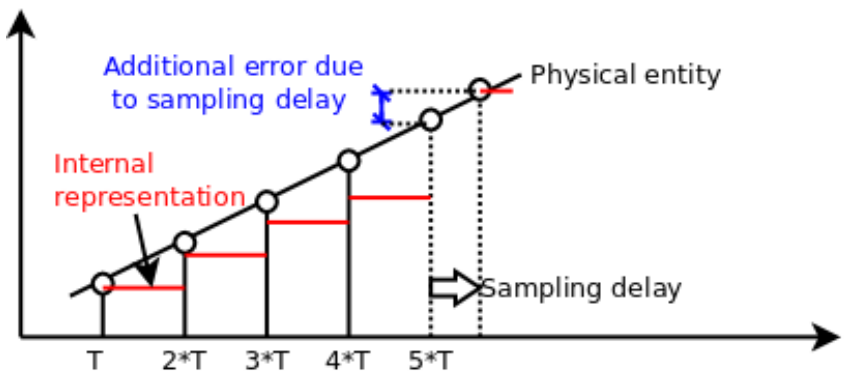


Periodic sampling illustration

Functional requirements

Data gathering

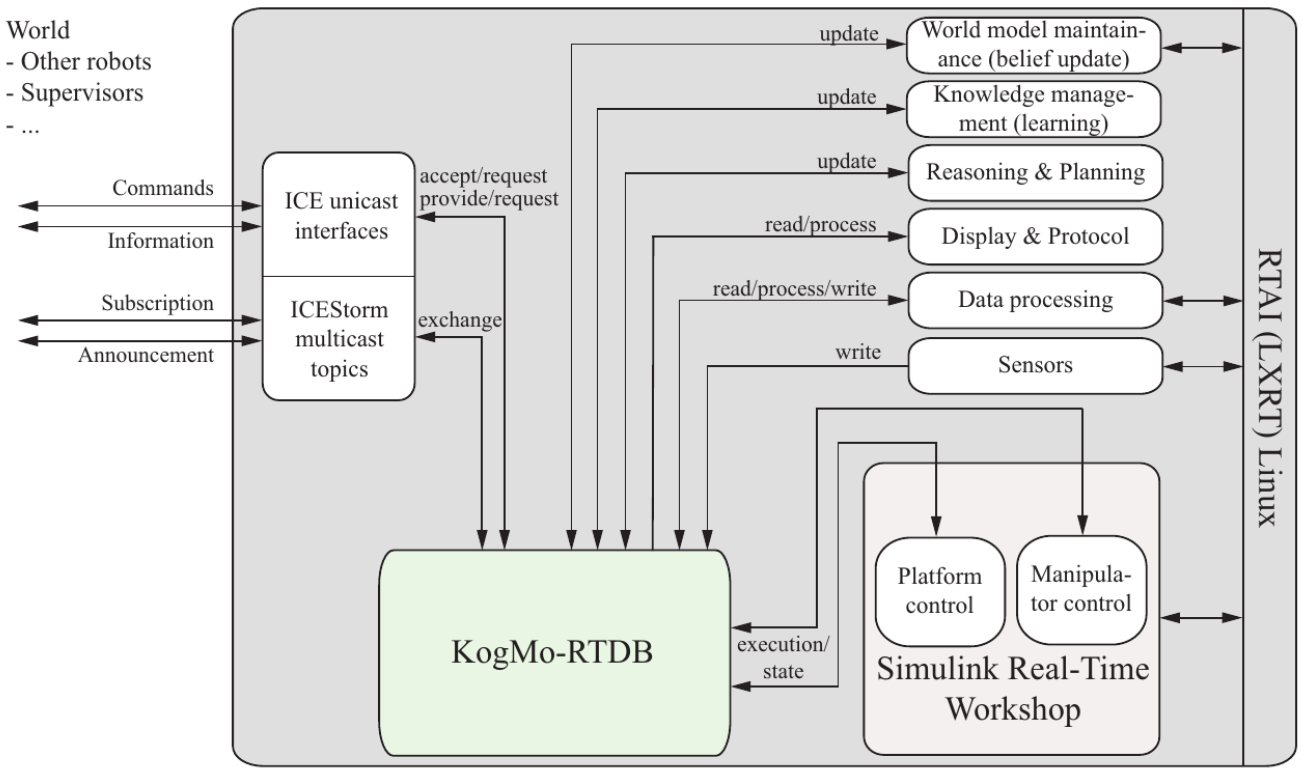
- The Real-Time computer operates on **local images** (internal variables) that represent the physical entities
- Each image of a real-time entity has a limited time validity, due to the temporal dynamics of the physical process
- The set of images of the real-time entities forms a **Real-Time Database**
- The real-time database must be updated to keep consistency between the physical world and its the internal representation



Impact of time delay in sampling error

Functional requirements

Illustration of a RTDB for Multi-Robot Systems



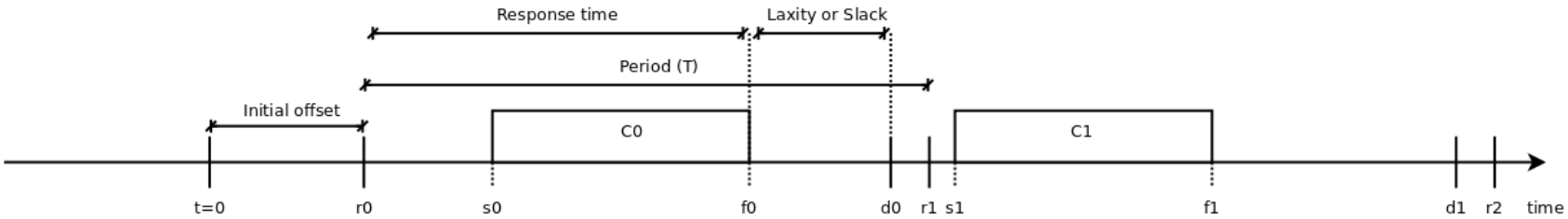
An Architecture for Real-Time Control in Multi-robot Systems, DOI:10.1007/978-3-642-10403-9_5

Temporal requirements

- Usually arise from the physical dynamics of the process to be managed or controlled
- Impose restrictions:
  - Delays the observation of the system state
  - Delays computing the new control values (acting)
  - Variations of previous delays (jitter)
- that must be followed in **all instances** (including the worst case) and not only on average

Temporal requirements

Terminology (1/2)



Initial offset (ϕ) : time before the first release/activation (job) of a task.

Period (T) : time between successive jobs of a task. Can be a Minimum Inter-Arrival time (MIT) for sporadic tasks.

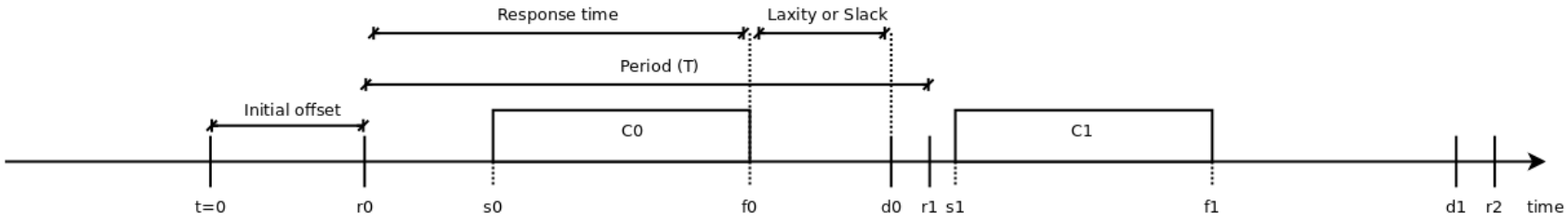
Release/activation/arrival time (r_i, a_i) : time instant of the i_{th} job of a task. $r_i = \phi + k \cdot T_i$ for periodic tasks.

Start time (s_i) : time at which the i_{th} job of a task start executing.

Finish/completion time (t_i) : time instant in which the i_{th} job of a task terminates.

Temporal requirements

Terminology (2/2)



Execution/computation time (C_i) :time necessary to the processor for executing the task instance without interruption.

Absolute deadline (d_i) : time instant by which the i_{th} execution of a task must complete

Response time (R_i) : time elapsed between the release of the i_{th} job of a task and its completion ($R_i = f_i - r_i$)

Slack/Laxity (L_i) : maximum time a task can be delayed on its activation to complete within its deadline ($L_i = d_i - r_i - C_i$)

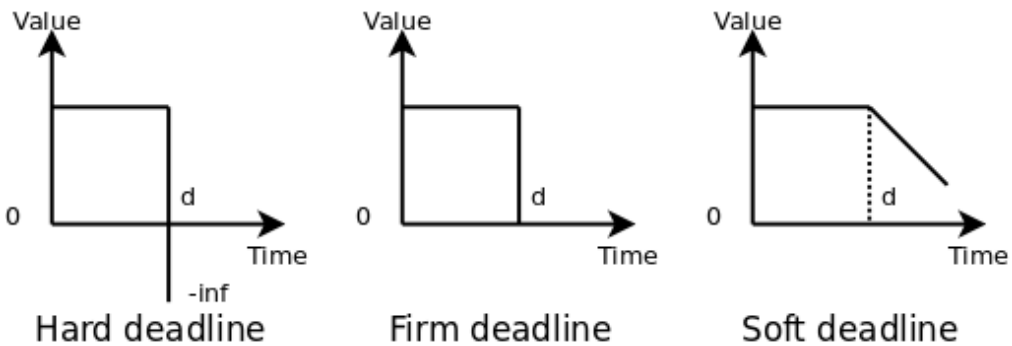
Temporal requirements

Classification of the temporal constraints, according with the **usefulness** of the result:

Soft: temporal constraint in which the result retains some utility to the application, even after a temporal limit D, although affected by a degradation of quality of service.

Firm: temporal constraint in which the result loses any usefulness to the application after a temporal limit D.

Hard: temporal restriction that, when not met, can lead to a catastrophic failure.



Temporal requirements

Classification of the Real-Time Systems, according with the **temporal** constraints:

Soft Real-Time: The system only has **firm or soft** real-time constraints (e.g., simulators, multimedia systems)

Hard Real-Time The system has **at least one hard real-time constraint** . These are the so-called safety-critical systems (e.g. airplane control, missile control, nuclear plants control, control of dangerous industrial processes)

Best Effort: The system **is not subject** to real-time constraints

Dependability requirements

- Real-time systems are typically used in critical applications, in which failures may endanger human lives or result in high economic impact/losses.
- This results in a requirement of **High Reliability**
  - Hard real-time systems have typically ultra-high reliability requirements ($< 10^{-9} failures/hour$)
  - Cannot be verified experimentally!
  - Validation requires solid analytic support (among other aspects)

Dependability requirements

- Important aspect to consider in safety-critical systems:
  - **Stable interfaces** between the critical and the remaining subsystems, in order to avoid error propagation between each other.
  - **Well defined worst case scenarios** . The system must have an adequate amount of resources to deal with worst case scenarios without resorting to probabilistic arguments, i.e. must provide service guarantees even in such scenarios.
  - Architecture composed of autonomous subsystems, whose properties can be checked independently of the others (**composability**).

Summary

- Notion of real-time and real-time system
- Antagonism between real-time and best effort
- Objectives of the study of RTS – how to guarantee the adequate temporal behavior
- Aspects to consider: execution time, response-time and regularity of periodic events
- Requirements of RTS: functional, temporal and dependability
- Constraints soft, firm and hard, and hard real-time vs soft real-time
- The importance of consider the worst-case scenario