Introductions to Real-time

Sebti Mouelhi, Benaoumeur Senouci
sebti.mouelhi@ece.fr
benaoumeur.senouci@ece.fr

ECE Paris – École d'ingénieurs

Real-time lectures, ING 4/5 Systèmes Embarqués

Content

What is real-time ?

Real-time

In computer science, **real-time computing** describes **computers** or **embedded systems (ESs)** whose operation is subject to **timing constraints**.

The system **correctness** depends not only on the **outputs of computation**, but also on the **delay after which** these outputs are produced.

Real-time is used in several industrial areas: **transportation** (avionics, aerospace, railway, automotive, etc), **energy** (nuclear, etc), **defense** (armament, navy and air force systems, etc), **entertainment** (video games, live streaming, augmented realty, etc), **robotics**, **cyber-security**, **nanotechnology**, **finance** ...

Machine clock

A **machine clock** is **fine discretization** of the ambient real time flow.

In real-time applications, **time units** are usually the **millisecond** (ms), the **microsecond** (μs) and sometimes the **nanosecond** (ns).

High-Integrity Safety-Critical (HISC) systems

Safety-critical system

A system is safety-critical when a **failure or malfunction** of its operation may result in **catastrophic outcomes**:

- **death** or **serious injury** to people,
- **loss or severe damage** to equipment and properties,
- **environmental harm** ...

The software of a safety-critical ES is built under **highly-integrity design** with respect to **safety certification** stipulated by **norms/standards**.

Examples:

- **Overheating prevention** in the core of a **nuclear power plant**;
- **Speed control** in **intelligent transportation systems** (ITS) (automotive, railway, etc);
- **Avionic/aerospace control** systems ...

Standards: EN 50128:2011 (railway), ISO 26262 (automotive), DO-178[A/B/C] (avionic) ...

Communications-Based Train Controller (CBTC)

Example of HISC systems

Communications-Based Train Controller (CTBC) [IEEE Standard 1474.1], specification for **automatic** and **smart** railway control systems:

- It covers **Automatic Train Protection (ATP)** functions (**safety-critical**):
  - continuously compute **precise locations** of trains and the **interlocking schema** based on the **current cartography**;
  - send back **movement authority limits** to trains to ensure **speed control**, **trains anti-collision** and **passengers security**;
- It is composed of **Carborne** and **Wayside** devices.

Communications-Based Train Controller (CBTC)

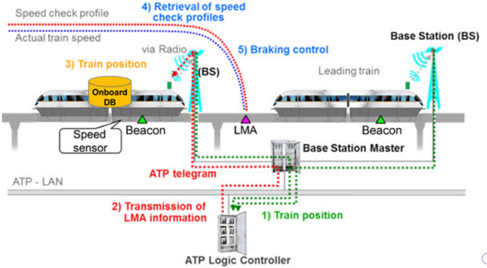
Example of HISC systems

Communications-Based Train Controller (CTBC) [IEEE Standard 1474.1], specification for **automatic** and **smart** railway control systems:

- It covers **Automatic Train Protection (ATP)** functions (**safety-critical**):
    - continuously compute **precise locations** of trains and the **interlocking schema** based on the **current cartography**;
    - send back **movement authority limits** to trains to ensure **speed control**, **trains anti-collision** and **passengers security**;
- It is composed of **Carborne** and **Wayside** devices.

Example: simplified ATP functions covered by the CBTC equipments:

- **trains mapping and positions**,
- computation of the **Limit of Movement Authority (LMA)**
- **speed control**,
- triggering of **emergency brakes** ...



Large-scale HISC systems development

The main steps of analyzing, designing and implementing real-time systems are:

- **Requirements analysis** and **functional and timing specifications**: what should the system do ?
- **Design** and **operational analysis**: how to build it ?
- **Hardware analysis**: with which hardware components ?
- **Software development**: how design is implemented?
- **Verification and integration**: is the software compliant with the functional specifications and satisfy requirements ? and by which mechanisms the timing constraints can be respected ?
- **Target testing and user validation**: is the combination of software and hardware harmonious and fulfills the user expectations ?

Design languages: UML/SysML, SCADE, Formal methods (B Method, etc) ...

Implementation languages: Ada, C, ...

Verification methods: Testing, Formal proof ...

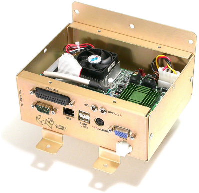
Content

Embedded system/software

Embedded system

An **embedded system** (ES) is a computer system:

- With a **dedicated control function** in a **larger system** interacting with the **physical world**;
- It may **interact** with **embedded sub-systems** and/or the **physical environment** (via **sensors/actuators**);



Embedded software

Embedded software (ESW) is written to **operate** an **embedded system**. It may be executed

- directly on a **microcontroller**;
- on top of an **embedded OS** running on a **System on a Chip (SoC)**.

System model

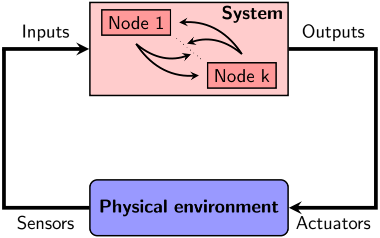
System

A **system** consists of **one or more computing nodes** interacting between each other via an **arbitrary network** (wired or wireless) in order to **control** and **monitor** a **physical processes**.

Node

A node is an ES executes some **behavioral control tasks** (or **processes**) under a **run-time environment**, like an operating system (OS).

The system's nodes interact with each other to achieve a **common objective** typically the **control** and **monitoring** of the environment in order to guarantee some **desired** and **safe behavior**.



Task definition

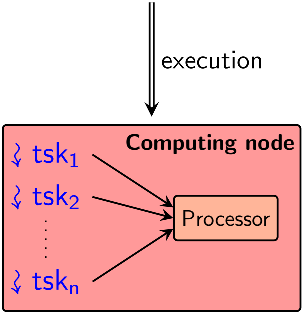
Task

A **task** forms a **logical unit of computation** in a processor.

An **application program** is a **static specification** of the computation logic of one or several tasks.

At run time, each task is executed by the processor as a single live running **thread of control**.

```
program Application
  task tsk_1; --body of tsk_1
  task tsk_2; --body of tsk_2
  ...
  task tsk_n; --body of tsk_n
begin
  initialization;
  --launch tasks concurrently
end
```



Task definition

Task

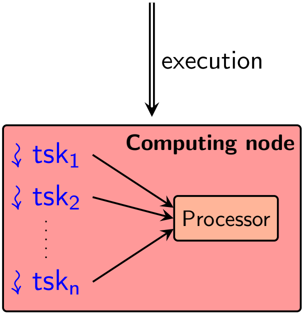
A **task** forms a **logical unit of computation** in a processor.

An **application program** is a **static specification** of the computation logic of one or several tasks.

At run time, each task is executed by the processor as a single live running **thread of control**.

On a **mono-core** processor, the execution of tasks is **interleaved**. On a **multi-core** processor, several tasks may be **dispatched and executed in parallel** by several **processing cores**.

```
program Application
  task tsk_1; --body of tsk_1
  task tsk_2; --body of tsk_2
  ...
  task tsk_n; --body of tsk_n
begin
  initialization;
  --launch tasks concurrently
end
```



Time representation

Time

Time is the **absolute current clock** value of a given computing node *i.e.*, the half-open real-time interval starting from **epoch** (an origin point like the system initialization). It is measured in **sut** the **smallest** representative **unit of time**.

Time instant

An **instant** $t \in \mathbb{N}$ is a **position** in Time stamp corresponding to (epoch + t sut).

Time span

A **time span** (or **delay**) $p \in \mathbb{N}$ corresponds to the **amount of time** p sut.

The unit "sut" and epoch are omitted.

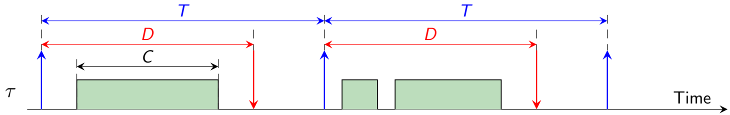
Real-time task

Real-time tasks

The timing model of a **real-time task** τ is a tuple (C, D, T) defined by **static chronological parameters** denoting **prefixed delays**:

- $C \in \mathbb{N}^*$ is its **worst case execution time WCET**, the delay during which the processor is fully allocated to τ;
- $D \in \mathbb{N}^*$ is its **relative deadline**, the maximum acceptable delay to run it;
- $T \in \mathbb{N}^*$ is its **period** (if defined), the delay separating two requests of τ:
  - If T is defined, τ is called **periodic**. Otherwize, it is called **aperiodic**.

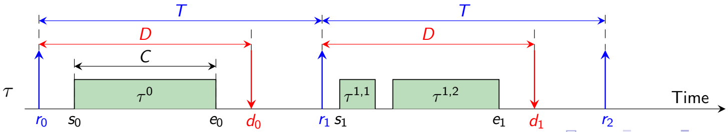
The task τ is **well-formed** if $C \leq D$ (if it is periodic, we have also $D \leq T$).



Periodic tasks: dynamic parameters

If τ is periodic, there are also some additional basic **dynamic chronometric parameters** denoting **time instants** useful to analyze its timing behavior:

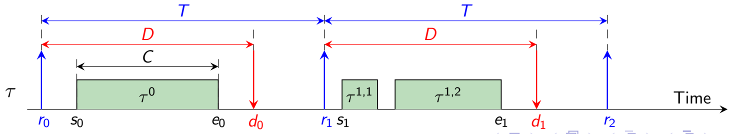
- The **release time** r is an instant of triggering a periodic request of τ:



Periodic tasks: dynamic parameters

If τ is periodic, there are also some additional basic **dynamic chronometric parameters** denoting **time instants** useful to analyze its timing behavior:

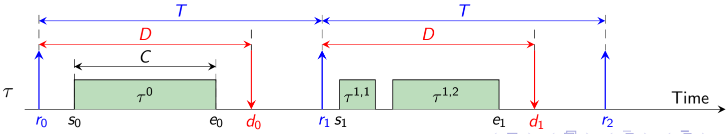
- The **release time** r is an instant of triggering a periodic request of τ:
  - r_0 is the initial release time of τ and $r_k = r_0 + kT$ is the $(k+1)^{th}$ one ($k \in \mathbb{N}$);



Periodic tasks: dynamic parameters

If τ is periodic, there are also some additional basic **dynamic chronometric parameters** denoting **time instants** useful to analyze its timing behavior:

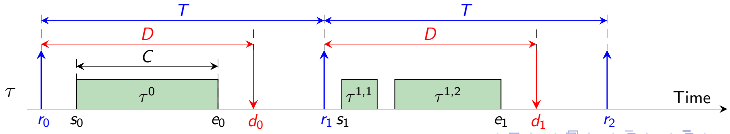
- The **release time** r is an instant of triggering a periodic request of τ:
  - r_0 is the initial release time of τ and $r_k = r_0 + kT$ is the $(k+1)^{th}$ one ($k \in \mathbb{N}$);
  - τ is hence a recurrence of **finite** (or **infinite**) number of **jobs** τ^k: each of them may be split into several **sub-jobs** $\tau^{k,1}, ..., \tau^{k,n}$ if its execution is suspended and resumed several times;



Periodic tasks: dynamic parameters

If τ is periodic, there are also some additional basic **dynamic chronometric parameters** denoting **time instants** useful to analyze its timing behavior:

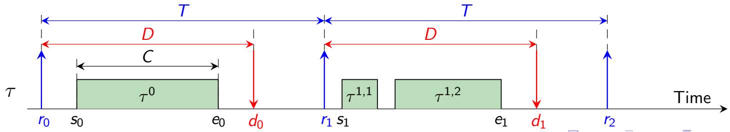
- The **release time** r is an instant of triggering a periodic request of τ:
  - r_0 is the initial release time of τ and $r_k = r_0 + kT$ is the $(k+1)^{th}$ one ($k \in \mathbb{N}$);
  - τ is hence a recurrence of **finite** (or **infinite**) number of **jobs** τ^k: each of them may be split into several **sub-jobs** $\tau^{k,1}, \ldots, \tau^{k,n}$ if its execution is suspended and resumed several times;
- The **absolute deadline** from a given release time r is $d = r + D$;



Periodic tasks: dynamic parameters

If τ is periodic, there are also some additional basic **dynamic chronometric parameters** denoting **time instants** useful to analyze its timing behavior:

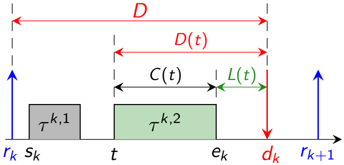
- The **release time** r is an instant of triggering a periodic request of τ:
  - r_0 is the initial release time of τ and $r_k = r_0 + kT$ is the $(k+1)^{th}$ one ($k \in \mathbb{N}$);
  - τ is hence a recurrence of **finite** (or **infinite**) number of **jobs** τ^k: each of them may be split into several **sub-jobs** $\tau^{k,1}, \ldots, \tau^{k,n}$ if its execution is suspended and resumed several times;
- The **absolute deadline** from a given release time r is $d = r + D$;
- Given a release time r, we have also s and e resp. the **start** and **finish instants** of executing τ since r and before r' the next release time:
  - We deduce obviously that $r \leq s \leq e \leq r'$.



Periodic tasks: other parameters

Some static other parameters are derived for τ:

- $U = \frac{C}{T} \leq 1$ is the **processor utilization factor** of τ;
- $H = \frac{C}{D} \leq 1$ is the **processor load factor** of τ;
- $L = D - C$ is is the **nominal laxity** of τ and denotes the maximum delay to start and run τ without suspension from the current release time.



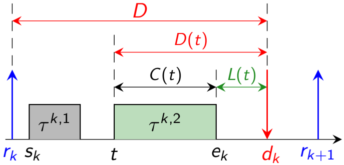
Periodic tasks: other parameters

Some static other parameters are derived for τ:

- $U = \frac{C}{T} \leq 1$ is the **processor utilization factor** of τ;
- $H = \frac{C}{D} \leq 1$ is the **processor load factor** of τ;
- $L = D - C$ is is the **nominal laxity** of τ and denotes the maximum delay to start and run τ without suspension from the current release time.

Given a release time r of τ and an instant $t \in [r, d]$, the following dynamic parameters are useful to define properly scheduling (to be studied later):

- $D(t) = d - t$ is the **residual relative deadline** at t: $0 \leq D(t) \leq D$;
- $C(t)$ is the **pending execution time** at t: $0 \leq C(t) \leq C$;



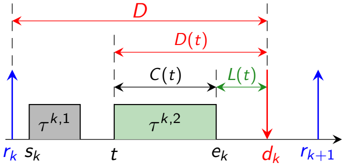
Periodic tasks: other parameters

Some static other parameters are derived for τ:

- $U = \frac{C}{T} \leq 1$ is the **processor utilization factor** of τ;
- $H = \frac{C}{D} \leq 1$ is the **processor load factor** of τ;
- $L = D - C$ is is the **nominal laxity** of τ and denotes the maximum delay to start and run τ without suspension from the current release time.

Given a release time r of τ and an instant $t \in [r, d]$, the following dynamic parameters are useful to define properly scheduling (to be studied later):

- $D(t) = d - t$ is the **residual relative deadline** at t: $0 \leq D(t) \leq D$;
- $C(t)$ is the **pending execution time** at t: $0 \leq C(t) \leq C$;
- $L(t) = D(t) - C(t)$ is the **residual nominal laxity** at t and denotes the maximum delay to run τ without suspension from $t + L(t)$;
- $H(t) = \frac{C(t)}{D(t)}$ is the **residual load factor** at t;



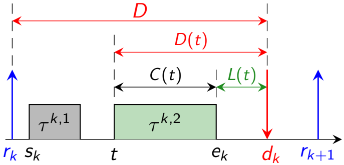
Periodic tasks: other parameters

Some static other parameters are derived for τ:

- $U = \frac{C}{T} \leq 1$ is the **processor utilization factor** of τ;
- $H = \frac{C}{D} \leq 1$ is the **processor load factor** of τ;
- $L = D - C$ is is the **nominal laxity** of τ and denotes the maximum delay to start and run τ without suspension from the current release time.

Given a release time r of τ and an instant $t \in [r, d]$, the following dynamic parameters are useful to define properly scheduling (to be studied later):

- $D(t) = d - t$ is the **residual relative deadline** at t: $0 \leq D(t) \leq D$;
- $C(t)$ is the **pending execution time** at t: $0 \leq C(t) \leq C$;
- $L(t) = D(t) - C(t)$ is the **residual nominal laxity** at t and denotes the maximum delay to run τ without suspension from $t + L(t)$;
- $H(t) = \frac{C(t)}{D(t)}$ is the **residual load factor** at t;



- $R = e - r$ is the **task response time**: $C \leq R$ and R should be $\leq$ then D.

Aperiodic vs. Sporadic tasks

An **aperiodic** task τ is **triggered randomly** by unpredictable **external events**:

- As events are random, their **arrival times are irregular**;
- There is no **time reference** to identify the release times of τ, that's why **no period is defined**;
- Analyzing the the **timing behavior** (deadline respectfulness) of τ under **random arrival time instants** is **impossible**.

Aperiodic vs. Sporadic tasks

An **aperiodic** task τ is **triggered randomly** by unpredictable **external events**:

- As events are random, their **arrival times are irregular**;
- There is no **time reference** to identify the release times of τ, that's why **no period is defined**;
- Analyzing the the **timing behavior** (deadline respectfulness) of τ under **random arrival time instants** is **impossible**.

However, aperiodic tasks often deal with **environment critical events** and hence their **deadlines are particularly important**.

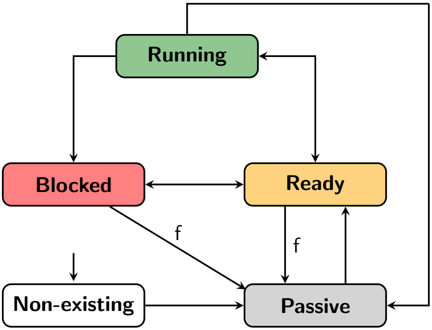
If a **minimum arrival time** M exists between any two arriving events of τ:

- **Timing behavior is analyzable** under arbitrarily deadlines;
- In this case, τ is called **sporadic**.

Task life cycle

A task may occupy a variety of states:

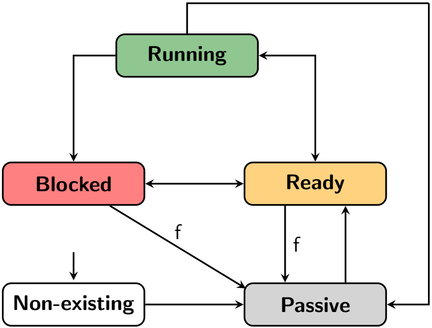
- It is first **non-existing**;
- Once created, it is **passive** and waits for requests;



Task life cycle

A task may occupy a variety of states:

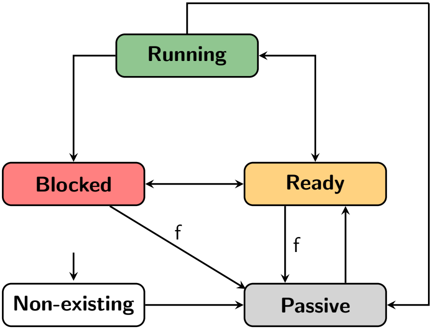
- It is first **non-existing**;
- Once created, it is **passive** and waits for requests;
- Once requested, it is **ready** and waits for election to run, in this case $L(t)$ and $D(t)$ decrease;



Task life cycle

A task may occupy a variety of states:

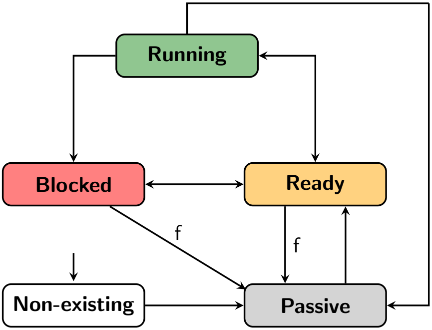
- It is first **non-existing**;
- Once created, it is **passive** and waits for requests;
- Once requested, it is **ready** and waits for election to run, in this case $L(t)$ and $D(t)$ decrease;
- Being elected, it becomes **running**, a processor is allocated to the task, in this case $C(t)$ and $D(t)$ decrease and $L(t)$ doesn't decrease;



Task life cycle

A task may occupy a variety of states:

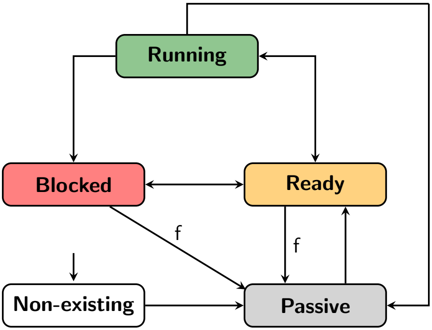
- It is first **non-existing**;
- Once created, it is **passive** and waits for requests;
- Once requested, it is **ready** and waits for election to run, in this case $L(t)$ and $D(t)$ decrease;
- Being elected, it becomes **running**, a processor is allocated to the task, in this case $C(t)$ and $D(t)$ decrease and $L(t)$ doesn't decrease;
- If it is suspended, it becomes **blocked**, the task wait for a resource, a message or a synchronization signal: $L(t)$ and $D(t)$ decrease.



Task life cycle

A task may occupy a variety of states:

- It is first **non-existing**;
- Once created, it is **passive** and waits for requests;
- Once requested, it is **ready** and waits for election to run, in this case $L(t)$ and $D(t)$ decrease;
- Being elected, it becomes **running**, a processor is allocated to the task, in this case $C(t)$ and $D(t)$ decrease and $L(t)$ doesn't decrease;



- If it is suspended, it becomes **blocked**, the task wait for a resource, a message or a synchronization signal: $L(t)$ and $D(t)$ decrease.

Transitions labeled by "f" represents the task interruptions caused by **operation/time faults**.

Other task characteristics

In addition of timing parameters, real-time tasks are described by other features:

- **Non-preemptive** tasks: once elected, they should not be stopped before the end of their execution, they are called also **immediate** tasks.
- **Preemptive** tasks: once elected, they may be stopped and moved back to the ready state in order to allocate the processor to other tasks;
- **Dependency**: tasks may be dependent according to several criteria:
  - tasks may **interact** according to a **static precedence relationship** fixed by **message transmission** or by **explicit synchronization**;
  - tasks may **share resource** other than processor which could be exclusive *i.e.*, they must be used in **mutual exclusion** by a sequence of instructions called **critical section**. Only one task is allowed to run in critical section;
- **External priority**: is a constant priority prefixed by the designer during the development according to its importance in the application.

Real-time systems

Real-time system

A **real-time system (ES)** executes **real-time tasks** and should **respect their timing requirements as well as possible**.

A real-time system may be:

- **Soft**: the deadlines of some of its real-time tasks **can be missed without compromising the system's integrity**.
- **Hard (or Firm)**: the **damage incurred by missing the deadlines** of some of its real-time tasks **is greater** than **any possible value of their regular timely executions**.

A real-time system is **safety-critical** if the damage caused by corrupted computations or missed hard deadlines has **catastrophic consequences**.

Examples of real-time systems

Real-time computer graphics

A graphical animation with **30 frames generated per second**:

- A frame has to be generated each 1/30 second;
- If this rate is not respected, the animation is slowed down (**QoS problem**);
- **Soft** RT system.

Examples of real-time systems

Real-time computer graphics

A graphical animation with **30 frames generated per second**:

- A frame has to be generated each 1/30 second;
- If this rate is not respected, the animation is slowed down (**QoS problem**);
- **Soft** RT system.

Overheating prevention in nuclear power plants

The temperature of a nuclear core is checked **every millisecond**:

- Sensor data are analyzed continuously by a plant controller.
- If data are delayed, the **controller's command are corrupted** and may cause **serious damage and harm**;
- **Hard** RT system.

Content

Scheduling and scheduler (recall)

Scheduling algorithm

A **scheduling algorithm** is a computing operation with a **predefined specification** by which tasks are **given access to the processor** in order to **achieve a target performance of execution** (such as respecting real-time).

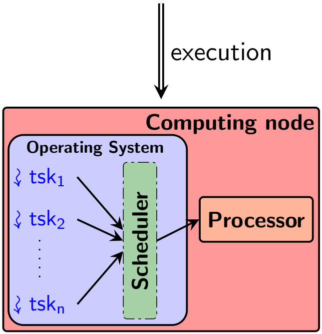
Schedule

A scheduling algorithm outputs a **schedule**: an execution **planning** of tasks by the processor.

Scheduler

A **scheduler** is a program implementing a **scheduling algorithm**. It is a part of the OS.

```
program Application
  task tsk_1; --body of tsk_1
  task tsk_2; --body of tsk_2
  ...
  task tsk_n; --body of tsk_n
begin
  initialization;
  --launch tasks concurrently
end
```



Real-time scheduling

Feasible schedule

A scheduling algorithm outputs a **schedule** for an input task set. This schedule is **feasible** if all the tasks meet their deadlines.

Schedulable task set

A task set is **schedulable** when there exists a scheduling algorithm able to **produce a feasible schedule** for it.

Real-time scheduling

Feasible schedule

A scheduling algorithm outputs a **schedule** for an input task set. This schedule is **feasible** if all the tasks meet their deadlines.

Schedulable task set

A task set is **schedulable** when there exists a scheduling algorithm able to **produce a feasible schedule** for it.

Real-time scheduling

A scheduling algorithm is **real-time** if it is able to **produce a feasible schedule** of an input task set.

Real-time scheduling

Feasible schedule

A scheduling algorithm outputs a **schedule** for an input task set. This schedule is **feasible** if all the tasks meet their deadlines.

Schedulable task set

A task set is **schedulable** when there exists a scheduling algorithm able to **produce a feasible schedule** for it.

Real-time scheduling

A scheduling algorithm is **real-time** if it is able to **produce a feasible schedule** of an input task set.

Optimal scheduling

A scheduling algorithm is **optimal** if it is able to **produce a feasible schedule** for **any schedulable** input task set **under some assumptions**.

General-purpose vs Real-time scheduling taxonomy

A scheduling algorithm may be generally of three kinds:

1. **Time-sharing**: it switches tasks on **regular clocked interrupts and events**;
2. **Event-driven**: it selects tasks according to the **freshness of their requests**, or the **duration of their execution times**, etc;

General-purpose vs Real-time scheduling taxonomy

A scheduling algorithm may be generally of three kinds:

1. **Time-sharing**: it switches tasks on **regular clocked interrupts and events**;
2. **Event-driven**: it selects tasks according to the **freshness of their requests**, or the **duration of their execution times**, etc;
3. **Priority-driven**: it orchestrates tasks based on their **priorities**, the higher the priority of a task, the more likely has to run undisturbed.

General-purpose vs Real-time scheduling taxonomy

A scheduling algorithm may be generally of three kinds:

1. **Time-sharing**: it switches tasks on **regular clocked interrupts and events**;
2. **Event-driven**: it selects tasks according to the **freshness of their requests**, or the **duration of their execution times**, etc;
3. **Priority-driven**: it orchestrates tasks based on their **priorities**, the higher the priority of a task, the more likely has to run undisturbed.

The **first two categories** of scheduling policies **cannot serve real-time**:

- None of them considers **tasks urgency** (based on their deadlines);
- Used in **general-purpose operating systems** (GPOSs).

The **third one** can however serve real-time because they consider urgency of tasks by **assigning to them priorities**:

- The higher the urgency criterion, the higher the priority;
- They are used in **real-time operating systems** (RTOSs).

Off-line vs. On-line scheduling taxonomy

Off-line scheduling builds a prior complete planning of tasks by knowing all the timing parameters **before execution**:

- The schedule is **known statically** and can be implemented efficiently;
- **Run-time low overhead** and **algorithmic complexity independence**;
- **Rigid** approach: all parameters (including release times) are fixed and cannot be adapted in case of need.

Off-line vs. On-line scheduling taxonomy

Off-line scheduling builds a prior complete planning of tasks by knowing all the timing parameters **before execution**:

- The schedule is **known statically** and can be implemented efficiently;
- **Run-time low overhead** and **algorithmic complexity independence**;
- **Rigid** approach: all parameters (including release times) are fixed and cannot be adapted in case of need.

On-line scheduling allows **choosing at run time the next task to be elected**. It has knowledge only of tasks currently being executed:

- When a new event occurs, a new task may be requested and elected without necessarily knowing in advance its arrival time and timing parameters;
- It manages the **unpredictable arrival of tasks** and allows progressive creation of schedules;
- **Flexible** and **dynamic** approach providing less precise statements about task than the off-line one, and it has **higher implementation overhead**;
- It is used to **cope with aperiodic tasks**.

Non-Preemptive vs. Preemptive scheduling

Non-preemptive scheduling doesn't in any case stop the current being executed:

- Critical resource sharing is easier since it does not require any concurrence;
- **Drawback**: It may result frequently **timing faults** that preemptive algorithms can easily avoid especially **under on-line configurations**.

Non-Preemptive vs. Preemptive scheduling

Non-preemptive scheduling doesn't in any case stop the current being executed:

- Critical resource sharing is easier since it does not require any concurrence;
- **Drawback**: It may result frequently **timing faults** that preemptive algorithms can easily avoid especially **under on-line configurations**.

Premptive scheduling: an elected task may be preempted to allocate the processor to a **more urgent task with higher priority**:

- The preempted task is moved to the ready queue, awaiting for a later election;
- It supports only preemptive tasks and **can ensure real-time determinism**;
- Modern schedulers are preemptive.

Real-time operating systems

Real-time operating system

A **real-time operating system (RTOS)** is an OS intended to serve real-time systems by **respecting the best possible the timing requirements** of its tasks using schedulers implementing **real-time scheduling algorithms**.

Two main kinds are distinguished:

- **Soft** RTOS: it can **usually respect deadlines**;
- **Hard** RTOS: it **always meets deadlines deterministically**.

Real-time operating systems

Real-time operating system

A **real-time operating system (RTOS)** is an OS intended to serve real-time systems by **respecting the best possible the timing requirements** of its tasks using schedulers implementing **real-time scheduling algorithms**.

Two main kinds are distinguished:

- **Soft** RTOS: it can **usually respect deadlines**;
- **Hard** RTOS: it **always meets deadlines deterministically**.

Some real-times OSs and APIs:

- **Linux-RT (the patch Preempt_RT)**,
- Wind River VxWorks,
- Xenomai (kernel-parallel API),
- FreeRTOS,
- RTEMS,
- QNX Neutrino, Micrium μC/OS, Windows CE, ...

RTOS low-level characteristics

A GPOS prioritizes **low-demanded system tasks** which penalizes drastically real-time determinism.

RTOS low-level characteristics

A GPOS prioritizes **low-demanded system tasks** which penalizes drastically real-time determinism.

A RTOS privileges to **user tasks with highest priorities**:

- It may **run user tasks in kernel mode** for a greater efficiency;

RTOS low-level characteristics

A GPOS prioritizes **low-demanded system tasks** which penalizes drastically real-time determinism.

A RTOS privileges to **user tasks with highest priorities**:

- It may **run user tasks in kernel mode** for a greater efficiency;
- It **masks** (prevent access to shared resources by) **OS calls and interrupts** as much as possible when a higher priority user task has to execute:
  - Once resources are released, pending interrupts and OS calls are executed;

RTOS low-level characteristics

A GPOS prioritizes **low-demanded system tasks** which penalizes drastically real-time determinism.

A RTOS privileges to **user tasks with highest priorities**:

- It may **run user tasks in kernel mode** for a greater efficiency;
- It **masks** (prevent access to shared resources by) **OS calls and interrupts** as much as possible when a higher priority user task has to execute:
  - Once resources are released, pending interrupts and OS calls are executed;
- **Minimal latency** of interrupts and thread switching;

RTOS low-level characteristics

A GPOS prioritizes **low-demanded system tasks** which penalizes drastically real-time determinism.

A RTOS privileges to **user tasks with highest priorities**:

- It may **run user tasks in kernel mode** for a greater efficiency;
- It **masks** (prevent access to shared resources by) **OS calls and interrupts** as much as possible when a higher priority user task has to execute:
  - Once resources are released, pending interrupts and OS calls are executed;
- **Minimal latency** of interrupts and thread switching;
- User tasks of **higher priority** has **exclusive access to shared resources** (including CPU and peripheral devices);
- They allow **priority inheritance** by lower priority tasks holding resources needed by a higher priority one to avoid the **priority inversion problem**;

RTOS low-level characteristics

A GPOS prioritizes **low-demanded system tasks** which penalizes drastically real-time determinism.

A RTOS privileges to **user tasks with highest priorities**:

- It may **run user tasks in kernel mode** for a greater efficiency;
- It **masks** (prevent access to shared resources by) **OS calls and interrupts** as much as possible when a higher priority user task has to execute:
  - Once resources are released, pending interrupts and OS calls are executed;
- **Minimal latency** of interrupts and thread switching;
- User tasks of **higher priority** has **exclusive access to shared resources** (including CPU and peripheral devices);
- They allow **priority inheritance** by lower priority tasks holding resources needed by a higher priority one to avoid the **priority inversion problem**;
- **Memory leak** (unused dynamically allocated memory) is not allowed:
  - All required memory allocation is **specified statically at compile time**;
- Magnetic and solid-state drive has much longer and unpredictable response time, **swapping** to disk files is not allowed.

Thank you for your attention