1

Introduction to Real-time Scheduling

The aim of this chapter is to introduce real-time scheduling. To do this, after a presentation of the context, we focus on material and software architectures commonly employed in the programming of real-time systems. Next, from a few examples of programs, classical task models are introduced. The techniques implemented to validate these models are developed throughout the chapters of this book.

1.1. Real-time systems

Real-time systems are very extensively used: from wide-consumption technological products (smartphones, games) to terrestrial transport systems (trains and cars) as well as aerial (aeroplanes) and spatial (satellites, shuttles and rockets) transport systems, through non-embedded critical systems such as power plant control, factory machinery control, or bank transaction systems. These are computer programs subject to temporal constraints. Non-compliance with the temporal constraints can lead to a discomfort of use for some programs referred to as *soft* real-time constraint programs (games, vehicle comfort

---

Chapter written by Emmanuel GROLLEAU.

functionalities such as air conditioning), or it can have catastrophic consequences for *strict* real-time constraint programs (such as the braking system of a vehicle or the control functionality of an aeroplane).

A real-time system can be either embedded or not: an embedded system embeds its own computing hardware and its own energy source. The energy sources can be electrical batteries, motors fed by fuel, ambient energy such as solar power, or even a combination of several sources. Embedded systems are characterized by low energy consumption computing capabilities in favor of autonomy, small size compared to non-embedded computing capabilities in order to reduce the footprint and the weight of the control system.

In the remainder of the chapter, we will call the global entity *system*, for instance, the control system of a vehicle. A system provides various *functionalities*, such as, for example in the case of a vehicle, braking, controlling the on-board radio, autonomous parking, and so on. The functionalities are generally ensured, on complex systems, by subsystems, which can be distributed over several CPUs and networks.

The functionalities and systems can be subject to temporal constraints. We can distinguish between local constraints and end-to-end constraints: an end-to-end constraint is typically derived from high-level requirements on the functionalities. A requirement describes what a functionality has to perform (functional requirement), or what properties it has to have (non-functional requirement). Generally, temporal constraints are considered non-functional since they characterize the response time the functionality has to have. A very widespread requirement in critical systems is segregation, which enforces two implementations of a same functionality to use different computing and communication resources.

EXAMPLE 1.1 (Braking and steering control system).– The following example, taken from a vehicle case study, illustrates the concepts of system, functionality, subsystem and requirements.

We consider a subset of the braking and steering correction functionalities on a passenger vehicle. Figure 1.1 represents different CPUs (called ECU for electronic control units in this context) as well as the main sensors and actuators, and the communication bus allowing the calculators to exchange information. The antilock braking system (ABS) functionality consists of measuring the speed of the various wheels and calculating a slip ratio. Above a certain value, the ABS control unit has to act on the hydraulic pressure regulating valve in order to reduce the exerted pressure, thus allowing the skidding wheels to regain traction, and therefore to reduce the braking distance. A non-functional requirement concerning this functionality could be that the maximum delay between the moment when the wheels skid and the moment when the pressure is reduced to be lower than 50 ms. In a simplified view, we could envision that the ABS functionality is performed by the subsystem that is running on the ABS control unit.

Let us now consider the steering correction functionality. This has to take into account the driver's intent (the angle of the steering wheel), as well as the speed of the vehicle, and can use gyro meters (measuring the angular speed) or an inertial unit (measuring the attitude using the measures of angular speed, heading and acceleration) in order to measure the rotational speed of the vehicle. Depending on the speed of the vehicle, the difference between the command attitude (angle of the steering wheel) and the rotational angle of the vehicle the extra sensory perception (ESP) control unit is able to determine whether the vehicle is in oversteer (the vehicle starts to go into a spin since the rear is drifting) or understeer (the front has a tendency to skid and the vehicle

continues forward instead of following the curve). The outputs of the ESP control unit, interpreted by the ABS control unit, are translated as a braking of the front outside wheel in order to compensate an oversteer, or the rear inside wheel for an understeer. We can thus see that the ESP functionality is distributed over several subsystems, running on several ECUs. The ESP can also be subject to end-to-end temporal constraints, which will be translated as local temporal constraints on the ECUs and communication buses involved in the functionality.

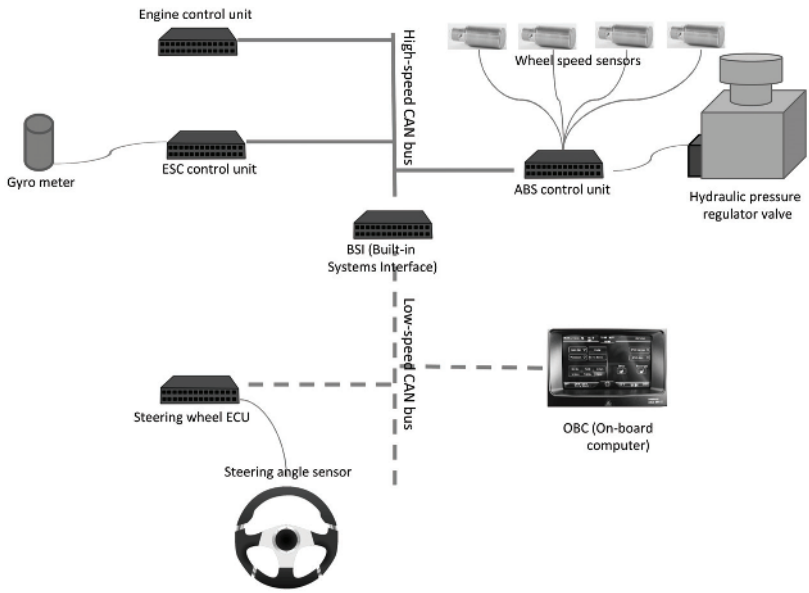

Figure 1.1. *Distributed system ensuring the braking and steering correction functionalities*

Some temporal constraints can be purely local: the pieces of information circulating in a network are typically cut up into frames (series of bytes). Embedded CPUs typically do not have a memory of more than a single received frame. Consequently, a requirement that we could expect to have would be for a CPU

to be able to read and memorize a frame before the reception of the next frame, under penalty of losing a frame.

On each subsystem running on a CPU, the requirements are mirrored by temporal constraints. A temporal constraint is a time frame in which a process must always be executed in its entirety.

1.2. Material architectures

From example 1.1, we have an overview of the main material elements composing a real-time system: CPUs, communication networks, sensors and actuators.

1.2.1. *CPUs*

In this section, we consider central processing units (CPUs) based on a Van Neumann or Harvard architecture, in other words CPUs that separate the memory and calculation units. Most of the CPUs in use since the invention of computing are indeed based on one of these architectures.

A CPU is a processor allowing the execution of programs. A program, while running, has its instructions copied into memory: it then becomes a process. A process can be composed of several algorithms that need to be run in parallel, these are tasks. A task is composed of a series of instructions to be executed sequentially. An instruction can be arithmetical and logical, a conditional or unconditional jump, movements between the memory and the registers, access to an input/output device, etc.

CPUs can be single- or multi-core: each computing core allows the execution of a task at a given time. A process is thus parallelizable since several tasks of the same process can be run simultaneously, on the other hand we generally

consider tasks not to be parallelizable. This means that a task cannot simultaneously be run on several cores.

The execution of an instruction consists of loading the instruction from memory, decoding it and running it. The time-stepping of the execution of the instructions is ensured by a clock, used as a time reference called the cycle, in the cores.

If all these operations were executed sequentially, then the processor cores would be underutilized. Indeed, the circuits specialized in the processing of instructions are available during the loading and decoding of the instruction. Moreover, the memory could be slower to respond than the execution time of an instruction. This is called a memory bottleneck, since the processor can be led to wait several cycles before the instruction arrives from memory. CPUs can therefore integrate local optimizations, or have particular architectures allowing, on average, the acceleration of certain processes. For instance, cache memory allows the storage of central memory data in rapid-access memories. These memories are closer to the processor and faster, but are of smaller size than the central memory and can therefore only memorize a part of the data. The working principle is that when the processor wants to read from an address in memory, the cache, if it has stored the content of that address, sends the content to the processor, which then does not have to wait for the central memory. When the requested address is not present in the cache, the cache stores it for an ulterior use. If it is full, a cache-managing strategy has to be used to decide which content will be replaced. This optimization brings, on the architectures on which it is employed, very significant increases in performance. This is due to the locality principle: a program often contains loops and manipulates the same data, consequently when the processor has to load an instruction or a piece of data, it is often to be found in cache memory. On newer architectures, there are several levels of

cache memory depending on the size and the speed. Moreover, on multi-core architectures, certain levels of cache memory can be shared by certain cores. In consequence, the parallel execution by several cores has an effect on the shared cache memory.

A CPU can be associated with specific circuits (*application specific integrated circuit* (ASIC)) allowing it to be relieved from time-costly functions, such as for example polling the arriving data on a communication bus, or computing the attitude (pitch angles, roll and heading) depending on the sensors of an inertial unit.

When an input/output device needs to communicate an event to the processor, as, for example, pressing a key on a keyboard or the arrival of a message on a communication bus, a hardware interrupt is triggered. After processing each instruction, a processor has to check whether a hardware interrupt has occurred. If this is the case, it has to process the interrupt. It stops the current processing, and executes the instructions of an interrupt handler routine.

From a real-time point of view, a CPU is thus a computing resource that runs tasks. The execution of each instruction takes time, expressed in cycles. Though the execution is sequential, numerous factors (material optimizations, interrupts) interfering with the execution of a task complicate the study of the duration of these processes. The field of study of the duration of tasks is called *timing analysis*.

1.2.2. *Communication networks*

A communication network is a medium allowing CPUs to communicate by sending each other data. The communication networks used in critical real-time systems have to be able to give guarantees regarding maximum delays in the transmission of messages. We therefore use deterministic

networks, generally with a decentralized arbitration (no CPU is indispensable for the network to work). This is the case of a *controller area network* (CAN), which is a synchronous deterministic network used mainly in vehicles and aeroplanes, or a switched Ethernet such as *avionics full duplex* (AFDX) employed in civil avionics that enables us to reach high throughputs.

From a general point of view, CPUs connected by communication networks transmit, on a physical level, frames (i.e. a series of bytes). From a real-time point of view, a transmission medium is seen as a frame-transmitting resource, the transmission time of a frame is obtained simply from the throughput and the length of the medium. The main difficulty, from a message transmission point of view, is to take into account the utilization of the shared media (see Chapter 6, Volume 2), or the wait in queues in the case of a switched network (see Chapter 7, Volume 2). We consider that the emission of a frame cannot be interrupted.

With the recent emergence of multi-core and *manycore* CPUs (we refer to several tens or hundreds of cores as *manycores*) a new kind of communication network has appeared: *networks on chip* (NoC). These networks connect computing cores. As it is not physically possible to directly connect all the cores, we could consider that the cores are the vertices of a two-dimensional grid, and that communication media (the NoC) connect a core to its four neighbors. In order to facilitate the integration on a single chip, this grid can have more than two dimensions, and constitute a cube or a hypercube. In this case, the transmitted packets are relatively small in size in order for them to be easily stored in the cores, which will then work as routers transferring the frames from one source core to a destination core.

1.2.3. *Sensors and actuators*

A sensor is a device capable of reading a physical quantity (temperature, pressure, speed, etc.). There is a very large variety of sensors, their common feature is that in order to interface with a computer system, they have to offer at least a digital or analog interface, or have a communication bus interface. A digital or analog interface uses an electric quantity to represent the measured physical quantity. A communication bus interface allows the sensor to transmit frames containing measures in a binary format.

An actuator is a device which allows us to control a physical element (flight control surfaces, solenoid valves, engines, etc.). Just like a sensor, it has to have a digital or analog interface or a bus.

It may be noted that digital and analog inputs as well as buses that can be found on CPUs can be of two types: polling and interrupt-based. Polling inputs allow a program to read the binary representation of an electrical signal in input. Interrupt-based inputs trigger, on certain events, a hardware interrupt on the CPU, which will then have to execute an interrupt handler routine.

1.3. Operating systems

To facilitate the exploitation of material resources (CPUs, networks, memories, etc.) by an application, the operating system provides services and primitives that ease the programming. The aim of this section is to present the general aspects of operating systems and to characterize what makes an operating system real-time. It also aims to present the primitives that can be found in real-time applications.

1.3.1. *Generalities*

An operating system can be broken down into three layers:

– The kernel manages the memory, the processor and the hardware interrupts. The time sharing of the cores of a CPU between the tasks and/or processes is called scheduling.

– The executive is a kernel combined with device drivers, high-level access functions at the inputs/outputs, and protocol-related drivers (TCP/IP, CAN, etc.).

– An operating system is an executive that also integrates an organ of dialog with the system (such as a *shell* or a windowing system), diagnostics, surveillance, adjustment, updates and development, etc.

Since this book deals with real-time scheduling, we will focus, in the following, on the functioning of the operating system kernel. A kernel provides the necessary primitives for the creation of tasks and for communication and synchronization. If it is a multi-process kernel, it also provides the corresponding primitives for the processes. Kernels in embedded systems, which represent a large part of critical real-time systems deal, for the most, with only one process, and consequently, we will mainly focus on the handling of tasks.

1.3.2. *Real-time operating systems*

Operating systems can be generalist or real-time. A generalist operating system prioritizes flexibility, ease of use and average processing speed. It has to be noted that accelerating the average processing speed using local optimizations can cause instances of tasks whose processing time would be longer than without any optimization. For instance, the principle of instruction preloading will preload and pre-decode the next instructions during the processing of an instruction. However, if the next instruction depends on

the result of an operation (conditional jump), the next preloaded instruction could correspond to the wrong operational branch. In this case, which happens rarely for well-designed prediction algorithms, the length of the instructions in time without any optimization could be shorter than the length of instructions with preloading optimization. Moreover, the determinism of the processing time is very much affected by the optimizations. This is the same for devices that prioritize flexibility (for example virtual memory, with the exchange mechanism between central memory and mass storage), or ease of use (for example the automatic update which will use up resources at moments difficult or impossible to predict).

The two most widespread generalist operating systems are *Microsoft Windows* and *Unix*. Both of these occupy a large disk space (around a gigabyte) and have a significant (several hundreds of megabytes) memory footprint (central memory usage).

Embedded CPUs, usually having a small amount of memory (a few kilobytes to a few megabytes of central memory) and a limited computing power (a few megahertz to a few hundreds of megahertz), for a mass storage round one gigabyte, real-time operating systems (RTOS) prioritize memory footprint and simplicity. Moreover, as we will see throughout this book, real-time is not fast, it is deterministic. Indeed, with a few exceptions, the temporal validation methods are conservative: when the system is validated, it is validated for the worst case. Indeed, an important metric characterizing an RTOS is kernel latency: this duration describes the worst delay in time that can elapse between a task-release event and when it is effectively being taken into account by the kernel. The internal architecture of an RTOS is designed to minimize this delay; to the detriment of the average processing speed.

There is a very large number of RTOSs and numerous standards defining RTOSs, implemented in various operating systems. We can point to the *portable operating system interface* (POSIX) standard *pthreads* 1003.1, which defines generalist RTOSs, the Ada standard that is very well adapted to very critical applications such as those which can be found in military and aerospace avionics, the OSEK standard developed by a consortium of European vehicle manufacturers, characterized by a very small memory footprint and a low increase in cost, and proprietary RTOSs such as VxWorks of WindRiver, or *real-time executive for multiprocessor systems* (RTEMS), which define their own primitives and provide a POSIX 1003.1-compliant interface.

1.3.3. *Primitives provided by the kernel*

Regardless of the generalist or real-time operating system, a certain number of primitives are provided for the management of parallelism. Since most RTOSs are mono-process, we will focus on the management of tasks. Figure 1.2 represents the possible states of a task such as they are perceived by the kernel. Only the ready tasks compete for the acquisition of computing resources, in other words for a core of a CPU.

– Task creation/deletion: the creation of a task consists of an initialization phase, followed by a launch phase. Initialization consists of designating the entry point of the task, which is generally a subprogram, attributing a control block that will serve to memorize the information important to the kernel in order to manage the task, (identifier, location to save the context of the task when it is preempted, such as the core registers of a CPU) and, except for special cases, allocating a stack for it which it will use to call subprograms and allocating its local variables. The launch phase consists of moving the process to the *ready* state, in other words notifying

the scheduling that it is ready to be executed and needs computing resources.

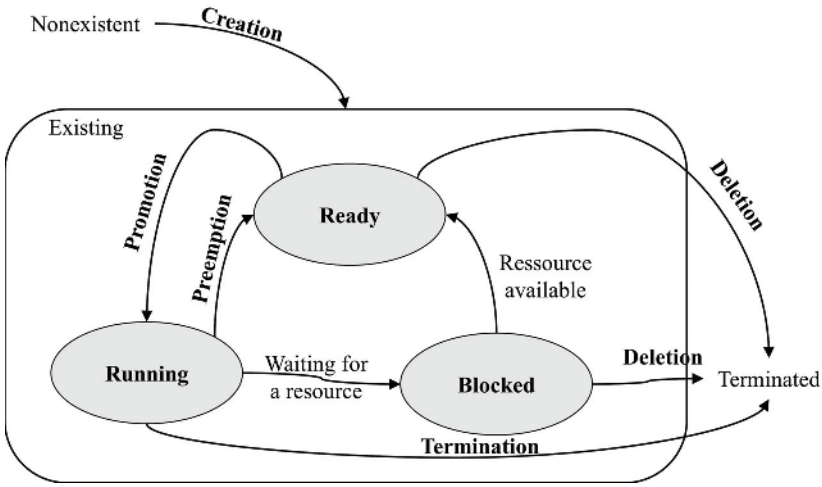

Figure 1.2. *Possible states for a task*

– Time management: most operating systems provide wait primitives either until a given date, or during at least a certain amount of time. A task running a wait primitive is moved to the *blocked* state and no longer competes for the acquisition of computing resources. At the given date, the operating system moves the task back into the *ready* state. Let us note that on most material architectures, the management of time is based on programmable clock systems (*timers*) allowing it to trigger a hardware interrupt after a required number of clock cycles. The kernel therefore uses the hardware interrupts generated by the clock in order to wake up the tasks at the end of their wait. There is therefore no computing resource usage by a task during the wait.

– Synchronization: when tasks share critical resources (same memory zone, material element that cannot be accessed in a concurrent manner, etc.), it is necessary to protect access to the critical resources by a synchronization mechanism that

guarantees the mutual exclusion of access. Current operating systems propose at least the semaphore tool and some, such as those based on the Ada standard (protected objects) or the POSIX standard (conditional variables), propose the Hoare monitor. When a task is blocked during the access to its critical section, it is moved to the *blocked* state, in other words it is the task that will release the critical section which will move another blocked task to the *ready* state.

– Message-based communication: most operating systems propose mailbox mechanisms based on the producer/consumer paradigm. A producer task generates messages into a buffer and can possibly move to a *blocked* state if the buffer is full. The consumer task can be put to wait for data: it is blocked if the buffer is empty and is woken up at the arrival of a message in the buffer.

– Inputs/outputs: when a task needs to perform blocking input/output, for instance accessing a mass storage unit, reading input from the keyboard, waiting for a frame on the network, etc., it starts the input/output, which moves it to the *blocking* state. The kernel, following the hardware interrupt corresponding to the expected answer from the input/output device moves the task to the *ready* state.

1.4. Scheduling

Scheduling, given a set of ready tasks, consists of choosing, on each core, at most one task to run.

1.4.1. *Online and offline scheduling*

Scheduling is based on a strategy of choice, which can be static or be based on an algorithm.

In the static case, we use a table in which we have predefined the allocation times of the tasks to the cores, the scheduler is then called a sequencer since it merely follows an

established sequence; we then refer to offline scheduling. This type of scheduling is only possible when the times the tasks will be ready are known beforehand, in other words the sequence-creation algorithm has to be clairvoyant.

When we use a selection algorithm on the task(s) to be executed based on the immediate state of the system, we refer to online scheduling. Online schedulers consider the set of ready tasks at certain moments of time in order to choose between them the allocation of the computing resources. No matter the method used to design the online strategy, it has to be resource-efficient. Indeed, the time spent by the computing resources for the scheduler to execute its strategy is called the processor overhead, since it does not directly contribute to the execution of the functionalities of the system. The computational complexity of schedulers must be linear, or even quadratic, in the worst case, depending on the number of tasks.

The moments when the scheduler has to make a choice correspond either to the moments when a task changes state (wakeup, blocking), in other words when there is a change of state in a task of the system, or to moments that are fixed by a time quantum. In the second case, the scheduler is activated at each time quantum and makes a decision depending on the immediate state of the tasks.

In RTOSs, most proposed scheduling algorithms are based on priorities. As we will see later, these priorities can be fixed-task (assigned to a task once and for all its jobs), fixed-job (assigned to a job once it is released) or dynamic.

A scheduling sequence can be represented on a Gantt diagram, such as in Figure 1.3. The time is in abscissa, each line represents the execution of a task, an ascending arrow represents the activation of a task whereas a descending arrow represents a deadline. In this figure, we consider two tasks τ_1 and τ_2 executed once on a processor, with respective

durations of 2 and 3 units of time. τ_1 is woken up at time 1 and τ_2 at time 0. We assume that the strategy of the scheduler is based on priorities, and that τ_1 is of higher priority than τ_2. Thus, at time 0, only τ_1 is ready and is given a processor. When τ_1 is woken up, the scheduler preempts τ_2, since the set of ready tasks is $\{\tau_1, \tau_2\}$ with τ_1 of higher priority than τ_2. At the termination of τ_1, the only ready task is τ_2, it therefore obtains the processor. At time 4, τ_2 misses its deadline.

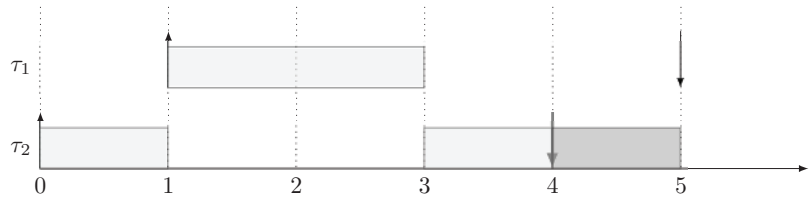

Figure 1.3. *Gantt diagram*

1.4.2. *Task characterization*

In a real-time system, most processing is recurrent, or even periodic. Each recurrent execution is called an instance or a *job*. Thus, in example 1.1, the reading of the gyro meter will presumably be periodic. Ideally, this would happen continuously, but since we are using a processor, our only choice is to discretize the process. The same applies to most of the input rhythms of the system, which are either periodic when the system needs to scan the state of a sensor, or triggered by the arrival of a message on the network or by another external event.

Each job of each task executes instructions, and consequently, uses up time on the computing resources. A task is therefore characterized by the duration of its jobs. Given that there could be loops with a number of iterations depending on the data, conditional paths, more or less efficient material optimizations determined by the execution

time or the data values, etc., the duration of the jobs of a task cannot be considered fixed. In a real-time context, it will be characterized by a *worst-case execution time* (*WCET*). The techniques used to determine the WCET are presented in Chapter 5.

Compliance with time-related requirements is mirrored on tasks by deadlines. In most models, each job is given a deadline that it has to comply with. Since every task can generate a potentially infinite number of jobs, each with a deadline, the temporal constraints are generally represented on a task τ_i by a relative deadline, often denoted by D_i. The relative deadline represents the size of the temporal window in which a job has to be executed from its activation.

Following their activation types, we distinguish between three kinds of tasks:

– Periodic tasks: tasks activated in a strictly periodic manner. A periodic task τ_i of period T_i is characterized by an initial release time, denoted r_i (as in *release*) or O_i (as in *offset*) in the literature. The first job starts at time $r_{i_1} = r_i$, and the task potentially generates an infinity of jobs, $\tau_{i,k}, k \geq 1$. The k^{th} job $\tau_{i,k}$ is woken up at time $r_{i,k} = r_i + (k-1)T_i$, and its deadline is $d_{i,k} = r_{i,k} + D_i$.

– Sporadic tasks: tasks activated by an event, in such a way that there is a minimal delay between two successive activations. This delay is seen as a minimal period T_i. In general the release time $r_{i,1}$ of the first job of a sporadic task τ_i is unknown, the activation time $r_{i,k} \geq r_{k-1} + T_i$, for all $k > 1$.

– Aperiodic tasks: tasks for which there is no known minimal delay separating two successive activations. In the case where an aperiodic task is given a deadline, we generally retain the right to accept or to refuse processing depending on whether or not we can guarantee compliance with the deadline. When it is not given a deadline, our goal will often be

to minimize its response time without affecting the deadline-compliance of the tasks under temporal constraints.

When an external event is periodic, before declaring a task triggered by this event as periodic, it has to be ensured that the timebase (material clock) used by every periodic task of the system is identical. Indeed, let us assume that a task run by a CPU is activated by a periodic message coming from another distant CPU, even if the sending of the message is completely regular from the point of view of the distant CPU, there is a drift, even a small one, which undermines the periodicity hypothesis. Indeed, as we go along, the effective release times of the task woken up by these messages will be more and more offset with respect to a periodic release.

The typical implementation of a periodic task on a RTOS is given in Figure 1.4, and that of a sporadic or aperiodic task is given in Figure 1.5. In the sporadic or aperiodic case, the trigger event is typically indicated by the occurrence of a hardware interrupt, for example, the arrival of a frame on an input/output bus or on a communication network. The distinction between sporadic and aperiodic comes from what characterizes the event expected by the wake-up of the task. In certain cases, the trigger event is characterized by a minimal delay between two successive activations. In case that the minimal delay between two successive activations cannot be determined, the task is aperiodic.

```
periodic task τ_i
    release=origin+r_i // origin gives the time reference
    do
        wait for release
        // code corresponding to a job of the task
        release=release+T_i
    while true
```

Figure 1.4. *Typical implementation of a periodic task*

In the case of periodic tasks, the initial release time is usually known, and we then refer to concrete tasks, whereas

in the sporadic and aperiodic cases, the trigger event often being external, it is difficult to know in advance the moment when the first job is triggered. We then refer to non-concrete tasks.

```
sporadic or aperiodic task τ_i
    do
        wait for trigger event
        // code corresponding to a job of the task
    while true
```

Figure 1.5. *Typical implementation of a sporadic task and an aperiodic task*

For periodic or sporadic tasks, the relationship between period and relative deadline is of great importance in the temporal study of the systems. We therefore distinguish between three cases for a task τ_i:

– Implicit deadline ($D_i = T_i$): this is historically the first model that has been studied. The deadline of a job corresponds to the release time of the next job.

– Constrained deadline ($\exists i, D_i < T_i$): the deadline of a job precedes the activation of the next job. In this case and in the implicit deadline case+, two jobs of the task τ_i can never be in competition for the computing resources.

– Arbitrary deadline ($\exists \tau_i, D_i > T_i$): jobs of τ_i can potentially be in competition for a processor. However, in general, we consider that a job has to be completely terminated before the next job can be executed. We refer, in this case, to the non-reentrance of tasks. Most RTOSs implicitly offer the principle of non-reentrance. For example, on Figure 1.4, a job, which corresponds to an iteration of the "`While true`" loop, cannot be executed before the end of the preceding job (iteration).

1.4.3. *Criticality*

In most temporal analyses, tasks are considered to have strict constraints, in other words in no circumstance can a

deadline be violated. Various studies have, however, been carried out on systems with fewer constraint tasks. For instance, the model $(m, k) - firm$ considers that m deadlines out of k have to be respected. More recently, the multi-criticality model was inspired by the criticality levels in civil avionics proposed by the DO178B/C. These levels of criticality represent the cohabitation of more or less strict-constraint tasks in a system of tasks.

1.4.4. *Metrics related to scheduling*

The main problem dealt with in the temporal study of an application concerns the deadline-compliance of the tasks, but several other metrics are of interest. In order to illustrate a few typical metrics used to describe a scheduling process, Figure 1.6 represents a fixed-task priority scheduling (the priority is assigned to the tasks, each job of a task is given the priority of the task). Considering the system of tasks, S, is composed of three periodic tasks with implicit deadlines (deadline equal to the period) τ_1, τ_2, τ_3 with respective WCETs of 3, 3, 3, with respective periods of 6, 9, 18 and with respective release times of 0, 1, 2. Given a system execution log, which can be called a scheduling sequence, we could characterize for example:

– Start time of a job: time at which the job acquires a computing resource for the first time. For example, in Figure 1.6, the second job $\tau_{2,1}$ of the task τ_1 has a starting time of 3.

– End time of a job: the job $\tau_{1,1}$ has an end time of 3.

– Response time of a job: corresponds to the difference between the end time and the release time. The response time of the job $\tau_{2,1}$ is thus 5. A job complies with its deadline if and only if its response time is not greater than its relative deadline.

– Response time of a task: corresponds to the maximum response time among the jobs of the task. Since there is a potentially infinite number of jobs, we will see in the next chapters how to determine this response time. In Figure 1.6, since the state of the system is the same at time 0 and at time 18 (all the jobs are terminated with the exception of a job of task τ_1 which arrived at that moment), then the infinite scheduling sequence is given by the infinite repetition of the sequence obtained in the time interval $[0..18[$. Consequently, the response time of the task τ_2 is given by the worst response time of its jobs, in other words 6. For τ_1, we can observe a response time of 3, and for τ_3 a response time of 16. Therefore, since every job complies with its deadline, the scheduling sequence is valid. We say that the system is schedulable by the chosen scheduling policy.

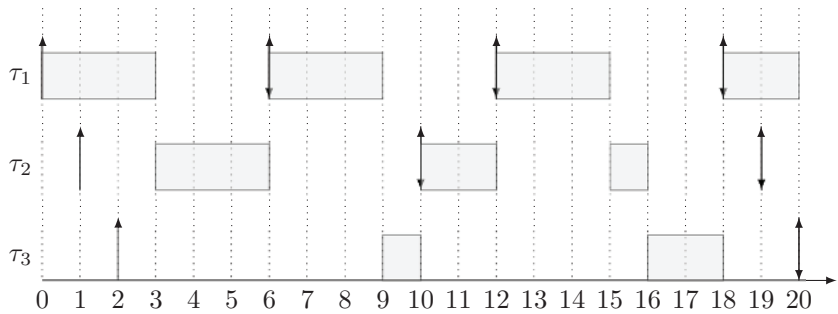

Figure 1.6. *Fixed-task priority scheduling of the system S*

– Latency of a job: difference between the current time instant and the deadline of the job. At time 5, the latency of the job $\tau_{2,1}$ is 5.

– Laxity or slack of a job: given by the difference between the latency and the remaining processing time to finish the job. At time 5, the laxity of the job $\tau_{2,1}$ is 4.

– Input/output delay of a job: difference between the start time and end time. Indeed, we often consider for tasks that

the inputs are performed at the beginning of the task, and the outputs at the end of the task. The input/output delay of $\tau_{2,1}$ is 3.

– Sampling jitter, response time jitter, input/output jitter: represents the variation, respectively, of the starting time (taking input, sampling for an acquisition task), of the response time of the jobs, and of the input/output delay. The jitters have an impact mainly on the quality of control performed by a corrector from the field of automatic control.

1.4.5. *Practical factors*

Various elements influence the behavior of the tasks, mainly the mutual exclusion during the access to critical resources, the synchronous communications between tasks implying precedences between jobs of different tasks, or the access to input/output devices which lead to the suspension of jobs.

1.4.5.1. *Preemptibility and mutual exclusion*

In real applications, tasks may have to share critical resources (shared variables, communication networks, a particular material element, etc.). In this case, as in every parallel application, the resources are protected in order to guarantee the mutual exclusion of their access. Typically, we use mutual exclusion semaphores or Hoare monitors which encapsulate the access to critical resources. This implies that parts of tasks cannot be mutually preempted. Figure 1.7 presents a typical task using a critical shared resource through a semaphore. In this case, we will differentiate between the duration of the code before, during and after the critical section.

Let us consider a system S_2 of three periodic tasks τ_1, τ_2, τ_3 run in parallel. Their temporal parameters, of the form r_i (release time), WCET (C_i, worst-case execution time), D_i

(relative deadline) and T_i (period), are given in Table 1.1. Tasks τ_1 and τ_3 share a critical resource for their entire duration. The mutual exclusion can for instance be ensured by a semaphore s. Thus, the entry into the critical section is subject to taking the semaphore s, and when a job tries to take s when it is already taken, it is moved to the *blocked* state. At the release of the semaphore, a job waiting for it is moved to the *ready* state and is thus put back into competition for a computing resource.

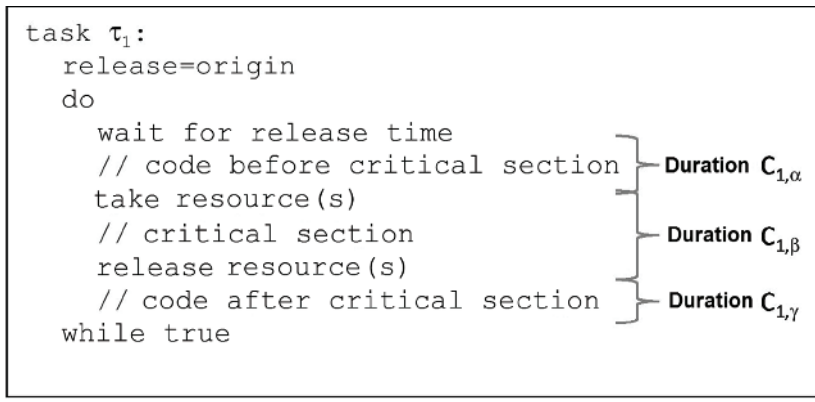

Figure 1.7. *Typical implementation of a task using a critical resource*

	r_i	C_i	D_i	T_i
τ_1	0	2	6	8
τ_2	0	6	15	16
τ_3	0	6	16	16

Table 1.1. *S_2 system parameters*

We assume that the scheduler is fixed-task priority based, and that the task τ_1 has higher priority than τ_2 which has higher priority than τ_3. Figure 1.8 represents a scheduling of the system when every task lasts as long as their WCET. The

dark parts represent the critical sections, which prevent τ_1 and τ_3 from preempting each other.

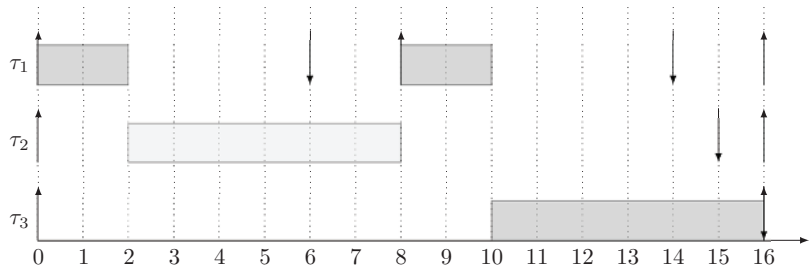

Figure 1.8. *Scheduling of system S_2*

The scheduling sequence is valid, and the system is in the same state at time 16 as at time 0. In consequence this scheduling sequence can be indefinitely repeated. However, if the system is scheduled online, it would be a serious mistake to conclude that the system is schedulable, since during the execution of the system, it is possible, and even very frequent, that the duration of the tasks is lower than their WCET. Therefore, if the task τ_2 only uses 5 units of time to run, we obtain the sequence in Figure 1.9.

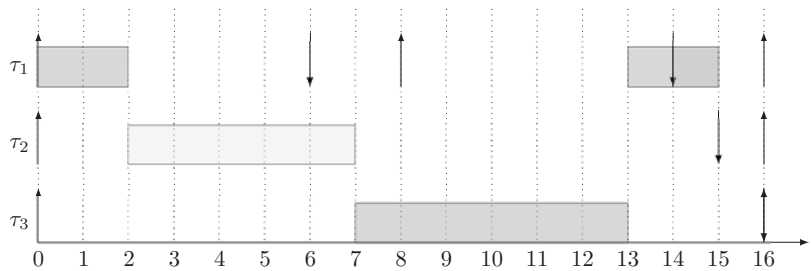

Figure 1.9. *Scheduling of system S_2 with τ_2 being shorter than expected*

In this sequence, at time 7, $\tau_{3,1}$ is the only active job, it therefore acquires the processor and begins its critical section. At time 8, even though $\tau_{1,2}$ is the active job with the highest priority, it will move into the *blocked* state since it

cannot enter into its critical section. It is only at the end of $\tau_{3,1}$'s critical section that $\tau_{1,2}$ is woken up and acquires the processor.

The observed phenomenon is called a scheduling anomaly: while the tasking system needs less computing resources, the created scheduling sequence is less favorable, in the sense that the response time of some tasks increases. When the scheduling is online, several parameters can vary: the parameter which can always vary in the online case is the execution time. When the tasks are sporadic, the period can also vary. In most realistic systems, tasks are likely to share critical resources. An online scheduling of tasks which shares resources can be subject to anomalies when the durations decrease. We say that it is not $C-$sustainable: the concept of sustainability is presented in section 1.5.2. This is the same for the period: if it increases, some response times might increase. Consequently, the online scheduling of tasks under resource constraints is not $T-$sustainable.

The non-preemptive case can be seen as a specific case of critical resource sharing: everything happens as if every task shared the same critical resource, preventing them from being mutually preempted.

1.4.5.2. *Precedence constraints*

When tasks communicate by messages, mainly when the messages are transmitted through mailboxes, also called message queues, a task waiting for a message has to wait for the transmission of a message by another task. The receiving task is thus subject to a precedence constraint. On the left-hand side of Figure 1.11, a system of four tasks communicating by messages is presented. The simple precedence model is such that two tasks subject to a precedence constraint (predecessor and successor) have the same period. In every known real-time model, precedence constraints form an acyclic graph. The typical code of the task τ_1 is given in Figure 1.10.

```
task τ1
    do
        // code before transmission of m1
        send message m1
        // code after transmission
        wait for message m2
        // code after reception of m2
    while true
```

Figure 1.10. *Typical implementation of the task τ_1 presented in Figure 1.11*

A reduction to normal form consists, when every task has the same period, in cutting the tasks around the synchronization points, in other words putting the transmission at the end of task and the wait for messages at the beginning of task. On the right-hand side of Figure 1.11, the task τ_1 is thus cut into three tasks $\tau_{1,1}$, $\tau_{1,2}$ and $\tau_{1,3}$. The reduction is such that, from a scheduling point of view, the systems before and after the reduction are equivalent.

When the communicating tasks have different periods, we refer to multi-periodic precedence constraints. In this case, the reduction is done on job-level instead of task-level, since the jobs of a same task are subject to different precedence constraints.

1.4.5.3. *Activation jitter*

The activation jitter is a practical factor commonly employed in cases where the tasks wait for messages coming from a network, or to model the delay that can be taken up by the messages in switched networks. The activation jitter represents the uncertainty regarding the possibility to execute a task as soon as it is woken up. Usually denoted as J_i, from a task model point of view, the jitter is such that, given the release time $r_{i,j}$ of a job of the task τ_i, it is possible that it is only able to start its execution at an uncertain moment of time, between $r_{i,j}$ and $r_{i,j} + J_i$. For instance, let us consider a periodic task executed on a distributed system,

supposed to be waiting for the message m_k with same period coming from the network. We could pose J_i = response time(m_k) to represent the fact that the arrival delay of the message m_k with respect to the expected release time of the job can vary between 0 and J_i.

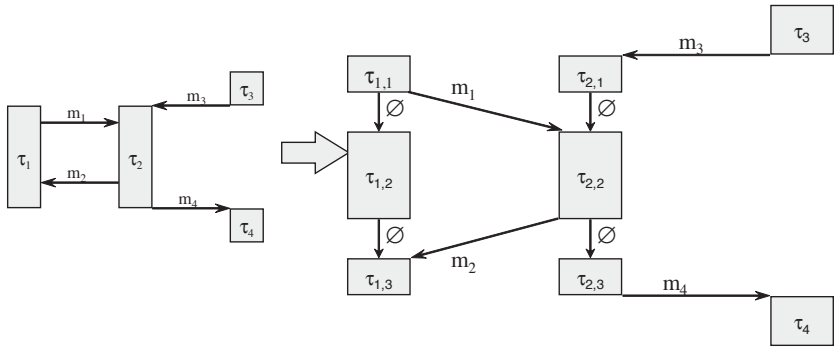

Figure 1.11. *Reduction to normal form of precedence constraints when the periods are identical*

1.4.5.4. *Suspensions*

Suspensions are another practical factor considered by some task models. Indeed, a task performing, for example, an input/output, is suspended during the operation. For instance, a task which accesses a file on a mass storage device first initiates the input/output and is then suspended, in other words moved to the blocked state, and is reactivated when the input/output device responds. The task is thus suspended during the input/output. The duration of suspension is usually difficult to predict exactly, and is generally bounded by an upper limit.

1.4.6. *Multi-core scheduling*

Recent material architectures build on the miniaturization of transistors in order to integrate several computing cores on a single chip. We refer in this case to multi-core processors.

These architectures are already present in non-critical fields, from personal computers to smartphones. They are also to be generalized in critical systems, since from a technological point of view, the miniaturization of transistors approaching the size of an atom, the propagation speed of electric current in the circuits for a reasonable energy and therefore the computing frequency of the processors will soon reach a limit. Since 2005, the increase in processor computing power is therefore mainly ensured by the increase of the number of computing cores and no longer by the increase of computing frequency as before.

Multi-core architectures are more and more complex, and with the increase in the number of cores, the passage of data between the cores is becoming a real problem and can even become a bottleneck. In real-time scheduling, we are therefore not only interested in the scheduling of processes in the cores, but also in the internal network traffic of the processors, which is called *network-on-chip* or NoC. Finally, let us note that numerous multi-core architectures use hierarchical cache memories, which complicates the computing of the WCET.

Figure 1.12 presents these three views: in part (a), a completely abstract architecture, assuming uniform access to memory, is used by the task-scheduling analysis on the processors. In this architecture, we generally ignore the data transmission delays between the cores as well as the migration delays. The migration of a task consists of starting it on one core and continuing on another core. In the mono-core case, the preemption delay is usually assumed to be included in the WCET of the tasks. However, the architecture presented in part (c) shows that this hypothesis is very restrictive, since in the case that a task migrates from core 1 to core 2, only the data from the task present in level 1 cache has to be synchronized, whereas if a task migrates from core 1 to core 3, the data in the level 1 and 2 caches has to be

synchronized, which would take more time. In case (c), the data will have a different path to travel depending on the original core and the destination core, which would take a different amount of time. It is therefore important to consider the effective hypotheses after reading the results presented in the multi-processor chapter.

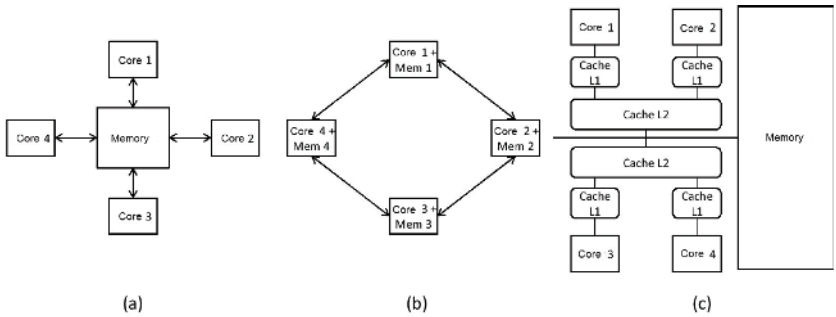

Figure 1.12. *Three multi-core views: (1) simplified view of scheduling in the cores, (2) NoC and (3) cache hierarchy and WCET*

One way to reduce the impact of the restrictive material architecture-accounting hypotheses is to limit, or even to remove, migrations. In the multi-processor case, we therefore consider several hypotheses of migration:

– Total: every job can migrate at any given moment in time.

– Task-level: a job can not migrate, but a task can migrate between its jobs.

– None: tasks are assigned to a single core.

These hypotheses result in three types of multiprocessor schedulers:

– Global scheduling: there is a single scheduler for all cores, and a single set of ready tasks. For m cores, the scheduler chooses, at every moment in time, up to m ready jobs to be assigned to a computing core.

– Partitioned scheduling: there is a scheduler for each core. Tasks are assigned to a single core which is then managed as a uniprocessor core. The issue of scheduling is in this case an issue of assignment, which is a knapsack-type of problem.

– Semi-partitioned scheduling: only certain jobs are allowed to migrate, in such a way as to increase scheduling performance with respect to partitioned scheduling, while limiting the impact of migration cost.

We usually consider, even in the multi-core case, that the tasks and the jobs are not parallelizable. However, in some cases, parts of jobs can be simultaneously executed on several cores. For example, we may find *directed acyclic graph* (DAG) task models or parallelizable tasks. In the case of DAG tasks, each graph node is a part of a task which can be parallelized with relation to the other parts of tasks while respecting the precedence constraints between parts of tasks.

1.5. Real-time application modeling and analysis

1.5.1. *Modeling*

This section summarizes the different parameters and practical factors commonly employed in temporal analysis. These elements are based on the way tasks work in reality.

– BCET, WCET C_i: best and worst execution time of each job of a task, also used to represent the transmission time of messages on a communication medium.

– r_i: release time of the first job of a task, only known when the task is concrete.

– D_i: relative deadline of a task. We distinguish between constrained deadlines, implicit deadlines and arbitrary deadlines.

– T_i: period of a task, minimum delay between the activation of two successive jobs of a sporadic task.

– J_i: release jitter of a task.

– Critical resources: represents mutual exclusion.

– Precedence constraints: expressed as a DAG, these constraints represent the precedence, often linked to the data, between jobs. Simple precedences link tasks with same periods, while multi-periodic precedences link tasks with different periods.

– Suspensions: represent the suspension time linked to input/output accesses.

Various models have been proposed to closely represent the relationships between the tasks. Thus, for instance, for some sporadic tasks, even if we do not know the activation time of a task in advance, we know the activation scheme of some tasks if we know the activation of the first. Let us assume, for example, that a task is activated by the arrival of a frame on the network, and that this frame is always followed, after a certain delay, by a second frame activating a second task. The activation of the second task is therefore conditional to the activation of the first.

1.5.2. *Analysis*

We can break down the view of a real-time system into three parts:

– Computational resources: processors, networks, or switches, these resources allow the execution of jobs or the transmission of messages.

– Scheduling or arbitration: technique implemented to rationally distribute the resources to the jobs or messages.

– Jobs or messages: of recurrent nature, jobs or messages represent the need in computational resources of the application. If the application is real-time, these elements are subject to temporal constraints represented by deadlines.

We will call configuration the set of computational resources, scheduling policies (or arbitration), and jobs or messages together with their constraints.

The temporal analysis of the system can be broken down into several problems:

1) Scheduling: consists of ensuring that for a given configuration, the temporal constraints of the jobs or messages are always respected.

2) Sensitivity analysis: based on a configuration in which certain parameters of jobs or messages are ignored, consists of establishing a domain for these parameters such that the configuration obtained for any value of the domain is schedulable.

3) Dimensioning: based on a configuration in which the computational resources are unknown or partially known, consists of choosing the minimum number of resources such that the configuration is schedulable.

4) Choice of policy: consists of choosing a scheduling policy such that the configuration obtained is schedulable.

5) Architecture exploration: given computational resources, find a task assignment, and communication mapping into messages, mapped into networks, such that the temporal constraints are met.

Every problem is based on the scheduling problem, which is therefore of central importance in this book. The standard definitions related to this problem are as follows:

DEFINITION 1.1 (Schedulability).– *A system of tasks is schedulable by a scheduling algorithm if every job created will meet its deadline.*

DEFINITION 1.2 (Feasibility).– *A system of tasks is feasible if it is schedulable by at least one scheduling algorithm.*

DEFINITION 1.3 (Schedulability test).– *A schedulability test is a binary test returning yes or no depending on whether the configuration is schedulable, or whether the system is feasible.*

Schedulability tests are usually conservative, and we then refer to sufficient tests: if the answer is yes, the configuration is schedulable, but if the answer is no, then it is possible that the configuration is schedulable but the test cannot prove it. In some simple academic cases, accurate tests are available, in other words tests that are necessary and sufficient.

We have seen that in certain cases, such as in the presence of critical resources, scheduling anomalies may occur. In consequence, the sustainability of schedulability tests should be defined.

DEFINITION 1.4 (Sustainability).– *A schedulability test is $C-$sustainable ($T-$, $D-$, $J-$ sustainable, respectively) if, in the case that the answer is yes, the configuration remains schedulable if we reduce the execution times (increase the periods or relative deadlines, reduce jitter, respectively).*

The concept of sustainability, mainly $C-$sustainability, is paramount for every online scheduler. A schedulability test has to be $C-$sustainable in order to be used online. In the case that a test is not $C-$sustainable, for example a simulation for a system sharing resources, then it can only be used offline. Indeed, the scheduling sequence in Figure 1.8 remains valid if it is indefinitely played offline by a sequencer.

1.6. System architecture and schedulability

The challenges of temporal analysis are strongly linked to the real-time systems design engineer whose role, during the design of the system, is to fashion the CPUs and the networks and to assign the functionalities of a system to the tasks. This section therefore creates the link between the choice of assigning two communicating functions on tasks, CPUs, networks and the worst-case delay between the beginning of the first function and the end of the second function, assuming, for the sake of simplicity, that the tasks will have an implicit deadline. To simplify things, let us consider two functions A and B running periodically with same period T, represented at the center of Figure 1.13. Function A precedes function B. For each choice of architecture, we compute the worst-case end-to-end delay D, in other words the worst-case delay between the release of A and the end of B.

In case (a), the resulting architecture is the sequence in the same task. The task having an implicit deadline and the system deemed schedulable, the worst-case response time of the task is less than or equal to the period, the worst-case end-to-end delay is therefore less than or equal to the period of the task: $D \leq T$.

In case (b), the function A is executed in a periodic task τ_A with period T, which precedes the task τ_B executing B, with the same period. The end of execution of τ_A triggers task τ_B. There are several possible models to represent the two tasks: the first will presumably be modeled by a periodic task with period T. If the system is strictly periodic, the task τ_B may be modeled by a task whose release time is offset, for example by T with respect to τ_A. In this case, the worst-case execution time is $D \leq 2T$ since each task has to execute in its window of size T and that the two windows are offset one after the other.

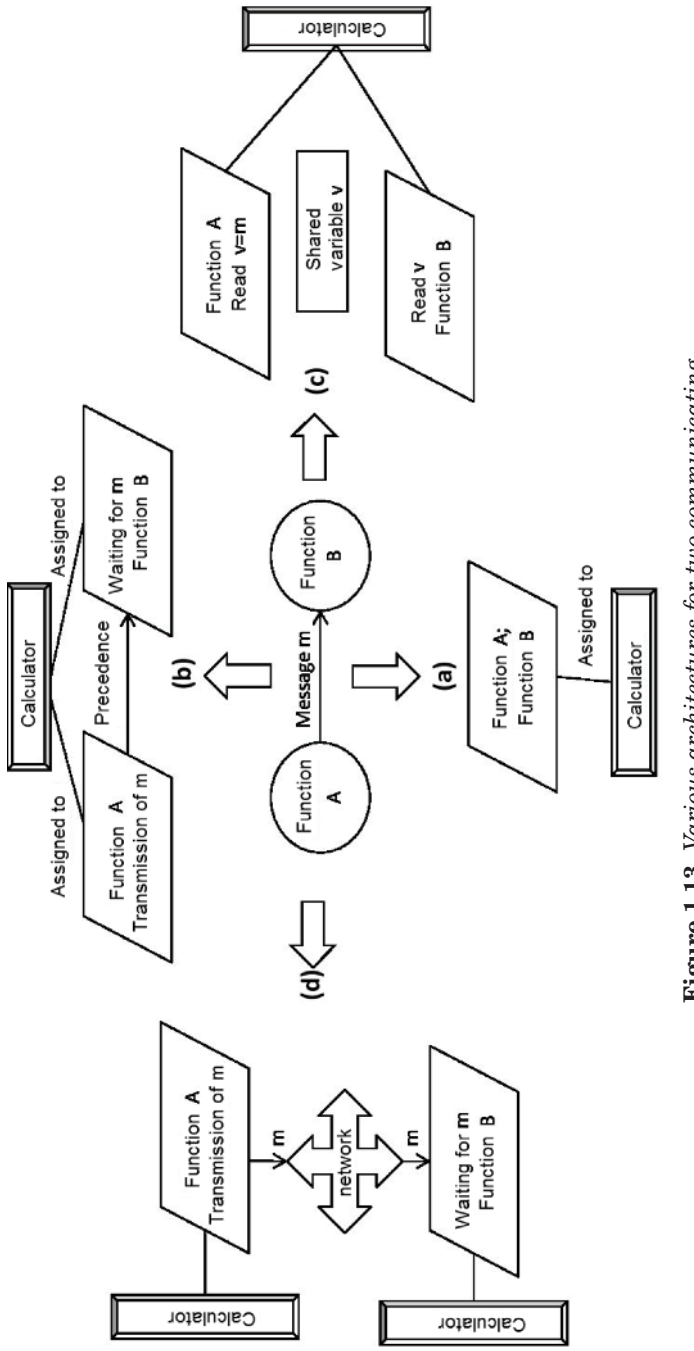

Figure 1.13. *Various architectures for two communicating functionalities*

In case (c), the communication of m is carried out in an asynchronous fashion by a shared variable v. Assuming that the two tasks are periodic with period m, the release case that is the most unfavorable for the end-to-end delay occurs when the task τ_A which runs A terminates the latest, in other words the variable v is modified T units of time after the release of τ_A. Moreover, we consider that τ_B has started the earliest possible and read the variable v just before its modification by τ_A. The variable v will be read, at the latest, at the end of the newt period of τ_B, in other words $2T$ later. In consequence, the worst-case delay is $D \leq 3T$.

In case (d), the tasks are placed on two different CPUs, and the message m is transmitted through a communication network. We assume that the message, with period m, has a transmission delay less than or equal to T (this will be proven by the schedulability analysis of the network). In the worst case, from the point of view of the processor running the task τ_A, the maximum delay between the release of τ_A and the transmission of the message on the network is assumed to be T (the system is deemed schedulable), and afterward the transmission delay on the network is at most T. At most $2T$ after the release of τ_A, the message arrives on the second processor. Depending on the type of implementation of the task τ_B running B, we could use here several models for this task. If the task is triggered by the arrival of the message, then there has to be T between the moment the message arrives and the moment where τ_B terminates, which gives $D \leq 3T$. If the task τ_B is time-triggered, in other words it is executed periodically and at each release it considers the last arrived message, then we are in the case of an asynchronous communication between the arrival of the message on the CPU and its taking into account by the task τ_B, as in case (c), this asynchronism costs at most $2T$, in this case, the end-to-end delay is therefore $D \leq 4T$.

We can thus see that the choice of the software implementation (tasks) and material implementation (allocation to processors and networks) has a very large impact on the end-to-end response time, ranging here from T to $4T$ depending on the choice of architecture. A close analysis of schedulability can strongly help in reducing this delay, for instance by tuning the relative deadlines, it is possible to reduce the windows in which the tasks are executed.