Unit 4

## [Synchronization in Distributed Cyber-Physical Systems](#_bookmark3)

#### [Introduction and Motivation](#_bookmark3)

* In distributed systems, many of the computations are periodic in nature and are triggered by periodic timers based on the local clocks.
* The local clocks’ relative skew can be bounded but cannot be entirely eliminated.
  + - Embedded system designers model these distributed computations using the “**globally asynchronous, locally synchronous” (GALS).**
    - Because of the clock skews in a GALS system, a small timing difference in the execution and communication delay can lead to **distributed race conditions.**
    - To illustrate this problem, consider an example of a triplicated redundant control system that receives a new reference position or setpoint command from a supervisory controller. Because of the non-zero clock skews, one controller (say, controller A) could be in its local clock–based period *j* + 1, while the other two (controllers B and C) are still in their local clock–based period *j* at the time when they receive the setpoint.
    - Since they receive the setpoint at different periods, controller A’s control command might diverge from the other two control commands and, therefore, would be voted out.
    - This race condition leads to an invalid failure detection of controller A (even though it is not actually faulty).

Period *j* Period *j*+1 Period *j*+2



supervisory controller

controller A

controller B

controller C

clock skew

clock skew

clock skew

**Figure 8.1:** *Triplicated control in a GALS system*

##### [Challenges in Cyber-Physical Systems](#_bookmark3)

* A system may operate correctly for a long time, even for years, but suddenly fail after a change in some logically unrelated hardware, software, or workloads.
* Tracking down the root causes of these problems is like finding a needle in a haystack. This may lead to the state space explosion problem in a GALS system.
* For example, in a fly-by-wire aircraft, the control surfaces are each locally controlled by higher-level supervisory controllers operating at different rates.
* These controllers are deployed redundantly for fault tolerance..

##### [A Complexity-Reducing Technique for Synchronization](#_bookmark4)

* Synchronization in the traditional systems happens at the networking layer at the expense of specialized hardware.
* These techniques, whether implemented in hardware or software, also have a limitation in that they often rely on architecture-dependent semantics to guarantee consistent views and actions which increases cost of verification and certification.
* “**Physically asynchronous, logically synchronous” (PALS)** system, is a formal architectural pattern that eliminates race conditions arising from the asynchronous interactions.
* This pattern makes use of the basic features of the networked cyber-physical systems like, **bounded clock skew** and **bounded end-to-end delay** to simplify the interactions of the distributed computations.
* The pattern distributes this synchronous design over a physically asynchronous architecture without any modification of application logic and properties.
* Any temporal logic formula that holds under a globally synchronous system also holds for an asynchronous system PALS.
* PALS system can work with commercial off-the- shelf (COTS) components and without specialized hardware.

#### [Basic Techniques](#_bookmark4)

##### [Formal Software Engineering](#_bookmark4)

* A pattern can be viewed as a design template of the solution to a generic problem.
* Synchronous design languages and tools, such as Simulink, SCADE, and Lustre, are also widely used in CPS development.
* The software components in a synchronous design language were originally intended to be only centralized and driven by a global clock.
* As a result, these techniques, by default, lack support for architectural-level analysis of distributed software components.
* The PALS architecture pattern complements these languages and tools.

##### [Distributed Consensus Algorithms](#_bookmark4)

* Distributed consensus is a fundamental concept in distributed systems and theory, and **virtual synchronization**.
* This virtual synchrony model guarantees that the behavior of the replicated processes is indistinguishable from the behavior of a single reference process on a non faulty node.
  + - The group communication service maintains a list of active processes and notifies the processes of join or crash events, known as **view change events.**
    - Unfortunately, these techniques do not provide hard real-time guarantees or timing bounds for when a synchronization will be complete.
    - When an application needs to send real-time messages, it presents the message with timing constraints to an admission controller, which then performs online **schedulability analysis**.
    - Real-time messages that can be scheduled are admitted; otherwise, they are rejected.
    - **First**, these services are primarily used as the transport-layer services for reliable and consistent message communications in a group of computations.
    - **Second,** the group communication services, by default, guarantee reliable multicast of individual application messages, which have more complexity than is needed in many real-time applications.
    - **Third**, the group communication services bundle various fault- tolerance mechanisms such as group membership and state transfer upon initialization.
    - A famous theory on the impossibility of distributed consensus suggests that **no algorithm can always reach consensus in a bounded time** in this model of asynchronous system, even with a single process crash/failure.

##### [Synchronous Lockstep Executions](#_bookmark4)

* Various Protocols have implemented a synchronous model onto different asynchronous architectures, such as a **Loosely Time-Triggered Architecture (LTTA)** and an **asynchronous bounded delay (ABD**) network.
* In LTTA, the mapping is achieved through an intermediate finite FIFO platform (FFP) layer. Although correctness is achieved in spite of unpredictable network delays and clock skews, these approaches **do not provide the hard real-time guarantee** required for synchronization and consistent views in cyber-physical systems. Furthermore, this work **does not handle any failure and multirate computations.**
* An ABD network primarily assumes that the message transmission delay is bounded.
* The protocols simulating ABD, define the logical synchronization period in terms of the **round intervals** for different network topologies, where each round interval gives an upper bound for the message transmission delay.
* However, they **do not assume that a fault-tolerant clock synchronizer** is applied to the local clocks that are required in net- worked control systems such as avionics. As a result, they require a complex reinitialization procedure to correct the clock drift errors
* They do not discuss **node failure and reliable message communication, and multirate computations.**
* Another solution gives three protocols for achieving logical synchronization: **α synchronizer, β synchronizer, and γ synchronizer**. It requires a leader node for synchronizing activity. As a result, these solutions require longer synchronization periods and have high overhead to maintain a verifiable leader-election logic with respect to failures and other asynchronous events.

##### [Time-Triggered Architecture](#_bookmark4)

* Time-triggered architecture (TTA) was one of the earliest system archi- tectures that introduced distributed real-time clock sources for main- taining consistency.
* The core functions of TTA are implemented in a custom network architecture, such as TTP/A and TTP/C, for reliable message communications.
* TTA-Group recently introduced TTEthernet, a real-time switched Ethernet network system that employs TDMA-based message deliveries and hardware-based time synchronization.
* Based on this clock, TTA can ensure that the temporal order of events can be recovered from their timestamps, provided that the difference between their timestamps is equal to or greater than 2 ticks.
* An additional effort is required **to coordinate the distributed interactions**.
* PALS system, do not require knowledge of the global message schedule of the time-triggered network architectures to perform logical synchronization.
* The PALS pattern uses only the system’s performance parameters to abstract away the underlying network architecture

#### [Advanced Techniques](#_bookmark4)

we require a new technique to address these issues that, at the same time, simplifies the verification of these systems.

##### [Physically Asynchronous, Logically Synchronous Systems](#_bookmark4) (PALS)

* In PALS, for each dispatch, a task reads the messages from its input ports, processes the messages, and sends its output messages to other nodes.
* In this lockstep execution, messages generated during period *j* are always consumed by their destination tasks in period *j* + 1.

1. Globally Synchronous System

Task 1

msgs

msgs

Period *j*

Period *j*  1

Task 2

Task 3

msgs

msgs

Period = *T*

1. PALS System

PALS clock PALS clock

Task 1

Task 2

Task 3

2

period *j* period *j*  1

msgs msgs

msgs msgs

PALS clock period = *T*

Time

**Figure :** *Logically equivalent globally synchronous system and PALS system*

* The PALS system guarantees an equivalent lockstep (synchronous) execution in the physically asynchronous architecture.
* In this system, each node defines a logical clock, called the PALS clock of period *T*, for each distributed computation.
* Each node triggers the computation logic at the start of a PALS clock period.
* The *j*th PALS clock period starts at the local clock time of a node, which is defined as *c*(*t*) = *jT*.
* The local clocks are asynchronous. Thus, the PALS clock periods do not

start at the same global time.

* Instead, the PALS clock periods at dif- ferent nodes begin within well-

defined intervals in global time.

* Despite the physical asynchrony of the PALS clocks, this pattern ensures that messages generated during period *j* are consumed by their destination tasks in period *j* + 1.

###### PALS System Assumptions

For a **CPS** to be structured as a PALS system, its architecture must satisfy a set of assumptions.

The PALS system assumptions can be organized into three categories: **system context, timing, and external interface** constraints.

###### System Context

The PALS pattern is applicable in hard real-time networked systems with the following features and bounded performance parameters:

* Each node has a monotonically **nondecreasing local clock,**

***c***: Time → Time (Time = *R* ≥ 0). Here, *c*(*t*) = *x* is the local clock time *x* at the “ideal” global time *t*.

* The local clocks are synchronized to the extent that corresponding local clock times happen within a **2ε interval in global time**. Let *c*(*t*) = *x*. Then *t* happens in the global time interval [*x* − ε, *x* + ε], where **ε is defined as the worst-case clock skew** with respect to the global time.
* Response time of a computation task α is bounded: 0 < αmin ≤ α ≤ αmax.
* Message transmission delay μ is bounded: 0 < μmin ≤ μ ≤ μmax.
* Nodes demonstrate fail-stop behavior and may recover later. A failed node must not be able to send extra messages during the current period.

###### Timing Constraints

The following constraints on the system parameters must be also satis- fied in the physically asynchronous architecture:

* + *PALS causality or output hold constraint:* Since the PALS clock events are not perfectly synchronized, delivering a task’s outputs too early may result in the violation of the lockstep synchronization. A task is **not allowed to send a message earlier than** *t* = (*jT* + *H*) in period *j*, where *H* = max(2ε − μmin, 0).
  + *PALS clock period constraint:* The distributed computation of a node requires at least the delay of end-to-end computation and message transmission to know the state of the computations in other nodes. The PALS system defines an **optimal lower bound for the PALS clock period,**  *T* > 2ε + max(αmax, 2ε−μmin) + μmax.

###### External Interface Constraints

* The PALS pattern defines additional constraints for interaction with external components.
* The environment input synchronizer, to coordinate the logically synchronous delivery of external messages.
* In its simplest form, an environment input synchronizer is a periodic task satisfying both the PALS clock period constraint and the PALS causality constraint.
* Suppose the environment input synchronizer receives external messages during period *j*.
* The environment input synchronizer then forwards these messages to the final destination tasks during period *j* + 1 so that the receiving tasks receive the external messages consistently in the same PALS clock period.
* Similarly, one may use an environment output synchronizer to maintain a consistent view of the PALS computations at the external observers.

###### Pattern Extension for Multirate Computations

* + - The PALS system has been extended to support the logical synchroniza- tion of multirate distributed computations in a pattern called multirate PALS system.
    - This extension also defines a sequence of periodic PALS clock events, when the local clock C equals *jT*hp, ∀ *j* ∈ *N*. Here *T*hp is the PALS clock period.
    - In multirate PALS system, the PALS clock period is chosen as the hyper-period or the least-common-multiple (LCM) of the periods of the participating tasks.
    - Suppose that *M*1 and *M*2 are two communicating periodic tasks of period *T*1 and *T*2, respectively.
    - Let the PALS clock period equal *T*hp, where *T*hp = LCM(*T*1, *T*2).
    - The multirate PALS system pattern guarantees that *M*1 receives a total of *n*2 = *T*hp/*T*2 messages from *M*2 generated during the PALS clock period *j*.
    - Similarly, *M*2 receives a total of *n*1 = *T*hp/*T*1 messages from *M*1 in the same PALS clock period. The pattern filters these received messages and delivers a selected message identically during the next PALS clock period. Figure illustrates this example.

PALS clock period *j*

*M*1 *M*2



*d*1

*d*1

*d*2

*d*1

*d*2

*T*1

*T*2

*Thp*

1

2

1

1

3

2

1

PALS clock period *j*  1

**Figure :** *Multirate PALS system-based interaction*

###### PALS Architectural Specifications

* The PALS system pattern is particularly well suited for formal specifi- cation and analysis in architectural modeling languages.
* One can vali- date the architectural models of a PALS system to manage the complexity of logical synchronization in cyber-physical systems.
* These requirements identify a set of simple constraints that are readily checked in the AADL (Architecture Analysis and Design Language) models.
* By ensuring that the models satisfy these constraints, we can easily guarantee the correctness of the logical synchronization in the physically asynchronous models.
* In AADL, there are two specifications for the PALS system design: **Synchronous AADL and PALS AADL**.

###### Synchronous AADL Specification

* The first step of the PALS system design is the design and verification of the synchronous model.
* This Synchronous AADL specification models the lockstep execution of the globally synchronous computations.
* This AADL specification includes just two types of components:

(1) the **components** that participate in the distributed computation in a lock-step manner and

(2) the **external environment**, which sends external inputs to them. Both types of components are modeled using the AADL thread construct that dispatches messages periodically at the same rate.

* The Synchronous AADL specification models the message com- munications between the threads using the delayed data port con- nection semantic.
* AADL models are not executable, however, and they cannot be automatically verified.
* To support formal verification, the Synchronous AADL models are automatically translated to Real-Time Maude for model checking and simulation.

###### PALS AADL Specification

* + The next step in the PALS system design process is to map the Synchronous AADL model to a physically asynchronous system model. The concept of this specification is based on architectural transformation.
  + In AADL, this transformation involves component inheritance—that is, annotation of the components and connections with the pattern-specific properties.
  + For example, in the asynchronous model, we annotate the threads and connections of the corresponding Synchronous AADL model with two properties: PALS\_Properties::PALS\_Id and PALS\_Properties::PALS\_Connection\_ Id.
  + These properties accept a **string literal** as the value. With these two properties, we define a **logical synchronization group** that consists of a set of components and connections with the same value for these properties.
  + An AADL model checker can load the system model and validate the PALS system assumptions.
    - For example, for each thread *Mi* in a logical synchronization group, we validate the PALS clock period mentioned earlier and PALS causality constraints by simply evaluating the following conditions:
    - *Mi.Period* > 2ε + max(*Mi.Deadline*, 2ε − μmin) + μmax
    - min(*Mi.PALS Properties::PALS\_Output\_Time*) > max(2ε − μmin, 0)
  + Here, *Mi*.*Period* is the extracted value of the AADL property period defined by the thread *Mi*.
  + A thread defines an AADL property, called PALS Properties::PALS\_Output\_Time.
  + It gives the earliest and the latest time at which a thread transmits its outputs since the dispatch.
  + The remaining performance parameters are computed based on other standard AADL properties, such as latency for message transmission delays (μmin, μmax) and Clock\_Jitter for the local clock skew ε.

###### PALS Middleware Architecture

* A distributed middleware, called PALSware, which implements the task execution and communication services for logically synchronous interactions.
* Figure shows the software architecture of the PALS middleware. It consists of three layers: **infrastructure, middle-ware, and application**.
* This layered architecture makes the software portable and extendable in different platforms.
* At the *infrastructure layer*, the middleware defines **generic interfaces** to integrate a fault-tolerant clock synchronizer and a real-time network architecture.
* As long as the pattern’s assumptions are satisfied, the **application logic is not affected** when these infrastructure services change or they are developed with COTS components.
* At the *middleware layer*, **PALSware addresses several implementation challenges**. In particular, task failures increase the system’s complexity. For example, a task may send the same message sequentially to different destination tasks. If this task suddenly stops, only a subset of the destination tasks may receive this message , which subsequently leads to inconsistent states.

a. Application layer

b. Middleware layer

Each

period

pals\_send

pals\_recv

Current skew

Current messages

time

End- markers

c. Infrastructure layer

Real-time network architecture

Clock synchronizer

PALS communication library

PALS task library

PALS task logic

PALS fault manager

**Figure :** *PALS middleware architecture layers*

* + PALSware uses a communication protocol to guarantee atomicity in the distributed interactions, so that the destination tasks have consistent information about the working or failed state of a source task.
  + At the *application layer*, application developers implement the

**periodic application logic** for the distributed tasks.

* + This layer also supports runtime monitoring for any faults that may affect the logical synchronization. For example, if the clock skew assumption is violated for some reason, the distributed tasks will no longer be logically synchronized within the 2ε interval in global time.

###### Example: Dual-Redundant Control System

* The Dual-Redundant Control System also called as active–standby system consists of two physically separated controllers: **Side1 and Side2**.
* Each controller periodically receives sensor data from a sensor subsystem.
* Only one controller should operate as the active controller and deliver the control output, while the other controller operates in the standby mode.
* Users can send commands to these controllers to flip the active/standby modes of these controllers.
* In this example, both controllers synchronize the computations so that they agree on which controller is active.

###### Application of PALS AADL Specification and Analysis

* In this example, the two sides communicate via a real-time network archi- tecture.
* To illustrate the motivation of the PALS AADL specification and corresponding analysis, consider a hypothetical requirement change that upgrades the network architecture.
* The linear shared-bus architecture is replaced by a switched networked architecture.
* Figure gives a sub-set of the system model during this configuration change. In the old configuration, the active/standby status of Side1 (side1\_status) flows through the shared bus to Side2.
* In the new configuration, this message flows through a number of networked switches to the destination node.
* In this example system, the status message is critical for the correctness of the coordination of these controllers.
* For example, if the standby controller does not receive the status message for the active controller, it assumes that the active controller has failed and, therefore, becomes active itself.
* If any inconsistency arises within the implementation, both controllers may incorrectly act as the active or standby controller during the same period.
* The PALS AADL specification requires the annotation of a number of AADL properties, such as Latency, Clock\_Jitter, and PALS\_ properties::PALS\_Output\_Time. Table 8.1 gives some sample values for these properties in these two configurations.

Side1 sensor data

User command (e.g., switch active)

Side2 sensor data

Controller status

Side2

Side1

Side1 control output Side2 control output

**Figure:** *Active–standby system*

* A static analyzer based on the PALS AADL specification

can easily detect that the new configuration is not a valid

PALS system. In particular, the PALS causality constraint

may be violated.

* For example, Side1’s status might be incorrectly delivered

in the same period in which the message is transmitted as

shown in Figure.

Side 2

side1\_ status

bus

bus

Side 1

Side 2

bus

switch

switch

bus side1\_status

Side 1

* + - * 1. Old

configuration

* + - * 1. New

configuration

**Figure :** *System model during configuration changes*

**Table :** *Sample Values for the AADL Properties*

|  |  |  |
| --- | --- | --- |
| AADL Property | Old Configuration | New Configuration |
| **Latency (from Side1 to Side2)** | 0.1–0.5 ms | 0.05–0.9 ms |
| **Clock\_Jitter** | 0.5 ms | 1 ms |
| **PALS\_properties::PALS\_Output\_Time** | 1–2 ms | 1–2 ms |

A static analyzer based on the PALS AADL specification can easily detect that the new configuration is not a valid PALS system.

In particular, the PALS causality constraint may be violated. For example, Side1’s status might be incorrectly delivered in the same period in which the message is transmitted as shown in figure below.

###### C++ API

PALSware provides an easy-to-use and extendable C++ API for the PALS applications. It defines the task execution and communication classes based on the operating system–independent interfaces. The main classes of this middleware are as follows:

* + - PALS\_task: This class is used for the execution of a task participat- ing in the PALS system. During each period, the PALS event gen- erator logic of PALS\_task invokes a placeholder function called each\_pals\_period. An application developer extends this class and defines the periodic logic in this function.

*c*1 (*t*1) = *jT*

1 ms *µ* = 0.8 ms

side 1\_ status

(*j* 1)*T*

Time

2 = 2ms

*c*2 (*t*4) =

*c*2 (*t*2) = *jT*

Side 1

Side 2

*c*1 (*t*3) = (*j* 1)*T*

*t*1 *t*2 *t*3 *t*4

**Figure 8.7:** *Violation of the PALS causality constraint*

* + PALS\_comm\_client: This class provides simple wrapper interfaces for logically synchronous message communications. Application tasks use two functions, pals\_send(connection name, payload) and pals\_recv(connection name, payload), to send and receive messages, respectively. In both functions, connection\_name is the input identifier for a connection from a source task to a group of destination tasks.

The following code snippet uses these classes:

Bool Side1\_task::each\_pals\_period() {

…

// 1. Define task-specific default values for messages. int8\_t side1 = NO\_MSG;

int8\_t side2 = NO\_MSG; bool user\_cmd = false;

// 2. Receive previous period's data, given that the source has not

// crashed.

**comm\_client->pals\_recv("side1\_status",&side1,1)**; **comm\_client->pals\_recv("side2\_status",&side2,1)**; **comm\_client->pals\_recv("user\_cmd",&user\_cmd,1)**;

// 3. Decide which side is active, based on information received

// from Side1, Side2, and the user.

next\_side1\_state = **active\_standy\_logic**(side1, side2, user\_cmd);

// 4. Send current period's state.

**comm\_client->pals\_send("side1\_status",&next\_side1\_state,1)**;

…

return true;

}

Here, the class Side1\_task extends PALS\_task and implements the periodic control computations of Side1 in the function each\_pals\_ period. In each period, Side1\_task reads the controllers’ statuses and the user command by using the pals\_recv function and sends its own active/standby status by using the pals\_send function. In the pals\_recv function, if the source task fails, the default input value is used.

These simple interfaces abstract away the details of the interactions with reliable communication services and management of logical syn- chronization groups.

###### Practical Considerations

A brief summary of the existing technologies available for the PALS system development are Listed.

###### AADL Tools for PALS System Modeling

Several tools are available for the PALS system modeling and analysis in AADL and Real-Time Maude:

* + - *SynchAADL2Maude:* Developed by [Bae12], this tool automatically translates the Synchronous AADL model of the PALS system to a Real-Time Maude model. In Real-Time Maude, the synchronous design can be verified by model checking.
    - *Static checker:* We have developed a static checker in our lab to vali- date the correctness of the pattern implementation with multirate computations. This tool also translates the single-rate PALS model to the Synchronous AADL model. This allows users to refactor the legacy components with a logically synchronous design directly in the physically asynchronous model. Users can generate the Syn- chronous AADL model and verify it.
    - *Rockwell Collins META Toolset:* This AADL tool also supports the PALS system analysis [Cofer11]. In general, this toolset supports design transformations, compositional verification, and static analysis for various architectural patterns. With respect to the PALS system, designers can instantiate the PALS design specifi- cation and validate the assumptions in a static checker, called Lute.
    - *ASIIST:* This tool is used to perform schedulability and bus analysis of AADL models [Nam09]. ASIIST complements the static check- ers for the PALS system. Designers can use ASIIST to compute

end-to-end delays, worst-case response, and output time in a system model with hardware components. The timing constraints of the PALS system can be validated once these timing properties are known.

###### PALS Middleware

The distributed middleware called PALSware is available for PALS system development in C++. It allows the use of COTS computers and networks, thereby lowering the synchronization overhead by poten- tially one to two orders of magnitude relative to custom or semi-custom legacy systems. The middleware has the following features:

* + A simple C++ library that is extendable in different system archi- tectures
  + A fault-tolerant real-time multicast protocol
  + Support for atomic distributed interactions
  + Integration library for clock synchronization algorithms

Recently, a C version of PALSware has been developed to support formal verification at the source code level using CBMC [CBMC]. In addition, several commonly used fault-tolerant applications that can take advantage of PALS are checked at the source code level [Nam14].

## [Real-Time Scheduling for Cyber-Physical Systems](#_bookmark4)

#### [Introduction and Motivation](#_bookmark4)

* CPS are composed of software and physical processes that must be synchronized.
* For instance, the airbag of a car must begin to inflate when a crash is detected, and it must be completely inflated before the driver gets too close to the steering wheel.
* In this example, the physical process is the driver moving toward the steering wheel during a crash as well as the airbag inflating.
* The software in the system is in charge of detecting the crash and triggering and synchronizing the inflation of the airbag so as to intercept the driver’s movement at the right time.
* As CPS increase in complexity and need to interact with more uncertain physical processes, the periodic sampling abstraction breaks.
* Another aspect of CPS that needs to be considered for the timing verification is the uncertainty in the environment. For instance, the execution time of a collision avoidance algorithm in an autonomous vehicle depends on the number of objects in its field of vision.

#### [Basic Techniques](#_bookmark4)

The focus is on the basic techniques developed for single-core processors with fixed timing parameters.

##### [Scheduling with Fixed Timing Parameters](#_bookmark4)

* Real-time scheduling is concerned with the timing verification of the interactions between the software and the physical processes of a CPS.
* At that time, the software that interacted with the physical processes ran on simple hardware, and it was possible to fully characterize it with fixed timing parameters such as periods, deadlines, and worst-case execution time (WCET).
* Such systems are called *real-time systems*.
* Real-time systems are designed to guarantee the timing of their

interactions with the physical world.

* The study of real-time systems includes the **operating system scheduling algorithms** that decide the execution order of threads (e.g., based on their priorities), the **timing analysis algorithms** that verify the execution of these threads is completed at the required time (e.g., 20 ms for the front airbag of a car), and **techniques to obtain the worst-case execution time** of the programs run by these threads, which is assumed to be known by the timing analysis algorithms.

###### Determining the Worst-Case Execution Time

* Two main approaches have been taken to obtain the WCET of a program—**measurements** and **static analysis**.
* Over the years, directly measuring the WCET has been the preferred approach for most practtioners.
* This approach relies on experiments designed to generate the maximum execution time in the programs.
* These experiments are run a large number of times to measure the WCET, and then the WCET is inflated to account for uncertainty of the measurements.
* This challenge of calculating WCET has spawned new research efforts focused on WCET based on analysis of the **code** of the program, its **access** to memory, and the **characteristics** of the hardware architecture on which such programs run.
* One program can also influence the memory behavior of another program. An example is the so-called cache-related preemption delay (CRPD);
* CRPD is where a program suffers from longer execution because some of its cache blocks are evicted because the program was preempted by another, higher-priority program.

###### Representation and Formalisms

* To describe the model of a system (also known as formalism),
* Consider the inverted pendulum presented in Figure.
* The control software of the inverted pendulum is designed to move a cart back and forth as needed to keep the pendulum as vertical as possible.
* This sensing and moving sequence must be executed periodically in a loop, known as a **control loop**.
* The time between one execution of the loop and the next one—known as the **period** of the control loop.



**Figure :** *Inverted pendulum in a control loop*

* Model used this period to develop a periodic task model in which each task has a fixed period and a **worst-case execution time**.
* A **task** is considered to execute periodically; each of these periodic executions is known as a ***job***.
* A task can be seen as an infinite number of jobs that arrive periodically.
* A **job** of a task is then considered to execute correctly if it finished before the arrival of the next job.
* This next-job arrival point is known as the ***deadline***of a job.
* We say that a task is schedulable if all its (infinite number of) jobs execute correctly. A task is then characterized as follows:

*i*  (*Ci* ,*Ti* , *Di* )

* + - Here ***Ci***represents the **WCET** of the **task** represented by ***i***, ***Ti*** repre- sents its **period**, and ***Di***represents the **deadline** of a job of task *i* rela- tive to the arrival of this job.
    - **The deadline is assumed to be equal to the period**;
    - The proportion of the processor that a task uses over time is known as its **utilization** and is calculated as follows:
* *U*  *Ci*

*i*

*Ti*

* + - It is defined a real-time system as a set of periodic tasks that are executed in a processor under a specific scheduling policy.
    - The set of tasks is deemed schedulable if all the tasks are schedulable, as expected.
    - This is denoted as  {1 , 2 ,, *n* }.
    - The total utilization of the set of tasks (also known as the taskset) is then calculated as follows:

*U*   *Ci*

 *i*  *Ti*

* In fixed-priority scheduling, tasks are given a priority and the scheduler always runs the task that is ready to run;
* The job of a task is executed to completion if no other job from a higher-priority task arrives.
* The periodic task model is nowadays successfully used in practice to verify the schedulability of tasksets.
* Table 9.1 presents a subset of the tasks in this system.
* In fixed-priority assignment, priorities are assigned to task at design time and all their jobs inherit the same priority.
* In dynamic-priority assignment, the priorities are assigned to the jobs when they arrive (at runtime). Different jobs from the same task are allowed to have different priorities.

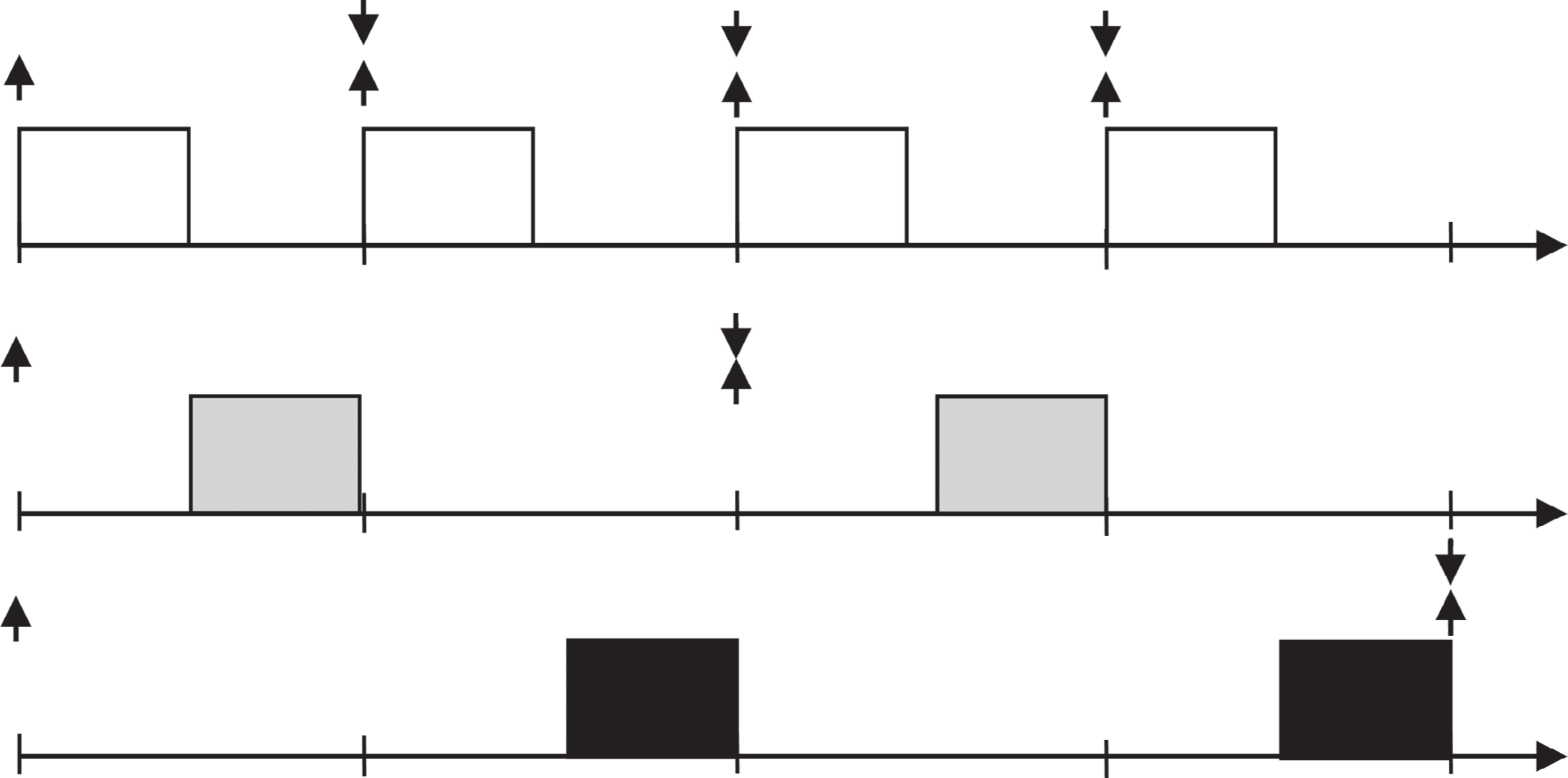
###### Fixed-Priority Assignment

* + In RM-based (rate-monotonic based) assignment, tasks with higher rates (or smaller periods) are given higher priority than those with lower rates.
  + The combination of RM priority assignment with fixed-priority scheduling is known as rate-monotonic scheduling (RMS).
  + The execution of three periodic tasks scheduled under RMS when they arrive together at time 0 is depicted in Figure.
  + In the figure, note that no two tasks can be executing (indicated by a rectangle) at the same time.
  + However, we canobserve that 1 starts executing at time 0 and both 2 and 3 delay their starts because 1 is still excuting.
  + Similarly, once 2 starts executing at time 5, the start of 3 is delayed further while this task waits for 2 to finish.
  + Moreover, once 2 finishes at time 10, a second job from 1 arrives at time 10 and starts executing, which keeps 3 waiting even longer.
  + When 3 is finally able to run at time 15, it is preempted by one job from
  + 1 at time 20, another job from 2 at time 25 and yet another job from 1
  + again at time 30.
  + After the last interruption, 3 is able to finish its remain- ing five units of execution from time 35 to 40.

**Table 9.1:** *Sample Tasks from a Generic Avionics System*

|  |  |  |  |
| --- | --- | --- | --- |
| **Task** | **Description** | **Period (***T* **)** | **WCET (***C***)** |
| Aircraft flight data | Determine estimates of position, velocity, altitude, and so on | 55 ms | 8 ms |
| Steering | Compute steering cues for cockpit display | 80 ms | 6 ms |
| Radar control (ground search mode) | Detect and identify ground objects | 80 ms | 2 ms |
| Target designation | Designate a tracked object as a target | 40 ms | 1 ms |
| Target tracking | Track the target | 40 ms | 4 ms |
| Heads-up display (HUD) | Display naviga- tion and target data | 52 ms | 6 ms |
| Multipurpose display (MPD) HUD display | Back up the HUD display | 52 ms | 8 ms |
| MPD tactical display | Display radar contacts and target | 52 ms | 8 ms |

*t*1  (*C*1  5, *T*1  10)



5

5

5

5

0 10 20 30 40

5

5

0

10

20

30

40

5

5

*t*2  (*C*2  5, *T*2  20)

*t*3  (*C*3  10, *T*3  40)

0

10 20 30 40

time

**Figure 9.2:** *Sample execution of three periodic tasks under RMS*

* With fixed-priority assignment, there are three main forms to test for schedulability:
* An **absolute bou**nd on processor utilization, a **parameterized bound** that depends on the number of scheduled tasks, and a **response-time** test.
* The absolute bound states that a taskset  scheduled under RMS is schedulable if ***U*  ln 2.**
* The parameterized bound, in contrast, states that a taskset of *n* tasks is schedulable if the following is true:

1

***U*  *n*(2*n*  1)**

* Finally, response-time test to determine the worst-case finish time of a task.:

**Please see the notes regarding Equation**

* This equation, the indices of the tasks are assumed to represent the priority precedence—that is, the smaller the index, the higher the prior- ity.
* The RMS priority assignment was later generalized to allow tasks that have deadlines that are shorter than their periods.
* This type of assignment is known as deadline-monotonic priority.
* In this case, tasks with shorter deadlines receive higher priorities. The response-time test, however, can still be used by considering the corresponding priorities.
* Similarly, the strictly periodic nature of a task *i* with a period *Ti*, which requires that the interval between two jobs’ arrivals be exactly *Ti*, was relaxed to allow this interval to be equal to or larger than *Ti*. This type of task is known as sporadic with a minimum inter-arrival*Ti*, and the results from RMS and DMS still apply.

###### Dynamic-Priority Assignment

* + - Figure below presents the scheduling of a taskset whose total utilization is 100% and that is schedulable according to the **response-time test** but is not schedulable according to either the **absolute bound or the parameterized bound**.
    - This scenario is depicted in Figure, where the second job of 1 pre- empts the first job of 2 even though the former has a deadline of 200 and the latter has a deadline of 141. We identify this as a suboptimal priority assignment.
    - The suboptimal job priority assignment led to the development of a job-based priority assignment known as **earliest-deadline first** **(EDF)** assignment.
    - The priority of each job is assigned individually at runtime based on its deadline.
    - Specifically, priorities are assigned at the arrival of a job, thereby ensuring that jobs with the earliest deadlines are assigned higher priorities than jobs with later deadlines.
    - For the case of the taskset of the jobs in Figure 9.3, the priority of the second job of task 1 is assigned a lower priority than the first job of 2 , thereby avoiding a deadline miss.
    - This scenario is depicted in Figure.
    - Given that EDF is able to utilize 100% of the processor, its schedulabil- ity test is much simpler.
    - Specifically, it suffices to ensure that the utilization of the taskset does not exceed 100% to ensure that it is schedulable under EDF (for tasksets with implicit deadlines).
    - It is proved that RM is an optimal task fixed-priority assignment algorithm and that EDF is an optimal dynamic priority assignment algo- rithm.
    - Thus, if a priority assignment for a taskset exists to make it sched- ulable, then the optimal priority assignment algorithm (e.g., RM) will also be able to produce a priority assignment to make it schedulable.

###### Synchronization

* The task model here assumes that tasks do not delay the execution of other tasks.
* When the synchronization of tasks takes the form of a mutually

exclusive access, it is reflected as a delay that a task  *j*, which is currently accessing the resource (or holding the resource), imposes on another task *i*, which is waiting for the resource.

* This situation is known as a **priority inversion** when *j* has a lower priority than *i*, and the waiting time is known as **blocking**.

*t* 1  (*C*1

4

4

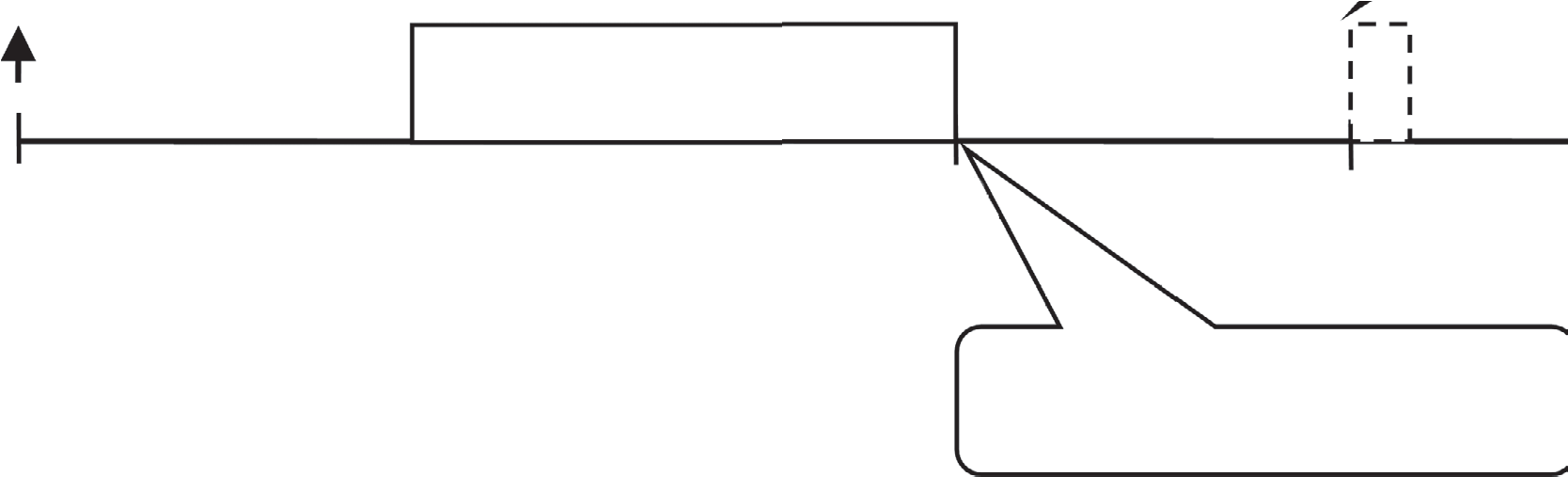
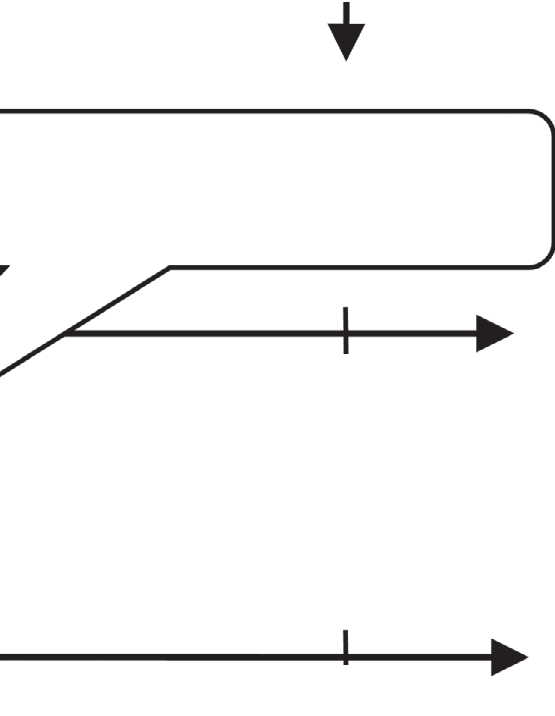
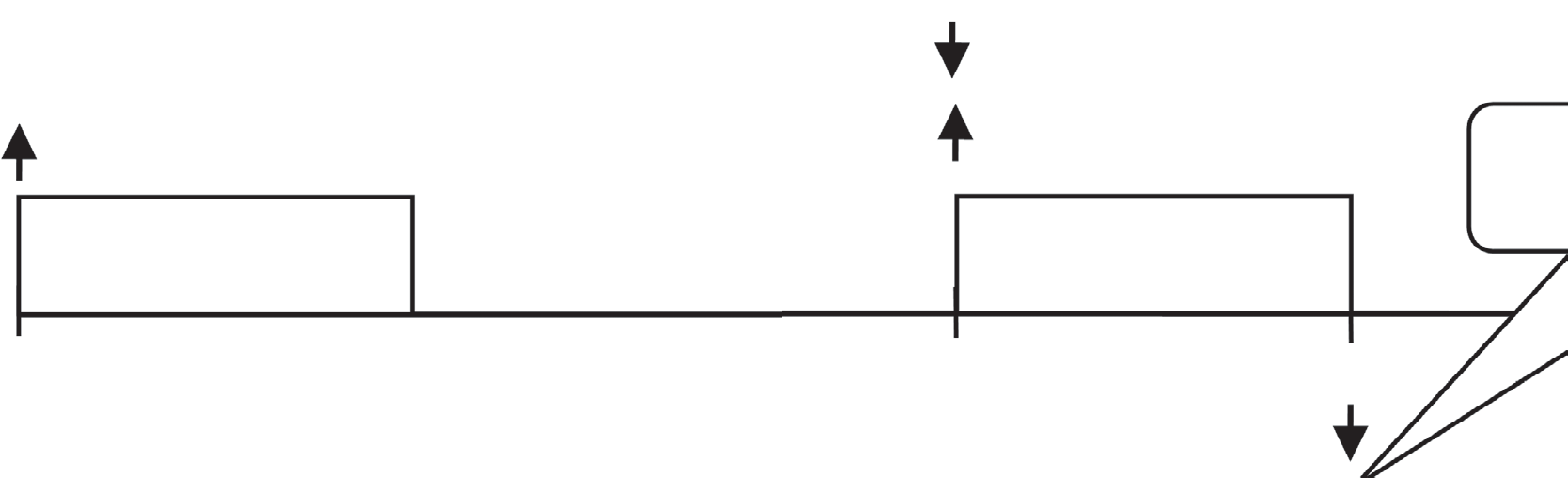
Deadline miss



41,



100)



41, *T*1 100)

Deadline miss

41

41

0

60, *T*2

100

41

200

141)

59

0

100

1

41

200

Preempted by

1

*t* 2  (*C*2

14

14

100

200

200

59

100

0

0



60,

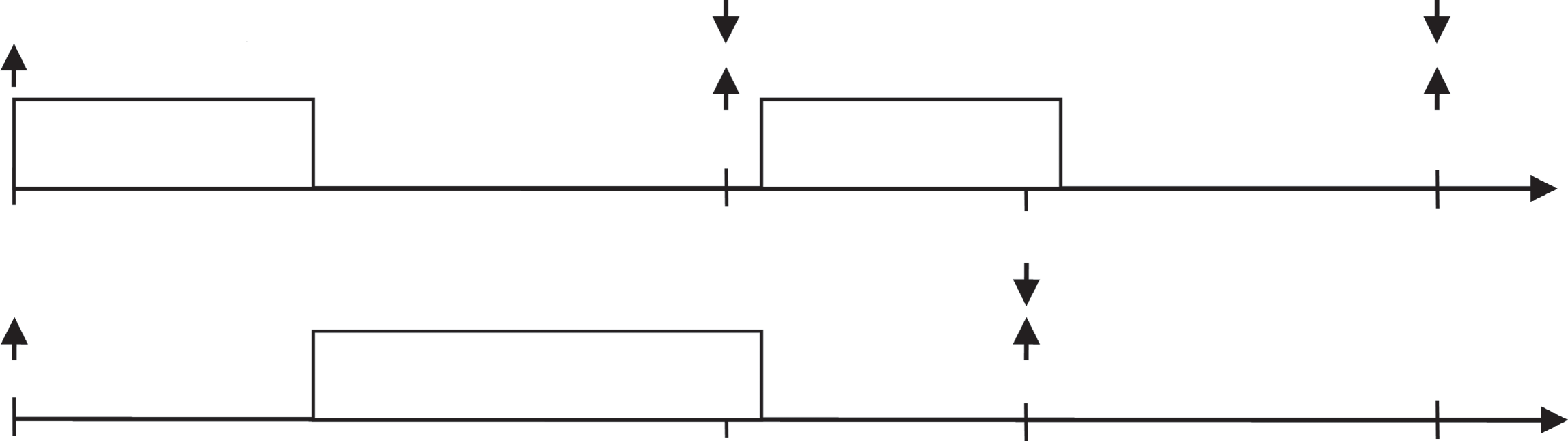


141)

**Figure :** *Suboptimal job priority assignment in RMS*

Preempted by

*t*

*t*1  (*C*1  41, *T*1  100)

41 41

*t*0  (*C*

 60, *T*

 141)

100

141 200

2 2 2

0

**Figure:***EDF assignment*

60

100

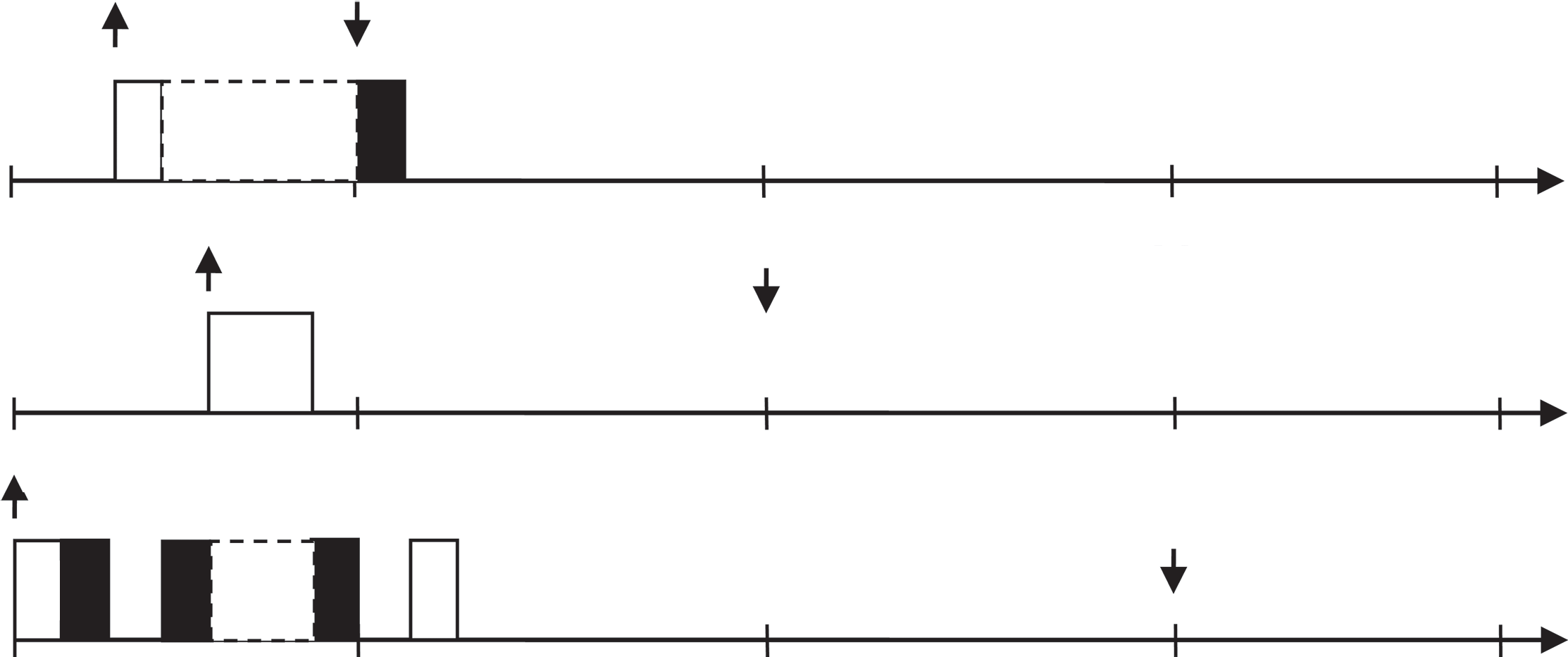
141 200

* + This situation is depicted in Figure below with a new parameter *CSi*, which captures the time a task spends holding the shared resource in a code section known as the critical section (depicted as a black segment in the figure).
    - A synchronization protocols are developed that prevents medium-priority tasks from increasing the priority-inversion blocking time with a technique known as **priority inheritance**.
    - In these protocols, when a task *i* tries to access a shared resource that is being used by task  *j* , then if the priority of *i* is higher than the priority of  *j*,  *j* inherits *i* ’s priority. This approach prevents medium-priority tasks from preempting *i*.
    - Two priority-inheritance protocols were developed
    - **Basic priority-inheritance** protocol (PIP) and the **Priority-ceiling protocol (PCP)**.
    - In PIP, the inherited priority is determined dynamically when a task is blocked while waiting for a resource.
    - In PCP, a priority ceiling is defined for a resource beforehand, as the highest priority among the potential users of the resource.
    - Under this scheme, when a task locks a resource, it inherits the priority ceiling of the resource.
    - Beyond its control over priority inversion, PCP has the added benefit of preventing deadlocks. This is possible because when a task *i* locks a resource *Su*, the inherited priority prevents any other task  *j* that can access *Su* from preempting it. As a consequence,  *j* cannot lock another resource *Sv* that might potentially be requested by *i* before acquiring *Su*, which would create a circular waiting situation.
    - A bounded blocking term *Bi* for each task *i* is calculated according

to the inheritance protocol used.

* + - For PCP, for each set of resources that are locked in a nested fashion, *Bi* is calculated as the maximum blocking among all the lower-priority tasks that can block *i* due to a resource in the set (i.e., only one task can block *i* ).
    - For PIP, it is necessary to add the blockings of all tasks. In addition, in PIP, it is necessary to consider tasks *k* that may block  *j*, which in turn may be blocking *i* . This scheme is known as **transitive blocking**.
    - Similarly, when a task *i* inherits a priority, it blocks medium-priority tasks, which experience this effect as a blocking from lower-priority tasks—a scheme known as **push-through blocking.**
    - All of these effects must be considered when calculating the blocking term of a task.

1)



*ti* = (*Ci* = 2, *Ti* = 7, *CSi* =

1

1 3

1

10

20

30

4

5

20

*tk* = (*Ck* = 4, *Tk* = 15)

30

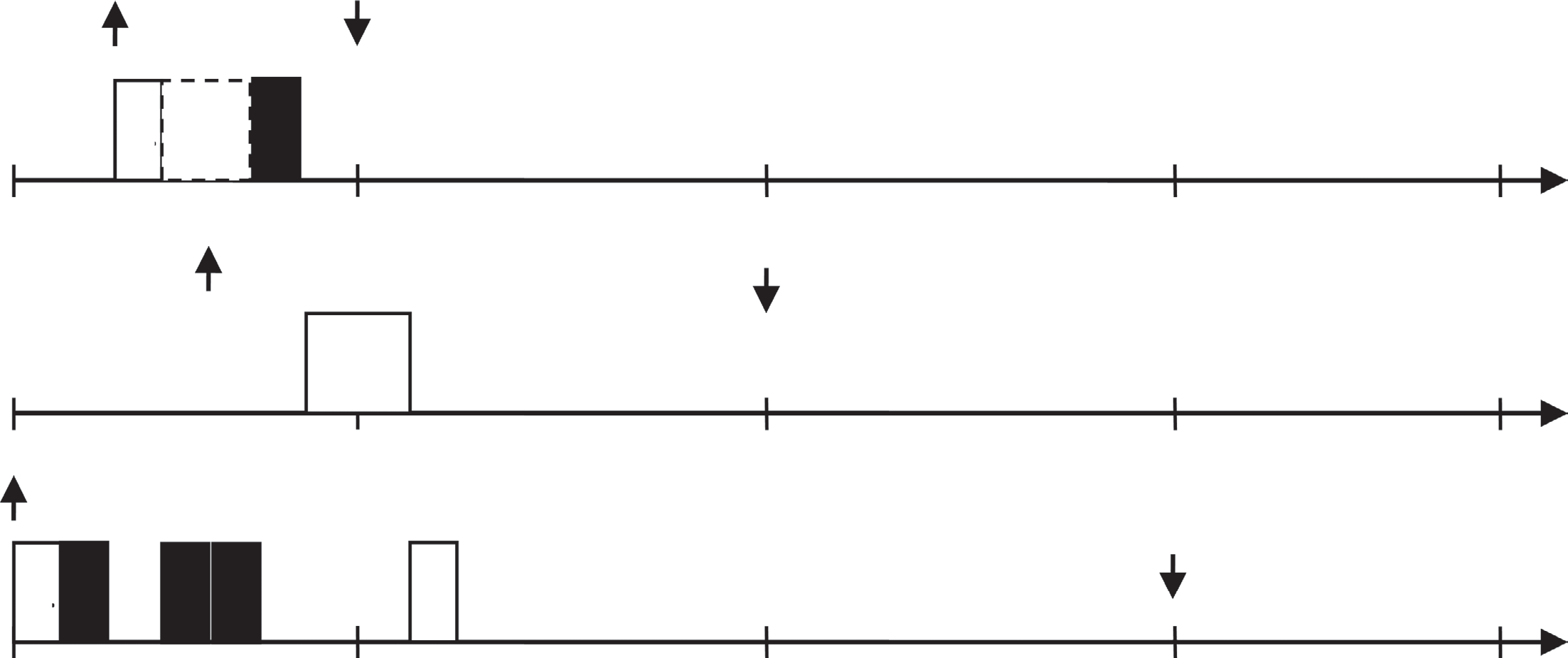
*tj* = (*Cj* = 5, *Tj* = 30, *CSj* =

1 1 1 1 1

3)

20 30

**Figure 9.5:** *Unbounded priority inversion*



*ti* = (*Ci* = 2, *Ti* = 7, *CSi* = 1)

1 1

1 3

10

20

30

*tk* = (*Ck* = 4, *Tk* = 15)

4

5

20

30

*tj* = (*Cj* = 5,*Tj* = 30, *CSj* = 3)

1 1 1 1 1

20 30

**Figure 9.6:** *Bounded priority inversion with priority inheritance*

**Memory Effects**

* The computer systems have complex memory systems.
* As a consequence, the execution time of a task depends on the schedule.
  + The data items stored in the cache when a job resumes execution may differ from the data items stored in the cache when the same job was preempted, because some of these data items may have been evicted by the preempting task(s).
  + Even when jobs are not preempted, they may share data. This situation can make the execution time of one job dependent on the job that executed before it.
  + Also, even for a single processor, there can be multiple requestors to the memory system. For example, a job *J* might instruct an I/O device to perform direct memory access (DMA), so that when job *J* finishes and another job *J’* starts to execute, the DMA operation continues dur- ing the execution of job *J’* .
  + Such a situation can cause the execution time of a job to increase and a DMA I/O operation to take longer to complete. Methods exist for analyzing these effects.
  + The method in can be used to find the execution time of a task considering DMA I/O devices operating in cycle-stealing mode.
  + In addition, the execution of a task can affect the amount of time it takes for a DMA operation to complete.

#### [Advanced Techniques](#_bookmark4)

The real-time scientific community is currently creating solutions to a number of issues stemming from advanced features in CPS application and hardware platforms.

##### [Multiprocessor/Multicore Scheduling](#_bookmark4)

* When a taskset  is run on a computer with multiple processors, we consider this as a multiprocessor scheduling problem.
* Two basic forms of scheduling exist for multiprocessor scheduling: **partitioned** and **global** scheduling (Figure below).
* In **partition scheduling**, the taskset is divided into subsets, with each subset being assigned to one processor and a uniprocessor scheduling scheme being used to schedule it.
* In **global scheduling**, all the tasks are scheduled on the set of *m* processors by selecting which subset of *m* tasks should run on the *m* processors at any given time.
* For instance, global EDF selects the *m* jobs with the shortest deadlines. The multicore processors are a special type of multiprocessor where cores share common hardware resources that will also influence scheduling.
* In Figure, the right side of the figure illustrates partitioned scheduling: Tasks 1 and task 2 are assigned to processor 1, and task 3 is assigned to processor 2.

Global Partitioned

*t* 1 *t* 2 *t* 3

Processor 2

Processor 1

*t* 1

*t* 3

*t* 2

Processor 2

Processor 1

**Figure 9.7:** *Tasksets in partitioned and global scheduling*

###### Global Scheduling

* Global scheduling is a class of scheduling algorithms in which proces- sors share a single queue storing jobs that are eligible for execution;
* On each processor, a dispatcher selects the highest-priority job from this queue.
* Let *eligjobs*(*t*) denote the set of jobs that are eligible at time *t*.
* Since there are *m* processors, at each instant, the *min*(|*eligjobs*(*t*)|, *m*) highest-priority eligible jobs are selected to execute on the processors.

###### Task-Static Priority Scheduling

* In a task-static priority scheduling algorithm, each task is assigned a priority and each job of this task has the priority of the task.

**Table 9.2:** *Example Implicit-Deadline Sporadic Tasksets*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Taskset 1** |  |  | **Taskset 2** |  |  | **Taskset 3** |  |
| *n* = *m* + 1 |  |  | *n* = *m* + 1 |  |  | *n* = *m* + 1 |  |
| *T*1 = 1 | *C*1 = 2ε |  | *T*1 = 1 | *C*1 = ½ + ε |  | *T*1 = 1 | *C*1 = ½ + ε |
| *T*2 = 1 | *C*2 = 2ε |  | *T*2 = 1 | *C*2 = ½ + ε |  | *T*2 = 1 | *C*2 = ½ + ε |
| *T*3 = 1 | *C*3 = 2ε |  | *T*3 = 1 | *C*3 = ½ + ε |  | *T*3 = 1 | *C*3 = 2ε |
| … | … |  | … | … |  | … | … |
| *Tm* = 1 | *Cm* = 2ε |  | *Tm* = 1 | *Cm* = ½ + ε |  | *Tm* = 1 | *Cm* = 2ε |
| *Tm+*1 = 1 + ε | *Cm+*1 = 1 |  | *Tm+*1 = 1 + ε | *Cm+*1 = ½ + ε |  | *Tm+*1 = 1 | *Cm+*1 = 2ε |

* Consider taskset 1 in Table above to be scheduled on *m* processors using global scheduling with RM and assume that 2 \* *m*  1. The priority assignment RM assigns the highest priority to the tasks 1 , 2 ,, *m* and the lowest priority to task *m*1.
* Let us consider the situation where all tasks generate a job simulta-
* neously, and let time 0 denote this instant (Figure below).
* During the time interval [0, 2), the jobs of tasks 1 , 2 , , *m* are the highest-priority eligible jobs, so they execute, one job on each processor, and finish at time 2.
* During the time interval [2, 1), a single job of task *m* 1 executes on one of the processors (the other *m*  1 processors are idle).
* Note that at time 1, the job of task *m*1 has executed 1  2 time units and its execution time is 1, so the job of task *m*1 has not yet finished execution at time 1.
* Then, at time 1, each of the tasks 1 , 2 ,, *m* generates a new job.
* Hence, during the time interval [1, 1  2), the jobs of tasks
* 1 , 2 , , *m* are the highest-priority eligible jobs, so they execute, one job on each processor, and finish at time 1  2.
* Note that the job of *m*1 had its deadline at time 1  .
* From the arrival time until its deadline, however, this job executed only 1  2 time units, even though its exe- cution time is 1 time unit.
* Thus, this job missed its deadline.
* We can repeat this argument for any  and for any *m*.

Letting  0 and *m*   gives us the case in which there exists a taskset for which:

1 \**u*  0

*i*

*m* *i* 

*Processor 2*

time

time

*Processor 1*

*t*2

*t*2

*t*1

*t*1

|  |  |  |  |
| --- | --- | --- | --- |
| *t*m | *tm*  1 | *tm* |  |

**Figure 9.8:** *The utilization bound of global RM*

*Processor m*

time

* + This expression indicates that a deadline is missed when the taskset is scheduled by global RM.
  + In turn, the utilization bound of global RM is zero.
  + This outcome is referred to as **Dhall’s effect.**
  + We can also see that changing the priority assignment makes the taskset schedulable. For example, assigning the highest priority to task
  + *m* 1 and assigning a lower priority to all the tasks 1 , 2 , , *m* makes the taskset schedulable.
  + Hence, RM is not the optimal priority assignment scheme for global scheduling.

* + Recognizing these problems, the scientific community has developed priority assignment schemes that avoid this issue and, therefore, perform better than RM.
  + We can categorize a task *i* as “heavy” or “light,” where a task is con- sidered heavy if the following is true:

*m*



*ui* 3*m*  2

* Otherwise, it is light. We then assign the highest priority to the heavy tasks, assign a lower priority to the light tasks, and assign the relative priority order between the light tasks according to RM.
* Indeed, the priority assignment scheme RM-US(*m*/(3*m* − 2)) does exactly that; for *m* ≥ 2, its utilization bound has been proved to be

*m*/ (3*m* − 2).

* + This utilization bound is greater than **⅓,** so the assignment scheme circumvents Dhall’s effect.
  + Further improvements have been made. The algorithm RM-US(*y*), where *y* is a solution to a nonlinear equation where *y* ≈ 0.374, operates similarly; its utilization bound is *y*  0.374.

SM-US  2 

 3  5 

* Its utilization bound follows:

2  0.382



5

3 

* One can show (using taskset 2 in Table 9.2) that for each job-static prior- ity scheduling algorithm, it holds that the utilization bound of the algo- rithm is at most 0.5.
* It is possible to use the POSIX call sched\_setscheduler to assign a priority to a task. In addition, one can set a processor affinity mask to allow a task to execute on any of the processors.

###### Job-Static Priority Scheduling

* + In a job-static priority scheduling algorithm, each job is assigned a priority, and this priority does not change with time.
  + Clearly, every task- static priority scheduling algorithm is also a job-static priority scheduling algorithm.
  + EDF is not the optimal priority assignment scheme for multiprocessors.  
    The utilization bound for the multiprocessor case is zero— this can be seen by applying the same reasoning as in the previous example.
  + The scientific community has developed a priority assignment scheme that avoids this problem and, therefore, performs better than EDF.
  + We can categorize a task *i* as “heavy” or “light” where a task is heavy if the following is true:

*m*



*ui* 2*m*  1

* Otherwise, it is light.
* We can then assign the highest priority to the heavy tasks, a lower pri- ority to the light tasks, and the relative priority order between the light tasks according to EDF.

 *m* 

EDF-US

 2*m*  1

* Its utilization bound is proven to be the following:

*m*

2*m*  1

* This utilization bound is greater than **½** and, therefore, this scheme cir- cumvents Dhall’s effect, too.
* One can show (using taskset 2 in Table 9.2) that for each task-static priority scheduling algorithm, it holds that the utilization bound of this algorithm is at most 0.5.
* With some extra effort, one can use the Linux kernel and schedule the tasks to behave according to the scheme as follows: Designate a special process that runs at the highest priority (use the POSIX call sched\_setscheduler to assign the priority).
* Whenever a task has finished the execution of its job, it should notify this special process that the task is waiting for a given event (e.g., a timer to expire).
* When the special process observes this event, it should unblock the task (e.g., a call signal on a semaphore where the task previously called is waiting on that semaphore) and assign the correct (according to the scheme) priority to the task that woke up; this can be done with sched\_setscheduler.
* Because global scheduling allows task migration, a job may poten- tially experience additional cache misses when it resumes execution on another processor than the one it was preempted on.
* This may prolong the execution time of a job—a factor that might need to be taken into account in the schedulability analysis.
* In either job-static or task-static global scheduling, a context switch can occur only when a job arrives or when a job finishes execution.
* Thus, for a given time interval, we can compute an upper bound on the number of possible context switches, and we can compute an upper bound on the number of migrations.
* If an upper bound on the overhead of a single migration is known, we can then compute an upper bound on the overhead for migrations— another factor that can be incorporated into the schedulability analysis.

###### Partitioned Scheduling

* In partitioned scheduling, the taskset is partitioned into tasks and each partition(task) is assigned to a processor;
* At runtime, a task is allowed to execute only on the processor to which it is assigned.

###### Task-Static Priority Scheduling

* + - In partitioned, task-static priority scheduling, before runtime a task is assigned to a processor and cannot migrate;
    - At runtime each processor is scheduled using a uniprocessor scheduling algorithm based on task-static priorities.
    - For a single processor, RM is the optimal task-static priority schemes; thus, we can, with no performance loss, use RM on each processor. We then need to discuss only the assignment of tasks to processors.
    - An intuitive way to assign tasks to processors would be to use load- balancing offline.
    - That is, we could ensure that tasks are considered one by one, and assign the currently considered task to the processor that currently has the lowest utilization.
    - Unfortunately, the utilization bound of such a load-balancing algorithm is zero, as we will see in the following example.
    - Consider the taskset 1 in Table 9.2 to be scheduled on *m* processors using load-balancing (i.e., tasks are considered one by one), and assign
    - the currently considered task to the processor that currently has the lowest utilization.
    - Also, assume that 2 \* *m*  1. Initially, no tasks have been assigned, so the utilization of each processor is zero.
    - Now we consider tasks one at a time, beginning with task 1.
    - It should be assigned to the processor that currently has the lowest utilization.
    - Since all processors have utilization zero, this task can be assigned to any processor.
    - Next, consider 2 . It can be assigned to any processor except the proces-sor to which 1 is assigned.
    - Continuing with tasks 3 , 4 , , *m*, we end up with an assignment where each task is assigned to its own processor.
    - Now, only one task, *m*1, remains. If *m*1 is assigned to processor 1, then the utilization of processor 1 would be as follows:

2  1

1 

* + - This value is strictly greater than 1; if *m*1 is assigned to processor 1, then, a deadline miss would occur on processor 1.
    - It turns out that the same reasoning applies to any of the other processors: If *m*1 is assigned there, a deadline miss would occur.
    - Regardless of where *m* 1 is assigned, a deadline miss will result. We can repeat this argument for any  and for any *m*.
    - Letting  0 and *m*   tells us that there exists a taskset for which we get the following expression:

1 \**u*  0

*i*

*m* *i* 

* A deadline is missed when the taskset is scheduled by this type of load- balancing.
* Hence the utilization bound of this type of load-balancing is zero.
* The zero utilization bound of the load-balancing scheme presented in this example behooves us to develop better algorithms for task assignment.
* We can observe that this type of load-balancing performs poorly due to three conditions:
  + There are more tasks than processors.
  + At least one task (we will call it the *large task* in this discussion) requires so much capacity that it cannot be assigned to the same processor with any other task (and still be schedulable).
  + This large task is assigned to a processor once each processor was already assigned at least one task.
    - One idea for designing a better task assignment scheme is to assign a task to a processor where tasks have already been assigned, so as to try to fill up the half-full processors first and avoid adding tasks to empty processors if possible.
    - This scheme has the potential to avoid the performance problem seen in this example.
    - Consider taskset 3 in Table. The task 1 can be assigned to any proces- sor. Then we consider task 2 . If we follow the idea of assigning a task to a processor where other tasks have already been assigned, then we should assign task 2 to the same processor to which 1 has been assigned. The utilization of that processor would then be as follows:

1  1 

2  2

1 1

* This value is strictly greater than 1, so a deadline miss would occur on this processor.
* Therefore, whenever we attempt to assign a task, we must perform a uniprocessor schedulability analysis.
* The scientific community has developed task assignment schemes based on these two ideas:
* Assign a task to a processor on which other tasks have been assigned, and use a uniprocessor schedulability test to ensure that after a task has been assigned, the tasbjkset on each processor is schedulable.
* Consider the task assignment algorithm known as first-fit for task assignment for a taskset (taskset-to-be-assigned) and a set of processors

(processor set).

* + Its pseudocode follows:
    - * 1. If taskset-to-be-assigned is not empty then

Fetch **next** task *i* from taskset-to-be-assigned

* + - * 1. Else

Return success

* + - * 1. Fetch first processor *pj* in the processor set
        2. If *i* is schedulable when assigned to processor *pj* then

Assign *i* to processor *pj*

Remove *i* from taskset-to-be-assigned

Jump to step 1

* + - * 1. Else if there is a next processor *pj* in the processor set then

Fetch the next *pj* processor from the processor set and jump to step 4

* + - * 1. Else

Return failure

* If the uniprocessor in the utilization-based schedulability test is used in step 4, then the overall algorithm has utilization bound
* 1  0.41



2

* A more advanced algorithm is also known that has the utilization bound 0.5 [Andersson03].
* One can show (using taskset 2 in Table 9.2) that for each partitioned task-static priority scheduling algorithm, it holds that the utilization bound of the algo- rithm is at most 0.5.
* Note that the Linux kernel supports partitioned fixed-priority sched- uling. It is possible to use the POSIX call sched\_setscheduler to assign a priority to a task, and one can also set a processor affinity mask to man- date that a task is allowed to execute only on a certain processor.

###### Job-Static Priority Scheduling

* For job-static priority scheduling on a single processor, we know that EDF is optimal.
* For designing a good task assignment scheme for job-static priority partitioned scheduling, we use schedulability tests for the uniprocessor scheduling algorithm (in our case, uniprocessor EDF).
* Fortunately, an exact schedulability test for the uniprocessor case is very simple:
* If the sum of utilization of all tasks on a processor does not exceed 1, then the taskset on this processor is schedulable.
* With this reasoning, a first-fit algorithm can be designed with utilization bound 0.5.

###### Algorithms Using Fairness

* + In some scheduling algorithms, jobs can migrate to any processor at any time; many of these algorithms have the utilization bound 100%.
  + A recent algorithm known as **RUN** generates many fewer preemptions than the other algorithms that rely on fairness.
  + With RUN, the number of preemptions per job is logarithmic as a function of the number of processors.
  + RUN has a drawback, however: It often generates scheduling segments that are very small and cannot be realized in practice.
  + RUN cannot schedule sporadic tasks—only periodic tasks.

###### Algorithms Using Task-Splitting

So far, we have seen **two classes of algorithms**: **algorithms with few preemptions and utilization bounds at most 0.5, and algorithms with a larger number of preemptions and utilization 1.**

* + It would be desirable to use a scheduling algorithm with a utilization bound greater than 0.5 but still have few preemptions.
  + **Task-splitting** is a category of algorithms that offers this capability.
    - With a taskset 2 in Table the partitioned scheduling algorithm, there is no way to assign tasks to processors so that after each task has been assigned to a processor, it holds that for each processor, the utilization is at most 1.
    - We can achieve this goal, however, if we allow a task to be “split” into two or more “pieces.” Consider taskset 2 in Table. We can assign 1 to processor 1, assign 2 to processor 2, and assign
    - *m* to processor *m*. Then we split task *m* 1 into two pieces as follows:

*Tm*+1’ = 1, *Cm*+1’ = ¼ + ε/2

*Tm*+1’’ = 1, *Cm*+1’’ = ¼ + ε/2

* + After this splitting, task *m*1’ can be assigned to processor 1 and task
  + *m*1’’ can be assigned to processor 2. After this assignment, for each processor, it holds that the utilization is at most ½ + ε + ¼ + ε/2. For ε  1/6, we obtain that after this assignment, for each processor, it holds
  + that the utilization is at most 1.
  + Hence, we can apply uniprocessor EDF on each processor and then all deadlines will be met.
  + This scheme will work if we can schedule the two pieces of *m*1 independently and we are allowed to execute the two pieces simultaneously.
  + The two pieces belong to an original task *m*1, which is a program with internal data dependencies.
  + Thus, unless we know the internal structure of the program that represents task *m*1, we must schedule task *m* 1 while assuming that the two pieces of *m* 1 must not execute simultaneously.
  + For scheduling sporadic tasks, the research literature offers different approaches to dispatch the pieces of a split task:
  + ***Slot-based split-task dispatching****:* With this approach, time is subdivided into time slots of equal size. In each time slot, a reserve is used for the execution of the split tasks. Hence, a task *i* that is split between processor *p* and processor *p* + 1 is assigned to the reserve on processor *p* and the reserve on processor *p* + 1. The phasing and duration of these reserves are selected such that for two consecutive processors, for those reserves that are used for a task that is split between these two processors, there is no overlap in time between these reserves.
  + *Suspension-based split-task dispatching:* In suspension-based split- task dispatching, the double-prime piece becomes non eligible when the single-prime piece of a task executes.
  + *Window-based split-task dispatching:* In window-based split-task dispatching, the double-prime piece becomes eligible only when the single-prime piece has finished execution. This ensures no overlap. The deadline of the respective pieces must be assigned so that the **sum of their deadlines does not exceed the deadline** of the task before it got split.

###### Memory Effects

* Contention on the bus from the memory controller to the mem- ory modules increases with the number of processors.
* Many multiprocessors are implemented on a single chip (multicore processors), in which case the processors (processor cores) typically share cache memories.
* Then, a cache block belonging to a task can obviously be evicted when tasks executing on the same processor preempt it.
* In a multicore processor, however, cache blocks belonging to a task can also be evicted by other tasks executing in parallel on other processors.
* The method known as ***cache coloring***eliminates this effect by setting up the virtual-to-physical address translation so that cache sets of different tasks do not intersect.
* The main memory of computers today uses dynamic random access memory (DRAM) that is organized as a set of banks, where a bank comprises multiple rows.
* In each bank, at each instant, there can be at most one row open. When a task performs a memory operation, it selects one memory bank (determined by the physical address) and then opens a row (determined by the physical address) in that memory bank and then reads data from this row.
* If the row corresponding to a given memory operation is already open, then the row does not need to be opened and the memory operation can complete faster.
* Often, programs have locality of reference such that, for many memory operations, there is no need to open a new row (because a previous memory operation has opened the row that a memory operation accesses).
* When multiple tasks execute on different processors and access different rows in the same bank, however, a task *i* that opens a new row closes the row opened by another task *j* and, therefore, increases the execution time of *j*.
* The method called ***bank coloring***(which works similarly to cache coloring) eliminates this effect.
* Since both cache coloring and bank coloring configure the virtual memory system for different purposes, it is necessary to coordinate this configuration if both cache coloring and bank coloring are to be achieved.
* Cache and bank coloring schemes have limitations due to the small number of partitions that may be created if full isolation is required.

##### [Accommodating Variability and Uncertainty](#_bookmark4)

To accommodate variations in the timing parameters of the taskset and uncertainty from the environment the following topics are given.

###### Resource Allocation Tradeoff Using Q-RAM

* + We consider tasks that can be configured to different quality-of-service levels, which in turn modifies the required resources.
  + This is the case, for instance, in a video playback application where, depending on the frames per second (FPS) required, the application consumes different amounts of CPU cycles.
  + In this scenario, the problem is shifted to the optimal selection of the quality-of-service (QoS) level.
  + To solve this problem, a Quality-of- Service Resource Allocation Model (Q-RAM) was developed.
  + Q-RAM maps QoS levels of the different applications to system utility (as perceived by the user) and allocates resources to these applications, ensuring that the total system utility is maximized (optimal).
  + Q-RAM primarily takes advantage of the fact that as applications (e.g., video streaming) increase their QoS level, the incremental utility to the user keeps decreasing, yielding diminishing returns.
  + In other words, the utility gain obtained by moving from a QoS level *i* to *i* + 1 is larger than that obtained by moving from level *i* + 1 to *i* + 2.
  + For instance, in a video streaming application, increasing the FPS from 15 to 20 gives the user more additional utility (i.e., perceived quality) than increasing from 20 to 25 frames per second.
  + Q-RAM uses the utility functions to perform an optimal allocation of resources to different tasks exploiting the **concave property of diminishing returns**, which manifests as a monotonically decreasing utility- to-resource ratio.
  + This ratio, which is known as **marginal utility**, is the derivative in a continuous utility function.
  + Q-RAM uses the marginal utility to perform the allocation one increment at a time, starting with the increment that derives the largest utility for the smallest allocation of resources (largest marginal utility).
  + In each of the subsequent steps, it selects the next largest marginal utility increment, continuing in this fashion until the entire resource (e.g., CPU utilization) has been allocated.
  + When Q-RAM reaches an exact (optimal) solution, the marginal utility (or derivative) of the last allocation (QoS level) of all the tasks is the same.
  + Figure below depicts two sample application utility curves.
  + In this case, if the total number of resource units is 6 (e.g., 6 Mbps), the step- by-step allocation would be as follows: Give 1 unit of resource to Application A, give 1 unit of resource to Application B, raise the allocation to Application A to 2 units, raise the allocation to Application B to 3 units, and raise the allocation to Application A to 3 units.
  + Applications can have multiple QoS dimensions that provide different utility to the user.
  + As a result, each application *i* has a dimensional resource utility

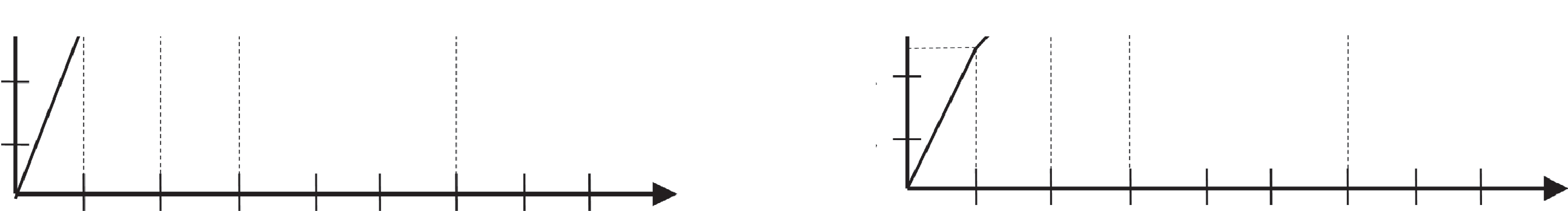
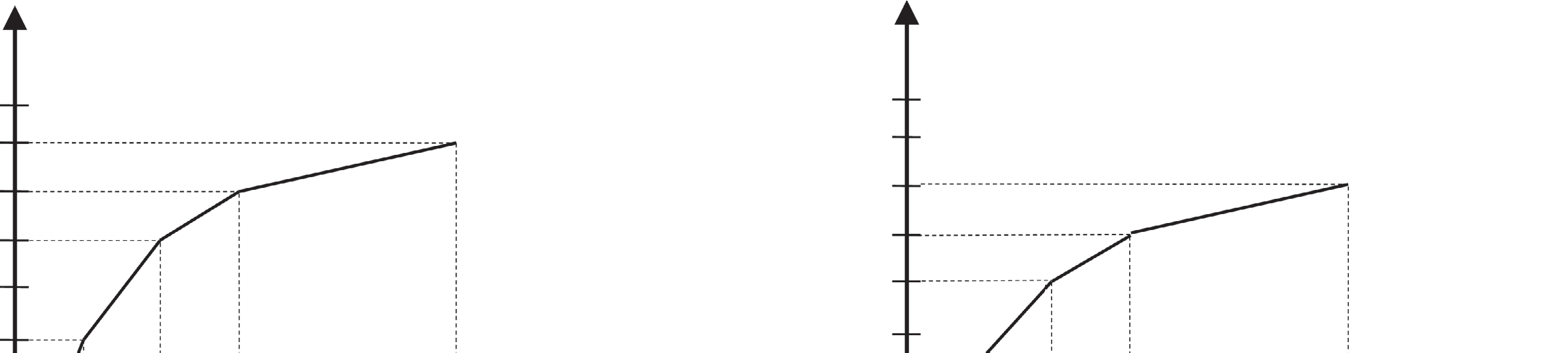
*Ui,k*  *Ui,k* (*R*) for an allocation of *R* resources for each of the

different QoS dimensions *Qk* 1  *k*  *d* of the application.

8

8

7



8

7

6

5

4

3

2

1

7

6

6

5

Utility

Utilit**y**

5

4

4

3

3

2

2

1

1 2 3 4 5 6 7 8 1 2 3 4 5 6 7 8

Resource

Resource

Utility Curve Application A Utility Curve Application B

**Figure 9.9:** *Sample utility curves*

.

* To reach a specific QoS level, the application may need more than one resource.
* For instance, in a video streaming application, it may need CPU cycles as well as network bandwidth if the streaming occurs across a network. In that case, the dimensional utility requires multiple
* resources: *Ui,k*  *Ui,k* (*Ri* ,1 , *Ri* ,2 ,, *Ri,m* ) .
* The resource allocation is then

considered an allocation vector of the form *Ri*  *R*

*i* ,1

, *Ri* ,2

,, *R*

*i* ,*m* .

* When the QoS dimensions of an application are independent, they generate their own utility, and the utility of the application as a whole

*d*

is the sum of the dimensions: *Ui*  *Ui* ,*k* .

*k*1

* In contrast, when the dimensions are dependent, their allocation needs to be considered as a combined allocation to reach a single utility level, as previously presented.
* The utility curve then becomes a surface, for the two-dimensional case, and the allocation steps are selected from the steepest trajectory on the surface with combined increments in both dimensions.
* Q-RAM allows the assignment of weights to the applications of a system to encode their relative importance. The utility of the system is

*n*

then calculated as *U*  *w U* (*R* ).

*i*

*i i*

*i*1

###### Mixed-Criticality Scheduling

* + Traditional real-time scheduling theory considers all tasks as being equally critical and focuses on ensuring that the deadlines of all tasks are met.
  + In reality, it is common to have real-time systems with different levels of criticality.
  + These variations can come from the uncertainty in the environment (e.g., larger number of objects to avoid) or from different degrees of pessimism required from certification practices for the worst-case execution time.
  + For these cases, a common practice is to separate low-criticality tasks from high-criticality tasks, and to deploy these categories of tasks in different processors to prevent a low-criticality task from interfering with high-criticality tasks, while anticipating variations in the worst-case execution time.
  + In recent years, this separation has been implemented through the operating system to support temporal protection.
  + This temporal protection is typically implemented as a symmetric temporal protection. That is, it prevents a low-criticality task from overrunning its CPU budget (*C*) and blocking high-criticality tasks from finishing before its deadline.
  + At the same time, it prevents high-criticality tasks from overrunning their CPU budget so as to allow low-criticality tasks to complete by their deadlines.
  + Unfortunately, in this latter case, a high-criticality task is delayed to allow a low-criticality task to run.
  + When this situation, which is known as a criticality inversion, makes a higher-criticality task miss its deadline, we say that a criticality violation has occurred.
  + A new scheduler called Zero- Slack Rate Monotonic (ZSRM) that provides temporal protection for mixed-criticality systems that do not suffer from criticality violations.
  + In particular, ZSRM prevents low-criticality tasks from interfering with higher-criticality tasks, but allows the latter to steal CPU cycles from the former.
  + This scheme is known as asymmetric temporal protection.
  + ZSRM allows jobs to start executing with their rate-monotonic priorities even in the presence of criticality inversion, in what is called a rate-monotonic execution mode.
  + A job then switches to a critical execution mode, stopping lower-criticality tasks at the last instant possible, so as to ensure that it will not miss its deadline (i.e., suffer a criticality violation).
  + This last instant, which is known as the zero-slack instant of task
  + *i* (*Zi*), is calculated using a modified version of the response-time test
  + that combines the two modes of execution.
  + Now, to accommodate variations in the worst-case execution time, tasks are given two execution budgets: the nominal execution budget
  + (*Ci*), which is considered the worst-case execution time when the task does not overrun, and the overloaded execution budget (*C*o), which is

*i*

* + the worst-case execution time when it overruns. As a result, tasks in ZSRM are defined as follows:

  (*C* , *Co* ,*T* , *D* , )

*i i i i i i*

* Here *i* represents the criticality of the task (following the convention that a lower number means higher criticality).
* With this task model, the response-time test is run for each task *i* with a timeline that starts with the rate-monotonic execution mode (RM mode) and ends with the critical execution mode (critical mode).
* In each of the modes, a different set of interfering tasks is considered: In RM mode, the set of all higher- priority tasks ( *rm*) is considered; in critical mode, only the set (*c*) of
* *i i*
* tasks with higher criticality (including the tasks with lower priority) is considered. *Zi* is calculated by first executing the whole job *Ji,k* from task *i* in critical mode;
* A sample response-time timeline and the *Zi* are shown in Figure below.
* ZSRM provides the following task-based guarantee:

*j i*

executes for more than *Cj* . This guarantee also provides a graceful deg- radation guarantee ensuring that if deadlines are missed, they are

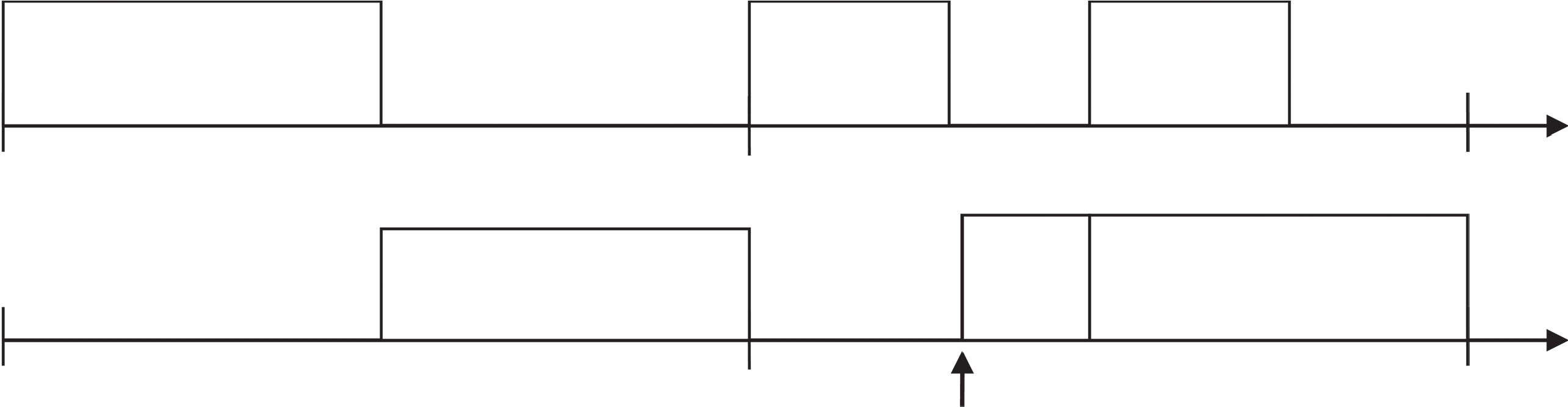
missed in reversed order of criticality.

4

0

8

2



2

1

1

4

8

2

1

2

2 **1**

2

*t*1 = (*C*1 = 2, *C o* = 2, *T*1 = 4, *D*1 = 4, *z*1 = 2) Z2

*t*2 = (*C*2 = 2.5, *Co* = 5, *T*2 = 8, *D*2 = 8, *z*2 = 1)

4

2

2

0

2

2

8

1

2

**Figure :** *ZSRM response-time timeline*

* Other models of mixed-criticality focus on different levels of assurance for the worst-case execution time as the source of their variation.
* In such models, tasks have different WCET for each of the criticality levels of the system and the schedulability of a task is evaluated based on its required level of criticality (which maps to a level of assurance).