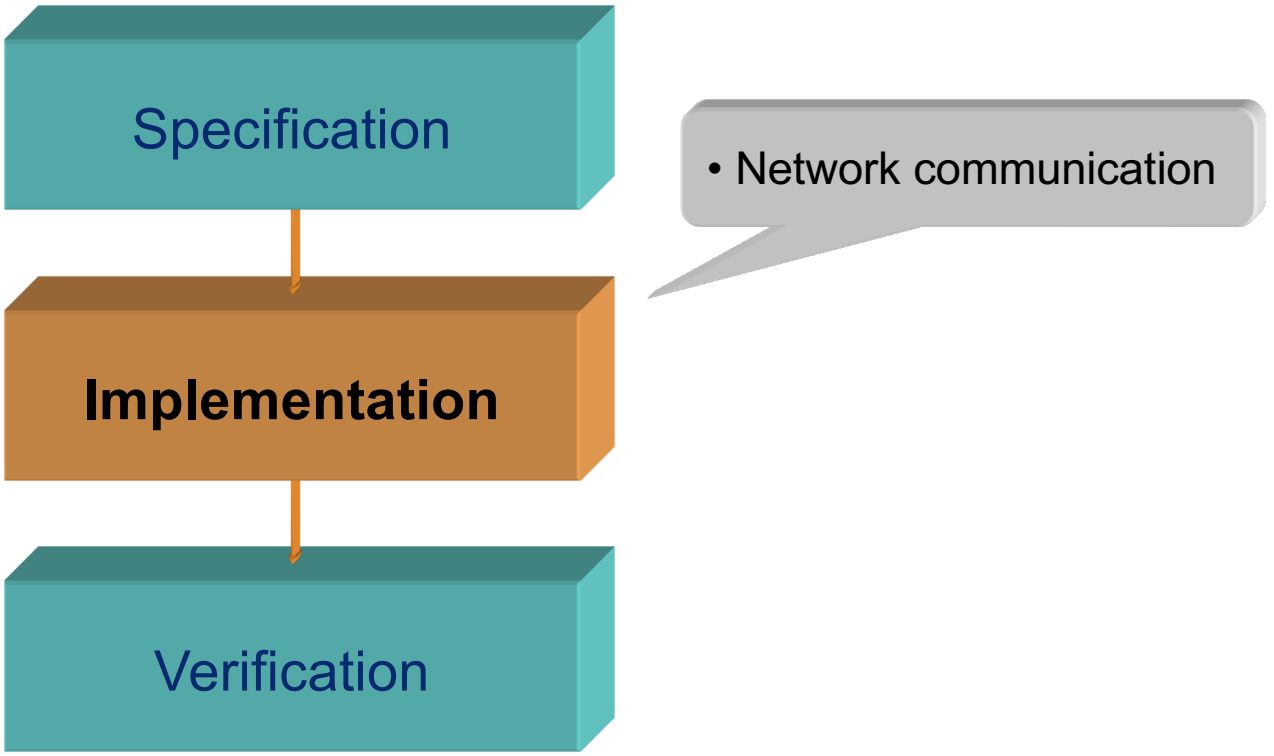
Real-Time Systems

Lecture #8

Victor Wallsten

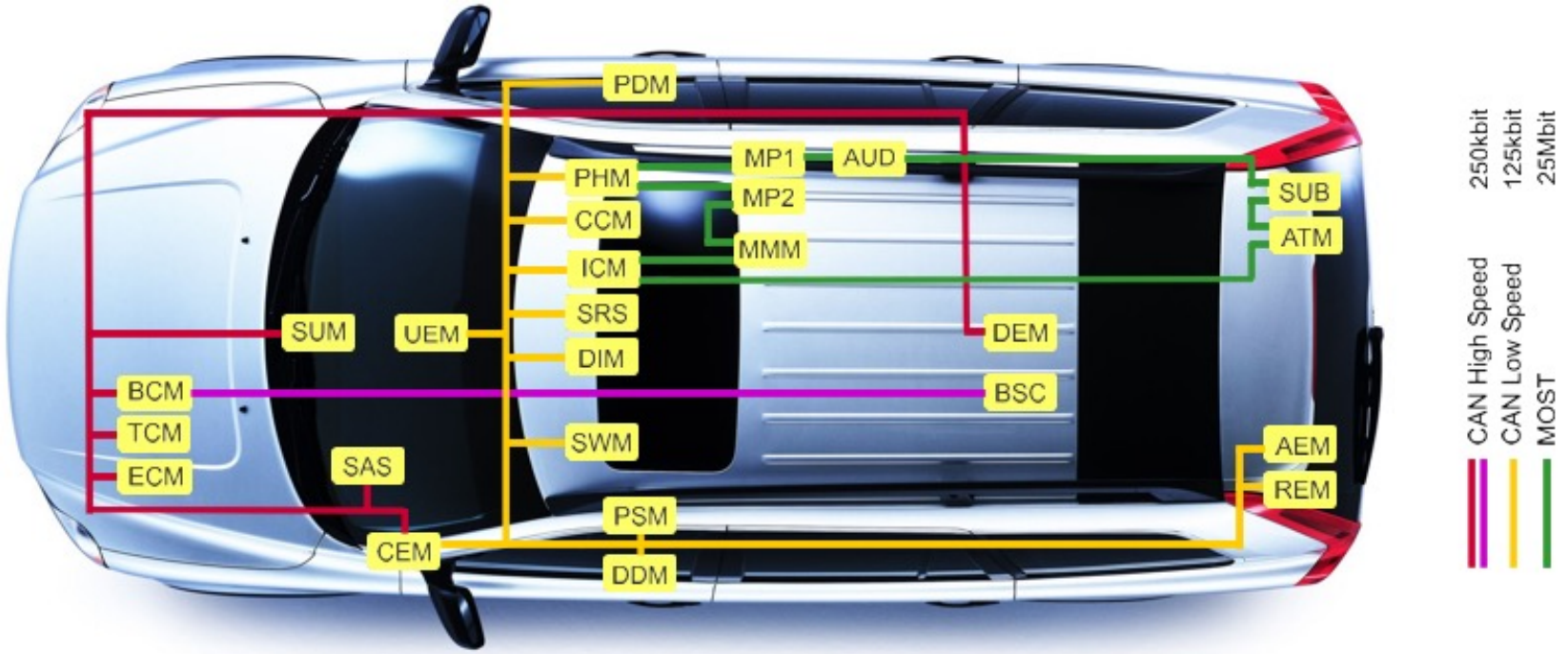
Department of Computer Science and Engineering
Chalmers University of Technology

Real-Time Systems

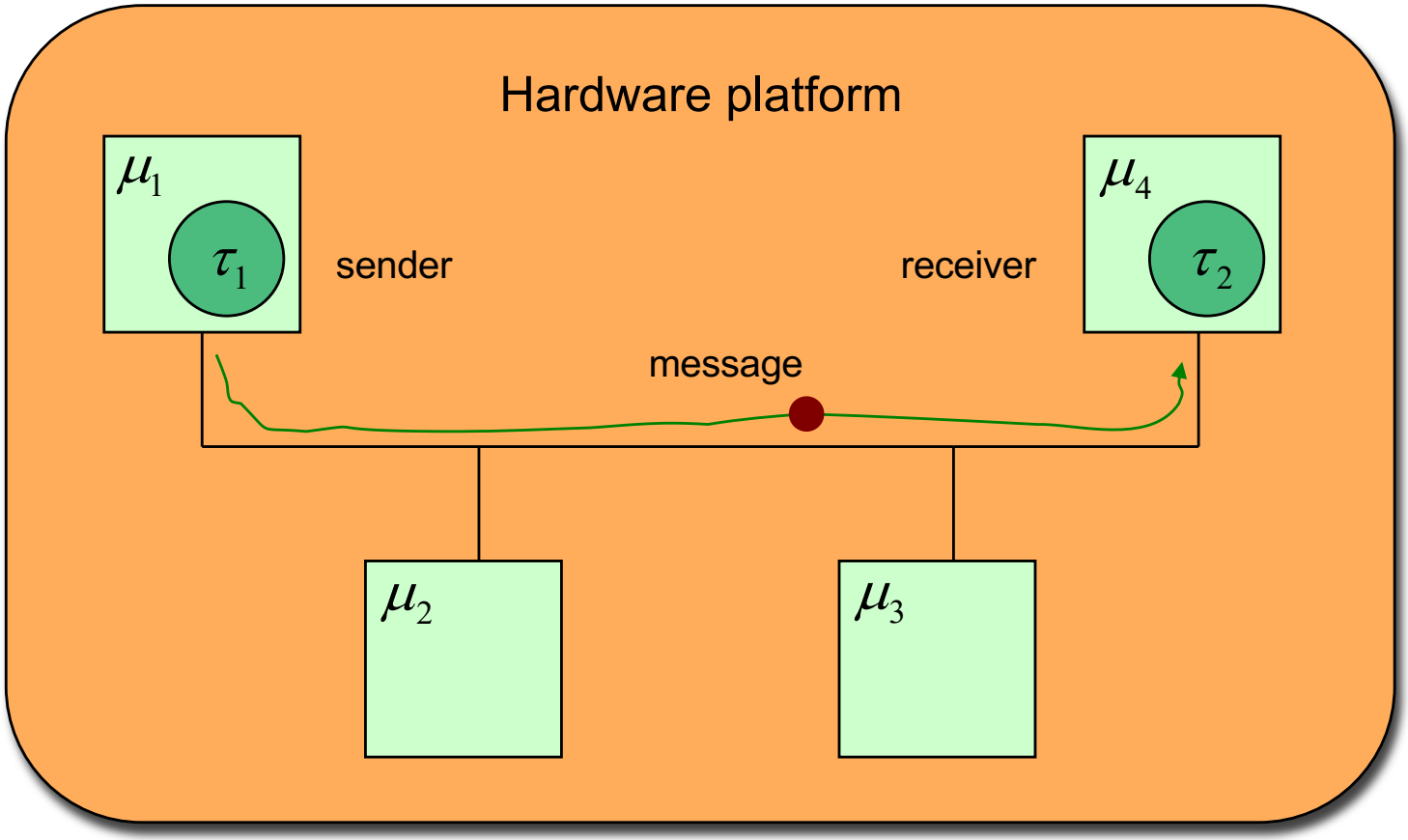


Network communication

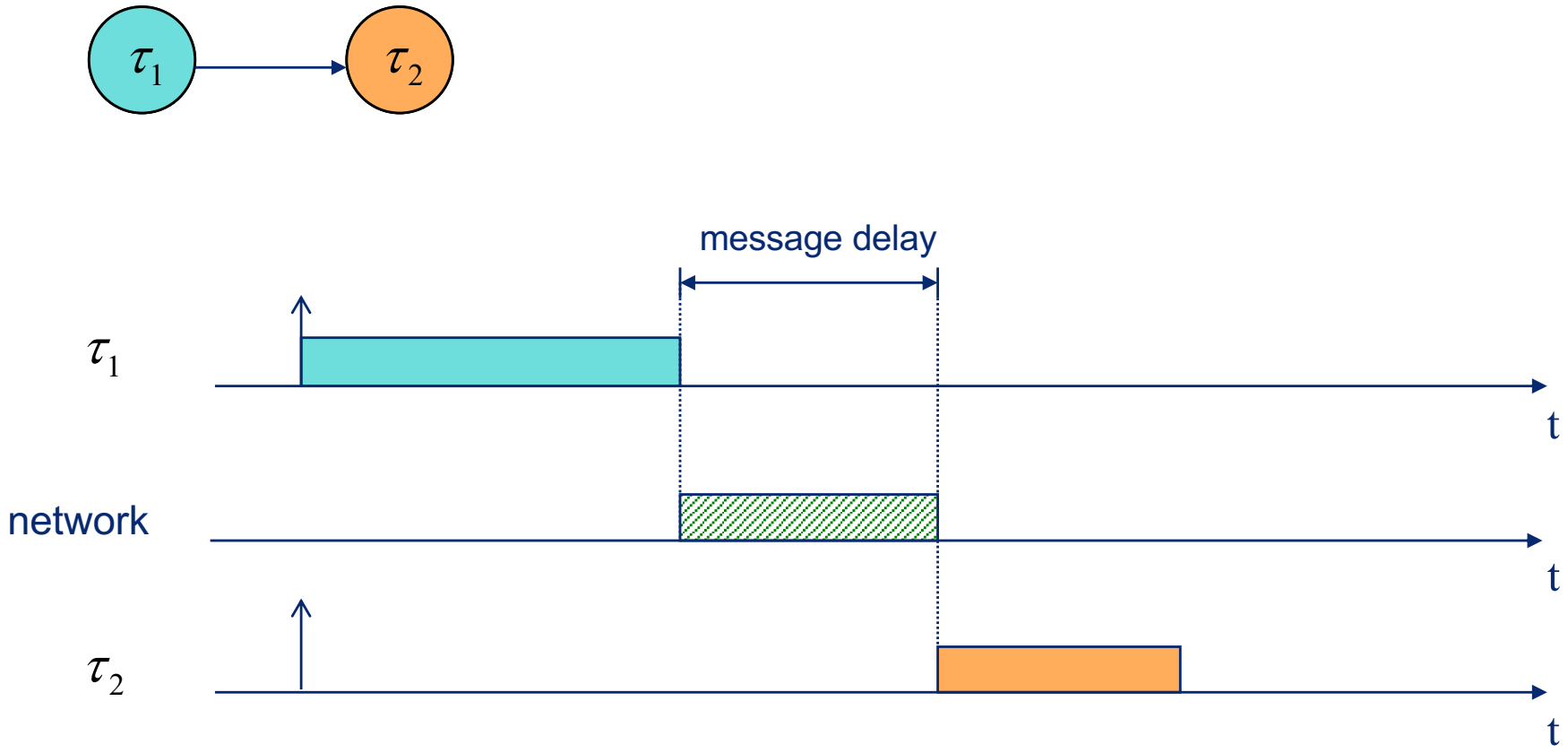
Embedded systems in the aircraft and automotive domain require support for real-time network communication



Network communication



Network communication



Network communication

Message delay:

- Message delays are caused by the following overheads:
  - Formatting (packetizing) the message
  - Queuing the message, while waiting for access to medium
  - Transmitting the message on the medium
  - Notifying the receiver of message arrival
  - Deformatting (depacketizing) the message

Formatting/deformatting overheads are typically included in the execution time of the sending/receiving task.

Network communication

Queuing delay:

- The cause of the queuing delay for a message depends on the actual network used. For example:
  - Waiting for a corresponding time slot (e.g., FlexRay)
  - Waiting for a transmission token (e.g., Token Ring)
  - Waiting for a contention-free transmission (e.g., Ethernet)
  - Waiting for network priority negotiation (e.g., CAN)
  - Waiting for removal from priority queue (e.g., Switched Ethernet)

To be used in a real-time system with hard timing constraints the queuing delay must be bounded.

Network communication

Transmission delay:

- The delay for transmitting the message is the sum of:

  a frame delay
  - message length (bits)
  - data rate (bits/s)

$$t_{\text{frame}} = \frac{N_{\text{frame}}}{R}$$

  and a propagation delay
  - communication distance (m)
  - signal propagation velocity (m/s)

$$t_{\text{prop}} = \frac{L}{v}$$

Network communication

How is the message transfer synchronized between communicating tasks?

- Asynchronous communication:
  - Sending and reception of messages are performed as independent operations at run-time.
- Synchronous communication:
  - Sending and receiving tasks synchronize their network medium access at run-time.

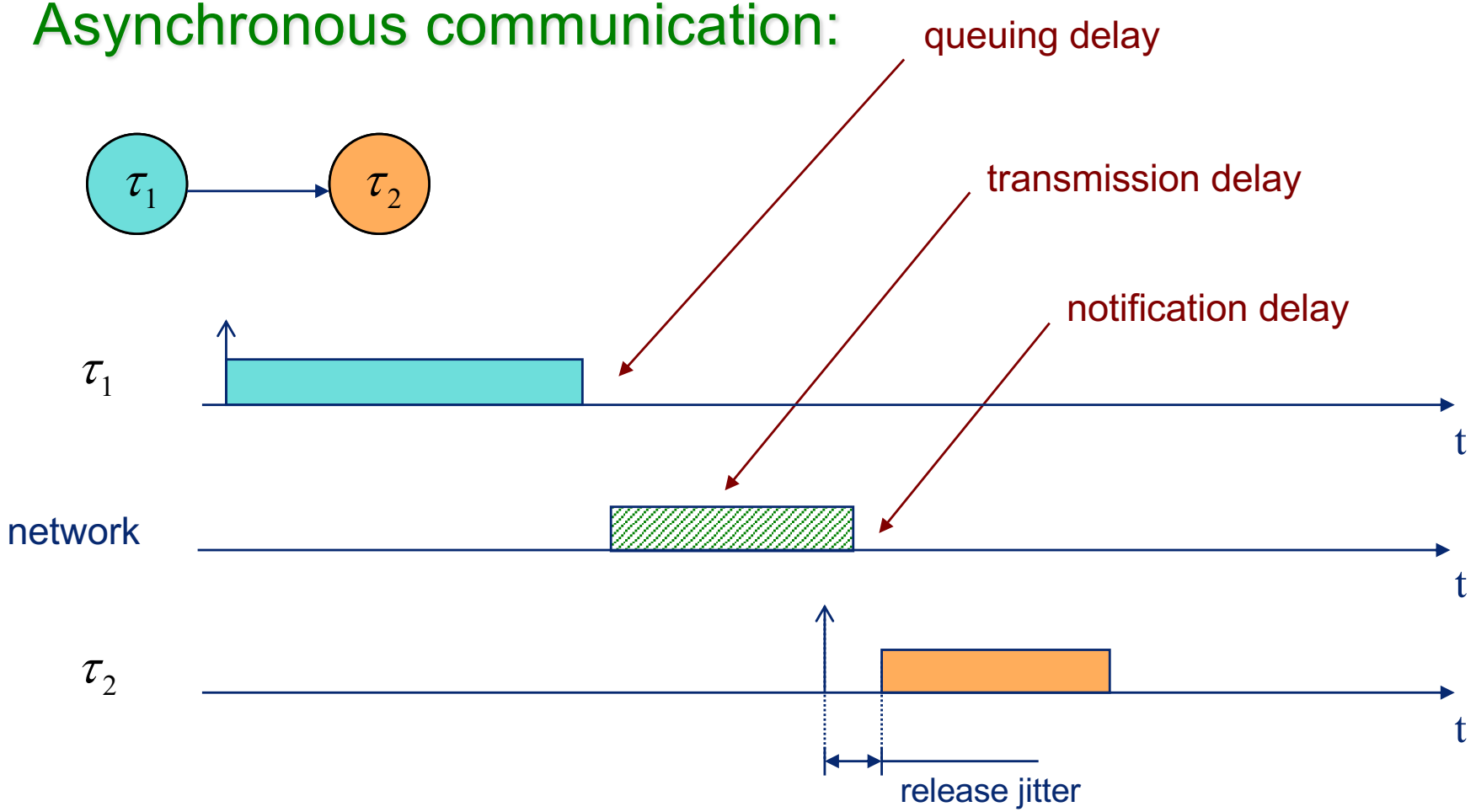
Network communication

Asynchronous communication

- Implementation:
  - Network controller chip administrates message transmission and reception (example: CAN, Ethernet)
  - Interrupt handler notifies the receiver
- Release jitter:
  - Queuing delays at sender and notification delay at receiver cause variations in message arrival time
  - Arrival-time variations gives rise to release jitter at receiving task (which may negatively affect schedulability)
  - Release jitter is minimized by adding offsets to receiving tasks

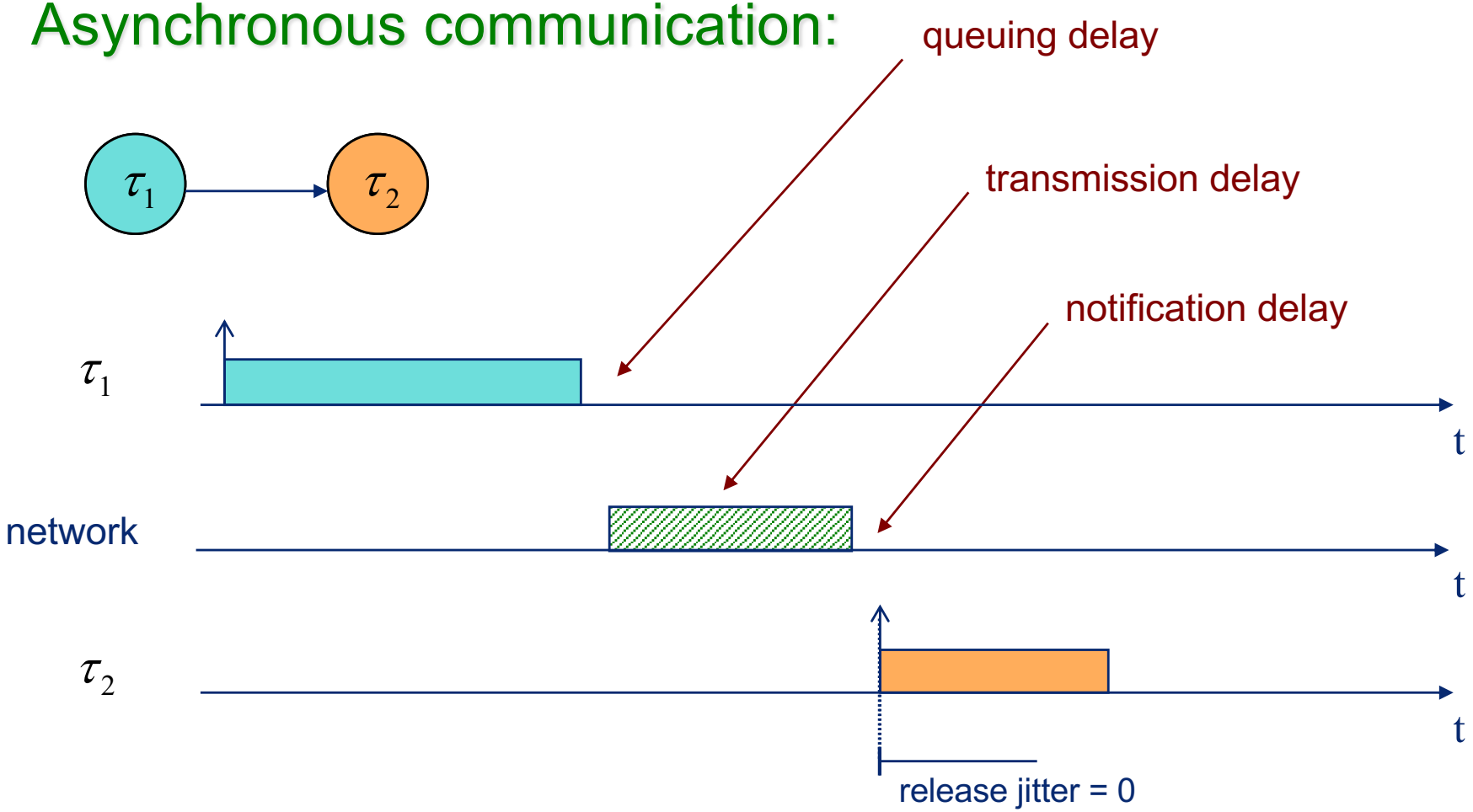
Network communication

Asynchronous communication:



Network communication

Asynchronous communication:



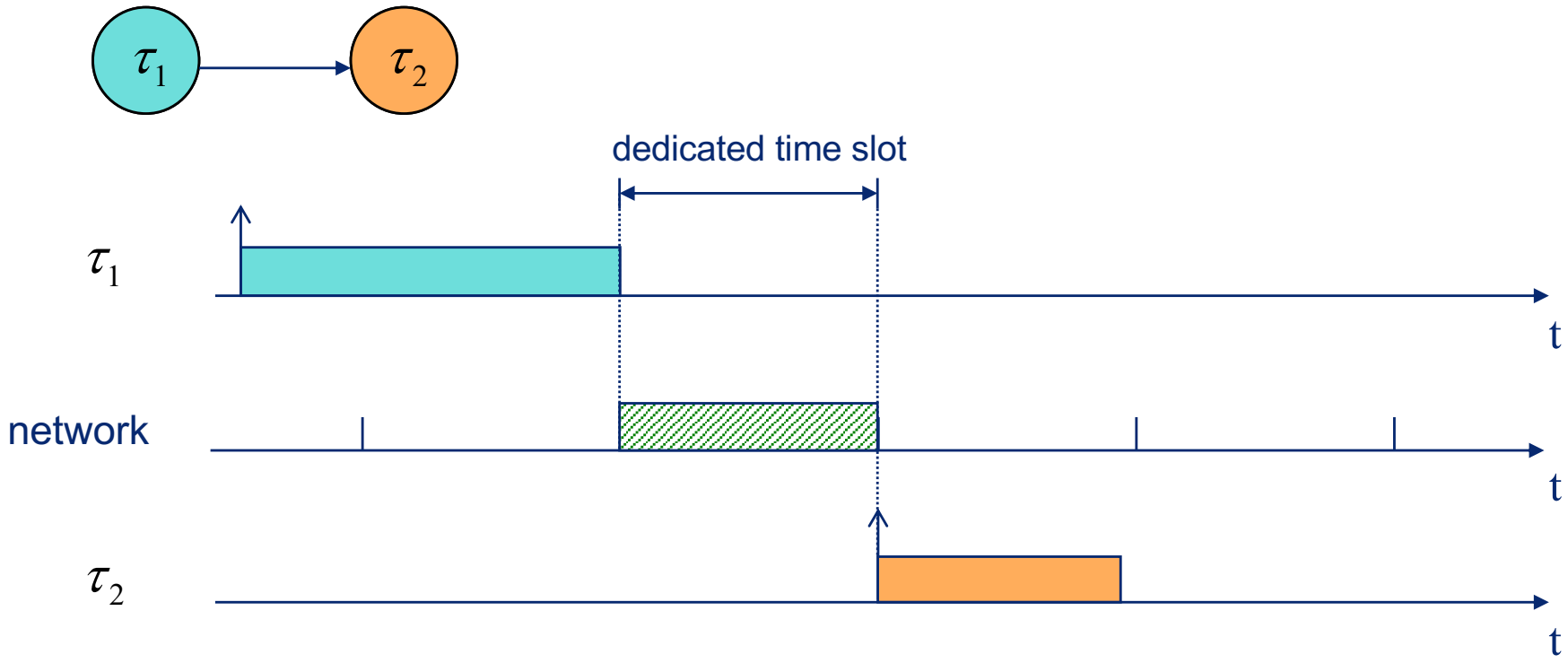
Network communication

Synchronous communication

- Implementation:
  - Network controller chip makes sure message transmission and reception occurs within a dedicated time slot in a TDMA bus network (example: FlexRay)
  - Off-line static (time-table) scheduling is used for matching the time slot with the execution of sending and receiving tasks
  - Queuing and notification delays can be kept to a minimum by instructing the off-line scheduling algorithm to use jitter minimization as the scheduling objective

Network communication

Synchronous communication:



Network communication

How is the message transferred onto the medium?

- Contention-free communication:
  - Senders need not contend for medium access at run-time
  - Examples: TTCAN, FlexRay, Switched Ethernet
- Token-based communication:
  - Each sender using the medium gets one chance to send its messages, based on a predetermined order
  - Examples: Token Ring, FDDI
- Collision-based communication:
  - Senders may have to contend for the medium at run-time
  - Examples: Ethernet, CAN

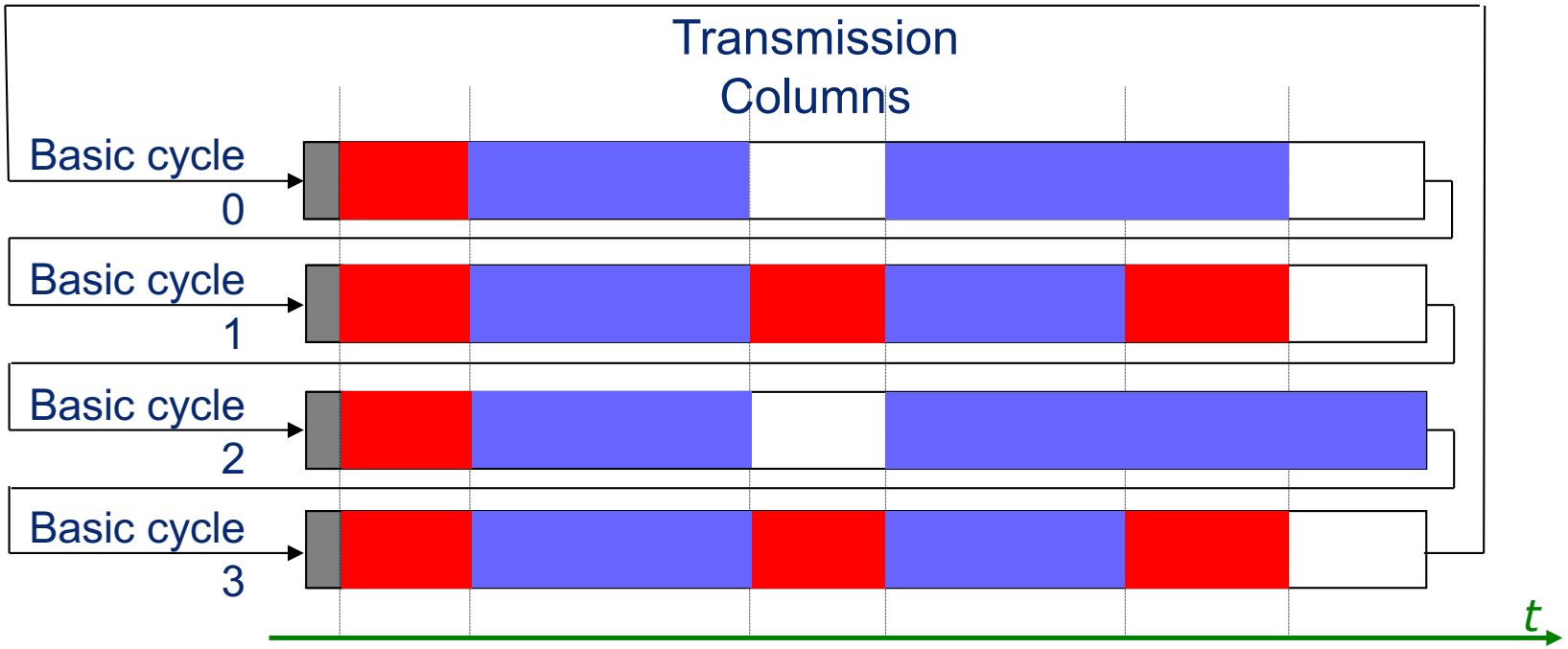
Network communication

Contention-free communication:

- One or more dedicated time slots for each task/processor
  - Shared communication bus
  - Medium access is divided into communication cycles (normally related to least-common-multiple of task periods)
  - Dedicated time slots provide bounded queuing delays
  - TTCAN ("exclusive mode"), FlexRay ("static segment")
- One sender only for each communication line
  - Point-to-point communication networks with link switches
  - Output and input buffers with deterministic queuing policies in switches provide bounded queuing delays
  - Switched Ethernet

The TTCAN protocol

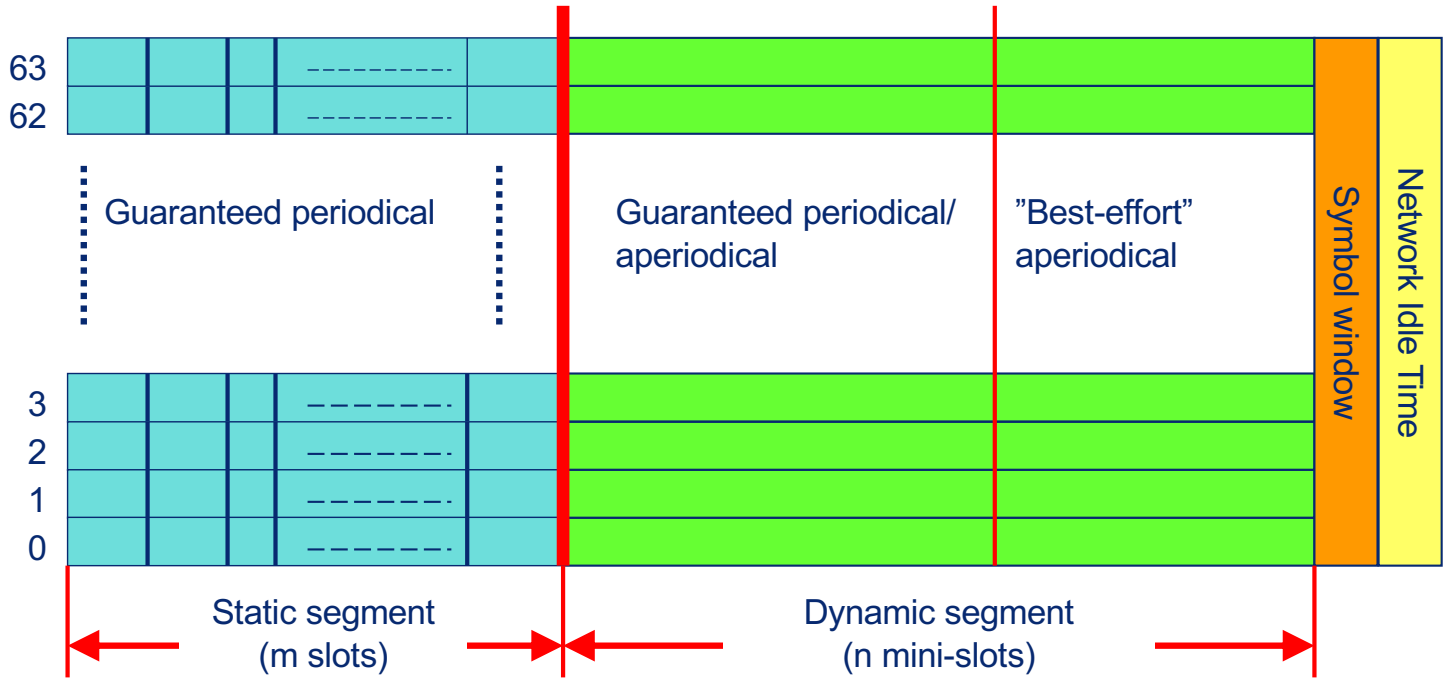
- ”Exclusive” – guaranteed service
- ”Arbitration” – guaranteed service (high ID), best effort (low ID)
- ”Reserved” – for future expansion...



The FlexRay protocol

”Static segment” (compare w/ TTCAN ”Exclusive”)
– guaranteed service

”Dynamic segment” (compare w/ TTCAN ”Arbitration”)
– guaranteed service (high ID), ”best effort” (low ID)



Max 64 nodes on a Flexray network.

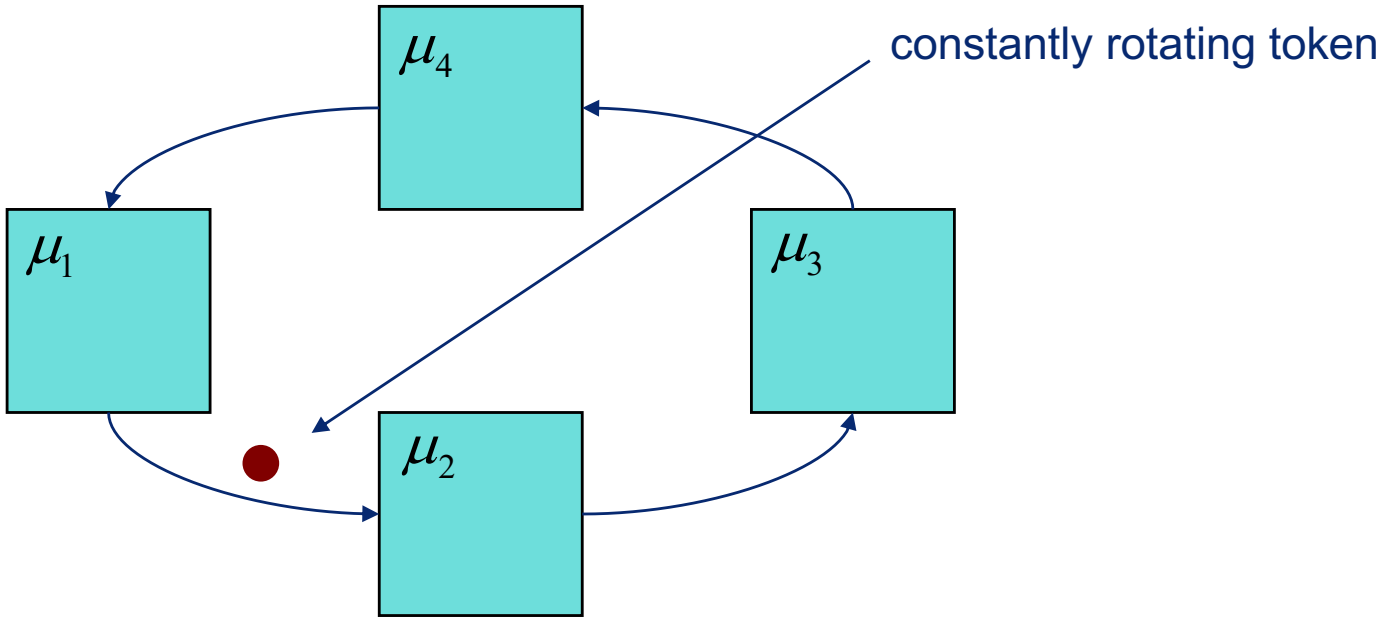
Network communication

Token-based communication:

- Utilize a token for the arbitration of message transmissions on a shared medium
  - The sender is only allowed to transmit its messages when it possesses the token
  - Message priorities can provide bounded queuing delays
- Examples:
  - Token Ring (IEEE 802.5)
  - FDDI (ANSI X3T9.5)

Token-based communication

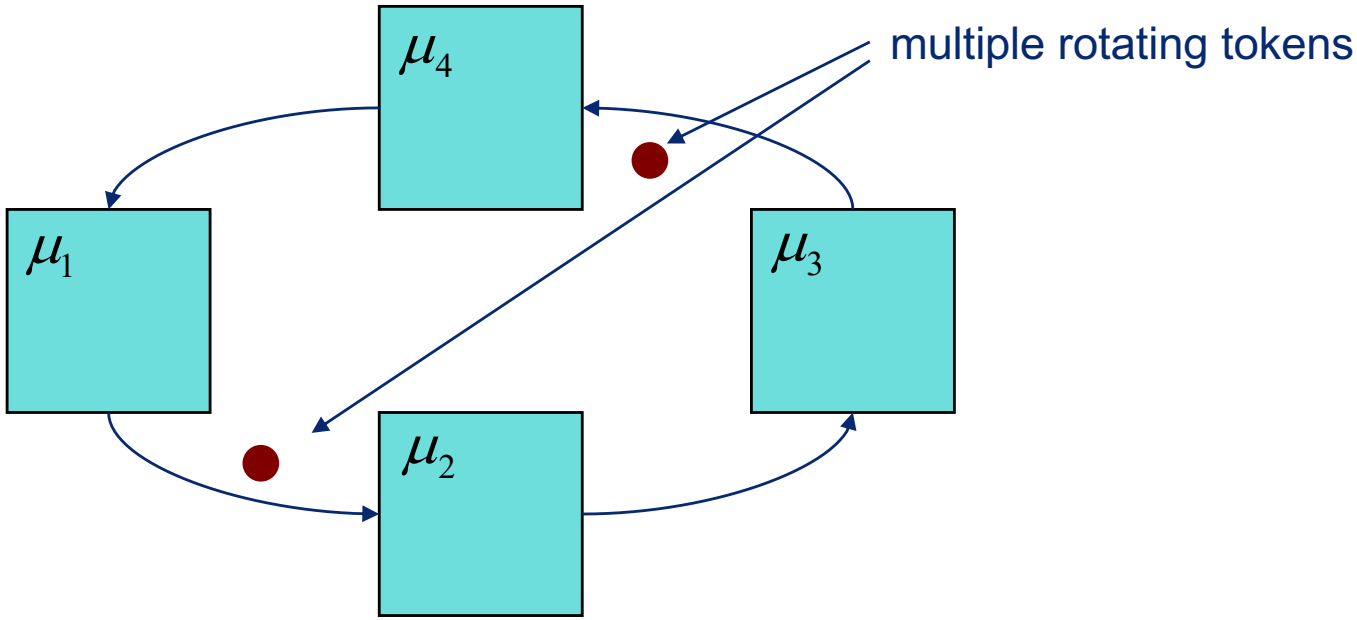
Token Ring: (IEEE 802.5)



Single rotating ring, twisted pair, 4 Mbit/s

Token-based communication

Fiber Distributed Data Interface: (ANSI X3T9.5)



Dual counter-rotating rings, optical fibre, 100 Mbit/s

Network communication

Collision-based communication:

- Utilize collision-detect mechanism to determine validity of message transmissions on a shared medium
  - The sender tries to send messages independently of other senders' intention to do so
  - Attempts may be done at any time or when some specific network state occurs
- Examples:
  - Ethernet w/ multiple senders (IEEE 802.3)
  - CAN (ISO 11898)

Collision-based communication

Ethernet protocols w/ multiple senders:

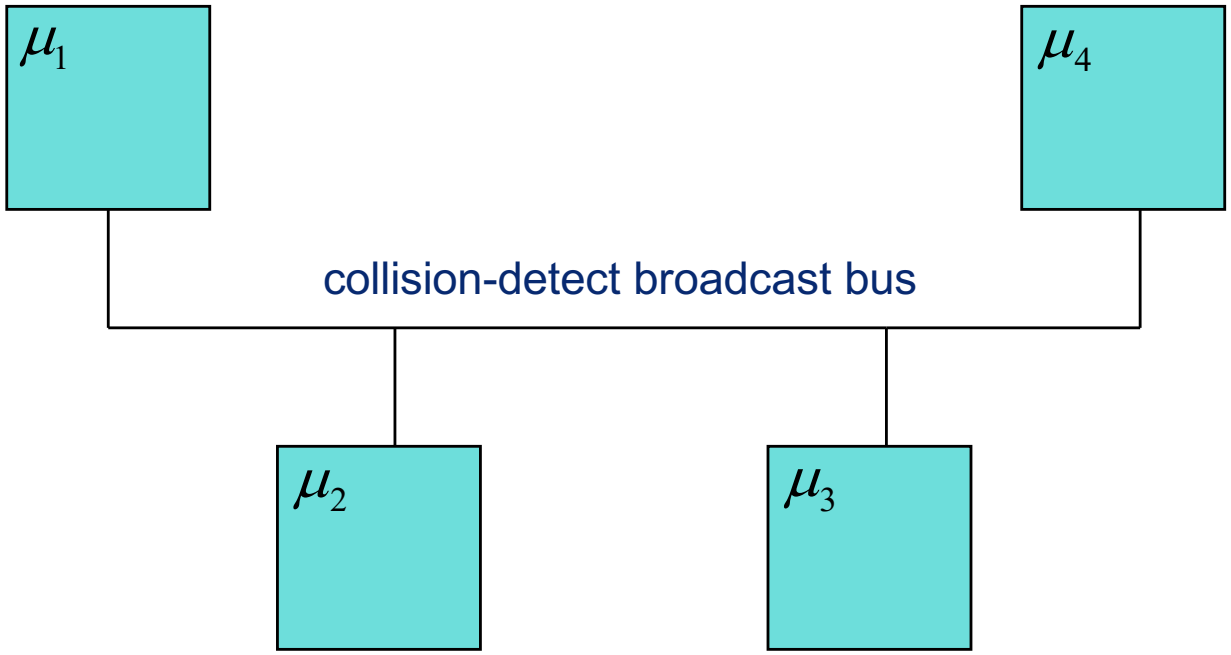
- Senders attempt to send a complete message
- If messages collide, all transmissions are aborted
- After collision, re-transmission is made after a random delay

Message queuing delay can in general not be bounded!

Therefore, these protocols do not give any guarantees for meeting imposed message deadlines!

Collision-based communication

Controller Area Network (CAN): (ISO 11898)



Broadcast serial bus, dual wire (resistor terminated), 1 Mbit/s

Collision-based communication

Controller Area Network (CAN):

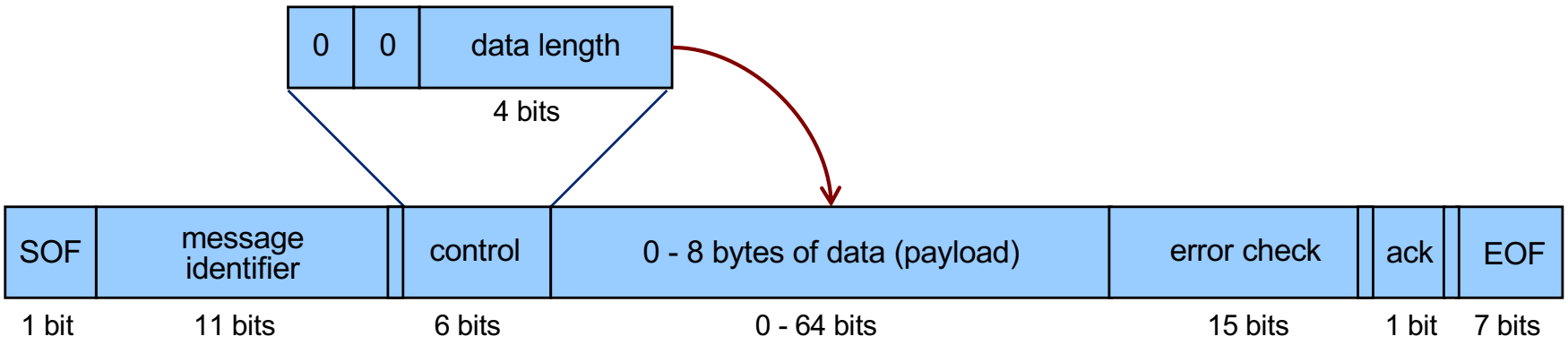
- Senders transmit a message header (with an identifier)
- If messages collide, a hardware-supported protocol is used to determine what sender will be allowed to send the rest of the message; transmissions by other senders are aborted

Message queuing delay can be bounded with appropriate identifier assignment.

Therefore, this protocol makes it possible to meet imposed message deadlines.

The CAN protocol

CAN message frame format: (short format)

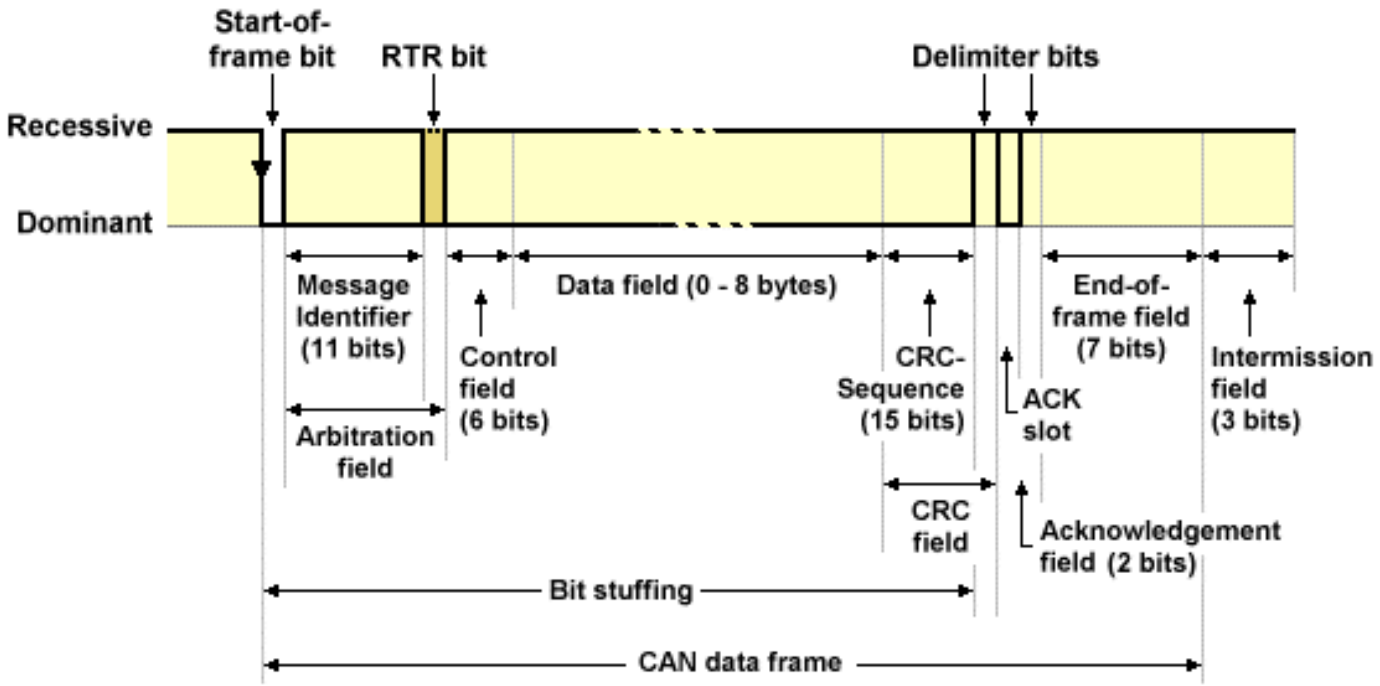


Message identifier can be used for several purposes:

- enable receiver to filter messages (original purpose)
- assign a priority to the message (low number ⇒ high priority)

The CAN protocol

CAN message frame format: (short format)



The CAN protocol

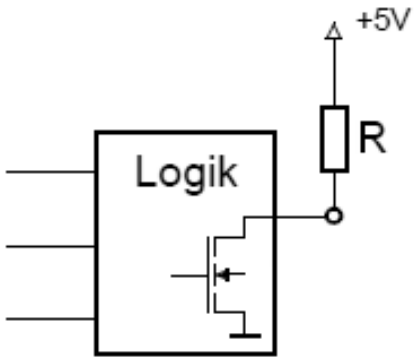
CAN protocol: (binary countdown)

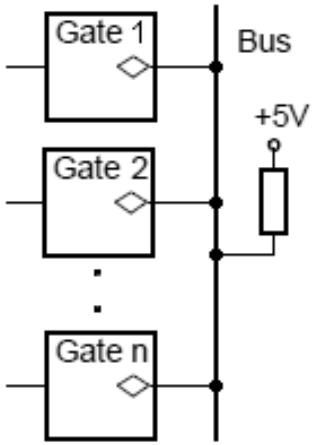
Wired-AND:

Each node monitors the bus while transmitting.

If multiple nodes are transmitting simultaneously
and one node transmits a ’0’,
then all nodes will see a ’0’.

If all nodes transmit a ’1’,
then all nodes will see a ’1’.





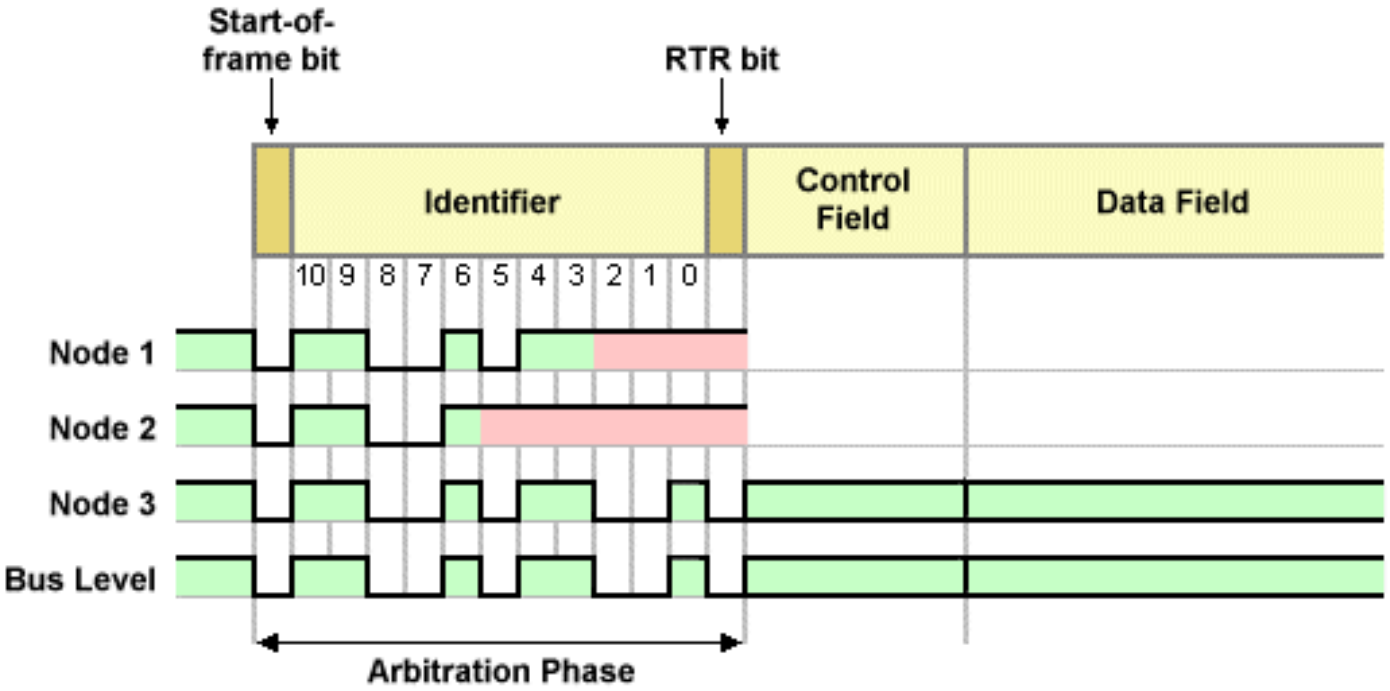
The CAN protocol

CAN protocol: (binary countdown)

1. Each node with a pending message waits until bus is idle.
2. The node begins transmitting the highest-priority message pending on the node. Identifier is transmitted first, in the order of most-significant bit to least-significant bit.
3. If a node transmits a recessive bit (’1’) but sees a dominant bit (’0’) on the bus, then it stops transmitting since it is not transmitting the highest-priority message in the system.
4. The node that transmits the last bit of its identifier without detecting a bus inconsistency has the highest priority and can start transmitting the rest of the message frame.

The CAN protocol

CAN protocol: (binary countdown)



Interrupt handlers in TinyTimber

Example: implementing a CAN interrupt handler:

1. Define class `Can`, and add state variables for:
   - the hardware base address of the device
   - call-back information for a method if data received by the handler needs to be taken care of by the user-level code (the call back should be done using an `ASYNC()` call)
   - necessary local storage (buffers, queues, etc)
2. Define a symbol `CAN_PORT0` representing the hardware base address of the device.

```
#define CAN_PORT0 device_hardware_address
```

3. Create an object `can0` of class `Can`, and initialize it with:
   - the hardware base address `CAN_PORT0`
   - any possible call-back information

Interrupt handlers in TinyTimber

Example: implementing a CAN interrupt handler (cont'd):

In file 'application.c':

```
App app = { initObject(), 0, 'X' };

void receiver(App*, int);

Can can0 = initCan(CAN_PORT0, &app, receiver);

void receiver(App *self, int unused) {    // call-back function
    CANMsg msg;
    CAN_RECEIVE(&can0, &msg);
    SCI_WRITE(&sci0, "Can msg received: ");
    SCI_WRITE(&sci0, msg.buff);
}
```

Interrupt handlers in TinyTimber

Example: implementing a CAN interrupt handler (cont'd):

4. Write an interrupt handler as a method `can_interrupt` and associate it with the object.
5. Declare a symbol `CAN_IRQ0` and assign to it the TinyTimber kernel's logical number of the hardware interrupt:

```
#define CAN_IRQ0 interrupt_logical_number
```

6. Inform the TinyTimber kernel that the method is a handler for interrupt `CAN_IRQ0`, by making a call to

```
INSTALL(&can0, can_interrupt, CAN_IRQ0);
```

This should be done before the call to `TINYTIMBER()`

Interrupt handlers in TinyTimber

Example: implementing a CAN interrupt handler (cont'd):

7. Provide an operation `CAN_INIT()` that takes care of performing any remaining initialization of the device.
8. Call `CAN_INIT()` in the "kick-off" method that was supplied as argument to the `TINYTIMBER()` call.

Interrupt handlers in TinyTimber

Example: implementing a CAN interrupt handler (cont'd):

In file 'application.c':

```
void startApp(App *self, int arg) {
    CANMsg msg;
    CAN_INIT(&can0);
    …
    CAN_SEND(&can0, &msg);
}

int main() {
    INSTALL(&can0, can_interrupt, CAN_IRQ0);
    TINYTIMBER(&app, startApp, 0);
}
```

Interrupt handlers in TinyTimber

Example: implementing a CAN interrupt handler (cont'd):

In file 'canTinyTimber.h':

```
typedef unsigned char uchar;

typedef struct {
    uchar msgId;  // Valid values: 0-127
    uchar nodeId; // Valid values: 0-15
    uchar length;
    uchar buff[8];
} CANMsg;
```

`CANMsg` holds the user-relevant parts of the CAN message frame.

- `nodeID` holds the four least-significant bits of the identifier field
- `msgID` holds the seven most-significant bits of the identifier field

Real-Time Systems

7.5 credit points

Professor Jan Jonsson

Department of Computer Science and Engineering
Chalmers University of Technology

Course teachers

Jan Jonsson:

- Examiner and course responsible
- Lectures (+ a few exercise sessions)
- Laboratory sessions

Victor Wallsten:

- Course assistant
- Exercise sessions (+ a few lectures)
- Laboratory sessions

Course organization

Lectures (16 of them)

Lectures are offered in full class, with the goal of introducing the real-time programming paradigm and basic scheduling theory as well as demonstrating how the paradigm and theory are applied in practice.

Special sessions (3 of them)

Special sessions are offered in full class on certain weeks, and should be seen as a complement to the lectures. Examples of topics for the special sessions:

- introduction to sound generation and music theory
- discuss solutions to exercise problems or old exam problems

Course organization

Exercise sessions (7 of them)

Exercise sessions are offered in full class, with the goal of further exploring aspects of the laboratory assignment and the scheduling theory.

Laboratory assignment (one big assignment)

The assignment is done in groups and gives the student practical experience with programming of a time-critical embedded system, using the C programming language and the TinyTimber kernel.

The assignment runs for a restricted time period, during which loan equipment is used for solving programming problems.

Laboratory sessions are used for guidance and debugging, as well as examination of the groups' software solutions.

Course aim

After the course, the student should be able to:

- Formulate requirements for embedded systems with strict constraints on computational delay and periodicity.
- Construct concurrently-executing tasks (software units) for real-time applications that interface to sensors and actuators.
- Describe the principles and mechanisms used for designing run-time systems and communication networks for real-time applications.
- Demonstrate knowledge about the terminology used within the theory of scheduling and computational complexity.
- Apply the basic analysis methods used for verifying the temporal correctness of a set of executing tasks.

Course examination

Written exam

The course contents are examined by means of a written exam.
Date and time of ordinary exam: March 17, 2025 @ 08:30-12:30.

Laboratory assignment

The assignment is examined by means of parts 0, 1 and 2 of the Lab-PM.

Final grade

The written exam and the laboratory assignment are both given a score with grade (U, 3, 4, 5). To pass the course the score of each much be equivalent to a grade of 3 or higher. The final grade is based on the scores from the exam and the assignment.

Changes in the course

Since 2018:

- The final course grade is based on the scores for the laboratory assignment and the written exam
- The number of course participants is restricted; pass grades in prerequisite courses guarantee a seat for laboratory assignment

Since 2021/22:

- The work with the laboratory assignment can partly be done at home using loan equipment
- Written exam has a new grading procedure

Since 2025:

- The examination of the laboratory assignment no longer encompasses a written project report

Course material

To download: (via Canvas system)

- Lecture notes and exercise notes. [Powerpoint hand-outs]
- Research articles and book excerpts. [recommended reading only]
- Exercise compendium.
- Lab-PM − Part 0, 1 and 2.
- “Programming with the TinyTimber kernel” compendium.
- Template code. [for target computer software]
- Handbooks and data sheets. [for target computer hardware]
- Development tools. [available for Windows, macOS, Linux]

Course information and support

Examiner/lecturer:

- Questions related to the course are primarily answered in conjunction with lectures, exercises or laboratory sessions.
- Otherwise ask questions via email or Canvas messaging.

Canvas system:

- Get complete information about the course
- Download course material
- Form project groups and submit reports
- View examination progress and awarded grades

https://chalmers.instructure.com/courses/33127

Course contents

What this course is all about:

1. Construction methods for real-time systems
   - Specification, implementation, verification
   - Application constraints: origin and implications
2. Programming of concurrent real-time applications
   - Task and communication models (C with TinyTimber kernel)
   - I/O and interrupt programming (C with TinyTimber kernel)
3. Verification of system's temporal correctness
   - Derivation of worst-case task execution times
   - Fundamental scheduling theory

What is a real-time system?

"A real-time system is one in which the correctness of the system depends not only on the logical result of computation, but also on the time at which the results are generated"

J. Stankovic, "Misconceptions of Real-Time Computing", 1988

What is a real-time system?

It is not only about high-performance computing!

Real-time systems must meet timing constraints

High-performance computing maximizes average throughput

Average performance says nothing about correctness!

"A statistician drowned while crossing a stream
that was, on average, 6 inches deep"

Real-time system are instead usually optimized with respect to perceived "robustness" (control systems) or "comfort" (multimedia)

What is a real-time system?

Typical properties of a real-time system:

- Application-specific design
  - Part of a bigger system (“embedded system”)
  - Carefully specified system properties
  - Well-known operating environment
- Strict timing constraints
  - Responsiveness (= deadline)
  - Periodicity (= sampling rate)
  - Important design parameters in the context of system safety
  - Should be verified at design time to be met at run-time

What is a real-time system?

Typical properties of a real-time system (cont'd):

- Strict safety requirements

  "Safety must be considered early in the design process"

  – Safety standards and certification
    - IEC 61508 (industrial systems)
    - IEC 62304 (medical systems)
    - ISO 26262 (automotive systems)
    - DO-178C (airborne systems)

  – Programming language restrictions (e.g. MISRA C)

  – Run-time system restrictions (e.g. cyclic executive)

  – Thorough testing of software and hardware components

  – Reliability in presence of component faults ("fault tolerance")

What is a real-time system?

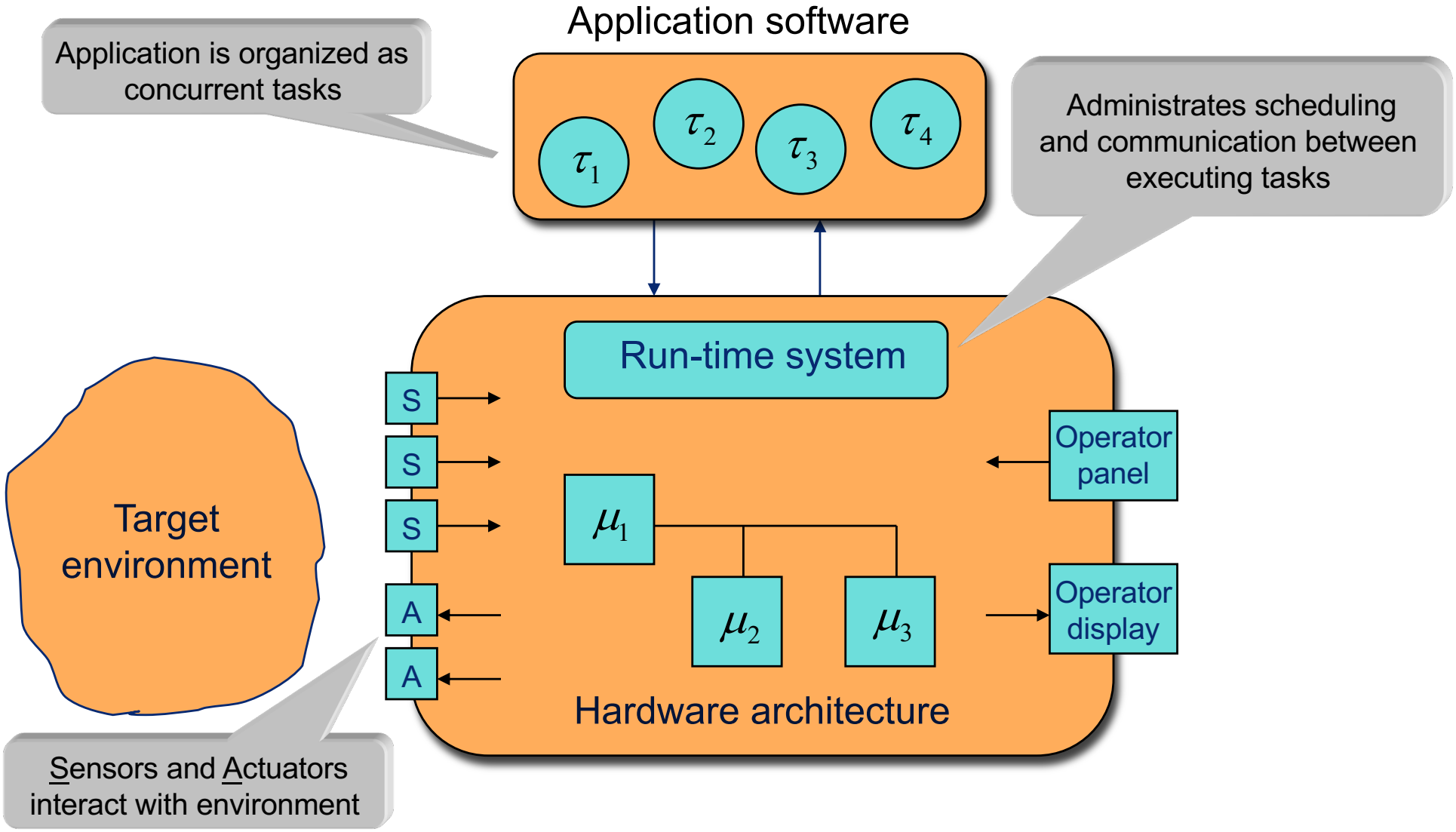
Examples of real-time systems:

- Control systems
  - Manufacturing systems; process industry
  - Cars, aircrafts, submarines, satellites

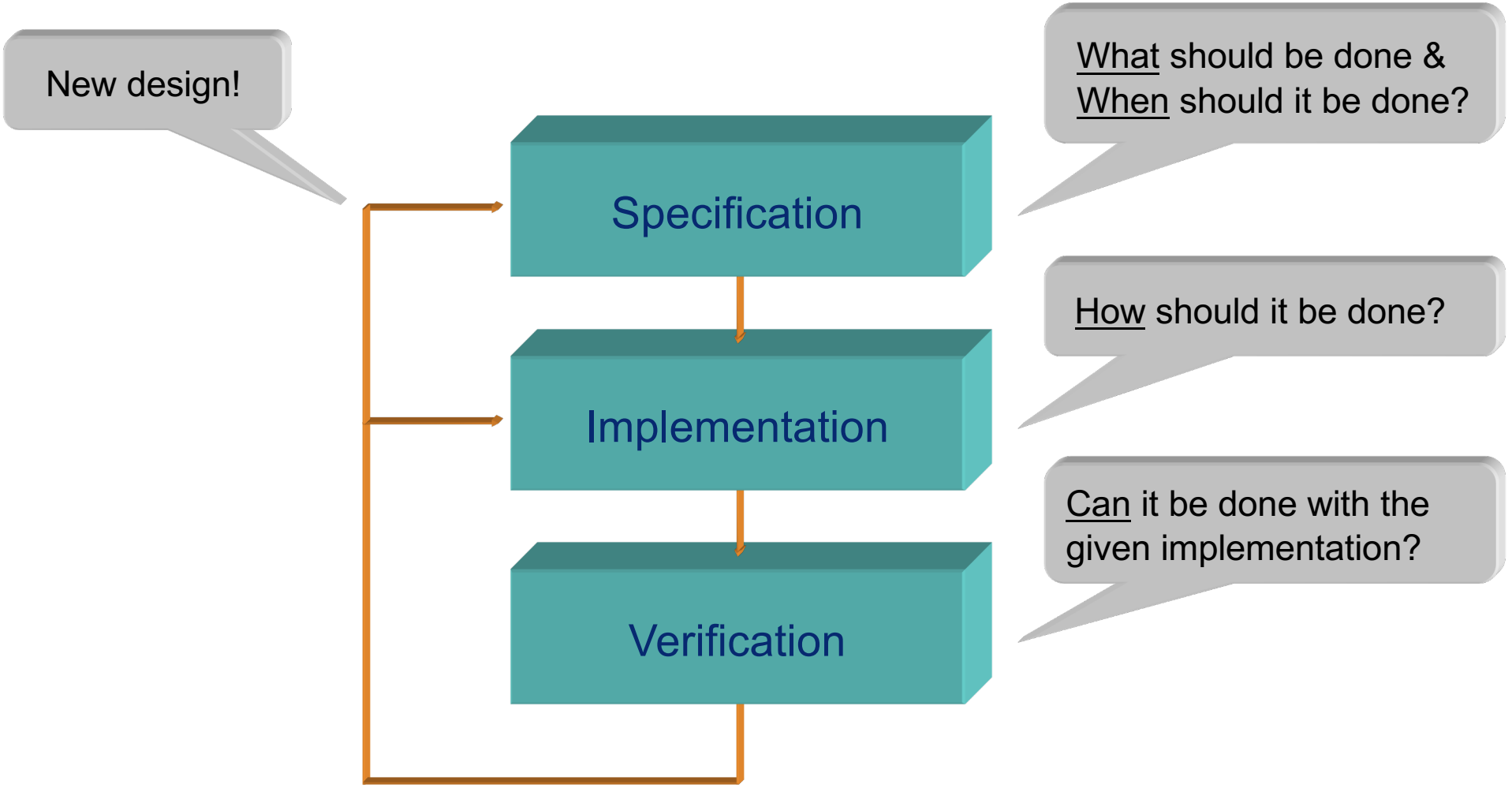
- Transaction systems
  - E-commerce; ticket booking; teller machines; stock exchange
  - Wireless phones; telephone switches

- Multimedia
  - Portable music players, streaming music
  - Computer games; video-on-demand, virtual reality

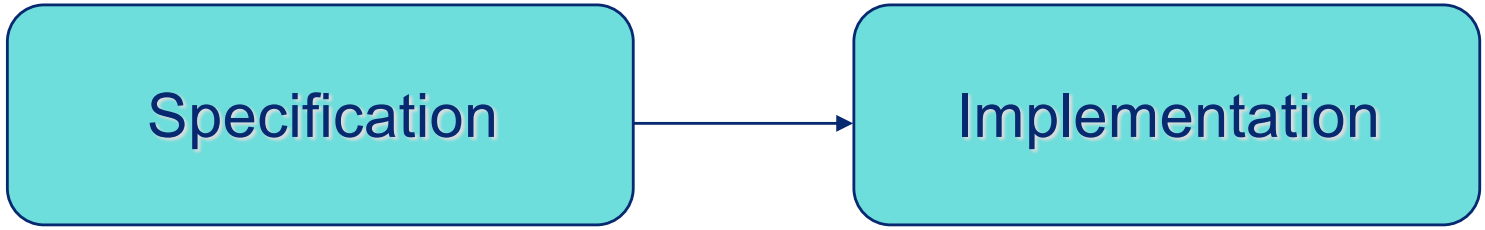
Real-time system components



Designing a real-time system



Specification



Requirements:		Constraints:
Sampling rate	→	Periodicity
Response time	→	Deadline
Reliability	→	Replication

Specification

Examples of application constraints:

- Timing constraints
  - A task must complete its execution within given time frames (example: task periodicity or deadline)
- Exclusion constraints
  - A task must execute a code region without being interrupted (example: a task needs exclusive access to a shared resource)
- Precedence constraints
  - A task must complete its execution before another task can start (example: a data exchange must take place between the tasks)

Specification

Where do the timing constraints come from?

- Laws of nature
  - Bodies in motion: robot arms in manufacturing, vehicles in traffic, missiles in flight
  - Inertia of the eye: minimal frame rate in film
- Mathematical theory
  - Control theory: recommended sampling rate
- Artificial derivation
  - Observable events: overall (global) timing constraints are given for a set of tasks, but individual (local) timing constraints are needed for each task.

    For example: data processing in a vehicle braking system

Specification

How critical are the constraints?

Hard constraints:

If the system fails to fulfill a timing constraint, the computational results is useless.

Non-critical: system can still function with reduced performance

- Navigational functions; diagnostics

Critical: system cannot continue to function

- Flight control system; control loop

Safety-critical: can cause serious damage or even loss of life

- Braking systems (ABS); defense system (missiles)

Correctness must be verified before system is put in mission!

Specification

How critical are the constraints?

Soft constraints:

Single failures to fulfill a timing constraint is acceptable, but the usefulness of the computational result is reduced (often to what can be considered useless).

- Reservation systems: seat booking for aircraft; teller machine
- E-commerce: stock trading, eBay
- Multimedia: video-on-demand, computer games, Virtual Reality

Statistical guarantees often suffice for these systems!

Implementation

Design choices during implementation:

- Application software
  - Programming language: determines run-time performance, code size and degree of timing verification that is possible.
  - Concurrency: determines degree of application parallelism that can potentially be exploited by the hardware.
- Run-time system:
  - Task and message scheduling policy: determines potential of meeting timing constraints, and sets limits of maximum processor and network utilization.

Implementation

Design choices during implementation:

- Hardware architecture:
  - Hardware parallelism: determines the degree of application parallelism that can actually be exploited
    - uniprocessor system: only pseudo-parallel execution is possible
    - multiprocessor system: truly parallel execution is possible
  - Microprocessor family: determines run-time performance, as well as difficulty in analyzing the worst-case execution time (WCET) of a software task.
  - Communication network technology: determines run-time performance, as well as difficulty in analyzing worst-case message delays.

Verification

Available approaches for checking correctness:

- Verification

  Capability: prove the absence of incorrect behavior

  – Requires formal methods based on mathematical models and theories (e.g., schedulability analysis)

- Testing and validation

  Capability: detect the presence of incorrect behavior

  – Testing: checking whether the system works correctly in a simplified environment (e.g. in a laboratory room)

  – Validation: checking whether the system works in its real target environment (e.g., in a vehicle or a satellite)

Verification

Existing challenges in checking correctness:

- Verification

  Model accurately the “sources of uncertainty” in the system

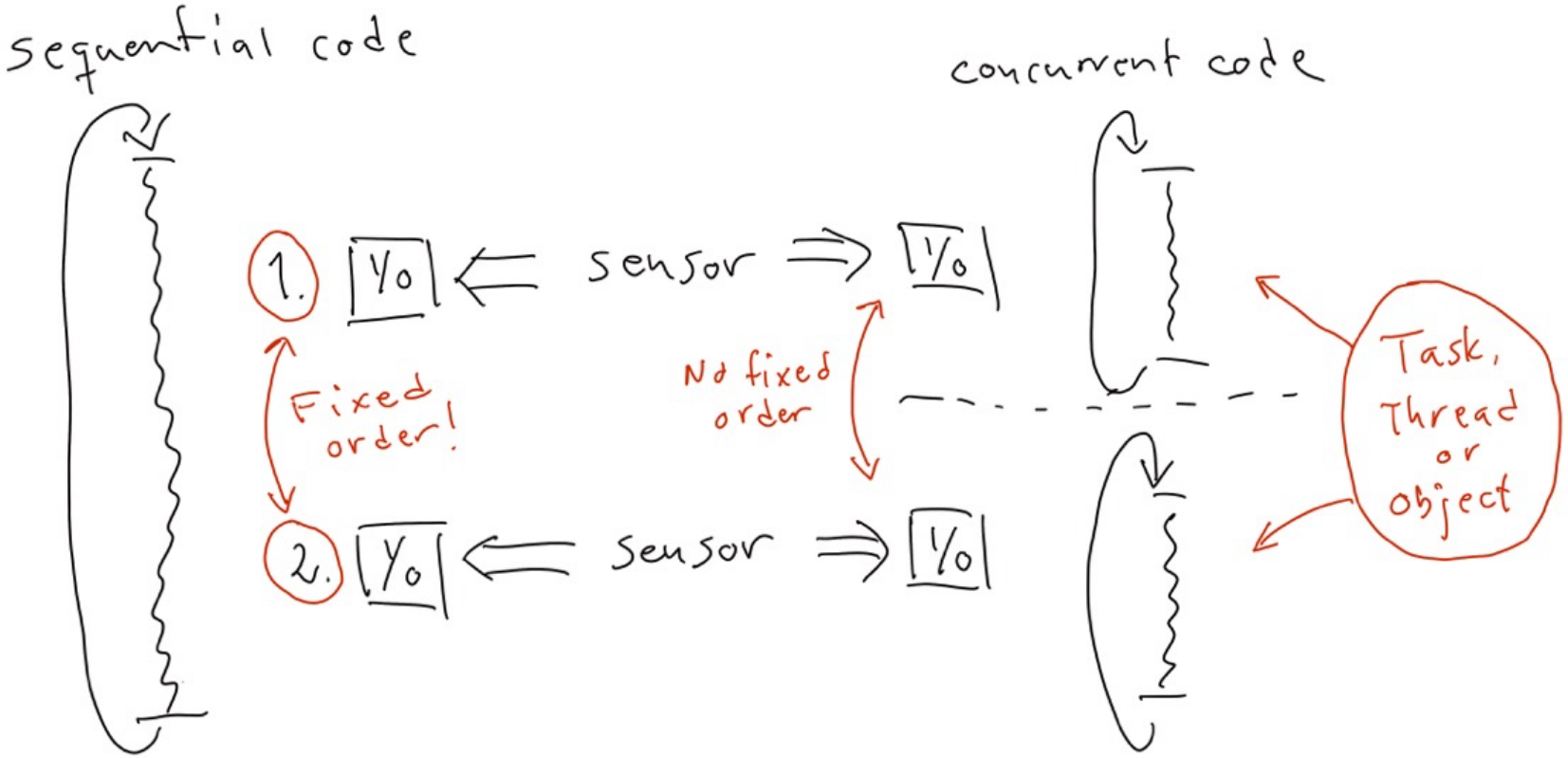
  – A single task’s WCET
  – Concurrent tasks’ execution interference on a processor
  – Concurrent tasks’ waiting times for shared resources

- Testing and validation

  Ensure that at least one case of incorrect behavior is detected

  – Considering only a subset of execution scenarios could mean that a pathological (worst) case will be overlooked
  – Considering all possible execution scenarios could require an unreasonable amount of time for testing or validation

Lecture #2 – blackboard scribble



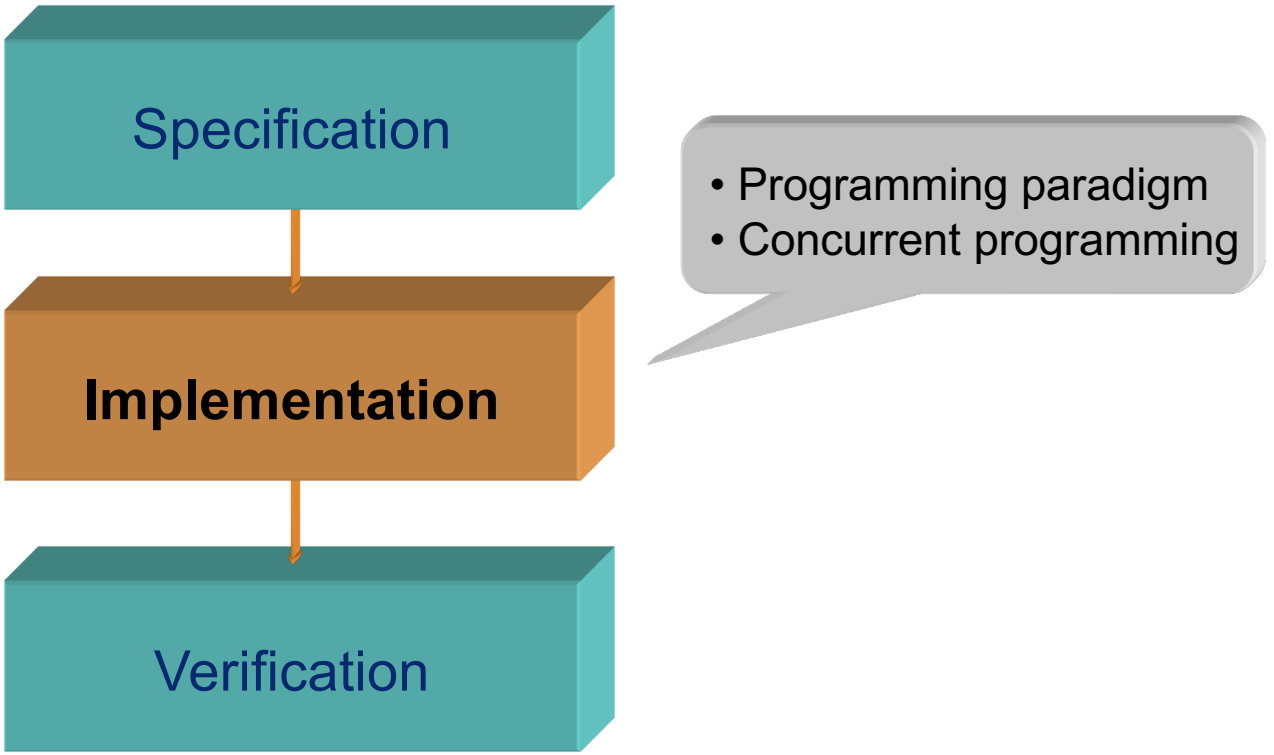
Real-Time Systems

Lecture #2

Professor Jan Jonsson

Department of Computer Science and Engineering
Chalmers University of Technology

Real-time systems



Real-time programming

Recommended programming paradigm:

- Concurrent programming
  - Reduces unnecessary dependencies between tasks
  - Enables a composable schedulability analysis
- Reactive programming
  - Certifies that tasks are activated only when work should be done; tasks are kept idle otherwise
  - Maps directly to the task model used in schedulability analysis
- Timing-aware programming
  - Certifies that timing constraints are visible at the task level
  - Enables priority-based scheduling of tasks, which in turn facilitates schedulability analysis

Real-time programming

Desired properties of a real-time programming language:

- Support for partitioning software into units of concurrency
  - tasks or threads (Ada 95, Java or C/C++ thread library)
  - object methods (C/C++ using the TinyTimber kernel)
- Support for communication with the environment
  - access to I/O hardware (e.g. view I/O registers as variables)
  - machine-level data types (e.g. bit-field type, address pointers)
- Support for the schedulability analysis
  - notion of (high-resolution) time (⇒ timing-aware programming)
  - task priorities (reflects constraints ⇒ timing-aware programming)
  - task delays (idle while not doing useful work ⇒ reactive model)
  - hardware interrupt handlers (event generators ⇒ reactive model)

Real-time programming

What programming languages are suitable?

- C, C++
  - Support for machine-level programming
  - Concurrent programming via run-time system (POSIX, TinyTimber)
  - Priorities and notion of time via run-time system (POSIX, TinyTimber)
- Java
  - Support for machine-level programming
  - Support for concurrent programming (threads)
  - Support for priorities and notion of time (Real-Time Java)
- Ada 95
  - Support for machine-level programming
  - Support for concurrent programming (tasks)
  - Support for priorities and notion of time

Why concurrent programming?

Most real-time applications are inherently parallel

- Events in the target system's environment often occur in parallel
- By viewing the application as consisting of multiple tasks, this parallel reality can be reflected
- While a task is waiting for an event (e.g., I/O or access to a shared resource) other tasks may execute

Enables a composable schedulability analysis

- First, the local timing properties of each task are derived
- Then, the interference between tasks are analyzed

System can obtain reliability properties

- Redundant copies of the same task makes system fault-tolerant

Issues with concurrent programming

Access to shared resources

- Many hardware and software resources can only be used by one task at a time (e.g., processor, data structures)
- Only pseudo-parallel access is possible in many cases

Synchronization and information exchange

- System modeling using concurrent tasks also introduces a need for synchronization and information exchange.

Concurrent programming must hence be supported by an advanced run-time system that handles the scheduling of shared resources and communication between tasks.

Support for concurrent programming

Support via programming language semantics:

- Program is easier to read and comprehend, which means simpler program maintenance
- Program code can be easily moved to another operating system
- For some embedded systems, a full-fledged operating system is unnecessarily expensive and complicated
- Examples: Ada 95, Java, Modula, Occam, ...

Example:

Ada 95 offers support via task, rendezvous & protected objects

Java offers support via threads & synchronized methods

Support for concurrent programming

Support via add-on run-time libraries:

- Suitable solution for languages that do not support concurrency as part of the language semantics (e.g. the C language)
- Some run-time libraries for operating systems are standardized, which makes program code easy to move between operating systems (e.g., POSIX for Linux, macOS, and Windows)

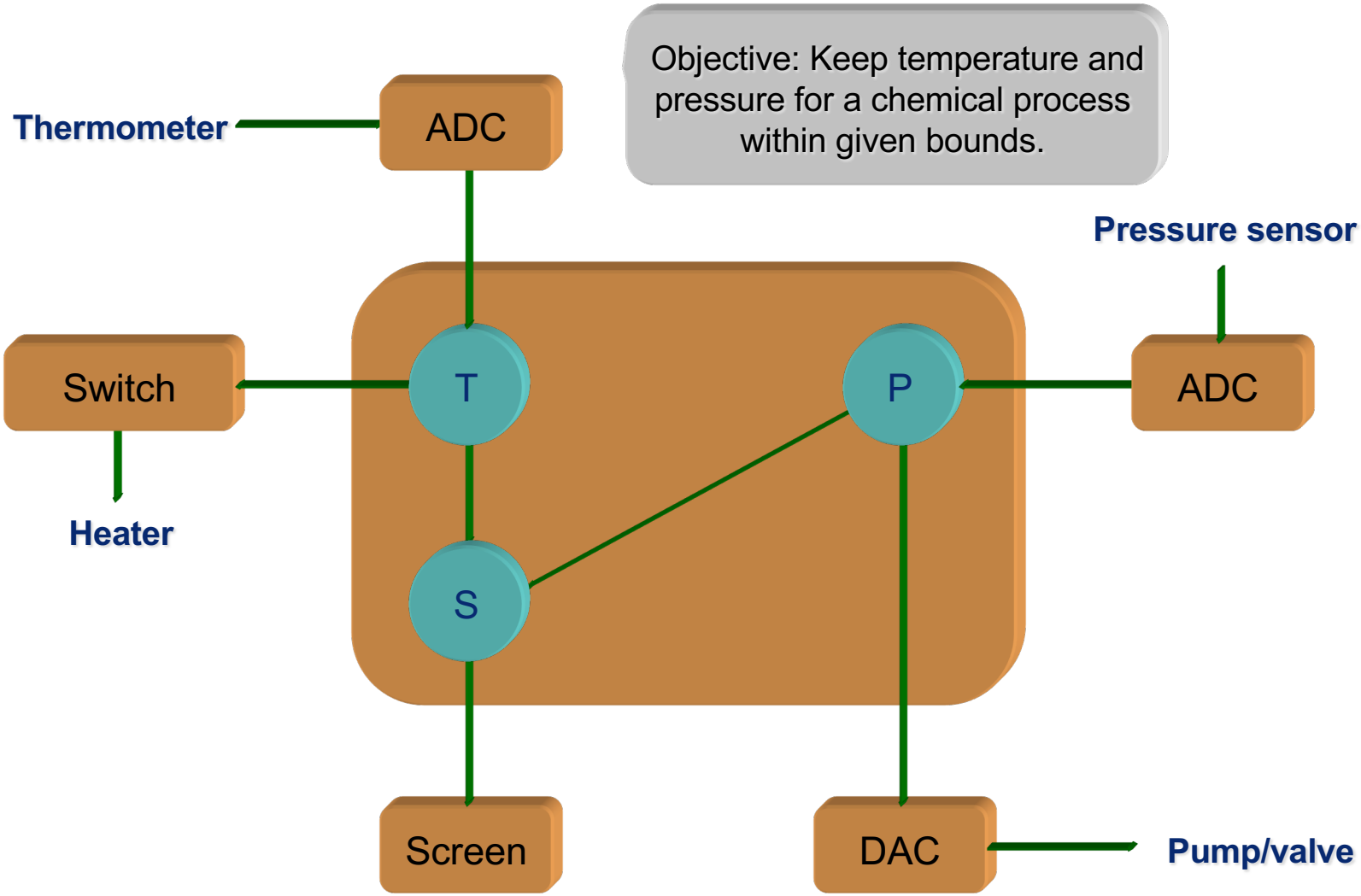
Example:

UNIX offers support via fork, semctl & msgctl

POSIX offers support via threads & mutex methods

TinyTimber offers support via reactive objects & mutex methods

Example: a simple control system



Sequential solution (Ada 95)

```
procedure Controller is
  TR : Integer;
  PR : Integer;
  HS : Integer;
  PS : Integer;
begin
  loop
    TR := T_Read;                      -- read temperature
    HS := Temp_Convert(TR);            -- convert to heater setting
    T_Write(HS);                       -- set heater switch
    PrintLine("Temperature: ", TR); -- write message on operator screen

    PR := P_Read;                      -- read pressure
    PS := Pressure_Convert(PR);        -- convert to pump setting
    P_Write(PS);                       -- set pump control
    PrintLine("Pressure: ", PR);    -- write message on operator screen
  end loop;
end Controller;
```

Sequential solution (Java)

```
public class Controller {
  public static void main(String[] args) {
    int TR;
    int PR;
    int HS;
    int PS;

    while (true) {
      TR = T_Read();                  // read temperature
      HS = Temp_Convert(TR);          // convert to heater setting
      T_Write(HS);                    // set heater switch
      PrintLine("Temperature: ", TR); // write message on operator screen

      PR = P_Read();                  // read pressure
      PS = Pressure_Convert(PR);      // convert to pump setting
      P_Write(PS);                    // set pump control
      PrintLine("Pressure: ", PR);    // write message on operator screen
    }
  }
}
```

Sequential solution

Drawback:

- the inherent parallelism of the application is not exploited
  - procedures `T_Read` and `P_Read` block the execution until a new temperature or pressure sample is available from the sensor
  - while waiting to read the temperature, no attention can be given to the pressure (and vice versa)
- the independence of the control functions are not considered
  - temperature and pressure must be read with the same interval
  - the iteration frequency of the loop is mainly determined by the blocking time of the calls to `T_Read` and `P_Read`.
  - if the call for reading the temperature does not return because of a fault, it is no longer possible to read the pressure

Improved sequential solution (Ada 95)

```
procedure Controller is
   ...
begin
   loop
      if Ready_Temp then
         TR := T_Read;
         HS := Temp_Convert(TR);
         T_Write(HS);
         PrintLine("Temperature: ", TR);
      end if;
      if Ready_Pres then
         PR := P_Read;
         PS := Pressure_Convert(PR);
         P_Write(PS);
         PrintLine("Pressure: ", PR);
      end if;
   end loop;
end Controller;
```

The Boolean function **Ready_Temp** indicates whether a sample from the sensor is available

The Boolean function **Ready_Pres** indicates whether a sample from the sensor is available

Improved sequential solution (Java)

```
public class Controller {
  public static void main(String[] args) {
    ...

    while (true) {
      if (Ready_Temp()) {
        TR = T_Read();
        HS = Temp_Convert(TR);
        T_Write(HS);
        PrintLine("Temperature: ", TR);
      }

      if (Ready_Pres()) {
        PR = P_Read();
        PS = Pressure_Convert(PR);
        P_Write(PS);
        PrintLine("Pressure: ", PR);
      }
    }
  }
}
```

The Boolean method **Ready_Temp** indicates whether a sample from the sensor is available

The Boolean method **Ready_Pres** indicates whether a sample from the sensor is available

Improved sequential solution

Advantages:

- the inherent parallelism of the application is exploited
  - pressure and temperature control do not block each other

Drawbacks:

- the program spends a large amount of time in “busy wait” loops
  - processor capacity is unnecessarily wasted
  - schedulability analysis is made complicated/impossible
- the independence of the control functions is not considered
  - temperature and pressure must be read with the same interval
  - if the call for reading the temperature does not return because of a fault, it is no longer possible to read the pressure

Concurrent solution

Step 1: Make concurrent:

- Partition the software into units of concurrency

  Ada 95:

  Create two units of type **task**, `T_Controller` and `P_Controller`, each containing the code for handling the data from respective sensor. The concurrent execution of the code will be automatically initiated.

  Java:

  First declare two classes, `T_Controller` and `P_Controller`, each class being a subclass of the predefined `Thread` class. Each class should provide a `run` method containing the code for handling the data from respective sensor.

  Then, create one thread object from each declared class. Finally, initiate the concurrent execution of the code by calling the predefined `start` method associated with each thread object.

Concurrent solution

Step 2: Make reactive:

- Tasks should be idle if there is no work to be done

  Ada 95:
  The task calls the blocking procedure `T_Read` or `P_Read` to idle.

  Java:
  The thread calls the blocking method `T_Read` or `P_Read` to idle.

- Activate task as a reaction to an incoming event

  Ada 95:
  The call to procedure `T_Read` or `P_Read` unblocks when data becomes available at a sensor, thus activating the calling task.

  Java: The call to method `T_Read` or `P_Read` unblocks when data becomes available at a sensor, thus activating the calling thread.

Concurrent solution (Ada 95)

```
procedure Controller is

task T_Controller;
task P_Controller;

task body T_Controller is
begin
  loop
     TR := T_Read;
     HS := Temp_Convert(TR);
     T_Write(HS);
     PrintLine("Temperature: ", TR);
  end loop;
end T_Controller;

task body P_Controller is
begin
  loop
     PR := P_Read;
     PS := Pressure_Convert(PR);
     P_Write(PS);
     PrintLine("Pressure: ", PR);
  end loop;
end P_Controller;

begin
  null;        -- begin concurrent execution of the two tasks
end Controller;
```

Reference only, not examined

Concurrent solution (Java)

```
public class T_Controller extends Thread {
  public void run() {
    while (true) {
      TR = T_Read();
      HS = Temp_Convert(TR);
      T_Write(HS);
      PrintLine("Temperature: ", TR);
    }
  }
}

public class P_Controller extends Thread {
  public void run() {
    while (true) {
      PR = P_Read();
      PS = Pressure_Convert(PR);
      P_Write(PS);
      PrintLine("Pressure: ", PR);
    }
  }
}
```

Reference only, not examined

Concurrent solution (Java)

```
public class Controller {
  public static void main(String[] args) {
    T_Controller TC = new T_Controller; // create temperature thread
    P_Controller PC = new P_Controller; // create pressure thread

    TC.start();       // begin concurrent execution of first thread
    PC.start();       // begin concurrent execution of second thread
  }
}
```

Concurrent solution

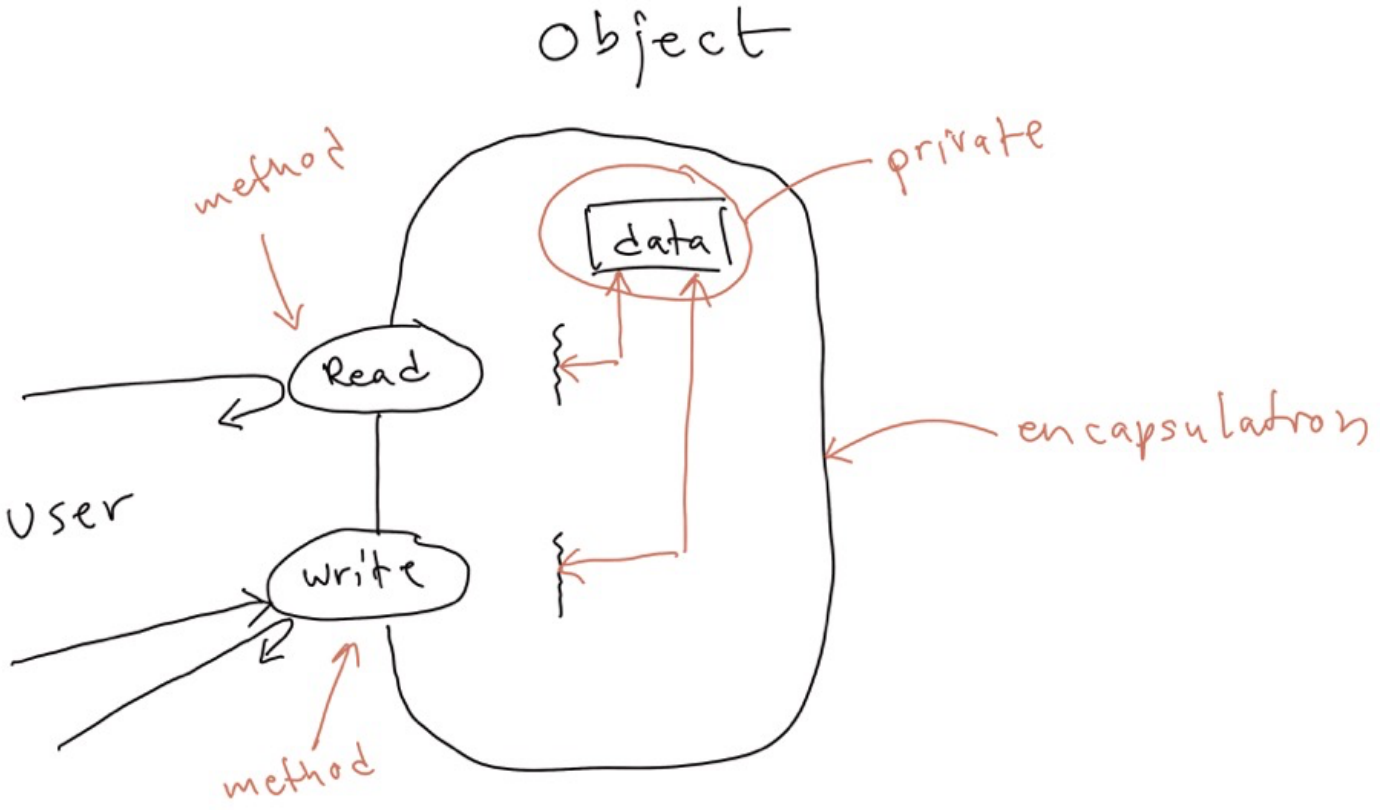
Advantages:

- the inherent parallelism of the application is fully exploited
  - pressure and temperature control do not block each other
  - the control functions can work at different frequencies
  - no processor capacity are unnecessarily consumed
  - the application becomes more reliable

Drawbacks:

- the parallel tasks share a common resource
  - the screen can only be used by one task at a time
  - a resource handler must be implemented, for controlling the access to the screen (to avoid garbled text)
  - the resource handler must guarantee *mutual exclusion* (*mutex*)

Lecture #3 – blackboard scribble



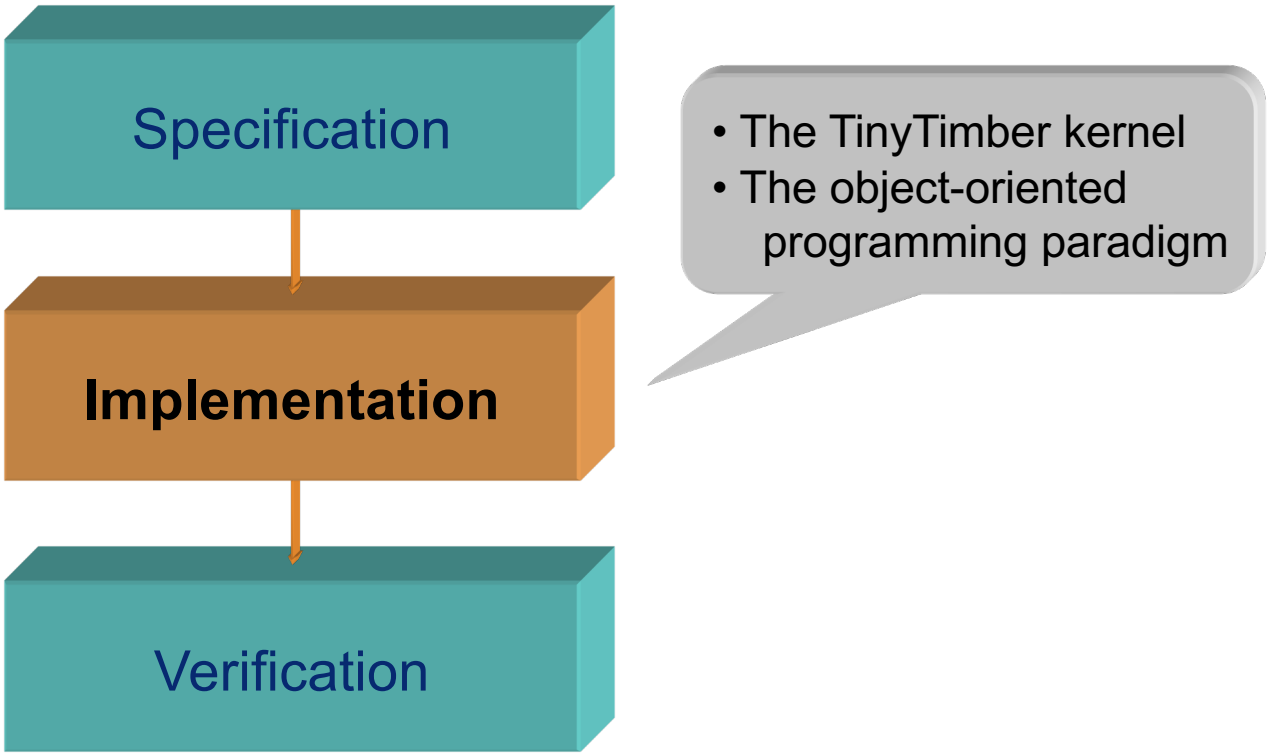
Real-Time Systems

Lecture #3

Victor Wallsten

Department of Computer Science and Engineering
Chalmers University of Technology

Real-time systems



TinyTimber – the context

Timber – the programming language:

- Full-featured language:
  - Higher-order and strongly typed language
  - Semantics in the purely functional tradition (non-lazy)
  - Features time-constrained concurrent reactive objects
- International research project:
  - Participants: Chalmers, Luleå University of Technology, Oregon Graduate Institute, Portland State University and Kansas State University (project period: 2000–2003).
- Direct descendant of O'Haskell:
  - An object-oriented extension of the Haskell language
  - Result of PhD work at Chalmers (J. Nordlander, 1999)

TinyTimber – the context

Timber – the programming language:

- Salient feature #1: “CORT” properties
  - Concurrent: Code execution concurrency is implicit, by means of the Timber object, and thus does not require the use of any dedicated concurrency constructs (e.g. threads or tasks).
  - Object oriented: Timber uses objects to encapsulate a local state and methods to manipulate this state.
  - Reactive: a Timber object is a passive entity, and the relative execution order of its methods is solely determined by events (e.g. hardware interrupts or invocations from other methods).
  - Timing aware: each reaction (method invocation) is associated with a programmable timing window, that can optionally be used to constrain start time and/or completion time of the reaction.

TinyTimber – the context

Timber – the programming language:

- Salient feature #2: “Deterministic” properties
  - Through its language design Timber code is free from indefinitely blocking language constructs. Thus, an object is always fully responsive when not actively executing code.

    This is in contrast to the common infinite event-loop pattern in other languages, where blocking calls are used to partition a linear thread of execution into event-handling fragments.
  - Through its language design Timber methods that belong to the same object are mutually exclusive. Consequently, object state is guaranteed to always be consistent.

TinyTimber – the context

Timber – the programming language:

- Recall the desired properties of a RT language:
  - Concurrent
  - Reactive
  - Timing aware
- What does the object-oriented (OO) approach offer?
  - Encapsulation (entity with data and code)
  - Reliability and maintainability (of object data and code)
  - Responsibility-driven design (clear roles assigned to objects)
  - Natural unit of concurrency (object with run-time context)
  - Natural place to implement mutual exclusion (“mutex”).

OO programming

Prominent features of OO programming:

- Encapsulation:
  - Encapsulation implies the existence of an entity that binds together data and a set of operations that manipulate the data.
  - The operations are referred to as methods.
  - The conventions for calling a method (e.g. parameter types, return value type) is referred to as the interface of the method.
  - The strongest form of encapsulation does not allow external code to directly access the data in the entity, but stipulates access through methods only. Thus, the data is for all practical purposes considered to be hidden from the external code.

OO programming

Prominent features of OO programming:

- Classes:
  - A detailed description of the internal format of encapsulated data (members) and set of methods (code) is called a class.
  - A class may be composed of members that refer to a class.
  - A class may be defined by extending the functionality of an existing class by means of class inheritance.
- Objects:
  - An instantiation of a class (i.e. memory storage is allocated to its members) is called an object.
  - The contents (values) of the object members is referred to as the state of the object.

OO programming

Advantages with OO programming:

- Provides reliability:
  - Encapsulation means that the object state is safe from outside (deliberate or unintentional) interference and misuse.
- Supports maintainability:
  - Encapsulation means that calling code does not need to be edited even if the internal format (data or code) of the class should change, as long as method interfaces stay the same.
- Supports responsibility-driven design:
  - Partitioning of program code into separate objects makes it easier to focus the design on the actions that each object is responsible for and the information that the object shares.

OO programming

Advantages with OO programming:

- Natural unit of concurrency :
  - The object state can be extended to include the run-time context (e.g. saved PC and SP registers) of its methods.
  - Methods belonging to different objects will thereby get independent run-time contexts, and could then execute concurrently on a run-time system that supports concurrency.
  - Offered by Java (thread objects) and Timber (reactive objects).

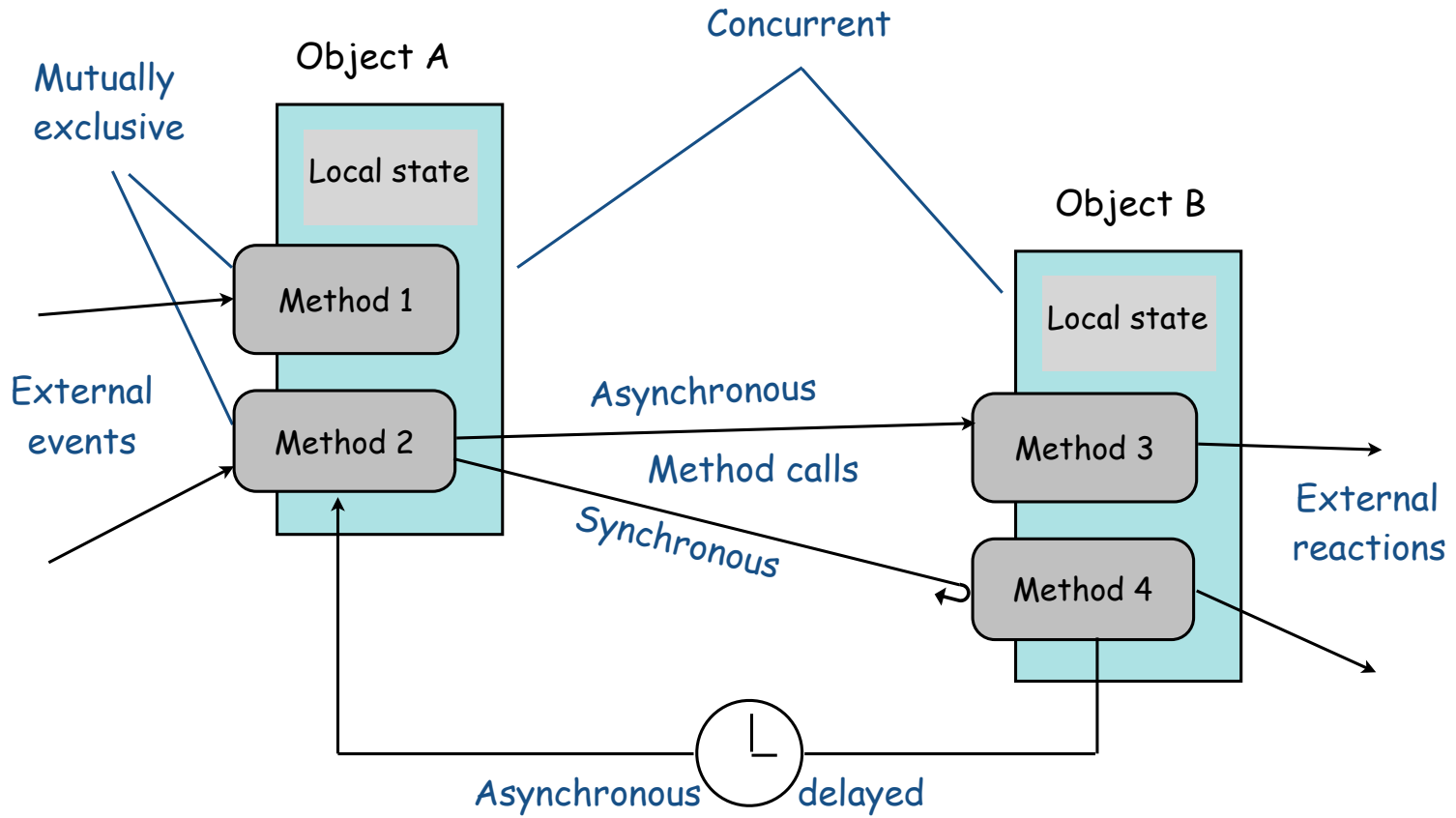
OO programming

Advantages with OO programming:

- Natural place to implement mutual exclusion:
  - Encapsulation facilitates “locking” the object while manipulating it via method calls, thereby guaranteeing state consistency.
  - Among an object’s methods only one may execute at a time, and must also complete its code before the object is unlocked (such code is commonly referred to as a “critical region”).
  - Methods belonging to the same object can therefore not execute concurrently (i.e., they are mutually exclusive)
  - Offered by Ada 95 (protected objects), Java (synchronized methods) and Timber (mutex methods).

TinyTimber – the context

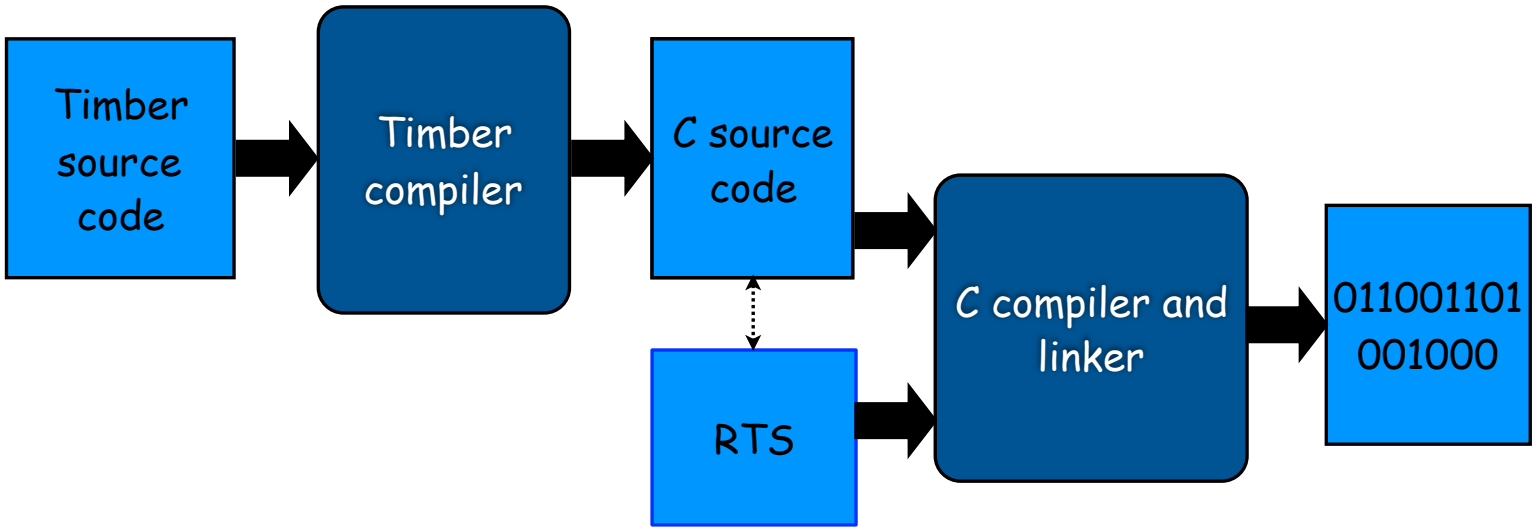
OO programming – visualized as an access graph:



An OO program is a collection of objects that act on each other via method calls

TinyTimber – the context

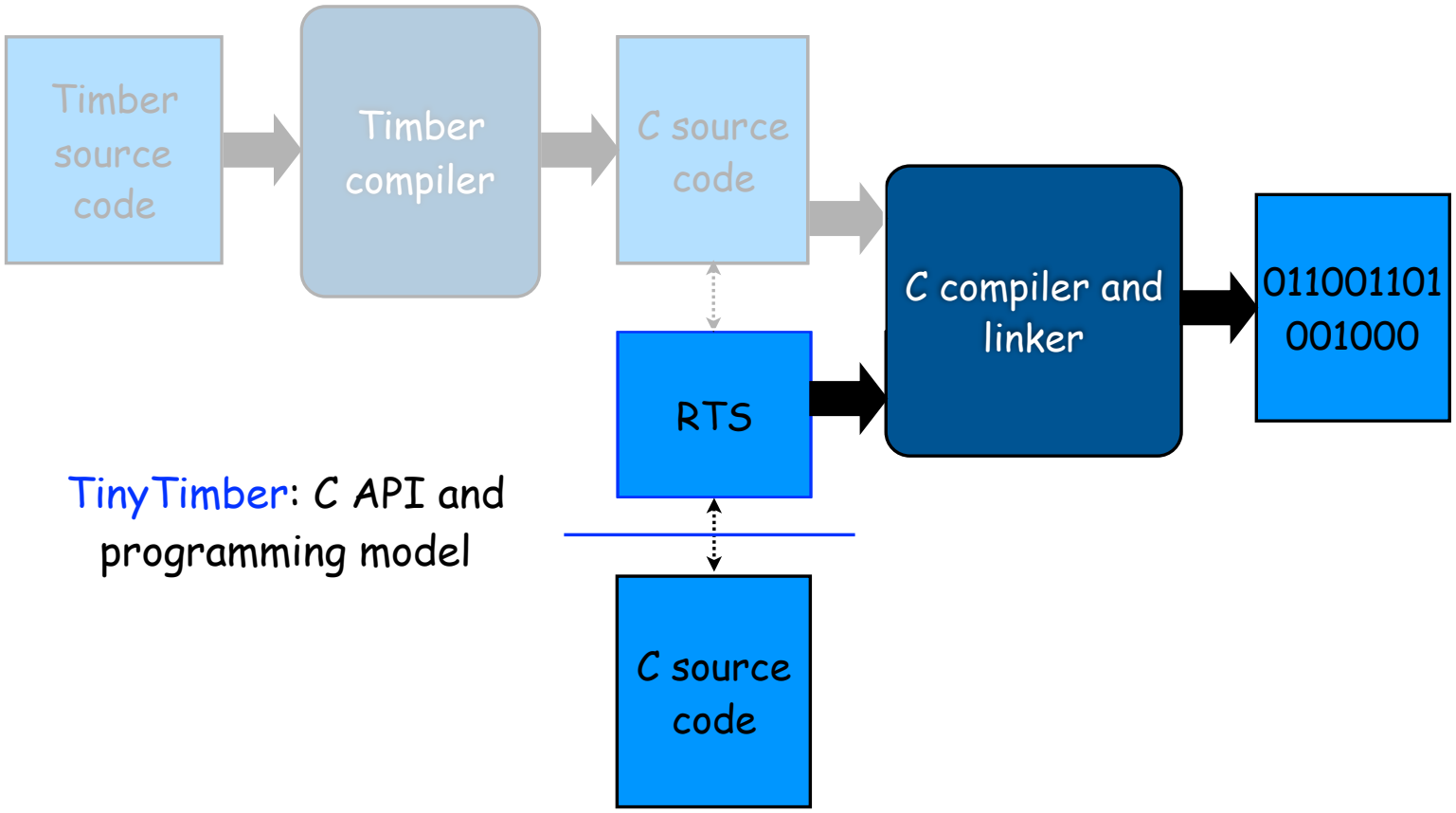
Timber – the compiler workflow:



Note: C language back-end

TinyTimber – the context

TinyTimber – the real-time kernel:



TinyTimber: C API and programming model

TinyTimber – the context

TinyTimber – the real-time kernel:

- Originates from the Timber run-time system:
  - TinyTimber was redesigned to make it a standalone run-time system, thereby not requiring the use of the Timber language.
  - TinyTimber is completely written in the C language.
    (apart for a few short parts written in assembly code)
- Retains the API of the Timber run-time system:
  - Caveat: requires special C coding conventions for the method interface.
- Fully supports the salient features #1 and #2 of Timber:
  - Caveat: requires special C coding conventions to retain these features of Timber.

TinyTimber – the context

TinyTimber – C coding conventions (OO programming):

- Use a C struct to define the members of a class:
  - The first member in the C struct must be of type `Object`, a predefined parent class containing run-time information.
- Use a C function to define a method:
  - The function must have two parameters, and a return type of either `int` or `void`.
  - The first parameter must be a pointer to the class to which the method belongs. The second parameter must be of type `int`.

  (Work-arounds for these restrictions will be given in Exercise #2)
- Use a variable of the C struct type to create an object:
  - The predefined `initObject()` macro should be used as a constructor for the first member in the class.

TinyTimber – the context

Example – C coding conventions (OO programming):

```
// TinyTimber class
typedef struct {
  Object super;   // NOTE: 'Object' type makes struct a TinyTimber class
  int theData;    // NOTE: 'theData' cannot be encapsulated (hidden)
} SharedInteger;

// TinyTimber methods
int Read(SharedInteger *self, int unused) {     // NOTE: methods are not
    return self->theData;                       //  part of the class ...
}
void Write(SharedInteger *self, int newValue) { // ... and therefore need
    self->theData = newValue;                   //  a pointer to the class
}

// TinyTimber object constructor (NOTE: not part of the class)
#define initSharedInteger(initialValue) { initObject(), initialValue }

SharedInteger myData = initSharedInteger(42);  // Create TinyTimber object
```

Reference only, not examined

TinyTimber – the context

Compare with Java implementation (OO programming):

```
// Java class
class SharedInteger
{
  private int theData;     // NOTE: 'theData' can be encapsulated (hidden)

  public SharedInteger(int initialValue) {    // Java object constructor
    theData = initialValue;                   // NOTE: part of the class
  }
  public synchronized int Read() {            // Java methods (mutex)
    return theData;                           // NOTE: part of the class
  }
  public synchronized void Write(int newValue) {
    theData = newValue;
  }
}

SharedInteger myData = new SharedInteger(42);      // Create Java object
```

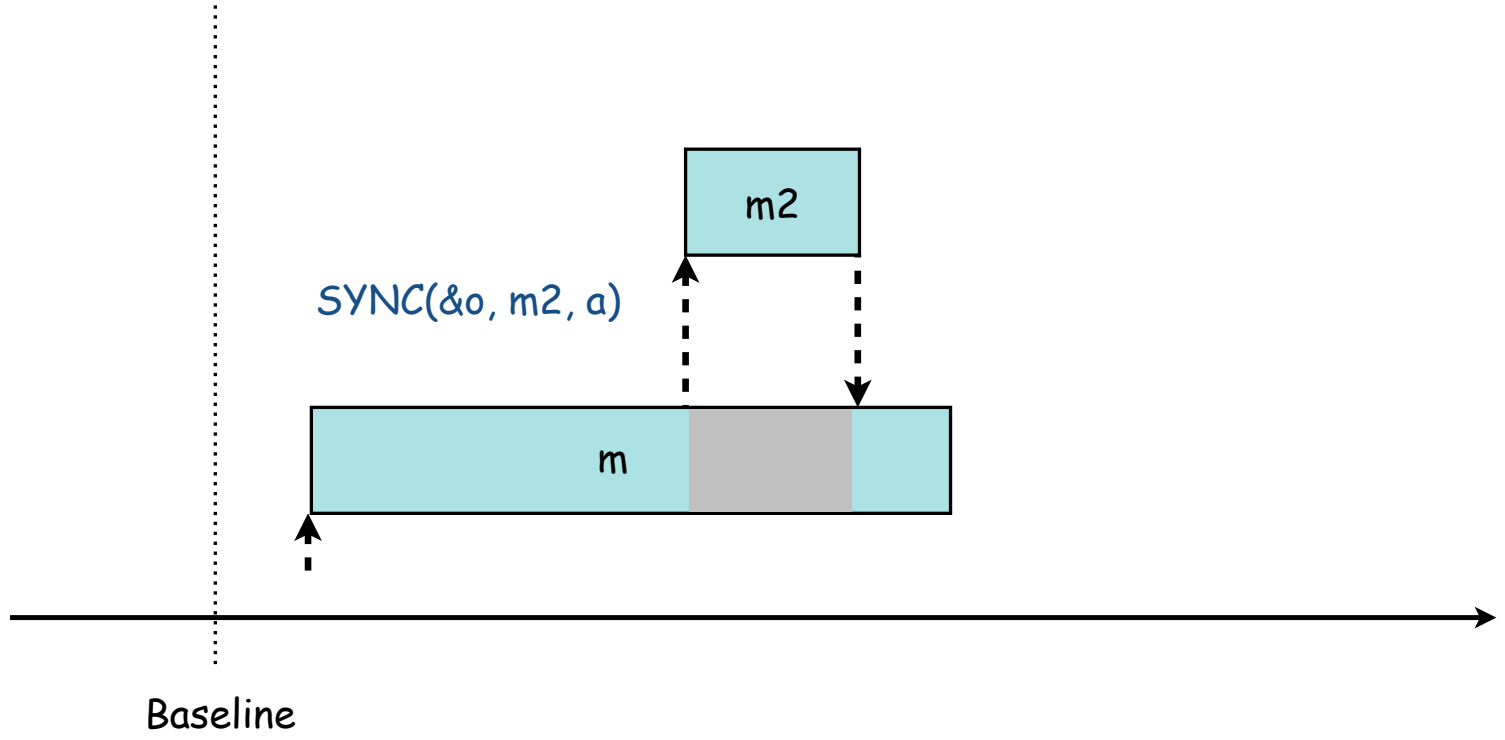
TinyTimber – the context

TinyTimber – C coding conventions (OO programming):

- Ensure encapsulation:
  - Access to class members should be done via methods calls only, even if the C language does not provide any mechanism for hiding the members in the corresponding C struct.
- Ensure determinism (mutex methods):
  - Mutex method calls are done synchronously or asynchronously.
  - A synchronous call is done via the predefined **`SYNC()`** macro; the calling code waits until the method call returns.
  - An asynchronous call is done via the predefined **`ASYNC()`** macro; this spawns a concurrent execution of the method code, and the calling code continues to execute (without waiting).

TinyTimber – the context

The SYNC() call – visualized as a timing diagram:

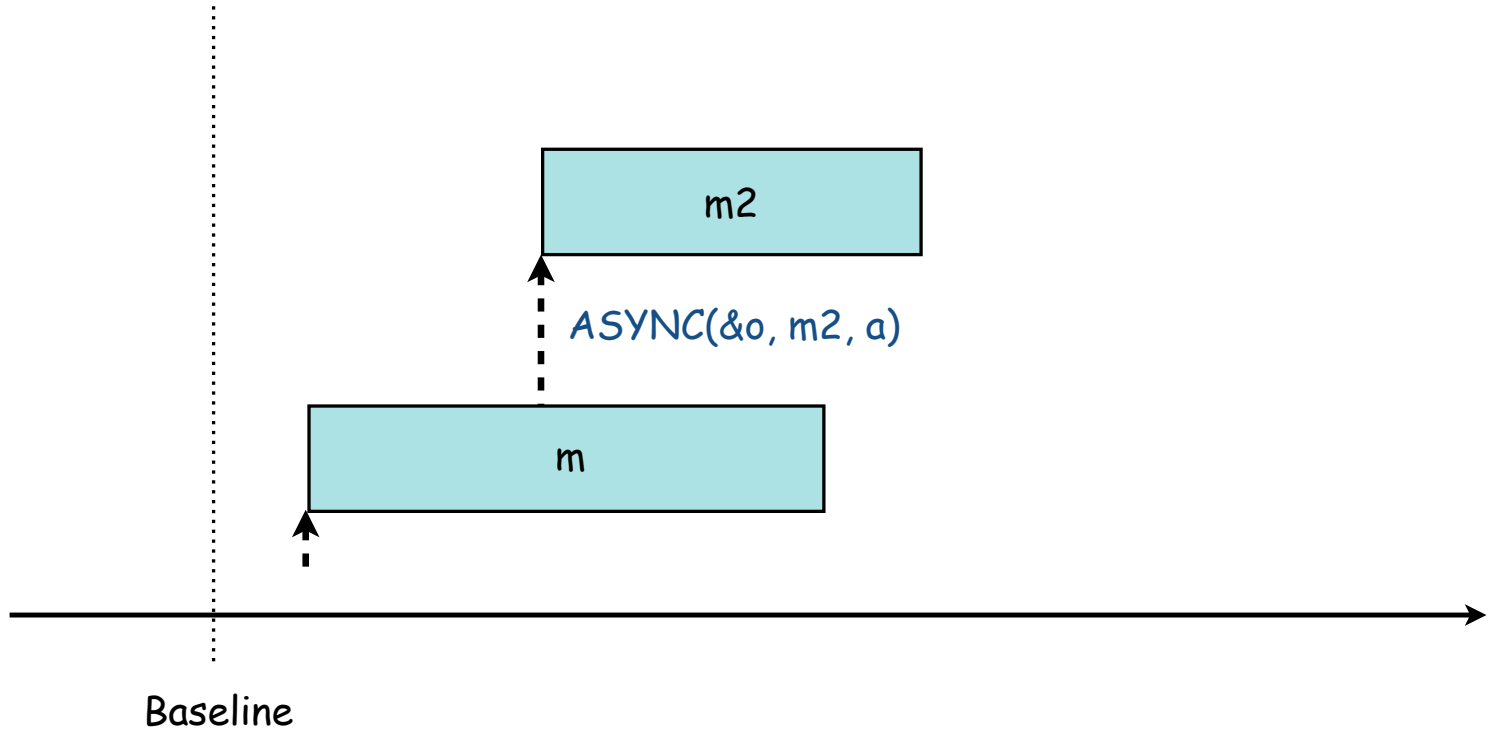


No concurrency: the caller cannot execute while the called method executes.

Consequently: the caller will always terminate after the called method.

TinyTimber – the context

The ASYNC() call – visualized as a timing diagram:



Concurrency: the caller and the called method could potentially execute in parallel

Observe: the caller may terminate before the called method (as in this example).

TinyTimber – the context

TinyTimber – C coding conventions (OO programming):

- Ensure determinism (non-blocking property):
  - The object methods cannot contain indefinitely blocking code
  - No-no #1: infinite loops (e.g. 'while (1)') must not be used
  - No-no #2: synchronous calls to a method within the same object as the calling code must not be used (as it will lead to deadlock)
- Ensure concurrency:
  - Program code that should execute concurrently must reside in methods that belong to separate objects.
  - Recall that code in methods that belong to the same object can never execute concurrently (due to mutex methods).

TinyTimber – the context

TinyTimber – C coding conventions (OO programming):

- Ensure timing awareness:
  - Programmable timing windows may be used for method calls by means of the following set of predefined macros.

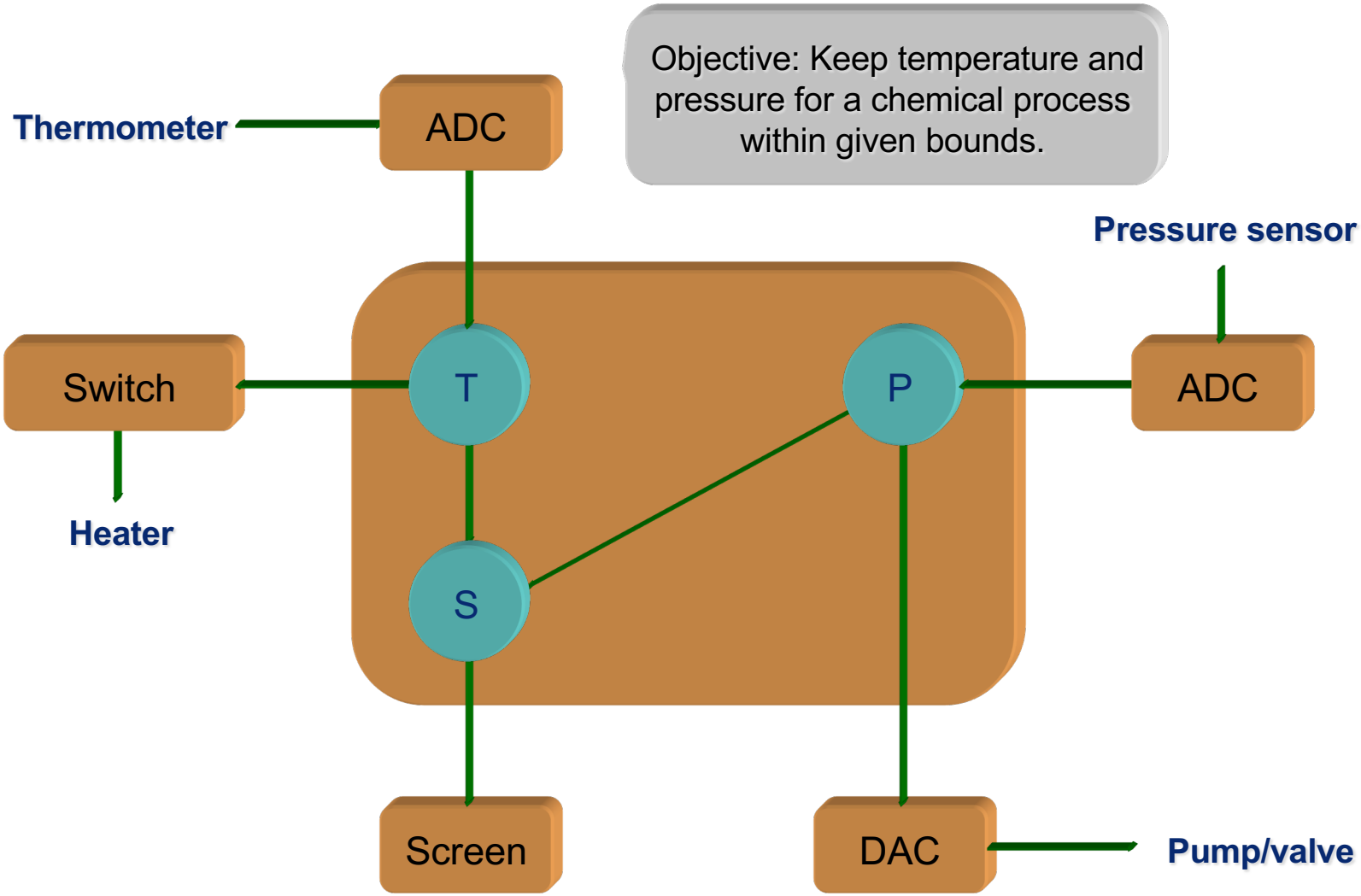
    `AFTER()`: corresponds to an `ASYNC()` call that takes place after an initial delay (offset).

    `BEFORE()`: corresponds to an `ASYNC()` call with a deadline on the method code completion.

    `SEND()`: equal to a combined `AFTER()` and `BEFORE()` call.

(More details regarding timing windows will be given in Lecture #6)

A simple control system (revisited)



Concurrent solution

Step 1: Make concurrent:

- Partition the software into units of concurrency

  TinyTimber:

  First declare a class `Task` with its first member being of predefined type `Object`, and define two methods associated with the declared class, `T_Controller` and `P_Controller`, containing the code for handling the data from respective sensor.

  Then, create two objects from the declared class, one for each of the defined methods. This will allow for concurrent execution of the code.

  Finally, create two interrupt handlers, one for each sensor, that will call the respective method when data becomes available on the sensor.

Concurrent solution

Step 2: Make reactive:

- Tasks should be idle if there is no work to be done

  TinyTimber:
  Since methods `T_Controller` and `P_Controller` must be called to be activated they are by default idle.

- Activate task as a reaction to an incoming event

  TinyTimber:
  An interrupt handler calls (activates) its respective method when data becomes available at a sensor.

  (More details on interrupt handlers and how to associate them with reactive objects will be given in Lecture #5)

Concurrent solution (TinyTimber)

```
typedef struct {
  Object super;      // NOTE: 'Object' type makes struct a TinyTimber class
} Task;

#define initTask() { initObject() }

Task T_Task = initTask();    // Create two new concurrent objects
Task P_Task = initTask();

// Declare the methods for each new object

void T_Controller(Task *, int);
void P_Controller(Task *, int);

// Define two new objects of class Sensor (definition not shown here),
// representing the sensors

Sensor sensor_t = initSensor(SENSOR_PORT0, &T_Task, T_Controller);
Sensor sensor_p = initSensor(SENSOR_PORT1, &P_Task, P_Controller);
```

Concurrent solution (TinyTimber)

```
// Define the methods for handling the input data. Each method is
// called with the data from the sensor as parameter.

void T_Controller(Task *self, int data) {
  int HS;

  HS = Temp_Convert(data);          // convert to heater setting
  T_Write(HS);                      // set heater switch
  PrintLine("Temperature: ", data); // write message on operator screen
}

void P_Controller(Task *self, int data) {
  int PS;

  PS = Pressure_Convert(data);      // convert to pump setting
  P_Write(PS);                      // set pump control
  PrintLine("Pressure: ", data);    // write message on operator screen
}

...
```

Concurrent solution (TinyTimber)

```
...

// Initialize the two sensor objects

void kickoff(Task *self, int unused) {
  SENSOR_INIT(&sensor_t);
  SENSOR_INIT(&sensor_p);
}

// Install interrupt handlers for the sensors, and then kick off
// the TinyTimber run-time system

void main() {
  INSTALL(&sensor_t, sensor_interrupt, SENSOR_INT0);
  INSTALL(&sensor_p, sensor_interrupt, SENSOR_INT1);
  TINYTIMBER(&P_Task, kickoff, 0);
}
```

Concurrent solution

Advantages:

- the inherent parallelism of the application is fully exploited
  - pressure and temperature control do not block each other
  - the control functions can work at different frequencies
  - no processor capacity are unnecessarily consumed
  - the application becomes more reliable

Drawbacks:

- the parallel tasks share a common resource
  - the screen can only be used by one task at a time
  - a resource handler must be implemented, for controlling the access to the screen (to avoid garbled text)
  - the resource handler must guarantee *mutual exclusion* (*mutex*)

Solid concurrent solution (TinyTimber)

```
/*
 * TinyTimber objects guarantee mutual exclusion for their declared
 * methods, if the caller uses a synchronous or asynchronous call:
 * the call to the method will then be blocked if any of the methods
 * in the object are currently being used.
 */

typedef struct {
  Object super;      // NOTE: 'Object' type makes struct a TinyTimber class
} ScreenController;

#define initScreenController() { initObject() }

ScreenController myScreen = initScreenController();   // Create new object

void T_Printline(ScreenController *self, int data) {  // NOTE: methods are
    PrintLine("Temperature: ", data);                 // not part of class
}

void P_Printline(ScreenController *self, int data) {
    PrintLine("Pressure: ", data);
}
```

Solid concurrent solution (TinyTimber)

```
/*
 * In a TinyTimber synchronous call the caller will be blocked
 * if any of the methods in the object are currently being used.
 */

void T_Controller(Task *self, int data) {
  Heater_Setting HS;

  HS = Temp_Convert(data);
  T_Write(HS);
  SYNC(&myScreen, T_PrintLine, data);       // NOTE: calling code waits
}                                           // for T_PrintLine to complete

void P_Controller(Task *self, int data) {
  Pressure_Setting PS;

  PS = Pressure_Convert(data);
  P_Write(PS);
  SYNC(&myScreen, P_PrintLine, data);       // NOTE: calling code waits
}                                           // for P_PrintLine to complete
```

Solid concurrent solution (TinyTimber)

```
/*
 * In a TinyTimber asynchronous call the caller can continue executing
 * immediately after posting the method call, regardless of whether
 * any of the methods in the object are currently being used or not.
 */

void T_Controller(Task *self, int data) {
  Heater_Setting HS;

  HS = Temp_Convert(data);
  T_Write(HS);
  ASYNC(&myScreen, T_PrintLine, data);      // NOTE: spawns a concurrent
}                                           // call to T_PrintLine

void P_Controller(Task *self, int data) {
  Pressure_Setting PS;

  PS = Pressure_Convert(data);
  P_Write(PS);
  ASYNC(&myScreen, P_PrintLine, data);      // NOTE: spawns a concurrent
}                                           // call to P_PrintLine
```

Solid concurrent solution (Ada 95)

```
-- In Ada95 protected objects can guarantee mutual exclusion for their
-- declared procedures: a calling task will be blocked if any of the
-- procedures in the object are currently being used.

protected type Screen_Controller is
  procedure T_Printline(data : Integer);
  procedure P_Printline(data : Integer);
end Screen_Controller;

protected body Screen_Controller is
begin
  procedure T_Printline(data : Integer) is
  begin
    Printline("Temperature: ", data);
  end T_Printline;
  procedure P_Printline(data : Integer) is
  begin
    Printline("Pressure: ", data);
  end P_Printline;
end Screen_Controller;

myScreen : Screen_Controller;              -- Create new object
```

Solid concurrent solution (Ada 95)

```
...

task body T_Controller is
begin
  loop
     TR := T_Read;
     HS := Temp_Convert(TR);
     T_Write(HS);
     myScreen.T_PrintLine(TR);
  end loop;
end T_Controller;

task body P_Controller is
begin
  loop
     PR := P_Read;
     PS := Pressure_Convert(PR);
     P_Write(PS);
     myScreen.P_PrintLine(PR);
  end loop;
end P_Controller;
```

Reference only, not examined

Solid concurrent solution (Java)

```
// Objects in Java can guarantee mutual exclusion if their methods are
// declared as synchronized: a call to the method will then be blocked
// if any of the methods in the object are currently being used.

class ScreenController
{
  public synchronized void T_Printline(int data) {
     Printline("Temperature: ", data);
  }
  public synchronized void P_Printline(int data) {
     Printline("Pressure: ", data);
  }
}

ScreenController myScreen = new ScreenController();   // Create new object
```

Solid concurrent solution (Java)

```
...

public class T_Controller extends Thread {
  public void run() {
    while (true) {
      TR = T_Read();
      HS = Temp_Convert(TR);
      T_Write(HS);
      myScreen.T_PrintLine(TR);
    }
  }
}

public class P_Controller extends Thread {
  public void run() {
    while (true) {
      PR = P_Read();
      PS = Pressure_Convert(PR);
      P_Write(PS);
      myScreen.P_PrintLine(PR);
    }
  }
}
```

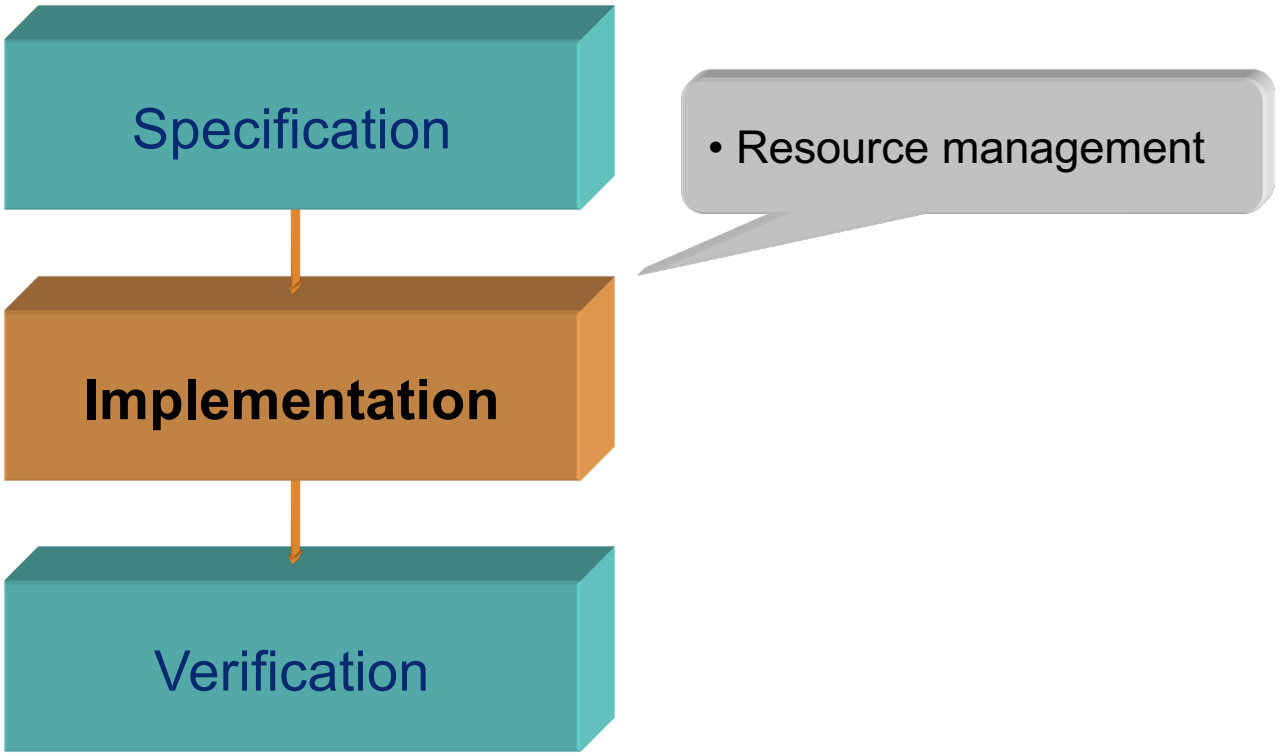
Real-Time Systems

Lecture #4

Professor Jan Jonsson

Department of Computer Science and Engineering
Chalmers University of Technology

Real-Time Systems



Resource management

Resource management is a general problem that exists at several levels in a real-time system.

- Shared resources internal to the the run-time system:
  - CPU time
  - Memory pool (for dynamic allocation of memory)
  - Data structures (queues, tables, buffers, …)
  - I/O device access (ports, status registers, …)
- Shared resources specific to the application program:
  - Data structures (buffers, state variables, databases…)
  - Displays (to avoid garbled text if multiple tasks use it)
  - Entities in the application environment (seats in a cinema or an aircraft, a car parking facility, etc)

Resource management

Classification of resources:

- Exclusive access: there must be only one user at a time.
  - Exclusiveness is guaranteed through mutual exclusion
  - Program code that is executed while mutual exclusion applies is called a critical region
  - Examples: manipulation of data structures or I/O device registers
- Shared access: there can be multiple users at a time.
  - Resource manager makes sure that the number of users are within acceptable limits
  - The program code for the resource manager is a critical region
  - Classic computer science example: Dining Philosophers problem

Resource management

Operations for resource management:

- `acquire`: to request access to a resource
- `release`: to release a previously acquired resource

The `acquire` operation can be either blocking or non-blocking:

- Blocking: the task that calls `acquire` is blocked if the resource is not available. Blocked tasks are stored in a queue, in FIFO or priority order. When the requested resource becomes available one of the blocked tasks is unblocked and is activated via a *call-back functionality*.
- Non-blocking: `acquire` returns a status code to the calling task indicating whether access to the resource was granted or not.

Resource management

Problems with resource management:

- Deadlock: tasks blocks each other and none of them can use the resource.
  - Deadlock can only occur if the tasks require access to more than one resource at the same time
  - Deadlock can be avoided by following certain guidelines
- Starvation: Some task is blocked because resources are always assigned to other (higher priority) tasks.
  - Starvation can occur in most resource management scenarios
  - Starvation can be avoided by granting access to resources in FIFO order

In general, deadlock and starvation are problems that must be solved by the program designer!

Resource management

Example #1: Assume that two tasks, A and B, want to use two different resources at the same time …

```
R1, R2 : Shared_Resource;

task A;
task body A is
begin
  R1.Acquire;
  R2.Acquire;
  ...          -- program code using both resources
  R2.Release;
  R1.Release;
end A;

task B;
task body B is
begin
  R2.Acquire;
  R1.Acquire;
  ...          -- program code using both resources
  R1.Release;
  R2.Release;
end B;
```

A task switch from A to B after this code line causes deadlock.

Resource management

Example #1: Assume that two tasks, A and B, want to use two different resources at the same time …

```
R1, R2 : Shared_Resource;

task A;
task body A is
begin
  R1.Acquire;
  R2.Acquire;
  ...          -- program code using both resources
  R2.Release;
  R1.Release;
end A;

task B;
task body B is
begin
  R1.Acquire;
  R2.Acquire;
  ...          -- program code using both resources
  R2.Release;
  R1.Release;
end B;
```

Deadlock can be avoided if the tasks acquire the resources in the same order.

Resource management

Example #2: Dining Philosophers problem …

- Five philosophers live together in a house.
- The house has one round dinner table with five plates of rice.
- There are five sticks available: one stick between every pair of plates.
- The philosophers alternate between eating and thinking. To be able to eat the rice, a philosopher needs two sticks.
- Sticks are a scarce resource: only two philosophers can eat at the same time.

How is deadlock and starvation avoided?

Resource management

Example #2: Dining Philosophers problem …

- The following solution will cause deadlock if all philosophers should happen to take the left stick at exactly the same time:

```
loop
  Think;
  Take_left_stick;
  Take_right_stick;
  Eat;
  Drop_left_stick;
  Drop_right_stick;
end loop;
```

- One way to avoid deadlock and starvation is to only allow four philosophers at the table at the same time.

Resource management

Example #3: A potential issue in our daily life …

Deadlock

Conditions for deadlock to occur:

1. Mutual exclusion
   - Only one task at a time can use a resource.
2. Hold and wait
   - There must be tasks that hold one resource at the same time as they request access to another resource.
3. No preemption
   - A resource can only be released by the task holding it.
4. Circular wait
   - There must exist a cyclic chain of tasks such that each task holds a resource that is requested by another task in the chain.

Deadlock

Guidelines for avoiding deadlock:

1. Tasks should, if possible, only use one resource at a time.
2. If (1) is not possible, all tasks should request resources in the same order.
3. If (1) and (2) are not possible, special precautions should be taken to avoid deadlock.

   For example: a task could request an exclusive resource using a non-blocking call; then a special status code could be returned by the resource handler in the run-time system to notify the task that a deadlock situation has been detected.

Resource management

Program constructs for resource management:

- Ada 95 uses protected objects.
- Older languages (e.g. Concurrent Pascal, Modula) use monitors.
- Java uses reentrant locks (can be used to build e.g. monitors) or synchronized methods.

When programming in languages (e.g. C and C++) that do not provide the constructs mentioned above, mechanisms provided by the real-time kernel or operating system must be used.

- POSIX offers semaphores and methods with mutual exclusion.
- The TinyTimber kernel offers methods with mutual exclusion.

  To allow TinyTimber to support general `acquire` and `release` operations a suitable object type (e.g. monitor or semaphore) must be added to the kernel.

Protected objects

Protected objects:

- A protected object is a synchronization mechanism offered by Ada 95.
- A protected object offers operations with mutual exclusion for data being shared by multiple tasks.
- A protected operation can be an entry, a procedure or a function. The latter is a read-only operation.
- Protected entries are guarded by a Boolean expression called a barrier.

  The barrier must evaluate to ”true” to allow the entry body code to be executed. If the barrier evaluates to “false”, the calling task will block until the barrier condition changes.

Protected objects

Implementing an exclusive resource in Ada 95:

```
protected type Exclusive_Resource is
  entry Acquire;
  procedure Release;
private
  Busy : Boolean := false;
end Exclusive_Resource;

protected body Exclusive_Resource is
  entry Acquire when not Busy is
  begin
    Busy := true;
  end Acquire;

  procedure Release is
  begin
    Busy := false;
  end Release;
end Exclusive_Resource;
...
```

Reference only, not examined

Protected objects

```
R : Exclusive_Resource;          -- resource with one user

task A, B;          -- tasks using the resource

task body A is
begin
   ...
   R.Acquire;
   ...              -- critical region with code using the resource
   R.Release;
   ...
end A;

task body B is
begin
   ...
   R.Acquire;
   ...              -- critical region with code using the resource
   R.Release;
   ...
end B;
```

Monitors

Monitors:

- A monitor is a synchronization mechanism originally offered by some older languages (e.g., Concurrent Pascal, Modula.)
- A monitor encapsulates data structures that are shared among multiple tasks and provides procedures to be called when a task needs to access the data structures.
- Execution of monitor procedures are done under mutual exclusion.
- Synchronization of tasks is done with a mechanism called condition variable. Each such variable represents a given Boolean condition for which the tasks should synchronize.

Monitors

Monitors vs. protected objects:

- Monitors are similar to protected objects in the sense that both are objects that can guarantee mutual exclusion during calls to procedures manipulating shared data.
- The difference between monitors and protected objects are in the way they handle synchronization:
  - Protected objects use entries with barriers (auto wake-up)
  - Monitors use condition variables (manual wake-up)

Java offers a class that facilitates creation of monitors:

The `ReentrantLock` class includes support for mutual exclusion and the possibility to create `Condition` objects, that directly correspond to the monitor's condition variables.

Monitors

Operations on condition variables:

`wait(c_var)`: the calling task is blocked and is inserted into a queue corresponding to condition `c_var`.

`signal(c_var)`: wake up first task in the queue corresponding to condition `c_var`. No effect if the queue is empty.

Properties:

1. After a call to `wait` the monitor is released (e.g., other tasks may execute the monitor procedures).
2. A call to `signal` must be located after the procedure code that manipulates the internal state of the monitor.

Reference only, not examined

Monitors

Implementing an exclusive resource with Java monitor:

```
class Exclusive_Resource {
  private boolean Busy;
  private final Lock lock = new ReentrantLock();
  private final Condition notBusy = lock.newCondition();
  public Exclusive_Resource() {
    Busy = false;
  }
  ...
```

Reference only, not examined

Monitors

```
  public void Acquire() {
    lock.lock();
    try {
      while (Busy)
        notBusy.await();      // block the task if resource busy

      Busy = true;
    } finally {
      lock.unlock();
    }
  }

  public void Release() {
    lock.lock();

    Busy = false;             // manipulate internal state of monitor
    notBusy.signal();         // then wake up a blocked task (if one exists)

    lock.unlock();
  }
} // class Exclusive_Resource

...
```

Reference only, not examined

Monitors

```
Exclusive_Resource R = new Exclusive_Resource(); // resource with one user

public class A extends Thread {  // concurrent thread using the resource
  public void run() {
    ...
    R.Acquire();
    ...                  // critical region with code using the resource
    R.Release();
    ...
  }
}

public class B extends Thread {  // concurrent thread using the resource
  public void run() {
    ...
    R.Acquire();
    ...                  // critical region with code using the resource
    R.Release();
    ...
  }
}
```

Semaphores

Semaphores:

- A semaphore is a passive synchronization primitive that is used for protecting shared and exclusive resources.
- Synchronization is done using two operations, `wait` and `signal`. These operations are atomic (indivisible) and are themselves critical regions with mutual exclusion.

A semaphore is less powerful than a protected object or a monitor, but is still quite useful as it can implement resource handlers for both exclusive (single-user) and shared (multi-user) resources.

Semaphores

A semaphore `s` is an integer variable with value domain ≥ 0

Atomic operations on semaphores:

`Init(s,n):` assign `s` an initial value `n`

`Wait(s):`

```
if s > 0 then
  s := s - 1;
else
  ”block calling task”;
```

`Signal(s):`

```
if ”any task that has called Wait(s) is blocked”
then
  ”allow one such task to execute”;
else
  s := s + 1;
```

Semaphores

Implementing semaphores in Ada 95:

```
protected type Semaphore (InitialValue : Natural := 0) is
    entry Wait;
    procedure Signal;
private
    Value : Natural := InitialValue;
end Semaphore;

protected body Semaphore is
    entry Wait when Value > 0 is
    begin
      Value := Value - 1;
    end Wait;
    procedure Signal is
    begin
      Value := Value + 1;
    end Signal;
end Semaphore;

...
```

Reference only, not examined

Semaphores

```
R : Semaphore(1);      -- exclusive resource (only one user)

task A, B;             -- tasks using the resource

task body A is
begin
   ...
   R.Wait;
   ...                 -- critical region with code using the resource
   R.Signal;
   ...
end A;

task body B is
begin
   ...
   R.Wait;
   ...                 -- critical region with code using the resource
   R.Signal;
   ...
end B;
```

Semaphores

Implementing semaphores in Java:

```
class Semaphore {
  private int Value;
  private final Lock lock = new ReentrantLock();
  private final Condition notBusy = lock.newCondition();
  public Semaphore(int InitialValue) {
    Value = InitialValue;
  }
  ...
```

Reference only, not examined

Semaphores

```
...

public void Wait() {
  lock.lock();
  try {
    while (Value == 0)
      notBusy.await();      // block the task if resource busy

    Value = Value - 1;
  } finally {
    lock.unlock();
  }
}

public void Signal() {
  lock.lock();

  Value = Value + 1;        // manipulate internal state of monitor
  notBusy.signal();         // then wake up a blocked task (if one exists)

  lock.unlock();
}
} // class Semaphore
```

Reference only, not examined

Semaphores

```
Semaphore R = new Semaphore(1);      // exclusive resource (only one user)
public class A extends Thread {  // concurrent thread using the resource
  public void run() {
    ...
    R.Wait();
    ...                  // critical region with code using the resource
    R.Signal();
    ...
  }
}
public class B extends Thread {  // concurrent thread using the resource
  public void run() {
    ...
    R.Wait();
    ...                  // critical region with code using the resource
    R.Signal();
    ...
  }
}
```

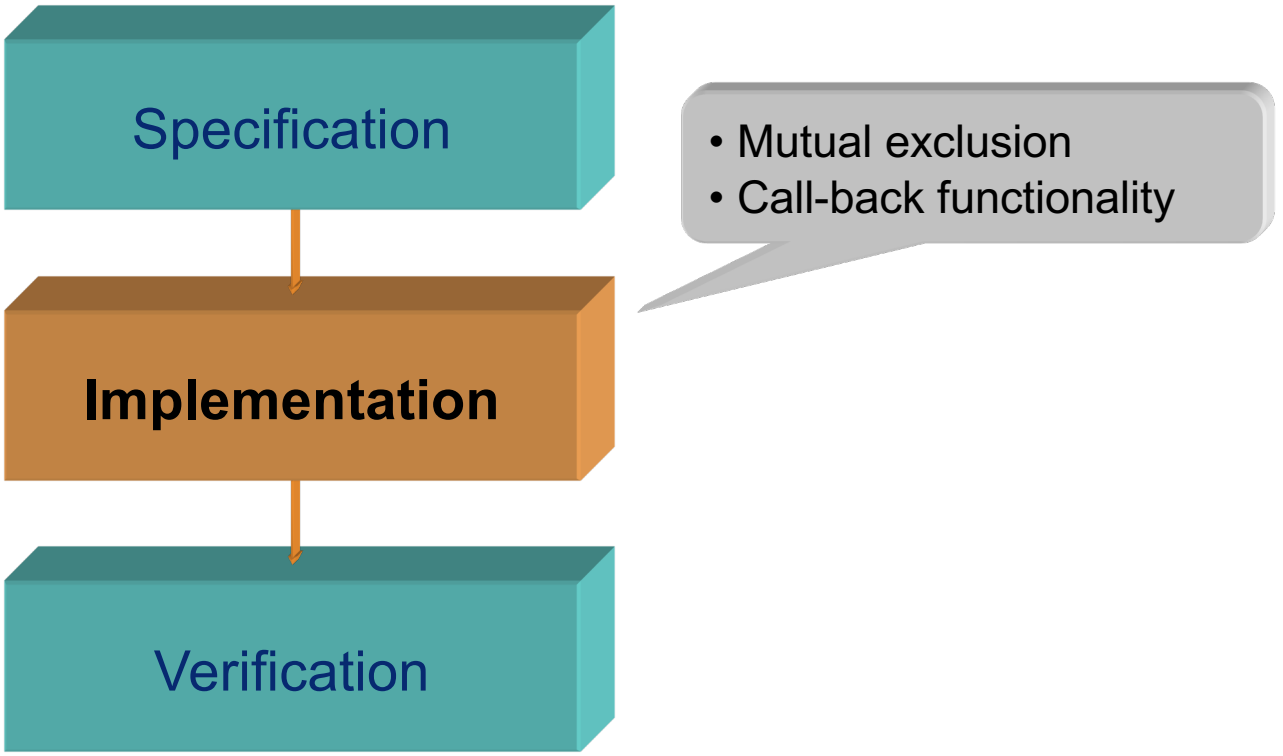
Real-Time Systems

Lecture #5

Professor Jan Jonsson

Department of Computer Science and Engineering
Chalmers University of Technology

Real-Time Systems



Mutual exclusion

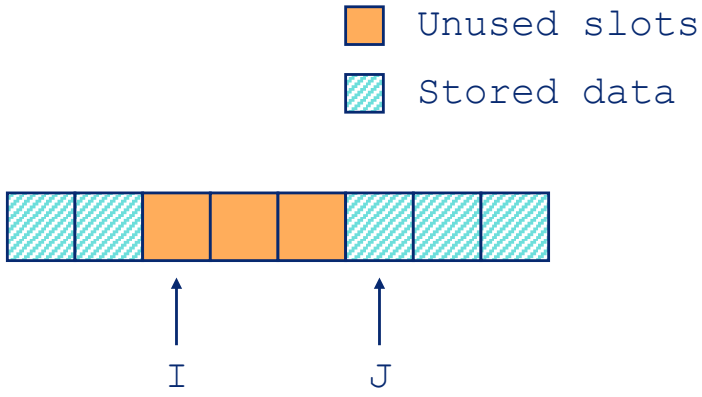
Systems with cooperating concurrent tasks often use shared data structures.

- A problem that has to be solved is then how to guarantee that the data structure is always kept in a consistent state.

  Data structures such as queues, lists and data bases will not work as intended if their state becomes inconsistent.
- A working solution is achieved if one makes sure that only one task at a time receive access to the data structure.
- Exclusive access to a data structure can be achieved by making sure that the program code (i.e., the critical region) that manipulates the data structure can execute without being preempted in the most critical moment.

Example: circular buffer in TinyTimber

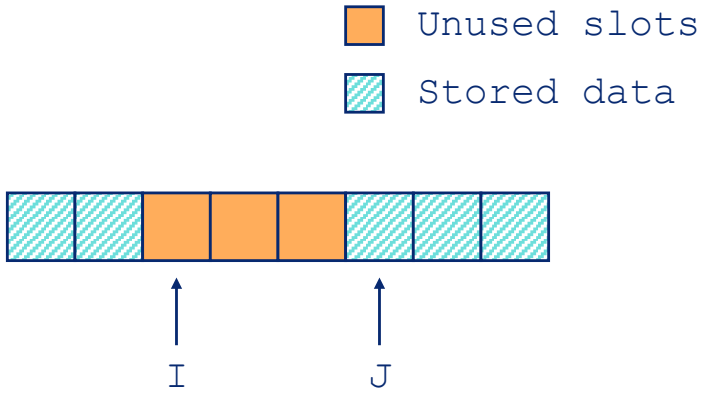
```
// Define a class Circular_Buffer with space for 8 natural numbers (≥ 0)

#define BSize  8

typedef struct {
    Object    super;
    int       count;
    int       I;
    int       J;
    int       A[BSize];
} Circular_Buffer;
```




```
// If the buffer is full, Put should return the value -1.
// If the buffer is empty, Get should return the value -1.

int Put(Circular_Buffer*, int); // Insert new element
int Get(Circular_Buffer*, int); // Remove old element

// Define an instance of the buffer

Circular_Buffer Buf = { initObject(), 0, 0, 0 };   // empty buffer
```

Example: circular buffer in TinyTimber

```
int Put(Circular_Buffer *self, int data) {
    if (self->count < BSize) {
        self->A[self->I] = data;
        self->I = (self->I + 1) % BSize;
        self->count = self->count + 1;
        return 0;
    }
    else
        return -1;
}

int Get(Circular_Buffer *self, int unused) {
    if (self->count > 0) {
        int data = self->A[self->J];
        self->J = (self->J + 1) % BSize;
        self->count = self->count - 1;
        return data;
    }
    else
        return -1;
}
```



Mutual exclusion

In TinyTimber the methods `Put` or `Get` must be called using `SYNC()` in order to guarantee mutual exclusion.

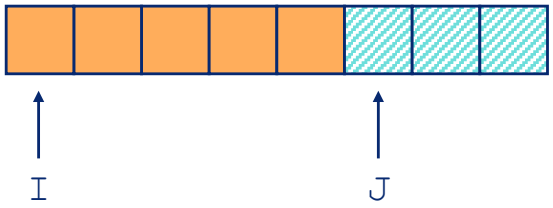
If `Put` or `Get` would be called as regular functions in C, mutual exclusion can not be guaranteed.

In the latter case, the buffer data structure could very easily become corrupt and give rise to data inconsistencies.

The following example demonstrates one such case ...

Mutual exclusion

Assume that the buffer has the following state:



Now, investigate what happens if `Put` is called as a regular C function by two concurrent tasks:

```
void T1(App *self, int c) {
    ...
    Put(&Buf,X);
    ...
    ASYNC(self,T1,c);
}
```

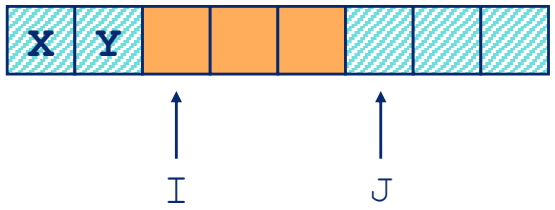
```
void T2(App *self, int c) {
    ...
    Put(&Buf,Y);
    ...
    ASYNC(self,T2,c);
}
```

Mutual exclusion

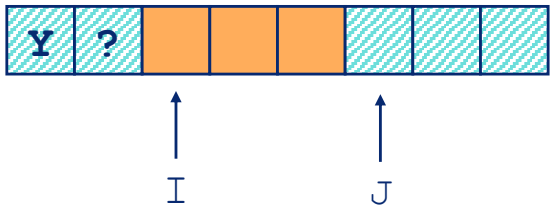
The following execution order causes data inconsistency:

```
Put(&Buf,X):                    Put(&Buf,Y):                    Comment:

 A(I) = X;
                                 A(I) = Y;                       // X is overwritten
                                 I = (I + 1) % BSize;
                                 count = count + 1;
I = (I + 1) % BSize;                                             // old value remains
count = count + 1;                                               // in last data slot
```

What we want is consistent data:

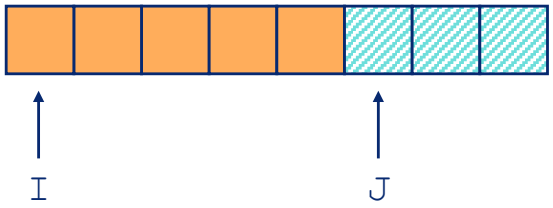


What we get is inconsistent data:



Mutual exclusion

Again, assume that the buffer has the following state:



This time observe the result when `Put` is called using `SYNC()` (synchronous method call) by the two concurrent tasks:

```
void T1(App *self, int c) {
    ...
    SYNC(&Buf,Put,X);
    ...
    ASYNC(self,T1,c);
}
```

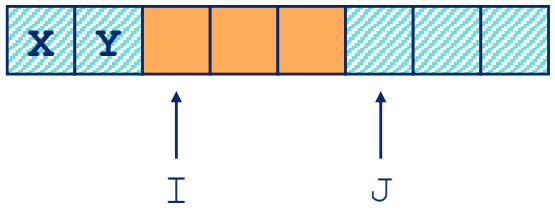
```
void T2(App *self, int c) {
    ...
    SYNC(&Buf,Put,Y);
    ...
    ASYNC(self,T2,c);
}
```

Mutual exclusion

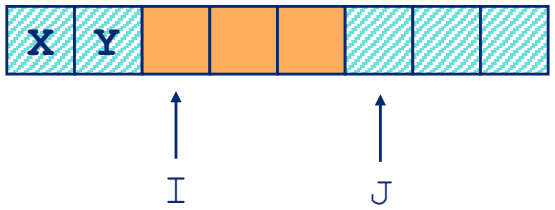
With `SYNC()` we get data consistency:

```
SYNC(&Buf,Put,X):          SYNC(&Buf,Put,Y):          Comment:

A(I) = X;                                             // data store is done
I = (I + 1) % BSize;                                  // correctly
count = count + 1;

                           A(I) = Y;                  // data store is done
                           I = (I + 1) % BSize;       // correctly
                           count = count + 1;
```

What we want is consistent data:



What we get is consistent data:



Machine-level mutual exclusion

To guarantee mutual exclusion in the critical regions of e.g. semaphore operations or mutex methods some even more fundamental support is needed.

For this purpose there are two mechanisms offered at the lowest (machine-code) level:

- Disabling the processor's interrupt service mechanism
  - Should involve any interrupt that may lead to a task switch
  - Only suitable for single-processor systems
- Atomic processor instructions

  For example: the test-and-set instruction
  - Variables can be tested and updated in one operation
  - Necessary for systems with two or more processors

Disabling processor interrupts

In single-processor systems, the mutual exclusion is guaranteed by disabling the processor's interrupt service mechanism ("interrupt masking") while the critical region is executed.

This way, unwanted task switches in the critical region (caused by e.g. timer interrupts) are avoided. However, all other tasks are unable to execute during this time.

Therefore, critical regions should only contain such instructions that really require mutual exclusion (e.g., code that handles the operations `wait` and `signal` for semaphores).

Note: this method is not used in multi-processor systems since interrupt management is typically not synchronized between the processors.

Disabling processor interrupts

```
task A;
task B;

task body A is
begin
  Disable_Interrupts;          -- turn off interrupt handling
  ...                          -- critical region
  Enable_Interrupts;           -- A leaves critical region
  ...                          -- remaining program code
end A;

task body B is
begin
  Disable_Interrupts;          -- turn off interrupt handling
  ...                          -- critical region
  Enable_Interrupts;           -- B leaves critical region
  ...                          -- remaining program code
end B;
```

Atomic processor instruction

In multi-processor systems with shared memory, a test-and-set instruction is used for handling critical regions.

A test-and-set instruction is a processor instruction that reads from and writes to a variable in one atomic operation.

The functionality of the test-and-set instruction can be illustrated by the following Ada procedure:

```
procedure testandset(lock, previous : in out Boolean) is
 begin
   previous := lock;          -- lock is read and its value saved
   lock := true;              -- lock is set to "true"
 end testandset;
```

The combined read and write of `lock` must be atomic. In a multi-processor system, this is guaranteed by locking (disabling access to) the memory bus during the entire operation.

Reference only, not examined

Atomic processor instruction

```
lock : Boolean := false;                -- shared flag
task A, B;

task body A is
  previous : Boolean;
begin
  loop
    testandset(lock, previous);         -- A waits if critical region is busy
   exit when not previous;
  end loop;
  ...                                   -- critical region
  lock := false;                        -- A leaves critical region
  ...                                   -- remaining program code
end A;

task body B is
  previous : Boolean;
begin
  loop
    testandset(lock, previous);         -- B waits if critical region is busy
   exit when not previous;
  end loop;
  ...                                   -- critical region
  lock := false;                        -- B leaves critical region
  ...                                   -- remaining program code
end B;
```

Reference only, not examined

Atomic processor instruction

```
lock : Boolean := false;                -- shared flag
task A, B;

task body A is
  previous : Boolean;
begin
  loop
    testandset(lock, previous);         -- A waits if critical region is busy
   exit when not previous;
  end loop;
  ...                                   -- critical region
  lock := false;                        -- A leaves critical region
  ...                                   -- remaining program code
end A;

task body B is
  previous : Boolean;
begin
  loop
    testandset(lock, previous);         -- B waits if critical region is busy
   exit when not previous;
  end loop;
  ...                                   -- critical region
  lock := false;                        -- B leaves critical region
  ...                                   -- remaining program code
end B;
```

Call-back functionality

Operations for resource management:

- `acquire`: to request access to a resource
- `release`: to release a previously acquired resource

The `acquire` operation can be either blocking or non-blocking:

- Blocking: the task that calls `acquire` is blocked if the resource is not available. Blocked tasks are stored in a queue, in FIFO or priority order. When the requested resource becomes available one of the blocked tasks is unblocked and is activated via a *call-back functionality*.
- Non-blocking: `acquire` returns a status code to the calling task indicating whether access to the resource was granted or not.

To support the reactive programming paradigm (that is, no "busy waiting" code) we should use the blocking approach.

Call-back functionality

Protected objects:

```
protected type Exclusive_Resource is
  entry Acquire;
  procedure Release;
private
  Busy : Boolean := false;
end Exclusive_Resource;

protected body Exclusive_Resource is
  entry Acquire when not Busy is
  begin
    Busy := true;
  end Acquire;

  procedure Release is
  begin
    Busy := false;
  end Release;
end Exclusive_Resource;
```

If task blocks here, call-back information must be saved in order to wake up the task later.

Call-back functionality

Monitors:

```
public void Acquire() {
  lock.lock();
  try {
    while (busy)
      notBusy.await();      // block the task if resource busy

    Busy = true;
  } finally {
    lock.unlock();
  }
}

public void Release() {
  lock.lock();

  Busy = false;             // manipulate internal state of monitor
  notBusy.signal();         // then wake up a blocked task (if one exists)

  lock.unlock();
}
} // class Exclusive_Resource
```

If task blocks here, call-back information must be saved in order to wake up the task later.

Call-back functionality

Semaphores:

```
protected type Semaphore (InitialValue : Natural := 0) is
    entry Wait;
    procedure Signal;
private
    Value : Natural := InitialValue;
end Semaphore;

protected body Semaphore is
    entry Wait when Value > 0 is
    begin
      Value := Value - 1;
    end Wait;

    procedure Signal is
    begin
      Value := Value + 1;
    end Signal;
end Semaphore;
```

If task blocks here, call-back information must be saved in order to wake up the task later.

Call-back functionality

Call-back information:

- As shown in the previous examples, the implementation of resource management mechanisms such as protected objects, monitors and semaphores make use of call-back information to be able to wake up a blocked task when the requested resource becomes available.
- Since multiple tasks may want to request access to a resource that is currently unavailable, call-back information for each of these tasks must be stored in a suitable data structure, e.g., a queue.

Call-back functionality

Call-back functionality in TinyTimber:

- TinyTimber has inherent method call blocking and call-back functionality, via the `SYNC()` call, in its implementation of an object (with its internal state) as an exclusive resource.
- However, TinyTimber cannot perform blocking or call back based on conditions relating to the contents of an object.

  If a generic `acquire/release` type of mechanism for shared resources, such as semaphores, is to be added to TinyTimber a separate call-back functionality must be implemented for that mechanism (= this week's exercise).
- TinyTimber also has call-back functionality in the *device drivers* for the serial port and CAN interfaces, in support of the reactive programming paradigm.

Call-back functionality

Device driver programming:

- A device driver is a software module that allows the user to interact with peripheral devices, such as serial ports or network interfaces, in a hardware-independent fashion.
- The device driver conceals the details in the cooperation between software and hardware by defining a set of operations on the device, e.g., *initialize*, *read*, and *write*.
- The device driver also contains *interrupt handler* code for any hardware event that may be associated with the device.
- If a user task is idle waiting for an event to happen on the device (e.g., data becomes available), the interrupt handler will require call-back information from the user of the device.

Interrupt handlers in TinyTimber

Guidelines for interrupt handling in TinyTimber:

- Interrupts must be handled using objects.
- An interrupt handler must be written as a method in the object.
- Data being processed by the interrupt handler must be stored in state variables in the object.
- Reading and writing such data from the user's program code must be done via synchronous calls to methods in the object, i.e., `SYNC()` calls.

We will now study the device driver for the serial port (SCI) in more detail.

Interrupt handlers in TinyTimber

Example: implementing an SCI interrupt handler:

1. Define class `Serial`, and add state variables for:
   - the hardware base address of the device
   - call-back information for a method if data received by the handler needs to be taken care of by the user-level code (the call back should be done using an `ASYNC()` call)
   - necessary local storage (buffers, queues, etc)
2. Define a symbol `SCI_PORT0` representing the hardware base address of the device.

```
#define SCI_PORT0 device_hardware_address
```

3. Create an object `sci0` of class `Serial`, and initialize it with:
   - the hardware base address `SCI_PORT0`
   - any possible call-back information

Interrupt handlers in TinyTimber

Example: implementing an SCI interrupt handler (cont'd):

In file 'application.c':

```
App app = { initObject(), 0, 'X' };

void reader(App*, int);

Serial sci0 = initSerial(SCI_PORT0, &app, reader);

void reader(App *self, int c) {    // call-back function
    SCI_WRITE(&sci0, "Rcv: \'");
    SCI_WRITECHAR(&sci0, c);
    SCI_WRITE(&sci0, "\'\n");
}
```

Interrupt handlers in TinyTimber

Example: implementing an SCI interrupt handler (cont'd):

4. Write an interrupt handler as a method `sci_interrupt` and associate it with the object.
5. Declare a symbol `SCI_IRQ0` and assign to it the TinyTimber kernel's logical number of the hardware interrupt:

```
#define SCI_IRQ0 interrupt_logical_number
```

6. Inform the TinyTimber kernel that the method is a handler for interrupt `SCI_IRQ0`, by making a call to

```
INSTALL(&sci0, sci_interrupt, SCI_IRQ0);
```

This should be done before the call to `TINYTIMBER()`

Interrupt handlers in TinyTimber

Example: implementing an SCI interrupt handler (cont'd):

7. Provide an operation `SCI_INIT()` that takes care of performing any remaining initialization of the device.
8. Call `SCI_INIT()` in the "kick-off" method that was supplied as argument to the `TINYTIMBER()` call.

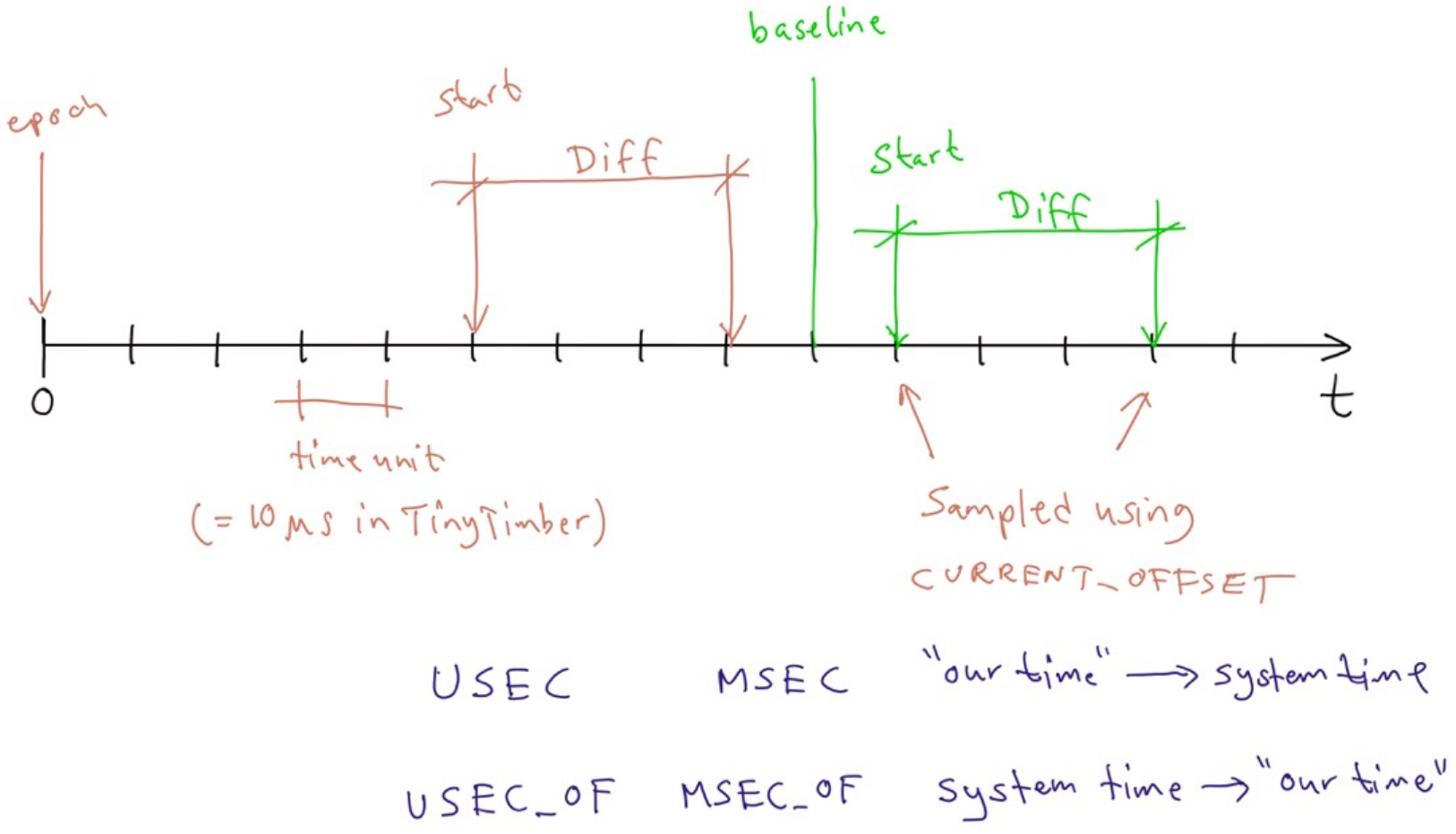
Interrupt handlers in TinyTimber

Example: implementing an SCI interrupt handler (cont'd):

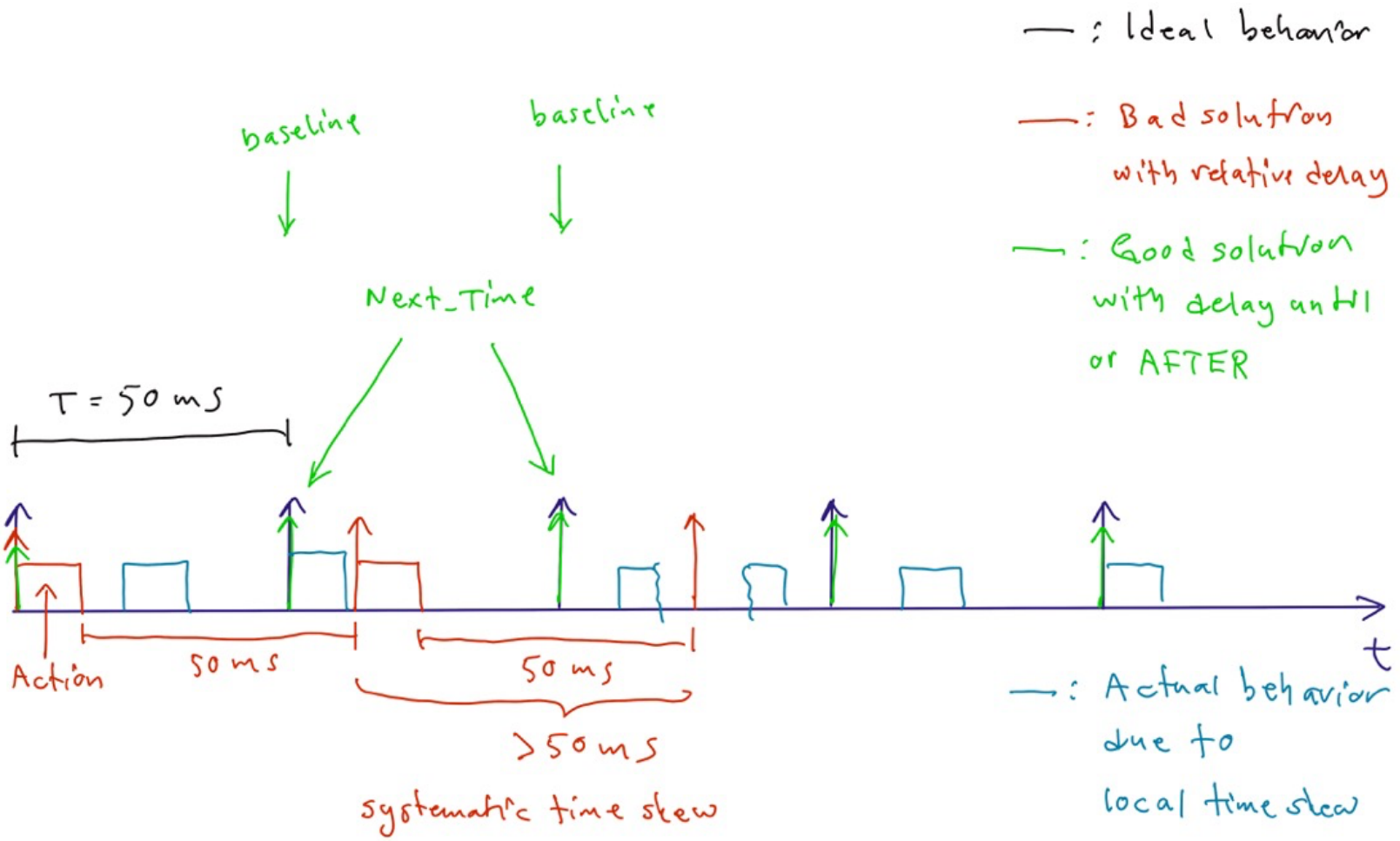
In file 'application.c':

```
void startApp(App *self, int arg) {
    SCI_INIT(&sci0);
    SCI_WRITE(&sci0, "Hello, hello...\n");
    …
}

int main() {
    INSTALL(&sci0, sci_interrupt, SCI_IRQ0);
    TINYTIMBER(&app, startApp, 0);
}
```

Lecture #6 – blackboard scribble



Lecture #6 – blackboard scribble



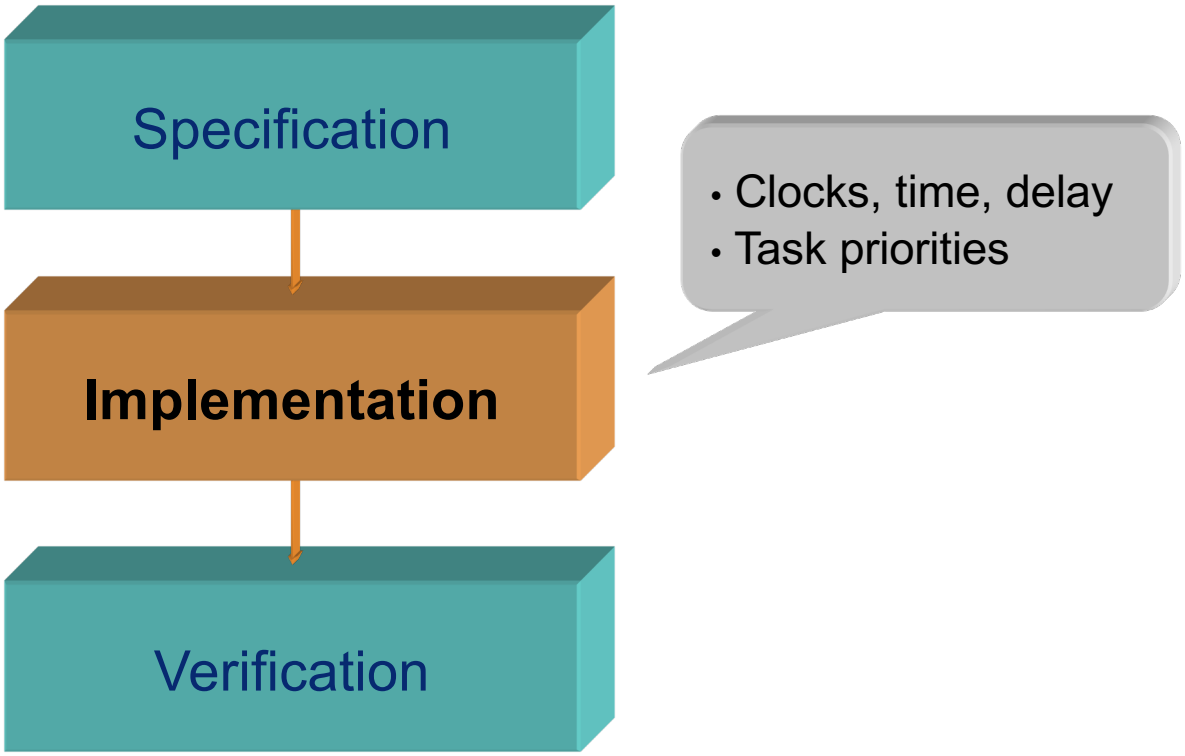
Real-Time Systems

Lecture #6

Professor Jan Jonsson

Department of Computer Science and Engineering
Chalmers University of Technology

Real-Time Systems



Recollection from an earlier lecture

Desired properties of a real-time programming language:

- Support for partitioning software into units of concurrency
  - tasks or threads (Ada 95, Java or POSIX C)
  - object methods (C/C++ using the TinyTimber kernel)
- Support for communication with the environment
  - access to I/O hardware (e.g. view I/O registers as variables)
  - machine-level data types (e.g. bit-field type, address pointers)
- Support for the schedulability analysis
  - notion of (high-resolution) time (⇒ timing-aware programming)
  - task priorities (reflects constraints ⇒ timing-aware programming)
  - task delays (idle while not doing useful work ⇒ reactive model)
  - hardware interrupt handlers (event generators ⇒ reactive model)

Clocks and time

To construct a real-time system, the chosen programming language or the run-time system must support a notion of (high-resolution) time that can be used for modeling the system's time constraints.

"Real-time" time is represented by a system clock, that can be read in order to report current time.

The system clock is typically implemented using a free-running timer, giving the following properties:

- Time is strictly monotonic (cannot be adjusted backwards)
- Time is measured in elapsed time units since an epoch.
- Time unit and epoch are both implementation dependent.

Real-time clocks in Ada 95

The Real-Time Systems annex in Ada 95 defines a data type `Time` that represents real time with a resolution of 1 ms or better. The current value of the real time can be read by calling the function `Clock`.

Convert human-perceived time to internal representation of time.

```
task body Controller is
  Start, Diff : Time;
  Limit: Time_Span := Milliseconds(17);
begin
  loop
    Start := Clock;
    ...                -- program code whose execution time is measured
    Diff := Clock - Start;
    if Diff > Limit then
        ...            -- program code for error handling
    end if;
  end loop;
end Controller;
```

Real-time clocks in TinyTimber

TinyTimber defines a data type `Time` that represents real time with a resolution of 10 µs for the MD407 card (lab system).

Method executions in TinyTimber have a baseline, which is a timestamp (of type Time) representing an earliest start time for the execution of the method.

- The baseline of a method is the baseline of its caller, except when a new explicit baseline is provided by the caller (using the `AFTER()` or `SEND()` operation.)
- The baseline of an interrupt-handler method is the time of the interrupt.

Real-time clocks in TinyTimber

TinyTimber defines a data type `Time` that represents real time with a resolution of 10 µs for the MD407 card (lab system).

Method executions in TinyTimber have a baseline, which is a timestamp (of type Time) representing an earliest start time for the execution of the method.

- A sample value of the real time can be read by calling the function `CURRENT_OFFSET()`, which returns the current time measured from the current baseline.
- The current baseline can be bookmarked by calling the function `T_RESET()` with an object of class `Timer`. The time duration from the bookmark to the baseline of a later event can then be calculated by calling the function `T_SAMPLE()` with the same object.

Real-time clocks in TinyTimber

```
void Controller(Object *self, int unused) {
    Time Start, Diff;
    Time Limit = MSEC(17);

    Start = CURRENT_OFFSET();
    ...                 // program code whose execution time is measured
    Diff = CURRENT_OFFSET() - Start;
    if (Diff > Limit) {
        ...             // program code for error handling
    }

    ASYNC(self, Controller, unused);
}
```

Convert human-perceived time to internal representation of time.

Macros for converting human-perceived time (s, ms, µs) to internal representation of time (and the other way around) are available in the file **"TinyTimber.h"** in the lab system source code package.

Periodic activities

The majority of embedded real-time applications rely on periodic activities, that is, tasks executing at regular intervals as part of e.g. a control loop.

Typically, control theory dictates the choice of execution interval for the periodic activities.

To support the reactive programming model, tasks should be idle while not doing useful work.

Therefore, there must exist support in the programming language or in the run-time system to delay (idle) the execution of a task until it is time for its next activation.

Periodic activities

How can the execution of a task be delayed in Ada 95?

- Use the (relative) `delay` statement:

```
delay 0.05;        -- wait for 0.05 seconds
```

- The `delay` statement guarantees that the task executing it will be idle at least the indicated number of seconds.

- The actual idle time could be longer because the re-activated task may have to wait for other tasks to complete their execution.

  The length of the actual idle time will then largely depend on the priority-assignment policy used in the run-time system.

Periodic activities

Example: Execute a task periodically every 50 milliseconds.

```
task body T is
    Interval : constant Duration := 0.05;
begin
  loop
    Action;                -- procedure doing useful work
    delay Interval;
  end loop;
end T;
```

Note: this solution gives rise to a systematic time skew

- The code for `Action` takes a certain time Δ_{action}
- The code for administrating the loop construct takes a certain time Δ_{loop}

⇒ The minimum interval between two executions of `Action` is:

$50 + \Delta_{action} + \Delta_{loop}$ milliseconds.

Periodic activities

How can systematic time skew be avoided in Ada 95?

- Use the (absolute) `delay` statement:

```
delay until Later;        -- wait until clock becomes Later
```

- The absolute `delay` statement causes the task executing to be idle until the given time instant at the earliest.

```
task body T is
  Interval : constant Duration := 0.05;
  Next_Time : Time;
begin
  Next_Time := Clock + Interval;
  loop
    Action;                          -- procedure doing useful work
    delay until Next_Time;
    Next_Time := Next_Time + Interval;
  end loop;
end T;
```

Periodic activities

How are periodic activities implemented in TinyTimber?

- Use the `AFTER()` operation:

```
AFTER(base_off, object, method, argument);
```

- The `AFTER()` operation guarantees that the specified method does not begin executing until time `baseline` at the earliest:

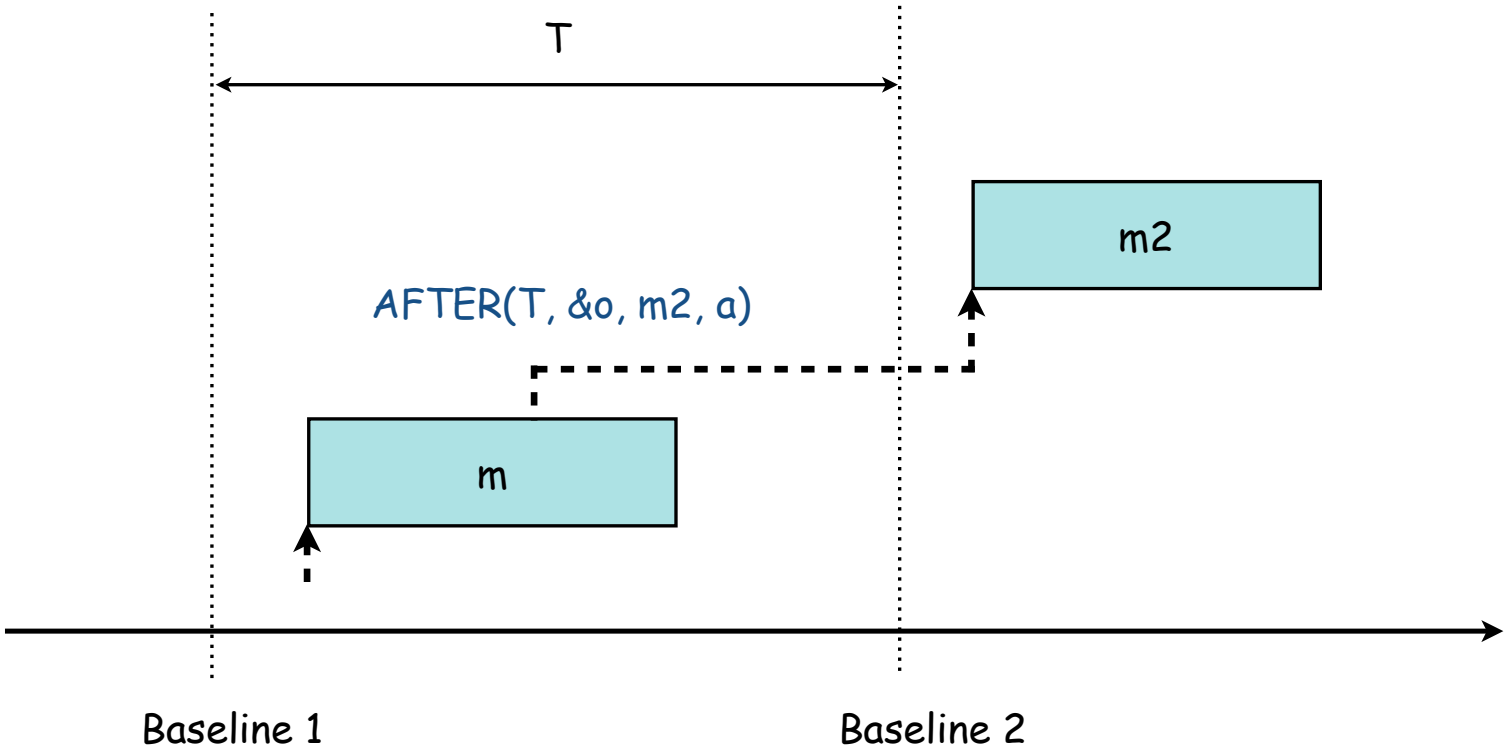
```
baseline = current_baseline + base_off
```

Here, `current_baseline` is the current baseline of the method posting the call with the `AFTER()` operation.

```
void T(Object *self, int unused) {
    Time Interval = MSEC(50);

    Action();                          // procedure doing useful work
    AFTER(Interval, self, T, unused);
}
```

Periodic activities

The AFTER() call – visualized as a timing diagram:



In this example, the AFTER() call defines a new earliest start time for a method.

Periodic activities

Note that both the `delay until` statement (in Ada95) and the `AFTER()` operation (in TinyTimber) may suffer from local time skew:

- Other active tasks/methods with same or higher priority may interfere so that the task/method cannot begin its execution at the desired time instant.
- In the case of periodic tasks/methods, the local time skew may vary between different activations of the same task/method.
- Local time skew can be reduced/eliminated by using suitable scheduling algorithms, or be determined with the aid of special analysis methods.

Task priorities

To be able to guarantee a predictable (and thereby analyzable) behavior of a real-time system, the programming language and run-time system must have support for task priorities.

Task priorities are used for selecting which task that should be executed if multiple tasks contend over the CPU resource.

In a real-time system, the priority should reflect the time-criticality of the task.

The priority of a task can be given in two different ways:

- Static priorities: based on task characteristics that are known before the system is running, e.g., iteration frequency or deadline.
- Dynamic priorities: based on task characteristics that are derived at certain times while the system is running, e.g., remaining execution time or remaining time to deadline.

Priority support in Ada 95

Ada 95 can use both static and dynamic priorities, although only static priorities are supported in the core language.

A static priority may be given to a task using the pragma `Priority`, which is placed in the task specification.

```
task P1 is
  pragma Priority(5);
end P1;
```

The Real-Time Systems annex of Ada 95 provides support for dynamic priorities:

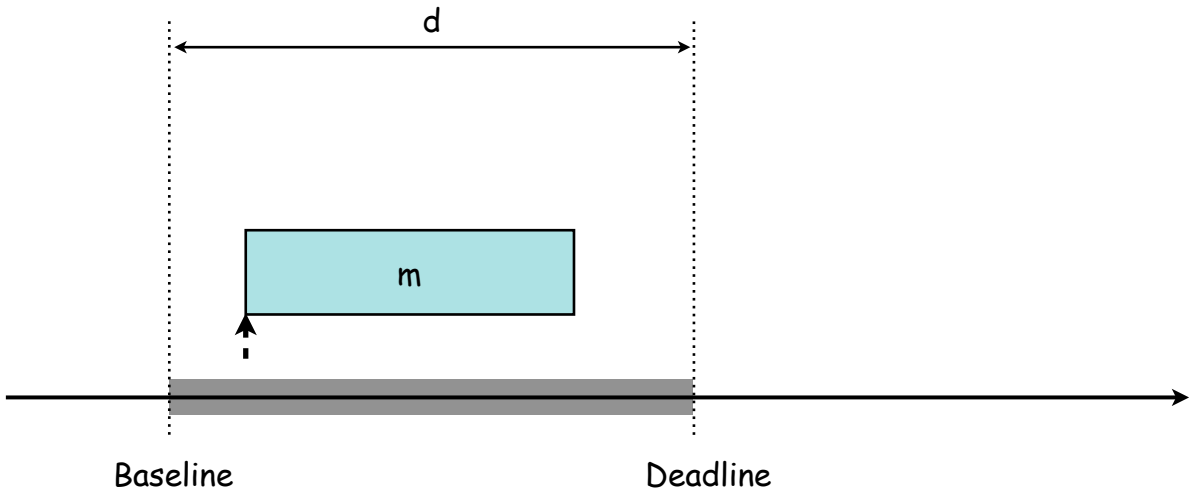
```
package Ada.Dynamic_Priorities is
  procedure Set_Priority(...);
  function Get_Priority(...) return Priority;
end Ada.Dynamic_Priorities;
```

Priority support in TinyTimber

TinyTimber uses dynamic priorities exclusively, by means of the earliest-deadline-first (EDF) priority-assignment policy:

"The method whose deadline is earliest in time receives highest priority"

- The existence of a baseline (earliest possible start time) and a deadline (latest allowable completion time) then defines a timing window for the execution of a TinyTimber method:



Priority support in TinyTimber

TinyTimber uses dynamic priorities exclusively, by means of the earliest-deadline-first (EDF) priority-assignment policy:

"The method whose deadline is earliest in time receives highest priority"

- Time-critical method calls can be done by means of the `BEFORE()` operation, which performs an asynchronous call with an explicit deadline:

```
BEFORE(rel_deadline, object, method, argument);
```

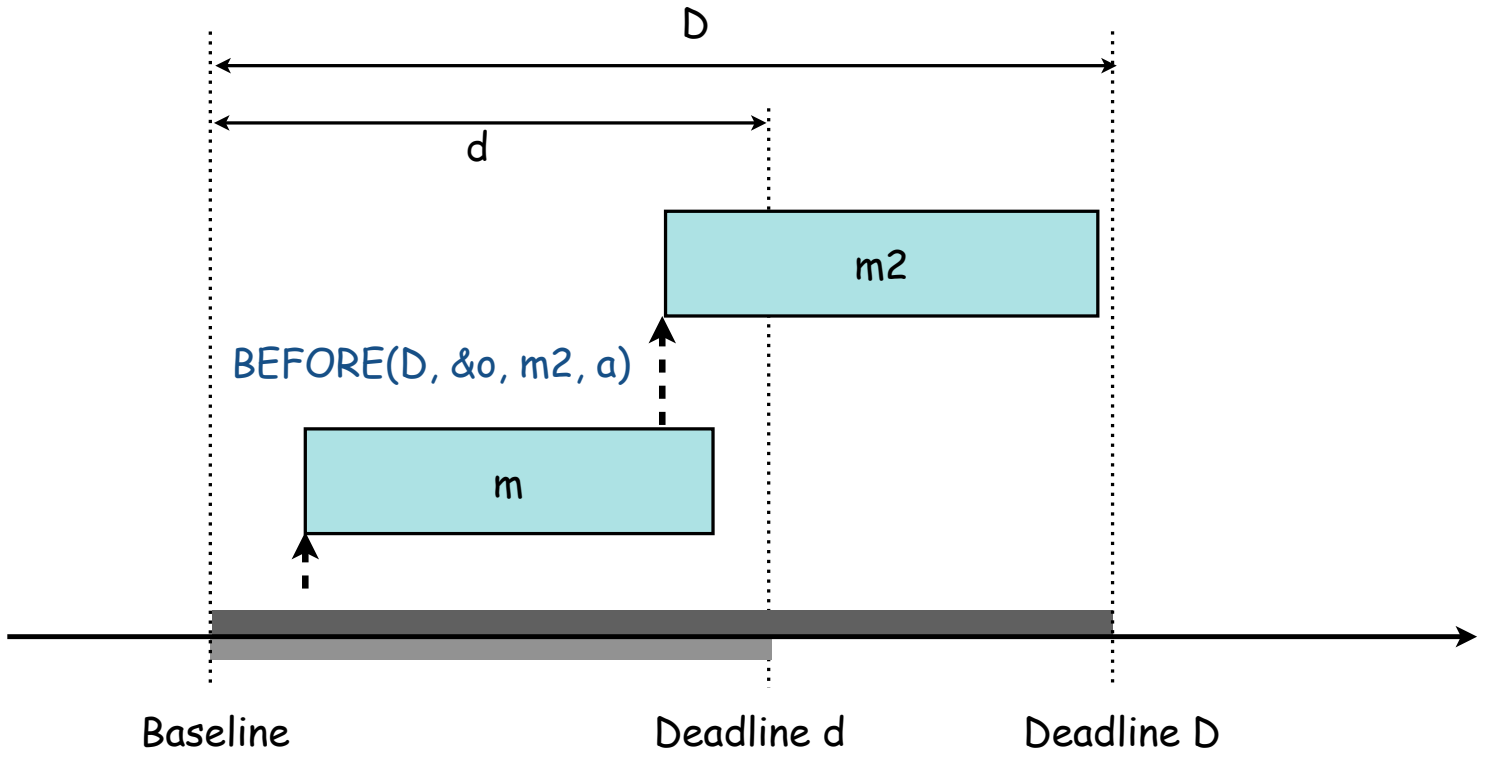
- The `BEFORE()` operation requests that the specified method should complete its execution by `deadline` at the latest:

```
deadline = current_baseline + rel_deadline
```

Here, `current_baseline` is the current baseline of the method posting the call with the `BEFORE()` operation.

Priority support in TinyTimber

The BEFORE() call – visualized as a timing diagram:



In this example, the BEFORE() call extends the timing window of a method.

Priority support in TinyTimber

TinyTimber uses dynamic priorities exclusively, by means of the earliest-deadline-first (EDF) priority-assignment policy:

"The method whose deadline is earliest in time receives highest priority"

- Time-critical method calls can also be done via the use of the `SEND()` operation, which performs an asynchronous call with a new baseline and an explicit deadline:

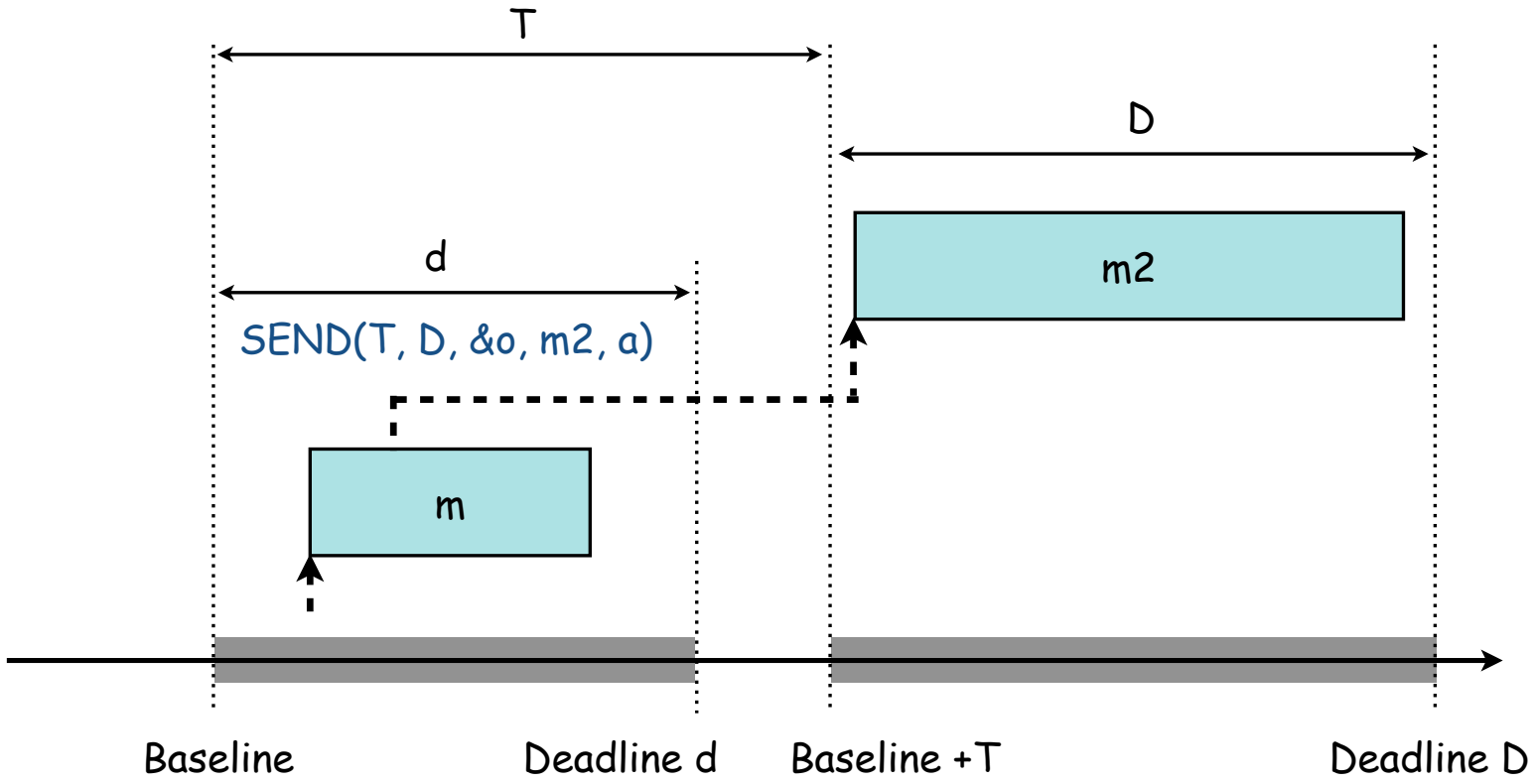
```
SEND(base_off, rel_deadline, object, method, argument);
```

- The `SEND()` operation requests that the specified method should begin its execution by `baseline` at the earliest and complete its execution by `deadline` at the latest:

```
baseline = current_baseline + base_off
deadline = baseline + rel_deadline
```

Priority support in TinyTimber

The SEND() call – visualized as a timing diagram:



In this example, the SEND() moves the entire timing window of a method.

Priorities and shared objects

When task priorities are used to introduce determinism and analyzability to the system, this must also encompass the handling of shared (mutex) objects.

Such analysis includes deriving an upper bound of each task's blocking time. This is relatively simple as long as a task can only be blocked by tasks with higher priority.

The analysis becomes more difficult when mutex objects are used, as a task can then also be blocked by tasks with lower priority that may not even use the object.

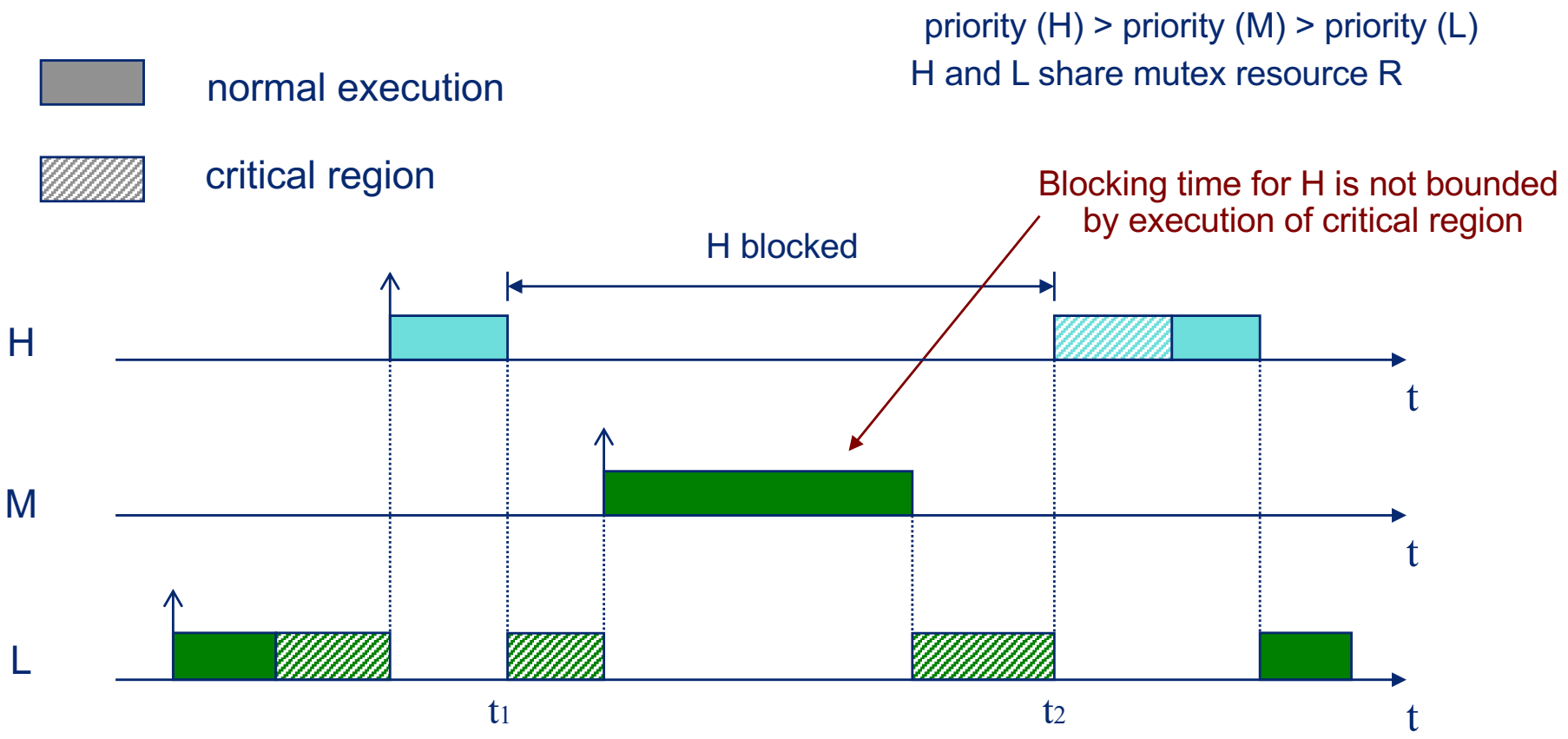
- If static priorities are used such a scenario is referred to as priority inversion.
- If dynamic priorities are used such a scenario is referred to as deadline inversion.

Priority inversion

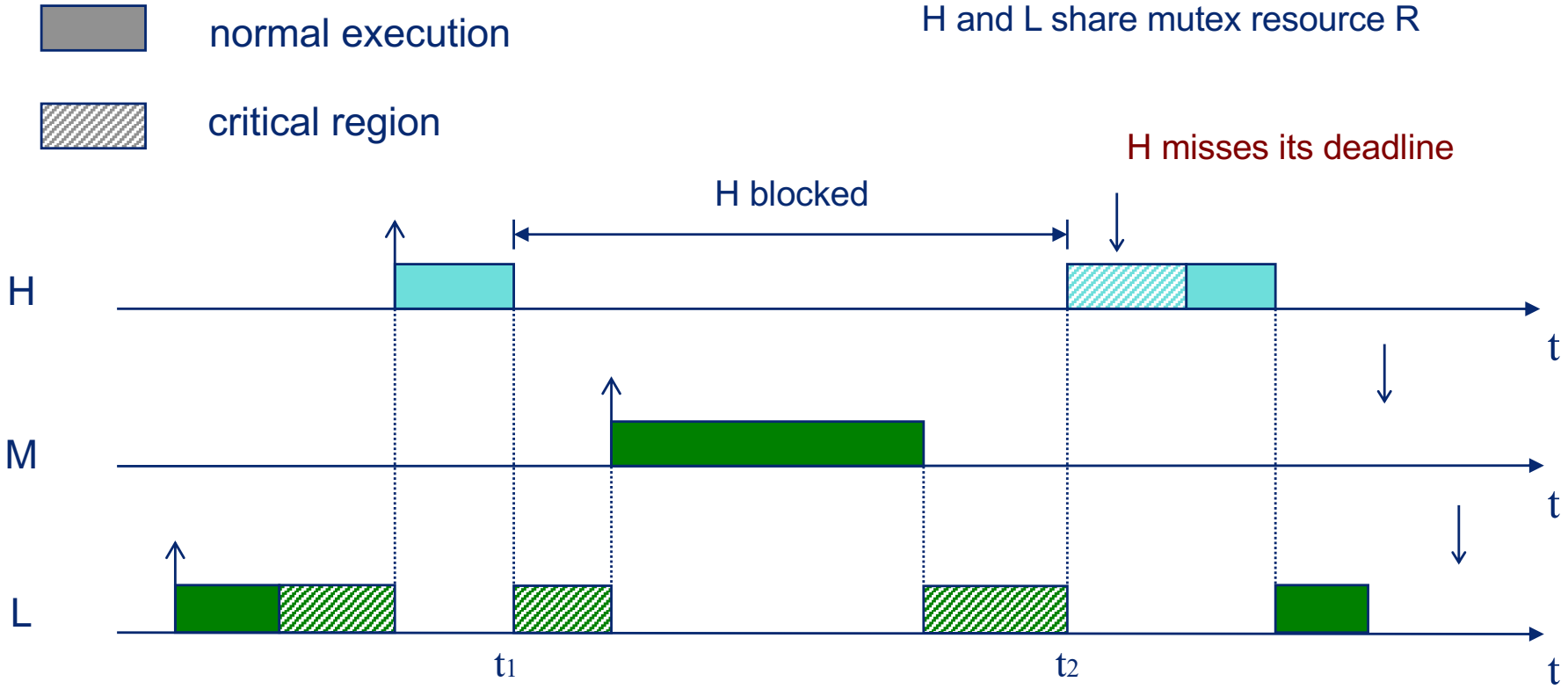
Assume three tasks H, M and L (decreasing priorities) where H and L share a mutex object.

1. Assume that task L with lowest priority requests and acquires a mutex object (critical region).
2. Task H, which has highest priority, then starts and requests the mutex object. As only one task at a time can execute code in a mutex object, H must wait until L releases the object.
3. Task M, which has medium priority, preempts task L according to the priority rules and then starts its execution.
   - Priority inversion has now occurred because task M preempted a task (H) with higher priority.
   - The blocking time for task H now depends on a task (M) with lower priority that does not even use the mutex object.
   - If task M should use another mutex object there would also be a potential risk that deadlock could occur.

Priority inversion



Deadline inversion



Mars Pathfinder 1997

A thrilling read: the infamous priority inversion bug in the Mars Pathfinder spacecraft project:

- Risat Pathan's overview (from a Chalmers PhD student course)
- Mike Jones' report (from the RTSS'97 conference)
- Glenn Reeves' comments (Pathfinder's software team leader)

Found in Canvas under 'Resources' / 'Miscellaneous information'

"Even when you think you've tested everything that you can possibly imagine, you're wrong!" (Glenn Reeves)

Priorities and shared resources

Avoiding priority and deadline inversion:

- Non-preemptive critical regions:
  - May create unnecessary blocking
  - Only recommended for short critical regions
- Access-control protocols for critical regions:
  - Priority Inheritance Protocol (PIP) [static priority]
  - Deadline Inheritance Protocol (DIP) [dynamic priority]
  - Priority Ceiling Protocol (PCP) [static priority]
  - Stack Resource Policy (SRP) [static and dynamic priority]

Priorities and shared resources

Priority Inheritance Protocol:

- Basic idea:

  When a task τ_i blocks one or more higher-priority tasks, it temporarily assumes (inherits) the highest priority of the blocked tasks.

- Advantage:
  - Prevents medium-priority tasks from preempting τ_i and prolonging the blocking duration experienced by higher-priority tasks.

- Disadvantage:
  - May deadlock: priority inheritance can cause deadlock
  - Chained blocking: the highest-priority task may be blocked once by every other task executing on the same processor.

Priorities and shared resources

Priority Ceiling Protocol:

- Basic idea:

  Each resource is assigned a priority ceiling equal to the priority of the highest-priority task that can lock it.

  Then, a task τ_i is allowed to enter a critical region only if its priority is higher than all priority ceilings of the resources currently locked by tasks other than τ_i.

  When a task τ_i blocks one or more higher-priority tasks, it temporarily inherits the highest priority of the blocked tasks.

- Advantage:
  - No deadlock: the use of priority ceilings prevent deadlocks
  - No chained blocking: a task can be blocked at most the duration of one critical region.

Priorities and shared resources

Ada 95, Real-Time Java and POSIX provide support for the Immediate Ceiling Priority Protocol (ICPP), a simpler-to-implement version of PCP.

TinyTimber provides support for the Deadline Inheritance Protocol (DIP), which is similar to PIP but uses EDF priorities instead of static priorities:

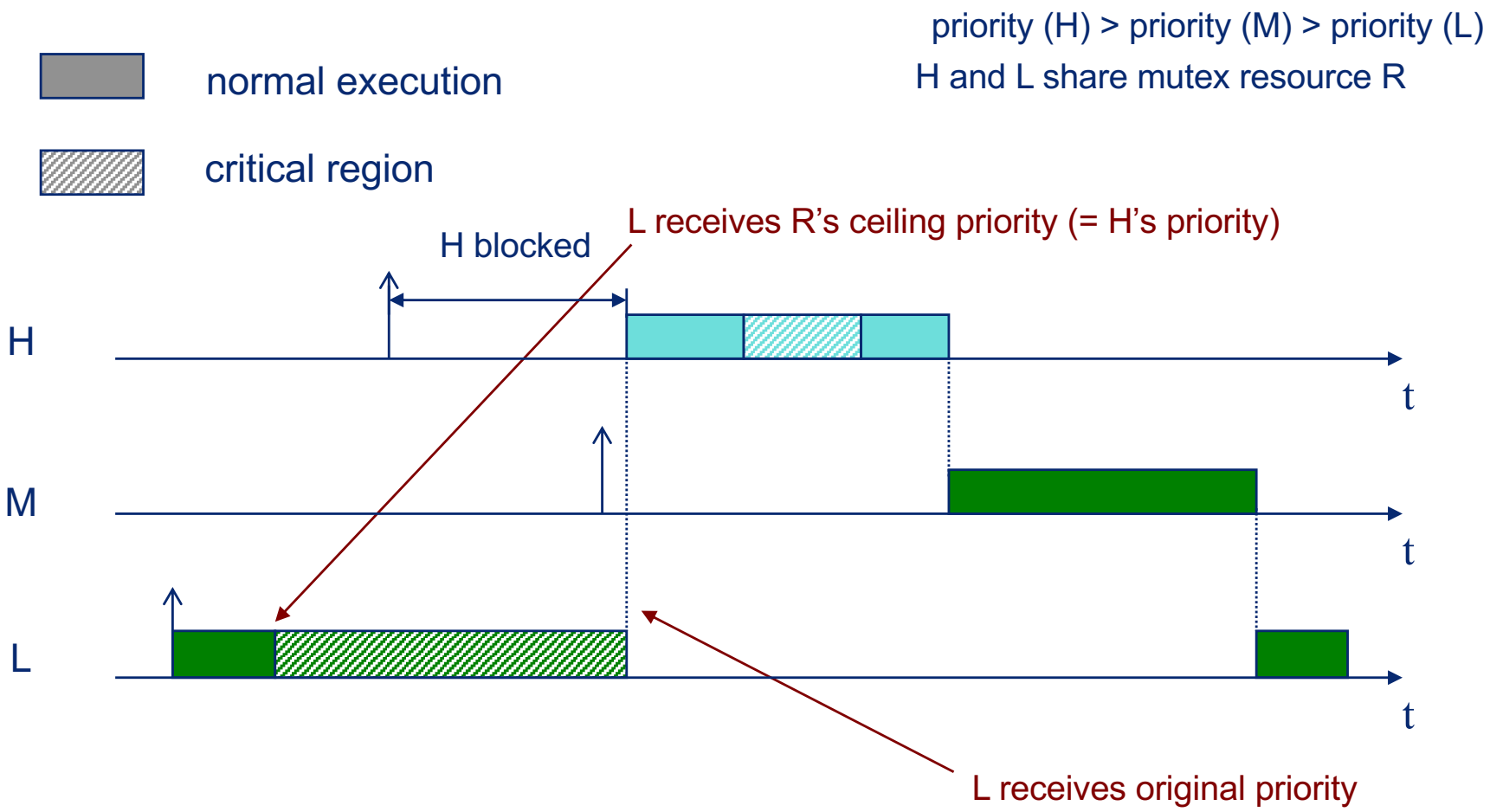
"When a task blocks one or more tasks with deadlines earlier in time, it temporarily assumes (inherits) the deadline earliest in time of the blocked tasks."

Priorities and shared resources

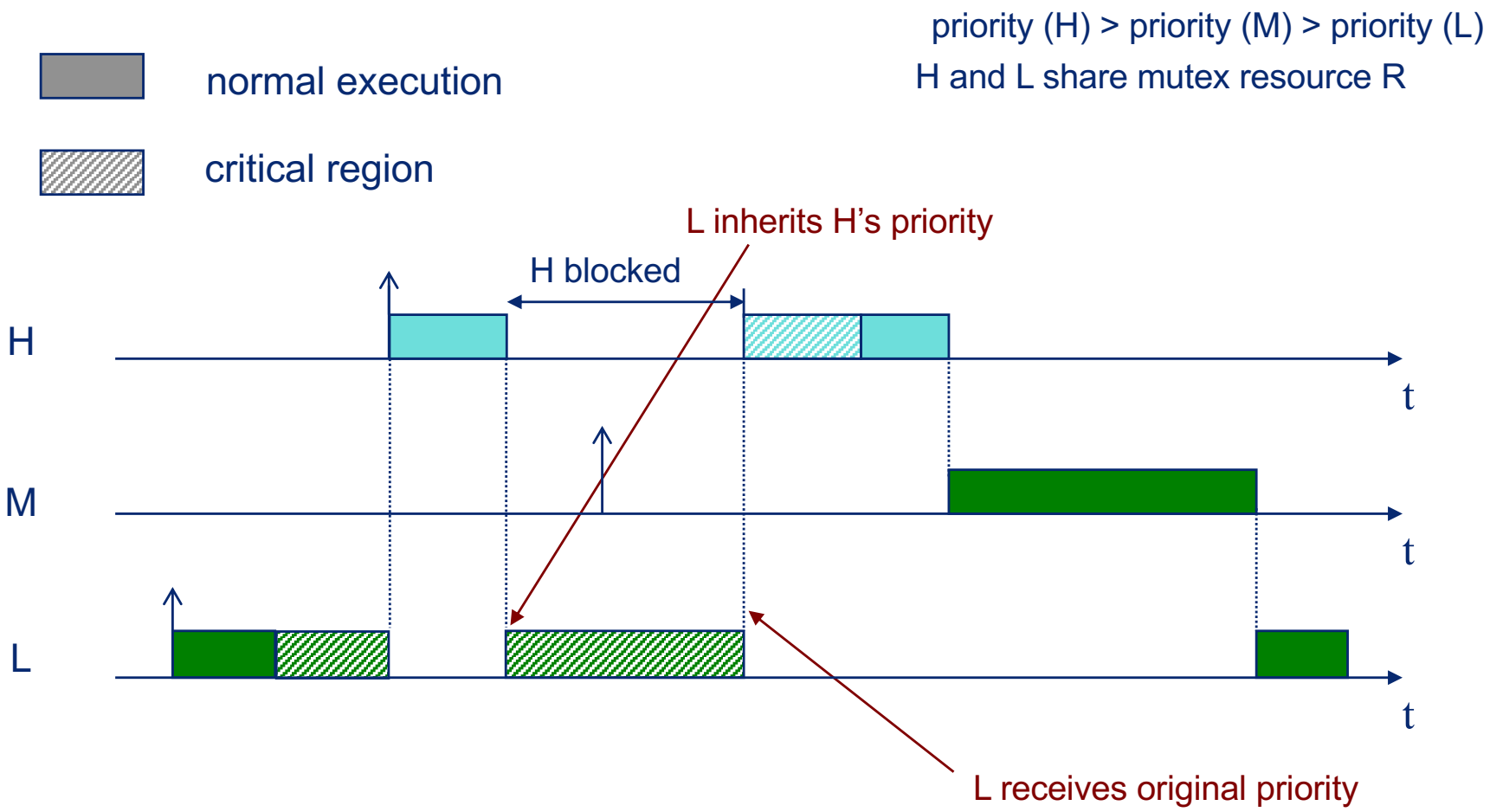
Comparison of original PCP and ICPP:

- Similarities:
  - The worst-case behavior of the two ceiling schemes is identical from a scheduling view point, and they both prevent deadlock.
- Differences:
  - In original PCP, a task X's priority is raised only when a higher-priority task tries to acquire a resource that X has locked.
  - In ICPP, a task X's priority is immediately raised to the ceiling priority of a resource when X locks the resource.
  - ICPP is simpler to implement as blocking relationships need not be continuously monitored, and also has fewer task switches as blocking happens prior to first execution of a higher-priority task.

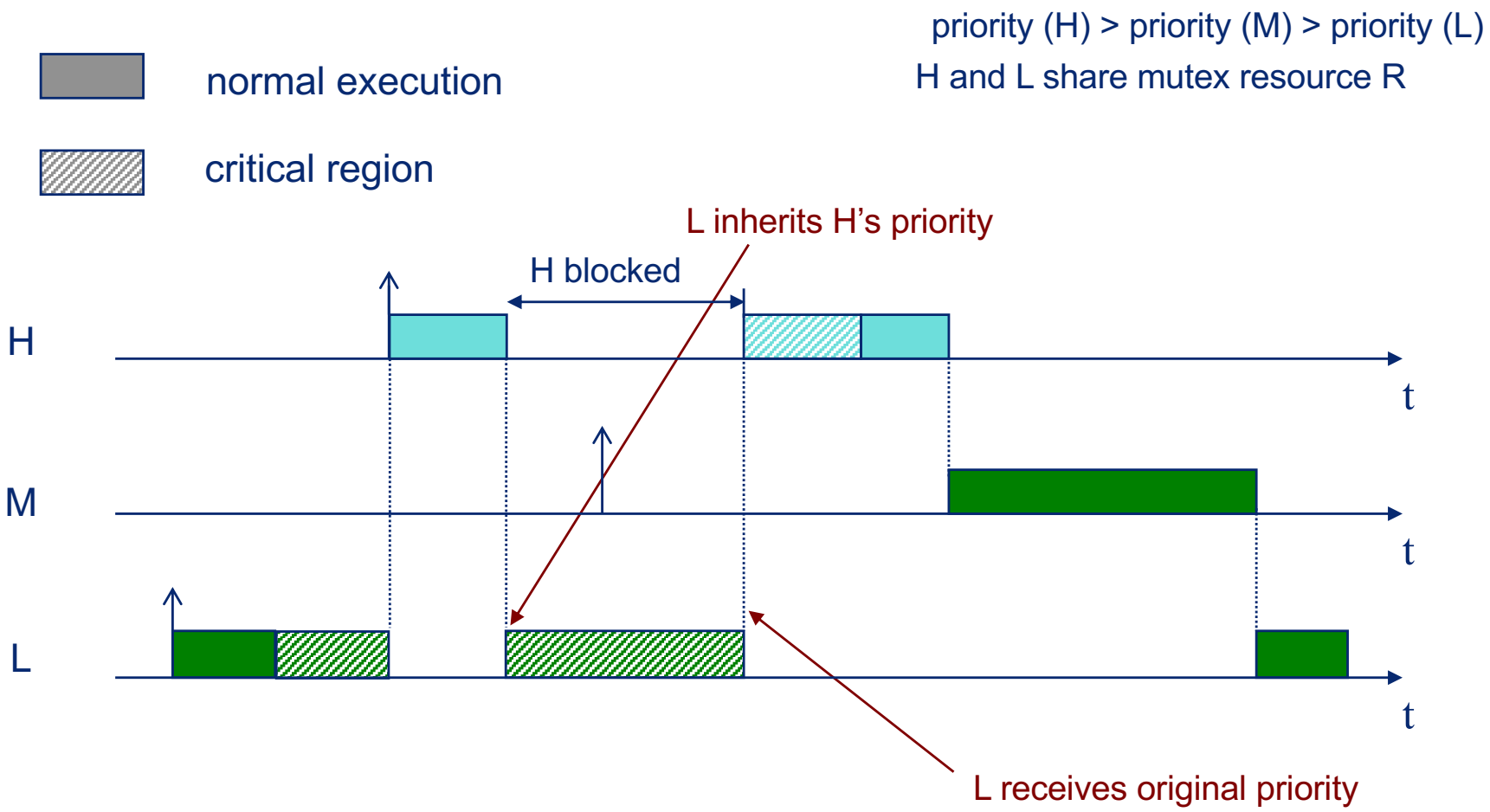
ICPP


priority (H) > priority (M) > priority (L)
H and L share mutex resource R
normal execution
critical region
L receives R's ceiling priority (= H's priority)
H blocked
H
M
L
t
t
t
L receives original priority

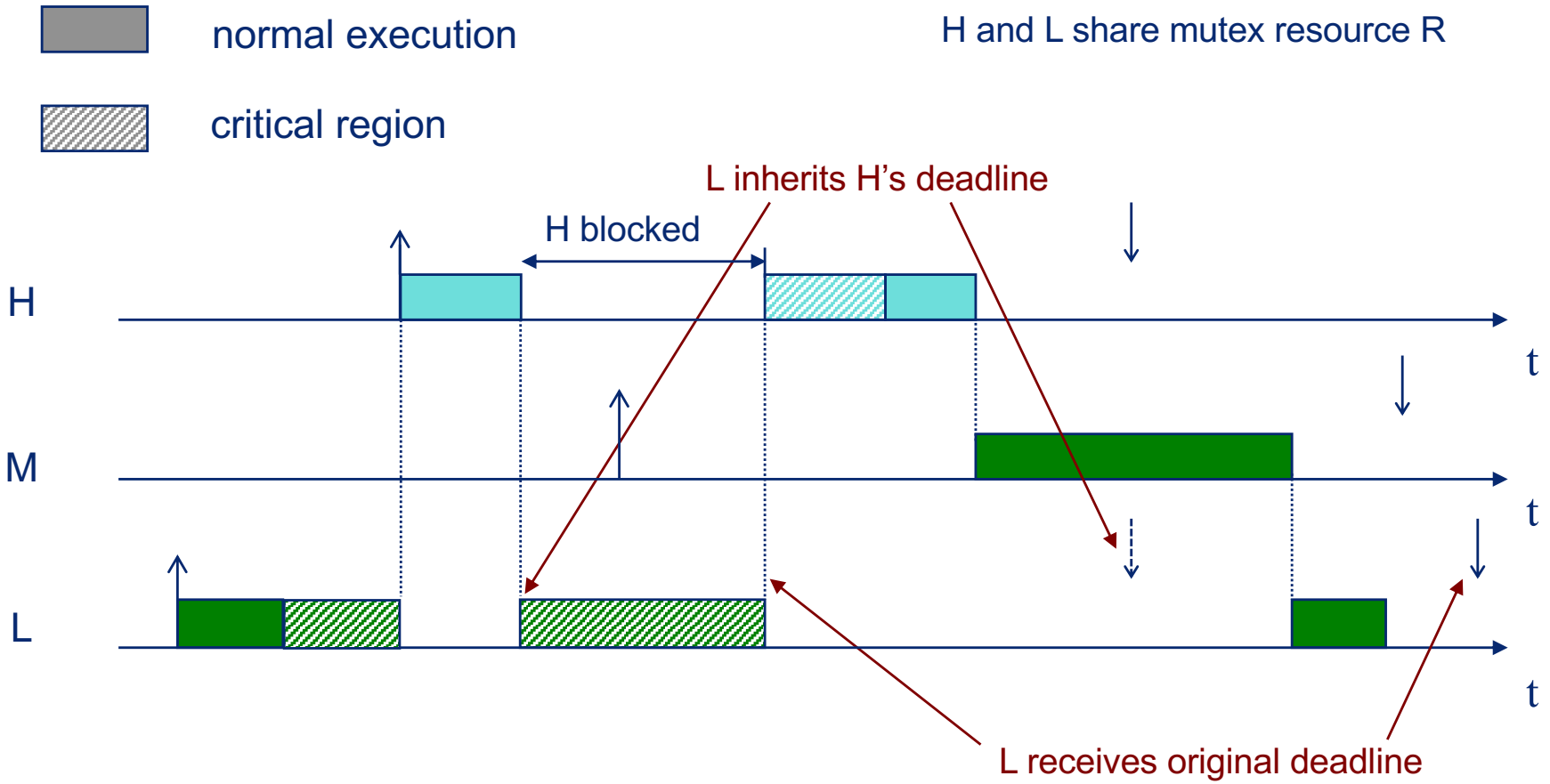
Original PCP



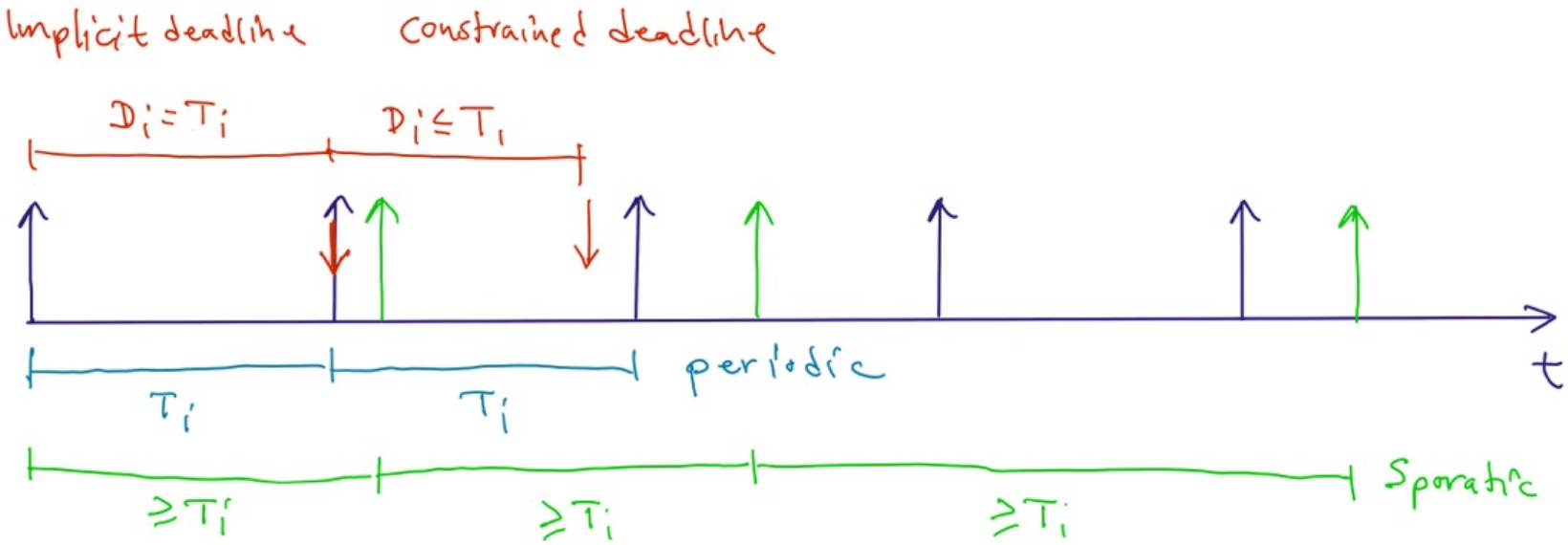
PIP


priority (H) > priority (M) > priority (L)
H and L share mutex resource R
normal execution
critical region
L inherits H's priority
H blocked
H
M
L
t
t
t
L receives original priority

DIP


normal execution
critical region
H and L share mutex resource R
L inherits H's deadline
H blocked
H
M
L
t
t
t
L receives original deadline

Lecture #7 – blackboard scribble



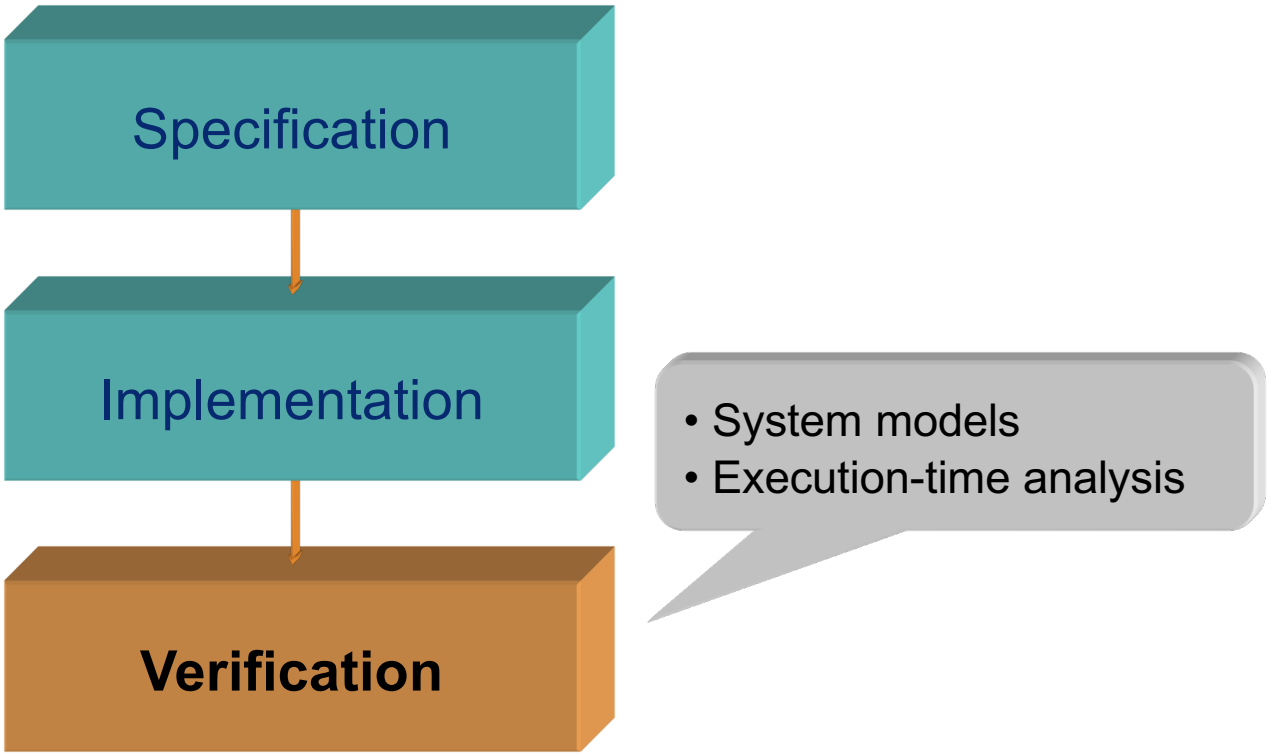
Real-Time Systems

Lecture #7

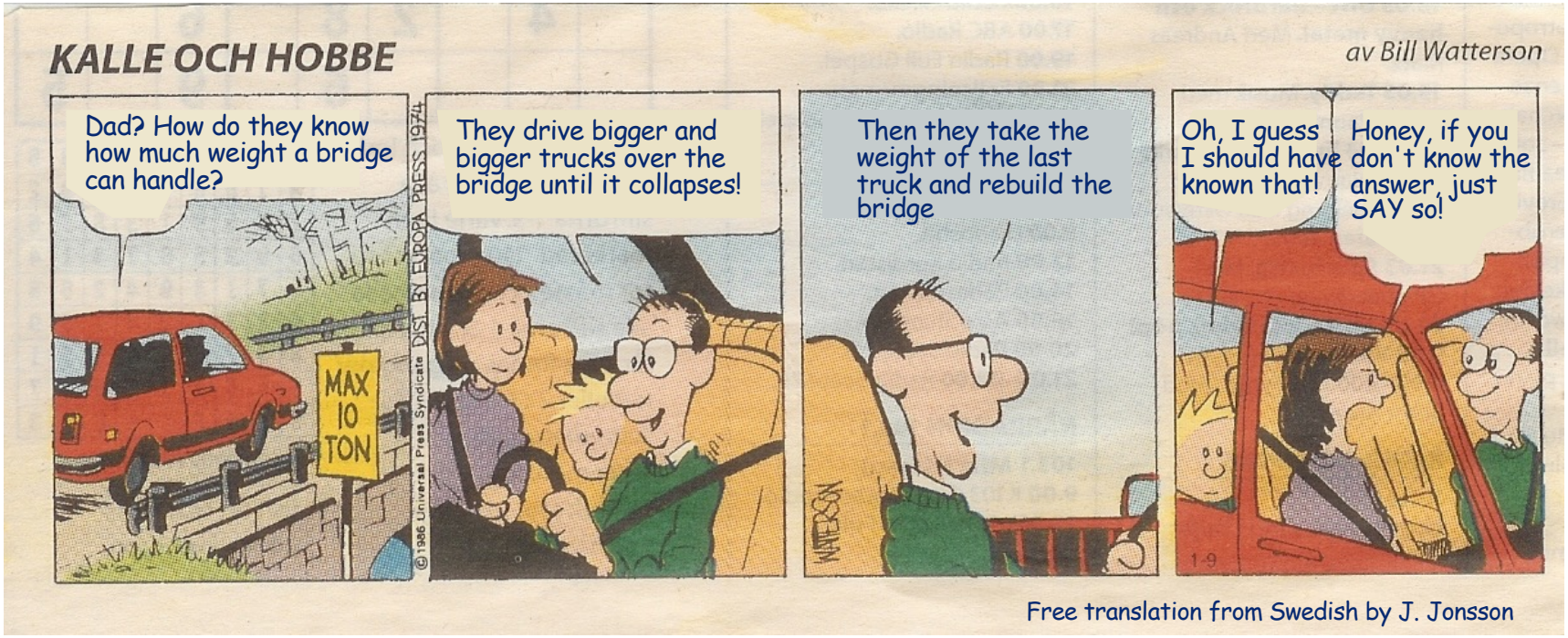
Professor Jan Jonsson

Department of Computer Science and Engineering
Chalmers University of Technology

Real-Time Systems



Verification by testing



Free translation from Swedish by J. Jonsson

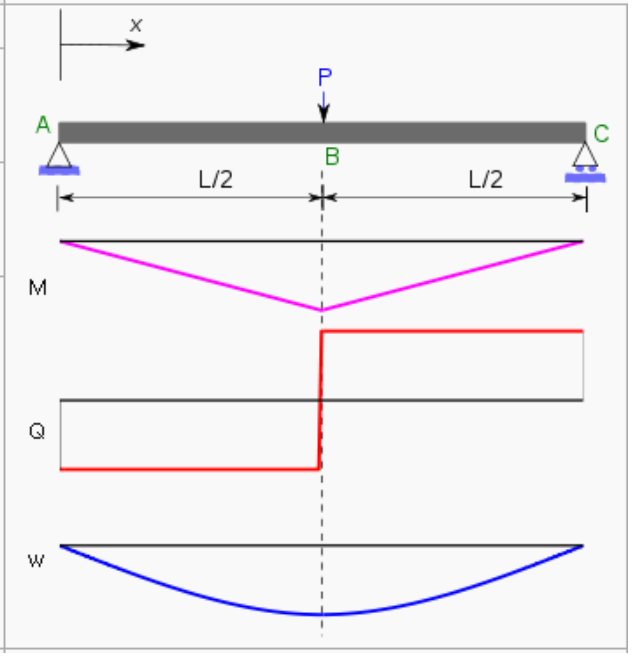
Verification by testing

So, is this how bridges (or other mechanical constructions) are built?

Of course not! There are models (properties of materials) and theories (laws of mechanics) involved to determine in advance that a construction will withstand the predicted load.

Verification by models & theory

Distribution	Max. value
Simply supported beam with central load	
$M(x) = \begin{cases} \frac{Px}{2}, & \text{for } 0 \le x \le \frac{L}{2} \\ \frac{P(L-x)}{2}, & \text{for } \frac{L}{2} < x \le L \end{cases}$	$M_{L/2} = \frac{PL}{4}$
$Q(x) = \begin{cases} \frac{P}{2}, & \text{for } 0 \le x \le \frac{L}{2} \\ \frac{-P}{2}, & \text{for } \frac{L}{2} < x \le L \end{cases}$	$\lvert Q_0 \rvert = \lvert Q_L \rvert = \frac{P}{2}$
$w(x) = \begin{cases} -\frac{Px(4x^2-3L^2)}{48EI}, & \text{for } 0 \le x \le \frac{L}{2} \\ \frac{P(x-L)(L^2-8Lx+4x^2)}{48EI}, & \text{for } \frac{L}{2} < x \le L \end{cases}$	$w_{L/2} = \frac{PL^3}{48EI}$



So, why cannot computer systems be built and verified in advance using models and theories?

Well, they can … using system models and schedulability analysis

Verification

How do we perform schedulability analysis?

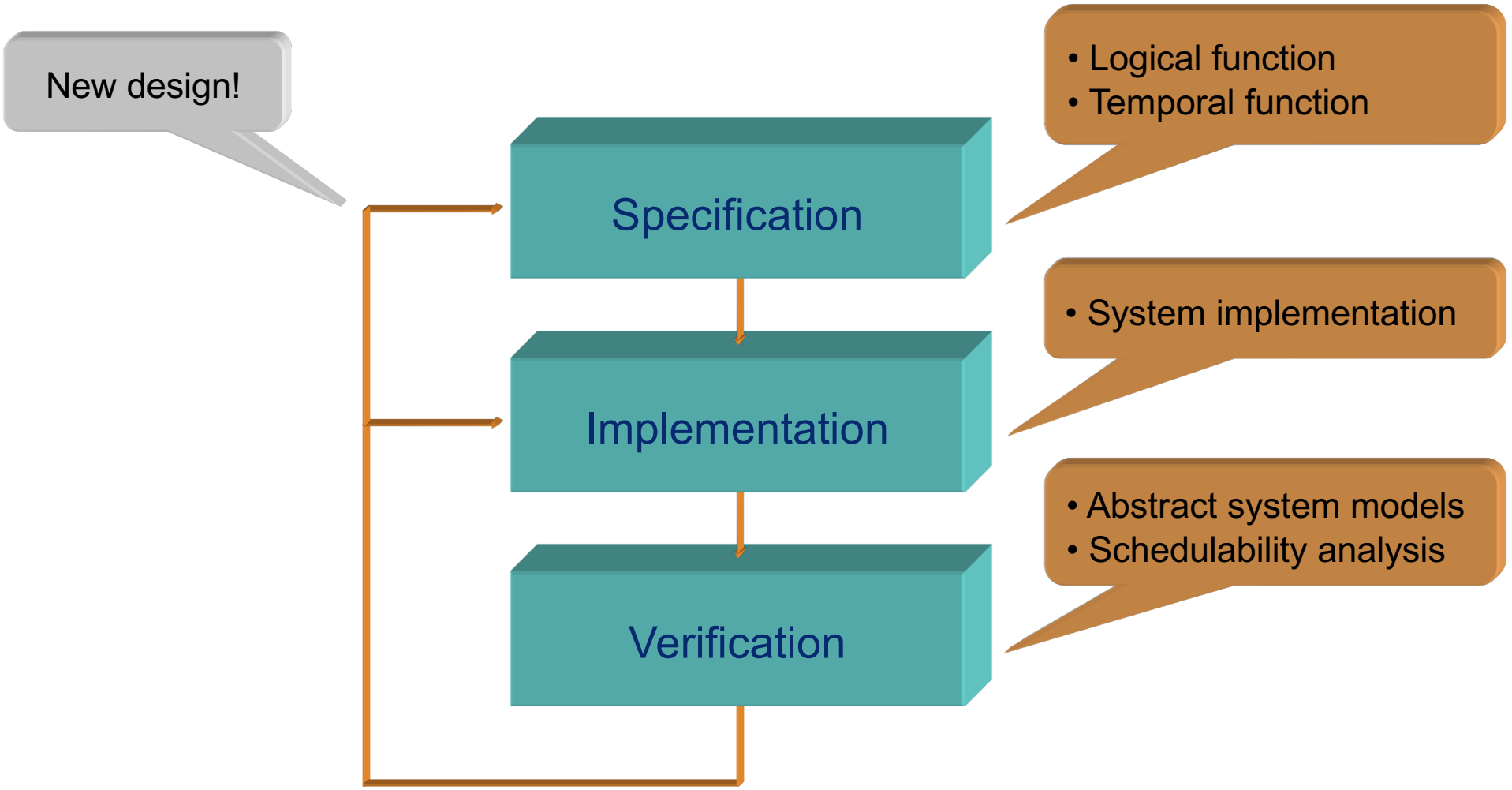
- Introduce abstract models of system components:
  - Task model (computation requirements, timing constraints)
  - Processor model (resource capacities)
  - Run-time model (task states, dispatching)
- Predict whether task executions will meet constraints
  - Use timing-correct abstract system models
  - Make sure that computation requirements never exceed resource capacities
  - Generate a (partial or complete) run-time schedule resulting from task executions and detect worst-case scenarios

Verification

How do we simplify schedulability analysis?

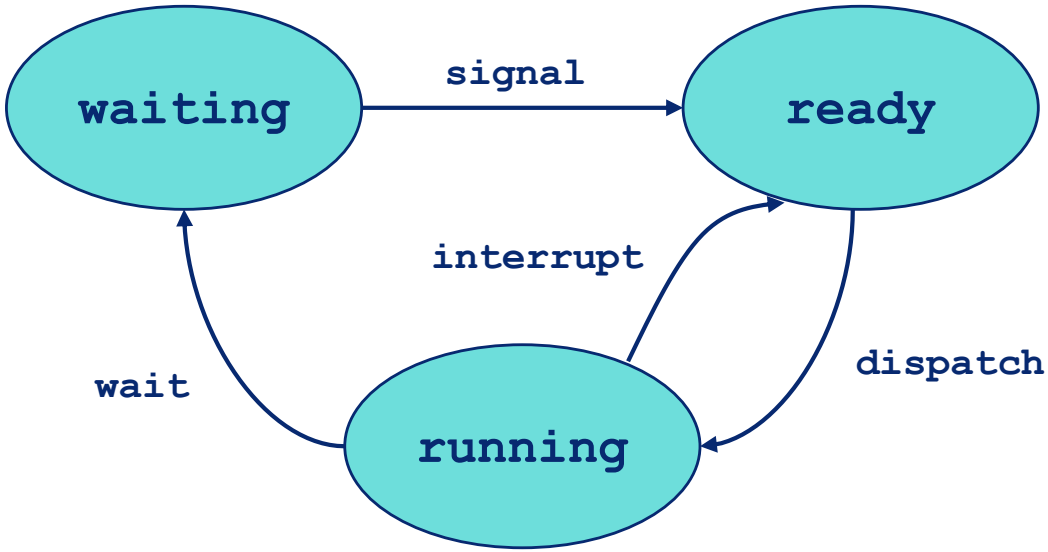
- Concurrent and reactive programming paradigm
  - Suitable schedulable entity (thread, method, …)
  - Language constructs for expressing application constraints for schedulable entities (data types, annotations, macros, …)
  - Estimated WCET for schedulable entities
- Deterministic task execution
  - Time tables or static/dynamic task priorities
  - Preemptive task execution
  - Run-time protocols for access to shared resources (dynamic priority adjustment and non-preemptable code sections)

Designing a real-time system



Run-time model

The run-time model expresses the state of a task:

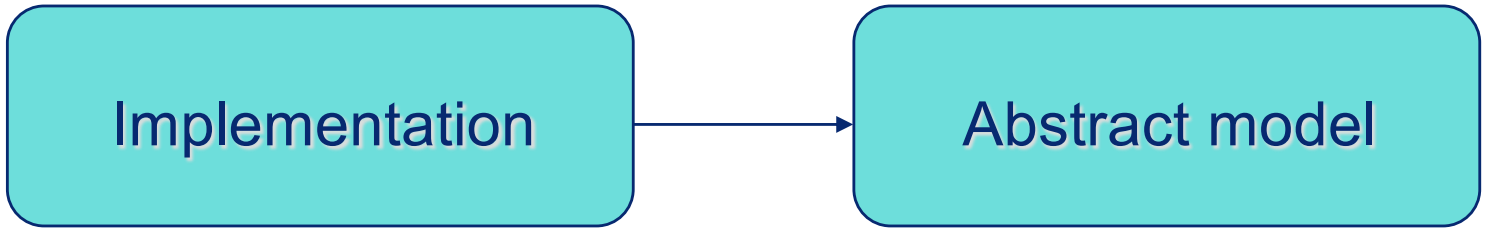


Running: Currently executing task

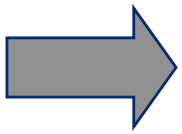
Ready: Task that is available for execution

Waiting: Task that cannot execute because it is needs access to a resource other than the processor

Task model




```
void task1(Object *self, int p) {
    Action1();
    SEND(Period1, Deadline1, self, task1, p);
}

void task2(Object *self, int p) {
    Action2();
    SEND(Period2, Deadline2, self, task2, p);
}

void kickoff(Object *self, int p) {
    AFTER(Offset1, &app1, p);
    AFTER(Offset2, &app2, p);
}

main() {
    TINYTIMBER(&app_main, kickoff, 0);
}
```



τ_1 $\quad \tau_1 = \{ C_1, T_1, D_1, O_1 \}$

τ_2 $\quad \tau_2 = \{ C_2, T_2, D_2, O_2 \}$

Task model

The task model expresses the timing behavior of a task:

- The static parameters describe characteristics of a task that apply independent of other tasks.
  - These parameters are derived from the specification or the implementation of the system
  - For example: period, deadline, WCET
- The dynamic parameters describe effects that occur during the execution of a task.
  - These parameters are a function of the run-time system and the characteristics of other tasks
  - For example: start time, completion time, response time

Task model

Static task parameters:

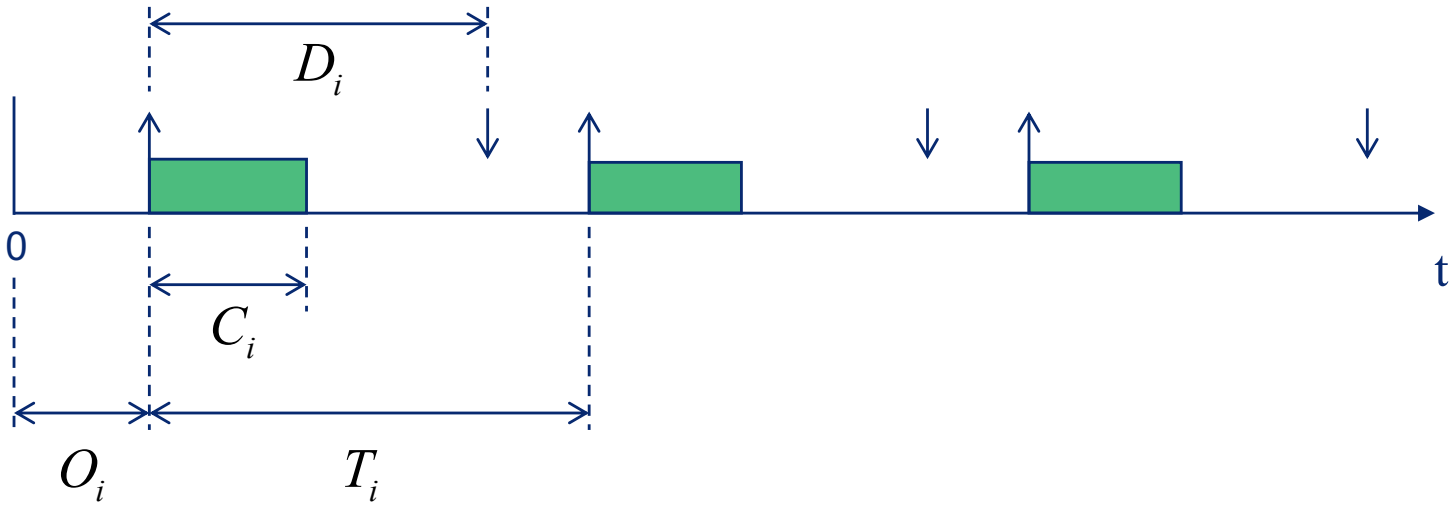
τ_i $\quad \tau_i = \{ C_i, T_i, D_i, O_i \}$

C_i : (undisturbed) WCET

T_i : period

D_i : (relative) deadline

O_i : (absolute) time offset



Task model

Dynamic task parameters:

τ_i $\quad \tau_i = \{ C_i, T_i, D_i, O_i \}$

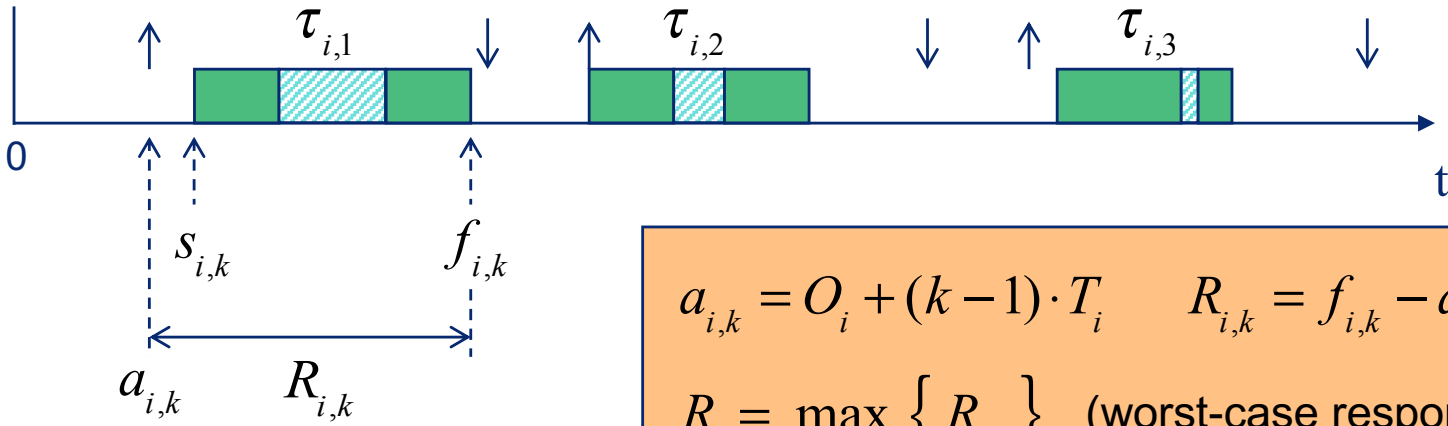
$a_{i,k}$: arrival time of k^{th} instance

$s_{i,k}$: start time of k^{th} instance

$f_{i,k}$: completion time of k^{th} instance

$R_{i,k}$: response time of k^{th} instance

$\tau_{i,k}$: the k^{th} instance of τ_i



$$a_{i,k} = O_i + (k-1) \cdot T_i \qquad R_{i,k} = f_{i,k} - a_{i,k}$$

$$R_i = \max_{\tau_i \in \mathbf{T}, k \geq 1} \{ R_{i,k} \}$$ (worst-case response time)

Task model

Synchronous and asynchronous task sets:

- In a *synchronous* task set the offsets of tasks are identical, that is: $\forall i, j : O_i = O_j$
- In an *asynchronous* task set the offsets of at least one pair of tasks are not identical, that is: $\exists i, j : i \neq j, O_i \neq O_j$

Asynchronous task sets are typically used to reduce local skew (jitter) or to remove the need for resource access protocols.

Note: Two tasks with identical periods, but different offsets, will never arrive simultaneously during the lifetime of the system. This means that the worst-case response times of the tasks will be lower than if the offsets of the tasks were equal.

Task model

Task deadline/period relationship:

- Implicit-deadline tasks
  - For each task it applies that $D_i = T_i$
- Constrained-deadline tasks
  - For each task it applies that $D_i \leq T_i$
- Arbitrary-deadline tasks
  - No restrictions placed on the relation between D_i and T_i

For most of the scheduling problems the implicit-deadline and constrained-deadline cases can be solved in reasonable time.

The arbitrary-deadline case, on the other hand, is in general the most time-consuming to solve.

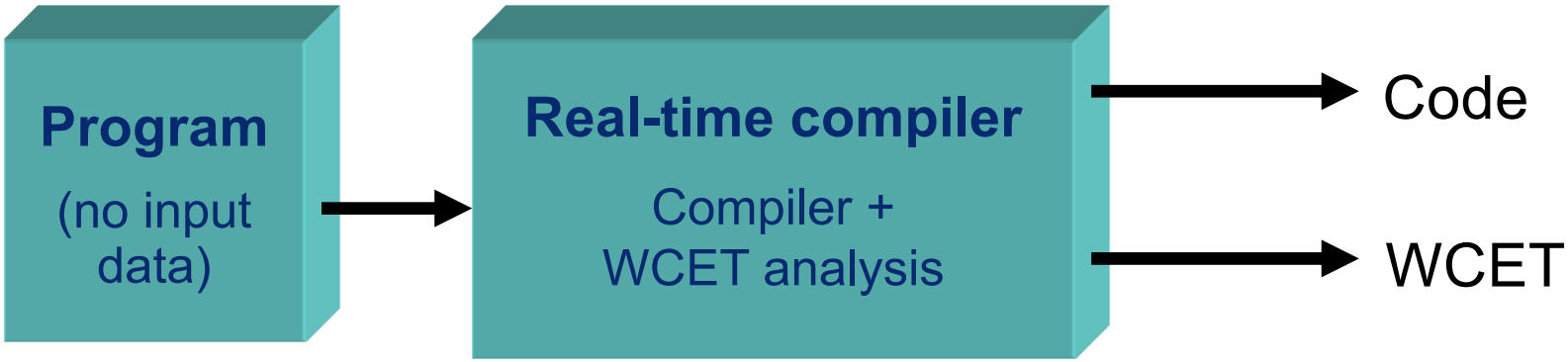
Task model

Task arrival patterns:

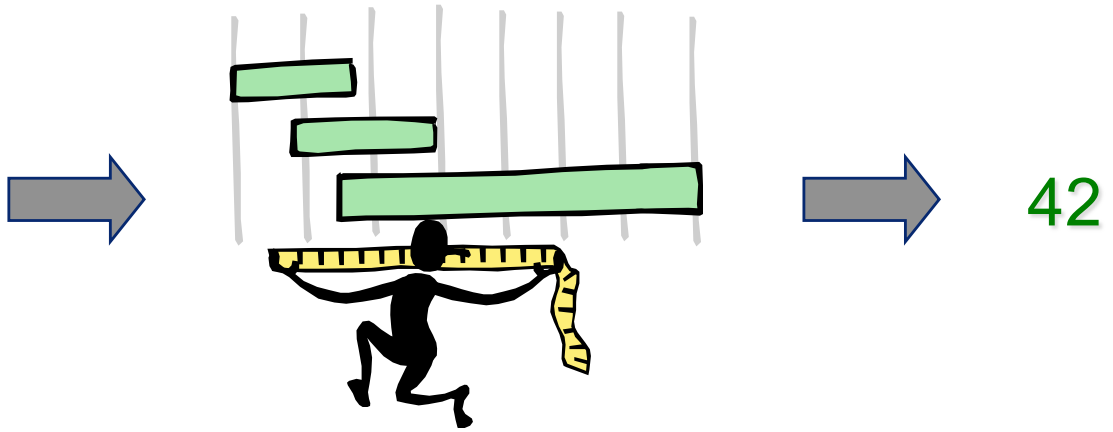
- Periodic tasks
  - A periodic task arrives with a time interval T_i
- Sporadic tasks
  - A sporadic task arrives with a time interval ≥ T_i
- Aperiodic tasks
  - An aperiodic task has no guaranteed minimum time between two subsequent arrivals

⇒ *A priori* schedulable (hard) real-time systems can only contain periodic and sporadic tasks.

Execution-time analysis




```
for (i=1; i<=N; i++) {
  if (A > K)
    A = K-1;
  else
    A = K+1;
  if (A < K)
    A = K;
  else
    A = K-1;
}
```



Execution-time analysis

Background:

- Worst-case execution time (WCET) is needed to
  - perform (hard) schedulability analysis
  - identify resource needs early in the design phase
  - perform program tuning (critical loops and interrupt handlers)
- The WCET of a task depends on
  - program structure + initial system state + input data
  - temporal properties of the system (OS + hardware)
  - internal and external system events
- WCET estimates can be obtained via
  - measurements
  - static analysis

Execution-time analysis

Requirements:

- A WCET estimate must be pessimistic but tight

  0 ≤ ”Estimated WCET” – “Real WCET” < ε

  (ε small compared to real WCET)

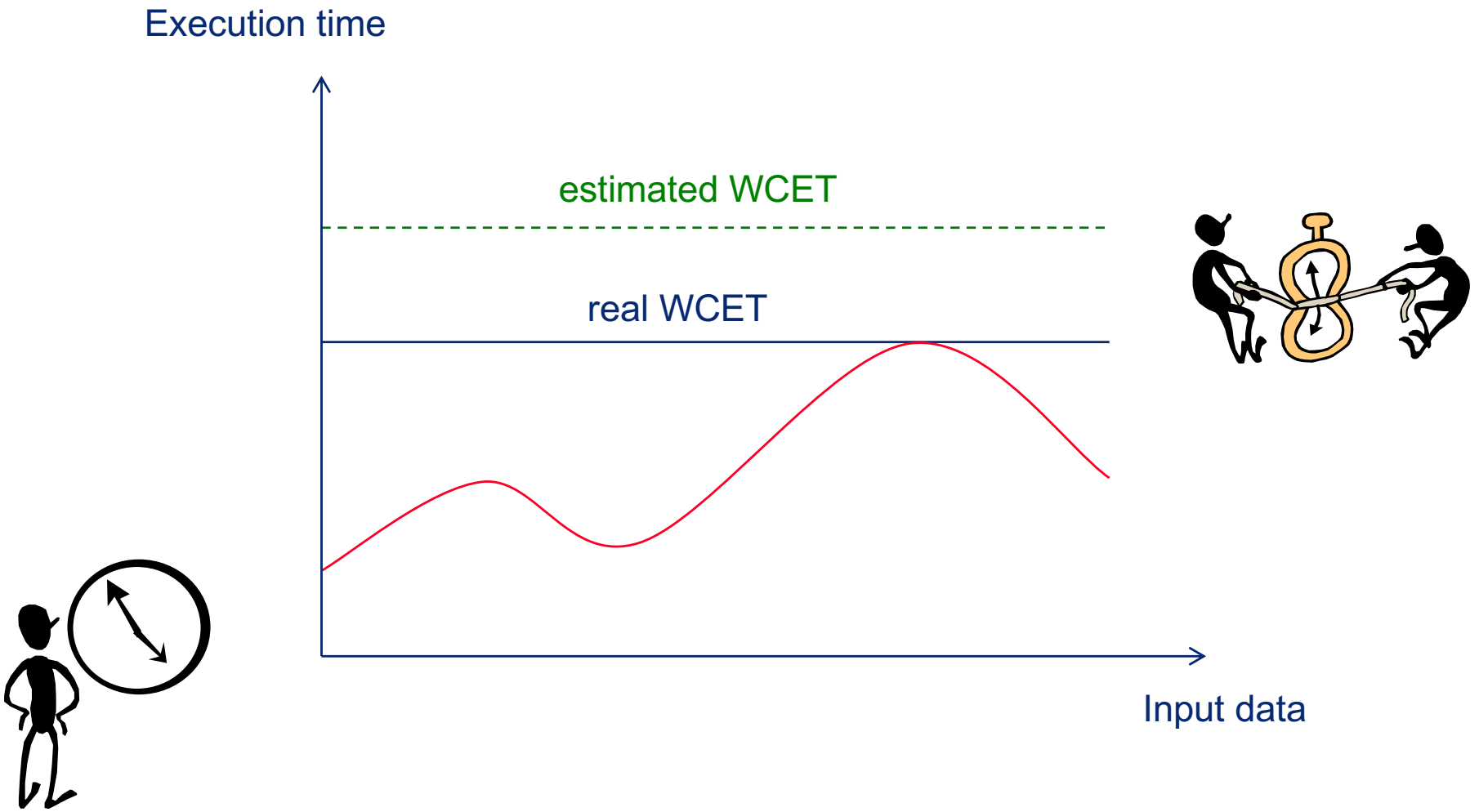
Pessimistic:

to make sure assumptions made in the schedulability analysis of hard real-time tasks also apply at run time

Tight:

to avoid unnecessary pessimism in the schedulability analysis, which could cause feasibility tests to be too inaccurate to be useful

Execution-time analysis



Execution-time analysis

Estimating WCET via measurements:

- Methodology:
  - identify potential worst-case scenario
  - run program code on hardware using worst-case scenario
  - measure the execution time
  - add a safety margin
- Measuring techniques:
  - system clocks, cycle-level simulators, in-circuit emulators
  - observe hardware signals with oscilloscope or logic analyzer
- Reflection:
  - measured execution time will never exceed real WCET
  - how large must safety margin be to get a pessimistic estimate?

Execution-time analysis

Estimating WCET via static analysis:

- Methodology:
  - determine the longest execution time of the program code without actually running it
  - uses models based on properties of software and hardware
  - typically integrated with the compiler tools
- Analysis techniques:
  - Path analysis: bound the number of times that different program parts may be executed
  - Timing analysis: bound the execution time of program parts
- Reflection:
  - real WCET will never exceed estimated execution time
  - how accurate must the models be to get a tight estimate?

A simple (yet challenging) example

Derive WCET for the following program:

```
for (i=1; i<=N; i++) {
  if (A > K)
    A = K-1; (T1)
  else
    A = K+1; (E1)
  if (A < K)
    A = K;   (T2)
  else
    A = K-1; (E2)
}
```

Issues to consider:

- Input data is unknown
  - Iteration bounds must be known to facilitate analysis
- Path explosion
  - 4^N paths in this example
- Exclusion of non-executable (false) paths
  - T1 + E2 is a false path in the example

A simpler (but non-trivial) example

Derive WCET for the following statement:

```
A = A / B;
```

Issues to consider:

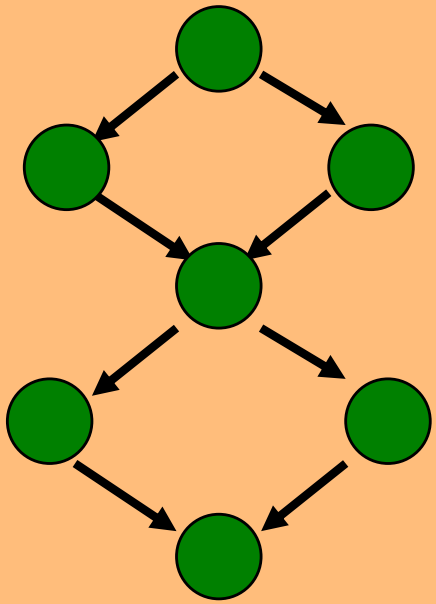
- Execution time:
  - affected by cache misses, pipeline conflicts, exceptions ...
  - depends on previous and (!) subsequent instructions
  - also depends on (unknown) input data
- Observations:
  - accurate estimation of WCET must be based on a detailed timing model of the system architecture
  - uncertainties are handled by making worst-case assumptions

Fundamental issues

- In the path analysis:
  - how to bound the number of iterations in a loop / recursion
  - how to eliminate false (non-executable) paths cause by e.g. if-then-else statements
- In the timing analysis:

  Everything that takes time must be modeled in a realistic fashion (or at least not optimistically)
  - must accurately model the temporal behavior of hardware (influence of, e.g., cache memories, pipelining, …)
  - must account for consequences of run-time events (e.g.: exceptions, interrupts, context switches)

Path analysis



A control flow graph (CFG) describes the structure of the program

Path analysis problem:

Find the longest executable path in the program's CFG

Path analysis

Shaw's Timing Schema (1989):

```
for (i=1; i<=N; i++) {
  if (A > K)
    A = K-1; (T1)
  else
    A = K+1; (E1)
  if (A < K)
    A = K;   (T2)
  else
    A = K-1; (E2)
}
```

The estimated WCET (WCETe) is the execution time of the longest structural path through the program

```
WCETe =
 N*(WCET(loop) +
 WCET(I1) +
 max(WCET(T1), WCET(E1)) +
 WCET(I2) +
 max(WCET(T2), WCET(E2)))
```

Methods for path analysis

Manual method:

Programmer must provide information

Annotation of loop bounds:

- Provide upper bounds on loop indices and catch potential exceptions at run time

Elimination of false paths:

- Enumerate all possible paths and list the set of false paths so that these can be avoided in the analysis

Requires very detailed knowledge of the program's function, and is therefore also very prone to errors!

Methods for path analysis

Automated method:

Support from the compiler

Derive upper bounds on loop indices:

- Requires an explicit loop index
- May not work for complicated termination conditions

Elimination of false paths:

- Symbolically execute the program to detect non-executable program statements

Current methods are promising but only for fairly simple programs where the analysis is trivial!

Methods for path analysis

The reality?

Shaw's timing schema implicitly assume that the execution time of each language statement is constant and known

This is a realistic assumption for older types of processors, that:

- lack execution pipelines
- lack cache memories
- do not generate exceptions

However, for modern RISC type processor architectures, these methods yield very pessimistic results!

Timing analysis for RISC processors

RISC processors have several advanced mechanisms (pipelining, caching, branch prediction, out-of-order execution, …) that cause significant variation in the execution time of a processor instruction.

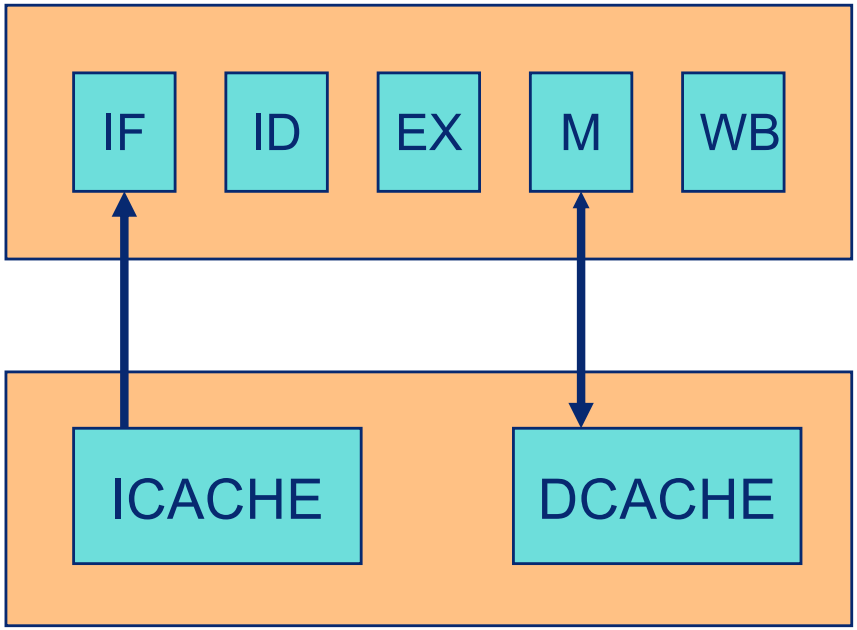
We must therefore estimate the execution time for each executable path through the program and at the same time account for these mechanisms.

This can be solved by partitioning the program code into code blocks and analyze each block separately.

Today, mature methods for timing analysis only exist for pipelining and caching.

Timing analysis for RISC processors

Processor with pipeline:



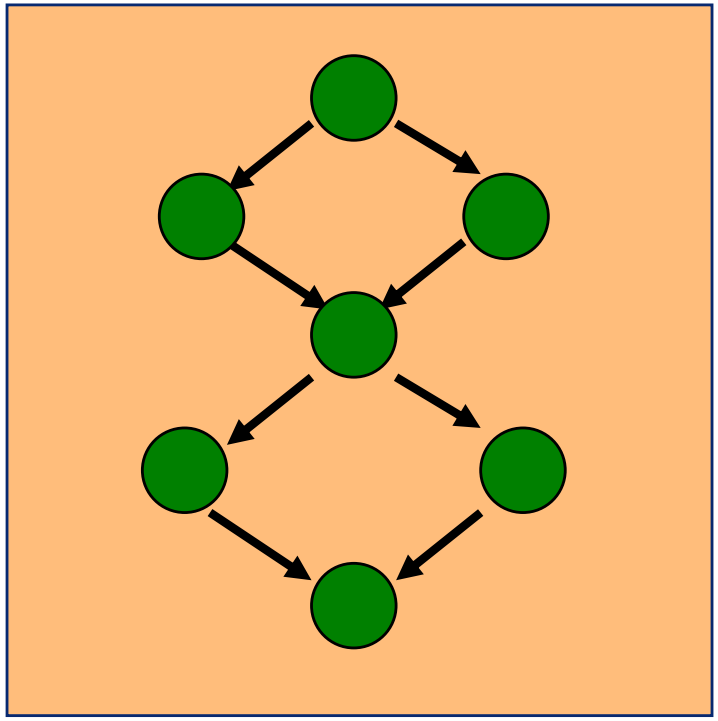
Sources of time variations:

- data conflicts
- branch conflicts

Sources of time variations:

- cache misses
  (have order-of-magnitude higher access times than cache hits)

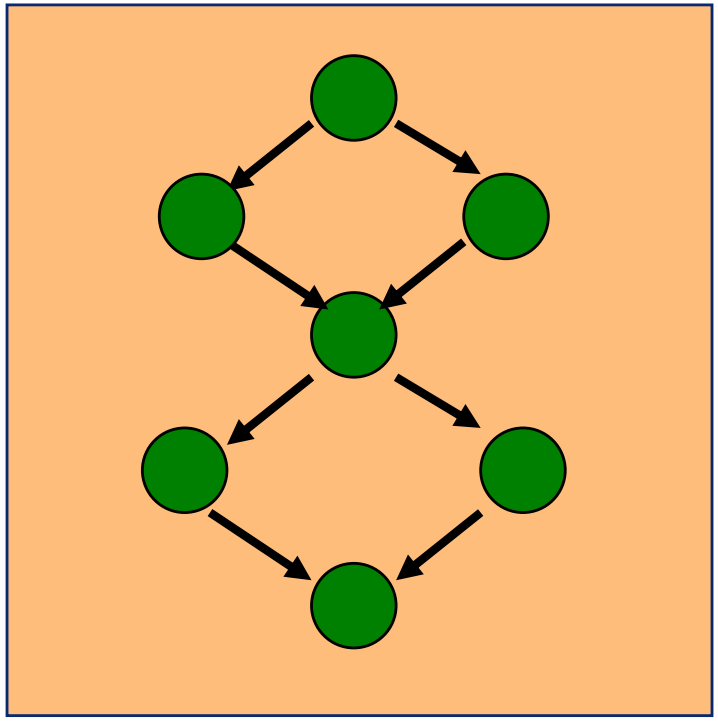
Timing analysis of cache memory



Issues:

- Not enough to investigate an isolated code block
  - miss/hit depends on previous executions of the code
- Instruction cache behavior is predictable for each path
  - known sequence of code
- Data cache behavior is more difficult to analyze
  - data addresses can depend on the program's input data

Timing analysis of pipeline



Issues:

- Not enough to investigate an isolated code block
  - conflicts may occur on the boundary between code blocks
- Pipeline behavior is predictable for each path
  - known sequence of code

Methods for timing analysis

Extension of Shaw's Timing Schema

- Analysis is performed at code block level
- Merging of paths at certain code locations by estimating the effects of worst-case situations (reduces path explosion)

Data flow analysis:

- Analysis performed at code block level
- Propagation of pipeline and cache states between blocks

Integer Linear Programming

- Formulate an ILP problem as a function of execution time and number of executions at code block level

Challenges

So far, non-preemptive execution of program code on a single processor has been assumed.

In reality, pseudo-parallel execution is typically used, something which requires preemptive execution.

- Preemptions will affect system state (i.e., cache contents will change and pipeline will be flushed) and must therefore be accounted for in the analysis.
- However, it is difficult to account for these effects in the analysis of WCET, which means that it must be handled at a higher level (i.e., in the schedulability analysis).

Challenges

So far, non-preemptive scheduling of program code on a single processor has been assumed.

In reality, multicore processors are used in real-time systems, something which presents new problems:

- Several processors may have copies of the same code and data in their local cache memories, and any updates will invalidate the other copies. This must be accounted for in the analysis.
- ...