CHALMERS UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG

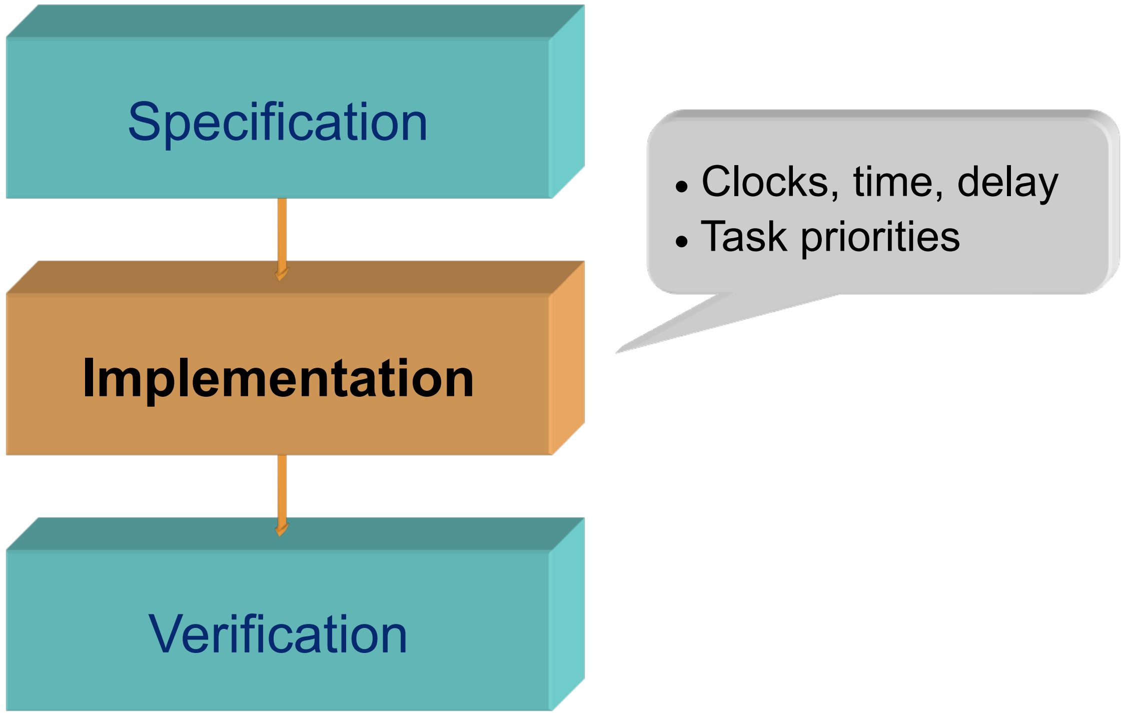
Real-Time Systems

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 (Ada95, 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
```

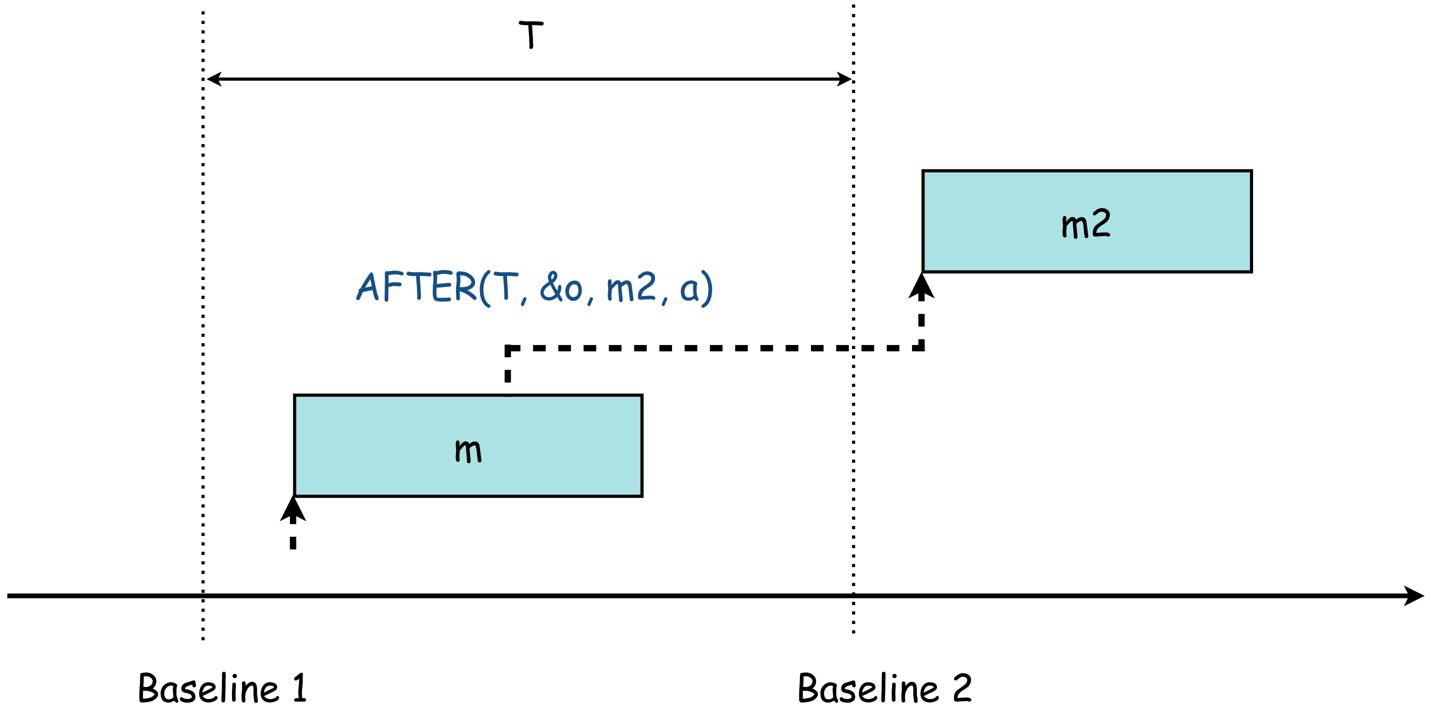
Here, `current_baseline` is the current baseline of the method posting the call with the `AFTER()` operation.

```
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 <u>specification</u>.

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

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

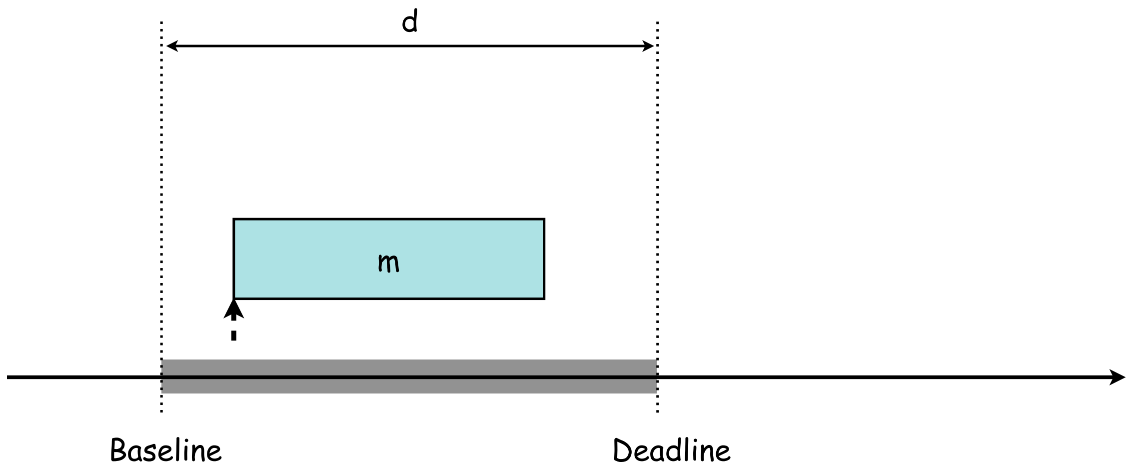
```
package Ada.Dynamic_Priorities is
  procedure Set_Priority(...);
  function Get_Priority(...) return Priority;
end Ada.Dynamic_Priorities;
```

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 closest 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 closest 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);
```

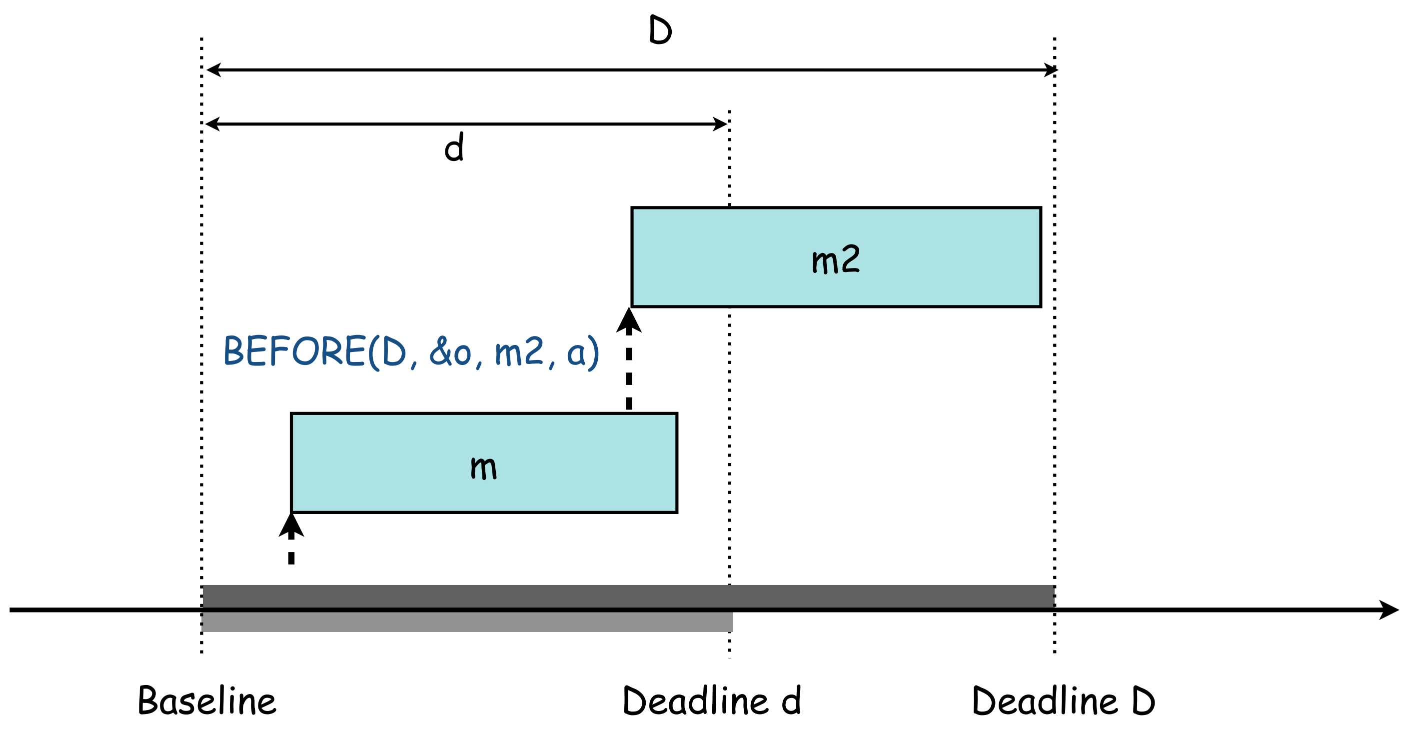
- The `BEFORE()` operation requests that the specified method should complete its execution by `deadline` at the latest:

```
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 closest 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:

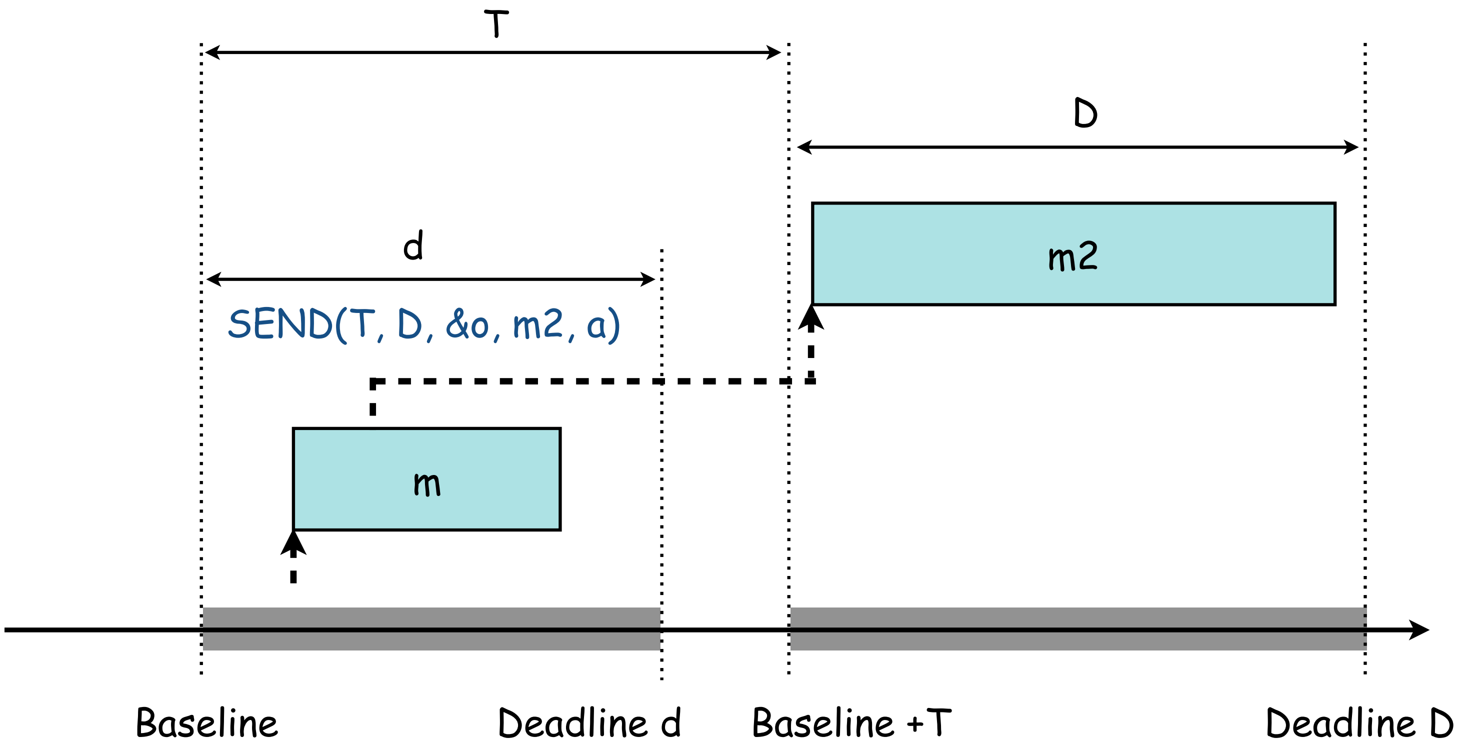
```
SEND(base_off, rel_deadline, object, method, argument);
```

- 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.

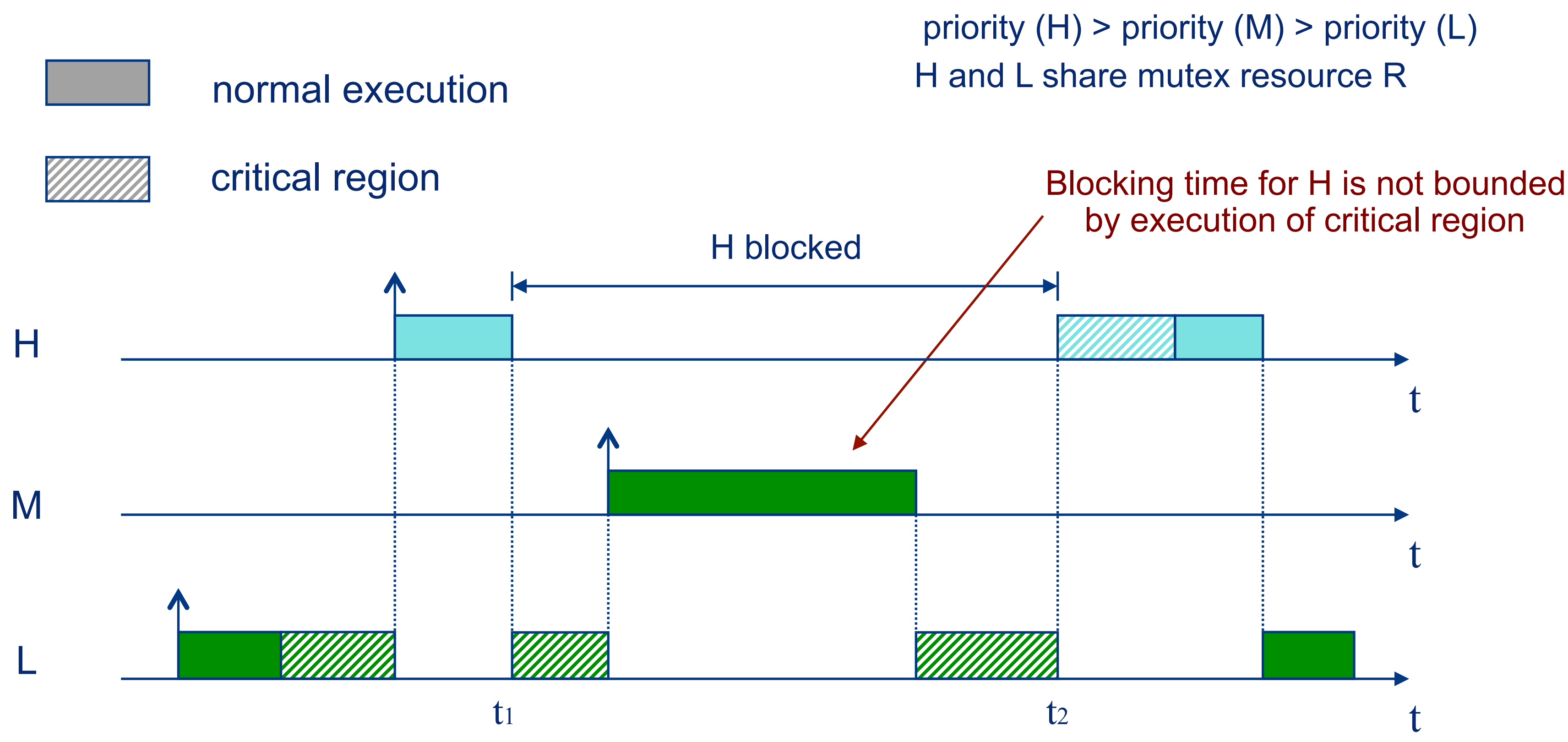
- If static priorities are used such a scenario is referred to as priority inversion.
- If dynamic priorities are used such a scenario is referred to as deadline inversion.

Priority inversion

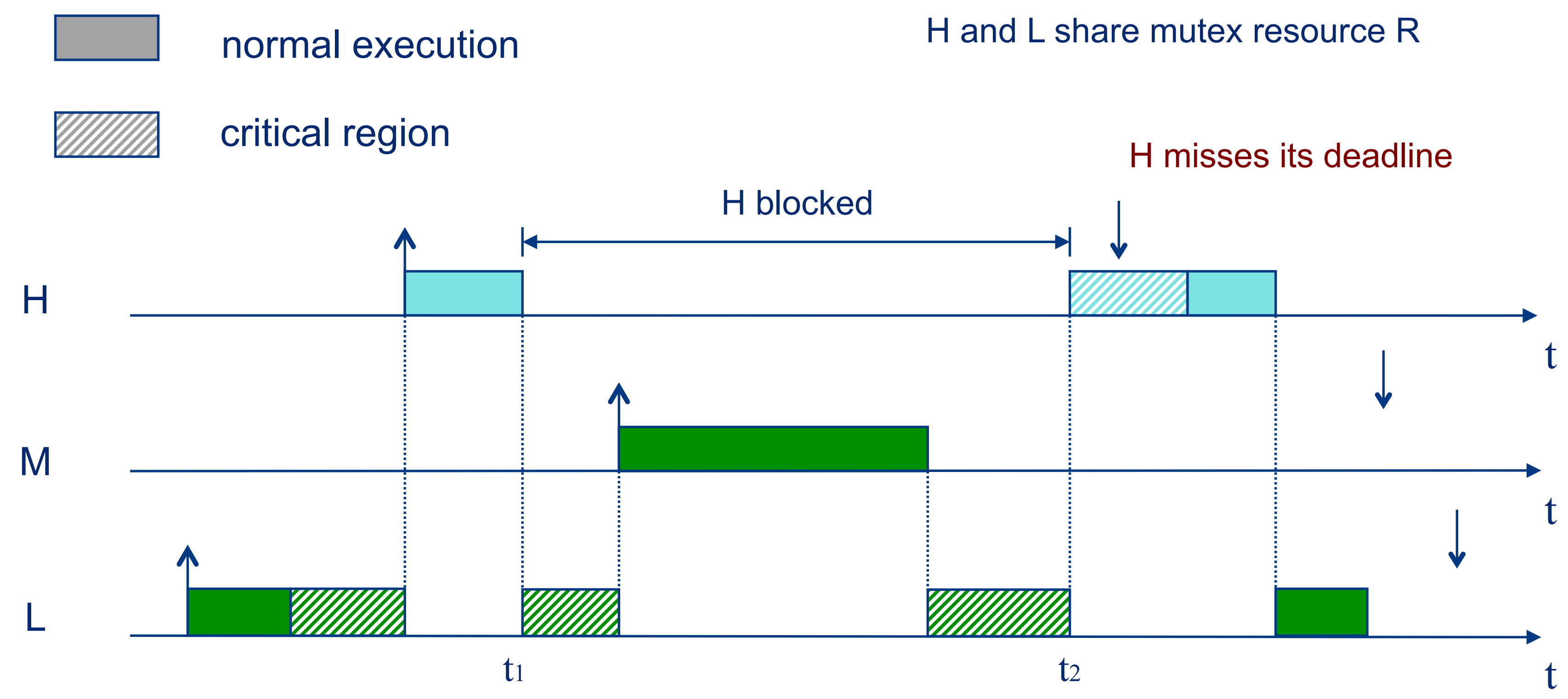
Assume three tasks H, M and L (decreasing priorities) where H and L share a mutex object.

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 <u>priority ceiling</u> 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 closer in time, it temporarily assumes (inherits) the deadline closest in time of the blocked tasks."

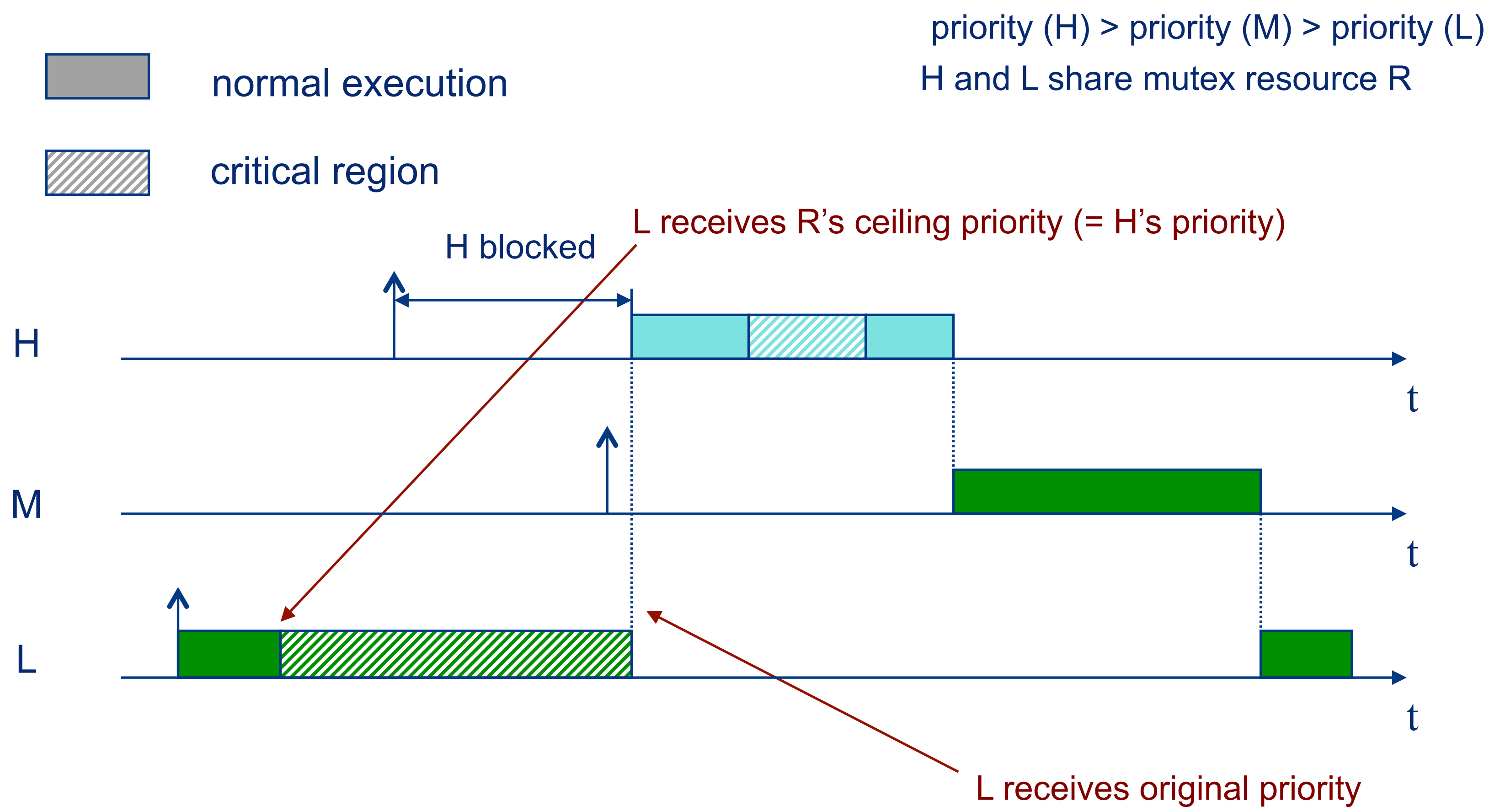
To help avoid the potential deadlock problem associated with the simpler inheritance protocols (PIP, DIP), TinyTimber also offers a deadlock detection mechanism (that indicate deadlock situations via the return value of the `SYNC()` operation.)

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.
  - Thus, ICPP is simpler to implement as blocking relationships need not be continuously monitored. ICPP also has fewer task switches as blocking happens prior to the first execution of a higher-priority task.

Immediate Ceiling Priority Protocol



Deadline Inheritance Protocol