http://cs.LTH.se/EDA040

Real-Time and Concurrent Programming

Lecture 6 (F6):

Scheduling and bounded response times

Klas Nilsson

2015-10-06

1 Software and timing
- Periodic activities
- The importance of proper timing.
- Worst-case execution time (WCET)
- Worst-case blocking time

2 Scheduling principles
- Static scheduling
- Dynamic scheduling – priority-based
- Dynamic scheduling

3 Bounded blocking time
- Priority inversion and avoidance
- Limiting max blocking despite multiple resources.

1 Software and timing
- Periodic activities
- The importance of proper timing.
- Worst-case execution time (WCET)
- Worst-case blocking time

2 Scheduling principles
- Static scheduling
- Dynamic scheduling – priority-based
- Dynamic scheduling

3 Bounded blocking time
- Priority inversion and avoidance
- Limiting max blocking despite multiple resources.

Periodic processes/events

We often want to perform something cyclically/periodically

```
class Control extends PeriodicThread {
  Control() {super(20);} // Sampling period 20ms

  public void perform() {
    // Perform ONE sample (i.e., no loop here!).
    double yr = getSetPoint();
    double y = sample();
    double u = controlPID(yr,y);
    setControlOutput(u);
  }
}
```

The cyclic execution (`while (!isInterrupted()) {...}`), keeping the period (20ms in this example) without drift (`sleepUntil` like for the ticking in lab 1), and catching `InterruptedException` around the sleeping/blocking, is all provided by the base class.

Supporting Java classes

The provided base classes (for you to subclass) are:

`CyclicThread:`	Cyclically recurrent task *without* a specific period.
`PeriodicThread:`	Cyclically recurrent task *with* a specific period.
`SporadicThread:`	Cyclically recurrent task with a *minimum period*.
`RTThread:`	Base class for the above classes. Not a subclass of `java.lang.Thread`
`JThread:`	Subclass of `java.lang.Thread` providing the `perform` method (as well as the `sleepUntil` as of lab 1 and the `mailbox` for lab 3).

Since lab 3 is about message-based *concurrency* (not real time) you better use `JThread`.

Supporting Java classes

The provided base classes (for you to subclass) are:

`CyclicThread:`	Cyclically recurrent task *without* a specific period.
`PeriodicThread:`	Cyclically recurrent task *with* a specific period.
`SporadicThread:`	Cyclically recurrent task with a *minimum period*.
`RTThread:`	Base class for the above classes. Not a subclass of `java.lang.Thread`
`JThread:`	Subclass of `java.lang.Thread` providing the `perform` method (as well as the `sleepUntil` as of lab 1 and the `mailbox` for lab 3).

Since lab 3 is about message-based *concurrency* (not real time) you better use `JThread`.

The remaining slides cover theory; as of the theory part of the exam.

Computer Controlled Systems

As control actions (also referred to as control signals) are typically computed by embedded software, timing requirements make it *real-time software*. Small delays in sampling time can yield large control errors, for instance via prediction errors. Two examples:

- PID control (linear extrapolation as prediction of future)

Computer Controlled Systems

As control actions (also referred to as control signals) are typically computed by embedded software, timing requirements make it *real-time software*. Small delays in sampling time can yield large control errors, for instance via prediction errors. Two examples:

- PID control (linear extrapolation as prediction of future)
- Model-based control (predicting based on dynamic process model)

Computer Controlled Systems

As control actions (also referred to as control signals) are typically computed by embedded software, timing requirements make it *real-time software*. Small delays in sampling time can yield large control errors, for instance via prediction errors. Two examples:

- PID control (linear extrapolation as prediction of future)
- Model-based control (predicting based on dynamic process model)

Refer to (click on:) the online version of *"Feedback Systems – An Introduction for Scientists and Engineers"*, by Åström and Murray for an introduction to control (and the next figure).

Extract from "Feedback Systems – ..."

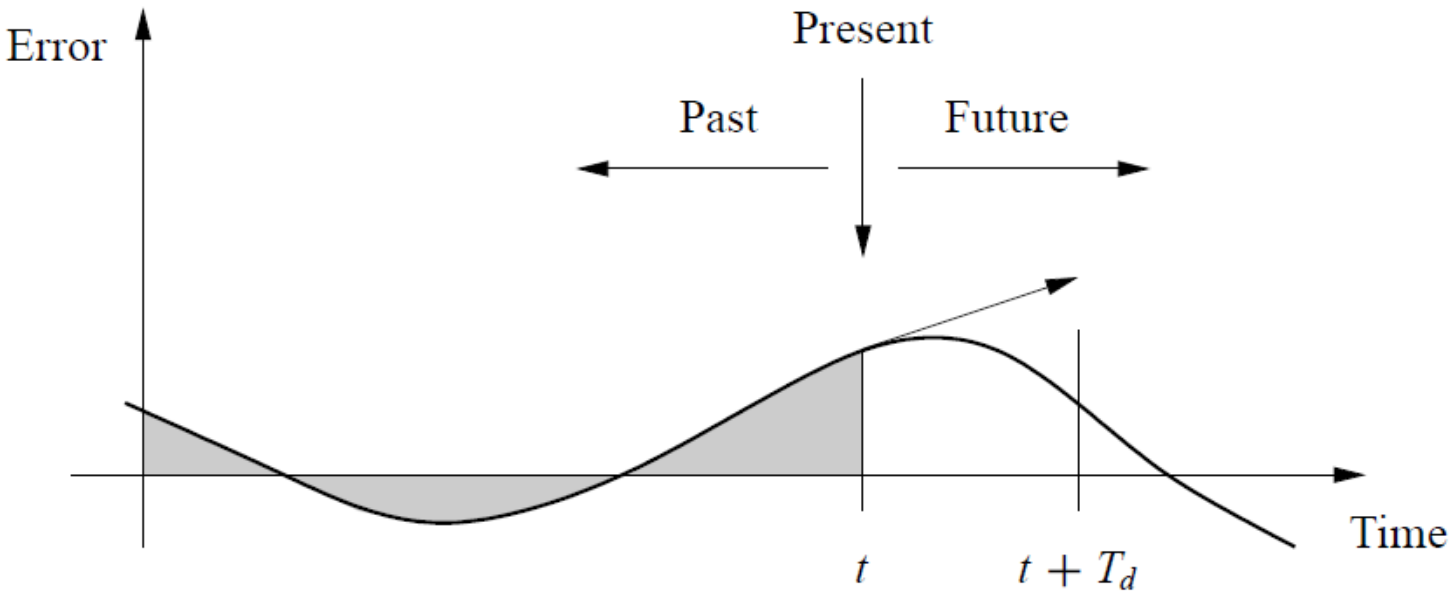
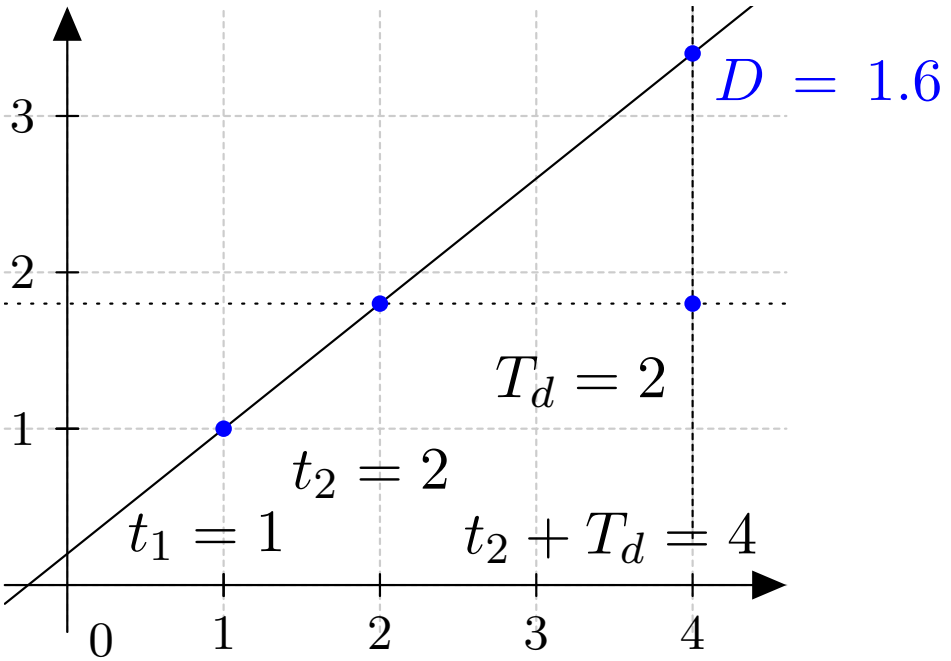
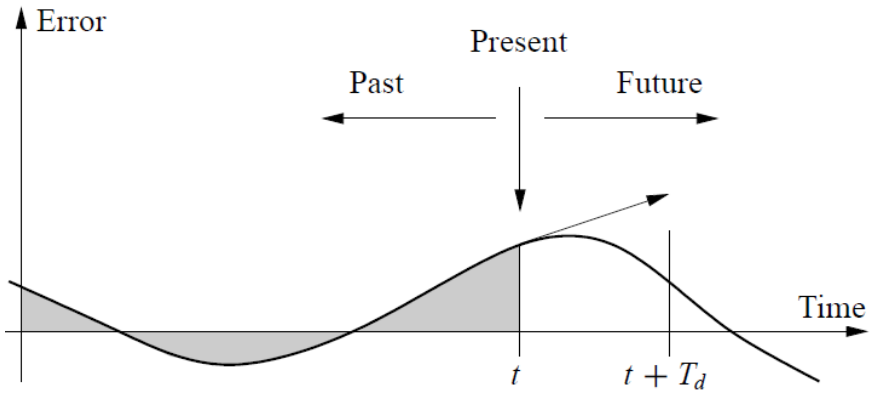


Figure 1.17: Action of a PID controller. At time t, the proportional term depends on the instantaneous value of the error. The integral portion of the feedback is based on the integral of the error up to time t (shaded portion). The derivative term provides an estimate of the growth or decay of the error over time by looking at the rate of change of the error. T_d represents the approximate amount of time in which the error is projected forward (see text).

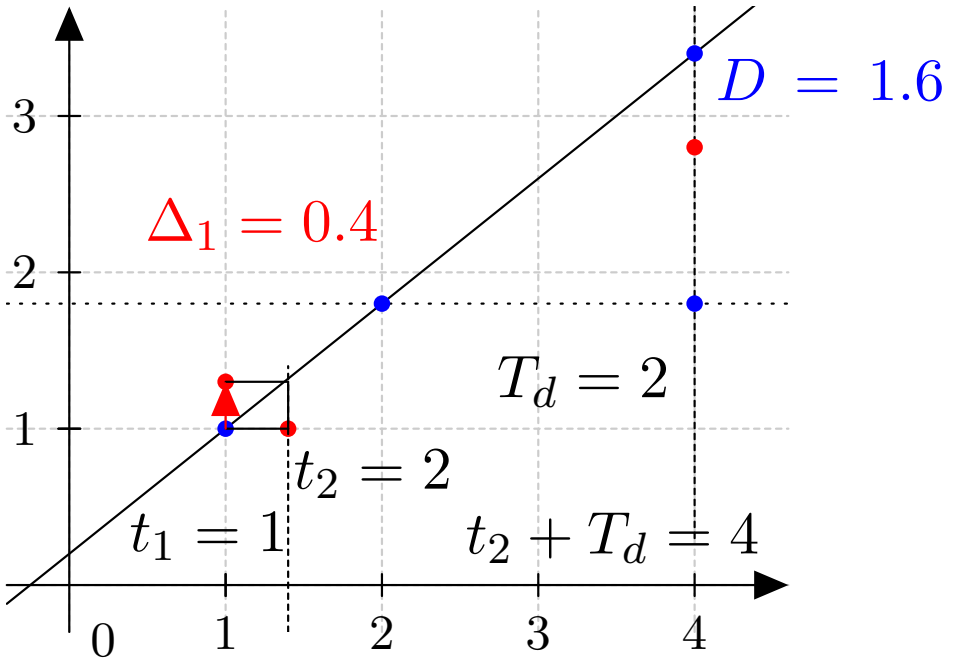
Timing errors can result in deficient or unstable control



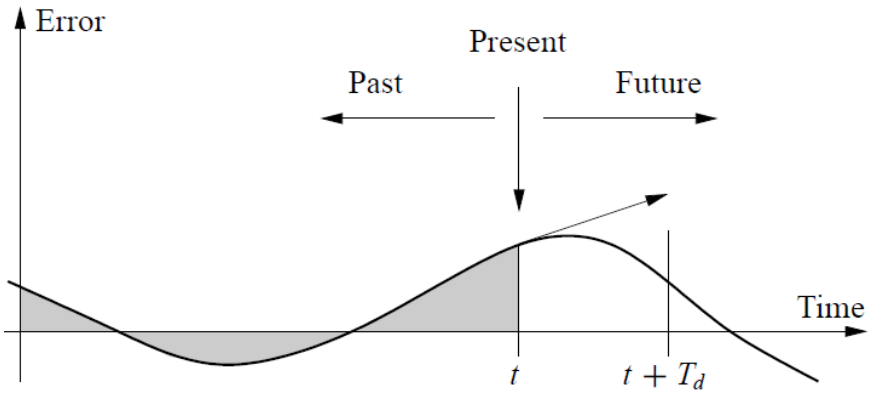
Predicting the output a time T_d into the future.



Timing errors can result in deficient or unstable control

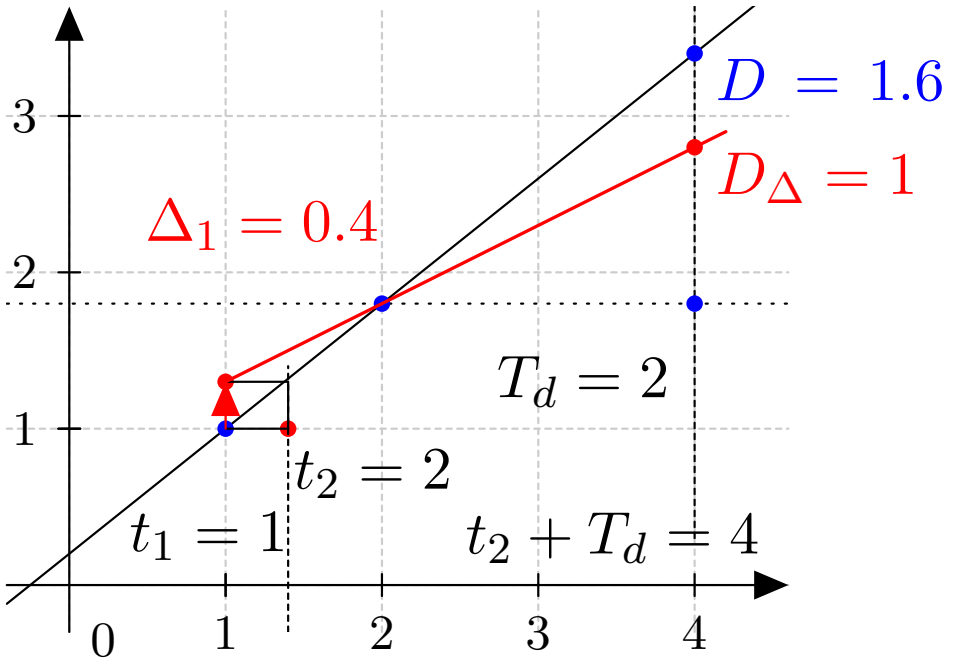


Predicting the output a time T_d into the future.

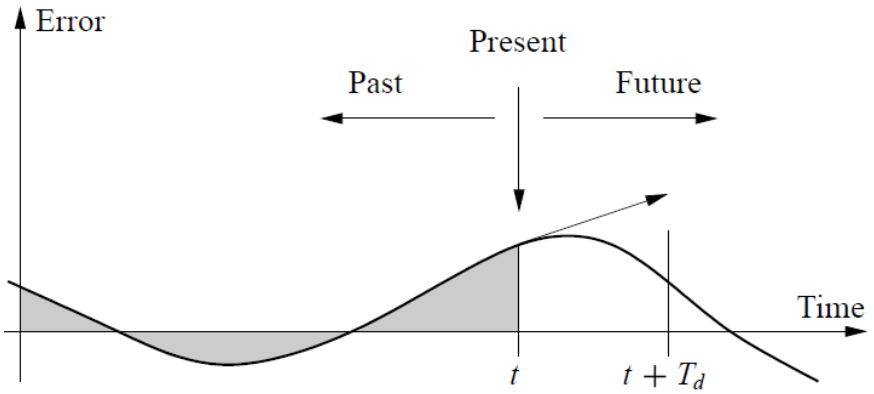


Here, as an example, with sampling period 1 [time unit], the sample at time $t = t_1 = 1$ was delayed by $\Delta = 0.4$ time units (40% of a period), and the predictive D-part at $t = t_2 = 2$ changes accordingly.

Timing errors can result in deficient or unstable control

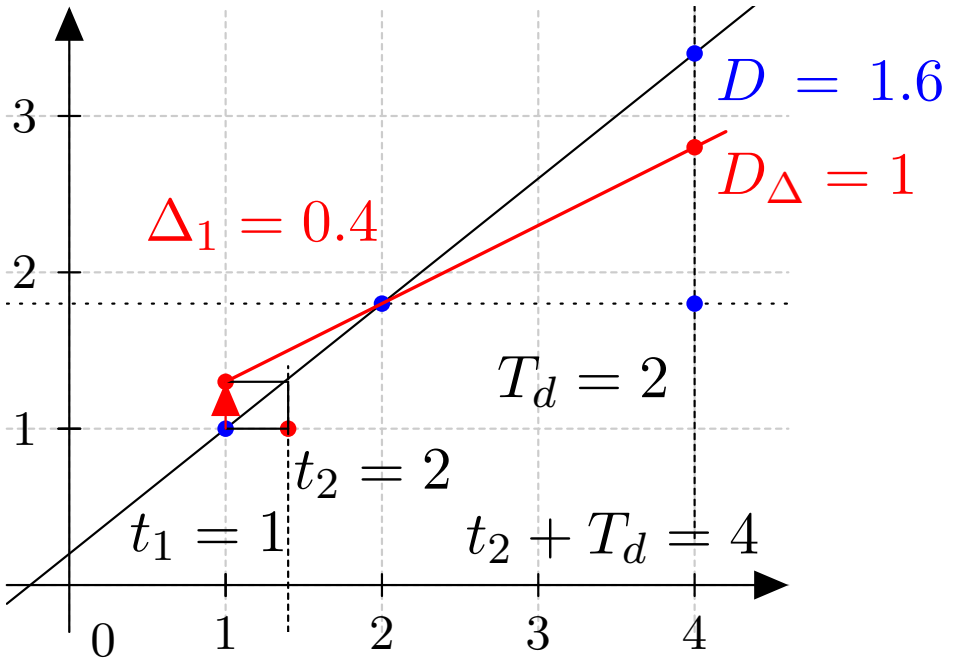


Predicting the output a time T_d into the future.

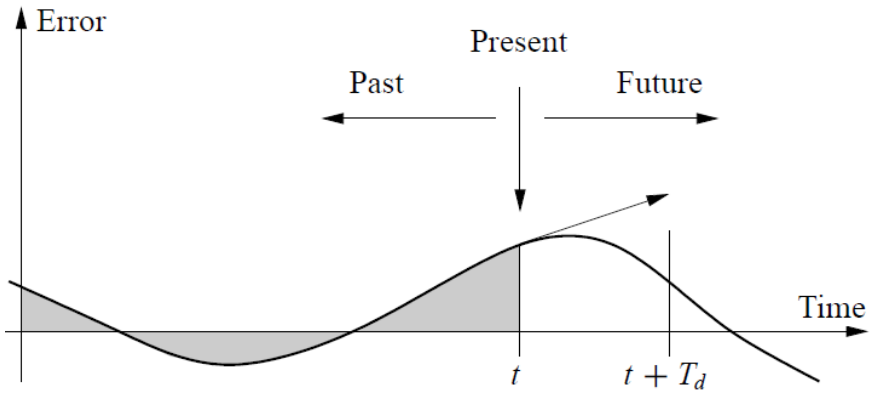


Here, as an example, with sampling period 1 [time unit], the sample at time $t = t_1 = 1$ was delayed by $\Delta = 0.4$ time units (40% of a period), and the predictive D-part at $t = t_2 = 2$ changes accordingly.

Timing errors can result in deficient or unstable control



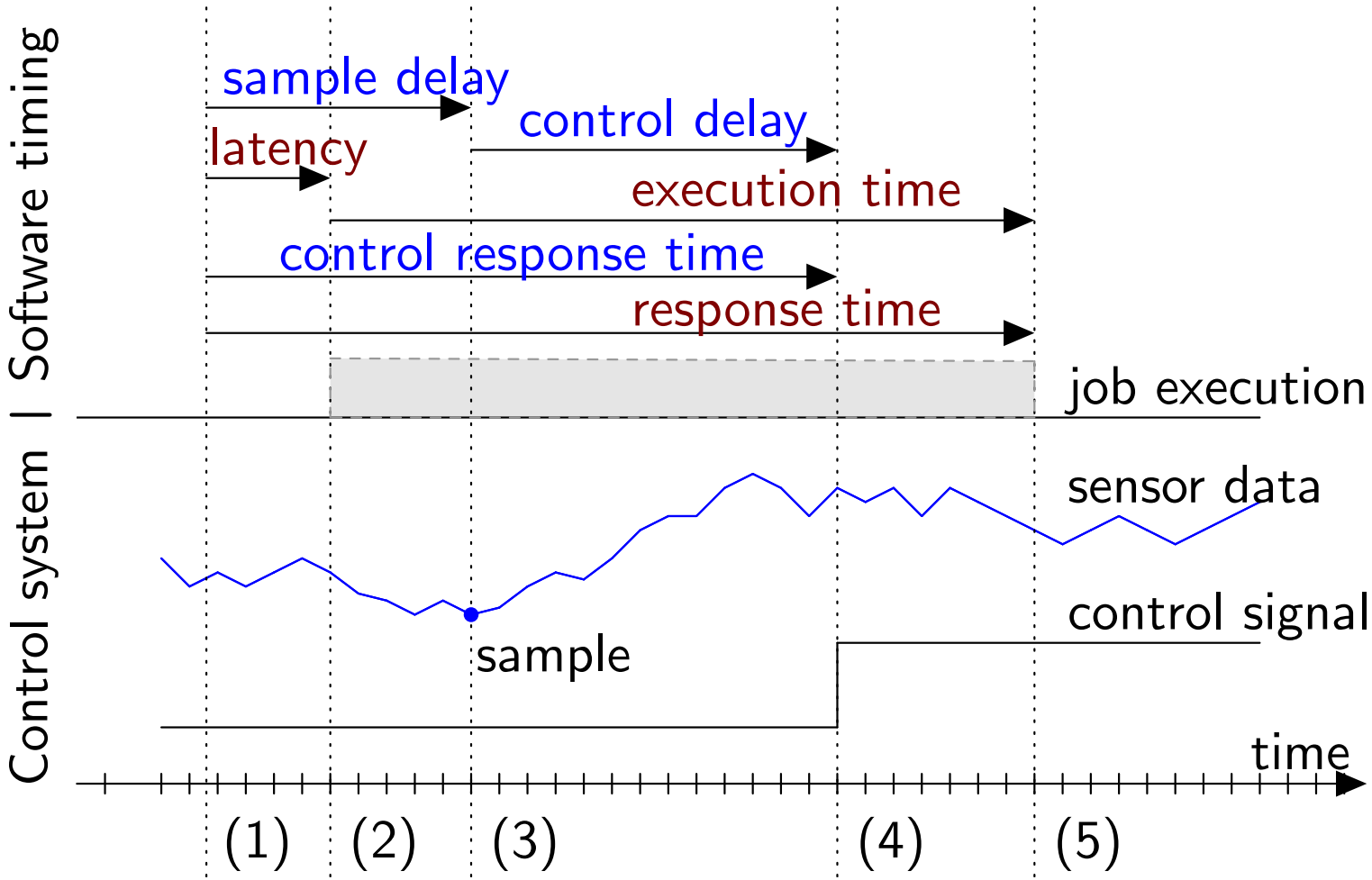
Predicting the output a time T_d into the future.



Here, as an example, with sampling period 1 [time unit], the sample at time $t = t_1 = 1$ was delayed by $\Delta = 0.4$ time units (40% of a period), and the predictive D-part at $t = t_2 = 2$ changes accordingly.

The predictive D-part normally improves damping, but with timing errors the system can even become unstable!!

Latencies and delays in feedback control



Points in time in previous figure

The time-axis markers (1)-(5) in previous slide denote the following events:

1. **Release time** for control job; desired start of period/job.
2. Start time after context switch; invocation of control computation.
3. The sensed output of the controlled process is sampled.
4. The computed control action is output physically.
5. **Response time**; control and execution is completed, including update of control states and any preparation for next sample/job.

The *control delay* and the *control response time* are to be considered in control engineering, whereas (we in this course care about) the *execution time* and the *response time* are considered from a software point of view.

WCET

How can we guarantee a maximum response time?

- Worst-case execution time (WCET)
- Worst-case response time (R)

Since we are to give guarantees we are from now on talking about the worst possible case.

Highest priority thread

Maximum start time; max latency

Time for context switch

Maximum response time

Maximum latency (according to the previous item)
+
Worst-case time to execute the code in the thread
+ For each used resource: maximum blocking time.

Lower priority threads

Maximum start time

Time for context switch
+ Sum WCET for all higher priority threads
+ Sum WCET for other threads with equal priority

Maximum response time

Maximum start time (according to items above)
+
Worst-case time to execute the code in the thread [1]
+ Sum WCET for all higher/equal priority threads (due to preemption)
+ For each used resource: maximum blocking time.

[1] Including time for context switches.

1 Software and timing
- Periodic activities
- The importance of proper timing.
- Worst-case execution time (WCET)
- Worst-case blocking time

2 Scheduling principles
- Static scheduling
- Dynamic scheduling – priority-based
- Dynamic scheduling

3 Bounded blocking time
- Priority inversion and avoidance
- Limiting max blocking despite multiple resources.

Static scheduling

Used for extremely time-critical threads and in simple control systems.

- Time is divided into short slots.
- All activities must be made small enough to fit in one slot.
- All activities are scheduled into time slots in advance.
- Cyclic execution schedule.

Advantages

+ Guaranteed scheduling – on time!
+ An activity can always finish – no critical regions
+ Easy to calculate worst-case response times

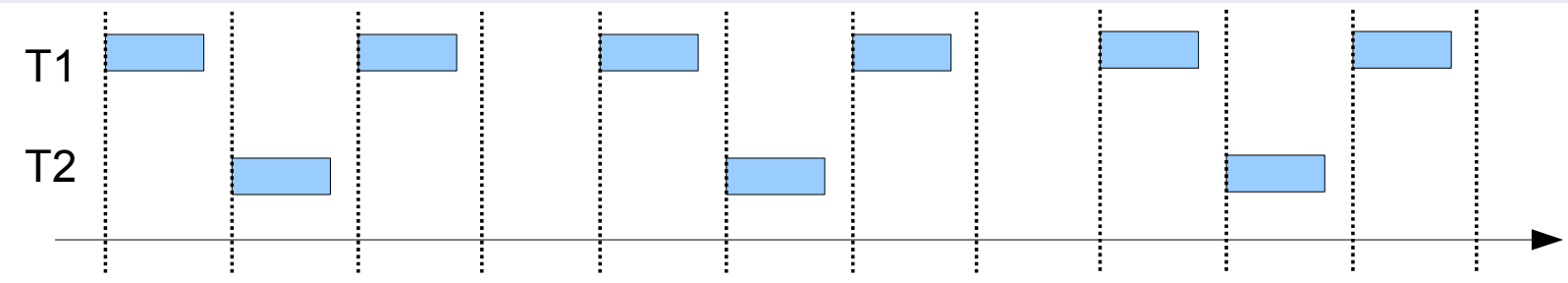
Disadvantages

– Fragmentation, lost CPU time.
– An activity must never use more than one slot. Activities might have to be artificially partitioned.
– Complex schedule which must be redone when program changes.

Static scheduling – examples

Fictive illustrative example

Two threads; **T1:** 50 times/second **T2:** 25 times/second
Divide time into 10 ms slots (e.g., interrupt triggered).
Schedule accomplished by arranging the source code:



Industrial examples

The following were presented at lecture by classic overheads:

- ABB Robotics: DSP-based motor-control implementation
- Saab Aircraft: System computer in fighter jet.

Scheduling analysis

Will all high-priority threads always meet their deadlines?

Strict priority order – the thread with the highest priority within the runnable threads are assigned to the CPU.

Round robin - threads are assigned to the CPU in turn (FIFO).

We must assume strict priorities (and hence, desktop computers are not real-time computers since they use round-robin for application fairness), otherwise we would need to know all threads in the entire system.

Scheduling analysis

Will all high-priority threads always meet their deadlines?

Strict priority order – the thread with the highest priority within the runnable threads are assigned to the CPU.

Round robin - threads are assigned to the CPU in turn (FIFO).

We must assume strict priorities (and hence, desktop computers are not real-time computers since they use round-robin for application fairness), otherwise we would need to know all threads in the entire system.

Are there any rules to how select the priority?

- The priority is determines how the threads will be scheduled.
- For guaranteed max response time, the worst case must be analyzed; $\rightarrow$ scheduling analysis using WCET.

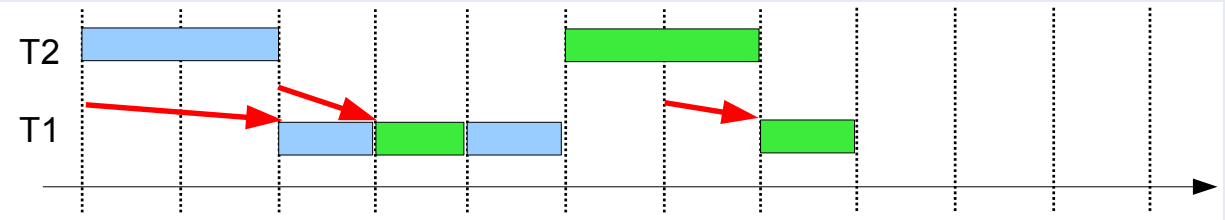
Fixed-priority scheduling

- With dynamic scheduling, which is our default, the run-time system (OS, JVM, ..) determines online what thread (out of the ready ones) that will run.
- Most system schedulers are based on priorities, as reflected in the Java classes.
- We also assume an interrupt driven scheduler, and hence preemption, but otherwise the `java.lang.Thread.yield` method is available for making a re-schedule from the application level.
- If those priorities are fixed, after being assigned at the creation of the thread, we have *fixed-priority scheduling*.
- The remaining question then is: How to determine the priority?

How do we assign priorities?

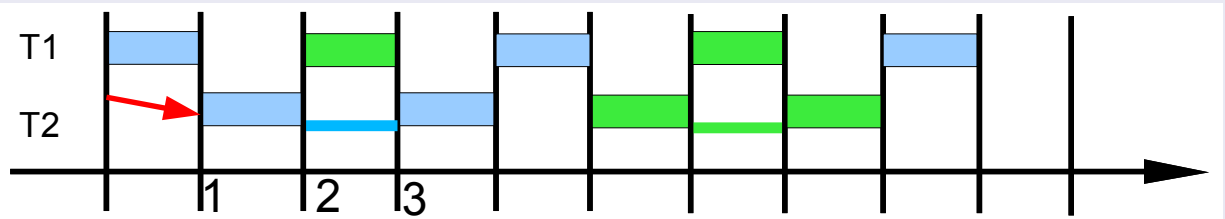
Example: T1 executes 1ms every 2ms, and T2 executes 2ms every 5ms.
Threads should execute once each period (finish before next release).

Give T2 higher priority than T1



Failure: T1 is not allowed to execute first period.

Give T1 higher priority than T2

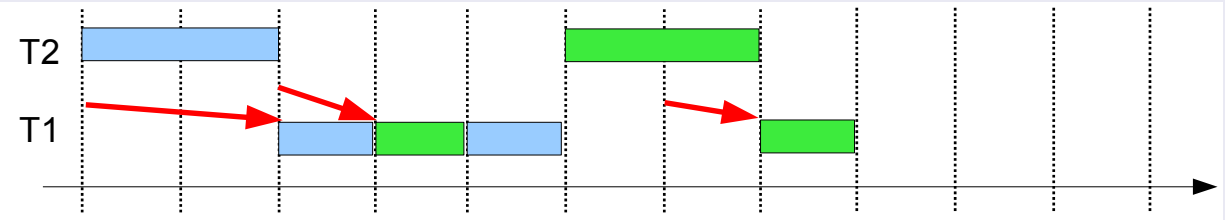


T2 is preempted by T1 at time 2 and 6.

How do we assign priorities?

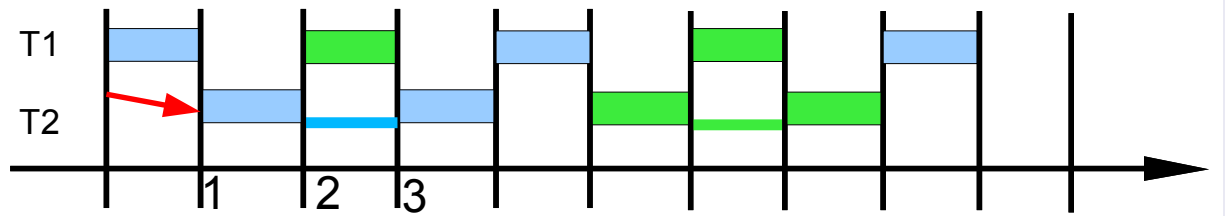
Example: T1 executes 1ms every 2ms, and T2 executes 2ms every 5ms.
Threads should execute once each period (finish before next release).

Give T2 higher priority than T1



Failure: T1 is not allowed to execute first period.

Give T1 higher priority than T2



T2 is preempted by T1 at time 2 and 6.

Scheduling possible using these priorities:
Highest frequency
↕
Highest priority

RMS – Rate Monotonic Scheduling

RMS Rule:
Priority according to period;
Short period $\leftrightarrow$ high priority

- How good is it? Can we say something about **when** it works?
- How much of the CPU time can we use? 100%?

Scheduling analysis: How high CPU utilization can we have and still guarantee schedulability?

RMS – Rate Monotonic Scheduling

RMS Rule:
Priority according to period;
Short period $\leftrightarrow$ high priority

- How good is it? Can we say something about **when** it works?
- How much of the CPU time can we use? 100%?

Scheduling analysis: How high CPU utilization can we have and still guarantee schedulability?

Simplifications:

Initially, we assume:

- Periodic threads
- No blocking
- Deadline = period

RMS – Rate Monotonic Scheduling

RMS Rule:
Priority according to period;
Short period $\leftrightarrow$ high priority

- How good is it? Can we say something about **when** it works?
- How much of the CPU time can we use? 100%?

Scheduling analysis: How high CPU utilization can we have and still guarantee schedulability?

Simplifications:

Initially, we assume:

- Periodic threads
- No blocking
- Deadline = period

Notation

For each thread we know:

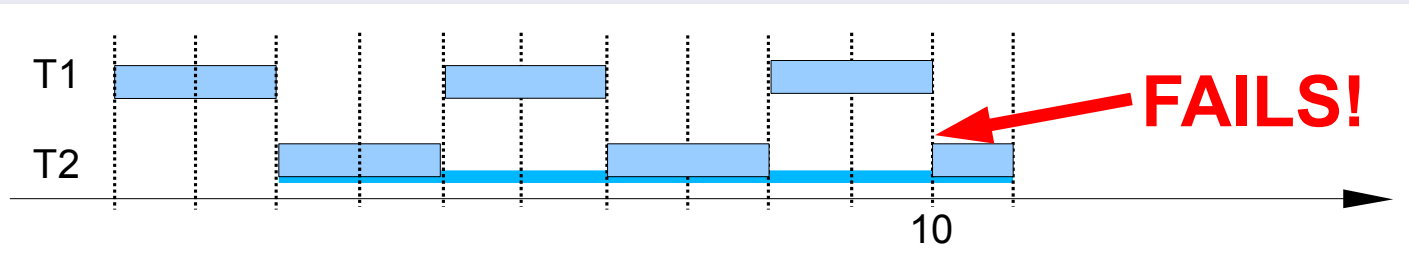
T	=	Period
C	=	Execution time (WCET)
U	=	C/T = CPU utilization

RMS examples

Aiming at 100% CPU load

T1: C=2ms T=4ms C/T=0.5 T2: C=5ms T=10ms C/T=0.5

Scheduling diagram:



But T2: C=2ms T=4ms C/T=0.5 works!

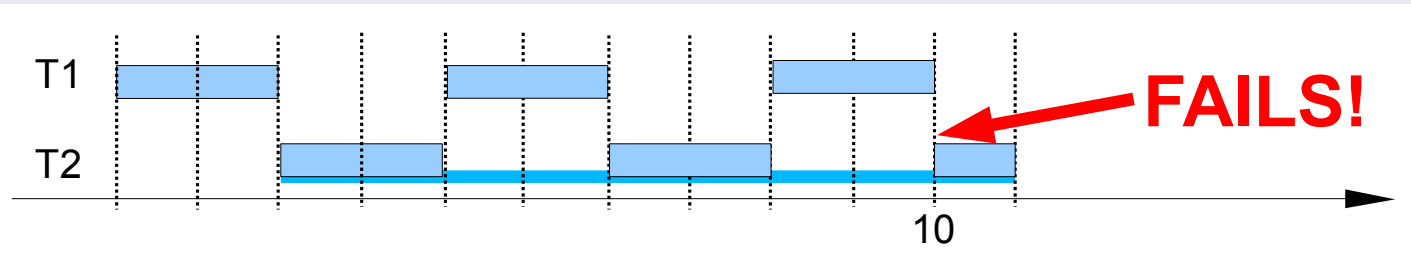
Thus, depends on relationship between the periods.

RMS examples

Aiming at 100% CPU load

T1: C=2ms T=4ms C/T=0.5 T2: C=5ms T=10ms C/T=0.5

Scheduling diagram:



But T2: C=2ms T=4ms C/T=0.5 works!

Thus, depends on relationship between the periods.

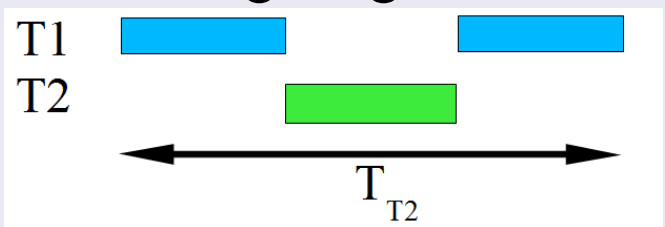
Searching the worst scheduling situation that still works (two threads)

With T1: C=1ms, T=2ms (C/T=0.5).

Then T2: C=1ms, T=3ms yields the lowest C/T while still schedulable.

Total C/T = 1/2+1/3≈ 0.83.

Scheduling diagram:



RMS – Analysis (Liu & Layland, 1973)

Generally possible to guarantee schedulability if (n = number of threads)

$$\sum \frac{C_i}{T_i} < n(2^{1/n} - 1)$$

n=1 U=1
n=2 U $\approx$ 0.83
n=3 U $\approx$ 0.78
n=∞ U $\approx$ 0.69

Note: A system might be schedulable even if the CPU utilization is higher than the bound above. Exact analysis is then required!

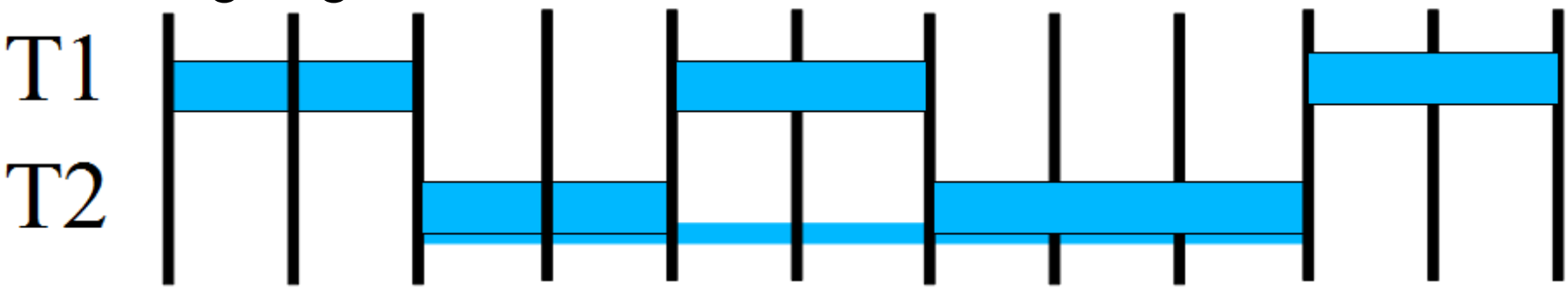
EDF - Earliest Deadline First

Always assign the CPU to the thread which is closest to its deadline.

T1: C=2ms T=4ms

T2: C=5ms T=10ms

Scheduling diagram:



100% CPU usage possible,
but expensive to implement (on top of priority-based schedulers),
and bad behavior at overload (all deadlines missed).

1 Software and timing
- Periodic activities
- The importance of proper timing.
- Worst-case execution time (WCET)
- Worst-case blocking time

2 Scheduling principles
- Static scheduling
- Dynamic scheduling – priority-based
- Dynamic scheduling

3 Bounded blocking time
- Priority inversion and avoidance
- Limiting max blocking despite multiple resources.

Response times in case of temporary blocking

Real-time threads should run according to absolute priority order. A high-priority thread should not wait for lower priority threads an arbitrary long time.

Therefore:

- The CPU is always allocated to the highest priority thread.
- Semaphores/monitors with priority queues.
- Shared monitor "enter" queue for new and previously blocked threads.

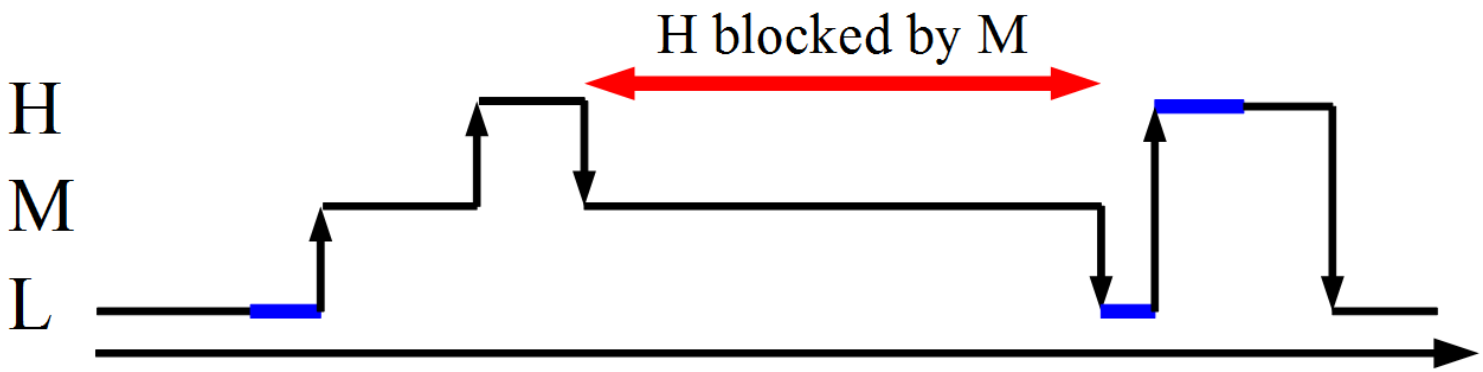
Can we ensure timing for high-priority threads without knowing all low or medium priority threads? What about blocking on shared resources?

Priority Inversion

A problematic scenario

Three threads with H(igh), M(edium), and L(ow) priority:

1. L executes and enters a critical region.
2. M preempts and starts executing.
3. H preempts and tries to allocate the shared resource. H is blocked.
4. M continues executing for an arbitrary long period of time, blocking both L and H!



How do we avoid priority inversion?

The cure against priority inversion: *Priority inheritance protocols*

General idea

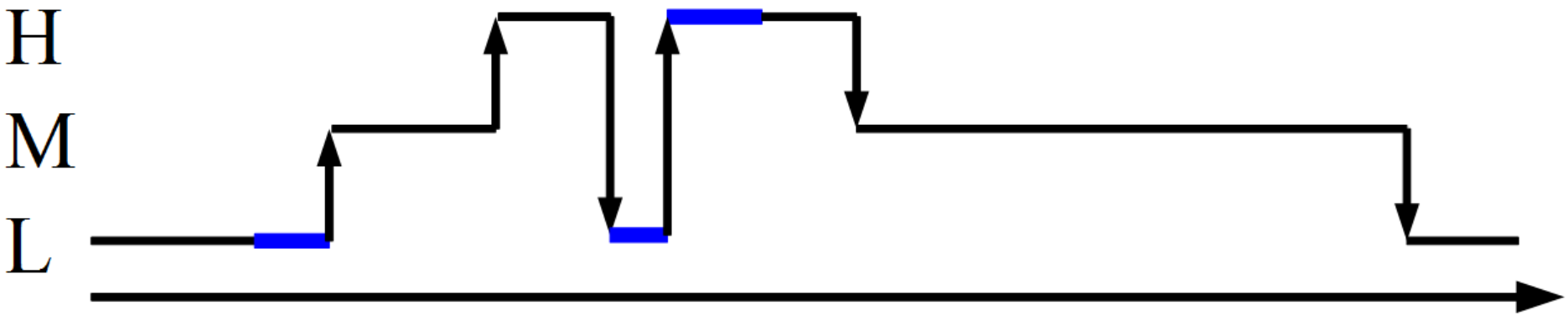
Temporarily (by the run-time system during blocking) raise the priority of threads holding resources needed by higher-priority threads.

Protocols:

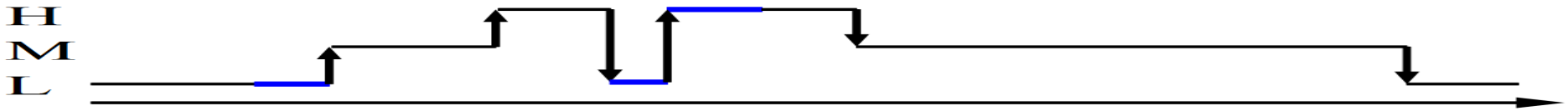
- Basic priority inheritance (swe: dynamiskt prioritetsarv)
- Priority-ceiling protocol
- Immediate inheritance protocol

Basic Priority Inheritance Protocol

- When a thread is blocked, the priority of the thread holding the requested resource is (temporarily) raised to the priority of the blocked thread.
- Consider the three threads, having Low, Medium, and High priority.
- Thread L and H share a resource/monitor.
- Recall that thread M is for something else, not using the monitor, and perhaps its existence in unknown.



Basic Priority Inheritance Protocol – caption



The scenario cured by priority inheritance

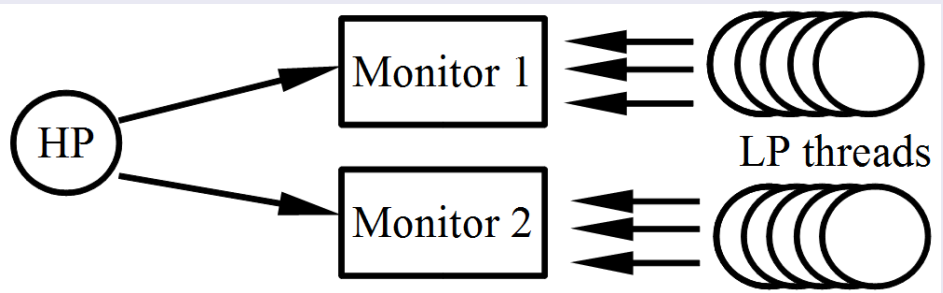
1. L executes and enters a critical region.
2. M preempts and starts executing.
3. H preempts and starts executing.
4. H tries to allocate the resource held by L, H is blocked. L inherits the priority of H and completes the critical region.
5. Leaves the critical region and its priority is lowered. H is given access to the critical region.
6. H finish and M continues executing.
7. Neither M or H is ready to execute. L continues.

Problem: Multiple blocking

WCET for the highest priority thread

Worst-case time to execute the code in the thread
+ For each used resource: maximum blocking time

Basic Inheritance Protocol:



Can block once for each used resource.

Enhanced priority inheritance to avoid multiple blocking

Default: Basic Inheritance Protocol – raise prio for LP temporarily, dynamically when HP blocked, locally for each resource.

Two enhancements:

Priority Ceiling

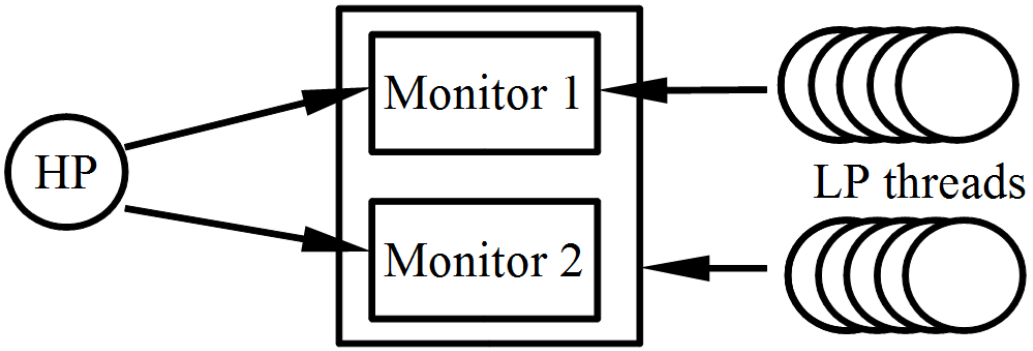
Allow only one LP to access the resources required by HP at any time.

Immediate Inheritance

The prio of the LP is always raised to the ceiling prio in critical region.

Properties

The effect is, to the cost of managing (and registering them in the source code) multiple resources together, a fence around that set of resources.



Extra benefit: Deadlock free!