# Embedded Systems Programming - PA8001

http://goo.gl/YdEcZU

Lecture 5

Mohammad Mousavi

m.r.mousavi@hh.se

HALMSTAD UNIVERSITY

Center for Research on Embedded Systems

School of Information Science, Computer and Electrical Engineering

# Real Time?

In what ways can a program be related to time in the environment (the *real time*)?

Salvador Dali, The Persistence of Memory.

# Real Time

An external process to . . .

- Sample: reading a clock,
- React: a handler for an interrupt clock, and
- Constraint: a deadline to respect.

# Sampling the time

Requires a hardware clock (read as an external device)

## Multitude of alternatives

- Units? Seconds? Milliseconds? CPU cycles?
- Since when? Program start? System boot? Jan 1, 1970?
- Real time? Time stops when: other threads are running? when CPU sleeps? Time that cannot be set and always increases?

# Timestamps

Relative timing: prevalent in reactive systems, reactions are relative to events

### Example

Teacher left 15 min. after the start of the lecture.

In embedded programming, time-stamping an event: reading performed around the event detection.

# Time spans

The difference between two time-stamps: a time span independent of the nominal clock values (modulo clock resolution).

## The meaning of time-stamp

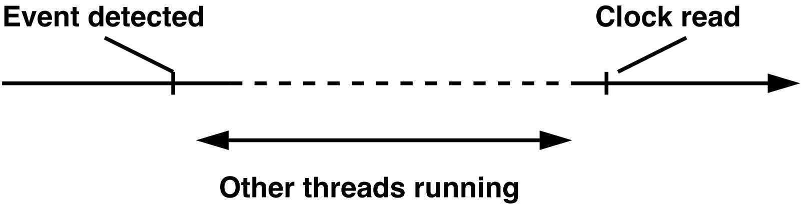
- The time of some arbitrary program instruction?
- The beginning or end of a function call?
- The time of sending or receiving an asynchronous message?

Too much program dependent!

# In a scheduled system

What looks like . . .


might very well be . . .




Close proximity is not the same as subsequent statements!

# Time-stamping events

Solution: to time-stamp events that *drive* a system

### Idea!

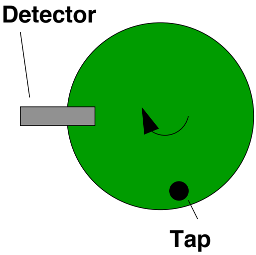
Read the clock in the interrupt handler detecting the event

- Disable other interrupts, hence no threads might interfere
- Tight predictable upper bound on the time-stamp error

# Example

## Calculate the speed

For a rotating wheel, measuring the time between two subsequent detections of a passing tap.




```
typedef struct{
 Object super;
 int previous;
 Other *client;
} Speedo;
...
Speedo speedo;
int main(){
   INSTALL(&speedo, detect, SIG_XX);
   return TINYTIMBER(...)
}
```

# Example

## Calculate the speed

For a rotating wheel, measuring the time between two subsequent detections of a passing tap.

```
int detect(Speedo *self, int sig){
   int timestamp = TCNT1;
   ASYNC(self -> client,
        newSpeed,
        PERIMETER/DIFF(timestamp,self->previous));
   self->previous=timestamp;
}
```

`DIFF` will have ot take care of timer overflows!

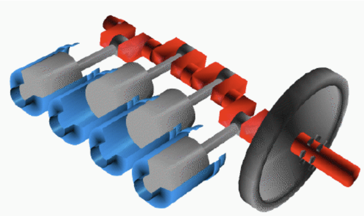
# Real-time events to react to

So far: how to sample the real-time clock to know about time

Now: how to take action after a certain amount of time

**Example**

The wheel is an engine crankshaft and we have to emit ignition signals to each cylinder



How to postpone program execution until certain time

# Reacting to real time events

## Very poor man's solution

Consume a fixed amount of CPU cycles in a (silly) loop

```
int i;
for(i=0;i<N;i++);  // wait
do_future_action();
```

## Problems

1. Determine `N` by testing!
2. `N` will be highly platform dependent!
3. A lot of CPU cycles will simply be wasted!

# Reacting to real time events

## The nearly as poor man's solution

Configure a timer/counter with a known clock speed, and busy-wait for a suitable time increment

```
unsigned int i = TCNT1+N;
while(TCNT1<i); // wait
do_future_action();
```

## Problems

1. Determine `N` by calculation
2. Still a lot of wasted CPU!

# Reacting to real time events

## The standard solution

Use the OS to *fake* busy-waiting

```
delay(N);   // wait (blocking OS call)
do_future_action();
```

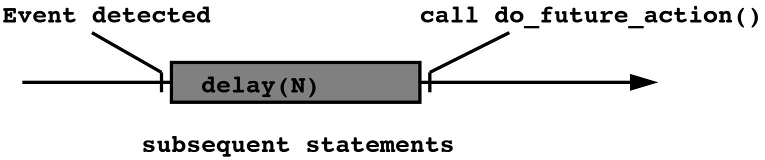
- No platform dependency!
- No wasted CPU cycles (at the expense of a complex OS)

Still a problem ...

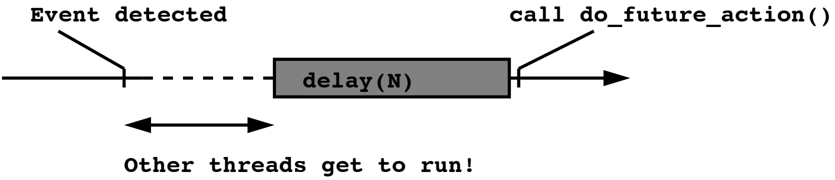
... common to all solutions ...

# In a scheduled system

What looks like . . .




might very well be . . .




Had we known the scheduler's choice, a smaller `N` had been used!

# Relative delays

The problem: relative time without fixed references:

- The constructed real-time event will occur at after `N` units from *now*.
- What is *now*?!

Other common OS services share this problem: `sleep`, `usleep` and `nanosleep`.

We are not going to use OS services in the course.

# Yet another problem

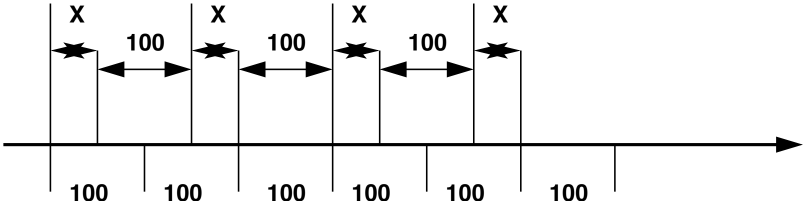
Threads and interleaving make it worse

### Example

Consider a task running a CPU-heavy function `do_work()` every 100 millisecods. The naive implementation sing `delay()`:

```
while(1){
  do_work();
  delay(100);
}
```

# Accumulating drift

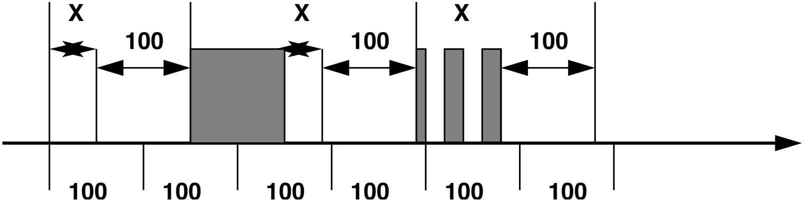



X is the time take to `do_work`

Each turn takes at least 100+X milliseconds.

A drift of X milliseconds will accumulate every turn!

# Accumulating drift




With threads and interleaving, the bad scenario gets worse!

Even with a known `X`, delay time is not predictable.

# A stable reference

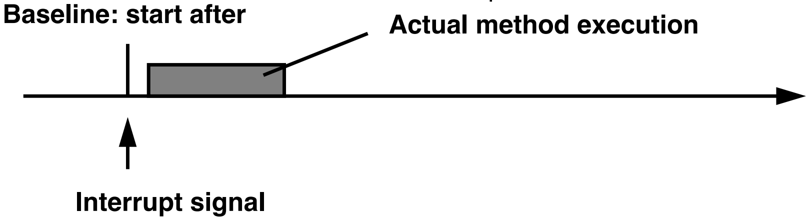
What we need is a stable time reference to use as a basis whenever we specify a relative time (instead of now).

## Baselines

We introduce the baseline of a message to mean the earliest time a message is allowed to start.

## Time stamps of interrupts!

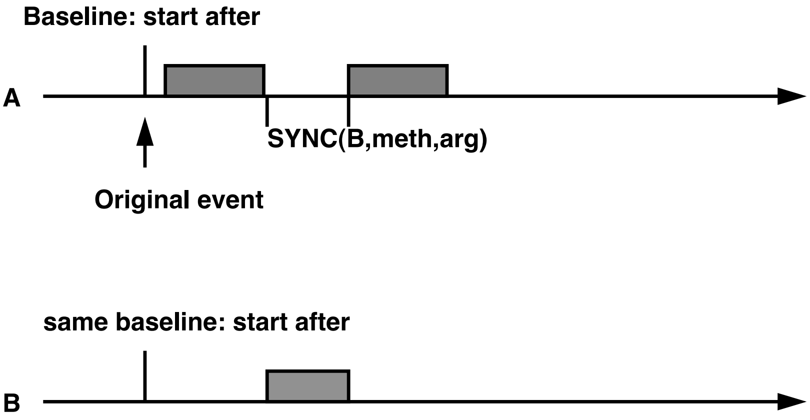
The baseline of an event is its time-stamp:



# A stable reference

## SYNC

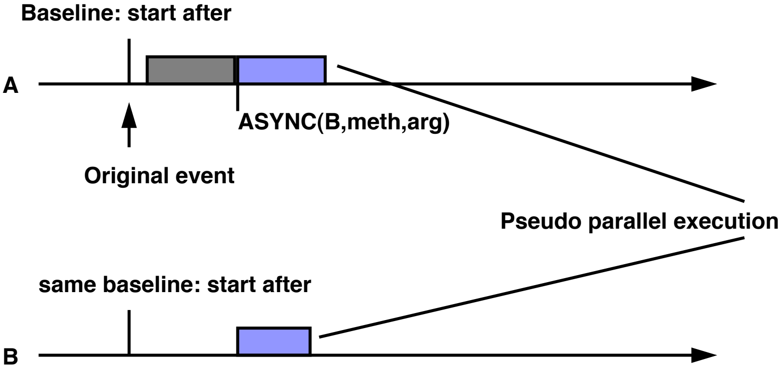
Calling methods with `SYNC` doesn't change the baseline (the call inherits the baseline)



# A stable reference

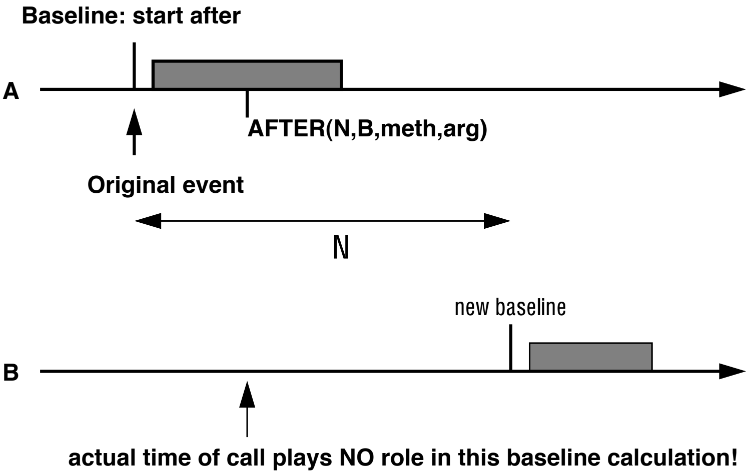
## ASYNC

By default `ASYNC` method calls will inherit the baseline



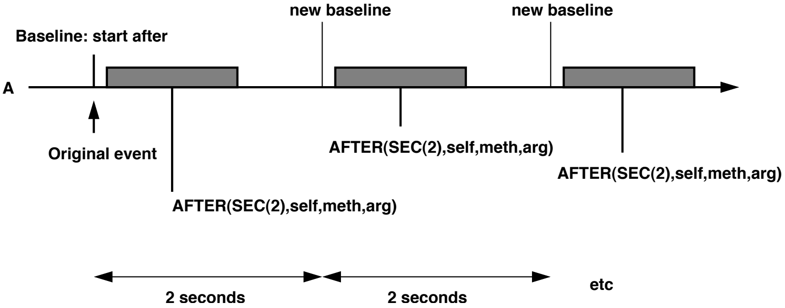
# A stable reference

For `ASYNC` we may also consider adding a baseline offset `N`!



# Periodic tasks

To create a cyclic reaction, simply call **self** with the same method and a new baseline:




SEC is a convenient macro that makes the call independent of current timer resolution.

# Implementing `AFTER`

1. Let the baseline be stored in every message (as part of the `Msg` structure)
2. `AFTER` is the same as `ASYNC`, but
   - New baseline is

     `MAX(now, offset+current->baseline)`
   - If `baseline > now` , put message in a timerQ instead of readyQ
   - Set up a timer to generate an interrupt after earliest baseline
   - At each timer interrupt, move first timerQ message to readyQ and configure a new timer interrupt

In fact `ASYNC` can now be defined as

```
#define ASYNC(to,meth,arg) AFTER(0,to,meth,arg)
```

# Priority assignment

## Question

How do we set thread/message priority for the purpose of meeting deadlines?

## Static priorities

Assign a fixed priority to each thread and keep it constant until termination.

## Dynamic priorities

Determine the priority at run-time from factors such as the time remaining until deadline.

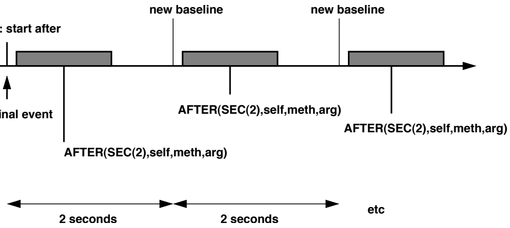
## :-(

In neither case a method exists that is both predictable and generally applicable to all programs!

## :-)

It is possible to get by if we concentrate on programs of a restricted form.

# Initial restricted model




- Only periodic reactions
- Fixed periods
- No internal communication
- Known, fixed WCETs
- Deadlines = periods

If time allows, we will discuss how to remove these restrictions.

# Static priorities – method

## Rate monotonic (RM)

Under the given assumptions, there exists a static priority assignment rule that is really simple

The shorter the period, the higher the priority

d For RM, the actual priority values do not matter, only their relative order.

Because of our inverse priority scale, we can simply implement RM by letting $P_i = D_i$ $(=T_i)$

# RM example

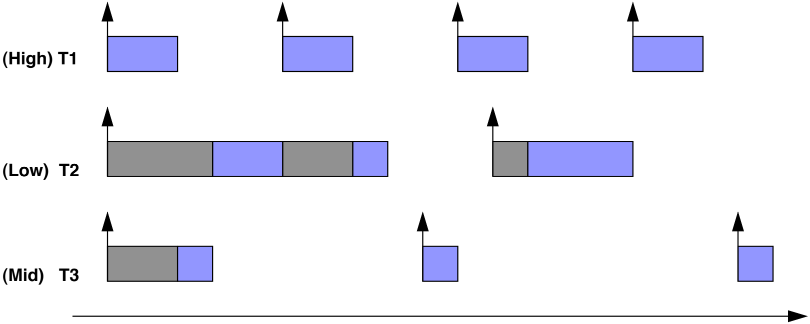
Given a set of periodic tasks with periods

$$T1 = 25ms$$
$$T2 = 60ms$$
$$T3 = 45ms$$

Valid priority assignments

| | | |
|---|---|---|
| $P1 = 10$ | $P1 = 1$ | $P1 = 25$ |
| $P2 = 19$ | $P2 = 3$ | $P2 = 60$ |
| $P3 = 12$ | $P3 = 2$ | $P2 = 45$ |

# RM example




Period = Deadline. Arrows mark start of period.
Blue: running. Gray: waiting.

# Dynamic priorities – method

## Earliest Deadline First – EDF

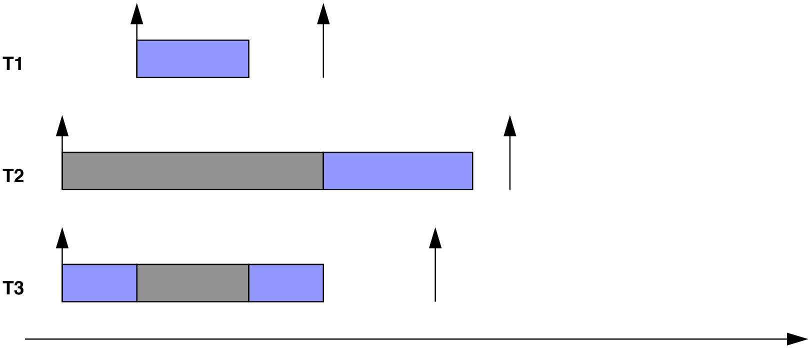
Dynamic priority assignment rule:

> The shorter the time remaining until deadline, the higher the priority

To use absolute deadlines: priorities = remaining clock cycles (before missing the deadline)

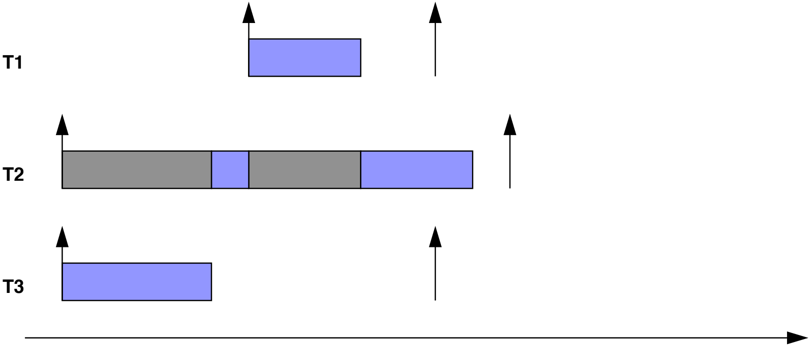
Under EDF, each activation n of periodic task i will receive a new priority: $P_{i(n)} = \mathrm{baseline}_{i(n)} + D_i$

# EDF example



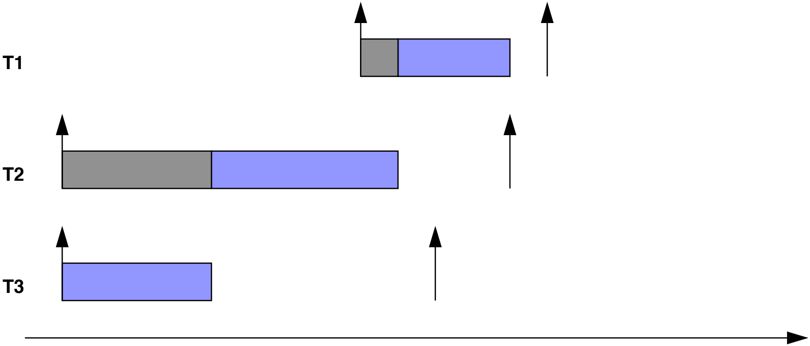

T1 arrives later, but its deadline is earlier than both T2's and T3's absolute deadlines!

# EDF example




Deadline of T1 $<$ Deadline of T2

# EDF example




(absolute) Deadline of T1 > (absolute) Deadline of T2

# Optimality

Multiple ways assigning priorities to meet deadlines

Optimal: a method which fails only if every other method fails

- RM is optimal among static assignment methods
- EDF is optimal among dynamic methods

# Schedulability

An optimal method may also fail
A set of task may not be schedulable at all

## Example

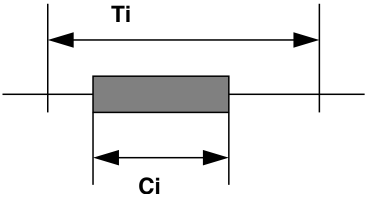
The shortest path from A to B is 200km (the optimal scheduling).
We have only one hour to reach the destination and the maximum speed is 120 km/h (deadline and platform constraints).
Can we be there on time (schedulability analysis)

# Schedulability

To determine whether task set is at all schedulable (with optimal methods)

Schedulability must take the WCETs of tasks into account.

# Utilization-based analysis




For a periodic task set, an important measure is how big a fraction of each turn a task is actually using the CPU.

That is, the CPU utilization of a periodic task $i$ is the ratio $\frac{C_i}{T_i}$, where $C_i$ is the WCET and $T_i$ is the period.

Note

Any task for which $C_i = T_i$ will effectively need exclusive access to the CPU!

# Utilization-based analysis (RM)

Given a set of simple periodic tasks, scheduling with priorities according to RM will succeed if

$$U \equiv \sum_{i=1}^{N} \frac{C_i}{T_i} \leq N(2^{1/N} - 1)$$

where N is the number of threads.

That is, the sum of all CPU utilizations must be less than a certain bound that depends on N.

# Utilization bounds

| N | Utilization bound |
|---|---|
| 1 | 100.0 % |
| 2 | 82.8 % |
| 3 | 78.0 % |
| 4 | 75.7 % |
| 5 | 74.3 % |
| 10 | 71.8 % |

Approaches 69.3% asymptotically

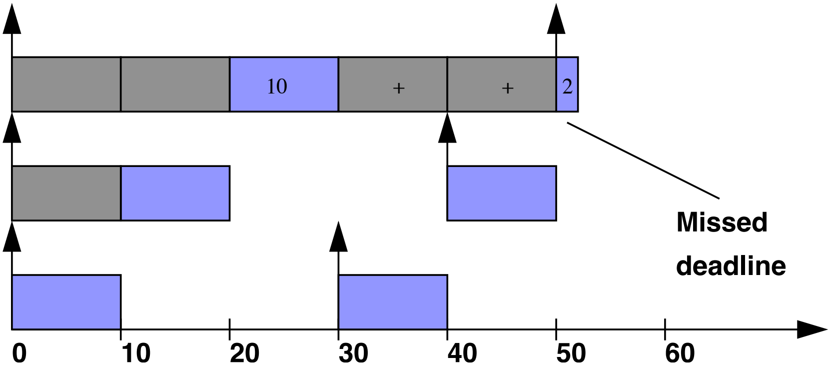
# Example A

| Task | Period | WCET | Utilization |
|---|---|---|---|
| $i$ | $T_i$ | $C_i$ | $U_i$ |
| 1 | 50 | 12 | 24% |
| 2 | 40 | 10 | 25% |
| 3 | 30 | 10 | 33% |

The combined utilization U is 82%, which is above the bound for 3 threads (78%).

The task set fails the utilization test.

# Time-line for example A



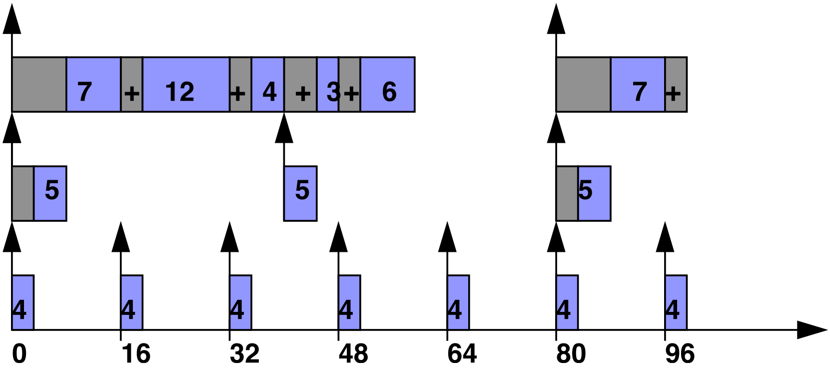
# Example B

| Task | Period | WCET | Utilization |
|---|---|---|---|
| $i$ | $T_i$ | $C_i$ | $U_i$ |
| 1 | 80 | 32 | 40% |
| 2 | 40 | 5 | 12.5% |
| 3 | 16 | 4 | 25% |

The combined utilization U is 77.5%, which is below the bound for 3 threads (78%).

The task set will meet all its deadlines!

# Time-line for example B



# Example C

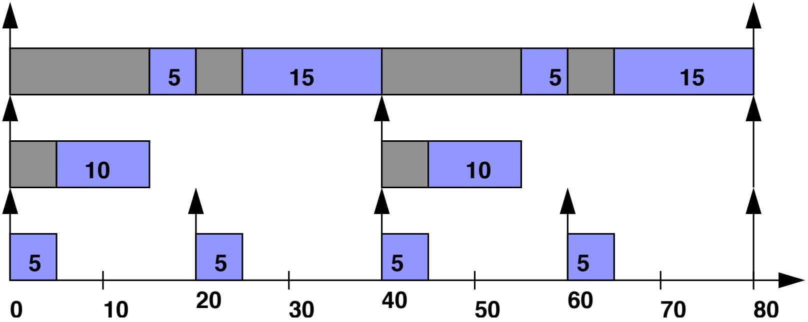
| Task | Period | WCET | Utilization |
|---|---|---|---|
| $i$ | $T_i$ | $C_i$ | $U_i$ |
| 1 | 80 | 40 | 50% |
| 2 | 40 | 10 | 25% |
| 3 | 20 | 5 | 25% |

The combined utilization U is 100%, which is well above the bound for 3 threads (78%).

However, this task set still meets all its deadlines!

How can this be??

# Time-line for example C



# Characteristics

## The utilization-based test

- Is sufficient (pass the test and you are OK)
- Is not necessary (fail, and you might still have a chance)

## Why bother with such a test?

- Because it is so simple!
- Because only very specific sets of tasks fail the test and still meet their deadlines!

# Utilization-based analysis (EDF)

Given a set of simple periodic tasks, scheduling with priorities according to EDF will succeed if

$$U \equiv \sum_{i=1}^{N} \frac{C_i}{T_i} \leq 1$$

That is, the sum of all CPU utilizations must be less than or equal 100%, independent of the number of tasks.

Unlike the case for RM, the utilization-based test for EDF is both sufficient and necessary (demand more than 100% of the CPU and you are bound to fail!)

# EDF vs RM

## Similarities

- Both algorithms are optimal within their class
- Both are easy to implement in terms of priority queues
- Both have simple utilization-based schedulability tests
- Both can be extended in similar ways

## Advantages of EDF

- Close relation to terminology of real-time specifications
- Directly applicable to sporadic, interrupt-driven tasks
- superior CPU utilization

# EDF vs RM

## Drawbacks of EDF

- It exhibits random behaviour under transient overload (but so does RM, in fact, in a different way)
- RM predictably skips low priority tasks under constant overload (but EDF rescales task priorities instead)
- Utilization-based test becomes more elaborate for EDF when $D_i \leq T_i$ (but is still feasible)
- Operating systems generally don't support it (priority scales lack granularity, no automatic time-stamping)
- Few languages allow for natural deadline constraints

However, for reactive objects, EDF fits nice as an alternative to RM

# Implementation (RM)

In `TinyTimber.c`

```
struct msg_block{
  ...
  Time baseline;
  Time priority;
  ...
};

void async(Time offset, Time prio,
           OBJECT *to, METHOD meth, int arg){
   ...
   m->baseline=MAX(TIMERGET(),
                   current->baseline+offset);
   m->priority = prio;
   ...
}
```

# Implementation (EDF)

In `TinyTimber.c`

```
struct msg_block{
  ...
  Time baseline;
  Time deadline;
  ...
};

void async(Time BL, Time DL,
           OBJECT *to, METHOD meth, int arg){
   ...
   m->baseline=MAX(TIMERGET(),
                   current->baseline+BL);
   m->deadline = m->baseline+DL;
   ...
}
```

# Loosening the assumptions

### $T_i \neq D_i$

Deadlines less than periods: infrequent, urgent tasks

### Sporadic Tasks

Sporadic tasks: no fixed period (interrupt handlers), urgent deadlines

# Deadline Monotonic

## Basic Principle

$C_i < D_i < T_i$

Lower deadline values get higher priority: a priority assignment is valid when $P_i < P_j$ iff $D_i < D_j$.

## Naive Schedulability Analysis

$$U \equiv \sum_{i=1}^{N} \frac{C_i}{D_i} \leq N(2^{1/N} - 1)$$

# More Precise Schedulability Analysis

## Pre-Processing

Sort the tasks by increasing order of deadlines:

$$i < j \text{ iff } D_i < D_j$$

## Schedulability Analysis

For each and every $i \leq n$:

$$C_i + \sum_{j=1}^{i-1} \left\lceil \frac{D_j}{T_j} \right\rceil C_j \leq D_i$$

# Loosening the assumptions

## Sporadic Tasks

Sporadic tasks: no fixed period (interrupt handlers), urgent deadlines

Characteristics needed for schedulability analysis

## Characteristics

Minimum inter-arrival time: minimum time between two events causing sporadic tasks (e.g., key strokes, signal updates)

Period $T$ interpreted as inter-arrival time

For sporadic tasks: $D < T$

# Loosening the assumptions

## Sporadic Tasks

Sporadic tasks: no fixed period (interrupt handlers), urgent deadlines

Characteristics needed for schedulability analysis

## Characteristics

Minimum inter-arrival time: minimum time between two events causing sporadic tasks (e.g., key strokes, signal updates)

Period $T$ interpreted as inter-arrival time

For sporadic tasks: $D < T$

# Scheduling Sporadic Tasks

## Polling Servers

A task with period $T_s$

Fixed capacity $C_s$

## Scheduling

Sporadic events scheduled in the server when there is capacity left

Capacity is replenished every $T$ units

# Scheduling Sporadic Tasks

## Polling Servers

A task with period $T_s$
Fixed capacity $C_s$

## Scheduling

Sporadic events scheduled in the server when there is capacity left
Capacity is replenished every $T$ units

# Polling Servers

## Schedulability Analysis

$$U \equiv \frac{C_s}{T_s} + \sum_{i=1}^{N} \frac{C_i}{T_i} \leq (N+1)(2^{1/(N+1)} - 1)$$

# More on real-time

### Other analysis

Response-time analysis: more powerful technique than utilization based

More on this in specialized courses on real-time (such as distributed real time systems)

# More on real-time

## Other analysis

Response-time analysis: more powerful technique than utilization based

More on this in specialized courses on real-time (such as distributed real time systems)