Real-time Systems SMD138

Lecture 10:
Scheduling & feasibility
(Burns & Wellings ch. 13)
Recap: timely reaction

Initial event

- baseline "start after"
- deadline "finish before"
Recap: late reaction

Where will this reaction deadline be defined?

In informal comments only?

Or in concrete source code?
Recap: timely reaction

ASYNC(&B, meth, arg)

baseline
"start after"
deadline
"finish before"

same baseline
same deadline

Pseudo-parallel execution
Recap: late reaction

ASYNC(&B, meth, arg)

baseline “start after”

deadline “finish before”

same baseline

same deadline

Pseudo-parallel execution

But what if B actually needs a deadline of its own?
Adjusted deadlines

baseline
“start after”

BEFORE(dl, &B, meth, arg)

same baseline

new deadline

dl
Late reaction

Baseline: "start after"
Deadline: "finish before"

BEFORE(dl, &B, meth, arg)

same baseline

new deadline

dl
Deadlines and AFTER

AFTER(bl, &B, meth, arg)

baseline "start after"

same deadline

new baseline

deadline "finish before"

bl
Deadlines and AFTER

AFTER(bl, &B, meth, arg)

**baseline**  
“start after”

**deadline**  
“finish before”

new baseline

same deadline

$$\text{MAX}( \text{now, current->baseline + bl} )$$
Deadlines and AFTER

A

baseline
“start after”

deadline
“finish before”

WITHIN(bl, dl, &B, meth, arg)

new baseline

new deadline

B

bl
dl
Late reaction

WITHIN(bl, dl, &B, meth, arg)

"start after"

"finish before"

new baseline

new deadline

new baseline

new deadline
Interrupt handler deadline

Note: interrupt handlers are scheduled by the CPU hardware; i.e., they will run as fast as possible without regard to any deadline!
Expressing deadlines

In TinyTimber.h:

```
#define BEFORE( dl, to, meth, arg ) \n    WITHIN( -1, dl, to, meth, arg );

#define AFTER( bl, to, meth, arg ) \n    WITHIN( bl, -1, to, meth, arg );

#define ASYNC( to, meth, arg) \n    WITYHIN( -1, -1, to, meth, arg )

#define WITHIN( bl, dl, meth, arg) \n    async( bl, dl, (Object*)to, (Method)meth, arg )
```

Default values for interrupt handlers:

```
baseline = timestamp, deadline = infinity (encoded as 0)
```
Deadlines and priorities

- Using **BEFORE**, we can both define the deadline for a chain of reactions to an external interrupt, and fork off a new chain of reactions with its own deadline at any point.

- Inside the kernel, the priorities used will determine in which order messages are scheduled, and hence affect the time when a reaction is able to complete.

- **Core question:** what will be the preferred relation between deadlines and priorities?
How do we set thread/message priorities for the purpose of meeting deadlines?

Two major approaches:

1. **Static** priorities — assign a fixed priority to each thread/message, and keep it constant until termination

2. **Dynamic** priorities — determine the priority at runtime from factors such as time remaining until deadline

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

However, we’ll get by if we concentrate on programs of a restricted form
Initial restricted model

That is:
- Only periodic reactions
- Fixed periods
- No internal communication!
- Known, fixed WCETs
- Deadlines = periods

We'll remove some of these restrictions later...
Each reactive object \( obj \) executes a message (thread/task/job) \( m_i \) in a periodic fashion.

For each message \( m_i \) we know:

- its period \( T_i \) (given, determines the \( \text{AFTER} \) offset)
- its WCET \( C_i \) (measured, or analyzed)
- its relative deadline \( D_i \) (given, equal to \( T_i \) for now)

We want to determine the priority \( P_i \) for each message.
In concrete code

void ignite() {
    BEFORE( D1, &obj1, m1, arg1 );
    BEFORE( D2, &obj2, m2, arg2 );
    \vdots
    BEFORE( Dn, &objn, mn, argn );
}

STARTUP( ignite() );

Each Di = Ti

Classi obji = newClassi();

int mi( Classi *self, int arg ) {
    // read ports
    // compute
    // update self state
    // write ports
    WITHIN( Ti, Di, self, mi, arg );
}
Under the given assumptions, there exists a static priority assignment rule that is really simple:

- “The shorter the period, the higher the priority”

This rule is called Rate Monotonic Priority Assignment, or RM for short.

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
  - $T_1 = 25$ ms
  - $T_2 = 60$ ms
  - $T_3 = 45$ ms
- Examples of valid RM priority assignments are

  $P_1 = 10$  $P_1 = 1$  $P_1 = 25$
  $P_2 = 19$  $P_2 = 3$  $P_2 = 60$
  $P_3 = 12$  $P_3 = 2$  $P_3 = 45$
RM example

Period = relative deadline

Absolute deadline
Under the given assumptions, there exists a dynamic priority assignment rule that is really simple:

• “The shorter the time remaining until deadline, the higher the priority”

This rule is called Earliest Deadline First, or EDF for short.

Because EDF will want to distinguish between messages on basis of their absolute deadlines, priority values must use the same units as the system clock.

Under EDF, each activation n of periodic task i will receive a new priority: \( P_i(n) = \text{baseline}_i(n) + D_i \)
EDF example

Arrives later, but absolute deadline is earlier!

T1

T2

T3

Relative deadline

Absolute deadline
EDF example

Deadline of T1 < deadline of T2

Relative deadline

Absolute deadline
EDF example

Deadline of T1 > deadline of T2 & T3

T1

T2

T3

Relative deadline

Absolute deadline
Under some given assumptions, there might be several ways of assigning priorities so that deadlines are met.

Clearly, a method that only fails if every other method also fails is preferred — such a method is called optimal.

RM is optimal among static priority assignment methods.

EDF is optimal among dynamic methods.

However, knowing that a priority assignment is the best one possible is not the same thing as knowing that it is “good enough”; i.e., knowing that deadlines actually will be met.
Assume all we know is that our priority assignment method is optimal. This is like knowing where the shortest path from A to B lies, but still not knowing if that path is short enough so that B can be reached in time.

To answer whether our tasks will actually meet their deadlines at run-time, we need to determine if our task set is at all schedulable (recall that an optimal priority assignment method will produce a successful schedule if such a schedule exists).

Clearly, the question of schedulability must take the WCETs of tasks into account.
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 $C_i / T_i$, where $C_i$ is the WCET and $T_i$ is the period.

Note that 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

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

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

<table>
<thead>
<tr>
<th>N</th>
<th>Utilization bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100.0 %</td>
</tr>
<tr>
<td>2</td>
<td>82.8 %</td>
</tr>
<tr>
<td>3</td>
<td>78.0 %</td>
</tr>
<tr>
<td>4</td>
<td>75.7 %</td>
</tr>
<tr>
<td>5</td>
<td>74.3 %</td>
</tr>
<tr>
<td>10</td>
<td>71.8 %</td>
</tr>
</tbody>
</table>

Approaches 69.3% asymptotically
Example A

<table>
<thead>
<tr>
<th>Task</th>
<th>Period</th>
<th>WCET</th>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>Ti</td>
<td>Ci</td>
<td>Ui</td>
</tr>
<tr>
<td>1</td>
<td>50</td>
<td>12</td>
<td>24%</td>
</tr>
<tr>
<td>2</td>
<td>40</td>
<td>10</td>
<td>25%</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>10</td>
<td>33%</td>
</tr>
</tbody>
</table>

The combined utilization is 82%, which is above the bound for 3 threads (78%)
The task set fails the utilization test
Time-line for example A

Missed deadline
Example B

<table>
<thead>
<tr>
<th>Task</th>
<th>Period</th>
<th>WCET</th>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>Ti</td>
<td>Ci</td>
<td>Ui</td>
</tr>
<tr>
<td>1</td>
<td>80</td>
<td>32</td>
<td>40%</td>
</tr>
<tr>
<td>2</td>
<td>40</td>
<td>5</td>
<td>12.5%</td>
</tr>
<tr>
<td>3</td>
<td>16</td>
<td>4</td>
<td>25%</td>
</tr>
</tbody>
</table>

The combined utilization is 77.5%, which is below the bound for 3 threads (78%). Hence the task set will meet all its deadlines.
Time-line for example B

```
0  16  32  48  64  80  96

5
4
4
4
4
4
4
4
4
```

```
7 + 12 + 4 + 3 + 6
7 +
5
5
```
Example C

<table>
<thead>
<tr>
<th>Task</th>
<th>Period</th>
<th>WCET</th>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>Ti</td>
<td>Ci</td>
<td>Ui</td>
</tr>
<tr>
<td>1</td>
<td>80</td>
<td>40</td>
<td>50%</td>
</tr>
<tr>
<td>2</td>
<td>40</td>
<td>10</td>
<td>25%</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td>5</td>
<td>25%</td>
</tr>
</tbody>
</table>

The combined utilization is 100%, which is well above the bound for 3 tasks (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’re ok)
  - 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 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 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’re 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 (next lecture)

- **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 behavior 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 periods instead!)
  - Utilization-based test becomes much more elaborate for EDF when $D_i < T_i$ (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 our system of reactive objects, EDF fits nicely as an alternative to RM
Implementation (RM)

In StaticTinyTimber.c:

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

void async( Time offset, Time prio, OBJECT *to, METHOD meth, int arg ) {
    ...
    m->baseline = offset < 0 ? current->baseline : MAX( TIMERGET(), current->baseline + offset);
    m->priority = prio;
    ...
}```
Implementation (EDF)

In TinyTimber.c:

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

void async( Time BL, Time DL, OBJECT *to, METHOD meth, int arg ) {
    ...
    m->baseline = BL < 0 ? current->baseline : MAX( TIMERGET(), current->baseline + BL);
    m->deadline = DL < 0 ? current->deadline : m->baseline + DL;
    ...
}
```

// Renaming of "priority", for clarity