2020 T2 Week 05a
Real Time Systems Basics
@GernotHeiser
Incorporating material by Stefan Petters and Anna Lyons
Copyright Notice

These slides are distributed under the Creative Commons Attribution 3.0 License

• You are free:
  • to share—to copy, distribute and transmit the work
  • to remix—to adapt the work

• under the following conditions:
  • Attribution: You must attribute the work (but not in any way that suggests that the author endorses you or your use of the work) as follows:

    “Courtesy of Gernot Heiser, UNSW Sydney”

The complete license text can be found at http://creativecommons.org/licenses/by/3.0/legalcode
Real-Time Basics
Real-Time Systems
What’s a Real-Time System?

A real-time system is a system that is required to react to stimuli from the environment (including passage of physical time) within time intervals dictated by the environment.

[Randell et al., Predictably Dependable Computing Systems, 1995]

Real-time systems have timing constraints, where the correctness of the system is dependent not only on the results of computations, but on the time at which those results arrive.

[Stankovic, IEEE Computer, 1988]

Issues:

- Correctness: What are the temporal requirements?
- Criticality: What are the consequences of failure?
Strictness of Temporal Requirements

- Hard real-time systems
- Weakly-hard real-time systems
- Firm real-time systems
- Soft real-time systems
- Best-effort systems
Real-Time Tasks

Real-time tasks have deadlines
- Usually stated relative to release time
- Frequently *implicit*: next release time

```c
void main(void) {
    init(); // initialise system
    while (1) {
        wait(); // timer, device interrupt, signal
        doJob();
    }
}
```
Real Time ≠ Real Fast

<table>
<thead>
<tr>
<th>System</th>
<th>Deadline</th>
<th>Single Miss Conseq</th>
<th>Ultimate Conseq.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Car engine ignition</td>
<td>2.5 ms</td>
<td>Catastrophic</td>
<td>Engine damage</td>
</tr>
<tr>
<td>Industrial robot</td>
<td>5 ms</td>
<td>Recoverable?</td>
<td>Machinery damage</td>
</tr>
<tr>
<td>Air bag</td>
<td>20 ms</td>
<td>Catastrophic</td>
<td>Injury or death</td>
</tr>
<tr>
<td>Aircraft control</td>
<td>50 ms</td>
<td>Recoverable</td>
<td>Crash</td>
</tr>
<tr>
<td>Industrial process</td>
<td>100 ms</td>
<td>Recoverable</td>
<td>Lost production, plant/environment damage</td>
</tr>
<tr>
<td>Pacemaker</td>
<td>100 ms</td>
<td>Recoverable</td>
<td>Death</td>
</tr>
</tbody>
</table>
Example: Industrial Control
Hard Real-Time Systems

- Safety-critical: Failure $\Rightarrow$ death, serious injury
- Mission-critical: Failure $\Rightarrow$ massive financial damage

- Deadline miss is *catastrophic*
- Steep and real *cost* function

---

Deadline

Cost

Triggering Event

Time
Challenge: Execution-Time Variance

WCET/BCET may be orders of magnitude!

- Data-dependent execution paths
- Microarchitecture (caches)
Weakly-Hard Real-Time Systems

- Most feedback control systems (incl life-support!)
  - Control compensates for occasional miss
  - Becomes unstable if too many misses
- Typically integrated with fault tolerance for HW issues

Tolerate small fraction of deadline misses

In practice, certifiers treat critical avionics as hard RT

Cost vs. Time (Deadline) - Triggering Event vs. Event
Firm Real-Time Systems

Result obsolete if deadline missed (loss of revenue)

- Forecast systems
- Trading systems

![Graph showing time, gain, deadline, and triggering event](image)
Soft Real-Time Systems

Deadline miss undesirable but tolerable, affects QoS

- Media players
- Web services

<table>
<thead>
<tr>
<th>Cost</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Deadline</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Triggering Event</th>
<th>Deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td></td>
</tr>
</tbody>
</table>

Bounded Tardiness

- Media players
- Web services

In computer science, real-time computing and reactive computing describe hardware systems subject to a "real-time constraint."
Best-Effort Systems

No deadline

In practice, duration is rarely totally irrelevant
Real-Time Operating System (RTOS)

• Designed to support real-time operation
  • Fast context switches, fast interrupt handling
  • More importantly, *predictable* response time

• Main duty is scheduling tasks to meet their deadline

Traditional RTOS is very primitive
• single-mode execution
• no memory protection
• inherently cooperative
• *all code is trusted*

Requires analysis of worst-case execution time (WCET)

RT vs OS terminology:
• “task” = thread
• “job” = execution of thread resulting from event
Real-Time Scheduling

- Ensuring all deadlines are met is harder than bin-packing
- Reason: time is not fungible

A: needs 1 slot every 3
B: needs 3 slots every 9

Deadline missed!
Real-Time Scheduling

• Ensuring all deadlines are met is harder than bin-packing
• Time is not fungible

Terminology:
• A set of tasks is **feasible** if there is a known algorithm that will schedule them (i.e. all deadlines will be met).
• A scheduling algorithm is **optimal** if it can schedule all **feasible** task sets.
Cyclic Executives

- Very simple, completely static, scheduler is just table
- Deadline analysis done off-line
- Fully deterministic

Drawback: Latency of event handling is hyper-period

while (true) {
    wait_tick();
    job_1();
    wait_tick();
    job_2();
    wait_tick();
    job_1();
    wait_tick();
    job_3();
    wait_tick();
    job_4();
}
Are Cyclic Executives Optimal?

• Theoretically yes if can slice (interleave) tasks
• Practically there are limitations:
  • Might require very fine-grained slicing
  • May introduce significant overhead

```c
while (true) {
    wait_tick();
    job_1();
    wait_tick();
    job_2();
    wait_tick();
    job_1();
    wait_tick();
    job_3();
    wait_tick();
    job_4();
}
```
On-Line RT Scheduling

• Scheduler is part of the OS, performs scheduling decision on-demand
• Execution order not pre-determined
• Can be preemptive or non-preemptive
• Priorities can be
  • fixed: assigned at admission time
    • scheduler doesn’t change prios
    • system may support dynamic adjustment of prios
  • dynamic: prios potentially different at each scheduler run
Fixed-Priority Scheduling (FPS)

• Classic L4 scheduling is a typical example:
  • always picks highest-prio runnable thread
  • round-robin within prio level
  • will preempt if higher-prio thread is unblocked or time slice depleted

FPS is not optimal, i.e. cannot schedule some feasible sets

In general may or may not:
• preempt running threads
• require unique prios
Rate Monotonic Priority Assignment (RMPA)

- Higher rate ⇒ higher priority:
  - $T_i < T_j \Rightarrow P_i > P_j$

- Schedulability test: Can schedule task set with periods \( \{T_1 \ldots T_n\} \) if
  \[
  U \equiv \sum \frac{C_i}{T_i} \leq n(2^{1/n} - 1)
  \]

Assumes “implicit” deadlines: release time of next job

<table>
<thead>
<tr>
<th>n</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>10</th>
<th>∞</th>
</tr>
</thead>
<tbody>
<tr>
<td>U [%]</td>
<td>100</td>
<td>82.8</td>
<td>78.0</td>
<td>75.7</td>
<td>74.3</td>
<td>71.8</td>
<td>log(2) = 69.3</td>
</tr>
</tbody>
</table>

RMPA is optimal for FPS
Rate-Monotonic Scheduling Example

RMPA schedulability bound is sufficient but not necessary

<table>
<thead>
<tr>
<th>Task</th>
<th>T</th>
<th>P</th>
<th>C</th>
<th>U [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>t₁</td>
<td>80</td>
<td>1</td>
<td>20</td>
<td>25</td>
</tr>
<tr>
<td>t₂</td>
<td>40</td>
<td>2</td>
<td>10</td>
<td>25</td>
</tr>
<tr>
<td>t₃</td>
<td>20</td>
<td>3</td>
<td>10</td>
<td>50</td>
</tr>
</tbody>
</table>

WCET

<table>
<thead>
<tr>
<th>3</th>
<th>3</th>
<th>3</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

blocked | preempted | 1

100

© Gernot Heiser 2019 – CC Attribution License
Another RMPA Example

<table>
<thead>
<tr>
<th></th>
<th>P</th>
<th>C</th>
<th>T</th>
<th>D</th>
<th>U [%]</th>
<th>release</th>
</tr>
</thead>
<tbody>
<tr>
<td>t₃</td>
<td>3</td>
<td>5</td>
<td>20</td>
<td>20</td>
<td>25</td>
<td>5</td>
</tr>
<tr>
<td>t₂</td>
<td>2</td>
<td>8</td>
<td>30</td>
<td>20</td>
<td>27</td>
<td>12</td>
</tr>
<tr>
<td>t₁</td>
<td>1</td>
<td>15</td>
<td>50</td>
<td>50</td>
<td>30</td>
<td>0</td>
</tr>
</tbody>
</table>

Deadline
Preemption
Release
Dynamic Prio: Earliest Deadline First (EDF)

- Job with closest deadline executes
  - priority assigned at job level, not task (i.e. thread) level
  - deadline-sorted release queue
- Schedulability test: Can schedule task set with periods \( \{T_1 \ldots T_n\} \) if

\[
U \equiv \sum \frac{C_i}{T_i} \leq 1
\]

Preemptive EDF is optimal
FPS vs EDF

RMPA

EDF
### FPS vs EDF

<table>
<thead>
<tr>
<th>Task</th>
<th>P</th>
<th>C</th>
<th>T</th>
<th>D</th>
<th>U [%]</th>
<th>release</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_3$</td>
<td>3</td>
<td>5</td>
<td>20</td>
<td>20</td>
<td>25</td>
<td>5</td>
</tr>
<tr>
<td>$t_2$</td>
<td>2</td>
<td>8</td>
<td>30</td>
<td>20</td>
<td>27</td>
<td>12</td>
</tr>
<tr>
<td>$t_1$</td>
<td>1</td>
<td>15</td>
<td>40</td>
<td>40</td>
<td>37.5</td>
<td>0</td>
</tr>
</tbody>
</table>

**Misses deadline!**

89.5
FPS vs EDF

RMPA $t_3$
$t_2$
$t_1$

EDF $t_3$
$t_2$
$t_1$

Misses deadline!

EDF schedules
Resource Sharing
Challenge: Sharing

Vehicle control must see consistent state

Sharing introduces dependencies

Updates

Vehicle Control

Shared Data (waypoints etc)

Navigation

Ground Comms
Critical Sections: Locking vs Delegation

RT terminology: Resource Server
Implementing Delegation

`serv_local()` {
  
  ... wait(ep);
  while (1) {
    /* critical section */
    Reply&wait(ep);
  }
}

`client()` {
  while (1) {
    
    ... call(ep);
    
    ... signal(not_ry);
    
    ... wait(not_rq);
  }
}

`serv_remote()` {
  while (1) {
    
    ... while (1) {
      
      wait(not_rq);
      
      /* critical section */
      
      signal(not_ry);
      
    }
  }
}

Hoare-style monitor
Suitable intra-core

Semaphore synchronisation
Suitable inter-core
Problem: Priority Inversion

- High-priority job is blocked by low-prio for a long time
- Long wait chain: t₁→t₄→t₃→t₂
- Worst-case blocking time of t₁ bounded by total WCET: C₂+C₃+C₄
Solution 1: Priority Inheritance ("Helping")
Solution 1: Priority Inheritance ("Helping")

If $t_1$ blocks on a resource held by $t_2$, and $P_1 > P_2$, then
- $t_2$ is temporarily given priority $P_1$
- when $t_t$ releases the resource, its priority reverts to $P_2$
Solution 1: Priority Inheritance (“Helping”)

If \( t_1 \) blocks on a resource held by \( t_2 \), and \( P_1 > P_2 \), then

- \( t_2 \) is temporarily given priority \( P_1 \)
- when \( t_t \) releases the resource, its priority reverts to \( P_2 \)

Long blocking chains!
Solution 1: Priority Inheritance ("Helping")

If \( t_1 \) blocks on a resource held by \( t_2 \), and \( P_1 > P_2 \), then
- \( t_2 \) is temporarily given priority \( P_1 \)
- when \( t_i \) releases the resource, its priority reverts to \( P_2 \)

Deadlock!

Priority Inheritance:
- Easy to use
- Potential deadlocks
- Complex to implement
- Bad worst-case blocking times
Solution 2: Priority Ceiling Protocol (PCP)

- Aim: Block at most once, avoid deadlocks
- Idea: Associate *ceiling priority* with each resource
  - Ceiling = Highest prio of jobs that may access the resource
  - On access, bump prio of job to ceiling

![Diagram of Immediate prio ceiling protocol (IPCP)]
IPCP vs PIP

The diagram illustrates the comparison between IPCP and PIP using a timeline. For both IPCP and PIP, the timeline is divided into time intervals labeled t1, t2, t3, and t4. The IPCP diagram shows a sequence of tasks with their respective durations, while the PIP diagram follows a similar pattern. The tasks are represented by colored bars, and the diagram highlights the differences in execution times and scheduling efficiency between the two methods. The IPCP diagram suggests a more streamlined execution process compared to PIP, indicating potential improvements in real-time systems.
ICPC Implementation With Delegation

Each task must declare all resources at admission time
- System must maintain list of tasks using resource
- Defines ceiling priority

Immediate Priority Ceiling:
- Requires correct prio config
- Deadlock-free
- Easy to implement
- Good worst-case blocking times

$$P_S = \max (P_1, P_2) + 1$$

Easy to enforce with caps
Comparison of Locking Protocols

- Original Priority-Ceiling Protocol
- Priority-Inheritance Protocol
- Immediate Priority-Ceiling Protocol
- Non-Preemptible Critical Sections

Priority Inversion Bound vs Implementation Complexity
Scheduling Overloaded RT Systems
Naïve Assumption: Everything is Schedulable

Standard assumptions of classical RT systems:
• All WCETs known
• All jobs complete within WCET
• Everything is trusted

More realistic: Overloaded system:
• Total utilisation exceeds schedulability bound
• Cannot trust everything to obey declared WCET

Which job will miss its deadline?
Overload: FPS

<table>
<thead>
<tr>
<th>Task</th>
<th>P</th>
<th>C</th>
<th>T</th>
<th>D</th>
<th>U [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>t_3</td>
<td>3</td>
<td>5</td>
<td>20</td>
<td>20</td>
<td>25</td>
</tr>
<tr>
<td>t_2</td>
<td>2</td>
<td>12</td>
<td>20</td>
<td>20</td>
<td>60</td>
</tr>
<tr>
<td>t_1</td>
<td>1</td>
<td>15</td>
<td>50</td>
<td>50</td>
<td>30</td>
</tr>
</tbody>
</table>

Old

New

115
Overload: FPS
Overload: FPS vs EDF

\[ t_1, t_2, t_3 \]

FPS

EDF
Overload: EDF

"EDF behaves badly under overload"
Mixed-Criticality Systems
Mixed Criticality Systems
Mixed Criticality

NW driver must preempt control loop
- ... to avoid packet loss
- Driver must run at high prio (i.e. RMPA)
-Driver must not monopolise CPU

Runs every 100 ms for a few milliseconds

Control loop

Sensor readings

NW driver

Runs frequently but for short time (order of µs)

NW interrupts
Mixed Criticality

NW driver must preempt control loop
• … to avoid packet loss
• Driver must run at high prio (i.e. RMPA)
• Driver must not monopolise CPU

Certification requirement:
More critical components must not depend on any less critical ones! [ARINC-653]

Critical system certification:
• expensive
• conservative assumptions
  • eg highly pessimistic WCET

• Must minimise critical software
• Need temporal isolation: Budget enforcement
Mixed-Criticality Support

For supporting *mixed-criticality systems* (MCS), OS must provide:

- *Temporal isolation*, to force jobs to adhere to declared WCET

- Mechanisms for *safely sharing resources* across criticalities

**Will discuss seL4 approach next lecture!**