2023 T3 Week 07 Part 1

Real-Time Systems Basics

@GernotHeiser
Copyright Notice

These slides are distributed under the Creative Commons Attribution 4.0 International (CC BY 4.0) 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/4.0/legalcode
Today’s Lecture

• Real-time systems (RTS) basics
  • Types or RTS
  • Basic concepts & facts
• Resource sharing in RTS
• Scheduling overloaded RTS
• Mixed-criticality systems (MCS)
Presented by Dr Anna Lyons

Work
• 2022-23, secure kernel team @ Apple
• 2019-22, platform team @ Ghost
• 2010-18 Research Engineer @ Trustworthy Systems
• 2007-2018 Tutor - OS, AOS, COMP19**
• 2010 summer intern @ Microsoft - Bing
• 2008-10 Part-time @ Atlassian
• 2007 summer ToR @ NICTA 2007-08

Education
• 2012-2018 PhD w/ Gernot
• 2006-11 B Sci (Computer Science) / BA (Philosophy)
Presented by Dr Anna Lyons

Work at Trustworthy Systems

• Initial port of AOS to seL4 w/ Adrian Danis, then aarch64 + pico tcp + nfsv3
• Shepherd AOS from nslu2 to imx6 then odroid c2
• PhD: MCS kernel extensions
• I did AOS on the slug —> w/ OKL4
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?
Real Time → time isn’t **fungible**

Fungible: *replaceable* by another identical item

<table>
<thead>
<tr>
<th>Fungible</th>
<th>Not fungible</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chocolate chip cookies</td>
<td>Human Beings</td>
</tr>
<tr>
<td>Memory (e.g RAM)</td>
<td>The seconds after you hit the brake</td>
</tr>
</tbody>
</table>
Real-time = Real confusion

❌ Real-time Applications

Real-time apps are those that react to changes anywhere in a connected system.

❌ Real-time Processing

They actually mean “not batch processed”
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

```
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>Combustion 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>

**Criticality**
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
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
Firm Real-Time Systems

Result obsolete if deadline missed (loss of revenue)

- Forecast systems
- Trading systems

Diagram:
- Gain
- Deadline
- Triggering Event
- Time
Soft Real-Time Systems

Deadline miss undesirable but tolerable, affects QoS

- Media players
- Web services

### Diagram

- **Cost** vs. **Time**
  - **Deadline** and **Triggering Event**
  - **Tardiness** and **Deadlines**

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

No deadline

In practice, duration is rarely totally irrelevant

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 a 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_3();
    wait_tick();
    job_4();
    wait_tick();
    job_1();
    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
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
  \[
  U = \sum \frac{C_i}{T_i} \leq n(2^{1/n} - 1)
  \]

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>( \infty )</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>t₃</td>
<td>20</td>
<td>3</td>
<td>10</td>
<td>50</td>
</tr>
<tr>
<td>t₂</td>
<td>40</td>
<td>2</td>
<td>10</td>
<td>25</td>
</tr>
<tr>
<td>t₁</td>
<td>80</td>
<td>1</td>
<td>20</td>
<td>25</td>
</tr>
</tbody>
</table>

WCET

C/T

Blocked

Preempted
Another RMPA Example

<table>
<thead>
<tr>
<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>
</tr>
<tr>
<td>t_2</td>
<td>2</td>
<td>8</td>
<td>30</td>
<td>20</td>
<td>27</td>
</tr>
<tr>
<td>t_1</td>
<td>1</td>
<td>15</td>
<td>50</td>
<td>50</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>82</td>
</tr>
</tbody>
</table>

Preemption

Deadline

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

32
FPS vs EDF

<table>
<thead>
<tr>
<th>Task</th>
<th>P</th>
<th>C</th>
<th>T</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>25</td>
<td>5</td>
</tr>
<tr>
<td>t_2</td>
<td>2</td>
<td>8</td>
<td>30</td>
<td>27</td>
<td>12</td>
</tr>
<tr>
<td>t_1</td>
<td>1</td>
<td>15</td>
<td>40</td>
<td>37.5</td>
<td>0</td>
</tr>
</tbody>
</table>

Total Misses: 89.5

Misses deadline!
FPS vs EDF

RMPA

\[ t_1 \quad t_2 \quad t_3 \]

EDF

\[ t_1 \quad t_2 \quad t_3 \]

Misses deadline!

EDF schedules
Resource Sharing
Challenge: Sharing

Sharing introduces dependencies

Vehicle control must see consistent state

Updates

Vehicle Control → Shared Data (waypoints etc) → Navigation → Ground Comms
Critical Sections: Locking vs Delegation

RT terminology: Resource

Server
Buffer

Client
Lock()
Unlock()
Send()
Receive()
or Poll()

Shared
Buffer

Client
Lock()
Unlock()

Client
Send()
Receive()
or Poll()
Implementing Delegation

Hoare-style monitor
Suitable intra-core

Semaphore synchronisation
Suitable inter-core

serv_local() {
    ...
    Wait(ep);
    while (1) {
        /* critical section */
        ReplyWait(ep);
    }
}

client() {
    while (1) {
        ...
        Call(ep);
        ...
        Signal(not_ry);
        ...
        Wait(not_rq);
    }

    /* critical section */
    
    serv_remote() {
        ...
        while (1) {
            ...
            Wait(not_rq);
            /* critical section */
            Signal(not_ry);
            ...
            Wait(not_rq);
        }
    }
Problem: Priority Inversion

- High-priority job is blocked by low-prio for a long time
- Long wait chain: $t_4 \rightarrow t_1 \rightarrow t_3 \rightarrow t_2$
- Worst-case blocking time of $t_4$ bounded by total WCET: $C_1+C_2+C_3$
Solution 1: Priority Inheritance ("Helping")

Diagram showing the priority inheritance process over time.

- At time $t_1$, process 1 is executing at priority 1.
- At time $t_2$, process 2 is executing at priority 2.
- At time $t_3$, process 3 is executing at priority 3.
- At time $t_4$, process 4 is executing at priority 4.

The priority inheritance process involves lifting the priority of a lower-priority process when a higher-priority process needs to execute.

Legend:
- $Q$: Quiescent state
- $V$: Running state
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 \( P_2 \)

Priority Inheritance:
- Easy to use
- Potential deadlocks
- Complex to implement
- Bad worst-case blocking times

Deadlock!
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
IPCP vs PIP
ICPC Implementation With Delegation

Given the server priority $P_s$:

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

The immediate priority ceiling is:

- Requires correct prioritization configuration
- Deadlock-free
- Easy to implement
- Good worst-case blocking times

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

EDF: Floor of deadlines

Easy to enforce with caps
Comparison of Locking Protocols

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

Implementation Complexity vs. Priority Inversion Bound
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>

$ \text{Total} = 115$
Overload: FPS

Old

New
Overload: FPS vs EDF

 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

Runs frequently but for short time (order of μs)
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
Remember: Delegation of Critical Sections

Client may frequently invoke server without using much of its own time!

Client may run on clients time slice, its own or a combination

No accurate accounting for time
MCS Model: Scheduling Contexts

**Classical thread attributes**
- Priority
- Time slice

**MCS thread attributes**
- Priority
- Scheduling context capability

**Scheduling context object**
- \( T \): period
- \( C \): budget (\( \leq T \))

- Not runnable if null
- Limits CPU access!
- Per-core SchedControl capability conveys right to assign budgets (i.e. perform admission control)

\[ \begin{align*}
C &= 2 \\
T &= 3 \\
C &= 250 \\
T &= 1000
\end{align*} \]
Delegation with Scheduling Contexts

Client is charged for server’s time

Client 1

Client 2

Passive servers support migrating thread model!

Server runs on client’s scheduling context

Scheduling-context capabilities: a principled, light-weight OS mechanism for managing time [Lyons et al, EuroSys’18]
Mixed-Criticality Support

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

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

- Mechanisms for *safely sharing resources* across criticalities

What if budget expires while shared server executing on Low’s scheduling context?
Timeout Exceptions

Policy-free mechanism for dealing with budget depletion

Possible actions:

• Provide emergency budget to leave critical section
• Cancel operation & roll-back server
• Reduce priority of low-crit client (with one of the above)
• Implement priority inheritance (if you must…)

Arguable not ideal: better prevent timeout completely
RFC-14: Adding budget limit thresholds to endpoints for SC Donation
Isn’t a Fixed-Prio Scheduler Policy?

Implementing scheduling policy at user level

Client runs for period, then time-faults (or explicitly yields by calling EP)

Scheduler waits for client timeout

User-level Scheduler

Scheduler runs client by replying

Client 1: $C_1 = P_1 = D_1$

Client 2: $C_2 = P_2 = D_2$
User-Level EDF Scheduler Performance

Linux in-kernel
WCET Analysis

Program binary → Control Flow Graph

Micro-architecture model → Loop bounds

Analysis tool → Integer linear equations → ILP solver → WCET

Accurate & sound model of pipeline, caches

Infeasible path info → Pessimism!

Scalability!
WCET Analysis on ARM11

WCET presently limited by verification practicalities
- without regard to verification achieved 50 µs
- 10 µs seem achievable
- BCET ~ 1µs
- [Blackham‘11, ‘12] [Sewell’16]
Presented by Dr Anna Lyons

Internship!
search “secure kernel engineering intern apple”

Contact
linked in: https://www.linkedin.com/in/annamlyons/
email: anna.lyons@apple.com
Fun links

For the dark nights of AOS debugging: “The Night Watch”
https://www.usenix.org/system/files/1311_05-08_mickens.pdf
Real world priority inversion: NASA
https://www.rapitasystems.com/blog/what-really-happened-software-mars-pathfinder-spacecraft
Real world mess: (When real time is wrong) Toyota breaking