Real-Time Basics

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, 1968]

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>Simple 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 ⇒ death, serious injury
- Mission-critical: Failure ⇒ massive financial damage

Challenge: Execution-Time Variance

WCET/BCE 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
Firm Real-Time Systems

- Result obsolete if deadline missed (loss of revenue)

---

Soft Real-Time Systems

- Deadline miss undesirable but tolerable, affects QoS

---

Best-Effort Systems

- No deadline

---

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

---

Real-Time Scheduling

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

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
  - 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

Rate Monotonic Priority Assignment (RMPA)

- Higher rate ⇒ higher period:
  \[ T_i < T_j \Rightarrow P_i > P_j \]
- Schedulability test: Can schedule task set with periods \( T_1 \ldots T_n \) if
  \[ \frac{U}{n} \leq \sum C/T_i \leq n(2^{1/n} - 1) \]

Assumes “implicit” deadlines: release time of next job

RMPA is optimal for FPS

Rate-Monotonic Scheduling Example

Task | T | P | C | U [%] | WCET
---|---|---|---|---|---
t_1 | 20 | 3 | 10 | 50 |
t_2 | 40 | 2 | 10 | 25 |
t_3 | 80 | 1 | 20 | 25 |

RMPA schedulability bound is sufficient but not necessary
Another RMPA Example

<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_1</td>
<td>3</td>
<td>5</td>
<td>20</td>
<td>25</td>
<td>20</td>
</tr>
<tr>
<td>t_2</td>
<td>2</td>
<td>8</td>
<td>30</td>
<td>27</td>
<td>27</td>
</tr>
<tr>
<td>t_3</td>
<td>1</td>
<td>5</td>
<td>50</td>
<td>30</td>
<td>30</td>
</tr>
</tbody>
</table>

Deadline

Preemption

Release

Deadline

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

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

Preemptive EDF is optimal

Representation of Schedulability Test

FPS vs EDF

.resource-sharing

Resource Sharing
Challenge: Sharing

Vehicle control must see consistent state

Updates

Navigation

Shared Data (waypoints etc)

Vehicle Control

Ground Comms

Critical Sections: Locking vs Delegation

Client 1

Server

Client 2

Server

Lock()

Unlock()

Send()

Lock()

Unlock()

Send()

RT terminology: Resource Server

Implementing Delegation

Hoare-style monitor
Suitable intra-core

Semaphore synchronisation
Suitable inter-core

Problem: Priority Inversion

• High-priority job is blocked by low-priority job for a long time
• Long wait chain: \(t_1\rightarrow t_4\rightarrow t_3\rightarrow t_2\)
• Worst-case blocking time of \(t_1\) bounded by total WCET: \(C_2+C_3+C_4\)

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 \(t_1\) 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 \( t_1 \) releases the resource, its priority reverts to \( P_2 \)

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

### Transitive Inheritance

- Long blocking chains!

### 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

### Immediate Priority Ceiling Protocol (IPCP)

- Requires correct prio config
- Deadlock-free
- Easy to implement
- Good worst-case blocking times

### Comparison of Locking Protocols

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Implementation Complexity</th>
<th>Priority Inversion Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original Priority-Ceiling Protocol</td>
<td>Non-Preemptible Critical Sections</td>
<td></td>
</tr>
<tr>
<td>Immediate Priority-Ceiling Protocol</td>
<td>Easy to enforce with caps</td>
<td></td>
</tr>
<tr>
<td>Priority-Inheritance Protocol</td>
<td>Require correct prio config</td>
<td></td>
</tr>
<tr>
<td>Non-Preemptible Critical Sections</td>
<td>Deadlock-free</td>
<td></td>
</tr>
</tbody>
</table>

### Implementation With Delegation

\[ P_S = \text{max}(P_1, P_2) + 1 \]

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

**EDF:** Floor of deadlines
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

Overload: FPS

Overload: FPS vs EDF

Overload: EDF

Which job will miss its deadline?
Mixed Criticality Systems

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

Runs every 100 ms for a few milliseconds
Sensor readings → Control loop → NW driver

Runs frequently but for short time (order of µs)
NW driver → NW interrupts

Need temporal isolation!