What is real-time?

Real-Time Basics

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 Anna Lyons, UNSW Sydney”

The complete license text can be found at http://creativecommons.org/licenses/by/3.0/legalcode
What is real-time?

“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.”


All systems are real-time

- Must model a system to define correctness in terms of result arrival times.
- Most systems are best effort, deliver the results when you can
  - CFS in Linux
  - Socket.io

Parameters of Real-Time Systems

- Criticality – the consequences of failure
- Correctness – what temporal requirements define failure/correctness?

Correctness in real-time systems

- Hard real-time systems
- Weakly-hard real-time systems
- Firm real-time systems
- Soft real-time systems
- Best-effort systems

Typical RT task

```c
int main(void) {
  init(); // initialise system
  while (1) {
    sleep(); // go to sleep
    // wake due to some external event
    // timeout or hardware interrupt or signal
    doJob();
  }
}
```
Types of Real-Time Systems

- Real-time systems typically deal with deadlines:
  - A deadline is a time instant by which a response has to be completed
  - A deadline is usually specified as relative to an event
    - The relative deadline is the maximum allowable response time
    - Absolute deadline: event time + relative deadline

Typical RT task

```c
int main(void) {
    init(); // initialise system
    while (1) {
        sleep(); // go to sleep
        // wake due to some external event
        // timeout or hardware interrupt or signal
        doJob();
        // must complete by some deadline
    }
}
```

Sporadic task model

- WCET – upper bound on execution
- Arrival
- Period (T) – minimal inter-arrival time
- Deadline (implicit)

Hard Real-Time Systems

- Deadline miss is "catastrophic":
  - safety-critical system: failure results in death, severe injury
  - mission-critical system: failure results in massive financial damage
- Steep and real "cost" function

Real-Time Systems

- Example RT requirements in industrial automation:
- High-speed PLC
- Simple PLC
- High-speed drives
- Standard motor control
- High and motion control
- Interrupt reaction time
- 1ms
- 10ms
- 100ms
- 1000ms
- Source: Siemens
Real-Time ≠ Real Fast

<table>
<thead>
<tr>
<th>System</th>
<th>Deadline</th>
<th>Single Miss Consequences</th>
<th>Ultimate Consequence</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>Catastrophic</td>
<td>Machinery damage</td>
</tr>
<tr>
<td>Airbag</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>

Challenge of real-time systems: Guaranteeing deadlines

Challenge: Execution-Time Variance

Variance may be orders of magnitude!
- Data-dependent execution path
- Micro-architectural features: pipelines, caches

Weakly-Hard Real-Time Systems
- Tolerate a (small) fraction of deadline misses
  - Most feedback control systems (including life-supporting ones!)
    - Occasionally missed deadline can be compensated at next event
  - System becomes unstable if too many deadlines are missed
  - Typically integrated with other fault tolerance
    - Electro-magnetic interference, other hardware issues

Firm Real-Time Systems
- Deadline miss makes computation obsolete
  - Typical examples are forecast systems
    - Weather forecast
    - Trading systems
  - Cost may be loss of revenue (gain)

Soft Real-Time Systems
- Deadline miss is undesired but tolerable
  - Frequently results in quality-of-service (QoS) degradation
    - Eg audio, video rendering
    - Sleep "cost" function
  - Cost of deadline miss may be abstract

Best-Effort Systems
- No deadlines, timeliness is not part of required operation
- In reality, there is at least a nuisance factor to excessive duration
  - Response time to user input
- Again, "cost" may be reduced gain
Real-Time Operating System (RTOS)

- Designed to support real-time operation
  - Fast context switches, fast interrupt handling?
  - Yes, but predictable response time is more important
  - “Real time is not real fast”
- Support for scheduling policies appropriate for real time
- Classical RTOSes very primitive
  - Single-mode execution
  - No memory protection
  - Essentially a scheduler with a threads package
  - “Real-time executive”
  - Inherently cooperative
  - Inherently trust all code
- Many modern uses require actual OS technology for isolation
  - Generally microkernels
  - QNX, Integrity, VxWorks, L4 kernels

Real-time scheduling

- How to meet all deadlines?
- Harder than bin-packing as time is not fungible
- Terminology
  - A set of real-time tasks, with parameters, is feasible if it can be scheduled by any known scheduling algorithm.
  - A scheduler is optimal if can schedule all feasible task sets.
- Challenges:
  - Meet all deadlines
  - Get high CPU utilisation

Cyclic executives

- Easiest option, completely static
- Deadline analysis offline
- Scheduler is just a table
- Advantages
  - Fully deterministic
  - “Cyclic executive” is trivial
  - Minimal overhead
- Disadvantages
  - Big latencies if event rate doesn’t match base rate (hyper-period)

- Infeasible
Are Cyclic executives optimal?

- Can they schedule every feasible task set?
  - If tasks cannot be sliced, no
  - Even then, only theoretically optimal if tasks can be sliced into infinitesimal small slices to achieve optimality
  - Impractical as overhead of switching tasks becomes too high, limiting CPU utilisation
  - So practically NO

Online RT scheduling

- Scheduler is online:
  - Order not predetermined
- Two types: task-level vs. job-level priorities
- Preemptive or non-preemptive

Fixed-Priority Scheduling (FPS)

- Task-level (thread level)
- Assign priorities, always pick highest-priority task
- Obviously easy to implement, low overhead
- Drawbacks: inflexible, sub-optimal
  - Cannot schedule some systems which are schedulable preemptively
  - Note: “Fixed” in the sense that system doesn’t change them
  - OS may support dynamic adjustment
  - Requires on-the-fly (re-)admission control

Choosing Prios: Rate-Monotonic

- Rate-Monotonic Scheduling (RMS): Standard approach to fixed priority assignment
  - \( T_i < T_j \Rightarrow P_i > P_j \)
  - RMS is optimal for task-level priorities
- RMS schedulability condition is sufficient but not necessary
- Schedulability test: RMS can schedule \( n \) tasks with \( D=T \) if
  \[ U = \sum_{i=1}^{n} C_i / T_i \]
  \[ \lim_{n \to \infty} U = \log 2 \]
- If \( D < T \) replace by deadline-monotonic scheduling (DMS):
  - DMS is also optimal (but schedulability bound is more complex)

Rate-Monotonic Scheduling

RMS schedulability condition is sufficient but not necessary

FPS Example

Preemption

Deadline

Release
**Non-Preemptive Scheduling**
- Minimises context-switching overhead
  - Significant cost on modern processors (pipelines, caches)
- Easy to analyse timeliness
- Drawbacks:
  - Larger response times for “important” tasks
  - Reduced utilisation, schedulability
  - In many cases cannot produce schedule despite plenty idle time
  - Can’t re-use slack (eg for best-effort)
- Only used in very simple systems

**Choosing Priors: Earliest Deadline First (EDF)**
- Dynamic scheduling policy with job-level priorities
- Job with closest deadline executes
- Preemptive EDF with implicit deadlines is optimal. $n$ jobs can be scheduled if
  \[ U \equiv \sum C_i / T_i \leq 1 \]
- Necessary and sufficient condition
- No easy test for explicit deadlines

**FPS vs EDF**

<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>1</td>
<td>3</td>
<td>20</td>
<td>20</td>
<td>25</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>30</td>
<td>20</td>
<td>27</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>40</td>
<td>40</td>
<td>37.5</td>
<td>0</td>
</tr>
</tbody>
</table>

**Resource Sharing**
Example: SMACCMcopter Drone

Flight Control Computer
- HW Sensors
- ARM M3
- Radio
- Motors

Mission Computer
- HW Control
- SW Monitor
- Mission Plan
- Sensor Filtering
- eChronos RTOS
- CAN

SMACCMcopter Mission Computer Architecture
- UART Rx
- UART Tx
- CAN Rx
- CAN Tx
- Server
- CAN

Critical Sections as Resource Servers
- Hoare-style monitor
- Semaphore synchronisation

Problem: Priority Inversion
- High-priority job is blocked for a long time by a low-prio job
- Long wait chain: t₁→t₄→t₃→t₂
- Worst-case blocking time of t₁ bounded by total WCET: C₂+C₃+C₄
- Must find a way to do better!

Priority Inheritance (“Helping”)
- If t₁ blocks on a resource held by t₂, and P₂>P₁, then
  - t₂ is temporarily given priority P₁
  - when t₁ releases the resource, its priority reverts to P₂

Priority Inheritance
- If t₁ blocks on a resource held by t₂, and P₂>P₁, then
  - t₁ is temporarily given priority P₁
  - when t₁ releases the resource, its priority reverts to P₂
Priority Inheritance

- 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 \( t_2 \) releases the resource, its priority reverts to \( P_2 \)

Transitive Inheritance
- Potential long blocking chains
- Potential for deadlock
- Frequently blocks much longer than necessary

Priority Inheritance Protocol (PIP)

- 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 \( t_2 \) releases the resource, its priority reverts to \( P_2 \)
- Transitive inheritance
- Frequently blocks much longer than necessary

Priority Ceiling Protocol (PCP)

- Purpose: ensure job can block at most once on a resource
  - avoid transitivity, potential for deadlocks
- Idea: associate a ceiling priority with each resource
  - equal to the highest priority of jobs that may use the resource
  - when job accesses its resource, immediately bump priority to ceiling
- Also called:
  - immediate ceiling priority protocol (ICPP)
  - ceiling priority protocol (CPP)
  - stack-based priority-ceiling protocol (SBCP)
    - because it allows running all jobs on the same stack (i.e. thread)
- Improved version of the original ceiling priority protocol (OCPP)
  - OPCP bumps priority to ceiling of all waiting threads
  - IPCP requires global tracking of ceiling priority

Priority Ceiling:
- Requires correct priority configuration
- Deadlock-free
- Easy to implement
- Good worst-case blocking times

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

IPCP Implementation

- Each task must declare all resources at admission time
  - System must maintain list of tasks associated with resource
  - Priority ceiling derived from this list
- For EDF the “ceiling” is the floor of relative deadlines

seL4: “resource declaration” is implicit in capability distribution
- Using critical section requires cap for server’s request endpoint
Comparison of Locking Protocols

Temporal isolation

Naïve Assumptions: Everything Schedulable

Standard assumptions of classical RT systems:
• All WCETs known
• All jobs complete within WCET
• Everything is Trusted
What happens if those assumptions are not met?
• Overloaded system:
  – Not all tasks complete within WCET or
  – Total utilisation exceeds schedulability bounds
• System no longer schedulable – what loses?

Overload: FPS

Overload: FPS vs EDF
Overload: EDF

\[ t_3 > t_2 > t_1 > t_3 \]

Overload: FPS vs EDF

- On overload, (by definition!) lowest-prio jobs miss deadlines
- Result is well-defined and understood for FPS
  - Treats highest-prio task as "most important"
  - ...but that may not always be appropriate!
  - Under transient overload may miss deadlines of higher-priority tasks
- Result is unpredictable (seemingly random) for EDF
  - May result in all tasks missing deadlines!
  - Under constant overload will scale back all tasks
  - No concept of task "importance"
  - "EDF behaves badly under overload"
  - Main reason EDF is unpopular in industry

Why Have Overload?

- Faults (software, EMI, hardware)
- Incorrect assumptions about environment
- Optimistic WCET
  - Computing WCET of non-trivial programs is hard, often infeasible!
  - Safe WCET bounds tend to be highly pessimistic (orders of magnitude!)
  - WCET often very unlikely and orders of magnitude worse than "normal"
  - Estimation inaccuracies from caches, pipelines, under-specified hardware...
  - "normal" vs "exceptional" operating conditions
  - Some systems have effectively unbounded execution time
  - e.g. object tracking

Why Overload? SWaP Challenge

Traditional embedded-systems approach: one µ-controller per function
- Automotive reached 100 ECUs in top-of-line cars 10 years ago
- ECUs must be robust – expensive
  - Tolerant to wide temperature range
  - Resistant to dust, water, grease, acid
  - Resistant to Vibrations
- SWaP: space, weight and power
  - Overhead of packaging, cabling
- Autonomous vehicles require even more functions than traditional
  - Also integration/cooperation of functionality
  - Infotainment/driver assist: consumer electronics + automotive control
- General challenge for cyber-physical systems (CPS)
  - Robots, autonomous aircraft, smart factories

The seL4 MCS kernel

(my phd 😊)
Criticality

- Higher criticality =
  - Increased severity of failure
  - Increased cost to develop
  - Higher certification requirements

Mixed-Criticality Systems

- Air bags
- Object avoidance
- Navigation
- GPS
- Traffic information
- Music

Hardware separation

- Wastes resources
  - Redundancy & overprovisioning
- Scalability
  - Size, weight and power
- Sensor fusion
  - Prevents shared resources

Mixed-criticality systems

- Running software of different criticality on shared hardware

Mixed-criticality systems

- Running software of different criticality on shared hardware
- Without strong spatial and temporal isolation, everything is promoted to the highest criticality level
MCS Challenge:
• We know how to do spatial isolation.
• How to achieve temporal isolation?

Temporal isolation = ?

Memory isolation
• We know how to do spatial isolation.
• Capability = Access Token:
  – Prima-facie evidence of privilege
• How to achieve temporal isolation?

Temporal isolation = ?
• Capabilities to time?
• Can’t treat like memory
• Time is not fungible

ARINC 653
• Avionics mixed-criticality standard
• Cyclic executive provides temporal isolation

Model
Operating system
• Enforces scheduling parameters
### Scheduling control
- New scheduling contexts do not represent any time at all
- Scheduling control

### Resource sharing

### Priority
- **Stays attribute of thread**
  - Not separate scheduling context
- **Offers native implementation of IPCP**
  - Set server priority to highest prio client + 1
- **Or Non-blocking critical sections**
  - Server prio same as clients
- Can implement other protocols at user level

*PIP would prevent policy by user, increase preemption by default.*

### Charging

### Enforcement
Deferrable server

- Recharge every $T$ to a max of $C$

\[
\text{Max Utilisation} = \frac{C}{T}
\]

Sporadic server

- Track when a task starts executing: $t$
- When it stops, schedule recharge of that mount at $t + T$

\[
\text{Complex implementation} \Rightarrow \text{must bound replenishments}
\]
- Too many replenishments $\Rightarrow$ high overhead
  - IRQ overhead $\Rightarrow$ IRQ per replenishment
  - Tracking replenishments (memory)
  - Merging and processing replenishments (CPU time)
- Too few $\Rightarrow$ tasks lose time
  - Can’t track more than bound
  - Replenishments over the bound merged

Multicore
API – Sched(Context|Control)

- seL4_SchedControl_Configure(control_cap, sc_cap, budget, period, extra_refills)
- seL4_SchedContext_Bind(sc_cap, tcb_cap or ntfn_cap)
- seL4_SchedContext_Unbind(sc_cap)
- seL4_SchedContext_Consumed(sc_cap)
- seL4_SchedContext_YieldTo

New object

Reply objects are first class

- seL4_CNode_SaveCaller
- seL4_Wait(ep, &badge)
- seL4_ReplyRecv(ep, info, &badge)
- seL4_NBSendRecv(ep, info, &badge, reply)