Real-Time Basics

Real-Time System: Definition

A real-time system is any information processing system which has to respond to externally generated input stimuli within a finite and specified period.

- Correctness depends not only on the logical result (function) but also the time it was delivered.
- Failure to respond is as bad as delivering the wrong result!

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 Systems

Types of Real-Time Systems

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

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

Eg RT Requirements in Industrial Automation

Source: Siemens
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>

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 on quality-of-service (QoS) degradation
    - e.g., audio, video rendering
    - Steep “cost” function
  - Cost of deadline miss may be abstract

\[\text{Cost} \rightarrow \text{Deadline} \quad \text{Time} \rightarrow \text{Triggering Event}\]

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

\[\text{Cost} \rightarrow \text{Triggering Event} \quad \text{Time}\]

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”
      - Analysis of worst-case execution time (WCET)
  - 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

Approaches to Real Time

- Clock-driven (cyclic)
  - Periodic scheduling
  - Typical for control loops
  - Fixed order of actions, round-robin execution
  - Statically determined (static schedule) if periods are fixed
    - Need to know all execution parameters at system configuration time
- Event-driven
  - Sporadic scheduling
  - Typical for reactive systems (sensors & actuators)
  - Static or dynamic schedules
  - Analysis requires bounds on event arrivals

\[\text{Emulation on event-driven system: treat clock tick as event}\]

\[\text{Emulation on clock-driven system: buffer event (IRQ) until timer tick}\]
Real-Time System Operation

- **Time-triggered**
  - Pre-defined temporal relation of events
  - Event is not serviced until its defined *release time* has arrived

- **Event-triggered**
  - Timer interrupt
  - Asynchronous events

- **Rate-based**
  - Activities get assigned CPU shares ("rates")

Real-Time Task Model

- **Job**: unit of work to be executed
  - ... resulting from an event or time trigger

- **Task**: set of related jobs which provide some system function
  - A *task* is a sequence of *jobs* (typically executing same function)
  - Job $i+1$ of a task cannot start until job $i$ is completed/aborted

- **Periodic tasks**
  - Time-driven and all relevant characteristics known a priori
    - Task $t_i$ characterized by period $T_i$, deadline $D_i$, and execution time $C_i$
    - Applies to all jobs of task

- **Aperiodic tasks**
  - Event driven, characteristics are not known a priori
    - Task $t_i$ characterized by period $T_i$, deadline $D_i$, and arrival distribution

- **Sporadic tasks**
  - Aperiodic but with known minimum inter-arrival time $T_i$
  - Treated similarly to periodic task with period $T_i$

Standard Task Model

- **C**: Worst-case computation time (WCET)
- **T**: Period (periodic) or minimum inter-arrival time (sporadic)
- **D**: Deadline (relative, frequently "implicit deadlines" D=T)
- **J**: Release jitter
- **P**: Priority: higher number means higher priority
- **B**: Worst-case blocking time
- **R**: Worst-case response time
- **U**: Utilisation; $U = \frac{C}{T}$

OS terminology:

- "task" = thread
- "job" = unblocking of thread by event

Task Constraints

- **Deadline constraint**: must complete before deadline
- **Resource constraints**:  
  - Shared (R/O), exclusive (W-X) access
  - Energy
  - Precedence constraints:  
    - $t_i \Rightarrow t_j$: $t_2$ execution cannot start until $t_1$ is finished
  - Fault-tolerance requirements  
    - Eg redundancy

- **Scheduler's job**: to ensure that constraints are met!
Scheduling

- Preemptive vs non-preemptive
- Static (fixed, off-line) vs dynamic (on-line)
- Clock-driven vs priority-based
  - clock-driven is static, only works for very simple systems
  - priorities can be static (pre-computed and fixed) or dynamic
  - dynamic priority adjustment can be at task-level (each job has fixed prio) or job-level (jobs change prios)

Clock-Driven (Time-Triggered) Scheduling

- Typically implemented as time “frames” adding up to “base rate”
- Advantages
  - fully deterministic
  - “cyclic executive” is trivial
  - minimal overhead
- Disadvantage:
  - Big latencies if event rate doesn’t match base rate (hyper-period)
  - Inflexible

```
while (true) {
  wait_tick();
  job_1();
  wait_tick();
  job_2();
  wait_tick();
  job_1();
  wait_tick();
  job_3();
  wait_tick();
  job_4();
}
```

Hyper-period (inverse base rate)

Synchronous Distributed RT Systems

- Can treat like single system if clocks synchronised?
  - Issue clock drift: can only synchronise within certain accuracy
- Time-triggered architecture
  - Idea: use sparse time:
    - Restrict events to active interval \( \pi \)
    - Separated by silence interval \( \Delta \)
    - \( \Delta \) allows for clock drift and communications time

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
Fixed-Priority Scheduling (FPS)

- Real-time priorities are absolute:
  - Scheduler always picks highest-priority job
- Obviously easy to implement, low overhead
- Drawbacks: inflexible, sub-optimal
  - Cannot schedule some systems which are schedulable preemptively
- Note: “Fixed” prios 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 Scheduling (RMS)

- RMS: Standard approach to fixed priority assignment
  - \( T_i < T_j \Rightarrow P_i > P_j \)
  - \( 1/T \) is the “rate” of a task
- RMS is optimal for fixed priorities
- Schedulability test: RMS can schedule \( n \) tasks with \( D = T \) if
  \[
  U = \sum C_i/T_i \leq n(2^{1/n} - 1); \quad \lim_{n \to \infty} U = \log 2
  \]
- If \( D < T \) replace by deadline-monotonic scheduling (DMS):
  - \( D_i < D_j \Rightarrow P_i > P_j \)
- DMS is also optimal (but schedulability bound is more complex)

Rate-Monotonic Scheduling

RMS schedulability condition is sufficient but not necessary

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

FPS Example

Preemption | Release | Deadline

<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>t₁</td>
<td>15</td>
<td>50</td>
<td>50</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>t₁</td>
<td>15</td>
<td>50</td>
<td>50</td>
<td>20</td>
<td>12</td>
</tr>
<tr>
<td>t₂</td>
<td>8</td>
<td>30</td>
<td>30</td>
<td>27</td>
<td>5</td>
</tr>
<tr>
<td>t₃</td>
<td>3</td>
<td>20</td>
<td>20</td>
<td>25</td>
<td>5</td>
</tr>
</tbody>
</table>
Choosing Prios: Earliest Deadline First (EDF)

• Dynamic scheduling policy
• Job with closest deadline executes
• Preemptive EDF with D=T is optimal: n jobs can be scheduled iff
  \[ U \equiv \sum \frac{C_i}{T_i} \leq 1 \]
  - necessary and sufficient condition
  - no easy test if D\#T
Resource Sharing

SMACCMcopter Mission Computer Architecture

Sharing: Critical Sections as Servers

Hoare-style monitor
Suitable intra-core

Semaphore synchronisation
Suitable inter-core

```c
server() {
  while (1) {
    wait(ep);
    /* critical section */
    reply(wait(ep));
  }
}
```

```c
client() {
  while (1) {
    call(ep);
  }
}
```

```c
server() {
  while (1) {
    wait(eap_rq);
    /* critical section */
    signal(eap_ry);
  }
}
```

```c
client() {
  signal(eap_ry);
}
```
Problem: Priority Inversion

- High-priority job is blocked for a long time by a low-priority job
- 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: \( C_2 + C_3 + C_4 \)
- Must find a way to do better!

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

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

- 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

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

- Transitive inheritance
  - potentially long blocking chains
  - potential for deadlock
- 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
    - because it allows running all jobs on the same stack (i.e. thread)
- Improved version of the original ceiling priority protocol (OCPP)
  - … which is also called the basic priority ceiling protocol
  - OPCP bumps priority to ceiling of all waiting threads
  - IPCP requires global tracking of ceiling priorities

Priority Inheritance:

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

(Immediate) Priority Ceiling Protocol
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

Immediate Priority Ceiling:

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

Comparison of Locking Protocols

Implementation Complexity

Immediate Priority-Ceiling Protocol

Priority-Inheritance Protocol

Non-Preemptible Critical Sections

Priority Inversion Bound

Scheduling Overloaded RT Systems

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

<table>
<thead>
<tr>
<th>t</th>
<th>P</th>
<th>C</th>
<th>T</th>
<th>D</th>
<th>U [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>t₁</td>
<td>1</td>
<td>15</td>
<td>50</td>
<td>50</td>
<td>30</td>
</tr>
<tr>
<td>t₂</td>
<td>2</td>
<td>12</td>
<td>20</td>
<td>20</td>
<td>60</td>
</tr>
<tr>
<td>t₃</td>
<td>3</td>
<td>5</td>
<td>20</td>
<td>20</td>
<td>25</td>
</tr>
</tbody>
</table>

Overload: FPS vs EDF

Overload: EDF
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
    - Requires massive over-provisioning
  - Some systems have effectively unbounded execution time
    - e.g. object tracking

WCET Analysis

- Program binary
- Control Flow Graph
- Micro-architecture model
- Analysis tool
- Integer linear equations
- Infeasible path info
- ILP solver
- WCET
- Accurate & sound model of pipeline, caches
- Loop bounds
- Pessimism!
- Scalability!

WCET Analysis on ARM11

- Observed
- Computed

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

Forces consolidation of multiple functions on single processor

---

**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”
    - thanks to caches, pipelines, under-specified hardware
    - requires massive over-provisioning
- **Consolidation of functionality**

**Way out?**

- Need explicit notion of importance: **criticality**
- Expresses effect of failure on the system mission
  - Catastrophic, hazardous, major, minor, no effect
- Orthogonal to scheduling priority!

---

**Mixed-Criticality Systems**

**Consolidation: Mixed-Criticality Systems (MCS)**

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

Higher criticality certification:
- More expensive
- More pessimistic WCET
  \[ \Rightarrow \text{Minimise high-crit!} \]

- Eg autopilot
- Eg sensor inputs, audit logs
- Eg flight control

- Less critical subsystem
- Shared subsystem
- Highly-critical subsystem
- TCB
- OS
Criticality ≠ Priority

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

OS Support For Mixed Criticality

MCS need strong OS-enforced isolation
- Spatial isolation: memory protection
  - Address space
- Temporal isolation: enforce CPU time limit enforcement
  - Time budget
- Criticality notion:
  - Stop LOW from overrunning budget
  - Get out of jail if HIGH overruns optimistic budget
    - Must be fast, as the cost of change must be included in analysis!

Mixed Criticality Example

<table>
<thead>
<tr>
<th>Criticality</th>
<th>T</th>
<th>U_HIGH (%)</th>
<th>U_MED (%)</th>
<th>U_LOW (%)</th>
<th>U_average (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>High</td>
<td>10</td>
<td>50%</td>
<td>20%</td>
<td>20%</td>
<td>0.05%</td>
</tr>
<tr>
<td>Medium</td>
<td>1</td>
<td>N/A</td>
<td>60%</td>
<td>20%</td>
<td>2.5%</td>
</tr>
<tr>
<td>Low</td>
<td>100</td>
<td>N/A</td>
<td>N/A</td>
<td>unknown</td>
<td>10%</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>50%</td>
<td>80%</td>
<td>over 12.55%</td>
<td></td>
</tr>
</tbody>
</table>

- HIGH alone has poor utilisation ⇒ gain from consolidation
- HIGH + MEDIUM can be scheduled for med-crit WCET
- HIGH + MEDIUM cannot be scheduled for most conservative WCET
- Idea: schedule under optimistic assumptions
  - Prioritise HIGH if it overruns its MEDIUM WCET
  - Change HIGH budget to pessimistic value

Budgets in EDF: CPU Bandwidth Reservations

- Idea: Utilisation U = C/T can be seen as required CPU bandwidth
  - Account time use against reservation C
  - Not runnable when reservation exhausted
  - Replenish every T

- Can support over-committing
  - Reduce LOW reservations if HIGH reservations fully used

- Advantages:
  - Allows dealing with jobs with unknown (or untrusted) deadlines
  - Allows integrating sporadic, asynchronous and soft tasks

- Modelled as a "server" which hands out time to jobs
  - Effectively a simple (FIFO) sub-scheduler
  - Constant-bandwidth server (CBS) [Abeni & Buttazzo '98]
Mixed Criticality Implementation

- Whenever running **LOW** job, ensure no **HIGH** job misses deadline
- Switch to *critical mode* when not assured
  - Various approaches to determine switch
    - eg. *zero slack*: **HIGH** job’s deadline = its WCET
- Criticality-mode actions:
  - FP: temporarily raise all **HIGH** jobs’ prios above that of all others
    - Simply preempting present job won’t help!
  - EDF: drop all **LOW** deadlines earlier than next **HIGH** deadline
- Issues:
  - Treatment of **LOW** jobs still rather indiscriminate:
    - EDF: switch will force deadline misses
    - FPS: **LOW** gets second chance
  - Need to determine when to switch to normal mode, restore prios
  - Switch must be fast – must be allowed for in schedulability analysis!

---

Separate Scheduling Properties & Threads

**Classical Thread Attributes**
- Priority
- Time slice

**New Thread Attributes**
- Priority
- Scheduling context capability

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

- Upper bound, not reservation!
- Not yet in mainline!

- SchedControl capability conveys right to assign budgets (i.e. perform admission control)

---

Challenge: Shared Servers

**Server**

- Running
- Server runs on client’s scheduling context

**Client**

- Running
- Client is charged for server’s time

- Budget expiry during server execution?

---

Shared Server with Scheduling Contexts

**Server**

- Running

**Client**

- Running
- Can effectively DoS same-prio threads!
Budget Expiry Options

• Multi-threaded servers (COMPOSITE [Parmer '10])
  – Forcing all servers to be thread-safe is policy 😞
  – Optional in seL4 model

• Bandwidth inheritance with “helping” (Fiasco [Steinberg ‘10])
  – Ugly dependency chains 😞
  – Wrong thread charged for recovery cost 😞

• Use timeout exceptions to trigger one of several possible actions:
  – Provide emergency budget to leave critical section
  – Cancel operation & roll-back server
  – Change criticality
  – Implement priority inheritance (if you must…)