Real-Time Systems

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!
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

Soft Real-Time Systems

- Deadline miss is undesired but tolerable
  - Frequently results on quality-of-service (QoS) degradation
    - eg audio, video rendering
    - Steep “cost” function
  - Cost of deadline miss may be abstract

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)
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

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”
  - 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
  - Many modern uses require actual OS technology for isolation
    - generally microkernels

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
### 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 characterized by period T, deadline D, and execution time C,
    - Applies to all jobs of task

- **Aperiodic tasks**
  - Event driven, characteristics are not known a priori
    - Task t characterized by period T, deadline D, and arrival distribution

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

### Standard Task Model

C: Worst-case computation time (WCET)
T: Period (periodic) or minimum inter-arrival time (sporadic)
D: Deadline (relative, frequently 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=C/T

OS terminology:
- "task" = thread
- "job" = event-based activation of thread

### Task Constraints

- **Deadline constraint**: must complete before deadline
- **Resource constraints**:
  - Shared (R/O), exclusive (W-X) access
  - Energy
- **Precedence constraints**:
  - T1 ⇒ T2: T2 execution cannot start until T1 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)

Non-Preemptive Scheduling

• Minimises context-switching overhead
  • Significant cost on modern processors (pipelinies, 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

Clock-Driven (Time-Triggered) Scheduling

• Typically implemented as time “frames” adding up to “base rate”
• Advantages
  • fully deterministic
  • “cyclic executive” is trivial
    – loop waiting for timer tick, followed by function calls to jobs
    – minimal overhead
• Disadvantage:
  • Big latencies if event rate doesn’t match base rate (hyper-period)
  • Inflexible

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” in the sense that system doesn’t change them
  • OS may support dynamic adjustment
  • Requires on-the-fly (re-)admission control
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 (as far as fixed priorities go)
- Schedulability test: RMS can schedule \( n \) tasks with \( D=T \) if
  \[
  U = \sum C_i / T_i \leq n(2^{1/n}-1) \quad \text{lim}_{n \to \infty} U = \log 2
  \]
  - sufficient but not necessary condition

<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>69.3</td>
</tr>
</tbody>
</table>
- 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)

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
  \[
  U = \sum C_i / T_i \leq 1
  \]
  - necessary and sufficient condition
  - no easy test if \( D \neq T \)
<table>
<thead>
<tr>
<th>t_1</th>
<th>t_2</th>
<th>t_3</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_1</td>
<td>3</td>
<td>5</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>25</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>t_2</td>
<td>2</td>
<td>8</td>
<td>30</td>
<td>20</td>
<td>20</td>
<td>27</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>t_3</td>
<td>1</td>
<td>15</td>
<td>40</td>
<td>40</td>
<td>40</td>
<td>37.5</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

| Misses deadline |

EDF schedules

<table>
<thead>
<tr>
<th>Overload: FPS</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>t_1</th>
<th>t_2</th>
<th>t_3</th>
<th>P</th>
<th>C</th>
<th>T</th>
<th>D</th>
<th>U [%]</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>t_1</td>
<td>t_2</td>
<td>t_3</td>
<td>3</td>
<td>5</td>
<td>20</td>
<td>20</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>t_2</td>
<td>t_3</td>
<td>t_1</td>
<td>1</td>
<td>15</td>
<td>50</td>
<td>50</td>
<td>30</td>
<td>115</td>
</tr>
</tbody>
</table>

| Misses deadline |

<table>
<thead>
<tr>
<th>Overload: FPS</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>t_1</th>
<th>t_2</th>
<th>t_3</th>
<th>P</th>
<th>C</th>
<th>T</th>
<th>D</th>
<th>U [%]</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>t_1</td>
<td>t_2</td>
<td>t_3</td>
<td>3</td>
<td>5</td>
<td>20</td>
<td>20</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>t_2</td>
<td>t_3</td>
<td>t_1</td>
<td>1</td>
<td>15</td>
<td>50</td>
<td>50</td>
<td>30</td>
<td>115</td>
</tr>
</tbody>
</table>

| Misses deadline |

EDF schedules
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
- System model
- Loop bounds
- Infeasible path info
- Analysis tool
- Integer linear equations
- ILP solver
- WCET
- Accurate & sound model of pipeline, caches
- WCET bounds
- Infeasible path info

**Pessimism!**

**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

**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**

- A mixed-criticality system supports multiple criticalities concurrently
  - Eg in avionics: consolidation of multiple functionalities
  - Higher criticality requires more pessimistic analysis, higher certification
  - Needs more than just scheduling support: strong OS-level isolation
  - In overload scheduler drops lowest criticality
- Current research issue

**sel4 WCET Analysis [Blackham et al ’11, ’12]**

- WCET presently limited by verification practicalities
  - without regard to verification achieved 50 µs
  - 10 µs seem achievable

**Critality**

<table>
<thead>
<tr>
<th>Critality</th>
<th>T</th>
<th>( U_{\text{worst}} )</th>
<th>( U_{\text{expected}} )</th>
<th>( U_{\text{average}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>High</td>
<td>10</td>
<td>50%</td>
<td>50%</td>
<td>0.05%</td>
</tr>
<tr>
<td>Medium</td>
<td>1</td>
<td>(200%)</td>
<td>10%</td>
<td>2.5%</td>
</tr>
<tr>
<td>Low</td>
<td>100</td>
<td>(100%)</td>
<td>20%</td>
<td>10%</td>
</tr>
</tbody>
</table>

**Must handle**

**Not really known**
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 drop all LOW jobs’ prios below that of critical HIGH
    - 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
  - Need to determine when to switch to normal mode, restore prios

- Alternative: use reservations

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)

- Popular theoretical model suitable for EDF [Abeni & Buttazzo ’98]
- CBS schedules specified bandwidth
  - server has a period, T and a budget, Q = U × T
  - generates appropriate absolute EDF deadlines on the fly
  - when executing a job, budget is consumed
  - when budget goes to zero, new deadline is generated with new budget
    - $D_{i+1} = D_i + T$
    - Schedulability: $\sum U_i \leq 1$

Message-Based Synchronisation

- Tasks may communicate via messages
  - blocking IPC
  - Enforces precedence relations
  - Allows sharing resources (services)
  - Tag prios/deadlines onto messages

  - Classical L4 approach: time-slice donation:
    - Receiver continues on sender’s time slice (and prio)
    - Avoids scheduler invocation
Synchronisation Issues

- Thread invoked by IPC is essentially a Hoare-style monitor
  - Typical in client-server scenario
  - Blocks other threads IPCing to same thread
  - How long?
  - Time-slice preemption during monitor?
- Worse: priority inversion – general issue with shared resources

![Diagram of thread interactions]

Shared Resources

- Problem is not restricted to synchronous communication
- Typical in client-server scenario
- Blocks other threads IPCing to same thread
- How long?
  - Time-slice preemption during monitor?
- Worse: priority inversion – general issue with shared resources

```c
// critical section */
wait(sem);
signal(sem);
```

- High-priority job is blocked, waiting for low-priority job
- Priority inversion!
- Undermines scheduling policy
- Must limit and control enough to still allow analysis of timeliness

Priority Inversion

- High-priority job is blocked for a long time by a low-priority job
- Long wait chain: \( t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow t_4 \)
- Worst-case blocking time of \( t_1 \) bounded only by WCET of \( C_2 + C_3 + C_4 \)
- Must find a way to do better!

![Priority Inversion Diagram]

Priority Inheritance (“Helping”)

- High-priority job is blocked for a long time by a low-priority job
- Long wait chain: \( t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow t_4 \)
- Worst-case blocking time of \( t_1 \) bounded only by WCET of \( C_2 + C_3 + C_4 \)
- Must find a way to do better!

![Priority Inheritance Diagram]
Priority Inheritance

• If $t_1$ blocks on a resource held by $t_2$, and $P_1 > P_2$, then
  – $t_1$ 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
  • $t_1$ 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 Inheritance:

• Easy to use, potential deadlocks
• Complex to implement
• Bad worst-case blocking times
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 prio 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
- **Improved version of the original ceiling priority protocol (OCPP)**
  - ... which is also called the basic priority ceiling protocol
  - Requires global tracking of ceiling prios

PCP 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

- In seL4:
  - Have the server run at the ceiling prio
  - Ceiling is max prio of threads holding a send cap on server EP
    - Obviously hard to determine automatically at admission time
    - Could use trusted server to hand out caps
    - In any case a user-level (system design) problem

- Challenge: proper time accounting not supported by present seL4
  - Solution: new scheduling model!

(Immediate) Priority Ceiling Protocol

New seL4 Scheduling Model
**Temporal Isolation Requirements**

1. Enforcement of CPU time limits
   - Irrespective of priority

2. Support for mixed criticality:
   - Priority orthogonal to criticality
   - Asymmetric temporal isolation: controlled overrun by hi-crit

3. Support for shared resources:
   - Server time charged to client
   - Sharing across priorities and criticalities

4. Efficient
   - Minimal overheads and algorithmic losses
   - No hierarchical scheduling

5. Policy-free mechanisms

**Learn from Resource Kernels [Rajkumar ’01]**

**Principles:**
- Timeliness through reservations
- Efficient resource utilisation
- Enforcement and protection

**Resource Kernel mechanisms:**
- Scheduling
- Enforcement
- Accounting
- Admission

**Missing:**
1. Shared resources
2. Mixed criticality

**Policy – doesn’t belong in microkernel!**

**Idea: Separate Scheduling Context from Thread**

**Old Thread attributes**
- Priority
- Time slice

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

**Upper bound, not reservation!**

**Full Budgets**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th>253</th>
<th>254</th>
<th>255</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- e = 4
- p = 4
- t1

- e = 5
- p = 5
- t2

- e = 4
- p = 4
- t3
General Budgets

![Diagram of budget allocation]

Might be trusted not to use budget, except in emergencies

**Admission**

- New capability: SchedControl
- Anyone (with access to Untyped) can create scheduling contexts
- Only holder of SchedControl cap can populate scheduling contexts
  - Trusted to implement policy
  - Authority to manage rates

**Temporal Isolation Requirements**

1. Enforcement of CPU time limits
   - Irrespective of priority
2. Support for mixed criticality:
   - Priority orthogonal to criticality
   - Asymmetric temporal isolation: controlled overrun by high-crit
3. Support for shared resources:
   - Server time charged to client
   - Sharing across priorities and criticalities
4. Efficient
   - Minimal overheads and algorithmic losses
5. Policy-free mechanisms

**Task model aka “I’m done for now”**

```c
while (1) {
    seL4_Wait (release);
    doJob();
}
```
### Criticallity

**Old Thread attributes**
- Priority
- Time slice

**New Thread Attributes**
- Priority
- Scheduling context capability
- Release notification
- Time exception handler
- Criticality

- System criticality
  - Only schedule threads with at least that criticality
  - SchedControl holder can change (on time exception)

### Asymmetric Protection

- Low Criticality
- High Criticality

- e = 100
  - p = 100

- e = 3
  - p = 20

- e = 5
  - p = 10

- e = 100
  - p = 100

- e = 30
  - p = 100

**SchedControl_SetCriticality()**

- t₀
- t₁
- t₂
- t₃
- t₄
- t₅

### Temporal Isolation Requirements

1. Bandwidth enforcement:
   - Enforced limits on CPU time consumption

2. Support for mixed criticality:
   - Priority orthogonal to criticality
   - Asymmetric temporal isolation: controlled overrun by high-crit

3. Support for shared resources:
   - Server time charged to client
   - Sharing across priorities and criticalities

4. Efficient
   - Minimal overheads and algorithmic losses

5. Policy-free mechanisms
Temporal Isolation Issues: Message-Passing (IPC)

Client is charged for server’s time

Running

Client 1

Client 2

Server

Running

Server runs on client’s scheduling context

Budget expiry during server execution?

Temporal Isolation Requirements

1. Bandwidth enforcement:
   • Enforced limits on CPU time consumption

2. Support for mixed criticality:
   • Priority orthogonal to criticality
   • Asymmetric temporal isolation: controlled overrun by high-crit

3. Support for shared resources:
   • Server time charged to client
   • Sharing across priorities and criticalities

4. Efficient
   • Minimal overheads and algorithmic losses

5. Policy-free mechanisms
   • Can implement EDF, Linux CFS, DO-178, ARINC-653, ...

Budget Expiry Options

• Multi-threaded servers (COMPOSITE [Parmer ‘10])
  • Model allows this
  • Forcing all servers to be thread-safe is policy

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

• Use temporal exceptions to trigger one of several possible actions:
  • Provide emergency budget
  • Cancel operation & roll-back server
  • Change criticality