COMP9242
Advanced Operating Systems
S2/2014 Week 9:
Real-Time Systems
@GernotHeiser
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, [Institution]”, where [Institution] is one of “UNSW” or “NICTA”

The complete license text can be found at http://creativecommons.org/licenses/by/3.0/legalcode

Note: Substantial re-use of material from Stefan M Petters (ex-NICTA)
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!
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
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)
  – Typical for control loops
  – Fixed order of actions, round-robin execution
  – Statical determined (static schedule)
    • need to know all execution parameters at system configuration time

• Event-driven
  – Typical for reactive systems (sensors & actuators)
  – Static or dynamic schedules
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 $T_i$, 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 $t$ 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 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: \( t_1 \Rightarrow t_2 \): 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 prio)
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
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
- Only used in very simple systems
Fixed-Priority Scheduling (FPS)

- Real-time priorities are absolute:
  - Scheduler always picks highest-priority job
- Fixed priorities 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 (RM) Scheduling

• RM: Standard approach to fixed priority assignment
  – \( T_i < T_j \Rightarrow P_i > P_j \)
  – \( 1/T \) is the “rate” of a task

• RM is *optimal* (as far as fixed priorities go)

• Schedulability test: RM can schedule \( n \) tasks with \( D=\)T if
  \[ U \equiv \sum C_i/T_i \leq n(2^{1/n}-1); \quad \lim_{n \to \infty} U = \log 2 \]
  • sufficient but not necessary condition

\[
\begin{array}{|c|c|c|c|c|c|c|}
\hline
n & 1 & 2 & 3 & 4 & 5 & 10 & \infty \\
\hline
U [%] & 100 & 82.8 & 78.0 & 75.7 & 74.3 & 71.8 & 69.3 \\
\hline
\end{array}
\]

• If \( D<\)T replace by *deadline-monotonic* (DM):
  – \( D_i < D_j \Rightarrow P_i > P_j \)

• DM is also optimal (but schedulability bound is more complex)
### FPS Example

**Table:**

<table>
<thead>
<tr>
<th></th>
<th>P</th>
<th>C</th>
<th>T</th>
<th>D</th>
<th>U [%]</th>
<th>release</th>
</tr>
</thead>
<tbody>
<tr>
<td>t₃</td>
<td>3</td>
<td>5</td>
<td>20</td>
<td>20</td>
<td>25</td>
<td>5</td>
</tr>
<tr>
<td>t₂</td>
<td>2</td>
<td>8</td>
<td>30</td>
<td>20</td>
<td>27</td>
<td>12</td>
</tr>
<tr>
<td>t₁</td>
<td>1</td>
<td>15</td>
<td>50</td>
<td>50</td>
<td>30</td>
<td>0</td>
</tr>
</tbody>
</table>

**Graph:**

- Release
- Deadline

**Deadline:**

82
Earliest Deadline First (EDF)

- Dynamic scheduling policy
- Job with closest deadline executes
- Preemptive EDS with D=T is optimal: n jobs can be scheduled iff $U \equiv \sum C_i/T_i \leq 1$
  - necessary and sufficient condition
  - no easy test if D≠T
FPS vs EDF

\[ t_3 \]
\[ t_2 \]
\[ t_1 \]

\[ t_3 \]
\[ t_2 \]
\[ t_1 \]
### FPS vs EDF

#### Table

<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>t₃</td>
<td>3</td>
<td>5</td>
<td>20</td>
<td>20</td>
<td>25</td>
</tr>
<tr>
<td>t₂</td>
<td>2</td>
<td>8</td>
<td>30</td>
<td>20</td>
<td>27</td>
</tr>
<tr>
<td>t₁</td>
<td>1</td>
<td>15</td>
<td>40</td>
<td>40</td>
<td>37.5</td>
</tr>
</tbody>
</table>

Average Misses deadline: 89.5%
FPS vs EDF

Misses deadline

EDF schedules
Overload: FPS

<table>
<thead>
<tr>
<th></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>

115
Overload: FPS

Old

New
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 (apparently 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”
    - thanks to caches, pipelines, under-specified hardware
    - requires massive over-provisioning
  - Some systems have effectively unbounded execution time
    - e.g. object tracking
WCET Analysis

Program binary → Control Flow Graph → System model → Analysis tool → Integer linear equations → ILP solver → WCET

- Program binary
- Control Flow Graph
- System model
- Analysis tool
- Integer linear equations
- ILP solver
- WCET

Accuracy & sound model of pipeline, caches

Pessimism!

Scalability!
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

<table>
<thead>
<tr>
<th>Criticality</th>
<th>T</th>
<th>$U_{\text{worst}}$</th>
<th>$U_{\text{expec}}$</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>(1000%)</td>
<td>20%</td>
<td>10%</td>
</tr>
</tbody>
</table>

Not really known

Must handle
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 $U = \frac{C}{T}$ 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
Constand Bandwidth Server (CBS)

- Popular theoretical model suitable for EDF [Abeni & Buttazzo ’98]
- CBS schedules specified bandwidth
  - server has a period, $T$ and a budget, $Q = U \times T$
  - 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: timeslice 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
Shared Resources

- Problem is not restricted to synchronous communication

```
t_low() {
    ....
    wait(sem);
    /* critical section */
    signal(sem);
    ...
}
t_high() {
    ....
    wait(sem);
    /* critical section */
    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-prio job
- Long wait chain: $t_1 \rightarrow t_4 \rightarrow t_3 \rightarrow t_2$
- 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 ("Helping")

- At time $t_4$, the process with priority 4 is running.
- At time $t_3$, the process with priority 3 is running.
- At time $t_2$, the process with priority 2 is running.
- At time $t_1$, the process with priority 1 is running.

The diagram illustrates the priority inheritance mechanism, where a higher-priority task (Q) preemptively runs a lower-priority task (V) when it becomes ready.
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

- 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 $P_2$

Deadlock!
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 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
(Immediate) Priority Ceiling Protocol

The diagram illustrates the Immediate Priority Ceiling Protocol with time periods $t_1$, $t_2$, $t_3$, and $t_4$. The protocol involves setting priorities and releasing resources.

- **t_1**: Setting priority for $Q$ (Priority Ceiling Protocol (PCP) at $t_1$).
- **t_2**: Setting priority for $V$ and releasing $Q$ (PCP at $t_2$).
- **t_3**: Setting priority for $Q$ again and releasing $V$ (PCP at $t_3$).
- **t_4**: Setting priority for $V$ and releasing $Q$ (PCP at $t_4$).

The protocol ensures that higher-priority tasks are executed before lower-priority tasks, maintaining resource allocation and management efficiently.
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
  – Work in progress – stay tuned!