Execution Architecture for Real-Time Systems

Dr. A.P. Kostelijk (Ton)
Content

• Discussion on performance issues
• Introductory examples
• OS: process
  context switch, process-creation, thread, co-operative / preemptive multi-tasking, scheduling, EDF, RMS, RMA.
• How to design concurrency / multi-tasking

You:
• Discussion
  • Various scheduling exercises
    • RMA exercise
Discussion on performance issues

SW


HW
Model: Levels of execution

1. Task and priority assignment

2. Algorithms, source code

3. Machine code, CPU


5. Device access
Content

- Discussion on performance issues
- Introductory examples
- OS: process context switch, process-creation, thread, preemptive multi-tasking, scheduling, EDF, RMS, RMA.
- How to design concurrency / multi-tasking

You:
- Discussion

- Various scheduling exercises
- RMA exercise
Example 1: a coffee machine

\[ a = \text{place new filter}; \quad 1 \]
\[ b = \text{add new coffee}; \quad 1 \]
\[ c = \text{fill water reservoir}; \quad 2 \]
\[ d = \text{heat water and pour}; \quad 2 \]

\[
\begin{align*}
\text{main()} & \{ \text{a()}; \text{b()}; \text{c()}; \text{d()}; \} \quad t = 6 \\
\text{main()} & \{ \text{c()}; \text{a()}; \text{b()}; \text{d()}; \} \quad t = 6 \\
\text{main()} & \{ \text{a}_i(); \text{c}_i(); \text{a}_f(); \text{b()}; \text{c}_f(); \text{d()}; \} \quad t = 4
\end{align*}
\]
Observations

• Timing requirements of actions are determined by dependency relations and deadlines.
• Hard-coded schedule of actions:
  + Reliable, easy testable
  + For small systems might be the best choice.
Example 2: a TV system

50 fields per second
Odd field
Even field
Txt lines in vertical blanking

TV signal

625

720 pixels

Remote control

\[ x_i + y_i \approx 1 \text{ ms} \]
Example 2: a TV system

• Simultaneous TV-system activities, e.g.,
  – 1) TXT processing and
  – 2) be able to respond to a Remote Cntrl key-press.
• One can include RC command checks in the TXT processing code. Mix unrelated things.
• RC-key press Timing Requirement is 0.5 s, TXT processing Timing Req. is 20 ms.
Observations revised

• Timing requirements of actions are determined by dependency relations and deadlines.

• Interrupts can be used for concurrency.
  – The RC-bit level is handled in this way.

• Hard-coded schedule:
  – it mixes unrelated functionality, and
  – lacks extendibility.
  – For small systems might be the best choice.
Device - CPU Access

- **Polling**
  
  Each x ms
  
  ‘check status’;

- **Interrupt**
  
  - signal: handshake
  
  - forces next instruction to be a predetermined routine call
Operating system, overview

• Supports **concurrency**, based on the concept of ‘**process**’, a virtual processor.

• Supports functions to handle problems caused by concurrency, e.g., mutual exclusion for a single-client resource.

• Auxiliary functions, like date/time, file system, networking, security, etc. etc.
Why multiple processes?

• **Ease of programming**: Separate programs execute quasi-parallel on a CPU.

• **Handle urgency** in particular for real-time activities.

• **Utilisation of idle time**. Continue with other processing when an activity is waiting for external response.
Processes

• A process is a **unique execution** of a program or function, managed by the OS.
  – Several copies of a program may run simultaneously or at different times.

• A process has its own state:
  – processor status (registers, IR)
  – memory
    • stack, heap, process-status
Processes and CPUs

- **Activation record**: copy of process state.
- **Context switch**:  
  - current CPU context goes out;  
  - new CPU context goes in.
Threads

- Separate memory spaces per process require a Memory Management Unit (by using virtual memory).
- **Thread = lightweight process**: a process that shares memory space with other processes.
Context switching

• Initiation?
• Switch to what other process?

• Answers = Characteristic of Operating System
  – Preemptive multitasking
Preemptive context switching

- OS saves current process’s state in an activation record.
- OS chooses next process \( p \) to run (scheduling).
- OS installs activation record \( p \) as current CPU state, and the next process resumes.
- Do CPU caches improve context switching?
Process state

- A process can be in one of three states:
  - *executing* on the CPU;
  - *ready* to run;
  - *blocking / waiting* for data.

- Context switch caused when other process is made ready, like IPC, mutex, semaphores, etc.
OS-call and scheduler

OS-call
\[ P_1 \]

Resume \( P_x \)

OS layer

Scheduler

\( AR_1 \)  \( AR_2 \)

\( AR_a \)  \( AR_b \)

Ready

Blocked
Interrupts and scheduler

P₁ P₁ Resume px

Interrupt Routine

AR₁

Interrupt

Resume

Interrupt with OS-call in Routine

OS layer

Scheduler

Ready

Blocked

AR₁ AR₂

ARₐ ARₐ
Preemptive multitasking

• Most powerful form of multitasking:
  – OS controls when contexts switch; (cause)
  – OS determines what process runs next.

• Cause:
  – interrupts, e.g., a timer,
  – inter-process-calls, etc.
  \(\rightarrow\) anything that can make a process ready to run
Flow of control with preemption

Timer-tick  TXT-slicer interrupt

cause  cause

P1  OS  P1  OS  P2

SW Animation  SW Animation  TXT processing
Embedded vs. general-purpose scheduling

• Workstations try to avoid starving processes of CPU access.
  – Fairness = access to CPU.

• Embedded systems must meet deadlines.
  – Low-priority processes may not run for a long time. Risk of starvation.
Priority-driven scheduling

• Each process has a priority.
• CPU goes to highest-priority process that is ready.
• Priorities determine scheduling policy:
  – fixed priority;
  – time-varying priorities.
  – round-robin scheduling in case of equal priorities
Priority-based Tasks

Interrupt-levels

Process / Thread

Priority

k

\vdots

2

1

m

\vdots

2

1
Priority-driven scheduling example

• Rules:
  – each process has a fixed priority (3 = highest);
  – highest-priority ready process gets CPU;
  – process continues until done.

• Processes
  – P1: priority 3, execution time 10
  – P2: priority 2, execution time 30
  – P3: priority 1, execution time 20
Priority-driven scheduling example

P3 ready \( t=18 \)
P2 ready \( t=0 \)  P1 ready \( t=15 \)
Simplified model

- Zero context switch time.
- No data dependencies between processes.
- Process execution time is constant.
- Deadline is at end of period.
- Highest-priority ready process runs.
Earliest-deadline-first scheduling

• **EDF**: dynamic priority scheduling scheme.

• Process closest to its deadline is given highest priority. In other words: the deadlines must be available.

• Requires recalculating process-priorities at every context switch-cause.
Exercise: Earliest Deadline First

<table>
<thead>
<tr>
<th>Thread</th>
<th>Period = deadline</th>
<th>Processing</th>
<th>Load</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 1</td>
<td>9</td>
<td>3</td>
<td>33.3 %</td>
</tr>
<tr>
<td>Thread 2</td>
<td>15</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>Thread 3</td>
<td>23</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

Suppose at $t = 0$, all threads are ready to process the arrived trigger.

```
0  9  15  18  23
Thread 1
Thread 2
Thread 3
```
Answer to exercise: EDF

<table>
<thead>
<tr>
<th>Thread</th>
<th>Period = deadline</th>
<th>Processing</th>
<th>Load</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 1</td>
<td>9</td>
<td>3</td>
<td>33.3 %</td>
</tr>
<tr>
<td>Thread 2</td>
<td>15</td>
<td>5</td>
<td>33.3 %</td>
</tr>
<tr>
<td>Thread 3</td>
<td>23</td>
<td>5</td>
<td>21.7 %</td>
</tr>
</tbody>
</table>

**Total Load:** 88.3 %
EDF evaluation

• EDF can utilize 100% of CPU.
• Overhead in context-switching is large.
• Deadlines (not only repetition rates) must explicitly be available in the system.

• Theoretically attractive, but hardly ever used.
Rate-Monotonic Scheduling

- **RMS**: static priority scheduling scheme.
- Priority assignment: the shorter deadline, the higher the priority.
Exercise: Rate-Monotonic S

<table>
<thead>
<tr>
<th>Thread</th>
<th>Priority</th>
<th>Period = deadline</th>
<th>Processing</th>
<th>Load</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 1</td>
<td></td>
<td>9</td>
<td>3</td>
<td>33.3%</td>
</tr>
<tr>
<td>Thread 2</td>
<td></td>
<td>15</td>
<td>5</td>
<td>33.3%</td>
</tr>
<tr>
<td>Thread 3</td>
<td></td>
<td>23</td>
<td>5</td>
<td>21.7%</td>
</tr>
</tbody>
</table>

Suppose at $t = 0$, all threads are ready to process the arrived trigger.
Answer to exercise: RMS (vs EDF)

<table>
<thead>
<tr>
<th>Thread</th>
<th>Priority</th>
<th>Period = deadline</th>
<th>Processing</th>
<th>Load</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 1</td>
<td>High</td>
<td>9</td>
<td>3</td>
<td>33.3%</td>
</tr>
<tr>
<td>Thread 2</td>
<td>Medium</td>
<td>15</td>
<td>5</td>
<td>33.3%</td>
</tr>
<tr>
<td>Thread 3</td>
<td>Low</td>
<td>23</td>
<td>5</td>
<td>21.7%</td>
</tr>
</tbody>
</table>

---

**Graphical Representation:**

- **Thread 1**:
  - Priority: High
  - Period = deadline: 9
  - Processing: 3
  - Load: 33.3%

- **Thread 2**:
  - Priority: Medium
  - Period = deadline: 15
  - Processing: 5
  - Load: 33.3%

- **Thread 3**:
  - Priority: Low
  - Period = deadline: 23
  - Processing: 5
  - Load: 21.7%

**Total Load:** 88.3%
Answer to exercise: RMS (vs EDF)

Thread 1
Thread 2
Thread 3

Version 0.1
Ton Kostelijk - Philips Digital Systems Labs
RMS evaluation

• RMS cannot utilize 100% = 1.0 of CPU, but for 1,2,3,4 … \( \infty \) processes:
  
  \[
  1.00, \ 0.83, \ 0.78, \ 0.76, \ \ldots \ \log \ 2 = 0.69.
  \]

• RMS guarantees that all processes will always meet their deadlines, for any interleaving of processes.

• With fixed priorities, context switch overhead is limited.
RMS evaluation (cont’d)

• For specific cases utilization bound higher, up to 0.88 load for large n.
• A processor running only hard-real-time processes is rare. For soft-RT less a problem.
• A lot of additional theory exists.
Real-time scheduling theory, utilization bound

- Set of n tasks with periods $T_i$, and process time $P_i$, load $u_i = P_i / T_i$,
- Schedule is at least possible when tasks are independent and:
  
  \[
  \sum_{i} u_i \leq n \left( \frac{1}{2^n} - 1 \right)
  \]

- 1.00, 0.83, 0.78, 0.76, …. $\log 2 = 0.69$. 
Content

- Discussion on performance issues
- Introductory examples
- OS: process
  - context switch, process-creation, thread, co-operative / preemptive multi-tasking, scheduling, EDF, RMS, RMA.
- How to design concurrency / multi-tasking

You:
- Discussion
- Various scheduling exercises
- RMA exercise
How to design concurrency

• Introduction: Why? Grounds?
• Timing requirements
• Active versus passive
• Execution architecture steps
• Issues resulting from concurrency
## Multiple processes?

### Why?
- Handle urgency (meet various deadlines simultaneously)
- Ease of programming unrelated functionality
- Utilisation of idle time

### Why not?
- Ease of programming for strongly related functionality
  - Reduce unpredictability
- Context-switch overhead
  - and inter-process communication
- Memory cost (e.g. multiple stacks)

**Design of concurrency is a crucial, non-trivial part of an architecture.**
Execution architecture design based on RMA.

Grounds:

• **Function to Task mapping** based on Gomaa’s CODARTS rules.
• **Scheduling of tasks** based on **RMS = rate-monotonic scheduling**
• **Deadline analysis** is known as **Rate Monotonic Analysis (RMA)**.
Execution architecture:
What are Timing Requirements?

Event / Trigger
Required deadline
Actual response

Multiple TRs: concurrent responses.
What are Timing Requirements?

• What happens if a process doesn’t finish by its timing requirement?
  – **Hard deadline**: system fails if missed.
  – **Soft deadline**: user may notice, but system doesn’t necessarily fail.

  – **Periodic** events: cyclic.
  – **Aperiod** events, e.g. user-input.
What are timing requirements?

• Event
  – external: signal: e.g. device or timer
  – active or passive = interrupt or polling
  – internal: handover some datastructure
  – Dependency tree of actions = action flow
Terminology

- **Action** = response function.
- **Task** is a virtual processor, executing a set of actions.
  - A **process** or **thread** is a sequential execution of a set of response functions, managed by the OS.
  - An interrupt routine is a function that may be triggered by an interrupt. Each **interrupt-level** can be regarded as a task, executing a set of interrupt routines.
Priority-based Tasks

Interrupt-levels

Process / Thread
Notation

Encapsulation

Thread

Queue
Passive versus Active

Objects / modules communicate via (member) functions
A passive module runs its functions on the task of a caller. The function is synchronous.
Passive versus Active

Active objects / functions defer execution to another task. The function is asynchronous, or decoupled.
Redirection

• Redirection data:
  – function pointer,
  – function arguments,
  – execution unit id.

• Can be generalized to a pattern to support simple change.

• Decouple functional and dynamical design.
Overview of exec arch steps

1. Get an overview of all triggers, actions and their timing requirements.

2. Action to task mapping

3. Task prioritisation

4. Measurement & RM Analysis

5. Tuning

Let's Party
Step 1: Inventorize triggers, actions and timing requirements
This is too complex! ?

Yes, we have a problem and it’s complex

It’s a matter of
   – beginning and
   – simplification
3 dimensions of simplification

• Highest priorities are independent of others.
  – do interrupt domain first. Reapply highest priority simplification in case it’s still difficult.

• Select critical scenario’s
  – for a TV: 1) play, 2) zap.

• Simplify by taking worst-case estimates. When it analyses to ‘trouble’, either you can relax in a more precise model, or you are in trouble.
Step 2: Action to Task mapping

Action set = Function set

Exec. Params $P$

Exec. Units $U$
Action to Task Mapping

• 1: Task Structuring: Identify potential active functions / modules.

• 2: Task Cohesion: Some may share a task.

• Criteria: see next sheet.

• Example:
  – 1) SettopBox: 80 -> 20
  – 2) TV: 150 -> 6
TM: Structuring criteria: active

• GOMAA - CODARTS structuring criteria:
  – asynchronous device I/O
  – resource monitors
  – periodic functions
  – control (object following a state-transition diagram)
  – user role (“sequential application”)
TM: Cohesion step

• Characteristics:
  – also known as ‘task-merging’
  – global scope (architect)
  – consider mapping of active objects on the same execution unit (task)
  – aim: reduce task-switching overhead and memory requirements by reducing the number of execution units
TM: Cohesion criteria

• GOMAA - CODARTS cohesion criteria:
  – temporal cohesion (= same priority)
    • different actions from the same event
    • actions with similar periods (when independent)
  – functional sequential cohesion (= no interference)
  – control cohesion (= no interference, exclusive calls)
  – Assign priorities according to deadlines.
Step 3. Task prioritisation

• Rate-monotonic =
  shorter deadline $\iff$ higher priority
Step 4: Analysis

<table>
<thead>
<tr>
<th>Specification</th>
<th>Design &amp; Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>System event</td>
<td>Period</td>
</tr>
<tr>
<td>E1</td>
<td>20</td>
</tr>
<tr>
<td>E2</td>
<td>15</td>
</tr>
</tbody>
</table>

- Situation table
- Measure processing times, and do RMA
Step 4: RMA

Situation table

<table>
<thead>
<tr>
<th>Requirements</th>
<th>Design and Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>• sys events</td>
<td>• action flow</td>
</tr>
<tr>
<td>• period</td>
<td>• execution unit</td>
</tr>
<tr>
<td>• deadline</td>
<td>• priority</td>
</tr>
<tr>
<td></td>
<td>• (shared resource)</td>
</tr>
<tr>
<td></td>
<td>• process time</td>
</tr>
</tbody>
</table>

- Calculated (< 70%, or more sophisticated)

- response time

compared
Step 5: Tuning step

• Only when a deadline is not met:
  – either the processor is idle now and then, and you could benefit more from concurrency:
    • redo from cohesion onwards
  – or some bursts of context-switches appear
    • use more cohesion here, or priority-setting should be changed
  – otherwise: speed-up critical processing part *(only now)*
Exercise!
### Issues resulting from concurrency

<table>
<thead>
<tr>
<th><strong>Issues</strong></th>
<th><strong>Means</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>• Reentrancy</td>
<td>Among others</td>
</tr>
<tr>
<td>• Synchronisation</td>
<td>• semaphores</td>
</tr>
<tr>
<td>• Shared resources</td>
<td>• separate execution unit</td>
</tr>
<tr>
<td>– Large blocking time</td>
<td>(queue + task)</td>
</tr>
<tr>
<td>– Deadlocks</td>
<td></td>
</tr>
<tr>
<td>– Starvation</td>
<td></td>
</tr>
</tbody>
</table>
Concurrency issue: Semaphores

• **Semaphore**: OS primitive for controlling access.

• Protocol:
  – Get access with \( P(s) \).
  – Perform critical region operations.
  – Release access with \( V(s) \).

• In general, initial value of \( n \) supports access to a resource of \( n \) items.

Choice of size is crucial!!
Concurrency issue: Reentrancy

• Ability of a program or function to execute multiple times concurrently.
• This requires separate data per call.
  – Either use local data (i.e., no global data) or protect global data as being a shared resource.
Concurrency issue: Synchronisation

• Use semaphores, with initial value 0.
  – P(): probeer
  – V(): verhoog

• When \( P(s) \) is called, it waits until a \( V(s) \) has happened.
Concurrency issue: Shared resources

• Example: shared date in memory, devices
• How to implement mutual exclusion:
  – disable interrupts (better: partly) or
  – disable task switching (even better: partly)
  – but what about real-time deadlines?
  – even better ...
Concurrency issue:

Shared resources (2)

- How to implement mutual exclusion (2)
  - semaphores
    - risk of priority inversion
      - ex: small kitchen, bad temper, dishwashing, a fridge
      - solution: priority inheritance / priority ceiling protocol
    - use extra “blocking time” in addition to processing time for the relevant events
  - Risk of deadlocks
    - thread decoupling: with “job queue” (e.g., I/O )
Concurrency issue: Shared resources (3) deadlocks

- Result of mutual exclusion that contain each other. Mutex calls form a cyclic graph.
- Example:
Concurrency issue: Shared resources (4) deadlocks prevention

- Exclude a cyclic order:
  - Order all modules based on their position in the entire system based on usage structure.

- Module of order N is only allowed to synchronously call methods from modules of order <N

- Example:
  - ‘down’ calls may be synchronous, but ‘up’ calls must be asynchronous (decoupled)
  - ‘down-stream calls’ are synchronous, up-stream decoupled.
Concurrency issue: Shared resources (5) deadlocks prevention

- Absence of deadlocks is guaranteed because semaphores are always passed (locked, ‘P’) in the same order, i.e., the order given by the module ordering.
- Now modules can implement their own (local) protection schemes while guaranteeing global absence of deadlocks.
- Yes, a specialized task (thread decoupling) works as well!
- Critical sections must be kept short.
Concurrency issue: Priorities: starvation

• Actually this is impossible when applying RMA with hard deadlines.
• However, an example:
  – a monkey sitting on a keyboard
Conclusion

• Design of execution architecture, by using concurrency is a crucial, non-trivial part of an architecture.
  – Requirements, function to taskmapping, analysis.
  – shared resources / synchronisation
• Upper part of the whole dynamic issue of a system ...
Model: Levels of execution

1. Task and priority assignment
2. Algorithms, source code
3. Machine code, CPU
5. Device access