Real-time systems SMD138

Lecture 15:
Repetition +
(In)famous real-time systems
Course focus

- **Concurrency** - how to write programs using parallel threads of execution
- **Reactivity** - how to write programs whose purpose is to react to events (ultimately in the form of interrupts)
- **Real-time** - how to write programs whose correctness depend on their real-time behavior
Course contents

- Introduction to real-time systems & bare metal programming in C
  - C vs. Java, pointers, type-casts, the execution stack
- Bit manipulation & hardware interfacing
  - (memory-mapped) ports, status changes, busy-waiting
- Concurrency and mutual exclusion
  - problems sequential programs, threads, critical sections
- The inner workings of a kernel
  - setjmp/longjmp, fresh stacks, the ready queue, yield, interrupts
Course contents

- Events, interrupts and reaction
  - problems with busy-waiting, the CPU as a reactive object
- A model of reactive objects
  - state, methods, self, SYNC/ASYNC, cyclic calls, deadlock, idling
- Clocks, timers and periodic execution
  - time-stamps, delays, period drift, event baselines + offsets
- Examples of reactive systems
  - counter, event filter, signal processor, clocks, reactive buffer, proxy
- Deadlines and priorities
  - WCET, timely reaction, (pre-emptive) scheduling, static/dynamic prio
Course contents

- Scheduling and feasibility
  - restricted model, RM, EDF, optimality, schedulability tests
- Priority inversion
  - sporadic tasks, DM, response times, blocking time, inheritance/ceiling
- POSIX thread programming
  - trad. OS services, thread creation, mutexes, ticks, POSIX clocks
- More on inter-process communication
  - POSIX signals & timers, event-loops, parking, select(), semaphores
- Real-time languages
  - monitors, condvars, Java threads & monitors, Ada guards & messages
Text book

- **Core parts**
  - Chapters 6 (except 6.5), 7, 8, 9, 10.6, 11.7, 12, 13 (except 13.12), 15 (except 15.9), 17

- **Parts covered lightly**
  - Chapters 1, 2, 3, 4, 15.9, 18

- **Parts not covered at all**
  - Chapters 5, 6.5, 10 (except 10.6), 11 (except 11.7), 13.12, 14, 16

- **Regarding sections on specific languages:**
  - C/POSIX parts should be **fully understood**
  - For other languages, only **key concepts** should be recognized when reading examples
Final words

Regarding details:
- The exam will not try to test knowledge about details that can easily be looked up in a manual.
- Knowing what techniques to use, and their limitations, is what is essential.

Regarding the remaining days until examination:
- Read the slides and the book, in that order.
- Don’t hesitate to ask questions if anything is unclear!
- I’ll try to check email frequently the remaining days until the exam.

Regarding your input:
- Please fill out the on-line course evaluation form!
Goals:
- To give some examples of real real-time systems that have actually been produced
- To illustrate various ways that the concept of time may affect system design
- To warn about the consequences of bad real-time system design!
- To motivate programmers to do better!
The Apollo 11 lunar lander
- Overloaded control system that almost caused abortion of the first lunar landing

The Therac-25 medical radiation machine
- Various software race conditions that caused death and serious injury in the 80’s

The Mars Pathfinder
- Spurious system resets on the Mars surface caused by priority inversion

The Ariane 5 satellite launch rocket
- Bad floating-point exception handling that lead rocket self destruction in 1996
The Apollo 11 lunar lander

Basic facts:
- First manned lunar landing (July 20 1969)
- A small 2-person spacecraft descended to the lunar surface, while mothership stayed in lunar orbit
- Both spacecraft equipped with a computer for navigation and guidance: 100 kHz 16-bit CPU, 38 K RAM & core memory, programmed in assembly language, priority-based event-driven OS of 2 K
During the landing phase, the role of the lunar lander computer was to measure altitude, and control the overall attitude of the spacecraft.

In the middle of descent, with only minutes to go before landing, the computer started to display “1202 alarm”

This alarm kept repeating itself every 10 seconds.

As the minimum altitude for aborting was approaching, the software engineers had to decide whether the alarm was fatal or not.

The advice was to ignore the alarm, and the landing also proceeded successfully.

The first lunar landing events
Lunar lander system design

Problem: computer too slow to handle all tasks concurrently

- Rendezvous radar
- Other sensors
- Landing radar

Interrupt handlers

Periodic tasks

Actuators
Solution: working modes

Descent

Switched off to avoid the load of the interrupt handler

- Rendezvous radar
- Other sensors
- Landing radar

Interrupt handler

Periodic task

Periodic tasks

Actuators

Periodic task

Switched off to avoid the load of the interrupt handler
Solution: working modes

Ascent

- Rendezvous radar
- Landing radar
- Other sensors

Interrupt handlers

Switched off to avoid the load of the interrupt handler

Periodic tasks

(disabled)
Cause of the alarms

Consequence 1: event buffer overflow every 10 s

Consequence 2: about 20% of the CPU cycles spent in the rendezvous radar interrupt handler

Not switched off, due to oversight in landing procedures!
The engineers knew “1202 alarm” meant overflow of some event buffer, not clear which one.

However, because of the fixed priorities used, judgement was that whatever events and computations being lost, they would be the least important ones.

The decision to ignore the alarms was taken on basis of that “gut feeling”, rather than knowledge.

Analysis and simulations during the 24h moon stay found the exact alarm cause (the erroneous switch).

(The engineer in charge of the decision to proceed was later awarded the president’s medal together with the astronauts!)
The Therac-25 was a computer controlled therapeutic radiation machine for the treatment of tumors. Deployed in the mid 80's, it was a modernized successor to a highly successful, but slow and bulky machine: the Therac-20. Massive radiation overdoses generated by the machine resulted in deaths or severe injuries for at least six people in the USA and Canada 1985-1987. The machine was redesigned in 1987 as the result of multiple federal investigations.
Therac-25 functionality

Electron mode

Low energy electron beam

X-ray mode

High energy electron beam

X-ray beam

Tungsten shield on turntable
DEC PDP-11 responsible for
- Scanning operator input
- Positioning the turntable with the tungsten shield
- Setting up the electron gun, bending magnets, and various other devices
- Performing treatment timing
- Executing extensive safety checks

Controlled via an ASCII-based terminal in a remote room, using cursor keys for moving between input fields
Therac-25 software design

Therac-25 software design

Shared memory area

- word
- phase_number,
- entry_complete_flag,
- offset_params,
- work_mode,
- energy,
- callibration_table[X],
- malfunction_flag,
- loop_counter,
- ...

No synchronization besides flag variables!!
Reconstructed accident cause 1

- The operator erroneously enters X-ray mode, realizes the mistake, and switches back to electron mode – all within 8 seconds.
- During that time window, the treatment phase task is ignoring the keyboard entry flag because it is delaying in a busy-wait loop while bending magnets are being set up. Thus the new mode is never copied over to the variable read by the gun emission control task.
- The other tasks register the edit, though, so the turntable is moved and the screen updated accordingly.
- This results in the patient being exposed to unshielded, high energy radiation, with no indication to the operator.
Reconstructed accident cause 2

- When input parameters are unverified or inconsistent, the treatment monitor task periodically runs a procedure that increments a counter.
- This counter is used as a flag by the housekeeping task, indicating whether gun firing should be enabled or not.
- However, as the counter is only 8 bits, it will overflow every 256 ticks, and the “flag” will temporarily indicate a zero condition!
- If the “set” command is given at that instant, inconsistencies are not checked, and unshielded high-energy radiation may result.
The Therac-25 software was written in assembly language by a single person, who also wrote the context-switching “kernel.” Few people seem to have had any clear idea of how the software really worked. Safety analyses for the machine never took timing errors into account. It turned out that the Therac-20 also contained the same bugs, but there hardware sensors and fuses were used as an extra safety net against high radiation. Basic lesson: never synchronize using flags!
The Mars Pathfinder

- Unmanned spacecraft that landed on Mars in 1997
- Famous for its high-resolution panorama pictures of the Mars surface
- Also famous for utilizing balloons in a “bouncing” landing procedure, and for deploying a surface vehicle on wheels
- Not so well-known: severe computer problems on the Pathfinder that resulted in frequent system resets and loss of data
- The cause: a classical example of priority inversion
Pathfinder software design

- **Info bus administration** (high priority)
- **Meterological data gathering** (low priority)
- **Communication** (mid priority)
- **(Other tasks)**
The priority inversion problem

1. Locks bus for writing
2. Tries to lock, blocked
3. Activated by interrupt, takes over CPU
4. Blocked very long
5. Watchdog task notices info bus admin isn’t running, assumes a fatal error, and issues a system reset
The Pathfinder software ran under the VxWorks RTOS. VxWorks can be run with a tracing option, which records every activity within in the kernel. During simulations on an exact replica of the Pathfinder, engineers were able to reproduce the resets, and the problem was identified. Fortunately, a C interpreter had been left in the Pathfinder’s version of VxWorks, and a fix could be uploaded that turned on priority inheritance for the info bus mutex. (The mutex in question was actually hidden within a higher-level synchronization construct, that didn’t offer any priority inheritance option!)
Launch of Ariane-5 took place in French Guyana, June 4 1996

The rocket contained a fairly sophisticated, fault-tolerant computer system, with software written in Ada

About 40 seconds after take-off, the rocket self-destructed at an altitude of approximately 4000 meters

Telemetry data, and memory readouts from computer units recovered from the debris, enabled the cause of events to be determined at a high level of detail
Part of the Ariane-5 design

- Gyro
- Inertial Reference System computer (SRI 1)
- Inertial Reference System computer (SRI 2)
- On-board computer (OBC)
- Engine nozzles
Reconstructed event-chain

- The rocket started to disintegrate at 39 s after lift-off, which caused automatic self-destruction.
- Disintegration was the result of an angle-of attack of more than 20°, which in turn was caused by full nozzle deflections on all engines.
- Nozzle deflections were commanded by the OBC software on basis of data transmitted by SRI2.
- SRI2 was actually not transmitting inertial data, but bit-patterns corresponding to post-mortem debug information.
- SRI2 had aborted because of an unhandled floating-point exception, and so had SRI1 one cycle earlier.
The FP exception was due to an error fitting a 64-bit floating-point value into a 16-bit integer.

Exception handling for this conversion had been turned off in order to squeeze CPU utilization below 80% (as dictated by RM analysis).

The unexpected value occurred in a task used for guiding the rocket while still at the launch pad.

This task was left running for 40 s after lift-off, due to extra time allocated in case of short pauses during countdown (to avoid realigning the gyros).

The analysis that outruled exceptional values for the variable after lift-off was most likely based on trajectory data for Ariane-4!
Conclusions

- If something can go wrong in a real-time concurrent system, it will eventually go wrong.
- Because of the time dimension, the argument “it works now” has little value for a real-time system.
- Correctness of a real-time system should ideally be established by some form of formal verification, with clearly stated assumptions (including timing).
- Current technology only allows us to go a few steps in this direction.
- Extensive testing can find many errors, but never give full correctness guarantees.
- We as programmers can help by choosing program structures that are clearly correct by construction.