Multicore for safety-critical embedded systems: challenges and opportunities

Giuseppe Lipari

CRIStAL - Émeraude

March 15, 2016

Outline

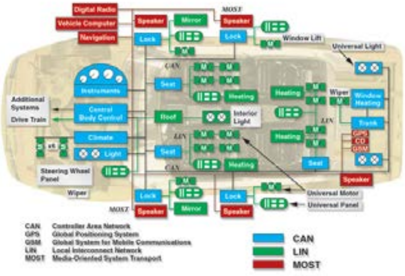
1. An history of multicore in automotive
2. Multicore scheduling
3. Current platforms

Outline

1. An history of multicore in automotive
2. Multicore scheduling
3. Current platforms

Why multicore systems?

- In the past:
  - one functionality → one board (ECU)
  - lot of network cables
  - high end car → tens of ECUs



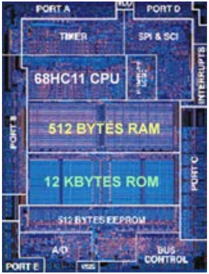
- Need to integrate functions into single boards (ECUs)
  - Reduces cabling
  - Reduces total cost
- Example:
  - Power train and gear-shift in one single board
  - Increased chances of fast coordination

Challenges

- From single core to multi-core, what exactly is going to change?
  - do we need a new programming model?
  - can we re-use existing code?
  - how can we perform (real-time) analysis?
- We will come back to these questions after a historical perspective

Historical perspective

A typical processor architecture for automotive applications in 1998/2000:



- Sample SoC
  - 68HC11 micro
  - 12Kb ROM
  - 512 bytes RAM in approximately the same space
  - No cache, no MMU

- Processors for automotive would feature up to 16 Kb RAM
  - Need to optimise RAM as much as it was possible
  - Put all constants and code into ROM

The OSEK/VDX standard

- Automotive applications are programmed in C using the OSEK/VDX interface standard
  - Interrupts + simple tasks
  - Application code and kernel linked together in the same memory space
  - Periodic (clock-driven) tasks:
    - periodic sampling of sensor data, execution of control algorithms
  - Aperiodic (event-driven) tasks:
    - Activated by external events (interrupts)
    - Example: network drivers, drive shaft
  - Heavy use of global variables
    - to reduce the amount of stack memory
  - A configuration file (OIL) is used to create the tasks and initialize the static parameters

Run-to-completion

OSEK adopts two models of tasks

- Normal tasks

```
Task mytask() {
    int local;
    initialization();
    for (;;) {
        do_instance();
        end_instance();
    }
}
```

- One stack per task is needed

- Run-to-completion tasks

```
int local;
TASK(mytask) {
  do_instance();
}

int main() {
  initialization();
}
```

- Stack frame is created and destroyed at each instance

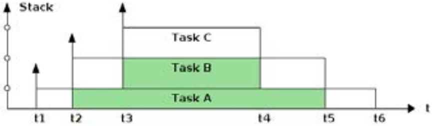
Priority-based execution and stack

- How much stack space do we need?
  - Suppose we have one RTC task for each priority level
  - then, in the worst case

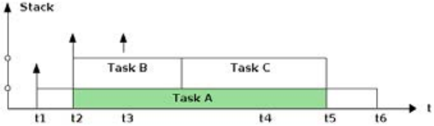
$$\text{Stack}_{tot} = \sum_i \text{Stack}_i$$

  - To reduce the size of the stack, we can reduce preemption
    - increasing delay

- Free preemption
  - C can preempt B



- Preemption threshold
  - C cannot preempt B

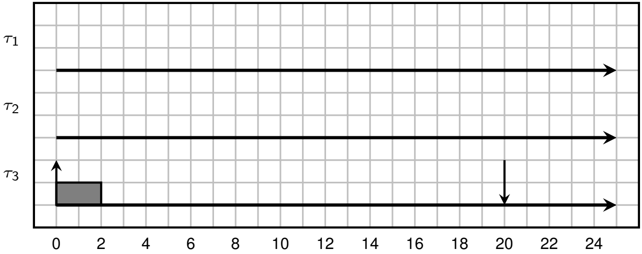


Blocking

- The technique for reducing stack works if the task does never block
- However, tipically a task access shared memory
  - and given the high number of global variables, this is quite likely to happen
- How to guarantee that the program remains consistent?
  - use mutex semaphores
- However, two problems arise
  - Interleaving (no stack can be shared)
  - priority inversion

Priority Inversion

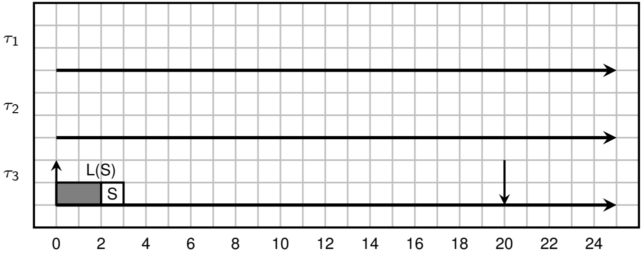
Priority inversion:



- Two problems
  - unbounded priority inversion: τ_2 delays τ_1
  - Interleaving (no stack-based execution)

Priority Inversion

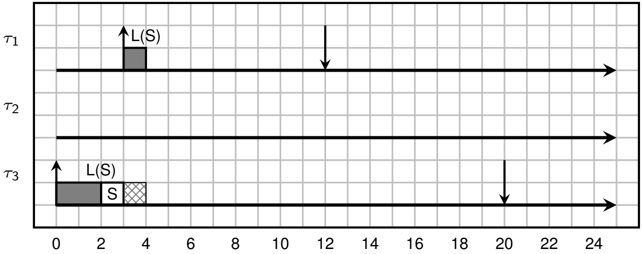
Priority inversion:



- Two problems
  - unbounded priority inversion: τ_2 delays τ_1
  - Interleaving (no stack-based execution)

Priority Inversion

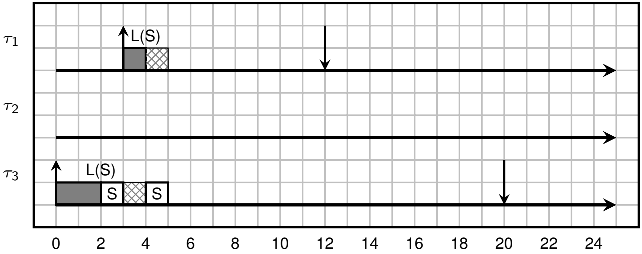
Priority inversion:



- Two problems
  - unbounded priority inversion: τ_2 delays τ_1
  - Interleaving (no stack-based execution)

Priority Inversion

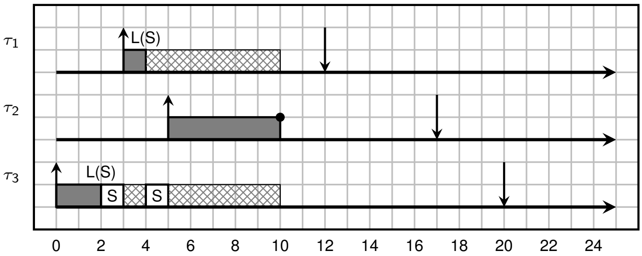
Priority inversion:



- Two problems
  - unbounded priority inversion: τ_2 delays τ_1
  - Interleaving (no stack-based execution)

Priority Inversion

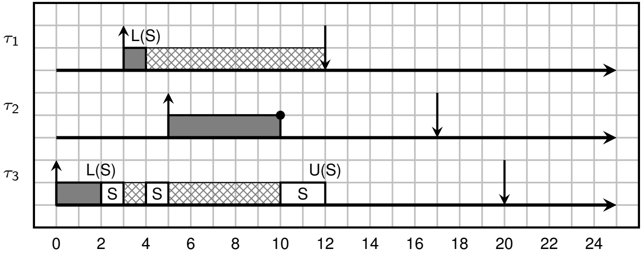
Priority inversion:



- Two problems
  - unbounded priority inversion: τ_2 delays τ_1
  - Interleaving (no stack-based execution)

Priority Inversion

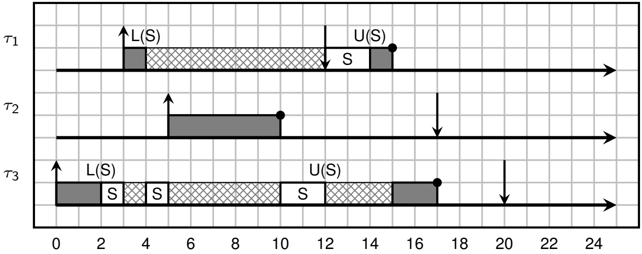
Priority inversion:



- Two problems
  - unbounded priority inversion: τ_2 delays τ_1
  - Interleaving (no stack-based execution)

Priority Inversion

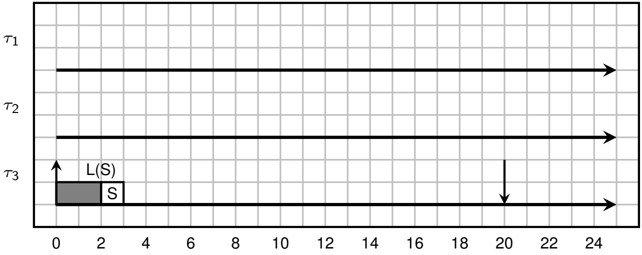
Priority inversion:



- Two problems
  - unbounded priority inversion: τ_2 delays τ_1
  - Interleaving (no stack-based execution)

Priority Inheritance

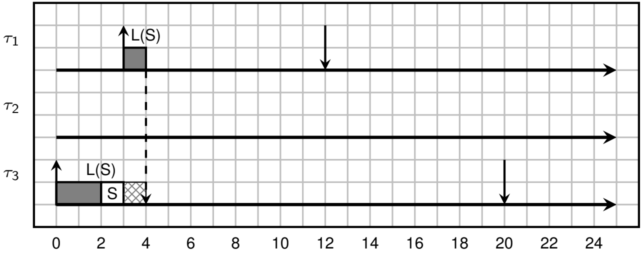
- Priority Inheritance consisting in giving the locking tasks the priority of the locked task



- We still have interleavings, so it cannot be used in a stack-based execution

Priority Inheritance

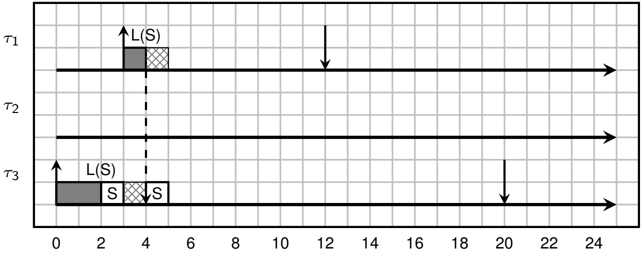
- Priority Inheritance consisting in giving the locking tasks the priority of the locked task



- We still have interleavings, so it cannot be used in a stack-based execution

Priority Inheritance

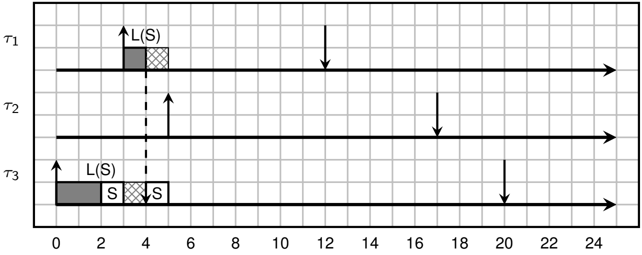
- Priority Inheritance consisting in giving the locking tasks the priority of the locked task



- We still have interleavings, so it cannot be used in a stack-based execution

Priority Inheritance

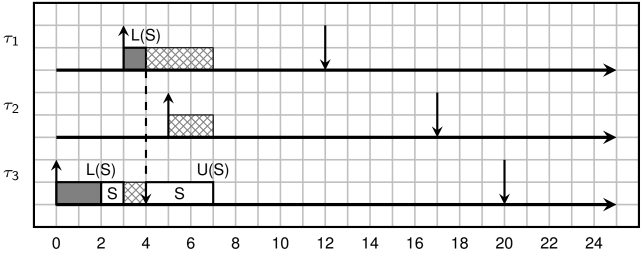
- Priority Inheritance consisting in giving the locking tasks the priority of the locked task



- We still have interleavings, so it cannot be used in a stack-based execution

Priority Inheritance

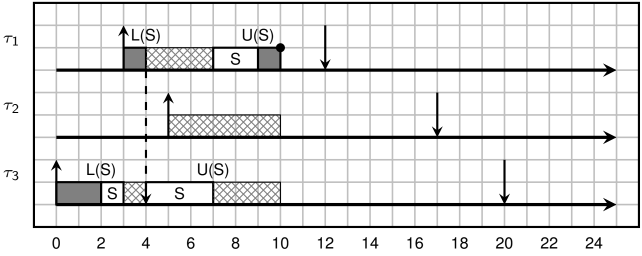
- Priority Inheritance consisting in giving the locking tasks the priority of the locked task



- We still have interleavings, so it cannot be used in a stack-based execution

Priority Inheritance

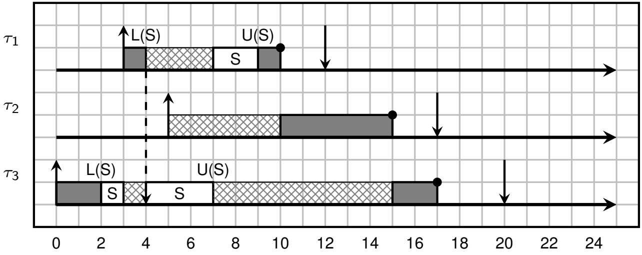
- Priority Inheritance consisting in giving the locking tasks the priority of the locked task



- We still have interleavings, so it cannot be used in a stack-based execution

Priority Inheritance

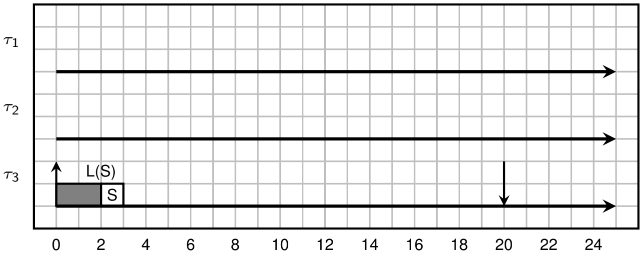
- Priority Inheritance consisting in giving the locking tasks the priority of the locked task



- We still have interleavings, so it cannot be used in a stack-based execution

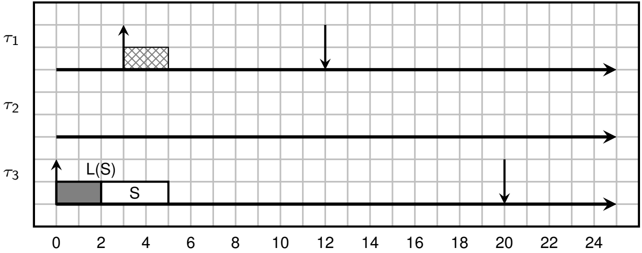
The Stack Resource Policy

- Solution: selectively disable preemption
  - resource ceiling: highest priority of all tasks that use that semaphore
  - system ceiling: highest resource ceiling of all locked semaphores
  - A task cannot start execution unless prio > sysceiling
- The previous example



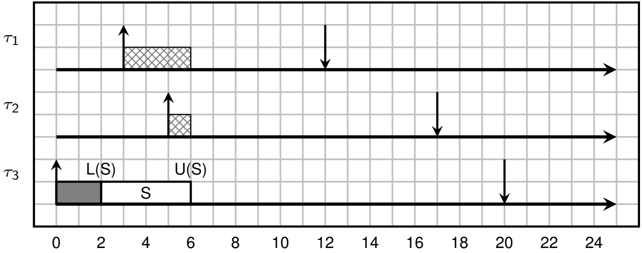
The Stack Resource Policy

- Solution: selectively disable preemption
  - resource ceiling: highest priority of all tasks that use that semaphore
  - system ceiling: highest resource ceiling of all locked semaphores
  - A task cannot start execution unless prio > sysceiling
- The previous example



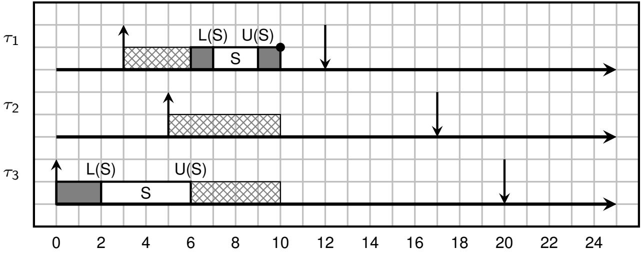
The Stack Resource Policy

- Solution: selectively disable preemption
  - resource ceiling: highest priority of all tasks that use that semaphore
  - system ceiling: highest resource ceiling of all locked semaphores
  - A task cannot start execution unless prio > sysceiling
- The previous example



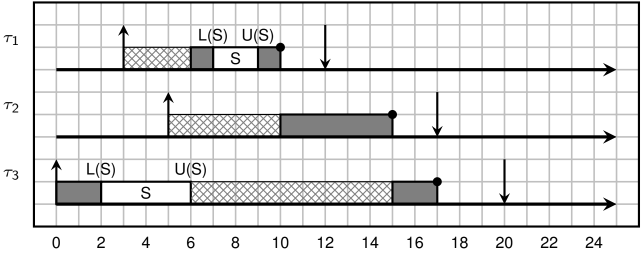
The Stack Resource Policy

- Solution: selectively disable preemption
  - resource ceiling: highest priority of all tasks that use that semaphore
  - system ceiling: highest resource ceiling of all locked semaphores
  - A task cannot start execution unless prio > sysceiling
- The previous example



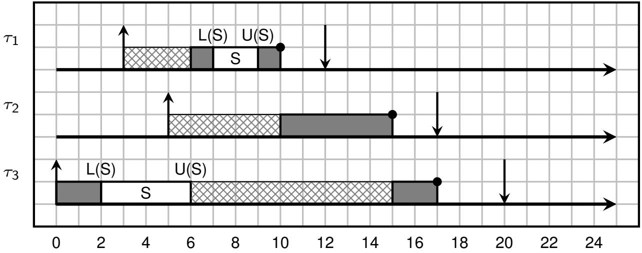
The Stack Resource Policy

- Solution: selectively disable preemption
  - resource ceiling: highest priority of all tasks that use that semaphore
  - system ceiling: highest resource ceiling of all locked semaphores
  - A task cannot start execution unless prio > sysceiling
- The previous example



The Stack Resource Policy

- Solution: selectively disable preemption
  - resource ceiling: highest priority of all tasks that use that semaphore
  - system ceiling: highest resource ceiling of all locked semaphores
  - A task cannot start execution unless prio > sysceiling
- The previous example



- No interleavings!
  - Max blocking time = maximum lenght of critical sections

Outline

1. An history of multicore in automotive

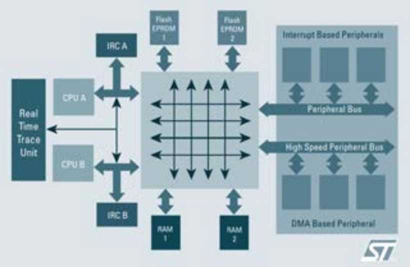
2. Multicore scheduling

3. Current platforms

Going multicore

- This was the state of the art for automotive RT software in 1999
- Around 2000, ST Microelectronics, Magneti Marelli (FIAT Group), and PARADES, decided to design a double core chip, code name JANUS

- symmetric dual processor (2 ARM7TDMI)
- 2 RAM banks, connected through a crossbar switch
- specialized I/O for engine control
- 11% additional silicon area with respect to single-ARM solution



Challenges: partitioned or global?

Main problem: how to program this new architecture?

- Need to adapt the OSEK standard to deal with multicore systems
- simplest choice: partitioned scheduling
  - tasks are statically allocated to processors

Requirements:

- Stack minimization
- Same interface

Proposal:

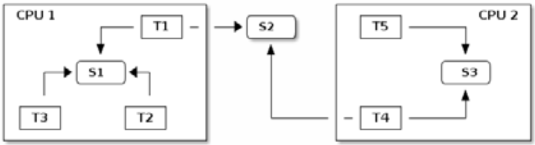
- Researchers at Scuola Sant'Anna founded a spin-off Evidence S.r.l.
  - to produce a new μ-kernel called ERIKA
- Main ideas:
  - Static allocation of tasks to processors (extending OIL language)
  - Extension of the SRP protocol to support multicore systems

Challenges: shared resources

- Tasks allocated to different processors may access the same memory
  - need semaphores
  - however, priority tricks do not work
- We proposed the M-SRP protocol
  - uses spin-locks to extend SRP

The M-SRP

- Local resources = used only by tasks allocated on the same processor
  - use standard SRP technique
- Global resources = used by tasks on different processors
  - use spin-lock



- Spin-lock (BW – Busy Waiting)
  - if a task accesses a semaphores locked by another task on a different processor, it start to busy-wait
  - while in BW, raises the ceiling to the maximum (no preemption is possible)
    - if necessary, disables interrupts
  - Multiple tasks in BW are served in FIFO order

M-SRP properties

- Paper:

  Paolo Gai, Giuseppe Lipari, Marco Di Natale, Stack Size Minimization for Embedded Real-Time Systems-on-a-Chip, Design Automation for Embedded Systems, vol. 7, n. 1-2, pp. 53–87 2002

- Stack-based execution
  - can be used to reduce the total stack size using preemption thresholds
- Deadlock
  - still possible if nested critical sections on global resources
  - can be detected by static analysis
- Bounded blocking time
  - one critical section for local resources
  - sum of critical sections for global resources
- Minimise global resources
  - by properly allocating tasks
- Accepted in the AUTOSAR 4.0 standard in 2014

The end

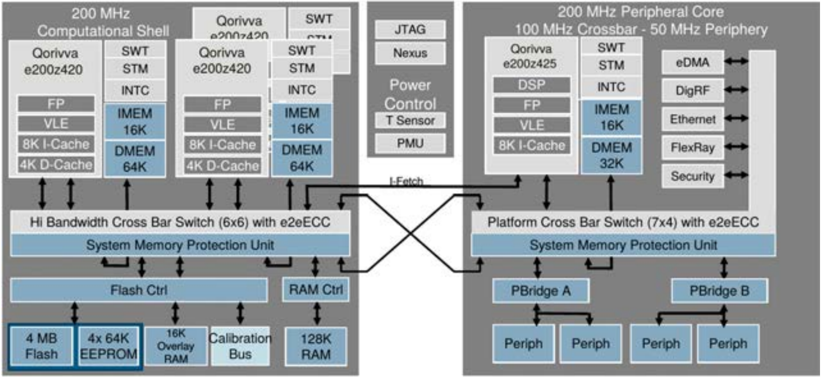
- Unfortunately, the Janus project was cancelled after 2 years for *lack of interest*
  - Clients in automotive did not know how to use a multicore
  - They were too afraid of possible software problems
- Multicore systems have started to be accepted in automotive since 2010, more than 10 years after the Janus project
- Evidence continued development of ERIKA for
  - single and multi-core platforms
  - reconfigurable FPGAs (Altera Nios)

Outline

1. An history of multicore in automotive
2. Multicore scheduling
3. Current platforms

Toward a "real" symmetric processor

- Since 2000, many things have changed
  - Memory is less costly, chips can feature several kB of memory
  - they now include MMU and caches
- Freescale Qorivva 32-bit MCU



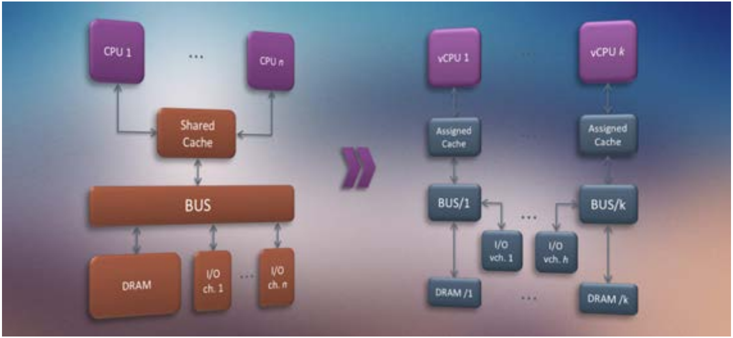
Additional features

- Fault-tolerance
  - there is an interest in using the two processors in *lock-step* mode
  - The two processors execute the same instructions and results are compared, to quickly detect faults
- Certification and isolation are the two main keywords
- Certification of multicore automotive software is still an open challenge
  - a lot of work in proof of μ-kernel
  - Still difficult for a complex distributed system
- Memory and temporal isolation are needed
  - when integrating software from third parties into the same ECU, guarantee that one does not jeopardise the others
- Additional functionalities

Isolation

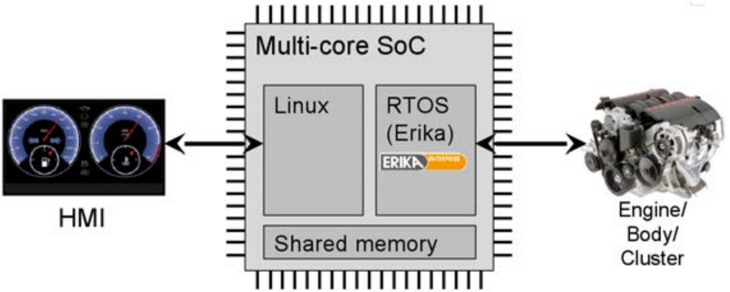
- One of the main drivers of multi-core technology
  - possibility to integrate different applications in the same ECU
  - thus reducing the number of ECUs and the lenght of the network cables
- It is important to isolate one application from another
  - Applications can have different level of criticalities
  - But they share the same memory, bus, etc.
  - They are developed by different companies / teams
- Isolation:
  - Memory protection via MMU
  - Time Triggered Access to bus
  - Separate caches, cache coloring or cache locking

Single core equivalence



- It uses a technique called *MemGuard* for sharing the bus
  - Can be implemented in software on all architectures
  - At the OS level: the task is blocked after a certain number of cache misses
  - Done @ UIUC

Experiments with Virtualization



- One core runs Linux, for the command interfaces
- One core runs ERIKA for low level – critical control sw
- Timing isolation for separating access to processors and resources

Certification?

- Interactions: From Linux it is possible to
  - Stop and reload the RTOS (ERIKA)
  - Set an Alarm
  - Activate a critical task
  - Increment a counter
- Verifying the whole system is impossible
  - Linux is too big to be verified
- However, there is some hope to verify
  - The hypervisor (XEN)
  - the RTOS (ERIKA)
- One viable approach?
  - By reasoning in terms of *isolation*, we could perform a "component-based" verification of the critical part

Conclusions

- Automotive Embedded Systems have evolved during the last 15 years
  - Introduction of multi-core chips
  - MMU and caches
- Requirements are always the same
  - Reduce the amount of memory used by the application
  - Real-time constraints
  - Certification
- New requirements
  - Fault-tolerance
  - Reduce energy consumption
  - Integrate application with different criticality levels in the same ECU
  - Component-based certification
- There is still space for doing research at the system level
  - timing isolation
  - predictability