Resource sharing in (hierarchical) real-time systems
– SAN meeting –

Martijn van den Heuvel - Reinder J. Bril - Johan J. Lukkien

System Architecture and Networking (SAN)
Department of Mathematics and Computer Science
Eindhoven University of Technology
The Netherlands

4th March 2011
Outline

1. Introduction to real-time systems
2. Task Synchronization
3. An alternative approach: Stack Resource Policy (SRP)
4. Concluding remarks
5. Hierarchical Scheduling Recapitulated
6. Inter-subsystem resource sharing
7. Conclusions
Many embedded devices provide **multiple, integrated functionalities**.

In such systems it's important to deliver correct functionality **on time**.

**Non-real-time systems**
- Correct function if produced result is correct

**Real-time systems**
- Correct function if produced result is correct and **delivered on time**
  - System does the right thing
  - ...and it does it on time

These functionalities share both **logical** and physical **resources**.
A *real-time system* is a system which has to fulfill:

1. **Functional correctness**
2. **Timeliness correctness**

---

courtesy by Dr. Isovic
Definition

A real-time system is a system which has to fulfill:

1. Functional correctness
2. Timeliness correctness

Requires Analysis!
Definition

A *task* is a sequence of actions that must be performed

Definition

A *job* is an instantiation of a task
Real-time Task Model

Definition

A task is a sequence of actions that must be performed

Definition

A job is an instantiation of a task

Properties of a real-time task:

1. Timing parameters known upfront, i.e.:
   - minimum interarrival time,
   - worst-case computation time,
   - deadline.

2. We assume a fixed priority assignment;

3. Tasks may use shared resources,
   e.g. buffers, memory-mapped devices, OS services.
A practical view: MicroC/OS-II

MicroC/OS-II is

- a commercial RTOS
- targeted at embedded systems
- open source
- available at http://micrium.com/

It provides

- a portable and configurable kernel
- a fixed-priority, preemptive task/thread scheduler
- basic services (mailboxes, mutexes and counting semaphores)
Visualization of scheduling behavior:

M. Holenderski, M. van den Heuvel, R. J. Bril, and J. J. Lukkien. Grasp: Tracing, visualizing and measuring the behavior of real-time systems.
In 1st WATERS, July 2010.

MicroC/OS-II port to OpenRISC platform

OpenRISC: Architectural Simulator
http://opencores.org/openrisc, or1ksim
Outline

1. Introduction to real-time systems
2. Task Synchronization
3. An alternative approach: Stack Resource Policy (SRP)
4. Concluding remarks
5. Hierarchical Scheduling Recapitulated
6. Inter-subsystem resource sharing
7. Conclusions
MicroC/OS-II: Basic Synchronization Services

- Disable and enable interrupts
  - share variables or data structures with an ISR.

Software events:
- Mailboxes: communicate messages between tasks; FIFO or Priority-based.
- Flags: synchronize the occurrence of multiple events.
- (Counting) semaphores: e.g. signaling of tasks.
- Mutexes: resolve priority inversion.

Blocking for an event means keeping track of waiting tasks!
MicroC/OS-II: Basic Synchronization Services

- Disable and enable interrupts
  - share variables or data structures with an ISR.

Software events:

- Mailboxes
  - communicate messages between tasks;
  - FIFO or Priority-based.
- Flags
  - synchronize the occurrence of multiple events.
- (Counting) semaphores
  - e.g. signaling of tasks
- Mutexes
  - resolve priority inversion
MicroC/OS-II: Basic Synchronization Services

- Disable and enable interrupts
  - share variables or data structures with an ISR.

Software events:
- Mailboxes
  - communicate messages between tasks;
  - FIFO or Priority-based.
- Flags
  - synchronize the occurrence of multiple events.
- (Counting) semaphores
  - e.g. signaling of tasks
- Mutexes
  - resolve priority inversion

Blocking for an event means keeping track of waiting tasks!
The easiest and fastest way to gain exclusive access to a shared resource
MicroC/OS-II: Disable and Enable Interrupts

The easiest and fastest way to gain exclusive access to a shared resource

Be careful, because:

- disabling interrupts for too long affects the response to interrupts (i.e. interrupt latency);
- a non-preemptive system becomes easily unschedulable.
The easiest and fastest way to gain exclusive access to a shared resource

**Be careful, because:**
- disabling interrupts for too long affects the response to interrupts (i.e. interrupt latency);
- a non-preemptive system becomes easily unschedulable.

**However,**
- it is the only way that a task can share variables or data structures with an ISR;
MicroC/OS-II: Disable and Enable Interrupts

The easiest and fastest way to gain exclusive access to a shared resource

Be careful, because:
- disabling interrupts for too long affects the response to interrupts (i.e. interrupt latency);
- a non-preemptive system becomes easily unschedulable.

However,
- it is the only way that a task can share variables or data structures with an ISR;

Keep interrupts disabled for as little time as possible!
The price of increased preemptiveness:

What really happened on Mars Rover Pathfinder?
Mutual Exclusion: Priority Inheritance Protocol

A solution:
Mutual Exclusion: Priority Inheritance Protocol

A solution:

Multi-resource issues:
- deadlocks;
- chained-blocking.
microC/OS-II’s mutexes: Priority calling

Priority calling is similar to *priority inheritance protocol* (PIP):

**Priority Inheritance Rule:**
when a higher-priority task blocks on a resource, the lower-priority task holding the resource inherits the higher priority;
Priority calling is similar to *priority inheritance protocol* (PIP):

**Priority Inheritance Rule:**

when a higher-priority task blocks on a resource, the lower-priority task holding the resource inherits the higher priority;

<table>
<thead>
<tr>
<th>Priority Calling</th>
<th>Priority Inheritance</th>
</tr>
</thead>
<tbody>
<tr>
<td>unique priority for each resource</td>
<td>run-time priority adaption</td>
</tr>
<tr>
<td>inherits a predefined priority</td>
<td>transitive priority inheritance</td>
</tr>
<tr>
<td>non-transparent</td>
<td>transparent</td>
</tr>
</tbody>
</table>
Priority calling is similar to *priority inheritance protocol* (PIP):

**Priority Inheritance Rule:**
when a higher-priority task blocks on a resource, the lower-priority task holding the resource inherits the higher priority;

<table>
<thead>
<tr>
<th>Priority Calling</th>
<th>Priority Inheritance</th>
</tr>
</thead>
<tbody>
<tr>
<td>unique priority for each resource</td>
<td>run-time priority adaption</td>
</tr>
<tr>
<td>inherits a predefined priority</td>
<td>transitive priority inheritance</td>
</tr>
<tr>
<td>non-transparent</td>
<td>transparent</td>
</tr>
</tbody>
</table>

It is not: Highest Locker Protocol (HLP) or Stack Resource Policy (SRP).

- microC/OS-II: a task inherits a higher priority *only* when a higher priority task is blocked;
- in HLP/SRP a task immediately inherits a priority *when it locks a resource.*
microC/OS-II: protocol classification

microC/OS-II’s synchronization protocol suffers from deadlock:

Legend:
- active
- holding
- mutex

Conclusion:
microC/OS-II implements a non-transparent, priority-inheritance protocol; but in other OSes...

microC/OS-II’s synchronization protocol suffers from deadlock:

\[
\begin{array}{c}
\text{Task1} \\
\text{Task2} \\
\text{Idle}
\end{array}
\]

Legend:
- active
- holding
- mutex

Conclusion: microC/OS-II implements a non-transparent, priority-inheritance protocol; But in other OSes...

D. Polock and D. Zobel.
Conformance testing of priority inheritance protocols.
1 Introduction to real-time systems
2 Task Synchronization
3 An alternative approach: Stack Resource Policy (SRP)
4 Concluding remarks
5 Hierarchical Scheduling Recapitulated
6 Inter-subsystem resource sharing
7 Conclusions
Stack Resource Policy (SRP)

- Each resource has a statically determined *resource ceiling*:

**Definition of a resource ceiling:**
the maximum priority of any task that could use the resource.
Stack Resource Policy (SRP)

- Each resource has a statically determined *resource ceiling*:

**Definition of a resource ceiling:**
the maximum priority of any task that could use the resource.

- A dynamically updated *system ceiling* is maintained:

**Definition of the system ceiling**
the maximum resource ceiling of any resource currently being locked in the system.
Stack Resource Policy (SRP)

- Each resource has a statically determined *resource ceiling*:

**Definition of a resource ceiling:**
the maximum priority of any task that could use the resource.

- A dynamically updated *system ceiling* is maintained:

**Definition of the system ceiling**
the maximum resource ceiling of any resource currently being locked in the system.

- A task can only be selected for execution if
  1. it has the highest priority among all ready tasks;
  2. its priority is higher than the current system ceiling.
SRP properties

- SRP provides **non-blocking primitives**: blocking occurs upon an attempt to preempt, rather than upon an attempt to access a resource (**no wait queues**).
SRP properties

- SRP provides non-blocking primitives:
  - blocking occurs upon an attempt to preempt, rather than upon an attempt to access a resource (no wait queues).
- it therefore allows tasks to share their execution stack;

Sharing of the execution stack is possible, if two tasks execute non-interleaved.

Diagram:
- T1
- T2
- interleaved execution
- T1
- T2
- T3
- not interleaved execution
SRP properties

- SRP provides **non-blocking primitives**: blocking occurs upon an attempt to preempt, rather than upon an attempt to access a resource (**no wait queues**). It therefore allows tasks to share their execution stack;

**Sharing of the execution stack is possible, if**
two tasks execute non-interleaved.

- Why does stack-sharing not work in microC/OS-II?
SRP’s Implementation Consequences

- SRP is non-transparent:
  - similar to microC/OS-II’s (PIP-based) priority calling implementation.
  - each resource has a statically computed ceiling (represents the highest priority task that uses the resources).
SRP’s Implementation Consequences

- SRP is non-transparent:
  - similar to microC/OS-II’s (PIP-based) priority calling implementation.
  - each resource has a statically computed ceiling (represents the highest priority task that uses the resources).

- Maintaining the system ceiling can be implemented using a stack data structure:
  - we stack the resource ceilings of used resources in a monotonically increasing order;
  - the top of the stack represents the system ceiling.
microC/OS-II: Our SRP extension

Deadlocks are resolved:

Legend:
- active
- holding
- mutex

Easy implementations (approx. 170 lines of code); Extended microC/OS-II scheduler with SRP’s preemption rule.
microC/OS-II: Our SRP extension

Deadlocks are resolved:

- Easy implementations (approx. 170 lines of code);
- Extended microC/OS-II scheduler with SRP’s preemption rule.
Assuming fixed-priority scheduling:

Immediate Priority Ceiling exhibits the same execution behavior as SRP

Instead of changing the scheduler:
- Lock: raise a task’s priority equal to the resource ceiling;
- Unlock: restore a task’s priority.
Alternative SRP implementation

Assuming fixed-priority scheduling:

Immediate Priority Ceiling exhibits the same execution behavior as SRP

Instead of changing the scheduler:
- **Lock**: raise a task’s priority equal to the resource ceiling;
- **Unlock**: restore a task’s priority.
Alternative SRP implementation

Assuming fixed-priority scheduling:
Immediate Priority Ceiling exhibits the same execution behavior as SRP

Instead of changing the scheduler:
- Lock: raise a task’s priority equal to the resource ceiling;
- Unlock: restore a task’s priority.

**Note 1:** Immediate Priority Ceiling \( \equiv \) Highest Locker Protocol!

**Note 2:** Add the priority inheriting task in the ready queue before the blocking task!

**Note 3:** SRP/IPC/HLP do not need waiting lists of blocked tasks!
Outline

1. Introduction to real-time systems
2. Task Synchronization
3. An alternative approach: Stack Resource Policy (SRP)
4. Concluding remarks
5. Hierarchical Scheduling Recapitulated
6. Inter-subsystem resource sharing
7. Conclusions
We presented (so far):

- an overview of task-synchronization methods;

- a classification of microC/OS-II’s priority-inversion protocol;

- an efficient task-level SRP implementation;
We presented (so far):

- an overview of task-synchronization methods;
- a classification of microC/OS-II’s priority-inversion protocol;
- an efficient task-level SRP implementation;

Keep in mind:

- Be careful with protocol specifications versus implementations;
- Keep durations of disabled interrupts short and predictable;
- Be aware of the consequences of blocking.
T. Baker.
Stack-based scheduling of realtime processes.

G. Buttazzo.
*Hard real-time computing systems - predictable scheduling algorithms and applications (2nd edition).*

L. Sha, R. Rajkumar, and J. Lehoczky.
Priority inheritance protocols: an approach to real-time synchronisation.
Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip.  

J. J. Labrosse.  
*Microc/OS-II*.  

M. M. H. P. van den Heuvel, R. J. Bril, J. J. Lukkien, and M. Behnam.  
Extending a HSF-enabled open-source real-time operating system with resource sharing.  
1. What are the disadvantages of SRP compared to PIP?

2. What are the (dis)advantages of both SRP implementation methods?
Why hierarchical scheduling: an example for automotive

- Reduce the number of nodes
- Trend: Fewer and more powerful nodes

**Courtesy by Dr. Nolte**
Integration Problem

1. Isolation: applications shall not *interfere*
   - Temporal isolation (processor);
   - Spatial isolation (memory).
Integration Problem

1. **Isolation**: applications shall not *interfere*
   - Temporal isolation (processor);
   - Spatial isolation (memory).

2. **Development and analysis versus integration**
   - Independent analysis of application on *virtual platforms*;
   - Application specific scheduling algorithms;
   - Composition of virtual platforms.
Integration Problem

1. Isolation: applications shall not *interfere*
   - Temporal isolation (processor);
   - Spatial isolation (memory).

2. Development and analysis versus integration
   - Independent analysis of application on *virtual platforms*;
   - Application specific scheduling algorithms;
   - Composition of virtual platforms.

3. Applications may share logical resources.
A Solution: Hierarchical Scheduling

- subsystem: server, set of tasks and local (task) scheduler
- server: a budget allocated each period
A Solution: Hierarchical Scheduling

- subsystem: server, set of tasks and local (task) scheduler
- server: a budget allocated each period

Tasks, located in arbitrary subsystems, may share logical resources
i.e. deal with local and global priority inversion!
Outline

1. Introduction to real-time systems
2. Task Synchronization
3. An alternative approach: Stack Resource Policy (SRP)
4. Concluding remarks
5. Hierarchical Scheduling Recapitulated
6. Inter-subsystem resource sharing
7. Conclusions
Global resource sharing problem

*Budget depletion* during a critical section can lead to excessive blocking times:

- **SRP locally**
- **SRP globally**

Legend:
- Active
- Holding mutex

Task1
Task2
Task3
OS-ServerIdle
Server1
Server2

Martijn van den Heuvel (TU/e, SAN) Resource Sharing in Real-time Systems 4th March 2011 31 / 39
Global resource sharing problem

Two SRP-based solutions for fixed-priority scheduling:

- **HSRP**: React upon budget depletion while a resource is locked; i.e. allow to use an overrun budget
  1. **with payback**: the consumed overrun budget is subtracted from the next budget provisioning;
  2. **no payback**: no penalty for overrun consumption.
Global resource sharing problem

Two SRP-based solutions for fixed-priority scheduling:

- **HSRP**: React upon budget depletion while a resource is locked; i.e. allow to use an overrun budget
  1. **with payback**: the consumed overrun budget is subtracted from the next budget provisioning;
  2. **no payback**: no penalty for overrun consumption.

- **SIRAP**: Prevent budget depletion during resource access; i.e. before granting access, first check the remaining budget.
HSRP provides overrun budget (optionally a payback mechanism):

Legend:
- active
- holding mutex
SIRAP uses a skipping mechanism:
## HSRP and SIRAP implementation overhead and issues

<table>
<thead>
<tr>
<th>Event</th>
<th>HSRP</th>
<th>SIRAP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock resource</td>
<td>-</td>
<td>spinlock</td>
</tr>
<tr>
<td>Unlock resource</td>
<td>overrun completion</td>
<td>-</td>
</tr>
<tr>
<td>Budget depletion</td>
<td>overrun</td>
<td>-</td>
</tr>
<tr>
<td>Budget replenishment</td>
<td>overrun completion, payback (optionally)</td>
<td>spinlock-completion</td>
</tr>
</tbody>
</table>
### HSRP and SIRAP implementation overhead and issues

<table>
<thead>
<tr>
<th>Event</th>
<th>HSRP</th>
<th>SIRAP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock resource</td>
<td>-</td>
<td>spinlock</td>
</tr>
<tr>
<td>Unlock resource</td>
<td>overrun completion</td>
<td>-</td>
</tr>
<tr>
<td>Budget depletion</td>
<td>overrun</td>
<td>-</td>
</tr>
<tr>
<td>Budget replenishment</td>
<td>overrun completion, payback (optionally)</td>
<td>spinlock-completion</td>
</tr>
</tbody>
</table>

- **HSRP:**
  - close to default SRP;
  - expensive queue manipulations to track overrun budget;
  - complex implementation due to explicit event handling.
## HSRP and SIRAP implementation overhead and issues

<table>
<thead>
<tr>
<th>Event</th>
<th>HSRP</th>
<th>SIRAP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock resource</td>
<td>-</td>
<td>spinlock</td>
</tr>
<tr>
<td>Unlock resource</td>
<td>overrun completion</td>
<td>-</td>
</tr>
<tr>
<td>Budget depletion</td>
<td>overrun</td>
<td>-</td>
</tr>
<tr>
<td>Budget replenishment</td>
<td>overrun completion, payback (optionally)</td>
<td>spinlock-completion</td>
</tr>
</tbody>
</table>

### HSRP:
- close to default SRP;
- expensive queue manipulations to track overrun budget;
- complex implementation due to explicit event handling.

### SIRAP:
- spinlocking is executed within a task’s context, but wastes budget;
- alternatively: suspend (i.e. block) and resume a task.
Outline

1. Introduction to real-time systems
2. Task Synchronization
3. An alternative approach: Stack Resource Policy (SRP)
4. Concluding remarks
5. Hierarchical Scheduling Recapitulated
6. Inter-subsystem resource sharing
7. Conclusions
Conclusions

We presented:

- a classification of microC/OS-II’s synchronization protocol;
- an efficient task-level SRP implementation;
- two alternative hierarchical SRP-implementations, i.e. SIRAP, HSRP;
- a side-by-side integration of these protocols in a single HSF;
- global protocol transparency from the application perspective.
Conclusions

We presented:

- a classification of microC/OS-II’s synchronization protocol;

- an efficient task-level SRP implementation;

- two alternative hierarchical SRP-implementations, i.e. SIRAP, HSRP;

- a side-by-side integration of these protocols in a single HSF;

- global protocol transparency from the application perspective.

To probe further:

- a resource-efficient protocol-selection criterium;

- wider range of budgeting/server models.

