Implementation and Evaluation of the Synchronization Protocol Immediate Priority Ceiling in PREEMPT-RT Linux

Andreu Carminati, Rômulo Silva de Oliveira, Luís Fernando Friedrich, Rodrigo Lange

Federal University of Santa Catarina (UFSC)
Florianópolis, Brazil
May/2010
Introduction

- In real-time operating systems, task synchronization mechanisms:
  - Provide internal consistency of resources and data structures
  - Provide determinism of waiting time
  - Avoid unbounded priority inversions

- In mainline Linux priority inversions occur frequently

- Patch PREEMP-RT uses Priority Inheritance
  - for priority inversion control
Objective

- The objective of this paper is the implementation of an alternative protocol, the Immediate Priority Ceiling (IPC)

- We intend to use it in dedicated, real-time device-drivers

- For example:
  - an embedded Linux supports an specific known application
  - that does not change task priorities after its initialization

- It is not the objective of this paper to propose a complete replacement of the existing protocol
  - but an alternative for use in some situations

- This work considers only single processors
--Unbounded Priority Inversion
--Priority Inheritance Protocol 1/2
Immediate Priority Ceiling

- It is a variation of Priority Ceiling Protocol
  - Highest Locker Priority

- Every resource has a priority ceiling
  - The highest priority of all task that access this resource

- The priority is adjusted immediately when occurs a resource acquisition
  - and not when the resource becomes necessary to a higher priority tasks

- Maximum blocking time of Ti under fixed priority:
  - The larger critical section of the system whose priority ceiling is higher than the priority of task Ti and is also used by a lower priority task
Immediate Priority Ceiling

- Ceiling of R1 = T0
- Ceiling of R2 = T1
Description of the Implementation

- The Immediate Priority Ceiling Protocol was implemented
- Based on the code of rt_mutexes already in the patch PREEMPT-RT
- The rt_mutexes implement the Priority Inheritance protocol
- Kernel version used was the 2.6.31.6 [10] with PREEMPT-RT patch rt19
  - rt_mutexes are implemented in PREEMPT-RT also for multiprocessors
  - Our implementation considers only single processors
Description of the Implementation

- Implementation was made for use in device-drivers (kernel space)
- For example, tasks share a critical section accessed through an ioctl() system call to a device-driver
Description of the Implementation: Data Structure

- The type that implements the Immediate Priority Ceiling protocol is defined as `struct ipc_mutex`

```c
struct ipc_mutex {
    atomic_spinlock_t wait_lock;
    struct plist_head wait_list;
    struct plist_node on_task_entry;
    struct task_struct *owner;
    int ceiling;
    ...
    // other rt_mutex fields
};
```
Description of the Implementation: Data Structure

- The type that implements the Immediate Priority Ceiling protocol is defined as `struct ipc_mutex`

- `wait_lock` is the spin-lock that protects the access to the structure
- `wait_list` is an ordered (by priorities) list that stores pending lock requests
- `on_task_entry` serves to manage the locks acquired by a task
  - and, consequently, control its effective priority
- `owner` stores a pointer to the task owner of the mutex
  - or null pointer if the mutex is available
- `ceiling` stores the priority ceiling of the mutex
Description of the Implementation: Operations

- **DEFINE_IPC_MUTEX( mutexname, priority)**

- Defines an Immediate Priority Ceiling mutex
- mutexname is the identifier of the mutex
- priority is the ceiling of the mutex

- The current version can only create mutexes with priorities set at compile time
- It is somewhat restrictive
- It is acceptable when an embedded Linux runs a known application that does not change task priorities after its initialization
Description of the Implementation: Operations

- `void ipc_mutex_lock( struct ipc_mutex * lock)`

- Lock operation
- In single-processor computers this is a non-blocking function
- If a task requests a resource, it is because this resource is available

- It manages the priority of the calling task along with the resource blocking, taking into account all `ipc_mutexes` acquired so far
Description of the Implementation: Operations

- `void ipc_mutex_unlock( struct ipc_mutex * lock)`

- Unlock operation
- Releases the resource and adjusts the priority of the calling task
Test Scenario

- We developed a device-driver that provides critical sections to perform the tests
- This device-driver exports a single service: unlocked_ioctl
- It multiplexes the calls of three tasks in their correspondent critical sections
- This device-driver provides critical sections to run with both
  - Immediate Priority Ceiling
  - Priority Inheritance
Test Scenario

- A set of sporadic tasks executes in user space
- All critical sections are executed within the device-driver
- Measurement of 1000 activations of T0
- Task set configuration:

<table>
<thead>
<tr>
<th>Task</th>
<th>T0/High</th>
<th>T1/Med.</th>
<th>T2/Low</th>
</tr>
</thead>
<tbody>
<tr>
<td>Priority</td>
<td>70</td>
<td>65</td>
<td>60</td>
</tr>
<tr>
<td>Activation interval</td>
<td>rand in [400,800] ms</td>
<td>rand in [95,190] ms</td>
<td>rand in [85,170] ms</td>
</tr>
<tr>
<td>Resource</td>
<td>R1</td>
<td>R1,R2</td>
<td>R2</td>
</tr>
<tr>
<td>Critical section size</td>
<td>aprox. 17 ms</td>
<td>aprox. 2x17 ms</td>
<td>aprox. 17 ms</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Task</th>
<th>T0/High</th>
<th>T1/Med.</th>
<th>T2/Low</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action 1</td>
<td>Lock(R1)</td>
<td>Lock(R1)</td>
<td>Lock(R2)</td>
</tr>
<tr>
<td>Action 2</td>
<td>Critical Sec.</td>
<td>Critical Sec.</td>
<td>Critical Sec.</td>
</tr>
<tr>
<td>Action 3</td>
<td>Unlock(R1)</td>
<td>Lock(R2)</td>
<td>Unlock(R2)</td>
</tr>
<tr>
<td>Action 4</td>
<td>Critical Sec.</td>
<td>Critical Sec.</td>
<td>Unlock(R2)</td>
</tr>
<tr>
<td>Action 5</td>
<td>Unlock(R2)</td>
<td>Unlock(R2)</td>
<td>Unlock(R1)</td>
</tr>
<tr>
<td>Action 6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Test Scenario

- All tasks were set at only one CPU
- Priority ceiling of mutex R1 is 70 (priority of task T0)
- Priority ceiling of mutex R2 is 65 (priority of task T1)

<table>
<thead>
<tr>
<th>Task</th>
<th>T0/High</th>
<th>T1/Med.</th>
<th>T2/Low</th>
</tr>
</thead>
<tbody>
<tr>
<td>Priority</td>
<td>70</td>
<td>65</td>
<td>60</td>
</tr>
<tr>
<td>Activation interval</td>
<td>rand in [400,800] ms</td>
<td>rand in [95,190] ms</td>
<td>rand in [85,170] ms</td>
</tr>
<tr>
<td>Resource</td>
<td>R1</td>
<td>R1,R2</td>
<td>R2</td>
</tr>
<tr>
<td>Critical section size</td>
<td>aprox. 17 ms</td>
<td>aprox. 2x17 ms</td>
<td>aprox. 17 ms</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Task</th>
<th>T0/High</th>
<th>T1/Med.</th>
<th>T2/Low</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action 1</td>
<td>Lock(R1)</td>
<td>Lock(R1)</td>
<td>Lock(R2)</td>
</tr>
<tr>
<td>Action 2</td>
<td>Critical Sec.</td>
<td>Critical Sec.</td>
<td>Critical Sec.</td>
</tr>
<tr>
<td>Action 3</td>
<td>Unlock(R1)</td>
<td>Lock(R2)</td>
<td>Unlock(R2)</td>
</tr>
<tr>
<td>Action 4</td>
<td>Critical Sec.</td>
<td>Critical Sec.</td>
<td></td>
</tr>
<tr>
<td>Action 5</td>
<td>Unlock(R2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Action 6</td>
<td>Unlock(R1)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Test Results: Activation Latency Histogram of T0

- Time between entering the ready queue and actually running
- Priority Inheritance

- Immediate Priority Ceiling
  - Task waits at the activation
Test Results: Locking Time of T0

- Time waiting at the lock operation
- Priority Inheritance
  - Task waits at the lock

- Immediate Priority Ceiling
Test Results: Response Time of T0

- Priority Inheritance

- Immediate Priority Ceiling
Test Results: Response Time of T0

- Priority Inheritance

- Immediate Priority Ceiling
Test Results: Response Time of T0

- Priority Inheritance

- Immediate Priority Ceiling
Test Results: Comments

- Immediate Priority Ceiling is in general similar to Priority Inheritance

- Immediate Priority Ceiling
  - Better average response time for T0
  - Worst-case response time is almost one critical section smaller
  - The length of a critical section is 17 ms, the difference between then is 14 ms

| Protocol: Average response time: Std dev: Max:  |
|---------------|---------------|---------------|---------------|
| PI            | 22,798,549 ns | 11,319,355 ns | 64,157,591 ns |
| IPC           | 21,014,311 ns | 8,723,159 ns  | 50,811,328 ns |
Test Results: Overhead

- Overhead = Decreasing of CPU time available to the rest of the system, given the presence of a set of higher priority tasks sharing resources protected
  - Test used a set of tasks that access the device-driver (mutexes)
  - Measuring task has lowest priority (priority 51)
    - Lower priority than tasks that access the mutexes inside the device-driver
    - Higher priority than threaded irq handlers and softirqs
  - All tasks fixed to a single CPU
Test Results: Overhead

- Any CPU time that is not used by tasks accessing the device-driver
  - is assigned to the measurement task
- The measurement task implements a loop
  - that just increments a variable

- The overhead will be noticed by how much the measurement task could increment a count
  - When Priority Inheritance is used
  - When Immediate Priority Ceiling is used
Test Results: Overhead

- The collected data indicate that there is a difference between the counts when the other tasks use
  - Mutexes with Priority Inheritance
  - Mutexes with Immediate Priority Ceiling
- The counting task (lowest priority) receives more processor time when Priority Inheritance mutexes are used
- The implementation of Priority Inheritance mutexes is more efficient
- Likely due to the fast path used
  - Our implementation of Immediate Priority Ceiling does not use a fast path

<table>
<thead>
<tr>
<th>Protocol</th>
<th>IPC</th>
<th>PI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average($\mu$)</td>
<td>189,136,685.72</td>
<td>189,682,847.91</td>
</tr>
<tr>
<td>Var($S^2_x$)</td>
<td>524,003,070,588.27</td>
<td>191,603,683,258.35</td>
</tr>
<tr>
<td>Minimum</td>
<td>187,882,717</td>
<td>188,776,389</td>
</tr>
<tr>
<td>Maximum</td>
<td>190,338,011</td>
<td>190,539,875</td>
</tr>
</tbody>
</table>
Summary

- We implemented the Immediate Priority Ceiling protocol in Linux with patch PREEMPT-RT
- This protocol is suitable for dedicated applications on single processors
- Offers smaller maximum response time for high priority tasks
  - Maximum blocking time is always at most one critical section
- Requires manual determination of the priority ceiling of each mutex
  - Feasible for small/static control applications
  - Dedicated device-drivers accessed by some well known tasks

- Overhead is higher than traditional Priority Inheritance
  - Fast path implementation is on the way
- It is possible to implement a version with adaptive ceiling
  - Priority ceiling is detected automatically in run-time for each mutex
Implementation and Evaluation of the Synchronization Protocol Immediate Priority Ceiling in PREEMPT-RT Linux

Andreu Carminati, Rômulo Silva de Oliveira, Luís Fernando Friedrich, Rodrigo Lange
Federal University of Santa Catarina (UFSC)
Florianópolis, Brazil

romulo@das.ufsc.br
andreu_carminati@yahoo.com.br