Fast on Average, Predictable in the Worst Case

*Exploring Real-Time Futexes in LITMUS\(^{RT}\)*

Roy Spliet, Manohar Vanga, Björn Brandenburg, Sven Dziadek
Real-Time Locking API

Mutex API

```c
kernel_do_lock();
// critical section
kernel_do_unlock();
```
Real-Time Locking API

Mutex API

```c
kernel_do_lock();
// critical section
kernel_do_unlock();
```

- No other task is currently holding the lock
- No other task is waiting for the lock

Operations are typically **uncontended** at runtime
Futexes: Fast Userspace Mutexes

A mechanism (API) in the Linux kernel

- For optimizing the uncontended case of locking protocol implementations
- Avoid kernel invocation during uncontended operations
Futexes: Fast Userspace Mutexes

A mechanism (API) in the Linux kernel

- For optimizing the uncontended case of locking protocol implementations
- Avoid kernel invocation during uncontended operations

Why optimize the uncontended case?

- Improved throughput for soft real-time workloads
- Increasing available slack time in the system
# Real-Time Locking Implementations

<table>
<thead>
<tr>
<th></th>
<th>PRIO_INHERIT (Priority Inheritance Protocol)</th>
<th>Futex Implementation</th>
</tr>
</thead>
</table>

- PRIO_INHERIT
- Immediate Priority Ceiling Protocol
- Classic Priority Ceiling Protocol (PCP)
- Multiprocessor PCP (MPCP)
- FMLP
- Futex Implementation

- Linux
# Real-Time Locking Implementations

<table>
<thead>
<tr>
<th>Futex Implementation</th>
<th>PRIO_INHERIT (Priority Inheritance Protocol)</th>
<th>PRIO_PROTECT (Immediate Priority Ceiling Protocol)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>![Green Check]</td>
<td>![Red X]</td>
</tr>
</tbody>
</table>

Linux
# Real-Time Locking Implementations

<table>
<thead>
<tr>
<th></th>
<th>PRIO_INHERIT (Priority Inheritance Protocol)</th>
<th>PRIO_PROTECT (Immediate Priority Ceiling Protocol)</th>
<th>Classic Priority Ceiling Protocol (PCP)</th>
<th>Multiprocessor PCP (MPCP)</th>
<th>FMLP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Futex Implementation</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

- **Linux**
- **LITMUS**

- **PRIO_INHERIT**: Priority Inheritance Protocol
- **PRIO_PROTECT**: Immediate Priority Ceiling Protocol
- **Classic Priority Ceiling Protocol (PCP)**
- **Multiprocessor PCP (MPCP)**
- **FMLP**
Real-Time Locking Implementations

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>✔</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
<td>❌</td>
</tr>
</tbody>
</table>

Choice between futex implementation and better analytical properties
# Real-Time Locking Implementations

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Linux</th>
<th>LITMUS\textsuperscript{RT}</th>
</tr>
</thead>
</table>

**Choice** between futex implementation and better analytical properties

**Why is it challenging to have both?**
# Real-Time Locking Implementations

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Reactive</td>
<td>![Green Checkmark] (✓)</td>
<td>![Red X] (✗)</td>
<td>![Red X] (✗)</td>
<td>![Red X] (✗)</td>
<td>![Red X] (✗)</td>
</tr>
</tbody>
</table>

Anticipatory
Real-Time Locking Protocol Dichotomy

- **Reactive Locking Protocols**
  React to contention on a lock only when it occurs.

- **Anticipatory Locking Protocols**
  Anticipate problem scenarios and take measure to minimize priority-inversion (pi) blocking.
Real-Time Locking Protocol Dichotomy

Reactive Locking Protocols

React to contention on a lock only when it occurs.

Simple futex implementation

Anticipatory Locking Protocols

Anticipate problem scenarios and take measure to minimize priority-inversion (pi) blocking.

Tricky to implement futexes without violating semantics
## Contributions

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>![Checkmark]</td>
<td>![X]</td>
<td>![X]</td>
<td>![X]</td>
<td>![X]</td>
</tr>
</tbody>
</table>

- **Contributions**:
  - PRIO_INHERIT (Priority Inheritance Protocol)
  - PRIO_PROTECT (Immediate Priority Ceiling Protocol)
  - Classic Priority Ceiling Protocol (PCP)
  - Multiprocessor PCP (MPCP)
  - FMLP
## Contributions

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Design and implementation of futexes for **three** anticipatory real-time locking protocols
Contributions

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>❌</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Observed overhead reduction: 92%  85%  84%

Design and implementation of futexes for three anticipatory real-time locking protocols
Contributions

<table>
<thead>
<tr>
<th></th>
<th>PRIO_INHERIT (Priority Inheritance Protocol)</th>
<th>PRIO_PROTECT (Immediate Priority Ceiling Protocol)</th>
<th>Classic Priority Ceiling Protocol (PCP)</th>
<th>Multiprocessor PCP (MPCP)</th>
<th>FMLP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Futex Implementation</td>
<td>![Yes]</td>
<td>![No]</td>
<td>![Yes]</td>
<td>![Yes]</td>
<td>![Yes]</td>
</tr>
</tbody>
</table>

Observed overhead reduction:

- 92%
- 85%
- 84%

Design and implementation of futexes for three anticipatory real-time locking protocols

Method of using page faults to implement futexes
Overview of Talk
Overview of Talk

Challenge → Implementation → Evaluation
Overview of Talk

Challenge

Implementation

Evaluation
Futexes — Overview and Intuition

*The problem with the vanilla approach*

```c
kernel_do_lock();
// critical section
kernel_do_unlock();
```
Futexes — Overview and Intuition

The problem with the vanilla approach

Kernel invoked on every lock and unlock operation

Mutex API

```c
kernel_do_lock();
// critical section
kernel_do_unlock();
```
Futexes — Overview and Intuition

The problem with the vanilla approach

Kernel invoked on every lock and unlock operation

Mutex API

kernel_do_lock();
// critical section
kernel_do_unlock();
Futexes — Overview and Intuition

Exporting Lock State to Userspace Processes

Shared lock state between userspace and kernel
Futexes — Overview and Intuition

Exporting Lock State to Userspace Processes

Futex Pseudocode

```c
try_to_acquire_lock(lock);
if (lock is contended)
    kernel_do_lock(lock);
// critical section
release_lock(lock);
if (there are blocked tasks)
    kernel_do_unlock();
```
**Futexes — Overview and Intuition**

*Fast-Path for Uncontended Operations*

---

**Futex Pseudocode**

```c
try_to_acquire_lock(lock);
if (lock is contended)
    kernel_do_lock(lock);
// critical section
release_lock(lock);
if (there are blocked tasks)
    kernel_do_unlock();
```

---

**Fast-path** when operation is uncontended (kernel not invoked)
Futuxes — Overview and Intuition

Slow-Path for Contended Operations

Futex Pseudocode

```c
try_to_acquire_lock(lock);
if (lock is contended)
    kernel_do_lock(lock);
// critical section
release_lock(lock);
if (there are blocked tasks)
    kernel_do_unlock();
```

Kernel invoked only when lock or unlock operation is contended
Priority Inheritance Protocol

*Futexes for reactive locking protocols*

Fixed-Priority Scheduling
Priority Inheritance Protocol

Futexes for reactive locking protocols

Uncontended
No kernel intervention

Lock 1

Lock 2

Lock 2

...
Priority Inheritance Protocol

*Futexes for reactive locking protocols*

*Uncontended*
No kernel intervention

*Contended*
Kernel intervention needed

Fixed-Priority Scheduling
Priority Ceiling Protocol (PCP)

- Each lock has a **priority ceiling** — the highest priority of all tasks that can acquire that lock.
Priority Ceiling Protocol (PCP)

- Each lock has a priority ceiling — the highest priority of all tasks that can acquire that lock.

- System ceiling — Currently held lock with highest ceiling
Priority Ceiling Protocol (PCP)

- Each lock has a **priority ceiling** — the highest priority of all tasks that can acquire that lock.

- **System ceiling** — Currently held lock with highest ceiling

- A lock may be acquired by a task only if:
  - Task priority > System Ceiling Priority
  - Or the task is **holding the system ceiling**
Priority Ceiling Protocol (PCP)

• Each lock has a **priority ceiling** — the highest priority of all tasks that can acquire that lock.

• **System ceiling** — Currently held lock with highest ceiling

• A lock may be acquired by a task only if:
  
  • Task priority > System Ceiling Priority
  
  • Or the task is holding the system ceiling

• When **under contention, priority inheritance** (raises priority of lower priority task to that of higher priority task).
PCP Futexes — Challenge

- **HI**: Raises system ceiling prio. to **HI**
- **MED**: L1
- **LO**: L1
- **L2**: Lowers system ceiling prio. to **None**
PCP Futexes — Challenge

Valid acquisition
MED > System Ceiling Prio.

Time

L1

Valid acquisition
MED > System Ceiling Prio.

L1

L2

L1

LO

HI

MED

L1

L2

L1

L1

HI

LO

L1

L1

L2

Time

raises system ceiling prio. to HI

lowers system ceiling prio. to None

Fixed-Priority Scheduling
PCP Futexes — Challenge

Fixed-Priority Scheduling
PCP Futexes — Challenge

Time

HI

LO

MED

L1

L2

L1

L2

HI

LO

MED

Time

Preemption

Sys. ceiling prio. = HI

Raises system ceiling prio. to HI

Fixed-Priority Scheduling
PCP Futexes — Challenge

- L2 free
- But invalid acquisition

- Sys. Ceiling Prio. > MED

- Raises system ceiling prio. to HI

- Preemption

- Sys. ceiling prio. = HI

Fixed-Priority Scheduling
PCP Futexes — Challenge

Problem — Simply relying on lock state (as in reactive case) may violate PCP semantics. Must consider system ceiling.

 Raises system ceiling prio. to HI

Preemption
Sys. ceiling prio. = HI

L2 free
But invalid acquisition
Sys. Ceiling Prio. > MED

Fixed-Priority Scheduling
PCP Futexes — Challenge

Problem — Simply relying on lock state (as in reactive case) may violate PCP semantics. Must **consider system ceiling**.

Problem — system ceiling potentially updated at every lock acquisition and release — must **always invoke kernel** to avoid protocol violations.

Fixed-Priority Scheduling
Overview of Talk

Challenge

Implementation

Evaluation
Overview of Talk

Challenge

Implementation

Evaluation
PCP Futexes — Our Approach

PCP Futex Pseudocode

```c
if (lock is contended)
    kernel_do_lock(lock);
else
    acquire_lock_locally(lock)
// critical section
release_lock(lock);
if (there are blocked tasks)
    kernel_do_unlock();
```
PCP Futexes — Our Approach

Whether lock acquisition is contended is determined in PCP by the current system ceiling

PCP Futex Pseudocode

```
if (lock is contended)
    kernel_do_lock(lock);
else
    acquire_lock_locally(lock)
// critical section
release_lock(lock);
if (there are blocked tasks)
    kernel_do_unlock();
```
PCP Futexes — Our Approach

PCP Futex Pseudocode

```c
if (lock is contended)
    kernel_do_lock(lock);
else
    acquire_lock_locally(lock)
// critical section
release_lock(lock);
if (there are blocked tasks)
    kernel_do_unlock();
```

Whether lock acquisition is contended is determined in PCP by the `current system ceiling`

The presence of blocked tasks is determined by the `size of the wait-queue` corresponding to the lock.
PCP Futexes — Our Approach

Whether lock acquisition is contended is determined in PCP by the current system ceiling.

The presence of blocked tasks is determined by the size of the wait-queue corresponding to the lock.

PCP Futex Pseudocode

```plaintext
if (lock is contended)
    kernel_do_lock(lock);
else
    acquire_lock_locally(lock)
// critical section
release_lock(lock);
if (there are blocked tasks)
    kernel_do_unlock();
```

Cannot change **while a task is scheduled** on a uniprocessor!
PCP Futexes — Deferred Updates

Scheduling Interval

Invoke kernel
PCP Futexes — Deferred Updates

Predetermine contention conditions and communicate them to task

Make changes to local copy of lock states (if uncontended)

Update kernel state based on copy at context switch
PCP Futexes — Deferred Updates

Predetermine contention conditions and communicate them to task

Make changes to local copy of lock states (if uncontended)

Update kernel state based on copy at context switch

Requires a bidirectional communication channel between tasks and the kernel
Bidirectional Communication — Lock Page

Lock Page

Per-process **page of memory** writable by both the process and kernel.

<table>
<thead>
<tr>
<th>bitmap: lock_states</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool: tasks_blocked</td>
</tr>
</tbody>
</table>
Bidirectional Communication — Lock Page

Lock Page

Per-process page of memory writable by both the process and kernel.

Lock-state bitmap used to communicate acquisitions and releases of locks to kernel

- **bitmap**: lock_states
- **bool**: tasks_blocked
Bidirectional Communication — Lock Page

Lock Page

Per-process page of memory writable by both the process and kernel.

- **bitmap**: lock_states
- **bool**: tasks_blocked

**Lock-state bitmap** used to communicate acquisitions and releases of locks to kernel.

**Boolean** set by kernel to indicate the presence of blocked tasks (indicates that release operation is contended).
Bidirectional Communication — Lock Page

**Lock Page**

Per-process page of memory writable by both the process and kernel.

- **bitmap**: lock_states
- **bool**: tasks_blocked

**Lock-state bitmap** used to communicate acquisitions and releases of locks to kernel.

Boolean set by kernel to indicate the presence of blocked tasks (indicates that release operation is contended).

A “permission bit” set by the kernel to indicate whether lock acquisition is allowed (system ceiling check).
Communicating the Permission Bit

PCP Futex Pseudocode

```c
if (permission bit is set)
    set_bit_in_bitmap(lock)
else
    kernel_do_lock(lock);
// critical section
release_lock(lock);
if (tasks_blocked is set)
    kernel_do_unlock();
```
Communicating the Permission Bit

PCP Futex Pseudocode

```c
if (permission bit is set) set_bit_in_bitmap(lock)
else
    kernel_do_lock(lock);
// critical section
release_lock(lock);
if (tasks_blocked is set) kernel_do_unlock();
```

Challenge:

Checking for permission and setting lock bit **must be atomic**.

Why? Preemption between them may change the permission.
We proposed two approaches for the Classic PCP

**Page Faults (PCP-DU-PF)**

Uses *page faults* to invoke kernel when locking is prohibited

**CMPXCHG (PCP-DU-BOOL)**

*Boolean* for permission bit.

*Compare-and-exchange* operation to check and acquire locks atomically
Communicating the Permission Bit

We proposed two approaches for the Classic PCP:

- **Page Faults (PCP-DU-PF)**
  Uses *page faults* to invoke kernel when locking is prohibited

- **CMPXCHG (PCP-DU-BOOL)**
  Compare-and-exchange operation to check and acquire locks atomically

See paper for details!
PCP Futexes — Page Fault Approach

If (locking is allowed)
set lock page writable
else
set lock page unwritable

Attempting to write lock bit will invoke kernel automatically if locking is not permitted.
PCP Futexes — Page Fault Approach

Can be implemented using segmentation faults as well

Lock code-path is entirely branch-free

If (locking is allowed)
set lock page writable
else
set lock page unwritable

Attempting to write lock bit will invoke kernel automatically if locking is not permitted
PCP Futexes — Page Fault Approach

Can be implemented using **segmentation faults** as well

Lock code-path is **entirely branch-free**

Requires an **optimized page-fault handler**

If (locking is allowed)

- set lock page writable
else

- set lock page unwritable

Attempting to write lock bit will **invoke kernel automatically** if locking is not permitted
Communicating the Permission Bit

We proposed two approaches for the Classic PCP:

Page Faults (PCP-DU-PF):

Uses page faults to invoke kernel when locking is prohibited.

Lock code-path is entirely branch-free.

CMPXCHG (PCP-DU-BOOL):

Boolean for permission bit.

Compare-and-exchange operation to check and acquire locks atomically.

Avoid page faults at cost of atomic op + branch.
Overview of Talk

Challenge

Implementation

Evaluation
Overview of Talk

- Challenge
- Implementation
- Evaluation
Evaluation — Platform

- Boundary Devices Sabre Lite Board
  - Freescale i.MX6Q Quad Core SoC
  - ARM Cortex A9 (1 GHz)
- Implemented in LITMUS$^{RT}$ 2013.1 (based on Linux 3.10.5)
Evaluation — Benchmarking Methodology

- Test program **locks and unlocks once every period**
  - Uniprocessor tests: 5 threads
- **Randomly assigned parameters** for threads
  - *Critical section length* — 25-45μs
  - *Execution time* — 25-65μs
  - *Period* — 600-800μs
- Measured using processor **timestamp counters**
Evaluation — Benchmarking Methodology

- Test program **locks and unlocks once every period**
  - Uniprocessor tests: 5 threads
- **Randomly assigned parameters** for threads
  - Critical section length — 25-45µs
  - Execution time — 25-65µs
  - Period — 600-800µs
- Measured using processor **timestamp counters**

**Demanding workload** — thousands of operations per second
PCP Futex Evaluation — Baseline

We compare all futex implementations against the non-futex PCP implementation in LITMUS\textsuperscript{RT}.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LITMUS\textsuperscript{RT}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
PCP Futex Evaluation — Baseline

- PRIO_PROTECT (Linux PCP)
- PCP (LITMUS^RT)

**Overhead in Cycles**

- **Lock (avg)**: 1,604 (PRIO_PROTECT), 645 (PCP)
- **Lock (max)**: 3,850 (PRIO_PROTECT), 1,221 (PRIO_PROTECT), 1,211 (PCP)
- **Unlock (avg)**: 9,027 (PRIO_PROTECT), 6,655 (PCP)
- **Unlock (max)**: 9,027 (PRIO_PROTECT), 6,655 (PCP)
Our baseline PCP implementation already incurs lower overheads than Linux’s.
How much reduction do we see in average uncontended-case overheads?
PCP Futex Evaluation — Uncontended Case

Based on 6.6 million samples per protocol
PCP Futex Evaluation — Uncontended Case

Our PCP futex implementation overheads are significantly lower than Linux’s PIP futex implementation.

Based on 6.6 million samples per protocol
PCP Futex Evaluation —Contended Case

How much additional overhead do we incur in the maximum contended-case overheads?
PCP Futex Evaluation — Contended Case

Based on 900K samples per protocol
PCP Futex Evaluation — Contended Case

Linux page-fault handler not optimized for real-time locking protocol implementation.

Based on 900K samples per protocol
PCP Futex Evaluation — Contended Case

Based on 900K samples per protocol
PCP Futex Evaluation — Contended Case

Up to \textbf{37\% increase} in worst-case contended overheads as a result of the futex approach.

Based on 900K samples per protocol
PCP Futex Evaluation — Contended Case

Up to 37% increase in worst-case contended overheads as a result of the futex approach.

But no worse than Linux’s futex implementation

Based on 900K samples per protocol
PCP Futex Evaluation — Summary

Up to **92% reduction in average uncontended overheads** at a cost of 37% increase in maximum contended overheads (**no worse than Linux**).

**Page faults** can be used to implement futexes for anticipatory real-time locking protocols.
## Summary

### Design and implementation of futexes for three anticipatory real-time locking protocols

<table>
<thead>
<tr>
<th>Method of using page faults to implement futexes</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

### Observed overhead reduction:

- 92%
- 85%
- 84%
Thanks!

Linux Testbed for Multiprocessor Scheduling in Real-Time Systems

All source code available at: http://www.litmus-rt.org/