Deterministic Futexes Revisited

Alexander Zuepke, Robert Kaiser

first.last@hs-rm.de
Futexes

- **Futexes**: underlying mechanism for thread synchronization in Linux

- libc provides:
  - Mutexes and Condition Variables
  - Semaphores, Reader-Writer Locks, Barriers, ...

- Linux kernel provides system calls to:
  - suspend the calling thread
  - wake a given number of waiting threads

- First prototype in Linux kernel version 2.5.7
Futexes

• Linux Futex API

```c
#include <linux/futex.h>

int SYS_futex(int *uaddr, int op, int val,
               const struct timespec *timeout, int *uaddr2, int val3);
```

• Operations

- **FUTEX_WAIT**  Suspend calling thread on futex uaddr
- **FUTEX_WAKE**  Wake val threads waiting on futex uaddr
- **FUTEX_REQUEUE**  Move threads waiting on uaddr to uaddr2
- … more operations available → see FUTEX(2) man page
Mutex Example

- **mutex_lock / mutex_unlock**
  - Fast path: use atomic operations to change a 32-bit integer variable in user space
  - No system call involved!

- **mutex_lock on contention**
  - Atomically indicate pending waiters
    - **futex_wait** system call
      - Look-up wait queue
      - Check futex value again
      - Enqueue calling thread in wait queue
      - Suspend calling thread

- **mutex_unlock on contention**
  - **futex_wake** system call
    - Look-up wait queue
    - Wake first waiting thread
Condition Variable Example

- `cond_signal/broadcast`

  - Atomically increment futex value
  - Call `futex_requeue` to move one/all waiters from condition variable wait queue to mutex wait queue
Futexes

- Futex ↔ generic *compare-and-block* mechanism
- Implement POSIX synchronization mechanisms in user space

- Can we use it in safety critical systems?
- Certification?
- WCET?
- Interference analysis?
Futexes

- Futex ↔ generic *compare-and-block* mechanism
- Implement POSIX synchronization mechanisms in user space

- Can we use it in safety critical systems?
- Certification?
- WCET?
- Interference analysis?
Futexes

• Futex ↔ generic *compare-and-block* mechanism
• Implement POSIX synchronization mechanisms in user space

- But:
  - Can we use futexes in real-time systems?
  - WCET?
  - Interference?
  - Determinism?
Outline

- Linux implementation
- Our OSPERT 2013 approach
- Requirements for determinism
- Our new approach
- Discussion
Linux Implementation
Linux Implementation

- Hash of *shared* wait queues
  - `num_cpus` x 256 hash buckets
  - all operations in O(1) time

- Wait queues
  - Priority-sorted linked list
    - O(n) find
    - O(p) insertion
    - O(1) removal

- Locking: per hash bucket
Linux Implementation

• Hash of *shared* wait queues
  – \text{num\_cpus} \times 256 \text{ hash buckets}
  – all operations in \text{O}(1) time

• Wait queues
  – Priority-sorted linked list
    • \text{O}(n) find
    • \text{O}(p) insertion
    • \text{O}(1) removal

• Locking: per hash bucket
Linux Implementation

- **Hash of shared wait queues**
  - `num_cpus x 256` hash buckets
  - all operations in O(1) time

- **Wait queues**
  - Priority-sorted linked list
    - O(n) find
    - O(p) insertion
    - O(1) removal

- **Locking:** per hash bucket
Linux Implementation

• Hash of *shared* wait queues
  – `num_cpus x 256` hash buckets
  – all operations in O(1) time

• Wait queues
  – Priority-sorted linked list
    • O(n) find
    • O(p) insertion
    • O(1) removal

• Locking: per hash bucket
Linux Implementation

- Hash of *shared* wait queues
  - `num_cpus` x 256 hash buckets
  - all operations in O(1) time

- Wait queues
  - Priority-sorted linked list
    - O(n) find
    - O(p) insertion
    - O(1) removal

- Locking: per hash bucket
Linux Implementation

- Hash of *shared* wait queues
  - num_cpus x 256 hash buckets
  - all operations in O(1) time
- Wait queues
  - Priority sorted linked list
    - O(n) find
    - O(p) insertion
    - O(1) removal
- Locking: per hash bucket
  - Hash Collision!
Linux Implementation

• Experiment
  - Two processes
  - 2048 blocked threads each
  - Process $\alpha$ requeues 2048 threads from $\alpha_{\text{src}}$ to $\alpha_{\text{dst}}$
  - Process $\beta$ requeues 1 thread from $\beta_{\text{src}}$ to $\beta_{\text{dst}}$
  - Measure $\beta$’s execution time
  - Note: four wait queues!
#1: $\beta$ requeues to distinct futex wait queues

$\alpha$ not involved

#1 distinct wait queues

$\beta_{\text{src}} \neq \beta_{\text{dst}}, \alpha \neq \beta$

operations in constant time

Core 2 Duo @ 3.00 GHz, Linux 4.6.12-rt5, Ubuntu 16.04
#2: β requeues to same futex wait queue

α not involved

#2 shared wait queue for β

$\beta_{src} = \beta_{dst}$, $\alpha \neq \beta$...

linear search in plist, threads remain in place

#1 distinct wait queues

$\beta_{src} \neq \beta_{dst}$, $\alpha \neq \beta$...

operations in constant time
#3: β requeues to same futex wait queue
α also requeues 2048 threads
Linux Implementation

- Drawbacks of Linux implementation
  - Shared wait queues
  - Dynamic memory allocations for PI mutexes
  - Not preemptive
Our OSPERT 2013 Approach
OSPERT 2013 Approach*

- Save thread ID of first waiter next to the futex in user space
  \[\rightarrow O(1) \text{ look-up of wait queue}\]
- FIFO ordering in wait queue
  \[\rightarrow \text{wait queues use linked lists}\]
- `futex_requeue` appends whole linked lists
  \[\rightarrow \text{in } O(1) \text{ time}\]
- other operations: also \(O(1)\) time

---

* A. Zuepke, “Deterministic Fast User Space Synchronization”, OSPERT 2013
Limitations of paper version:
- Limited set of futex operations
- No “wake all” operation (for barriers)
- Only FIFO ordering

Overcoming these limitations is possible:
- Priority ordering of wait queue
- Preemptive “wake_all” operation

* A. Zuepke, “Deterministic Fast User Space Synchronization”, OSPERT 2013
• Particular Problems:
  – Consistency of linked list during deletion/timeout handling while another thread walks this list
  – “Sneak-in”: prevent woken threads from re-entering a wait queue
  – Scalability: requires a global lock design

→ Result: complex implementation
→ Take a fresh look ...

* A. Zuepke, “Deterministic Fast User Space Synchronization”, OSPERT 2013
Requirements for Determinism

Use dedicated wait queues!
Requirements

0. Do not share wait queues
1. No dynamic memory allocations
2. Priority ordered wait queues; FIFO order on tie
3. Wait queue: use binary search tree (BST)
4. Wait queue look-up: use BST as well
5. Preemptible “wake/preempt all” operations
6. Prevent “sneak-in”
7. Transparent preemption
8. Fine granular locking
Requirements

0. Do not share wait queues
1. No dynamic memory allocations
2. Priority ordered wait queues; FIFO order on tie
3. Wait queue: use binary search tree (BST)
4. Wait queue look-up: use BST as well
5. Preemptible “wake/preempt all” operations
6. Prevent “sneak-in”
7. Transparent preemption
8. Fine granular locking
Requirements

0. Do not share wait queues

1. **No dynamic memory allocations**

2. Priority ordered wait queues; FIFO order on tie

3. Wait queue: use binary search tree (BST)

4. Wait queue look-up: use BST as well

5. Preemptible “wake/preempt all” operations

6. Prevent “sneak-in”

7. Transparent preemption

8. Fine granular locking

No interference

No unexpected failures
Requirements

0. Do not share wait queues
1. No dynamic memory allocations
2. Priority ordered wait queues; FIFO order on tie
3. Wait queue: use binary search tree (BST)
4. Wait queue look-up: use BST as well
5. Preemptible “wake/preempt all” operations
6. Prevent “sneak-in”
7. Transparent preemption
8. Fine granular locking
Requirements

0. Do not share wait queues
1. No dynamic memory allocations
2. Priority ordered wait queues; FIFO order on tie
3. Wait queue: use binary search tree (BST)
4. Wait queue look-up: use BST as well
5. Preemptible “wake/preempt all” operations
6. Prevent “sneak-in”
7. Transparent preemption
8. Fine granular locking
0. Do not share wait queues
1. No dynamic memory allocations
2. Priority ordered wait queues; FIFO order on tie
3. Wait queue: use binary search tree (BST)
4. Wait queue look-up: use BST as well
5. Preemptible “wake/preempt all” operations
6. Prevent “sneak-in”
7. Transparent preemption
8. Fine granular locking
New Approach*

(* this paper)
New Approach

- Use *two* nested BST
- Keep all data in TCB
- Create wait queues on demand
- *Address tree* for wait queue look-up
- *Wait queue* keeps blocked threads
New Approach

- Wait queue changes require care
- Example: timeout of \textit{a}
  - \textit{a} is wait queue root
  - \textit{c} becomes new root
- Copy WQ information
- Swap \textit{a} and \textit{c} in address tree
- Remove \textit{a}
New Approach

- Wait queue changes require care
- Example: timeout of $a$
  - $a$ is wait queue root
  - $c$ becomes new root
  - Copy WQ information
  - Swap $a$ and $c$ in address tree
  - Remove $a$
New Approach

- Wait queue changes require care
- Example: timeout of a
  - a is wait queue root
  - c becomes new root
- Copy WQ information
  - Swap a and c in address tree
  - Remove a
New Approach

- Wait queue changes require care
- Example: timeout of a
  - a is wait queue root
  - c becomes new root
  - Copy WQ information
  - Swap a and c in address tree
  - Remove a
New Approach

- Wait queue changes require care
- Example: timeout of a
  - a is wait queue root
  - c becomes new root
  - Copy WQ information
  - Swap a and c in address tree
  - Remove a
New Approach

• Preemptible Operations
  – For `futex_wake(all)` and `futex_requeue(all)`
  – Prevent re-insertion in wait queues (“sneak in”)
  – Use lowest bit of wait queue’s futex address
    • Open: allow adding threads
    • Closed: only allow wake-up/requeueing
  – `futex_*`(all) operations close wait queues
    • Clear open bit → order in address tree is preserved
  
  → but now multiple closed wait queues may exist
New Approach

• Preemptible Operations
  - Problem: multiple wait queues in *closed* state
New Approach

- Preemptible Operations
  - Problem: multiple wait queues in closed state
  - Global 64-bit drain counter
  - On close: drain counter++, draw a drain ticket
New Approach

- Preemptible Operations
  - Problem: multiple wait queues in *closed* state
  - Global 64-bit *drain counter*
  - On close: drain counter++, draw a *drain ticket*

Loop:
- look up closed wait queues
- Stop if wait queue’s drain ticket > drawn drain ticket
- perform wake up/requeue operation on one thread
- next preemption point ...
New Approach

- Preemptible Operations
  - Problem: multiple wait queues in closed state
  - But: It is OK to drain older wait queues?
  - Requeue and wake-up all operations are not atomic!
- **Preemptible Operations**
  - Problem: multiple wait queues in *closed* state
  - But: It is OK to drain older wait queues?
  - Requeue and wake-up all operations are not atomic!

- **Condition Variables**
  - POSIX: caller of `cond_broadcast()` should have the support mutex locked → uses requeue internally
New Approach

• Preemptible Operations
  – Problem: multiple wait queues in closed state
  – But: It is OK to drain older wait queues?
  – Requeue and wake-up all operations are not atomic!

• Condition Variables
  – POSIX: caller of cond_broadcast() should have the support mutex locked → uses requeue internally

• Barriers
  – POSIX does not guarantee any scheduling order
New Approach

- Fine Granular Locking
  - Idea: nested locks
  - Example: Look-up a wait queue
    - Lock address tree
    - Locate wait queue & lock wait queue
    - Unlock address tree
New Approach

- **Fine Granular Locking**
  - Idea: nested locks
  - Example: Look-up a wait queue
    - Lock address tree
    - Locate wait queue & lock wait queue
    - Unlock address tree
  - Problem:
    - Removal of empty wait queues
    - Frequent changes of wait queue anchor threads
New Approach

• Fine Granular Locking
  – Idea: nested locks
  – Example: Look-up a wait queue
    • Lock address tree
    • Locate wait queue & lock wait queue
    • Unlock address tree
  – Problem:
    • Removal of empty wait queues
    • Frequent changes of wait queue anchor threads

Not solved → use a single lock
New Approach

• Summarized
  – Dedicated wait queues per futex
  – No dynamic memory allocations
  – O(log n) look-up/insert/remove of wait queues
  – O(log n) handling inside wait queues
  – Preemptible operations of \textit{wake all} and \textit{requeue all}
    • Maximum of $n-1$ threads in \textit{all} operations
Discussion
Discussion

- Our new approach compared to Linux
  - Missing use cases
    - No “wake arbitrary number of threads”
      - but not needed for POSIX synchronization mechanisms
    - No priority inheritance protocol for mutexes
    - No FUTEX_WAKE_OP
    - No FUTEX_WAIT/WAKE_BITSET
  - Different API
    - Caller of futex_wait must provide requeue target futex
  - But
    - Priority ceiling protocol possible (unrelated to futex API)
## Discussion

<table>
<thead>
<tr>
<th>Our new approach</th>
<th>Our old approach</th>
<th>Linux</th>
</tr>
</thead>
<tbody>
<tr>
<td>Futexes share wait queues</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Wait queue look-up</td>
<td>BST $O(\log m)$</td>
<td>via TID $O(1)$</td>
</tr>
<tr>
<td>Wait queue implementation</td>
<td>priority-sorted BST $O(\log n)$</td>
<td>FIFO-ordered linked list $O(1)$</td>
</tr>
<tr>
<td>- find</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- insertion</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- removal</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Locking</td>
<td>global</td>
<td>global</td>
</tr>
<tr>
<td><code>futex_requeue</code></td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>- one thread</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>- arbitrary number of threads</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>- all threads</td>
<td>yes</td>
<td>not needed</td>
</tr>
<tr>
<td>- preemptive implementation</td>
<td>yes</td>
<td>not needed</td>
</tr>
<tr>
<td><code>futex_wake</code></td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>- one thread</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>- arbitrary number of threads</td>
<td>yes</td>
<td>not provided</td>
</tr>
<tr>
<td>- all threads</td>
<td>yes</td>
<td>not needed</td>
</tr>
<tr>
<td>- preemptive implementation</td>
<td>yes</td>
<td>not needed</td>
</tr>
<tr>
<td>Priority ceiling protocol</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Priority inheritance protocol</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

for $n$ threads, $m$ futexes, and $p$ priority levels
Thank you for your attention!

Questions?
Backup Slides
Priority Inheritance Mutexes

- Priority Inheritance Mutexes
  - POSIX: PTHREAD_PRIO_INHERIT
  - On contention, blocked threads boost the scheduling priority of the current lock holder
  - Implemented in Linux via FUTEX_LOCK_PI API

- Problems
  - Nested locks: applied recursively ...
  - Potentially unbounded recursion!
  - Cycles in dependency graph lead to deadlocks
Priority Ceiling Mutexes

- Priority Ceiling Mutexes
  - POSIX: PTHREAD_PRIO_PROTECT
  - Each mutex has an assigned ceiling priority
  - Before locking: increase scheduling priority to ceiling priority
  - After locking: restore previous scheduling priority
  - Implemented independently of futex API

- Can be implemented without system calls
  → Fast User Space Priority Switching, OSPERT 2014
The End