# Proxy Execution

Peter Zijlstra
Intel OTC

# Why

- Scheduler dependent inheritance schemes:
  - Static priority; FIFO/RR
    - Priority inheritance
  - Job priority; DEADLINE
    - Deadline inheritance; insufficient
    - Bandwidth inheritance; (missing, but doable)
  - Weighted Fair Queueing; NORMAL (CFS)
    - People are 'inheriting' nice values → wrong!

# Proxy Execution

- Integrates the 'priority' protocol with the scheduling function
- Splits 'task' into a scheduling context and execution context
- Keeps 'blocked' (on mutexes) tasks on the runqueue

# PE #1

```
rq->proxy = next = pick_next_task(rq, rq->proxy)
if (unlikely(next->blocked_on))
    next = proxy(rq, next);
```

```
static struct task_struct *
proxy(struct rq *rq, struct task_struct *next)
{
        struct task_struct *p, *owner;
        struct mutex *mutex;

        for (p = next; p->blocked_on; p = owner) {
                mutex = p->blocked_on;
                owner = mutex->owner;
        }

        return owner;
}
```

# Mutex #1

```
Task-A                Task-B

mutex_lock(m)
  if (xchg(&m->lock, 1))

                      mutex_lock()
                        if (xchg(&m->lock, 1))
                         mutex_lock_slow();
                           for (;;) {
                              set_current_state(TASK_UNINTERRUPTIBLE);
                              if (mutex_trylock(m))
                                break;
                              schedule();
                                *BOOM*
                           }

    mutex_lock_slow(m);
```

- Fast path with atomic owner
- No optimistic spinning
- Must set: `current->blocked_on = mutex`

# PE #2

```
static struct task_struct *
proxy(struct rq *rq, struct task_struct *next)
{
    struct task_struct *p, *owner;
    struct mutex *mutex;

    for (p = next; p->blocked_on; p = owner) {
        mutex = p->blocked_on;
        owner = __mutex_owner(mutex);
          /* atomic_long_read(&mutex->lock) & ~MUTEX_FLAGS */
    }

    return owner;
}
```

- What if owner is blocked on !mutex..?

# PE #3

```
static struct task_struct *
proxy(struct rq *rq, struct task_struct *next)
{
    struct task_struct *p, *owner;
    struct mutex *mutex;

    for (p = next; p->blocked_on; p = owner) {
        mutex = p->blocked_on;
        owner = __mutex_owner(mutex);

        if (!owner->on_rq)
            goto blocked;

        owner->blocked_task = p;
    }

    return owner;

blocked:
    for (; p; p = p->blocked_task) {
        p->on_rq = 0;
        deactivate_task(rq, p, DEQUEUE_SLEEP);
        list_add(&p->blocked_entry, &owner->blocked_entry);
    }
    return NULL;
}
```

# PE #3 cont

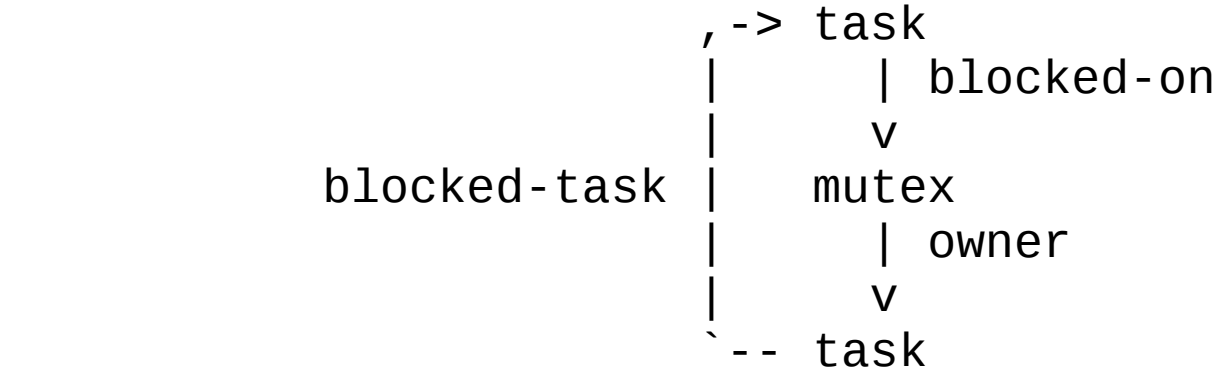
```
again:
    rq->proxy = next = pick_next_task(rq, rq->proxy)
    next->blocked_task = NULL;
    if (unlikely(next->blocked_on)) {
        next = proxy(rq, next);
        if (!next)
            goto again;
    }
```

```
static inline void
ttwu_activate(struct rq *rq, struct task_struct *p, int en_flags)
{
        activate_task(rq, p, en_flags);
        p->on_rq = TASK_ON_RQ_QUEUED;

        while (!list_empty(&p->blocked_entry)) {
                struct task_struct *pp =
                        list_first_entry(&p->blocked_entry,
                                         struct task_struct,
                                         blocked_entry);

                list_del_init(&pp->blocked_entry);
                activate_task(rq, pp, en_flags);
                pp->on_rq = TASK_ON_RQ_QUEUED;
                resched_curr(rq);
        }
}
```

# Block-chain



# Mutex #2

- Mutex handoff (unlock)
  - `current->blocked_task`
- Can be `NULL`....
  - Suppose owner is highest prio owner
- Must reschedule

# SMP

- Races!
  - against: `mutex_unlock()` (PE #4)
    - Owner is already dead
    - Owner is you
  - against: wakeups (PE #5)
    - Wakeup can miss us being added to the blocked_entry
- That fun affinity thing..
  - Running 1 task on 2 cpus
  - 'breaking' affinity
    - Blocked tasks have no affinity

# PE #4

```
static struct task_struct *
proxy(struct rq *rq, struct task_struct *next)
{
    struct task_struct *p, *owner;
    struct mutex *mutex;

    for (p = next; p->blocked_on; p = owner) {
        mutex = p->blocked_on;

        raw_spin_lock(&mutex->wait_lock);
        owner = __mutex_owner(mutex);
        if (owner == p)
            goto owned;
        if (!owner->on_rq)
            goto blocked;
        raw_spin_unlock(&mutex->wait_lock);

        owner->blocked_task = p;
    }

    return owner;

owned:
    owner->blocked_on = NULL;
    owner->state = TASK_RUNNING;
    raw_spin_unlock(&mutex->wait_lock);
    return owner;

    /* ... */
}
```

# PE #5

```
static struct task_struct *
proxy(struct rq *rq, struct task_struct *next)
{
     /* ... */

blocked_task:
     raw_spin_lock(&owner->blocked_lock);
     if (owner->on_rq) {
          raw_spin_unlock(&owner->blocked_lock);
          goto retry_owner;
     }

     for (; p; p = p->blocked_task) {
          p->on_rq = 0;
          deactivate_task(rq, p, DEQUEUE_SLEEP);
          list_add(&p->blocked_entry, &owner->blocked_entry);
     }
     raw_spin_unlock(&owner->blocked_lock);
     raw_spin_unlock(&mutex->wait_lock);

     return NULL; /* retry task selection */
}
```

```
static inline void
ttwu_activate(struct rq *rq, struct task_struct *p, int en_
{
     raw_spin_lock(&p->blocked_lock);
     activate_task(rq, p, en_flags);
     p->on_rq = TASK_ON_RQ_QUEUED;

     while (!list_empty(&p->blocked_entry)) {
          struct task_struct *pp =
               list_first_entry(&p->blocked_entry,
                                struct task_struct,
                                blocked_entry);

          list_del_init(&pp->blocked_entry);
          activate_task(rq, pp, en_flags);
          pp->on_rq = TASK_ON_RQ_QUEUED;
          resched_curr(rq);
     }
     raw_spin_unlock(&p->blocked_lock);
}
```

# SMP #2

- Migrate 'blocked' tasks towards the executable task.
  - Migration is tricky:
    - Migrate at first cpu-crossing in the block chain
    - Migrate towards the observed remote CPU
      - No guarantee the task is still there, or is in fact the final executable task.
    - Migrate from the idle task

# SMP #3

```
/*
 * CPU0            CPU1
 *
 *              B mutex_lock(X)
 *
 * A mutex_lock(X) <- B
 * A __schedule()
 * A pick->A
 * A proxy->B
 * A migrate A to CPU1
 *              B mutex_unlock(X) -> A
 *              B __schedule()
 *              B pick->A
 *              B switch_to (A)
 *              A ... does stuff
 * A ... is still running here
 *
 *      * BOOM *
 */
```

# SMP #4

- Waking 'migrated' tasks up is tricky
  - Check affinity; put to sleep

```
/*
 *                                lock(&rq->lock);
 *                                  proxy()
 *
 * mutex_unlock()
 *   lock(&wait_lock);
 *   owner = current->blocked_task;
 *   unlock(&wait_lock);
 *
 *   wake_up_q();
 *     ...
 *       ttwu_remote()
 *         __task_rq_lock()
 *                                lock(&wait_lock);
 *                                owner == p
 */
```