Demystifying Proxy Execution

John Stultz <jstultz@google.com>

LINUX PLUMBERS CONFERENCE
Vienna, Austria
Sept. 18-20, 2024

Thank yous!

Proxy Execution has been worked on by numerous folks, who deserve a lot of credit

Watkins, Straub, Niehaus (RTLWS11)

Peter Zijlstra (RTSumit17)

Juri Lelli (2018 patchset, OSPM19)

Valentin Schneider (LPC20 slides)

Connor O'Brien (2022 patchset)

With additional help from and thanks to:

Joel Fernandes, Dietmar Eggemann, Qais Yousef,
Metin Kaya, K Prateek Nayak and others!

Background

Proxy Execution: Why?

Android uses concept of FOREGROUND vs BACKGROUND apps

As devices memory grows, we can keep more apps running in the background, so one can switch between apps faster.

Android tasks run mostly as SCHED_NORMAL (fair)

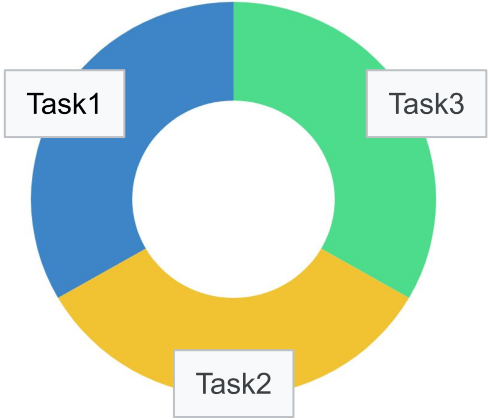
Which means each runnable task gets ~equal time on the cpu as every other runnable task.

More running tasks => proportionately less time per task

But tasks aren't equally important.

Performance of BACKGROUND tasks doesn't matter as much as FOREGROUND task being actively used.

Want to make sure BACKGROUND tasks don't negatively affect FOREGROUND tasks.



Proxy Execution: Why?

Use cgroups to restrict background tasks:

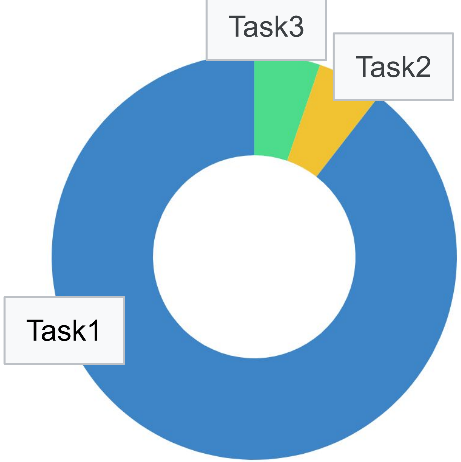
Bound background tasks to “small” cpus with cpusets, and use cpu.share cgroup to further restrict cputime of background tasks

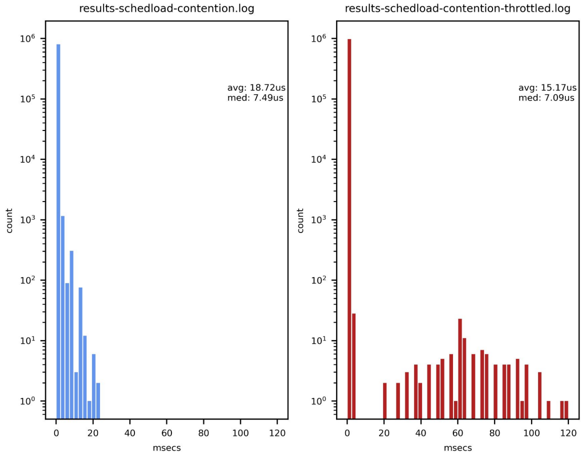
But this runs into trouble:

While this configuration often improves FOREGROUND performance on average, we see really bad outliers.

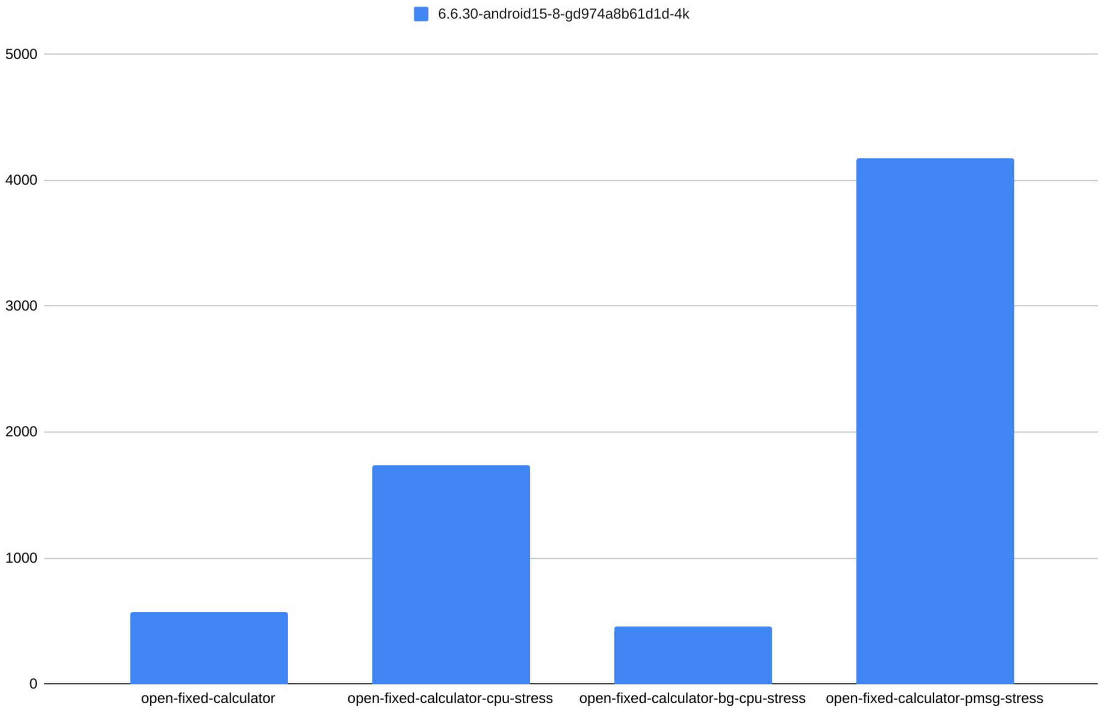
Classic Priority Inversion

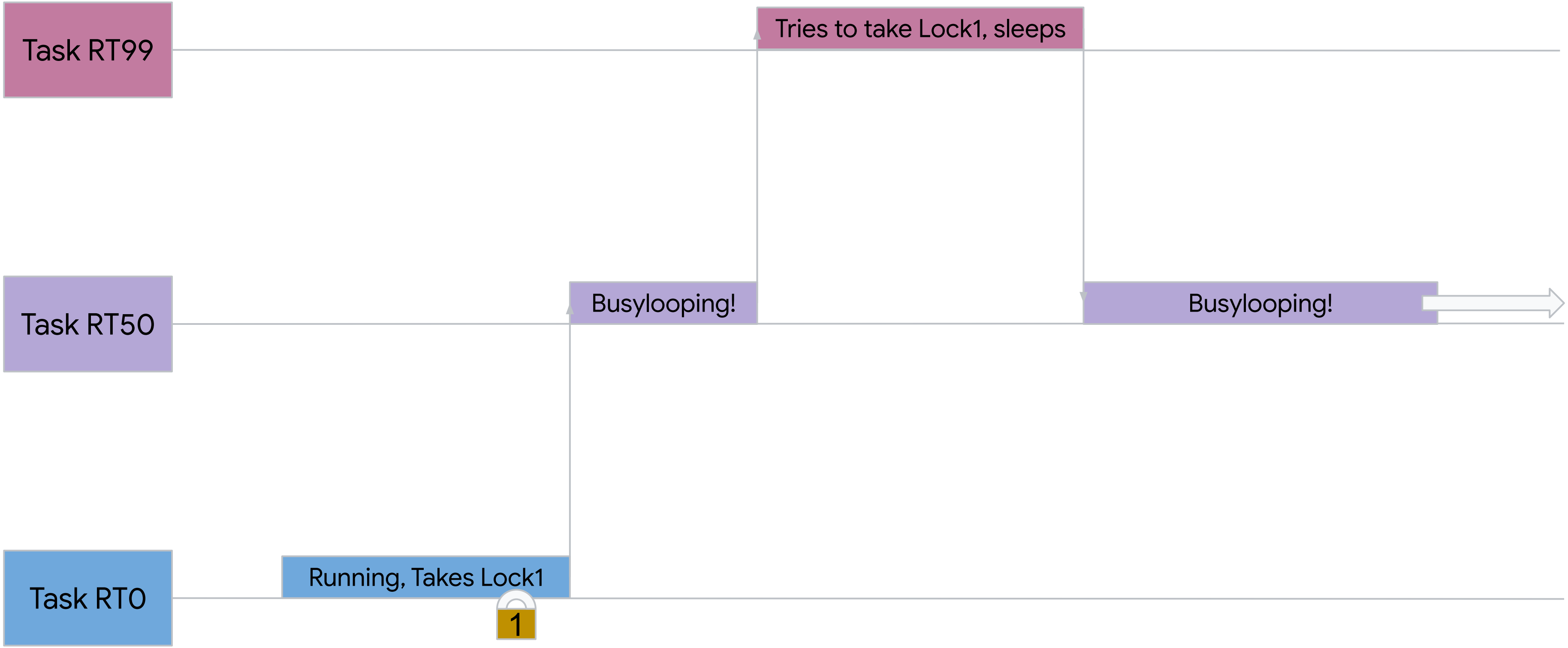
If background task manages to take a lock, it may be some time before it can run long enough to release it! Won’t deadlock, but may be longer then we like





Proxy Execution: Why?





Task RT99

Task RT50

Task RT0

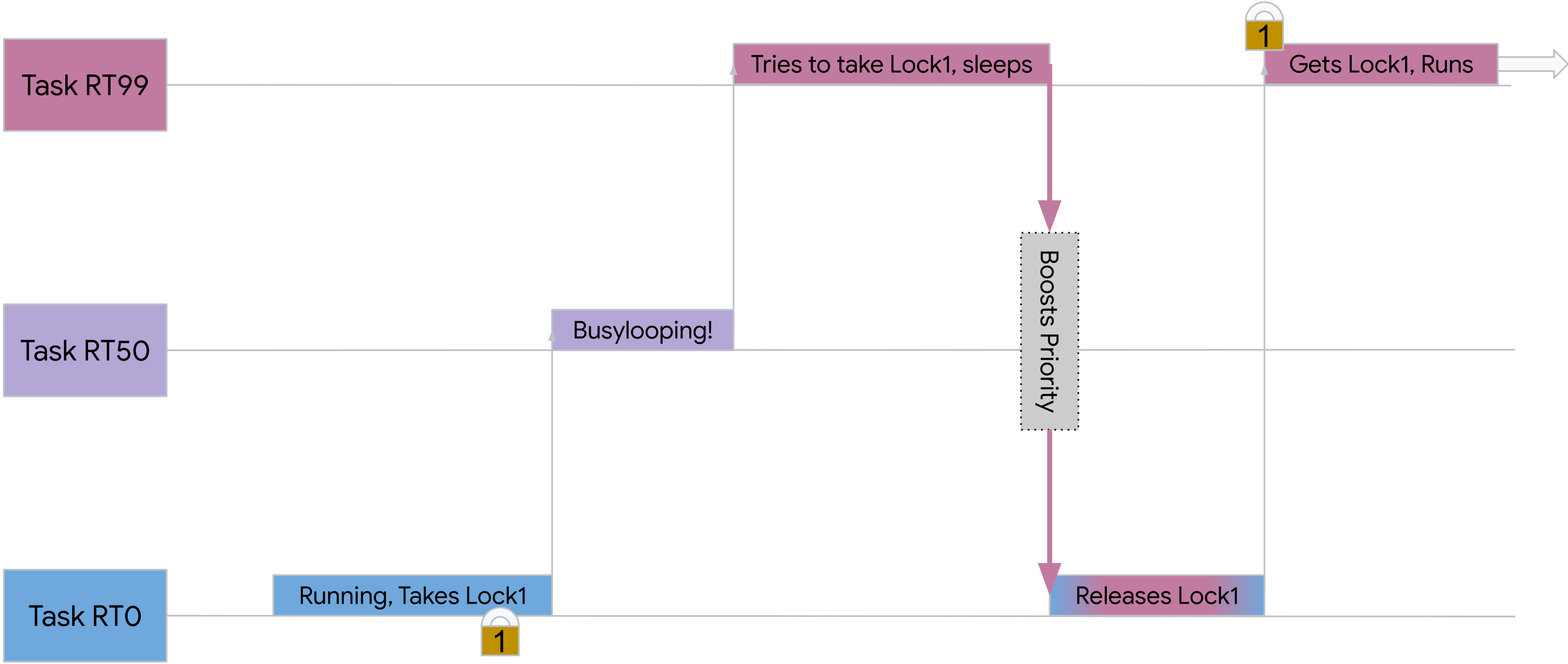
Running, Takes Lock1

1

Busylooping!

Tries to take Lock1, sleeps

Busylooping!



Task RT99

Task RT50

Task RT0

Running, Takes Lock1

1

Busylooping!

Tries to take Lock1, sleeps

Boosts Priority

Releases Lock1

1

Gets Lock1, Runs

Generalized Priority Inheritance

SCHED_NORMAL doesn't have strict linear priority order!

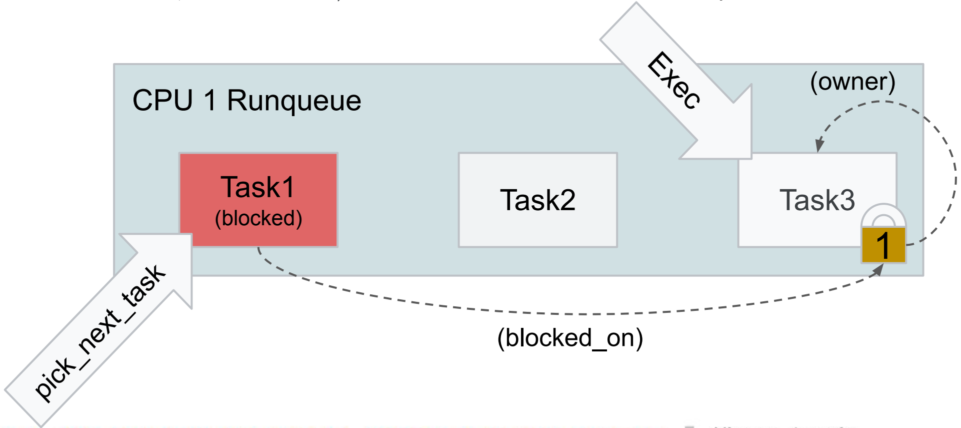
Priority inheritance won't work as selection is based on dynamic vruntime values and nests into cgroups, so there isn't a singular value to inherit.

So the idea is to use the scheduler selection function itself.

1) **Leave the mutex blocked tasks on the runqueue**
2) Use `pick_next_task()` to pick *whatever* is the most best task to run at a given time
3) If its mutex blocked, **find the owner, and run that!**

Simplified code:

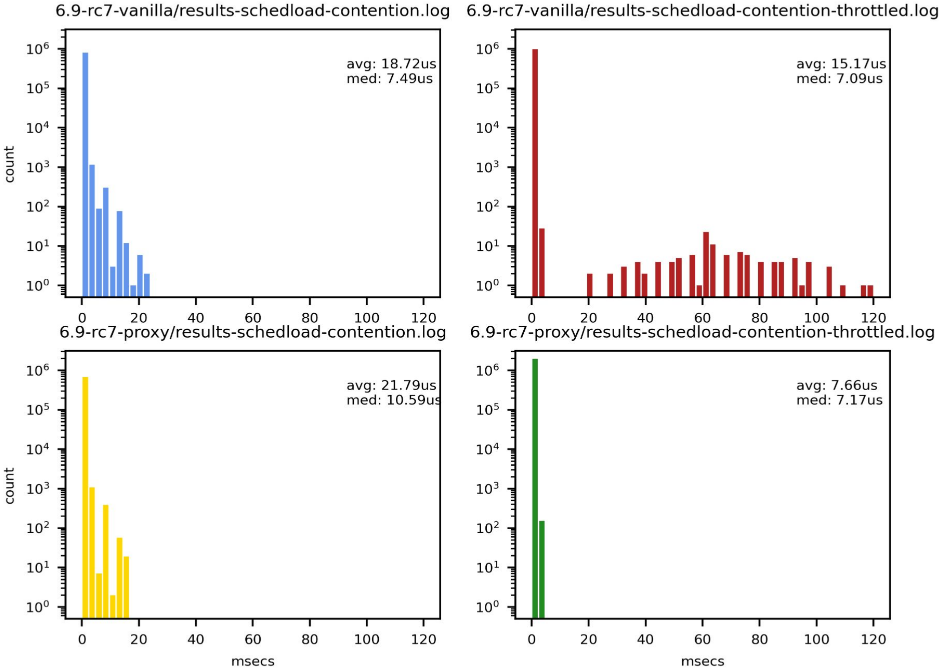
```
__schedule():
    ...
    next = pick_next_task(rq, prev, &rf);
    rq_set_donor(rq, next);
    if (unlikely(task_is_blocked(next)))
        next = find_proxy_task(rq, next, &rf);
    ...
    rq = context_switch(rq, prev, next, &rf);
```



Proxy Execution: Benefits

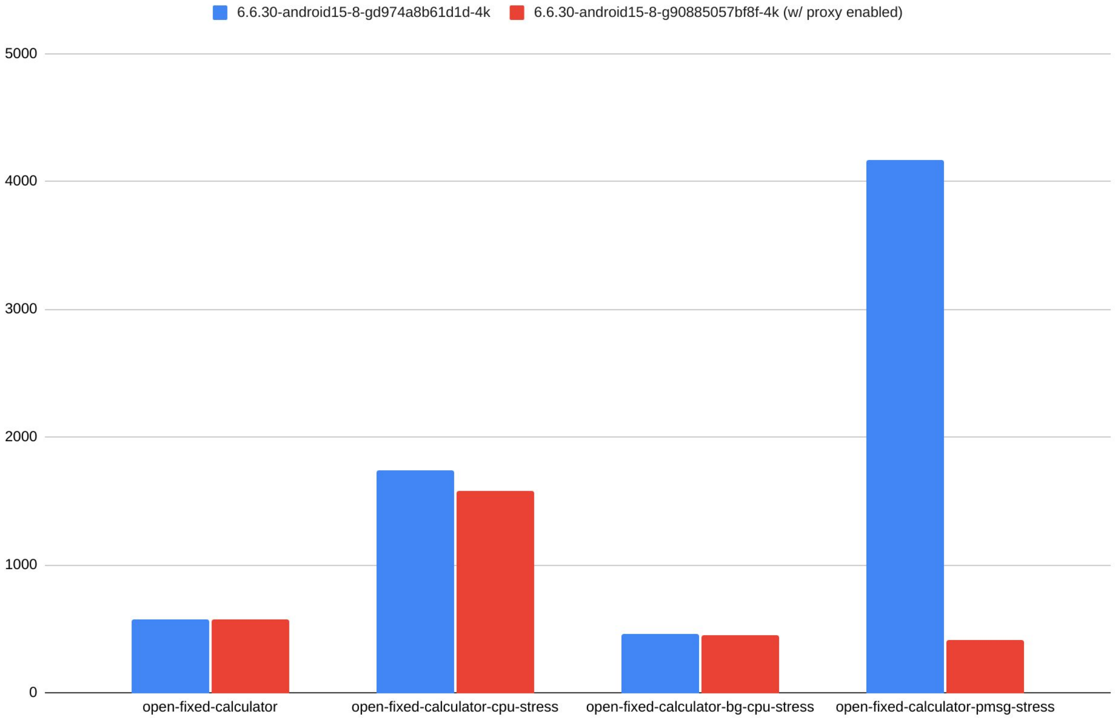
Vanilla:

Proxy-exec:



https://github.com/johnstultz-work/priority-inversion-demo

Proxy Execution: Benefits



Dual contexts

In a way, we have two “running” tasks

Task that is waiting for the mutex, that was chosen to run, that the proxy task runs on behalf of.

- `rq->donor`
- If mutex blocked, can’t actually run
- Also called the “scheduler context”, “waiter” or “donor” task

Task that owns the mutex that is actually run

- `rq->curr`
- Also called “execution context”, or the “owner” or “proxy” task
- Runs on behalf of the donor, using the donor’s “scheduler context”

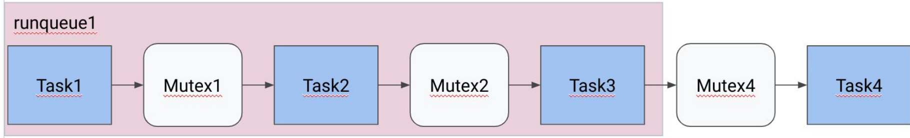
While we run the `rq->curr`, in most cases, we do accounting, etc using `rq->donor`.

See related patch in series

Task/Mutex Chains

In order to figure out what task to run, we have to look at the mutex we're blocked on and find it's owner.

- Add `task->blocked_on` ptr to point to mutex
- `mutex->owner` points to owning task.



Problem: This alternating type makes locking complex

- `task->blocked_on_lock` serializes task related state
- `mutex->wait_lock` serializes mutex related state
- Lockdep won't let us take blocked_on_lock -> wait_lock, and wait_lock -> blocked_on_lock
- Have to let go of the locks when traversing task->blocked_on pointer!

Holding the rq->lock to keep tasks from disappearing

Also, when we hold the mutex->wait_lock, we know the mutex->owner task can't disappear on us.

This lets us safely look at one task off the current runqueue in the chain.

Lock ordering:

1) `task->pi_lock`
2) `rq->lock`
3) `mutex->wait_lock`
4) `task->blocked_on_lock`

Related patch in series

Simple Proxying

(Same CPU)

Keeping mutex blocked tasks on the runqueue

```
static void __sched notrace __schedule(unsigned int sched_mode)
        ...
        prev = rq->curr;
        ...
        rq_lock(rq, &rf);
        ...
        prev_state = READ_ONCE(prev->__state);
        if (!(sched_mode & SM_MASK_PREEMPT) && prev_state) {
                try_to_deactivate_task(rq, prev, prev_state,
                                       !task_is_blocked(prev));
                switch_count = &prev->nvcsw;
        }
        ...
```

Annotations:

Save current running task as `prev`

Note for most of `__schedule`, we are holding the `rq->lock`

If `prev` is not runnable, call `try_to_deactivate_task()`, which will only deactivate if `prev` is mutex blocked (`!task_is_blocked(prev)`).

This is what keeps the mutex-blocked tasks on the runqueue.

Related patch in series

Further down in __schedule() logic

```
static void __sched notrace __schedule(unsigned int sched_mode)
        ...
pick_again:
        next = pick_next_task(rq, rq->donor, &rf);
        rq_set_donor(rq, next);
        next->blocked_donor = NULL;
        if (unlikely(task_is_blocked(next))) {
                next = find_proxy_task(rq, next, &rf);
                if (!next) {
                        zap_balance_callbacks(rq);
                        goto pick_again;
                }
                if (next == rq->idle)
                        preserve_need_resched = true;
        }
        if (!preserve_need_resched)
                clear_tsk_need_resched(prev);
        ...
```

Annotations:

Pick the next task as usual

Save the chosen task as the `rq->donor`

If chosen task is blocked, walk the mutex/task chain to find a runnable owner.

If `find_proxy_task()` returned null, we have to start over. `zap_balance_callbacks()` to undo callback state set by `pick_next_task()` and `goto pick_again`

If `find_proxy_task()` returned the idle task, it means we want to take action on `current`, so we have to switch to idle quickly first.

If we are switching quickly to idle, preserve the need_resched bit, so we will enter into `__schedule` again right after we switch to idle.

Related patch in series

find_proxy_task(): walking the chain

```
static struct task_struct *
find_proxy_task(struct rq *rq, struct task_struct *next,...)
        ...
        for (p = next; task_is_blocked(p); p = owner) {
                mutex = p->blocked_on;
                if (!mutex) return NULL;
                raw_spin_lock(&mutex->wait_lock);
                raw_spin_lock(&p->blocked_lock);
                if (mutex != get_task_blocked_on(p))
                        goto out;
                if (task_current(rq, p))
                        curr_in_chain = true;
                owner = __mutex_owner(mutex);
                < complex logic here >
                raw_spin_unlock(&p->blocked_lock);
                raw_spin_unlock(&mutex->wait_lock);
                owner->blocked_donor = p;
        }
        return owner;
```

Annotations:

Note: we currently hold the rq->lock when calling (for the sake of this slide, assume all the tasks we visit are on this rq)

Starting with next, iterate p through the task/mutex chain while it is mutex blocked.

Grab mutex->wait_lock and p->blocked_lock

Re-validate unlocked mutex = p->blocked_on access is still valid after we have taken the locks

If p is current, set curr_in_chain flag (used later)

Get the mutex owner

Let go of the locks, and set reverse trail via blocked_donor

Loop, moving p to point to the owner

When we have hit an unblocked owner, return it!

Related patch in series

Proxying Across Runqueues

(Proxy Migration)

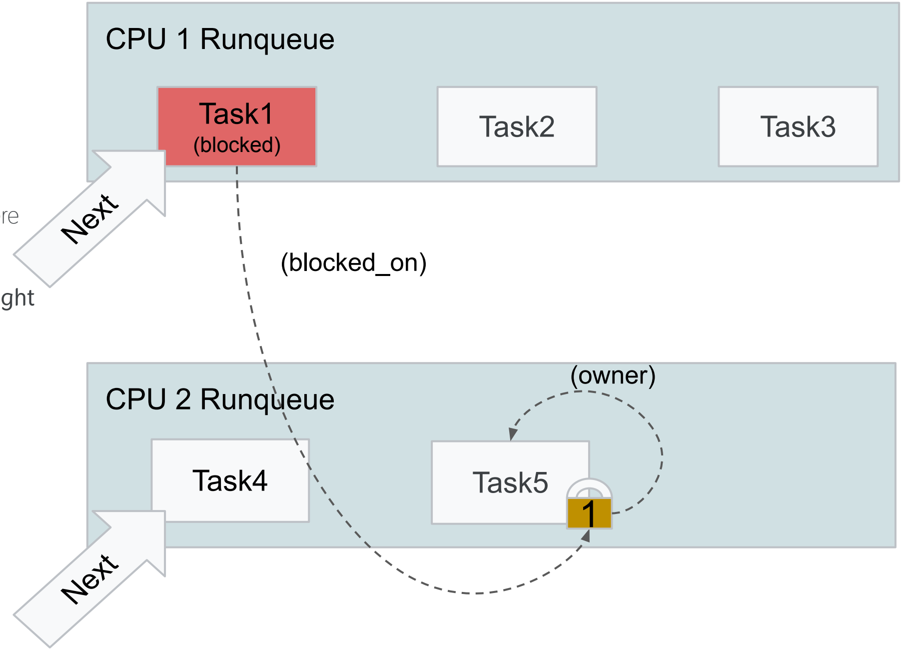
Proxying across cpu runqueues

The lock owner may not be on the same cpu as the blocked waiter

If the owner is on another cpu, there are **two options**

1) Migrate the owner to the waiters' cpu and run it there
2) Migrate the waiter to the owner's cpu, and boost it there

Unfortunately, #1 won't always work, as **owner's cpu affinity might not allow** it to run on the waiters's cpu



Proxying across cpu runqueues

The lock owner may not be on the same cpu as the blocked waiter

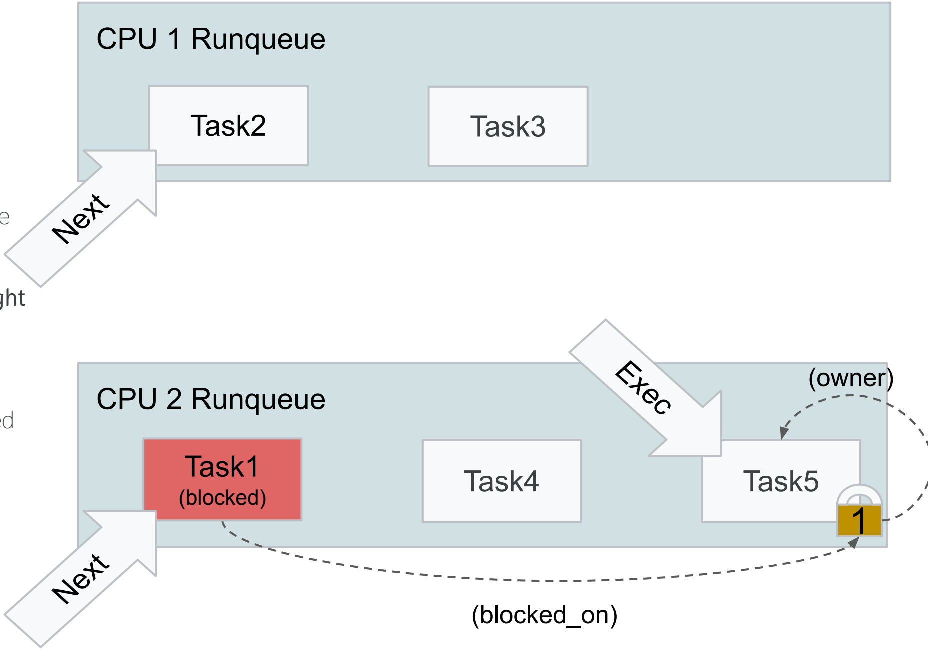
If the owner is on another cpu, there are **two options**

1) Migrate the owner to the waiters' cpu and run it there
2) Migrate the waiter to the owner's cpu, and boost it there

Unfortunately, #1 won't always work, as **owner's cpu affinity might not allow** it to run on the waiters's cpu

So we **migrate waiter to remote runqueue**, and let it be selected as the rq->donor to boost the lock owner.

The donor doesn't actually run, so this is ok.



Proxying across cpu runqueues

The lock owner may not be on the same cpu as the blocked waiter

If the owner is on another cpu, there are **two options**

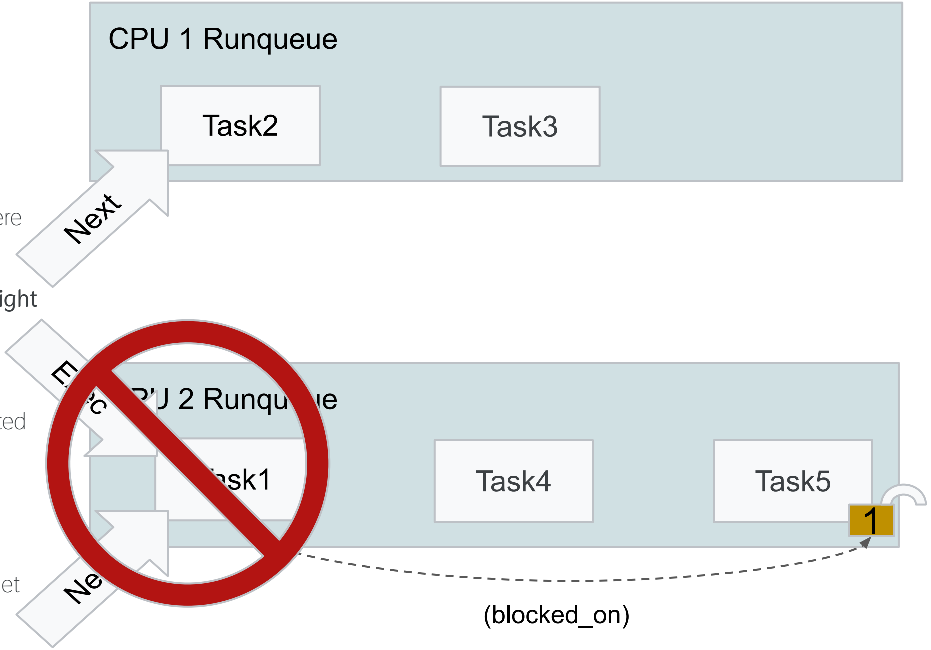
1) Migrate the owner to the waiters' cpu and run it there
2) Migrate the waiter to the owner's cpu, and boost it there

Unfortunately, #1 won't always work, as **owner's cpu affinity might not allow** it to run on the waiters's cpu

So we **migrate waiter to remote runqueue**, and let it be selected as the rq->donor to boost the lock owner.
The donor doesn't actually run, so this is ok.

Have to be careful! If lock owner releases the lock, we can't just let the donor run on the remote cpu! Its affinity may not allow it.

Need to make sure it's affinity allows it, and do **return-migration** back to a cpu it can run on (more on this later)



find_proxy_task(): owner on remote rq?

```
for (p = next; task_is_blocked(p); p = owner) {

        ...

        owner_cpu = task_cpu(owner);
        if (owner_cpu != cur_cpu) {
                raw_spin_unlock(&p->blocked_lock);
                raw_spin_unlock(&mutex->wait_lock);
                if (curr_in_chain)
                        return proxy_resched_idle(rq, next);

                proxy_migrate_task(rq, rf, p, owner_cpu);
                return NULL;
        }
```

Annotations:

Continuing find_proxy_task loop, walking through the chain

If we find the owner's cpu isn't the current cpu, we can't go any further!

Let go of the locks

If current is in the chain, we can't migrate it, since its running right now! So return idle, to quickly switch and we will try again.

Migrate p to the owner_cpu, and return NULL (forcing pick_again)

Related patch in series

proxy_migrate_task(): part 1

```
static void proxy_migrate_task(struct rq *rq, struct rq_flags *rf,
                               struct task_struct *p, int target_cpu)
        ...
        put_prev_task(rq, rq->donor);
        rq_set_donor(rq, rq->curr);
        set_next_task(rq, rq->curr);

        for (; p; p = p->blocked_donor) {
                deactivate_task(rq, p, 0);
                proxy_set_task_cpu(p, target_cpu);
                list_add(&p->migration_node, &migrate_list);
        }

        zap_balance_callbacks(rq);
        rq_unpin_lock(rq, rf);
        raw_spin_rq_unlock(rq);
        ...
```

Annotations:

We can't hold the rq lock if we want to push a task to another rq. So we have a bunch of things to undo to make it safe to drop the rq lock before we do the migration and start over.

Earlier we called put_prev_task() on prev. But if we are going to release the rq lock we have to undo all that. So put_prev_task on donor, set rq->curr (same as prev at this point) as donor and call set_next_task() on it as well.

Walk backward up the chain, using blocked_donor ptr, deactivating each task from this rq and setting the task_cpu to target_cpu and add each task to the migration_list

Zap callbacks setup by pick_next_task, then unpin and unlock the rq lock.

Related patch in series

Another related patch in series

proxy_migrate_task(): part 2

```
static void proxy_migrate_task(struct rq *rq, struct rq_flags *rf,
                              struct task_struct *p, int target_cpu)
        ...
        raw_spin_rq_lock(target_rq);
        while (!list_empty(&migrate_list)) {
                p = list_first_entry(&migrate_list,
                                     struct task_struct,
                                     migration_node);
                list_del_init(&p->migration_node);
                activate_task(target_rq, p, 0);
                check_preempt_curr(target_rq, p, 0);
        }
        raw_spin_rq_unlock(target_rq);
        raw_spin_rq_lock(rq);
        rq_repin_lock(rq, rf);

        proxy_resched_idle(rq, rq->curr);
}
```

Annotations:

Since we've let go of the rq lock, grab the target_rq lock

Iterate through the migrate_list, activating each task on the target_rq, and seeing if it should preempt the target_rq->curr

Now the migration is done, let go of target_rq, and re-grab the rq lock

Resched the idle task, and return so we can pick again.

[Related patch in series]

[Another related patch in series]

Proxy Return-Migration

Ensuring proper return migration

To ensure we return-migrate tasks, we need more state

We include a blocked_on_state in the task struct. This tri-state ensures that when a mutex has been released and the task's blocked_on pointer is cleared, we still need to evaluate if the task needs to be return migrated before it can be run.

Thought: It feels like we might be able to merge this state into the task_state (TASK_RUNNING, TASK_INTERRUPTABLE, etc), but I've not quite worked out how.

```
enum blocked_on_state {
        BO_RUNNABLE,
        BO_BLOCKED,
        BO_WAKING,
};

static inline
bool task_is_blocked(struct task_struct *p)
{
    return !!p->blocked_on &&
           p->blocked_on_state != BO_RUNNABLE;
}
```

Related patch in series

try_to_wakeup() details:

```
...
if (!ttwu_state_match(p, state, &success)) {
     /*
      * If we're already TASK_RUNNING, and BO_WAKING
      * continue on to ttwu_runnable check to force
      * proxy_needs_return evaluation
      */
     if (!(READ_ONCE(p->__state) == TASK_RUNNING &&
           READ_ONCE(p->blocked_on_state) == BO_WAKING))
               break;
}
...
smp_rmb();
if (READ_ONCE(p->on_rq) && ttwu_runnable(p, wake_flags))
    break;
...
```

Annotations:

Normally if the ttwu_state_match failed, the task was already runnable, no reason to wake it.

But special case when we're TASK_RUNNING, but BO_WAKING (ie: possibly in need of return migration), and don't break early.

Instead carry on with the wakeup process.

Mutex blocked tasks are kept on the rq, so we will continue on to checking ttwu_runnable.

Related patch in series

ttwu_runnable() details:

```
    ret = 0;
    ...
    rq = __task_rq_lock(p, &rf);
    if (task_on_rq_queued(p)) {
        if (!task_on_cpu(rq, p)) {
            ...
        }
        if (proxy_needs_return(rq, p))
            goto out;

        ttwu_do_wakeup(p);
        ret = 1;
    }
out:
    __task_rq_unlock(rq, &rf);
   return ret;
```

Annotations:

Grab’s the task->pi_lock and the rq->lock (convenient as we need these in proxy_needs_return!)

Mostly this function is left unchanged

One special case, where if proxy_needs_return() returns true, we skip the ttwu_do_wakeup, and return zero.

This is because proxy_needs_return will deactivate the mutex blocked task that was on the rq! So afterwards it’s back to not being runnable!

Related patch in series

proxy_needs_return():

```
static inline
bool proxy_needs_return(struct rq *rq, struct task_struct *p)
        bool ret = false;
        raw_spin_lock(&p->blocked_lock);
        if (get_task_blocked_on(p) &&
             p->blocked_on_state == BO_WAKING) {
                if (!task_current(rq, p) &&
                    (p->wake_cpu != cpu_of(rq))) {
                        if (task_current_donor(rq, p)) {
                                put_prev_task(rq, p);
                                rq_set_donor(rq, rq->idle);
                        }
                        deactivate_task(rq, p, DEQUEUE_NOCLOCK);
                        ret = true;
                }
                p->blocked_on_state = BO_RUNNABLE;
                resched_curr(rq);
        }
        raw_spin_unlock(&p->blocked_lock);
        return ret;
```

Annotations:

Called in `ttwu_runnable()`, which took `__task_rq_lock()` so we hold needed locks.

We only need to do something if the task is BO_WAKING, and assuming it isn't current (so already running), and the wake_cpu isn't this cpu.

If p is the current donor, put_prev_task and set the donor to idle

Remember, blocked tasks kept on the rq, so deactivate p on this rq, so we will later activate it in try_to_wake_up() up the call stack on its wake_cpu.

Set the task BO_RUNNABLE, and resched cur.

Return true only if we deactivated the task

Related patch in series

try_to_wakeup() details: (continued)

```
...
WRITE_ONCE(p->__state, TASK_WAKING);
set_blocked_on_runnable(p);

...
cpu = select_task_rq(p, p->wake_cpu, wake_flags | WF_TTWU);
if (task_cpu(p) != cpu) {
    ...
    wake_flags |= WF_MIGRATED;
    psi_ttwu_dequeue(p);
    set_task_cpu(p, cpu);
}
...
```

Annotations:

After we’ve checked ttwu_runnable(), which through proxy_needs_return() deactivated the task we’re waking, we set the task as BO_RUNNABLE

Go through the normal wakeup runqueue selection (unchanged) which will utilize the saved wake_cpu to return migrate the now mutex unblocked task to a cpu its allowed to run on.

Related patch in series

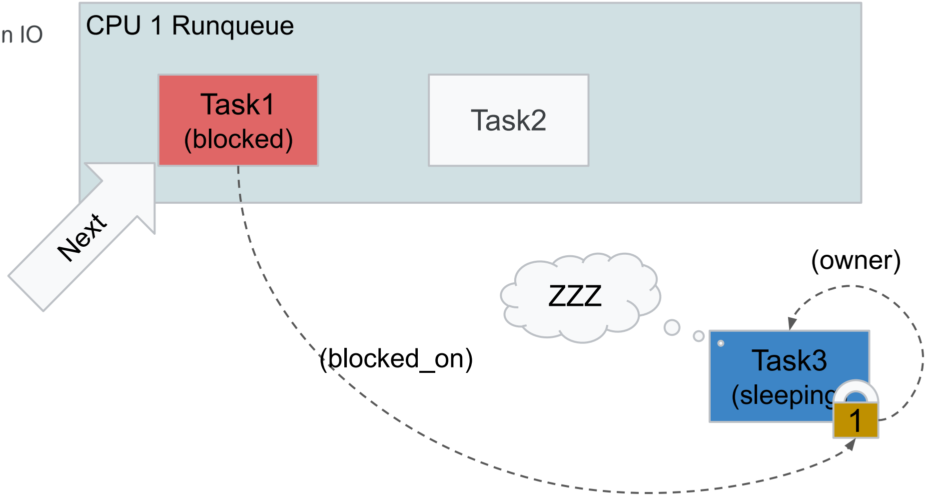
Sleeping Owner Enqueuing

(And Blocked Entities Activation)

Blocked on a sleeping mutex owner

The lock owner may have ended up sleeping or blocked on IO

Nothing we can do to make it run!



Blocked on a sleeping mutex owner

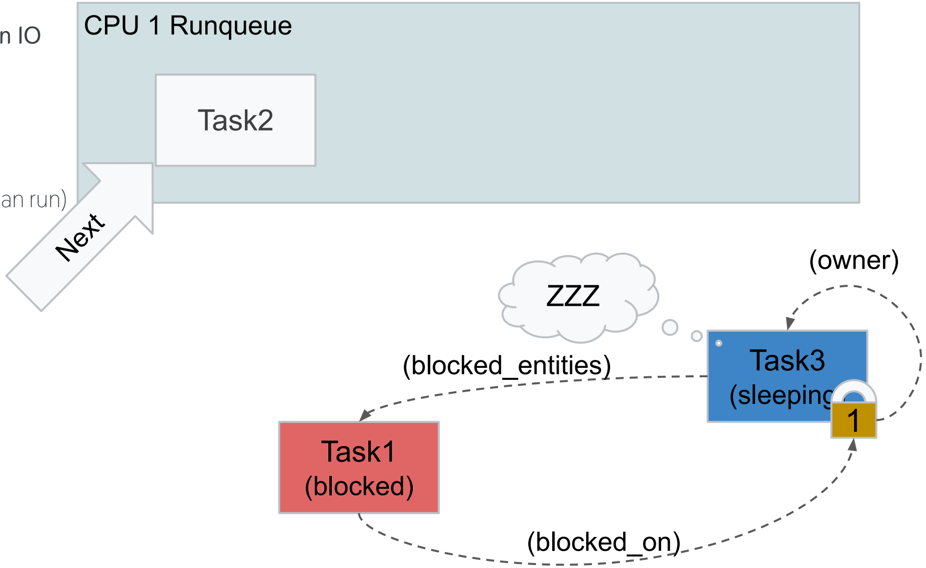
The lock owner may have ended up sleeping or blocked on IO

Nothing we can do to make it run!

So we **deactivate it from the runqueue** (so something else can run)

And **enqueue it on the sleeping owner**

When the sleeping task is woken up, **activate all it's blocked_entities**



find_proxy_task(): owner is sleeping

```
for (p = next; task_is_blocked(p); p = owner) {
    ...
    if (!owner->on_rq) {
      if (curr_in_chain) {
        raw_spin_unlock(&p->blocked_lock);
        raw_spin_unlock(&mutex->wait_lock);
        return proxy_resched_idle(rq, next);
      }
      if (owner != p) {
        raw_spin_unlock(&p->blocked_lock);
        raw_spin_lock(&owner->blocked_lock);
      }
      proxy_resched_idle(rq, next);
      proxy_enqueue_on_owner(rq, owner, next);

      raw_spin_unlock(&owner->blocked_lock);
      raw_spin_unlock(&mutex->wait_lock);
      return NULL; /* retry task selection */
    }
```

Annotations:

Continuing find_proxy_task loop, walking through the chain

If the owner isn’t on a runqueue, check current isn’t in the chain, and if it is resched idle and return (we’ll get back here again after we switch)

Switch to holding the owner’s blocked_lock.

Resched idle (since we’re not going to run next), and enqueue the chosen task onto the owner.

Drop the locks and return null, so we pick_again.

[Related patch in series]

proxy_enqueue_on_owner(): Adding waiter to sleeping task

```
static void
proxy_enqueue_on_owner(struct rq *rq, struct task_struct *owner,
                                   struct task_struct *next)
{
        if (!owner->on_rq) {
                deactivate_task(rq, next, DEQUEUE_SLEEP);
                get_task_struct(owner);
                next->sleeping_owner = owner;
                list_add(&next->blocked_node, &owner->blocked_head);
        }
}
```

Annotations:

Assuming the owner is still not on_rq, deactivate the waiting task next

Take a reference to the owner struct

Keep track of who the waiter is enqueued on

Add waiter to the owner's blocked_head

[Related patch in series]

activate_blocked_waiters(): The nightmare!

Unfortunately, activate_blocked_waiters() is too complicated to cover on a slide.

Iterating through the list of tasks on the waking task's blocked_head and activating them is relatively simple enough. Though we have to take the task->pi_lock and rq->lock and release them for each task activated.

But we also have to activate all the tasks that are blocked on those tasks. It can be a tree structure.

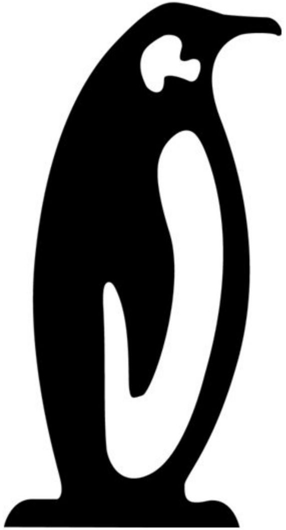
Lots of dropping and taking of locks, with lots of races possible, including sub-tree wakeups!

[Related patch in series]

Thank you!

John Stultz <jstultz@google.com>

Full patch set referenced in these slides:
https://github.com/johnstultz-work/linux-dev/commits/proxy-exec-v12-6.11-rc5/



Linux Plumbers Conference

Vienna, Austria | September 18-20, 2024