Much Ado About Blocking: Wait/Wake in the Linux Kernel

LinuxCon China – June 2017. Beijing, China.

Davidlohr Bueso <dave@stgolabs.net>
SUSE Labs.

SUSE
We adapt. You succeed.

Agenda

1. Fundamentals of Blocking

2. Flavors of blocking
   - Wait-queues
   - Simple wait-queues
   - Lockless wake-queues
   - rcuwait

Fundamentals of Blocking

Blocking 101

- As opposed to busy waiting, sleeping is required in scenarios where the wait time can be too long.
  - Other tasks ought to therefore run instead of wasting cycles.
  - Overhead of context switch vs wasting cycles.

Blocking 101

- As opposed to busy waiting, sleeping is required in scenarios where the wait time can be too long.
  - Other tasks ought to therefore run instead of wasting cycles.
  - Overhead of context switch vs wasting cycles.

- `TASK_{UNINTERRUPTIBLE,INTERRUPTIBLE,KILLABLE}`

- There are three elements to consider when waiting on an event:
  - The wakee (wake_up, wake_up_process, etc).
  - The waker (schedule, schedule_timeout, io, etc).
  - The condition/event

sleep_on()

```
CPU0                      CPU1

while (!cond)             cond = true
  sleep_on()              wake_up()
```

sleep_on()

```
CPU0                      CPU1
while (!cond)             cond = true
  sleep_on()              wake_up()
```

- Inherently racy on SMP.
  - Missed wakeups
- Synchronization must be provided.
  - Locking
  - Memory barriers

SMP-safe Blocking

```
for (;;) {
  set_current_state();
  if (cond)
    break;
  schedule();
}
__set_current_state(TASK_RUNNING);
```

SMP-safe Blocking

```
for (;;) {
    set_current_state();
    if (cond)
      break;
    schedule();
  }
  __set_current_state(TASK_RUNNING);
```

Pairs with barrier in wake_up()

Wait-queues

Wait-queues

- Provide different wrappers, such that users do not suffer from races aforementioned.

```
for (;;) {
  long __int = prepare_to_wait_event(&wq, &__wait,
  state);
  if (condition)
    break;
  schedule();
}
finish_wait(&wq, &__wait);
```

Wait-queues

- Provide different wrappers, such that users do not suffer from races aforementioned.

```
for (;;) {
  long __int = prepare_to_wait_event(&wq, &__wait,
  state);
  if (condition)
    break;
  schedule();
}
finish_wait(&wq, &__wait);
```

Wait-queues

- Provide different wrappers, such that users do not suffer from races aforementioned.

```
for (;;) {
  long __int = prepare_to_wait_event(&wq, &__wait,
  state);
  if (condition)
    break;
  schedule();
}
finish_wait(&wq, &__wait);
```

Wait-queues

- Provide different wrappers, such that users do not suffer from races aforementioned.

```
for (;;) {
   long __int = prepare_to_wait_event(&wq, &__wait,
   state);
   if (condition)
      break;
   schedule();
}
finish_wait(&wq, &__wait);
```

removal keeps the process from seeing multiple wakeups

When Wait-queues Were Simple

- Basic linked list of waiting threads.

- A wake_up() call on a wait queue would walk the list, putting each thread into the runnable state.

Wait-queues

- Exclusive Wait
  - Only the first N tasks are awoken.

- Callback mechanism were added for asynchronous I/O facility could step in instead.

Wait-queues

- Lockless waitqueue_active() checks

```
CPU0 - waker

cond = true;
smp_mb();
if (waitqueue_active(wq))
  wake_up(wq);
```

```
CPU1 - waiter

for (;;) {
 prepare_to_wait(&wq, &wait, state);
   // smp_mb() from set_current_state()
   if (cond)
     break;
   schedule();
}
finish_wait(&wq, &wait);
```

Wait-queues as Building Blocks

- Completions
  - Allows processes to wait until another task reaches some point or state.
  - Similar to pthread_barrier().
  - Documents the code very well.

- Bit waiting.

Wait-queues and RT

- Waitqueues are a big problem for realtime as they use spinlocks, which in RT are sleepable.
  - Cannot convert convert to raw spinlock due to callback mechanism.
  - This means we cannot perform wakeups in IRQ context.

Simple Wait-queues (swait)

Simple wait-queues

- Similar to the traditional (bulky) waitqueues, yet guarantees bounded irq and lock hold times.
  - Taken from PREEMPT_RT.

- To accomplish this, it must remove wq complexities:
  - Mixing INTERRUPTIBLE and UNINTERRUPTIBLE blocking in the same queue. Ends up doing O(n) lookups.
  - Custom callbacks (unknown code execution).
  - Specific task wakeups (maintaining list order is tricky).

Simple wait-queues

- Similar to the traditional (bulky) waitqueues, yet guarantees bounded irq and lock hold times.
  - Taken from PREEMPT_RT.

- To accomplish this, it must remove wq complexities:
  - Mixing INTERRUPTIBLE and UNINTERRUPTIBLE blocking in the same queue. Ends up doing O(n) lookups.
  - Custom callbacks (unknown code execution).
  - Specific task wakeups (maintaining list order is tricky).

- Results in a simpler wq, less memory footprint.

Simple wait-queues

- swait_wake_all(): task context

```
raw_spin_lock_irq(&q->lock);
list_splice_init(&q->task_list, &tmp);
while (!list_empty(&tmp)) {
   curr = list_first_entry(&tmp, typeof(*curr), task_list);
   wake_up_state(curr->task, TASK_NORMAL);
   list_del_init(&curr->task_list);

   if (list_empty(&tmp))
      break;
   raw_spin_unlock_irq(&q->lock);
   raw_spin_lock_irq(&q->lock);
}
raw_spin_unlock_irq(&q->lock);
```

Simple wait-queues

- swait_wake_all(): task context

```
raw_spin_lock_irq(&q->lock);
list_splice_init(&q->task_list, &tmp);
while (!list_empty(&tmp)) {
   curr = list_first_entry(&tmp, typeof(*curr)
   wake_up_state(curr->task, TASK_NORMAL);
   list_del_init(&curr->task_list);

   if (list_empty(&tmp))
      break;
   raw_spin_unlock_irq(&q->lock);
   raw_spin_lock_irq(&q->lock);
}
raw_spin_unlock_irq(&q->lock);
```

drop the lock (and IRQ disable) after every wakeup

Lockless Wake Queues

Wake Queues

- Internally acknowledge that one or more tasks are to be awoken, then call wake_up_process() after releasing a lock.
  - wake_q_add()/wake_up_q()

- Hold reference to each task in the list across the wakeup thus guaranteeing that the memory is still valid by the time the actual wakeups are performed in wake_up_q().

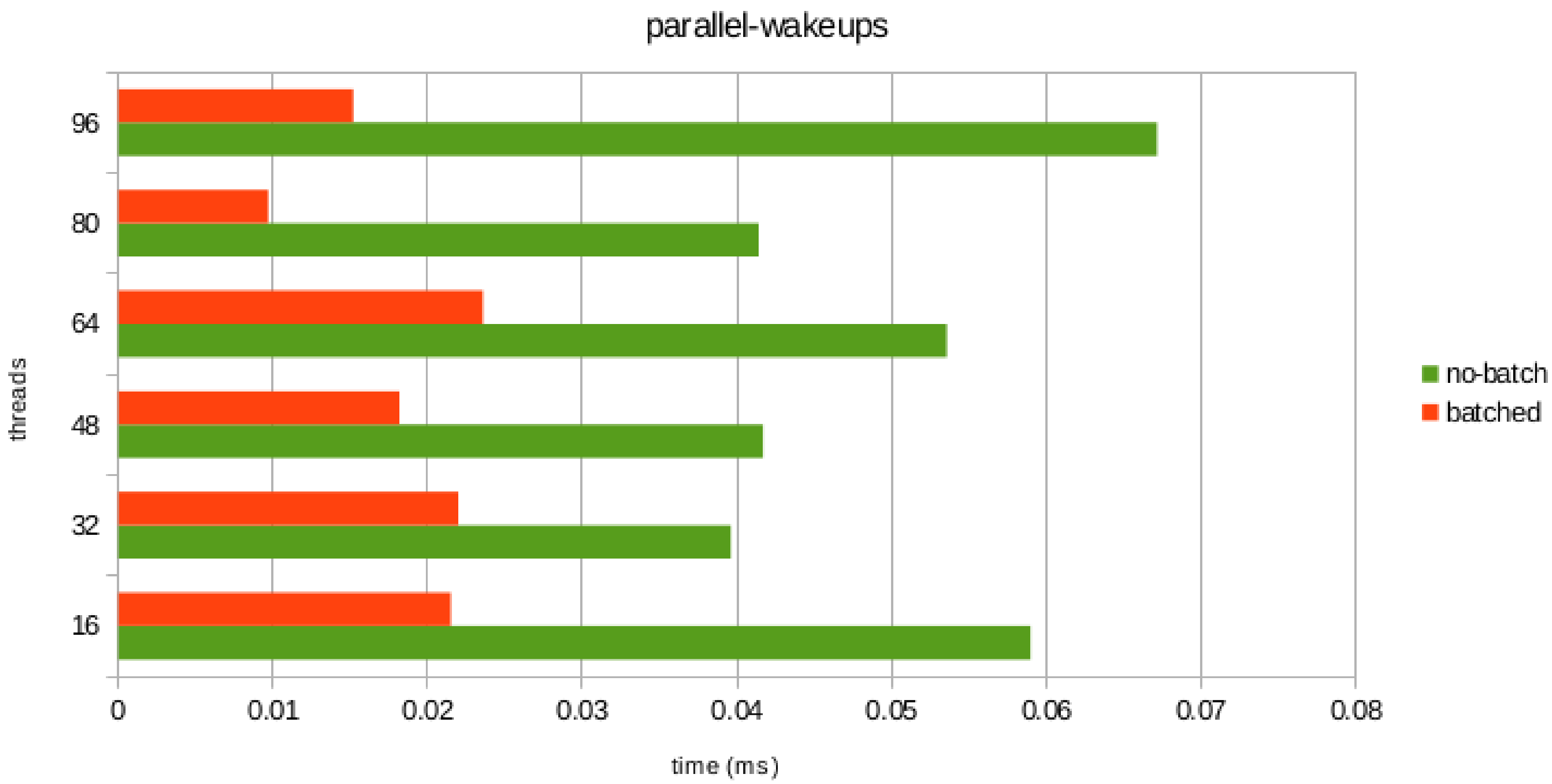
SUSE

Wake Queues

- Maintains caller wakeup order.

- Works particularly well for batch wakeups of tasks blocked on a particular event.
  - Futexes, locking, ipc.

Wake Queues



26-core, 2 socket x86-64 (Haswell)

rcuwait

rcuwait

- task_struct is not properly rcu protected unless dealing with an rcu aware list
  - find_task_by_*().
- delayed_put_task_struct() (via release_task()) can drop the last reference to a task.
  - Bogus wakeups, etc.
- Provides a way of blocking and waking up a single task in an rcu-safe manner.

rcuwait

- But what about task_rcu_dereference()?
  - Task freeing detection
  - probe_kernel_read()
  - May return a new task (false positives)

rcuwait

- If we ensure a waiter is blocked on an event before calling do_exit()/exit_notify(), we can avoid all of task_rcu_dereference overhead an magic.

- Currently used in percpu-semaphores.

References

References

- Documentation/memory-barriers.txt
- Documentation/scheduler/completion.txt
- Simple Wait Queues LWN
- Return of Simple Waitqueues (LWN)
- Source Code:
  - kernel/sched/{wait,swait}.c
  - kernel/sched/exit.c (rcuwait)
  - include/linux/sched/wake_q.h

Thank you.

We adapt. You succeed.