Pthreads condvars: POSIX compliance and the PI gap

Torvald Riegel
Principal Software Engineer
2016/10/11
Agenda

• Part 1: New glibc condition variable
  • POSIX requirements that required a new algorithm
  • How blocking with futexes makes this complicated
  • Brief overview of the new algorithm

• Part 2 by Darren: How PI makes this even more complicated
Condition variable

- Wait until a condition holds:

  ```
  pthread_mutex_lock(m);
  while (!condition) // Spurious condvar wake-ups are allowed
      pthread_cond_wait(cond, m);
  pthread_mutex_unlock(m);
  ```

- Satisfy a condition and signal that it (may have) changed:

  ```
  pthread_mutex_lock(m); // Optional
  condition = true;
  pthread_cond_signal(cond);
  pthread_mutex_unlock(m); // Optional
  ```
Condvar is an order of events, not just a counter

- POSIX, C++14: signals must wake one of the waiters that started to wait before the signal and have not been woken
  - Program can observe / construct ordering because cond_wait must release mutex atomically wrt start of waiting
  - Condvar must adhere to any ordering the program may have observed
- Condvar synchronization must model an order of waiters/signalers
  - For each signal, there is a set of eligible waiters allowed to consume the signal
  - Former (/ still current) algorithm did not prevent non-eligible waiters to steal signals from eligible waiters → new condvar algorithm required
If we only spin-wait, a simple sequence is enough

- Eligibility for wake-up determined through sequence of waiters (wseq, a simple shared counter)
- Waiters basically take 3 steps:
  1) Acquire position in wseq: Become eligible for subsequent signals
  2) Release mutex
  3) Spin-wait until as many signals sent as our position in wseq
- Signalers (assume program signals while having acquired the mutex):
  - If number of signals sent (ssent) >= wseq, nothing to do
  - Otherwise, increment ssent
- Results in FIFO condvar wake-up
- Timeouts, cancellation: Waiters send artificial signals to prevent lost wake-ups
  - Pretend they just consumed such an artificial signal immediately
1\textsuperscript{st} attempt at using futexes

- Instead of spin-waiting, call `futex_wait` eventually (w/ `ssent` as futex word)
- Problem: Futex wake-up order (step 3 on previous slide) can be different from wseq order (step 1)
  - Waiters can only `futex_wait` after releasing the mutex
  - Futexes provide no wake-up ordering guarantees (non-PI case) nor means to request a certain order that relates to the wseq order we chose
  - Waking all threads blocked in `futex_wait` is bad for performance
- Workaround: Eligibility can also be claimed if a waiter's `futex_wait` happens before a signal's `futex_wake`
  - Waiters wake up if `ssent` is larger than their wseq position
  - Waiters \textit{also} wake up if `futex_wake` returns 0
- Does this work?
1\textsuperscript{st} attempt bug 1: 
(wseq-ssent) < \#blocked waiters

- Scenario: Program can count how many waiters are still blocked, and only send that many signals
- If 2 waiters wake because of one cond\_signal call (1 through observing ssent, 1 through futex\_wait), then ssent is not incremented by 2 \rightarrow lost wake-ups
- Can waiters increment ssent if futex\_wait returns 0?
  - cond\_signal's ssent>=wseq check will hit early, so might run one futex\_wake less \rightarrow lost wake-ups
  - We might be able to count these events and find a work-around
- Any workaround will probably result in spurious condvar wake-ups whenever wseq order does not match futex wake-up order
1st attempt bug 2: Can't distinguish spurious futex wake-ups

- But… the kernel doesn't wake spuriously?!
  - POSIX requires that mutexes can be destroyed as soon as no thread is blocked anymore on the mutex (similar for condvars)
  - General futex design: Userspace fastpaths and futex ops are not atomic
  - Spurious wake-ups in practice because of this and memory reuse:
    1) Thread 1 releases mutex in userspace, gets suspended
    2) Thread 2 acquires mutex in userspace, destroys it, reuses memory for another futex
    3) Thread 1 resumes, calls futex_wake, other futex is woken spuriously
- Condvar can't distinguish between spurious and non-spurious wakeups
  - Spurious wake-ups don't increment ssent → We're back to bug 1, but worse
2\textsuperscript{nd} attempt: Maintain groups of eligible and non-eligible waiters

- New waiters start as non-eligible (group G2)
- Eligible group (G1) contains only eligible waiters
  - Each signal wakes some thread in G1: All eligible, a counter is sufficient
- When G1 is completely signaled, G2 becomes new G1
G1/G2 are roles mapped to 2 group slots in pthread_cond_t

- Condvar keeps track of which slot has which role
  - There always is a G2 for waiters to enter
  - wseq is still maintained, so waiters can detect aliasing of groups
- Reusing G1 as G2 requires quiescence to avoid ABA in futex_wait
  - Only need to wait for completion of futex_wait calls
- Incoming signal switches groups if G1 fully signaled
  - Quiesce G1 and make it the new G2
  - Make G2 the new G1 and add a signal to it
- G2 to G1 switch is simple
  - No change for existing G2 threads, no need to switch futexes
Priority Inheritance

Darren Hart
Principal Engineer / Intel Open Source Technology Center
2016/10/11
Unbounded Priority Inversion
Priority Inheritance Goals

1. Guarantee wakeup of highest priority eligible waiter
2. Avoid the thundering herd
Implementation Restrictions

- `rt_mutex` cannot be in a state with waiters and no owner.
- PI futexes impose value policy on the futex word (stores the TID and WAITERS), so cannot encode sequence information.
Considerations

• Concerned with Unbounded Priority Inversion with respect to the target mutex and locking implementation (not forward progress toward satisfying the condition)

• Priority Inheritance applies to SCHED_FIFO, SCHED_RR – but not SCHED_DEADLINE

• What are we interesting in solving?
Discussion
PI problem: Group quiescence

- When switching from G1 to G2, need to avoid futex_wait ABA
  - Need to quiesce group 1: Threads that ran futex_wake need to confirm that they have been woken
- Need to boost prio of those threads, but they have not acquired a lock
- No helper-futex-per-waiter possible because we need to support process-shared condvars
Potential solutions for the PI gap

- What do you really want? Is it really a condvar?
- Make the base condvar algorithm simpler
  - Other futex_wait conditions than simple inequality (eg, make wake-up conditional on futex word value and some relation)?
  - Let callers request a certain wake-up order?
- Solve PI vs. quiescence
  - 64b futex operations so we can version futex words and make ABA impossible in practice?
  - PI mechanism to boost all threads blocked on or having acquired a lock without actually acquiring the lock?
    - Requeueing threads is not sufficient, we need confirmation that they are not going to run a pending futex_wait call next to avoid the ABA issue
    - FUTEX_WAIT_REQUEUE_PI is just requeueing, but not preventing pending old futex_wait calls