redhat.

Updating glibc concurrency

Torvald Riegel

2015/08/09

Why update? How?

- Concurrency = code involving synchronization with other threads

- Several deficiencies in glibc's concurrent code
  - Custom, imprecisely and insufficiently defined memory model (e.g., atomic operations)
  - Bugs in (low-level) synchronization (e.g., generic code that is not portable)
  - General lack of code documentation
  - Some POSIX requirements not implemented
  - Many unnecessary arch-specific variants of concurrent data structures
- Improve by:
  - Use C11 memory model
  - Review and document existing concurrent code
  - Revise concurrent NPTL data structures

Agenda

- Basics: Memory models, atomic operations
- Improving low-level synchronization
- Pthreads changes: Mutex destruction, new semaphores, …
- POSIX issues
- Future work and how you can help

- More background: My C/C++ concurrency tutorial today at 3pm

Memory models

- Define semantics of accesses to memory by one or more threads

- Hardware-level memory models
  - Arch-specific, reasoning at HW instruction level
  - Model effects of caches and cache coherency protocols

- Programming-language-level memory models
  - Define basic abstractions (e.g., what is a “memory location”?)
  - Capture effects of HW memory models (portably!)
  - Model effects of compiler optimizations (!!!)
  - C/C++: At level of abstract machine
    Thus, at higher level of abstraction and at higher layer of implementation

Our current “model”

- Custom set of atomic operations (*atomics*) and semantics
  - All archs provide mostly same set of operations
  - Semantics of atomics are not documented (e.g., ordering constraints)
    - Actual semantics differ between archs in some cases
  - Plenty of use of plain memory accesses to concurrently accessed/modified memory
  - Mix between acquire/release semantics and reordering barriers (e.g., load/store barriers)

- Compiler interaction?
  - Keep fingers crossed … (plain memory accesses)
  - Enforce a HW load here and there (atomic_forced_read)

C11 memory model

- Why
  - The only non-hand-wavy specification of behavior of multi-threaded C programs (that is also not too simplistic)
  - Portability and no performance loss for our existing code
  - Make glibc concurrency easier for new developers

- Our approach: C11 memory model without using C11
  - Atomic operations with very similar function names to C11 atomics
    - Same semantics!
  - No explicitly atomic types, but use atomics for all concurrent accesses
    - No data races! (i.e., concurrent accesses to same memory location, at least one a write, at least one not atomic)
  - Otherwise, the same model!
    - Same reasoning, terminology, ...

How to transition to new atomics and memory model

- Incremental steps:
  - One logically isolated piece of synchronization at a time
  - But always use new atomics for new pieces of code, or when changing concurrent code
- Within each step,
  - Understand the high-level synchronization scheme
  - Ensure absence of data races (also in code using locks!)
  - Change to new C11-like atomics where necessary (e.g., existing atomics, or to prevent data races)
  - Review synchronization for correctness based on C11 model
  - Document in the code how and why it works
- Some more details: https://sourceware.org/glibc/wiki/Concurrency

Example: pthread_once

- Generic code not portable (lacked memory barriers)
  - Made code data-race-free
  - Reviewed, fixed, and documented synchronization
- Custom x86 implementation for performance:
  Added C fast-path function to help compiler
- Result is a single portable pthread_once implementation

```
+   /* We need acquire memory order for this load because if the value
+      signals that initialization has finished, we need to see any
+      data modifications done during initialization.  */
-   val = *once_control;
+   val = atomic_load_acquire (once_control);
    /* Check if the initialization has already been done.  */
    if (__glibc_likely ((val & __PTHREAD_ONCE_DONE) != 0))
      return 0;
```

nptl and POSIX

Mutex destruction: Disagreement about requirements on programs

- glibc: all other threads must have returned from mutex functions before mutex is destroyed
  - Need other synchronization to enforce this happens-before (e.g., pthread_join, semaphores, …)
- C++11: no other thread must be blocked on mutex when mutex is destroyed
  - Can use mutex itself to satisfy that condition
  - Example:
    - Thread A acquires mutex, starts thread B, releases mutex
    - Thread B acquires mutex, releases it, destroys it
- POSIX:
  - Non-normative example like C++11, but normative text unclear
  - Austin Group clarified that it wants C++11-like behavior

Mutex destruction: How it affects nptl

- No accesses to mutex data structure after ownership “hand-off”
  - Hand-off is reset of mutex to Not-Acquired state in shared memory
  - Identify all hand-off points and make hand-off a final atomic action
- Futex wake syscall must be allowed after hand-off action
  - Required by the design of futexes
    - No way around it without futex redesign or performance losses
    - Other work-arounds replicate futex functionality in userspace and/or don't work for process-shared mutexes
  - Result: futex wake can target memory location reused for something else
    - Spurious wake-ups possible on unrelated futexes
    - Futex wake can fail (e.g., targets unmappged memory)
  - Resolution: weasel-wording in the Linux futex docs that programs need to be prepared for it – seems not to be an issue in practice
- Similar issues in semaphores, barriers

Futex updates

- Futex specification not sufficiently precise
  - Unclear synchronization semantics (futexes are a mix of userspace and kernel synchronization)
  - Signal interruption semantics unclear
  - Error conditions underspecified
- manpages updated based on input from kernel and glibc
  - Branch draft_futex at git://git.kernel.org/pub/scm/docs/man-pages/man-pages.git
  - Thanks to Michael Kerrisk!
- New glibc-internal API with detailed docs: futex-internal.h
  - Transition away from old futex API provided by lowlevellock code
  - Status: most code moved to new API

Semaphore

- Similar issue as mutex destruction
  - Thread A calls sem_post, Thread B calls sem_wait and then sem_destroy
  - Old implementation: Separate counters for #tokens and #waiters (we want #waiters to decrease sem_post latency)
- New algorithms for archs with 64b and 32b atomic ops
  - 64b archs just puts both counters into one atomic var
  - 32b archs use one bit in #tokens for conservative estimate of whether #waiters > 0
- Removed arch-specific implementations (sparc is special…)
- Status: Committed

Barrier

- Same issue as mutex destruction: Accesses can happen after barrier has been legally destroyed
- New algorithm
- Avoid using a barrier-internal lock
- Status: Posted for review

Condition variable

- POSIX and C++11 unclear about exact ordering requirements between waiters and signals
  - Both clarified that they want a strict order: If cond_wait happens-before cond_signal, the latter must see and wake the former
- Old glibc implementation: More recent waiters can “steal” wake-up from older ones although only the older ones are eligible
- Solution: New algorithm
  - No simple solution because futexes don't give FIFO wake-up guarantees and because of process-shared
  - Have implemented a simpler algorithm and have a scheme for a more complex one
  - Simple algorithm can wake-up waiters spuriously more often
- Status: In testing, I'm working on a fix for one issue

Reader-writer lock

- Existing code works (after a fix) but is not scalable
  - Uses internal lowlevellock – costly compared to atomic increment/decrement of a #readers counter
  - Also, current lowlevellocks aren't highly scalable
- I'm working on a new algorithm
  - Same destruction requirements as for mutexes – makes this rather difficult

POSIX clarifications / influence

- Issues mentioned so far mostly resolved: (will be) updated in POSIX
- All POSIX Threads operations are full barriers?
  - E.g., mutex acquire / release is a full barrier?
  - Only makes a difference in weird (ab)uses of POSIX synchronization facilities
    - E.g., build atomics using sem_trywait and then do Dekker synchronization
- Use C11 memory model instead of simplistic and underspecified current one?

Future work

- Finish transition to new atomics
- Finish transition to C11 memory model
  - Make all of glibc data-race-free
  - Review existing old-style synchronization
- Spinning vs. blocking (in lots of synchronization code: mutexes, …)
- Execution agents: lighter-weight than POSIX threads
  - E.g., centralized thread pools for parallelism

What you can do

- Help with the transition
  - Pick a somewhat independent piece of synchronization
  - Transform code, following the new guidelines
  - Review synchronization for correctness, add documentation
  - Ask for review, discussion, or just help
  - No need to have a perfect patch!

- Help define how we communicate about concurrency
  - Review docs of concurrent code by others
  - Review guidelines
  - Speak up if something isn't clear to you