Concurrency Bugs in the Network Stack of NetBSD

Ryota Ozaki
ozaki-r@{iij.ad.jp,netbsd.org}

AsiaBSDCon 2018 BoF
Table of Contents

- Brief overview of NetBSD internals
- Examples of deadlocks
- Examples of race conditions
Brief Overview of NetBSD Internals

- Software Interrupts
- Synchronization primitives
- Lock primitives
Software Interrupts

- softint(9) -- machine-independent software interrupt framework
- A softint has thread (LWP) context
  - It can suspend and resume on sleep/block
- It can use synchronization primitives that implicitly sleep or block
  - Adaptive mutex(9), rwlock(9), etc.
- It can’t use synchronization primitives that explicitly sleep or block
  - condvar(9), kmem(9) with KM_SLEEP, etc.
Software Interrupts

- **Priority levels**
  - softclock < softbio < softnet < softserial

- **Dedicated LWPs**
  - LWPs of each priority level are created on each CPU
  - E.g., softnet/0 and softserial/1

- **API**
  - softint_establish -- register a softint with a priority level
  - softint_schedule -- schedule the registered handler
Software Interrupts

- **Dispatch points (fast softint)**
  - Immediate after (hardware) interrupt handler execution (fast softint)
  - Just before returning to the user mode
- **Dispatch order**
  - Higher priority level first
  - FIFO for each handler on a priority level
    - Handlers are listed
    - If one handler gets stuck, subsequence handlers never run
  - Normal LWPs are dispatched after all pending softints are done
callout(9)

- Timer -- execute a function after a specified length of time
- It is a softint handler (softclock)
- It runs expired timers one by one (FIFO)
  - If one handler gets stuck, subsequence handlers never run
Synchronization Primitives

- condvar(9)
- xcall(9)
condvar(9)

- Condition variable, condvar, cv
- API
  - cv_wait(cv, mtx): sleep on the cv until someone wakes up
  - cv_broadcast(cv): wake up LWPs sleeping on the cv
  - Both APIs need to be called with holding a mutex to avoid race conditions
xcall(9)

- Machine-independent cross call interface
- A user can run an arbitrary function on each CPU
- Typical usage
  - `xc_wait(xc_broadcast(XC_HIGHPRI, func, arg1, arg2))`
    - XC_HIGHPRI uses softints to run a callback
    - xc_wait waits for completions of all callbacks
- Note that xcall processes just one request at a time
  - Subsequent requests need to wait for the completion of a running request
Lock Primitives

- mutex(9)
- rwlock(9)
- pserialize(9)
- psref(9)
- localcount(9)
mutex(9)

- Two types
  - Spin -- busy wait
  - Adaptive -- busy wait & sleep
- Recursive acquisition is not supported
- softnet_lock
  - An adaptive mutex for the network stack
  - It used to be used to protect the network stack instead of KERNEL_LOCK
  - softint/callout handlers typically tries to hold it at the beginning of their handlers
rwlock(9)

- Usual readers-writer lock
pserialize(9)

- Provide a facility to wait for an object to be released by any LWPs
  - Like Linux (classic) RCU

- API for readers: pserialize_read_{enter,exit}
  - An object acquired inside a read section is guaranteed to be not destroyed inside the section

- API for writers: pserialize_perform
  - Wait for an object to be released any LWPs
pserialize(9) -- Typical Usage

● Reader
  ○ \texttt{s = pserialize\_read\_enter()};
  ○ \texttt{PSLIST\_FOREACH(item, ...)} {
  ○   \texttt{if (match(item))} {
  ○     \texttt{// do something useful}
  ○   }
  ○ }
  ○ \texttt{pserialize\_read\_exit(s)};

● Constraints for readers
  ○ Must not sleep/block inside the section

● Writer
  ○ \texttt{mutex\_enter(mtx)};
  ○ \texttt{PSLIST\_REMOVE(item, ...)};
  ○ \texttt{pserialize\_perform()};
  ○ \texttt{mutex\_exit(mtx)};
  ○ \texttt{// destroy the item}

● Constraints for writers
  ○ Removing an item needs to be serialized
  ○ \texttt{pserialize\_perform} also needs to be serialized
pserialize(9)

- pserialize_perform
  - Very slow
    - Waits until context switches take place on each CPU three times
psref(9)

- Passive reference
- Allow to acquire/release a reference of an object cheaply
  - No atomic operations involved
- Can hold a reference over sleeps/blocks unlike pserialize(9)
  - LWP migrations between CPUs are not allowed
- Waiting for reference releases is quite heavy like pserialize(9)
psref(9)

- API readers: `psref_{acquire,release}`
- API writers: `psref_target_destroy`
  - Wait until all references to a target object have been released
  - Use `xcall(9)` to check references on each CPU
  - Very slow
- Another API: `curlwp_bind` and `curlwp_bindx`
  - It suppresses the current LWP from being migrated between CPUs
  - Needed for uses of psref in normal LWP contexts
localcount(9)

- Reference counting without atomic operations
- Have per-CPU counters on a target object
  - The data size increases as per the number of CPUs
- Allow holding a reference over sleeps/blocks and LWP migrations
- API for readers: localcount_{acquire,release}
- API for writers: localcount_drain
  - Wait until all references to a target object have been released
  - Use condvar(9)
  - Very slow
Examples of Deadlocks

- `pserialize_perform` and `callout`
- `localcount_drain` and `pserialize_perform`
pserialize_perform and callout

- If `pserialize_perform` is called with holding a mutex that can be held in callout handlers, a deadlock can occur.
- Resource dependency graph:
  - `softnet_lock` \(\Rightarrow\) `pserialize_perform` \(\Rightarrow\) `kpause` \(\Rightarrow\) `callout` \(\Rightarrow\) `softnet_lock`
pserialize_perform and callout

- The check instructions of pserialize_perform
  - do {
    xc_wait(xc_broadcast(XC_HIGHPRI, nullop, ...));
    kpause(...);
  } while (!finished());
- kpause sleeps a specified period by using callout(9)
- If a callout handler takes a mutex that is held by an LWP that executes pserialize_perform, kpause never finish
localcount and pserialize_perform

- Resource dependency graph
  - localcount_drain => xc => mtx => pserialize_perform => xc

- A code snippet that causes a deadlock
  - `mutex_enter(&mtx);`
  - `PSLIST_REMOVE(item, ...);`
  - `pserialize_perform(psz);`
  - `localcount_drain(&item->localcount, &cv, &mtx);`
  - `mutex_exit(&mtx);`
localcount and pserialize_perform

- A code snippet that causes a deadlock
  - `mutex_enter(&mtx);`
  - `PSLIST_REMOVE(item, ...);`
  - `pserialize_perform(psz);`
  - `localcount_drain(&item->localcount, &cv, &mtx);`
  - `mutex_exit(&mtx);`

- Explanation
  - Thread A: calls `localcount_drain`, it releases temporarily mtx then calls a xcall
localcount and pserialize_perform

- A code snippet that causes a deadlock
  - `mutex_enter(&mtx);`
  - `PSLIST_REMOVE(item, ...);`
  - `pserialize_perform(psz);`
  - `localcount_drain(&item->localcount, &cv, &mtx);`
  - `mutex_exit(&mtx);`

- Explanation
  - Thread B: calls `pserialize_perform` with holding `mtx` but `pserialize_perform` gets stuck on `xcall` that is used by `localcount_drain` of Thread A
localcount and pserialize_perform

- A code snippet that causes a deadlock
  - `mutex_enter(&mtx);`
  - `PSLIST_REMOVE(item, ...);`
  - `pserialize_perform(psz);`
  - `localcount_drain(&item->localcount, &cv, &mtx);`
  - `mutex_exit(&mtx);`

- Explanation
  - Thread C (xc_thread, a callback for localcount_drain): tries to take mtx but fails because it’s held by Thread B
localcount and pserialize_perform

- Resource dependency graph
  - localcount_drain (A) => xc (C) => mtx (B) => pserialize_perform
    (B) => xc
Examples of Race Conditions

- A xcall bug
- curlwp_bind and LWP migration
- Reference leaks on callout_reset
A xcall Bug

- **Typical usage**
  - `xc_wait(xc_broadcast(XC_HIGHPRI, func, arg1, arg2))`
    - XC_HIGHPRI uses softints to run a callback
    - `xc_wait` waits for completions of all callbacks

- **xcall manages running callbacks and finished callbacks with two global counters: xc_headp and xc_donep**
  - When one request is accepted, `xc_headp += N` where N is the number of CPUs
  - When one callback finishes, `xc_donep++`
  - Once `xc_donep == xc_headp`, the request is competed
A xcall Bug

● The bug
  ○ xc_donep++ was done before executing a callback

● Impacts
  ○ xc_wait can return before the last request has been done
  ○ A subsequent request can be accepted

● Solution
  ○ xc_donep++ after executing a callback
curlwp_bind and LWP migration

- **curlwp_bind and psref**
  - `bound = curlwp_bind();`
  - `psref_acquire(...);`
  - `psref_release(...);`
  - `curlwp_bindx(bound);`

- **psref_release** has an assertion that checks whether a current LWP hadn't migrated

- But the assertion rarely failed for some reason...

- **curlwp_bind** couldn’t surely prevent migrations

- What happened?
curlwp_bind and LWP migration (Explanation)

- curlwp_bind just sets the LP_BOUND flag to the current LWP
- The flag suppresses a migration
- A migration takes place on a context switch *if scheduled*
- The scheduler load-balances LWPs between CPUs
  - It forces to migrate a hogging LWP to another CPU
  - It periodically checks all LWPs in a kthread, schedules migrations
  - It checks LP_BOUND and skips LWPs with the flags
- A context switch (mi_switch) *doesn’t check* the flag
curlwp_bind and LWP migration (Explanation)

- Thread A: is running on one CPU
- Scheduler: does load balancing on another CPU
  - And schedule Thread A to be migrated
- Thread A: calls curlwp_bind and psref_acquire
- Thread A: is preempted and is migrated to another CPU
- Thread A: is dispatched again and calls psref_release
  - psref_release notices the migration and boom!
curlwp_bind and LWP migration (Explanation)

- **Solution**
  - Check the flag in mi_siwtch too
That's it