An insane idea on reference counting

Gleb Smirnoff
glebius@FreeBSD.org

BSDCan 2014
Ottawa

May 16, 2014

The configuration problem

Problem:

- A structure in memory describing a “configuration”
- Multiple readers at high rate
- Sporadic writers

The configuration problem

Problem:

- A structure in memory describing a “configuration”
- Multiple readers at high rate
- Sporadic writers

Examples:

- local IP addresses hash
- interfaces list, particular ifnet
- firewall rules

Ready to use solutions

What we use in FreeBSD:

Ready to use solutions

What we use in FreeBSD:

- rwlock(9)
  - Acquiring thread == Releasing thread
  - Very expensive: all readers do atomic(9) on the same word

Ready to use solutions

What we use in FreeBSD:

- rwlock(9)
  - Acquiring thread == Releasing thread
  - Very expensive: all readers do atomic(9) on the same word
- rmlock(9)
  - Acquiring thread == Releasing thread
  - Does sched_pin(9) for the entire operation

Ready to use solutions

What we use in FreeBSD:

- rwlock(9)
  - Acquiring thread == Releasing thread
  - Very expensive: all readers do atomic(9) on the same word
- rmlock(9)
  - Acquiring thread == Releasing thread
  - Does sched_pin(9) for the entire operation
- refcount(9)
  - Acquiring thread != Releasing thread
  - Expensive: is atomic(9)

Patented solution

- RCU
  - Acquiring thread == Releasing thread
  - Patented :(

Goals

- Ultra lightweight for a reader
- Acquiring thread != Releasing thread

Goals

- Ultra lightweight for a reader
- Acquiring thread != Releasing thread

Sounds like refcount(9) w/o atomics.

counter(9) as refcount?

counter(9) - new facility in FreeBSD 10

```
counter_u64_t cnt;

cnt = counter_u64_alloc(M_WAITOK);
counter_u64_add(cnt, 1);
```

- lightweight, due to per-CPU memory
- counter_u64_add is single instruction on amd64

Suggested API

```
struct lwref {
  void *ptr;
  counter_u64_t cnt;
};

typedef struct lwref * lwref_t;
```

Suggested API

```
void *lwref_acquire(lwref_t lwr, counter_u64_t *c);
```

- Returns the “configuration” pointer from lwr
- Increments counter(9) in lwr
- Returns the counter(9)

Suggested API

```
void lwref_change(lwref_t lwr, void *newptr,
    void (*freefn)(void *, void *), void *freearg);
```

- Changes the “configuration” pointer in lwr to *newptr*
- Allocates new counter(9) for the lwr
- Asynchronously frees old pointer and old counter(9) when it is safe

Suggested API

- *lwref_acquire* must be safe against *lwref_change*
- *lwref_acquire* must not be expensive
- *lwref_change* is allowed to be expensive

naive racy *lwref_acquire*

```
void *
lwref_acquire(lwref_t lwr, counter_u64_t *cp)
{
  void *ptr;

  ptr = lwr->ptr;
  cp = &lwr->cnt;

  counter_u64_add(*cp, 1);

  return (ptr);
}
```

Hypothetical *lwref_change* operation

- Update contents of lwref_t on all CPUs
- Check if *lwref_acquire* is running on any CPU

Hypothetical *lwref_change* operation

- Update contents of lwref_t on all CPUs
- Check if *lwref_acquire* is running on any CPU
-
    - How check that?
    - And what if it is running?

lwref_change is SMP rendezvous

```
void
lwref_change(lwref_t lwr, void *newptr,
    void (*freefn)(void *, void *), void *freearg)
{
    struct lwref_change_ctx ctx;

    ctx->lwr = lwr;
    ctx->newptr = newptr;
    ctx->newcnt = counter_u64_alloc();

    smp_rendezvous(lwref_change_action, &ctx);
}
```

lwref_change_action code

```
void
lwref_change_action(void *v)
{
    struct lwref_change_ctx *ctx = v;
    lwref_t lwr = ctx->lwr;

    lwr->ptr = ctx->newptr;
    lwr->refcnt = ctx->newcnt;

    /*
     * Check if we interrupted lwref_acquire().
     */

    ...
}
```

interruption possibilities

- The rendezvous IPI interrupted *lwref_acquire*

interruption possibilities

- The rendezvous IPI interrupted *lwref_acquire*
- Any other interrupt (usually timer) interrupted *lwref_acquire* and the thread went on scheduler's run queue, prior to *lwref_change* execution

restartable *lwref_acquire*

```
ENTRY(lwref_acquire)
        mov     (%rdi), %rax
        mov     0x8(%rdi), %rcx
        mov     %rcx, (%rsi)
        mov     $__pcpu, %rdx
        sub     %rdx, %rcx
        addq    $1, %gs:(%rcx)
        ret
END(lwref_acquire)
```

restartable *lwref_acquire*

```
ENTRY(lwref_acquire)
        mov     (%rdi), %rax
        mov     0x8(%rdi), %rcx
        mov     %rcx, (%rsi)
        mov     $__pcpu, %rdx
        sub     %rdx, %rcx
        addq    $1, %gs:(%rcx)
.globl  lwref_acquire_ponr
lwref_acquire_ponr:
        ret
END(lwref_acquire)
```

When restart?

- Option 1: whenever any interrupt interrupts *lwref_acquire*
- Option 2: whenever *lwref_change* interrupts *lwref_acquire*

Option 1: Any interrupt rolls back

The PUSH_FRAME() macro in amd64/include/asmacros.h should check and fix up %rip in pushed frame.

Option 1: Any interrupt rolls back

The PUSH_FRAME() macro in amd64/include/asmacros.h should check and fix up %rip in pushed frame.

- Pros: very simple
- Cons: extra instructions on every interrupt

change to PUSH_FRAME() macro

```
@@ -167,7 +167,14 @@
        movw    %es,TF_ES(%rsp) ;
        movw    %ds,TF_DS(%rsp) ;
        movl    $TF_HASSEGS,TF_FLAGS(%rsp) ;
-       cld
+       movq    TF_RIP(%rsp), %rax ;
+       cmpq    %rax, lwref_acquire ;
+       jb      2f ;
+       cmpq    %rax, lwref_acquire_ponr ;
+       jae     2f ;
+       movq    lwref_acquire, %rax ;
+       movq    %rax, TF_RIP(%rsp) ;
+2:     cld
```

Option 2: *lwref_change* rolls back

```
void
lwref_change_action(void *v)
{
    struct trapframe *tf;
    ...

    /*
     * Check if we interrupted lwref_acquire().
     */
    tf = (struct trapframe *)
      ((register_t *)__builtin_frame_address(1) + 2);
    lwref_fixup_rip(&tf->tf_rip);
}
```

lwref_fixup_rip

```
static void
lwref_fixup_rip(register_t *rip)
{

    if (*rip >= (register_t )lwref_acquire &&
        *rip < (register_t )lwref_acquire_ponr)
          *rip = (register_t )lwref_acquire;
}
```

What about scheduler run queues?

New function:

```
void sched_foreach_on_runq(void(*)(void *));
```

lwref_change rolls back (continued)

```
void
lwref_change_action(void *v)
{
    ...
    sched_foreach_on_runq(lwref_fixup_td);
}
```

naive *lwref_fixup_td*

```
static void
lwref_fixup_td(void *arg)
{
    struct thread *td = arg;

    tf = (struct trapframe *)
      ((register_t *)(***(void ****)(td->td_pcb->
         pcb_rbp)) + 2);

    lwref_fixup_rip(&tf->tf_rip);
}
```

```
static void
lwref_fixup_td(void *arg)
{
  struct thread *td = arg;
  struct trapframe *tf;
  register_t *rbp, rip;

  for (rbp = (register_t *)td->td_pcb->pcb_rbp;
       rbp && rbp < (register_t *)*rbp;
       rbp = (register_t *)*rbp) {

          rip = (register_t )*(rbp + 1);

          if (rip == (register_t )timerint_ret ||
              ...
              rip == (register_t )
                 ipi_intr_bitmap_handler_ret) {
                  tf = (struct trapframe *)(rbp + 2);
                  lwref_fixup_rip(&tf->tf_rip);
          }
  }
```

hint from jhb@

Use td->td_frame to get access to frame :)

Questions?