How to hold onto things in a multiprocessor world

Taylor 'Riastradh' Campbell
campbell@mumble.net
riastradh@NetBSD.org

AsiaBSDcon 2017
Tokyo, Japan
March 12, 2017

Slides'n'code

- Full of code! Please browse at your own pace.
- Slides: `https://tinyurl.com/ho2cdhq`[1]
- Paper: `https://tinyurl.com/h9kqccf`[2]

Slides: Paper:

[1] `https://www.NetBSD.org/gallery/presentations/riastradh/asiabsdcon2017/mp-refs-slides.pdf`

[2] `https://www.NetBSD.org/gallery/presentations/riastradh/asiabsdcon2017/mp-refs-paper.pdf`

Resources

- Network routes
  - May be tens of thousands in system.
  - Acquired and released by packet-processing path.
  - Same route may be used simultaneously by many flows.
  - Large legacy code base to update for parallelism.
  - Update must be incremental!
- Device drivers
  - Only a few dozen in system.
  - Even wider range of legacy code to safely parallelize.
- File system objects ('vnodes')
- User credential sets
- ...

The life and times of a resource

- Birth:
  - Create: allocate memory, initialize it.
  - Publish: reveal to all threads.
- Life:
  - Acquire: thread begins to use a resource.
  - Release: thread is done using a resource.
  - ... rinse, repeat.
  - Concurrently by many threads at a time.
- Death:
  - Delete: prevent threads from acquiring.
  - Destroy: free memory...

The life and times of a resource

- Birth:
  - Create: allocate memory, initialize it.
  - Publish: reveal to all threads.
- Life:
  - Acquire: thread begins to use a resource.
  - Release: thread is done using a resource.
  - . . . rinse, repeat.
  - Concurrently by many threads at a time.
- Death:
  - Delete: prevent threads from acquiring.
  - Destroy: free memory. . . *after* all threads have released.

Problems for an implementer

If you are building an API for some class of resources. . .

- You MUST ensure nobody frees memory still in use!
- You MUST satisfy other API contracts, e.g. mutex rules.
- You MAY want to allow concurrent users of resources.
- You MAY care about performance.

Serialize all resources — layout

```
struct foo {
        int key;
        ...;
        struct foo *next;
};

struct {
        kmutex_t lock;
        struct foo *first;
} footab;
```

Serialize all resources — create/publish

```
struct foo *f = alloc_foo(key);

mutex_enter(&footab.lock);
f->next = footab.first;
footab.first = f;
mutex_exit(&footab.lock);
```

Serialize all resources — lookup/use

```
struct foo *f;

mutex_enter(&footab.lock);
for (f = footab.first; f != NULL; f = f->next) {
        if (f->key == key) {
                ...use f...
                break;
        }
}
mutex_exit(&footab.lock);
```

Serialize all resources — delete/destroy

Delete/destroy:

```
struct foo **fp, *f;

mutex_enter(&footab.lock);
for (fp = &footab.first; (f = *fp) != NULL; fp = &f->next) {
        if (f->key == key) {
                *fp = f->next;
                break;
        }
}
mutex_exit(&footab.lock);

if (f != NULL)
        free_foo(f);
```

Serialize all resources — slow and broken!

- No parallelism.
- Not allowed to wait for I/O or do long computation under mutex lock.
- (This is a NetBSD rule to put bounds on progress for `mutex_enter`, which is not interruptible.)

Mutex and reference counts — layout

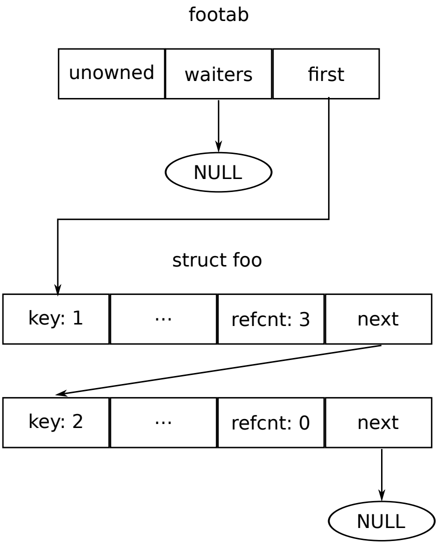
(a) Add reference count to each object.

(b) Add condition variable for notifying `f->refcnt == 0`.

```
struct foo {
        int key;
        ...;
        unsigned refcnt;         // (a)
        struct foo *next;
};

struct {
        kmutex_t lock;
        kcondvar_t cv;           // (b)
        struct foo *first;
} footab;
```

Mutex and reference counts — layout



Mutex and reference counts — create/publish

```
struct foo *f = alloc_foo(key);

f->refcnt = 0;

mutex_enter(&footab.lock);
f->next = footab.first;
footab.first = f;
mutex_exit(&footab.lock);
```

Mutex and reference counts — lookup/acquire

```
struct foo *f;

mutex_enter(&footab.lock);
for (f = footab.first; f != NULL; f = f->next) {
        if (f->key == key) {
                f->refcnt++;
                break;
        }
}
mutex_exit(&footab.lock);
if (f != NULL)
        ...use f...
```

Mutex and reference counts — release

```
mutex_enter(&footab.lock);
if (--f->refcnt == 0)
        cv_broadcast(&footab.cv);
mutex_exit(&footab.lock);
```

Mutex and reference counts — delete/destroy

```
struct foo **fp, *f;

mutex_enter(&footab.lock);
for (fp = &footab.first; (f = *fp) != NULL; fp = &f->next) {
        if (f->key == key) {
                *fp = f->next;
                while (f->refcnt != 0)
                        cv_wait(&footab.cv, &footab.lock);
                break;
        }
}
mutex_exit(&footab.lock);

if (f != NULL)
        free_foo(f);
```

Mutex lock and reference counts — summary

- If this works for you, stop here!
- Easy to prove correct.
- Just go to another talk.
- . . . but it does have problems:
- Only one lookup at any time.
- **Contention over lock for every object.**
- Hence not scalable to many CPUs.

Hashed locks

- Randomly partition resources into buckets.
- If distribution on resource use is uniform, lower contention for lookup!

Hashed locks — layout

```
struct {
        struct foobucket {
                kmutex_t lock;
                kcondvar_t cv;
                struct foo *first;
        } b;
        char pad[roundup(
            sizeof(struct foobucket),
            CACHELINE_SIZE)];
} footab[NBUCKET];
```

Hashed locks — acquire

```
size_t h = hash(key);

mutex_enter(&footab[h].b.lock);
for (f = footab[h].b.first; f != NULL; f = f->next) {
        if (f->key == key) {
                f->refcnt++;
                break;
        }
}
mutex_exit(&footab[h].b.lock);
```

Hashed locks

- Randomly partition resources into buckets.
- **If distribution on resource use is uniform,** lower contention for lookup!
- What if many threads want to look up same object?
- Still only one lookup at a time for that object.
- **Still contention for releasing resources after use.**

Mutex lock and *atomic* reference counts

- Use atomic operations to manage most uses of a resource.
- No need to acquire global table lock to release a resource if it's not the last one.

Mutex lock and *atomic* reference counts — acquire

```
struct foo *f;

mutex_enter(&footab.lock);
for (f = footab.first; f != NULL; f = f->next) {
        if (f->key == key) {
                atomic_inc_uint(&f->refcnt);
                break;
        }
}
mutex_exit(&footab.lock);
if (f != NULL)
        ...use f...
```

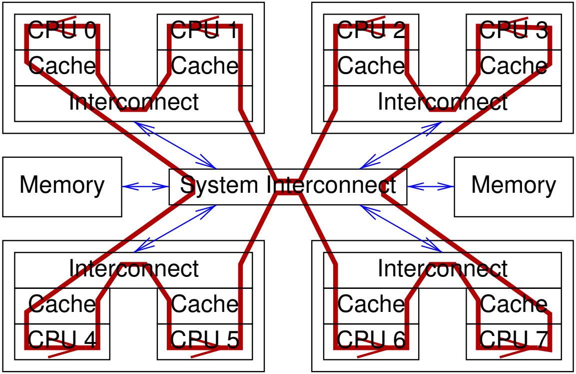
Mutex lock and *atomic* reference counts — release

```
do {
        old = f->refcnt;
        if (old == 1) {
                mutex_enter(&footab.lock);
                if (f->refcnt == 1) {
                        f->refcnt = 0;
                        cv_broadcast(&footab.cv);
                } else {
                        atomic_dec_uint(&f->refcnt);
                }
                mutex_exit(&footab.lock);
                break;
        }
} while (atomic_cas_uint(&f->refcnt, old, new) != old);
```

Atomics: still not scalable

- We avoid contention over global table lock to release.
- But if many threads want to use the same foo...
- Atomic operations are not a magic bullet!
- Single atomic is slightly faster and uses less memory than a mutex lock enter/exit.
- But contended atomics are just as bad as contended locks!

Atomics: interprocessor synchronization[3]



[3] Diagram Copyright © 2005–2010, Paul E. McKenney. From Paul E. McKenney, *Is Parallel Programming Hard, And, If So, What Can You Do About It?*, 2011. `https://www.kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.2011.01.02a.pdf`

Reader/writer locks for lookup

- Instead of mutex lock for table, use rwlock.
- At any time, either one writer or many readers.
- Allows concurrent lookups, not just concurrent resource use.
- If lookups are slow, great!
- If lookups are fast, reader count is just another reference count managed with atomics—contention!

Basic problem: to read, we must write!

- All approaches here require *readers* to coordinate *writes*.
  - Acquire table lock: *write* who owns it now.
  - Acquire read lock: *write* how many readers.
  - Acquire reference count: *write* how many users.
- Can we avoid writes to read?
- Are there more reads than creations or destructions?
- Can we make reads cheaper, perhaps at the cost of making creation or destruction more expensive?

No-contention references in NetBSD

- Passive serialization.
  - Like read–copy–update, RCU in Linux.
  - ...but US patent expired sooner!
- Passive references.
  - Similar to hazard pointers.
  - Similar to OpenBSD SRP.
- Local counts—per-CPU reference counts.
  - Similar to sleepable RCU, SRCU, but simpler.

Coordinate *publish* and read

- Linked-list insert and read can coordinate with *no* atomics.
- …as long as they write and read in the correct order.
- One writer, any number of readers!
- Same principle for hash tables (‘hashed lists’), radix trees.

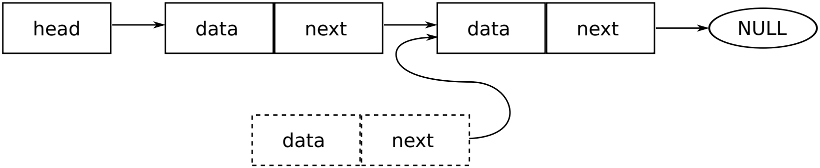
Publish

- Write data first.
- Then write pointer to it.

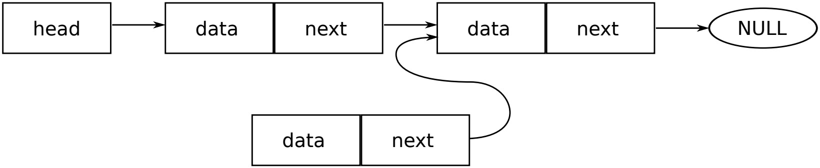
```
struct foo *f = alloc_foo(key);

mutex_enter(&footab.lock);
f->next = footab.first;
membar_producer();
footab.first = f;
mutex_exit(&footab.lock);
```

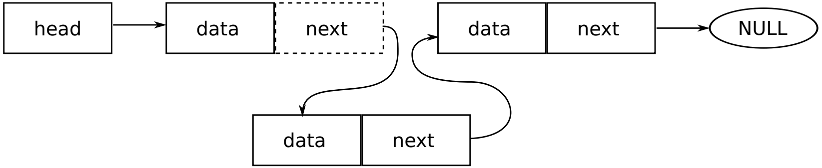
Publish 1: after writing data



Publish 2: after write barrier



Publish 3: after writing pointer

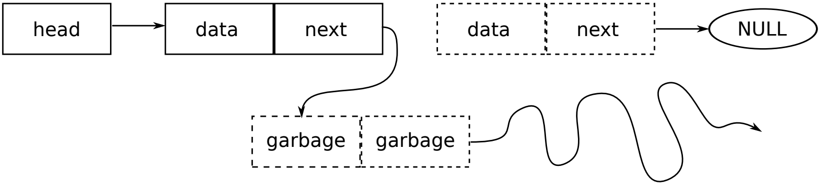


Read

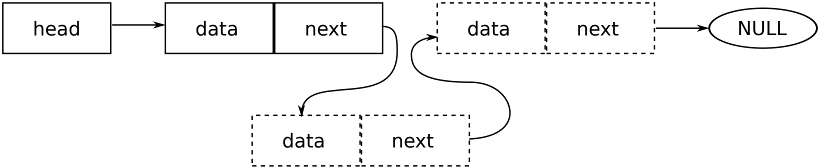
- Read pointer first.
- Then read data from it.
- . . . Yes, in principle stale data could be cached.
- Fortunately, `membar_datadep_consumer` is a no-op on all CPUs other than DEC Alpha.

```
for (f = footab.first; f != NULL; f = f->next) {
        membar_datadep_consumer();
        if (f->key == key) {
                use(f);
                break;
        }
}
```

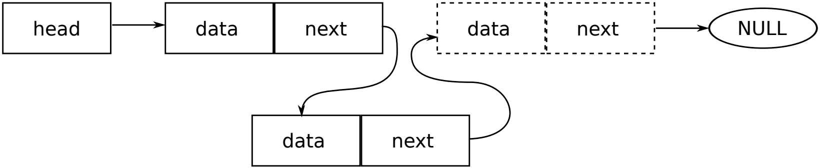
Read 1: after reading pointer



Read 2: after read barrier



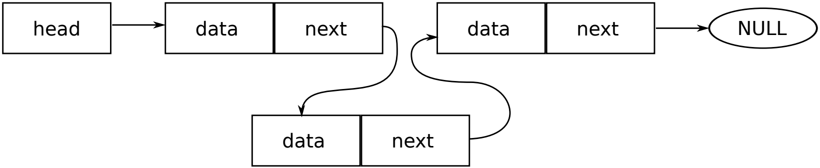
Read 3: after reading data



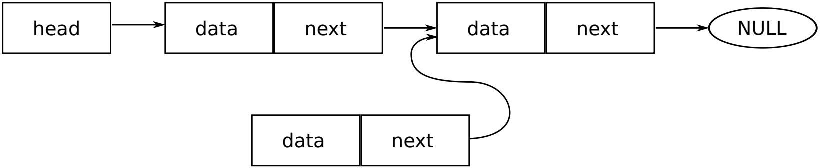
Delete

- Deletion is even easier!
- `*fp = f->next;`
- . . . but there is a catch.

Delete 1: before delete



Delete 1: after delete



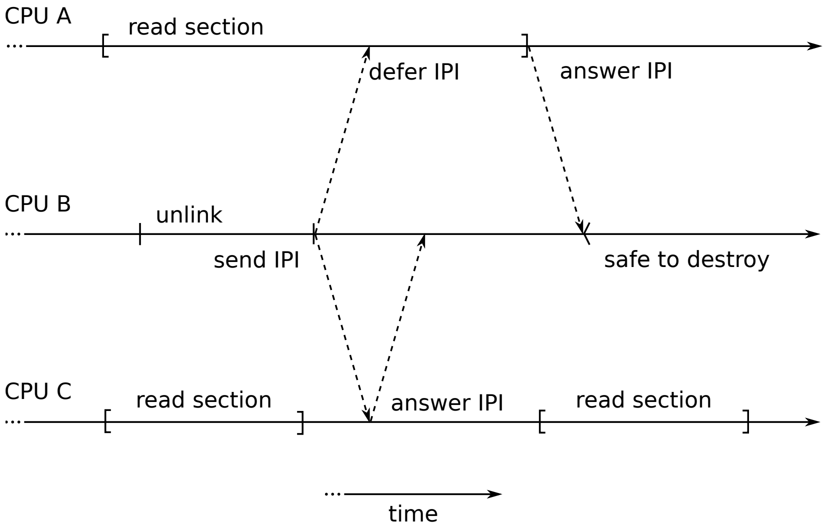
The catch

- All well and good for publish and use!
- All well and good for *delete*!
- But when can we *destroy* (free memory, *etc.*)?
- No signal for when all users are done with a resource.
- How to signal release without contention?

Passive serialization: `pserialize(9)`

- Lookup/use:
  1. Acquire: Block interrupts on CPU.
  2. Look up resource.
  3. Use it.
  4. Release: Restore and process queued interrupts on CPU.
  5. (Cannot use resource any more after this point!)
- Delete/destroy:
  1. Remove resource from list: `*fp = f->next`.
  2. Send *interprocessor interrupt* to all CPUs.
  3. Wait for it to return on all CPUs.
  4. All users that *could have seen* this resource have exited.

Passive serialization



Passive serialization — lookup/use

1. Acquire: Block interrupts with `pserialize_read_enter`.
2. Lookup: Read pointer.
3. Memory barrier!
4. Use: Read data.
5. Release: Restore and process queued interrupts with `pserialize_read_exit`.

```
s = pserialize_read_enter();
for (f = footab.first; f != NULL; f = f->next) {
        membar_datadep_consumer();
        if (f->key == key) {
                use(f);
                break;
        }
}
pserialize_read_exit(s);
```

Passive serialization — delete/destroy

(a) Delete from list to prevent new users.

(b) Send IPI to wait for existing users to drain.

(c) Free memory.

```
mutex_enter(&footab.lock);
for (fp = &footab.first; (f = *fp) != NULL; f = f->next) {
    if (f->key == key) {
        /* (a) Prevent new users. */
        *fp = f->next;
        /* (b) Wait for old users. */
        pserialize_perform(footab.psz);
    }
}
mutex_exit(&footab.lock);

if (f != NULL)
    /* (c) Destroy. */
    free_foo(f);
```

Passive serialization — lists

- `sys/queue.h` macros *do not* have correct memory barriers.
- So we provide `PSLIST(9)`, like `LIST` in `sys/queue.h`.
- Linked list with constant-time insert and delete...
- ...and correct memory barrier for insert and read.

Passive serialization — PSLIST(9)

```
struct foo { ... struct pslist_entry f_entry; ... };
struct { ... struct pslist_head head; ... } footab;

mutex_enter(&footab.lock);
PSLIST_WRITER_INSERT_HEAD(&footab.head, f, f_entry);
mutex_exit(&footab.lock);

s = pserialize_read_enter();
PSLIST_READER_FOREACH(f, &footab.head, struct foo,
    f_entry) {
        if (f->key == key) {
                ...use f...;
                break;
        }
}
pserialize_read_exit(s);
```

Passive serialization pros

- Zero contention!
- Serially fast readers!
  - We use *software* interrupts, so cheap to block and restore.
  - No hardware interrupt controller reconfiguration!
- Constant memory overhead—no memory per resource, per CPU!

Passive serialization cons

- Interrupts *must be blocked* during read.
- Thread *cannot sleep* during read.
- What if we want to pserialize the network stack?
- Code was written in '80s before parallelism mattered. . .
- . . . and does memory allocation in packet path (*e.g.*, to prepend a header in a tunnel). . .
- . . . and simultaneously re-engineering the whole network stack is hard!
- Can we do it incrementally with different tradeoffs?

Passive references: `psref(9)`

- Record per-CPU *list* of all resources in use.
- Lookup: use pserialize for table lookup.
- To acquire resource: put it on the list.
- Can now do anything on the CPU—sleep, eat, watch television. . .
- To release resource: remove it from the list.
- To wait for users: send IPI to *check for resource* on each CPU's list.
- Note: Reader threads must not switch CPUs!

Passive references — create/publish

```
struct foo { ... struct psref_target target; ... };
struct { ... struct psref_class *psr; ... } footab;

struct foo *f = alloc_foo(key);

psref_target_init(&f->target, footab.psr);

mutex_enter(&footab.lock);
PSLIST_WRITER_INSERT_HEAD(&footab.head, f_entry, f);
mutex_exit(&footab.lock);
```

Passive references — lookup/acquire

psref_acquire inserts entry on CPU-local list: no atomics!

```
struct psref fref;
int bound, s;

/* Bind to current CPU and lookup. */
bound = curlwp_bind();
s = pserialize_read_enter();
PSLIST_READER_FOREACH(f, &footab.head, struct foo,
    f_entry) {
        if (f->key == key) {
                psref_acquire(&fref, &f->target,
                    footab.psr);
                break;
        }
}
pserialize_read_exit(s);
```

Passive references — release

- `psref_remove` removes entry on CPU-local list, and notifies destroyer if there is one.
- No atomics *unless* another thread is waiting to destroy the resource.

```
/* Release psref and unbind from CPU. */
psref_release(&fref, &f->target, footab.psr);
curlwp_bindx(bound);
```

Passive references — delete/destroy

- `psref_target_destroy` marks the resource as being destroyed.
- Thus, future `psref_release` will wake it.
- Then `psref_target_destroy` repeatedly checks for references on all CPUs and sleeps until there are none left.

Passive references — delete/destroy

```
/* (a) Prevent new users. */
mutex_enter(&footab.lock);
PSLIST_WRITER_FOREACH(f, &footab.head, struct foo,
    f_entry) {
        if (f->key == key) {
                PSLIST_WRITER_REMOVE(f, f_entry);
                pserialize_perform(footab.psz);
                break;
        }
}
mutex_exit(&footab.lock);
if (f != NULL) {
        /* (b) Wait for old users.  */
        psref_target_destroy(&f->target, footab.psr);
        /* (c) Destroy. */
        free_foo(f);
}
```

Passive references — notes

- Threads can sleep while holding passive references.
- Binding to CPU is not usually a problem.
- Much of network stack already runs bound to a CPU anyway!
- Bonus: can write precise asserts for diagnostics!

```
KASSERT(psref_held(&f->target, footab.psr));
```

- Modest memory cost:
  $O(\#CPU) + O(\#resource) + O(\#references)$.
- Network routes: tens of thousands in system.
- Network routes: a handful per CPU at any time.

Local counts: `localcount(9)`

- Global reference count per resource $\implies$ contention.
- What about a per-CPU reference count per resource?
- High memory cost: $O(\#CPU \times \#resource)$.
- So use only for small numbers of resources, like device drivers.
- Device drivers: dozens in system.
- Device drivers: maybe thousands of *uses* at any time during heavy I/O loads.

Local counts — create/publish

```
struct foo { ... struct localcount lc; ... };

struct foo *f = alloc_foo(key);

localcount_init(&f->lc);

mutex_enter(&footab.lock);
PSLIST_WRITER_INSERT_HEAD(&footab.head, f_entry, f);
mutex_exit(&footab.lock);
```

Local counts — lookup/acquire

`localcount_acquire` increments a CPU-local counter—no atomics!

```
s = pserialize_read_enter();
PSLIST_READER_FOREACH(f, &footab.head, struct foo,
    f_entry) {
        if (f->key == key) {
                localcount_acquire(&f->lc);
                break;
        }
}
pserialize_read_exit(s);
```

Local counts — release

- `localcount_release` increments a CPU-local counter.
- If there is a destroyer, updates destroyer's global reference count.
- No atomics *unless* another thread is waiting to destroy the resource.

```
localcount_release(&f->lc);
```

Local counts — delete/destroy

- `localcount_destroy` marks resource as being destroyed.
- Sends IPI to compute global reference count by adding up each CPU's local reference count.
- (Fun fact: local reference counts can be negative, if threads have migrated!)
- Waits for all IPIs to return and reference count to become zero.

Local counts — delete/destroy

```
/* (a) Prevent new users. */
mutex_enter(&footab.lock);
PSLIST_WRITER_FOREACH(f, &footab.head, struct foo,
    f_entry) {
        if (f->key == key) {
                PSLIST_WRITER_REMOVE(f, f_entry);
                pserialize_perform(footab.psz);
                break;
        }
}
mutex_exit(&footab.lock);
if (f != NULL) {
        /* (b) Wait for old users.  */
        localcount_destroy(&f->lc);
        /* (c) Destroy. */
        free_foo(f);
}
```

Local counts — notes

- Not yet integrated in NetBSD—still on an experimental branch!
- To be used for MP-safely unloading device driver modules.
- Other applications? Probably yes!

Summary

- Avoid locks! Locks don't scale.
- Avoid atomics! Atomics don't scale.
- pserialize: short uninterruptible reads, fast but limited.
- psref: sleepable readers, modest time/memory cost, flexible.
- localcount: migratable readers, fast but memory-intensive.

Questions?

riastradh@NetBSD.org