Holding onto things in a multiprocessor world

Taylor R. Campbell
`riastradh@NetBSD.org`

January 13, 2017

Abstract

We present three mechanisms—passive serialization, passive references, and local counts—by which one thread can safely acquire, use, release references to resources such as hash table entries, network routes, and device drivers, while another thread may be creating new ones or trying to free, destroy, or unload existing ones.

Unlike mutex locks, reader/writer locks, atomic operations for reference counts, *etc.*, these mechanisms all scale in parallel to many cores.

We compare the different serial, parallel, and memory performance characteristics of the three mechanisms to more traditional approaches, and examine their impacts on API contracts and incremental development of a parallel network stack.

1 Introduction

The NetBSD kernel is a complex multiprocessor system that manages many kinds of hardware and software resources, such as network routes, device drivers, and cached file system objects that may be destroyed or freed from time to time when no longer in use.

Consider a packet arriving at a network interface. The NetBSD kernel's job is to decide where to route it and transmit it from the appropriate network interface. What does the kernel do when the packet arrives?

1. The kernel must find a route.
2. Another thread may try to delete the route, but must wait until all users of the route are complete.
3. The kernel may have to sleep in order to allocate memory for prepending header data to the packet, if it is destined to an encapsulated tunnel.
4. A userland process may attempt an ioctl on the NIC where the packet is destined.
5. Another userland process might try to unload the device driver for that NIC. That process must wait until all ioctls are complete *and* all routes destined to that NIC are purged.
6. After the packet has been forwarded, the kernel must notify the thread trying to delete the route that it is now safe to delete.
7. After the ioctl completes, the kernel must notify the process trying to unload the device driver that it is now safe to unload.

Different kinds of resources in the kernel have different usage patterns. For example:

Network routes There may be tens of thousands of of routes in the kernel at any given time, so the per-resource memory overhead should be moderate. The kernel may process on the order of hundreds of thousands or millions of packets per second. We are concerned first with scaling to many CPUs, since packet-processing is embarrassingly parallelizable, and secondarily with serial performance of the packet path.

Device drivers There will usually be only a few dozen device drivers in the kernel, so higher memory overhead is acceptable. However, applications may rely on the serial performance of each call to a device driver's entry point, which may not be parallelizable.

2 Acquisition and release

Before using a resource, a process, thread, or CPU—let's say a thread—must typically do something to **acquire** a reference to it, so that it is guaranteed not to be destroyed by another part of the system, until the reference is **released**, after which the thread may no longer use it.

A resource's lifetime can be broken up into three parts:

Create/publish Before a resource is made available for use, any data structures in memory for it must be initialized, and the initialization must be visible to all CPUs before the resource is published.

Lookup/use A resource must be looked up before it can be used—*e.g.*, finding a network route in the routing table given a destination address—and the lookup must acquire a reference for the caller. The caller may then use the resource until it releases the reference.

Remove/destroy When the kernel decides to destroy a resource—*e.g.*, when a userland process asks it to delete a route—it must

(a) first prevent any new users from acquiring the resource, *e.g.* by removing it from a table;

(b) then wait for all existing users to release the resource; and

(c) finally destroy the resource when it can guarantee nobody is trying to use it.

Often the resources are collected in a central table. For resources on which threads perform long I/O operations that are indexed in a table, threads may acquire references to the *table entries* and the *resources* in different ways, handing one off to the other.

3 Traditional approaches

Figure 1 shows a prototypical example of a collection of resources listed in a table, using a single global mutex lock for the table and all resources, and a reference count per resource to determine when a resource is no longer in use.

Publishing a newly created resource, Figure 2, is a matter of putting it in the table, excluding all other threads modifying the table. Destroying a resource, Figure 5, takes three stages as before: prevent new users, wait for existing users, and finally destroy the resource. To acquire a resource (Figure 3), a thread must

1. acquire a mutex lock to acquire a reference to all the *table entries*;
2. look up an entry for a resource of interest;
3. increment the resource's reference count to acquire a reference; and then
4. release the mutex on the table to release its reference to all the table entries.

At that point, the resource is guaranteed not to be destroyed. Then, when the thread is done with the resource, to release the resource (Figure 4) it must

1. decrement the resource's reference count, and
2. if the count went to zero, notify the thread waiting to destroy the resource, if there is one.

If this idiom provides performance adequate for your application, stop here! It is easy to write, easy to prove correct, and easy to audit. But this idiom does not scale in parallel for applications that need higher performance.

3.1 Improving lookup concurrency: reader/writer locks

The first scalability limitation of this prototypical example is that only one thread at a time can look up a resource in the table. We could replace the table's *mutex lock* by a *reader/writer lock*: use a writer lock for create/publish and remove/destroy, and a reader lock for for lookups.

With a reader/writer lock, there can be either up to one writer, or any number of readers, at any given time—a thread holding a writer lock excludes all other writers or readers, but a thread holding a reader lock excludes only writers. This helps if lookups take a long time, because there can be more than one lookup in flight.

However, for short-lived mutex sections or reader sections alike, there is another cost: contention over the mutex owner or the reader count itself. Both are implemented in terms of *atomic operations*, which are read/modify/write operations on a single word in memory that are guaranteed to run to completion as if only a single CPU were performing the operation on memory without interruption.

In a modern CPU with large caches and write queues, atomic operations done by multiple CPUs on the same

```
struct foo {
    uint64_t key;
    ...
    unsigned refcnt;
    struct foo *next;
};

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

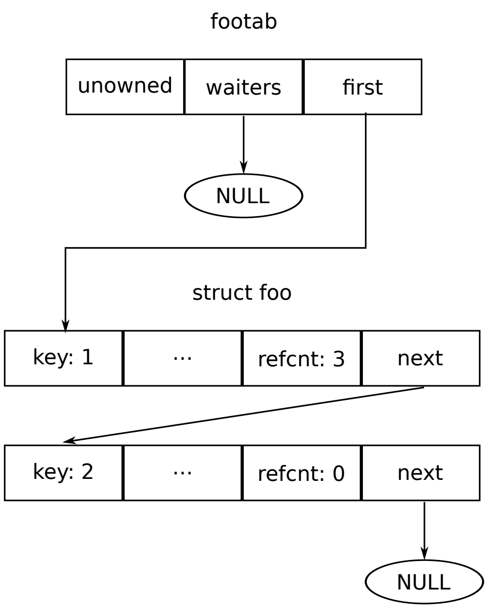


Figure 1: Prototypical resource and global state.

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

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

Figure 2: Prototypical create/publish.

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

Figure 3: Prototypical lookup/acquire.

```
mutex_enter(&footab.lock);
if (--f->refcnt == 0) {
    /* Last user.  Notify destroy. */
    cv_broadcast(&footab.cv);
}
mutex_exit(&footab.lock);
```

Figure 4: Prototypical release.

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

mutex_enter(&footab.lock);
for (fp = &footab.first;
     (f = *fp) != NULL;
     fp = &f->next) {
    if (f->key == key) {
        /* (a) Prevent new users. */
        *fp = f->next;
        /* (b) Wait for old users. */
        while (f->refcnt)
            cv_wait(&footab.cv,
                &footab.lock);
        break;
    }
}
mutex_exit(&footab.lock);

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

Figure 5: Prototypical remove/destroy.

part of memory—such as the mutex owner or the reader count—require the processors to communicate over the system bus to repeatedly invalidate one another's caches as they compete for ownership of the memory. This cost grows *worse* with more CPUs—it is not actually scalable in parallel. See [4, Section 4.1 *Why Isn't Concurrent Counting Trivial?*, p. 30] for more detail.

3.2 Distributing load: hashed locks

Contention over a single mutex or reader/writer lock can sometimes be mitigated by splitting the work into several independent tasks. Instead of a single global list of resources, we can use a hash table of resources, with some identifier determining which bucket to use. Each bucket can have an independent lock stored in a separate cache line to avoid contention between buckets, as in Figure 6. Lookup need only be slightly modified to compute the hash first; the other operations must be modified similarly.

Hashed locks are an easy generic way to attain better parallelism—provided that the distribution on resources being used is uniform. If every CPU still wants to use the same resource, it doesn't help!

Another similar approach is to associate a separate lock and condition variable with each resource to handle its reference count, which would cost more memory in exchange for less contention over the table lock when releasing resources. But this does nothing help to reduce contention of lookups, or to reduce the cost of using the same resource on many CPUs. In fact, on NetBSD, condition variables themselves are shared and hashed, so a separate condition variable per resource may actually cost more memory for no benefit in reducing contention or spurious wakeups.

3.3 Distributing load: atomic reference counts

Another source of contention is taking the global or hashed lock—whatever kind of lock it is, mutex or reader/writer—just to release a resource when a thread is done with it. The thread need not use the table any more, and it need not even coordinate with another thread trying to destroy the resource as long as there

To mitigate this contention, we can use atomic operations to manage each resource's reference count, and avoid taking the central lock except when the reference count would transition from nonzero to zero. This is illustrated in Figure 7 and Figure 8.

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

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);
```

Figure 6: Hashed resource lookup/acquire.

Conveniently, the protocol for destruction is identical to that for mutex-locked reference counts in Figure 5, because the transition from nonzero reference count to zero reference count in release happens under the same lock and is signalled via the same condition variable.

Atomic reference counts are safe, in the sense that if any thread releases the last reference it is guaranteed to wake any thread trying to destroy the resource: the only state transition of interest to a thread trying to destroy the resource is when it goes from nonzero to zero. Unfortunately, as with hashed locks, this doesn't help if any one resource is in hot demand by many CPUs—even if the CPUs need only read something from it nonexclusively, because the reference count updates must be coordinated among the CPUs.

4 No-contention approaches

The basic problem inhibiting scaling in parallel is that the CPUs must all agree on writes to some shared memory, the cost of which grows with the number of CPUs. How can we avoid this contention? Can we exchange some convenience for performance, or performance of one operation for performance of another?

```
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);
```

Figure 7: Atomic reference count acquire.

```
unsigned old, new;
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;
    }
    new = old - 1;
} while (atomic_cas_uint(&f->refcnt,
    old, new) != old);
```

Figure 8: Atomic reference count release.

NetBSD provides two no-contention APIs, and one in development, with different performance characteristics and different implications for API contracts:

- **Passive serialization**, or **pserialize**, scales well to many CPUs with zero memory overhead and almost no serial performance overhead, but is usable only for short-lived uninterruptible readers, such as looking up a hash table entry to acquire another kind of reference.
- **Passive references**, or **psref**, scale well to many CPUs with modest memory overhead and can be held arbitrarily long like reference counts, but only on a single CPU have somewhat more substantial serial performance overhead.
- **Local counts** scale well to many CPUs, can be held arbitrarily long and be passed from CPU to CPU, and have little serial performance overhead, but use memory proportional to the product of the number of resources and the number of CPUs or threads.

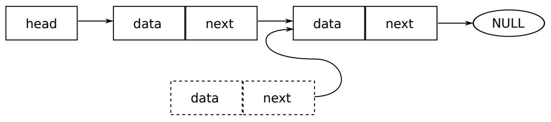
4.1 Passive serialization

Inserting an entry into a singly-linked list requires a single-pointer write to memory—nothing else is needed to preserve any internal invariants for the data structure, as long as any other content in the entry is fully initialized already. The same is true of deleting an entry. Simply reading a singly-linked list requires no writes at all, and so can be done safely in parallel by any number of threads.

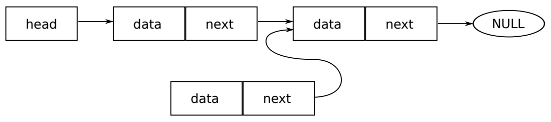
If some class of resources were never destroyed, a *single* thread inserting or deleting resources could operate in parallel with *any number* of threads looking up and using the resources, with no synchronization. The only restriction is that the content of new resources be visible no later than the pointers to new resources.

The publisher must ensure that it has issued any writes to memory initializing the content of a list entry before it issues the write inserting the entry into the list. Similarly, after any user reads a pointer to an entry, it must ensure that the CPU has read the content of the entry after it has read the pointer to the entry.[1] Publishing and use are illustrated in Figure 9 and Figure 10.

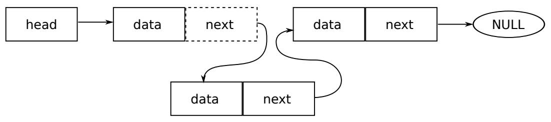
[1]This may be counterintuitive—surely if the writer issues writes to initialize the content before issuing a write to publish the pointer, and the reader has already read the pointer, the reader must also see the content. But in some CPU microarchitectures, the reader's CPU may have had stale content cached, yet read the pointer afresh.



(a) First write content: initialize data and set next pointer of new entry to next entry in list.

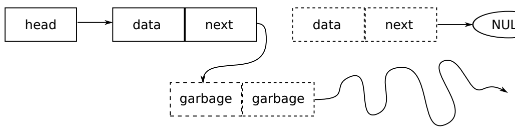


(b) Then issue memory barrier so data and next pointer will be published to all CPUs before any subsequent writes.

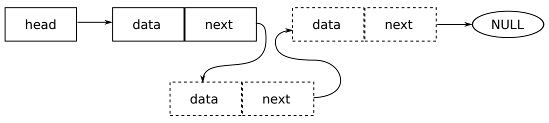


(c) Last write pointer: set next pointer of previous entry to point at new entry. Eventually the new entry, fully initialized, will be published to all other CPUs.

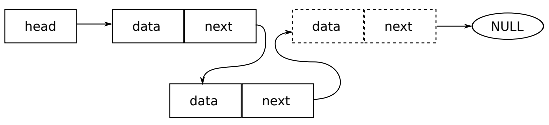
Figure 9: Memory-ordered linked-list insertion. Dotted boxes represent memory locations that the current CPU has written to, but which are not yet guaranteed to be published to other CPUs.



(a) First read pointer to new entry.



(b) Then issue memory barrier so data and next pointer will be freshly read and uncached in any subsequent read.



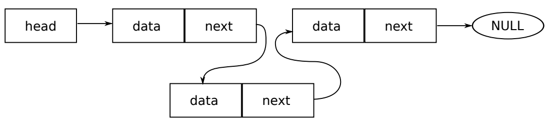
(c) Last read content.

Figure 10: Memory-ordered linked-list read. Dotted boxes represent memory locations that the current CPU has not yet read during a lookup operation, but show what the current CPU *would* observe if it did read them. Before the memory barrier, the CPU may see stale garbage from its cache.

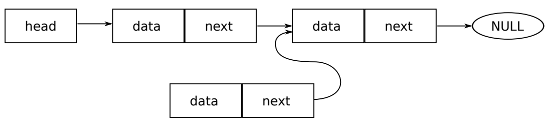
To satisfy this restriction, the publisher need only initialize the content of a new entry and then issue a write-before-write barrier before publishing a pointer to it. Similarly, the user need only read the pointer to an entry and then a read-before-data-dependent-read barrier before dereferencing it—and, fortuitously, a read-before-data-dependent-read barrier is actually a no-op on most CPUs. Deletion is even easier: a single pointer write and no memory barrier is required at all to prevent new readers from finding an entry in a linked list, as in Figure 11.

The catch, of course, is that this does not address resource *destruction*. A thread can delete a pointer to a resource, but how can it know when all other threads are done *using* it before destroying it?

The basic concept of passive serialization—and the closely related read–copy–update used extensively in the Linux kernel—is to make sure users take only short time, during which they block interrupts to inhibit preëmption and interprocessor interrput processing. Then, once a CPU has deleted a resource from the table to prevent new users, it can wait for all existing



(a) Before delete.



(b) After delete.

Figure 11: Memory-ordered linked list deletion. No memory barrier required because there is only one write.

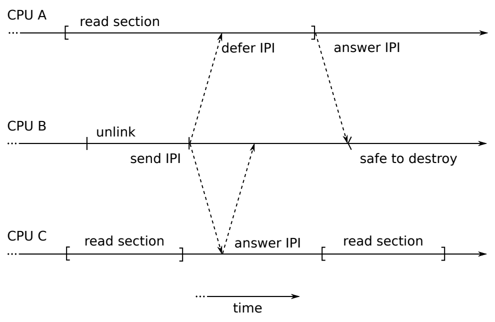


Figure 12: Passive serialization wait with `pserialize_perform`.

users to complete by sending an interprocessor interrupt to all CPUs and waiting until all have responded. If any other CPU were in the middle of using the resource, that CPU would also block processing of the interprocessor interrupt until done. The time between when a table entry is deleted and when all readers potentially using it have completed is called the **grace period**. The sequence of operations is illustrated in Figure 12.

Thus, the stages of passive serialization are as follows:

Create/publish To publish a newly created resource in a table, as in Figure 13, a CPU with exclusive access to the table must:

(a) Initialize memory for the resource.

(b) Issue a write-before-write memory barrier.

(c) Publish a pointer to the memory.

Lookup/use To look up and use a resource in a table, as in Figure 14, a CPU must:

(a) Use `pserialize_read_enter` to block interrupts.

(b) Look up a pointer to the resource.

(c) Issue a data-dependent-read memory barrier.

(d) Use the memory for the resource.

(e) Use `pserialize_read_exit` to restore and process queued interrupts.

The time between blocking and restoring interrupts is called a **pserialize read section**. It must be short to minimize the duration of blocked interrupts, and may not yield control of the CPU to another thread, *e.g.* by sleeping.

Delete/destroy To destroy a resource, as in Figure 15, a CPU with exclusive access to table must:

(a) Remove the pointer to the resource so that no new users can find it.

(b) Call `pserialize_perform` to send an interprocessor interrupt to all CPUs that will execute after any current users complete, and wait for it to finish on all CPUs.

(c) Destroy the resource.

Note that there is *zero* per-resource, per-reference, or per-CPU memory overhead for any particular application of pserialize—the overhead is $O(1)$ per subsystem that uses pserialize.[2] The serial performance impact of pserialize for readers is negligible; `pserialize_perform`, is very expensive, and so useful only when deletion is rare and/or batched.

To facilitate the correct memory barriers, NetBSD provides `PSLIST`, for 'pserialize-safe list', an alternative to the traditional `sys/queue.h` `LIST` macros. A `PSLIST` is a linked list just like `sys/queue.h`'s `LIST`, supporting constant-time insertion at the head or before or after any element and constant-time removal of an element, with all the necessary memory barriers for passive serialization on the writer side the reader side.

Pserialize is used for many table lookups in NetBSD. Often it is used in tandem with another mechanism such as atomic reference counts, passive references, or local counts, in order to at least provide scalable lookups if the resource in the table cannot be used and released immediately. Pserialize is particularly amenable to chained hash tables and radix trees, and we plan to use it soon for the routing table, although the current routing table code is old and crufty and not designed for multiprocessor environments.

Pserialize is similar to the read–copy–update found in the Linux kernel. The main difference is that pserialize was documented in a patent[3] that expired several years before the first read–copy–update patent[7], making it legally safer to use with a similar API. But now the first read–copy–update patent has expired, so we may replace the implementation of the `pserialize(9)` API in NetBSD by the read–copy–update algorithm, which is simpler and faster.

[2]Actually, there are exactly two bits of overhead per CPU in the current implementation, owing to the overcomplicated algorithm that we use, which is no longer necessary. See below about the patent situation of passive serialization versus read–copy–update.

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

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

Figure 13: Pserialize create/publish.

```
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);
```

Figure 14: Pserialize lookup/use.

```
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);
}
```

Figure 15: Pserialize delete/destroy.

4.2 Passive references

What a thread may do while it holds a pserialize reference—*i.e.*, in a pserialize read section—is very limited. The reason is that the criterion by which a thread trying to destroy a resource determine whether it is still in use is extremely coarse: after the resource has been deleted from the table, *any* activity on all other CPUs is taken to mean the resource must no longer be in use.

Instead of taking *activity* as an indicator for use, we could simply record a per-CPU list of exactly which resources are in use on that CPU. A **passive reference** is an entry on a per-CPU list of resources in use by that CPU. The protocol for acquiring, releasing, and destroying—assuming some other mechanism, typically pserialize, for publishing, looking up, and deleting in a table—is:

Acquire Performed by `psref_acquire`:

(a) Block interrupts.
(b) Allocate a list entry on the stack pointing at the resource and representing the reference.
(c) Insert the entry into the current CPU's list of resources in use.
(d) Restore interrupts.

Release Performed by `psref_release`:

(a) Block interrputs.
(b) Remove the entry from the current CPU's list of resources in use.
(c) Restore interrupts.
(d) Check a flag indicating whether there is a thread waiting to destroy the resource, predicted to be unset. If set, notify that thread that a reference was released.

Destroy Performed by `psref_target_destroy`:

(a) Set a flag indicating that there is a thread waiting to destroy the resource.
(b) Send an interprocessor interrupt to each CPU that checks whether the resource in question is on the CPU's list of resources in use.
(c) Wait for all CPUs to answer.
(d) If the resource still in use, wait for a few milliseconds or until woken by a thread releasing a resource, and try again.

The timed wait in destruction is necessary because the test for a flag in a releasing thread cannot be interlocked with a destroying thread without requiring atomic operations in `psref_release`, which is exactly what we want to avoid. In fact, the explicit wakeups from a releasing thread are only a potentially unnecessary optimization.

The memory overhead of psref grows as $O(\text{\#resources}) + O(\text{\#CPUs}) + O(\text{\#references})$. The serial performance impact of psref for readers is that of adjusting a per-CPU linked list threaded through likely-cached memory on the stack, which is noticeable but small—in network packet routing, it has been measured to reduce the maximum serial bandwidth by a few percent. For deletion, psref costs a little more time than pserialize, because it must do additional work in an interprocessor interrupt to search for references.

We would use pserialize alone in more applications, but the restriction on what a CPU can do in a pserialize read section means it is not a drop-in replacement for reader/writer locks and atomic reference counts. Instead, we use it in tandem with passive references, which together are close to a drop-in replacement for reader/writer locks and atomic reference counts, shown in Figure 16, Figure 17, and Figure 18. This enables us to incrementally re-engineer subsystems to be scalable in parallel, at some cost in serial performance, without requiring substantial cost up front to redesign them to be pserialize-safe.

The only API contract requirement that NetBSD psref imposes beyond a reader/writer lock is that the holder of a passive reference must remain on the same CPU from acquisition to release. For many applications in NetBSD, e.g. soft interrupts for processing network packets, the threads acquiring passive references are often already bound to a CPU anyway, so this requirement is not a burden. For other applications, this requires conditionally setting a single flag bit in the `struct lwp` object ('lightweight process') representing the thread, and conditionally clearing it when done.

We could have required that passive references be associated with a *thread* rather than a *CPU*. However, there are typically many times more threads than CPUs in a running system, and most threads hold no passive references. This would increase the memory and time required by passive references: one list head for storage and one interrupt for deletion per thread, rather than one list per CPU. Since most threads holding passive references do not hold them for very long times, and many are bound to CPUs anyway, it is not important

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

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

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

Figure 16: Create/publish with pserialize/psref.

to let them switch CPUs. Thus, there is some cost and negligible benefit to associating passive references with threads instead of CPUs.

4.3 Local counts

Instead of a *global* reference count, for objects that are relatively few in number, such as device drivers, we can allocate memory to each CPU for **local counts** of references that have been acquired or released on that CPU. The protocol for acquiring, releasing, and destroying is:

Acquire (a) Increment a CPU-local integer.

Release (a) Decrement a CPU-local integer.

(b) If there is a thread waiting to destroy the resource, decrement its temporary global reference count, and if that went to zero, notify the thread.

Destroy (a) Initialize a temporary global reference count.

(b) Send an interprocessor interrupt to all CPUs to contribute each CPU's local reference count to the temporary global reference count.

(c) Wait until the interprocessor interrupt has completed.

(d) Wait until the temporary global reference count is zero.

Acquiring a resource's local count entails only incrementing a CPU-local integer in memory—guaranteed never to be contended. Releasing a resource's local count *usually* entails only decrementing a CPU-local integer in memory, unless there is a thread trying to destroy the resource, in which case it may have to acquire a lock and notify a condition variable to wake that

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

/* Bind to current CPU and lookup. */
bound = curlwp_bind();
s = pserialize_read_enter();
for (f = footab.first;
     f != NULL;
     f = f->next) {
    if (f->key == key) {
        psref_acquire(&fref,
            &f->target, footab.psr);
        break;
    }
}
pserialize_read_exit(s);

if (f == NULL)
    goto fail;
KASSERT(psref_held(&f->target, footab.psr));
...use f...

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

Figure 17: Lookup/use with pserialize/psref.

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

if (f != NULL) {
    /* (b) Wait for old users.  */
    psref_target_destroy(&f->target,
        footab.psr);
    /* (c) Destroy. */
    free_foo(f);
}
```

Figure 18: Delete/destroy with pserialize/psref.

thread. As with passive serialization and passive references, destroying requires expensive interprocessor interrupts.

The sequential time cost of local count acquisition and release is slightly smaller than passive references—incrementing and decrementing a CPU-local counter, rather than maintaining a list of pointers. But more importantly, local counts are more flexible than passive references because they are not required to remain on the same CPU. Consequently they are safe to drop in to any nontrivial existing code base with negligible impact on serial performance, parallel scalability, or API contract.

However, the memory cost of local counts is much higher than anything else mentioned so far: $O(\#\text{CPU} \times \#\text{resource})$. Local counts are currently an experiment on a branch in NetBSD for making device driver unloading MP-safe without inflicting a sequential or parallel performance penalty on all device driver operations.

5 Related work

There have been many abstractions built for holding onto things in a multiprocessor world. Here is a brief listing of related work:

Hazard pointers are a pattern for listing the pointers to resources in use,[6] prefiguring the design of passive references, but with a somewhat more complicated presentation and no definite API presented out of the box.

Shared reference pointers or SRP in OpenBSD are a cheaper variant of hazard pointers that use an array, rather than linked list, of pointers in use by a CPU.[1] The API is considerably more involved than the NetBSD `psref(9)` API, and requires more memory barriers and atomic operations under the hood.

Read–copy–update or RCU for short is essentially the same general idea as passive serialization, and is widely used in the Linux kernel.[5] The API is a little more elaborate than the pserialize API—for example, it includes a mechanism for queueing a list of callbacks for when all extant read sections are complete. The implementation is a little less complex and faster than the pserialize implementation, owing to patent issues.

Quiescent-state-based and epoch-based reclamation are variations on RCU[2] that avoid interprocessor interrupts—and thus are easier to adapt to userland libraries—at the expense of more onerous API contracts requiring users to identify not when they *are* using resources, but when they are *not* using any resources.

References

[1] David Gwynne and Jonathan Matthew. `srp_enter(9)`: Shared reference pointers. OpenBSD Manual, 2017. URL: `http://man.openbsd.org/OpenBSD-current/man9/srp_enter.9`.

[2] Thomas Edwart Hart. Comparative performance of memory reclamation strategies for lock-free and concurrently-readable data structures. Master's thesis, University of Toronto, 2005.

[3] James P. Hennessy, Damian L. Osisek, and W. Seigh II Joseph. Passive serialization in a multitasking environment. U.S. Patent 4,809,168, 1989.

[4] Paul McKenney. *Is Parallel Programming Hard, And, If So, What Can You Do About It?* 2011. URL: `https://www.kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.2011.01.02a.pdf`.

[5] Paul McKenney and Jonathan Walpole. What is rcu, fundamentally? Linux Weekly News, 2007. URL: `https://lwn.net/Articles/262464/`.

[6] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. on Parallel and Distributed Systems*, 15(6):491–504, June 2004.

[7] John D. Slingwine and Paul McKenney. Apparatus and method for achieving reduced overhead mutual exclusion and maintaining coherency in a multiprocessor system utilizing execution history and thread monitoring. U.S. Patent 5,442,758, 1995.