Is Parallel Programming Hard, And, If So, What Can You Do About It?

Paul E. McKenny

August 22, 2024

Contents

1 Tools of the Trade

1.1 POSIX Multiprocessing

1.2 Alternatives to POSIX Operations

1.3 Accessing Shared Variables

```
1 ptr = global_ptr;
2 if (ptr != NULL && ptr < high_address)
3     do_low(ptr);
```

Listing 1: Living Dangerously Early 1990s Style

```
1 if (global_ptr != NULL &&
2     global_ptr < high_address)
3     do_low(global_ptr);
```

Listing 2: C Compilers Can Invent Loads

1.3.1 Shared-Variable Shenanigans

Given code that does plain loads and stores,the compiler is within its rights to assume that the affected variables are neither accessed nor modified by any other thread. This assumption allows the compiler to carry out a large number of transformations, including load tearing, store tearing, load fusing, store fusing, code reordering, invented loads, invented stores, store-to-load transformations, and dead- code elimination.

- **Load tearing** occurs when the compiler uses multiple load instructions for a single access.
- **Store tearing** occurs when the compiler uses multiple store instructions for a single access.
- **Load fusing** occurs when the compiler uses the result of a prior load from a given variable instead of repeating the load.

```
while (!need_to_stop)
    do_something_quickly();
```

Listing 3: Inviting Load Fusing

```
if (!need_to_stop)
    for (;;) {
        do_something_quickly();
        do_something_quickly();
        do_something_quickly();
        do_something_quickly();
        do_something_quickly();
        do_something_quickly();
        do_something_quickly();
        do_something_quickly();
        do_something_quickly();
        do_something_quickly();
        do_something_quickly();
        do_something_quickly();
        do_something_quickly();
        do_something_quickly();
        do_something_quickly();
        do_something_quickly();
        do_something_quickly();
    }
```

Listing 4: C Compilers Can Fuse Loads

- **Store fusing** can occur when the compiler notices a pair of successive stores to a given variable with no intervening loads from that variable.

```
1  void shut_it_down(void)
2  {
3      status = SHUTTING_DOWN; /* BUGGY!!! */
4      start_shutdown();
5      while (!other_task_ready) /* BUGGY!!! */
6          continue;
7      finish_shutdown();
8      status = SHUT_DOWN; /* BUGGY!!! */
9      do_something_else();
10 }
11
12 void work_until_shut_down(void)
13 {
14     while (status != SHUTTING_DOWN) /* BUGGY!!! */
15         do_more_work();
16     other_task_ready = 1; /* BUGGY!!! */
17 }
```

Listing 5: C Compilers Can Fuse Stores

- **Code reordering**. It might seem futile to prevent the compiler from changing the order of accesses in cases where the underlying hardware is free to reorder them. However, modern machines have *exact exceptions* and *exact interrupts*, meaning that any interrupt or exception will appear to have happened at a specific place in the instruction stream. This means that the handler will see the effect of all prior instructions, but won't see the effect of any subsequent instructions.

- **Invented loads** were illustrated by the code in Listings 1 and 2, in which the compiler optimized away a temporary variable, thus loading from a shared variable more often than intended.

  - **Invented stores**: For example, a compiler emitting code for `work_until_shut_down()` in Listing 5 might notice that `other_task_ready` is not accessed by `do_more_work()`, and stored to on line 16. If `do_more_work()` was a complex inline function, it might be necessary to do a register spill, in which case one attractive place to use for temporary

storage is `other_task_ready`. After all, there are no accesses to it, so what is the harm?

```
if (condition)
    a = 1;
else
    do_a_bunch_of_stuff(&a);
```

Listing 6: Inviting an Invented Store

```
a = 1;
if (!condition) {
    a = 0;
    do_a_bunch_of_stuff(&a);
}
```

Listing 7: Compiler Invents an Invited Store

– **Store-to-load transformations** can occur when the compiler notices that a plain store might not actually change the value in memory.

```
r1 = p;
if (unlikely(r1))
    do_something_with(r1);
barrier();
p = NULL;
```

Listing 8: Inviting a Store-to-Load Conversion

– **Dead-code elimination**

1.3.2 A Volatile Solution

To summarize, the `volatile` keyword can prevent load tearing and store tearing in cases where the loads and stores are machine-sized and properly aligned. It can also prevent load fusing, store fusing, invented loads, and

```
r1 = p;
if (unlikely(r1))
    do_something_with(r1);
barrier();
if (p != NULL)
    p = NULL;
```

Listing 9: Compiler Converts a Store to a Load

invented stores. However, although it does prevent the compiler from reordering volatile accesses with each other, it does nothing to prevent the CPU from reordering these accesses. Furthermore, it does nothing to prevent either compiler or CPU from reordering non-`volatile` accesses with each other or with `volatile` accesses. Preventing these types of reordering requires the techniques described in the next section.

1.3.3 Assembling the Rest of a Solution

```
#define barrier() __asm__ __volatile__ ("" : : : "memory")
```

In the barrier() macro, the `__asm__` introduces the asm directive, the `__volatile__` prevents the compiler from optimizing the asm away, the empty string specifies that no actual instructions are to be emitted, and the final `"memory"` tells the compiler that this do-nothing asm can arbitrarily change memory. In response, the compiler will avoid moving any memory references across the barrier() macro. This means that the real-time- destroying loop unrolling shown in Listing 4 can be prevented by adding `barrier()` calls as shown on lines 2 and 4 of Listing 4.28. `barrier()` is for compiler. For hardware, we need `smp_mb` These two lines of code prevent the compiler from pushing the

```
while (!need_to_stop) {
    barrier();
    do_something_quickly();
    barrier();
}
```

Listing 10: Preventing C Compilers From Fusing Loads

load from `need_to_stop` into or past `do_something_quickly()` from either direction.

However, this does nothing to prevent the CPU from reordering the references.

```
// arch-arm/arch-arm.h
#define smp_mb()  __asm__ __volatile__("dmb" : : : "memory")

// arch-x86/arch-x86.h
#define smp_mb() __asm__ __volatile__("mfence" : : : "memory")

// arch-ppc64/arch-ppc64.h
#define smp_mb()  __asm__ __volatile__("sync" : : : "memory")

// arch-arm64/arch-arm64.h
#define smp_mb()  __asm__ __volatile__("dmb ish" : : : "memory")
```

2 Deferred Processing

2.1 Running Example

The value looked up and returned will also be a simple integer, so that the data structure is as shown in Figure 1, which directs packets with address 42 to interface 1, address 56 to interface 3, and address 17 to interface 7.

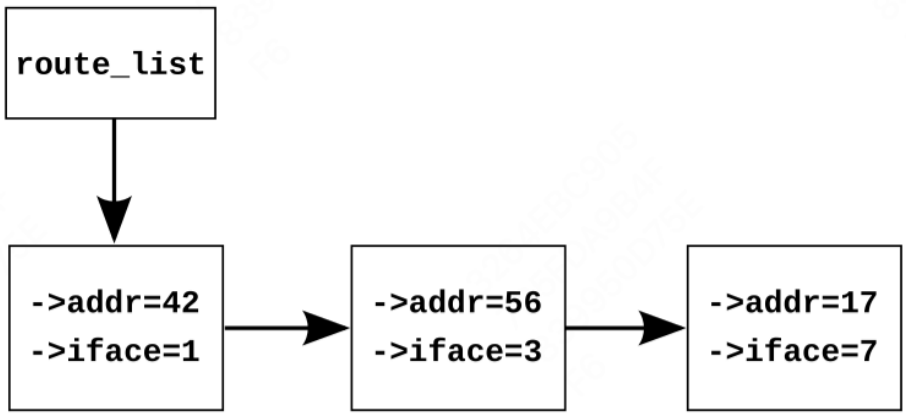

Figure 1: Pre-BSD Packet Routing List

```
struct route_entry {
    struct cds_list_head re_next;
    unsigned long addr;
    unsigned long iface;
};
CDS_LIST_HEAD(route_list);

unsigned long route_lookup(unsigned long addr)
{
    struct route_entry *rep;
    unsigned long ret;

    cds_list_for_each_entry(rep, &route_list, re_next) {
        if (rep->addr == addr) {
            ret = rep->iface;
            return ret;
        }
    }
    return ULONG_MAX;
}

int route_add(unsigned long addr, unsigned long interface)
{
    struct route_entry *rep;

    rep = malloc(sizeof(*rep));
    if (!rep)
        return -ENOMEM;
    rep->addr = addr;
    rep->iface = interface;
    cds_list_add(&rep->re_next, &route_list);
    return 0;
}

int route_del(unsigned long addr)
{
    struct route_entry *rep;

    cds_list_for_each_entry(rep, &route_list, re_next) {
        if (rep->addr == addr) {
            cds_list_del(&rep->re_next);
            free(rep);
            return 0;
        }
    }
    return -ENOENT;
}
```

Listing 11: Sequential Pre-BSD Routing Table

Listing 11 (`route_seq.c`) shows a simple single-threaded implementation corresponding to Figure 1.

2.2 Reference Counting

Starting with Listing 12, line 2 adds the actual reference counter, line 6 adds a `->re_freed` use-after-free check field, line 9 adds the `routelock` that will be used to synchronize concurrent updates, and lines 11–15 add `re_free()`, which sets `->re_freed`, enabling `route_lookup()` to check for use-after-free bugs. In `route_lookup()` itself, lines 29–30 release the reference count of the prior element and free it if the count becomes zero, and lines 34–42 acquire a reference on the new element, with lines 35 and 36 performing the use-after-free check.

Remark. Why bother with a use-after-free check?

To greatly increase the probability of finding bugs

In Listing 13, lines 11, 15, 24, 32, and 39 introduce locking to synchronize concurrent updates. Line 13 initializes the `->re_freed` use-after-free-check field, and finally lines 33–34 invoke `re_free()` if the new value of the reference count is zero.

Remark. Why doesn't `route_del()` in Listing 13 use reference counts to protect the traversal to the element to be freed?

Because the traversal is already protected by the lock, so no additional protection is required.

```
struct route_entry {
    atomic_t re_refcnt;
    struct route_entry *re_next;
    unsigned long addr;
    unsigned long iface;
    int re_freed;
};
struct route_entry route_list;
DEFINE_SPINLOCK(routelock);

static void re_free(struct route_entry *rep)
{
    WRITE_ONCE(rep->re_freed, 1);
    free(rep);
}

unsigned long route_lookup(unsigned long addr)
{
    int old;
    int new;
    struct route_entry *rep;
    struct route_entry **repp;
    unsigned long ret;

retry:
    repp = &route_list.re_next;
    rep = NULL;
    do {
        if (rep && atomic_dec_and_test(&rep->re_refcnt))
            re_free(rep);
        rep = READ_ONCE(*repp);
        if (rep == NULL)
            return ULONG_MAX;
        do {
            if (READ_ONCE(rep->re_freed))
                abort();
            old = atomic_read(&rep->re_refcnt);
            if (old <= 0)
                goto retry;
            new = old + 1;
        } while (atomic_cmpxchg(&rep->re_refcnt,
                                old, new) != old);
        repp = &rep->re_next;
    } while (rep->addr != addr);
    ret = rep->iface;
    if (atomic_dec_and_test(&rep->re_refcnt))
        re_free(rep);
    return ret;
}
```


Listing 12: Reference-Counted Pre-BSD Routing Table Lookup (BUGGY)

```
int route_add(unsigned long addr, unsigned long interface)
{
    struct route_entry *rep;

    rep = malloc(sizeof(*rep));
    if (!rep)
        return -ENOMEM;
    atomic_set(&rep->re_refcnt, 1);
    rep->addr = addr;
    rep->iface = interface;
    spin_lock(&routelock);
    rep->re_next = route_list.re_next;
    rep->re_freed = 0;
    route_list.re_next = rep;
    spin_unlock(&routelock);
    return 0;
}

int route_del(unsigned long addr)
{
    struct route_entry *rep;
    struct route_entry **repp;

    spin_lock(&routelock);
    repp = &route_list.re_next;
    for (;;) {
        rep = *repp;
        if (rep == NULL)
            break;
        if (rep->addr == addr) {
            *repp = rep->re_next;
            spin_unlock(&routelock);
            if (atomic_dec_and_test(&rep->re_refcnt))
                re_free(rep);
            return 0;
        }
        repp = &rep->re_next;
    }
    spin_unlock(&routelock);
    return -ENOENT;

}
```

Listing 13: Reference-Counted Pre-BSD Routing Table Add/Delete (BUGGY)

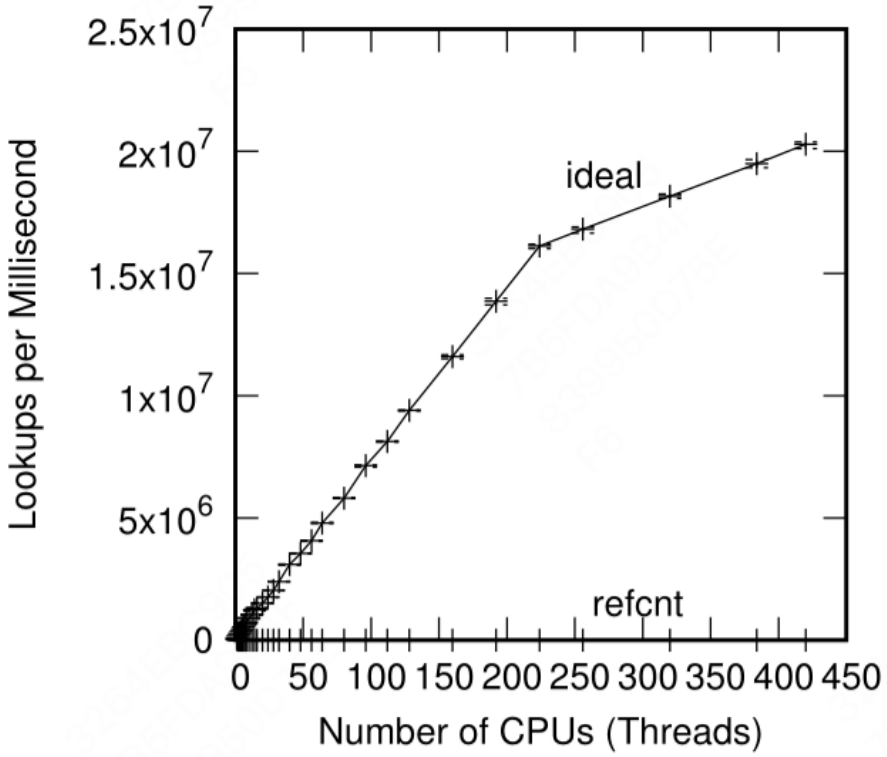

Figure 2: Pre-BSD Routing Table Protected by Reference Counting

Ideal is from Listing 11. Refcnt performance is abysmal.

Remark. Why the break in the "ideal" line at 224 CPUs in Figure 9.2? Shouldn't it be a straight line?

The break is due to hyperthreading. On this particular system, the first hardware thread in each core within a socket have consecutive CPU numbers, followed by the first hardware threads in each core for the other sockets, and finally followed by the second hardware thread in each core on all the sockets. On this particular system, CPU numbers 0–27 are the first hardware threads in each of the 28 cores in the first socket, numbers 28–55 are the first hardware threads in each of the 28 cores in the second socket, and so on, so that numbers 196–223 are the first hardware threads in each of the 28 cores in the eighth socket. Then CPU numbers 224–251 are the second hardware threads in each of the 28 cores of the first socket, numbers 252–279 are the second hardware threads in each of the 28 cores of the second socket, and so on until numbers 420–447 are the second hardware threads in each of the 28 cores of the eighth socket.

Why does this matter?

Because the two hardware threads of a given core share resources, and this workload seems to allow a single hardware thread to consume more than half of the relevant resources within its core. Therefore, adding the second hardware thread of that core adds less than one might hope. Other workloads might gain greater benefit from each core's second hardware thread, but much depends on the details of both the hardware and the workload.

One sequence of events leading to the use-after-free bug is as follows, given the list shown in Figure 11:

1. Thread A looks up address 42, reaching line 32 of `route_lookup()` in Listing 12. In other words, Thread A has a pointer to the first element, but has not yet acquired a reference to it.

2. Thread B invokes `route_del()` in Listing 12 to delete the route entry for address 42. It completes successfully, and because this entry's `->re_refcnt` field was equal to the value one, it invokes `re_free()` to set the `->re_freed` field and to free the entry.

3. Thread A continues execution of `route_lookup()`. Its rep pointer is non-`NULL`, but line 35 sees that its `->re_freed` field is non-zero, so line 36 invokes `abort()`

2.3 Hazard Pointers

One way of avoiding problems with concurrent reference counting is to implement the reference counters inside out, that is, rather than incrementing an integer stored in the data element, instead store a pointer to that data element in per-CPU (or per-thread) lists. Each element of these lists is called a **hazard pointer**.

The value of a given data element's "virtual reference counter" can then be obtained by counting the number of hazard pointers referencing that element. Therefore, if that element has been rendered inaccessible to readers, and there are no longer any hazard pointers referencing it, that element may safely be freed.

```
1  /* Parameters to the algorithm:
2   *  K: Number of hazard pointers per thread.
3   *  H: Number of hazard pointers required.
4   *  R: Chosen such that R = H + Omega(H).
```

```
5   */
6   #define K 2
7   #define H (K * NR_THREADS)
8   #define R (100 + 2*H)
9
10  /* Must be the first field in the hazard-pointer-protected structure. */
11  /* It is illegal to nest one such structure inside another. */
12  typedef struct hazptr_head {
13  ^^Istruct hazptr_head *next;
14  } hazptr_head_t;
15
16  typedef struct hazard_pointer_s {
17  ^^Ivoid *  __attribute__ ((__aligned__ (CACHE_LINE_SIZE))) p;
18  } hazard_pointer;
19
20  /* Must be dynamically initialized to be an array of size H. */
21  hazard_pointer *HP;
22
23  void hazptr_init(void);
24  void hazptr_thread_exit(void);
25  void hazptr_scan();
26  void hazptr_free_later(hazptr_head_t *);
27  void hazptr_free(void *ptr); /* supplied by caller. */
28
29  #define HAZPTR_POISON 0x8
30
31  static hazptr_head_t __thread *rlist;
32  static unsigned long __thread rcount;
33  static hazptr_head_t __thread **gplist;
```

The `hp_try_record()` macro on line 16 is simply a casting wrapper for the `_h_t_r_impl()` function, which attempts to store the pointer referenced by `p` into the hazard pointer referenced by `hp`. If successful, it returns the value of the stored pointer. If it fails due to that pointer being `NULL`, it returns `NULL`. Finally, if it fails due to racing with an update, it returns a special `HAZPTR_POISON` token.

Remark. Given that papers on hazard pointers use the bottom bits of each pointer to mark deleted elements, what is up with `HAZPTR_POISON`?

Line 6 reads the pointer to the object to be protected. If line 8 finds that this pointer was either `NULL` or the special `HAZPTR_POISON` deleted-object token, it returns the pointer's value to inform the caller of the failure. Otherwise, line 9 stores the pointer into the specified hazard pointer, and line 10 forces full ordering of that store with the reload of the original pointer on line 11. If the value of the original pointer has not changed, then the haz-

```
static inline void *_h_t_r_impl(void **p,
                                hazard_pointer *hp)
{
    void *tmp;

    tmp = READ_ONCE(*p);
    if (!tmp || tmp == (void *)HAZPTR_POISON)
        return tmp;
    WRITE_ONCE(hp->p, tmp);
    smp_mb();
    if (tmp == READ_ONCE(*p))
        return tmp;
    return (void *)HAZPTR_POISON;
}

#define hp_try_record(p, hp) _h_t_r_impl((void **)(p), hp)

static inline void *hp_record(void **p,
                              hazard_pointer *hp)
{
    void *tmp;

    do {
        tmp = hp_try_record(p, hp);
    } while (tmp == (void *)HAZPTR_POISON);
    return tmp;
}


static inline void hp_clear(hazard_pointer *hp)
{
    smp_mb();
    WRITE_ONCE(hp->p, NULL);
}
```

Listing 14: Hazard-Pointer Recording and Clearing

ard pointer protects the pointed-to object, and in that case, line 12 returns a pointer to that object, which also indicates success to the caller. Otherwise, if the pointer changed between the two `READ_ONCE()` invocations, line 13 indicates failure. ({the second read ensures that `p` is not changed between the read and write})

The `hp_clear()` function is even more straightforward, with an `smp_mb()` to force full ordering between the caller's uses of the object protected by the hazard pointer and the setting of the hazard pointer to `NULL`.

Once a hazard-pointer-protected object has been removed from its linked data structure, so that it is now inaccessible to future hazard-pointer readers, it is passed to `hazptr_free_later()`, which is shown on lines 48–56 of Listing 15. Lines 50 and 51 enqueue the object on a per-thread list rlist and line 52 counts the object in rcount. If line 53 sees that a sufficiently large number of objects are now queued, line 54 invokes `hazptr_scan()` to attempt to free some of them.

The `hazptr_scan()` function is shown on lines 6–46 of the listing. This function relies on a fixed maximum number of threads (`NR_THREADS`) and a fixed maximum number of hazard pointers per thread (`K`), which allows a fixed-size array of hazard pointers to be used. Because any thread might need to scan the hazard pointers, each thread maintains its own array, which is referenced by the per-thread variable `gplist`. If line 14 determines that this thread has not yet allocated its gplist, lines 15–18 carry out the allocation. The memory barrier on line 20 ensures that all threads see the removal of all objects by this thread before lines 22–28 scan all of the hazard pointers, accumulating non-`NULL` pointers into the `plist` array and counting them in `psize`. The memory barrier on line 29 ensures that the reads of the hazard pointers happen before any objects are freed. Line 30 then sorts this array to enable use of binary search below.

Lines 31 and 32 remove all elements from this thread's list of to-be-freed objects, placing them on the local tmplist and line 33 zeroes the count. Each pass through the loop spanning lines 34–45 processes each of the to-be-freed objects. Lines 35 and 36 remove the first object from tmplist, and if lines 37 and 38 determine that there is a hazard pointer protecting this object, lines 39–41 place it back onto rlist. Otherwise, line 43 frees the object.

The Pre-BSD routing example can use hazard pointers as shown in Listing 16 for data structures and `route_lookup()`, and in Listing 9.7 for `route_add()` and `route_del()` (`route_hazptr.c`). As with reference counting, the hazard-pointers implementation is quite similar to the sequential algorithm shown in Listing 11, so only differences will be discussed.

Starting with Listing 16, line 2 shows the `->hh` field used to queue ob-

```
int compare(const void *a, const void *b)
{
    return ( *(hazptr_head_t **)a - *(hazptr_head_t **)b );
}

void hazptr_scan()
{
    hazptr_head_t *cur;
    int i;
    hazptr_head_t *tmplist;
    hazptr_head_t **plist = gplist;
    unsigned long psize;

    if (plist == NULL) {
        psize = sizeof(hazptr_head_t *) * K * NR_THREADS;
        plist = (hazptr_head_t **)malloc(psize);
        BUG_ON(!plist);
        gplist = plist;
    }
    smp_mb();
    psize = 0;
    for (i = 0; i < H; i++) {
        uintptr_t hp = (uintptr_t)READ_ONCE(HP[i].p);

        if (!hp)
            continue;
        plist[psize++] = (hazptr_head_t *)(hp & ~0x1UL);

    }
    smp_mb();
    qsort(plist, psize, sizeof(hazptr_head_t *), compare);
    tmplist = rlist;
    rlist = NULL;
    rcount = 0;
    while (tmplist != NULL) {
        cur = tmplist;
        tmplist = tmplist->next;
        if (bsearch(&cur, plist, psize,
                    sizeof(hazptr_head_t *), compare)) {
            cur->next = rlist;
            rlist = cur;
            rcount++;
        } else {
            hazptr_free(cur);
        }
    }
}

void hazptr_free_later(hazptr_head_t *n)
{
    n->next = rlist;
    rlist = n;
    rcount++;
    if (rcount >= R) {
        hazptr_scan();
    }
}
```

```
struct route_entry {
    struct hazptr_head hh;
    struct route_entry *re_next;
    unsigned long addr;
    unsigned long iface;
    int re_freed;
};
struct route_entry route_list;
DEFINE_SPINLOCK(routelock);
hazard_pointer __thread *my_hazptr;

unsigned long route_lookup(unsigned long addr)
{
    int offset = 0;
    struct route_entry *rep;
    struct route_entry **repp;

retry:
    repp = &route_list.re_next;
    do {
        rep = hp_try_record(repp, &my_hazptr[offset]);
        if (!rep)
            return ULONG_MAX;
        if ((uintptr_t)rep == HAZPTR_POISON)
            goto retry;
        repp = &rep->re_next;
    } while (rep->addr != addr);
    if (READ_ONCE(rep->re_freed))
        abort();
    return rep->iface;
}
```

Listing 16: Hazard-Pointer Pre-BSD Routing Table Lookup

jects pending hazard-pointer free, line 6 shows the `->re_freed` field used to detect use-after-free bugs, and line 21 invokes `hp_try_record()` to attempt to acquire a hazard pointer. If the return value is `NULL`, line 23 returns a not-found indication to the caller. If the call to `hp_try_record()` raced with deletion, line 25 branches back to line 18's retry to re-traverse the list from the beginning. The do–while loop falls through when the desired element is located, but if this element has already been freed, line 29 terminates the program. Otherwise, the element's `->iface` field is returned to the caller.

Note that line 21 invokes `hp_try_record()` rather than the easier-to-use `hp_record()`, restarting the full search upon `hp_try_record()` failure. And such restarting is absolutely required for correctness. To see this, consider a hazard-pointer-protected linked list containing elements A, B, and C that is subjected to the following sequence of events:

1. Thread 0 stores a hazard pointer to element B (having presumably traversed to element B from element A).

2. Thread 1 removes element B from the list, which sets the pointer from element B to element C to the special `HAZPTR_POISON` value in order to mark the deletion. Because Thread 0 has a hazard pointer to element B, it cannot yet be freed.

3. Thread 1 removes element C from the list. Because there are no hazard pointers referencing element C, it is immediately freed.

4. Thread 0 attempts to acquire a hazard pointer to now-removed element B's successor, but `hp_try_record()` returns the `HAZPTR_POISON` value, forcing the caller to restart its traversal from the beginning of the list.

Therefore, hazard-pointer readers must typically restart the full traversal in the face of a concurrent deletion.

These hazard-pointer restrictions result in great benefits to readers, courtesy of the fact that the hazard pointers are stored local to each CPU or thread, which in turn allows traversals to be carried out without any writes to the data structures being traversed.

```
int route_add(unsigned long addr, unsigned long interface)
{
    struct route_entry *rep;

    rep = malloc(sizeof(*rep));
    if (!rep)
        return -ENOMEM;
    rep->addr = addr;
    rep->iface = interface;
    rep->re_freed = 0;
    spin_lock(&routelock);
    rep->re_next = route_list.re_next;
    route_list.re_next = rep;
    spin_unlock(&routelock);
    return 0;
}


int route_del(unsigned long addr)
{
    struct route_entry *rep;
    struct route_entry **repp;

    spin_lock(&routelock);
    repp = &route_list.re_next;
    for (;;) {
        rep = *repp;
        if (rep == NULL)
            break;
        if (rep->addr == addr) {
            *repp = rep->re_next;
            rep->re_next = (struct route_entry *)HAZPTR_POISON;
            spin_unlock(&routelock);
            hazptr_free_later(&rep->hh);
            return 0;
        }
        repp = &rep->re_next;
    }
    spin_unlock(&routelock);
    return -ENOENT;
}
```

Listing 17: Hazard-Pointer Pre-BSD Routing Table Add/Delete

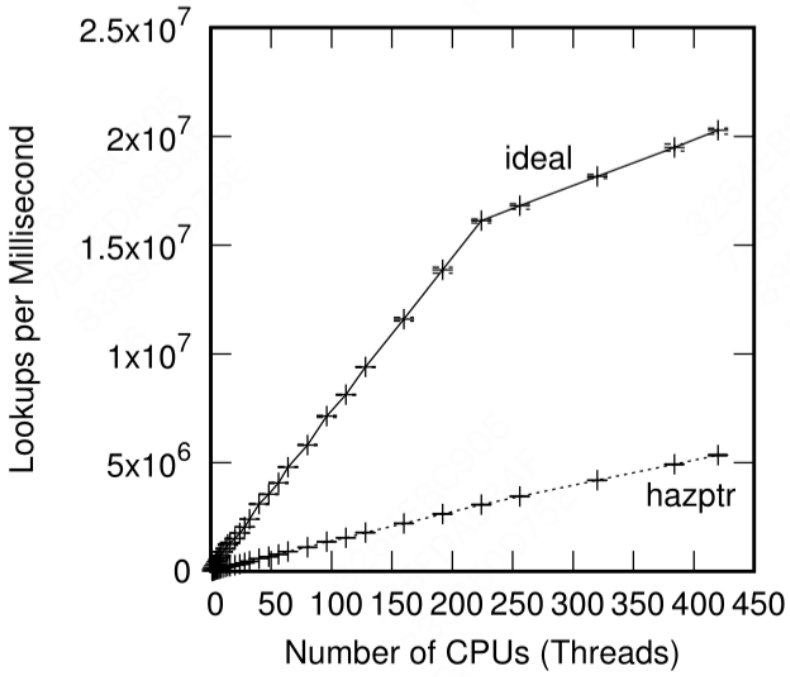

Figure 3: Pre-BSD Routing Table Protected by Hazard Pointers

Remark.

- Figure 3 shows no sign of hyperthread-induced flattening at 224 threads. Why is that?

  Modern microprocessors are complicated beasts, so signif- icant skepticism is appropriate for any simple answer. That aside, the most likely reason is the full memory barriers required by hazard-pointers readers. Any delays resulting from those memory barriers would make time available to the other hardware thread sharing the core, resulting in greater scalability at the expense of per-hardware-thread performance.

- [**?**] shows that hazard pointers have near-ideal performance. Whatever happened in Figure 3?

2.4 Sequence Locks

The key component of sequence locking is the sequence number, which has an even value in the absence of updaters and an odd value if there is an update in progress. Readers can then snapshot the value before and after each

access. If either snapshot has an odd value, or if the two snapshots differ, there has been a concurrent update, and the reader must discard the results of the access and then retry it. Readers therefore use the `read_seqbegin()` and `read_seqretry()` functions shown in Listing 18 when accessing data protected by a sequence lock. Writers must increment the value before and after each update, and only one writer is permitted at a given time. Writers therefore use the `write_seqlock()` and `write_sequnlock()` functions shown in Listing 19 when updating data protected by a sequence lock.

```
1 do {
2     seq = read_seqbegin(&test_seqlock);
3     /* read-side access. */
4 } while (read_seqretry(&test_seqlock, seq));
```

Listing 18: Sequence-Locking Reader

```
1 write_seqlock(&test_seqlock);
2 /* Update */
3 write_sequnlock(&test_seqlock);
```

Listing 19: Sequence-Locking Writer

Remark. Why not have `read_seqbegin()` in 20 check for the low-order bit being set, and retry internally, rather than allowing a doomed read to start?

That would be a legitimate implementation. However, if the workload is read-mostly, it would likely increase the overhead of the common-case successful read, which could be counter-productive. However, given a sufficiently large fraction of updates and sufficiently high-overhead readers, having the check internal to `read_seqbegin()` might be preferable

Line 17 orders this snapshot before the caller's critical section. Line 26 orders the caller's prior critical section before line 27's fetch.

Remark.

- Why is the `smp_mb()` on line 26 of Listing 20 needed?

  If it was omitted, both the compiler and the CPU would be within their rights to move the critical section Preceding the call to `read_seqretry()` down below this function. This would prevent the sequence lock from protecting the critical section. The `smp_mb()` primitive prevents such reordering.

```
typedef struct {
    unsigned long seq;
    spinlock_t lock;
} seqlock_t;

static inline void seqlock_init(seqlock_t *slp)
{
    slp->seq = 0;
    spin_lock_init(&slp->lock);
}

static inline unsigned long read_seqbegin(seqlock_t *slp)
{
    unsigned long s;

    s = READ_ONCE(slp->seq);
    smp_mb();
    return s & ~0x1UL;
}

static inline int read_seqretry(seqlock_t *slp,
                                unsigned long oldseq)
{
    unsigned long s;

    smp_mb();
    s = READ_ONCE(slp->seq);
    return s != oldseq;
}


static inline void write_seqlock(seqlock_t *slp)
{
    spin_lock(&slp->lock);
    ++slp->seq;
    smp_mb();
}
static inline void write_sequnlock(seqlock_t *slp)
{
    smp_mb();
    ++slp->seq;
    spin_unlock(&slp->lock);
}
```

Listing 20: Sequence-Locking Implementation

- Can't weaker memory barriers be used in the code in Listing 20?
- Why isn't seq on line 2 of Listing 20 unsigned rather than `unsigned long`? After all, if `unsigned` is good enough for the Linux kernel, shouldn't it be good enough for everyone?

  Overflow issue, 32 bit can be easily overflowed

It suffers use-after-free failures. The problem is that the reader might encounter a segmentation violation due to accessing an already-freed structure before `read_seqretry()` has a chance to warn of the concurrent update.

Remark. Can this bug be fixed? In other words, can you use sequence locks as the only synchronization mechanism protecting a linked list supporting concurrent addition, deletion, and lookup?

One trivial way of accomplishing this is to surround all accesses, including the read-only accesses, with `write_ seqlock()` and `write_sequnlock()`. Of course, this solution also prohibits all read-side parallelism, resulting in massive lock contention, and furthermore could just as easily be implemented using simple locking.

If you do come up with a solution that uses `read_ seqbegin()` and `read_seqretry()` to protect read-side accesses, make sure that you correctly handle the following sequence of events:

1. CPU 0 is traversing the linked list, and picks up a pointer to list element A.
2. CPU 1 removes element A from the list and frees it.
3. CPU 2 allocates an unrelated data structure, and gets the memory formerly occupied by element A. In this unrelated data structure, the memory previously used for element A's ->next pointer is now occupied by a floating-point number.
4. CPU 0 picks up what used to be element A's -> next pointer, gets random bits, and therefore gets a segmentation fault.

One way to protect against this sort of problem requires use of "type-safe memory". Roughly similar solutions are possible using the hazard pointers. But in either case, you would be using some other synchronization mechanism in addition to sequence locks!

```
struct route_entry {
    struct route_entry *re_next;
    unsigned long addr;
    unsigned long iface;
    int re_freed;
};

struct route_entry route_list;
DEFINE_SEQ_LOCK(sl);

unsigned long route_lookup(unsigned long addr)
{
    struct route_entry *rep;
    struct route_entry **repp;
    unsigned long ret;
    unsigned long s;

retry:
    s = read_seqbegin(&sl);
    repp = &route_list.re_next;
    do {
        rep = READ_ONCE(*repp);
        if (rep == NULL) {
            if (read_seqretry(&sl, s))
                goto retry;
            return ULONG_MAX;
        }
        repp = &rep->re_next;
    } while (rep->addr != addr);
    if (READ_ONCE(rep->re_freed))
        abort();
    ret = rep->iface;
    if (read_seqretry(&sl, s))
        goto retry;
    return ret;
}
```

Listing 21: Sequence-Locked Pre-BSD Routing Table Lookup (BUGGY)

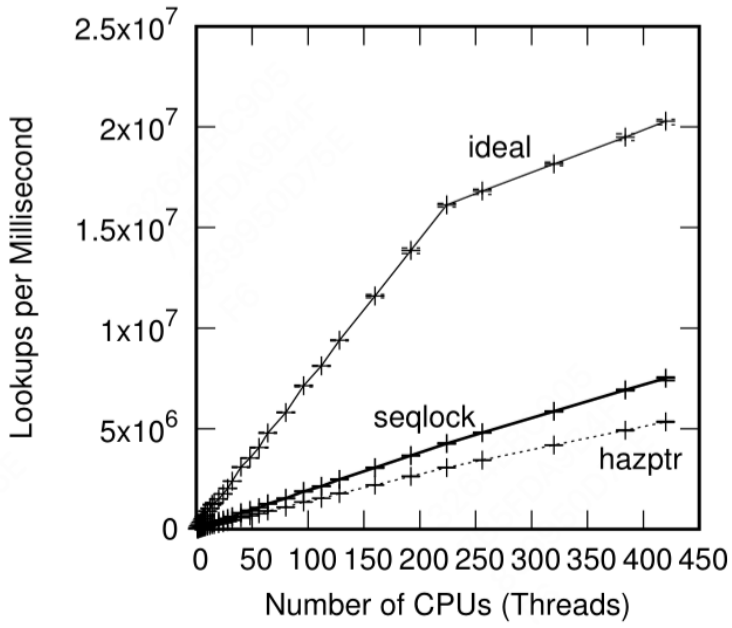

Figure 4: Pre-BSD Routing Table Protected by Sequence Locking

2.5 Read-Copy Update (RCU)

All of the mechanisms discussed in the preceding sections used one of a number of approaches to defer specific actions until they may be carried out safely. The reference counters discussed in Section 2.2 use explicit counters to defer actions that could disturb readers, which results in read-side contention and thus poor scalability. The hazard pointers covered by Section 2.3 uses implicit counters in the guise of per-thread lists of pointer. This avoids read-side contention, but requires readers to do stores and conditional branches, as well as either full memory barriers in read-side primitives or real-time-unfriendly inter-processor interrupts in update-side primitives. The sequence lock presented in Section 2.4 also avoids read-side contention, but does not protect pointer traversals and, like hazard pointers, requires either full memory barriers in read-side primitives, or inter-processor interrupts in update-side primitives.

2.5.1 Introduction to RCU

To minimize implementability concerns, we focus on a minimal data structure, which consists of a single global pointer that is either `NULL` or references a single structure.

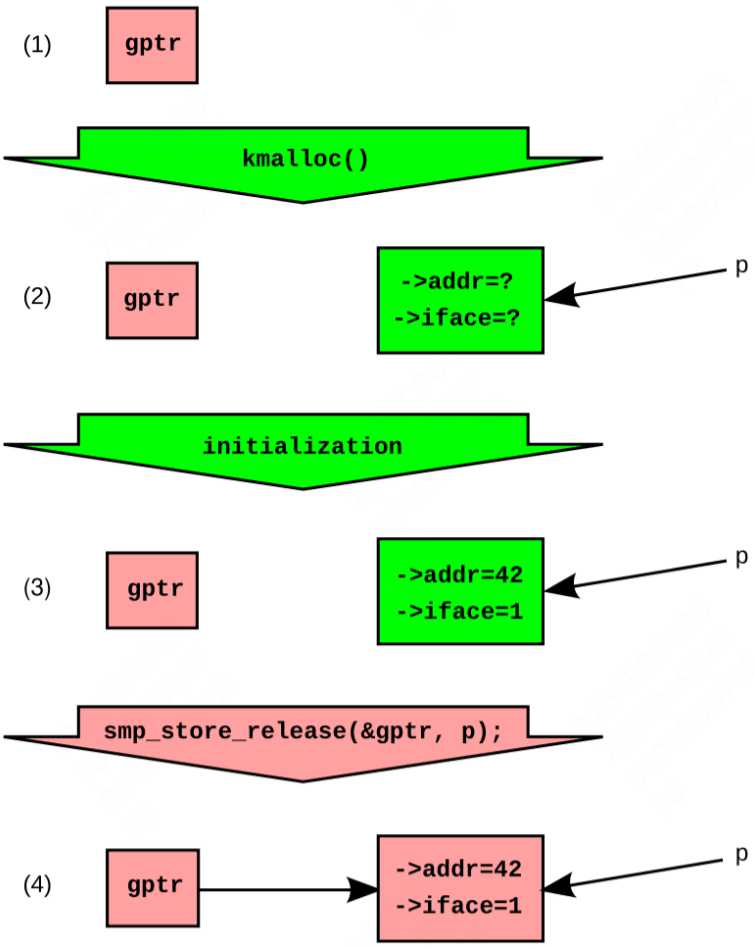

Figure 5: Insertion With Concurrent Readers

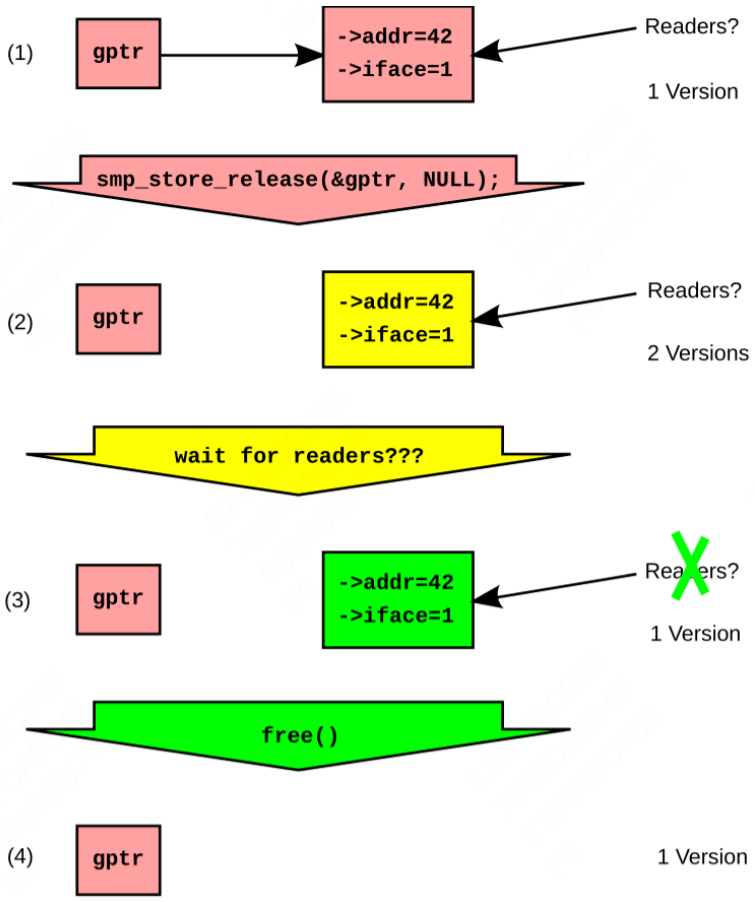

Figure 6: Insertion With Concurrent Readers

Remark. • Why does Figure 6 use `smp_store_release()` given that it is storing a `NULL` pointer? Wouldn't `WRITE_ONCE()` work just as well in this case, given that there is no structure initialization to order against the store of the `NULL` pointer?

Yes, it would.

Because a `NULL` pointer is being assigned, there is nothing to order against, so there is no need for `smp_store_ release()`. In contrast, when assigning a non-`NULL` pointer, it is necessary to use `smp_store_release()` in order to ensure that initialization of the pointed-to structure is carried out before assignment of the pointer.

In short, `WRITE_ONCE()` would work, and would save a little bit of CPU time on some architectures. However, as we will see, software-

engineering concerns will motivate use of a special `rcu_assign_pointer()` that is quite similar to `smp_store_release()`.

- Readers running concurrently with each other and with the procedure outlined in Figure 6 can disagree on the value of `gptr`. Isn't that just a wee bit problematic???

  Not necessarily.

  As hinted at in Sections 3.2.3 and 3.3, speed-of-light delays mean that a computer's data is always stale com- pared to whatever external reality that data is intended to model.

  Real-world algorithms therefore absolutely must tolerate inconsistancies between external reality and the in-computer data reflecting that reality. Many of those algorithms are also able to tolerate some degree of inconsistency within the in-computer data. Section 10.3.4 discusses this point in more detail.

  Please note that this need to tolerate inconsistent and stale data is not limited to RCU. It also applies to reference counting, hazard pointers, sequence locks, and even to some locking use cases. For example, if you compute some quantity while holding a lock, but use that quantity after releasing that lock, you might well be using stale data. After all, the data that quantity is based on might change arbitrarily as soon as the lock is released.

  So yes, RCU readers can see stale and inconsistent data, but no, this is not necessarily problematic. And, when needed, there are RCU usage patterns that avoid both staleness and inconsistency

3 Appendices IGNORE

.1 Why Memory Barriers

.1.1 Cache Structure

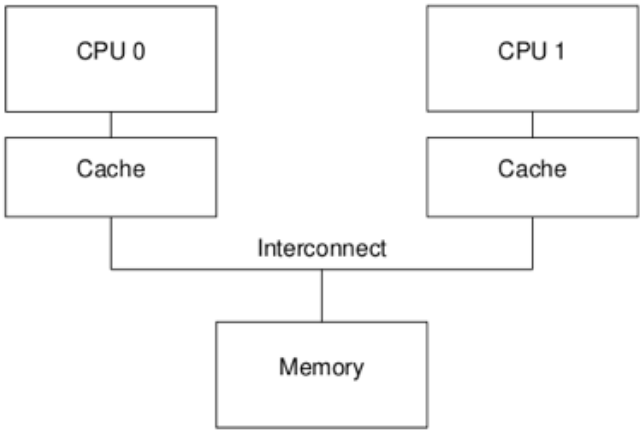

Figure 7: Modern Computer System Cache Structure

Data flows among the CPUs' caches and memory in fixed-length blocks called "cache lines", which are normally a power of two in size, ranging from 16 to 256 bytes. When a given data item is first accessed by a given CPU, it will be absent from that CPU's cache, meaning that a "cache miss" (or, more specifically, a "startup" or "warmup" cache miss) has occurred. The cache miss means that the CPU will have to wait (or be "stalled") for hundreds of cycles while the item is fetched from memory. However, the item will be loaded into that CPU's cache, so that subsequent accesses will find it in the cache and therefore run at full speed.

	Way 0	Way 1
0x0	0x12345000	
0x1	0x12345100	
0x2	0x12345200	
0x3	0x12345300	
0x4	0x12345400	
0x5	0x12345500	
0x6	0x12345600	
0x7	0x12345700	
0x8	0x12345800	
0x9	0x12345900	
0xA	0x12345A00	
0xB	0x12345B00	
0xC	0x12345C00	
0xD	0x12345D00	
0xE	0x12345E00	0x43210E00
0xF		

Figure 8: CPU Cache Structure

This cache has sixteen "sets" and two "ways" for a total of 32 "lines", each entry containing a single 256-byte "cache line", which is a 256-byte-aligned block of memory.

Each box corresponds to a cache entry, which can contain a 256-byte cache line. Since the cache lines must be 256-byte aligned, the low eight bits of each address are zero, and the choice of hardware hash function means that the next-higher four bits match the hash line number.

What happens when it does a write? Because it is important that all CPUs agree on the value of a given data item, before a given CPU writes to that data item, it must first cause it to be removed, or "invalidated", from other CPUs' caches. Once this invalidation has completed, the CPU may safely modify the data item. If the data item was present in this CPU's cache, but was read-only, this process is termed a "write miss". Once a given CPU has completed invalidating a given data item from other CPUs' caches, that CPU may repeatedly write (and read) that data item.

Later, if one of the other CPUs attempts to access the data item, it will incur a cache miss, this time because the first CPU invalidated the item in order to write to it. This type of cache miss is termed a "communication

miss", since it is usually due to several CPUs using the data items to communicate (for example, a lock is a data item that is used to communicate among CPUs using a mutual-exclusion algorithm).

Clearly, much care must be taken to ensure that all CPUs maintain a coherent view of the data. With all this fetching, invalidating, and writing, it is easy to imagine data being lost or (perhaps worse) different CPUs having conflicting values for the same data item in their respective caches.

.1.2 Cache-Coherence Protocols

1. MESI States MESI stands for "modified", "exclusive", "shared", and "invalid", the four states a given cache line can take on using this protocol.