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

Paul E. McKenny

August 12, 2024

Contents

1 Deferred Processing

1.1 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.

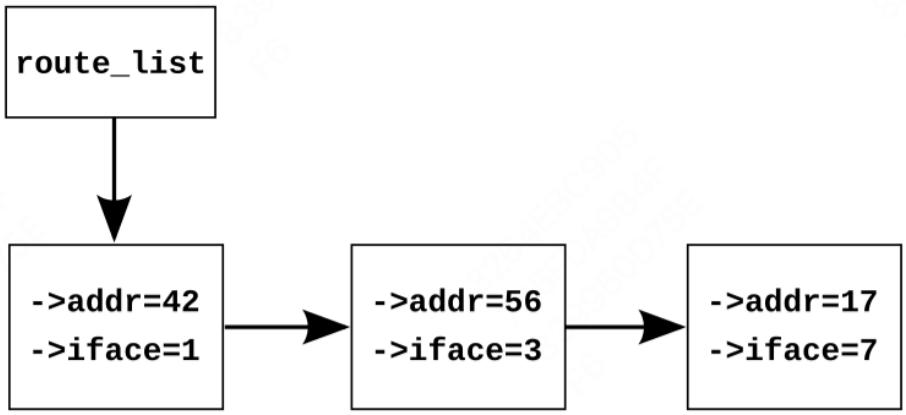

Figure 1: Pre-BSD Packet Routing List

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

1.2 Reference Counting

Starting with Listing **??**, 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.

```
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 1: Sequential Pre-BSD Routing Table

```
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 2: 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 3: Reference-Counted Pre-BSD Routing Table Add/Delete (BUGGY)

2 Appendices IGNORE

.1 Why Memory Barriers

.1.1 Cache Structure

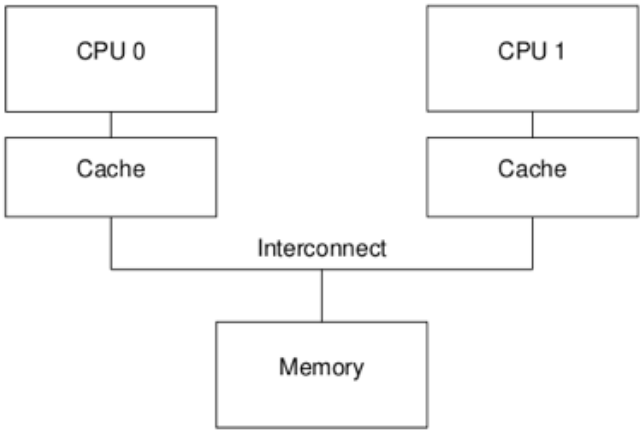

Figure 2: Modern Computer System Cache Structure

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 3: 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.