Case study: An example on non-linear effects of scheduling: "Receive livelocks in interrupt-driven kernel"

- Interrupt-driven kernel : triggers high-priority interrupt on every receive event.
- Interrupt handler processes the received message (network packet) and adds it to the desired queue for further processing (e.g., sending a reply). Thus the interrupt handler acts as a producer for this queue.
- Another lower-priority thread consumes from the queue. However, the lower priority thread can get interrupted if the network device generates another interrupt for a received packet.
- If the rate of received packets is high, the lower-priority consumer thread will never get to run, resulting in packets getting dropped at the queue, and a total of zero throughput.
- Draw the curve between input packet rate and the throughput of the kernel.
- Better to have polled the network device -- excess input rate would cause a drop at the network device queue (no wasted work!)
- Result: Graceful saturation of throughput with increasing input network rate.
- Tradeoff between polling and interrupts
    - Interrupts provide lower latency, and less overhead at small input rates. But behave poorly at high input rates. e.g., telephone calls
    - Polling provides higher throughput at high input rates. But results in very high latencies for low input rates. e.g., emails

Multi-processor coordination without locks

How to avoid using locks. Even with scalable locks, their use can be expensive. Let's start with a simple data structure and try to allow for concurrent access, so that we get a feel for the issues.

Searchable stack using locks

```
struct element {
    int key;
    int value;
    struct element *next;
};

struct element *top;

void push(struct element *e) {
    e->next = top;
    top = e;
}

struct element *pop(void) {
    struct element *e = top;
    top = e->next;
    return e;
}

int search(int key) {
    struct element *e = top;
    while (e) {
 if (e->key == key)
     return e->value;
        e = e->next;
    }
    return -1;
}
```

This is clearly not going to work on a concurrent system

global spinlock: performance how many concurrent ops? just one CPU at a time bus interactions? bounce cache line for both reads, writes

global read-write lock: performance

- how many concurrent ops? theoretically, could do one per CPU
- bus interactions? bounce cache line for both reads, writes
- draw timing diagram of CPU interactions
- we might be slower on two CPUs than on a single CPU!

is it going to get better if we allocate a read-write lock for each item?

- concurrency still possible
- but penalty even worse: N_{elem} cache line bounces for each search traversal

more scalable locking schemes

- example: brlock
- N cpus (or threads) would have N locks, one per CPU
- readers lock their CPU's lock, writers must lock each CPU's lock
- must use up N 64-byte cache lines, even though each lock is just a bit.. (why? because anything at a finer granularity will still cause cache bouncing)

Avoiding locks

why do we want to avoid locks?

- performance
- complexity
- deadlock
- priority inversion

Recall compare-and-swap:

```
int cmpxchg(int *addr, int old, int new) {
    int was = *addr;
    if (was == old)
 *addr = new;
    return was;
}
```

Usage:

```
void push(struct element *e) {
   do {
    e->next = top;
 } while (cmpxchg(&top, e->next, e) != e->next);
}

struct element *pop(void) {
   do {
    struct element *e = top;
 } while (cmpxchg(&top, e, e->next) != e);
  return e;
}
```

What about search? Can be the same as in the non-concurrent case?

But before that: why is this better than not having locks?

- readers no longer generate spurious updates, which incurred performance hit
- not having to think about deadlock is great

problem 1: lock-free data structures require careful design, hw support

- suppose we want to remove arbitrary elements, not just the first one
- what could go wrong if we try to use the same cmpxchg?
- need DCAS (double-compare-and-swap) to implement remove properly
  - must make sure neither previous nor next element changed
- x86 hardware doesn't have DCAS
  - can do some tricks because x86 provides 8-byte cmpxchg
  - only for sequential memory addresses
  - can potentially play tricks by staggering the 8 bytes across two pages
  - unlikely to win in performance once we start invalidating TLB entries

problem 2: memory reuse
when can we free a memory block, if other CPUs could be accessing it? other CPU's search might be traversing any of the elements

- Solution: we could try bumping a refcount on each access, but that introduces cacheline bounces
  - and what about CPUs that are just about to bump the refcount?

What if the memory is freed (and reused) while a search() thread is still holding a reference to it?

Worse, reusing a memory block can corrupt list

- stack contains three elements
  top -> A -> B -> C
- CPU 1 about to pop off the top of the stack, preempted just before cmpxchg(&top, A, B)

- CPU 2 pops off A, B, frees both of them. top -> C
- CPU 2 allocates another item (malloc reuses A) and pushes onto stack top -> A -> C
- CPU 1: cmpxchg succeeds, stack now looks like top -> B -> C
- This is not right: it should have looked as if two pops, one push and one pop have occurred (i.e., three pops and one push to leave one element on the stack). The problem occurred because A was reused and cmpxchg only relies on the address and not on the logical element for disambiguation.

Memory reuse should be prevented till one is sure that no thread could be holding a reference to it (in its stack or registers), i.e., free() should be prevented till the time any thread could be holding a reference to something that was just popped off.

But how do we know about when it is safe to reuse memory? Would waiting for a long time (say one hour) before freeing work? Well almost -- at least the probability that another thread could be holding a reference to such a location becomes smaller with time, and so the probability of a failure is very low. How does one decide how long to wait before reusing memory? If too small, can result in incorrect behaviour. If too large, cannot reuse memory for a long time, which can be a problem is the available memory is small.