

Outline

- 1 Cache coherence – the hardware view
- 2 Synchronization and memory consistency review
- 3 C11 Atomics
- 4 Avoiding locks

1 / 47

Important memory system properties

- **Coherence** – concerns accesses to a single memory location
 - There is a total order on all updates
 - Must obey program order if access from only one CPU
 - There is bounded latency before everyone sees a write
- **Consistency** – concerns ordering across memory locations
 - Even with coherence, different CPUs can see the same write happen at different times
 - Sequential consistency is what matches our intuition (As if operations from all CPUs interleaved on one CPU)
 - Many architectures offer weaker consistency
 - Yet well-defined weaker consistency can still be sufficient to implement [thread API contract from concurrency lecture](#)

2 / 47

Multicore cache coherence

- **Performance requires caches**
 - Divided into chunks of bytes called lines (e.g., 64 bytes)
 - Caches create an opportunity for cores to disagree about memory
- **Bus-based approaches**
 - “Snoopy” protocols, each CPU listens to memory bus
 - Use write-through and invalidate when you snoop a memory write
 - Bus-based schemes limit scalability
- **Modern CPUs use networks (e.g., AMD infinity fabric, intel UPI, CXL [between CPUs and devices])**
 - CPUs pass each other messages about cache lines

3 / 47

MESI coherence protocol

- **Modified**
 - Exactly one cache has a valid copy
 - That copy is dirty (needs to be written back to memory)
 - Must invalidate all copies in other caches before entering this state
- **Exclusive**
 - Same as Modified except the cache copy is clean
- **Shared**
 - One or more caches and memory have a valid copy
- **Invalid**
 - Doesn't contain any data
- **Owned (for enhanced “MOESI” protocol)**
 - Cached copy may be dirty (like Modified state)
 - But have to broadcast modifications (sort of like Shared state)
 - Can have one owned + multiple shared copies of cache line

4 / 47

Core and Bus Actions

- **Actions performed by CPU core**
 - Read
 - Write
 - Evict (modified/owned? must write back)
- **Transactions on bus (or interconnect)**
 - Read: without intent to modify, data can come from memory or another cache
 - Read-exclusive: with intent to modify, must invalidate all other cache copies
 - Writeback: contents put on bus and memory is updated

5 / 47

cc-NUMA

- **Old machines used *dance hall* architectures**
 - Any CPU can “dance with” any memory equally
- **An alternative: Non-Uniform Memory Access (NUMA)**
 - Each CPU has fast access to some “close” memory
 - Slower to access memory that is farther away
 - Use a directory to keep track of who is caching what
- **Originally for esoteric machines with many CPUs**
 - But AMD and then intel integrated memory controller into CPU
 - Faster to access memory controlled by the local socket (or even local die in a multi-chip module)
- **cc-NUMA = cache-coherent NUMA**
 - Rarely see non-cache-coherent NUMA (BBN Butterfly 1, Cray T3D)

6 / 47

Real World Coherence Costs

- See [David] for a great reference. Xeon results:
 - 3 cycle L1, 11 cycle L2, 44 cycle LLC, 355 cycle local RAM
- If another core in same socket holds line in modified state:
 - load: 109 cycles (LLC + 65)
 - store: 115 cycles (LLC + 71)
 - atomic CAS: 120 cycles (LLC + 76)
- If a core in a different socket holds line in modified state:
 - NUMA load: 289 cycles
 - NUMA store: 320 cycles
 - NUMA atomic CAS: 324 cycles
- But only a partial picture
 - Could be faster because of out-of-order execution
 - Could be slower if interconnect contention or multiple hops

7/47

NUMA and spinlocks

- Test-and-set spinlock has several advantages
 - Simple to implement and understand
 - One memory location for arbitrarily many CPUs
- But also has disadvantages
 - Lots of traffic over memory interconnect (especially w. > 1 spinner)
 - Not necessarily fair (lacks bounded waiting)
 - Even less fair on a NUMA machine
- Idea 1: Avoid spinlocks altogether (today)
- Idea 2: Reduce interconnect traffic with better spinlocks (next lecture)
 - Design lock that spins only on local memory
 - Also gives better fairness

8/47

Outline

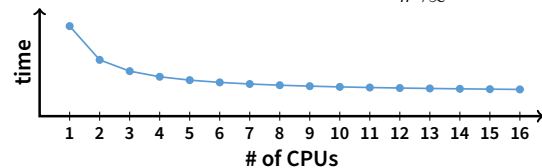
- 1 Cache coherence – the hardware view
- 2 Synchronization and memory consistency review
- 3 C11 Atomics
- 4 Avoiding locks

9/47

Amdahl's law

$$T(n) = T(1) \left(B + \frac{1}{n}(1 - B) \right)$$

- Expected speedup limited when only part of a task is sped up
 - $T(n)$: the time it takes n CPU cores to complete the task
 - B : the fraction of the job that must be serial
- Even with massive multiprocessors, $\lim_{n \rightarrow \infty} = B \cdot T(1)$



- Places an ultimate limit on parallel speedup
- Problem: synchronization increases serial section size

10/47

Locking basics

```
mutex_t m;

lock(&m);
cnt = cnt + 1; /* critical section */
unlock(&m);
```

- Only one thread can hold a mutex at a time
 - Makes critical section atomic
- Recall thread API contract
 - All access to global data must be protected by a mutex
 - Global = two or more threads touch data and at least one writes
- Means must map each piece of global data to one mutex
 - Never touch the data unless you locked that mutex
- But many ways to map data to mutexes

11/47

Locking granularity

- Consider two lookup implementations for global hash table:

```
struct list *hash_tbl[1021];
```

coarse-grained locking

```
mutex_t m;
:
:
mutex_lock(&m);
struct list_elem *pos = list_begin (hash_tbl[hash(key)]);
/* ... walk list and find entry ... */
mutex_unlock(&m);
```

fine-grained locking

```
mutex_t bucket_lock[1021];
:
:
int index = hash(key);
mutex_lock(&bucket_lock[index]);
struct list_elem *pos = list_begin (hash_tbl[index]);
/* ... walk list and find entry ... */
mutex_unlock(&bucket_lock[index]);
```

- Which implementation is better?

12/47

Locking granularity (continued)

- **Fine-grained locking admits more parallelism**
 - E.g., imagine network server looking up values in hash table
 - Parallel requests will usually map to different hash buckets
 - So fine-grained locking should allow better speedup
- **When might coarse-grained locking be better?**

13 / 47

Locking granularity (continued)

- **Fine-grained locking admits more parallelism**
 - E.g., imagine network server looking up values in hash table
 - Parallel requests will usually map to different hash buckets
 - So fine-grained locking should allow better speedup
 - **When might coarse-grained locking be better?**
 - Suppose you have global data that applies to whole hash table
- ```
struct hash_table {
 size_t num_elements; /* num items in hash table */
 size_t num_buckets; /* size of buckets array */
 struct list *buckets; /* array of buckets */
};
```
- Read `num_buckets` each time you insert
  - Check `num_elements` on insert, possibly expand buckets & rehash
  - Single global mutex would protect these fields
- **Can you avoid serializing lookups to growable hash table?**

13 / 47

## Readers-writers problem

- **Recall a `mutex` allows access in only one thread**
- **But a data race occurs only if**
  - Multiple threads access the same data, **and**
  - At least one of the accesses is a write
- **How to allow multiple readers or one single writer?**
  - Need lock that can be *shared* amongst concurrent readers
- **Can implement using other primitives (next slides)**
  - Keep integer `i` - # of readers or -1 if held by writer
  - Protect `i` with `mutex`
  - Sleep on condition variable when can't get lock

14 / 47

## Implementing shared locks

```
struct sharedlk {
 int i; /* # shared lockers, or -1 if exclusively locked */
 mutex_t m;
 cond_t c;
};

void AcquireExclusive (sharedlk *sl) {
 lock (&sl->m);
 while (sl->i) { wait (&sl->m, &sl->c); }
 sl->i = -1;
 unlock (&sl->m);
}

void AcquireShared (sharedlk *sl) {
 lock (&sl->m);
 while (&sl->i < 0) { wait (&sl->m, &sl->c); }
 sl->i++;
 unlock (&sl->m);
}
```

15 / 47

## Implementing shared locks (continued)

```
void ReleaseShared (sharedlk *sl) {
 lock (&sl->m);
 if (!--sl->i)
 signal (&sl->c);
 unlock (&sl->m);
}

void ReleaseExclusive (sharedlk *sl) {
 lock (&sl->m);
 sl->i = 0;
 broadcast (&sl->c);
 unlock (&sl->m);
}
```

- **Any issues with this implementation?**

16 / 47

## Implementing shared locks (continued)

```
void ReleaseShared (sharedlk *sl) {
 lock (&sl->m);
 if (!--sl->i)
 signal (&sl->c);
 unlock (&sl->m);
}

void ReleaseExclusive (sharedlk *sl) {
 lock (&sl->m);
 sl->i = 0;
 broadcast (&sl->c);
 unlock (&sl->m);
}
```

- **Any issues with this implementation?**
  - Prone to starvation of writer (no bounded waiting)
  - Lots of coherence traffic to acquire shared lock
  - How might you fix these problems?

16 / 47

## Review: Test-and-set spinlock

```
struct var {
 int lock;
 int val;
};

void atomic_inc (var *v) {
 while (test_and_set (&v->lock))
 ;
 v->val++;
 v->lock = 0;
}

void atomic_dec (var *v) {
 while (test_and_set (&v->lock))
 ;
 v->val--;
 v->lock = 0;
}
```

- Is this code correct without sequential consistency?

17/47

## Memory reordering danger

- Suppose no sequential consistency (& don't compensate)
- Hardware could violate program order

| Program order on CPU #1                                                                    | View on CPU #2                                                                     |
|--------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------|
| <pre>v-&gt;lock = 1; register = v-&gt;val; v-&gt;val = register + 1; v-&gt;lock = 0;</pre> | <pre>v-&gt;lock = 1; v-&gt;lock = 0; /* danger */; v-&gt;val = register + 1;</pre> |

- If atomic\_inc called at */\* danger \*/*, bad val ensues!

18/47

## Ordering requirements

```
void atomic_inc (var *v) {
 while (test_and_set (&v->lock))
 ;
 v->val++;
 /* danger */
 v->lock = 0;
}
```

- Must ensure all CPUs see the following:
  1. v->lock = 1 ran before v->val was read and written
  2. v->lock = 0 ran after v->val was written
- How does #1 get assured on x86?
  - Recall test\_and\_set uses xchgl %eax, (%edx)
- How to ensure #2 on x86?

19/47

## Ordering requirements

```
void atomic_inc (var *v) {
 while (test_and_set (&v->lock))
 ;
 v->val++;
 /* danger */
 v->lock = 0;
}
```

- Must ensure all CPUs see the following:
  1. v->lock = 1 ran before v->val was read and written
  2. v->lock = 0 ran after v->val was written
- How does #1 get assured on x86?
  - Recall test\_and\_set uses xchgl %eax, (%edx)
  - xchgl instruction always "locked," ensuring barrier
- How to ensure #2 on x86?

19/47

## Ordering requirements

```
void atomic_inc (var *v) {
 while (test_and_set (&v->lock))
 ;
 v->val++;
 asm volatile ("sfence" ::: "memory");
 v->lock = 0;
}
```

- Must ensure all CPUs see the following:
  1. v->lock = 1 ran before v->val was read and written
  2. v->lock = 0 ran after v->val was written
- How does #1 get assured on x86?
  - Recall test\_and\_set uses xchgl %eax, (%edx)
  - xchgl instruction always "locked," ensuring barrier
- How to ensure #2 on x86?
  - Might need fence instruction after, e.g., non-temporal stores
  - Definitely need compiler barrier

19/47

## Gcc extended asm syntax [gnu]

```
asm volatile (template-string : outputs : inputs : clobbers);
```

- Puts *template-string* in assembly language compiler output
  - Expands %0, %1, ... (a bit like printf conversion specifiers)
  - Use "%%" for a literal % (e.g., "%%cr3" to specify %cr3 register)
- *inputs/outputs* specify parameters as "**constraint**" (*value*)

```
int outvar, invar = 3;
asm ("movl %1, %0" : "=r" (outvar) : "r" (invar));
/* now outvar == 3 */
```
- *clobbers* lists other state that get used/overwritten
  - Special value "memory" prevents reordering with loads & stores
  - Serves as *compiler barrier*, as important as hardware barrier
- *volatile* indicates **side effects other than result**
  - Otherwise, gcc might optimize away if you don't use result

20/47

## Correct spinlock on alpha

- Recall implementation of `test_and_set` on alpha (with much weaker memory consistency than x86):

```
_test_and_set:
 ldq_l v0, 0(a0) # v0 = *lockp (LOCKED)
 bne v0, 1f # if (v0) return
 addq zero, 1, v0 # v0 = 1
 stq_c v0, 0(a0) # *lockp = v0 (CONDITIONAL)
 beq v0, _test_and_set # if (failed) try again
 mb
 addq zero, zero, v0 # return 0
1: ret zero, (ra), 1
```

- **Memory barrier instruction `mb` (like `mfence`)**
  - All processors will see that everything before `mb` in program order happened before everything after `mb` in program order
- **Need barrier before releasing spinlock as well:**

```
asm volatile ("mb" ::: "memory");
v->lock = 0;
```

21 / 47

## Memory barriers/fences

- Fortunately, consistency need not overly complicate code
  - If you do locking right, only need a few fences within locking code
  - Code will be easily portable to new CPUs
- Most programmers should stick to mutexes
- But advanced techniques may require lower-level code
  - Later this lecture will see some wait-free algorithms
  - Also important for optimizing special-case locks (E.g., linux kernel `rw_semaphore`, ...)
- Algorithms often explained assuming sequential consistency
  - Must know how to use memory fences to implement correctly
  - E.g., see [Howells] for how Linux deals with memory consistency
  - And another plug for [Why Memory Barriers](#)
- Next: How C11 allows portable low-level code

22 / 47

## Outline

- 1 Cache coherence – the hardware view
- 2 Synchronization and memory consistency review
- 3 C11 Atomics
- 4 Avoiding locks

23 / 47

## Atomics and portability

- Lots of variation in atomic instructions, consistency models, compiler behavior
  - Changing the compiler or optimization level can invalidate code
- Different CPUs today: Many laptops (not Apple) are x86, while your cell phone uses ARM
  - x86: Total Store Order Consistency Model, CISC
  - arm: Relaxed Consistency Model, RISC
- Could make it impossible to write portable kernels and applications
- Fortunately, the C11 standard has builtin support for atomics
- Also available in C++11, but won't discuss today
  - But use C++ in production, because C atomics mostly follow C++

24 / 47

## Background: C memory model [C11]

- Within a thread, many evaluations are *sequenced*
  - E.g., in "`f1(); f2();`", evaluation of `f1` is sequenced before `f2`
- Across threads, some operations *synchronize with others*
  - E.g., releasing mutex `m` synchronizes with a subsequent acquire `m`
- Evaluation *A happens before B*, which we'll write  $A \rightarrow B$ , when:
  - *A* is sequenced before *B* (in the same thread),
  - *A* synchronizes with *B*,
  - *A* is dependency-ordered before *B* (ignore for now—means *A* has release semantics and *B* consume semantics for same value), or
  - There is another operation *X* such that  $A \rightarrow X \rightarrow B$ .<sup>1</sup>

## C11 Atomics: Big picture

- C11 says a *data race* produces **undefined behavior (UB)**
  - A write *conflicts* with a read or write of same memory location
  - Two conflicting operations *race* if not ordered by happens before
  - Undefined can be anything (e.g., delete all your files, ...)
  - Think UB okay in practice? See [Wang], [Lattner]
- Spinlocks (and hence mutexes that internally use spinlocks) **synchronize across threads**
  - Synchronization adds happens before arrows, avoiding data races
- Yet hardware supports other means of synchronization
- C11 atomics provide direct access to synchronized lower-level operations
  - E.g., can get compiler to issue x86 `lock` prefix in some cases

<sup>1</sup>Except chain of " $\rightarrow$ " cannot end: ..., dependency-ordered, sequenced before

25 / 47

26 / 47

## C11 Atomics: Basics

- Include new `<stdatomic.h>` header
- New `_Atomic` type qualifier: e.g., `_Atomic int foo;`
  - Convenient aliases: `atomic_bool`, `atomic_int`, `atomic_ulong`, ...
  - Must initialize specially:

```
#include <stdatomic.h>
_Atomic int global_int = ATOMIC_VAR_INIT(140);
 :
 Atomic(int) *dyn = malloc(sizeof(*dyn));
 atomic_init(dyn, 140);
```
- Compiler emits read-modify-write instructions for atomics
  - E.g., `+=`, `-=`, `|=`, `&=`, `^=`, `++`, `--` do what you would hope
  - Act atomically and synchronize with one another
- Also functions including `atomic_fetch_add`, `atomic_compare_exchange_strong`, ...

27 / 47

## Locking and atomic flags

- Implementations might use spinlocks internally for most atomics
  - Could interact badly with interrupt/signal handlers
  - Can check if `ATOMIC_INT_LOCK_FREE`, etc., macros defined
  - Fortunately modern CPUs don't require this
- `atomic_flag` is a special type guaranteed lock-free
  - Boolean value without support for loads and stores
  - Initialize with: `atomic_flag mylock = ATOMIC_FLAG_INIT;`
  - Only two kinds of operation possible:
    - `_Bool atomic_flag_test_and_set(volatile atomic_flag *obj);`
    - `void atomic_flag_clear(volatile atomic_flag *obj);`
  - Above functions guarantee sequential consistency (atomic operation serves as memory fence, too)

28 / 47

## Exposing weaker consistency

```
enum memory_order { /*...*/ };

_Bool atomic_flag_test_and_set_explicit(
 volatile atomic_flag *obj, memory_order order);
void atomic_flag_clear_explicit(
 volatile atomic_flag *obj, memory_order order);

C atomic_load_explicit(
 const volatile A *obj, memory_order order);
void atomic_store_explicit(
 volatile A *obj, C desired, memory_order order);

bool atomic_compare_exchange_weak_explicit(
 A *obj, C *expected, C desired,
 memory_order succ, memory_order fail);
```

- Atomic functions have `_explicit` variants
  - These guarantee coherence but *not* sequential consistency
  - May allow compiler to generate faster code

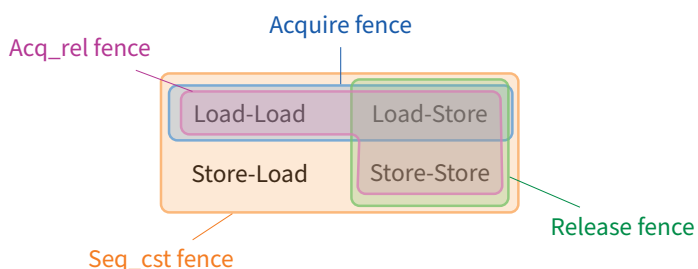
29 / 47

## Memory ordering

- Six possible `memory_order` values:
  1. `memory_order_relaxed`: no memory ordering
  2. `memory_order_consume`: super tricky, see [Preshing] for discussion
  3. `memory_order_acquire`: for start of critical section
  4. `memory_order_release`: for end of critical section
  5. `memory_order_acq_rel`: combines previous two
  6. `memory_order_seq_cst`: full sequential consistency
- Also have fence operation not tied to particular atomic:  
`void atomic_thread_fence(memory_order order);`
- Suppose thread 1 releases and thread 2 acquires
  - Thread 1's preceding accesses can't move past **release** store
  - Thread 2's subsequent accesses can't move before **acquire** load
  - Warning: other threads might see a completely different order

30 / 47

## Types of memory fence<sup>2</sup>



- X-Y fence = operations of type X sequenced before the fence happen before operations of type Y sequenced after the fence

<sup>2</sup>Credit to [Preshing] for explaining it this way

31 / 47

## Example: Atomic counters

```
_Atomic(int) packet_count;

void recv_packet(...)
{
 atomic_fetch_add_explicit(&packet_count, 1,
 memory_order_relaxed);
 :
}

void reset_counter()
{
 atomic_store_explicit(&packet_count, 0,
 memory_order_relaxed);
}
```

- Need to count packets accurately
- Don't need to order other memory accesses across threads
- Relaxed memory order can avoid unnecessary overhead
  - On x86, `recv_packet` doesn't benefit, but `reset_counter` uses `mov` compared to `xchgl` for `packet_count = 0`

32 / 47

## Example: Producer, consumer 1

```
struct message msg_buf;
_Atomic_Bool msg_ready;

void send(struct message *m) {
 msg_buf = *m;
 atomic_thread_fence(memory_order_release);
 /* Prior loads+stores happen before subsequent stores */
 atomic_store_explicit(&msg_ready, 1,
 memory_order_relaxed);
}

struct message *recv(void) {
 _Bool ready = atomic_load_explicit(&msg_ready,
 memory_order_relaxed);
 if (!ready)
 return NULL;
 atomic_thread_fence(memory_order_acquire);
 /* Prior loads happen before subsequent loads+stores */
 return &msg_buf;
}
```

33 / 47

## Example: Producer, consumer 2

```
struct message msg_buf;
_Atomic_Bool msg_ready;

void send(struct message *m) {
 msg_buf = *m;
 atomic_store_explicit(&msg_ready, 1,
 memory_order_release);
}

struct message *recv(void) {
 _Bool ready = atomic_load_explicit(&msg_ready,
 memory_order_acquire);
 if (!ready)
 return NULL;
 return &msg_buf;
}
```

- This is potentially faster than previous example
  - E.g., atomic other stores after send can be moved before msg\_buf
- Also, thread sanitizer doesn't understand fences

34 / 47

## Example: Test-and-set spinlock

```
void
spin_lock(atomic_flag *lock)
{
 while(atomic_flag_test_and_set_explicit(lock,
 memory_order_acquire))
 ;
}

void
spin_unlock(atomic_flag *lock)
{
 atomic_flag_clear_explicit(lock, memory_order_release);
}
```

35 / 47

## Example: Better test-and-set spinlock

```
void spin_lock(atomic_bool *lock)
{
 while(atomic_exchange_explicit(lock, 1,
 memory_order_acquire)) {
 while(atomic_load_explicit(lock, memory_order_relaxed)) {
 #if __i386 || __x86_64
 __builtin_ia32_pause();
 #elif __ARM_ARCH >= 7 && __clang__
 __builtin_arm_yield();
 #endif
 }
 }
}

void spin_unlock(atomic_bool *lock)
{
 atomic_store_explicit(lock, 0, memory_order_release);
}
```

- See [\[Rigtorp\]](#) for a good discussion

36 / 47

## Outline

- 1 Cache coherence – the hardware view
- 2 Synchronization and memory consistency review
- 3 C11 Atomics
- 4 **Avoiding locks**

37 / 47

## Recall producer/consumer (lecture 3)

```
/* PRODUCER */
for (;;) {
 item *nextProduced
 = produce_item ();

 mutex_lock (&mutex);
 while (count == BUF_SIZE)
 cond_wait (&nonfull,
 &mutex);

 buffer[in] = nextProduced;
 in = (in + 1) % BUF_SIZE;
 count++;
 cond_signal (&nonempty);
 mutex_unlock (&mutex);
}

/* CONSUMER */
for (;;) {
 mutex_lock (&mutex);
 while (count == 0)
 cond_wait (&nonempty,
 &mutex);

 nextConsumed = buffer[out];
 out = (out + 1) % BUF_SIZE;
 count--;
 cond_signal (&nonfull);
 mutex_unlock (&mutex);

 consume_item (nextConsumed);
}
}
```

38 / 47

## Eliminating locks

- One use of locks is to coordinate multiple updates of single piece of state
- How to remove locks here?
  - Factor state so that each variable only has a single writer
- Producer/consumer example revisited
  - Assume one producer, one consumer
  - Why do we need count variable, written by both? To detect buffer full/empty
  - Have producer write in, consumer write out (both `_Atomic`)
  - Use in/out to detect buffer state (sacrifice one buffer slot to distinguish completely full and empty)
  - But note next example busy-waits, which is less good

39 / 47

## Lock-free producer/consumer

```
atomic_int in, out;

void producer (void *ignored) {
 for (;;) {
 item *nextProduced = produce_item ();
 while (((in + 1) % BUF_SIZE) == out) thread_yield ();
 buffer[in] = nextProduced;
 in = (in + 1) % BUF_SIZE;
 }
}

void consumer (void *ignored) {
 for (;;) {
 while (in == out) thread_yield ();
 nextConsumed = buffer[out];
 out = (out + 1) % BUF_SIZE;
 consume_item (nextConsumed);
 }
}
```

[Note fences not needed because no relaxed atomics]

40 / 47

## Version with relaxed atomics

```
void producer (void *ignored) {
 int slot = atomic_load(&in);
 for (;;) {
 item *nextProduced = produce_item ();
 int next = (slot + 1) % BUF_SIZE;
 while (atomic_load_explicit(&out, memory_order_acquire) ==
 next) // Could you use relaxed? ~~~~~
 thread_yield();
 buffer[slot] = nextProduced;
 atomic_store_explicit(&in, next, memory_order_release);
 slot = next;
 }
}

void consumer (void *ignored) {
 // Use memory_order_acquire to load in (for latest buffer[myin])
 // Use memory_order_release to store out
}
```

41 / 47

## Non-blocking synchronization

- Design algorithm to avoid critical sections
  - Any threads can make progress if other threads are preempted
  - Which wouldn't be the case if preempted thread held a lock
- Requires that hardware provide the right kind of atomics
  - Simple test-and-set is insufficient
  - Atomic compare and swap is good: CAS (mem, old, new) If `*mem == old`, then swap `*mem ← new` and return true, else false
- Can implement many common data structures
  - Stacks, queues, even hash tables
- Can implement any algorithm on right hardware
  - Need operation such as atomic compare and swap (has property called *consensus number* =  $\infty$  [Herlihy])
  - Entire kernels have been written without locks [Greenwald]

42 / 47

## Example: non-blocking stack

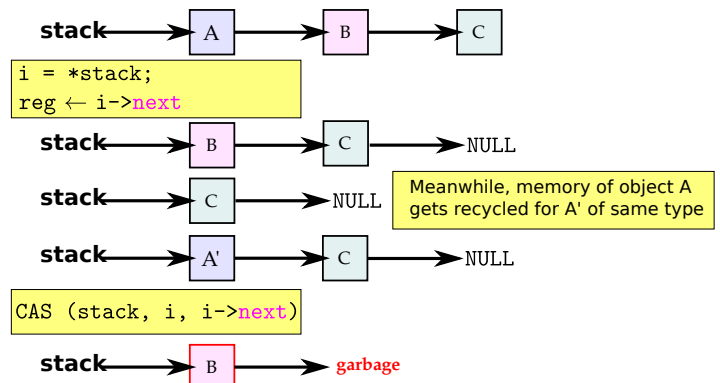
```
struct item {
 /* data */
 _Atomic (struct item *) next;
};
typedef _Atomic (struct item *) stack_t;

void atomic_push (stack_t *stack, item *i) {
 do {
 i->next = *stack;
 } while (!CAS (stack, i->next, i));
}

item *atomic_pop (stack_t *stack) {
 item *i;
 do {
 i = *stack;
 } while (!CAS (stack, i, i->next));
 return i;
}
```

43 / 47

## Wait-free stack issues



- “ABA” race in pop if other thread pops, re-pushes i
  - Can be solved by counters or hazard pointers to delay re-use

44 / 47

## “Benign” races

- Could also eliminate locks by having race conditions
- Maybe you think you care more about speed than correctness
- Maybe you think you can get away with the race (**NOT!**, really)

```
++hits; /* each time someone accesses web site */
if (!initialized) {
 lock (m);
 if (!initialized) {
 initialize ();
 atomic_thread_fence (memory_order_release); /* why? */
 initialized = 1;
 }
 unlock (m);
}
```

- But don't do this [Vyukov], [Boehm]! Not benign at all
  - Again, UB really bad! Like use-after free or array overflow bad
  - If needed for efficiency, use relaxed-memory-order atomics

45 / 47

## Read-copy update [McKenney]

- Some data is read way more often than written
  - Routing tables consulted for each forwarded packet
  - Data maps in system with 100+ disks (updated on disk failure)
- Optimize for the common case of reading without lock
  - E.g., global variable: `routing_table *rt;`
  - Call `lookup (rt, route);` with no lock
- Update by making copy, swapping pointer

```
routing_table *newrt = copy_routing_table (rt);
update_routing_table (newrt);
atomic_thread_fence (memory_order_release);
rt = newrt;
```
- Is RCU really safe? Stay tuned next lecture...

46 / 47

## Next class

- The exciting conclusion of RCU
  - Spoiler: safe on all architectures except on alpha
- Building a better spinlock
- What interface should kernel provide for sleeping locks?
- Deadlock
- Scalable interface design

47 / 47