Overview of previous and current lectures

- **Locks create serial code**
  - Serial code gets no speedup from multiprocessors
- **Test-and-set spinlock has additional disadvantages**
  - Lots of traffic over memory bus
  - Not fair on NUMA machines
- **Idea 1: Avoid spinlocks**
  - We saw lock-free algorithms last lecture
  - Mentioned RCU last time, dive deeper today
- **Idea 2: Design better spinlocks**
  - Less memory traffic, better fairness
- **Idea 3: Hardware turns coarse- into fine-grained locks!**
  - While also reducing memory traffic for lock in common case

Outline

1. RCU
2. Improving spinlock performance
3. Kernel interface for sleeping locks
4. Deadlock
5. Transactions
6. Scalable interface design

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**
  - Have global variable: `_Atomic(routing_table *) rt;`
  - Use it with no lock

```
#define RELAXED(var) \
  atomic_load_explicit(&(var), memory_order_relaxed)

/* ... */

  route = lookup(RELAXED(rt), destination);
```

- **Update by making copy, swapping pointer**

```
/* update mutex held here, serializing updates */
routing_table *newrt = copy_routing_table(rt);
update_routing_table(newrt);
atomic_store_explicit(&rt, newrt, memory_order_release);
```


Is RCU really safe?

- **Consider the use of global `rt` with no fences:**

```
lookup(RELAXED(rt), route);
```

- **Could a CPU read new pointer but then old contents of `*rt`?**
- **Yes on alpha, No on all other existing architectures**
- **We are saved by *dependency ordering* in hardware**
  - Instruction *B* depends on *A* if *B* uses result of *A*
  - Non-alpha CPUs won't re-order dependent instructions
  - If writer uses release fence, safe to load pointer then just use it
- **This is the point of `memory_order_consume`**
  - Should be equivalent to acquire barrier on alpha
  - But should compile to nothing (be free) on other machines
  - But hard to get semantics right (temporarily deprecated in C++)

Preemptible kernels

- **Recall *kernel process context* from lecture 1**
  - When CPU in kernel mode but executing on behalf of a process (e.g., might be in system call or page fault handler)
  - As opposed to interrupt handlers or context switch code
- **A *preemptible kernel* can preempt process context code**
  - Take a CPU core away from kernel process context code between any two instructions
  - Give the same CPU core to kernel code for a different process
- **Don't confuse with:**
  - Interrupt handlers can always preempt process context code
  - Preemptive threads (always have for multicore)
  - Process context code running concurrently on other CPU cores
- **Sometimes want or need to disable preemption**
  - Code that must not be migrated between CPUs (per-CPU structs)
  - Before acquiring spinlock (could improve performance)

Garbage collection

- **When can you free memory of old routing table?**
  - When you are guaranteed no one is using it—how to determine?
- **Definitions:**
  - *temporary variable* – short-used (e.g., local) variable
  - *permanent variable* – long lived data (e.g., global `rt` pointer)
  - *quiescent state* – when all a thread's temporary variables dead
  - *quiescent period* – time during which every thread has been in quiescent state at least once
- **Free old copy of updated data after quiescent period**
  - How to determine when quiescent period has gone by?
  - E.g., keep count of syscalls/context switches on each CPU
- **Restrictions:**
  - Can't hold a pointer across context switch or user mode (Never copy `rt` into another permanent variable)
  - Must disable preemption while consuming RCU data structure

Outline

Useful macros

- **Atomic compare and swap:** `CAS (mem, old, new)`
  - In C11: `atomic_compare_exchange_strong`
  - On x86: `cmpxchg` instruction provides this (with `lock` prefix)
  - If `*mem == old`, then swap `*mem`↔`new` and return `true`, else `false`
- **Atomic swap:** `XCHG (mem, new)`
  - C11 `atomic_exchange`, can implement with `xchg` on x86
  - Atomically exchanges `*mem`↔`new`
- **Atomic fetch and add:** `FADD (mem, val)`
  - C11 `atomic_fetch_add`, can implement with `lock add` on x86
  - Atomically sets `*mem += val` and returns *old* value of `*mem`
- **Atomic fetch and subtract:** `FSUB (mem, val)`
- **Note: atomics return previous value (like `x++`, not `++x`)**
- **All behave like sequentially consistent fences**
  - Unlike `_explicit` versions, which take a `memory_order` argument

MCS lock

- **Idea 2: Build a better spinlock**
- **Lock designed by Mellor-Crummey and Scott**
  - Goal: reduce bus traffic on cc machines, improve fairness
- **Each CPU has a `qnode` structure in local memory**

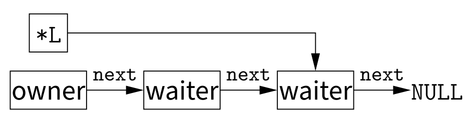
```
typedef struct qnode {
  _Atomic (struct qnode *) next;
  atomic_bool locked;
} qnode;
```

  - Local can mean local memory in NUMA machine
  - Or just its own cache line that gets cached in exclusive mode
- **While waiting, spin on *your local* `locked` flag**
- **A lock is a `qnode` pointer:** `typedef _Atomic (qnode *) lock;`
  - Construct list of CPUs holding or waiting for lock
  - `lock` itself points to tail of list list (or NULL when unlocked)

MCS Acquire

- **If unlocked, `L` is `NULL`**
- **If locked, no waiters, `L` is owner's `qnode`**
- **If waiters, `*L` is tail of waiter list:**

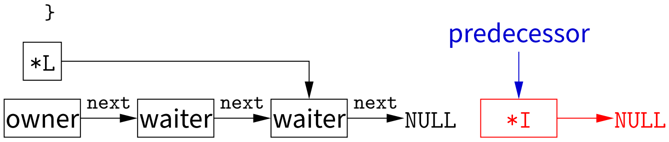
```
acquire (lock *L, qnode *I) {
  I->next = NULL;
  qnode *predecessor = I;
  XCHG (*L, predecessor);
  if (predecessor != NULL) {
    I->locked = true;
    predecessor->next = I;
    while (I->locked)
      ;
  }
}
```



MCS Acquire

- **If unlocked, `L` is `NULL`**
- **If locked, no waiters, `L` is owner's `qnode`**
- **If waiters, `*L` is tail of waiter list:**

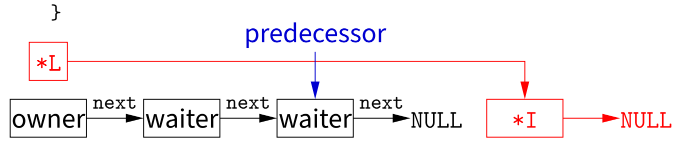
```
acquire (lock *L, qnode *I) {
  I->next = NULL;
  qnode *predecessor = I;
  XCHG (*L, predecessor);
  if (predecessor != NULL) {
    I->locked = true;
    predecessor->next = I;
    while (I->locked)
      ;
  }
}
```



MCS Acquire

- **If unlocked, `L` is `NULL`**
- **If locked, no waiters, `L` is owner's `qnode`**
- **If waiters, `*L` is tail of waiter list:**

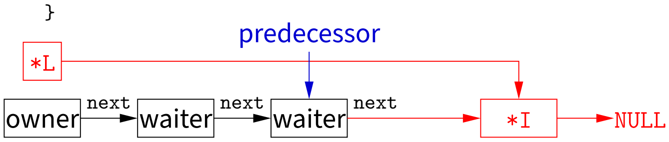
```
acquire (lock *L, qnode *I) {
  I->next = NULL;
  qnode *predecessor = I;
  XCHG (*L, predecessor);
  if (predecessor != NULL) {
    I->locked = true;
    predecessor->next = I;
    while (I->locked)
      ;
  }
}
```



MCS Acquire

- **If unlocked, `L` is `NULL`**
- **If locked, no waiters, `L` is owner's `qnode`**
- **If waiters, `*L` is tail of waiter list:**

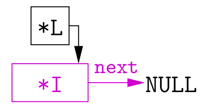
```
acquire (lock *L, qnode *I) {
  I->next = NULL;
  qnode *predecessor = I;
  XCHG (*L, predecessor);
  if (predecessor != NULL) {
    I->locked = true;
    predecessor->next = I;
    while (I->locked)
      ;
  }
}
```



MCS Release with CAS

```
release (lock *L, qnode *I) {
  if (!I->next)
    if (CAS (*L, I, NULL))
      return;
  while (!I->next)
    ;
  I->next->locked = false;
}
```

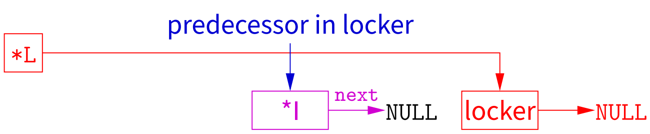
- **If `I->next` NULL and `*L == I`**
  - No one else is waiting for lock, OK to set `*L = NULL`



MCS Release with CAS

```
release (lock *L, qnode *I) {
  if (!I->next)
    if (CAS (*L, I, NULL))
      return;
  while (!I->next)
    ;
  I->next->locked = false;
}
```

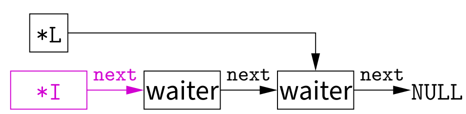
- **If `I->next` NULL and `*L != I`**
  - Another thread is in the middle of `acquire`
  - Just wait for `I->next` to be non-NULL



MCS Release with CAS

```
release (lock *L, qnode *I) {
  if (!I->next)
    if (CAS (*L, I, NULL))
      return;
  while (!I->next)
    ;
  I->next->locked = false;
}
```

- **If `I->next` is non-NULL**
  - `I->next` oldest waiter, wake up with `I->next->locked = false`



MCS Release w/o CAS

- **What to do if no atomic CAS (consensus number ∞), but do have XCHG (consensus number 2)?**
- **Be optimistic—read `*L` with two `XCHG`s:**
  1. Atomically swap NULL into `*L`
     - If old value of `*L` was `I`, no waiters and we are done
  2. Atomically swap old `*L` value back into `*L`
     - If `*L` unchanged, same effect as `CAS`
- **Otherwise, we have to clean up the mess**
  - Some "userper" attempted to acquire lock between **1** and **2**
  - Because `*L` was `NULL`, the userper succeeded
    (May be followed by zero or more waiters)
  - Graft old list of waiters on to end of new last waiter
    (Sacrifice small amount of fairness, but still safe)

MCS Release w/o C&S code

```
release (lock *L, qnode *I) {
  if (I->next)
    I->next->locked = false;
  else {
    qnode *old_tail = NULL;
    XCHG (*L, old_tail);
    if (old_tail == I)
      return;

    /* old_tail != I? CAS would have failed, so undo XCHG */
    qnode *userper = old_tail;
    XCHG (*L, userper);
    while (I->next == NULL)
      ;
    if (userper)  /* someone changed *L between 2 XCHGs */
      userper->next = I->next;
    else
      I->next->locked = false;
  }
}
```

Outline

1. RCU
2. Improving spinlock performance
3. Kernel interface for sleeping locks
4. Deadlock
5. Transactions
6. Scalable interface design

Kernel support for sleeping locks

- **Sleeping locks must interact with scheduler**
  - For processes or kernel threads, must go into kernel (expensive)
  - Common case is you can acquire lock—how to optimize?
- **Idea: never enter kernel for uncontested lock**

```
struct lock {
  atomic_flag busy;
  _Atomic (thread *) waiters; /* wait-free stack/queue */
};
void acquire (lock *lk) {
  while (atomic_flag_test_and_set (&lk->busy)) { /* 1 */
    atomic_push (&lk->waiters, self);            /* 2 */
    sleep ();
  }
}
void release (lock *lk) {
  atomic_flag_clear(&lk->busy);
  wakeup (atomic_pop (&lk->waiters));
}
```


Race condition

- **Unfortunately, previous slide not safe**
  - What happens if release called between lines 1 and 2?
  - `wakeup` called on `NULL`, so `acquire` blocks
- ***futex* abstraction solves the problem [Franke]**
  - Ask kernel to sleep only if memory location hasn't changed
- `void futex (int *uaddr, FUTEX_WAIT, int val...);`
  - Go to sleep only if `*uaddr == val`
  - Extra arguments allow timeouts, etc.
- `void futex (int *uaddr, FUTEX_WAKE, int val...);`
  - Wake up at most `val` threads sleeping on `uaddr`
- `uaddr` **is translated down to offset in VM object**
  - So works on memory mapped file at different virtual addresses in different processes

Futex example

```
struct lock {
  atomic_flag busy;
};
void acquire (lock *lk) {
  while (atomic_flag_test_and_set (&lk->busy))
    futex(&lk->busy, FUTEX_WAIT, 1);
}
void release (lock *lk) {
  atomic_flag_clear (&lk->busy);
  futex(&lk->busy, FUTEX_WAKE, 1);
}
```

- **What's suboptimal about this code?**
- **See [Drepper] for these examples and a good discussion**

Futex example

```
struct lock {
  atomic_flag busy;
};
void acquire (lock *lk) {
  while (atomic_flag_test_and_set (&lk->busy))
    futex(&lk->busy, FUTEX_WAIT, 1);
}
void release (lock *lk) {
  atomic_flag_clear (&lk->busy);
  futex(&lk->busy, FUTEX_WAKE, 1);
}
```

- **What's suboptimal about this code?**
  - `release` requires a system call (expensive) even with no contention
- **See [Drepper] for these examples and a good discussion**

Futex example, second attempt

```
static_assert (ATOMIC_INT_LOCK_FREE >= 2);

struct lock {
  atomic_int busy;
};
void acquire (lock *lk) {
  int c;
  while ((c = FADD(&lk->busy, 1)))                 /* 1 */
    futex((int*) &lk->busy, FUTEX_WAIT, c+1);  /* 2 */
}
void release (lock *lk) {
  if (FSUB(&lk->busy, 1) != 1) {
    lk->busy = 0;
    futex((int*) &lk->busy, FUTEX_WAKE, 1);
  }
}
```

- **Now what's wrong with this code?**

Futex example, second attempt

```
static_assert (ATOMIC_INT_LOCK_FREE >= 2);

struct lock {
  atomic_int busy;
};
void acquire (lock *lk) {
  int c;
  while ((c = FADD(&lk->busy, 1)))                /* 1 */
    futex((int*) &lk->busy, FUTEX_WAIT, c+1);  /* 2 */
}
void release (lock *lk) {
  if (FSUB(&lk->busy, 1) != 1) {
    lk->busy = 0;
    futex((int*) &lk->busy, FUTEX_WAKE, 1);
  }
}
```

- **Now what's wrong with this code?**
  - Two threads could interleave lines 1 and 2, never sleep
  - Could even overflow the counter, violate mutual exclusion

Futex example, third attempt

```
struct lock {
  // 0=unlocked, 1=locked no waiters, 2=locked+waiters
  atomic_int state;
};
void acquire (lock *lk) {
  int c = 1;
  if (!CAS (&lk->state, 0, c)) {
    XCHG (&lk->state, c = 2);
    while (c != 0) {
      futex ((int *) &lk->state, FUTEX_WAIT, 2);
      XCHG (&lk->state, c = 2);
    }
  }
}
void release (lock *lk) {
  if (FSUB (&lk->state, 1) != 1) { // FSUB returns old value
    lk->state = 0;
    futex ((int *) &lk->state, FUTEX_WAKE, 1);
  }
}
```


Outline

The deadlock problem

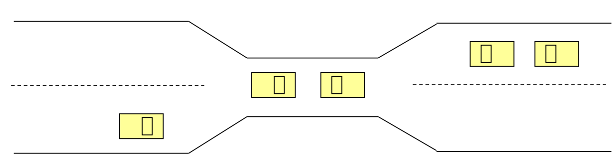
```
mutex_t m1, m2;

void p1 (void *ignored) {
  lock (m1);
  lock (m2);
  /* critical section */
  unlock (m2);
  unlock (m1);
}
void p2 (void *ignored) {
  lock (m2);
  lock (m1);
  /* critical section */
  unlock (m1);
  unlock (m2);
}
```

- **This program can cease to make progress – how?**
- **Can you have deadlock w/o mutexes?**

More deadlocks

- **Same problem with condition variables**
  - Suppose resource 1 managed by c_1, resource 2 by c_2
  - A has 1, waits on $c2$, B has 2, waits on $c1$
- **Or have combined mutex/condition variable deadlock:**
  - `lock (a); lock (b); while (!ready) wait (b, c);`
    `unlock (b); unlock (a);`
  - `lock (a); lock (b); ready = true; signal (c);`
    `unlock (b); unlock (a);`
- **One lesson: Dangerous to hold locks when crossing abstraction barriers!**
  - I.e., `lock (a)` then call function that uses condition variable

Deadlocks w/o computers



- **Real issue is *resources* & how required**
- **E.g., bridge only allows traffic in one direction**
  - Each section of a bridge can be viewed as a resource.
  - If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).
  - Several cars may have to be backed up if a deadlock occurs.
  - Starvation is possible.

Deadlock conditions

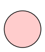
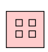
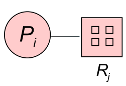
1. **Limited access (mutual exclusion):**
   - Resource can only be shared with finite users
2. **No preemption:**
   - Once resource granted, cannot be taken away
3. **Multiple independent requests (hold and wait):**
   - Don't ask all at once
     (wait for next resource while holding current one)
4. **Circularity in graph of requests**

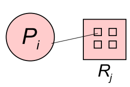
- **All of 1–4 necessary for deadlock to occur**
- **Two approaches to dealing with deadlock:**
  - Pro-active: prevention
  - Reactive: detection + corrective action

Prevent by eliminating one condition

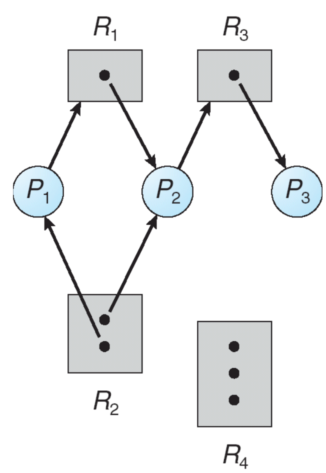
1. **Limited access (mutual exclusion):**
   - Buy more resources, split into pieces, or virtualize to make "infinite" copies
   - Threads: threads have copy of registers = no lock
2. **No preemption:**
   - Physical memory: virtualized with VM, can take physical page away and give to another process!
3. **Multiple independent requests (hold and wait):**
   - Wait on all resources at once (must know in advance)
4. **Circularity in graph of requests**
   - Single lock for entire system: (problems?)
   - Partial ordering of resources (next)

Resource-allocation graph

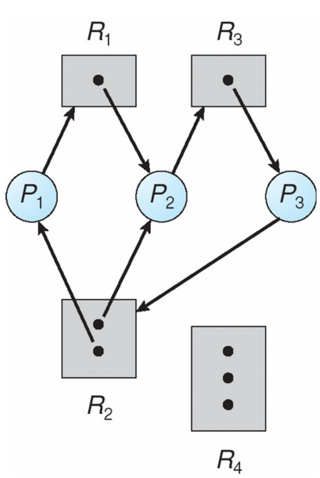
- **View system as graph**
  - Processes and Resources are nodes
  - Resource Requests and Assignments are edges
- **Process:** 
- **Resource with 4 instances:** 
- P_i **requesting** R_j**:** 

- P_i **holding instance of** R_j**:** 

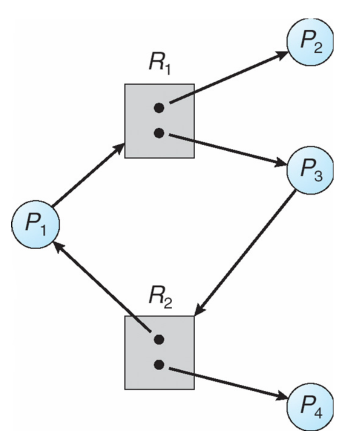
Example resource allocation graph



Graph with deadlock



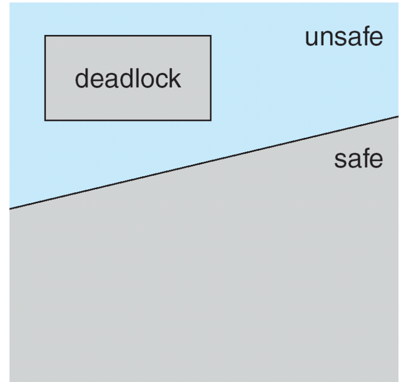
Is this deadlock?



Cycles and deadlock

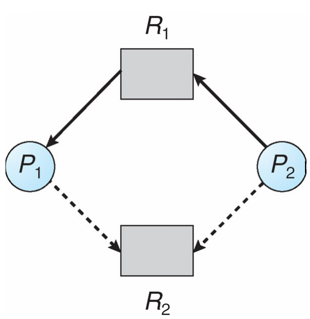
- **If graph has no cycles $\Longrightarrow$ no deadlock**
- **If graph contains a cycle**
  - Definitely deadlock if only one instance per resource
  - Otherwise, maybe deadlock, maybe not
- **Prevent deadlock with partial order on resources**
  - E.g., always acquire mutex m_1 before m_2
  - Usually design locking discipline for application this way

Prevention



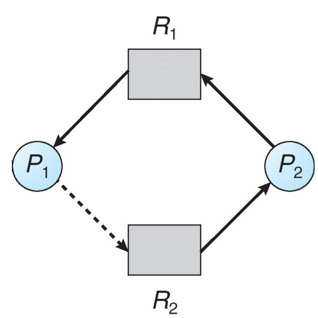
- **Determine safe states based on *possible* resource allocation**
- **Conservatively prohibits non-deadlocked states**

Claim edges



- **Dotted line is *claim edge***
  - Signifies process *may* request resource

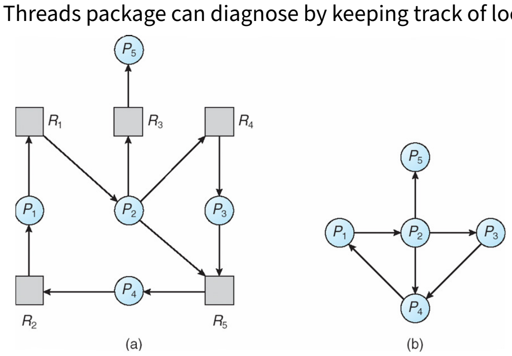
Example: unsafe state



- **Note cycle in graph**
  - P_1 might request R_2 before relinquishing R_1
  - Would cause deadlock

Detecting deadlock

- **Static approaches (hard)**
- **Dynamically, program grinds to a halt**
  - Threads package can diagnose by keeping track of locks held:



Resource-Allocation Graph   Corresponding wait-for graph

Fixing & debugging deadlocks

- **Reboot system / restart application**
- **Examine hung process with debugger**
- **Threads package can deduce partial order**
  - For each lock acquired, order with other locks held
  - If cycle occurs, abort with error
  - Detects *potential* deadlocks even if they do not occur
- **Or use *transactions*. . .**
  - Another paradigm for handling concurrency
  - Often provided by databases, but some OSes use them
  - *Vino* OS used transactions to abort after failures [Seltzer]

Outline

1. RCU
2. Improving spinlock performance
3. Kernel interface for sleeping locks
4. Deadlock
5. Transactions
6. Scalable interface design

Transactions

- **A *transaction* T is a collection of actions with**
  - *Atomicity* – all or none of actions happen
  - *Consistency* – T leaves data in valid state
  - *Isolation* – T's actions all appear to happen before or after every other transaction
  - *Durability*[1] – T's effects will survive reboots
  - Often hear mnemonic *ACID* to refer to above
- **Transactions typically executed concurrently**
  - But *isolation* means must *appear* not to
  - Must roll-back transactions that use others' state
  - Means you have to record all changes to undo them
- **When deadlock detected just abort a transaction**
  - Breaks the dependency cycle

[1] Not applicable to topics in this lecture

Transactional memory

- **Some modern processors support *transactional memory***
- **Transactional Synchronization Extensions (TSX) [intel1§16]**
  - `xbegin abort_handler` – begins a transaction
  - `xend` – commit a transaction
  - `xabort $code` – abort transaction with 8-bit code
  - Note: nested transactions okay (also `xtest` tests if in transaction)
- **During transaction, processor tracks accessed memory**
  - Keeps read-set and write-set of cache lines
  - Nothing gets written back to memory during transaction
  - Transaction aborts (at `xend` or earlier) if any conflicts
  - Otherwise, all dirty cache lines are "written" atomically (in practice switch to non-transactional M state of MESI)

Using transactional memory

- **Idea 3: Use to get "free" fine-grained locking on a hash table**
  - E.g., concurrent inserts that don't touch same buckets are okay
  - Should *read* spinlock to make sure not taken (but not write) [Kim]
  - Hardware will detect there was no conflict
- **Can also use to poll for one of many asynchronous events**
  - Start transaction
  - Fill cache with values to which you want to see changes
  - Loop until a write causes your transaction to abort
- **Note: Transactions are never guaranteed to commit**
  - Might overflow cache, get false sharing, see weird processor issue
  - Means abort path must always be able to perform transaction (e.g., you do need a lock on your hash table)

Hardware lock elision (HLE)

- **Idea: make it so spinlocks rarely need to spin**
  - Begin a transaction when you acquire lock
  - Other CPUs won't see lock acquired, can also enter critical section
  - Okay not to have mutual exclusion when no memory conflicts!
  - On conflict, abort and restart without transaction, thereby visibly acquiring lock (and aborting other concurrent transactions)
- **Intel support:**
  - Use `xacquire` prefix before `xchgl` (used for test and set)
  - Use `xrelease` prefix before `movl` that releases lock
  - Prefixes chosen to be noops on older CPUs (binary compatibility)
- **Hash table example:**
  - Use `xacquire xchgl` in table-wide test-and-set spinlock
  - Works correctly on older CPUs (with coarse-grained lock)
  - Allows safe concurrent accesses on newer CPUs!

Outline

1. RCU
2. Improving spinlock performance
3. Kernel interface for sleeping locks
4. Deadlock
5. Transactions
6. Scalable interface design

Scalable interfaces

- **Not all interfaces can scale**
- **How to tell which can and which can't?**
- **Scalable Commutativity Rule:** ***"Whenever interface operations commute, they can be implemented in a way that scales"*** **[Clements]**

Are fork(), execve() broadly commutative?

```
pid_t pid = fork();
if (!pid)
  execlp("bash", "bash", NULL);
```


Are fork(), execve() broadly commutative?

```
pid_t pid = fork();
if (!pid)
  execlp("bash", "bash", NULL);
```


- **No,** `fork()` **doesn't commute with memory writes, many file descriptor operations, and all address space operations**
  - E.g., `close(fd); fork();` vs. `fork(); close(fd);`
- `execve()` **often follows** `fork()` **and undoes most of** `fork()`**'s sub operations**
- `posix_spawn()`**, which combines** `fork()` **and** `execve()` **into a single operation, is broadly commutative**
  - But obviously more complex, less flexible
  - Maybe Microsoft will have the last laugh?

Is open() broadly commutative?

```
int fd1 = open("foo", O_RDONLY);
int fd2 = open("bar", O_RDONLY);
```


Is open() broadly commutative?

```
int fd1 = open("foo", O_RDONLY);
int fd2 = open("bar", O_RDONLY);
```


- **Actually** `open()` **does not broadly commute!**
- **Does not commute with any system call (including itself) that creates a file descriptor**
- **Why? POSIX requires new descriptors to be assigned the lowest available integer**
- **If we fixed this,** `open()` **would commute, as long as it is not creating a file in the same directory as another operation**