Readers-Writers Problem

- Multiple threads may access data
  - *Readers* – will only observe, not modify data
  - *Writers* – will change the data

- Goal: allow multiple readers or one single writer
  - Thus, lock can be *shared* amongst concurrent readers

- Can implement with other primitives (next slide)
  - Keep integer \( i \) – # or readers or -1 if held by writer
  - Protect \( i \) with mutex
  - Sleep on condition variable when can’t get lock
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);
}
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);
}

• Note: Must deal with starvation
Review: Test-and-set spinlock

```c
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;
}
```
Review: Test-and-set on alpha

- `ldl_l` – load locked
  
- `stl_c` – store but sets reg to 0 if not atomic with `ldl_l`

```assembly
_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
```

- **Note**: Alpha memory consistency weaker than S.C.
- **Memory barrier instruction**, `mb`, is there to compensate
  - Also need `mb` before releasing spinlock
Relaxed consistency model

- Suppose no sequential consistency & don’t compensate
  - E.g., say we omit mb instruction on alpha?

- Hardware could violate program order

  PROGRAM ORDER
  read/write: v->lock = 1;
  read: v->val;
  write: v->val = read_val + 1;
  write: v->lock = 0;

  VIEW ON OTHER CPU
  v->lock = 1;
  v->lock = 0;
  /* danger */
  v->val = read_val + 1;

- If atomic_inc called where danger, bad val results
Relaxed consistency (continued)

- Use memory barriers to preserve program order of memory accesses with respect to locks
  - \texttt{mb in test\_and\_set preserves program order}
    - All ops before \texttt{mb} in program order appear before on all CPUs
    - All ops after \texttt{mb} in program order appear after on all CPUs
- Many examples in this lecture assume S.C.
  - Useful on non-S.C. hardware, but must add barriers
- Dealing w. memory consistency important
  - E.g., see [Howells] for how Linux deals
- Fortunately need not overly complicate code
  - If you do locking right, only need to add a few barriers
  - Code will be easily portable to new CPUs
Cache coherence

- Performance requires caches
- Sequential consistency requires cache coherence
- Barrier & atomic ops require cache coherence
- Bus-based approaches
  - “Snoopy” protocols, each CPU listens to memory bus
  - Use write through and invalidate when you see a write
  - Or have ownership scheme (e.g., Pentium MESI bits)
  - Bus-based schemes limit scalability
- Cache-Only Memory Architecture (COMA)
  - Each CPU has local RAM, treated as cache
  - Cache lines migrate around based on access
  - Data lives only in cache
cc-NUMA

- Previous slide had *dance hall* architectures
  - Any CPU can “dance with” any memory equally
- An alternative: Non-Uniform Memory Access
  - 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 machines with many CPUs
  - But AMD Opterons integrated mem. controller, essentially NUMA
  - Now intel CPUs are like this, too
- cc-NUMA = cache-coherent NUMA
  - Can also have non-cache-coherent NUMA, though uncommon
  - BBN Butterfly 1 has no cache at all
  - Cray T3D has local/global memory
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 bus (especially when \( > 1 \) spinner)
  - Not necessarily fair (same CPU acquires lock many times)
  - Even less fair on a NUMA machine
  - Allegedly Google had fairness problems even on Opterons

• **Idea 1: Avoid spinlocks altogether**

• **Idea 2: Reduce bus traffic with better spinlocks**
  - Design lock that spins only on local memory
  - Also gives better fairness
Recall producer/consumer (lecture 3)

/* PRODUCER */
for (;;) {
    /* produce item, put in nextProduced */

    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);
    /* use item in nextConsumed */
}

11/27
Eliminating locks

- One use of locks is to coordinate multiple updates of a single piece of state
- How to remove locks here?
  - Factor state so each variable only has a single writer

- Producer/consumer example revisited
  - Assume for example you have sequential consistency
  - 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
  - Use in/out to detect buffer state
  - But note next example busy-waits, which is less good
Lock-free producer/consumer

void producer (void *ignored) {
    for (;;) {
        /* produce an item and put in nextProduced */

        while (((in + 1) % BUF_SIZE) == out)
            ; // do nothing
        buffer [in] = nextProduced;
        in = (in + 1) % BUF_SIZE;
    }
}

void consumer (void *ignored) {
    for (;;) {
        while (in == out)
            ; // do nothing
        nextConsumed = buffer[out];
        out = (out + 1) % BUF_SIZE;

        /* consume the item in nextConsumed */
    }
}
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 atomic instructions available on some CPUs

- E.g., atomic compare and swap: CAS (mem, old, new)
  - If *mem == old, then set *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 = ∞ [Herlihy])
  - Seldom used because inefficient (lots of retries), but entire cache kernel written w/o locks using double CAS [Greenwald]
Example: stack

```c
struct item {
    /* data */
    struct item *next;
};
typedef 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;
}
```
Wait-free stack issues

i = *stack;
reg ← i->next

Meanwhile, memory of object A gets recycled for A' of same type

CAS (stack, i, i->next)

stack          A     B     C

stack          B     C     NULL

stack          C     NULL

stack          A'    C     NULL

stack          B     garbage

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

• Can also eliminate locks by having race conditions
• Sometimes “cheating” buys efficiency...
• Care more about speed than accuracy
  
  hits++;  // each time someone accesses web site

• Know you can get away with race

  if (!initialized) {
    lock (m);
    if (!initialized) {
      initialize ();
      initialized = 1;
    }
  }
  unlock (m);
Read-copy update [McKenney]

- Some data is read way more often than written
- Routing tables
  - Consulted for each packet that is forwarded
- Data maps in system with 100+ disks
  - Updated when disk fails, maybe every $10^{10}$ operations
- Optimize for the common case of reading w/o lock
  - E.g., global variable: `routing_table *rt;`
  - Call `lookup (rt, route);` with no locking
- Update by making copy, swapping pointer
  - E.g., `routing_table *nrt = copy_routing_table (rt);`
  - Update `nrt`
  - Set global `rt = nrt` when done updating
  - All `lookup` calls see consistent old or new table
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
  - Can’t hold a pointer across context switch or user mode (Preemptable kernel complicates things slightly)
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**
  ```c
  typedef struct qnode {
    struct qnode *next;
    bool locked;
  } qnode;
  ```
  - Local can mean local memory in NUMA machine
  - Or just its own cache line that gets cached in exclusive mode

- **A lock is just a pointer to a `qnode**
  ```c
  typedef qnode *lock;
  ```

- **Lock is list of CPUs holding or waiting for lock**
- **While waiting, spin on your local locked flag**
MCS Acquire

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

• If unlocked, L is NULL
• If locked, no waiters, L is owner’s qnode
• If waiters, *L is tail of waiter list:
MCS Acquire

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

- **If unlocked,** \( L \) **is** NULL
- **If locked, no waiters,** \( L \) **is owner’s** qnode
- **If waiters,** \( *L \) **is tail of waiter list:**

![Diagram]

\( *L \) \hspace{1cm} \text{predecessor} \hspace{1cm} \text{owner} \rightarrow \text{waiter} \rightarrow \text{waiter} \rightarrow \text{NULL} \rightarrow *I \rightarrow \text{NULL} \)
MCS Acquire

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

- If unlocked, \( L \) is NULL
- If locked, no waiters, \( L \) is owner’s qnode
- If waiters, \( *L \) is tail of waiter list:

\[
\begin{align*}
&*L \\
&\text{predecessor} \\
&\text{owner} \quad \text{next} \quad \text{waiter} \quad \text{next} \quad \text{waiter} \quad \text{next} \quad \text{NULL} \\
&\text{NULL} \quad \text{next}
\end{align*}
\]
MCS Acquire

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

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

```
owner -> waiter -> waiter -> *I -> NULL
```
MCS Release w. 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

![Diagram of lock and qnode]
MCS Release w. 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

predecessor in locker
MCS Release w. 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 w. I->next->locked = false

![Diagram showing the release process with CAS](attachment:image.png)
MCS Release w/o CAS

- What to do if no atomic compare & swap?
  - Be optimistic—read *L w. two XCHGs:
    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)
  - Stick old list of waiters on to end of new last waiter
MCS Release w/o C&S code

```c
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;

        qnode *userper = old_tail;
        XCHG (*L, userper);
        while (I->next == NULL);

        if (userper != NULL) {
            /* Someone changed *L between 2 XCHGs */
            userper->next = I->next;
        }
    } else
        I->next->locked = false;
}
```
Kernel support for synchronization

- 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: only go into kernel if you can’t get lock

```c
struct lock {
    int busy;
    thread *waiters;
};

void acquire (lock *lk) {
    while (test_and_set (&lk->busy)) { /* 1 */
        atomic_push (&lk->waiters, self); /* 2 */
        sleep ();
    }
}

void release (lock *lk) {
    lk->busy = 0;
    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
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 $T'$
  - Durability* – $T$’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