Synchronization review

Libraries for User Apps

Pthread and other libraries

In System Programming and OOP courses, we have seen that there are libraries that allow us to write multithreaded programs.

Pthread library (Posix thread library) is such an example that provides an API for multithreaded user programs. In the pthread library, we have seen two different synchronization tools: mutex and condition variables.

Mutexes

```
//m1 is a mutex variable
mutex_lock(m1);//acquire lock
critical_section
mutex_unlock(m1); //release lock
```

- You can consider lock as **“a mic among participants that controls who has the right to speak”**,
    - i.e whoever has the mic (m1) has the right to speak (in our case do ops on the memory shared among the participants).

Condition Variables

```
//two threads: e.g. producer/consumer

//Condition variables: c1 is for one condition, c2 is for another condition
//Locks: m1 to control the access to the critical section.
```

//1st thread
mutex_lock(m1);
/*critical section entry*/
while(need_to_wait_1){
    cond_wait(**c1**, m1);
}
 	
/*critical section exit*/
cond_signal(**c2**) //or broadcast
mutex_unlock(m1);
	// 2nd thread
	mutex_lock(m1);
	/*critical section entry*/
	while(need_to_wait_2){
	    cond_wait(**c2**, m1);
	};
	
	/*critical section exit*/
	cond_signal(**c1**) //or broadcast
	mutex_unlock(m1);

Semaphores

```
sem_t s;
sem_init(&s, 0, 10);
//a thread that wants to occupy a chair
sem_wait(&s); //down the value of s by 1

//After done with the chair
sem_post(&s); //up the value of s by 1
```

- If the return value of `sem_wait` is negative the thread waits as done in the mutex locks.
  - This happens when the value of s before `sem_wait` is 0.
- `sem_wait` and `sem_post` can be called from different processes/threads.

In Linux kernel

Pthread API is implemented by using NPTL (Native POSIX Thread Library - Wikipedia). It uses the system calls such as clone and futex, and atomic operations to create a library in glibc (The GNU C Library) (see pthread_create.c source code [glibc/nptl/pthread_create.c] - Codebrowser).

There are also alternative/different pthread implementations for Linux.

In the kernel space, similarly to processes, you can also spawn threads by using kernel threads(kthreads). There is also similar lock mechanism you can use(see locking — The Linux Kernel documentation)

Lock “Free” Multithreading:

Non-blocking synchronization

Atomic operations

Read-modify-write (RMW) atomic instructions

Memory barriers (see memory-barriers.txt)

In Linux kernel (mb(), smp_mb(), etc.),

assembly instruction `asm volatile ("mfence" : : : "memory")`

C11 atomic library, Linux system calls

RCU

Acquire/release semantics

Passing information reliably between threads about a variable.

– Ideal in producer/consumer type situations (pairing!!).

– After an ACQUIRE on a given variable, all memory accesses preceding any prior RELEASE on that same variable are guaranteed to be visible.

– All accesses of all previous critical sections for that variable are guaranteed to have completed.

– C++11's **memory_order_acquire**, **memory_order_release** and **memory_order_relaxed.**

https://elinux.org/images/a/ab/Bueso.pdf

Spinlocks with release/acquire semantics and atomic ops spinlocks

CPU0

CPU1

spin_lock(&l)

RELEASE

spin_unlock(&l)

spin_lock(&l)

ACQUIRE

spin_unlock(&l)

smp_store_release(lock→val, 0) <-> **cmpxchg_acquire**(lock→val, 0, LOCKED)

https://elinux.org/images/a/ab/Bueso.pdf

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);
}
```

http://www.scs.stanford.edu/21wi-cs140/notes/

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
  - But note next example busy-waits, which is less good

http://www.scs.stanford.edu/21wi-cs140/notes/

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);
    }
}
```

http://www.scs.stanford.edu/21wi-cs140/notes/

[Note fences not needed because no relaxed atomics]

Version with relaxed atomics

```
void producer (void *ignored) {
  for (;;) {
    item *nextProduced = produce_item ();
    int myin = atomic_load_explicit(&in, memory_order_relaxed);
    for (;;) {
      if ((myin + 1) % BUF_SIZE !=
          atomic_load_explicit(&out, memory_order_acquire))
          // Could you get away with relaxed here?^^^^^^^^
        break;
      thread_yield ();
    }
    buffer[myin] = nextProduced;
    atomic_store_explicit(&in, (myin+1) % BUF_SIZE,
                          memory_order_release);
  }
}

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

http://www.scs.stanford.edu/21wi-cs140/notes/

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`$\longleftrightarrow$`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]

http://www.scs.stanford.edu/21wi-cs140/notes/

Example: non-blocking stack

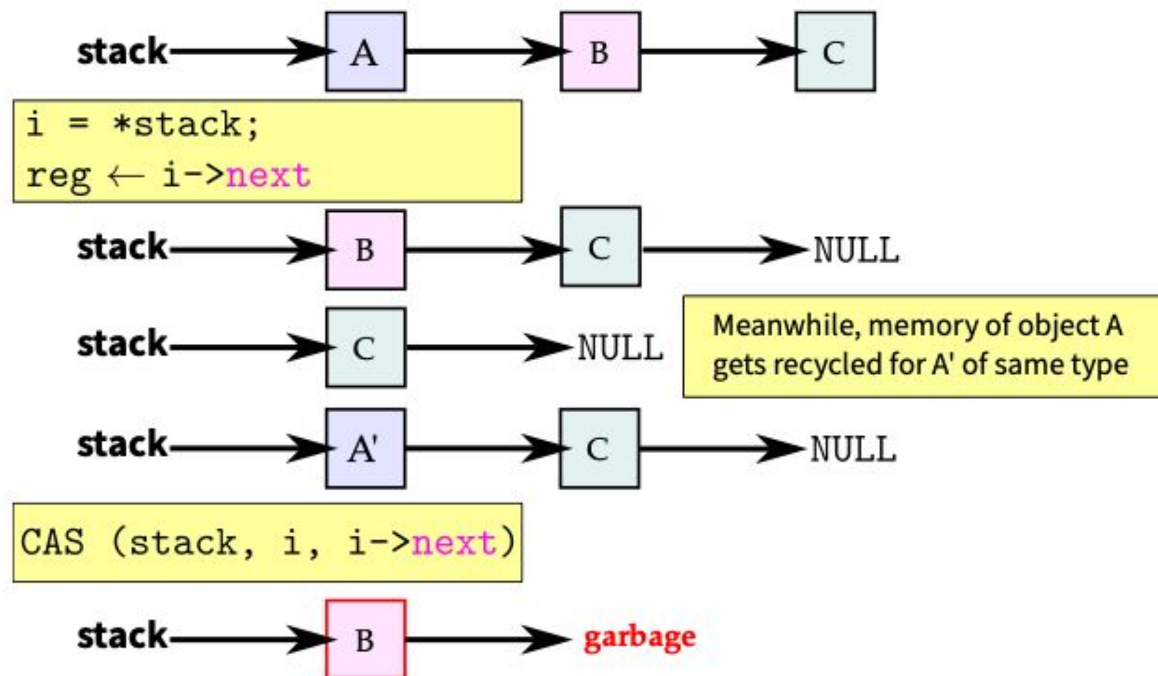
```
struct item {
  /* data */
  _Atomic (struct item *) next;
};
typedef _Atomic (struct item *) stack_t;

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;
}
```

http://www.scs.stanford.edu/21wi-cs140/notes/

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

"Benign" races

- Could also eliminate locks by having race conditions
- Maybe you think you care more about speed than correctness

```
++hits; /* each time someone accesses web site */
```

- Maybe you think you can get away with the race

```
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
  - Get undefined behavior—akin to out-of-bounds array access in C11
  - If needed for efficiency, use relaxed-memory-order atomics

http://www.scs.stanford.edu/21wi-cs140/notes/

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...**

http://www.scs.stanford.edu/21wi-cs140/notes/

Implementation of Locks

In kernel space: implementation by disabling and enabling interrupts

In user space: Atomic operations and futex system call

Recall: Naïve use of Interrupt Enable/Disable

- How can we build multi-instruction atomic operations?
  - Recall: dispatcher gets control in two ways.
    - » Internal: Thread does something to relinquish the CPU
    - » External: Interrupts cause dispatcher to take CPU
  - On a *uniprocessor*, can avoid context-switching by:
    - » Avoiding internal events (although virtual memory tricky)
    - » Preventing external events by disabling interrupts
- Consequently, naïve Implementation of locks:

```
LockAcquire { disable interrupts; }
LockRelease { enable interrupts; }
```

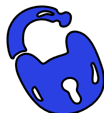
- Problems with this approach:
  - Can't let user do this! Consider following:

```
LockAcquire();
While(TRUE) {;}
```

  - Real-Time system—no guarantees on timing!
    - » Critical Sections might be arbitrarily long
  - What happens with I/O or other important events?
    - » "Reactor about to meltdown. Help?"

Recall: Better Implementation of Locks by Disabling Interrupts

- Key idea: maintain a lock variable and impose mutual exclusion only during operations on that variable



```
int value = FREE;

Acquire() {
     disable interrupts;
     if (value == BUSY) {
          put thread on wait queue;
          Go to sleep();
          // Enable interrupts?
     } else {
          value = BUSY;
     }
     enable interrupts;
}
```

```
Release() {
     disable interrupts;
     if (anyone on wait queue) {
          take thread off wait queue
          Place on ready queue;
     } else {
          value = FREE;
     }
     enable interrupts;
}
```

- Really only works in kernel – why?

New Lock Implementation: Discussion

- Why do we need to disable interrupts at all?
  - Avoid interruption between checking and setting lock value.
  - *Prevent switching to other thread that might be trying to acquire lock!*
  - Otherwise two threads could think that they both have lock!

```
Acquire() {
    disable interrupts;
    if (value == BUSY) {
        put thread on wait
queue;
        Go to sleep();
        // Enable interrupts?
    } else {
        value = BUSY;
    }
    enable interrupts;
}
```

"Meta-" Critical Section

- Note: unlike previous solution, this "meta-"critical section is very short
  - User of lock can take as long as they like in their own critical section: doesn't impact global machine behavior
  - Critical interrupts taken in time!

Interrupt Re-enable in Going to Sleep

- What about re-enabling ints when going to sleep?

```
Acquire() {
    disable interrupts;
    if (value == BUSY) {
        put thread on wait queue;
        Go to sleep();
    } else {
        value = BUSY;
    }
    enable interrupts;
}
```

Interrupt Re-enable in Going to Sleep

- What about re-enabling ints when going to sleep?

Enable Position →

```
Acquire() {
    disable interrupts;
    if (value == BUSY) {
        put thread on wait queue;
        Go to sleep();
    } else {
        value = BUSY;
    }
    enable interrupts;
}
```

- Before Putting thread on the wait queue?

Interrupt Re-enable in Going to Sleep

- What about re-enabling ints when going to sleep?

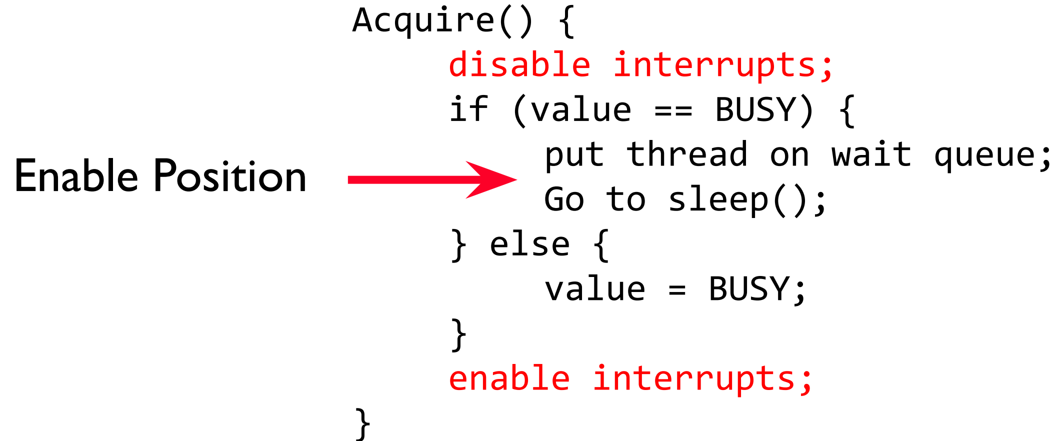
```
Acquire() {
    disable interrupts;
    if (value == BUSY) {
        put thread on wait queue;
        Go to sleep();
    } else {
        value = BUSY;
    }
    enable interrupts;
}
```

Enable Position

- Before Putting thread on the wait queue?
  – Release can check the queue and not wake up thread

Interrupt Re-enable in Going to Sleep

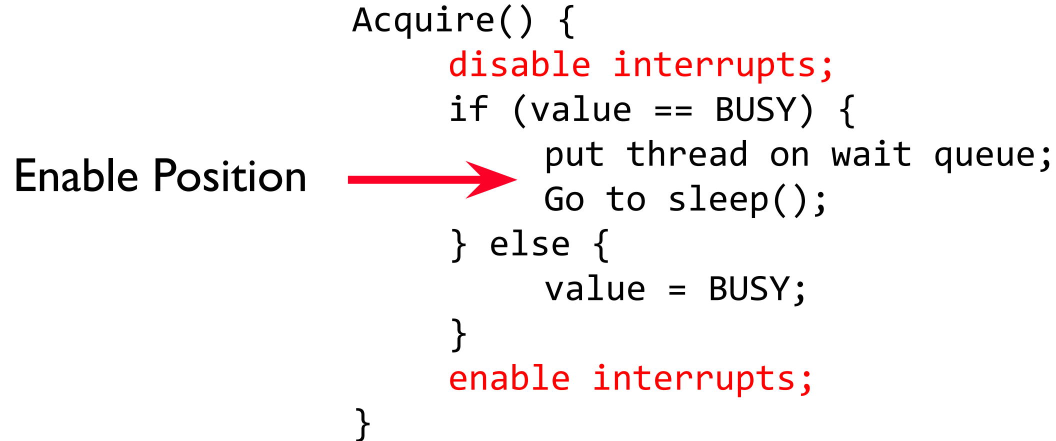
- What about re-enabling ints when going to sleep?



- Before Putting thread on the wait queue?
  – Release can check the queue and not wake up thread
- After putting the thread on the wait queue

Interrupt Re-enable in Going to Sleep

- What about re-enabling ints when going to sleep?

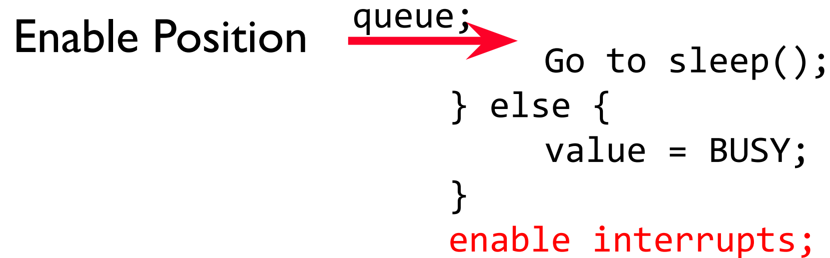



```
Acquire() {
    disable interrupts;
    if (value == BUSY) {
        put thread on wait queue;
        Go to sleep();
    } else {
        value = BUSY;
    }
    enable interrupts;
}
```

- Before Putting thread on the wait queue?
  - Release can check the queue and not wake up thread
- After putting the thread on the wait queue?
  - Release puts the thread on the ready queue, but the thread still thinks it needs to go to sleep
  - Misses wakeup and still holds lock (deadlock!)

Interrupt Re-enable in Going to Sleep

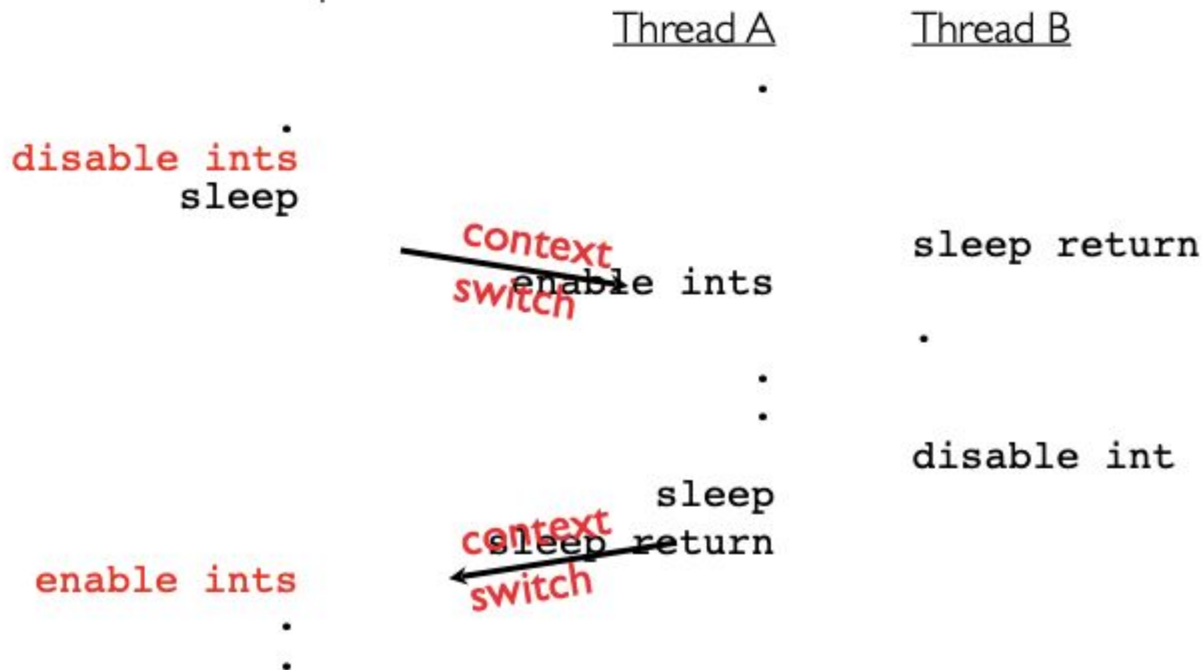
- What about re-enabling ints when going to sleep?



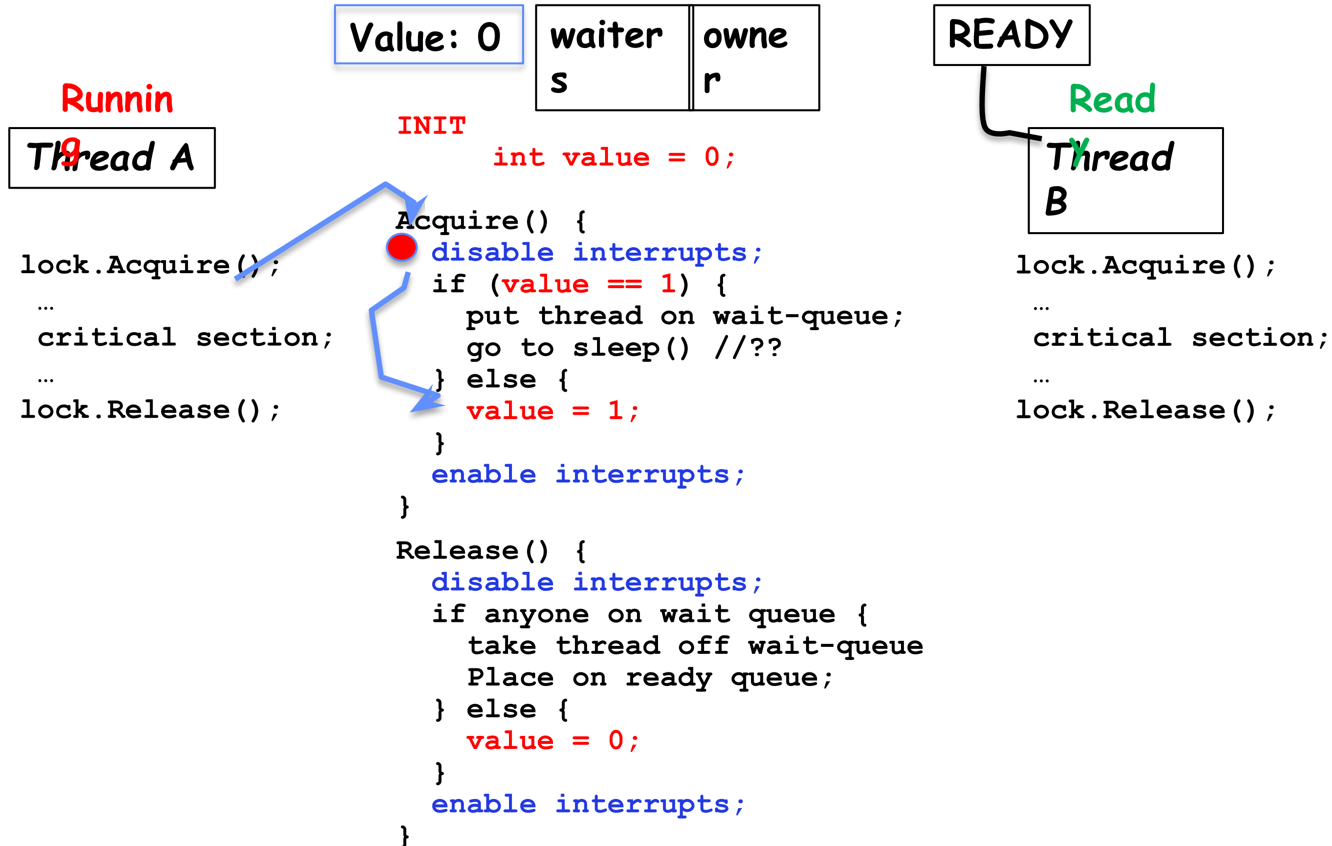
- Before Putting thread on the wait queue?
  – Release can check the queue and not wake up thread
- After putting the thread on the wait queue?
  – Release puts the thread on the ready queue, but the thread still thinks it needs to go to sleep
  – Misses wakeup and still holds lock (deadlock!)
- Want to put it after `sleep()`. But – how?

How to Re-enable After Sleep()?

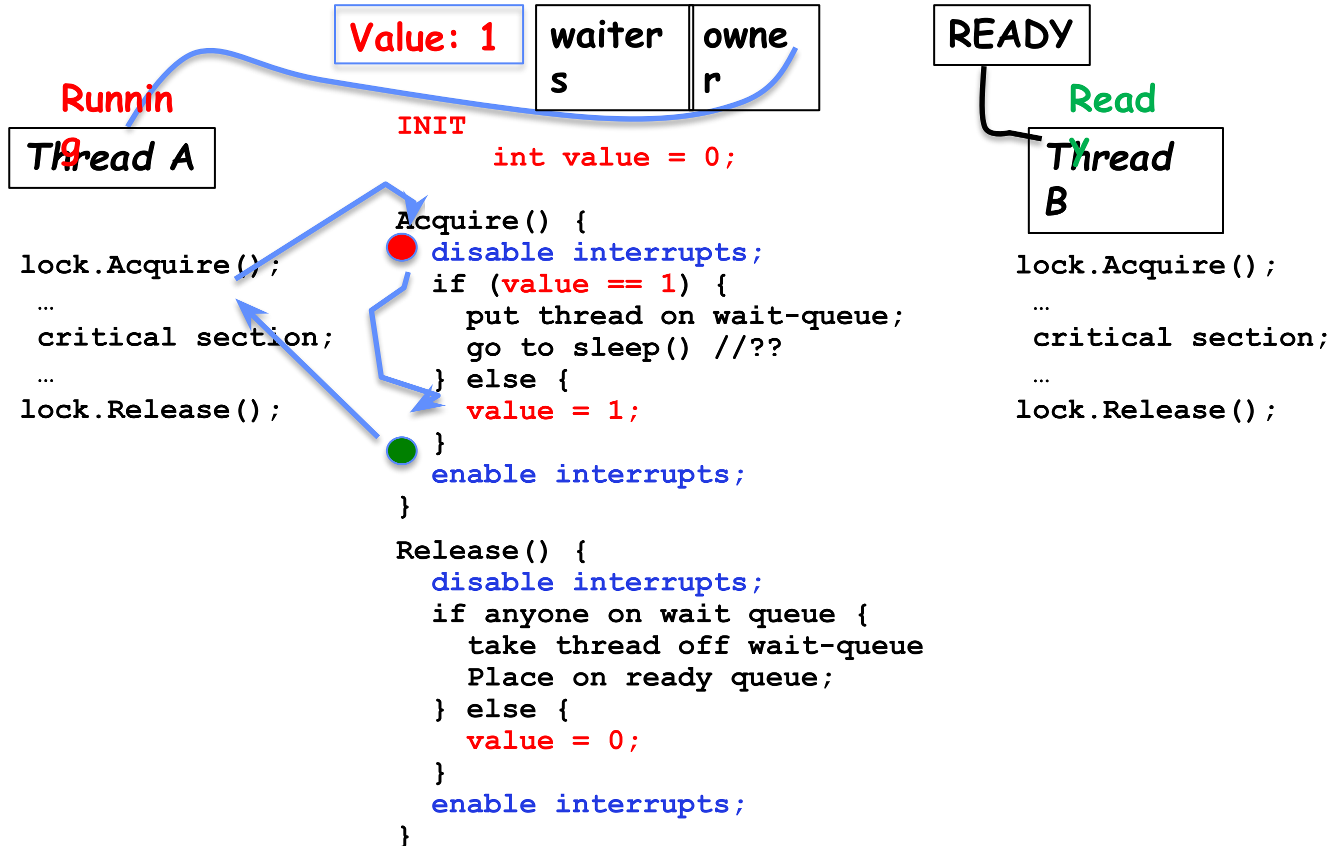
- In scheduler, since interrupts are disabled when you call sleep:
  - Responsibility of the next thread to re-enable ints
  - When the sleeping thread wakes up, returns to acquire and re-enables interrupts



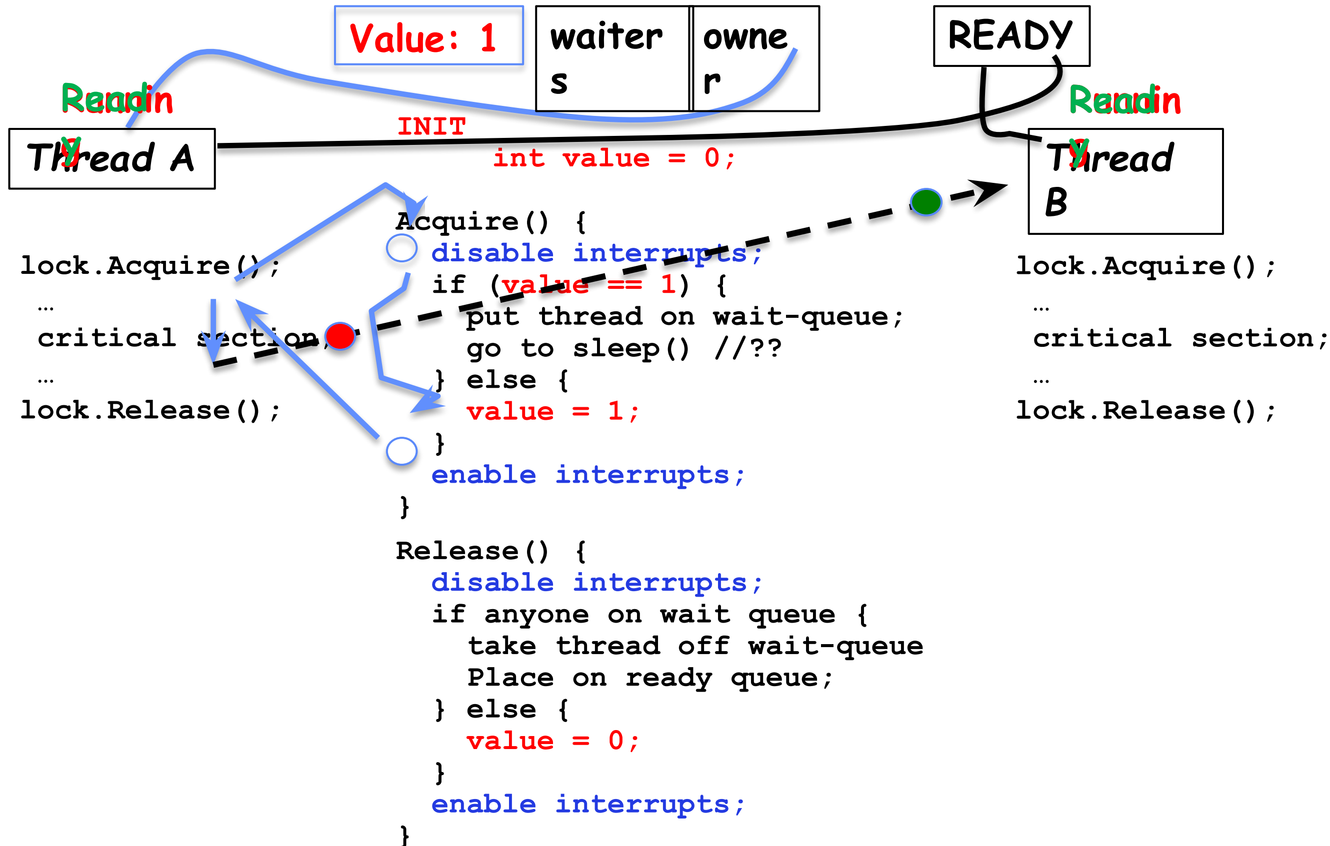
In-Kernel Lock: Simulation



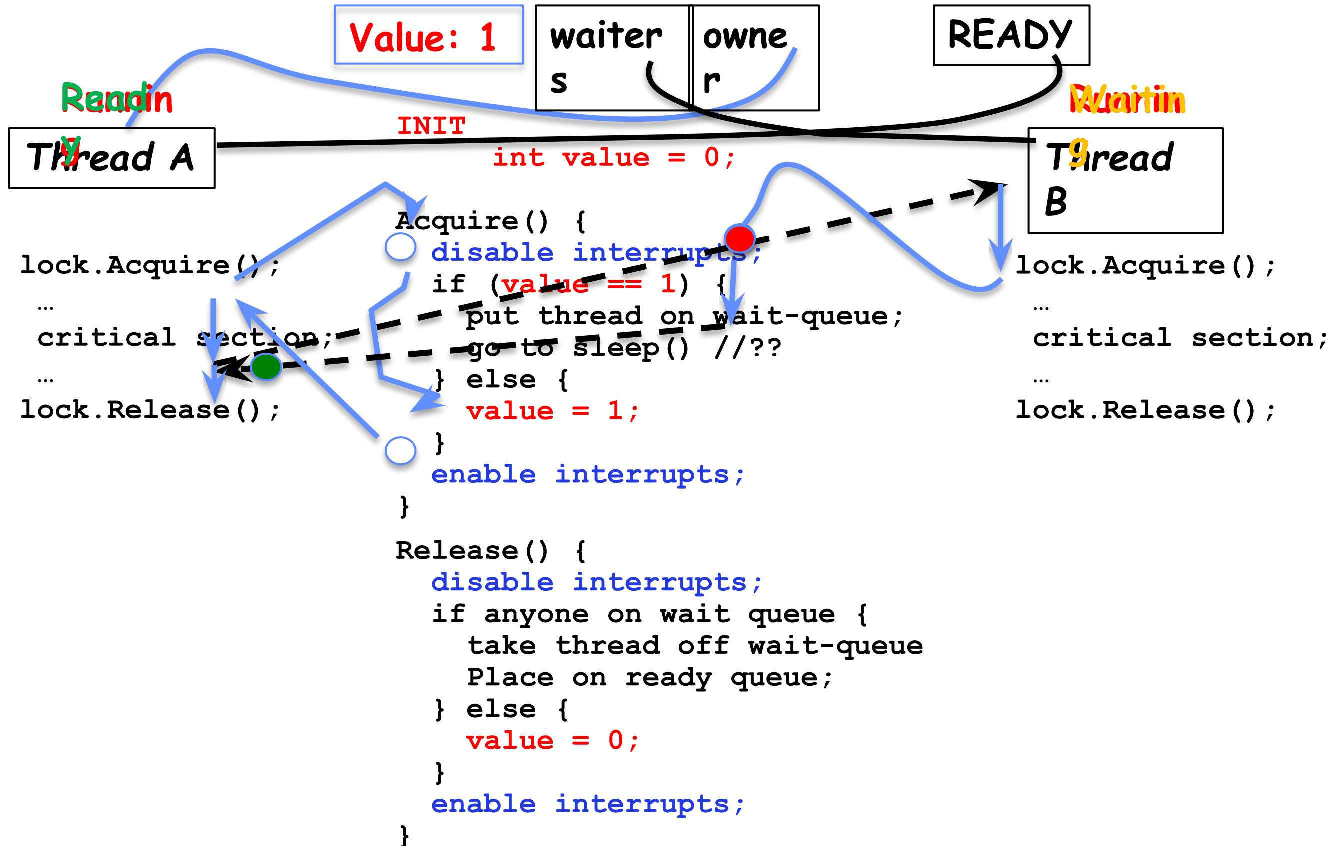
In-Kernel Lock: Simulation



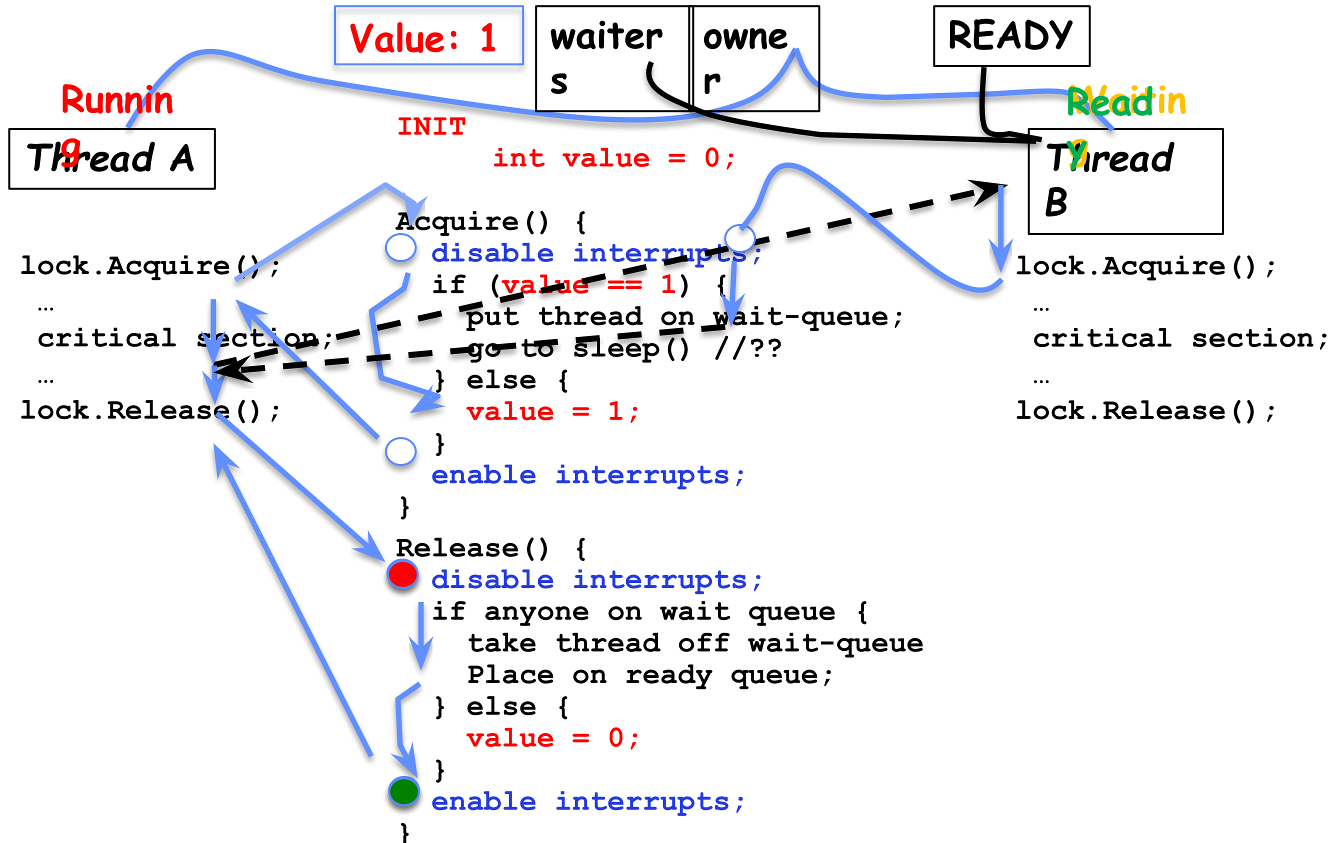
In-Kernel Lock: Simulation



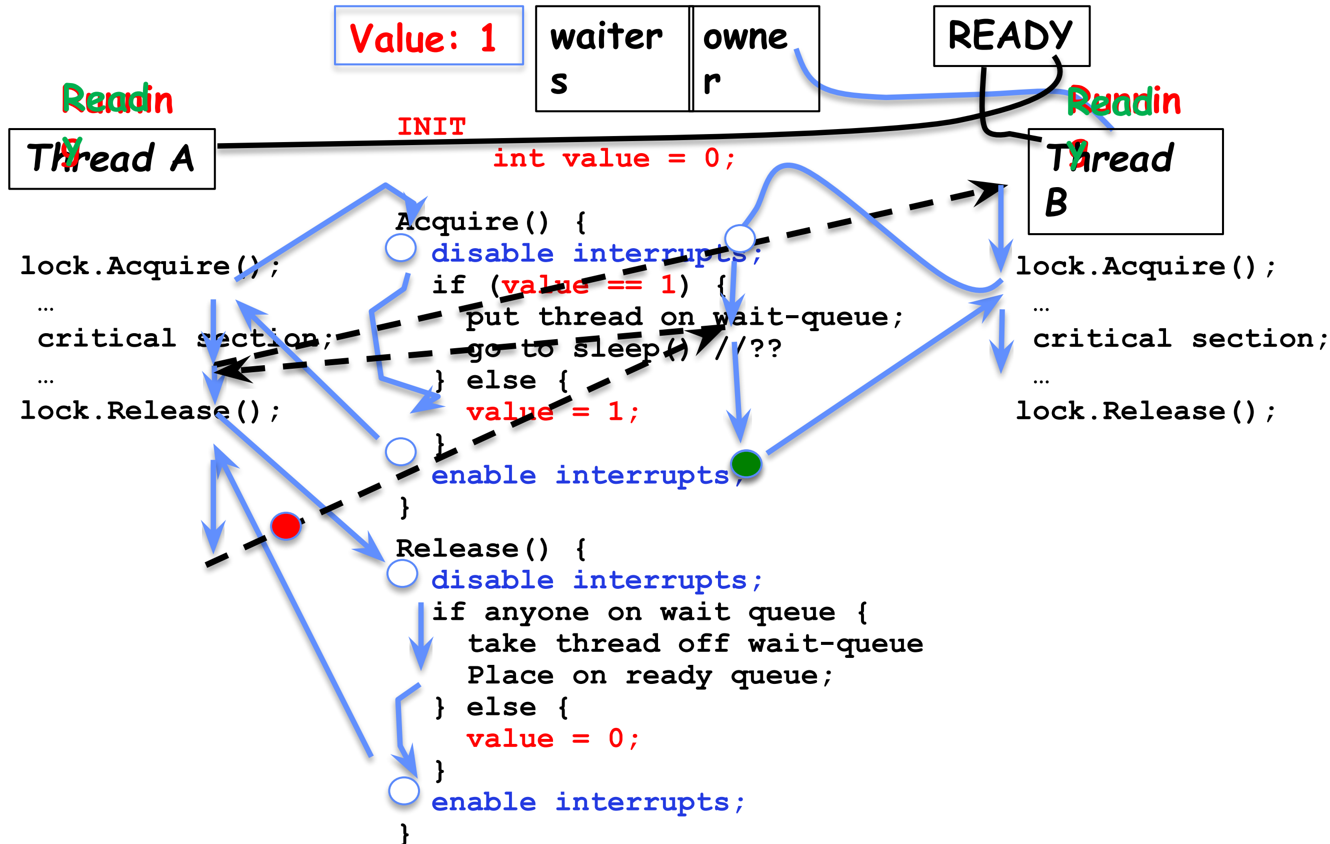
In-Kernel Lock: Simulation



In-Kernel Lock: Simulation



In-Kernel Lock: Simulation



User space implementation

Atomic Read-Modify-Write Instructions

- Problems with previous solution:
  - Can't give lock implementation to users
  - Doesn't work well on multiprocessor
    - » Disabling interrupts on all processors requires messages and would be very time consuming
- Alternative: atomic instruction sequences
  - These instructions read a value and write a new value atomically
  - Hardware is responsible for implementing this correctly
    - » on both uniprocessors (not too hard)
    - » and multiprocessors (requires help from cache coherence protocol)
  - Unlike disabling interrupts, can be used on both uniprocessors and multiprocessors

Examples of Read-Modify-Write

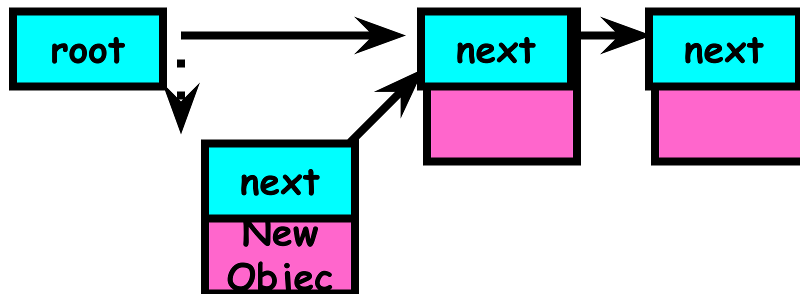
```
• test&set (&address) {           /* most architectures */
      result = M[address];        // return result from "address" and
      M[address] = 1;             // set value at "address" to 1
      return result;
  }
• swap (&address, register) {     /* x86 */
      temp = M[address];          // swap register's value to
      M[address] = register;      // value at "address"
      register = temp;
  }
• compare&swap (&address, reg1, reg2) { /* x86 (returns old value), 68000 */
      if (reg1 == M[address]) {   // If memory still == reg1,
          M[address] = reg2;      // then  put reg2 => memory
          return success;
      } else {                    // Otherwise do not change memory
          return failure;
      }
  }
• load-linked&store-conditional(&address) { /* R4000, alpha */
      loop:
           ll r1, M[address];
           movi r2, 1;           // Can do arbitrary computation
           sc r2, M[address];
           beqz r2, loop;
  }
```

Using of Compare&Swap for queues

- ```
  compare&swap (&address, reg1, reg2) { /* x86, 68000 */
     if (reg1 == M[address]) {
          M[address] = reg2;
          return success;
     } else {
          return failure;
     }
  }
  ```

  Here is an atomic add to linkedlist function:

  ```
  addToQueue(&object) {
     do {     // repeat until no conflict
          ld r1, M[root]    // Get ptr to current head
          st r1, M[object]  // Save link in new object
     } until (compare&swap(&root,r1,object));
  }
  ```


  ```

Implementing Locks with test&set

- Simple lock that doesn't require entry into the kernel:

```
// (Free) Can access this memory location from user space!
int mylock = 0; // Interface: acquire(&mylock);
                //            release(&mylock);
acquire(int *thelock) {
     while (test&set(thelock)); // Atomic operation!
}
release(int *thelock) {
     *thelock = 0;         // Atomic operation!
}
```

- Simple explanation:
  – If lock is free, test&set reads 0 and sets lock=1, so lock is now busy. It returns 0 so while exits.
  – If lock is busy, test&set reads 1 and sets lock=1 (no change) It returns 1, so while loop continues.
  – When we set thelock = 0, someone else can get lock.
- Busy-Waiting: thread consumes cycles while waiting
  – For multiprocessors: every test&set() is a write, which makes value ping-pong around in cache (using lots of network BW)

Problem: Busy-Waiting for Lock

- Positives for this solution
  - Machine can receive interrupts
  - User code can use this lock
  - Works on a multiprocessor
- Negatives
  - This is very inefficient as thread will consume cycles waiting
  - Waiting thread may take cycles away from thread holding lock (no one wins!)
  - Priority Inversion: If busy-waiting thread has higher priority than thread holding lock ⇒ no progress!
- Priority Inversion problem with original Martian rover
- For higher-level synchronization primitives (e.g. semaphores or monitors), waiting thread may wait for an arbitrary long time!
  - Thus even if busy-waiting was OK for locks, definitely not ok for other primitives
  - Homework/exam solutions should avoid busy-waiting!

User Space Lock implementation with Futex and atomic operations

- Three (3) states:
  - UNLOCKED: No one has lock
  - LOCKED: One thread has lock
  - CONTESTED: Possibly more than one (with someone sleeping)
- Lock grabbed cleanly by either
  - `compare_and_swap()`
  - First `swap()`
- No overhead if uncontested!
- Could build semaphores in a similar way!

no one has to be woken up, and so no syscalls at all.

```
typedef enum { UNLOCKED,LOCKED,CONTESTED } Lock;
Lock mylock = UNLOCKED; // Interface:
acquire(&mylock);
                        //
release(&mylock);

acquire(Lock *thelock) {
    // If unlocked, grab lock!
    if (compare&swap(thelock,UNLOCKED,LOCKED))
        return;

    // Keep trying to grab lock, sleep in futex
    while (swap(mylock,CONTESTED) != UNLOCKED))
        // Sleep unless someone releases hear!
        futex(thelock, FUTEX_WAIT, CONTESTED);
}
release(Lock *thelock) {
    // If someone sleeping,
    if (swap(thelock,UNLOCKED) == CONTESTED)
        futex(thelock,FUTEX_WAKE,1);
}
```

- See also futex_demo.c in futex(2) - Linux manual page