CS194-24
Advanced Operating Systems Structures and Implementation
Lecture 8

Synchronization Continued

February 19th, 2014
Prof. John Kubiatowicz
http://inst.eecs.berkeley.edu/~cs194-24

Goals for Today

- Synchronization (Con't)
  - Locks
  - Semaphores
  - Monitors

Interactive is important!
Ask Questions!

Note: Some slides and/or pictures in the following are adapted from slides ©2013

Recall: Using Locks to Protect Shared Variable

- Consider Previous Example:

```
Deposit(acctId, amount) {
  Acquire(depositlock);
  acct = GetAccount(actId);
  acct->balance += amount;
  StoreAccount(acct);
  Release(depositlock);
}
```

Critical Section

- Locking Behavior:
  - Only one critical section can be running at once!
    - » Second Acquire() before release ⇒ second thread waits
  - As soon as Release() occurs, another Acquire() can happen
  - If many threads request lock acquisition at same time:
    - » Might get livelock, depending on what happens on Release()
- Result of using locks: three instructions in critical section become Atomic! (cannot be separated)

Naïve use of Interrupt Enable/Disable

- How can we build multi-instruction atomic operations?
  - Scheduler 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 Ints; }
LockRelease { enable Ints; }
```

- 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?"

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;
}
```


Typical Linux Interfaces

- Disabling and Enabling Interrupts on the Linux Kernel:

```
local_irq_disable();
/* interrupts are disabled ... */
local_irq_enable();
```

  - These operations often single assembly instructions
    - » They *only* work for local processor!
    - » If competing with another processor, must use other form of synchronization
  - Dangerous if called when interrupts already disabled
    - » Then, when you code reenables, you will change semantics
- Saving and restoring interrupt state first:

```
unsigned long flags;

local_irq_save(flags);     // Save state
/* Do whatever, including disable/enable*/
local_irq_restore(flags);  // Restore
```

- State of the system

```
in_interrupt(); // In handler or bottom half
in_irq();       // Specifically in handler
```


Linux Interrupt control (Con't)

- No more global `cli()`!
  - Used to be that `cli()/sti()` could be used to enable and disable interrupts on all processors
  - First deprecated (2.5), then removed (2.6)
    - » Could serialize device drivers across all processors!
    - » Just a bad idea
  - Better option?
    - » Fine-grained spin-locks between processors (more later)
    - » Local interrupt control for local processor
- Disabling specific interrupt (nestable)

```
disable_irq(irq);          // Wait current handlers
disable_irq_nosync(irq); // Don't wait current handler
enable_irq(irq);           // Reenable line
synchronize_irq(irq);      // Wait for current handler
```

  - Not great for buses with multiple interrupts per line, such as PCI! More when we get into device drivers.

How to implement locks? Atomic Read-Modify-Write instructions

- Problem with previous solution?
  - Can't let users disable interrupts! (Why?)
  - 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 from memory 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

Atomic Instructions

- test&set (&address) { /* most architectures */
    result = M[address];
    M[address] = 1;
    return result;
  }
- swap (&address, register) { /* x86 */
    temp = M[address];
    M[address] = register;
    register = temp;
  }
- compare&swap (&address, reg1, reg2) { /* 68000 */
    if (reg1 == M[address]) {
      M[address] = reg2;
      return success;
    } else {
      return failure;
    }
  }
- load-linked&store conditional(&address) {
    /* R4000, alpha */
    loop:
      ll r1, M[address];
      movi r2, 1; /* Can do arbitrary comp */
      sc r2, M[address];
      beqz r2, loop;
  }

Implementing Locks with test&set: Spin Lock

- Another flawed, but simple solution:

```
int value = 0; // Free
Acquire() {
  while (test&set(value)); // while busy
}
Release() {
  value = 0;
}
```

- Simple explanation:
  - If lock is free, test&set reads 0 and sets value=1, so lock is now busy. It returns 0 so while exits.
  - If lock is busy, test&set reads 1 and sets value=1 (no change). It returns 1, so while loop continues
  - When we set value = 0, someone else can get lock
- Problem: While spinning, performing writes
  - Lots of cache coherence data – not capitalizing on cache

Better: Test&Test&Set Lock

- The Test&Test&Set lock:

```
int value = 0; // Free
Acquire() {
  while (true) {
    while(value); // Locked, spin with reads
    if (!test&set(value))
      break;     // Success!
  }
}
Release() {
  value = 0;
}
```

- Significant problems with Test&Test&Set?
  - Multiple processors spinning on same memory location
    - » Release/Reacquire causes lots of cache invalidation traffic
    - » No guarantees of fairness – potential livelock
  - Scales poorly with number of processors
    - » Because of bus traffic, average time until *some* processor acquires lock grows with number of processors
- Busy-Waiting: thread consumes cycles while waiting

Better option: Exponential backoff

- The Test&Test&Set lock with backoff:

```
int value = 0; // Free
Acquire() {
  int nfail = 0
  while (true) {
    while(value); // Locked, spin with reads
    if (!test&set(value))
      break;     // Success!
    else
      exponentialBackoff(nfail++)
  }
}
Release() {
  value = 0;
}
```

- If someone grabs lock between when we see it free and attempt to grab it, must be contention
- Backoff exponentially:
  - Choose backoff time randomly with some distribution
  - Double mean time with each iteration
- Why does this work?

Better Locks using test&set

- Can we build test&set locks without busy-waiting?
  - Can't entirely, but can minimize!
  - Idea: only busy-wait to atomically check lock value

```
int guard = 0;
int value = FREE;
```

```
Acquire() {
  // Short busy-wait time
  while (test&set(guard));
  if (value == BUSY) {
    put thread on wait queue;
    go to sleep() & guard = 0;
  } else {
    value = BUSY;
    guard = 0;
  }
}
```

```
Release() {
  // Short busy-wait time
  while (test&set(guard));
  if anyone on wait queue {
    take thread off wait queue
    Place on ready queue;
  } else {
    value = FREE;
  }
  guard = 0;
```

- Note: sleep has to be sure to reset the guard variable
  - Why can't we do it just before or just after the sleep?

Administrivia

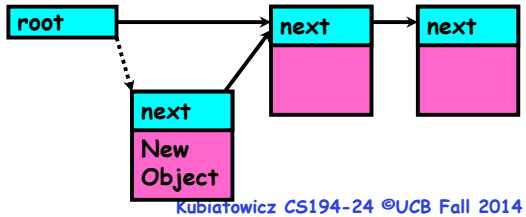
- Midterm I
  - 3 weeks from today
  - All topics up to that Monday
  - No class day of exam
- Readings
  - Love Chapter 6: Data structures
  - Love Chapter 10: Synchronization methods

Using of Compare&Swap for queues

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

Here is an atomic add to linked-list 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));
}
```



Portable Atomic operations in Linux

- Linux provides `atomic_t` for declaring an atomic integer
  - Also, `atomic64_t` for declaring atomic 64-bit variant
  - Not necessarily same as a regular integer!
    - » Originally on SPARC, `atomic_t` ⇒ only 24 of 32 bits usable
- Example usage:

```
atomic_t v;          /* define v */
atomic_t u = ATOMIC_INIT(0); /* define and init u=0 */

atomic_set(&v, 4);  /* v=4 (atomically) */
atomic_add(2, &v);  /* v = v + 2 (atomically) */
atomic_inc(&v);     /* v = v + 1 (atomically) */

int final = atomic_read(&v);  /* final == 7 */
```

- Some operations (see Love, Ch 10, Table 10.1/10.2):

```
atomic_inc()/atomic_dec()   /* Atomically inc/dec */
atomic_add()/atomic_sub()   /* Atomically add/sub */
int atomic_dec_and_test()   /* Sub 1.  True if 0 */
int atomic_inc_return()     /* Add 1, return result */
```

Portable bit operations in Linux

- Atomic bitwise operations operate on regular Ints.
  - For example, to set nrth bit, starting from addr:

```
void set_bit(int nr, void *addr);
```

  - Atomicity ensures that bit transitions are always seen atomically – regardless of competing concurrency
    - » When bit is set and cleared – actually reflected as stores
    - » When two different bits set – end up with two bits set, rather than one set operation erasing result of another
- Some operations (see Love, Ch 10, Table 10.3):

```
void set_bit()            /* Atomically set bit */
void clear_bit()          /* Atomically clear bit */
void change__bit()        /* Atomically toggle bit */
int test_and_set_bit()    /* set bit, return previous*/
int test_and_clear_bit() /* clear bit, return prev */
int test_and_change_bit()/* toggle bit, return prev */
int test_bit()            /* Return value of bit*/
```


Portable Spinlock constructs in Linux

- Linux provides lots of synchronization constructs
  - We will highlight them throughout the term
- Example: Spin Lock support: Not recursive!
  - Only a lock on multiprocessors: Becomes simple preemption disable/enable on uniprocessors

```
#include <linux/spinlock.h>
DEFINE_SPINLOCK(my_lock);

spin_lock(&my_lock);
/* Critical section … */
spin_unlock(&my_lock);
```

- Disable interrupts and grab lock (while saving and restoring state in case interrupts already disabled):

```
DEFINE_SPINLOCK(my_lock);
unsigned long flags;

spin_lock_irqsave(&my_lock, flags);
/* Critical section … */
spin_unlock_irqrestore(&my_lock);
```


Locks (Mutexes) in POSIX Threads

- To create a mutex:

```
#include <pthread.h>
pthread_mutex_t amutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_init(&amutex, NULL);
```

- To use it:

```
int pthread_mutex_lock(amutex);
int pthread_mutex_unlock(amutex);
```

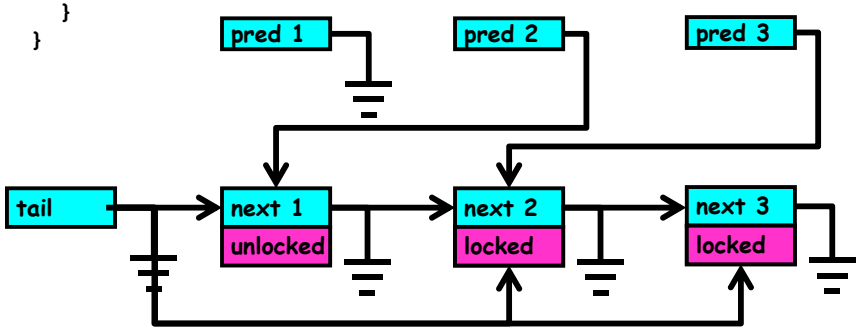
- To deallocate a mutex

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```


Mellor-Crummey-Scott Lock

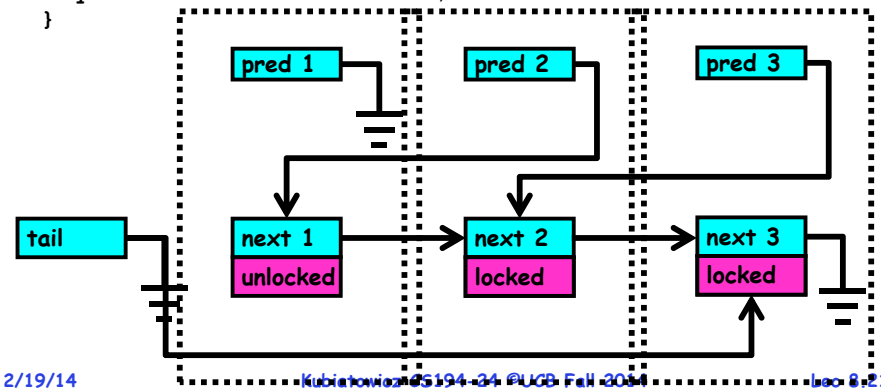
- Another lock-free option – Mellor-Crummey Scott (MCS):

```
public void lock() {
  QNode qnode = myNode.get();
  qnode.next = null;
  QNode pred = swap(tail,qnode);
  if (pred != null) {
    qnode.locked = true;
    pred.next = qnode;
    while (qnode.locked); // wait for predecessor
  }
}
```



Mellor-Crummey-Scott Lock (con't)

```
public void unlock() {
  QNode qnode = myNode.get();
  if (qnode.next == null) {
    if (compare&swap(tail,qnode,null))
      return;
    // wait until predecessor fills in my next field
    while (qnode.next == null);
  }
  qnode.next.locked = false;
}
```



Mellor-Crummey-Scott Lock (Con't)

- Nice properties of MCS Lock
  - Never more than 2 processors spinning on one address
  - Completely fair – once on queue, are guaranteed to get your turn in FIFO order
    - » Alternate release procedure doesn't use compare&swap but doesn't guarantee FIFO order
- Bad properties of MCS Lock
  - Takes longer (more instructions) than T&T&S if no contention
  - Releaser may be forced to spin in rare circumstances
- Hardware support?
  - Some proposed hardware queueing primitives such as QOLB (Queue on Lock Bit)
  - Not broadly available

Comparison between options

- Lots of threads trying to:
  `acquire();`
  `wait for bit();`
  `release();`
- Metric:
  Average time in critical section
  - Total # critical sections/total time
- Lesson: need something to handle high contention situations
  - Exponential backoff
  - Queue lock

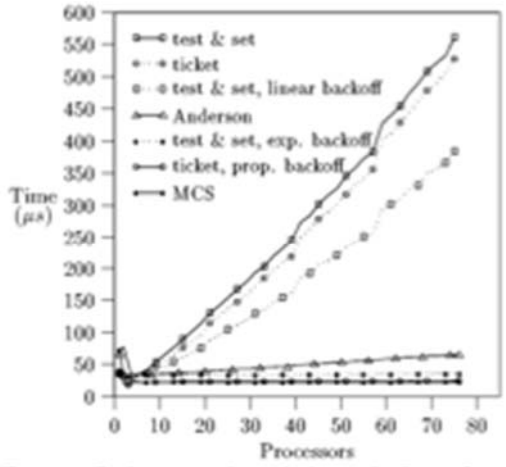
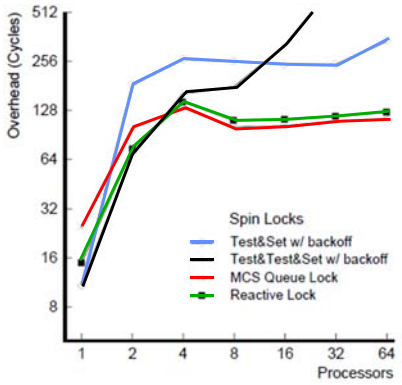


Figure 1: Performance of spin locks on the Butterfly.

Reactive Locks

- Try to determine if you are in a high-contention or low-contention domain
  - Choose a protocol accordingly
  - Dynamic choice
- Why is this useful?
  - Because algorithms go through phases
  - Because different machines have different characteristics



Higher-level Primitives than Locks

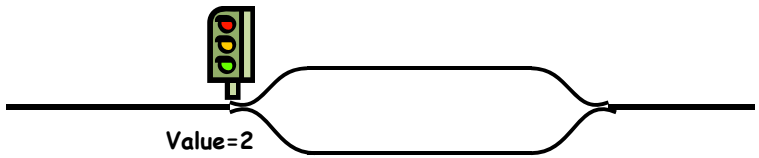
- What is the right abstraction for synchronizing threads that share memory?
  - Want as high a level primitive as possible
- Good primitives and practices important!
  - Since execution is not entirely sequential, really hard to find bugs, since they happen rarely
  - UNIX is pretty stable now, but up until about mid-80s (10 years after started), systems running UNIX would crash every week or so – concurrency bugs
- Synchronization is a way of coordinating multiple concurrent activities that are using shared state
  - This lecture and the next presents a couple of ways of structuring the sharing

Recall: Semaphores

- Semaphores are a kind of generalized lock
  - First defined by Dijkstra in late 60s
  - Main synchronization primitive used in original UNIX
- Definition: a Semaphore has a non-negative integer value and supports the following two operations:
  - P(): an atomic operation that waits for semaphore to become positive, then decrements it by 1
    - » Think of this as the wait() operation
  - V(): an atomic operation that increments the semaphore by 1, waking up a waiting P, if any
    - » This of this as the signal() operation
  - Note that P() stands for "*proberen*" (to test) and V() stands for "*verhogen*" (to increment) in Dutch

Semaphores Like Integers Except

- Semaphores are like integers, except
  - No negative values
  - Only operations allowed are P and V – can't read or write value, except to set it initially
  - Operations must be atomic
    - » Two P's together can't decrement value below zero
    - » Similarly, thread going to sleep in P won't miss wakeup from V – even if they both happen at same time
- Semaphore from railway analogy
  - Here is a semaphore initialized to 2 for resource control:



Two Uses of Semaphores

- Mutual Exclusion (initial value = 1)
  - Also called "Binary Semaphore".
  - Can be used for mutual exclusion:

```
semaphore.P();
// Critical section goes here
semaphore.V();
```

- Scheduling Constraints (initial value = 0)
  - Locks are fine for mutual exclusion, but what if you want a thread to wait for something?
  - Example: suppose you had to implement ThreadJoin which must wait for thread to terminiate:

```
Initial value of semaphore = 0
ThreadJoin {
    semaphore.P();
}
ThreadFinish {
    semaphore.V();
}
```

Producer-Consumer with a Bounded Buffer

- Problem Definition
  - Producer puts things into a shared buffer (wait if full)
  - Consumer takes them out (wait if empty)
  - Use a fixed-size buffer between them to avoid lockstep
    - » Need to synchronize access to this buffer
- Correctness Constraints:
  - Consumer must wait for producer to fill buffers, if none full (scheduling constraint)
  - Producer must wait for consumer to empty buffers, if all full (scheduling constraint)
  - Only one thread can manipulate buffer queue at a time (mutual exclusion)
- Remember why we need mutual exclusion
  - Because computers are stupid
- General rule of thumb:
  Use a separate semaphore for each constraint
  - `Semaphore fullBuffers; // consumer's constraint`
  - `Semaphore emptyBuffers;// producer's constraint`
  - `Semaphore mutex;  // mutual exclusion`

Full Solution to Bounded Buffer

```
Semaphore fullBuffer = 0; // Initially, no coke
Semaphore emptyBuffers = numBuffers;
                          // Initially, num empty slots
Semaphore mutex = 1;      // No one using machine

Producer(item) {
    emptyBuffers.P();     // Wait until space
    mutex.P();            // Wait until buffer free
    Enqueue(item);
    mutex.V();
    fullBuffers.V();      // Tell consumers there is
                          // more coke
}
Consumer() {
    fullBuffers.P();      // Check if there's a coke
    mutex.P();            // Wait until machine free
    item = Dequeue();
    mutex.V();
    emptyBuffers.V();     // tell producer need more
    return item;
}
```


Portable Semaphores in Linux

- Initialize general semaphore:

```
struct semapore my_sem;
sema_init(&my_sem, count);    /* Initialize semaphore */
```

- Initialize mutex (semaphore with count = 1)

```
static DECLARE_MUTEX(my_mutex);
```

- Acquire semaphore:

```
down_interruptible(&my_sem);  // Acquire sem, sleeps with
                              // TASK_INTERRUPTIBLE state
down(&my_sem);                // Acquire sem, sleeps with
                              // TASK_UNINTERRUPTIBLE state
down_trylock(&my_sem);        // Return ≠ 0 if lock busy
```

- Release semaphore:

```
up(&my_sem);                  // Release lock
```


Portable (native) mutexes in Linux

- Interface optimized for mutual exclusion

```
DEFINE_MUTEX(my_mutex);   /* Static definition */

struct mutex my_mutex;
mutex_init(&my_mutex);          /* Dynamic mutex init */
```

- Simple use pattern:

```
mutex_lock(&my_mutex);
/* critical section … */
mutex_unlock(&my_mutex);
```

- Constraints
  - Same thread that grabs lock must release it
  - pProcess cannot exit while holding mutex
  - Recursive acquisitions not allowed
  - Cannot use in interrupt handlers (might sleep!)
- Advantages
  - Simpler interface
  - Debugging support for checking usage
- Prefer mutexes over semaphores unless mutex really doesn't fit your pattern!

Alternative: Completion Patterns

- One use pattern that does not fit mutex pattern:
  - Start operation in another thread/hardware container
  - Sleep until woken by completion of event
- Can be implemented with semaphores
  - Start semaphore with count of 0
  - Immediate down() – puts parent to sleep
  - Woken with up()
- More efficient: use "completions":

```
DEFINED_COMPLETION();          /* Static definition */

struct completion my_comp;
init_completion(&my_comp);     /* Dynamic comp init */
```

- One or more threads to sleep on event:

```
wait_for_completion(&my_comp);  /* put thead to sleep */
```

- Wake up threads (can be in interrupt handler!)

```
complete(&my_comp);
```


Review: Definition of Monitor

- Semaphores are confusing because dual purpose:
  - Both mutual exclusion and scheduling constraints
  - Cleaner idea: Use *locks* for mutual exclusion and *condition variables* for scheduling constraints
- Monitor: a lock and zero or more condition variables for managing concurrent access to shared data
  - Use of Monitors is a programming paradigm
- Lock: provides mutual exclusion to shared data:
  - Always acquire before accessing shared data structure
  - Always release after finishing with shared data
- Condition Variable: a queue of threads waiting for something *inside* a critical section
  - Key idea: allow sleeping inside critical section by atomically releasing lock at time we go to sleep
  - Contrast to semaphores: Can't wait inside critical section

Review: Programming with Monitors

- Monitors represent the logic of the program
  - Wait if necessary
  - Signal when change something so any waiting threads can proceed
- Basic structure of monitor-based program:

```
lock
while (need to wait) {
   condvar.wait();
}
unlock

do something so no need to wait

lock

condvar.signal();

unlock
```

Check and/or update state variables, Wait if necessary

Check and/or update state variables

Summary (Synchronization)

- Important concept: Atomic Operations
  - An operation that runs to completion or not at all
  - These are the primitives on which to construct various synchronization primitives
- Talked about hardware atomicity primitives:
  - Disabling of Interrupts, test&set, swap, comp&swap, load-linked/store conditional
- Showed several constructions of Locks
  - Must be very careful not to waste/tie up machine resources
    - » Shouldn't disable interrupts for long
    - » Shouldn't spin wait for long
  - Key idea: Separate lock variable, use hardware mechanisms to protect modifications of that variable
- Talked about Semaphores, Monitors, and Condition Variables
  - Higher level constructs that are harder to "screw up"