Thread Coordination: Basic Lock Implementation

David E. Culler
CS162 – Operating Systems and Systems Programming
Lecture 9
Sept 19, 2014

Reading: A&D 5.7-5.9
HW 2 out
Proj 1 out: CP1
Objectives

• Demonstrate a structured way to approach concurrent programming (of threads)
  – Synchronized shared objects (in C!)
• Introduce the challenge of concurrent programming
• Develop understanding of a family of mechanisms
  – Flags, Locks, Condition Variables & semaphores
• Understand how these mechanisms can be implemented
Concurrency Coordination Landscape

Concurrent Applications

Shared Coordinated Objects

Synchronization Variables

Atomic Operations

Hardware

- Concurrency
- Coordination Landscape
- Concurrent Applications
- Shared Coordinated Objects
- Synchronization Variables
- Atomic Operations
- Hardware

- Bounded Queue
- Ordered List
- Dictionary
- Barrier
- Monitors
- Semaphore
- Condition Variables
- Locks
- Flag
- Interrupt Disable/Enable
- Test-and-Set
- Interrupts
- Controllers
- Multiple Processors
- cmp&swap
- xchng
- fetch&inc
- LL + SC

lecture 8
Recall

• Two key aspects of coordination
  – Mutually exclusive access to shared objects so that they can be manipulated correctly
  – Conveying precedence from one computational entity to another

• Atomic: sequence of actions that is indivisible (from a certain perspective)

• Critical section: segment of computation that is performed under exclusive control
  – While locking others out
### Illustration: “Too much milk”

<table>
<thead>
<tr>
<th>Time</th>
<th>Person A</th>
<th>Person B</th>
</tr>
</thead>
<tbody>
<tr>
<td>3:00</td>
<td>Look in Fridge. Out of milk</td>
<td></td>
</tr>
<tr>
<td>3:05</td>
<td>Leave for store</td>
<td></td>
</tr>
<tr>
<td>3:10</td>
<td>Arrive at store</td>
<td>Look in Fridge. Out of milk</td>
</tr>
<tr>
<td>3:15</td>
<td>Buy milk</td>
<td>Leave for store</td>
</tr>
<tr>
<td>3:20</td>
<td>Arrive home, put milk away</td>
<td>Arrive at store</td>
</tr>
<tr>
<td>3:25</td>
<td></td>
<td>Buy milk</td>
</tr>
<tr>
<td>3:30</td>
<td></td>
<td>Arrive home, put milk away ...</td>
</tr>
</tbody>
</table>
Definitions

- **Synchronization**: using atomic operations to ensure cooperation between threads
  - For now, only loads and stores are atomic
  - We’ll show that it is hard to build anything useful with only reads and writes

- **Critical Section**: piece of code that only one thread can execute at once

- **Mutual Exclusion**: ensuring that only one thread executes critical section
  - One thread excludes the other while doing its task
  - Critical section and mutual exclusion are two ways of describing the same thing
Too Much Milk: non-Solution

• Still too much milk but only occasionally!

```
Thread A
if (noMilk)
  if (noNote) {
    if (noMilk)
      if (noNote) {
        leave Note;
        buy milk;
        remove note;
      }
    leave Note;
    buy milk;
  }
```

• Thread can get context switched after checking milk and note but before leaving note!

• Solution makes problem worse since fails intermittently
  – Makes it really hard to debug…
  – Must work despite what the thread dispatcher does!
Recall: Simplest synchronization

- Alternating protocol of a single producer and a single consumer can be coordinated by a simple flag
- Integrated with the shared object

```c
typedef struct sharedobject {
    FILE *rfile;
    int flag;
    int linenum;
    char *line;
} so_t;

int markfull(so_t *so) {
    so->flag = 1;
    while (so->flag) {}
    return 1;
}

int markempty(so_t *so) {
    so->flag = 0;
    while (!so->flag) {}
    return 1;
}
```
More Definitions

• **Lock**: prevents someone from doing something
  – Lock before entering critical section and before accessing shared data
  – Unlock when leaving, after accessing shared data
  – Wait if locked
    • Important idea: all synchronization involves waiting

• Example: fix the milk problem by putting a lock on refrigerator
  – Lock it and take key if you are going to go buy milk
  – Fixes too much (coarse granularity): roommate angry if only wants orange juice

  – Of Course – We don’t know how to make a lock yet
Too Much Milk: Solution

• Suppose we have some sort of implementation of a lock (more in a moment)
  – `Lock.Acquire()` – wait until lock is free, then grab
  – `Lock.Release()` – unlock, waking up anyone waiting
  – These must be atomic operations – if two threads are waiting for the lock, only one succeeds to grab the lock

• Then, our milk problem is easy:

```java
milklock.Acquire();
if (nomilk)
    buy milk;
milklock.Release();
```

• Once again, section of code between `Acquire()` and `Release()` called a “Critical Section”
How to Implement Lock?

- **Lock**: prevents someone from accessing something
  - Lock before entering critical section (e.g., before accessing shared data)
  - Unlock when leaving, after accessing shared data
  - Wait if locked
    - Important idea: all synchronization involves waiting
    - Should sleep if waiting for long time

- Hardware lock instructions?
  - Is this a good idea?
    - We will see various atomic read-modify-write instructions
  - What about putting a task to sleep?
    - How do handle interface between hardware and scheduler?
  - Complexity?
    - Each feature makes hardware more complex and slower
• 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 Ints; }
  LockRelease { enable Ints; }
Lock vs Disable

Only disable for the implementation of the lock itself
Not what you are going to do under it!

```
LockAcquire { disable Ints; }  
While(TRUE) {;}  
LockRelease { enable Ints; }  
```
An OS Implementation of Locks

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

```c
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
        Put at front of ready queue
    } else {
        value = FREE;
    }
    enable interrupts;
}
```

Checking and Setting are indivisible - otherwise two thread could see !BUSY

Critical Section
Locks

```
int value = 0;
Acquire() {
    disable interrupts;
    if (value == 1) {
        put thread on wait-queue;
        go to sleep() //??
    } else {
        value = 1;
        enable interrupts;
    }
}

Release() {
    disable interrupts;
    if anyone on wait queue {
        take thread off wait-queue
        Place on ready queue;
    } else {
        value = 0;
    }
    enable interrupts;
}
```

lock.Acquire();
...
critical section;
...
lock.Release();

If one thread in critical section, no other activity (including OS) can run!
Interrupt re-enable in going to sleep

- What about re-enabling ints when going to sleep?
  ```c
  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!)

- Want to put it after sleep(). But, how?
How to Re-enable After Sleep()?

- Since ints 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

---

Thread A

- 
- 
- disable ints
- sleep
- 

Thread B

- 
- 
- yield return
- enable ints
- 
- 
- disable int
- yield

---

- context switch!
Administrative Break

• hmmm

• HW2: experience with sockets & fork
  – experience with threads as separate exercise

• Proj 1:
  – think, read, think, design, simple start, think, write
  – then code code code
Semaphores

- Semaphores are a kind of generalized locks
  - 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)
  – Allow thread 1 to wait for a signal from thread 2, i.e., thread 2 schedules thread 1 when a given constrained is satisfied
  – Example: suppose you had to implement ThreadJoin which must wait for thread to terminate:
    ```
    Initial value of semaphore = 0
    ThreadJoin {
        semaphore.P();
    }
    ThreadFinish {
        semaphore.V();
    }
    ```
Structured concurrent programming

• Use locks for mutual exclusion
  – Including manipulation of data structures
  – Locks more structured than semaphores
    • Ownership: acquirer must release

• Use Condition Variables (more soon) for Scheduling constraints
  – A => B. “stateless”

• Integrate these into concurrent objects
  – Synchronized methods effect the protocol

• But ...
A thread-safe function is one that can be safely (i.e., it will deliver the same results regardless of whether it is) called from multiple threads at the same time.

pthreads_mutex_t mymalloclock;

void *my_malloc(size_t size) {
    void *res;
    pthread_mutex_lock(&mymalloclock);
    res = malloc(size);
    pthread_mutex_unlock(&mymalloclock);
    return res;
}

void my_free(void *ptr) {
    ...
}

...
Thread <> Interrupt Handler

• Interrupt handlers are not threads
• Only threads can share locks
  – Ownership
• Yet in the kernel interrupt handlers and threads need to coordinate access to shared data structures

• The statefull aspect of semaphores makes the pending waiters work
eg. Pintos Locks (synch.c)

```c
void lock_init (struct lock *lock) {
    ASSERT (lock != NULL);
    lock->holder = NULL;
    sema_init (&lock->semaphore, 1);
}

void lock_acquire (struct lock *lock) {
    ASSERT (lock != NULL); ASSERT (!intr_context ());
    ASSERT (!lock_held_by_current_thread (lock));
    
    sema_down (&lock->semaphore);
    lock->holder = thread_current ();
}

void lock_release (struct lock *lock) {
    ASSERT (lock != NULL);
    ASSERT (lock_held_by_current_thread (lock));
    
    lock->holder = NULL;
    sema_up (&lock->semaphore);
}
```

- Implements semaphores for synchronization and builds locks and CVs on top.
void sema_down (struct semaphore *sema) {
    enum intr_level old_level;

    ASSERT (sema != NULL);
    ASSERT (!intr_context ());

    old_level = intr_disable ();
    while (sema->value == 0)
    {
        list_push_back (&sema->waiters,
                        &thread_current ()->elem);
        thread_block ();
    }
    sema->value--;
    intr_set_level (old_level);
}
void sema_down (struct semaphore *sema)
{
    enum intr_level old_level;

    ASSERT (sema != NULL);
    ASSERT (!intr_context ());

    old_level = intr_disable ();
    ASSERT (intr_get_level () == INTR_OFF);
    ASSERT (cur->status != THREAD_RUNNING);
    ASSERT (is_thread (next));

    if (cur != next)
    {
        prev = switch_threads (cur, next);
        thread_schedule_tail (prev);
    }
    thread_current ()->status = THREAD_BLOCKED;

    while (sema->value == 0)
    {
        list_push_back (&sema->waiters, &thread_current ()->elem);
        thread_block ()
    }
    sema->value--;
    intr_set_level (old_level);
}

static void schedule (void) {
    struct thread *cur = running_thread ();
    struct thread *next = next_thread_to_run ();
    struct thread *prev = NULL;

    ASSERT (intr_get_level () == INTR_OFF);
    ASSERT (cur->status != THREAD_RUNNING);
    ASSERT (is_thread (next));

    if (cur != next)
    {
        prev = switch_threads (cur, next);
        thread_schedule_tail (prev);
    }
    thread_current ()->status = THREAD_BLOCKED;

    while (sema->value == 0)
    {
        list_push_back (&sema->waiters, &thread_current ()->elem);
        thread_block ()
    }
    sema->value--;
    intr_set_level (old_level);
}
void sema_down (struct semaphore *sema) {
    enum intr_level old_level;

    ASSERT (sema != NULL);
    ASSERT (!intr_context());

    old_level = intr_disable();

    while (sema->value == 0) {
        list_push_back (&sema->waiters, &thread_current()->elem);
        thread_block();
    }

    sema->value--;
    intr_set_level (old_level);
}

void thread_block (void) {
    ASSERT (!intr_context());
    ASSERT (intr_get_level() == INTR_OFF);

    thread_current()->status = THREAD_BLOCKED;
    schedule();
}

static void schedule (void) {
    struct thread *cur = running_thread();
    struct thread *next = next_thread_to_run();
    struct thread *prev = NULL;

    ASSERT (intr_get_level() == INTR_OFF);
    ASSERT (cur->status != THREAD_RUNNING);
    ASSERT (is_thread(next));

    if (cur != next)
        prev = switch_threads(cur, next);
    thread_schedule_tail(prev);
}

static void switch_threads (void) {
    # Save caller's register state.
    pushl %ebx
    pushl %ebp
    pushl %esi
    pushl %edi

    # Get offset of (struct thread, stack).
    .globl thread_stack_ofs
    mov thread_stack_ofs, %edx

    # Save current stack pointer to old thread's stack, if any.
    movl SWITCH_CUR(%esp), %eax
    movl %esp, (%eax,%edx,1)

    # Restore stack pointer from new thread's stack.
    movl SWITCH_NEXT(%esp), %ecx
    movl (%ecx,%edx,1), %esp

    # Restore caller's register state.
    popl %edi
    popl %esi
    popl %ebp
    popl %ebx

    ret
}

sema->value--;
intr_set_level (old_level);
void sema_up (struct semaphore *sema) {
  enum intr_level old_level;

  ASSERT (sema != NULL);

  old_level = intr_disable ();
  if (!list_empty (&sema->waiters))
    thread_unblock (list_entry (list_pop_front (&sema->waiters),
                                 struct thread, elem));
  sema->value++;
  intr_set_level (old_level);
}
Concurrent Applications

Shared Coordinated Objects

Synchronization Variables

Atomic Operations

Hardware

Concurrent Applications

- Bounded Queue
- Flag
- Ordered List
- Dictionary
- Barrier
- Monitors

Synchronization Variables

- Locks
- Condition Variables
- Semaphore
- Test-and-Set
- xchng
- cmp&swap
- fetch&inc
- LL + SC

Shared Coordinated Objects

- Ordered List
- Dictionary
- Barrier
- Monitors

Atomic Operations

- Interrupt Disable/Enable
- Interrupts
- Controllers
- Multiple Processors

Hardware

- Interrupts
- Controllers
- Multiple Processors