Lecture 5

Threads and Pthreads II

Christopher Mitchell, Ph.D.

cmitchell@cs.nyu.edu || http://z80.me
• We’re exploring the layers of an application running on top of multicore hardware.

- Task-oriented application (* Lecture 4)
- Concurrent Objects (*)
- Tasks (*) and synchronization primitives (today)
Today’s Agenda

• Add the grader; email both of us your commit hashes
  • Github: Nitesh (Github account: nits3392)
  • Email: ns3664@nyu.edu

• Concrete Pthread Creation

• A Simple Mutual Exclusion Algorithm

• Semaphores and “Blocking” Mutices

• Condition Variables and the Wakeup Problem
Concrete Pthread Creation

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 4

void *printHello(void *thread_id) {
    size_t tid = (size_t)thread_id;
    printf("Hello World! It's me, thread \%ld!\n", tid);
    pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
    pthread_t threads[NUM_THREADS];
    for(size_t t = 0; t < NUM_THREADS; t++) {
        printf("In main: creating thread \%ld\n", t);
        int rval = 0;
        if (0 != (rval = pthread_create(&threads[t], NULL,
            printHello, (void *)t))) {
            fprintf(stderr, "ERROR; return code from pthread_create() is \%d\n", rval);
            break;
        }
        pthread_detach(threads[t]);
    }
    pthread_exit(NULL);
    return 0;
}
```
Understanding Mutices

• The literature on mutual exclusion algorithms is extensive.

• In practice, mutual exclusion is implemented with some form of hardware and OS support (coming soon!)

• But we’ll first develop our algorithmic and correctness evaluation skills by looking at some very elegant solutions that use only shared memory.
Understanding Mutices: Review

• Mutual exclusion algorithms will provide two methods, `lock()` and `unlock()`, that allow us to mark the beginning and the end of critical sections in our code.
  • `lock()` will block the caller until there is no other thread in the critical section
  • `unlock()` would allow other thread in the critical section if there’s one waiting
Peterson’s Algorithm

• Mutual exclusion solution in “three” lines of code.

```c
int victim;
bool flag[2] = { false, false };

void lock(void) {
    int me = my_tid % 2
    int other = 1 - me
    1 flag[me] = true
    2 victim = me
    3 while (flag[other] &&
             victim == me) {
             no-op;
        }
}

void unlock(void) {
    int me = my_tid % 2
    flag[me] = false
}
```
Uncontested Case

• Solved in two simple tests and no wait.

```c
int victim = 0;
bool flag[2] = { true, false };

void lock(void) {
    int me = my_tid % 2
    int other = 1 - me
    flag[me] = true
    victim = me
    while (flag[other] &&
        victim == me) {
        no-op;
    }
}

void unlock(void) {
    int me = my_tid % 2
    flag[me] = false
}
```

Thread 0

Current State
void unlock(void) {
    int me = my_tid % 2
    flag[me] = false
}

void lock(void) {
    int me = my_tid % 2
    int other = 1 - me
    flag[me] = true
    victim = me
    while (flag[other] && victim == me) {
        no-op;
    }
}

int victim = 1;
bool flag[2] = { true, true };
Simultaneous Case

• Assumes both cannot write to same mem addr at once.

```c
void lock(void) {
    int me = my_tid % 2
    int other = 1 - me
    flag[me] = true
    victim = me
    while (flag[other] &&
    victim == me) {
        no-op;
    }
}

int victim = ?;
bool flag[2] = { true, true };

void unlock(void) {
    int me = my_tid % 2
    flag[me] = false
}
```

Current State

**Mutices**  **Semaphores**  **Cond Vars**
Analyzing Peterson’s Algorithm

• Three criteria
  1. Mutual exclusion
  2. Progress (no deadlock)
  3. Bounded waiting (fairness)
Analyzing Peterson’s Algorithm

• Does Peterson’s Algorithm guarantee mutual exclusion?

• Assume that both threads could pass the tests:
  while (flag[other] && victim == me)

• This would have meant that each would have set the victim to be itself and each would have seen the victim as the other thread

• Is this possible?
Analyzing Peterson’s Algorithm

• Is it starvation free? Fair?
• If a thread `unlock()`s then it would set victim to be itself
• The contenting thread would have the chance to pass `lock()` then.
• Now, nothing prevents the `unlock()` thread from going ahead and trying a `lock()` again.
  • But in that case, that thread changes the victim to be itself before getting in the loop.
Deadlock on Simultaneous Case

• Using flag[] alone doesn’t work
• Could it be simpler? Instead of flag[] and victim, let’s have one or the other

```c
bool flag[2] = { true, true }; // Current State

void lock(void) {
    int me = my_tid % 2
    int other = 1 - me
    flag[me] = true
    while (flag[other]) {
        no-op;
    }
}

void unlock(void) {
    int me = my_tid % 2
    flag[me] = false
}
```
Deadlock on the Uncontested Case

• Peterson’s is as simple as it gets.
• Now with ‘victim’ only

```c
int victim;

void lock(void) {
    int me = my_tid % 2
    int other = 1 - me
    victim = true
    while (flag[other]) {
        no-op;
    }
}

void unlock(void) {
}
```
Observations

• We’ll be using the reasoning we developed here in other algorithms.

• Peterson’s lock is important because it was arguably the first one to show 2-thread mutual exclusion can be solved in a simple way

• It makes several assumptions, though, and these turn out not to be so in practice
  • Number of threads is known a priori
  • Program execution is strictly sequential and each instruction is atomic

• The algorithm uses busy waiting
Semaphores

• Invented by Edsger Dijkstra in 1968.

• A synchronization primitive that enables waiting without busy-wait

• A semaphore has an initial value, usually 1 (binary semaphore), and two operations: `up()` and `down()`

• There are some differences on that interface in practice (and some flavors of interface)
  • We’ll reason using up/down
Semaphores

- **down()**
  - Atomically check that value is greater than 0 and decrement it, allowing the thread to continue
  - Otherwise, suspend the thread, waiting on the counter value to be greater than 0

- **up()**
  - Atomically increment the counter.
Exercise: Semaphor-Based Mutex

- Required methods: `lock()` and `unlock()`
- Assume unlocking an unlocked mutex is okay
  - ie, handle this case properly
  - Without needing to handle this, we can use a single semaphore initially set to 1 as a mutex!
- Semaphores are powerful!
Semaphore-Based “Blocking” Mutex

S: private semaphore, initial count 1
locked: Boolean, initially false
holder: thread ID
M: private semaphore, initial count 1

```c
void lock(void) {
    M.down();
    S.down(); // Critical v
    locked = true;
    holder = self();
    S.up();   // Critical ^
}
```

```c
void unlock(void) {
    S.down(); // Critical v
    if (!locked ||
        holder != self()) {
        S.up();
        return;
    }
    locked = false;
    S.up();   // Critical ^
    M.up();
}
```
Semaphore-Based “Blocking” Mutex

- Starvation free!
  - `lock()` can block at `M.down()`, but only until `unlock()`
  - Nothing can prevent `unlock()`

```c
void lock(void) {
  M.down();
  S.down(); // Critical v
  locked = true;
  holder = self();
  S.up();   // Critical ^
}

void unlock(void) {
  S.down(); // Critical v
  if (!locked ||
    holder != self()) {
    S.up();
    return;
  }
  locked = false;
  S.up();   // Critical ^
  M.up();
}
```
Semaphore-Based "Blocking" Mutex

- Deadlock free!
  - $S$ protects the critical sections, never left locked.
- No cycles

```c
void lock(void) {
    M.down();
    S.down(); // Critical \n
    locked = true;
    holder = self();
    S.up(); // Critical ^
}
```

```c
void unlock(void) {
    S.down(); // Critical \n
    if (!locked ||
        holder != self()) {
        S.up();
        return;
    }

    locked = false;
    S.up(); // Critical ^
    M.up();
}
```
Semaphore Considerations

• In order to sleep instead of busy wait, need to call the OS
  • But if at each semaphore operation we incur one system call, doing synchronization may be expensive
• “Ideal” mutex (“blocking”)
  • Use shared memory to store the state of the lock
  • Uncontended case requires access to memory only
  • In contended case, ask OS to sleep until value of shared memory changes

• Reasoning about semaphore algorithms can be daunting.
Condition Variable Semantics

- Allows a thread to wait on a given predicate to change
- Associated with a mutex that protects the predicate state
- Operations
  - `wait()` – atomatically suspends the execution of the thread and unlock the associated mutex
  - `signal()` – if there’s at least one thread suspended on the cond var, then dequeue it and resume execution, again, atomically
  - `broadcast()` – if there are any threads suspended on the cond var, resume execution for all of them. They’ll contend for the associated lock.
Implementing `wait()` and `signal()`

- `wait()` is always inside a loop that checks the predicate
  - Easier to implement in terms of thread scheduling
  - Allows signal on every predicate change

- `signal()` should be done inside the lock (although it might be correct to do so outside)

```c
// element dequeue()  
pthread_mutex_lock(queue_lock)  
...  
while (empty) {  
    pthread_cond_wait(cond, queue_lock)  
}  
// !empty  
...  
```

```c
// enqueue(element)  
pthread_mutex_lock(queue_lock)  
...  
empty = false  
pthread_cond_signal(cond)  
...  
```
Fair Condition Variables Using Semaphores

• Reported by Andrew Birrell in 1993

cond has: a queue of waiters and a semaphore
thread has: a private semaphore with initial count 0

wait(cond, mutex) {
    // assumes mutex held
    // assumes cond is false
    Thread self = Thread.self()
    cond.sem.down()
    enqueue(self) on cond's queue
    cond.sem.up()
    release mutex
    self.sem.down()
    acquire mutex
}

signal(cond) {
    // assumes mutex held
    cond.sem.down()
    if a thread t is waiting {
        dequeue(t)
        t.sem.up()
    }
    cond.sem.up()
}

Fair Condition Variables Using Semaphores

- Reported by Andrew Birrell in 1993

cond has: a queue of waiters and a semaphore
thread has: a private semaphore with initial count 0

```
wait(cond, mutex) {
    // assumes mutex held
    // assumes cond is false
    Thread self = Thread.self()
    cond.sem.down()  
    enqueue(self) on cond's queue
    cond.sem.up()  
    release mutex
    self.sem.down()  
    acquire mutex
}

signal(cond) {
    // assumes mutex held
    cond.sem.down()
    if a thread t is waiting {
        dequeue(t)
        t.sem.up()
    }
    cond.sem.up()
}
```

Equivalent to atomically releasing the lock and going to sleep. What if `t.sem.up()` runs before `self.sem.down()`? What could happen if we released the mutex lock earlier?
Lost Wakeup Race Condition

- A deadly (and common) mistake

```java
wait(cond, mutex)
    // assumes mutex held
    // assumes cond is false
    release mutex
    Thread self = Thread.self()
    cond.sem.down()
    enqueue(self) on cond's queue
    cond.sem.up()
    self.sem.down()
    acquire mutex

signal(cond)
    // assumes mutex held
    cond.sem.down()
    if a thread t is waiting
    dequeue(t)
    t.sem.up()
    cond.sem.up()
```
Conclusion

• Mutices: How can we build our own?
• Semaphores: Making mutices, working with the OS
• Cond Vars: Implementation gotchas

• Lab 1: Questions? Problems?