We’re exploring the layers of an application running on top of multicore hardware.
Today’s Agenda

- Semaphores
  - Building Mutices with Semaphores
- Monitors
- Condition Variables and the Lost Wakeup Problem
  - Building Condition Variables with Semaphores
  - Using Monitors with Condition Variables
  - Building Semaphores with Mutices
Semaphore Motivation

• Problem with lock: mutual exclusion, but no ordering; may want more

• E.g. Producer-consumer problem
  • $ cat 1.txt | sort | uniq | wc
  • **Producer**: creates a resource
  • **Consumer**: uses a resource
  • **Bounded Buffer** between them
  • Scheduling order: **producer waits if buffer full, consumer waits if buffer empty**
Semaphore Definition

- Invented by Edsger (sic) Dijkstra in 1968.
- A synchronization primitive that enables waiting without busy-wait.
- Contains an integer value
  - Can’t access directly
  - Must initialize to some value
    - `sem_init(sem_t *s, int pshared, unsigned int value)`
- Has two operations to manipulate this integer
  - `sem_wait()`, or `down()`, or `P()` (from Dutch)
  - `sem_post()`, or `up()`, or `V()` (from Dutch)
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

```c
int sem_wait(sem_t *s) {
    wait until:
        value of semaphore s is greater than 0
    decrement the value of semaphore s by 1
}
```
Semaphores

- **up()**
  - Atomically increment the counter.
  - (Allow a waiting thread to proceed)

```c
int sem_post(sem_t *s) {
    increment the value of semaphore s by 1
    if there are 1 or more threads waiting:
        wake 1
}
```
Semaphore Uses

- Mutual Exclusion: Semaphore as mutex
  - What should initial value be?
    - Binary semaphore: X=1
    - Counting semaphore: X>1

- Producer-Consumer Queue

- Scheduling Order
  - One thread waits for another
  - What should initial value be?

```c
// thread 0
// 1st half of computation
// ...
sem_post(s);
```

```c
// thread 1
sem_wait(s);
... // 2nd half of computation
```
Exercise: Semaphore-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!
Simplified Semaphore-Based “Blocking” Mutex

- Assumes we never try to unlock an unlocked mutex

```c
void lock(void) {
    M.down();
}

void unlock(void) {
    M.up();
}
```

M: private semaphore, initial count 1
Semaphore-Based “Blocking” Mutex

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

- 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 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();
}
```
Exercise: Mutex-Based Semaphore

(We need another component first: waking up waiting threads.)
Producer-Consumer (Bounded-Buffer) Problem

- Bounded buffer: size ‘N’
  - Access entry 0, ..., N-1, then “wrap around” to 0 again
- Producer process writes data to buffer
  - Must not write more than ‘N’ items more than consumer “ate”
- Consumer process reads data from buffer
  - Should not try to consume if there is no data
Producer-Consumer Solution v1

- Two semaphores
  - sem_t full; // # of filled slots
  - sem_t empty; // # of empty slots
- Problem: mutual exclusion?

```c
sem_init(&full, 0, 0);
sem_init(&empty, 0, N);
```

```c
producer() {
    sem_wait(empty);
    ... // fill a slot
    sem_post(full);
}
```

```c
consumer() {
    sem_wait(full);
    ... // empty a slot
    sem_post(empty);
}
```
Producer-Consumer Solution v2

- Two semaphores
  - `sem_t full;` // # of filled slots
  - `sem_t empty;` // # of empty slots
  - `sem_t mutex;

```c
sem_init(&full, 0, 0);
sem_init(&empty, 0, N);
sem_init(&mutex, 0, 1);
```

```c
producer() {
    sem_wait(empty);
    sem_wait(mutex);
    ... // fill a slot
    sem_post(mutex);
    sem_post(full);
}
```

```c
consumer() {
    sem_wait(full);
    sem_wait(mutex);
    ... // empty a slot
    sem_post(mutex);
    sem_post(empty);
}
```
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.
Monitors

• Background
  • Concurrent programming meets object-oriented programming
  • Simultaneous interest in parallel and object-oriented programming
  • People started to think about ways to make parallel programming more structured

• **Monitor**: object with a set of procedures for which only one thread may be active (i.e. running one of the monitor procedures) at a time
Schematic view of a Monitor

- Can think of a monitor as one big lock for a set of operations/methods
- In other words, a language implementation of mutices
Implementing a Monitor

Compiler automatically inserts lock and unlock operations upon entry and exit of monitor procedures

```java
class Account {
    int balance_;  
    public synchronized void deposit() {
        ++balance_;  
    }  
    public synchronized void withdraw() {
        --balance_;  
    }
}  
```

```java
m.lock();
++balance_;  
m.unlock();

m.lock();
--balance_;  
m.unlock();
```
Monitor + Producer/Consumer

- Mutex would be locked around semaphore waiting: deadlock
- Need something else.
  - Key concept: unlock mutex while waiting
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.
Condition Variables vs. Semaphores

- Semaphores are **sticky**: they have memory
  - `sem_post()` will increment the semaphore, even if no one has called `sem_wait()`

- Condition variables are not
  - if no one is waiting for a `signal()`, this `signal()` is not saved until there’s a `wait()`er
Using 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
    ...
}

enqueue(element) {
    pthread_mutex_lock(queue_lock)
    ...
    empty = false
    pthread_cond_signal(cond)
    ...
}
Fair Condition Variables Using Semaphores

Reported by Andrew Birrell in 1993¹

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()
}

¹https://birrell.org/andrew/papers/ImplementingCVs.pdf
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

1. `signal_one()` only when empty -> non-empty

2. Not covering condition check with mutex

```plaintext
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()
```
Producer-Consumer with Monitors

```c
monitor ProducerConsumer {
    int nfull = 0;
    cond not_full, not_empty;

    producer() {
        if (nfull == N)
            wait (not_full);
        ... // fill a slot
        ++nfull;
        signal (not_empty);
    }

    consumer() {
        if (nfull == 0)
            wait (not_empty);
        ... // empty a slot
        --nfull
        signal (not_full);
    }
};
```

- **(Explicit) mutex no longer needed!**
- **nfull**: number of filled buffers
  - Need to do our own counting for condition variables
- **not_full** and **not_empty**: two condition variables
  - notfull: not all slots are full
  - notempty: not all slots are empty
Condition Variable Semantics: Monitor Example

- Problem: when signal() wakes up a waiting thread, which thread to run inside the monitor, the signaling thread, or the waiting thread?

- **Hoare semantics**: suspends the signaling thread, and immediately transfers control to the woken thread
  - Difficult to implement in practice

- **Mesa semantics**: signal() moves a single waiting thread from the blocked state to a runnable state, then the signaling thread continues until it exits the monitor
  - Easy to implement
  - **Problem: race!** E.g. before a woken consumer continues, another consumer comes in and grabs the buffer
Exercise: Mutex-Based Semaphore

class Sema {
    pthread_mutex_t lock_;
    pthread_cond_t wait_;
    int value;

    Sema(const int count) {
        value_ = count;
        pthread_cond_init(&wait_, NULL);
        pthread_mutex_init(&lock_, NULL);
        return;
    }

    public:
    void down() { }
    void up() { }
};
Exercise: Mutex-Based Semaphore

class Sema {
    public:
        void down() {
            pthread_mutex_lock(&_lock);
            value_--;
            if (value_ < 0) {
                pthread_cond_wait(&wait_, &lock);
            }
            pthread_mutex_unlock(&lock_);
            return;
        }
        void up() {
            pthread_mutex_lock(&lock_);
            value_++;
            if (value_ <= 0) {
                pthread_cond_signal(&wait_);
            }
            pthread_mutex_unlock(&lock_);
            return;
        }
};
Conclusion

- Semaphores
  - Mutices with Semaphores
- Monitors
  - Producer-Consumer
- Condition Vars
  - Condition Vars with Semaphores
  - Semaphores with Mutices
Assignments

• Lab 1
  • Questions? Problems?

• Homework 2
  • Assigned today
  • Due next Tuesday, June 26th
  • Six midterm-style questions