

Condition Synchronization

Synchronization

- ◆ Now that you have seen locks, is that all there is?
- ◆ No, but what is the “right” way to build a parallel program.
 - People are still trying to figure that out.
- ◆ Compromises:
 - between making it easy to modify shared variables AND
 - restricting when you can modify shared variables.
 - between really flexible primitives AND
 - simple primitives that are easy to reason about.

Beyond Locks

- ◆ Synchronizing on a condition.
 - When you start working on a synchronization problem, first define the mutual exclusion constraints, then ask “when does a thread wait”, and create a separate synchronization variable representing each constraint.
- ◆ Bounded Buffer problem – producer puts things in a fixed sized buffer, consumer takes them out.
 - What are the constraints for bounded buffer?
 - 1) only one thread can manipulate buffer queue at a time (*mutual exclusion*)
 - 2) consumer must wait for producer to fill buffers if none full (*scheduling constraint*)
 - 3) producer must wait for consumer to empty buffers if all full (*scheduling constraint*)

Beyond Locks

- ◆ Locks ensure mutual exclusion
- ◆ Bounded Buffer problem – producer puts things in a fixed sized buffer, consumer takes them out.
 - Synchronizing on a condition.

```

Class BoundedBuffer{
...
void* buffer[];
Lock lock;
int count = 0;
}
                    
```

What is wrong with this?

```

BoundedBuffer::Deposit(c){
lock->acquire();
while (count == n); //spin
Add c to the buffer;
count++;
lock->release();
}
                    
```

```

BoundedBuffer::Remove(c){
lock->acquire();
while (count == 0); // spin
Remove c from buffer;
count--;
lock->release();
}
                    
```

Beyond Locks

```

Class BoundedBuffer{
...
void* buffer[];
Lock lock;
int count = 0;
}
                    
```

What is wrong with this?

```

BoundedBuffer::Deposit(c){
while (count == n); //spin
lock->acquire();
Add c to the buffer;
count++;
lock->release();
}
                    
```

```

BoundedBuffer::Remove(c){
while (count == 0); // spin
lock->acquire();
Remove c from buffer;
count--;
lock->release();
}
                    
```

Beyond Locks

```

Class BoundedBuffer{
...
void* buffer[];
Lock lock;
int count = 0;
}
                    
```

What is wrong with this?

```

BoundedBuffer::Deposit(c){
if (count == n) sleep();
lock->acquire();
Add c to the buffer;
count++;
lock->release();
if(count == 1) wakeup(remove);
}
                    
```

```

BoundedBuffer::Remove(c){
if (count == 0) sleep();
lock->acquire();
Remove c from buffer;
count--;
lock->release();
if(count==n-1) wakeup(deposit);
}
                    
```

Beyond Locks

```

Class BoundedBuffer{
...
void* buffer[];
Lock lock;
int count = 0;
}

```

What is wrong with this?

```

BoundedBuffer::Deposit(c){
lock->acquire();
if (count == n) sleep();
Add c to the buffer;
count++;
if(count == 1) wakeup(remove);
lock->release();
}

```

```

BoundedBuffer::Remove(c){
lock->acquire();
if (count == 0) sleep();
Remove c from buffer;
count--;
if(count==n-1) wakeup(deposit);
lock->release();
}

```

Beyond Locks

```

Class BoundedBuffer{
...
void* buffer[];
Lock lock;
int count = 0;
}

```

What is wrong with this?

```

BoundedBuffer::Deposit(c){
while(1) {
lock->acquire();
if(count == n) {
lock->release();
continue;}
Add c to the buffer;
count++;
lock->release();
break;
}
}

```

```

BoundedBuffer::Remove(c){
while(1) {
lock->acquire();
if (count == 0) {
lock->release();
continue;
}
Remove c from buffer;
count--;
lock->release();
break;
}
}

```

Introducing Condition Variables

- Correctness requirements for bounded buffer producer-consumer problem
 - Only one thread manipulates the buffer at any time (mutual exclusion)
 - Consumer must wait for producer when the buffer is empty (scheduling/synchronization constraint)
 - Producer must wait for the consumer when the buffer is full (scheduling/synchronization constraint)
- Solution: condition variables
 - An abstraction that supports conditional synchronization
 - Condition variables are associated with a monitor lock
 - Enable threads to wait inside a critical section by releasing the monitor lock.

Condition Variables: Operations

- Three operations
 - Wait()
 - Release lock
 - Go to sleep
 - Reacquire lock upon return
 - Java Condition interface await() and awaitUninterruptably()
 - Notify() (historically called Signal())
 - Wake up a waiter, if any
 - Condition interface signal()
 - NotifyAll() (historically called Broadcast())
 - Wake up all the waiters
 - Condition interface signalAll()
- Implementation
 - Requires a per-condition variable queue to be maintained
 - Threads waiting for the condition wait for a notify()

Wait() usually specified a lock to be released as a parameter

Implementing Wait() and Notify()

```

Condition::Notify(lock){
schedLock->acquire();
if (lock->numWaiting > 0) {
Move a TCB from waiting queue to ready queue;
lock->numWaiting--;
}
schedLock->release();
}

```

```

Condition::Wait(lock){
schedLock->acquire();
lock->numWaiting++;
lock->release();
Put TCB on the waiting queue for the CV;
schedLock->release();
switch();
lock->acquire();
}

```

Why do we need schedLock?

Using Condition Variables: An Example

- Coke machine as a shared buffer
- Two types of users
 - Producer: Restocks the coke machine
 - Consumer: Removes coke from the machine
- Requirements
 - Only a single person can access the machine at any time
 - If the machine is out of coke, wait until coke is restocked
 - If machine is full, wait for consumers to drink coke prior to restocking
- How will we implement this?
 - What is the class definition?
 - How many lock and condition variables do we need?

Coke Machine Example	
<pre> Class CokeMachine{ ... Storage for cokes (buffer) Lock lock; int count = 0; Condition notFull, notEmpty; } </pre>	<pre> CokeMachine::Deposit(){ lock->acquire(); while (count == n) { notFull.wait(&lock); } Add coke to the machine; count++; notEmpty.notify(); lock->release(); } CokeMachine::Remove(){ lock->acquire(); while (count == 0) { notEmpty.wait(&lock); } Remove coke from to the machine; count--; notEmpty.notify(); lock->release(); } </pre>

Word to the wise...	
<ul style="list-style-type: none"> ◆ Always wait and notify condition variables with the mutex held. ◆ Period. <ul style="list-style-type: none"> ➢ Fine print: There are cases where notification outside of a lock can be safe, but the code tends to be fragile, error-prone, and easy for another developer to break. ➢ In many cases you can lose notifications and hang (liveness) ➢ Moreover there is no clear advantage to breaking this convention. So just don't do it. 	14

Java syntax for condition variables	
<ul style="list-style-type: none"> ◆ Condition variables created from locks <pre> import java.util.concurrent.locks.ReentrantLock; public static final aLock = new ReentrantLock(); public static ok = aLock.newCondition(); public static int count; aLock.lock(); try { while(count < 16){ok.awaitUninterruptably()} } finally { aLock.unlock(); } return 0; </pre>	15

Summary	
<ul style="list-style-type: none"> ◆ Non-deterministic order of thread execution → concurrency problems <ul style="list-style-type: none"> ➢ Multiprocessing <ul style="list-style-type: none"> ◆ A system may contain multiple processors → cooperating threads/ processes can execute simultaneously ➢ Multi-programming <ul style="list-style-type: none"> ◆ Thread/process execution can be interleaved because of time-slicing ◆ Goal: Ensure that your concurrent program works under ALL possible interleaving ◆ Define synchronization constructs and programming style for developing concurrent programs <ul style="list-style-type: none"> ◆ Locks → provide mutual exclusion ◆ Condition variables → provide conditional synchronization 	16