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.

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

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;  
}

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();
}

What is wrong with this?
Class BoundedBuffer{
    ...
    void* buffer[];
    Lock lock;
    int count = 0;
}

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();
]
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
- \textbf{Wait()}
  - Release lock
  - Go to sleep
  - Reacquire lock upon return
  - Java Condition interface \texttt{await()} and \texttt{awaitUninterruptibly()}
- \textbf{Notify()} (historically called \texttt{Signal()})
  - Wake up a waiter, if any
  - Condition interface \texttt{signal()}
- \textbf{NotifyAll()} (historically called \texttt{Broadcast()})
  - Wake up all the waiters
  - Condition interface \texttt{signalAll()}

Implementation
- Requires a per-condition variable queue to be maintained
- Threads waiting for the condition wait for a notify()
Implementing `Wait()` and `Notify()`

```cpp
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?
Class CokeMachine{
    ...
    Lock lock;
    int count = 0;
    Condition notFull, notEmpty;
}

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--;
    notFull.notify();
    lock.release();
}

**Liveness issue**
Summary

- Non-deterministic order of thread execution → concurrency problems
  - Multiprocessing
    - A system may contain multiple processors → cooperating threads/processes can execute simultaneously
  - Multi-programming
    - 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
  - Locks → provide mutual exclusion
  - Condition variables → provide conditional synchronization