Programming with Threads

notify() vs. notifyAll()

- Very tricky to use notify() correctly
  - notifyAll() much safer
- Need:
  - All waiters are equal
  - Each notify only needs to wake up 1 thread
  - handle InterruptedException correctly

InterruptedException

- Threads t1 and t2 are waiting
- Thread t3 performs a notify
  - thread t1 is selected
- Before t1 can acquire lock, t1 is interrupted
- t1's call to wait throws InterruptedException
  - t1 doesn't process notification
  - t2 doesn't wake up

Handling InterruptedException

```
• synchronized (this) {
   while (!ready) {
           try {
                   wait(); }
           catch (InterruptedException e) {
                   notify();
                   throw e;
                   }
           // do whatever
           }
```

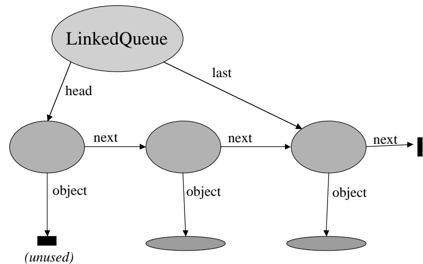
Lock/Monitor Granularity

- How many locks?
  - a single lock simplifies design
  - but creates a bottleneck
- How long do you hold them?
  - Holding a lock blocks others threads
  - Releasing and reacquiring lock
    - may introduce more overhead
    - give wrong semantics (transaction isn't atomic)

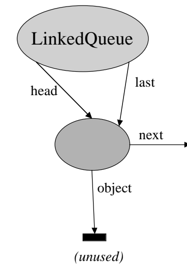
Concurrent Queue

- Need to prohibit concurrent enqueue's
- Need to prohibit concurrent dequeue's
- Can we allow a concurrent enqueue and dequeue?
  - Yes, most of the time

LinkedQueue [Lea 99]



When queue is empty



LinkedQueue

```
class LinkedQueue {
    protected Node head = new Node(null);
    protected Node last = head;

    static class Node {
      Object object;
      Node next = null;
      Node (Object x) { object = x; }
      }
```

Enqueue

```
synchronized void enqueue(Object x) {
   Node n = new Node(x);
   last.next = n;
   last = n;
   notifyAll();
   }
```

Try to Dequeue

```
synchronized Object tryToDequeue() {
   Object x = null;
   Node first = head.next;
   if (first != null) {
        x = first.object;
        first.object = null;
        head = first;
        }
   return x;
   }
```

Dequeue

```
synchronized Object dequeue()  throws
        InterruptedException {
   while (head.next == null) wait();
   Node first = head.next;
   Object x = first.object;
   first.object = null;
   head = first;
   return x;
   }
```

Concurrent enqueue and dequeue

- Allow for concurrent enqueue and dequeue
- Can't use same monitor
- If queue is full, working separate ends of list
- If queue is empty, working on same cell

Separate monitors

- Have separate utility monitor objects for enqueue and dequeue
- Also lock Node
  - If two threads need to exchange information, there needs to be a lock handoff on a common object
- Multiple locks complicates waiting
  - we'll skip waiting in this example

LinkedQueue

```
class LinkedQueue {
    protected final Object enqueueLock = new Object();
    protected final Object dequeueLock = new Object();
  /** Guarded by enqueueLock */
    protected Node head = new Node(null);
  /** Guarded by dequeueLock */
    protected Node last = head;

    static class Node {
      Object object;
      Node next = null;
      Node (Object x) { object = x; }
      }
```

Node locking

- Need to lock node n to access
  - n.next
  - n.next.object

Enqueue

```
void enqueue(Object x) {
  Node n = new Node(x);
  synchronized (enqueueLock) {
    synchronized(last) {
      last.next = n;
      last = n;
      }}}
```

Try to Dequeue

```
Object tryTodequeue() {
  synchronized (dequeueLock) {
    synchronized(head) {
      Object x = null;
      Node first = head.next;
      if (first != null) {
            x = first.object;
            first.object = null;
            head = first;
            }
      return x;
      }}}
```

Even more parallelism

- Say you had many concurrent enqueue and dequeue threads
- How to handle?
- Assume you are willing to give up strict first-in, first-out ordering…

Create multiple queues

- Each thread either picks a queue at random
- Or has a default queue, and goes to other queues if nothing available

When should you worry about blocking?

- On a single processor system, blocking is essentially never an issue,
  - so long as you don't hold any locks while you perform any operations that are
    - computationally expensive, or
    - potentially blocking (e.g., I/O)

Custom locks

- Can design custom locks
  - special features, such as trying for a lock or read locks
- Built using standard locks
- ReadWriteLock example
  - doesn't handle recursive locks

Read/Write locks

```
class ReadWriteLock {
   int readLocks = 0;
   boolean writeLocked = false;

   synchronized acquireReadLock()
              throws InterruptedException {
        while (writeLocked) wait();
        readLocks++;
        }
```

Read/Write locks

```
synchronized void releaseReadLock() {
        readLocks--;
        notifyAll();
        }
```

Read/Write locks

```
synchronized void acquireWriteLock()
        throws InterruptedException {
    while (writeLocked || readLocks > 0) wait();
    writeLocked = true;
    }
synchronized void releaseWriteLock() {
    writeLocked = false;
    notifyAll();
    }
```