Fachhochschule Nordwestschweiz

Assignment 3: Condition Synchronisation

- **Fair Semaphores**
- **Blocking Queue**

Semaphore: fair Locks do not help

```
public final class Semaphore {
   private volatile int value;
   private final ReentrantLock lock = new ReentrantLock(true);
   private final Condition cond = lock.newCondition();

   public Semaphore(int initial) {
      if (initial < 0) throw new IllegalArgumentException();
      value = initial;
   }

   public int available() {
      return value;
   }

   …
```

Semaphore: fair Locks do not help

```
  public void acquire() {
    lock.lock();
    try {
      while (value <= 0) {
        try { cond.await(); } catch (InterruptedException e) {}
      }
      value--;
    } finally { lock.unlock(); }
  }

  public void release() {
    lock.lock();
    try {
      value++;
      cond.signal();
    } finally { lock.unlock(); }
  }
}
```

- `new Semaphore(0);`
- T1 calls `acquire`, requests the lock, sees that value <= 0 and calls cond.await()
- T2 invokes `release` and gets the lock.
- T3 calls `acquire` and discovers that the lock is used. T3 queues up for the lock.
- T2 increments value and invokes cond.signal(). As a consequence T1 is awaken and tries to get the lock. It queues up behind T3!
- T2 releases the lock and T3 gets the lock as T3 acquired the lock before T1 reacquired it.

Semaphore1 with a Thread queue

```
public final class Semaphore {
   private int value;
   private final List<Thread> queue = new LinkedList<Thread>();

   public Semaphore(int initial) {
      if (initial < 0) throw new IllegalArgumentException();
      value = initial;
   }

   public synchronized int available() {
      return value;
   }

   …
```

Semaphore1 with a Thread queue

```
   public synchronized void acquire () { // uninterruptibly
      queue.add(Thread.currentThread());
      while (value <= 0 || queue.get(0) != Thread.currentThread()) {
         try {
            wait();
         } catch (InterruptedException e) {
         }
      }
      // value > 0 && queue.get(0) == Thread.currentThread()
      queue.remove(0); notifyAll(); // as queue changed
      value--;
   }

   public synchronized void release() {
      value++;
      notifyAll(); // all need to be woken up!
   }
}
```

Semaphore2 with a Condition queue

```
public final class Semaphore {
   private int value;
   private final Lock lock = new ReentrantLock();
   private final LinkedList<Condition> waitQueue = new LinkedList<>();

   public Semaphore(int initial) {
      if (initial < 0) throw new IllegalArgumentException();
      value = initial;
   }

   public int available() {
      lock.lock();
      try {
         return value;
      } finally { lock.unlock(); }
   }

   …
```

Semaphore2 with a Condition queue

```
public void release() {
    lock.lock();
    try {
        value++;
        if (!waitQueue.isEmpty()) waitQueue.getFirst().signal();
    } finally {
        lock.unlock();
    }
}

…
```

Semaphore2 with a Condition queue

```
   public void acquire() { // uninterruptibly
      lock.lock();
      try {
         Condition curr = lock.newCondition();
         waitQueue.add(curr);
         while (available() <= 0 || waitQueue.getFirst() != curr ) {
            try { curr.await(); } catch (InterruptedException e) { }
         }
         value--;
         waitQueue.removeFirst();
         if (!waitQueue.isEmpty()) waitQueue.getFirst().signal();
      } finally { lock.unlock(); }
   }

}
```

Assignment 3: Condition Synchronisation

- **Fair Semaphores**
- **Blocking Queue**

Queue

```
public Object dequeue() {
   used.acquire();
   Object result;
   synchronized(this) {
      result = buf[head];
      head = (head + 1) % SIZE;
   }
   free.release();
   return result;
}
```

```
public void enqueue(Object x) {
   free.acquire();

   synchronized(this) {
      buf[tail] = x;
      tail = (tail + 1) % SIZE;
   }
   used.release();

}
```

- Dequeue and Enqueue operations do not need to exclude each other
  - Dequeue: changes `head` field only
  - Enqueue: changes `tail` field only
  - No *size* field which is adjusted by enqueue and dequeue methods

Queue

```
private Object deq =
                new Object();

public Object dequeue() {
   used.acquire();
   Object result;
   synchronized(deq) {
      result = buf[head];
      head = (head + 1) % SIZE;
   }
   free.release();
   return result;
}
```

```
private Object enc =
                    new Object();

public void enqueue(Object x) {
   free.acquire();

   synchronized(enc) {
      buf[tail] = x;
      tail = (tail + 1) % SIZE;
   }
   used.release();

}
```

Queue

```
private Object deq =
                  new Object();

public Object dequeue() {
   used.acquire();
   Object result;
   synchronized(deq) {
      result = buf[head];
      free.release();
      head = (head + 1) % SIZE;
   }
   return result;
}
```

```
private Object enc =
                    new Object();

public void enqueue(Object x) {
   free.acquire();

   synchronized(enc) {
      buf[tail] = x;
      used.release();
      tail = (tail + 1) % SIZE;
   }

}
```

- Release operations on semaphores may already be executed within synchronized block