PA2 Tutorial

CS 131a

Java's "`synchronized`" keyword

- Locks on the instantiated object
- Mutual exclusion with other "`synchronized`" methods

Busy Waiting

- One way of acquiring a lock
- Protocol:
  - Try to acquire the lock
    - Got the lock: great! use it.
    - Didn’t get the lock: try again
- Advantage: simple implementation

Busy Waiting

```
while (true) {
   boolean result = tryToAcquireLock();
   if (result)
      break; // we got the lock!
   continue; // we didn't get the lock… try
             // again!
```

Busy Waiting

- Advantages: simple implementation
- Disadvantages: costly, “busy” waiting
  - CPU usage often 100%
- In practice, rarely used
  - … so why do you have to learn it?

    we’ll come back to that...

Monitors

- Another way to acquire a lock
- Protocol:
  - Try to acquire a lock
    - Got the lock: great! use it.
    - Didn’t get the lock:
      - Add myself to the monitor’s wait queue
      - Sleep / wait
      - When the lock becomes available, I’ll be woken up
- More complex implementation...

Monitors

- Three tools that you learned in lecture:
  - **`Object#wait()`**: **sleep** the current thread **until** `Object`'s monitor is **notified**
    - Adds the current thread to the waiting set
  - **`Object#notify()`**: **notify** the **next thread** in `Object`'s waiting set
  - **`Object#notifyAll()`**: **notify ALL threads** in `Object`'s waiting set.

Monitors

(not the same “consumer” as in “producer/consumer”)

Consumer

```
while (true) {
    synchronized (this) {
        result = tryToAcq();
        if (result)
            break;
        // didn't get lock
        addToWaitQ(this);
        this.wait();
    }
}
```

Resource

```
public void addToWaitQ(obj){
    // put obj into a queue
    q.add(obj);
}
public void resourceAvail(){
    // get next waiting obj
    Object o = q.pop();
    synchronized (o) {
        o.notify();
    }
}
```

Monitors

- Advantages: doesn’t hog the CPU
- Disadvantages: complexity, overhead

- So, why ever use busy-waiting?
  - When the *expected wait cost* for a resource is less than the overhead cost of monitors.

Reminders

- Use nothing from `java.util.concurrent`
  - This includes `BlockingQueue`
- Do not interact with the `AntLog`
  - It's for evaluation purposes only
  - Incorrect code sometimes passes, correct code always passes the included test cases
- Always strive for "correct" code over "clever" code
  - Almost anything involving reflection is "clever"

PA1 FeedBack

Most common mistakes:

- `Join()`ing immediately after `start()`ing.
- Not using `BlockingQueue` per pair of subcommands.

`join()` immediately after `start()`


```
T1.start();
T1.join();
            T2.start();
            T2.join();
                        T3.start();
                        T3.join();
```

`join()` immediately after `start()`

In this way, T1, T2, T3 are not running concurrently.

T2 has to wait T1 to finish to start;

same as T3 to wait T2 to finish to start.

Monitors

(not the same “consumer” as in “producer/consumer”)

Consumer

```
while (true) {
    synchronized (res) {
        result = res.acq();
        if (result)
            break;
        // didn't get lock
        res.wait();
    }
}
res.release();
```

Resource

```
public sync void release() {
    locked = false;
    synchronized (this) {
        this.notify();
    }
}
public sync boolean acq(){
    if (locked)
        return false;
    locked = true;
    return true;
}
```

Monitors

(not the same “consumer” as in “producer/consumer”)

Consumer

```
get() {
    synchronized (q) {
        while (q.isEmpty()) {
            q.wait();
        }
        o = q.dequeue();
        q.notifyAll();
    }
    return o;
}
```

Resource

```
Queue q;
```

Producer

```
put(o) {
    synchronized (q) {
        while (q.isFull()) {
            q.wait();
        }
        q.enqueue(o);
        q.notifyAll();
    }
}
```