Concurrent Programming Practice

COMP755 Advanced OS

Dr. Ken Williams

"Parallel machines are hard to program and we should make them even harder - to keep the riff-raff off them."

Gary Montry
Massively Parallel Computing Research Laboratory
Sandia National Laboratories

Exam

- The first exam in COMP755 will be this Wednesday, September 18, 2013
- You are allowed one 8 ½ x 11" page on notes
- The exam will cover everything since the beginning of class
- There will be a concurrent program to write

Why Bother?

- Most new computers have multiple cores. They can execute multiple threads or programs simultaneously. This is of little value if your program has only one thread.
- If you have N threads, your program might run almost N times faster.

Programming Ease

- When your program is doing more than one unsynchronized activity, it can be much easier to write with multiple threads.
- If your program needs to call more than one blocking function, threads are needed.

Concurrency Patterns

- **Mutual Exclusion** – Used to ensure that only one task executes a particular segment of code. Can be used for resource allocation.
- **Synchronization** – Coordinates the action of different tasks.
- **Barrier Synchronization** – Waits until all tasks are complete.

Frequent Solutions

- Producer / Consumer is a common part of a concurrent program
- Resource allocation, similar to the dining philosophers
- Reader / Writers

Keyboard and Network Input

- Consider a program with input from two sources, user keyboard input and network input. Whenever a number is received from either source they need to call crunchNumber
- Assume crunchNumber take a lengthy time to execute
- Assume the **`read`** statement for both keyboard and network is blocking

Possible Solution

- One solution is to use three threads
  - Reads from the network and puts the number on a queue
  - Reads from the keyboard and puts the number on a queue
  - Takes numbers from the queue and calls crunchNumber
- This is an example of a producer / consumer problem

Possible Semaphore Solution

Semaphore mutex = new Semaphore(1);

Start network and keyboard threads

```
network {
do forever {
    read from net;
    mutex.p();
    put on queue;
    mutex.v();
}
}
```

```
keyboard{
do forever {
        read from keyboard;
        mutex.p();
        put on queue;
        mutex.v();
}
}
```

Possible Synchronized Solution

- Start two threads

```
Network {
   do forever {
         read from net;
         put(num);
   }
}
```

```
Keyboard{
   do forever {
         read from keyboard;
         put(num);
   }
}
```

Crunch Thread

```
while ( true ) {
        int num = get();
        crunchNumber( num );
}
```

Java Producer Consumer

```
public class prodcon {
  queue q;
   public synchronized void put (int x) {
     q.add(x);
     notify();
   }
   public synchronized int get( ) {
     while (q.empty()) wait();
     int y = q.remove();
     return y;
   }
}
```

Quality Producer/Consumer

- Producer threads create objects and call **`put`** to put them on a queue.
- The put function evaluates the objects quality *(unknown to the producer)* and puts the object on one of two queues, good or bad.
- Consumers call the methods **`getGood`** and **`getBad`**.

```
public class Quality {
   Queue goodQ, badQ;
   public synchronized void put(Object thing) {
       if (thing.isGood()) {
             goodQ.add(thing);
       } else {
             badQ.add(thing);
       }
       notify();
   }

   public synchronized Object getGood() {
       while (goodQ.empty()) wait();
       return goodQ.remove();
   }

   public synchronized Object getBad() {
       while (badQ.empty()) wait();
       return badQ.remove();
   }
}
```

/* This does not work */

```
public class Quality {          /* This should work */
   MyList goodList, badList;
   public void put(Object thing) {
       if (thing.isGood()) {
              goodList.add(thing);
       } else {
              badList.add(thing);
       }
   }

   public Object getGood() {
       return goodList.get();
   }

   public Object getBad() {
       return badList.get();
   }
}
```

```
public class MyList {
   Queue q;
   public synchronized void put(Object thing) {
       q.add(thing);
       notify();
   }

   public synchronized Object get() {
       while (q.empty()) wait();
       return q.remove();
   }
}
```

Implementing Synchronized Methods

- Using semaphores, create the infrastructure for synchronized methods
- Write the methods
  - entry
  - exit
  - wait
  - notify
  - notifyAll