Concurrent Programming Practice

COMP755 Advanced OS

Dr. Ken Williams

Why Bother?

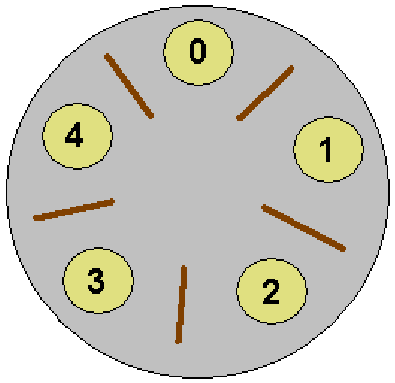
- Most new computers have multiple cores. They can execute multiple threads or programs simultaneously. This is of little value if your program has only one thread.
- If you have N threads, your program might run almost N times faster.

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.

Example: Dining Philosophers

There are 5 philosophers who sit around a table. Between each philosopher is a chopstick.



Dining Philosophers Problem

- The philosophers spend their time thinking and eating on independent schedules. To eat, each philosopher needs both the chopstick to her right and the chopstick to her left.
- Note that adjacent philosophers cannot eat at the same time because they both need the same chopstick.

Dining Philosophers Solution

- Each philosopher is a separate thread
- This problem has 5 resources (the chopsticks) that require exclusive use.
- A semaphore can be assigned to each chopstick to provide exclusive access.

Dining Philosophers Solution

Each chopstick resource is protected by a semaphore, `chopstick`, initialized to 1.

```
philosopher i
do forever {
  think;
  p(chopstick(i));          // right
  p(chopstick((i+1)%5));    // left
  eat;
  v(chopstick(i));
  v(chopstick((i+1)%5));
}
```

Potential Problem

- If all five philosophers get hungry at the same time and each grabs the chopstick to their right, none of them will have another chopstick to use and they will all starve.
- A possible solution is to have the odd numbered philosophers grab the right and then the left while even numbered philosophers grab the chopsticks in the reverse order.

Better Solution

```
philosopher i
do forever {
  think;
  if( i is odd) {
     p(chopstick(i));          // right
     p(chopstick((i+1)%5));  // left
  } else {
     p(chopstick((i+1)%5));  // left
     p(chopstick(i));          // right
  }
  eat;
  v(chopstick(i));
  v(chopstick((i+1)%5));
}
```

Semaphore Use

- There are only two functions that act upon a semaphore, P() and V() *[or signal and wait]*
- You cannot inspect the counter of the semaphore or if there are threads queued upon it.
- There must always be a V call for every P call

Class Participation

- Divide into groups of about 4 students
- Write solutions to the problems.

Synchronized Dining Philosophers

- Write a solution to the dining philosophers problem using Java synchronized methods instead of semaphores

Synchronized Dining Philosophers

```
public class Chopstick{
    boolean inUse = false;
    synchronized void pickUp() {
        while (inUse) wait();
        inUse = true;
    }
    synchronized void putDown() {
        inUse = false;
        notify();
    }
}
```

```
stick = array of 5 Chopstick objects
philosopher i
do forever {
        think;
        stick[i].pickUp();
        stick[(i+1)%5].pickUp();
        eat;
        stick[i].putDown();
        stick[(i+1)%5] .putDown();
}
```

GUI and Network Updates

- Consider a program with input from two sources, user keyboard input and network input. Whenever input is received from either source, a common list is updated.
- Assume the **`read`** statement for both keyboard and network is blocking.

Possible Semaphore Solution

Semaphore mutex = 1;

Start network and keyboard threads

```
network {
do forever {
    read from net;
    p(mutex);
    update list;
    v(mutex);
}
}
```

```
keyboard{
do forever {
        read from keyboard;
        p(mutex);
        update list;
        v(mutex);
}
}
```

Possible Synchronized Solution

- Start two threads

```
synchronized void update(Widget thing) {
    update list;
}
```

```
Network {
  do forever {
        read from net;
        update(thing);
  }
}
```

```
Keyboard{
  do forever {
        read from keyboard;
        update(thing);
  }
}
```