Writing Programs with Multiple Threads

COMP755

Advanced OS

Graduate Colloquium

- Tuesday, Sept. 10th from 12:15 to 1:00 p.m.
- Graham 210
- All Computer Science graduate students are requested to attend and be on time

Thread Programs

- The four thread programs are due by midnight on Friday, September 13
- Questions 1, 3 and 4 are logically similar
- Question 2 is a Reader / Writers problem plus an object update

A Good Program *(and one that gets a good grade)*

- uses multiple processes or threads
- produces the correct result
- will not deadlock under any situation
- does not use busy waiting
- has maximum concurrency
- utilizes an efficient algorithm
- To aid in understanding the operation of the algorithm, you may wish to print useful information whenever an important event occurs

Threads in Java

- Threads are built into Java as part of the standard class library.
- There are two ways to create threads in Java:
  - Extend the class **`Thread`**
  - Implement the interface **`Runnable`**

run Method

- Both means of creating threads in Java require the programmer to implement the method:

```
public void run() { … }
```

- When a new thread is created, the **run** method is executed in parallel with the calling thread.

Extending Java Thread Class

```
class Example extends Thread {
  int classData;   // example data
  // optional constructor
  Example(int something) {
     classData= something;
  }

  public void run() {
  // runs in parallel
  . . .
  }
}
```

Starting a Thread Object

- Parallel execution of the run method of a Thread object is initiated by:

```
// create Thread Object
Example xyz = new Example(143);
// start execution of run method
xyz.start();
```

Runnable Interface

- Java does not support multiple inheritance. If a class extends Thread, it cannot extend another class
- Programmers frequently want to use multiple threads and extend another class, such as Applet
- The Runnable interface allows a program use multiple threads in a single object

Implementing Runnable Interface

```
class RExample implements Runnable {
  int classData;   // example data
  // optional constructor
  RExample(int something) {
     classData= something;
  }

  public void run() {
  // runs in parallel
  . . .
  }
}
```

Starting a Runnable Object

- Parallel execution of the run method of a Runnable object is initiated by:

```
// create Thread Object
RExample xyz = new RExample(143);
// start execution of run method
new Thread(xyz).start();
```

Java Thread Termination

- Similar to pthreads, Java threads terminate when the run method returns
- The System.exit(int) method will terminate the entire process

Only One Run

- An annoying feature about Java threads is that they always start a method named run. Other thread systems allow you to specify the method to be executed.
- The run method has no parameters. Data must be passed to the method through common variables.
- A thread can only be started once.

```
public class MultiRun extends Thread {
    int counter = 0;
    public void run() {
        System.out.print(counter);
        counter++;
    }
    public static void main(String[] args) {
        MultiRun yarn = new MultiRun();
        yarn.start();
        yarn.start(); // This causes an error
    }
}
```

```
public class MultiRun extends Thread {
    static int counter = 0;
    public void run() {
        System.out.print(counter);
        counter++;
    }
    public static void main(String[] args) {
        MultiRun yarn = new MultiRun();
        yarn.start();
        MultiRun yarn2 = new MultiRun();
        yarn2.start();
    }
}
```

What Does This Display?

```
public class MultiRun extends Thread {
    static int counter = 0;
    public void run() {
        System.out.print(counter);
    }
    public static void main(String[] args) {
        MultiRun yarn = new MultiRun();
        yarn.start();
        counter++;
        MultiRun yarn2 = new MultiRun();
        yarn2.start();
    }
}
```

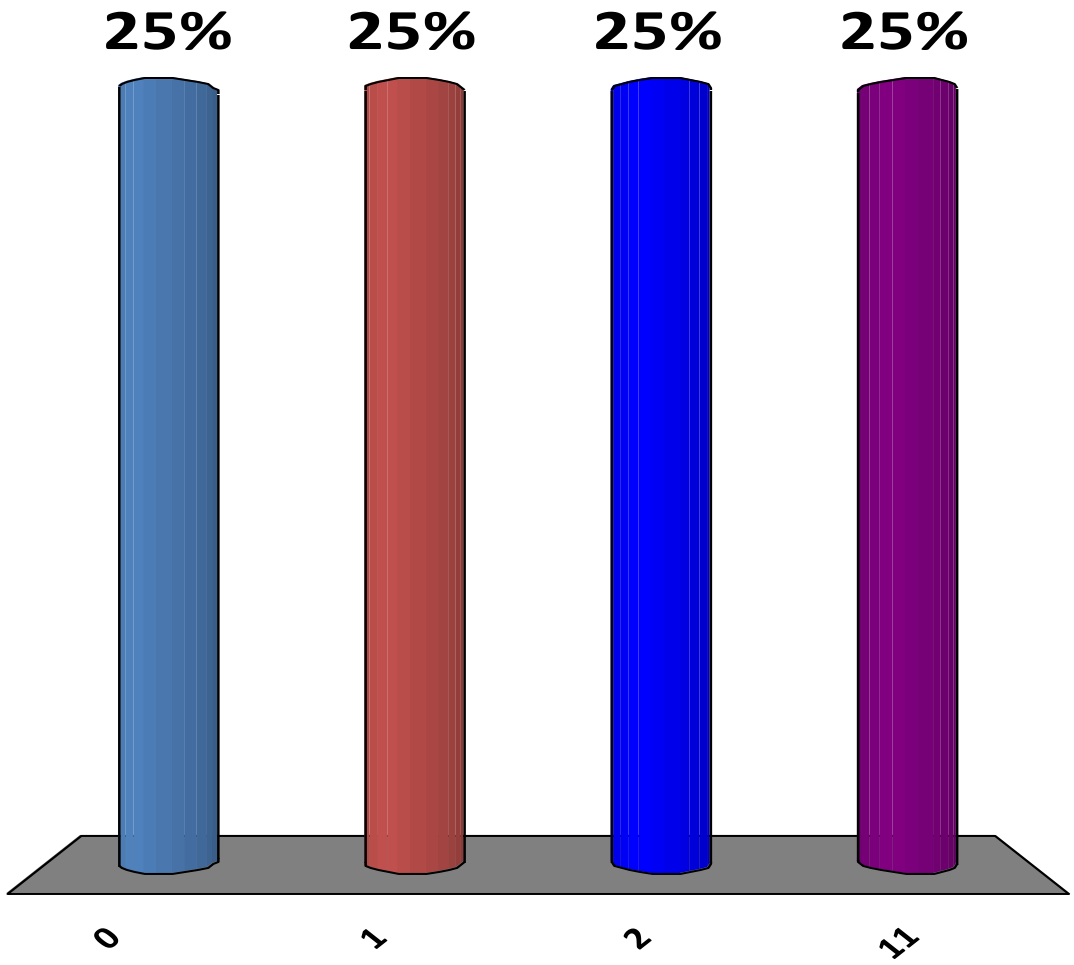
What Does MultiRun Display?

A. 00

B. 01

C. 02

D. 11



Thread Parameters

- Unlike many other thread systems, Java does not provide a parameter to the run method
- Each thread has to determine what to do
- An easy solution is a synchronized initialization method

```
private int threadNum = 0;
int synchronized getNum() {
    return threadNum++;
}
```

Example Thread Class

```
class Example extends Thread {
  private int multUse = 0;
  synchronized void doIt() {
     multUse++;
  }
  public void run() {
     …
     doIt();
     …
  }
}
```

Misusing Example

```
Example abc = new Example();
Example xyz = new Example();
abc.start();
xyz.start();
```

- Is multUse protected against concurrent use?

Classes and Objects

- A synchronized method provides mutual exclusion within an object
- If the synchronized methods are in different objects, they will not be synchronized

volatile Variables

- volatile is a Java modifier keyword

```
volatile int rabbit;
```

- The Java Memory Model ensures that all threads see a consistent value for a volatile variable
- Volatile variables are always accessed from RAM
- Shared variables are best marked volatile

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.

Mutual Exclusion with semaphores

```
semaphore s = new semaphore(1);


s.p();
  // Critical section;
s.v();
```

Mutual Exclusion with synchronized methods

```
public class Example {
  public synchronized object getThing() {
     Critical section; // nothing to it
  }
}
```

Synchronization

- Multiple threads often need to interact. When one thread has completed something, it may need to inform another thread.
- In the producer/consumer problem, the producers need to be informed when the consumers have made a new item.

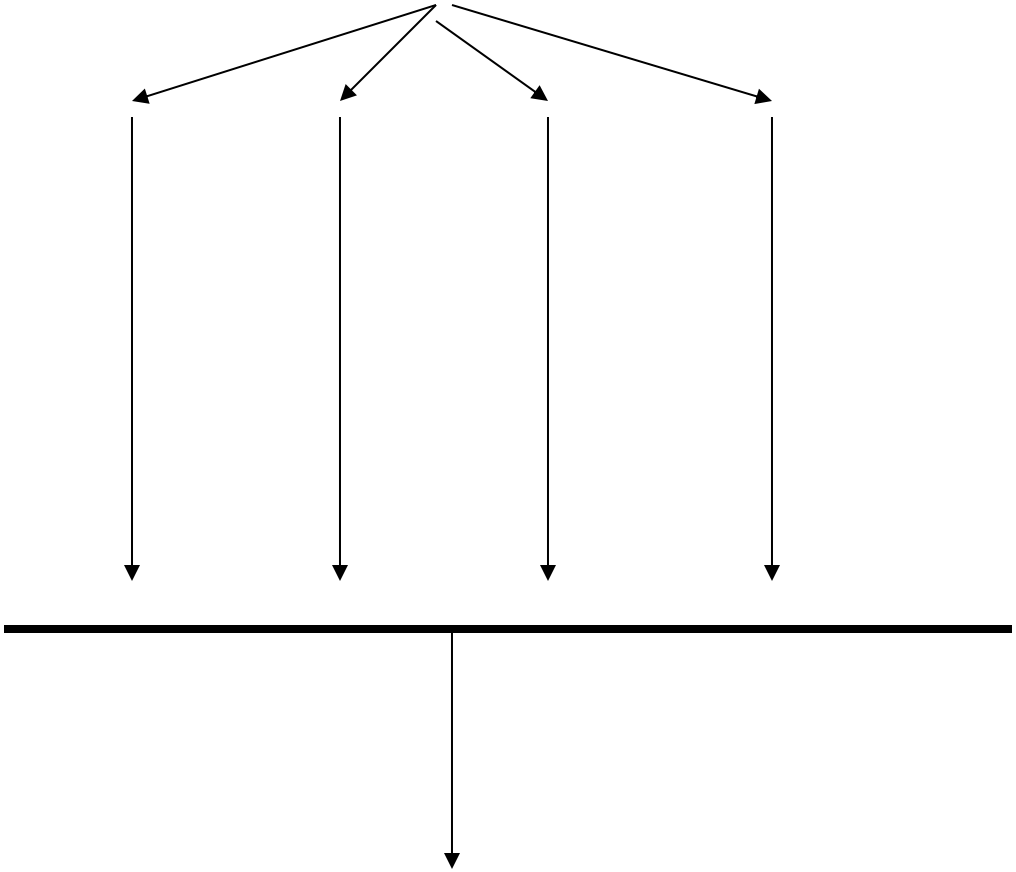
Synchronization with semaphores

```
semaphore numReady =
                    new semaphore(0);
void makeIt() {
  numReady.v();
}
void useIt() {
  numReady.p();
}
```

Synchronization with Synchronized Methods

```
public class numReady {
int available = 0;
public synchronized void makeIt(){
  available++;
  notify();
}
public synchronized void useIt() {
  while (available == 0) wait();
  available--;
}
```

Barrier Synchronization



One task continues after all tasks complete.

Barrier Synchronization with Semaphores

```
semaphore mutex = 1;
int numRunning = N;
// start threads and run the parallel stuff here
mutex.p();
numRunning--;
if (numRunning != 1) {
    mutex.v();
    exit();  // all but last task terminates
}
mutex.v();
```

Barrier Synchronization with Synchronized Methods

```
int numRunning = N;
public void run() {
  // run the parallel stuff here
  synchronized (this) {
     numRunning—;
     if(numRunning==1)morestuff();
  }
}
```

Java Barrier Synchronization with Join

```
SubThread[] t = SubThread[N];

// Start the parallel threads
for (i = 0; i < N; i++) {
  t[i] = new SubThread();
  t.start();
}

// Wait for thread to terminate
for (i = 0; i < N; i++)
     t[i].join();
```

Phaser class

```
void runTasks(List<Runnable> tasks) {
  final Phaser phaser = new Phaser(1); // "1" to register self
  // create and start threads
  for (final Runnable task : tasks) {
    phaser.register();
    task.start();
  }
  // allow threads to start and deregister self
  phaser.arriveAndDeregister();
 // all other threads have terminated by now
}

public class Task Thread() {
      public void run() {
         // do something useful here
        phaser.arrive();                    // signal thread is done
      }
    }
```

Semaphore Use

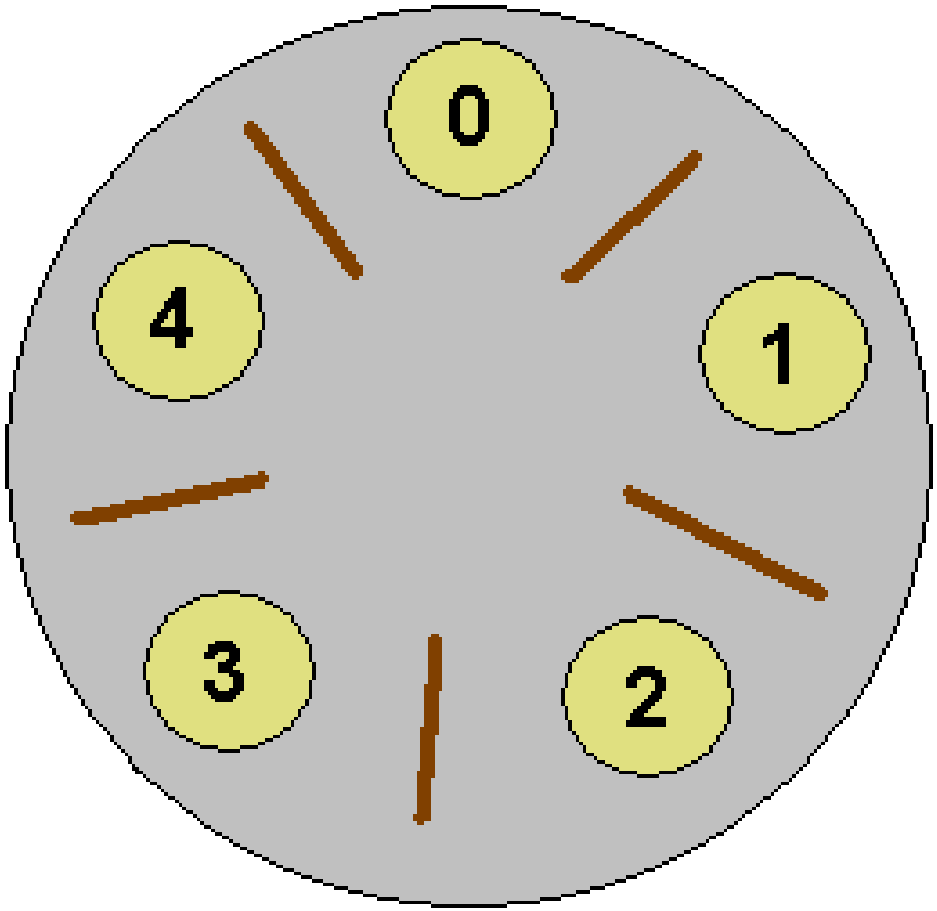
- If you have shared data, you will need a semaphore to provide mutual exclusion
- Do a P() before you access the shared data and a V() afterwards
- If a thread is going to have to wait for something, it will have to do a P() on a semaphore
- There has to be one V() for every P()

Classical Thread Problems

- Dining Philosophers
- Reader / Writers
- Sleeping Barber
- Smokers

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.

Poor Solution

- Use one semaphore to lock the table
- This provides a "correct" solution, but does not provide for optimal parallelism

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;
  chopstick(i).p();                 // right
  chopstick((i+1)%5).p();           // left
  eat;
  chopstick(i).v();
  chopstick((i+1)%5).v();
}
```

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) {
     chopstick(i).p();          // right
     chopstick((i+1)%5).p();    // left
  } else {
     chopstick((i+1)%5).p();    // left
     chopstick(i).p();          // right
  }
  eat;
  chopstick(i).v();
  chopstick((i+1)%5).v();
}
```

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

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();
}
```

Graduate Colloquium

- Tuesday, Sept. 10th from 12:15 to 1:00 p.m.
- Graham 210
- All Computer Science graduate students are requested to attend and be on time

Thread Programs

- The four thread programs are due by midnight on Friday, September 13
- Questions 1, 3 and 4 are logically similar
- Question 2 is a Reader / Writers problem plus an object update