Writing Parallel Programs

COMP360

"We stand at the threshold of a many core world. The hardware community is ready to cross this threshold. The parallel software community is not."

Tim Mattson

principal engineer at Intel

Remaining Schedule

	Wednesday, April 26 Concurrent Programming	Friday, April 28 Concurrent Programming
Monday, May 1 **Exam 3**	Wednesday, May 3 final review	
Tuesday, May 9 **Final Exam** 10:30am – 12:30pm		

X++ in Machine Language

Thread 1		Thread 2
LOAD	R1,X	
ADD	R1,1	
	OS switches to another thread	
		LOAD R4,X
		ADD R4,1
		STORE R4,X
	OS switches to another thread	
STORE	R1,X	

Mutual Exclusion

- To avoid problems when sharing data between threads, each thread has to have exclusive access to the data
- A segment of code that can only be executed by one thread at a time is called a ***critical section***
- If a second thread attempts to execute the critical section while another thread is already executing there, the second thread will be suspended until the first thread exits the critical section

Mutual Exclusion Format

- The general format for creating a mutually exclusive critical section is:

…

```
Critical Section Entry code
// Critical Section
Critical Section Entry code
```

…

- Some languages do this automatically

Parallel Programming Environments

- Shared memory
  - All threads or processes can access all the memory
  - Any thread can run on any CPU
  - Threads can communicate by changing variables
- Separate memory
  - Processes run on separate CPUs with separate memory
  - Threads communication by sending messages

Some Parallel Programming Languages

- Java
- Ada
- APIs providing parallel programming
  - pThreads – C++ functions
  - Message Passing Interface – runs on both shared and separate memory systems
  - OpenMP – Extension to C++ or Fortran
  - CUDA – Allows C++ programs to access GPU
  - Hadoop – Big data Map Reduce

Have you written a Java program with multiple threads?

A. Yes

B. No

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

```
public class Example extends Thread {
  int classData;    // example data

  public Example(int something)  {  // optional constructor
      classData= something;
  }

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

Starting a Thread

- The start() method executes the run() method
- The run method starts executing as a separate thread
- Control is immediately returned to the caller of start

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 JFrame
- The Runnable interface allows a program use multiple threads and inheritance

Implementing Runnable Interface

```
public class RExample implements Runnable {
 int classData;    // example data
 // optional constructor
  public 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 many threading systems, 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

Java Thread Errors

```
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
     }
}
```

Fixing the Java Thread Errors

```
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();
    }
}
```

Semaphores

- Semaphores were originally described by E. Dykstra
- A semaphore has an integer counter and a queue of threads suspended on the semaphore
- Semaphores can only be modified by two functions **P** *(or wait)* and **V** *(or signal)*
- Supported by POSIX Threads (pthreads) and Java

Problems with semaphores

- They are pretty low-level
  - When using them for mutual exclusion, for example (the most common usage), it's easy to forget a P or a V, especially when they don't occur in strictly matched pairs
- Their use is scattered all over the place
  - If you want to change how processes synchronize access to a data structure, you have to find all the places in the code where they touch that structure, which is difficult and error-prone

Monitors

- Monitors were an attempt to address these two weaknesses of semaphores
- They were suggested by Edsger W. Dijkstra, developed more thoroughly by Brinch Hansen, and formalized nicely by Tony Hoare *(a real cooperative effort!)* in the early 1970s
- Several parallel programming languages have incorporated monitors as their fundamental synchronization mechanism
  - none incorporate the precise semantics of Hoare's formalization

Monitor Mechanisms

- A monitor has private internal data values and public methods. Only one method of a given monitor may be active at a given point in time
- A process that calls a busy monitor is delayed until the monitor is free
  - On behalf of its calling process, any operation may suspend itself by waiting on a condition
  - An operation may also signal a condition, in which case one of the waiting processes is resumed, usually the one that waited first

Synchronized Methods

- Java and C# support synchronized methods

public synchronized int myfunc(char x){ … }

- Only one thread at a time can execute in any synchronized method of an object
- If another thread calls that or any other synchronized method in the object, its execution will be suspended until the first thread leaves the method

Synchronized Method Example

```
public class COMP755 {
 private int numThings = 0;
 public synchronized void add(int x) {
    numThings += x;
 }
 public synchronized void sub(int x) {
    numThings -= x;
 }
}
```

`wait()` Method

- **`wait()`** is a method of Object and thus a method of every object
- When a thread executes the wait method, it is suspended until another thread executes the notify or notifyAll method on this object
- **`wait()`** must be executed inside a synchronized method or block

`wait( )` Method

- When a thread calls **`wait()`** inside a synchronized method, the method then becomes available to other threads
- If you call **`wait()`** without specifying an object, it refers to **`this`** object
- Calls to **`wait()`** should be in a while statement that checks on the condition allowing the thread to proceed

`notify()` Method

- **`notify()`** will activate **`one`** thread that has been suspended by executing the **`wait()`** function on the same object
- If no thread is suspended on that object, calling **`notify()`** has no effect. Calling **`notify()`** before the other thread calls **`wait()`** will not unlock the waiting thread

`notifyAll()` method

- **`notifyAll()`** will activate **all** threads that have been suspended by executing the **`wait()`** function on the same object
- If no thread is suspended on that object, calling **`notifyAll()`** has no effect

Producer Consumer Problem

- Assume there are many producer threads and consumer threads
- Producer threads make stuff and give them to any available consumer thread
- Consumer threads must wait if no stuff is available

- This is the basis for most concurrent programming solutions

Producer Consumer with Semaphores

Semaphore mutex = 1, qsize = 0;

```
void producer (Thing x) {
   p(mutex);
   put thing on queue
   v(mutex);
   v(qsize);
}
```

```
Thing consumer () {
   p(qsize);
   p(mutex);
   get thing from queue
   v(mutex);
   return thing;
}
```

Java Synchronized Method

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

Synchronized Blocks

- In addition to synchronized methods, you can restrict a block of code to only one thread

```
synchronized (object) { … }
```

- Only one thread can execute inside the block

Write a Parallel Program

- Assume we have a bag of Widgets
- Many threads need to acquire a Widget and later release the Widget
- There are N Widgets in the pool
- If a thread requests a Widget and none are available, it should wait

Possible Solution

```
public class WidgetPool {
        Bag pool = contains N widgets;
        Widget get() {
                while (pool.empty()) wait();
                return pool.get();
        }
        void release(Widget thing) {
                pool.put( thing );
                notify();
        }
}
```

Locking Objects

- A synchronized method or block locks the object involved. The object is unlocked at the end of the synchronized method or block
- Locking an object does not prevent other threads from accessing non-private fields of the object
- Locking is only done through synchronized methods or blocks
- Java does not prevent or detect deadlock

Multiple locks

- A single thread can lock the same object multiple times
- A lock on an object already locked by this thread has no effect

```
public static void main(String[] args) {
        Test t = new Test();
        synchronized(t) {
                synchronized(t) {
                        System.out.println("made it!");
                }
        }
}
```

Can you have recursive synchronized methods?

A. Yes

B. No

Multiple Threads and GUIs

- The GUI thread waits most of the time waiting for the user to interact with the interface
- If the GUI thread is busy executing elsewhere, the GUI will not respond to user input
- It is wise to use another thread for long or blocking calculations
- SwingUtilities.invokeLater(new Runnable() {public void run() { new MainFrame("SwingWorker Demo"); } });

Updating GUI from Another Thread

- The invokeLater method will safely update a GUI field avoiding multi-threading issues

```
javax.swing.SwingUtilities.invokeLater(new Runnable() {
   public void run() {
      labelObj.setText("Background results");
   }
} );
```

Remaining Schedule

	Wednesday, April 26 Concurrent Programming	Friday, April 28 Concurrent Programming
Monday, May 1 **Exam 3**	Wednesday, May 3 final review	
Tuesday, May 9 **Final Exam** 10:30am – 12:30pm		