Concurrent Programming

Prof. Sanjeev Setia
Distributed Software Systems
CS 707
Spring 2000

Hardware Architectures

- Uniprocessors
- Shared-memory multiprocessors
- Distributed-memory multicomputers
- Distributed systems

Concurrent Programming

- Process = Address space + one thread of control
- Concurrent program = **multiple threads of control**
  - Multiple single-threaded processes
  - Multi-threaded process

Application classes

- Multi-threaded Programs
  - Processes/Threads on same computer
  - Window systems, Operating systems
- Distributed computing
  - Processes/Threads on separate computers
  - File servers, Web servers
- Parallel computing
  - On same (multiprocessor) or different computers
  - Goal: solve a problem faster or solve a bigger problem in the same time

Concurrent Systems

- Essential aspects of any concurrent system
  - Execution context - state of a concurrent entity
  - Scheduling - deciding which context will run next
  - Synchronization - mechanisms that enable execution contexts to coordinate their use of shared resources

Threads: Motivation

- Traditional UNIX processes created and managed by the OS kernel
- Process creation expensive - fork system call
- Context switching expensive
- Cooperating processes - no need for protection (separate address spaces)

Threads

- Execute in same address space
  - separate execution stack, share access to code and (global) data
- Smaller creation and context-switch time
- Can exploit fine-grain concurrency
- Easier to write programs that use asynchronous I/O or communication

Threads cont'd

- Less protection against programming errors
- User-level vs kernel-level threads
  - kernel not aware of threads created by user-level thread package (e.g. Pthreads), language (e.g. Java)
  - user-level threads typically multiplexed on top of kernel level threads in a user-transparent fashion

Creating and Using threads

- Solaris Multi-threading Library
  - supports Pthreads API + own Solaris threads API
  - pthread_create, pthread_join, pthread_self, pthread_exit, pthread_detach
- Java
  - provides a Runnable interface and a Thread class as part of standard Java libraries
    - users program threads by implementing the Runnable interface or extending the Thread class

Creating threads

```
class Simple implements Runnable {
   public void run() {
     System.out.println("this is a thread");
   }
}

Runnable s = new Simple();
Thread t = new Thread(s);
t.start();
```

Alternative strategy: Extend Thread class (not recommended unless you are creating a new type of Thread)

Cooperating concurrent processes

- Shared Memory
  - Semaphores, mutex locks, condition variables, monitors
  - Mutual exclusion
- Message-passing
  - Pipes, FIFOs (name pipes)
  - Message queues

Synchronization Mechanisms

- Pthreads
  - Semaphores
  - Mutex locks
  - Condition Variables
  - Reader/Writer Locks
- Java
  - Each object has an (implicitly) associated lock and condition variable

Mutual exclusion in Java

```
class Interfere {
   private int data = 0;
   public synchronized void update() {
      data++;
  }
}
```

```
class Interfere {
   private int data = 0;
   public void update() {
         synchronized(this) {
            data++;
         }
   }
}
```


The Reader/Writer Problem

```
class RWbasic {              // basic read or write (no synch)
  protected int data = 0;  // the "database"
  protected void read() {
          System.out.println("read:  " + data);
   }
  protected void write() {
      data++;
      System.out.println("wrote:  " + data);
   }
}
```



```
class ReadersWriters extends RWbasic {  // Readers/Writers
    int nr = 0;

    private synchronized void startRead() {
         nr++;
    }

    private synchronized void endRead()  {
          nr--;
          if (nr==0)  notify();  // awaken waiting Writers
    }

   public void read() {
      startRead();
      System.out.println("read:  " + data);
      endRead();
  }
```



```
  public synchronized void write() {
     while (nr>0)
         try { wait(); }
            catch (InterruptedException ex  {return;}
     data++;
     System.out.println("wrote:  " + data);
     notify();              // awaken another waiting Writer
   }
}
class Reader extends Thread {
  int rounds;
  ReadersWriters RW;
  public Reader(int rounds, ReadersWriters RW) {
    this.rounds = rounds;
    this.RW = RW;
  }
  public void run() {
    for (int i = 0; i<rounds; i++) {
       RW.read();
  } } }
```

```
class Writer extends Thread {
   int rounds;
   ReadersWriters RW;
   public Writer(int rounds, ReadersWriters RW) {
            this.rounds = rounds;    this.RW = RW;
   }
   public void run() {
            for (int i = 0; i<rounds; i++) {
               RW.write();    }
   }
}
class Main {          // driver program -- two readers and one writer
    static ReadersWriters RW = new ReadersWriters();
    public static void main(String[] arg) {
        int rounds = Integer.parseInt(arg[0],10);
        new Reader(rounds, RW).start();
        new Reader(rounds, RW).start();
        new Writer(rounds, RW).start();
     }
}
```