Concurrent Programming (Review)

Prof. Sanjeev Setia
Distributed Software Systems
CS 707

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

Address space
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

Client and server with threads

[Diagram showing the interactions between client, server, and threads.]

Thread 2 makes requests to server
Receipt & queuing
Requests
N threads
Input-output

Client
Server
**Alternative server threading architectures**

- a. Thread-per-request
- b. Thread-per-connection
- c. Thread-per-object

**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)

**State associated with execution environments and threads**

<table>
<thead>
<tr>
<th>Execution environment</th>
<th>Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address space tables</td>
<td>Saved processor registers</td>
</tr>
<tr>
<td>Communication interfaces, open files</td>
<td>Priority and execution state (such as BLOCKED)</td>
</tr>
<tr>
<td>Semaphores, other synchronization objects</td>
<td>Software interrupt handling information</td>
</tr>
<tr>
<td>List of thread identifiers</td>
<td>Execution environment identifier</td>
</tr>
<tr>
<td></td>
<td>Pages of address space resident in memory; hardware cache entries</td>
</tr>
</tbody>
</table>
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

Java thread constructor and management methods

Thread(ThreadGroup group, Runnable target, String name)
Creates a new thread in the SUSPENDED state, which will belong to group and be identified as name; the thread will execute the run() method of target.

setPriority(int newPriority), getPriority()
Set and return the thread’s priority.

run()
A thread executes the run() method of its target object, if it has one, and otherwise its own run() method (Thread implements Runnable).

start()
Change the state of the thread from SUSPENDED to RUNNABLE.

sleep(int millisecs)
Cause the thread to enter the SUSPENDED state for the specified time.

yield()
Enter the READY state and invoke the scheduler.

destroy()
Destroy the thread.
Creating threads

```java
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

Java thread synchronization calls

thread.join(int millisecs)
Blocks the calling thread for up to the specified time until thread has terminated.

thread.interrupt()
Interrupts thread: causes it to return from a blocking method call such as sleep().

object.wait(long millisecs, int nanosecs)
Blocks the calling thread until a call made to notify() or notifyAll() on object wakes the thread, or the thread is interrupted, or the specified time has elapsed.

object.notify(), object.notifyAll()
Wakes, respectively, one or all of any threads that have called wait() on object.
Mutual exclusion in Java

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