Lecture 3

Operating Systems

Lecture

- Anderson: synchronization continued:
  - lec4: Independent versus cooperating threads
  - lec6: hardware synchronization
  - lec7: semaphores
  - lec8: monitors
- Wrapper Pattern (also called Buffer Pattern)
- Runnable Service Pattern
- Hw 2

Runnable Service Pattern

- Design and implementation convention for encoding a single method as a standalone service.
- Motivation: a time consuming method should be run by a separate thread if it can be run in parallel to the caller.
- Examples: File and network I/O are encapsulated within threads.

Design choices

- In hw 1 you read from an input stream using a separate thread and you read from the error stream using a separate thread.
- The Runnable Service Pattern describes some of the design choices you have

Form of Runnable Service

```
public class ClassForMethod implements Runnable {
  private ARG1 arg1_;
  private ARG2 arg2_;
  public ClassForMethod(ARG1 arg1, ARG2 arg2){
    arg1_ = arg1; arg2_ = arg2;
  }
  public void run() {
    // code that was in Method
    // use data members arg1, arg2 instead of
    // parameters
  }
}
```

```
Runnable r = new ClassForMethod(a1,a2);
Thread t = new Thread(r);
t.start();
other_in_parallel();
```

Explanation

- `run` can neither take arguments nor return results. Therefore this control information must be managed by runnable command object.
- Use constructor to send parameters.
- Many runnable service objects are used only once.

Client-controlled versus server-controlled activation

- Client-controlled: Object that constructs Runnable object also constructs and starts an associated thread.
- `new Thread(new ClassForMethod(a1,a2)).start();`

```
Runnable r = new ClassForMethod(a1,a2);
Thread t = new Thread(r);
t.start();
other_in_parallel();
```

Client-controlled versus server-controlled activation

- server-controlled: Runnable object itself creates and starts thread.
- `new Thread(this).start()`

Semaphores in Java

```
public final class CountingSemaphore {
  private int count_ = 0;
  public CountingSemaphore(int inC) {
    count_ = inC;}
  public void P() { // down
    while (count_ <= 0)
      try {wait();}
        catch (InterruptedException ex) {}
      --count_;}
  public void V() { // up
    ++count_; notify(); }
}
```

From: Concurrent Programming in Java by Doug Lea

Readers and Writers

```
public abstract class RW {
  protected int activeReaders_ = 0; //threads executing read_
  protected int activeWriters_ = 0; //always zero or one
  protected int waitingReaders_ = 0; //threads not yet in read_
  protected int waitingWriters_ = 0; // same for write_

  protected abstract void read_(); //implement in subclasses
  protected abstract void write_();//implement in subclasses
  public void read(){beforeRead(); read_();afterRead();}
  public void write(){beforeWrite(); write_();afterWrite();}
  protected boolean allowReader() {
    return waitingWriters_ == 0 && activeWriters_ == 0;}
  protected boolean allowWriter() {
    return activeReaders_==0 && activeWriters_ == 0;}
```

From: Concurrent Programming in Java by Doug Lea

Readers and Writers

```
// continued: public abstract class RW {
  protected synchronized void beforeRead() {
    ++ waitingReaders_;
    while (!allowReader())
      try {wait();} catch (InterruptedException ex) {}
    -- waitingReaders_;
    ++ activeReaders_; }
  protected synchronized void afterRead() {
    --activeReaders_;
    notifyAll();}
```

Readers and Writers

```
// continued: public abstract class RW {
  protected synchronized void beforeWrite() {
    ++ waitingWriters_;
    while (!allowWriter())
      try {wait();} catch (InterruptedException ex) {}
    -- waitingWriters_;
    ++ activeWriters_; }
  protected synchronized void afterWrite() {
    --activeWriters_;
    notifyAll();}
}
```