From the on-line textbook, pp. 147-154 do:

- Ch 4 Exercises (pencil-and-paper): 4.6
- Programming Project: 4.8 and 4.11
- Exploration Projects: 4.2

Extra Credit: Programming Project: 4.9 or 4.10

Exercises: 4.6

4.6 State licensing rules require a child-care center to have no more than three infants present for each adult. You could enforce this rule using a semaphore to track the remaining capacity, that is, the number of additional infants that may be accepted. Each time an infant is about to enter, an **acquire** operation is done first, with a **release** when the infant leaves. Each time an adult enters, you do three **release** operations, with three **acquire** operations before the adult may leave.

(a) Although this system will enforce the state rules, it can create a problem when two adults try to leave. Explain what can go wrong, with a concrete scenario illustrating the problem.

(b) The difficulty you identified in the preceding subproblem can be remedied by using a mutex as well as the semaphore. Show how.

(c) Alternatively, you could abandon semaphores entirely and use a monitor with one or more condition variables. Show how.

Solution:

a) The semaphore value might be 3 or 4 above the 3-infants-to-1 adult rule when two adults try to leave concurrently. By interleaving their `acquire` operations, the semaphore could reach 0 before either completes their third `acquire` operation. Thus, both adults wait to leave eventhough one should have been able to leave.

b) We just need to make sure that only one adult trying to leave can be performing the three `acquire` operation on the semaphore. The `adultLeave` code would be something like:

```
void adultLeave():
   mutex.lock()
   if semaphore.value >= 3 then
      print "Okay, you can leave"
      semaphore.acquire()
      semaphore.acquire()
      semaphore.acquire()
   else
      print "Sorry you cannot leave"
   mutex.unlock()
```

c) Using a Java-like monitor without semaphores would look something like:

```
class babyMonitor {
   private int numInfants = 0;
   private int numAdults = 0;

   public synchronized void enterInfant() {
      while (numAdults*3 < numInfants) {
         wait();
      } // end while
      numInfants++;
   } // end enterInfant

   public synchronized void infantLeaves() {
      numInfants--;
      notifyAll;
   } // end infantLeaves

   public synchronized void enterAdult() {
      numAdults++;
      notifyAll;
   } // end enterAdult

   public synchronized void adultLeaves() {
      while ((numAdults-1)*3 < numInfants) {
         wait();
      } // end while
      numAdults--;
   } // end adultLeaves

} // end babyMonitor
```

Programming Project: 4.8 and 4.11

4.8 Define a Java class for readers/writers locks, analogous to the **Bounded Buffer** class of Figure 4.17 (page 119). Allow additional readers to acquire a reader-held lock even if writers are waiting. As an alternative to Java, you may use another programming language with support for mutexes and condition variables.

```
/*
  File:  RWLocks.java
  Description:  Readers / Writers Locks solution which allows new
  readers to start reading even if there are waiting writers..
  NOTE(s):  When I/O done in Producer/Consumer loops, I/O waits causes
            them to alternate so the buffer does not fill past one.
            So, I added println's to the insert and retrieve methods.
*/
public class RWLocks
{
   public static void main(String [] args)
   {
       RWLockManager lockManager = new RWLockManager();
       Writer w1 = new Writer(lockManager, 1);
       Writer w2 = new Writer(lockManager, 2);
       Writer w3 = new Writer(lockManager, 3);
       Reader r1 = new Reader(lockManager, 1);
       Reader r2 = new Reader(lockManager, 2);
       Reader r3 = new Reader(lockManager, 3);

       w3.start();
       w2.start();
       r2.start();
       r3.start();
       try {
           Thread.sleep(6000);
       } catch(InterruptedException e) {
           // won't happen
       } // end try-catch
       w1.start();
   } // end main

} // end class RWLocks
```

```
class RWLockManager {
   private int writersOccupied = 0;
   private int readersOccupied = 0;


   public synchronized void readLock()
       throws InterruptedException
   {
       while(writersOccupied > 0)
           // wait for writer to finish
           wait();
       System.out.println("readLock granted");
       readersOccupied++;
   } // end readLock

   public synchronized void writeLock()
       throws InterruptedException
   {
       while(writersOccupied > 0 || readersOccupied > 0)
           // wait for all readers and writer to finish
           wait();
       System.out.println("writeLock granted");
       writersOccupied++;
   } // end writeLock

   public synchronized void rwUnlock()
       throws InterruptedException
   {
       if (writersOccupied > 0 && readersOccupied > 0) {
           System.out.println("ERROR:  unlocking, but both reader and writers!!!");
       }
       if (writersOccupied > 0) {
           writersOccupied--;
           System.out.println("unlocking by a writers!!!");
           notifyAll();
       } else if (readersOccupied > 0) {
           readersOccupied--;
           System.out.println("unlocking by a reader!!!");
           notifyAll();
       } else {
           System.out.println("ERROR:  unlocking, but NO readers or writers!!!");
       } // end if

   }// end rwUnlock
} // end class RWLockManager


class Writer extends Thread {
   private RWLockManager lockManager;
   private int writeTime;

   Writer(RWLockManager lockMgr, int timeToWrite) {
       lockManager = lockMgr;
       writeTime = timeToWrite;
   }
   public void run() {
       try {
           System.out.println("Writer trying to acquire writeLock ");
           lockManager.writeLock();
           System.out.println("Writer acquired writeLock: starting write for  " + writeTime);
           Thread.sleep(writeTime*1000);
           System.out.println("Writer done writing after " + writeTime);
           System.out.println("Writer unlocking writeLock");
           lockManager.rwUnlock();
       } catch (InterruptedException e) {}

   }

} // end class Writer

class Reader extends Thread {
   private RWLockManager lockManager;
   private int readTime;

   Reader(RWLockManager lockMgr, int timeToRead) {
       lockManager = lockMgr;
       readTime = timeToRead;
   }
   public void run() {
       try {
           System.out.println("Reader trying to acquire readLock ");
           lockManager.readLock();
           System.out.println("Reader acquired readLock: starting reading for  " + readTime);
           Thread.sleep(readTime*1000);
           System.out.println("Reader done reading after " + readTime);
           System.out.println("Reader unlocking readLock");
           lockManager.rwUnlock();
       } catch (InterruptedException e) {}

   }

} // end class Reader
```

4.11 Define a Java class for barriers, analogous to the **BoundedBuffer** class of Figure 4.17 (page 119). Alternatively, use another programming language, with support for mutexes and condition variables.

```
/*
  File:  BarrierTest.java
  Description:  Defines a and tests Barrier class to implement a Barrier
  synchronization pattern.
  NOTE(s):  For testing, I create a "thread pool" to avoid the overhead
  of creating new Worker threads for each timeStep.
*/
import java.util.concurrent.*;

public class BarrierTest
{

       public static void main(String [] args) {
              ExecutorService exec = Executors.newFixedThreadPool(4);
              final int NUM_THREADS = 4;
              final int NUM_TIME_STEPS = 5;
              Barrier myBarrier;

              for (int timeStep=0; timeStep < NUM_TIME_STEPS; timeStep++) {

                     myBarrier = new Barrier(NUM_THREADS);

                     for (int threadID=0; threadID < NUM_THREADS; threadID++) {
                         exec.execute(new Worker(threadID, timeStep, myBarrier));
                     } // end for threadID
              } // end for timeStep

              exec.shutdown();
       } // end main

} // end class BarrierTest

class Barrier {
   private int threadsWaitingFor = 0;

   public Barrier ( int numThreads ) {
       threadsWaitingFor = numThreads;
   } // end Barrier

   public synchronized void waitOnBarrier()
       throws InterruptedException
   {
       threadsWaitingFor--;
       System.out.println("arrived at barrier");
       while( threadsWaitingFor > 0) {
           // wait for space remainding threads to reach barrier
           wait();
       } // end while
       notifyAll();
       System.out.println("leaving barrier");
   } // end waitOnBarrier

} // end class Barrier


class Worker extends Thread {
   private Barrier theBarrier;
   private int ID;
   private int step;

   Worker(int threadID, int timeStep, Barrier myBarrier) {
       ID = threadID;
       step = timeStep;
       theBarrier = myBarrier;
   } // end Worker constructor

   public void run() {
       try {
           System.out.println("Worker "+ID+" working on step " + step);
           Thread.sleep((ID+1)*1000);
           theBarrier.waitOnBarrier();
           System.out.println("Worker " + ID + " after the barrier");
       } catch (InterruptedException e) {}

   }

} // end class Worker
```

Exploration Projects: 4.2

4.2 The Java program in Figure 4.30 simulates the dining philosophers problem, with one thread per philosopher. Each thread uses two nested `synchronized` statements to lock the two objects representing the forks to the philosopher's left and right. Each philosopher dines many times in rapid succession. In order to show whether the threads are still running, each thread prints out a message every 100000 times its philosopher dines.

(a) Try the program out. Depending on how fast your system is, you may need to change the number 100000. The program should initially print out messages, at a rate that is not overwhelmingly fast, but that keeps you aware the program is running. With any luck, after a while, the messages should stop entirely. This is your sign that the threads have deadlocked. What is your experience? Does the program deadlock on your system? Does it do so consistently if you run the program repeatedly? Document what you observed (including its variability) and the circumstances under which you observed it. If you have more than one system available that runs Java, you might want to compare them.

(b) You can guarantee the program won't deadlock by making one of the threads (such as number 0) acquire its right fork before its left fork. Explain why this prevents deadlock, and try it out. Does the program now continue printing messages as long as you let it run?

Answers:

a) The Philosopher.java program always seemed to deadlocked quickly on student.cs.uni.edu.

b) The below PhilosopherB.java program continues for run because deadlock is prevented.
Deadlock is prevented because Philosopher 0 acquires his right fork before his left fork. Thus, preventing the formation of a cycle of waiting threads.

```
/* File:  PhilosopherB.java
  Description:  Solve the Dining Philosopher problem by preventing
  deadlock from occurring.  Philosopher 0 acquires his right fork before
  his left fork, so the deadlock cycle is prevented.
*/
public class PhilosopherB extends Thread{

	private Object leftFork, rightFork;
	private int myNumber;

	public PhilosopherB(Object left, Object right, int number){
		leftFork = left;
		rightFork = right;
		myNumber = number;
	}

	public void run(){
		int timesDined = 0;
		while(true){
			synchronized(leftFork){
				synchronized(rightFork){
					timesDined++;
				}
			}
			if(timesDined % 100000 == 0)
				System.err.println("Thread " + myNumber + " is running.");
		}
	}

	public static void main(String[] args){
		final int PHILOSOPHERS = 5;
		Object[] forks = new Object[PHILOSOPHERS];
		for(int i = 0; i < PHILOSOPHERS; i++){
			forks[i] = new Object();
		}

		PhilosopherB p = new PhilosopherB(forks[1], forks[0], 0);
```

```
            p.start();

            for(int i = 1; i < PHILOSOPHERS; i++){
                int next = (i+1) % PHILOSOPHERS;
                p = new PhilosopherB(forks[i], forks[next], i);
                p.start();
            }
        }
}
```

Extra Credit: Programming Project: 4.9 or 4.10

4.9 Modify your readers/writers locks from the prior project so no additional readers may acquire a reader-held lock if writers are waiting.

4.10 Modify your readers/writers locks from either of the prior two projects to support an additional operation that a reader can use to upgrade its status to writer. (This is similar to dropping the read lock and acquiring a write lock, except that it is atomic: no other writer can sneak in and acquire the lock before the upgrading reader does.) What happens if two threads both hold the lock as readers, and each tries upgrading to become a writer? What do you think a good response would be to that situation?