The following questions ask you to analyze some code fragments and to write some code fragments. When you analyze some code, your analysis should be written in complete sentences organized into paragraphs. Do not write sentence fragments and do not write the most terse answer that you can think of (even if it is essentially correct). You are being graded on your ability to communicate, not just on your ability to arrive at correct solutions.

When you write code fragments, you do not need to write compilable code. Just make sure that your code is not in any way ambiguous.

Write all your answers neatly using a computer document format. You can write your answers in a plain text file, a MS Word document, an HTML page, or even in LaTeX. Put your exam, and any supporting files that you might like to submit (like compilable code), in a zip file and submit your zip file to me using Blackboard.

Each person should work on this exam by them self. If you have any questions about the exam, feel free to send me an e-mail.

This exam should be turned in by Friday, December 19.

1. The following code outlines a synchronization pattern. Assume that the two threads begin at the same time, each thread runs on its own core, and there are no other (significant) threads running on the cores.

```
void *thread1(void *vargp)
{  while(1)
   {  << do Calculation A >>
      sem_post(&semaphore1);
      << do Calculation B >>
      sem_post(&semaphore2);
      sem_wait(&semaphore3);
   }
}

void *thread2(void *vargp)
{  while(1)
   {  sem_wait(&semaphore1);
      << do Calculation C >>
      sem_post(&semaphore3);
      sem_wait(&semaphore2);
   }
}

sem_t semaphore1, semaphore2, semaphore3;

int main()
{  pthread_t tid;
   sem_init(&semaphore1, 0, 0); // not signaled
   sem_init(&semaphore2, 0, 0); // not signaled
   sem_init(&semaphore3, 0, 0); // not signaled
   pthread_create(&tid, NULL, thread1, NULL);
   pthread_create(&tid, NULL, thread2, NULL);
   while(1){ Sleep(1000); }
}
```

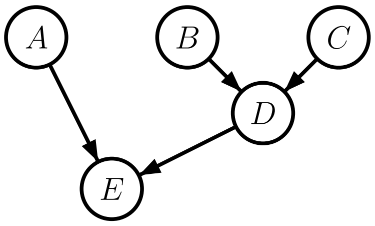
(a) (15 points) In what way are the two threads synchronized? Give your answer in terms of how the three calculations, A, B, and C, are ordered in time. Explain carefully what role each of the three semaphores plays in the synchronization.

Solution:

(b) (15 points) Rewrite this program using condition variables.

Solution:

2. Suppose that we have five C functions that together solve some problem. Suppose these functions, labeled `A` through `E`, depend on each other according to the following graph.



Each edge of the graph denotes a dependency between two of these functions. For example, the edge from node B to node D means that `functionB` must be called, and must return, before `functionD` can be called.

(a) (10 points) What is wrong with this sketch of a C program that uses Pthreads to execute the five functions in parallel in a way that adheres to the above dependency graph? How would you improve this program (but still use five worker threads and only the Pthreads functions `pthread_create()` and `pthread_join()`)?

```
void *threadA(void *vargp){ functionA(); }
void *threadB(void *vargp){ functionB(); }
void *threadC(void *vargp){ functionC(); }
void *threadD(void *vargp){ functionD(); }
void *threadE(void *vargp){ functionE(); }
int main()
{  pthread_t tidA, tidB, tidC, tidD, tidE;
   pthread_create(&tidB, NULL, threadB, NULL);
   pthread_create(&tidC, NULL, threadC, NULL);
   pthread_join(tidB, NULL);
   pthread_join(tidC, NULL);
   pthread_create(&tidA, NULL, threadA, NULL);
   pthread_create(&tidD, NULL, threadD, NULL);
   pthread_join(tidA, NULL);
   pthread_join(tidD, NULL);
   pthread_create(&tidE, NULL, threadE, NULL);
   pthread_join(tidE, NULL);
}
```

Solution:

(b) (10 points) Write another sketch of a Pthreads program to execute the above five functions in a way that is maximally parallel (i.e., always runs as many threads in parallel as possible), adheres to the above dependency graph, and uses the minimal number of threads possible (including the `main()` thread). Your solution should still use only `pthread_join()` for synchronization.

Solution:

(c) (10 points) Write a sketch of a Java program that uses Fork-Join to execute the above five functions in a way that adheres to the above dependency graph, is maximally parallel, and uses the minimum number of threads.

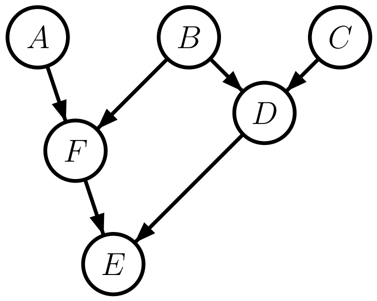
Assume that the five functions are static methods in a class called `MyTasks.java`. Write a sketch of a `main()` method and whatever classes that you need that extend `java.util.concurrent.RecursiveAction`.

Solution:

3. (15 points) Suppose that we have six C functions

```
void functionA(void);   void functionD(void);
void functionB(void);   void functionE(void);
void functionC(void);   void functionF(void);
```

that together solve some problem. Suppose these function depend on each other according to the following dependency graph.



Write a sketch of a C program that uses Pthreads to execute the above six functions in a way that is maximally parallel, but adheres to the above dependency graph. Give a written explanation of how your code solves the problem. You can use any synchronization mechanism you want (join, condition variables, semeaphores, etc.).

Solution:

4. Suppose we wanted to implement our own mutex class in Java.

```
class MyMutex {
   public Thread locked = null;

   public void lock() throws InterruptedException {
      if ( this.locked != null )
         this.wait();
      this.locked = Thread.currentThread();
   }

   public void unlock() {
      if ( this.locked == Thread.currentThread() ) {
         this.notify();
         this.locked = null;
      } else
         throw new IllegalStateException();
   }
}
```

(a) (10 points) Correct the mistakes in this impelentation of a mutex class. Explain the reason for each of your corrections. In particular, what could go wrong if each particular correction isn't made? (You can ignore the `throws InterruptedException`. It is needed for the code to compile.)

Solution:

(b) (5 points) The documentation for `pthread.h`

`http://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread_mutex_lock.html`

says that the pthread library implements several kinds of mutexes, `PTHREAD_MUTEX_NORMAL`, `PTHREAD_MUTEX_ERRORCHECK`, `PTHREAD_MUTEX_RECURSIVE`, and `PTHREAD_MUTEX_DEFAULT`. Use the Java documentation

`https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html`

to determine which of these kinds of pthread mutexes our repaired Java mutex is most similar to with respect to the `lock()` method. Explain why.

Solution:

(c) (5 points) With respect to the `unlock()` method, which of the above kinds of pthread mutexes is our repaired Java mutex most similar to? Explain why.

Solution:

(d) (5 points) The proper way to use one of our MyMutex objects looks like this.

```
mutex.lock();
// do something
mutex.unlock();
```

Explain carefully what would happen, and why, if you used one of our corrected MyMutex objects like this.

```
mutex.lock();
// do something
mutex.notify();
```

Solution: