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 of your answers neatly on separate pieces of paper. If you want, you can write your answers using a text editor, Word, or even LaTeX. If your answers are all contained in an electronic document (e.g., text file, Word document, pdf file, etc.), then you can submit your exam to me by Blackboard. If you write up your solutions long hand, then turn in your exam in my mailbox in the Mathematics Department office or in class.

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 or call me on the phone.

This exam should be turned in on Thursday, October 25.

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) processes running on the cores.

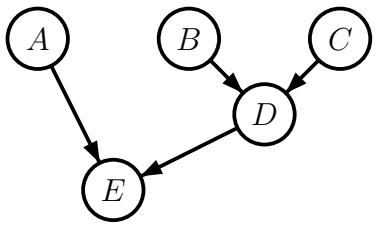
```
void *thread1(void *vargp)
{  while(1)
   {  << do Calculation A >>
      sem_post(&semaphore1);
      << do Calculation B >>
      sem_post(&semaphore2);
      sem_wait(&semaphore3);
   }
}

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) (10 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.

(b) (10 points) Rewrite this program fragment 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 *thread1(void *vargp){ functionA(); }
void *thread2(void *vargp){ functionB(); }
void *thread3(void *vargp){ functionC(); }
void *thread4(void *vargp){ functionD(); }
void *thread5(void *vargp){ functionE(); }
int main()
{  pthread_t tid1, tid2, tid3, tid4, tid5;
   pthread_create(&tid2, NULL, thread2, NULL);
   pthread_create(&tid3, NULL, thread3, NULL);
   pthread_join(tid2, NULL);
   pthread_join(tid3, NULL);
   pthread_create(&tid1, NULL, thread1, NULL);
   pthread_create(&tid4, NULL, thread4, NULL);
   pthread_join(tid1, NULL);
   pthread_join(tid4, NULL);
   pthread_create(&tid5, NULL, thread5, NULL);
   pthread_join(tid5, NULL);
}
```

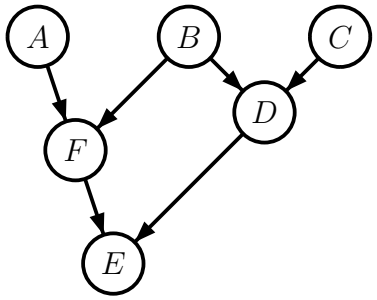
(b) (10 points) Write another sketch of a Pthreads program to execute the above five functions in a way that is maximally parallel, adheres to the above dependency graph, and uses the minimal number of threads possible (including the `main()` thread). Your solution should still use only `pthread_join()`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

(c) (10 points) Write a sketch of a C program that uses OpenMP to execute the above five functions in a way that is maximally parallel, but adheres to the above dependency graph.

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

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

Solution:

4. (15 points) Suppose that we have a computer with 4 cores. Suppose we use OpenMP to parallelize a for-loop that initializes to zero the upper triangle of a 100×100 matrix.

```
#pragma openmp parallel for private(j) schedule( ... )
for (i = 0; i < 99; i++)
   for (j = i+1; j < 100; j++)
   {
      a[i][j] = 0.0;
   }
```

Notice that the `schedule()` clause has been left undefined. Below are six example schedule clauses that could be used. Rank these clause from slowest to fasted. In particular, since each iterate of the inner loop above does just one assignment, we can estimate the execution time by counting how many assignments each thread does (notice that all together, there are exactly 4,950 assignments).

So for each schedule clause, estimate how long the parallelized loop will run. Explain how you arrived at your estimates.

- `schedule(static)`
- `schedule(static, 10)`
- `schedule(static, 1)`
- `schedule(dynamic, 1)`
- `schedule(dynamic, 10)`
- `schedule(dynamic, 20)`

Solution:

5. Consider the following OpenMP program.

```
#include <stdio.h>
#include <omp.h>
#define NUM_OF_COLUMNS 6
#define NUM_OF_ROWS   (3*(NUM_OF_COLUMNS - 1))

int whichThread[NUM_OF_ROWS][NUM_OF_COLUMNS];

void fillColumn(int j)
{  int i;
   #pragma omp for
   for (i = 0; i < NUM_OF_ROWS; i++)
      whichThread[i][j] = omp_get_thread_num();
}

int main()
{  int i, j;
   for (i = 0; i < NUM_OF_ROWS; i++)  // initialize the array
      for (j= 0; j < NUM_OF_COLUMNS; j++)
         whichThread[i][j] = -1;

   #pragma omp parallel num_threads(NUM_OF_COLUMNS - 1)
      fillColumn(0);

   #pragma omp parallel for num_threads(NUM_OF_COLUMNS - 1)
   for (j = 1; j < NUM_OF_COLUMNS; j++)
      fillColumn(j);

   for (i = 0; i < NUM_OF_ROWS; i++)  // print out the results
   {  for (j= 0; j < NUM_OF_COLUMNS; j++)
         printf(" %2d ", whichThread[i][j]);
      printf("\n");
   }
   return 0;
}
```

(a) (10 points) What is the output of this program if it is compiled without the `-fopenmp` compiler flag? Briefly explain why.

(b) (10 points) What is the output of this program if it is compiled with the `-fopenmp` compiler flag? Explain, very carefully, why the output differs from the output of the serial version of the program.