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.

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 themself. If you have any questions about the exam, feel free to send me an e-mail.

This exam should be turned in to Blackboard by Friday, December 14.

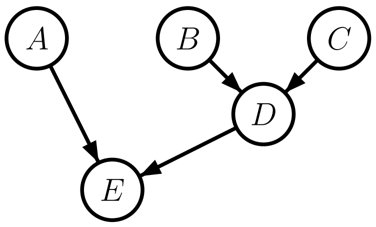
1. The following code outlines a synchronization pattern. Assume that the two threads begin at the same time, each thread runs on its own CPU core, and there are no other (significant) threads running on the cores.

```
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) (8 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 what parts are sequential and what parts are in parallel. Explain carefully what role each of the three semaphores plays in the synchronization.

(b) (12 points) Rewrite this program using condition variables.

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) (8 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()`)?

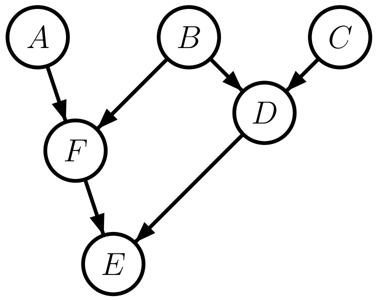
```
void *threadA(void *vargp){ functionA(); }
void *threadB(void *vargp){ functionB(); }
void *threadC(void *vargp){ functionC(); }
void *threadD(void *vargp){ functionD(); }
void *threadE(void *vargp){ functionE(); }
int main()
{  pthread_t tidA, tidB, tidC, tidD, tidE;
   pthread_create(&tidB, NULL,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

(b) (12 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.

3. (12 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, semaphores, etc.).

4. (15 points) This question is about I/O redirection and pipes on the Windows command-line. Explain what each of the following possible command-lines mean. In each part, you need to associate an appropriate meaning to the symbols `a`, `b` and `c` (for example "b names a program, a and c name files" or "a and b name programs and c is a parameter to b"). Also give an example of a runnable command-line with the given format (using programs like `dir`, `sort`, `more`, etc.). You might find the following web pages useful.

   `http://ss64.com/nt/syntax-redirection.html`
   `https://technet.microsoft.com/en-us/library/bb490954.aspx`

   (a) `z:\> a > b < c`

   (b) `z:\> a | b > c`

   (c) `z:\> a < b | c`

   (d) `z:\> a < b & c`

   (e) `z:\> a b | c`

   (f) `z:\> a & b | c`

   (g) `z:\> a & (b | c)`

5. (8 points) What problem is there with each of the following two command lines?

   (a) `z:\> a | b < c`

   (b) `z:\> a > b | c`

6. (5 points) Draw a picture illustrating the processes, streams, pipes, and files in this command-line.

   `z:\> a < b | c 2> d | e > f 2> d`

7. (10 points) Draw a picture that illustrates all the processes, pipes, file descriptors, and (possibly shared) streams after the following code has executed.

```
int main() {
   int pipefd_1[2];
   int pipefd_2[2];
   pipe(pipefd_1);
   pipe(pipefd_2);

   int fork_rv1 = fork();
   if ( fork_rv1 > 0 ) {
      int fork_rv2 = fork();
      if ( fork_rv2 > 0 ) {
         close(4);
         close(5);
         dup(6);
         close(6);
         while(1)/* do work... */;
      } else {
         close(5);
         close(6);
         close(3);
         close(1);
         dup(4);
         close(4);
         while(1)/* do work... */;
      }
   } else {
      close(3);
      close(4);
      close(6);
      close(0);
      dup(5);
      close(5);
      while(1)/* do work... */;
   }
   return 0;
}
```

8. (10 points) Draw a picture that illustrates all the processes, pipes, file descriptors, and (possibly shared) streams after the following code has executed.

```
int main() {
   int pipefd_1[2];
   int pipefd_2[2];
   pipe(pipefd_1);
   pipe(pipefd_2);

   int fork_rv1 = fork();
   if ( fork_rv1 > 0 ) {
      int fork_rv2 = fork();
      if ( fork_rv2 > 0 ) {
         close(3);
         close(4);
         close(5);
         close(6);
         while(1)/* do work... */;
      } else {
         close(4);
         close(6);
         close(0);
         dup(3);
         close(3);
         dup(5);
         close(5);
         while(1)/* do work... */;
      }
   } else {
      close(3);
      close(5);
      close(1);
      dup(4);
      close(4);
      dup(6);
      close(6);
      while(1)/* do work... */;
   }
   return 0;
}
```