CSC 374/407: Computer Systems II: Final (2016 Summer II)

Joe Phillips
Last modified 2016 August 23

Name: _____

Distance Learning Students Only!

If you want your graded final returned to you please write your address below:

_____

_____

_____

4 points free, then 16 points per question

1. **Optimization and Compilers**

   There are at least 4 optimizations that can be made in `optimizeMe()`. Find four optimization and for each:

   a. *do* it,
   b. tell whether the *compiler* or *programmer* should make it,
   c. tell *why* either the compiler or programmer (as opposed to the other) should make it

```
//  PURPOSE:  To harass Computer Systems II students.  Computes some arbitrary
//      function of 'array', 'arrayLen', 'numTextPtr0' and 'numTextPtr1' that I
//      pulled out of my a**.   Returns its value.
unsigned int    optimizeMe      (char*  cPtr0,
                                 char*  cPtr1
                                )
{
  unsigned int  sum     = 0;
  int           i;

  for  (i = 0;  i < strlen(cPtr0);  i++)
  {
    int                 j;
    unsigned int        temp    = 5;

    for  (j = 0;  j < strlen(cPtr1);  j++)
    {
```

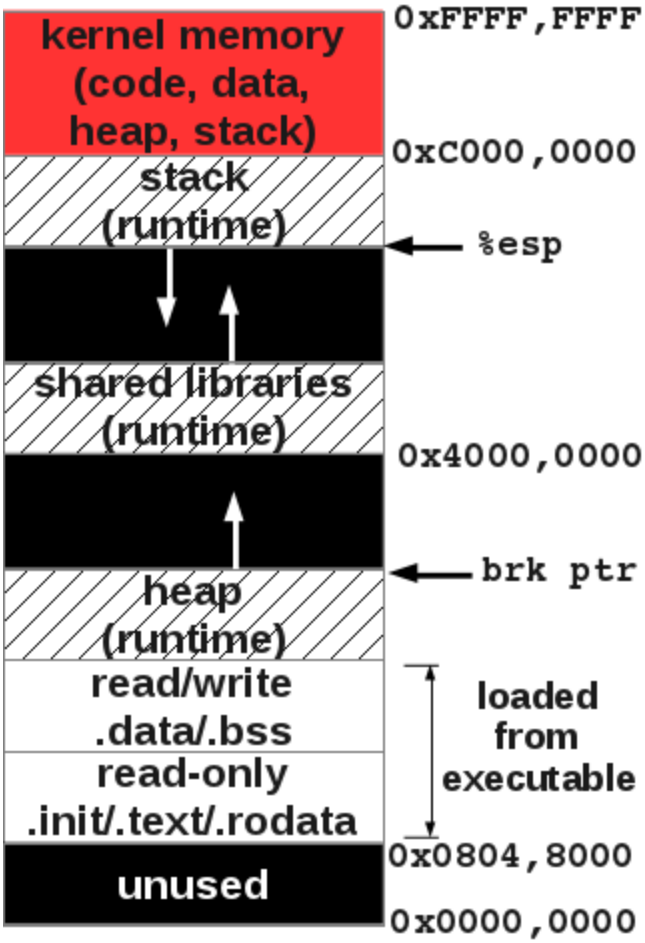
```
      if  ( isalpha(cPtr1[j]) )
        temp     = 4 * temp;
      else
      if  ( isspace(cPtr1[j]) )
        temp     = 5 * temp;
      else
        temp    /= 2;

    }

    sum += temp;
  }

  return(sum);
}
```

Num	Optimitization (just do above)	Compiler or Programmer?	Why done by the person (or program) you said?
(i)			
(ii)			
(iii)			
(iv)			

2. **Memory**

A running program has:

- `%esp` == `0xA000,0000`.
- The program uses `0x0001,0000` bytes of shared library code.
- The `break` pointer == `0x0810,0000`
- There are 0x0000,8000 bytes of *read-only code in the executable file*
- There are 0x000B,0000 bytes of *global vars*



HINTS:

- Compute the boundaries like where the global vars begin, heap begins, *etc*
- Remember this is *hexadecimal:* 0x800 + 0x800 = 0x1000

Questions:

a. **How big is the executable file?**

b. **How much memory does it use for local variables, return addresses, etc.?**

c. **How much memory did it get from `malloc()`, `new`, etc.?**

d. **How much memory does the process have for things like `glibc`?**

3. **Processes, Exceptions and Signals**

A parent process `fork()`s two child processes. These two children are trying to do the same task, but use different approaches. As soon as one child process finishes, it should notify the parent process. Then, the parent process should tell the other process to stop working on that task. (*Note:* the finishing child only needs to tell the parent that it has finished, nothing else.)

Then, the parent should give both child processes another task to do two different ways (one for each child), and both children should work on the new task until one of them finishes.

A. Are signals able to accomplish the child-to-parent communication? Tell how you would do it. *Note:* The only thing the child has to communicate to the parent is that it is finished, **but** the parent should be able to distinguish which child finished.

   If signals **are** sufficient, tell how you would do it.

   If signals are **not** sufficient, tell how an alternative would work.

B. When the parent is notified by one child process, it should tell the other process to stop working (not to quit running, just stop working on that task).

   If signals **are** sufficient, tell how you would do it.

   If signals are **not** sufficient, tell how an alternative would work.

C. The parent process has to give both children the next task to do. The description of this task is a class or struct that takes several bytes to describe.

   If signals **are** sufficient to communicate the task, tell how you would do it.

   If signals are **not** sufficient, tell how an alternative would work.

D. A parent process `open()`s a file, and reads the first 10 lines, which are comments and should be ignored. It then `fork()`s a child process. The child process wants to access the same file and also ignore the same first 10 lines.
   Must the child process re-open the file, and re-read the first 10 lines?
   If so, why?
   If not, why not?

4. **Threads**

   Two *chocolativorous professors* (one female, the other male) **need** `NUM_PIECES_OF_CHOCOLATE_TO_EAT` each to teach what they know to *long-suffering student*. They both steal chocolate by decrementing `numPiecesOfChocolate`, then teach some more. When either has taught `NUM_PIECES_OF_CHOCOLATE_TO_EAT` times, that thread stops and increments `grade` by 50.

   A single *long-suffering student* needs a value of `grade` of `100`. The student puts out pieces of chocolate, one-by-one, by incrementing `numPiecesOfChocolate`.

Only one thread at a time should be able to access `grade`, and only one thread at a time should be able to access `numPiecesOfChocolate`. There is no limit to how high `numPiecesOfChocolate` can get, however, it cannot be decremented lower than `0`.

Your job is to make both `chocolativoreProf()` and the `longSufferingStudent()` **thread-safe**. (All three threads have already be created and joined, **you** must add the mutex(es) and condition(s).)

Stop! Think!

- What must be protected?
- What needs to be signaled?

chocolativoreProfs.c:

```
#include        <stdlib.h>
#include        <stdio.h>
#include        <pthread.h>

const int       NUM_PIECES_OF_CHOCOLATE_TO_EAT  = 5;

int             numPiecesOfChocolate            = 0;
int             grade                           = 0;

//  Perhaps add conditions and mutexes here:


void*           chocolativoreProf       (void*  vPtr
                                        )
{
  int           genderIndex     = *(int*)vPtr;
  int           i;
  const char*   profName        = genderIndex ? "Male Prof" : "Female Prof";
  const char*   genderName      = genderIndex ? "man" : "woman";

  for  (i = 0;  i < NUM_PIECES_OF_CHOCOLATE_TO_EAT;  i++)
  {

    while  (numPiecesOfChocolate <= 0)
    {
      printf("%s: \"You can't expect a %s to teach on an empty stomach!\"\n",
             profName,genderName
            );
    }

    printf("%s: \"Chocolate!  Yummy!   I'll take one of those!\"\n",profName);
    numPiecesOfChocolate--;
    printf("%s: \"Now, as I was saying, blah blah blah . . .\"\n",profName);


    sleep(rand() % 3);  // Please leave this OUT of the critical section
```

```
  }

  printf("%s \"Very good!  I'm submitting your grade now.\"\n",profName);

  grade += 50;
  return(NULL);
}


void*           longSufferingStudent    (void*  vPtr)
{
  int   numPiecesOfChocolatePutOut      = 0;

  while  (1)
  {

    if  (grade >= 100)
      break;


    printf("Student: \"I have to do still *more* work?!?\"\n");

    printf("Student: \"Okay I'll learn some more, "
           "and leave some chocolate out.\"\n"
          );
    numPiecesOfChocolate++;
    numPiecesOfChocolatePutOut++;

    sleep(rand() % 3);  // Please leave this OUT of the critical section
  }

  printf("Student \"I *finally* got my grade!  "
         "And only took %d pieces of chocolate.\"\n",
         numPiecesOfChocolatePutOut
        );
  return(NULL);
}


int     main    ()
{
  pthread_t     femaleProfThread;
  pthread_t     maleProfThread;
  pthread_t     studentThread;
  int           femaleProfIndex = 0;
  int           maleProfIndex   = 1;

  //  Perhaps initialize mutex(es) and condition(s) here:

  pthread_create(&femaleProfThread,NULL,chocolativoreProf,&femaleProfIndex);
  pthread_create(&maleProfThread,NULL,chocolativoreProf,&maleProfIndex);
  pthread_create(&studentThread,NULL,longSufferingStudent,NULL);

  pthread_join(femaleProfThread,NULL);
  pthread_join(maleProfThread,  NULL);
```

```
  pthread_join(studentThread,   NULL);

  //  Perhaps destroy mutex(es) and condition(s) here:

  return(EXIT_SUCCESS);
}
```

Sample output:

```
$ ./chocolativoreProfs
Female Prof: "You can't expect a woman to teach on an empty stomach!"
Student: "I have to do still *more* work?!?"
Student: "Okay I'll learn some more, and leave some chocolate out."
Male Prof: "Chocolate!  Yummy!   I'll take one of those!"
Male Prof: "Now, as I was saying, blah blah blah . . ."
Female Prof: "You can't expect a woman to teach on an empty stomach!"
Student: "I have to do still *more* work?!?"
Student: "Okay I'll learn some more, and leave some chocolate out."
Student: "I have to do still *more* work?!?"
Student: "Okay I'll learn some more, and leave some chocolate out."
Male Prof: "Chocolate!  Yummy!   I'll take one of those!"
Male Prof: "Now, as I was saying, blah blah blah . . ."
Female Prof: "Chocolate!  Yummy!   I'll take one of those!"
Female Prof: "Now, as I was saying, blah blah blah . . ."
Student: "I have to do still *more* work?!?"
Student: "Okay I'll learn some more, and leave some chocolate out."
Female Prof: "Chocolate!  Yummy!   I'll take one of those!"
Female Prof: "Now, as I was saying, blah blah blah . . ."
Female Prof: "You can't expect a woman to teach on an empty stomach!"
Male Prof: "You can't expect a man to teach on an empty stomach!"
Student: "I have to do still *more* work?!?"
Student: "Okay I'll learn some more, and leave some chocolate out."
Student: "I have to do still *more* work?!?"
Student: "Okay I'll learn some more, and leave some chocolate out."
Male Prof: "Chocolate!  Yummy!   I'll take one of those!"
Male Prof: "Now, as I was saying, blah blah blah . . ."
Female Prof: "Chocolate!  Yummy!   I'll take one of those!"
Female Prof: "Now, as I was saying, blah blah blah . . ."
Student: "I have to do still *more* work?!?"
Student: "Okay I'll learn some more, and leave some chocolate out."
Female Prof: "Chocolate!  Yummy!   I'll take one of those!"
Female Prof: "Now, as I was saying, blah blah blah . . ."
Male Prof: "You can't expect a man to teach on an empty stomach!"
Female Prof: "You can't expect a woman to teach on an empty stomach!"
Student: "I have to do still *more* work?!?"
Student: "Okay I'll learn some more, and leave some chocolate out."
Male Prof: "Chocolate!  Yummy!   I'll take one of those!"
Male Prof: "Now, as I was saying, blah blah blah . . ."
Male Prof: "You can't expect a man to teach on an empty stomach!"
Student: "I have to do still *more* work?!?"
Student: "Okay I'll learn some more, and leave some chocolate out."
Student: "I have to do still *more* work?!?"
Student: "Okay I'll learn some more, and leave some chocolate out."
```

```
Student: "I have to do still *more* work?!?"
Student: "Okay I'll learn some more, and leave some chocolate out."
Female Prof: "Chocolate!  Yummy!   I'll take one of those!"
Female Prof: "Now, as I was saying, blah blah blah . . ."
Male Prof: "Chocolate!  Yummy!   I'll take one of those!"
Male Prof: "Now, as I was saying, blah blah blah . . ."
Female Prof "Very good!  I'm submitting your grade now."
Student: "I have to do still *more* work?!?"
Student: "Okay I'll learn some more, and leave some chocolate out."
Male Prof "Very good!  I'm submitting your grade now."
Student "I *finally* got my grade!  And only took 12 pieces of chocolate."
```

5. **Practical C Programming**

   a. (4 Points) Why should we use `snprintf()` instead of `sprintf()`, `strncpy()` instead of `strcpy()`, *etc.*? Seriously, how bad can using `sprintf()`, `strcpy()`, *etc.* be?

   b. (4 Points) What does `extern` mean?
   What does it tell the compiler to do?

   c. (8 Points) The program below will compile well but run poorly. Please make it *do error checking* and fix it to make it proper:

```
#include        <stdlib.h>
#include        <stdio.h>
#include        <string.h>

const int       LINE_LEN        = 1024;

int     main    (int    argc,
                 char*  argv[]
                )
{
  const char*   filename        = argv[1];
  const char*   limitNumText    = argv[2];
  FILE*         fp              = fopen(filename,"r");
  int           limit           = strtol(limitNumText,NULL,10);
  int           haveReachedEnd  = 0;
  char*         line;
  int           counter;
```

```
  while  (1)
  {
    for  (counter = 0;  counter < limit;  counter++)
    {
      if  (fgets(line,LINE_LEN,fp) == NULL)
      {
        haveReachedEnd  = 1;
        break;
      }

      printf(line);
    }

    if  (haveReachedEnd)
      break;

    printf("Press enter to see the next %d lines:",limit);
    gets(line);
  }

  return(EXIT_SUCCESS);
}
```

6. **Sockets and General I/O**

Please finish the server of this client/server application.

1. In `handleClient()` the server receives a 4-byte integer *in network endianness* over file descriptor `fd` that tells the length of the string that the client will send to the server. Call the length of this string `strLength`.
2. The server receives precisely `strLength` characters from the client. The characters are a string (and already include the null character '\0') that tell the name of a directory.
3. The server attempts to open the named directory:
   - if the server *can* open it, it sends the client the 4-byte integer `SUCCESS`
   - if the server can *not* open it, it sends the client the 4-byte integer `FAILURE` and `return`s.

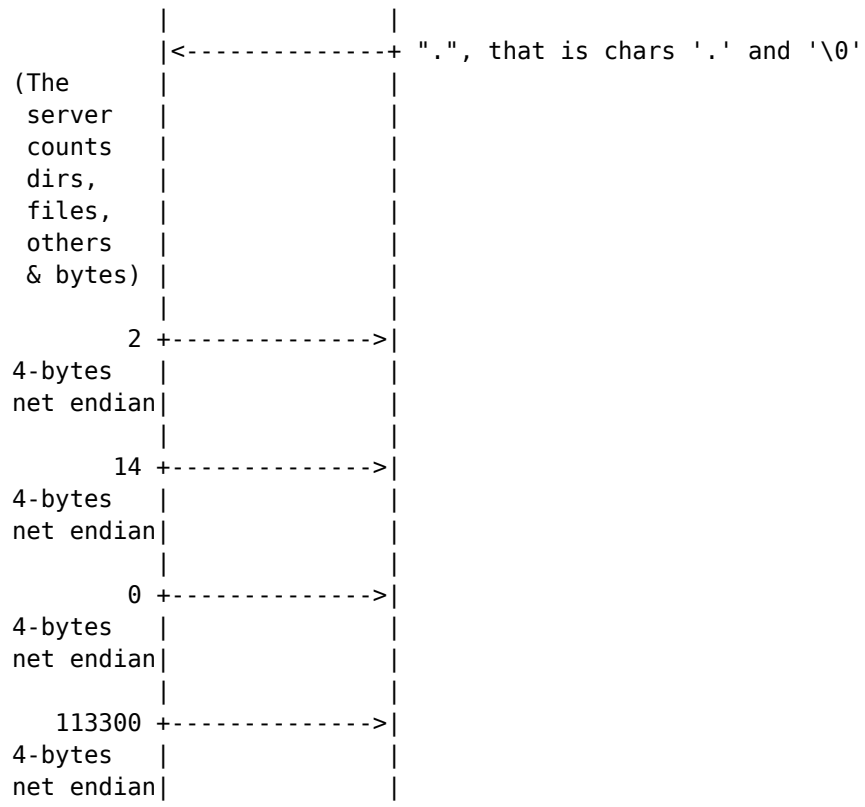
   Either value is sent back to the client in *network endianness*.
4. If the server did open the directory, then it iterates through all entries in the directory and counts:
   - the total number of directories
   - the total number of files
   - the total number of entries that are neither directory nor file
   - the total number of bytes in all files

   It then sends back those four 4-byte integer values in *network endianness* to the client, and closes the directory.

I have done all the boring `socket()`, `bind()`, `listen()`, `accept()` calls for you. Please finish `handleClient()` that handles one client after they have `connect()`ed.

Sample protocol:

```
Directory "." has:
   2 directories
  14 files with a total of 113300 bytes among them
   0 other entries


        server          client
          |               |
          |<--------------+ connect()
          |               |
accept()  +-------------->|
          |               |
          |               |
          |               |
          |<--------------+ 2 (the length of string "." including the '\0')
          |               |   (the 2 is a 4-byte integer in network endianness)
```




```
void    handleClient    (int    fd
                         )
{
  //  I.  Application validity check:

  //  II.  Get counts of requested dir:
  //  II.A.  Get requested dir:
  //  II.A.1.  Get length of dir name:
  int           strLength;
  int           networkInt;

  //  (a) YOUR CODE HERE

  //  II.A.2.  Get dir name:
  char          dirName[BUFFER_LEN];

  //  (b) YOUR CODE HERE

  //  II.B.  Generate response:
  //  II.B.1.  Attempt to open dir:
  DIR*  dirPtr  = NULL;

  //  (c) YOUR CODE HERE

  //  II.B.2.  Generate response depending on success or failure:
  int status;

  if  (dirPtr == NULL)
  {
```

```
    status      = FAILURE;
    //  (d) YOUR CODE HERE
    return;
  }

  status        = SUCCESS;
  //  (e) YOUR CODE HERE

  unsigned int  dirCount        = 0;
  unsigned int  fileCount       = 0;
  unsigned int  otherCount      = 0;
  unsigned int  totalFileSize   = 0;
  struct dirent*        entryPtr;

  while  ( (entryPtr = NULL /* <-- (f) YOUR CODE: REPLACE NULL */ ) != NULL )
  {
    struct stat statBuffer;
    char        path[BUFFER_LEN];

    snprintf(path,BUFFER_LEN,"%s/%s",dirName,entryPtr->d_name);
    //  (g) YOUR CODE HERE

    if  ( false /* <-- (h) YOUR CODE: REPLACE false */ )
    {
      fileCount++;
      totalFileSize     += statBuffer.st_size;
    }
    else
    if  ( false /* <-- (i) YOUR CODE: REPLACE false */ )
      dirCount++;
    else
      otherCount++;
  }

  //  (j) YOUR CODE HERE

  //  III.  Finished:
}
```