Q1) Consider an application with one ***master thread***, one ***producer thread***, and **N *consumer threads***. (N >=4)

The application has a central data structure: An **N (**this is equal to the number of consumers) element array of doubly-linked lists (you will need a head pointer and a tail pointer to represent each doubly-linked list). Each linked list consists of a task queue for one of the four consumer threads. Each element in the linked list stores an "oldvalue" and "newvalue" (in addition to pointers for the next and previous elements).

The producer thread reads an ***input file*** that contains a number of entries. Each entry contains the id of the consumer thread that should handle this task, and two other integers ("oldvalue" and "newvalue") (like 0 1 1 …). After reading an entry, the producer inserts this task at the tail of the task queue for the corresponding consumer. It then sleeps for a random amount of time (between 0 to 1 milli-seconds. (Explore functions like [usleep](http://man7.org/linux/man-pages/man3/usleep.3.html)/[nanosleep](http://man7.org/linux/man-pages/man2/nanosleep.2.html)) and then reads the next entry in the input file.

The consumer thread pulls out the head of its task queue (does nothing if the queue is empty) and appends the values of "old value" and "new value" to its output file. It then sleeps for a random amount of time (between 0 to 4 milli-seconds) and checks the task queue again. At the end of the program, the output file (for the input file listed above) should be available for all the consumers.

While all of the above is happening, the master thread keeps walking through all the linked lists and incrementing the value of "new value" in every element. After walking through every linked list, it sleeps for a random amount of time (between 0 to 1 Nano-second).

Write your implementations (two versions) with

1. Using pthread\_mutex\_lock
2. Using pthread\_mutex\_trylock

2) Implement the following functionality in a C program using appropriate system calls.

a) A process accepts n (n>=1) command line arguments say arg1, arg2, ...argn.

b) Assume that the command line arguments are filenames.

c) The initial process needs to create **n** Threads.

d) Individual Threads should get as argument one of each of arg1, arg2....

e) The individual Threads need to find the size of the files, which they have received as argument.

f) The threads need to send this information back to the main thread. (Basically they need to update a ***global data structure*** ***atomically*** for this purpose, which can be read by the main thread.)

g) The main, while waiting for the other threads to complete, shall retrieve the returned information and display it that is in this form ***Filename, Filesize***. (It can scan the updated global data structure and print these information)

h) The main thread exits after completion of this task (It should free all dynamic memories allocated, if any)

3) Implement the following functionality in a C program using appropriate system calls.

a) A process accepts n (n>=2) command line arguments say arg1, arg2, ...argn.

b) Assume that the command line arguments are filenames.

c) An unnamed pipe is created by this process.

d) The initial process creates two threads say T1 & T2.

e) T1 opens the file, indicated by arg1, reads the file and writes the data over the pipe, which is supposed to be read by T2. It does the same for all files up to argn-1 and then exits.

f) T2 Reads the file's data, which is sent by T1 over the pipe. It writes the data to a new file named by ***argn*** (the last argument).

This means T2 should create file argn, which is the concatenation of data of files arg1, agr2..argn-1.

(If the user specifies only 2 arguments then the file denoted by the 2nd argument will be copy of the first one.)

g) The parent waits for the termination of both T1 and T2. It then print the contents of the concatenated file and then exits.