Task 2: My code satisfies the 3 conditions of the critical section solution. There is mutual exclusion in the entry section of each thread where the mutex is locked and only one thread has access to incrementing counter->value. There is progress as each thread is has work to complete in their critical section and no thread stays in the critical section because each thread unlocks the mutex after their critical section is finished executing. There is bounded waiting as after a thread makes a request to enter their critical section, there is a limit to the amount of times the other thread will execute before the original request is granted.

Code:

```
/*
 * Author: Samuel Casto
 * PantherID: 6330314
 * Description: This program will initialize two threads that will update a single counter variable
independently and
 * concurrently using mutex locks. Each thread will perform 2000000 actions and will increment the
global variable up
 * to 4000000. The program will then output the number of times it performed an action, applied a
bonus if applicable,
 * and will output the value of the counter variable. The program will then output the value of the
counter variable
 * again after both threads have finished processing.
 */
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>


#define MAX_UPDATES 2000000


//struct variable that houses our counter variable
struct sharedData {
	int value;
};
```

```
//global shared variable
struct sharedData *counter;


//mutex variables
pthread_mutex_t mutex;
pthread_cond_t cond;


//thread1 variable to keep track of "bonus"
int bonus = 0;


//thread1 sample code
void *thread1() {
        int i;
        while(i < MAX_UPDATES) {
                //entry section
                pthread_mutex_lock(&mutex);


                //critical section
                //performing our increment but for thread1 when counter % 100 == 0 then we need to
increment by 100
                if(counter->value % 100 == 0 && (counter->value <= 3999900)) {
                        counter->value+= 100;
                        bonus++;
                        i++;
                }
                //we can't increment our value over 4000000
                else if(counter->value <= 3999999) {
                        //testing statement for error checking
                        //printf("This is counter->value before incrementing: %d\n",counter->value);
```

```
            counter->value++;

            i++;

        }

        else {

            i++;

        }

        //exit section

        pthread_cond_signal(&cond);

        pthread_mutex_unlock(&mutex);


    }//end of for loop


    //remainder section

    printf("I'm thread1, I did %d updates and I performed the bonus %d times, counter =
%d.\n",i,bonus,counter->value);

    return NULL;

}


//thread2 code

void *thread2() {

    int i;

    while(i < MAX_UPDATES) {

        //entry section

        pthread_mutex_lock(&mutex);


        //critical section

        //performing our increment

        if(counter->value <= 3999999) {

            counter->value++;
```

```
                i++;
            }
            else {
                i++;
            }
            //exit section
            pthread_cond_signal(&cond);
            pthread_mutex_unlock(&mutex);

        }//end of for loop

        //remainder section
        printf("I'm thread2, I did %d updates, counter = %d\n",i,counter->value);
        return NULL;
}

//main sample code
int main() {

        //create threads
        pthread_t tid[2];
        int rc;

        /* Allocate memory for shared data */
        counter = (struct sharedData *) malloc(sizeof(struct sharedData));
        counter->value = 0;

        /* Initialize mutex lock */
        if ((pthread_mutex_init(&mutex, NULL))) {
```

```
        printf("Error occured when initialize mutex lock.");
        exit(0);
}
pthread_attr_t attr;
if ((pthread_attr_init(&attr))) {
        printf("Error occured when initialize pthread_attr_t.");
        exit(0);
}

/* Create thread1 */
if ((rc = pthread_create(&tid[0], &attr, thread1, NULL))) {
        fprintf(stderr, "ERROR: pthread_create, rc: %d\n", rc);
        exit(0);
}
//create thread2
if ((rc = pthread_create(&tid[1], &attr, thread2, NULL))) {
        fprintf(stderr, "ERROR: pthread_create, rc: %d\n",rc);
        exit(1);
}

/* Wait for threads to finish */
pthread_join(tid[0], NULL);
pthread_join(tid[1], NULL);

//print final wanted information
printf("from parent counter %d\n",counter->value);



/* Clean up */
```

```
    pthread_mutex_destroy(&mutex);
    free(counter);
    pthread_exit(NULL);

    return 0;
}
```