Name: Thomas Gaul
Section: 3
University ID: 925483586

Lab 3 Report

Summary:

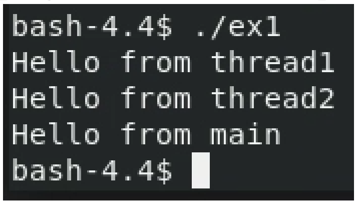
20pts

In this lab, I started getting experience with threads in the C on the Linux machines. I first investigated the the relation between threads and the function that called them, learning that the calling function must avoid exiting to get it to execute. I then explored locking procedures for when threads need to share information. Expanding off locking, I explored pthread conditions for ensuring the order of execution between threads. Putting all these concepts together, I created a basic program with producers creating more supplies when consumers consume all the supplies. Throughout this lab, particularly the tail end, I struggled to ensure I got the code to work with the best practices. There were a few instances where the code worked as desired but it didn't seem like it was the "intended" solution, making use of all the functions intended but believe I figure it out. After this lab, I feel a lot more confident with threads and hope to explore more of their potential in the future.

Lab Questions:

3.1:

10pts To make sure the main terminates before the threads finish,
add a sleep(5) statement in the beginning of the thread functions.
Can you see the threads' output? Why?
No, the thread's output cannot be seen because the main terminates and exits before the threads get the chance to print out, leaving the threads unable to print out.
5pts Add the two *pthread_join* statements just before the printf statement in main. Pass a value of NULL for the second argument. Recompile and rerun the program. What is the output? Why?



```
bash-4.4$ ./ex1
Hello from thread1
Hello from thread2
Hello from main
bash-4.4$ 
```

This is the output because main waits for both thread1 and thread2 functions to complete before continuing on and then exiting whereas before, it would just exit without waiting.

5pts Include your commented code.

```
/*
Introduction to threads in lab 3 of CPRE 308
Basic program to give simple expousure to two thread scernarios
@author Thomas Gaul
*/

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

void* thread1();
void* thread2();

int main (int argc, char *argv[])
{
        pthread_t i1, i2;
        pthread_create(&i1, NULL, (void*)&thread1, NULL); //creates thread 1
        pthread_create(&i2, NULL, (void*)&thread2, NULL); //creates thread 2
        pthread_join(i1, NULL);   //makes main wait for thread1 to finish
        pthread_join(i2, NULL);   //makes main wait for thread2 to finish
        printf("Hello from main\n");
}

void* thread1(){
        sleep(5);//wait 5s
        printf("Hello from thread1\n");
}

void* thread2(){
        sleep(5);//wait 5s
        printf("Hello from thread2\n");
}
```

3.2:

3.2.1:

5 pts Compile and run t1.c, what is the output value of v?
v=0

15 pts Delete the *pthread_mutex_lock* and *pthread_mutex_unlock* statement in both increment and decrement threads. Recompile and rerun t1.c, what is the output value of v? Explain why the output is the same or different.
v=-990 This value is different from 3.2.1 because it locked the variable each time, and it had to wait for it to be unlocked, and the two threads went back and forth, undoing each other. When the locks were deleted, the two threads ran simultaneously, overwriting each other's changes, but the decrement finished later, so its result was all that was left.

3.2.2:

20 pts Include your modified code with your lab submission and comment on what you added or changed.

```
/* t2.c
  synchronize threads through mutex and conditional variable
  To compile use: gcc -o t2 t2.c -lpthread
*/

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

void*   hello();    // define two routines called by threads
void*   world();
void*   again();    // third routine added for a third thread for "again"

/* global variable shared by threads */
pthread_mutex_t        mutex;                    // mutex
pthread_cond_t                  done_hello;     // conditional variable
pthread_cond_t                  done_world;     // conditional variable for world to be
completed
int                             done = 0;        // testing variable
int                     doneWorld = 0;  // testing variable for printing again

int main (int argc, char *argv[]){
   pthread_t     tid_hello, // thread id
                 tid_world,
                 tid_again;
   /*  initialization on mutex and cond variable  */
   pthread_mutex_init(&mutex, NULL);
   pthread_cond_init(&done_hello, NULL);
   pthread_cond_init(&done_world, NULL); // initialized the conditional variable for
the world complete
   pthread_create(&tid_hello, NULL, (void*)&hello, NULL); //thread creation
   pthread_create(&tid_world, NULL, (void*)&world, NULL); //thread creation
   pthread_create(&tid_again, NULL, (void*)&again, NULL); //thread creation for
printing again
   /* main waits for the two threads to finish */
   pthread_join(tid_hello, NULL);
   pthread_join(tid_world, NULL);
   pthread_join(tid_again, NULL); //waits to exit main for again to be printed

   printf("\n");
   return 0;
}

void* hello() {
   pthread_mutex_lock(&mutex);
   printf("hello ");
   fflush(stdout);        // flush buffer to allow instant print out
   done = 1;
   pthread_cond_signal(&done_hello);          // signal world() thread
   pthread_mutex_unlock(&mutex);  // unlocks mutex to allow world to print
   return ;
}


void* world() {
   pthread_mutex_lock(&mutex);

   /* world thread waits until done == 1. */
   while(done == 0)
         pthread_cond_wait(&done_hello, &mutex);

   printf("world ");
   fflush(stdout);
   doneWorld = 1;
   pthread_cond_signal(&done_world); // marks completed world print for again to be
printed
   pthread_mutex_unlock(&mutex); // unlocks mutex

   return ;
}

void* again(){
         pthread_mutex_lock(&mutex);

   /* world thread waits until doneWorld == 1 to make sure it does not sleep forever*/
   while(doneWorld == 0)
         pthread_cond_wait(&done_world, &mutex);

   printf("again");
   fflush(stdout);
   pthread_mutex_unlock(&mutex); // unlocks mutex

   return ;
}
```

3.3:

20pts Include your modified code with your lab submission and comment on what you added or changed.

```
/*
 * Fill in the "producer" function to satisfy the requirements
 * set forth in the lab description.
 */

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

```
#include <time.h>

/*
 * the total number of consumer threads created.
 * each consumer thread consumes one item
 */
#define TOTAL_CONSUMER_THREADS 100

/* This is the number of items produced by the producer each time. */
#define NUM_ITEMS_PER_PRODUCE  10

/*
 * the two functions for the producer and
 * the consumer, respectively
 */
void *producer(void *);
void *consumer(void *);


/********** global variables begin *******/

pthread_mutex_t  mut;
pthread_cond_t          producer_cv;
pthread_cond_t          consumer_cv;
int             supply = 0;  /* inventory remaining */

/*
 * Number of consumer threads that are yet to consume items.  Remember
 * that each consumer thread consumes only one item, so initially, this
 * is set to TOTAL_CONSUMER_THREADS
 */
int  num_cons_remaining = TOTAL_CONSUMER_THREADS;

/************** global variables end ****************************/



int main(int argc, char * argv[])
{
  pthread_t prod_tid;
  pthread_t cons_tid[TOTAL_CONSUMER_THREADS];
  int      thread_index[TOTAL_CONSUMER_THREADS];
  int      i;

  /********** initialize mutex and condition variables ***********/
  pthread_mutex_init(&mut, NULL);
  pthread_cond_init(&producer_cv, NULL);
  pthread_cond_init(&consumer_cv, NULL);
  /***************************************************************/
  /* create producer thread */
  pthread_create(&prod_tid, NULL, producer, NULL);

  /* create consumer thread */
  for (i = 0; i < TOTAL_CONSUMER_THREADS; i++)
  {
    thread_index[i] = i;
    pthread_create(&cons_tid[i], NULL,
                           consumer, (void *)&thread_index[i]);
  }

  /* join all threads */
  pthread_join(prod_tid, NULL);
  for (i = 0; i < TOTAL_CONSUMER_THREADS; i++)
    pthread_join(cons_tid[i], NULL);

  printf("All threads complete\n");

  return 0;
}




/***************** Consumers and Producers *****************/

void *producer(void *arg)
{
  int producer_done = 0;

  while (!producer_done)
  {
//CHANGES MADE HERE
    pthread_mutex_lock(&mut);
    while(supply != 0) //waits for supply to reach 0 to resupply
            pthread_cond_wait(&producer_cv, &mut);

            printf("Resupply\n");
            fflush(stdin);
            supply = supply +NUM_ITEMS_PER_PRODUCE; //update supply
            pthread_cond_signal(&consumer_cv);


            if(num_cons_remaining==0) //no more consumers producer exits
            producer_done =1;
            pthread_mutex_unlock(&mut);
//CHANGES MADE HERE
  }
  return NULL;
```

```
}



void *consumer(void *arg)
{
  int cid = *((int *)arg);

  pthread_mutex_lock(&mut);
  while (supply == 0)
    pthread_cond_wait(&consumer_cv, &mut);

  printf("consumer thread id %d consumes an item\n", cid);
  fflush(stdin);

  supply--;
  if (supply == 0)
    pthread_cond_broadcast(&producer_cv);

  num_cons_remaining--;

  pthread_mutex_unlock(&mut);

  return NULL;
}
```