Name: Nihaal K Zaheer
Section: 2
University ID: 105254206

Lab 3 Report

Summary:

20pts

In this lab we learned about pthreads. We started of understanding the use of pthread_create and join by performing tasks that showed us what would happen if we didn't have these statements in our code.
Next we learnt about mutex and conditional variables, this is something that I found really interesting. I understood that mutex and conditional variables, controls process scheduling and locks the process scheduling so that one function is entirely executed before moving onto the next one. We do this by using mutex_lock and unlock and pthread cond signal, control and broadcast.
Finally we put it all together by simulating a producer-consumer model.

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, we don't see the thread outputs.

When I ran the executable ./ex1, it displayed "Hello From main". The main function did not start the threads, thread1 or thread2. We know this because it did not run the sleep(5).

This happens because we did not have the pthread_join command. This is what starts the thread. It makes the main wait for the 2 threads to finish or terminate.

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-4.2$ ./ex1
Hello from thread1
Hello from thread2
Hello from main
```

The output is as shown above. Thread1 compiles first, it gives the output, the moves on to thread2 after thread1 terminates (pthread_join). Then finally we see the statement “Hello from main” after the 2 threads are terminated

5pts Include your commented code.

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

/* Headers*/
/* define two routines called by threads*/

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


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

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

int main(int argc, char* argv){
  /* thread id or type*/


        pthread_t i1;
        pthread_t i2;

  /* thread creation */
        pthread_create(&i1, NULL, (void*)&thread1, NULL);
        pthread_create(&i2, NULL, (void*)&thread2, NULL);

  /* main waits for the two threads to finish */

        pthread_join(i1, NULL);
        pthread_join(i2, NULL);

        printf("Hello from main \n");

        return 0;
}
```

3.2:

3.2.1:

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

```
bash-4.2$ ./t1
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.

```
bash-4.2$ ./t1
v=-990
```

Output is different, we only get the output for decrement.
This is because we don't use mutex, what mutex_lock() does is blocks the process so that it finishes executing, and then mutex_unlock(), unblocks it once it's done executing entirely.

Since we deleted the mutex lines, the OS doesn't know that it has to complete one process before moving on to the next, and so messes up and only ends up showing decrement() as the final output without taking into account increment().

3.2.2:

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

```c
/* 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();

/* global variable shared by threads */
pthread_mutex_t         mutex;                  // mutex
pthread_cond_t          done_hello;     // conditional variable
pthread_cond_t          done_world;     // conditional variable
int                     done = 0;       // testing variable

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);

    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

    /* main waits for the two threads to finish */
    pthread_join(tid_hello, NULL);
    pthread_join(tid_world, NULL);
    pthread_join(tid_again, NULL);

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

void* hello() {
    usleep(5000);
    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() {
    usleep(2000);
    pthread_mutex_lock(&mutex);

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

```
        pthread_cond_signal(&done_world);       // signal world() thread
}
    printf("world");
    fflush(stdout);
    pthread_mutex_unlock(&mutex); // unlocks mutex

    return ;
}

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

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

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

    return ;
}
```

I added a new function called again(), and also made a new pthread_cond_t called done_world(). I also made a new pthread called tid_again.
Basically what I did was, called a signal within world() called done_world. Now within the new function, again(), I had the pthread_comd_wait that called done_world. This would wait until the world() function would be done executing, but because we had a pthread_cond_wait(&done_hello,&mutex), this would in-turn wait for hello() to finish executing. Because of this it would go in order and give us the required output as follows:

```
bash-4.2$ ./t2
hello world again!
```

3.3:

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

Part of the output:

```
consumer thread id 89 consumes an item
Supply increased by 10
consumer thread id 90 consumes an item
consumer thread id 91 consumes an item
consumer thread id 92 consumes an item
consumer thread id 93 consumes an item
consumer thread id 94 consumes an item
consumer thread id 95 consumes an item
consumer thread id 96 consumes an item
consumer thread id 97 consumes an item
consumer thread id 98 consumes an item
consumer thread id 99 consumes an item
No more consumers
All threads complete
```

I changed the producer() function to perform the needful. I added a mutex lock and unlock, and within ther while loop, I added another while loop which took care of performing required functions when supply was not 0. Within this while loop there was an if statement that checked of num_cons_remaining was 0, and if it was if would break out of the top while loop, and else, supply would be increased by 10.

```c
/*
 * 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 ***************************/


```

```c
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;
}
```

```c
/***************** Consumers and Producers *****************/

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

  while (!producer_done)
  {
  pthread_mutex_lock(&mut);
  while(supply != 0){
    pthread_cond_wait(&producer_cv, &mut);
  }
    if (num_cons_remaining == 0){
      producer_done = 1;
      printf("No more consumers \n");
    }
    else {
        supply += 10;
        printf("Supply increased by 10 \n");
    }
    fflush(stdout);
    pthread_cond_broadcast(&consumer_cv);
  pthread_mutex_unlock(&mut);
  }
  return NULL;
}
```

```
112
113 void *consumer(void *arg)
114 {
115   int cid = *((int *)arg);
116
117   pthread_mutex_lock(&mut);
118   while (supply == 0)
119     pthread_cond_wait(&consumer_cv, &mut);
120
121   printf("consumer thread id %d consumes an item\n", cid);
122   fflush(stdin);
123
124   supply--;
125   if (supply == 0)
126     pthread_cond_broadcast(&producer_cv);
127
128   num_cons_remaining--;
129
130   pthread_mutex_unlock(&mut);
131
132   return NULL;
133 }
```