Name: Khushveen Kaur Umra
Section: 2
University ID: 947005108

Lab 3 Report

Summary:

20pts

In this lab, we experiment programming with multiple threads, and learned about concurrent programming using pthread. I learned how to create and manage multiple threads in C.

3.1) In this step, we learned to code a simple program using pthread, along with the functions pthread_create() and pthread_join().

3.2) In this step, we learned how to synchronize threads that share a common data source. Here we used mutexes and conditional variables as synchronization mechanisms for pthreads.

3.3) Here we learned how to code for program that runs a group of consumers and a single producer in synchronization.

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, one cannot see the threads' output, because it lacks the pthread_join function for both the threads. Since sleep(5) is added in the beginning to both the threads, the main method is terminated before "sleep(5)" is even complete. This is why, one cannot see the threads' output, because the program does not print the thread functions.

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$ Hello from thread1
Hello from thread2
Hello from main
```

We can see the threads' output after adding the statement "pthread_join" for both the threads, because the function "pthread_join" blocks the calling thread until the thread, which is associated with it, is completed. The main function waits for each thread to finish, before it continues and prints the message.

5pts Include your commented code.

```c
/* Author: Khushveen Kaur Umra, kumra@iastate.edu */

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

/* To create two thread functions */

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


int main() {

        pthread_t i1, i2; /* variables of datatype pthread_t */

        /* Functions to create both threads i1 & i2 */

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

        /* Wait for the threads to finish before printing the message */

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

        printf("Hello from main\n");

        return 0;
}

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

void* thread2() {
        sleep(5); /* Prints after 5 seconds */
        printf("Hello from thread2\n");
}
```

3.2:

3.2.1:

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

The output is as follows:

```
bash-4.2$ gcc -c t1.c
bash-4.2$ gcc -o t1 -lpthread  t1.o
bash-4.2$ ./t1 &
[2] 15295
bash-4.2$ 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 &
[2] 16080
bash-4.2$ v=-990
```

The output is different because there is no lock from increase or decrease, as without the pthread_mutex_lock and pthread_mutex_unlock statement, there is no form of thread sychronization. Both the threads run simultaneously, as they depend on the same global variable. This further causes a conflict, and the program prints a value that is not zero, and this case, a value of -990. This happens because, after we remove the statements, there is no way to which thread will be called first, that will either output a value of 0 if it calls decrease thread, or a value of -990, if it calls the increase thread.

3.2.2:

20 pts Include your modified code with your lab submission and comment on what you added and 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;    // added another 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); //added another thread creation statement

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

    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);
    done = 1;
    pthread_cond_signal(&done_world); //signal world() thread
    pthread_mutex_unlock(&mutex); // unlocks mutex

    return ;
}

//Added a new function to print again in a different thread
void* again() {
  pthread_mutex_lock(&mutex);

  // world thread wating until done == 1

  while(done == 0)
    pthread_cond_wait(&done_world, &mutex);

  printf("Again!");
  fflush(stdout);
  pthread_mutex_unlock(&mutex); //unlocks mutex to allow again to print

  return ;
}
```

3.3:

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

The last few lines of the output:

```
Produced 10 more items
consumer thread id 70 consumes an item
consumer thread id 71 consumes an item
consumer thread id 72 consumes an item
consumer thread id 73 consumes an item
consumer thread id 74 consumes an item
consumer thread id 75 consumes an item
consumer thread id 76 consumes an item
consumer thread id 77 consumes an item
consumer thread id 78 consumes an item
consumer thread id 79 consumes an item
Produced 10 more items
consumer thread id 80 consumes an item
consumer thread id 81 consumes an item
consumer thread id 82 consumes an item
consumer thread id 83 consumes an item
consumer thread id 84 consumes an item
consumer thread id 85 consumes an item
consumer thread id 86 consumes an item
consumer thread id 87 consumes an item
consumer thread id 88 consumes an item
consumer thread id 89 consumes an item
Produced 10 more items
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
All threads complete
bash-4.2$
```

CODE IS ON THE NEXT PAGE

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

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

  while (!producer_done)
  {
    /* fill in the code here */

    //Lock for the while loop
    pthread_mutex_lock(&mut);

    //Wait until the cunsumer reaches 10
    while (supply > 0)
      pthread_cond_wait(&producer_cv, &mut);

    //Consumer threads are complete
    if(num_cons_remaining == 0)
      return ;

    //Print 10 more items
    printf("Produced 10 more items \n");
    fflush(stdin);

    //Increase the supply chain
    supply += 10;
    pthread_cond_broadcast(&consumer_cv);

    //Unlocks the mutex
    pthread_mutex_unlock(&mut);
  }
  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;
}
```