Name: Akira DeMoss
Section: M
University ID: 040736696

Lab 3 Report

*Note: Please upload a PDF version of your lab report and make sure all the answers are readable.

Summary:

In this lab I learned more about how threads work. This was especially helpful in learning more about mutex and lock threads, as someone who learns by doing this was the perfect way to get more hands on experience and knowledge! I look forward to applying this knowledge in future labs, projects, and in my on independent coding work.

Lab Questions:

3.1:

12pts 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?

Output
Hello, I am main process.

No, we can't actually see the printing of the threads. The sleep statements make no difference because the threads are never joined.

4pts 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?

Output
Hello, I am thread 1.
Hello, I am thread 2.
Hello, I am main process.

The output is that the threads print first min the order they are joined, and then the main is printed. With sleep still in the functions, we can tell that the main process is blocked while the child threads execute.

4pts Include your commented code.

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

// Thread function prototypes
```

```c
void* thread1();
void* thread2();

void
main()
{
    // Pthread vars
    pthread_t   i1;
    pthread_t   i2;

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

    // Wait for thread completion
    pthread_join(i1, NULL);
    pthread_join(i2, NULL);
    printf("Hello, I am main process.\n");
}

// Function definitions
void* thread1()
{
    sleep(5);
    printf("Hello, I am thread 1.\n");
    return NULL;
}

void* thread2()
{
    sleep(5);
    printf("Hello, I am thread 2.\n");
    return NULL;
}
```

3.2:

3.2.1:

4pts Compile and run t1.c, what is the output value of v?

The output of v is 0.

16pts 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

Basically the two threads can access shared data and they try to change it at the same time. Because the thread scheduling algorithm can swap between threads at any time, you don't know the order in which the threads will attempt to access the shared data. Hence, why the thread lock and unlock are critical in maintaining the integrity of the data.

3.2.2:
20pts Include your modified code with your lab submission and comment on what you added or changed.

Basically I just added a new thread and join for the again line of text to be outputted. I then followed the pattern of using signals, waits, and locks to indicate when it is appropriate for the “again!” print statement to execute.

```
/* 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             done_again = 0; // testing variable again

int main (int argc, char *argv[]){
   pthread_t   tid_hello,  tid_world, tid_again;// thread id

   /*  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() {
   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_again = 1;
    pthread_cond_signal(&done_world);

   pthread_mutex_unlock(&mutex); // unlocks mutex

   return ;
}

void again() {
    pthread_mutex_lock(&mutex);

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


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

    return ;
}
```

3.3:

40pts Include your modified code with your lab submission and comment on what you added or changed.

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

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

 while (!producer_done)
 {
   /* fill in the code here */
   // Lock for the while loop i
   pthread_mutex_lock(&mut);

   // Stop producing if no more consumers
   if(num_cons_remaining < 1){
     producer_done = 1;
     continue;
   }
    // Wait for supply to be diminished before producing again,
produce 10
   while(supply > 0)
     pthread_cond_wait(&producer_cv, &mut);

   supply += 10;

   //call the consumer to work
   pthread_cond_broadcast(&consumer_cv);

   // Unlock the mutex and cycle the loop
  pthread_mutex_unlock(&mut);
 }
 return NULL;
}
```