Name: Sean Hinchee

Section: G

University ID: 068799755

Lab 3 Report

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

Summary:

10pts

Throughout this exercise I learned a fair bit about threading and thread coordination. I am not very familiar with the libpthread style of threading and am more used to Go and Plan 9's libthread model which is more CSP-style. I found the learning experience informative and appreciated the opportunity to get hands on examples with pthread usage.

Overall, I feel that I better understand the usage of mutexes and pthread calls and will apply the knowledge and refinements I gained in this lab to my further cpre 308 projects and assignments.

Lab Questions:

3.1:

6pts 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?

The output without pthread_join() and sleep is:

```
% make
gcc -o ex1 -lpthread ex1.c
% ./ex1
Hello, I am the main process.
%
```

In this case, we do not see the printing of the child threads because the main process exits without ever joining the child threads. The sleep calls do not matter as the child threads are never executed.

2pts 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?

The output with the pthread_join's added:

```
% make
gcc -o ex1 -lpthread ex1.c
% ./ex1
Hello, I am thread 1.
Hello, I am thread 2.
Hello, I am the main process.
%
```

The output is ordered as the threads are executed in a manner which blocks the main process while the child threads execute. That is, main will no return while child threads are running.

2pts Include your commented code.

pthread example:

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

/* function prototypes for threads */
void* thread1();
void* thread2();

void
main()
{
    /* declare variables */
    pthread_t    i1;
    pthread_t    i2;

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

    /* wait for threads to complete, then print */
    pthread_join(i1, NULL);
    pthread_join(i2, NULL);
    printf("Hello, I am the main process.\n");
}

/* function definitions for threads */
void*
thread1()
{
    printf("Hello, I am thread 1.\n");
    return NULL;
}

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

3.2:

3.2.1:

2pts Compile and run t1.c, what is the output value of v?

Output:

```
% gcc -lpthread t1.c
% ./a.out
v=0
%
```

The output value of v is 0.

8pts 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.

Output:

```
% gcc -lpthread t1.c
% ./a.out
v=-990
%
```

The output value of v is -990. The output differs because because a race condition occurs between the two threads. As a result, the two threads will both modify v at the same time, creating a garbled output value.

3.2.2:
10pts Include your modified code with your lab submission and comment on what you added or changed.

The most notable changes below are the addition of further locking variables and a further sequential step to call again() after world() is called. A mutex unlock is not performed by again() as it is not strictly necessary (no other threads will care about the mutex after again() returns).

t2.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 again
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);  // signal again() thread

    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:

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

The only notable changes to the source code below is the producer function, which is thoroughly commented.

t3.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 ****************************/




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

    // Stop producing if there are 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;

    // Awaken the hungering masses
    pthread_cond_broadcast(&consumer_cv);

    // Unlock the mutex and cycle the loop
    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;
}
```