**CSC 345: Home-Work 3:** *Producer Consumer Problem using pthreads*
Date: April 19th, 2016 **(Due April 27th, 2016 @ 8:00 am)**

## Table of Contents



## Objective of Home Work

- Implement functions in C and pass arguments using pointers
- To create multiple processes (threads) using pthreads() in C
- The use of mutex for process synchronization
- To implement the producer-consumer problem for inter-process communication

## Instructions to follows

1. The assignment consists of three parts.
2. You are expected to examine the code provided in each part and create respective make-files and create a report as instructed in each task.
3. The code provided in Part 3 consist of missing C constructs, you are expected to effectively de-bug the code and make changes to it as instructed in the tasks.

**CSC 345: Home-Work 3:** *Producer Consumer Problem using pthreads*
Date: April 19th, 2016 **(Due April 27th, 2016 @ 8:00 am)**

## Part 1: Functions and pointers in C

### a. Passing arguments to functions in C:

Implementation of a program that generates the sum of numbers with a specified offset using functions in C.

CODE 1: Sum-offset.c

```
#include <stdio.h>
#define NUM_LOOPS 500000

long long sum = 0;

void counting_function(int offset)
{
      for(int i=0; i<NUM_LOOPS; i++)
      {
      sum=sum+offset;
      }

}

int main(void)
{
      counting_function(1);
      printf("Sum = %lld\n", sum);
      return 0;
}
```

The following is the make file to execute the above code1.

```
gcc -o a Sum-offset.c
time ./a
```

**Observation:** you are passing an argument called offset from the main program through the function interface.

**Task 1.** Execute the code in Part 1(a) .
Place the output in your report

## b. Passing arguments to functions using pointers in C:

Implementation of a program that generates the sum of numbers with a specified offset but passing it as a pointer.

CODE 2: Sum-offset.c

```
#include <stdio.h>
#define NUM_LOOPS 500000

long long sum = 0;

void counting_function(void *ptr)  //passing a pointer to fn
{
      int offset =*(int *) ptr;    // dereferencing a pointer var
      for(int i=0; i<NUM_LOOPS; i++)
      {
      sum=sum+offset;
      }

}

int main(void)
{
      int offset1 =100;             // creating a variable to be
                                     passed to fn
      counting_function(&offset1);// passing address of variable
      printf("Sum = %lld\n", sum);
      return 0;
}
```

The following is the make file to execute the above code 2.

```
gcc -o a Sum-offset.c
time ./a
```

**Observation:** you are passing an argument called offset1 from the main program through the function interface as a pointer to the counting function.

**Task 2.** Execute the code in Part 1(b).
Place the output in your report

**CSC 345: Home-Work 3:** *Producer Consumer Problem using pthreads*
Date: April 19th, 2016 **(Due April 27th, 2016 @ 8:00 am)**

## Part 2: Threads using pthreads

### a. Creating your first thread using functions and pointers in C:

Implementation of a program that generates the sum of numbers with a specified offset using the threads.

NOTE: pay close attention to the code highlighted in red.

CODE 3: Sum-offset.c

```
#include <stdio.h>
#include <pthread.h>
#define NUM_LOOPS 500000

long long sum = 0;  // a global variable

void* counting_function(void *ptr)
{
      int offset =*(int *) ptr;
      for(int i=0; i<NUM_LOOPS; i++)
      {
      sum=sum+offset;
      }
      pthread_exit(NULL);  // return arguments from thread
}

int main(void)
{
      int offset1 =100;
      pthread_t id1;       // Declare an object of type thread

      pthread_create(&id1, NULL, counting_function, &offset1);
       // Create a thread using pthread_create with the following
       // arguments
       // First argument  -> The thread object id
       // Second argument -> The attributes of the thread
       //                    (Default = NULL)
       // Third argument  -> The function invoked by the thread
       // Fourth argument --> The variable passed into the
       //          function executed by the thread (as a pointer)

      pthread_join(id1, NULL); // Joining of thread

      printf("Sum = %lld\n", sum);
      return 0;
}
```

The following is the make file to execute the above code 3.

```
gcc -o a Sum-offset.c -pthread
time ./a
```

**Observation:** you are passing an argument called offset1 from the main program through the function interface as a pointer to the counting function.

**Task 3.** Execute the code in Part 2(a).
Place the output in your report

### b. Creating multiple threads

Implementation of a program that generates the sum of numbers with a different offsets using multiple threads.

NOTE: pay close attention to the code highlighted in red.

The following is the make file to execute code 4 (following page).

```
gcc -o a Sum-offset.c -pthread
time ./a
```

**Observation:** You have created two treads and are passing an arguments called offset1 (increment) in thread id1 and offset 2 (decrement) in thread id2.

**Task 4.** Execute the code in Part 2(b).
Place the output in your report

CODE 4: Sum-offset.c

```
#include <stdio.h>
#include <pthread.h>
#define NUM_LOOPS 500000

long long sum = 0;  // a global variable

void* counting_function(void *ptr)
{
      int offset =*(int *) ptr;
      for(int i=0; i<NUM_LOOPS; i++)
      { sum=sum+offset; }
      pthread_exit(NULL);
}

int main(void)
{
      int offset1 =1;
      int offset2 = -1; // Declare a second offset
      pthread_t id1, id2; // Declare another object of thread

      // Spawn Treads
      pthread_create(&id1, NULL, counting_function, &offset1);
      pthread_create(&id2, NULL, counting_function, &offset2);
                        // Create your second thread
      // Join Threads
      pthread_join(id1, NULL);
      pthread_join(id2, NULL); // Join the threads back to main
      printf("Sum = %lld\n", sum);
      return 0;
      }
```

### c. Using the mutex synchronization with multi threads

Implementation of a program that implements mutex operations over the critical section of a process while working with multiple threads.

**Observation:** The output of task 4 would not generate consistent results with separate executions. This is because the variable sum is shared between two threads, and therefore is part of the critical section.
The objective of part 2c is to enable the use of mutex operators to handle the critical section between shared processes.

**NOTE:** pay close attention to the code highlighted in red.
The following is the make file to execute the code 5 that follows.

```
gcc -o a Sum-offset.c -pthread
time ./a
```

**Observation:** You have created two treads and are passing an arguments called offset1 (increment) in thread id1 and offset 2 (decrement) in thread id2. Furthermore, you are addressing the race conditions created by the variable sum with the use of the mutex lock and unlock.

**Task 5.** Execute the code in Part 2(c).
Place the output in your report

CODE: Sum-offset.c

```
#include <stdio.h>
#include <pthread.h>
#define NUM_LOOPS 500000

long long sum = 0;  // a global variable

pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;


void* counting_function(void *ptr)
{
      int offset =*(int *) ptr;
      for(int i=0; i<NUM_LOOPS; i++)
      {
            // Start of crtical section
            pthread_mutex_lock(&mutex);
            sum=sum+offset;
            // End of critical section
            pthread_mutex_unlock(&mutex);
      }
      pthread_exit(NULL);  // passing return arguments from thread
}

int main(void)
{
      int offset1 =1;
      int offset2 = -1;
      pthread_t id1, id2;

      // Spawn Threads
      pthread_create(&id1, NULL, counting_function, &offset1);
      pthread_create(&id2, NULL, counting_function, &offset2);

      // Join Threads
      pthread_join(id1, NULL);
      pthread_join(id2, NULL);

      printf("Sum = %lld\n", sum);
      return 0;
      }
```

**CSC 345: Home-Work 3:** *Producer Consumer Problem using pthreads*
Date: April 19th, 2016 **(Due April 27th, 2016 @ 8:00 am)**

## Part 3: The Producer Consumer Problem

The following code 6 is an implementation of the producer consumer problem using a software locking mechanism.

Your tasks here require you to debug the code with the intent of achieving the following task:
**Task 6:** Identifying the critical section
**Task 7:** Identify the software locks and replace them with a simplified mutex lock and unlock.

HINT: The code provided relies heavily on the in and out pointers of the buffer. You should make the code run on a single count variable.

**Place the edited code and the corresponding output in your report**

CODE 6 : `prod_cons_count.c`

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

#define MAXSIZE 100
#define ITERATIONS 1000


int buffer[MAXSIZE]; // buffer
int nextp, nextc; // temporary storage
int count=0;
```

```
void printfunction(void * ptr){
  int count = *(int *) ptr;
 if (count==0) {
    printf("All items produced are consumed by the consumer \n");
  }
  else{
  for (int i=0; i<=count; i=i+1){
    printf("%d, \t",buffer[i]);
  }
  printf("\n");
}
}
```

```
void *producer(void *ptr) {
  int item, flag=0;
  int in = *(int *) ptr;

  do{
    item = (rand()%7)%10;
    flag=flag+1;
    nextp=item;
    buffer[in]=nextp;
    in=((in+1)%MAXSIZE);
    while(count <= MAXSIZE)
    {
      count=count+1;
      printf("Count = %d in produced at Iteration = %d\n", count, flag);
    }
   } while (flag<=ITERATIONS);
  pthread_exit(NULL);

}
```

```
void *consumer(void *ptr) {
  int item, flag=ITERATIONS;
  int out = *(int *) ptr;

  do{
      while (count >0){
      nextc = buffer[out];
      out=(out+1)%MAXSIZE;

      printf("\t\tCount = %d in consumer at Iteration = %d\n", count,
flag);
      count = count-1;
      flag=flag-1;
      }
      if (count <= 0)
      {
      printf("consumer made to wait...faster than producer.\n");
      }
    }while (flag>=0);
  pthread_exit(NULL);
}
```

```
int main(void) {

      int in=0, out=0; //pointers

      pthread_t pro, con;

      // Spawn threads
      pthread_create(&pro, NULL, producer, &count);
      pthread_create(&con, NULL, consumer, &count);

        if (rc1) {
          printf("ERROR; return code from pthread_create() is %d\n", rc1);
          exit(-1);
         }
          if (rc2) {
          printf("ERROR; return code from pthread_create() is %d\n", rc2);
          exit(-1);
         }
      // Wait for the threads to finish
      // Otherwise main might run to the end
      // and kill the entire process when it exits.
      pthread_join(pro, NULL);
      pthread_join(con, NULL);

      printfunction(&count);



}
```

The following is the make file to execute the above code.

```
gcc -o a2 prod_cons_count.c –pthread
time ./a2
```