西北工業大學

NORTHWESTERN POLYTECHNICAL UNIVERSITY

Lab I Report

Report Subject: OS Experiment - Lab 5

Student ID : 2019380141

Student Name : ABID ALI

Experiment Name : **Synchronization**

Objective:

The Pthread library offers the *pthread_mutex_t* data type, which is much like a binary semaphore and therefore somewhat of limited utility in the solution of synchronization problems. Fortunately, POSIX gives you the more general-purpose semaphore in the *sem_t* data data type. In this lab, you will learn to work with these two mechanisms for thread synchronization as you implement a solution to the bounded-buffer problem. Including:

- Learn to work with Linux and Pthread synchronization mechanisms.
- Practice the critical-section problem
- examine several classical synchronization problems

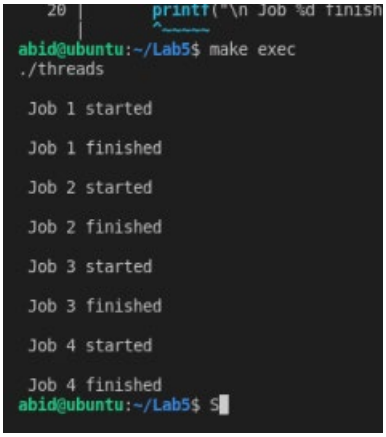
Equipment:

VMWare with Ubuntu Linux

Methodology:

Experiment 1: Thread Synchronization Problems

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

pthread_t tid[2];
sem_t m;
int counter;
void* doSomeThing(void *arg)
{

   unsigned long i = 0;
   sem_wait(&m);
   counter += 1;

     printf("\n Job %d started\n", counter);

   for(i=0; i<1000;i++);

   //sem_wait (&m);

   printf("\n Job %d finished\n", counter);
   sem_post(&m);

   return NULL;
}
int main(void)

{
   int i = 0;
   int err;
   sem_init(&m,0,1);
   while(counter < 2)

   {
      err = pthread_create(&(tid[i]), NULL, &doSomeThing, NULL);
      if (err != 0)
         printf("\ncan't create thread :[%s]", strerror(err));

      i++;
   }
   pthread_join(tid[0], NULL);
   pthread_join(tid[1], NULL);
   return 0;
}
```



1. Do you think that the counter increase correctly? If not, what is wrong? And please increase the counter correctly using multithreads

Solution:

The counter was not increasing correctly, we can see that the output was very random.After modifying the code ,the jobs are starting at the right time.
Modification was
Declaring m as our semaphore sem_t m;
Then, we initialize sem_init(&m,0,1);
We can see that, job 1 started and job 1 finished and job 2 started and so on.
Therefore,the code is running correctly.

2. if Thread 1 must run before Thread 2, how do you do?

Solution:

At first ,we initialize the semaphore = 0 that means we are setting some constrains so that we can make the Thread 1 run before Thread 2.
sem_wait(&m);
Thread 2 will wait for thread 1 to execute.

Experiment 2: The Bounded-Buffer Problem:

Code:

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include "buffer.h"

/* buffer.h */
const int max  = 5; // this is just my constant variable
//int buffer[5]; // this is my buffer, for storing items
int size = 0;
int In  = 0;  // this is for keeping track where at in the buffer is the next item to put in
int Out = 0; // this is for keeping track where at in the buffer is the next item to take out
int n = 0; // keeping track how many are in the buffer
// mutex lock
sem_t  mutex;  // my mutex lock for keeping track who is in the buffer
// counting semaphore
sem_t  empty; // this is my variable for checking if it is empty
sem_t  full;  // this is for keeping track if the buffer is full

// this is my consumer thread, where it will call to the remove method to get the item, and display
it
void * Consumer(void * i)
{
   int item = 0; // storing temporary item
   while (1)
   {
      int sleepTime = rand() % 3 + 1; // generate random number for sleep
      /* sleep for a random period of time */
      printf ("Consumer thread %ld sleeping for %d seconds.\n", (long) i, sleepTime);
      sleep(sleepTime); // telling the thread to go to sleep for sleepTime
      // this if statement is for check if it actually get to remove the item
      if (remove_Item(& item) < 0)
      {
         printf ("Consumer error\n");
      }
      else
      {
         printf ("Consumer thread %ld removed value %d.\n", (long) i , item);
      }
   }
}
```

```
// this is my producer thread, where it will generate a number, and it will call to the insert method to insert
// the item that it just generate.
void * Producer(void * i)
{
   int item = 0;  // this is my temporary storage for using to put as a parameter when calling insert method
   while(1)
   {
      /* generate random number */
      int sleepTime = rand() % 3 + 1; // generate some random number
      /* sleep for random amount of time */
      printf ("Producer thread %ld sleeping for %d seconds.\n", (long) i, sleepTime);
      sleep(sleepTime); // thread is going to sleep
      /* generate a random number - this is the producer's product */
      item = rand() % 100 + 1; // generate some number to put in the buffer, and store it in the item
      if (insert_Item(item) < 0) // this if statement is for checking if the producer actually get
      {                          // to insert the item
         printf ("Producer error\n");
      }
      else
      {
         printf ("Producer thread %ld inserted value %d.\n", (long) i , item);
      }
   }
}
// this is my insert method, it will get the item in the parameter, and it will use to store it in the buffer if it not lock
int insert_Item(int items)
{
   int i = 0;  // loop counter
   int check = 0; // for checking if the item got remove from the buffer
   int out = 0;  // running total for how many available space left are in the buffer
   int count = 0;  // running total for how many item are in the buffer
   do
   {
      /* produce an item in next produced */
      sem_wait(&empty); // lock if buffer is empty
      sem_wait(&mutex); // lock if there is a process is in the buffer
      /* add next produced to the buffer */
      if (n == 0) // this is for checking if the buffer is empty. if so, then reset the In, and Out so it won't
      {          // just store in different slot when when the buffer is empty.
         In = 0;
         Out = 0;
      }
```

```
        printf ("Insert_item inserted item %d at position %d\n",items, In);
        buffer[In] = items; // putting the item in the buffer
        In = (In + 1) % max; // this is for setting the next empty slot in the buffer
        check++;
        n++;
        for (i = 0; i < 5; i++) // this for loop is for checking the buffer for empty, and getting the item
to dislpay
        {
           if (buffer[i] == (int)NULL)
           {
              printf ("[empty]");
              out++;
           }
           else if (buffer[i] != (int)NULL)
           {
              printf ("[%d]", buffer[i]);
              count++;
           }
        }
        printf ("in = %d, out = %d.\n", count, out);
        i = 0;   // reset the value for next thread
        count = 0;
        out = 0;
        // release the lock for the next thead to come in
        sem_post(&mutex);
        sem_post(&full);
        // returning the value that i get from above
        if ( check == 0)
        {
           return -1;
        }
        else
        {
           check = 0;
           return 0;
        }
    }while (1);
}
// this is my remove method, it would get the item from the buffer, and return it to the consumer,
so it can display the item to the user
int remove_Item(int * items)
{
    int i = 0;  // loop counter
    int out = 0;  // running total for how many available space left are in the buffer
    int count = 0; // running total for how many item are in the buffer
    int check = 0;  // for checking if the item got remove from the buffer
```

```
    do
    {   // locking if it is full, and setting mutex
        sem_wait(&full);
        sem_wait(&mutex);
        /* remove an item from buffer to next consumed */
        printf ("Remove_item removed item %d at position %d\n",*items, Out);
        *items = buffer[Out];  // getting the item from the buffer
        buffer[Out] = (int)NULL; // setting the item that took out to NULL, reset for the next item
        Out = (Out + 1) % max; // settitng the Out to the next available space which one up
        check++; // increment check since it already remove
        n--; // this is keeping count how many are in the buffer
        // this for loop is for checking for item in the buffer, and display them
        for (i = 0; i < 5; i++)
        {
            // this if statement is checking if it is empty
            if (buffer[i] == (int)NULL)
            {
                printf ("[empty]");
                out++;
            } // so if the buffer is not empty then it will run this if statement
            else if (buffer[i] != (int)NULL)
            {
                printf ("[%d]", buffer[i]);
                count++;
            }
        } // this will print out the result getting from the for loop. Which i use out for empty, and count
for
        // item in the buffer
        printf ("in = %d, out = %d.\n", count, out);
        i = 0; // resetting the counter value to 0
        count = 0; // resetting the running total of item in the buffer value to 0
        out = 0; // resetting the running total for empty to 0
        sem_post(&mutex); // release the lock for mutex
        sem_post(&empty); // release the lock for empty
        // this will return the result from removing
        if ( check == 0)
        {
            return -1;
        }
        else
        {
            check = 0;
            return 0;
        }
    }while (1);
}
```

/* 1. Command line arguments argv[1],argv[2],argv[3] */

```
int main(int argc, char *argv[])
{
   // this is for storing value return from creating pthread
   int producers, consumers;
   // this is use to store the amount of producer, and consumer
   int run = 0, numProducer = 0, numConsumer = 0;

   // doing if statment to check to make sure the user give the right format that it suppose to be
   if (argc != 4)
   {
      printf ("Sorry that's not the right format.\n");
      printf ("Please use this format ./name (int)seconds (int)producer (int)consumer.\n");
      exit (1);
   }
   else
   {
      // if the top if statement is false, then the else will run, and turing the value of string into int.
      run = atoi(argv[1]);  // how long the main will sleep
      numProducer = atoi(argv[2]); // how many produer to make
      numConsumer = atoi(argv[3]);  // how many consumer to make
   }
```

/* 2. Initialize buffer */

```
   // creating the link to the pthread
   pthread_t producerT[numProducer];
   pthread_t consumerT[numConsumer];

   printf ("Main thread beginning.\n");

   // initialize the semaphore for full, empty, and mutex
   // counting semaphore for full, and empty
   int em = sem_init(&empty, 0, max); // MAX buffers are empty to begin with...
   int ful = sem_init(&full, 0, 0);    // ... and 0 are full

   // mutex locks
         int mut = sem_init(&mutex, 0, 1);   // mutex = 1 because it is a lock (NEW LINE)

   // setting the buffer to null so easier to do if statement to check for empty
   int i = 0;  // loop counter
   for (i = 0; i < max; i++)
   {
      buffer[i] = (int)NULL;
   }
```

```
/// this is for checking if the initialization of the mutex, full, and empty is fail
if (em == -1 || ful == -1 || mut == -1)
{
   printf("Fail to initialize the semaphore.\n");
   exit(1); // this would exit if it fail
}
```

/* 3. Create producer threads. */

```
// creating the producer threads by doing for loop
for (i = 0; i < numProducer; i++)
{
   printf ("Creating producer thread with id %d.\n", i);
   fflush(stdout); // want the OS to flush the statement above
   producers = pthread_create(&producerT[i], NULL, Producer, (void*)(long)i);
   // this is for checking if the pthread is fail
   if (producers)
   {
      printf ("Error:unable to create thread, %d.\n", producers);
      exit(1); // this would exit if it fail
   }
}
```

/* 4. Create consumer threads. */

```
// creating the consumer thread by doing for loop
for (i = 0; i < numConsumer; i++)
{
   printf ("Creating consumer thread with id %d.\n", i);
   fflush(stdout);  // want the OS to flush the statement above
   consumers = pthread_create(&consumerT[i], NULL , Consumer, (void*)(long)i);
   // this is for checking if the pthread is fail
   if (consumers)
   {
      printf ("Error:unable to create thread, %d.\n", consumers);
      exit(1);  // this would exit if it fail
   }
}
```

/* 5. Sleep. */

```
printf ("Main thread sleeping for %d seconds.\n", run);
// main sleep for run time
sleep(run);
```

/* 6. Kill threads and exit. */

```
// killing all threads, and exit
```

```
    for (i = 0; i < numProducer; i++)
    {   // killing the producer thread
        pthread_cancel(producerT[i]);
    }
    for (i = 0; i < numConsumer; i++)
    {   // killing the consumer thread
        pthread_cancel(consumerT[i]);
    }
    printf ("Main thread exiting.\n");
    exit (1);
}
```

1) The running command should be the following.

Prod_com [num_prod] [num_cons] [sleep_time]

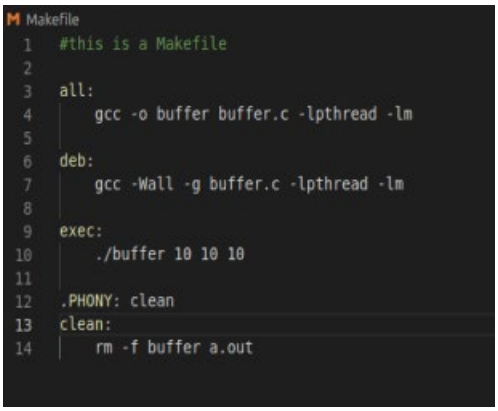
where:

- [num_prod] is the number of producer threads
- [num_cons] is the number of consumer threads

[sleep_time] is the number of seconds to sleep before the process terminates

All the requirements were fulfilled correctly.

2)Modify your makefile to compile it.



3)Compile and run the resulting program.

The result I found is given in text file that is output.txt

Experiment 3: Dining Philosophers

The Philosopher threads request chopsticks by doing two successive locks: one for the chopstick on the left and one for chopstick on the right. Once a Philosopher passes through the wait on a mutex, it is assumed to have picked up the corresponding chopstick. Since a Philosopher must pick up both the left and the right chopsticks in order to be able to eat, we made the "eating" section as the critical section of the Philosopher 's code.

- You must have an array chopsticks[] of five mutexes, which is visible to all Philosopher threads and correctly initialized.

```
pthread_mutex_t chopsticks[5];
unsigned int seeds[5] = {100, 101, 102, 103, 104};
```

- Modify your Philosopher thread so that it works with the chopsticks[] array of mutexes.

```
void * Philosopher (void *id){
  int ID = (int) id;
  while(1){
    if (ID < (ID+1)%5){ // true for every philosopher except last one
      printf("Philosopher %d is thinking.\n", ID);
      wait(2, ID);
      printf("Philosopher %d is hungry.\n", ID);
      pthread_mutex_lock(&chopsticks[ID]); // attempt to use left chopstick
      printf("Philosopher %d is picking up chopstick %d\n", ID, ID);
      //wait(1000, ID);
      pthread_mutex_lock(&chopsticks[(ID+1)%5]); //attempt to use right chopstick
      printf("Philosopher %d is picking up chopstick %d\n", ID, (ID+1)%5);
      printf("Philosopher %d starting to eat.\n", ID);
      wait(1, ID);
      printf("Philosopher %d is done eating.\n", ID);
      pthread_mutex_unlock(&chopsticks[(ID+1)%5]); // release right chopstick
      printf("Philosopher %d is putting down chopstick %d\n", ID, (ID+1)%5);
      pthread_mutex_unlock(&chopsticks[ID]); //release left chopstick
      printf("Philosopher %d is putting down chopstick %d\n", ID, ID);

    }
    else{ // ID > (ID+1)%5
      printf("Philosopher %d is thinking.\n", ID);
      wait(2, ID);
      printf("Philosopher %d is hungry.\n", ID);
      pthread_mutex_lock(&chopsticks[(ID+1)%5]); //attempt to use right chopstick
      printf("Philosopher %d is picking up chopstick %d\n", ID, (ID+1)%5);
      //wait(1000, ID);
      pthread_mutex_lock(&chopsticks[ID]); // attempt to use left chopstick
      printf("Philosopher %d is picking up chopstick %d\n", ID, ID);
      printf("Philosopher %d starting to eat.\n", ID);
      wait(1, ID);
      printf("Philosopher %d is done eating.\n", ID);
      pthread_mutex_unlock(&chopsticks[ID]); //release left chopstick
      printf("Philosopher %d is putting down chopstick %d\n", ID, ID);
      pthread_mutex_unlock(&chopsticks[(ID+1)%5]); // release right chopstick
      printf("Philosopher %d is putting down chopstick %d\n", ID, (ID+1)%5);
    }
  }
```

- Each Philosopher thread will make calls to pthread_mutex_lock and pthread_mutex_unlock to simulate picking up and putting down chopsticks. You should use the following scheme for numbering the chopsticks: the chopstick to the left of Philosopher i is numbered i, while the chopstick to the right is numbered (i+1)%5 (remember that Philosopher 0 is to the "right" of Philosopher 4).

Take right chopstick and pthread_mutex_lock:
pthread_mutex_lock(&chopsticks[(ID+1)%5]);

Release right chopstick and pthread_mutex_unlock:
pthread_mutex_unlock(&chopsticks[(ID+1)%5]);

1) Do you think that this solution is good? If not, what is wrong?

Solution:

We can solve the Dining Philosophers problem in many ways
I think this is one of the solution that can be presented for Dining Philosophers.

Case :
We are using even person rule,two person picking up two chopstick at the same time.
We can see two person eating at the same time others are waiting / thinking.

The problem I saw that,a philosopher after eating then wanted to eat again after few turn without waiting for all of the philosopher to finish their eating turn once.

OS schedule this turn randomly so they demand about eating randomly.
I make them in Synchronization system as much possible.

We can solve the Dining Philosophers problem in many ways.I choose this process because it's easier to understand and implement.

2) Can you modify this solution to meet the requirements? And use semaphores to implement it.

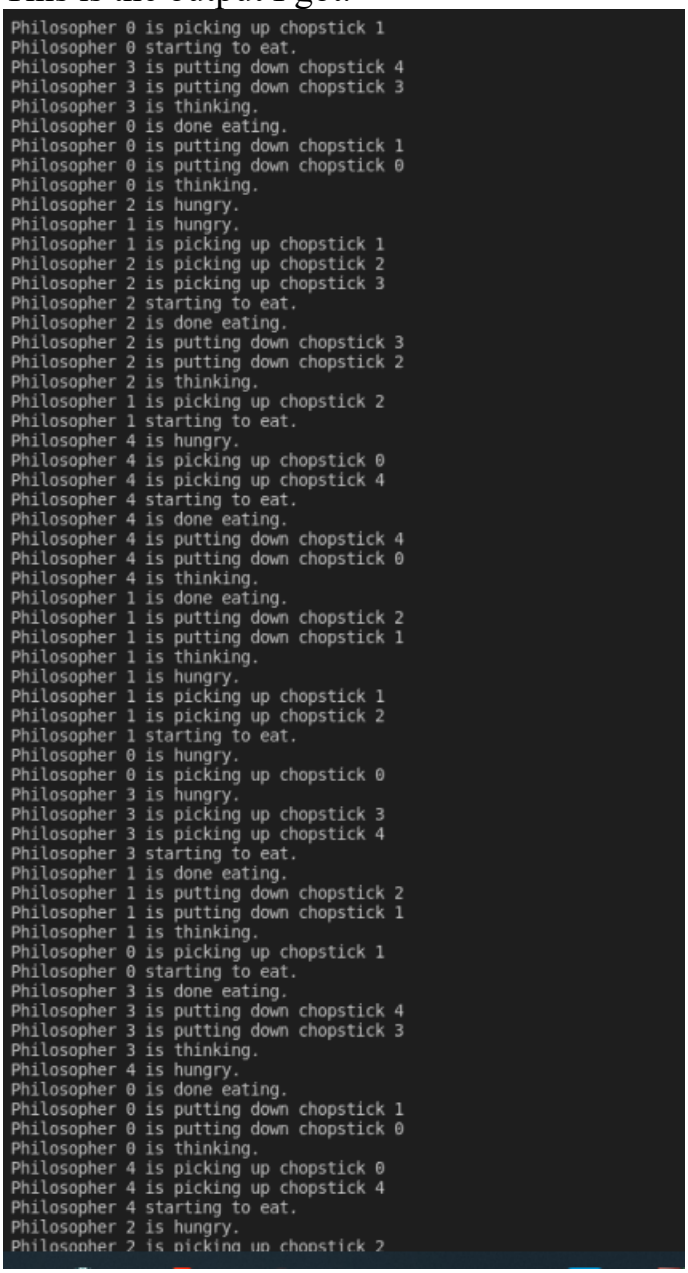
Solution:

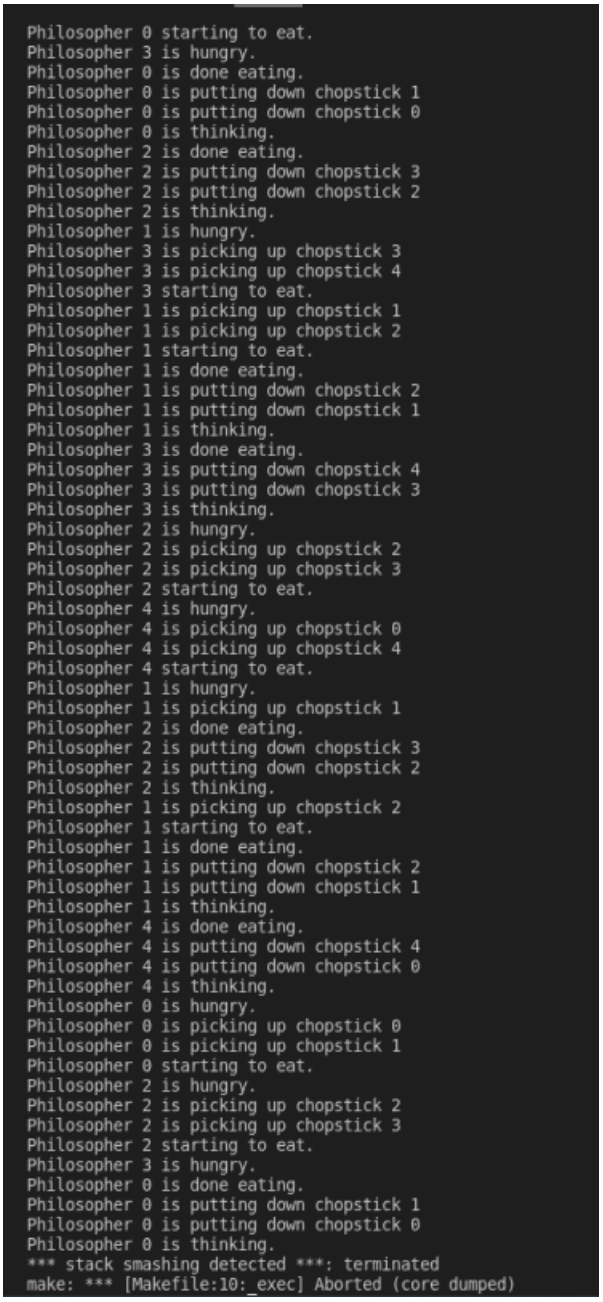
I already used semaphores to solve the problem.But,we can use other cases of Dining Philosophers problem the solution that I got.That is reasonable and clear to understand.
I used semaphores in the program.One of those is given below:
pthread_mutex_t chopsticks[5];

I used semaphores as much possible,in this problem it's hard to control the threads to execute sequentially all the time.

We can see that,if we look at the code that at file 3.There was use of semaphores to control the thread as much possible.

This is the output I got.





Solution:

To solve those problems I looked for information in internet. In order to understand some questions and procedure I also asked the teacher to help me understand them. And provided instructions helped to solve some of my errors during the experiment.

Problems:

The problem that I faced during was how to use different keywords and functions of semaphores and how to implement.I was having problem with synchronization because the output was coming randomly and hard to control the execution of different thread.

Conclusion:

At the beginning, I was unfamiliar with semaphores, synchronization and how to implement it. Gradually, reading lot of article and reading teachers ppt then I solved those problem one by one. In this experiment ,small helps and suggestions from the teacher was very helpful that saved my time.I enjoyed the practical and learned lot of interesting things.

Attachments:

1) ABID ALI_2019380141_OS(Lab 5).docx
2) ABID ALI_2019380141_OS(Lab 5).pdf
3)Code_ABID ALI_2019380141(Lab-5).zip