**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>

pthread\_t tid[2];

int counter;

void\* doSomeThing(void \*arg)

{

unsigned long i = 0;

counter += 1;

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

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

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

return NULL;

}

int main(void)

{

int i = 0;

int err;

while(count < 2)

{

err = pthread\_create(&(tid[i]), NULL, &doSomeThing, NULL);

if (err != 0)

printf("\ncan't create thread :[%s]", strerror(err));

i++;

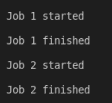
}

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 output was random at the beginning if we use semaphore .

Then,it works correctly.

A semaphore is a variable or abstract data type used to control access to a common resource by multiple threads and avoid critical section problems in a concurrent system such as a multitasking operating system.

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

**Solution:**

We call the wait function by using semaphore.

We set restriction by using wait function.Then,thread 2 will wait for thread 1

**Experiment 2: The Bounded-Buffer Problem:**

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#define TRUE 1

typedef int buffer\_item;

#define BUFFER\_SIZE 5

buffer\_item START\_NUMBER;

buffer\_item buffer[BUFFER\_SIZE];

pthread\_mutex\_t mutex;

sem\_t empty;

sem\_t full;

int insertPointer = 0, removePointer = 0;

int insert\_item(buffer\_item item);

int remove\_item(buffer\_item \*item);

void \*producer(void \*param);

void \*consumer(void \*param);

int insert\_item(buffer\_item item)

{

buffer[insertPointer] = item;

insertPointer = (insertPointer + 1) % BUFFER\_SIZE;

return 0;

}

int remove\_item(buffer\_item \*item)

{

\*item = buffer[removePointer];

removePointer = (removePointer + 1) % BUFFER\_SIZE;

return 0;

}

void \*producer(void \*param)

{

buffer\_item item;

while(TRUE) {

sleep(2);

sem\_wait(&full);

pthread\_mutex\_lock(&mutex);

item = START\_NUMBER++;

insert\_item(item);

printf("Producer %u produced %d \n", (unsigned int)pthread\_self(), item);

pthread\_mutex\_unlock(&mutex);

sem\_post(&empty);

}

}

void \*consumer(void \*param)

{

buffer\_item item;

while(TRUE){

sleep(2);

sem\_wait(&empty);

pthread\_mutex\_lock(&mutex);

remove\_item(&item);

printf("Consumer %u consumed %d \n", (unsigned int)pthread\_self(), item);

pthread\_mutex\_unlock(&mutex);

sem\_post(&full);

}

}

int main(int argc, char \*argv[])

{

int sleepTime, producerThreads, consumerThreads;

int i, j;

if(argc != 5)

{

fprintf(stderr, "Useage: <sleep time> <producer threads> <consumer threads> <start number>\n");

return -1;

}

sleepTime = atoi(argv[1]);

producerThreads = atoi(argv[2]);

consumerThreads = atoi(argv[3]);

START\_NUMBER = atoi(argv[4]);

pthread\_mutex\_init(&mutex, NULL);

sem\_init(&full, 0, BUFFER\_SIZE);

sem\_init(&empty, 0, 0);

pthread\_t pid, cid;

for(i = 0; i < producerThreads; i++){

pthread\_create(&pid,NULL,&producer,NULL);

}

for(j = 0; j < consumerThreads; j++){

pthread\_create(&cid,NULL,&consumer,NULL);

}

sleep(sleepTime);

return 0;

}