COEN 177: Operating Systems
Lab assignment 5: Synchronization using semaphores, lock, and condition variables

Objectives

1. To use semaphores, lock, and condition variables for synchronization
2. To develop a C program to solve the producer – consumer problem

Guidelines

You have learned in class that threads run concurrently and when they read/write to a shared memory, the program behavior is undefined. This is due to the fact the CPU scheduler switches rapidly between threads to provide concurrent execution. One thread may only partially complete execution before another thread is scheduled. Therefore, a thread may be interrupted at any point in its instruction stream, and the CPU may be assigned to execute instructions of another thread, and so the thread schedule is non-deterministic and the resulting output is non-reproducible. To control the non-deterministic and non-reproducible behavior of multi-threaded programs, synchronization is required.

Each thread has a segment of code that involves data sharing with one or more threads. This code segment is referred to as a critical section. Synchronization imposes a rule that when one thread is executing in its critical section, no other thread is allowed to execute in its critical section. Each thread must request permission to enter its critical section, formally defined as the entry section. When a thread completes execution in the critical section, it leaves through an exit section to the remaining code of the program. The general structure of synchronization is therefore defined as follows:

```
do {
   entry section
      critical section
   exit section
   remainder section
  } while (1);
```

Variety of synchronization tools exit. In this lab, semaphores, mutex lock, and condition variables are used for demonstration. A semaphore is considered a generalized lock and it supports two operations:

- P(): an atomic operation that waits for semaphore to become positive, then decrements it by 1. This operation is referred to as wait() operation
- V(): an atomic operation that increments the semaphore by 1, waking up a waiting P, if any. This operation is referred to as signal() operation.

P() stands for “proberen” (to test) and V() stands for “verhogen” (to increment) in Dutch. Linux provides a high-level APIs for semaphores in the <semaphore.h> library:

```
sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);
int sem_destroy(sem_t *sem);
```

Note: MacOs does not support sem_init and sem_destroy (unnamed semaphores). If you are using MacOS, use a named semaphore with sem_open, and sem_unlink as follows:

```
sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int value);
int sem_unlink(const char *name);
```

Mutex lock is a synchronization variable, if one thread holds, no other thread can hold it. A lock has two operations: lock (acquire) and unlock (release), Linux provides a high-levl API for condition variables in the <pthread.h> library:

```
pthread_mutex_t lock; //Declare a lock
pthread_mutex_init(&lock, NULL); //Create a lock
pthread_mutex_lock(&lock); //lock acquire
pthread_mutex_unlock(&lock); //lock release
pthread_mutex_destroy(&mutex); // delete lock
```

Condition variables provide another way for threads to synchronize. They allow threads to synchronize based upon the actual value of data (note: mutex implement synchronization by controlling thread access to data)

A condition variable is a synchronization object that lets a thread efficiently wait for a change to shared state that is protected by a lock. A condition variable is designed to work in conjunction with locks. Linux provides a high-levl API for condition variables in the <pthread.h> library:

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
```

C Program with semaphores

In lab 4, theadHello.c program was demonstrated. In this lab, the program is implemented with semaphores. Demonstrate each of the following steps to the TA to get a grade on this part of the lab assignment.

Step 1. Download the threadSync.c program from Camino, then compile and run several times. The comment at the top of program explains how to compile and run the program.

Explain what happens when you run the threadSync.c program? How does this program differ from theadHello.c program.

Step 2. Modify threadSync.c in Step1 using mutex Locks.

Producer – Consumer as a classical problem of synchronization

Step 3. Write a program that solves the producer - consumer problem using semaphores. You may use the following pseudo code for implementation.

```
//Shared data: semaphore full, empty, mutex;
//pool of n buffers, each can hold one item
//mutex provides mutual exclusion to the buffer pool
//empty and full count the number of empty and full buffers
//Initially: full = 0, empty = n, mutex = 1

//Producer thread
do {
   ...
   produce next item
    ...
   wait(empty);
   wait(mutex);
    ...
   add the item to buffer
    ...
   signal(mutex);
   signal(full);
} while (1);

//Consumer thread
do {
   wait(full)
   wait(mutex);
    ...
   remove next item from buffer
    ...
   signal(mutex);
   signal(empty);
    ...
```

```
    consume the item
     …
} while (1);
```

Step 4. Write a program that solves the producer - consumer problem using condition variables. You may use the following pseudo code for implementation.

```
//Producer thread
do {
   …
   produce next item
    …
   lock(mutex);
   while (buffer is full)
         condV.wait(empty, mutex);
    …
   add the item to buffer
    …
   condV.signal(full);
   unlock(mutex);
} while (1);

//Consumer thread
do {
   lock(mutex)
   while (buffer is empty)
         condV.wait(full, mutex)
    …
   remove next item from buffer
    …
   condV.signal(empty);
   unlock(mutex);
    …
   consume the item
    …
} while (1);
```

Requirements to complete the lab

1. Show the TA correct execution of the C programs.
2. Submit your answers to questions, observations, and notes as .txt file and upload to Camino
3. Submit the source code for all your programs as .c file(s) and upload to Camino.

Be sure to retain copies of your .c and .txt files. You will want these for study purposes and to resolve any grading questions (should they arise)

Please start each program/ text with a descriptive block that includes minimally the following information:

```
# Name: <your name>
# Date: <date> (the day you have lab)
# Title: Lab1 – task
# Description: This program computes … <you should
# complete an appropriate description here.>
```