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

Objectives

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

Guidelines

You have learned in class that multiple threads may run concurrently, and that when multiple threads access the same data concurrently, the program behavior may become undefined (in cases where at least one thread is writing to that memory location). This is due to the fact the CPU scheduler switches rapidly between threads to provide apparently concurrent execution--in fact, a thread may be interrupted at any point in its instruction stream as determined by the thread scheduler. Synchronization mechanisms are thus required to control the behavior of multi-threaded programs which may write to shared memory.

Each thread has one or more segments of code that involve data sharing with one or more other threads. These code segments are referred to as critical sections. Synchronization mechanisms impose the rule that when one thread is executing in a critical section, no other thread can execute code within a corresponding critical section. To implement this, each thread must request permission to enter its critical section using an entry section. Similarly, when a thread completes execution in the critical section, it must leave through an exit section before executing any other code. A pseudocode example of the usage of synchronization follows:

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

A variety of synchronization tools exist. In this lab, we will be using semaphores, mutex locks, and condition variables. A semaphore is generalization of a mutex lock, supporting 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_t sem;
sem_init(&sem, 0, unsigned int value);
sem_wait(&sem);
sem_post(&sem);
sem_destroy(&sem);
```

Note: MacOS does not support sem_init and sem_destroy as they operate on unnamed semaphores. If you are using MacOS, create a named semaphore with sem_open, and sem_unlink instead. Upon failure, sem_open returns SEM_FAILED. Dereferencing SEM_FAILED causes undefined behavior, generally a segmentation fault.

```
sem_ptr = sem_open("sem_name", O_CREAT, 0644, unsigned int value);
sem_unlink("sem_name");
```

A mutex lock is a synchronization variable that behaves like a semaphore restricted to values of 0 and 1. Once one thread locks a mutex, no other thread can lock that mutex until the first thread unlocks it. Linux provides a high-level API for mutex locks in the <pthread.h> library:

```
pthread_mutex_t lock;
pthread_mutex_init(&lock, NULL);
pthread_mutex_lock(&lock);
pthread_mutex_unlock(&lock);
pthread_mutex_destroy(&lock);
```

Condition variables are used in conjunction with locks. A condition variable lets a thread efficiently wait for a change to a shared state that is protected by a lock by telling the OS that the thread is blocked until another thread signals it. Linux provides a high-level API for condition variables in the <pthread.h> library:

```
pthread_cond_t cond;
pthread_cond_init(&cond, NULL);
pthread_cond_wait(&cond,&mutex);
pthread_cond_signal(&cond);
pthread_cond_destroy(&cond);
```

C Program with semaphores

In lab 4, we demonstrated the theadHello.c program. In this lab, the program is reimplemented 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 the threadHello.c program?

Step 2. Modify threadSync.c from Step 1 to use mutex Locks.

Producer – Consumer as a classical problem of synchronization

Step 3. Write a program that solves the producer-consumer problem using semaphores. Use the following pseudocode.

```
//Shared data: semaphores called "full", "empty", and "lock"
//Buffer to hold n items
//"lock" provides mutual exclusion to the buffer pool
//"empty" and "full" count the number of empty and full slots in the buffer
//Initially: full = 0, empty = n, lock = 1

//Producer thread
do {
   …
   "produce" next item (print)
    …
   wait(empty);
   wait(lock);
    …
   add item to buffer
    …
   signal(lock);
   signal(full);
} while (1);

//Consumer thread
do {
   wait(full)
   wait(lock);
    …
   remove next item from buffer
    …
   signal(lock);
   signal(empty);
    …
   "consume" the item (print)
    …
} while (1);
```

Step 4. Write a program that solves the producer-consumer problem using a mutex lock and condition variables. Use the following pseudocode.

```
//New shared data: condition variables called "full" and "empty", mutex lock called "lock"
//Variable to keep track of how full the buffer is

//Producer thread
do {
   …
   "produce" next item (print)
    …
   lock(lock);
   while (buffer is full)
          condV.wait(empty, lock);
    …
```

```
    add the item to buffer
     …
    condV.signal(full);
    unlock(lock);
} while (1);

//Consumer thread
do {
    lock(lock)
    while (buffer is empty)
            condV.wait(full, lock)
     …
    remove next item from buffer
     …
    condV.signal(empty);
    unlock(lock);
     …
    "consume" the item (print)
     …
} 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 a .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 file with a descriptive block that includes at minimum the following information:

```
//Name:
//Date:
//Title: Lab5 –
//Description:
```