CSEN 177: Operating Systems
Lab 5: Synchronization using semaphores, mutex locks, and condition variables

Objectives

1. To use semaphores, mutex locks, and condition variables for synchronization
2. To develop a C program to coordinate the access of producer and consumer threads to a shared buffer

Guidelines

You have learned in class that threads run concurrently, and when they read and write concurrently to shared memory, the program's behavior is undefined. This is because the CPU scheduler switches rapidly between threads to provide the appearance of concurrent execution. One thread may only partially execute before another thread is scheduled. Therefore, a thread may be interrupted at any point in its instruction stream, and the CPU may be reassigned to execute the instructions of another thread. As the thread schedule is non-deterministic, the resulting output is not reproducible. Synchronization is therefore required to prevent this non-deterministic behavior in multi-threaded programs.

Each thread has some segment of code that involves data sharing with one or more other threads. This code segment is referred to as a critical section. Synchronization imposes a rule that for as long as one thread is executing within its own critical section, no other thread can execute within their corresponding critical section. To enable this, each thread must first request permission to enter its critical section. This is formally defined as the entry section. When a thread completes execution in the critical section, it must leave through an exit section to signal that it is no longer within its critical section to reach the remaining code of the program. The general structure of synchronization is thus defined as follows:

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

A variety of synchronization tools exist for this purpose. This lab uses semaphores, mutex locks, and condition variables. A semaphore is considered a generalized lock, and it supports two operations:

- P(): an atomic (indivisible) operation that waits for the semaphore to become positive, then decrements it by 1. This operation is referred to as wait() operation
- V(): an atomic (indivisible) 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; //Declare a semaphore
int sem_init(&sem, 0, unsigned int value); //Initialize a semaphore at value
int sem_wait(&sem); //Wait for a semaphore's value to be greater than 0, then decrement
int sem_post(&sem); //Increment a semaphore's value
int sem_destroy(&sem); //Destroy a semaphore
```

Note: MacOS does not support sem_init and sem_destroy, as they function on unnamed semaphores. If you are using MacOS, initialize a named semaphore with sem_open and destroy it with 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);
```

Furthermore, note if you attempt to open a named semaphore that has not been unlinked (for instance, due to interrupting a program by sending it a SIGINT via Ctrl-C), the resulting sem_t* will actually be the symbolic constant SEM_FAILED, ***which will cause a segmentation fault if dereferenced!*** To avoid this, you will either need to check if the sem_t* is equal to SEM_FAILED, or set up a signal handler using the <signal.h> library and the signal(SIGINT, sig_handler) system call, which calls the void sig_handler(int signo) function instead of exit().

Mutex locks are synchronization variables that can be held by at most 1 thread. A lock has two operations: lock (acquire) and unlock (release), Linux provides a high-level API for mutex locks in the <pthread.h> library:

```
pthread_mutex_t lock; //Declare a lock
int pthread_mutex_init(&lock, NULL); //Create a lock
int pthread_mutex_lock(&lock); //Acquire a lock
int pthread_mutex_unlock(&lock); //Release a lock
int pthread_mutex_destroy(&mutex); //Delete a lock
```

Condition variables provide another way for threads to synchronize. They allow threads to synchronize based upon the actual value of data, unlike mutex locks, which 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 a shared state protected by a lock. Condition variables are designed to work in conjunction with locks. Linux provides a high-level API for condition variables in the <pthread.h> library:

```
pthread_cond_t cond; //Create a condition variable
int pthread_cond_wait(&cond, &lock); //Unlocks lock then relocks it on a signal
int pthread_cond_signal(&cond); //Signals at least one other thread waiting on cond
```

C Program with semaphores

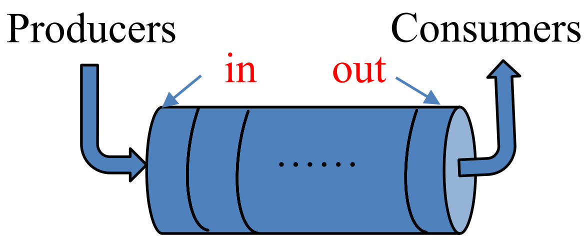
In lab 4, you demonstrated the use of threads and multi-threaded programs that do not have dependencies. In this lab, you will use semaphores to illustrate the use of dependent threads. Demonstrate each of the following steps to the TA to get a grade on this part of the lab assignment.

Step 1. Download the threadSync.c program from Camino, then compile and run it several times. Explain what happens when you run the program and how it differs from the thread programs you used in Lab4.

Step 2. Modify threadSync.c in Step 1 using mutex locks.

Producer – Consumer as a classical problem of synchronization

Step 3. The following diagram illustrates the bounded-buffer producer-consumer problem. A consumer can consume only when the number of (consumable) items is at least 1. A producer can produce only when the number of empty spaces is at least 1. The index to point to a produced item is in, and the index to point to a consumed item is out. Recall that the buffer is a shared variable and is accessible by both producers and consumers.



Write a program illustrating producer and consumer threads coordinating to access the shared buffer using semaphores. You may use the following pseudo-code.

```
//Shared data: 3 semaphores, called full, empty, and mutex
//pool of n<10 buffers, each can hold one item
//mutex semaphore provides mutual exclusion to the buffer pool
//empty and full semaphores count the number of empty and full buffers
//Initially: full = 0, empty = n, mutex = 1

//Producer thread
for (i=0;i<20;i++) {
   …
   produce next item
    …
   wait(empty);
   wait(mutex);
    …
   add the item to buffer
    …
   signal(mutex);
   signal(full);
}

//Consumer thread
for (i=0;i<20;i++) {
   wait(full)
   wait(mutex);
    …
   remove next item from buffer
    …
   signal(mutex);
```

```
    signal(empty);
     …
    consume the item
     …
}
```

Step 4. When employing only mutex locks to coordinate access to a shared buffer between producers and consumers, the issue of busy waiting, or spin locking, may arise. To address this, a condition variable is utilized as an explicit queue where threads can place themselves when certain conditions of execution are not met. This allows threads to wait on the condition, thereby avoiding wasteful busy waiting. A lock is employed to facilitate one thread signaling to others that they may proceed if they are waiting for a particular condition to be met. Write a C program to illustrate the use of condition variables for coordinating the access of the producer and consumer threads to the shared buffer. You may use the following pseudo-code for implementation.

```
//Shared data: 2 condition variables, called full and empty, and a mutex lock, called mutex.
//Same pool of n<10 buffers, but no need to initialize full, empty, and mutex to specific values.

//Producer thread
for (i=0;i<20;i++) {
    …
    produce next item
     …
    lock(mutex);
    while (buffer is full)
          condV.wait(empty, mutex);
     …
    add the item to buffer
     …
    condV.signal(full);
    unlock(mutex);
}

//Consumer thread
for (i=0;i<20;i++) {
    lock(mutex);
    while (buffer is empty)
          condV.wait(full, mutex);
     …
    remove next item from buffer
     …
    condV.signal(empty);
    unlock(mutex);
     …
    consume the item
     …
};
```

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 of lab>
# Title: Lab5 – Step <step#> - <task>
# Description: This program computes <complete description here>
```