#### COEN 177: Operating Systems

**Lab 5: Synchronization using semaphores, lock, and condition variables**

**Objectives**

##### To demonstrate various solution attempts for the Too Much Milk problem

##### To use semaphores, lock, and condition variables for synchronization

##### To develop a C program to solve the producer-consumer problem

##### 

##### **Guidelines**

You have learned in class that threads run concurrently, and the program behavior is undefined when they read/write to a shared memory. This is because 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. So, the thread schedule is non-deterministic, and the resulting output is non-reproducible. To control multi-threaded programs' non-deterministic and non-reproducible behavior, synchronization is required.

Four solutions to the Too Much Milk problem have been demonstrated in the class to provide insight into the synchronization problem. Please review.

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 no other thread is allowed to execute in its critical section when one thread is executing in its critical section. Each thread must request permission to enter its critical section, formally defined as the [entry section](https://jigsaw.vitalsource.com/books/9781119320913/epub/OPS/glossary.xhtml#glo362). When a thread completes execution in the critical section, it leaves through an [exit section](https://jigsaw.vitalsource.com/books/9781119320913/epub/OPS/glossary.xhtml#glo380) 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. This lab demonstrates the use of semaphores, mutex locks, and condition variables for demonstration. A semaphore is considered a generalized lock, and it supports two operations:

* P(): an atomic 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 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 API 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-level 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(&lock); // 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 implements 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. A condition variable is designed to work in conjunction with locks. Linux provides a high-level 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);

**Too Much Milk Problem**

The four solutions to the Too Much Milk problem have been discussed in Class. To provide more insights into the problem solutions, demonstrate the solution attempts to the TA to get a grade on this part of the lab assignment.

1. [20 points] Download the buyingMilksol1.c, buyingMilksol2.c, buyingMilksol3.c, and buyingMilksol4.c programs from Camino, then compile and run. Write down your observation on each solution attempt.

**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.

1. [10 points] Download the threadSync.c program from Camino, then compile and run it several times.

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

1. [10 points] Modify threadSync.c in Step 2 using mutex Locks.

**Producer – Consumer as a classical problem of synchronization**

1. [30 points] 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);

1. [30 points] 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 the correct execution of the C programs.
2. Submit your answers to questions, observations, and notes on Camino
3. Submit the source code for all your programs as .c file(s) and upload them 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 the following information minimally:

# Name: <your name>

# Date: <date> (the day you have the lab)

# Title: Lab5 – task

# Description: This program computes … <you should

# complete an appropriate description here.>