Recitation 5: Synchronization

Department of Computer Science and Engineering
University of Minnesota

July 7, 2025

Guidelines of File Types for Submissions

Please strictly follow the requirements of file types for submissions for future assignments. The generale guidlines are as follows:

- For labs, please submit **.zip** files to Gradescope.
- For programming assignments, please submit **.tar.gz** files to Canvas.

Also, please read handouts of assignments(labs and programming assignments) carefully for any potential changes.

Overview

- Mutex locks.
- Condition variables.
- Exercise 1: Producer-Consumer.
- Exercise 2: Barrier.

Mutex locks

- Locks critical section to ensure exclusive access to a single thread.
- Only one thread could acquire lock at any given point of time.
- Sample code: samples/p1.c (Run with and without the locking).

Mutex locks

```
#include <pthread.h>
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; // mutex var
int pthread_mutex_lock(pthread_mutex_t *mutex); // acquire lock
// CRITICAL SECTION
int pthread_mutex_unlock(pthread_mutex_t *mutex); // release lock
// Returns 0 on success and non-zero error code on failure.
```

Wasting CPU Cycles

Say a thread t1 will start executing its task only if a variable x becomes 1 and x is shared with another thread t2, which has to set it to 1. If we use locks to access x, then there is a continuous need to do locking and checking of x, which is a wastage of CPU cycles.

```
t1_foo(){

while(1){
    lock();
    if (x == 1){
        unlock(); break;
    }
    unlock();
}

...
}
```

```
t2_foo(){

lock();
x = 1;
unlock();

...
}
```

Condition variables

- Use condition variables to suspend/sleep threads until a condition is satisfied
- Coupled with a mutex lock, condition variables should be used within a critical section encapsulated by a lock
- On **signal/wait**, lock is **acquired/released** *implicitly* by the **recipient/calling** thread respectively

```
t1_foo() {
    lock(&mutex);
    while(x != 1){
        // release lock and sleep
        // till a signal
        wait(&cv, &mutex);
        // wake up on signal and
        // acquire lock
    }
    unlock(&mutex);
    ...
}
```

```
t2_foo() {
    ...
    lock(&mutex);
    x = 1;
    signal(&cv);
    // signals a thread waiting
    // on cv
    unlock(&mutex);
    ...
}
```

Condition variables

```
#include <pthread.h>
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
// wait for a signal from another thread
int pthread_cond_signal(pthread_cond_t *cond);
// send signal to one waiting thread
int pthread_cond_broadcast(pthread_cond_t *cond);
// send signal to all waiting threads
// Returns 0 on success and non-zero error code on failure. (samples/p2.c)
```

Exercise 1(prodcons.c)

You have a producer and consumer. The producer makes an item and waits for the consumer to consume it. The consumer waits for the producer to produce an item.
In **prodcons.c**, you have a **mutex** lock and two **condition variables** (cv). One cv is for the producer, which waits for a signal from consumer saying they have consumed. One cv is for the consumer, which waits for a **signal** from the producer saying they have produced.
Complete the given code using hints provided and the provided samples.

Exercise 2(barrier.c)

A **barrier** for a group of threads means any thread must stop at this point and cannot proceed until all other threads reach this barrier.
In **barrier.c**, you will complete the code for **barrier()**. There is a mutex and condition variable provided to you. Use the condition variable to make threads wait until all threads reach the call to wait. The final thread which reaches the wait should not wait, instead broadcast a signal to all threads asking them to move ahead with the execution. Complete the given code using hints and the samples provided.

Deliverables

A target **deliver** is present in the Makefile. You could call **make deliver** from within the exercise folder to create **deliver.zip** outside the exercise folder. The deliver.zip should have all the required files (prodcons.c, barrier.c, Makefile) for submission.

Submit deliver.zip to Gradescope by July 11th, 11:59 pm.

If you are zipping the required files by yourself, **ensure that there are no hidden files and other folders.**