COSC 6360-Operating Systems
Assignment #3: Post Offices and Semaphores-
(Due Wednesday, November 28 at 11:59 PM)

OBJECTIVE

You are to learn how to use POSIX threads, POSIX mutexes and POSIX condition variables.

THE PROBLEM

You are to simulate the behavior of customers in a post office.

Each of your customers will be simulated by a separate thread created by your main program. Customers arriving at the post office will wait until a clerk is available then leave the post office when they are done.

Your post office should have a single FIFO queue that you will represent using a mutex and a condition variable.

All thread synchronization tasks are to be performed using POSIX mutexes and condition variables. You are not allowed to use semaphores of any kind.

YOUR PROGRAM

The number of clerks helping customers in your post office should be read from the command line as in:

```
./a.out 3
```

All other parameters will be read from the—redirected—standard input. Each input line will describe one customer arriving at the post office and will contain three integers representing:

a) A customer serial number,

b) The number of seconds elapsed since the arrival of the previous customer (it will be equal to zero for the first customer arriving at); and

c) The number of seconds the customer will take to get processed by the clerk.

One possible set of inputs could be:

```
1  0  10
2  3   5
3  4  8
4  2     75
```

Your program should track off the number of free clerks, the number of customers that arrived at the post office and he number of customers that had to wait. It should store these two values in global variables so all your threads can access them

Your program should print out a descriptive message including the customer serial number every time a customer:

a) Arrives at the post office;

b) Starts getting helped; and

c) Leaves the post office.

At the end your program should display:

a) The total number of customers that got serviced;

b) The number of customers that did not have to wait; and

c) The number customers that had to wait.

PTHREADS

1. Don't forget the pthread include:

   ```
   #include <pthread.h>
   ```

2. All variables that will be shared by all threads must be declared `static` as in:

   ```
   static int nFreeClerks;
   ```

3. If you want to pass an integer value to your thread function, you should declare it **`void`** as in:

   ```
   void *customer(void *arg) {
       int seqNo;
       seqNo = (int) arg;
       …
   } // customer
   ```

   Since most C++ compilers treat the cast of a **`void`** into an **`int`** as a fatal error, you must use the flag **`-fpermissive`**.

4. To start a thread that will execute the customer function and pass to it an integer value use:

```
pthread_t tid;
int i;
…
pthread_create(&tid, NULL,
   customer, (void *) seqNo);
```

Had you wanted to pass more than one argument to the **customer** function, you should have put them in a single array or a single structure.

5. To terminate a given thread from inside the thread itself, use:

```
pthread_exit((void*) 0);
```

Otherwise, the thread will terminate with the function.

6. To terminate another thread function, use:

```
#include <signal.h>
pthread_kill(pthread_t tid,
             int sig);
```

Note that **pthread_kill(…)**is a dangerous system call because its default action is to immediately terminate the target thread even when it is in a critical section. The safest alternative to kill a thread that repeatedly executes a loop is through a shared variable that is periodically tested by the target thread.

7. To wait for the completion of a specific thread identified by its thread ID, use:

```
pthread_join(tid, NULL);
```

Note that the pthread library has no way to let you wait for an unspecified thread and do the equivalent of:

```
for (i = 0; i < nchildren; i++)
    wait(0);
```

Your main thread will have to keep track of the thread id's of all the threads of all the threads it has created:

```
pthread_t custtid[MAXCUSTOMERS];
for (i = 0; i < nCustomers; i++)
    pthread_join(custtid[i],
                 NULL);
```

PTHREAD MUTEXES

1. To be accessible from all threads pthread mutexes must be declared **static**:

```
static pthread_mutex_t access;
```

2. To create a mutex use:

```
pthread_mutex_init(&one, NULL);
```

Your mutex will be automatically initialized to one.

3. To acquire the lock for a given resource, do:

```
pthread_mutex_lock(&one);
```

4. To release your lock on the resource, do:

```
pthread_mutex_unlock(&one);
```

PTHREAD CONDITION VARIABLES

1. The easiest way to create a condition variable is:

```
static pthread_cond_t ok =
    PTHREAD_COND_INITIALIZER;
```

2. Your condition waits must be preceded by a successful lock request on the mutex that will be passed to the wait:

```
pthread_mutex_lock(&one);
while (ncustomers > maxNCustomers)
    pthread_cond_wait(&ok, &one);
…
pthread_mutex_unlock(&one);
```

3. To avoid unpredictable scheduling behavior, any thread that calls **pthread_cond_signal()** must ***own*** the mutex that the thread calling **pthread_cond_wait()** had specified in its call:

```
pthread_mutex_lock(&one);
…
pthread_cond_signal(&ok);
pthread_mutex_unlock(&one);
```

All programs passing arguments to a thread must be compiled with the **-fpermissive** ***flag. Without it, a cast from a*** **void** ***to anything else will be flagged as an error by some compilers.***

This document was updated last on Saturday, November 10, 2018.