COSC 6360-Operating Systems
Assignment #3: Trying on new clothes

(Now due Thursday, December 10, 2020 anywhere on earth)

OBJECTIVE

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

THE PROBLEM

You are to simulate the behavior of patrons trying new clothes in the fitting room a store.

Each of your patrons will be simulated by a separate thread created by your main program. Patrons arriving at the fitting rom will wait until a cubicle is available, try on the clothes they have taken with them and leave the room when they are done.

Patrons waiting for a free cubicle should form 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

All your program parameters will be read from the—redirected—standard input. The two first lines will contain the number of cubicles in the fitting room and the average time patrons will take to try on a clothing item. Each of the remaining input lines will describe an arriving patron arriving will contain three values representing:

1. The patron's first name, which will never contain spaces,
2. The number of seconds elapsed since the arrival of the previous patron (it will be equal to zero for the first incoming patron); and
3. The number of clothing items the patron is bringing in.

One possible set of inputs could be:

```
2
3
Ann 0  2
Beatriz  3   4
Iman  2 2
```

Your program should keep track of the number of free cubicles, the number of patrons that used the fitting room and the number of patrons that had to wait. It should store these three values in global variables so all your threads can access them

Your program should print out a descriptive message that includes the patron's first name each time a patron:

1. Arrives in front of the fitting room;
2. Gets their own cubicle; and
3. Leaves the fitting room.

Once all patrons are done, your program should display the total number of patrons that used the fitting room, how many of them had to wait and how many did not

PTHREADS

1. Don't forget the pthread include:

   `#include <pthread.h>`

   and the **`-pthread`** compilation flag. (Try **`-lpthread`** if it does not work.)

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

   ```
   static int nFreeCubicles;
   ```

3. If you want to pass arguments to a thread function, you must pack them in a single array or **`struct`** and declare it **`void`** as in:

   ```
   void *patron(void *arg) {
       struct x argStruct;
       argStruct = (struct x) arg;
       …
   } // patron
   ```

   Since some C and C++ compilers treat the cast of a **`void`** into ***anything else*** as a fatal error, you ***may*** have to use the flag **`-fpermissive`**.

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

```
pthread_t tid;
int i;
…
pthread_create(&tid, NULL,
   patron, (void *) myArgs);
```

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, you can 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 patrontid[MAXPATRONS];
for (i = 0; i < nPatrons; i++)
    pthread_join(patrontid[i],
                 NULL);
```

PTHREAD MUTEXES

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

```
static pthread_mutex_t one;
```

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 (nFreeCubicles == 0)
    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);
```