COSC 3360—Fundamentals of Operating Systems

Assignment #3 A Post Office

Due Monday, April 30, 2024, at 11:59 PM

Objective

You are to learn how to use POSIX threads and their advanced synchronization feature.

The Problem

You are to simulate the behavior of up to 128 patrons at a post office. As they do in real life, they will arrive at the post office, wait for an available clerk, get the help they need, and leave the post office.

Each of these patrons will be represented by a separate thread created by your main program. This thread will synchronize with other threads to ensure that each patron will have exclusive access to one of the post office clerks. To achieve that goal, they ***must*** use Pthread mutexes and condition variables. Solutions using semaphores will not be accepted.

Your program

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

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

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

A. A name that will never contain spaces,

B. The number of seconds elapsed since the arrival of the previous patron, and

C. The number of seconds the patron will take to get processed by the clerk.

One possible set of input could be:

```
Alice  0  10
Bob  3  5
Carolina  4  8
Dean  2  7
```

Your program should print out a descriptive message including the patron's name every time a patron:

A. Arrives at the post office;

B. Starts getting help; and

C. Leaves the post office.

At the end, your program should display:

A. The total number of patrons that got serviced;

B. The number of patrons that did not have to wait; and

C. The number of patrons that had to wait.

Non-deterministic outputs

You will notice your program will produce non-deterministic outputs each time two events happen at the same time, say, when a patron arrives just when another leaves. These non-deterministic outputs occur because we have no control over the way our Pthreads are scheduled. The sole way to guarantee a deterministic output is to come up with input data that generate schedules where each event happens at a different unique time.

Pthreads

1. Don't forget the Pthread include:

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

   and the **`-lpthread`** library option in your compilation options

2. All variables that will be shared by all threads must be declared outside of any function as in:

   ```
   static int nFreeClerks, nPatrons;
   ```

3. If you want to pass any data to your thread function, you should declare them **`void`** as in:

   ```
   void *patron(void *arg) {
       lData = (struct pData) arg;
       …
   } // patron
   ```

   You ***must immediately copy*** the contents of **`pData`** into a local variable**.**

Since some C++ compilers treat the cast of a **void** into anything else as a fatal error, you might want 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[MAXTHREADS];
…
pthread_create(&tid[i], NULL,
       patron, (void *) pData);
```

Since you have to pass a string and two integers to the **patron** function, you should put them into a structure.

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

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

Otherwise, the thread will terminate with the function.

6. If you have to terminate another thread function, you may 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. ***You will not have to use it in your program.***

7. To wait for the completion of a specific thread 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 tid[MAXPATRONS];
. . .
for (i=0; i< nPatrons; i++)
    pthread_join(tid[i], NULL);
```

PTHREAD MUTEXES

1. To be accessible from all threads, all Pthread mutexes must be declared outside of any function:

```
static pthread_mutex_t alone;
```

2. To create a mutex use:

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

Your mutex will be automatically initialized to one.

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

```
pthread_mutex_lock(&alone);
```

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

```
pthread_mutex_unlock(&alone);
```

PTHREAD CONDITION VARIABLES

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

```
static pthread_cond_t freeClerks =
    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(&alone);
while (nFreeClerks == 0)
    pthread_cond_wait(&freeClerks,
     &alone);
. . .
pthread_mutex_unlock(&alone);
```

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

```
pthread_mutex_lock(&alone);
. . .
pthread_cond_signal(&freeClerks);
pthread_mutex_unlock(&alone);
```

These specifications were updated last **Wednesday, April 3, 2024**. Check for updates, corrections, and extensions on Teams or Prulu.