COSC 3360/6310 — FUNDAMENTALS OF OPERATING SYSTEMS
COSC 6360 — OPERATING SYSTEMS
ASSIGNMENT #3: THE OBSERVATION PLATFORM

➔Now due Friday, December 3, 2021 at 11:59:59.99 pm🡸

OBJECTIVE

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

THE PROBLEM

A state park has a small observation platform that can only be accessed through a fixed ladder. The platform can accommodate up to ***three*** people and the ladder can only carry one person a time. You are to model the behavior of visitors getting there, possibly waiting for previous customers to leave the platform, climbing the ladder, spending some time on platform, and going down the ladder.

Each visitor will be represented by a separate pthread forked by your main program. Each thread will wait for access to the platform, check that the ladder is not used by another thread, spend some time climbing the ladder, spend some time on the platform, check that the ladder is not used by another visitor, and spend some time going down the ladder.

YOUR PROGRAM

All your program inputs will be read from the—redirected—standard input. Each line will describe one visitor arriving at the platform and will contain one string followed by four integers representing:

1. A short visitor nickname containing no spaces,
2. The number of minutes elapsed since the arrival of the previous visitor (It will be equal to zero for the first visitor arriving at the center.),
3. The number of minutes the visitor will take to climb the ladder,
4. The number of minutes the visitor will remain on the platform, and
5. The number of minutes the visitor will take to go down the ladder

One possible set of input could be:

```
Alice  0  2 8 1
Bob    1  1 10 1
Carol  3  15 1
Daniel  2 1 3 1
Emilia 1 1 5 1
```

For obvious reasons, these minutes will be represented by seconds in our simulation.

You will notice that there is no direct way for a thread to know whether it actually had to wait before getting helped. Your program should thus keep track of the platform occupancy, the status of the ladder as well as the number of visitor threads that had to wait. It should store these three values in global variables so all your threads could access them. Your program should print out a descriptive message including the visitor's name every time a visitor:

1. Arrives near the platform,
2. Climbs the ladder,
3. Reaches the platform,
4. Decide to leave the platform
5. Go down the stairs, and
6. Leaves.

as in:

```
Alice arrives at the platform.
Alice climbs the ladder.
Alice is on the platform.
Alice decides to leave.
Alice goes down the ladder.
Alice leaves.
```

At the end your program should display the total number of visitors that came, and the number of visitors that had to wait as in:

```
8 visitor(s) came to the platform.
4 of them 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 visitor arrives just when another leaves the platform. These non-deterministic outputs occur because we have no control on the way our pthreads are scheduled. The sole way to guarantee a deterministic output is to come 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>
```

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

   ```
   static int nVisitors, nHadToWait;
   ```

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

   ```
   void *visitor(void *arg) {
       struct visit vData;
       vData = (struct visit) arg;
       …
   } // visitor
   ```

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

   Since some C++ compilers treat the cast of a `void` into anything else as a fatal error, you might want use the flag `-fpermissive`.

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

   ```
   pthread_t tid;
   …
   pthread_create(&tid, NULL,
          visitor, (void *) cData);
   ```

   Since you have to pass a string and three integers to the `visitor` function, you should have 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 many 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 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 vtid[MAXVISITORS];
   . . .
   for (i=0; i< nVisitors; i++)
       pthread_join(vtid[i], NULL);
   ```

PTHREAD MUTEXES

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

   ```
   static pthread_mutex_t access;
   ```

2. To create a mutex use:

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

   Your mutex will be automatically initialized to one.

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

   ```
   pthread_mutex_lock(&access);
   ```

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

   ```
   pthread_mutex_unlock(&access);
   ```

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(&access);
   while (nTop == 3 || nLadder == 1)
       pthread_cond_wait(&ok, &access);
   …
   pthread_mutex_unlock(&access);
   ```

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

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