CS-3013, Operating Systems
C-Term, 2018

Project 3 (100 points)
Assigned: Tuesday, January 30, 2018
Checkpoint: Wednesday, February 7, 2018
Due: Wednesday, February 14, 2018

Project 3: Threads and Synchronization

Introduction

This project is intended to provide you experience in working with threads and synchronization tools in user-space. You may carry it out on most Linux or Unix systems, but it will be graded on the virtual machine of this course.

This project must be implemented in *C*. You may use any of the **`pthread`** synchronization tools provided in Linux. *You should test it on a system with as many processors as you can.*

Project Specification

For this project, you will simulate access control to a communal bathroom used by both sexes.[1] Individuals will be simulated by threads, and the access controls must be implemented by data structures and synchronization primitives to cause individuals to wait or proceed. Your program should have at least *one hundred* threads, but preferably many more.

Access to the communal bathroom is controlled by a sign on the door indicating that the bathroom is in one of three states:–

- *vacant*
- *occupied by women*
- *occupied by men*

There is no valid state in which the bathroom is *occupied by both women and men*.

When a person approaches the bathroom, he or she may enter if and only if it is either vacant or occupied by others of the same sex. Otherwise, he or she must wait until it becomes vacant. Upon entering, the user sets the sign on the door to the appropriate state if it was previously vacant.

Multiple people may be waiting at the same time. Obviously, waiting people will all be of the same sex, which is the opposite of those in the bathroom. Arriving people may "jump the queue" if they are of the same sex as those using the bathroom. When the last person leaves the bathroom, he/she sets the sign to indicate that it is vacant. This allows all waiting people to enter one at a time, setting the sign appropriately.

For this problem, no time-of-day scheduling is allowed (e.g., women in one half of each hour and men in the other half of each hour). All arrivals will be at random, and the amount of time that any individual spends in the bathroom is also random.

---

[1] This is essentially Problem #51 on p. 174 of *Modern Operating Systems*, 3rd edition, by Andrew Tanenbaum.

Controlling Access (40 points)

Your mechanism for controlling access to the bathroom must be implemented as a *module* in *C* — that is, as an interface file and an accompanying program module in the form of one or more **`.c`** files. The interface file should be named **`bathroom.h`** and should define types and functions such as:–

```
enum gender = {male, female};
void Enter(gender g);    /* enter the bathroom, but wait until
              vacant if occupied by the opposite gender. Set
              state accordingly */
void Leave(void);        /* Leave the bathroom. Set the state
              to "vacant" if this thread is the last one out */
```

Since we are dealing with only a single bathroom, the program file should implement a global (i.e., compile-time) bathroom object, including synchronization primitives, to maintain its state and support waiting by an *arbitrary* number of users. It must keep track of how many people are currently in the bathroom and of which gender. This data structure must be protected from race conditions among the threads representing individual users. For example, it must not be the case that one user is leaving and setting the state to *vacant* while another user of the same gender is entering and increasing the count.

The **`bathroom.h`** interface and program module must provide an *Initialize* function:–

```
void Initialize( ... )
```

This should have parameters as needed and would only be called by the *master thread* (see below) at the start of the program.

During operation, the bathroom object should keep track of the number of usages of the bathroom, the total amount of time that the bathroom was vacant, the total amount of time that it was occupied, the average queue length, and the average number of persons in the bathroom at the same time.

The **`bathroom.h`** interface and program module must also provide a *Finalize* function:–

```
void Finalize( ... )
```

This should be called after all user threads have exited (see below), and it should print the statistics gathered by the bathroom itself.

In your write-up of this project, you must explain in detail the synchronization method you use to protect the data and preserve the rules of the program. In particular, you should specify the *invariants* that you preserve from one call of **`Enter()`** or **`Leave()`** to the next. Alternatively, you may show the states of the bathroom, the various state transitions, and how you can guarantee that at most one gender will be using it at any time.

Multithreaded Test Program (60 points)

The **`main()`** function of this test program is the *master thread*. It interprets the command line arguments, initializes the bathroom data structure by calling **`Initialize()`**. It then creates a number of individual threads using **`pthread_create()`**. These represent users of the bathroom. See **`man 3 pthread_create`** for examples of its usage.

The master thread then waits until all individual threads have finished, calls **`Finalize()`**, prints a summary of how many user threads have been executed, and exits.

Each individual thread is implemented by a function **`Individual()`**, which is passed as a function pointer to **`pthread_create()`**. **`Individual()`** takes at least the following arguments:–

- *gender* of the bathroom user;
- mean *arrival* time before next needing to use the bathroom;
- mean time to *stay* in the bathroom; and
- *loop count*, indicating of times to repeat itself.

The *gender* of an individual thread is defined by a random variable (with 50% probability for each value) at the time the thread is created. Likewise, *loop count* is a random variable generated at the time the thread is created based on the mean specified in the command line. The mean *arrival* and *stay* times are passed directly to threads as arguments to **`Individual()`**.

Each individual thread operates in a loop with a counter specified by its *loop count* parameter. In the body of the loop, the thread does the following:–

- First, it generates a random number specifying how long to wait until its user needs to use the bathroom. This random number must have a *normal* distribution with a mean specified by the *arrival* time on the command line. The standard deviation should be one-half the mean, but the time should never be less than zero.
- After waiting that long, the user attempts to enter the bathroom using the function **`Enter()`**. This could take a long time if the bathroom is occupied by the users of the opposite sex.
- When the user has entered, the thread generates a second random number with a normal distribution and a mean specified by the *stay* time of the command line. The standard deviation should be one half of the mean, but the stay time should never be less than zero. The user stays in the bathroom this amount of time.
- Finally, the user leaves the bathroom by calling the **`Leave()`** function.
- This loop is repeated the number of times specified by *loop count* for this thread.

The thread for each individual should keep statistics about how much time it spends waiting in the queue for the bathroom — i.e., the number of time units that elapse from the time it invokes **`Enter()`** to the time **`Enter()`** returns.

After the thread has completed all iterations of its loop, it prints the following information:–

- Its own thread number
- Its gender and number of loops
- The minimum, average, and maximum of the time it spent in the queue.

It then informs the master thread that it is done, and it exits.

You are responsible for figuring out how the threads inform the master that they are finished and how both the user threads and the master thread exit cleanly. For example, you might use **`pthread_join()`** or one of the barrier synchronization mechanisms. You might also discover that you need to protect against multiple threads trying to print at the same time, so that their output does not get mixed up or interleaved.

Time units in this assignment are not specified. However, it is suggested that you make them small — say, ten or twenty milliseconds — so that you can simulate a large number of users in a short amount of run time. To simulate arrival and stay times, each thread may use the **`usleep()`** function; see

```
man 3 usleep
```

Command Line

You main program should be called **`bathroomSim`**. It should take a command line containing four arguments in the following format:–

```
bathroomSim nUsers meanLoopCount meanArrival meanStay
```

In this command line

- **`nUsers`** is the number of people in the simulation. They should be randomly distributed between men and women with equal probability.
- **`meanLoopCount`** is the average number of times that a user repeats the bathroom usage loop. The actual number for a particular user should be generated from a normal distribution with this mean. The actual number of loop iterations should never be less than one.
- **`meanArrival`** is the mean of the arrival intervals. That is, each user will arrive at the bathroom an average of **`meanArrival`** units of time after the start of the simulation and also after leaving the bathroom from the previous use. Each thread generates an actual arrival time for each iteration from a normal distribution based on this mean. The standard deviation of this distribution should be one-half the mean, but no arrival interval should be less than zero.
- **`meanStay`** is the average length of time that a person stays in the bathroom, once he or she has entered. Stay times should be generated from a normal distribution with this mean. The standard deviation of this distribution should be one-half the mean, but no stay should be less than zero.

Random number distributions

Linux has several random number generators. **`rand()`** is the simple random number generator specified in *Kernighan and Ritchie*, and generates integers in the range **`0..32767`**. A better random number generator is **`drand48()`** , which generates uniformly distributed floating point numbers in the range **`[0.0..1.0)`**.[2]

[2] The thread-safety of random number generators has been debated extensively. See the following URL for a discussion:– http://www.evanjones.ca/random-thread-safe.html. More important is that the multi-threaded nature of this assignment means that the sequence is not repeatable. To keep this assignment simple, you may ignore these issues.

Students from previous terms have used following method (called the *Box-Muller* transform) for generating random numbers with an approximate normal distribution:–

> After generating two unit random numbers **a** and **b** (in the range **[0, 1)**), use this formula:– **z = sqrt(-2 * log(a)) * cos(2 * pi * b)**, where **log(..)** is the *natural logarithm* function defined in **<math.h>**.

Suggestions

It is suggested that for debugging purposes, you make the number of threads and the arrival times small, while you make the time a thread stays in the bathroom large, in order to maximize the probability that several threads have to wait.

However, once you believe that you have debugged your program, you should make the number of threads large and the amount of time staying the bathroom small so that you really exercise it. In particular, you should run it a number of times with different parameters in order to find out under what conditions one gender can "starve" the other from access to the bathroom. You should also run it on a system with a large number of processors — for example, the Zoo Lab workstations with your virtual machine set to use all eight processors.

Structuring your code

Your code must comprise *at least two modules* in *C*.

- One module called **bathroom.o** must implement the bathroom itself. An accompanying interface file called **bathroom.h** file must define the *interface* to the bathroom — i.e., the functions or methods that can be called, but not the data structures that are private to the bathroom implementation.
- One or more *separate* modules must implement the random test program. This includes the **main()** function, the **Individual()** function, and any supporting functions. These access the bathroom module via the functions or methods defined in **bathroom.h**.

When you compile your code, you *must* specify the **-Wall** flag. This prints all warnings. It is best to include this in the **CFLAGS** variable of the **makefile**. In addition, you should specify the **-pthread** option to **gcc**. This causes **gcc** to link your program with the **pthread** library, but it also invokes other compile-time options pertaining to multi-threaded code.

Notes and Debugging

In previous terms, some submissions had bugs that allowed both sexes in the bathroom at the same time. It might be a good idea to keep separate counters of the number of men and the number of women in the bathroom. One invariant — which could be tested at run time — is that at least one of these counters must be zero. This can be tested with the **assert()** macro (see **man 3 assert**), which evaluates an expression and aborts the program if false.

Another common bug had members of both sexes in the wait queue at one time. This meant that the synchronization was not working properly. The **assert()** macro could also be used verify this invariant.

In general, a multi-threaded program is harder to debug than a single-threaded program. For example, if you set a breakpoint in the function implementing the individual thread, it will

cause all threads to break at that point. An alternative is to use conditional breakpoints; see the **`Eclipse`** or **`gdb`** documentation about these.

Most students in previous terms have reported that the maximum number of threads supported by the virtual machine of this course is about 200-350 in one process.[3] This appears to be due to the size of the stack for each thread. It is possible to change the stack size of each thread by specifying something in the **`attr`** parameter of **`pthread_create()`**. To do this, you need to create and set an **`attr`** object to pass to **`pthread_create()`**. You can use the **`pthread_attr_setstacksize()`** function to change the stack size. See the man pages for these functions.

Project Teams

You may work on this assignment as an individual project or in teams of two. However, you are strongly urged to collaborate with other students and other teams on your testing. That is, students may share the **`bathroom.h`** interfaces so that one may run his/her test program against another's **`bathroom.o`** module and vice versa.

You should discuss technical issues of the project with each other — for example, by the InstructAssist Forum or on a whiteboard — in particular, how to approach the problems of synchronization, creating of multiple threads, etc. You may also help each other test your programs, debug synchronization issues, etc.

Each individual or team is responsible for writing the solution to the problem in your/its own code and your/its own coding style and for submitting. Copying is not permitted, except for the **`.h`** file describing the bathroom interface.[4]

Submitting this Project

Be sure to put your name and team at the top of every file you submit and at the top of every file that you edit!

You must submit the following for this project:–

- A write-up (in **`.doc`**, **`.docx`**, or **`.pdf`** format) documenting both parts of this assignment. It must include a clear, cogent explanation of *why* your control module is correct, and it must also include a clear, cogent explanation of how your multi-threaded test program works, why it fully exercises your control module, how it avoids conflicts when printing out the statistics of each thread, and how it manages to safely exit from all threads.

  In your documentation, be sure to describe the programming invariants — i.e., the logical statements that are true when no thread is changing the state of the bathroom. For example, if you use **`pthread_mutex`** for locking the bathroom data structures, the invariants should describe the state of the variables when the **`mutex`** in not locked.

---

[3] One student, running on a 64-bit Linux desktop system, reported that he was able to spawn over 20,000 threads.

[4] Shared **`bathroom.h`** interface files should be posted on the Forum of InstructAssist. Each such posted interface should be signed by an individual or a team.

Your write-up must also list the names and the contents of the files in your submission, and it must include instructions on how to run your program and the recommended settings of the parameters.

- The **`.h`** files and the **`.c`** files of this assignment. These must compile correctly without warnings on your virtual machine.
- A **`makefile`** to **`make all`**, **`make clean`**, and **`make`** the individual modules.
- Output of at least three test cases (including one long test case) showing the behavior of users of this communal facility under different parameters.

Zip all of your files together into a single zip file named **`Project3_username.zip`** or **`Project3_teamname.zip`** and submit via *InstructAssist*.

Grading Rubric

Bathroom Module:–

- Compiles without warnings under **`-Wall`** flag – 1 point
- Correct implementation of **`Enter()`** under each of the three states – 9 points (three point per state)
- Correct implementation of **`Leave()`** – 5 points for # users = 1, 5 points for # users > 1
- Correctly maintaining the state and the count of the number of users in the bathroom – 10 points
- Satisfactory write-up, including analysis of *invariant* or states of the bathroom and an explanation of how you know that no two users of the opposite sex are in the bathroom at the same time – 10 points
- Total – 40 points

Multi-threaded test program:–

- Master thread compiles without warnings under **`-Wall`** flag, parses arguments, and spawns each thread with random *loop count* – 5 points
- Master thread prints statistics and exits cleanly after all user threads have terminated – 5 points

User threads:–

- Random number generation with normal distribution – 10 points
- Implement arrival, stay, and loop count iterations correctly – 10 points
- Collect and print statistics correctly per thread without race conditions, and exit cleanly – 10 points
- Output of three student test cases – 5 points
- Correct operation with graders' test cases – 5 points
- Satisfactory write-up[5] commenting upon the queue behavior – 10 points
- Total (master and user) – 60 points

[5] The two write-ups may be in the same document file but must be in clearly separated sections.