Programming Assignment IV

Due on Friday, November 16, 2018 @ 11pm

100 points

This is a **ThreadMentor** programming assignment, and you can only use semaphores and locks. The use of any other system and/or synchronization primitives is unacceptable. **Note that you are now able to lock `stdout` so that a thread can print a line completely.**

Party Room Problem

A landlord has a big apartment with a fantastic social room. Many students stay in this apartment and have parties in the social room. Since the neighbors frequently complained about party noise, the landlord comes and checks his apartment frequently to ensure his fantastic social room is being used properly, and breaks up the party if there are too many students in the room.

Here is a list of rules:

- Any number of students can be in this room at the same time.
- Students can enter and leave the room freely unless the landlore is in the room checking.
- The landlord enters this room freely.
  - If there are more than ***n*** students in the room, where ***n*** is an input, the landlord breaks up the party and all students must leave.
  - If there are no students or there are no more than ***n*** students in the room, the landlord leaves and comes back later.
- While the landlord is in the room, no students may enter and no students may leave.
- The landlord does not leave the room until all students have left if the lanlord breaks up the party.
- If this is the landlord's last check, all students must leave no matter what. Then, the landlord leaves. Note that only the landlord knows when the last check will occur.
- There is only one landlord!

The landlord is a thread with the following pattern. The landlord only checks the room ***m*** times and retires because he feels it is too tedious and too stressful to handle this situation. After ***m*** iterations, he sends all students home, sells his apartment, and goes to Bahama for a long long vacation.

```
for (i = 0; i < m; i++) {     // m is an input value
     Delay();                 // take some rest
     CheckRoom(...);          // check the social room
     Delay();                 // take some rest
}
```

`CheckRoom()` is a function for the landlord to check the room based on the above rules.

Each student is also a thread:

```
while (1) {
     Delay();                 // study for a while
     GoToParty(...);          // go to the party
     Delay();                 // well, everyone needs some sleep
}
```

Your job is to fill all the needed synchronization and other activities into two functions `CheckRoom()` and `GoToParty()`. **Your implementation must satisfy the stated conditions. Otherwise, your implementation is not a correct one and you will receive zero point.**

Write a C++ program using **ThreadMentor** to simulate these activities. Note that you can only use mutex locks and semaphores. **You may execute `Delay()` a random number of times to add more randomness to your program.** **Your program will not be considered as a correct one if any other synchronization primitives are used.**

Input and Output

The input of your program consists of the following:

- There are three command line arguments:
  1. ***m***: the number of times for the landlord to check the room.
  2. ***n***: the maximum number of students in the room without being broken up by the landlord.
  3. ***k***: the total number of students in the apartment. Note that ***k*** must be equal to or larger than ***n***.
- These three command line arguments are supplied to your program as follows:

  ```
  ./prog4  m n k
  ```

  Thus, `prog4 7 9 20` means the landlord checks the room 7 times, if there are more than 9 students in the room the landlord breaks up the party, and there are 20 students in his apartment. If ***m*** is 0, the default value is 5; if ***n*** is 0, the default value is 5; and if ***k*** is 0, the default value is 10. For example, `prog4 0 6 0` means that the landlord checks the room 5 times, if there are more than 6 students in the room the landlord breaks up the party, and there are 10 students in total.
- You many assume all command line arguments being non-negative and are no more than 30. Furthermore, you may assume that ***n*** is always less than or equal to ***k***.
- Each student prints out the following messages. Note that a student may enter and leave the room multiple times. **Note also that when the landloard is checking the room no students can enter and leave. As a result, there could be a delay between waiting to enter and being in the room. The same holds true for waiting to leave and being out of the room.**

  ```
  Student 5 starts    // show this when the thread of student 5 starts
  .....
  Student 5 waits to enter the room
                      // show this when student 5 is about to join the party
  .....
  Student 5 enters the room (XX students in the room)
                      // show this after student 5 successfully enters the room
                      // XX is the number of students, including student 5, in the room
  .....
  Student 5 waits to leave the room
                      // show this when student 5 is about to leave the room
  .....
  Student 5 leaves the room (XX students in the room)
                      // show this after student 5 has left the room
                      // XX is the number of students, excluding student 5, in the room
                      //     when student 5 left
  .....
  Student 5 terminates //show this when the thread of student 5 terminates
  ```

- The landlord can enter and leave the room freely.

  ```
  The landlord checks the room the YY-th time // YY runs from 1, 2, 3 ....
            // no students in the room can leave and
            //     no students outside of the room can enter
  .....     //     while the landlord is checking
  ```

  After the landlord finishes checking the room, s/he prints out one of the following messages:
  - The room is empty when the landlord is checking. This is the simplest one. The following is the second message by the landlord **AFTER** s/he enters the room.

    ```
    The landlord finds the room has no students and leaves.
    ```

  - The room does not have enough number of students. As a result, the landlord simply leaves. This is the second message by the landlord **AFTER** s/he enters the room.

```
The landlord finds there are XX students in the room (condition being good) and leaves.
            // XX is the current number of students in the room
            // note that XX is less than or equal to m, the max allowed number
```

- The room has more students than expected, and the landlord breaks up the party. The following messages are shown by the landlord **AFTER** s/he enters the room.

```
The landlord finds XX students in the room and breaks up the party
     .....
     Student 7 waits to leave the room  // student 7 cannot leave because the landlord is checking
     .....
     Student 2 waits to enter the room  // student 2 cannot enter because the landlord is checking
     .....
The landlord finishes checking and forces everyone to leave
                                        // now every one in the room must leave
     .....
     Student 3 leaves the room (XX students in the room)
                      // note that XX does not include student 3
     .....
     Student 7 leaves the room (XX students in the room)
                      // note that XX does not include student 7
     .....
     Student 9 waits to enter the room  // student 9 cannot enter because the landlord is in the room
     .....
The landlord leaves the room and the room is empty
                    // between "finishes" and "leaves" all students leaves
                    //   even though some other students may wait to enter
                    // all students who were in the room must leave before
                    //   the landlord print the "leaves" message
```

- After a specific number of room checking, the landlord retires and the whole system ends. **Note that all students must leave the room and no students will be admitted. After all students have left, the landlord prints the following message:**

```
After checking the room YY times, the landlord retires and is on vacation!
            // this is always the last message your program prints.
            // note that by now all students must have left the room
            //    and all student threads must have already terminated.
```

- The landlord prints from position 1, and each student prints from position 6 (*i.e.*, 5 leading spaces).

Submission Guidelines

General Rules

1. All programs must be written in C++.
2. Use the `submit` command to submit your work. You can submit as many times as you want, but only the last on-time one will be graded.
3. Unix filename is case sensitive, `THREAD.cpp`, `Thread.CPP`, `thread.CPP`, etc are ***not*** the same.
4. We will use the following approach to compile and test your programs:

```
make noVisual      <-- make your program
./prog4 8 4 20     <-- test your program
```

   This procedure may be repeated a number of times with different input files to see if your program works correctly.
5. Your implementation should fulfill the program specification as stated. Any deviation from the specification will cause you to receive **zero** point.
6. A `README` file is always required.
7. **No late submission will be graded.**
8. **Programs submitted to wrong class and/or wrong section will not be graded.**

Compiling and Running Your Programs

This is about the way of compiling and running your program. If we cannot compile your program due to syntax errors, wrong file names, etc, **we cannot test your program, and, as a result, you receive 0 point**. If your program compiles successfully but fails to run, **we cannot test your program, and, again, you receive 0 point**. **Therefore, before submitting your work, make sure your program can compile and run properly.**

1. **Not-compile programs receive 0 point.** By **not-compile**, I mean any reason that could cause an unsuccessful compilation, including missing files, incorrect filenames, syntax errors in your programs, and so on. Double check your files before you submit, since I will **not** change your program. Note again: Unix filenames are ***case sensitive***.
2. **Compile-but-not-run programs receive 0 point. Compile-but-not-run** usually means you have attempted to solve the problem to some degree but you failed to make it working properly.
3. **A meaningless or vague program receives 0 point even though it compiles successfully.** This usually means your program does not solve the problem but serves as a placeholder or template just making it to compile and run.

Program Style and Documentation

This section is about program style and documentation.

1. For each file, the first piece should be a program header to identify yourself like this:

```
// -----------------------------------------------------------
// NAME : John Smith                       User ID: xxxxxxxx
// DUE DATE : mm/dd/yyyy
// PROGRAM ASSIGNMENT #
// FILE NAME : xxxx.yyyy.zzzz (your unix file name)
// PROGRAM PURPOSE :
//    A couple of lines describing your program briefly
// -----------------------------------------------------------
```

   Here, **User ID** is the one you use to login. It is ***not*** your social security number nor your M number.

   For each function in your program, include a simple description like this:

```
// -----------------------------------------------------------
// FUNCTION  xxyyzz : (function name)
//     the purpose of this function
// PARAMETER USAGE :
//    a list of all parameters and their meaning
// FUNCTION CALLED :
//    a list of functions that are called by this one
// -----------------------------------------------------------
```

2. Your programs must contain enough concise and to-the-point comments. Do not write a novel!
3. Your program should have good indentation.
4. Do not use global variables!

Program Specification

Your program must follow exactly the requirements of this programming assignment. Otherwise, you receive 0 point even though your program runs and produces correct output. The following is a list of potential problems.

1. Your program does not use the indicated algorithms/methods to solve this problem.
2. Your program does not follow the structure given in the specification. For example, your program is not divided into functions and files, etc when the specification says you should.
3. Any other significant violation of the given program specification.
4. **Incorrect output format.** This will cost you some points depending on how serious the violations are. The grader will make a decision. Hence, carefully check your program output against the required one.
5. **Your program does not achieve the goal of maximum parallelism.**

Program Correctness

If your program compiles and runs, we will check its correctness. We normally run your program with two sets of input data, one posted on this programming assignment page (the public one) and the other prepared by the grader (the private one). You program must deliver correct results for both data sets. Depending on the seriousness of the problem(s), significant deduction may be applied. For example, if your program delivers all wrong results for the public data set, you receive 0 point for that component.

The `README` File

A file named `README` is required to answer the following questions:

1. **Question:** How did you make sure that no students can enter while the landlord is in the room and checking? Explain your approach in details.
2. **Question:** How did you make sure that the landlord will not leave until all students have left the room? Explain your approach in details.
3. **Question:** How did you make sure the message `"After checking the room XX times, the landlord retires and is on vacation!"` is the last message printed by your program?

You should elaborate your answer and provide details. **When answering the above questions, make sure each answer starts with a new line and have the question number (*e.g.*, Question X:) clearly shown. Separate two answers with a blank line.**

Note that the file name has to be `README` rather than `readme` or `Readme`. Note also that there is ***no*** filename extension, which means filename such as `README.TXT` is ***NOT*** acceptable.

`README` must be a plain text file. We do not accept files produced by any word processor. Moreover, watch for very long lines. More precisely, limit the length of each line to no more than 80 characters with the Return/Enter key for line separation. Missing this file, submitting non-text file, file with long lines, or providing incorrect and/or vague answers will cost you many points. Suggestion: Use a Unix text editor to prepare your `README` rather than a word processor.

Final Notes

1. Your submission should include the following files:
   1. File `thread.h` that contains all class definitions of your threads.
   2. File `thread.cpp` contains all class implementations of your threads.
   3. File `thread-support.cpp` contains all supporting functions such as `CheckRoom()`, `GoToParty()`, other functions designed and implemented by you as needed.
   4. File `thread-main.cpp` contains the main program.
   5. File `Makefile` is a makefile that compiles the above three files to an executable file `prog4`. **Since we may use any lab machine to grade your programs, your makefile should make sure all paths are correct. Note also that without following this file structure your program is likely to fall into the compile-but-not-run category, and, as a result, you may get a low grade. Before submission, check if you have the proper file structure and correct makefile.**
   6. Your `Makefile` **should not** activate the visualization system.
   7. Your `Makefile` **should not** use a path leading to your local directory.
   8. The `README` file.

   Note also that without following this file structure your program is likely to fall into the compile-but-not-run category, and, as a result, you may get low grade. Therefore, before submission, check if you have the proper file structure and a correct makefile.
2. **Always start early, because I will not grant any extension if your home machine, network connection, your phone line or the department machines crash in the last minute.**
3. **Since the rules are all clearly stated, no leniency will be given and none of the above conditions is negotiable. So, if you have anything in doubt, please ask for clarification.**
4. **Click here to see how your program will be graded.**