Programming Assignment V

100 points

River Crossing

A particular river crossing is shared by both cannibals and missionaries. A single boat is used to cross the river, but it only seats three people, and must always carry a full load. In order to guarantee the safety of the missionaries, you cannot put one missionary and two cannibals in the same boat (because the cannibals would gang up and eat the missionary), but all other combinations are acceptable. You are to write a Hoare style monitor to implement this policy.

Each cannibal is a thread with the following pattern:

```
while (1) {
     Delay();                // take a rest
     CannibalArrives(...);   // register to cross the river
     // other stuffs
     // come back for another river crossing
}
```

The following shows a similar missionary thread:

```
while (1) {
     Delay();                   // take a rest
     MissionaryArrives(...);    // register to cross the river
     // other stuffs
     // come back for another river crossing
}
```

Both cannibal and missionary threads will run an infinite loop. When a cannibal arrives the river bank, it must register to the monitor with a call to the monitor procedure CannibalArrives() with some arguments. A missionary thread, on the other hand, calls MissionaryArrives(). The caller will be blocked in the monitor until the monitor can find a safe boat load. The boat is also simulated by a thread and has the following pattern:

```
while (1) {
     Delay();          // take a rest
     BoatReady(...);   // ready for the next round
     Delay();          // row the boat
     BoatDone(...);    // all people are on the other side
     // come back for another river crossing
}
```

The boat calls the monitor procedure BoatReady() when it is ready (*i.e.*, empty) for the next boat load. It may be blocked in the monitor until a safe boat load is possible. At that time, the

boat is released from the monitor with the passenger names, and starts rowing. It also uses Delay() to simulate spending sometime to cross the river. Then, the boat calls monitor procedure BoatDone() to indicate one boat load is completed, and comes back for another boat load. Note that the call to BoatDone() may or may not block the boat depending on your implementation. But, you should be aware of the fact that all passengers have to be off before the boat can be used again.

Your monitor maintains all necessary information to guarantee a safe boat load. When a safe boat load is possible, your monitor tells the boat to send the selected three passengers (*i.e.*, three cannibals, three missionaries, or two missionaries and one cannibal) cross river. After a boat load completes, each of these three passengers returns to their respect thread function, does something and comes back magically for another river crossing. The boat will also magically come back to the river bank for the next boat load. **Thus, all river crossings are in the *same* direction.** Once a safe boat load is assembled, your monitor should display one of the following three messages depending upon the type of the current boat load. Note that is is "your policy" to assemble a safe boat load, but you have to be sure that once a safe boat load is possible, you have to allow the boat to move. In other words, you should not delay the move of a possible safe boat load.

```
MONITOR(x): three cannibals (c, c, c) are selected
MONITOR(x): three missionaries (m, m, m) are selected
MONITOR(x): one cannibal (c) and two missionaries (m, m) are
selected
```

where x is the number of safe boat loads have been made, and c and m are the cannibal number and missionary number, respectively.

It is **very important** that no one can jump off the boat before the boat reaches the other side and requests for another cross, and that no one can jump on the boat before getting the permission to do so from the monitor. Additionally, no one can cross the river without being on the boat.

After some number of boat loads have been made, your monitor should display the following message and exit the system:

```
MONITOR: nnn crosses have been made.
MONITOR: This river cross is closed indefinitely for renovation.
```

where nnn is a positive integer taken from the command line and passed to the monitor in the initialization stage. **Note that a boat load completes only if the boat comes back for the next boat load.** After displaying this message, the system terminates.

You must terminate your program gracefully. More precisely, if b boat loads complete, then your program completes. However, your program cannot terminate while the last boat load is still crossing the river.

Except for locking stdout, no mutex locks and semaphores should be used. Note also that because all control mechanisms are in the monitor, your program should not use any global entities except for possible mutex locks or semaphores for locking stdout. Violating this rule will receive no point for this assignment.

Input and Output

The input of your program consists of the following:

- The number of cannibals c, the number of missionaries m, and the number of safe boat loads b should be taken from the command line arguments as follows:

```
./prog5  c  m  b
```

  Thus, prog5 15 8 10 means there are 15 cannibals and 8 missionaries, and after 10 safe boat loads the program terminates. If any one of these command line arguments is 0, its default value is used (*i.e.*, c = 8, m = 8 and b = 5). For example, prog5 0 3 0 means that there are 8 cannibals and 3 missionaries, and that after 5 safe boat loads the program terminates. **All command line arguments are always non-negative integers.**

- Your program should generate an output similar to the following:

```
     Cannibal 5 starts
   Cannibal 3 starts
     ..........
***** BOAT thread starts
     ..........
Missionary 1 starts
     ..........
 Missionary 2 starts
***** The boat is ready
     ..........
Cannibal 1 arrives
   Cannibal 3 arrives
  Missionary 2 arrives
  Cannibal 2 arrives
     ..........
MONITOR(1): three cannibals (1, 2, 3) are selected
     Missionary 5 arrives
     ..........
***** Boat load (1): Passenger list (c1, c2, c3)
     ..........
***** Boat load (1): Completed
     ..........
   Missionary 3 arrives
     ..........
MONITOR(2): one cannibal (4) and two missionaries (3, 5) are
selected
          ..............................
```

```
***** Boat load (2): Passenger list (c4, m3, m5)
          ..............................
  Cannibal 2 arrives
   Missionary 3 arrives
          ..............................
***** Boat load (2): Completed
     ..........
     Missionary 5 arrives
    Missionary 4 arrives
Missionary 1 arrives
     ..........
MONITOR: 10 crosses have been made.
MONITOR: This river cross is closed indefinitely for
renovation.
```

- In the above, monitor's output always starts on column 1. The output lines from cannibal ***i*** and missionary ***i*** start on column ***i***.
- The number of cannibals, the number of missionaries, and the number of safe boat loads are given as command line arguments. You may assume that the input data are correct and there are no more than 30 threads in total.
- All messages from the boat thread starts with *****, and c and m in the passenger list represent cannibal and missionary, respectively, followed by their number.
- Note that messages from cannibals, missionaries, the boat and the monitor must be printed from cannibal threads, missionary threads, the boat thread and the monitor.
- Due to the dynamic behavior of multithreaded programs, you will not be able to generate an output that is exactly identical to the above.

Submission Guidelines

General Rules

1. All programs must be written in C++.
2. Unix filenames are case sensitive, THREAD.cpp, Thread.CPP, thread.CPP, etc are *not* the same.
3. We will use the following approach to compile and test your programs:

```
make noVisual        <-- make your program w/o visualization
./prog5 c m b        <-- test your program
```

   This procedure may be repeated a number of times with different input to see whether your program works correctly.

4. Your implementation should fulfill the program specification as stated. Any deviation from the specification may cause you to receive zero points.
5. The binary created by your makefile must be named **prog5**.

Compiling and Running Your Programs

This is about 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 you receive low point. If your program compiles successfully but fails to run, you receive low point. Therefore, before submitting your work, make sure your program can compile and run properly.

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
// 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
// ------------------------------------------------------------
```

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 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.

The README File

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

1. The logic of your program, why does your program work?

2. Why every boat load has *exactly* three persons and is safe?

3. Clearly state your policy of assembling a safe boat load.

4. Why do CannibalArrives(), MissionaryArrives(), BoatReady() and BoatDone() work properly? More precisely, explain that why your version of CannibalArrives(), MissionaryArrives(), BoatReady() and BoatDone() can always guarantee:

    a) No one can get off the boat before a crossing completes.
    b) While the boat is crossing the river, no one can jump on and get off the boat.
    c) Everyone on the boat has been registered properly. More precisely, no one who was not registered can be on the boat.
    d) No one who is on the boat for a crossing can come back and register for another crossing *before* the current boat load completes.

    Note that argument such as "*because a monitor and condition variables are used to ....., the indicated situation cannot happen*" will not be counted as convincing. You should explain why the situation will not happen through the use of monitor and condition variables. Providing arguments like that will receive low or very low grade.

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.

Your 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

Your submission should include the following files:

1. thread.h that contains all class definitions of your threads.
2. thread.cpp contains all class implementations of your threads.
3. boat-monitor.h contains the class definition of your monitor.
4. boat-monitor.cpp contains the class implementation.
5. thread-main.cpp contains the main program.
6. Makefile is a makefile that compiles the above three files to an executable file prog5.
7. README

Note that without following this file structure your program is likely to fall into the compile-but-not-run category, and you may get low grade. So, before submission, check if you have the proper file structure and correct makefile. Note that your Makefile **should not** activate the visualization system.