

# Programming Assignment V

**Due on Friday, December 7, 2018 @ 11pm**

**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.

Because you have a chance to set a policy for your monitor, you need to answer the following questions *with elaborations*:

- The working logic of your monitor, including the meaning of each local variable and condition variable. You must also include a discussion of your implementation of `CannibalArrives()`, `MissionaryArrives()`, `BoatReady()` and `BoatDone()`.
- Why does your policy work? More precisely, why does your policy can always guarantee safe boat loads?
- Show that your policy will not cause deadlocks. If your policy is not designed carefully, it could cause deadlocks within your monitor.

These should be presented in your `README` file. Detailed discussion and correct reasoning are expected. Vague discussion risks lower grade.

**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. Use the `submit` command to submit your work. You may 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
./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.

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). Your 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:

- The logic of your program
- Why does your program work?
- The meaning, initial value and the use of each variable. Explain why their initial values and uses are correct. Justify your claim.
- Every boat load has *exactly* three persons and is safe.

- State your policy of assembling a safe boat load clearly.
- Why do `CannibalArrives()`, `MissionaryArrives()`, `BoatReady()` and `BoatDone()` work properly? More precisely, explain to me that your version of `CannibalArrives()`, `MissionaryArrives()`, `BoatReady()` and `BoatDone()` can always guarantee
  1. No one can get off the boat before a crossing completes.
  2. While the boat is crossing the river, no one can jump on and get off the boat.
  3. Every one on the boat has been registered properly. More precisely, no one who was not registered can be on the boat.
  4. 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 must terminate your program gracefully. More precisely, if `b` boat loads completes, then your program completes. However, your program cannot terminate while the last boat load is still crossing the river.

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 `boat-monitor.h` contains the class definition of your monitor.
  4. File `boat-monitor.cpp` contains the class implementation.
  5. File `thread-main.cpp` contains the main program.
  6. File `Makefile` is a makefile that compiles the above three files to an executable file `prog5`. **Note that without following this structure your program is likely to fall into the compile-but-not-run category, and, as a result, 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.**
  7. File `README`.
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](http://www.csl.mtu.edu/cs3331.ck/www/PROG/PG5/prog5.html) to see how your program will be graded.**