# Programming Assignment V

**Due on Friday, April 15, 2017 @ 11pm**

**100 points**

---

## Baboons Crossing a Canyon

A canyon cuts through the territory of a colony of baboons. The baboons use a rope stretching across the canyon to cross from one side to the other. The rope is strong enough to permit any number of baboons to cross in the same direction at the same time. However, the rope is too thin for the baboons to cross the canyon in both directions at the same time. Consequently, a baboon that wants to cross from west to east must wait until all westward-moving baboons have finished crossing and the rope is free. If the rope is being used by westward-moving baboons, then other baboons may start to cross from east to west no matter how many eastward-moving baboons are waiting on the other side.

Unfortunately, this simple protocol could cause *starvation*. **Why? Figure it out yourself**. To overcome this problem, when a baboon that wants to cross to the east (*resp*, west) arrives at the rope and finds baboons crossing to the west (*resp.*, east), it waits until the rope is empty, but no more westward-moving (*resp.*, eastward-moving) baboons are allowed to start crossing until at least one waiting baboon has crossed the other way.

In the system, baboons are simulated by threads. Initially, there are $e > 0$ baboons going eastward, and $w > 0$ baboons going westward. Each baboon makes $t > 0$ trips and then disappears forever (*i.e.*, exits the system). Each eastward-moving (*resp.*, westward-moving) baboon goes to the east (*resp.*, west) end and magically comes back to the west (*resp.*, east) end for the next trip. This repeats $t$ times. The following shows a template of a baboon thread, where `DIRECTION` is the moving direction, and `GetOnRope()` and `GetOffRope` are monitor procedures:

```
for (int i = 1; i <= t; i++) {
     Delay();                       // plays for a while
                                    // You may execute the Delay() function
                                    //   for a random number of times
                                    //   to produce more realistic results
     GetOnRope(DIRECTION, ...);     // makes a request to move on the rope
     Delay();                       // has the permission
                                    // gets on the rope and
                                    // moves for a while
     GetOffRope(DIRECTION, ...);    // gets off the rope
     // at the other end now, print message
     // magically comes back
}
```

Your Hoare type monitor maintains all necessary information to make sure baboons will cross the canyon based on the give conditions. In particular, your monitor must guarantee the following:

1. Baboons on the rope only move in the same direction.
2. Baboons must use the rope to cross the canyon. In other word, baboons cannot *magically* cross the canyon and *magically* come back. This magic only applies to the returning trip.
3. No baboon can get on the rope without a permission.

4. When a baboon arrives and there are baboons on the rope moving in the opposite direction, no baboon can get on the rope and move in the opposite direction. More precisely, if an eastward-moving baboon arrives and there are a number of westward-moving baboons on the rope, then no other westward-moving baboons, whether they are waiting or just arrive, can be on the rope.

Write a C++ program using **ThreadMentor** to simulate this activities. **You can only use one Hoare type monitor to solve this problem. Your program will not be counted as a correct one if any other synchronization primitives and/or non-Hoare monitors are used.**

## Input and Output

The input of your program consists of the following:

- The number of eastward-moving baboons `e`, westward-moving baboons `w`, and and the number of trips `t` to be made for each baboon should be taken from the command line arguments as follows:

```
./prog5  e w t
```

  Thus, `./prog5 15 8 12` means there are 15 eastward-moving baboons, 8 westward-moving baboons, and each baboon will make 12 trips. If any one of these command line arguments is 0, the default value 10 should used. For example, `./prog5 0 3 0` means that there are 10 eastward-moving baboons, 3 westward-moving baboons and 10 trips. **You can assume that all command line arguments are non-negative integers, and both `e` and `w` are less than or equal to 15.**
- Your program should generate an output similar to the following:

```
    Eastward-moving baboon 3 started
Westward-moving baboon 1 started
      Eastward-moving baboon 4 started
         ......
Eastward-moving baboon 1 arrives at the west end
    Westward-moving baboon 3 arrives at the east end
         ......
Eastward-moving baboon 1 is on the rope
    Eastward-moving baboon 3 is on the rope
      Eastward-moving baboon 4 is on the rope
  Westward-moving baboon 2 arrives at the east end
        Eastward-moving baboon 5 arrives at the west end
    Eastward-moving baboon 3 completes crossing the canyon (1)
Westward-moving baboon 1 arrives at the east end
      Eastward-moving baboon 4 completes crossing the canyon (3)
Eastward-moving baboon 1 completes crossing the canyon (2)
  Westward-moving baboon 2 is on the rope
         ......
  Westward-moving baboon 2 completes all (xx) crossings and retires.  Bye-Bye!
```

  Due to the dynamic behavior of multithreaded programs, you will not be able to generate an output that is exactly identical to the above.
- The following output line tells us that the eastward-moving baboon 3 has started:

```
   Eastward-moving baboon 3 started
```

- The following output line tells us that eastward-moving baboon 1 arrives at the west end:

```
Eastward-moving baboon 1 arrives at the west end
```

- The following output line tells us that eastward-moving baboon 4 completes a crossing. The number

at the end of the message indicates the number of crossing this baboon has completed.

```
Eastward-moving baboon 4 completes crossing the canyon (3)
```

- The following output line tells us that eastward-moving baboon 3 is on the rope:

```
Eastward-moving baboon 3 is on the rope
```

- The following output lines show the effect of an important point. After eastward-moving baboons 1, 3 and 4 are on the rope, westward-moving baboon 2 arrives at the other end. Even though eastward-moving baboon 5 arrives at the west end and wants to cross the canyon in the same direction, it cannot get on the rope, because it arrives later than westward-moving baboon 2! Hence, after eastward-moving baboons 1, 3 and 4 arrive at the east end, westward-moving baboon 2 can get on the rope.

```
Eastward-moving baboon 1 is on the rope
    Eastward-moving baboon 3 is on the rope
      Eastward-moving baboon 4 is on the rope
  Westward-moving baboon 2 arrives at the east end
        Eastward-moving baboon 5 arrives at the west end
    Eastward-moving baboon 3 completes crossing the canyon (1)
Westward-moving baboon 1 arrives at the east end
      Eastward-moving baboon 4 completes crossing the canyon (3)
Eastward-moving baboon 1 completes crossing the canyon (2)
  Westward-moving baboon 2 is on the rope
```

- The following output line tells us that westward-moving baboon 2 has completed all of its required trips, indicated by the number in parenthesis, and retires:

```
Westward-moving baboon 2 completes all (xx) crossings and retires.  Bye-Bye!
```

- Note the indentation in the output. More precisely, baboons that move in the same direction are numbered as 1, 2, 3, etc. Baboon *i* has an indentation of 2×*i* spaces. For easy grading purpose, use the above output style. **Do not invent your own output, because our grader does not have enough time to digest your output.**
- You may have to execute the `Delay()` function a random number of times ro produce more realistic results.

---

## 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
./prog5 e w t       <-- 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:

- 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, and justify your claim.
- Convince me that `GetOnRope()` and `GetOffRope()`, work properly. **Make sure you will have a convincing argument for each of these questions.** An 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.
    1. Baboons on the rope only move in the same direction.
    2. Baboons must use the rope to cross the canyon. In other word, baboons cannot *magically* cross the canyon and *magically* come back. This magic only applies to the returning trip.
    3. No baboon can get on the rope without a permission.
    4. When a baboon arrives and there are baboons on the rope moving in the opposite direction, no baboon can get on the rope and move in the opposite direction. More precisely, if an eastward-moving baboon arrives and there are a number of westward-moving baboons on the rope, then no other westward-moving baboons, whether they are wait or just arrive, can be on the rope.
- You must terminate your program gracefully. More precisely, after all baboons complete `t` trips, your program terminates, and your program cannot terminate if there are baboons on the rope and waiting

on either end of the canyon.

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-main.cpp` contains the main program.
    4. File `Makefile` is a makefile that compiles the above three files to an executable file `prog5`. **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.** Note that your `Makefile` **should not** activate the visualization system.
    5. 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.**