Programming Assignment IV

100 points

Hungry Eagles

A family of eagles has one mother eagle, ***n*** baby eagles, and ***m*** feeding pots, where 0 < ***m*** <= ***n***. Each baby eagle must eat using a feeding pot and each feeding pot can only serve one baby eagle at any time. Therefore, no more than ***m*** baby eagles can eat at the same time. The mother eagle eats elsewhere and does not require a feeding pot. **All feeding pots are assumed to be empty at the very beginning and the mother eagle is sleeping.**

Each baby eagle plays for a while and eats. Before a baby eagle can eat, he must wait until a feeding pot has food. After eating, the corresponding feeding pot becomes empty and has to be refilled by the mother eagle.

The mother eagle is very tired and needs sleep. So, she goes to sleep until a baby eagle wakes her up. Should this happen, the mother eagle hunts for food and fill ***all*** feeding pots at the same time. Then, she sleeps again. It is possible that when a baby eagle wants to eat and finds out that all feeding pots are empty. This baby eagle should wake up the mother eagle. As mentioned in the previous paragraph, the mother eagle takes some time to hunt for food and fill all feeding pots. Once all feeding pots are filled, no more than ***m*** waiting hungry baby eagles can eat. Since the mother eagle does not want to wake up too often, she insists that exactly one baby eagle who found out all feeding pots being empty can wake the mother eagle up. More precisely, even though two or more hungry baby eagles may find out all feeding pots being empty, only one of them can wake the mother eagle up and all the others just wait until food will become available.

Note that a baby eagle should not wake up the mother if there are baby eagles eating and the remaining pots are empty.

After providing the ***t***-th meal, the mother eagle retires and sends all of her baby eagles away to start an independent life. Perhaps, she needs a vacation, and, therefore, the feeding cycle ends!

In this system, eagles are simulated by threads. Initially, the ***m*** feeding pots are all empty. Each baby eagle thread has the following pattern:

```
while (1) {
     Delay();             // play for a while
     ready_to_eat(...);   // get hungry
     Delay();             // eat for a while
                          // you may call Delay() multiple times
```

```
                          //    (use a random number generater)
                          //    to have a longer and more random
delay.
     finish_eating(...); // finish eating
                          // do some other thing
}
```

The mother eagle thread has the following pattern:

```
while (not time to retire) {
     goto_sleep(...);    // take a nap
     Delay();            // prepare food
                         // you may call Delay() multiple times
                         //    (use a random number generater)
                         //    to have a longer and more random
                         //    delay.
     food_ready(...);    // make food available
     Delay();            // do something else
                         // you may call Delay() multiple times
                         //    (use a random number generate)
                         //    to have a longer and more random
                         //    delay.
}
```

Here, functions `ready_to_eat()`, `finish_eating()`, `goto_sleep()` and `food_ready()` are functions written by you. (Note that Delay() is provided by the ThreadMentor library so you do not need to write it.) The functions have the following meaning:

- `ready_to_eat()` blocks the caller, a baby eagle, if all feeding pots are empty. One and only one baby eagle who finds out all feeding pots being empty should wake up the mother eagle. This function returns only if there is a feeding pot with food available to this baby eagle.
- `finish_eating()` should be called when a baby eagle finishes eating.
- `goto_sleep()` is only called by the mother eagle when she wants to take a nap.
- `food_ready()` is called when the mother eagle has finished adding food in all ***m*** feeding pots.

In this way, all controls and synchronization mechanisms are localized in these four functions and the eagle threads look very clean. Write a C++ program using the semaphore and mutex lock capability of **ThreadMentor** to simulate these activities.

Your implementation must correctly handle the following ***important situations*** among others:

- At any time, there are no more than ***m*** baby eagles eating.
- A baby eagle must wait when he wants to eat but has no free feeding pot and/or all free feeding pots are empty.
- If there is a non-empty feeding pot available, a hungry and ready-to-eat baby eagle can eat.
- No hungry baby eagle will eat using an empty feeding pot.

- At any time, a feeding pot can only be used by one eating baby eagle.
- Only one baby eagle among all baby eagles who want to eat can wake up the mother eagle.
- The mother eagle does not do her work until a baby eagle wakes her up.
- While the mother eagle is preparing food, no baby eagle can wake up the mother again until the mother comes back to take a nap.
- Before all ***m*** feeding pots are filled, no hungry baby eagle can eat.
- Once the feeding pots are refilled, the mother eagle must allow baby eagles to eat immediately. Then, she goes back to sleep.
- You must terminate your program gracefully. More precisely, if ***t*** feedings are needed, then your program **cannot** terminate right after the mother eagle refills the feeding pots ***t*** times. Instead, your program must wait until all feeding pots become empty, even though there may be baby eagles waiting for food.
- No race conditions and deadlocks should happen. This is a basic requirement.

You can only use semaphores and mutex locks. The use of any other synchronization primitives may result in a grade of zero or a significant point penalty.

Input and Output

The input to your program consists of the following:

- The number of feeding pots ***m***, number of baby eagles ***n***, and number of feedings ***t*** should be taken from the command line arguments as follows:

  ```
  ./prog4  m n t
  ```

  Thus, `prog4 8 15 12` means there are 8 feeding pots, 15 baby eagles and 12 feedings. If any one of these command line arguments is 0, the default value 10 should be used. For example, `prog4 3 0 0` means that there are 3 feeding pots, 10 baby eagles and 10 feedings. **You may assume that the command line always supply 3 integers, and these integers satisfy 0 < *m* <= *n* <= 20 and *t* > 0.**

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

```
MAIN: There are 10 baby eagles, 5 feeding pots, and 8
feedings.
MAIN: Game starts!!!!!
          ......
   Baby eagle 3 started.
          ......
Mother eagle started.
          ......
      Baby eagle 6 started.
          ......
   Baby eagle 3 is ready to eat.
          ......
```

```
Mother eagle takes a nap.
          ......
     Baby eagle 5 is ready to eat.
          ......
  Baby eagle 2 sees all feeding pots are empty and wakes up
the mother.
        Baby eagle 8 is ready to eat.
Mother eagle is awoke by baby eagle 2 and starts preparing
food.
 Baby eagle 1 is ready to eat.
          ......
Mother eagle says "Feeding (1)"
          ......
     Baby eagle 5 is eating using feeding pot 3.
          ......
Mother eagle takes a nap.
          ......
   Baby eagle 3 is eating using feeding pot 1.
          ......
     Baby eagle 5 finishes eating.
          ......
          ......
Mother eagle says "Feeding (8)"
 Baby eagle 1 is ready to eat.
          ......
Mother eagle retires after serving 8 feedings. Game ends!!!
```

Due to the dynamic behavior of multithreaded programs, you will not be able to generate an output that is exactly identical to the above.

- All output lines from the mother eagle start on column 1 and baby eagle *i*'s output lines have an indentation of *i* spaces.
- The following output line tells us that baby eagle 3 has started:

```
Baby eagle 3 started.
```

- The following output line tells us that baby eagle 5 is ready to eat:

```
Baby eagle 5 is ready to eat.
```

- The following output line tells us that baby eagle 5 is eating using feeding pot 3:

```
Baby eagle 5 is eating using feeding pot 3.
```

- The following output line tells us that baby eagle 2 sees all feeding pots being empty and wakes up the mother eagle.

```
  Baby eagle 2 sees all feeding pots are empty and wakes up
the mother.
```

- The following output line tells us that baby eagle 5 finishes eating:

```
Baby eagle 5 finishes eating.
```

- The following output line tells us that the mother eagle has started:

```
Mother eagle started.
```

- The following output line tells us that the mother eagle takes a nap:

```
Mother eagle takes a nap.
```

- The following output line tells us that baby eagle 2 wakes up the mother eagle and the mother starts preparing food:

```
Mother eagle is awoke by baby eagle 2 and starts preparing
food.
```

- The following output line tells us that the mother eagle has finished food preparation. The number in () is the feeding count. This is the third feeding.

```
Mother eagle says "Feeding (3)"
```

- The following output line tells us that the required number of feedings have completed and the mother retires. The feeding game ends here. **But, the mother should keep running until all baby eagles finish eating even though the mother decides to retire. In other words, your system should end gracefully. And the following output has to be the last message printed by your program.**

```
Mother eagle retires after serving 8 feedings. Game is over!!!
```

- Please note the indentation in the output. For easy grading purpose, use the above output style. **Do not invent your own output, because our grader does not have enough time to handle output that deviates from this. Also note that each output line from a thread MUST appear on the same line rather than separated into multiple outlines. Otherwise, your program is considered to be incorrect and you risk a significant deduction.**

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
./prog4 m n t       <-- test your program
```

4. This procedure may be repeated any 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 may incur a significant point penalty.
6. A `README` file is required.
7. Note we use "**noVisual**" option to make your program.

Compiling and Running Your Programs

If we cannot compile your program due to syntax errors, wrong file names, or your program compiles successfully but fails to run, we cannot test your program, and you may receive 0 point. Therefore, before submitting your work, make sure your program can compile and run properly.

1. Programs that do to compile may receive 0 points. This includes any reason that causes an unsuccessful compilation, including missing files, incorrect filenames, syntax errors in your programs, and so on. Double check your files before you submit then, since the grader does not have time to modify individual programs in a class this large. Note again: Unix filenames are case sensitive.
2. Programs that compile but do not run may receive 0 points. This usually means that you have attempted to solve the problem to some degree but you failed to make it work properly.
3. A meaningless or vague program may receive 0 points even though it compiles successfully. This usually means your program does not solve the problem but serves as a placeholder or template that only compiles and run but does not implement the required functionality.

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 should contain appropriate concise and to-the-point comments. Do not write a novel!
3. Your program should have good indentation.
4. Do not use global variables! Only semaphores, mutex locks and some common control variables such as counters can be global.

Program Specification

Your program must follow exactly the requirements of this programming assignment. Otherwise, you may receive no points 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. There should be no busy waiting, race conditions, and deadlocks.
4. Any other significant violation of the given program specification.
5. 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.
6. 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:

> Convince me that your program works properly. More precisely, explain how your program ensures the following. Make sure you will have a convincing argument for each item. Note that an argument such as *because a semaphore is 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 a semaphore or semaphores.
>
> 1. At any time, there are no more than *m* baby eagles eating.
> 2. A baby eagle must wait when he wants to eat but has no free feeding pot and/or all free feeding pots are all empty.
> 3. If there is a non-empty feeding pot, a hungry and ready-to-eat baby eagle can eat.
> 4. No hungry baby eagle will eat using an empty feeding pot.
> 5. At any time, a feeding pot can only be used by one eating baby eagle.
> 6. Only one baby eagle among all baby eagles who want to eat can wake up the mother eagle.
> 7. The mother eagle does not run until a baby eagle wakes her up.

8. While the mother eagle is preparing food, no baby eagle can wake up the mother again until the mother goes back to take a nap.
9. Before all *m* feeding pots are filled, no hungry baby eagle can eat.
10. Once the feeding pots are refilled, the mother eagle must allow baby eagles to eat. Then, she goes back to sleep.
11. You program terminates gracefully. More precisely, if *t* feedings are needed, then your program cannot terminate right after the mother eagle refills the feeding pots *t* times. Instead, your program must wait until all feeding pots become empty, even though there may be baby eagles waiting for food. How do you implement this requirement?

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 of less than two printed pages. 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:

    - File **`thread.h`** that contains all class definitions of your threads.
    - File **`thread.cpp`** contains all class implementations of your threads.
    - File **`thread-support.cpp`** contains all supporting functions such as **`ready_to_eat()`**, **`finish_eating()`**, **`goto_sleep()`** and **`food_ready()`.**
    - File **`thread-main.cpp`** contains the main program.
    - File **`Makefile`** is a makefile that compiles the above three files to an executable file **`prog4`**. Since we will use lab machines 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 low grade. So, before submission, make sure you have the proper file structure and correct makefile. Note that your `Makefile` should not activate the visualization system.

- File `README`.

Note also that without following this file structure your program is likely to fall into the *compiles but does not run* category, and, as a result, you may get a low grade. Therefore, before submission, check if you have the proper file structure and a correct makefile.

Always start early, because it may be difficult to get the turnaround time you need on questions that arise at the last minute.