HUNGRY LIZARD CROSSING

OVERVIEW

This assignment will provide experience with multi-threaded applications and the synchronization primitives (mutex lock and semaphore, specifically) that are required for their proper use. You will create a program that synchronizes access between threads to a shared data structure.

THE SCENARIO

My house has a herd of lizards and two cats. The cats sleep most of the time, and their favorite toy is the lizards. The lizards live in the sago palm but find food across the driveway in the monkey grass. If too many lizards try to cross the driveway at once, the cats will see them and "play" with them. Your job is to synchronize the lizards crossing the driveway to prevent any lizards from becoming cat toys.

THE PROGRAM

Write a program in C++ to create *N* threads, each representing a lizard, and two other threads, each representing a cat. Each lizard thread will sleep for some random amount of time and wake up hungry. The hungry lizard thread will attempt to cross the driveway safely, as shown in the pseudo-code below. Crossing the driveway will take some number of seconds. Once on the other side of the driveway, the lizard will eat for some random amount of time in the monkey grass. After eating, the lizard will return home to the sago palm as soon as it safely can and sleep again. Each of the two cat threads will periodically sleep. When awake, they will look at the driveway to check on the lizards. If a cat sees too many lizards, it will play with them, causing the entire program to terminate. The two cat threads will not differ in their behavior, but they will vary in who is awake to check on the lizards and which sleeps.

Use one or more locks and semaphores (do not use monitors) to control access to the driveway (the shared resource). Make sure your implementation follows these rules:

- Do not allow too many lizards to cross the driveway at once.
- Do not use busy waits to control lizards.
- Allow the maximum possible number of lizards to cross simultaneously.

STARTER CODE & CHANGES

You will be provided with a single file that contains a beginning implementation of the project solution. The file `lizards.cpp` contains many comments and hints about completing the project. The file can be downloaded from the Canvas course shell. Make changes and additions to the file, but no deletions. You must not create any other source code files. Functions marked 'Completed' must remain unchanged.

In the source code, make in-line comments with your team members' initials to mark the lines of sample code you changed or added. You will not be making any deletions. If you do not include comments in your source code to make it more readable, points will be deducted. Here is an example of how a team with members named Pupil Uno and Lerner Zwei would mark a change:

Original code: `int counter = 1;`

Modified code: `int counter = 0; //PU LZ`

Implementation Suggestions

Each lizard thread will follow an algorithm similar to the one given below. The algorithm is in pseudo-code and **NOT IN C/C++**. Do not attempt to make this code run as-is. However, the names of the functions provided in `lizards.cpp` will be very familiar when you read the code and comments.

```
while (world has not ended)
    sleep for up to MAX_LIZARD_SLEEP seconds
    wait until [sago -> monkey grass] crossing is safe
    cross [sago -> monkey grass]
        it takes up to CROSS_SECONDS seconds to cross
    eat in the monkey grass
        it takes up to MAX_LIZARD_EAT seconds to eat
    wait until [monkey grass -> sago] crossing is safe
    cross [monkey grass -> sago]
        it takes up to CROSS_SECONDS  seconds to cross
```

Each cat thread will follow an algorithm that entails the action described above. You will see the code of the cat implemented in the starter code. Access to variables shared between the threads must be protected. Lock(s) and semaphore(s) must be properly initialized, used, and destroyed.

Report

The report *analysis.pdf* is a PDF-converted document that must contain the following information:

- a short description of the problem and how your code changes solve it
- a discussion of all the changes made to the code
- a table with results from multiple runs with different run times and number of lizards and cast. Use constant WORLDEND in the source code to control the length of the simulation.
- issues encountered in developing the solution

TABLE 1 SAMPLE RESULTS TABLE

WORLDEND (s)	Maximum Number of Lizards Crossing	Lizards safe?
30	4	Yes
180	4	Yes

Try to confirm the maximum number of lizards crossing by printing the appropriate counter values as a sum to the screen.

Extra Credit

An extra credit challenge is to prevent bidirectional travel. If one or more lizards are crossing in one direction, other lizards wanting to cross in the opposite direction must wait. This addition is not simple and could take a great deal of time to complete. Your extra credit solution will only be graded if a README.txt file is also submitted, indicating that you completed the extra credit portion. The way to toggle between the bidirectional and unidirectional modes is by changing the UNIDIRECTIONAL variable in the provided file `lizards.cpp`. However, to avoid possible corruption of the bidirectional solution, write the code for the unidirectional solution in a separate file named `lizardUni.cpp`, and submit both files as your solution.

Deliverables

Your project submission should follow the instructions below. Any submissions that do not follow the stated requirements will not be graded.

1. Follow the submission requirements of your instructor as published on *eLearning* under the Content area.
2. You must submit the following files for this assignment:
    a. `lizards.cpp` (the source code file, UNIDIRECTIONAL set to 0 and WORLDEND to 180)
    b. `lizardsUni.cpp` (only submit this if you completed the extra credit assignment)
    c. analysis.pdf (the results from an experiment and the changes in the file)
    d. Makefile
    e. README.txt if you completed the extra credit unidirectional option or completed a partial solution of the problem

Testing & Evaluation

Your program will be evaluated on the department's public Linux servers according to the steps shown below. Notice that warnings and errors are not permitted and will make grading quick!

1. Program compilation with Makefile. The options *–g* and *–Wall* must be enabled in the Makefile. See the sample Makefile that I uploaded in *Canvas*.
    - If errors occur during compilation, there will be a substantial deduction. The instructor will not fix your code to get it to compile.
    - If warnings occur during compilation, there will be a deduction. The instructor will test your code.
2. Perform several runs with input of the grader's own choosing. At a minimum, the test runs address the following questions.
    - Have threads been created to simulate lizards and cats?
    - Are all race conditions addressed?
    - Are the maximum number of lizards allowed to cross also able to cross the driveway?
    - Is the main thread waiting for the termination of the lizard and cat threads?
    - Does the solution avoid busy loop implementations?

Due Date

The project is due, as indicated in the schedule in the syllabus and calendar in *Canvas*. Upload your complete solution to the dropbox. I will not accept submissions emailed to me or the grader. Upload ahead of time, as last-minute uploads may fail.

Grading

This project is worth 100 points. You will earn ten additional points for submitting the correct unidirectional solution provided in `lizardsUni.cpp`. The rubric used for grading is included below. Keep in mind that there will be deductions if your code does not compile, has memory leaks, or is otherwise poorly documented or organized. The points will be given based on the following criteria:

Correct Submission Format	**Perfect**	**Deficient**		
Canvas	5 points individual files have been uploaded	0 points files are missing		
Compilation	**Perfect**	**Good**	**Attempted**	**Deficient**
Makefile	5 points make file works; includes the clean rule	3 points missing clean rule	2 points missing rules; doesn't compile project	0 points make file is missing
compilation	10 points no errors, no warnings	7 points some warnings	3 points many warnings	0 points errors
Documentation & Program Structure	**Perfect**	**Good**	**Attempted**	**Deficient**
documentation & program structure	10 points follows documentation and code structure guidelines	7 points follows mostly documentation and code structure guidelines; minor deviations	3 points some documentation and/or code structure lacks consistency	0 points missing or insufficient documentation and/or code structure is poor; review sample code and guidelines
Simulation Program	**Perfect**	**Good**	**Attempted**	**Deficient**
creates threads	10 points correct, completed	7 points minor errors	3 points incomplete	0 points missing or does not compile
synchronizes threads to limit access to shared resource using semaphores (driveway)	15 points correct, completed	11 points minor errors	4 points incomplete	0 points missing or does not compile
avoids race conditions	15 points correct, completed	11 points minor errors	4 points incomplete	0 points missing or does not compile
joins with all threads at the end of execution	10 points correct, completed	7 points minor errors	3 points incomplete	0 points missing or does not compile
Report	**Perfect**	**Good**	**Attempted**	**Deficient**
problem description & results from multiple runs	10 points correct, completed	7 points minor errors	3 points incomplete	0 points missing
discussion of changes	10 points correct, completed	7 points minor errors	3 points incomplete	0 points missing
discussion of issues encountered	5 points correct, completed	3 points minor errors	2 points incomplete	0 points missing

Comments

The provided code is designed to accept an optional argument on the command line. If the `-d` option is given, debugging output statements will be printed during execution. I suggest using the debugging option while developing your program.