University of West Florida – Electrical and Computer Engineering

Systems and Networks I COP 4634

Project 3: Lizards

Due By: 11/10/2019

Aurburn Jones

Derick Masters

**Abstract**

This program is designed to demonstrate the necessity and methodology for synchronizing threads which share access to a common resource. To accomplish these tasks, the similar but differing structures of the thread mutex and semaphores will be used to control execution of lizard simulation.

**Overview**

The basic scenario of the simulation is that of a rudimentary prey, lizards, and predator, cats, relationship. Serving as the backdrop for the simulation is Dr. Reichherzer’s driveway which is flanked on either side by Monkey Grass and Sago Palms. The lizards in question enjoy hiding in the Sago Palms but must periodically cross the driveway to reach the Monkey Grass where they eat. Cats, being fickle hunters, will only “play” with the lizards when the lizards have amassed in great enough numbers to pique their interest. Lizards, being averse to the cats’ idea of “playing”, must in turn ensure that they limit their crossings such that the interest threshold is not surpassed.

Each predator and prey will be represented using a thread and defining the behavior of both animals will be simple algorithms which take into account the parameters discussed above. Presented in Figure 1 below, using pseudocode, is the algorithm which determines the behavior of each lizard thread.

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

**Figure 1:** Lizard behavior algorithm in pseudocode.

As can be seen above, the motivations for each lizard thread are quite straightforward. In between bursts of activity, such as crossing the driveway or eating, the lizard will sleep and keep cycling through each step until the simulated world comes to an end. Even simpler, is the pseudocode representing the cats’ behavior shown below in Figure 2.

while (world has not ended)  
 sleep for up to **MAX\_CAT\_SLEEP** seconds  
 check if number of lizards crossing > **MAX\_LIZARD\_CROSSING**  
 if true, “play” and end simulation

**Figure 2:** Cat behavior algorithm in pseudocode.

From the behavior of the cat, it is plain to see that should the lizards avoid kitty playtime the simulation will continue to execute until the full runtime has been reach. Otherwise, should the lizards become unsynchronized one of the cats pounce, the simulation will terminate prematurely and the full runtime will not be reached.

**Procedure**

Running the simulation is very simple, after compiling the program using the provided makefile the following command should be issued:

./lizard

Optionally, if it is desired to observe the state of each thread in real time, the modified command

./lizard -d

can be used to run the simulation in ‘debug’ mode. In its default state the program will create twenty lizards, two cats, and run for thirty seconds. To test the synchronization of the lizards and ensure that the lizards are in fact avoiding crossing in too great a number, the total runtime of the simulation will be gradually increased and the maximum number of lizards crossing will also be increased for each runtime. This should prove proper synchronization as the longer the simulation runs, the more likely excess lizards are to be caught.

**Discussion**

In this scenario, the shared resource is the driveway itself and access should be limited to the maximum number of lizards which can cross without garnering the attention of the cats. Therefore, in order for a lizard thread to determine if a potential crossing is safe it must know how many lizards are currently crossing and should it decide to cross then that number must be increased accordingly.

Keeping track of trips in either direction are the two provided global variables numCrossingSago2MonkeyGrass and numCrossingMonkeyGrass2Sago whose purposes are self-evident. Since each variable will be incremented and decremented by threads as they enter and exit the driveway they must be protect during modification. This is accomplished by using additional pthread mutex variables sago2grass and grass2sago which guard their corresponding variables from potential corruption.

The two mutex variables discussed above ensure that the two global counter variables cannot be inadvertently damaged by multiple lizard threads seeking to update their crossing status, but they do not address the issue of ensuring that the number of crossing lizards does not exceed MAX\_LIZARD\_CROSSING. This can be done using a single standard semaphore structure, named sem\_liz, which has been initialized to allow for MAX\_LIZARD\_CROSSING threads to cross the driveway at any given time.

To summarize in context of the lizard thread behavior, the following occurs: if a lizard wishes to cross in either direction it must test the value of sem\_liz and then use the appropriate mutex, sago2grass or grass2sago, to modify the correct trip counter. The semaphore, sem\_liz, is the main mechanism for ensuring that the number of crossing lizards is always kept below the designated threshold. In all, the modifications made to the program include:

1. Inclusion of semaphore.h
2. Creation of mutex/semaphore variables sago2grass, grass2sago, and sem\_liz
3. Completion of main() as indicated by the provided comments
4. Change of the \*(int \*) to (intptr\_t) typecast is lizardThread and catThread
5. Implementation of the lizard algorithm according to the provided pseudocode
6. Use of sem\_wait() in sago\_2\_monkeyGrass\_is\_safe() and monkeyGrass\_2\_sago\_is\_safe()
7. Use of pthread\_mutex\_lock()/pthread\_mutex\_unlock() in cross\_sago\_2\_monkeyGrass() and cross\_monkeyGrass\_2\_sago()
8. Use of sem\_post() in made\_it\_2\_monkeyGrass() and made\_it\_2\_sago()

Illustrating the success of the synchronization are the result for runtimes from thirty to one hundred eighty seconds, shown in Table 1 below.

**Table 1:** Simulation results with various runtimes.

|  |  |  |
| --- | --- | --- |
| **WORLDEND (s)** | **Maximum Number of Lizards Crossing** | **All Lizards Safe?** |
| 30 | 4 | Yes |
| 30 | 8 | Yes |
| 60 | 4 | Yes |
| 60 | 8 | Yes |
| 90 | 4 | Yes |
| 90 | 8 | Yes |
| 120 | 4 | Yes |
| 120 | 8 | Yes |
| 150 | 4 | Yes |
| 150 | 8 | Yes |
| 180 | 4 | Yes |
| 180 | 8 | Yes |

As can be seen in the above table, all test runs terminated normally, and no lizards were caught by the cats.

**Conclusion**

The results discussed above demonstrate that thread synchronization was successfully implemented in the multi-threaded simulation. As was done in the previous project, pthread mutex variables were used to protect specific variables which must be modified by multiple threads. Adding to this concept was the use of semaphores which allowed for a specific number of threads to access a shared resource, i.e. the driveway, at any given time. Semaphores efficiently helped to synchronize the threads and mutex variables helped to protect shared memory from improper modification.