Adam Mooers
Luke Kledzik
System and Networks 1
Project 3

Problem Description and Solution Summary

A finite number of threads (lizards) must share a limited resource (the driveway). If more threads use the resource than it can handle, the simulation terminates at a loss to the programmer. In its default state, the lizards operate independently and conflicts occur at random intervals as the lizards cross the driveway. The key change is using a counting semaphore to control which lizards have access to the driveway at a given time. The semaphore seamlessly handles mutual exclusion, progress, and bounded waiting. Lizards are not left stranded on either side of the road. mutex locks complement the semaphore by providing a way for threads to access shared counters.

Discussion of changes made to the code.

The code uses two mutex locks. liz_lock controls access to the lizard id number, numCrossingSago2Monkeygrass, and numCrossingMonkeyGrass2Sago. cat_lock controls access to the cat id number. The code could have used a single lock. This would have, however, prevented the cat threads and lizards threads from being created asynchronously. Since the lizard threads go on to use their lock for other purposes, the cat threads may stop progressing in the worst case. This would result in an unfair simulation.

A semaphore allows the lizards to cross in the order they arrive, similar to cars at an all-way stop sign. This keeps lizards from waiting too long. Unlike cars at a stop sign, a lizard that has just crossed must alert the next lizard that is waiting. There are six references to it in the code. First, the semaphore is initialized in main to the maximum number of lizards which can cross the driveway. Second, when a lizard seeks to cross the road it calls a sem_wait. When the lizard arrives on the opposite side, sem-post is called to allow the next waiting lizard to cross. Since the lizard crosses the road twice, two sem_posts and two sem_waits are called. Lastly, the semaphore is destroyed at the end of main.

Translating the pseudocode algorithm described in the project description to code in the lizard thread function was straightforward. All the pseudocode functions were already mapped to c functions.

Lastly, some glue logic for creating and destroying threads was inserted at the beginning and end of main.

The following Bash command simplified confirming the total number of lizards crossing the road at a given time:

./lizards -d | grep Lizards

Simulation Trials and Outcomes

WORLDEND (s)	Maximum Number of Lizards Crossing	Lizards safe?
30	4	Yes
180	4	Yes
500	4	Yes

Table 1: Simulation Times for Four Lizards and Two Cats

Issues encountered in developing the code

There were two major problems. The first was forgetting to sem_post in the made_it_to_sago function. Without sem-posting, the first lizard to finish the round trip would not pass on the message. Eventually, all the lizards were waiting at the sago palm, each waiting for the signal to cross.

The other issue was with the lock while creating cat and lizard threads. Originally, no lock was used when the thread was called. The problem is that the id number used by the thread sometimes changes in the main thread before the new thread can read it. The end result was lizards sporadically having the same id.