CS444 Concurrency Assignment 3

Mark Bereza and Morgan Patch

PROBLEM 1

Design

For Problem 1, the desired functionality was achieved using three different locking mechanisms. First, a semaphore is used to keep track of how many threads have access to the shared resource. Each time a thread ”acquires” the resouce, the semaphore counter is decremented by 1. Similarly, whenever a thread is finished with said resource, the semaphore is incremented by 1. When the semaphore’s value reaches zero, threads attempting to access the resource will block. The last thread to successfully access the thread will flip a global flag, the second locking mechanism, that will prevent any threads for accessing the resource or touching the semaphore. This flag is flipped back once all three threads that had access to the resource release it, thus bringing the semaphore’s counter back to 3. The final locking mechanism was a mutex used to ensure that the semaphore value was not altered between when the value was queueried and when it gets printed. Thus all threads attempting to increment the semaphore, decrement the semaphore, or query the semaphore’s value must first acquire this lock. The lock is released immediately after the increment/decrement/query operation completes.

Running the Program

The program can be ran by first running `make` and the running the problem1 program produced by make. If a numerical argument is provided to the program, that many threads will be spawned. If no arguments are provided, the program will spawn a default five threads. To terminate the program, simply send it an interrupt signal using Ctrl+C.

Ascertaining Correctness

The solution to Problem 1 was determined to be working correctly by simply inserting print statements whenever the semaphore’s state changes (indicating the new semaphore value and the thread responsible for the change) and running the program. Just looking at the printed output will show that no more than three threads ever have access to the shared resource simultaneously and as soon as the third thread accesses it, the resource is locked until all three threads release it. Outside of this locked state, threads can freely access the resource so long as the semaphore’s value is above zero. Thus, the solution achieves the functionality outlined in the problem description.

PROBLEM 2

Design

For Problem 2, the functionality was implemented using an extended mutex. A list of locks is maintained which maps locking threads to the type of lock they have; to determine if a type of lock is held, check if it is in the value set. A queue of waiting threads is also held for each type of lock. When a thread attempts to gain the lock, it checks if it can according to the below rules. If so, it locks the mutex by the type of lock it wants to hold; if not, it "parks" itself, losing control and de-scheduling itself until another thread unparks it. Whenever a thread unlocks the mutex, it checks if there's another thread that can lock the mutex next; if so, it unparks that thread so that it can lock the mutex.

The rules for locking a thread are as follows: A READ operation can begin if there are no DELETE operations occuring; an INSERT operation can begin if there are no INSERT or DELETE locks; a DELETE operation will block any new threads from locking the mutex, but cannot begin if *any* operation is occuring.

The rules for unlocking threads are as follows: Unlocking a READ will only look for available DELETEs, as READs and INSERTs have been happening; unlocking an INSERT will look for available INSERTs and DELETEs; unlocking a DELETE will look for any threads that can be unlocked.

Running the Program

All following commands are in the `Problem2` directory.

If Maven is installed on the computer you are running, simply run `mvn package`, then `java -jar target/Concurrency3-1.0.jar`. If not, then the following commands will compile and run the tool:

```
mkdir target/
javac -d target -source 1.8 -sourcepath src/main/java src/main/java/os2/group40/concurrency3/*
jar cef os2.group40.concurrency3.ProblemTwo target/Concurrency3.jar -C target os2
java -jar target/Concurrency3.jar
```

If no arguments are provided, the program will spawn a default ten threads. To terminate the program, send it an interrupt signal using Ctrl+C.

Demonstrating Correctness

The sample application spawns a number of threads (default 10) which lock the mutex, "work" for 1-500 ms, unlock the mutex, then sleep for 1-2000 ms. While no formal proof of correctness can be given, the output on STDOUT should be sufficient to demonstrate that the application works according to the above rules; it prints before and after each lock and unlock with the thread number and timing information. This will show that the locks only occur when the mutex is available, and unlocking one thread causes another to lock in its place.