Concurrency 3: Categorical Mutual Exclusion

CS 444: Operating Systems II

Fall 2017

Abstract

In our third concurrency assignment, we tackle two questions that explore categorical mutual exclusion. The first examines how best to share access to a common resource such that each has full access to the resource, but when there are three threads using the resource, all three must leave before any new processes can gain access. The second deals with how best to allow different types of processes - searchers, inserters, and deleters - to all work concurrently on the same singly-linked list. This document presents our design decisions and testing plan for the two above problems.

Jacob Mahugh

Alannah Oleson

November 26, 2017

1 Solution Description and Design Decisions

1.1 Problem 1

Research into the context of the question found that problem 1 is actually the Sushi Bar problem from The Little Book of Semaphores (one of the resources listed on the class website). Given this, we were able to find example pseudocode online that described various solutions to the problem.

A key part of this problem is the use of semaphores, rather than mutexes, to accomplish mutual exclusion between processes. The difference between the two structures lies in the way that they let go of the locks they create. While a mutex can only be unlocked by the thread that created it, a semaphore can be unlocked by any thread that has the same parent process as the thread that locked the semaphore in the first place. In the context of Problem 1, this allows each of the threads to signal to the others that they want control over the shared resource without hanging when more than three processes are using it.

Our solution starts by spawning 12 different threads (one for each district of Panem in the Hunger Games - see comments in the file for more context). Each thread calls its starting function, which kicks off an infinite while loop. Inside this loop, the thread tries to take control of the semaphore that represents the shared resource. If there are more than three threads already using the resource, it hangs here until it can get access. Otherwise, the thread takes a spot at the shared resource and "uses" it for a random number of seconds (i.e., it sleeps while it has the lock). When it wakes up, the thread unlocks its part of the semaphore and checks to see if the resource is currently being used by three threads. If it is, it makes sure that all the threads that are using the semaphore unlock their holds before it cedes the resource. If not, it will only release its own lock and not worry about the others.

1.2 Problem 2

This problem had a pretty straightforward solution. The key to this problem was realizing that we need to identify what kind of process had the mutex; an inserter or a deleter. One of these mutexes would be for the inserters, and the other for the deleters. This would allow the searchers to execute concurrently with the inserters.

The general construction of the program is to initialize a queue, then spawn threads to interact with it. In this program we use the sys/queue.h library list as a singly linked list (i.e. we never travel backwards). We declare the list head as a global variable so that it is shared across the threads. We also declare the mutex as a global variable, and create two identifier variables (isInsLock and isDelLock). These variables will allow our other processes to see what kind of process is holding the lock.

After that, we initialize the queue with three nodes and spawn threads for four searchers, three inserters, and two deleters. These numbers were specifically chosen to minimize the chance of creating an empty queue. Each of these threads have their own thread specific function that they are told to execute.

The searcher function follows this algorithm:

1. Block while the isDelLock variable is set.
2. When isDelLock is not set, generate a random node position based off the list length.
3. Iterate through the list.
4. If this is the randomly selected node, read it (pring in this program)!
5. Sleep between 0 and 4 seconds.
6. Rinse and repeat.

Note that searchers will run concurrently with each other and with inserters. The only thing they block on is the lock identifier for deleters being set. I gave them the shortest sleep time since they do not block anything and are very quick to execute.

The inserter function follows this algorithm:

1. Block while the lock is in use (isInsLock or isDelLock is set).
2. When available, grab the lock (NOTE: I triple check to one-up Santa).
3. Set the isInsLock identifier variable.
4. Create and add a node.
5. Release the lock and unset the isInsLock identifier variable.
6. Sleep between 0 and 6 seconds.
7. Rinse and repeat.

Note that this means inserters will not run concurrently with each other or with a delete, but will run with searchers. I gave the inserters a medium sleep length to help make sure the queue does not become empty.

The deleter function follows this algorithm:

1. Block while the lock is in use (isInsLock or isDelLock is set).
2. When available, grab the lock (NOTE: I triple check to one-up Santa).
3. Set the isDelLock identifier variable.
4. Determine what node to delete (a random integer modded by the list length).
5. Iterate through the list.
6. If this is the randomly selected node, delete it!
7. Release the lock and unset the isDelLock identifier variable.
8. Sleep between 0 and 8 seconds.
9. Rinse and repeat.

Note that deleters will not allow anything to run concurrently with them! I gave them the longest sleep time to avoid empty queues and to make things a bit quicker since deletion blocks all tasks.

This certainly is not the only way to implement a solution to this problem, but this solution seemed the most intuitive to me.

2 Testing Details

First, run `make all` so that the PDF and both executables are created.

2.1 Problem 1

The Makefile compiles an executable called `prob1`. Run `prob1` to start the program. You will see a message of the form "Tribute X raids the cornucopia for Y minutes" appear on the screen. This will signify that the Xth thread has taken a lock on the shared resource (the cornucopia). The Y represents how long the thread will sleep after locking to simulate "using" the resource. After Y seconds, you will see a corresponding message of the form "Tribute X leaves the cornucopia with Z useful items". This represents the thread releasing its lock on the semaphore so that other threads can use the resource. This program will run until the user stops it with Ctrl-C.

To ensure correctness, inspect the output to ensure that a thread (tribute) is never releasing its thread before it locks it, since that would indicate trouble with the semaphore. In addition, ensure that when three threads are using the resource (i.e. there are three "raid" statements in a row without any "leave" messages in between), there are then three corresponding "leave" statements before the next "raid" statement. This shows the correctness of the case where the shared resource is being used by three threads at a time and must empty itself completely before accepting new thread locks.

2.2 Problem 2

The Makefile compiles an executable creatively named `prob2`. Run `prob2` to start the program. You will see three kinds of print statements: searchers, inserts, and deletes. Here is a selection of some example prints:

> I1 inserted 5 | InsLock:1 - DelLock:0
>
> D2 deleted 4 | InsLock:0 - DelLock:1
>
> S4 searched 1 | InsLock:0 - DelLock:0

The first letter number combination will tell you what thread is printing. The letter corresponds to inserter, deleter, or searcher, and the number is what number thread that is. The next segment is the action followed by the ID of the node. The ID of the node is determined by the length of the list when the node is inserted. On the other side of the pipe is the status of the identifier variables at the time of printing. This will give information about what process holds the lock, if any.

In order to probably determine correctness, it is important to realize that **I initialize the queue with three nodes in it already**. To ensure correctness, look for these things in the output:

- No searcher or inserter prints that DelLock is 1 (set).
- No deleter prints that InsLock is 1 (set).
- At least one searcher prints that InsLock is set (NOTE: this may take awhile before it happens).
- Inserters insert a node with an ID equal to the last inserted ID plus one minus the number of deletions since the last insert.
- Searchers never print an ID that has not been inserted.
- Deleters never delete an ID that has not been inserted.