CS-377 Parallel Programming
Assignment 3 Lab Report

Eliot Cowley

April 5, 2015

1 Answers to Questions in Description

Even though this solution doesn't suffer from deadlock, it does suffer from something else. What?

After a call to `upc_lock_attempt()`, a null strict reference is implied, which returns a 1. This means that reordering of memory references and/or operations is not allowed. Even if reordering wouldn't affect the final result and would optimize performance, it is not allowed after a call to `upc_lock_attempt()`, and therefore this solution may suffer from a performance hit.

Unfortunately, [replacing the `upc_lock_attempt()` statements with `upc_lock()` statements] will introduce the possibility of deadlock into the program. Why?

`upc_lock_attempt()` tries to access a lock, and if it is blocked, it returns and indicates that it didn't acquire the lock so that the process can go on to do something else. In this way, deadlock is avoided. However this is not the case with `upc_lock()`, which will simply block until the process can access the lock, which could result in deadlock.

2 Experience

This lab was an interesting one, in that it wasn't quite as straightforward as I initially thought it would be. I had some trouble with `upc_lock()` at first because I didn't realize that its return type was void, and I had its result being output to variables `get_left` and `get_right`, having only changed the original code from `upc_lock_attempt()` to `upc_lock()`. I tried many different things to try to figure out what was wrong before Professor Smith solved my problem. Looking back, it should really have been obvious that its return type is void, since `upc_lock()` simply blocks until it can access the lock, at which point it obtains said lock and continues.

The first two implementations of the Dining Philosophers problem were fairly straightforward. They were simply a matter of identifying which thread was running, and picking up their forks in the appropriate order. The third one was more challenging, however. For a time I struggled over how exactly I would implement the "waiter."At first, I thought that maybe I would have one thread be the waiter process that seats all the other philosopher threads, but I decided against it because that would mean that in order to run the program, the user would have to remember to add a thread for the waiter process, which I didn't think was ideal. So I decided to have each philosopher essentially seat herself, by giving them access to a shared array of seats at the table. I also created an `int` value `num_seated` that kept track of how many philosophers were currently sitting down. A philosopher would first check if `num_seated` was less than the number of available seats (`THREADS-1`), and if it was, the philosopher would sit down. In addition to the `state`s that each philosopher had, I gave them each a `table_state`, `STANDING` or `SITTING`, which kept track of whether a philosopher was seated at the table or not. Finally, in order to avoid race conditions and keep the shared value `num_seated` accurate, I put a lock on the table that a philosopher had to access before she could increment, decrement, or otherwise access `num_seated`.

After doing this assignment, I can see why it would be preferable to use `upc_lock_attempt()` rather than `upc_lock()`. If you use `upc_lock()`, you have to remember to use `upc_unlock()` or other threads won't ever be able to access the lock and the program will suffer from deadlock. I ran into this problem when I was implementing number three–a philosopher that was waiting to sit at the table would obtain the `table` lock, find that there were no seats available, and never release the lock. I fixed this by adding an `else` case where the philosopher would release the lock if there were no seats available. Using `upc_lock_attempt()` would have made things easier, because if a philosopher were unable to obtain the lock, she would simply continue running and deadlock would be avoided.