CMPUT 379 Assignment 4

name: Waridh 'Bach' Wongwandanee
ccid: waridh
sid: 1603722

Objectives

From my point of view, we learned how to use mutexes to protect critical sections of the code (shared resources) and synchronize threads. We also needed to prevent deadlocks using one of the methods we learnt in lecture. The method for deadlock prevention that was used in this assignment was by only running the program when all resources are available

Design Overview

This program was built extremely robustly, so let me go point by point from the beginning of the program.

1. This time, I created a header file for the program that has structure declaration and function prototypes.
2. I use structs that contain a std c++ map/unordered_map to hold many field. This allows our code to be very dynamic, with fast lookup time.
3. There are error handling everywhere I could have thought to place it in
4. We parse the input text file with some error handling, but also some dynamic solutions. If the user decided to have multiple lines with the resource allocation tag, we are able to add all of them, even if some has repeating resource names. We also do not care about what type of whitespaces are used to separate the tokens being used for input.
5. We read through the input file twice. The first time to get the resources, the second the create the task threads.
6. To synchronize all the threads, we are using a combination of pthread_mutex, pthread_barriers, and pthread_cond. Barriers were used to keep the program moving together, mutexes to keep critical sections safe, and cond for waiting for resources to be returned.
7. There is a global debug flag that will provide users with more information.
8. The monitor thread was created before the rest of the task threads, but it used a barrier to halt start before all the task threads are ready as well. If the debug flag is on, the monitor thread will also print out the resources information, showing the max amount along with the amount that is currently being held.
9. The task threads are created and synchronized using barriers. After all the task threads are ready, it can access the main loop, where it will wait for all its required resources to be ready before it changes state from wait to run. The running task output will be printed to stdout after the program finishes its idle stage which is placed at the end of the loop in my implementation.
10. The waiting logic used for the task thread when there isn't enough resources uses cond. In high level, the thread will go into wait until a thread that was running has completed running. The idea is that when a task has finished running, some resources will be released, and all the waiting threads should check if the resources it needs has been made available. We completed this using pthread cond for signaling to stop waiting, and pthread mutex as our binary semaphore.

11. The termination is synchronized again using barriers. The output that shows the information of the threads are actually being outputted by the threads themselves. This is not the most optimal way to print out details about things in order, especially since you could access this information in the main thread, however, I saw this as an opportunity to practice with more thread synchronization. It's a little random, but we are relying on pthread mutex to allow even progress to let the thread with the correct index access the section. As the threads are able to create output, there will be less and less threads trying to access the index. This is how I was able to make the threads print their final outputs in order. In the future, I would try to implement a more efficient synchronization method for syncing the final output, however, this little extra feature does not seem to affect main objective being asked by the assignment.
12. We are using times.h to get the millisecond count of the program. A strange interaction to note is that we are not getting true single digit millisecond accuracy in the output. I wasn't sure why that is, since it was still happening when I swapped to using c++ std::chrono. Currently, we are getting msec data that are accurate to the 10s of msec. The final digit seems to only ever be 0. What is interesting is that this issue seems to also affect the example output given in the rubric, so I assume it's not an issue that only I am facing. Hopefully this doesn't really affect the grading of the assignment as it's not the main point of the assignment.
13. Things that are not mentioned here should be following the given guideline.
14. Finally, we have a little indicator at the beginning printing out the threads that are starting.

Project Status

The project should be working completely as specified by the assignment rubric with perhaps some extra features. There are no current known issues.
As for difficulties that I encountered during this assignment, I have had to face many. Unlike other students who most likely finished this assignment in 8-10 hours, I believe I have spent upwards of 40 hours on this assignment. Some of the strange things I had done in this assignment was using pthread_mutex instead of posix semaphores, making me design for binary semaphore instead of a counting semaphore, which was aledgedly easier. I also outputted some prints through the task threads instead of the main thread. Here are some of the main difficulties I encountered.

1. First issue I encountered was manual dynamic allocation of memory. I used to have much more dynamically allocated arrays, however, failure to destroy pthread barrier objects in a different text segment caused garbage collection in the text parsing function to cause a segmentation fault. To solve this issue, I decided to use static sized character arrays instead of dynamically allocated character arrays in functions. As for global variables that are being used for critical sections, I used std::map/std::unordered_map to store resource information and the monitor information as well.
2. The second issue I had encounterd was that I was trying to use circular wait prevention by ordering the resources and forcing all tasks to grab these resources in descending order. This would give better concurrency and evenly distribute the progress of the tasks. The first issue I ran into was that I was grabbing one resource unit at a time instead of taking a chunk. This method works well with test cases where there are only one unit of resources for a resource type, however, once there are much more resource units, and multiple threads want to grab a large amount of this resource, those threads will be deadlocked.
3. After I swapped the implementation so that it would take in the full chunk and wait if there are not enough resources, we were still having a mysterious deadlock that was happening rarely and randomly. After some hours trying to find out what was causing this issue, I gave in and swapped to

the recommended deadlock prevention method of waiting for all resources to be available before grabbing them and running the program. The flaw I saw with this method is that program progression will be a little more random, however, it is much more reliable in test cases than my orignal method.

Testing and Results

I created three custom test cases for this assignment and used the given test case as well. The first custom testcase was simply to run a quick circular wait scenario to check if the program worked, and was able to avoid simple deadlocks. The second test case is a stress test case where we have maximum resource types and task threads. The resource units for each resource types also vary in this test, and the tasks will contest each other when grabbing these resources. This test was done with 1000 iterations to try and cause a deadlock to occur. The third test case was created to test weird timing cases where a task doesn't have to do an idlewait while others don't need to do a busy wait, or both. It's really just a check for robustness and if a deadlock would occur because of a mistake. Finally the given test case is there to more sure that the outputs are somewhat matching.
The result from lastest testing was that I was unable to cause a deadlock using the code. The format matches with the one given by the assignment rubric. The concurrency of the program is there as threads are able to be in the RUN state at the same time, however the distribution of progress is not as good as when the program was using circular wait prevention as opposed to the current resouce checking method. Since resource checking is somewhat more random than circular wait prevention, the waiting time output for the given example test case will vary by 100 msec from around 50 msec wait to 250 msec wait. We were able to run the 1000 iteration test case 2 around 20 times as I was writing this README, so I believe that there are no deadlocks with the cases that I created.

Acknowledgments

1. GeeksForGeeks
2. I think one of the function in the code is from APUE
3. The TAs