Assignment 4 – Project Report
CMPUT 379 – Fall Term 2018
Student Name: Jacob Bakker

Objectives

The objectives of this assignment are to become familiar with the use of multi-threading and the synchronization of threads by recognizing instances of critical sections, possible deadlocks, and whether a synchronization problem can be solved by relating it to examples like Lamport's bakery algorithm or the reader-writers problem.

How to Run

Extract the contents of the "submit.tar" file into an empty folder, then use "make -f a4Makefile" to compile the program.

Design Overview

Session Class

- Parses the input file, saving the system resource info from the "resources" line to a SessResDict and – for each Task line – creates a Task instance and saves it to the Task Manager.
- After initializing all Tasks and resources, the Task Monitor is ran on its own thread just before the Task Manager is made to run each Task on its own thread. Immediately after, the Task Manager runs all Tasks.

Task Class

- When created, Tasks are given a pointer to the Session's ResDict through which they attempt to acquire their needed resources before running.
- To prevent deadlocks, Tasks are prohibited from acquiring only some of their need resources. If any one of the Task's needed resources are not available, the Task will wait 10 milliseconds before querying the SessResDict again.
- To ensure that the Task Monitor cannot print while Tasks are changing status without starving the monitor, a mutex system is implemented that allows the Monitor to lock out any Tasks not currently changing status from doing so until the Monitor has printed.
    - Tasks currently changing status can finish, then the Monitor polls and prints all Task statuses.
    - This problem is treated as an instance of the readers-writers problem with writers preference implemented, where readers are the Tasks and the writer is the Monitor. Any number of Tasks can change status concurrently, similar to how any number of readers may concurrently read without conflict; the Monitor – when polling and printing – requires that all Tasks have their status locked just like how writers need for the file being changed to have no readers as they write to it.

SessResDict and TaskResDict

- The Session Resource Dictionary enforces mutual exclusion between Tasks using a single mutex; only 1 Task can query the SessResDict for needed resources at a time.
- Queries to SessResDict for the acquisition of resources will return true/false to indicate the operation's success. When true is returned, the SessResDict will have already transferred the required resources to the TaskResDict that requested them.

Task Manager Class

- The Task Manager is responsible for running all Task threads and polling the Tasks for their current status.
    - Tasks threads are executed sequentially, with each call to "pthread_create" preceded by a lock on the mutex for running threads. This is to prevent the thread from running until the "pthread_create" call has resolved to try and ensure that the Task argument reaches the thread.
    - Task statuses can be polled and saved to a dictionary format. The poll method does not prevent Tasks from changing status, as the class calling the poll method is expected to handle the exclusion.

Project Status

The project appears to function without issue. The input file can be parsed even with an arbitrary amount of whitespace, thread creation and execution work as expected, and the final results of the simulation mimic those in the assignment spec. Tasks appear to follow mutual exclusion and the Task Monitor is capable of printing Task status without those Tasks changing status partway. Issues largely resulted from memory management; for example, threads would not be deallocated if they terminated before the call to "pthread_join". though this was solved by specifying a DETACHABLE flag during the thread's creation.

Testing and Results

- Added arbitrary amounts of whitespace to random parts of the input file such as the beginning and end of lines and in-between tokens. Provided the white space was composed only of space characters, the parsing of each line worked as intended.
- Attempted running the program for 1, 5, 10, and 30 iterations to test for errors that may result after some number of iterations (e.g. memory leaks, corruption). No memory issues were apparent during testing.
- Tested exclusion and concurrency by having one Task require all resources while having each of the other Tasks required only a subset (e.g. SysResources={A:2, B:2}, Task1={A:1}, Task2={B:1}, Task3={A:2, B:2}).
    - The Monitor thread showed either Task3 exclusively running or Task1 and Task2 running simultaneously, but Task3 was never shown as running alongside another Task. In addition, the Monitor reported no duplicate tasks between status categories.

Assumptions

- The input file is assumed to have whitespace composed only of space characters. This is based on the following eClass forum post: https://eclass.srv.ualberta.ca/mod/forum/discuss.php?d=1087835#p2877039

Acknowledgements

- In the Task class, the method “change_status” implements a mutex system taken from the pseudocode solution to the “Second readers-writers problem” found at the following link: https://en.wikipedia.org/wiki/Readers%E2%80%93writers_problem
    - The method “run” in the Monitor class implements the writer side of this solution, while the Task method implements the reader side.
- The functions in the “MutexLib” files are taken from the file “raceC.c” provided in the CMPUT 379 document “Experiments involving race conditions in multithreaded programs”.
- The structure of the “a4Makefile” is taken from https://stackoverflow.com/a/30602701.