Objectives

The purpose of this assignment was to give us practice handling threads. With thread creation, the main challenge is thread synchronization and handling race conditions so that unexpected behaviour is kept to a minimum. One method to avoid deadlocks and race conditions is the use of mutexes. In my assignment, the use of mutexes proved invaluable for my solution to the problem. For me personally, the most valuable part of this assignment was being able to develop a practical understanding of what thread synchronization is, why it is necessary, and how it can be achieved.

Design Overview

The basic design of my program follows the specification of the assignment. When the user runs the program, they must also enter the input file to be read, followed by the monitor time and number of iterations per task.

Some key highlights of the program:

- When the program is run, a new thread is immediately created for the monitor which prints out the states of every task every *N* seconds where *N* is the specified monitor time when running the program.
- When the input file is parsed, it puts all resources (and their counts) in a global map called 'resourceMap' where each resource is put in a key/value pair of form <resourceKey, resourceCount>. All the threads utilize this map in order to procure and release required resources. After mapping the resources from the file, every task and their details is saved in a global vector array called 'taskList'. Tasks will be assigned using this taskList.
- There are three mutexes used within the program. The first is the threadMutex which is locked when the main thread creates a new thread that will be assigned a task. After a task is assigned the mutex is unlocked. The second mutex is the iteratorMutex, which is the mutex that threads lock when procuring resources (or returning resources) from the resource map and unlocked after procuring or returning resources to the map. The third mutex is the monitorMutex which is locked by the monitor before printing the task states, and unlocked after printing. It is also locked by the threads before they attempt to change their state and unlocked after successfully changing state. This monitorMutex ensures that states cannot be changed while the monitor is printing.
- When a thread acquires the iteratorMutex, it first checks to see if all the required resources are in the resource map, if there are not enough resources, it releases mutex and waits until it gets the mutex again to once again check the resource map. If there are enough resources, the thread 'procures' the resources by decrementing the resources from the map. After acquiring the resources, the thread releases the mutex.
- When a thread is done executing (delaying by specified busy time) it acquires the mutex once again and returns the resources to the map.
- Threads will switch states depending on if they are waiting for required resources, running, or in an idle state (immediately after running or if no more iterations to run)
- The program keeps track of each threads total wait time, run time and idle time

- Upon termination of the program, the system and task information is displayed in the exact same manner as the examples from the assignment specification
- I reused some blocks of code from the lab exercise. More specifically, I borrowed the code for the delay and mutex functions from the *RaceC.c* file on eClass.

Project Status

The implementation of this assignment is complete and adheres to all the requirements from the assignment specifications. As far as I can tell, the program executes exactly how I want it to with the expected results. It is difficult to guage the correctiveness of the program, however, due to the nature of threads and race conditions. Every instance of the program will vary in terms of execution time and order of task executed to a minor degree.

One fault currently in the system is if the user enters non integers when specifying monitor time and iteration. If you want to break the system, that is an easy way to do it.

Testing

Testing was a little difficult with this assignment. There is no 'expected' output. Of course you can look at the output and determine if it makes sense, but since tasks were executed somewhat randomly it was more difficult to verify the correctiveness of the system. I used my own test file in my testing. I started out small (3-4 resources and 3-4 tasks using these resources) and started to expand (more resources and more tasks) when I felt confident in the program. I also tinkered with varying busy times and idle times for tasks to see how my program would behave, and I am happy with how it handled those changes.

Assumptions

I am assuming that the input file to be tested will be provided. As such I did not include it in my submission.

Acknowledgments

Undefined Reference to ‘pthread_create’
URL: https://stackoverflow.com/questions/17264984/undefined-reference-to-pthread-create
Answered By: Mats Peterson
Date: June 23, 2013

General Pthreads Information
URL: https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html
Author: Greg Ippolito, 2004

CMPUT 379: Experiments involving race conditions in multithreaded programs
URL: http://webdocs.cs.ualberta.ca/~c379/F18/379only/lab-pthreads.html
Code used from *RaceC.c*
Author: E. Elmallah (with contributions from Tobias Renwick)