Christian Loera

CSC 462 – C++ Multithreading

Jetson's Writeup

My application utilizes a lot of tools described in C++ Concurrency In Action such as mutexes, unique locks, and condition variables. I also used some of the techniques to avoid deadlocking like not nesting locks, locking in a certain order, and not calling user-defined functions when there is a lock around it. This forced me to think about the architecture of my application in terms of function interaction rather than object interaction. Thinking in terms of functions made it easier for me to isolate critical regions and create functions with specific tasks in mind. This has also allowed me to figure out the type of function I would need to keep all 23 threads spinning on thread creation.

There are four classes that are used to spawn all the threads, FileThread, CoordinatorThread, PlaybackThread, and WaveOutThread. In main() FileThread, CoordinatorThread, and PlaybackThread are instantiated as pointers and spawn fileThread, coordThread, and playbackThread of type std::thread. WaveOutThread, on the other hand, is instantiated as one of PlaybackThread's data members and spawns waveThreads within PlaybackThread's PlaybackMain() function. Each of these classes have a "main" function which is used to continuously loop on thread creation and will only break on certain conditions.

FileThread's FileMain() function has a simple condition, it will continue to loop until all 23 .wav files have been loaded into its buffer. This is done within a for loop and since the number of .wav files is predetermined we can hard-code the number used in the loop's conditional. Once a file has been loaded into pOriginalFileData FileMain() will continuously

sleep and check the buffStatus enum in a while loop until it has been set to ContainterStatus::EMPTY, at which point it will break out of the while loop. This is the most important "main" function out of all the classes since all other "main" functions are dependent on FileMain()'s longevity. As long as FileMain() has files to load, all other threads will continue to stay alive and process data.

In the CoordinatorThread class, the CoordMain() function will continue to loop until it cannot pull anymore data from FileThread's buffers and all of the data from frontBackBuffer[] has been depleted. Since CoordMain()'s condition is dependent on the current status of FileThread it takes a fileThread reference and uses it to call TransferBuffer() within PullData(). TransferBuffer() will return a ThreadStatus enum and memcpy() data from FileThread's buffer to CoordinatorThread's buffer. If fileThread is still active then it will send ThreadStatus::ALIVE. Otherwise, it will return ThreadStatus::DONE if the FileMain() loop has ended. Once the CoordinatorThread knows the status of FileThread, CoordinatorThread will wait for the front buffer to deplete before determining if it needs to swap the front and back buffers or exit the while loop. If it exits the while loop it will set coordStatus to ThreadStatus::DONE so that the PlaybackThread will know when to stop requesting data.

The PlaybackMain() function in PlaybackThread is a bit more complicated than other "main" functions in regards to the amount of objects that it has to manage. PlaybackMain() creates 20 WaveOutThread objects as well as the corresponding waveThreads associated with those objects. This is so that PlaybackMain() and PlaybackLoop() can orchestrate the communication between pCoordThread and waveOutThreads. PlaybackThread needs to call waveOutThreads' prepareHeader(), play(), and unprepareHeader() functions. It also needs to delegate pCoordThread to push data into already played waveOutThreads. If a waveOutThread

has already been played then waveMessenger will decrement its numWaves counter within the waveOutProc() callback and cause the inner most while loop in PlaybackLoop() to break out. Now the PlaybackThread can continue requesting data pushes and triggering playback until the pCoordThread signals that it is done, at which point PlaybackThread will begin clean up on waveOutThreads and waveThreads.

WaveOutThread's WaveOutMain() is a very passive function in that it relies heavily on conditional variables to signal when it can write the WAVEHDR pHeader. WaveOutMain() loops over writeHeader() which is a wrapper for waveOutWriteHeader(). Within writeHeader is a unique_lock that locks the waveOutWriteHeader and canPlay flag and waits for play() to notify the cv.wait() to unlock and loop through again. The wait() function has a predicate that checks to see if the canPlay flag is true. I chose to use predicates in all of my wait() functions to prevent any spurious wakeups from unlocking the mutex, which would be very dangerous in writeHeader() because it could throw off the synchronization between CoordinatorThread, PlaybackThread, and WaveOutThread. WaveOutThread also has an unprepareHeader() function that is called on all WaveOutThreads at the end of PlaybackMain() so that it can not only unprepared the pHeader, but also kill off all the WaveOutThreads that are still waiting for a signal inside writeHeader().

Working on this assignment was extremely headache inducing in that it was difficult trying to wrap my head around all the possible thread interactions. Since variables can change line by line it was difficult for me to figure out the cause of certain bugs. There were times where I didn't even know how to begin debugging my code once I added threads, mutexes, and condition variables. This is especially true when I was synchronizing the FileThread and CoordinatorThread to transfer data.