JETSONS PROJECT

CSC 462-OPTIMIZED MULTITHREADING

Name: Jacob Zackaria Date: 11/06/2021

Student ID: 1988030

DESCRIPTION:

The assignment is about using multithreaded architecture in an efficient way to play the "Jetson's theme song" that we all hear and love. As an input, we receive the theme song split into 23 individual raw-header less wave(.wav) files. One of the requirements of this assignment is that minimum of 24 total threads should be used excluding the main (Kill thread, File thread, Coordinator thread, Playback thread and 20 Wave threads) to play the wave files. Other requirements include 200ms sleep only in File thread and the threads can communicate each other using condition variables, mutexes or future-promise pairs.

THREAD CREATION:

Each thread in this project is created as class objects, which are launched from main thread using a standard thread call in a launch function. The main thread is responsible for allocating the FileThread, CoordinatorThread, PlaybackThread and the KillThread class objects and launching them. The PlaybackThread constructs 20 WaveBufferThread's and launches them during its launch call.

- FileThread => This thread is responsible for loading the individual wave files to the file buffer (Buffer class). File buffer is a 512KB (512 * 1024 bytes) memory used to store the wave data. Each individual wave files are less than or equal to 512KB as of assignment rule. At the start stage, it preloads two wave files and for the subsequent wave files, it sleeps for 200ms after each read. After each file load, the thread waits for the CoordinatorThread to move the file buffer data to its own buffer. When the file buffer is reading or writing, its mutex is locked to avoid concurrent access. After reading all the 23 wave files, it notifies the CoordinatorThread and then exits.

- CoordinatorThread => This thread is responsible for moving the loaded file buffer to its own buffers and transferring required data to the wave buffers. The thread contains two buffers, each of 512KB memory named as front and back buffers.

Initially, the thread waits for both the buffers to get filled from file buffer (i.e., preload of two wave files of FileThread). During its thread loop, it has 2 states of operation.

If the thread is in “FILL” state, it moves the data from file buffer to its back buffer then notifies the FileThread and then switch to “DRAIN” state. While in “DRAIN” state, the thread reads commands from PlaybackThread queue(uses CommandPool) and executes them. The command to this thread contains pointer locations to which WaveBufferThread it should transfer its data from its front buffer. While in this state, the CoordinatorThread communicates with PlaybackThread continuously to execute the commands in order they were sent. If the front buffer is emptied while executing the commands, the thread swaps the buffer pointers and then switches the state to “FILL” to fill the emptied back buffer.

Before switching to the “FILL” state, the thread also checks if the FileThread is still running. If not, it will only run in “DRAIN” state and thus waits until both the buffers are emptied before notifying the PlaybackThread and exiting.

- PlaybackThread => This thread manages the wave threads and its order of playback. During construction of its object, it opens the audio device for playback (waveOutOpen) and creates 20 WaveBufferThread objects. Similar destruction sequence is done on its destructor where the 20 WaveBufferThread objects are destroyed and closes the audio device (waveOutClose).

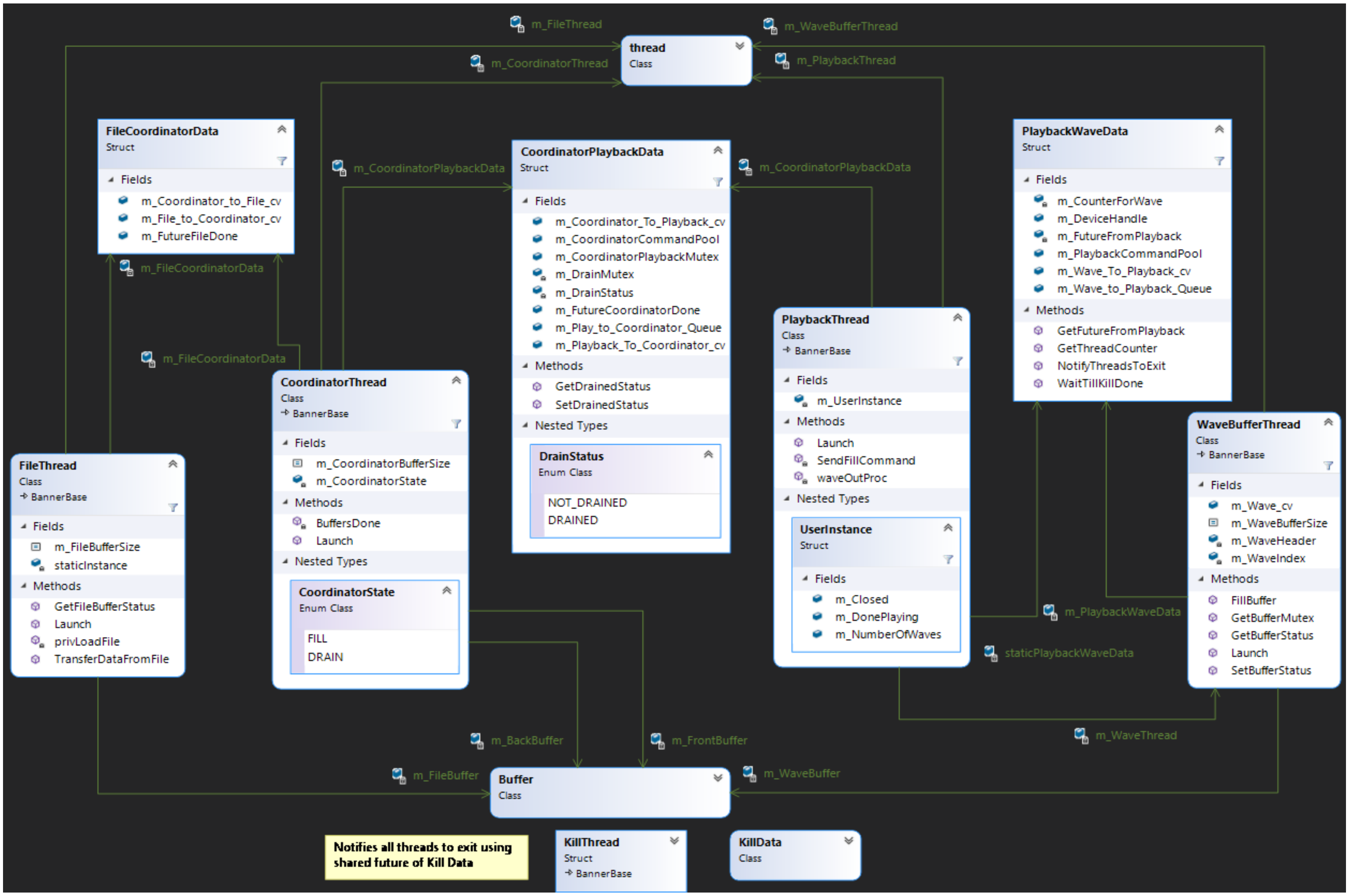
  After its launch, it immediately launches 20 WaveBufferThread and sends 20 commands to fill its buffer. The thread then waits till the 20 commands are executed in order by the coordinator before entering the thread loop. In its thread loop, it continuously pops commands of the input queue(uses CommandPool) from wave callback function(waveOutProc). wave callback function sends command to the PlaybackThread whenever a wave is done playing. The command contains the wave index of the wave thread which just finished playing and when PlaybackThread receives this command, it attaches a command to the CoordinatorThread to fill the buffer of wave thread which just finished playback.

  The thread after sending a command to the CoordinatorThread waits for its response, so that it can notify the filled wave thread to write data for playback. It again goes to a wait state to receive response from the filled wave thread implying the write was successful. If the coordinator has finished its buffers, the playback stops receiving anymore commands and wait for the waves playing to die out to start the killing sequence.

- WaveBufferThread => In its loop, waits to receive notification from the PlaybackThread to write(waveOutWrite) the wave buffer data to the audio device. The wave buffer is a 2KB (2 * 1024 bytes) memory which is filled by the CoordinatorThread. During its construction, it prepares the wave header(waveOutPrepareHeader) and assigns the wave index as user instance to be used in wave callback. The destruction phase takes care of unpreparing the wave headers(waveOutUnprepareHeader).

- KillThread => It is launched from the main thread using std::async() to use the future.get() functionality(stops main thread until this thread exits) at the end of main thread. KillThread receives notification from PlaybackThread to start its kill operation. The kill operation involves making all the shared future invalid and then waiting for all the threads to exit. Every sub-thread except this thread contains a RAII(Resource Acquisition Is Initialization) counter, which helps the KillThread to identify if there are threads waiting to exit.

COMMUNICATION: (class diagram)



Given is a class diagram showing the different threads and its communication mechanisms. Every thread except KillThread(uses std::async call) consist of a launch function where the thread begins its operation. There is one file buffer for FileThread, two coordinator buffers for CoordinatorThread and one wave buffer for each WaveBufferThread. The buffer class contains its own buffer mutex as well as a mutex for its buffer status(EMPTY or FULL).

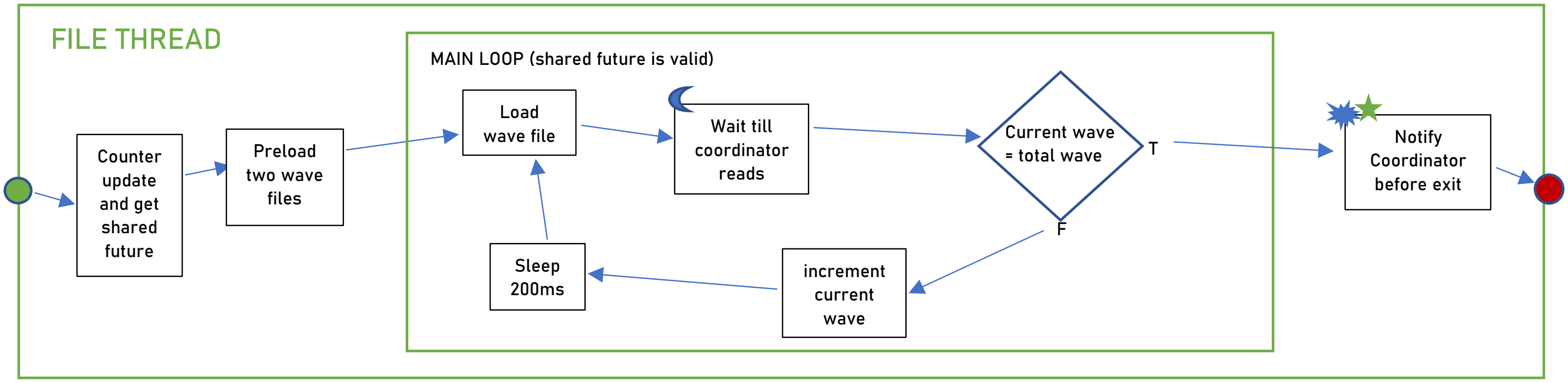
The FileThread and CoordinatorThread shares a "FileCoordinatorData" structure which contains the variables shared by both. The FileThread uses "m_File_to_Coordinator_cv" condition variable to send notification to waiting CoordinatorThread about its file load during the preload stage. The CoordinatorThread uses "m_Coordinator_to_File_cv" condition variable to send notification back when the data is moved from file buffer to its back buffer in its "FILL" state. The structure also contains a future which becomes invalid when FileThread ends.

The CoordinatorThread and PlaybackThread communicates using "CoordinatorPlaybackData" structure shared by both. When the coordinator is in "DRAIN" state, it waits for the playback to send a command through the queue from the command pool and is woken up by receiving a notification through the "m_Playback_to_Coordinator_cv" condition variable. After executing the command, the coordinator notifies waiting PlaybackThread using "m_Coordinator_to_Playback_cv" condition variable. While performing the drain operation, both the threads use a "DrainStatus" structure with mutex protection inside the shared data to synchronize the operations. The shared data also contains a future which becomes invalid when CoordinatorThread ends.

All the WaveBufferThread shares "PlaybackWaveData" structure created by the PlaybackThread. When the CoordinatorThread notifies the PlaybackThread about drain operation, the playback then notifies the WaveBufferThread using "m_Wave_cv" condition variable in the appropriate thread. After writing data, wave thread notifies back to PlaybackThread using "m_Wave_to_Playback_cv" condition variable to start another drain operation. After playing each wave, the wave callback function creates a command to be passed to the playback thread from the command pool of shared data. The shared data also contains a future which becomes invalid when PlaybackThread ends. It also has a thread count instance shared by the wave threads to exit cleanly when playback ends.

The KillThread along with its "KillData" shared across all the threads will perform the kill operation by invalidating the future shared by the threads and waiting until the thread count reaches zero for it to cleanly exit. The thread count instance is stored in the "KillData" structure.

DATA MOVEMENT: (Please refer keys at end of this section)



FILE THREAD

MAIN LOOP (shared future is valid)

Counter update and get shared future

Preload two wave files

Load wave file

Wait till coordinator reads

Current wave = total wave

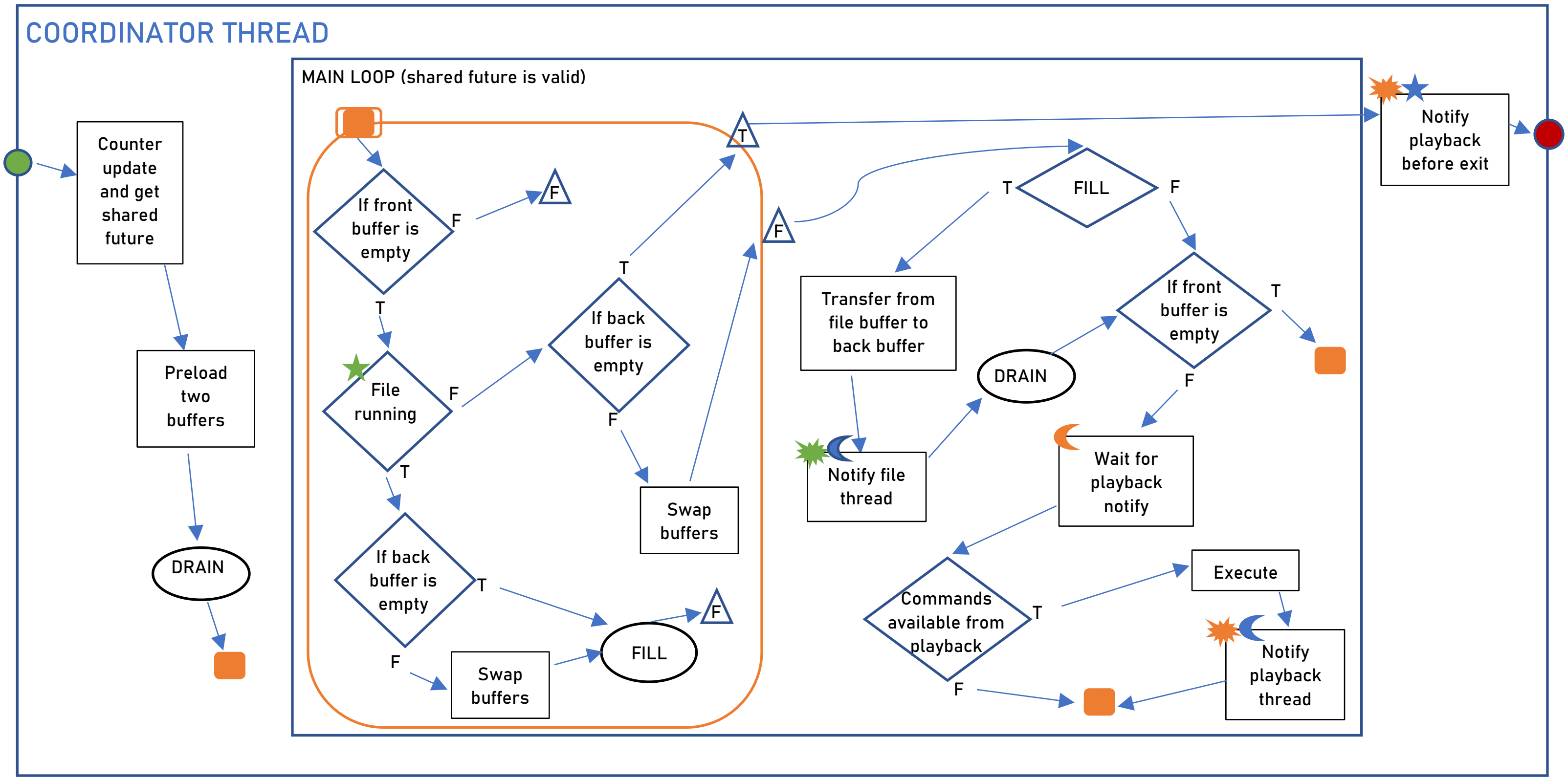
T

F

increment current wave

Sleep 200ms

Notify Coordinator before exit



COORDINATOR THREAD

MAIN LOOP (shared future is valid)

Counter update and get shared future

Preload two buffers

DRAIN

If front buffer is empty

F

T

File running

F

T

If back buffer is empty

T

F

Swap buffers

FILL

If back buffer is empty

T

F

Swap buffers

FILL

T

F

Transfer from file buffer to back buffer

Notify file thread

DRAIN

If front buffer is empty

T

F

Wait for playback notify

Commands available from playback

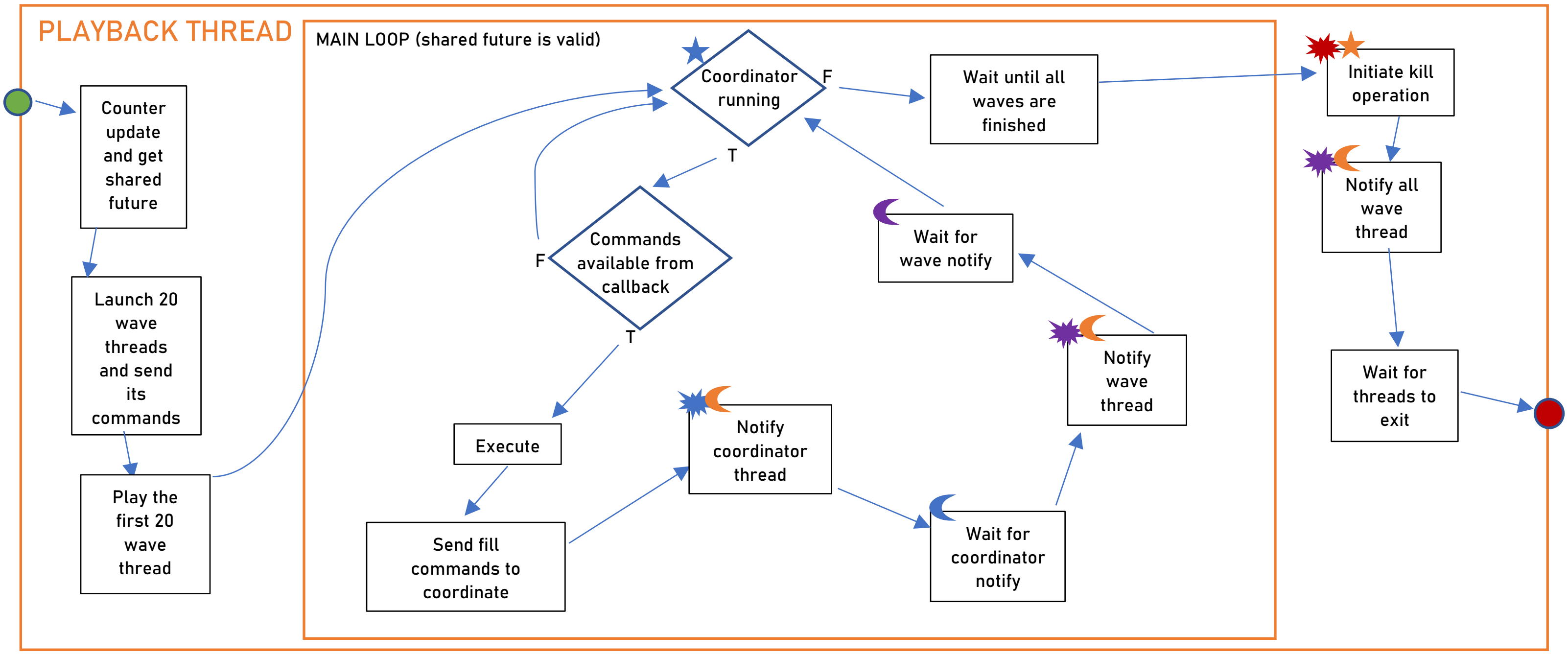
T

F

Execute

Notify playback thread

Notify playback before exit



PLAYBACK THREAD

MAIN LOOP (shared future is valid)

Counter update and get shared future

Launch 20 wave threads and send its commands

Play the first 20 wave thread

Coordinator running

F

T

Commands available from callback

F

T

Execute

Send fill commands to coordinate

Notify coordinator thread

Wait for coordinator notify

Notify wave thread

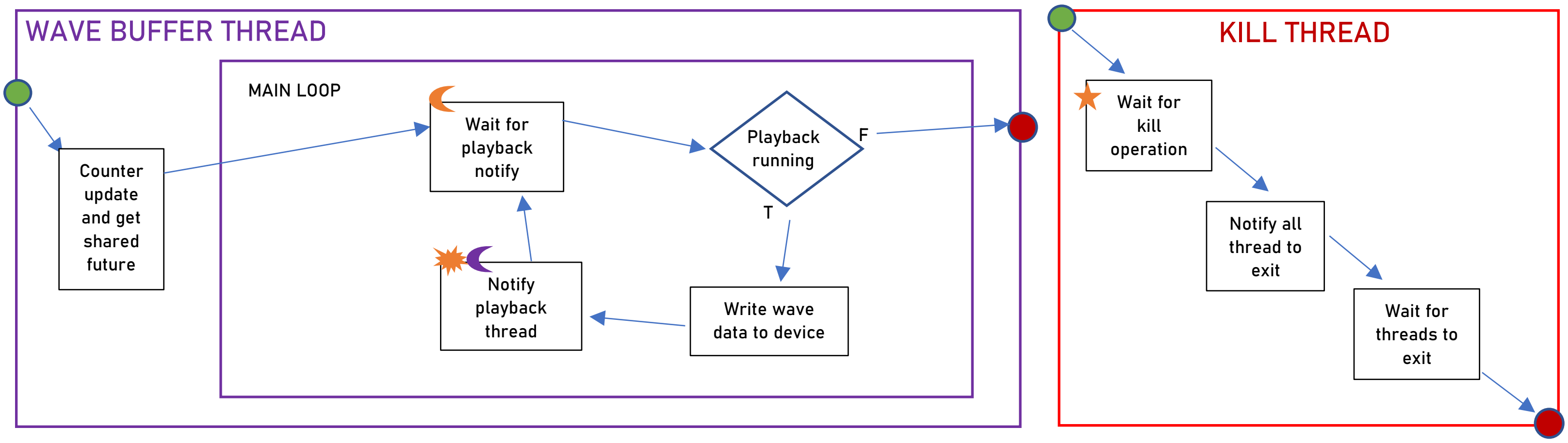
Wait for wave notify

Wait until all waves are finished

Initiate kill operation

Notify all wave thread

Wait for threads to exit



Data flow diagram keys:

Color schema

🟩 => FileThread
🟦 => CoordinatorThread
🟧 => PlavbackThread
🟪 => WaveBufferThread
🟥 => KillThread

● => Thread start
● => Thread end
★ => Future
☾ => Condition-wait
✸☾ => Notify condition wait

✸★ => Notify future
▬ => Function call
[▬] => Function
△ => Function return value

↗ => Data flow
execute => Operation
◇ => If condition
T : True F: False

CHALLENGES:

The assignment was divided into two parts. The first part is where you make the file loading and transfer to coordinator works. We simulated the playback part by sending commands at specific intervals. The challenge's during this phase are, creating a good basic design and to make the coordinator run freely without any delay for file load. A good basic design involved understanding how different threads will interact each other, which all data structures need to be shared, means of communication etc. I took a more object-oriented approach by creating multiple classes for shared data, thread counters, circular queue etc. The next problem is to solve the data race issue, or deadlock situations occurring between FileThread and CoordinatorThread. By adding mutex's to buffers and buffer status helped me avoid data races. To avoid deadlocks I implemented condition variables to wait / notify accordingly when a file is loaded.

The second part of the assignment is integrating the first part to create the whole project.(i.e., Implement the wave playback side). The challenges during this phase included making the coordinator work for both the FileThread and PlaybackThread at same time and to avoid out of order wave execution. To make the CoordinatorThread work for both the file and playback thread, I made a coordinator state. If the state was in "FILL" mode, it loads data from file if available. If the state was in "DRAIN" mode it executes playback commands and fills the waves. The next challenge after filling the wave threads was to play the waves in order they were requested to fill from callback. For this synchronization, I implemented condition variables between the wave threads and playback thread to notify after filling the buffer and after writing the data to device.