MECHENG 313 A2 REPORT

---

jdu853
dpar783

1.0 GETTING STARTED

The program `task2.3.exe` reads audio input from the user's device, optionally performs real-time pitch shifting, and outputs the result. The user can toggle between passthrough mode and pitch shift mode. In pitch shift mode, the user can control the pitch of the output, ranging from one octave lower to one octave higher.

To compile the application in Windows using g++, PortAudio must first be installed. Set the directory to the folder containing task2.3.cpp in the command line and run:

```
g++ task2.3.cpp smbPitchShift.cpp -lportaudio -o task2.3.exe
```

To run the program type:

```
task2.3
```

The application enters in passthrough mode. Press 'Enter' after inputting keys to interact with the program. 'q' exits the program. 's' enables pitch shift mode. 'p' enables passthrough mode. 'u' and 'd' raise or lower the output pitch when in pitch shift mode.

2.0 MULTITHREADING

The program utilises multithreading to allow for real-time audio processing. Once PortAudio is initialised and set up, four threads are spawned: a reader, processor, writer, and user input handler. These threads execute concurrently and act on a shared ring buffer.

The reader thread runs `ReadStream()` to read audio blocks from the PortAudio stream into the ring buffer using `Pa_ReadStream()`. The processor thread which runs `ProcessBlock()` waits for the status update and a notification from the condition variable. It then calls `smbPitchShift()` on the block, and proceeds to update the status allowing the writer thread which runs `WriteBlock()`, to output the block to the stream via `Pa_WriteStream()`. This is illustrated in the loop in Figure 1.

An independent user input thread runs the `GetUserInput()` function which manipulates the `SHIFT_AMOUNT` global variable, changing the pitch shift factor. To avoid concurrency issues, access to `SHIFT_AMOUNT` is protected with a mutex.

When the user quits the program, the shared atomic boolean `kill_thread` is toggled and all condition variables are told to signal all threads to end their loops and return to the main thread, as shown in the figure. Any threads waiting for access to audio blocks are notified to prevent potential deadlocks. The main thread then closes the stream and terminates PortAudio.

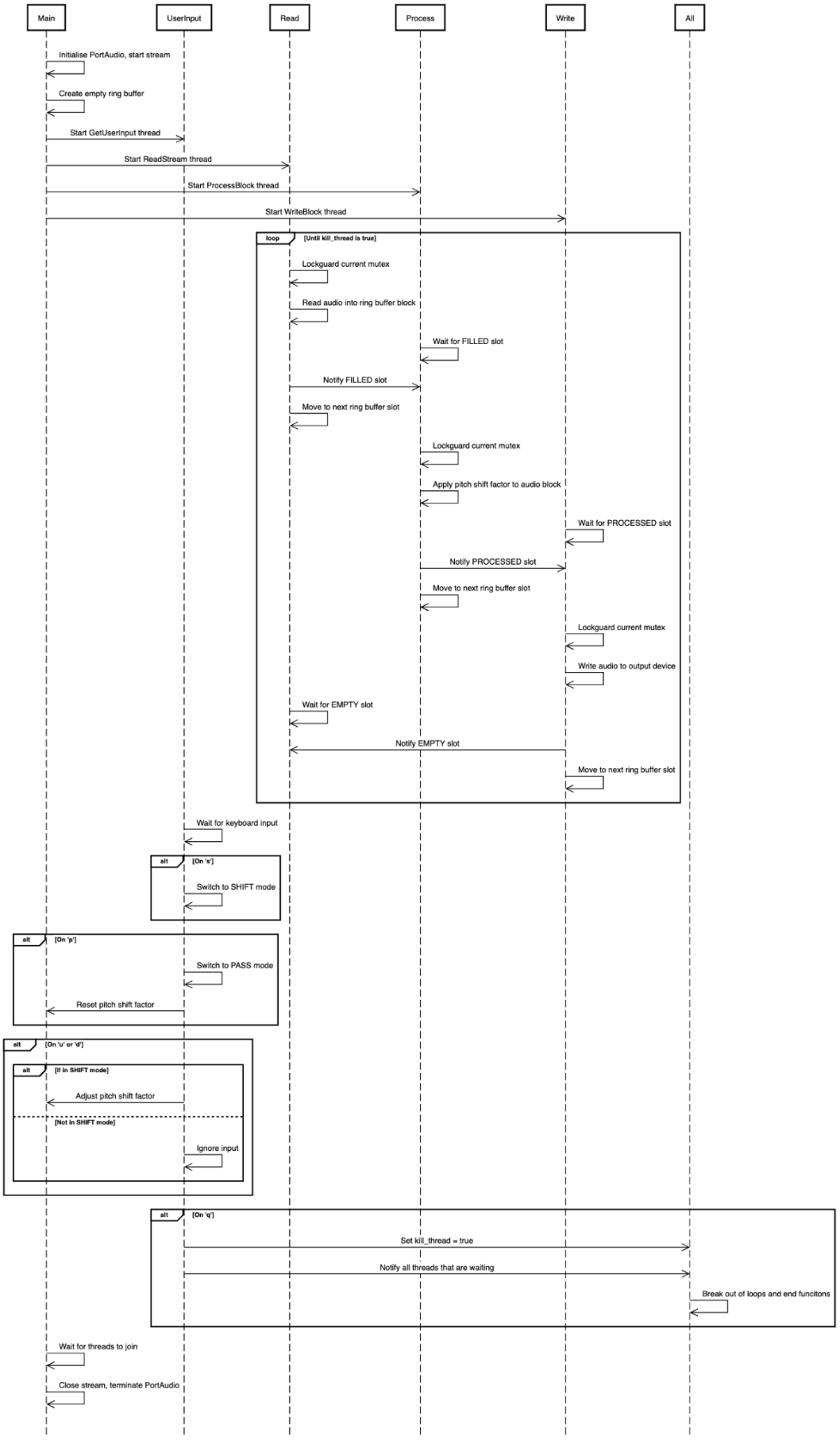

Figure 1: Sequence Diagram for threads in `task2.3`

3.0 DESIGN DECISIONS

Each slot in the ring buffer is a `struct` that holds a block of audio samples, a mutex, a condition variable, and a status enum (`EMPTY`, `FILLED`, or `PROCESSED`). Using mutexes and condition variables allows threads to communicate and synchronise, without busy waiting and avoiding race conditions. The buffer size was set to 128 to balance latency and reliability. The ring buffer contains 16 slots to allow for some slack between threads if one runs slower than others.

As the global variable `SHIFT_AMOUNT` is read in the processing thread and updated in the user input thread, a mutex is used to protect the reading/writing of this variable. However, instead of holding the lock throughout processing, we decided to quickly copy `SHIFT_AMOUNT` into a local variable and release the mutex immediately to ensure minimal blocking of the user input thread.

The user input handling is structured as a simple finite state machine with two states: passthrough and pitch shift. In passthrough mode, `SHIFT_AMOUNT` is reset to 1.0 as a parameter for `smbPitchShift()`, and user input for changing pitch has no effect. In shift mode, the ‘u’ and ‘d’ keyboard inputs are valid. Pressing ‘s’ or ‘p’ toggles modes while ‘q' exits from any state.

To allow for real-time shutdown, the atomic flag `kill_thread` was used for thread-safe signalling. The program also wakes all threads using `notify_all()` to ensure no threads remain indefinitely blocked on condition variables during exit (deadlocks).

On starting the program, it outputs to the user all valid inputs to make use of the program. State changes and pitch changes are also displayed, as well as when pitch limits have been reached so the user can keep track. The program is intended to be easy to understand and use for all audiences.

4.0 REFLECTIONS

The assignment was really fun and helped us learn a lot about multithreading and real-time software. The most enjoyable aspect was building a robust application that was easy and fun to use. It was rewarding to see the theoretical concepts used for something tangible.

Although a good skill to have, we did struggle with environment setup as some of the instructions made it more confusing, and some external documentation was outdated. We also feel that some concepts delivered in lectures were slightly unclear, especially relating to synchronisation techniques. However, this section of the course as a whole was engaging and we can see how it will be useful in a real world setting.