Ayser Jamshidi
Design

Design

The program has multiple parts to it in the following hierarchy:

- Main process
    - Child Process 1 (Logger and Signal Handler)
        - Thread 1 (Logger thread)
        - Thread 2 (SIGUSR1 Handler)
        - Thread 3 (SIGUSR1 Handler)
        - Thread 4 (SIGUSR2 Handler)
        - Thread 5 (SIGUSR2 Handler)
    - Child Process 2 (Signal Generator)
    - Child Process 3 (Signal Generator)

Alongside this, the project is structured in an intuitive way where the entire main function is in main.cpp and all other functions that are reserved for separate threads/children are in "signalhandler". Inside signalhandler are unique class files per task.

- receiver.cpp : exclusively used for receiver threads
- logger.cpp : houses the spin-locking logging function
- initialize.cpp : contains a multitude of signal blocking functions and setup for creating and managing receiver threads.
- generator.cpp : houses the signal generator functions

Once the program starts, it does several things: first, it removes any previous log files to ensure we have a new, clean output. Second, it checks to see if the user has passed any parameters. If they have, it checks to see if the **first** parameter is "0". If it is, the user has enabled what I called "Time Trial" mode. In "Time Trial" mode the program has exactly 30 seconds of execution which it will then stop all threads and processes peacefully and also output the logs. After ensuring the program is running in its desired mode it then starts creating the first child process – the Logger and Signal Handler.

The **Logger and Signal Handler** reside in the previously mentioned "initialize.cpp". Once called, this function immediately blocks both SIGUSR1 and SIGUSR2 for the child that runs it, along with unblocking SIGTERM (just in case, even if it wasn't previously blocked). Afterwards, it sets up a signal handler for SIGTERM to allow the child to peacefully exit when the time comes or through user intervention (Ctrl^C). Once all of the signal masks are set up, the child will now spawn 4 extra threads, all signal receivers, with the first two handling SIGUSR1 and the last two SIGUSR2.

The **Receiver Threads** are all running under Child 1 at this point, and have several jobs. Upon its creation, it checks to see what signal it was assigned and immediately unblocks it. Afterwards, it announces that the specific thread is ready and waiting for signal handling, which it *then* assigns a signal handler for the signal it was given. At this point, it's now stuck in a spin-lock until the program is ready to terminate, which it will peacefully exit the spin-lock and terminate cleanly.

The **Signal Handler** function – every time it's called – receives a specific signal. Before handling any signals though, the program checks if should or shouldn't end. Doing so prevents unnecessary time waste and excess lost signals. If it doesn't need to end, the receiver figures out what signal it was given. It must do this as there's only one function for both SIGUSR1/2. Having a unified function for both signals saves lines and (to me) makes it more legible. Once the signal is known, it acquires the counter and mutex lock for that respective signal then safely increments its counter (via pointers) by locking, incrementing, then unlocking the respective signal's counter and mutex. After safely incrementing the respective counter, the receiver thread now checks to see if the logger queue is full. Full in this case is if the queue contains 16 (or more) log entries in it. If it does, the receiver thread then calls a signal "MAX_SIGNALS" to indicate to the logger that the logger queue is full and ready to be dealt with. Right after, the receiver thread then waits until the logger sends out its own signal – "EMPTIED_SIGNALS" – indicating to all receiver threads that the logger queue is now emptied and can continue on. At this point, the receiver thread *finally* creates a new log entry that contains the time (since runtime) it took to be dealt with, the receiver thread ID that handed it along with what signal it's handling. It then gets pushed to the logger queue, and repeats that entire signal handling procedure every time its respective signal is sent to it.

The **Logger Thread** is a simple function that ensure the first child process is in a spin-lock until the program is ready to terminate. It safely accesses the logger queue by locking the logger mutex and checking if the amount of entries in the logger queue are less than 16. If it is, it then waits for any receiver threads to signal "MAX_SIGNALS" which will then allow the logger thread to continue. Once it continues, it will open an output stream to the file "signalreceive_log.txt" and spit out an error if the process was unable to create an output stream to said file for any reason. Afterwards, we deal with the entire queue, each entry as follows:

- Get the first entry in the queue and pop it out of the queue
- Output the Thread ID, signal received (in text form e.g. SIGUSR1) and the time it was received.

After that, it adds the amount of received SIGUSR1/2 entries to a separate, respective logger counter. Every 16 logs the logger will also output the current total count it has logged. This serves mainly to let the user know that the program is still working and not just frozen. After that, the function simply signals "EMPTIED_SIGNALS" to the receiver threads that the logger has emptied the logger queue. It unlocks its logger mutex then repeats this entire process until the program is ready to close peacefully.

The **Signal Generator** is very simple. It checks to see if the program is in "Time Trial" mode and, if so, checks to see if it has gone over its allotted time (30 seconds). If so, the program will immediately call all threads and processes to peacefully exit via SIGTERM. If not, the generator continues and decides to signal SIGUSR1 or SIGUSR2 by a random number generator that returns 0 or 1. 0 will tell it to signal SIGUSR1 and 1 will be SIGUSR2. It does this until the program is attempting to terminate, which it will then peacefully exit.