Alex Sawicki - Project 4 - Documentation

User Manual

In order to use the program, you must give the command one argument. This argument consists of either a 0 or a 1. For example, “./main 0” or “./main 1” are valid commands/arguments to use. The 0 argument will allow the program to run for about 30seconds long before stopping execution and the 1 argument will allow the program to run for 100,000 signals before stopping execution. After execution, the console should print out the statistics of the execution, including the following: SIGUSR1 signals generated, SIGUSR2 signals generated, SIGUSR1 signals received, SIGUSR2 signals received, SIGUSR1 signal loss, and SIGUSR2 signal loss. That information will also be output to the “sigRecLog.txt” along with the information of the average time between SIGUSR1 signals and the average time between SIGUSR2 signals per 16 signals received. Furthermore, you can see the signal generation log in the “sigGenLog.txt” file. Where it shows the type of signal generated along with the time that it was generated.

Design Specifications

The program is split into three different files: main.cpp(main), signalGen.cpp(signalGen), and signalHandler.cpp(signalHandler). Main.cpp is where the program starts, and handles the forking of the process into multiple different processes that will be directed to the signalGen.cpp and signalHandler.cpp files. signalGen.cpp handles the generation and sending of signals to the signalHandler.cpp file. The signalHandler.cpp file will receive the signals from signalGen.cpp and handle the signals accordingly.

Main starts out by erasing all previous “sigRecLog.txt” and “sigGenLog.txt” values and starts the clock that will be used to time when signals are generated and received. Then main.cpp will send SIGTERM signals to the commands outside of main.cpp when it’s time to

end execution, furthermore, it will print out the execution details to the console and to the “sigRecLog.txt” file. To send the SIGTERM signals it waits until the “runCondition” variable is changed from true to false and the process that was originally in the signalHandler file exits and starts the SIGTERM signals to the other processes. Afterwards main will print out the signal information to the console and to the “sigRecLog.txt” file. In order to send the SIGTERM signal, main blocks the SIGTERM signal in it’s signal mask so it won’t be affected when the other processes handle the SIGTERM signal.

SignalGen is the collection of three processes that create signals for the signalHandler to service. SignalGen has a random number generator that is seeded with the pid of the process inside of the file, this is because for each instance of signalGen, we are in a different process. Therefore, each signal generator will have a different seed for it’s signal generator. SignalGen then blocks each process from SIGUSR1 and SIGUSR2 signals so that the signalGen processes won’t be affected by the signals when they are producing them. It also unblocks the SIGTERM signal so when I want the process to end it will hit a handleTermination() function that will change a boolean variable “runCondition” to false, which will effectively stop the execution of the programs and exit each process. Inside of a while loop that is checked against the “runCondition” variable, is where the signal generation happens. The signalGen process while loop lifecycle looks like the following: sleep for a random amount of time, send a random signal type to the signalHandler, log the signal generated, and repeat. The sleep at the start of the lifecycle is a random amount of time between 0.01 seconds and 0.1 seconds, this will include some variability between the time that the signals are generated and will provide interesting results. Then it will send either SIGUSR1 or SIGUSR2 to the signalHandler process to be serviced. Finally, it takes the current time of the process and logs the signal type and the time the signal was generated to the “sigGenLog.txt” file. Then once the while loop stops executing because the SIGTERM signal was received by the process, each process will stop executing.

SignalHandler is a single process with an additional 5 created threads. There are 2 threads for handling SIGUSR1 signals, 2 threads for handling SIGUSR2 signals, and a logger thread that will log the signal type, thread ID, time received, and the average difference between the times that SIGUSR1 and SIGUSR2 are received separately. In order to receive certain signals, I created signal masks such that two threads could only receive one type of signal, and another two threads could only receive the other type of signal. I then used the signal() command to set the SIGUSR1 and SIGUSR2 signals to be handled by a signalHandler() function. The signalHandler() function will then increment a counter corresponding to the signal type and create a log to be added to my logger queue(which is a priority queue that will sort the logs by time). This logger queue will add logs to the logger queue until it reaches a size of 16. Once the logger queue reaches a size of 16 it will signal the logger thread that the logger thread can now log the group of 16 logs. Then once the logger thread is done logging, it uses pthread_cond_signal() to signalHandler() that was using pthread_cond_wait() to wait until the logger queue was no logger full. Then it will push the log to the logger queue and unlock the logger queue lock that it locked when it first wanted to see the size of the logger queue. These receiver threads will then return from their separate thread functions and join with the rest of the threads once the “runCondition” variable turns to false.

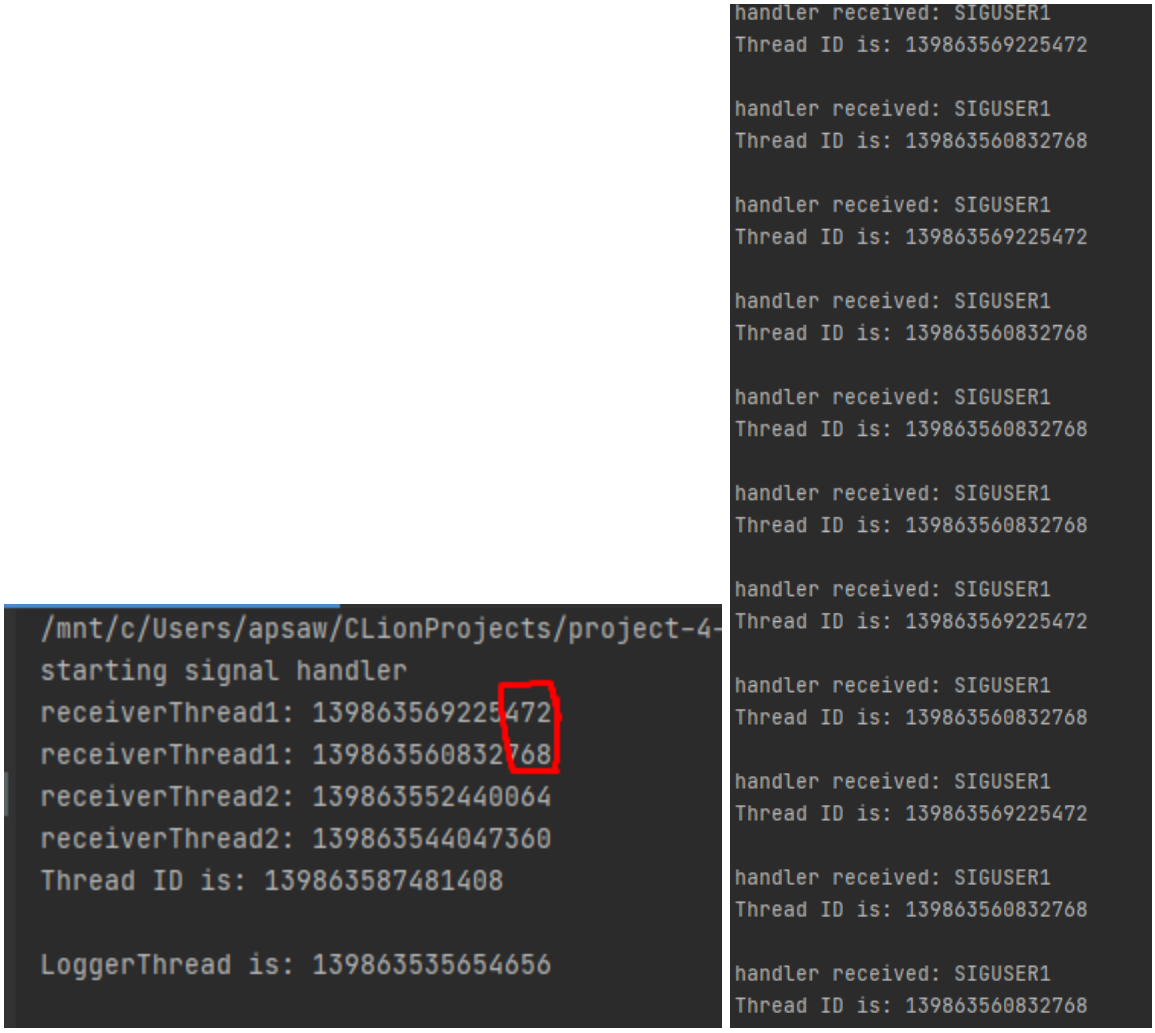
The logger thread uses a lock on the logger queue and waits until the size of the logger queue is full. Once the logger thread is signaled that the logger queue is full by a signal from the receiver threads that add to the logger queue(as specified in the paragraph above), it will take each of the logs in the logger queue and separate them into two vector data structures. With each vector holding the logs of the SIGUSR1 signal and SIGUSR2 signal separately. As it adds these logs to the vectors it writes to the “sigRecLog.txt” the information inside of the 16 logs. Then I get the average of the difference between the times in for each log in the separate vectors, giving me the average of the difference between the times that the signals are received per type of signal that I write to the “sigRecLog.txt” file. The logger thread then does a

pthread_cond_signal() to the receiver threads that the logger queue is now empty and available to have logs put into it. Then the logger queue will stay in this life cycle loop until the “runCondition” variable is false, where it will return from the logger thread function and eventually join together with the other threads.

Finally the main thread of the signalHandler will be continuously spin-locking to check if the “runCondition” variable should be changed from true to false depending on what option the user chose to run the program in when giving the command arguments. The two options are either to wait for 100,000 signals to be run, or to have a 30 second interval of signals being put through the processes. After the “runCondition” variable is switched to false, all the other threads within the process will then join together in the function where the threads were created and return back to the main.cpp file to start the final steps of killing the child processes and logging the last amount of information needed. One thing to note: when getting the count for the signal generators, I counted the number of signals from the log file that I made. I did this because I’m pretty sure that the counters for the different processes in signalGen aren’t shared since fork() doesn’t share data. Furthermore, I never returned the signalGen processes to the main function so I wouldn’t be able to share any counters that they made.

Testing

Testing this program wasn’t too hard since I was able to print out each time a signal was generated, received, and what threads/processes did those actions. To make sure that my threads were working correctly with the signal masks, I was able to only send a mass amount of a single signal to the two threads to show that both threads will be used when put under a heavy enough load of signals. (see below)



As you can see, the receiverThread1's (which take in SIGUSR1 signals) were both able to do some work to service the signals received when given a heavy enough load. This is a problem to test normally as only one thread per type of signal is used heavily over the other thread, meaning that you never see if the other thread is actually doing it's job.

A way to test the logging for the 16 logs is to not only output the average difference between signals but to also look at the signals and calculate the average difference by hand and match it up with the value shown in the log file. (see below for the logs referenced)

```
SIGUSR1 received by thread [140287501559552] at [318353505ns]
SIGUSR2 received by thread [140287484774144] at [318374947ns]
SIGUSR1 received by thread [140287501559552] at [318408870ns]
SIGUSR2 received by thread [140287484774144] at [318430085ns]
SIGUSR1 received by thread [140287501559552] at [318451549ns]
SIGUSR1 received by thread [140287501559552] at [318471194ns]
SIGUSR1 received by thread [140287501559552] at [318492496ns]
SIGUSR2 received by thread [140287484774144] at [318526460ns]
SIGUSR2 received by thread [140287484774144] at [318567234ns]
SIGUSR1 received by thread [140287501559552] at [318588873ns]
SIGUSR2 received by thread [140287484774144] at [318610956ns]
SIGUSR1 received by thread [140287501559552] at [318646122ns]
SIGUSR2 received by thread [140287484774144] at [318667364ns]
SIGUSR2 received by thread [140287484774144] at [318693642ns]
SIGUSR2 received by thread [140287484774144] at [318725705ns]
SIGUSR1 received by thread [140287501559552] at [318754308ns]
SIGUSR1 Avg Time Between Signals: 57257
SIGUSR2 Avg Time Between Signals: 50108
```

When testing the 30 seconds of process runtime I was getting a signal loss rate of 0.61 and 0.13. I can see how these signals are lost when the types of threads get a mass amount of received signals within a short time frame. For example, when testing my logger queue, I made it such that the signal generators would still produce signals, however, the logger queue wouldn't release the 16 logs in order to add more logs to the log files. This showed me how the 16 logs will be put into the logger queue, and each receiver thread will hold onto a log, but each signal generated when the logger queue was full and each receiver already held onto a log was lost since there was no thread to catch it. Thus, the signals are lost whenever the logger queue needs to dump it's logs, and it isn't done in time, making the receiver threads hold a job each, and the signals that are generated will not get received by any thread. (see below for 30 seconds results)

30 seconds

```
SIGUSR1 signals generated: 821
SIGUSR2 signals generated: 780
SIGUSR1 signals received: 816
SIGUSR2 signals received: 779
SIGUSR1 signal loss: 816/821 = 0.609013%
SIGUSR2 signal loss: 779/780 = 0.128205%
```

The same idea applies to the 50k received signals runtime. The percentage of signal loss is comparable to the signal loss seen in the 30 second runtime, and the idea that there is signal loss when the logger queue and all the threads are full of logs holds up with this data.

50k signals

```
SIGUSR1 signals generated: 25093
SIGUSR2 signals generated: 24983
SIGUSR1 signals received: 25062
SIGUSR2 signals received: 24951
SIGUSR1 signal loss: 25062/25093 = 0.123540%
SIGUSR2 signal loss: 24951/24983 = 0.128087%
```