Casey Brooks
CSCE 313 - 504
MP4 Report

In machine problem 3, set set up a simple client-server program where threads were used to communicate to the server concurrently and improve the speed of execution. In this machine problem, we were to use the same client and server to do the same thing, with the exception that the method of concurrency was to be implemented by monitoring the open file descriptors with the select() system call, rather than a new thread for each channel.

While having many threads was a viable option to implement concurrency, it comes at the cost of using a lot of system resources. Each thread has its own stack, which takes about 10 MB of memory per thread. So when you have upwards of 100 threads, as in the last problem, you are approaching a gigabyte of memory for your process, much of which is not used if the threads aren't doing much more than funneling requests into a buffer. So to try and get the same level of concurrency, we use the select() system call, which monitors all of the open file descriptors for activity, which in this case means the pipe has been written to and is ready for reading. This implementation allows us the same kind of concurrency because we do not need to wait for one file descriptor to be ready before working on another, but it uses far less system resources because each pipe doesn't need its own thread.

However, while the threads could execute asynchronously, meaning multiple threads could be executed at the same time if you are using a multicore processor, the select() system call is synchronous. It works by running within an "infinite" loop, and on each iteration notifying the system of which file descriptors can be written to or read from. We then must iterate among all the open file descriptors and read from them individually, meaning that at any given point only one is truly being read from at a time. So it is concurrent in that we do not need to wait for one file descriptor to be ready before executing the next--we simply pass over the ones not ready each loop--but it is not asynchronous because each file descriptor is handled within a single thread of execution.

In my implementation, I was able to get my program working about 99% correctly. Everything with the select system call works correctly, and it correctly exits the select loop, but for some reason that I was not able to debug, it hangs after this. Sometimes I will see the correct histogram from one Person, but no more. And yet, I knew it was still running mostly correctly, and completely correctly with respect to the main focus of this machine problem, the select system call. So to measure execution time, I was not able to measure from the beginning to end of execution, but I took the final measurement immediately after it would exit the loop, leaving only the time to print the histogram unaccounted for. So although the program must be forced close, I am confident that I still have correct results, which works correctly (except for the finish) for any number of requests, buffer size, and number of open channels. The results are shown in the table below, compared to the data from machine problem 3. The program is compiles with 'make' and run with 'make run'. The input flags can be changed inside the makefile, or it can be run with './client -n <number of requests> -b <size of buffer> -w <number of concurrent channels>

Number of open channels, buffer at 250	Execution time for 10,000 requests (sec)	Execution time from MP3 (sec)	Difference (sec)	Difference (%)
1	109.102868	108.05	1.05	0.97
2	54.245494	53.50	0.75	1.38
3	36.466360	35.88	0.59	1.62
4	27.169579	26.89	0.28	1.03
5	21.894077	21.56	0.33	1.54
10	10.942759	10.77	0.17	1.59
15	7.519793	7.22	0.30	4.07
20	5.943684	5.43	0.51	9.03
25	4.617428	4.37	0.24	5.51
30	4.277919	3.66	0.62	15.57
50	3.6345493	2.28	1.35	45.80
70	3.230935	1.71	1.53	61.56
100	2.127329	1.35	0.77	44.71

As the table shows, the resulting times are nearly identical. I decided to not take multiple runs and get the average time, because it takes a long time to run the code and tabulate that much data. But I do not feel it is necessary, because the trend is execution time from both MP3 and MP4 is the same, with difference in execution times typically less than a second, which gives a percent difference that increases as . Nevertheless, executions consistently between 0.1 and 1 second slower is likely just differences in the activity on the unix server where the code is being run, as major bottlenecks in the code would likely show as a more consistent percent difference in execution rather than absolute difference.

In the end, using the system call select to monitor file descriptors is a perfectly viable way to implement concurrency, and in fact is a better solution than having gross numbers of threads, which use a lot of system resources and must be synchronized correctly. It is also clear that this is a valid solution, because it is how sockets are established and monitored in networked computing, which will be the focus of the next machine problem.