Name: Brian Jon Stout
Date: 06MAY2017
Current Module: Operating System
Project Name: Signaler

Project Goals

The goal of the project is to create a program which automatically counts through the prime numbers. The program should also be able to receive SIGUSR1, SIGUSR2, and SIGHUP signals and act accordingly.

Considerations:

- Upon receiving the SIGUSR1 signal, the program must skip the next prime number
- Upon receiving the SIGUSR2 signal, the program must reverse the order it is printing prime numbers
- Upon receiving the SIGHUP signal, the program must start the count from 2

Initial Design:

Because of the smaller program size (only four functions) the program is entirely stored in the signaler.c file to avoid unnecessary confusion. The main function is responsible for running the argument checkers, setting up the signaler, and running the main logic for the prime number counting.

The signal handler function receives the signal and appropriately modifies the volatile sig_atomic_t variables used by the main function.

Data Flow:

The program starts by running the argv line and the argc data through the argument_checker function. The argument checker checks the argv line for common errors, like dashes without values afterwards, or negative values for the numbers.

Afterwards the program runs through the getopt() command till all the arguments have been parsed. S follows a number and specifies the starting point for the prime number. E follows a number and specifies the end point. R specifies that a count should decrement instead of increment. All the switch options set a flag so the program exits if multiple of the same switch options are used.

Following this the program checks to see if the count is reversed but still starting at 2. If it is the program stops since its pointless.

After this small check, the program sets up the signal handler struct, specifying that the signal_handler function will parse the signal commands. There's some quick checks to catch errors on the signals.

The final piece of the program is an infinite for loop (I have no idea why, it's in the style guide.) The first thing the loop does is check for a sighup signal, if there is, it resets the counter to 2 and resets the signal as well. The next check is to see if the program has passed the end number. It does this by first seeing if the end number is not 0 (has actually been set) and then to see if the counter is greater than that number.

After all this the program checks to see if the counter is on a prime number with the is_prime function. The is prime function checks common ways to eliminate prime numbers first (If it is divisible by 2 or 3) Then brute forces every number up to the square root of the counted number. When the program determines that a number is prime, it checks to see if the sigusr1 flag is set, and if it is it will skip the printing portion of the prime number check. If it's not, it checks to see if the last printed prime number was the current prime number, to avoid printing out a number more than one if the count gets reversed. When all this has been checked, it will print out the PID of the process and the number which is prime. If the counter is less than or equal to 2 and it's going in reverse, than the program will exit to avoid counting into the negative numbers. The last step is to usleep for 3 quarter of a second, to give the appearance that a number is being printed every second.

The loop then checks to see if it should be decrementing the counter or incrementing the counter.

Communication Protocol:

Not applicable.

Potential Pitfalls:

The most obvious pitfall is related to the approximately one number every second. THe easiest way to do this is hold up the program and have faith that it'll finish its calculations fast enough. The problem is with very large numbers the calculations might take so long it will hold up the program. Having an efficient primality check is important to avoid this pitfall, but more could be done so that the calculations are done regardless of the timing so that earlier numbers aren't printing out faster than later numbers, the numbers will just print out after a second has passed, and the calculations get finished.

Test Plan:

User Test:

Run the program from various different inputs (size, sign, and empty strings) to test properly for input sanitation. Afterwards, send various signals to test the logic for the program, then spam them to stress test the program.

Test Cases:

The program handles all the input I could think to throw at it. Runs a bit slow at higher numbers, but that's to be expected since I didn't implement a much more complicated primality test.

Conclusion:

The program was simple but a good introduction to signal uses. I wanted to avoid overcomplicating the program by fixing the timing issue, or implementing a more complicated cacheing version of the primality test, but everything is optimized to meet the basic requirements.

A good solution to the timing problem is just to save the time (in microseconds) when the last check was completed, and check the time when the current check was completed. If the difference is greater than a second, print out the number, else sleep for the difference.

Another solution is to keep the timing on a separate which sends a signal every second that sets good to print flag, which gets reversed whenever a value gets printed.