Assignment 1 Data transfer

Mikael Kajasvirta 899363

1. Introduction

Before the development of the program few implementation methods were decided. When morse encoded data is sent between two processes, it is needed to distinguish two separate symbols from each other. It is possible to, for example, either keep short pauses between the symbols or send a different signal between them. For better efficiency and easier implementation it is better to use an additional signal. We also need to decide how we implement the two programs which communicate between each other. It was decided that only fork() is used in the implementation. Lastly, the charset used is the same as in the testbench.

2. Development

2.1. Structure

The program is separated into three files: a main source file where sending and receiving signals takes place; a library file which has function to initialize the input and output file descriptors and functions to encode to and decode from morse code; and a file to hard code the morse table which contains the respective morse codes for each character.

2.2. Command Line Argument Functionality and Forking the Process

The development of the program was started by implementing the command line argument functionality which determines the input and output sources. The functionality was implemented by referencing the *as1_mockup.c*-file. After that the process was forked using fork()-function. The forked child process acts as the program B and the parent process as the program A.

2.3. Signals

After successfully forking the process, a simple implementation of the communication between the processes was created and tested. The communication was implemented by using a pipe to buffer signals to ensure that there are no missed signals. In the first implementation no data was read or written to any file. The goal was to only determine the functionality of the communication between two processes. In the first version, when sending two signals to the parent process A only one of the sent signals were received. After some thought it turned out that the signal handler and pipe were initialized in the wrong place, thus, no second signal was sent.

At first data was transmitted using SIGUSR1, SIGUSR2 and SIGALRM. SIGUSR1 referred to a dot, SIGUSR2 a dash and SIGALRM end of symbol. Later a problem arose when many signals were sent in a short duration. At times, signals would be lost, resulting in wrong symbols received and occasional error. This was a result of using so-called normal signals instead of real-time signals. This problem was more time consuming than initially expected, though the solution was not complex in the end. Initially it seemed that the problem was that the signals were lost when reading data from the pipe. If a signal interrupted the read call it would simply be restarted, since the SA_RESTART flag was set in the signal handler as in the examples. This would then result in lost signals. To fix this, an implementation was tried where the parent process would signal back to the child process after receiving a signal to allow the child process to send the next signal. This implementation faced a similar problem where the communication was not reliable and only parts of the data were transmitted.

The final implementation uses only real-time signals with the sa_flags SA_RESTART and SA_SIGINFO set. However, compared to the previous implementations only the SIGRTMIN real-time signal is used with three different sival_int values. Because the morse table was already created at this point SIGUSR1 or 10 denotes a dot, SIGUSR2 or 12 a dash, and SIGTSTP or 20 end of symbol. To denote end of transmission two SIGTSTPs are sent in a row. This implementation passes the testbench and works relatively fast.

2.4. Data Input and Output

There are different ways to read the data from the input and write the data to the output. For example, data could be read in blocks of size n and then processed accordingly. However, in this case it was easier to read a character at a time from input, and then send them individually before reading the next character. Then the parent process A writes the received characters to the output.

2.5. Encoding and Decoding

The encoding is done with help of the code() -function found in the library file lib.c. The function first converts the character to uppercase then loops through the charset to see if it is found in the charset.

The decoding is done in a similar way looping through the morse table and comparing the signals with each other. For example if the character A was sent, the receiver would have to decode the following set of signals: {10, 12, 0, 0, 0, 0}. Then the decode function finds the index of the morse table where the set of signals were the same and returns the corresponding index of the character set.

To implement the encoding and decoding functions in this way, the character set and morse table must be in the same order. However, it is possible to make the functions more efficient by using a more optimized data structure when storing the sets, for example, using a hash table. Since the character set is so small, it is sufficient to use a naive approach and simply loop through the sets without compromising the efficiency of the program too much.

It would also be possible to implement the morse table dynamically where the morse coded characters were read from a file, but implementing this turned out to be too time consuming without any real benefits, thus, it was smarter to simply hard code the morse table into a source file.

2.6. Problems With the Very Long transfers

When running the testbench with the very long files (-l flag) the child process needs to send multiple million signals to transmit the data. At some point the sigqueue buffer was reached and the program would halt. One way to solve this problem would be to implement some kind of dynamic flow control. In theory, when the buffer is full the child could wait and send the same signal again after the buffer is full no more. However, in this case there was a much simpler solution, which would allow the program to pass the testbench with the very long files flag. Since the transmitter was sending the signals too fast, an easy way to solve this was to throttle the transmission rate by sleeping 100ms every time a symbol is sent. This shouldn't affect the transfer rate, since the bottleneck seemed to be the receiver and decoding. The program was initially tested on an Intel i7-12700k and passed all tests.

3. Logs and Errors

The program creates three separate log files found in the Logs folder which is created when the Makefile is executed. There is one log file for the whole program (Log.txt), one for the child process (Log_tx.txt) and one for the parent process (Log_rx.txt). This is done to avoid racing conditions when writing to the log file, therefore, making the implementation easier and simpler.

In most error cases the program writes to the corresponding log file and prints an error message to the console. In some cases the user is only notified in the console, for example, the error messages which report failed opening operations for the input and output files. The error functionality could be improved, but the implementation, though simple, functions in practice.

4. Testing

The program was developed on a debian virtual machine. The program was tested continuously throughout the development process using short and log test files and inputs. With some implementations only short files were transmitted correctly or sometimes the program would break unexpectedly and even crash the development VM. These errors were usually due to bad error handling, segmentation faults and wrong use of signals.

The encoding, decoding and initializing functions were tested separately so further development was easier and more straightforward. In the end the program was tested multiple times with the testbench to ensure no unexpected behavior would occur during execution of the program.

To ensure compatibility with other systems the testbench was successfully run in a vdi.Aalto Ubuntu system with the very long files (-l) flag.

5. Improvements

The assignment was surprisingly difficult and time consuming. However, in my opinion it taught the basics of signals very well. Though, signals are not usually used in this kind of way. Maybe an alternative for the first assignment would be to have several smaller tasks which cover the most important aspects of the first part of the course. I can see how the task could be quite intimidating at first and maybe some kind of exercise/help session at most once a week would improve the course overall and make learning the material more efficient.