Assignment 1 Report
ECSE427 Operating Systems

Owais Khan
ID: 260617913
owais.khan2@mail.mcgill.ca

October 6, 2018

1 Introduction

This report pertains to the first assignment of this course. The main aim of this assignment was to implement a tiny shell in various different ways, measure the time each takes to execute a predetermined set of commands and then explain any differences in performances of the different versions.

This report will first outline the different implementations and how they differ from each other, followed by their execution times for the commands run on them and finally explain the differences in performances.

The final part of this assignment requires us to implement inter process communication with the use of named pipes (FIFOs) .

2 Implementation using system()

In this version of the tiny shell we use a linux library function called system() to execute the commands entered by the user. This function creates a child process and then executes the specified shell command inside the child process. After executing the command the child process terminates and system() returns with an integer that specifies the exit status of the command that was run.

After successfully implementing this the tiny shell was tested with a mixture of commands that executed properly and also those that crashed or terminated prematurely. The behaviour of the shell was as expected, the failure of the commands running in the child process did not affect the calling parent process and the shell continued to prompt the user for the next command. In the case of commands that executed properly, the output of the commands was seen on stdout and after the execution completed the tiny shell was ready to accept the next command.

3 Implementation using fork()

In this and all subsequent implementations of the tiny shell we were required to write our own implementation of the system() function using various system calls available at our disposal. The functionality that my_system() needs to demonstrate is that it should spawn a child process and inside that child process it should run the command passed in by the user. The parent process should then wait for the child process to complete before asking the user for the next command. In addition to this the tiny shell must also be fault tolerant in that any failure in the execution of the child process should not make the tiny shell crash.

The behaviour of the fork() function is that after it is called it creates an exact copy of the parent process and then two instances of the same program run simultaneously. fork() returns 0 in the child process and the PID of the child process is returned in the parent process. This behaviour can be used for conditional execution where the child process executes the user command using execvp() (one of the variants of the exec family of functions which is used to replace the current process image with a new image) and the parent process waits for the child process to complete execution using the waitpid() method. The child process uses an integer to capture the return status of the execvp() command and on failure it prints out an error message to stdout, otherwise it prints the output of the command to stdout. Once the child process completes the parent process proceeds and prompts the user for the next command.

4 Implementation using vfork()

The vfork() method is very similar to the fork() method in execution and was created to make the process creation step more efficient. Making an exact copy of the parent process requires a lot of time and is a waste of work in this situation because the process image eventually gets replaced by a new process image. So the process created by vfork() shares the same memory space as the parent until the child process calls one of the functions from the exec() to replace the process image. One difference in execution of fork() and vfork() is that the parent process does not execute until the child process calls the exec() function. After the funtion is called both processes run concurrently which is not what we want so we again use the waitpid() function in the parent process to wait for the child to finish execution before getting the next command. The child process again captures the return status of the execvp() command using an integer and handles errors in a similar manner to the fork implementation.

As can be seen vfork() provides a much more efficient way of spawning processes saving on processing time and memory.

5 Implementation using clone()

The clone() function is a Linux specific lower-level system call that enables us to create processes. Like fork(), the clone() function also creates a copy of the parent process image but with the use of flags it gives us a lot more fine grained control as to what is shared between the two processes. By manipulating the flags the desired performance and functionality can be obtained. The entire list of flags can be found in the man page for the clone() function.

In this implementation of the tiny shell, in addition to all the requirements of the previous versions we had to make sure that the **cd** command works properly. Inorder to do that the processes need to share the same filesystem and must not work on a copy of their own. To ensure this we pass in the CLONE_FS flag which makes both processes share the same filesystem. Also to ensure that the parent process waits for the child process to terminate before resuming execution we pass the **__WCLONE** to the waitpid() method. This flag makes the parent process wait for all *cloned* children to terminate before continuing.

6 Timing

This section discusses how the timing functionality was implemented to time the various commands run on the different shell implementations.

A C library function **clock_gettime()** returns the current time in a timespec construct. The timespec construct has two fields *tv_sec* and *tv_nsec* which store the seconds and nanoseconds respectively. Inorder to facilitate code readability a helper method gettime() was defined in the provided myheader.h file. The gettime() method converts the time into milliseconds and returns it. To calculate the running time we call gettime() on either side of the my_system() call and after completion we subtract both times to get the execution time in milliseconds. The following table includes the timing information for some of the commands tested on the shell.

Command	system() / ms	fork() / ms	vfork() / ms	clone() / ms
“ls -al”	2.630	1.716	1.480	1.534
“pwd”	0.717	0.700	0.626	0.711
“date”	1.357	0.792	0.695	0.721
“gcc hello.c -o hello”	56.07	51.48	50.98	51.72
“whoami”	0.776	1.087	1.014	1.100

Table 1: Comparison of execution times of commands in the different versions of the tiny shell

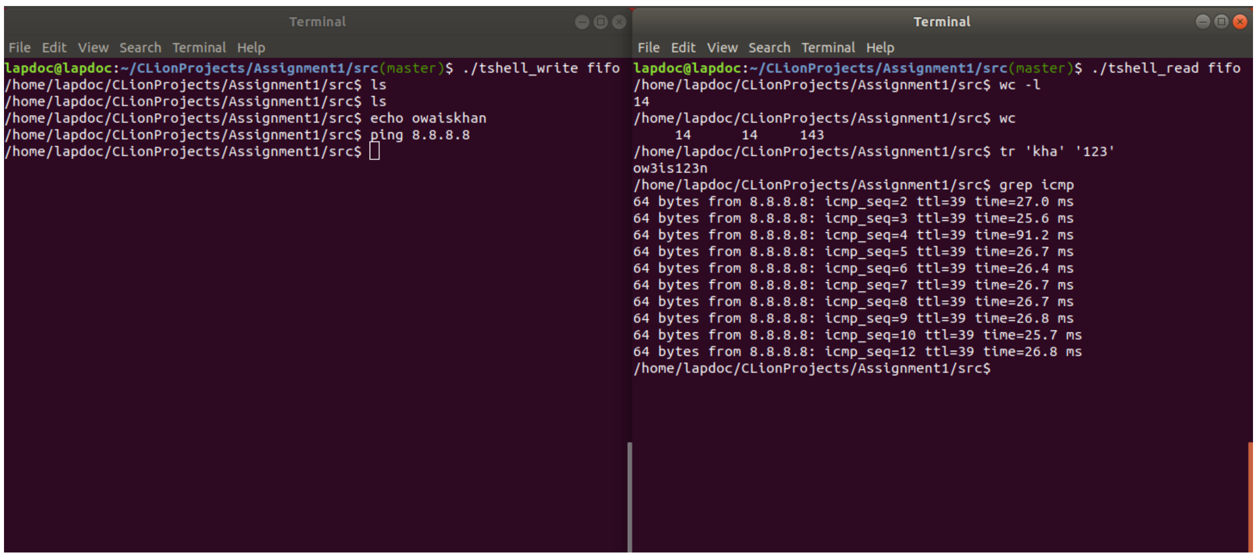
7 Interprocess Communication using FIFOs

The last part of this assignment required us to implement command piping using a named pipe or FIFO. Command piping is the process of piping the output of one command as input to another command. The general idea is to rewire the stdout of the writing process to write to a FIFO instead of stdout and rewire the reading process to read from the FIFO instead of stdin.

We take the name of the FIFO as a command line argument and differentiate between the reading and writing versions through compiler flags (PIPE_READ and PIPE_WRITE). The program first checks if the name of the FIFO was passed otherwise it exits immediately. There is no check to see if the FIFO exists (assumed to be created beforehand).

In the writing version of the shell we use open() to open the FIFO for writing only and then duplicate the file descriptor returned on stdout. This was the output of all commands gets redirected to the FIFO instead of being shown on the screen. Similarly in the reading version of the shell we use open() to open the same FIFO, but this time for reading only. The file descriptor returned in this case is duplicated on stdin so that all input comes from the FIFO instead of from stdin.

The prerequisite for this implementation to run properly is that the writing version should be run first with a command that produces output and writes it to the FIFO after which the reading version should be used so that the FIFO contains something that can be used as input for the command being run. Also the commands run in the writing version should only execute commands that MUST write to stdout and conversely the reading version should only execute commands that MUST read from stdin. The following image shows a few examples of this in action.


Terminal
File Edit View Search Terminal Help
Terminal
File Edit View Search Terminal Help

```
lapdoc@lapdoc:~/CLionProjects/Assignment1/src(master)$ ./tshell_write fifo
/home/lapdoc/CLionProjects/Assignment1/src$ ls
/home/lapdoc/CLionProjects/Assignment1/src$ ls
/home/lapdoc/CLionProjects/Assignment1/src$ echo owaiskhan
/home/lapdoc/CLionProjects/Assignment1/src$ ping 8.8.8.8
/home/lapdoc/CLionProjects/Assignment1/src$
```

```
lapdoc@lapdoc:~/CLionProjects/Assignment1/src(master)$ ./tshell_read fifo
/home/lapdoc/CLionProjects/Assignment1/src$ wc -l
14
/home/lapdoc/CLionProjects/Assignment1/src$ wc
     14      14     143
/home/lapdoc/CLionProjects/Assignment1/src$ tr 'kha' '123'
ow3is123n
/home/lapdoc/CLionProjects/Assignment1/src$ grep icmp
64 bytes from 8.8.8.8: icmp_seq=2 ttl=39 time=27.0 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=39 time=25.6 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=39 time=91.2 ms
64 bytes from 8.8.8.8: icmp_seq=5 ttl=39 time=26.7 ms
64 bytes from 8.8.8.8: icmp_seq=6 ttl=39 time=26.4 ms
64 bytes from 8.8.8.8: icmp_seq=7 ttl=39 time=26.7 ms
64 bytes from 8.8.8.8: icmp_seq=8 ttl=39 time=26.7 ms
64 bytes from 8.8.8.8: icmp_seq=9 ttl=39 time=26.8 ms
64 bytes from 8.8.8.8: icmp_seq=10 ttl=39 time=25.7 ms
64 bytes from 8.8.8.8: icmp_seq=12 ttl=39 time=26.8 ms
/home/lapdoc/CLionProjects/Assignment1/src$
```

Figure 1: Example of pipe implementation