Assignment 1: Tiny Shell Report
ECSE 427: Operating Systems

Muhammad Anza Khan
ID: 260618490
muhammad.anza.khan@mail.mcgill.ca

October 6, 2018

1 Introduction

I am writing this report to outline all the work I did regarding the first assignment for ECSE 427 this semester. The main goal of this assignment was to essentially write a version of the Linux shell using different systems calls and than testing their performances relative to each other by calculating the time they took to execute various Linux shell commands. Also, along with the various shell implementations, we also had to perform interprocess communication between two unrelated processes using named pipes i.e **mkfifo** system call.

In this report, I will attempt to walk the reader through the design process of each version of the shell, explaining what the steps I took and the rational behind. I will than also compare each version of the shell and provide an explanation for the results and possible improvements to the code that could have yielded better results.

One more thing which I should mention here as it pertains to all versions of the tiny shell I made, the exit status for the child was captured by making a local variable called child_exit_status and checking its return value, if -1 that meant a failure. I do this for all versions of tiny shell other than the implementation using system().

2 tiny shell using system()

This was the first part of the assignment which asked us to implement the shell using the Linux library function **system()** to run the command entered by the user, taken as a string. When **system()** was called it would create a child process which would than run the command entered by the user and execute the command entered, so say the user entered **ls**, a child process would be created to run ls and print the contents of the current working directory. Once the child process would finish executing, system returns the exit status via the integer value returned by system().

I tested this implemenaton of the shell by entering several different commands to test how versatile it was, I particularly focused on edge cases that would crash the process or exit it before successful completion. An example of this was entering nothing and than running system to see how it would handle nothing being entered, whether it would exit or ask the user to enter a proper command. The shell performed pretty much as expected, commands such as **ls** would print to stdout and ask the user for another command.

3 tiny shell using fork()

In this part of the assignment, we were asked to implement the tiny shell using the Linux system call **fork()**, this is a system call that creates a new process which is an identical copy of the calling process which is the parent process of the newly created child process. As I mentioned the child process is an exact copy of the parent in very sense of the phrase, from the code it runs to the file descriptor table it gets, everything is identical.

The reader at this point is wondering what is the point of going through such an exercise to create another process which is exact copy, that seems interesting nor particularly educating to the student at all. Well to that end reader, I say please do not throw this report away in boredom for the fun is just beginning. Now fork() is one of the few Linux functions that returns twice, it returns **0** in the child process and in the parent process returns the process ID **(PID)** of the child process. Based on the return value, we can check whether the child is running and within the child process we run another system call called **execvp**, this is a system call that belongs to the **exec** family of system calls.

What it does is basically changes the memory image of the currently running process and runs the one entered by the user. What does this mean exactly? Basically it means that say the user enters a shell command, our old friend ls will come handy here. What happens when execvp or any exec family member is called is changes the currently executing process to run the command entered by the user, in this change it runs the binary matching to ls and executes ls. Within the parent process, I call **waitpid()** which is a

system call we call for the parent to wait for the child process to complete its execution. Once done, we ask the user to enter another command and the cycle runs again.

4 tiny shell using vfork()

Following our implementation of the shell using fork, we had to do a similar exercise but this time we had to use the **vfork()** system call instead of fork. vfork is similar to fork in the sense that it too creates a child process but it is slightly more efficient. This improvement comes in the way that memory is handled between the child and parent process. As we saw earlier in the fork section of this report, the resulting child process in fork was an in essence a copy of the parent but ultimately got replaced very quickly to run a new binary matching the command entered by the user. Creating a new address space, allocating and establishing all the kernel per-process data structures just to replace them again within the next 10 lines of code is an **extremely inefficient** way to create a shell.

This is where the optimizations introduced by vfork greatly show its improvement over fork. The child process created by vfork shares the memory space with the parent until the child calls execvp (or really any family member of the exec family) to replace its binary image. This is a great improvement because you reduce the overhead to create a new process and set up all the process details by a substantial fraction. Unfortunately, once execvp (in my case) is called, the child and parent compete concurrently for the CPU thus we must again resort to our old friend waitpid to have the parent wait for the child process to finish its run and than ask the user for another command.

5 tiny shell using clone()

This implementation of the shell was done using the **clone()** system call. The clone() is a Linux specific lower-level system call for creating processes. You pass the appropriate flags to get different behaviors from the clone() system call. Like the fork() call, the clone() makes another replica of the calling process. However, the sharing between the caller (parent) and child can be precisely controlled by the flags. Also, the child runs the func specified in the call and the parameters can be passed to this function. We need to specify a stack for the child as well. The hardest thing about this implementation of the shell was that we had to introduce support for the Linux shell command **cd** which changes the current working directory. This for me was extremely challenging because I did not know how to proceed initially because firstly none of the previous implementations required us to support the cd command.

But this is where the brilliance of the flags come in. As I mentioned earlier, clone has several flags to make it more versatile than fork, the flag I used to introduce support for the cd command was the CLONE_FS flag. If CLONE_FS is set, the caller and the child process share the same filesystem information. This includes the root of the filesystem, the current working directory, and the umask. If CLONE_FS is not set, the child process works on a copy of the filesystem information of the calling process at the time of the clone() call. This allowed me to introduce support for the cd command. I should also mention I use the flag If CLONE_VFORK as well. If this flag is set, the execution of the calling process is suspended until the child releases its virtual memory resources via a call to one of the exec() family members or exit() (as with vfork()). If CLONE_VFORK is not set, then both the calling process and the child are schedulable after the call, and an application should not rely on execution occurring in any particular order.

Thus using these two flags I was able to make in my opinion probably the most optimized version of the tiny shell because it allowed us much greater controlling in resource allocation and also allowed us to execute far more commands than the fork() brothers.

6 Timing

Command	system() / ms	fork() / ms	vfork() / ms	clone() / ms
“ls -alt”	2.780	2.160	1.940	2.240
“pwd”	1.160	1.080	0.800	1.101
“date”	1.730	0.820	0.710	1.008
“gcc hello.c -o hello”	64.100	101.810	99.130	57.810
“cd”	NA	NA	NA	0.310
“whoami”	2.450	1.190	1.630	1.210

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

To calculate the timings for the various shell commands entered I used the Linux library function **clock_gettime()** by calling it twice, once immediately before the user enters the command which was when the clock started and the second time I called it after it was done executing the command, that became the stop time. Taking the difference of the two yielded the time that elapsed for the child process to execute the command in milliseconds.

If we inspect the values in the table we clearly see an indication for the performance for the system calls that were used to process those commands. As expected, the biggest gap we notice is between fork() and vfork(). This can be attributed to the fact that vfork() is a much optimized version of fork() because there is no creation of a separate address in vfork as is in the case of fork, because the child and parent share the same address space in vfork() and the parent is suspended in vfork() until a member of the exec() family is called or until the child exits.

But what is even more interesting is the performance for clone(). As we saw earlier, clone() is much more optimized than fork() and vfork() because the designer can tailor it to his or her leisure by controlling exactly what they want by setting the various flags accordingly. Indeed we can see that the more appropriately one selects the flags, the more finely grained performance one can get. In my case I use the flags CLONE_FS flag and CLONE_VFORK flag. If I were to delve into the man pages in more detail I am sure I could find even better flags which could improve performance by a greater factor. Indeed such is the performance of clone against the forks that compilation with clone takes half the time it does in those two.

I believe a word or two at this point is warranted about our solitary friend the Linux library function system(), I do not think we can get much improvements here because all we do is literally pass it a string and it performs all the process management under the hood and does not allow us to optimize it any way, thus the timings that I got for system were pretty much invariant and were middle of the road, not too bad but at the same time left much to be desired.

7 Interprocess Communication using FIFOs

The final part of the assignment was for me quite simply the hardest part of the assignment for me because it required an understanding of how Linux does interprocess communication (IPC). Now this is not that hard, as we have seen in class how IPC is done, but that was betweem related process i.e between parent and child, this was something that the professor explain to us in great detail and depth, this is called anonymous piping. Performing IPC between unrelated process which is performed using named pipes was something I simply could not wrap my head around at the beginning but after much internet surfing, I got a working understanding of how this version of IPC works. Using that as a launching pad, I began coding.

To explain what I did, I will firstly discuss the writing end, followed by the reading end of the fifo. I initially decided to combine the reading and writing ends of the pipe in one file but immediately I was running into many errors from the onset. These errors had to do with invalid file descriptors being returned. After spending hours trying to run backtraces on my core dump, I decided to separate the reading and writing ends into two separate files and this slashed my troubles by a massive factor.

The writing end of the fifo closes stdout and dups that into the file descriptor that gets passed to the fifo that you provide to the code by running **mkfifo** followded by the appropriate modifications. This ensures that anything writing to stdout now instead writes to the writing end of our fifo. For reading I did an almost exact exercise but instead closed stdin and rewired that to the reading end of our fifo. As a result anything taking input from stdin now will instead take it from the reading end of our fifo.

Now the only real issue I have with regards to this part of the assignment is that it works fine if you write first followed by read, i.e it works perfectly when you firstly write to the fifo and than read whatever got written. An example of this was running **ls** and than **cat**. However the reverse I noticed did not work, i.e running read before a write causes the program to hang. I believe what happens is that since read expects some input in order to work, it blocks until it gets some input to read, that input should come from write and as it is not there, it will just stay there blocked and not make a process state change until something is written to the fifo.