Sadhvi Mehta
260747607
ECSE-427
6th October, 2018

Mini Assignment 1 – Report:

Section 1: Observations in implementations of system() call:

Fork():

When using fork(), the implementation required the spawning of two processes: a child and parent. There were two critical aspects to deal with over here: waiting for the child to finish and error handling. To ensure that the child process executed before the termination of the parent process, a simple wait() command was used. More importantly, the returned value of this wait() command was needed to handle and log any errors that might have occurred in the child process. This was done by passing the address of a variable named 'status' to store the result of the child process. Following this, it was important to decipher whether the execvp() command within the child process ended due to a success, error in passed-in arguments, or because of a signal. This was done by passing the return value in 'status' to the macros WIFEXITED and WIFSIGNALED. The former would check if the process exited successfully or not, meaning if the passed in input-command was valid or not. The latter was responsible for checking if some sort of signal was raised when execvp() tried to execute the input-command. In this way, I was able to differentiate as to whether the input-command itself was incorrect or that some sort of fault such as a segmentation fault was raised while trying to execute a valid command. Lastly, it is also important to note that the child process always terminated after every attempt at executing a command by either calling exit() or a successful run of execvp().

VFork():

VFork was extremely similar to fork() except for two main differences. The first was since vfork() makes child and parent share same address space, one had to avoid performing any operations within the child process except for calling execvp() and exit(). This was to prevent the data within the parent process from getting corrupt. Also, the second difference was that since the child and parent share the same address space, one was forced to _exit() and not simple exit(). This is because _exit() does no cleanup before process exit and thus keeping the data of parent process as it originally was.

Clone():

Clone was noticeably different from fork() and vfork(). The first difference was that the child process does not just begin from where the clone() function was called. On the other hand, it begins from the passed in function and terminates when that function is returned or exited. The second, more prominent difference is that clone() allows the child to share same address space as parent. Thus, to properly set up the environment for the child process, we had to choose the right flags to pass to the clone function. For my implementation, I turned on the flags CLONE_VFORK, CLONE_FS, and the SIGCHLD macro. I turned on CLONE_VFORK because I needed to suspend the parent process until the child executed the given command through execvp(). I needed to turn on CLONE_FS as the assignment wanted us to implement the 'cd' command where a change in cwkdir of the child should affect that of the parents's. This could only be done if the two processes shared a filesystem. Lastly, I discovered that my error handling during my clone() implementation was ignoring SIGCHLD signal. This in turn was not allowing my parent process to recognize any child processes if the execvp() command failed. Consequently, my error handling was also failing. However, by passing the SIGCHLD signal, I was able to let my parent process know exactly when my child process terminated as well as with what exit status. The error handling was then the same as that of fork() and vfork().

Section 2: Implementation of FIFO:

The named pipe (aka: FIFO) was implemented by simply creating a pipe using the mkfifo() command with its respective arguments. To actually have one end of the pipe act as one process's stdin and another's stdout, it was a simple toggle of entries within the respective file descriptor tables. For the process who was writing to the pipe, one simply had to close the default stdout file descriptor ('0') and then open the FIFO's file descriptor within the process. This is because a file descriptor takes on the earliest empty entry within the table. The same mechanism was applied to the reader of the pipe except the stdin file descriptor was now closed ('1'). Also, the respective processes were given either only write or only read permissions over the named pipe. Apart from that, the rest of the implementation was the same for actually executing the passed in commands.

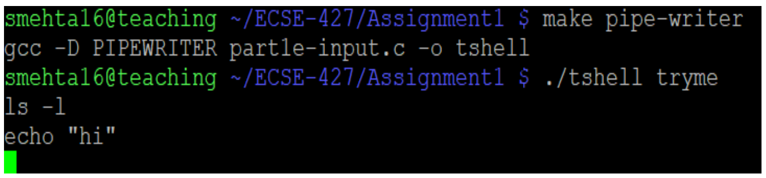
Section 3: Timings:

	System()	**Fork()**	**Vfork()**	**Clone()**
Commands:				
ls -l	11663.750 microseconds	27054.25 microseconds	8754.25 microseconds	12983.0 microseconds
echo "hi"	1763.25 microseconds	2240.25 microseconds	2441.75 microseconds	2996.75 microseconds

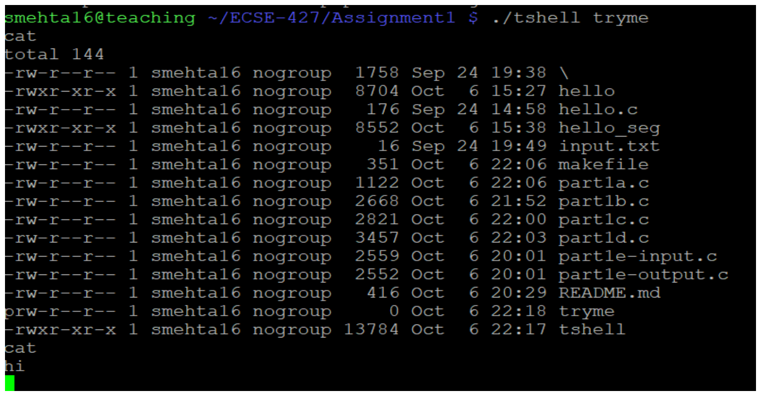
Above the results are shown for the timings. For the most part, system() and vfork() are the fastest. This makes sense as vfork() does not copy the address space as fork(). This also helps explain why fork() is the slowest out of all implementation because it copies the address space for the child.

Section 4: FIFO Testing Results

Below, you can find the test cases showing that the FIFO actually works. (Note: reader on bottom, writer on top).



```
smehta16@teaching ~/ECSE-427/Assignment1 $ make pipe-writer
gcc -D PIPEWRITER part1e-input.c -o tshell
smehta16@teaching ~/ECSE-427/Assignment1 $ ./tshell tryme
ls -l
echo "hi"
```



```
smehta16@teaching ~/ECSE-427/Assignment1 $ ./tshell tryme
cat
total 144
-rw-r--r-- 1 smehta16 nogroup  1758 Sep 24 19:38 \
-rwxr-xr-x 1 smehta16 nogroup  8704 Oct  6 15:27 hello
-rw-r--r-- 1 smehta16 nogroup   176 Sep 24 14:58 hello.c
-rwxr-xr-x 1 smehta16 nogroup  8552 Oct  6 15:38 hello_seg
-rw-r--r-- 1 smehta16 nogroup    16 Sep 24 19:49 input.txt
-rw-r--r-- 1 smehta16 nogroup   351 Oct  6 22:06 makefile
-rw-r--r-- 1 smehta16 nogroup  1122 Oct  6 22:06 part1a.c
-rw-r--r-- 1 smehta16 nogroup  2668 Oct  6 21:52 part1b.c
-rw-r--r-- 1 smehta16 nogroup  2821 Oct  6 22:00 part1c.c
-rw-r--r-- 1 smehta16 nogroup  3457 Oct  6 22:03 part1d.c
-rw-r--r-- 1 smehta16 nogroup  2559 Oct  6 20:01 part1e-input.c
-rw-r--r-- 1 smehta16 nogroup  2552 Oct  6 20:01 part1e-output.c
-rw-r--r-- 1 smehta16 nogroup   416 Oct  6 20:29 README.md
prw-r--r-- 1 smehta16 nogroup     0 Oct  6 22:18 tryme
-rwxr-xr-x 1 smehta16 nogroup 13784 Oct  6 22:17 tshell
cat
hi
```