Eric Sun
260673824

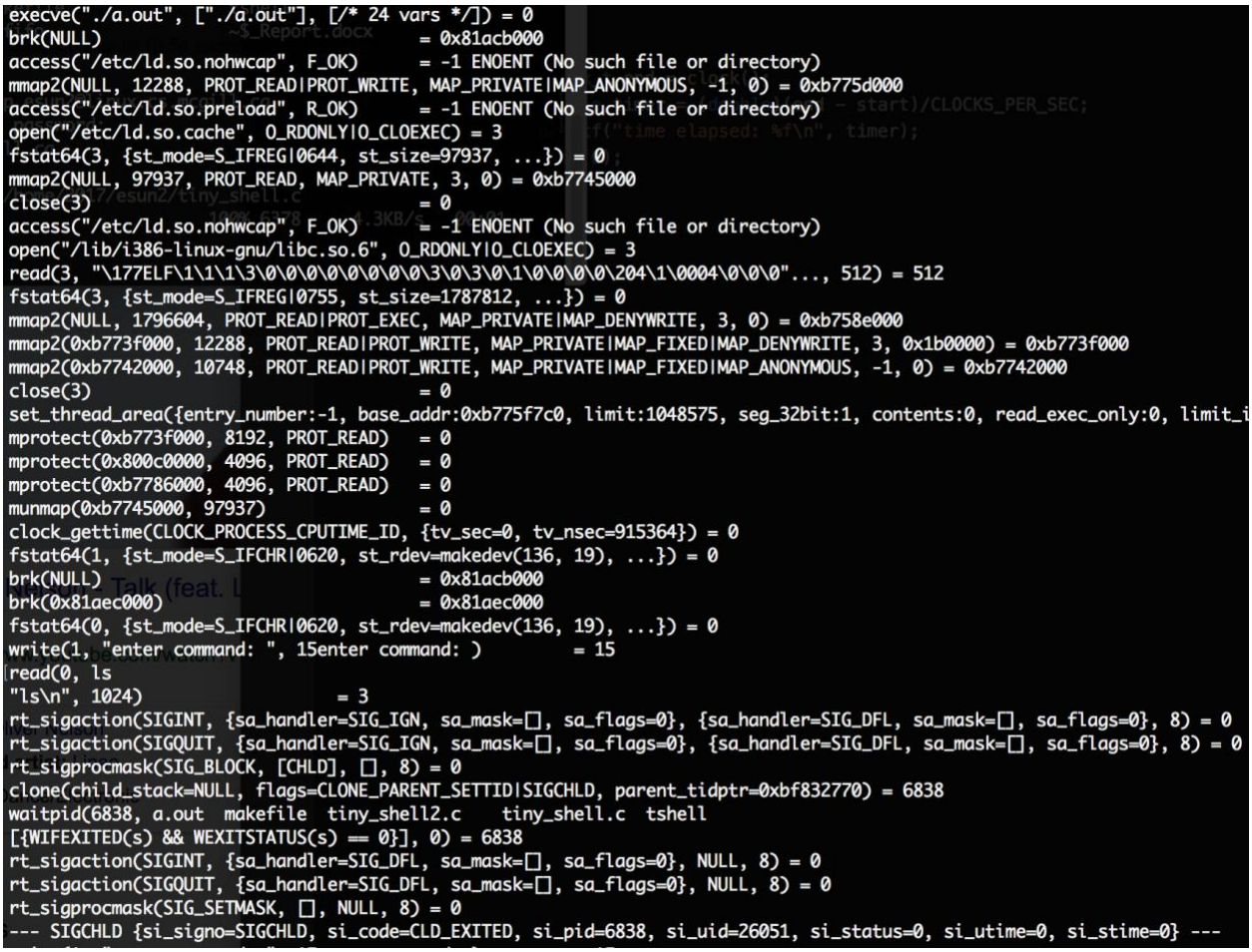
COMP 310 Assignment 1 Report

Assignment Questions

Try commands that crash or prematurely terminate as well as commands that complete their execution. What happens to your shell? Does your shell crash when a command crashes?

> When we input a command that prematurely terminates or crashes (i.e. the hello_seg program) the shell indicates an error was encountered and returns to the prompt. So the shell does not crash, it simply returns back to its default state.

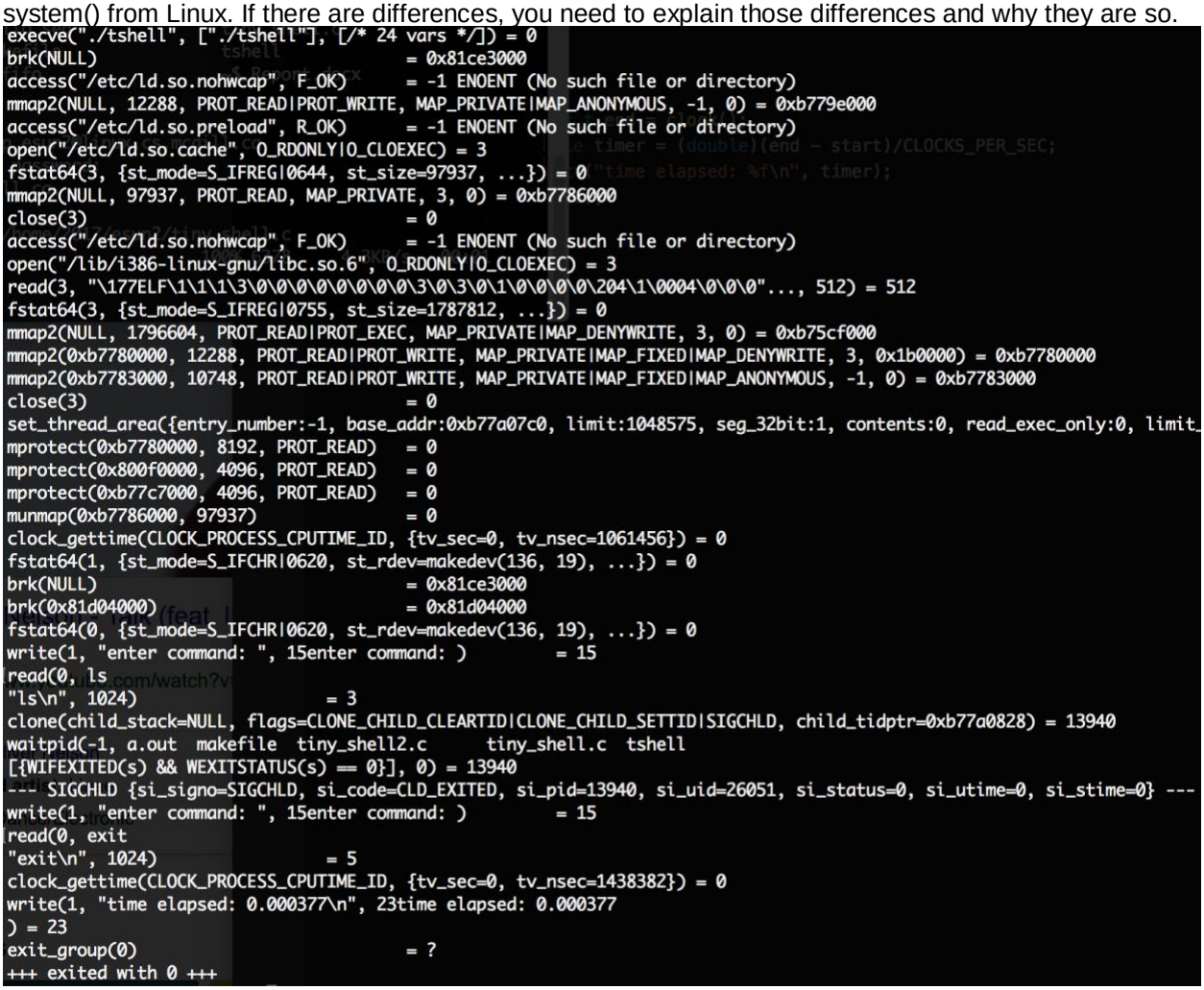
Now can you identify how the library routine system() should be implemented? That is, determine the system calls that are used in implementing the system() function.



```
execve("./a.out", ["./a.out"], [/* 24 vars */]) = 0
brk(NULL)                               = 0x81acb000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
mmap2(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb775d000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=97937, ...}) = 0
mmap2(NULL, 97937, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7745000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib/i386-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\1\1\1\3\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\0\204\1\0004\0\0\0"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1787812, ...}) = 0
mmap2(NULL, 1796604, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xb758e000
mmap2(0xb773f000, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1b0000) = 0xb773f000
mmap2(0xb7742000, 10748, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0xb7742000
close(3)                                = 0
set_thread_area({entry_number:-1, base_addr:0xb775f7c0, limit:1048575, seg_32bit:1, contents:0, read_exec_only:0, limit_i
mprotect(0xb773f000, 8192, PROT_READ)   = 0
mprotect(0x800c0000, 4096, PROT_READ)   = 0
mprotect(0xb7786000, 4096, PROT_READ)   = 0
munmap(0xb7745000, 97937)               = 0
clock_gettime(CLOCK_PROCESS_CPUTIME_ID, {tv_sec=0, tv_nsec=915364}) = 0
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 19), ...}) = 0
brk(NULL)                               = 0x81acb000
brk(0x81aec000)                         = 0x81aec000
fstat64(0, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 19), ...}) = 0
write(1, "enter command: ", 15enter command: )        = 15
read(0, ls
"ls\n", 1024)                   = 3
rt_sigaction(SIGINT, {sa_handler=SIG_IGN, sa_mask=[], sa_flags=0}, {sa_handler=SIG_DFL, sa_mask=[], sa_flags=0}, 8) = 0
rt_sigaction(SIGQUIT, {sa_handler=SIG_IGN, sa_mask=[], sa_flags=0}, {sa_handler=SIG_DFL, sa_mask=[], sa_flags=0}, 8) = 0
rt_sigprocmask(SIG_BLOCK, [CHLD], [], 8) = 0
clone(child_stack=NULL, flags=CLONE_PARENT_SETTID|SIGCHLD, parent_tidptr=0xbf832770) = 6838
waitpid(6838, a.out  makefile  tiny_shell2.c    tiny_shell.c  tshell
[{WIFEXITED(s) && WEXITSTATUS(s) == 0}], 0) = 6838
rt_sigaction(SIGINT, {sa_handler=SIG_DFL, sa_mask=[], sa_flags=0}, NULL, 8) = 0
rt_sigaction(SIGQUIT, {sa_handler=SIG_DFL, sa_mask=[], sa_flags=0}, NULL, 8) = 0
rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0
--- SIGCHLD {si_signo=SIGCHLD, si_code=CLD_EXITED, si_pid=6838, si_uid=26051, si_status=0, si_utime=0, si_stime=0} ---
```

> Ignoring the signal handling calls that system() calls, which we will not be implementing, the main system calls come from the clone function, and a waitpid call. These appear to be key in spawning a child process which then processes the command entered by the user. The waitpid then signals the parent process to wait until execution of the child ends. There also are the execve command calls for running the actual shell, and several open/close calls for file descriptors and as well as read/write calls for outputs and user input commands.

Once you have the my_system() implementation (using fork()), run the tiny shell version with your my_system again and see whether it behaves the same way as the implementation that used the system() from Linux. If there are differences, you need to explain those differences and why they are so.



> The system calls appear to be largely the same, many of the signal commands that are called in the standard system() call do not appear in the simplified fork() implementation of the call.

While designing the my_system() using clone() you don't need to completely emulate fork() or vfork(). You can set the flags for maximum performance. What would that be? However, it is still necessary to obtain isolation like parent proceeding while child crashing.

> CLONE_VM, share memory, and CLONE_FILES, to copy files
> These are all that are key parts to the clone flags that will ensure an accurate emulation of fork() or vfork().

How would you set the flags if you want to execute a command like cd in the child and have the parent be affected by it? If you want subsequent commands executed by the parent to run in a specific directory that was changed by a previous cd command, what should the cloning do? That is, what kind of sharing should be performed while cloning?

> To execute a command cd you have to add an extra flag CLONE_FS to copy the file system over to the child process, this way any changes to the file system in the execution

process will carry over back to the parent process and the main program. This will also allow subsequent commands to stay in the changed directory

Code Specific Details

For flags in the clone, I used:

```
CLONE_VM| CLONE_FILES| CLONE_FS| SIGCHLD| CLONE_VFORK
```

This ensured that memory, files, and file systems were shared, as well as a sigchld command which would help track the state of the cloned process. Finally the vfork flag ensured that no background process was running so that the child process could execute properly.

For the wait process for each implementation was identical:

```
if(waitpid(-1, &status, 0) != 0) //wait for child process to finish
{
    status = -1;
}
return status;
```

The parent process checked the status of the child process until the process terminated, if the termination returned a non-zero status then the status was set and the my_system call returned to main. This was how I tracked for errors and checked the functionality of the child processes.

The pipe implementation was similarly fairly straightforward, I used two global variables, one Boolean ‘direct’ and one string pointer. The string pointer is used to move the fifo pipe name description to the my_system pipe call. The pipe was then split into the stdin and stdout of two instances of tshell. Determined by their Boolean instruction. And the piping occurred normally.

Commands seemed to run fairly cohesively, one thing that I did notice, was that while I still had a command prompt line in the tinyshell code. When redirecting from a file, the prompt would actually print out after the commands had run, which seemed to me that the next line of code was being read before the printf could process.

Implementation Timing

Timing for different implementations of this shell was done using a simple shell script, attached with this submission (if the grader wishes to see it). Each program and command ran 100 times, results were then parsed and averaged in an excel file (also attached). The final results can be seen below:

Implementation	Call	Time	Call	Time	Call	Time	Call	Time	Call	Time
System()	Ls -a	0.000379	pwd	0.000386	date	0.000397	who	0.000403	Ps	0.000391
Fork()		0.000357		0.000426		0.000445		0.000426		0.000389
Vfork()		0.000337		0.000402		0.000428		0.000399		0.000402
Clone()		0.000335		0.000398		0.000429		0.000413		0.000410

As you can see, the implementation speeds appeared to be fairly consistent, with all of the average runtimes being within a margin of error of each other. This indicates that the differences within the implementations, especially for small commands such as these, the differences in memory is not apparent, and will not be obvious. The differences may begin to be much more apparent with larger programs and scripts.