Joshua Dunne
Assignment 7

Abstract

Today we're undertaking an interesting task. We're going to develop a shell and demonstrate it's usage and functionality. We're going to run an strace on top of it and examine the system calls that our shell uses to implement its functionality.

Introduction

As I've already done this before, and the shell I developed in the past has functionality far surpassing what's expected of this assignment, I'm going to be using the code that I've written before. We need to make a few changes to stay constant with the expectations of this class, and those have been undertaken.

Summary of Results

Code examination

Let's first demonstrate an understanding of what's being done here. A shell is in fact a multiprocessing environment. In the simplest case, we continue to iterate while we're lead to expect further input, and fork processes. Those processes are then overtaken by exec. They carry out their process to completion while the results continue to be printed into our shell. On completion, control is returned to the shell so that further commands might be run.In examining our code, that is exactly what we see happening.

```c
while (1){              /* Program terminates normally inside setup */
    int status;
    int pid_finished;
    if (!background) {
            int pid_finished = doWait(&status);
            background = 0;
            doHandleChildProcessEnded(pid_finished, &status);
    }
    sleep(1);
    background = 0;
    doPrompt();
    fflush(stdout);
    setup(inputBuffer,args,&background);        /* get next command */

    if (inputBuffer[0] != '\0') {
            if (isBuiltIn(inputBuffer, args) == -1) {
                    if (isCommand(inputBuffer, args, background) == -1) {
                            printf("failed to find %s\n", inputBuffer);
                    }
            }
    }
}
```

Here we iterate, while given reason to, we continue to take input from the user.

```c
int doForegroundCommand(char* cmd, char** args) {
        pid_t child_pid;
        int child_status;

        static int count = 0;
        child_pid = fork();
        if (child_pid == 0) {
                /* continue as child process */
                execvp(cmd, args);

                /* continues only if execvp fails */
                printf("failed to invoke %s\n", cmd);
                return -1;
        } else {
                PIDsOfBackgroundProcesses[CommandCount - 1] = child_pid;
                printf("[Child pid = %d, background = %s]\n", child_pid, "FALSE");
                /* as it is a foreground command, we'll wait
                 * here until it finished */
                waitpid(child_pid, &child_status, WUNTRACED | WCONTINUED);
                setBackgroundProcessAs(child_pid, child_status);
                doHandleChildProcessEnded(child_pid, &child_status);
        }
        return 0;
}
```

This input is then used to find the command we'd wish to run, and supply that command with proper parameters as specified by the user .As discussed in class, and documented in the code, at the call fork(), we branch into two processes, the child_pid will be set to 0 in the forked process. Otherwise, the child_pid will not be zero, in which case, we wait on the child process to exit with the waitpid() system call. Other tasks are undertaken, but they are less important to the understanding of what is happening here. Let's instead run the commands requested to demonstrate correctness.

Running expectations

```
jdshell[2]: ls -la /home/jdizzle/Desktop
[Child pid = 3160, background = FALSE]
total 12
drwxr-xr-x   3 jdizzle jdizzle 4096 Nov 25 00:36 .
drwxr-x---  22 jdizzle jdizzle 4096 Nov 25 00:34 ..
drwxrwx---+  2 jdizzle jdizzle 4096 Oct  9 13:50 CS497
3160 exited, status 0
jdshell[3]: cat /home/jdizzle/Desktop/test.txt
[Child pid = 3161, background = FALSE]
cat: /home/jdizzle/Desktop/test.txt: No such file or directory
3161 exited, status 1
jdshell[4]: ps
[Child pid = 3162, background = FALSE]
    PID TTY          TIME CMD
   3043 pts/0    00:00:00 bash
   3155 pts/0    00:00:00 a.out
   3162 pts/0    00:00:00 ps
3162 exited, status 0
jdshell[5]: top
[Child pid = 3163, background = FALSE]

top - 00:36:37 up 2 min,  1 user,  load average: 0.85, 0.53, 0.21
Tasks: 237 total,   1 running, 236 sleeping,   0 stopped,   0 zombie
%Cpu(s):  9.4 us,  0.0 sy,  0.0 ni, 90.6 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :   3907.1 total,   1587.4 free,   1021.4 used,   1298.3 buff/cache
MiB Swap:   3898.0 total,   3898.0 free,      0.0 used.   2598.5 avail Mem
```

```
3163 exited, status 0
jdshell[6]: nano /home/jdizzle/Desktop/test.txt
[Child pid = 3172, background = FALSE]
3172 exited, status 0
jdshell[7]: sudo reboot
[Child pid = 3184, background = FALSE]
[sudo] password for jdizzle:
sudo: no password was provided
sudo: a password is required
3184 exited, status 1
jdshell[8]: exit
jdshell exiting
```

Here we can see that, as we'd expect, our shell does indeed demonstrate the basic concepts. It iterates and expects input. It forks and executes those processes. And it returns control to the shell. Notable here are a couple things. I've stopped sudo reboot from completing so that I can keep a consistent stack trace. Too, I've truncated the output of top for the sake of neatness.

Examining stack trace

For the sake of sanity, I’m going to break down our stack trace into parts. Firstly, there is a significant section that does everything necessary for our shell to run.

```
3677  execve("./a.out", ["./a.out"], 0x7fff6b0c28d8 /* 45 vars */) = 0
3677  brk(NULL)                               = 0x55eff9e6a000
3677  arch_prctl(0x3001 /* ARCH_??? */, 0x7ffcb5b8d990) = -1 EINVAL (Invalid argument)
3677  mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f3723759000
3677  access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
3677  openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
3677  newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=58771, ...}, AT_EMPTY_PATH) = 0
3677  mmap(NULL, 58771, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f372374a000
3677  close(3)                                = 0
3677  openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
3677  read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\237\2\0\0\0\0\0"..., 832) = 832
3677  pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784
3677  pread64(3, "\4\0\0\0 \0\0\0\5\0\0\0GNU\0\2\0\0\300\4\0\0\0\3\0\0\0\0\0\0\0"..., 48, 848) = 48
3677  pread64(3, "\4\0\0\0\24\0\0\0\3\0\0\0GNU\0\244;\374\204(\337f#\315I\214\234\f\256\271\32"..., 68, 896) = 68
3677  newfstatat(3, "", {st_mode=S_IFREG|0755, st_size=2216304, ...}, AT_EMPTY_PATH) = 0
3677  pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784
3677  mmap(NULL, 2260560, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f3723400000
3677  mmap(0x7f3723428000, 1658880, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x28000) = 0x7f3723428000
3677  mmap(0x7f37235bd000, 360448, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1bd000) = 0x7f37235bd000
3677  mmap(0x7f3723615000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x214000) = 0x7f3723615000
3677  mmap(0x7f372361b000, 52816, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f372361b000
3677  close(3)                                = 0
3677  mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f3723747000
3677  arch_prctl(ARCH_SET_FS, 0x7f3723747740) = 0
3677  set_tid_address(0x7f3723747a10)   = 3677
3677  set_robust_list(0x7f3723747a20, 24) = 0
3677  rseq(0x7f37237480e0, 0x20, 0, 0x53053053) = 0
3677  mprotect(0x7f3723615000, 16384, PROT_READ) = 0
3677  mprotect(0x55eff7efd000, 4096, PROT_READ) = 0
3677  mprotect(0x7f3723793000, 8192, PROT_READ) = 0
3677  prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY}) = 0
3677  munmap(0x7f372374a000, 58771)     = 0
3677  rt_sigaction(SIGQUIT, {sa_handler=0x55eff7efafde, sa_mask=[], sa_flags=SA_RESTORER|SA_RESTART|SA_INTERRUPT|SA_RESETHAND|0xffffffff01b8db58, sa_restorer=0x7f3723442520}, NULL, 8) = 0
3677  getpid()                          = 3677
3677  newfstatat(1, "", {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0), ...}, AT_EMPTY_PATH) = 0
3677  getrandom("\x7f\x99\x2d\xbc\x94\xe5\xc1\x2b", 8, GRND_NONBLOCK) = 8
3677  brk(NULL)                         = 0x55eff9e6a000
3677  brk(0x55eff9e8b000)               = 0x55eff9e8b000
3677  write(1, "Welcome to kbshell. My pid is 36"..., 35) = 35
3677  getpid()                          = 3677
3677  waitid(P_PID, 3677, 0x7ffcb5b8d740, WNOHANG, NULL) = -1 EINVAL (Invalid argument)
3677  clock_nanosleep(CLOCK_REALTIME, 0, {tv_sec=1, tv_nsec=0}, 0x7ffcb5b8d7a0) = 0
```

Notable here are a few things. We execute our compiled shell “a.out” with execve. We open shared libraries “.so”, copy their contents, create necessary memory mappings, and protect portions of that memory as read only. Once the overhead is taken care of, we getpid() to tell the user the shell's pid, and wait for the user to give input.

Our first command was

```
read(0, "ls -la /home/jdizzle/Desktop\n", 80) = 29
```

```
3677  write(1, "jdshell[2]: ", 12)       = 12
3677  read(0, "ls -la /home/jdizzle/Desktop\n", 80) = 29
3677  clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD, child_tidptr=0x7f3723747a10) = 3684
3684  set_robust_list(0x7f3723747a20, 24 <unfinished ...>
3677  write(1, "[Child pid = 3684, background = "..., 39 <unfinished ...>
3684  <... set_robust_list resumed>)     = 0
3677  <... write resumed>)               = 39
3684  execve("/usr/local/sbin/ls", ["ls", "-la", "/home/jdizzle/Desktop"], 0x7ffcb5b8db68 /* 45 vars */ <unfinished ...>
3677  wait4(3684,  <unfinished ...>
3684  <... execve resumed>)              = -1 ENOENT (No such file or directory)
3684  execve("/usr/local/bin/ls", ["ls", "-la", "/home/jdizzle/Desktop"], 0x7ffcb5b8db68 /* 45 vars */) = -1 ENOENT (No such file or directory)
3684  execve("/usr/sbin/ls", ["ls", "-la", "/home/jdizzle/Desktop"], 0x7ffcb5b8db68 /* 45 vars */) = -1 ENOENT (No such file or directory)
3684  execve("/usr/bin/ls", ["ls", "-la", "/home/jdizzle/Desktop"], 0x7ffcb5b8db68 /* 45 vars */) = 0
```

And we can see that in the stack trace. Firstly, our process is cloned, this needs to be so because we’re still relying on bash. We can also see here as execve tries first /usr/local/bin/ls, then /usr/sbin/ls, and lastly, /usr/bin/ls, where it finds ls at last. We then go through another spat of loading libraries and setup. As we ran strace with -f, we get everything, including the trace of those processes our shell forks. In looking through those we find things of interest, such as

```
3684  statx(AT_FDCWD, "/home/jdizzle/Desktop/CS497",
```

and

```
3684  newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=2852, ...}, AT_EMPTY_PATH) = 0
3684  newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=2852, ...}, AT_EMPTY_PATH) = 0
```

Both are variations of stat, and both return results describing the files given as parameters.

Secondly we executed

```
read(0, "touch /home/jdizzle/Desktop/test"..., 80) = 37
```

As this is a fairly simple idea, the necessary system calls here are minimal, while there is significant overhead

```
openat(AT_FDCWD, "/home/jdizzle/Desktop/test.txt", O_WRONLY|O_CREAT|O_NOCTTY|O_NONBLOCK, 0666) = 3
dup2(3, 0)                              = 0
close(3)                                = 0
utimensat(0, NULL, NULL, 0)             = 0
close(0)                                = 0
close(1)                                = 0
close(2)                                = 0
```

At the end of the day, all touch is doing is opening a file and closing the file handle so that the file is created or the timestamps updated.

Next up was

```
read(0, "cat /home/jdizzle/Desktop/test.t"..., 80) = 35
```

I'm guessing the first call here was to allocate any memory necessary to contain the contents of the file, and the subsequent system call was to read from the file.

```
mmap(NULL, 139264, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fe6116c600
read(3, "", 131072)                     = 0
```

However, since the file was yet empty, we do not have a subsequent system call to write the nonexistent contents.

Ps came next, and blew up the trace.

```
read(0, "ps\n", 80)                     = 3
```

Much to my surprise, ps does infact rely upon the faux system directory /proc to do much of it's work.

```
openat(AT_FDCWD, "/proc/1/status", O_RDONLY) = 6
read(6, "Name:\tsystemd\nUmask:\t0000\nState:"..., 1024) = 1024
read(6, "00000000,00000000,00000000,00000"..., 1024) = 326
close(6)                                = 0
newfstatat(AT_FDCWD, "/proc/2", {st_mode=S_IFDIR|0555, st_size=0, ...}, 0) = 0
openat(AT_FDCWD, "/proc/2/stat", O_RDONLY) = 6
read(6, "2 (kthreadd) S 0 0 0 0 -1 212998"..., 2048) = 149
```

There are oodles of calls following this, but almost all are interrogating various subdirectories or files of /proc and examining the results, we can see it going through everything in /proc/pid and examining them.

I'm skipping over top as it accomplishes its function in a very similar way to ps. The main difference being the number of libraries it loaded to accomplish this in an interactive way.

Then nano

```
read(0, "nano /home/jdizzle/Desktop/test."..., 80) = 36
```

As I entered and wrote some text using nano, we can see towards the end of its trace that

```
openat(AT_FDCWD, "/home/jdizzle/Desktop/test.txt", O_WRONLY|O_CREAT|O_TRUNC, 0666) = 3
rt_sigaction(SIGINT, {sa_handler=SIG_IGN, sa_mask=[], sa_flags=SA_RESTORER, sa_restorer=0x7f8dc5e42520}, NULL, 8) = 0
ioctl(0, TCGETS, {B38400 -opost isig -icanon -echo ...}) = 0
ioctl(0, TCGETS, {B38400 -opost isig -icanon -echo ...}) = 0
ioctl(0, SNDCTL_TMR_START or TCSETS, {B38400 -opost -isig -icanon -echo ...}) = 0
ioctl(0, TCGETS, {B38400 -opost -isig -icanon -echo ...}) = 0
rt_sigprocmask(SIG_UNBLOCK, [WINCH], NULL, 8) = 0
fcntl(3, F_GETFL)                       = 0x8001 (flags O_WRONLY|O_LARGEFILE)
rt_sigaction(SIGTSTP, {sa_handler=SIG_IGN, sa_mask=[], sa_flags=SA_RESTORER|SA_RESTART, sa_restorer=0x7f8dc5e42520}, {sa_handler=0x559faa28afd0, sa_mask=~[KILL STOP RTMIN RT_1], sa_flags=SA_RESTORER, sa_restorer=0x7f8dc5e42520}, 8) = 0
poll([{fd=0, events=POLLIN}], 1, 0) = 0 (Timeout)
poll([{fd=0, events=POLLIN}], 1, 0) = 0 (Timeout)
write(1, "\33[67;130H", 9)        = 9
write(1, "\33[1K ", 5)            = 5
write(1, "\33(B\33[0;7m[ Writing... ]\33(B\33[m\33[K", 32) = 32
rt_sigaction(SIGTSTP, {sa_handler=0x559faa28afd0, sa_mask=~[KILL STOP RTMIN RT_1], sa_flags=SA_RESTORER, sa_restorer=0x7f8dc5e42520}, NULL, 8) = 0
newfstatat(3, "", {st_mode=S_IFREG|0664, st_size=0, ...}, AT_EMPTY_PATH) = 0
write(3, "hello people\n", 13)    = 13
fsync(3)                          = 0
close(3)                          = 0
```

It write those change to the file we had opened /home/jdizzle/Desktop/test.txt

Similarly, as I didn't want to have to differing stack trace, the output from sudo reboot isn't present here

Lastly, and thank god for that, we exit

```
read(0, "exit\n", 80)                 = 5
write(1, "jdshell exiting\n", 16) = 16
exit_group(0)                         = ?
```

As we'd expect, this results in the typical call that induces the system to clean up the process, and return to the parent process, namely bash.

Conclusion

Through this basic example and the analysis of its results, we've gained greater insight in to the basic process by which a shell works. We've created a simple example that works as its own shell, using system calls in linux through wrappers in c as an interface. Using strace, and correlating the results it produced to our inputs, we can see what was done by the processes it spawned, that is, which system calls were made to carry out the functionality necessary for commands like cat, touch, and ps.