## handout02.txt Jan 31, 24 8:58 Page 1/4 CS 202, Spring 2024 Handout 2 (Class 4) 2 The handout is meant to: --illustrate how the shell itself uses syscalls --communicate the power of the fork()/exec() separation 8 --give an example of how small, modular pieces (file descriptors, 10 11 pipes, fork(), exec()) can be combined to achieve complex behavior 12 far beyond what any single application designer could or would have specified at design time. (We will not cover pipes in lecture today.) 13 1. Pseudocode for a very simple shell 15 17 while (1) { write(1, "\$ ", 2); 18 readcommand(command, args); // parse input 19 if ((pid = fork()) == 0) // child? 20 execve(command, args, 0); 21 else if (pid > 0) // parent? 22 23 wait(0); //wait for child else 24 25 perror("failed to fork"); 26 27 2. Now add two features to this simple shell: output redirection and 28 29 backgrounding 30 By output redirection, we mean, for example: 31 32 \$ ls > list.txt 33 By backgrounding, we mean, for example: 34 \$ myprog & 35 37 while (1) { write(1, "\$ ", 2); 38 readcommand(command, args); // parse input 39 if ((pid = fork()) == 0) { // child? 41 if (output\_redirected) { 42 close(1); open (redirect\_file, O\_CREAT | O\_TRUNC | O\_WRONLY, 0666); 43 44 45 // when command runs, fd 1 will refer to the redirected file 46 execve(command, args, 0); } else if (pid > 0) { // parent? if (foreground\_process) { 48 49 wait(0); //wait for child 50 } else { perror("failed to fork"); 52 53 54 55

```
handout02.txt
                                                                             Page 2/4
Jan 31, 24 8:58
   3. Another syscall example: pipe()
57
        The pipe() syscall is used by the shell to implement pipelines, such as
           $ ls | sort | head -4
59
60
         We will see this in a moment; for now, here is an example use of
61
         pipes.
62
63
            // C fragment with simple use of pipes
64
            int fdarray[2];
65
            char buf[512];
66
67
            int n;
68
            pipe(fdarray);
70
            write(fdarray[1], "hello", 5);
71
            n = read(fdarray[0], buf, sizeof(buf));
            // buf[] now contains 'h', 'e', 'l', 'l', 'o'
72
74
   4. File descriptors are inherited across fork
75
            // C fragment showing how two processes can communicate over a pipe
76
77
78
            int fdarray[2];
            char buf[512];
79
            int n, pid;
81
82
            pipe(fdarray);
            pid = fork();
83
            if(pid > 0){
85
              write(fdarray[1], "hello", 5);
86
            } else {
              n = read(fdarray[0], buf, sizeof(buf));
88
89
```

## handout02.txt Page 3/4 Jan 31, 24 8:58 5. Putting it all together: implementing shell pipelines using fork(), exec(), and pipe(). 91 93 // Pseudocode for a Unix shell that can run processes in the 94 // background, redirect the output of commands, and implement 95 96 // two element pipelines, such as "ls | sort" 97 98 void main\_loop() { 99 100 while (1) write(1, "\$ ", 2); 101 readcommand(command, args); // parse input 102 103 if ((pid = fork()) == 0) { // child? if (pipeline\_requested) { 104 105 handle\_pipeline(left\_command, right\_command) 106 } else { if (output\_redirected) { 107 108 close(1); open (redirect\_file, O\_CREAT | O\_TRUNC | O\_WRONLY, 0666); 109 110 exec(command, args, 0); 111 112 } else if (pid > 0) { // parent? 113 114 if (foreground\_process) { wait(0); // wait for child 115 116 } else { 117 perror("failed to fork"); 118 119 120 121 122 123 void handle\_pipeline(left\_command, right\_command) { 124 int fdarray[2]; 125 126 127 if (pipe(fdarray) < 0) panic ("error"); if ((pid = fork ()) == 0) { // child (left end of pipe) 128 129 dup2 (fdarray[1], 1); $\ //\$ make fd 1 the same as fdarray[1], $\ //\$ which is the write end of the 130 131 // pipe. implies close (1). 132 133 close (fdarray[0]); close (fdarray[1]); 134 parse(command1, args1, left\_command); 135 136 exec (command1, args1, 0); 137 138 } else if (pid > 0) { // parent (right end of pipe) 139 dup2 (fdarray[0], 0); // make fd 0 the same as fdarray[0], 140 // which is the read end of the pipe. 141 142 // implies close (0). close (fdarray[0]); 143 144 close (fdarray[1]); 145 parse(command2, args2, right\_command); 146 exec (command2, args2, 0); 147 148 } else { printf ("Unable to fork\n"); 149 150 151 152

## Jan 31, 24 8:58 **handout02.txt** Page 4/4

```
6. Commentary
153
       Why is this interesting? Because pipelines and output redirection
155
156
       are accomplished by manipulating the child's environment, not by
157
       asking a program author to implement a complex set of behaviors.
       That is, the *identical code* for "ls" can result in printing to the
158
       screen ("ls -l"), writing to a file ("ls -l > output.txt"), or
159
       getting ls's output formatted by a sorting program ("ls -l | sort").
160
161
162
       This concept is powerful indeed. Consider what would be needed if it
163
       weren't for redirection: the author of 1s would have had to
       anticipate every possible output mode and would have had to build in
164
165
       an interface by which the user could specify exactly how the output
166
       is treated.
167
       What makes it work is that the author of 1s expressed their
168
169
       code in terms of a file descriptor:
           write(1, "some output", byte_count);
170
       This author does not, and cannot, know what the file descriptor will
171
       represent at runtime. Meanwhile, the shell has the opportunity, *in
172
       between fork() and exec()*, to arrange to have that file descriptor
173
174
       represent a pipe, a file to write to, the console, etc.
```