

```

1 CS 202, Spring 2020
2 Handout 2 (Class 3)
3
4 The handout is meant to:
5
6 --illustrate how the shell itself uses syscalls
7
8 --communicate the power of the fork()/exec() separation
9
10 --give an example of how small, modular pieces (file descriptors,
11 pipes, fork(), exec()) can be combined to achieve complex behavior
12 far beyond what any single application designer could or would have
13 specified at design time. (We will not cover pipes in lecture today.)
14
15 1. Pseudocode for a very simple shell
16
17     while (1) {
18         write(1, "$ ", 2);
19         readcommand(command, args); // parse input
20         if ((pid = fork()) == 0) // child?
21             execve(command, args, 0);
22         else if (pid > 0) // parent?
23             wait(0); //wait for child
24         else
25             perror("failed to fork");
26     }
27
28 2. Now add two features to this simple shell: output redirection and
29    backgrounding
30
31    By output redirection, we mean, for example:
32        $ ls > list.txt
33
34    By backgrounding, we mean, for example:
35        $ myprog &
36        $
37
38    while (1) {
39        write(1, "$ ", 2);
40        readcommand(command, args); // parse input
41        if ((pid = fork()) == 0) { // child?
42            if (output_redirected) {
43                close(1);
44                open(redirect_file, O_CREAT | O_TRUNC | O_WRONLY, 0666);
45            }
46            // when command runs, fd 1 will refer to the redirected file
47            execve(command, args, 0);
48        } else if (pid > 0) { // parent?
49            if (foreground_process) {
50                wait(0); //wait for child
51            }
52            } else {
53                perror("failed to fork");
54            }
55        }

```

```

56 3. Another syscall example: pipe()
57
58 The pipe() syscall is used by the shell to implement pipelines, such as
59 $ ls | sort | head -4
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
65 int fdarray[2];
66 char buf[512];
67 int n;
68
69 pipe(fdarray);
70 write(fdarray[1], "hello", 5);
71 n = read(fdarray[0], buf, sizeof(buf));
72 // buf[] now contains 'h', 'e', 'l', 'l', 'o'
73
74 4. File descriptors are inherited across fork
75
76 // C fragment showing how two processes can communicate over a pipe
77
78 int fdarray[2];
79 char buf[512];
80 int n, pid;
81
82 pipe(fdarray);
83 pid = fork();
84 if(pid > 0) {
85     write(fdarray[1], "hello", 5);
86 } else {
87     n = read(fdarray[0], buf, sizeof(buf));
88 }
89

```

Feb 06, 20 13:59

handout02.txt

Page 3/4

```

90 5. Putting it all together: implementing shell pipelines using
91 fork(), exec(), and pipe().
92
93
94 // Pseudocode for a Unix shell that can run processes in the
95 // background, redirect the output of commands, and implement
96 // two element pipelines, such as "ls | sort"
97
98 void main_loop() {
99
100     while (1) {
101         write(1, "$ ", 2);
102         readcommand(command, args); // parse input
103         if ((pid = fork()) == 0) { // child?
104             if (pipeline_requested) {
105                 handle_pipeline(left_command, right_command)
106             } else {
107                 if (output_redirected) {
108                     close(1);
109                     open(redirect_file, O_CREAT | O_TRUNC | O_WRONLY, 0666);
110                 }
111                 exec(command, args, 0);
112             }
113         } else if (pid > 0) { // parent?
114             if (foreground_process) {
115                 wait(0); // wait for child
116             }
117         } else {
118             perror("failed to fork");
119         }
120     }
121 }
122
123 void handle_pipeline(left_command, right_command) {
124
125     int fdarray[2];
126
127     if (pipe(fdarray) < 0) panic ("error");
128     if ((pid = fork ()) == 0) { // child (left end of pipe)
129
130         dup2 (fdarray[1], 1); // make fd 1 the same as fdarray[1],
131                             // which is the write end of the
132                             // pipe. implies close (1).
133
134         close (fdarray[0]);
135         close (fdarray[1]);
136         parse(command1, args1, left_command);
137         exec (command1, args1, 0);
138     } else if (pid > 0) { // parent (right end of pipe)
139
140         dup2 (fdarray[0], 0); // make fd 0 the same as fdarray[0],
141                             // which is the read end of the pipe.
142                             // implies close (0).
143
144         close (fdarray[0]);
145         close (fdarray[1]);
146         parse(command2, args2, right_command);
147         exec (command2, args2, 0);
148     } else {
149         printf ("Unable to fork\n");
150     }
151 }
152

```

Feb 06, 20 13:59

handout02.txt

Page 4/4

```

152
153 6. Commentary
154
155 Why is this interesting? Because pipelines and output redirection
156 are accomplished by manipulating the child's environment, not by
157 asking a program author to implement a complex set of behaviors.
158 That is, the *identical code* for "ls" can result in printing to the
159 screen ("ls -l"), writing to a file ("ls -l > output.txt"), or
160 getting ls's output formatted by a sorting program ("ls -l | sort").
161
162 This concept is powerful indeed. Consider what would be needed if it
163 weren't for redirection: the author of ls would have had to
164 anticipate every possible output mode and would have had to build in
165 an interface by which the user could specify exactly how the output
166 is treated.
167
168 What makes it work is that the author of ls expressed their
169 code in terms of a file descriptor:
170     write(1, "some output", byte_count);
171 This author does not, and cannot, know what the file descriptor will
172 represent at runtime. Meanwhile, the shell has the opportunity, *in
173 between fork() and exec()*, to arrange to have that file descriptor
174 represent a pipe, a file to write to, the console, etc.

```

Feb 04, 20 23:18

our\_head.c

Page 1/1

```

1  /*
2  * our_head.c -- a C program that prints the first L lines of its input,
3  *   where L defaults to 10 but can be specified by the caller of the
4  *   program.
5  *
6  *   (This program is inefficient and does not check its error
7  *   conditions. It is meant to illustrate filters aka pipelines.)
8  */
9  #include <stdlib.h>
10 #include <unistd.h>
11 #include <stdio.h>
12
13 int main(int argc, char** argv)
14 {
15     int i = 0;
16     int nlines;
17     char ch;
18     int ret;
19
20     if (argc == 2) {
21         nlines = atoi(argv[1]);
22     } else if (argc == 1) {
23         nlines = 10;
24     } else {
25         fprintf(stderr, "usage: our_head [nlines]\n");
26         exit(1);
27     }
28
29     for (i = 0; i < nlines; i++) {
30
31         do {
32
33             /* read in the first character from fd 0 */
34             ret = read(0, &ch, 1);
35
36             /* if there are no more characters to read, then exit */
37             if (ret == 0) exit(0);
38
39             write(1, &ch, 1);
40
41         } while (ch != '\n');
42
43     }
44
45     exit(0);
46 }

```

Feb 04, 20 12:28

our\_yes.c

Page 1/1

```

1  /*
2  * our_yes.c -- a C program that prints its argument to the screen on a
3  *   new line every second.
4  *
5  */
6  #include <stdlib.h>
7  #include <string.h>
8  #include <unistd.h>
9  #include <stdio.h>
10
11 int main(int argc, char** argv)
12 {
13     char* repeated;
14     int len;
15
16     /* check to make sure the user gave us one argument */
17     if (argc != 2) {
18         fprintf(stderr, "usage: our_yes string_to_repeat\n");
19         exit(1);
20     }
21
22     repeated = argv[1];
23
24     len = strlen(repeated);
25
26     /* loop forever */
27     while (1) {
28
29         write(1, repeated, len);
30
31         write(1, "\n", 1);
32
33         sleep(1);
34     }
35
36 }

```