

Sep 08, 15 12:26

handout01.txt

Page 1/5

```

1 CS 202, Spring 2015
2 Handout 1 (Class 2)
3
4 1. gcc's calling convention
5
6 Example: here is some C code:
7     int main(void) {
8         return f(8)+1;
9     }
10    int f(int x) {
11        return g(x);
12    }
13    int g(int x) {
14        return x+3;
15    }
16
17 Here is the corresponding assembly code:
18
19     _main:
20         prologue
21         pushl %ebp
22         movl %esp, %ebp
23
24         body
25         pushl $8
26         call _f
27         addl $1, %eax
28
29         epilogue
30         movl %ebp, %esp
31         popl %ebp
32         ret
33
34     _f:
35         prologue
36         pushl %ebp
37         movl %esp, %ebp
38
39         body
40         pushl 8(%esp)
41         call _g
42
43         epilogue
44         movl %ebp, %esp
45         popl %ebp
46         ret
47
48 <small version of _g>:
49         movl 4(%esp), %eax
50         addl $3, %eax
51         ret
52
53 <longer version of _g>:
54         prologue
55         pushl %ebp
56         movl %esp, %ebp
57
58         save %ebx
59         pushl %ebx
60
61         body
62         movl 8(%ebp), %ebx
63         addl $3, %ebx
64         movl %ebx, %eax
65
66         restore %ebx
67         popl %ebx
68
69         epilogue
70         movl %ebp, %esp
71         popl %ebp
72         ret
73

```

Sep 08, 15 12:26

handout01.txt

Page 2/5

```

74 The rest of this handout is meant to:
75
76     --communicate the power of the fork()/exec() separation
77
78     --illustrate how the shell itself uses syscalls
79
80     --give an example of how small, modular pieces (file descriptors,
81     pipes, fork(), exec()) can be combined to achieve complex behavior
82     far beyond what any single application designer could or would have
83     specified at design time. (We will not cover pipes in lecture today.)
84
85 1. Pseudocode for a very simple shell
86
87     while (1) {
88         write(1, "$ ", 2);
89         readcommand(command, args); // parse input
90         if ((pid = fork()) == 0) // child?
91             execve(command, args, 0);
92         else if (pid > 0) // parent?
93             wait(0); //wait for child
94         else
95             perror("failed to fork");
96     }
97
98 2. Now add two features to this simple shell: output redirection and
99     backgrounding
100
101     By output redirection, we mean, for example:
102     $ ls > list.txt
103     By backgrounding, we mean, for example:
104     $ myprog &
105     $
106
107     while (1) {
108         write(1, "$ ", 2);
109         readcommand(command, args); // parse input
110         if ((pid = fork()) == 0) { // child?
111             if (output_redirected) {
112                 close(1);
113                 open(redirect_file, O_CREAT | O_TRUNC | O_WRONLY, 0666);
114             }
115             // when command runs, fd 1 will refer to the redirected file
116             execve(command, args, 0);
117         } else if (pid > 0) { // parent?
118             if (foreground_process) {
119                 wait(0); //wait for child
120             }
121         } else {
122             perror("failed to fork");
123         }
124     }
125

```

```

126 3. Another syscall example: pipe()
127
128     The pipe() syscall is used by the shell to implement pipelines, such as
129     $ ls | sort | head -4
130     We will see this in a moment; for now, here is an example use of
131     pipes.
132
133     // C fragment with simple use of pipes
134
135     int fdarray[2];
136     char buf[512];
137     int n;
138
139     pipe(fdarray);
140     write(fdarray[1], "hello", 5);
141     n = read(fdarray[0], buf, sizeof(buf));
142     // buf[] now contains 'h', 'e', 'l', 'l', 'o'
143
144 4. File descriptors are inherited across fork
145
146     // C fragment showing how two processes can communicate over a pipe
147
148     int fdarray[2];
149     char buf[512];
150     int n, pid;
151
152     pipe(fdarray);
153     pid = fork();
154     if(pid > 0){
155         write(fdarray[1], "hello", 5);
156     } else {
157         n = read(fdarray[0], buf, sizeof(buf));
158     }
159

```

```

160 5. Putting it all together: implementing shell pipelines using
161     fork(), exec(), and pipe().
162
163
164     // Pseudocode for a Unix shell that can run processes in the
165     // background, redirect the output of commands, and implement
166     // two element pipelines, such as "ls | sort"
167
168     void main_loop() {
169
170         while (1) {
171             write(1, "$ ", 2);
172             readcommand(command, args); // parse input
173             if ((pid = fork()) == 0) { // child?
174                 if (pipeline_requested) {
175                     /* NOTE: lab2's logic is different from this */
176                     handle_pipeline(left_command, right_command)
177                 } else {
178                     if (output_redirected) {
179                         close(1);
180                         open(redirect_file, O_CREAT | O_TRUNC | O_WRONLY, 0666);
181                     }
182                     exec(command, args, 0);
183                 }
184             } else if (pid > 0) { // parent?
185                 if (foreground_process) {
186                     wait(0); // wait for child
187                 }
188             } else {
189                 perror("failed to fork");
190             }
191         }
192     }
193
194     void handle_pipeline(left_command, right_command) {
195
196         int fdarray[2];
197
198         if (pipe(fdarray) < 0) panic ("error");
199         if ((pid = fork ()) == 0) { // child (left end of pipe)
200
201             dup2 (fdarray[1], 1); // make fd 1 the same as fdarray[1],
202                                 // which is the write end of the
203                                 // pipe. implies close (1).
204
205             close (fdarray[0]);
206             close (fdarray[1]);
207             parse(command1, args1, left_command);
208             exec (command1, args1, 0);
209
210         } else if (pid > 0) { // parent (right end of pipe)
211
212             dup2 (fdarray[0], 0); // make fd 0 the same as fdarray[0],
213                                 // which is the read end of the pipe.
214                                 // implies close (0).
215
216             close (fdarray[0]);
217             close (fdarray[1]);
218             parse(command2, args2, right_command);
219             exec (command2, args2, 0);
220
221         } else {
222             printf ("Unable to fork\n");
223         }
224     }
225

```

Sep 08, 15 12:26

handout01.txt

Page 5/5

223

6. Commentary

224

225

226

227

228

229

230

231

232

233

234

235

236

237

238

239

240

241

242

243

244

245

Why is this interesting? Because pipelines and output redirection are accomplished by manipulating the child's environment, not by asking a program author to implement a complex set of behaviors. That is, the *identical code* for `ls` can result in printing to the screen (`ls -l`), writing to a file (`ls -l > output.txt`), or getting `ls`'s output formatted by a sorting program (`ls -l | sort`).

This concept is powerful indeed. Consider what would be needed if it weren't for redirection: the author of `ls` would have had to anticipate every possible output mode and would have had to build in an interface by which the user could specify exactly how the output is treated.

What makes it work is that the author of `ls` expressed his or her code in terms of a file descriptor:

```
write(1, "some output", byte_count);
```

This author does not, and cannot, know what the file descriptor will represent at runtime. Meanwhile, the shell has the opportunity, *in* between `fork()` and `exec()`, to arrange to have that file descriptor represent a pipe, a file to write to, the console, etc.

Feb 07, 15 18:30

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.)
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     exit(0);
45 }
46
```

Feb 07, 15 18:30

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         write(1, repeated, len);
29
30         write(1, "\n", 1);
31
32         sleep(1);
33     }
34 }
35
36 }
```