```
1/30/22. 9:30 AM
handout w3a
                                          CS5600, Cheng Tan
      CS5600
      Handout week03a
  4
      The handout is meant to:
        --illustrate how the shell itself uses syscalls
 8
        --communicate the power of the fork()/exec() separation
 9
  10
        --qive an example of how small, modular pieces (file descriptors,
        pipes, fork(), exec()) can be combined to achieve complex behavior
  11
  12
        far beyond what any single application designer could or would have
  13
        specified at design time.
  14
 15 1. Pseudocode for a very simple shell
 16
 17
          while (1) {
           write(1, "$ ", 2);
 18
 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
          By backgrounding, we mean, for example:
             $ myprog &
  34
  35
  36
  37
          while (1) {
  38
            write(1, "$ ", 2);
  39
            readcommand(command, args); // parse input
  40
            if ((pid = fork()) == 0) {
                                          // child?
  41
              if (output redirected) {
  42
                close(1);
  43
                open(redirect_file, 0_CREAT | 0_TRUNC | 0_WRONLY, 0666);
  44
  45
              // when command runs, fd 1 will refer to the redirected file
  46
              execve(command, args, 0);
  47
              } else if (pid > 0) {
                                          // parent?
  48
                if (foreground process) {
  49
                  wait(0);
                                          //wait for child
  50
  51
              } else {
  52
                perror("failed to fork");
  53
  54
  55
```

```
handout w3a

CS5600, Cheng Tan

1/30/22. 9:30 AM

CS5600, Cheng Tan
```

```
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));
      // buf[] now contains 'h', 'e', 'l', 'l', 'o'
72
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
```

Page 1 of 4 Page 2 of 4

CS5600, Cheng Tan

90 5. Putting it all together: implementing shell pipelines using fork(), exec(), and pipe().

// Pseudocode for a Unix shell that can run processes in the

// background, redirect the output of commands, and implement

handout w3a

92

93

94

95

141

142

143

144

145 146

147 148

149

150

close (fdarray[0]);

close (fdarray[1]);

} else {

}}

exec (command2, args2, 0);

printf ("Unable to fork\n");

parse(command2, args2, right\_command);

1/30/22. 9:30 AM

```
// two element pipelines, such as "ls | sort"
96
97
98
        void main loop() {
99
100
          while (1) {
            write(1, "$ ", 2);
101
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
109
                  open(redirect_file, 0_CREAT | 0_TRUNC | 0_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");</pre>
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
            close (fdarray[0]);
134
            close (fdarray[1]);
135
            parse(command1, args1, left_command);
136
            exec (command1, args1, 0);
137
138
          } else if (pid > 0) {
                                           // parent (right end of pipe)
139
140
            dup2 (fdarray[0], 0);
                                           // make fd 0 the same as fdarray[0],
```

// which is the read end of the pipe.

// implies close (0).

handout w3a 1/30/22. 9:30 AM CS5600, Cheng Tan

```
152
153
    Commentary
154
155
    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.
157
    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").
161
162
     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
165
    an interface by which the user could specify exactly how the output
166
167
168 What makes it work is that the author of ls expressed their
    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
173 between fork() and exec()*, to arrange to have that file descriptor
174 represent a pipe, a file to write to, the console, etc.
```

Page 3 of 4 Page 3 of 4