```
Handout for CS 372H
Class 5
31 January 2011

The first four panels, taken together, are meant to:

    --communicate the power of the fork()/exec() separation

    --illustrate how the shell itself uses syscalls

    --give an example of how small, modular pieces (file descriptors,
    pipes, fork(), exec()) can be combined to achieve complex behavior
    far beyond what any single application designer could or would have
    specified at design time.

1. Pseudocode for a very simple shell (see minish.c at the back of the
handout for a non-pseudocode version of what's below).

        while (1) {
                write(1, "$ ", 2);
                readcommand(command, args); // parse input
                if ((pid = fork()) == 0) // child?
                        exec(command, args, 0);
                else if (pid > 0) // parent?
                        wait(0); //wait for child
                else
                        perror("failed to fork");
        }

2. Now add two features to this simple shell: output redirection and
   backgrounding (see redirsh.c at the back of the handout for a non-pseudocode
   version of the redirection handling).

    By output redirection, we mean, for example:
        $ ls > list.txt
    By backgrounding, we mean, for example:
        $ myprog &
        $

        while (1) {
            write(1, "$ ", 2);
            readcommand(command, args); // parse input
            if ((pid = fork()) == 0) { // child?
                if (output_redirected) {
                    close(1);
                    open(redirect_file, O_CREAT | O_TRUNC | O_WRONLY, 0666);
                }
                // when command runs, fd 1 will refer to the redirected file
                exec(command, args, 0);
            } else if (pid > 0) { // parent?
                if (foreground_process) {
                    wait(0);  //wait for child
                }
            } else {
                    perror("failed to fork");
            }
        }
```



```
3. Another syscall example: pipe()

    The pipe() syscall is used by the shell to implement pipelines, such as
        $ ls | sort | head -4
     We will see this in a moment; for now, here is an example use of
     pipes.

        // C fragment with simple use of pipes

        int fdarray[2];
        char buf[512];
        int n;

        pipe(fdarray);
        write(fdarray[1], "hello", 5);
        n = read(fdarray[0], buf, sizeof(buf));
        // buf[] now contains 'h', 'e', 'l', 'l', 'o'

4. File descriptors are inherited across fork

        // C fragment showing how two processes can communicate over a pipe

        int fdarray[2];
        char buf[512];
        int n, pid;

        pipe(fdarray);
        pid = fork();
        if(pid > 0){
          write(fdarray[1], "hello", 5);
        } else {
          n = read(fdarray[0], buf, sizeof(buf));
        }
```

```
5. Putting it all together: implementing shell pipelines using
   fork(), exec(), and pipe(). (See pipesh.c at the back of the
   handout for a non−pseudocode version of the pipeline handling.)


    // Pseudocode for a Unix shell that can run processes in the
    // background, redirect the output of commands, and implement
    // two element pipelines, such as "ls | sort"

     void main_loop() {

        while (1) {
            write(1, "$ ", 2);
            readcommand(command, args); // parse input
            if ((pid = fork()) == 0) { // child?
                if (pipeline_requested) {
                    handle_pipeline(left_command, right_command)
                } else {
                    if (output_redirected) {
                        close(1);
                        open(redirect_file, O_CREAT | O_TRUNC | O_WRONLY, 0666);
                    }
                    exec(command, args, 0);
                }
            } else if (pid > 0) { // parent?
                if (foreground_process) {
                    wait(0);  // wait for child
                }
            } else {
                    perror("failed to fork");
            }
        }
    }

    void handle_pipeline(left_command, right_command) {

        int fdarray[2];

        if (pipe(fdarray) < 0) panic ("error");
        if ((pid = fork ()) == 0) {  // child (left end of pipe)

           close (1);
           dup2 (fdarray[1], 1);  // make fd 1 the same as fdarray[1],
                                  // which is  the write end of the pipe
           close (fdarray[0]);
           close (fdarray[1]);
           parse(command1, args1, left_command);
           exec (command1, args1, 0);

        } else if (pid > 0) {        // parent (right end of pipe)

           close (0);
           dup2 (fdarray[0], 0);  // make fd 0 the same as fdarray[0],
                                  // which is the read end of the pipe
           close (fdarray[0]);
           close (fdarray[1]);
           parse(command2, args2, right_command);
           exec (command2, args2, 0);

        } else {
           printf ("Unable to fork\n");
        }
    }

```



```
6. Commentary

     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 02, 12 15:11 **our_head.c** Page 1/1

```c
/*
 * our_head.c -- a C program that prints the first L lines of its input,
 *   where L defaults to 10 but can be specified by the caller of the
 *   program.
 *
 *   (This program is inefficient and does not check its error
 *   conditions. It is meant to illustrate filters.)
 */
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char** argv)
{
    int i = 0;
    int nlines;
    char ch;
    int ret;

    if (argc == 2) {
        nlines = atoi(argv[1]);
    } else if (argc == 1) {
        nlines = 10;
    } else {
        fprintf(stderr, "usage: our_head [nlines]\n");
        exit(1);
    }

    for (i = 0; i < nlines; i++) {

        do {

            /* read in the first character from fd 0 */
            ret = read(0, &ch, 1);

            /* if there are no more characters to read, then exit */
            if (ret == 0) exit(0);

            write(1, &ch, 1);

        } while (ch != '\n');

    }

    exit(0);
}
```

Feb 02, 12 15:11 **our_yes.c** Page 1/1

```c
/*
 * our_yes.c -- a C program that prints its argument to the screen on a
 * new line every second.
 *
 */
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char** argv)
{
    char* repeated;
    int len;

    /* check to make sure the user gave us one argument */
    if (argc != 2) {
        fprintf(stderr, "usage: our_yes string_to_repeat\n");
        exit(1);
    }

    repeated = argv[1];

    len = strlen(repeated);

    /* loop forever */
    while (1) {

        write(1, repeated, len);

        write(1, "\n", 1);

        sleep(1);
    }

}
```

```c
/* By David Mazieres */

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/wait.h>

char **av;
int avsize;

void
avreserve (int n)
{
  int oldavsize = avsize;

  if (avsize > n + 1)
    return;

  avsize = 2 * (oldavsize + 1);
  av = realloc (av, avsize * sizeof (*av));
  while (oldavsize < avsize)
    av[oldavsize++] = NULL;
}

void
parseline (char *line)
{
  char *a;
  int n;

  for (n = 0; n < avsize; n++)
    av[n] = NULL;

  a = strtok (line, " \t\r\n");
  for (n = 0; a; n++) {
    avreserve (n);
    av[n] = a;
    a = strtok (NULL, " \t\r\n");
  }
}

void
doexec (void)
{
  execvp (av[0], av);
  perror (av[0]);
  exit (1);
}

int
main (void)
{
  char buf[512];
  char *line;
  int pid;

  avreserve (10);

  for (;;) {
    write (2, "$ ", 2);
    if (!(line = fgets (buf, sizeof (buf), stdin))) {
      write (2, "EOF\n", 4);
      exit (0);
    }
    parseline (line);
    if (!av[0])
      continue;

    switch (pid = fork ()) {
    case -1:
```



```c
      perror ("fork");
      break;
    case 0:
      doexec ();
      break;
    default:
      waitpid (pid, NULL, 0);
      break;
    }
  }
}
```

```c
/* By David Mazieres */

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/wait.h>

char **av;
char *infile;
char *outfile;
char *errfile;
int avsize;

void
avreserve (int n)
{
  int oldavsize = avsize;

  if (avsize > n + 1)
    return;

  avsize = 2 * (oldavsize + 1);
  av = realloc (av, avsize * sizeof (*av));
  while (oldavsize < avsize)
    av[oldavsize++] = NULL;
}

void
parseline (char *line)
{
  char *a;
  int n;

  infile = outfile = errfile = NULL;
  for (n = 0; n < avsize; n++)
    av[n] = NULL;

  a = strtok (line, " \t\r\n");
  for (n = 0; a; n++) {
    if (a[0] == '<')
      infile = a[1] ? a + 1 : strtok (NULL, " \t\r\n");
    else if (a[0] == '>')
      outfile = a[1] ? a + 1 : strtok (NULL, " \t\r\n");
    else if (a[0] == '2' && a[1] == '>')
      errfile = a[2] ? a + 2 : strtok (NULL, " \t\r\n");
    else {
      avreserve (n);
      av[n] = a;
    }
    a = strtok (NULL, " \t\r\n");
  }
}

void
doexec (void)
{
  int fd;

  if (infile) {
    if ((fd = open (infile, O_RDONLY)) < 0) {
      perror (infile);
      exit (1);
    }
    if (fd != 0) {
      dup2 (fd, 0);
      close (fd);
    }
  }

  if (outfile) {
```



```c
    if ((fd = open (outfile, O_WRONLY|O_CREAT|O_TRUNC, 0666)) < 0) {
      perror (outfile);
      exit (1);
    }
    if (fd != 1) {
      dup2 (fd, 1);
      close (fd);
    }
  }

  if (errfile) {
    if ((fd = open (errfile, O_WRONLY|O_CREAT|O_TRUNC, 0666)) < 0) {
      perror (outfile);
      exit (1);
    }
    if (fd != 2) {
      dup2 (fd, 2);
      close (fd);
    }
  }

  execvp (av[0], av);
  perror (av[0]);
  exit (1);
}

int
main (void)
{
  char buf[512];
  char *line;
  int pid;

  avreserve (10);

  for (;;) {
    write (2, "$ ", 2);
    if (!(line = fgets (buf, sizeof (buf), stdin))) {
      write (2, "EOF\n", 4);
      exit (0);
    }
    parseline (line);
    if (!av[0])
      continue;

    switch (pid = fork ()) {
    case -1:
      perror ("fork");
      break;
    case 0:
      doexec ();
      break;
    default:
      waitpid (pid, NULL, 0);
      break;
    }
  }
}
```

```
1   /* By David Mazieres */
2
3   #include <stdio.h>
4   #include <unistd.h>
5   #include <stdlib.h>
6   #include <string.h>
7   #include <fcntl.h>
8   #include <sys/types.h>
9   #include <sys/wait.h>
10
11  char **av;
12  char *infile;
13  char *outfile;
14  char *errfile;
15  char *outcmd;
16  int avsize;
17
18  void
19  avreserve (int n)
20  {
21    int oldavsize = avsize;
22
23    if (avsize > n + 1)
24      return;
25
26    avsize = 2 * (oldavsize + 1);
27    av = realloc (av, avsize * sizeof (*av));
28    while (oldavsize < avsize)
29      av[oldavsize++] = NULL;
30  }
31
32  void
33  parseline (char *line)
34  {
35    char *a;
36    int n;
37
38    outcmd = infile = outfile = errfile = NULL;
39    for (n = 0; n < avsize; n++)
40      av[n] = NULL;
41
42    a = strtok (line, "\t\r\n");
43    for (n = 0; a; n++) {
44      if (a[0] == '<')
45        infile = a[1] ? a + 1 : strtok (NULL, "\t\r\n");
46      else if (a[0] == '>')
47        outfile = a[1] ? a + 1 : strtok (NULL, "\t\r\n");
48      else if (a[0] == '|') {
49        if (!a[1])
50          outcmd = strtok (NULL, "");
51        else {
52          outcmd = a + 1;
53          a = strtok (NULL, "");
54          while (a > outcmd && !a[-1])
55            *--a = ' ';
56        }
57      }
58      else if (a[0] == '2' && a[1] == '>')
59        errfile = a[2] ? a + 2 : strtok (NULL, "\t\r\n");
60      else {
61        avreserve (n);
62        av[n] = a;
63      }
64      a = strtok (NULL, "\t\r\n");
65    }
66  }
67
68  void
69  doexec (void)
70  {
71    int fd;
72    int pipefds[2];
73
```



```
74    while (outcmd) {
75      if (outfile) {
76        fprintf (stderr, "syntax error: > in pipe writer\n");
77        exit (1);
78      }
79
80      if (pipe (pipefds) < 0) {
81        perror ("pipe");
82        exit (0);
83      }
84
85      switch (fork ()) {
86      case -1:
87        perror ("fork");
88        exit (1);
89      case 0:
90        if (pipefds[1] != 1) {
91          dup2 (pipefds[1], 1);
92          close (pipefds[1]);
93        }
94        close (pipefds[0]);
95        outcmd = NULL;
96        break;
97      default:
98        if (pipefds[0] != 0) {
99          dup2 (pipefds[0], 0);
100         close (pipefds[0]);
101       }
102       close (pipefds[1]);
103       parseline (outcmd);
104       if (infile) {
105         fprintf (stderr, "syntax error: < in pipe reader\n");
106         exit (1);
107       }
108       break;
109     }
110   }
111
112   if (infile) {
113     if ((fd = open (infile, O_RDONLY)) < 0) {
114       perror (infile);
115       exit (1);
116     }
117     if (fd != 0) {
118       dup2 (fd, 0);
119       close (fd);
120     }
121   }
122
123   if (outfile) {
124     if ((fd = open (outfile, O_WRONLY|O_CREAT|O_TRUNC, 0666)) < 0) {
125       perror (outfile);
126       exit (1);
127     }
128     if (fd != 1) {
129       dup2 (fd, 1);
130       close (fd);
131     }
132   }
133
134   if (errfile) {
135     if ((fd = open (errfile, O_WRONLY|O_CREAT|O_TRUNC, 0666)) < 0) {
136       perror (errfile);
137       exit (1);
138     }
139     if (fd != 2) {
140       dup2 (fd, 2);
141       close (fd);
142     }
143   }
144
145   execvp (av[0], av);
146   perror (av[0]);
```

```
147   exit (1);
148 }
149
150 int
151 main (void)
152 {
153   char buf[512];
154   char *line;
155   int pid;
156
157   avreserve (10);
158
159   for (;;) {
160     write (2, "$ ", 2);
161     if (!(line = fgets (buf, sizeof (buf), stdin))) {
162       write (2, "EOF\n", 4);
163       exit (0);
164     }
165     parseline (line);
166     if (!av[0])
167       continue;
168
169     switch (pid = fork ()) {
170     case -1:
171       perror ("fork");
172       break;
173     case 0:
174       doexec ();
175       break;
176     default:
177       waitpid (pid, NULL, 0);
178       break;
179     }
180   }
181 }
```