CS 202, Spring 2015
Handout 1 (Class 2)

1. gcc’s calling convention

Example: here is some C code:

```
    int main(void) {
        return f(8)+1;
    }
    int f(int x) {
        return g(x);
    }
    int g(int x) {
        return x+3;
    }
```

Here is the corresponding assembly code:

```
    _main:
                    prologue
        pushl %ebp
        movl %esp, %ebp

                    body
        pushl $8
        call _f
        addl $1, %eax

                    epilogue
        movl %ebp, %esp
        popl %ebp
        ret

    _f:
                    prologue
        pushl %ebp
        movl %esp, %ebp

                    body
        pushl 8(%esp)
        call _g

                    epilogue
        movl %ebp, %esp
        popl %ebp
        ret
```

```
<small version of _g>:
        movl 4(%esp), %eax
        addl $3, %eax
        ret

<longer version of _g>:
                prologue
        pushl %ebp
        movl %esp, %ebp

                save %ebx
        pushl %ebx

                body
        movl 8(%ebp), %ebx
        addl $3, %ebx
        movl %ebx, %eax

                restore %ebx
        popl %ebx

                epilogue
        movl %ebp, %esp
        popl %ebp
        ret
```

The rest of this handout is 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. (We will not cover pipes in lecture today.)

1. Pseudocode for a very simple shell

```
while (1) {
        write(1, "$ ", 2);
        readcommand(command, args); // parse input
        if ((pid = fork()) == 0) // child?
                execve(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

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
        execve(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().

```
// 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) {
                /* NOTE: lab2’s logic is different from this */
                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)

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

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

       dup2 (fdarray[0], 0);  // make fd 0 the same as fdarray[0],
                              // which is the read end of the pipe.
                              // implies close (0).
       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.

```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);
}
```

our_yes.c

```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);
    }

}
```