CS 202, Spring 2024
Handout 2 (Class 4)

The handout is meant to:

--illustrate how the shell itself uses syscalls

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

--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) {
                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 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 between fork() and exec()*, to arrange to have that file descriptor represent a pipe, a file to write to, the console, etc.