Week 4.a
CS3650
01/29 2024
https://naizhengtan.github.io/24spring/

0. process birth (cont'd)
1. Shell crash course
2. Shell internals, part I
3. File descriptors
4. Shell internals, part II

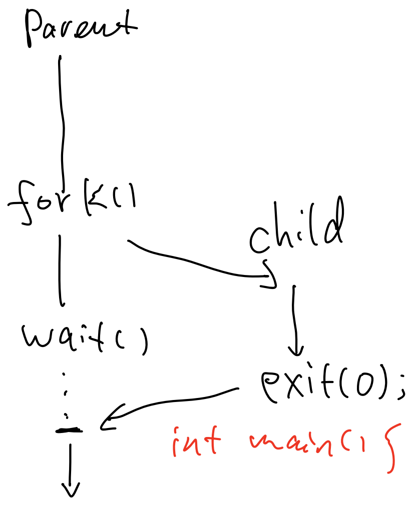
Q: $\underbrace{x == *\&x ;}$ x ⇒ int

Q: $\underset{1}{\text{exit(0) ;}}$ $\underset{1}{\text{return 0;}}$ in main

"$?" → last cmd

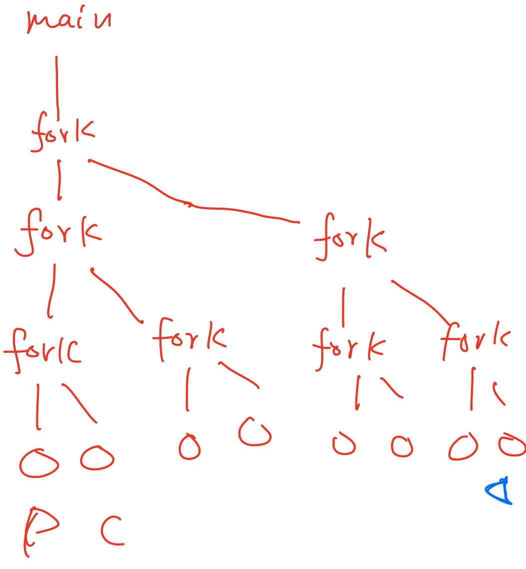
→ 0: okay ; non-0: a problem

- fork()



Q:
creat_Process(...)
why not? why fork?

int main() {
fork();
fork();
fork();
loop ←
}



- 3:
4:
8:
14:
7:

lab2

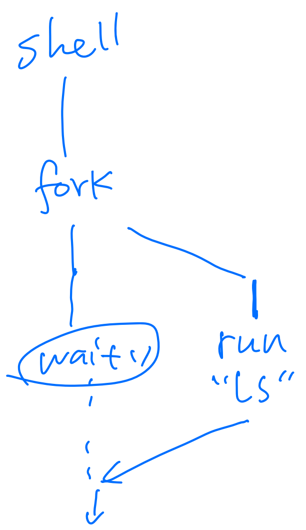
Crash course, Shell

1. run cmd

```
"$ ls" and "$ ls -a"
```

2. output redirection

```
"$ ls" prints to screen
"$ ls > files.txt"
```



3. backgrounding

```
"$ web-server &
 $  "
```

4. pipe

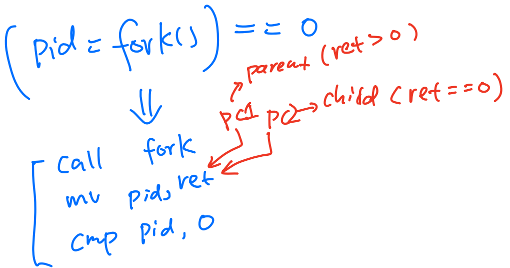
```
-- "$ cat students.txt | shuf -n 1"
-- equivalent to
   "$ cat students.txt > /tmp/tmpfile
    $ shuf -n 1</tmp/tmpfile
    $ rm /tmp/tmpfile"
```

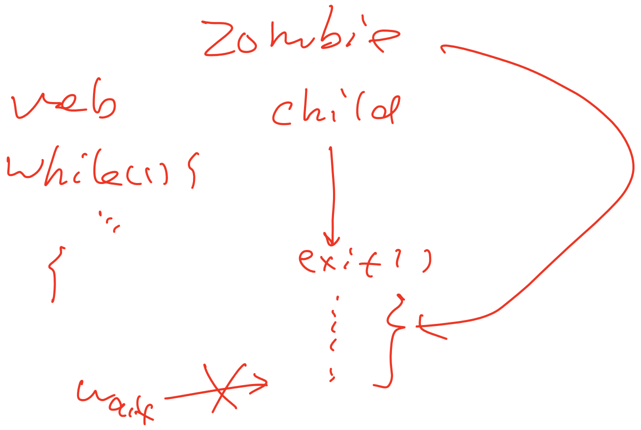
lab2

5. Shell builtin cmds vs. program

```
-- "echo/pwd/which" vs. "ls"
-- use "which" to tell
   program:   "$ which ls"
                 => "/bin/ls"
   built-in: "$ which which"
                 => "which: shell built-in command"
```

- shell internal I





Q: merge 2 files on Windows or MacOS.

$ cat file1.txt file2.txt > newfile.txt

CS3650 24spring
Handout week.04a

The handout is meant to:

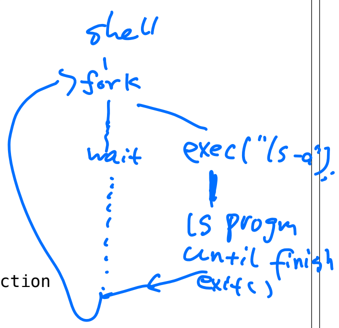
--illustrate how the shell itself uses syscalls

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

--give an example of how small, modular pieces (file descriptors, 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 // run a cmd

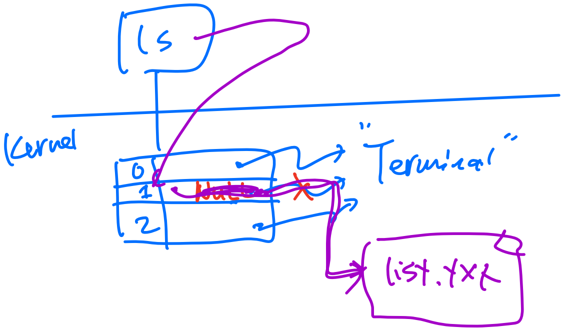
```
while (1) {
    write(1, "$ ", 2);                  // print "$"
    readcommand(command, args);  // parse input  $ ls -a
    if ((pid = fork()) == 0) {   // child? ✓
      execve(command, args, 0);
    } else if (pid > 0) {        // parent? ✓
      wait(0);  NULL             //wait for child
    } else {
      perror("failed to fork");
    }
}
```



2. Now add two features to this simple shell: output redirection

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

```
while (1) {
    write(1, "$ ", 2);            write 2 bytes of "$_" to fd 1
    readcommand(command, args);     // parse input // "ls > list.txt"
    if ((pid = fork()) == 0) {      // child?
      if (output_redirected) {
        close(1);
        open(redirect_file, O_CREAT | O_TRUNC | O_WRONLY, 0666);
      }        "list.txt"
      // when command runs, fd 1 will refer to the redirected file
      execve(command, args, 0);
    } else if (pid > 0) {           // parent?
      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. 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").

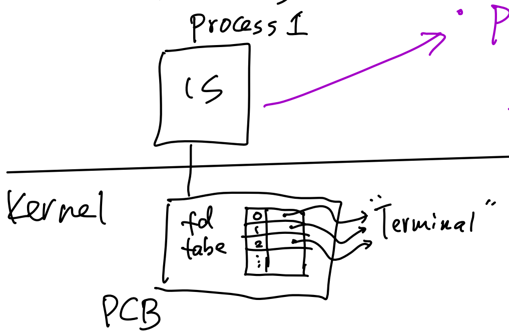
This concept is powerful indeed. Consider what would be needed if it weren't for redirection: the author of ls would have had to anticipate every possible output mode and would have had to build in an interface by which the user could specify exactly how the output is treated.

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.

- file descriptors.

```
int fd = open("/bin/ls", ....);
```

0 : std input
1 : std output
2 : std error



- find all files
- print them to std output