Week 4.b
CS3650
01/31 2024
https://naizhengtan.github.io/24spring/

1. pipelines
2. what makes a good abstraction?
3. process memory layout
4. Crash course in x86-64 assembly
5. Stack frames

----

fork / exec
wait / exit

Q: fork() ret val?

parent: pid of child
child: 0

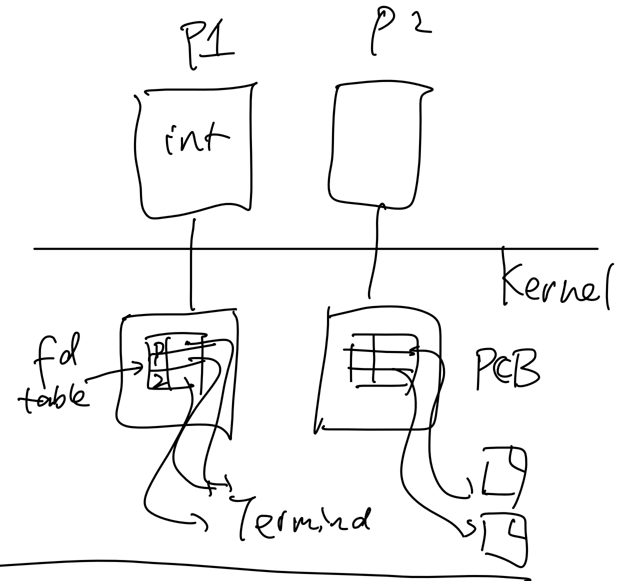
Q: fd 0/1/2?

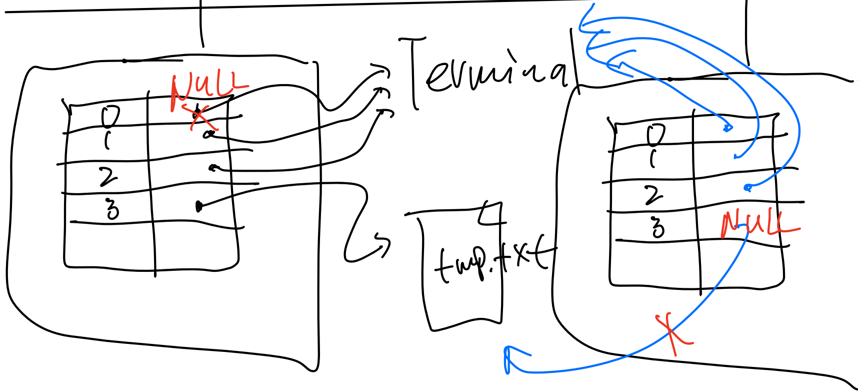
0: std in
1: stdout
2: std err

"cat"

Q: "ls > log.txt"?

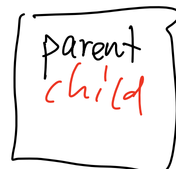
Terminal

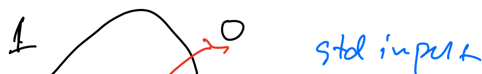




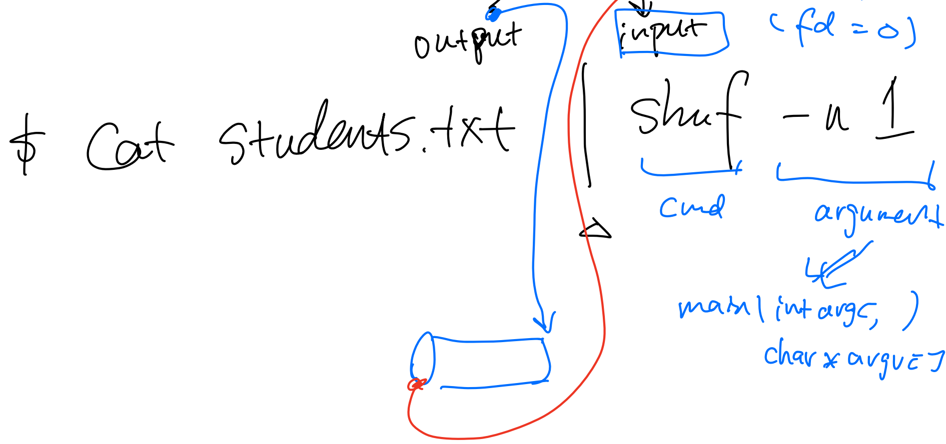
display:
[child] fd = 3
[parent] fd = 3

fib:





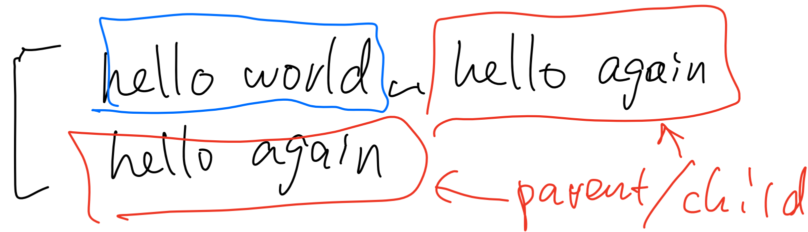
• pipe



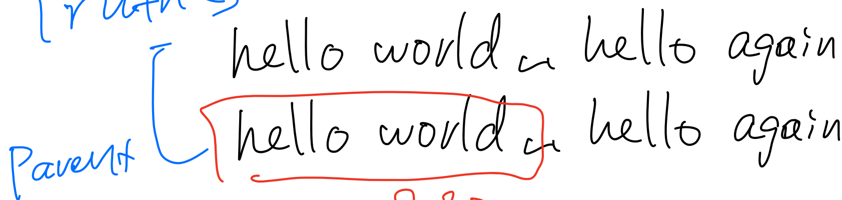
"A fork() in the road" (HotOS'19)

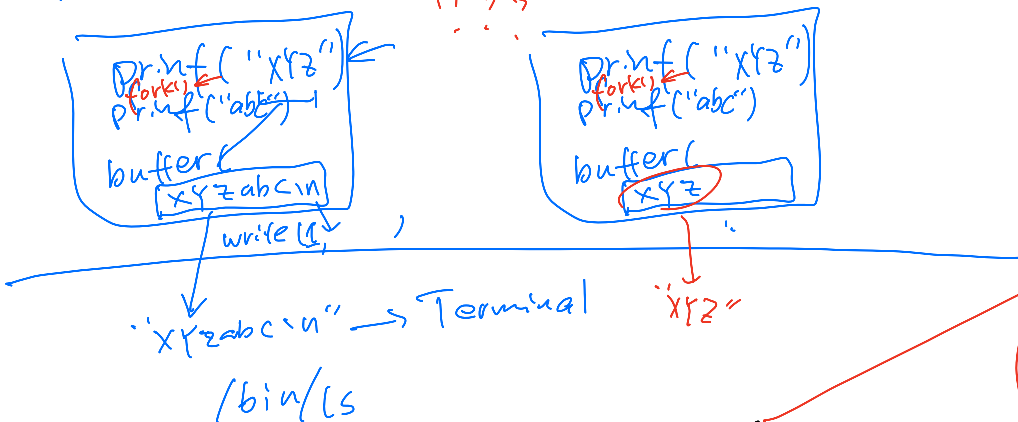
“Fork today is a convenient API for a single-threaded process with a small memory footprint and simple memory layout that requires fine-grained control over the execution environment of its children but does not need to be strongly isolated from them. In other words, a shell."

parent only

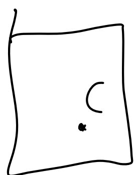


Truth :)



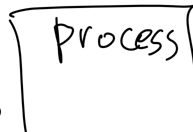


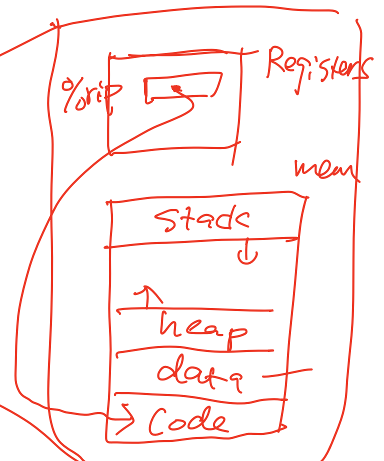
/bin/ls



compile

exec





← shell →

fork/exec/...

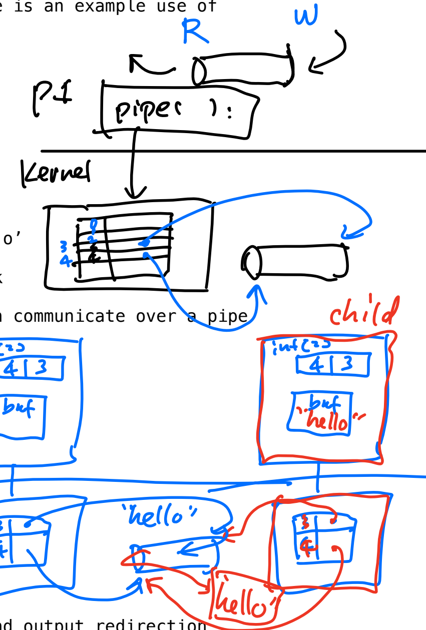
← Process mem layout →

← NEXT →

```
 1  CS3650 24spring
 2  Handout week.04a
 3
 4  The handout is meant to:
 5
 6    --illustrate how the shell itself uses syscalls
 7
 8    --communicate the power of the fork()/exec() separation
 9
10    --give an example of how small, modular pieces (file descriptors,
11    fork(), exec()) can be combined to achieve complex behavior
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) {
18        write(1, "$ ", 2);
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
29
30      By output redirection, we mean, for example:
31          $ ls > list.txt
32
33      while (1) {
34        write(1, "$ ", 2);
35        readcommand(command, args);   // parse input
36        if ((pid = fork()) == 0) {    // child?
37          if (output_redirected) {
38            close(1);
39            open(redirect_file, O_CREAT | O_TRUNC | O_WRONLY, 0666);
40          }
41          // when command runs, fd 1 will refer to the redirected file
42          execve(command, args, 0);
43          } else if (pid > 0) {       // parent?
44            wait(0);                  //wait for child
45          } else {
46            perror("failed to fork");
47          }
48      }
49
```



```
50  3. Another syscall example: pipe()
51
52    The pipe() syscall is used by the shell to implement pipelines, such as
53        $ ls | sort | head -4
54    We will see this in a moment; for now, here is an example use of
55    pipes.
56
57    // C fragment with simple use of pipes
58
59    int fdarray[2];
60    char buf[512];
61    int n;
62
63    pipe(fdarray);
64    write(fdarray[1], "hello", 5);
65    n = read(fdarray[0], buf, sizeof(buf));
66    // buf[] now contains 'h', 'e', 'l', 'l', 'o'
67
68  4. File descriptors are inherited across fork
69
70    // C fragment showing how two processes can communicate over a pipe
71
72    int fdarray[2];
73    char buf[512];
74    int n, pid;
75
76    pipe(fdarray);
77    pid = fork();
78    if(pid > 0){
79      write(fdarray[1], "hello", 5);
80    } else {
81      n = read(fdarray[0], buf, sizeof(buf));
82    }
83
84  5. Commentary
85
86  Why is this interesting? Because pipelines and output redirection
87  are accomplished by manipulating the child's environment, not by
88  asking a program author to implement a complex set of behaviors.
89  That is, the *identical code* for "ls" can result in printing to the
90  screen ("ls -l"), writing to a file ("ls -l > output.txt"), or
91  getting ls's output formatted by a sorting program ("ls -l | sort").
92
93  This concept is powerful indeed. Consider what would be needed if it
94  weren't for redirection: the author of ls would have had to
95  anticipate every possible output mode and would have had to build in
96  an interface by which the user could specify exactly how the output
97  is treated.
98
99  What makes it work is that the author of ls expressed their
100 code in terms of a file descriptor:
101 write(1, "some output", byte_count);
102 This author does not, and cannot, know what the file descriptor will
103 represent at runtime. Meanwhile, the shell has the opportunity, *in
104 between fork() and exec()*, to arrange to have that file descriptor
105 represent a pipe, a file to write to, the console, etc.
```



Crash course of x86-64 assembly

64bits

ISA CPU

- x86: Intel, AMD
- ARM: Apple
- RISC-V:

opcode — oprands (register / memory / immediate)

```
* movq PLACE1, PLACE2

* pushq %rax   [  subq $8, %rsp
                  movq %rax, (%rsp) ]

* popq %rax    [ movq (%rsp), %rax
                 addq $8, %rsp       ]

* call 0x12345   [ pushq %rip
                   movq $0x12345, %rip]

* ret            [ popq %rip ]
```

8bytes

%rax = 123