EE488: System Software Design
Fall 2022

Lecture 1: Introduction

Youjip Won

KAIST

Course Synopsis

Course Synopsis

- Instructor: Prof. Youjip Won(ywon@kaist.ac.kr, N1-309)
- Homepage:
- Class: Tuesday: 14:30 - 16:00, Thursday: 14:30 - 16:00
- Office hour
  - Tuesday: 16:00 - 17:00 @ N1-310. or online slack channel

    `https://join.slack.com/t/oslab-class/shared_invite/zt-1fa90yrq9-xfXLHepQ_FBM2K3fxGEwWA`
- two exams (midterm and final) and homeworks
- prerequisite: C/C++, Data Structures, EE415
- grading: homework(50%), midterm(25%), final(25%)

Resources

- Course Materials
  - main materials: lecture notes
  - xv6 book (https://pdos.csail.mit.edu/6.828/2018/xv6/book-rev11.pdf)
  - xv6 code (git://github.com/mit-pdos/xv6-public.git)
  - xv6 code commentary (https://pdos.csail.mit.edu/6.828/2018/xv6/xv6-rev11.pdf)
- Class homepage:[oslab.kaist.ac.kr/2022-fall-ee488]
- Office hour (online): slack channel

https://join.slack.com/t/oslab-class/shared_invite/zt-1fa90yrq9-xfXLHepQ_FBM2K3fxGEwWA

- Q&A and class announcements: piazza

piazza.com/kaist.ac.kr/fall2022/ee488

To Do

- Create an account
- Register at piazza

```
piazza.com/kaist.ac.kr/fall2022/ee488
```

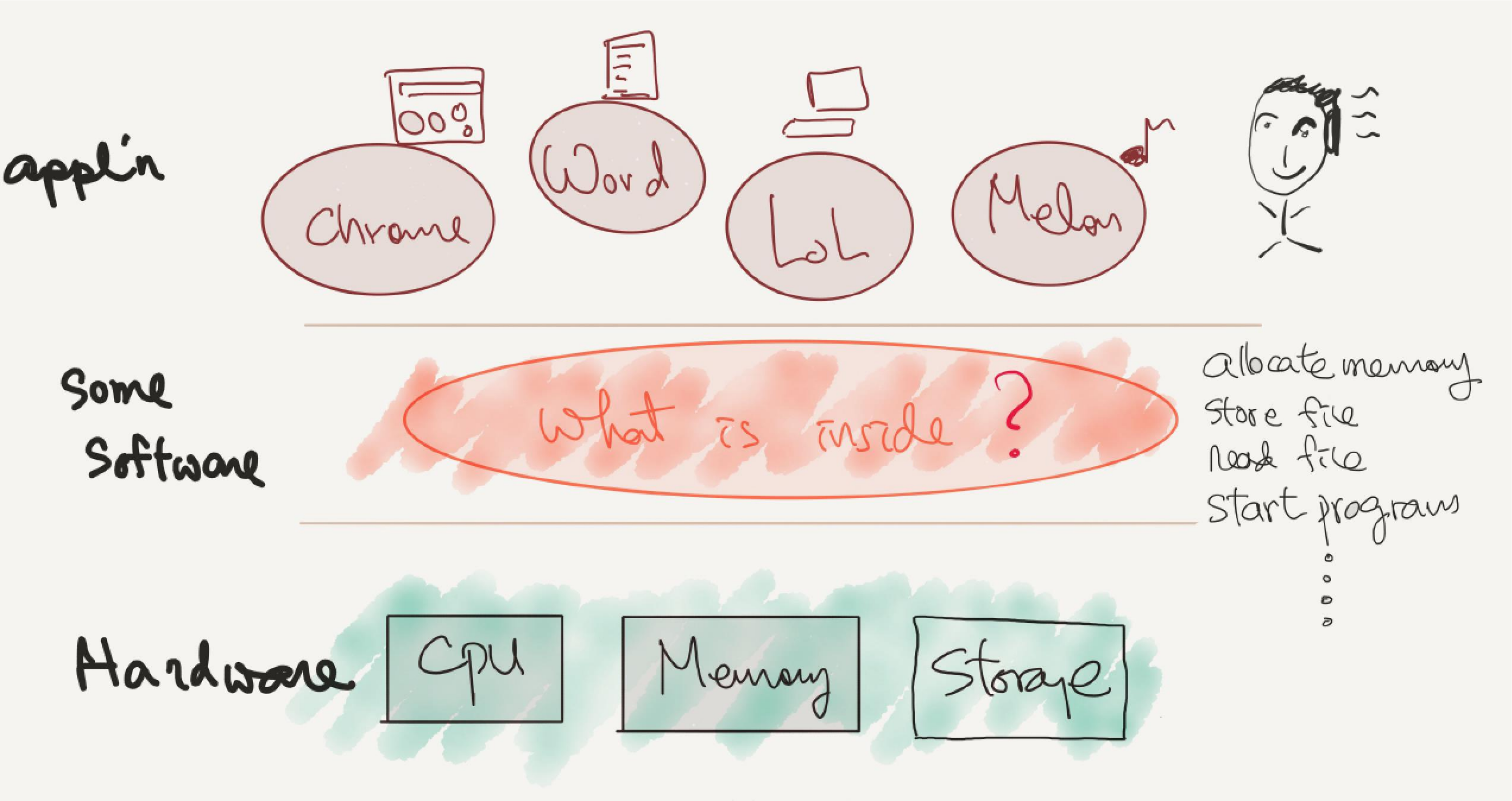
- Join slack workspace

```
https://join.slack.com/t/oslab-class/shared_invite/zt-1fa9
0yrq9-xfXLHepQ_FBM2K3fxGEwWA
```

- Find a team mate: Homeworks can be done in a group of maximum of two.

- Learn tools. (we will cover the basics of the following tools)
  - ctags, cscope, gdb, make

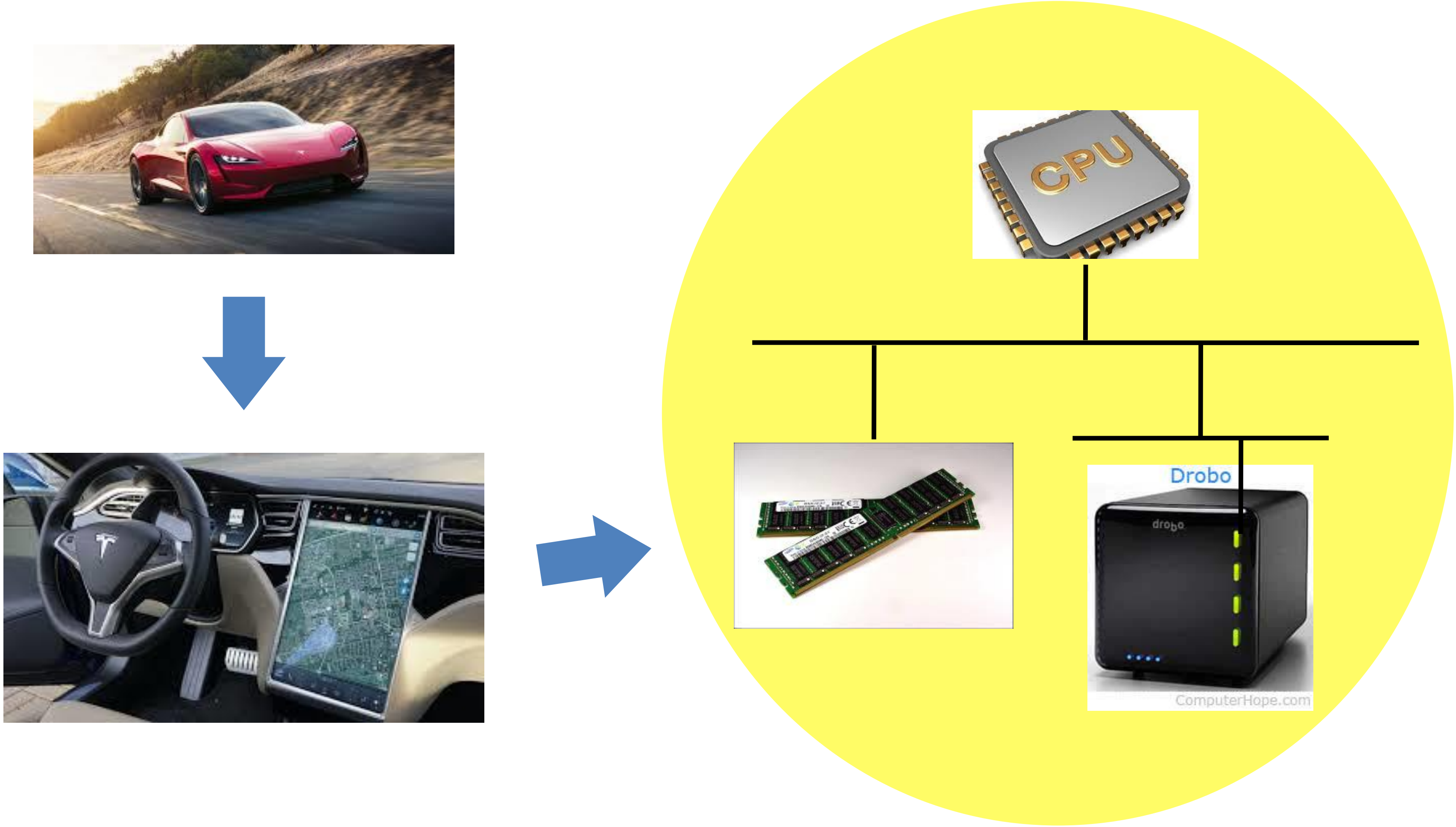
What are we going to learn?



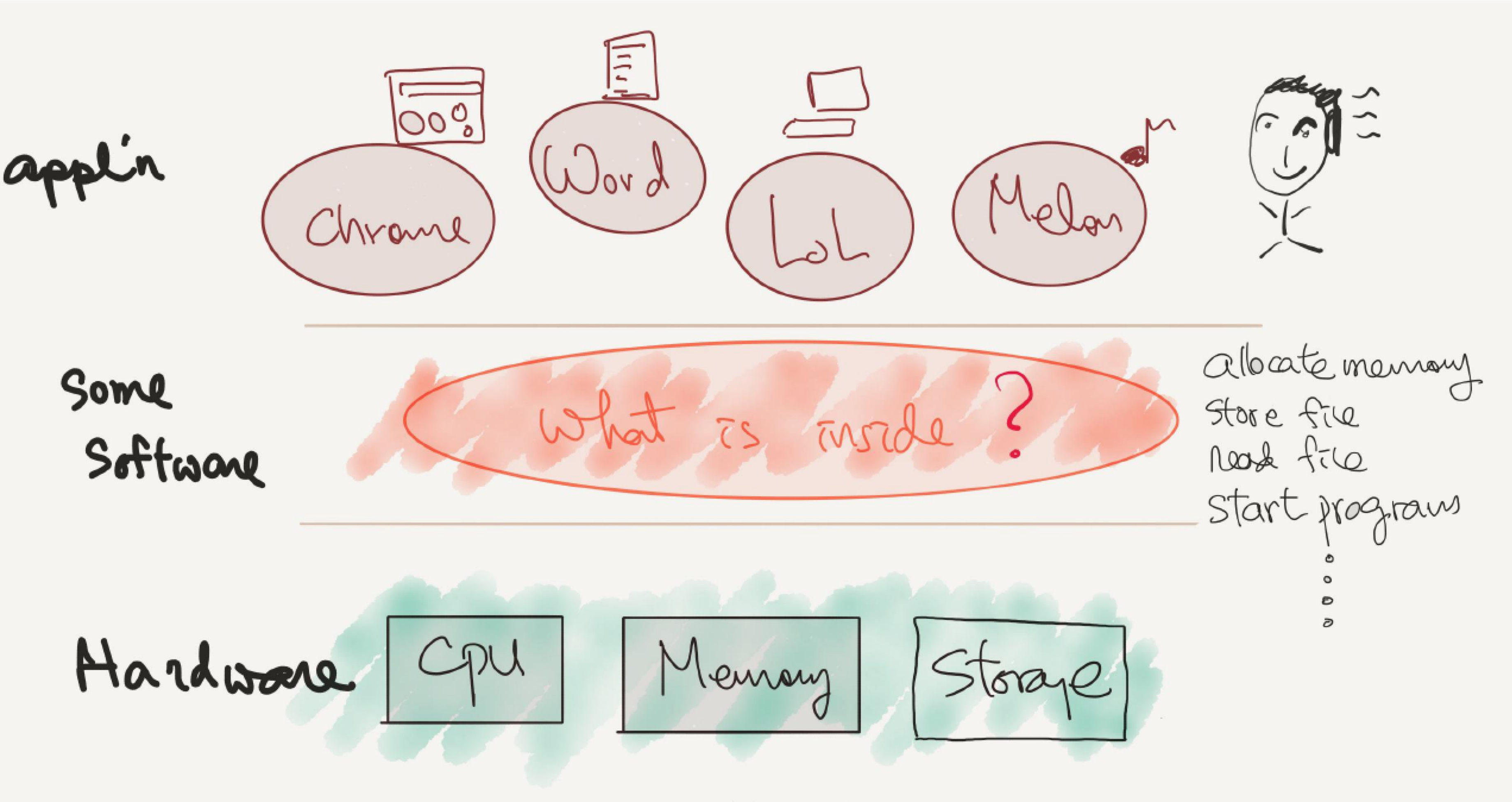
Computing Device

Applications

in essence from hardware



What are we going to learn?



Operating System

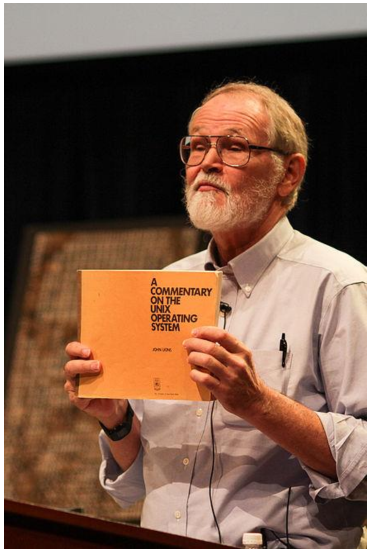
- What is Operating System?

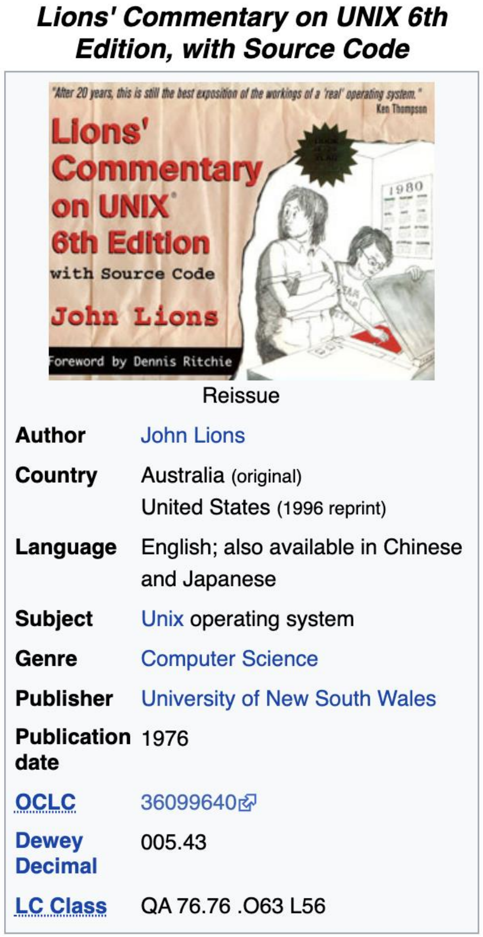
- Software that runs hardware.
- Where the hardware and software meet.

- Windows, Linux, iOS, MacOS,…
- We will look inside the OS and will learn how it works.

XV6

- xv6: x86 port of archaic SV6 OS (Unix version 6).
  - Unix Version 6 was developed for PDP11/40 in mid 70's
  - 9K lines
  - Let's Hack !!!



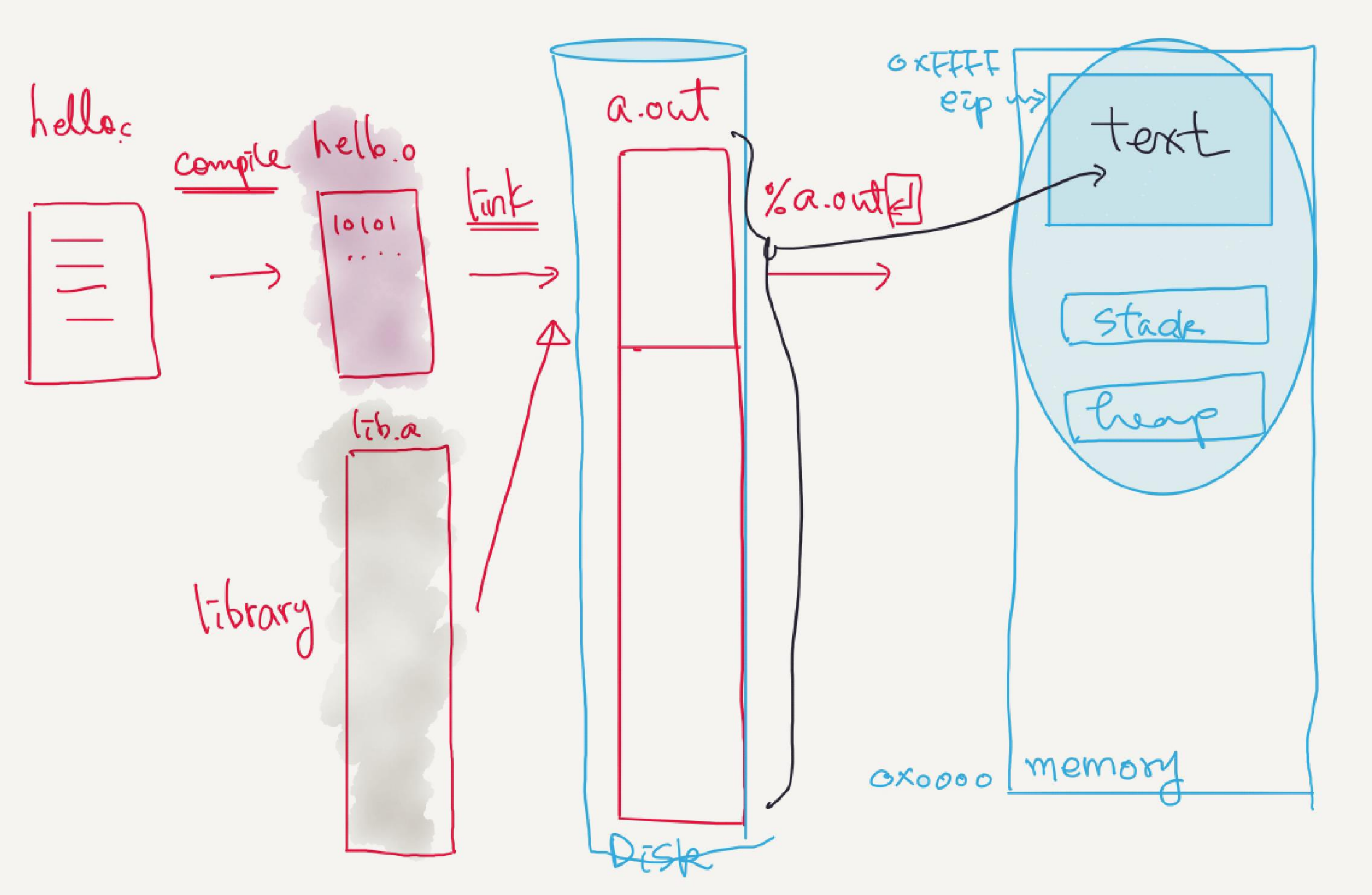

"After 20 years, this is still the best exposition of the workings of a 'real' operating system."
Ken Thompson
Lions' Commentary on UNIX 6th Edition
with Source Code
John Lions
Foreword by Dennis Ritchie
1980

Lions' Commentary on UNIX 6th Edition, with Source Code

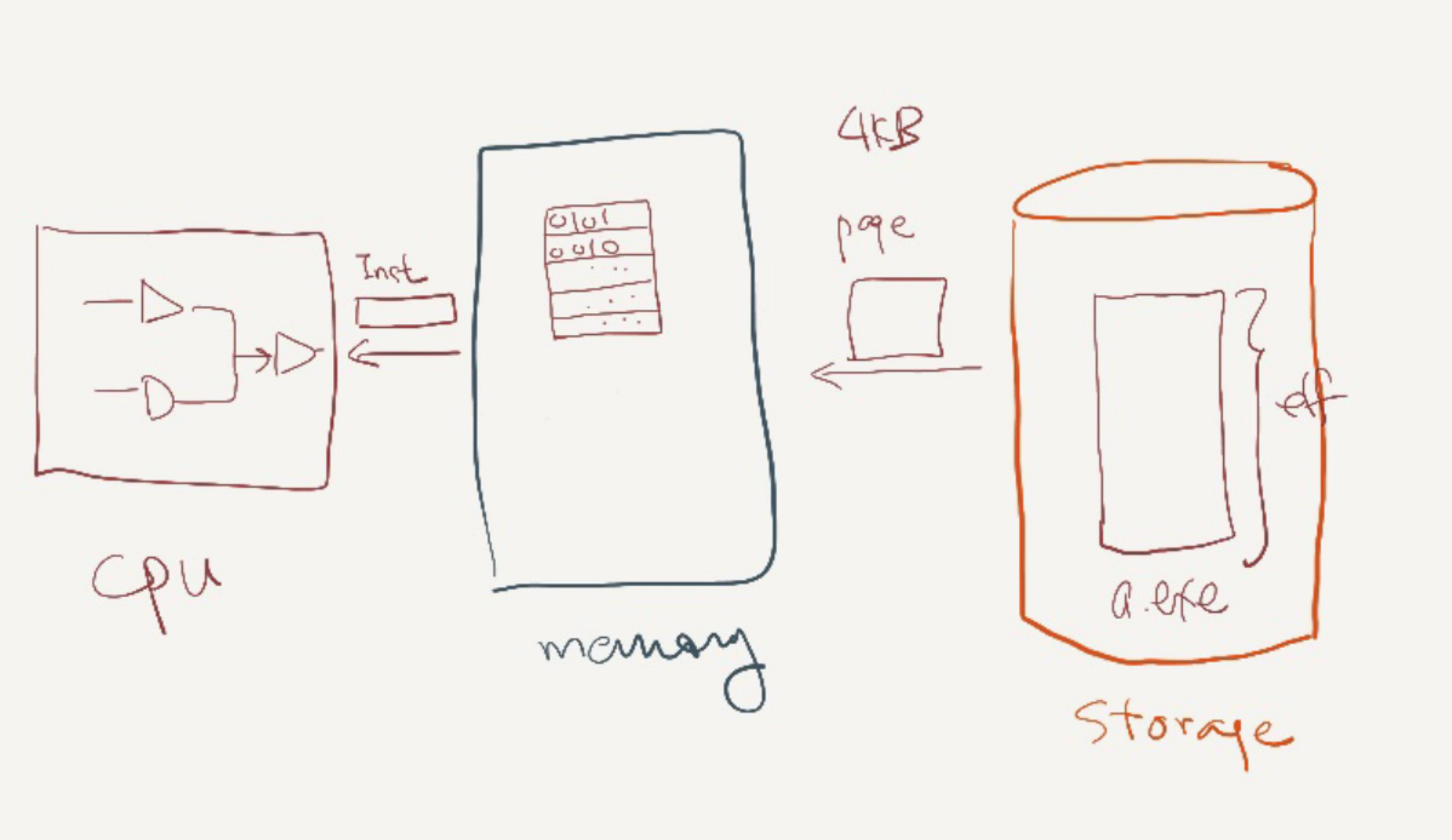
Reissue

Author	John Lions
Country	Australia (original) United States (1996 reprint)
Language	English; also available in Chinese and Japanese
Subject	Unix operating system
Genre	Computer Science
Publisher	University of New South Wales
Publication date	1976
OCLC	36099640
Dewey Decimal	005.43
LC Class	QA 76.76 .O63 L56

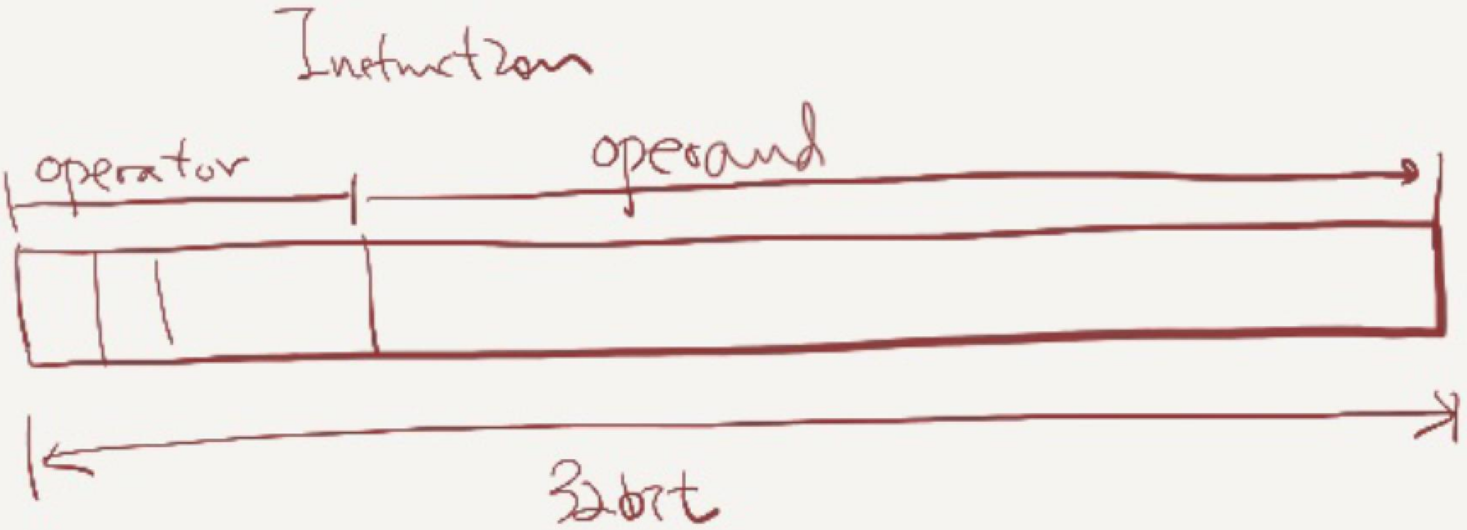
Life of a program



Execution of a program

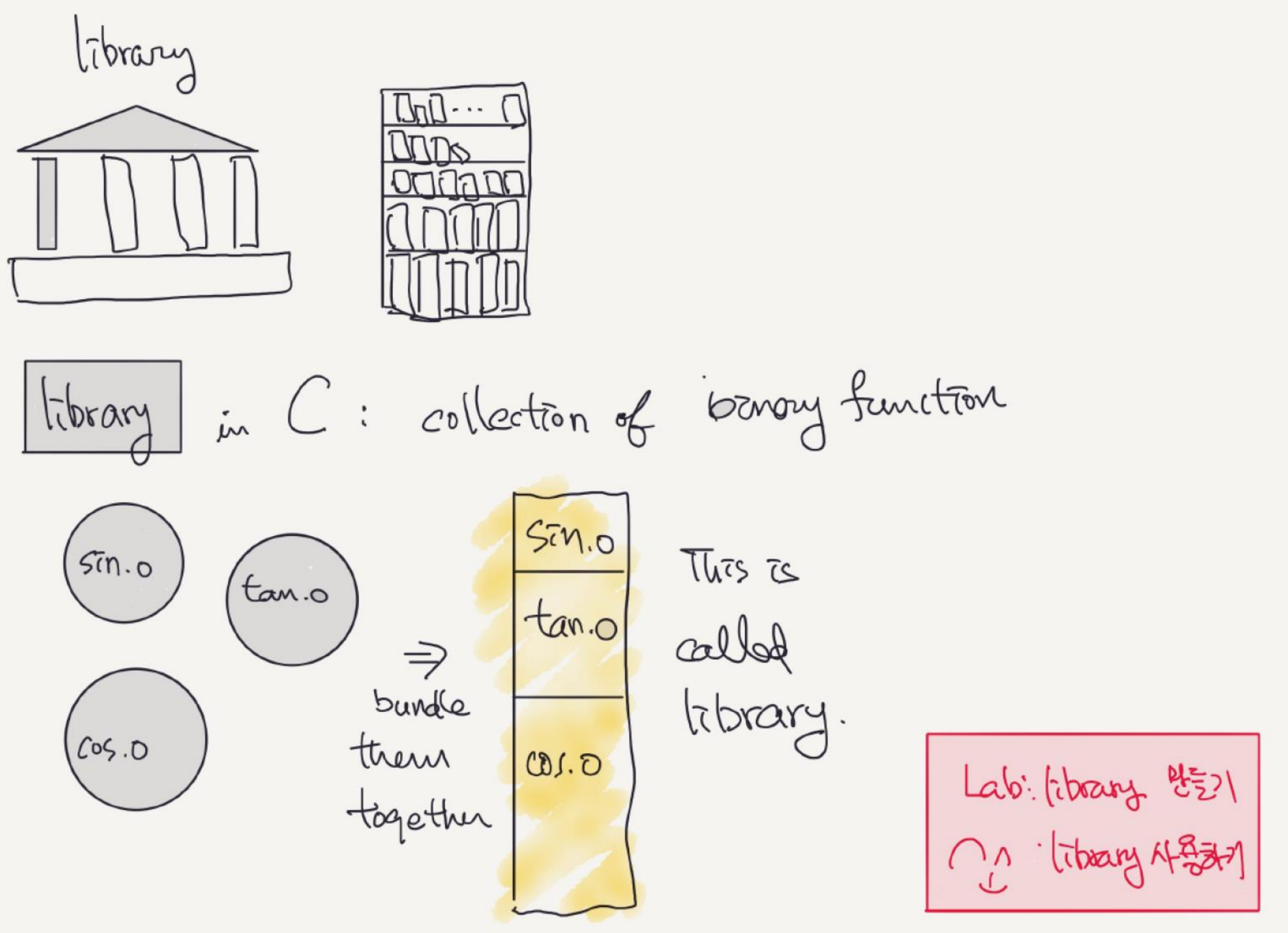


cpu



- address mode : direct, indirect
immediate

library



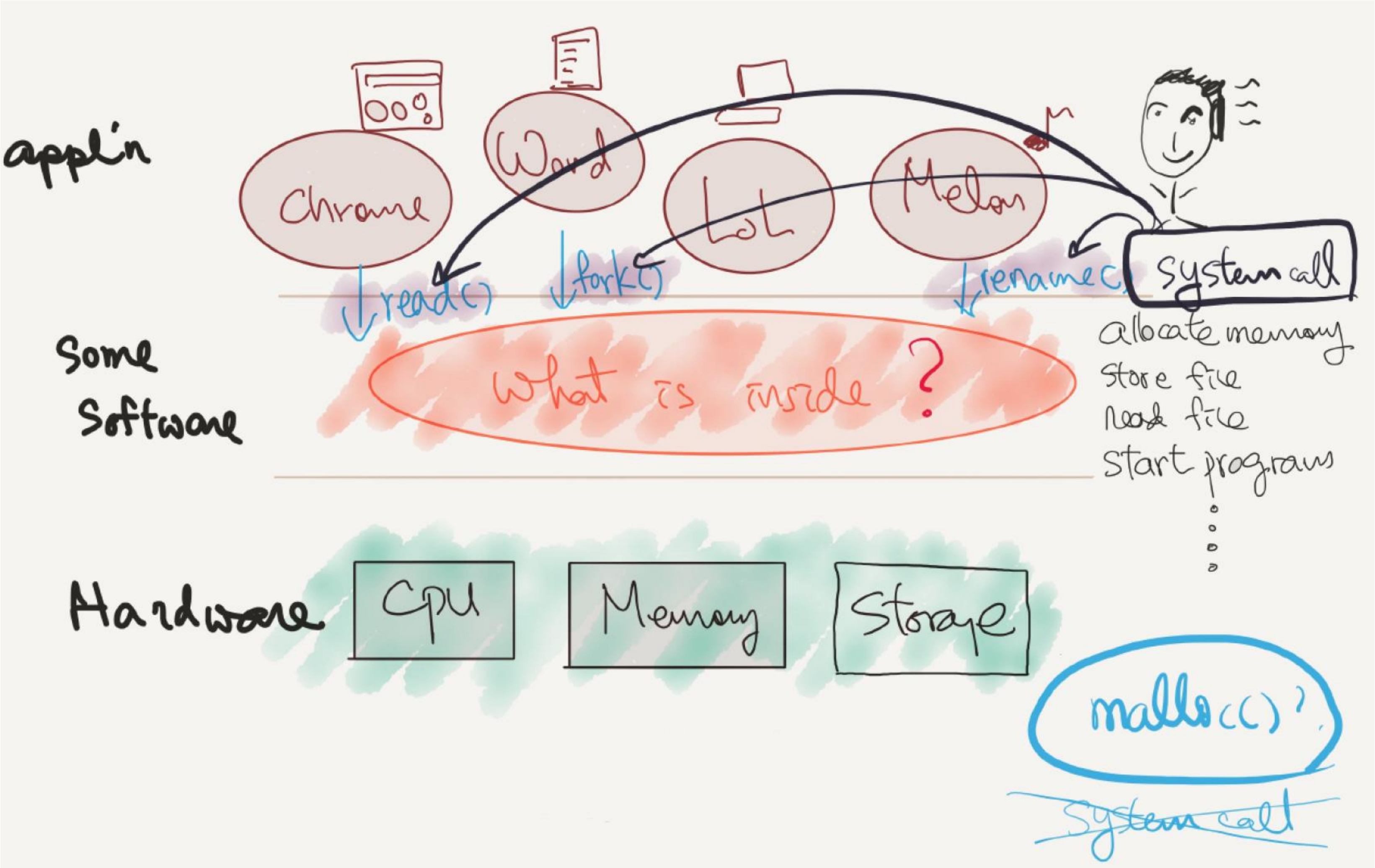
library in C : collection of binary function

This is called library.

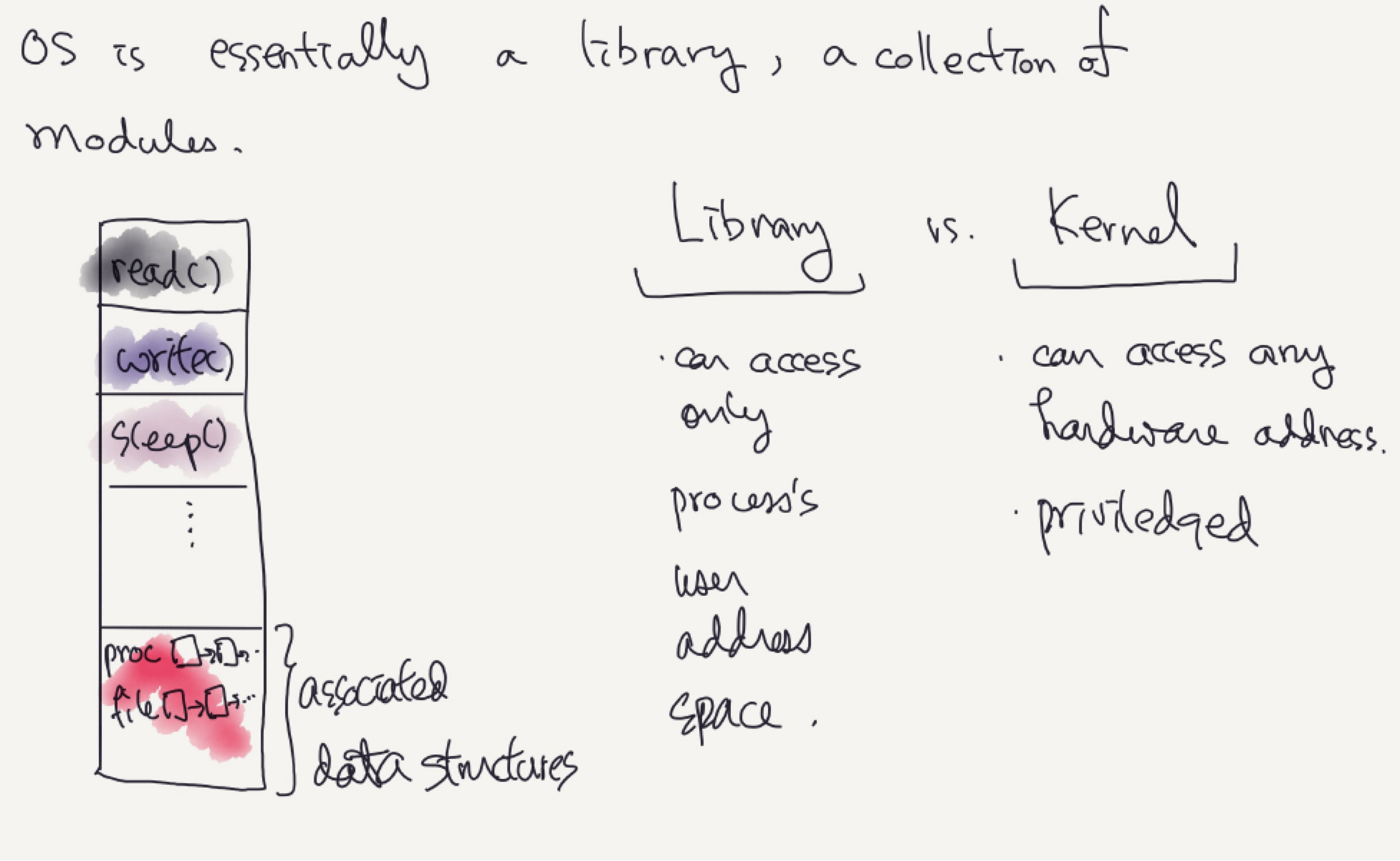
Lab: library 만들기

숙제 : library 사용하기

system calls



library call vs. system call



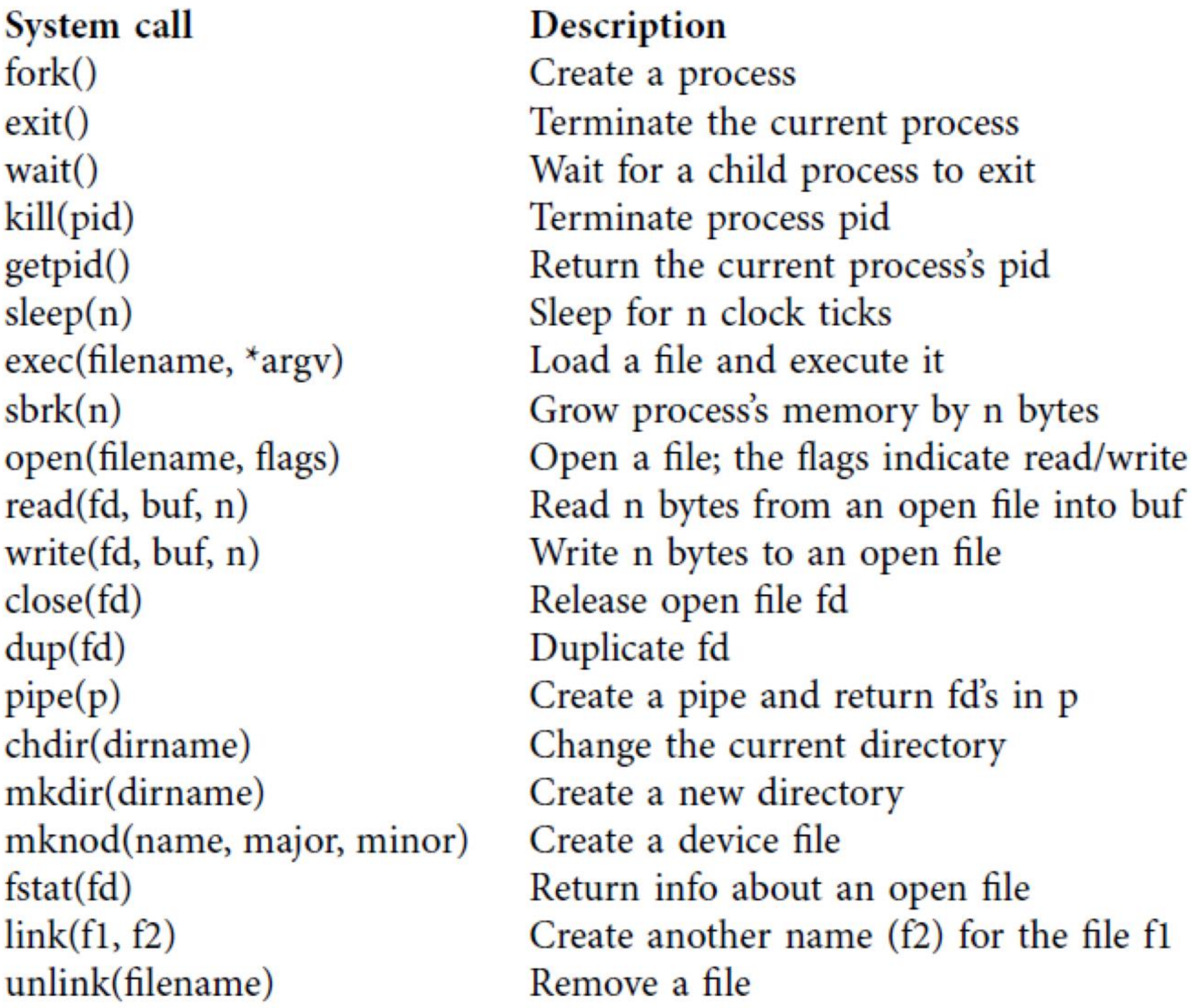
OS is essentially a library, a collection of modules.

Library vs. Kernel

Library
- can access only process's user address space.

Kernel
- can access any hardware address.
- priviledged

system calls in xv6



System call	Description
fork()	Create a process
exit()	Terminate the current process
wait()	Wait for a child process to exit
kill(pid)	Terminate process pid
getpid()	Return the current process's pid
sleep(n)	Sleep for n clock ticks
exec(filename, *argv)	Load a file and execute it
sbrk(n)	Grow process's memory by n bytes
open(filename, flags)	Open a file; the flags indicate read/write
read(fd, buf, n)	Read n bytes from an open file into buf
write(fd, buf, n)	Write n bytes to an open file
close(fd)	Release open file fd
dup(fd)	Duplicate fd
pipe(p)	Create a pipe and return fd's in p
chdir(dirname)	Change the current directory
mkdir(dirname)	Create a new directory
mknod(name, major, minor)	Create a device file
fstat(fd)	Return info about an open file
link(f1, f2)	Create another name (f2) for the file f1
unlink(filename)	Remove a file

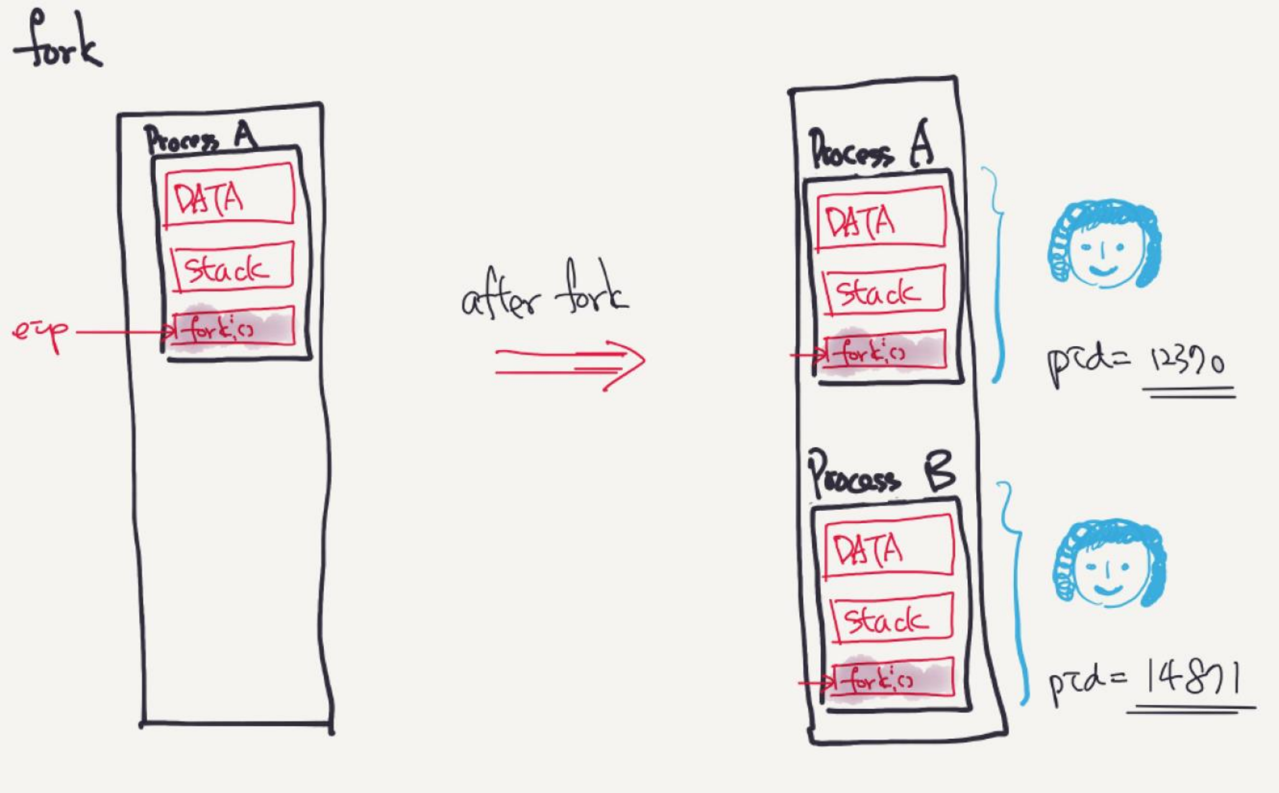
Process and Memory

Process and memory

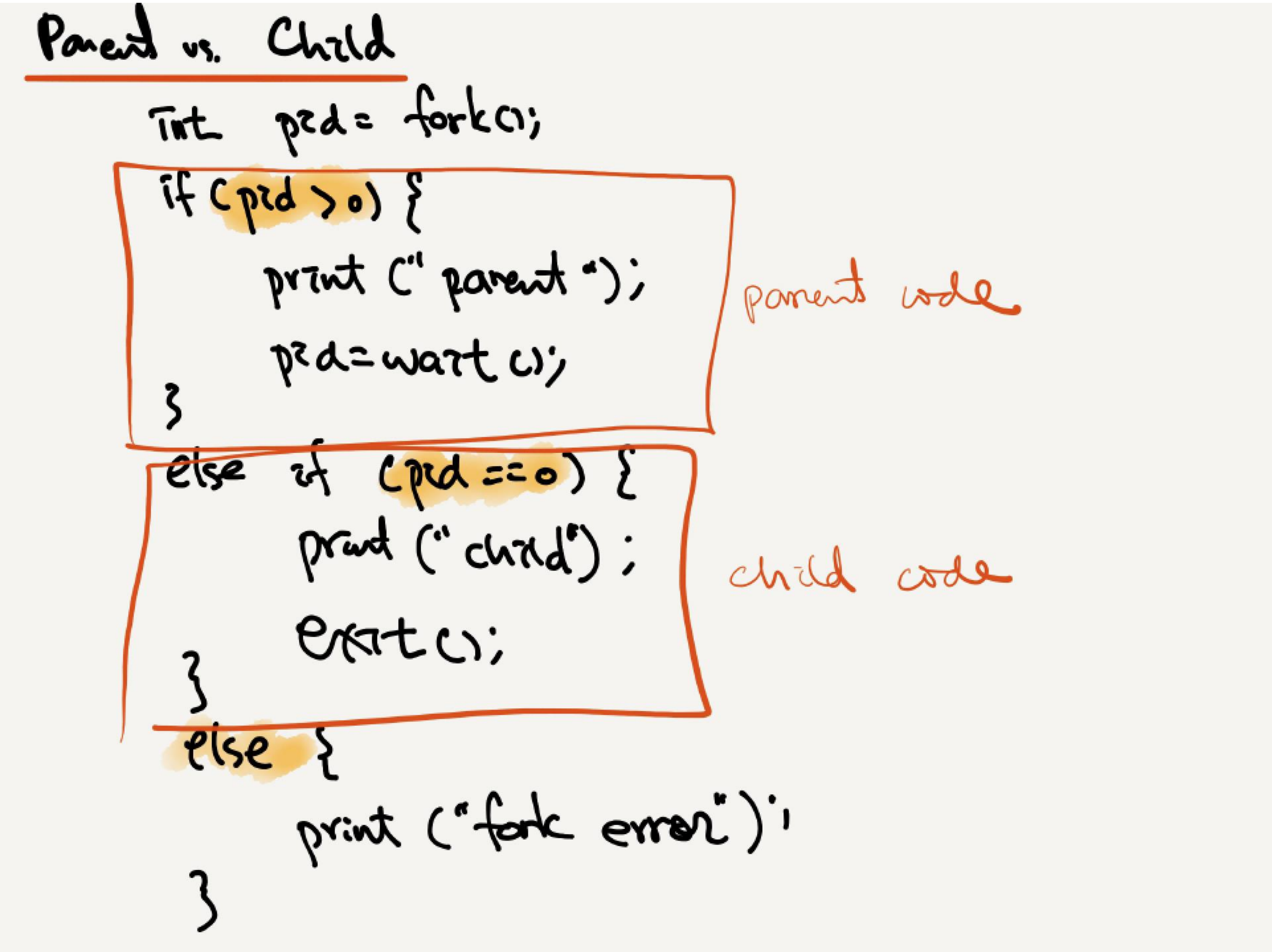
- process = user memory (instructions, stacks and data) + process state
- context switch to execute multiple processes
- each process has pid
- System calls
  - `fork`
  - `wait`
  - `exit`

fork ()

- creates child process
  - child process is allocated separate memory space from the process. The child process has the same memory contents
  - for parent, fork() returns PID of child process; for child process, fork() returns 0.



`fork()`: parent vs. child



Parent vs. Child

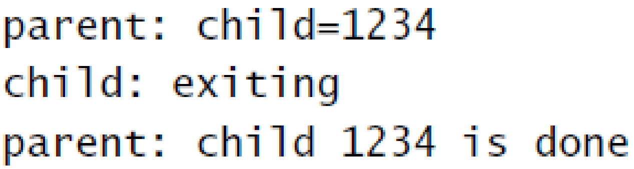
```
    Int  pid= fork();
    if (pid > 0) {
        print (" parent ");
        pid=wait ();
    }
    else  if  (pid ==0) {
        print (" child") ;
        exit ();
    }
    else {
        print ("fork error");
    }
```

parent code

child code

fork ()

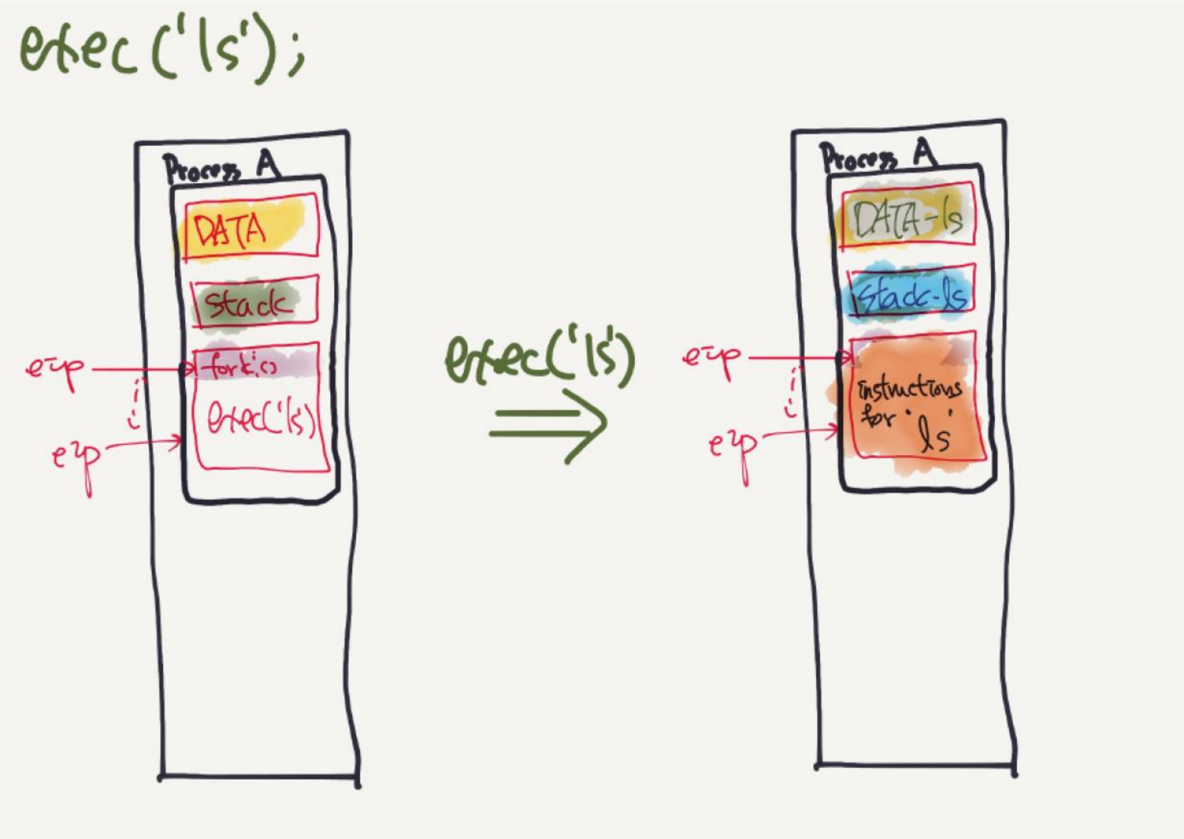
```
int pid = fork();
if(pid > 0) {
  printf("parent: child=%d\n", pid);
  pid = wait();
  printf("child %d is done\n", pid);
} else if(pid == 0) {
  printf("child: exiting\n");
  exit();
} else {
  printf("fork error\n");
}
```



```
parent: child=1234
child: exiting
parent: child 1234 is done
```

exec ()

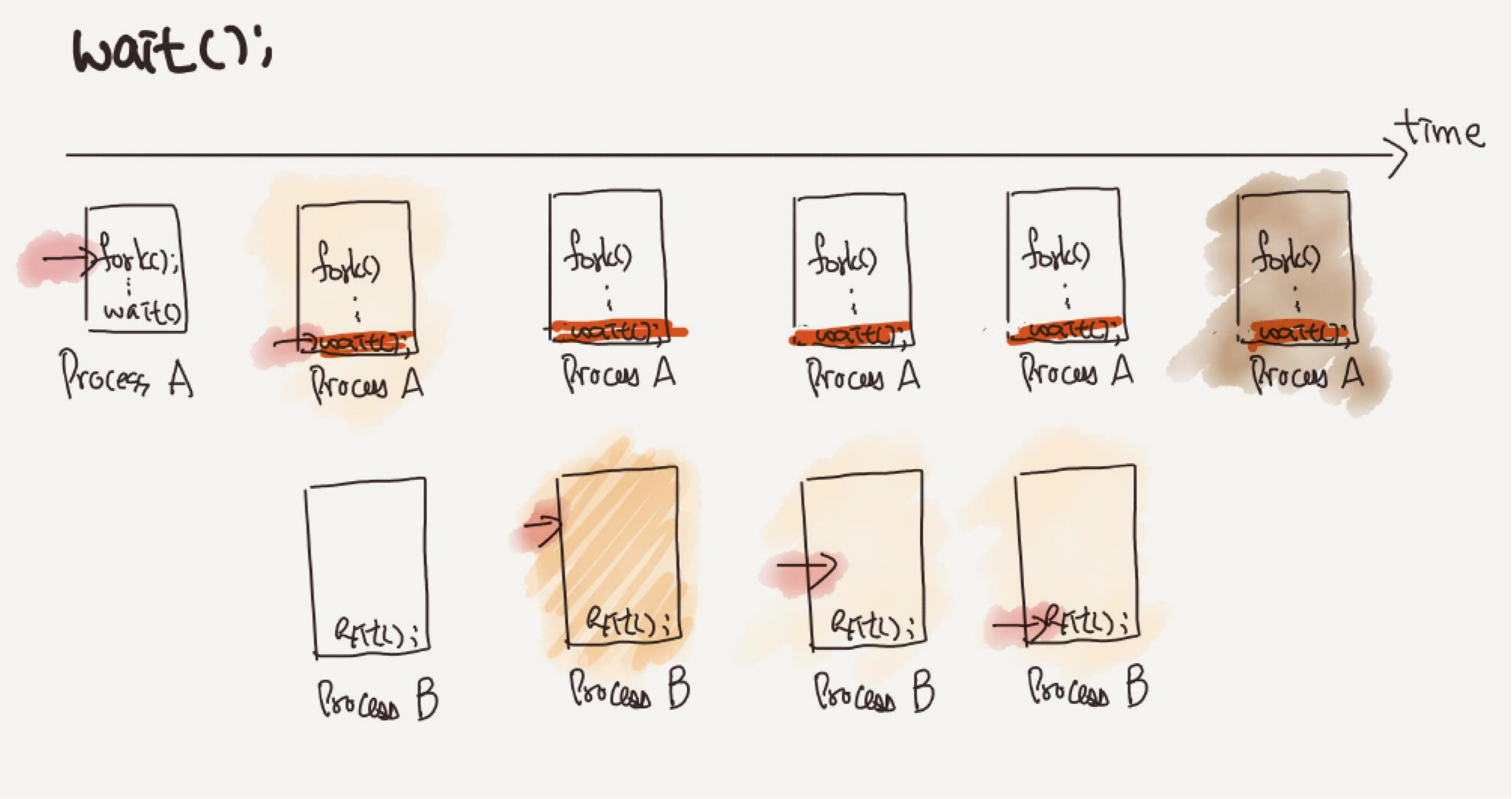
- Replace the text segment with a new text segment, set up the new stack and heap.
- When succeeds, it starts to execute the newly loaded binary file.
- Parameter of `exec():` name of executable and array of parameters



exec ()

```
char *argv[3];

argv[0] = “echo”;
argv[1] = “hello”;
argv[2] = 0;
exec(“/bin/echo”, argv);
printf(“exec error\n”);
```

wait ()

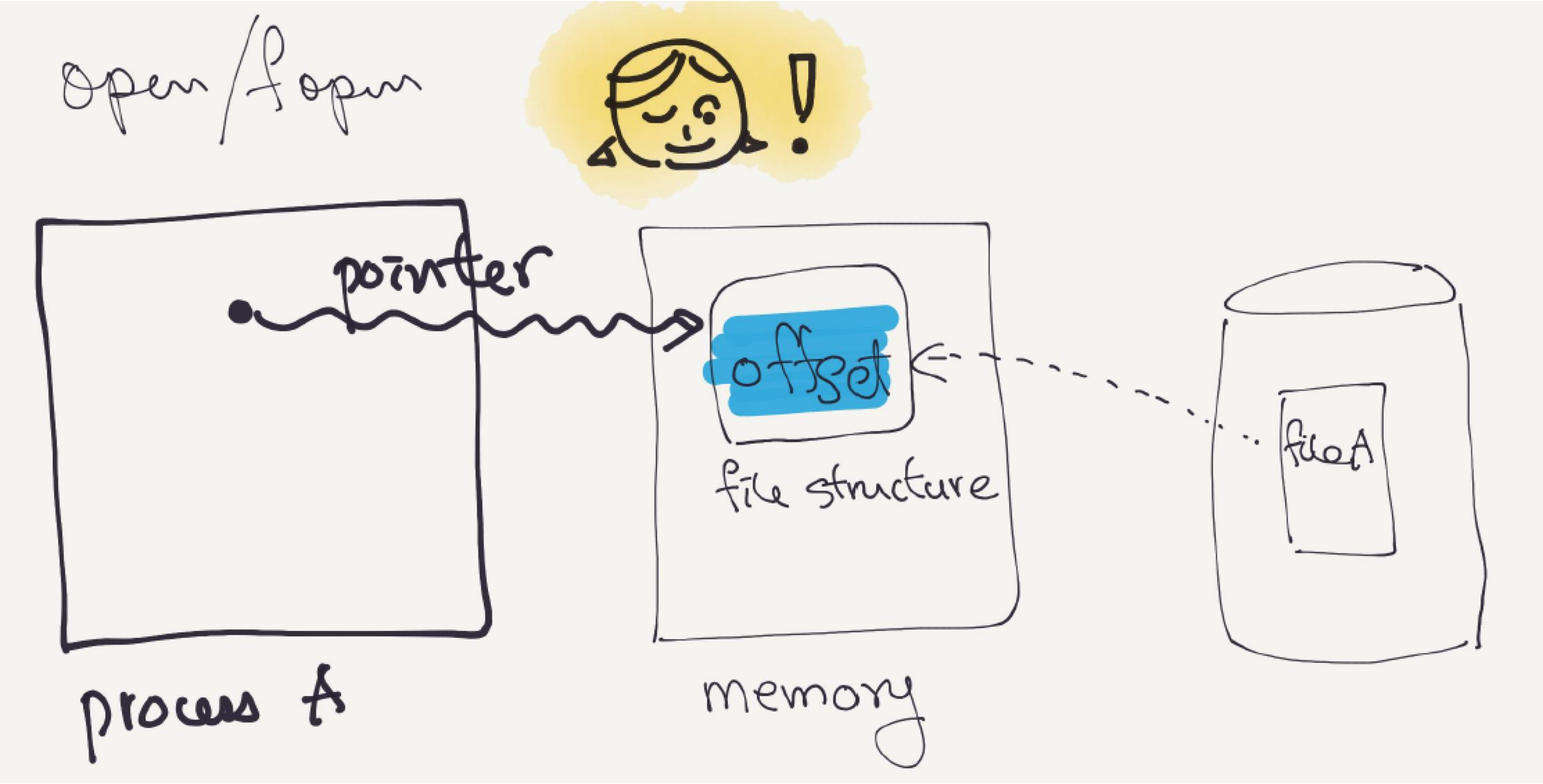

wait();
time
fork();
wait()
Process A
Process A
Process A
Process A
Process A
Process A
exit();
Process B
Process B
Process B
Process B

File

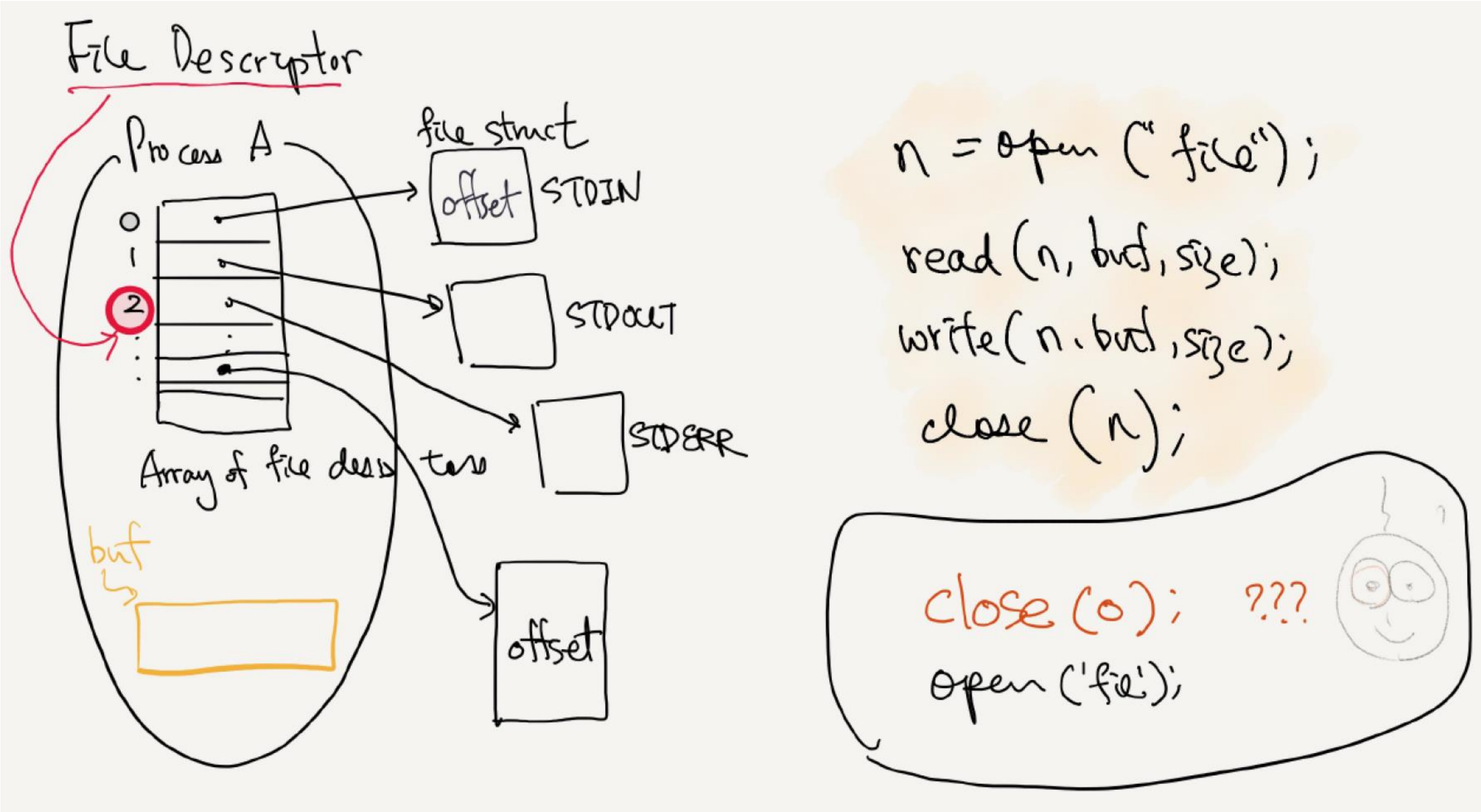
File

- File descriptor
  - an integer that represents a file, a pipe, a directory and a device
  - In most OS, file descriptor is an index in the per-process file descriptor table.
  - File descriptor 0 (Standard Input), 1 (Standard Output), 2 (Standard Error).
  - Shell exploits these default file descriptors to implement redirection and pipe.
    - Redirection: `% cat < ”input.txt”`
    - Pipe: `% ls | wc`

I/O and File descriptor (Cont.)



I/O and File descriptor (Cont.)



```
n = open("file");
read(n, buf, size);
write(n.buf, size);
close(n);
```

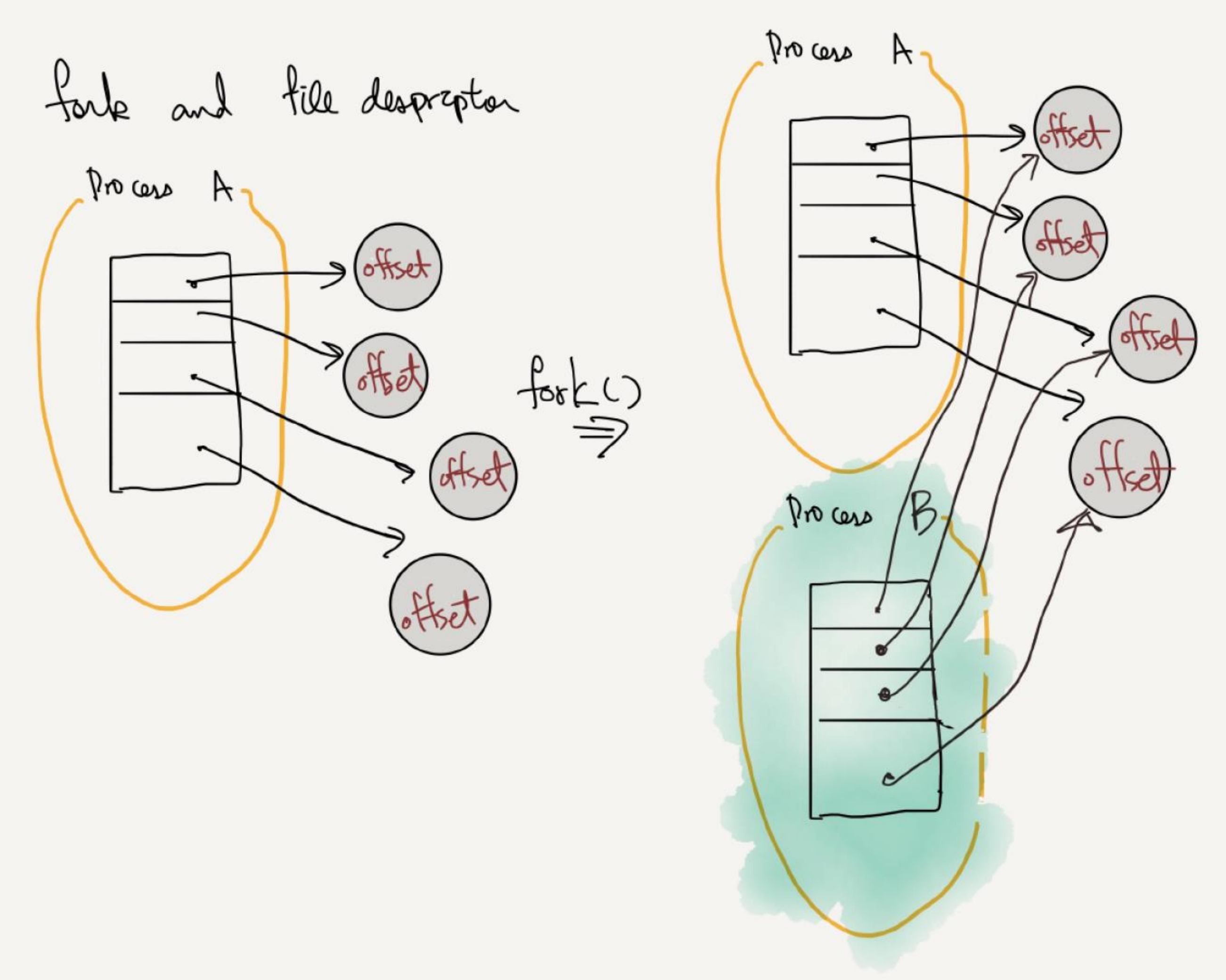
```
close(0);
open('file');
```

I/O and File descriptor (Cont.)

- `close(fd)`
  - deallocate the File descriptor 'fd'.
  - When allocating the new file descriptor, it uses the smallest 'free' file descriptor from the file descriptor table.

- File descriptor and system call
  - `fork()` copies the File descriptor table from the parent to child process.
  - `exec()` retains the File descriptor table.
  - It makes the I/O redirection through `fork(), reopen(), and exec().`

I/O and File descriptor (Cont.)



```
if(fork() == 0) {
  write(1, "hello ", 6);
  exit();
} else {
  wait();
  write(1, "world\n", 6);
}
```

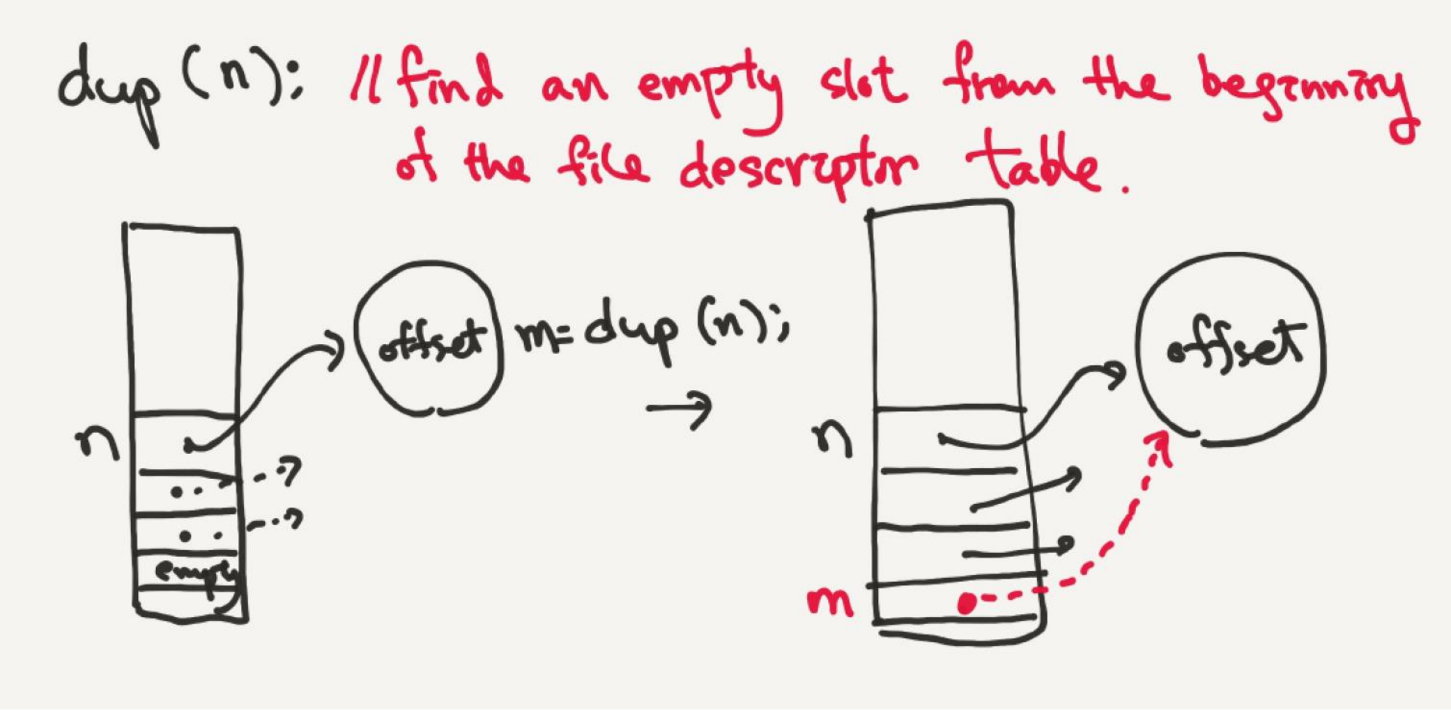
IO redirection

- Redirection
  - Close File descriptor 0~2 and then open new file. —> Then, the user can use fd 0,1,2 to access regular file.
  - In shell, you can use ‘>’. ex) `% ls > test.out`
- what happens in the following piece of code?

```
char *argv[2];

argv[0] = “cat”;
argv[1] = 0;
if(fork() == 0) {
   close(0);
   open(“input.txt”, O_RDONLY);
   exec(“cat”, argv);
}
```

dup (fd)

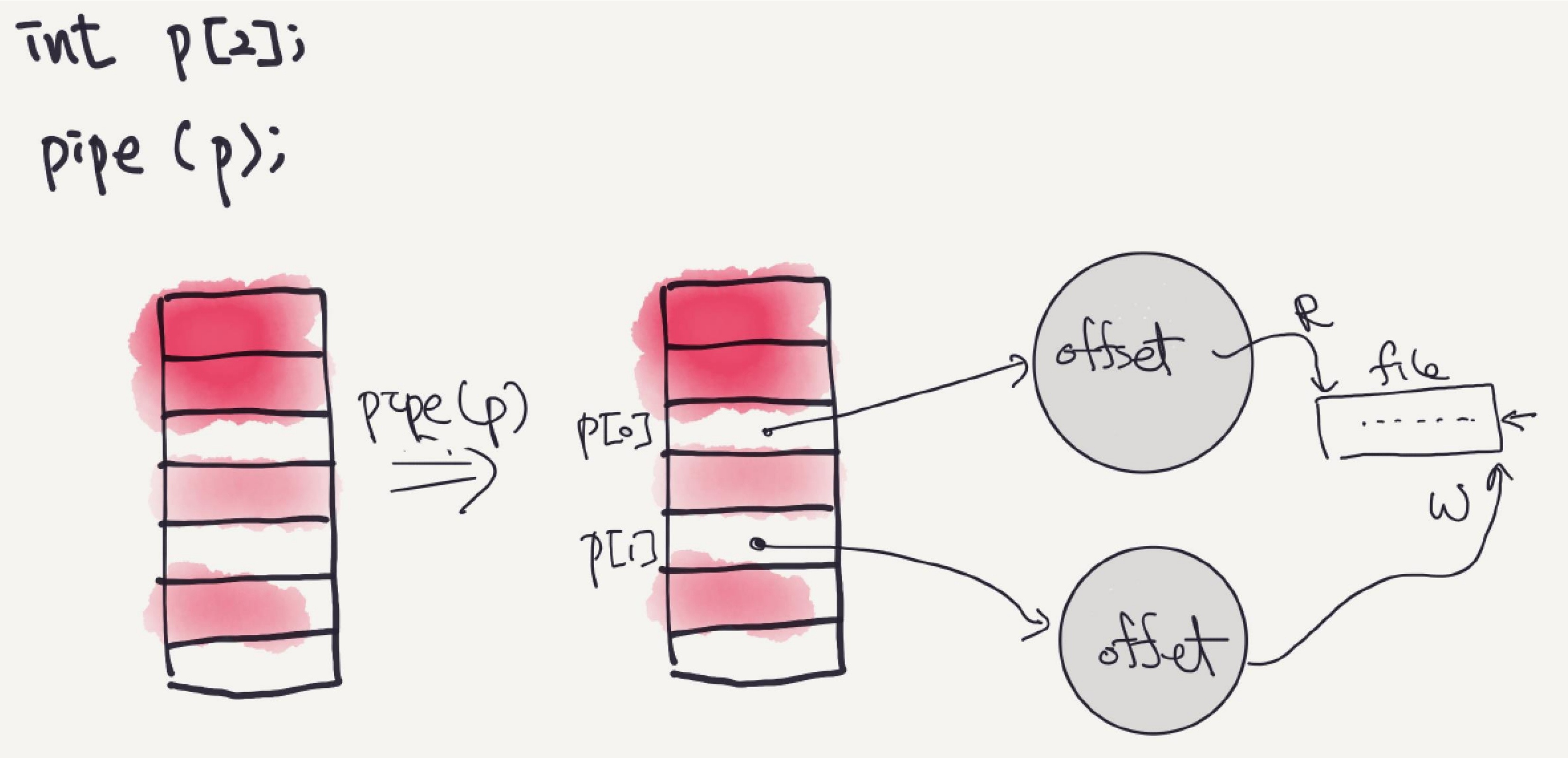
- Duplicate a file descriptor and return new file descriptor.


dup (n): // find an empty slot from the beginning of the file descriptor table.
n
empty
offset
m= dup (n);
n
m
offset

```
fd = dup(1);
write(1, "hello ", 6);
write(fd, "world\n", 6);
```

Pipe

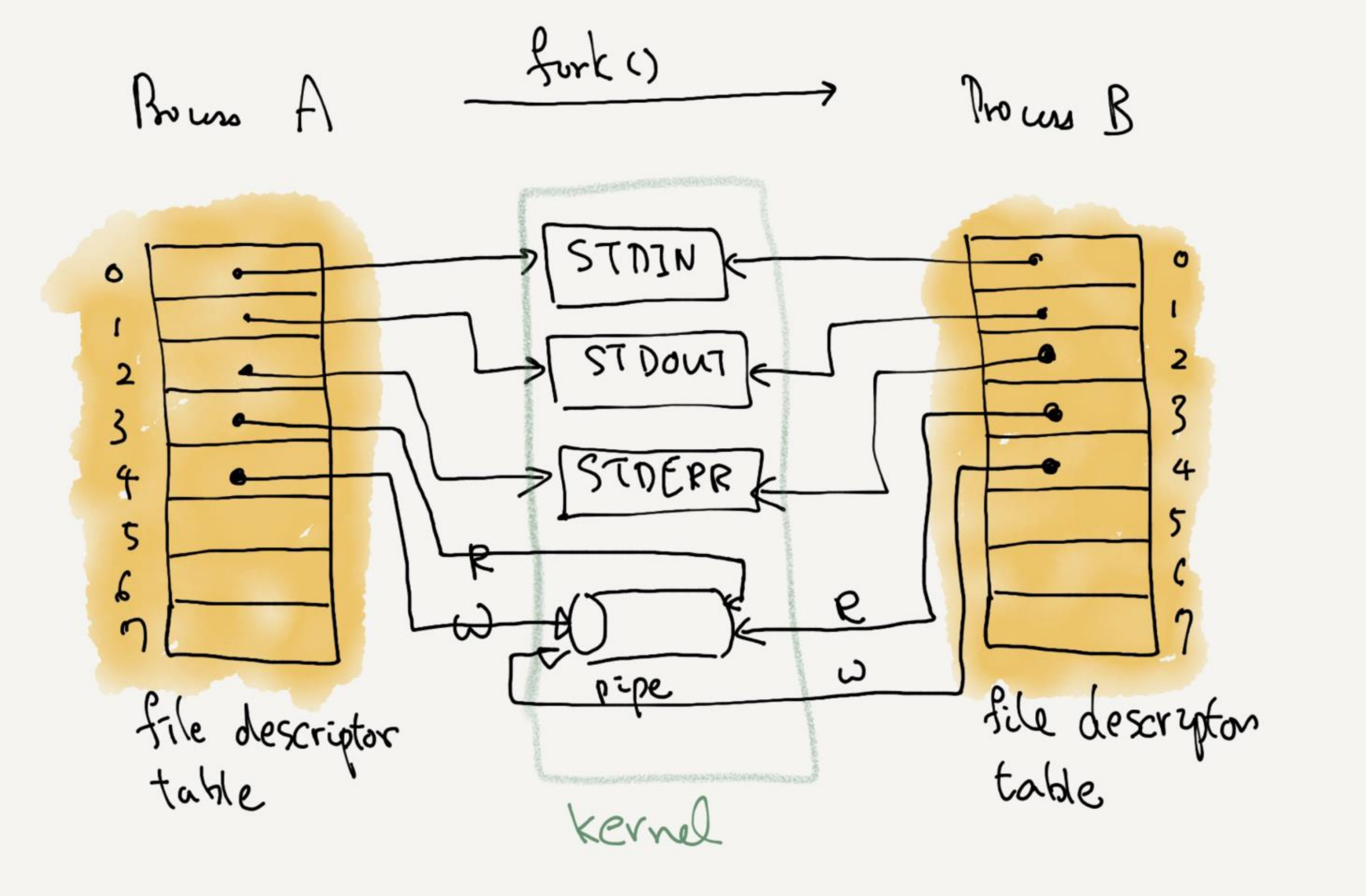
- special type of file, a kernel buffer that is exposed to a process via a pair of file descriptors: one for read and one for write.



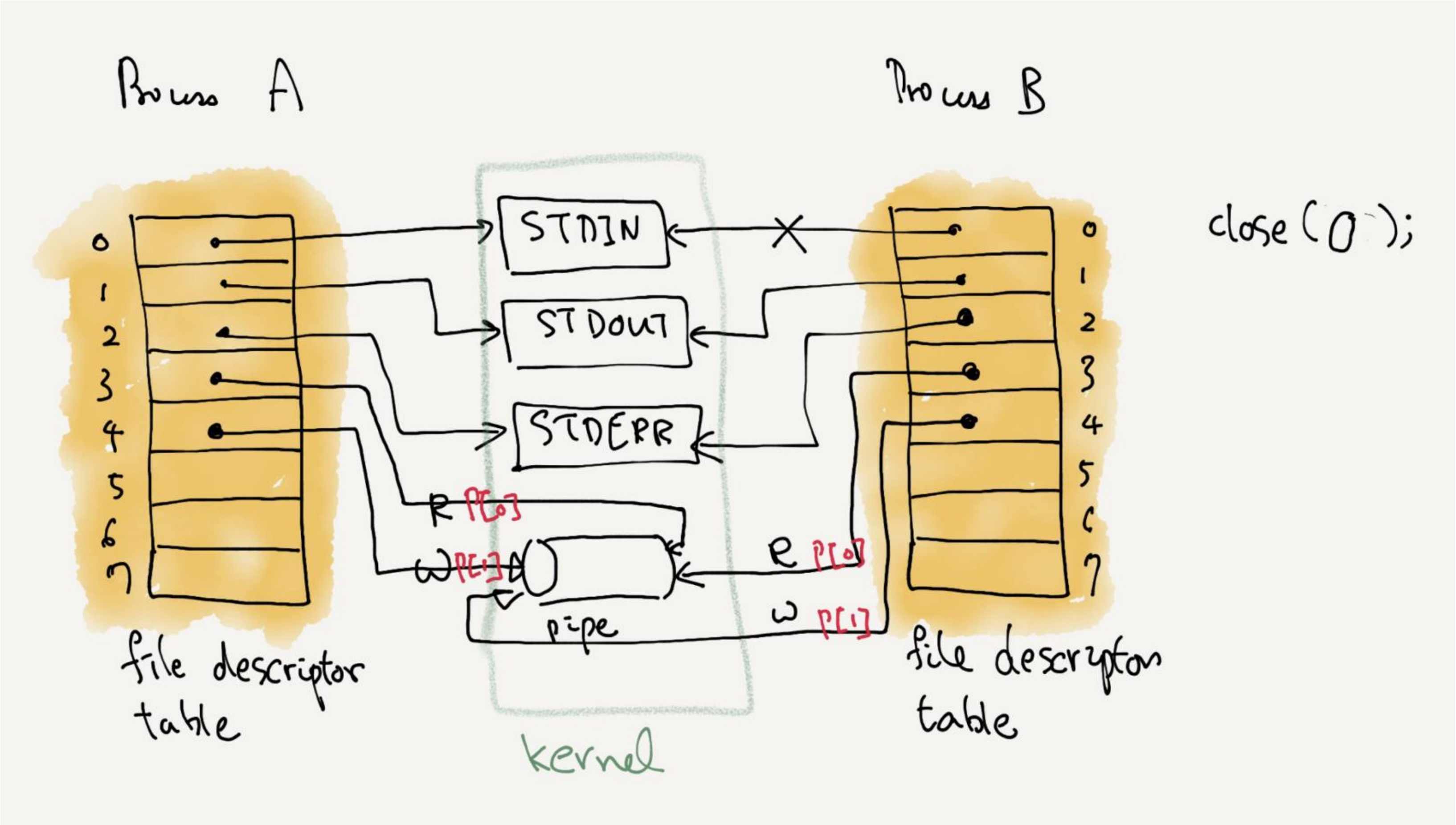
Pipes and **wc** (word count)

```
int p[2];
char *argv[2];

argv[0] = “wc”;
argv[1] = 0;

pipe(p);
if(fork() == 0) {// child
   close(0);
   dup(p[0]);
   close(p[0]);
   close(p[1]);
   exec(“/bin/wc”, argv);
} else { // parent
   close(p[0]);
   write(p[1], “hello world\n”, 12);
   close(p[1]);
}
```

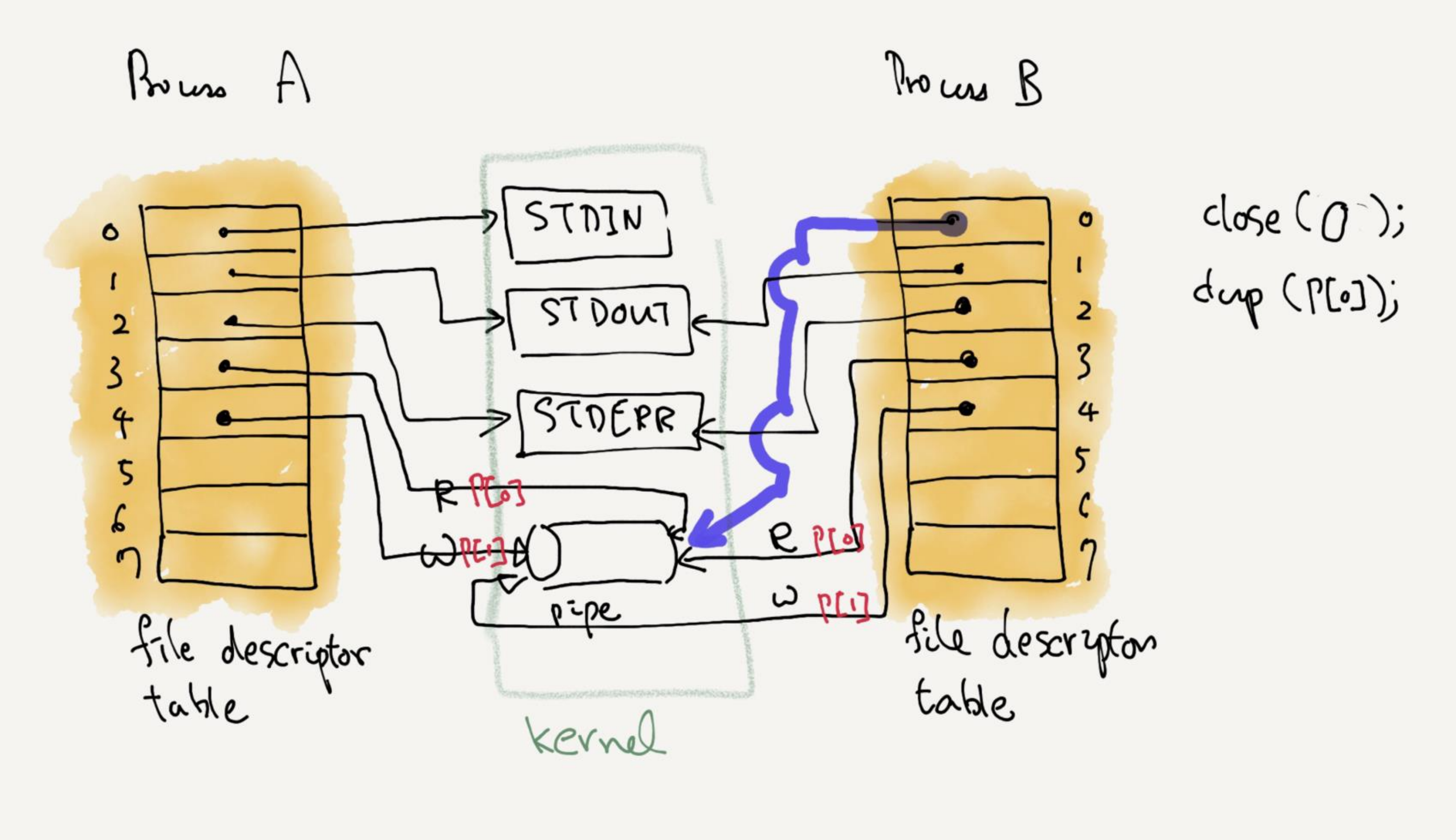
pipe and fork



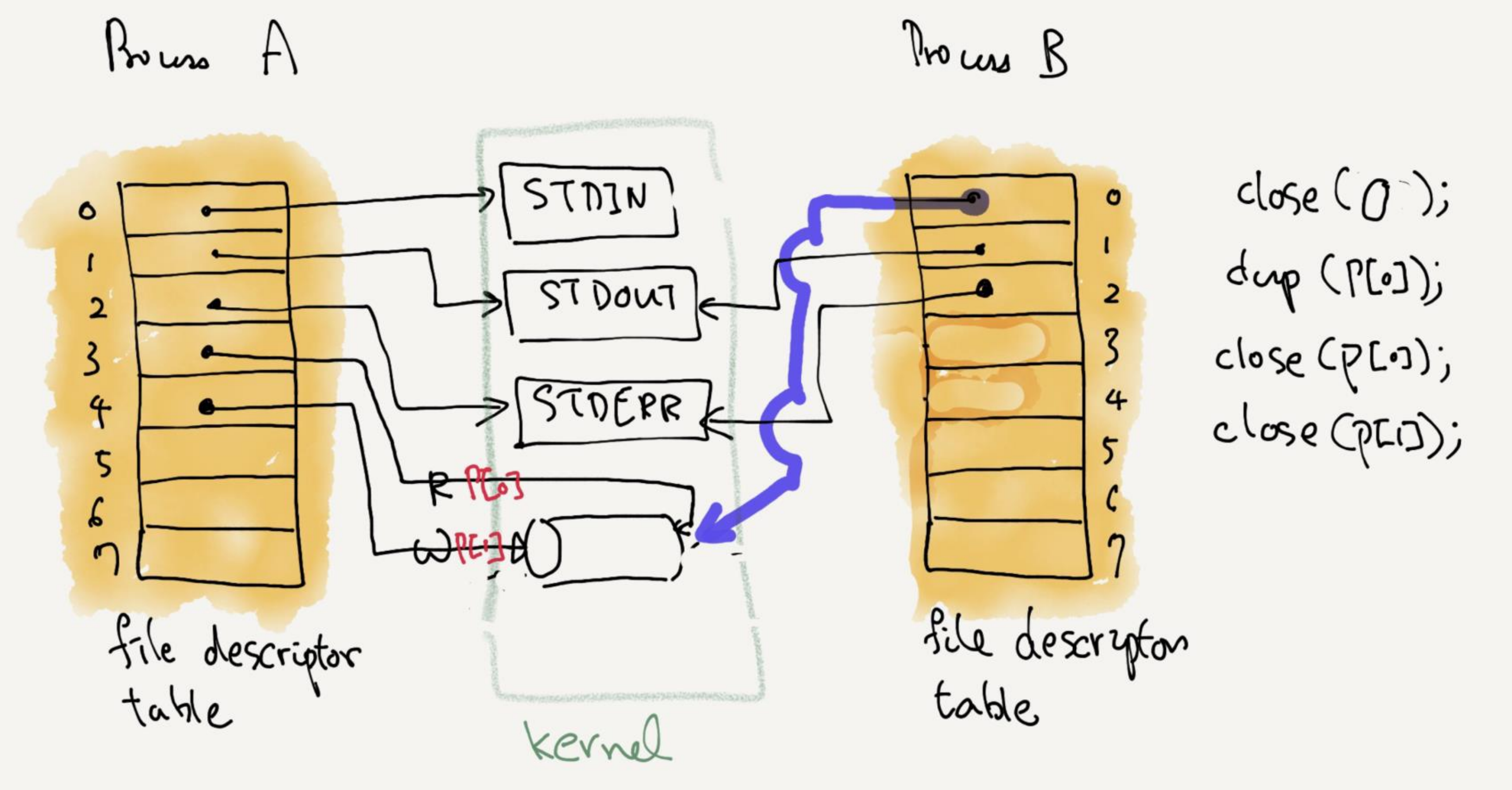
pipe and fork



pipe and fork

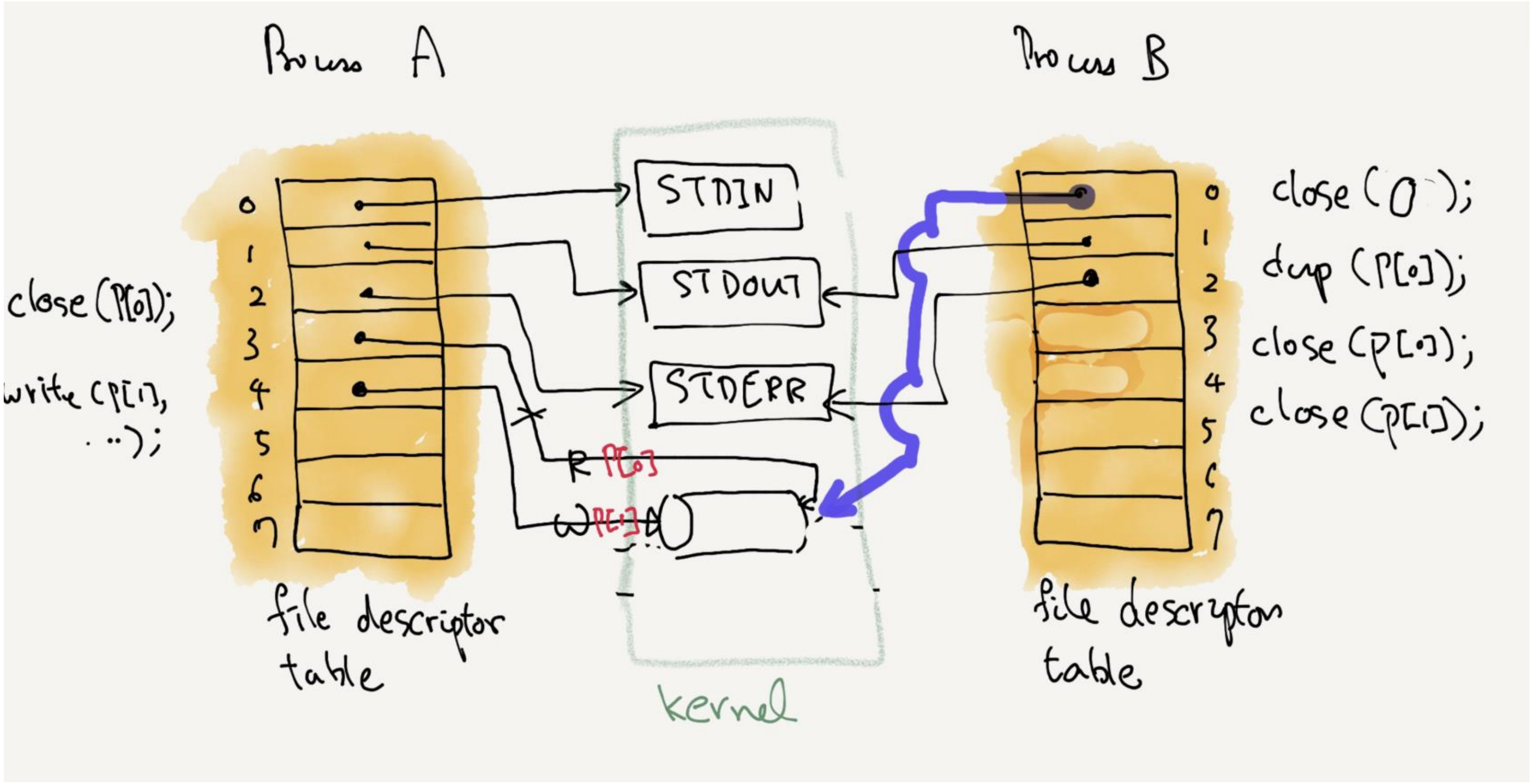


pipe and fork

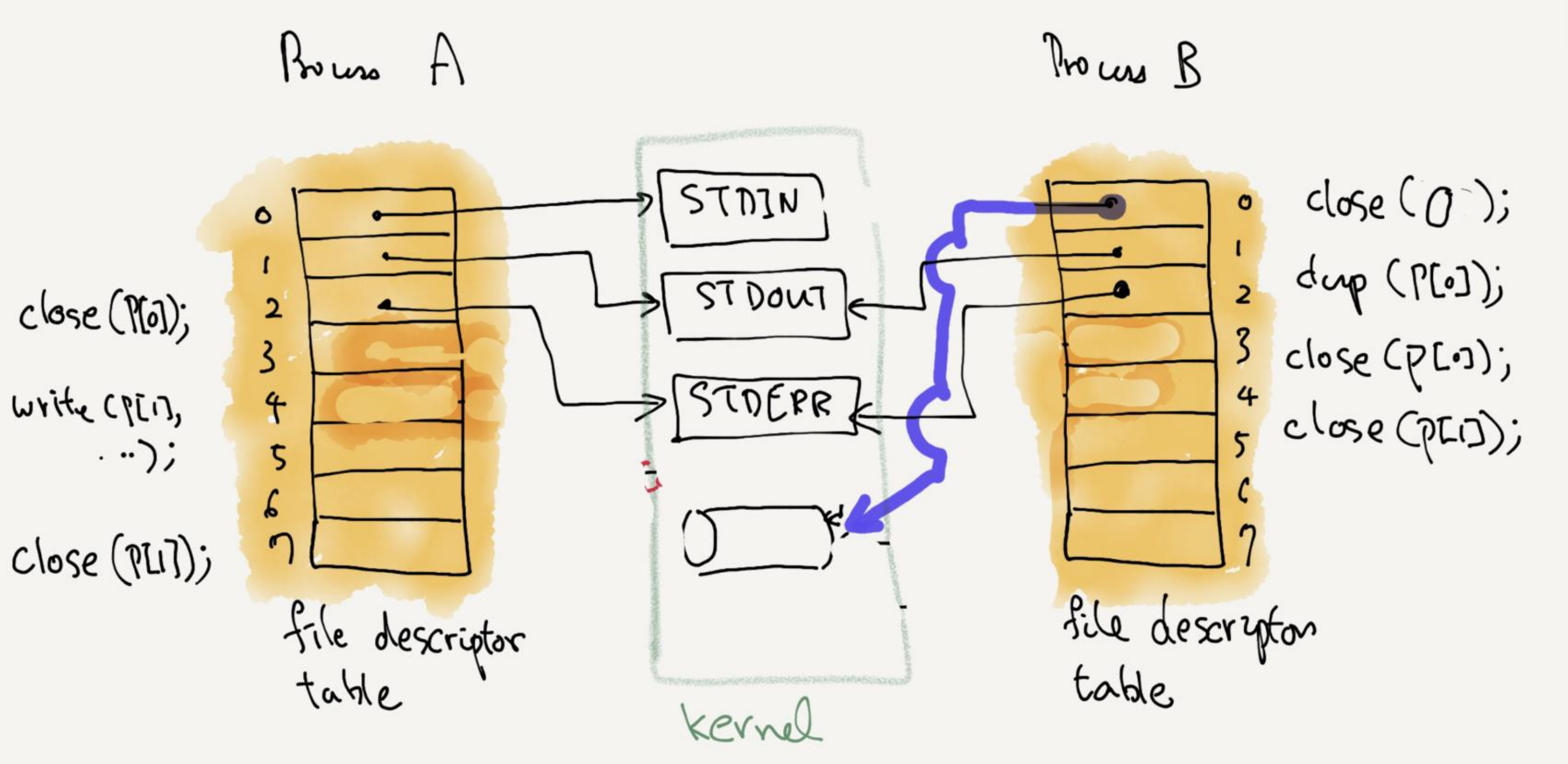


```
close(0);
dup(P[0]);
close(p[0]);
close(p[1]);
```

pipe and fork



pipe and fork



Pipes

```
echo hello world | wc
```

vs.

```
echo hello world > ttmp/xyz ; wc </tmp/xyz
```

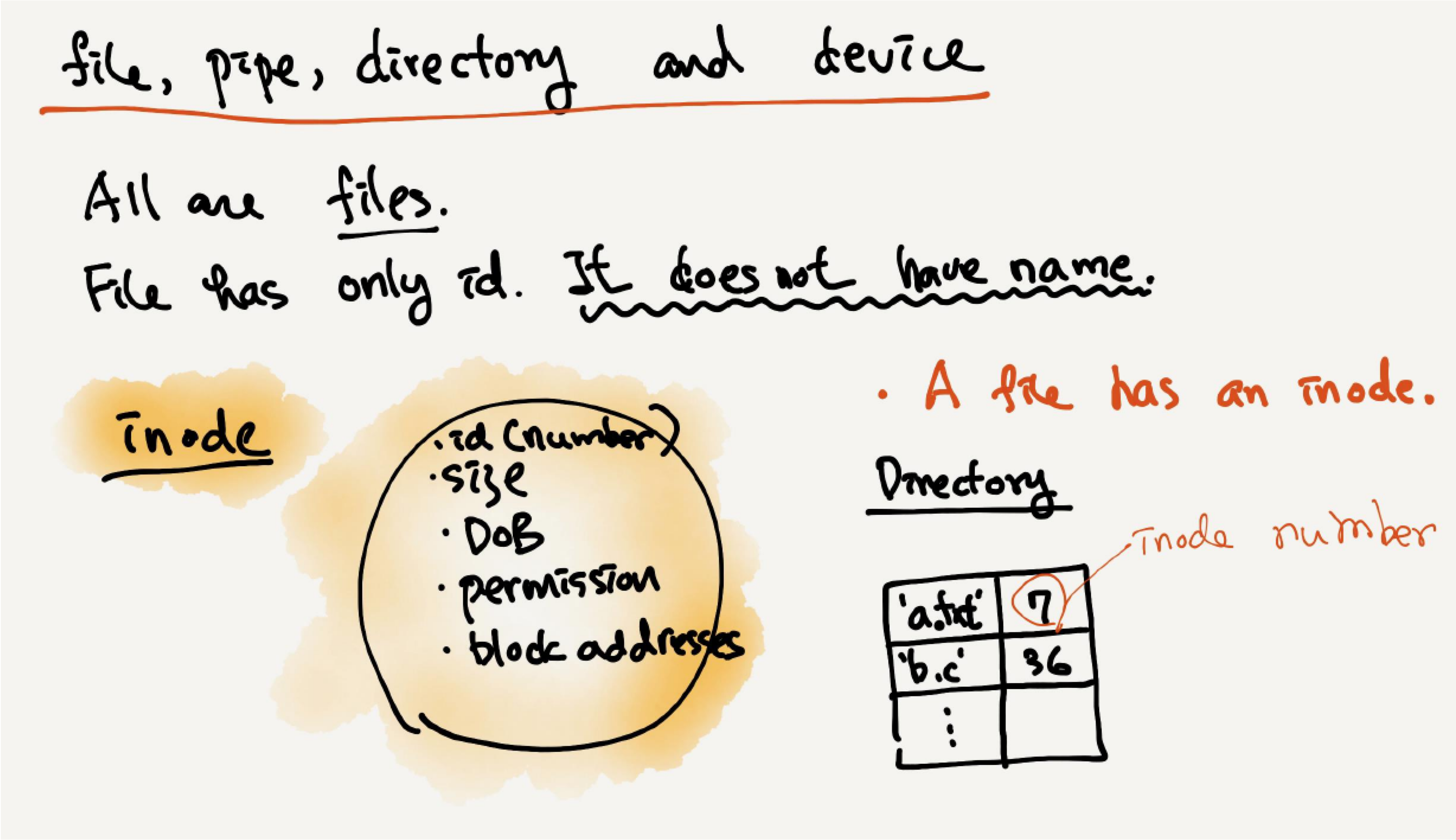
- advantages of pipes over using redirection with temporary files
  - pipe automatically clean themselves up. When using temporary file, the user has to explicitly delete it.
  - pipe can pass arbitrarily long data while file redirection requires sufficient available disk space.
  - In pipe, reader and write can proceed in parallel while in redirection, the one has to finish for the others to start.
  - To implement inter-process communication, blocking reads and writes are more efficient than non-blocking ones.

Filesystem

- creating a file
  - mkdir : creating a directory.
  - open with O_CREATE : create a new file.
  - mknode : create a new device file.

```
mkdir(“/dir”);
fd = open(“/dir/file”, O_CREAT|O_WRONLY);
close(fd);
mknod(“/console”, 1, 1);
```

File system (Cont.)



file, pipe, directory and device

All are files.

File has only id. It does not have name.

- A file has an inode.

inode

- id (number)
- size
- DoB
- permission
- block addresses

Directory

Inode number

'a.txt'	7
'b.c'	36
⋮	

File system (Cont.)

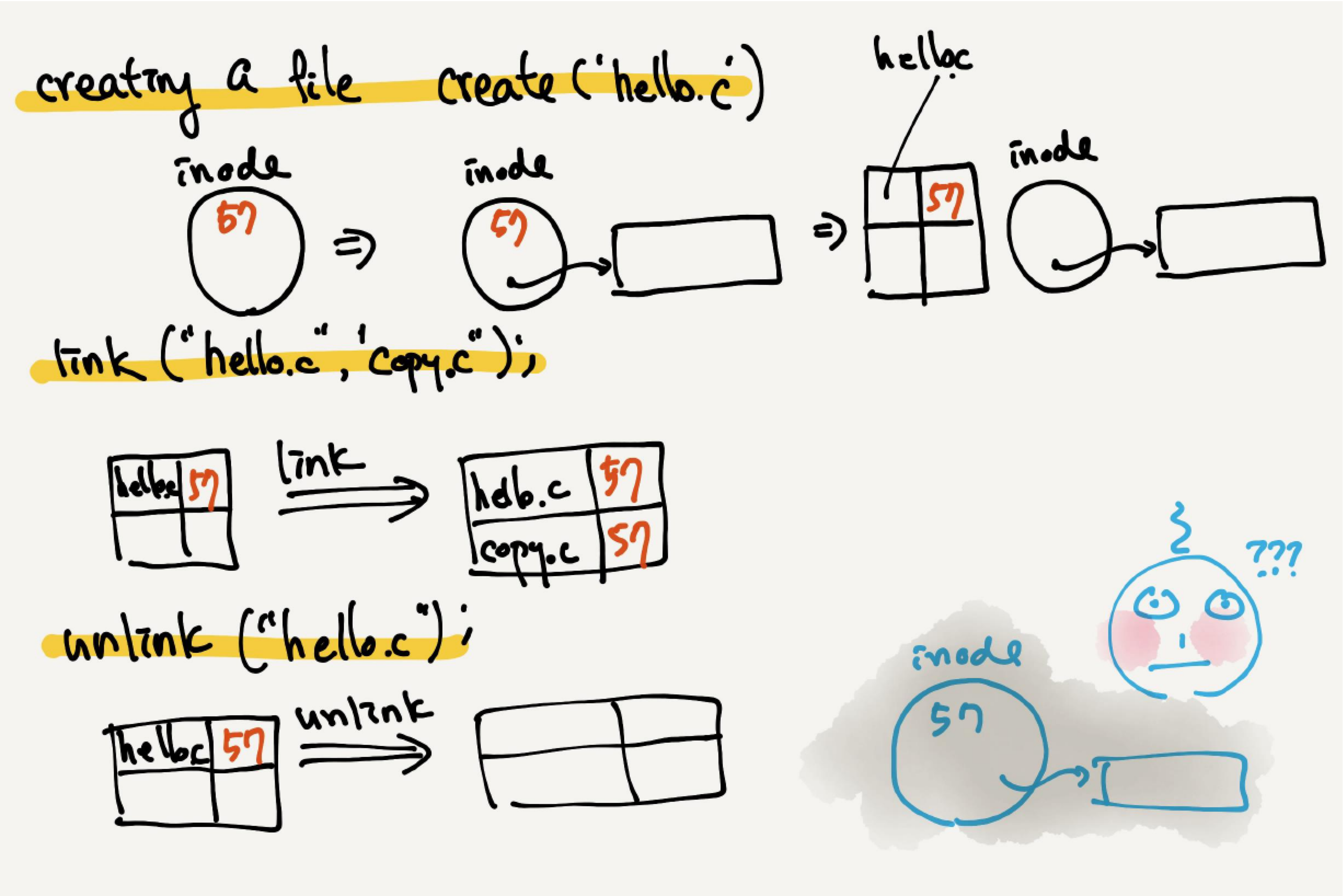
- `link`
  - creates another name for an inode.
  - same inode number, so are the results of the fstat.
  - nlink: the number of links to an inode.

```
open(“a”, O_CREAT|O_WRONLY);
link(“a”, “b”);
```

- `unlink`
  - remove the link between the inode and the name.
  - Operating system reclaims the inode and the associated disk space when nlink becomes 0 inode and there is no file descriptor associated with it.

```
open(“a”, O_CREAT|O_WRONLY);
link(“a”, “b”);
unlink(“a”);
```

File system (Cont.)



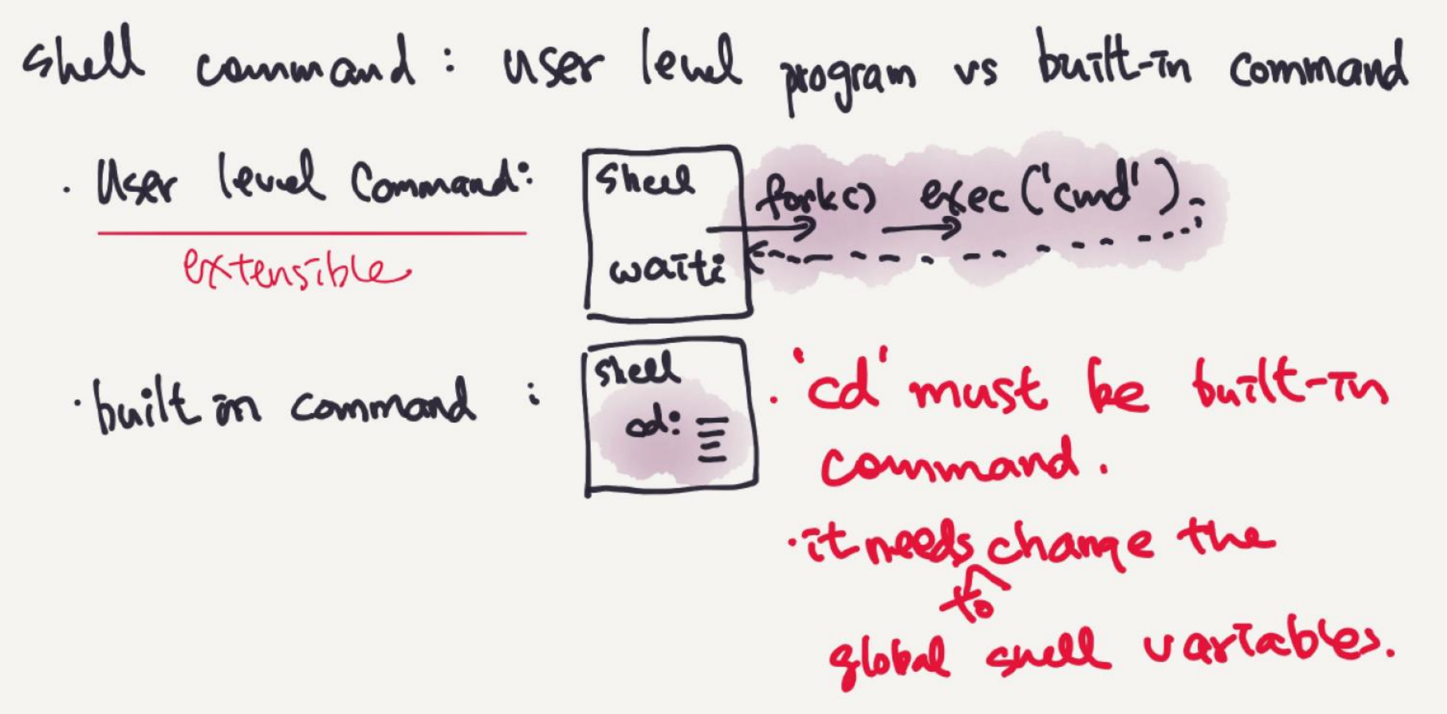
creating a file create ('hello.c')

link ("hello.c", "copy.c");

unlink ("hello.c");

command types in shell

- user program with fork()/exec(): `mkdir`, `ln`, `rm`
- built-in command: `cd`
  - 'cd' needs to change the current directory. When the shell calls `fork()` and calls exec('cd'), it changes the current directory of the child process, not the shell itself. ‘cd’ should be implemented as a shell itself, not as a user program.



shell command : user level program vs built-in command

- User level Command: extensible
- built in command :
- 'cd' must be built-in command.
- it needs to change the global shell variables.

Summary

- What is system software?
- Basics of “process/memory” and “file”
- pipe() (and signal) is heart of the modern Unix OS.:
  - pipe enables shell programming.
  - Shell program enables to build a large program with a set of small programs.