Operating Systems

Process API

Overview

- `fork()`
- `exec()`
- `wait()`
- Separation of `fork()` and `exec()`
  - IO redirection
  - pipe

Creating a child process

`fork()`

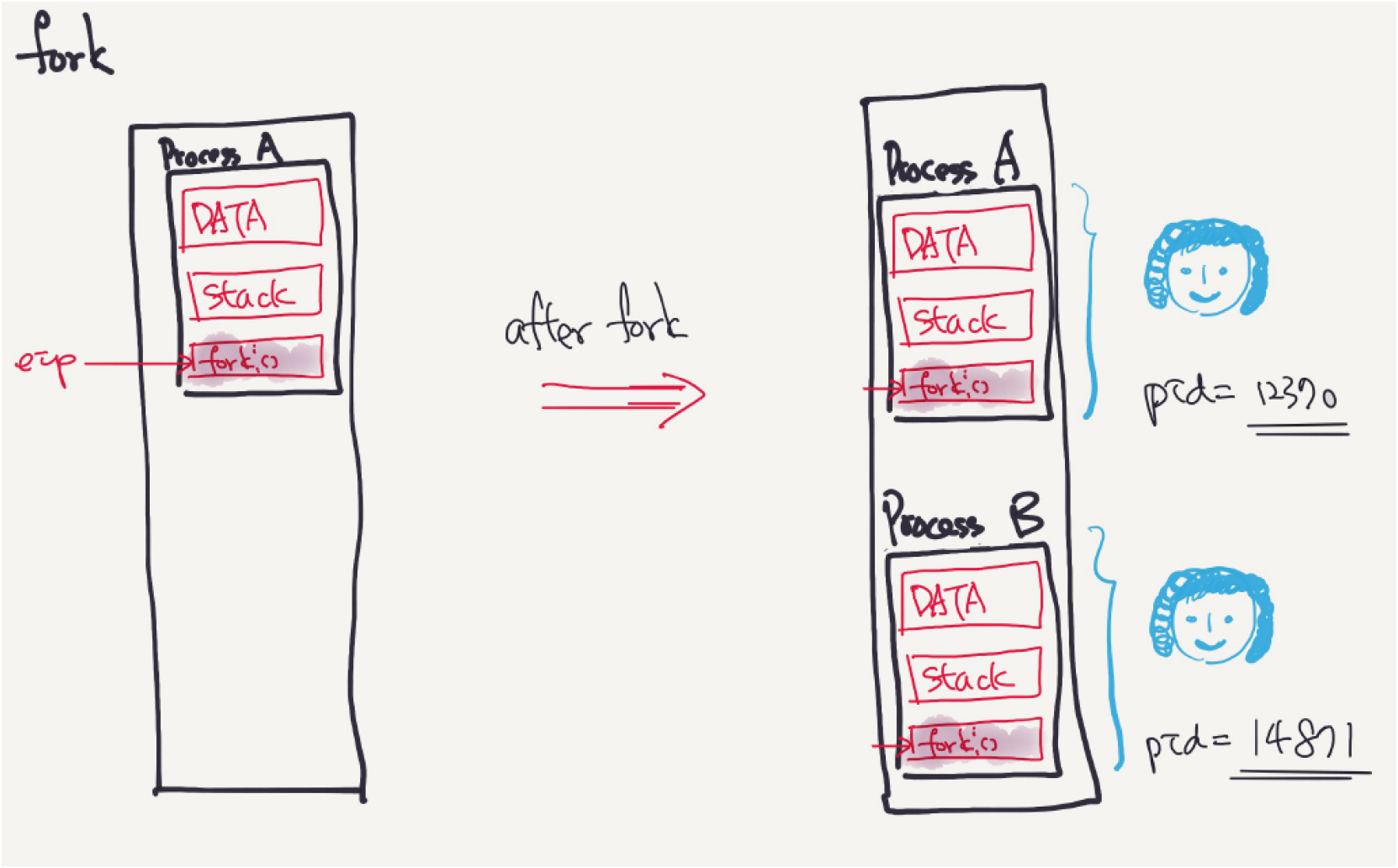
- Create a child process
  - child process is allocated separate memory space from the process. The child pro cess has the same memory contents as the parents.
  - The child process has its own **registers**, and program counter register(**PC)**.
  - The newly created process becomes independent after it is created.
  - for parent, `fork()` returns PID of child process; for child process, `fork()` returns 0.

Usage of `fork()`

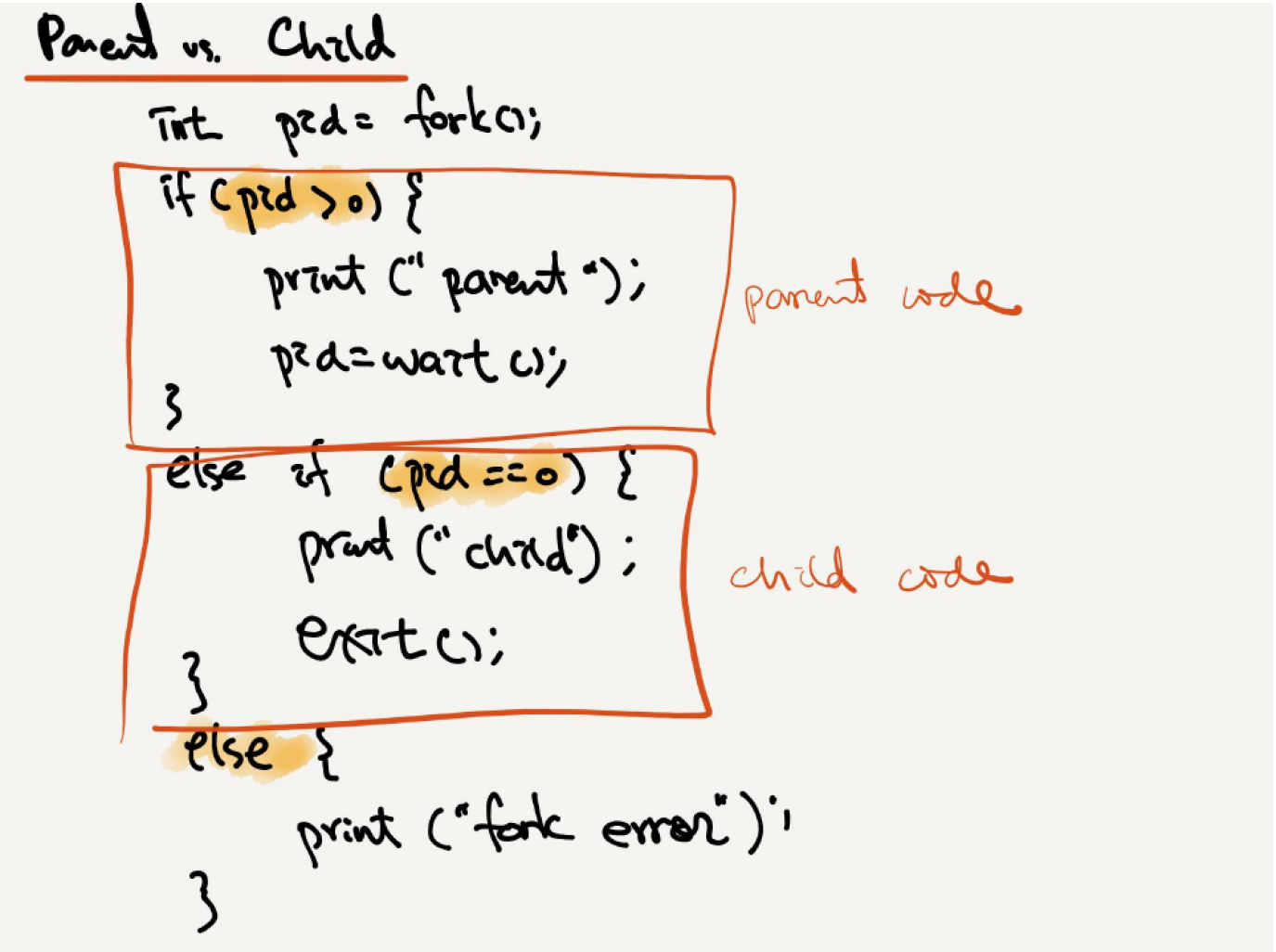
p1.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]){
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {   // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
    } else {                    // parent goes down this path (main)
        printf("hello, I am parent of %d (pid:%d)\n",
        rc, (int) getpid());
    }
    return 0;
}
```


fork
Process A
DATA
Stack
fork();
esp
after fork
Process A
DATA
Stack
fork();
pid= 12390
Process B
DATA
Stack
fork();
pid= 14871

fork(): parent vs. child



Let's run it.

```
prompt> ./p1
hello world (pid:29146)
hello, I am parent of 29147 (pid:29146)
hello, I am child (pid:29147)
prompt>
```

or

```
prompt> ./p1
hello world (pid:29146)
hello, I am child (pid:29147)
hello, I am parent of 29147 (pid:29146)
prompt>
```

Create the dependency between bewteen the processes

`wait()`

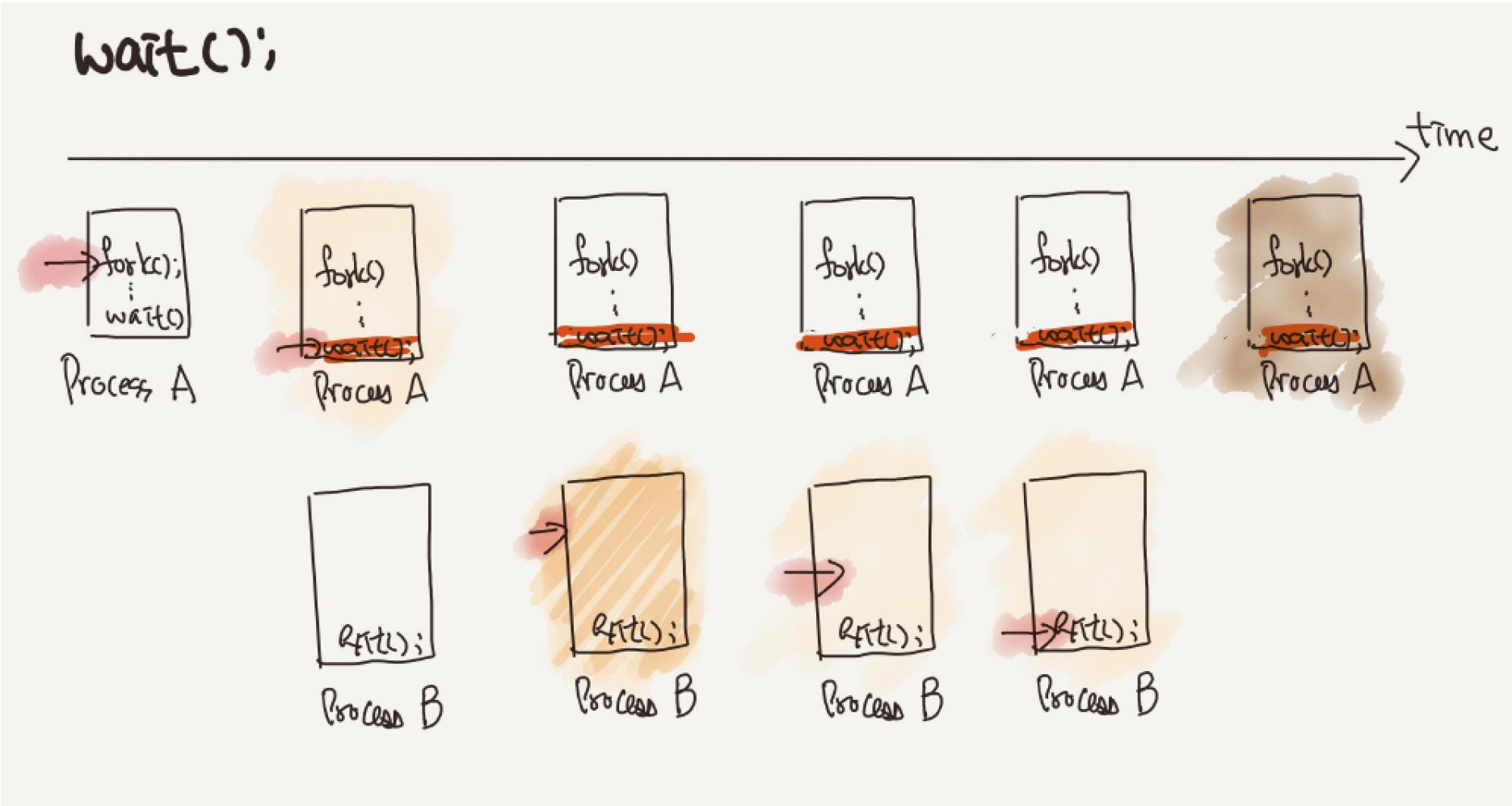
- When the child process is created, `wait()` in the parent process won't return until the child has run and exited.
- The parent and the child does not have any dependency.
- In some cases, the application wants to enforce the order in which they are executed, e.g. the parent exits only after the child finishes.

The usage of `wait()` System Call

p2.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char *argv[]){
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {          // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
    } else {               // parent goes down this path (main)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
        rc, wc, (int) getpid());
    }
    return 0;
}
```


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

The wait() System Call (Cont.)

Result (Deterministic)

```
prompt> ./p2
hello world (pid:29266)
hello, I am child (pid:29267)
hello, I am parent of 29267 (wc:29267) (pid:29266)
prompt>
```

Running a new program

`exec()`

- The caller wants to run a program that is different from the caller itself.
  - Launch an editor
  - `% ls -l`
- OS needs to load a new binary image, initialize a new stack, initialize a new heap for the new program.
- two parameters
  - The name of the binary file
  - The array of arguemtns

```
char *argv[3];

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

Usage of `exec()`

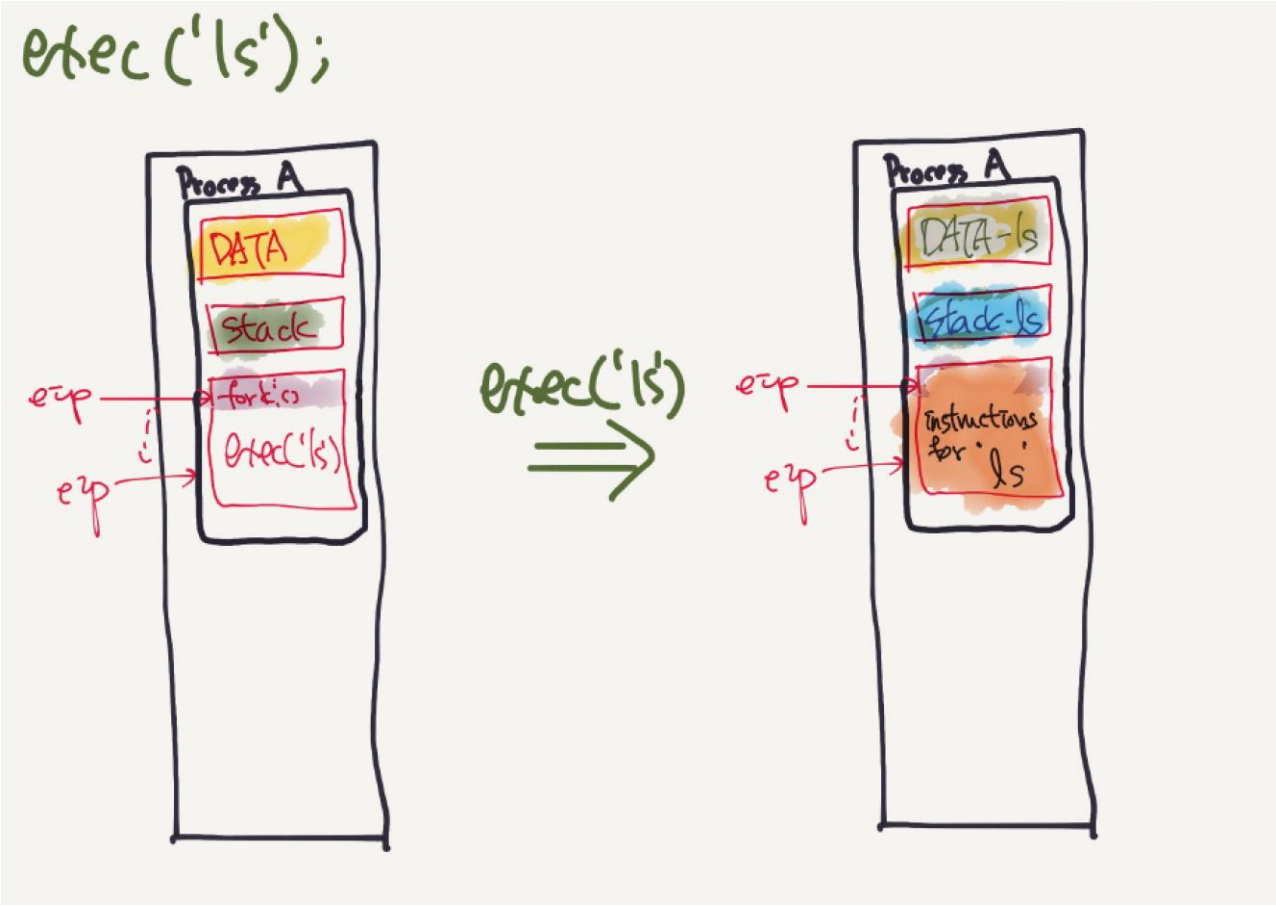
p3.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>

int main(int argc, char *argv[]){
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {              // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {      // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
        char *myargs[3];
        myargs[0] = strdup("wc");                  // program: "wc" (word count)
        myargs[1] = strdup("p3.c");     // argument: file to count
        myargs[2] = NULL;               // marks end of array
        execvp(myargs[0], myargs); // runs word count
        printf("this shouldn't print out");
    } else {                                // parent goes down this path (main)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
            rc, wc, (int) getpid());
    }
    return 0;
}
```

When exec() is called,…

- Replace the existing contents of the memory with the new memory contents from the new binary file.
- `exec()` does not return. It starts to execute the new program.



Usage of `exec()`

Result

```
prompt> ./p3
hello world (pid:29383)
hello, I am child (pid:29384)
29 107 1030 p3.c
hello, I am parent of 29384 (wc:29384) (pid:29383)
prompt>
```

Why separating `fork()` and `exec()`?

- Why don't we just use something like "`forkandexec("ls", "ls -l")`"?
- Via separating `fork()` and `exec()`, we can manipulate various settings just before executing a new program and **make the IO rediction and pipe possible**.

  - IO redirection

```
% cat w3.c > newfile.txt
```

  - pipe

```
% echo hello world | wc
```

'pipe' is the heart of the shell programming.

IO redirection

```
% wc w3.c > newfile.txt
```

- Save the result of '`wc w3.c`' to `newfile.txt`.
- How?
- Shell is a program that `fork()` and `exec()` the command with argument.
  - `% ls -l` ➔ shell calls `fork()` and `exec("ls", "ls-l")` ;
  - Before calling `exec("wc", "wc w3.c")`, the shell closes `STDOUT (close(1))` and opens `newfile.txt` (`open("newfile.txt")`).

Details of IO redirection

p4.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <sys/wait.h>

int
main(int argc, char *argv[]){
    int rc = fork();
    if (rc < 0) {   // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child: redirect standard output to a file
        close(STDOUT_FILENO);
        open("./p4.output", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);
 // now exec "wc"...
        char *myargs[3];
        myargs[0] = strdup("wc");                  // program: "wc" (word count)
        myargs[1] = strdup("p4.c");      // argument: file to count
        myargs[2] = NULL;                // marks end of array
        execvp(myargs[0], myargs);       // runs word count
    } else {                             // parent goes down this path (main)
        int wc = wait(NULL);
    }
    return 0;
}
```

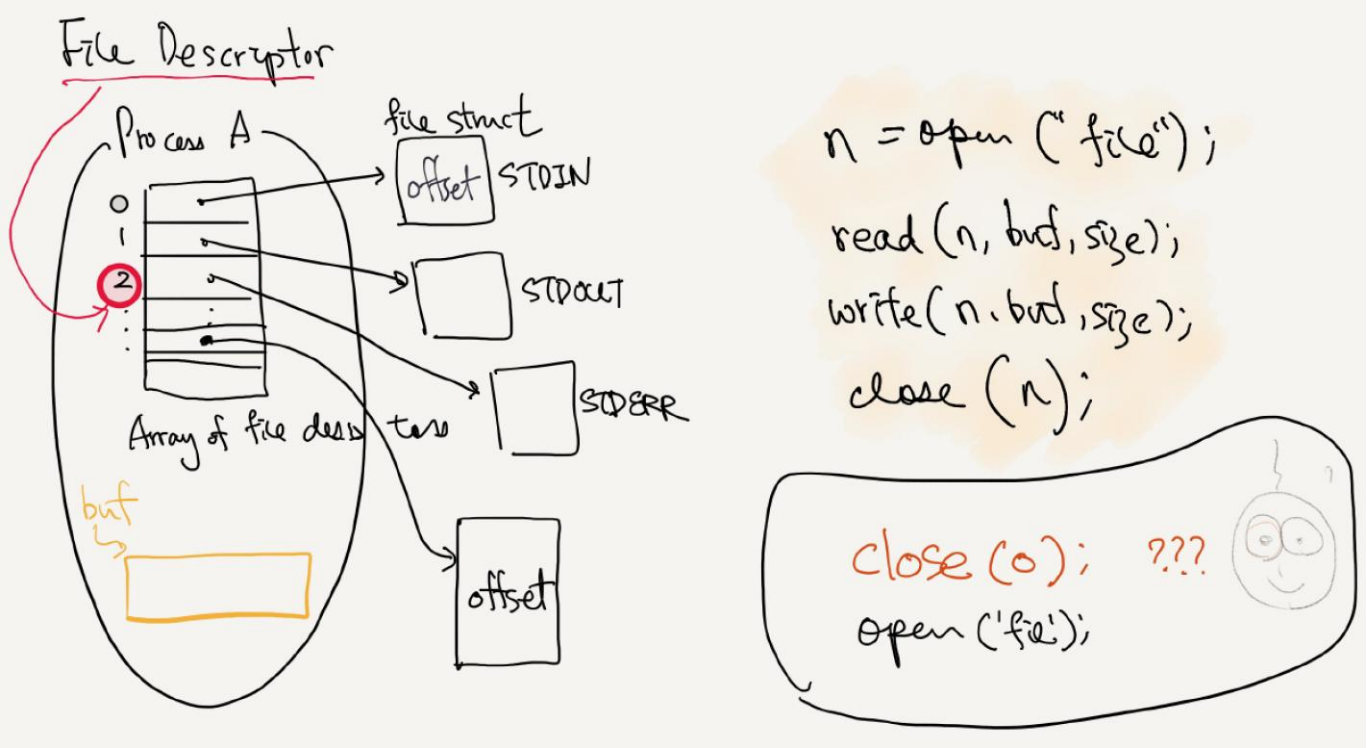
IO redirection

Result

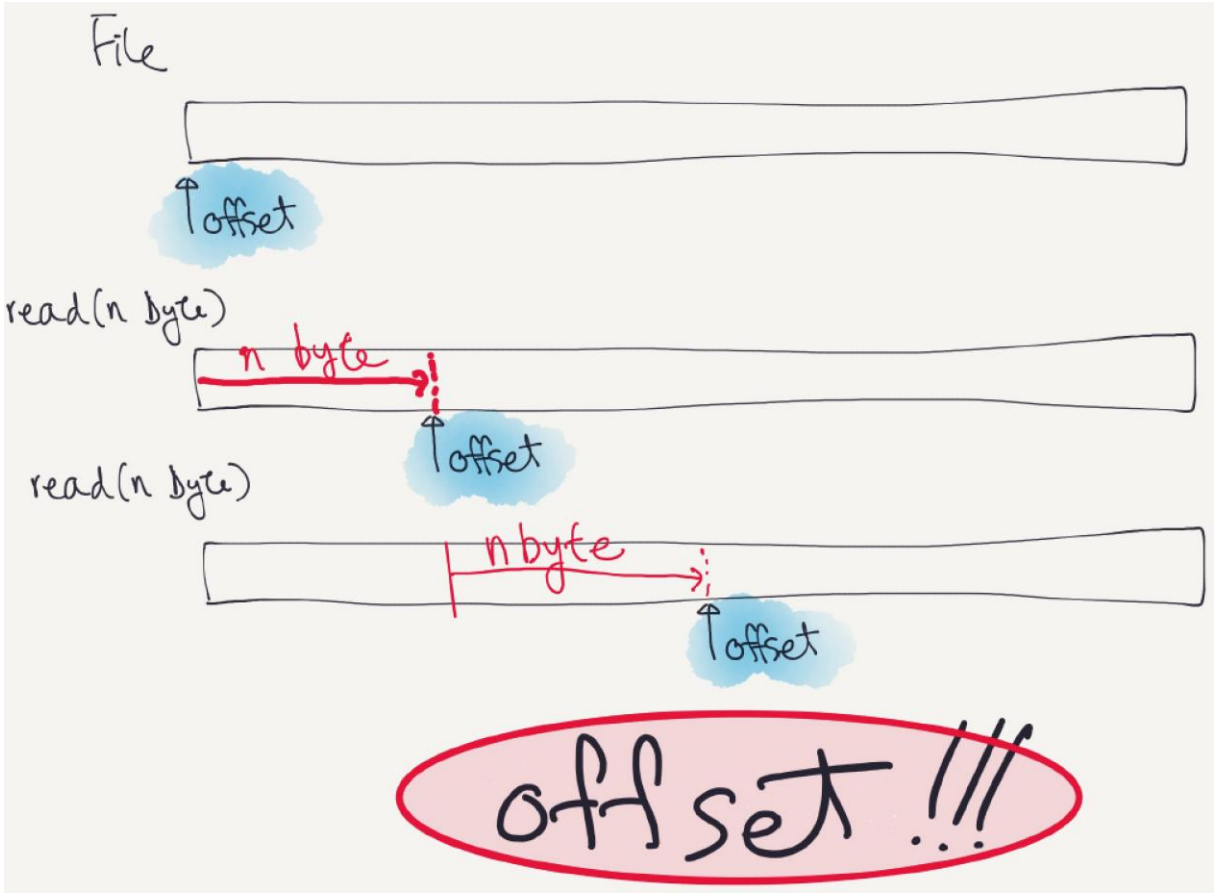
```
prompt> ./p4
prompt> cat p4.output
32 109 846 p4.c
prompt>
```

File descriptor and file descriptor table

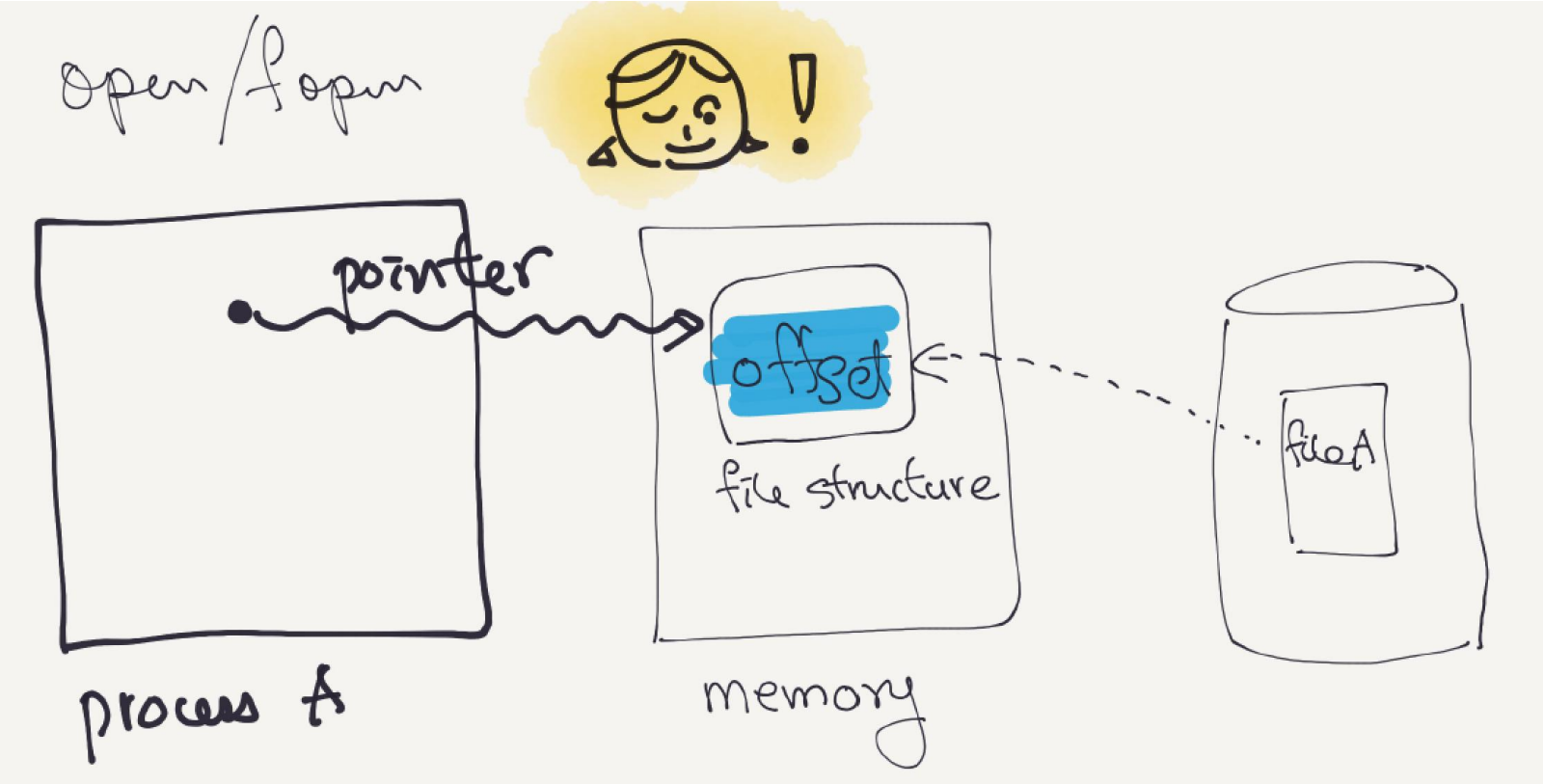
- File descriptor
  - an integer that represents a file, a pipe, a directory and a device
  - A process uses a file descriptor to open a file and directory.
  - each process has its own file descriptor table.
  - File descriptor 0 (Standard Input), 1 (Standard Output), 2 (Standard Error)



File offset


File
offset
read(n byte)
n byte
offset
read(n byte)
n byte
offset
offset !!!

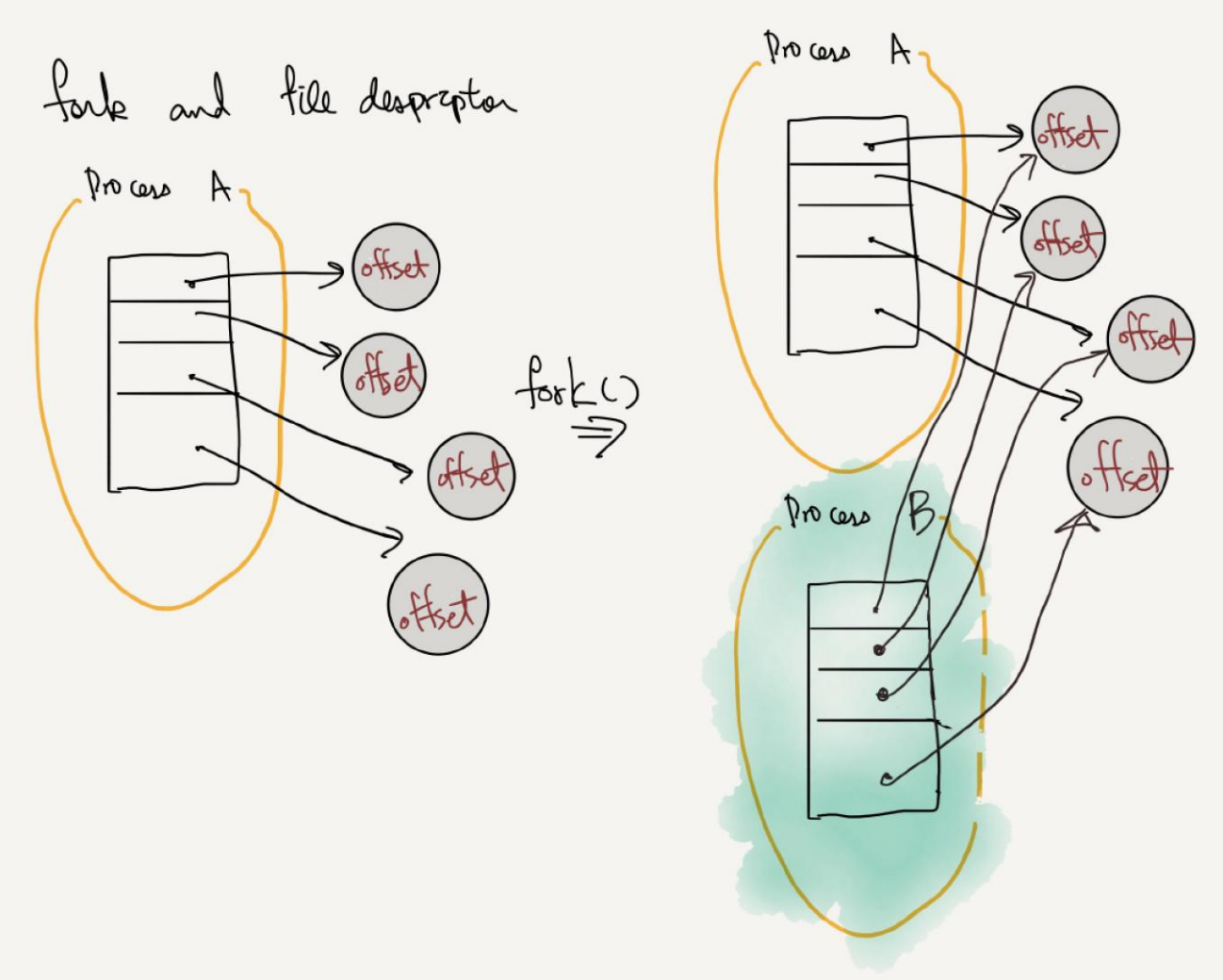
File structure (`struct file` in Linux)



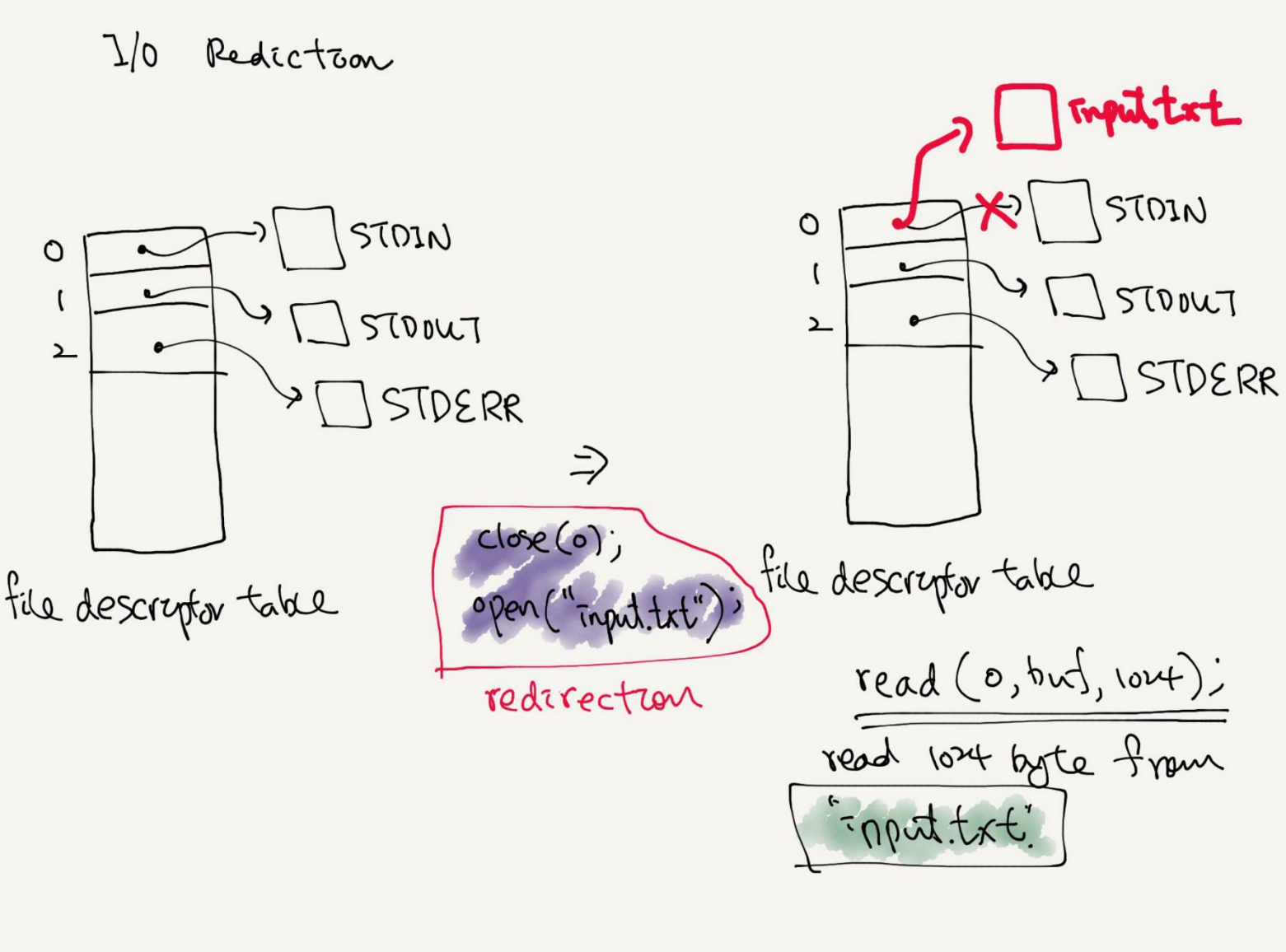
file descriptor and system calls

- `open()`
  - Allocate a new file object, allocate new file descriptor and set the newly allocated file descript or to point to the new file object.
  - When allocating the new file descriptor, it uses the smallest ‘free’ file descriptor from the file descriptor table.
- `close()`
  - deallocate the file descriptor ‘fd’.
  - Deallocate the file object if there is no file descriptor associated with it.
- `fork()`
  - copies the file descriptor table from the parent to child process.
- `exec()`
  - retains the file descriptor table.

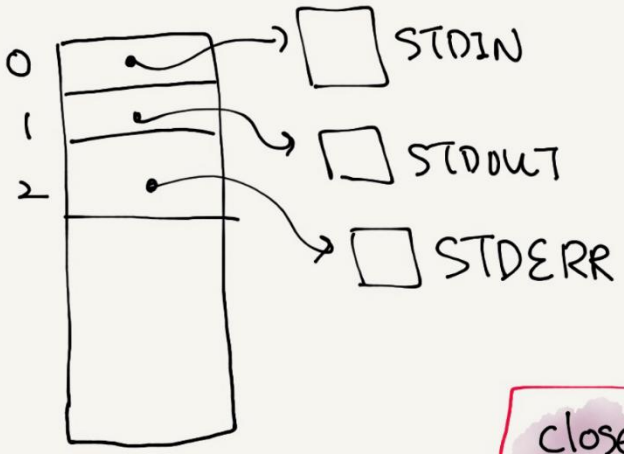
`fork()` and file descriptors




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


I/O Redirection
0
1
2
STDIN
STDOUT
STDERR
file descriptor table
close(0);
open("input.txt");
redirection
input.txt
0
1
2
STDIN
STDOUT
STDERR
file descriptor table
read(0, buf, 1024);
read 1024 byte from
"input.txt"

I/O Redirection

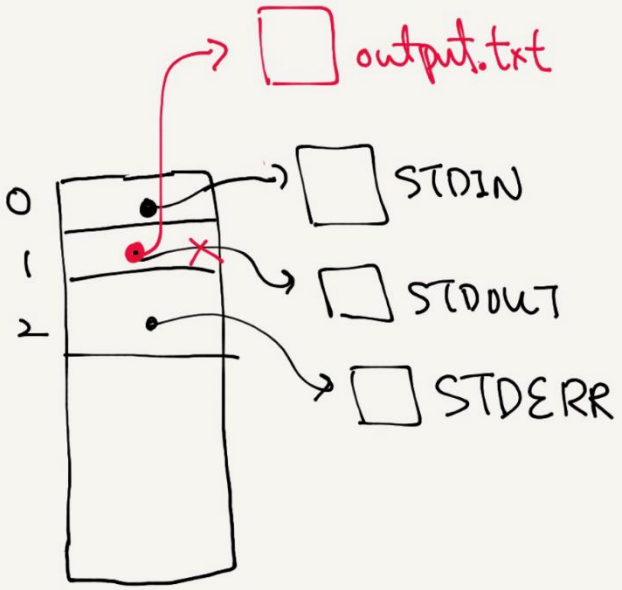


file descriptor table

⇒

```
close(1);
open("output.txt");
```

redirection



file descriptor table

```
write(1, buf, 1024);
```

write the content of buf to "output.txt".

cat and IO redirection

```
% cat input.txt
```

```
char *argv[2];

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

```
char buf[512];
int n;

for(;;) {
  n = read(0, buf, sizeof(buf));
  if(n == 0)
    break;
  if(n < 0) {
    fprintf(2, "read error\n");
    exit();
  }
  if(write(1, buf, n) != n) {
    fprintf(2, "write error\n");
    exit();
  }
}
```

Standard Input

Standard Error

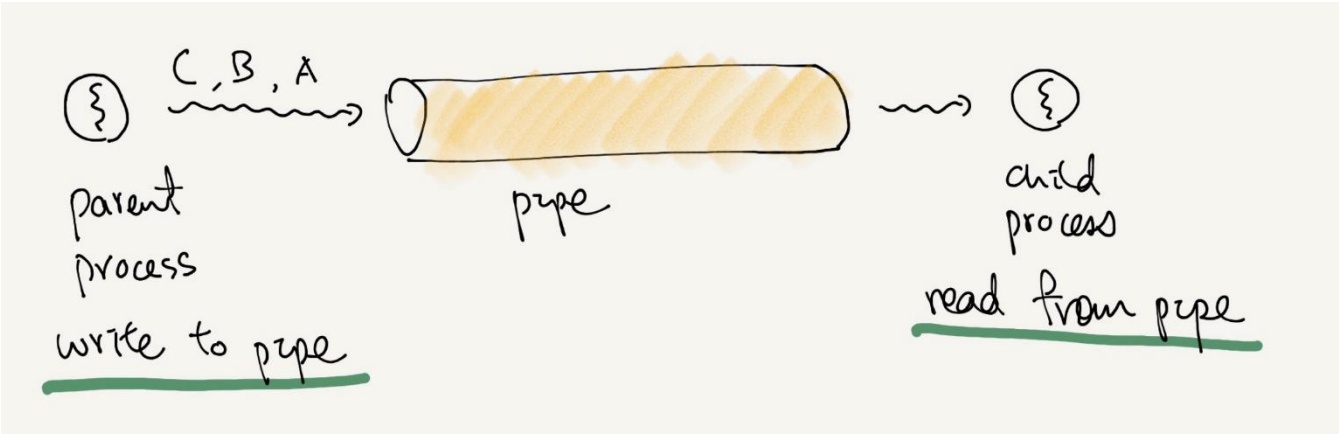
Standard Output

pipe: '|'

```
% echo hello world | wc
```

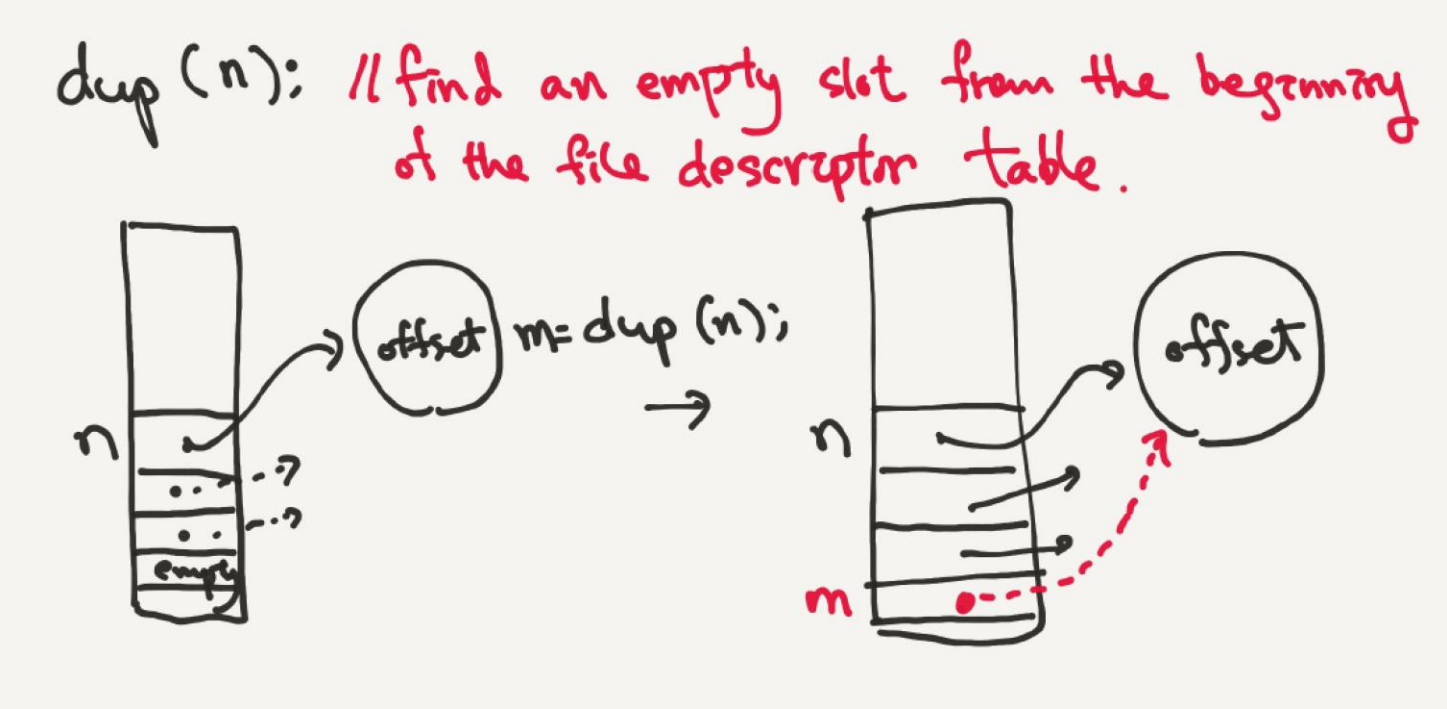
pipe

- Output to STDOUT of one process is fed to STDIN of another process.
- Implemented with dup() and pipe().
- Key innovation of UNIX shell.



dup(fd)

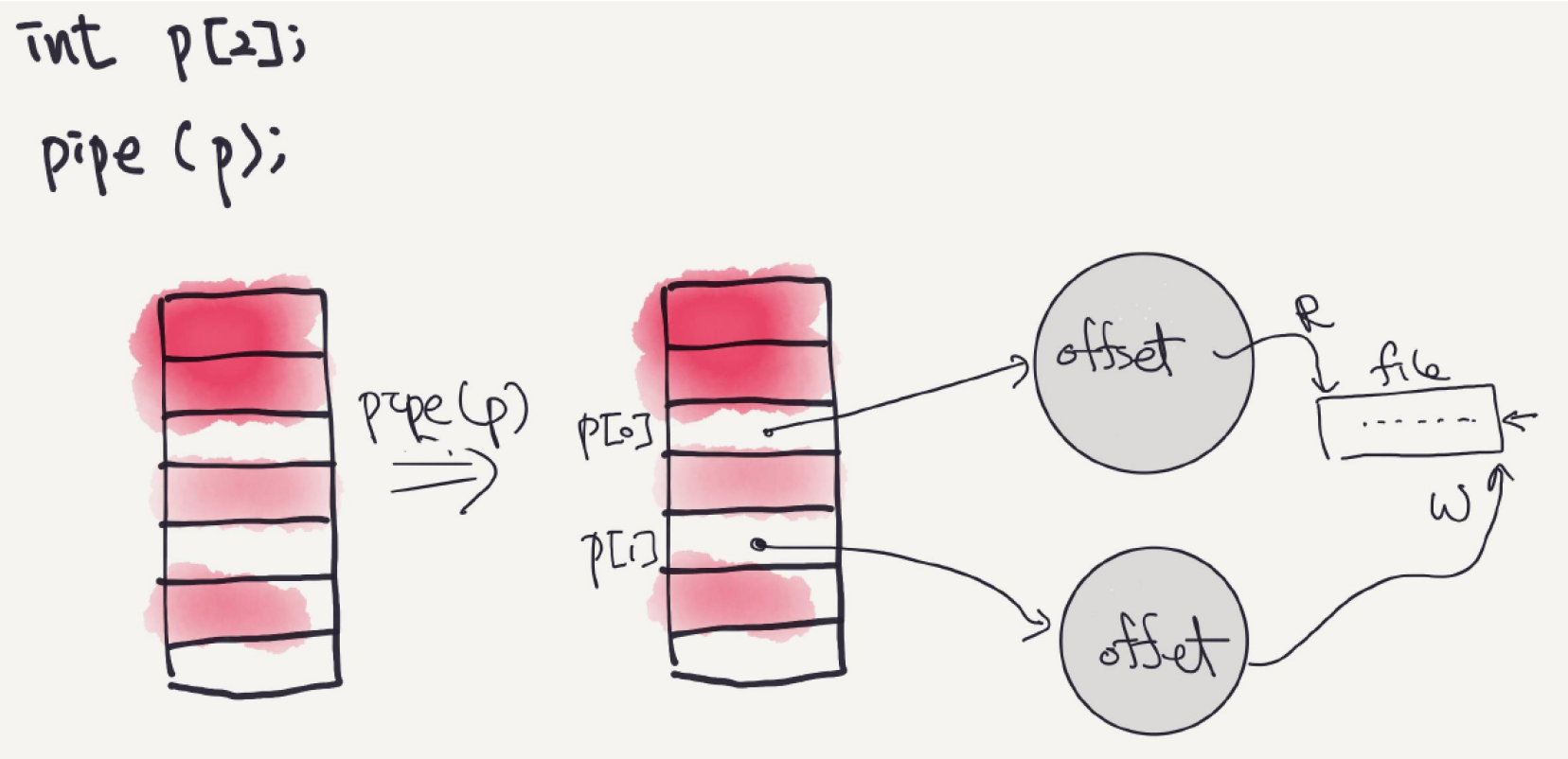
- duplicate file descriptor: dup() system call



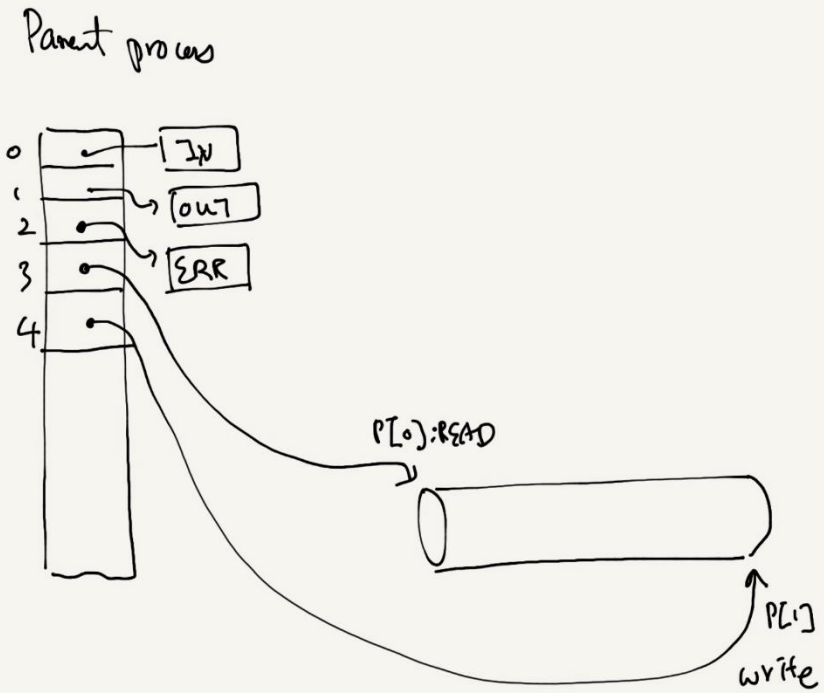

```
fd = dup(1);
write(1, "hello ", 6);
write(fd, "world\n", 6);
```

pipe()

- special type of file, a kernel buffer that is exposed to a process via a pair of file de scriptors: p[0] for read end and p[1] for write end.
- The reader blocks when there is no data to read.



pipe



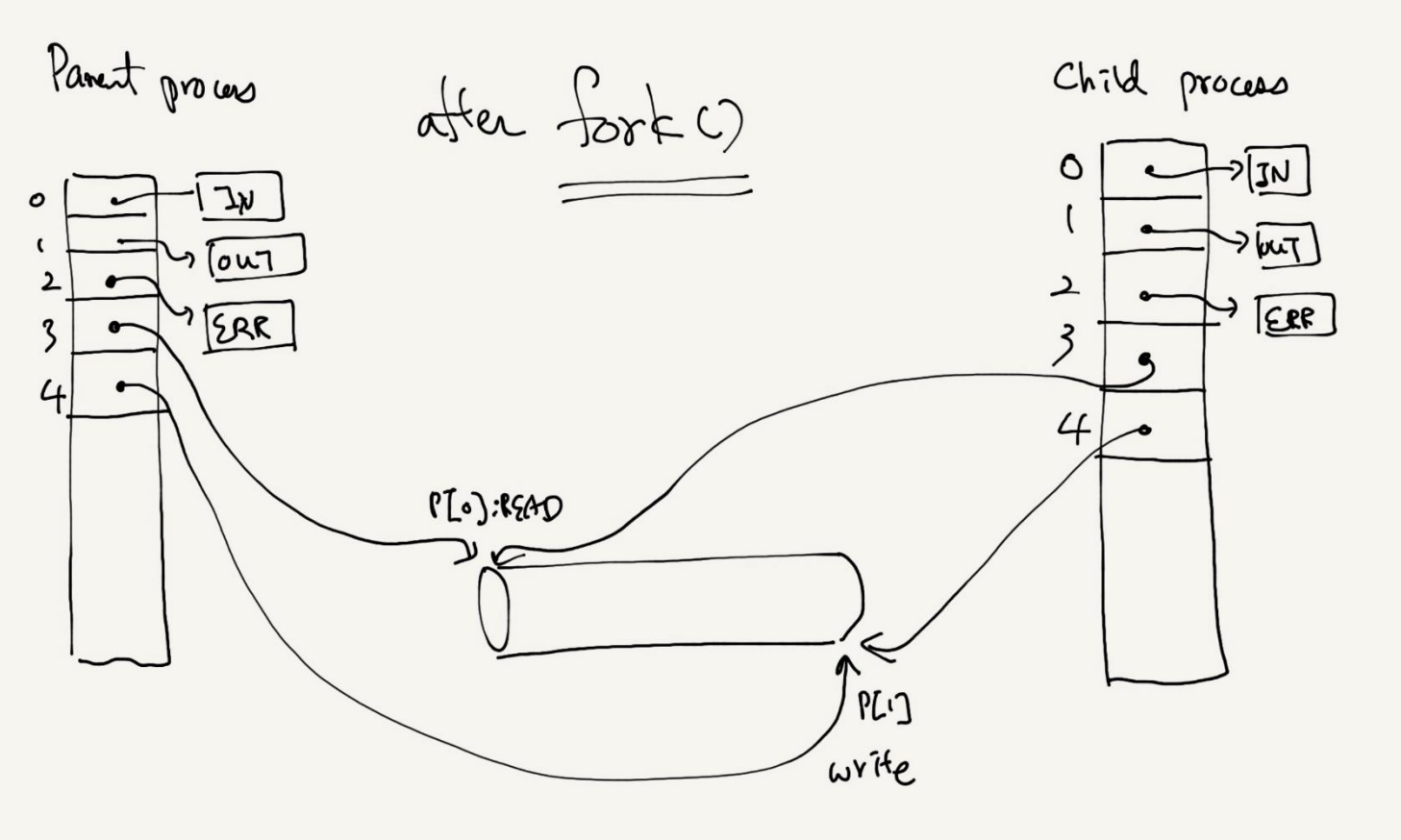

```
int p[2];
char *argv[2];

argv[0] = "wc";
argv[1] = 0;

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

```
% echo hello world | wc
```

pipe




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

argv[0] = "wc";
argv[1] = 0;

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

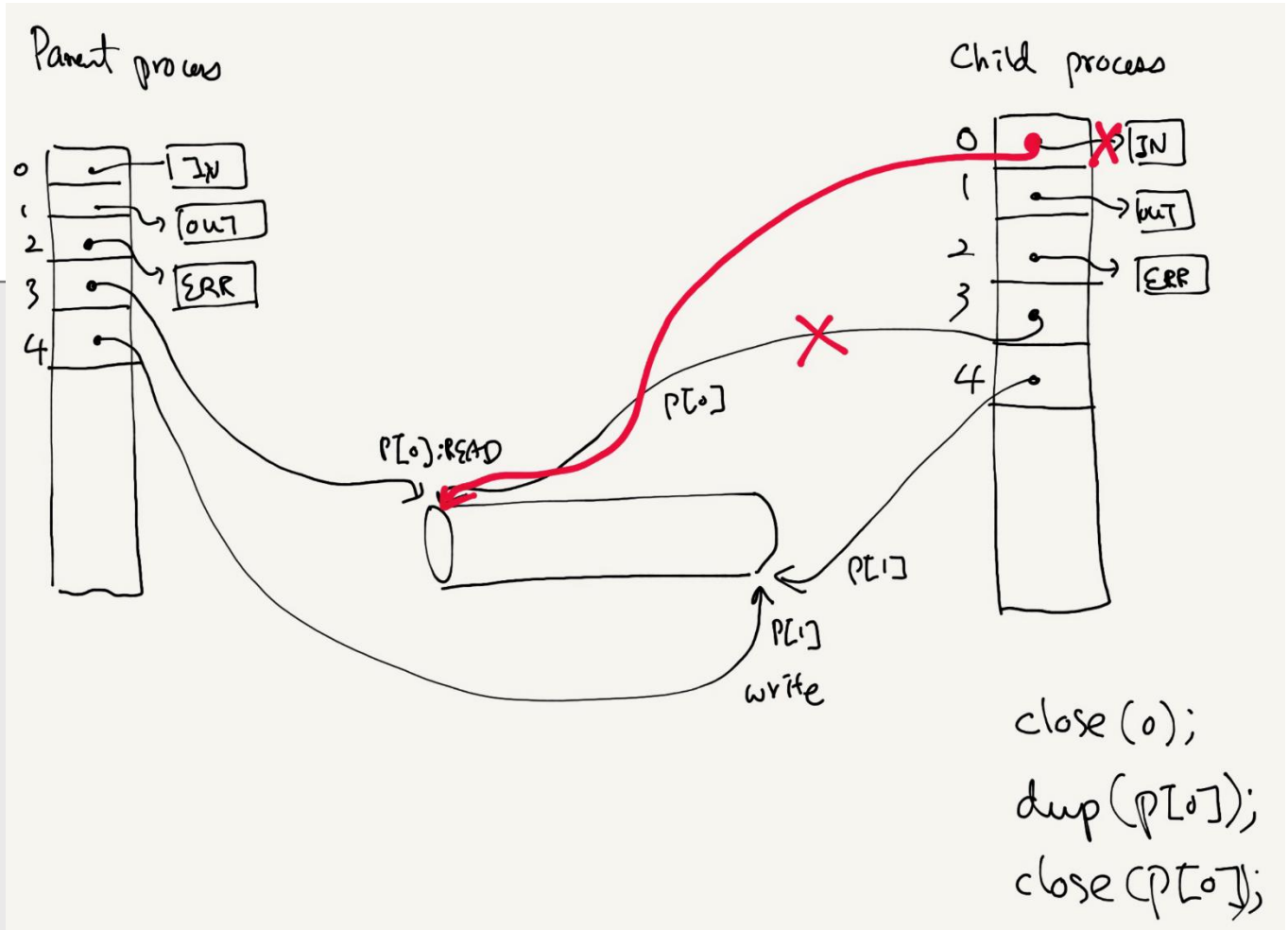
```
% echo hello world | wc
```

pipe

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

argv[0] = "wc";
argv[1] = 0;

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



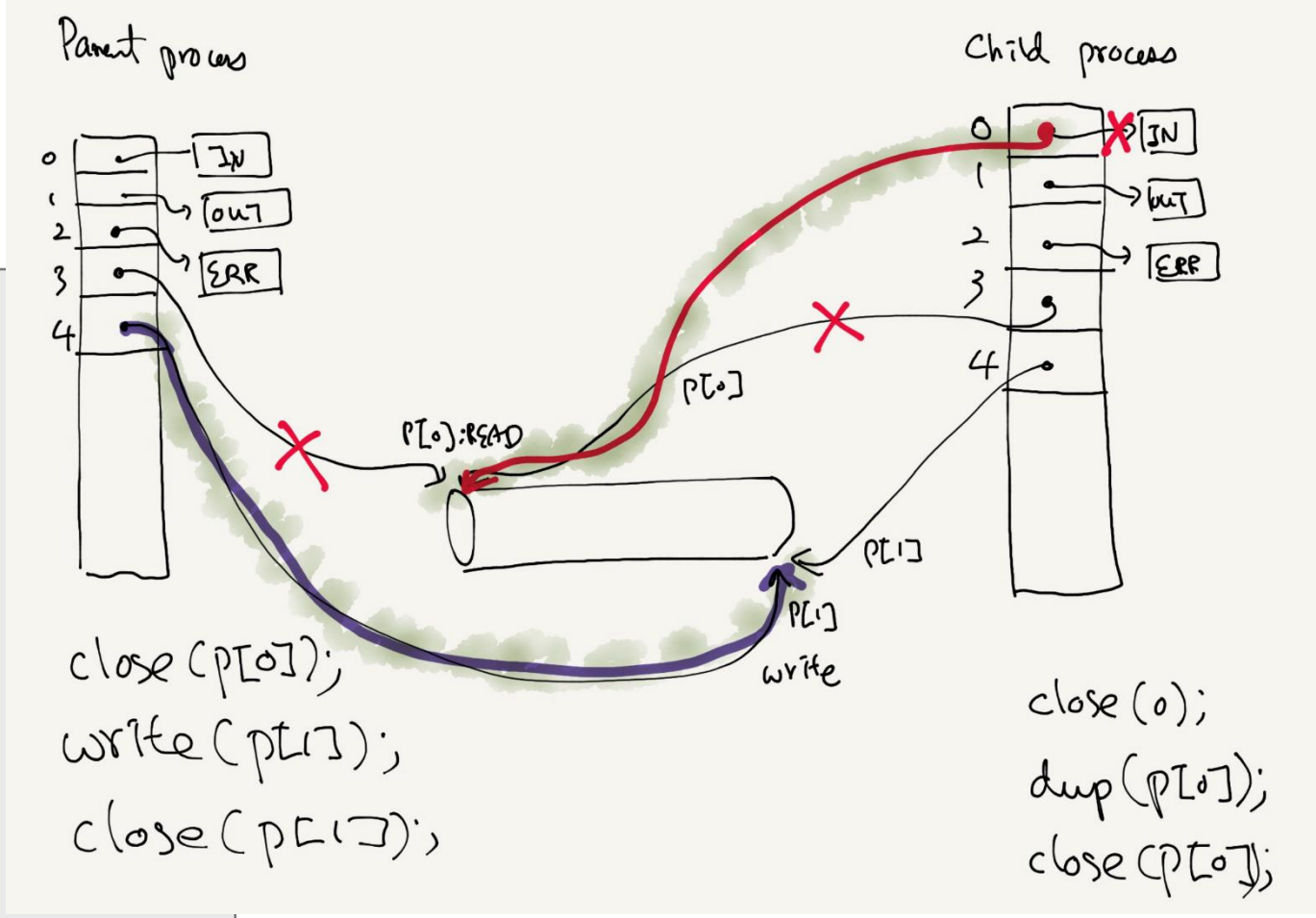

```
% echo hello world | wc
```

pipe

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

argv[0] = "wc";
argv[1] = 0;

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




```
% echo hello world | wc
```

pipe vs. IO redirection

- advantages of pipes over using redirection with temporary files

```
echo hello world | wc
```

vs.

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

  - pipe automatically clean themselves up. When using temporary file, the user has t o explicitly delete it.
  - pipe can pass arbitrarily long data while file redirection requires sufficient availabl e disk space.
  - In pipe, reader and write can proceed in parallel while in redirection, the one has t o finish for the others to start.

summary

- process
- Process API
  - `fork()`
  - `exec()`
  - `wait()`
- separation of `fork()` and `exec()`
  - IO redirection
  - pipe