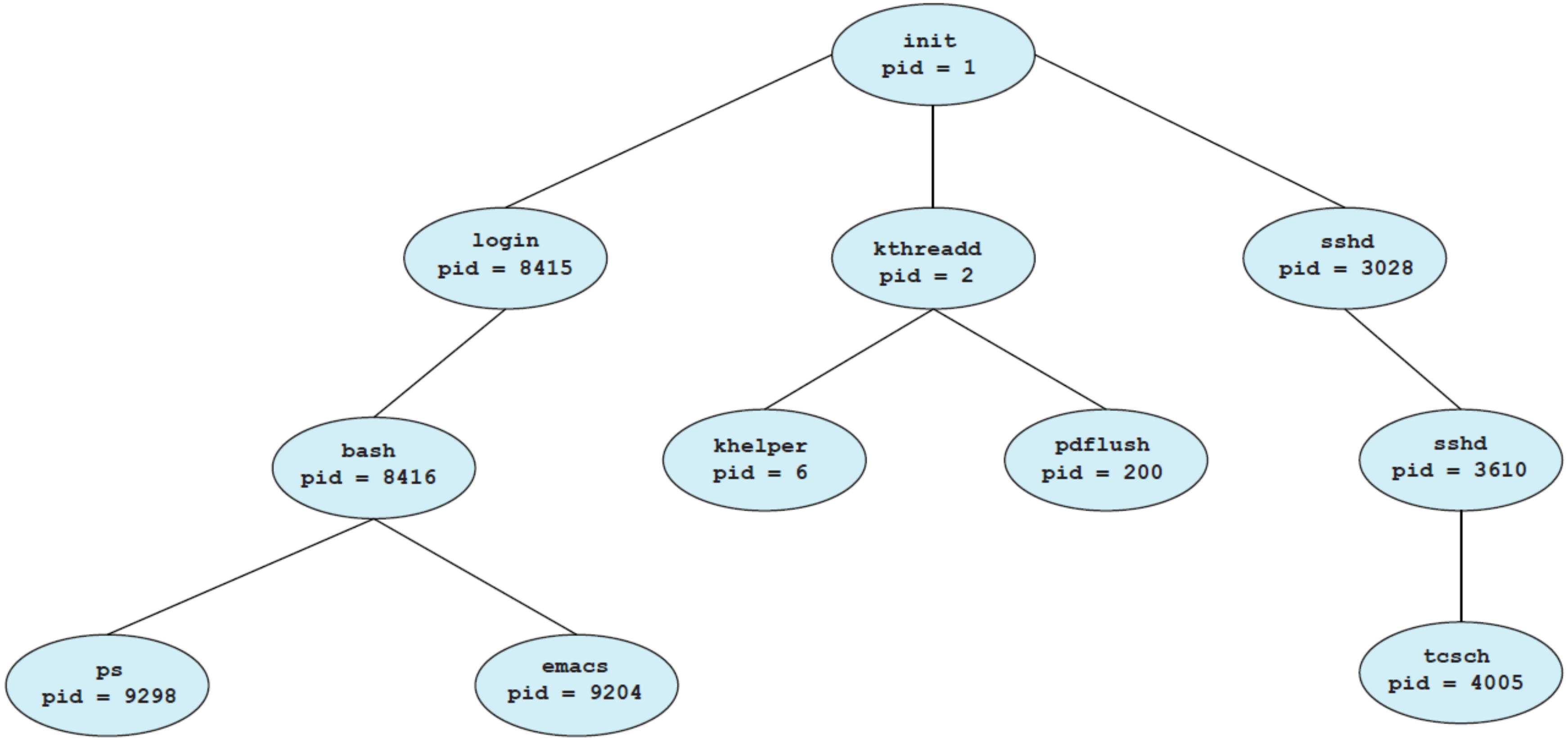
Operating Systems

The Processes API

These slides include content from the work of:
Youjip Won, KIAT OS Lab

Tree of Processes



`pstree`

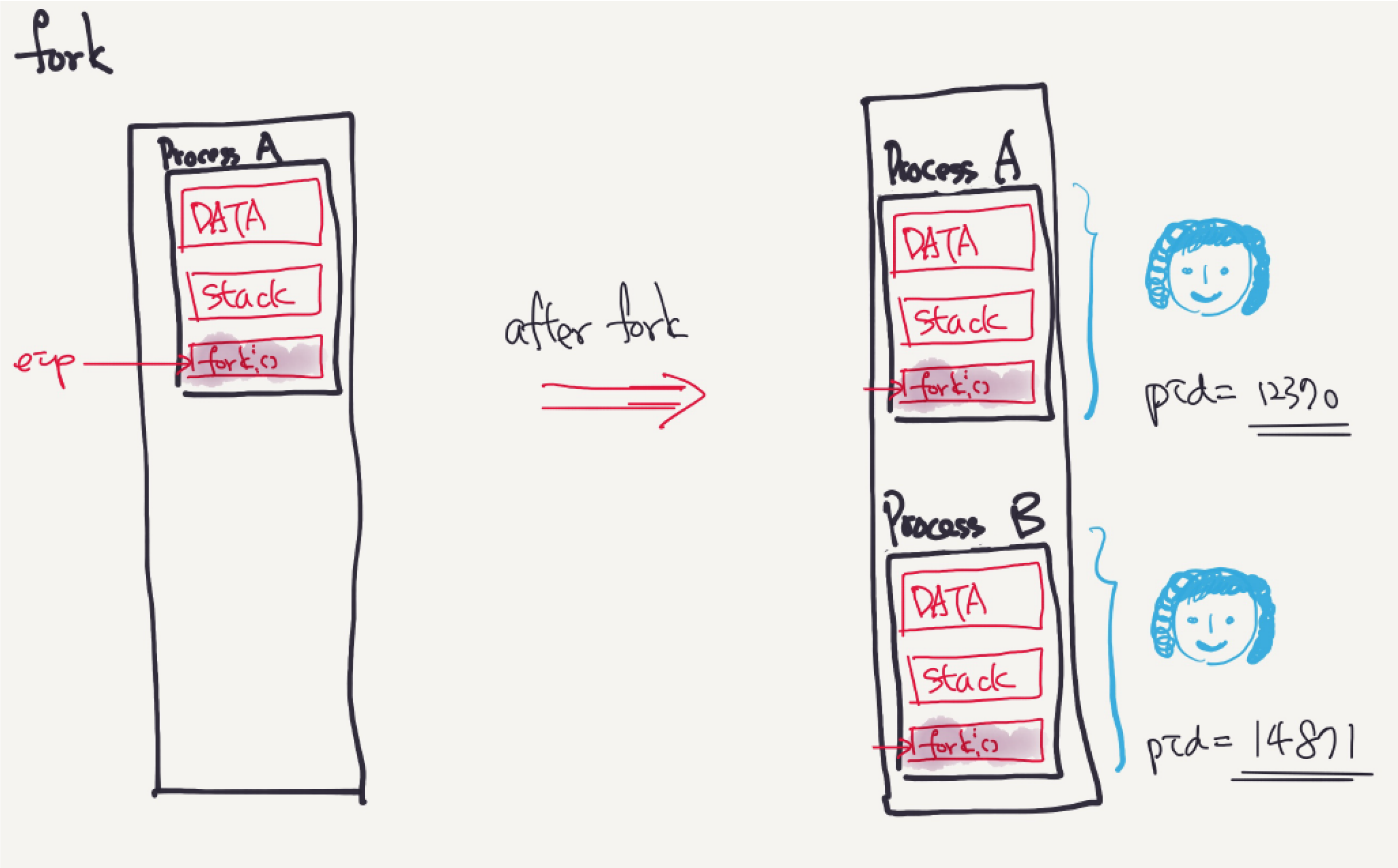
Overview

- System call is the services provided by OS kernel.
- In C programming, it often uses functions defined in libc which provides a wrapper for many system calls.
- `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 process 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.


fork
Process A
DATA
stack
fork()
eip
after fork
Process A
DATA
stack
fork()
pid= 12370
Process B
DATA
stack
fork()
pid= 14871

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;
}
```

The child doesn't start running at main; it just comes into life as if it had called fork() itself.

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 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

```
#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;
}
```

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
  - `% echo “hello”`
- OS needs to load code (and static data) from that executable and overwrites the current code segment (and current static data) with it
- It initializes a new stack, initializes a new heap for the new program.
- two parameters
  - The name of the binary file
  - The array of arguments

```
char *argv[3];

argv[0] = “echo”;
argv[1] = “hello”;
argv[2] = 0;
execvp(“/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] = "wc"; // program: "wc":the number of lines, words, and bytes
        myargs[1] = "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;
}
```

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>
```

how many lines, words, and bytes are found in the file

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.

Process of command execution in shell:

The shell is just a user program. It shows you a **prompt** and then waits for you to type something into it. You then type a command (i.e., the name of an executable program, plus any arguments) into it;

- the shell then calls `fork()` to create a new child process to run the command,
- calls some variant of `exec()` to run the command,
- then waits for the command to complete by calling `wait()`
- When the child completes, the shell returns from `wait()`
- prints out a prompt again, ready for your next command

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 redirection and pipe possible**.

  - IO redirection

```
% wc p4.c > newfile.txt
```

  - pipe

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

'pipe' is the heart of the shell programming.

IO redirection

```
% wc p4.c > p4.output
```

- Save the result of ’`wc p4.c`‘ to `p4.output`.
- How?
- The shell is a program that uses `fork()` to create a new process and `exec()` to execute a command with arguments.
- Before calling `exec(“wc”, “wc p4.c”)`, the child process closes `STDOUT (close(1))` and opens `newfile.txt` (`open(“p4.output)`).

Details of IO redirection

```
% wc p4.c > p4.output
```

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] = "wc";           // program: "wc" (word count)
        myargs[1] = "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)

Consider a process that has opened three files (`stdin`, `stdout`, `stderr`), and additionally opened `file.txt` and `log.txt`. The file descriptor table might look like this:

File Descriptor	Description	Associated File
0	Standard Input	Keyboard Input
1	Standard Output	Console Output
2	Standard Error	Console Error
3	File	`file.txt`
4	File	`log.txt`

file descriptor and system calls

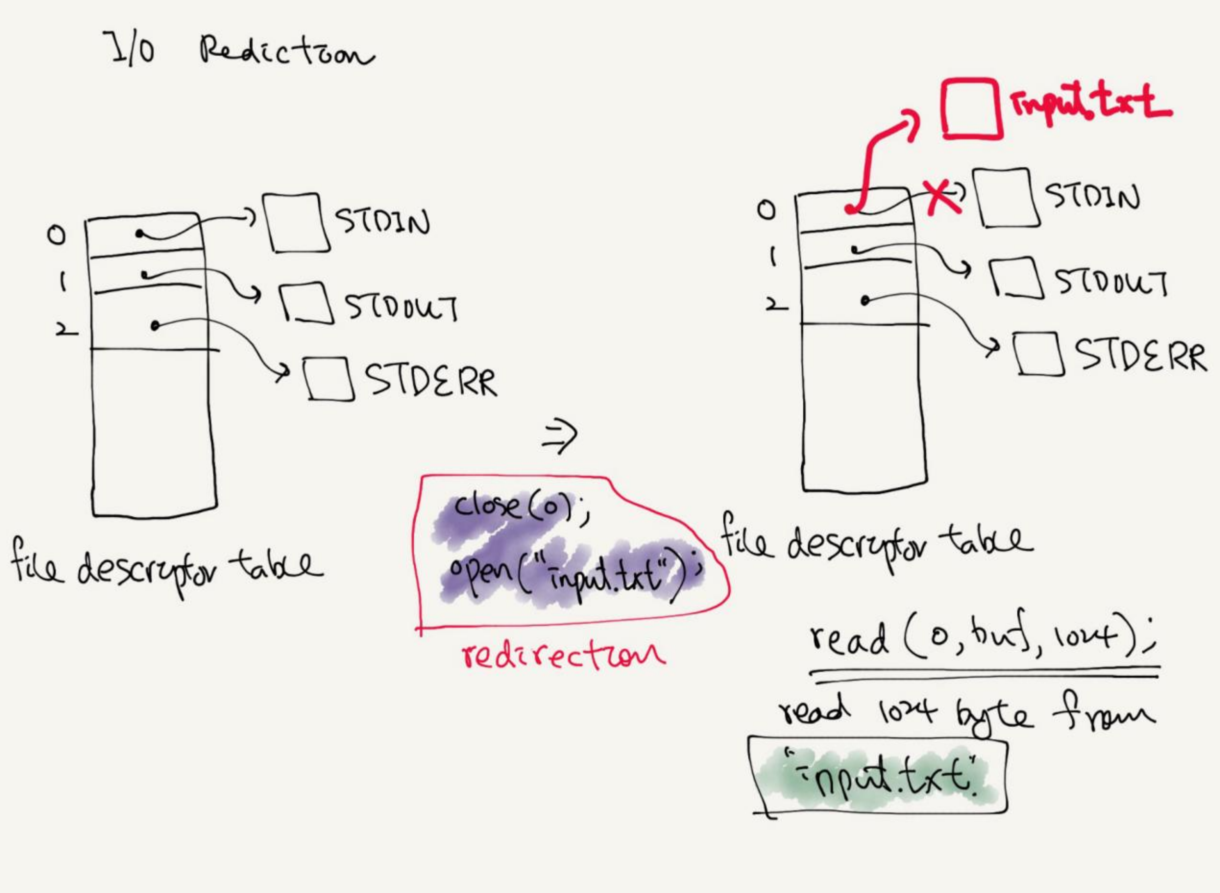
- `open()`
  - Allocate a new file object, allocate new file descriptor and set the newly allocated file descriptor to point to the new file object.
  - When allocating the new file descriptor, it uses the smallest ‘free’ file descriptor from the file descriptor table.
- `close()`
  - deallocate the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

`fork()` and file descriptors

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

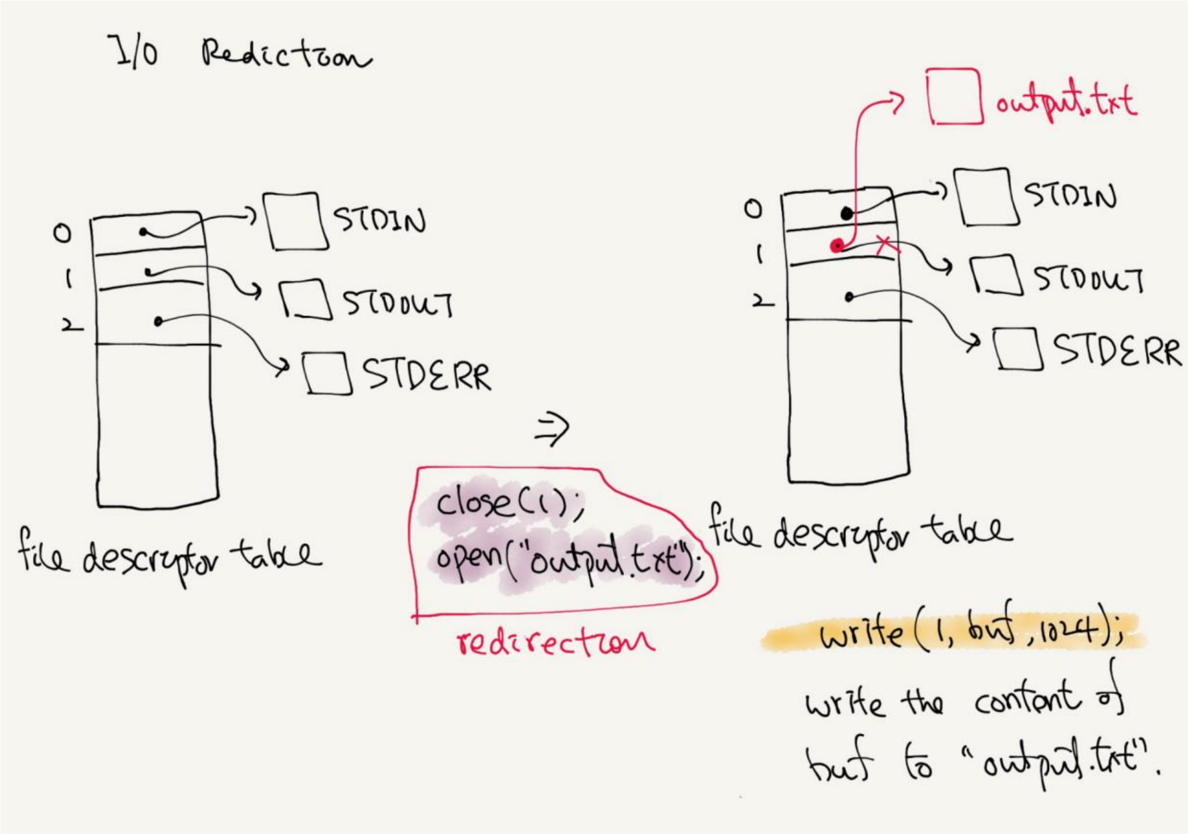
I/O redirection

- Input redirection:
  - close (0)
  - open(“input.txt”)
  - read(0,buff,1024)



I/O redirection

- Output redirection:
  - close (1)
  - open("output.txt")
  - write(1,buff,1024)



Example： `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);
}
```

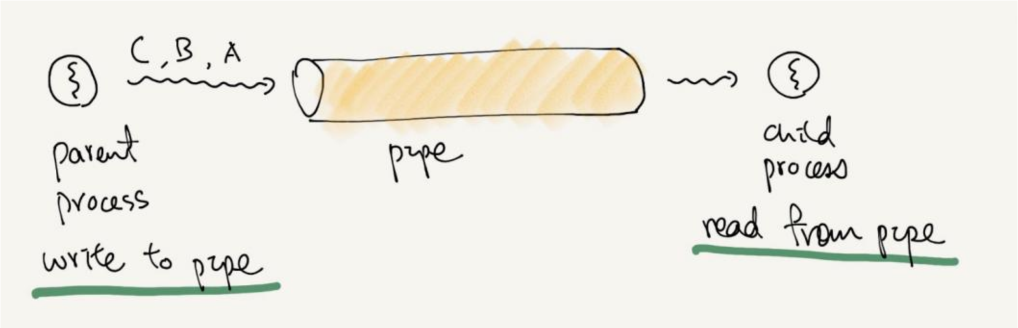
Without any file arguments, `cat` reads from the standard input (stdin)

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.
- Parent process writes to pipe
- Child process reads from pipe



dup(fd)

- duplicate file descriptor: dup() system call
- n = dup (fd) // find an empty slot (n) from the beginning of the descriptor table and duplicate the fd file desciptor

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

pipe()

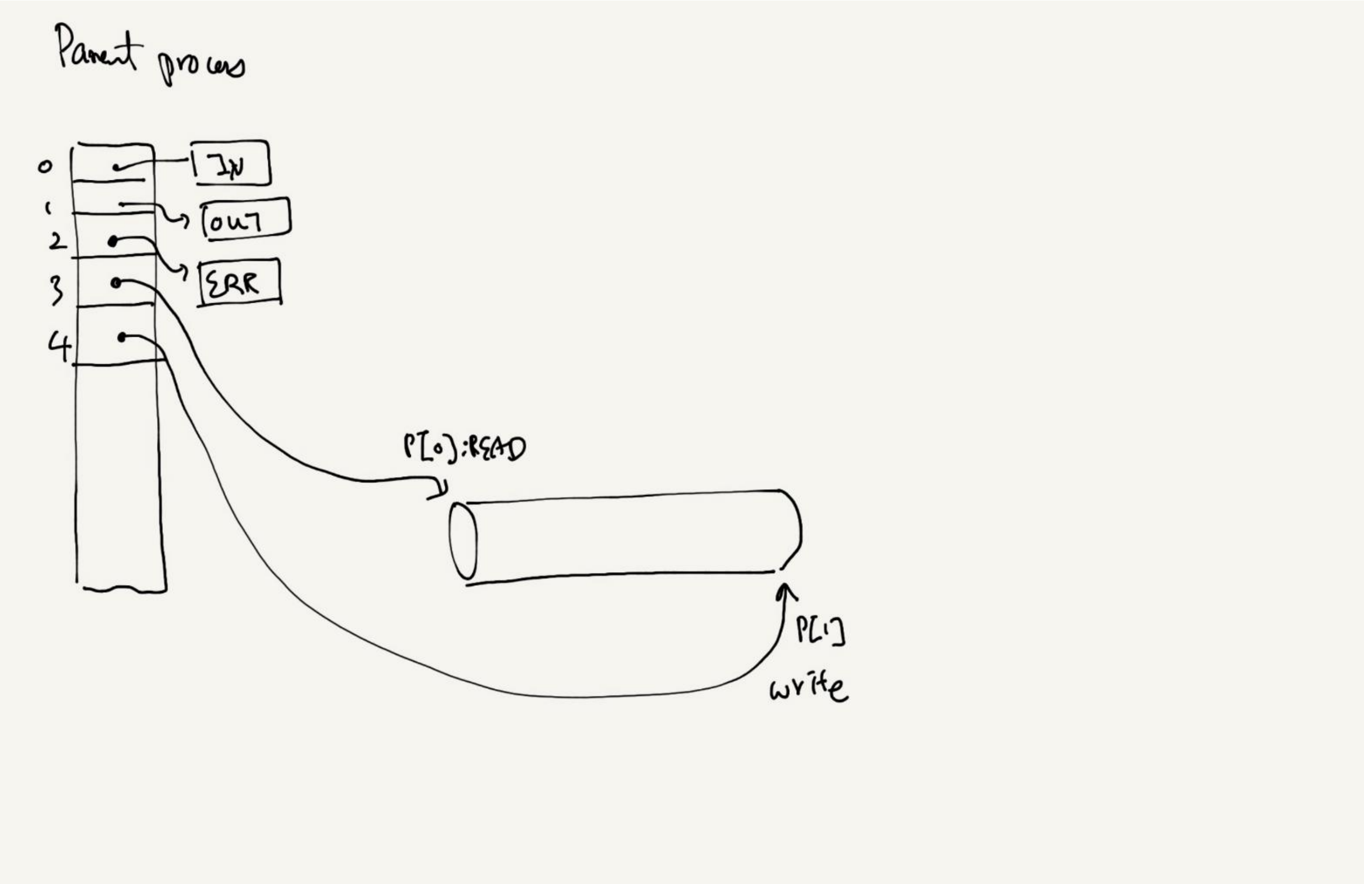
- special type of file, a kernel buffer that is exposed to a process via a pair of file descriptors: p[0] for read end and p[1] for write end.
- The reader blocks when there is no data to read.
- `int p[2];`
- `pipe (p);`

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]);
}
```



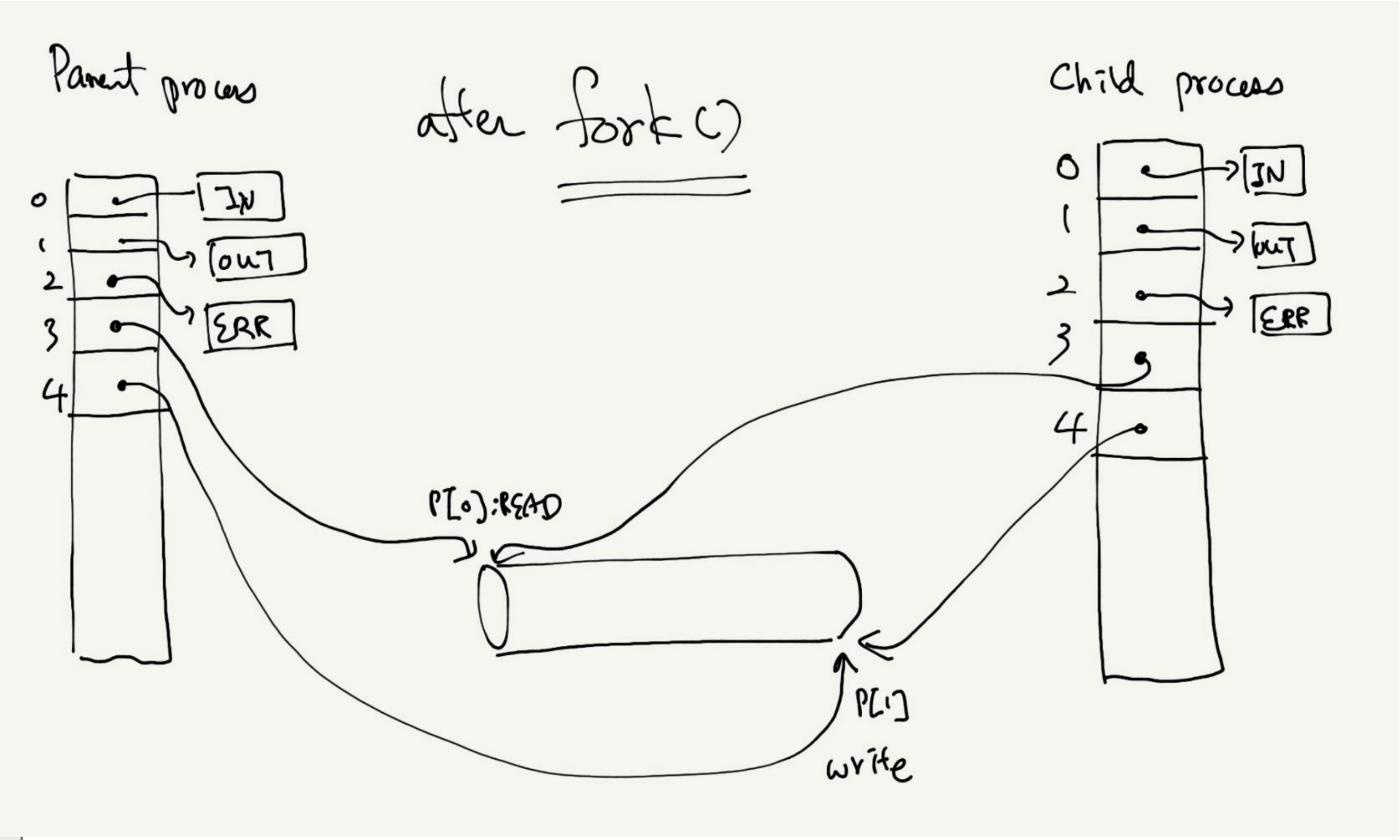

```
% echo hello world | wc
```

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]);
}
```



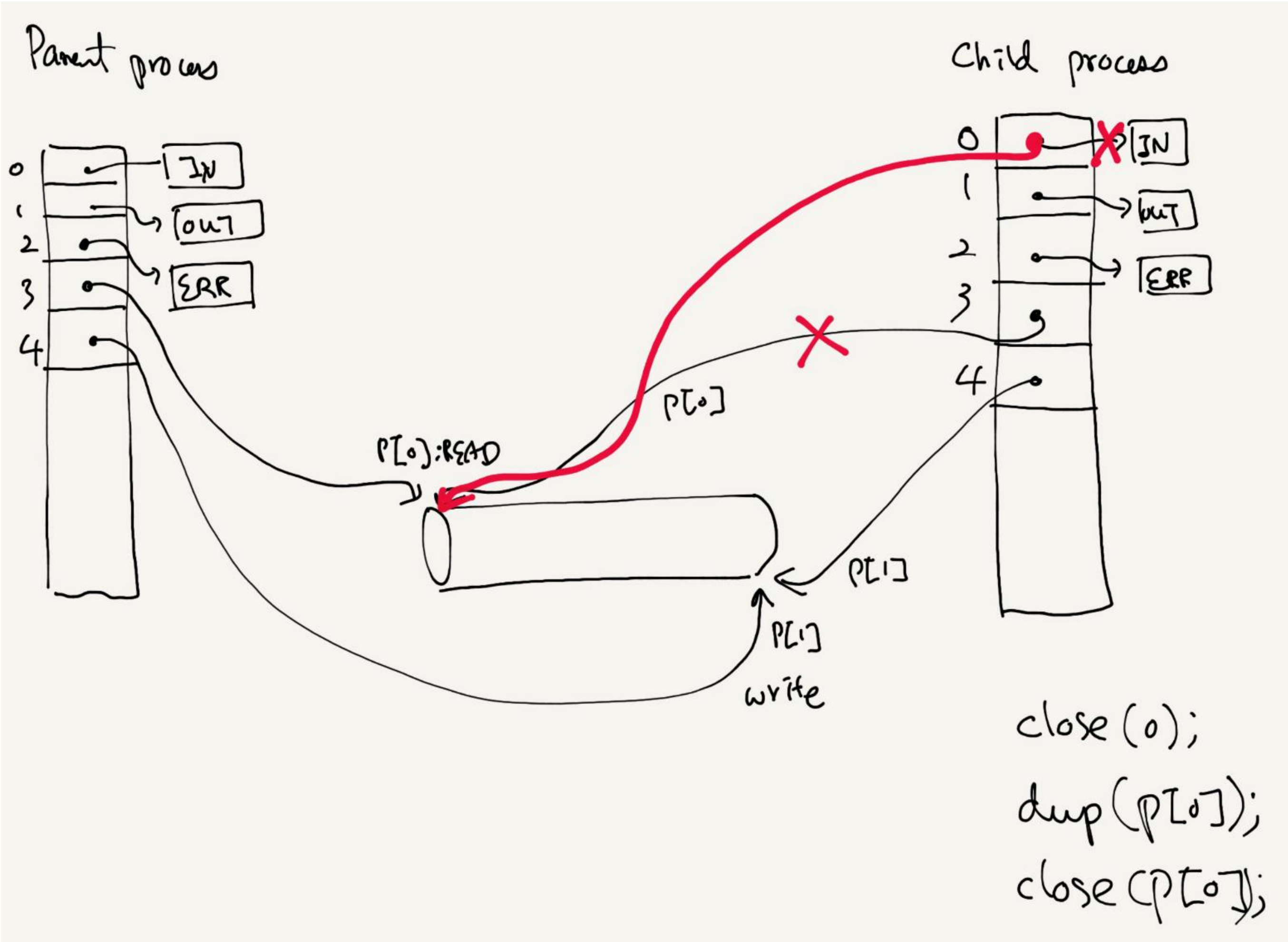

```
% echo hello world | wc
```

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]);
}
```



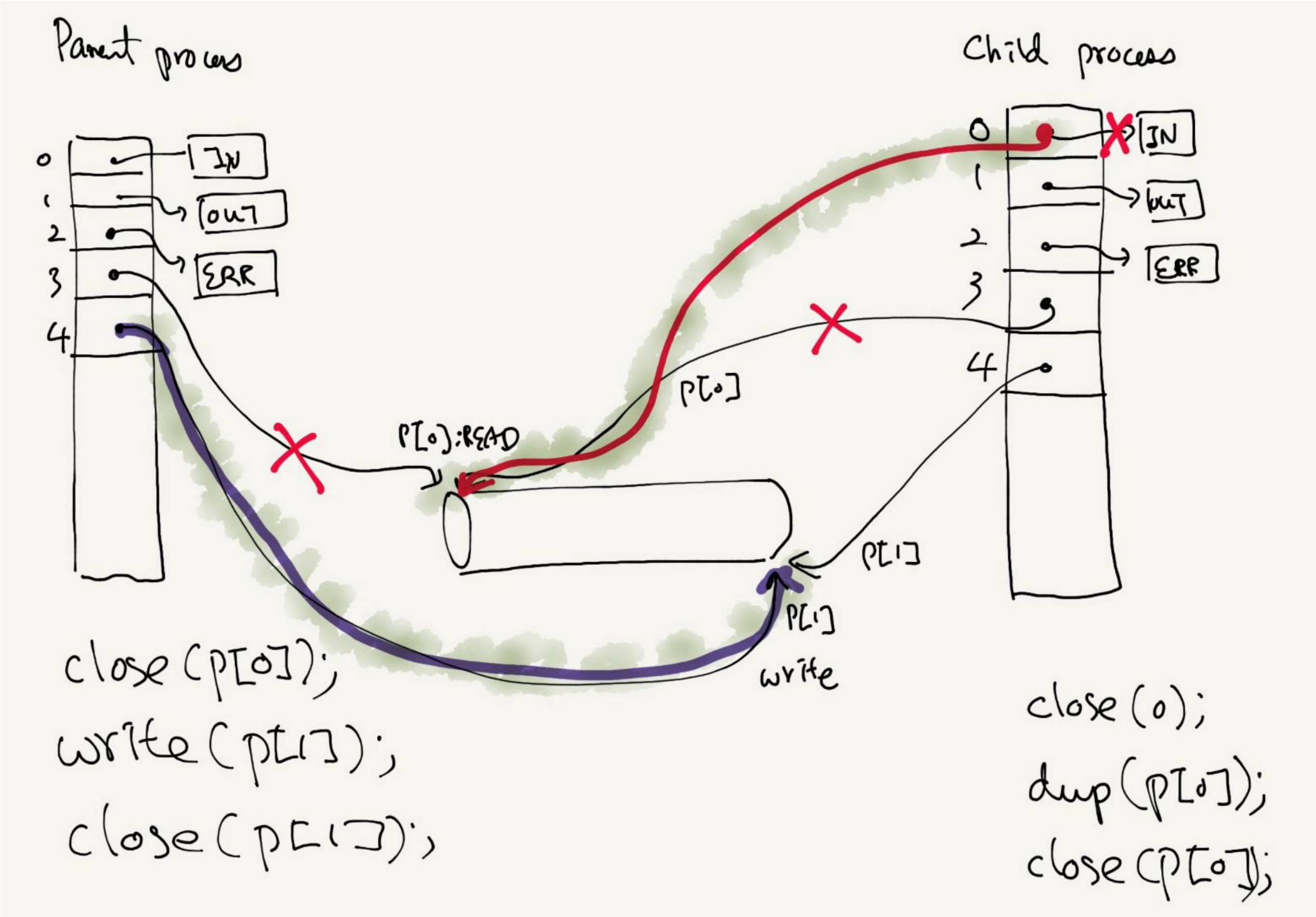

```
% echo hello world | wc
```

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
```