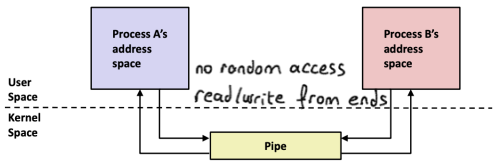
pipes

unnamed pipes:



system call → fd[2] → those are open files in open file table

`int pipe(int * fd);` ↳ returns -1 if error

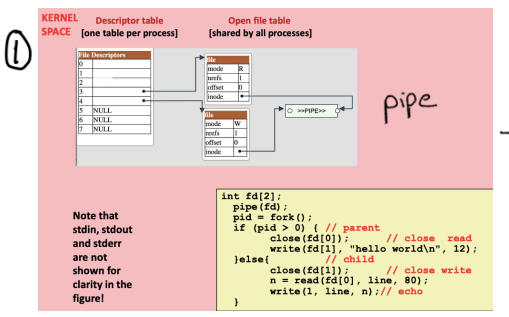
- `fd[0]` → open for reading
- `fd[1]` → open for writing

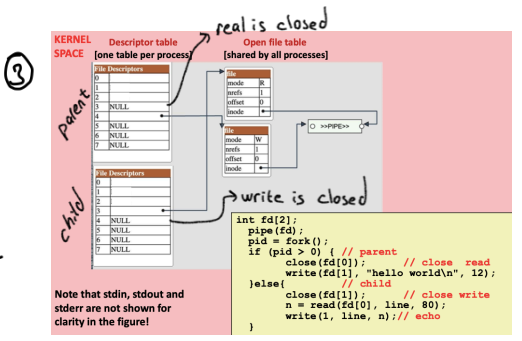
- used for communication between parent and child

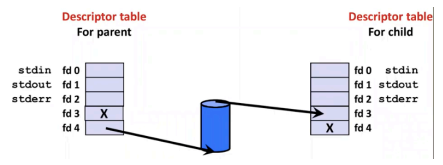
```
#include <unistd.h>
#include <stdio.h>
int main(void){
  int n;                  // to keep track of num bytes read
  int fd[2];              // to hold fds of both ends of pipe
  pid_t pid;              // pid of child process
  char line[80];          // buffer to hold text read/written
if (pipe(fd) < 0)                    // create the pipe
       perror("pipe error");

if ((pid = fork()) < 0) {            // fork off a child
       perror("fork error");
  } else if (pid > 0) {                 // parent process
       close(fd[0]);                   // close  read end
       write(fd[1], "hello world\n", 12); // write to  it
  }else {                               // child process
       close(fd[1]);                   // close write end
       n = read(fd[0], line, 80);      // read from  pipe
       write(1, line, n);              // echo  to  screen
  }
  exit(0);
}
```

parent (writes only) → child (reads only) → one directional







fd3 = fd[0] read

fd4 = fd[1] write

- used in shell with redirection:

`cat /etc/passwd | wc -l` → 3 processes (shell + cat + wc)

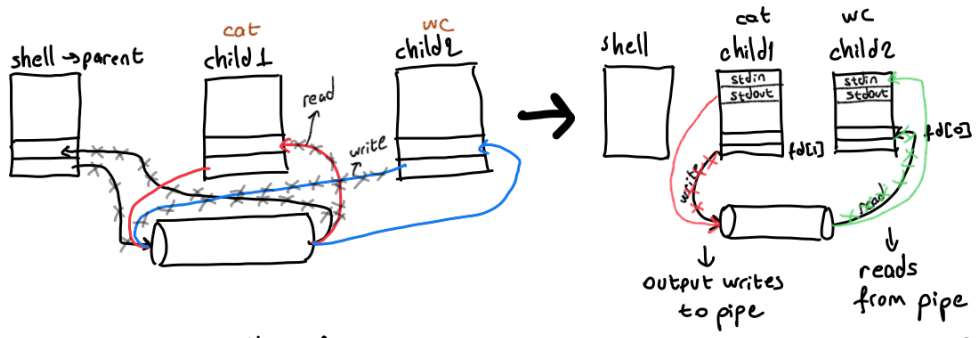
cat /etc/passwd: will display the user account info

wc -l: word count

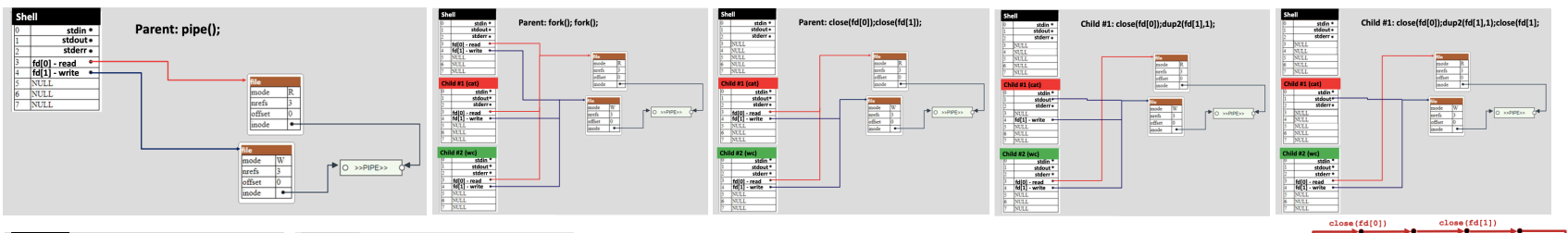
| → pipe → redirects the output

```
  int fd[2], c;              // to hold fds of both ends of pipe
  if (pipe(fd) < 0)  perror("pipe error"); // create the pipe
  if (fork()) {
     if (fork()) {           // parent process
       close(fd[0]);
       close(fd[1]);      // not going to use it
       wait(&c);
       wait(&c);           // wait for both
     } else {                  // pipe reader child
       close(fd[1]);           // not using write end
       dup2(fd[0],0);          // redirect stdin to read end
       close(fd[0]);           // it is duplicated, close
       execl("/usr/bin/wc","wc","-l",NULL); // run binary
     }
  } else {
     close(fd[0]);             // not using read end
     dup2(fd[1],1);            // redirect stdout to write end
     close(fd[1]);             // it is duplicated, close
     execl("/bin/cat","cat","/etc/passwd",NULL);// run binary
  }
```

shell → parent; child1 → cat; child2 → wc



- wc waits naturally for an input to come, does not execute first



limitations: processes need to be relatives, they are temporary

- cant broadcast P → C1, C2 when one of children reads from pipe, it is gone → no distinguish btw several readers/writers
- no structure, byte stream only

fifo (named pipes):

they are persistent path on the file system, have name and permissions

- any process that knows the name and have permission can access
- they persist even there is no process that is connected to them

in shell →

- **First, create your pipes**
  - `$ mkfifo pipe1`
  - `$ mkfifo pipe2`
  - `$ mkfifo pipe3`
- **Then, attach a data source to your pipes**
  - `$ ls -l >> pipe1 &`
  - `$ cat myfile >> pipe2 &`
  - `$ who >> pipe3 &`
- **Then, read from the pipes with your reader process**
  - `$ cat < pipe1 | lpr`
  - `$ spell < pipe2`
  - `$ sort < pipe3`
- **Finally, delete your pipes**
  - `$ rm pipe[1-3]`

system call →

```
#include <stdio.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>

void child(char *path)
{
    int fd;
    char buf[] = "123456789";

    fd = open(path, O_WRONLY);
    write(fd, buf, sizeof(buf));
    printf("Child sends: %s\n", buf);
    close(fd);
}

void parent(char *path)
{
    int fd;
    char buf[512];

    fd = open(path, O_RDONLY);
    read(fd, buf, sizeof(buf));
    printf("Parent receives: %s\n", buf);
    close(fd);
}
```

```
int main()
{
    char *path = "/tmp/fifo";
    pid_t pid;

    setlinebuf(stdout);
    unlink(path);          // delete if exist
    mkfifo(path, 0600);    // mode

    pid = fork();
    if (pid == 0) {
        child(path);
    } else {
        parent(path);
    }
    return 0;
}
```

C → P communication

```
Parent receives: 123456789
Child sends: 123456789
```