Last lecture: pipe & dup2

↳ int fds[2]; //at least length 2

pipe(fds); //reserves 2 lowest unused file descriptor nums

↳ any text written & not read just resides in "pool" maintained by OS

$\swarrow$ #define'd as 0

dup2(fds[0], FILENO_STDIN);

tells file descriptor 0 to abandon previous binding (e.g. the keyboard) and link to what fd[0] links to

int dup(int source) just chooses lowest unused fd and points it to same open file entry as source

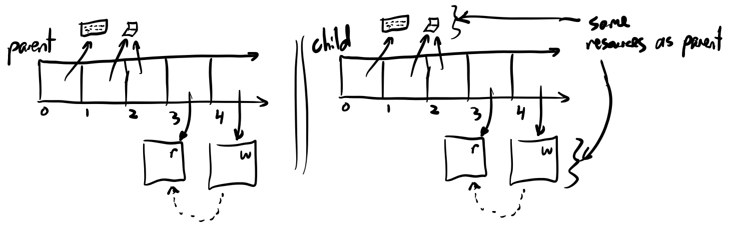
sort

default behavior (as w/ most built-in unix executables): read from stdin

```
typedef struct {
    pid_t pid; //of child process
    int supplyfd;
} subprocess_t;


subprocess_t subprocess(char * command) {
}


int main(→) {
    subprocess_t sp = subprocess("/usr/bin/sort");
    const char * words[] = {
    waitpid(sp.pid, NULL, 0);
    return 0;
} ... see slides!
```

see dprintf in stdlib set

↳ we're running a program as child process & can programmatically feed it input via stdin

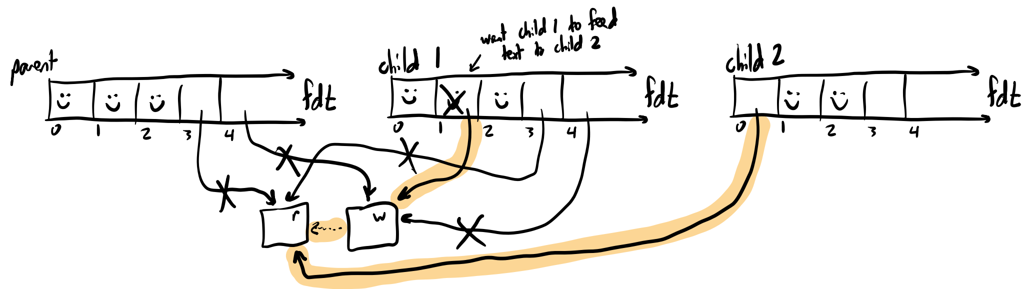


```
subprocess_t subprocess(char *command);
    int fds[2];
    pipe(fds);

    pid_t pid = fork();
    if (pid == 0) {
        close(fds[1]); //child is not writing
        dup2(fds[0], FILENO_STDIN); //at this point the read side of cant has refcount=3
        close(fds[0]); // back to 2
        char * argv[] = {"/bin/sh", "-c", command, NULL}; //running a helper shell that does the parsing of command str
        execvp(argv[0], argv);
    }

    subprocess_t = {pid, fds[1]};
    close(fds[0]);
    return sp;
```

} //at end: left w/ r/w refcounts both equal to 1 - ideal!

<u>pipelines</u>

"|" separates commands in terminal prompt to run in <u>parallel</u>



```
void pipeline(char *argv1[], char* argv2[], pid_t pids[]) {
    int fds[2];
    pipe(fds);

    pids[0] = fork();
    if (pid[0] == 0) {
        close(fds[0]);
        dup2(fds[1], FILENO_STDOUT);
        close(fds[1]);
        execvp(argv1[0], argv1);
    }
    //child 2 doesn't even need fds[1] - the write side of pipe
    close(fds[1]);
    pids[1] = fork();
    if(pids[1] == 0) {
        dup2(fds[0], FILENO_STDIN);
        close(fds[0]);
        execvp(argv2[0], argv2);
    }
    close(fds[0]);
```

lots of closes... have to produce & declare all shared resources **before** forking

↳ pipe2(fds, O_CLOEXEC);

↓

fds marked as "self-closing on execvp boundaries" - can do w/o these close statements