

Pipes

- Pipes

interprocess communication

system provided facilities

passing information

read/write

“special file” with limited capacity

PIPE_SIZE

`<linux/limits.h>`

FIFO

- Pipes

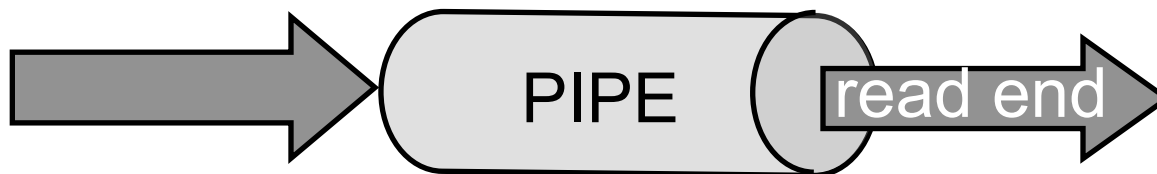
write end

read end

system maintains pointer to current location

system manages current location

no seek permitted



- Synchronization
 - done by system
 - atomic read/write
 - no seek
 - blocks on
 - write full
 - empty read
 - open for read with no open for write
 - use read/write system call
 - unbuffered I/O

- Synchronization

```
ssize_t write (int fd, const void *buf, size_t count);
```

write count bytes from buffer to file descriptor fd

returns number of bytes written

-1 on error

sets errno

- errors
 - i/o error
 - bad file pointer
 - resource temporarily unavailable
 - bad address
 - file too large
 - no space left on device
 - numerical result out of range

- **write**
 - append to end of pipe
 - writes of `PIPE_BUF` size or less are guaranteed to **not** be interleaved with other write requests to same pipe
 - write to a pipe not opened for read generates a `SIGPIPE` signal (default action is to terminate)
 - if both `ON_NONBLOCK` and `O_NDELAY` flags are clear, `write` blocks if device is busy
 - set in `fcntl` system call
 - if both `ON_NONBLOCK` and `O_NDELAY` flags are not clear `write` will not block

- read

```
ssize_t read (int fd, void *buf, size_t count);
```

reads count bytes

unbuffered I/O read system call

returns

- actual number of bytes read

- zero if at end of pipe

- 1 on error

- sets errno

- possible errors
 - I/O error
 - No such file or address)file descriptor)
 - Bad file descriptor (not opened for reading)
 - bad address (buffer)

- read
 - initiated from current position
 - no seeks
 - If both `ON_NONBLOCK` and `O_NDELAY` flags are clear read blocks on empty until data is written or pipe is closed
 - if pipe is not opened for writing returns a 0

- unnamed pipes
 - used only by related processes
 - parent/child
 - child/child
 - exists as long as they are in use
- named pipes
 - exists as directory entries
 - have file access permissions
 - can be used by unrelated processes

- unnamed pipes

- pipe system call

```
int pipe( int filedes[2] );
```

- Returns a pair of integer file descriptors

- filedes[0]

- filedes[1]

- they reference two data streamsLinux

- half duplex: unidirectional

- Solaris

- full duplex: bidirectional

- unnamed pipes
 - half duplex
 - `filedes[0]`
 - reading
 - `filedes[1]`
 - writing
 - full duplex
 - agree on use of read/write ends
 - both opened for reading and writing

- possible errors
 - file table overflow
 - too many opened files
 - bad address (invalid file descriptor)

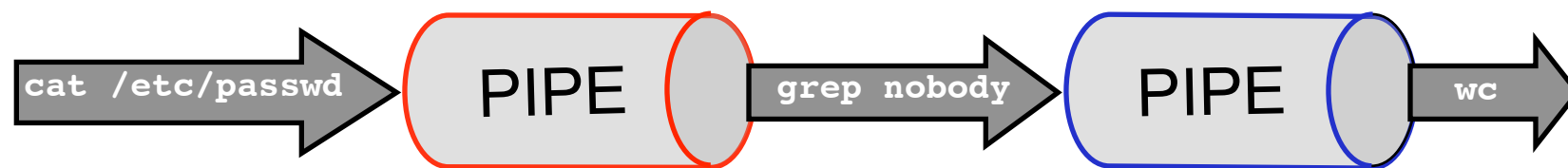
```
int main(int argc, char *argv[ ]) {
    int f_des[2];
    static char message[BUFSIZ];
    if (argc != 2) {
        cerr << "Usage: " << *argv << " message\n";
        return 1;
    }
    if (pipe(f_des) == -1) {                // generate the pipe
        perror("Pipe");    return 2;
    }
    switch (fork( )) {
        case -1:
            perror("Fork");    return 3;
        case 0:                            // In the child
            close(f_des[1]);    // close write end of pipe
            if (read(f_des[0], message, BUFSIZ) != -1) {
                cout << "Message received by child: [" << message
                    << "]" << endl;
                cout.flush();
            } else {
                perror("Read");    return 4;
            }
            break;
        default:                            // In the Parent
            close(f_des[0]);    // close read end of pipe
            if (write(f_des[1], argv[1], strlen(argv[1])) != -1) {
                cout << "Message sent by parent    : [" <<
                    argv[1] << "]" << endl;
                cout.flush();
            } else {
                perror("Write");    return 5;
            }
    }
    return 0;
}
```

```
$ ./a.out Hello
Message sent by parent    : [Hello]
Message received by child: [Hello]
```

pipe to send first argument from parent to child

- I/O redirection
 - associate standard input/output to pipe

```
cat /etc/passwd | grep nobody | wc
```



- dup and dup2

```
int dup ( int oldfd );
```

- duplicates an opened file descriptor
- new descriptor
 - references file system table entry for **next available nonnegative file descriptor**
 - always returns next lowest available file descriptor
 - share the same file pointer (offset) as original
 - same access control as original
 - share locks
 - remain open across exec
 - do not share *close-on_exec* flag

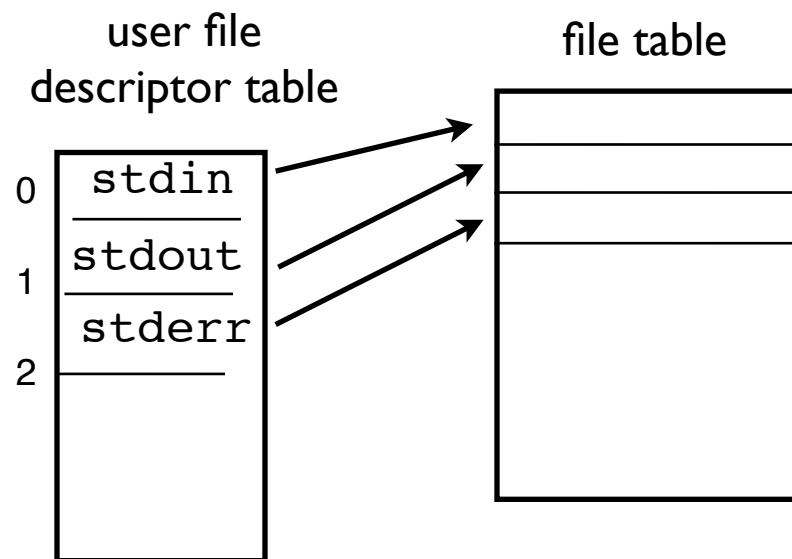
- dup

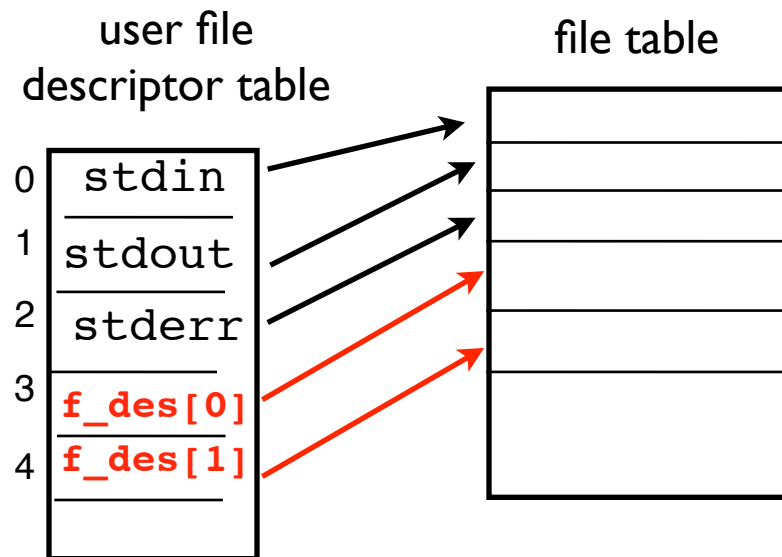
*always return the next **lowest** available descriptor

```
int f_des_[2];  
pipe(f_des);  
close(fileno(stdout) ); // close standard output  
dup(f_des[1]);           // duplicate 1st free descriptor  
                        // as write end of pipe  
// any write to stdout will go to write end of pipe
```

race condition between close and dup

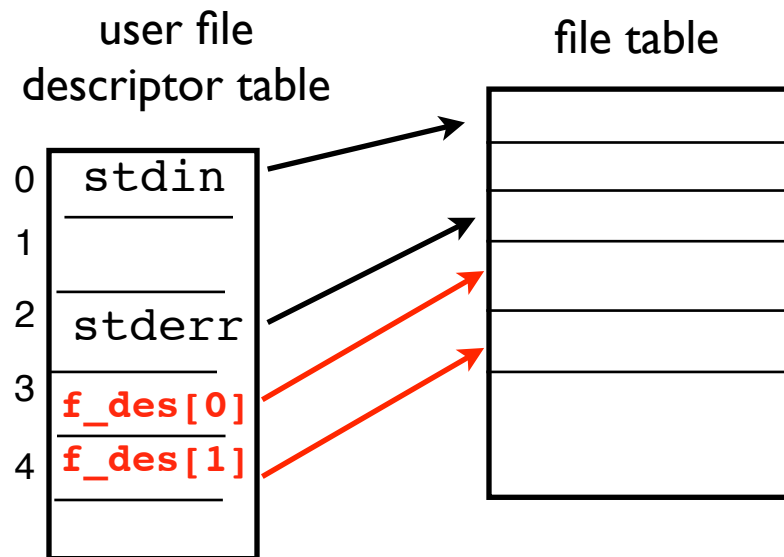
for example a signal-catcher that closes a file



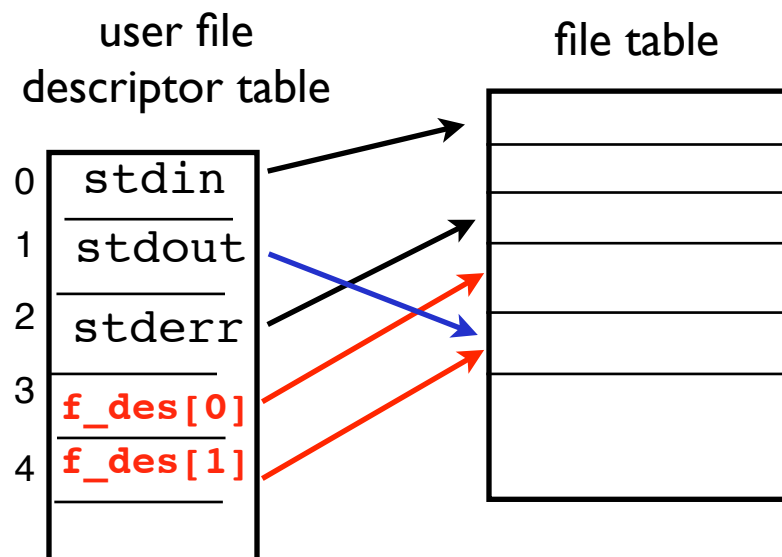


```
int f_des[2];
```

```
pipe(f_des);
```



```
int f_des_[2];  
  
pipe(f_des);  
  
close(filenno(stdout) ); // close standard output
```



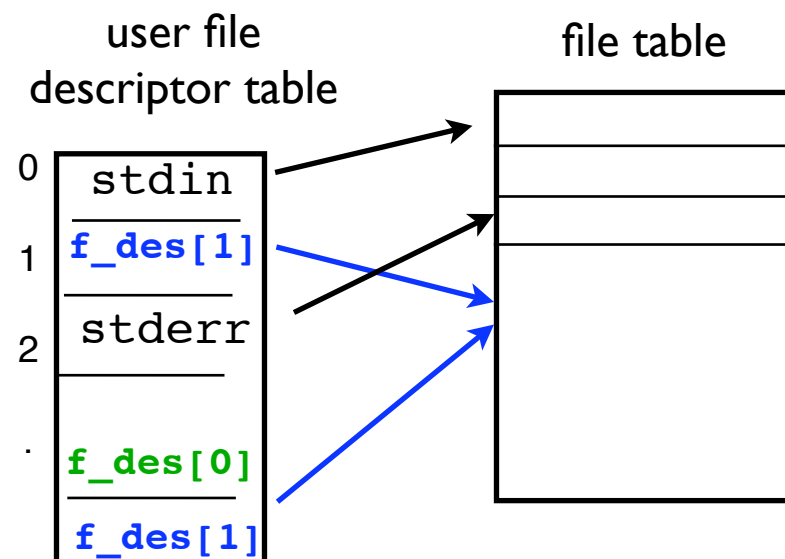
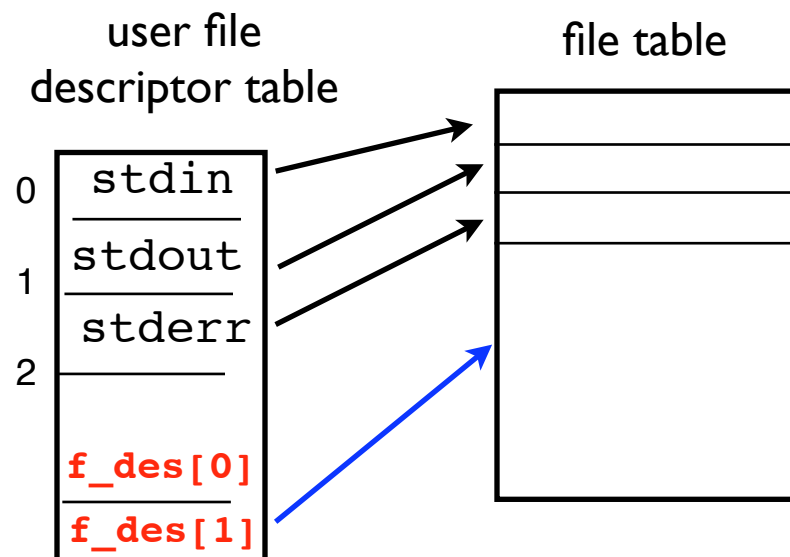
stdout is still entry 1 in file descriptor table

```
int f_des_[2];
```

```
pipe(f_des);
```

```
close(filenno(stdout) ); // close standard output
```

```
dup(f_des[1]);           // duplicate 1st free descriptor  
                          // as write end of pipe
```



`stdout` is still entry 1 in file descriptor table

```
int f_des_[2];

pipe(f_des);

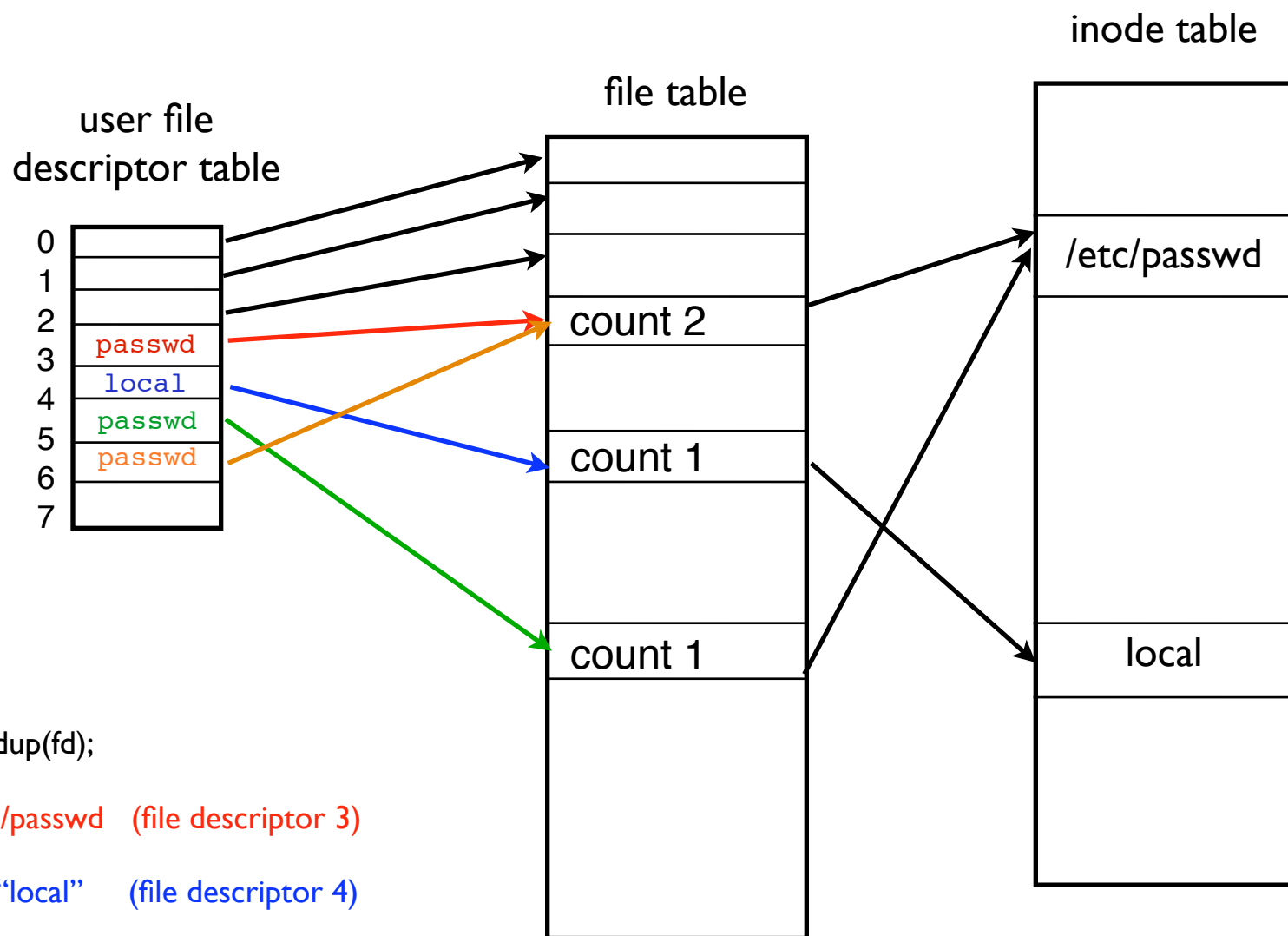
close(filenno(stdout) ); // close standard output

dup(f_des[1]);           // duplicate 1st free descriptor
                        // as write end of pipe
```

- newfd = dup(fd)
 - open /etc/passwd (file descriptor 3)
 - open file “local” (file descriptor 4)
 - open /etc/passwd (file descriptor 5)
 - dup fd 3 (file descriptor 6)

user file
descriptor table

0	
1	
2	
3	passwd
4	local
5	passwd
6	passwd
7	



```
newfd = dup(fd);
```

```
open /etc/passwd (file descriptor 3)
```

```
open file "local" (file descriptor 4)
```

```
open /etc/passwd (file descriptor 5)
```

```
dup fd 3 (file descriptor 6)
```

- dup2

```
int dup2 ( int oldfd, int newfd );
```

Closes and duplicates file descriptor

atomic operation

if **newfd** is already open

closed before duplicating

```
/* A home grown last | sort cmd pipeline. */
```

```
enum { READ, WRITE };
```

```
int
```

```
main( ) {
```

```
    int      f_des[2];
```

```
    if (pipe(f_des) == -1) {
```

```
        perror("Pipe");
```

```
        return 1;
```

```
    }
```

```
    switch (fork( )) {
```

```
    case -1:
```

```
        perror("Fork");
```

```
        return 2;
```

```
    case 0:
```

```
        // In the child
```

```
        dup2( f_des[WRITE], fileno(stdout));
```

```
        close(f_des[READ] );
```

```
        close(f_des[WRITE]);
```

```
        execl("/usr/bin/last", "last", (char *) 0);
```

```
        return 3;
```

```
    default:
```

```
        // In the parent
```

```
        dup2( f_des[READ], fileno(stdin));
```

```
        close(f_des[READ] );
```

```
        close(f_des[WRITE]);
```

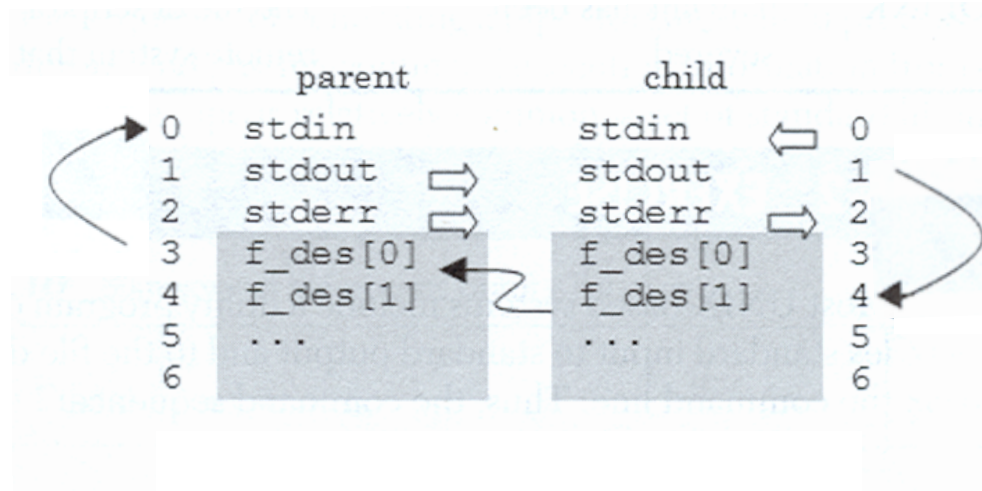
```
        execl("/bin/sort", "sort", (char *) 0);
```

```
        return 4;
```

```
    }
```

```
    return 0;
```

```
}
```



- Summary - unnamed pipes
- Create pipes
- Generate child process
- Close/duplicate file descriptors
- Close unneeded ends of pipe
- Communicate

- **popen pclose system calls**

```
FILE *popen ( const char *command, const char *type);
```

```
int pclose ( FILE *stream);
```

fork child process
duplicate file descriptors
pass command

popen

- duplicate file descriptors

- fork child process which will exec /bin/sh

- /bin/sh executes passed shell command

- duplicate file descriptors

 - W: parent process can write to standard input of new shell
file pointer referenced by popen

 - R: parent process can read from standard output of new
shell

- fully buffered

- pass command argument

pclose

- close data stream penned by popen

- returns exit status of shell command

```
/* Using the popen and pclose I/O commands*/
```

```
.....
```

```
;
int
main(int argc, char *argv[ ]) {
    FILE    *fin, *fout;
    char     buffer[PIPE_BUF];
    int      n;
    if (argc < 3) {
        cerr << "Usage " << argv << "cmd1 cmd2" << endl;
        return 1;
    }

```

```
    fin = popen(argv[1], "r");
```

forks child that executes command argv[1] output read by parent with file pointer fin

```
    fout = popen(argv[2], "w");
```

forks child that executes command argv[2] input read from parent's output

```
    fflush(fout);
```

```
    while ((n = read(fileno(fin), buffer, PIPE_BUF)) > 0)
```

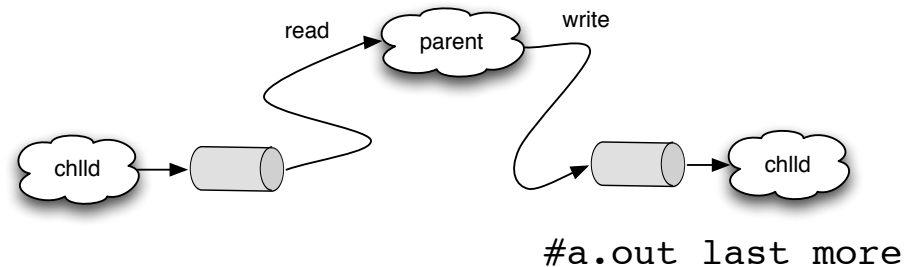
```
        write(fileno(fout), buffer, n);
```

```
    pclose(fin);
```

```
    pclose(fout);
```

```
    return 0;
```

```
}
```



- **named pipes (FIFO)**
 - directory entry
 - permissions
 - can be used by unrelated processes
 - data stored in kernel not in file system
 - creation
 - **mknod** **name p**
 - **mkfifo** **name p**

mknod: generate special files (block character device files in /dev)
nonprivileged users can only generate named pipes

```
%mkfifo APIPE p
```

```
prw-r--r--      1 dcanas  dcanas    0 Sep 16 11:21 APIPE
```



```
%mkfifo APIPE p
```

```
prw-r--r--      1 dcanas  dcanas    0 Sep 16 11:21 APIPE
```

```
% cat copy.c > APIPE &
```

```
[1] 287
```

```
% cat < APIPE
```

```
this is the contents of copy.c  
a test file for pipes  
last line
```

```
[1]+  Done
```

- System call

```
int mknod(const char *pathname, mode_t mode, dev_t dev);
```

- creates a file descriptor by pathname
- file types and permissions are ORed
- dev argument only for character, block devices
 - use 0 for FIFO
- File types;

S_FIFO	FIFO special (non privileged users)
S_IFCHR	character special
S_IFDIR	directory
S_IFBLK	block special
S_IFREG	ordinary file

- System call

```
int mkfifo(const char *pathmane, mode_t mode );
```

uses mknod to create a FIFO