

Fundamentos de los Sistemas Operativos (FSO)

Departamento de Informàtica de Sistemes y Computadoras (DISCA)

Universitat Politècnica de València

Part 4: File systems and I/O

Seminar 11

Unix file system calls

fSO

DISCA



UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

- **Goals**

- To know the **file descriptor** concept
- To know the **file descriptor table** and its utility
- To use **Unix file system calls** in C programs
- To use **Unix process input/output redirection**
- To do **pipe based process communication** in Unix

- **Bibliography**

- “UNIX Systems Programming”, Kay A. Robbins, Steven Robbins. Prentice Hall. ISBN 0-13-042411-0, chapters 4, 5 and 6

- **Unix files**
- Unix file system calls
- Redirections and pipes
- Redirection and pipe system calls
- C examples

- Secondary storage abstraction
- File types:
 - Regular:** common files that contain data or binary code (text files, image files, executable files, etc.)
 - Directory:** file containers which content is directory entries
 - Pipe:** unnamed sequential access files for interprocess communication
 - FIFO:** named sequential access files for interprocess communication
 - Special:** hardware or virtual device system abstraction, for instance:
 - Console devices are `/dev/ttyX` ($X=0,1,..$)
 - Sound card is `/dev/dsp`
 - Virtual sink is `/dev/null`

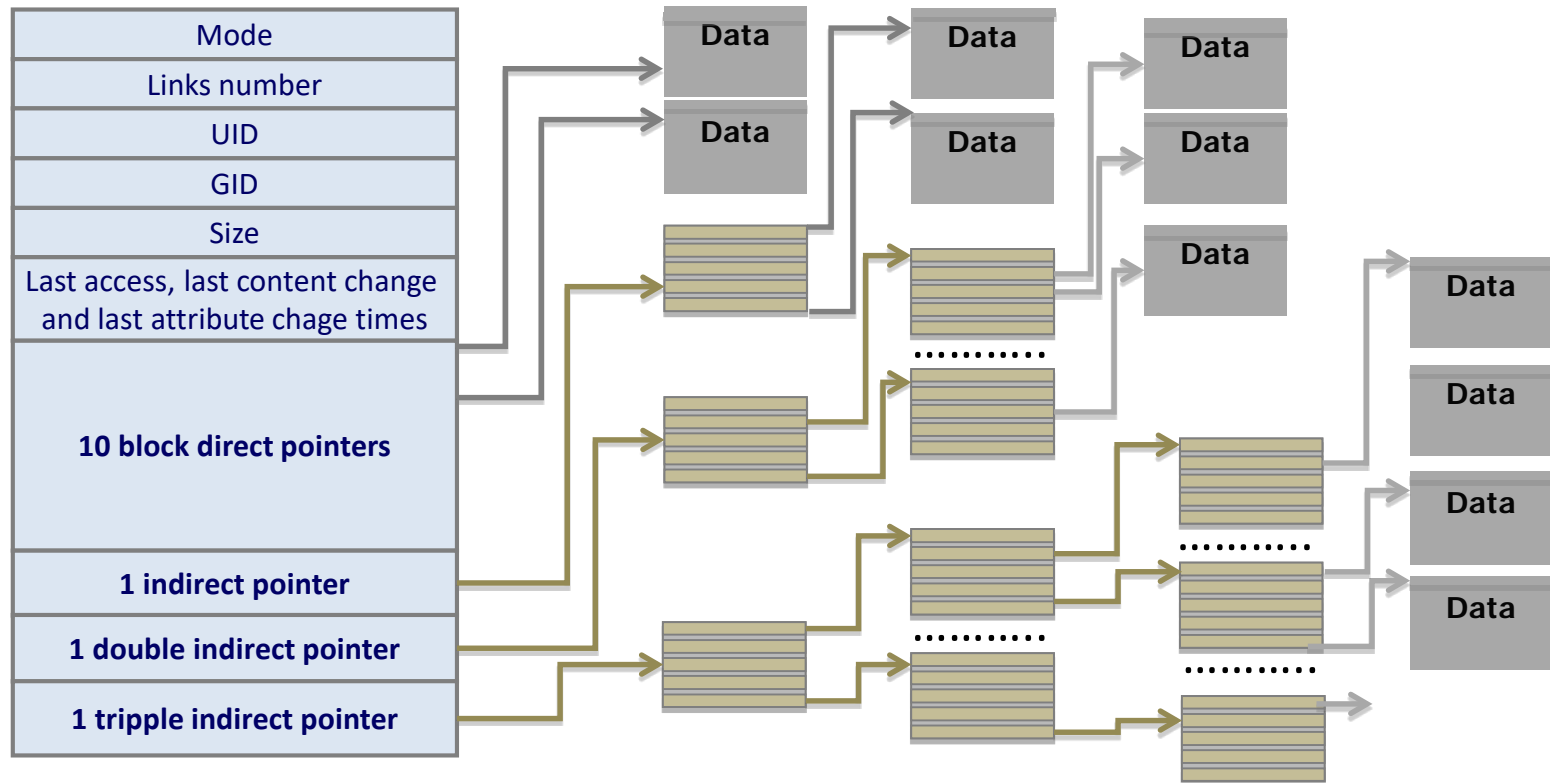
Note: In the UNIX shell the file type is shown by `ls -la` command as:

- Regular file: `'-'`
- Directory: `'d'`
- Special: `'c'` (character device) o `'b'` (block device)
- FIFO: `'p'`

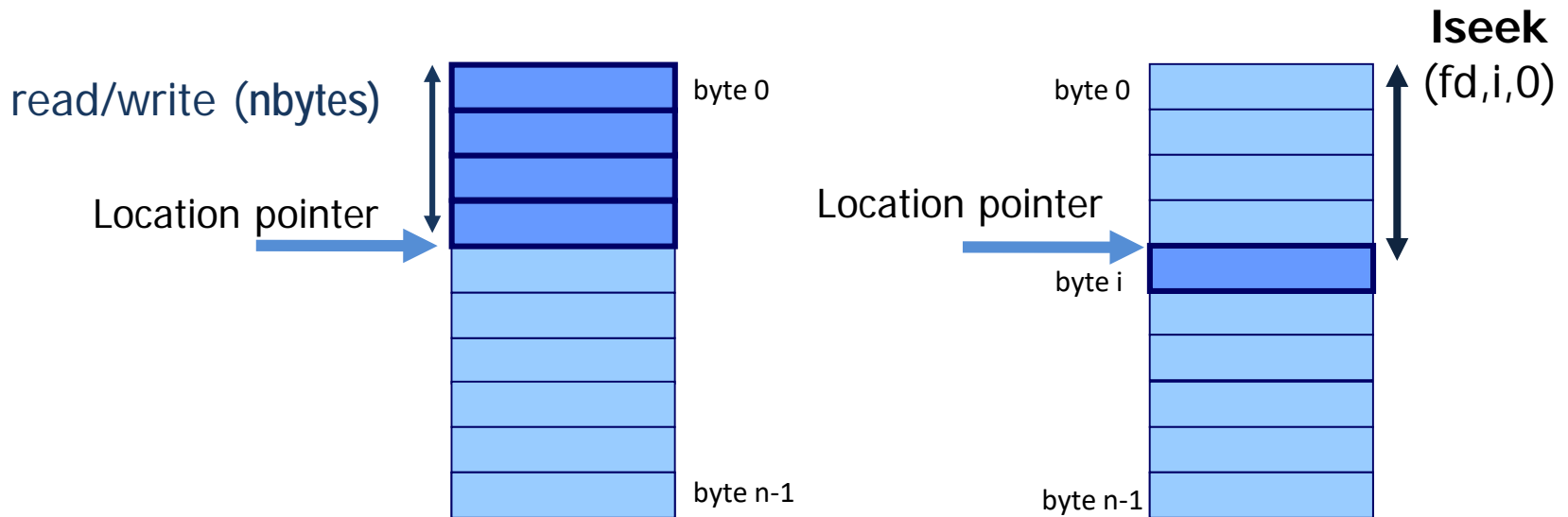
- **File attributes in Unix**
 - File type
 - Owner user (UID)
 - Owner group (GID)
 - Access permissions (permission bits)
 - Number of links
 - Creation, last access and last change time stamps
 - Size

• Unix directory entry: i-node

- OS data structure to store file attributes except its name (a file can have several names or links)
 - Every Unix file has one i-node
 - It points to file content using indexed block allocation that can be direct, indirect, double indirect and triple indirect



- **File structure**
 - Vector of bytes
- **File access mode**
 - **Sequential access** with *read/write* calls:
 - *read/write (fd, buffer, nbytes)*
 - ***lseek*** allows direct access specifying an offset from the file start, end or actual location
 - *lseek (fd, offset, from_where)*



- **File descriptor**

- To read or write a file it must be first opened and last closed

Open: `fd = open(filename, mode)`

Access: `read(fd,...)`, `write(fd,...)`,
`lseek(fd,...)`, ...

Close: `close (fd)`

- A file descriptor (fd) is an abstract file identifier local to every process
 - File access inside a process is done through the file descriptor (table index) given by **open**
 - Working with file descriptors does file access more efficient, avoids looking for them in disk for every access

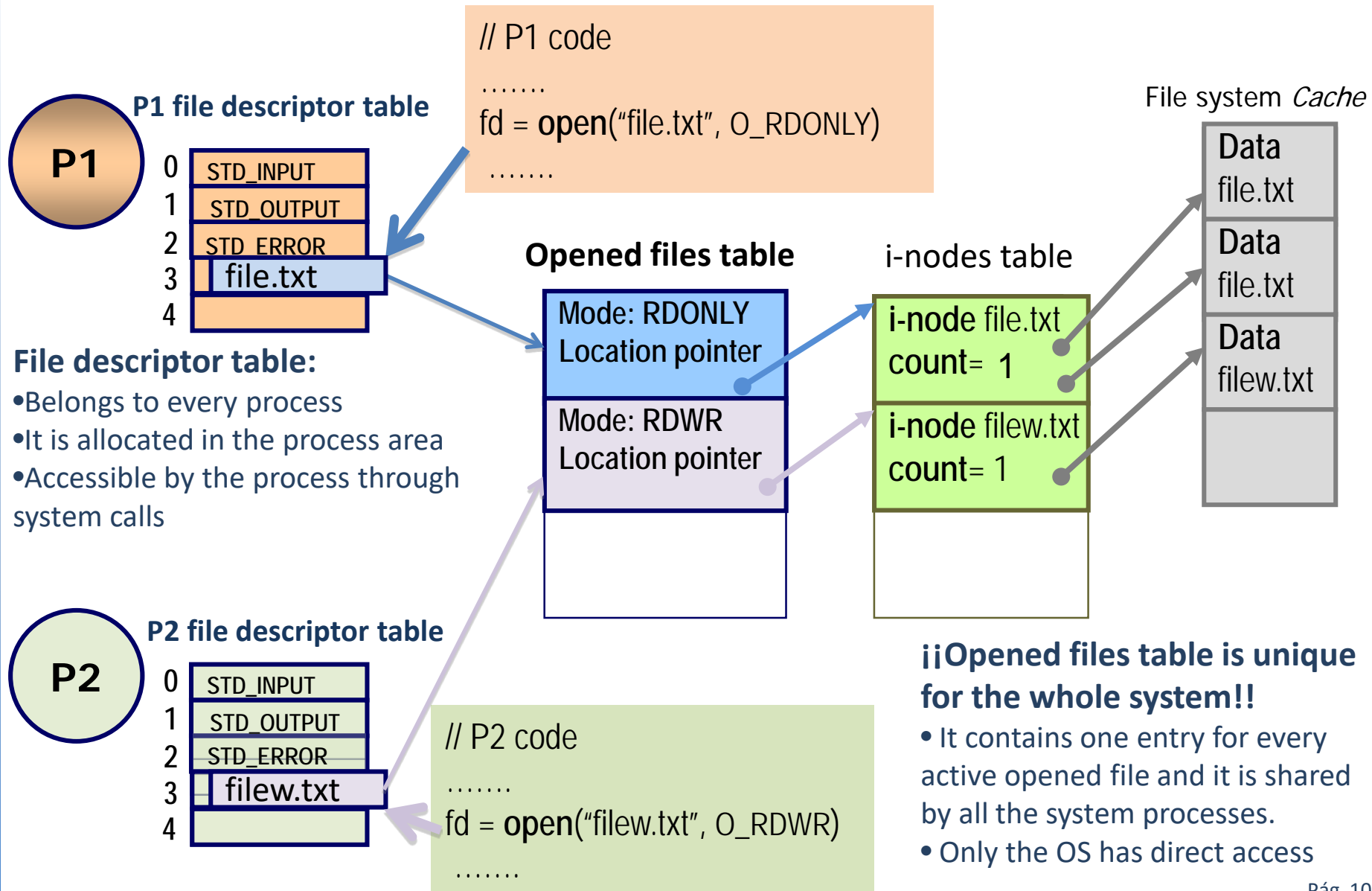
- **Open file operation**

- It looks for the file in the directory structure and brings its attributes to an entry in the opened files table located in main memory
- It registers some additional attributes like:
 - Location pointer
 - Number of active open calls
 - Disk location of data
- The file content is brought partially into memory buffers

- **Close file operation**

- It frees the corresponding entry in the opened files table

• Opened files table vs file descriptor table

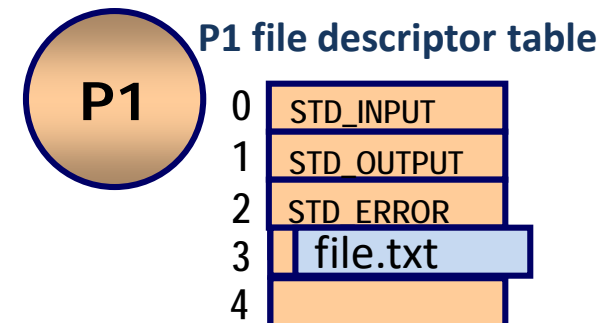


- File descriptors and standard I/O
 - The first three file descriptors in a process have a proper name:

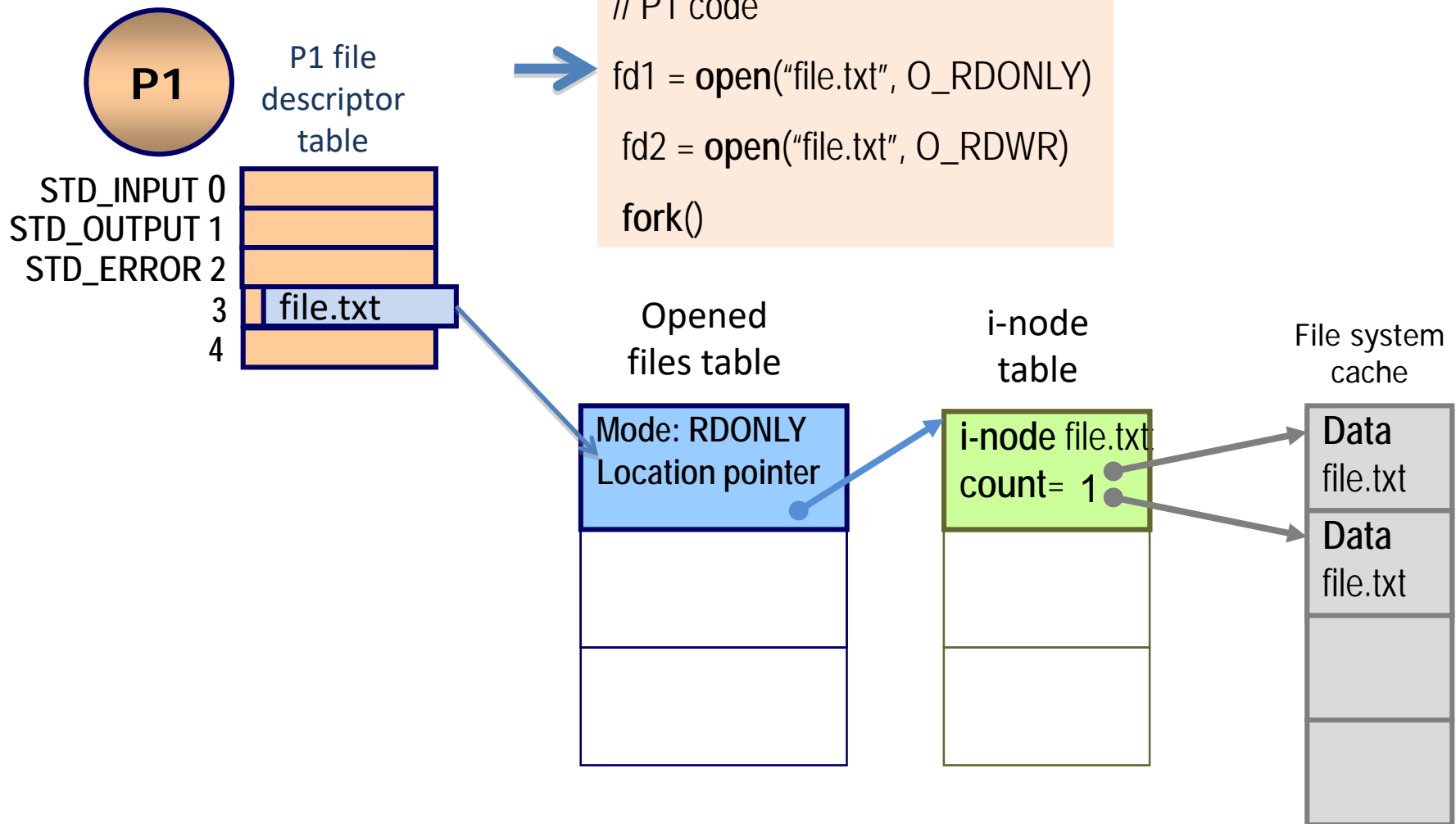
Descriptor	Constants	FILE *
0	STDIN_FILENO	<i>stdin</i> , standard input
1	STDOUT_FILENO	<i>stdout</i> , standard output
2	STDERR_FILENO	<i>stderr</i> , standard error

- By default these file descriptors are associated to the console
 - Console devices are `/dev/tty` or `/dev/ptn/n`
 - This associations can be modified using pipes or redirections

- Use examples:
 - **From the C library** `scanf` reads from standard input and `printf` writes on the standard output
 - **From the Shell:** its commands read and write on the standard I/O, for instance, command “`ls`” writes the file listing on the standard output and writes the error message “No such file or directory” in the standard error

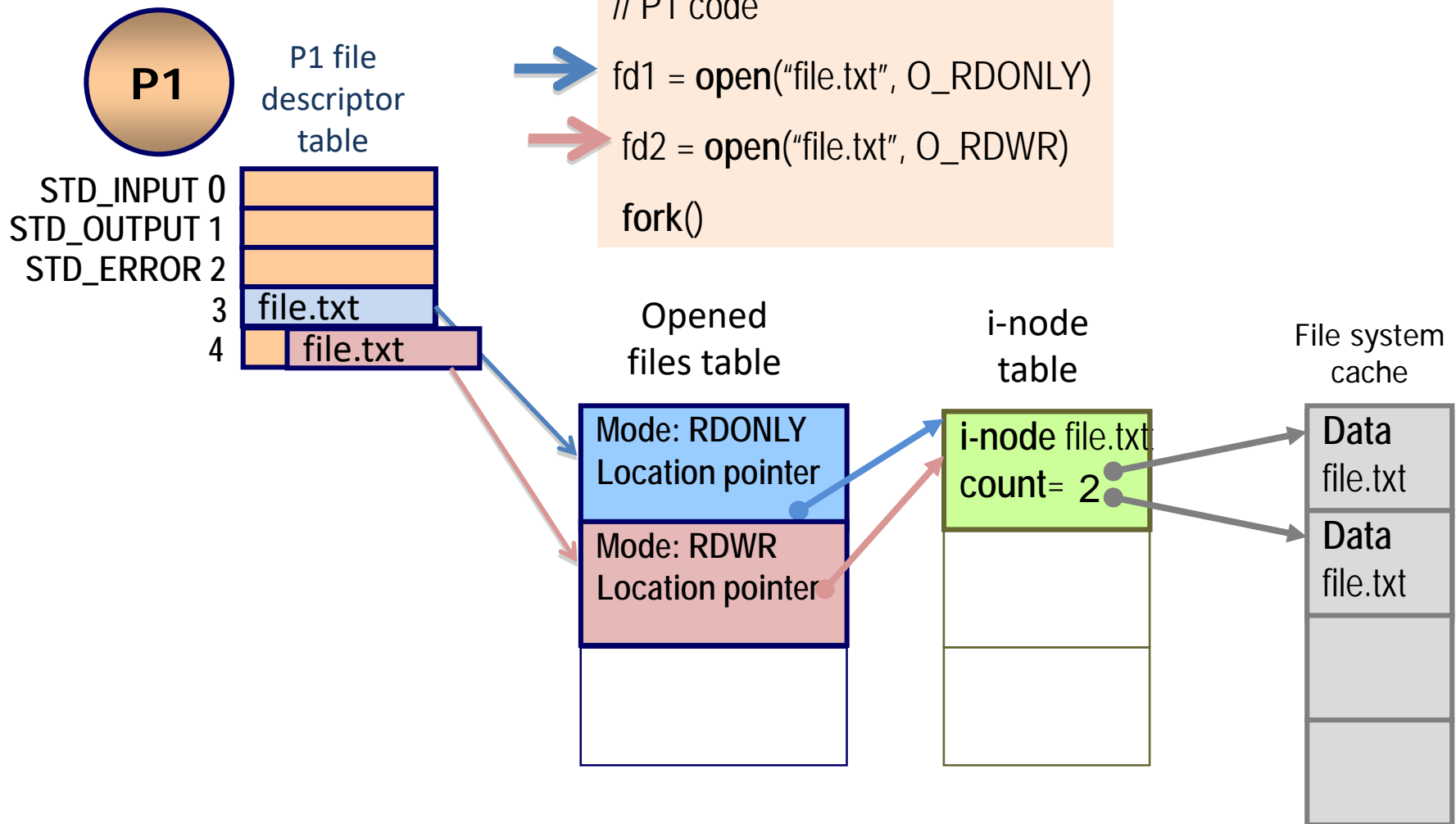


• Inheriting the file descriptor table

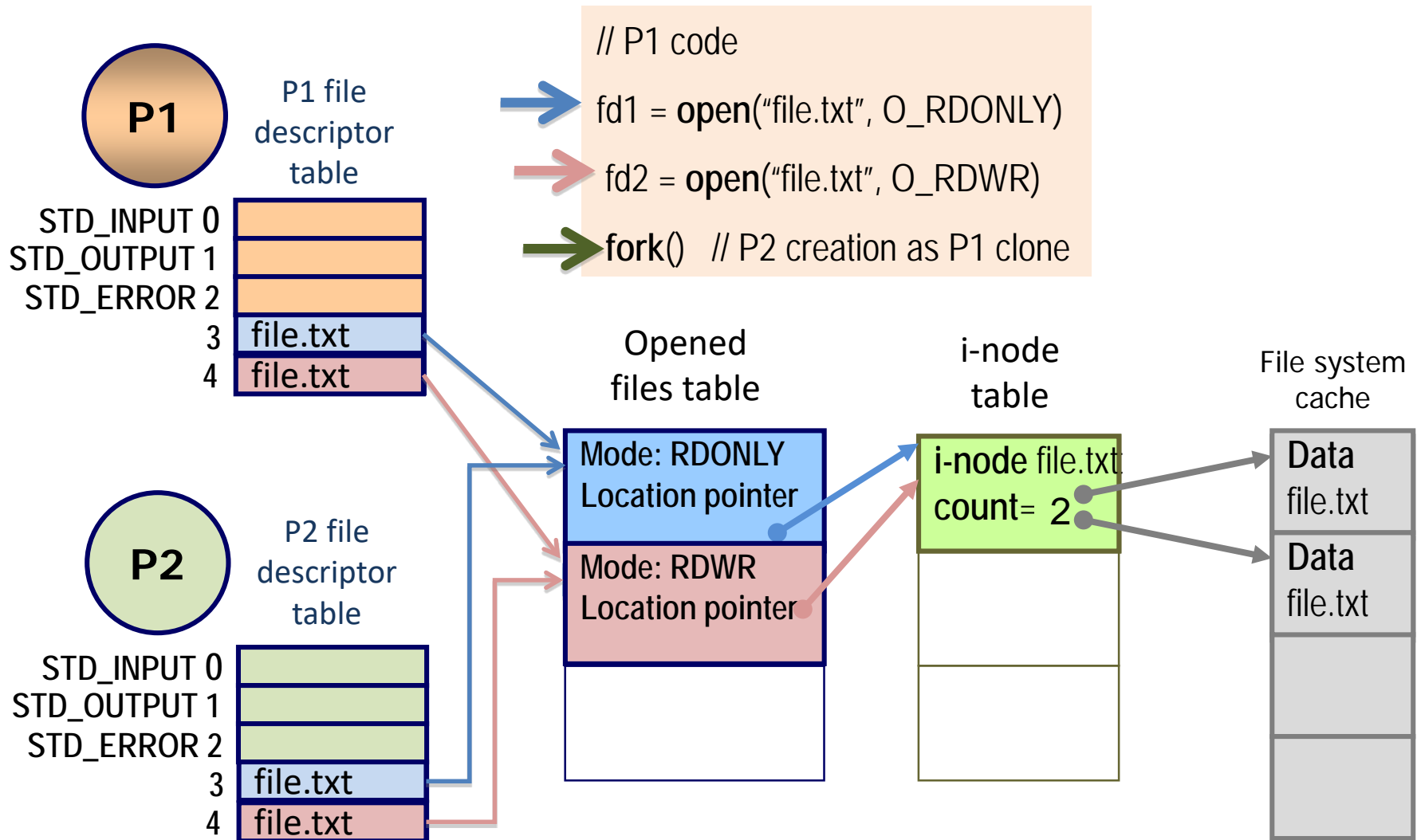


¡¡There is only one opened files table in the system!!

• Inheriting the file descriptor table



• Inheriting the file descriptor table



!!!Opened file descriptor are inheritable attributes

- Unix files
- **Unix file system calls**
- Redirections and pipes
- Redirection and pipe system calls
- C examples

- System call to work with files and devices
 - Unix implements a unified interface to access files and I/O devices

	Description
open	It opens/creates files
read	It reads files
write	It writes files
close	It closes a file
lseek	It sets file pointer location
stat	It gets information from file i-node

Note. Sytem calls “read” and “write” don’t perform any format conversion, so formatted I/O functions in C like *scanf* and *printf* include format conversion code

open: opening/creating files

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *path, int flags)
int open(const char *path, int flags, mode_t mode)
```

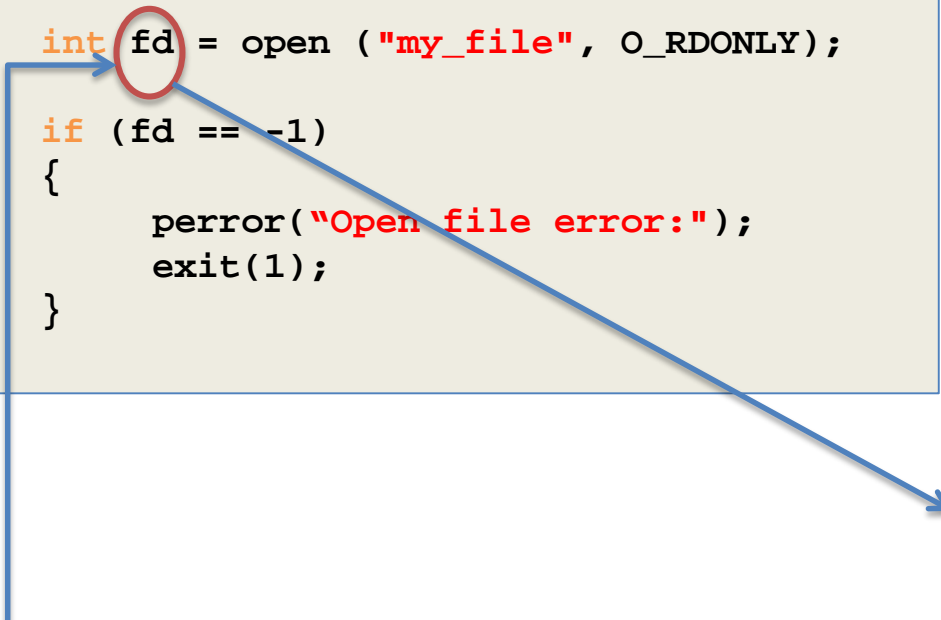
Description

- It associates a file descriptor with a file or hardware device
 - The lower free descriptor in the file descriptor table is chosen
 - Opened file descriptors are inheritable attributes
- **Flags**
 - `O_RDONLY`, `O_WRONLY`, `O_RDWR`
 - `O_CREAT`, `O_EXCL`, `O_TRUNC`, `O_APPEND`
- **Mode Examples:** `0755`, `04755`
 - `S_IRUSR`, `S_IWUSR`, `S_IXUSR`, `S_IRWXU`
 - `S_IRGRP`, `S_IWGRP`, `S_IXGRP`, `S_IRWXG`
 - `S_IROTH`, `S_IWOTH`, `S_IXOTH`, `S_IRWXO`
 - `S_ISUID`, `S_ISGID`

- open(): opening/creating a file
 - Example

```
#include <fcntl.h>
#include <stdlib.h>

main ( int argc, char* argv[] )
{
    int fd = open ( "my_file", O_RDONLY);
    if (fd == -1)
    {
        perror("Open file error:");
        exit(1);
    }
}
```



File named “my_file” is associated to entry 3 on the file descriptor table → fd = 3

[0]	/dev/tty0
[1]	/dev/tty0
[2]	/dev/tty0
[3]	
[4]	

read/write: reading/writing files

```
#include <unistd.h>
```

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

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

Description

- **read**: it asks for reading “nbyte” bytes from the file with **fd** file descriptor
 - Read bytes are stored in **buf**
 - It can read less bytes than the ones asked for if the end of file is reached
- **write**: It asks for writing “nbyte” bytes taken from **buf** in the file with **fd** file descriptor.
- By default **read** and **write** are blocking and can be interrupted by a signal
- **Returning value**
 - integer > 0**: it corresponds to the number of read/written bytes
 - 1: error** or interrupted by a signal (errors: fd is not a valid descriptor, **buf** is not a valid address, disk full, not allowed operation, etc.)
 - 0**: a read attempt after end of file

close: closing a file

```
#include <unistd.h>
```

```
int close(int fd)
```

Description

- It closes the fd file descriptor freeing that file descriptor table location
- Returning value
 - 0: success
 - 1: error and sets appropriate *errno* printable with “perror” (errno.h)
i.e. EBADF: fd is not a valid file descriptor

- Unix files
- Unix file system calls
- **Redirections and pipes**
- Redirection and pipe system calls
- C examples



- Standard I/O redirection from the shell

–Standard input redirection

```
$ mail gandreu < mensaje
```

0	mensaje
1	/dev/tty
2	/dev/tty

–Standard output redirection

```
$ echo hola > f1.txt
```

0	/dev/tty
1	f1.txt
2	/dev/tty

–Standard error redirection

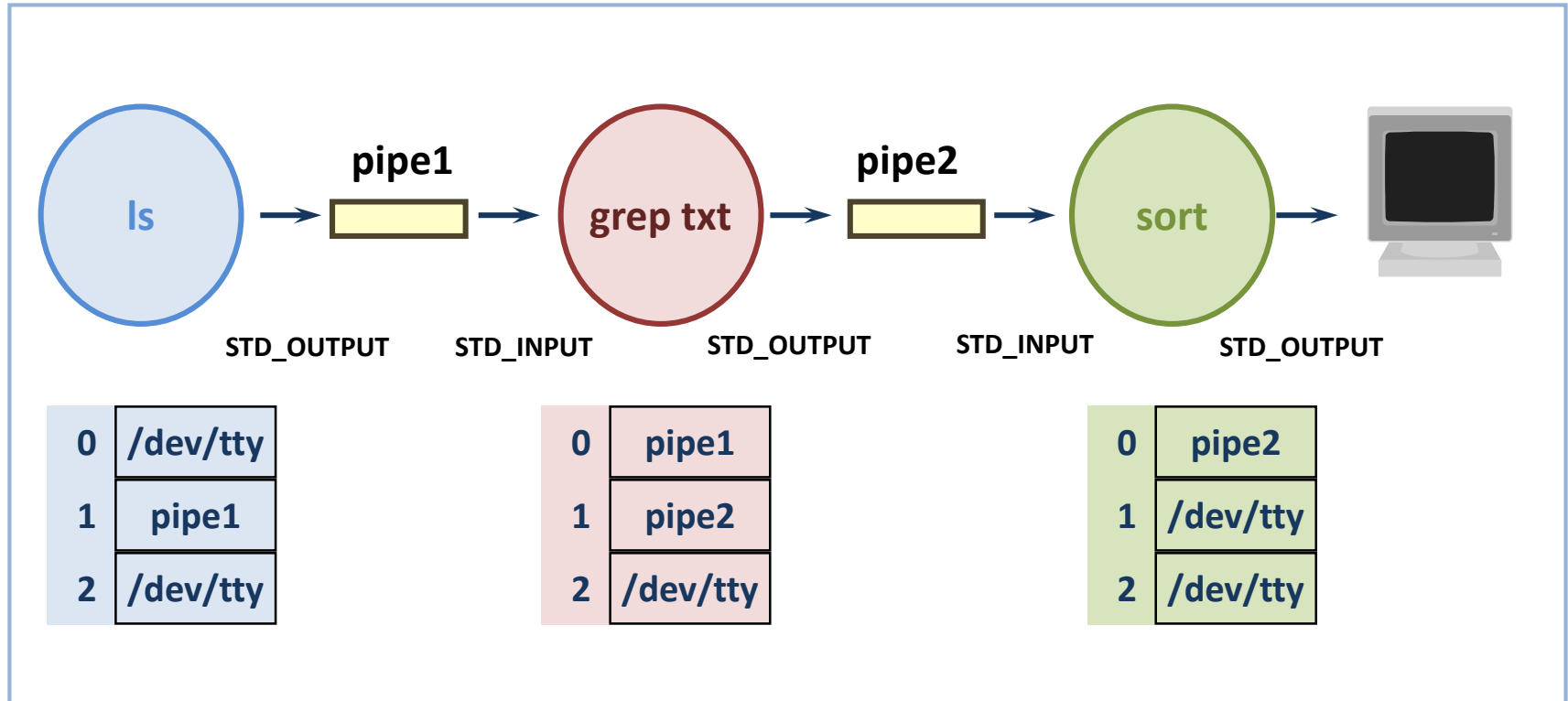
```
$ gcc prg.c -o prg 2> error
```

0	/dev/tty
1	/dev/tty
2	error

- Unix interprocess communication

- Unix provides the **pipe** mechanism to support interprocess communication
 - They are a special type of sequential access files with limited capacity
 - They can be shared due to inheritance mechanism

\$ **ls** | **grep txt** | **sort**



- Unix files
- Unix file system calls
- Redirections and pipes
- **Redirection and pipe system calls**
- C examples

- They allow performing communication between parent and children processes relying on inheritance

	Description
dup2	It duplicates a file descriptor
pipe	It creates an unnamed pipe
mkfifo	It creates a named pipe

dup, dup2: duplicating a file descriptor

```
#include <unistd.h>
int dup(int fd)
int dup2(int oldfd, int newfd)
```

Description

- **dup**: it returns a file descriptor value of a file which content is a copy of the one that corresponds to parameter **fd**
 - The returning descriptor is the lowest available in the process file descriptor table
- **dup2**: it closes **newfd** descriptor and then it copies **oldfd** into **newfd**
- Returning value
 - New descriptor file
 - -1 **error**: fd is not a valid descriptor. It surpasses the maximum number allowed of opened files (OPEN_MAX)

Example: dup2

// P1 code

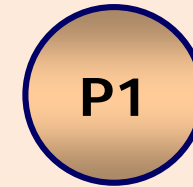
```
#define NEWFILE (O_WRONLY | O_CREAT | O_TRUNC)
```

```
#define MODE644 (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
```

→ `fd = open("file.txt", NEWFILE, MODE644);`

→ `dup2 (fd, STDOUT_FILENO);`

→ `close (fd);`



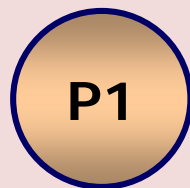
P1 file
descriptor
table

0	STD_INPUT
1	STD_OUTPUT
2	STD_ERROR
3	
4	



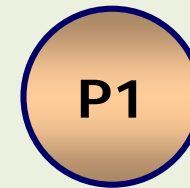
P1 file
descriptor
table

0	STD_INPUT
1	STD_OUTPUT
2	STD_ERROR
3	file.txt
4	



P1 file
descriptor
table

0	STD_INPUT
1	file.txt
2	STD_ERROR
3	file.txt
4	



P1 file
descriptor
table

0	STD_INPUT
1	file.txt
2	STD_ERROR
3	
4	

Example: dup

// P1 code

```
#define NEWFILE (O_WRONLY | O_CREAT | O_TRUNC)
```

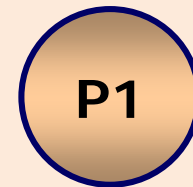
```
#define MODE644 (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
```

→ `fd = open("file.txt", NEWFILE, MODE644);`

→ `close(STDOUT_FILENO);`

→ `dup(fd);`

`dup2(fd, STDOUT_FILENO);`



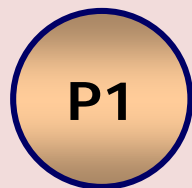
P1 file
descriptor
table

0	STD_INPUT
1	STD_OUTPUT
2	STD_ERROR
3	
4	



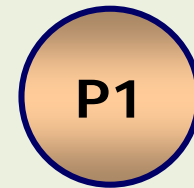
P1 file
descriptor
table

0	STD_INPUT
1	STD_OUTPUT
2	STD_ERROR
3	file.txt
4	



P1 file
descriptor
table

0	STD_INPUT
1	
2	STD_ERROR
3	file.txt
4	



P1 file
descriptor
table

0	STD_INPUT
1	file.txt
2	STD_ERROR
3	file.txt
4	

pipe: creating a pipe

```
#include <unistd.h>
int pipe(int fildes[2])
```

Description

- It creates a *pseudofile* (pipe) structured as a FIFO queue of bytes that is initially empty
 - Pipes are a interprocess communication provided by UNIX
 - The maximum pipe capacity is limited by the OS
- After **pipe** call, **fildes[0]** is a file descriptor to read the pipe and **fildes[1]** is a file descriptor to write the pipe
- Pipes together with the access descriptors are inherited by children processes and preserved after **exec** call
- Returning value
 - 0 success
 - -1 error: **fildes** is not valid, it surpasses the maximum number allowed of opened files (OPEN_MAX)



Pipe operation

- **read**

- If there are bytes available, at most the requested **nbytes** are read
- If pipe is empty, **read** suspends the calling process until bytes are available in the pipe
- When there is no pipe writing descriptor (belonging to the reading process or any other one) **read** doesn't suspend the process and returns 0, noticing in this way the ending data condition (end of file)

- **write**

- If there is enough pipe capacity to allocate the nbytes to write they are stored into the pipe in FIFO order
- If there is not enough capacity (pipe full) the writing process is suspended until space is available
- If writing is done into a pipe that doesn't own a reading descriptor (belonging to the reading process or any other one) the process that intends to write receives a SIGPIPE signal
 - This mechanism eases automatic removing of a pipe communicating process chain when one of its components aborts unexpectedly

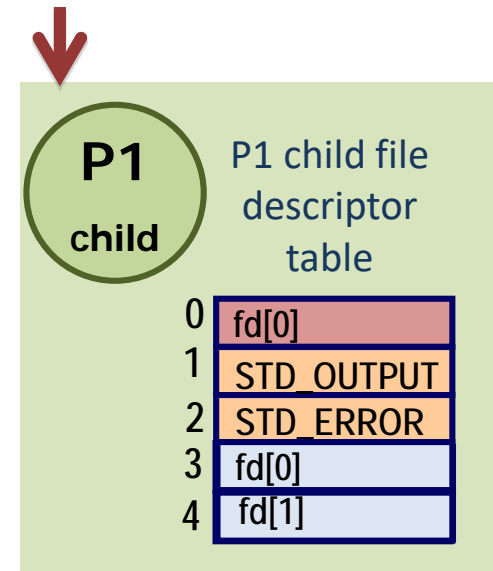
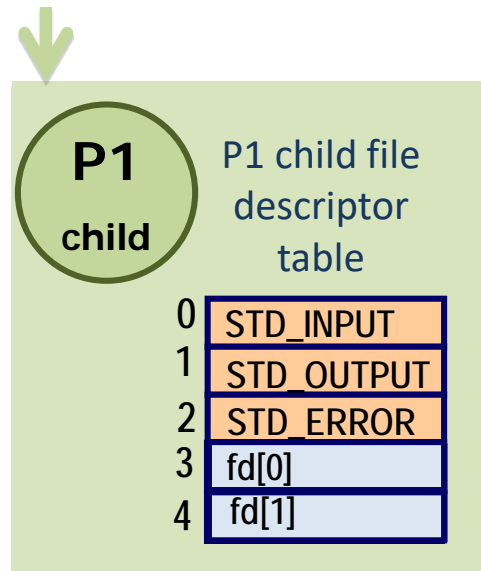
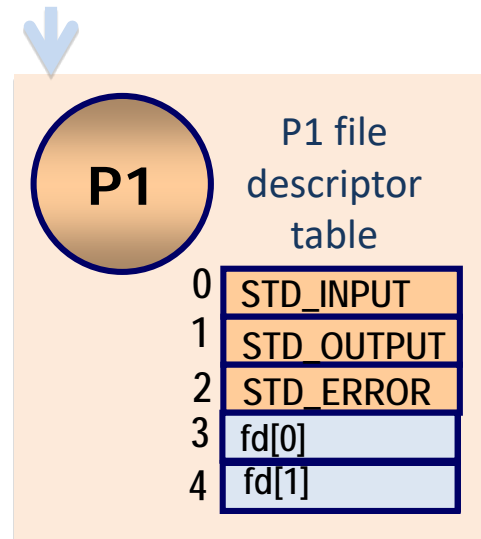
Redirection and pipe system calls

```
// P1 process code
pipe(fd);
if (fork() == 0) {

    // Child process code
    dup2 (fd[0], STDIN_FILENO);
    close (fd[0]);
    close (fd[1]);

    ...
} else {
    // Parent process code
    dup2 (fd[1], STDOUT_FILENO);
    close (fd[0]);
    close (fd[1]);

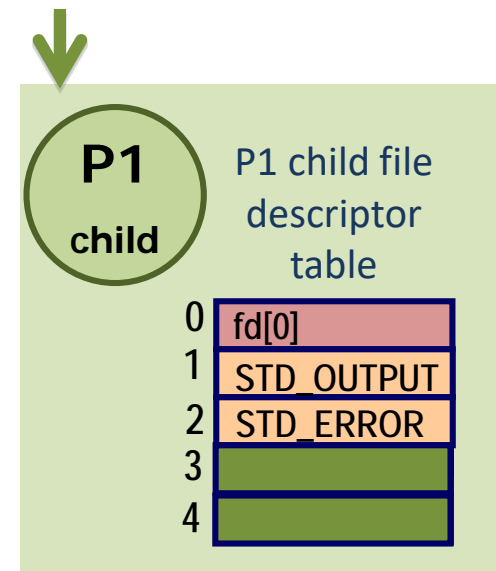
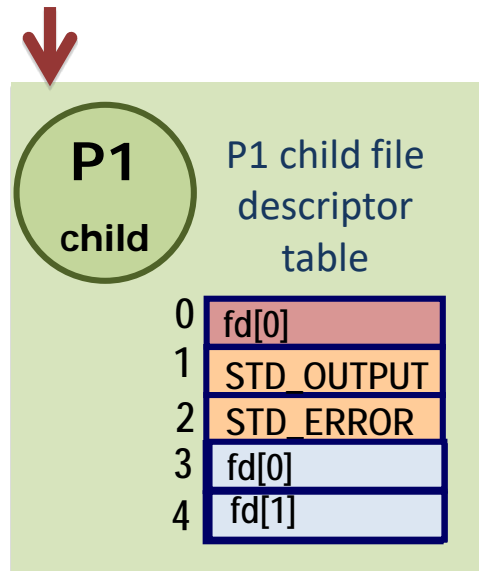
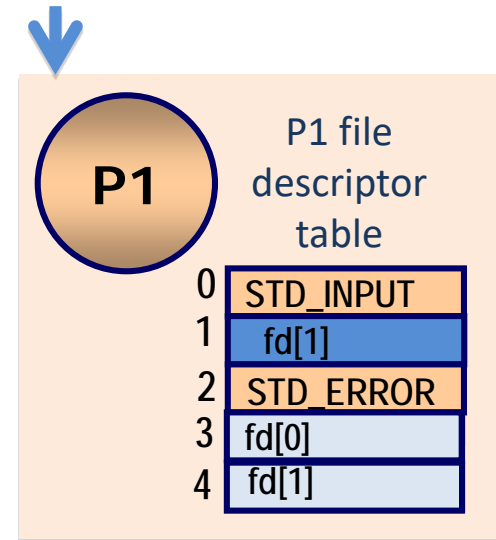
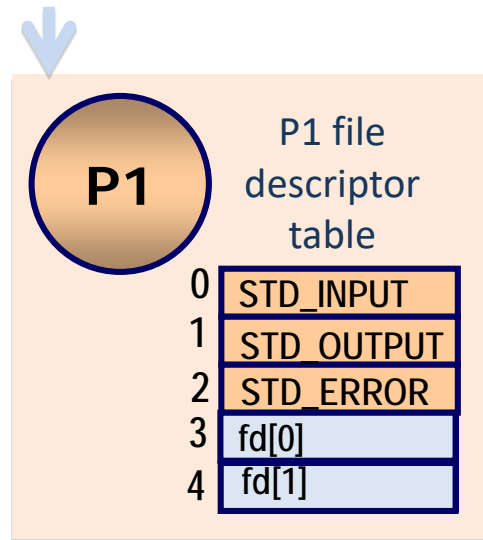
    ...
}
```



Redirection and pipe system calls

```
// P1 process code
pipe(fd);
if (fork() == 0) {

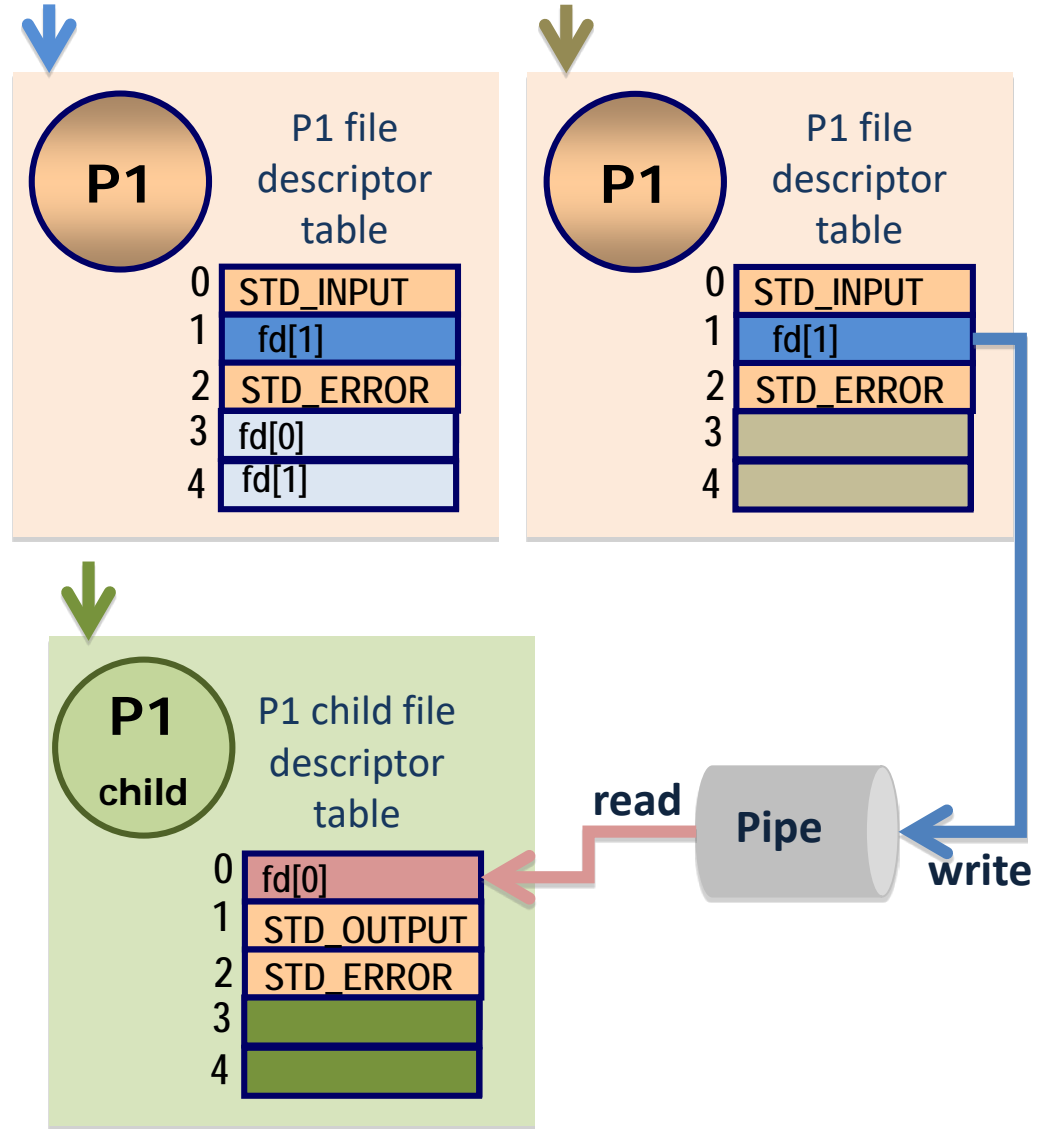
    // Child process code
    dup2 (fd[0], STDIN_FILENO);
    close (fd[0]);
    close (fd[1]);
    ...
} else {
    // Parent process code
    dup2 (fd[1], STDOUT_FILENO);
    close (fd[0]);
    close (fd[1]);
    ...
}
```



Redirection and pipe system calls

```
// P1 process code
pipe(fd);
if (fork() == 0) {

    // Child process code
    dup2 (fd[0], STDIN_FILENO);
    close (fd[0]);
    close (fd[1]);
    ...
} else {
    // Parent process code
    dup2 (fd[1], STDOUT_FILENO);
    close (fd[0]);
    close (fd[1]);
    ...
}
```



- Unix files
- Unix file system calls
- Redirections and pipes
- Redirection and pipe system calls
- **C examples**

- **Example: open, write and read**

```
int main(int argc, char *argv[]) {
    int from_fd, to_fd;
    int count;
    char buf[BLKSIZE];

    if (argc != 3) {
        fprintf(stderr, "Usage: %s from_file to_file\n", argv[0]);
        exit(1);
    }
    if ( (from_fd = open(argv[1], O_RDONLY)) == -1) {
        fprintf(stderr, "Could not open %s: %s\n", argv[1], strerror(errno));
        exit(1);
    }
    if ( (to_fd = open(argv[2], NEWFILE, MODE600)) == -1) {
        fprintf(stderr, "Could not create %s: %s\n", argv[2], strerror(errno));
        exit(1);
    }
    while ( (count= read(from_fd, buf, sizeof(buf))) >0 ) {
        if (write(to_fd, buf, count) != count){
            fprintf(stderr, "Could not write %s: %s\n", argv[2], strerror(errno));
            exit(1);
        }
    }
    if (count== -1){
        fprintf(stderr, "Could not read %s: %s\n", argv[1], strerror(errno));
        exit(1);
    }
    close(from_fd);
    close(to_fd);
    exit(0);
}
```

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

#define BLKSIZE 1
#define NEWFILE (O_WRONLY | O_CREAT | O_EXCL)
#define MODE600 (S_IRUSR | S_IWUSR)
```

To compile and execute do:

```
$ gcc my_copy.c -o my_copy
$ echo 'Hello read write' > hi.txt
$ ./my_copy hi.txt hi_copy.txt
$ cat hi_copy.txt
```

- **Example: dup2**

```
int redirect_output(const char *file) {
    int fd;
    if ((fd = open(file, NEWFILE, MODE644)) == -1) return -1;
    if (dup2(fd, STDOUT_FILENO) == -1) return -1;
    close (fd);
    return 0;
}

int main(int argc, char *argv[]) {
    int from_fd, to_fd;
    if (argc < 3) {
        fprintf(stderr, "Usage: %s to_file command args\n", argv[0]);
        exit(1);
    }
    if (redirect_output(argv[1]) == -1){
        fprintf(stderr, "Could not redirect output to: %s\n", argv[1]);
        exit(1);
    }
    if (execvp(argv[2], &argv[2]) < 0){
        fprintf(stderr, "Could not execute: %s\n", argv[2]);
        exit(1);
    }
    return 0;
}
```

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

#define NEWFILE (O_WRONLY | O_CREAT | O_EXCL)
#define MODE644 (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
```

To compile and execute do:

```
$ gcc dup2.c -o dup2
$ ./dup2 prueba ls
$ cat prueba
```

- Example: pipe

```
int main(int argc, char *argv[]) {
    int i, fd[2];
    if (argc < 2) {
        fprintf(stderr, "Usage: %s filter\n", argv[0]);
        exit(1);
    }
    pipe(fd);
    for(i=0; i<2; i++) {
        if (fork() == 0) { // children
            dup2 (fd[1], STDOUT_FILENO);
            close (fd[0]);
            close (fd[1]);
            execlp("/bin/ls", "ls", NULL);
            perror("The exec of ls failed");
        }
    }
    // parent
    dup2 (fd[0], STDIN_FILENO);
    close (fd[0]);
    close (fd[1]);
    execvp(argv[1],&argv[1]);
    fprintf(stderr,"The exec of %s failed", argv[1]);
    exit(1);
}
```

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

To compile and execute do:

```
$ gcc pipe.c -o pipe
$ ./pipe wc
```