Fundamentos de los Sistemas Operativos (FSO)

Departamento de Informática de Sistemas y Computadoras (DISCA)
Universitat Politècnica de València

Part 3: File systems and I/O

Seminar 7

Unix file system calls

fSO

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DISCA

Goals and bibliography

- **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

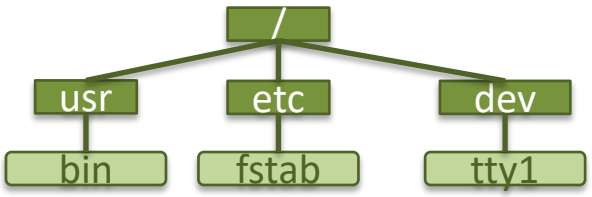
File system architecture

System calls (user library)
It does file and directory management from the programmer sight

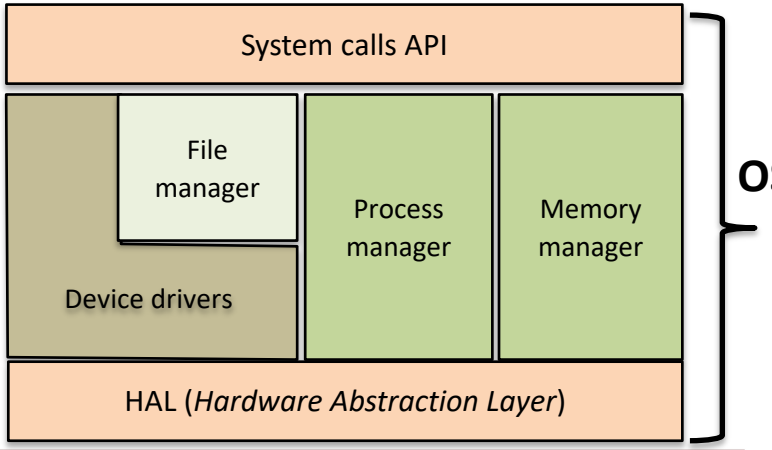
File manager
- It uses a device driver to do information transfer between disk and memory
- It implements mechanisms to provide **coherency**, **security** and **protection**
- **It optimizes** performance
- It creates basic user interface elements: **files** and **directories**

Device driver
- It dialogs with the device controller
- It starts physical operations and processes the end of I/O

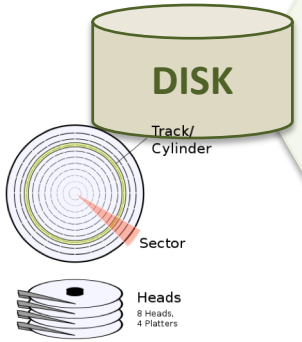
Physical level
Device + controller
- Block device
- Disk geometry
  Disk = Block vector
  Coordinates: CHS

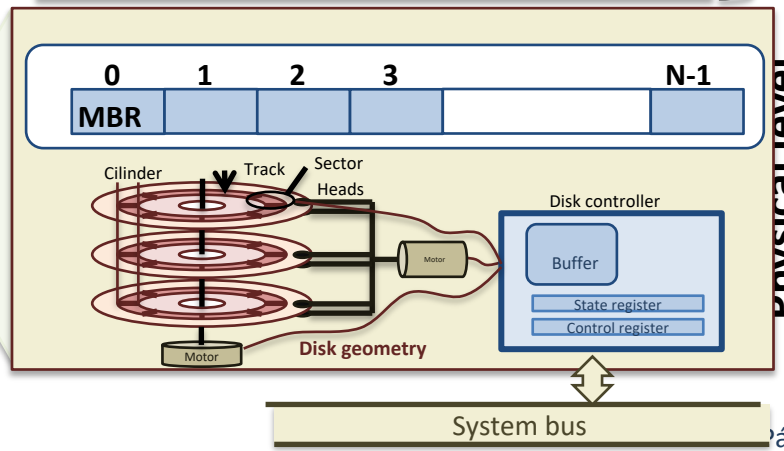


User level



OS





Physical level

File concept

- **File** = Attributes + Data

METADATA
Attributes
required to manage the file system

DATA
File content, like for instance text, binary code, etc

- **File attributes**
  - The change from system to system
    - Type
    - Size
    - Protection info
    - Owner
    - Creation date and time
- **File data**
  - The OS sees the file content as a byte vector, it is up to the application to give meaning to them
  - A file can store different information types: program source, text data, binary code, graphics, sound, etc
  - The file data can have a certain structure set by the file type (i.e. wav files, jpeg files, etc)
  - An executable file is a file made up by a sequence of code sections that the OS is able to load and execute

File list showing file attributes

```
gandreu@shell-labs:~/sisop/FSO/ejemplosC$ ls -lhi
total 136K
11928522 -rw-r--r-- 1 gandreu disca-upvnet  470 sep 20  2013 aritmetica_punteros.c
11930469 -rw-r--r-- 1 gandreu disca-upvnet  453 sep 26  2011 aritmetica_punteros.c~
11930470 -rwxr-xr-x 1 gandreu disca-upvnet 8,8K sep 26  2011 cir
11928236 -rw-r--r-- 1 gandreu disca-upvnet  193 sep 22  2011 circulo.c
11930472 -rwxr-xr-x 1 gandreu disca-upvnet 8,9K sep 26  2011 cua
11928246 -rwxr-xr-x 1 gandreu disca-upvnet 8,3K sep 16 16:41 cuad
11928433 -rwxr-xr-x 1 gandreu disca-upvnet 8,9K sep 20  2013 cuadrado
11928435 -rw-r--r-- 1 gandreu disca-upvnet  214 sep 22  2011 cuadrado.c
11928418 -rw-r--r-- 1 gandreu disca-upvnet  193 sep 22  2011 cuadrado.c~
11928437 -rwxr-xr-x 1 gandreu disca-upvnet 8,9K sep 22  2011 cuadro
11933192 -rw-r--r-- 1 gandreu disca-upvnet   80 sep 10  2013 error
11930471 -rw-r--r-- 1 gandreu disca-upvnet  579 sep 26  2011 hipotenusa.c
11930468 -rw-r--r-- 1 gandreu disca-upvnet  453 sep 26  2011 hipotenusa.c~
11927803 -rwxr-xr-x 1 gandreu disca-upvnet  424 jun 27  2014 hola.c
11928409 -rwxr-xr-x 1 gandreu disca-upvnet  241 jun 20  2014 hola.c~
11930473 -rwxr-xr-x 1 gandreu disca-upvnet 8,8K sep 26  2011 punt
11928436 -rwxr-xr-x 1 gandreu disca-upvnet 8,8K sep 22  2011 puntero
11928438 -rw-r--r-- 1 gandreu disca-upvnet  315 sep 22  2011 punteros.c
11928434 -rw-r--r-- 1 gandreu disca-upvnet  214 sep 22  2011 punteros.c~
11928243 -rw-r--r-- 1 gandreu disca-upvnet  525 sep 22  2011 variables.c
```

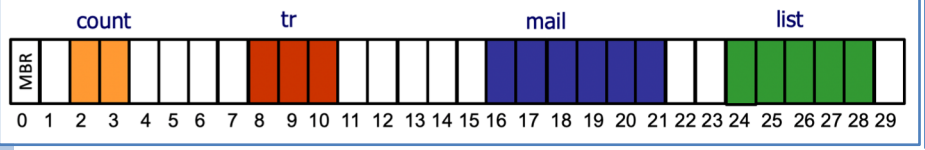
```c
#include <stdio.h>

main() {
  int x; /*variable entera*/
  int y;/*variable entera*/
  int *px; /* puntero a entero*/
  x=5;
  px=&x;/*px=direccion de x*/
  y=*px;/* y contiene el dato apuntado por px*/

  printf("x=%d\n",x);
  printf("y=%d\n",y);
  printf("*px=%d\n",*px);
 printf("px = %p\n", px);

}
```

File content

File block allocation

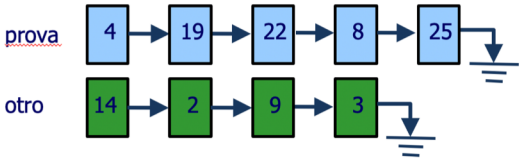
- **Contiguous allocation**
  - A file is allocated as a set of consecutive disk blocks
  - It is defined for every file as the first allocated block address and the file length in blocks

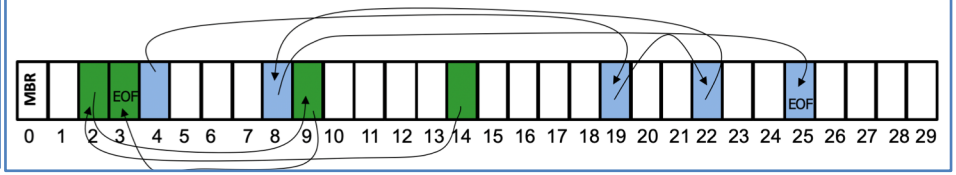
Directory		
File	Start	Length
count	2	2
tr	8	3
mail	16	6
list	24	5



- **Linked allocation**
  - File allocated blocks do not need to be contiguous, then every block is linked to the next by means of a pointer

Directory	
File	Start
prova	4
otro	14

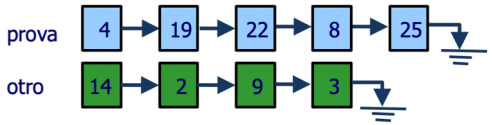


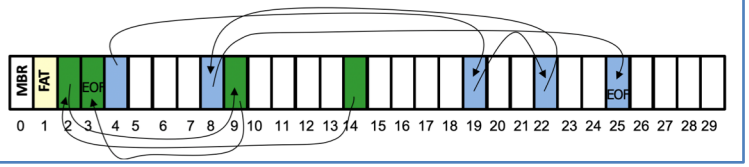


- **FAT** – variation of linked allocation
  - Pointers are not inside the disk blocks but in a disk dedicated area (File Allocation Table)
  - EOF marks the list end

0	reservado
1	reservado
2	9
3	Eof
4	19
5	-
6	-
7	4
8	25
9	3
10	-
11	-
12	-
13	-
14	2
15	-
16	-
17	-
18	-
19	22
20	-
21	-
22	8
23	-
24	-
25	Eof
26	-
27	-
28	-

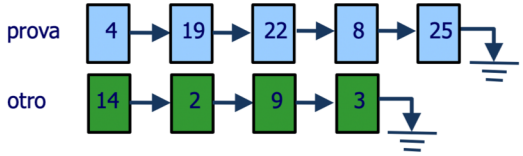
Directory	
File	Start
prova	4
otro	14

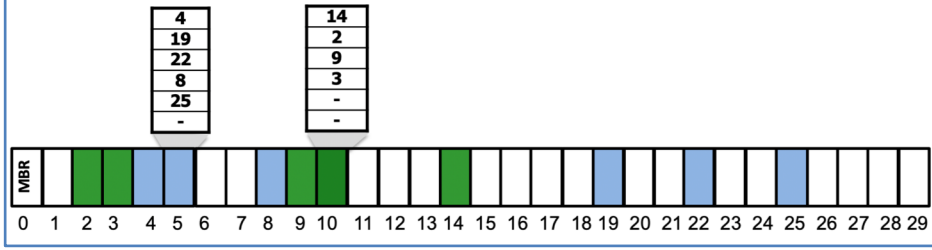




- **Indexed allocation**
  - A block could be index block or data block, a index block contains pointers to data blocks

Directory	
File	Start
prova	5
otro	10





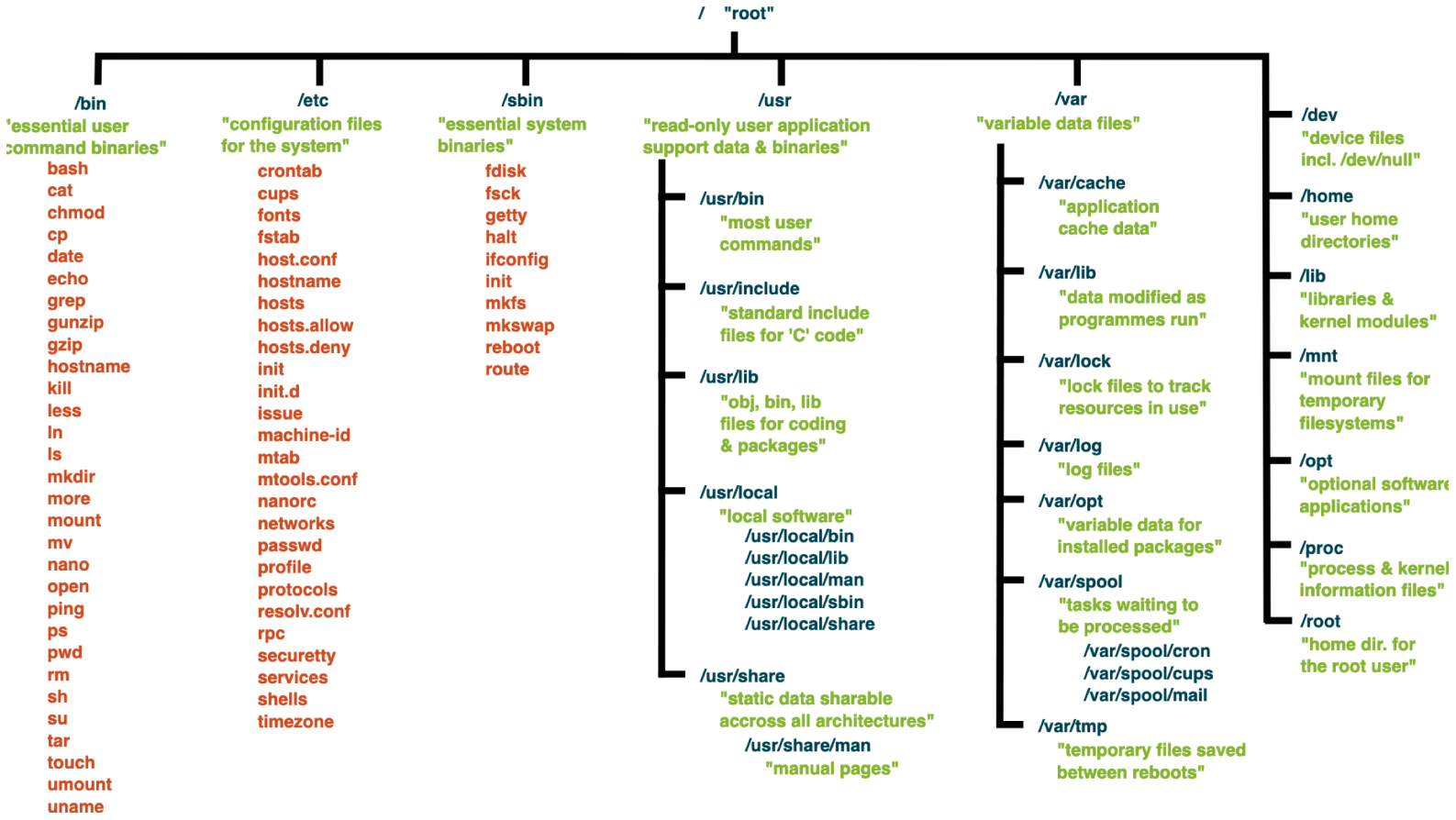
- Allocation types analysis

	Advantages	Disadvantages
Contiguous	It is the more efficient It supports sequential and direct access Stable access speed Perfect for read only devices (CD, DVD, etc)	Complex space management (i.e. finding the best gap, relocation due to file growth, etc) It suffers from external fragmentation (compatation required from time to time)
Linked	It doesn't contrain file growing	It doesn't support direct access It is litte robust against failures
FAT	If FAT is copied in memory then it supports direct acces It makes easy free space management	It FAT doesn't fit in main memory then it lacks from any advantage -> only useful for low capacity devices It is little robust against failures
Indexed	It supports sequential and direct access	It constrains file growing (index block size)
Multilevel Indexed	It doesn't contrain file growing	To locate a block several disk accesses may be required

NOTE. In every case there is **internal fragmentation**, half of last block is wasted in average

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

Unif file hierarchy




```
usuario@lgiifso:/dev$ ls tt*
tty    tty17  tty26  tty35  tty44  tty53  tty62    ttyS12  ttyS21  ttyS30
tty0   tty18  tty27  tty36  tty45  tty54  tty63    ttyS13  ttyS22  ttyS31
tty1   tty19  tty28  tty37  tty46  tty55  tty7     ttyS14  ttyS23  ttyS4
tty10  tty2   tty29  tty38  tty47  tty56  tty8     ttyS15  ttyS24  ttyS5
tty11  tty20  tty3   tty39  tty48  tty57  tty9     ttyS16  ttyS25  ttyS6
tty12  tty21  tty30  tty4   tty49  tty58  ttyprintk ttyS17  ttyS26  ttyS7
tty13  tty22  tty31  tty40  tty5   tty59  ttyS0    ttyS18  ttyS27  ttyS8
tty14  tty23  tty32  tty41  tty50  tty6   ttyS1    ttyS19  ttyS28  ttyS9
tty15  tty24  tty33  tty42  tty51  tty60  ttyS10   ttyS2   ttyS29
tty16  tty25  tty34  tty43  tty52  tty61  ttyS11   ttyS20  ttyS3
usuario@lgiifso:/dev$
```

Unix files

- 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'

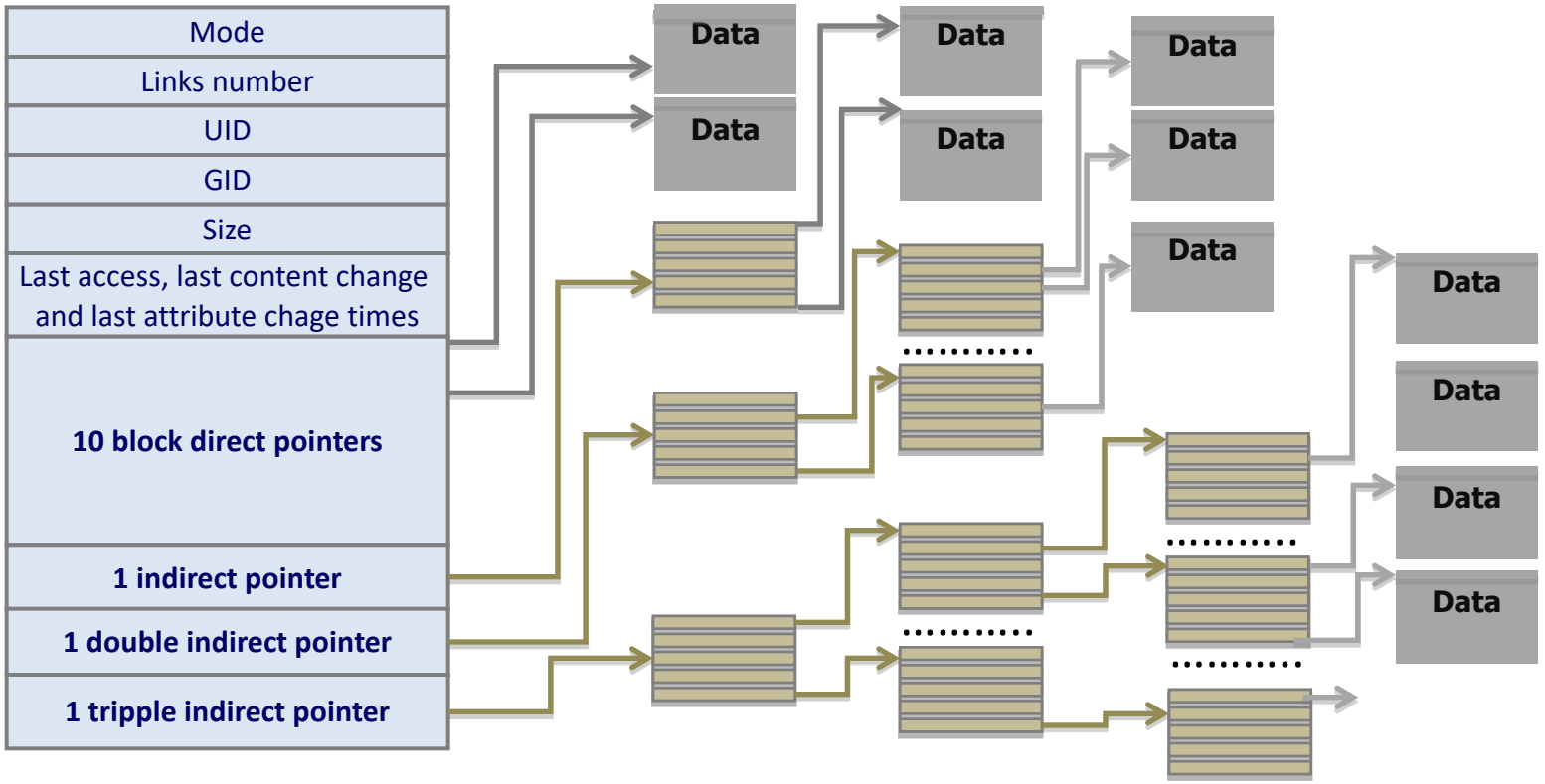
```
usuario@lgiifso:/dev$ ls -al tt*
crw-rw-rw- 1 root tty     5,  0 nov 18 10:45 tty
crw--w---- 1 root tty     4,  0 nov 18 10:37 tty0
crw--w---- 1 root tty     4,  1 nov 18 10:37 tty1
crw--w---- 1 root tty     4, 10 nov 18 10:37 tty10
crw--w---- 1 root tty     4, 11 nov 18 10:37 tty11
```

Unix files

- **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

```
usuario@lgiifso:~/SUTs$ ls -al
total 40
drwxrwxr-x  7 usuario usuario  4096 nov 17 18:51 .
drwxr-xr-x 20 usuario usuario  4096 nov 18 11:06 ..
drwxrwxr-x  2 usuario usuario  4096 nov  4 11:32 exercises
drwxrwxr-x  2 usuario usuario  4096 oct 21 08:59 SUT03
drwxrwxr-x  2 usuario usuario  4096 oct 20 17:22 SUT05
-rw-rw-r--  1 usuario usuario 10240 oct 20 11:20 SUT05.tar
drwxrwxr-x  2 usuario usuario  4096 nov 17 19:49 SUT06
drwxrwxr-x  2 usuario usuario  4096 nov 18 11:36 SUT07
```

- **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

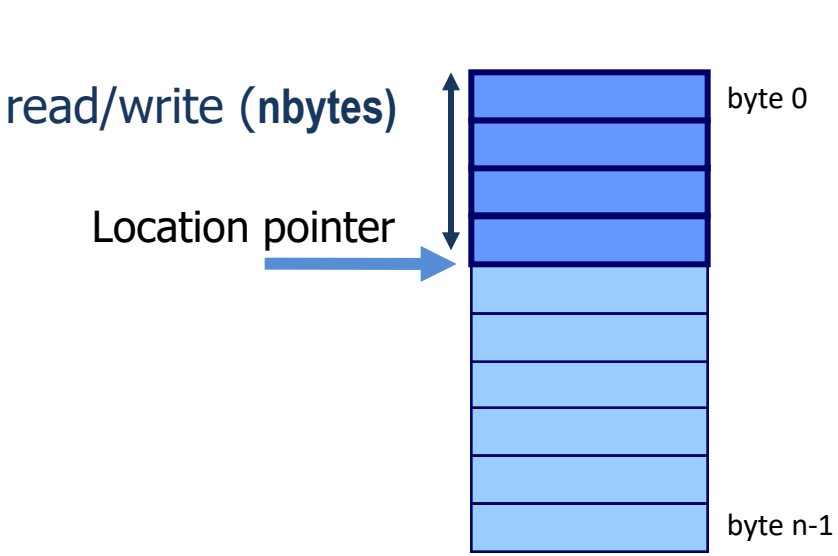


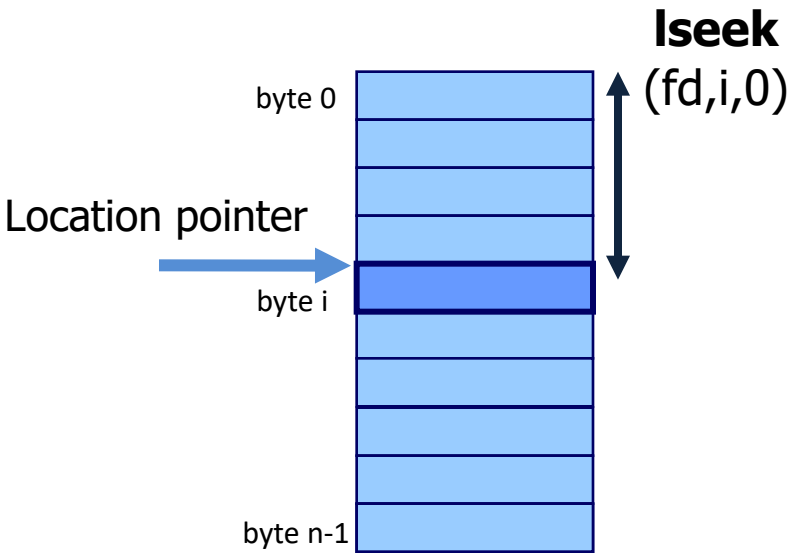
- **Unix directory entry: i-node**
  - OS data str
    can have se
    - Every Un
    - It points
      indirect,

Mode
Links numbe
UID
GID
Size
Last access, last conte and last attribute cha
10 block direct po
1 indirect poin
1 double indirect p
1 tripple indirect p

```
usuario@lgiifso:~/SUTs$ sudo tune2fs -l /dev/sda1
[sudo] contraseña para usuario:
tune2fs 1.44.1 (24-Mar-2018)
Filesystem volume name:   <none>
Last mounted on:          /
Filesystem UUID:          089dfed0-bda3-4d54-bfbb-c527483ff3cf
Filesystem magic number:  0xEF53
Filesystem revision #:    1 (dynamic)
Filesystem features:      has_journal ext_attr resize_inode dir_inde
ds_recovery extent 64bit flex_bg sparse_super large_file huge_file d
_isize metadata_csum
Filesystem flags:         signed_directory_hash
Default mount options:    user_xattr acl
Filesystem state:         clean
Errors behavior:          Continue
Filesystem OS type:       Linux
Inode count:              983040
Block count:              3931648
Reserved block count:     196582
Free blocks:              1776255
Free inodes:              687557
First block:              0
Block size:               4096
Fragment size:            4096
Group descriptor size:    64
Reserved GDT blocks:      1024
```

- **File structure**
  - Array 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)**





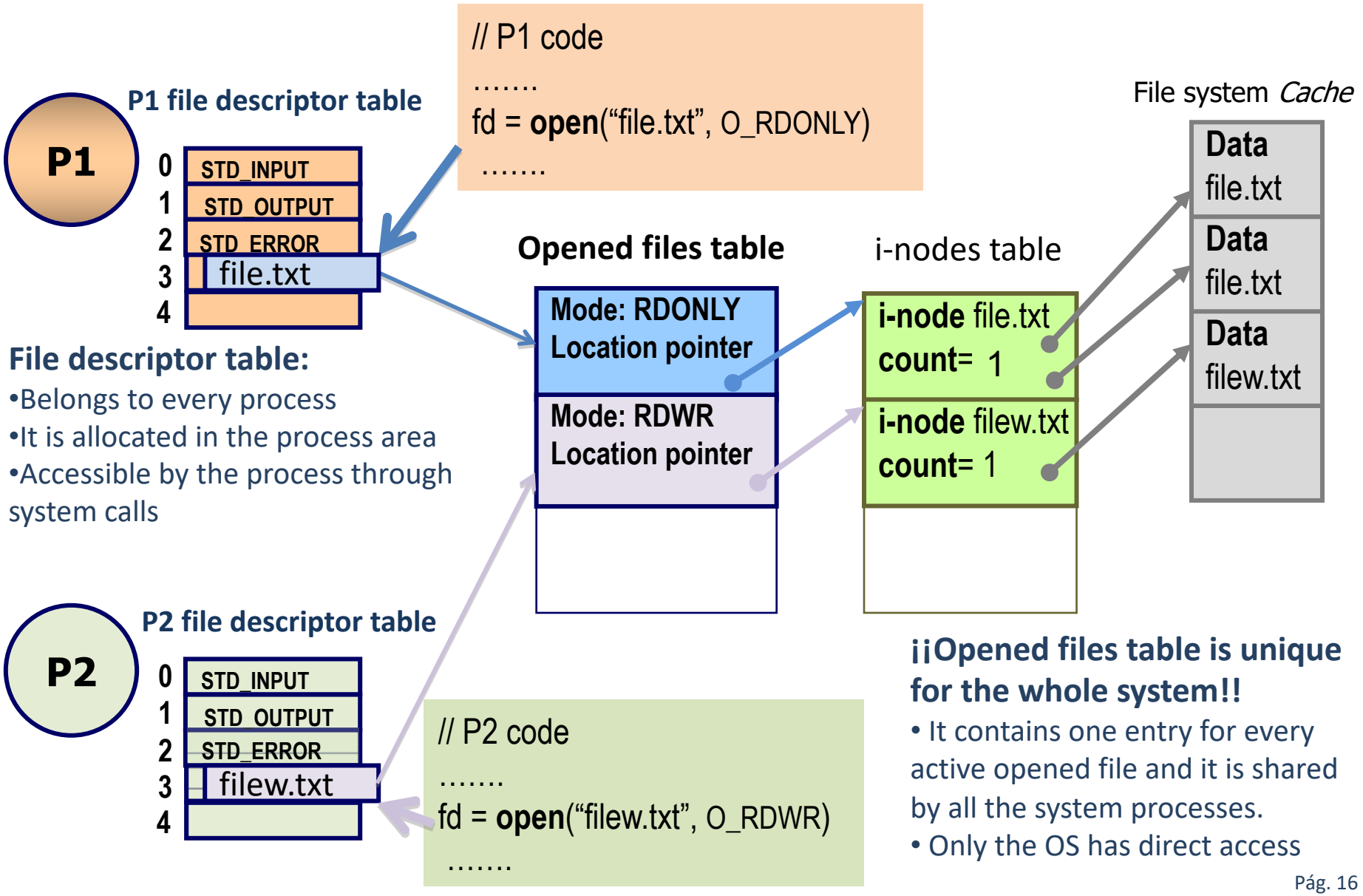
- **File descriptor**
  - To read or write a file it must be first opened and last closed

```
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 descriptor table:

- Belongs to every process
- It is allocated in the process area
- Accessible by the process through system calls

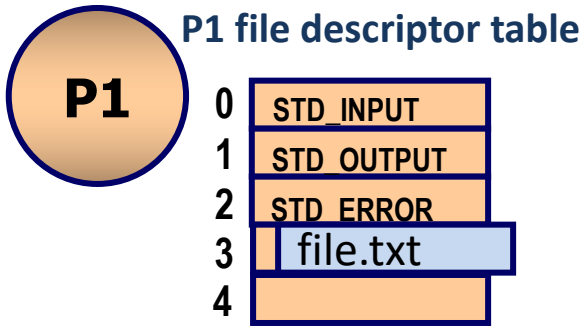
¡¡Opened files table is unique for the whole system!!

- It contains one entry for every active opened file and it is shared by all the system processes.
- Only the OS has direct access

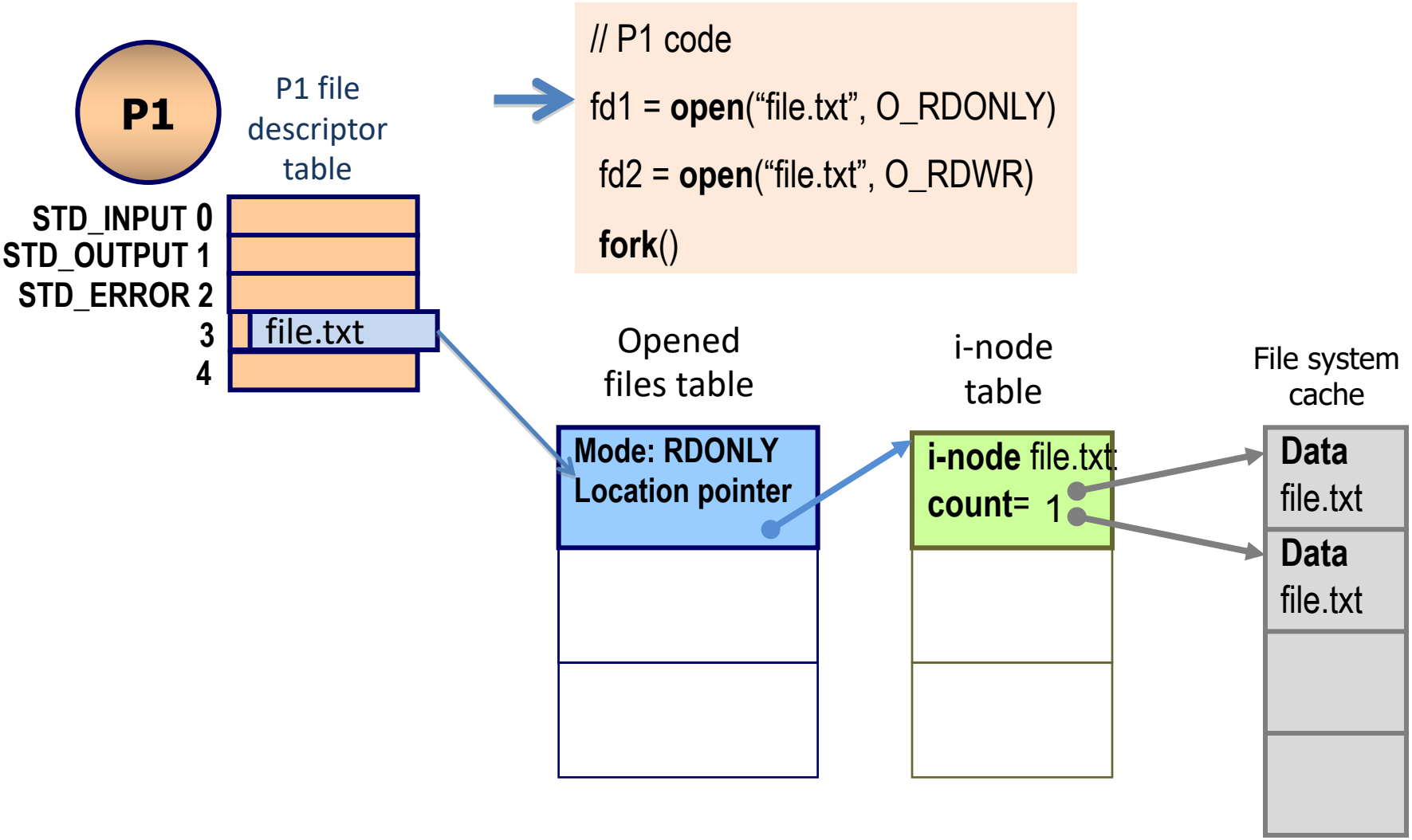
- File descriptors and standard I/O
  - The first three file descriptors in a process have a proper name:

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

  - By default these file descriptors are associated to the console
    - Console devices are /dev/ttyn 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 outputand writes the error message "No such file or directory" in the standard error

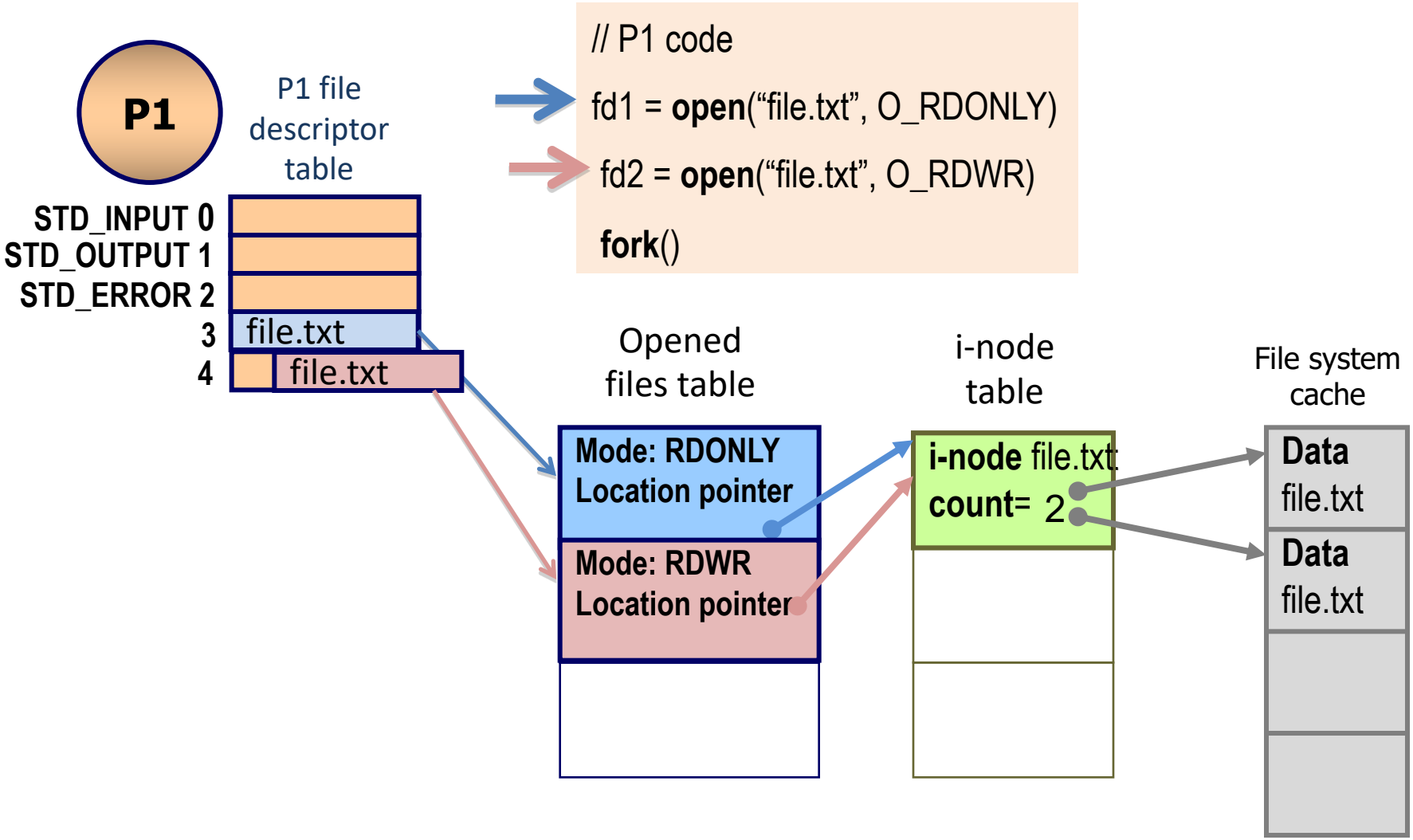


- **Inheriting the file descriptor table**

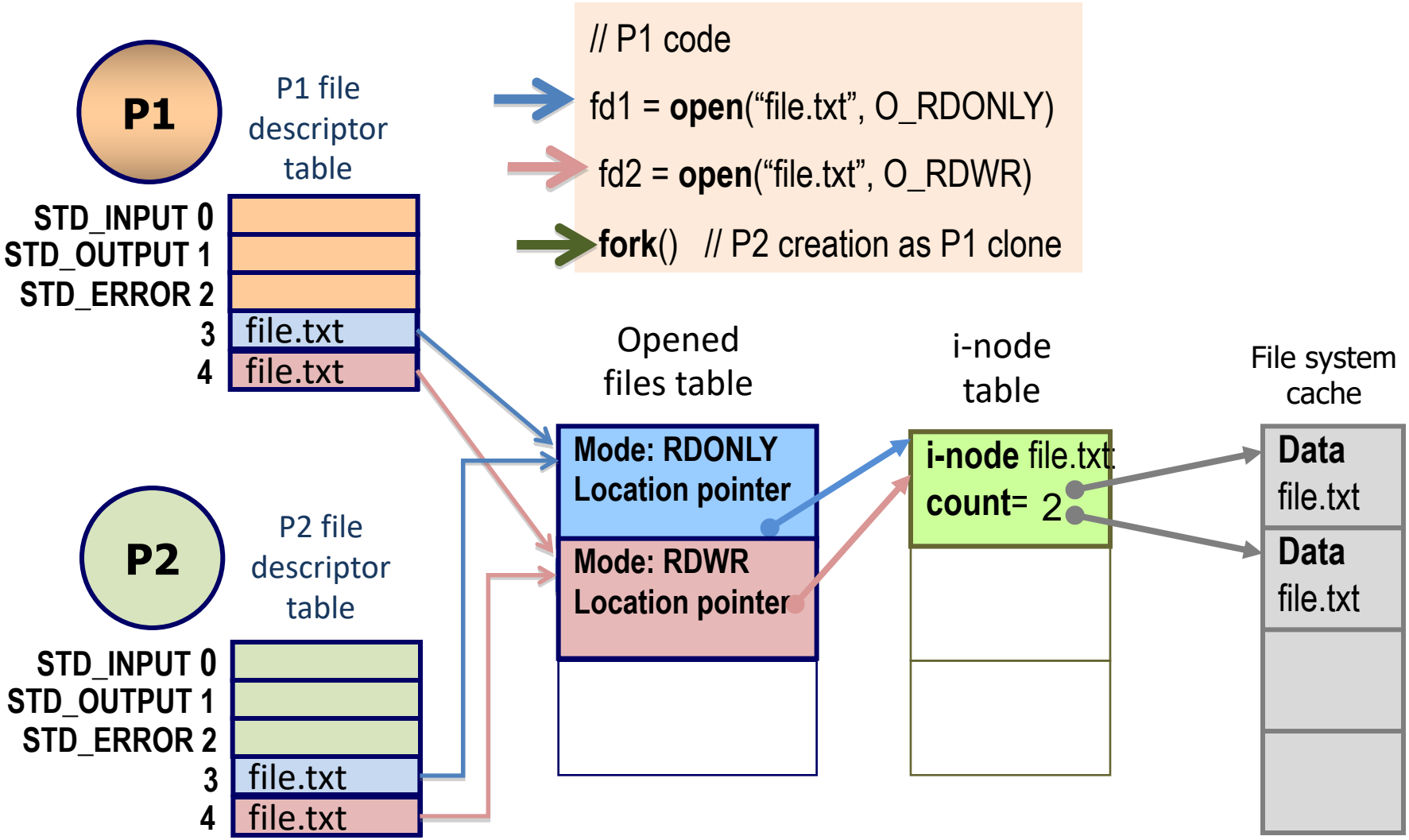


¡¡There is only one opened files table in the system!!

- **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

Unix file system calls

- 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 formated 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`

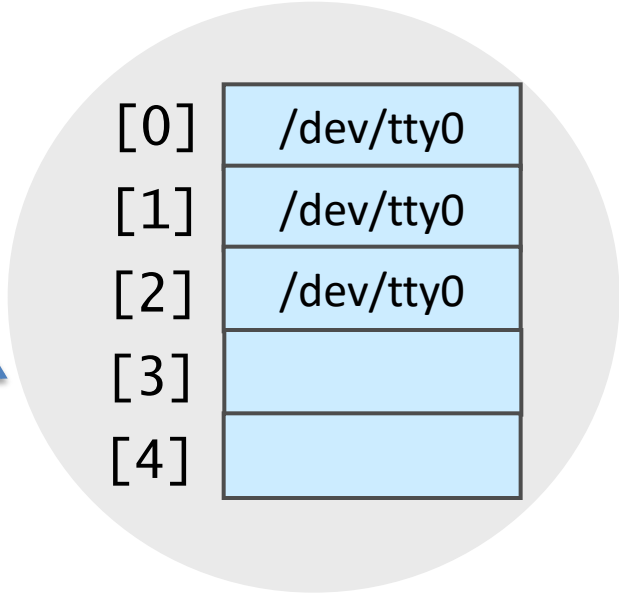
  `O_RDONLY, O_WRONLY, O_RDWR`
  `O_CREAT, O_EXCL, O_TRUNC, O_APPEND`
- `Mode` `Examples: 0755, 04755`

  `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`

Unix file system calls

- 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

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

Unix file system calls

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

stat : display file or file system status

```
#include <unistd.h>

int  stat(int fd)
```

Description

- Display file info
- 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

```
usuario@lgiifso:~/SUTs/SUT07$ stat kk.txt
  Fichero: kk.txt
  Tamaño: 1234592       Bloques: 2416       Bloque E/S: 4096   fichero regular
Dispositivo: 801h/2049d Nodo-i: 280231      Enlaces: 1
Acceso: (0600/-rw-------)  Uid: ( 1000/ usuario)   Gid: ( 1000/ usuario)
Acceso: 2020-11-17 19:48:48.245523868 +0100
Modificación: 2020-11-17 19:48:27.353523350 +0100
      Cambio: 2020-11-17 19:48:27.353523350 +0100
    Creación: -
```

Content

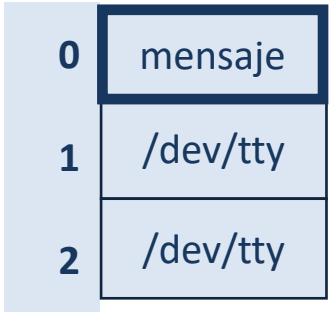
- Unix files
- Unix file system calls
- **Redirections and pipes**
- Redirection and pipe system calls
- C examples

Redirections and pipes

- Standard I/O redirection from the shell

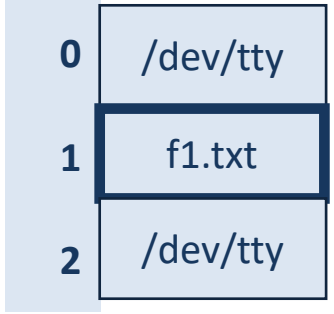
—Standard input redirection

```
$ mail gandreu < mensaje
```



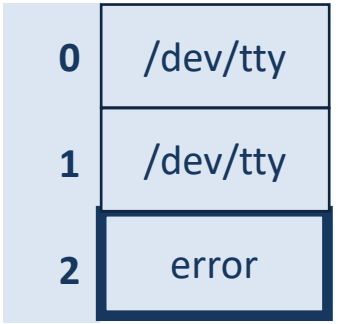
—Standard output redirection

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



—Standard error redirection

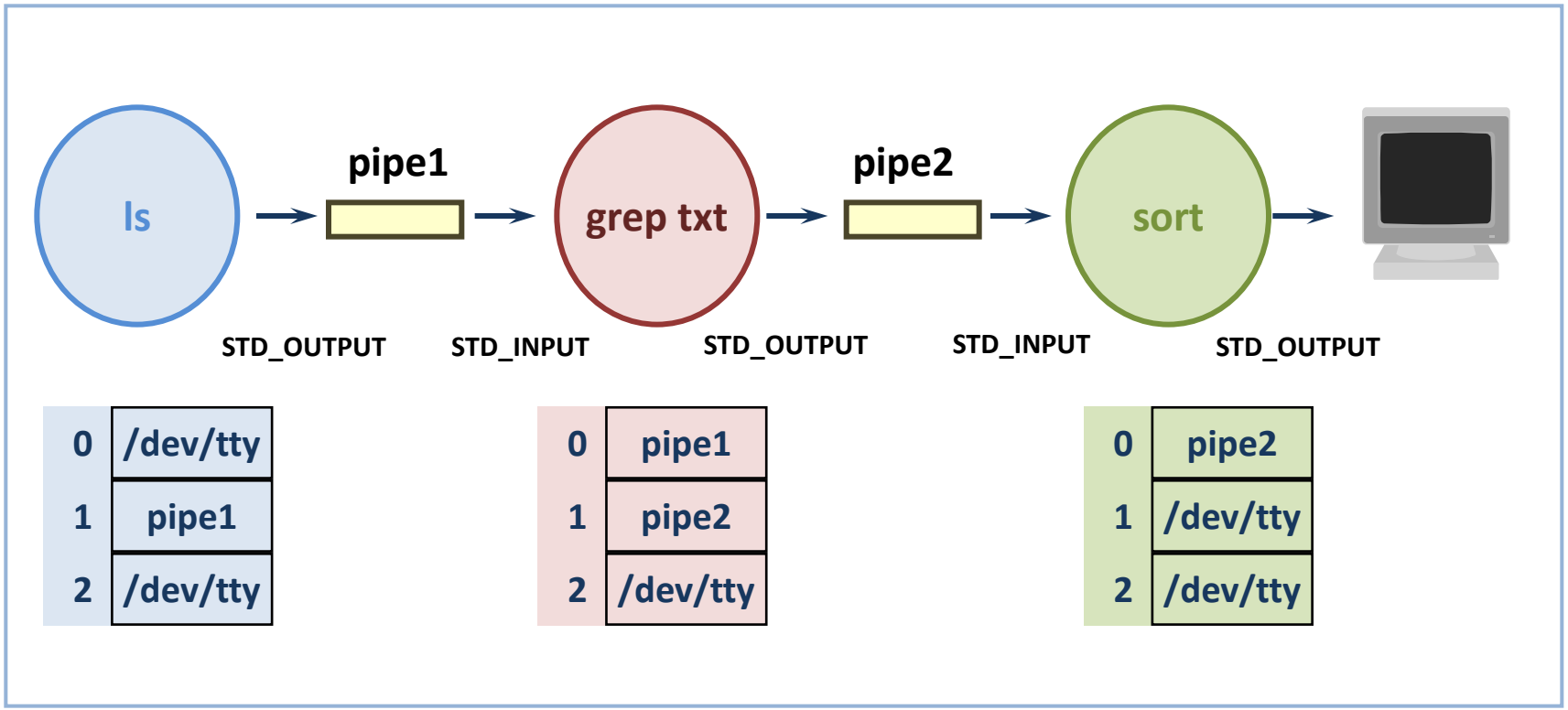
```
$ gcc prg.c -o prg 2> error
```



Redirections and pipes

- **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



Content

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

Redirection and pipe system calls

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

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

dup, dup2: duplicating a file descriptor

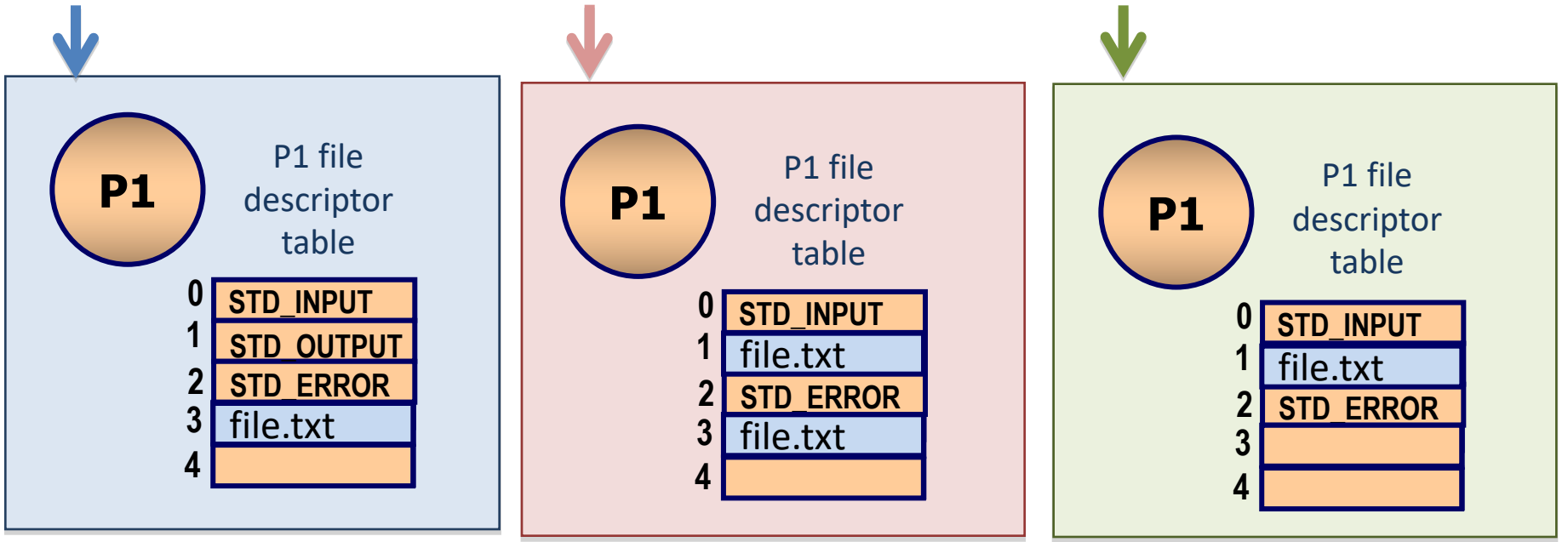
```
#include <unistd.h>
int dup(int fd)
int dup2(int oldfd, int newfd)
```

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

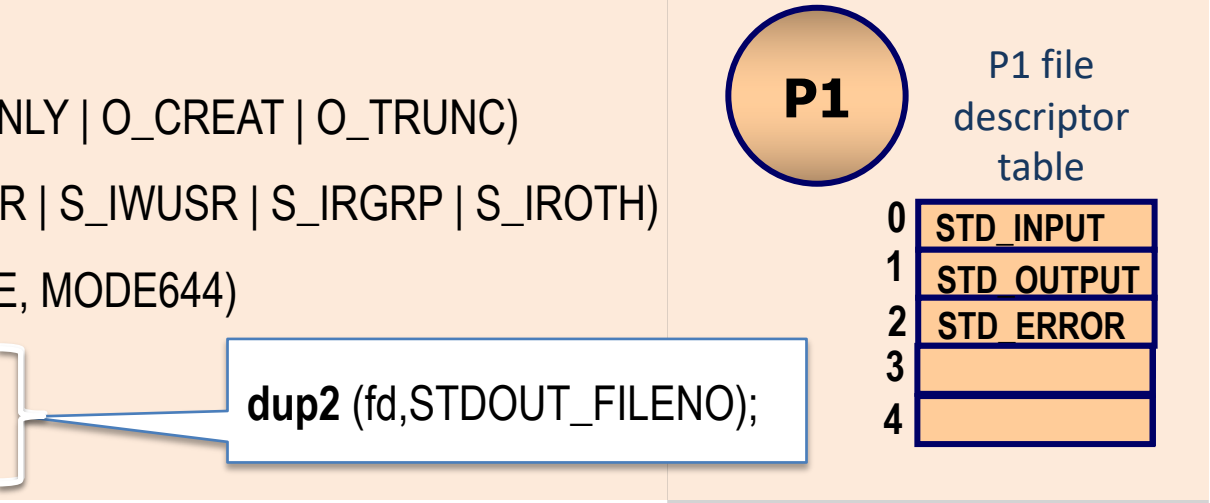
```
// P1 code
#define NEWFILE  (O_WRONLY | O_CREAT | O_TRUNC)
#define MODE644  (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
fd  = open("file.txt", NEWFILE, MODE644)
dup2 (fd,STDOUT_FILENO);
close (fd);
```

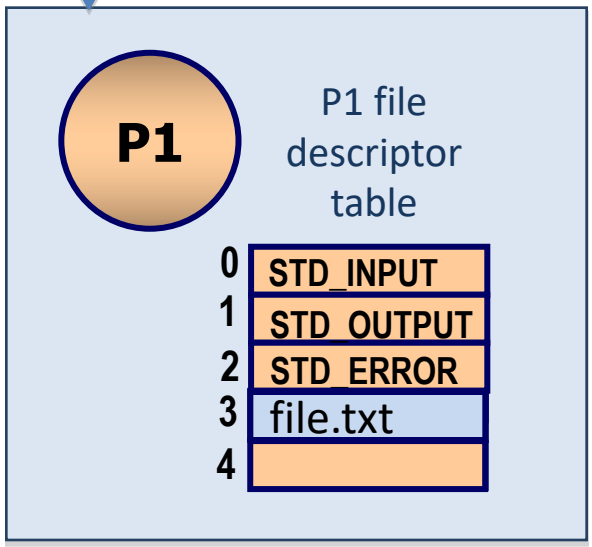


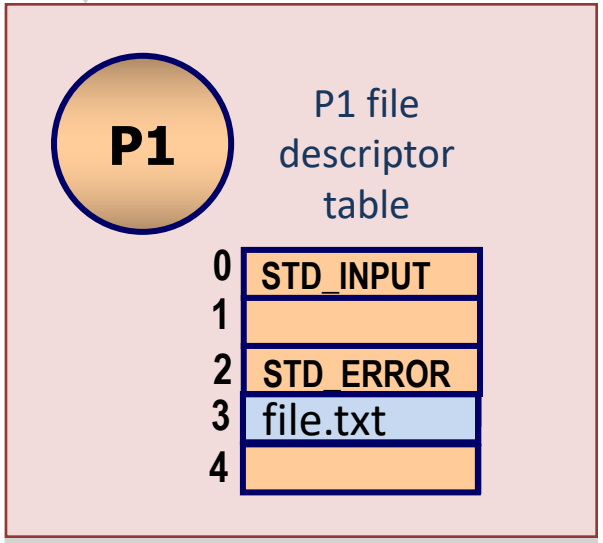
Example: dup

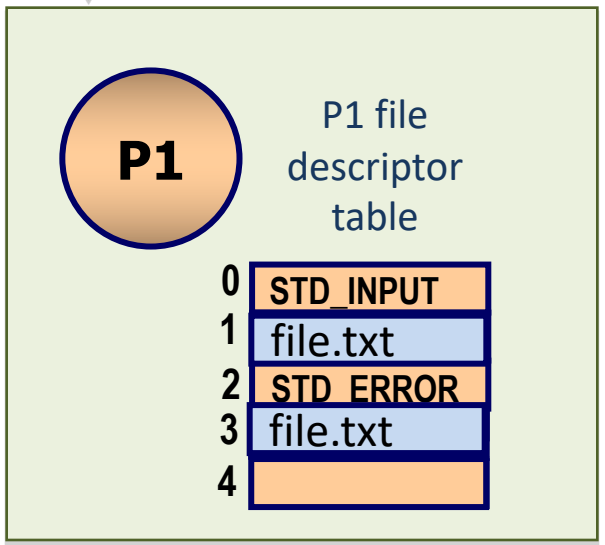
```
// P1 code
#define NEWFILE  (O_WRONLY | O_CREAT | O_TRUNC)
#define MODE644  (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
fd  = open("file.txt", NEWFILE, MODE644)
close (STDOUT_FILENO);
dup (fd);
```

dup2 (fd,STDOUT_FILENO);









pipe: creating a pipe

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

Description

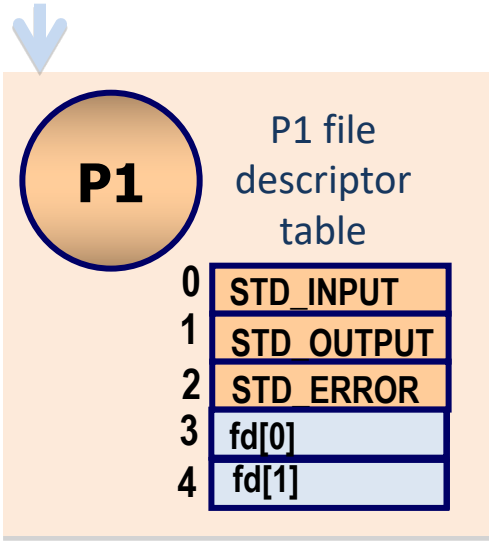
- It creates a *pseudofile* (pipe) structured as a FIFO queue of bytes that is initially empty
  - Pipes are a interprocess communication provided by UNIX
  - The maximum pipe capacity is limited by the OS
- After **pipe** call, **fildes[0]** is a file descriptor to read the pipe and **fildes[1]** is 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)

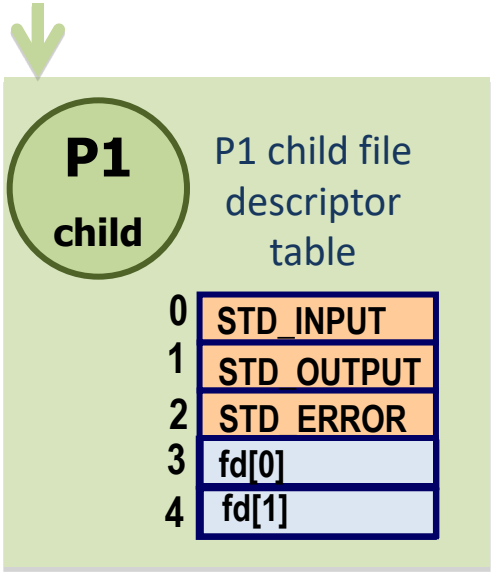
Pipe operation

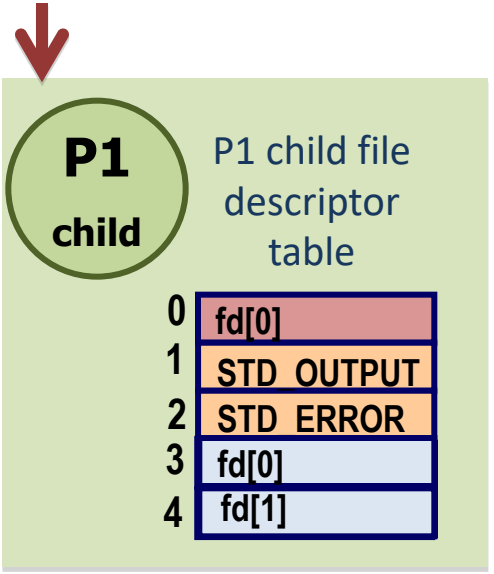
- **read**
  - If there are bytes available, at most the requested **nbytes** are read
  - It pipe is empty, **read** suspends the calling process until bytes are available in the pipe
  - When there is no pipe writing descriptor (bellowing to the reading process or any other one) **read** doesn't suspend 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 (bellowing 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]);
  ...
}
```

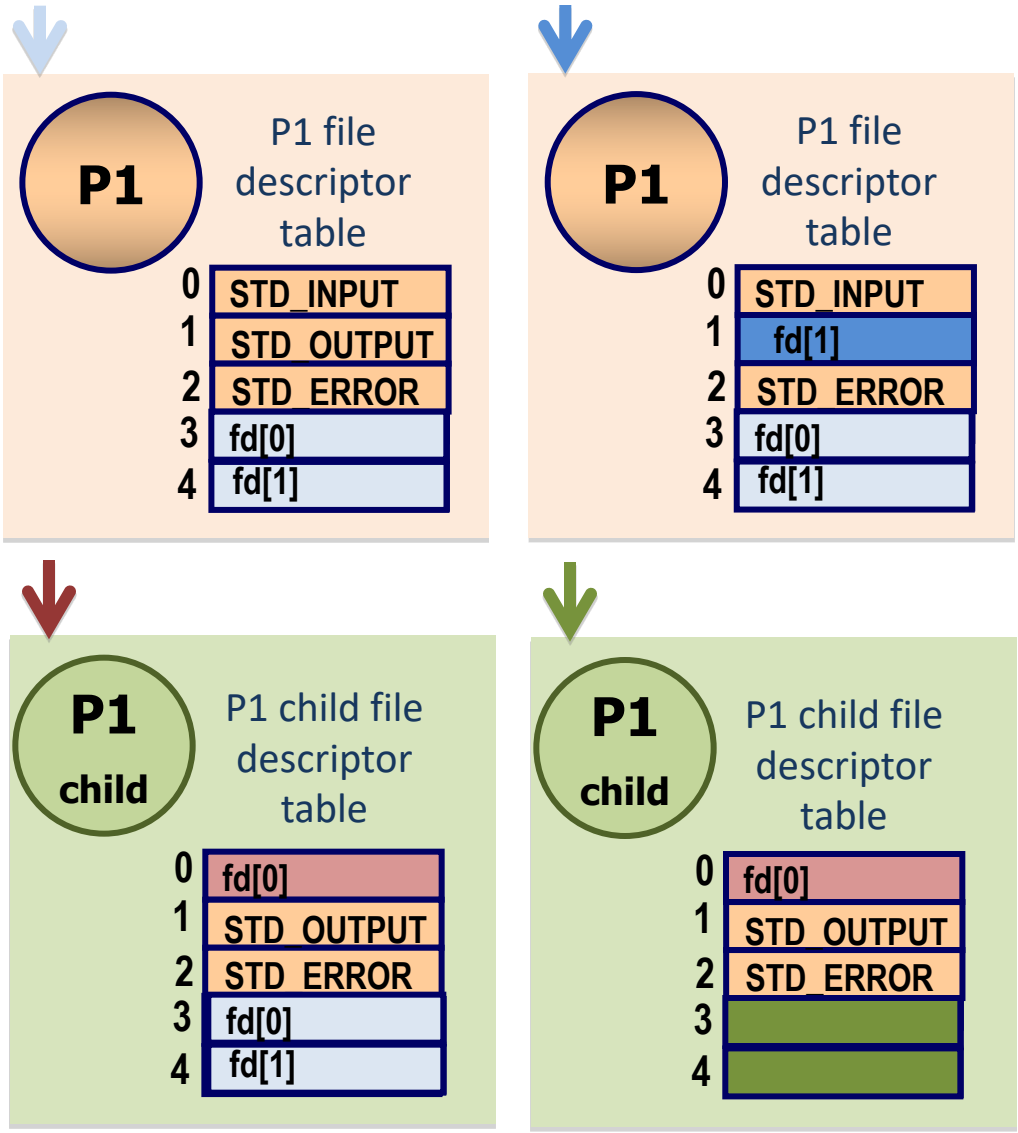






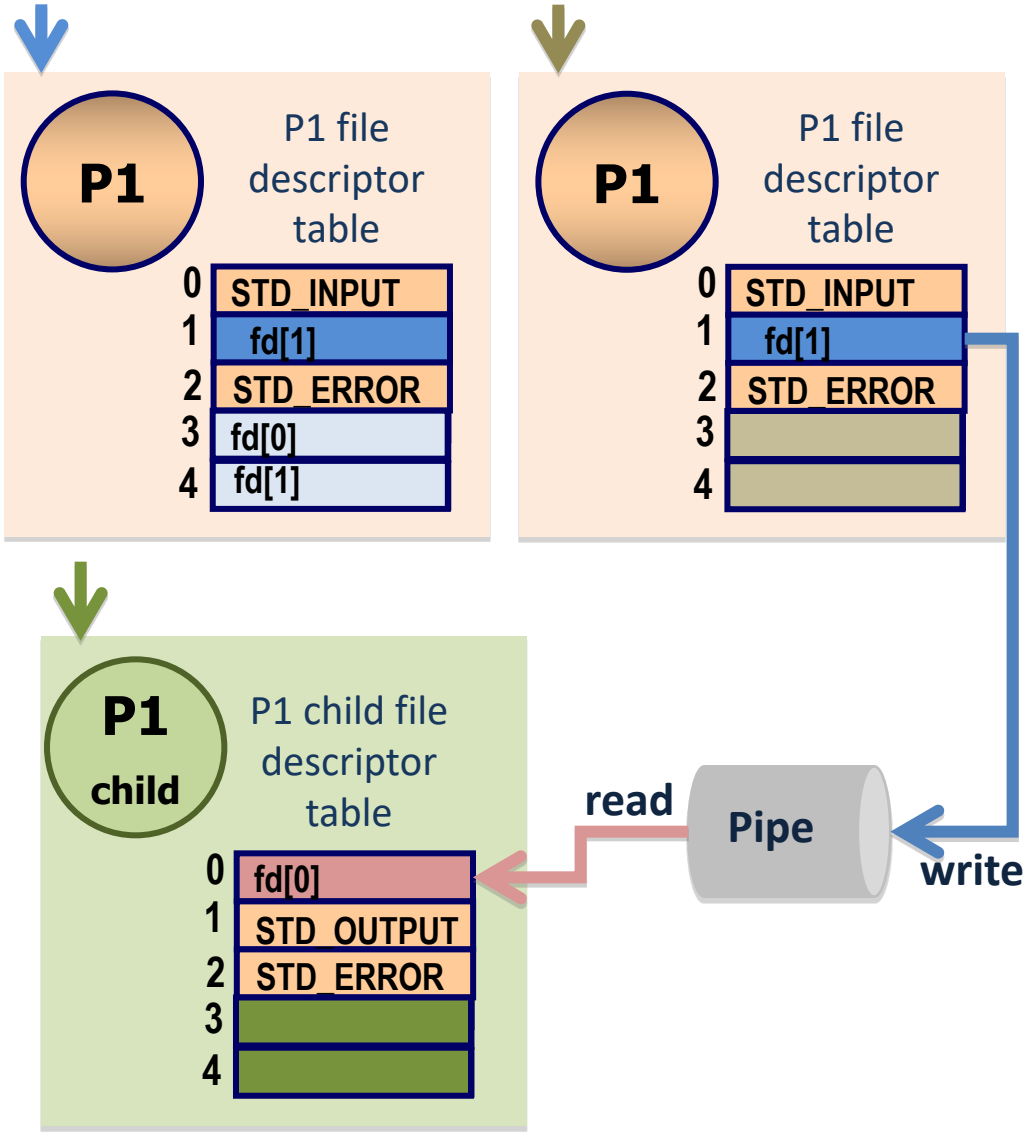
Redirection and pipe system calls

```
// 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]);
  ...
}
```



Content

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

C examples

- **Example_01:** open, write and read

```
#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)
```

```
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);
}
```

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_02:** dup2

```
#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)
```

```
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;
}
```

To compile and execute do:

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

- **Example_03:** pipe

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

```
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);
}
```

To compile and execute do:

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