Fundamentos de los Sistemas Operativos

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

fSO

Lab session 6
File system calls in UNIX

Contenido

1. Objectives

The Unix file system presents a single interface for managing devices and files. Calls to the system related files are widely used in application programming. The lab session focuses mainly on the use of them for communication between processes and I/O redirection. The objectives are:

- Working with Unix system calls for file handling: open, close, read, write, pipe and dup2.
- Studying the mechanisms of redirection of I/O to regular files and pipes (pipe).
- Understanding the inheritance mechanism that allows the communication between processes.

2. File management in Unix

The handling of files in UNIX follows the session model. To work with files first you must open it with the open() call. Open() returns a file descriptor, a positive integer that identifies the file in future operations. This value is the position of the table of descriptors of the process that contains the pointer to the open files system table. Finally we must close the file with the close() call to release the resources allocated to the file.

UNIX uses the same interface for working with files than with I/O devices. Descriptors 0, 1 and 2, are set to standard I/O devices, they are inherited from process parent to child through the inheritance mechanism. Descriptor 0 corresponds to standard input (default keyboard), 1 descriptor to stdout (default screen) and descriptor 2 to the display device of errors (default screen). The use of descriptors allows the system to be much more efficient when working with files.

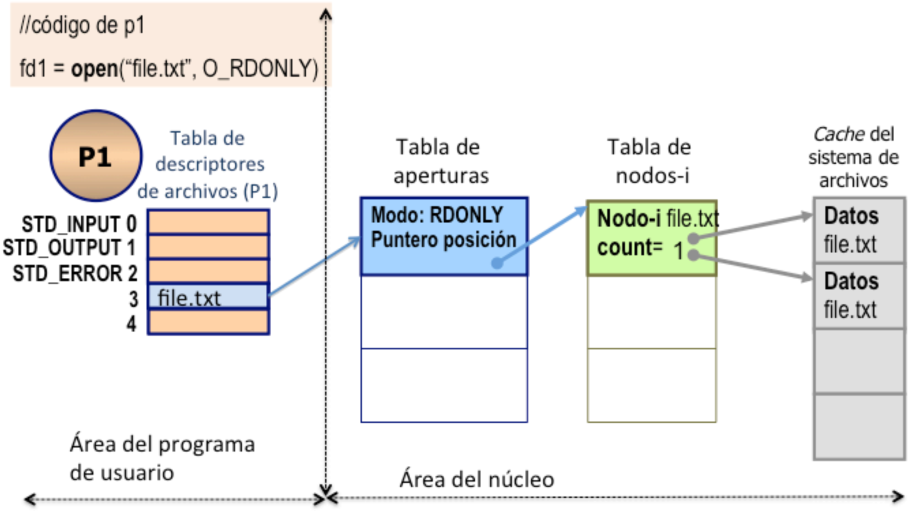

Figure 1: File descriptors table for process after calling open and OS file management structures

UNIX access files sequentially, you can do direct access using the lseek() call. Every file opening has a pointer's position that increases with each read or write the number of bytes equal to the number of bytes read or written. The lseek() call allows you to position the pointer at a given position in the file.

3. Opening and closing files

Figure 1 graphically represents the effect of performing an open() call. UNIX processes receive their table of file descriptors through the inheritance mechanism. Descriptors 0, 1 and 2 correspond to the standard input (STDIN), standard output (STDOUT) and standard error output (STDERR). UNIX uses the *open()* call to assign a file descriptor to a file or a physical device.

3.1 Exercise 1: File descriptors, function open()

The content of the file *descriptor.c* provided with this lab session material is as follows:

```
// descriptor.c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>

int main (int argc,char *argv[]) {
    int fda,fdb;

    if (argc!=2) {
        fprintf(stderr, "Required read/write file \n");
        exit(-1);
    }

    if ((fda=open(argv[1],O_RDONLY))<0)
        fprintf(stderr,"Open failed \n");
    else
        fprintf(stderr, "Read %s descriptor = %d \n",argv[1],fda);

    if ((fdb=open(argv[1],O_WRONLY))<0)
        fprintf(stderr, "Open failed \n");
    else
        fprintf(stderr, "Write %s descriptor=%d \n",argv[1],fdb);
    return(0);
}
```

To run *desriptor.c* put as a parameter the name of the file to open. Compile and run it:

```
$gcc descriptor.c -o descriptor

$./descriptor descriptor.c
```

Question 1: Analyze the code and the result of the execution and answer the following:

1. What variables correspond to file descriptors in the proposed code? fda and fdb
2. Explain the number assigned by the system to the variable fda stdin is descriptor 0, stdout is 1, stderr is 2, so the next one free is 3, which is assigned to fda
3. Explain the number assigned by the system to the variable fdb After assigning 3 to fda, the next free file descriptor is 4, which is assigned to fdb

3.2 Exercise 2: Standard output descriptor, function close()

The function *close(fd)* releases the descriptor fd from the file descriptors table. In this exercise confirm that the descriptor of standard output, and therefore the terminal, corresponds to descriptor number 1, to accomplish this work with the code in file *descriptor_salida.c*

```
//descriptor_output.c

#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main (int argc,char *argv[]) {
    char *men1="men1: Writing in descriptor 1 (std_output)\n";
    char *men2="men2: Writing in descriptor 2 (std_error)\n";
    char *men3="men3: Writing in descriptor 1 (std_output)\n";
    char *men4="men4: Writing in descriptor 2 (std_error)\n";
    char *men5="men5: Writing in descriptor 1 (std_output)\n";
    char *men6="men6: Writing in descriptor 2 (std_error)\n";

    write(1,men1, strlen(men1));
    write(2,men2, strlen(men2));
    close(1);
    write(1,men3, strlen(men3));
    write(2,men4, strlen(men4));
    close(2);
    write(1,men5, strlen(men5));
    write(2,men6, strlen(men6));
    return(0);
}
```

Compile *descriptor_output.c* and run it with:

```
$gcc descriptor_output.c -o descriptor_output

$./descriptor_output
```

Question 2: Analyze the code and the result of the execution and answer the following:

1. What messages are printed on the screen?

   men1 (std_output), men2 (std_error), men4 (std_error)

2. Explain why are not printed every one of the missing messages

   When file descriptor 1 is closed, messages that are printed to stdout are not printed, as it is closed. When file descriptor 2 is closed, the same happens with stderr.

3. Fill the table of opened file descriptors corresponding to the process just before return (0)

0	stdin
1	
2	
3	
4	

4. Inheritance of file descriptors

When a call is made by a process to fork(), the child process created inherits lot of attributes from his parent, working directory, scheduling attributes, etc; and the table of open file descriptors. Because of this inheritance processes share the read/write pointer position of the opened file before calling to fork().

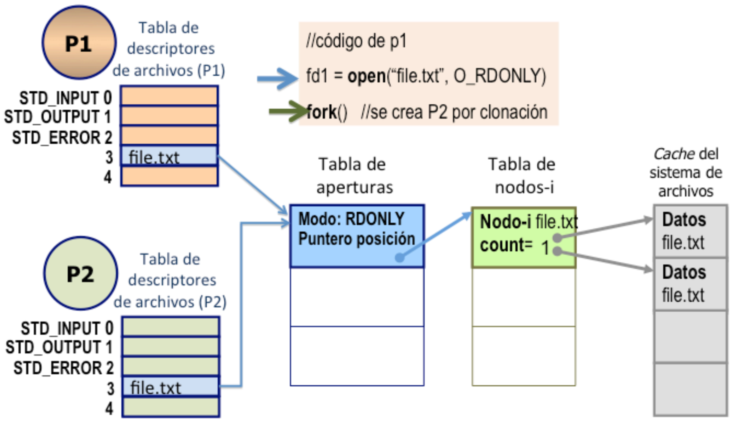

Figure 2: Inheritance of the file descriptors table and their relationship with the system table of opened files and other system structures.

4.1 Exercise 3: parent and child processes share a file

The *share_file.c* code, establishes a communication between processes by using a file.

```
// share_file.c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main (int argc,char *argv[]) {
    int fd;
    pid_t pid;
    mode_t fd_mode=S_IRWXU;// file permissions
    char *parent_message = "parent message \n";
    char *child_message = "child message \n";

    fd = open("messages.txt",O_RDWR | O_CREAT,fd_mode);
    write(fd,parent_message,strlen(parent_message));

    pid = fork();
    if (pid == 0) {
        write(fd,child_message,strlen(child_message));
        close(fd);
        exit(0);
    }
    wait(NULL);
    write(fd,parent_message,strlen(parent_message));
    close(fd);
    return(0);
}
```

Compile *share_file.c* , run it and displays the contents of file *messages.txt.*

```
$gcc share_file.c -o share_file
$./share_file
$ cat messages.txt
```

Question 3: Analyze the code and the result of the execution and answer the following:

1. What is the content of messages.txt file?

   parent message
   child message
   parent message

2. Both the parent process and the child process have written its message in messages.txt file. What mechanisms/calls made it possible?

   They wrote their content making use of the write calls. The second parent message didn't overwrite the child's message because the same file position pointer is accessed and modified by both the parent and the child.

3. Fill the tables of opened file descriptors corresponding to the process parent and child just before executing close (fd)

0	stdin
1	stdout
2	stderr
3	
4	

0	stdin
1	stdout
2	stderr
3	
4	

5. Redirection: function dup2

Redirection of standard input allows a process to "read" data from a source other than the terminal, through the descriptor 0. For example:

```
$ mailx fso10 < mensaje
```

The message to be sent by mailx application in the “message” file.

Standard output redirection allows a process to "write" data to a destination other than the terminal, through the descriptor 1. For example:

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

The result of command echo is written to the f1.txt file.

Standard error output redirection allows a process to "write" error messages on a destination other than the terminal, through the descriptor 2. For example:

```
$ gcc program1.c -o program 2 > errors
```

where errors of compilation of “program1.c” file is written to the file “errors”.

Redirection of input, output, or standard error output in UNIX is performed by calling function dup2().

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

Dup2 closes descriptor **newfd** and copies the pointer associated with the descriptor **oldfd** into **newfd**. In the annex of this guide is described in detail dup2().

5.1 Exercise 4: standard output redirection to a file

In this exercise you will practice how to use dup2 (), so that everything that is written on the standard output is redirected to a file. For this you must use the code provided with the practice in the file *redir_output.c* and shown below:

```
//codigo redir_output.c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

int main (int argc,char *argv[]) {
    int fd;
    char *arch=”output.txt”;
    mode_t fd_mode=S_IRWXU;// file permissions

    fd=open(arch,O_RDWR | O_CREAT,fd_mode);
    if (dup2(fd,STDOUT_FILENO)==-1) {
        printf("Error calling dup2 \n");
        exit(-1);
    }
    fprintf(stdout,"output redirected\n");
    fprintf(stderr,"error not redirected\n");
    fprintf(stderr,"Check file %s \n", arch);
    close(fd);
    return(0);
}
```

Compile *redir_output.c* , run it and displays the contents of the file *output.txt*

```
$ gcc redir_output.c -o redir_output
$ ./redir_output
$ more output.txt
```

Question 4: Analyze the code and the result of the execution and answer the following:

1. Explain using the instructions in the code the contents of file “output.txt”

   The file contains "out: Output redirected", because stdout was closed and the file descriptor of the opened "output.txt" file was set to occupy the place of stdout by using dup2.

2. Explain why the open() function is called with flags "O_RDWR|O_CREAT"

   RDWR stands for read and write, and CREAT for create file if it doesn't exist.

3. Fill the opened file descriptors table corresponding to the process just before the "if (dup...)" sentence

0	stdin
1	stdout
2	stderr
3	
4	

4. Fill the opened file descriptors table corresponding to the process just before return (0)

0	stdin
1	
2	stderr
3	
4	

5.2 Exercise 5: standard output redirection to a file with exec()

The table of opened file descriptors of a process does not change after calling exec(), the process preserves opened files and redirections that had been conducted prior to exec().
Write a program called *ls_redir.c* that when executing it the command "ls -la" will be executed redirecting the output to the file “ls_output.txt”, as if it was the shell command:

```
$ls -la > ls_output.txt
```

To do this copy *redir_output.c* into *ls_redir.c*. Edit *ls_redir.c* and add the execl() call in the apropriate location, make sure that the output is redirected to “ls_output.txt”.

```
execl(“/bin/ls”,”ls”,”-la”,NULL)
```

Compile *ls_redir.c* , run it and display the content of file “ls_ouput.txt”.

```
$ gcc ls_redir.c -o ls_redir
$ ./ls_redir
$ more ls_output.txt
```

Question 5: Analyze the code and the result of the execution and answer the following:

After the execution explain where is stored the contents of the working directory

They are stored inside ls_output.txt

5.3 Exercise 6: standard input redirection from a file

Write a program called *cat_redir.c* that when executing it the command “cat” will be executed redirecting its input to file “ls_output.txt”, as in the following shell command:

```
$ cat < ls_output.txt
```

To do this copy *ls_redir.c* into *cat_redir.c*. Edit *cat_redir.c* and change it to be sure that the file *ls_output.txt* is opened only for reading, when calling open().

Compile *cat_edir.c*, and run it. Make sure that the “ls_output.txt” file exists

```
$ gcc cat_redir.c -o cat_redir
$ ./cat_redir
```

Question 6: Analyze the code and the result of the execution and answer the following:

What has been necessary to change in the code of exercise 5 to carry out exercise 6?

The execl call is now `execl("/bin/cat", "cat", NULL)

O_RDWR has been changed to O_RDONLY

The dup2 call is now `dup2(fd, STDIN_FILENO)`

6. Pipes creation: function pipe()

Unix pipes are a mechanism for interprocess communication. A pipe is an unnamed file, with two descriptors one for reading and one for writing. These two descriptors allow using different positions for read and write pointers, so that the reading pointer advances only when read and write operations are performed when write operations are performed. On UNIX the system call to create a pipe is *pipe()* (see the annex):

```
int pipe(int fildes[2])
```

The pipe() call creates a buffer with FIFO management, the descriptor fildes[0] is used for reading (input) and fildes[1] for writing (output). Pipes do not have any external name, so they can only be used via their file descriptors by the process that creates them and by processes that inherit the table of descriptors after calling fork().

Processes must share the pipe and redirect their input or output to the pipe. For example, the command line:

```
$ ls | grep txt
```

This command line displays on the standard output the file names in the current directory containing the string txt.

6.1 Exercise 7: communicating two processes using pipe()

Write a program that will execute the following command:

```
$ ls -la | wc -l
```

As shown in Figure-3 the ls command must redirect its output to the pipe, while the wc command must redirect their input to read from the pipe.

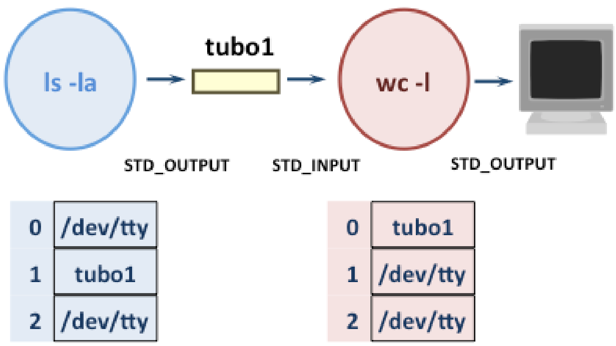

Figure 3: Redirection with a pipe implemented in exercise 7.

Therefore the student must develop a code where:

1. The parent process creates a pipe
2. The parent process creates a child process
    a. The child redirects the pipe and closes descriptors
    b. The child changes its memory image executing command **ls**

3. The parent process creates another child process
    a. The child redirects the pipe and closes descriptors
    b. The child changes its memory image executeing command ws
4. The parent process closes descriptors and waits for its children to finish

As a starting point the file *a_pipe.c* is given. It contains comment lines that the student should replace with instructions and system calls to complete the program.

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

int main (int argc,char *argv[]) {
    int i;
    char* arguments1 [] = { "ls", "-la", 0 };
    char* arguments2 [] = { "wc", "-l", 0 };
    int fildes[2];
    pid_t pid;
    // Parent process creates a pipe
    if ((pipe(fildes)==-1)) {
        fprintf(stderr,"Pipe failure  \n");
        exit(-1);
    }
    for (i=0;i<2;i++) {
        pid=fork(); // Creates a child process
        if ((pid==0) && (i==0)) {
            // Child process redirects its output to the pipe

            // Child process closes file descriptors

            // Child process changes its memory image
            if ( execvp("ls",arguments1)<0) {
                fprintf(stderr,”ls not found \n”);
                exit(-1);´
            }
        } else if ((pid==0) && (i==1)) {
            // Child process redirects its input to the pipe

            // Child process closes pipe descriptors

            // Child process changes its memory image
            if (execvp("wc",arguments2)<0) {
                fprintf(stderr,”wc not found \n”);
                exit(-1);
            }
        }
    }

    // Parent process closes pipe descriptors
    close(fildes[0]);
    close(fildes[1]);
    for (i = 0; i < 2; i++) wait();
    return(0);
}
```

Before you run your program a_pipe check what must be the result of the shell commands that you try to implement. You then run the program and check that the result is the same. Compile *a_pipe.c* and run it.

```
$ gcc a_pipe.c -o a_pipe
$ ls -la | wc -l
$ ./a_pipe
```

Question 7: Analyze the code and the result of the execution and answer the following:

1. What shows the process in the standard output?

21 (the amount of lines returned by the command "ls -la")

6.2 Exercise 8 (Optional): two pipes with three processes

Based on the scheme followed in exercise 7, develop a program called *two_pipes.c* that runs the following command line:

```
$ ls -la | grep ejemplo | wc -l > result.txt
```

The necessary redirection structure is shown in Figure 4.

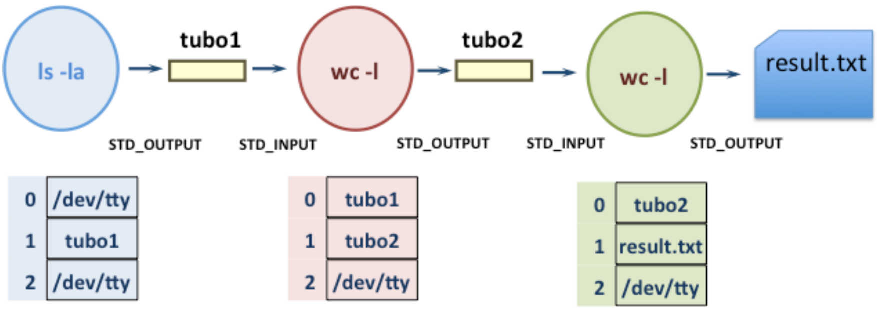

Figure 4: Diagram of redirections and pipes to be implement in the exercise 8

7. Annex: system calls syntax

7.1 open() and close ()

open()

```
#include < sys/types.h >
#include < sys/stat.h >
#include < fcntl.h >
int open(const char * pathname, int flags [, mode_t mode]);
```

The open call opens the file designated by path and returns the associated file descriptor.

pathname points to the name of a file.

flags is used to indicate the file open mode. This mode is built by combining the following values:

O_RDONLY: read mode.
O_WRONLY: writing mode.
O_RDWR: mode read and write.
O_CREAT: If the file does not exist, create it with the permissions given in mode.
O_EXCL: If O_CREAT is set and the file exists, the call fails.
O_APPEND: opens the file and offset points to the end of it. Provided that it is written in the file shall be the end of it.

mode is optional and allows you to specify the permissions that you want to have the file in case that you are creating.

Returns

> 0; It returns a positive number that corresponds to the file descriptor if successful
-1; If there is error

close()

```
#include < unistd.h >

int close (int fd);
```

Description

Close frees a file descriptor table position

Return

= 0; It returns 0 if no error
-1; If there are any errors.

7.3 read () and write()

read()

```
#include < unistd.h >

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

The so-called read reads a number of bytes given by count of the file to which fd file descriptor referenced and moves them from the address of memory pointed to by buf.

Return

> 0 if successful; Returns the number of bytes read,
= 0 if you find the end of the file and
=-1 if there is error.

write()

```
#include < unistd.h >

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

The so-called write writes a number of bytes given by count in the file whose file descriptor given by fd. The bytes to write must be from the position of memory shown in buf.

<u>Return</u>

> > 0 if successful;Returns the number of bytes written
> -1 if there is an error.

7.5 pipe()

```
#include < unistd.h >

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

Creates a channel of communication. The fildes to the return parameter contains two file descriptor, fildes [0] contains the descriptor fildes [1] and reading of Scripture.

Read in fildes [0] operation accesses the data written in a FIFO queue and fildes [1] (first come, first serve is).

<u>Return</u>

0 if no error
-1 if there is an error

7.6 dup() and dup2()

```
#include < unistd.h >

int dup (int oldfd);

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

Duplicates a file descriptor.

DUP duplicates the descriptor oldfd about the first entry in the table of descriptors of the process that is empty.

dup2 doubles oldfd about newfd descriptor descriptor. In the case that this already include a reference to a file, closes it before duplicating.

<u>Return</u>

> 0; Both return the value of the new file descriptor
-1 in case of error