Práctica 2

Sistema de ficheros

2.1. Objetivos

Comprender las llamadas al sistema y funciones en GNU/Linux para manejo de ficheros y directorios. Entender cómo se realiza la gestión de un sistema de ficheros.

2.2. Introducción

Esta práctica se centra en el Sistema de Ficheros (SF), su estructura básica en Linux así como la creación de un sistema de ficheros propio, implementando un conjunto de funciones que doten a este sistema de una funcionalidad mínima. Se usarán fundamentalmente llamadas al sistema (consultar man 2 syscalls, por ejemplo) y funciones de biblioteca (consultar man 3 strcmp, por ejemplo).

Para ello usaremos la biblioteca FUSE[1] (Filesystem in Userspace) que nos proporciona una API sencilla que nos permite implementar sistemas de ficheros en espacio de usuario y sin

[1] `http://libfuse.github.io/doxygen`

la necesidad de privilegios especiales (simplemente la inclusión del usuario en el grupo del sistema `fuse`)

2.2.1. Recuerda: gestión de errores

El chequeo y la gestión de errores es tema de gran importancia en cualquier circunstancia y también por lo que se refiere al uso de llamadas al sistema. En este caso el error se manifiesta mediante un determinado valor de retorno al efectuar la llamada, típicamente el valor entero `-1`, y se detalla mediante una variable especial, `errno` (algunas funciones de biblioteca recurren a un mecanismo semejante). Esta variable está definida en `<errno.h>` como:

```
extern int errno;
```

Su valor sólo es válido inmediatamente después de que una llamada dé error (devuelva -1) ya que es legal que la variable sea modificada durante la ejecución con éxito de una llamada al sistema. Conviene aclarar además que en programas de un solo thread, `errno` es una variable global; en caso de un proceso multithread se emplea un `errno` local por cada thread, para evitar problemas de coherencia.

El valor de `errno` puede ser leído o escrito directamente; se corresponde con una descripción textual de un error específico que está documentado en `<asm/errno.h>`. La biblioteca de C proporciona una serie de funciones útiles para traducir el valor de `errno` por su representación textual. Las principales funciones son:

```
#include <stdio.h>
void perror(const char *str); // Print a system error message associated
                              // with errno, along with a string passed as an
                              // argument
#include <string.h>
char *strerror(int errnum);   // returns a string describing errnum
```

2.3. Manejo de ficheros

El manejo de ficheros en Linux se hace típicamente de dos formas: (1) mediante llamadas al sistema conocido como acceso de bajo nivel, y (2) mediante funciones de la biblioteca estándar de entrada/salida (Standard I/O). En esta práctica vamos a manejar ficheros usando llamadas de bajo nivel. Será de especial ayuda consultar las páginas de manual de `stat` (2), `fstat`, `lseek`, `read` (2) y `write` (2).

2.3.1. Acceso de bajo nivel a ficheros

El acceso de bajo nivel emplea las siguientes llamadas representativas (consultar `man 2 nombre_llamada` para más información):

- Creación, eliminación y cambio de nombre: `creat()`, `unlink()`, `rename()`
- Apertura y cierre: `open()`, `close()`
- Lectura y escritura: `read()`, `write()`
- Miscelánea: `dup()`, `dup2()`, `lseek()`, `umask()`, `truncate()`, `ftruncate()`, `fcntl()`, `ioctl()`, `fsync()`, etc.

Estas llamadas se caracterizan por representar internamente el fichero de trabajo mediante un descriptor de fichero (tipo `int`) y considerar un fichero como una secuencia de bytes cuyo índice es su posición en el fichero. Un fichero abierto está caracterizado por: su descriptor, su posición actual en la secuencia de bytes y el tipo de apertura vigente.

Habitualmente existen 3 ficheros predeterminados abiertos y se suelen corresponder con la entrada y salida por el terminal asignado a la sesión:

- Entrada estándar `STDIN_FILENO` (fd=0; acceso de sólo-lectura),
- Salida estándar `STDOUT_FILENO` (fd=1, acceso de sólo-escritura)
- Salida de error `STDERR_FILENO` (fd=2, acceso de sólo-escritura).

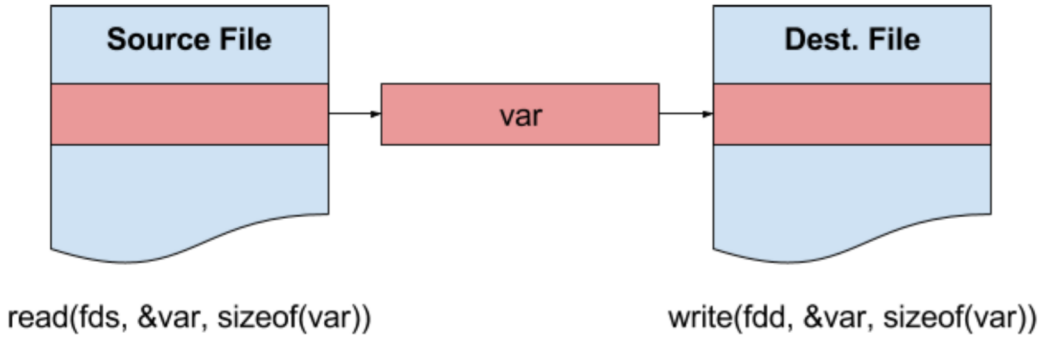

Figura 2.1: Copia de ficheros usando llamadas al sistema.

Ejemplo 1: Implementar un programa que copie el contenido de un fichero en otro (ver Figura 2.1).

Sintaxis: `copy source_file dest_file [BLOCKSIZ]`

Operación: El programa copia el contenido del fichero `source_file` sobre el fichero `dest_file`. La copia se realiza mediante transferencias de `BLOCKSIZ` bytes que opcionalmente puede ser especificada como argumento.

Ejercicio 1: Comparar el tiempo de ejecución de la copia de un fichero de gran tamaño (> 1MB) para diferentes valores de `BLOCKSIZ`. Explicar los valores obtenidos. Para ello ejecutar el programa con una orden como la siguiente:

```
time ./copy example1.pdf example1_1024.pdf 1024
```

El comando `time` muestra en pantalla el tiempo de ejecución del programa que se le pasa como argumento desglosado en tres valores: tiempo de CPU en modo usuario, tiempo de CPU en modo sistema y tiempo real transcurrido.

2.3.2. La biblioteca de E/S estándar (`stdio`)

La biblioteca de E/S estándar proporciona una interfaz sencilla para controlar la E/S en C. El descriptor de fichero para las funciones `stdio` no es un `int` sino un tipo (`FILE*`) donde `FILE` es una estructura de datos que encapsula, entre otros parámetros, la información de bajo nivel de acceso (descriptor entero, tipo de apertura, ...).

Correspondiendo a los descriptores estándar de bajo nivel, disponemos habitualmente de tres descriptores estándar `stdio` que son:

- Entrada estándar **`stdin`** (relacionado con el descriptor `0`),
- Salida estándar **`stdout`** (relacionado con el descriptor `1`)
- Salida error **`stderr`** (relacionado con el descriptor `2`).

Algunas funciones `stdio` habituales son, por ejemplo, **fopen()**, **fclose()**, **fread()**, **fscanf()**, **fwrite()**, **fprintf()**, **fseek()**, etc.

2.4. Administración de ficheros

Los atributos de cada fichero en UNIX están contenidos en un *nodo-i*. El contenido de los *nodos-i* puede ser consultado, al menos parcialmente, con las llamadas `stat()`, `fstat()` o `lstat()`. La información obtenida es la que se describe en la estructura `struct stat` declarada en `<sys/stat.h>` (consultar `man 2 stat`).

```
struct stat {
  dev_t    st_dev;           /* dispositivo ('major' y 'minor')
                              * del sistema de ficheros */
  ino_t    st_ino;           /* número de nodoi */
  mode_t   st_mode;          /* tipo de fichero y permisos */
  link_t   st_nlink;         /* número de enlaces rígidos */
  uid_t    st_uid;           /* ID del usuario propietario */
  gid_t    st_gid;           /* ID del grupo propietario */
  dev_t    st_rdev;          /* dispositivo ('major' y 'minor'
                              * si el fichero es "especial") */
  off_t    st_size;          /* tamaño total, en bytes */
  unsigned long st_blksize;  /* tamaño de bloque para la E/S
                              * del sistema de ficheros */
  unsigned long st_blocks;   /* número de bloques asignados */
  time_t   st_atime;         /* fecha de último acceso */
  time_t   st_mtime;         /* fecha de última modificación */
  time_t   st_ctime;         /* fecha de último cambio de estado */
};
```

Algunas llamadas que permiten cambiar o consultar algunos de los atributos de un fichero son:

- Consultar la posibilidad de acceder a un fichero (examinar los permisos): `access()`
- Modificación de los permisos y propietarios (usuario y grupo): `chmod()`, `fchmod()`, `chown()`, `fchown()`, `lchown()`
- Modificación de las fechas asociadas de acceso y modificación: `utime()`

Ejemplo 2: Emular el comportamiento de la orden `stat` de Linux que muestra los atributos de un fichero.

Sintaxis: `status file1 file2 ...`

Operación: Este programa consulta con la llamada `stat` los atributos del *nodo-i* correspondiente a cada uno de los ficheros especificados como argumento y muestra por la salida estándar una relación de todos los atributos que la llamada proporciona.

Ejercicio 2: En el ejemplo 2, si el fichero es un enlace simbólico, los atributos mostrados se refieren al *nodo-i* del fichero al que apunta el enlace. Incorporar una opción (`-L`) al programa anterior para que, cuando se especifique, haga que la consulta de atributos para un enlace simbólico se refiera al fichero apuntado, y cuando no se dé tal opción, se refiera a los atributos del propio enlace.

2.5. Manejo de directorios

En UNIX se distinguen 7 tipos de ficheros: ordinarios, directorios, FIFOS, dispositivos de caracteres, dispositivos de bloques, enlaces simbólicos y sockets.

La creación de cada uno de ellos se realiza empleando valores específicos en los argumentos de ciertas funciones:

- Ficheros ordinarios (altas y bajas en directorios): `mknod()`, `open()`, `link()`
- Enlaces simbólicos: `symlink()`, `readlink()`
- Dispositivos: `mknod()`
- FIFOS: `mknod()`
- Sockets: `socket()`

Los directorios son ficheros cuya creación, eliminación y acceso se efectúa con operaciones particulares, diferentes a las de los ficheros ordinarios: `mkdir()`, `rmdir()`, `chdir()`, `getcwd()`

Las funciones que se aplican para actuar sobre directorios (abrir, cerrar y recorrer) están definidas en `<dirent.h>` y básicamente son: `opendir()`, `closedir()`, `readdir()`, `seekdir()`, `telldir()` y `rewinddir()`.

Estas funciones hacen uso de los siguientes tipos:

- **DIR** (descriptor de directorio): tiene una definición opaca al usuario
- **struct dirent** (campos relevantes de cada entrada individual en un directorio):

```
struct dirent {
  ino_t     d_ino;     /* inode number */
  off_t     d_off;        /* offset */
  ushort    d_reclen;  /* register length */
  char*     d_name;    /* file name */
};
```