Práctica 2

Sistema de ficheros

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.

2.1. Introducción

Esta práctica consta de dos partes bien diferenciadas. La primera parte versa sobre la gestión de ficheros y directorios, y se incluye como repaso (Sección 2.2). La segunda parte se centra en el sistema de ficheros, 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 (Secciones 2.3 y 2.6).

La forma más sencilla de obtener documentación es a través de las páginas del manual en línea. La sección 2 ofrece documentación de llamadas al sistema (consultar `man 2 syscalls` por ejemplo). La sección 3 documenta funciones de biblioteca (consultar `man 3 readline` por ejemplo).

2.1.1. 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); // Muestra el texto asociado a errno
                              // junto con la cadena indicada como
                              // argumento
#include <string.h>
char *strerror(int errnum);   // Devuelve el texto explicativo del
                              // error errnum
```

2.2. 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).

2.2.1. 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).

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

Sintaxis: `copiar fichero_fuente fichero_destino [TBLOQUE]`

Operación: El programa copia el contenido del `fichero_fuente` sobre el `fichero_destino`. La copia se realiza mediante transferencias de `TBLOQUE` bytes que opcionalmente puede ser especificada como argumento.

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

```
time ./copiar ejemplo1.pdf ejemplo1_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.

2.2.2. 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.

2.3. 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`).

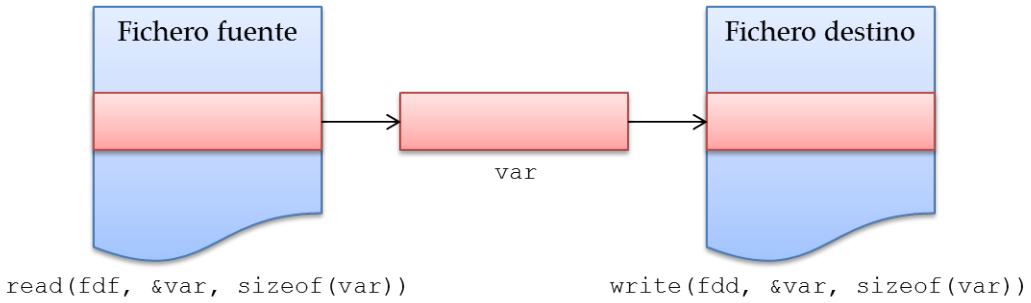

Figura 2.1: Copia de ficheros usando llamadas al sistema.

```
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: `estado ficheros...`

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: 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 que, 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.

2.4. 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;    /* Número de nodo-i */
  off_t    d_off;    /* Desplazamiento */
  ushort   d_reclen; /* Longitud del registro */
  char*    d_name;   /* Nombre de fichero */
};
```

Ejemplo 3: Calcular, dentro de un subconjunto de la jerarquía de ficheros, la distribución de ficheros según tamaños y la distribución de directorios según número de entradas.

Sintaxis: `distribucion [-t|-n] directorio`

Operación: Este programa recorre los ficheros y directorios del esquema jerárquico de una instalación UNIX tomando como punto de partida o raíz el directorio especificado como argumento, y va generando los datos de sendos histogramas de número de ficheros según tamaños (opción `-t`) y/o de número de directorios según entradas que contienen (opción `-n`). Si no especifica ninguna opción se aplica la opción `-t` por defecto. Para ello, el programa lee recursivamente las entradas de los directorios que se encuentra.

Ejercicio: Mostrar los resultados de aplicar el programa con ambas opciones al directorio `$HOME` del usuario.

2.5. Tiempos

Por lo general, las dos principales medidas de tiempo son:

- Fecha o tiempo real: tiempo transcurrido desde el inicio de los tiempos o *epoch*, que la mayor parte de los UNIX sitúan en el 1 de Enero de 1970 a las 00:00:00 GMT. Se obtiene mediante las llamadas `time()` y `gettimeofday()` y se puede dar formato mediante las funciones: `localtime()`, `gmtime()`, `asctime()`, `ctime()`, `mktime()`, `strftime()`, `strptime()`.

```
SINOPSIS
#include <time.h>

 time_t time(time_t *tp);
 struct tm *localtime(const time_t *tp);
 size_t strftime(char *s, size_t max, const char *fmt; const struct tm *tmp)
```

```
SINOPSIS
#include <sys/time.h>
#include <unistd.h>

 int gettimeofday(struct timeval *tv, struct timezone *tz);
 int settimeofday(const struct timeval *tv,const struct timezone *tz);
```

  El parámetro `timezone` no debe utilizarse (puntero a `NULL`).

- Tiempo de CPU: tiempo consumido por la ejecución de un programa (usuario y/o sistema). Se obtiene mediante las llamadas `times()` y `getrusage()`, o mediante la función `clock()`.

```
SINOPSIS
#include <systime.h>

 clock_t times(struct tms *buf);
```

```
SINOPSIS
#include <time.h>

 clock_t clock(void);
```

```
SINOPSIS
#include <sys/time.h>
#include <sys/resource.h>
#include <unistd.h>

 int getrusage (int who, struct rusage *usage);
```

- Otros comandos y funciones relacionados con la gestión de tiempos son: `date`, `sleep()`, `usleep()`, `nanosleep()`, `sleep`, `getitimer()`, `setitimer()`

2.6. Ejercicio Propuesto

En esta práctica implementaremos un mini-sistema de ficheros tipo UNIX. La estructura completa de este sistema de ficheros existirá en un único archivo. Almacenaremos, modificaremos y eliminaremos archivos de este sistema de ficheros. Este sistema de ficheros se ilustra conceptualmente en la Figura 2.2. Una estructura C posible para gestionarlo podría ser como sigue:

```
typedef struct MiSistemaDeFicheros {
  int discoVirtual;                    // Archivo que almacena todo el
                                       // sistema de ficheros
  EstructuraSuperBloque superBloque;   // Superbloque
  BIT mapaDeBits[TAM_MAPA_BITS];       // Mapa de bits
  EstructuraDirectorio directorio;     // Directorio raíz
  EstructuraNodoI* nodosI[MAX_NODOSI]; // Nodos-i
  int numNodosLibres;                  // Número de nodos-i libres
  EstructuraNodoI nodoActual;          // Nodo-i actual
} MiSistemaDeFicheros;
```

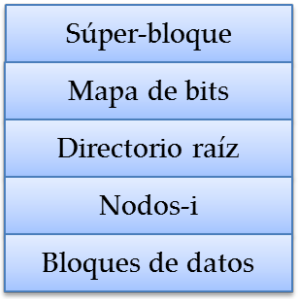

Figura 2.2: Esquema del sistema de ficheros.

El superbloque es el primer bloque en el disco que contiene información global sobre el sistema de ficheros. Podemos almacenar en el superbloque cosas como el número total de bloques libres, el tamaño total del sistema de ficheros y cualquier otra cosa que consideremos útil para hacer el sistema de ficheros persistente. Se propone la siguiente estructura:

```
typedef struct EstructuraSuperBloque {
  int tamSuperBloque;       // Tamaño de la info. de superbloque
  int tamDirectorio;        // Tamaño de la info. de directorio
  int tamNodoI;             // Tamaño de la info. de nodo-i

  int tamDiscoEnBloques;    // Núm. de bloques en disco
  int numBloquesLibres;     // Núm. de bloques libres

  int tamBloque;            // Tamaño de bloque
  int maxTamNombreArchivo;  // Tamaño máx. de nombre de archivo
  int maxBloquesPorArchivo; // Tamaño máx. de bloques por archivo
} EstructuraSuperBloque;
```

El directorio raíz mantiene el mapeo entre los nombres de fichero y los números de nodo-i (o punteros). Se proponen las siguientes estructuras:

```
typedef struct EstructuraArchivo {
  int  idxNodoI;                                  // Nodo-i asociado
  char nombreArchivo[MAX_TAM_NOMBRE_ARCHIVO+1]; // Nombre archivo
  BOOLEAN libre;                                  // Archivo libre
} EstructuraArchivo;

typedef struct EstructuraDirectorio {
  int numArchivos;                                      // Núm. archivos
  EstructuraArchivo archivos[MAX_ARCHIVOS_POR_DIRECTORIO]; // Archivos
} EstructuraDirectorio;
```

Los nodos-i tienen información de los bloques de datos en el archivo:

```
typedef struct EstructuraNodoI {
  int numBloques;                                 // Núm. bloques
  int tamArchivo;                                 // Tamaño archivo
  time_t tiempoModificado;                        // Tiempo modif.
  DISK_LBA idxBloques[MAX_BLOQUES_POR_ARCHIVO]; // Bloques
  BOOLEAN libre;                                  // Nodo libre
} EstructuraNodoI;
```

Por simplicidad, podemos asumir lo siguiente:

- Nuestro sistema de ficheros no necesita soporte para directorios multinivel. Hay un solo directorio en nuestro sistema. Este directorio contiene como mucho 100 archivos.
- Cada archivo contiene a lo sumo 100 bloques de datos. Por lo tanto, el tamaño de cada archivo es menor que $100 \times \text{tam.bloque}$. Podemos definir el tamaño de bloque por nuestra cuenta.
- Los nodos-i en nuestro sistema contienen a lo sumo 100 punteros directos a bloques de datos. No se requieren punteros indirectos.

2.6.1. Uso del programa

- `MiSistemaDeFicheros -mkfs tamDisco nombreArchivo`
  - Orden que crea un disco virtual con el nombre y tamaño definidos en los argumentos.

2.6.2. Pasos a seguir

Nos descargamos el esqueleto de la aplicación a desarrollar (*MiSistemaDeFicheros.zip*), y trabajaremos sobre él implementando algunas funciones:

- `int myMkfs(MiSistemaDeFicheros* miSistemaDeFicheros, int tamDisco, char* nombreArchivo);`
  - Esta función crea un archivo bajo el directorio actual. Este archivo simula el disco virtual basado en el cual construiremos nuestro sistema de ficheros. Además, esta función inicializa todas las estructuras de datos mencionadas anteriormente.

- `int myImport(char* nombreArchivoExterno, MiSistemaDeFicheros* miSistemaDeFicheros, char* nombreArchivoInterno);`
  - Esta función crea un nuevo archivo llamado `nombreArchivoInterno` en nuestro sistema y copia el archivo `nombreArchivoExterno` desde nuestro sistema de ficheros Linux en él. Si el tamaño del archivo Linux es mayor que el tamaño permitido por nuestro sistema de ficheros o que el espacio libre en disco, esta función falla y retorna 0. Si el archivo `nombreArchivoInterno` existe, salimos sin hacer nada.
- `int myExport(MiSistemaDeFicheros* miSistemaDeFicheros, char* nombreArchivoInterno, char* nombreArchivoExterno);`
  - Esta función copia el archivo `nombreArchivoInterno` de nuestro sistema de ficheros al archivo `nombreArchivoExterno` en Linux. Si `nombreArchivoExterno` existe, salimos de la función sin hacer nada.
- `int myQuota(MiSistemaDeFicheros* miSistemaDeFicheros);`
  - Esta función devuelve la cantidad de espacio libre en el disco virtual (ya está implementada).
- `int myRm(MiSistemaDeFicheros* miSistemaDeFicheros, char* nombreArchivo);`
  - Borra el archivo `nombreArchivo` en nuestro sistema. Si falla, devuelve 0 de lo contrario retorna 1.
- `void myLs(MiSistemaDeFicheros* miSistemaDeFicheros);`
  - Ya implementado en el esqueleto proporcionado. Devuelve una lista que contiene información de todos los archivos en el directorio raíz. De cada archivo retorna el nombre, tamaño y la última vez que el archivo fue modificado.