Prácticas de Sistemas Operativos

Toñi Reina, David Ruiz y Juan Antonio Álvarez

Boletín ♯3: Archivos

Curso 2004/05

Índice

---

1. Introducción

El interfaz de Entrada/Salida a nivel de fichero que ya conocemos por cursos anteriores, nos permite a través de un tipo abstracto de datos (el tipo FILE *) realizar operaciones de apertura, escritura/lectura con o sin formato, desplazamiento, conversión automática de caracteres en ficheros de texto, etc. sobre ficheros lógicos (*streams*). Todo acceso al fichero físico es realizado a través de *buffers*.

En determinado momento puede ser deseable acceder a un fichero sin este nivel de abstracción, es decir, accediendo directamente y sin *buffers*[1] intermedios a sus datos "crudos" en binario, sin ningún tipo de formateo ni de conversión.

El acceso a bajo nivel a los ficheros no se hace a través de un puntero a FILE, sino que se hace mediante lo que se llama un descriptor de fichero. Un descriptor de fichero es un entero no negativo, que el sistema emplea para identificar un fichero (este entero se emplea para indexar una tabla en la cual se almacena toda la información necesaria sobre el mismo). Dado que el sistema identifica a un fichero mediante su descriptor, en un mismo proceso no existirán nunca dos ficheros abiertos con el mismo descriptor.

Existen tres descriptores de fichero predefinidos:

- 0: Asociado a la entrada estándar (variable stdin definida en *stdio.h*).
- 1: Asociado a la salida estándar (variable stdout definida en *stdio.h*).
- 2: Asociado a la salida estándar de error (variable stderr definida en *stdio.h*).

El mecanismo para acceder a bajo nivel a un fichero será:

1. Abrir el fichero, asociándole un descriptor de fichero. La operación de apertura puede crear también el fichero físico.
2. Acceder al fichero identificado por el descriptor, mediante las funciones de lectura/escritura
3. Cerrar el fichero identificado por el descriptor, mediante la función de cierre.

La mayoría de las llamadas al sistema, y en particular las que vamos a ver en este tema, pueden fallar por diversas causas. Cuando se hace una llamada al sistema mediante una función en lenguaje C, normalmente se detecta si ha habido fallo o no por el valor devuelto por dicha función. Es decir, la función devuelve un valor especial en caso de haber fallado por alguna causa. Pero, ¿por qué causa?. Para determinar la causa del fallo de una función se emplea la variable global errno.

Esta es una variable global declarada en la librería del compilador, y para poder emplearla sólo necesitamos incluir el fichero de cabecera <errno.h>. La mayoría de las funciones que se traducen en llamadas al sistema establecen un valor identificativo del error ocurrido, caso de producirse uno. El valor en concreto depende de cada función, y generalmente existen constantes definidas para los distintos valores asociados a los posibles errores.

2. Acceso al directorio de trabajo

La función getcwd par obtener el directorio actual de trabajo:

```
#include <unistd.h>
char *getcwd(char *buf, size_t size);
```

Devuelve NULL en caso de error y establece un valor identificativo del mismo en la variable global errno. En caso, de que no se produzca un error, devuelve un puntero a la zona de memoria donde se almacena el nombre del directorio de trabajo actual.

[1] Cuando nos referimos a que la entrada/salida se efectúe sin *buffer*, nos referimos a que nuestro programa no emplea un *buffer* propio en su espacio de memoria para el acceso a los datos. No obstante, el Sistema Operativo sí que utilizará *buffers* para acelerar el acceso a los datos. No obstante, este hecho nos resulta trasparente.

buf es la variable dónde se almacenará el nombre del directorio actual de trabajo. Si buf es un puntero no nulo, el directorio de trabajo actual se almacenará en la zona de memoria apuntada por buf. Si buf es un puntero nulo, `getcwd` reservará memoria dinámica con `malloc`. En este caso, el puntero devuelto por `getcwd` puede ser utilizado en una llamada posterior a `free()`.

size es el tamaño en bytes de buf.

Ejemplo 1 *El siguiente programa imprime la ruta de acceso del directorio de trabajo activo.*

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <limits.h>
#ifndef PATH_MAX
#define PATH_MAX 255
#endif

int main(void)
{
   int error=0;
   char mycwd[PATH_MAX + 1];

   if (getcwd(mycwd, PATH_MAX + 1) == NULL)
   {
       perror("No se tuvo acceso al directorio");
       error=1;
   }
   printf("Directorio de trabajo: %s\n", mycwd);

   return(error);
}
```

3. Acceso a las entradas de un directorio

Las funciones opendir, readdir, rewinddir y close dir sirven para acceder las entradas de un directorio:

```
#include <sys/types.h>
#include <dirent.h>

DIR *opendir(const char *name);
struct dirent *readdir(DIR *dir);
void rewinddir(DIR *dir);
int closedir(DIR *dir);
```

- La función opendir, dado el nombre de un directorio, devuelve un identificador de bloque al directorio para el resto de funciones. En caso de error, esta función devuelve NULL.
- Cada llamada subsecuente a la función readdir devuelve un puntero a una estructura que contiene información sobre la siguiente entrada del directorio. Cuando llega al final del directorio devuelve NULL.
- La función rewinddir sirve para volver a empezar a recorrer el directorio.
- closedir se utiliza para terminar.
- Los campos de *struct dirent* son:

```
struct  dirent {
   ino_t          d_ino;    /* Número de i-nodo */
```

```
        off_t           d_off;    /* Offset de la siguiente entrada del directorio */
        unsigned short  d_reclen; /* Longitud de la estructura de la
                                     entrada de directorio */
        char            d_name[MAXNAMELEN]; /* Nombre del archivo*/
        };
```

Ejemplo 2 *El siguiente programa muestra los nombres de los archivos contenidos en el directorio cuyo nombre se pasa como argumento.*

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <dirent.h>
#include <errno.h>

int main(int argc, char *argv[])
{
    int error=0;
    DIR *dirp;
    struct dirent *direntp;

    if (argc != 2)
    {
       fprintf(stderr,"Usar: %s directorio\n", argv[0]);
       error=1;
       return (error);
    }

    if ((dirp = opendir(argv[1])) == NULL)
    {
       fprintf(stderr, "No se puede abrir el directorio %s. %s\n",
               argv[1], strerror(errno));
       error=2;
       return(error);
    }

    while ( (direntp = readdir( dirp )) != NULL )
       printf("%s\n", direntp->d_name );

   closedir(dirp);


}
```

4. Obtener información de una entrada de un directorio

Las funciones stat, fstat y lstat sirven para obtener información de una entrada de un directorio a través de su nombre o de su descriptor de fichero.

```
#include <sys/stat.h>
#include <unistd.h>
int stat(const char *file_name, struct stat *buf);
int fstat(int  filedes, struct stat *buf);
int lstat(const char *file_name, struct stat *buf);
```

- La función stat sirve para recuperar la información contenida en el *inodo* del archivo *file_name*. Esta información se almacena en una variable de tipo struct stat.
- Las funciones fstat y lstat tienen el mismo cometido que la función stat pero para obtener información de un archivo apuntado por el descriptor *fildes* o al enlace simbólico *file_name* respectivamente.
- Los campos de *struct stat*[2] son:

```
struct stat
{
    dev_t         st_dev;      /* device */
    ino_t         st_ino;      /* inode */
    mode_t        st_mode;     /* protection */
    nlink_t       st_nlink;    /* number of hard links */
    uid_t         st_uid;      /* user ID of owner */
    gid_t         st_gid;      /* group ID of owner */
    dev_t         st_rdev;     /* device type (if inode device) */
    off_t         st_size;     /* total size, in bytes */
    unsigned long st_blksize;  /* blocksize for filesystem I/O */
    unsigned long st_blocks;   /* number of blocks allocated */
    time_t        st_atime;    /* time of last access */
    time_t        st_mtime;    /* time of last modification */
    time_t        st_ctime;    /* time of last change */
};
```

- La estructura struct stat se le debe proporcionar a las funciones stat, lstat y fstat. Es decir, que estas funciones no reservan memoria.
- Todas las funciones devuelven 0 en caso de que la operación vaya bien y -1 en caso contrario.
- En <sys/stat.h> se definen algunas constantes para hacer más fácil las comprobaciones (S_IFMT, S_IFDIR, S_IFCHR, S_IFREG, S_IREAD, S_IWRITE, S_IEXEC) tal y como se muestra en el siguiente ejemplo.

Ejemplo 3 *El siguiente programa hace uso de* stat *para determinar si el nombre de nombre de archivo que se pasa como primer argumento es un directorio. En caso afirmativo, imprime la fecha de última modificación.*

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <time.h>
#include <sys/types.h>
#include <sys/stat.h>

int main(int argc, char *argv[])
{
    struct stat statbuf;
    int error=0;

    if (argc != 2)
    {
        fprintf(stderr, "Usar: %s fichero\n", argv[0]);
        error=1;
    }
    if (stat(argv[1], &statbuf) == -1)
    {
        fprintf(stderr, "Error al hacer stat sobre %s: %s\n",
```

[2]Los campos de esta estructura pueden variar entre diferentes implementaciones de Unix

```
                argv[1], strerror(errno));
            exit(1);
        }
        if (statbuf.st_mode & S_IFDIR)
        {
            printf("%s es un directorio. ", argv[1]);
            printf("Última modificación %s\n", ctime(&statbuf.st_mtime));
        }
        else
            printf("%s is not a directory\n", argv[1]);

        return error;
    }
```

5. Apertura de un fichero

La apertura a bajo nivel de un fichero se realiza mediante la función open:

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

int open (char *name, int how_to_open, int mode);
```

Devuelve un entero no negativo que es el descriptor que se le asocia al fichero abierto. En caso de error, la función devuelve -1, y establece un valor identificativo del error en la variable global errno.

name es el nombre del fichero que se abre.

how_to_open indica la forma en que se abre, y puede ser una de las siguientes constantes predefinidas en el archivo de cabecera <`fcntl.h`>, o la combinación mediante el operador OR de bits (|) de dos o más de ellas:

- O_RDONLY: Abre el fichero sólo para leer.
- O_WRONLY: Abre el fichero sólo para escribir
- O_RDWR: Abre el fichero para leer y escribir
- O_APPEND: Añade al fichero si se escribe, en vez de truncar.
- O_TRUNC: Trunca el fichero (lo deja con 0 bytes) si se abre para escribir.
- O_CREAT: Crea el fichero si no existe. El tercer parámetro indicarán los permisos de acceso al fichero.
- O_EXCL: Produce error si el fichero se intenta crear pero ya existe.
- O_NDELAY: No bloquea el proceso al acceder al fichero. No nos detendremos en este aspecto.

mode tiene relevancia sólo cuando se crea el fichero (se añade O_CREAT como opción en el segundo parámetro) y especifica los permisos de acceso al fichero. Tampoco profundizaremos en este tema.

6. Lectura/Escritura de un fichero

La lectura a bajo nivel de un fichero se realiza mediante la función read, y la escritura se realiza mediante la función write:

```
#include <sys/file.h>
#include <unistd.h>
```

```
int read (int fd, void *buffer, int num_bytes);

int write (int fd, void *buffer, int num_bytes);
```

Devuelven el número de bytes leídos/escritos, o -1 en caso de error. En el caso de read, se devuelve 0 cuando se llega al final del fichero.

fd es el descriptor asociado mediante open al fichero del que se lee/escribe. Para read, el fichero debe estar abierto en un modo que permita la lectura, y para write, el fichero debe haber sido abierto en un modo que permita la escritura.

buffer es un puntero al *buffer* que se lee/escribe.

num_bytes es el número de caracteres que se leen/escriben.

7. Cierre de un fichero

El cierre de un fichero se realiza mediante la función close:

```
#include <sys/file.h> #include <unistd.h>

int close (int fd);
```

Devuelve 0 si el cierre se realiza satisfactoriamente, o -1 en caso de error.

fd es el descriptor asociado por open al fichero que se cierra.

Es importante destacar que además de cerrar el fichero, la llamada al sistema close libera el descriptor de fichero. Esto quiere decir que otra llamada a open puede devolver (y de hecho lo hará) el mismo número de descriptor que se acaba de cerrar (los descriptores de fichero se reutilizan).

8. La llamada fdopen

A veces puede ser deseable convertir un descriptor de fichero a bajo nivel ya asociado a un fichero a un puntero a FILE, con lo que podemos emplear a partir de ese momento todas las funciones de entrada/salida que ya conocemos de stdio.h: fprintf, fputs, fscanf, fgets, etc.

Esto se puede realizar mediante la función fdopen, cuyo prototipo se encuentra declarado en el fichero stdio.h:

```
#include <stdio.h>

FILE *fdopen (int fd, char *mode);
```

Devuelve un puntero a la estructura FILE creada, o un puntero a NULL en caso de error.

fd es el descriptor de fichero que queremos convertir a puntero a FILE.

mode es el modo en que se abre el fichero, de la misma forma en que se le pasaría a la función fopen. El modo que se solicite debe ser compatible con el modo en que se abrió el fichero en la llamada open. Por compatible, se quiere decir que no podemos abrir para lectura, si en la llamada a open el fichero se abrió como sólo escritura. No obstante, si podemos abrir como solo lectura, si por ejemplo el fichero se abrió como lectura y escritura en la llamada a open.

Que se haya creado un puntero a FILE para el descriptor de fichero no implica que el descriptor no siga siendo válido; es decir, se pueden seguir usando las funciones de bajo nivel sobre el descriptor junto con las funciones de alto nivel sobre el puntero a FILE. No obstante, se recomienda tener cuidado con esta práctica, ya que las funciones

de alto nivel emplean *buffers* y las de bajo nivel no (se ”saltan”el *buffer*), por lo que podemos obtener resultados inesperados si mezclamos indiscriminadamente ambos métodos de acceso al fichero.

Si el fichero es cerrado empleando la función fclose sobre el puntero a FILE, el descriptor de fichero queda también cerrado, ya que la función fclose llama internamente a la función close.

Ejemplo 4 *A continuación presentamos un programa de ejemplo que escribe en un fichero, empleando primero funciones de bajo nivel, y después empleando un puntero a FILE:*

```
#include <stdio.h>
#include <string.h>
#include <sys/file.h>

#define TAM_BUF 255
int main (void)
{
int fd;
FILE *fp;
char buf[TAM_BUF];

    strcpy(buf,"Esta línea del fichero se escribe a bajo nivel\n");
    fd= open ("mifich.dat", O_RDWR | O_CREAT | O_TRUNC, 0700);
    if (fd<0)
        {
        fprintf (stderror,"Error abriendo el fichero\n");
        exit(2);
        }
    else
    {
        write (fd, buf, strlen (buf));
        fp= fdopen(fd, "a");
        fprintf (fp, "\t\t en cambio esta se escribe a alto nivel\n");
        fclose (fp);
    }
exit (0);
}
```

9. Ejercicios

1. Realice un programa *copia* al que se le pase como argumento el nombre de un archivo *nombrefichero* y si tiene permiso de lectura sobre el mismo haga una copia del mismo con nombre *nombrefichero.bak*.
2. Realice un programa *listado* que dado un archivo *f* o un directorio *d* genere otro archivo con nombre *listado.txt* que contenga en cada línea la siguiente información para *f* o para cada fichero de *d*: nombre, número de *inodo*, *id* del usuario del propietario y tamaño en bytes.
3. Realice un programa *min-may* que lea (a bajo nivel) de la entrada estándar líneas y escriba (a bajo nivel) en la salida estándar la misma línea en la que las letras minúsculas se han pasado a mayúsculas y viceversa.
4. Realice un programa *usuarios* que lea la salida del comando last ordenada por nombre de usuarios, y muestre un informe con el número de conexiones que ha realizado cada usuario. Realice este programa de dos formas distintas:

    - Mediante la la llamada al sistema `system()`. En este caso, se le deberá introducir al programa por la línea de comandos el número de las últimas conexiones que se desea vigilar, de la siguiente forma.

      ```
      $ usuarios 20
      Usuario:i4991, 1 conex,
      Usuario:i5439, 1 conex.
      ```

```
Usuario:i5671, 4 conex.
Usuario:i6216, 1 conex.
Usuario:i6311, 5 conex.
Usuario:i6387, 4 conex.
Usuario:i6525, 1 conex.
Usuario:i6648, 1 conex.
Usuario:i6858, 1 conex.
```

- Haciendo que el comando tome la información de los usuarios de la entrada estándar. En este caso, la forma de invocar al programa será la siguiente:

```
$ last -n 20 | sort | usuarios
Usuario:i4991, 1 conex,
Usuario:i5439, 1 conex.
Usuario:i5671, 4 conex.
Usuario:i6216, 1 conex.
Usuario:i6311, 5 conex.
Usuario:i6387, 4 conex.
Usuario:i6525, 1 conex.
Usuario:i6648, 1 conex.
Usuario:i6858, 1 conex.
```

Nota 1:Para que sea más fácil hacer las dos versiones, se aconseja la creación de un módulo `usuarios.c`, donde escriba el código de las funciones para tratar el archivo, y otro módulo `principal.c`, que será el que varíe entre la versión del apartado 1 y la del apartado 2. Nota 2: es aconsejable utilizar el servidor de prácticas murillo. Pida ayuda de la función `system()`.

5. Realice un programa `arbol` que dado un directorio escribe en la salida estándar la estructura arbórea de dicho directorio.

```
druiz@macarena:~ > arbol mp3
 +---caratulas
 +---downloads
 +---Español
 |   +---Alejandro Sanz
 |   +---Angeles del Infierno
 +---Ingles
     +---ACDC
     +---Dire Straits
         +---ALCHEMY
         +---Mark Knopfler
druiz@macarena:~ >
```

6. ***(Ejercicio de Exámen 1ª CONV ITI 2002-03)***

   Realice un programa `cuentadirectorios`, que dado un directorio escriba en la salida estándar el número total de directorios y subdirectorios que contiene, a cualquier nivel. La invocación al programa se hará:

   `cuentadirectorios <nombre_directorio>`

7. ***(Ejercicio de Exámen 1ª CONV II 2002-03)***

   Realice un programa `busqueda` que dado un fichero o directorio imprima su ruta absoluta. El programa sólo realizará la búsqueda en la estructura arbórea del directorio desde donde se hace la llamada al ejecutable. En caso de no encontrarse el archivo o directorio, se deberá dar un mensaje de error. Si hubiera más de una entrada con el mismo nombre, se devolvería la información de la primera que se encontrase. Por ejemplo, si tenemos la estructura de directorios de la Figura 1 , una invocación al programa sería:

   ```
   murillo:/export/home/alumno>busqueda practicas
   ```

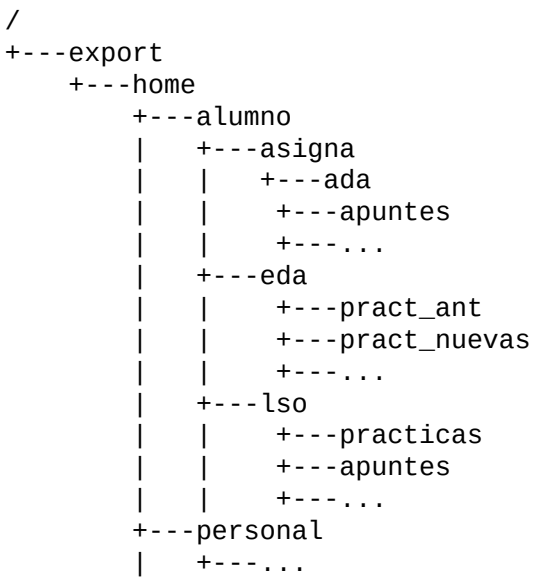

Figura 1: Estructura de directorios

y el resultado de la ejecución del mismo, debería ser:

```
 /export/home/alumno/asigna/lso/practicas
```

Por el contrario, si hacemos la siguiente invocación:

```
murillo:/export/home/alumno>busqueda pp
```

el resultado debería ser:

```
No se ha encontrado ninguna entrada llamada pp
```

8. ***(Ejercicio de Exámen 2ª CONV II 2002-03)***

   Realice un programa `chequeacopia` que tome como argumentos de entrada dos ficheros o dos directorios, e indique dando un mensaje por la salida estándar:

   - Para dos directorios, si tienen el mismo número de entradas.
   - Para dos ficheros, si tienen el mismo tamaño.

   En caso de que como argumentos se introduzcan un archivo y un directorio, se dará un mensaje de error. Algunos ejemplos de ejecución del comando serían los que se muestran a continuación.

```
$ chequeacopia inicio.c inicio.c Los archivos tienen el mismo
tamaño: 57 bytes.

$ chequeacopia nuevo copia Los directorios tienen el mismo número
de entradas: 10.

$ chequeacopia inicio.c final.c Los archivos NO tienen el mismo
tamaño.
```

   NOTAS:

   - No se permite la utilización de la llamada al sistema `system()` para resolver el ejercicio.
   - Se aconseja que implemente la siguientes funciones para resolver el problema:
     - `void tratar_directorios (char dir1[], char dir2[]);`
     - `void tratar_archivos (struct stat f1, struct stat f2);`
     - `int contar_entradas (char dir1[]);`