Contenido

4. Manejo de Directorios y Archivos Especiales

4.1 Acceso a Directorios

Los directorios son los archivos que le proporcionan al sistema la estructura jerárquica de árbol invertido. Estos archivos tienen una estructura de datos que es interpretada y mantenida por el kernel. Los directorios, como cualquier otro archivo, se pueden abrir mediante una llamada a `open` y pueden ser leídos mediante la llamada `read`, pero a ningún usuario le está permitido escribir directamente sobre ellos mediante llamadas a `write`.

Para leer la información de un directorio, debemos conocer su estructura. Como la organización de los datos de un directorio depende del sistema, los programas que escribamos para manejarlos no van a ser transportables. Para solucionar este inconveniente y hacer que la estructura del directorio sea transparente al programador de aplicaciones, se ha propuesto una biblioteca estándar de funciones de manejo de directorios. Esta biblioteca la estudiaremos más adelante.

Aunque no está permitido modificar directamente la estructura de un directorio, hay una serie de llamadas que van a permitir crear o borrar directorios y añadirles nuevas entradas, lo cual es una vía indirecta de modificarlos.

4.1.1 Creación de un Directorio – mknod y mkdir

Hemos visto que para crear un archivo ordinario disponemos de dos llamadas, una de ellas es open, con el indicador `O_CREAT` activo, y la otra es `creat`, pero en Unix hay otros tres tipos de archivos: directorios, archivos especiales y pipes. Para crear archivos de estos tres tipos se utiliza la función mknod. Su declaración es como sigue:

```
#include <sys/types.h>
#include <sys/stat.h>
int mknod(const char *path, mode_t mode, int dev);
```

`mknod` crea un archivo cuya ruta es la indicada por el parámetro `path`. El modo del archivo viene especificado en el parámetro `mode`. El significado de este parámetro es el mismo que vimos para la llamada `chmod`. En el archivo cabecera `<sys/stat.h>` hay definidas unas constantes simbólicas que podemos utilizar para construir el parámetro `mode` mediante el operador `OR` (`|`) a nivel de bits. A continuación, se muestran estas constantes:

Constante	Significado
S_IFMT **S_IFREG** **S_IFDIR** **S_IFCHR** **S_IFBLK** **S_IFIFO** **S_IFSOCK**	Tipo de archivo: Ordinario Directorio Especial modo carácter Especial modo bloque FIFO Conecto (Socket)
S_ISUID	Activar ID del usuario al ejecutar
S_ISGID	Activar ID del grupo al ejecutar
S_ISVTX	Stiky Bit
S_IRUSR	Permiso de lectura para el usuario
S_IWUSR	Permiso de escritura para el usuario.
S_IXUSR	Permiso de ejecución para el usuario
S_IRGRP	Permiso de lectura para el grupo
S_IWGRP	Permiso de escritura para el grupo
S_IXGRP	Permiso de ejecución para el grupo
S_IROTH	Permiso de lectura para otros
S_IWOTH	Permiso de escritura para otros
S_IXOTH	Permiso de ejecución para otros

El parámetro `dev` tiene aplicación únicamente cuando vamos a crear un archivo de dispositivo (modo bloque o carácter), siendo ignorado en el resto de los casos. `dev` codifica el "`major number`" y el "`minor number`" del dispositivo y es dependiente de la implementación y de la configuración del sistema. `dev` puede ser creado con la macro `makedev`, definida en `<sys/sysmacros.h>`.

mknod sólo puede ser invocado por un usuario con privilegios de superusuario, excepto a la hora de crear un pipe, situación en la que cualquier usuario puede usar mknod. Por este motivo, mknod no es muy útil para crear nuevos directorios.

Si mknod se ejecuta satisfactoriamente, devuelve el valor 0; en caso contrario, devuelve -1 y en errno estará el código de error producido.

Para superar los problemas que plantea mknod a la hora de crear directorios, algunos sistemas suministran la llamada mkdir. Su declaración es la siguiente:

```
#include <sys/stat.h>
int mkdir(const char *path, mode_t mode);
```

mkdir crea un nuevo archivo directorio cuya ruta es la indicada en el parámetro path. Los bit de permiso del nuevo directorio son inicializados de acuerdo con el parámetro mode y son modificados por la máscara de modo de creación del proceso (ver umask(2)). Cada bit activado en la máscara de modo de creación aparecerá desactivado en la máscara de modo del directorio recién creado.

Las constantes simbólicas definidas en <sys/stat.h> y referidas a los bits de permiso (desde S_IRUSR, permiso de lectura de propietario, hasta S_IXOTH, permiso de ejecución por otros) pueden utilizarse para crear el parámetro modo que le pasamos a mkdir.

Si mkdir se ejecuta satisfactoriamente, devuelve el valor 0; en caso contrario, devolverá -1 y en errno estará el código del error producido.

mkdir puede escribirse a partir de mknod y cualquier usuario con los privilegios de superusuario podrá ejecutarlo de esta forma. Una posible secuencia de código para mkdir es:

```
#include <sys/stat.h>
#include <sys/types.h>
.
.
int mi_mkdir(const char *path, mode_t mode) {
      return mknod(path, S_IFDIR | mode, 0);
}
```

Desde la línea de comandos del sistema podemos crear directorios invocando el programa estándar `mkdir`.

4.1.2 Borrado de un directorio – rmdir

Si queremos borrar un directorio de la jerarquía del sistema, usaremos la llamada `rmdir`. Su declaración es:

```
int rmdir(const char *path);
```

`path` es un apuntador a la ruta del directorio que queremos borrar. Para que la llamada se ejecute satisfactoriamente, se debe cumplir: que el directorio esté vacío, es decir, que sus dos únicas entradas sean "." Y ".."; no sea directorio de trabajo de ningún proceso.

`rmdir` devuelve 0 si se ejecuta correctamente; en caso contrario, devuelve -1 y en `errno` se colocará el código del tipo de error producido.

4.1.3. Creación de Nuevas Entradas de un Directorio – link, unlink y REMOVE

Ya hemos visto que `open`, `creat` y `mknod` permiten crear nuevos archivos, lo que se traduce en crear nuevas entradas para los directorios donde se encuentran esos archivos. Los directorios, en realidad, se utilizan para asignarle un nombre de archivo a un nodo-i y puede haber varios nombres conectados con el mismo nodo-i. Este mecanismo se conoce como enlace y cada nodo-i tiene un campo para llevar la contabilidad del total de nombres de archivos que están unidos a él. Para enlazar un nombre con un nodo-i, se emplea `link`, cuya declaración se muestra a continuación:

```
#include <sys/types.h>
#include <sys/stat.h>
int link(const char *path1, const char *path2);
```

`path1` debe apuntar a la ruta de un archivo existente y a cuyo nodo-i queremos unir otro nombre de archivo que viene indicado en `path2`. `path2` es añadido como nueva entrada en el directorio que se indique. Así, por ejemplo, la llamada:

```
link("/bin/ls", "/usr/local/bin/dir");
```

Hace que el programa `ls` que está en el directorio `/bin` pueda ser accedido como `dir` desde el directorio `/usr/local/bin`. Así, si en la variable de entorno `PATH` figura el directorio `/usr/local/bin`, se podrá ejecutar `ls` escribiendo `dir`.

`link` devuelve 0 cuando se ejecuta correctamente y -1 cuando se produce algún error; en este caso, `errno` tendrá el código del tipo de error producido.

Con `unlink` vamos a poder deshacer el enlace existente entre un nombre de archivo y su nodo-i asociado, eliminando así una entrada de directorio. La declaración de `unlink` es:

```
#include <sys/types.h>
#include <sys/stat.h>
int unlink(const char *path);
```

Donde `path` es un apuntador a la ruta del archivo que queremos eliminar del directorio. Cuando todos los enlaces de un archivo son borrados y no hay ningún proceso que tenga abierto el archivo, el espacio ocupado en el disco es liberado y el archivo deja de existir. Si uno o más procesos tienen abierto el archivo cuando se rompe el último enlace, el borrado del archivo se pospone hasta que se cierran todas las referencias al archivo.

`unlink` devuelve 0 si se ejecuta satisfactoriamente, y –1 en caso contrario.

También es posible remover un directorio o archivo con la función `remove`. Para un archivo, `remove` funciona de manera idéntica a `ulink`. Para un directorio, `remove` trabaja igual que `rmdir`.

```
#include <stdio.h>
int remove(const char *path);
```

4.1.2 Enlaces Simbólicos

Un enlace simbólico es un apuntador indirecto a un archivo, a diferencia de un enlace duro que apunta directamente al nodo-i de un archivo. Los enlaces simbólicos fueron introducidos con el fin de evitar las limitaciones de los enlaces duros:

- Los enlaces duros normalmente requieren que la liga y el archivo se encuentren en el mismo sistema de archivos.
- Solo el superusuario puede crear un enlace duro hacia un archivo.

El enlace simbólico no tiene limitaciones con respecto al sistema de archivos al que hace referencia, y cualquiera puede hacer un enlace simbólico hacia un directorio. Los enlaces simbólicos son típicamente usados para mover un archivo o un directorio entero hacia otra localidad.

Cuando usamos funciones que hacen referencia al nombre de un archivo por su ruta de acceso, siempre necesitamos conocer si la función es capaz de manejar el enlace simbólico. Si la función lo hace, seguirá el enlace; a pesar de la ruta que se le dé. Al hacer esto, la función trabajará con el archivo al cual se hace referencia. De otra forma, la función solo trabajará con el archivo de enlace simbólico. Las funciones `mkdir`, `mkfifo`, `mknod` y `rmdir` no son capaces de hacer esto y regresan un error cuando la ruta es un enlace simbólico. De manera similar, aquellas funciones que manejan un descriptor, como `fstat` y `fchmod`, no son capaces de seguir el descriptor de un enlace simbólico creado con alguna otra función (usualmente open).

`open` es capaz de seguir un enlace símbolo, excepto cuando esta función es llamada con los parámetros `O_CREAT` y `O_EXCL`. En este caso, si la ruta hace referencia a un enlace simbólico, open fallará y `errno` tendrá el valor `EEXIST`. Esto se hace con el fin de evitar que un proceso privilegiado no pueda ser engañado a escribir en los archivos equivocados.

Otro caso especial del `open` se da cuando se intenta abrir un enlace simbólico y el archivo al cual hace referencia el enlace no existe, open regresa un mensaje de error indicando que el archivo no se puede abrir.

4.1.3 Funciones symlink y readlink

Un enlace simbólico puede ser creado con la función `symlink`.

```
#include <unistd.h>
int symlink(const char *sympath, const char *actualpath);
```

Una nueva entrada es creada en el directorio, `sympath`, y hace referencia a `actualpath`. No es necesario que `actualpath` exista al momento de crear el enlace simbólico. De igual forma, `actualpath` y `sympath` no necesitan pertenecer al mismo sistema de archivos.

Debido a que `open` es capaz de seguir un enlace simbólico, necesitamos una manera de abrir el enlace simbólico y leer el nombre del enlace. La función `readlink` hace esto.

```
#include <unistd.h>
ssize_t readlink(const char* restrict path,  const char *restrict buf, size_t bufsize);
```

Esta función combina las acciones de `open`, `read` y `close`. Si la función termina exitosamente, regresa el nombre de bytes puestos en `buf`. El contenido del enlace simbólico, regresado dentro de `buf`, no tiene carácter nulo de terminación.

4.1.4 Directorios Asociados a un Proceso – chdir, chroot

Cuando se arranca un proceso de Unix, se le asignan una serie de directorios sobre los que va a trabajar por defecto mientras no se indique lo contrario.

El primer directorio es el directorio de trabajo actual. Este es el directorio que se va a considerar como inicial cuando se indique el nombre de un archivo mediante una ruta relativa. El nombre que tiene asociado el directorio de trabajo actual es "`.`".

Otro directorio importante es el directorio raíz "`/`", que le indica al proceso cuál es el archivo de directorio al que se le asigna el nombre "`/`". Por lo general, el directorio raíz es el de arranque del sistema y no se suele modificar.

Para cambiar el directorio de trabajo actual de un proceso se utiliza la función `chdir`, que se declara como sigue:

```
#include <sys/types.h>
#include <sys/stat.h>
int chdir(const char *path);
```

`path` es la ruta del nuevo directorio de trabajo. `chdir` devuelve 0 si se ejecuta bien, y -1 si se produce algún error.

En la biblioteca estándar C hay una función que devuelve la ruta absoluta del directorio de trabajo actual (consultar `getcdw(3)`). La declaración de esta función es:

```
char* getcwd(char *buf, size_t size);
```

`buf` es el apuntador a la zona de memoria donde se va a guardar la ruta del directorio de trabajo y `size` es el tamaño de buf. Si se ejecuta correctamente, `getcwd` regresa el mismo apuntador que le pasamos a través de buf; en caso contrario, devuelve `NULL`. La siguiente secuencia de código hace una llamada a `getcwd`:

```
char cwd[256];
..
getcwd(cwd, sizeof(cwd) – 1);
```

Para cambiar el directorio raíz que tiene asociado un proceso se emplea la función `chroot`. Su declaración es:

```
#include <sys/types.h>
#include <sys/stat.h>
int chroot(const char *path);
```

`path` apunta a una cadena con la ruta del directorio que actuará como nuevo directorio raíz. Para poder ejecutar esta llamada hay que tener privilegios de superusuario. Después de ejecutarse correctamente, `chroot` devuelve el valor 0; en caso contrario, devuelve -1 y en errno el código del error producido.

`chroot` se usa cuando estamos desarrollando alguna aplicación que actúa sobre los archivos de configuración del sistema. Imaginemos, por ejemplo, que estamos desarrollando un programa para añadir nuevos usuarios al sistema. Este programa deberá modificar, entre otros, el archivo `/etc/passwd`. Como habrá otros usuarios trabajando en el sistema mientras nosotros desarrollamos el programa, no podremos tocar los archivos de configuración, ya que se producirían interferencias molestas y si por eso dañamos algunos de los archivos de configuración, el sistema podría quedar inutilizado. La solución es cambiar el directorio raíz de nuestros procesos y engañarlos haciendo que cuando modifiquen `/etc/passwd` estén modificando realmente otro archivo. Así, si hacemos que el nuevo directorio raíz sea el directorio `/users/desarrollo`, por ejemplo, cuando el programa modifique `/etc/passwd`, en realidad estará modificando el archivo `passwd` situado en el directorio `/users/desarrollo/etc`.

4.1.5 Biblioteca Estándar de acceso a Directorios

Ahora estudiaremos una serie de funciones que nos van a permitir leer el contenido de un directorio sin que tengamos que preocuparnos por la estructura de este. Esta biblioteca inicialmente fue implementada para el sistema 4.3BSD, que introducía una estructura de directorios más genérica que la del System V, pero no tardó en ser adoptada por casi todos los sistemas Unix.

La filosofía de esta biblioteca es muy parecida a la de cualquier biblioteca de alto nivel pensada para realizar entrada/salida de archivos. Así, va a tener funciones para abrir y cerrar directorios (`opendir` y `closedir`), para leer entrada de un directorio (`readdir`) y para controlar la posición del apuntador de lectura (`seekdir`, `telldir` y `rewinddir`).

4.1.5.1 Apertura de un Directorio – opendir

La función para abrir un archivo directorio es `opendir` y su declaración es la siguiente:

```
#include <sys/types.h>
#include <dirent.h>
DIR* opendir(const char *dirname);
```

`dirname` es un apuntador a la ruta del directorio que queremos abrir. Después de ejecutarse correctamente la llamada, `opendir` devuelve un apuntador a una estructura del tipo `DIR`. Esta estructura la llamaremos stream de directorio y la utilizarán otras funciones de la familia para identificar al directorio sobre el que deben trabajar.

`opendir` hace una llamada a `malloc` para reservar memoria para el stream de directorio que va a devolver. Si se produce un error en la apertura del directorio, opendir devolverá el valor de `NULL` y en la variable `errno` colocará el código del tipo de error producido.

El tipo `DIR` está definido en `<dirent.h>` y su estructura es la siguiente:

```
typedef struct  __dirdesc {
      int dd_fd; /* Descriptor del archivo asociado al directorio. */
      long dd_loc; /* Offset en el buffer actual. */
      long dd_size; /* Número de datos devueltos por getdirentries. */
      long dd_bsize; /* Cantidad de entradas leídas al mismo tiempo. */
      char *dd_buf; /* Buffer de datos. */
} DIR;
```

La siguiente puede ser una secuencia de código para abrir a un directorio:

```
#include <dirent.h>
..
DIR *dir;
char *directorio = "/usr/include/sys");
..
if ((dir = opendir(directorio) == NULL) {
      perror(directorio);
      exit(-1);
}
```

4.1.5.2 Lectura de las Entradas de un Directorio – readdir

Para leer las entradas de un directorio abierto con `opendir`, emplearemos la función `readdir`. Su declaración es:

```
#include <sys/types.h>
#include <dirent.h>
struct dirent *readdir(DIR *dirp);
```

`dirp` es el apuntador a un stream de directorio ya abierto. Si funciona correctamente, `readdir` va a leer la entrada donde se encuentre situado el apuntador de lectura de ese directorio y va a actualizar el apuntador a la siguiente entrada para permitir así una lectura de tipo secuencial. `readdir` va a devolver la entrada leída a través de un apuntador a una estructura de tipo estructura `dirent` (también definida en el archivo cabecera `<dirent.h>`). `readdir` devuelve `NULL` cuando llega al final del directorio o cuando se produce algún error de lectura.

Internamente, `readdir` no reserva memoria para la estructura `dirent`, ya que la dirección que va a devolver corresponde a una variable local estática. Así, cuando escribamos código con `readdir`, no debemos preocuparnos por llamar a `free` para liberar las estructuras devueltas. La composición del tipo `struct dirent` es la siguiente:

```
struct dirent {
       ino_t d_ino; /* Nodo-i asociado a esta entrada de directorio. */
       off_t d_off; /* Desplazamiento a la siguiente entrada. */
       unsigned short d_reclen; /* Longitud de esta entrada. */
       unsigned char d_type; /* Tipo de archivo. */
       char d_name[_MAXNAMLEN + 1]; /* Nombre del archivo. */
};
```

4.1.5.2 Cierre de un Directorio – closedir

La declaración de `closedir` es:

```
#include <sys/types.h>
#include <dirent.h>
int closedir(DIR *dirp);
```

`dirp` es el stream del directorio que queremos cerrar. `closedir` se va a encargar de llamar a `free` para liberar la memoria reservada por `malloc`, y de llamar a `close` para cerrar el archivo abierto por la llamada `open` realizada internamente en `opendir`. Si se realiza satisfactoriamente, `closedir` devuelve 0, y en caso contrario, -1, indicando el tipo de error producido a través de `errno`.

4.1.5.4 Control del Apuntador de Lectura de un Directorio – seekdir, telldir y rewinddir

`seekdir` permite situar el apuntador de lectura de un directorio.

`telldir` devuelve la posición actual del apuntador de lectura de un directorio.

`rewinddir` sitúa el apuntador de lectura al principio del directorio y deja su stream asociado tal y como quedó después de la llamada a `opendir`.

Las declaraciones de estas funciones son las siguientes:

```
#include <sys/types.h>
#include <dirent.h>
void seekdir(DIR *dirp, long loc);
long telldir(DIR *dirp);
void rewinddir(DIR *dirp);
```

Todas estas funciones reciben como primer parámetro `dirp`, el stream del directorio sobre el que queremos trabajar. `seekdir` recibe, además, un segundo parámetro `loc`, que es el valor de la entrada a la que queremos en la que se coloque el apuntador de lectura.

Después de ejecutarse correctamente, `telldir` devuelve la posición actual del apuntador de lectura. Si falla, devuelve -1 y en `errno` sitúa el tipo de error producido.

4.1.5.5 Ejemplo – Programa tree

Como ejemplo de aplicación de las funciones de directorio anteriores, vamos a escribir el programa `tree`. Este programa debe ser invocado de la siguiente forma,

```
$ tree [opciones] directorio
```

Y lo vamos a utilizar para recorrer una jerarquía de directorios, `directorio` es el nombre del directorio a partir del cual queremos que se empiece a analizar la estructura del árbol. `opciones` representa los parámetros que se le pueden pasar a `tree`. Si `tree` no recibe ninguna opción, nos mostrará únicamente los nombres de los directorios que forman parte de la jerarquía analizada. Por ejemplo:

```
$ tree /home/manchas/cpps
compilador (d)
graficas (d)
       lib (d)
       include (d)
       bin (d)
       doc (d)
       src (d)
fileio (d)
uniones (d)
dirio (d)
```

Si `tree` recibe la opción `–f`, no sólo mostrará los directorios y subdirectorios, sino que también mostrará los archivos que guardan. Además, junto con el nombre del archivo, se presentará, entre paréntesis, su tipo según la siguiente tabla:

- `(d)` – directorio.
- `(o)` – ordinario.
- `(b)` – especial modo bloque.
- `(c)` – especial modo carácter.
- `(p)` – pipe.
- `(x)` – archivo ejecutable (cuando sea ejecutable por el propietario, por el grupo o por otros).

El resultado de la ejecución tree con este parámetro activo será:

```
$ tree –f /home/manchas/cpps
compilador
             (o) main.c
             (o) main.pre
             (o) main
graficas
```

```
	lib
	include
			(o) polinomio.h
```

El código del programa tree es el siguiente:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <dirent.h>
#include <string.h>

#define NEQ(str1, str2) (strcmp(str1, str2) != 0)

struct opciones {
	unsigned mostrar_archivos:1;
};

enum boolean {SI, NO};

void tree(const char *path, struct opciones opciones) {
	DIR *dirp;
	struct dirent *dp;
	static nivel = 0;
	struct stat buf;
	int ok, i;
	char archivo[256], tipo_archivo;

	if ((dirp = opendir(path)) == NULL) {
		perror(path);
		return;
	}

	while((dp = readdir(dirp)) != NULL) {
		if (NEQ(dp->d_name, ".") && NEQ(dp->d_name, "..")) {
			sprintf(archivo, "%s/%s", path, dp->d_name);
			ok = stat(archivo, &buf);
			if (ok != -1 && ((buf.st_mode & S_IFMT) == S_IFDIR)) {
				for (i = 0; i < nivel; i++) {
					fprintf(stdout, "\t");
				}
				fprintf(stdout, "%s %s\n",
					dp->d_name,
					(opciones.mostrar_archivos? "(d)" : ""));
				++nivel;
				tree(archivo, opciones);
				--nivel;
			}
			else if (ok != -1 && opciones.mostrar_archivos == SI) {
				for (i = 0; i <= nivel; i++) {
					fprintf(stdout, "\t");
				}
				switch (buf.st_mode & S_IFMT) {
					case S_IFREG:
						tipo_archivo = (buf.st_mode & 0111)? 'x' :
'o';
						break;
					case S_IFCHR:
```

```
                                        tipo_archivo = 'c';
                                        break;
                                case S_IFBLK:
                                        tipo_archivo = 'b';
                                        break;
                                case S_IFIFO:
                                        tipo_archivo = 'p';
                                        break;
                                default:
                                        tipo_archivo = '?';
                                }
                                fprintf(stdout, "(%c) %s\n", tipo_archivo, dp->d_name);
                        }
                }
        }
        closedir(dirp);
}

int main(int argc, char *argv[]) {
        struct opciones opciones = {NO};
        int i, j;

        if (argc < 2) {
                fprintf(stderr, "forma de uso: %s [-f] directorio\n", argv[0]);
                return -1;
        }
        for (i = 1; i < argc; i++) {
                if (argv[i][0] == '-') {
                        for (j = 1; j < argv[i][j] != '\0'; j++) {
                                switch (argv[i][j]) {
                                case 'f':
                                        opciones.mostrar_archivos = SI;
                                        break;
                                default:
                                        fprintf(stderr,        "opcion        [-%c]
desconocida\n", argv[i][j]);
                                        return -1;
                                }
                        }
                }
        }
        for (i = 1; i < argc; i++) {
                if (argv[i][0] != '-') {
                        tree(argv[i], opciones);
                }
        }
}
```

4.2 Acceso a Archivos Especiales

Otro de los grupos de archivos que se pueden manipular en Unix, está constituido por los archivos especiales. Dentro de este grupo se engloban prácticamente todos los periféricos que hay conectados a la computadora. El gran acierto en la concepción del sistema Unix reside en el esfuerzo realizado para no darle trato especial a los periféricos, y conseguir que puedan ser manipulados como cualquier otro archivo.

Un dispositivo periférico va a ser accesible a través de su archivo de dispositivo asociado. Este archivo de dispositivo se encuentra en `/dev`. Para leer o escribir datos en un periférico, procederemos de igual forma que con un archivo ordinario: abriremos su archivo de dispositivo mediante un `open`, leeremos de él mediante `read` y escribiremos datos en él mediante `write`. Los datos que maneja un dispositivo dependerán de la naturaleza de éste y será responsabilidad del programa interpretar qué es lo que está leyendo del dispositivo, y enviarle datos que tengan significado para él.

4.2.1 Entrada/Salida sobre Terminales

Las terminales son dispositivos especiales que trabajan en modo carácter. Todo proceso que se ejecuta en Unix tiene asociados 3 descriptores de archivo que le dan acceso a su terminal de control. Estos descriptores son: 0 para entrada estándar, 1 para salida estándar y 2 para la salida estándar de errores. El archivo de dispositivo que permite a un proceso acceder a su terminal de control es `/dev/tty`. Si el sistema no reservara de manera automática los descriptores anteriores, podríamos hacerlo nosotros con las siguientes llamadas:

```
close(0);
open("/dev/tty", O_RDONLY);
close(1);
open("/dev/tty", O_WRONLY);
close(2);
open("/dev/tty", O_WRONLY);
```

En el sistema hay una terminal especial llamada consola (dispositivo `/dev/console`) que es empleada durante el arranque para sacar los mensajes relativos a este proceso. Cada usuario que inicia una sesión de trabajo interactiva, lo hace a través de una terminal. Esta terminal tiene asociado un archivo de dispositivo que localmente se puede abrir como `/dev/tty`, pero que visto por otros usuarios tiene la forma `/dev/ttyXX`, donde XX representa un número de dos dígitos.

Como ejemplo para ilustrar el acceso a terminales, vamos a escribir una versión simplificada de la orden `write`. Esta orden se emplea para enviar mensajes a los usuarios que hay conectados en el sistema. Su forma de uso es:

```
$ write usuario
Línea de texto1
Línea de texto2
…
Línea de textoN
^D
```

Esta secuencia hará que el usuario reciba, a través de su terminal, las n líneas de texto que le enviamos.

Para poder enviar el mensaje, tenemos que saber si el usuario existe y si tiene iniciada una sesión de trabajo. También hay que conocer cuál es el archivo de dispositivo que tiene asociado a su terminal. Para poder obtener respuesta a estas dos preguntas, hay que consultar el archivo `/etc/utmp`. Este archivo es gestionado por el sistema y contiene información administrativa de

los procesos que hay ejecutándose en un instante determinado. En la entrada `utmp(4)` del manual podemos ver una explicación detalla de la estructura de este archivo.

Para hacer que la lectura de `utmp` sea independiente de la estructura, vamos a emplear la función estándar `getutent`. Su declaración es:

```
#include <sys/types.h>
#include <sys/utmp.h>
struct utmp* getutent();
```

Con cada llamada a `getutent` se lee un registro del archivo `/etc/utmp`. Si el archivo no está abierto, la llamada se encarga de abrirlo. Después de leer un registro, la función devuelve un apuntador a una estructura del tipo `utmp`, definida en el archivo cabecera `<sys/utmp.h>`. Cuando `getutent` llega al final del archivo, devuelve un apuntador a `NULL`. La definición de la estructura `utmp` es la siguiente:

```
struct utmp {
       char ut_user[8]; /* Nombre del usuario.*/
       char ut_id[4]; /* Identificador de "/etc/inittab". */
       char ut_line[12]; /* Nombre del archivo de dispositivo asociado
                            ("console", "ttyXX", "lnXX", etc.). */
       pid_t ut_pid; /* Identificador del proceso. */
       short ut_type; /* Tipo de entrada:
                           EMPTY
                           RUN_LVL
                           BOOT_TIME
                           OLD_TIME
                           NEW_TIME
                           INIT_PROCESS
                           LOGIN_PROCESS
                           USER_PROCESS
                           DEAD_PROCESS
                           ACCOUNTING */
       struct exit_status {
              short e_terminatio; /*Estado del terminación del proceso. */
              short e_exit; /* Estado de salida del proceso. */
       }
       unsigned short ut_reserved1; /* Se aplica a las entradas cuyo tipo
                                       es DEAD_PROCESS. */
       char ut_host[16]; /* Nombre del nodo. */
       unsigned long ut_addr; /* Dirección internet del nodo. */
};
```

La forma de averiguar si un usuario está o no conectado al sistema, es buscar una entrada en el archivo `utmp` cuyo campo `ut_user` coincida con el nombre del usuario que buscamos. Para saber cuál es el archivo de dispositivo que tiene asociado su terminal, tenemos que utilizar el campo `ut_line`.

A continuación, veremos el código del programa `mensaje`, que es el equivalente a la orden estándar `write`.

```
#include <fcntl.h>
#include <utmp.h>
#include <stdio.h>
```

```
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[]) {
	int tty;
	char terminal[20], mensaje[256], *logname;
	struct utmp *utmp;  //, *getutent();

	if (argc != 2) {
		fprintf(stderr, "Forma de uso: %s usuario\n", argv[0]);
		return -1;
	}

	while ((utmp = getutent()) != NULL &&
		strncmp(utmp->ut_user, argv[1], 8) != 0);
	if (utmp == NULL) {
		fprintf(stderr, "El usuario %s no esta en sesion\n", argv[1]);
		return -1;
	}
	sprintf(terminal, "/dev/%s", utmp->ut_line);
	if ((tty = open(terminal, O_WRONLY)) < 0) {
		perror(terminal);
		return -1;
	}

	logname = getenv("LOGNAME");
	sprintf(mensaje,  "\n\t\tMENSAJE  PROCEDENTE  DEL  USUARIO  %s\07\07\07\n",
logname);
	write(tty, mensaje, strlen(mensaje));

	while (fgets(mensaje, 255, stdin) != NULL) {
		write(tty, mensaje, strlen(mensaje));
		sprintf(mensaje, "\n");
		write(tty, mensaje, strlen(mensaje));
	}

	sprintf(mensaje, "\n<EOT>\n");
	write(tty, mensaje, strlen(mensaje));
	close(tty);

	return 0;
}
```

4.2.2 Control de Terminales

Para programar la forma de trabajo de un dispositivo, Unix tiene definida la llamada `ioctl`. Esta llamada se puede aplicar a todos los dispositivos que soportan el acceso en modo carácter, y su declaración es:

```
#include <sys/ioctl.h> /*BSD y Linux*/
#include <unistd.h> /*System V*/
int ioctl (int filedes, int request, arg);
```

`filedes` es el descriptor de un archivo previamente abierto o creado.

`request` codifica el tipo de operación que vamos a realizar y los valores que puede tomar dependen del tipo de dispositivo con el que estemos trabajando.

`arg` contiene los parámetros con los que vamos a programar el dispositivo. Si la operación es de lectura del estado del dispositivo, `arg` contendrá los valores con los que ya está programado.

Tanto `request` como `arg` van a depender del tipo de dispositivo que queremos controlar. En la sección 7 del manual de Unix encontrarás una descripción detallada de todos los dispositivos que el fabricante de una computadora pone a nuestra disposición. Para el caso de terminales asíncronos, la entrada a consultar es `termio(7)`. Esta entrada es muy extensa, por lo que no vamos a detenernos en considerar todos y cada uno de los pormenores relativos al control de una terminal, sobre todo teniendo en cuenta que muchos detalles apenas si se usan en la mayoría de los programas.

4.3 Administración del Sistema de Archivos

Ahora estudiaremos una serie de llamadas al sistema diseñadas para realizar tareas de administración de los sistemas de archivos. Muchas de las órdenes de administración vistas en el tema anterior están construidas a partir de estas llamadas.

4.3.1 Montaje y Desmontaje de un sistema de archivos – mount y umount

Sabemos que un sistema de archivos es accesible a través del archivo de dispositivo del disco o sección del disco sobre la que está montado. También es accesible a través de la jerarquía de directorios, pero para ello hay que montarlo previamente sobre un directorio de entrada que se va a convertir en el directorio raíz del sistema montado.

Podemos montar un sistema de archivos desde un programa mediante la función `mount`, cuya sintaxis se muestra a continuación:

```
int mount (const char *spec, const char *dir, int rwflag);
```

`spec` es un apuntador a la ruta del archivo de dispositivo del disco donde se encuentra el sistema de archivos que vamos a montar.

`dir` es un apuntador a la ruta del directorio sobre el que se va a montar el sistema de archivos. SI la llamada se ejecuta satisfactoriamente, las referencias a `dir` van a dar acceso al directorio raíz del sistema montado.

El bit menos significativo de `rwflag` se utiliza para revisar los accesos de escritura sobre el sistema de archivos. Si vale 1, la escritura estará prohibida, por lo que sólo se podrán hacer acceso de lectura; en caso contrario, la escritura estará permitida, pero de acuerdo con los permisos individuales de cada archivo.

Si la llamada se ejecuta satisfactoriamente, devolverá el valor 0; en caso contrario, devolverá -1 y en `errno` se encontrará el código del error producido.

El siguiente es un ejemplo de llamada a `mount` para montar la partición 2 del disco sobre el directorio `/usr`. El sistema se monta en modo lectura/escritura:

```
mount ("/dev/sda2", "/usr", 0);
```

Cuando un sistema de archivos deja de ser utilizado, puede ser desmontado. La función para llevar a cabo esta acción es `umount` y su declaración es la siguiente:

```
int umount (const char *name);
```

`name` es un apuntador a la ruta del archivo de dispositivo que da acceso al sistema de archivos que queremos desmontar.

Las llamadas `mount` y `umount` no actualizan el archivo `/etc/mtab`, que contiene la tabla de todos los sistemas de archivos montados. Por lo tanto, si decidimos montar un sistema desde programa, habrá que actualizar también desde programa ese archivo.

4.3.3 Estado de un Sistema de Archivos

La información administrativa y estadística de un sistema de archivos se encuentra en su superbloque. Para acceder a los aspectos más relevantes de esta información podemos usar la función `ustat`. Su declaración es la siguiente:

```
#include <sys/types.h>
#include <ustat.h>
int ustat (dev_t dev, struc ustat *buf);
```

`dev` es el número de dispositivo de la sección del disco donde se encuentra el sistema de archivos. Este número codifica los `major` y `minor number` del dispositivo.

`buf` es un apuntador a una estructura de tipo `struct ustat` donde la llamada devolverá los datos más importantes del sistema de archivos. Esta estructura se define como sigue:

```
struct ustat {
       daddr_t f_tfree; /* Número de bloques libres. */
       ino_t f_tinode; /* Número de nodos-i libres. */
       char f_fname[6]; /* Nombre del sistema de archivos. */
       char f_fpack[6]; /* Nombre del paquete del sistema de archivos. */
};
```

La llamada trabaja con el parámetro `dev` (número de dispositivo), que es incómodo de manejar. A través de la llamada `stat` podemos obtener el número de dispositivo asociado a un archivo de dispositivo especificado a través de su ruta, por lo que la forma más cómoda de trabajar con `ustat` es combinar esta llamada con `stat`. En las siguientes líneas de código se muestra la forma de llamar a `ustat` para leer la información relativa al sistema de archivos que hay en `/dev/root`.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <ustat.h>
..
struct stat sbuf;
struct ustat ubuf;
..
if (stat("/dev/root", &sbuf) < 0) {
       /* ERROR */
```

```
} else if (ustat(sbuf.st_dev, &ubuf) < 0) {
      /* ERROR */
} else {
      /* PROCESAR DATOS LEIDOS */
}
```

La información de devuelve `ustat` es pobre en comparación con la que se almacena en el superbloque. La forma de acceder al resto de la información es tratando directamente con el archivo de dispositivo que da acceso al sistema de archivos. Para ello, hay que abrir ese archivo mediante una llamada `open`. Es conveniente abrir el archivo en modo sólo lectura, para evitar posibles daños debidos a una manipulación inadecuada del superbloque. Una vez abierto el archivo de dispositivo, hay que posicionarse en su segundo bloque físico mediante `lseek`. Recordemos que el primer bloque está dedicado al programa `boot`. El contenido del superbloque se debe leer con `read`, recogiendo los datos en una estructura adecuada. El fabricante del sistema incluye en el archivo cabecera `<sys/filsys.h>` la definición de la estructura del superbloque. Lamentablemente, esta estructura no está estandarizada en su definición ni en su nombre, por ello los programas que traten directamente con el superbloque no va a ser transportables a nivel de código fuente.

Referencias

Márquez, F. (2004). *Unix Programación Avanzada.* Colombia: Alfa-Omega.

Stevens, W. R. (2008). *Advanced programming in the UNIX environment.* Addison-Wesley.