Introducción a Sistemas Operativos: Ficheros

Clips xxx
Francisco J Ballesteros

1. Lectura de directorios

Los directorios en UNIX son ficheros que contienen una tabla de ficheros pertenecientes al directorio. En cada entrada de la tabla hay dos cosas:

- Un nombre de fichero
- Un número (de *i-nodo*) que identifica al fichero.

Cuando se crea un fichero con `creat`, `open`, o `mkdir` se añade automáticamente la entrada de directorio para el fichero o directorio que se ha creado. Recordarás que `link` (o `ln`) puede usarse para establecer nuevos enlaces para ficheros que existen (nuevas entradas de directorio con el número de un fichero existente) y que `unlink` o `rmdir` pueden utilizarse para eliminar nombres.

Hace tiempo UNIX permitía utilizar `read` y `write` para leer y escribir directorios. No obstante, esto causó tantos problemas cada vez un usuario escribía incorrectamente la tabla que desde hace tiempo no se puede utilizar `read` o `write` para leer una tabla de directorio.

En lugar de *read(2)*, para directorios tenemos la llamada al sistema *getdirentries(2)*. No obstante, es mucho más fácil utilizar las funciones *opendir(3)*, *readdir(3)* y *closedir(3)* que tenemos en la librería de C. No sólo es más fácil. Es más portable. Piensa que el formato exacto de una entrada de directorio depende del tipo de sistema de ficheros que utilices (del programa que implementa los ficheros en esa partición o donde quiera que estén los ficheros, y que forma parte del kernel).

Aquí tienes un ejemplo de cómo listar un directorio en C.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/dir.h>
#include <dirent.h>
#include <err.h>

int
main(int argc, char* argv[])
{
    DIR *d;
    struct dirent *de;

    d = opendir(".");
    if(d == NULL) {
        err(1, "opendir");
    }
    while((de = readdir(d)) != NULL) {
        printf("%s\n", de->d_name);
    }
    if (closedir(d) < 0) {
        err(1, "closedir");
    }
    exit(0);
}
```

Este programa lista el contenido del directorio actual. Por ejemplo,

```
unix$ lsdot
.
..
lsdot.c
lsdot
ch03e.w
unix$
```

Como verás, aparecen tanto "`.`" como "`..`".

Cada llamada a `readdir` devuelve una estructura de tipo `struct dirent`. Dicha estructura no debes liberarla, según indica el manual, y si llamas de nuevo a `readdir` la estructura antigua se sobreescribe con la nueva muy posiblemente.

Esta estructura cambia mucho de un tipo de UNIX a otro, aunque en la mayoría las entradas de directorio son similares. Por ejemplo, esta es la que utiliza Linux:

```
struct dirent {
    ino_t          d_ino;       /* inode number */
    off_t          d_off;       /* not an offset; see NOTES */
    unsigned short d_reclen;    /* length of this record */
    unsigned char  d_type;      /* type of file; not supported
                                   by all filesystem types */
    char           d_name[256]; /* filename */
};
```

Normalmente, siempre tienes el campo `d_name`. En Linux y sistemas BSD tienes `d_type` también. El resto de campos suelen variar de nombre y tal vez no estén en tu UNIX. Si utilizas algo más que `d_name`,

seguramente tu código deje de ser portable a otros tipos de UNIX.

Cuando tienes `d_type`, este campo vale `DT_DIR` para directorios, `DT_REG` para ficheros (ficheros regulares o normales) y otros valores para otros tipos de fichero que veremos más adelante.

Naturalmente, nunca se escribe un directorio, este se actualiza creando, borrando y renombrando los ficheros que contiene. Ya conoces todo lo necesario, salvo por cómo renombrar ficheros. La llamada al sistema en cuestión es *rename(2)*.

Este programa renombra un fichero, similar al uso más sencillo del comando *mv(1)*.

```
#include <stdio.h>
#include <stdlib.h>
#include <err.h>

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

    if (argc != 3) {
        fprintf(stderr, "usage: %s old new\n", argv[0]);
        exit(1);
    }
    if (rename(argv[1], argv[2]) < 0) {
        err(1, "rename %s", argv[1]);
    }
    exit(0);
}
```

Podemos usarlo para mover un fichero a notro nombre:

```
unix$ echo hola >a
unix$ cat a
hola
unix$ mvf a b
unix$ ls -l a b
ls: a: No such file or directory
-rw-------  1 nemo  staff  5 Aug 20 22:13 b
unix$ cat b
hola
unix$
```

Has de tener en cuenta que si el fichero de destino existe, se borra (su nombre) durante el *rename*. Además, es un error hacer un *rename* hacia un directorio no vacío. Y otra cosa, si recuerdas el comando *mv(1)*, verás que

```
unix$ mv a /tmp
```

mueve el fichero `a` a `/tmp/a`. No obstante,

```
unix$ mvf a /tmp
mvf: rename a: Permission denied
```

nuestro programa no es tan listo e intenta hacer que `a` tenga como nombre `/tmp` (que es un directorio que ya existe).

2. Globbing

Dado que gran parte de las "palabras" que escribimos en una línea de comandos corresponden a nombres de fichero, el shell da facilidades para expresar de un modo compacto nombres de fichero que tienen cierto aspecto. Dicho de otro modo, el shell permite utilizar expresiones que *generan* nombres de fichero o que *encajan* con nombres de fichero. A esta herramienta se la conoce como *globbing*.

Ya sabes que en general una buena forma de saber cómo funcionan las cosas es experimentar con ellas, así que vamos a crear un directorio con ciertos ficheros dentro para experimentar, tal y como harías tu.

```
unix$ mkdir -p /tmp/d/d2/d3
unix$ touch /tmp/d/a /tmp/d/ab /tmp/d/abc
unix$ touch /tmp/d/d2/j /tmp/d/d2/k
unix$ touch /tmp/d/d2/d3/x  /tmp/d/d2/d3/y
unix$
```

El flag `-p` de `mkdir` hace que se cree el directorio indicado como argumento y los directorios padre si es que no existen.

Podemos utilizar el comando *du(1)* (*disk usage*) para que liste el árbol de ficheros que hemos creado. Este comando en realidad lista cuándo disco consumen ficheros y directorios, pero es una forma práctica de listar árboles de ficheros (¡Más práctica que utilizando `ls`!).

```
unix$ du -a /tmp/d
0    /tmp/d/1
0    /tmp/d/2
0    /tmp/d/a
0    /tmp/d/ab
0    /tmp/d/abc
0    /tmp/d/d2/d3/x
0    /tmp/d/d2/d3/y
0    /tmp/d/d2/d3
0    /tmp/d/d2/j
0    /tmp/d/d2/k
0    /tmp/d/d2
0    /tmp/d
```

El flag `-a` de `du` hace que liste no sólo los directorios, sino también los ficheros.

Pero continuemos. Cuando el shell ve que un nombre en la línea de comandos contiene ciertos caracteres especiales, intenta encontrar paths de fichero que encajan con dicho nombre y, si los encuentra, cambia ese nombre los pos paths, separados por espacio. En realidad, deberíamos llamar *expresión* a lo que estamos llamando "nombre".

Por ejemplo, en este comando

```
unix$ echo /tmp/d/*
/tmp/d/1 /tmp/d/2 /tmp/d/a /tmp/d/ab /tmp/d/abc /tmp/d/d2
```

el shell ha tomado "`/tmp/d/*`" y, como puedes ver por la salida de `echo`, lo ha cambiado por los nombres de fichero en "`/tmp/d`".

Pero mira este otro comando:

```
unix$ echo /tmp/d/a*
/tmp/d/a /tmp/d/ab /tmp/d/abc
```

Aquí el shell ha cambiado "`/tmp/d/a*`" por los nombres de fichero en `/tmp/d` que comienzan por "`a`".

Veamos otro más...

```
unix$ echo /*/ls
/bin/ls
```

El shell ha buscado en "/" cualquier fichero que sea un directorio y contenga un fichero llamado "ls".

Las expresiones de globbing son fáciles de entender:

- El shell las recorre componente a componente mirando cada una de las subexpresiones que corresponden a nombres de directorio en el path.
- Naturalmente, "/" separa unos componentes de otros.
- En cada componente intenta encontrar todos los ficheros cuyo nombre encaja en la subexpresión para dicho componente.
- En cada componente podemos tener caracteres normales o alguna de estas expresiones:
  - "*" encaja con cualquier string, incluso con el string vacío.
  - "[...]" encaja con cualquier carácter contenido en los corchetes. Aquí es posible indicar rangos de caracteres como en "[a-z]" (las letras de la "a" a la "z").
  - "?" encaja con un sólo carácter.

Por ejemplo,

```
unix$ cd /tmp/d
unix$ ls
1    2    a    ab    abc    d2
unix$ echo [12]*
1 2
unix$ echo [ab]*
a ab abc
unix$
```

La primera expresión quiere decir: "un 1 o un 2 y luego cualquier cosa" La segunda quiere decir: "una a o una b y luego cualquier cosa".

Otro ejemplo:

```
unix$ ls /tmp/*/d*/?
/tmp/d/d2/j    /tmp/d/d2/k
```

Esto es, dentro de "/tmp", cualquier nombre de directorio que tenga dentro un directorio que comience por "d" y tenga dentro un fichero cuyo nombre sea un sólo carácter y nada más.

Quizá el uso más común sea en comandos como

```
unix$ ls *.[ch]
```

para listar los fuentes en C (ficheros cuyo nombre es cualquier cosa, luego un "." y luego o bien una "c" o una "h", y como

```
unix$ rm *.o
```

para borrar todos los ficheros objeto (cualquier nombre terminado en ".o").

Recuerda que ninguno de estos comandos sabe nada respecto a globbing, ni entienden qué quiere decir "*" ni lo ven siquiera. Es el shell cuando analiza la línea de comandos el que detecta que hay una expresión que contiene caracteres de globbing, y cambia dicha expresión por los nombres de los ficheros que encajan en la expresión. Los comandos simplemente reciben los argumentos con los paths y se limitan a hacer su trabajo.

En ocasiones resultará útil hacer que el shell no haga globbing, como en esta sesión:

```
unix$ touch '*'
```

Hemos creado un fichero llamado "*". Esta vez a sabiendas, pero podría haber sido por error. Para hacerlo simplemente utilizamos las comillas simples para hacer que el shell tome *literalmente* y sin cambiar nada lo que incluyen las comillas (y como una sóla palabra).

Y ahora tenemos el dilema.

```
unix$ ls
*    1    2    a    ab    abc    d2
unix$ echo *
* 1 2 a ab abc d2
```

Si utilizamos

```
unix$ rm *
```

para eliminar "*", ¡borraremos todos los ficheros! y no es lo que queremos. Pero podemos hacer esto:

```
unix$ rm '*'
unix$ ls
1    2    a    ab    abc    d2
```