Introducción a Sistemas Operativos: Ficheros

Clips xxx
Francisco J Ballesteros

1. Metadatos

Los ficheros, como ya sabes en este punto, no sólo tienen datos. Tienen además atributos, o datos sobre los datos, o **metadatos**. Por ejemplo, el *uid* del dueño del fichero, el *gid* del grupo al que pertenece, los permisos, el contador de referencias, etc.

En UNIX, todos estos metadatos se guardan en la estructura de datos que representa al fichero, que es un record que UNIX llama *i-nodo*. La llamada al sistema *stat(2)* sirve para recuperar los metadatos de un fichero. El siguiente programa mejora nuestro programa para listar directorios y muestra distintos atributos de cada fichero contenido en los directorios cuyo nombre se indica como argumento (el directorio actual si no se indica ninguno).

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/dir.h>
#include <sys/stat.h>
#include <dirent.h>
#include <string.h>
#include <time.h>
#include <err.h>
typedef unsigned long long uvlong_t;
static int
list(char *fname)
{
    struct stat st;

    if (stat(fname, &st) < 0)
        return -1;
    printf("%s:\n\t", fname);
    if ((st.st_mode & S_IFMT) == S_IFDIR)
        printf("dir");
    else if ((st.st_mode & S_IFMT) == S_IFREG)
        printf("file");
    else
        printf("type %xu", st.st_mode&S_IFMT);
    printf(" perms %o", st.st_mode & 0777);
    printf(" sz %llu", (uvlong_t) st.st_size);
    printf(" uid %d gid %d", st.st_uid, st.st_gid);
    printf("\n\tlinks %d", (int)st.st_nlink);
    printf(" dev %d ino %llu\n\t", (int)st.st_dev, (uvlong_t) st.st_ino);
    printf("mtime %llus = %s", (uvlong_t)st.st_mtime, ctime(&st.st_mtime));
    return 0;
}
```

El propósito de `list` es listar los atributos de fichero cuyo path se indica en `fname`. La llamada a `stat`

rellena una estructura `st` con dichos atributos. Igual que sucedía con la lectura de directorios, el contenido de esta estructura varía de unos sistemas UNIX a otros. Así que hay que tener cuidado para que el código sea portable. Los campos que utilizamos en nuestro programa suelen estar disponibles en todos los UNIX. En el caso de Linux, la estructura está definida así:

```
struct stat {
    dev_t     st_dev;     /* ID of device containing file */
    ino_t     st_ino;     /* inode number */
    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 ID (if special file) */
    off_t     st_size;    /* total size, in bytes */
    blksize_t st_blksize; /* blocksize for filesystem I/O */
    blkcnt_t  st_blocks;  /* number of 512B 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 status change */
};
```

El campo `st_mode` contiene tanto los permisos como otros bits que indican el tipo de fichero. La constante `S_IFMT` indica qué bits contienen el tipo de fichero. Además, tenemos constantes `S_IFDIR`, `S_IFREG`, etc. para comprobar qué tipo de fichero tenemos. Observa que utilizamos "`&`" dado que el campo `st_mode` tiene más bits y no es posible utilizar sólo "`==`".

Otros campos como `st_size`, `st_uid` y `st_gid` debieran ser obvios a estas alturas. El campo `st_nlink` indica cuántas referencias (nombres) tiene el fichero.

Para ver si dos ficheros son el mismo habría que comprobar si tanto `st_dev` como `st_ino` coinciden. El primero indica en qué dispositivo se encuentra el fichero y el segundo indica qué número de fichero dentro de dicho dispositivo (qué número de *i-nodo*) tenemos.

Los campos `st_mtime`, `st_atime` y `st_ctime` contienen las fechas de la última modificación del fichero, del último acceso a los datos del mismo y de la última vez que se cambiaron los atributos (o de cuándo se creó el fichero si no se han cambiado). Estas fechas están codificadas como un número de segundos desde una fecha data, pero podemos utilizar *ctime(3)* para convertir dicho número a un string con la fecha en un formato que un humano pueda leer. En breve veremos la gestión del tiempo en UNIX más despacio.

Lamentablemente, hoy en día, UNIX utiliza tipos como `ino_t`, `dev_t`, `size_t` y otros muchos en lugar de `int`, `long`, etc. Eso quiere decir que tendremos problemas para utilizar los formatos de `printf` con cada uno de ellos. Una solución portable para imprimir dichos valores es la que puedes ver en el código: convertimos el entero en cuestión a otro entero de mayor o igual tamaño, en nuestro caso a `unsigned long long`, y utilizamos el formato para dicho entero (en nuestro caso "`%llu`").

Habría sido mejor utilizar otro tamaño de entero y usar simplemente "`int`" y "`%d`" dado que hoy día la memoria es barata. Pero esto es tan sólo una opinión.

Para que puedas ver el programa en su conjunto y repasar cómo se leían directorios, comprobaban argumentos y otros detalles, mostramos a continuación el resto del fichero fuente que contiene nuestro programa. Observa cómo las funciones se toman molestias en devolver una indicación de error si tienen problemas y cómo se comprueban los errores e imprimen mensajes de error. En aquellos casos en que es posible continuar el trabajo tras informar de un error, el código hace justo eso. Además, se cierran los ficheros que

se abren, tengamos errores o no. Pero antes de verlo, esta es una ejecución del programa:

```
unix$ ll
./ll:
    file perms 755 sz 9092 uid 501 gid 20
    links 1 dev 16777220 ino 7791046
    mtime 1471771999s = Sun Aug 21 11:33:19 2016
./guide:
    file perms 644 sz 1396 uid 501 gid 20
    links 1 dev 16777220 ino 7766071
    mtime 1471700478s = Sat Aug 20 15:41:18 2016
./ll.c:
    file perms 644 sz 1731 uid 501 gid 20
    links 1 dev 16777220 ino 7788145
    mtime 1471771995s = Sun Aug 21 11:33:15 2016
./writef.c:
    file perms 644 sz 512 uid 501 gid 20
    links 1 dev 16777220 ino 7782848
    mtime 1471700968s = Sat Aug 20 15:49:28 2016
```

Y este es el código que falta:

```
static int
ldir(char *dir)
{
    DIR *d;
    struct dirent *de;
    char path[1024];
    int n, rc;

    d = opendir(dir);
    if(d == NULL) {
        warn("opendir: %s", dir);
        return -1;
    }
    rc = 0;
    while((de = readdir(d)) != NULL) {
        n = snprintf(path, sizeof path, "%s/%s", dir, de->d_name);
        if (n >= sizeof path-1) { // -1 for '\0'
            warn("path %s/%s too long", dir, de->d_name);
            rc = -1;
        } else if (list(path) < 0) {
            warn("list: %s", path);
            rc = -1;
        }
    }
    if (closedir(d) < 0) {
        warn("closedir: %s", dir);
        return -1;
    }
    return rc;
}
```

```
int
main(int argc, char* argv[])
{
    int sts, i;

    sts = 0;
    if (argc == 1) {
        if (ldir(".") < 0) {
            sts = 1;
        }
    } else {
        for (i = 1; i < argc; i++) {
            if (ldir(argv[i]) < 0) {
                sts = 1;
            }
        }
    }
    exit(sts);
}
```

Por cierto, en ocasiones tendrás un descriptor de fichero del que desees obtener los metadatos, en lugar de tener el path para dicho fichero. En ese caso puedes utilizar *fstat(2)* en lugar de *stat(2)*, que recibe un descriptor de fichero en lugar de un path. En muchas otras llamadas se sigue el mismo convenio y dispones de funciones que comienzan por "`f`" para utilizar un descriptor en lugar de un path para identificar el fichero con que quieres trabajar. ¡El manual es tu amigo!