Capítulo 15: Interludio. Archivos y directorios

En este capítulo añadiremos una pieza clave a la virtualización: el almacenamiento persistente. Un ejemplo de éstos es un disco duro o un disco SSD que guardan datos de manera permanente a diferencia de la memoria (RAM) que los datos se pierden cuando se apaga el dispositivo. El S.O. tiene que tener mucho cuidado con estos dispositivos

- Archivos y directorios

Dos abstracciones principales que se han desarrollado a la hora de guardar datos son los ficheros y los directorios

Los archivos son un array simple de bytes en los que se puede escribir y leer. Cada archivo tiene un nombre de bajo nivel, normalmente un número que el usuario no es consciente de su existencia. Por razones históricas, éste número es llamado como número de inodo

Los directorios, como los archivos, tienen un nombre de bajo nivel, pero su contenido es diferente. Contiene una lista de pares. Al colocar un directorio dentro de otro, los usuarios pueden construir arbitrariamente árboles de directorios en los que guardar directorios o archivos

La jerarquía de directorios empieza en un directorio raíz o root y usa un separador para indicar los subdirectorios. La dirección absoluta de un archivo será la dirección del directorio en el que está y el nombre del archivo

- Creando archivos

La manera más sencilla de crear un archivo es mediante la función open() y pasando como argumento el código O_CREAT para que cree con el nombre que digamos un archivo en el directorio actual

Esta rutina tiene una gran cantidad de códigos, pero sólo veremos O_CREAT para crear un archivo si no existe ya, O_WRONLY para asegurarnos de que sólo se pueda escribir y O_TRUNC para que si el archivo ya existe se eliminen los datos

Open() devuelve un descriptor de archivo, que es un integer, privado para cada proceso y usado en UNIX para acceder a archivos, suponiendo que tenemos permisos de acceso

- Leer y escribir archivos

Una vez que tengamos archivos querremos leerlos o escribir en ellos. Empezaremos leyendo un archivo ya existente. Usaremos el comando cat para que se imprima en pantalla su contenido

Lo primero que hace cat es abrir el archivo para su lectura. Tendremos en cuenta que open() devolverá en este caso un 3 ya que ya tenemos 2 archivos abiertos de manera predeterminada en un proceso, la salida y entrada estándar (estándar input/output)

- Lectura y escritura no secuencial

El acceso que hemos visto anteriormente a archivos era secuencial, esto es que podemos leer y escribir un archivo desde el principio al final

A veces es útil poder escribir o leer en un lugar específico dentro de un archivo. Para hacer esto podemos usar la función lseek() que tiene 3 argumentos

El primer argumento es el archivo que vamos a leer, el segundo argumento es el desplazamiento del archivo en una ubicación en particular y el tercer argumento determina exactamente dónde se va a realizar la búsqueda

Para cada archivo que abre un proceso, el S.O. localiza el desplazamiento actual que determina dónde empezará la siguiente lectura o escritura. Este desplazamiento se guarda en una estructura de cada archivo. Con esta estructura, el S.O. puede determinar cuando un archivo se abre si se puede leer o escribir en él. Estas estructuras representan todos los archivos abiertos en el sistema y se guardan en una tabla de archivos abiertos

Cuando se abre un fichero y se quiere leer haciendo la llamada read(), la tabla irá moviendo el desplazamiento del archivo hasta que nos pasemos o lo cerremos, un ejemplo será:

System Calls	Return Code	OFT[10] Current Offset	OFT[11] Current Offset
`fd1 = open("file", O_RDONLY);`	3	0	–
`fd2 = open("file", O_RDONLY);`	4	0	0
`read(fd1, buffer1, 100);`	100	100	0
`read(fd2, buffer2, 100);`	100	100	100
`close(fd1);`	0	–	100
`close(fd2);`	0	–	–

Podemos usar lseek para colocar el desplazamiento donde queramos, por lo que podemos elegir desde dónde empezar a leer el archivo

- Entradas de tabla de archivos compartidos: fork () y dup ()

En muchos casos, el mapeo del descriptor de archivos a una entrada en la tabla de archivos abiertos es una asignación uno a uno. Cundo un proceso se ejecuta y decide abrir un archivo, leerlo y cerrarlo, el archivo tendrá una única entrada en la tabla de archivos abiertos. Si en otro proceso se intente abrir ese mismo archivo a la vez, cada vez que se abra tendrá una entrada en la tabla

Sin embargo, hay casos en los que una entrada en la tabla es compartida. Uno de esos casos ocurre cuando un proceso padre crea un hijo con un fork(). Cuando una entrada de la tabla es compartida, la cuenta de referencia es incrementada. Sólo cuando ambos procesos cierran el archivo la entrada se elimina

Compartir entradas de la tabla entre padres e hijos es muy útil ya que se pueden crear varios procesos hijos y todos ellos trabajando con la misma entrada

Otra manera de compartir entradas es mediante la función dup(), y sus variantes dup2() y dup3(). Esta función permite al proceso crear un nuevo descriptor de archivos que haga referencia al mismo archivo abierto, teniendo este archivo ya un descriptor correspondiente

- Escribir inmediatamente con fsync ()

Muchas veces, cuando un programa usa la función write() es solo para decir que se escriban unos datos en la memoria. El archivo del sistema colocará estos datos en el buffer durante un tiempo y después, estos datos serán guardados en los dispositivos de almacenamiento

Muchos sistemas ofrecen un control adicional a las APIs para evitar que se escriba forzosamente en el disco todo el tiempo. Cuando un proceso usa la función fsync() para un descriptor de archivos, el archivo del sistema responderá forzando todos los datos de suciedad (parte no escrita aún) en el disco hacia el archivo al que hace referencia el descriptor del archivo. Fsync() termina una vez que todas esas escrituras se completan

- Renombrando archivos

Una vez tengamos un archivo, a veces es útil cambiarle de nombre. Cuando queremos esto usamos el comando mv para renombrarlo

Podemos saber que mv usa la función rename, que necesita dos argumentos, el nombre del fichero original y el nuevo nombre. Si el sistema deja de funcionar mientras esta función está en ejecución no habrá una mezcla de nombres, o mantiene su nombre antiguo o se cambia, no hay otra opción

- Obteniendo información sobre archivos

Más allá del acceso a los archivos, esperamos que el sistema de archivos mantenga una cantidad justa de información sobre cada archivo que está almacenando. Generalmente llamamos a esos datos metadatos de archivos. Estas llamadas llevan un nombre de ruta (o descriptor de archivo) a un archivo y completan una estructura de estado:

```
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 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
};
```

- Eliminar archivos

Para eliminar un archivo tendremos que usar el comando rm. Cuando hacemos esto se hace una llamada a la función unlink() que coge el nombre del archivo a eliminar y devuelve un 0 en caso de que se haya realizado correctamente. No se elimina como tal, pero para entender el porqué tendremos que saber más sobre archivos y directorios

- Creando directorios

Para crear un directorio tendremos que usar la llamada mkdir. Cuando un directorio es creado se considera vacío, aunque tiene un contenido mínimo. Estos directorios tienen dos entradas, una refiriéndose a sí mismo y otra al directorio padre

- Leyendo directorios

Ahora que hemos creado un directorio querremos leerlo. Para ello usaremos el comando ls. En vez de abrir el directorio como si fuese un archivo tendremos que usar otro tipo de llamadas. Ls usa tres llamadas, opendir(), readdir() y closedir() y con un simple bucle el programa va imprimiendo todos los nombres e inodo de cada archivo del directorio

- Eliminando directorios

Para terminar, para eliminar un directorio tendremos que usar el comando rmdir. Es distinto a la eliminación de archivos ya que eliminar un directorio es peligroso porque puedes terminar eliminando una gran cantidad de datos contenidos en él con un simple comando

Si se intenta eliminar un directorio no vacío con rmdir(), éste devolverá un fallo

- Enlaces duros

Volvemos al misterio del unlink(). La función link() necesita dos argumentos, una dirección vieja y una nueva. Cuando enlazas un nuevo archivo a uno viejo realmente estás creando una nueva manera de referencias a el mismo archivo. Link() simplemente crea un nuevo nombre en el directorio en el que estás y lo enlaza al mismo número de inodo que el archivo original

El motivo por el que unlink() funciona es porque cuando el archivo del sistema desenlaza un archivo, cambia la cuenta de referencias del número de inodo. Esta cuenta de referencia permite al archivo del sistema localizar cuántos archivos han sido enlazados a este inodo. Por tanto, se elimina este enlace estableciendo la cuenta a 0 haciendo que el archivo del sistema libere ese inodo, “eliminando” el archivo

- Enlaces simbólicos

Hay otra manera de enlazar que es muy útil y son llamados los enlaces simbólicos o enlaces suaves. Los enlaces duros están limitados ya que no puedes crear uno hacia un directorio ni a archivos en otras particiones del disco

Creando un enlace simbólico, el fichero original puede ser accedido mediante varios nombres

Sin embargo, son bastante diferentes estos tipos de enlaces. La primera diferencia es que un enlace simbólico es un archivo en sí. Si hacemos un ls veremos que el primer dato, en vez de ser un d de directorio, sería un l de un link

Es obvio que estos links tendrán menos peso que el archivo original ya que sus datos son el nombre del archivo al que están referidos

- Bits de permisos y listas de control de acceso

La abstracción de un proceso proviene de dos virtualizaciones, de la CPU y de la memoria. Cada una de ellas da la ilusión al proceso de que tiene su propia CPU y su propia memoria. El archivo del sistema también presenta una vista virtual del disco, transformándolo en unos bloques para guardar más fácilmente archivos y directorios

Un sistema más completo de mecanismos para permitir varios grados de uso compartido suele estar presente dentro de los sistemas de archivos. La primera forma de estos mecanismos son los bits de permisos en UNIX

Estos permisos consisten en tres grupos, el grupo del propietario del archivo, los pertenecientes a un mismo grupo y el resto. Los permisos son de leer, escribir y ejecutar

El propietario de los archivos puede cambiar estos permisos mediante el comando chmod

Más allá de los bits de permisos, algunos sistemas de archivos, como el sistema de archivos distribuido conocido como AFS, incluyen controles más sofisticados

- Hacer y montar un sistema de archivos

Para crear un sistema de archivos, muchos ofrecen una herramienta referida como mkfs que los crea. La idea es que, dada la herramienta, como entrada, un dispositivo y un sistema de archivos simplemente escribe en un sistema de archivos vacío, empezando un nuevo directorio root en es partición del disco

Sin embargo, una vez que se crea dicho sistema de archivos debe hacerse accesible dentro del árbol del sistema. Esta tarea se consigue a través del programa de montaje (mount()) que coge un directorio existente como un objetivo del puntero de montaje y copia un nuevo sistema de archivos en ese punto