Capítulo 16: Implementación del sistema de archivos

En este capítulo introduciremos una implementación simple de un sistema de archivos llamado como vsfs (very simple file system). Este sistema de archivos es una versión simplificada del típico usado en UNIC y nos sirve para introducir sus estructuras básicas, métodos de acceso y varias políticas que se encuentran en varios sistemas de archivos en la actualidad

- La forma de pensar

Al pensar en sistema de archivos, se sugiere que se piensen en dos aspectos diferentes. Si éstos se entienden, prácticamente se entenderá como un sistema de archivos funciona

El primero es la estructura de datos de un sistema de archivos. Es decir, qué tipos de estructuras en discos utiliza el sistema de archivos para organizar sus datos y metadatos. El primer sistema de archivos que veremos usa estructuras sencillas, como los array de bloques

El segundo aspecto de un sistema de archivos son los métodos de acceso. Cómo se mapean las llamadas de un proceso como las de open(), read() o write() en las estructuras

Si se entiende las estructuras de datos y los métodos de accesos de un sistema de archivos, tenemos una buena imagen de cómo funciona en realidad

- Organización general

Ahora se está desarrollando la organización general en disco de las estructuras de datos del sistema de archivos vsfs. Lo primero que tendremos que hacer es dividir el disco en bloques ya que los sistemas de archivos simples trabajan con tamaños de bloques, normalmente de 4KB

Una vista de la partición del disco que estamos creando en nuestro sistema de archivos es simple. Una serie de bloques, cada uno de 4KB y direccionados desde 0 hasta N-1, siendo N el número de bloques. Tendremos una partición de tamaño 4N KB.

Supondremos que tenemos un pequeño disco de solo 64 bloques. Lo primero que tenemos que ver es dónde meter los datos del usuario que es lo que más espacio necesita en cualquier sistema de archivos. A la región del disco reservada para los datos del usuario se llama región de datos y en nuestro pequeño disco corresponderá a los últimos 56 bloques de los 64

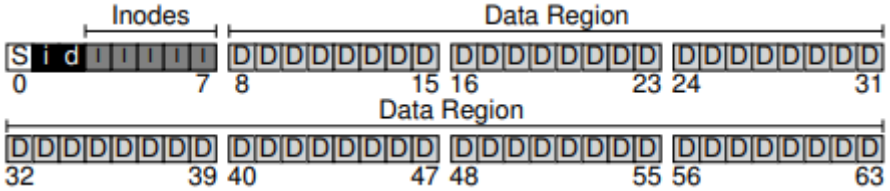
El sistema de archivos tiene que rastrear información sobre cada archivo. Esta información es una pieza clave de los metadatos y rastrea cosas como qué bloques de datos que contienen un archivo, su tamaño, su propietario y permisos de acceso, hora de la última modificación y acceso etc

Para guardar toda esta información el sistema de archivos tiene una estructura llamada inodo que tiene reservada espacio en el disco también. Esa parte del disco es llamada tabla de inodos que simplemente contiene un array de los inodos del disco

Nuestro sistema de archivos por ahora tiene bloques de datos e inodos, pero nos faltan cosas. Un componente primario que necesitamos es una manera de localizar dónde están localizados o libres los inodos y los bloques de datos. Estas estructuras de asignación son un requisito elemental en cualquier sistema de archivos

Podríamos usar una lista de libres que apuntase a el primer bloque libre, o elegir una simple y popular estructura conocida como mapa de bits, uns para la región de datos y otra para los inodos

Tendremos que reservar también un bloque para el conocido como superbloque, que contiene información sobre este sistema de archivos, incluyendo cuando inodos y bloques de datos hay, donde empiezan los inodos, etc. Por tanto, el disco de ejemplo nos quedaría repartido tal que:



- Organización de archivos. El inodo

Una de las estructuras de los discos más importantes en los sistemas de archivos son los inodos. El nombre de inodo viene de nodo indexado. El nombre de inodo es usado en muchos sistemas de archivos para describir la estructura que contiene los metadatos de un archivo dado

Cada inodo es referenciado con un número, i-número, que anteriormente hemos llamado número de bajo nivel en los archivos. Dado un i-número puedes ser capaz directamente de calcular a qué parte del disco corresponde este inodo. Para ello hay que calcular primero el desplazamiento que se obtiene mediante la operación:

$$desp = (i - número * sizeof(inode\_t) \ / \ blockSize$$

Siendo inode_t el tamaño de los inodos. Ahora podremos calcular la dirección del sector:

$$sector = (desp \ * blockSize) + innodeStartAddr \ / \ sectorSize$$

Siendo inodeStartAddr la dirección en la que los inodos comienzan

Size	Name	What is this inode field for?
2	mode	can this file be read/written/executed?
2	uid	who owns this file?
4	size	how many bytes are in this file?
4	time	what time was this file last accessed?
4	ctime	what time was this file created?
4	mtime	what time was this file last modified?
4	dtime	what time was this inode deleted?
2	gid	which group does this file belong to?
2	links_count	how many hard links are there to this file?
4	blocks	how many blocks have been allocated to this file?
4	flags	how should ext2 use this inode?
4	osd1	an OS-dependent field
60	block	a set of disk pointers (15 total)
4	generation	file version (used by NFS)
4	file_acl	a new permissions model beyond mode bits
4	dir_acl	called access control lists

Esto es una tabla que contiene los metadatos, la información del sistema de archivo que no son datos del usuario

Cada nodo contiene punteros directos al bloque de datos del que tienen la información

- Índice multinivel

Para soportar grandes archivos los diseñadores de sistemas han introducido diferentes estructuras. Una idea es tener punteros especiales conocidos como punteros indirectos. En vez de apuntar a un bloque que contiene los datos del usuario, apuntan a un bloque que contiene más punteros, cada uno apuntando a diferentes bloques de datos. Si un archivo crece demasiado se crea un bloque indirecto

No es sorprendente que, en este enfoque, es posible que desee soportar incluso archivos más grandes. Para hacerlo, simplemente se añade otro puntero al inodo: el doble puntero indirecto. Sin embargo, es posible que desee aún más, y esto nos lleva al triple puntero indirecto

En general, este árbol desequilibrado se conoce como el enfoque de índice multinivel para señalar bloques de archivos. Muchos sistemas de archivos usan extensiones de índice multinivel en lugar de punteros simples

- Organización de directorios

Los directorios tienen una organización sencilla, básicamente contienen una lista de pares. Por cada archivo o directorio hay una cadena y un número en los bloques de datos del directorio

Al eliminar un archivo se puede quedar un espacio vacío en mitad del directorio, por tanto, tiene que haber alguna manera de marcarlo. Para esto se guarda la longitud de la cadena

Hay muchas maneras de gestionar el espacio libre, los mapas de bits son una opción. Los primeros sistemas de archivos usaban listas de libres donde un puntero en el superbloque apuntaba al primer bloque libre y éste bloque al siguiente, etc

Los sistemas modernos usan estructuras de datos más sofisticadas. Algunos usan un tipo de árbol-B para representar de forma compacta que partes del disco están libres

- Gestión del espacio libre

Un sistema de archivos tiene que localizar qué inodos y bloques de datos están libres y cuales no para que cuando se quiera crear un nuevo archivo o directorio, saber dónde puede caber. La gestión del espacio libre es importante para todos los sistemas de archivos y en vsfs tenemos dos sencillos mapas de bits para esta tarea

- Vías de acceso: lectura y escritura

Ahora que tenemos una idea de cómo se guardan en disco los archivos y directorios, podremos saber las operaciones que se realizan cuando se quiere leer o escribir en un archivo. Comprender lo que pasa en esta ruta de acceso es la segunda clave para desarrollar una comprensión de cómo un sistema de archivos funciona

Para los siguientes ejemplos vamos a suponer que el sistema de archivos ya ha sido montado y que el superbloque ya está en memoria. El resto, inodos y directorios, están en el disco

Leyendo un archivo desde el disco

Cuando hacemos una llamada con open(), el sistema de archivos lo primero que hace es buscar el inodo correspondiente a el nombre del archivo que se quiere abrir para obtener la información básica de éste. El sistema de archivos debe viajar por el nombre de ruta y, por lo tanto, localizar el inodo deseado

Todos los viajes empiezan en la raíz del sistema de ficheros, el directorio raíz que es llamado /. Primero tendrá que buscar el inodo del directorio raíz y tendremos que saber el i-número de éste. Normalmente encontraremos este i-número en un archivo o directorio en su directorio padre, pero como el directorio raíz no tiene padre, su i-número tendrá que ser “well known”, el sistema tendrá que saberlo de antemano y suele ser 2 en los sistemas UNIX

El siguiente paso es buscar recursivamente a través de la dirección hasta que el inodo deseado se encuentra. Una vez abierto el archivo, el programa puede usar read() para leerlo. Llegará un momento en el que se cierre el archivo y, por tanto, el descriptor del archivo tendrá que ser desasignado

Hay que tener en cuenta que la cantidad de I/O generada al abirr un archivo es proporcional a la longitud de la ruta. Por cada directorio adicional en la ruta tendremos que leer su inodo y sus datos. Una manera de empeorar esto será con la presencia de grandes directorios

- Escribiendo un archivo en el disco

Escribir un archivo es un proceso similar al de leerlo. Primero tendremos que abrirlo y el programa usará write() para actualizar el archivo con nuevos contenidos. Escribir en el archivo también puede asignar un bloque. Al escribir un nuevo archivo, cada escritura no solo tiene que escribir datos en el disco, sino que primero debe decidir qué bloque va a ser asignado para el archivo y actualizar las estructuras del disco para ello

Cada escritura en el archivo genera cinco I/O. Una para leer el mapa de bits de datos, otro para escribir en el mapa de bits, dos más para leer y escribir en el inodo y el último para escribir en el bloque real

La cantidad de tráfico generado al escribir es peor, aunque se considere una operación simple y común, como la ceración de archivos. Para crear un archivo el sistema no solo debe asignar un inodo, sino también un espacio dentro del directorio. La cantidad de trafico de I/O para ello es bastante

- Almacenamiento en caché y almacenamiento en búfer

Leer y escribir en archivos puede ser caro ya que se hacen muchas peticiones I/O que ralentizan el disco. Para arreglar esto muchos sistemas usan memoria de sistema (DRAM) para guardar los bloques importantes

Los primeros sistemas de archivos incluyeron una caché de tamaño variable para guardar bloques importantes. Las estrategias LRU y sus variantes decidían qué bloques se mantenían y cuales se tiraban

Esta partición estática de memoria podía ser un desperdicio ya que las páginas no utilizadas en la caché de archivos no pueden reutilizarse para algún otro uso y, por lo tanto, se desperdician

Los sistemas modernos usan un particionamiento dinámico. Específicamente, muchos S.O. modernos integran páginas de memoria virtual y páginas del sistema de archivos en una caché de página unificada

De este modo, la memoria se puede asignar de manera más flexible a través de la memoria virtual y del sistema de archivos, dependiendo de cuál necesite más memoria en un momento dado

Consideremos el efecto de guardar en la caché las escrituras. Mientras que las lecturas I/O se pueden evitar por completo, el tráfico de escritura tiene que ir al disco para ser persistente. Por tanto, una caché no usa el mismo tipo de filtro en el tráfico de escritura como de lectura

Primero, al retrasar las escrituras, el sistema de archivos puede procesar por lotes algunas actualizaciones en un conjunto más pequeño de I/O. En segundo lugar, almacenando varias escrituras en la memoria intermedia, el sistema puede entonces programar las I/O subsiguientes y así aumentar el rendimiento

Algunas escrituras se evitan por completo retrasándolas. Por ejemplo, si una aplicación crea un archivo y luego lo elimina, se retrasan las escrituras para evitar la creación completa del archivo en el disco