Índice general, Mostrar marcos, Sin marcos

Capítulo 9
The File system

Este cap´itulo describe cómo el kernel de Linux gestiona los ficheros en los sistemas de ficheros soportados por éste. Describe el Sistema de Ficheros Virtual (VFS) y explica cómo los sistemas de ficheros reales del kernel de Linux son soportados.

Una de los rasgos más importantes de Linux es su soporte para diferentes sistemas de ficheros. Ésto lo hace muy flexible y bien capacitado para coexistir con muchos otros sistemas operativos. En el momento de escribir ésto, Linux soporta 15 sistemas de ficheros; `ext`, `ext2`, `xia`, `minix`, `umsdos`, `msdos`, `vfat`, `proc`, `smb`, `ncp`, `iso9660`, `sysv`, `hpfs`, `affs` and `ufs`, y sin duda, con el tiempo se añadirán más.

En Linux, como en Unix, a los distintos sistemas de ficheros que el sistema puede usar no se accede por identificadores de dispositivo (como un número o nombre de unidad) pero, en cambio se combinan en una simple structura jerárquica de árbol que representa el sistema de ficheros como una entidad única y sencilla. Linux añade cada sistema de ficheros nuevo en este simple árbol de sistemas de ficheros cuando se monta. Todos los sistemas de ficheros, de cualquier tipo, se montan sobre un directorio y los ficheros del sistema de ficheros son el contenido de ese directorio. Este directorio se conoce como directorio de montaje o punto de montaje. Cuando el sistema de ficheros se desmonta, los ficheros propios del directorio de montaje son visibles de nuevo.

Cuando se inicializan los discos (usando fdisk, por ejemplo) tienen una estructura de partición inpuesta que divide el disco físico en un número de particiones lógicas. Cada partición puede mantener un sistema de ficheros, por ejemplo un sistema de ficheros `EXT2`. Los sistemas de ficheros organizan los ficheros en structuras jerárquicas lógicas con directorios, enlaces flexibles y más contenidos en los bloques de los dispositivos físicos. Los dispositivos que pueden contener sistemas de ficheros se conocen con el nombre de dispositivos de bloque. La partición de disco IDE `/dev/hda1`, la primera partición de la primera unidad de disco en el sistema, es un dispositivo de bloque. Los sistemas de ficheros de Linux contemplan estos dispositivos de bloque como simples colecciones lineales de bloques, ellos no saben o tienen en cuenta la geometría del disco físico que hay debajo. Es la tarea de cada controlador de dispositivo de bloque asignar una petición de leer un bloque particular de su dispositivo en términos comprensibles para su dispositivo; la pista en cuestión, sector y cilindro de su disco duro donde se guarda el bloque. Un sistema de ficheros tiene que mirar, sentir y operar de la misma forma sin importarle con que dispositivo está tratando. Por otra parte, al usar los sistemas de ficheros de Linux, no importa (al menos para el usuario del sistema) que estos distintos sistemas de ficheros estén en diferentes soportes controlados por diferentes controladores de hardware. El sistema de ficheros puede incluso no estar en el sistema local, puede ser perfectamente un disco remoto montado sobre un enlace de red. Considerese el siguiente ejemplo donde un sistema Linux tiene su sistema de ficheros raíz en un disco SCSI:

```
A          E          boot       etc        lib        opt        tmp        usr
C          F          cdrom      fd         proc       root       var        sbin
D          bin        dev        home       mnt        lost+found
```

Ni los usuarios ni los programas que operan con los ficheros necesitan saber que `/C` de hecho es un sistema de ficheros VFAT montado que está en el primer disco IDE del sistema. En el ejemplo (que es mi sistema Linux en casa), `/E` es el disco IDE primario en la segunda controladora IDE. No importa que la primera controladora IDE sea una controladora PCI y que la segunda sea una controladora ISA que también controla el IDE CDROM. Puedo conectarme a la red donde trabajo usando un modem y el protocolo de red PPP y en este caso puedo remotamente montar mis sistemas de ficheros Linux Alpha AXP\ sobre `/mnt/remote`.

Los ficheros en un sistema de ficheros son grupos de datos; el fichero que contiene las fuentes de este capítulo

es un fichero ASCII llamado `filesystems.tex`. Un sistema de ficheros no sólo posee los datos contenidos dentro de los ficheros del sistema de ficheros, además mantiene la estructura del sistema de ficheros. Mantiene toda la información que los usuarios de Linux y procesos ven como ficheros, directorios, enlaces flexibles, información de protección de ficheros y así. Por otro lado debe mantener esa información de forma eficiente y segura, la integridad básica del sistema operativo depende de su sistema de ficheros. Nadie usaria un sistema operativo que perdiera datos y ficheros de forma aleatoria[1].

`Minix`, el primer sistema de ficheros que Linux tuvo es bastante restrictivo y no era muy rápido. `Minix`, the first file system that Linux had is rather restrictive and lacking in performance. Sus nombres de ficheros no pueden tener más de 14 caracteres (que es mejor que nombres de ficheros 8.3) y el tamaño máximo de ficheros es 64 MBytes. 64 MBytes puede a primera vista ser suficiente pero se necesitan tamaños de ficheros más grandes para soportar incluso modestas bases de datos. El primer sistema de ficheros diseñado especificamente para Linux, el sistema de Ficheros Extendido, o `EXT`, fue introducido en Abril de 1992 y solventó muchos problemas pero era aun falto de rapidez. Así, en 1993, el Segundo sistema de Ficheros Extendido, o `EXT2`, fue añadido. Este es el sistema de ficheros que se describe en detalle más tarde en este capítulo.

Un importante desarrollo tuvo lugar cuando se añadió en sistema de ficheros EXT en Linux. El sistema de ficheros real se separó del sistema operativo y servicios del sistema a favor de un interfaz conocido como el sistema de Ficheros Virtual, o VFS. VFS permite a Linux soportar muchos, incluso muy diferentes, sistemas de ficheros, cada uno presentando un interfaz software común al VFS. Todos los detalles del sistema de ficheros de Linux son traducidos mediante software de forma que todo el sistema de ficheros parece idéntico al resto del kernel de Linux y a los programas que se ejecutan en el sistema. La capa del sistema de Ficheros Virtual de Linux permite al usuario montar de forma transparente diferentes sistemas de ficheros al mismo tiempo.

El sistema de Ficheros Virtual está implementado de forma que el acceso a los ficheros es rápida y tan eficiente como es posible. También debe asegurar que los ficheros y los datos que contiene son correctos. Estos dos requisitos pueden ser incompatibles uno con el otro. El VFS de Linux mantiene una antememoria con información de cada sistema de ficheros montado y en uso. Se debe tener mucho cuidado al actualizar correctamente el sistema de ficheros ya que los datos contenidos en las antememorias se modifican cuando cuando se crean, escriben y borran ficheros y directorios. Si se pudieran ver las estructuras de datos del sistema de ficheros dentro del kernel en ejecución, se podria ver los bloques de datos que se leen y escriben por el sistema de ficheros. Las estructuras de datos, que describen los ficheros y directorios que son accedidos serian creadas y destruidas y todo el tiempo los controladores de los dispositivo estarian trabajando, buascando y guardando datos. La antememoria o caché más importantes es el Buffer Cache, que está integrado entre cada sistema de ficheros y su dispositivo de bloque. Tal y como se accede a los bloques se ponen en el Buffer Cache y se almacenan en varias colas dependiendo de sus estados. El Buffer Cache no sólo mantiene buffers de datos, tambien ayuda a administrar el interfaz asíncrono con los controladores de dispositivos de bloque.

9.1 The Second Extended File system (EXT2)

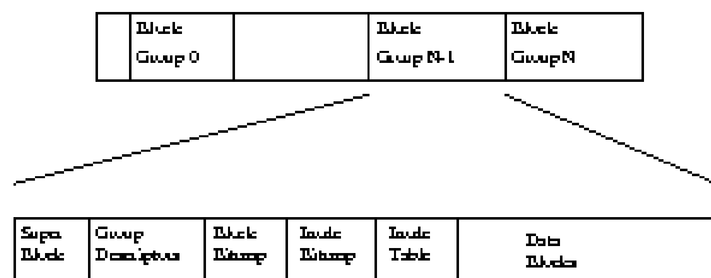


Figure 9.1: Physical Layout of the EXT2 File system

El Segundo sistema de ficheros Extendido fue pensado (por Rémy Card) como un sistema de ficheros

extensible y poderoso para Linux. También es el sistema de ficheros más éxito tiene en la comunidad Linux y es básico para todas las distribuciones actuales de Linux. El sistema de ficheros EXT2, como muchos sistemas de ficheros, se construye con la premisa de que los datos contenidos en los ficheros se guarden en bloques de datos. Estos bloques de datos son todos de la misma longitud y, si bien esa longitud puede variar entre diferentes sistemas de ficheros EXT2 el tamaño de los bloques de un sistema de ficheros EXT2 en particular se decide cuando se crea (usando `mke2fs`). El tamaño de cada fichero se redondea hasta un numero entero de bloques. Si el tamaño de bloque es 1024 bytes, entonces un fichero de 1025 bytes ocupará dos bloques de 1024 bytes. Desafortunadamente esto significa que por media se desperdicia un bloque por fichero.
==
Normalmente en ordenadores se Usually in computing you trade off CPU usage for memory and disk space utilisation. In this case Linux, along with most operating systems, trades off a relatively inefficient disk usage in order to reduce the workload on the CPU.
== No todos los bloques del sistema de ficheros contienen datos, algunos deben usarse para mantener la información que describe la estructura del sistema de ficheros. EXT2 define la topologia del sistema de ficheros describiendo cada fichero del sistema con una estructura de datos inodo. Un inodo describe que bloques ocupan los datos de un fichero y también los permisos de acceso del fichero, las horas de modificación del fichero y el tipo del fichero. Cada fichero en el sistema de ficheros EXT2 se describe por un único inodo y cada inodo tiene un único número que lo identifica. Los inodos del sistema de ficheros se almacenan juntos en tablas de inodos. Los directorios EXT2 son simplemente ficheros especiales (ellos mismos descritos por inodos) que contienen punteros a los inodos de sus entradas de directorio.

La figura 9.1 muestra la disposición del sistema de ficheros EXT2 ocupando una serie de bloques en un dispositivo estructurado bloque. Por la parte que le toca a cada sistema de ficheros, los dispositivos de bloque son sólo una serie de bloques que se pueden leer y escribir. Un sistema de ficheros no se debe preocupar donde se debe poner un bloque en el medio físico, eso es trabajo del controlador del dispositivo. Siempre que un sistema de ficheros necesita leer información o datos del dispositivo de bloque que los contiene, pide que su controlador de dispositivo lea un número entero de bloques. El sistema de ficheros EXT2 divide las particiones lógicas que ocupa en Grupos de Bloque (Block Groups). Cada grupo duplica información crítica para la integridad del sistema de ficheros ya sea valiendose de ficheros y directorios como de bloques de información y datos. Esta duplicación es necesaria por si ocurriera un desastre y el sistema de ficheros necesitara recuperarse. Los subapartados describen con más detalle los contenidos de cada Grupo de Bloque.

9.1.1 The EXT2 Inode

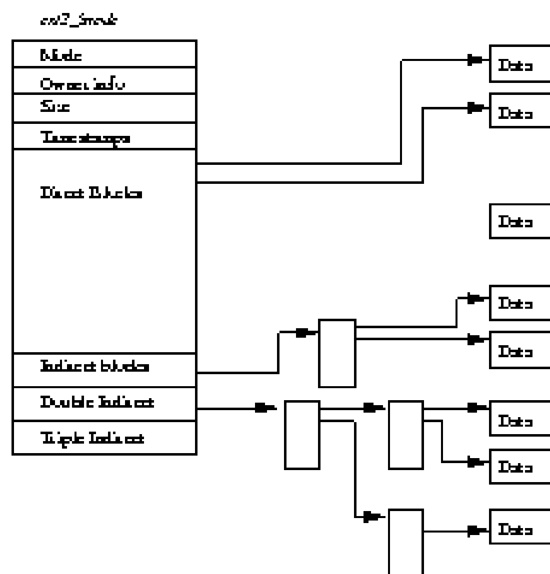


Figure 9.2: EXT2 Inode

En el sistema de ficheros EXT2, el inodo es el bloque de construcción básico; cada fichero y directorio del sistema de ficheros es descrito por un y sólo un inodo. Los inodos EXT2 para cada Grupo de Bloque se almacenan juntos en la table de inodos con un mapa de bits que permite al sistema seguir la pista de inodos reservados y libres. La figura 9.2 muestra el formato de un inodo EXT2, entre otra información, contiene los siguientes campos:

mode
: Esto mantiene dos partes de información; qué inodo describe y los permisos que tienen los usuarios. Para EXT2, un inodo puede describir un ficheros, directorio, enlace simbólico, dispositivo de bloque, dispositivo de caracter o FIFO.

Owner Information
: Los identificadores de usuario y grupo de los dueños de este fichero o directorio. Esto permite al sistema de ficheros aplicar correctamente el tipo de acceso,

Size
: El tamaño en del fichero en bytes,

Timestamps
: La hora en la que el inodo fue creado y la última hora en que se modificó,

Datablocks
: Punteros a los bloques que contienen los datos que este inodo describe. Los doce primeros son punteros a los bloques físicos que contienen los datos descritos por este inodo y los tres últimos punteros contienen más y más niveles de indirección. Por ejemplo, el puntero de doble indirección apunta a un bloque de punteros que apuntan a bloques de punteros que apuntan a bloques de datos. Esto significa que ficheros menores o iguales a doce bloques de datos en longitud son más facilmente accedidos que ficheros más grandes.

Indicar que los inodos EXT2 pueden describir ficheros de dispositivo especiales. No son ficheros reales pero permiten que los programas puedan usarlos para acceder a los dispositivos. Todos los ficheros de dispositivo de `/dev` están ahi para permitir a los programas acceder a los dispositivos de Linux. Por ejemplo el programa mount toma como argumento el fichero de dispositivo que el usuario desee montar.

9.1.2 The EXT2 Superblock

El Superbloque contiene una descripción del tamaño y forma base del sistema de ficheros. La información contenida permite al administrador del sistema de ficheros usar y mantener el sistema de ficheros. Normalmente sólo se lee el Superbloque del Grupo de Bloque 0 cuando se monta el sistema de ficheros pero cada Grupo de Bloque contiene una copia duplicada en caso de que se corrompa sistema de ficheros. Entre otra información contiene el:

Magic Number
: Esto permite al software de montaje comprobar que es realmente el Superbloque para un sistema de ficheros EXT2. Para la versión actual de EXT2 éste es 0xEF53.

Revision Level
: Los niveles de revisión mayor y menor permiten al código de montaje determinar si este sistema de ficheros soporta o no características que sólo son disponibles para revisiones particulares del sistema de ficheros. También hay campos de compatibilidad que ayudan al código de montaje determinar que nuevas características se pueden usar con seguridad en ese sistema de ficheros,

Mount Count and Maximum Mount Count
: Juntos permiten al sistema determinar si el sistema de ficheros fue comprobado correctamente. El contador de montaje se incrementa cada vez que se monta el sistema de ficheros y cuando es igual al contador máximo de montaje muestra el mensaje de aviso «maximal mount count reached, running e2fsck is recommended»,

Block Group Number
: El número del Grupo de Bloque que tiene la copia de este Superbloque,

Block Size
: El tamaño de bloque para este sistema deficheros en bytes, por ejemplo 1024 bytes,

Blocks per Group
: El número de bloques en un grupo. Como el tamaño de bloque éste se fija cuando se crea el sitema de ficheros,

Free Blocks
: EL número de bloques libres en el sistema de ficheros,

Free Inodes
: El número de Inodos libres en el sistema de ficheros,

First Inode
: Este es el número de inodo del primer inodo en el sistema de ficheros. El primer inodo en un sistema de ficheros EXT2 raíz seria la entrada directorio para el directorio '/'.

9.1.3 The EXT2 Group Descriptor

Cada Grupo de Bloque tiene una estructura de datos que lo describe. Como el Superbloque, todos los descriptores de grupo para todos los Grupos de Bloque se duplican en cada Grupo de Bloque en caso de corrupción del sistema de fichero. Cada Descriptor de Grupo contiene la siguiente información:

Blocks Bitmap
: El número de bloque del mapa de bits de bloques reservados para este Grupo de Bloque. Se usa durante la reseva y liberación de bloques,

Inode Bitmap
: El número de bloque del mapa de bits de inodos reservados para este Grupo de Bloques. Se usa durante la reserva y liberación de inodos,

Inode Table
: El número de bloque del bloque inicial para la tabla de inodos de este Grupo de Bloque. Cada inodo se representa por la estructura de datos inodo EXT2 descrita abajo.

Free blocks count, Free Inodes count, Used directory count

Los descriptores de grupo se colocan uno detrás de otro y juntos hacen la tabla de descriptor de grupo. Cada Grupo de Bloques contiene la tabla entera de descriptores de grupo despues de su copia del Superbloque. Sólo la primera copia (en Grupo de Bloque 0) es usada por el sistema de ficheros EXT2. Las otras copias están ahi, como las copias del Superbloque, en caso de que se corrompa la principal.

9.1.4 EXT2 Directories

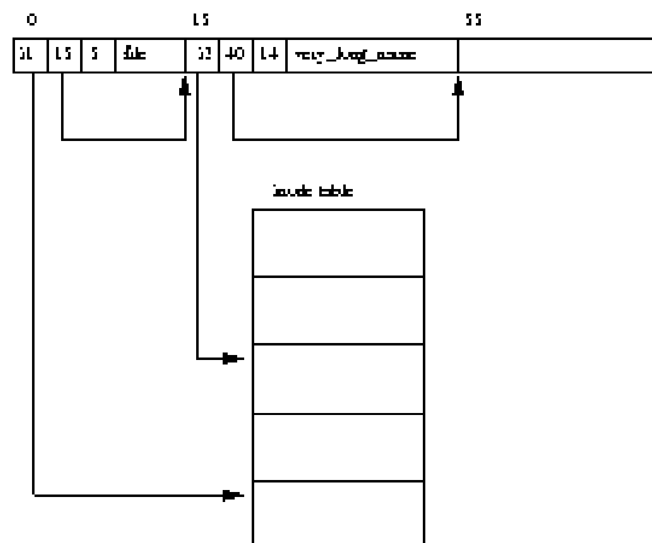


Figure 9.3: EXT2 Directory

En el sistema de ficheros EXT2, los directorios son ficheros especiales que se usan para crear y mantener rutas de acceso a los ficheros en el sistema de ficheros. La figura 9.3 muestra la estructura de una estrada directorio en memoria. Un fichero directorio es una lista de entradas directorio, cada una conteniendo la siguiente información:

inode
: El inodo para esta entrada directorio. Es un índice al vector de inodos guardada en la Tabla de Inodos del Grupo de Bloque. En la figura 9.3, la entrada directorio para el fichero llamado `file` tiene una referencia al número de inodo `i1`,

name length
: La longitud de esta entrada directorio en bytes,

name
: El nombre de esta entrada directorio.

Las dos primeras entradas para cada directorio son siempre las entradas estandar «.» y «..» significando «este directorio» y «el directorio padre» respectivamente.

9.1.5 Finding a File in an EXT2 File System

Un nombre de fichero Linux tiene el mismo formato que los nombres de ficheros de todos los Unix\. Es una serie de nombres de directorios separados por contra barras (« `/`») y acabando con el nombre del fichero. Un ejemplo de nombre de fichero podria ser `/home/rusling/.cshrc` donde `/home` y `/rusling` son nombres de directorio y el nombre del fichero es `.cshrc`. Como todos los demas sistemas Unix¸ Linux no tiene encuenta el formato del nombre del fichero; puede ser de cualquier longitud y cualquier caracter imprimible. Para encontrar el inodo que representa a este fichero dentro de un sistema de ficheros `EXT2` el sistema debe analizar el nombre del fichero directorio a directorio hasta encontrar el fichero en si. El primer inodo que se necesita es el inodo de la raíz del sistema de ficheros, que está en el superbloque del sistema de ficheros. Para leer un inodo EXT2 hay que buscarlo en la tabla de inodos del Grupo de Bloque apropiado. Si, por ejemplo, el número de inodo de la raíz es 42, entonces necesita el inodo 42avo de la tabla de inodos del Grupo de Bloque 0. El inodo raíz es para un directorio EXT2, en otras palabras el modo del inodo lo describe como un directorio y sus bloques de datos contienen entradas directorio EXT2.

`home` es una de las muchas entradas directorio y esta entrada directorio indica el número del inodo que describe al directorio `/home`. Hay que leer este directorio (primero leyendo su inodo y luego las entradas directorio de los bloques de datos descritos por su inodo) para encontrar la entrada `rusling` que indica el numero del inodo que describe al directorio `/home/rusling`. Finalmente se debe leer las entradas directorio apuntadas por el inodo que describe al directorio `/home/rusling` para encontrar el número de inodo del fichero `.cshrc` y desde ahi leer los bloques de datos que contienen la información del fichero.

9.1.6 Changing the Size of a File in an EXT2 File System

Un problema común de un sistema de ficheros es la tendencia a fragmentarse. Los bloques que contienen los datos del fichero se esparcen por todo el sistema de ficheros y esto hace que los accesos secuenciales a los bloques de datos de un fichero sean cada vez más ineficientes cuanto más alejados estén los bloques de datos. El sistema de ficheros EXT2 intenta solucionar esto reservando los nuevos bloques para un fichero, fisicamente juntos a sus bloques de datos actuales o al menos en el mismo Grupo de Bloque que sus bloques de datos. Sólo cuando esto falla, reserva bloques de datos en otros Grupos de Bloque.

Siempre que un proceso intenta escribir datos a un fichero, el sistema de ficheros Linux comprueba si los datos exceden el final del último bloque para el fichero. Si lo hace, entonces tiene que reservar un nuevo bloque de datos para el fichero. Hasta que la reserva no haya acabado, el proceso no puede ejecutarse; debe esperarse a que el sistema de ficheros reserve el nuevo bloque de datos y escriba el resto de los datos antes de continuar. La primera cosa que hacen las rutinas de reserva de bloques EXT2 es bloquear el Superbloque EXT2 de ese sistema de ficheros. La reserva y liberación cambia campos del superbloque, y el sistema de ficheros Linux no puede permitir más de un proceso haciendo ésto a la vez. Si otro proceso necesita reservar más bloques de

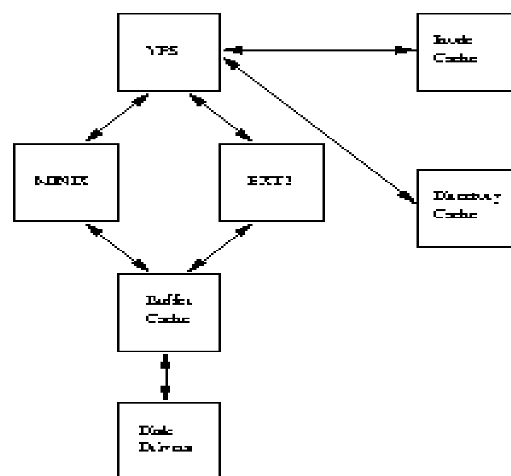
datos, debe esperarse hasta que el otro proceso acabe. Los procesos que esperan el superbloque son suspendidos, no se pueden ejecutar, hasta que el control del superbloque lo abandone su usuario actual. El acceso al superbloque se garantiza al primero que llega, ** Access to the superblock is granted on a first come, first served basis and once a process has control of the superblock, it keeps control until it has finished. ** Teniendo bloqueado el superbloque, el proceso comprueba que hay suficientes bloques libres en ese sistema de ficheros. Si no es así, el intento de reservar más bloques falla y el proceso cederá el control del superbloque del sistema de ficheros.

Si hay suficientes bloques en el sistema de ficheros, el proceso intenta reservar uno. Si el sistema de ficheros EXT2 se ha compilado para prereservar bloques de datos entonces se podrá usar uno de estos. La prereserva de bloques no existe realmente, sólo se reservan dentro del mapa de bits de bloques reservados. El inodo VFS que representa el fichero que intenta reservar un nuevo bloque de datos tiene dos campos EXT2 específicos, `prealloc_block` y `prealloc_count`, que son el numero de bloque del primer bloque de datos prereservado y cuantos hay, respectivamente. Si no habian bloques prereservados o la reserva anticipada no está activa, el sistema de ficheros EXT2 debe reservar un nuevo bloque. El sistema de ficheros EXT2 primero mira si el bloque de datos despues del último bloque de datos del fichero está libre. Logicamente, este es el bloque más eficiente para reservar ya que hace el acceso secuencial mucho más rápido. Si este bloque no está libre, la búsqueda se ensancha y busca un bloque de datos dentro de los 64 bloques del bloque ideal. Este bloque, aunque no sea ideal está al menos muy cerca y dentro del mismo Grupo de Bloque que los otros bloques de datos que pertenecen a ese fichero.

Si incluso ese bloque no está libre, el proceso empieza a buscar en los demás Grupos de Bloque hasta encontrar algunos bloques libres. El código de reserva de bloque busca un cluster de ocho bloques de datos libres en cualquiera de los Grupos de Bloque. Si no puede encontrar ocho juntos, se ajustará para menos. Si se quiere la prereserva de bloques y está activado, actualizará `prealloc_block` y `prealloc_count` pertinentemente.

Donde quiera que encuentre el bloque libre, el código de reserva de bloque actualiza el mapa de bits de bloque del Grupo de Bloque y reserva un buffer de datos en el buffer caché. Ese buffer de datos se identifica unequivocamente por el identificador de dispositivo del sistema y el número de bloque del bloque reservado. El buffer de datos se sobreescribe con ceros y se marca como «sucio» para indicar que su contenido no se ha escrito al disco físico. Finalmente, el superbloque se marca como «sucio» para indicar que se ha cambiado y está desbloqueado. Si hubiera otros procesos esperando, al primero de la cola se le permitiria continuar la ejecución y terner el control exclusido del superbloque para sus operaciones de fichero. Los datos del proceso se escriben en el nuevo bloque de datos y, si ese bloque se llena, se repite el proceso entero y se reserva otro bloque de datos.

9.2 The Virtual File System (VFS)



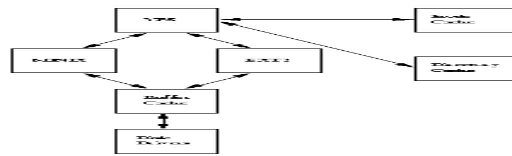


Figure 9.4: A Logical Diagram of the Virtual File System

La figura 9.4 muestra la relación entre el Sistema de Ficheros Virtual del kernel de Linux y su sistema de ficheros real. El sistema de ficheros vitual debe mantener todos los diferentes sistemas de ficheros que hay montados en cualquier momento. Para hacer esto mantiene unas estructuras de datos que describen el sistema de ficheros (virtual) por entero y el sistema de ficheros, montado, real. De forma más confusa, el VFS describe los ficheros del sistema en términos de superbloque e inodos de la misma forma que los ficheros EXT2 usan superbloques e inodos. Como los inodos EXT2, los inodos VFS describen ficheros y directorios dentro del sistema; los contenidos y topología del Sistema de Ficheros Virtual. De ahora en adelante, para evitar confusiones, se escribirá inodos CFS y superbloques VFS para distinguirlos de los inodos y superbloques EXT2.

Cuando un sistema de ficheros se inicializa, se registra él mismo con el VFS. Esto ocurre cuando el sistema operativo se inicializa en el momento de arranque del sistema. Los sistemas de ficheros reales están compilados con el nucleo o como módulos cargables. Los módulos de Sistemas de Ficheros se cargan cuando el sistema los necesita, así, por ejemplo, si el sistema de ficheros `VFAT` está implementado como módulo del kernel, entonces sólo se carga cuando se monta un sistema de ficheros `VFAT`. Cuando un dispositivo de bloque base se monta, y éste incluye el sistema de ficheros raíz, el VFS debe leer su superbloque. Cada rutina de lectura de superbloque de cada tipo de sistema de ficheros debe resolver la topología del sistema de ficheros y mapear esa información dentro de la estructura de datos del superbloque VFS. El VFA mantiene una lista de los sitema de ficheros montados del sistema junto con sus superbloques VFS. Cada superbloque VFS contiene información y punteros a rutinas que realizan funciones particulares. De esta forma, por ejemplo, el superbloque que representa un sistema de ficheros EXT2 montado contiene un puntero a la rutina de lectura de inodos específica. Esta rutina, como todas las rutinas de lectura de inodos del sistema de ficheros espeífico, rellena los campos de un inodo VFS. Cada superbloque VFS contiene un puntero al primer inodo VFS del sistema de ficheros. Para el sistema de ficheros raíz, éste es el inodo que representa el directorio «`/`». Este mapeo de información es muy eficiente para el sistema de ficheros EXT2 pero moderadamente menos para otros sistema de ficheros.

Ya que los procesos del sistema acceden a directorios y ficheros, las rutinas del sistema se dice que recorren los inodos VFS del sistema. Por ejemplo, escribir ls en un directorio o cat para un fichero hacen que el Sistema de Ficheros Virtual busque através de los inodos VFS que representan el sistema de ficheros. Como cada fichero y directorio del sistema se representa por un inodo VFS, un número de inodos serán accedidos repetidamente. Estos inodos se mantienen en la antememoria, o caché, de inodos que hace el acceso mucho más rápido. Si un inodo no está en la caché, entonces se llama a una rutina específica del sistema de ficheros para leer el inodo apropiado. La acción de leer el inodo hace que se ponga en la caché de inodos y siguientes accesos hacen que se mantenga en la caché. Los inodos VFS menos usados se quitan de la caché.

Todos los sistemas de ficheros de Linux usan un buffer caché común para mantener datos de los dispositivos para ayudar a acelerar el acceso por todos los sistemas de ficheros al dispositivo físico que contiene los sistemas de ficheros. Este buffer caché es independiente del sistema de ficheros y se integra dentro de los mecanismos que el núcleo de Linux usa para reservar, leer y escribir datos. Esto tiene la ventaja de hacer los sistemas de ficheros de Linux independientes del medio y de los controladores de dispositivos que los soportan. Tofos los dispositivos estructurados de bloque se registran ellos mismos con el núcleo de Linux y presentan una interfaz uniforme, basada en bloque y normalmente asíncrona. Incluso dispositivos de bloque relativamente complejos como SCSI lo hacen.

Cuando el sistema de ficheros real lee datos del disco físico realiza una petición al controlador de dispositivo de bloque para leer los bloques físicos del dispositivo que controla. Integrado en este interfaz de dispositivo de bloque está el buffer caché. Al leer bloques del sistema de ficheros se guardan en un el buffer caché global compartido por todos los sistemas de ficheros y el núcleo de Linux. Los buffers que hay dentro se identifican por su número de bloque y un identificador único para el dispositivo que lo ha leido. De este modo, si se

necesitan muy a menudo los mismos datos, se obtendrán del buffer caché en lugar de leerlos del disco, que tarda más tiempo. ** Some devices support read ahead where data blocks are speculatively read just in case they are needed. **

El VFS también mantiene una caché de directorios donde se pueden encontrar los inodos de los directorios que se usan de forma más frecuente. Como experimento, probar a listar un directorio al que no se haya accedido recientemente. La primera vez que se lista, se puede notar un pequeño retardo pero la segunda vez el resultado es inmediato. El caché directorio no almacena realmente los inodos de los directorios; éstos estarán en el caché de inodos, el directorio caché simplemente almacena el mapa entre el nombre entero del directorio y sus números de inodo.

9.2.1 The VFS Superblock

Cada sistema de ficheros montado está representado por un superbloque VFS; entre otra información, el superbloque VFS contiene:

Device
: Es el identificador de dispositivo para el dispositivo bloque que contiene este a este sistema de ficheros. Por ejemplo, `/dev/hda1`, el primer disco duro IDE del sistema tiene el identificador de dispositivo 0x301,

Inode pointers
: El puntero de inodo `montado` apunta al primer inodo del sistema de ficheros. El puntero de inodo `cubierto` apunta al inodo que representa el directorio donde está montado el sistema de ficheros. El superbloque VFS del sistema de ficheros raíz no tiene puntero `cubierto`,

Blocksize
: EL tamaño de bloque en bytes del sistema de ficheros, por ejemplo 1024 bytes,

Superblock operations
: Un puntero a un conjunto de rutinas de superbloque para ese sistema de ficheros. Entre otras cosas, estas rutinas las usa el VFS para leer y escribir inodos y superbloques.

File System type
: Un puntero a la estructura de datos `tipo_sistema_` del sistema de ficheros montado,

File System specific
: Un puntero a la información que necesaria este sistema de ficheros.

9.2.2 The VFS Inode

Como el sistema de ficheros EXT2, cada fichero, directorio y demás en el VFS se representa por uno y solo un inodos VFS. La infomación en cada inodo VFS se construye a partir de información del sistema de ficheros por las rutinas específicas del sistema de ficheros. Los inodos VFS existen sólo en la memoria del núcleo y se mantienen en el caché de inodos VFS tanto tiempo como sean útiles para el sistema. Entre otra información, los inodos VFS contienen los siguientes campos:

device
: Este es el identificador de dispositivo del dispositivo que contiene el fichero o lo que este inodo VFS represente,

inode number
: Este es el número del inodo y es único en este sistema de ficheros. La combinación de `device` y `inode number` es única dentro del Sistema de Ficheros Virtual,

mode
: Como en EXT2 este campo describe que representa este inodo VFS y sus permisos de acceso,

user ids
: Los identificadores de propietario,

times

Los tiempos de creación, modificación y escritura,

block size

El tamaño de bloque en bytes para este fichero, por ejemplo 1024 bytes,

inode operations

Un puntero a un bloque de direcciones de rutina. Estas rutinas son espefíficas del sistema de ficheros y realizan operaciones para este inodo, por ejemplo, truncar el fichero que representa este inodo.

count

El número de componentes del sistema que están usando actualmente este inodo VFS. Un contador de cero indica que el inodo está libre para ser descartado o reusado,

lock

Este campo se usa para bloquear el inodo VFS, por ejemplo, cuando se lee del sistema de ficheros,

dirty

Indica si se ha escrito en este inodo, si es as´i el sistema de ficheros necesitará modificarlo,

file system specific information

9.2.3 Registering the File Systems

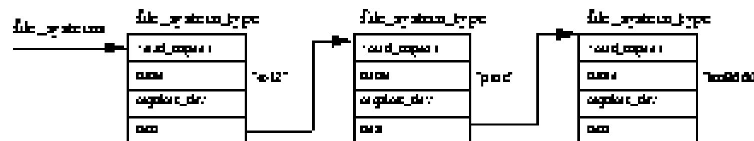


Figure 9.5: Registered File Systems

Cuando se compila el núcleo de Linux se pregunta si se quiere cada uno de los sistemas de ficheros soportados. Cuando el núcleo está compilado, el códifo de arranque del sistema de ficheros contiene llamadas a las rutinas de inicialización de todos los sistemas de ficheros compilados. Los sistemas de ficheros de Linux también se pueden compilar como módulos y, en este caso, pueden ser cargados cuando se les necesita o cargarlos a mano usando insmod. Siempre que un módulo de sistema de ficheros se carga se registra él mismo con el núcleo y se borra él mismo cuando se descarga. Cada rutina de inicialización del sistema de ficheros se registra con el Sistema de Ficheros Virtual y se representa por una estructura de datos `tipo_sistema_` que contiene el nombre del sistema de ficheros y un puntero a su rutina de lectura de superbloque VFS. La figura 9.5 muestra que las estructuras de datos `tipo_sistems_ficheros` se ponen en una lista apuntada por el puntero `sistemas_ficheros`. Cada estructura de datos `tipo_sistema_ficheros` contiene la siguiente información:

Superblock read routine

Esta rutina se llama por el VFS cuando se monta una instancia del sistema de ficheros,

File System name

El nombre de este sistema de ficheros, por ejemplo `ext2`,

Device needed

Necesita soportar este sistema de ficheros un dispositivo? No todos los sistemas de ficheros necesitan un dispositivo. El sistema de fichero `/proc`, por ejemplo, no requiere un dispositivo de bloque,

Se puede ver que sistemas de ficheros hay rgistrados mirando en `/proc/filesystems`. Por ejemplo:

```
      ext2
nodev proc
      iso9660
```

9.2.4 Mounting a File System

Cuando el superusuario intenta montar un sistema de ficheros, el núcleo de Linux debe primero validar los argumentos pasados en la llamada al sistema. Aunque mount hace una comprobación básica, no conoce que

sistemas de ficheros soporta el kernel o si existe el punto de montaje propuesto. Considerar el siguiente comando mount:

```
$ mount -t iso9660 -o ro /dev/cdrom /mnt/cdrom
```

Este comando mount pasa al núcleo tres trozos de información; el nombre del sistema de ficheros, el dispositivo de bloque físico que contiene al sistema de fichros y, por último, donde, en la topología del sistema de ficheros existente, se montará el nuevo sistema de ficheros.

La primera cosa que debe hacer el Sistema de Ficheros Virtual es encontrar el sistema de ficheros. Para hacer ´esto busca a través de la lista de sistemas de ficheros conocidos y mira cada estructura de datos dstipo_sistema_ficheros en la lista apuntada por `sistema_ficheros`. Si encuentra una coincidencia del nombre ahora sabe que ese tipo de sistema de ficheros es soportado por el núcleo y tiene la dirección de la rutina específica del sistema de ficheros para leer el superbloque de ese sistema de ficheros. Si no puede encontrar ninguna coindidencia no todo está perdido si el núcleo puede cargar módulos por demanda (ver Capítulo modules-chapter). En este caso el núcleo piede al demonio del núcleo que cargue el módulo del sistema de ficheros apropiado antes de continuar como anteriormente.

Si el dispositivo físico pasado por mount no está ya montado, debe encontrar el inodo VFS del directorio que será el punto de montaje del nuevo sistema de ficheros. Este inodo VFS debe estar en el caché de inodos o se debe leer del dispositivo de bloque que soporta el sistema de ficheros del punto de montaje. Una vez que el inodo se ha encontrado se comprueba para ver que sea un directorio y que no contenga ya otro sistema de ficheros montado. El mismo directorio no se puede usar como punto de montaje para más de un sistema de ficheros.

En este punto el códifo de montaje VFS reserva un superbloque VFS y le pasa la información de montahe a la rutina de lectura de superblque para este sistema de ficheros. Todos los superbloques VFS del sistema se mantienen en el vector `super_bloques` de las estructuras de datos `super_bloque` y se debe reservar una para este montaje. La rutina de lectura de superbloque debe rellenar los campos básicos del superbloque VFS con información que lee del dispositivo físico. Para el sistema de ficheros EXT2 este mapeo o traducción de información es bastante facil, simplemente lee el superbloque EXT2 y rellena el superbloque VFS de ahí. Para otros sistemas de ficheros, como el MS DOS, no es una tarea tán facil. Cualquiera que sea el sistema de ficheros, rellenar el superbloque VFS significa que el sistema de ficheros debe leer todo lo que lo describe del dispositivo de bloque que lo soporta. Si el dispositivo de bloque no se puede leer o si no contiene este tipo de sistema de ficheros entonces el comando mount fallará.

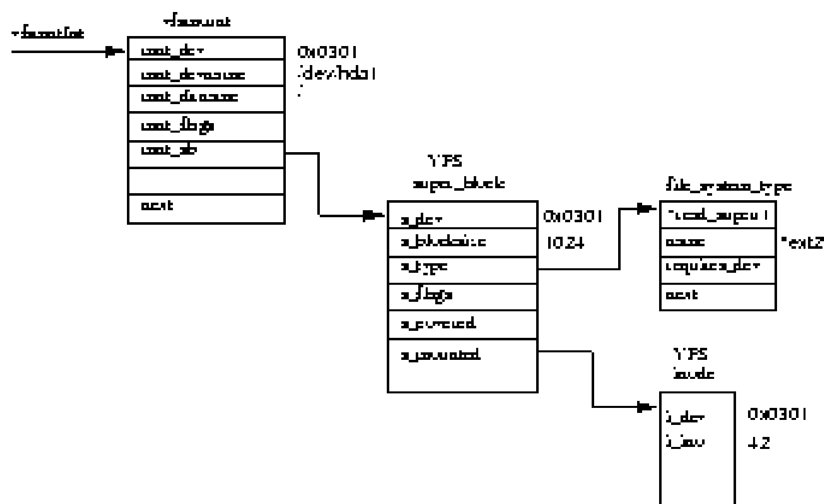


Figure 9.6: A Mounted File System

Cada sistema de ficheros montado es descrito por una estructura de datos `vfsmount`; ver figura 9.6. Estos son

puestos en una cola de una lista apuntada por `vfsmntlist`. Otro puntero, `vfsmnttail` apunta a la última entrada de la lista y el puntero `mru_vfsmnt` apunta al sistemas de ficheros más recientemente usado. Cada estructura `vfsmount` contiene el número de dispositivo del dispositivo de bloque que contiene al sistema de ficheros, el directorio donde el sistema de ficheros está montado y un puntero al superbloque VFS reservado cuando se montó. En cambio el superbloque VFS apunta a la estructura de datos `tipo_sistema_ficheros` para este tipo de sisetma de ficheros y al inodo ra´iz del sistema de ficheros. Este inodo se mantiene residente en el caché de inodos VFS todo el tiempo que el sistema de ficheros esté cargado.

9.2.5 Finding a File in the Virtual File System

Para encontrar el inodo VFS de un fichero en el Sistema de Ficheros Virtual, VFS debe resolver el nombre del directorio, mirando el inodo VFS que representa cada uno de los directorios intermedios del nombre. Mirar cada directorio envuelve una llamada al sistema de ficheros específico cuya dirección se mantiene en el inodo VFS que representa al directorio padre. Esto funciona porque siempre tenemos el inodo VFS del raíz de cada sistema de ficheros disponible y apuntado por el superbloque VFS de ese sistema. Cada vez que el sistema de ficheros real mira un inodo comprueba el caché de directorios. Si no está la entrada en el caché de directorios, el sistema de ficheros real toma el inodo VFS tanto del sistema de ficheros como del caché de inodos.

9.2.6 Creating a File in the Virtual File System

9.2.7 Unmounting a File System

The workshop manual for my MG usually describes assembly as the reverse of disassembly and the reverse is more or less true for unmounting a file system. A file system cannot be unmounted if something in the system is using one of its files. So, for example, you cannot umount `/mnt/cdrom` if a process is using that directory or any of its children. If anything is using the file system to be unmounted there may be VFS inodes from it in the VFS inode cache, and the code checks for this by looking through the list of inodes looking for inodes owned by the device that this file system occupies. If the VFS superblock for the mounted file system is dirty, that is it has been modified, then it must be written back to the file system on disk. Once it has been written to disk, the memory occupied by the VFS superblock is returned to the kernel's free pool of memory. Finally the `vfsmount` data structure for this mount is unlinked from `vfsmntlist` and freed.

9.2.8 The VFS Inode Cache

As the mounted file systems are navigated, their VFS inodes are being continually read and, in some cases, written. The Virtual File System maintains an inode cache to speed up accesses to all of the mounted file systems. Every time a VFS inode is read from the inode cache the system saves an access to a physical device. The VFS inode cache is implmented as a hash table whose entries are pointers to lists of VFS inodes that have the same hash value. The hash value of an inode is calculated from its inode number and from the device identifier for the underlying physical device containing the file system. Whenever the Virtual File System needs to access an inode, it first looks in the VFS inode cache. To find an inode in the cache, the system first calculates its hash value and then uses it as an index into the inode hash table. This gives it a pointer to a list of inodes with the same hash value. It then reads each inode in turn until it finds one with both the same inode number and the same device identifier as the one that it is searching for.

If it can find the inode in the cache, its count is incremented to show that it has another user and the file system access continues. Otherwise a free VFS inode must be found so that the file system can read the inode from memory. VFS has a number of choices about how to get a free inode. If the system may allocate more VFS inodes then this is what it does; it allocates kernel pages and breaks them up into new, free, inodes and puts them into the inode list. All of the system's VFS inodes are in a list pointed at by `first_inode` as well as in the inode hash table. If the system already has all of the inodes that it is allowed to have, it must find an inode that is a good candidate to be reused. Good candidates are inodes with a usage count of zero; this indicates that the system is not currently using them. Really important VFS inodes, for example the root inodes of file systems always have a usage count greater than zero and so are never candidates for reuse. Once a candidate for reuse has been located it is cleaned up. The VFS inode might be dirty and in this case it needs to be written

back to the file system or it might be locked and in this case the system must wait for it to be unlocked before continuing. The candidate VFS inode must be cleaned up before it can be reused.

However the new VFS inode is found, a file system specific routine must be called to fill it out from information read from the underlying real file system. Whilst it is being filled out, the new VFS inode has a usage count of one and is locked so that nothing else accesses it until it contains valid information.

To get the VFS inode that is actually needed, the file system may need to access several other inodes. This happens when you read a directory; only the inode for the final directory is needed but the inodes for the intermediate directories must also be read. As the VFS inode cache is used and filled up, the less used inodes will be discarded and the more used inodes will remain in the cache.

9.2.9 The Directory Cache

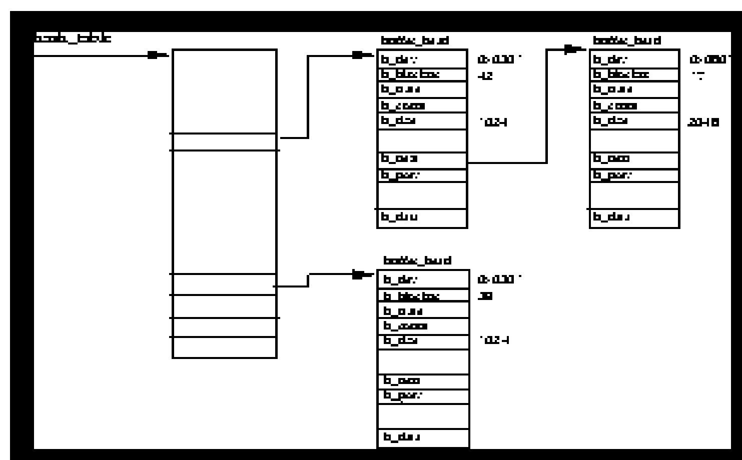
To speed up accesses to commonly used directories, the VFS maintains a cache of directory entries. As directories are looked up by the real file systems their details are added into the directory cache. The next time the same directory is looked up, for example to list it or open a file within it, then it will be found in the directory cache. Only short directory entries (up to 15 characters long) are cached but this is reasonable as the shorter directory names are the most commonly used ones. For example, `/usr/X11R6/bin` is very commonly accessed when the X server is running.

The directory cache consists of a hash table, each entry of which points at a list of directory cache entries that have the same hash value. The hash function uses the device number of the device holding the file system and the directory's name to calculate the offset, or index, into the hash table. It allows cached directory entries to be quickly found. It is no use having a cache when lookups within the cache take too long to find entries, or even not to find them.

In an effort to keep the caches valid and up to date the VFS keeps lists of Least Recently Used (LRU) directory cache entries. When a directory entry is first put into the cache, which is when it is first looked up, it is added onto the end of the first level LRU list. In a full cache this will displace an existing entry from the front of the LRU list. As the directory entry is accessed again it is promoted to the back of the second LRU cache list. Again, this may displace a cached level two directory entry at the front of the level two LRU cache list. This displacing of entries at the front of the level one and level two LRU lists is fine. The only reason that entries are at the front of the lists is that they have not been recently accessed. If they had, they would be nearer the back of the lists. The entries in the second level LRU cache list are safer than entries in the level one LRU cache list. This is the intention as these entries have not only been looked up but also they have been repeatedly referenced.

NOTA DE REVISIÓNDo we need a diagram for this?

9.3 The Buffer Cache



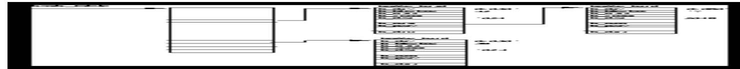


Figure 9.7: The Buffer Cache

As the mounted file systems are used they generate a lot of requests to the block devices to read and write data blocks. All block data read and write requests are given to the device drivers in the form of `buffer_head` data structures via standard kernel routine calls. These give all of the information that the block device drivers need; the device identifier uniquely identifies the device and the block number tells the driver which block to read. All block devices are viewed as linear collections of blocks of the same size. To speed up access to the physical block devices, Linux maintains a cache of block buffers. All of the block buffers in the system are kept somewhere in this buffer cache, even the new, unused buffers. This cache is shared between all of the physical block devices; at any one time there are many block buffers in the cache, belonging to any one of the system's block devices and often in many different states. If valid data is available from the buffer cache this saves the system an access to a physical device. Any block buffer that has been used to read data from a block device or to write data to it goes into the buffer cache. Over time it may be removed from the cache to make way for a more deserving buffer or it may remain in the cache as it is frequently accessed.

Block buffers within the cache are uniquely identfied by the owning device identifier and the block number of the buffer. The buffer cache is composed of two functional parts. The first part is the lists of free block buffers. There is one list per supported buffer size and the system's free block buffers are queued onto these lists when they are first created or when they have been discarded. The currently supported buffer sizes are 512, 1024, 2048, 4096 and 8192 bytes. The second functional part is the cache itself. This is a hash table which is a vector of pointers to chains of buffers that have the same hash index. The hash index is generated from the owning device identifier and the block number of the data block. Figure 9.7 shows the hash table together with a few entries. Block buffers are either in one of the free lists or they are in the buffer cache. When they are in the buffer cache they are also queued onto Least Recently Used (LRU) lists. There is an LRU list for each buffer type and these are used by the system to perform work on buffers of a type, for example, writing buffers with new data in them out to disk. The buffer's type reflects its state and Linux currently supports the following types:

clean
: Unused, new buffers,

locked
: Buffers that are locked, waiting to be written,

dirty
: Dirty buffers. These contain new, valid data, and will be written but so far have not been scheduled to write,

shared
: Shared buffers,

unshared
: Buffers that were once shared but which are now not shared,

Whenever a file system needs to read a buffer from its underlying physical device, it trys to get a block from the buffer cache. If it cannot get a buffer from the buffer cache, then it will get a clean one from the appropriate sized free list and this new buffer will go into the buffer cache. If the buffer that it needed is in the buffer cache, then it may or may not be up to date. If it is not up to date or if it is a new block buffer, the file system must request that the device driver read the appropriate block of data from the disk.

Like all caches, the buffer cache must be maintained so that it runs efficiently and fairly allocates cache entries between the block devices using the buffer cache. Linux uses the `bdflush` kernel daemon to perform a lot of housekeeping duties on the cache but some happen automatically as a result of the cache being used.

9.3.1 The `bdflush` Kernel Daemon

The `bdflush` kernel daemon is a simple kernel daemon that provides a dynamic response to the system

having too many dirty buffers; buffers that contain data that must be written out to disk at some time. It is started as a kernel thread at system startup time and, rather confusingly, it calls itself «kflushd» and that is the name that you will see if you use the ps command to show the processes in the system. Mostly this daemon sleeps waiting for the number of dirty buffers in the system to grow too large. As buffers are allocated and discarded the number of dirty buffers in the system is checked. If there are too many as a percentage of the total number of buffers in the system then `bdflush` is woken up. The default threshold is 60will be woken up anyway. This value can be seen and changed using the update command:

```
# update -d

bdflush version 1.4
0:    60 Max fraction of LRU list to examine for dirty blocks
1:   500 Max number of dirty blocks to write each time bdflush activated
2:    64 Num of clean buffers to be loaded onto free list by refill_freelist
3:   256 Dirty block threshold for activating bdflush in refill_freelist
4:    15 Percentage of cache to scan for free clusters
5:  3000 Time for data buffers to age before flushing
6:   500 Time for non-data (dir, bitmap, etc) buffers to age before flushing
7:  1884 Time buffer cache load average constant
8:     2 LAV ratio (used to determine threshold for buffer fratricide).
```

All of the dirty buffers are linked into the `BUF_DIRTY` LRU list whenever they are made dirty by having data written to them and `bdflush` tries to write a reasonable number of them out to their owning disks. Again this number can be seen and controlled by the update command and the default is 500 (see above).

9.3.2 The update Process

The update command is more than just a command; it is also a daemon. When run as superuser (during system initialisation) it will periodically flush all of the older dirty buffers out to disk. It does this by calling a system service routine that does more or less the same thing as `bdflush`. Whenever a dirty buffer is finished with, it is tagged with the system time that it should be written out to its owning disk. Every time that update runs it looks at all of the dirty buffers in the system looking for ones with an expired flush time. Every expired buffer is written out to disk.

9.4 The /proc File System

The `/proc` file system really shows the power of the Linux Virtual File System. It does not really exist (yet another of Linux's conjuring tricks), neither the `/proc` directory nor its subdirectories and its files actually exist. So how can you cat `/proc/devices`? The /proc file system, like a real file system, registers itself with the Virtual File System. However, when the VFS makes calls to it requesting inodes as its files and directories are opened, the `/proc` file system creates those files and directories from information within the kernel. For example, the kernel's `/proc/devices` file is generated from the kernel's data structures describing its devices.

The `/proc` file system presents a user readable window into the kernel's inner workings. Several Linux subsystems, such as Linux kernel modules described in chapter modules-chapter, create entries in the the `/proc` file system.

9.5 Device Special Files

Linux, like all versions of Unix presents its hardware devices as special files. So, for example, /dev/null is the null device. A device file does not use any data space in the file system, it is only an access point to the device driver. The EXT2 file system and the Linux VFS both implement device files as special types of inode. There are two types of device file; character and block special files. Within the kernel itself, the device drivers

implement file semantices: you can open them, close them and so on. Character devices allow I/O operations in character mode and block devices require that all I/O is via the buffer cache. When an I/O request is made to a device file, it is forwarded to the appropriate device driver within the system. Often this is not a real device driver but a pseudo-device driver for some subsystem such as the SCSI device driver layer. Device files are referenced by a major number, which identifies the device type, and a minor type, which identifies the unit, or instance of that major type. For example, the IDE disks on the first IDE controller in the system have a major number of 3 and the first partition of an IDE disk would have a minor number of 1. So, `ls -l` of `/dev/hda1` gives:

```
$ brw-rw----   1 root    disk       3,    1  Nov 24  15:09 /dev/hda1
```

Within the kernel, every device is uniquely described by a `kdev_t` data type, this is two bytes long, the first byte containing the minor device number and the second byte holding the major device number. The IDE device above is held within the kernel as 0x0301. An EXT2 inode that represents a block or character device keeps the device's major and minor numbers in its first direct block pointer. When it is read by the VFS, the VFS inode data structure representing it has its `i_rdev` field set to the correct device identifier.

Footnotes:

[1] Bueno, no con conocimiento, sin embargo me he topado con sistemas operativos con más abogados que programadores tiene Linux

---

File translated from T_EX by T_TH, version 1.0.

© 1996-1998 David A Rusling derechos de autor.