LECCIÓN 17: SUPERBLOQUE DE EXT2

LECCIÓN 17: SUPERBLOQUE DE EXT2....1
17.1 Introducción....2
17.2 Segundo sistema de ficheros extendido (Ext2)....7
17.3 Estructura Física de un SF EXT2....12
17.4 Estructuras de datos....14
El superbloque....14
Estructura ext2_sb_info....21
Descriptores de grupo de bloques....25
17.5 Operaciones vinculadas al superbloque....27
ext2_error....29
ext2_warning....30
ext2_put_super....30
ext2_parse_options....32
ext2_setup_super....35
ext2_check_descriptors....36
ext2_fill_super....38
ext2_commit_super....45
ext2_sync_super....46
ext2_write_super....46
ext2_remount....48
ext2_statfs....50
ext2_init_ext2_fs....51
17.6 Journaling....52
17.7 Tercer sistema de ficheros extendido (Ext3)....54
17.8 Sistema de ficheros ReiserFS....56

17.1 Introducción

En los primeros tiempos de Linux, allá por 1991, éste se desarrolló bajo el sistema operativo Minix. Era más sencillo compartir disquetes entre ambos sistemas que desarrollar un nuevo sistema de ficheros, por ello Linus Torvalds decidió implementar soporte para el Minix FS en Linux. El Minix FS era un software eficiente y relativamente libre de errores. El sistema de ficheros Minix presentaba dos limitaciones importantes:

- Las direcciones de los bloques se guardan en enteros de 16 bits, de modo que el tamaño máximo de todo el sistema de ficheros está restringido a 64 Mb.
- Los directorios contienen entradas de tamaño fijo, siendo la longitud máxima de un nombre de archivo 14 caracteres.

Las restricciones en el diseño del Minix FS eran tan incomodas que la comunidad empezó a pensar y trabajar en la implementación de nuevos FS en Linux.

Para facilitar la incorporación de nuevos FS en el núcleo de Linux se desarrolló la capa VFS (Virtual File System). Ésta fué inicialmente escrita por Cris Provenzano, y posteriormente reescrita por el propio Linux Torvalds antes de integrarla en el núcleo.

Tras la integración del VFS en el núcleo, un nuevo FS denominado Extended File System se introdujo en Abril de 1992, en el núcleo 0.96c. Se eliminaban así dos grandes limitaciones del Minix FS: su tamaño máximo era ahora 2 Gb y la longitud máxima del nombre de un fichero se extendía a 255 caracteres. Era una mejora, pero aún había problemas:

- Ext no soportaba registro separado (timestamps) de acceso, modificación de inodo y modificación de datos.
- Se empleaban listas enlazadas para mantener información de los bloques libres y de los inodos y todo esto provocaba un mal comportamiento: a lo largo del tiempo las listas se iban desordenando y el FS terminaba fragmentado.

Es por esto que surgió el segundo sistema de ficheros extendido (Ext2fs) Remy Card en 1994. Además de incluir varias características nuevas, cabe resaltar la alta eficiencia y robustez.

Durante los últimos años, el Ext2 o Ext2fs (Second EXTended FileSystem, segundo sistema de ficheros extendido) ha sido de hecho el sistema de ficheros más utilizado entre los sistemas Linux. Sin embargo, a medida que Linux ha ido desplazando a Unix y otros sistemas operativos en más servidores y ambientes computacionales, al Ext2 se le ha llevado a su límite. De hecho, muchos de los requisitos más comunes de hoy en día como grandes particiones de discos duros, recuperación rápida de caídas del sistema, rendimiento alto en operaciones de

Entrada / Salida (I/O) y la necesidad de almacenar miles y miles de ficheros representando Terabytes de información, ha superado las posibilidades del Ext2.

En la siguiente tabla se resumen las características fundamentales de varios FS:

	Minix FS	Ext FS	Ext2 FS
Max FS size	64 MB	2 GB	4 TB
Max file size	64 MB	2 GB	2 GB
Max file name	16/30 c	255 c	255 c
3 times support	No	No	Si
Extensible	No	No	Si
Var. block size	No	No	Si
Maintained	Si	No	Si

Afortunadamente, un número variado de otros sistemas de fichero alternativos de Linux han aparecido en escena para intentar solventar las limitaciones del Ext2. Estos son; el Ext3, Ext4 el ReiserFS, el XFS y el JFS. Además de los requisitos especificados más arriba, todos estos sistemas de ficheros soportan también el Journaling (o diario en español), una característica ciertamente demandada por las empresas pero que beneficia a cualquiera que trabaje con Linux. Un sistema de ficheros Journaling puede simplificar los reinicios, reducir los problemas derivados de la fragmentación y acelerar las operaciones I/O. Un sistema con journaling es un sistema de ficheros tolerante a fallos en el cual la integridad de los datos está asegurada porque las modificaciones de la meta-información de los ficheros son primero grabadas en un registro cronológico (log o journal, que simplemente es una lista de transacciones) antes de que los bloques orgininales sean modificados. En el caso de un fallo del sistema, un sistema con journaling asegura que la consistencia del sistema de ficheros es recuperada.

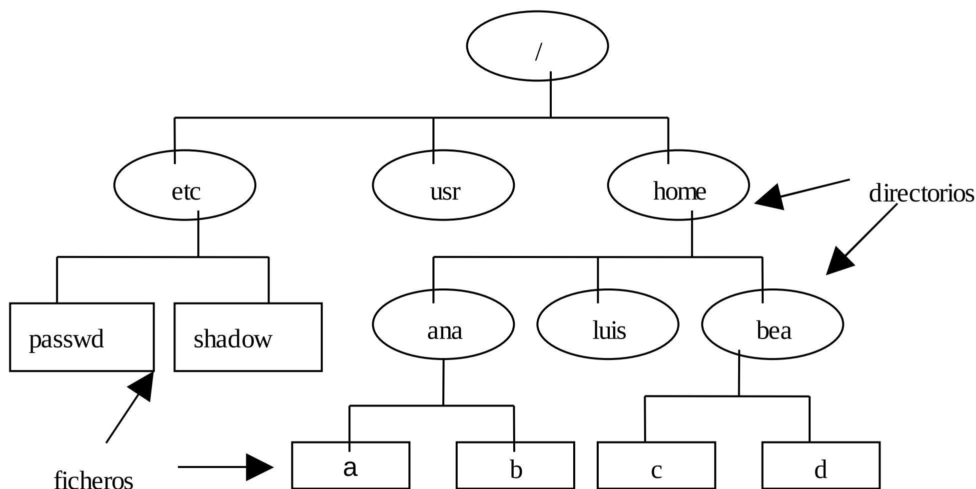
Podemos clasificar los sistemas de ficheros soportados por Linux en 3 categorías: Basados en disco: Estos sistemas son, ext2, ext3, ReiserFX, XFS, JFS, ISO9660, etc. Sistemas remotos (de red): NFS, Coda, Samba, etc. Sistemas especiales: procfs, ramfs y devfs.

Sistema de Ficheros

El sistema de ficheros es una de las partes más importantes de un sistema operativo. El sistema de ficheros almacena y gestiona los datos que los usuarios tienen almacenados en los discos duros, y asegura que los datos que se obtienen son idénticos a los que se almacenaron originalmente. Además de almacenar los datos de usuarios en ficheros (elemento lógico del sistema de ficheros), el sistema de ficheros también crea y gestiona información sobre los ficheros y sobre sí mismo. Por otro lado, además de garantizar la integridad de los datos, los sistemas de ficheros se esperan que sean extremadamente fiables y que tengan un alto nivel de rendimiento.

Desde el punto de vista del sistemas, toda la información de un ordenador existe como bloques de datos dentro de algún dispositivo de almacenaje, organizados utilizando estructuras especiales dentro de particiones (subconjunto lógico del medio de almacenamiento) las cuales se organizan a su vez en ficheros, directorios y espacio no asignado (libre).

Los sistemas de ficheros se crean en particiones de discos para permitir a las aplicaciones almacenar y organizar la información en forma de ficheros y directorios. Linux, como los sistemas basados en Unix, utilizan un sistema de ficheros jerarquizado compuesto como ya hemos dicho por ficheros y directorios, los cuales pueden contener tanto ficheros como otros directorios.



Para que los usuarios tengan acceso a los ficheros y directorios del sistema de ficheros previamente hay que realizar el comando mount, el cual se lleva acabo como parte del proceso de arranque del sistema. El listado de los sistemas de ficheros montados en el arranque se almacenan en el fichero /etc/fstab (fstab – **F**ile**S**ystem **TAB**le), mientras que el listado de los sistemas de fichero montados en el sistema se almacenan en el fichero /etc/mtab (mtab –**M**ount **TAB**le). Durante el arranque se montaran aquellos sistemas de ficheros especificados en el fichero ‘fstab’.

Cuando un sistema de ficheros se monta durante el proceso de arranque, un bit en la cabecera del sistema de ficheros (conocido como “Bit Clean” o Bit limpio) permanece a cero indicando que el sistema de ficheros esta en uso (que las estructuras de datos encargadas de la gestión de asignación de espacios disponibles y de la organización de ficheros y directorios están actualmente en uso).

Se dice que un sistema de ficheros es “Consistente” cuando todos los bloques de datos en el sistema de ficheros están o en uso o libres y que cada bloque de datos corresponde a un fichero o directorio accesibles en cualquier momento y lugar. Cuando un sistema Linux es intencionadamente apagado usando su comando específico, todos los sistemas de ficheros de desmontan automáticamente. El desmontado del sistema de ficheros durante un apagado estándar pone a uno el Bit Limpio en la cabecera indicando que el sistema de ficheros fue correctamente desmontado y que por la tanto se supone Consistente.

Años de depuración y rediseño del sistema de ficheros y el uso de algoritmos extremadamente efectivos en la escritura en disco han logrado eliminar las corrupciones causadas por accesos a disco tanto de aplicaciones como del propio núcleo. Pero la eliminación de corrupción de datos debido a caídas de tensión y otras situaciones indebidas es arena de otro costal. Cuando un sistema Linux cae o es simplemente apagado sin seguir los procedimientos estándar de apagado, el Bit Limpio de la cabecera del sistema de ficheros no se actualiza como es debido (no se pondrá a uno). La siguiente vez que el sistema arranque, el proceso de montaje detectara que sistemas de ficheros no tienen a uno el Bit Limpio, y verificara la consistencia de los datos usando la utilidad ‘fsck’ (FileSystem ChecK, Chequeo del Sistema).

Ejecutar ‘fsck’ en un número de sistemas de ficheros grandes puede llevar bastante tiempo. La razón por la que pueden existir inconsistencias en un sistema de ficheros que no haya sido desmontado como es debido, puede deberse al hecho de que se estuvieran realizando escrituras en disco en el momento del apagado. Puede que ciertas aplicaciones estuvieran actualizando información contenida en ficheros o el sistema podría haber estado actualizando los metadatos (metadata) del sistema de ficheros, los cuales son “datos sobre los datos del sistema de ficheros”, en otras palabras, información sobre que bloques están asignados a que ficheros, que ficheros se encuentran en que directorios. Las inconsistencias en ficheros de datos son bastante perjudiciales, pero las inconsistencias en los metadatos del sistema de ficheros son las que producen perdidas de ficheros y todo tipo de pesadillas operacionales.

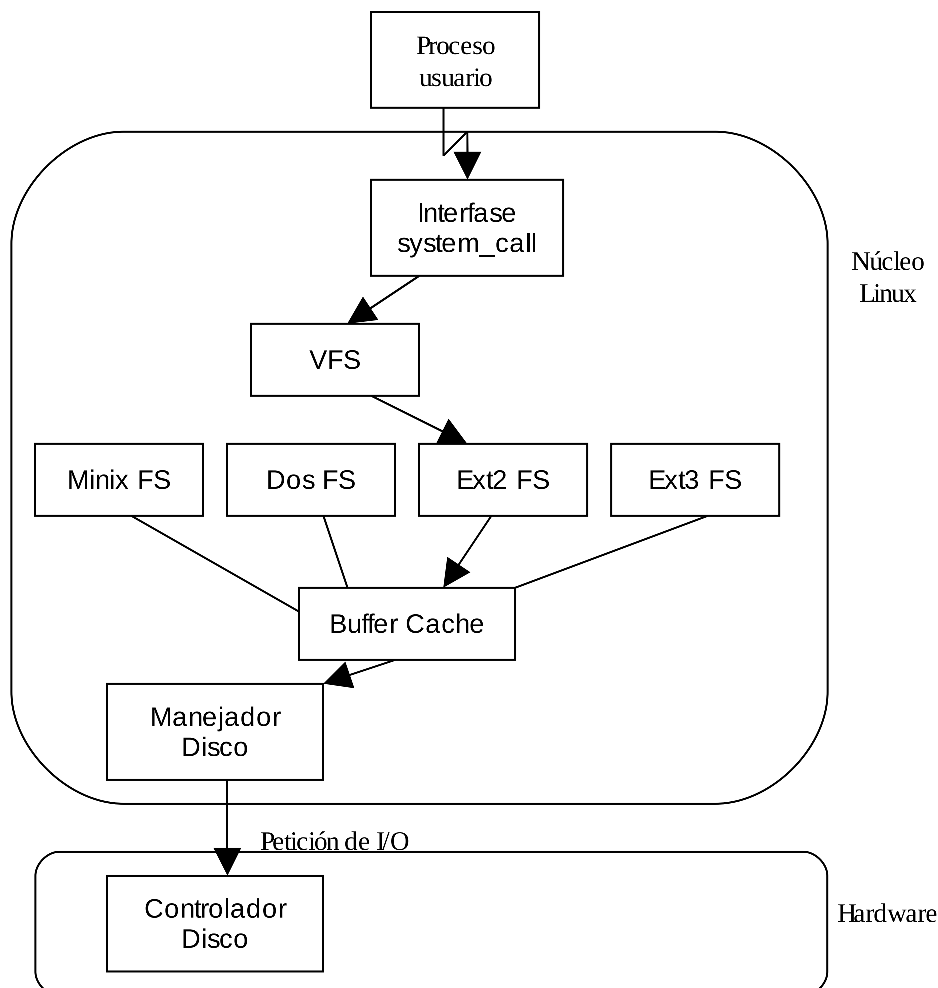
Criterios de Selección

Criterios a la hora de elegir un sistema de ficheros:

ESTABILIDAD Y FIABILIDAD – Con respecto a la pérdida de datos
RENDIMIENTO (“lo capaz que es”) – Requisitos de rendimiento
CAPACIDAD (“la capacidad que tiene”) – Los límites del sistema de ficheros
MANTENIMIENTO – El tipo de mantenimiento que necesita
AFINAMIENTO (tuning) – Facilidad de configuración y afinamiento del sistema
SOPORTE – Por parte del grupo de desarrollo

Sistema Virtual de Archivos (VFS)

Los distintos SO suelen usar su propio Sistema de Ficheros, esto complica la compartición de datos en un disco duro entre distintos SO. Linux ofrece soporte para distintos Sistemas de Ficheros como EXT2, Minix, FAT…
Los procesos necesitan de un acceso uniforme a los datos, sin preocuparse de qué Sistema de Ficheros se trata, el VFS es la interfaz entre las llamadas al sistema y la gestión de archivos. Las llamadas se dirigen al VFS, que redirige la petición al módulo que gestiona el archivo.



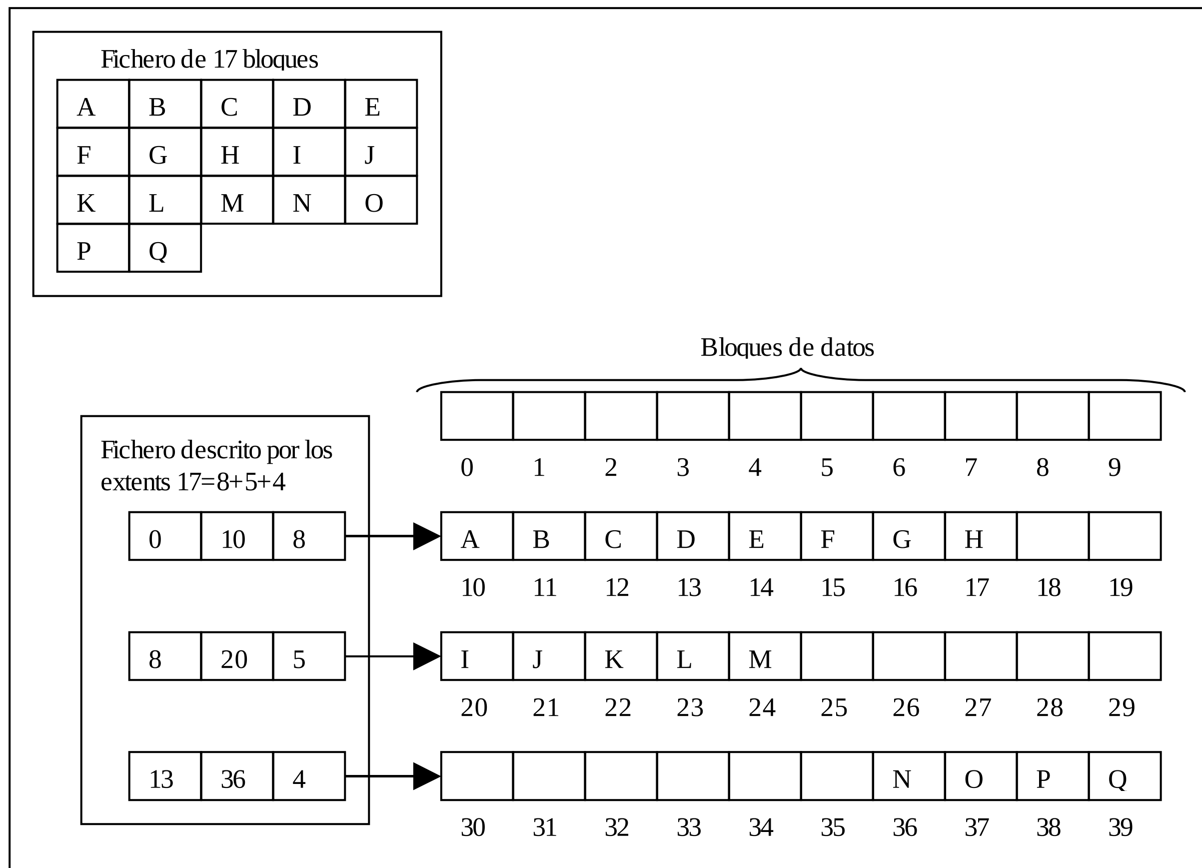
17.2 Segundo sistema de ficheros extendido (Ext2)

Características de un sistema de ficheros como el Ext2:

Todo sistema de ficheros implementa una serie de conceptos básicos derivados del sistema operativo Unix [Batch 1986]. Los archivos son representados por inodos, los directorios son símplemente archivos que contienen una lista de entradas (que apuntan a otros directorios o a archivos) y los dispositivos de entrada salida pueden ser accedidos mediante ficheros especiales. Veamos algunos de estos conceptos básicos.

BLOQUE lógico es la unidad más pequeña de almacenamiento que puede ser asignado por el sistema de ficheros. Un bloque lógico se mide en bytes, un fichero se almacenara en uno o varios bloques.
VOLUMEN lógico puede ser un disco físico o un subconjunto del espacio de un disco físico. Un volumen lógico se conoce como un *PARTICÍON* de disco.
Asignación de bloques es un método donde el sistema de fichero asigna un bloque cada vez. En este método, un puntero a cada bloque se mantiene y almacena.
FRAGMENTACIÓN INTERNA ocurre cuando un fichero no ocupa por completo un bloque. Por ejemplo, si tenemos un fichero de 10K y un tamaño de bloque de 8K, el sistema de ficheros asignara dos bloques para alojar el fichero, pero 6K del segundo bloque se desaprovecharan. Nótese que a mayor tamaño de bloque mayor será el espacio desperdiciado.
FRAGMENTACIÓN EXTERNA ocurre cuando los bloques lógicos que forman un fichero se encuentran esparcidos por todo el disco (no están contiguos). Este tipo de fragmentación produce un bajo rendimiento.
EXTENT es un conjunto de bloques contiguos. Cada Extent se describe como una tripleta, que consiste en un offset del fichero, el número del bloque de comienzo y el tamaño. El offset del fichero es el desplazamiento del primer bloque del extent desde el comienzo del fichero (nos dice a que bloque corresponde dentro del fichero), el numero del bloque de comienzo es el numero del primer bloque del extent y el tamaño es numero de bloques del extent. Los extent se asignan y se monitorizan como una unidad independiente (forman un todo) por lo que un único puntero monitorizara un grupo de bloques. Para ficheros muy grandes, la asignación de extents (o extent allocation) es una técnica mucho más eficiente que la técnica de asignación de bloques. La figura muestra el uso de los extents.

El fichero requiere 17 bloques y el sistema de ficheros puede realizar una asignación de un extent de 8 bloques, un segundo extent de 5 bloques y un tercero de otros 4. El sistema de ficheros será algo como en la figura de arriba. El primer extent tiene un desplazamiento de cero (bloque ‘A’ del fichero), localización 10 (numero del bloque de comienzo del extent) y un tamaño de 8. El segundo extent tiene un offset de 8 (bloque ‘I’ del fichero), localización 20 y tamaño 5. El último extent tiene un offset de 13 (bloque ‘N’ del fichero), localización 35 y tamaño 4.

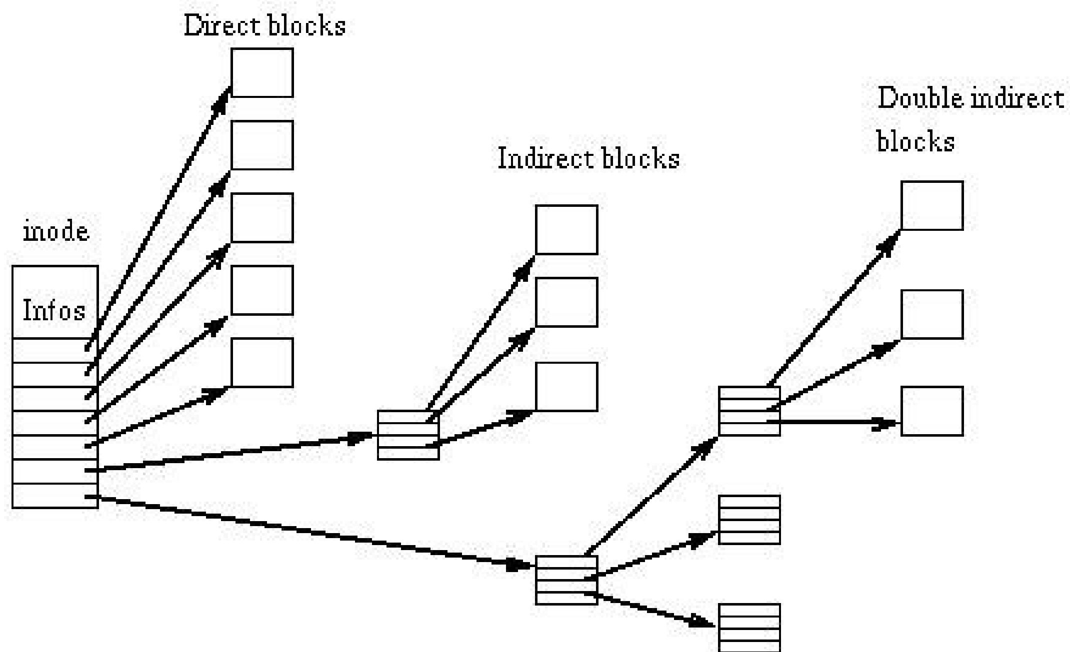


LISTA DE CONTROL DE ACCESO (Access Control List –ACL). En vez de clasificar los usuarios de un fichero en tres clases –propietario, grupo y otros –esta lista se asocia con cada fichero para especificar los derechos de acceso para cualquier usuario específico o combinación de usuarios.

METADATOS del sistema de ficheros son las estructuras de datos internas del sistema de ficheros, información acerca del sistema de fichero excepto la información dentro de los ficheros, es la información necesaria para gestionar el SF y los bloques. Los metadatos incluyen la fecha y hora, la información sobre el propietario, los permisos de acceso de los ficheros, otra tipo de información de seguridad como la lista de control de acceso (ACL) si existe.

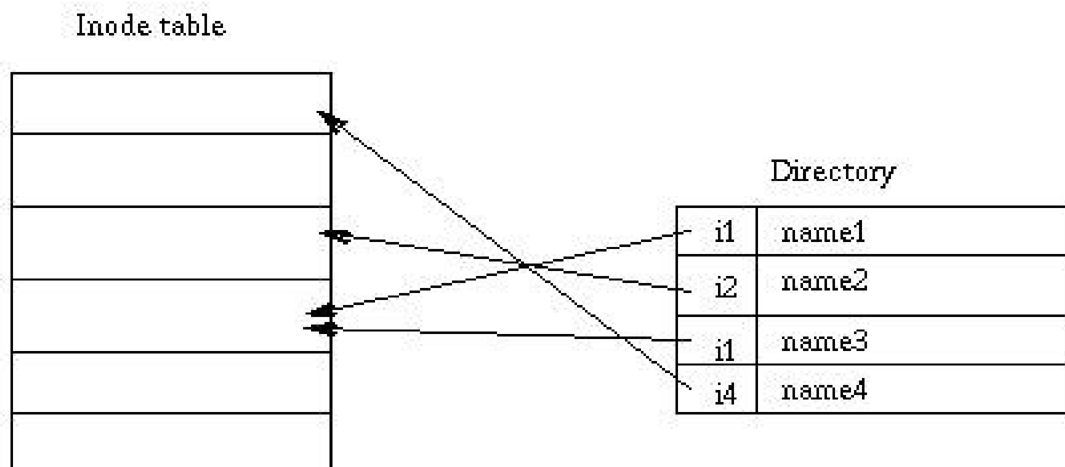
INODE Cada archivo se representa mediante una estructura llamada inodo. Cada inodo contiene la descripción del archivo: tipo, derechos de acceso, propietarios, marcas de tiempo, tamaño y punteros a los bloques de datos. Las direcciones de los bloques de datos asignados al archivos se guardan en su inodo. Cuando un usuario solicita una operación de E/S en un archivo, el núcleo convierte el offset actual (posición dentro del archivo dónde se desea leer o escribir) a un número de bloque, emplea este número como un índice en la tabla de direcciones de bloques y lee o escribe el bloque físico.

La siguiente figura representa el campo de direcciones de los bloques de datos dentro de un inodo:



DIRECTORIO Los directorios se estructuran en un árbol jerárquico. Cada directorio puede contener archivos y subdirectorios. Los directorios se implementan como un tipo especial de archivos. Realmente un directorio es un archivo que contiene una lista de entradas. Cada entrada contiene un número de inodo y un nombre de archivo. Cuando un proceso emplea una ruta (path), el núcleo busca en los directorios el correspondiente número de inodo. Una vez que el nombre ha sido convertido en un número de inodo, éste se carga en la memoria y se usa en peticiones posteriores.

La siguiente figura muestra la estructura de un directorio:



ENLACES Los FS Unix implementan el concepto de enlace. Distintos nombres pueden asociarse con un inodo. El inodo contiene un campo que contabiliza el número de enlaces que apuntan al archivo. Añadir un enlace consiste únicamente en crear una nueva entrada de directorio en la que el número de inodo apunta al inodo que se está enlazando, e incrementar el contador de enlaces en el inodo. Cuando se borra un enlace el núcleo decrementa el contador de enlaces y libera el inodo en caso de que éste llegue a cero.
Este tipo de enlaces se denomina **enlace duro** y únicamente se puede emplear dentro de un mismo FS: es imposible crear enlaces duros entre distintos FS. Los enlaces duros pueden apuntar únicamente a ficheros, no a directorios. Esto es así para evitar la aparición de ciclos en el árbol de directorios.

Existe otro tipo de enlace denominado **enlace simbólico**. Los enlaces simbólicos son únicamente archivos que contiene un nombre de archivo. Cuando el núcleo encuentra un enlace simbólico mientras realiza la conversión de ruta a inode, reemplaza el nombre del enlace por su contenido y reinicia la interpretación de la ruta. Como un enlace simbólico no apunta a un inodo es posible crear enlaces simbólicos entre distintos FS. Sin embargo los enlaces simbólicos consumen espacio en disco (inodo + bloques de datos), y pueden causar sobrecargas en el sistema debido a que el núcleo debe reiniciar la interpretación cuando encuentra un enlace simbólico.

FICHEROS DE DISPOSITIVO En los sistemas operativos de la familia Unix los dispositivos de E/S se acceden a través de archivos especiales. Un archivo especial de dispositivo no usa espacio en el FS. Es únicamente un punto de acceso al controlador del dispositivo.
Existen dos tipos de archivos de dispositivo: de caracter y de bloque. Los de caracter permiten operadiones E/S en modo caracter, mientras que los de bloque requieren que la información sea escrita en modo bloque mediante el buffer cache. Cuando se realiza una petición E/S en un fichero especial lo que realmente se hace es redireccionar a un (pseudo) controlador de dispositivo. Un archivo especial se referencia por un número mayor, que identifica el dispositivo, y un número menor, que identifica la unidad.

Otras características del Ext2:

Ext2fs soporta los tipos de ficheros estandar de Unix: ficheros regulares, directorios, ficheros especiales de dispositivo y enlaces simbólicos.
Ext2fs puede gestionar FS creados en particiones grandes, hasta 4 Tb.
Ext2fs permite nombres de fichero largos. Emplea entradas de directorio de longitud variable. El tamaño máximo de nombre de fichero es 255 caracteres, aunque este límite puede ser extendido hasta 1012.
Ext2fs reserva algunos bloques para el superusuario (root). Normalmente el 5% de los bloques se reservan para éste. Esto permite al administrador recuperar el sistema fácilmente ante situaciones en las que los procesos de usuario llenen el sistema de ficheros.

Ext2fs soporta algunas extensiones que normalmente no se encuentran en sistemas de ficheros tipo Unix.

Los atributos de archivo permiten a los usuarios modificar el comportamiento del núcleo cuando actua sobre un conjunto de ficheros. Se puede establecer atributos para archivos y directorios. En el caso de un directorio, los archivos nuevos creados en él heredan los atributos.

Ext2fs permite al administrador elegir el tamaño del bloque lógico cuando se crea el FS. Tamaños típicos de bloque son 1, 2 o 4 Kb. Usar tamaños de bloque grandes puede acelerar las operaciones de E/S ya que se requiere una menor cantidad de peticiones, sin embargo puede producir un malgasto de espacio debido a la fragmentación interna (el último bloque de cada archivo puede no estar lleno). Muchas de las ventajas de emplear tamaños de bloque grandes se obtienen emplenado técnicas de preasignación implementadas en Ext2fs, pudiendo elegir sin perjuicio del rendimiento tamaños de bloque más razonables.

Ext2fs implementa enlaces simbólicos rápidos. Éstos no utilizan bloques de datos. El nombre destino se almacena en el propio inodo, de este modo se ahorra espacio. Como el espacio disponible en un inodo es limitado el tamaño máximo del destino de un enlace simbólico rápido es de 60 caracteres.

Ext2fs mantiene información sobre el estado del FS. El núcleo emplea un campo en el superbloque (explicado posteriormente) para indicar el estado del FS. Cuando un FS se monta en modo lectura/escritura se marca su estado como "no limpio". Cuando se desmonta o se remonta en modo de sólo lectura su estado se marca como "limpio". Durante el arranque del sistema se comprueba esta información para actuar consecuentemente. El núcleo también guarda errores en este campo. Ésta información se comprueba durante el arranque a pesar de que el estado de FS sea "limpio".

No realizar comprobaciones del FS puede ser peligroso, por ello Ext2fs proporciona dos maneras de forzar estas comprobaciones a intervalos regulares. Por un lado tenemos un contador de montajes. Cada vez que el sistema se monta en modo lectura/escritura, se incremente el contador. Cuando el contador alcanza una determinada cifra se fuerza la comprobación. En el superbloque también se almacena el momento de la última comprobación, así como el intervalo máximo entre comprobaciones. Estos campos permiten al administrador fijar comprobaciones periódicas.

El sistema de ficheros preasigna bloques de datos en disco a ficheros regulares antes de ser usados. Así, cuando el fichero aumenta de tamaño, varios bloques ya están reservados en posiciones adyacentes, reduciendo así la fragmentación de los ficheros.

Soporta chequeos automáticos de consistencia del sistema de ficheros en el inicio del sistema. Los chequeos son realizados por un programa llamado e2fsck.

Soporta los llamados ficheros ‘inmutables’ (ficheros que no pueden ser modificados, borrados o renombrados) y los ficheros de solo agregar (append-only –la información solo se les puede añadir por el final del mismo).

Borrado Lógico. Permite a los usuarios la recuperación sencilla, si es necesario, del contenido de un fichero previamente eliminado.

Compatibilidad tanto con el SVR4 (Unix System V Release 4) y con la semántica BSD.

17.3 Estructura Física de un SF EXT2

Organización del disco: BLOQUES Y GRUPOS DE BLOQUES

La estructura física de Ext2fs está fuertemente influenciada por la distribución (layout) del sistema de ficheros BSD [McKusic et al. 1984]. Un sistema de archivos se compone de grupos de bloques. Los grupos de bloques son análogos a los grupos de cilindros de los BSD FFS. Sin embargo los grupos de bloques no se encuentran atados a la disposición física de los bloques en el disco.

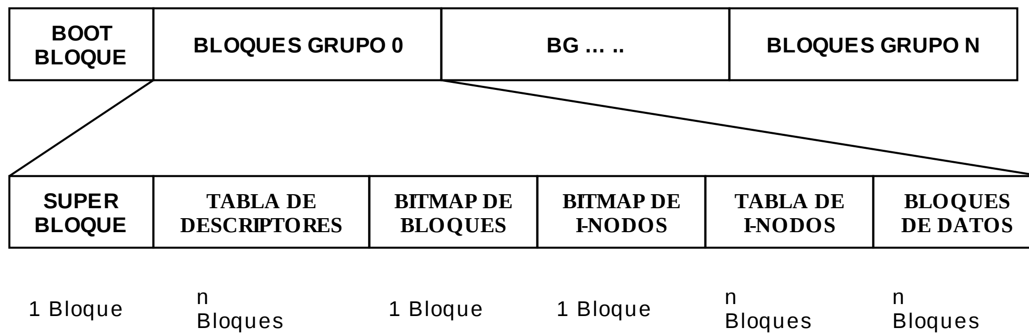
Ext2fs divide el disco en pequeñas unidades lógicas llamadas '**bloques**'. Estas son las unidades mínimas de disco que pueden ser asignadas y su tamaño puede ser escogido (1, 2 o 4 KB) en el momento de creación del sistema de ficheros.

El ext2 divide las particiones lógicas que ocupa en **Grupo de Bloques (BG),** conjuntos de bloques secuenciales. En realidad, el sistema de ficheros es manejado globalmente como una serie de Grupos de Bloques, lo que mantiene la información relacionada físicamente cercana en el disco y simplifica las tareas de gestión. Como resultado, la mayor parte de la gestión del sistema de ficheros se reduce a la gestión de un sólo grupo de bloques.

La estrcutura física de un sistema de un fichero es como sigue:

Boot Sector	Block Grupo 1	Block Grupo 2	... ...	Block Grupo N

Cada grupo contiene una copia redundante de información de control crucial del FS (el superbloque y los descriptores del FS) además de parte del propio FS (block bitmap, inode bitmap, una porción de la tabla de inodos, y bloques de datos). La estructura de un grupo de bloques es como sigue:



Cada Grupo de Bloques BG contiene:

- Una copia del superbloque.
- Tabla con los descriptores del grupo.
- Un bitmap de bloques.
- Un bitmap de inodos.
- Una parte de la tabla de inodos.
- Bloques de datos.

Usar grupos de bloques es un gran avance en términos de fiabilidad: el hecho de guardar una copia de las estructuras de control superbloque y tabal de descriptores en cada grupo facilita la recuperación del FS ante una corrupción del superbloque. También ayuda a alcanzar un buen rendimiento: reduciendo la distancia entre la tabla de inodos y los bloques de datos es posible reducir las búsquedas de las cabezas de los discos.

Los directorios en Ext2fs son listas enlazadas con entradas de longitud variable. Cada entrada contiene el número de inodo, la longitud de la entrada, el nombre de archivo y su longitud. La estructura de un directorio se muestra a continuación:

inode number	entry length	name length	filename

Como ejemplo se muestra a continuación la estructura de un directorio que contiene 3 archivos: file1, long_file_name, y f2:

i1	16	05	file1
i2	40	14	long_file_name
i3	12	02	f2

Optimizaciónes de rendimiento

El sistema de ficheros Ext2fs contiene muchas optimizaciones orientadas a mejorar el rendimiento a la hora de realizar lecturas y escrituras. Aprovechando la gestión del buffer cache realizando lecturas por adelantado: cuando se lee un bloque el código del núcleo solicita varios bloques contiguos. De este modo se pretende asegurar que el siguiente bloque a leer ya se encuentra cargado en el buffer cache. Ext2fs realiza lecturas por adelantado en lecturas secuenciales de archivos y en lecturas de directorios, ya sean explícitas o implícitas.

Así mismo Ext2fs cuenta con numerosas optimizaciones de asignación de bloques. Los grupos de bloques se emplean para reunir inodos y sus datos, de modo que el núcleo siempre intenta situar un inodo y sus bloques de datos en el mismo grupo de bloques.

A la hora de escribir Ext2fs preasigna 8 bloques adjacentes cuando se asigna un nuevo bloque, de este modo se evita solicitar espacio bloque a bloque, además de conseguir que la información de un mismo archivo se encuentre en bloques contiguos.

17.4 Estructuras de datos

El superbloque

El superbloque es un bloque que contiene la información más relevante y describe al sistema de ficheros. Se encuentra en el offset fijo 1024 del disco y ocupa 1024 bytes.

El superbloque contiene una descripción del tamaño y forma del sistema de ficheros. Esta información permite usar y mantener dicho sistema de ficheros. En condiciones normales únicamente el superbloque situado en el grupo de bloques 0 se lee cuando el sistema de ficheros es montado, sin embargo cada grupo de bloques contiene una copia duplicada. Entre otra información el superbloque contiene:

Número Mágico (s_magic): Permite al software que reliaza el montaje comprobar que se trata efectivamente del superbloque para un sistema de ficheros Ext2.

Nivel de revisión (s_rev_level): Los números mayor y menor de revisión permiten identificar ciertas características del sistema de archivos que encuentran presentes únicamente en ciertas versiones.

Contador de montajes (s_mnt_count) y máximo número de montajes (s_max_mnt_count): Juntos permiten determinar si se debe realizar una comprobación completa del sistema de ficheros, “e2fsck”.

Número del grupo de bloques (s_block_group_nr): El número del grupo de bloques que contiene la copia del superbloque.

Tamaño de bloque (s_log_block_size): El tamaño del bloque en bytes.

Bloques por grupo (s_blocks_per_group): El número de bloques en un grupo. Igual que el tamaño de bloque, este valor se fija cuando se crea el sistema de ficheros.

Bloques libres (s_free_blocks_count): El número de bloques libres en el sistema de ficheros.

Inodos libres (s_free_inodes_count): El número de inodos libres en el sistema de ficheros.

Primer Inodo (s_first_ino): El número del primer inodo del sistema de ficheros. El primer inodo en un sistema Ext2 raíz debe ser la entrada de directorio para el directorio '/'.

En general el superbloque contiene 3 tipos de información:

Parámetros fijos

Definidos durante la creación del sistema de ficheros, no pueden ser modificados posteriormente. Entre ellos encontramos:

- Tamaño de bloque.
- Número de inodos disponibles
- Número de bloques disponibles
- Número de bloques por grupo
- Número de inodos por grupo
- Sistema operativo que creó el sistema de archivos
- Número de revisión del sistema de ficheros
- Tamaño del fragmento
- Número de fragmentos por grupo

Parámetros modificables

En contraposición a los parámetros fijos, inmutables una vez que el sistema de ficheros ha sido creado, hay varios parámetros que pueden ser modificados durante el funcionamiento del mismo.

- Número de bloques reservados al superusuario
- UID (identificador de usuario) del superusuario por defecto
- GID (identificador de grupo) del superusuario por defecto

Estado del sistema de ficheros
Encontramos información de uso:

- Inodos libres
- Bloques libres
- Directorio donde se montó la última vez
- Estado del sistema de archivos
- Número de montajes
- Marca de tiempo del último montaje
- Marca de tiempo de la última escritura

- Marca de tiempo de la última comprobación

Econtramos información sobre actuaciones:
- Máximo número de montajes sin comprobación
- Máximo tiempo entre comprobaciones
- Comportamiento cuando se detectan errores

Manejo de errores

Ext2 no realiza manejo de errores, se limita a informar al núcleo del sistema operativo de la existencia de los mismos. La comprobación del sistema de ficheros y la corrección de errores es llevada a cabo por herramientas externas como e2fsch o debugfs de Teodore Ts'o, o EXT2ED de Gadi Oxman. Estos programas hacen uso de la información sobre errores y su tratamiento que se encuentra en el superbloque.

El bit 0 del campo s_state del superbloque se pone a 0 durante el montaje de una partición, asignandole un valor de 1 cuando el sistema de archivos es desmontado, de este modo si en el momento del montaje el bit 0 se encuentra a 0 puede deducirse que el sistema no se desmontó o se desmontó incorrectamente y actuar en consecuencia. El bit 1 de s_tate será puesto a 1 por el núcleo cuándo detecte algún error.

El campo s_errors define cual ha de ser el comportamiento en caso de detectar un error:
1, error ignorado
2, se remonta el sistema de archivos en modo de sólo lectura
3, se entrará en un “kernel panic”

Estructura ext2_super_bloq

En el archivo <include/linux/ext2_fs.h> se encuentra definida la estructura ext2_super_block.

```
370/*
371 * Structure of the super block
372 */
373struct ext2_super_block {
374        __le32  s_inodes_count;         /* Inodes count */
375        __le32  s_blocks_count;         /* Blocks count */
376        __le32  s_r_blocks_count;       /* Reserved blocks count */
377        __le32  s_free_blocks_count;    /* Free blocks count */
378        __le32  s_free_inodes_count;    /* Free inodes count */
379        __le32  s_first_data_block;     /* First Data Block */
380        __le32  s_log_block_size;       /* Block size */
381        __le32  s_log_frag_size;        /* Fragment size */
382        __le32  s_blocks_per_group;     /* # Blocks per group */
383        __le32  s_frags_per_group;      /* # Fragments per group */
384        __le32  s_inodes_per_group;     /* # Inodes per group */
385        __le32  s_mtime;                /* Mount time */
386        __le32  s_wtime;                /* Write time */
387        __le16  s_mnt_count;            /* Mount count */
388        __le16  s_max_mnt_count;        /* Maximal mount count */
389        __le16  s_magic;                /* Magic signature */
```

```
390       __le16  s_state;              /* File system state */
391       __le16  s_errors;             /* Behaviour when detecting errors */
392       __le16  s_minor_rev_level;    /* minor revision level */
393       __le32  s_lastcheck;          /* time of last check */
394       __le32  s_checkinterval;      /* max. time between checks */
395       __le32  s_creator_os;         /* OS */
396       __le32  s_rev_level;          /* Revision level */
397       __le16  s_def_resuid;         /* Default uid for reserved blocks */
398       __le16  s_def_resgid;         /* Default gid for reserved blocks */
399       /*
400        * These fields are for EXT2_DYNAMIC_REV superblocks only.
401        *
402        * Note: the difference between the compatible feature set and
403        * the incompatible feature set is that if there is a bit set
404        * in the incompatible feature set that the kernel doesn't
405        * know about, it should refuse to mount the filesystem.
406        *
407        * e2fsck's requirements are more strict; if it doesn't know
408        * about a feature in either the compatible or incompatible
409        * feature set, it must abort and not try to meddle with
410        * things it doesn't understand...
411        */
412       __le32  s_first_ino;          /* First non-reserved inode */
413       __le16   s_inode_size;        /* size of inode structure */
414       __le16  s_block_group_nr;     /* block group # of this superblock */
415       __le32  s_feature_compat;     /* compatible feature set */
416       __le32  s_feature_incompat;   /* incompatible feature set */
417       __le32  s_feature_ro_compat;  /* readonly-compatible feature set */
418       __u8    s_uuid[16];           /* 128-bit uuid for volume */
419       char    s_volume_name[16];    /* volume name */
420       char    s_last_mounted[64];   /* directory where last mounted */
421       __le32  s_algorithm_usage_bitmap; /* For compression */
422       /*
423        * Performance hints.  Directory preallocation should only
424        * happen if the EXT2_COMPAT_PREALLOC flag is on.
425        */
426       __u8    s_prealloc_blocks;    /* Nr of blocks to try to preallocate*/
427       __u8    s_prealloc_dir_blocks;  /* Nr to preallocate for dirs */
428       __u16   s_padding1;
429       /*
430        * Journaling support valid if EXT3_FEATURE_COMPAT_HAS_JOURNAL set.
431        */
432       __u8    s_journal_uuid[16];   /* uuid of journal superblock */
433       __u32   s_journal_inum;       /* inode number of journal file */
434       __u32   s_journal_dev;        /* device number of journal file */
435       __u32   s_last_orphan;        /* start of list of inodes to delete */
436       __u32   s_hash_seed[4];       /* HTREE hash seed */
437       __u8    s_def_hash_version;   /* Default hash version to use */
438       __u8    s_reserved_char_pad;
439       __u16   s_reserved_word_pad;
440       __le32  s_default_mount_opts;
441       __le32  s_first_meta_bg;      /* First metablock block group */
442       __u32   s_reserved[190];      /* Padding to the end of the block */
443};
```

Tipo	Campo	Descipción
unsigned int	s_inodes_count	Número de inodos
unsigned int	s_blocks_count	Número de bloques
unsigned int	s_r_blocks_count	Número de bloques reservados
unsigned int	s_free_blocks_count	Número de bloques libres
unsigned int	s_free_inodes_count	Número de inodos libres
unsigned int	s_first_data_block	Primer bloque de datos
unsigned int	s_log_block_size	Tamaño de bloque
unsigned int	s_log_frag_size	Tamaño de fragmento
unsigned int	s_blocks_per_group	Número de bloques por grupo
unsigned int	s_frags_per_group	Número de fragmentos por grupo
unsigned int	s_inodes_per_group	Número de inodos por grupo
unsigned int	s_mtime	Fecha del último montaje
unsigned int	s_wtime	Fecha de la última escritura
unsigned short	s_mnt_count	Número de montajes realizados desde la última comprobación
unsigned short	s_max_mnt_count	Máximo número de montajes entre comprobaciones
unsigned short	s_magic	Número mágico
unsigned short	s_state	Estado del sistema de archivos

Tipo	Campo	Descipción
unsigned short	s_errors	Comportamiento en caso de error
unsigned short	s_minor_rev_level	Número menor de revisión
unsigned int	s_last_check	Fecha de la úlitma coprobación
unsigned int	s_check_interval	Máximo intervalo de tiempo entre comprobaciones
unsigned int	s_creator_os	Sistema operativo que creó el FS
unsigned int	s_rev_level	Número de revisión
unsigned short	s_def_resuid	UID del superusuario por defecto
unsigned short	s_def_resuid	GID del superusuario por defecto
Los siguientes campos son para superbloques EXT2_DINAMIC_REV sólamente		
unsigned int	s_first_ino	Primer inodo no reservado
unsigned short	s_inode_size	Tamaño de la estructura inodo
unsigned short	s_block_group_nr	Número de grupo que contiene éste superbloque
unsigned int	s_feature_compat	Indicador de características compatibles
unsigned int	s_feature_incompat	Indicador de características incompatibles
unsigned int	s_feature_ro_compat	Compatibilidad con sólo lectura
unsigned char	s_uuid[16]	UID de 128 bits, para el volumen
char	s_volume_name[16]	Nombre del volumen

Tipo	Campo	Descipción
char	s_last_mounted`[64]	directorio sobre el que se montó la última vez
unsigned int	s_algorithm_usage_bitmap	Para compresión
Campos relacionado con el rendimiento. Preasignación de directorios únicamente debe realizarse si el flag EXT2_COMPAT_PREALLOC está activo.		
unsigned char	s_prealloc_blocks	Número de bloques que debe intentar preasignarse
unsigned char	s_prealloc_dir_blocks	Número de bloques a preasignar para directorios
unsigned short	s_padding1	Número de bloques de relleno
Campos relacionados con journaling, válidos si EXT3_FEATURE_COMPAT_HAS_JOURNAL está activo.		
unsigned char	s_journal_uuid[16]	UID del superbloque journal
unsigned int	s_journal_inum	Número de inodo del archivo journal
unsigned int	s_journal_dev	Número del dispositivo journal
unsigned int	s_last_orphan	Inicio de la lista de inodos a borrar
unsigned int	s_hash_seed[4]	Semilla hash de HTREE
unsigned char	s_def_hash_version	Versión por defecto de hash a usar
unsigned char	s_reserved_char_pad	Caracter de relleno por defecto
unsigned short	s_reserved_word_pad	Palabra de relleno por defecto
unsigned int	s_default_mount_opts	Opciones de montaje por defecto

Tipo	Campo	Descipción
unsigned int	s_first_meta_bg	Primer grupo de bloques
unsigned int	s_reserved[190]	Reserva de bloques

Estructura ext2_sb_info

Esta estructura representa la manera en la que un superbloque va a ser almacenado y utilizado internamente por el núcleo de Linux, en memoria interna. Se encuentra definido en el fichero linux/include/linux/ext2_fs_sb.h

```
68/*
69 * second extended-fs super-block data in memory
70 */
71struct ext2_sb_info {
72      unsigned long s_frag_size;      /* Size of a fragment in bytes */
73      unsigned long s_frags_per_block;/* Number of fragments per block */
74      unsigned long s_inodes_per_block;/* Number of inodes per block */
75      unsigned long s_frags_per_group;/* Number of fragments in a group */
76      unsigned long s_blocks_per_group;/* Number of blocks in a group */
77      unsigned long s_inodes_per_group;/* Number of inodes in a group */
78      unsigned long s_itb_per_group;  /* Number of inode table blocks per group */
79      unsigned long s_gdb_count;      /* Number of group descriptor blocks */
80      unsigned long s_desc_per_block; /* Number of group descriptors per block */
81      unsigned long s_groups_count;   /* Number of groups in the fs */
82      unsigned long s_overhead_last;  /* Last calculated overhead */
83      unsigned long s_blocks_last;    /* Last seen block count */
84      struct buffer_head * s_sbh;     /* Buffer containing the super block */
85      struct ext2_super_block * s_es; /* Pointer to the super block in the buffer */
86      struct buffer_head ** s_group_desc;
87      unsigned long  s_mount_opt;
88      unsigned long s_sb_block;
89      uid_t s_resuid;
90      gid_t s_resgid;
91      unsigned short s_mount_state;
92      unsigned short s_pad;
93      int s_addr_per_block_bits;
94      int s_desc_per_block_bits;
95      int s_inode_size;
96      int s_first_ino;
97      spinlock_t s_next_gen_lock;
98      u32 s_next_generation;
99      unsigned long s_dir_count;
100      u8 *s_debts;
101      struct percpu_counter s_freeblocks_counter;
102      struct percpu_counter s_freeinodes_counter;
103      struct percpu_counter s_dirs_counter;
104      struct blockgroup_lock *s_blockgroup_lock;
105      /* root of the per fs reservation window tree */
106      spinlock_t s_rsv_window_lock;
107      struct rb_root s_rsv_window_root;
```

```
108      struct ext2_reserve_window_node s_rsv_window_head;
109};
110
111static inline spinlock_t *
112sb_bgl_lock(struct ext2_sb_info *sbi, unsigned int block_group)
113{
114      return bgl_lock_ptr(sbi->s_blockgroup_lock, block_group);
115}
```

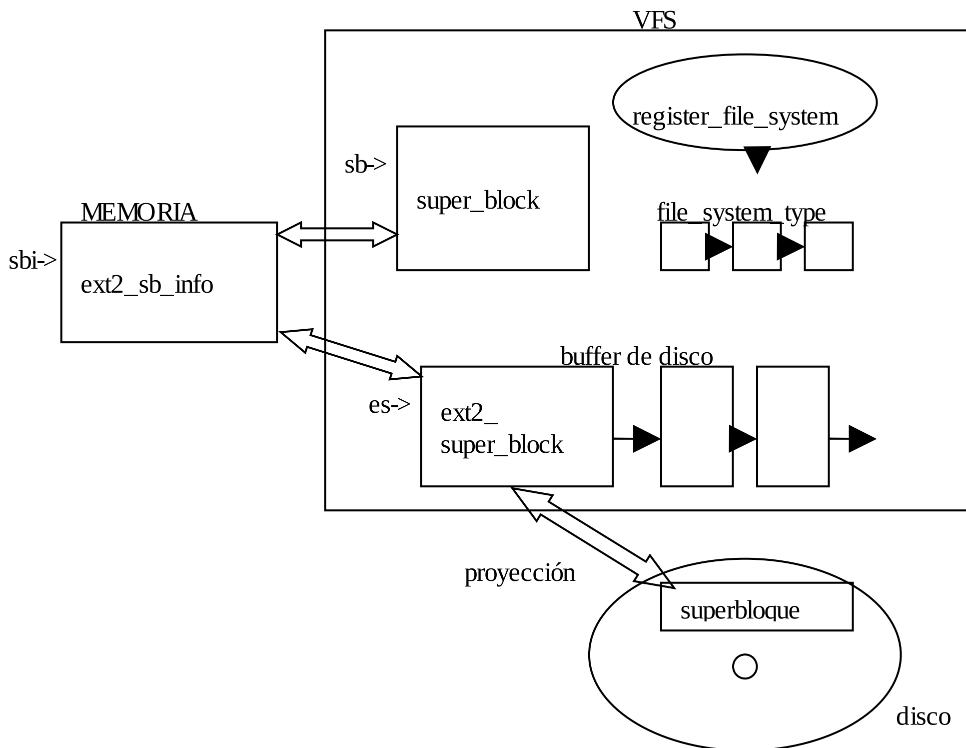
Tipo	Campo	Descipción
unsigned long	s_frag_size	Tamaño de un fragmento en bytes
unsigned long	s_frags_per_block	Número de fragmentos por bloque
unsigned long	s_inodes_per_block	Número de inodos por bloque
unsigned long	s_frags_per_group	Número de fragmentos en un grupo
unsigned long	s_blocks_per_group	Número de bloques en un grupo
unsigned long	s_inodes_per_group	Número de inodos en un grupo
unsigned long	s_itb_per_group	Nº bloques de la tabla de inodos por grupo
unsigned long	s_gdb_count	Nº bloques del descriptor de grupo
unsigned long	s_desc_per_block	Nº de descriptores de grupo por bloque
unsigned long	s_groups_count	Número de grupos en el fs
unsigned long	s_overhead_last	Último overhead calculado
unsigned long	s_blocks_last	Último bloque contado
struct buffer head *	s_sbh	Buffer que contiene el superbloque
struct ext2_super_block	s_es	Puntero al buffer superbloque

Tipo	Campo	Descipción
struct buffer head **	s_group_desc	Puntero al descriptor de grupo
unsinged long	s_mount_opt	Opciones de montaje
unsinged long	s_sb_block	Numero de bloque del superbloque
uid_t	s_resuid	Superusuario por defecto
gid_t	s_resgid	Grupo del superusuario por defecto
unsigned short	s_mount_state	Estado del montaje
unsigned short	s_pad	
int	s_addr_per_block_bits	
int	s_desc_per_block_bits	
int	s_inode_size	Tamaño de inodo
int	s_first_ino	Bloque del primer inodo
spinlock_t	s_next_gen_lock	
u32	s_next_generation	
unsigned long	s_dir_count	Número de directorios
u8 *	s_debts	
struct percpu_counter	s_freeblocks_counter	Número de bloques libres
struct percpu_counter	s_freeinodes_counter	Número de inodos libres
struct	s_dirs_counter	Número de directorios

Tipo	Campo	Descipción
percpu_counter		
struct blockgroup_lock	s_blockgroup_lock	
spinlock_t	s_rsv_window_lock	
struct rb_root	s_rsv_window_root	ventana reserva para root
struct ext2_reserve_window_node	s_rsv_window_head	principio de la reserva

Hemos visto tres estructuras relacionadas con el superbloque:

En el capitulo de sistema de ficheros virtual VFS, **super_block**, la variable que lo referencia es “**sb**”.
En este capitulo superbloque para el sistema de ficheros ext2, **ext2_super_block**, esta estructura se proyecta en el buffer de disco, la variable que lo referencia es “**es**”.
Estructura **ext2_sb_info**, superbloque ext2 en memoria, se referencia con la variable “**sbi**”.



Descriptores de grupo de bloques

(ext2_group_desc)

Los descriptores de grupo se colocan todos juntos, y forman la **tabla de descriptores de grupo**, de la cual se almacena una copia en cada BG, justo detrás de la copia del superbloque. De todas estas copias, sólo se usa la del BG 0. Las demás sólo se emplearán en caso de que el sistema resulte dañado.
Estos descriptores contienen información para gestionar los bloques y los inodos presentes en cada grupo.

En el fichero de cabecera include/linux/ext2_fs.h se encuentra la definición de la estructura ext2_group_desc, que define la tabla de descriptores de grupo.

```
struct ext2_group_desc
 {
        __le32  bg_block_bitmap;          /* Dirección del Bloque del bitmap de bloques */
        __le32  bg_inode_bitmap;          /*Dirección del  Bloque del bitmap de inodos  */
        __le32  bg_inode_table;           /* Dirección del Bloque de la tabla de inodos */
        __le16  bg_free_blocks_count;  /* número  de bloques libres */
        __le16  bg_free_inodes_count;  /* número de inodos libres */
        __le16  bg_used_dirs_count;     /* número de directorios */
```

```
        __le16  bg_pad;                   /* no utilizado */
        __le32  bg_reserved[3];           /* reservado para futura extensión */
};
```

Tipo	Campo	Descripción
unsigned int	bg_block_bitmap	Bloque de inicio del mapa de bloques
unsigned int	bg_inode_bitmap	Bloque de inicio del mapa de inodos
unsigned int	bg_inode_table	Bloque de inicio de la tabla de inodos
unsigned short	bg_free_blocks_count	Número de bloques libres
unsigned short	bg_free_inodes_count	Número de inodos libres
unsigned short	bg_use_dirs_count	Número de directorios
unsigned short	bg_pad;	
unsigned int	bg_reserved[3]	Reservado

Todos estos campos son inicializados el momento de creación del sistema de ficheros y posteriormente actualizados por el sistema de ficheros.

bg_block_bitmap y **bg_inode_bitmap** contienen el número de bloque del bitmap de asignación de bloques y del bitmap de asignación de inodos respectivamente, que son empleado para asignar y desasignar cada bloque o inodo en el Grupo de Bloques.

El sistema de ficheros se vale del **bitmap de asignación de bloques** de cada grupo para guardar la información relativa a la asignación de bloques dentro del BG. Cada bit en el bitmap de bloques representa un bloque del BG y si está libre (bit a 0) o no (bit a 1).

bg_inode_table contiene el número del primer bloque donde comienza la tabla de inodos del Grupo de Bloques actual.

bg_free_blocks_count, **bg_free_inodes_count** y **bg_used_dirs_count** mantienen información estadística sobre el uso de estos recursos en el Grupo de Bloques y son utilizados por el núcleo para equilibrar la carga entre los diferentes grupos de bloques.

Cada grupo de bloques tiene una copia del superbloque así como una copia de los descriptores de grupos. Estos descriptores contienen las coordenadas de las estructuras de control presentes en cada Grupo. Como el superbloque, todos los descriptores de grupo para todos los grupos de bloques se duplican en cada grupo, como medida de seguridad ante posibles errores.

Mapas de bits

El sistema Ext2 emplea, para localizar bloques e inodos libres, mapas de bits. De este modo si una posición contiene el valor 1, entonces el inodo o bloque correspondiente se encuentra libre, en caso contrario se interpreta que está ocupado.

17.5 Operaciones vinculadas al superbloque

En el capitulo de sistema de fichero virtual vimos la estructura super_operations (linux/include/linux/fs.h) que contiene punteros a funciones que trabajan con la estructura "*super_block*", que se rellenará con la funciones propias de cada sistema de ficheros real.

```
struct super_operations {
        /*Lee un inode de disco de un sistema de ficheros*/
        struct inode *(*alloc_inode)(struct super_block *sb);
        void (*destroy_inode)(struct inode *);
        void (*dirty_inode) (struct inode *);
/*escribe el contenido de un inode del VFS en el inode del disco*/
        void (*write_inode) (struct inode *, int);
/*libera la memoria usada por un inode cuando deja de utilizarse*/
         void (*drop_inode) (struct inode *);
/*devuelve un inode puesto a cero*/
        void (*delete_inode) (struct inode *);
/*libera la memoria usada por superbloque cuando el fs se desmonta*/
        void (*put_super) (struct super_block *);
/*escribe el contenido del superbloque del VFS en disco*/
        void (*write_super) (struct super_block *);
        int (*sync_fs)(struct super_block *sb, int wait);
        int (*freeze_fs) (struct super_block *);
        int (*unfreeze_fs) (struct super_block *);
/*el VFS lee información de control del sistema de ficheros*/
        int (*statfs) (struct super_block *, struct statfs *);
/*se varían las condiciones de montaje*/
       int (*remount_fs) (struct super_block *, int *, char *);
       void (*clear_inode) (struct inode *);
       void (*umount_begin) (struct super_block *);
        int (*show_options)(struct seq_file *, struct vfsmount *);
```

```
         int (*show_stats)(struct seq_file *, struct vfsmount *);
#ifdef CONFIG_QUOTA
        ssize_t (*quota_read)(struct super_block *, int, char *, size_t, loff_t);
        ssize_t (*quota_write)(struct super_block *, int, const char *, size_t, loff_t);
#endif
};
```

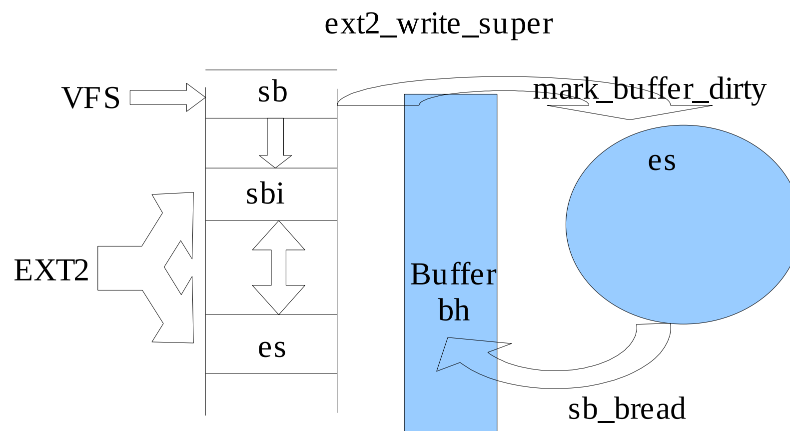
Las operaciones sobre el superbloque para EXT2 se implementan en el archivo fuente fs/ext2/super.c.

El VFS realiza la correspondencia entre las funciones llamadas por procesos y la específica para ese SF. P. ej.: ext2_write_inode -> write_inode
El siguiente array de punteros a funciones, indica esta correspondencia

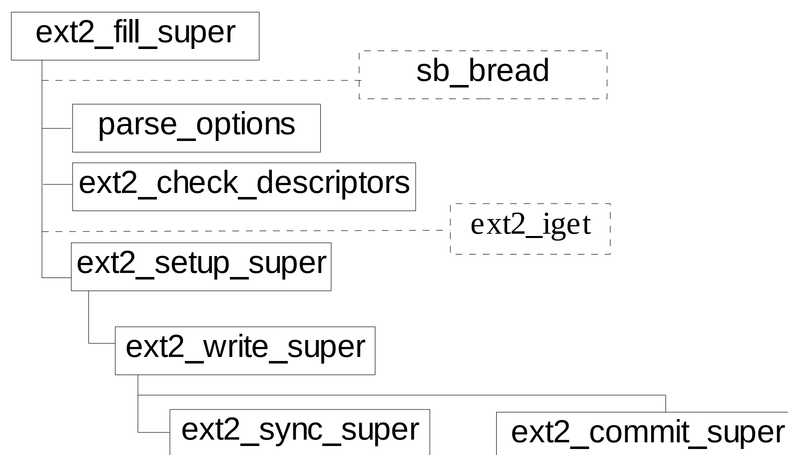
```
static struct super_operations ext2_sops = {
        .alloc_inode    = ext2_alloc_inode,
        .destroy_inode  = ext2_destroy_inode,
        .write_inode    = ext2_write_inode,
        .delete_inode   = ext2_delete_inode,

        .put_super      = ext2_put_super,
        .write_super    = ext2_write_super,
        .statfs         = ext2_statfs,
        .remount_fs     = ext2_remount,
        .clear_inode    = ext2_clear_inode,
    .show_options       = ext2_show_options,
#ifdef CONFIG_QUOTA
        .quota_read     = ext2_quota_read,
        .quota_write    = ext2_quota_write,
#endif
};
```

Las funciones principales relacionadas con el superbloque de un sistema de ficheros ext2 las podemos encontrar en el fichero super.c, vamos a estudiar alguna de ellas relacionadas con leer y escribir el superbloque desde el disco al superbloque del sistema de ficheros virtual en memoria.



Siguiendo la filosofia de Linux, encontramos que unas funciones llaman a otras para realizar funciones mas concretas. Podemos ver las siguientes dependencias:



ext2_error

· ¿Qué hace?

Imprime un mensaje de error, mediante la llamada la función printk, durante el montaje de un sistema de archivos y, en función del comportamiento definido en el superbloque, aborta el montaje, o monta en modo sólo lectura.

Código:

```
  45void ext2_error (struct super_block * sb, const char * function,
```

```
  46                    const char * fmt, ...)
  47{
  48          va_list args;
  49          struct ext2_sb_info *sbi = EXT2_SB(sb);
  50          struct ext2_super_block *es = sbi->s_es;
  51
  52          if (!(sb->s_flags & MS_RDONLY)) {
  53                  sbi->s_mount_state |= EXT2_ERROR_FS;
  54                  es->s_state =
  55                          cpu_to_le16(le16_to_cpu(es->s_state) |
EXT2_ERROR_FS);
  56                  ext2_sync_super(sb, es);
  57          }
  58
  59          va_start(args, fmt);
  60          printk(KERN_CRIT "EXT2-fs error (device %s): %s: ",sb->s_id,
function);
  61          vprintk(fmt, args);
  62          printk("\n");
  63          va_end(args);
  64
  65          if (test_opt(sb, ERRORS_PANIC))
  66                  panic("EXT2-fs panic from previous error\n");
  67          if (test_opt(sb, ERRORS_RO)) {
  68                  printk("Remounting filesystem read-only\n");
  69                  sb->s_flags |= MS_RDONLY;
  70          }
  71}
```

ext2_warning

· *¿Qué hace?*

Imprime un aviso (la operación se ha llevado a cabo, pero hay aspectos que tener en cuenta) mediante la llamada la función printk.

Código:

```
  73void ext2_warning (struct super_block * sb, const char * function,
  74                      const char * fmt, ...)
  75{
  76          va_list args;
  77
  78          va_start(args, fmt);
  79          printk(KERN_WARNING "EXT2-fs warning (device %s): %s: ",
  80                 sb->s_id, function);
  81          vprintk(fmt, args);
  82          printk("\n");
  83          va_end(args);
  84}
```

ext2_put_super

· *¿Qué hace?*

Implementa la operación sobre sistema de archivos *put_super*. Es llamada por el VFS cuando un sistema de archivos se desmonta. Guarda el superbloque ext2 (es) en disco en caso de haberse modificado, para posteriormente eliminar las memorias ocupadas por los descriptores de grupos de bloques y la memoria intermedia del superbloque (sbi).

Comprueba en primer lugar si se ha modificado el superbloque en disco, en cuyo caso se ha de guardar.

Libera las memorias intermedias que contienen los descriptores del sistema de archivos, llamando a la función *brelse.*

Libera los punteros a esas memorias llamando a *kfree_s*.

Libera las memorias intermedias asociadas a los bloques de *bitmap* cargados en memoria.

Libera la memoria intermedia sbi que contiene el superbloque del sistema de archivos.

Código:

```
 111static void ext2_put_super (struct super_block * sb)
 112{
 113        int db_count;
 114        int i;
 115        struct ext2_sb_info *sbi = EXT2_SB(sb);
 116
 117        ext2_xattr_put_super(sb);
//Se comprueba si hay que actualizar el SB en disco
 118        if (!(sb->s_flags & MS_RDONLY)) {
//Se actualiza 'es' con la memoria intermedia sbi
 119                struct ext2_super_block *es = sbi->s_es;
 120
 121                es->s_state = cpu_to_le16(sbi->s_mount_state);
//Se actualiza en disco
 122                ext2_sync_super(sb, es);
 123        }
//Se liberan las memorias que contienen los descriptores de los grupos
//de bloques mediante la función brelse
 124        db_count = sbi->s_gdb_count;
 125        for (i = 0; i < db_count; i++)
 126                if (sbi->s_group_desc[i])
 127                        brelse (sbi->s_group_desc[i]);
//Se liberan los punteros a esas memorias
 128        kfree(sbi->s_group_desc);
 129        kfree(sbi->s_debts);
//Se liberan las memorias asociadas a los bitmaps de bloques
 130        percpu_counter_destroy(&sbi->s_freeblocks_counter);
//Se liberan las memorias asociadas a los bitmaps de inodos
 131        percpu_counter_destroy(&sbi->s_freeinodes_counter);
 132        percpu_counter_destroy(&sbi->s_dirs_counter);
//Se libera la memoria intermedia sbi que contiene el superbloque
```

```
 133        brelse (sbi->s_sbh);
 134        sb->s_fs_info = NULL;
 135        kfree(sbi);
 136
 137        return;
 138}
```

ext2_parse_options

· *¿Qué hace?*

Realiza un análisis de las opciones de montaje especificadas. Analiza la cadena de caracteres pasada como parámetro e inicializa las opciones de montaje.

Comprueba si se han pasado opciones de montaje. Si no se le pasa ninguna opción se sale de la función.

Se comprueba cada una de las opciones pasadas como parámetro y va inicializando las opciones de montaje.

La función devuelve 1 si es correcto, 0 en caso contrario.

Código:

```
 430static int parse_options (char * options,
 431                          struct ext2_sb_info *sbi)
 432{
 433        char * p;
 434        substring_t args[MAX_OPT_ARGS];
 435        int option;
 436    //Se comprueba si se han pasado opciones de montaje
 437        if (!options)
 438                return 1;
 439
 440        while ((p = strsep (&options, ",")) != NULL) {
 441                int token;
 442                if (!*p)
 443                        continue;
 444
 445                token = match_token(p, tokens, args);
 446                switch (token) {
 447                case Opt_bsd_df:
 448                        clear_opt (sbi->s_mount_opt, MINIX_DF);
 449                        break;
 450                case Opt_minix_df:
 451                        set_opt (sbi->s_mount_opt, MINIX_DF);
 452                        break;
 453                case Opt_grpid:
 454                        set_opt (sbi->s_mount_opt, GRPID);
 455                        break;
 456                case Opt_nogrpid:
 457                        clear_opt (sbi->s_mount_opt, GRPID);
```

```
 458                    break;
 459            case Opt_resuid:
 460                    if (match_int(&args[0], &option))
 461                            return 0;
 462                    sbi->s_resuid = option;
 463                    break;
 464            case Opt_resgid:
 465                    if (match_int(&args[0], &option))
 466                            return 0;
 467                    sbi->s_resgid = option;
 468                    break;
 469            case Opt_sb:
 470                    /* handled by get_sb_block() instead of here */
 471                    /* *sb_block = match_int(&args[0]); */
 472                    break;
 473            case Opt_err_panic:
 474                    clear_opt (sbi->s_mount_opt, ERRORS_CONT);
 475                    clear_opt (sbi->s_mount_opt, ERRORS_RO);
 476                    set_opt (sbi->s_mount_opt, ERRORS_PANIC);
 477                    break;
 478            case Opt_err_ro:
 479                    clear_opt (sbi->s_mount_opt, ERRORS_CONT);
 480                    clear_opt (sbi->s_mount_opt, ERRORS_PANIC);
 481                    set_opt (sbi->s_mount_opt, ERRORS_RO);
 482                    break;
 483            case Opt_err_cont:
 484                    clear_opt (sbi->s_mount_opt, ERRORS_RO);
 485                    clear_opt (sbi->s_mount_opt, ERRORS_PANIC);
 486                    set_opt (sbi->s_mount_opt, ERRORS_CONT);
 487                    break;
 488            case Opt_nouid32:
 489                    set_opt (sbi->s_mount_opt, NO_UID32);
 490                    break;
 491            case Opt_nocheck:
 492                    clear_opt (sbi->s_mount_opt, CHECK);
 493                    break;
 494            case Opt_debug:
 495                    set_opt (sbi->s_mount_opt, DEBUG);
 496                    break;
 497            case Opt_oldalloc:
 498                    set_opt (sbi->s_mount_opt, OLDALLOC);
 499                    break;
 500            case Opt_orlov:
 501                    clear_opt (sbi->s_mount_opt, OLDALLOC);
 502                    break;
 503            case Opt_nobh:
 504                    set_opt (sbi->s_mount_opt, NOBH);
 505                    break;
 506#ifdef CONFIG_EXT2_FS_XATTR
 507            case Opt_user_xattr:
 508                    set_opt (sbi->s_mount_opt, XATTR_USER);
 509                    break;
 510            case Opt_nouser_xattr:
 511                    clear_opt (sbi->s_mount_opt, XATTR_USER);
 512                    break;
 513#else
 514            case Opt_user_xattr:
 515            case Opt_nouser_xattr:
```

```
 516                            printk("EXT2 (no)user_xattr options not
supported\n");
 517                            break;
 518#endif
 519#ifdef CONFIG_EXT2_FS_POSIX_ACL
 520                    case Opt_acl:
 521                            set_opt(sbi->s_mount_opt, POSIX_ACL);
 522                            break;
 523                    case Opt_noacl:
 524                            clear_opt(sbi->s_mount_opt, POSIX_ACL);
 525                            break;
 526#else
 527                    case Opt_acl:
 528                    case Opt_noacl:
 529                            printk("EXT2 (no)acl options not supported\n");
 530                            break;
 531#endif
 532                    case Opt_xip:
 533#ifdef CONFIG_EXT2_FS_XIP
 534                            set_opt (sbi->s_mount_opt, XIP);
 535#else
 536                            printk("EXT2 xip option not supported\n");
 537#endif
 538                            break;
 539
 540#if defined(CONFIG_QUOTA)
 541                    case Opt_quota:
 542                    case Opt_usrquota:
 543                            set_opt(sbi->s_mount_opt, USRQUOTA);
 544                            break;
 545
 546                    case Opt_grpquota:
 547                            set_opt(sbi->s_mount_opt, GRPQUOTA);
 548                            break;
 549#else
 550                    case Opt_quota:
 551                    case Opt_usrquota:
 552                    case Opt_grpquota:
 553                            printk(KERN_ERR
 554                                    "EXT2-fs: quota operations not
supported.\n");
 555
 556                            break;
 557#endif
 558
 559                    case Opt_reservation:
 560                            set_opt(sbi->s_mount_opt, RESERVATION);
 561                            printk("reservations ON\n");
 562                            break;
 563                    case Opt_noreservation:
 564                            clear_opt(sbi->s_mount_opt, RESERVATION);
 565                            printk("reservations OFF\n");
 566                            break;
 567                    case Opt_ignore:
 568                            break;
 569                    default:
 570                            return 0;
 571                    }
 572            }
```

```
 573        return 1;
 574}
 575
```

ext2_setup_super

· ¿Qué hace?

Esta función actualiza la estructura superbloque de ext2 (es) incrementado el número de montajes y llama a ext2_write_super para actualizar campos de (es) a partir de (sb) y escribir en el disco. Se llama cuando se monta el sistema de ficheros.

Almacena el "sb" en la memoria intermedia "sbi".

Comprueba que la versión del super bloque no es mayor que la máxima permitida, en cuyo caso se producirá un error.

Si el sistema de fichero se monto de solo lectura se sale de la función, no tiene sentido escribir en disco.

En caso de no haberse montado correctamente el sistema de ficheros la última vez, se mostrará un mensaje con printk y se recomienda ejecutar e2fsck. Se comprueba si el sistema de ficheros ha sido testeado, si tiene errores, si se ha superado el número máximo de montajes, si se ha superado el tiempo de revisión.

Incrementa el número de veces que se ha montado el sistema de ficheros.

Llama a la función ext2_write_super que se encargará de actualizar los campos de "es" a partir de "sb" y posterior escitura en disco.

Código:

```
 576static int ext2_setup_super (struct super_block * sb,
 577                              struct ext2_super_block * es,
 578                              int read_only)
 579{
 580        int res = 0;
//Se almacena el "sb" en la memoria intermedia sbi
 581        struct ext2_sb_info *sbi = EXT2_SB(sb);
 582//Si la versión de SB es mayor que la máxima permitida --> ERROR
 583        if (le32_to_cpu(es->s_rev_level) > EXT2_MAX_SUPP_REV) {
 584                printk ("EXT2-fs warning: revision level too high, "
 585                        "forcing read-only mode\n");
 586                res = MS_RDONLY;
 587        }//Si es de solo lectura
 588        if (read_only)
```

```
 589                return res;
//Se comprueba que el sistema se montó correctamente la última vez
 590        if (!(sbi->s_mount_state & EXT2_VALID_FS))
 591                printk ("EXT2-fs warning: mounting unchecked fs, "
 592                        "running e2fsck is recommended\n");
 593        else if ((sbi->s_mount_state & EXT2_ERROR_FS))
 594                printk ("EXT2-fs warning: mounting fs with errors, "
 595                        "running e2fsck is recommended\n");
 596        else if ((__s16) le16_to_cpu(es->s_max_mnt_count) >= 0 &&
 597                 le16_to_cpu(es->s_mnt_count) >=
 598                 (unsigned short) (__s16) le16_to_cpu(es-
>s_max_mnt_count))
 599                printk ("EXT2-fs warning: maximal mount count reached,
"
 600                        "running e2fsck is recommended\n");
 601        else if (le32_to_cpu(es->s_checkinterval) &&
 602                (le32_to_cpu(es->s_lastcheck) + le32_to_cpu(es-
>s_checkinterval) <= get_seconds()))
 603                printk ("EXT2-fs warning: checktime reached, "
 604                        "running e2fsck is recommended\n");
 605        if (!le16_to_cpu(es->s_max_mnt_count))
//Se incrementa el número de veces que se ha montado el FS
 606                es->s_max_mnt_count =
cpu_to_le16(EXT2_DFL_MAX_MNT_COUNT);
 607        es->s_mnt_count=cpu_to_le16(le16_to_cpu(es->s_mnt_count) + 1);
//Actualiza los campos de "es" a partir de "sb" y escribe en disco.
 608        ext2_write_super(sb);
//Opciones de depuración y chequeo
 609        if (test_opt (sb, DEBUG))
 610                printk ("[EXT II FS %s, %s, bs=%lu, fs=%lu, gc=%lu, "
 611                        "bpg=%lu, ipg=%lu, mo=%04lx]\n",
 612                        EXT2FS_VERSION, EXT2FS_DATE, sb->s_blocksize,
 613                        sbi->s_frag_size,
 614                        sbi->s_groups_count,
 615                        EXT2_BLOCKS_PER_GROUP(sb),
 616                        EXT2_INODES_PER_GROUP(sb),
 617                        sbi->s_mount_opt);
 618        return res;
 619}
```

ext2_check_descriptors

· ¿Qué hace?

Verifica la validez de los descriptores de conjuntos leídos desde el disco. Para cada descriptor, verifica que los bloques de bitmap y la tabla de inodos están contenidos en el grupo.

Comprueba si se han pasado opciones de montaje. Si no se le pasa ninguna opción se sale de la función.

Comprueba cada una de las opciones pasadas como parámetro y va inicializando las opciones de montaje.

La función devuelve 1 si es correcto, 0 en caso contrario.

Código:

```
 621static int ext2_check_descriptors(struct super_block *sb)
 622{
 623         int i;
 624         struct ext2_sb_info *sbi = EXT2_SB(sb);
 625         unsigned long first_block = le32_to_cpu(sbi->s_es-
>s_first_data_block);
 626         unsigned long last_block;
 627
 628         ext2_debug ("Checking group descriptors");
 629//Se van recorriendo todos los descriptores de grupo
 630         for (i = 0; i < sbi->s_groups_count; i++) {
 631                 struct ext2_group_desc *gdp = ext2_get_group_desc(sb,
i, NULL);
 632//Se selecciona un descriptor de Grupo de Bloques
 633                 if (i == sbi->s_groups_count - 1)
 634                         last_block = le32_to_cpu(sbi->s_es-
>s_blocks_count) - 1;
 635                 else
 636                         last_block = first_block +
 637                                 (EXT2_BLOCKS_PER_GROUP(sb) - 1);
 638//Si bitmap de bloque no está contenido en el grupo --> ERROR
 639                 if (le32_to_cpu(gdp->bg_block_bitmap) < first_block ||
 640                     le32_to_cpu(gdp->bg_block_bitmap) > last_block)
 641                 {
 642                         ext2_error (sb, "ext2_check_descriptors",
 643                                     "Block bitmap for group %d"
 644                                     " not in group (block %lu)!",
 645                                     i, (unsigned long) le32_to_cpu(gdp-
>bg_block_bitmap));
 646                         return 0;
 647                 }
//Si bitmap de inodo no está contenido en el grupo --> ERROR
 648                 if (le32_to_cpu(gdp->bg_inode_bitmap) < first_block ||
 649                     le32_to_cpu(gdp->bg_inode_bitmap) > last_block)
 650                 {
 651                         ext2_error (sb, "ext2_check_descriptors",
 652                                     "Inode bitmap for group %d"
 653                                     " not in group (block %lu)!",
 654                                     i, (unsigned long) le32_to_cpu(gdp-
>bg_inode_bitmap));
 655                         return 0;
 656                 }
//Si tabla de inodo no está contenido en el grupo --> ERROR
 657                 if (le32_to_cpu(gdp->bg_inode_table) < first_block ||
 658                     le32_to_cpu(gdp->bg_inode_table) + sbi-
>s_itb_per_group - 1 >
 659                     last_block)
 660                 {
 661                         ext2_error (sb, "ext2_check_descriptors",
 662                                     "Inode table for group %d"
 663                                     " not in group (block %lu)!",
 664                                     i, (unsigned long) le32_to_cpu(gdp-
>bg_inode_table));
 665                         return 0;
```

```
 666                }
 667                first_block += EXT2_BLOCKS_PER_GROUP(sb);
 668        }//Si no se produjo ningún error se devuelve 1
 669        return 1;
 670}
 671
```

ext2_fill_super

· *¿Qué hace?*

Lee del disco el superbloque del sistema de ficheros a montar. Se llama cuando se monta un sistema de ficheros.

Se lee el superbloque desde el disco, se guarda en memoria (en buffer bh, en es y en sbi).

Comprueba su validez mediante el número mágico (*s_magic*).

Comprueba las opciones de montaje con la función *parse_options*.

Comprueba el tamaño de bloque, y en caso de ser incorrecto, se libera y se lee de nuevo el superbloque de disco y se almacena en bh, es y sbi.

Se inicializan campos del superbloque sbi, y se hace una comprobación de que se vaya a montar un sistema de ficheros ext2 mediante su número mágico.

Comprueba que los siguientes campos son correctos: Tamaño de bloque, tamaño de fragmento de bloques por grupo, fragmentos por grupo e inodos por grupo.

Lee de disco, se chequean y almacena en sbi los campos de descriptores de grupo.

Rellena la estructura sb->s_op con las operaciones propias del sitema de ficheros ext2 leidas del disco.

Realiza el mount, grabando el directorio raiz del nuevo sistema de ficheros en sb_root.

Actualiza el superbloque en disco.

Código:

```
 739static int ext2_fill_super(struct super_block *sb, void *data, int
silent)
 740{//bh es un puntero al buffer que contiene el ext2_superblock
 741        struct buffer_head * bh;
 742        struct ext2_sb_info * sbi;
 743        struct ext2_super_block * es;
 744        struct inode *root;
 745        unsigned long block;
```

```
 746         unsigned long sb_block = get_sb_block(&data);
 747         unsigned long logic_sb_block;
 748         unsigned long offset = 0;
 749         unsigned long def_mount_opts;
 750         long ret = -EINVAL;
 751         int blocksize = BLOCK_SIZE;
 752         int db_count;
 753         int i, j;
 754         __le32 features;
 755         int err;
 756//Se obtiene memoria para el "sbi"
 757         sbi = kzalloc(sizeof(*sbi), GFP_KERNEL);
 758         if (!sbi)
 759                 return -ENOMEM;
 760         sb->s_fs_info = sbi;
 761         sbi->s_sb_block = sb_block;
 762 
 763         /*
 764          * Se calcula el tamaño de bloque (blocksize) para el
 765          * dispositivo, y  se utiliza como el blocksize. De lo
 766          * contrario, (o si el blocksize es menor que el de por
 767          * defecto) se usa el de por defecto. Esto es importante
 768          * para dispositivos que tienen un tamaño de sector hardware
 769          */ que es mayor que el valor por defecto
 770         blocksize = sb_min_blocksize(sb, BLOCK_SIZE);
 771         if (!blocksize) {
 772                 printk ("EXT2-fs: unable to set blocksize\n");
 773                 goto failed_sbi;
 774         }
 775 
 776         /*
 777          * Si el superbloque no comienza en el principio de un
 778          * sector hardware, calcula el desplazamiento
 779          */
 780         if (blocksize != BLOCK_SIZE) {
 781                 logic_sb_block = (sb_block*BLOCK_SIZE) / blocksize;
 782                 offset = (sb_block*BLOCK_SIZE) % blocksize;
 783         } else {
 784                 logic_sb_block = sb_block;
 785         }
 786//Se lee y almacena el superbloque en una estructura buffer_head "bh"
 787         if (!(bh = sb_bread(sb, logic_sb_block))) {
 788                 printk ("EXT2-fs: unable to read superblock\n");
 789                 goto failed_sbi;
 790         }
 791         /* Nota: s_es debe inicializarse tan pronto como sea posible
 792          * ya que algunas macro-instructiones de ext2 dependen de su
 793          * valor  "es" apunta al buffer bh corregido con el offset que
 794          */ contiene el superbloque ext2 leido del disco
 795         es = (struct ext2_super_block *) (((char *)bh->b_data) +
offset);
//Se almacena el superbloque ext2 "es" en memoria intermedia "sbi"
 796         sbi->s_es = es;
 797         sb->s_magic = le16_to_cpu(es->s_magic);
 798//Se verifica la validez del SB comparándolo con el número mágico
 799         if (sb->s_magic != EXT2_SUPER_MAGIC)
 800                 goto cantfind_ext2;
 801
```

```
 802         /* Se analizan las opciones del mount  y se colocan los valores
por defecto en "sbi"
 803         def_mount_opts = le32_to_cpu(es->s_default_mount_opts);
 804         if (def_mount_opts & EXT2_DEFM_DEBUG)
 805                 set_opt(sbi->s_mount_opt, DEBUG);
 806         if (def_mount_opts & EXT2_DEFM_BSDGROUPS)
 807                 set_opt(sbi->s_mount_opt, GRPID);
 808         if (def_mount_opts & EXT2_DEFM_UID16)
 809                 set_opt(sbi->s_mount_opt, NO_UID32);
 810#ifdef CONFIG_EXT2_FS_XATTR
 811         if (def_mount_opts & EXT2_DEFM_XATTR_USER)
 812                 set_opt(sbi->s_mount_opt, XATTR_USER);
 813#endif
 814#ifdef CONFIG_EXT2_FS_POSIX_ACL
 815         if (def_mount_opts & EXT2_DEFM_ACL)
 816                 set_opt(sbi->s_mount_opt, POSIX_ACL);
 817#endif
 818
 819         if (le16_to_cpu(sbi->s_es->s_errors) == EXT2_ERRORS_PANIC)
 820                 set_opt(sbi->s_mount_opt, ERRORS_PANIC);
 821         else if (le16_to_cpu(sbi->s_es->s_errors) ==
EXT2_ERRORS_CONTINUE)
 822                 set_opt(sbi->s_mount_opt, ERRORS_CONT);
 823         else
 824                 set_opt(sbi->s_mount_opt, ERRORS_RO);
 825
 826         sbi->s_resuid = le16_to_cpu(es->s_def_resuid);
 827         sbi->s_resgid = le16_to_cpu(es->s_def_resgid);
 828
 829         set_opt(sbi->s_mount_opt, RESERVATION);
 830//Se analizan las opciones del montaje con la función parse_options
 831         if (!parse_options ((char *) data, sbi))
 832                 goto failed_mount;
 833
 834         sb->s_flags = (sb->s_flags & ~MS_POSIXACL) |
 835                 ((EXT2_SB(sb)->s_mount_opt & EXT2_MOUNT_POSIX_ACL) ?
 836                  MS_POSIXACL : 0);
 837
 838         ext2_xip_verify_sb(sb); /* see if bdev supports xip, unset
 839                                    EXT2_MOUNT_XIP if not */
 840//Verificaciones de otras características generales
 841         if (le32_to_cpu(es->s_rev_level) == EXT2_GOOD_OLD_REV &&
 842             (EXT2_HAS_COMPAT_FEATURE(sb, ~0U) ||
 843              EXT2_HAS_RO_COMPAT_FEATURE(sb, ~0U) ||
 844              EXT2_HAS_INCOMPAT_FEATURE(sb, ~0U)))
 845                 printk("EXT2-fs warning: feature flags set on rev 0 fs,
"
 846                        "running e2fsck is recommended\n");
 847         /*
 848          * Check feature flags regardless of the revision level, since
we
 849          * previously didn't change the revision level when setting the
flags,
 850          * so there is a chance incompat flags are set on a rev 0
filesystem.
 851          */
 852         features = EXT2_HAS_INCOMPAT_FEATURE(sb,
~EXT2_FEATURE_INCOMPAT_SUPP);
 853         if (features) {
```

```
 854                 printk("EXT2-fs: %s: couldn't mount because of "
 855                        "unsupported optional features (%x).\n",
 856                        sb->s_id, le32_to_cpu(features));
 857                 goto failed_mount;
 858         }
 859         if (!(sb->s_flags & MS_RDONLY) &&
 860             (features = EXT2_HAS_RO_COMPAT_FEATURE(sb,
~EXT2_FEATURE_RO_COMPAT_SUPP))){
 861                 printk("EXT2-fs: %s: couldn't mount RDWR because of "
 862                        "unsupported optional features (%x).\n",
 863                        sb->s_id, le32_to_cpu(features));
 864                 goto failed_mount;
 865         }
 866//Se inicializa el tamaño de bloque
//Se realiza la correspondencia 0->1kB; 1->2kB; 2->4kB
 867         blocksize = BLOCK_SIZE << le32_to_cpu(sbi->s_es-
>s_log_block_size);
 868
 869         if (ext2_use_xip(sb) && blocksize != PAGE_SIZE) {
 870                 if (!silent)
 871                         printk("XIP: Unsupported blocksize\n");
 872                 goto failed_mount;
 873         }
 874
 875         /* Se comprueba que el tamaño del blocksize sea correcto
//Si no se libera memoria y se vuelve a leer el SB
 876         if (sb->s_blocksize != blocksize) {
//Se liberan las memorias buffer que contienen los descriptores
        //del sistema de ficheros
 877                 brelse(bh);
 878
 879                 if (!sb_set_blocksize(sb, blocksize)) {
 880                         printk(KERN_ERR "EXT2-fs: blocksize too small
for device.\n");
 881                         goto failed_sbi;
 882                 }
 883
 884                 logic_sb_block = (sb_block*BLOCK_SIZE) / blocksize;
 885                 offset = (sb_block*BLOCK_SIZE) % blocksize;
 886                 bh = sb_bread(sb, logic_sb_block);
 887                 if(!bh) {
 888                         printk("EXT2-fs: Couldn't read superblock on "
 889                                "2nd try.\n");
 890                         goto failed_sbi;
 891                 }
//Se vuelve a asignar el superbloque a las variables "es" y "sbi"
 892                 es = (struct ext2_super_block *) (((char *)bh->b_data)
+ offset);
 893                 sbi->s_es = es;
//Se comprueba la validez del superbloque
 894                 if (es->s_magic != cpu_to_le16(EXT2_SUPER_MAGIC)) {
 895                         printk ("EXT2-fs: Magic mismatch, very
weird !\n");
 896                         goto failed_mount;
 897                 }
 898         }
 899
 900         sb->s_maxbytes = ext2_max_size(sb->s_blocksize_bits);
 901//Comprobaciones acerca de los niveles de revisión
```

```
 902         if (le32_to_cpu(es->s_rev_level) == EXT2_GOOD_OLD_REV) {
 903                 sbi->s_inode_size = EXT2_GOOD_OLD_INODE_SIZE;
 904                 sbi->s_first_ino = EXT2_GOOD_OLD_FIRST_INO;
 905         } else {
 906                 sbi->s_inode_size = le16_to_cpu(es->s_inode_size);
 907                 sbi->s_first_ino = le32_to_cpu(es->s_first_ino);
 908                 if ((sbi->s_inode_size < EXT2_GOOD_OLD_INODE_SIZE) ||
 909                     !is_power_of_2(sbi->s_inode_size) ||
 910                     (sbi->s_inode_size > blocksize)) {
 911                         printk ("EXT2-fs: unsupported inode size:
%d\n",
 912                                 sbi->s_inode_size);
 913                         goto failed_mount;
 914                 }
 915         }
 916//Se rellenan los campos del SB en memoria intermedia "sbi"
 917         sbi->s_frag_size = EXT2_MIN_FRAG_SIZE <<
 918                            le32_to_cpu(es->s_log_frag_size);
 919         if (sbi->s_frag_size == 0)
 920                 goto cantfind_ext2;
 921         sbi->s_frags_per_block = sb->s_blocksize / sbi->s_frag_size;
 922
 923         sbi->s_blocks_per_group = le32_to_cpu(es->s_blocks_per_group);
 924         sbi->s_frags_per_group = le32_to_cpu(es->s_frags_per_group);
 925         sbi->s_inodes_per_group = le32_to_cpu(es->s_inodes_per_group);
 926
 927         if (EXT2_INODE_SIZE(sb) == 0)
 928                 goto cantfind_ext2;
 929         sbi->s_inodes_per_block = sb->s_blocksize /
EXT2_INODE_SIZE(sb);
 930         if (sbi->s_inodes_per_block == 0 || sbi->s_inodes_per_group ==
0)
 931                 goto cantfind_ext2;
 932         sbi->s_itb_per_group = sbi->s_inodes_per_group /
 933                                 sbi->s_inodes_per_block;
 934         sbi->s_desc_per_block = sb->s_blocksize /
 935                                 sizeof (struct
ext2_group_desc);
 936         sbi->s_sbh = bh;
 937         sbi->s_mount_state = le16_to_cpu(es->s_state);
 938         sbi->s_addr_per_block_bits =
 939                 ilog2 (EXT2_ADDR_PER_BLOCK(sb));
 940         sbi->s_desc_per_block_bits =
 941                 ilog2 (EXT2_DESC_PER_BLOCK(sb));
 942//Si no es SF ext2(número mágico) --> ERROR
 943         if (sb->s_magic != EXT2_SUPER_MAGIC)
 944                 goto cantfind_ext2;

 945//Se comprueba que sea correcto el tamaño de bloque
 946         if (sb->s_blocksize != bh->b_size) {
 947                 if (!silent)
 948                         printk ("VFS: Unsupported blocksize on dev "
 949                                 "%s.\n", sb->s_id);
 950                 goto failed_mount;
 951         }
 952
 953         if (sb->s_blocksize != sbi->s_frag_size) {
 954                 printk ("EXT2-fs: fragsize %lu != blocksize %lu (not
supported yet)\n",
```

```
 955                            sbi->s_frag_size, sb->s_blocksize);
 956                    goto failed_mount;
 957            }
 958//Se comprueba que sea correcto el tamaño de fragmento de bloques de
//grupo
 959            if (sbi->s_blocks_per_group > sb->s_blocksize * 8) {
 960                    printk ("EXT2-fs: #blocks per group too big: %lu\n",
 961                            sbi->s_blocks_per_group);
 962                    goto failed_mount;
 963            }//Se comprueba que sea correcto los fragmentos por grupos
 964            if (sbi->s_frags_per_group > sb->s_blocksize * 8) {
 965                    printk ("EXT2-fs: #fragments per group too big: %lu\n",
 966                            sbi->s_frags_per_group);
 967                    goto failed_mount;
 968            }//Se comprueba que sea correcto los inodos por grupos
 969            if (sbi->s_inodes_per_group > sb->s_blocksize * 8) {
 970                    printk ("EXT2-fs: #inodes per group too big: %lu\n",
 971                            sbi->s_inodes_per_group);
 972                    goto failed_mount;
 973            }
 974
 975            if (EXT2_BLOCKS_PER_GROUP(sb) == 0)
 976                    goto cantfind_ext2;
 977            sbi->s_groups_count = ((le32_to_cpu(es->s_blocks_count) -
 978                                    le32_to_cpu(es->s_first_data_block) -
1)
 979                                            / EXT2_BLOCKS_PER_GROUP(sb)) +
1;
 980            db_count = (sbi->s_groups_count + EXT2_DESC_PER_BLOCK(sb) -
1) /
 981                       EXT2_DESC_PER_BLOCK(sb);
//Se crea memoria para buffer_head para s_group_desc
 982            sbi->s_group_desc = kmalloc (db_count * sizeof (struct
buffer_head *), GFP_KERNEL);
 983            if (sbi->s_group_desc == NULL) {
 984                    printk ("EXT2-fs: not enough memory\n");
 985                    goto failed_mount;
 986            }
 987            bgl_lock_init(&sbi->s_blockgroup_lock);
//Se crea memoria para s_debts
 988            sbi->s_debts = kcalloc(sbi->s_groups_count, sizeof(*sbi-
>s_debts), GFP_KERNEL);
 989            if (!sbi->s_debts) {
 990                    printk ("EXT2-fs: not enough memory\n");
 991                    goto failed_mount_group_desc;
 992            }//Se lee cada uno de los descriptores de grupo desde disco
 993            for (i = 0; i < db_count; i++) {
 994                    block = descriptor_loc(sb, logic_sb_block, i);
 995                    sbi->s_group_desc[i] = sb_bread(sb, block);
//Se chequean los descriptores de grupo
//Si se produce algun error se libera la memoria asignada
 996                    if (!sbi->s_group_desc[i]) {
 997                            for (j = 0; j < i; j++)
 998                                    brelse (sbi->s_group_desc[j]);
 999                            printk ("EXT2-fs: unable to read group
descriptors\n");
1000                            goto failed_mount_group_desc;
1001                    }
1002            }
```

```
1003         if (!ext2_check_descriptors (sb)) {
1004                 printk ("EXT2-fs: group descriptors corrupted!\n");
1005                 goto failed_mount2;
1006         }
1007         sbi->s_gdb_count = db_count;
1008         get_random_bytes(&sbi->s_next_generation, sizeof(u32));
1009         spin_lock_init(&sbi->s_next_gen_lock);
1010
1011         /* per fileystem reservation list head & lock */
1012         spin_lock_init(&sbi->s_rsv_window_lock);
1013         sbi->s_rsv_window_root = RB_ROOT;
1014         /*
1015          * Add a single, static dummy reservation to the start of the
1016          * reservation window list --- it gives us a placeholder for
1017          * append-at-start-of-list which makes the allocation logic
1018          * _much_ simpler.
1019          */
1020         sbi->s_rsv_window_head.rsv_start =
EXT2_RESERVE_WINDOW_NOT_ALLOCATED;
1021         sbi->s_rsv_window_head.rsv_end =
EXT2_RESERVE_WINDOW_NOT_ALLOCATED;
1022         sbi->s_rsv_window_head.rsv_alloc_hit = 0;
1023         sbi->s_rsv_window_head.rsv_goal_size = 0;
1024         ext2_rsv_window_add(sb, &sbi->s_rsv_window_head);
1025//Se rellenan más campos del superbloque de memoria intermedia "sbi"
1026         err = percpu_counter_init(&sbi->s_freeblocks_counter,
1027                                 ext2_count_free_blocks(sb));
1028         if (!err) {
1029                 err = percpu_counter_init(&sbi->s_freeinodes_counter,
1030                                 ext2_count_free_inodes(sb));
1031         }
1032         if (!err) {
1033                 err = percpu_counter_init(&sbi->s_dirs_counter,
1034                                 ext2_count_dirs(sb));
1035         }
1036         if (err) {
1037                 printk(KERN_ERR "EXT2-fs: insufficient memory\n");
1038                 goto failed_mount3;
1039         }
1040         /* Se rellena la estructura "sb" en las operaciones
1041          * relacionadas con el superbloque (sb->s_op) con las
1042          */ operaciones propias del sistema de ficheros ext2
1043         sb->s_op = &ext2_sops;
1044         sb->s_export_op = &ext2_export_ops;
1045         sb->s_xattr = ext2_xattr_handlers;
//Lee el nodo raiz del nuevo fs a montar
1046         root = ext2_iget(sb, EXT2_ROOT_INO);
1047         if (IS_ERR(root)) {
1048                 ret = PTR_ERR(root);
1049                 goto failed_mount3;
1050         }
1051         if (!S_ISDIR(root->i_mode) || !root->i_blocks || !root->i_size)
{
1052                 iput(root);
1053                 printk(KERN_ERR "EXT2-fs: corrupt root inode, run
e2fsck\n");
1054                 goto failed_mount3;
1055         }
1056
```

```
//Almacena en root el nodo raiz del nuvevo fs a montar en root
1057        sb->s_root = d_alloc_root(root);
//Si error --> se elimina el nodo raiz
1058        if (!sb->s_root) {
1059                iput(root);
1060                printk(KERN_ERR "EXT2-fs: get root inode failed\n");
1061                ret = -ENOMEM;
1062                goto failed_mount3;
1063        }
1064        if (EXT2_HAS_COMPAT_FEATURE(sb,
EXT3_FEATURE_COMPAT_HAS_JOURNAL))
1065                ext2_warning(sb, __FUNCTION__,
1066                        "mounting ext3 filesystem as ext2");
//Se llama a ext2_setup_super que incrementará el numero de montajes
//en "es"y guardará "es" en disco
1067        ext2_setup_super (sb, es, sb->s_flags & MS_RDONLY);
1068        return 0;
1069
1070cantfind_ext2:
1071        if (!silent)
1072                printk("VFS: Can't find an ext2 filesystem on dev
%s.\n",
1073                       sb->s_id);
1074        goto failed_mount;
//Etiquetas en caso de error
1075failed_mount3:
1076        percpu_counter_destroy(&sbi->s_freeblocks_counter);
1077        percpu_counter_destroy(&sbi->s_freeinodes_counter);
1078        percpu_counter_destroy(&sbi->s_dirs_counter);
1079failed_mount2:
1080        for (i = 0; i < db_count; i++)
1081                brelse(sbi->s_group_desc[i]);
1082failed_mount_group_desc:
1083        kfree(sbi->s_group_desc);
1084        kfree(sbi->s_debts);
1085failed_mount:
1086        brelse(bh);
1087failed_sbi:
1088        sb->s_fs_info = NULL;
1089        kfree(sbi);
1090        return ret;
1091}
```

ext2_commit_super

· ¿Qué hace?

Esta función es llamada para guardar en disco las modificaciones efectuadas en el superbloque.

Guarda la fecha de la última escritura en el superbloque.

Marca como modificado el buffer que contiene el superbloque, mediante la llamada a *mark_buffer_dirty*. De esta manera, el contenido del buffer será escrito a disco en la siguiente operación de guardar en el buffer cache.

Coloca el campo dirt a limpio.

Código:

```
1093static void ext2_commit_super (struct super_block * sb,
1094                               struct ext2_super_block * es)
1095{
//Obtiene la hora de la última escritura en el superbloque
1096        es->s_wtime = cpu_to_le32(get_seconds());
//Se marca como modificado el buffer que contiene el superbloque
//De ello se encarga la función mark_buffer_dirty
1097        mark_buffer_dirty(EXT2_SB(sb)->s_sbh);
1098        sb->s_dirt = 0;
1099}
```

ext2_sync_super

· *¿Qué hace?*

Se encarga de actualizar el superbloque desde memoria a disco.

Actualiza campos de "es", número de bloques e inodos libres, fecha de la última escritura (s_wtime).

Llama a las funciones *mark_buffer_dirty* y *sync_dirty_buffer para escribir en el buffer y luego que se escriba en disco.*

Pone sb->s_dirt a 0 a limpio como que se actualizó.

Código:

```
1101static void ext2_sync_super(struct super_block *sb, struct
ext2_super_block *es)
1102{
1103        es->s_free_blocks_count =
cpu_to_le32(ext2_count_free_blocks(sb));
1104        es->s_free_inodes_count =
cpu_to_le32(ext2_count_free_inodes(sb));
// actualiza fecha de escritura
1105        es->s_wtime = cpu_to_le32(get_seconds());
// marca el buffer como sucio para su escritura en disco
1106        mark_buffer_dirty(EXT2_SB(sb)->s_sbh);
// escribe el buffer en disco
1107        sync_dirty_buffer(EXT2_SB(sb)->s_sbh);
1108        sb->s_dirt = 0;
1109}
```

ext2_write_super

· *¿Qué hace?*

Escribe el superbloque en el buffer asignado a disco para su posterior escritura en disco.

Se comprueba si el FS está en modo lectura/escritura.

Si es un FS válido (libre de errores), se actualizan los campos de estado (*s_state*), números de bloques, inodos libres y fecha del último montaje (*s_mtime*).

Se escribe el superbloque en disco llamando a la función ext2_sync_super.

Si no es fs valido entonces se llama a la función *ext2_commit_super, para que se escriba en disco.*

Código:

```
/*
1112 * En ext2 no es necesario escribir el superbloque ya que se usa un
1113 * mapeo del superbloque de disco en un buffer
1114 * Sin embargo, esta función todavía es utilizada para establecer el
1115 * flag de validez a 0. Se necesita poner este flag a 0 ya que el FS
1116 * puede haber sido comprobado cuando se montaba  y el e2fsck
1117 * puede haber puesto s_state a
1118 * EXT2_VALID_FS después de algunas correcciones.
1119 *
1120 */
1121
1122void ext2_write_super (struct super_block * sb)
1123{
1124         struct ext2_super_block * es;
//Se bloquea el kernel
1125         lock_kernel();
//Se comprueba que el FS no esté en modo de solo lectura
1126         if (!(sb->s_flags & MS_RDONLY)) {
1127                 es = EXT2_SB(sb)->s_es;
1128//Se comprueba que sea un sistema de fichero válido
1129                 if (le16_to_cpu(es->s_state) & EXT2_VALID_FS) {
1130                         ext2_debug ("setting valid to 0\n");
//Se actualiza es->s_state del SF montado
1131                         es->s_state = cpu_to_le16(le16_to_cpu(es-
>s_state) &
1132                                                          ~EXT2_VALID_FS);
//Se actualiza el número de bloques y de inodos libres
1133                         es->s_free_blocks_count =
cpu_to_le32(ext2_count_free_blocks(sb));
1134                         es->s_free_inodes_count =
cpu_to_le32(ext2_count_free_inodes(sb));
//Se actualiza el s_mtime (fecha de la última modificación del SB
1135                         es->s_mtime = cpu_to_le32(get_seconds());
//Se escribe el superbloque "es" en disco
1136                         ext2_sync_super(sb, es);
1137                 } else
//Si no es un sistema de fichero válido se actualiza el SB
1138                         ext2_commit_super (sb, es);
1139         }
1140         sb->s_dirt = 0;
//Se desbloquea el kernel
```

```
1141        unlock_kernel();
1142}
```

ext2_remount

· *¿Qué hace?*

Implementa la operación del sistema de ficheros *remount_fs*. Se monta de nuevo el sistema de ficheros ya montado. Se reconfigura el sistema de ficheros, no lo lee de disco de nuevo.

Decodifica las opciones de montaje con *parse_options.*

Actualizan varios campos del descriptor de superbloque.

Llama a *ext2_setup_super* para inicializar el superbloque.

Código:

```
1144static int ext2_remount (struct super_block * sb, int * flags, char *
data)
1145{
1146        struct ext2_sb_info * sbi = EXT2_SB(sb);
1147        struct ext2_super_block * es;
1148        unsigned long old_mount_opt = sbi->s_mount_opt;
1149        struct ext2_mount_options old_opts;
1150        unsigned long old_sb_flags;
1151        int err;
1152
1153        /* Store the old options */
1154        old_sb_flags = sb->s_flags;
1155        old_opts.s_mount_opt = sbi->s_mount_opt;
1156        old_opts.s_resuid = sbi->s_resuid;
1157        old_opts.s_resgid = sbi->s_resgid;
1158
1159        /*
1160         * Allow the "check" option to be passed as a remount option.
1161         */* Se comprueban las opciones de montaje
1162        if (!parse_options (data, sbi)) {
1163                err = -EINVAL;
1164                goto restore_opts;
1165        }
1166//Se actualizan los siguientes campos del descriptor de fichero
1167        sb->s_flags = (sb->s_flags & ~MS_POSIXACL) |
1168                ((sbi->s_mount_opt & EXT2_MOUNT_POSIX_ACL) ?
MS_POSIXACL : 0);
1169
1170        ext2_xip_verify_sb(sb); /* see if bdev supports xip, unset
1171                                EXT2_MOUNT_XIP if not */
1172
1173        if ((ext2_use_xip(sb)) && (sb->s_blocksize != PAGE_SIZE)) {
1174                printk("XIP: Unsupported blocksize\n");
1175                err = -EINVAL;
1176                goto restore_opts;
1177        }
1178
```

```
1179         es = sbi->s_es;
1180         if (((sbi->s_mount_opt & EXT2_MOUNT_XIP) !=
1181             (old_mount_opt & EXT2_MOUNT_XIP)) &&
1182             invalidate_inodes(sb))
1183                 ext2_warning(sb, __FUNCTION__, "busy inodes while
remounting "\
1184                              "xip remain in cache (no functional
problem)");
//Si se está en modo solo lectura se sale de la función
1185         if ((*flags & MS_RDONLY) == (sb->s_flags & MS_RDONLY))
1186                 return 0;
//Sistema en modo lectura y estaba en modo lectura/escritura
1187         if (*flags & MS_RDONLY) {
//Se comprueba que el estado del FS sea válido
1188                 if (le16_to_cpu(es->s_state) & EXT2_VALID_FS ||
1189                     !(sbi->s_mount_state & EXT2_VALID_FS))
1190                         return 0;
1191                 /*
1192                  * OK, we are remounting a valid rw partition rdonly,
so set
1193                  * the rdonly flag and then mark the partition as valid
again.
1194                  */
//Se monta de nuevo una partición válida lect/esc como sólo lectura
        //Luego se marca de nuevo como una partición válida
1195                 es->s_state = cpu_to_le16(sbi->s_mount_state);
1196                 es->s_mtime = cpu_to_le32(get_seconds());
1197         } else {
1198                 __le32 ret = EXT2_HAS_RO_COMPAT_FEATURE(sb,
1199 
~EXT2_FEATURE_RO_COMPAT_SUPP);
1200                 if (ret) {
1201                         printk("EXT2-fs: %s: couldn't remount RDWR
because of "
1202                                "unsupported optional features (%x).\n",
1203                                sb->s_id, le32_to_cpu(ret));
1204                         err = -EROFS;
1205                         goto restore_opts;
//Si se va a volver a montar una partición lect/esc a solo
lectura //RDONLY, se vuelve a leer el flag de válidez (puede haber sido

//cambiado por e2fsck desde que se montó originalmente la partición.
1206                 }
      /*
       * Si se va a volver a montar una partición lect/esc a solo
       * lectura RDONLY, se vuelve a leer el flag de válidez
       * (puede haber sido cambiado por e2fsck desde que se montó
       * originalmente la partición).
       */
1207                 /*
1208                  * Mounting a RDONLY partition read-write, so reread
and
1209                  * store the current valid flag.  (It may have been
changed
1210                  * by e2fsck since we originally mounted the
partition.)
1211                  */
1212                 sbi->s_mount_state = le16_to_cpu(es->s_state);
//Se llama a ext2_setup_super para incrementar el número de montajes
```

```
//en "es" y guardar en disco
1213                if (!ext2_setup_super (sb, es, 0))
1214                        sb->s_flags &= ~MS_RDONLY;
1215        }
Se actualiza el superbloque en disco con la función ext2_sync_super
1216        ext2_sync_super(sb, es);
1217        return 0;
1218restore_opts:
1219        sbi->s_mount_opt = old_opts.s_mount_opt;
1220        sbi->s_resuid = old_opts.s_resuid;
1221        sbi->s_resgid = old_opts.s_resgid;
1222        sb->s_flags = old_sb_flags;
1223        return err;
1224}
```

ext2_statfs

· ¿Qué hace?

Implementa la operación del sistema de ficheros *statfs*. Copia las estadísticas de uso del sistema de ficheros desde el descriptor del superbloque en la variable pasada por parámetro.

Calcula el tamaño de las cabeceras de un SB.

Lee la información del superbloque (número mágico, tamaño de bloque, número de bloque y número de bloques libres).

Almacena los datos en el buffer pasado por parámetro.

Código:

```
1226static int ext2_statfs (struct dentry * dentry, struct kstatfs * buf)
1227{
1228        struct super_block *sb = dentry->d_sb;
1229        struct ext2_sb_info *sbi = EXT2_SB(sb);
1230        struct ext2_super_block *es = sbi->s_es;
1231        u64 fsid;
1232
1233        if (test_opt (sb, MINIX_DF))
1234                sbi->s_overhead_last = 0;
1235        else if (sbi->s_blocks_last != le32_to_cpu(es->s_blocks_count))
{
1236                unsigned long i, overhead = 0;
1237                smp_rmb();
1238
1239                /*
1240                 * Calcula los bloques que ocupan los metadatos
1241                 * estructuras del FS
1242                 * .
1243                 */
1244
1245                /*
```

```
1246                     * Todos los bloques anteriores a first_data_block son
1247                     * los metadatos
1248                     */
1249                    overhead = le32_to_cpu(es->s_first_data_block);
1250
1251                    /*
1252                     * Añade el superbloque y los grupos de bloques
1253                     *
1254                     *
1255                     */
1256                    for (i = 0; i < sbi->s_groups_count; i++)
1257                            overhead += ext2_bg_has_super(sb, i) +
1258                                    ext2_bg_num_gdb(sb, i);
1259
1260                    /*
1261                     * Cada grupo de bloques tiene un bitmap de inodo,
1262                     * un bitmap de bloque y una tabla de inodo.
1263                     */
1264                    overhead += (sbi->s_groups_count *
1265                                 (2 + sbi->s_itb_per_group));
1266                    sbi->s_overhead_last = overhead;
1267                    smp_wmb();
1268                    sbi->s_blocks_last = le32_to_cpu(es->s_blocks_count);
1269            }//Se lee información del sistema de ficheros a partir
1270 del superbloque En buf se almacena la salida
1271            buf->f_type = EXT2_SUPER_MAGIC;
1272            buf->f_bsize = sb->s_blocksize;
//Bloque de datos(no metadatos)
1273            buf->f_blocks = le32_to_cpu(es->s_blocks_count) - sbi-
>s_overhead_last;
//Los bloques disponibles, no se cuentan los reservados para el
//superusuario
1274            buf->f_bfree = ext2_count_free_blocks(sb);
1275            es->s_free_blocks_count = cpu_to_le32(buf->f_bfree);
1276            buf->f_bavail = buf->f_bfree - le32_to_cpu(es-
>s_r_blocks_count);
1277            if (buf->f_bfree < le32_to_cpu(es->s_r_blocks_count))
1278                    buf->f_bavail = 0;
1279            buf->f_files = le32_to_cpu(es->s_inodes_count);
1280            buf->f_ffree = ext2_count_free_inodes(sb);
1281            es->s_free_inodes_count = cpu_to_le32(buf->f_ffree);
1282            buf->f_namelen = EXT2_NAME_LEN;
1283            fsid = le64_to_cpup((void *)es->s_uuid) ^
1284                   le64_to_cpup((void *)es->s_uuid + sizeof(u64));
1285            buf->f_fsid.val[0] = fsid & 0xFFFFFFFFUL;
1286            buf->f_fsid.val[1] = (fsid >> 32) & 0xFFFFFFFFUL;
1287            return 0;
1288}
```

ext2_init_ext2_fs

· *¿Qué hace?*

Registra el sistema de ficheros EXT2.

Llama a la función register_filesystem.

Código:

```
1409static int __init init_ext2_fs(void)
1410{
1411        int err = init_ext2_xattr();
1412        if (err)
1413                return err;
1414        err = init_inodecache();
1415        if (err)
1416                goto out1;
1417        err = register_filesystem(&ext2_fs_type);
1418        if (err)
1419                goto out;
1420        return 0;
1421out:
1422        destroy_inodecache();
1423out1:
1424        exit_ext2_xattr();
1425        return err;
1426}
```

17.6 Journaling

Para minimizar las inconsistencias en los sistemas de ficheros y minimizar a su vez el tiempo de arranque, los sistemas de ficheros Journaling mantienen un seguimiento de los cambios que se llevaran a cabo en el sistema de ficheros antes de que realmente se lleven a cabo. Estos registros se almacenan en una parte separada del sistema de ficheros, normalmente conocida como el “journal” (diario) o “log”. Una vez que los registros journal (también conocidos como registros “log”) se han escrito correctamente, el sistema de ficheros aplica esos cambios sobre al sistema de ficheros y después elimina esas entradas del registro journal correspondiente.

Los sistemas de ficheros journaling maximizan la consistencia del sistema de ficheros ya que los logs se escriben antes de que los cambios se vayan a realizar, almacenando esos registros hasta que los cambios hayan sido completados satisfactoriamente sobre el sistema de ficheros. Cuando reiniciamos un ordenador que utiliza un sistema de ficheros journaling, el programa de montaje puede garantizar la consistencia del sistema de ficheros simplemente buscando en el registro journal los cambios pendientes (sin acabar) y finalizarlos sobre el sistema de ficheros. En la mayoría de los casos, el sistema no tiene que comprobar la consistencia del sistema de ficheros, por lo que los ordenadores que utilicen un sistema de ficheros journaling estarán disponibles casi instantáneamente después del retrancarlos. También, la perdida de datos debido a inconsistencias en el sistema de ficheros se reduce drásticamente.

En el sistema Journaling se hace una analogía entre al concepto de base de datos SQL y el sistema de ficheros. El concepto es familiar para cualquiera que conozca como trabaja una base de datos SQL con la lógica de las transacciones. Dentro de esta lógica, “roll forward” se conoce al proceso de volver a ejecutar una operación que fue interrumpida y completarla y “roll back” al proceso de llevar a esa operación a su estado inicial. Esta metodología de trabajo que se desarrollo para prevenir la perdida de datos en las bases de datos SQL, es también extensible a la gestión de los sistemas de ficheros. Podríamos decir sin duda que este es el gran beneficio de los sistemas de ficheros Journaling.

Existen varios sistemas de ficheros journaling disponibles para el sistema operativo Linux. Los más conocidos son; el XFS –desarrollado por Silicon Graphics pero no es open source (código abierto), el ReiserFS –especialmente desarrollado para Linux, el JFS –originalmente desarrollado por IBM del cual se dispone actualmente una versión open source, y el sistema de ficheros ext3 –desarrollado por Stephen Tweedie en Red Hat con la colaboración de otros contribuidores.

17.7 Tercer sistema de ficheros extendido (Ext3)

Este nuevo sistema de ficheros ha sido diseñado con dos conceptos en mente:
Que sea un sistema de ficheros Journaling
Que sea todo lo compatible posible con el Ext2
El Ext3 consigue ambos objetivos muy bien. En particular, esta totalmente basado en el Ext2, por lo que las estructuras de disco son esencialmente idénticas. De hecho, si un sistema de ficheros Ext3 se desmonta limpiamente, puede volver a montarse como un sistema de ficheros Ext2. Al revés se puede hacer lo mismo; crear un sistema Journal a partir de un sistema de ficheros Ext2 remontándolo hasta conseguir un sistema de ficheros Ext3 es una operación sencilla y rápida.
Gracias a la compatibilidad entra el Ext3 y el Ext2, todo lo visto anteriormente para el Ext2 es totalmente aplicable al Ext3. Por lo que ahora simplemente nos vamos a centrar en las características propias del sistema journaling.

Sistema Journal para el Ext3

En un sistema de ficheros Ext2, actualizaciones en los bloques del sistema de ficheros se mantienen durante un tiempo en memoria dinámica antes de ser volcados en el disco. Un evento dramático, como un fallo en la corriente del sistema o un fallo en si del sistema, puede hacer que el sistema se quede en un estado de inconsistencias. Para sobreponerse a este problema, cada sistema tradicional del Unix es chequeado antes de ser montado; si no es ha sido desmontado correctamente, un programa específico ejecuta un exhaustivo chequeo (que consume mucho tiempo) y corrige todas las posibles inconsistencias que encuentre en el sistema de ficheros a nivel de metadatos.
En el sistema de ficheros Ext2 el estado se almacena en el campo s_mount_state del superbloque del disco. La utilidad *e2fsck* en invocada por el script de arranque para chequear el valor almacenado en este campo; si no es igual a EXT2_VALID_FS, el sistema de fichero no se desmonto correctamente, y por lo tanto e2fsck comienza a chequear todas las estructuras de datos del sistema de ficheros del disco.
Claramente, el tiempo dedicado al chequeo de la consistencia del sistema de ficheros depende en gran medida del número de ficheros y directorios a examinar, y del tamaño del disco. Hoy en día, con sistemas de ficheros alcanzando los cientos de gigabytes, un simple chequeo de consistencia puede llevar horas.
La idea del Journaling Ext3 es realizar cualquier cambio de alto nivel en el sistema de ficheros en dos pasos. Primero, una copia de los bloques que serán escritos se almacena un Journal (diario); entonces, cuando la operación E/S de transferencia de los datos al Journal se completa (realizándose un *commit* sobre el Journal), los bloques se escriben en el sistema de ficheros. Cuando la operación de E/S de transferencia de datos al sistema de ficheros se completa (realizándose un *commit* sobre el sistema de ficheros), la copia de los bloques en el Journal se descartan.
Mientras se recupera de un fallo en el sistema, el programa *e2fsck* distingue dos casos:

El fallo en el sistema ocurrió antes de que se realizara un *commit* en el Journal. Por lo que, o las copias de los bloques relativos a los cambios se han perdido del Journal, o están incompletos; en ambos casos el *e2fsck* los ignora.
El fallo en el sistema ocurrió después de que se realizara el *commit* en el Journal. La copia de los bloques son válidos y el e2fsck los escribe en el sistema de ficheros.
En el primero de los casos, los cambios sobre el sistema de ficheros se han perdido, pero el estado del mismo es consistente. En el segundo de los casos, el e2fsck aplica todos los cambios previstos sobre el sistema de ficheros, arreglando cualquier inconsistencia debida a operaciones E/S de transferencia de datos inacabadas sobre el sistema de ficheros.
Un sistema de ficheros Journaling no guarda normalmente todos los bloques en el Journal. De hecho, existen dos tipos de bloques: aquellos que contienen los llamados metadatos del sistema de ficheros y los que contienen los datos en sí. De hecho, los registros log de los metadatos son suficientes para recuperar la consistencia de las estructuras del sistema de ficheros. De todas formas, si las operaciones sobre los bloques de un fichero de datos no se almacenan, nada previene que un fallo del sistema corrompa el contenido del mismo.
El Ext3 puede configurarse para que realice un seguimiento tanto en las operaciones que afectan a los metadatos como en aquellas que afectan a los bloques de datos de un fichero. Esto depende de cómo se haya montado el sistema de ficheros. Estos son los diferentes modos de Journal:
Journal
Todos los cambios en los datos y metadatos del sistema de ficheros se “logean” en el Journal. Este modo minimiza la posibilidad de perdida de las actualizaciones hechos sobre cada fichero, pero requiere muchos accesos adicionales a disco. Por ejemplo, cuando un fichero se crea, todos sus bloques de datos tienen que duplicarse como registros de log. Este es el método más seguro pero el más lento de los modos Journaling.
Ordered
Solo los cambios en los metadatos del sistema de ficheros se almacenan el Journal. De todas maneras, el Ext3 agrupa los metadatos y sus bloques de datos relacionados de manera que los bloques de datos sean escritos en disco antes que los metadatos. De esta manera, la posibilidad de tener datos corruptos en un fichero se reducen. Este es el método por defecto del Ext3.
Writeback
Solo los cambios en los metadatos del sistema de ficheros se almacenan el Journal. Este es el método más rápido, pero el más inseguro.
El modo Journaling se especifica en una opción del comando de montaje del sistema. Por ejemplo, para montar un sistema de ficheros Journaling en la partición /dev/hda1 en el punto de montaje /jdisk con el modo *writeback* será:

```
# mount -t ext3 -o data=writeback /dev/hda1 /jdisk
```

Superbloque en Ext2

Capa del dispositivo de bloque Journaling (Journaling Block Device –JBD)

El Journal del Ext3 normalmente se almacena en el fichero oculto llamado *.journal* que se encuentra en el directorio root del sistema de ficheros.
El Ext3 no maneja el fichero journal por si mismo, si no que utiliza una capa superior del núcleo llamada dispositivo de bloque Journaling, o JBD. El Ext3 invoca a las rutinas del JBC para asegurarse que las sucesivas operaciones no corrompan las estructuras de datos en caso de fallo en el sistema. La interacción entre el JBD y el Ext3 se basa esencialmente en tres unidades principales:
Registros Log.- Describen la actualización sobre un bloque del sistema de ficheros Journaling.
Manejador de Operación Atómica.- Incluye los registros Log relativos a un cambio en el sistema de ficheros. Normalmente cada llamada al sistema que modifica el sistema de ficheros utiliza un único manejador de operación atómica.
Transacción.- Incluye varios manejadores de operación atómica. No es más que un conjunto de operaciones controladas por los manejadores de operación atómica correspondientes.

17.8 Sistema de ficheros ReiserFS

Una vez vistos los distintos tipos de sistemas Journaling, vamos hablar ahora un poco de uno de ellos, el sistema de ficheros ReiserFS. ReiserFS fue diseñado originalmente por Hans Reiser. La primera versión se lanzó a mediados de los 90, llegando a formar parte del núcleo en la versión 2.4.1 del mismo (ReiserFS v3.5). La versión 3.6 es el sistema de ficheros por defecto de las distribuciones SUSE, GENTOO y LINDOWS. La versión actual de prueba (testing) es la v4.

Las características principales del ReiserFS son:

Rápido, pero eficiente en la gestión del espacio.

10 veces más rápido que el Ext2

Sin penalización para ficheros pequeños

Fiable

Soporte Journaling

Compatible con la semántica de UNIX, pero Extensible a través de un sofisticado sistema de Plugins

En esencia, el sistema de ficheros ReiserFS en un sistema Journaling que trata a toda la partición del disco como si fuera una única tabla de una base de datos. Los directorios, ficheros y metadatos se organizan en una eficiente estructura de datos llamada “árbol balanceado”. Esto difiere bastante de la manera tradicional de trabajar de otros sistemas de ficheros, pero ofrece grandes mejoras de velocidad en

muchas aplicaciones, especialmente aquellas que utilizan gran cantidad de ficheros pequeños.

Leer y escribir en ficheros grandes, como imágenes de CDROM, normalmente esta limitado por la velocidad del hardware implicado o por el canal de entrada / salida. Pero en cambio, los accesos a pequeños ficheros como los scripts de la shell estan normalmente limitados por la eficiencia del sistema de ficheros. Esto es debido a que la apertura de un fichero requiere que primero el sistema localice el fichero, lo que implica la lectura del directorio en el que se encuentra. Después, el sistema necesita examinar los metadatos para saber si el usuario tiene los permisos de accesos necesarios, lo conlleva una serie de lecturas adicionales del disco. En definitiva, el sistema pierde más tiempo decidiendo si permite el acceso al fichero que el tiempo que finalmente se tarda en obtener la información del mismo.

ReiserFS utiliza los árboles balanceados (Árboles B*) para racionalizar el proceso de la localización de los ficheros y la obtención de la información de los metadatos (y otro tipo de información adicional). Para ficheros extremadamente pequeños, toda la información del fichero puede físicamente almacenarse cerca de los metadatos, de manera que ambos pueden ser accedidos simultáneamente con un poco o nada de movimiento en el mecanismo de búsqueda del disco. Si una aplicación necesita abrir ficheros muy pequeños rápidamente, este planteamiento mejora el rendimiento significativamente.

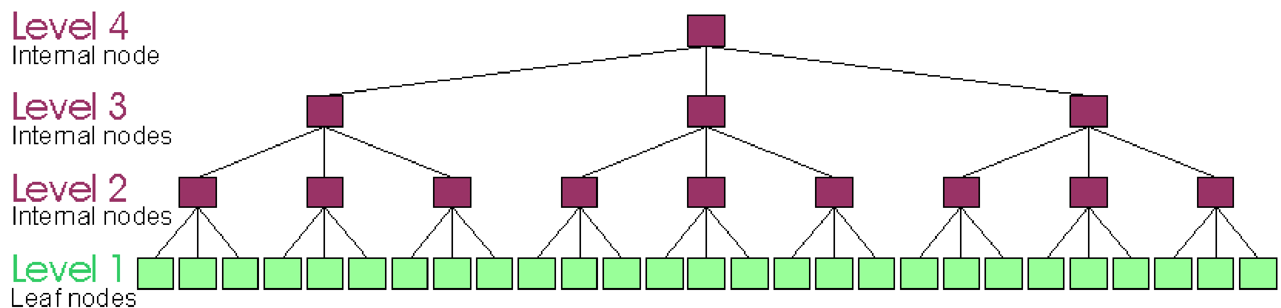


Figura 2 (Ejemplo de un de la estructura de árbol –Árbol B*)

Los nodos hojas (leaf nodes), almacenan los datos en sí, y no tienen hijos. Los nodos que contienen información se dice que están formateados. Solo los nodos hoja pueden estar no formateados. Los punteros se almacenan en nodos formateados por lo que los nodos intermedios (internal nodes) necesariamente tienen que estar formateados. El nodo raíz es de donde nace el árbol.

Otra característica de ReiserFS es que los árboles balanceados no solo almacenan metadatos, si que también almacenan los datos en sí. En un sistema de ficheros tradicional como el ext2, el espacio en disco se reparte en bloques de tamaño desde 512 bytes a 4096 bytes. Si el tamaño de un fichero no es múltiplo del tamaño de bloque habrá cierto espacio que se desperdiciara (fragmentación interna). Por ejemplo, supongamos un tamaño de bloque de 1024 bytes y un espacio requerido de 8195 bytes. 8192 bytes estarán almacenados en 8 bloques (8 x 1024), y los restantes 3 bytes se tendrán que ubicar en un bloque de 1024, desperdiciándose de

esta manera 1021 bytes. De los 9 bloques que se han tenido que asignar casi uno por completo se desperdicia, lo que representa un 11% del total solicitado (1 byte / 9 bytes). Ahora imaginemos un fichero de 1025 bytes. Casi cabe por completo en un solo bloque, pero requiere dos. El espacio desperdiciado es del 50% (1/2). El peor de los casos se da con ficheros muy pequeños, como un script muy trivial de una sola línea. Este fichero podría ser de unos 50 bytes, teniendo que ocupar un bloque completo. Si como hemos dicho, el tamaño de bloque es de 1024 bytes, el espacio desperdiciado será de un 95 % del espacio asignado. Como se puede ver, el espacio desperdiciado (en porcentaje) es menor cuanto mayor es el tamaño del fichero.

ReiserFS no utiliza el enfoque clásico de bloques a la hora dividir el espacio de disco, en vez de eso se apoya en la estructura arbórea para mantener un seguimiento de los bytes asignados. En ficheros pequeños, esto puede ahorrar como ya hemos dicho mucho espacio de almacenamiento. Es más, debido a que los ficheros tienden a colocarse cercanos unos de otros, el sistema es capaz de abrir y leer muchos ficheros pequeños con un único acceso físico al disco. Esto conlleva un ahorro de tiempo al disminuir el tiempo de búsqueda de la cabeza lectora del disco.

Algunas aplicaciones se benefician más que otras de este tipo de optimización. Imagínate un directorio con cientos de pequeños ficheros PNG o GIF que son utilizados como iconos en una página Web muy visitada. Esta situación esta hecha a la medida del sistema de ficheros ReiserFS. Así como una pagina Web con miles de ficheros HTML, cada uno de unos poco kilobytes de tamaño, es un excelente candidato. Por otro lado, una partición de disco que alacena imágenes ISO9660 de CDROM, cada una de cientos de megabytes, vera poca mejora en su rendimiento con el ReiserFS. Como con otras muchas cosas en el mundo de la informática, el mejor rendimiento se obtiene a través de la selección de la herramienta que mas se ajuste a las necesidades especificas del problema a resolver. (Nota: esto no significa que ReiserFS sea más lento que Ext2 a la hora de gestionar ficheros de gran tamaño, solo que no se notara mucha diferencia de rendimiento en esos casos.)

Para almacenar grandes ficheros, ReiserFS versión 3 utiliza el método BLOB (Binary Large OBject) que desbalanceaba el árbol reduciendo así el rendimiento. En este método se almacenan (en los nodos hoja formateados) punteros a nodos que contienen el fichero en su conjunto (como muestra la imagen). De manera que desbalancea al árbol.

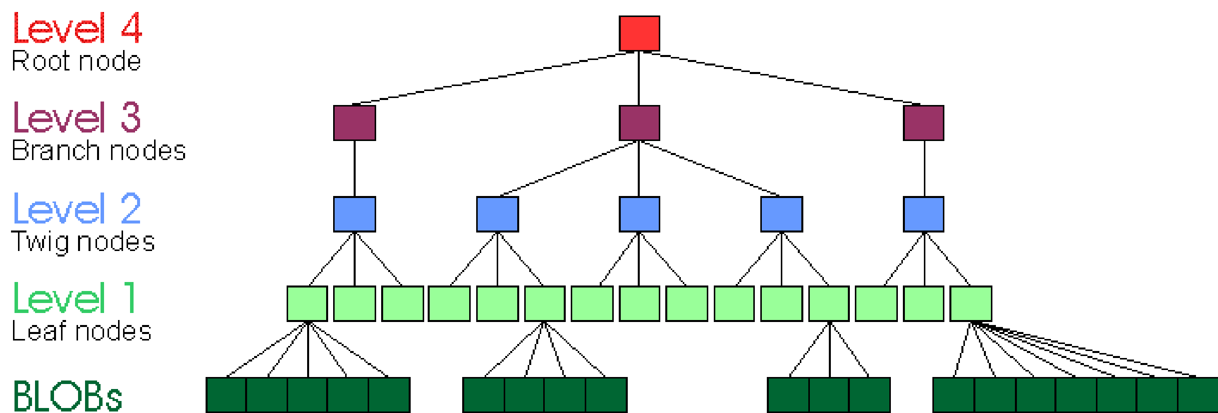


Figura 3 (Árbol B* no balanceado de ReiserFS v3)

En la versión 4 de ReiserFS se intenta solucionar este problema haciendo que todos los punteros a ficheros se almacenen en el mismo nivel.

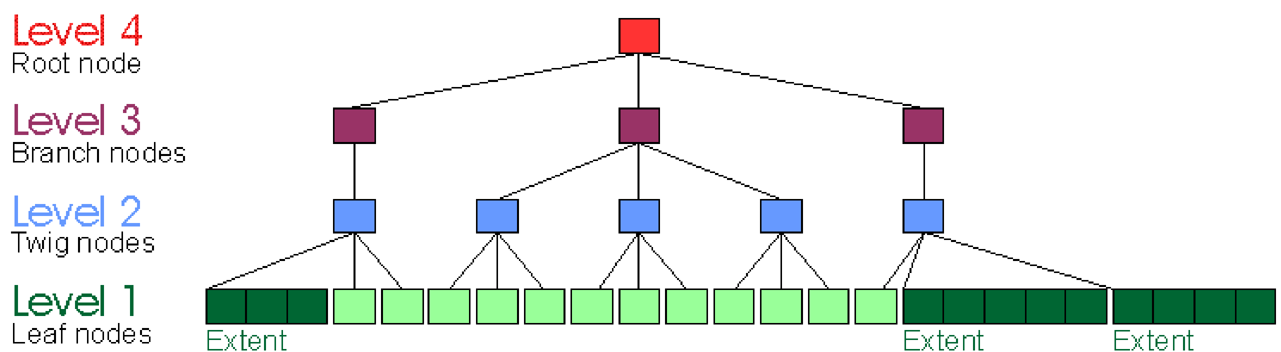


Figura 4 (Árbol B* balanceado de ReiserFS v4, inclusión de nodos Twig, Branch y punteros Extendidos)

Los nodos Twig, son los padres de los nodos hojas. Los nodos Branch son nodos internos que no son Twig. En ReiserFS, los nodos Twig no solo apuntan a nodos hoja, sino que también apuntan a nodos Extendidos (Extended nodes). De esta manera se mantiene al árbol balanceado, almacenando los ficheros pequeños en nodos hoja y los grandes en nodos a través de punteros extendidos.

Otra característica destacable del ReiserFS es la utilización de *Plugins*, lo que permite la adaptación del sistema de ficheros a nuevas características sin la necesidad de formateo. Los *Plugins* trabajan de forma independiente sobre el núcleo del sistema de ficheros. Todas las operaciones se implementan en Plugins, por lo que permite que se modifique la funcionalidad del sistema de ficheros (añadiendo o eliminando Plugins existentes) sin necesidad de modificar el núcleo.

Por encima de todo, ReiserFS es un verdadero sistema de ficheros Journaling como el xfs, el Ext3 y el JFS de IBM. Cada uno de estos sistemas de ficheros implementan a su manera las características Journaling, pero los efectos son los mismos: una

muy buena fiabilidad y una velocidad muy rápida de recuperación después de un apagado inadecuado. Esta es la principal ventaja de un sistema de ficheros Journaling contra otro tradicional.

Bibliografía

The Linux Kernel Book. Rémy Card, Éric Dumas y Frank Mével

[Tanenbaum 1987] A. Tanenbaum. Operating Systems: Design and Implementation. Prentice Hall, 1987.

[Bach 1986] M. Bach. The Design of the UNIX Operating System. Prentice Hall, 1986.

[McKusick et al. 1984] M. McKusick, W. Joy, S. Leffler, and R. Fabry. A Fast File System for UNIX. ACM Transactions on Computer Systems, 2(3):181--197, August 1984.

Arbol de funciones