TEMA 4. SISTEMA DE ARCHIVOS

4.1. CARACTERÍSTICAS DEL SISTEMA DE ARCHIVOS DE UNIX.

- Un sistema de archivos permite realizar una abstracción de los dispositivos físicos de almacenamiento de la información para que sean tratados a nivel lógico, como una estructura de más alto nivel y más sencilla que la estructura de su arquitectura hardware particular.

- El sistema de archivos UNIX se caracteriza por:
  - Poseer una estructura jerárquica.
  - Realizar un tratamiento consistente de los datos de los archivos.
  - Poder crear y borrar archivos.
  - Permitir un crecimiento dinámico de los archivos.
  - Proteger los datos de los archivos.
  - Tratar a los dispositivos y periféricos (terminales, unidades de disco, cinta, etc.) como si fuesen archivos.

- El sistema de archivos está organizado, a nivel lógico, en forma de árbol invertido, con un nodo principal conocido como nodo raíz ("/"). Cada nodo dentro del árbol es un directorio y puede contener a su vez otros nodos (subdirectorios), archivos normales o archivos de dispositivo.

- Los nombres de los archivos *(pathname)* se especifican mediante la ruta (*path*), que describe cómo localizar un archivo dentro de la jerarquía del sistema. La ruta de un archivo puede ser absoluta (referida al nodo raíz) o relativa (referida al directorio de trabajo actual, CWD *current work directory*).

4.2. ESTRUCTURA GENERAL DE UN SISTEMA DE ARCHIVOS DE UNIX.

- Los sistemas de archivos suelen estar situados en dispositivos de almacenamiento modo bloque, tales como discos o cintas.

- Un sistema UNIX puede manejar uno o varios discos físicos, cada uno de los cuales puede contener uno o varios sistemas de archivos. Los sistemas de archivos son particiones lógicas del disco.

- Hacer que un disco físico contenga varios sistemas de archivos permite una administración más segura, ya que si uno de los sistemas de archivos se daña, perdiéndose la información que hay en él, este accidente no se habrá transmitido al resto de los sistemas de archivos que hay en el disco y se podrá seguir trabajando con ellos para intentar una restauración o una reinstalación.

- El *kernel* del sistema operativo trabaja con el sistema de archivos a un nivel lógico y no trata directamente con los discos a nivel físico. Cada disco es considerado como un dispositivo lógico que tiene asociados unos números de dispositivo (*minor number* y *major number*). Estos números se utilizan para acceder al controlador del disco. Un controlador del disco se va a encargar de transformar las direcciones lógicas (*kernel*) de nuestro sistema de archivos a direcciones físicas del disco.

- Un sistema de archivos se compone de una secuencia de bloques lógicos, cada uno de los cuales tiene un tamaño fijo (homogéneo). El tamaño del bloque es el mismo para todos el sistema de archivos y suele ser múltiplo de 512.

- A pesar de que el tamaño del bloque es homogéneo en un sistema de archivos, puede variar de un sistema a otro dentro de una misma configuración UNIX con varios sistemas de archivos. El tamaño elegido para el bloque va a influir en las prestaciones globales del sistema. Por un lado, interesa que los bloques sean grandes para que la velocidad de transferencia entre el disco y la memoria sea grande. Si embargo, si los bloques lógicos son demasiado grandes, la capacidad de almacenamiento del disco se puede ver desaprovechada cuando abundan los archivos pequeños que no llegan a ocupar un bloque completo. Valores típicos para el tamaño del bloque: 512, 1024 y 2048 (bytes).

- En la Figura 4.1 podemos ver, en un primer nivel de análisis, la estructura que tiene un sistema de archivos UNIX:

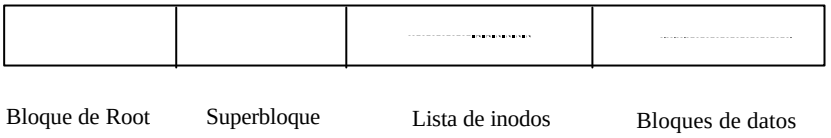


Figura 4.1. Primer nivel en la estructura de un sistema de archivos.

- En la figura anterior podemos observar cuatro partes:
  - El *bloque de arranque* (boot). Ocupa la parte del principio del sistema de archivos, normalmente en el primer sector, y puede contener el código de arranque. Este código es un pequeño programa que se encarga de buscar el sistema operativo y cargarlo en memoria.
  - El *superbloque* describe el estado de un sistema de archivos. Contiene información acerca de su tamaño, total del archivo que puede contener, qué espacio libre queda, etc.
  - La lista de inodos (nodos índice). Se encuentra a continuación del superbloque. Esta lista tiene una entrada por cada archivo, donde se guarda una descripción del mismo: situación del archivo en el disco, propietario, permisos de acceso, fecha de actualización, etc. El administrador del sistema es el encargado de especificar el tamaño de la lista de inodos al configurar el sistema.
  - Los bloques de datos empiezan a continuación de la lista de inodos y ocupa el resto del sistema de archivos. En esta zona es donde se encuentra situado el contenido de los archivos a los que hace referencia la lista de inodos. Cada uno de los bloques destinados a datos sólo puede ser asignado a un archivo, tanto si lo ocupa totalmente como si no.

4.3. REPRESENTACIÓN INTERNA DE LOS ARCHIVOS EN UNIX.

4.3.1. El Superbloque (superblock).

- Como hemos dicho anteriormente, en el superbloque está la descripción del estado del sistema de archivos. Los campos que forman el superbloque son:
  - Tamaño del sistema de archivos.
  - Número de bloques libres disponible en el sistema de archivos.
  - Lista de bloques libres en el sistema de archivos.
  - Índice al siguiente bloque libre en la lista de bloques libres.
  - Tamaño de la lista de inodos.
  - Número total de inodos libres en el sistema de archivos.
  - Lista de inodos libres en el sistema de archivos.
  - Índice al siguiente inodo libre en la lista de inodos libres.
  - Campos de bloqueo de elementos para la listas de bloques y inodos libres. Estos campos se emplean cuando se realiza un petición de bloques o de inodos libres.
  - Indicador (flag) que informa que el superbloque ha sido modificado o no.

- Al arrancar el sistema, se montan los Sistemas de Archivos locales y remotos y se lee el superbloque a memoria

- Cada vez que, desde un proceso, se accede a un archivo, es necesario consultar el superbloque y la listas de inodos. Como el acceso a disco suele degradar bastante el tiempo de ejecución de un programa, lo normal es que el *kernel* realice la E/S con el disco a través del buffer caché y que el sistema tenga siempre en memoria una copia del superbloque y de la tabla de inodos. Esto va a plantear problemas de inconsistencia de los datos, ya que una actualización en memoria del superbloque y de la tabla de inodos no implica una actualización inmediata en disco. La solución a este problema consiste en que el *kernel* escribe periódicamente el superbloque y lista de inodos en el disco si había sido modificado, de forma que siempre sea consistente con los datos del sistema de archivos. De esta tarea se encarga un *demonio* (por *demonio* se debe de entender a todo programa del sistema que se está ejecutando en segundo plano *(background)* y realiza tareas periódicas relacionadas con la administración del sistema) denominado *syncer* que se arranca al inicializar el sistema y se encarga de actualizar periódicamente los datos de administración del sistema. Obviamente, antes de apagar el sistema hay que actualizar el superbloque y la tabla de inodos del disco. El programa *shutdown* se encarga de esta tarea y es fundamental tener presente que no se deba realizar una parada del sistema sin haber invocado previamente al programa *shutdown*, porque de lo contrario el sistema de archivos podría quedar seriamente dañado.

4.3.2. Inodos y Operaciones con inodos.

4.3.2.1. Estructura de un inodo (inode).

- Cada archivo en un sistema UNIX tiene asociado un inodo. El inodo contiene la información necesaria para que un proceso pueda acceder al archivo. Esta información incluye: propietario, derechos de acceso, tamaño, localización en sistema de archivos, etc.

- La lista de inodos se encuentra situada en los bloques que hay a continuación del superbloque. Durante el proceso de arranque del sistema, el *kernel* lee la lista de inodos y carga un copia en memoria conocida como *tabla de inodos* o *caché de inodos* (tabla de igual forma que el *buffer caché*). Las manipulaciones que haga el subsistema de archivos (parte del código del *kernel*) sobre los archivos van a involucrar la tabla de inodos pero no a la lista de inodos. Mediante este mecanismo se consigue una mayor velocidad de acceso a los archivos, ya que la tabla de inodos está cargada siempre en memoria.

- Los campos de que se compone un inodo en disco son los siguientes:
  - *Identificador del propietario del archivo*. La posesión se divide entre un propietario individual y un grupo de propietarios, y define el conjunto de usuarios que tienen derecho de acceso al archivo. El superusuario tiene derecho de acceso a todos los archivos del sistema de archivos.
  - *Tipo de archivo*. Los archivos pueden ser ordinarios de datos (regulares), directorios, especiales de dispositivo (en modo carácter o en modo bloque) y tuberías (o pipes).
  - *Tipo de acceso al archivo*. El sistema protege a los archivos estableciendo tres niveles de permisos: permisos del propietario, del grupo de usuarios al que pertenece el propietario y el resto de los usuarios. Cada clase de usuarios puede tener habilitados o deshabilitados los derechos de lectura, escritura y ejecución. Para los directorios el derecho de ejecución significa poder acceder o no a los archivos que contiene.
  - *Tiempos de acceso al archivo*. Estos campos dan información sobre la fecha de la última modificación del archivo, la última vez que se accedió a él (día y hora de la última modificación y acceso al archivo) y la última vez que se modificaron los datos de su inodo (día y hora de la última modificación de su inodo).

  - *Número de enlaces (de nombres) del archivo*. Representa el total de los nombres que el archivo tiene en la jerarquía de directorios. Como veremos más adelante, un archivo puede tener asociados diferentes nombres que correspondan a diferentes rutas, pero a través de los cuales accedemos a un mismo inodo y por consiguiente a los mismos bloques de datos. Es importante destacar que en el inodo no especifica el (los) nombre(s) del archivo.
  - *Entradas para los bloques de dirección de los datos de un archivo*. Si bien los usuarios tratan los datos de un archivo como si fueran una secuencia de bytes contiguos, el *kernel* puede almacenarlos en bloques que no tienen por qué ser contiguos. En los bloques de dirección es donde se especifican los bloques de disco que contienen los datos del archivo.
  - *Tamaño del archivo*. Los bytes de un archivo se pueden direccionar indicando un desplazamiento a partir de la dirección de inicio del archivo (desplazamiento 0). El tamaño del archivo es igual al desplazamiento del byte más alto incrementado en una unidad. Por ejemplo, si un usuario crea un archivo y escribe en el sólo un byte en la posición 2000, el tamaño del archivo es 2001 bytes.
- Hay que especificar que el nombre del archivo no queda especificado en su inodo. Como veremos mas adelante, es en los archivos de tipo directorio donde a cada nombre de archivo se le asocia su inodo correspondiente.
- También hay que indicar la diferencia entre escribir el contenido de un inodo en disco y escribir el contenido del archivo. El contenido del archivo (sus datos) cambia sólo cuando se escribe en él. El contenido de un inodo cambia cuando se modifican los datos del archivo o la situación administrativa del mismo (propietario, permisos, enlaces, etc.).
- La tabla de inodos (caché de inodos) contiene en memoria la misma información que la lista de inodos además de la siguiente información adicional (descriptor del inodo):
  - Estado del inodo, que indica:
    + Si el inodo esta bloqueado (prevención de acceso al inodo).
    + Si hay algún proceso esperando que el inodo quede desbloqueado (flag de petición de aviso por este inodo).
    + Si la copia del inodo que hay en memoria difiere de la que hay en el disco (flag de discrepancias entre inodo memoria/disco).
    + Si la copia de los datos del archivo que hay en memoria difiere de los datos que hay en disco (caso de la escritura en el archivo a través del *buffer caché*).
    + Si el archivo es un punto de montaje.
  - El número de dispositivo lógico del sistema de archivos que contiene al archivo.
  - El número de inodo. Como en muchos casos los inodos se almacenan en disco en un array lineal, al cargarlo en memoria, el *kernel* le asigna un número en función de su posición en el array, mientras que el inodo del disco no necesita esa información.
  - Punteros a otros inodos cargados en memoria. El *kernel* enlaza los inodos sobre una lista hash y sobre una lista de inodos libre (igual que ocurría con el *buffer caché*). Las claves de acceso a la lista hash nos la dan el número de dispositivo lógico del inodo y el número de inodo.
  - Un contador (contador de referencias) que indica el número de copias del inodo que están activas (por ejemplo, porque el archivo está abierto por varios procesos). Su uso es muy variado: instancias activas de un archivo, activa la solicitud de un proceso, Libres cuando tiene valor 0 (un buffer está libre cuando no está bloqueado).
- Como hemos podido comprobar, la información almacenada en los descriptores de los inodos en memoria es muy similar en estructura de la información almacenada en los buffers (gestión del *buffer caché*):
  - La estructura del descriptor de los inodos en memoria es similar a la estructura del descriptor de los buffers en el *buffer caché*.
  - La gestión y manejo de los inodos es similar a la gestión y manejo de los buffers en el *buffer caché*.
  - Diferencia entre la gestión de los inodos y de los buffers en el *buffer caché* es la cuenta de referencia (contador LRU) que guía la política de reemplazo de buffers.

4.3.2.2. Operaciones con inodos.

- Los algoritmos que se describen ⇒ capa inmediatamente superior a los algoritmos vistos en el capítulo anterior que gestionan el *buffer caché* (Figura 4.2).
  - *iget*: Obtiene un inodo, leído del disco a través del *buffer caché*.
  - *iput*: Libera el inodo.
  - *bmap*: Fija parámetros del *kernel* para acceder a un archivo.
  - *namei*: Convierte el nombre de un archivo ⇒ inodo ⇒ utilizando los algoritmos *iget, iput* y *bmap*. Traslada un *pathname* a un número de inodo.
  - *alloc* y *free*: Asignan y liberan bloques de disco para los archivos.
  - *ialloc* e *ifree*: Asignan y liberan los inodos para los archivos.

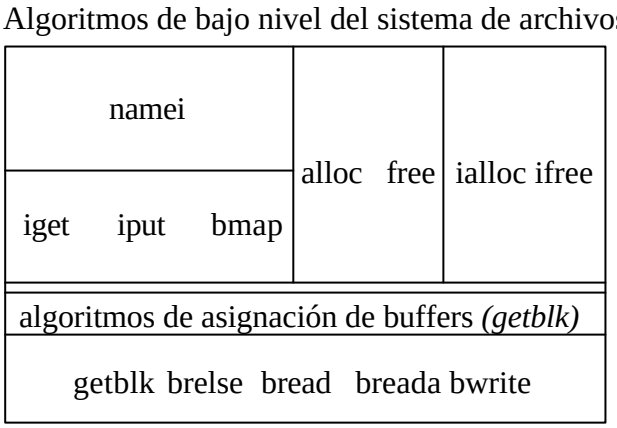


Figura 4.2. Algoritmos del sistema de archivos.

- Identificación de inodos ⇒ N° inodo + nº sistema de archivos ⇒ permiten al *kernel* identificar el inodo.

- Asignación de los inodos ⇒ Se trata de asignar una copia en memoria para un inodo.

4.3.2.2.1. Operaciones para la Asignación de inodos: iget.

- El *kernel* busca el inodo en la lista hash (buffer caché): n° dispositivo lógico y n° inodo.

- Si no está allí:
  - Le asigna uno de la lista de libres y lo bloquea. Tareas a realizar:
    + Realiza la lectura del inodo de disco.
    + Calcula el n° bloque de disco en el que se encuentra:
      * ((n° inodo – 1) / n° inodos_bloque) + bloque inicio en la lista de inodos.
    + Lee el bloque (algoritmo *bread*).
    + Calcula el byte de comienzo del inodo dentro del bloque:
      * ((n° inodo – 1) mod (n° inodos_bloque)) * tamaño del inodo en disco.
    + Elimina el inodo de la lista de libres y lo pone en lista hash adecuada.
    + Pone a 1 su cuenta de referencia.
    + Copia los campos del inodo de disco al inodo en memoria.
    + Devuelve el inodo bloqueado.

- Si lista de libres = ∅ ⇒ Error. Tratamiento diferente al realizado por el *buffer caché*.

- Si el inodo está en la tabla de inodos:
  - El proceso A lo encuentra en lista hash, y mira si está bloqueado por el proceso B.
  - Si el inodo está bloqueado:
    + El proceso A duerme y señala con un indicador (flag) que en el inodo que espera, esté libre.
    + El proceso B desbloquea el inodo y despierta los procesos dormidos en dicho evento.
  - El proceso A consigue al fin el inodo requerido.
    + Lo bloquea y otros procesos no puedan acceder a él.
    + Si cuenta de referencia = 0 ⇒ el inodo estaba en la lista de libres ⇒ *kernel* lo quita de ahí (lista de inodos libres).
    + El *kernel* incrementa la cuenta de referencia y devuelve inodo bloqueado.

4.3.2.2.2. Acciones que Realiza el Kernel cuando Libera un inodo: iput.

- Decrementa en 1 la cuenta de referencia para el inodo.
- Si la cuenta de referencia = 0:
  - Si la copia en memoria es diferente a la copia en disco ⇒ el *kernel* escribe el inodo a disco.
  - Coloca el inodo en la lista de libres.
  - Si n° enlaces del archivo = 0 ⇒ libera los inodos y los bloques datos en disco.

4.3.3. Estructura de los Archivos.

4.3.3.1. Archivos Regulares.

- Los archivos regulares contienen bytes de datos organizados como un array lineal. Las operaciones que se pueden realizar con los datos de un archivo son las siguientes:
  - Leer o escribir cualquier byte del archivo.
  - Añadir bytes al final del archivo, con lo que aumentan su tamaño.
  - Truncar el tamaño de un archivo a cero bytes. Esto es como si borrásemos el contenido del archivo.
- Hay que tener presente que las siguientes operaciones no están permitidas con los archivos:
  - Insertar bytes en un archivo, excepto al final. No hay que confundir la inserción de bytes con la modificación de los que ya existen (proceso de escritura).
  - Borrar bytes de un archivo. No hay que confundir el borrado de bytes con la puesta a cero de los que ya existen.
  - Truncar el tamaño de un archivo a un valor distinto de cero.
- Dos o más procesos pueden leer y escribir concurrentemente sobre un mismo archivo. Los resultados de esta operación dependerán del orden de las llamadas de entrada/salida individuales de cada proceso y de la gestión y planificación que el *scheduler* haga de los procesos, y en general son impredecibles. Los archivos regulares, como tales, no tienen nombre y el acceso a ellos se realiza a través de los inodos.
- Los bloques de datos están situados a partir de la lista de inodos. Cada inodo tiene unas entradas (direcciones) para localizar dónde están los datos de un archivo en el disco. Como cada bloque del disco tiene asociada una dirección, las entradas de direcciones consisten en un conjunto de direcciones de bloques del disco.
- Si los datos de un archivo se almacenasen en bloques consecutivos, para poder acceder a ellos nos bastaría con conocer la dirección del bloque inicial y el tamaño del archivo. Sin embargo, esta política de gestión del disco va a provocar un desaprovechamiento del mismo, ya que tenderán a proliferar áreas libres demasiado pequeñas para poder ser utilizadas. Por ejemplo, supongamos que un usuario crea tres archivos A, B y C, cada uno de los cuales ocupa 10 bloques en el disco. Supongamos que el sistema sitúa estos tres archivos en bloques contiguos del disco. Si el usuario necesita añadir 5 bloques al archivo central (archivo B) porque ha aumentado de tamaño, el sistema va a tener que buscar 15 bloques contiguos que se encuentren libres para copiar el archivo B en esa zona, dejando libre los 10 primeros bloques que tenia asignados.

- La situación anterior plantea dos inconvenientes:
  - El primero es el tiempo que se pierde en copiar los 10 bloques del archivo B que no varían de contenido.
  - El segundo es que los 10 bloques que quedan libres sólo pueden contener archivos con un tamaño inferior o igual a 10 bloques, provocando a la larga microfragmentación del disco que lo puede dejar inservible.

- El *kernel* puede minimizar la fragmentación del disco ejecutando periódicamente procesos para compactarlo, pero esto produce una degradación de las prestaciones del sistema en cuanto a velocidad.

- Para obtener una mayor flexibilidad, el *kernel* reserva los bloques para los archivos de uno en uno, y permite que los datos de un archivo estén esparcidos por todo el sistema de archivos. Este esquema de reserva va a complicar la tarea de localizar los datos.

- Las entradas de direcciones del inodo van a consistir en una *Tabla de Direcciones* de los bloques que contiene los datos del archivo. Un cálculo sencillo y simple nos muestra que almacenar la lista de bloques del archivo en un inodo es algo difícil de implementar. Por ejemplo, si los bloques son de 1 Kbyte y el archivo ocupa 10 Kbytes (10 bloques), vamos a necesitar almacenar 10 direcciones en el inodo. Pero si el archivo es de 100 Kbytes (100 bloques), necesitaremos 100 direcciones para acceder a todos los datos. Así pues, esta gestión nos impone que el tamaño del inodo sea variable, ya que si éste es fijo (valor concreto), estamos fijando el tamaño máximo de los archivos que podemos manejar. Debido a lo poco manejable que resulta, la idea de inodos de tamaño variable es algo que no implementa ningún sistema.

- Para conseguir que el tamaño del un inodo sea pequeño y a la vez podamos manejar archivos grandes, las entradas de direcciones de un inodo se ajustan al esquema que muestra la figura 4.3.
  - 10 Entradas directas.
  - 1 Entrada indirecta simple.
  - 1 Entrada indirecta doble.
  - 1 Entrada indirecta triple.

- En el UNIX System V, los inodos tienen una tabla de direcciones con 13 entradas. Las 10 entradas marcadas como directas en la Figura 4.3 contienen direcciones de bloques en los que hay datos del archivo. La entrada marcada cono indirecta simple direcciona un bloque de datos que contiene una tabla de direcciones de bloque de datos. Para acceder los datos a través de una entrada indirecta, el *kernel* debe leer el bloque cuya dirección nos indica la entrada indirecta y buscar en él la dirección del bloque donde realmente está el dato, para a continuación leer el bloque y acceder al dato. La entrada marcada como indirecta doble contiene la dirección de un bloque cuyas entradas actúan como entradas indirectas simples, y la entrada marcada como indirecta triple direcciona un bloque cuyas entradas son entradas indirectas dobles.

- Este método puede extenderse para soportar entradas indirectas cuádruples, quíntuples, etc., pero en la práctica tenemos más que suficiente son una indirección triple.

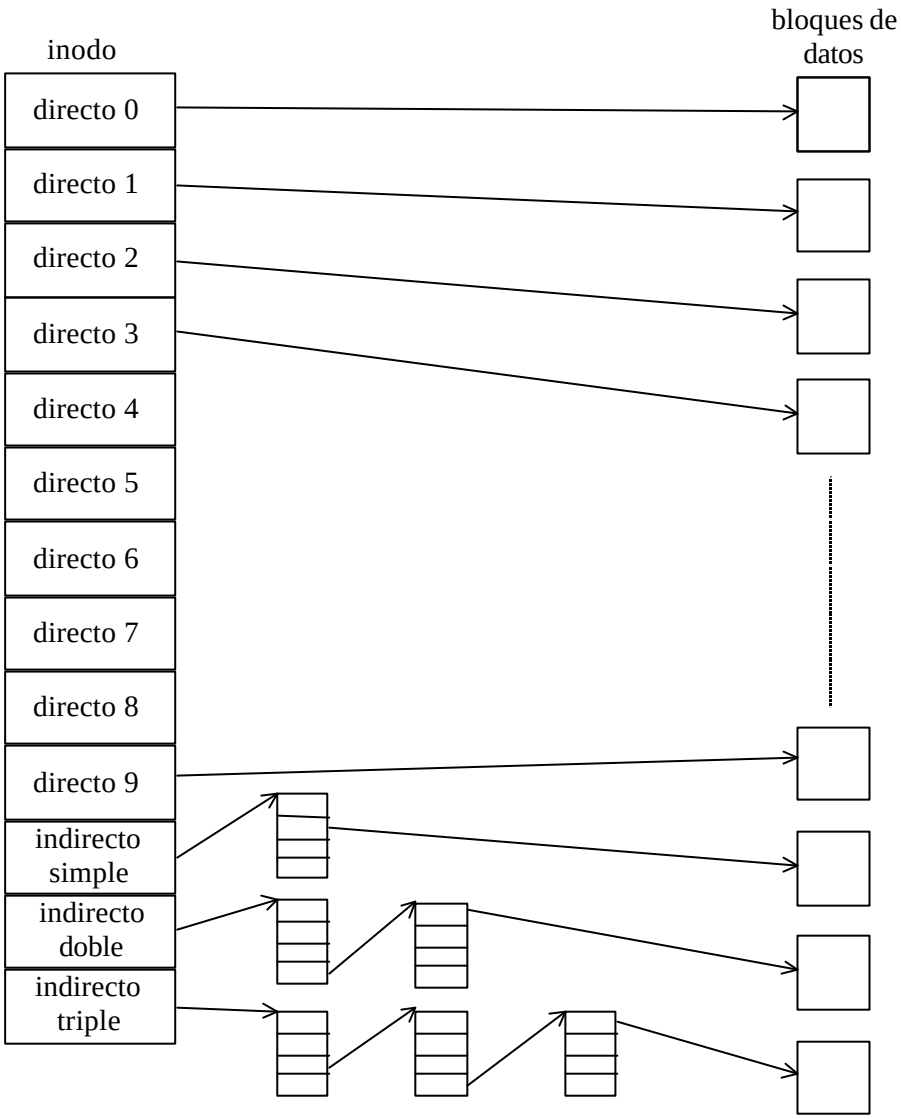


Figura 4.3. Tabla de direcciones de bloque de un inodo.

- Vamos a ver el caso práctico en el que los bloques son de 1 Kbytes y el bloque se direcciona con 32 bits (2^{32} = 4 GigaDirecciones posibles). En esta situación, un bloque de datos puede almacenar 256 direcciones de bloque y un archivo podría llegar a tener un tamaño del orden de 16 GigaBytes, según se indica en la siguiente tabla:

Tipo de Entrada	**Total de bloques accesibles**	**Total de bytes accesibles**
10 entradas directas	10 bloques de datos	10 Kbytes
1 entrada indirecta simple	1 bloque indirecto simple **(Þ)** 256 bloques de datos	256 Kbytes
1 entrada indirecta doble	1 bloque indirecto doble **(Þ)** 256 bloque indirecto simple **(Þ)** 65.536 bloques de datos	64 Mbytes
1 entrada indirecta triple	1 bloque indirecto triple **(Þ)** 256 bloque indirectos dobles **(Þ)** 65.536 bloques indirectos simples **(Þ)** 16.777.216 bloques de datos	16 Gbytes

Tabla 4.1. Capacidad de direccionamiento de los bloques de direcciones de un inodo (bloques = 1Kbyte).

- Teniendo en cuenta que el campo *tamaño del archivo* de inodo es de 32 bits, el tamaño máximo de un archivo no es del orden de 16 Gbytes, como indica la tabla anterior, si no de 4 Gbytes. Este tamaño, en la práctica es más que suficiente.
- Los procesos acceden a los datos de un archivo indicando la posición independientemente de la representación, con respecto al inicio del archivo, del byte que queremos leer o escribir. El archivo es visto por el proceso como un secuencia de bytes que empieza en el byte número 0 y llega al byte cuya posición, con respecto al inicial, coincide con el tamaño del archivo menos 1. El *kernel* se encarga de transformar las posiciones de los bytes, tal y como las ve el usuario, a direcciones de los bloques de disco y por ello, podemos ver a un archivo como un conjunto de bloques.
- Traducción (bmap), transformar las posiciones de los bytes en bloques de disco.
  - Objetivo: Dado un inodo + desplazamiento $\Rightarrow$ nº de bloque en el sistema de archivos + desplazamiento dentro del bloque + bytes de E/S en el bloque.
    + Calcular el número de bloque lógico en el archivo.
    + Calcular el byte de comienzo en el bloque para la E/S.
    + Calcular el número de bytes a copiar para el usuario.
    + Determinar el nivel de indirección.
    + Mientras no se esté en el nivel de indirección adecuado:
      * Calcular el índice en el inodo o el bloque indirecto a partir del nº de bloque lógico del archivo.
      * Obtener el nº de bloque de disco del inodo o del bloque indirecto.
      * Liberar el buffer de la lectura previa, si existe (algoritmo brelse).
      * Si no hay más niveles de indirección $\Rightarrow$ Devolver el nº de bloque.
      * Leer el bloque de disco (algoritmo bread).
      * Ajustar el nº de bloque lógico en el archivo de acuerdo al nivel indirección.
- Vamos a ver un ejemplo para aclarar estos conceptos. Supongamos un archivo cuyos datos están en los bloques que nos indican las entradas de direcciones del inodo descrito en la Figura 4.4.

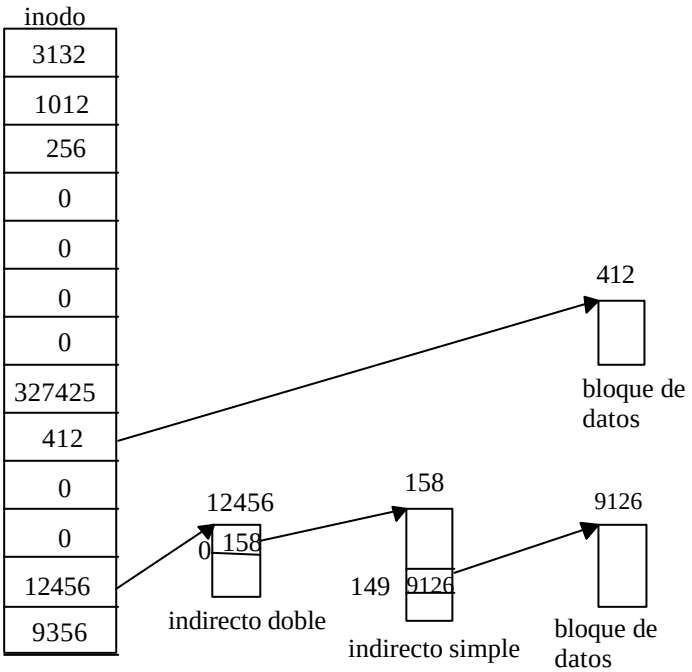


Figura 4.4. Ejemplo de tabla de direcciones de un inodo.

- Vamos a seguir suponiendo que el bloque tiene un tamaño de 1024 bytes. Si un proceso quiere acceder a un byte que se encuentra en la posición 9125 de archivo, el *kernel* calcula que ese byte en el bloque número 8 del archivo (empezando a numerar los bloques lógicos del archivo desde 0).

$$bloque_{archivo} = \lfloor desplazamiento_{archivo} / tamaño_{bloque} \rfloor = \lfloor 9125\ _{bytes} / 1024\ _{bytes/bloque} \rfloor = 8\ _{bloques}$$

- Para ver a qué bloque del disco corresponde el bloque número 8 del archivo, hay que consultar el número de bloque que almacena la entrada número 8 (entrada de dirección directa) de la tabla de direcciones del inodo. En el caso de la figura ejemplo, el bloque de disco buscado es el 412. Dentro de este bloque, el byte 9125 del archivo se corresponde con el byte 933 con respecto al inicio del bloque (los bytes del bloque se numeran desde 0 a 1023).

  $desplazamiento_{bloque} = desplazamiento_{archivo} \% tamaño_{bloque} = 9125\ _{bytes} \% 1024\ _{bytes/bloque} = 933\ _{bytes}$

- En el ejemplo anterior el cálculo ha sido sencillo porque el byte buscado era accesible desde una entrada directa del inodo.
- Vamos a ver lo que ocurre si queremos localizar en el disco el byte que se encuentra en la posición 425000 del archivo. Su calculamos su bloque lógico de archivo, que se encuentra en el bloque 415,

  $bloque_{archivo} = \lfloor desplazamiento_{archivo} / tamaño_{bloque} \rfloor = \lfloor 425000\ _{bytes} / 1024\ _{bytes/bloque} \rfloor = 415\ _{bloques}$

- El total de bloques al que podemos acceder con las entradas directas es de 10. con la entrada indirecta simple podemos acceder a 256 bloques, y el la indirecta doble, a 65536. Luego el byte buscado estará direccionado por la entrada indirecta doble.
- A esta entrada pertenecen los bloques comprendidos entre los número lógicos de archivo 266 y 65581 (ambos inclusive) y el bloque buscado es el 415. Si nos fijamos en la figura ejemplo, el número de bloque que contiene la entrada indirecta doble es el 12456, que es el bloque de disco donde se encuentran las direcciones de los bloques con entradas indirectas simples.
- La entrada número 0 del bloque indirecto doble nos da acceso a los bloques de archivo comprendidos entre 266 y el 521, ya que cada entrada actúa de indirección simple y da acceso a 256 bloques de datos. En la entrada 0 vemos que el número de bloque de disco del bloque indirecto simple que buscamos es 158. Dentro del bloque indirecto simple, la entrada que nos interesa es la diferencia entre 415 (bloque lógico del archivo) y 266 (bloque inicial al que da acceso el bloque indirecto doble); por lo tanto el resultado es 149. Según la figura ejemplo, la entrada 149 del bloque de disco 158 contiene el número 9126. Es el bloque de disco 9126 donde se encuentra el dato que buscamos, y en el byte 40 de este bloque está el byte 425000 de nuestro archivo.

  $desplazamiento_{bloque} = desplazamiento_{archivo} \% tamaño_{bloque} = 425000\ _{bytes} \% 1024\ _{bytes/bloque} = 40\ _{bytes}$

- Si observamos la Figura 4.4 con más detalle, podemos ver que hay algunas entradas del inodo que están a 0. Esto significa que no referencian a ningún bloque del disco y que los bloques lógicos correspondientes del archivo no tienen datos. Esta situación se da cuando se crea un archivo y nadie escribe en los bytes correspondientes a estos bloques, por lo que permanecen en su valor inicial 0. Al no reservar el sistema bloques de disco para estos bloques lógicos, se consigue un ahorro de recursos del disco. Imaginemos que creamos un archivo y sólo escribimos un byte en la posición 1048276, esto significa que el archivo tiene un tamaño de 1 Mbyte. Si el sistema reservase bloques de disco para este archivo en función de su tamaño y no en función de los bloques lógicos que realmente tiene ocupados, nuestro archivo ocuparía 1024 bloques de disco en lugar de 1, como ocupa en realidad.
- La conversión de un desplazamiento de byte de archivo a su dirección de disco es un proceso laborioso, sobre todo si se ve involucrada una indirección triple. En este caso, el *kernel* necesita acceder a bloques a tres bloques de direcciones, además de al inodo y al bloque de datos. Aun en el caso de que el *kernel* encuentre los bloques en el *buffer caché*, la operación sigue siendo costosa porque debe hacer varios accesos al *byffer caché* y puede que tenga que esperar a que algunos *buffers* queden desbloqueados. La efectividad de este algoritmo queda limitada en la práctica por la frecuencia de uso de los archivos grandes.

4.3.3.2. Directorios.

- Los directorios son los archivos que nos permiten darle una estructura jerárquica a los sistemas de archivo de UNIX. Su función fundamental consiste en establecer la relación que existe entre el nombre de un archivo y su inodo correspondiente.
- En el UNIX System V, un directorio es un archivo cuyos datos están organizados como una secuencia de entradas, cada una de las cuales tiene un número de inodo y el nombre de un archivo que pertenece al directorio. Al par (inodo , nombre_del_archivo) se le conoce como enlace (link) y puede haber varios nombres de archivos, distribuidos por la jerarquía de directorios, que estén enlazados con un mismo inodo.
- El tamaño de cada entrada del directorio es de 16 bytes, dos dedicados al inodo y 14 dedicados al nombre del archivo. En la siguiente figura podemos ver la estructura típica de un directorio.

Desplazamiento	Número de inodo (*inode*)	Nombres de archivos
0	83	.
16	2	..
32	1798	init
48	1276	fsck
64	85	clri
80	1268	motd
96	1799	mount
112	88	mknod
128	2114	passwd
144	1717	umount
160	1851	checklist

Figura 4.5. Ejemplo de estructura de un directorio para el UNIX System V.

- Las dos primeras entradas de un directorio reciben los nombres de "." y "..". El archivo "." tiene asociado el inodo correspondiente al directorio actual y al archivo ".." se le asocia el inodo del directorio padre del actual. Estas dos entradas están presentes en todo directorio, y en el caso del directorio "/" (raíz), el programa mkfs (make file system, programa mediante el cual se crea un sistema de archivos) se encarga de que el archivo ".." se refiera al propio directorio raíz.
- El *kernel* maneja los datos de un directorio con los mismo procedimientos con que se manejan los datos de los archivos regulares (ordinarios), utilizando la estructura inodo (*inode*) y los bloques de acceso directos e indirectos.
- Los procesos pueden leer el contenido de un directorio como si se tratase de un archivo de datos, sin embargo no pueden modificarlos. El derecho de escritura en un directorio está reservado al *kernel*. Esto es una medida de seguridad que se toma para evitar que una manipulación incorrecta de un directorio pueda destruir un sistema de archivos.
- No hay razones estructurales por las que no puedan existir múltiples enlaces a un directorio, al igual que ocurre con los archivos regulares (ordinarios). Si embargo, esto complicaría bastante la escritura de los programas que recorren el sistema de archivos, por lo que el *kernel* lo prohíbe.
- Los permisos de acceso a un directorio tiene los siguientes significados:
  - Permiso de lectura. Permite que un proceso pueda leer ese directorio.
  - Permiso de escritura. Permite a un proceso crear una nueva entrada en el directorio o borrar alguna ya existente. Esto deberá hacerlo el *kernel* a través de las llamadas: creat, mknod, link o unlink.
  - Permiso de ejecución. Autoriza a un proceso para buscar el nombre de un archivo dentro de un directorio.

4.3.3.3. Archivos Especiales.

- Los archivos especiales o archivos de dispositivo van a permitir a los procesos comunicarse con los dispositivos periféricos (discos, cintas, terminales, impresoras, redes, etc.). Hay dos tipos de archivos de dispositivo: archivos de dispositivo modo bloque y archivos de dispositivo modo carácter.

- Los archivos especiales en modo bloque se corresponden a dispositivos estructurados en bloques (array de bloques de tamaño fijo) y el *kernel* gestiona un buffer caché para acelerar la velocidad de transferencia de los datos. La transferencia de información entre el dispositivo y el *kernel* se efectúa con mediación del buffer caché y el bloque es la unidad mínima que se transfiere en cada operación de entrada/salida. El *buffer caché* se implementa vía software y no hay que confundirlo con las memorias caché de acceso rápido que disponen la mayoría de los computadores. Ejemplos típicos de dispositivos modo bloque son los discos y las unidades de cinta.

- En los archivos de dispositivo en modo carácter la información no se organiza según una estructura concreta y es vista por el *kernel*, o por el usuario, como una secuencia lineal de bytes. En la transferencia de datos entre el *kernel* y el dispositivo no participa el *buffer caché* y por lo tanto se va a realizar a menor velocidad. Ejemplos típicos de dispositivos modo carácter son los terminales serie y las líneas de impresora.

- Un mismo dispositivo físico puede soportar los dos modos de acceso: bloque y carácter, y de hecho esto suele ser habitual en el caso de los discos.

- Los módulos del *kernel* que gestionan la comunicación con los dispositivos se conocen como controladores de dispositivo. Lo normal es que cada dispositivo tenga su controlador, aunque puede haber controladores que manipulen a toda la familia de dispositivos con características comunes (por ejemplo, el controlador que manipula los terminales).

- El sistema también puede soportar dispositivos software (o pseudodispositivos) que no tienen asociados un dispositivo físico. Por ejemplo, si una parte de la memoria del sistema se gestiona como un dispositivo, los procesos que quieran acceder a esa zona de memoria tendrán que utilizar las mismas llamadas al sistema que hay para el sistema de archivos, pero sobre el archivo de dispositivo /dev/mem (archivo de dispositivo genérico para acceder a memoria). En esta situación, la memoria es tratada como un periférico más.

- Los archivos de dispositivo, al igual que el resto de los archivos, tienen asociado un inodo. En el caso de los archivos regulares o de los directorios, el inodo nos indica los bloques donde se encuentran los datos del archivo, pero en el caso de los archivos de dispositivo no hay datos a los que referenciar. En su lugar, el inodo contiene dos números conocidos como *major number* y *minor number*. El *major number* indica el tipo de dispositivo de que se trata (disco, cinta, terminal, etc.) y le *minor number* indica el número de unidad dentro del dispositivo. En realidad, estos número los utiliza el *kernel* para buscar dentro de una tablas (*block device switch table* y *character device switch table*)m una colección de rutinas que permiten manejar el dispositivo. Esta colección de rutinas constituyen realmente el controlador del dispositivo.

- Cuando se invoca a una llamada al sistema para realizar una operación de entrada/salida sobre un archivo especial, el *kernel* se encarga de llamar al controlador del dispositivo adecuado. Lo que ocurre a continuación es algo que compete al diseñador del controlador y es completamente transparente para el usuario.

- Naturalmente, nosotros también podemos convertirnos en diseñadores de controladores de dispositivo y añadir a una arquitectura hardware determinada, los dispositivos que necesitemos.

4.3.3.4. PIPES (Tuberías con Nombre).

- Una tubería con nombre es un archivo con una estructura similar a la de un archivo regular (archivo ordinario). La diferencia principal con éstos es que los datos de una tubería son transitorios. Esto quiere decir que una vez que un dato es leído de una tubería, desaparece de ella y nunca más podrá ser leído.

- Las tuberías se utilizan para comunicar procesos. Lo normal es que un proceso abra la tubería para escribir en ella y el otro para leer de ella. Los datos escritos en la tubería se leen en el mismo orden en el que fueron escritos (de ahí el nombre de *fifo*, *first in first out*). La sincronización del acceso a la tubería es algo de lo que se encarga el *kernel.*

- El almacenamiento de los datos en una tubería se realiza de la misma manera que en un archivo regular (ordinario), excepto que el *kernel* sólo utiliza las entradas directas de la tabla de direcciones del bloque del inodo de la tubería. Por lo tanto, una tubería con nombre podrá almacenar 10 Kbytes a los sumo (suponiendo bloques de tamaño 1024 bytes). Esto no supone un problema grave, ya que los datos de la tubería son transitorios; sin embargo, si la velocidad a la que se introducen datos es mayor que la velocidad de extracción, la tubería podría rebosar y perderse información.

- Los accesos de lectura/escritura en la tubería se realizan con las mismas llamadas que empleamos para los archivos regulares (open, close, read, write, etc.). Además, estos acceso son de tipo atómico (acceso indivisible en el sentido de que cuando un proceso está accediendo a la tubería para escribir, no se ve interrumpido por otro proceso que quiera realizar una operación de escritura sobre la misma tubería. Lo mismo ocurre con la operación de lectura. Naturalmente, cuando un proceso intenta leer datos de una tubería vacía, la operación puede quedar durmiendo mientras espera a que algún proceso escriba datos en la tubería), con lo que se garantiza el sincronismo en el caso de que varios procesos compitan concurrentemente por el uso de la misma tubería.

- En contraposición a las tuberías con nombre tenemos a las tubería sin nombre que no tienen asociado ningún nombre de archivo y que sólo existen mientras algún proceso está unido a ellas. Las tuberías sin nombre también actúan como un canal de comunicación entre dos procesos y tienen asociado un inodo temporal mientras existen. Un ejemplo típico de la tubería sin nombre es la que se crea desde el intérprete de órdenes a través del carácter de tubería "|". Por ejemplo $ls | sort –r. En esta línea de órdenes le indica al intérprete que debe arrancar dos proceso: uno para ejecutar "ls" (listar por pantalla el contenido del directorio actual) y otro para "sort –r" (filtro de texto que lee líneas de texto de la entrada estándar y las presenta en pantalla en orden alfabético inverso). Además, el intérprete debe crear una tubería sin nombre a la que se van a unir "ls" y "sort –r". "ls" va a dirigir su salida hacia la tubería y "sort" va a leer datos de la tubería.

- Las tubería sin nombre son creadas a través de la llamada "pipe" (desde un proceso), mientras que las tuberías con nombre se crean con la orden "mknod" (desde la línea de ordenes del sistema) o con la llamada "mknod" (desde un proceso).

4.3.3.5. Conversión de un Nombre (*pathname*) en un inodo (*inode*). La función *namei.*

- Desde el punto de vista del usuario, vamos a referenciar a los archivos mediante su *nombre* (*pathname*). Llamadas como open, chdir o link reciben como parámetro de entrada el nombre (*pathname)* de un archivo y su inodo (*inode*). El *kernel* es quien se encarga de transformar el nombre (*pathname*) de un archivo a su inodo (*inode*) correspondiente.
- El algoritmo que realiza la transformación (llamado *namei*) se encarga de analizar los componentes del nombre (*pathname*) y de leer los inodos (*indoes*) intermedios necesarios para verificar que se trata de un nombre (*pathname*) correcto y que el archivo realmente existe.

- Si el nombre (*pathname*) es absoluto, la búsqueda del inodo (*inode*) del archivo se iniciaría en el directorio raíz. Si el nombre (*pathname*) es relativo, la búsqueda se iniciaría en el directorio de trabajo actual que tiene asociado el proceso que va a acceder al archivo.
- A medida que se van recorriendo los inodos (*inodes*) intermedios, se van verificando los permisos para comprobar si el proceso tiene derecho a acceder a los directorios intermedios.
- De forma esquemática las acciones que realiza el *kernel* para la transformación del nombre (*pathname*) de un archivo a su inodo (*inode*) correspondiente, *namei*, son las siguientes:
  - Si el nombre (*pathname*) empieza en el directorio raíz ⇒ inodo (*inode*) de trabajo = inodo (*inode*) del directorio raíz.
  - Si el nombre (*pathname*) empieza en el directorio actual ⇒ inodo (*inode*) de trabajo = inodo (*inode*) del directorio actual.
  - Mientras no se encuentre el archivo:
    + Leer el siguiente componente del nombre (*pathname*), *(bmap, bread y brelse)*.
    + Verificar si el inodo (*inode*) de trabajo es un directorio y sus permisos.
    + Buscar linealmente dentro del directorio asociado al inodo (*inode*) de trabajo, el siguiente componente leído del nombre (*pathname*).
    + Si no se encuentra ⇒ No existe ese inodo (*inode*).
    + Si se encuentra ⇒ inodo (*inode*) de trabajo = obtener inodo (*inode*), utilizando para ello las funciones *iput* (liberar el inodo anterior) *iget* (obtener el nuevo inodo).
- Como ejemplo, vamos a suponer que un proceso quiere abrir el archivo "/etc/passwd" (archivo que tiene una entrada por cada usuario del sistema y donde, entre otra información, figura el *password* cifrado del usuario). Cuando el *kernel* emplea el análisis del nombre (*pathname*), encuentra que empieza por "/" y pasa a leer el inodo (*inode*) asociado al directorio raíz. Este inodo (*inode*) pasa a ser el inodo (*inode*) de trabajo y el *kernel* verifica si corresponde a un directorio y si el proceso tiene permiso para buscar archivos dentro de él. Suponiendo que los permisos están en regla, el *kernel* toma el siguiente elemento del nombre (*pathname*), que es "etc", y busca, dentro del directorio raíz, alguna entrada cuyo nombre de archivo sea "etc". Cuando el *kernel* la encuentra, libera el inodo (*inode*) del directorio raíz y lee el inodo (*inode*) del archivo "etc". En este inodo (*inode*) se refleja que "etc" es otro directorio y por lo tanto en él podemos buscar el archivo "passwd" (tercer elemento del nombre (*pathname*)). El *kernel*, tras verificar los permisos de acceso a "etc", repite el mismo proceso de búsqueda para el nombre del archivo "passwd" y tras encontrarlo, verificar que es un archivo y no un directorio (como se esperaba a la vista de su nombre (*pathname*)), y que tenemos permiso para acceder a él, nos habilita una estructura para poder trabajar con ese archivo.

4.3.3.6. Asignar un inodo (*inode*) Libre para un Nuevo Archivo. La Función *ialloc*.

- En líneas generales el proceso que sigue el *kernel* para la asignación de inodos libre a un nuevo archivo se describe en el siguiente algoritmo:

```
algoritmo ialloc          // Entrada: Sistema de archivos, Salida: inodo Bloqueado
{
    while (True)
    {
        if (SuperBloque esté Bloqueado)
        {
            sleep(evento: Espera por SuperBloque);
            continue;
        }
        if (ListaEnSuperBloque está vacía)
        {
            BLOQUEA SuperBloque;
            // Desde el recordado en disco, hasta completar lista o no hay en Disco.
            // Usar bread y brelse sucesivamente
            CargaNodosILibres();
            DESBLOQUEA SuperBloque;
            wakeup(evento: Espera por SuperBloque);
            if (NoHayNodosIEnDisco)
                return NO_HAY_NODOI;
            Actualizar contadores;
        }
        // Ya hay inodos en el SuperBloque
        Tomar siguiente inodo de la lista;
        Recuperar inodo desde disco (iget);
        if (inodo No estaba Libre)
        {
            // Condición de concurso
            Escribir inodo a Disco;
            Liberar inodo (iput)
            continue;
        }
        // El inodo está disponible
        Inicializa inodo;
        Escribe el inodo a disco;
        Actualiza contadores;
        return (inodo);
    }
}
```

- Situación de concurso: Un proceso que ha conseguido un inodo libre necesita comprobar que realmente esta disponible.
  - Procesos A, B y C:
  - *Kernel* a petición del proceso A asigna el inodo libre I. Bloquea al proceso A en la lectura del inodo (ialloc->iget->bread).
  - El proceso B solicita un inodo. *Kernel* encuentra la lista vacía (en SuperBloque) e intenta recargarla. Si recarga desde un iRec < I (al liberar inodos > iRec, no se actualiza), entonces *kernel* encontrará I disponible y lo llevará a su lista
  - Mas tarde el proceso C encontrará I disponible ($\Rightarrow$ control adicional).

4.3.3.7. Asignar Bloques Libres (de disco) para Nuevos Datos. La Función *alloc*.

- En líneas generales el proceso que sigue el *kernel* para la asignación de bloques (de disco) para nuevos datos se describe en el siguiente algoritmo:

```
algoritmo ialloc        // Entrada: Número del Sistema de Archivos, Salida: Buffer para el nuevo bloque
{
    while (SuperBloque Bloqueado)
        sleep(evento: espera por SuperBloque);
    B = Siguiente número de bloque de ListaBloquesLibres;
    if (B era el último bloque de la ListaBloquesLibres)
    {
        Bloquear el SuperBloque;
        LeerBloque B (bread);
        Copiar bloque B como ListaBloquesLibres;
        Liberar buffer usado para leer bloque B (brelse);
        Desbloquear el SuperBloque;
        wakeup(evento: espera por SuperBloque);
    }
    Obtener un buffer para el bloque B (getblk);
    Limpiar el buffer;
    Decrementar contador de bloques libres;
    Marcar SuperBloque Modificado;
    Return buffer;
}
```

4.3.4. Tablas de Control de Acceso a los Archivos.

- Estructuras de datos: Tabla de inodos, tabla de archivos y tabla de descriptores de archivos.

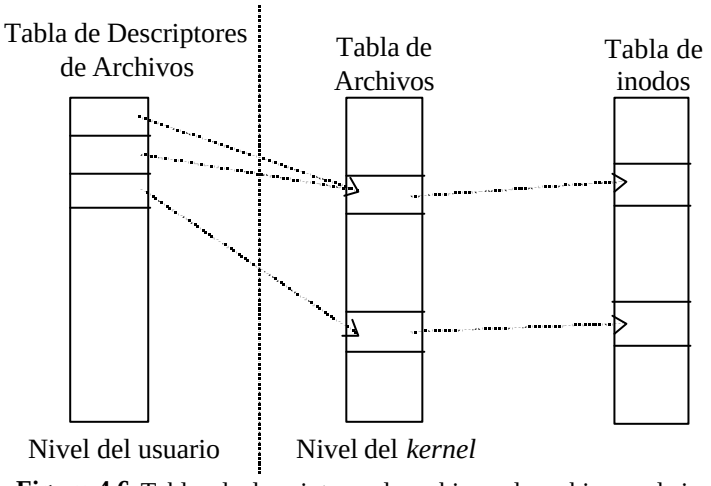


Figura 4.6. Tablas de descriptores de archivos, de archivos y de inodos.

- La *Tabla de inodos*, es una copia en memoria de la lista de inodos que hay en el disco, a la que se le añade información adicional. Esta tabla se copia en disco para conseguir un acceso más rápido.
- La *Tabla de archivos*, es una estructura global del *kernel* y en ella hay una entrada por cada archivo distinto que los procesos del *kernel* o los procesos del usuario tienen abiertos. La tabla de archivos es una estructura de datos orientadas a objetos. Cada vez que un proceso abre o crea un archivo nuevo, se reserva una nueva entrada en la tabla. Cada entrada de la tabla contiene un bloque de datos y un array de punteros a funciones que traducen las operaciones genéricas que se pueden realizar con los archivos (leer, escribir, cerrar, posicionar, ...) en acciones concretas asociadas a cada tipo de archivo. Las operaciones que se deben implementar para cada tipo son:
  - Funciones para leer (read) y escribir (write) en un archivo.
  - Una función para realizar la multiplexación síncrona de la entrada salida (select).
  - Una función para controlar los modos de entrada/salida (ioctl).
  - Una función para cerrar un archivo (close).

  Se puede observar que no hay ninguna rutina de apertura de archivos en esta definición de objeto. Esto es debido a que el sistema sólo empieza a tratar a los elementos de esta tabla como objetos a partir de que se haya abierto el archivo. Cada entrada de la tabla de archivos tiene también un puntero a una estructura de datos que contiene información del estado actual del archivo. Entre otros, se deben tener en cuenta los siguientes campos:
  - *inode* asociado a la entrada de la tabla.
  - *offsets* de lectura y escritura que indican sobre qué byte del archivo van a tener efecto las siguientes operaciones de lectura o escritura. Estos *offsets* quedan actualizados cada vez que se realiza una de las operaciones anteriores.
  - Permisos de acceso para el proceso que ha abierto el archivo.
  - *flags* del nodo de apertura del archivo, que se verificarán cada vez que se realice una operación sobre el mismo para ver si es congruente. Por ejemplo, si un archivo se abre para leer solamente, el *kernel* no va a permitir que se realicen operaciones de escritura sobre él.
  - Un contador para indicar cuántas entradas de la *tabla de descriptores de archivos* tiene asociadas esta entrada de la tabla de archivos.
- La *Tabla de descriptores de archivos* es una estructura local a cada proceso. Esta tabla identifica todos los archivos abiertos por un proceso. Cuando utilizamos las llamadas *open*, *creat*, *dup* o *link*, el *kernel* devuelve un descriptor de archivo, que es un índice para poder acceder a las entradas de la tabla anterior (*tabla de archivos*). En cada una de las entradas de la tabla hay un puntero a una entrada de la tabla de archivos del sistema.

  Los procesos no van a manipular directamente ninguna de las tablas anteriores (esto lo hace el *kernel*), sino que van a acceder a los archivos manipulando su descriptor asociado, que es un número. Cuando un proceso invoca una llamada para realizar una operación sobre un archivo, le va a pasar al *kernel* el descriptor de ese archivo. El *kernel* va a usar este número para acceder a la tabla de descriptores de archivos del proceso y buscar en ella cuál es la entrada de la *tabla de archivos* que le da acceso a su inodo (*inode*). Este mecanismo puede parecer artificioso, pero va a ofrecer una gran flexibilidad cuando queramos que un proceso acceda simultáneamente a un mismo archivo en modos distintos, o que varios procesos compartan archivos.

  El tamaño de la tabla de descriptores de archivos tiene un valor limitado para cada proceso. Este valor depende de la configuración del sistema (*limits*, constante OPEN_MAX), aunque está bastante extendido que sea 20. Así, un proceso no puede tener abiertos más de 20 archivos distintos en un instante de tiempo determinado. Si algún proceso necesita manipular más de 20 archivos, hemos de tener presente que algunos deberán estar cerrados para poder manipular otros, ya que todos no pueden estar abiertos simultáneamente. Una situación semejante a ésta se plantea cuando intentamos compilar un proyecto y consta de más de 20 archivos distribuidos entre archivos de cabecera, archivos fuentes, archivos objeto y librerías. Aunque el compilador debe de poder generar el archivo ejecutable sin imponer límite al número de archivos de nuestro proyecto.

  Cuando se arranca un proceso en UNIX, el sistema abre para él, por defecto, tres archivos que van a ocupar las tres primeras entradas de la tabla de descriptores de archivos. Estos archivos se conocen como *archivo estándar de entrada* (*stdin*), que tiene asociado el descriptor número 0; el *archivo estándar de salida* (*stdout*), que tiene asociado el descriptor número 1; y el *archivo estándar de errores* (*stderr*), que tiene asociado el descriptor número 2.

4.4. LLAMADAS AL SISTEMA PARA EL SISTEMA DE ARCHIVOS DE UNIX.

- Después de exponer la estructura interna del sistema de archivos UNIX, vamos a estudiar la interfaz que ofrece el sistema operativo para comunicarnos con el *kernel* y poder acceder a los recursos del sistema de archivos.
- Las llamadas al sistema para el sistema de archivos implican el acceso a datos del archivo y a estructuras de datos del archivo en disco.
- Estructuras de datos en memoria:
  - Tabla de descriptores ⇒ Puntero a tabla de archivos.
  - Tabla de archivos:
    + Puntero a la tabla de inodos (*inode*).
    + Cuenta de referencia.
    + Modo de apertura.
    + Desplazamiento (*offset*).
  - Tabla de inodos (*inode*).

- Clasificación de las llamadas: Figura 4.7.
  - Devuelven descriptores de archivo para su uso en posteriores llamadas.
  - Utilizan el algoritmo *namei.*
  - Asignan y liberan inodos, utilizando los algoritmos *ialloc* e *ifree*.
  - Manipulan los atributos de un archivo.
  - Realizan operaciones de E/S para un proceso, utilizando los algoritmos *alloc*, *free* y los vistos para los buffers (*buffer caché*).
  - Cambian la estructura del sistema de archivos.
  - Permiten a un proceso cambiar su visión de la estructura del sistema de archivos.

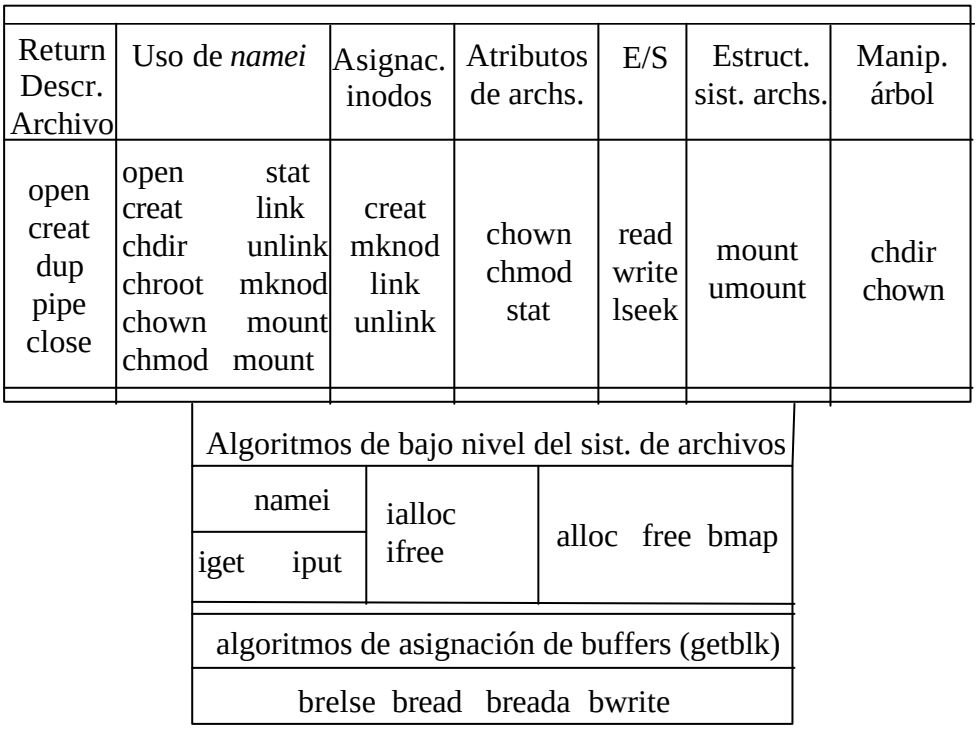


Figura 4.7. Llamadas al sistema de archivos y su relación con otros algoritmos.

4.4.1. open (Apertura de un Archivo).

- Permite el acceso a datos del archivo.
- Paso previo ⇒ acceso al inodo ⇒ Traer copia a memoria.
- descriptor = **open**(nombre, flags, modos)
  - Descriptores estándar.
  - Búsqueda del descriptor en la tabla de descriptores de archivos.
- Acciones (algoritmo):
  - El *kernel* busca en el sistema de archivos el archivo pasado como parámetro: *namei.*
  - Si el archivo no existe o no se tienen permisos ⇒ ERROR.
  - Obtiene copia en memoria del inodo.
  - Asigna una entrada en la tabla de archivos.
    + Puntero al inodo.
    + desplazamiento = 0 o bien desplazamiento = tamaño del archivo.
  - Si modo es igual a O_TRUNC, entonces liberar los bloques (disco) del archivo y establecer el tamaño del archivo (en inodo) a cero.
  - Asigna una entrada en la tabla de descriptores de archivos de usuario.
- Ejemplo.
  - Un proceso ejecuta el siguiente código:

    ```
    fd1 = open("/etc/passwd", O_RDONLY);
    fd2 = open("local", O_RDWR);
    fd3 = open("/etc/passwd", OWRONLY);
    ```

    + Resultado: Figura 4.8.
  - A continuación, otro proceso ejecuta el siguiente código:

    ```
    fd1 = open("/etc/passwd", O_RDONLY);
    fd2 = open("private", O_RDONLY);
    ```

    + Resultado: Figura 4.9.

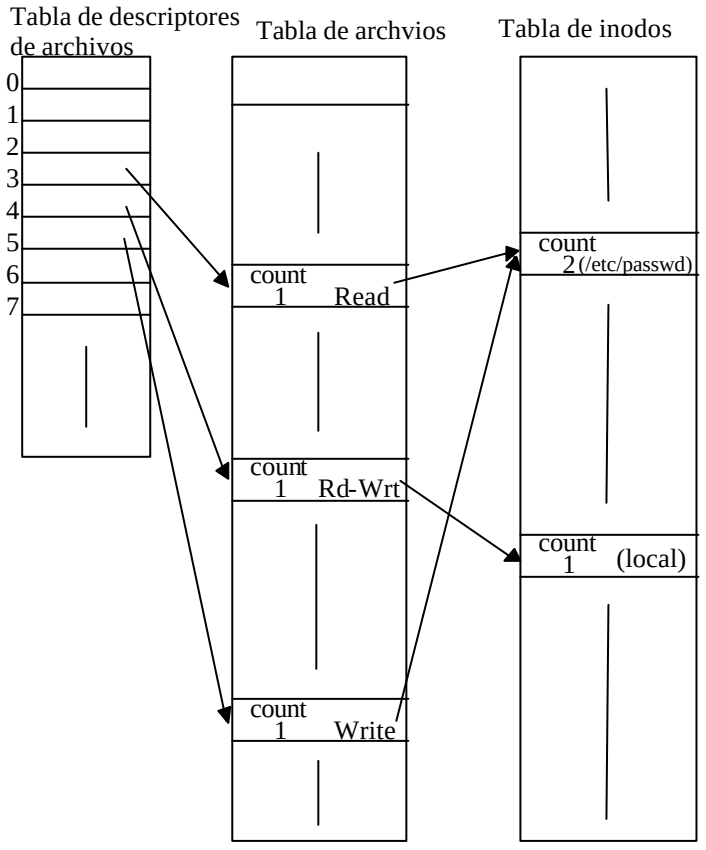


Figura 4.8. Estructuras de datos después de los **open**.

- Modos *(fcntl.h)*:
  - O_RDONLY ⇒ Solo lectura; O_APPEND ⇒ Añadir al final.
  - O_WRONLY ⇒ Solo escritura; O_TRUNC ⇒ Truncamiento.
  - O_RDWR ⇒ Lectura/Escritura; O_EXCL ⇒ Apertura exclusiva.
  - O_CREAT ⇒ Creación; O_NDELAY ⇒ s/espera (pipes).

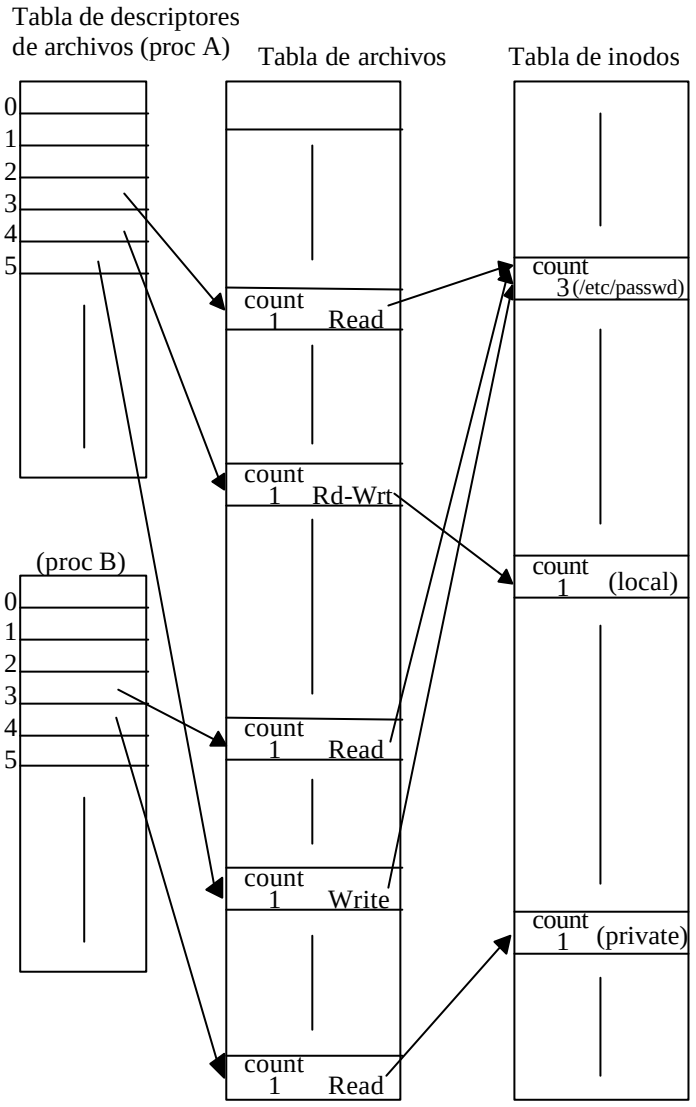


Figura 4.9. Estructuras de datos después de que dos procesos abran archivos (**open**).

4.4.2. read (Lectura de Datos de un Archivo).

- nº_bytes_leídos = **read**(descriptor, buffer, número_bytes_a_leer)
- Acciones:
  - Obtener la entrada en la tabla de archivos a partir del descriptor del archivo.
  - Comprobar los permisos.
  - Obtener el inodo (*inode*) siguiendo los punteros.
  - Hasta el fin de la lectura o error o no quedan bytes en el archivo:
    + Desplazamiento ⇒ nº de bloque de disco (*bmap*).
    + Calcula el desplazamiento en el bloque para comenzar la E/S.
    + Calcular el nº de bytes a leer dentro del bloque.
    + Si el nº bytes a leer = 0 ⇒ Fin de búsqueda.
    + Si no:
      * Leer el bloque en un buffer (*buffer caché*).
      * Copiar los datos del buffer ⇒ dirección destino del proceso de usuario.
      * Actualiza el desplazamiento en el archivo, dirección en el proceso de usuario, nº de bytes que quedan por leer.
  - Actualizar el desplazamiento en la tabla de archivos.
- Observaciones:
  - El control de acceso se realiza en cada llamada pues los permisos pueden cambiar y además el *kernel* no tiene forma de saber que el proceso está accediendo a un archivo permitido.
  - Se usa el u-Area para algunas variables temporales a fin de evitar pasarlas como parámetros en las funciones.
  - El inodo permanece bloqueado por el *kernel* durante la lectura para mantener la coherencia puesto que el proceso podría ser dormido durante la llamada read (por ejemplo en el acceso a los bloques de disco) y otro proceso podría modificar el inodo (lo importante es que el *kernel* sea coherente durante toda la llamada, aunque no se pueda asegurar coherencia de datos entre llamadas si por ejemplo otro proceso esta escribiendo datos en el mismo archivo). El *kernel* no puede bloquear el inodo entre sucesivas llamadas porque nada asegura que el proceso lo soltará en un tiempo razonable. Para asegurar coherencia entre sucesivas llamadas se puede disponer de file y record locking.
  - La utilización de *bread* o *breada* se decide en base al comportamiento del proceso. Es decir, el *kernel* determina si es posible la lectura anticipada. Si el proceso lee dos bloques secuencialmente ⇒ el *kernel* asume que futuras lecturas también serán secuenciales.
  - El fin de archivo (EOF) se detecta en la siguiente lectura al intentar leer las allá de lo que hay grabado.
  - El buffer intermedio (del buffer caché) se libera en cada iteración por lo que si un bloque se lee a trozos es posible que en el siguiente acceso haya que repetir el acceso a disco. Por último, hay que destacar que nada impide a un proceso acceder al mismo archivo a través de dos descriptores de archivos (fd) diferentes que apuntarán a dos entradas en la tabla de archivos también diferentes (cada una con su desplazamiento de acceso respectivo), aunque ambas apuntando a un único inodo.
  - La lectura de un bloque vacío intermedio no modifica el archivo y se sirve desde los buffers del *kernel.*

4.4.3. write (Escritura de Datos en un Archivo).

- nº_bytes_escritos = **write**(descriptor, buffer, nº_bytes_a_escribir)

- Acciones (algoritmo) similares a la operación de lectura (*read*).

- Diferencias:
  - Si archivo no contiene el bloque de desplazamiento a escribir ⇒ el *kernel* obtiene un bloque nuevo y asigna el nº a su posición de tabla de inodos (*inode*).
  - Si el desplazamiento ⇒ bloque indirecto ⇒ el *kernel* obtiene los bloques necesarios.
  - Durante *write* ⇒ el inodo (*inode*) permanece bloqueado, aunque el contenido del inodo (*inode*) puede variar.
  - En cada iteración, determina si tiene que escribir:
    + Parte del bloque ⇒ lo lee de disco para no sobrescribir lo que no varía.
    + Bloque entero ⇒ no lo lee porque va a sobrescribir su contenido.
  - Escritura ⇒ bloque a bloque, pero el *kernel* utiliza escritura retardada para escribir los datos ⇒ deja los datos en el *buffer caché* para la posterior manipulación por parte de otros procesos.
  - Escritura retardada ⇒ especialmente indicada para pipes (tuberías).

4.4.4. close (Cierre de un Archivo).

- Utilizamos la llamada **close** para indicarle al *kernel* que dejamos de trabajar con un archivo previamente abierto. El *kernel* se encargará de liberar las estructuras que había montado para trabajar con el archivo.

- **close**(descriptor)

- Al cerrar un archivo, la entrada que ocupaba en la tabla de descriptores de archivos del proceso queda libre para que la pueda utilizar una llamada a **open**. Por otro lado, el *kernel* analiza la entrada correspondiente en la tabla de archivos del sistema y si el contador de referencias que tiene asociado ese archivo es 0 (esto quiere decir que no hay más procesos que estén unidos a esta entrada), esa entrada también se libera. Si un proceso no cierra los archivos que tiene abiertos, al terminar su ejecución el *kernel* analiza la tabla de descriptores de archivos asociada a dicho proceso y se encarga de cerrar los procesos que aún estén abiertos.

- Acciones (algoritmo):
  - Acceder a la tabla de archivos y decrementar la cuenta de referencia.
  - Si cuenta de referencia del inodo = 0 ⇒ liberar la entrada (iput(inodo)).
  - Acceder al inodo (*inode*) y decrementar cuenta de referencia.
  - Si la cuenta de referencia de la tabla de archivos = 0 ⇒ liberar el inodo (*inode*) en memoria ⇒ liberar la entrada en la tabla de descriptores de archivos.

- Ejemplo:
  - A continuación, otro proceso ejecuta el siguiente código:

    close(fd1);
    close(fd2);
    + Resultado: Figura 4.10.

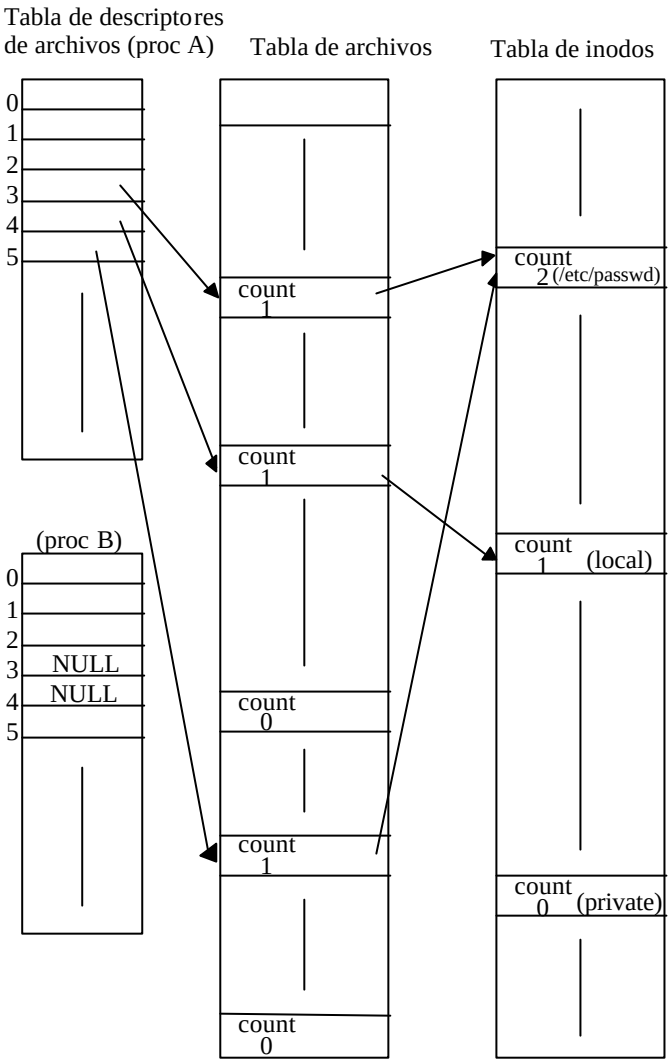


Figura 4.10. Tablas después de cerrar un archivo.

4.4.5. creat (Creación de un Archivo).

- Crea un archivo nuevo en el sistema de archivos. Tiene la misma funcionalidad que **open**.
- Semántica de la llamada:
  - Si el archivo no existía ⇒ se crea uno nuevo.
  - Si ya existía ⇒ trunca su contenido.
- descriptor = **creat**(nombre, modos)
- Acciones (algoritmo):
  - Obtener el inodo (*inode*) a partir del nombre (*pathname*).
  - Si el archivo no existía ⇒ Asignar el inodo (*inode*) libre y crear la entrada en el directorio padre si tenemos permisos de escritura en el directorio padre.
  - Si ya existía:
    - Si el acceso no está permitido ⇒ Retornar error.
    - Si el acceso está permitido:
      - Truncar el archivo si tenemos permiso de escritura sobre el archivo.
      - Liberar todos sus bloques de datos.
  - En cualquier caso:
    - Asignar una entrada en la tabla de archivos, cuenta de referencia = 1.
    - Asignar una entrada en la tabla de descriptores de archivos.

4.4.6. dup (Duplicado de un Descriptor).

- La llamada dup va a duplicar un descriptor de archivo que ya ha sido asignado y que está ocupando una entrada en la tabla de descriptores de archivos.
- **dup**(descriptor)
- La llamada a dup va a recorrer la tabla de descriptores y va a marcar como ocupada la primera entrada que encuentre libre, pasando a devolver el descriptor asociado a esa entrada. Si falla en su ejecución, devolverá un error.
- Los dos descriptores (original y duplicado) tienen en común que comparten el mismo archivo, por lo que a la hora de leer o escribir podemos utilizarlos indistintamente. La mayor utilidad de esta llamada es cuando utilizamos tuberías sin nombre.

4.4.7. lseek (Acceso Aleatorio y Posicionamiento en un Archivo).

- Con la llamada a lseek vamos a modificar el puntero (desplazamiento) de lectura/escritura de un archivo.
- **lseek**(descriptor, desplazamiento, desde)
- lseek modifica el puntero de lectura/escritura del archivo asociado al "descriptor" de la siguiente forma:
  - Si "desde" vale SEEK_SET, el puntero avanza "desplazamiento" bytes con respecto al inicio del archivo.
  - Si "desde" vale SEEK_CUR, el puntero avanza "desplazamiento" bytes con respecto a su posición actual.
  - Si "desde" vale SEEK_END, el puntero avanza "desplazamiento" bytes con respecto al final del archivo.
- Si "desplazamiento" es un número positivo, los avances deben entenderse en su sentido natural; es decir, desde el inicio del archivo hacia el final del mismo. Sin embargo se puede conseguir que el puntero retroceda pasándole a lseek un "desplazamiento" negativo.
- Cuando lseek se ejecuta satisfactoriamente, devuelve un número entero no negativo, que es el nuevo "desplazamiento" del puntero de lectura/escritura medido con respecto al principio del archivo. Si lseek falla, devuelve un error.
- Hay que tener presente que en algunos no está permitido el acceso aleatorio y por lo tanto la llamada lseek no tiene sentido. Ejemplos de estos archivos son los pipes (fifo) y los archivos de dispositivo en los que la lectura se realiza siempre a través de un mismo registro o posición de memoria.

4.4.8. fsync (Consistencia de un Archivo).

- La entrada con el disco se realiza a través del *buffer caché* para agilizar la transferencia de datos. Sin embargo, hay aplicaciones cuyas especificaciones obligan a que se prescinda del *buffer caché* y que las escrituras en un archivo se reflejen deforma inmediata en el disco. UNIX ofrece dos soluciones para este tipo de aplicaciones:
  - La primera consiste en abrir el archivo indicándole a open que toda operación posterior de escritura debe ir acompañada de una actualización en el disco. Esto se consigue pasándole a open, dependiendo del sistema, determinados flags.
  - Otra solución es hacer llamadas a fsync allí donde queremos asegurar que se produzca una escritura en disco.
- **fsync**(descriptor)
- "descriptor" es el descriptor del archivo que queremos sincronizar con el disco. La sincronización consiste en escribir en el disco todos los buffers asociados al archivo.

4.4.9. Montaje (mount) y Desmontaje (umount) de Sistemas de Archivos.

- Unidad de disco física:
  - Formada por secciones lógicas, particionadas por el controlador del disco.
  - Cada sección tiene un nombre de archivo de dispositivo.

- Los procesos pueden acceder a los datos de cada sección abriendo el archivo de dispositivo apropiado, y leyendo y escribiendo en el archivo, tratado como una secuencia de bloques de disco.

- Cada sección puede contener un sistema de archivos lógico.

- mount (montaje de sistemas de archivos).
  - **mount**(nombre_especial, punto de montaje, opciones)
  - Conecta un sistema de archivos a la jerarquía del sistema de archivos ya existente.
  - Permite a los usuarios acceder a los datos de una sección de disco como un sistema de archivos y no como un conjunto de bloques de datos.

- umount (desmontaje de un sistema de archivos). Hace todo lo contrario a mount.
  - **umount**(nombre_especial)

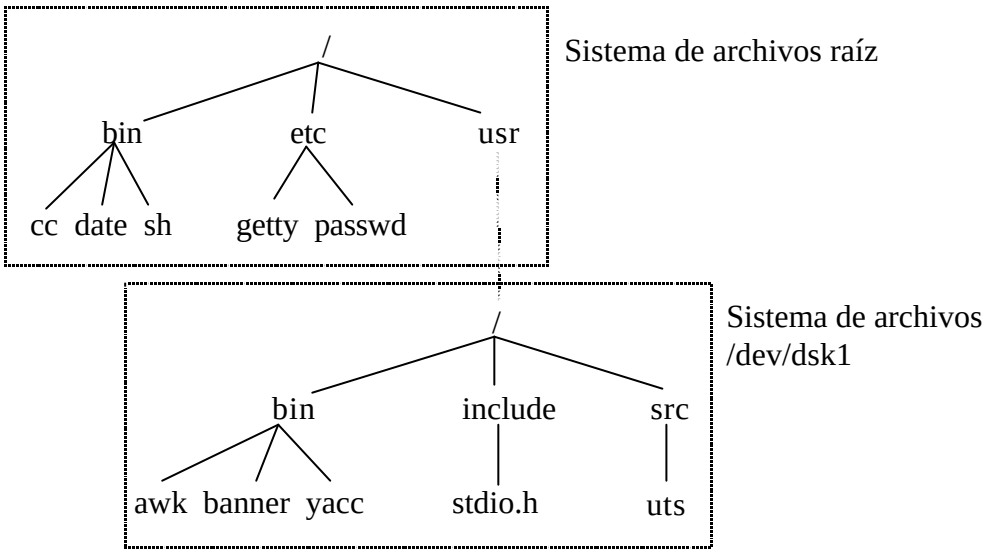


Figura 4.11. Arbol del sistemas de archivos antes y después de montar.

- El *kernel* mantiene la *tabla de montaje* con entradas para los sistemas de archivos montados.
  - Nº de dispositivo que identifica el sistema de archivos montado.
  - Puntero a un buffer que contiene el superbloque del sistema de archivos.
  - Puntero al inodo (*inode*) raíz del sistema de archivos montado.
  - Puntero al inodo (*inode*) del directorio punto de montaje ("/usr" en el ejemplo de la Figura 4.11).

4.4.9.1. Montar y Desmontar de un Sistema de Archivos. Cruce de Sistema de Archivos.

- Inserción de una entrada en la tabla de montaje.

- Borrado de la misma.

- Uso de los datos contenidos en ella.

4.4.9.2. Montaje de un Sistema de Archivos.

- Sólo permitido a los procesos propiedad del superusuario.
- *PASO 1*: Identificación de origen y destino. Comprobaciones:
  - El *kernel* busca el nodo–i (*inode*) del archivo especial (= sistema de archivos a montar).
  - Extrae el nº mayor y menor (que identifica la sección exacta del disco), major y minor number.
  - Busca el inodo (*inode*) del directorio en el que se va a realizar el montaje ⇒ Cuenta de referencia de directorio no puede ser > 1 (Efectos laterales).
- *PASO 2*: Coger entradas en estructuras de datos
  - El *kernel* asigna una entrada libre en la tabla de montaje.
  - Marca la entrada como ocupada.
  - Coloca el nº de dispositivo en el campo correspondiente de la entrada.
- *PASO 3*: Rellenar las estructuras anteriores. Acceso al sistema de archivos a montar.
  - El *kernel* invoca a la llamada *open* sobre el dispositivo de bloques que contiene al sistema de archivos a montar.
  - El *kernel* reserva un buffer libre del *buffer caché* y lee el superbloque.
  - El *kernel* guarda el puntero al inodo (*inode*) del directorio punto de montaje (paths con ".." puedan atravesar el punto de montaje).
  - Puntero al inodo (*inode*) raíz del sistema de archivos montado en la tabla de montaje.
- *PASO 4*: Inicialización del superbloque.
  - El *kernel* inicializa los campos del superbloque del sistema de archivos montado.
  - Bloqueo de listas de inodos (*inodes*) y bloques libres.
  - Número de inodos (*inodes*) libres en el superbloque = 0.
    + Evitar la corrupción de datos.
    + Evitar la sobrecarga (overhead).
  - El *kernel* marca el inodo (*inode*) = punto de montaje.
- *TRANSPARENCIA* ⇒ La tabla de montaje establece la equivalencia entre el directorio de montaje y el directorio raíz del sistema de archivos.

4.4.9.3. Cruce de Puntos de Montaje en los Nombres (*pathname*).

- Cruce desde el sistema en el que se monta el sistema de archivos al sistema de archivos montado y viceversa.
- Ejemplo:

```
mount /dev/dsk1 /usr
cd /usr/src/uts
cd ../../..
```

- Afecta a *namei* y a *iget.*

4.4.9.4. Cruce del Punto de Montaje desde el Sistema de Archivos Global al Sistema de Archivos Montado.

- *iget*: exactamente igual.
- Excepto:
  - Se comprueba si el inodo (*inode*) es un punto de montaje.
  - Si el inodo (*inode*) está marcado como "punto de montaje".
    + El *kernel* busca la entrada en la tabla de montaje correspondiente al inodo (*inode*).
    + Anota el nº dispositivo del sistema de archivos montado.
    + Utilizando el nº dispositivo y el nº inodo (*inode*) raíz, accede al inodo (*inode*) raíz y devuelve dicho inodo (*inode*).

4.4.9.5. Cruce desde el Sistema de Archivos Montado al Sistema de Archivos Global.

- *Namei*: igual al ya visto.
- Excepto:
  - Buscar el nº de inodo (*inode*) de un componente del nombre en un directorio.
  - El *kernel* comprueba si el nº de inodo (*inode*) es el del inodo (*inode*) raíz de un sistema de archivos.
  - Si así es, y el inodo (*inode*) de trabajo actual también lo es y el componente del nombre es "..".
    + El *kernel* identifica el inodo (*inode*) como un punto de montaje.
    + Busca en la tabla de montaje la entrada con el nº de dispositivo = nº de inodo (*inode*).
    + Obtiene el inodo (*inode*) del directorio punto de montaje y continúa su búsqueda para "..", utilizando el inodo (*inode*) de montaje como inodo (*inode*) de trabajo.

4.4.9.6 Desmontaje de un Sistema de Archivos.

- **umount**(nombre_archivo_especial), donde "nombre_archivo_especial" = sistema de archivos a desmontar.
- Acciones del *kernel*:
  - Accede al inodo (*inode*) del dispositivo a desmontar.
  - Obtiene el número de dispositivo del archivo especial.
  - Libera el inodo (*inode*).
  - Busca en tabla de montaje la entrada cuyo nº de dispositivo = nº de dispositivo obtenido.
  - Antes de desmontar el sistema de archivos el *kernel* se asegura de que ningún archivo del sistema de archivos se está utilizando.
    + Realiza la búsqueda en tabla de inodos (*inodes*) de todos los archivos cuyo nº de dispositivo coincida con el del sistema de archivos a desmontar.
    + Si encuentra algún inodo (*inode*), la llamada *umount* falla.
  - Escribe los bloques de "escritura retardada" que hay en el *buffer caché*.
  - Escribe superbloque al disco y actualiza la copia en disco de todos los inodos (*inodes*) que lo necesiten.
  - Libera el inodo (*inode*) raíz del sistema de archivos montado.
  - Invalida los buffers del *buffer caché* que tienen datos del sistema de archivos desmontado.
    + Los mueve a la cabeza de la lista de buffers libres.
    + Los bloques válidos permanezcan en el *buffer caché* más tiempo.
  - Quita el *flag* "punto de montaje" del inodo (*inode*) de montaje y lo libera.
  - Libera la entrada utilizada de la tabla de montaje.

4.4.10. Crear (link) y Eliminar (unlink) Enlaces.

4.4.10.1. Creación de Enlaces *(link)*.

- Esta llamada al sistema crea un nuevo nombre para un archivo ya existente. El sistema de archivos contiene un *pathname* por cada link que tiene el archivo y cualquier proceso puede acceder al archivo usando cualquiera de los *pathnames* que tiene definidos (el *kernel* no tiene un tratamiento especial para ninguno de los *pathnames* válidos, incluido el original).
- El *kernel* permite únicamente al *superusuario* establecer links entre directorios, de tal forma que puede simplificar el código que recorre el árbol de directorios (no hace comprobaciones de lazos sin salida). Se debe recordar también que el contador de enlaces del archivo origen debe ser incrementado en la llamada. Por ejemplo, si ejecutamos *link("fuente", "dir/nuevo_archivo")* y asumimos que el archivo "fuente" tiene asignado el inodo 74, al finalizar la llamada en el directorio "dir" existirá una nueva entrada (nuevo_archivo) con el mismo número de inodo (74), mientras que el contador de enlaces en el inodo 74 quedará incrementado en 1 (este contador lleva cuenta del número de entradas de directorio que hacen referencia a un archivo).

- Acciones (algoritmo): link(pathname_origen, pathname_nuevo)
  - Obtener el inodo del pathname_origen (namei).
  - Comprobaciones.
    - + Máximo número de enlaces.
    - + Permisos de *superusuario* (links de directorios).
  - En caso de error: Liberar el nodo (iput) y devolver Error.
  - Incrementar el contador de enlaces del inodo.
  - Actualizar la copia en disco del inodo.
  - Desbloquea el inodo (evitar el interbloqueo).
  - Obtener inodo del directorio padre (namei).
  - Si ya existe la entrada en ese directorio (anotar entrada libre)
    - + Rehacer los cambios y devolver error.
  - El la primera entrada libre anotar pathname_nuevo y inodo.
  - Liberar inodo de directorio padre y del archivo_origen (iput).

- Observación. El desbloqueo del inodo del archivo origen se produce en mitad de la llamada al sistema. Esto es así para evitar situaciones de interbloqueo. Por ejemplo, suponga la ejecución simultánea de las siguientes llamadas: proceso A: *link("a/b/c/d", "e/f/g")*; proceso B: *link("e/f", "a/b/c/d/ee")*;. Supongamos ahora que el proceso A busca el inodo de "a/b/c/d" y lo consigue, y que en ese momento el proceso B obtiene el inodo que corresponde a "e/f". Si la llamada link mantuviera el inodo bloqueado puede darse la situación que el proceso A se bloquee (sleep) a la espera del inodo "e/f" (bloqueado por el proceso B), y que el proceso B se bloquee (sleep) a la espera del inodo "a/b/c/d" bloqueado por el proceso A. Por ello el *kernel* libera el inodo después de incrementar el contador de referencias. Incluso puede darse que el proceso se bloquee a sí mismo *(link("a/b/c", "a/b/c/d"))* cuando intentara acceder al inodo "/a/b/c" en el *parser* del archivo destino (que el mismo habría bloqueado al obtener el inodo del archivo origen). Recuerde que todos los procesos que intentaran pasar a través de estos inodos bloqueados quedarían a la espera de que el inodo quedara libre.

4.4.10.2. Eliminación de Enlaces (unlink).

unlink(pathname). Elimina la entrada de directorio que corresponde al pathname indicado.

- Acciones (algoritmo):
  - Obtener el inodo del directorio padre (variación de namei).
  - Si último componente es ".".
    - + incrementar contador de referencias (resolver como ejercicio).

    Sino
    - + Obtener el inodo del pathname.
  - Si pathname es directorio y user no es superusuario
    - + Liberar inodos.
    - + Devolver Error.
  - Si se trata de RegionTextoCompartida y contador de referencias e igual a 1
    - + Eliminar la entrada de la tabla de regiones.
  - Modificar directorio padre (inodo cero en la entrada).
  - Liberar inodo del directorio padre (iput).
  - Decrementar el contador de referencias.
  - Liberar inodo (iput). *iput* controla si el contador de enlaces es cero y libera los bloques del archivo (free) y libera el inodo (ifree).

4.4.11. Otras Llamadas Básicas al Sistema para el Sistema de Archivos.

Cambio de nombre de un archivo. La llamada al sistema *rename* permite modificar el nombre de un archivo, y por tanto renombrarlo o desplazarlo entre dos subdirectorios.
int rename(const char *oldpath, const char *newpath);

Cambio de tamaño de un archivo. Un proceso puede modificar el tamaño de un archivo, buen truncando su contenido o bien aumentando su tamaño.
int truncate(const char *pathname, size_t length);
int ftruncate(int fd, size_t length);

Derechos de acceso sobre un archivo. Los derechos de acceso sobre un archivo pueden establecerse en la ceración del archivo a través de la llamada al sistema *open*. Pero también un proceso puede modificarlos y comprobarlos en tiempo de ejecución.
int chmod(const char *pathname, mode_t mode);
int fchmod(int fd, mode_t mode);
int access(const char *pathname, int mode);

Modificación del usuario propietario del archivo. En la creación de un archivo, el usuario y el grupo propietario se inicializan según la identidad del proceso que llama.
int chown (const char *pathname, uid_t owner, gif_t group);
int fchown(int fd, uid_t owner, gif_t group);

Llamadas al sistema para la gestión de directorios. Operaciones como crear, eliminar directorios, obtener el directorio actual y el directorio raíz local, así como la exploración de directorios.
int mkdir(const char *pathname, mode_t mode);
int rmdir(const char *pathname);
int chdir(const char *pathname);
int fchdir(int fd);
char *getcwd(char *buf; size_t size);
int chroot(const char *pathname);
DIR *opendir(const char *pathname);
struct dirent *readdir (DIR *dir);
int closedir (DIR *dir);
El tipo DIR está definido en el archivo *<dirent.h>* y representa un descriptor de directorio abierto. Su contenido no es accesible, de igual modo que el tipo FILE utilizado por la librería estándar de entrada/salida.

Enlaces simbólicos. Los enlaces simbólicos constituyen un tipo especial de archivo y no pueden manipularse con las mismas llamadas al sistema que los archivos regulares. Linux proporciona dos llamadas al sistema que permiten crear un enlace simbólico y leer el nombre del archivo al cual apunta.
int symlink(const char *oldpath, const char *newpath);
int readlink(const char *pathname, char *buf, size_t bufsiz);

Lectura y escritura de varias memorias intermedias (buffers). Las llamadas al sistema read y write permiten a un proceso leer o escribir en un archivo. Un proceso que quiera leer o escribir datos en/desde varios buffers debe utilizar varias llamadas a estas primitivas, lo que provoca los siguientes inconvenientes: (1) el proceso debe utilizar la misma llamada al sistema varias veces lo que provoca varios cambios de modo de ejecución (de modo usuario al modo *kernel*, y viceversa) $\Rightarrow$ deterioro en el rendimiento del sistema; (2) mientras que la escritura con write es atómica, una serie de estas llamadas puede interrumpirse y los datos pueden ser escritos por otro proceso mientras el proceso actual está bloqueado en medio de una secuencia de llamadas write. Por estas razones, Linux ofrece las dos siguientes llamadas al sistema que permiten efectuar lecturas y escrituras a partir de varios buffers no contiguos en memoria.
int readv(int fd, const struct iovec *vector, size_t count);
int writev(int fd, const struct iovec *vector, size_t count);

Atributos de archivos. Llamadas al sistema que permiten obtener los atributos de un archivo.
int stat(const char *pathname, struct stat *buf);
int fstat(int fd, struct stat *buf);
int lstat(const char *pathname, struct stat *buf);

Fechas asociadas a los archivos. Las fechas asociadas a cada archivo pueden modificarse: toda operación sobre un archivo puede actualizar una ovarias fechas ((atime, ctime, mtime). También es posible que un proceso modifique explícitamente las fechas atime y mtime utilizando las siguientes primitivas:
int utime(const char *pathname, structu utimbuf *buf);
int utimes(char *pathname, struct timeval *tvp);

Propiedades de los archivos abiertos. Linux proporciona la primitiva fcntl que permite efectuar operaciones diversas y variadas sobre un archivo abierto.
int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, long arg);

Control de procesos *bdflush*. El contenido de las memorias intermedias (buffers) modificadas se rescribe periódicamente en disco. Dos procesos se encargan de esta reescritura: (1) update, que ejecuta la primitiva sync cada 30 segundos con el objetivo de rescribir el contenido de todos los buffers modificados, Y (2) y bdflush. Este último es un proceso interno del *kernel* creado automáticamente en el arranque del sistema y efectúa las reescrituras de buffers modificados teniendo en cuenta las prioridades asociadas a dicho buffers. Por ejemplo, un buffer que contenga una estructura de control de un sistema de archivos es más prioritaria que un buffer que contenga datos. El proceso bdflush es ejecutado automáticamente por el *kernel* cuando surge la necesidad, aunque se puede llamar a través de la siguiente llamada al sistema.
int bdflush(int func, long data);

Bloqueo de un archivo. La llamada al sistema flock permite bloquear un archivo completo.
int flock(int fd, int operation);

Bloqueo de una sección de un archivo. La llamada al sistema fcntl permite bloquear una parte de un archivo, pero la función lockf ofrece una interfaz simplificada a las posibilidades de bloque de fcntl.
int lockf(int fd, int cmd, off_t len);

Información sobre un sistema de archivos. Las llamadas al sistema statfs y fstatfs permiten obtener las estadísticas de uso de un sistema de archivos.
int statfs(const char *pathname, struct statfs *buf);
int fstatfs(int fd, struct statfs *buf);

Información sobre los tipos de sistemas de archivos soportados. La llamada al sistema sysfs permite conocer los tipos de sistemas de archivos soportados por el *kernel.*
int sysfs(int option, unsigned int fs_index, char *buf);
int sysfs(int option, const char *fsname);
int sysfs(int option);

Manipulación de cuotas de disco. Las coutas de disco asociadas a los usuarios pueden manipularse con la llamada al sistema quotactl.
int quotactl(int cmd, const char *spacial, int id, caddr_t addr);

4.5. CONSISTENCIA Y MANTENIMIENTO DEL SISTEMA DE ARCHIVOS DE UNIX.

- El *kernel* ordena operaciones de escritura en disco para minimizar en lo posible la corrupción del sistema de archivos en caso de caída del sistema.
- Nada más crear el sistema de archivos, debemos revisarlo para verificar la consistencia y asegurar que todos sus bloques son accesibles. Esto se consigue mediante el programa *fsck*. *fsck* revisa y repara de forma interactiva las posibles inconsistencias que encuentra en los sistemas de archivos de UNIX. Si el sistema no presenta inconsistencias, el programa *fsck* informa sobre el número de archivos, número de bloques utilizados y número de bloques libres de que dispone el sistema. Si el sistema es inconsistente, *fsck* proporciona mecanismos para corregirlo.
- Las inconsistencias que revisa *fsck* son las siguientes:
  - Bloques reclamados por más de un inodo (*inode*) o la lista de bloques libres.
  - Bloques reclamados por un inodo (*inode*) o la lista de bloques libres, pero que están fuera del rango del sistema.
  - Contadores de enlaces incorrectos.
  - Revisión de tamaños:
    + Número de bloques demasiado grande.
    + Tamaños de directorios inadecuado.
  - Formato inadecuado para los inodos (*inodes*).
  - Bloques no registrados por nadie (inodos (*inodes*), lista de bloques libres, etc.).
  - Revisión en directorios:
    + Archivos que apuntan a inodos (*inodes*) no asignados.
    + Números de inodos (*inode*s) fuera de rango.
  - Revisión en el superbloque $\Rightarrow$ más bloques para inodos (*inode*s) de los que hay en el sistema de archivos.
  - Formato incorrecto de la lista de bloques libres.
  - Total de bloques libres o contados de inodos (*inodes*) libres incorrecto.
- Si *fsck* encuentra un archivo o directorio cuyo directorio padre no puede determinarse, colocará al archivo huérfano en el directorio *lost + found* perteneciente al sistema que se está revisando. Puesto que el nombre del archivo registra en su directorio *home* y éste es desconocido, a la hora de guardarlo en *lost+found* no podemos hacerlo con su verdadero nombre, por lo que se nombrará con su número de inodo (*inode*).
- La utilidad *fsck* no sólo se utiliza para revisar el sistema de archivos recién creado. Su misión principal es revisar sistemas estropeados por alguna causa accidental como una parada incontrolada del sistema. Como sabemos, el sistema mantiene copias en memoria tanto del superbloque como de la lista de inodos. Además, el acceso a disco se realiza a través del *buffer caché*. Esto crea inconsistencias entre el disco y la memoria, inconsistencias que se corrigen periódicamente con al intervención de los demonios *syncer* y *update*, que se encargan de realizar las llamadas a *sync* para sincronizar el disco con la memoria. Si por cualquier circunstancia el sistema deja de funcionar antes de que se produzca una sincronización, la próxima vez que se intente ese sistema de archivos, será necesario repararlo en la medida de lo posible con la ayuda de *fsck*.

- Ejemplo: Eliminar un archivo.
  - Borra el nombre de archivo del directorio padre y escribe de forma síncrona el directorio al disco, antes de liberar los bloques de datos del archivo y su inodo (*inode*). En el caso de que el sistema cayera antes de destruir el contenido del archivo, los daños serían mínimos.
  - Si la escritura del directorio no fuera síncrona, habría la posibilidad de que existiera una entrada en un directorio que apuntaría a un inodo (*inode*) libre (o reasignado) después de la caída del sistema de archivos.

- Ejemplo: Eliminar un archivo.
  - Al eliminar el contenido de un archivo y su inodo (*inode*), es posible liberar primero los bloques de datos o liberar y escribir primero el inodo (*inode*).
    - *Caso1*: Liberar primero los bloques de datos y caída del sistema.
      - Reinicio ⇒ inodo (*inode*) referencia a bloques que no tienen datos del archivo.
      - El *kernel* ve al archivo aparentemente correcto.
      - El usuario notaría el problema al acceder al archivo.
      - Puede que los bloques de datos se asignasen a otro(s) archivo(s) ⇒ *fsck* ⇒ gran esfuerzo para limpiar el sistema de archivos.
    - *Caso 2*: Escribir primero el inodo (*inode*) y después cae el sistema.
      - El usuario no notará nada anormal en el sistema de archivos.
      - Los bloques de datos asignados al archivo no son accesibles, pero los usuarios no tendrán conciencia del problema.
      - *fsck* busca los bloques de datos que no pertenecen a ningún archivo.

- En lo que respecta al mantenimiento del sistema de archivos, debemos recordar que en el funcionamiento normal del sistema ⇒ el *kernel* mantiene la consistencia del sistema de archivos.
- Caída del sistema ⇒ puede dejar al sistema de archivos en estado inconsistente ⇒ la mayor parte de los datos del sistema de archivos son válidos.
- *fsck* chequea el sistema de archivos y lo repara, en caso de encontrar inconsistencias.

- Ejemplo: Un bloque de disco pertenece a más de un inodo (*inode*) o a lista de libres y a un inodo (*inode*).
  - Al crear un sistema de archivos ⇒ todos los bloques de disco están en la lista de libres.
  - Asignar un bloque de disco ⇒ el *kernel* lo elimina de lista de libres y lo asigna a un inodo (*inode*).
  - No se le puede reasignar a otro inodo (*inode*) hasta que quede libre otra vez.
  - Puede suceder que el *kernel*:
    - Libera un bloque de disco de un archivo.
    - Coloca el nº de bloque en lista de libres del superbloque en memoria.
    - Asigna el bloque de disco a un archivo nuevo.
    - Si *kernel* escribe el inodo (*inode*) y bloque del archivo nuevo al disco y el sistema cae antes de actualizar el inodo (*inode*) del archivo antiguo, ambos inodos (*inode*s) apuntarán al mismo número de bloque de disco.
  - O puede suceder que:
    - El *kernel* escribe el superbloque y su lista de libres y cae antes de escribir el inodo (*inode*) antiguo ⇒ el bloque de disco aparecería en la lista de libres y en el antiguo inodo (*inode*).

- Ejemplo: Un nº de bloque no aparece en la lista de libres ni en algún inodo (*inode*).
  - El sistema de archivos está en un estado inconsistente porque todo nº de bloque debe aparecer en algún sitio, y puede suceder que:
    - Se libera un bloque de un archivo
    - Se coloca su número de bloque en el superbloque.
    - Se escribe el inodo (*inode*) del archivo en el disco.
    - Cae el sistema antes de actualizar la lista de libres del superbloque.

- Ejemplo: Un inodo (*inode*) con nº de enlaces diferente de 0 y no aparece en la estructura de directorios. Bajo esta situación, puede suceder que:
  - El sistema cae después de crear un archivo pero antes de crear la entrada en su directorio padre.
  - El Nº de enlaces al inodo (*inode*) ≠ 0, pero parecerá que no existe en el sistema de archivos.

- Ejemplo: Formato incorrecto del inodo (*inode*).
  - Por ejemplo, el campo del tipo de archivo tiene un valor indefinido.
  - Podría ocurrir que el administrador del sistema hubiera montado un sistema de archivos que no se había formateado apropiadamente.
- Ejemplo: El nº de inodo (*inode*) aparece en la entrada de directorio, pero el inodo (*inode*) está libre.
  - El sistema de archivos es inconsistente y puede suceder que:
    + El *kernel* crea un archivo y escribe la entrada en el directorio al disco pero no escribe el inodo (*inode*) en el disco antes de la caída del sistema.
    + Esta situación se evita ordenando las escrituras apropiadamente.
- Ejemplo: El nº de bloques o de inodos (*inodes*) libres del superbloque es diferente a los que existen en disco ⇒ sistema de archivos inconsistente ⇒ información del superbloque siempre debe coincidir con el estado del sistema de archivos.

4.6. EL SISTEMA DE ARCHIVOS VIRTUAL (VIRTUAL FILE SYSTEM, VFS) DE LINUX

- Un importante avance tuvo lugar cuando se añadió en sistema de archivos EXT en Linux. El sistema de archivos real se separó del sistema operativo y servicios del sistema a favor de un interfaz conocido como el Sistema de Archivos Virtual, o VFS. VFS permite a Linux soportar muchos, incluso muy diferentes, sistemas de archivos, cada uno presentando un interfaz software común al VFS. Todos los detalles del sistema de archivos de Linux son traducidos mediante software de forma que todo el sistema de archivos parece idéntico al resto del *kernel* de Linux y a los programas que se ejecutan en el sistema. La capa del sistema de archivos virtual de Linux permite al usuario montar de forma transparente diferentes sistemas de archivos al mismo tiempo.
- El sistema de archivos virtual está implementado de forma que el acceso a los archivos es rápida y tan eficiente como es posible. También debe asegurar que los archivos y los datos que contiene son correctos. Estos dos requisitos pueden ser incompatibles uno con el otro. El VFS de Linux mantiene un buffer con información de cada sistema de archivos montado y en uso. Se debe tener mucho cuidado al actualizar correctamente el sistema de archivos ya que los datos contenidos en los buffers se modifican cuando se crean, escriben y borran archivos y directorios. Si se pudieran ver las estructuras de datos del sistema de archivos dentro del *kernel* en ejecución, se podría ver los bloques de datos que se leen y escriben por el sistema de archivos. Las estructuras de datos, que describen los archivos y directorios que son accedidos serian creadas y destruidas y todo el tiempo los controladores de los dispositivo estarían trabajando, buscando y guardando datos. El caché más importante es el Buffer Caché, que esté a integrado entre cada sistema de archivos y su dispositivo de bloque. Tal y como se accede a los bloques se ponen en el Buffer Caché y se almacenan en varias colas dependiendo de sus estados. El Buffer Caché no sólo mantiene buffers de datos, también ayuda a administrar el interfaz asíncrono con los controladores de dispositivos modo bloque.
- El VFS proporciona funcionalidades independientes del sistema de archivos físico e implementa por ejemplo, un caché de inodos, un caché nombres (en alguna bibliografía se le conoce como caché de directorios o dentry caché) y un buffer caché. Las funciones internas de VFS aseguran también las partes comunes de las llamadas al sistema que operan sobre archivos. En su mayor parte, estas llamadas comparten los mismos pasos dentro de su implementación: (1) verificación de argumentos; (2) conversión de nombres de archivos en número de dispositivos e inodos; (3) verificación de los permisos; y (4) llamada a la función del sistema de archivos correspondiente. Con el objetivo de llamar a las funciones correspondientes, el VFS utiliza una "aproximación orientada a objetos": a cada sistemas de archivos montado, archivo abierto, e inodo en curso de utilización se les asigna operaciones implementadas en el código del sistema de archivos correspondiente. Se puede ver cada entidad como un objeto de una cierta clase, a la que se asocian métodos, redefinidos en cada módulo que contiene la implementación de un tipo de sistema de archivos. Como el *kernel* de Linux está programado en lenguaje C, esta orientación a objetos se implementa asociando un descriptor a cada objeto, y este descriptor contiene una serie de punteros a funciones. En general, se definen cuatro conjuntos de operaciones:

- Operaciones sobre sistemas de archivos. Son operaciones dependientes del formato físico del sistema de archivos, como la lectura o la escritura de un inodo (operaciones sobre superbloques).
- Operaciones sobre inodos en curso de utilización. Son las operaciones directamente vinculadas a los inodos, como la supresión de un archivo (operaciones sobre inodos).
- Operaciones sobre archivos abiertos. Estas son las operaciones correspondientes a primitivas de E/S sobre archivos, como la escritura o lectura de datos (operaciones sobre archivos abiertos).
- Operaciones sobre cuotas. Estas operaciones son las que se llaman para validar la asignación de bloques e inodos (operaciones sobre cuotas de disco).

4.6.1. Principio y Estructura del VFS

- La figura 4.12 muestra la relación entre el Sistema de Archivos Virtual (VFS) del *kernel* de Linux y su sistema de archivos real. El sistema de archivos virtual debe mantener todos los diferentes sistemas de archivos que hay montados en cualquier momento. Para hacer esto mantiene unas estructuras de datos que describen el sistema de archivos (virtual) por entero y el sistema de archivos, montado, real. De forma más general, el VFS describe los archivos del sistema en términos de superbloque e inodos. Los inodos VFS describen archivos y directorios dentro del sistema; los contenidos y topología del sistema de archivos virtual. De ahora en adelante, para evitar confusiones, se escribirá inodos VFS y superbloques VFS.

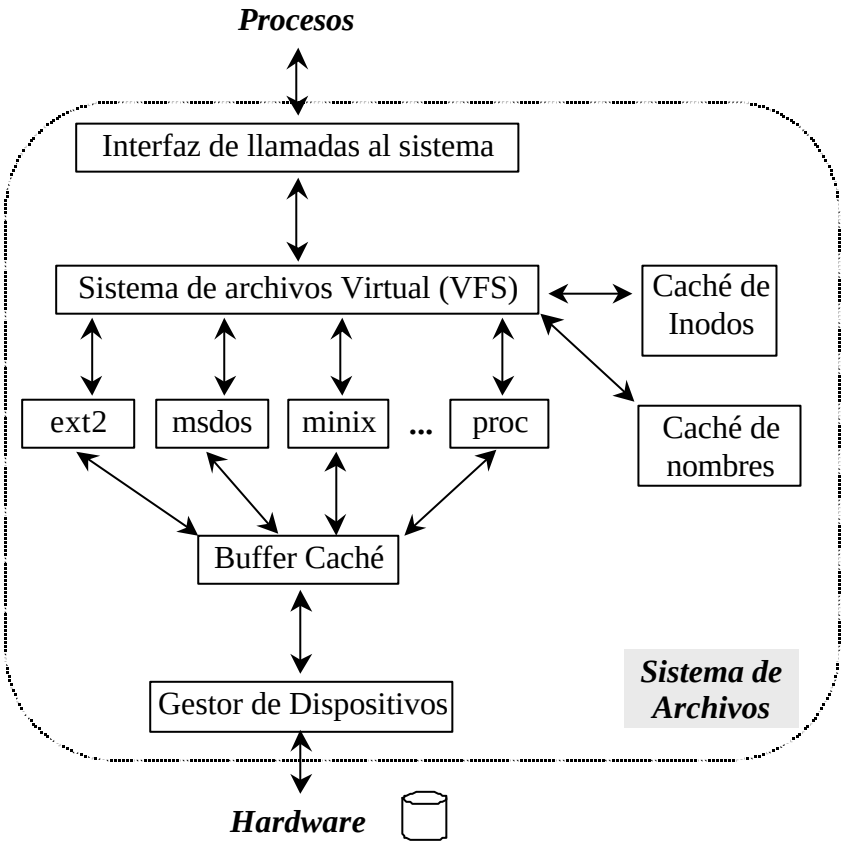


Figura 4.12. Diagrama lógico del Sistema de Archivos Virtual (VFS)

- Cuando un sistema de archivos se inicializa, se registra él mismo con el VFS. Esto ocurre cuando el sistema operativo se inicializa en el momento de arranque del sistema. Los sistemas de archivos reales están compilados con el *kernel* o como módulos cargables. Los módulos de los sistemas de archivos se cargan cuando el sistema los necesita, así, por ejemplo, si el sistema de archivos VFAT está implementado como módulo del *kernel*, entonces sólo se carga cuando se monta un sistema de archivos VFAT. Cuando un dispositivo de bloque base se monta, y éste incluye el sistema de archivos raíz, el VFS debe leer su superbloque. Cada rutina de lectura de superbloque de cada tipo de sistema de archivos debe resolver la topología del sistema de archivos y mapear esa información dentro de la estructura de datos del superbloque VFS. El VFS mantiene una lista de los sistema de archivos montados del sistema junto con sus superbloques VFS. Cada superbloque VFS contiene información y punteros a rutinas que realizan funciones particulares. De esta forma, por ejemplo, el superbloque que representa un sistema de archivos EXT2 montado contiene un puntero a la rutina de lectura de inodos específica. Esta rutina, como todas las rutinas de lectura de inodos del sistema de archivos específico, rellena los campos de un inodo VFS. Cada superbloque VFS contiene un puntero al primer inodo VFS del sistema de archivos. Para el sistema de archivos raíz, éste es el inodo que representa el directorio "/". Este mapeo de información es muy eficiente para el sistema de archivos EXT2 pero moderadamente menos para otros sistema de archivos.
- Ya que los procesos del sistema acceden a directorios y archivos, las rutinas del sistema se dice que recorren los inodos VFS del sistema. Por ejemplo, escribir ls en un directorio o cat para un archivo hacen que el sistema de archivos virtual busque a través de los inodos VFS que representan el sistema de archivos. Como cada archivo y directorio del sistema se representa por un inodo VFS, un número de inodos serán accedidos repetidamente. Estos inodos se mantienen en la **caché de inodos VFS** (VFS inode cache) que hace el acceso mucho más rápido. Si un inodo no está en la caché de inodos, entonces se llama a una rutina específica del sistema de archivos para leer el inodo apropiado. La acción de leer el inodo hace que se ponga en la caché de inodos y siguientes accesos hacen que se mantenga en la caché de inodos. Los inodos VFS menos usados se quitan de la caché de inodos (según una política LRU). El caché de inodos VFS se implementa como una tabla hash cuyas entradas son punteros a listas de inodos VFS que tienen el mismo valor de hash. El valor hash de un inodo se calcula teniendo en cuenta su número de inodo y el identificador de dispositivo lógico asociado al dispositivo físico subyacente que contiene el sistema de archivos.
- Todos los sistemas de archivos de Linux usan un **buffer caché** común para mantener datos de los dispositivos para ayudar a acelerar el acceso por todos los sistemas de archivos al dispositivo físico que contiene los sistemas de archivos. Este buffer caché es independiente del sistema de archivos y se integra dentro de los mecanismos que el *kernel* de Linux usa para reservar, leer y escribir datos. Esto tiene la ventaja de hacer los sistemas de archivos de Linux independientes del medio y de los controladores de dispositivos que los soportan. Tofos los dispositivos estructurados de bloque se registran ellos mismos con el *kernel* de Linux y presentan una interfaz uniforme, basada en bloque y normalmente asíncrona. Incluso dispositivos de bloque relativamente complejos como SCSI lo hacen.
- Cuando el sistema de archivos real lee datos del disco físico realiza una petición al controlador de dispositivo de bloque para leer los bloques físicos del dispositivo que controla. Integrado en este interfaz de dispositivo de bloque está el buffer caché. Al leer bloques del sistema de archivos se guardan en un el buffer caché global compartido por todos los sistemas de archivos y el *kernel* de Linux. Los buffers que hay dentro se identifican por su número de bloque y un identificador único para el dispositivo que lo ha leído. De este modo, si se necesitan muy a menudo los mismos datos, se obtendrán del buffer caché en lugar de leerlos del disco, que tarda más tiempo. Algunos dispositivos pueden realizar lecturas anticipadas, mediante lo cual se realizan lecturas antes de necesitarlas, especulando con que se utilizarán más adelante.

- El VFS también mantiene un **caché de nombres** donde se pueden encontrar los inodos de los directorios que se usan de forma más frecuente. Como experimento, probar a listar un directorio al que no se haya accedido recientemente. La primera vez que se lista, se puede notar un pequeño retardo pero la segunda vez el resultado es inmediato. El caché de nombres no almacena realmente los inodos de los directorios; éstos estarán en el caché de inodos, el caché de nombres simplemente almacena el mapeo entre el nombre entero del directorio y sus números de inodo. El caché nombres (también denominado caché de nombres) consiste en una tabla hash, donde cada entrada de dicho caché apunta a una lista LRU (las nuevas entradas se añaden al principio de la lista, una entrada no utilizada se desplaza poco a poco hacia el final de la lista y se elimina cuando alcanza el final) de entradas del caché de nombres (las estructuras del caché de nombres están definidas en el archivo fuente fs/dcache.c junto con las funciones que las manipula), teniendo todos ellos el mismo valor hash.

4.6.1.1. El Caché de inodos y la Interacción con el Caché de Nombres.

- Como sabemos, para soportar múltiples sistemas de archivos, Linux contiene un nivel especial de interfaces del *kernel* llamado VFS (Sistemas de Archivos Virtuales). El caché de inodos de Linux es implementado en un archivo, *fs/inode.c*, el cual consiste de 977 lineas de código. La estructura del **caché de inodos** Linux es como sigue:
  - Una tabla global hash, *inode_hashtable*, donde cada inodo es ordenado por el valor del puntero del superbloque y el número de inodo de 32bit. Los inodos sin un superbloque (anónimos) (inode->i_sb == NULL) son añadidos a la lista doblemente enlazada encabezada por anon_hash_chain en su lugar. Ejemplos de inodos anónimos son los sockets creados por net/socket.c:sock_alloc(), llamado por fs/inode.c:get_empty_inode().
  - Una lista global del tipo "en uso" (inode_in_use), la cual contiene los inodos válidos con i_count > 0 y i_nlink > 0. Los inodos nuevamente asignados por get_empty_inode() y get_new_inode() son añadidos a la lista inode_in_use.
  - Una lista global del tipo "sin usar" (inode_unused), la cual contiene los inodos válidos con i_count = 0.
  - Una lista por cada superbloque del tipo "modificada" (sb->s_dirty) que contiene los inodos válidos con i_count>0, i_nlink>0 e i_state & I_DIRTY. Cuando el inodo es marcado como dirty, es añadido a la lista sb->s_dirty si el está también ordenado. Manteniendo una lista "modificada" por superbloque de inodos nos permite rápidamente sincronizar los inodos.
  - Un **caché de inodos** (un caché llamado inode_cachep). Cuando los inodos son asignados como libres, ellos son asignados y devueltos a este caché.
- Los tipos de listas son inicializadas desde inode->i_list, la tabla hash desde inode->i_hash. Cada inodo puede estar en una tabla hash y en uno, y en sólo uno, tipo de lista (en uso, sin usar o modificada). Todas estas listas están protegidas por un spinlock inode_lock.
- El caché de inodos es inicializado cuando la función inode_init() es llamada desde *init/main.c:start_kernel()*. La función es marcada como *init*, lo que significa que el código será lanzado posteriormente. Se le pasa un argumento simple (el número de páginas físicas en el sistema). Esto es por lo que el caché de inodos puede configurarse el misma dependiendo de cuanta memoria está disponible, esto es, crea una tabla hash más grande si hay suficiente memoria y más pequeña si no la hay. Las únicas estadísticas de información sobre el caché de inodos es el número de inodos sin usar, almacenados en inodes_stat.nr_unused y accesibles por los programas de usuario a través de los archivos */proc/sys/fs/inode-nr* y */proc/sys/fs/inode-state*.
- Para entender cómo trabaja el caché de inodos, seguimos el tiempo de vida de un inodo de un archivo regular en el sistema de archivos EXT2, por ejemplo, el cómo es abierto y cómo es cerrado:

  ```
  fd = open("file", O_RDONLY);
  close(fd);
  ```
- La llamada al sistema open es implementada en la función fs/open.c:sys_open y el trabajo real es realizado por la función *fs/open.c:filp_open()*, la cual está dividida en dos partes:
  - open_namei(). Rellena la estructura nameidata conteniendo las estructuras dentry y vfsmount (donde dentry es una entrada de directorio).

  - dentry_open(). Dado dentry y vfsmount, esta función asigna una nueva struct file y las enlaza a todas ellas; también llama al método específico del sistema de archivos *f_op->open()* el cual fue inicializado en inode->i_fop cuando el inodo fue leído en *open_namei()* (el cual suministra el inodo a través de dentry->d_inode).

- La función open_namei() interactúa con el caché de nombres a través de path_walk(), el cual en el retorno llama a *real_lookup()*, el cual llama al método específico del sistema de archivos inode operations->lookup(). La misión de este método es encontrar la entrada en el directorio padre con el nombre correcto y entonces hace iget(sb, ino) para coger el correspondiente inodo (el cual nos trae el caché de inodos). Cuando el inodo es leído, el dentry es instanciado por medio de d_add(dentry, inode). Mientras estamos en él, nótese que en los sistemas de archivos del estilo UNIX que tienen el concepto de número de inodos en disco, el trabajo del método lookup es mapear su bit menos significativo al actual formato de la CPU, por ejemplo, si el número de inodos en la entrada del directorio sin formato (específico del sistema de archivos) está en formato de 32 bits little-endian uno haría:

  ```
  unsigned long ino = le32_to_cpu(de->inode);
  inode = iget(sb, ino);
  d_add(dentry, inode);
  ```

- Por lo tanto, cuando abrimos un archivo nosotros llamamos a iget(sb, ino).
  - Intenta encontrar un inodo con el superbloque emparejado y el número de inodo en la tabla hash bajo la protección de inode_lock. Si el inodo es encontrado, su cuenta de referencia (i_count) es incrementada; si era 0 anteriormente al incremento y el inodo no estaba dirty, es quitado de cualquier tipo de lista (inode->i_list) en la que esté (tiene que estar en la lista inode_unused) e insertado en la lista del tipo inode in use; finalmente inodes_stat.nr_unused es decrementado.
  - Si el inodo está actualmente bloqueado (lock), esperaremos hasta que se desbloquee, por lo tanto está garantizado que iget() devolverá un inodo desbloqueado.
  - Si el inodo no fue encontrado en la tabla hash entonces es la primera vez que se pide este inodo, por lo tanto llamamos a get_new_inode(), pasándole el puntero al sitio de la tabla hash donde debería de ser insertado.
  - get_new_inode() asigna un nuevo inodo desde el caché inode_cachep, pero esta operación puede bloquear (asignación GFP_KERNEL), por lo tanto el spinlock que guarda la tabla hash tiene que ser quitado. Desde que hemos quitado el spinlock, entonces debemos de volver a buscar el inodo en la tabla; si esta vez es encontrado, se devuelve (después de incrementar la referencia por iget) el que se encontró en la tabla hash y se destruye el nuevamente asignado. Si aún no se ha encontrado en la tabla hash, entonces el nuevo inodo que tenemos acaba de será signado y es el que va a ser usado; entonces es inicializado a los valores requeridos y el método específico del sistema de archivos sb->s_op->read_inode() es llamado para propagar el resto del inodo. Esto nos proporciona desde el caché de inodos la vuelta al código del sistema de archivos (venimos del caché de inodos cuando el método específico del sistema de archivos lookup() llama a iget(). Mientras el método s_op->read_inode() está leyendo el inodo del disco, el inodo está bloqueado (i_state = I_LOCK); él es desbloqueado después de que el método read_inode() retorne y todos los que estén esperando por él hayan sido despertados.
- Ahora, veamos que pasa cuando cerramos este descriptor de archivos. La llamada al sistema close está implementada en la función fs/open.c:sys_close(), la cual llama a do close(fd, 1) el cual rompe (reemplaza con NULL) el descriptor del descriptor de archivos de la tabla del proceso y llama a la función filp_close(), la cual realiza la mayor parte del trabajo. La parte interesante sucede en fput(), la cual chequea si era la última referencia al archivo, y si es así llama a fs/file_table.c:_fput() la cual llama a fput() en la cual es donde sucede la interacción con caché de nombres (y entonces con la memoria intermedia de inodos (el caché de nombres es la "memoria intermedia" de inodos maestra). El fs/dcache.c:dput() hace dentry_iput() la cual nos ofrece la posibilidad de volver al caché de inodos a través de iput(inode), por lo tanto es importante comprender fs/inode.c:iput(inode):
  - Si el paráametro pasado a nosotros es NULL, no hacemos nada y regresamos.
  - Si hay un método específico del sistema de archivos sb->s_op->put_inode(), es llamada inmediato-mente sin mantener ningún spinlock (por lo tanto puede bloquear).

  - El spinlock inode_lock es tomado y i_count es decrementado. Si NO era la última referencia a este inodo entonces simplemente chequeamos si hay muchas referencias a el y entonces i_count puede urdir sobre los 32 bits asignados a el si por lo tanto podemos imprimir un mensaje de peligro y regresar. Nótese que llamamos a printk() mientras mantenemos el spinlock inode_lock (esto está bien porque printk() nunca bloquea, entonces puede ser llamado absolutamente en cualquier contexto.
  - Si era la última referencia activa entonces algún trabajo necesita ser realizado.
- El trabajo realizado por iput() en la última referencia del inodo es bastante complejo, por lo tanto lo separaremos en una lista de si misma:
  - Si i_nlink == 0 (por ejemplo, el archivo fue desenlazado mientras lo manteníamos abierto) entonces el inodo es quitado de la tabla hash y de su lista de tipos; si hay alguna página de datos mantenida en el caché de páginas para este inodo, son borradas por medio de truncate_all_inode_pages(&inode->i_data). Entonces el método específico del sistema de archivos s_op->delete_inode() es llamado, el cual normalmente borra la copia en disco del inodo. Si no hay un método s_op->delete_inode() registrado por el sistema de archivos entonces llamamos a clear_inode(inode), el cual llama a s_op->clear_inode() si está registrado y si un inodo corresponde a un dispositivo de bloques, esta cuenta de referencia del dispositivo es borrada por bdput(inode->i_bdev).
  - Si i_nlink != 0 entonces chequeamos si hay otros inodos en el mismo cubo hash y si no hay ninguno, entonces si el inodo no está dirty lo borramos desde su tipo de lista y lo añadimos a la lista inode_unused incrementando inodes_stat.nr_unused. Si hay inodos en el mismo cubo hash entonces los borramos de la lista de tipo y lo añadimos a la lista inode_unused. Si no había ningún inodo, entonces lo borramos de la lista de tipos y lo destruimos completamente.

4.6.2. El Modelo de Archivos Común (Common File Model).

La idea principal detrás del sistema de archivos virtual (VFS) consiste en introducir un modelo de archivos común capaz de representar todos los sistemas de archivos soportados. Este modelo refleja estrictamente el modelo de archivo suministrado por el sistema de archivos UNIX tradicional. Esto no es sorprendente, ya que Linux desea ejecutar su sistema de archivos nativo con el mínimo de sobrecarga. Sin embargo, cada implementación del sistema de archivos específico debe traducir su organización física en el modelo de archivos común de VFS.

Por ejemplo, en el modelo de archivos común, cada directorio se considera como un archivo, el cual consta de una lista de archivos y otros directorios. Sin embargo, varios sistemas de archivos basados en disco no UNIX utilizan una tabla de asignación de archivos (File Allocation Table, FAT), que almacena las posiciones de cada archivo en el árbol de directorios. En estos sistemas de archivos, los directorios no son archivos. Para cumplir el modelo de archivos común de VFS, la implementación de Linux de tales sistemas de archivos basados en FAT debe poder construirlos cuando sea necesario (on the fly), en el momento que se necesite los archivos se corresponden con directorios. Tales archivos existen sólo como objetos en la memoria del kernel.

Mas aún, el kernel de Linux no puede codificar una función para gestionar una operación como read() o ioctl(). En su lugar, éste debe utilizar un puntero para cada operación; el puntero se hace apuntar a la función apropiada para el sistema de archivos particular que está siendo accedido.

Ilustremos este concepto mostrando cómo la función read(), podría ser traducida por el kernel en una llamada específica del sistema de archivos MS-DOS.

```
inf = open("/floppy/TEST", 0_RDONLY, 0);                    // fs: MS-DOS
inf = open("/tmp/test", 0_WRONLY | 0_CREATE | 0_TRUNC, 0600);   // fs: EXT2
do {
        s = read, inf, buff, 4096);
        write(outf, buff, s);
        } while (s);
close(outf);
close(inf);
```

La llamada de la aplicación a read() hace que el kernel invoque la correspondiente rutina de sercivo sys_read(), igual que cualquier otra llamada al sistema. El archivo es representado por una estructura de

datos file en la memoria del kernel (veremos más en profundidad dicha estructura en un apartado a continuación). Esta estructura de datos contiene un campo denominado f_op que contiene punteros a funciones específicas a archivos MS-DOS, incluyendo una función que lee un archivo. La rutina de servicio sys_read() encuentra el puntero a esta función y la invoca. Por tanto, la aplicación de la función read() se transforma en una llamada indirecta:

```
file->f_op->read(...);
```

Así mismo, la operación write() dispara la ejecución de la función write de EXT2 apropiada asociada con el archivo de salida. En pocas palabras, el kernel es responsable de asignar el conjunto correcto de punteros a la variable file asociada con cada archivo abierto, y entonces invocar la llamada específica a cada sistema de archivos al que apunta el campo f_op.

Podemos pensar del modelo de archivos común como orientado a objetos, donde un objeto es una construcción software que define una estructura de datos y los métodos que operan sobre ella. Por razones de eficiencia, Linux no está implementado en un lenguaje orientado a objetos como es C++. Los objetos se implementan, por tanto, como estructuras de datos que algunos campos apuntan a funciones que corresponden a métodos de objetos.

El modelo de archivos común consiste en los siguientes tipos de objetos:

- El *objeto superbloque* (superblock object). Almacena información relacionada con un sistema de archivos montado. Para sistemas de archivos basados en disco, este objeto normalmente corresponde al bloque de control del sistema de archivos (filesystem control block) almacenado en disco.
- El *objeto inodo* (inode object). Almacena información sobre un archivo específico. Para sistemas de archivos basados en disco, este objeto normalmente se corresponde a un bloque de control de archivo (file control block) almacenado en disco. Cada objeto inodo se asocia con un número de inodo (inode number), que identifica de forma unívoca el archivo dentro del sistema de archivos.
- El *objeto archivo* (file object). Almacena información sobre la interacción entre un archivo abierto y un proceso. Esta información existe en la memoria del kernel durante el periodo cuanod cada proceso accede a un archivo.
- El *objeto dentry* (dentry object). Almacena información sobre el enlazado de una entrada de directorio con el correspondiente archivo. Cada sistema de archivos basado en disco almacena esta información de una forma particular y propia en disco.

La siguiente figura (Figura 4.13) ilustra en ejemplo sencillo de cómo procesos interactúan con archivos siguiendo el modelo de archivos común. En ella podemos observar tres procesos diferentes han abierto el mismo archivo, dos de ellos utilizan el mismo enlace fuerte (hard link). En este caso, cada uno de los tres procesos utiliza su propio objeto file (objeto archivo), mientras que sólo dos objetos dentry son necesarios (uno para cada enlace fuerte). Ambos objetos dentry se refieren al mismo objeto inodo, que identifica el objeto superbloque y, junto con el último, el archivo de disco común.

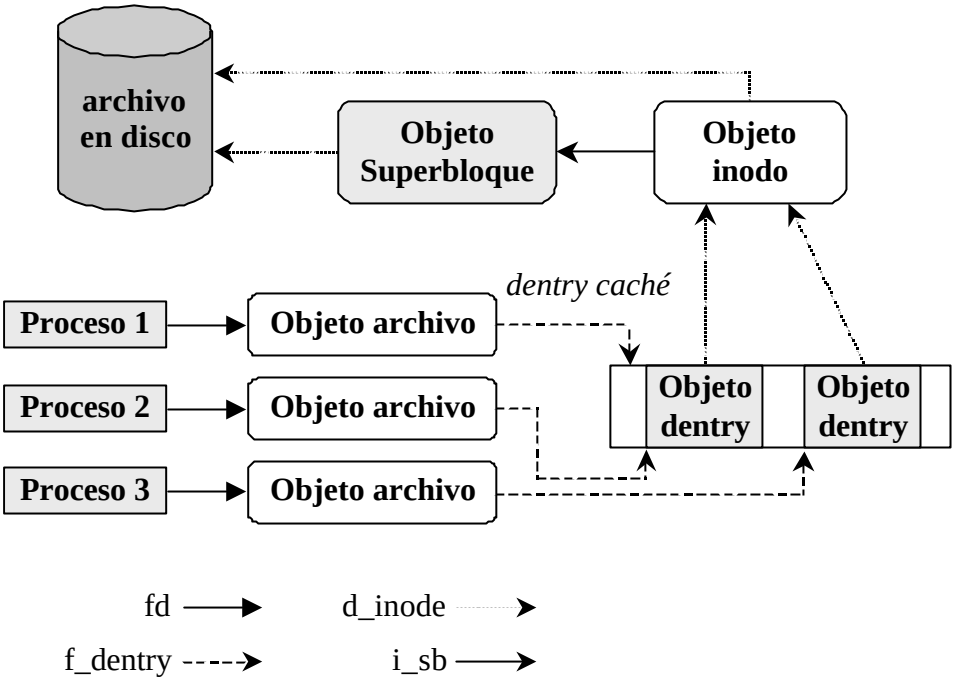


Figura 4.13. Interacción entre procesos y objetos VFS

Además, suministrando una interfaz común a la implementación de todos los sistemas de archivos, el sistema de archivos virtual (VFS) tiene otro papel importante relacionado con el rendimiento del sistema. Los objetos dentry más recientemente utilizados se mantienen en un caché de disco denominado el *dentry caché*, que acelera la traducción de un nombre (pathname) de archivo al inodo del último componente del nombre.

Hablando de forma general, el *caché de disco* es un mecanismo de software que permite al kernel mantener en RAM alguna información que está normalmente almacenada en disco, para que en futuros accesos a datos pueden obtenerse más rápidamente sin necesidad de tener que acceder a disco (acceso más lento).

4.6.3. Registro/Desregistro de Sistemas de Archivos

El *kernel* de Linux suministra un mecanismo para los nuevos sistemas de archivos para ser registrados con el mínimo esfuerzo. Vamos a considerar los pasos requeridos para implementar un sistema de archivos bajo Linux. El código para implementar un sistema de archivos puede ser un módulo dinámicamente cargado o estar estáticamente enlazado en el *kernel*, el camino es realizado por Linux de forma trasparente. Todo lo que se necesita es rellenar una estructura de datos struct *file_system_type* y registrarla con el VFS usando la función register_filesystem() como en el siguiente ejemplo de *fs/bfs/inode.c*:

```
#include <linux/module.h>
#include <linux/init.h>
static struct super_block *bfs_read_super(struct super_block *, void *, int);
static DECLARE_FSTYPE_DEV(bfs_fs_type, “bfs”, bfs_read_super);
static int __init init_bfs_fs(void)
{
        return register_filesystem(&bfs_fs_type);
}
static void __exit exit_bfs_fs(void)
{
        unregister_filesystem(&bfs_fs_type);
}
module_init(init_bfs_fs)
module_exit(exit_bfs_fs)
```

Las macros module_init()/module_exit() aseguran que, cuando BFS (sistema de archivos de disco) es compilado como un módulo, las funciones init_bfs_fs() y exit_bfs_fs() se convierten en init_module() y

cleanup_module() respectivamente; si BFS está estáticamente enlazado en el *kernel* el código exit_bfs_fs() lo hace innecesario.

La estructura struct file_system_type (describiendo cada tipo de sistema de archivos) es declarada de la siguiente forma en <include/linux/fs.h>:

```
struct file_system_type {
        const char *name;
        int fs_flags;
        struct super_block *(*read_super) (struct super_block *, void *, int);
        struct module *owner;
        struct vfsmount *kern_mnt;
        struct file_system_type * next;
        struct list_head fs_super;
};
```

Los campos de la estructura struct file_system_type son los siguientes:

name. Nombre del tipo de sistema de archivos, aparece en el archivo /proc/filesystems y es usado como clave para encontrar un sistema de archivos por su nombre; este mismo nombre es usado por el tipo de sistema de archivos en mount, y debería de ser único; sólo puede haber un sistema de archivos con un nombre dado. Para los módulos, los nombres de los punteros al espacio de direcciones del módulo no son copiados: esto significa que cat /proc/filesystems puede fallar si el módulo fue descargado pero el sistema de archivos aún está registrado.

fs_flags. Una o más (ORed) de los flags: FS_REQUIRES_DEV para sistemas de archivos que sólo pueden ser montados como dispositivos de bloque, FS_SINGLE para sistemas de archivos que pueden tener sólo un superbloque, FS_NOMOUNT para los sistemas de archivos que no pueden ser montados desde el espacio de usuario por medio de la llamada al sistema mount: ellos pueden de todas formas ser montados internamente usando la interfaz kern_mount(), por ejemplo, pipefs.

read_super. Un puntero a la función que lee el superbloque durante la operación de montaje (función de inicialización). Esta función es requerida; si no es suministrada, la operación de montaje (desde el espacio de usuario o desde el *kernel*) fallará siempre excepto en el caso FS_SINGLE donde fallará en get_sb_single(), intentando desreferenciar un puntero a NULL en fs_type->kern_mnt->mnt_sb con (fs_type->kern_mnt = NULL).

owner. Puntero al módulo que implementa este sistema de archivos. Si el sistema de archivos está enlazado estáticamente en el *kernel* entonces este campo es NULL. No se necesita establecer esto manualmente puesto que la macro THIS_MODULE lo hace automáticamente.

kern_mnt. Sólo para sistemas de archivos FS_SINGLE. Esto es establecido por kern_mount().

next. Enlaza a la cabecera de la lista simplemente enlazada (puntero de encadenamientos) file_systems *fs/super.c*. La lista está protegida por el spinlock read-write file_systems_lock y las funciones register/unregister_filesystem() modificadas por el enlace y desenlace de la entrada de la lista.

fs_super. Primer elementos (head) de una lista de objetos superblock.

Al inicializar el sistema, cada tipo de sistema de archivos cuyo soporte ha sido compilado en el *kernel*, se registra en el VFS, llamando a la función *register_file_system*. Esta función registra el descriptor del tipo de sistema de archivos en una lista encadenada referenciada por la variable file_systems. Al montar un sistema de archivos, el VFS explora esta lista encadenada buscando el descriptor del tipo de sistema de archivos solicitado, y luego llamada a la función read_super correspondiente.

El trabajo de la función read_super() es la de rellenar los campos del superbloque, asignando el inodo raíz e inicializando cualquier información privada del sistema de archivos asociadas por esta instancia montada del sistema de archivos. Por lo tanto, normalmente el read_super() hará:

- Lee el superbloque desde el dispositivo especificado a través del argumento sb->s_dev, usando la función del buffer caché bread(). Si se anticipa a leer unos pocos más bloques de metadatos inmediatamente siguientes, entonces tiene sentido usar breada() para planificar el leer bloque extra de forma asíncrona.
- Verifica que el superbloque contiene el número mágico válido y todo “parece” correcto.
- Inicializa sb->s_op para apuntar a la estructura struct super_block_operations. Esta estructura contiene las funciones específicas del sistema de archivos implementando las operaciones como “leer inodo”, “borrar inodo”, etc.

- Asigna el inodo y dentry al raíz usando d_alloc_root().
- Si el sistema de archivos no está montado como sólo lectura, entonces establece sb->s_dirt a 1 y marca el buffer que contiene el superbloque como dirty.

4.6.4. Administración de Descriptores de Archivos por Proceso

Bajo Linux, tiene varias formas de diferenciar el descriptor de archivos del usuario y la estructura de inodos del *kernel*. Cuando un proceso realiza la llamada al sistema open, el *kernel* devuelve un entero (short int) no negativo el cual puede ser usado para operaciones futuras de E/S en este archivo (descriptor del archivo). Cada estructura de archivo apunta a dentry a través de file->f_dentry. Y cada dentry apunta a un inodo a través de dentry->d_inode. Cada proceso contiene un campo tsk->files el cual es un puntero a la estructura *struct files_struct* definida en <include/linux/sched.h>:

```
struct files_struct {
        atomic_t count;
        rwlock_t file_lock;
        int max_fds;
        int max_fdset;
        int next_fd;
        int next_fd;
        struct file ** fd;              /* una matriz de descriptores de archivos */
        fd_set *close_on_exec;
        fd_set *open_fds;
        fd_set close_on_exec_init;
        fd_set open_fds_init;
        struct file ** fd_array;        /* array inicial de punteros a objetos file */
};
```

file->count. Es una cuenta de referencia de los descriptores de archivos asociados al proceso, incrementada por get_file() (normalmente llamada por fget()) y decrementada por fput() y por put_filp(). La diferencia entre fput() y put_filp() es que fput() hace más trabajo para archivos regulares, como la liberación de conjuntos de bloqueos, liberación de dentry, etc, mientras que put_filp() es sólo para manipular las estructuras de tablas de archivos, esto es, decrementa la cuenta, quita el archivo desde anon_list y lo añade a la free_list, bajo la protección del spinlock files_lock.
El puntero ***tsk->files*** puede ser compartido entre padre e hijo si el hilo hijo fue creado usando la llamada al sistema clone() con la bandera CLONE_FILES establecida en los argumentos de las banderas de clone. Esto puede ser visto en kernel/fork.c:copy_files() (llamada por do_fork()) el cual sólo incrementa el file->count si CLONE_FILES está establecido, en vez de la copia usual de la tabla de descriptores de archivos en la tradición respetable en el tiempo de los clásicos fork() de UNIX.
Cuando un archivo es abierto (struct file ** **fd** representa a los descriptores de archivos abiertos), la estructura del archivo asignada para él es instalada en el slot current->files->fd[fd] y un bit fd es establecido en el bitmap current->files->open_fds. Todo esto es realizado bajo la protección de escritura del spinlock read-write current->files->file_lock. Cuando el descriptor es cerrado un bit fd es (inicializado) en current->files->open_fds, y current->files->next_fd es establecido igual a fd como una indicación para encontrar el primer descriptor sin usar la próxima vez que este proceso quiera abrir un archivo.

Otra función que describe la información de gestión de archivos para cada proceso es fs_struct, definida en el archivo <include/linux/sched.h> y que contiene, los siguientes campos:

```
struct fs_struct {
        int count;
        rwlock_t lock;
        unsigned short umask;
        struct inode *root;
        struct inode *pwd;
        struct inode *altroot;
        struct vfsmount *rootmnt;
        struct vfsmount *pwdmnt;
```

```
    struct vfsmount *altrootmnt;

};
```

count. Indica el número de procesos que referencian este descriptor.
lock. Bloqueo de lectura/escritura para los campos de la tabla.
umask. Indica los derechos de acceso predeterminados utilizados en la creación de archivos.
root. Descriptor del inodo correspondiente a la raíz del sistema de archivos para el proceso.
pwd. Descriptor del inodo correspondiente al directorio actual del proceso.
altroot. Descriptor del inodo del directorio raíz alternativo o emulado.
rootmnt. Descriptor del objeto del sistema de archivos montado del directorio raíz.
pwdmnt. Descriptor del objeto del sistema de archivos montado del directorio de trabajo actual.
altrootmnt. Descriptor del del objeto del sistema de archivos montado del directorio raíz emulado.

4.6.5. Inodos en Curso de Utilización

Cada archivo, directorio y demás elementos de un sistema de archivos en VFS se representa por uno y solo un inodo VFS. La información en cada inodo VFS se construye a partir de información del sistema de archivos por las rutinas específicas del sistema de archivos. Los inodos VFS existen sólo en la memoria del *kernel* y se mantienen en el caché de inodos VFS tanto tiempo como sean útiles para el sistema.

La estructura de datos del VFS denominada *inode* mantiene la información sobre un archivo o directorio que reside en el disco y está en curso de utilización. Esta estructura strcut inode definida en <include/linux/fs.h> tiene la siguiente forma:

```
struct inode {
    kdev_t i_dev;
    unsigned long i_ino;
    umode_t i_mode;
    nlink_t i_nlink;
    uid_t i_uid;
    gid_t i_gid;
    kdev_t i_rdev;
    off_t i_size;
    time_t i_atime;
    time_t i_mtime;
    time_t i_ctime;
    unsigned long i_blksize;
    unsigned long i_blocks;
    unsigned long i_version;
    unsigned long i_nrpages;
    struct semaphore i_sem;
    struct inode_operations *i_op;
    struct super_block *i_sb;
    struct wait_queue *i_wait;
    struct file_lock *i_flock;
    struct vm_area_struct *i_mmap;
    struct page *i_pages;
    struct dquot *i_dquot[MAXQUOTAS];
    struct inode *i_next, *i_prev;
    struct inode *i_hash_next, *i_hash_prev;
    struct inode *i_bound_to, *i_bound_by;
    struct inode *i_mount;
    unsigned short i_count;
    unsigned short i_flags;
    unsigned char i_lock;
    unsigned char i_dirt;
    unsigned char i_pipe;
    unsigned char i_sock;
```

```
        unsigned char i_seek;
        unsigned char i_update;
        unsigned short i_writecount;
        union {
                struct pipe_inode_info pipe_i;
                struct minix_inode_info minix_i;
                struct ext_inode_info ext_i;
                struct ext2_inode_info ext2_i;
                struct hpfs_inode_info hpfs_i;
                struct msdos_inode_info msdos_i;
                struct umsdos_inode_info umsdos_i;
                struct iso_inode_info isofs_i;
                struct nfs_inode_info nfs_i;
                struct xiafs_inode_info xiafs_i;
                struct sysv_inode_info sysv_i;
                struct affs_inode_info affs_i;
                struct ufs_inode_info ufs_i;
                struct socket socket_i;
                void *generic_ip;
        } u;
};
```

i_dev. Este es el identificador de dispositivo del dispositivo que contiene el archivo o lo que este inodo VFS represente.
i_ino. Este es el número de inodo y es único en este sistema de archivos. La combinación de i_dev e i_ino es única dentro del sistema de archivos virtual.
i_mode. Este campo describe el modo del archivo (tipo y permisos de acceso).
i_nlink. Número de enlaces.
i_uid e ***i_gid***. Los identificadores de usuario y grupo propietario
i_rdev. Identificador del dispositivo si el inodo representa un archivo especial.
i_size. Tamaño del archivo en bytes.
i_atime, ***i_mtime*** e ***i_ctime***. Los tiempos (fechas) de último acceso, última modificación del contenido y última modificación del inodo, respectivamente.
i_blksize. El tamaño de bloque en bytes para este archivo, por ejemplo 1024 bytes.
i_blocks. Número de bloques de 512 bytes asignados al archivo.
i_version. Número de versión incrementado automáticamente con cada nuevo uso.
i_nrpages. Número de páginas cargadas en memoria de este archivo.
i_sem. Semáforo utilizado para serializar los accesos concurrentes al archivo.
i_op. Un puntero a un bloque de direcciones de operaciones vinculadas al inodo. Estas rutinas son específicas del sistema de archivos y realizan operaciones para este inodo. Por ejemplo, truncar el archivo que representa este inodo.
i_sb. Puntero al descriptor del sistema de archivos correspondiente.
i_wait. Variable utilizada para sincronizar los accesos concurrentes al inodo.
i_flock. Puntero al descriptor de bloqueos asociados al inodo.
i_mmap. Puntero al descriptor de secciones del inodo proyectados en memoria.
i_pages. Punteros a los descriptores de páginas del inodo proyectados en memoria.
i_dquot. Punteros a los descriptores de cuotas de disco asociados al inodo.
i_next. Puntero al inodo siguiente en la lista.
i_prev. Puntero al inodo anterior en la lista.
i_hash_next. Puntero al inodo siguiente en la lista de hash.
i_hash_prev. Puntero al inodo anterior en la lista de hash.
i_mount. Puntero al inodo raíz de un sistema de archivos en el caso de un punto de montaje.
i_count. El número de componentes del sistema que están usando actualmente este inodo VFS (número de usos del inodo). Un contador a cero indica que el inodo está libre para ser descartado o reusado.
i_flags. Opciones de montaje del sistema de archivos que contienen el inodo.
i_lock. Este campo se usa para bloquear el inodo VFS (indica si el inodo está bloqueado), por ejemplo, cuando se lee del sistema de archivos.
i_dirty. Indica si se ha modificado en este inodo.

i_pipe. Parámetro que indica si el inodo corresponde a una tubería.
i_sock. Elemento que indica si el inodo corresponde a un socket.
i_writecount. Número de aperturas en escritura de este archivo.
union u. Información dependiente del tipo de sistema de archivos. Este campo contiene también, entre otros, el campo void *generic_ip, que puede usarse para guardar datos privados.

Los descriptores de inodos (una de las principales estructura del sistema VFS) se encadenar en varias listas: Una lista global, cuya dirección del primer elemento se almacena en la variable first_inode, que contiene todos los descriptores de inodos.

Varias listas de hashing, una función de hashing, que actúa sobre el identificador de dispositivo y el número de inodo, permite colocar los descriptores en listas diferentes de tamaño más reducido que la lista global. De este modo, la búsqueda de un descriptor de inodo en una lista es más rápida que en la lista global.

La estructura inode_operations contiene punteros a funciones que son llamadas por el VFS e implementadas en el código de los sistemas de archivos.

```
struct inode_operations {
        struct file_operations *default_file_ops;
        int (*create) (struct inode *dir, const char *name, int len, int mode, struct inode **result);
        int (*lookup) (struct inode *dir, const char *name, int len, struct inode **restult);
        int (*link) (struct inode *inode, struct inode *dir, const char *name, int len);
        int (*unlink) (struct inode *dir, const char *name, int len);
        int (*symlink) (struct inode *dir, const char *name, int len, const cahr *symname);
        int (*mkdir) (struct inode *dir, const char *name, int len, int mode);
        int (*rmdir) (struct inode *dir, const char *name, int len, int mode);
        int (*mknod) (struct inode *dir, const char *name, int len, int mode, int rdev);
        int (*rename) (struct inode *old_dir, const char *old_name, int old_len, struct inode *new_dir, const
        char *new_name, int new_len);
        int (*readlink) (struct inode *inode, char *buf, int bufsize);
        int (*follow_link) (struct inode *dir, struct inode *inode, int flag, int mode, struct inode **result);
        int (*truncate) (struct inode *inode);
        int (*permission) (struct inode *inode, int perm);
};
```

default_file_ops. Proporciona la dirección de las operaciones sobre archivos asociados del inodo a utilizar de modo predeterminado al abrir un archivo.
create. La operación create es llamada por el VFS para crear una nueva entrada (archivo) en el directorio referenciado por el parámetro dir. El nombre de la entrada a crear es especificado por el parámetro name y len, mientras que mode indica el modo (tipo y derechos de acceso) del archivo a crear. El inodo correspondiente al archivo creado se devuelve en el parámetro result.
lookup. La operación lookup es llamada por el VFS para efectuar la búsqueda de una entrada en el directorio referenciado por el parámetro dir. El nombre de la entrada a buscar es especificado por los parámetros name y len. En inodo correspondiente a la entrada se devuelve por el parámetro result.
link. La operación link es llamada por el VFS para crear un enlace a un inodo referenciado por el parámetro inode. El nombre del enlace a crea se especifica por los parámetros name y len, y el directorio en el que el enlace debe crearse se indica por el parámetro dir.
unlink. La operación unlink se llama para suprimir una entrada en el directorio referenciado por el parámetro dir. El nombre de la entrada a suprimir se especifica por los parámetros name y len.
symlink. La operación symlink se llama para crear un enlace simbólico en el directorio referencia por el parámetro dir. Los parámetros name y len especifican el nombre del enlace a crear. El nombre del destino se indica en symname, en forma de una cadena de caracteres finalizada por el carácter nulo.
mkdir. La operación mkdir es llamada por VFS para crear un subdirectorio en el directorio referenciado por el parámetro dir. El nombre del subdirectorio a crear se especifica por los parámetros name y len. El parámetro mode indica los derechos de acceso del directorio a crear.
rmdir. La operación rmdir es llamada por VFS para suprimir un subdirectorio en el directorio referenciado por el parámetro dir. El nombre del subdirectorio a crear se especifica por bs parámetros name y len. El parámetro mode indica los derechos de acceso del directorio a crear.

mknod. La operación mknod es llamada por VFS para crear un archivo especial en el directorio referenciado por el parámetro dir. El nombre del archivo a crear se especifica por los parámetros name y len; y mode indica el tipo y los derechos de acceso del directorio a crear. Por último, el parámetro rdev contiene el identificador del dispositivo correspondiente al archivo especial.
rename. La operación rename se llama para renombrar una entrada de directorio. Los parámetros old_dir, old_name, y old_len especifican el nombre de entrada a renombrar, mientras que el nuevo nombre se indica en los parámetros new_dir, new_name y new_len.
readlink. La operación readlink se llama para leer el contenido del enlace simbólico referenciado por el parámetro inode. El resultado se devuelve en el buffer apuntado por buf, cuya longitud en bytes se indica en bufsize.
follow_link. La operación follow_link se llama para resolver un enlace simbólico referenciado por el parámetro inode. El parámetro dir especifica el inodo del directorio a partir del cual se debe efectuar la interpretación. El inodo resultante se devuelve en result.
truncate. La operación truncate se llama para modificar el tamaño del archivo referenciado por el parámetro inode. El VFS modifica el campo i_size del inodo antes de llamar esta operación.
permission. La operación permission se llama para verificar que el proceso que llama posee derechos de acceso suficientes sobre el archivo referenciado por el parámetro inode. El parámetro perm indica los derechos de acceso deseados.

4.6.6. Administración de Estructuras de Archivos Abiertos

A cada archivo abierto en el sistema corresponde un descriptor. La estructura struct *file* se define en <include/linux/fs.h> y contiene los siguientes campos:

```
struct fown_struct {
        int pid;                /* pid o pgrp donde SIGIO debería de ser enviado */
        uid_t uid, euid;        /* uid/euid del proceso estableciendo el dueño */
        int signum;             /* posix.1b rt señal para ser enviada en ES */
};

struct file {
        struct list_head f_list;
        struct dentry *f_dentry;
        struct vfsmount *f_vfsmnt;
        struct file_operations *f_op;
        atomic_t f_count;
        unsigned int f_flags;
        mode_t f_mode;
        loff_t f_pos;
        unsigned long f_reada, f_ramax, f_raend, f_ralen, f_rawin;
        struct file *f_next;
        struct file *f_prev;
        struct inode *f_inode;
        struct fown_struct f_owner;
        unsigned int f_uid, f_gid;
        int f_error;
        unsigned long f_version;
        void *private_data;
};
```

Veamos algunos de los campos de la estructura struct file:
f_list. Este campo enlaza la estructura del archivo con una (y sólo una) de las listas: (a) sb->s_files lista de todos los archivos abiertos en este sistema de archivos, si el correspondiente inodo no es anónimo, entonces dentry_open() (llamado por filp_open()) enlaza el archivo en esta lista; (b) fs/file_table.c:free_list, que contiene las estructuras de archivos sin utilizar; (c) fs/file_table.c:anon_list, cuando una nueva estructura de archivos es creada por get_empty_filp() es colocada en esta lista. Todas estas listas son protegidas por el spinlock files_lock.

f_dentry. La dentry (entrada de directorio) correspondiente a este archivo. La dentry es creada en tiempo de búsqueda de nombre y datos (nameidata) por open_namei() (o más bien path_walk() la cual lo llama a él) pero el campo actual file->f_dentry es establecido por dentry_open() para contener la dentry encontrada.
f_vfsmnt. El puntero a la estructura vfsmount del sistema de archivos conteniendo el archivo. Esto es establecido por dentry_open(), pero es encontrado como una parte de la búsqueda de nameidata por open_namei() (o más bien path_init() la cual lo llama a él).
f_op. El puntero a file_operations, el cual contiene varios métodos que pueden ser llamados desde el archivo. Esto es copiado desde inode->i_fop que es colocado aquí durante la búsqueda nameidata. Operaciones relacionadas con el archivo abierto.
f_count. Cuenta de referencia manipulada por get_file/put_filp/fput.
f_flags. Flags O_XXX desde la llamada al sistema open() copiadas allí (con ligeras modificaciones de filp_open()) por dentry_open() y después de inicializar O_CREAT, O_EXCL, O_NOCTTY, O_TRUNC. Modo de apertura del archivo especificado en la llamada a open.
f_mode. Una combinación de flags del espacio de usuario y modos, establecido por dentry_open() para determinar el modo de apertura del archivo especificado. El punto de conversión es almacenar los accesos de lectura y escritura en bits separados, por lo tanto uno los chequearía fácilmente como (f_mode & FMODE_WRITE) y (f_mode & FMODE_READ). Este campo es un poco particular, se basa en el modo de apertura especificado en la llamada a open y está formado por la conjunción de las constantes FMODE_READ y FMODE_WRITE, que indican respectivamente si son posibles las operaciones de lectura y escritura en ese archivo.
f_pos. La actual posición en el archivo para la siguiente lectura o escritura (posición actual en bytes desde el inicio del archivo). Bajo i386 es del tipo long long, esto es un valor de 64 bits.
f_reada, ***f_ramax***, ***f_raend***, ***f_ralen***, ***f_rawin***. Para soportar readahead (lectura anticipada). ***f_reada*** indica el número de bloques a leer de manera anticipada. ***f_ramax*** indica el número máximo de bloques a leer de manera anticipada. ***f_raend*** se refiere a la posición del primer byte en el archivo, tras la última página leída de manera anticipada. ***f_ralen*** indica el tamaño en bytes del último grupo de datos leídos de manera anticipada. ***f_rawin*** se refiere al tamaño de la ventana de lectura anticipada.
f_owner. Número de proceso o grupo de procesos propietario del archivo de E/S a recibir las modificaciones de E/S asíncronas a través del mecanismo SIGIO (fs/fcntl.c:kill_fasync()).
f_uid, ***f_gid***. Establece el identificador del usuario y el identificador del grupo del proceso que abrió el archivo, cuando la estructura del archivo es creada por get_empty_filp().
f_next. Puntero de encadenamiento al descriptor siguiente.
f_prev. Puntero de encadenamiento al descriptor anterior.
f_inode. Puntero al descriptor de inodo correspondiente.
f_error. Usado por el cliente NFS para devolver errores de escritura. Esto es establecido en fs/nfs/file.c y chequeado en mm/filemap.c:generic_file_write().
f_version. Mecanismo de versionado, incrementado (usando un event_global) cuando cambia **f_pos**. Es decir, indica el número de versión incrementado en cada uso del descriptor.
private_data. Datos privados para cada archivo, los cuales pueden ser usados por los sistemas de archivos o por otros controladores de dispositivos. Los controladores de dispositivos (en la presencia de devfs) pueden usar este campo para diferenciar entre múltiples instancias, en vez del clásico minor number codificado en file->f_dentry->d_inode->i_rdev.

Los descriptores de archivos se encadenan en una lista global, cuya dirección del primer elemento se almacena en la variable first_file, que contiene todos los descriptores de archivos.

Veamos ahora la estructura file_operations la cual contiene los métodos que serán llamados en los archivos. Destaquemos que es copiado desde inode->i_fop donde es establecido por el método s_op->read_inode(). Se declara en <include/linux/fs.h>:

```
struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct inode *, struct file *, loff_t, int);
        ssize_t (*read) (struct inode *, struct file *, char *, size_t, loff_t *);
        ssize_t (*write) (struct inode *, struct file *, const char *, size_t, loff_t *);
        int (*readdir) (struct inode *, struct file *, void *, filldir_t);
        int (*select) (struct inode *, struct file *, int, select_table *);
        unsigned int (*poll) (struct inode *, struct file *, struct poll_table_struct *);
```

```
        int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
        int (*mmap) (struct inode *, struct file *, struct vm_area_struct *);
        int (*open) (struct inode *, struct file *);
        int (*flush) (struct inode *, struct file *);
        int (*release) (struct inode *, struct file *);
        int (*fsync) (struct inode *, struct file *, struct dentry *, int datasync);
        int (*fasync) (struct inode *, struct file *, int);
        int (*check_media_change) (kdev_t dev);
        int (*revalidate) (kdev_t dev);
        int (*lock) (struct file *, int, struct file_lock *);
        ssize_t (*readv) (st struct inode *, ruct file *, const struct iovec *, unsigned long, loff_t *);
        ssize_t (*writev) (struct inode *, struct file *, const struct iovec *, unsigned long, loff_t *);
};
```

owner. Un puntero al módulo que es dueño del subsistema en cuestión. Sólo los controladores necesitan establecerlo a THIS_MODULE, los sistemas de archivos puede felizmente ignorarlos porque sus cuentas de módulos son controladas en el tiempo de montaje/desmontaje, en cambio los controladores necesitan controlarlo en tiempo de apertura/liberación.
llseek. Implementa la llamada al sistema lseek(). Normalmente es omitida y es usada fs/read_write.c:default_llseek(), la cual hace lo correcto.
read. Implementa la llamada al sistema read(). Los sistemas de archivos pueden usar mm/filemap.c:generic_file_read() para archivos regulares y fs/read_write.c:generic_read_dir().
write. Implementa la llamada al sistema write(). Los sistemas de archivos pueden usar mm/filemap.c:generic_file_write() para archivos regulares e ignorarlo para directorios.
readdir. Usado por los sistemas de archivos. Ignorado por los archivos regulares e implementa las llamadas al sistema readdir() y getdents() para directorios.
poll. Implementa las llamadas al sistema poll() y select().
ioctl. Implementa el controlador o los ioctls específicos del sistema de archivos. Nótese que los ioctls genéricos de los archivos como FIBMAP, FIGETBSZ, FIONREAD son implementados por niveles más altos y por lo tanto nunca leerán el método f_op->ioctl().
mmap. Implementa la llamada al sistema mmap(). Los sistemas de archivos pueden usar generic_file_mmap para archivos regulares e ignorarlo en los directorios.
open. Llamado en tiempo de apertura por dentry open(). Los sistemas de archivos raramente usan esto.
flush. Llamada en cada close() de este archivo, no necesariamente el último (ver el m´etodo release()). El único sistema de archivos que lo utiliza es en un cliente NFS para pasar a disco todas las páginas sucias. Nótese que esto puede devolver un error el cual será retornado al espacio de usuario que realizó la llamada al sistema close().
release. Llamado por la última close() de este archivo, esto es cuando file->f_count llega a 0. Aunque definido como un entero de retorno, el valor de retorno es ignorado por VFS.
fsync. Mapea directamente a las llamadas al sistema fsync()/fdatasync(), con el último argumento especificando cuando es fsync o fdatasync. Por lo menos no se realiza trabajo por VFS sobre esto, excepto el mapear el descriptor del archivo a una estructura de archivo (file = fget(fd)) y activar/desactivar el semáforo inode->i_sem. El sistema de archivos EXT2 ignora el último argumento y realiza lo mismo para fsync() y fdatasync().
fasync. Este método es llamado cuando cambia file->f_flags & FASYNC.
lock. Parte del mecanismo de bloqueo de la región del fcntl() POSIX de la porción específica del sistema de archivos. El único fallo es porque es llamado antes por una porción independiente del sistema de archivos, si tiene éxito pero el código de bloqueo estándar POSIX falla, entonces nunca será desbloqueado en un nivel dependiente del sistema de archivos
readv. Implementa la llamada al sistema readv().
writev. Implementa la llamada al sistema writev().

4.6.7. Objetos dentry y dentry Caché.

Hemos visto en la sección dedicada al modelo de archivos común que el VFS considera a cada directorio como un archivo que contiene una lista de archivos y otros directorios (veremos cuando hablemos del EXT2 cómo se implementa un sistema de archivos específico). Una vez que una entrada de directorio es leída en

memoria, ésta es transformada por el VFS en un objeto dentry basado en la estructura de datos dentry, cuyos campos se describen a continuación

```
struct dentry {
        atomic_t d_count
        unsigned int d_flags
        ruct inode *d_inode
        ruct dentry *d_parent
        ruct list_head d_hash
        ruct list_head d_lru
        ruct list_head d_child
        ruct list_head d_subdirs
        ruct list_head d_alias
        t d_mounted
        ruct qstr d_name
        signed long d_time
        ruct dentry_operations *d_op
        ruct super_block d_sb
        signed long d_vfs_flags
        id *d_fsdata
        signed char *d_iname
};
```

La descripción de cada uno de los campos se expone a continuación:
d_count. Contador de uso del objeto dentry
d_flags. Flags del objeto dentry
d_inode. Inodo asociado con el nombre de archivo
d_parent. Objeto dentry del directorio padre
d_hash. Punteros para la lista en la entrada de tabla hash
d_lru. Punteros para la lista de no utilizados
d_child. Punteros para la lista de objetos dentry incluidos en el directorio padre
d_subdirs. Para directories, lista de objetos dentry para subdirectorios
d_alias. Lista de inodos asociados (alias)
d_mounted. Flag establecido a 1 si y solo si el dentry es el punto de montaje par un sistema de archivos
d_name. Nombre de archivo
d_time. Utilizado por el método d_revalidate

El nombre del archivo se guarda en una estructura de datos extra, con su lognitud y valor hash.

```
sruct qstr{
        const unisgned char *name;
        unsigned int len;
        unsigned int hash;
}
```

El kernel crean un objeto dentry para cada componente de un nombre (pathname) de archivo que un proceso busca; el objeto dentry asocia el componente a su correspondiente inodo. Por ejemplo, cuando se busca el nombre /tmp/test, el kernel crea un objeto dentry para el directorio raíz /, un segundo objeto dentry para la entrada tmp del directorio raíz, y un tercer objeto dentry para la entrada test del directorio /tmp.

Notar que objetos dentry no tienen su correspondiente imagen en disco, y por tanto ningún campo se incluye en la estructura dentry para especificar que el objeto ha sido modificado. Los objetos dentry se almacenan en el caché de Slab Allocator denominado dentry_cache; los objetos dentry se crean y se destruyen mediante la invocación de las llamadas kmem_cache_alloc() y kmem_cache_free().

Cada objeto dentry puede estar en uno de los siguientes cuatro estados:

- *Libre (Free)*. El objeto dentry contiene información no válida y el VFS no lo utiliza. El área de memoria correspondiente es manejada por el Slab Allocator.

- *No usado (Unused)*. El objeto dentry no está correctamente utilizado por el kernel. El contador de uso d_counter del objeto está a 0, pero si el campo d_inode aún apunta al inodo asociado. El objeto dentry contiene información válida, pero sus contenido puede ser descartado si se requiere memoria.
- *En uso (In-use)*. El objeto dentry es utilizado actualmente por el kernel. El contador de uso d_count es positivo y el campo d_inode apunta al objeto inodo asociado. El objeto dentry contiene información válida y no puede ser descartado.
- *Negativo (Negative)*. El inodo asociado con el objeto dentry no existe, porque el correspondiente inodo de disco ha sido borrado o porque el objeto dentry fue creado para resolver un nombre (pathname)de un archivo inexistente. El campo d_inode del objeto dentry está a NULL, pero el objeto aún permanece dentro del dentry caché tal que futuras operaciones de búsqueda al mismo nombre de archivo puede ser resuelta rápidamente. El término "negative" puede confundirse, debido a que es posible que se den valore no negativos.

4.6.7.1. El dentry caché (Caché de Nombres)

Debido a que leer una entrada de directorio desde disco y construir el correspondiente objeto dentry requiere considerable tiempo, entonces tiene sentido mantener en memoria objetos dentry con los que se ha terminado pero que podrían utilizarse más tarde. Por ejemplo, usuario a menudo editan un archivo y entonces lo compilan, o lo editan y lo imprimen, o lo copian y entonces edita la copia. En este caso, el mismo archivo necesita ser accedido repetidamente.

Para maximizar eficientemente la gestión de objetos dentry, Linux utiliza el denominado dentry caché (o caché de nombres) que consta de dos tipos de estructuras de datos:

- Un conjunto de objetos dentry en los estados in-use, unused y negative,
- Una tabla hash para contener el objeto dentry asociado con un nombre de archivo dado y un directorio dado rápidamente. Como siempre, si el objeto requerido no se encuentra en el dentry caché, la función hash devuelve un valor nulo.

El dentry caché actúa también como un controlador para el *caché de inodos*. Los inodos en la memoria del kernel que están asociados con objetos dentry en estado "unused" no son descartados, ya que el dentry caché los está utilizando todavía. Por tanto, los objetos inodo se mantienen en RAM y pueden ser referenciados rápidamente por medio de los correspondientes objetos dentry.

Todos los objetos dentry en estado "unused" están incluidos en una lista LRU (Least Recentrly Used) doblemente enlazada LRU ordenada por tiempo de inserción. En otras palabras, el objeto dentry que fue descargado en última instancia se ubica al frente de la lista, por tanto los objetos dentry accedidos menos recientemente están siempre cerca del final de la lista. Cuando el dentry caché tiene que disminuir, el kernel elimina elementos del final de esta lista, preservando los objetos accedidos más recientemente. Las direcciones del primer y último elementos de la list aLRU se almacenan en los campos next y prev de la variable dentry_unused. El campo d_lru del objeto dentry contiene punteros a los objetos dentry adyacentes en la lista.

Cada objeto dentry "in-use" se inserta en la lista doblemente enlazada especificada por el campo i_dentry del correspondiente objeto inodo (ya que cada nodo podría estar asociado con varios enlaces fuertes (hard link, por lo que se necesita una lista). El campo d_alias del objeto dentry almacena la dirección de los objetos adyacentes en la lista. Ambos campos son del tipo list_head.

Un objeto dentry "in-use" puede pasar al estado "negative" cuando el último enlace fuerte al correspondiente archivo se elimina. En este caso, en este caso el objeto dentry se mueve a la lista LRU de objetos dentry "unused". Cada vez que el kernel disminuye el dentry caché, éste mueve objetos dentry en estado negative hacia el final de la lista LRU para que sean gradualmente liberados.

La tabla hash se implementa por medio de un array de dentry_hashtable.

```
static struct list_head *dentry_hastable;
```

Cada elemento es un puntero a una lista de objetos dentry que confluyen al mismo valor en la tabla hash. El tamaño de array depende de la cantidad de memoria instalada en el sistema. El campo d_hash del objeto dentry contiene punteros a los elementos adyacentes en la lista asociada con un único valor hash. La función hash genera su valor en función de la dirección del objeto dentry del directorio y del nombre del archivo.

Un nuevo objeto dentry se crea utilizando la función:

struct dentry ***d_alloc**(struct dentry *parent, const struct qstr *name);

Para ello el *kernel* asigna memoria para el nuevo objeto dentry, introduce el objeto dentry como parent, escribe la lista de hijos en la lsita subdirs de su padre. En este momento, el estado del objeto dentry es "negative" y no contiene ninguna información sobre inodo. Para realizar las operaciones restantes se utiliza la siguiente función:

void ***d_instantiate**(struct dentry *entry, struct inode *inode);

que valida el objeto dentry, teniendo en cuenta el inodo en el objeto dentry y la entrada i_entry del objeto inode en la entrada de la lista alias. Entonces, esta función coloca el objeto dentry en la lista hash correspondiente utilizando la función d_rehash().

El spin lock dcache_lock protege las estructuras de datos del dentry caché de accesos concurrentes en sistemas multiprocesador. La función d_lookup() busca en la tabla hash (en el dentry caché) en base a un objeto dentry padre dado y un nombre de archivo.

struct dentry ***d_lookup**(struct dentry *parent, struct qstr *name);

Los métodos asociados a un objeto dentry se denominan *operaciones dentry*, ellas se describen en la estructura dentry_operations, cuyos direcciones se almacenan en el campo d_op. Aunque alunos sistemas de archivos definen sus propios métodos dentry, los campos están normalmente a NULL y el VFS los reemplaza con funciones por defecto.

```
struct dentry_operations {
        int (*d_revalidate)(struct dentry *, int);
        int (*d_hash)(struct dentry *, struct qstr *);
        int (*d_compare)(struct dentry *, struct qstr *, struct qstr *);
        int (*d_delete)(struct dentry);
        int (*d_release)(struct dentry *);
        int (*d_iput)(struct dentry *, struct inode *);
};
```

A continuación se describen los métodos de la estructura dentry_operations, en el orden en que ellos aparecen en la tabla dentry_operations:

d_revalidate(dentry, flag). Determina si el objeto dentry es todavía válido antes de utilizarlo para traducir un nombre de archivo. La función VFS por defecto no hace nada, aunque sistemas en red pueden especificar sus propias funciones.

d_hash(dentry, name). Genera un valor hash; esta función es una función específica para el sistema de archivos para la tabla hash de objetos dentry. El parámetro dentry identifica el directorio que contiene el componente. El parámetro name de tipo qstr apunta a una estructura que contiene como componentes el nombre el nombre a buscar y el valor producido por la función hash.

d_compare(dir, name1, name2). Compara dos nombres de archivos, name1 debería pertenecer al directorio referenciado por el parámetro dir. La función VFS por defecto es un string-matching normal. Sin embargo, cada sistema de archivos puede implementar este método de forma propia. Por ejemplo, MS-DOS no distingue entre mayúsculas y minúsculas.

d_delete(dentry). Se llama cuando la última referencia a un objeto dentry es eliminada (d_count se pone a 0). La función VFS por defecto no hace nada.

d_release(dentry). Se llama cuando un objeto dentry va a ser liberado (descargada para el Slab Allocator). La función VFS por defecto no hace nada.

d_iput(dentry, inode). Se llama cuando un objeto dentry se convierte en "negative", es decir, pierde su inodo. La función VFS por defecto invoca iput() para liberar el objeto inodo.

4.6.8. Administración de Puntos de Montaje y Superbloque

Bajo Linux, la información sobre los sistemas de archivos montados es mantenida en dos estructuras separadas: super_block y vfsmount. El motivo para esto es que Linux permite montar el mismo sistema de archivos (dispositivo de bloque) bajo múltiples puntos de montaje, lo cual significa que el mismo super_block puede corresponder a múltiples estructuras vfsmount.

Veamos en primer lugar la estructura *struct super_block* (superbloque VFS), declarado en <include/linux/fs.h>:

```
struct super_block {
        struct list_head s_list;
        kdev_t s_dev;
        unsigned long s_blocksize;
        unsigned char s_blocksize_bits;
        unsigned char s_lock;
        unsigned char s_dirt;
        unsigned char s_rd_only;
        struct file_system_type *s_type;
        struct super_operations *s_op;
        struct dquot_operations *dq_op;
        unsigned long s_flags;
        unsigned long s_magic;
        struct inode *s_covered;
        struct inode *s_mounted;
        struct dentry *s_root;
        wait_queue_head_t s_wait;
        struct list_head s_dirty;
        struct list_head s_files;
        struct block_device *s_bdev;
        struct list_head s_mounts;
        struct quota_mount_options s_dquot;
        union {
                struct minix_sb_info minix_sb;
                struct ext2_sb_info ext2_sb;
                void *generic_sbp;
        } u;                    // información dependiente del tipo de sistema de archivos
        struct semaphore s_vfs_rename_sem;
        struct semaphore s_nfsd_free_path_sem;
};
```

Las diversos campos en la estructura *super_block* son:
s_list. Una lista doblemente enlazada de todos los superbloques activos; nótese que no he dicho "de todos los sistemas de archivos montados" porque bajo Linux uno puede tener múltiples instancias de un sistema de archivos montados correspondientes a un superbloque simple.
s_dev. Para sistemas de archivos que requieren un bloque para ser montado en él. Para los sistemas de archivos FS_REQUIRES_DEV, es la i_dev del dispositivo de bloques (identificador del dispositivo). Para otros (llamados sistemas de archivos anónimos) esto es un entero MKDEV(UNNAMED MAJOR, i) donde i es el primer bit no establecido en la matriz unnamed_dev_in_use, entre 1 y 255 incluidos (fs/super.c:get_unnamed_dev()/put_unnamed_dev()). Ha sido sugerido muchas veces que los sistemas de archivos anónimos no deberían de usar el campo s_dev.
s_blocksize, ***s_blocksize_bits***. Tamaño del bloque y log2(tamaño del bloque) en bytes.
s_lock. Indica cuando un superbloque está actualmente bloqueado por lock_super()/unlock_super().
s_rd_only. Indicador de lectura exclusiva.
s_dirt. Establece cuando el superbloque está modificado, e inicializado cuando es vuelto a ser escrito a disco.
s_type. Puntero a struct file_system_type del sistema de archivos correspondiente. El método read_super() del sistema de archivos no necesita ser establecido como VFS fs/super.c:read_super(), lo establece para ti si el read_super() que es específico del sistema de archivos tiene éxito, y se reinicializa a NULL si es que falla.
s_op. Puntero a la estructura struct super_operations, la cual contiene métodos específicos del sistema de archivos para leer/escribir inodos, etc. Es el trabajo del método read_super() del sistema de archivos inicializar s_op correctamente.
dq_op. Operaciones relacionadas con la cuota de disco.
s_flags. Flags de superbloque (opciones de montaje).

s_magic. Número mágico del sistema de archivos que le permite reconocer la presencia de un sistema de archivos en una partición. Usado por el sistema de archivos de minix para diferenciar entre múltiples tipos del mismo.
s_covered. Puntero al descriptor de inodo del punto de montaje.
s_mounted. Puntero al descriptor de inodo del directorio raíz del sistema de archivos.
s_root. dentry de la raíz del sistema de archivos. Es trabajo de read_super() leer el inodo raíz desde el disco y pasárselo a d_alloc_root() para asignar la dentry e instanciarlo. Algunos sistemas de archivos dicen "raiz" mejor que "/" y por lo tanto usamos la función más genérica d_alloc() para unir la dentry a un nombre.
s_wait. Cola de espera de los procesos esperando para que el superbloque sea desbloqueado. Es una variable utilizada para sincronizar los accesos concurrentes al descriptor.
s_dirty. Una lista de todos los inodos "dirty" (modificados). Recordar que si un inodo está "dirty" (inode->i_state & I_DIRTY) entonces su lista "dirty" específica del superbloque es enlazada a través de inode->i_list.
s_files. Una lista de todos los archivos abiertos en este superbloque. Útil para decidir cuándo los sistemas de archivos pueden ser "remontados" como de sólo lectura, ver fs/file_table.c:fs_may_remount_ro() el cual va a través de la lista sb->s_files y deniega el "remontar" si hay archivos abiertos para escritura (file->f_mode & FMODE_WRITE) o archivos con desenlaces pendientes (inode->i_nlink == 0).
s_bdev. Para FS_REQUIRES_DEV, éste apunta a la estructura block_device describiendo el dispositivo en el que el sistema de archivos está montado.
s_mounts. Una lista de todas las estructuras vfsmount, una por cada instancia montada de este superbloque.
s_dquot. Más miembros de diskquota.
s_nfsd_free_path_sem. Este campo es usado por knfsd cuando convierte un manejador de archivos (basado en el número de inodo) en una dentry. Tal como construye un camino en el árbol del directorio caché desde abajo hasta arriba, quizás exista durante algún tiempo un subcamino de dentrys que no están conectados al árbol principal. Este semáforo asegura que hay sólo siempre un camino libre por sistema de archivos. Nótese que los archivos no conectados (o otros no directorios) son permitidos, pero no los directorios no conectados.

Las operaciones de superbloque son descritas en la estructura *super_operations* declarada en <include/linux/fs.h>:

```
struct super_operations {
        void (*read_inode) (struct inode *);
        void (*write_inode) (struct inode *, int);
        void (*put_inode) (struct inode *);
        void (*delete_inode) (struct inode *);
        void (*notify_change) (struct inode *, struct iattr *);
        void (*put_super) (struct super_block *);
        void (*write_super) (struct super_block *);
        int (*statfs) (struct super_block *, struct statfs *);
        int (*remount_fs) (struct super_block *, int *, char *);
        void (*clear_inode) (struct inode *);
        void (*umount_begin) (struct super_block *);
};
```

read_inode. Lee el inodo desde el sistema de archivos. Esta operación es llamada sólo desde fs/inode.c:get_new_inode(), y desde iget4() (y por consiguiente iget()). Si un sistema de archivos quiere usar iget() entonces read_inode() debe de ser implementado (en otro caso get_new_inode() fallará). Mientras el inodo está siendo leido está bloqueado (inode->i state = I_LOCK). Cuando la función regresa, todos los que están esperando en inode->i_wait son despertados. El trabajo del método read_inode() del sistema de archivos es localizar el bloque del disco que contiene el inodo a ser leído y usar la función del buffer bread() para leerlo e inicializar varios campos de la estructura de inodos, por ejemplo el inode->i_op y inode->i_fop para que los niveles VFS conozcan qué operaciones pueden ser efectuadas en el inodo o archivo correspondiente. Los sistemas de archivos que no implementan read_inode() son ramfs y pipefs. Por ejemplo, ramfs tiene su propia función de generación de inodos ramfs_get_inode() con todas las operaciones de inodos llamándola cuando se necesita.
write_inode. Escribe el inodo en un sistema de archivos. Similar a read_inode() en que necesita localizar el bloque relevante en el disco e interactuar con el buffer llamando a mark_buffer_dirty(bh). Este método es

llamado en los inodos dirty (aquellos marcados como dirty por mark_inode_dirty()) cuando el inodo necesita ser sincronizado individualmente o como parte de la actualización entera del sistema de archivos.
put_inode. Llamado cuando la cuenta de referencia es decrementada, es decir, cuando un inodo deja de utilizarse.
delete_inode. Llamado cuando inode->i_count y inode->i_nlink llegan a 0. El sistema de archivos borra la copia en disco del inodo y llama a clear_inode() en el inodo VFS para "terminar con él con el perjuicio extremo".
notify_change. Esta operación es llamada por el sistema de archivos cuando los atributos de los inodos han sido modificados. El parámetro iattr indica las modificaciones efectuadas.
put_super. Llamado en las últimas etapas de la llamada al sistema umount() para notificar al sistema de archivos que cualquier información mantenida por el sistema de archivos sobre esa instancia tiene que ser liberada (el superbloque deja de utilizarse, es decir, cuando el sistema de archivos correspondiente se desmonta). Normalmente esto desencadena un brelse() sobre el bloque conteniendo el superbloque y kfree() cualesquiera bitmaps asignados para bloques libres, inodos, etc.
write_super. Llamado cuando el superbloque ha sido modificado y necesita rescribirse en disco. Debería de encontrar el bloque conteniendo el superbloque (normalmente mantenido en el área sb-private) y mark_buffer_dirty(bh). También debería de inicializar el flag sb->s_dirt.
statfs. Implementa las llamadas al sistema fstatfs()/statfs() para obtener la información de control del sistema de archivos. Nótese que el puntero a la estructura struct statfs pasado como argumento, es el puntero del *kernel*, no un puntero del usuario, por lo tanto no necesitamos hacer ninguna E/S al espacio de usuario. Si no está implementada entonces statfs() fallará con ENOSYS.
remount_fs. Llamado cuando el sistema de archivos está siendo "remontado", es decir, cuando las opciones de montaje se modifican por una llamada a la primitiva mount con la opción MS_REMOUNT.
clear_inode. Llamado desde el nivel VFS clear_inode(). Los sistemas que soportan datos privados en la estructura del inodo (a través del campo generic_ip) deben liberarse aquí.
umount_begin. Llamado durante el desmontaje forzado para notificarlo al sistema de archivos de antemano, por lo tanto puede ser lo mejor para asegurarse que nada mantiene al sistema de archivos ocupado. Actualmente usado sólo por NFS. Esto no tiene nada que hacer con la idea del soporte de desmontaje forzado del nivel genérico de VFS.

Ahora, observemos qué sucede cuando montamos un sistema de archivos en disco (FS_REQUIRES_DEV). La implementación de la llamada al sistema mount() está en fs/super.c:sys_mount() que es justo un interfaz que copia las opciones, el tipo del sistema de archivos y el nombre del dispositivo para la función do_mount(), la cual realiza el trabajo real:

El controlador del sistema de archivos es cargado si se necesita y la cuenta de referencia del módulo es incrementada. Nótese que durante la operación de montaje, la cuenta del sistema de archivos es incrementada dos veces (una vez por do_mount() llamando a get_fs_type() y otra vez por get_sb_dev() llamando a get_filesystem() si read_super() tuvo éxito). El primer incremento es para prevenir la descarga del módulo mientras estamos dentro del método read_super(), y el segundo incremento es para indicar que el módulo está en uso por esta instancia montada. Obviamente, do_mount() decrementa la cuenta antes de regresar, por lo tanto, después de todo, la cuenta sólo crece en 1 después de cada montaje.

Desde, fs type->fs_flags & FS REQUIRES DEV es verdad, el superbloque es inicializado por una llamada a get_sb_bdev(), la cual obtiene la referencia al dispositivo de bloques e interactúa con el método read_super() del sistema de archivos para rellenar el superbloque. Si todo va bien, la estructura super_block es inicializada y tenemos una referencia extra al módulo del sistema de archivos y una referencia al dispositivo de bloques subyacente.

Una nueva estructura *vfsmount* es asignada y enlazada a la lista sb->s_mounts y a la lista global vfsmntlist. El campo vfsmount de mnt_instances nos permite encontrar todas las instancias montadas en el mismo superbloque que este. El campo mnt_list nos permite encontrar todas las instancias para todos los superbloques a lo largo del sistema. El campo mnt_sb apunta a este superbloque y mnt_root tiene una nueva referencia a la dentry sb->s_root.

El *kernel* mantiene dos listas diferentes que referencian los sistemas de archivos montados. La tabla super_blocks contiene NR_SUPER descriptores utilizados por el VFS para todas las operaciones de E/S. La segunda lista memoriza las correspondencias entre los nombres de dispositivos sobres los que se encuentran

los sistemas de archivos y los nombres de punto de montaje. El tipo de elementos de esta lista es la estructura *vfsmount*, que está declarada en <include/linux/mount.h>:

```
struct vfsmount {
        kdev_t mnt_dev;             // Identificador de dispositivo
        char * mnt_devname;         // Nombre del archivo especial que representa el dispositivo
        char *mnt_dirname;          // Nombre del punto de montaje
        unsigned int mnt_flags;     // Opciones de montaje
        structu semaphore;          // Semáforo utilizado para bloquear el descriptor
        struct super_block *mnt_sb; // Puntero al descriptor del bloque correspondiente
        struct file *mnt_quotas[MAXQUOTAS];
                                    // Descriptores de archivos de descripción de cuotas
        time_t mnt_iexp[MAXQUOTAS];
                                // Lapso de espera utilizado en el desbordamiento de la cuota de inodos
        time_t mnt_bexp[MAXQUOTAS];
                                // Lapso de espera utilizado en el desbordamiento de la cuota de bloques
        struct vfsmount *mnt_next;  // Puntero al descriptor siguiente de la lista
};
```

4.6.9. Administración de Cuotas de Disco

Las cuotas de disco se gestionan mediante descriptores de cuotas. Cada uno de estos descriptores se asocia a un usuario o grupo particular. La estructura *struct dquot*, definida en el archivo de cabecera <include/linux/quota.h> tiene la siguiente forma:

```
struct dquot {
        unsigned int dq_id;         // Identificador de usuario o grupo a quien se aplica esta cuota
        short dq_type;              // Tipo de cuota, USRQUOTA o GRPQUOTA
        kdev_t dq_dev;              // Identificador de dispositivo
        short dq_flags;             // Estado del descriptor
        short dq_count;             // Número de utilizaciones del descriptor
        struct vfsmount *dq_mnt;    // Puntero al descriptor del SA montado afectado
        struct dqblk dq_dqb;        // Límites y uso
        struct wait_queue *dq_wait; // Variable utilizada para sincronizar los accesos concurrentes
                                    // al descriptor
        struct dquot *dq_prev;      // Puntero al descriptor anterior en la lista
        struct dquot *dq_next;      // Puntero al descriptor siguiente en la lista
        struct dquot *dq_hash_prev; // Puntero al descriptor anterior en la lista hash
        struct dquot *dq_hash_next; // Puntero al descriptor siguiente en la lista hash
};
```

De la misma manera que los descriptores de inodos, los descriptores de cuotas se encadenan en varias listas; una lista global que contiene todos los descriptores y listas hash para acelerar las búsquedas. Además el módulo de gestión de cuotas mantiene estadísticas sobre la gestión de descriptores de cuotas, para ello, podemos estudiar la estructura dqstats declarada en el archivo de cabecera <include/linux/quota.h>.

La estructura dquot_operations contiene punteros a funciones llamadas por el VFS. Los diferentes sistemas de archivos pueden implementar estas operaciones, existiendo una versión genérica en VFS.

```
struct dquot_operations {
        void (*initialize) (struct inode *, short);
        void (*drop) (struct inode *);
        int (*alloc_block) (const struct inode *, unsigned int);
        int (*alloc_inode) (const struct inode *, unsigned int);
        void (*free_block) (const struct inode *, unsigned int);
        void (*free_inode) (const struct inode *, unsigned int);
        int (*transfer) (const struct inode *, struct iattr *, char);
};
```

initialize. Se llama a esta operación para inicializar los descriptores de cuotas correspondientes al inodo referenciado por parámetro. El parámetro type especifica el tipo de cuota a inicializar: USRCUOTA, GRPQUOTA o el valor –1 para indicar los dos tipos.
drop. La operación drop se llama para liberar los descriptores de cuotas asociados al inodo referenciado por parámetro.
alloc_block. La operación alloc_block se lama con el objetivo de verificar que nuevos bloques pueden asignarse al inodo pasado como parámetro.
alloc_inodo. La operación alloc_inode se llama con el objetivo de verificar que es posible asignar nuevos inodos al propietario y al grupo del inodo pasado como parámetro. El número de inodos a signar se especifica en el parámetro unsigned int count.
free_block. La operación free_block se llama en la liberación de bloques para el inodo referenciado como parámetro, con el objetivo de actualizar el número de bloques contabilizados. El parámetro unsigned int count especifica el número de bloques a liberar.
free_inode. La operación free_inode se llama en la liberación de inodos al propietario y al grupo del inodo referenciado como parámetro, con el objetivo de actualizar el número de inodos contabilizados. El parámetro unsigned int count especifica el número de inodos a liberar.
transfer. La operación transfer se llama al cambiar de propietario o de grupo el archivo referenciado por el parámetro inodo. Transfiere el número de bloques e inodos contabilizados de un descriptor de cuotas u otro.

4.7. BUFFER CACHÉ

4.7.1. Introducción

- Archivos ordinarios ⇒ Dispositivos de almacenamiento masivo. Cada archivo en UNIX tiene asociado un inodo. El inodo contiene la información necesaria para que un proceso pueda acceder al archivo, como puede ser: propietario, derechos de acceso, tamaño, localización en el sistema de archivos, tipo de archivo, etc. En UNIX tenemos 4 tipos de archivos.
  - Archivos ordinarios (también llamados regulares o de datos) ⇒ Bytes organizados como un array lineal.
  - Directorios.
  - Archivos de dispositivo o archivos especiales.
  - Tuberías (pipes).
- Lectura/Escritura directa en disco en cada acceso al sistema de archivos ⇒ Minimizar el número de accesos al disco, utilizando un *buffer caché*.

4.7.1.1. Entradas/Salidas Utilizando Archivos Especiales.

- El acceso a los dispositivos se efectúa mediante archivos especiales o archivos de dispositivo. Un archivo especial aparece en el árbol de archivos de la misma manera que un archivo regular (tiene asociado un inodo), pero no hay datos a los que referenciar. A cada archivo especial corresponde un controlador de dispositivo cuyo código se integra al *kernel*. Estos archivos especiales van a permitir a los procesos comunicarse con los dispositivos periféricos (discos, cintas, impresoras, terminales, redes, etc.).

- Una vez que un proceso ha abierto un archivo especial, sus peticiones de lectura y escritura no se transmiten al sistema de archivos, sino al controlador de dispositivo correspondiente. Este último efectúa las entradas y salidas físicas sobre el dispositivo, cuando el proceso utiliza las llamadas al sistema *read* y *write*.

- Existen dos tipos de archivos especiales:
  - Los archivos especiales en *modo bloque*: corresponden a dispositivos estructurados en bloques (array de bloques de tamaño fijo), como los discos y las cintas, a los que se accede proporcionando un número de bloque al leer o escribir. Las entradas/salidas se efectúan mediante las funciones del *buffer caché* que el *kernel* gestiona para acelerar la velocidad de transferencia de los datos. El *buffer caché* se implementa vía software y no hay que confundirlo con las memorias caché de acceso rápido que disponen la mayoría de los computadores. Los bloques se cargan en memorias intermedias (buffers) llamando a las funciones apropiadas (*bread* y *breada*), que llaman a un módulo de E/S con el objetivo de leer o escribir bloques en disco. La función de este módulo de E/S consiste en mantener una lista de peticiones de E/S en curso y satisfacerlas. Cuando uno o más bloques deben leerse o escribirse en disco, la petición de E/S se añade a la lista correspondiente al dispositivo físico. Esta lista de dispositivo es explorada por el controlador de dispositivo que efectúa las E/S una a una. Con el objetivo de optimizar los tiempos de desplazamiento de las cabezas de L/E del disco, la lista de peticiones se mantiene ordenada por el número de sector físico. De este modo, las peticiones que respectan a sectores próximos se efectúan minimizando el desplazamiento de las cabezas, según el algoritmo de ascensor.
  - Los archivos especiales en *modo carácter*: corresponden a dispositivos no estructurados, como los puertos serie y paralelo, sobre los que se puede leer y escribir los datos byte a byte, generalmente de forma secuencial. Por lo tanto, en los archivos de dispositivo en modo carácter la información no se organiza según una estructura concreta y es vista por el *kernel*, o por el usuario, como una secuencia lineal de bytes. En la transferencia de datos entre el *kernel* y el dispositivo no participa el *buffer caché* y por lo tanto se va a realizar a menor velocidad.

- Como ya es sabido, en UNIX, los archivos de dispositivo, al igual que el resto de los archivos, tienen asociado un inodo. En el caso de los archivos ordinarios o de los directorios, el inodo nos indica los bloques donde se encuentran los datos del archivo (bloques de datos), pero en el caso de los archivos de dispositivo no hay datos a los que referenciar. En su lugar, el inodo asociado a un archivo de dispositivo contiene 2 números conocidos como *major number* y *minor number*. En realidad, estos dos últimos números los utiliza el *kernel* para buscar dentro de unas tablas (block device switch table y character device switch table) una colección de rutinas que le permiten controlar el dispositivo, estas rutinas constituyen realmente el controlador del dispositivo. Es decir, cada archivo especial se caracteriza por tres atributos:
  - Su tipo (bloque o carácter).
  - Su *major number* (número mayor): este número identifica el controlador que gestiona el dispositivo. Es decir, el tipo de dispositivo que se trata (disco, cinta, terminal, etc.).
  - Su *minor number* (número menor): este número permite al controlador conocer el dispositivo físico sobre el cual debe actuar. Es decir, el número de unidad dentro del dispositivo.

- Cuando se realiza una llamada al sistema para realizar una operación de Entrada/Salida sobre un archivo especial (archivo de dispositivo), el *kernel* se encarga de llamar al controlador de dispositivo adecuado y lo que ocurre a continuación es completamente transparente para el usuario.

- Los archivos especiales se encuentran generalmente en el directorio */dev*.

4.7.1.2. Gestión de Buffers en Memoria

- UNIX actualiza el estado de un buffer que forma parte de la memoria principal, para ello utiliza una tabla de descriptores y cada uno de ellos corresponde a una página de memoria.

- UNIX mantiene en memoria un *buffer caché*. Todos los buffers asociados a un dispositivo y un bloque de datos que se registran en una lista hash. Estas páginas pueden contener a código ejecutado por los procesos o al contenido de archivos proyectados en memoria. Recordamos que cada sistema de archivos (tipo UNIX) conserva en disco una tabla de descriptores de archivos. Estos descriptores, llamados inodos (todos los archivos tienen asociado un inodo que nos indica los bloques que forman el archivo y dónde se encuentran), contienen la información de control utilizada para gestionar los archivos, especialmente los atributos de archivos así como las direcciones de los bloques de datos que los componen. El número de archivo único es utilizado por el *kernel* como índice en la tabla de inodos, con el objetivo de convertirlo en descriptor de archivo.

- Este *buffer caché* se gestiona dinámicamente. Cuando el contenido de un buffer asociado a un dispositivo y a un bloque debe cargarse en memoria, se asigna un nuevo buffer, se inserta en el *buffer caché*, y su contenido se lee desde el disco. En los accesos posteriores al contenido del buffer, su carga ya no es necesaria porque está ya presente en memoria.

- Como sabemos, cuando UNIX está falto de memoria, selecciona buffers para su descarte. Los buffers presentes en el *buffer caché* que no han sido accedidos desde hace tiempo se descartan (LRU). Cuando se utiliza la llamada al sistema *read*, la lectura se efectúa cargando en memoria los buffers correspondientes al inodo, permitiendo utilizar el *buffer caché* y aprovechar buffers ya presentes en memoria.

4.7.2. El *Buffer Caché*

- Cabecera de buffers y lista de buffers. Los buffers son bloques de disco donde se almacenan datos que provienen o van a disco.
  - Inicialmente todos los buffers están vacíos y organizados en una lista de libres (free_list).
  - *Kernel* necesita un buffer ⇒ Elimina de libres, lo modifica y lo enlaza en una nueva lista hash (datos válidos) ⇒ Lista de válidos (hash_table).
  - El buffer debe contener información del dispositivo y del bloque, ya que éste puede contener información de diferentes dispositivos y diferentes bloques.
  - Para reducir tiempos de búsqueda de un bloque en la lista de válidos ⇒ Organizarla como una tabla hash según (número de dispositivo + número de bloque **Mod** número_listas_hash).
  - Punteros adicionales: puntero al área de datos, y punteros a los buffers siguiente y anterior en cada lista (libres y válidos).
  - Inicialización del buffer caché ⇒ reserva de espacio (Linux no fija el tamaño del buffer caché en la inicialización del sistema): número de buffers y tamaño de los buffers configurable de acuerdo con el tamaño de la memoria disponible y restricciones del sistema.

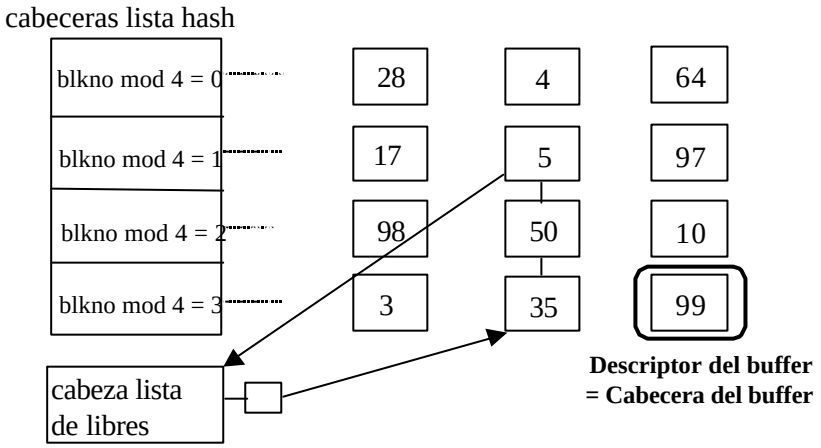


Figura 4.14. Cabecera de buffers y listas de buffers.

- La gestión del *buffer caché* interactúa con la gestión de memoria (paginación). El contenido de las memorias intermedias (buffers) se coloca en páginas de memoria y el estado de las páginas de memoria debe consultarse o modificarse a menudo. Por ejemplo, Linux, para realizar la gestión del *buffer caché* utiliza la tabla *mem_map* que contiene un descriptor para cada página de memoria. La estructura del descriptor de un buffer se presenta de forma detallada en el apartado (3) *Descriptor de buffer* y la estructura de datos que soporta el *buffer caché* se describe en el epígrafe (4) *Estructura del buffer pool.* Entre los campos más utilizados por el *buffer caché* y las funciones que los gestionan (según categorías) podemos destacar:
  - *count*: Campo que contiene el número de referencias al buffer ubicado en el *buffer caché*.
  - *flags*: Campo que contiene el estado del buffer.
  - *buffers*: Campo que contiene la dirección del primer descriptor de buffer cuyo contenido se sitúa en la página.
  - Función de inicialización del *buffer caché*.
  - Funciones para la gestión de las listas de buffers (memoria intermedias).
  - Funciones para la realización de entradas/salidas.
  - Funciones que modifican el tamaño del *buffer caché*.
  - Funciones para la gestión de dispositivos.
  - Funciones de servicio que permiten acceder a los buffers (memorias intermedias).
  - Funciones de reescritura del contenido de los buffers (memorias intermedias) en disco.
  - Funciones para la gestión de los *clusters* (grupo de buffers cuyos contenidos son contiguos en memoria y se corresponden con bloques contiguos en disco).

4.7.3. Descriptor de Buffer

- Inicialización del sistema ⇒ Asignar espacio a los buffers. En Linux la función *buffer_init* se llama al arrancar el sistema con el objetivo de inicializar el *buffer caché*. Calcula el número de listas hash en función del tamaño de memoria disponible, asigna estas listas llamando a una función especial (*vmalloc*), y las inicializa. Finalmente, llama a la función *grow_buffers* con el objetivo de asignar algunas memorias intermedias (buffers). Linux, contrariamente a otros sistemas operativos, el tamaño del buffer caché no se fija en la inicialización del sistema. El buffer caché está inicialmente vacío y cambia de tamaño en el transcurso de la ejecución del sistema. Cuando el *kernel* necesita buffers suplementarios, se asignan nuevas páginas de memoria en el buffer caché con el objetivo de ampliarlo. Por otro lado, cuando al sistema le falta memoria, puede eliminar ciertos buffers así como las páginas de memoria que las contienen, lo cual reduce el tamaño del buffer caché.

- Linux (igual que en cualquier sistema UNIX) actualiza el estado de cada página que forma parte de la memoria y para ello utiliza una tabla de descriptores, apuntada por la variable *mem_map*.

- Composición de cada buffer.
  - Array de memoria (*char *b_data*) ⇒ Copia en memoria del bloque de disco. Esta estructura de datos es direccionada por un puntero ubicado en la cabecera del buffer (*struct buffer_head *buffers*). El contenido de esta estructura de datos puede corresponder a código ejecutado por los procesos o al contenido de archivos proyectados en memoria.

  - Cabecera del buffer (Figura 4.15).
    + Número de dispositivo (*kdev_t b_dev*) ⇒ Campo para especificar a que dispositivo esta asociado el buffer (*unsigned nr_device*).
    + Número de bloque (*unsigned long b_blocknr*) ⇒ relacionado con el inodo correspondiente al contenido del buffer (*struct inode *inode*) y con el desplazamiento de dicho buffer en el inodo (*unsigned long offset*).
    + Puntero a un array de datos para el buffer ⇒ Puntero que direcciona el contenido donde se encuentra el buffer en memoria (*struct buffer_head *buffers*). Éste es un array en memoria cuyo contenido es el bloque de datos asociado a un archivo.
    + Contador (*b_count*) ⇒ Registra el número de referencias al buffer y sirve para seleccionar el buffer a descartar de la memoria según el algoritmo de sustitución establecido (LRU).
    + Estado del buffer (*b_state*).
      * Buffer bloqueado (utilizado por otro proceso, ocupado) ⇒ El buffer está bloqueado en memoria y debe liberarse tras el fin de la entrada/salida en curso.
      * Error ⇒ Se ha producido un error en la carga del buffer.
      * Contiene datos válidos ⇒ El buffer contiene datos válidos.
      * Contiene datos no válidos ⇒ El buffer no contiene datos válidos.
      * Escritura retardada ⇒ Escribir el contenido del buffer en disco antes de reasignarlo, para ello se realiza una escritura asíncrona en disco del contenido del buffer y cuando termina se libera dicho buffer y se pone a la cabeza de la lista de buffers libres.
      * Buffer modificado ⇒ Se ha realizado una operación de entrada/salida (escritura) sobre el buffer pero el contenido de dicho buffer no se ha actualizado en disco.
      * Leyendo o escribiendo el contenido del buffer en disco ⇒ El buffer ha sido accedido y está realizando una operación de entrada/salida.
      * Reserva de buffer ⇒ El buffer está reservado para un uso futuro, no es posible acceder a él.

+ Punteros, existe un conjunto de punteros para gestionar todos los buffers (número de dispositivo y número de bloque) en memoria asociados a un inodo de un archivo según una lista hash.
  * Puntero al siguiente buffer en la lista de buffer libres ⇒ Este puntero direcciona al siguiente buffer libre en la lista de buffer libres (*struct page *b_next_free*).
  * Puntero al anterior buffer en la lista de buffer libres ⇒ Este puntero direcciona al anterior buffer libre en la lista de buffer libres (*struct page *b_prev_free*).
  * Puntero al siguiente buffer en la lista hash ⇒ Este puntero direcciona al siguiente buffer en la lista hash (*struct page *b_next*).
  * Puntero al anterior buffer en la lista hash ⇒ Este puntero direcciona al anterior buffer en la lista hash (*struct page *b_prevt*).

▪ Además, el descriptor del buffer puede tener otros campos, como por ejemplo *b_flushtime*, el cual se pone a la fecha en la que el contenido del buffer debe escribirse en disco. *b_lru_time* que almacena la hora en la que el contenido del buffer se ha utilizado por última vez (se utiliza para el remplazamiento de los buffer menos usados recientemente). *b_wait* que es una variable (struct wait_queue *) que se utiliza para sincronizar los accesos concurrentes a dicho buffer. *b_reqnext* es un puntero al buffer siguiente que forma parte de la misma operación de entrada/salida. b_this_page es otro puntero al descriptor del buffer siguiente cuyo contenido está en la misma página de memoria.

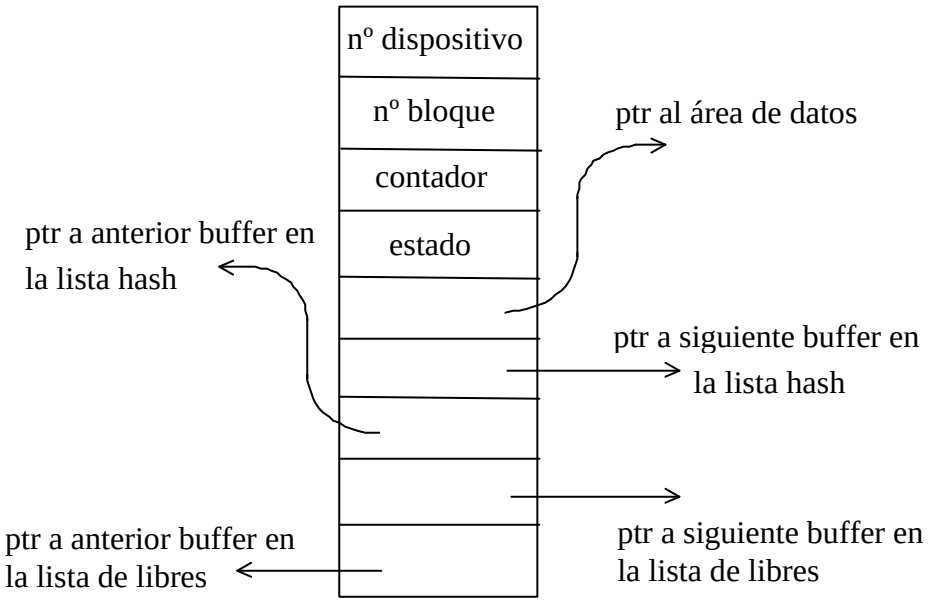


Figura 4.15. Descriptor de buffer.

4.7.3.1. Descriptor de buffer en Linux

El buffer caché en Linux gestiona los buffers (memorias intermedias) asociados a los bloques de disco. La estructura *buffer_head*, declarada en el archivo <include/linux/fs.h>, define el formato de los descriptores de buffer tal y como se indica a continuación:

```
struct buffer_head {
        unsigned long b_blocknr;        // Número de bloque en el dispositivo
        kdev_t b_dev;                   // Identificador del dispositivo lógico
        kdev_t b_rdev;                  // Identificador del dispositivo real (físico)
        unsigned long b_rsector;        // Número de sector de inicio del bloque en el
                                        // dispositivo físico
        struct buffer_head *b_next;     // Puntero al descriptor siguiente
        struct buffer_head *b_this_page; // Puntero al descriptor del buffer cuyo contenido está
```

```
                                    // en la misma página de memoria
    unsigned long b_state;          // Estado del buffer (por ejemplo, BH_Dirty, BH_Lock)
    struct buffer_head *b_next_free; // Puntero al descriptor libre siguiente
    unsigned int b_count;           // Número de utilizaciones de este bloque
    unsigned long b_size;           // Tamaño del bloque en bytes
    char *b_data;                   // Puntero al bloque de datos (contenido del buffer)
    unsigned int b_list;            // Lista en la cual aparece este buffer
    unsigned long b_flushtime;      // Momento en el cual el contenido de este buffer
                                    // (modificado) deberá escribirse en disco
    unsigned long b_lru_time;       // Momento cuando se utilizó por última vez este buffer
    struct wait_queue *b_wait;      // Variable utilizada para sincronizar los accesos
                                    // concurrentes a este buffer
    struct buffer_head *b_prev;     // Puntero al descriptor anterior en la lista hash
                                    // doblemente enlazada
    struct buffer_head *b_prev_free; // Puntero al descriptor libre anterior
    struct buffer_head *b_reqnext;  // Puntero al buffer siguiente que forma parte de la
                                    // misma petición de E/S
};
```

Los buffers (descriptores de tipo buffer_head que contienen toda la información necesaria para que el kernel sepa cómo gestionar el buffer) se colocan en varias listas encadenadas según su estado y su contenido (por ejemplo, lista de buffers modificados, bloqueados, compartidos, etc.). Además, una función hash, basada en el identificador de dispositivo y el número de bloque, se utiliza para colocar los buffers en varias listas de hashing. De este modo la búsqueda de un buffer determinado es mucho más rápida.

Como hemos visto en la estructura *buffer_head*, el campo b_data de cada buffer_head almacena la dirección de inicio del correspondiente buffer. Debido a que un marco de página puede almacenar varios buffers, el campo b_this_page apunta al siguiente buffer en la página, y además, este campo facilita el almacenamiento y la recuperación de marcos de página completos. El campo b_block_nr almacena el número de bloque lógico, es decir, el índice de bloque dentro de la partición de disco.

El campo b_state almacena los siguientes flags: (1) BH_Uptodate, determina si el buffer contiene datos válidos; (2) BH_Dirty, determina si el buffer está modificado (contiene datos que deben ser escritos a un dispositivo de bloque); (3) BH_Lock, determina si el buffer está bloqueado que ocurre cuando el buffer está involucrado en una transferencia de disco; (4) BH_Req, determina si el correspondiente bloque es solicitado y tiene datos válidos; (5) BH_Mapped, establece si el buffer es mapeado a disco, es decir, si los campos b_dev y b_blocar del correspondiente buffer son significativos; (6) BH_New, indica si el correspondiente bloque de archivo acaba de ser asignado o nunca ha sido accedido; (7) BH_Async, indica si el buffer está siendo procesado por una operación end_buffer_io_async(); (8) BH_Wait_IO, se utiliza para retardar la operación de flush a disco sobre el buffer cuando se reclama memoria; (9) BH_Launder, determina cuando el buffer está siendo enviado (flush) a disco en el momento que se reclama memoria; (10) BH_JBD, indica si el buffer se utiliza para sistemas de archivos “journaling”.

El campo bd_dev identifica el dispositivo virtual que contiene el bloque almacenado en el buffer, mientras que el campo b_rdev identifica el dispositivo real. Esta distinción, que no tiene sentido para discos duros normales, ha sido introducida para RAID (), que es un modelo que almacena unidades que consiste en varios discos operando en paralelo. Por razones de seguridad y eficiencia, archivos almacenados en un array RAID son esparcidos sobre varios discos que las aplicaciones consideran como un disco lógico único. Además, el campo b_blocar que representa el número de bloque lógico, es necesario para especificar la unidad de disco específica en el campo b_rdev y el correspondiente número de sector en el campo b_rsector.

4.7.4. Estructura del Buffer Pool

4.7.4.1. Estructura de los buffers libres (lista LRU).

- Estructura de los buffers libres ⇒ Lista de buffers circular doblemente enlazada con una cabecera de buffer falsa que identifica su comienzo y su final (Figura 4.16) ⇒ La lista de buffers libres mantienen el orden según la política de menos utilizado recientemente (LRU).
  - LRU se basa en la idea de que los buffers que se han utilizado mucho en las últimas instrucciones probablemente se utilizarán mucho en las siguientes. Por ello, los buffers que hace mucho tiempo que no se utilizan, probablemente seguirán sin utilizarse durante largo tiempo. Esta idea sugiere un algoritmo fácil de llevar a la práctica el cual nos indica que cuando ocurra una falta de página (buffer), se reemplazará el buffer que haya estado más tiempo sin utilizarse.

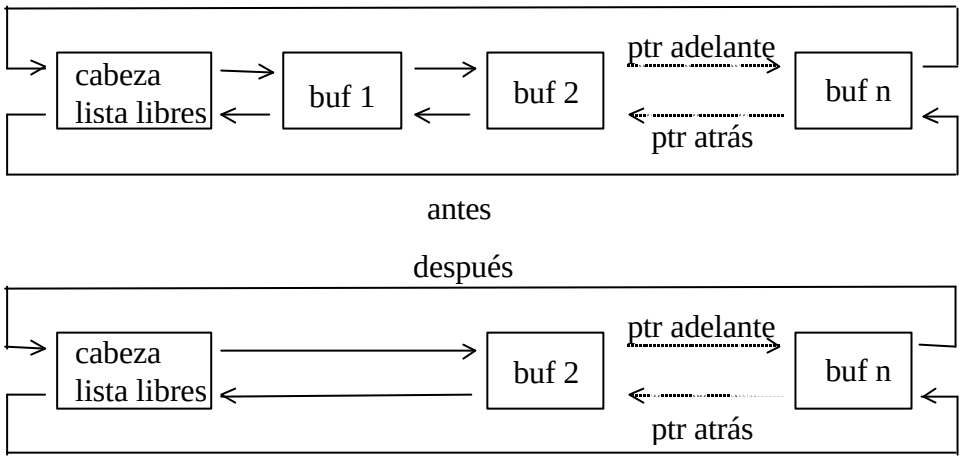


Figura 4.16. Lista de buffers libres.

- Extraer buffers libres de la lista de buffers libres:
  - De la cabeza de la lista de buffers libres cuando necesita un buffer libre.
  - Del medio si encuentra un buffer (bloque) particular que se estaba buscando (búsqueda secuencial en dicha lista).
- Devolver buffers libres al buffer pool, donde se colocarán:
  - Al final de la lista.
  - Alguna vez en la cabeza.
  - Nunca en el medio.

4.7.4.2. Organización del Buffer Pool (lista hash)

- Listas LRU separadas y doblemente enlazadas según una función hash.
  - *Función hash*(número de dispositivo, número de bloque).

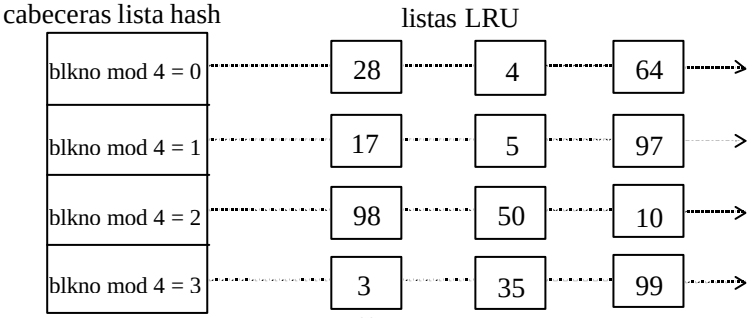


Figura 4.17. Buffers en listas hash.

4.7.4.3. Acceso a Bloques de Disco (a través del buffer caché)

- Buscar buffer[número de dispositivo, número de bloque]. Las claves de acceso a la lista hash nos las dan en función del número de dispositivo lógico del inodo y el número de bloque del inodo.
  - Organización del buffer pool en listas LRU separadas.
  - *Función hash*(número de dispositivo, número de bloque).
- En general, un buffer siempre se encuentra en una lista LRU (lista de libres o lista hash), de forma que el *kernel* tiene 2 formas de encontrarlo:
  - Buscando en la lista hash si se busca un buffer particular.
  - Buscando en lista de libres si se está buscando cualquier buffer libre.

4.7.5. Gestión de las Listas de Buffers

- Varias funciones internas permiten gestionar las listas (lista de buffers libres y lista hash) en las que los buffers (memorias intermedias) se registran. En Linux, las variables más importantes que se utilizan para mantener dichas listas son:
  - *hash_table* ⇒ Tabla que contiene los punteros al primer buffer de cada lista hash. *lru_list* ⇒ Lista que contiene los punteros al primer buffer de cada lista LRU.
  - *nr_hash* ⇒ Número de listas hash (listas LRU).
  - *free_list* ⇒ Lista que contiene los punteros al primer buffer disponible para cada tamaño de bloque (dispositivo), esta lista está organizada según la política LRU.
  - *nr_buffers* ⇒ Número de buffers asignados.
  - *nr_buffers_type* ⇒ Tabla que contiene el número de buffers registrados en cada lista.
  - *nr_buffers_size* ⇒ Tabla que contiene el número de buffers asignados para cada tamaño de bloque (dispositivo).

- En Linux las funciones más importantes que aseguran la gestión de estas listas son:
  - *__wait_on_buffer* y *wait_on_buffer* ⇒ Estas funciones permiten sincronizar varios procesos en modo *kernel* que acceden al mismo buffer (memoria intermedia).
  - *_hashfn* y *hashfn* ⇒ Estas funciones implementan la función hash utilizada en las listas hash.
  - *insert_into_hash_queue* ⇒ Esta función inserta un buffer en la lista de hash correspondiente.
  - *remove_from_hash_queue* ⇒ Esta función suprime un buffer de la lista de hash correspondiente.
  - *insert_into_lru_list* ⇒ Esta función inserta un buffer en la lista LRU correspondiente.
  - *remove_from_lru_list* ⇒ Esta función suprime un buffer de la lista LRU correspondiente.
  - *insert_into_free_list* ⇒ Esta función inserta un buffer en la lista de buffers libres.
  - *remove_from_free_list* ⇒ Esta función suprime un buffer de la lista de buffers libres.
  - *put_last_lru* ⇒ Esta función desplaza un buffer al final de la lista LRU correspondiente.
  - *put_last_free* ⇒ Esta función desplaza un buffer al final de la lista de buffers libres.
  - *find_buffer* ⇒ Esta función efectúa una búsqueda de un buffer. Explora la lista hash correspondiente y devuelve la dirección del descriptor o el valor NULL en caso de error.
  - *set_writetime* ⇒ Esta función posiciona el campo *b_flushtime* del descriptor del buffer a la fecha actual, si el contenido del buffer se ha modificado, en caso contrario este campo del descriptor se pone a cero.
  - *refile_buffer* ⇒ Esta función se llama cuando un proceso libera un buffer. Examina el estado de los buffers, y los cambia eventualmente de lista LRU. La fecha de última utilización del buffer se pone a la fecha actual de esta función.
  - *put_free_buffer* ⇒ Esta función se llama para liberar un buffer que ya no se utiliza. Añade el buffer a la lista de buffers libres apuntada por *free_list*, y utiliza la función *wake_up* sobre la variable *buffer_wait* con el objetivo de despertar los procesos en espera de buffers libres.

- *get_more_buffer_heads* ⇒ Esta función se llama para asignar descriptores de buffers suplementarios (libres) y repite el proceso para poder asignar nuevos buffers. Esta función en cada pasada realiza las siguientes acciones:
  + Comprueba la variable *free_list* para determinar si existen buffers libres (disponibles), si es así retorna al proceso que la llama.
  + Llama a *get_free_page* para obtener una página de memoria.
  + Si la asignación fracasa, llama a *sleep_on* sobre la variable *buffer_wait* con el objetivo de suspender al proceso que está en espera de un buffer libre.
  + Si la asignación tiene éxito, la página asignada se descompone en descriptores de buffer, y añade cada uno de ellos en la lista de buffers disponibles (libres) apuntada por *free_list*.
- *get_free_buffer_head* ⇒ Esta función se llama para obtener el primer descriptor de buffer libre de la lista apuntada por *free_list*.

4.7.6. Situaciones en la Asignación de un Buffer

- El *kernel* realiza la asignación de buffers como resultado de una llamada al sistema que requiera de un buffer para que el proceso pueda realizar una lectura o escritura de un bloque en un determinado dispositivo.
- Búsqueda de un bloque por parte del *kernel* ⇒ número de dispositivo y número de bloque en la lista hash.

  hash_table = hashfn(nr_device, nr_block) % nr_hash
- Acceso a disco ⇒ ¿Están los datos buscados en el buffer pool del *buffer caché*?.
  - Si ⇒ Se lee directamente del buffer en cuestión sin necesidad de acceder a disco.
  - No ⇒ Necesita un buffer libre para alojar los datos una vez que sean leídos desde disco.
- Escribir un bloque de datos en disco ⇒ el *kernel* busca el bloque en el buffer pool y si no lo encuentra asignará uno de los libres.
- Cuando el *kernel* encuentra un buffer ⇒ si está libre lo marca como ocupado (moviéndolo de la lista hash) y lo bloquea para que ningún otro proceso pueda utilizarlo mientras lo usa el proceso que lo ha solicitado. El proceso bloqueado competirá cuando el bloque (en el buffer caché) quede libre.

4.7.6.1. Asignar un Buffer para un Bloque de Disco (getblk). Situaciones

- (1) Buffer en lista hash y libre.
- (2) Buffer no en lista hash ⇒ asignar un buffer libre por el *kernel.*
- (3) Buffer no en lista hash ⇒ asignar por parte del *kernel* un buffer libre pero encuentra un buffer marcado como de escritura retardada ⇒ escribir a disco y asignar otro buffer libre.
- (4) Buffer no en lista hash y la lista de buffers libres está vacía (el *kernel* no encuentra ningún buffer libre).
- (5) Buffer en lista hash, pero está ocupado (bloqueado).

- Búsqueda de un buffer para asignar = algoritmo *getblk* ⇒ éste es el algoritmo que utiliza el *kernel* para asignar buffers del buffer pool (buffer caché) para la lectura y escritura de bloques en disco.

```
Algoritmo getblk   // Búsqueda de un buffer para asignar
entrada:    número de dispositivo (nr_device), número de bloque (nr_block)
salida:     buffer ocupado
{
    while (no se encuentre el bloque)
    {
        if (bloque está en lista hash)
        {
            if (buffer está ocupado)                                  // SITUACIÓN 5
            {
                dormir en evento: Espera_por_Buffer(nr_device+nr_block)_Ocupado;
                continuar;
            }
            marcar buffer como ocupado;                               // SITUACIÓN 1
            retirar buffer de lista de Libres;
            return buffer;
        }
        else
        {
            if (no hay buffer en lista de libres)                     // SITUACIÓN 4
            {
                dormir en evento: Espera_por_un_Buffer_Libre;
                continuar;
            }
            retirar buffer de lista de libres;                        // SITUACIÓN 3
            if ( el buffer está marcado con "escritura retardada")
            {
                escribir el buffer en el disco;
                continuar;
            }
            retirar el buffer de lista hash antigua;                  // SITUACIÓN 2
            poner el buffer en la nueva lista hash;
            return buffer;
        }
    }
}
```

Algoritmo para la búsqueda de un buffer para asignar (getblk).

4.7.6.2. Después de Asignar un Buffer para un Bloque de Disco. Situaciones

- El *kernel* puede leer los datos del disco al buffer y manipularlos o bien escribir datos en el buffer y posiblemente al disco.
- El *kernel* marca el buffer como ocupado para que otros procesos no puedan cambiar el contenido mientras esté ocupado ⇒ mantener la integridad de los datos en el buffer pool.

4.7.6.3. Liberar un Buffer (brelse)

- Finalizando el uso del buffer, el *kernel* lo libera.
- Fin de manipulación ⇒ liberar el buffer ocupado. Una vez liberado el buffer se coloca en la lista libres, según la política LRU (despertar procesos).
- El nivel de ejecución del procesador aumenta cuando se manipula la lista de buffers libres para cada inodo.
- Situaciones para la libración de un buffer ⇒ ponerlo al principio o al final de la lista de libres:
  - (1) Buffer con datos inválidos (error de entrada o error de salida) o buffer es antiguo ⇒ Principio de la lista de buffers libres.
  - (2) Buffer con datos válidos y buffer no antiguo ⇒ Final.

- El lugar en la lista de buffers libres depende del momento en que fueron usados ⇒ LRU. Por ello, cuando se utiliza un buffer y luego se libera ⇒ Final; y si no ha sido utilizado ⇒ Principio.

```
Algoritmo brelse   // Liberación de un buffer
entrada:    buffer bloqueado
salida:     void
{
    despertar todos los procesos dormidos en el evento: Espera_algún_buffer_libre;
    despertar todos los procesos dormidos en el evento: Espera_buffer_libre;     // buffer bloqueado
    elevar el nivel del procesador para bloquear interrupciones;
    if ( el buffer contiene datos válidos y el buffer no es antiguo )
        poner el buffer al final de la lista de libres;                          // SITUACIÓN 2
    else
        poner el buffer al principio de la lista de libres;                      // SITUACIÓN 1
    bajar el nivel del procesador para permitir interrupciones;
    desbloquear el buffer;                                                       // buffer desocupado
}
```

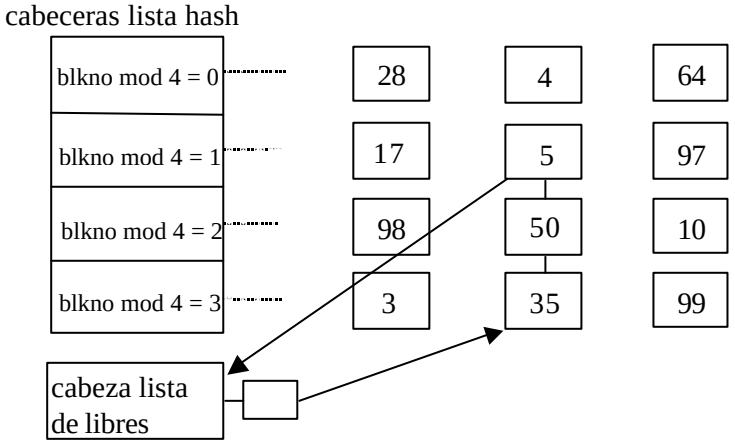
Algoritmo para la liberación de un buffer (brelse).

4.7.6.4. Asignar un Buffer para un Bloque de Disco (getblk). Algoritmos

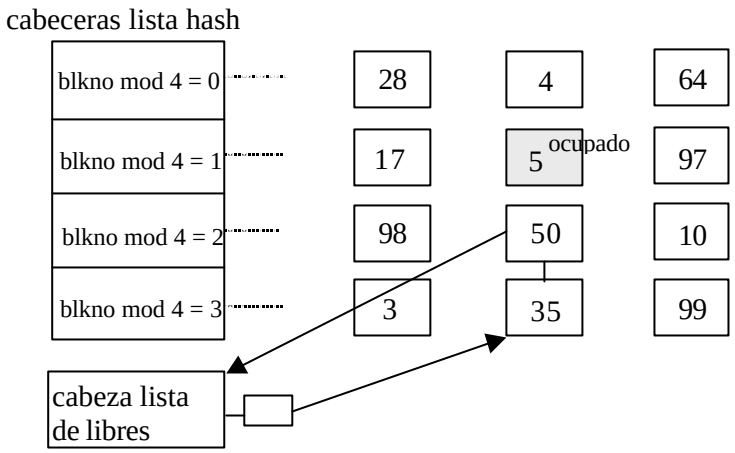
- Buscar un bloque (número de dispositivo y número de bloque).
- Lista hash = según la función hash y el valor obtenido previamente.

4.7.6.4.1. Situación 1. Buffer en Lista Hash y libre.

- Busca en la lista hash el buffer según el valor obtenido de la función hash y lo encuentra. Recordemos que la función hash depende del número de dispositivo y del número de bloque.
- ¿El buffer requerido está en la lista de buffers libres?
  - Si ⇒ Lo marca como ocupado y lo elimina de la lista de libres.
  - No ⇒ Pasamos a la situación 2.
- Ejemplo para está situación (Figura 4.17). Búsqueda del bloque 5.



(a) Búsqueda del bloque 5 en la primera lista hash (bloque libre).

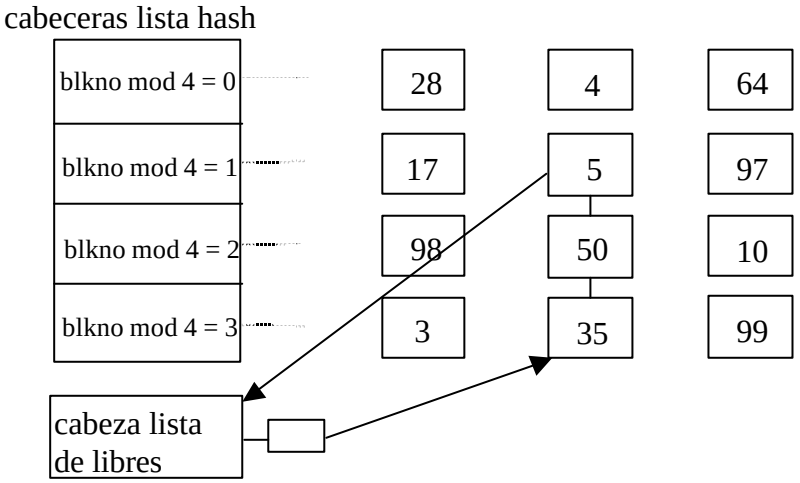


(b) Eliminar bloque 5 de la lista de libres (marcarlo como ocupado).

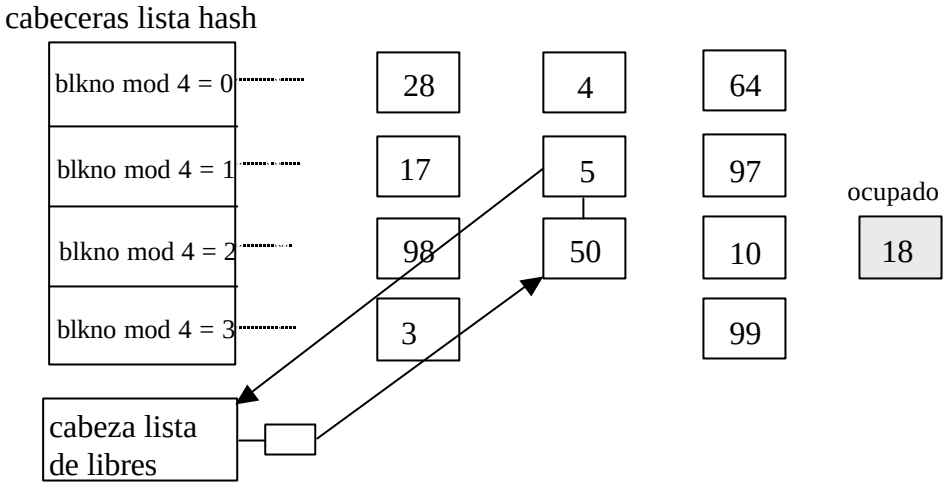
Figura 4.17. Búsqueda de un buffer. El buffer está en su lista hash.

4.7.6.4.2. Situación 2. Buffer no está en Lista Hash.

- Busca en la lista hash y no lo encuentra. No puede estar en otra lista hash ⇒ No está en el *buffer caché*.
- Elimina el primer buffer de la lista de libres.
- Sobre el buffer seleccionado de la lista de libres, se llevan a cabo las siguientes acciones:
  - Lo elimina de la lista de buffers libres.
  - Reasigna el número de dispositivo y de bloque de la cabecera del buffer.
  - Coloca el buffer en la lista hash apropiada (al final de la lista) y lo marca como ocupado.
- Ejemplo para está situación (Figura 4.18). Asignación de un buffer (bloque).



(a) Búsqueda del bloque 18. No está en el buffer caché.

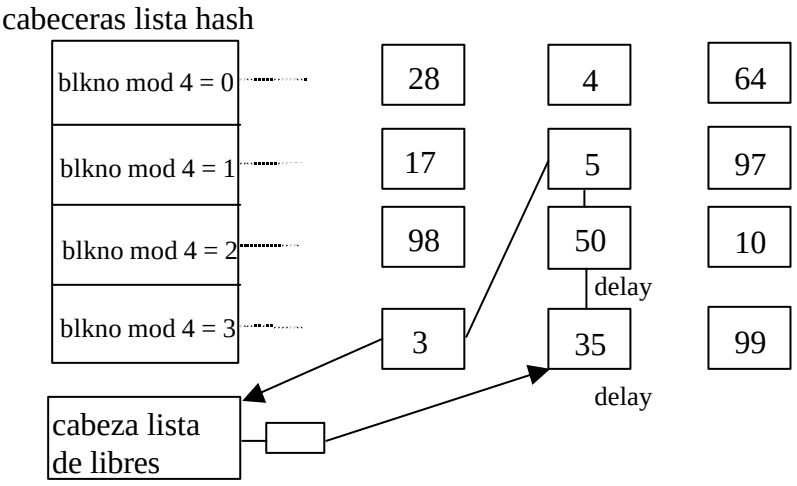


(b) Eliminar primer bloque (buffer 35) de lista de libres. Asignar un nuevo buffer al bloque 18 y marcarlo como ocupado.

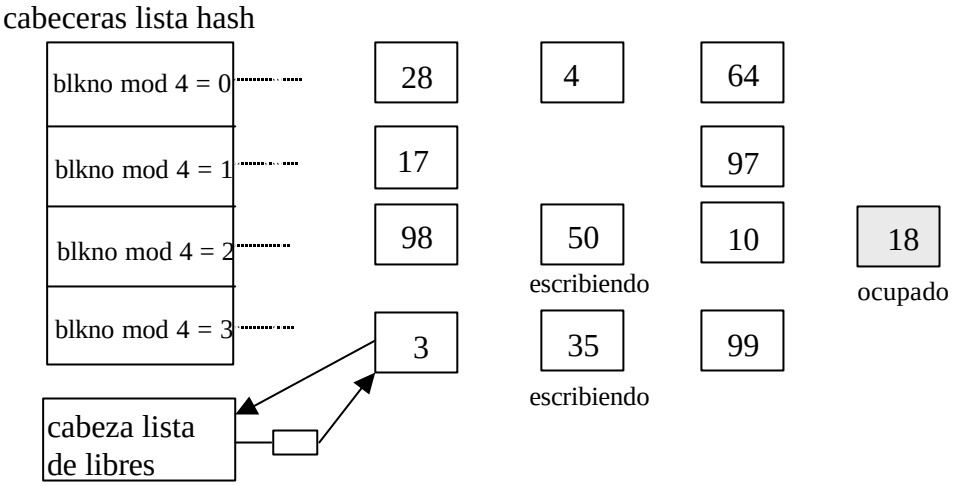
Figura 4.18. Asignación de un buffer cuando no está en la lista hash.

4.7.6.4.3. Situación 3. Buffer no en Lista Hash pero Existen Buffers de Escritura Retardada.

- Similar a la situación 2.
- Estado del buffer como de *escritura retardada* ⇒ Escribir el contenido del buffer en disco antes de reasignarlo, para ello se realiza una escritura asíncrona (cuando un buffer se está escribiendo en disco no está disponible) en disco del contenido del buffer y cuando termina se libera dicho buffer y se pone al principio de la lista de buffers libres. En general, la escritura retardad incida el estado que se produce cuando el proceso que usó el buffer ya ha pedido su escritura, pero el *kernel* no la ha realizado, quedando a la espera de que este bloque sea requerido nuevamente.
- Encuentra el buffer marcado como de escritura retardada.
- Escribe el contenido del buffer marcado como de escritura retardada en disco antes de reasignarlo.
  - Comienza una escritura asíncrona al disco.
  - Cuando termine, libera el buffer y lo pone al principio de la lista de libres.
- Intenta asignar un nuevo buffer de la lista de libres mientras escribe los buffers marcados como escritura retardada de forma asíncrona. Retirarlo de libres e insertarlo al final de su lista hash.
- Ejemplo para está situación (Figura 4.19). Tercera situación en la asignación de un buffer.



(a) Búsqueda del bloque 18. Hay dos bloques de escritura retardada (delay) en la lista de libres (buffers con los bloques 35 y 50).



(b) Escribiendo los bloques 35 y 50. Asignación de los buffers 5 y 18.

Figura 4.19. Tercera situación para la asignación de un buffer.

- Estado delay = escritura retardad. Además, un buffer que se está escribiendo en disco de forma asíncrona no está disponible ⇒ no está en lista de buffers libres.
- Proceso de situación 3:
  - Petición bloque 18.
  - Búsqueda no exitosa en la lista hash ⇒ el *kernel* debe asignar un buffer libre.
  - Al buscar en la lista de libres (algoritmo *bfreelist*) encuentra: 35 y 50 como buffers con su estado en escritura retardada (delay) ⇒ lanzados a escritura asíncrona en disco y movidos de la lista de buffers libres. Entonces encuentra el buffer 5 como buffer libre.
  - Buffer 5 ⇒ eliminado de la lista de buffers libres y enlazado al final de la lista hash para ser ocupado por el bloque ¡18.

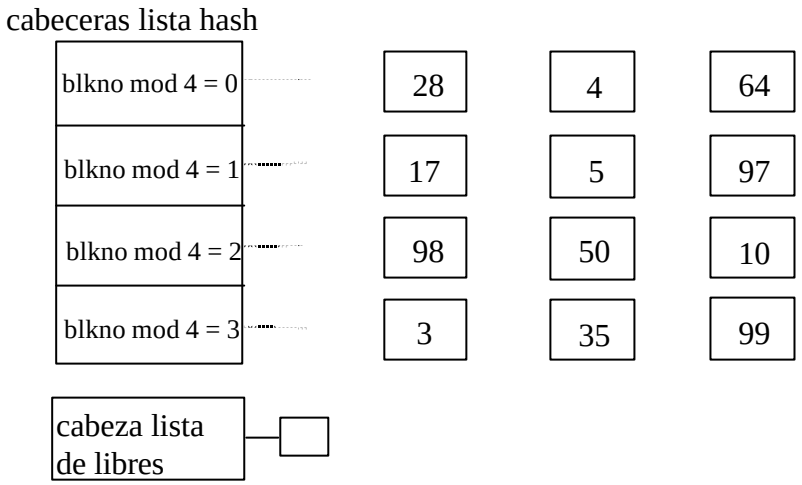
```
Algoritmo bfreelist        // búsqueda en la lista de buffers libres desde el principio
entrada:    lista de buffers libres
salida:     buffer o NULL
{
    if (lista de buffers libres vacía)
        return NULL;
    else
    {
        while (buffer marcado como "escritura retardada")
        {
            escribir el buffer en disco de forma asíncrona;
            al finalizar la escritura del buffer ⇒ buffer liberado y colocado al principio de la lista de
                buffer libres;
        }
        retirarlo de la lista de buffers libres;
        return buffer;
    }
}
```

Algoritmo para la búsqueda de buffers libres desde el principio (bfreelist).

4.7.6.4.4. Situación 4. Buffer no en Lista Hash y Lista de Libres Vacía.

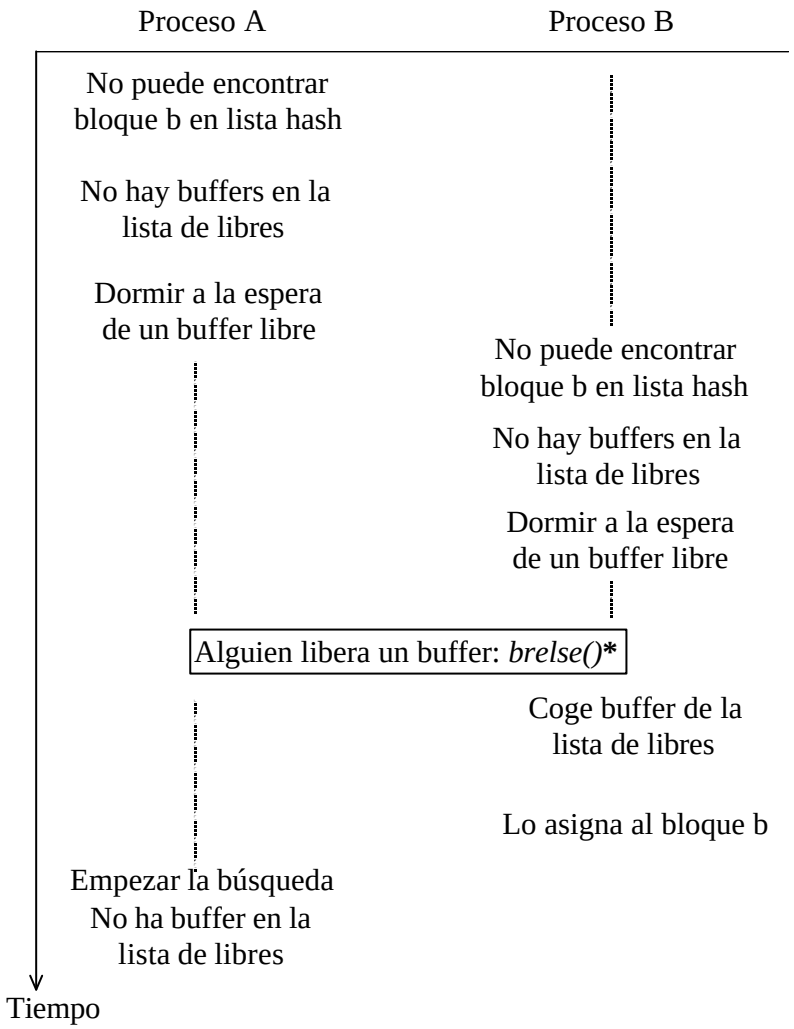
- El *kernel* dormirá el proceso que solicitó el buffer hasta que aparezca alguno disponible (buffer cualquier que quede libre).
- Ejemplo para está situación (Figura 4.20). Cuarta situación en la asignación de un buffer.
- Situación para el estudio del caso (dos procesos compitiendo por un buffer, Figura 4.21):
  - Proceso A en ejecución.

- Dicho proceso A no encuentra el bloque (buffer) requerido b en su lista hash ⇒ Intenta asignar un buffer de la lista de libres.
- No hay buffers en la lista de libres ⇒ El proceso en ejecución (proceso A) duerme hasta que otro proceso libere un buffer, haciendo un *brelse*.
- Mientras ocurre la liberación del buffer, otro proceso B solicita al *kernel* el mismo bloque (buffer) b, repitiéndose el mismo proceso de búsqueda y el *kernel* duerme también al proceso B en el mismo evento.
- Cuando el *kernel* planifique (*scheduler*) otra vez el proceso A.
  + Busca otra vez el bloque (buffer) en su lista hash.
  + Si varios procesos estuvieran esperando el mismo bloque (buffer) ⇒ Puede que no se le asigne un buffer y los procesos estén en espera del buffer de forma indefinida.
  + La Figura 4.21 muestra dos procesos compitiendo por un buffer libre en la lista de libres.



Búsqueda del bloque 18. Lista de buffers libres vacía.

Figura 4.20. Cuarta situación en la asignación de un buffer.



*** El *kernel* reactiva los procesos A y B, pero B se apodera del buffer**

Figura 4.21. Contienda (situación de concurso) por un buffer libre (cuando la lista de libres está vacía).

4.7.6.4.5. Situación 5. Buffer en Lista Hash, pero está Ocupado (bloqueado, en uso por otro proceso).

- El *kernel* duerme (bloqueado) al proceso en el evento que espera hasta que este buffer específico quede libre.
- Ejemplo para está situación (Figura 4.22). Quinta situación en la asignación de un buffer.
- Situación para el estudio del caso (dos procesos compiten por un buffer ocupado, Figura 4.23):
  - El proceso A, busca un bloque y se le asigna un buffer, pero va a dormir antes de liberarlo (buffer bloqueado a la salida de *getblk* ⇒ debe realizarse sobre él una operación de E/S desde el disco).
  - Mientras el proceso A duerme, el *kernel* planifica a través del scheduler un proceso B.
  - El proceso B intenta acceder al bloque cuyo buffer está ocupado por el proceso A.
  - El proceso B encontrará el bloque ocupado en la lista hash, marca el buffer "en demanda" y duerme esperando a que el proceso A lo libere.
  - El proceso A liberará el buffer y notará que está "bajo demanda" ⇒ Despertará a todos los procesos que esperaban que se liberase el buffer.
  - Cuando se planifique el proceso B, éste deberá verificar que el buffer está libre.
  - El proceso B debe verificar también que el buffer contiene el bloque solicitado.
  - Al final, el proceso B encontrará su bloque en el buffer correspondiente.

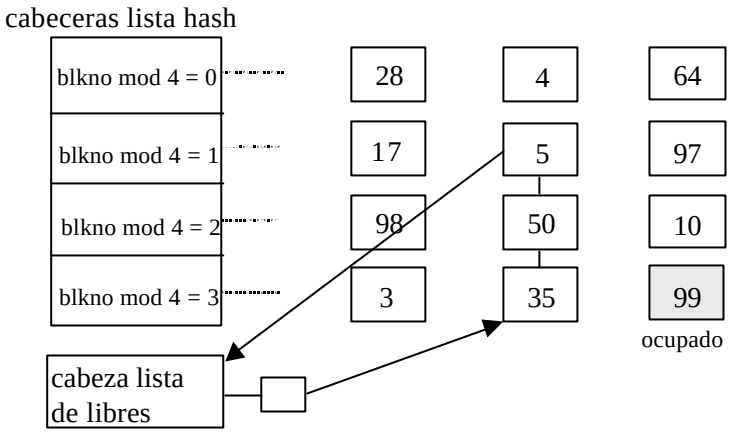


Figura 4.22. Quinta situación en la asignación de un buffer. Buffer 99 en lista hash, pero está ocupado.

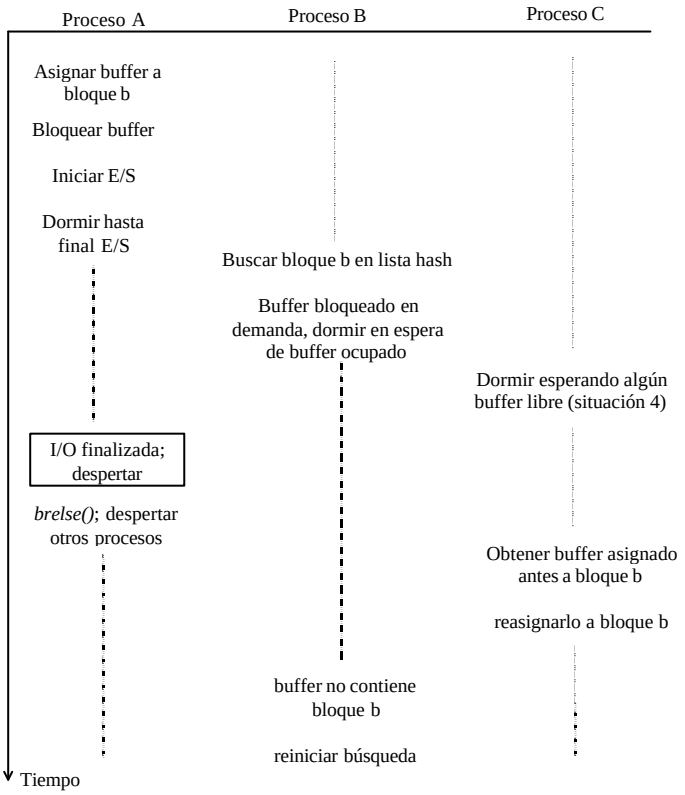


Figura 4.23. Contienda (situación de concurso) por un buffer ocupado.

4.7.6.4.6. Comentarios.

- El algoritmo de asignación de buffers (*getblk*) tiene que ser seguro, con los siguientes objetivos principales:
  - Los procesos no deben dormir indefinidamente.
  - En algún momento habrán de obtener un buffer.
- Ejemplo de posposición indefinida: Situación 4.
- Varios procesos duermen esperando que se libere un buffer (Situación 5).
- El *kernel* despierta a todos los procesos que esperen por un buffer en un determinado evento.
- El *kernel* no garantiza que obtendrá el buffer en el orden en que lo solicitaron.

4.7.7. Funciones para la Realización de Entradas/Salidas.

- En este grupo vamos a tener a las funciones de servicio que permiten a las otras partes del *kernel* realizar entradas/salidas, utilizando el *buffer caché*. Estas funciones afectan tanto a la creación de buffers (memorias intermedias) en páginas de memoria, las entradas/salidas sobre esas, las funciones de desbloqueo y asignación/desasignación de buffers llamadas tras el fin de una operación de entrada/salida. Las funciones principales para la realización de E/S en Linux son:

  - *create_buffers*: Esta función crea nuevos buffers (memorias intermedias) en una página de memoria, descompone la página en buffers, asigna descriptores de buffers y coloca su dirección en la lista de libres (*free_list*).
  - *free_async_buffers*: Esta función se llama para liberar de forma asíncrona los buffers (memorias intermedias) asociados a una página de memoria.
  - *brw_page*: Esta función se llama para efectuar una lectura o una escritura de datos en o desde una página de memoria. Asigna primero descriptores de buffers correspondientes al contenido de la página llamando a *create_buffers* y efectúa luego un bucle para tratar los buffers (memorias intermedias) creados. Dentro de este bucle, cada descriptor de buffer se inicializa y se activa el indicador mostrando que se trata de un buffer temporal que deberá liberarse al final de una Entrada/Salida. Si el buffer encontrado por la función *get_hash_table* se utiliza:
    + En el caso de una lectura de datos, su contenido se transfiere a la página a leer.
    + En el caso de una escritura, la parte correspondiente al contenido de la página se transfiere al buffer, y éste se marca como modificado.

    Tras dicho bucle se llama a la función que lanza la Entrada/Salida (*rw_block*), pasando como parámetros los descriptores de los buffers temporales creados para este propósito.
    Es importante destacar la funcionalidad de *rw_block*. Esta función la llama el *buffer caché*, los sistemas de archivos y el módulo de entrada/salida de bloques, a fin de efectuar una Entrada/Salida. Para ello, esta función verifica la validez de sus argumentos y crea una petición de entrada/salida llamando a *make_request*. El buffer correspondiente se bloquea mientras la entrada/salida no se haya llevado a cabo y se desbloqueará a final de la entrada/salida. De este modo, el proceso que llama puede ponerse en espera del fin de la entrada/salida llamando a la función *wait_on_buffer*.
  - *mark_buffer_uptodate*: Esta función se llama al final de una lectura de datos, cuando se ha efectuado la Entrada/Salida. Indica que el contenido del buffer correspondiente esta al día (actualizado), activando el indicador correspondiente en el campo *b_state*, y luego explora los buffers cuyo contenido se sitúa en la misma página de memoria, y si todos ellos están actualizados, la propia página se marca como actualizada.
  - *unlock_buffer*: Esta función se llama al final de una Entrada/Salida para desbloquear un buffer (memoria intermedia). Desactiva el indicador apropiado en *b_state*, y luego despierta a los procesos en espera sobre este buffer llamando a la función *wake_up*.
  - *generic_readpage*: Esta función se llama para leer el contenido de una página de memoria desde un archivo. Se efectúa un bucle para obtener las direcciones de los bloques que componen el contenido de la página, llamando a la función *bmap* asociada al inodo del archivo. Una vez calculada esta lista de números, la lectura se lanza llamando a la función *brw_page*.

4.7.8. Funciones para la Modificación del Tamaño del *Buffer Caché*.

- Bajo Linux, contrariamente a otros sistemas operativos, el tamaño del *buffer caché* no se fija en la inicialización del sistema. El *buffer caché* está inicialmente vacío y cambia de tamaño en el transcurso de la ejecución del sistema. Cuando el *kernel* necesita buffers (memorias intermedias) adicionales, se asignan nuevas páginas de memoria al *buffer caché* con el objetivo de ampliarlo. Cuando el sistema le falta memoria, puede liberar ciertos buffers así como las páginas de memoria que los contienen, lo cual reduce el tamaño del *buffer caché*. Las funciones más importantes que modifican el tamaño del *buffer caché* son las siguientes:

  - *grow_buffers*: Esta función se llama para ampliar el *buffer caché*. Asigna una página de memoria, crea los buffers según el tamaño de la página de memoria e inserta los nuevos buffers en la lista de buffers libre.
  - *try_to_free_buffer*: Esta función se llama para reducir la memoria asignada al *buffer caché*. Explora todos los buffers contenidos en una página de memoria comprobando si se utilizan. Si no se utilizan estos buffers, son liberados, no formando parte del buffer caché.
  - *shrink_specific_buffers*: Esta función reduce la memoria asignada a los buffers asociados a un tamaño de bloque especificado.
  - *refill_freelist*: Esta función se llama para llenar la lista de buffers libres.

4.7.9. Funciones para la Gestión de Dispositivos.

- En Linux, dos son las funciones que permiten actuar sobre los buffers asociados a un dispositivo especificado.

  - La función *invalidate_buffers* se llama para invalidar los buffers (memorias intermedias) correspondientes a un sistema de archivos especificado. Esta función explora las listas LRU, y actualiza el estado de los buffers correspondientes modificando el campo *b_state*, para indicar que los buffers no son válidos.
  - La función *set_blocksize* permite especificar el tamaño de los bloques lógicos presentes en un dispositivo. Memoriza este tamaño en la entrada correspondiente del *superbloque* asociada al dispositivo, y luego explora las listas de buffers. Todo buffer correspondiente al dispositivo cuyo tamaño no es el especificado se suprime de la lista hash y se inserta en la lista de buffers libres.

4.7.10. Funciones de Acceso a los Buffers. Lectura y Escritura de Bloques de Disco.

- En Linux, el módulo de gestión del *buffer caché* ofrece funciones de servicio utilizables por el sistema de archivo para acceder a los buffers (memorias intermedias).

  - *get_hash_table*: Esta función se llama para obtener un buffer existente correspondiente a un sistema de archivos y a un número de bloque. Llama a la función *find_buffer* para buscar el buffer. Si se encuentra el buffer correspondiente, su número de referencias (campo *b_count*) se incrementa y se devuelve su dirección; si no la función *get_hash_table* devuelve un valor NULL.
  - *getblk*: Esta función se llama para asignar un buffer (memoria intermedia) correspondiente a un dispositivo y a un número de bloque. Primero llama a la función *get_hash_table* para buscar el buffer, y devuelve el resultado si la búsqueda es positiva. En caso contrario, es decir, si no existe ningún buffer correspondiente a la petición, se llama a la función *refill_freelist*, y se lanza una nueva búsqueda llamando a *find_buffer*. Si esta nueva búsqueda es positiva, el tratamiento se reinicia. Si no, se asigna un descriptor de buffer de la lista de buffers libres, se inicializa y se inserta en las listas por *insert_into_hash_queue*, devolviendo su dirección. Como hemos visto en el apartado 6.4, se pueden plantear diferentes situaciones (5) para asignar un buffer a un bloque de disco.
  - *brelse*: Esta función se llama para liberar un buffer. Este buffer se cambia eventualmente de lista LRU llamando a la función *refile_buffer* y su número de referencias (campo *b_count*) se decrementa.

- *bforget*: Esta función se llama para liberar un buffer de la misma forma que *brelse*, pero lo inserta en la lista de buffers libres (*free_list*).
- *bread*: Esta función se llama para leer un bloque correspondiente a un dispositivo (nr_device) y a un número de bloque (nr_block). Primero llama a *getblk* para obtener el descriptor del buffer y eventualmente llama a *rw_block* con el objetivo de leer la página desde el disco. Devuelve la dirección del descriptor, o un valor nulo en caso de error.
- *breada*: Esta función es similar a *bread*, pero permite activar la lectura asíncrona (anticipada) de otros bloques. Llama a *getblk* para obtener el descriptor del buffer especificado, y luego eventualmente a *rw_block* a fin de leer su contenido desde el disco. Para cada bloque especificado, llama a *getblk* para obtener un descriptor de buffer, y lanza la lectura de todos los bloques llamando a *rw_block*. No espera a la finalización de esta lectura adicional, y devuelve la dirección del descriptor del buffer correspondiente al primer bloque.

4.7.10.1. Lectura de un Bloque de Disco.

- Un proceso busca el buffer requerido en el *buffer caché*.
- Si el buffer está en el *buffer caché*, el *kernel* lo devolverá inmediatamente sin necesidad de realizar una lectura física del disco.
- Si no está el buffer en el *buffer caché*, se realizan las siguientes acciones:
  - El *kernel* realiza una petición de lectura al controlador de disco.
  - El proceso duerme ⇒ Evento: operación de E/S.
  - El *kernel* informa al controlador del disco, que quiere leer datos.
  - El controlador de disco transmite los datos al buffer.
  - Finalmente, el controlador de disco interrumpe al procesador cuando la E/S se ha completado, para que despierte a los procesos dormidos.
  - El gestor de interrupciones de disco despierta al proceso que dormía: el contenido del bloque de disco ya está en el buffer.

```
algoritmo bread          // lectura de un bloque de disco
entrada:                 número de dispositivo (nr_device) y número de bloque (nr_block)
salida:                  buffer con los datos del bloque
{
    obtener un buffer para el bloque (algoritmo getblk);
    if (datos del buffer son válidos)
        return buffer;
    iniciar lectura de disco;                  // función rw_block
    sleep(evento: Espera_lectura_finalizada);  // wait_on_buffer (espera lectura finalizada)
    return buffer;
}
```

Algoritmo para la lectura de un bloque de disco.

- Explicación del algoritmo:
  - Entrada: nr_block y nr_device
  - Salidos: dirección del buffer con los datos del bloque solicitado
  - Asignar espacio para traer bloque ⇒ *getblk*.
  - Si el bloque ya está en el buffer caché y sus datos son válidos, ⇒ el algoritmo finaliza devolviendo la dirección de este buffer.
  - Si el bloque no está en el buffer caché ⇒ lectura del disco, y mientras ésta se realiza el proceso duerme a la espera de que la lectura esté finalizada.

4.7.10.2. Lectura Asíncrona (anticipada) de un Bloque de Disco.

- Anticipación de lectura de segundo bloque de disco ⇒ muy útil en lecturas secuenciales de archivos.
- Como ejemplo podemos ver que, cuando un proceso lee un archivo secuencialmente:
  - Solicitar la segunda operación de E/S asíncronamente.
  - Los datos estarán muy probablemente en memoria cuando se necesiten.
- Mejora el rendimiento del sistema respecto a operaciones de entrada/salida.
- El *kernel* comprueba si el primer bloque está en el *buffer caché*.
- Si no está ⇒ Solicita al controlador de disco que lo lea de forma inmediata.
- Si el segundo bloque no está en el *buffer caché* ⇒ Solicita al controlador de disco que lo lea de forma asíncrona.
- El proceso duerme ⇒ Evento: Espera fin lectura del primer bloque.
- Cuando el proceso se despierta ⇒ Devuelve el buffer para el primer bloque, pero no se preocupa por la finalización de la segunda operación de lectura.
- Al finalizar la segunda operación de lectura:
  - El controlador de disco interrumpe al procesador cuando la E/S se ha completado.
  - El manejador interrupciones nota que la E/S era asíncrona y devuelve el buffer (en cuyo contenido está el segundo bloque del disco).

```
algoritmo breada        // lectura anticipada de un bloque de disco
entrada:                (1) número de dispositivo y número de bloque para lectura inmediata.
                        (2) número de dispositivo y número de bloque para lectura asíncrona.
salida:                 buffer que contiene los datos de la lectura inmediata.
{
    if (primer bloque no está en el buffer caché)
    {
        obtener buffer para primer bloque (algoritmo getblk);
        if (datos buffer no válidos)
            iniciar lectura en disco;                    // función rw_block
    }
    if (segundo bloque no está en el buffer caché)
    {
        obtener buffer para segundo bloque (algoritmo getblk);
        if (datos buffer no válidos)
            liberar buffer (algoritmo brelse);
        else
            iniciar lectura en disco;                    // función rw_block
    }
    if (primer bloque estaba inicialmente en el buffer caché)
    {
        leer primer bloque (algoritmo bread);
        return buffer;
    }
    sleep(evento: primer buffer contiene datos válidos);  // wait_on_buffer
    return buffer;
}
```

Algoritmo para lectura asíncrona (anticipada) de un bloque.

- Explicación del algoritmo:
  - Entrada: nr_block y nr_device lectura inmediata
    nr_block y nr_device lectura asíncrona
  - Salidos: dirección del buffer con los datos del bloque de la lectura inmediata
  - El *kernel* busca si el primer bloque está en el buffer caché:
  - Si el bloque no está en el buffer caché ⇒ mandará leer el primer bloque y el segundo (de forma asíncrona). Entonces el proceso duerme en espera de que termine la operación de lectura del primer bloque. Al despertar el proceso por parte del *kernel*, el buffer ya tiene

datos válidos. Si el proceso desea más tarde el segundo bloque, éste estará en el buffer caché, ya que fue leído de forma asíncrona mientras se completaba la lectura del primer bloque.
- Si el primer bloque ya está en el buffer caché ⇒ el *kernel* devuelve inmediatamente la dirección del buffer asociado.

4.7.10.3. Escritura de un Bloque de Disco.

- El *kernel* informa al controlador del dispositivo (disco) que quiere escribir el contenido de un buffer.
- El controlador del dispositivo (disco) planifica el bloque para una operación de E/S.
- Si la escritura es síncrona:
  - El proceso va a dormir esperando su finalización.
  - Cuando despierta ⇒ libera el buffer.
- Si la escritura es asíncrona:
  - El *kernel* inicia la escritura, pero no espera su conclusión.
  - Liberará el buffer cuando la escritura se complete (escritura retardada).

```
algoritmo bwrite    // escritura de un bloque
entrada:            un buffer
salida:             void
{
        iniciar escritura en disco;                    // función rw_block
        if (E/S es síncrona)
        {
                sleep(evento: final E/S);
                liberar buffer (algoritmo brelse);
        }
        else if (buffer marcado de escritura retardada)
                marcar el buffer para colocarlo posteriormente al principio de la lista de buffers libres;
}
```

Algoritmo para la escritura de un bloque de disco.

4.7.10.3.1. Escritura Retardada.

- Es posible que el *kernel* no escriba inmediatamente los datos en el disco. Retrasa la escritura tanto como sea posible.
- Escritura retardada ⇒ Marca el buffer y continua sin planificar ninguna operación de E/S.
- El *kernel* escribirá y liberará el bloque antes de reasignar el buffer para otro bloque (situación 3 de *getblk*).

4.7.11. Reescritura de Buffers Modificados.

- La función *sync_buffers* reescribe los buffers modificados en disco. Esta función puede realizar hasta tres pasadas de exploración en las listas:
  - Durante la primera pasada, los buffers modificados se reescriben de manera asíncrona en disco llamando a la función *rw_block*, no teniéndose en cuenta los buffers bloqueados.
  - Durante las siguientes pasadas, *sync_buffers* se pone en espera sobre los buffers bloqueados llamando a *wait_on_buffer*.
- El número de pasadas que puede realizar la función *sync_buffers* depende del valor del parámetro *wait*. La primitiva *sync* inicializa el parámetro *wait* a 0 para que *sync_buffers* lance la entrada/salida sin esperar su fin (de forma asíncrona), mientras que la llamada al sistema *fsync* lo actualiza a 1 para que la función *sync_buffers* efectué varias pasadas, con el objetivo de que los buffers modificados se reescriban en disco al final de la función.

- El proceso *bdflush*, es un proceso que se crea al inicializar el sistema y su función es la reescritura de buffers modificados. Las funciones que permiten gestionar el funcionamiento del proceso *bdflush* son:

- *wake_up_bdflush*: Esta función utiliza a su vez la función *wake_up* para despertar al proceso *bdflush*, y espera que termine su tratamiento utilizando *sleep_on* sobre la cola de espera *bdflush_done*.
- *sync_old_buffers*: Esta función reescribe el contenido de los buffers en disco. Primero llama a las funciones *sync_supers* y *sync_inodes* para reescribir los descriptores del sistema de archivos y inodos en disco respectivamente. Luego explora la lista de buffers modificados. Para cada buffer, la función *refile_buffer* se llama si su contenido no se ha modificado, con el objetivo de desplazar el buffer a la lista correspondiente. Si el contenido del buffer se ha modificado, si no está ocupado (bloqueado) y su fecha de escritura se ha alcanzado, se llama a la función *rw_block* para reescribir el contenido de la memoria en disco. Para finalizar con esta función, se actualizan las estadísticas de utilización del buffer.
- *sys_bdflush*: Esta función implementa la llamada al sistema *bdflush*. Permite que el proceso que llama obtenga o modifique uno de los parámetros almacenados en la variable *bdf_prm* (9 parámetros) que controlan la ejecución del proceso *bdflush*. Estos parámetros toman valores referentes a tiempos, estadísticas y datos de utilización de los buffers modificados.
- *bdflush*: Esta función implementa el proceso *bdflush*. Tras la inicialización del descriptor del proceso *bdflush*, éste realiza un bucle infinito. En cada pasada de este bucle realiza un tratamiento similar al de la función *sync_old_buffers*, es decir, almacena en disco el contenido de los buffers modificados cuya fecha de reescritura se ha alcanzado, y luego suspende al proceso *bdflush*, llamando a *interruptible_sleep_on*, si el número de buffers modificados es inferior al porcentaje máximo especificado por el parámetro *nfract* de la variable *bdf_prm* que gestiona el proceso *bdflush*. El proceso *bdflush* despertará debido a la activación de la función *refile_buffer*, que llamará a *wake_up_bdflush*, cuando el proceso *bdflush* deba ejecutarse de nuevo.

4.7.12. Gestión de *clusters*.

- En Linux, se utiliza la noción de *cluster* para las entradas/salidas con el objetivo de indicar un grupo de buffers cuyos contenidos son contiguos en memoria y que corresponden a bloques contiguos en disco. La ventaja de los *clusters* reside en el hecho de que la lectura o escritura de los buffers puede efectuarse en una sola entrada/salida, mientras que hay que efectuar varias entradas/salidas (una por bloque) si los buffers correspondientes a bloques contiguos en disco están dispersos en memoria. Este mecanismo de *clustering* es utilizado en varios sistemas de archivos y para las entradas/salidas directas sobre dispositivos accesibles en modo bloque. Las funciones disponibles más importantes para gestionar los *clusters* son:

  - *try_to_reassign*: Esta función comprueba si todos los buffers contenidos en una página de memoria están disponibles y si es así, los asocia a un nuevo *cluster*. Explora los buffers contenidos en la página y verifica que cada buffer está disponible, se suprimen de las listas hash llamando a la función *remove_from_hash_queue*, sus direcciones en disco (número de dispositivo y número de bloque) se modifican y se registran de nuevo en las listas hash llamando a *insert_into_hash_queue*.
  - *reassign_cluster*: Esta función se llama para obtener un nuevo *cluster*. Primero llama a *refill_free_list* con el objetivo de obtener suficientes buffers disponibles, luego explora la lista de buffers libres. Para cada buffer de la lista de libres llama a *try_to_reassign* para intentar crear un nuevo *cluster* en la página que contiene a ese buffer.

- *try_to_generate_cluster*: Esta función intenta generar un *cluster* en una nueva página de memoria. Asigna una página de memoria, luego llama a *create_buffers* con el objetivo de crear buffers libres que apunten al contenido de la página de memoria. A continuación utiliza la función *find_buffer* para comprobar o verificar si los buffers corresponden al dispositivo y a los bloques especificados son contiguos en el disco y no existe ningún buffer referido ya a estos bloques. Si no es así, es imposible crear un *cluster*, libera los buffers creados y la página de memoria reservada, y devuelve 0. En el caso contrario, se puede crear un *cluster*, lo buffers creados se inicializan y se registran en la lista de buffers libres llamando a *insert_into_free_list*.
- *generate_cluster*: Esta función se llama para crear un *cluster* por una serie de bloques especificados para un dispositivo concreto. Verifica primero que los bloques especificados sean contiguos en el disco y que no exista ningún buffer referido ya a estos bloques. A continuación se llama a la función *try_to_generate_cluster* para asignar un *cluster*. Si la función *try_to_generate_cluster* fracasa, se comprueba la memoria disponible y si es limitada se llama a la función *reassign_cluster* para crear el *cluster* a partir de buffers existentes, si no, se crea un nuevo *cluster* llamando a *try_to_generate_cluster* de nuevo.

4.7.13. Ventajas y Desventajas del *Buffer Caché*.

Ventajas:

- Permite un acceso uniforme al disco independientemente del tipo de bloque.
- No hay restricciones de alineamiento de datos para los procesos de usuario que realizan operaciones de E/S, ya que el *kernel* realiza las alineaciones internamente.
- Reduce el tráfico de información al disco $\Rightarrow$ Aumenta el rendimiento del sistema y disminuye el tiempo de respuesta. Si varios procesos acceden al mismo bloque, el *kernel* mantendrá la última actualización de los datos de forma coherente.
- Ayuda a mantener la integridad del sistema, ya que mantienen una única imagen de los bloques de disco almacenados en el *buffer caché*.

Desventajas:

- El *kernel* no escribe inmediatamente los buffers en el disco marcados como de escritura retardada o modificados, el sistema es vulnerable a caídas, que dejarían los datos del disco en un estado incorrecto.
- El uso del *buffer caché* exige una copia de datos extra en las operaciones de lectura y escritura por parte de los procesos de usuario.

4.8. EL SEGUNDO SISTEMA DE ARCHIVOS EXTENDIDO (EXT2)

El Segundo Sistema de Archivos Extendido (The Second Extended File System, EXT2) fue pensado (por Remy Card) como un sistema de archivos extensible y poderoso para Linux. También es el sistema de archivos que más éxito tiene en la comunidad Linux y es básico para todas las distribuciones actuales de Linux. El sistema de archivos EXT2, como muchos sistemas de archivos, se construye con la premisa de que los datos contenidos en los archivos se guarden en bloques de datos. Estos bloques de datos son todos de la misma longitud y, si bien esa longitud puede variar entre diferentes sistemas de archivos EXT2, el tamaño de los bloques de un sistema de archivos EXT2 en particular se decide cuando se crea (usando mke2fs). El tamaño de cada archivo se redondea hasta un numero entero de bloques. Si el tamaño de bloque es 1024 bytes, entonces un archivo de 1025 bytes ocupará dos bloques de 1024 bytes. Desafortunadamente, esto significa que en promedio se desperdicia un bloque por archivo. No todos los bloques del sistema de archivos contienen datos, algunos deben usarse para mantener la información que describe la estructura del sistema de archivos. EXT2 define la topología del sistema de archivos describiendo cada archivo del sistema con una estructura de datos inodo. Un inodo describe que bloques ocupan los datos de un archivo y también los permisos de acceso del archivo, las horas de modificación del archivo y el tipo del archivo. Cada archivo en el sistema de archivos EXT2 se describe por un único inodo y cada inodo tiene un único número que lo identifica. Los inodos del sistema de archivos se almacenan juntos en tablas de inodos. Los directorios EXT2

son simplemente archivos especiales (ellos mismos descritos por inodos) que contienen punteros a los inodos de sus entradas de directorio.

El sistema de archivos EXT2 ofrece funcionalidades estándar. Soporta los archivos UNIX (archivos regulares, directorios, archivos especiales, enlaces simbólicos) y ofrece características avanzadas:

- Pueden asociarse atributos a los archivos para modificar comportamiento del *kernel*, los atributos reconocidos son los siguientes:
- Supresión segura. Cuando el archivo se suprime, su contenido se destruye previamente con los datos aleatorios.
- Undelete. Cuando el archivo se suprime, se guarda automáticamente a fin de poder restaurarlo posteriormente.
- Compresión automáticas. La lectura y la escritura de los datos en el archivo da lugar a una compresión al vuelo.
- Escrituras síncronas. Toda la información sobre el archivo se escribe de manera síncrona en disco.
- Inmutable. El archivo no puede modificarse ni suprimirse.
- Adición exclusiva. El archivo sólo puede modificarse si se ha abierto en modo adición, y no puede suprimirse.
- Compatibilidad con la semántica de UNIX System V o BSD. Una opción de montaje permite elegir el grupo asociado a los nuevos archivos. La semántica BSD especifica que el grupo se hereda desde el directorio padre, mientras que SVR4 utiliza el número de grupo primario del proceso que llama.
- Enlaces simbólicos rápidos. Ciertos enlaces simbólicos no utilizan bloque de datos: el nombre del archivo destino está contenido directamente en el inodo en disco, lo que permite economizar espacio en disco y acelerar la resolución de estos enlaces evitando una lectura de bloque.
- El estado de cada sistema de archivos se memoriza. Cuando el sistema de archivos se monta como inválido hasta que se desmonte. El verificador de la escritura, e2fsch, utiliza este estado para acelerar las verificaciones cuando no son necesarias.
- Un contador de montaje y una demora máxima entre dos verificadores pueden utilizarse para forzar la ejecución de e2fsck.
- El comportamiento del código de gestión puede adaptarse en caso de error. Puede mostrar un mensaje de error, "remontar" el sistema de archivos en lectura exclusiva a fin de evitar una corrupción de los datos, o provocar un error del sistema.
- Además, EXT2 incluye numerosas optimizaciones. En las lecturas de datos, se efectúan lecturas anticipadas. Esto significa que el código de gestión pide la lectura no sólo del bloque que necesita, sino también de otros bloques consecutivos. Esto permite cargar en memoria bloques que se usarían en la siguiente entradas/salidas. Este mecanismo se utiliza también en las lecturas de entradas de directorio, ya sean explícitas (por la llamada al sistema readdir) o implícitas (en la resolución de nombres de archivos en la operación sobre el inodo lookup).
- Las asignaciones de bloques e inodos también se han optimizado. Se usan grupos de bloques para agrupar bloques inodos emparentados así como sus bloques de datos. Un mecanismo de preasignación permite también asignar bloques consecutivos a los archivos: cuando debe asignarse un bloque, se reservan hasta 8 bloques consecutivos. De este modo, las asignaciones de bloques siguientes ya se han satisfecho y el contenido de archivos tiende a escribir bloques contiguos, lo que acelera su lectura, especialmente gracias a las técnicas de lectura anticipada.

La figura 4.24 muestra la disposición del sistema de archivos EXT2 ocupando una serie de bloques en un dispositivo modo bloque. Por la parte que le corresponde a cada sistema de archivos, los dispositivos de bloque son sólo una serie de bloques que se pueden leer y escribir. Un sistema de archivos no se debe preocupar donde se debe poner un bloque en el medio físico, eso es trabajo del controlador del dispositivo. Siempre que un sistema de archivos necesita leer información o datos del dispositivo de bloque que los contiene, pide que su controlador de dispositivo lea un número entero de bloques. El sistema de archivos EXT2 divide las particiones lógicas que ocupa, en Grupos de Bloques (Block Groups) o conjuntos de bloques. Cada grupo duplica información crítica para la integridad del sistema de archivos ya sea valiéndose de archivos y directorios como de bloques de información y datos. Esta duplicación es necesaria por si ocurriera un desastre y el sistema de archivos necesitara recuperarse. Los siguientes subapartados sobre el sistema de archivos EXT2 describen con más detalle los contenidos de cada Grupo de Bloques.

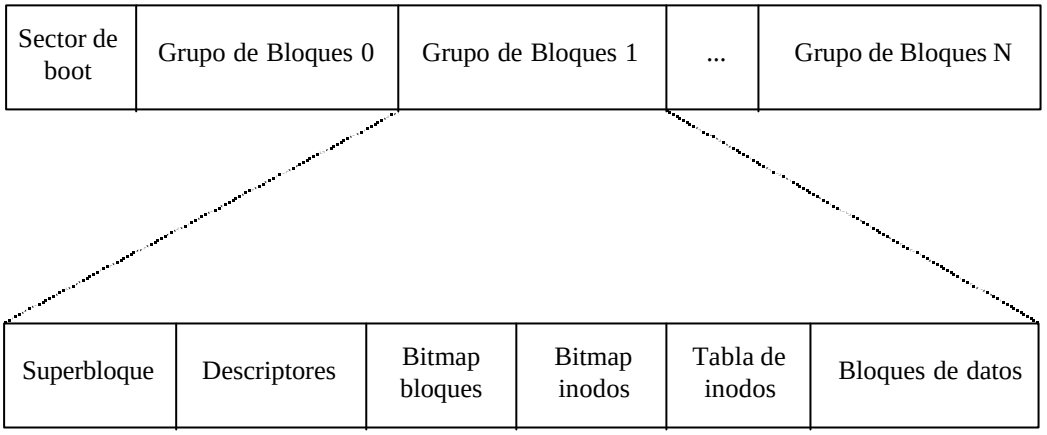


Figura 4.24. Estructura física del sistema de archivos EXT2

El sector de boot (sector de arranque) contiene el código máquina necesario para cargar el *kernel* en el arranque del sistema, y cada uno de los Grupos de Bloques se descompone a su vez en varios elementos como se puede observar en la parte inferior de la figura 4.24.

- Una copia del Superbloque. Esta estructura contiene la información de control del sistema de archivos y se duplica en cada Grupo de Bloques para permitir paliar fácilmente una corrupción del sistema de archivos.
- Una tabla de descriptores. Estos descriptores contienen las direcciones de bloques que a si vez contienen información crucial como los bloques de bitmap y la tabla de inodos; también se duplican en cada Grupo de Bloques.
- Un bloque de bitmap para los bloques. Este bloque contiene una tabla de bits (mapa de bits), donde a cada bloque del Grupo le asocia un bit indicando si el bloque está asignado (el bit está a 1) o disponible (el bit está a 0).
- Un bloque de bitmap para los inodos. Este bloque contiene una tabla de bits (mapa de bits), donde a cada bloque del grupo le asocia un bit indicando si el inodo está asignado (el bit está a 1) o disponible (el bit está a 0).
- Una tabla de inodos. Estos bloques contienen una parte de la tabla de inodos del sistema de archivos.
- Bloques de datos. El resto de los bloques del Grupo se utiliza para almacenar los datos contenidos en los archivos y directorios.

Un sistema de archivos se organiza en archivos y directorios. Un directorio es un archivo de tipo particular, que contiene entradas. Cada una de las entradas de directorio contiene varios campos: (1) el número de inodo correspondiente al archivo; (2) el tamaño de la entrada en bytes; (3) el número de caracteres que componen el nombre del archivo; y (4) el nombre del archivo. Por ejemplo, un directorio que contiene las entradas ".", "..", "arch1", "nombre_archivo_largo" y "f1" tendría la siguiente forma:

```
i1    12    01    .
i2    12    02    ..
i3    16    05    arch1
i4    28    19    nombre_archivo_largo
i5    12    02    f1
```

4.8.1. El Superbloque EXT2

El Superbloque contiene una descripción del tamaño y forma la base del sistema de archivos (información de control). La información contenida permite al administrador del sistema de archivos usar y mantener el sistema de archivos. Normalmente sólo se lee el Superbloque del Grupo de Bloques 0 (se sitúa al principio del sistema de archivos) cuando se monta el sistema de archivos pero cada Grupo de Bloques contiene una copia duplicada en caso de que se corrompa sistema de archivos. Entre otra información *ext2_super_block* contiene los siguientes campos <include/linux/ext2_fs_sb.h>:

s_inode_count. Número total de inodos.
s_blocks_count. Número total de bloques.

s_r_blocks_count. Número de bloques reservados al superusuario.
s_free_block_count. El número de bloques libres en el sistema de archivos
s_free_inode_count. El número de inodos libres en el sistema de archivos.
s_first_data_block. Este es el número del primer bloque de datos en el sistema de archivos. El primer inodo en un sistema de archivos EXT2 raíz seria la entrada de directorio para el directorio "/".
s_log_block_size. El tamaño lógico de los bloques para este sistema de archivos en bytes, por ejemplo 1024 bytes.
s_blocks_per_group. El número de bloques en un grupo. Como el tamaño de bloque, éste se fija cuando se crea el sistema de archivos.
s_frags_per_group. Número de fragmentos por grupo.
s_inodes_per_group. Número de inodos por grupo.
s_mtime *y* ***s_wtime***. Fecha del último montaje del sistema de archivos y de la última escritura del superbloque, respectivamente.
s_mnt_count. Número de montajes del sistema de archivos.
s_max_mnt_count. Número máximo de montajes del sistema de archivos. Estos dos últimos campos juntos (*s_mnt_count* y *s_max_mnt_count*) permiten al sistema determinar si el sistema de archivos fue comprobado correctamente. El contador de montaje se incrementa cada vez que se monta el sistema de archivos y cuando es igual al contador máximo de montaje muestra el mensaje de aviso "maximal mount count reached, running e2fsck is recommended".
s_magic. Esto permite al software de montaje comprobar que es realmente el superbloque para un sistema de archivos EXT2 (firma del sistema de archivos). Para la versión actual de EXT2 éste es 0xEF53.
s_state. Estado del sistema de archivos.
s_errors. Comportamiento del sistema de archivos en caso de errores.
s_minor_rev_level. Los niveles de revisión mayor y menor permiten al código de montaje determinar si este sistema de archivos soporta o no características que sólo son disponibles para revisiones particulares del sistema de archivos (número de revisión)). También hay campos de compatibilidad que ayudan al código de montaje determinar que nuevas características se pueden usar con seguridad en ese sistema de archivos.
s_lastcheck. Fecha de la última verificación del sistema de archivos.
s_chechinterval. Tiempo máximo entre dos verificaciones.
s_creator_os. Identificador del sistema de archivos bajo el cual se ha creado el sistema de archivos.
s_def_resuid. Identificador del usuario que puede usar lo bloques reservados al superusuario de modo predeterminado.
s_def_resuid. Identificador del grupo que puede usar lo bloques reservados al superusuario de modo predeterminado.

4.8.2. Los Descriptores de Grupos de Bloques EXT2

Cada Grupo de Bloques tiene una estructura de datos que lo describe (contiene 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 Bloque se duplican en cada Grupo de Bloques en caso de corrupción del sistema de archivos. Cada descriptor de grupo contiene la siguiente información en la estructura *ext2_group_desc* declarada en <include/linux/ext2_fs.h>:

bg_blocks_bitmap. El número de bloque del mapa de bits de bloques reservados para este Grupo de Bloques (dirección del bloque de bitmap para los bloques de este grupo). Se usa durante la reserva y liberación de bloques.
bg_inode_bitmap. El número de bloque del mapa de bits de inodos reservados para este Grupo de Bloques (dirección del bloque de bitmap para los inodos de este grupo). Se usa durante la reserva y liberación de inodos.
bg_inode_table. El número de bloque del bloque inicial para la tabla de inodos de este Grupo de Bloques (dirección del primer bloque de la tabla de inodos en este grupo). Cada inodo se representa por la estructura de datos inodo EXT2 descrita abajo.
bg_free_blocks_count. Número de bloques libres en este grupo.
bg_free_inodes_count. Número de inodos libres en este grupo.
bg_used_dirs_count. Número de directorios asignados en este grupo
bg_pad. No utilizado.
bg_reserved. Campo reservado para futuras extensiones.

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

4.8.3. El Inodo EXT2

La tabla de inodos se descompone en varias partes, y cada parte está contenido en un Grupo de Bloques. Esto permite utilizar estrategias de asignación, como por ejemplo, cuando un bloque debe asignarse, el *kernel* intenta asignarlo en el mismo grupo que su inodo con el objetivo de minimizar el desplazamiento de las cabezas de lectura/escritura en la lectura de un archivo.

En el sistema de archivos EXT2, el inodo es el bloque de construcción básico; cada archivo y directorio del sistema de archivos es descrito por un y sólo un inodo. Los inodos EXT2 para cada Grupo de Bloques se almacenan juntos en la tabla de inodos con un mapa de bits que permite al sistema seguir la pista de inodos reservados y libres. La Figura 4.25 muestra el formato de un inodo EXT2 *(ext2_inode)*, entre otra información, contiene los siguientes campos <include/linux/ext2_fs_i.h>:

i_mode. Esto mantiene dos tipos de información (modo del inodo); qué inodo describe y los permisos que tienen los usuarios. Para EXT2, un inodo puede describir archivos, directorio, enlace simbólico, dispositivo de bloque, dispositivo de carácter o FIFO.
i_uid e ***i_gid***. Los identificadores de usuario y grupo de los propietarios de este archivo o directorio. Esto permite al sistema de archivos aplicar correctamente el tipo de acceso.
i_size. El tamaño del archivo en bytes.
i_atime, i_ctime, i_mtime e ***i_dtime***. Indican la fecha del último acceso, última modificación del inodo, última modificación del contenido del archivo y la fecha de supresión del archivo, respectivamente.
i_links_count. Número de enlaces asociados al inodo.
i_blocks. Número de bloques de 512 bytes asignados al inodo.
i_flags. Atributos asociados al archivo.
i_block. Direcciones de bloques de datos asignados al inodo. Los primeros 12 elementos (valor de la constante EXT2_NDIR_BLOCKS) de la tabla contiene direcciones de bloques de datos directos. Los tres siguientes contienen punteros indirectos: (1) Puntero indirecto simple (EXT2_IND_BLOCK) contiene la dirección de un bloque que a su vez contiene la dirección de los bloques de datos; (2) Puntero indirecto doble (EXT2_DIND_BLOCK) contiene la dirección de un bloque que contiene la dirección de bloques que contienen a su vez la dirección de lo bloques de datos siguientes; y (3) Puntero indirecto triple (EXT2_TIND_BLOCK) contiene la dirección de un bloque que contiene la dirección de bloques que apuntan a su vez a los bloques indirectos que contienen la dirección de los bloques de datos. Por ejemplo, el puntero indirecto doble apunta a un bloque de punteros que apuntan a bloques de punteros que apuntan a bloques de datos.
i_version. Número de versión asociada al inodo.
i_file_acl. Dirección del descriptor de la lista de control de acceso asociada al archivo.
i_dir_acl. Dirección del descriptor de la lista de control de acceso asociada a un directorio.
i_pad1. No utilizado.
i_reserved1 e ***i_reserved2***. Campos reservados para extensiones futuras.

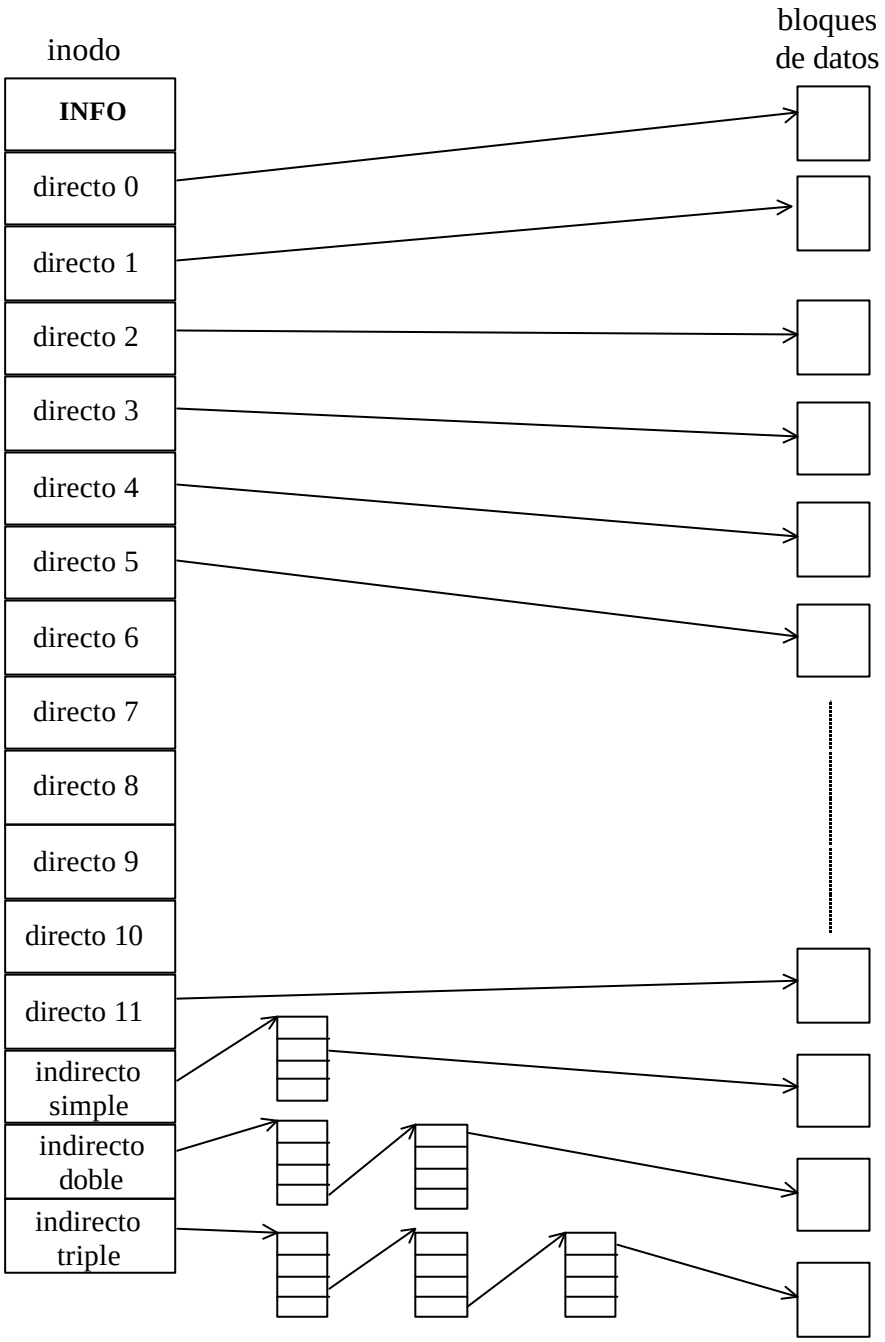


Figura 4.25. Inodo EXT2.

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

4.8.4. Directorios EXT2

En el sistema de archivos EXT2, los directorios son archivos especiales que se usan para crear y mantener rutas de acceso a los archivos en el sistema de archivos. La Figura 4.26 muestra la estructura de una estrada directorio en memoria. Un archivo directorio es *ext2_dir_entry* en <include/linux/ext2_fs.h> una lista de entradas directorio, cada una conteniendo la siguiente información:

inode. El inodo para esta entrada del archivo directorio. Es un índice al vector de inodos guardada en la Tabla de Inodos del Grupo de Bloques. En la Figura 4.26, la entrada directorio para el archivo llamado file tiene una referencia al número de inodo i1.
rec_len. Tamaño en bytes de la entrada de directorio.

name_len. Número de caracteres que componen el nombre del archivo.
name. El nombre de esta entrada directorio.

Las dos primeras entradas para cada directorio son siempre las entradas estándar "."? y ".." significando "directorio actual" y "directorio padre", respectivamente.

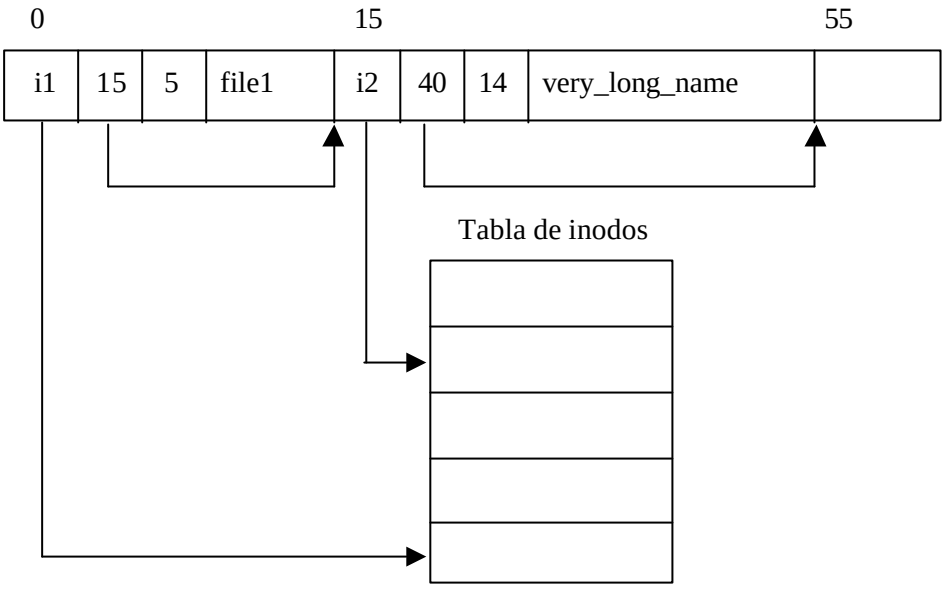


Figura 4.26. Directorio EXT2.

4.8.5. Operaciones Vinculadas con el Sistema de Archivos EXT2

Las operaciones relacionadas con el superbloque de un sistema de archivos EXT2 se implementan en el archivo fuente fs/ext2/super.c. La variable ext2_sops contiene los punteros a las funciones que aseguran estas operaciones. Las funciones ext2_error, ext2_panic y ext2_warning son llamadas por el código de gestión del sistema de archivos cuando se detecte un error y muestran un mensaje de error o de advertencia con la identificación del dispositivo afectado y la función interna que haya detectado el error llamando a la función printk. Se utilizan varias función para el montaje de un sistema de archivos: (1) parse_options que analiza las opciones de montaje especificadas; (2) ext2_setup_super que inicializa el descriptor del superbloque a partir del superbloque del sistema de archivos leído desde el disco; (3) ext2_check_descriptors que verifica la validez de los descriptores de grupos de bloques leídos desde el disco; (4) ext2_commit_super que se llama para guardar las modificaciones efectuadas en el superbloque.

La función ext2_read_super implementa la operación sobre el sistema de archivos read_super y se llama al montar el sistema de archivos. Las acciones que lleva a cabo son las siguientes: (1) analizar las opciones de montaje (parse_options); (2) leer el superbloque desde disco y verificar su validez; (3) inicializar el descriptor del superbloque; (4) asignar una tabla de punteros para los descriptores de grupos, cada descriptor se carga en un buffer (memoria intermedia) llamando a la función bread, y verificando la validez de estos descriptores con ext2_check_descriptors; y (5) leer el inodo de la raíz del sistema de archivos en memoria, llamando a la función iget, y se llama a ext2_setup_super para terminar la inicialización del descriptor del superbloque.

Además de ext2_read_super se implementar otras como ext2_write_super que implementa la operación write_super, actualizando temporizadores e indicando que el sistema de archivos está montado. La función ext2_put_super implementa la operación put_super sobre el sistema de archivos, llamándose cuando el sistema de archivos se desmonta y liberando todas las estructuras de datos que estén en memoria vinculadas con el superbloque. Por último, debemos decir que la función ext2_statfs implementa la operación statfs y copia las estadísticas de uso del sistema de archivos desde el descriptor del superbloque.

Las funciones de asignación y liberación de bloques e inodos se implementas en los archivos fuentes fs/ext2/balloc.c y fs/ext2/ialloc.c. Estos módulos definen funciones internas: get_group_desc (devuelve el descriptor correspondiente) a un grupo de bloques; read_block_bitmap, read_inode_bitmap (cargan en

memoria un bloque de bitmap); load_block_bitmap, load_inode_bitmap (estas funciones verifican que los números de bloques a liberar son válidos, mantienen un caché LRU de bloques cargados en las tablas s_block_bitmap y s_inode_bitmap, y llaman a read_blocks_bitmap y read_inode_bitmap para cargar los bloques en memoria). La liberación de bloques se efectúa por medio de la función ext2_free_blocks. La función ext2_new_block implementa la asignación de bloque y tras esta operación intenta a una preasignación de bloques consecutivos. La liberación de un inodo se efectúa por medio de la función ext2_free_inode y la asignación es tarea de la función ext2_new_inode. Por último, si hay asociadas funciones sobre cuotas al superbloque del sistema de archivos, las operaciones initialize y alloc_inode se llaman para tener en cuenta la asignación.

El archivo fuente fs/ext2/inode.c contiene las funciones de gestión de inodos en disco. La operación sobre inodo bmap implementa varias funciones: (1) inode_bmap (devuelve la dirección de un bloque contenido en el inodo); (2) block_bmap (devuelve la dirección de un bloque obtenida en una tabla contenida en un bloque de datos); (3) ext2_bmap (implementa la operación bmap que obtiene la dirección del bloque de datos especificado, llamando a inode_bmap y block_bmap para efectuar las diversas indirecciones). Hay además varias funciones relacionadas con la asignación de bloques: (1) ext2_discard_prealloc (está función se llama al cerrar un archivo, liberando los bloques previamente asignados); (2) ext2_alloc_block (función para asignar un bloque); (3) inode_getblk (esta función se llama para obtener un buffer que contiene un bloque cuya dirección se almacena en el inodo, utilizando getblk para obtener el buffer); (4) block_getblk (esta función se llama para obtener un buffer que contenga un bloque cuya dirección se almacena en una tabla contenida en un bloque de datos, es muy similar a inode_getblk); (5) ext2_getblk (se llama para obtener un buffer conteniendo un bloque asociado a un inodo, llamando a inode_getblk y block_getblk para obtener las diversas indirecciones); (6) ext2_getcluster (esta función intenta crear un cluster referido a varios bloques de un inodo); (7) ext2_bread (función para leer un bloque de datos asociado a un inodo). Otras funciones como para la gestión de inodos en disco son las siguientes: (1) ext2_read_inode (implementa la operación read_inode sobre el sistema de archivos); (2) ext2_update_inode (rescribe un inodo en disco); (3) ext2_write_inode (implementa la operación write_inode sobre el sistema de archivos, llamando a la función ext2_update_inode especificando que la reescritura del inodo no debe efectuarse inmediatamente, la función ext2_sync_inode efectúa la misma tarea pero exige una reescritura inmediata al llamar a ext2_update_inode); (4) ext2_put_inode (implementa la operación put_inode sobre el sistema de archivos).

El archivo fuente fs/ext2/dir.c contiene las funciones de gestión de directorios como por ejemplo: (1) ext2_chack_dir_entry (esta función se llama para controlar la validez de una estrada de directorio, si detecta un error muestra un mensaje llamando a la función ext2_error); y (2) ext2_readdir (esta función implenta la operación readdir, para ellos utiliza la función ext2_bread para leer cada bloque que compone el directorio e implementa una estrategia de lectura acticipada). Otras funciones de gestión de directorios se definen en fs/ext2/namei.c, como por ejemplo: (1) ext2_match (función interna que compara un nombre de archivo especificado con el nombre contenido en una entrada de directorio); (2) ext2_find_entry (esta función se llama para buscar una entrada en un directorio); (3) ext2_lookup (esta función implementa la operación lookup. Esta operación efectúa primero una búsqueda del nombre de archivo en el caché de directorios llamando a dcache_lookup, si el nombre se encuentra en el caché el inodo correspondiente se lee llamando a iget y se devuelve; si no, se busca el nombre en el directorio llamando a la función ext2_find_entry, el resultado se añade en el caché de directorios por medio de dcache_add, el indo se carga en memoria llamando a iget y se devuelve); (4) ext2_add_entry (esta función se llama para crear una nueva entrada en un directorio); (5) ext2_delete_entry (función para suprimir una entrada de directorio, libera la entrada poniendo a 0 el número de inodo de la entrada si se trata de la primera entrada de un bloque, fusionando esta entrada con la anterior en caso contrario). Otras funciones que implementan operaciones sobre inodos para la gestión de directorios que se encuentran en el archivo fuente fs/ext2/namei.c son: ext2_create, ext2_link, ext2_mkdir, ext2_mdnod, ext2_rename, ext2_rmdir, ext2_symlink y ext2_unlink; estas funciones son relativamente simples, porque se limitan a encadenar la llamadas de las otras funciones internas.

Las operaciones de entradas/salidas sobre archivos (read, write y release) se implementan en el archivo fs/ext2/file.c. La función generic_file_read es la función que implementa la operación de lectura de datos desde un archivo. La función ext2_file_write implementa la operación write, que verifica sus argumentos y efectúa un bucle de escritura: mientras queden datos por escribir, obtiene un buffer llamando a ext2_getblk, copia una parte de los datos a escribir en la memoria, y marca el buffer como modificado a mark_buffer_dirty. Una vez escritos todos los datos, el descriptor de archivos y el inodo se actualizan. Por otro lado, la función ext2_release_file implementa la operación release sobre el archivo (llama a

ext2_discard_prealloc para liberar los bloques preasignados por ext2_new_block en el último cierre del archivo).

4.8.6. Encontrar un Archivo en un Sistema de Archivos EXT2

Un nombre de archivo Linux tiene el mismo formato que los nombres de archivos de todos los UNIX. Es una serie de nombres de directorios separados por barras ("/") y acabando con el nombre del archivo. Un ejemplo de nombre de archivos podría ser /home/rusling/.cshrc donde /home y /rusling son nombres de directorio y el nombre del archivo es .cshrc. Como todos los demás sistemas Unix, Linux no tiene en cuenta el formato del nombre del archivo; puede ser de cualquier longitud y cualquier carácter imprimible. Para encontrar el inodo que representa a este archivo dentro de un sistema de archivos EXT2 el sistema debe analizar el nombre del archivo directorio a directorio hasta encontrar el archivo en sí. El primer inodo que se necesita es el inodo de la raíz del sistema de archivos, que está en el superbloque del sistema de archivos. Para leer un inodo EXT2 hay que buscarlo en la tabla de inodos del Grupo de Bloques apropiado. Si, por ejemplo, el número de inodo de la raíz es 42, entonces necesita el inodo 42avo de la tabla de inodos del Grupo de Bloques 0. El inodo raíz es para un directorio EXT2, en otras palabras el modo del inodo lo describe como un directorio y sus bloques de datos contienen entradas directorio EXT2.

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

4.8.7. Cambiar el Tamaño de un Archivo en un Aistema de Archivos EXT2

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

Siempre que un proceso intenta escribir datos a un archivo, el sistema de archivos Linux comprueba si los datos exceden el final del último bloque para el archivo. Si lo hace, entonces tiene que reservar un nuevo bloque de datos para el archivo. Hasta que la reserva no haya acabado, el proceso no puede ejecutarse; debe esperarse a que el sistema de archivos reserve el nuevo bloque de datos y escriba el resto de los datos antes de continuar. La primera cosa que hacen las rutinas de reserva de bloques EXT2 es bloquear el superbloque EXT2 de ese sistema de archivos. La reserva y liberación cambia campos del superbloque, y el sistema de archivos Linux no puede permitir más de un proceso haciendo esto a la vez. Si otro proceso necesita reservar más bloques de datos, debe esperarse hasta que el otro proceso acabe. Los procesos que esperan el superbloque son suspendidos, no se pueden ejecutar, hasta que el control del superbloque lo abandone su usuario actual. El acceso al superbloque se garantiza mediante una política "FIFO", y cuando un proceso tiene control sobre el superbloque lo bloquea hasta que no lo necesita más, el proceso comprueba que hay suficientes bloques libres en ese sistema de archivos. Si no es así, el intento de reservar más bloques falla y el proceso cederá el control del superbloque del sistema de archivos.

Si hay suficientes bloques en el sistema de archivos, el proceso intenta reservar uno. Si el sistema de archivos EXT2 se ha compilado para prereservar bloques de datos (ext2_new_block() en fs/ext2/balloc.c) entonces se podrá usar uno de estos. La prereserva de bloques no existe realmente, sólo se reservan dentro del mapa de bits de bloques reservados. El inodo VFS que representa el archivo que intenta reservar un nuevo bloque de datos tiene dos campos EXT2 específicos, prealloc_block y prealloc_count, que son el numero de bloque del primer bloque de datos prereservado y cuantos hay, respectivamente. Si no había bloques prereservados o la reserva anticipada no está activa, el sistema de archivos EXT2 debe reservar un nuevo bloque. El sistema de archivos EXT2 primero mira si el bloque de datos después del último bloque de datos del archivo está libre. Lógicamente, este es el bloque más eficiente para reservar ya que hace el acceso secuencial mucho más rápido. Si este bloque no está libre, la búsqueda se ensancha y busca un bloque de

datos dentro de los 64 bloques del bloque ideal. Este bloque, aunque no sea ideal está al menos muy cerca y dentro del mismo Grupo de Bloques que los otros bloques de datos que pertenecen a ese archivo. Si incluso ese bloque no está libre, el proceso empieza a buscar en los demás Grupos de Bloque hasta encontrar algunos bloques libres. El código de reserva de bloque busca un cluster de ocho bloques de datos libres en cualquiera de los Grupos de Bloque. Si no puede encontrar ocho juntos, se ajustará para menos. Si se quiere la prereserva de bloques y está activado, actualizará prealloc_block y prealloc_count, pertinentemente. Donde quiera que encuentre el bloque libre, el código de reserva de bloque actualiza el mapa de bits de bloque del Grupo de Bloques y reserva un buffer de datos en el buffer caché. Ese buffer de datos se identifica unívocamente por el identificador de dispositivo del sistema y el número de bloque del bloque reservado. El buffer de datos se sobrescribe con ceros y se marca como "dirty" para indicar que su contenido no se ha escrito al disco físico. Finalmente, el superbloque se marca como "dirty" para indicar que se ha cambiado y está desbloqueado. Si hubiera otros procesos esperando, al primero de la cola se le permitiría continuar la ejecución y tener el control exclusivo del superbloque para sus operaciones de archivo. Los datos del proceso se escriben en el nuevo bloque de datos y, si ese bloque se llena, se repite el proceso entero y se reserva otro bloque de datos.

4.9. EL SISTEMA DE ARCHIVOS */proc*

El sistema de archivos */proc* es algo particular: no da acceso a datos almacenados en disco, sino que hace accesibles, en forma de archivos virtuales, ciertas informaciones gestionadas por el kernel.

Los archivos accesibles en el sistema de archivos */proc* son los siguientes:

- *cmdline:* argumentos pasados al kernel en el arranque del sistema;
- *cpuinfo:* descripción del procesador o procesadores utilizados;
- *devices:* lista de gestores de dispositivos incluidos en el kernel;
- *dma:* lista de canales DMA utilizados por los gestores de dispositivos;
- *filesystems:* lista de tipos de sistemas de archivos soportados por el kernel;
- *interrupts:* lista de interrupciones de hardware utilizadas por los gestores de dispositivos;
- *iomem:* este archivo contiene la lista de memoria que está ocupada drivers hardware o por el kernel. Se indica la direcciones inicial y final del área ocupada y el nombre del hardware;
- *ioports:* lista de puertos de entrada/salida utilizados por los gestores de dispositivos;
- *kcore:* memoria asignada al kernel;
- *kmsg:* últimos mensajes mostrados por el kernel;
- *ksyms:* lista de símbolos del kernel utilizables por módulos;
- *loadavg:* carga del sistema;
- *locks:* lista de bloqueos asociados a los archivos;
- *malloc:* este archivo permita la monitorización de las operaciones kmalloc() y kfree();
- *md:* si Múltiple device driver support (CONFIG_BLK_DEV_MD) ha sido configurado, este archivo contiene las estadísticas de uso;
- *meminfo:* estado de ocupación de la memoria central;
- *modules:* lista de módulos cargados en el kernel;
- *mounts:* lista de sistemas de archivos montados actualmente;
- *partitions:* este archivo contiene información acerca de las particiones de todos los dispositivos modo bloque: el major number, el minor number, su tamaño y el nombre;
- *pci:* lista de dispositivos conectados en el bus PCI;
- *profile:* informaciones de *profiling* del kernel, utilizadas para determinar el tiempo pasado en ejecutar cada una de las funciones;
- *rtc:* informaciones relacionadas con el reloj en tiempo real;
- *slabinfo:* este archivo contiene una panorámica (overview) de todos los objetos caché utilizados;
- *smp:* este archivo contiene información sobre las CPUs individuales en sistemas SMP (Symmetric MultiProcessing), es decir, información relacionada con el tratamiento multiprocesadores;
- *stat:* estadísticas diversas sobre las operaciones efectuadas por el kernel (tiempo de procesador consumido, número de E/S en disco, número de cargas de páginas en memoria, número de interrupciones de hardware tratadas, número de cambios de contexto efectuados, fecha y hora de arranque del sistema, y número total de procesos creados);
- *swaps:* contiene los datos acerca de las áreas simples del swap;

- *uptime:* tiempo pasado desde el arranque del sistema;
- *version*: versión del kernel.

Además de estos archivos, el directorio */proc* contiene varios directorios:

- *bus/:* este directorio contiene los archivos del sistema de bus, generalmente, esta es la descripción del bus PCI;
- *ide/:* este directorio contiene los descriptores del controlador IDE en el sistema y los dispositivos afiliados;
- *net/:* archivos que contienen las informaciones relacionadas con los protocolos de red (es decir, directorio que contiene algunos de archivos que describen el estado del nivel de red);
- *scsi/:* archivos que contienen las informaciones relacionadas con los gestores de dispositivos SCSI;
- *self/:* enlace con el directorio correspondiente al proceso actual (este directorio contiene sobre los procesos que están accediendo al sistema de archivos *proc*);
- *sys/:* archivos que contienen las informaciones relacionadas con las variables del kernel gestionadas por la primitiva *sysctl*;
- *sysipc/:* este directorio contiene información que describe los mensajes, semáforos y áreas de memoria compartida utilizados en el sistema;
- *tty/:* este directorio contiene información que describe los terminales y sus drivers;
- un directorio por proceso existente en el sistema: el nombre de este directorio es el número del proceso, y este directorio contiene los archivos siguientes:
  - *cmdline:* lista de argumentos del proceso;
  - *cwd:* enlace al directorio actual del proceso;
  - *environ:'lista* de variables de entorno del proceso;
  - *exe:* enlace al _chivo _binario ejecutado por el proceso;
  - *fd:* directorio que contiene enlaces a los archivos abiertos por el proceso;
  - *maps:* lista de zonas de memoria contenidas en el espacio de direccionamiento del proceso;
  - *mem:* contenido del espacio de direccionamiento del proceso;
  - *root:* enlace al directorio raíz del proceso;
  - *stat, statm, status:* estado del proceso.

4.9.1. Entradas de */proc*

Las entradas del directorio */proc* (archivos o directorios) se gestionan de manera dinámica: una lista de descriptores es mantenida en memoria por el kernel, y se explora accediendo al contenido de */proc*.

La estructura *proc_dir_entry*, definida en el archivo <linux/proc_fs.h>, representa el tipo de estos descriptores. Contiene los campos siguientes:

```
struct proc_dir_entry {
        unsigned short low_ino;                 /* Número de inodo asociado a la entrada */
        unsigned short namelen;                 /* Tamaño del nombre de la entrada */
        const char *name;                       /* Nombre de la entrada */
        mode_t mode;                            /* Tipo y derechos de acceso */
        nlink_t nlink ;                         /* Número de enlaces */
        uid_t uid;                              /* Identificador del usuario propietario */
        gid_t gid;                              /* Identificador del grupo de usuarios asociado */
        unsigned long size;                     /* Tamaño en bytes */
        struct inode_operations *ops;           /* Operaciones relacionadas con la entrada */
        int (*) (char *, char **, off_t, int, int) get_info;    /* Puntero a la función llamada en una lectura */
        void (*) (struct inode) fill_inode;     /*Puntero a la función encargada de inicializar los atributos */
                                        /* de la entrada (tipo, derechos de acceso, usuario y grupo propietario) */
        struct proc_dir_entry *next;            /* Puntero al descriptor de la entrada siguiente */
        struct proc_dir_entry *parent;          /* Puntero al descriptor del directorio padre */
        struct proc_dir_entry *subdir;          /* Puntero al descriptor de la primera entrada de! Directorio */
        void *data;                             /* Datos privados asociados a la entrada */
};
```

Los números de inodos se asignan estáticamente a las entradas de */proc*. El archivo de cabecera define varios tipos en este sentido:

- *root_directory_inos*: números de inodos asociados a las entradas de */proc*, comprendidos entre 1 y 127;
- *net_directory_inos*: números de inodos asociados a las entradas de */proc/net*, comprendidos entre 128 y 255;
- *scsi_directory_inos*: números de inodos asociados a las entradas de */proc/scsi*, comprendidos entre 256 y 511.

Además, las constantes PROC_DYNAMIC_FIRST y PROC_NDYNAMIC definen los números de inodos que pueden asignarse de manera dinámica.

Los números de inodos asociados a las entradas de los directorios correspondientes a los procesos se calculan según el número del proceso. Este número se desplaza 16 bits hacia la izquierda para generar un número básico, y el tipo pid_directory_inos define un número a añadir a esta base para obtener el número de inodo. Por ejemplo, el número de inodo del archivo *root* contenido en el directorio correspondiente al proceso de número *p* será p*65536 + 6, porque la constante PROC_PID_ROOT tiene el valor 6.

4.9.2. Operaciones sobre Sistema de Archivos

Las operaciones relacionadas con el sistema de archivos se implementan en el archivo fuente *fs/proclroot.c*.

La función *proc_get_inode* se encarga de inicializar el contenido de un descriptor de inodo: llama a iget para cargar el inodo, e inicializa algunos de sus campos desde el descriptor de la entrada correspondiente. En esta función, la dirección del descriptor de entrada se guarda en el campo generic_ip del descriptor de inodo a fin de poder acceder en la ejecución de las operaciones sobre inodos.

La función *proc_read_super* se llama en el montaje del sistema de *archivos/proc*. Inicializa el descriptor de archivos y llama a la función proc_get_inode para inicializar el inodo correspondiente a la raíz del sistema de archivos.

La función *proc_put_super* se llama al desmontar el sistema de archivos. Se limita a poner a cero el campo s_dev del descriptor afectado para indicar que el sistema de archivos ya no está montado.

La función *proc_read_inode* se llama para leer el contenido de un inodo. Calcula un número de proceso desplazando el número de inodo de 16 bits hacia la derecha, y efectúa una búsqueda del descriptor de proceso correspondiente en la tabla task.

Tras esta búsqueda, pueden presentarse tres casos:

1. El número de proceso no se ha encontrado.
2. El número de inodo corresponde a una entrada estática de */proc*. El campo i_op del descriptor del inodo se inicializa entonces con la dirección de las operaciones sobre inodos específicos del archivo (por ejemplo, proc_kmsg_inode_operations para el archivo /proc/kmsg).
3. El número de proceso ha sido hallado: los 8 bits de menor peso del número de inodo se utilizan entonces para determinar cuál es la entrada del directorio correspondiente al proceso. El campo i_op del descriptor del inodo se inicializa con la dirección de las operaciones sobre inodos específicos del archivo (por ejemplo, proc_mem_inode_operations para el archivo *mem)*.

4.9.3. Operaciones para la Gestión de Directorios

El archivo fuente *fs/proclroot.c* contiene las funciones que permiten gestionar el contenido de los directorios. La variable global proc_root contiene el descriptor de la raíz del sistema de archivos.

La función *proc_register* registra una nueva entrada en un directorio, y la función proc_unregister suprime una entrada borrando su descriptor de la lista.

La función *proc_register_dynamic* atribuye dinámicamente un número de inodo a un descriptor y guarda la entrada correspondiente en la lista. La inicialización de la lista de archivos y directorios situados en */proc* se efectúa por proc_root_init. Esta función llama a la función proc_register para registrar las entradas contenidas en */proc*.

La función *proc_lookup* efectúa la exploración de un directorio. Explora la lista encadenada de las entradas registradas en el directorio y compara el nombre especificado con el nombre de archivo contenido en cada descriptor. proc_root_lookup efectúa la búsqueda de un nombre de entrada en el directorio raíz. Llama a proc_lookup y comprueba su resultado. Si proc_lookup ha encontrado la entrada especificada (por tanto, si el nombre se refiere a una entrada estática registrada llamando a proc_register), se devuelve el resultado. En el caso contrario, el nombre especificado debe representar un directorio correspondiente a un proceso. El nombre se convierte en número de proceso, se efectúa una búsqueda del proceso correspondiente en la tabla task, y se construye el número de inodo correspondiente a partir del número de proceso.

La función *proc_readdir* se llama para obtener una lista de entradas contenidas en un directorio. Se basa en el valor del campo f_pos del descriptor de archivo correspondiente al directorio: este campo contiene el Índice de la entrada a devolver. proc_readdir explora la lista de entradas registradas en el directorio, y llama a la función especificada por el parámetro filldir, a fin de colocar los nombres de las entradas en la memoria intermedia proporcionada por el usuario. La función proc_root_readdir implementa la llamada *readdir* para el directorio raíz. Si la posición en el directorio es inferior a FIRST_PROCESS_ENTRY, llama a proc_readdir para obtener los nombres de entradas registradas. Luego explora la tabla task, convierte los números de procesos en cadenas de caracteres, y las coloca en la memoria intermedia proporcionada por la llamada a la función especificada por el parámetro filldir.

Se utiliza un solo descriptor para referenciar todos los directorios correspondientes a procesos: la variable proc_pid definida en el archivo *fuentejs/proc/base.c*. Su contenido se inicializa por la función proc_base_init, llamada por proc_root_init. Esta función llama a proc_register para registrar las entradas *cmdline, cwd, environ, exe,jd, maps, mem, root, stat, statm* y *status*.

4.9.4. Operaciones sobre Inodos y sobre Archivos

El sistema de archivos */proc* utiliza los punteros a las operaciones sobre inodos y sobre archivos abiertos para diferenciar los tratamientos asociados a las entradas. La mayor parte de los tratamientos se implementa en el archivo fuente *js/proc/array.c*.

La función *array_read* implementa la operación sobre archivo read. Asigna primero una página de memoria, que servirá de memoria intermedia llamando a __get_free_page, y obtiene el descriptor de entrada asociada al inodo. Seguidamente, llama a la operación get_info asociada a la entrada, o la función fill_array si no se ha definido ninguna operación. Finalmente, copia el resultado devuelto en la memoria intermedia proporcionada por el proceso que llama.

La función *fill_array* se llama para leer el contenido de un archivo, cuando la entrada correspondiente no posee operación get_info. Llama a get_process_array si el inodo está contenido en un directorio correspondiente a un proceso, o get_root_array en caso contrario. Estas dos funciones comprueban el número de inodo del archivo, y llaman a una función que se encarga de convertir los datos internos del kernel en una cadena de caracteres.