Sistemas Operativos Avanzados (MUII)
Ejercicio de ficheros (16-mayo-2021)
Revisión de los ext[2-4] de Linux
Fernando Pérez Costoya

Planteamiento del trabajo

Uno de los puntos fuertes de Linux es su versatilidad a la hora de dar soporte a diferentes tipos de sistemas de ficheros provenientes de muy diversos ámbitos. Gracias al *Virtual File System* (VFS), que comentaremos brevemente un poco más adelante, y con el apoyo de la pujante comunidad de desarrollo de Linux, prácticamente cualquier tipo de sistema de ficheros acaba siendo integrado de una manera transparente dentro de este sistema operativo.

En cualquier caso, Linux ofrece su propio sistema de ficheros nativo, llamado *Extended*, que ha ido evolucionando progresivamente incorporando nuevas funcionalidades:

- *Extended* (*ext*): Apareció en 1992 para solventar algunas de las deficiencias que presentaba el sistema de ficheros de MINIX. Conviene recordar que Linux surgió como un experimento de Linus Torvalds, que tomó como punto de partida el sistema operativo de carácter didáctico MINIX de Andrew Tanenbaum, por lo que inicialmente usaba el sistema de ficheros de MINIX.
- *Ext2*, la segunda versión de este sistema de ficheros, surgió aproximadamente un año después y, además de eliminar todas las limitaciones de MINIX que se consideraban inconvenientes, proporciona (en tiempo presente, puesto que este sistema de ficheros sigue estando activo) buenas prestaciones y, sobre todo, una gran robustez. Se trata de la versión de referencia, como lo demuestra que muchas de las utilidades disponibles para el manejo del sistema de ficheros nativo incluyen en su nombre el término *e2fs* (*mke2fs*, *tune2fs*…), a pesar de que también se pueden usar con versiones posteriores de este sistema de ficheros.
- *Ext3*, que apareció en 2001, cuya principal novedad es la incorporación de la técnica del *journaling*, manteniendo compatibilidad con la versión previa.
- *Ext4*, la versión actual, que surgió en 2008 y que, entre otras funcionalidades, incluye el uso de *extents* para gestionar el espacio ocupado por un fichero.

El trabajo se plantea como una revisión de la funcionalidad de las sucesivas versiones de este sistema de ficheros nativo de Linux (también integrado en otros sistemas operativos), lo que va a permitir entender de forma genérica e incremental cómo ha sido la evolución de este componente del sistema operativo en las últimas décadas.

Si bien es cierto que hay algunas funcionalidades que están presentes en los sistemas de ficheros más modernos, como *btrfs*, que no están disponibles en *ext4* (como, por ejemplo, el mecanismo de instantáneas basado en la técnica del *copy-on-write* o la gestión de volúmenes incorporada al propio sistema de ficheros), en términos generales, *ext4* engloba la mayoría de las técnicas actuales de gestión de ficheros.

Este documento no es meramente el enunciado de un ejercicio, sino que se plantea como la documentación necesaria y suficiente para afrontarlo, de ahí su considerable longitud. Las primeras secciones revisan los conceptos teóricos requeridos para afrontar este ejercicio sin plantear ninguna cuestión. En la segunda parte del documento, a partir de la sección “El sistema de ficheros ext2” en la página 11, hay una sección por cada uno de los sistemas de ficheros (*ext2*, *ext3* y *ext4*) y es donde se plantean las preguntas del ejercicio. Si el lector conoce los fundamentos de los sistemas de ficheros de UNIX puede comenzar directamente en esa sección.

Características deseables en un sistema de ficheros

Un sistema de ficheros ofrece dos abstracciones básicas: los ficheros y los directorios.

La abstracción de fichero ofrece al usuario una visión que le hace creer que, a la hora de hacer persistente la información que maneja, tiene un "disco" dedicado para cada "objeto".

En cuanto a los directorios, esta abstracción permite establecer una organización jerárquica en el espacio de nombres de los ficheros que facilita al usuario su manejo. A nadie nos gustaría usar un sistema de ficheros sin directorios igual que a ninguna persona le agradaría trabajar en una oficina donde se apilan los documentos de papel sin ningún criterio en vez de usar carpetas y archivadores para almacenarlos.

Todo sistema de ficheros implementa ambas abstracciones, pero no nos basta con eso; sería deseable que un sistema de ficheros proporcionara (el orden es irrelevante):

- Gestión eficiente de la información de traducción. Para poder implementar la abstracción de fichero es necesario almacenar de alguna forma la información que vincula cada bloque del fichero con el bloque del disco donde está almacenado (la metainformación de traducción). La gestión de esa información debe ser eficiente tanto en el sentido de proporcionar un acceso rápido a cualquier parte de un fichero, sea este del tamaño que sea, como en el espacio ocupado por la propia metainformación.
- Asignación de espacio para un acceso eficiente. La estrategia de asignación de espacio a ficheros y directorios debe favorecer el acceso eficiente a los mismos, teniendo en cuenta el modo de operación de los discos tradiciones que penalizan los accesos dispersos. Asimismo, debe realizar un buen uso del espacio del disco.
- Eliminación y/o mejora de los límites en los parámetros de funcionamiento del sistema de ficheros. En el modo de operación de un sistema de ficheros se presentan distintos factores limitantes tales como el tamaño máximo del sistema de ficheros, el tamaño máximo de un fichero, el número máximo de ficheros existentes o el número máximo de entradas en un directorio. El objetivo sería, idealmente, eliminar estos parámetros limitantes o, al menos, hacerlos lo menos restrictivos que sea posible.
- Robustez. Cada operación sobre un sistema de ficheros implica normalmente varias acciones independientes con lo que, si se cae el equipo en medio de estas acciones, puede que el sistema de ficheros quede corrupto. Dada la importancia de la información que se almacena en un sistema de ficheros, la capacidad de tolerar adecuadamente estas situaciones patológicas o las causadas por el malfuncionamiento del propio dispositivo es clave a la hora de valorar un sistema de ficheros. Es necesario, además, que los procesos de recuperación ante contingencias sean ágiles.
- Flexibilidad en la gestión de los atributos de un fichero. Cada fichero tiene asociados atributos o propiedades tales como su dueño, los permisos de acceso o las fechas de distintos tipos de accesos al fichero, que, junto con la información de traducción, componen su metainformación. Una característica deseable es proporcionar un mecanismo extensible que permita incluir nuevos tipos de atributos, ya sea para añadir información de sistema que no estaba prevista cuando se diseñó el sistema de ficheros (por ejemplo, atributos vinculados con nuevos modelos de seguridad), como para permitir al usuario que defina sus propios atributos. Por ejemplo, un usuario podría incluir como un atributo definido por el mismo quién es el autor de un documento, que puede ser, obviamente, diferente del dueño del fichero. Con este mecanismo, esa información complementaria forma parte del propio fichero, en vez de tener que almacenarla en otro fichero independiente. Una extensión de este mecanismo permitiría almacenar en el mismo fichero múltiples contenidos alternativos, como, por ejemplo, versiones en distintos idiomas de un documento. Este último mecanismo está soportado por el NTFS de Windows bajo el nombre de *streams*.
- Acceso eficiente a directorios de gran volumen. Aunque, en principio, la propia idiosincrasia de un directorio es tener un número relativamente reducido de entradas (si tengo un directorio con un

número muy elevado de entradas, probablemente no esté organizando bien mi información), algunos directorios pueden contener un gran volumen de ficheros que se generan automáticamente, como una cache de un navegador. En esos casos, sí puede ser necesario asegurar el acceso eficiente a una entrada de un directorio a pesar de la existencia de un número muy elevado de entradas en el mismo.

- Adaptación a discos de tamaño variable. Un sistema de ficheros se almacena en un soporte físico que puede corresponder a un disco, una partición de un disco o un volumen, en el caso de usar un gestor de volúmenes. En los dos últimos casos, el tamaño de ese soporte puede modificarse y, por tanto, sería deseable que el sistema de ficheros pudiera reajustarse a las nuevas dimensiones del soporte mientras se mantiene el dispositivo operativo. Nótese que, con el uso de máquinas virtuales, incluso los discos pueden ser redimensionados.

El sistema de ficheros de UNIX

Linux es un sistema operativo de la familia UNIX y, por tanto, su sistema de ficheros nativo se basa en el de ese sistema operativo seminal. En esta sección repasaremos los principales aspectos del mismo, así como conceptos generales del diseño de sistemas de ficheros. El lector probablemente ya conocerá muchos de los aspectos que se presentan en esta sección, en la que no se propone ningún ejercicio, pero se pretende asegurar un bagaje común sobre este tema.

El inodo

Habitualmente, un sistema de ficheros gestiona algún tipo de descriptor para almacenar toda la metainformación del fichero. Por un lado, información de carácter general, como el tipo de fichero, que, como mínimo distinguiría entre ficheros normales y directorios (en un sistema UNIX, existen otros tipos como ficheros especiales de caracteres y bloques para los dispositivos, enlaces simbólicos, FIFOs o sockets del dominio local), el dueño, los permisos de acceso o las diversas fechas que almacena el sistema de ficheros por cada fichero (como la fecha del último acceso a los datos del fichero o la de la última modificación de los datos o de la propia metainformación).

Por otro lado, existe información de ubicación que permite conocer en qué bloque del disco (bloque físico) se almacena cada bloque del fichero (bloque lógico). Con respecto a este último aspecto, se puede considerar como una cuestión de diseño de estructuras de datos: cómo gestionar una colección de objetos que, en este caso, corresponden a los distintos bloques de disco asociados al fichero, para tener un acceso eficiente directo al enésimo elemento de esa colección. En principio, desde un punto de vista teórico, se presentan dos alternativas principales: listas o vectores.

- Con un esquema de tipo lista, cada bloque del fichero, además de los datos, almacenaría el número del bloque del disco que corresponde al siguiente bloque lógico del fichero, con un valor especial para indicar el fin de fichero. En el descriptor de fichero se almacenaría solamente el número del bloque del disco que contiene el primer bloque del fichero. Como ya se podía anticipar *a priori*, la lista no es el mecanismo adecuado para tener acceso directo al enésimo elemento de la misma, puesto que requeriría leer todos los bloques previos (complejidad *O(n)*). A pesar de esa deficiencia consustancial a su propio diseño, este esquema de lista es el usado por el sistema de ficheros FAT de Microsoft, con una modificación que mejora el rendimiento de los accesos directos: el puntero al siguiente bloque no se guarda en el propio bloque, sino que se externaliza guardándolo en una estructura almacenada en el sistema de ficheros denominada precisamente FAT (*File Allocation Table*). La FAT es una tabla con una entrada por cada bloque del disco tal que la entrada correspondiente a un cierto bloque físico, que contendría un determinado bloque lógico de un fichero, almacena el número del próximo bloque del disco que alberga el siguiente bloque lógico de ese fichero, con un valor especial para marcar el final de fichero y otro para indicar que ese bloque está libre y no está asociado a ningún fichero. Con este esquema, para acceder al bloque lógico enésimo de un fichero no hay que leer todos los bloques de datos previos del fichero, sino todas las entradas de la FAT correspondientes a esos bloques previos, lo que, suponiendo que el fichero está relativamente contiguo

en el disco, puede suponer un ahorro de hasta tres órdenes de magnitud (suponiendo, por ejemplo, que en cada bloque de disco caben 1024 direcciones de bloque de disco y asumiendo que el fichero está contiguo en el disco, leer un bloque de la FAT proporcionaría información de los siguientes 1024 bloques lógicos del fichero).

- El esquema más natural es un vector con una entrada por cada bloque del fichero. En el descriptor del fichero se almacena ese vector de manera que para acceder al bloque enésimo del fichero se consulta la posición enésima del vector obteniendo el número de bloque físico de disco donde está almacenado ese bloque lógico del fichero.

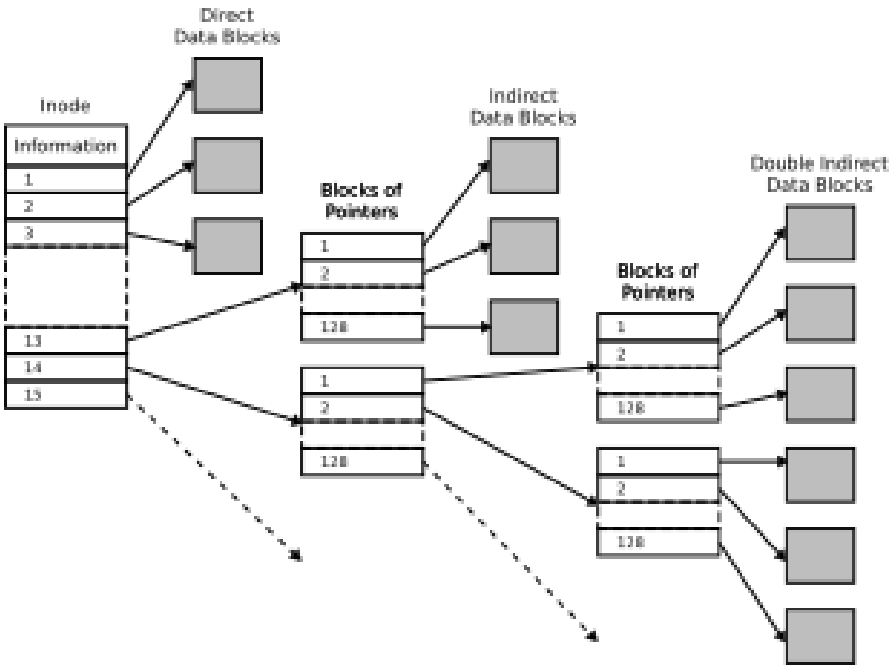
Con una aplicación directa del segundo esquema, el descriptor del fichero tendría un tamaño variable puesto que contendría el vector de traducción con una entrada por cada bloque del fichero. El uso de descriptores de tamaño variable dificulta la gestión de los ficheros. La solución de UNIX plantea el uso de un vector de traducciones, pero manteniendo un descriptor de tamaño fijo, denominado *inodo* (en inglés *inode*, nodo índice, porque indica donde están almacenados los bloques del fichero).

Para conseguir gestionar un vector de tamaño variable manteniendo un tamaño de descriptor fijo, se trocea ese vector almacenando cada trozo en su propio bloque del disco en lugar de en el inodo, donde sí se guardará la información necesaria para localizar cada trozo. Haciendo un símil, en vez de disponer del mapa del tesoro, tenemos un mapa para poder llegar al mapa del tesoro (nuevamente, aparece la indirección como la solución universal en la informática).

Con la aplicación de esta técnica de troceado, el diseño del inodo incluye un número fijo de números de bloques de disco (de punteros a bloques de disco), que se distribuyen habitualmente de la siguiente forma:

- El inodo guarda directamente las primeras posiciones del vector: típicamente, una docena de punteros directos a los bloques de disco que alojan esos doce primeros bloques lógicos del fichero. De esta forma, los ficheros con un tamaño inferior a ese límite no se ven afectados por una sobrecarga ni en lo que se refiere al tiempo de acceso al fichero ni al espacio ocupado por el mismo
- El decimotercer puntero guarda la dirección de un bloque del disco, denominado bloque de indirección simple, que contiene las siguientes *N* posiciones del vector de traducción, siendo *N* el número de direcciones de bloques de disco que caben en un bloque de disco. Por tanto, ese bloque indirecto simple almacenará el fragmento del vector que va desde la posición decimotercera hasta la que corresponde al número 13+*N*-1. En consecuencia, el acceso a un bloque lógico cuyo número esté comprendido en ese intervalo requiere un acceso adicional (al bloque indirecto simple) y ocupa un bloque adicional (ese indirecto simple).
- El decimocuarto puntero apunta a un bloque indirecto doble, es decir, a un bloque de disco que contiene *N* punteros que referencian a su vez a bloques indirectos simples, creándose un árbol con dos niveles, que penaliza tanto en la eficiencia en el acceso como en el gasto de almacenamiento adicional.
- El decimoquinto puntero crea un nivel de indirección adicional y, por tanto, un árbol de tres niveles: este puntero apunta a un bloque indirecto triple, es decir, a un bloque de disco que contiene *N* punteros que referencian a bloques indirectos dobles, cada uno de los cuales, a su vez, contiene *N* punteros que apuntan a bloques indirectos simples. Obsérvese que el acceso a un bloque lógico en este rango conlleva la lectura de tres bloques adicionales.

La siguiente figura, tomada de la Wikipedia, muestra esta estructura para el caso de un bloque de 512 bytes (demasiado pequeño para los estándares actuales) y un tamaño de la dirección de bloque de 4 bytes, habiendo, por tanto, 128 punteros por bloque. Nótese que en la misma no se aprecia el uso del puntero al bloque indirecto triple (puntero decimoquinto) al no estar desplegado explícitamente.



El inodo facilita la gestión eficiente de ficheros con huecos. Si un proceso crea un fichero, mueve el puntero a una determinada posición que corresponde a un cierto bloque lógico y escribe, por ejemplo, 1 byte, no hay que reservar espacio en disco para todos los bloques lógicos previos (para el hueco), sino que basta con almacenar un valor nulo en todos ellos y reservar solo espacio para ese bloque recién escrito y los bloques indirectos requeridos en caso de que sea preciso. Si un fichero lee de un hueco (es decir, hay un valor nulo en el puntero correspondiente al bloque leído), no es necesario acceder al dispositivo, devolviéndole directamente ceros por motivos de seguridad.

Gestión de nombres: el directorio

Un directorio es una colección de descriptores de ficheros que crea un espacio de nombres independiente facilitando el nombrado de los distintos objetos persistentes. Dado que uno de los ficheros que forman parte de un directorio puede ser a su vez un directorio, este mecanismo crea un árbol de nombrado jerárquico para todos los objetos del sistema, de manera que cada uno de ellos queda identificado por una ruta dentro del árbol (el camino para llegar desde el directorio raíz hasta el fichero correspondiente). El directorio es, a su vez, un fichero que almacena en sus datos los descriptores de los ficheros y directorios que contiene.

A la hora de analizar el diseño de esta abstracción, hay que tener en cuenta las dos principales operaciones que se realizan sobre un directorio:

- Leer sus entradas: se trata normalmente de una operación secuencial que va recorriendo las sucesivas entradas del directorio, ya sea solo para obtener el nombre de los ficheros que contiene (como en el mandato *ls*) o para conseguir información de atributos adicionales de esos ficheros (como en el mandato *ls -l*).
- Atravesar el directorio cuando se traduce una ruta que pasa por el mismo (operación denominada habitualmente *lookup*): implica una búsqueda del nombre del siguiente tramo de la ruta entre todas las entradas del directorio.

El diseño más natural de un esquema de directorios es que cada uno almacene una colección de los descriptores de cada fichero o directorio incluido en el mismo, siendo el nombre de un fichero un atributo más almacenado en el descriptor del mismo.

Frente a ese esquema de almacenar el descriptor *in situ*, que utiliza, por ejemplo, el sistema de ficheros FAT, la alternativa usada en UNIX es externalizar el descriptor almacenándolo de forma independiente y dejando en cada entrada del directorio solamente el nombre del fichero y un identificador de su descriptor (el número de su inodo), del que se elimina como atributo el nombre. En consecuencia, aunque parezca un poco sorprendente *a priori*, un inodo almacena todos los atributos de un fichero, excepto su nombre.

Al no estar el inodo incluido en el directorio, surge la cuestión de dónde almacenarlo. El sistema de ficheros de UNIX reserva una zona del disco de tamaño fijo, denominada tabla de inodos, para guardar todos los inodos que pueden existir en el sistema ordenados por su número.

Analizándolo desde un punto de vista de diseño de estructuras de datos, la primera opción correspondería a usar un vector que contiene los objetos asociados a una colección, tal que cada objeto contiene, entre otros atributos, su nombre, mientras que la segunda plantearía el uso de un mapa o diccionario que por cada objeto asocia su nombre con una referencia al objeto propiamente dicho, eliminando del objeto el atributo que almacena su nombre.

Comparando las dos soluciones, cada una tiene sus ventajas y sus inconvenientes, como revisamos a continuación, empezando por los puntos más débiles del esquema de UNIX.

En el esquema de UNIX, si se pretende obtener algunos atributos de todos los ficheros de un directorio es necesario ir leyendo de forma sucesiva y alternada los bloques del disco que contienen las entradas del directorio y los de la tabla de inodos que albergan los inodos asociados a esas entradas. Ese vaivén en los accesos al disco penaliza el rendimiento de la operación frente al uso de la primera alternativa donde se leen todos los descriptores de un tirón.

Asimismo, la solución UNIX establece lo que podríamos denominar como dos ámbitos en la gestión de recursos. Cuando se crea una nueva entrada con el primer esquema, basta con buscar espacio en el disco para almacenar esa nueva entrada que contiene el descriptor. Sin embargo, con el esquema de UNIX, además de buscar espacio para la nueva entrada, que contiene el nombre y el número de inodo del nuevo fichero, es necesario buscar un inodo libre en la tabla de inodos para almacenar el nuevo inodo.

El esquema de UNIX tiene como principal beneficio que permite de forma directa la asignación de varios nombres (alias, sinónimos o como queramos denominarlos) al mismo fichero, mediante el concepto de enlace, lo que enriquece el modelo de nombres soportado por el sistema de ficheros, como analizamos en la siguiente sección.

Sea cual sea la opción utilizada, si se permite que el nombre de un fichero (es decir, el nombre de cada componente de una ruta) tenga un tamaño máximo relativamente grande (pongamos como ejemplo 255 bytes), se dificulta la gestión, como analizamos a continuación.

En un sistema con un tamaño máximo pequeño para el nombre del fichero (por ejemplo, 14 caracteres en el UNIX original), se puede asignar ese espacio máximo al almacenamiento del nombre de todo fichero, aunque se desperdicie parte del mismo. Eso permite que el tamaño de todas las entradas sea el mismo (en el UNIX original, 16 bytes: 14 para el nombre y 2 para el número de inodo), lo que facilita considerablemente la gestión: una nueva entrada puede usar el espacio ocupado previamente por una entrada borrada.

Si el tamaño máximo es relativamente grande, no es tolerable asignar ese tamaño máximo para almacenar el nombre de cada fichero, sino que habrá que usar el espacio que realmente necesita, lo que provoca que las entradas de directorio tengan un tamaño variable. Eso complica significativamente la gestión del directorio. Para ilustrarlo, tenga en cuenta que, por ejemplo, el espacio liberado al borrar una entrada muy grande puede reutilizarse para almacenar dos entradas pequeñas, quedando incluso un hueco. En definitiva, es necesario implementar un esquema de gestión de espacio dinámico, como ocurre en otros contextos como, por ejemplo, la gestión del *heap*.

Una última consideración sobre la gestión de nombres en un sistema de ficheros es cómo manejar los árboles de nombres que se almacenan en los distintos dispositivos del sistema.

Una alternativa es gestionarlos de manera independiente identificando a cada uno por el nombre que se le asigna al dispositivo. Esta es la solución que se usa en el mundo Windows asignando el archiconocido nombre de la unidad (por ejemplo, *C:*) a cada dispositivo.

La apuesta de UNIX es, sin embargo, más ambiciosa creando un árbol de nombres integrado gracias a la operación de montaje. Esta operación permite vincular el espacio de nombres asociado a un determinado dispositivo (el dispositivo que se pretende montar) con un directorio de otro dispositivo que ya está montado. En el arranque solo está montado el dispositivo raíz al que se van incorporando sucesivamente el resto de los dispositivos de almacenamiento mediante operaciones de montaje.

Enlaces

En el esquema de directorios de UNIX, una entrada de directorio se puede considerar como un enlace que vincula un nombre de fichero con su inodo. Este modelo permite, por tanto, que varias entradas de directorio asocien distintos nombres con el mismo inodo. Para añadir un nuevo nombre a un fichero basta con crear en un directorio una nueva entrada que vincule el nuevo nombre con el número del inodo de ese fichero, teniendo este nuevo nombre las mismas características que el original (de hecho, son indistinguibles y se puede seguir accediendo al fichero con el nuevo nombre, aunque se elimine el original). Ese es el concepto de enlace físico de UNIX (*hard link*; accesible a través del servicio *link* y del mandato *ln*), que requiere asimismo almacenar en el inodo un contador de enlaces para poder determinar cuándo poder borrarlo (en el momento que el número de enlaces sea 0).

Nótese que este tipo de enlaces no puede hacer referencia a objetos almacenados en otros dispositivos montados, puesto que en la entrada se guarda el número del inodo y este solo tiene sentido en el contexto de este dispositivo. Tampoco se permiten que hagan referencia a a un directorio, ya que se podrían crear ciclos en el árbol de nombres. Esta segunda restricción tiene una excepción. En todo directorio siempre hay dos enlaces físicos a directorios: la entrada “.”, que tiene asociado el propio inodo del directorio (apunta a sí mismo), y la entrada “..” que está asociada al inodo del directorio padre. Estas entradas facilitan la especificación de rutas.

Existe en UNIX otro tipo de enlaces, denominados simbólicos (accesibles a través del servicio *symlink* y del mandato *ln -s*), que no tienen esas dos restricciones, pero que tienen una implementación radicalmente distinta. Un enlace simbólico es un nuevo fichero (recuerde que el enlace físico no es un nuevo fichero: es una nueva entrada que apunta a un inodo ya existente) cuyo contenido es una ruta que lleva al fichero al que hace referencia (retomando un símil previo, el enlace simbólico no contiene el tesoro, sino un mapa para llegar al mismo). En principio, esa ruta que define el enlace simbólico estará guardada en los bloques de datos del fichero, aunque, como una optimización, se puede almacenar en su propio inodo si es suficientemente corta.

No es posible implementar enlaces físicos en un sistema que use la primera alternativa planteada en el apartado anterior (el directorio como una colección de entradas) por la propia idiosincrasia de la solución. Sin embargo, sí es factible implementar enlaces simbólicos sobre este esquema.

Gestión del espacio libre

El modo de operación de un sistema de ficheros requiere conocer en todo momento qué bloques del disco están libres. Para realizar esta gestión se presentan, básicamente, dos alternativas:

- Usar una lista que contenga los números de los bloques del disco que están libres, de manera que cuando se queda libre un bloque se añade su número a la lista y cuando se requiera usar uno se elimina de la lista. Una ventaja de este esquema es que esta lista se puede almacenar usando algunos de los propios bloques

libres, no implicando, por tanto, un gasto de espacio adicional. Esta solución podría implementarse como una especie de fichero de sistema o metafichero cuyo contenido es la lista de bloques libres.

- Utilizar un mapa de bits que tenga un bit por cada bloque del dispositivo que refleje el estado de ocupación de dicho bloque. Este esquema conlleva un mayor gasto de espacio en disco al requerir almacenar información del estado de cada bloque, mientras que el anterior solo guarda información de los bloques libres usando, además, para ello los propios bloques libres.

Diferentes versiones de UNIX han usado alternativamente estas opciones, aunque la tendencia progresiva ha sido utilizar el mapa de bits. La principal ventaja del uso de un mapa de bits es que facilita la búsqueda de bloques libres que estén contiguos en el dispositivo a la hora de asignar espacio a un fichero, que es un aspecto clave para obtener un buen rendimiento en los accesos al mismo.

Esta preponderancia de la solución basada en un mapa de bits es razonable a pesar de implicar un mayor gasto de espacio en el disco porque la sobrecarga es tolerable. Veámoslo con un ejemplo de un dispositivo de 4GiB en un sistema de ficheros que usa un bloque de 4KiB. ¿Qué porcentaje del disco ocupa el mapa de bits de los bloques?

$$nº\ de\ bloques\ en\ el\ sistema = \frac{4GiB}{4KiB} = 1MiB\ bloques$$

$$tamaño\ del\ mapa\ de\ bits = 1MiB\ bits\ (1\ bit\ por\ bloque)$$

$$nº\ de\ bloques\ que\ ocupa\ el\ mapa\ de\ bits = \frac{1MiB}{(8 * 4KiB)} = 32\ bloques$$

$$\%\ de\ ocupación\ del\ mapa\ de\ bits = 100 * \frac{32\ bloques\ en\ el\ mapa}{1MiB\ bloques\ en\ el\ disco} = 0{,}003\%$$

En cuanto al esquema usado en los sistemas de ficheros FAT, en principio, no correspondería directamente con ninguna de las dos alternativas presentada y es difícil de clasificar puesto que cada entrada de la FAT tiene un doble rol: marcar que el bloque está libre en caso de que así sea, pero también guardar información de la ubicación del siguiente bloque lógico del fichero si el bloque está ocupado.

Hay que resaltar que en un sistema UNIX es necesario gestionar también el estado de ocupación de los inodos del sistema almacenados en la tabla de inodos, presentándose nuevamente la misma dicotomía: usar una lista con los números de los inodos libres frente a gestionar un mapa de bits de inodos.

Estructura del sistema de ficheros

Para completar el repaso del diseño original del sistema de ficheros de UNIX, en esta sección vamos a presentar cómo está dispuesta en el dispositivo la metainformación requerida por este sistema de ficheros asumiendo que se usan mapas de bits tanto para la gestión de los bloques libres como de los inodos. A continuación, se enumeran las distintas zonas que gestiona este sistema de ficheros, cada una de las cuales ocupa un número entero de bloques, especificadas en el orden en el que aparecen en el dispositivo:

- Ajeno al propio sistema de ficheros, el primer sector de un dispositivo está reservado para almacenar un sector de arranque para el caso de que el dispositivo esté concebido para tal fin.
- El primer bloque propiamente gestionado por el sistema de ficheros es el denominado superbloque, que almacena información general de ese sistema de ficheros, incluyendo aspectos tales como el tamaño del bloque, el del inodo, el número de inodos disponibles y la ubicación del resto de las zonas almacenadas en el dispositivo. Nótese que la información almacenada en este bloque es crítica, quedando el sistema de ficheros inutilizable si hay un problema en el dispositivo que deja inaccesible el contenido de este bloque.

- El mapa de bits de bloques, que, como se explicó previamente, contiene un bit por cada bloque del dispositivo.
- El mapa de bits de inodos, que contiene un bit por cada inodo.
- La tabla de inodos, que almacena todos los inodos del sistema.
- El resto de los bloques están disponibles para almacenar los bloques asignados a los ficheros y directorios, así como los diversos tipos de bloques de indirección.

Todas estas zonas tienen un tamaño fijo y no pueden redimensionarse una vez que se ha creado el sistema de ficheros. Un punto crítico a la hora de crear el sistema de ficheros es estimar correctamente el número de inodos (y, por tanto, de ficheros) que va a haber en el sistema.

Si uno se queda corto, no podrá crear un nuevo fichero, a pesar de tener bloques libres, porque se han agotado los inodos disponibles. En contraposición, en caso de dimensionar una tabla de inodos demasiado grande para el uso que se le va a dar al dispositivo, no será posible crear un nuevo fichero por falta de bloques de datos libres cuando la tabla de inodos está a medio ocupar.

Recapitulando, al crear un sistema de ficheros se debe especificar un número de inodos basándose en cuál es el tamaño medio estimado de los ficheros que se almacenarán en ese dispositivo (no es lo mismo usar un dispositivo para almacenar contenido multimedia que para guardar un repositorio de un proyecto de software), de manera que, si en promedio un fichero ocupa *N* bloques, haya un inodo por cada *N* bloques del disco. Haciendo un símil, si para elaborar un cierto producto comestible necesito 1 kilogramo de sal por cada 4 kilogramos de azúcar, deberé hacer los pedidos de suministros acordes a esa proporción para evitar un parón en el proceso de fabricación por la falta de un ingrediente habiendo suficiente disponibilidad del otro.

Para terminar esta sección, vamos a completar los cálculos de cuánto ocupa la metainformación de un sistema de ficheros retomando el ejemplo que se planteó previamente, con un dispositivo de 4GiB y un bloque de 4KiB, suponiendo además que el tamaño del inodo es de 128 bytes y el tamaño medio estimado de los ficheros es 8KiB, es decir, 2 bloques.

El primer cálculo que vamos a realizar es determinar cuántos inodos habrá en el sistema:

$$nº\ inodos\ en\ el\ sistema = \frac{nº\ bloques\ en\ el\ sistema}{nº\ bloques\ promedio\ por\ fichero} = \frac{1MiB}{2} = 512KiB\ inodos$$

Ya podemos determinar el tamaño del mapa de bits de inodos:

$$nº\ de\ bloques\ que\ ocupa\ el\ mapa\ de\ inodo = \frac{512KiB}{(8 * 4KiB)} = 16\ bloques$$

Y el de la tabla de inodos:

$$nº\ bloques\ tabla\ inodos = \frac{nº\ inodos * tamaño\ inodo}{tamaño\ del\ bloque} = \frac{512KiB * 128}{4KiB} = 16384\ bloques$$

Recapitulando los resultados:

- Superbloque: 1 bloque.
- Mapa de bits de bloques: 32 bloques, como se calculó previamente.
- Mapa de bits de inodos: 16 bloques.
- Tabla de inodos: 16384 bloques.

Por tanto, la metainformación ocupa un poco más del 1,5% del espacio total del dispositivo:

$$\%\ de\ ocupación\ de\ la\ metainformación = 100 * \frac{1 + 32 + 16 + 16384}{1MiB} = 1{,}57\%$$

El *Fast File System* de BSD

El diseño original del sistema de ficheros de UNIX presenta algunas deficiencias de rendimiento causadas por la dispersión de la información asociada con un fichero:

- Cuando se trabaja con un fichero, hay que acceder tanto a su inodo, guardado en la tabla de inodos, como a sus bloques de datos, que pueden estar almacenados en el extremo opuesto del dispositivo. Por mucho esfuerzo que haga el sistema operativo en asignar a un fichero los bloques de datos contiguos, evitando la fragmentación del fichero, hay que visitar ese *lejano* inodo para realizar un acceso al fichero.
- En el caso de que se pretenda obtener algunos de los atributos de los ficheros contenidos en un directorio de un tamaño considerable, es necesario acceder sucesivamente de forma alternada a los bloques de datos del directorio y a los inodos de los ficheros contenidos en ese directorio, que, en general, pueden estar muy *alejados* entre sí.

Para paliar estos problemas intentando acercar cada inodo a los bloques de datos asociados al mismo, en la versión BSD de UNIX se diseñó una mejora sobre el sistema de ficheros original de UNIX, que se denominó, por motivos obvios, *Fast File System* (FFS), y que se ha convertido en la solución estándar para mitigar estos problemas de rendimiento.

La idea es crear grupos de bloques independientes dentro del dispositivo de manera que cada grupo tenga sus propios mapas de bits y tabla de inodos. Siempre que haya espacio suficiente, los bloques de datos de un fichero estarán almacenados en el mismo grupo que su inodo, asegurando, por tanto, una cierta cercanía. Asimismo, dentro de lo posible, los ficheros de un directorio estarán incluidos en su mismo grupo, excepto, habitualmente, si se trata de subdirectorios del directorio raíz, en cuyo caso se ubican en otro grupo para favorecer la distribución.

El término usado para denominar a este nuevo elemento es *grupo de cilindros*, ya que abarca un conjunto de cilindros contiguos del dispositivo (recuerde que un cilindro corresponde a la misma pista de todas las superficies: todos los sectores de un cilindro se pueden acceder sin mover las cabezas).

Con la incorporación de este nuevo concepto, es necesario extender cierta información que aparecía originalmente en el superbloque, ya que se necesita especificar para cada grupo de cilindros. Así, alguna información, como el tamaño del bloque y del inodo, se mantiene sin cambios ya que afecta a todo el dispositivo. Sin embargo, se requiere indicar la ubicación de los mapas de bits y de la tabla de inodos de cada grupo. Para incorporar esa extensión, se distingue entre el suberbloque, que mantiene la información global aplicable a todo el disco, y la tabla de descriptores de grupos, que almacena una entrada por cada grupo conteniendo la información específica de ese grupo. En consecuencia, la estructura resultante es la siguiente:

- Superbloque: 1 bloque que contiene información global.
- Tabla de descriptores de grupos: puede ocupar varios bloques; cada entrada guarda la información específica de un grupo, que incluye la ubicación de sus mapas de bits y de su tabla de inodos.
- Por cada grupo, habrá un mapa de bits de bloques, un mapa de bits de inodos y una tabla de inodos, así como, evidentemente, el conjunto de bloques de datos asignados a ese grupo.

Además de mejorar el rendimiento del sistema de ficheros, FFS también pretendía aumentar su fiabilidad. Para ello, plantea que esté replicado tanto el superbloque como la tabla de descriptores de grupos, pudiendo haber una copia de ambas estructuras almacenada en cada grupo.

El *Virtual File System* de SunOS

El modelo de negocio de la informática ha cambiado radicalmente en los últimos 40 años. En aquellos tiempos pretéritos, cuando una organización adquiría un equipo, quedaba fuertemente vinculada con la compañía que se lo vendía, puesto que esta era la encargada de suministrarle toda la infraestructura requerida: el hardware, el software de sistema, las herramientas y las aplicaciones.

Este modelo cerrado fue evolucionando y abriéndose a la libre competencia, de manera que actualmente, a la hora de crear la infraestructura de una organización, se pueden utilizar en los distintos niveles soluciones de diferentes fabricantes. Con esta evolución, los aspectos de compatibilidad y portabilidad son claves en el éxito de un determinado producto.

Linux es un ejemplo paradigmático de todo este proceso y, dentro de esta compatibilidad, está la necesidad de dar soporte a diversos tipos de sistemas de ficheros definidos por distintas organizaciones. El mecanismo que usa Linux para articular esta compatibilidad y posibilitar la convivencia de distintos tipos de sistemas de ficheros es el Sistema de Ficheros Virtual (*Virtual File System*).

Para entender el modo de operación del sistema de ficheros virtual, podemos remontarnos a uno de los sistemas precursores en el uso de esta técnica: el sistema operativo SunOS, cuya denominación cambió con el tiempo a Solaris.

En este sistema operativo se enfrentaron con el reto de incorporar la funcionalidad del cliente NFS (la empresa *Sun Microsystems* creó el protocolo NFS) de manera que pudiera convivir con el sistema de ficheros nativo permitiendo que las aplicaciones pudiesen acceder de forma transparente tanto a los ficheros locales como a los remotos. En este proceso de integración detectaron que, aunque cada tipo de sistema de ficheros era radicalmente diferente, existía una cierta funcionalidad de alto nivel que era común a ambos. Antes esta circunstancia, rediseñaron el sistema de ficheros creando dos niveles.

En el nivel superior se ubica el sistema de ficheros virtual que proporciona servicios a las aplicaciones. Este nivel implementa la funcionalidad genérica común a todos los tipos de sistemas de ficheros.

En el nivel inferior se encuentran los distintos tipos de ficheros existentes en el sistema. Cuando el nivel VFS realiza una operación sobre un fichero, lleva a cabo toda la parte del procesamiento genérica y, a continuación, invoca la función correspondiente del sistema de fichero específico.

En el nivel superior se trabaja con un inodo genérico (el vnodo), que contiene la información genérica común y que tiene asociado un descriptor de fichero específico: un inodo en el caso del sistema de ficheros nativo y un rnodo para los ficheros remotos accesibles a través de NFS.

El uso generalizado de este mecanismo en sistemas como Linux ha permitido dar soporte de forma transparente e integrada a una gran cantidad de tipos de sistemas de ficheros.

El sistema de ficheros *ext2*

Como se comentó al principio del documento, en sus inicios, Linux tomó prestado el sistema de ficheros de MINIX como sistema de ficheros nativo. Sin embargo, este presentaba numerosas limitaciones que rápidamente llevaron al desarrollo de un nuevo sistema de ficheros para superarlas, denominado *Extended* (*ext*) por ser una extensión del de MINIX. Este primer producto, que ya no está soportado actualmente, evolucionó en muy poco tiempo hacia una segunda versión, llamada *ext2*, que completaba ese proceso de eliminar las trabas del sistema de ficheros MINIX, ofreciendo un notable grado de eficiencia y de robustez, que le han convertido en el sistema de ficheros de referencia en el mundo Linux.

En esta sección vamos a revisar algunas características de este sistema de ficheros, planteando algunos ejercicios prácticos para reforzar los conceptos presentados.

Uso del sistema de ficheros virtual

Como se comentó previamente, Linux implementa el mecanismo VFS para dar soporte a distintos tipos de ficheros. El soporte de un determinado tipo de sistema de ficheros puede configurarse en tiempo de compilación del sistema (el fichero */boot/config-$(uname -r)* contiene las opciones seleccionadas durante la generación de la imagen del sistema operativo activo actualmente en este equipo) para estar incluido en el

núcleo (por ejemplo, *CONFIG_VFAT_FS=y*) o como un módulo que se carga posteriormente (por ejemplo, *CONFIG_NFS_V4=m*), estando en ese último caso almacenado en */lib/modules/$(uname -r)/kernel/fs/*/*.ko*.

Cuando se ejecuta el código inicial de un manejador de un tipo de sistema de ficheros, sea en la iniciación del sistema, si forma parte del núcleo, o cuando se carga el módulo, en caso de estar configurado de esa forma, tiene que registrarse (función *register_filesystem*) especificando cuál es su operación de montaje específica. En el momento del montaje de un dispositivo con un sistema de ficheros de este tipo se ejecuta esa operación de montaje específica que ya registra sus operaciones propias que serán invocadas por el VFS para llevar a cabo la parte específica de cada servicio.

Hay que resaltar que, además de sistemas de ficheros destinados a almacenar información en un disco, Linux gestiona varios *seudo-sistemas de ficheros* que no están ligados a un dispositivo, sino que están ideados para implementar una cierta funcionalidad. Por ejemplo, el sistema de ficheros *proc* ofrece acceso a estructuras de datos internas del núcleo.

Límites del sistema de ficheros

Todo sistema de ficheros establece ciertas limitaciones en diversos parámetros de su funcionamiento interno. De hecho, una de las razones que impulsaron el desarrollo de *ext*, y posteriormente de *ext2*, fue mejorar significativamente algunas de las limitaciones que presentaba el sistema de ficheros de MINIX.

En esta sección nos vamos a centrar en dos de estos límites: el máximo tamaño de un sistema de ficheros y el máximo tamaño de un fichero.

El máximo tamaño del sistema de ficheros en *ext2* viene condicionado por el tamaño que se dedica a los números de bloque del disco (es decir, el tamaño de un puntero a un bloque), que es de 32 bits en este sistema, y, evidentemente, por el tamaño del bloque.

El máximo tamaño de un fichero en *ext2* viene condicionado por tres factores, siendo el mínimo entre:

- La resolución máxima del campo del inodo que guarda el tamaño del fichero en bytes, que en la primera versión de *ext2* ocupaba 32 bits, pero que posteriormente se extendió hasta 64 bits.
- El rango de cobertura del inodo con sus 15 punteros: 12 directos, un indirecto simple, uno doble y uno triple.
- En el inodo se almacena en un campo de 32 bits cuántos bloques de 512 bytes (el tamaño típico de un sector de disco) tiene realmente asignados el fichero, con independencia de cuál sea el tamaño de bloque que usa el sistema de ficheros. Nótese que puede no haber una correspondencia directa entre el tamaño en bytes del fichero y los bloques que ocupa. Así, como veremos más adelante, puede haber, por ejemplo, un fichero de tamaño 1GiB que no tenga asignado ningún bloque.

Ejercicio: dimensiones del sistema de ficheros *ext2*

La siguiente tabla muestra el valor de los dos límites identificados previamente para distintos tamaños de bloque. Téngase en cuenta que el tamaño de bloque debe ser potencia de 2, mayor o igual que el tamaño del sector del disco y menor o igual que el tamaño de la página.

Tamaño de bloque	1KiB	2KiB	4KiB
Tamaño máximo del sistema de ficheros	4TiB	8TiB	16TiB
Tamaño máximo del fichero	16GiB	256GiB	2TiB

Cuestión 1

Realice los cálculos para obtener esos 6 valores. A la hora de calcular el rango de cobertura del inodo se recomienda solo considerar la proporcionada por el puntero indirecto triple, puesto que la ofrecida por el resto de los punteros puede considerarse despreciable frente a ese valor.

Estructura del sistema de ficheros

Como se comentó previamente, *ext2* está implementado usando el diseño propuesto por el *Fast File System* de BSD, existiendo un pequeño cambio en la terminología: en vez de grupos de cilindros se usa el término grupo de bloques. Para entender el motivo de ese cambio hay que tener en cuenta que en los tiempos del FFS los discos se accedían especificando el número de cilindro, la superficie en ese cilindro y el sector dentro de esta (direccionamiento CHS: *Cylinder, Head, Sector*). Posteriormente, surgió el direccionamiento LBA (*Logical Block Addressing*), que ofrece una visión del dispositivo como un vector de bloques, ocultando la geometría del mismo. Por tanto, parece más razonable hablar de grupos de bloques que de grupos de cilindros.

De la misma manera que en FFS, en *ext2* el disco está organizado como un conjunto de grupos de bloques. Cada grupo de bloques se caracteriza porque tiene asignado un único bloque tanto al mapa de bits de bloques como al de inodos. Por tanto, suponiendo un tamaño de bloque de 4KiB, el número máximo de bloques y de inodos que puede contener un grupo de bloques es 32KiB, es decir, una capacidad de almacenamiento de 128MiB pudiendo tener hasta 32768 ficheros. Recuerde que en los sistemas de ficheros UNIX el número de inodos usados es un parámetro que debe especificar el usuario que crea el sistema de ficheros puesto que depende del uso que se vaya a hacer del disco (se especifica mediante la opción *-i* del mandato *mkfs.ext2*).

Cuando se crea el sistema de ficheros, se determina el número de grupos de bloques requerido teniendo en cuenta el tamaño del dispositivo y el tamaño máximo gestionado por un grupo de bloques, creándose tantas entradas en la tabla de descriptores de grupos de bloques como grupos se requieren. Cada entrada de esa tabla ocupa 32 bytes y se asignarán tantos bloques a la tabla como sean necesarios para almacenar las entradas requeridas. Nótese que, si el número de inodos especificados en la creación del sistema de ficheros es mayor que el número de bloques, ese, y no el número de bloques, sería el factor que determinaría cuántos grupos de bloques son necesarios. Sin embargo, habitualmente, el número de inodos es un divisor del número de bloques (en la opción *-i* se indica un número de bytes por inodo mayor o igual que el tamaño del bloque).

Como sucede con FFS, por fiabilidad, se almacenan copias del superbloque y de la tabla de descriptores de grupos de bloques, pudiendo haber una copia de ambas estructuras en cada grupo de bloques o solo en ciertos grupos prefijados (opción *sparse_super* a la hora de crear el sistema de ficheros).

Con respecto a la copia primaria del superbloque, se almacena a partir del byte 1024 del dispositivo. Recuerde que los primeros 512 bytes están reservados para el sector de arranque, incluso aunque el dispositivo no esté configurado para tal fin. Nótese que, dado que el superbloque ocupa 1024 bytes, si el tamaño del bloque del sistema de ficheros es 1024, el superbloque corresponde al segundo bloque (bloque 1), mientras que, si es 4096, el superbloque está almacenado dentro del primer bloque (bloque 0), a partir del byte 1024, quedándose sin usar la segunda mitad de ese bloque.

La estructura resultante es similar a la del FFS y corresponde a una colección de grupos de bloques tal que, suponiendo que hay copias del superbloque y de la tabla de descriptores de grupos en todos los grupos, cada grupo tiene la siguiente organización:

- Superbloque: Esta zona mantiene información global del sistema de ficheros como el tamaño del bloque y del inodo, el número total de bloques e inodos en el dispositivo y por cada grupo.
- Tabla de descriptores de grupos: puede ocupar varios bloques; cada entrada guarda la información específica de un grupo, tanto información estática, como la ubicación de sus mapas de bits y de su tabla de inodos, como dinámica, que sería el caso del número de bloques e inodos libres en ese grupo. Observe que, en caso de estar presentes, la ubicación en el grupo del superbloque y de la tabla de descriptores está prefijada (están situados en los primeros bloques del grupo), pero la de los mapas y la tabla de inodos puede variar al estar especificada en el propio descriptor del grupo.
- Por cada grupo, habrá un mapa de bits de bloques, un mapa de bits de inodos y una tabla de inodos, así como, evidentemente, el conjunto de bloques de datos asignados a ese grupo:

- El mapa de bits de bloques representa el estado de todos los bloques del grupo, incluidos aquellos que contienen el superbloque, los mapas de bits y la tabla de inodos. Por tanto, en el mapa de bits de un sistema de ficheros recién creado ya existen bloques en el mapa de bits de bloques ocupados.
- El mapa de bits de inodos también se inicia con algunos inodos reservados que se usan para ciertas funcionalidades. Así, el inodo 1 se usa para guardar una lista de bloques defectuosos. Por su parte, el directorio raíz tiene asignado el inodo 2. En el proceso de creación del sistema de ficheros se crea el directorio *lost+found* que usará el primer inodo libre, que, habitualmente, es el 11.
- En la zona de bloques de datos, además de los bloques de los datos de los ficheros y directorios, se almacenan los bloques indirectos y los que contienen atributos extendidos.

Hay que resaltar que las actualizaciones de la información tanto del superbloque como de la tabla de descriptores de grupos se realizan solo sobre la copia primaria de ambos, por lo que las copias secundarias no siempre mantendrán la información dinámica actualizada.

En los siguientes ejercicios, se va a analizar el modo de operación del sistema de ficheros *ext2* revisando sus estructuras internas y cómo estas evolucionan según se van realizando distintas operaciones en el sistema de ficheros. Para realizar este análisis, se van a usar las siguientes herramientas:

- *dumpe2fs*: muestra información del sistema de ficheros, imprimiendo datos del superbloque, de la tabla de descriptores de grupos y del estado de los mapas de bloques e inodos libres de cada grupo. Esta herramienta imprime la información contenida en las copias primarias del superbloque y de la tabla de descriptores de grupo. Se puede ver la información de una copia secundaria mediante la opción *-o superblock*.
- *debugfs*: permite conocer y modificar el estado de un sistema de ficheros. En nuestro caso, solo lo utilizaremos para consultar la información del sistema de ficheros, usando operaciones tales como:
    - Ver el contenido de un bloque (mandato *bd*)
    - Hay operaciones similares a los disponibles en la línea de mandatos del *shell*, pero que, normalmente, imprimen información interna adicional, tales como *stat*, *ls* o *cat*.
    - Mostrar el estado del mapa de bits de bloques y de inodos (*testb* y *testi*, respectivamente).
    - Comprobar a qué inodo pertenece un bloque (*icheck*).
    - Nótese que, en los mandatos internos de *debugfs* que reciben ficheros como argumentos, estos se pueden especificar usando el nombre o el número de inodo encerrado entre <>.
    - Así, por ejemplo, si quiere ver el contenido del inodo 12 asociado al fichero *FICHERO*, puede usar cualquiera de estas dos opciones (*debugfs:* es el *prompt* del mandato *debugfs*):

        debugfs miDisco2
        debugfs: stat <12>
        debugfs: stat FICHERO

- *DIY*: no se trata de ninguna herramienta; solo se pretende resaltar que podemos acceder directamente al dispositivo para conocer la información contenida en las estructuras del sistema de ficheros. Así, por ejemplo, este mandato accede al bloque 3 de un dispositivo que usa bloques de 4KiB y muestra en hexadecimal la información que contiene (sería equivalente al mandato de *debugfs*: *bd 3*):

    dd if=miDisco2 bs=4K count=1 skip=3 2>/dev/null | od -x

- *cmpbl* y *cphuecos*: Después de cada operación, consultaremos el estado del sistema de ficheros usando las herramientas previas para analizar qué cambios han sucedido. Sin embargo, puede ser

conveniente previamente conocer qué bloques del sistema de ficheros se han modificado durante la operación para centrarse en esos bloques. Por tanto, en el análisis de la evolución de la información del sistema de ficheros se plantea el siguiente ciclo: sacar una copia del dispositivo, realizar la operación y comparar el nuevo estado del dispositivo con su copia identificando qué bloques se han actualizado, que se analizarán con las herramientas pertinentes. Para ello, se proporcionan como material de apoyo dos programas: *cphuecos*, que realiza la copia, pero sin gastar espacio para los *huecos* que existen en el dispositivo como sí lo haría un *cp* convencional, y *cmpbl*, que lleva a cabo la comparación detectando qué bloques han cambiado. Nótese que ambos programas están diseñados asumiendo un tamaño de bloque de 4KiB, que es el que se usará en este ejercicio.

AVISO IMPORTANTE

Tenga en cuenta que estas herramientas muestran la información del sistema de ficheros almacenada en el dispositivo. Debido al uso intensivo de la caché que realiza un sistema de ficheros, habrá que esperar un cierto tiempo (hasta medio minuto) desde que se realiza una operación (como, por ejemplo, crear un nuevo fichero) hasta que los cambios en las estructuras pertinentes del sistema de ficheros queden reflejados en el disco. Nótese que, en algunas operaciones, ciertas acciones se actualizan en el disco inmediatamente, pero otras tienen que esperar a que llegue la operación periódica de volcado al dispositivo de la información en memoria de los sistemas de ficheros. Se ruega un poco de paciencia a la hora de afrontar este ejercicio.

Por esas mismas razones, en el caso del mandato *debugfs*, será necesario volver a arrancarlo cada vez que se realiza una operación.

Ejercicio: creación del sistema de ficheros

Comencemos el ejercicio, en el que, por flexibilidad y ahorro en el consumo de espacio, en vez de un disco real, usaremos un fichero que opere como tal (lo que se denomina un *loopback device*). A continuación, se especifican los pasos requeridos para acometer este ejercicio.

- Creamos un fichero de 512MiB que actuará como nuestro disco. Nótese que no hay que preocuparse por el gasto de espacio en el disco real ya que, como se analizará al final del ejercicio, ese fichero está inicialmente vacío, como se puede comprobar con el mandato *stat* (por curiosidad, puede comprobar cuántos bloques tiene asignado ese fichero al final de este ejercicio):

  truncate -s 512M miDisco2 # creo un fichero de 512MiB que no ocupa espacio

- Nótese que también estaría vacía una copia que hagamos del *dispositivo* usando el programa de copia proporcionado. Vamos a realizar una antes de crear el sistema de ficheros para poder comprobar qué bloques se modifican durante esa operación:

  ./cphuecos miDisco2 back

- Creamos un sistema de ficheros de tipo *ext2* en ese "dispositivo". Se van a eliminar dos de las opciones por defecto de este mandato. Por un lado, se desactiva la opción *sparse_super*, que especifica que no se haga una copia del superbloque y de la tabla de descriptores de grupos de bloques en cada grupo de bloques, sino solo en algunos (grupos 0, 1, 3, 5, 7…). Al eliminar esta opción, sí habrá una copia en cada grupo, que era el comportamiento por defecto del FFS. Por otro lado, se inhabilita la opción *resize_inode*, que reserva bloques para almacenar descriptores de grupos de bloques adicionales de manera que el sistema de ficheros pueda crecer *a posteriori* hasta 1024 veces su tamaño original (esta característica se añadió en *ext3* pero se incorporó retroactivamente en *ext2*), que se analizará más adelante. Observe que se ha optado por un tamaño del inodo de 128 puesto que es el que se usaba originalmente en *ext2*.

  mkfs.ext2 -i 8192 -b 4096 -I 128 -O "^sparse_super,^resize_inode" miDisco2

- Por último, obtenemos información sobre el estado inicial del dispositivo

    dumpe2fs miDisco2
    ./cmpbl miDisco2 back
    # acceso a los mapas de bits de bloques y de inodos del grupo 0 usando el mandato bd de debugfs o
    # leyendo directamente el fichero (usando, por ejemplo, un mandato dd como el que se explicó antes)

Cuestión 2

Analice la información obtenida en el último paso relacionando los datos mostrados por *dumpe2fs* con la lista de bloques cambiados y los mapas de bits obtenidos.

Ejercicio: operaciones sobre ficheros

Este ejercicio revisa cómo repercute en la información del sistema de ficheros almacenada en el dispositivo la creación de un fichero. Antes de iniciar cada paso del ejercicio, recuerde sacar una copia del mismo.

- Para poder trabajar con el sistema de ficheros, en primer lugar, hay que montarlo:

    sudo mount miDisco2 /mnt

Cuestión 3

Explique qué cambios en la información del sistema de ficheros almacenada en el disco se han producido tras el montaje. Para comprobar los cambios en el superbloque puede comparar el resultado de *dumpe2fs* de la copia primaria y de una secundaria (recuerde que las actualizaciones se realizan directamente sobre la copia primaria).

- En este punto, vamos a comentar un aspecto interno que, en principio, es transparente, pero que saldrá a la luz más adelante, cuando analicemos el redimensionamiento de un sistema de ficheros. Como se ha podido apreciar en ese mandato de montaje, Linux permite usar un fichero en vez de un dispositivo a la hora de especificar el recurso que hay que montar. Para implementar esta funcionalidad, crea automáticamente un dispositivo de bloques de tipo bucle (*loop*) asociado al fichero en la operación de montaje.

    losetup -l | grep miDisco2

- Recuerde que cuando no quiera usar más ese dispositivo deberá desmontarlo:

    sudo umount /mnt

- Si quiere evitar tener que trabajar como superusuario, puede dar permisos generalizados de acceso al directorio raíz del dispositivo (obviamente, si opta por realizar esta operación, esta se verá reflejada también en el disco y, si no quiere que afecte a los resultados de la siguiente operación, deberá esperar el medio minuto de rigor antes de dar el siguiente paso):

    sudo chmod 777 /mnt # por comodidad, se permite que cualquiera pueda escribir

- A continuación, creamos un fichero vacío (antes, como en cada ejercicio, sacaremos copia del estado del dispositivo):

    touch /mnt/FICHERO

Cuestión 4

Explique de forma razonada qué cambios en la información del sistema de ficheros almacenada en el disco se han producido tras esa operación. Recuerde que puede tener que esperar hasta medio minuto hasta que los cambios se actualicen en el disco.

- Por último, vamos a escribir un poco más de 4MiB de datos aleatorios en el fichero:

  dd if=/dev/random of=/mnt/FICHERO bs=4K count=1040

Cuestión 5

Explique de forma razonada qué cambios en la información del sistema de ficheros almacenada en el disco se han producido tras esa operación. Analice especialmente la metainformación de ubicación de los bloques del fichero usando el mandato *stat* de *debugfs*.

Directorios

Una de las limitaciones del sistema de ficheros de MINIX es que el nombre de un fichero está restringido a una longitud de 14 caracteres. Como se comentó en la parte inicial del documento, esa limitación facilita mucho la gestión de los directorios porque se puede usar un tamaño fijo para cada entrada del directorio dedicando 14 bytes al nombre del fichero, aunque realmente ocupe menos (la entrada "." solo ocupa 1 carácter), ya que el desperdicio es tolerable.

El sistema de ficheros *ext2* permite nombres de 255 caracteres lo que, para evitar malgastar el espacio en disco, conlleva que el tamaño de la entrada sea variable. Cada entrada de un directorio está formada por una parte de tamaño fijo que ocupa 8 bytes y el nombre del fichero que se extiende con bytes a 0 al final para que ocupe un espacio múltiplo de 4. A continuación, se especifica el formato de la parte de tamaño fijo:

- El número de inodo asociado a la entrada, que ocupa 4 bytes.
- Los dos siguientes bytes indican la longitud total que ocupa la entrada, incluyendo todos los campos.
- El próximo byte indica la longitud del nombre.
- El último byte incluye información de a qué tipo de fichero corresponde la entrada (1 fichero normal, 2 directorio…). La presencia de este dato rompe el esquema estándar de UNIX al incluir información del fichero en la entrada, pero agiliza ciertas operaciones (por ejemplo, permite obtener la lista de los nombres de los subdirectorios de un directorio sin leer los inodos de las entradas del directorio), aunque su presencia o ausencia puede configurarse a la hora de crear el sistema de ficheros (opción *filetype*), estando presente por defecto.

Hay que resaltar que, cuando se elimina una entrada, el espacio ocupado por la misma pasa a formar parte de la entrada anterior.

Ejercicio: operaciones sobre directorios

Este ejercicio revisa cómo repercute en la información del sistema de ficheros almacenada en el dispositivo la creación de un directorio y la creación y borrado de ficheros de un directorio.

- Cree un directorio en el sistema de ficheros.

  mkdir /mnt/DIRECTORIO

Cuestión 6

Explique de forma razonada qué cambios en la información del sistema de ficheros almacenada en el disco se han producido tras esa operación.

- A continuación, creamos un segundo fichero:

  touch /mnt/FICHERO2

Cuestión 7

Analice el contenido del directorio raíz del disco. Para ello, suponiendo que el bloque de datos del directorio es *X*, puede usar o bien el mandato *bd* de *debugfs* o bien acceder directamente al dispositivo usando "*dd*

if=miDisco2 bs=4096 skip=X count=1 2> /dev/null | od -cx". **Interprete los valores correspondientes a la primera entrada (".").**

- A continuación, se pretende analizar qué ocurre cuando se borra una entrada. Para ello, ejecute la siguiente secuencia y compare los ficheros *ANTES* y *DESPUES* para detectar qué ha cambiado en el bloque de datos del directorio por el borrado del fichero:

  dd if=miDisco2 bs=4096 skip=X count=1 2> /dev/null | od -cx > ANTES
  rm /mnt/FICHERO
  sleep 30 # esperemos un rato hasta que la metainformación se actualice en el disco
  dd if=miDisco2 bs=4096 skip=X count=1 2> /dev/null | od –cx > DESPUES
  diff ANTES DESPUES

Cuestión 8

Explique los datos obtenidos en el punto anterior.

Atributos de los ficheros

Como ya se ha explicado al principio del documento, el inodo almacena la metainformación del fichero. El estándar UNIX define cuál debe ser esa información y especifica el servicio y el mandato *stat* para acceder a la misma, como hemos visto en ejemplos previos.

Además de esta información estandarizada, el sistema de ficheros *ext2* gestiona una colección de atributos adicionales propios que se pueden consultar y modificar mediante los mandatos *lsattr* y *chattr*, respectivamente. Nótese que estos atributos no aparecen en la información que muestra el mandato estándar *stat*, mostrándose como *flags* si se usa el mandato *stat* de *debugfs*.

Ejercicio: atributos específicos de *ext2*

Escoja uno de los atributos específicos de *ext2* y añádaselo al fichero vacío *FICHERO2*.

Cuestión 9

Especifique el detalle del mandato *chattr* usado.

Como también se explicó al principio del documento, es deseable que exista un mecanismo que permita definir nuevos atributos: en *ext2*, los atributos extendidos. Un atributo extendido consiste en un nombre y un valor que se asocia con un fichero. Para fijar un atributo se usa el mandato *setfattr*, mientras que el mandato *getfattr* permite obtenerlo.

La implementación de esta funcionalidad en *ext2* solo dedica un bloque de datos para almacenar los atributos extendidos de un fichero, lo que establece una limitación sobre el uso de los mismos.

Estos atributos extendidos pueden usarse para guardar información del sistema que no estaba prevista cuando se diseñó el sistema de ficheros, como veremos en el tema de protección y seguridad. Asimismo, permite a un usuario definir sus propios atributos. Para evitar colisiones de nombres, existen distintos espacios de nombres predefinidos (*man xattr*). Todos los atributos definidos por el usuario deben comenzar por el prefijo "*user.*" produciéndose un error si no es así.

Ejercicio: atributos extendidos

A continuación, proponga un ejemplo de uso de atributos extendidos definidos por el usuario y ejecute el mandato para añadírselo a *FICHERO2*.

Cuestión 10

Especifique el detalle del mandato *setfattr* utilizado. Recuerde la necesidad de indicar el espacio de nombres de usuario (*user.*) como prefijo del nombre del atributo. **Analice qué ha cambiado en el sistema de ficheros**

durante esa operación. Nótese que, si se hubiera optado por un inodo de 256 bytes, esa información, siempre que sea de pequeño tamaño, se habría almacenado en el propio inodo.

El sistema de ficheros ext3

Esta nueva versión del sistema de ficheros nativo de Linux surgió por la necesidad de incorporar un mecanismo de *journaling* manteniendo la compatibilidad con *ext2*. Además de este mecanismo, en esta sección se analizarán otras dos funcionalidades de *ext3*: los directorios indexados y el redimensionamiento de un sistema de ficheros. Nótese que, por compatibilidad, esas dos últimas funcionalidades están incorporadas de forma retroactiva en *ext2* (*ext2* es, básicamente, un *ext3* sin *journaling*).

Aspectos de fiabilidad del sistema de ficheros

Como se comentó en la parte inicial del documento, una operación del sistema de ficheros implica normalmente una secuencia de accesos que modifican la información almacenada en el disco. Si el sistema se cae en medio de esta secuencia, la información del disco puede quedar corrupta. Cuando se intenta usar un dispositivo después de que un equipo se haya caído y haya vuelto a arrancar, se detecta al leer el superbloque que el disco no se desmontó correctamente (cuando se completa una operación de desmontaje, se actualiza el superbloque para reflejarlo), siendo necesario realizar un chequeo y, en caso de que sea necesaria, una reparación del dispositivo (mandato *fsck*).

La operación de chequeo debe leer toda la metainformación del disco, que incluye todos los bloques de datos de los directorios y todos los bloques indirectos para realizar comprobaciones como las siguientes:

- Si un bloque está asignado a un fichero, no debería estar asignado a ningún otro fichero ni debería aparecer como libre en el mapa de bits de bloques.
- Si una entrada de directorio hace referencia a un determinado inodo, ese inodo debería aparecer como ocupado en el mapa de bits de bloques.
- El número de enlaces almacenado en un inodo debe coincidir con el número de entradas de directorio que hacen referencia a ese inodo.

Cuando encuentra una de estas situaciones patológicas, el programa de chequeo debe solventarla modificando convenientemente el contenido del disco (por ejemplo, marcando como ocupado un inodo que está referenciado desde una entrada de directorio, pero aparece como libre en el mapa de bits de inodos). El objetivo es conseguir que la operación del sistema de ficheros interrumpida en mitad de su labor tenga un comportamiento atómico: el programa de chequeo debería realizar acciones correctivas de manera que o bien la operación se completa o se eliminan todos los vestigios de la misma.

Para intentar reducir las posibles inconsistencias en el estado del sistema de ficheros cuando un equipo se cae bruscamente, el diseño del sistema de ficheros debe estudiar minuciosamente cuál es el orden más adecuado a la hora de realizar las diversas acciones que conlleva cada operación del sistema de ficheros.

Journaling

Con el progresivo y significativo crecimiento en el tamaño de los discos, la operación convencional de chequeo sobre un dispositivo de gran volumen es muy lenta. Para paliar este problema se ideó la técnica del *journaling*. Con esta técnica, el sistema de ficheros va almacenando una bitácora (*journal*) en una zona del disco, aunque también puede usarse un dispositivo externo, de manera que al reiniciar un equipo después de una caída solo es necesario acceder a esa bitácora para chequear y reparar, en caso de que sea necesario, el dispositivo.

El objetivo de este mecanismo es convertir en atómicas las operaciones del sistema de ficheros que, como se comentó previamente, no lo son, ya que están compuestas por una serie escrituras sobre el dispositivo, de manera que, ante cualquier contingencia, pueda completarse la operación o descartarse, como ocurre con una escritura elemental, pero evitando que queden estados intermedios incoherentes.

A grandes rasgos y de forma simplificada, a continuación, se describe el modo de operación del *journaling* en *ext3* (denominado JBD2):

- Para lograr la atomicidad, se van a ejecutar todas las acciones asociadas a una operación en el contexto de una transacción, de manera que todas las actualizaciones que se realizan durante la operación no modifican los bloques originales, sino que esos bloques modificados se escriben consecutivamente en el *journal*. Al principio de cada transacción, antes de los bloques actualizados, se escribe un bloque descriptor que indica a qué bloques originales del disco corresponden los bloques que aparecen a continuación en la transacción. Así, por ejemplo, si una transacción tiene que modificar los bloques 8, 45 y 1000 del sistema de ficheros (que quizás correspondan a un bloque del mapa de inodos, a un inodo y al bloque de datos de un directorio, respectivamente) y la transacción comienza en el bloque 10 del *journal*, en ese bloque 10 se almacenará el bloque descriptor que indicará que los tres siguientes bloques del *journal* (11, 12 y 13) corresponden, respectivamente, a los bloques 8, 45 y 1000 del dispositivo.
- Cuando todos los bloques de la transacción se han escrito a disco, se escribe en el *journal* (en el ejemplo, en el bloque 14) un bloque de *commit*, que indica que la operación se considera comprometida, es decir, que pase lo que pase la operación se completará.
- Una vez escrito el bloque de *commit*, se van copiando los bloques de la transacción del *journal* a los originales. Cuando finaliza esa acción, el espacio ocupado por la transacción en el *journal* puede reutilizarse, puesto que la operación ya se ha completado.
- Por defecto, el *journal* se almacena como un fichero sin nombre en el propio dispositivo, normalmente, con el inodo 8. El espacio asociado a ese fichero se gestiona como un *buffer* circular, guardándose las referencias a cuáles son la primera y última transacción del *journal*.
- Al reiniciarse una máquina después de una caída, el programa de chequeo solo debe leer el *journal* descartando las transacciones que no tengan *commit* y, para aquellas que sí lo tengan, copiando los bloques contenidos en la transacción en los originales en caso de que difieran.

En *ext3* se pueden configurar en la operación de montaje tres modalidades de *journaling* dependiendo de cómo se manejen las escrituras de los datos (el tratamiento de las escrituras de los metadatos es igual en las tres modalidades). A continuación, se presentan en orden de fiabilidad decreciente, pero rendimiento creciente:

- Modalidad *journal*: En el *journal* se almacenan tanto los datos como los metadatos actualizados por la operación (*journaling* de datos y de metadatos). Es la modalidad con más sobrecarga, ya que hay que realizar dos escrituras de los datos, y con más gasto de espacio, puesto que se guardan temporalmente dos copias de los datos. A cambio, proporciona la mejor fiabilidad. Así, en un reinicio después de una caída, si la transacción tiene *commit*, además de los metadatos, se copian los bloques de datos desde el *journal* hasta su ubicación original y, si no lo tiene, al descartar la transacción, se eliminan también los datos.
- Modalidad *ordered* (por defecto): Solo hay *journaling* de metadatos, pero se asegura un cierto orden en la escritura de los datos. En esta modalidad, los datos se escriben directamente en los bloques del sistema de ficheros que les corresponden, no incluyéndose en el *journal*, pero hay que asegurar que ya están actualizados en el dispositivo antes de que se escriba el *commit* en el *journal*. En consecuencia, si una transacción ha podido escribir el bloque de *commit* antes de la caída de la máquina, el comportamiento es similar a la modalidad anterior. Sin embargo, no es así en el caso de que no se haya podido escribir el bloque de *commit*, ya que, en un reinicio después de una caída, en una transacción sin *commit* no podemos descartar los cambios que se hayan podido hacer sobre los datos ya que se actualizan directamente sobre los bloques originales. Analicemos cómo afectaría esta circunstancia a una escritura sobre un archivo que ha escrito los metadatos en el *journal* y los datos en los bloques asignados al fichero, pero que no ha podido realizar la escritura del bloque de *commit* en el *journal* porque se ha caído la máquina:

    - Si se trata de una escritura que extiende el fichero, al reiniciarse el equipo, se descarta la transacción con lo que todas las actualizaciones sobre metadatos desaparecen, mientras que las modificaciones sobre los datos se mantienen, pero no se tienen en cuenta ya que se han realizado sobre bloques marcados como libres. Por tanto, se cumple el requisito de que se eliminan totalmente los efectos de una transacción incompleta al reiniciarse el equipo.
    - En caso de una escritura que sobrescribe parte del fichero, nuevamente, desaparecen los cambios en los metadatos al volver a arrancarse la máquina, pero, en esta ocasión, las actualizaciones sobre los datos sí permanecen. Se produce una situación un tanto anómala en el sentido de que quedan como permanentes los cambios en los datos que ha realizado una operación que finalmente se considera que no existió.
- Modalidad *writeback*: Solo hay *journaling* de metadatos, pero, en este caso, no se asegura de que los datos, aunque sí los metadatos, se escriban en el disco antes de que se escriba el *commit* en el *journal*. Tenga en cuenta que no forzar un orden entre las escrituras de los datos en el dispositivo y la del registro *commit* en el *journal* permite obtener un mejor rendimiento en las operaciones sobre el dispositivo. Sin embargo, en este caso el comportamiento difiere con respecto a las dos modalidades previas incluso aunque haya dado tiempo a escribir el bloque de *commit* antes de la caída, ya que no podemos asegurar que los datos estén ya actualizados. Así, si se trata de una operación de escritura que hace crecer al fichero, los datos contenidos en la zona expandida pueden no corresponder a los que se pretendía escribir, puesto que se cayó la máquina antes de completar su escritura. Por tanto, nos podemos encontrar con un fichero tal que parte de su contenido es impredecible (nótese que en el inodo los nuevos bloques se conectaron al fichero, pero no dio tiempo a actualizarlos), lo que puede causar incluso problemas de seguridad.

Para terminar, se considera conveniente resaltar los siguientes aspectos sobre el *journaling*:

- Téngase en cuenta que también se consideran como metadatos los bloques indirectos y los bloques de datos de los directorios.
- Realmente, por eficiencia, en una transacción se pueden agrupar las acciones asociadas a múltiples operaciones independientes.
- El uso de *journaling* de datos no debe considerarse como una especie de copia de seguridad de los datos. El objetivo de esta técnica es mejorar el comportamiento del dispositivo ante una caída, pero no es un mecanismo de replicación como tal, puesto que una vez completada la transacción puede reutilizarse en cualquier momento el espacio ocupado por los datos de la misma en el *journal*.

Para poder ver el contenido del *journal*, puede usar el mandato *logdump* de *debugfs* que muestra los bloques de descriptores y de *commit* de cada transacción. Recuerde que un bloque de descriptores de una transacción guarda la correspondencia entre los bloques que aparecen justo a continuación en el *journal* formando parte de esa transacción (es decir, hasta el bloque de *commit*) y los bloques del sistema de ficheros vinculados con los mismos (el mandato *logdump -a* imprime el contenido de los bloques de descriptores mostrando esa correspondencia).

Ejercicio: *journaling*

Se van a repetir algunos de los ejercicios previos usando *journaling*:

- Cree el fichero *miDisco3* (el 3 es, obviamente, por *ext3*) con las mismas características de *miDisco2*.
- Cree el sistema de ficheros en el dispositivo con las mismas características pero usando *mkfs.ext3*. Por tanto, el sistema de ficheros usará *journaling*.
- Realice la operación de montaje de la misma manera que en ese ejercicio previo.

- Se plantean las tres mismas operaciones que en esos ejercicios anteriores: crear un fichero vacío (*touch /mnt/FICHERO*), escribir información aleatoria en el mismo (*dd if=/dev/random of=/mnt/FICHERO bs=4K count=1040*) y crear un directorio (*mkdir /mnt/DIRECTORIO*). Recuerde sacar una copia del dispositivo antes de cada operación y esperar un tiempo prudencial (hasta medio minuto) para asegurarse de que se consolidan las actualizaciones en el disco antes de analizar cada operación y pasar a la siguiente.

Cuestión 11

Para cada una de las tres operaciones, compare los resultados con los obtenidos con y sin *journaling* revisando para ello qué bloques de datos se han modificado en el dispositivo y qué datos muestra el mandato "logdump -a". Realice repetidamente la operación que comprueba los cambios en el dispositivo para que pueda apreciar como al usar la técnica de *journaling* las actualizaciones en el dispositivo se realizan en dos tandas.

- Realice el montaje indicando que se realice también *journaling* de datos.

  sudo mount -o data=journal miDisco3 /mnt

- Repita las tres mismas operaciones.

Cuestión 12

Compare los resultados con los obtenidos previamente con el *journaling* de tipo *ordered*, que es la opción por defecto.

Directorios indexados

Como se analizó al principio del documento, un requisito que debería satisfacer un sistema de ficheros es el manejo eficiente de directorios que tienen un número elevado de entradas. Supongamos, por ejemplo, el directorio donde un navegador almacena una caché de páginas con una entrada por cada página.

Con la implementación convencional de un directorio, la resolución (*lookup*) de un nombre que atraviesa un directorio muy grande es ineficiente ya que tiene que buscar secuencialmente todas las entradas del directorio (*O(n)*). Si revisamos este problema desde el punto de vista del diseño de estructuras de datos, se trata de una estructura que se accede mediante una clave (el nombre del fichero) para encontrar un valor (el número de inodo asociado). Por tanto, son aplicables las técnicas que se usan habitualmente para afrontar este tipo de problemas como son el uso de árboles de búsqueda y de funciones *hash*.

Precisamente, esa es la solución que utiliza *ext3*, y retroactivamente *ext2*, para gestionar eficientemente las búsquedas en los directorios: una estructura jerárquica basada en *hash* denominada *HTree*, que no estudiaremos en más detalle. Esta funcionalidad puede seleccionarse cuando se crea el sistema de ficheros mediante la opción *dir_index*, estando activa por defecto.

Ejercicio: directorios indexados

Para realizar este ejercicio, cree un directorio que ocupe más de un bloque de datos y verifique con *lsattr* que se está usando la técnica descrita en este apartado.

Cuestión 13

Describa el proceso. Aunque no vamos a entrar en detalle del modo de operación del *HTree*, por curiosidad, puede ver la información que gestiona este mecanismo usando el mandato *htree* de *debugfs*.

Redimensionamiento del sistema de ficheros

Como se comentó al inicio del documento, un sistema de ficheros debería permitir su redimensionamiento para adaptarse al nuevo tamaño del dispositivo, sea este una partición, un volumen o el disco proporcionado por una máquina virtual. En el caso de un aumento de tamaño, este redimensionamiento debería poder

realizarse estando el dispositivo operativo, es decir, teniéndolo montado, lo cual es técnicamente más exigente dado que el sistema operativo tiene información en memoria del estado del sistema de ficheros.

La manera de hacer crecer un sistema de ficheros es añadir nuevos grupos de bloques (a no ser que hubiera sitio en el último grupo de bloques). Cuando se crea al sistema de ficheros, como se analizó previamente, se calcula cuántos grupos de bloques se requieren teniendo en cuenta el tamaño del dispositivo y la cobertura de cada grupo (con bloques de 4KiB, 128 MiB). Dado que a la tabla de descriptores de grupos se le asigna el número de bloques completos del disco que sea suficiente para albergar las entradas requeridas, normalmente, habrá entradas sin usar en el último bloque asignado a esa tabla (tenga en cuenta que en ext2 cada descriptor de grupo de bloques ocupa 32 bytes, con lo que en un bloque de 4KiB caben 128 descriptores). Esas entradas sin usar van a permitir que el disco crezca proporcionalmente al número de entradas libres disponibles.

Surge de manera natural la cuestión: ¿qué ocurre si la capacidad de crecimiento que proporcionan estas entradas sobrantes no es suficiente para satisfacer las necesidades de un redimensionamiento?

- Si el disco no está montado, la operación de redimensionamiento (*resize2fs*) puede mover a otra ubicación el mapa de bits de bloques libres y asignar ese bloque a la tabla de descriptores de grupo. Recuerde que la ubicación de los mapas de bits no está prefijada y viene especificada en el descriptor de ese grupo.
- Si el disco está montado, la operación de redimensionamiento no puede, en principio, realizarse puesto que el sistema operativo guarda información en memoria de los metadatos del sistema de ficheros y estos no pueden cambiar de ubicación. Hay que matizar que con la opción *meta_bg* de *ext4*, que se comentará brevemente en la sección dedicada a ese formato, incorporada retroactivamente en *ext2* y *ext3*, actualmente, sí puede realizarse este redimensionamiento *en caliente*.

Para afrontar *a priori* esta necesidad de redimensionamiento, en *ext3*, y, retroactivamente, en *ext2*, se proporciona al crear un sistema de ficheros la opción *resize_inode*, habitualmente activa por defecto, que reserva para la tabla de descriptores de grupos varios bloques adicionales de manera que permite al sistema de ficheros crecer hasta 1024 veces su tamaño inicial.

Ejercicio: redimensionamiento del sistema de ficheros

Se pretende analizar cómo *ext3* y, retroactivamente, *ext2* dan soporte al redimensionamiento *en caliente* del sistema de ficheros. Para realizar el ejercicio retomamos el dispositivo creado con un sistema de ficheros *ext3* (*miDisco3*). A continuación, se explican los pasos requeridos para realizar este ejercicio:

- Estando montado el dispositivo, aumente el tamaño del disco:

  truncate -s 4G miDisco3

- Como se comentó previamente, cuando se monta un sistema de ficheros almacenado en un fichero en vez de en un dispositivo, se crea automáticamente un dispositivo de bloques de tipo bucle (*loop*). Cuando se cambia el tamaño del fichero que actúa de disco, es necesario reflejar este cambio en el dispositivo de bloques de tipo bucle asociado al fichero (*losetup -c*). Este paso no sería necesario, evidentemente, si trabajáramos con un dispositivo real.

  sudo losetup -c $(losetup -l | grep miDisco3 | awk '{print $1}')

- Proceda al redimensionamiento en línea del sistema de ficheros:

  sudo resize2fs miDisco3 # si estuviera desmontado, no requeriría ser superusuario

Cuestión 14

Analice usando *dumpe2fs* cómo ha cambiado la estructura del disco después del redimensionamiento. Calcule hasta cuánto podría crecer ese sistema de ficheros. Nótese que con el uso retroactivo de la opción meta_bg de ext4, que se explicará brevemente más adelante, el sistema de ficheros puede crecer más allá de ese límite calculado.

Ahora vamos a probar la reserva a priori de bloques para la tabla de descriptores de grupos de bloques.

- Cree un nuevo sistema de ficheros con las opciones por defecto, que incluyen las que facilitan el redimensionamiento (*sparse_super* y *resize_inode*) y móntelo:

  truncate -s 512M miDisco3bis
  mkfs.ext3 -i 8192 -b 4096 -I 128 miDisco3bis
  sudo mount miDisco3bis /mnt

- Realice un redimensionamiento, tal como se explicó en el ejercicio previo, pero en este caso hasta 64GiB.

Cuestión 15

Analice usando *dumpe2fs* qué diferencias existen entre la disposición inicial del sistema de ficheros al usar estas dos nuevas opciones por defecto. Asimismo, explique cómo ha cambiado la estructura del disco después del redimensionamiento.

El sistema de ficheros ext4

Esta nueva versión incluye un número significativo de cambios, lo que rompe la compatibilidad con *ext2/ext3* (mantiene compatibilidad solo hacia atrás, permitiendo montar como *ext4* un sistema de ficheros *ext2/ext3*, pero no al revés). En esta sección, nos centraremos solo en dos de estos aspectos novedosos: el uso de *extents* y la preasignación persistente. Sin embargo, *ext4* ha incorporado otras muchas funcionalidades que revisamos someramente a continuación y que no forman parte del ejercicio:

- Mejora significativa en los parámetros limitantes del sistema de ficheros al pasar a usar direcciones de bloque de 48 bits. Observe que esto conlleva un tamaño máximo del sistema de ficheros de 1EiB si se usan bloques de 4KiB. Se ha añadido incluso una extensión que maneja direcciones de 64 bits que permite que el sistema de ficheros pueda alcanzar hasta 64ZiB.
- Posibilidad de crear conjuntos de grupos de bloques para diversas funcionalidades:
  - En la disposición normal de *ext4*, un fichero cuyo tamaño sea mayor que la cobertura de un grupo de bloques no puede estar totalmente contiguo en el disco. Con la opción *flex_bg* se puede crear un conjunto formado por varios grupos de manera que los mapas de bits y las tablas de inodos de los miembros del conjunto estás agrupadas, permitiendo que las zonas de datos de todos los miembros aparezcan contiguas.
  - La tabla de descriptores de grupos de un sistema de ficheros extremadamente grande puede no caber en el espacio proporcionado por un grupo de bloques, creando un factor limitante. Con la opción *meta_bg*, se pueden crear conjuntos de grupos de bloques, tal que cada conjunto tiene su propia tabla de descriptores de grupo (se podría considerar, de forma simplificada, que, con respecto a la tabla de descriptores de grupos, cada conjunto se comporta como un sistema de ficheros independiente). Esta opción, trasladada retroactivamente a *ext2* y *ext3*, posibilita la extensión de un sistema de ficheros más allá de la tabla de descriptores de grupo original, creando un nuevo *metagrupo* para la extensión.
- Asignación de bloques diferida y múltiple. En las versiones previas, se asigna un bloque físico a un bloque lógico de un fichero cuando se produce la primera escritura sobre el mismo. Con esta optimización se difiere esta asignación hasta que se vuelca el bloque lógico modificado por primera vez desde la caché del

sistema de ficheros hacia el dispositivo. Este mecanismo evita asignar espacio en el dispositivo a ficheros temporales cuyo tiempo de vida es menor que el periodo de volcado a disco de la caché. Además, permite realizar en ese momento de volcado la asignación simultánea de múltiples bloques, permitiendo optimizar la ubicación de los bloques en el dispositivo.

- Mejora en la gestión de las fechas asociadas al fichero. En UNIX las fechas asociadas a un fichero se representan como el número de segundos transcurridos desde el 1 de enero de 1970. Esta circunstancia plantea dos limitaciones. Por un lado, dado que las versiones previas almacenan las fechas como un contador de 32 bits, aparece el "efecto 2038", puesto que en ese año se agotará dicho contador. Por otro lado, una resolución de un segundo en la gestión de las fechas es bastante pobre. Para solventar estos problemas, en *ext4* las fechas se gestionan con 64 bits lo que permite una resolución de nanosegundos y un aumento de dos bits en el contador de segundos. Esta mejora requiere un tamaño del inodo de 256, que es el valor por defecto en *ext4* y, retroactivamente, para *ext2* y *ext3*.
- Mejoras en la gestión de los atributos extendidos. Se permite que los atributos extendidos ocupen más de un bloque, como ocurre en las versiones previas, asignándoles su propio inodo que será referenciado desde el inodo del fichero. Esta opción, denominada *ea_inode*, es configurable en el momento de crear el sistema de ficheros y proporciona una funcionalidad similar a los *streams* del NTFS de Windows.
- Uso de *checksums* para los metadatos y para el *journal*, lo que mejora la fiabilidad del sistema de ficheros.

Extents

Como se ha comentado previamente, en aras de optimizar el rendimiento en los accesos a un fichero, el sistema de ficheros realiza un esfuerzo considerable para que los sucesivos bloques del fichero estén ubicados en bloques sucesivos del dispositivo.

Asumiendo que el sistema de ficheros logra hasta cierto punto este objetivo de contigüidad, si reflexionamos sobre el modo de operación del inodo, llegaremos a la conclusión de que no es la estructura de datos más adecuada para mantener la información de traducción bajo ese supuesto.

Intentemos entenderlo utilizando un símil. Suponga que está usando una agenda que le permite anotar qué actividad realiza cada 10 minutos. Con esa agenda, si tengo una reunión de 5 horas desde las 9 hasta las 14, tendré que incluir 30 anotaciones idénticas: de 9 a 9:10 estoy en la reunión, de 9:10 a 9:20 estoy en la reunión…

Es indudable que no voy a tardar mucho en cambiarme a otra agenda que permita definir las actividades por tramos, de manera que para la reunión planteada solo tengo que hacer una anotación: de 9 a 14 estoy en la reunión.

Aplicándolo al inodo, si un fichero se ubica de forma contigua en el disco a partir del bloque físico 1000, los punteros del inodo especificarán los valores 1000, 1001, 1002… No parece que sea la representación más adecuada cuando la contigüidad es la norma y no la excepción.

Los *extents*, usados también en NTFS, permiten definir un tramo, es decir, la correspondencia entre un rango de bloques lógicos del fichero contiguos y el rango de bloques físicos del dispositivo contiguos donde están ubicados.

En *ext4* un *extent* (un descriptor de tramo) queda definido por una estructura de 96 bits:

- El bloque lógico inicial del tramo, que ocupa 32 bits.
- El bloque físico inicial del tramo, que ocupa 48 bits.
- El número de bloques que ocupa el tramo, que ocupa 15 bits de una palabra de 16 bits, donde el bit de mayor peso indica sin el tramo está inicializado o no, como se apreciará en el ejercicio siguiente (*debugfs* muestra una *u* para indicar que un tramo no está inicializado).

En el inodo se almacenan hasta 4 *extents* para guardar la información de ubicación de un fichero. En caso de que no sean suficientes, se crea un árbol de bloques de disco que contienen *extents*, tal que en el inodo se almacena la raíz de ese árbol.

Para reforzar este nuevo concepto, analicemos por qué motivos un fichero puede necesitar varios *extents*:

- El fichero está contiguo en disco, pero es muy grande. Suponiendo que el tamaño del bloque es de 4KiB, un *extent* permite definir un tramo de 128MiB ($2^{15}x2^{12}$). Por tanto, habría que gastar un *extent* por cada 128MiB incluso aunque el fichero esté contiguo.
- El fichero no está contiguo en el disco (no contigüidad física): hay que gastar un *extent* por cada tramo contiguo en disco.
- El fichero tiene huecos debido a que se ha ido escribiendo en distintas partes del fichero (no contigüidad lógica): hay que gastar un *extent* por cada tramo contiguo en el fichero.

Ejercicio: *extents*

Para este ejercicio, necesita tener un dispositivo con un sistema de ficheros *ext4* (me temo que en este apartado sí que vamos a "gastar disco de verdad").

- Creamos y montamos un sistema de ficheros *ext4*:

```
truncate -s 1G miDisco4
mkfs.ext4 miDisco4 # usando opciones por defecto
sudo mount miDisco4 /mnt
```

- Creamos 3 ficheros de distintos tamaños:

```
dd if=/dev/zero of=/mnt/FICHERO1 bs=4K count=32
dd if=/dev/zero of=/mnt/FICHERO2 bs=4K count=32K
dd if=/dev/zero of=/mnt/FICHERO3 bs=4K count=128K
```

Cuestión 16

Analice usando los mandatos *stat* y *extents* de *debugfs* qué *extents* usa cada fichero.

Preasignación persistente

Algunas aplicaciones conocen *a priori* cuál será el tamaño final del fichero que gestionan por lo que sería en este caso conveniente reservar inicialmente bloques en el disco para el futuro contenido del fichero sin necesidad de tener que inicializarlos escribiendo algún valor (normalmente, un cero). Esta preasignación inicial facilita la asignación contigua de espacio al fichero. Se dispone del servicio y del mandato *fallocate* para realizar esta labor.

Ejercicio: preasignación

En este ejercicio vamos a comparar cuatro formas de crear un fichero de 20MiB:

- Escribiendo un fichero de 20MiB lleno de ceros:

```
dd if=/dev/zero of=/mnt/F1 bs=4K count=5K
```

- Escribiendo un cero justo en el último byte del fichero ($20x2^{20}-1$), haciendo, por tanto, que aparezca un hueco en todos los bytes previos:

```
dd if=/dev/zero of=/mnt/F2 bs=1 count=1 seek=20971519
```

- Usando *truncate*:

truncate -s 20MiB /mnt/F3

- Usando *fallocate*:

  fallocate -l 20MiB /mnt/F4

Cuestión 17

Escriba un informe comparando los cuatro métodos, explicando la información de traducción requerida en cada uno de ellos, así como el número de bloques asignados.

Plazo y modo de entrega

El plazo de entrega del trabajo es el 15 de junio de 2021.

La entrega se realiza en *triqui* ejecutando el mandato:

entrega.soa ficheros.2021

Este mandato realizará la recolección de los ficheros *autor.txt*, con los datos del alumno, y *memoria.pdf*, que debe incluir la solución del ejercicio, del directorio *~/DATSI/SOA/ficheros.2021*.

Se proporciona como material de apoyo el fichero `ficheros.tgz` con el siguiente contenido:

- `cphuecos.c`: copia un fichero en otro pero dejando huecos en los bloques del fichero original que solo tengan ceros.
- `cmpbl.c`: compara dos ficheros del mismo tamaño mostrando en qué bloques difieren.

Para cualquier duda sobre el ejercicio, consulte a Fernando Pérez Costoya (*fperez@fi.upm.es*).