Guía de trabajo autónomo

Tema 1

SISTEMAS DE ARCHIVOS: INTERFAZ E IMPLEMENTACION

Objetivos

Estudiaremos la interfaz del sistema de archivos y veremos las principales abstracciones suministradas por el sistema operativo. Con mayor detalle, nos dedicaremos a ver cómo se implementan estas abstracciones y cuales son las principales técnicas empleadas para su materialización. En concreto:

- Analizaremos las abstracciones que suministra el sistema operativo relacionadas con el almacenamiento permanente y como se implementan.
- Veremos las estructuras de datos de memoria que utiliza el sistema operativo para manejar archivos.
- Los métodos utilizados para asignar espacio en disco a los archivos, y cuales son sus principales ventajas e inconvenientes.
- Cómo se sigue la pista al espacio libre de disco.
- Estudiaremos el mecanismo de archivos proyectados en memoria extensamente utilizado en los sistema actuales tanto a nivel kernel como usuario.
- Comprenderemos la necesidad de introducir la planificación de disco y los principales algoritmos utilizados.
- Veremos que es un sistema RAID y qué tipos hay.

Recursos

Capítulos 11 y 12 del libro de Stallings.
Tema 6 de Sistemas Operativos I.
Repasar los apartados de memoria externa del Tema 6 de Introducción a los Computadores o Apéndice 11.A del Libro de Stallings.

Contenidos

Repaso de la estructura de disco
1 Persistencia de datos
    1.1 Gestión de archivos
2 Interfaz del sistema de archivos

3 Archivos: Atributos y operaciones
4 Directorios
5 Protección
6 Semánticas de consistencia
7 Expandiendo el sistema de archivos
8 Diseño software del sistema de archivos
    8.1 Estructura de capas
    8.2 Búferes de entradas/salidas y caché de búferes
9 Implementación de los sistemas de archivos
    9.1 Estructuras de datos en memoria
    9.2 Organización y control de flujo
    9.3 Gestión del almacenamiento secundario
        9.3.1 Métodos generales
        9.3.2 Sistema FAT
        9.3.3 Sistema ext2
10 Planificación de disco
11 Archivos proyectados en memoria

Plan de trabajo para el tema

Semana	Día	Trabajo presencial	Trabajo no presencial
1ª	Gestión de archivos: interfaz y estructura lógica		
		Presentación	
		Repaso de los principales conceptos de SOI	
		Interfaz de los sistemas de archivos: Apartados 1 a 3	
2ª	Continuamos describiendo la interfaz		
		Directorios: Apartado 4	
		CM: Apartados 5 y 6. TG: 1.1	TI: Próximo lunes estudiar Apartado 8 de la Guía.
		Sin clase: Día de Andalucía	
3ª	Implementación de sistemas de archivos		
		Dudas Apartado 8 CM: Apartado 9.1 y 9.2	TI: Leer para mañana el Apartado 9.3
		Dudas apartado 9.3 TG: 1.3 y 1.4	TI: Leer para el jueves el Apartado 9.3.1
		Dudas del Apartado 9.3.1 TG: 1.5 y 1.6	TI: Leer para el lunes próximo el Apartado 9.3.2
4ª	Continuación de la implementación de los sistemas de archivos		
		TG: 1.7, 1.8, 1.9	TI: Leer para jueves Apartado 10
		Dudas Apartado 10 TG: 1.10, 1.11	TI: Leer para jueves Apartado 11
		Dudas Apartado 11 TG. 1.12 y 13 CM: Archivos proyectados en memoria	

Abreviaciones: **CM** – Clase magistral; **TG** – trabajo en grupo; **TI** – Trabajo individual.

Repaso

Lectura 1.1: Repaso sobre la estructura de los dispositivos de almacenamiento masivo.

Si no recuerdas bien las estructura y características de los discos magnéticos, puedes repasar el Apéndice 11A del libro de texto, o el Capítulo 6 de libro de A. Prieto, J.C. Torres y A. Lloris, utilizado en la asignatura de primero de Introducción a los Computadores.

Esta información es útil para comprender ciertas organizaciones de sistemas de archivos en disco implementadas por los sistemas operativos.

Tiempo estimado: 15 minutos

1 Persistencia de datos

La estructura de almacenamiento básica utilizada por la mayoría de los sistemas operativos es el *archivo* o *fichero*. La siguiente lectura nos en el concepto de fichero y sistema de ficheros.

Lectura 1.2:

Lee atentamente los epígrafes "Ficheros y sistemas de archivos" y "Sistemas de gestión de archivos" del Apartado 12.1 "Descripción básica" del libro de texto. El epigrafe "Estructura de un fichero" lo habreis visto en bases de datos.

Anotas las dudas que te surjan para resolverlas en la próxima clase.

Duración estimada: 15 minutos.

1.1 Gestión de archivos

Las principales funciones de sistema opertivo relacionadas con la gestión de archivos son:

- *Gestión de disco* - estable cómo se organiza la información en disco para seguir la pista a la información que almacenan los archivos y la que necesita el sistema operativo para su gestión.
- *Designación* ("naming") - establece la forma en cómo se nombran los archivos por parte del usuario y cómo el sistema operativo traduce estos nombres en a los nombres asignados por el sistema operativo.
- *Protección* - el SO debe garantizar la protección de la información contenida en los archivos creados por un proceso/usuario frente a accesos no autorizados por parte de otros proceso/usuario.
- *Fiabilidad/durabilidad* - El SO debe garantizar que cuando se produce una ciada del sistema se mantenga la información en almacenada en disco y que no se pierdan las posibles operaciones pendientes sobre el mismo.
- *Control de concurrencia* o bloqueo de archivos – El SO debe permitir que varios usuarios/procesos, si lo desean, puedan acceder simultáneamente al mismo archivos, es decir, realicen un acceso concurrente al mismo. También, debe suministrar mecanismos para controlar ese acceso si es necesario establecer un orden para las operaciones.

2 Interfaz del sistema de archivos

Fijémonos en el Programa 1.1, un ejemplo típico para acceder a un archivo escrito en C utilizando las llamadas al sistema operativo de Unix.

Programa 1.1. Ejemplo 3.2 del libro de Robbins y Robbins para copiar un archivo.

```
#include <sys/types.h>
#include <sys/uio.h>
#include <stdio.h>
#include <unistd.h>
```

```
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#define BLKSIZE 1024

void main(int argc, char *argv[])
{
   int from_fd, to_fd;
   int bytesread, byteswritten;
   char buf[BLKSIZE];
   char *bp;

   if (argc != 3) {
      fprintf(stderr, "Usage: %s from_file to_file\n", argv[0]);
      exit(1);
   }

   if ((from_fd = open(argv[1], O_RDONLY)) == -1) {
      fprintf(stderr, "Could not open %s: %s\n",
              argv[1], strerror(errno));
      exit(1);
   }

   if ((to_fd = open(argv[2], O_WRONLY | O_CREAT | O_EXCL,
              S_IRUSR | S_IWUSR)) == -1) {
      fprintf(stderr, "Could not create %s: %s\n",
              argv[2], strerror(errno));
      exit(1);
   }

   while (bytesread = read(from_fd, buf, BLKSIZE)) {
      if ((bytesread == -1) && (errno != EINTR))
         break;             /* real error occurred on the descriptor */

      else if (bytesread > 0) {
         bp = buf;
         while(byteswritten = write(to_fd, bp, bytesread)) {
             if ((byteswritten == -1) && (errno != EINTR))
                break;
             else if (byteswritten == bytesread)
                break;
             else if (byteswritten > 0) {
                bp += byteswritten;
                bytesread -= byteswritten;
             }
         }
         if (byteswritten == -1)
             break;
      }
   }
   close(from_fd);
   close(to_fd);
   exit(0);
}
```

Los pasos que damos para la manipulación del archivos, vienen esquematizados en la Figura 1.1. Como se puede ver del programa anterior y de la figura, los pasos a seguidos son:

1) Abrimos el archivo a través de la llamada `open()` - Esto produce a su vez varias operaciones internas por parte del sistema operativo:
   a) *Traducimos* el nombre dado por el usuario, normalmente una cadena de caracteres, en el nombre interno por el cual es conocido por el sistema operativo (por ejemplo, en Unix, por un número de inodo).

b) Una vez traducido el nombre, el SO asigna las estructuras de datos necesarias para la manipulación del archivo (como veremos en el Apartado 9.1).
c) El SO realiza un *control de acceso* (como veremos en el Tema 4) para ver si el usuario tiene permisos para acceder al archivo. Si los tiene continua con los pasos siguientes. Si no los tiene, dará error.

2) El SO devuelve un identificador para el archivo, denominado *descriptor* en la terminología Unix y *handler* en Windows. Este identificador suele ser un índice en una tabla interna del sistema operativo que contiene la referencia al archivo. En nuestro ejemplo, `from_df` es el descriptor de uno de los archivos.
3) El usuario utiliza a continuación las llamadas `read`, `write`, `lseek`, etc. (como veremos en las prácticas) o las correspondientes funciones de la biblioteca de E/S de C, para manipular el archivo. Todas estas llamadas utilizan como argumento el descriptor de archivo devuelto por `open()`.
4) Tras las operaciones `read()`, `write()`, etc. el sistema realiza las acciones oportunas. Por ejemplo, con `read()` lee la información especificada del archivo y la deposita en el búfer que se pasa como argumento a la llamada.
5) Una vez que no necesitemos acceder al archivo, se lo indicamos al SO mediante la llamada `close()`, lo que permite liberar todos los recursos del sistema asignados previamente con el `open()`.

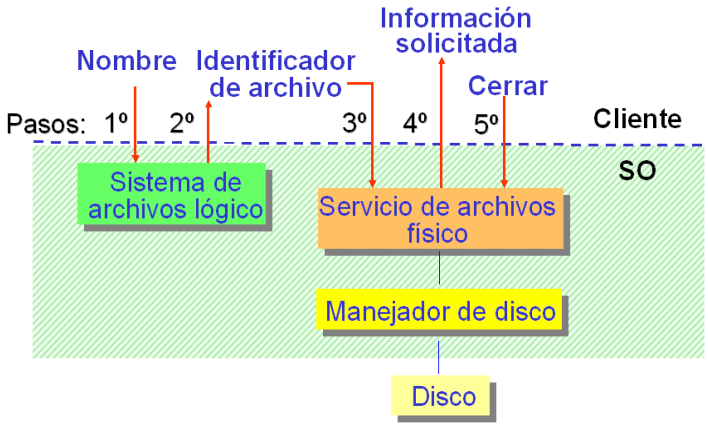


Figura 1.1.- Pasos en una solicitud al servicio de archivos.

El programa anterior nos permite hacer un comentario. La interfaz suministrada para acceder a archivos favorece el acceso secuencial. Como vemos del ejemplo, una llamada al sistema para leer del archivo utiliza la posición actual de puntero de lectura/escritura como punto de inicio de la operación. Si deseamos hacer un acceso aleatorio, tenemos que posicionar el puntero de lectura/escritura donde deseemos leer a través de la llamada al sistema `lseek()`. Por tanto, el acceso aleatorio es menos eficiente pues necesita de llamadas adicionales. Como comentario general, la interfaz de llamadas al sistema suministra una máquina virtual que no es "neutra", es decir, esta diseñada y optimizada para un uso concreto.

La interfaz de usuario de algunos sistemas operativos actuales (tales como Windows Vista o MacOS) están tendiendo a ocultar conceptos de bajo nivel como archivos tras metáforas de alto nivel tales como documentos.

- **Descriptor de archivos**

Como hemos visto en el ejemplo anterior, el SO genera un identificador de archivo para cada invocación de `open()`. Este identificador es un índice en una tabla interna del SO que nos permite acceder al archivo. La cuestión que nos planteamos aquí es ¿por qué hacerlo así?

Son dos las razones principales de hacerlo de esta forma. Primero, el descriptor de archivo representa una sesión de trabajo sobre un archivo. Definimos *sesión* de trabajo sobre un archivo al conjunto de operaciones sobre un archivo que se realizan desde que abrimos un archivo hasta que lo cerramos. Definir sesiones de trabajo tiene varias ventajas. Por ejemplo, si abrimos un archivo de solo lectura, el SO supervisa las operaciones de forma que si intentamos acceder a un fichero para escritura nos los indica con un error. También, podemos tener varias sesiones de trabajo, por ejemplo una para lectura y otra para escritura, para lo cual utilizamos diferentes descriptores de archivos.

Segundo, para evitar que en cada operación de acceso al archivo se realice un control de acceso, se realiza una única vez en su apertura y se crea un "puntero protegido" o capacidad (como veremos en el Tema 4) para posteriores accesos. Esta es la razón que el descriptor este en espacio del núcleo y al usuario solo se le pase su indice en la tabla que lo contiene.

Trabajos en Grupo 1.1: Interfaz del sistema de archivos.
Objetivo formativo: Reflexionar sobre la interfaz de los sistemas de ficheros. **Tareas del grupo**: Justificar por que si como hemos visto en el programa anterior podemos copiar un archivo en otro mediante el uso de las llamadas `read` y `write` (leyendo el archivo origen y escribiendo la información en el archivo destino), algunos sistemas como Windows definen en su interfaz de llamadas al sistema una función denominada `CopyFile()` que hace lo mismo. **Criterio de éxito**: Cualquier miembro del grupo podrá demostrar, en un resumen oral o escrito, o mediante respuestas a preguntas del profesor, que ha alcanzado los objetivos formativos propuestos.
Duración: 25 minutos.

3 Archivos: atributos y operaciones

Un archivo es una abstracción construida por el sistema operativo que permite poner nombre y unificar la manipulación de cierta información almacenada en disco. Para construir esta abstracción el sistema operativo mantiene en una estructura de datos la información relevante al archivo. Esta información se denomina *atributos*, o también *metadatos* del archivo, pues es información que mantiene el SO sobre otra información que son los datos almacenados en él.

Algunos de los atributos mantenidos por muchos sistemas operativos son:

- *Nombre* - nombre de usuario
- *Tipo* - caracteriza el contenido del archivo, por ejemplo, si es un archivo asociado a un tipo de aplicación (compilador de C, procesador de texto, etc.) o tiene un contenido especial (en Unix, los archivos son planos o regulares – contienen datos- o son especiales -como un directorio, un cauce, etc.)
- *Ubicación* – su localización en el dispositivo de almacenamiento (más tarde veremos diferentes métodos para guardar esta información.
- *Tamaño* - tamaño actual (bytes, bloques, ...)
- *Protección* - información de control de acceso: quién y qué puede hacer con él. Por ejemplo, en Unix la cadena de bits de protección.
- *Tiempos de creación*, *modificación*, y *último acceso* – por seguridad y vigilancia en el uso de los archivos se almacena el tiempo en el que se creo (por ejemplo, necesario para compilación), cuando fue modificado por última vez (por ejemplo, podemos ver si alguien ha modificado el archivo), y cuando fue accedido por última vez.

La Figura 1.2 muestra un ejemplo de los atributos almacenados por Windows. Aunque hay similitudes con Unix, también hay diferencias. Por ejemplo, el atributo "oculto" de Windows, no se considera en Unix. Como Linux implementa enlaces duros, mantiene un contador de enlaces como atributo del archivo, cosa que no ocurre en Windows. En Linux, podemos ver parte de los atributos de un archivo con `% ls -l archivo`

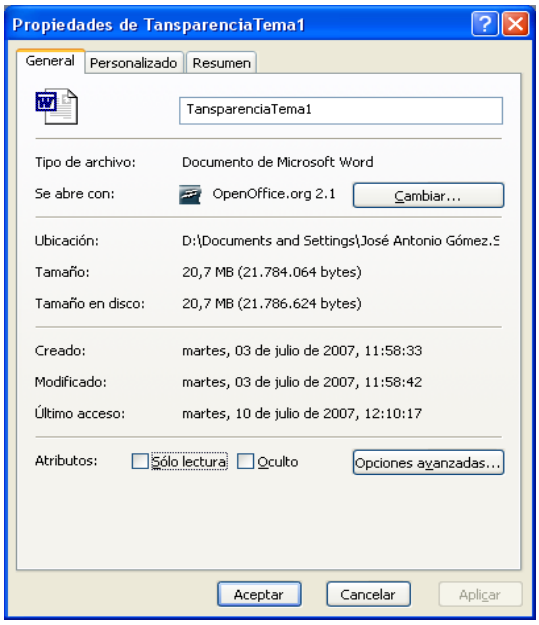


Figura 1.2.- Ejemplo de atributos de un archivo en Windows.

En la lectura del Apartado 12.1 del libro ya pudimos leer algo sobre las operaciones más frecuentes sobre archivo.

4 Directorios: organizaciones.

Lectura 1.3: Concepto de directorio y estructura.
Lee atentamente el apartado 12.3 "Directorios" del libro de texto. Anotas las dudas que te surjan para resolverlas en la próxima clase.
Duración estimada: 15 minutos.

Como se indica al final de la lectura, la estructura de árbol es las más utilizada en la mayoría de sistemas actuales, si bien es la más compleja. Esta estructura permite entre otras cosas compartir archivos entre directorios al suministrar de más de un nombre para un archivo mediante los enlaces.

Un *enlace* es un nombre para un archivo. El disponer de más de un nombre para un archivo simplifica la compartición de archivos ya que realmente solo necesitamos mantener una vez la información que contiene el archivo y solo debemos darle más de un nombre. Dependiendo de la implementación disponemos de diferentes tipos de enlaces.

Los sistemas Unix y Windows implementan *enlaces simbólicos*. Un enlace simbólico es un archivo cuyo tipo asociado en ambos sistemas es el de "enlace simbólico". Este tipo de archivo supone que la información que contiene el archivo debe interpretarse como un nombre absoluto de otro archivo. La Figura 1.3 muestra un esquema de este tipo de enlace y en la Figura 1.4, tenemos ejemplos de Linux y Windows.

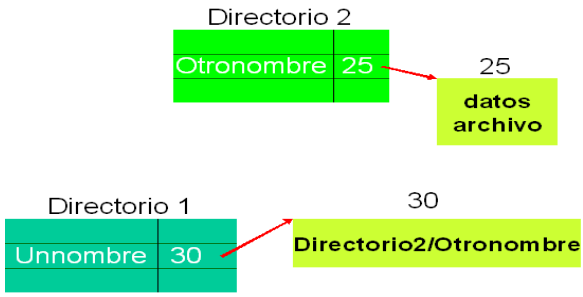


Figura 1.3.- Enlace simbólico a un archivo.

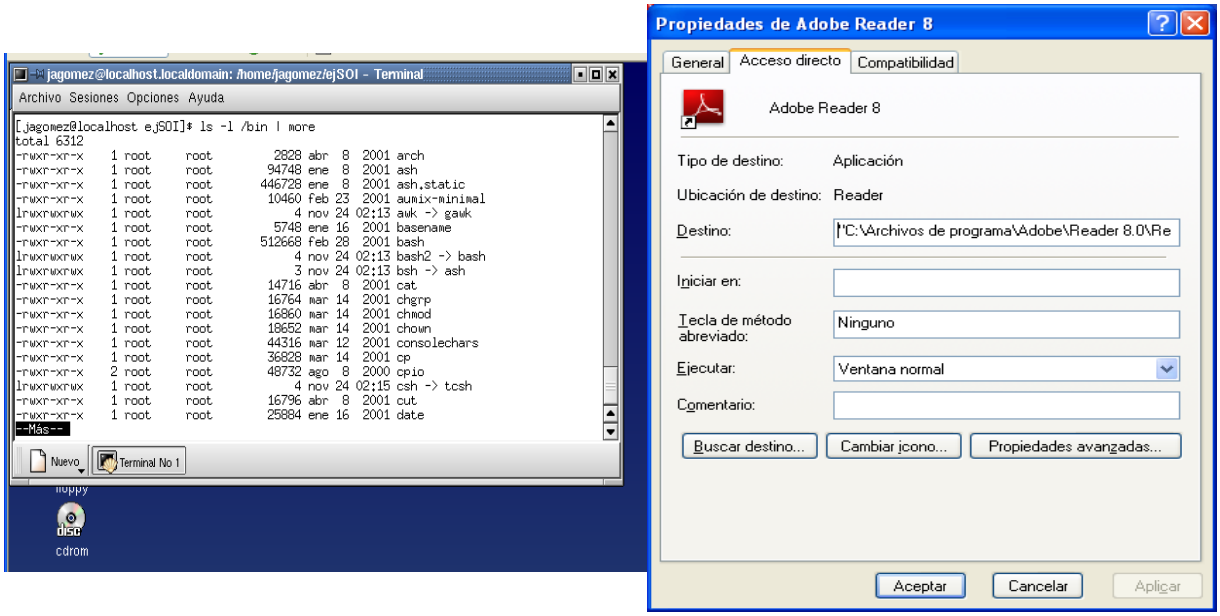


Figura 1.4.- Ejemplos de enlaces simbólicos en Linux y Windows.

Una cuestión a tratar con un enlace simbólico es que se pueda borrar, sin más, el archivo apuntado por el enlace. Esto se resuelve de forma sencilla, cuando el SO trata de resolver un enlace simbólico a un archivo que no existe, simplementa da un error similar a un acceso a archivo que no existe.

En sistemas tipos Unix, existe además otro tipo de enlace, denominado *enlace duro*. Para entenderlos, adelantaremos algunos elementos que veremos con detalle al hablar del sistema de archivos ext2. En concreto, indicar que en sistemas Unix, la estructura que representa un archivo se denomina *inodo*. Su nombre proviene de que cada inodo se identifica unívocamente en un sistema de archivos por un índice, un número que indica su posición en disco. Además, un directorio es un archivo que contiene parejas <nombre_de_archivo, inodo>. Pues bien, crear un enlace duro supone simplemente crear una entrada de directorio. Para compartir un archivo entre dos directorios, estos directorios deberán tener sendas entradas con diferentes nombres pero con el mismo número de inodo como se muestra en la Figura 1.5. Un ejemplo de este tipo de enlace aparece en la Figura 1.6.

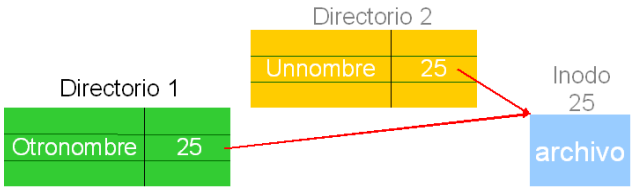


Figura 1.5.- Dos enlaces duros para un archivo.

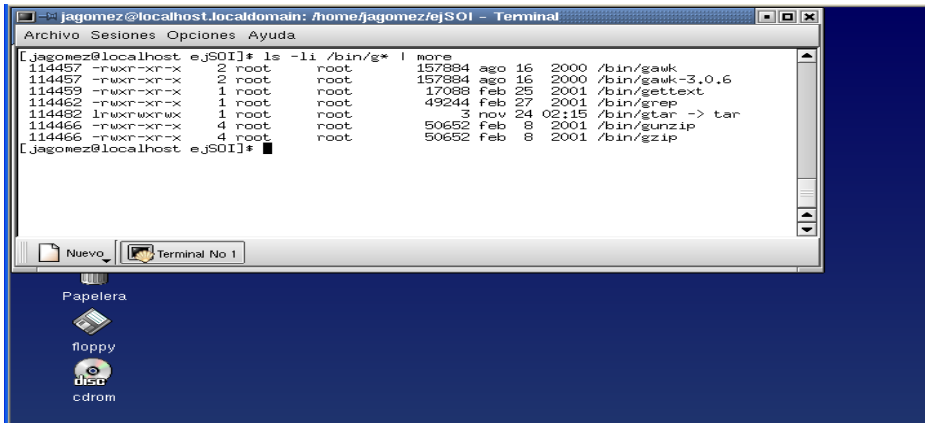


Figura 1.6.- Ejemplo de un enlace duro en Linux.

Cuando se utilizan enlaces duros, hay que asociar con el archivo un *contador de enlaces duros*, que indica cuantos enlaces existe para ese objeto. De esta forma, el objeto solo se borrará cuando el número de enlaces del objeto sea 0. Para que esto ocurra se han debido de ir borrando los enlaces (llamada al sistema `unlink`, u orden `rm`).

También al final de la lectura se hablaba de la implementación de directorios. Es normal implementar los directorios como archivos, es decir, vemos el directorio como un archivo cuyo contenido son entradas de directorio, una por cada archivo que contiene. Un directorio se suele implementar como un Tipo Abstracto de Datos. El sistema no sólo suministra la estructura de datos sino que ofrece los procedimientos para manipularla. A diferencia de un archivos regular, que es accedido y modificado sin más por el usuario, un directorio es manipulado por el usuario a través de los métodos destinados para ello. Esto se hace para preservar la estructura de árbol correcta, dado que modificar una entrada de forma incorrecta pondría en peligro la integridad del árbol.

Respecto a la estructura de árbol, para posibilitar el recorrido el árbol hacia arriba y abajo mediante las diferentes órdenes al respecto, debemos indicar que todo directorio siempre mantiene dos entradas, el "." como autoreferencia y el ".." como referencia al directorio padre.

En relación a los pruntos que acabamos de citar, indicar que sistemas Unix, la creación de enlaces entre directorios esta restringida al *root*, por ello, cuando utilizamos la orden `mkdir` esta se ejecuta con privilegios especiales (como veremos en el Tema de Seguridad y Protección) de *root*. Esto permite entre otras cosas poder crear los enlaces a "." y ".." que son enlaces entre directorios.

5 Protección

La siguiente lectura nos introduce en las formas de protección de ficheros necesarias en los sistemas multitarea. Más adelante dedicaremos un tema a la implementación de la protección en general (no solo de archivos) en los sistemas actuales.

Lectura 1.4:

Lee atentamente el apartado 12.4 "Protección de archivo" y el apartado 12.4 "Compartición de archivos" del libro de texto.

Anotas las dudas que te surjan para resolverlas en la próxima clase.

Duración estimada: 15 minutos.

6 Semánticas de consistencia

Los sistemas operativos multitarea/multiusuario deben abordar un nuevo problema respecto a los archivos. ¿Qué ocurre cuando una tarea o usuario modifica un archivo que esta siendo accedido por otra tarea o usuario? Es decir, qué ocurre cuando compartimos archivos entre tareas y las tareas modifican sus contenidos.

Para poner orden en esta situación, el sistema operativo debe acordar una semántica de acceso que defina cuando es visible una modificación realizada por un proceso desde otro proceso. Algunas semánticas utilizadas:

- *Semántica Unix*: si un proceso realiza una escritura en el archivo, esta es visible a otro proceso sólo con que realice una operación de lectura.
- *Semántica de sesión*: en este tipo, un proceso solo verá los cambio en un archivo realizados por otro, cuando este último lo cierre, es decir, es visible entre sesiones.
- *Semántica de archivo compartidos inmutables*: cuando un archivo se declara por un usuario de tipo compartido, éste automáticamente no puede modificarse por nadie, es de solo lectura.

7 Expandiendo el uso de sistemas de archivos

El algunos sistemas operativos, como Unix o BeOS, la interfaz del sistema de archivos es utilizada como método estándar para acceder a un amplio rango de servicios, no solo para acceder a información depositada en disco, hablamos entonces de *pseudo-sistemas de archivos*. Las operaciones de lectura/escritura convencionales de archivos pueden usarse para diferentes propósitos. Algunos ejemplos de Unix son:

- */proc*: es seudo sistema de archivos que no permite acceder a información almacenada en la memoria principal asignada al kernel o los procesos.
- */dev* o */devfs*: es la interfaz genérica para acceder a los dispositivos.

Trabajos en Grupo 1.2:

Objetivo formativo: Revisar algunos de los conceptos vistos en el tema.
Tarea del grupo: Resolver las siguientes cuestiones:

1. Explica cómo puede considerarse un directorio un tipo abstracto de dato (TAD).

2. En qué formas podría suministrar un sistema de archivos control de concurrencia, es decir, control del uso simultáneo de un archivo, o directorio, compartidos.

3. La interfaz de programación C ofrece para el manejo de archivos una función separada para acceder a una lugar dentro del archivo, *lseek*. ¿Por qué no se suministra una sola función que nos permita leer/escribir en un archivo indicando la posición, por ejemplo, *read (fd, posición, búfer, contador)*?

4. ¿Cuáles son las ventajas de tener una representación de archivos de flujos de bytes (stream-byte)? ¿Cuáles son las desventajas?

5. ¿Por qué no se le permite a un usuario escribir un archivo abierto en modo apertura, si el o ella tiene permiso para hacerlo?

6. ¿Qué problemas pueden producirse de un uso incorrecto de enlaces duros a directorios?

Criterio de éxito: Cualquier miembro del grupo podrá demostrar, en un resumen oral o escrito, o mediante respuestas a preguntas del profesor, que ha alcanzado los objetivos formativos propuestos.
Duración: 25 minutos.

8 Diseño software del sistema de archivos

8.1 Estructura de capas del sistema de archivos

Definimos *sistema de archivos* como todos los componentes del sistema que definen y establecen como se estructuran los archivos, cómo se identifican y cómo se manejan.

En la siguiente lectura vamos a estudiar cómo se estructrura por capas el software destinado a la gestión de archivos,

Lectura 1.5: Estructura del software del sistema de archivos
Lee atentamente el epígrafe "Estructura Lógica del sistema de E/S" del Apartado 11.3 "Aspectos de diseño del sistema operativos". Anotas las dudas que te surjan para resolverlas en la próxima clase.
Duración estimada: 15 minutos.

Con esta estructura software, un archivo tiene varias visiones dependiendo del nivel al que se trabaja, como muestra la Figura 1.7 (esta figura se complementa con la Figura 12.2 del libro de texto).

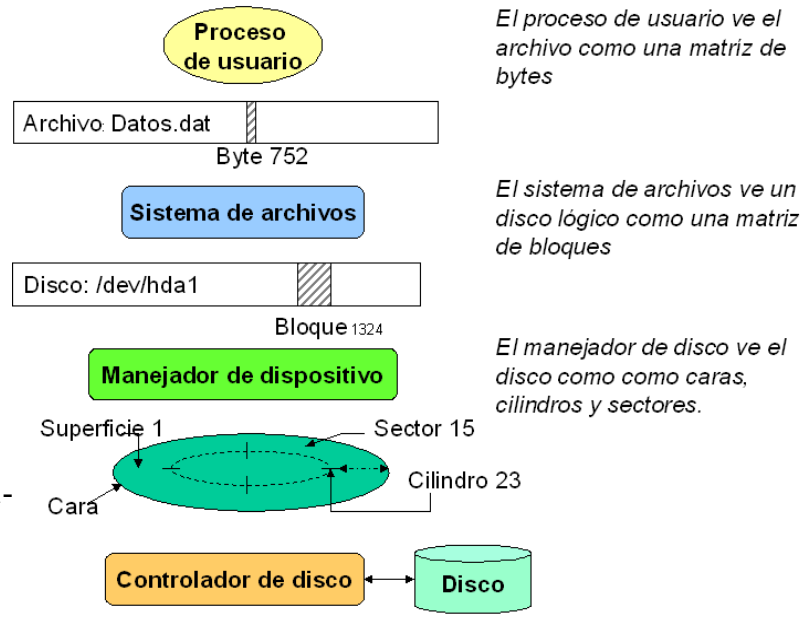


Figura .-

Figura 1.7.- Visiones de un archivo según la capa.

Hay que observar que a veces hablamos de sistema de archivos sin aclarar si no referimos al todo o a un nivel concreto. El contexto nos permite saber a que nivel nos estamos refiriendo.

8.2 Utilización de los búferes de entradas/salidas y caché de búferes

La diferencia entre la velocidad de acceso a los dispositivos y el acceso a memoria se solventa utilizando búferes de entradas/salidas. Si miramos la Figura 11.4 de libro, la ubucación de los búferes esta entre el componente "organización física del sistema de archivos" t

y el manejador del dispositivo, módulo “E/S de dispositivo”.La lectura siguiente no introduce en diferentes sistemas de búferes para analizar sus ventajas e inconvenientes.

Lectura 1.6: Utilización de búferes para lecturas/escritura de archivos.

Lee el epígrafe “” del Apartado 11.4 “Utilización de los búferes de e/s” y el Apartado 11.7 “Caché de disco”. No es necesario leer el epígrafe “Aspectos de rendimiento” de apartado 11.7.

Anotas las dudas que te surjan para resolverlas en la próxima clase.

Duración estimada: 15 minutos.

Como veremos en el Tema 2, el esquema de uso de búferes en sistemas de archivos reales hace que sean bastante complejos dado que gran parte de la eficiencia en el acceso a archivos descansa en ellos. En el caso, de sistemas Unix nos referemos a un sofisticado mecanismo que recibe el nombre de *bufer caché*.

9 Implementación de los sistemas de archivos

9.1 Estructuras de datos en memoria utilizadas por el sistema de archivos

La Figura 1.8 muestra las estructuras de datos que utiliza el sistema operativo para gestionar archivos, y cómo se relacionan estas.

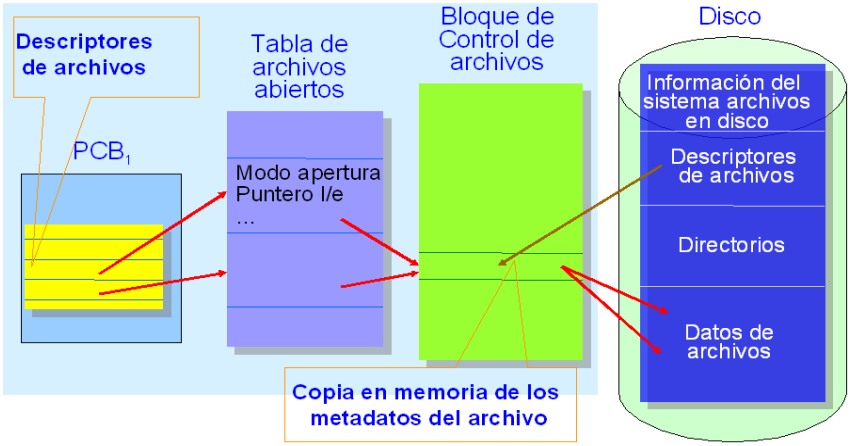


Figura 1.8.- Estructuras de datos del sistema de archivos (en disco y en memoria).

Las estructuras son las siguientes:

- *Bloque de control de archivos* – una estructura donde hay un único elemento por archivo abierto en el sistema que contiene los metadatos del archivo (que se copian del disco) y otra información adicional necesaria para su gestión por parte del sistema operativo.
- *Tabla de archivos abiertos* – contiene un elemento por cada archivo abierto en el sistema. Cada elemento es una estructura de datos que suele contener la siguiente información: el puntero de lectura/escritura que marca la posición en el archivo donde se producirá la siguiente operación de lectura o escritura), información sobre el estado de apertura del archivo (si es de lectura, escritura, el tipo de archivo, si esta bloqueado o no, etc.), y un puntero al bloque de control de archivos. Esta estructura es única para el sistema, e implementa el concepto de sesión de trabajo con un archivo.

- *Vector de descriptores de archivos* – cada proceso posee como parte de su PCB (Bloque de Control del Proceso) una matriz de descriptores que los archivos que tiene abiertos. Para acceder a un archivo utiliza el índice devuelto por la llamada open(). Esta estructura, es una estructura por proceso.

La estructura Bloque de Control de Archivos mantiene una copia de los metadatos de cada archivo abierto en el sistema para minimizar los accesos a disco en cada operación realizada sobre el archivo que afecta estos. Dado que ya tenemos una estructura por archivo ¿cual es el objetivo de tener una Tabla de Archivos Abiertos? Cómo indicamos antes, cada item de esta estructura implementa la sesión de trabajo sobre un archivo. De esta forma, si abrimos un mismo archivo a través de sendas llamadas `open()`, tendremos un único elemento en el Bloque de Control de Archivos, pero tantas estradas en la Tabla de Archivos Abiertos como llamadas de apertura realizadas.

La Figura 1.9 muestra la situación donde hemos abierto dos veces un mismo archivo, una para lectura y otra para escritura. Con esta estructura, además, podemos mantener diferentes punteros de lectura/escritura cada uno para una sesión de trabajo.

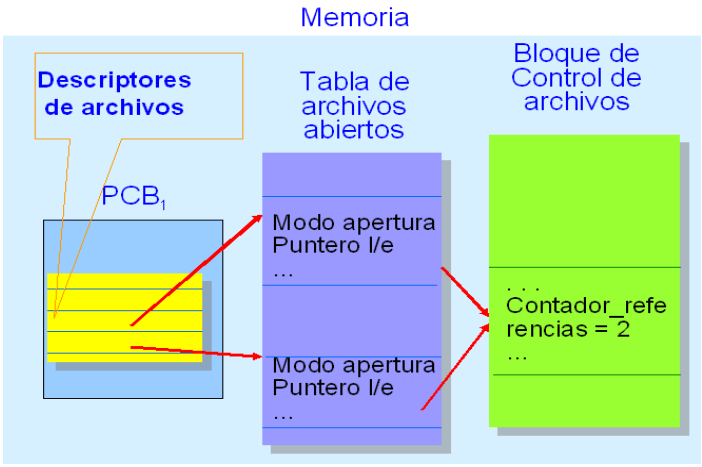


Figura 1.9.- Dos sesiones de trabajo sobre un mismo archivo.

Como podemos observar en la Figura cada entrada de BCA mantiene un *contador de referencias* que indica cuantas referencias existen hacia el, es decir, cuantos punteros estan apuntando a esa entrada. Este contador sirve al sistema de archivos para saber no solo las referencias sino que cuando el contador alcanza el valor 0, sabe que la entrada se puede limpiar por que no esta siendo utilizada por ningún otro componente del sistema.

9.2 Organización del sistema de archivos y control de flujo

Para tener una visión en conjunto de como funciona el sistema de archivos, vamos a representar los diferentes componentes del mismo y estructuras de datos que utilizan. En la Figura 1.10 representamos el flujo de control que se produce con la llamada `open()`, y el la Figura 1.11, el generado por la llamada `read()`.

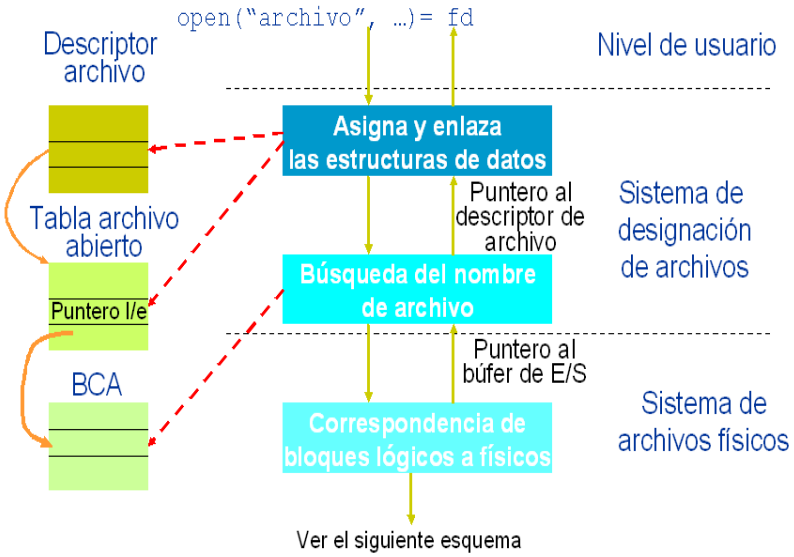


Figura 1.10.- Flujo de control del sistema de archivos con `open`.

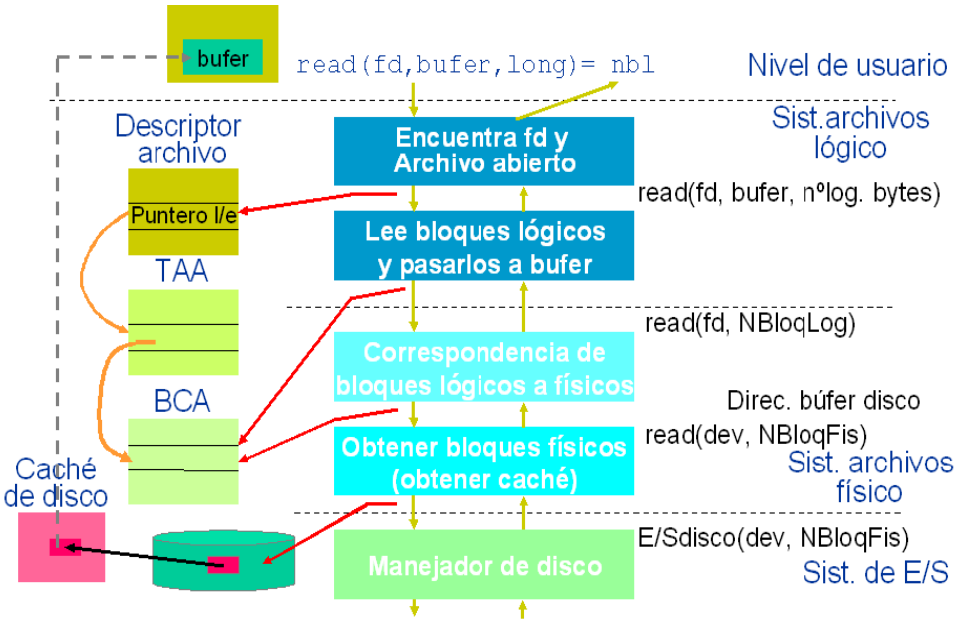


Figura 1.11.- Flujo de control de la llamada `read`.

9.3 Gestión del almacenamiento secundario

En este apartado, veremos como se gestionan los bloques de disco para asignarlos a los archivos para almacenar información.

Lectura 1.7: Gestión de almacenamiento secundario.
Estudia el apartado 12.6 "Gestión del almacenamiento secundario" del libro en el que se explicar los diferentes aspectos involucrados en la asignación de espacio a los archivos y algunos de los métodos de asignación más frecuentes: continuo, encadenado e indexado. Por otra parte, se estudian algunos métodos para seguir la pista a los bloques libres (no asignados) de disco.
Tiempo estimado: 30 minutos

Los métodos anteriores son métodos "puros" que tienen sus ventajas e inconvenientes como se recoge en la Tabla 12.3 del libro de texto. Nosotros vamos a ver con más detalle dos métodos de asignación muy utilizados: el Sistema FAT de Windows y el sistema Ext2 de Linux.

Trabajos en Grupo 1.3: Gestión del espacio en disco.

Objetivo formativo: Reflexionar sobre las ventajas e inconvenientes de los diferentes métodos de asignación de espacio en disco.
Tareas del grupo: ¿Qué organización de archivos elegiría para maximizar la eficiencia en términos de velocidad de acceso, uso del espacio de almacenamiento y facilidad de modificación (añadir/borrar /modificar), cuando los datos son:

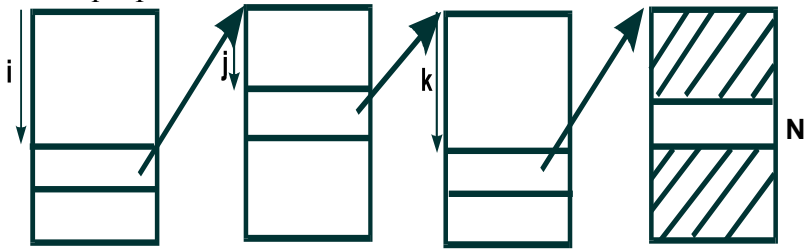
a) modificados infrecuentemente, y accedidos frecuentemente de forma aleatoria
b) modificados con frecuencia, y accedidos en su totalidad con cierta frecuencia
c) modificados frecuentemente y accedidos aleatoriamente y frecuentemente

Criterio de éxito: Cualquier miembro del grupo podrá demostrar, en un resumen oral o escrito, o mediante respuestas a preguntas del profesor, que ha alcanzado los objetivos formativos propuestos.

Duración: 25 minutos.

Trabajos en Grupo 1.4: Gestión del espacio en disco.

Objetivo formativo:
Tareas del grupo: Sobre conversión de direcciones lógicas dentro de un archivo a direcciones físicas de disco. Suponed que estamos utilizando la estrategia de indexación a tres niveles para asignar espacio en disco. Tenemos que tamaño de bloque es de 512 bytes, y el tamaño de puntero es de 4 bytes. Se recibe la solicitud por parte de un proceso de usuario de leer el carácter número N de determinado archivo. Suponemos que ya hemos leído la entrada del directorio asociada a este archivo, es decir, tenemos en memoria los datos PRIMER-BLOQUE y TAMAÑO. Calcule la sucesión de direcciones de bloque (indices i, j, k) que deben dar para leer el bloque de datos que posee el citado carácter N.



Criterio de éxito: Cualquier miembro del grupo podrá demostrar, en un resumen oral o escrito, o mediante respuestas a preguntas del profesor, que ha alcanzado los objetivos formativos propuestos.

Duración: 40 minutos.

Trabajos en Grupo 1.5: Gestión del espacio libre.

Objetivo formativo: Reflexionar sobre los métodos de gestión del espacio libre en disco.
Tareas del grupo:

1) Proponer una solución al siguiente problema: Suponed que utilizamos una lista enlazada para gestionar los bloques libres de disco, ¿podría el sistema operativo reconstruir dicha

lista? En caso afirmativo, indicar cómo. Si no, justificar.

2) Resolver el siguiente ejercicio: podemos implementar la gestión del espacio libre en disco como una lista encadenada con agrupación o mediante un mapa de bits. Si una dirección de disco requiere D bits, el disco tiene B bloques y de estos hay F libres, ¿en qué condiciones la lista utiliza menos espacio que el mapa de bits?

Criterio de éxito: Cualquier miembro del grupo podrá demostrar, en un resumen oral o escrito, o mediante respuestas a preguntas del profesor, que ha alcanzado los objetivos formativos propuestos.

Duración: 25 minutos.

9.3.1 El sistema de archivos FAT

Vamos a estudiar el sistema de archivos FAT (*File Allocation Table*) creado por Microsoft. Dado que este sistema no es muy complicado y esta soportado por casi todos los sistemas operativos, lo hace ideal para medios extraíbles como disquetes y memorias de estado sólido y como medio de compartir información entre diferentes sistemas operativos.

El sistema de archivos FAT es una variante del método enlazado en el que los indices se eliminan de los bloques y se almacena en una tabla enlazada, la FAT. Cada elemento de la tabla FAT representa un bloque, o agrupación de bloques (*cluster*), de disco. Un archivo viene representado por una lista enlazada de entradas de la FAT y el último elemento es un EOF.

La Figura ?? muestra la estructura actual de un disco formateado FAT. Una partición esta dividida en clusters del mismo tamaño, que son bloques contiguos de disco. El tamaño del cluster varía dependiente del tipo de FAT y del tamaño de la partición. El tamaño típico de los cluster varía entre 2KB y 32 KB.

El primer sector es el sector de arranque que incluye el *Bloque de parámetros de la BIOS* (información del sistema de archivos básico, en particular, su tipo, y punteros para localizar el resto de secciones), y el código de arranque (*boot loader*) del sistema operativo. También, podemos tener reservados otros sectores, como ocurre con la FAT32 que reserva el 6 sector como copia de seguridad del sector de arranque.

La región de FAT contiene dos copias de la FAT por motivos de redundancia, aunque la copia extra es raramente usada, incluso por las utilidades de recuperación de disco. La región de datos contiene los datos de los archivos y directorios.

Hay diferentes variantes del sistema FAT dependiendo del tamaño del puntero utilizado para direccionar cluster. Así, tenemos FAT12, FAT 16, FAT 32 si los punteros tienen 12, 16 ó 32 bits. Con el crecimiento de los tamaños de los discos, ha sido necesario ir incrementado la capacidad de direccionamiento del sistema de archivos. Cada variante tiene además, otras características adicionales. En nuestro caso, estudiaremos con mayor detalle la FAT32.

Tabla 1.1.- Valores de las entradas de la FAT.

FAT12	FAT16	FAT32	Descripción
0x000	0x0000	0x?0000000	Cluster Libre
0x001	0x0001	0x?0000001	Valor reservado; no usar
0x002 - 0xFEF	0x0002 - 0xFFEF	0x?0000002 - 0x?FFFFFEF	Cluster utilizado; apunta al siguiente cluster
0xFF0 - 0xFF6	0xFFF0 - 0xFFF6	0x?FFFFFF0 - 0x?FFFFFF6	Valores reservados; no usar
0xFF7	0xFFF7	0x?FFFFFF7	Sector malo en cluster o cluster reservado
0xFF8 - 0xFFF	0xFFF8 - 0xFFFF	0x?FFFFFF8 - 0x?FFFFFFF	Último cluster en el archivo

Hay que comentar que en la FAT 32, realmente solo se utilizan 28 bits para direccionar. Los cuatro bits superiores son 0 y están reservados por lo que no deben tocarse. En la Tabla ?? estos valores aparecen con el signo ?.

- **Tabla de directorios**

Una tabla de directorio o folder es un archivo especial que contiene un directorio. Cada archivo o directorio contenido en él viene representado por una entrada de 32 bytes con la estructura que aparece en la Figura 1.12:

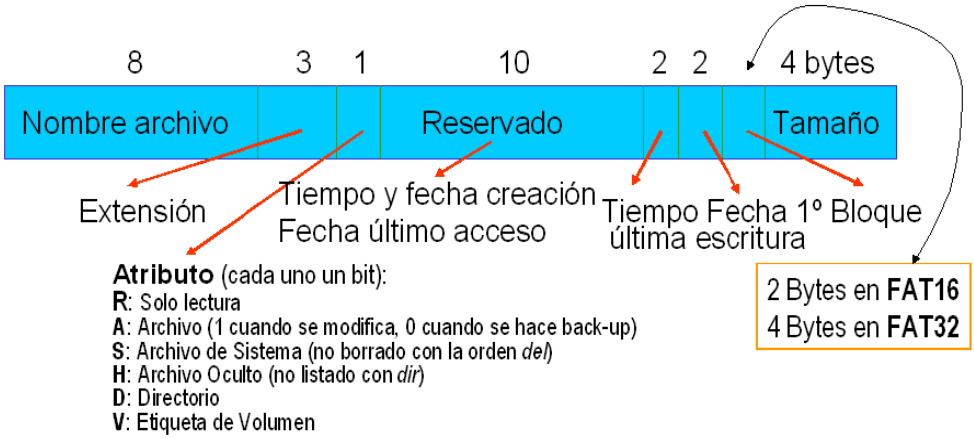


Figura 1.12.- Estructura de una entrada de directorio FAT.

La estructura de la Figura 1.12 permite utilizar el formato 8.3 para los nombres. Sin embargo, las versiones más modernas permite almacenar Nombres Largos de Archivos (LFN), de hasta 255 caracteres, a través de un truco en la entrada de directorio (en este caso se usa el término VFAT – *Virtual FAT*). En este caso, los nombres de archivos largos son traducidos a un "alias" en el formato 8.3. El truco esta en que el nombre se escribe a través de múltiples entradas de directorio, cada una de las cuales suministra 32 bytes.

Tabla 1.2.- Formato de una entrada LFN.

1 BYTE	Número de secuencia de registro LFN - Bits 5:0 mantiene el numero de secuencia LFN (1..63). Este número es base uno. Esto limita el número de entradas LFN para nombres largos a 63 entradas ó 63 * 13 = 819 caracteres por nombre. - Bit 6 se ajusta para el último registro LFN en la secuencia. - Bit 7 se ajusta su el registro LFN es una entrada de un nombre largo borrado.
10 BYTEs	5 caracteres UNICODE, primera parte del LFN.
1 BYTE	Atributo – Este campo contiene el valor especial 0Fh, que indica una entrada LFN.
1 BYTE	Reservado.
1 BYTE	Checksum de la entrada del nombre corto, usado para validar la entrada LFN.
12 BYTES	6 caracteres UNICODE, segunda parte LFN.
1 WORD	Número de cluster initial, que es siempre cero en entradas LFN.
4 BYTES	2 caracteres UNICODE, tercera parte LFN.

La Figura 1.13 muestra todas las entradas del directorio para el archivo, que tiene el nombre largo "The quick brown.fox". El nombre largo esta en Unicode, así en *Thequi~.fox* cada carácter del nombre utiliza 2 bytes en la entrada del directorio. El campo de atributos para la entrada de nombre largo tiene el valor `0x0F`, y para el nombre corto el valor `0x20`.

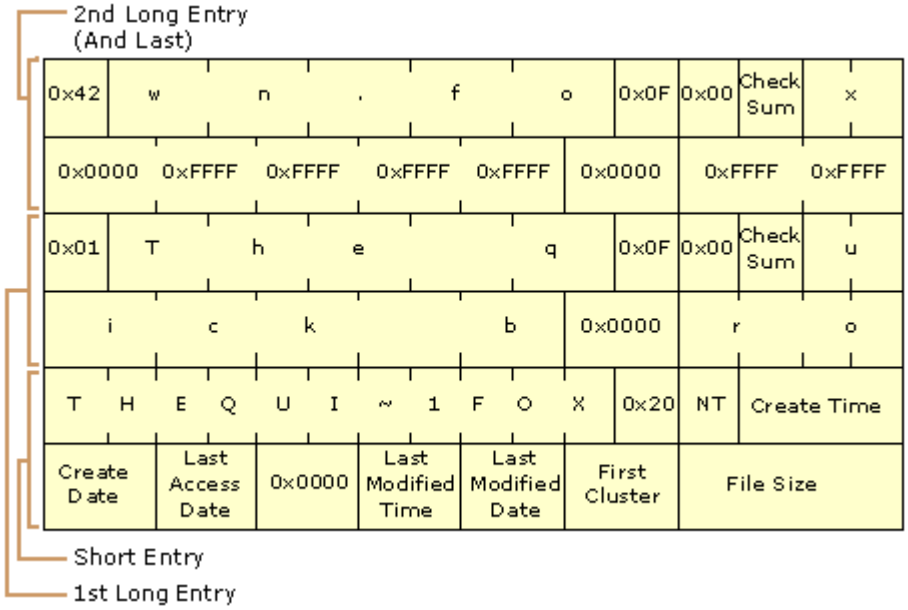


Figura 1.13.- Entradas de directorio para nombre largos (extraída de Microsoft).

- **Tamaño de la FAT**

El tamaño del área de datos se determina utilizando una enorme seudo-fórmula. La vamos a divir en sus partes, para que no sea tan grande.

```
Sectores_Raiz = Entradas_Directorio_Raiz * 32 / Bytes_por_Sector
Sectores_FAT = Número_de_FATs * Sectores_por_FAT
Sectores_Datos=Sectores_Totales-(Sectores_Reservados+Sectores_FATs
                +Sectores_Raiz)
Total_de_Cluster = Sectores_de_Datos / Sectores_por_Cluster
```

Para localizar el sector de inicio del área de datos podemos usar la siguiente fórmula:

```
Inicio_Area_Datos=Sectores_Reservados+Sectores_FAT+Sectores_Root
```

En volúmenes FAT32 el directorio raíz se construye como como una lista de cluster más (en versiones anteriores ocupaba siempre los cluster consecutivos a la zona de la FAT). Un campo en el Boor Record nos indica el número de cluster inicial de este directorio. Un vez que tenemos este número podemos obtener fácilmente el número inicial para la FAT32 de la forma:

```
Sector_inicio_raíz=((Cluster_Raiz-2)*Sectores_Por_Cluster)
                    +Inicio_Area_Datos
```

que es la misma forma de encontrar el número de sector de inicio de un archivo o directorio.

Trabajos en Grupo 1.6:Sistema de archivos FAT.
Objetivo formativo: Revisar algunos de los conceptos vistos en el tema. **Tarea del grupo**: Resolver el siguiente ejercicio: La Figura representa una hipotética FAT que gestiona 18 bloques de datos, que aparecen

como índices de la estructura. Por otra parte se representa una entrada de directorio, que por simplicidad suponemos tiene unicamente los siguinentes campos: nombre del archivo, tipo de archivo (F= archivo, D=directorio), la fecha de creación y el número del bloque inicial.

Entrada de directorio

Nombre	Tipo	Fecha	Nª Bloque
DATOS	F	8-2-90	3

FAT

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
		15															

Suponiendo que el tamaño de bloque es de 512 bytes, que marcamos con un * el último bloque de un archivo, y que todos los campos blancos indican que estan libres, represente los siguiente:

a) Creación del archivo “Datos1” el 1-3-90 con 10 bytes de contenido.
b) Creación del archivo “Datos2” el 2-3-90 con 1200 bytes.
c) El archivo “Datos”, ya creado inicialmente, aumenta en 2 bloques su tamaño inicial.
d) Creamos un directorio, denominado “Midirec” con 1 bloque de datos el 3-3-90.
e) Creamos el archivo “Cartas” el 13-3-90 con un tamaño de 2Kbytes.

Criterio de éxito: Cualquier miembro del grupo podrá demostrar, en un resumen oral o escrito, o mediante respuestas a preguntas del profesor, que ha alcanzado los objetivos formativos propuestos.

Duración: 25 minutos.

Trabajos en Grupo 1.7: Estructura en disco del sistema FAT.

Objetivo formativo: Conocer cómo se almacena en disco la información.
Tareas del grupo: Suponga la estructura en disco simplificada de un sistema FAT que aparece en la Figura de abajo. Supongamos el el sistema de archivos esta inicialmente vacío. Si a partir de aquí creamos el siguiente archivo *c:\directorio1\archivo1*. Representar cómo queda almacenada esta información en disco, es decir, cómo quedaría la FAT y cómo los bloques de datos.

Arranque	FAT	Bloques de datos

Criterio de éxito: Cualquier miembro del grupo podrá demostrar, en un resumen oral o escrito, o mediante respuestas a preguntas del profesor, que ha alcanzado los objetivos formativos propuestos.

Duración: 25 minutos.

Trabajos en Grupo 1.8: Características del sistema FAT32.
Objetivo formativo: Explicación de problemas reales relacionados con la FAT32. **Tareas del grupo**: Los sistemas Windows que soportaban sistemas de archivo FAT32 traían una utilidad, denominada *cvt.exe*, para convertir un sistema FAT16 en FAT32. Cuando pasamos esta utilidad a un sistema de archivos en uso podemos observar que el espacio libre del disco se duplica, e incluso triplica, respecto del anteriormente existente. Podeis dar una explicación razonada para este hecho. **Criterio de éxito**: Cualquier miembro del grupo podrá demostrar, en un resumen oral o escrito, o mediante respuestas a preguntas del profesor, que ha alcanzado los objetivos formativos propuestos.
Duración: 25 minutos.

Trabajos en Grupo 1.9: Calculo del tamaño de la FAT.
Objetivo formativo: Conocer el funcionamiento de FAT a través del cálculo de su tamaño. **Tareas del grupo**: Una de las acciones que debe de llevar a cabo un programa de formateo de un sistema FAT es calcular el tamaño que debe reservar para la FAT de un sistema de archivos teniendo en cuenta el tamaño del disco, el tamaño del bloque y el tipo de FAT a construir. Lo que se propone en este ejercicios es que calculéis de forma exacta (no estimada) cual debe ser el tamaño de la FAT para un disco que tiene 100MB de tamaño, los bloques son de 1KB y los punteros para direccionar son de 32 bits. Para ellos suponed un sistema simplificado en el que el disco solo contiene la FAT y bloques de datos. También, deberéis reflexionar sobre que esquema de asignación utiliza la FAT para gestionarse ella misma (no el que utiliza para asignar bloques de datos). **Criterio de éxito**: Cualquier miembro del grupo podrá demostrar, en un resumen oral o escrito, o mediante respuestas a preguntas del profesor, que ha alcanzado los objetivos formativos propuestos.
Duración: 25 minutos.

9.3.2 El sistema de archivos Ext2

El sistema de archivos *Second Extended Filesystem* (Ext2) es el sistema nativo de Linux y prácticamente todos los sistemas Linux lo utilizan. Veremos las características generales del sistema y describiremos las estructuras de datos de disco que lo constituyen. En el Tema 2, volveremos sobre el para ver las estructuras de datos en memoria.

Lectura 1.8: El sistema de archivos en Unix.
Lee el apartado 12.7 “Gestión de ficheros en Unix” da una visión muy general de los puntos que tratamos con más detalle en este apartado. No leas de momento el apartado 12.8, ya que lo estudiaremos en el Tema 2.
Tiempo estimado: 10 minutos

- ***Características generales***

Cada sistema operativo tipo Unix posee su propio sistema de archivos. Aunque todos ellos presentan una API conforme POSIX (parte de la cual estudiaremos en el Módulo I de prácticas), cada uno de ellos se implementa de manera diferente.

Las principales características que contribuyen a la eficiencia de Ext2 son:

a) Al crear el sistema de archivos con `mke2fs`, el administrador puede elegir el tamaño de bloque (desde 1KB hasta 4KB), dependiendo de la longitud media esperada de los archivos. Por ejemplo, un bloque de 1KB es preferible cuando la longitud media es menor de uno miles de bytes debido a que produce memos fragmentación interna. Por otro lado, el tamaño grande de bloque es preferible para archivos mayores de miles de bytes dado que producen menos transferencias de disco.

b) También se puede elegir cuantos inodos tiene en una partición, dependiendo del número de archivos almacenados en él. Esto maximiza el uso efectivo del espacio utilizable de disco.

c) El sistema de archivos particiona los bloques de disco en grupos. Cada grupo incluye bloques de datos e inodos almacenados en pistas adyacentes. Gracias a esta estructura, los archivos en un único grupo de bloques pueden ser accedidos con un tiempo de búsqueda medio menor.

d) El sistema de archivos preasigna bloques de datos de disco a archivos regulares antes de que estos se utilicen. Así, cuando un archivo incrementa su tamaño, ya dispone de varios bloques reservados en posiciones físicas adyacentes, reduciendo la fragmentación del archivo.

e) Soporta enlaces simbólicos rápidos. Si el nombre de camino del enlace simbólico tiene 60 bytes o menos, se almacena en el inodo y así puede traducirse sin leer un bloque de datos.

Es más, Ext2 tiene otras características que lo hacen más robusto y flexible:

a) Una implementación cuidadosa de la estrategia de actualización de archivos que minimiza el impacto de las caídas del sistema. Por ejemplo, cuando se crea un nuevo enlace duro para un archivo, primero se incrementa el contador de enlaces duros en el inodo en disco, y a continuación se añade el nuevo nombre en el directorio. De esta forma, si se produce una fallo hardware después de la actualización del inodo pero antes de cambiar el directorio, el directorio esta consistente, aunque el contador de enlaces sea incorrecto. Borrar el archivo no produce resultados catastróficos, aunque los bloques de datos del archivo no pueden ser automáticamente reclamados. Si se hiciera la reserva (cambiando el directorio antes de actualizar el inodo), el mismo fallo hardware produciría inconsistencias peligrosas: el borrado del enlace duro original eliminaría los bloques de datos del disco, aunque la entrada del directorio se refiera a un inodo que no exista. Si este inodo se utiliza posteriormente para otro archivo, la escritura en la entrada del directorio pasado corrompería el nuevo archivo.

b) Soporte para comprobaciones automáticas de consistencia sobre el estado del sistema de archivos en el arranque del sistema. Estas comprobaciones se realiza mediante el programa */sbin/e2fsck*, que puede activarse no solo tras caídas del sistema, sino después de un número predefinido de montajes (se incrementa un contador después de cada operación de montaje), o después de cierta cantidad de tiempo transcurrida desde la comprobación más reciente.

c) Soporte de archivos inmutables (no pueden ser modificados) y para archivos solo-añadir (solo podemos añadir datos al final de archivo). Ni el superusuario puede sobrepasar estas clases de protección.

d) Compatibilidad con las semánticas de los ID de grupo de un nuevo archivo de los sistemas de archivos de System V y BSD. In System V un nuevo archivo asume el ID de Grupo del proceso que lo crea; en BSD, un nuevo archivo hereda el ID de Grupo

del directorio que lo contiene. Ext2 incluye una opción de montaje que especifica que semántica utilizar.

Se están considerando otras características adicionales para la siguiente versión principal de Ext2, el Ext3. Algunas de estas se han codificado y están disponibles como parches externos. Otras solo esta en diseño, pero en algunos casos se han añadido campos en el inodo Ext2 para ellas. Las características más significativas tenemos:

- Fragmentación de bloques – para reducir la fragmentación, permite almacenar en un mismo bloque fragmentos de diferentes archivos.
- Manejo de archivos comprimidos y encriptados – Estas nuevas opciones, que deben especificarse al crear el archivo, permiten almacenar versiones comprimidas y/o encriptadas de los archivos en disco.
- Borrado lógico – Una opción *undelete* permitirá a los usuarios recuperar fácilmente, si es necesario, el contenido de un archivo eliminado previamente.
- Journaling – el *journaling* (diario) permite evitar las comprobaciones que se realizan automáticamente cuando se desmonta abruptamente un sistema de archivos que consumen mucho tiempo, como por ejemplo, cuando se produce una caída del sistema.

- ***Estructuras de datos de disco***

El primer bloque en cualquier partición Ext2 nunca es manejado por el sistema de archivos ya que esta reservado para el sector de arranque. El resto de la partición se divide en *grupos de bloques*, cada uno de los cuales tiene la estructura mostrada en la Figura 1.14. Como podemos observar en ella, algunas de las estructuras de datos deben encajar exactamente en un bloque mientras que otras requieren más de uno. Todos los grupos de bloques en el sistema de archivos tienen el mismo tamaño y se almacenan secuencialmente, así el kernel puede derivar la posición de un grupo de un bloque en disco directamente se su índice entero.

Los grupos de bloques reducen la fragmentación, dado que el kernel intenta mantener, si es posible, los bloques de datos de un archivo en el mismo grupo de bloques. Cada bloque en el grupo contiene una de las siguientes piezas de información:

a) Una copia del superbloque del sistema de archivos.
b) Una copia del grupo de descriptores de grupos de bloques.
c) Un mapa de bits de bloque.
d) Un grupo de inodos.
e) Un mapa de bits de inodos.
f) Un trozo de datos pertenecientes a archivo; esto es, bloques de datos (si un bloque no contiene ninguna información útil, se dice que esta libre).

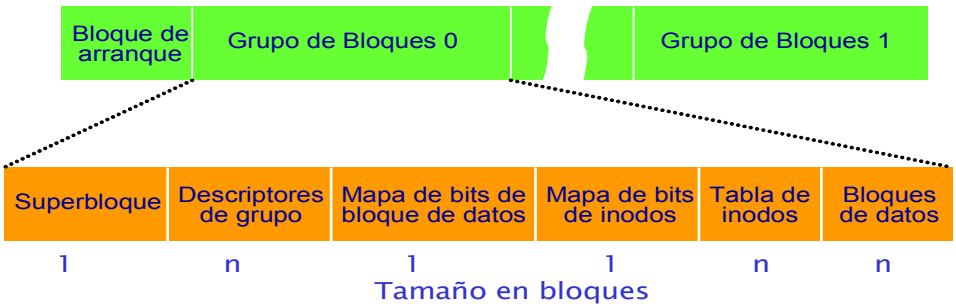


Figura 1.14.- Estructura de una partición Ext2 y de un grupo de bloques Ext2.

Como podemos ver en la Figura 3.1, tanto el superbloque como los descriptores de grupo están duplicados en cada grupo de bloques. Sólo el superbloque y los descriptores de grupos incluidos en el grupo de bloques 0 son utilizados por el kernel, mientras que las demás copias se dejan sin modificar; de hecho, nunca las mira. Cuando el programa */sbin/e2fsck* realiza una comprobación de consistencia, referencia el superbloque y los descriptores de grupos de bloques de grupo 0, después, los copia en el resto de grupos de bloques. Si se produce una corrupción de datos, y el superbloque y los descriptores de grupos del grupo 0 se hacen inválidos, el administrador puede indicar a */sbin/e2fsck* que referencie las viejas copias de otros grupos diferentes del 0. Usualmente, las copias redundantes tienen suficiente información para permitir al programa retornar la partición a un estado consistente.

El número de grupos de bloques depende tanto del tamaño de la partición como del tamaño de bloque. La principal restricción se debe a que el mapa de bits de bloque, que se utiliza para identificar los bloques dentro de un grupo que están en uso o libres, debe almacenarse en un único bloque. Por tanto, cada grupo de bloques debe tener como máximo *8*b* bloques, donde *b* es el tamaño de bloque en bytes. Así, el número total de grupos de bloques es aproximadamente *s/(8*b)*, donde *s* es el tamaño de la partición en bloques. Como ejemplo, consideremos una partición Ext2 de 8 GB con bloques de 4 KB de tamaño. En este caso, cada mapa de bits de bloques de 4KB describe 32 KB de bloques de datos, es decir, 128 MB. Por tanto, como máximo se necesitan 64 grupos de bloques. Claramente, a menor tamaño de bloque, mayor número de grupos de bloques.

- **Superbloque**

El superbloque de una partición Ext2 se almacena en una estructura de tipo `ext2_super_block`, cuyos campos se muestran en la Tabla 1.3.

Tabla 1.3.- Campos del superbloque Ext2.

Campo	Descripción
s_inode_count	Número total de inodos
s_blocks_count	Tamaño del sistema de archivos en bloques
s_r_blocks_count	Número de bloques reservados
s_free_blocks_count	Contador de bloques libres
s_free_inode_count	Contador de inodos libres
s_first_data_block	Número de primer bloque útil (siempre 1)
s_log_block_size	Tamaño de bloque
s_log_frag_size	Tamaño de fragmento
s_blocks_per_group	Número de bloques por grupo
s_frag_per_group	Número de fragmentos por grupo
s_inodes_per_group	Número de inodos por grupo
s_mtime	Tiempo del último montaje
s_wtime	Tiempo de la última escritura
s_mnt_count	Contador de operaciones de montaje
s_max_mnt_count	Número de montajes antes de comprobación
s_magic	Signatura mágica
s_state	Indicador de estado
s_errors	Comportamiento cuando detecta errores
s_mirror_rev_level	Nivel de revisión menor
s_lastcheck	Tiempo de la última comprobación
s_checkinterval	Tiempo entre comprobaciones
s_creator_os	SO donde se creo el sistema de archivos

s_rev_level	Nivel de revisión
s_def_resuid	UID por defecto para bloques reservados
s_def_resgid	GID por defecto para bloques reservados
s_first_ino	Número del primer inodo no reservado
s_inode_size	Tamaño del inodo en disco
s_block_group_nr	Nº de grupo de bloque de este superbloque
s_feature_compact	Mapa de bits de características compatibles
s_feature_incompat	Mapa de bits de características incompatibles
s_feature_ro_compat	Idem compatibles solo-lectura
s_uuid	Identificador de sistema de archivos de 128 bit
s_volume_name	Nombre de volumen
s_last_mounted	Pathname del último punto de montaje
s_algorithm_usage_bitmap	Utilizado para compresión
s_prealloc_blocks	Nº de bloques a preasignar
s_prealloc_dir_blocks	Nº bloques a preasignar para directorios
s_padding1	Alineamiento de palabra
s_reserved	Nulos para rellenar 1024 bytes

El campo `s_inodes_count` almacena el número de inodos, mientras que `s_blocks_count` almacena el número de bloques del sistema de archivos Ext2.

El campo `s_log_block_size` expresa el tamaño de bloque como una potencia de 2, utilizando 1024 bytes como unidad. Así, 0 denota bloques de 1024 bytes, 1 de 2048, y así sucesivamente. El campo `s_log_frag_size` es realmente igual a `s_log_block_size`, dado que la fragmentación de bloque no esta aún implementada.

El número de bloques, fragmentos, e inodos en cada grupo de bloques se almacenan en los campos `s_blocks_per_group`,`s_frags_per_group`, y `s_inodes_per_group`, almacenan, respectivamente.

Algunos bloques de disco están reservados al administrador (o algún otro usuario o grupo de usuarios seleccionados por los campos `s_def_resuid` y `s_def_resgid`). Estos bloques permiten al administrador continuar usando el sistema de archivos incluso cuando no hay más bloques libre para los usuarios normales.

Los campos `s_mnt_count`, `s_max_mnt_count`, `s_lastcheck`, y `s_checkinterval` permiten al sistema de archivos Ext2 ser chequeado automáticamente en el arranque. Esto campos provocan que */sbin/e2fsck* se ejecute después de un número predefinido de operaciones de montaje, o cuando ha transcurrido cierto periodo de tiempo desde la última comprobación (ambos pueden realizarse conjuntamente). La comprobación de consistencia también se activa en el arranque si el sistema de archivos no fue desmontado limpiamente (p. ej. caída del sistema), o cuando el kernel descubre algún error en él. El campo `s_state` almacena 0 si el sistema de archivos esta montado o no fue desmontado limpiamente, 1 si fue desmontado limpiamente, y 2 si contiene errores.

- **Descriptores de grupo y mapa de bits**

Cada grupo de bloques tiene su propio descriptor, una estructura `ext2_group_desc` cuyos campos se ilustran en la Tabla 1.4.

Tabla 1.4.- Campos del descriptor de grupo de Ext2.

Campo	Descripción
`bg_block_bitmap`	Número de bloque del mapa de bits de bloques
`bg_inode_bitmap`	Número de bloque del mapa de bits de inodos

b_inode_table	Nº bloque del 1º bloque de la tabla de inodos
bg_free_blocks_count	Nº de bloques libres en el grupo
bg_free_inodes_count	Nº de inodos libres en el grupo
bg_used_dirs_count	Nº de directorios en el grupo
bg_pad	Alineamiento de palabra
bg_reserved	Nulos de relleno de 24 bytes

Cuando se asignan nuevos bloques de inodos y de datos se utilizan los campos `bg_free_blocks_count`, `bg_free_inodes_count`, y `bg_used_dirs_count`. Estos campos determinan el bloque más aconsejable en el que asignar cada estructura de datos. Los mapas de bits son secuencias de bits: un 0 especifica que el correspondiente bloque de inodo o de datos esta libre, y un 1 especifica que esta en uso. Dado que cada mapa de bits debe almacenarse dentro de un único bloque y como el tamaño de bloque puede ser 1024, 2048, o 4096 bytes, un único mapa de bits describe el estado de 8192, 16.384, o 32.768 bloques.

- **Tabla de inodos**

La tabla de inodos consta de una serie de bloques consecutivos, cada uno de los cuales contiene un número de inodos predefinidos. El número de bloque del primer bloque de la tabla de inodos se almacena en el campo `bg_inode_table` del descriptor de grupo.

Todos los inodos tienen el mismo tamaño, 128 bytes. Un bloque de 1KB contiene 8 inodos, mientras que uno de 4KB tiene 32 inodos. Para hacernos una idea de cuantos bloques ocupa la tabla de inodos, dividimos el número total de inodos en un grupo (almacenado en `s_inodes_per_group` del superbloque) por el número de inodos por bloque.
Cada inodo Ext2 es una estructura `ext2_inode` cuyos campos aparecen en la Tabla 1.5.

Tabla 1.5.- Campos de un inodo Ext2 en disco.

Campo	Descripción
i_mode	Tipo de archivo y derechos de acceso
i_uid	Identificador del propietario
i_size	Longitud del archivo en bytes
i_atime	Tiempo de último acceso al archivo
i_ctime	Tiempo de último cambio del inodo
i_mtime	Tiempo del último cambio del archivo
i_dtime	Tiempo del borrado del archivo
i_gid	Identificador de grupo
i_links_count	Contador de enlace duros
i_blocks	Nº de bloques de datos del archivo
i_flags	Indicadores del archivo
osd1	Información específica del SO
i_block	Punteros a bloques de datos
i_version	Versión de archivo (para NFS)
i_file_acl	Lista de controlo de acceso del archivo
i_dir_acl	ACL de directorio
i_faddr	Dirección de fragmento
osd2	Información específica del SO

Muchos campo relacionados con las especificaciones POSIX son similares a los campos correspondientes del objeto inodo VFS y fueron discutidos en la sección “Objetos indodo” del apartado 2. Los restantes hacen referencia a la implementación específica de Ext2 y tratan principalmente con la asignación de bloques.

En particular, `i_size` almacena la longitud efectiva del archivo en bytes, mientras que `i_blocks` almacena el número de bloques de datos (en unidades de 512 bytes) que han sido asignados al archivo.

Los valores de `i_size` y `i_blocks` no están necesariamente relacionados. Dado que un archivo se almacena en un número entero de bloques, un archivo no vacío recibe al menos un bloque de datos (dado que la fragmentación no esta aún implementada) y `i_size` puede ser menor que 512*`i_blocks`. Por otra parte, como veremos en la sección de "Archivos con agujeros" en este apartado, un archivo puede contener agujeros. En este caso, `i_size` puede ser mayor que 512*`i_blocks`.

El campo `i_blocks` es una matriz de punteros `EXT2_N_BLOCKS` (normalmente 15) a bloques utilizados para identificar los bloques de datos asignados al archivo (ver sección "Direccionamiento de bloques de datos").

Los 32 bits reservados para el campo `i_size` limitan el tamaño de un archivo a 4GB. Realmente, el bit más significativo de este campo no se utiliza, así que el máximo tamaño para un archivo es de 2GB. Sin embargo, ext2 incluye un "truco sucio" que permite archivos mayores en arquitecturas de 64-bits. En arquitecturas de 32-bits se puede acceder a archivos mayores si los abrimos con el indicador `O_LARGEFILE`.

Recordar que el modelo VFS requiere que cada archivo tenga un número de inodo diferente. En Ext2, no hay necesidad de almacenar el número de inodo en el archivo en disco debido a que su valor se puede derivar del número de grupo de bloques y la posición relativa dentro de la tabla de inodos. Como ejemplo, supongamos que cada grupo de bloques tiene 4096 inodos y queremos conocer la dirección en disco del inodo 13021. En este caso, el inodo pertenece a tercer grupo de bloques y su dirección de disco se almacena en la entrada 733 de la correspondiente tabla de inodos. Como podemos ver, el número de inodo es sólo una la clave utilizada por las rutinas Ext2 para recuperar rápidamente el descriptor de inodo adecuado en disco.

- **Atributos extendidos de un inodo**

El formato de un inodo ext2 es una clase de camisa de fuerza para los diseñadores de sistemas de archivos. La longitud de un inodo debe ser una potencia de 2 para evitar fragmentación interna en los bloques que contienen la tabla de inodos. Realmente, la mayoría de los 128 caracteres de un inodo ext2 esta llenos de información, y queda poco espacio para nuevos campos. De otra parte, expandir el inodo a 256 bytes podría ser un desperdicio, además de introducir incompatibilidades con otros sistemas ext2.

Para sobrepasar esta limitación se ha introducido los *atributos extendidos*. Estos atributos se almacenan en un bloque de disco fuera del inodo. El campo `i_file_acl` apunta al bloque que contiene los atributos extendidos. Diferentes inodos que contienen el mismo conjunto de atributos extendidos pueden compartir el mismo bloque.

Existen varias llamadas al sistema utilizadas para manipular los atributos extendidos de un archivo. La funciones *setxattr()*, *lsetxattr()*, y *fsetxattr()* asignan un atributo. Las funciones *getxattr()*, *lgetxattr()* y *fgetxattr()* devuelven el valor de un atributo. Podemos listar todos los atributos extendidos con *listxattr()*, *llistxattr()* y *flistxattr()*. Finalmente, *removexattr()* y *lremovexattr()*, eliminan un atributo extendido de un archivo.

- **Listas de control de acceso**

Las listas de control de acceso (que veremos con más detalle en el tema de "Seguridad y protección") se introdujeron para mejorar el mecanismo de protección de archivos. En lugar de clasificar a los usuarios en tres clases, propietario, grupo y otros, se puede asociar con cada archivo una *Lista de Control de Acceso* (ACL). Con esta lista se pueden especificar para cada archivos los usuarios (o grupos de usuarios) y los privilegios dados a esos usuarios.

Linux 2.6 soporta completamente ACLs haciendo uso de los atributos extendidos de los inodos. Las funciones de la biblioteca *chacl()*, *setfacl()*, y *getfacl()*, que permiten manipular las ACLs, descansan en las llamadas al sistema *setxattr()* y *getxattr()*.

- **Cómo utilizan los bloques de disco varios tipos de archivos**

Los diferentes tipos de archivos reconocidos por Ext2 (archivos regulares, cauces, etc.) utilizan los bloques de datos de diferentes formas. Algunos archivos no almacenan datos y por tanto no necesitan bloques de datos. En este apartado comentaremos las necesidades de almacenamiento de cada tipo.

- **Archivos regulares**

Los archivos regulares son el tipo más extendido y reciben casi toda la atención de este tema. Estos archivos solo necesitan bloques de datos cuando comienzan a tener datos. Cuando se crea, un archivo regular esta vacío y no requiere bloques de datos, también puede vaciarse con la llamada al sistema `truncate()`. Ambas situaciones son comunes, por ejemplo, cuando invocamos una orden del shell que incluye una cadena >*nombrearchivo*, el shell crea un archivo vacío o trunca uno existente.

- **Directorios**

Ext2 implementa los directorios como una clase especial de archivos cuyos bloques de datos almacena nombres de archivos con su correspondiente número de inodo. En concreto, tales bloques de datos contienen una estructura de tipo `ext2_dir_entry`. Los campos se muestran en la Tabla 1.6. La estructura tiene una longitud variable, dado que el campo `name` es una matriz de longitud variable hasta `EXT2_NAME_LEN` caracteres (normalmente 255). Es más, por razones de eficiencia, la longitud de una entrada de directorio es siempre múltiplo de 4, y, por tanto, se añade un nulo (\0), si es necesario, para rellenar el final del nombre del archivo. El campo `name_len` almacena la longitud real del nombre de archivo (Figura 1.15).

Tabla 1.6.- Campos de la entrada de directorio Ext2.

Campo	Descripción
`inode`	Número de inodo
`rec_len`	Longitud de la entrada de directorio
`name_len`	Longitud del nombre de archivo
`file_type`	Tipo de archivo
`name`	Nombre del archivo

El campo `file_type` almacena un valor que especifica el tipo de archivo (ver Figura 2). El campo `rec_len` puede interpretarse como un puntero a la siguiente entrada de directorio válida: es el desplazamiento que hay que añadir a la dirección de inicio de la entrada de directorio para obtener la dirección de inicio de la siguiente entrada válida. Al objeto de borrar una entrada de directorio, es suficiente poner el campo `inode` a 0 e incrementar adecuadamente el valor del campo `rec_len` de la entrada anterior válida. Debemos leer el campo `rec_len` de la Figura 3.2 cuidadosamente, ya que por ejemplo, si borrásemos la entrada arc deberíamos poner el campo `inode` a 0, para indicar que la entrada esta libre, y el campo `rec_len` deberíamos ajustarlo a 16+12 (longitudes de la entrada *homes* y *arc*).

- **Enlaces simbólicos**

Como indicábamos antes, si el nombre de camino del enlace simbólico tiene un máximo de 60 caracteres se almacena en el campo `i_block` del inodo, que consta de una matriz de 15 enteros de 4 bytes, y no necesita bloques de datos. Sin embargo, si el nombre de camino es mayor de 60 caracteres solo se necesita un bloque de datos.

- **Archivos de dispositivos, cauces, y socket**

Para estas clases de archivos no se necesitan bloques de datos. Toda la información necesaria se almacena en el inodo.

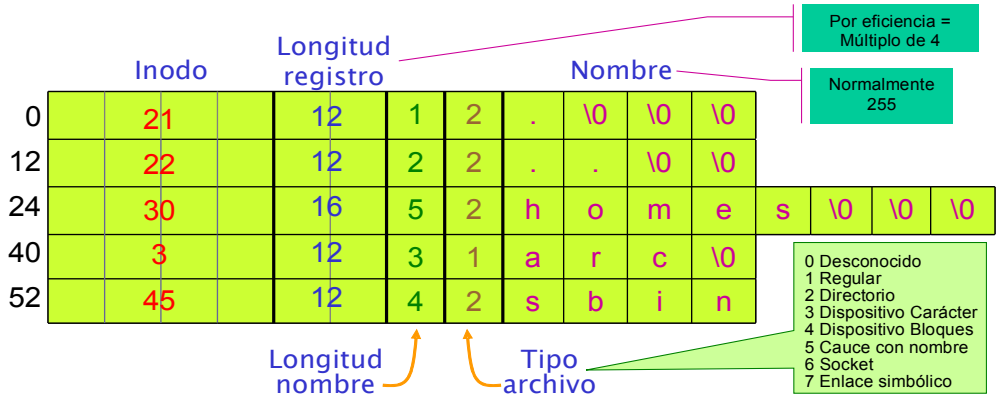


Figura 1.15.- Un ejemplo de un directorio Ext2.

- ***Creando un sistema de archivos Ext2***

No es lo mismo formatear una partición de disco o disquete que crear un sistema de archivos. El formateo permite al manejador de disco leer o escribir bloques sobre el disco, mientras crear el sistema de archivos significa establecer las estructuras descritas en detalle anteriormente vistas.

Los discos duros actuales vienen formateados de fabrica y no necesitan ser formateados, los disquetes pueden ser formateados con el programa de utilidad */usr/bin/superformat* o con *fdformat*.

Los sistemas de archivos Ext2 se crean con la utilidad */sbin/mke2fs*; esta supone las siguientes opciones por defecto, que se pueden modificar con las correspondientes opciones de la línea de órdenes:

- Tamaño de bloque: 1024 bytes.
- Tamaño de fragmento: tamaño de bloque (fragmentación no implementada).
- Número de inodos asignados: uno por cada grupo de 8192 bytes.
- Porcentaje de bloques reservados: 5%

El programa realiza las siguientes acciones:

1. Inicializa el superbloque y los descriptores de grupo.
2. Opcionalmente, comprueba si la partición contiene bloques defectuosos: si es así, crea una lista de bloques defectuosos.
3. Por cada grupo de bloques, reserva todos los bloques de disco necesarios para almacenar el superbloque, los descriptores de grupo, la tabla de inodos, y los dos mapas de bits.

4. Inicializa los mapas de bits y los datos de los mapas de bits de cada grupo de bloques a 0.
5. Inicializa la tabla de inodos para cada grupo de bloques.
6. Crea el directorio raíz.
7. Crea el directorio *lost+found*, que es usada por */sbin/e2fsck* para enlazar los bloques defectuosos y perdidos que encuentra.
8. Actualiza el mapa de bits de inodo y de bloques de datos del grupo de bloques en el que se han creado los dos directorios anteriores.
9. Si hay, agrupa los bloques defectuosos en el directorio *lost+found*.

Como ejemplo, vamos a considerar cómo */sbin/mke2fs* inicializa un disquete Ext2 de 1.4 MB con las opciones por defecto.
Una vez montado, esta aparecerá a VFS como un volumen consistente en 1390 bloques, cada uno con 1024 bytes de longitud. Para examinar los contenidos del disco, podemos ejecutar la orden Unix:

```
$ dd if=/dev/fd0 bs=lk count=1400 | od -tx1 -Ax > /tmp/dump_hex
```

para obtener en el directorio */tmp* un archivo con el volcado de contenido en hexadecimal de los contenidos del disco.
Mirando este archivo, podemos ver que, debido a la limitada capacidad del disco, es suficiente un único descriptor de grupo. Podemos observar que el número de bloques reservados es de 72 (5% de 1440) y que, de acuerdo con las opciones por defecto, la tabla de inodos debe incluir un inodo or cada 4096 bytes, es decir, 360 inodos almacenados en 45 bloques.
La Tabla 1.7 resume cómo se crea el sistema de archivos Ext2 en un disquete cuando se seleccionan las opciones por defecto.

Tabla 1.7:- Asignación de bloques Ext2 de un disquete.

Bloque	Contenido
0	Bloque de arranque
1	Superbloque
2	Bloque con un solo descriptor de grupos de bloques
3	Mapa de bits de bloques de datos
4	Mapa de bits de inodos
5-27	Tabla de inodos: hasta el 10,reservados; el 11 para *lost+found*; 12-128
28	libres.
29	Directorio raíz (incluye ., .., y *lost+found*)
30-40	Directorio *lost+found* (incluye . y ..)
41-1429	Bloques reservados preasignados para *lost+found*
	Bloques libres

- ***Gestión del espacio de disco***

El almacenaje de un archivo en disco difiere de la forma en que el programador ve el archivo en dos aspectos: los bloques están dispersos sobre el disco (si bien el sistema de archivos intenta mantenerlos ordenados secuencialmente para mejorar el tiempo de acceso), y los archivos pueden parecer al programador mayores de lo que son debido a que el programa puede introducir agujeros dentro de él.

Ahora, examinaremos como Ext2 gestiona el espacio en disco, es decir, cómo asigna y desasigna inodos y bloques de datos. Debemos abordar dos problemas básicos:

a) La gestión de espacio deber hacer todo el esfuerzo para evitar la *fragmentación de archivos*, es decir, el almacenamiento físico de un archivo en pequeñas piezas localizadas en bloques de disco no adyacentes. La fragmentación de archivos aumenta el tiempo medio de operaciones de lectura secuencial sobre el archivo, dado que la cabeza del disco debe reposicionarse frecuentemente durante la operación.
b) La gestión de espacio debe ser eficiente en tiempo, es decir, el kernel debe ser capaz de derivar rápidamente del desplazamiento de archivo el número de bloque lógico correspondiente en la partición Ext2. Haciendo esto, el kernel debe limitar en la medida de lo posible el número de accesos para direccionar tablas almacenadas en disco, dado que cada acceso intermedio aumenta considerablemente el tiempo medio de acceso a un archivo.

- **Creación de inodos**

La función que crea un inodo Ext2 en disco selecciona cuidadosamente el grupo de bloque que contiene al inodo de forma que se distribuyan los directorios no relacionados entre diferentes grupos, y a la vez, poner los archivos en el mismo grupo que sus directorios padres. Para equilibrar el número de archivos regulares y directorios en un grupo de bloques, ext2 introduce un parámetro `debt` para cada grupo.

La función distribuye los directorios y archivos de la siguiente forma:

- Directorios
  a) Los directorios que tienen como padre al raíz del sistema de archivos deben esparcirse entre todos los grupos de bloques. Así, la función busca los grupos de bloques mirando los grupos que tienen un número de inodos libres y bloques libres por encima de la media. Si no existe un grupo con estas características, pasa a punto 2c.
  b) Los directorios anidados, que no tienen al raíz como padre, deben ponerse en el grupo del padre si se satisfacen las siguientes reglas:
     1. El grupo no contiene demasiados directorios.
     2. El grupo se queda con suficientes inodos libres.
     3. El grupo tiene un `debt` pequeño (el `debt` de un grupo de bloques se almacena en el superbloque, se incrementa cada vez que se añade un nuevo directorio y se decrementa cada vez que se añade otro tipo de archivo).
     4. Si el grupo padre no satisface estas reglas, se toma el primer grupo que las satisfaga. Si no existe ninguno, pasa al punto c.
  d) Esta es una regla "alternativa", para usarse si no se encuentra el grupo adecuado. La función comienza con el grupo que contiene al directorio padre y selecciona el primer grupo de bloques que tiene más inodos libres que la media de inodos libres por grupo de bloques.

- Archivos que no son directorios

  a) Si el nuevo inodo no es un directorio, se selecciona el grupo empezando desde el que tiene el directorio padre y lo asigna en un grupo de bloques que tenga un inodo libre. Selecciona el grupo empezando desde el que contiene el directorio padre y alejandose de él, más precisamente:
     1. Realiza una búsqueda logarítmica empezando por el grupo de bloques que incluye al directorio padre. El algoritmo busca log(*n*) grupos de bloques, donde *n* es el número total de grupos de bloques, saltando hacia delante hasta que encuentra un grupo de bloques disponible. Por ejemplo, si denominamos como *i* al número de grupo de bloques de inicio, el algoritmo considera los grupos *i mod (n)*, *i+1 mod (n)*, *i+1+2 mod (n)*, *i+1+2+4 mod (n)*, …

2. Si falla la búsqueda logarítmica en encontrar el grupo de bloques con un inodo libre, realiza una búsqueda lineal exhaustiva empezando desde el grupo de bloques que contiene al directorio padre.

- **Direccionando bloques de datos**

Cada archivo regular no vacío consta de un grupo de bloques de datos. Tales bloques pueden referenciarse por la dirección relativa dentro de archivo (su número de bloque de archivo) o su posición dentro de la partición de disco.

Derivar el número de bloque lógico del correspondiente bloque de datos de un desplazamiento *f* dentro del archivo es un proceso de dos etapas:

- Derivar del desplazamiento *f* del número de bloque de archivo, es decir, el índice del bloque que contiene el carácter del desplazamiento *f*.
- Traducir el número de bloque de archivo al correspondiente número de bloque lógico.

Dado que los archivos en Unix no incluyen ningún carácter de control, es fácil derivar el número de bloque de archivo que contienen el *f*-ésimo carácter del archivo: tomar el cociente de dividir *f* entre el tamaño de bloque del sistema de archivos y redondear a la baja al entero más próximo.

Por ejemplo, supongamos un tamaño de bloque de 4KB. Si *f* es menor de 4096, el carácter esta en el primer bloque del archivo, que tiene 0 de número de bloque de archivo. Si es igual o mayor de 4096 y menor de 8192, el carácter esta contenido en el bloque 1, y así, sucesivamente.

Ahora bien, traducir el número de bloque de archivo en el correspondiente número de bloque lógico no es tan directo, ya que los bloques de datos de un archivo Ext2 no están necesariamente juntos en disco.

El sistema de archivos Ext2 debe suministrar un método para almacenar en disco la conexión entre cada número de bloque de archivo y el correspondiente número de bloque lógico. Esta correspondencia, que retorna de las primeras versiones de Unix de AT&T, se implementa parcialmente dentro del inodo. También, están involucrados algunos bloques de datos especializados, que pueden considerarse como una extensión del inodo para gestionar grandes archivos.

El campo `i_block` del inodo en disco es una matriz de `EXT2_N_BLOCKS` componentes que contienen números de bloques lógicos. En la siguiente discusión, asumiremos que `EXT2_N_BLOCKS` es 15. La matriz representa la parte inicial de una estructura de datos mayor, que se ilustra en la Figura 1.16. Como podemos ver en la Figura, los 15 componentes de la matriz son de cuatro tipos diferentes:

1) Los 12 primeros componentes dan los números de bloque lógico de los 12 primeros bloques del archivo, es decir, los bloques con números de bloque de archivo de 0 a 11.
2) El componente de índice 12 contiene el número de bloque lógico de un bloque que representa una matriz de segundo orden de números de bloques lógicos. Corresponde a los bloques de archivo que van desde el 12 a $b/4+11$ donde b es el tamaño del bloque del sistema de archivos (cada número de bloque lógico se almacena en 4 bytes, por eso dividimos por 4 en la formula). Por tanto, el kernel debe buscar en este componente un puntero a un bloque, que a su vez contiene puntero a bloques de datos.
3) El componente 13 contiene el número de bloque lógico que contiene una matriz de segundo orden de números de bloques lógicos; a su vez, las entradas de esta segunda matriz apuntan a una matriz de tercer orden, que almacena los números del bloques lógicos correspondientes a los números de bloques de archivo que van del $b/4+12$ hasta $(b/4)^2+(b/4)+11$.

4) Finalmente, el componente de índice 14 utiliza una triple indirección: las matrices de 4 orden almacenan los número de bloques lógicos correspondientes a los números de bloques de archivos que van del $(b+4)^2+(b/4)+12$ a $(b/4)^3+(b/4)^2+(b/4)+12$.

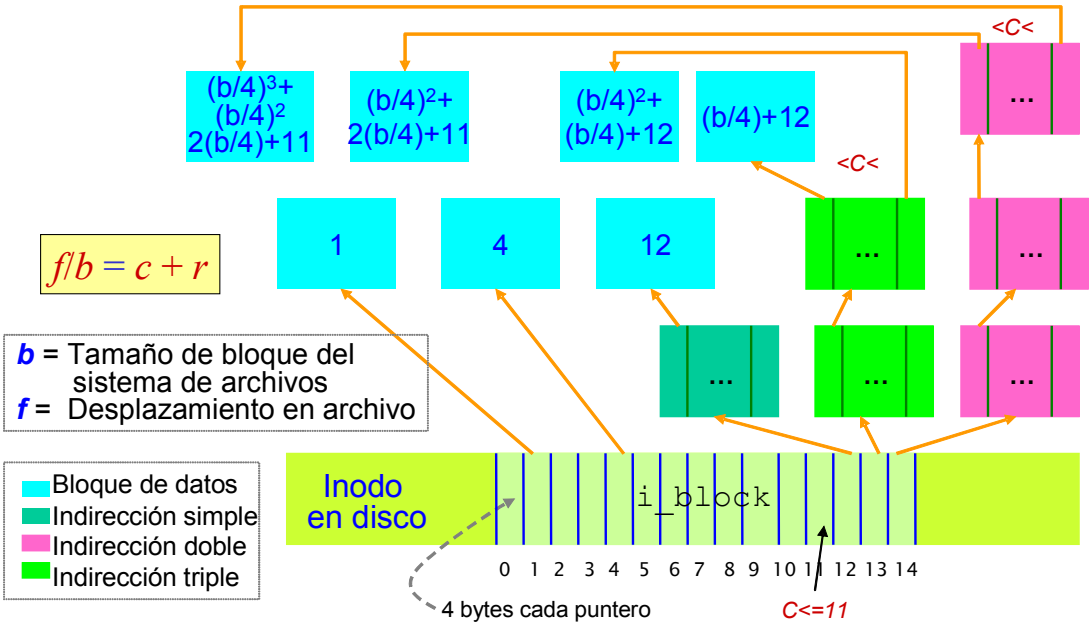


Figura 1.16.- Estructuras de datos utilizadas para direccionar bloques de datos.

Observar como el mecanismo favorece los archivos pequeños. Si el archivo no requiere más de 12 bloques de datos, cualquier dato puede recuperarse en dos accesos de disco: uno para leer el componente de la matriz `i_block` del inodo en disco, y otro para leer el dato solicitado. Para grandes archivos, sin embargo, pueden ser necesarios tres o cuatro accesos a disco. En la práctica, esto es una estimación del peor caso, ya que las cachés de páginas, búferes y dentro, contribuyen significativamente el número de lecturas de disco.
Observar también como el tamaño de bloque del sistema de archivos afecta al mecanismo de direccionamiento, ya que un tamaño de bloque grande permite a Ext2 almacenar más números de bloques lógicos dentro de un único bloque. La Tabla 1.8 muestra el límite superior del tamaño de archivo para varios tamaños de bloques y cada modo de direccionamiento.

Tabla 1.8.- Límites superiores de tamaños de archivos.

Tamaño bloque	Directo	1-indirección	2-indirección	3-indirección
1024	12 KB	268 KB	64.22 MB	16.06 GB
2048	24 KB	1.02 MB	513.02 MB	256.56 GB
4096	48 KB	4.04 MB	4 GB	~4 TB

Trabajos en Grupo 1.10: Direccionamiento de bloques de datos en inodos.

Objetivo formativo: Conocer el esquema de direccionamiento y traducción de bloques en un sistema de inodos.
Tareas del grupo: Supongamos un sistema de archivos tipo Unix simplificado donde el inodo contiene 10 entradas que almacenan punteros directos a bloques de datos, y 2 para indexación multinivel (1 para indexación a un nivel, y otro para indexación a dos niveles). El tamaño de bloque del sistema de archivos es de 1 KB y las direcciones son de 32 bits. Diseña una función que permita localizar el bloque de datos en disco donde se encuenta almacenado un byte genérico de un archivo que se pasa como argumento.
Criterio de éxito: Cualquier miembro del grupo podrá demostrar, en un resumen oral o

escrito, o mediante respuestas a preguntas del profesor, que ha alcanzado los objetivos formativos propuestos.

Duración: 25 minutos.

Trabajos en Grupo 1.11: Estructura en disco de la información de un sistema de inodos.

Objetivo formativo: Conocer cómo se distribuye en disco la información.
Tareas del grupo: Suponga la estructura en disco simplificada de un sistema ext2 que aparece en la Figura de abajo. Supongamos el el sistema de archivos esta inicialmente vacío. Si a partir de aquí creamos el siguiente archivo *c:\directorio1\archivo1*. Representar cómo queda almacenada esta información en disco.

Arranque	Mapa bits bloques	Mapa bits inodods	Tabla de inodos	Bloques de datos

Criterio de éxito: Cualquier miembro del grupo podrá demostrar, en un resumen oral o escrito, o mediante respuestas a preguntas del profesor, que ha alcanzado los objetivos formativos propuestos.

Duración: 25 minutos.

Trabajos en Grupo 1.12: Sistema de archivos ext2.

Objetivo formativo: Conocer las características de un sistema ext2.
Tareas del grupo: Supongamos un sistema de archivos tipo Unix tenemos un archivo regular llamado *Prueba.c* que ocupa 6410 KB. El tamaño del bloque lógico es de 1KB y cada bloque se direcciona utilizando punteros de 32 bits. Suponga que este sistema, un inodo utiliza 10 punteros directos a bloques de datos, 1 para direccionamiento indirecto, 1 para indirección a doble nivel, y 1 para triple nivel de indirección.
a) Si se crean un enlace simbólico y un enlace duro a dicho archivo, ¿cuánto espacio de almacenamiento se gasta para ello?
b) ¿Cuales y cuantos bloques índices se liberan si se borran los últimos 7168 Bytes del archivo *Prueba.c*?
Criterio de éxito: Cualquier miembro del grupo podrá demostrar, en un resumen oral o escrito, o mediante respuestas a preguntas del profesor, que ha alcanzado los objetivos formativos propuestos.

Duración: 25 minutos.

- **Agujeros en archivos**

Una peculiaridad del esquema Un agujero de archivo es una porción de un archivo regular que contiene caracteres nulos y que no son almacenados en ningún bloque de datos en disco. Los agujeros son un característica desde hace mucho tiempo en Unix. Por ejemplo, la siguiente orden crea un archivo en el que el primer bytes es un agujero:

```
% echo -n "X" | dd of=/tmp/hole bs=1024 seek=6
```

Ahora, */tmp/hole* tiene 6145 caracteres (6144 caracteres nulos y un carácter X), si bien el archivo ocupa solo un bloque de datos en disco.

Los archivos con agujeros se introdujeron para evitar gastar espacio en disco. Se utilizan extensivamente por aplicaciones de bases de datos, y, de forma más general, por aplicaciones que realizan *hashing* sobre archivos.

La implementación Ext2 de archivos con agujeros se basa en la asignación dinámica de bloques de datos: un bloque es realmente asignado a un archivo cuando el archivo necesita escribir datos en él. El campo `i_size` de cada inodo define el tamaño del archivo como se ve por el programa, incluyendo el agujero, mientras que el campo `i_blocks` almacena el número de bloques de datos efectivamente asignados al archivos (en unidades de 512 bytes).

En nuestro anterior ejemplo, la orden `dd` supone que el archivo */tmp/hole* estaba creado en una partición Ext2 con tamaño de bloque 4096. El campo `i_size` del correspondiente inodo en disco almacena el número 6145, mientras que `i_blocks` almacena 8 (ya que cada bloque de 4096 bytes incluye 8 bloques de 512 bytes). El segundo elemento de la matriz `i_block` (correspondiente al bloque que tiene número de bloque de archivo 1) almacena el número de bloque lógico del bloque asignado, mientras que los otros elementos de la matriz son nulos (ver Figura 1.17).

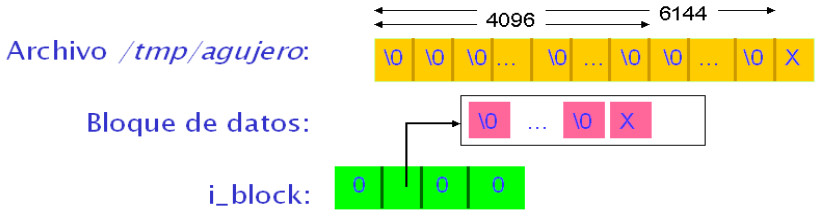


Figura 1.17.- Archivo con un agujero inicial

- **Asignación y liberación de bloques de datos**

Cuando el kernel tiene que asignar un nuevo bloque para almacenar datos de un archivo regular Ext2, invoca a la función `ext2_getblk()`. A su vez, esta función maneja las estructuras de datos descritas en la sección "Direccionando bloques de datos" e invoca, cuando es necesario, a la función `ext2_alloc_block()` para realmente buscar un bloque libre en la partición Ext2.

Para reducir la fragmentación, el sistema de archivos Ext2 intenta obtener un nuevo bloque para el archivo cerca del último bloque asignado al archivo. Si esto no es posible, el sistema de archivos busca un nuevo bloque en el grupo de bloques que incluyen en inodo del archivo. Como último recurso, el bloque libre se toma de otro grupo de bloques.

El sistema de archivos Ext2 utiliza preasignación de bloques. El archivo no solo obtiene el bloque solicitado, sino un grupo de hasta de 8 bloques adyacentes. El campo `i_prealloc_count` en la estructura `ext2_inode_info` almacena el número de bloques de datos preasignados a un archivo que no están en uso, y el campo `i_prealloc_block` almacena el número de bloque lógico del siguiente bloque preasignado para ser utilizado. Cualquier bloque preasignado que permanece sin usar se libera al cerrar el archivo, cuando se trunca, o cuando una operación de escritura no es secuencial con respecto a la operación que disparó la preasignación del bloque.

10 Planificación de disco

Lectura 1.9: Planificación de disco.
Estudia el apartado 11.5 "Planificación del disco" del libro de texto. En él se analizan los diferentes parámetros que intervienen en el rendimiento del disco, se repasan los algoritmos básicos utilizados de planificación.
Tiempo estimado: 15 minutos

Como indicábamos en la lectura, los algoritmos vistos son algoritmos básicos. En sistemas reales podemos encontrar variantes de estos algoritmos o algoritmos más sofisticados

pensados para tipos especiales de aplicaciones. Por ejemplo, el kernel 2.6 de Linux posee 4 algoritmos (*por plazos*, *anticipatorio*, *no-op*, *encolado completamente imparcial* -CFQ), que selecciona según diferentes parámetros como carga de trabajo, sistema de archivos, configuración hardware del sistema de entradas/salidas, etc.. Podéis ampliar más en http://www.kernel-labs.org/?q=blockiolayer).

Trabajos en Grupo 1.3: Algoritmos de planificación de disco.
Objetivo formativo: Manejar diferentes algoritmos de planificación de disco. **Tareas del grupo**: Un controlador en software de disco duro recibe peticiones para los cilindros 10, 22, 20, 2, 40, 6 y 38 en ese orden. El brazo tarda 6ms en moverse de un cilindro al siguiente Suponiendo que el brazo está inicialmente en el cilindro 20 ¿cuánto tiempo de posicionamiento se requiere siguiendo los algoritmos: 1. FCFS? 2. SSTF? 3. SCAN? **Criterio de éxito**: Cualquier miembro del grupo podrá demostrar, en un resumen oral o escrito, o mediante respuestas a preguntas del profesor, que ha alcanzado los objetivos formativos propuestos.
Duración: 25 minutos.

Lectura 1.10: Sistemas RAID.
Lee el apartado 1.6 “RAID” de libro en el que se describe qué es un sistema RAID y los diferentes tipo que podemos encontrar. El objetivo de esta lectura es conocer que es una sistema RAID y sus tipos, así como las principales ventajas e inconvenientes de cada tipo.
Tiempo estimado: 15 minutos

11 Archivos proyectado en memoria

Como hemos visto en multitud de ocasiones, el método tradicional para acceder a un archivo en un sistema operativo es abrir el archivo con la llamada *open* y después usar las llamadas *read, write* y *lseek* para realizar E/S. Este método es ineficiente por que requiere una llamada al sistema para cada operación de E/S (dos para accesos aleatorios). Esto se agrava si varios procesos acceden al mismo archivo y cada uno mantiene copia de los datos del archivo en su espacio de direcciones, gastando memoria innecesariamente. La Figura 1.18a describe la situación donde dos procesos leen la misma página de un archivo. Como puede verse hay tres copias de la página en memoria.

Ahora, consideremos un enfoque alternativo donde eliminamos la caché del kernel del sistema operativo y se proyecta la página leida en su espacio de direcciones (Figura 1.18b). El kernel crea esta proyección modificando simplemente algunas estructuras de datos de gestión de memoria. Cuando un proceso **A** intenta acceder a los datos de la página, genera una falta de página. El kernel la resuelve leyendo la página en memoria y modificando la tabla de páginas para que apunte hacia ella. Posteriormente, cuando el proceso **B** genera una falta por la página, la página ya esta en memoria, y el kernel solamente cambia la entrada en la tabla de páginas de **B** para que apunte hacia ella.

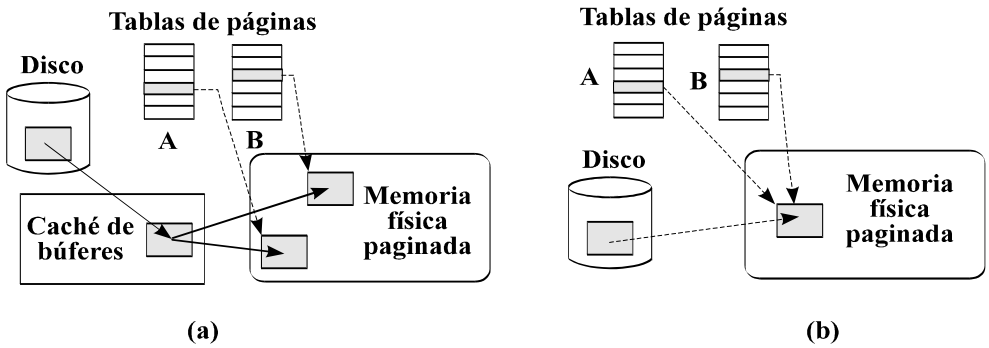


Figura 1.18.- Dos procesos (a) leen la misma página en UNIX tradicional, (b) la mapean en sus espacios de direcciones.

Esto ilustra los considerables beneficios del acceso a archivos mapeándolos en memoria. El coste total de las dos lecturas es un único acceso a disco. Realizado la proyección, no son necesarias más llamadas al sistema para leer/escribir datos. Como sólo hay una copia de la página en memoria, ahorramos dos páginas y dos operaciones de copia de memoria. Esta reducción en la demanda de memoria tiene beneficios adicionales al reducir las operaciones de paginación.

¿Qué ocurre cuando un proceso escribe una página proyectada? Un proceso puede establecer dos tipos de mapeos de archivos, *compartido* o *privado*. Para proyecciones compartidas, las modificaciones se hacen sobre el propio objeto mapeado. El kernel aplica directamente todos los cambios a esta copia compartida y la escribe de nuevo al archivo cuando se salva la página. Si un mapeo es privado, cualquier modificación provoca la realización de una copia privada de la página a la que se han aplicado los cambios. Estas escrituras no modifican el objeto subyacente; esto es el kernel no escribe la página en el archivo cuando salva la página.

Es importante observar que los mapeos privados no protegen frente a cambios realizados por otros que tiene mapeos compartidos sobre el archivo. Un proceso recibe su copia privada de la página sólo cuando intenta modificarla. Por tanto, ve todas las modificaciones realizadas por otros procesos entre el momento en que establece el mapeo y el momento en que intenta escribir en la página.

Las E/S de archivos mapeados en memoria son un pontente mecanismo que permite un acceso eficiente a los archivos. Sin embargo, no puede reemplazar a las tradicionales llamadas al sistema *read* y *write*. Una de las diferencias principales es la atomicidad de las E/S. Las llamadas *read* y *write* bloquean el inodo durante la transferencia de datos, garantizando que la operación es atómica. Para acceder a los archivos mapeados en memoria utilizamos instrucciones ordinarias de programa, por lo que como mucho podemos leer atómicamente una palabra. Este acceso no esta gobernado por la semántica tradicional de bloqueo de archivos, y la sincronización es enteramente responsabilidad de los procesos.

Otra diferencia importante es la visibilidad de los cambios. Si varios procesos comparten un mapeo a un archivo, los cambios realizados por uno son inmediatamente visibles al resto. Esto es completamente diferente del modelo tradicional, donde los otros procesos deben realizar otra lectura para ver los cambios. Con accesos mapeados, un proceso ve los contenidos de una página tal como están en el momento del acceso.

Sin embargo, estos temas están más relacionados con decisiones de la aplicación que usa el acceso mapeado al archivo. Esto no quita mérito y deseabilidad al mecanismo. De hecho, 4.3BSD especifica una interfaz para la llamada al sistema *mmap*, pero no suministra una implementación. A continuación describimos la semántica de la interfaz *mmap*.

- ***mmap* y llamadas al sistema relacionadas**

Para proyectar un archivo en memoria, debemos abrirlo primero con la llamada *open*. Esta es seguida de una llamada a *mmap* [UNIX92], que tiene la siguiente sintaxis:

```
#include <sys/types.h>
#include <sys/mman.h>

caddr_t mmap (caddr_t addr, size_t len, int prot,
                  int flags, int fd, int offset);
```

Retorna: dirección de inicio de la región mapeada si OK; -1 si error.

Esta llamada establece una proyección entre los bytes del rango `[offset, offset +len)` en el archivo representado por `fd`, y el rango de direcciones `[paddr, paddr+len)` en el proceso llamador. Los `flags` incluidos en el mapeo pueden ser `MAP_FIXE` , `MAP_SHARE` (las operaciones de modificación de la región alteran el archivo mapeado), y `MAP_PRIVATE` (las operaciones de modificación sobre la región provocan una copia del archivo mapeado). El llamador debe establecer la protección especificando `prot` como una combinación de `PROT_READ` (la página puede leerse), `PROT_WRITE` (puede escribirse), `PROT_EXECUTE` (puede ejecutarse), `PROT_NONE` (no puede accederse). La protección debe igualar el modo de apertura del archivo. Algunos sistemas, cuyo hardware no soporta permisos de ejecución separados, igualan `PROT_EXECUTE` a `PROT_READ`.

El sistema elige un valor adecuado para `paddr`. `paddr` nunca será 0, y el mapeo nunca recubrirá un mapeo existente. *mmap* ignora el parámetro `addr` salvo que el llamador especifique el indicador `MAP_FIXE`. En este caso, `paddr` debe ser exactamente el mismo que `addr`. Si `addr` no es aconsejable (bien si no esta alineado a página, bien no cae en el rango válido de direcciones de usuario), *mmap* devuelve un error. El uso de `MAP_FIXE` no es aconsejable, dado que produce código no portátil.

mmap funciona sobre páginas enteras. Esto exige que el desplazamiento este alineado a página. Si se ha especificado `MAP_FIXE`, `addr` también debe estar alineado a página. Si `len` no es múltiplo del tamaño de página, se redondeará por arriba para conseguirlo.

El mapeo permanece efectivo hasta que se deshaga con la llamada *munmap*, o remapeando el rango de direcciones con otro archivo invocando a *mmap* con el indicador `MAP_RENAME`. Se puede cambiar la protección a nivel de página con *mprotect*.

- **Un ejemplo de *mmap***

El Programa 1.2. realiza la copia de archivos (como la orden *cp*). Primero, abrimos los archivos y llamamos a *fstat* para obtener el tamaño del archivo de entrada, necesario para la llamada *mmap*. También, hay que ajustar el tamaño del archivo de salida. Para ello llamamos a *lseek* y escribimos un bytes en el. Esto hay que hacerlo para que la primera referencia al archivo no genere la señal `SIGBUS`. La copia la realizamos utilizando la función de biblioteca *memcpy* para copiar de un búfer a otro.

Programa 1.2.- Copia de archivos utilizando la llamada al sistema *mmap*.

```
#include     <sys/types.h>
#include     <sys/stat.h>
#include     <sys/mman.h> /* mmap() */
#include     <fcntl.h>
#include     "ourhdr.h"
#ifndef      MAP_FILE     /* 44BSD lo define y neceita para mapear archivos */
#define      MAP_FILE     0      /* para compilar en otros sistemas no 44BSD*/
```

```
#endif
int
main(int argc, char *argv[])
{
       int          fdin, fdout;
       char         *src, *dst;
       struct stat  statbuf;

       if (argc != 3)
              err_quit("usage: a.out <fromfile> <tofile>");
       if ( (fdin = open(argv[1], O_RDONLY)) < 0)
              err_sys("can't open %s for reading", argv[1]);
       if ((fdout = open(argv[2], O_RDWR | O_CREAT | O_TRUNC,FILE_MODE)) < 0)
              err_sys("can't creat %s for writing", argv[1]);
       if (fstat(fdin, &statbuf) < 0) /* necesitamos tamaño archivo salida */
              err_sys("fstat error");
                     /* ajustamos el tamaño del archivo de salida */
       if (lseek(fdout, statbuf.st_size - 1, SEEK_SET) == -1)
              err_sys("lseek error");
       if (write(fdout, "", 1) != 1)
              err_sys("write error");
       if ((src = mmap(0, statbuf.st_size, PROT_READ, MAP_FILE | MAP_SHARED,
                            fdin, 0)) == (caddr_t) -1)
              err_sys("mmap error for input");
       if ( (dst = mmap(0, statbuf.st_size, PROT_READ | PROT_WRITE,
                           MAP_FILE | MAP_SHARED, fdout, 0)) == (caddr_t) -1)
              err_sys("mmap error for output");
       memcpy(dst, src, statbuf.st_size);     /* realiza la copia de archivos */
       exit(0);
}
```

Si comparamos los tiempos necesarios para copiar un archivo con este programa y uno que utilice las llamadas al sistema *read*/*write* con un búfer de 8192 bytes, tendremos:

Tabla 1.9.- Tiempos empleados para copiar archivos con *read*/*write* y *mmap*.

	SPARC			**80386**		
	usuario	**Sistema**	**reloj**	**usuario**	**sistema**	**reloj**
read/*write*	0.0	2.6	11.0	11.0	5.1	11.2
mmap/*memcpy*	0.9	1.7	3.7	3.7	2.7	5.7

Las E/S mapeadas en memoria son más rápidas, pero tienen ciertas limitaciones. No se pueden utilizar para copiar entre diferentes archivos de dispositivos, p. ej. manejadores de red o terminales, y debemos tener cuidado si el archivo subyacente varia de tamaño durante el mapeo. Por otro lado, la llamada *mmap* es también útil para la compartición de memoria a través del dispositivo **`/dev/zero`** (ver [Stev92]).

Veamos un ejemplo que ilustra como el kernel de linux utiliza la proyección de archivos en memoria para construir el espacio de direcciones de un proceso. Para ellos partimos de un programa simple como el que aparece a continuación (no hace nada especial, la llamada `sleep()` permite parar el proceso para ver como es su espacio de direcciones):

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

main()
{
       int fd;
       fd=open("/root/soii/archivo", O_RDWR|O_CREAT);
       sleep(2000);
       exit(0);
}
```

Tras compilar el programa, lo traceamos para ver las llamadas al sistema que realiza.

```
% strace -cT ./ejemplo
```

```
execve("./map", ["./map"], [/* 24 vars */]) = 0
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 40.61    0.000335         335         1           munmap
 29.82    0.000246          62         4         1 open
 13.09    0.000108          22         5           old_mmap
  4.48    0.000037          37         1           uname
  3.03    0.000025          25         1           mprotect
  2.42    0.000020          10         2           close
  2.30    0.000019          10         2           fstat64
  2.18    0.000018          18         1           read
  1.09    0.000009           9         1           brk
  0.97    0.000008           8         1           getpid
------ ----------- ----------- --------- --------- ----------------
100.00    0.000825                    19         1 total
```

Vemos que realiza 5 llamadas `mmap()` y por tanto realiza 5 proyecciones de archivos (`old_mmap` es la llamada para proyección de archivo utilizada para bibliotecas antiguas, para las nuevas bibliotecas se usa `mmap2()`).

```
% strace -T ./ejemplo
execve("./map", ["./map"], [/* 24 vars */]) = 0
uname({sys="Linux", node="localhost.localdomain", ...}) = 0 <0.000066>
brk(0)                                  = 0x8049640 <0.000044>
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1,
0) = 0x40017000 <0.000055>
open("/etc/ld.so.preload", O_RDONLY)      = -1 ENOENT (No such file or
directory) <0.000068>
open("/etc/ld.so.cache", O_RDONLY)      = 3 <0.000074>
fstat64(3, {st_mode=S_IFREG|0644, st_size=55069, ...}) = 0 <0.000043>
old_mmap(NULL, 55069, PROT_READ, MAP_PRIVATE, 3, 0) = 0x40018000 <0.000049>
close(3)                                = 0 <0.000042>
open("/lib/i686/libc.so.6", O_RDONLY)   = 3 <0.000064>
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\200\302"..., 1024)
= 1024 <0.000051>
fstat64(3, {st_mode=S_IFREG|0755, st_size=5634864, ...}) = 0 <0.000042>
old_mmap(NULL, 1242920, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0) =
0x40026000 <0.000048>
mprotect(0x4014c000, 38696, PROT_NONE)  = 0 <0.000057>
old_mmap(0x4014c000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED, 3,
0x125000) = 0x4014c000 <0.000056>
old_mmap(0x40152000, 14120, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|
MAP_ANONYMOUS, -1, 0) = 0x40152000 <0.000055>
close(3)                                = 0 <0.005925>
munmap(0x40018000, 55069)               = 0 <0.000093>
getpid()                                = 1995 <0.000044>
open("/root/soii/archivo", O_RDWR|O_CREAT, 027777775310) = 3 <0.000078>
_exit(0)                                = ?
```

Visualizamos las regiones que tiene el proceso del ejemplo (suponemos que el PID de este procesos es el 1287).

```
% cat /proc/1287/maps
08048000-08049000 r-xp 00000000 08:05 14         /root/soii/m1
08049000-0804a000 rw-p 00000000 08:05 14         /root/soii/m1
40000000-40016000 r-xp 00000000 08:05 326414     /lib/ld-2.2.2.so
40016000-40017000 rw-p 00015000 08:05 326414     /lib/ld-2.2.2.so
40017000-40018000 rw-p 00000000 00:00 0
40026000-4014c000 r-xp 00000000 08:05 685461     /lib/i686/libc-2.2.2.so
4014c000-40152000 rw-p 00125000 08:05 685461     /lib/i686/libc-2.2.2.so
40152000-40156000 rw-p 00000000 00:00 0
bfffe000-c0000000 rwxp fffff000 00:00 0
```