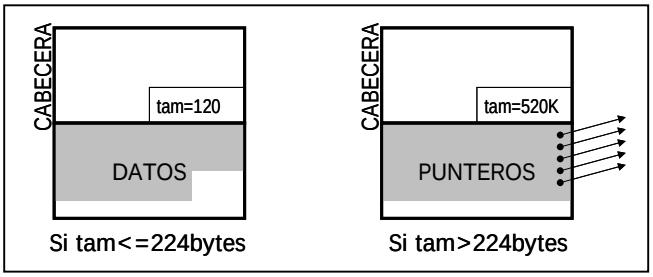
a) i-nodo para XLFS

XLFS guarda los ficheros muy pequeños reutilizando el espacio asignado originalmente para los punteros a bloques de datos. Esto quiere decir que si el fichero puede entrar en ese espacio dentro del i-nodo se mete ahí, pero si no entra se usa un bloque de datos.



¿Cuál es el tamaño, por debajo del cual los contenidos del fichero están en el i-nodo?

Este espacio será igual al ocupado por los punteros a bloques, es decir:

25 directos + 1 simple + 1 doble + 1 triple =

28 punteros a bloques x 8 bytes/dirección = 224 bytes

Si el fichero es <=224 bytes está en el i-nodo sino se usa la estructura de i-nodo habitual.

No es necesario ningún flag o dato especial para indicar que los datos están en el i-nodo o en bloques de datos. El atributo “tamaño de fichero” vale para discriminar cada caso.

b) i-nodo para FriserFS

Este caso es diferente, la característica principal de este SF es que el final del fichero es inferior al tamaño de un bloque, el espacio restante de dicho bloque se puede usar por el final de otro fichero. En nomenclatura de ReiserFS estas secciones de fichero se denominan colas. Aquellos ficheros muy pequeños (<4 KB) son ficheros compuestos enteramente por colas.

¿Cómo se pueden almacenar estas colas?

Pues cada i-nodo debe saber cuál es su último bloque de datos, que conocerá por la estructura de bloques directos, indirectos, etc. Para saber cuánto está ocupado de dicho último bloque se puede determinar a partir del tamaño del fichero (tamaño % 4KB). LO que puede ocurrir es que la ocupación de dicho bloque esté compartida con otra cola de fichero, en cualquier caso, el espacio ocupado por el fichero en cuestión puede estar al comienzo, en medio o al final del bloque. Por lo tanto es necesario que el i-nodo disponga de un “desplazamiento” para saber a partir de qué posición del último bloque están los últimos bytes del fichero. Para ello se puede reutilizar el siguiente puntero a datos, es decir que si un fichero ocupa 7KB, el primer puntero apunta a un bloque entero, el segundo puntero a un bloque compartido, del cual se ocuparán 3KB y el tercer puntero indicará el desplazamiento dentro de ese bloque compartido.

c) Gestión del espacio libre en FriserFS

Las estructuras de gestión del espacio libre son los bitmaps (de i-nodos y de datos). La gestión de i-nodos no varía, pero sí la de los bloques de datos. En este sistema de ficheros un bloque puede estar libre completamente, totalmente ocupad o parcialmente ocupado. Por lo tanto usar un bit para determinar el estado del bloque no es suficiente.

A continuación se presentan diferentes alternativas de solución, de la más simple (y desde luego menos elegante) hasta las más elaboradas. La calificación de este apartado dependerá de la calidad de la solución presentada. Las diferentes alternativas dependerán del grano de detalle al que se quiera repartir el espacio de los bloques compartidos. Además de estas posibles soluciones pueden haber otras que también pueden ser consideradas válidas.

Alternativa naïve:

Una opción es decir que los bloques se comparten a nivel de byte. De forma que de cada bloque se debería saber qué bytes están ocupados. Este nivel de detalle requeriría tener un bitmap de bytes (no de bloques) que indiquen si el byte *n* de cada bloque está libre u ocupado. Esto implica usar:

B bloques x 4KB de datos x 1bit de ocupación/8bits de datos =

512 x *B* bytes de bitmap

Grosso modo, esto implica tener 1/8 del espacio del disco como bitmap. Solución muy poco eficiente (0,5 puntos/2 puntos totales).

Particionamiento:
En lugar de permitir la ocupación del bloque byte a byte, se puede definir un tamaño mínimo de ocupación desde 1/8 de bloque (512bytes) hasta 1/64 de bloque (64 bytes). La solución será un compromiso entre el grado de aprovechamiento de los bloques de datos y el espacio (y dificultad) en la gestión del espacio libre.

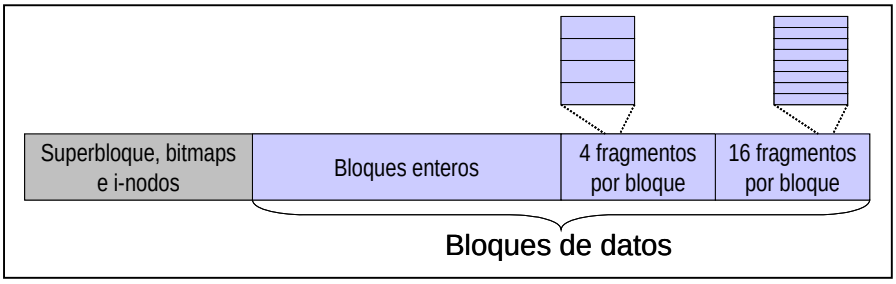
1/8 bloque: *B* bloques x 8 fragmentos/bloque x 1 bit/fragmento = *B* bytes
1/64 bloque: *B* bloques x 64 fragmentos/bloque x 1 bit/fragmento = 8x *B* bytes

Esta solución es más razonable (1,5 puntos/2 puntos totales), se puede usar como base para dos posibles refinamientos que se comentan a continuación:

Particionamiento diferenciado:
Utilizar un mismo nivel de detalle (mismo grano en la ocupación del disco), para todos los bloques puede no ser apropiado. Si hay un número significativo de bloques que están ocupados enteros (escenario que puede ser muy habitual), el espacio necesario para denotarlo en el bitmap es un desperdicio en los metadatos. En lugar de eso se puede definir que varios bloques de disco se identifiquen en el bitmap por un solo bit (el bloque entero está ocupado o no), otros bloques se pueden gestionar a nivel ½ bloque (2 bits en el bitmap), otros ¼ bloque (4 bits en el bitmap), etc.

Esta solución puede reducir el tamaño usado para el bitmap. Por ejemplo, supongamos la organización indicada en la figura: la mitad de los bloques se gestionan enteros, ¼ de los bloques se gestionan en 4 fragmentos, y otro ¼ de los bloques se gestionan 16 fragmentos:

B/2 bloques x 1 fragmento/bloque x 1 bit/fragmento +
B/4 bloques x 4 fragmentos/bloque x 1 bit/fragmento +
B/4 bloques x 16 fragmentos/bloque x 1 bit/fragmento=
B/16 bytes + B/8 bytes + B/2 bytes = 11/16 x B bytes



Las pegas que tiene esta solución es que puede aprovechar bien el espacio hasta el momento que los bloques que se gestionan en fragmentos de un determinado tamaño se terminan, teniendo que utilizar fragmentos de un tamaño mayor y por lo tanto desperdiciar espacio. Esta solución se considera muy buena (2 puntos/2 puntos totales).

Sistemas *buddy*:
Otro refinamiento posible es usar algoritmos de reserva de espacio (por ejemplo usados en gestión de memoria). Tal sería el caso del **sistema *buddy***, que, en términos generales utiliza varias listas de tamaños (potencias de dos) que gestiona agrupando las de un nivel (y pasándolas al nivel siguiente) o dividiéndolas (pasando el resto del espacio libre a niveles inferiores). Véase el tema de gestión de memoria para ver un esbozo de esta solución. La implementación de esta solución tiene sus pegas, pero no entraremos en detalle en su análisis por salirse del ámbito del examen.

Plantear una solución de este tipo (u otra similar a nivel de ingenio) implica 3puntos/2 puntos totales + ovación del publico.

d) Espacio ocupado en disco
Antes de empezar a ver cual sería el espacio que ocuparían en disco esta estructura vamos a repasar el tamaño de cada uno de sus componentes. Antes de ello vamos a hacernos con un par de parámetros de interés:

Tamaño de una entrada de directorio: 120 bytes (nombre)+8 bytes (dir. i-nodo)=
128 bytes → 4KB por bloque / 128 bytes/entrada = 32 entradas/bloque

Tamaño del i-nodo: 32 bytes (cabecera) +
(25 directos + 1 simple + 1 doble + 1 triple) x 8 bytes/dirección =
32 + (28 punteros a bloques x 8 bytes/dirección) = 256 bytes
Número de i-nodos por bloque: 4KB/bloque 256 bytes/i-nodo= 16 i-nodos/bloque

Esto implica que un bloque de datos destinado a almacenar el contenido de un directorio puede contener 32 entradas.

Directorio raíz /
4 entradas + 2 (entrada "." y "..") = 6 entradas x 128 bytes/entrada = 768 bytes

Directorio Jethro Tull
19 entradas + 2 (entrada "." y "..") = 21 entradas x 128 bytes/entrada = 2688 bytes

Directorio ZZTop
21 entradas + 2 (entrada "." y "..") = 23 entradas x 128 bytes/entrada = 2944 bytes

Directorio Yes
23 entradas + 2 (entrada "." y "..") = 25 entradas x 128 bytes/entrada = 3200 bytes

Directorio Yes/Rick_Wakeman
20 entradas + 2 (entrada "." y "..") = 22 entradas x 128 bytes/entrada = 2816 bytes
Los enlaces físicos consumen entrada de directorio, pero no i-nodo.

Directorio The_Kinks
47 entradas + 2 (entrada "." y "..") = 49 entradas x 128 bytes/entrada = 4KB+2176 bytes

Un aspecto que es común a todos los sistemas de ficheros es el número de i-nodos que usan. El uso de ninguno de ellos implica la necesidad de un número diferente de ellos. Necesitaremos, por lo tanto, 1 i-nodo por fichero, 1 i-nodo por directorio. No se necesita un nuevo i-nodo por enlace físico:

6 directorios (raíz+5) + (19+21+22+18+47) ficheros = 133 i-nodos / 16 i-nodos/bloque = 8 bloques(de i-nodos) + 5 i-nodos

Bloques de datos SIMPLOFS

Todos los ficheros de menos de 4KB de tamaño ocupan un bloque, esto incluye todos los directorios, salvo The_Kinks) es decir:
(directorio raíz) 1 +
(directorio Jethro_Tull) 1 + 10 + 5 +
(directorio ZZTop) 1 + 20 + 1x2 +
(directorio Yes) 1 + 22 +
(directorio Yes/Rick_Wakeman) 1 + 18 +
(directorio The_Kinks) 2 + 45 + 2x2 = 133 bloque 4KB/bloque = 532KB

A estos datos quedan por sumar los 4 ficheros de 1MB.
Un fichero de 1MB implica 1MB / 4KB/bloque = 256 bloques
Esto implica que consume los 25 punteros directos y que usa el puntero indirecto simple. Este puntero apunta a un bloque que contiene 4KB / 8bytes/dirección= 512 direcciones de bloques. Siendo suficientes bloques por lo tanto no necesitará usar el puntero doble y de éste otro bloque de punteros simples. Total:
256 bloques de datos + 1 bloque simple =
257 bloques x 4KB/bloque = 1028 KB

Tamaño total = 532KB + 4 ficheros x(1028KB) = 4644 KB

Bloques de datos de XLFS

XLFS almacenará en el i-nodo los contenidos de aquellos ficheros de tamaño <224 bytes (ver apartado a). Esto no se aplica a ninguno de los 6 directorios pero sí a muchos ficheros:
Ficheros <224 bytes: 10 + 5 + 20 + 22 + 18 + 45 = 120 ficheros.

Por lo tanto se ahorraría (respecto de la solución anterior) 120 bloques de datos x 4KB/bloque= 480KB

Tamaño total = 4644 KB - 480KB = 4164 KB

Bloques de datos de FriserFS
No vamos a considerar la solución naïve al problema puesto que ya hemos dicho que es irreal. Vamos a optar por una de las alternativas de particionamiento (diferenciado o no). Para ello vamos a considerar (como un valor razonable) que el fragmento de tamaño más pequeño con el que vamos a trabajar es de 64 bytes (1/64 de bloque).

Para determinar el espacio ocupado en disco vamos a ponernos en el mejor caso, es decir que no ha habido fragmentación y hemos conseguido rellenar todos los bloques parciales y no nos han borrado ningún fichero.

Desde el punto de vista de cálculo de tamaños lo que ocurriría es que el "tamaño de bloque" (no exactamente tal, pero sí del fragmento que usamos como unidad de ocupación) es 64 bytes. Lo cual implica que los ficheros de tamaño menor a 64 bytes, ocuparán un fragmento entero y el resto se dividirán en fragmentos de ese tamaño.

(directorio raíz) [768/64=12] +
(directorio Jethro_Tull) [2688/64=42] + 10 + 5x2 +
(directorio ZZTop) [2944/64=46] + 20 + [5KB/64B=80] +
(directorio Yes) [3200/64=50] + 22 +
(directorio Yes/Rick_Wakeman) [2816/64=44] + 18 +
(directorio The_Kinks) [6272/64=98] + 45 + 2x[6KB/64B=96] = 689 fragmentos x 64B/fragmento = 43KB + 64B = aprox. 43KB

Los 4 ficheros de 1MB ocuparían lo mismo que en el caso anterior. Cabría la posibilidad de aplicar el mismo mecanismo de particionamiento de bloques de datos a los bloques indirectos (simples, dobles, etc...), pero no vamos a aplicarlo en este caso, aunque bien podría ser así.

Tamaño total = 43KB + 4 ficheros x(1028KB) = 4155 KB

e) Accesos a disco
e.i) open(....)
Abrir un fichero requiere un total de accesos igual a los necesarios para decodificar la ruta + 1 acceso al i-nodo. Los bloques de datos del fichero no se leen.

Podemos considerar que el i-nodo del directorio raíz está en memoria:
1 bloque de datos del contenido del directorio raíz +
1 bloque (donde se encuentra el i-nodo del directorio The_Kinks) +
2 bloques de datos del contenido del directorio The_Kinks +
1 bloque (donde se encuentra el i-nodo del fichero solicitado) = 5 bloques

En este caso, cualquiera de los sistemas de ficheros tendrían que hacer los mismos accesos. Incluso si FriserFS usa fragmentos más pequeños para organizar el espacio libre la unidad de lectura del disco es el bloque. En el caso de XLFS, ninguno de los directorios leídos es tan pequeño como para entrar dentro del i-nodo (no se ganaría esa ventaja).

e.ii) creat(....); write(...)
Crear, al igual que abrir, un fichero requiere un total de accesos igual a los necesarios para decodificar la ruta + 1 acceso al i-nodo. En esta caso los contenidos necesarios para la decodificación de la ruta ya están en cache, por lo tanto no se necesita acceder a disco. El contenido del directorio The_Kinks se variaría con la operación de creación de una nueva entrada, pero esta escritura no se haría hasta sincronizar la cache con el sistema de ficheros. Lo que si se debería escribir es el nuevo i-nodo que se ha creado, puesto que a los metadatos se les suele aplicar la política *write-through* en la cache.

La operación de escritura, tampoco se mandaría a disco hasta que no se sincronice cache y SF. Lo que sí que se haría, es volver a reescribir el i-nodo puesto que el tamaño del fichero sí ha cambiado.

En el caso de que la política de escritura de metadatos no fuese *write-through*, entonces no se haría ningún acceso. Todo quedaría en memoria hasta que se sincronice la cache con el disco.

f) Modificaciones

f.i) Aumento del tamaño del i-nodo

A todos los sistemas de ficheros les concedería la ventaja de poder direccionar un tamaño máximo de fichero mayor (aunque no significativamente mucho mayor, los punteros directos suman 4KB al tamaño máximo, frente al orden de centenares GB del triple).

El principal beneficiado de este cambio, sin duda sería XLFS puesto que permitiría que se guardasen en el i-nodo ficheros más grandes de los 224 bytes del diseño original en concreto 4KB-32 bytes(cabeceras).

f.ii) Ficheros pequeños de sólo lectura

Este tipo de ficheros beneficia notablemente a FriserFS y a XLFS, puesto que los ficheros pequeños se pueden agrupar mejor en el primer caso o meterse en el propio i-nodo en el segundo. La característica de que los ficheros no crezcan y sí se lean muchas veces viene muy bien para FriserFS, puesto que elimina problemas de fragmentación interna y reubicación de datos en cada modificación del tamaño del fichero.

f.iii) Grandes ficheros de tamaño creciente

Esta característica ataca principalmente a FriserFS, puesto que su política de asignación, con fragmentos pequeños dentro de los bloques, implicaría mover las colas de ficheros de un bloque a otro, según va creciendo el fichero. De todas formas esta complejidad no es tan grave, puesto que una operación de E/S dura varios órdenes de magnitud más que la lógica de control, por muy compleja que sea ésta. El sistema de ficheros se vería penalizado por este aspecto, pero sí por tener que leer nuevos bloques donde asignar un final de fichero, una vez que se ve que no entra en un determinado bloque, es decir muchas más lecturas y escrituras. De todas formas este problema se puede reducir, haciendo que las asignaciones de espacio para estas colas estén ligeramente sobredimensionadas, pero habría que tener en cuenta que si se sobre dimensionan mucho, el concepto del SF de FriseFS se desvirtúa y estaríamos asignando bloques enteros y nunca compartiendo.

El uso de ficheros grandes no penaliza a XLFS, puesto que una vez que el fichero no entra en el i-nodo se comporta como SIMPLOFS. Además los ficheros grandes no permiten aprovechar las mejores características de XLFS y de FriserFS en la gestión de ficheros pequeños.