Sistemas de archivos

Sistemas Operativos

Franco Frizzo

9 de mayo de 2017

Ejercicio 1

Considerar un disco con bloques de 8 KB y un archivo de 10 MB. ¿Cuántas operaciones de disco son necesarias para...

(a) agregar un bloque al principio del archivo?
(b) agregar un bloque en la mitad del archivo?
(c) agregar un bloque al final del archivo?
(d) eliminar el primer bloque del archivo?
(e) eliminar el bloque en la mitad del archivo?
(f) eliminar el último bloque del archivo?

Responder para cada uno de los siguientes escenarios:

1. En el sistema de archivos, los bloques de un archivo se almacenan en forma contigua. Se conoce en qué posición del disco se encuentra el primer bloque del archivo. Suponer que hay bloques libres luego del final del archivo, pero no antes del comienzo.
2. El sistema de archivos tiene una FAT, que está cargada en memoria. Se conoce la dirección del primer bloque del archivo. ¿Cuántas posiciones de la FAT deben consultarse y cuántas deben modificarse en cada caso?
3. Se tiene un sistema de archivos con inodos, sin indirecciones. El inodo del archivo está cargado en memoria. ¿Cuántas posiciones del inodo deben consultarse y cuántas deben modificarse en cada caso?

Resolución

La cantidad de bloques del archivo es $\frac{|\text{archivo}|}{|\text{bloque}|} = \frac{5 \times 2^{21}\ \text{B}}{2^{13}\ \text{B}} = 5 \times 2^8$ bloques = 1280 bloques.

	1. Contiguo	2. FAT			3. inodos		
	Op. disco	Op. disco	Lect. FAT	Esc. FAT	Op. disco	Lect. inodo	Esc. inodo
(a)	2561	1	0	1	1	1280	1281
(b)	1281	1	640	2	1	640	641
(c)	1	1	1279	2	1	0	1
(d)	2558	0	1	0	0	1279	1279
(e)	1278	0	641	1	0	639	639
(f)	0	0	1278	1	0	0	0

Ejercicio 2

Se tiene un disco con capacidad de 128 GB, y bloques de 8 KB. Supongamos un sistema de archivos similar a FAT, donde la tabla se ubica desde la posición 0.

1. ¿Cuál es el tamaño que ocupará la tabla?
2. ¿Cuál es la cantidad máxima de archivos de 10 MB que es posible almacenar?
3. Se sabe que un archivo comienza en el bloque 20. Dada la siguiente FAT, indicar el tamaño de dicho archivo.

Bloque	0	1	2	3	4	5	6	...	20	21	22	...
Siguiente	EOF	2	23	4	5	0	7	...	21	22	3	...

Resolución

1. En primer lugar, calculemos la cantidad de bloques que tiene el disco.

$$\frac{|\text{disco}|}{|\text{bloque}|} = \frac{2^{27}\ \text{KB}}{2^{3}\ \text{KB}} = 2^{24}\ \text{bloques}$$

   Esto quiere decir que para direccionar un bloque hacen falta 24 bits. Este será el tamaño de cada una de las entradas de la FAT. Como la tabla tiene una entrada por cada bloque, ocupará en total $2^{24} \times 3$ B = 48 MB.

2. Dado que la tabla ocupa 48 MB, nos quedan en el disco 128 GB − 48 MB = 131024 MB. En este espacio, podemos almacenar como máximo 13102 archivos de 10 MB cada uno. Es importante tener en cuenta que no se desperdicia espacio dentro de los bloques por fragmentación interna, ya que tanto el tamaño de los archivos como el de la FAT son múltiplos del tamaño del bloque.

3. A partir de la tabla, podemos ver que nuestro archivo ocupa los bloques 20, 21, 22, 3, 4, 5, y 0, es decir, 7 bloques. Como cada bloque es de 8 KB, el archivo ocupa, como máximo, 56 KB en disco. Dado que el último bloque no necesariamente está lleno, el tamaño real del archivo está en el intervalo $(48, 56]$ KB.

Ejercicio 3

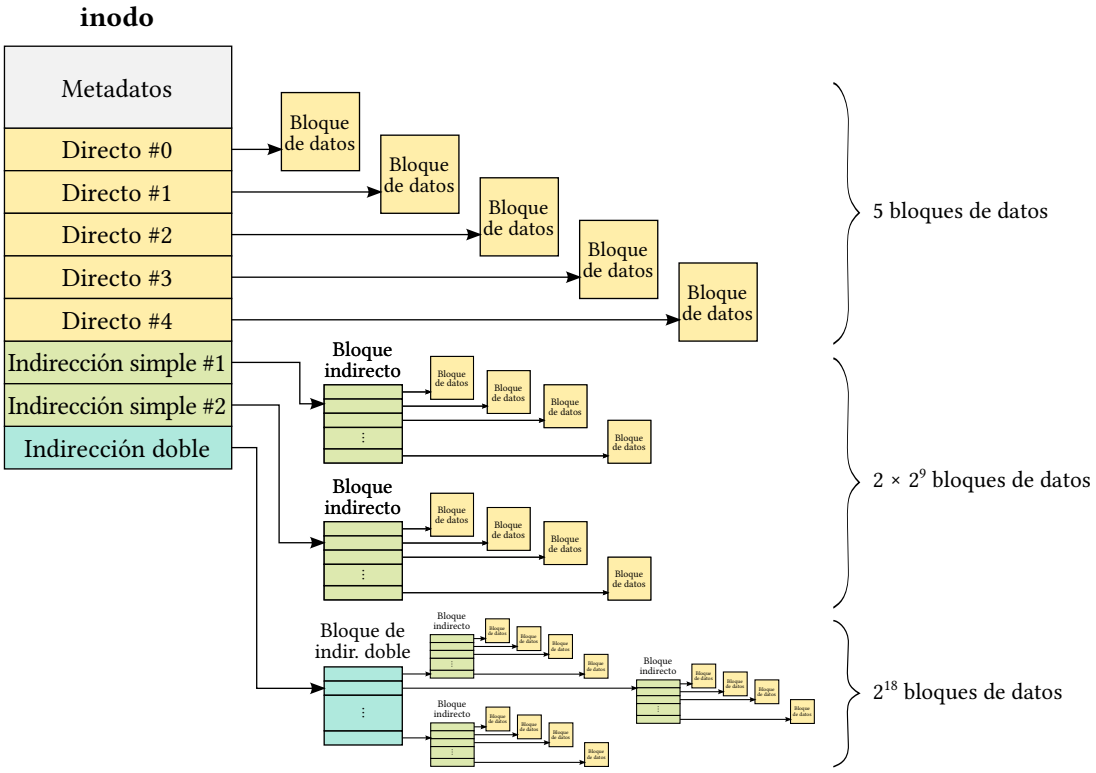
Un sistema de archivos de UNIX, tipo ext2, tiene bloques de tamaño 4 KB y el direccionamiento a bloques de disco (LBA) es de 8 bytes. A su vez, cada inodo cuenta con 5 entradas directas, dos indirectas y una doblemente indirecta.

1. ¿Cuál es el tamaño máximo de archivo que soporta?
2. Si el 50% del espacio en disco es ocupado por archivos de 2 KB, el 25% por archivos de 4 KB y el 25% restante por archivos de 8 KB, ¿qué porcentaje del espacio en disco es desperdiciado? (Considere solo el espacio utilizado en los bloques de datos).
3. ¿Cuántos bloques hace falta acceder para procesar completo un archivo de 5 MB?

Resolución

1. El tamaño máximo está dado por la cantidad de bloques que puede referenciarse desde un inodo. Calculemos esta cantidad por casos:

- **Directos:** estos punteros permiten referenciar 5 bloques.
- **Indirección simple:** cada entrada indirecta apunta a un bloque donde habrá entradas de 8 B. Esto quiere decir que podremos referenciar $\frac{4\text{ KB}}{8\text{ B}} = 2^9$ bloques de datos por cada entrada.
- **Indirección doble:** cada entrada doblemente indirecta apunta a un bloque que direcciona a 2^9 tablas de indirección simple, cada una de las cuales referencia, a su vez, 2^9 bloques de datos. Es decir, cada entrada doblemente indirecta puede direccionar 2^{18} bloques.



Esto quiere decir que, en total, un inodo puede referenciar $5 + 2 \times 2^9 + 2^{18}$ bloques. Considerando que los bloques son de 4 KB, el tamaño máximo de archivo es de 1052692 KB, es decir, aproximadamente 1028.02 MB.

2. Teniendo bloques de 4 KB, para los archivos de 4 KB y 8 KB no hay desperdicio alguno, ya que usarán 1 y 2 bloques completos, respectivamente. Los archivos de 2 KB, por su parte, deberán utilizar un bloque entero de 4 KB, desperdiciando entonces la mitad del bloque. Dado que ocupan el 50% del espacio, un 25% del disco está desperdiciado.

3. Por un lado, tendremos que acceder todos los bloques de datos que constituyen el archivo; se trata de $\frac{5\text{ MB}}{4\text{ KB}} = 1280$ bloques. Además, tendremos que acceder a:

   I. El bloque que contiene el inodo del archivo. Desde allí podremos acceder directamente a los primeros 5 bloques de datos.

   II. El bloque que contiene la primera tabla de indirección simple, porque desde allí se referencian los bloques entre el 6 y el $6 + 2^9 = 518$ del archivo.

   III. El bloque que contiene la segunda tabla de indirección simple, desde donde se referencian los bloques entre el 519 y el $519 + 2^9 = 1031$ del archivo.

   IV. Todavía falta encontrar las referencias de los últimos $1280 - 1031 = 249$ bloques del archivo. Todas ellas se pueden alcanzar accediendo a la tabla de indirección doble y, luego, a la primera tabla de indirección simple referenciada desde allí. Es decir, es necesario acceder a otros 2 bloques.

En total, es necesario realizar $1280 + 1 + 1 + 1 + 2 = 1285$ accesos a bloques para procesar completamente el archivo.

Ejercicio 4

1. ¿Qué ventajas tiene un sistema que utiliza inodos sobre uno de tipo FAT?
2. ¿Qué ventajas tiene un sistema de tipo FAT sobre uno que utiliza inodos?

Resolución

1. En un sistema con inodos, cada inodo se corresponde con un único archivo. Así, pueden cargarse solo los que son necesarios en un determinado momento, a diferencia de la FAT, que debe mantenerse completa en memoria; esto reduce la cantidad de memoria necesaria para utilizar el sistema de archivos. También se evita la contención que genera tener una única estructura compartida por todos los archivos: usando inodos, cada proceso solo necesita bloquear los inodos de los archivos que está empleando en un momento dado. Por otra parte, el sistema es más robusto, dado que si se corrompe un inodo, solo se pierde el archivo correspondiente, mientras que en caso de dañarse la FAT puede quedar comprometido todo el sistema de archivos.
2. Para unidades pequeñas, un sistema de tipo FAT resulta muy performante, y se puede implementar con gran sencillez. Es soportado por la inmensa mayoría de sistemas operativos, incluyendo sistemas antiguos o muy sencillos (por ejemplo, embebidos); esta compatibilidad lo vuelve muy útil para el intercambio de información, por lo que se utiliza comúnmente en *pendrives*, tarjetas de memoria, etc. En cambio, una implementación basada en inodos es considerablemente más compleja, y en ciertos escenarios las ventajas que presenta no son significativas.

Ejercicio 5

Escribir el pseudocódigo de la función

```
char* read_file(string name)
```

que devuelve el puntero a un *buffer* con el contenido de un archivo ubicado en el directorio *root* de un sistema de archivos FAT.

Se puede asumir que se dispone de:

- las funciones:
  - `lba_t fat_lookup(lba_t block_index)`.
  - `char* allocate_buffer(int size)`.
  - `void copy_block(lba_t orig_lba, char* dst_buffer)`.
- una estructura, `fat_dir_entry`, que representa una entrada de directorio de un sistema de archivos FAT.
- las constantes `BLOCK_SIZE`, `ROOT_START_BLOCK` y `ROOT_SIZE`.

Resolución

Para simplificar la resolución, asumimos que:

- El archivo existe, y está presente en el directorio raíz.
- El tamaño del archivo es múltiplo del tamaño del bloque; en caso contrario, tendríamos que tener cuidado a la hora de copiar al *buffer* el último bloque de datos.

```
char* read_file(string name) {
    // Recuperamos la entrada de directorio
    fat_dir_entry dir_entry = get_dir_entry(name);

    // Pedimos un buffer del tamaño necesario
    int file_size = dir_entry.size;
    char* ret_buffer = allocate_buffer(file_size);

    // Copiamos todos los bloques del archivo
    lba_t current_block = dir_entry.first_block;
    char* position_in_buffer = ret_buffer;
    while (current_block != EOF) {
        copy_block(current_block, position_in_buffer);
        position_in_buffer += BLOCK_SIZE;
        current_block = fat_lookup(current_block);
    }

    return ret_buffer;
}

// El siguiente algoritmo solo funciona si el archivo está en el directorio
// root, porque los bloques que lo conforman son contiguos. En otros directorios
// habría que ir consultando la FAT para saber cuál es el bloque siguiente.

fat_dir_entry get_dir_entry(string name) {
    char* dir_buffer = allocate_buffer(BLOCK_SIZE);
    lba_t current_block = ROOT_START_BLOCK;
    for (int i = 0; i < ROOT_SIZE; i++) {
        copy_block(current_block, dir_buffer);
        fat_dir_entry dir[] = (fat_dir_entry[]) dir_buffer;
        for (int j = 0; j < BLOCK_SIZE / sizeof(fat_dir_entry); j++) {
            if (dir[j].name == name) {
                return dir[j];
            }
        }
        current_block++;
    }
}
```

Disclaimer: El pseudocódigo anterior definitivamente **no** compila, y omite un montón de detalles técnicos que habría que tener en cuenta en una implementación real. La idea es tener una noción general del algoritmo que se lleva a cabo para recuperar un archivo a partir de su nombre en un sistema FAT.