Clase de Filesystem + Taller

Juan Pablo Darago

3 de octubre de 2013

Introducción General

Sistemas de Archivos

- ¿Qué es un sistema de archivos (*filesystem*)?
  - Una abstracción sobre almacenamiento permanente que nos permite manejar y estructurar la información mediante archivos.
  - Abstraer la ubicación física (la red, un sector de un disco, etc.) de la información en una ubicación abstracta (árbol de directorios, nombres).
- ¿Qué es un archivo?
  - En UNIX: ¡TODO!
  - Los directorios, los devices, archivos en disco, en un *filesystem* en la red, etc.
  - En general, secuencia de bytes sin estructura. La estructura se las da el que los usa (o el mismo filesystem. Ejemplo: Directorios).

Filesystems en lo que vamos a ver

- Hay muchos sistemas de archivos
  - Windows: FAT16, FAT32, NTFS
  - UNIX/Linux: UFS, Minix 3, Ext2, Ext3
- Nos vamos a concentrar en *filesystems* sobre disco y como organizar archivos ahi.
- Uno de los que vamos a ver (Ext2) también es la base de la capa de abstracción general (VFS) de Linux para todos los sistemas de archivos.
- Pero primero entonces, vendría bien un repaso de como es un disco duro.

Disco duro

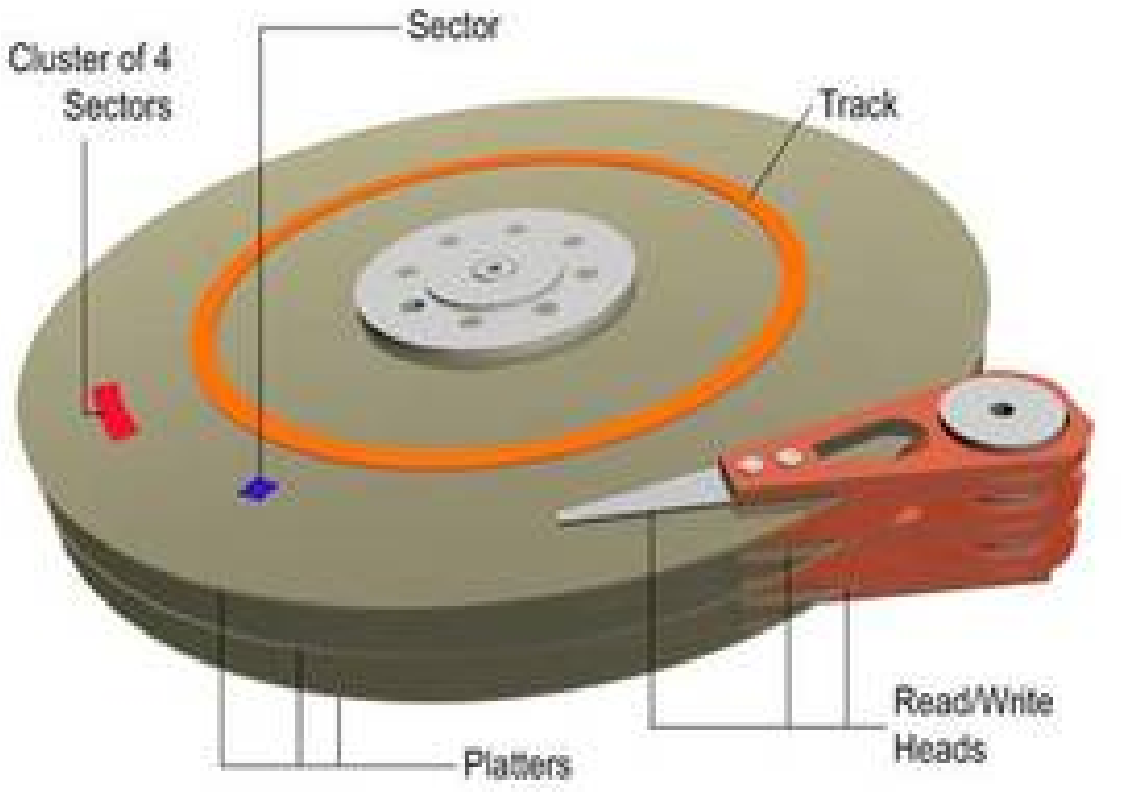


Figura: Disco duro

Ideas generales

Organizando Archivos

Asignación Continua

- **Concepto:**
  - Almacenamos cada archivo en sectores consecutivos.
- **Ventajas:**
  - Acceder al contenido de un archivo es muy muy rápido.
- **Desventajas:**
  - Borrar o modificar un archivo tiene muchos problemas.
  - Sufre de mucha fragmentación interna y externa.
    - Interna: Necesito si o si bloques consecutivos. Si el más chico en donde entra es enorme, el resto esta perdido.
    - Externa: Al crear muchos archivos chicos y borrarlos.

Lista enlazada

- **Concepto:**
  - En cada bloque de un archivo guardo un puntero al proximo bloque (si lo hay).
- **Ventajas:**
  - Menos fragmentación, uso mejor los bloques de disco.
  - No es malo para acceso secuencial.
  - Simple de implementar.
- **Desventajas:**
  - Corrupción de puntero == *archivo* inconsistente.
  - *Seeks* aleatorios son carísimos.
  - Si uso sectores lejanos incluso el acceso secuencial es malo.

FAT (*File Allocation Table*)

- **Concepto**:
  - ¡Metamos todos los punteros en una tabla!
- **Ventajas**:
  - Usando poca memoria RAM puedo acceder a cualquier lugar del disco en 1 acceso.
  - Muy muy sencillo de implementar y *debuggear*.
- **Desventajas**:
  - La tabla crece con el disco, para ciertos tamaños no es muy razonable.
  - Necesito la tabla entera en memoria.
  - Se daña el sector de la FAT == *filesystem* inconsistente.
  - FAT en memoria + *crash* == *filesystem* inconsistente.

Ejercicio sobre FAT

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

1. ¿Cuál es el tamaño de la tabla?
2. Se sabe que un archivo comienza en el bloque 2000. Dada la siguiente FAT, indicar el tamaño del archivo, asumiendo que ocupa totalmente todos los bloques que necesita.

B	0	1	...	614	...	703	704	...	2000	2001	...
S	1	2	...	EOF	...	704	EOF	...	2001	703	...

Inodos

- **Concepto**: Utilizar una estructura para cada archivo.
  - A cada archivo tiene un *i-nodo* con metadatos.
  - El *inodo* contiene punteros a bloques de datos y a bloques con más punteros
  - Los directorios en vez de punteros tienen entradas $<$ nombre, inodo del subpedazo $>$
- **Ventajas**:
  - Buen *trade-off* acceso secuencial y aleatorio.
  - Cuanta RAM consumo depende de los *inodos* cargados.
  - Robusto: se corrompe un inodo, no se cae todo el *filesystem*
  - Inodos tiene tamaño fijo, fácil meter en arreglos.
- **Desventajas**:
  - Tamaño máximo de archivo esta acotado.

Ejercicio *i-nodos*

En un FS con inodos de 256 bytes se desea acelerar el resultado de la operación `ls -la` que muestra los nombres de los archivos, sus atributos y su tamaño

```
% ls -la
drwxr-xr-x  20 root   wheel      2560 25 may 18:51 .
drwxr-xr-x  21 root   wheel       512  2 jun 00:21 ..
drwxr-xr-x   2 root   wheel       512 16 may 19:44 X11
-rw-r--r--   1 root   wheel       221 16 feb 23:19 amd.map
-rw-r--r--   1 root   wheel      1248 16 feb 23:19 apmd.conf
-rw-r--r--   1 root   wheel       241 16 feb 23:19 auth.conf
drwxr-xr-x   2 root   wheel       512 16 feb 23:19 bluetooth
-rw-r--r--   1 root   wheel       736 16 feb 23:19 crontab
```

- Si el FS tiene bloques de 512bytes, los *i*-nodos de un mismo directorio se encuentran contiguos y el primero comienza al principio de un bloque. En el listado del ejemplo, ¿cuántos accesos a bloques fueron necesarios?
- ¿Cómo podría modificarse el FS para resolverlo en un acceso? ¿Cuál sería el precio que se pagaría?

Aspectos más generales

- Ok, ya sabemos como organizar un archivo.
- Pero, también de disco *booteamos* y desde ahi el *filesystem* se reconoce a si mismo.
- ¿Cómo?
  - *Master Boot Record*: Ubicado en el primer sector del disco.
  - Levantado a memoria apenas termina el POST de la máquina.
    - Contiene código de para levantar la computadora al principio.
  - Tiene la *tabla de partición*: Las secciones del disco duro (donde están y su longitud).
- Ok, eso resuelve encender. ¿Como incializo el Filesystem?
- Lo vamos a ver analizando Ext2, El *Second Extended Filesystem*, durante muchos años el más popular de Linux.

Ext2

Presentación Taller

- En el taller que nos trajo a todos aca, tienen que trabajar con el sistema de archivos *Ext2*
  - Popular, muy documentado, mejorado en *Ext3* y *Ext4*.
  - Empezó como el *filesystem* de Linux pero implementaciones existen para BSD, Windows, etc.
- Les damos una implementación parcial de un *driver* para Ext2 y tienen que implementar unas funciones.
- Luego tienen que usar esas funciones para leer una imágen de disco y obtener unos datos.

Estructura

- El disco empieza con espacio para el *bootblock* de la partición.
- Después sigue un conjunto de *Block Groups*.
  - Empiezan con el *super block*, que tiene todos los datos del *filesystem*.
    - Replicado en cada grupo.
  - Sigue el *block group descriptors*, que tiene los datos de cada grupo.
    - También replicado como el *super block*.
- Luego sigue la información específica a ese grupo de bloques.
  - *Data block bitmap*
  - *Inode bitmap*
  - *Inode table*
  - *Data blocks*: Donde guardamos los datos efectivamente.

En detalle

- *Data block bitmap*: Un mapa de bits: que bloques están libres.
- *Inode bitmap*: Un mapa de bits: que inodos estan libres.
- *Inode table*: Arreglo con los datos de los *inodos*.

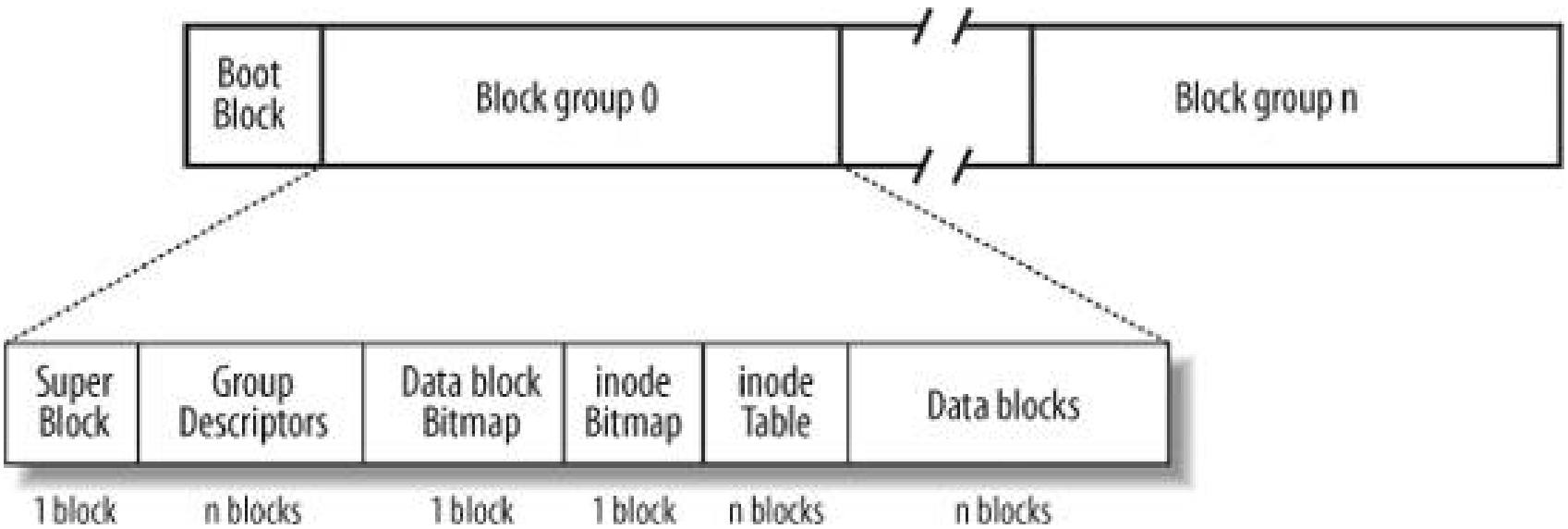
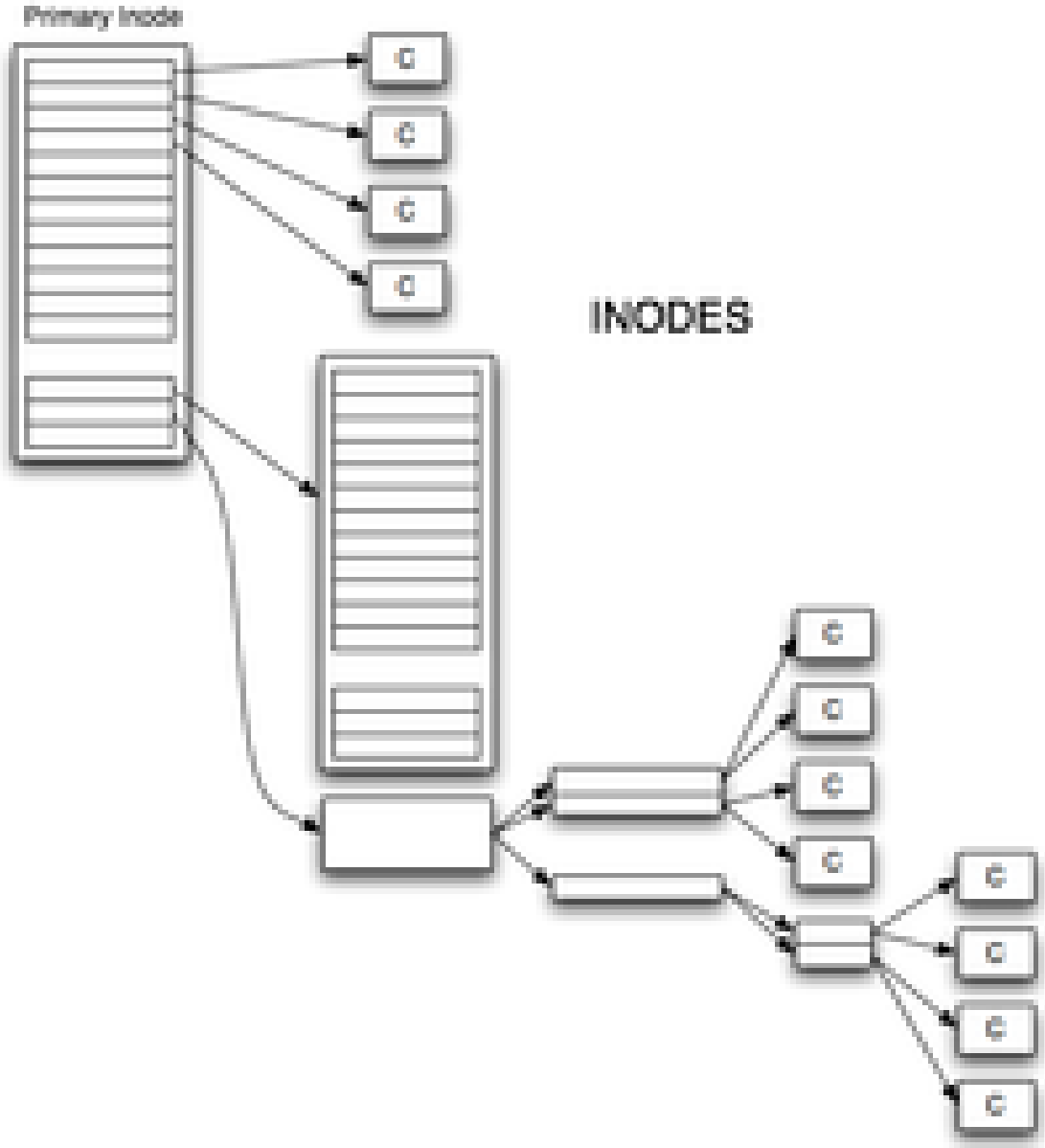


Figura: Estructura Ext2

Inodo de Ext2 en detalle.

- Cada inodo tiene 12 punteros a bloques con datos.
- Cada inodo tiene un puntero indirecto.
  - Puntero a un bloque con punteros a bloques
- Cada inodo tiene un puntero doble indirecto.
  - Puntero a un bloque con punteros a bloques con punteros.
- Los inodos pueden ser de varios tipos.
  - Pueden ser desde directorios hasta pipes, sockets, devices, etc.
- Los inodos directorio en vez de punteros tienen
  - En vez de punteros, tienen estructuras de directorio.
  - Cada estructura guarda la longitud de la entrada, longitud del nombre, nombre (no necesariamente null-terminated) e inodo.
  - Los pedazos se juntan con / en UNIX.
  - Un inodo fijo (el 2) es el *root node* (/) del filesystem.

Inodo esquemático



Estructuras en el código

- Veamos como es cada estructura en el código.
- Las mismas son copia exacta de como se ubican en disco (asumiendo Little-Endian), ya que los structs y clases de C++ tienen contenido consecutivo en memoria.
- Miremos entonces:
  - MBR
  - Super bloque
  - Descriptor de grupo de bloques
  - Inodo
  - Entrada de directorio.

Enunciado

Enunciado Taller

1. Completar la implementación de los siguientes métodos:
   1.1 get_block_address(inode,block_number)
   1.2 load_inode(inode_number)
   1.3 get_file_inode_from_dir_inode(from,filename)
2. Hacer un programa que, utilizando el FS programado en el punto anterior, imprima los 17 caracteres que se encuentran guardados en el archivo /grupos/gNUMERO/nota.txt (de la imágen de disco hdd.raw provista) a partir de la posición 14000.

Ayudas para el taller

- Haganlos en el órden dado.
- Funciones utiles:
  - read_block: Lee un bloque de disco a un buffer
  - strcmp: Te devuelve 0 si dos strings son iguales.
  - block_group: Devuelve el puntero a descriptor del blockgroup
  - blockgroup_for_inode: Devuelve el número de blockgroup del inodo.
  - blockgroup_inode_index: Devuelve el offset dentro de la tabla de inodos para el inodo.
- Datos sobre el *filesystem* en general: En el superbloque.
- Datos sobre un *block group*: En su descriptor de blockgroup.
- Directorio: Un archivo común cuyo CONTENIDO son entradas de directorio como la que vimos.
- Descompriman la imagen hdd.raw.gz para usarla.
- ¡Tienen estructuras para cada tipo necesario!

Documentación Taller

- http://www.nongnu.org/ext2-doc/ext2.html
- http://e2fsprogs.sourceforge.net/ext2intro.html
- http://wiki.osdev.org/Ext2
- http://oreilly.com/catalog/linuxkernel2/chapter/ch17.pdf

Ejercicio

Ejercicio de tarea para despues del taller

Se tiene un disco rígido con un sistema de archivos de tipo FAT. Se pide programar el algoritmo

```
loadFile(const char * path, void * buffer)
```

Para ello utilice las siguientes estructuras y funciones.

Estructuras

```
struct F32BR {
        unsigned char jumpCode[3];
        unsigned char OEMName[8];
        unsigned short bytes_per_sector;
        unsigned char sectors_per_cluster;
        unsigned short reserved_sectors;
        unsigned char number_of_fat_copies;
        unsigned short max_dir_entries_for_root;
        unsigned short small_num_of_sectors;
        unsigned char media_descriptor;
        unsigned short sectors_per_FAT;
        unsigned short sectors_per_track;
        unsigned short number_of_heads;
        unsigned int hidden_sectors;
        unsigned int number_of_sectors;
        unsigned short logical_drive_number;
        unsigned char extend_signature;
        unsigned int serial_number;
        unsigned char volume_name[11];
        unsigned char FAT_name[8];
        unsigned char code[448];
        unsigned char boot_record_signature[2];
} fat_bs; // Ya esta cargada en memoria
```

Mas estructuras

```
void hdd_load_sector(unsigned int sector, void * buffer);

struct dir_entry {
        unsigned char name[8];
        unsigned char extension[3];
        unsigned char attributes;
        unsigned char nada[10];
        unsigned short time;
        unsigned short date;
        unsigned short cluster;
        unsigned int file_size;
};

bool is_dir(struct dir_entry * de);
bool is_file(struct dir_entry * de);
int split_path(const char * path, char*** array);
```

- Documentación sobre FAT 32:
  - http://home.teleport.com/~brainy/fat32.htm
  - http://wiki.osdev.org/FAT

Algunas aclaraciones

- Consideraciones:
  - En FAT32 las direcciones de sectores son de 28 bits, los 4 más altos se ignoran.
  - En FAT32 se le llama cluster al sector en disco y sector al bloque.
  - En FAT32 hay varias FAT por redundancia.
  - Los directorios empiezan donde indica la entrada de directorio y contienen entradas de directorios consecutivas. Asuman que terminan en 0.
  - En FAT32, el layout del disco es

F32BR	Sectores Reserv.	FAT1	FAT2	RootDir	Datos

Dudas

- ¿Dudas?
- ¡A programar!