DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico Final

12 de julio de 2016 — Organización del Computador II

Alumno	LU	Correo electrónico
Gonzalo Ciruelos Rodríguez	063/14	gonzalo.ciruelos@gmail.com

Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2160 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (54 11) 4576-3359

http://www.fcen.uba.ar

Índice general

Introducción **4**

Trabajo previo 5

Motivación de este trabajo 5

1. ext2 **6**

1.1. El sistema de archivos ext2 6

1.1.1. Descripcion detallada de ext2 7

1.1.2. Otras implementaciones 14

1.2. Nuestra implementación 15

1.2.1. API de IO de DeliriOS 15

1.2.2. Visión general de la implementación 16

1.2.3. Visión detallada de la implementación 18

1.2.4. Cache de bloques 22

1.2.5. Bitmap 23

1.2.6. Testing 24

2. Dispositivos **26**

2.1. PCI 26

2.1.1. Cómo detectar dispositivos PCI 27

2.1.2. Detección de dispositivos PCI en DeliriOS 27

2.2. HDD 32

2.2.1. IDE 32
2.2.2. ATA PIO 33

3. Resultados **36**
3.1. Evaluación de nuestra implementación 36
3.2. Performance de nuestra implementación 36

4. Conclusión **37**
4.1. Trabajo futuro 37

Introducción

DeliriOS es un exokernel bare-metal. Su objetivo es proveer una base de desarrollo para programas donde ocurren muchas operaciones de sincronización entre threads.

El objetivo original de este trabajo era implementar, sobre el trabajo de Sebastián Nale y Julián Pinelli, el módulo de escritura para ext2. Luego, el trabajo derivó en otros asuntos que fui resolviendo, como el diseño de la API de IO de DeliriOS (hecho en conjunto con Sebastián y Julián) y la refactorización del driver para discos rígidos (IDE) y la detección de dispositivos PCI (requerida para la refactorización del driver).

Todos los objetivos fueron cumplidos, y el objetivo de este trabajo es explicarlos, documentarlos y exhibir resultados generales sobre los módulos implementados.

Trabajo previo

- Trabajo de noit y Goldsmith.

Motivación de este trabajo

Para poder ejecutar programas que analizan datos utilizando DeliriOS, necesitamos poder leer y escribir datos en un dispositivo de memoria no volátil, por ejemplo, un disco rígido (o discos de estado sólido).

La información dentro de un dispositivo de almacenamiento se organiza utilizando una especificación conocida como *filesystem* o sistema de archivos.

Las principales funciones de un sistema de archivos son la asignación de espacio a los archivos, la administración del espacio libre y del acceso a los datos guardados. Los sistemas de archivos determinan la estructura de la información guardada en el dispositivo de almacenamiento.

Los sistemas de archivos son muy variados, pero todos pueden ser clasificados en unas pocas familias. Una de esas familias es conocida como sistema de archivo basado en *inodos*. Entre ellos se encuentra el sistema de archivos *ext2*, que es el implementado en este trabajo.

Elegimos ext2 porque, desde nuestro punto de vista, es el balance ideal entre facilidad de implementación y *features*.

En el extremo de la facilidad de implementación se encuentran sistemas de archivos como *FAT*, pero su velocidad y sus prestaciones son muy pobres. En el otro extremo se encuentran sistemas de archivos como *ZFS* o *ext4*, que son muy rápidos y tienen muchas prestaciones, pero son difíciles de implementar, en comparación con ext2. Sin embargo, debe tenerse en cuenta que ext4 es una mera extensión (de hecho compatible hacia atrás) con ext2, con lo cual nuestra implementación puede extenderse a ext4 en caso de desearse.

Capítulo 1

ext2

1.1. El sistema de archivos ext2

Como dijimos anteriormente, ext2 es un filesystem basado en inodos. Los sistemas basados en inodos (explicaremos qué son los inodos más adelante) fueron creados para el sistema operativo UNIX, y son usados por todos sus descendientes.

El sistema ext2 (por *second extended*) es un sistema de archivos diseñado e implementado para ser usado por el kernel de Linux. Fue inicialmente diseñado por Rémy Card como un reemplazo para el *extended filesystem* (ext).

La implementación canónica de ext2 es el driver *ext2fs* del kernel de Linux. Otras implementaciones existen en GNU Hurd, MINIX 3, y kernels de BSD.

ext2 fue el sistema de archivos por defecto en varias distribuciones de Linux, hasta que fue suplantado más tarde por ext3, que es un sistema de archivos con *journaling*. ext2 es aún hoy el sistema de archivos de elección para dispositivos de almacenamiento flash, dado que su falta de *journal* mejora la performance y minimiza la cantidad de escrituras.

Como ext2 fue diseñado e implementado para Linux, carece de una especificación canónica, por lo que esto dificulta la implementación de cero.

1.1.1. Descripcion detallada de ext2

El sistema de archivos ext2 usa:

1. bloques unidad básica de almacenamiento,
2. inodos como medio de tener registro de archivos y objetos de ext2,
3. grupos de bloques para dividir lógicamente el disco en secciones más manejables,
4. directorios para proporcionar una organización jerárquica de archivos,
5. bitmaps de bloques e inodos para tener registro de bloques e inodos reservados,
6. superbloques para definir los parámetros del sistema de archivo y su estado general.

Bloques

Una partición o disco formateada con el sistema de archivos ext2 está dividida en pequeños grupos de sectores llamados *bloques*. Estos bloques están agrupados, además, en unidades más grandes llamadas grupos de bloques.

El tamaño del bloque se determina cuando se formatea el disco y tiene impacto en la performance, el máximo tamaño de archivo posible y el máximo tamaño del sistema de archivos. Los tamaños de bloques más comunes son 1KiB, 2KiB, 4KiB and 8KiB, aunque los últimos 2 son los más usados en el último tiempo, dado el gran tamaño de los discos actuales.

Grupos de bloques

Los bloques se agrupan en grupos de bloques para reducir la fragmentación y minimizar la cantidad de movimientos del cabezal del disco cuando se lee una gran cantidad de datos consecutivos. La información sobre cada grupo se almacena en una tabla de descriptores que se encuentra en los bloques inmediatamente después del superbloque. Dos bloques cerca del inicio de cada grupo están reservados para él bitmap de uso de bloques y el bitmal de uso de

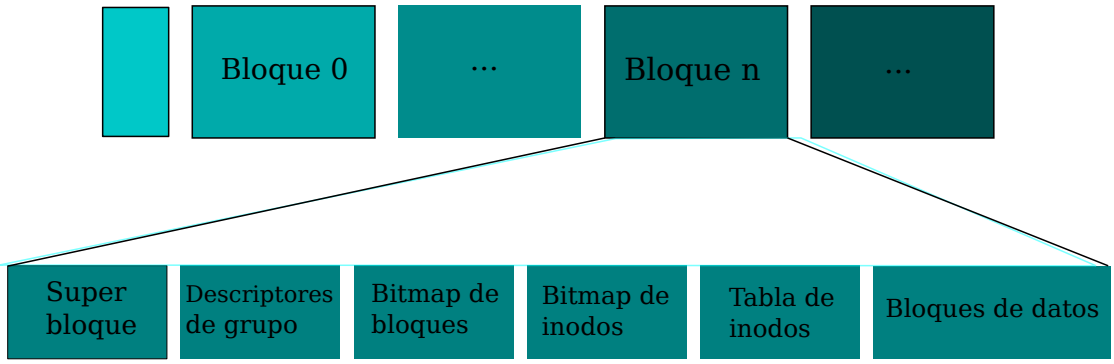

Figura 1.1: Estructura de un grupo de bloques.

inodos, que muestran cuales bloques e inodos estan siendo usados. Como cada bitmap está limitado a un único bloque, esto significa que el tamaño máximo de un bloque de grupos es 8 veces el tamaño del un bloque.

Los bloques que siguen a los bitmaps en cada grupo de bloques están designados a la tabla de inodos para ese grupo y el resto son bloques de datos.

Inodos

Como se ha dicho anteriormente, cada inodo representa un archivo o directorio. Los inodos contienen metadatos relevantes del archivo, a destacar:

- Los permisos de acceso. Incluyendo el clásico número octal de 3 cifras más otros 3 bits: Sticky bit que vale más que nada para forzar privilegios en los archivos de un directorio y los set process User/Group ID que setea los privilegios de Usuario/Grupo del archivo respectivamente al proceso si se ejecutase el archivo.
- El tipo de archivo. Puede ser: Archivo común, directorio, block device, character device, socket, fifo o symlink.
- El tamaño en bytes del archivo, vale para todos los tipos incluyendo directorios, más adelante se hablará de esto.
- Los tiempos de acceso, borrado, creación, etc.

Por último lo más importante: contiene los números de los bloques que una vez concatenados formarían el archivo completo. Un lector atento se dará

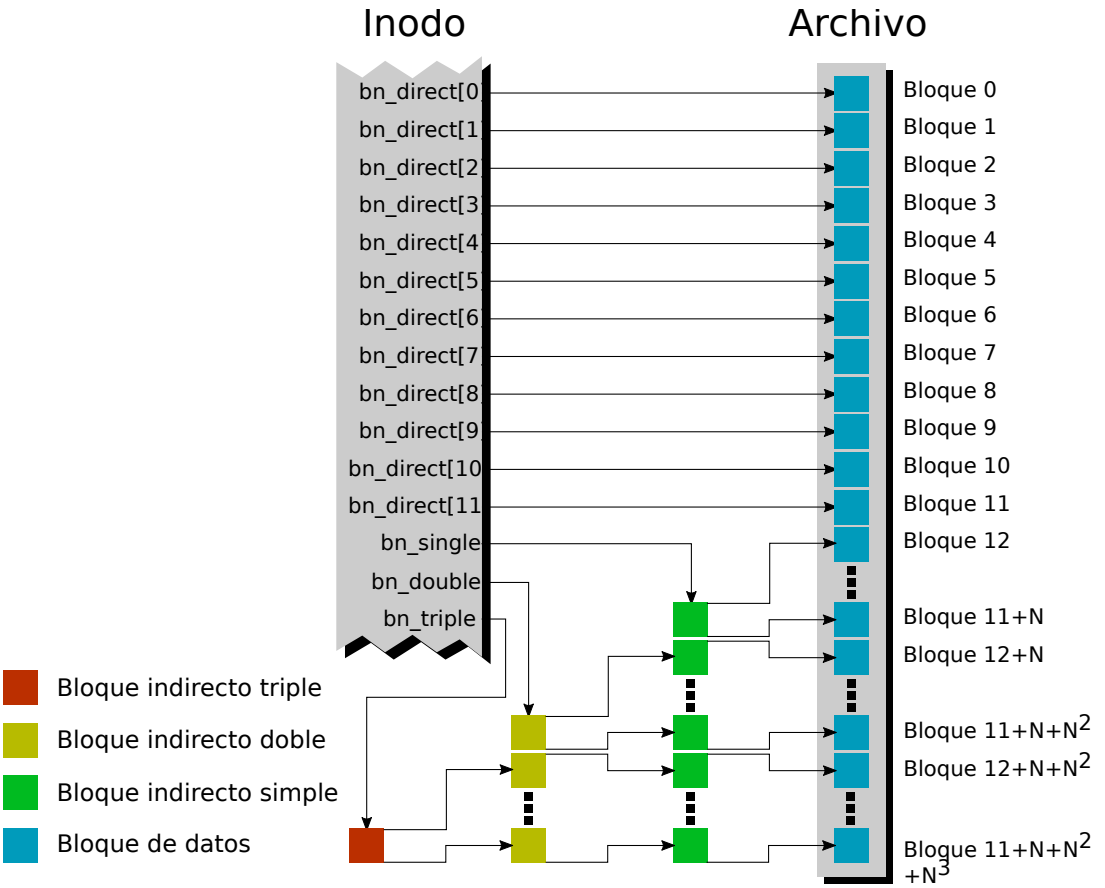

Figura 1.2: Estructura de un archivo junto con el inodo que lo representa, donde N es la cantidad de números de bloque que entran en un bloque. Esto es $N = \frac{blocksize}{4}$, ya que un número de bloque son 4 bytes.

cuenta de que debido al tamaño fijo de los inodos, no sería posible representar archivos de gran tamaño ya que esto supondría almacenar muchos números de bloque, podemos despejar dicha inquietud con la siguiente imagen:

Hay dos tipos distintos de bloques, estos son:

De datos Estos bloques forman el contenido del archivo. Se los identifica por su número absoluto en el filesystem como dijimos antes, en el inodo hay exactamente 12 de estos números (en orden) que apuntan a 12 bloques de datos, se los llama punteros a bloque directos. Si el archivo es lo suficientemente chico podrían no usarse los 12 punteros, en cuyo caso el resto de los números son inválidos. Los números que se les otorgaron en la imagen son relativos al archivo partiendo desde el 0, para poder comprender las distancias entre cada uno de los bloques mostrados y la magnitud que puede llegar a tener un archivo, no confundir con el número absoluto que los representa en el filesystem. Este número otorgado va a ser importante para entender la implementación más adelante.

Indirecto Estos bloques contienen punteros a otros bloques, esto es un arreglo de números de bloque que apuntan a otros bloques. Como dice la imagen, la cantidad de números de bloque que contienen es $\frac{blocksize}{4}$, ya que un número de bloque son 4 bytes. Podrían contener menos si el archivo no es lo suficientemente grande, en cuyo caso el resto de los valores son inválidos.

Y a su vez hay 3 tipos de bloques indirectos, en tres niveles de profundidad:

Simple Contiene punteros a bloque directos. El inodo contiene un número de bloque apuntando a uno de estos (bn_single en la imagen), si es que el tamaño del archivo es suficiente, el cual se lo denomina puntero a bloque simple.

Doble Contiene punteros a bloques simples, el inodo contiene un número de bloque apuntando a uno de estos (bn_double en la imagen) si es que el tamaño del archivo es suficiente, el cual se lo denomina puntero a bloque doble.

Triple Contiene punteros a bloques dobles. El inodo contiene un número de bloque apuntando a uno de estos (bn_triple) en la imagen, si es que el tamaño del archivo es suficiente, el cual se lo denomina puntero a bloque triple. Como es el nivel más grande posible de bloque indirecto solo puede haber uno por archivo, que es el apuntado por el inodo.

Gracias a esto, un inodo puede apuntar a una inmensa cantidad de bloques utilizando dichos bloques indirectos. La cantidad máxima de bloques que puede tener un archivo está delimitada por el tamaño de bloque, debido que a su vez esto delimita la cantidad de números de bloque dada por $N = \frac{blocksize}{4}$. En total nos quedan $12 + N + N^2 + N^3$ bloques de datos[1]. Para bloques de 1KiB, esto es 16GiB de tamaño máximo de archivo, sin contar el costo de los bloques indirectos.

Directorios

Lo único importante que nos falta para entender cómo leer archivos de EXT2 son los directorios. En EXT2, estos son simplemente un tipo especial de archivo. Esto es, están representados por un inodo al igual que un archivo. Se los diferencia con el campo de tipo en el inodo como vimos anteriormente, y poseen una estructura particular dentro del contenido del archivo que define el contenido del directorio.

Notar que es aquí donde se almacenan los nombres de los archivos dentro del directorio y no en el inodo del archivo. Basta con conocer el inodo del directorio raíz, para poder comenzar a recorrer el filesystem. Alegrará saber que dicho inodo está reservado y es siempre el número 2.

Solo falta entender la estructura interna del directorio. Está formado por entradas de directorio una seguida de la otra, de tamaño variable. A su vez cada entrada tiene los siguientes campos:

Inodo El número de inodo del archivo al que estamos apuntando en esta entrada. Si es 0, la entrada se considera nula, esto es libre para ser usada en caso de tener que escribir una entrada o libre de ser ignorada en caso de tener que avanzar.

Tamaño de la entrada El tamaño total de la entrada, contando todos los campos incluyendo el nombre y el padding necesario hasta la próxima entrada. Debe ser múltiplo de 4. Necesario para saber dónde está ubicada la

[1] Si tomamos los 12 bloques apuntados por el inodo, más los N bloques apuntados por el bloque indirecto simple, más los N^2 bloques apuntados por los N bloques simples apuntados por el bloque indirecto doble, más los N^3 bloques apuntados por los N^2 bloques simples apuntados por los N bloques dobles apuntados por el bloque indirecto triple

siguiente entrada. Para decirlo de otro modo, la suma de estos campos de todas las entradas debería dar el tamaño del bloque.

Tipo de archivo Esto originalmente era la parte alta del tamaño del nombre que es el siguiente campo, pero si se activa un flag en el superbloque (Que en general es activado de facto) se toma como tipo de la entrada. Notar que no son los mismos valores que en POSIX, sino un número del 0 al 7, ver tabla más adelante.

Tamaño del nombre El tamaño del nombre que comienza justo después de este campo. Debido a que el otro campo es generalmente usado para el tipo nos da un tamaño de nombre máximo por archivo de 255 caracteres.

Nombre El nombre del archivo, **no necesariamente un null terminated string**, hay que utilizar el tamaño del nombre para poder leerlo con seguridad.

Padding Luego del nombre viene el suficiente padding para completar el tamaño de la entrada.

Los valores para los tipos son:

Valor	Tipo
0	Indefinido/NULL
1	Archivo común
2	Directorio
3	Char Device
5	Block Device
4	FIFO
6	Socket
7	Symbolic Link

Podemos ver un ejemplo de esto en la siguiente imagen:

Es un directorio raíz (Inodo 2), que sucesivamente, contiene las siguientes entradas:

. Esta es una entrada estándar de tipo directorio (2) que poseen todos los directorios, la cual apunta al mismo directorio en cuestión, en este caso el inodo 2 que es este mismo.

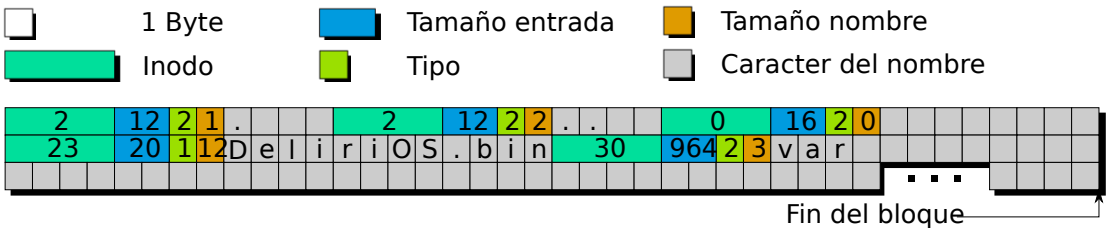

Figura 1.3: Estructura de un directorio raíz con bloques de 1KiB

.. Esta es otra entrada estándar de tipo directorio (2) que poseen todos los directorios, apunta al directorio que contiene a este. Al ser el directorio raíz, este se apunta a sí mismo también, el número 2.

Entrada Nula La podemos detectar por el campo de inodo en 0. Podemos ignorarla en caso de lectura o usar el espacio vacío si quisiéramos escribir una entrada adicional en el directorio, siempre y cuando alcance el espacio.

DeliriOS.bin Este es un archivo, lo podemos detectar por el tipo (1). Está apuntando al inodo 23, si quisiéramos leerlo solo tenemos que cargar el inodo 23 y parsearlo correctamente.

var Otro directorio (Tipo 2), ubicado en el inodo 30, si quisieramos leer su contenido tenemos que repetir el proceso actual pero con el inodo 30. Al ser la última entrada, necesariamente el tamaño abarca hasta el final del bloque. Para asegurarnos que llegamos al final, ya que podría haber más entradas en el bloque siguiente, chequeamos el tamaño del directorio (en este caso el raíz), si es 1024 entonces llegamos al final[2].

Con todo esto, ya podemos entender casi en completitud todo ext2, así que podemos pasar a ver las implementaciones del sistema de archivos.

Como dijimos anteriormente, ext2 carece de una especificación clara y canónica. Todas las implementaciones de ext2 parten de analizar el código de la implementación de Linux e imitarlo.

Sin embargo, existe una documentación no oficial, escrita por Dave Poirier, que es en la que nos basamos para realizar este trabajo [Poi11].

[2]Esto obliga a que el tamaño de los directorios sea siempre múltiplo del tamaño del bloque.

1.1.2. Otras implementaciones

- Breve descripción de como otros sistemas operativos resolvieron el problema de los filesystems en general y ext2 en particular.

1.2. Nuestra implementación

Este trabajo se trata sobre la implementación de la parte de escritura de ext2, es decir, las funciones que se ocupan de modificar el sistema de archivos de cualquier manera (creando, modificando y borrando archivos, en general).

La parte de lectura del sistema de archivos (carga, interpretación de flags de opciones y lectura de archivos) puede encontrarse muy bien explicada en el trabajo de noit y Goldsmith.

1.2.1. API de IO de DeliriOS

Primero comencemos viendo la API de IO de DeliriOS, dado que esto nos va a permitir motivar y explicar todas las funciones de ext2 que implementamos.

```
int64_t read(int64_t fd, void *buf, uint64_t count)

int64_t write(int64_t fd, const void *buf, uint64_t count)

int64_t seek(int64_t fd, int64_t offset, int64_t origin)

int64_t tell(int64_t fd)

uint8_t type(const char *path)

int64_t create(const char *path, uint8_t filetype)

int64_t remove(const char *path)

int64_t truncate(int64_t fd, uint64_t length)

int64_t open(const char *path)

int64_t close(int64_t fd)

int64_t opendir(const char *path)

int64_t closedir(int64_t dd)

int64_t rewinddir(int64_t dd)

dir_entry * readdir(int64_t dd)
```

Como puede verse, es básicamente la API estándar de UNIX, nos inspiramos en ella porque de esta manera sería más fácil hacer programas en DeliriOS.

Las funciones en las que nos concentraremos en este trabajo son: write, create, remove y truncate, dado que son aquellas para las cuales se necesita modificar el sistema de archivos.

El resto de las funciones son de lectura y fueron debidamente explicadas en el trabajo de noit y Goldsmith.

1.2.2. Visión general de la implementación

Pasaremos a dar una visión general del sistema de escritura, dado que el de lectura fue analizado en el trabajo de noit y Goldsmith.

Lo que necesitamos para una implementación exitosa de ext2 es, básicamente,

- una cache de bloques,
- funciones que reserven bloques e inodos,
- funciones que escriban datos en bloques,
- funciones que modifiquen inodos.

La función central, que será la de escribir, tiene la siguiente pinta.

```
1  uint64_t fs_write(fs_info* info, fs_file* file,
2                    const void *buffer, uint64_t count) {
3
4      block_cache_set_dirty(info->head_cache, info->sb);
5      block_cache_set_dirty(info->head_cache, info->gt);
6
7      block_cache* cache = info->cache;
8      uint64_t block_shift = info->block_shift;
9
10     uint64_t block_size = 1 << block_shift;
11     // remaining_after_eof es el numero de bytes que hay despues de que
           el
12     // archivo  termina, porque no esta usando todo el bloque.
13     // Cuando file->block_offset == block_size-1,
14     // quiero que remaining_after_eof valga 0
15     uint64_t remaining_after_eof =
16         ( - file->remaining - file->block_offset) % block_size;
```

```
17      // Escribo primero hasta el final del ultimo bloque, porque seguro
18      // puedo hacerlo. fst_count es eso, la cantidad de bytes que puedo
19      // escribir hasta quedarme sin bloques.
20      uint64_t fst_count = MIN(count, file->remaining +
            remaining_after_eof);
21      uint64_t written = 0;
22
23      uint64_t size = count;
24      while(size){
25          // block_size - file->block_offset es lo que me queda por
                escribir
26          // de un bloque.
27          uint64_t to_copy = MIN(size, block_size - file->block_offset);
28          // si me pase de lo que tenia alocado, tengo que pedir mas
29          if(written >= fst_count){
30              // pido un bloque, si devuelve distinto de 0, salgo
31              if(ext2_request_block(info, file)) break;
32          }
33          void* block_ptr = block_cache_load_dirty(cache, file->
                block_number);
34          void* base = (void*) ((uintptr_t) block_ptr + file->block_offset
                );
35          memcpy(base, buffer, to_copy);
36          block_cache_unload(cache, block_ptr);
37          // me fijo cuanto tengo que correr el eof (file->remaining)
38          if(file->remaining < to_copy){
39              uint32_t eof_advanced = to_copy - file->remaining;
40              block_cache_set_dirty(cache, file->inode);
41              // actualizo el tamano del inodo, me fijo primero si hay
                    overflow
42              if(file->inode->size_low + eof_advanced < file->inode->
                    size_low)
43                  file->inode->size_high++;
44              file->inode->size_low += eof_advanced;
45              file->remaining += eof_advanced;
46          }
47          ext2_seek(cache, block_shift, file, to_copy);
48          buffer = (void*) ((uintptr_t) buffer + to_copy);
49          size -= to_copy;
50          written += to_copy;
51      }
52      return written;
53  }
```

Veamos como interviene cada punto que nombramos antes en esta función.

- una cache de bloques: cuando queremos escribir un bloque, lo cargamos en la caché, lo escribimos, lo marcamos como "sucioz lo descargamos, todas estas son funciones que deberán ser implementadas por la cache de bloques,
- funciones que reserven bloques e inodos: si el archivo no tiene más es-

pacio disponible, debemos llamar a la función ext2_request_block, que se ocupa de reservar un bloque nuevo y actualizar todas las estructuras correspondientes,

- funciones que escriban datos en bloques: la función fs_write en si,
- funciones que modifiquen inodos: la función fs_write en si modifica inplace al inodo (actualizando por ejemplo el tamaño), pero también otras son necesarios otros procedimientos que serán utilizadas por funciones que creen y borren archivos.

Cuando se cargue un archivo para escritura, se reservaran PREALLOC_BLOCK_COUNT cantidad de bloques contiguos, de tal manera de tener un buffer de escritura que permita minimizar la fragmentación del archivo.

1.2.3. Visión detallada de la implementación

Pasaremos a describir función por función qué hace cada una y que rol cumple en el sistema general.

```
uint64_t fs_write(fs_info* info, fs_file* file, void *
    buffer, uint64_t count)
```

write escribe a partir del offset en el que se encuentra, la cantidad de bytes indicada por count del buffer.

Pueden pasar 3 cosas: 1. Entra la información y no hay que hacer ningún tipo de modificación al inodo. 2. Entra la información en los bloques que tenemos actualmente, pero el EOF se ve modificado, esto debe verse reflejado en el tamaño del archivo, que se guarda en el inodo. 3. La información no entra en los bloques asociados al archivo y debemos pedir más.

Es importante recordar que cierta información puede verse modificada (en particular el superbloque o la tabla de descriptores de grupos) si cambia la cantidad de bloques libres, entonces debemos marcar nuestras copias en la cache como /dirty/ para que sus cambios sean commiteados más tarde.

Cuando write encuentra que debe seguir escribiendo pero no tiene más bloques para hacerlo, llama a ext2_request_block, que se ocupa de poner un bloque para que write pueda seguir escribiendo sin hacer ningun tipo de trabajo extra.

```
uint32_t ext2_request_block(fs_info* info, fs_file* file)
```

Se ocupa de calcular cuantos bloques son necesarios para brindar un bloque de datos (dado que por la estructura de ext2 pueden llegar a ser necesarios más de uno) para ello llama a ext2_needed_blocks.

Luego chequea si el archivo tiene suficientes bloques preasignados y en caso contrario llama a ext2_balloc para que le asigne bloques al archivo.

Luego actualiza el inodo y asigna los bloques, armando correctamente la estructura de arbol del inodo de ext2, de tal manera de dejarle el bloque colocado de manera correcta a write para que pueda seguir escribiendo; para esto se utiliza la función ext2_assign_blocks.

```
uint64_t ext2_needed_blocks(fs_info* info, fs_file* file)
```

Cuenta cuantos bloques reales necesito para devolver un bloque real de datos.

```
uint32_t ext2_balloc(fs_info* info, fs_file* file)
```

Intenta alocar una constante fija de bloques para el archivo dado. La constante esta dada por PREALLOC_BLOCK_COUNT. Se podria usar tambien la constante que viene dada en el superbloque, pero es opcional.

El algoritmo es simple e intenta minimizar la fragmentacion externa: intenta alocar desde el ultimo bloque utilizado por el archivo en adelante. Si no encuentra en el grupo actual, sigue con los grupos siguientes.

En caso de que el ultimo bloque del archivo sea el 0 (o sea, no tiene bloques asignados hasta ahora, por ejemplo, si el archivo estaba vacio) entonces se empezará a buscar desde el primer grupo.

Para llevar esto a cabo, se debe cargar el block usage bitmap de cada grupo de bloques y usar la funcion bitmap_search_zeros que busca n (en este caso PREALLOC_BLOCK_COUNT) bits en 0 juntos y devuelve el offset en el que se encuentran.

En caso de tener exito, los reserva usando la funcion ext2_set_resvd_blocks.

```
uint32_t ext2_assign_blocks(fs_info* info, fs_file* file)
```

Arma la estructura de arbol del inodo como corresponda para expandir el tamaño efectivo del archivo en 1 bloque. Para esto, tiene que eventual-

mente escribir nuevos bloques indirectos simples, dobles o triples.

Esta es definitivamente la función más compleja del driver de ext2 (junto con ext2_unassign_blocks que es prácticamente igual), sin embargo la idea es muy simple.

Se divide en todos los casos posibles y se asignan todos los bloques según corresponda.

```
uint32_t ext2_set_resvd_blocks(fs_info* info, uint32_t
    from, uint32_t count, bool reserve)
uint32_t ext2_set_resvd_inodes(fs_info* info, uint32_t
    from, uint32_t count, bool reserve)
```

Estas funciones, muy similares, se ocupan de reservar/liberar bloques/inodos en el filesystem. Para eso, debe cargar el usage bitmap que corresponda y setear o limpiar los bits que correspondan, si se quiere reservar o liberar, respectivamente.

El cuarto parametro indica si lo que se quiere hacer es reservar o desreservar. Si es true se reserva y si es false se libera.

```
uint32_t ext2_get_next_prealloc(fs_file* file)
```

Devuelve el proximo bloque de los prealocados por el archivo. Además actualiza la estructiura interna del archivo (dado que aumenta el puntero al primer bloque prealocado y disminuye la cantidad de bloques prealocados).

```
uint32_t ext2_ialloc(fs_info* info, fs_dir* parent)
```

Busca, empezando por el grupo 0, algun inodo libre y devuelve su numero.

```
int64_t fs_create(fs_info* info, fs_dir* dir, dir_entry*
    file_info)
```

Crea un archivo en la carpeta pasada como parametro.

Para eso, pide un inodo con ext2_ialloc y escribe la direntry correspondiente en los bloques de datos del inodo del directorio. Hay dos casos: se puede meter en el medio de la carpeta (porque hay entradas obsoletas en el medio) o debe ponerse al final.

Además, debe setear información como mode (permisos), creation time, modification time, entre otros.

```
uint32_t ext2_unassign_block(fs_info* info, fs_file* file)
```

Le desasigna el bloque actual al archivo, que debe ser el ultimo (a menos que se quiera romper todo, o se sepa lo que se hace, dado que se rompe el invariante del archivo y queda un "hueco.en el medio).

La lógica es la misma que la de ext2_assign_blocks, solo que se deben descargar aquellos indirectos que están cargados y no se necesitan más tambien.

```
int64_t fs_truncate(fs_info* info, fs_file* file, uint64_t
    length)
```

Trunca al archivo y lo deja de tamaño length (bytes). Se asume que length es menor al tamaño del archivo (en caso de que no, se resuelve en la interfaz de io.c).

Para eso va hasta el final del archivo y va retrocediendo, desasignando bloques con ext2_unassign_block en caso de que sea necesario, actualizando toda la información que hace falta (el tamaño del archivo, que se encuentra en el inodo).

```
int64_t fs_remove(fs_info* info, fs_dir* dir)
```

Borra el archivo apuntado actualmente por el directorio.

Hay que sobreescribir la entrada marcandola como invalida (seteando el numero de inodo en 0) y el deletion time acordemente.

```
uint32_t ext2_has_super(fs_info* info, uint64_t group)
```

Dice si un grupo del filesystem debe contener o no un backup del superbloque y el group descriptor table.

Si el filesystem tiene seteado la feature de sparse superblock, hay que chequear si el numero de grupo es 0 o potencia de 3, 5 o 7.

```
uint32_t ext2_restore_backup(fs_info *info)
```

Escribe el superbloque y la group descriptor table en todos los grupos que deben tener el backup. Para eso, se va a usar ext2_has_super.

```
uint64_t ext2_new_group(fs_info* info);
```

Esta funcion se encarga de inaugurar nuevos grupos de bloques. Debe tener en cuenta no pasarse del tamano del drive y mantener el superbloque esparso si asi corresponde.

De todos modos, esta funcion se usa en casos muy borde, cuando nos quedamos sin el espacio actual alocado en todo el filesystem. De hecho la especificacion ni siquiera es clara si esta operacion se puede hacer mientras el disco esta online y montado.

Esta función no fue implementada, simplemente está su signatura, dado que no es una función estándar (linux no asegura que esto no rompa la partición) y sólo algunos sistemas operativos la implementan.

```
uint32_t ext2_group_first_block(fs_info* info);
```

Se trata de una función auxiliar que devuelve el primer bloque de un grupo. Es utilizada sobre todo para generar offsets

```
int32_t ext2_add_direntry(fs_info* info, fs_dir* dir, char
    * filename, uint32_t inode, uint16_t filetype);
```

Esta función se utilizará para crear nuevos archivos dentro de un directorio.

Primero, esta función generará el struct con toda la información del archivo.

Simplemente esta función recorre todo el directorio (que recordemos que es un archivo) buscando alguna posición vacía. Cuando la encuentre, escribirá el struct generado anteriormente ahí. En caso de que no haya lugares vacíos, lo escribirá al final. Y en el caso de que no haya más lugar en el archivo, el archivo se extenderá con bloques nuevos.

1.2.4. Cache de bloques

La implementación general de la cache de bloques fue detallada en el trabajo de noit y Goldsmith. Sin embargo, debió ser cambiada para implementar la parte de escritura de ext2.

En particular, debieron ser agregadas las siguientes funciones.

```
1 static void block_cache_flush_entry(block_cache* cache, uint64_t
      entry_number) {
2     block_cache_entry* vec = cache->vec;
3     uint64_t block_number = vec[entry_number].number;
4     hdd_write(block_number * cache->size_sectors + OFFSET_SECTORS,
5               cache->size_sectors, vec[entry_number].data);
6     kfree(vec[entry_number].data);
7     vec[entry_number].data = NULL;
```

```
8       vec[entry_number].flags &= ~BLOCK_CACHE_DIRTY;
9   }
```

```
1   void block_cache_set_dirty(block_cache *cache, void *block_ptr) {
2       if (block_ptr == NULL)
3           return;
4       block_cache_entry* vec = cache->vec;
5       // The index
6       uint64_t i = 0;
7       // We look it up and set the DIRTY flag on
8       uint64_t count = cache->count;
9       uintptr_t block_int = (uintptr_t) block_ptr;
10      while (i < count) {
11          // If it is any address inside the block
12          if (block_int >= (uintptr_t) vec[i].data &&
13              block_int < ((uintptr_t) vec[i].data + cache->size_bytes)){
14              vec[i].flags|=BLOCK_CACHE_DIRTY;
15              break;
16          }
17          i++;
18      }
19
20      if (i >= cache->count) {
21          KERNEL_PANIC("Direccion %p no es valida en ningun bloque",
                block_ptr);
22      }
23  }
```

Las funciones son bastante autoexplicativas.

La primera seatea el bloque en cuestión como no usado y lo escribe a disco. Además, libera el espacio en memoria que se pidió para guardar el bloque.

La segunda función marca un bloque como dirty. La única función de este procedimiento es permitir una optimización, dado que marcando bloques como dirty podemos ahorrarnos escrituras. Si al liberar un bloque, este esta marcado como dirty, debemos escribirlo a disco. En cambio, si no está marcado como dirty, nos podemos ahorrar esta escritura, dado que sabemos que no fue modificado.

1.2.5. Bitmap

Para toda la manipulación de los bitmaps de uso de bloques e inodos debí implementar funciones de bitmaps. Estas se encuentran en utils/bitmap.c, y son muy fáciles de entender. Solamente pondré sus signaturas y una breve descripción de cada una.

```
uint64_t clz8(int8_t byte)
uint64_t ctz8(uint8_t byte)
```

Cuentan cantidad de ceros delante/detras del número (count leading/trailing zeros). Esperan un numero de 8 bits.

```
uint64_t bitmap_count_zeros(const void* const _bitmap,
   uint64_t size, uint64_t offset)
```

Cuenta la cantidad de ceros consecutivos en el bitmap desde offset hasta size. Toma como parametros el bitmap en cuestion, el tamaño del bitmap en bits y desde que bit empezar a buscar ceros.

```
uint64_t bitmap_search_zeros(const void* _bitmap, uint64_t
    size, uint64_t offset, uint64_t count)
```

Busca count cantidad de bits en cero en el bitmap. Recibe como parámetros el bitmap en cuestión, el tamano del bitmap en bits, desde que bit empezar a buscar ceros y cuántos ceros buscar.

Devuelve la posición a partir de la cual encontro los bits. Si es size, significa que no encontro.

```
uint64_t bitmap_set_bits(const void* _bitmap, uint64_t
   size, uint64_t offset, uint64_t count)
uint64_t bitmap_clear_bits(const void* _bitmap, uint64_t
   size, uint64_t offset, uint64_t count)
```

Modifica el bitmap poniendo en 1 (o 0, si es clear) count cantidad de bits de offset en adelante.

Devuelve 0 si salió todo bien, 1 si se pasó de size y 2 si ya estaban seteados en 1 (o 0, si es clear).

1.2.6. Testing

Para testear, diseñamos un subsistema especial, que nos permitiera probar la correctitud de las funciones implementadas desde afuera de delirios. Para esto, utilizamos la clásica técnica de dependency injection.

Tenemos una imagen de ext2 en un archivo (hdd.img), y lo que hacemos es abrirla como si fuera un archivo común y corriente, y guardamos su file handler en una variable global llamada fp.

Luego, pasamos a lo vital de la dependency injection: overloadeamos as funciones hdd_read y hdd_write de la siguiente manera.

```
1 void hdd_read(uint32_t lba_address, uint32_t sectors, void* buffer) {
2     fseek(fp, lba_address * ATA_SECTSIZE, SEEK_SET);
3     fread(buffer, ATA_SECTSIZE, sectors, fp);
4 }
```

```
1 void hdd_write(uint32_t lba_address, uint32_t sectors, void* buffer) {
2     fseek(fp, lba_address * ATA_SECTSIZE, SEEK_SET);
3     fwrite(buffer, ATA_SECTSIZE, sectors, fp);
4 }
```

De esta manera, cuando las funciones de ext2 y de la cache de bloques las llamen, van a pensar que estan leyendo de un disco, mientras que simplemente estamos escribiendo un archivo.

Así, podemos testear todas las funciones que implementamos con extrema facilidad. En particular, todas las funciones presentadas en este informe tienen tests de todo tipo: funcionales, de error, de carga, entre otros.

Capítulo 2

Dispositivos

2.1. PCI

El bus PCI (Peripheral Component Interconnect) fue definido para establecer un bus local de alta performance y bajo costo que durara varias generaciones de productos. Combinando un camino transparente de mejoras desde 132 MB/s (32-bit a 33MHz) hasta 528 MB/s (64-bit a 66MHz) y ambientes de señalización tanto de 5V como de 3.3V, el bus PCI satiface las necesidades tanto de computadoras de escritorio de bajo costo como de servidores LAN de alto costo. El bus PCI es independiente del procesador, lo cual permite una transición eficiente a futuros procesadores, además de poder usar diferentes arquitecturas de procesadores.

Las desventajas del bus PCI es que es la limitada cantidad de cargas eléctricas que puede conducir. Un único bus PCI puede conducir un máximo de 10 cargas. (Recordemos que cuando contamos la cantidad de cargas de un bus, un conector cuenta como una carga y el dispositivo PCI cuenta como una carga, o a veces dos).

Cualquier sistema operativo que se precie de serlo debe poder detectar que dispositivos están conectados a la computadora. Por esta razón, decidimos implementar un brevisimo identificador de dispositivos PCI.

Este es un paso vital para trabajar en el futuro con cualquier tipo de dispositivo, por ejemplo USB o Ethernet.

2.1.1. Cómo detectar dispositivos PCI

La especificación del bus PCI nos provee de un sistema de inicialización y configuración que puede realizarse totalmente por software via un espacio de direcciones de configuración separado para cada dispositivo en el bus PCI.

Todos los dispositivos PCI, excepto el host bridge, están obligados a proveer 256 bytes de registros de configuración para ese propósito

Los ciclos de configuración de los ciclos de lectura y escritura son usados para acceder al espacio de configuración de cada dispositivo objetivo. El objetivo es seleccionado enviando una señal ISDEL al host bridge. La señal ISDEL actua como la clásica señal çhip select". Durante la fase de direcciones del ciclo de configuración, el procesador puede direccionar alguno de los 64 registros de 32-bits dentro del espacio de configuración del dispositivo, simplemente escribiendo el número de registro requerido en las lineas de direcciones 2 a 7 (AD[7..2]) y las lineas de byte enable.

Los dispositivos PCI son inherentemente little endian, lo que significa que todos los campos de más de un byte tienen los valores menos significativos en las direcciones más bajas. Esto requiere que cuando se programa un procesador big-endian, como un Power PC, se hagan las operaciones apropiadas de byte-swapping al leer y escribir de la memoria del dispositivo. Sin embargo, como DeliriOS es un sistema operativo para arquitectura AMD64, que es little-endian, nada de esto nos interesará.

2.1.2. Detección de dispositivos PCI en DeliriOS

La función principal del módulo PCI (llamada en kernel.c, durante la inicialización del sistema) es la siguiente.

```
1  void pci_check_buses(void) {
2      uint8_t header_type = pci_read_config(get_device(0, 0, 0),
           HEADER_TYPE);
3      if ((header_type & 0x80) == 0){
4          // Un solo PCI host controller
5          check_bus(0);
6      } else {
7          // Multiples PCI host controllers. Tengo que chequear las
               funciones
8          // (0, 0, f) . Si esta up, quiere decir que hay un host
               controller ahi
```

```
9           for (uint8_t function = 0; function < 8; function++) {
10              if (pci_read_config(get_device(0, 0, function), VENDOR_ID)
                    != 0xFFFF)
11                  break;
12              check_bus(function);
13          }
14      }
15  }
```

Primero nos fijamos cuantos PCI host controllers hay. En caso de que sea uno solo, chequeo el bus 0. En caso de que sean muchos, tengo que chequear la función de cada bus, porque si está UP quiere decir que ahi hay un host controller.

Pasemos a ver el resto de las funciones.

```
1  static inline uint16_t get_device(uint16_t bus, uint16_t slot, uint16_t
       func){
2      return ((bus << 8) | (slot << 3) | func);
3  }
```

Esta función nos permite generar el código de un dispositivo, dado su bus, función y slot.

```
1  uint32_t pci_read_config(uint16_t device, uint16_t field_and_size){
2      // Armo el pedido para informacion del dispositivo
3      uint8_t field = field_and_size >> 8;
4      uint8_t size  = field_and_size & 0x00ff;
5
6      uint32_t address = ((uint32_t) ((device << 8) | (field & 0xfc)))
7                         | 0x80000000;
8
9      // Pido esa entrada de la informacion del dispositivo
10     outl(0xCF8, address);
11
12         if (size == 4) {
13                 uint32_t t = inl(0xCFC);
14                 return t;
15         } else if (size == 2) {
16                 uint16_t t = inw(0xCFC + (field & 2));
17                 return t;
18         } else if (size == 1) {
19                 uint8_t t = inb(0xCFC + (field & 3));
20                 return t;
21         }
22     return 0xFFFF;
23 }
```

Esta función nos permite leer un field del espacio de configuración de un dispositivo. Field and size es un parámetro que contiene tanto el field como el tamaño de ese field.

Para leer el field, utilizamos dos puertos de 32-bits. El primero es 0xCF8, que se llama CONFIG_ADDRESS, y el segundo es 0xCFC, que se llama CONFIG_DATA. CONFIG_ADDRESS especifica la dirección del espacio de configuración que vamos a querer acceder, mientras que escrituras sucesivas a CONFIG_DATA van a hacer generar el acceso al espacio de configuración del dispositivo PCI.

```
1 static char * lookup_device(uint8_t class, uint8_t subclass){
2     for(uint64_t i = 0; i < DEVICE_CLASSES; i++){
3         if(device_info[i].class == class && device_info[i].subclass ==
              subclass)
4             return device_info[i].info;
5     }
6     return device_info[DEVICE_CLASSES - 1].info;
7 }
```

Esta función se ocupa de, dada una clase y una sublcase, devolver un string que contiene una descripción del dispositivo (en inglés). La tabla se encuentra al final de esta sección.

Las funciones que siguen son todas las funciones que usa pci_check_buses para probar todas las combinaciones posibles de slot, funcion y bus.

```
1  static void check_slot(uint8_t bus, uint8_t slot) {
2      uint16_t device = get_device(bus, slot, 0);
3
4      if (pci_read_config(device, VENDOR_ID) == 0xFFFF)
5          return; // No existe el dispositivo
6      check_function(device);
7      uint8_t header_type = pci_read_config(device, HEADER_TYPE);
8
9      if ((header_type & 0x80) != 0){
10         // Es un dispositivo multifuncion,
11         // entonces chequeamos todas las funciones
12         for(uint8_t function = 1; function < 8; function++) {
13             uint16_t device = get_device(bus, slot, function);
14
15             if(pci_read_config(device, VENDOR_ID) != 0xFFFF)
16                 check_function(device);
17         }
18     }
19 }
```

```
1 static void check_bus(uint8_t bus) {
2     for (uint8_t slot = 0; slot < 32; slot++)
3         check_slot(bus, slot);
4 }
```

La función que sigue es la más complicada de las tres, dado que es la última, por lo tanto la que hace todo el trabajo. Aquí es donde confirmamos la

existencia del dispositivo, y donde imprimimos a salida estándar una descripción de él.

En caso de que el dispositivo sea un disco IDE, lo inicializamos con la funcion ide_init, que analizaremos en la siguiente sección.

```
void check_function(uint16_t device) {
     uint8_t class = pci_read_config(device, CLASS);
     uint8_t subclass = pci_read_config(device, SUBCLASS);
     uint16_t device_id = pci_read_config(device, DEVICE_ID);
     uint16_t vendor_id = pci_read_config(device, VENDOR_ID);
     uint8_t prog_if = pci_read_config(device, PROG_IF);
     uint8_t irq = pci_read_config(device, PROG_IF);

     pci_devices[next_pci_device++] = (struct pci_device)
                                      {device_id, vendor_id, class,
                                       subclass, prog_if, irq};

     // Imprimo toda la inforamcion sobre el dispositivo
     printf("  (PCI DEVICE) (%d,%d,%d)\t",
             device >> 8, (device >> 3) & 31, device & 3);

     char* info = lookup_device(class, subclass);
     if (info)
         printf(info);
     else
         printf("0x%x 0x%x", class, subclass);

     printf(" - [0x%x 0x%x]\n", vendor_id, device_id);
     // Aca termino de imprimir la informacion sobre el dispositivo
     // Y procedo a registrarlo como corresponda

     if (class == 0x06 && subclass == 0x04)
         check_bus(pci_read_config(device, SECONDARY_BUS));
     if (class == 0x01 && subclass == 0x01){
         ide_init(pci_read_config(device, BAR0),
                  pci_read_config(device, BAR1),
                  pci_read_config(device, BAR2),
                  pci_read_config(device, BAR3),
                  pci_read_config(device, BAR4));
     }
}
```

Por último, veamos la lista de información de dispositivos que tenemos, que nos va a servir para saber que es un dispositivo dada su clase y sublase.

```
static struct class_info device_info[] = {
    {0x01, 0x01, "IDE Controller"},
    {0x01, 0x05, "ATA Controller"},
    {0x01, 0x06, "Serial ATA"},
    {0x02, 0x00, "Ethernet Controller"},
    {0x03, 0x00, "VGA-Compatible Controller"},
    {0x06, 0x00, "Host Bridge"},
```

```
8       {0x06, 0x01, "ISA Bridge"},
9       {0x06, 0x80, "Other Host Device"},
10      {0x0C, 0x03, "USB Controller"},
11      {0xFF, 0x00, NULL},
12  };
13
14  #define DEVICE_CLASSES (sizeof(device_info)/sizeof(struct class_info))
```

2.2. HDD

1. Descripcion general de la interfaz IDE.
2. Que habia antes y que cambié.

2.2.1. IDE

IDE es una sigla que se refiere a la especificación eléctrica de los cables que conectan unidades ATA con otro dispositivo. Las unidades usan la interfaz ATA (Advanced Technology Attachment). Un cable IDE generalmente termina siendo conectado a un controlador IDE, que a su vez está conectado a un bus PCI.

Cada dispositivo IDE aparece como un dispositivo en el bus PCI. Si el código de clase es 0x01 (Mass Storage Controller) y el código de subclase es 0x1 (IDE), entonces el dispositivo en cuestión es una unidad IDE. Una unidad IDE sólo utiliza cinco registros BAR de los seis disponibles.

BAR0 Dirección base del canal primario (espacio de I/O). Si es 0x0 o 0x1, entonces el puerto es 0x1F0.

BAR1 Dirección base del puerto de control del canal primario (espacio de I/O). Si es 0x0 o 0x1, entonces el puerto es 0x3F6.

BAR2 Dirección base del canal secundario (espacio de I/O). Si es 0x0 o 0x1, entonces el puerto es 0x170.

BAR3 Dirección base del puerto de control del canal secundario (espacio de I/O). Si es 0x0 o 0x1, entonces el puerto es 0x376.

BAR4 Bus Master IDE. Se refiere ala base del rango de I/O consistiendo de 16 puertos. Los primeros 8 puertos controlan el DMA del canal primario y los segundos ocho del secundario.

Un controlador de IDE que funcione en paralelo utilizará las IRQ 14 y 15; un IDE serial solo utiliza un IRQ. Para leer esta IRQ, hay que mirar el espacio de configuración del dispositivo PCI, como fue explicado en la sección anterior.

2.2.2. ATA PIO

En la parte anterior vimos como detectar dispositivos de almacenamientoATA y como comunicarnos con ellos. Ahora vemos como escribir y leer de ellos. Esto ya estaba implementado en DeliriOS, pero de una forma incorrecta. Mi trabajo en esta parte se basó en hacer la implementación correcta, conforme al estándar.

Esta forma se basa en usar el comando IDENTIFY, que nos permitirá obtener más información del dispositivo.

PIO es una manera de leer y escribir a dispositibos ATA usando polling. Veamos de que se trata.

De acuerdo con la especificación de ATA, el modo PIO debe estar soportado por todas las unidades ATA como el mecanismo de transferencia de datos por default.

PIO, al ser un método de polling, usa una cantidad tremenda de recursos de CPU, pues cada byte de datos transferido entre el disco y el CPU debe ser enviado a atraves del puerto IO del procesador.

Más detalles de ATA PIO

La especificación de ATA está hecha sobre otra más vieja llamada ST506. Con ST506, cada disco estaba conectado a una placa controladora por dos cables, un cable de datos y un cable de comandos.

La placa controladora estaba enchufada al bus de la placa madre. El procesador se comunicaba con la placa controladora a través de los puertos de IO del CPU, que estaban directamente conectados al bus del motherboard.

Lo que la especificación original de IDE hacía era despegar las placas controladores del motherboard, y pegar un controlador en cada disco rígido, permanentemente. Cuando el procesador accede al puerto IO del disco, hay un chip que se sirve de atajo entre los pins del bus IO del procesador y el cable IDE, por lo tanto el procesador podía acceder directamente a la placa controladora de la unidad IDE. El mecanismo de transferencia de datos entre el CPU y la controladora permaneció el mismo, y hoy es llamado modo PIO.

IDENTIFY

Hasta aquí describimos lo que ya estaba en DeliriOS, que de hecho se remonta a un sistema operativo diseñado por Juan Pablo Darago (juampiOS). En este trabajo mejoramos ese código, modernizandolo y haciendolo obediente al estándar.

El comando IDENTIFY es la mejor forma de obtener información sobre los discos ATA conectados a la placa madre.

Para usar el comando IDENTIFY, primero hay que seleccionar la unidad objetivo enviando 0xA0 para la unidad maestro o 0xB0 para la unidad esclavo al puerto IO DRIVE_SELECT. Luego, deben ponerse en 0 los puertos SECTOR_COUNT, LBALO, LBAMID, LBAHI (es decir, los puertos 0x1F2 a 0x1F5). Luego, debe enviarse el comando IDENTIFY (0xEC) al puerto IO comando (0x1F7). Finalmente leer el puerto status (0x1F7) nuevamente. Si el valor leido es 0, la undiad no existe. Para cualquier otro valor, hay que hacer polling del puerto status hasta que el bit 7 (BSY) se ponga en 0.

En ese momento, si ERR está en 0, la información esta lista para ser leida del puerto de datos (0x1F0). Hay que leer 256 valores de 16 bits y guardarlos.

Alguna información interesante devuelta por IDENTIFY:

- uint16_t 0: es útil si el dispositivo no es un disco rígido (lectora de discos removibles, por ejemplo).
- uint16_t 83: el bit 10 está seteado si el dispositivo soporta LBA48 mode (el modo de direccionamiento de sectores que utiliza DeliriOS)..
- uint16_t 88: los bits en el byte bajo nos dicen si se soportan modos UDMA y el byte alto nos dice cuál modo UDMA está activo.
- uint16_t 60 y 61: tomados como un uint32_t contiene el numero total de sectores LBA de 28 bits direccionables en el disco (Si es distinto de cero, el disco soporta LBA28).
- uint16_t 60 y 61: tomados como un uint32_t contiene el numero total de sectores LBA de 28 bits direccionables en el disco (Si es distinto de cero, el disco soporta LBA28).

- uint16_t 100 a 103: tomados como una uint64_t contiene el número total de sectores LBA de 48 bits direccionables en el disco.

Capítulo 3

Resultados

3.1. Evaluación de nuestra implementación

1. Evaluación teorica (en terminos de complejidad) de los algoritmos.
2. Comparación con otras implementaciones.

3.2. Performance de nuestra implementación

1. Tomar tiempos.

Capítulo 4

Conclusión

4.1. Trabajo futuro

Bibliografía

[DB74] G. Dahlquist y A. Bjork. *Numerical Methods*. Prentice-Hall, 1974.

[Poi11] D. Poirier. *Second Extended File System*. 2011. URL: http://www.nongnu.org/ext2-doc/.