DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico Final

Revitalización de DeliriOS

Organización del Computador II

Integrante	LU	Correo electrónico
Nale Sebastián	655/11	sebinale@gmail.com
Pinelli Julián	573/13	noit63@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

Contents

1 Introducción

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

El objetivo original de este trabajo era dar soporte para la lectura de un sistema de archivos **EXT2**, la carga estática de binarios en formato **ELF64**, y algún shell sencillo que permita hacer uso de estas funcionalidades. Además de eso, agregamos un soporte de *syscalls* más directo que el convencional, un heap (*kmalloc*), y una interfaz simple similar a un VFS para poder dar soporte a múltiples filesystems más fácilmente.

Sobre la marcha renovamos por completo la base de código existente para mejorar la mantenibilidad, extensibilidad y depurabilidad.

2 Refactor

Hay varias modificaciones en la estructura, esquema de booteo, inicialización y funcionamiento que se hizo para mejorar la legibilidad de DeliriOS.

2.1 Estructura del proyecto

Se eliminaron la mayoría de los directorios y se creó una estructura desde cero. Está formada por:

/	El directorio raíz; solo contiene el makefile como punto de entrada al proyecto.
/bochs/	Contiene el bochs para correr DeliriOS.
/build/	Acá se compila DeliriOS y se crea la imagen con grub. Contiene el linker script para el kernel y también el script para generar la tabla de símbolos.
/doc/	Contiene la documentación.
/grub/	Contiene la imagen de grub legacy y su menú que levanta DeliriOS.
/hdd/	Contiene lo necesario para crear una imagen de prueba ext2 para testear DeliriOS.
/obj/	Acá se destinan los archivos objeto.
/src/	Contiene los directorios que categorizan las distintas partes del código fuente de DeliriOS.
/test/	Contiene tests para verificar la correctitud de DeliriOS.

Los directorios de /src/ son los siguientes:

fs/	Contiene el código del filesystem y la caché de bloques.
interrupt/	Contiene el código de interrupciones: La IDT, las rutinas y el IOAPIC.
io/	Contiene el código de pantalla, teclado y disco.
kernel/	Contiene el código de inicialización del kernel de los 3 modos: Real (AP), Protegido y Largo.
memory/	Contiene el código de manejo de memoria: KMalloc, Loader y ELF.
multicore/	Contiene el código de multicore: La inicializacion y el LAPIC.
shell/	Contiene el código del shell con sus comandos.
syscall/	Contiene el código de syscalls: Timer (sleep), IO (write, read, etc), exit y core_id (id y count).
utils/	Contiene código genérico Generalmente implementaciones de funciones de libc y de excepciones (Kernel Panic y Calltrace).
include/	Contiene absolutamente todos los headers de DeliriOS.
macros/	Contiene todos los macros de NASM.

Cabe destacar que los directorios de /src/ son una categorización meramente estética. Efectivamente, todos los objetos serán linkeados por igual. Es posible que la agrupación en directorios de los archivos fuente modifique el orden en que se organiza cada sección en la imagen binaria pero esto no hace ninguna diferencia significativa.

2.2 Optimización en la generación del kernel

Se reemplazaron todos los makefiles previos y scripts varios por un solo makefile. Este makefile se encarga de compilar el kernel entero desde cero, y a la vez crear la imagen con el GRUB y correr DeliriOS. Es mucho más simple que la suma de los anteriores y mucho más fácil de extender. Se simplificó el script de linkeo quitándole la alineación a página de las secciones. Ahora que DeliriOS es un binario crudo, y dado que no hay protección, no cumplen un rol fundamental. Solo hay que tener en cuenta el orden, el cual es importante para la carga *Multiboot* de la que se habla más adelante. Como ya no hay más modulos, se han eliminado muchos scripts, dejando solo algunos scripts de bash. Además, el código ahora es capaz de compilar con optimizaciones, incluyendo la mayoría de los warnings de GCC como errores. Todo esto, sumado a la reestructuración del kernel, redujo el tiempo de compilación considerablemente.

2.2.1 Makefile

Como dijimos antes, este Makefile hace exclusivamente casi todo el trabajo necesario para compilar y correr DeliriOS. Vale destacar algunas cosas:

2.2.1.1 Flags de GCC

Ha sido necesario agregar algunos flags de compilación, los actuales son:

```
# Los flags del compilador
CFLAGS = -std=c11 -m64 -Wall -Wsign-compare -Werror -Os -I$(INCLUDE_DIR)
# Sin esto no podemos hacer call trace
CFLAGS += -fno-omit-frame-pointer
# Omite la sección .eh_frames del ABI de x86_64 para hacer stack unwinding.
CFLAGS += -fno-asynchronous-unwind-tables
# No tenemos protección, el stack es super volatil
CFLAGS += -fno-stack-protector
# No compilamos contra la libc ni nada, somos un kernel
CFLAGS += -ffreestanding
# No asumir que la red zone es safe por las interrupciones
CFLAGS += -mno-red-zone
```

Mas allá de las descripciones de los comentarios:

Wsign-compare Fue necesario agregar ya que ha habido al menos un bug[1] por comparar enteros signados con no signados. Forma parte de Wextra, pero la mayoría de estos warnings son bastante molestos así que se optó por agregar este individualmente.

fno-omit-frame-pointer Ahora DeliriOS tiene capacidad de generar un calltrace si algo sale mal, por lo tanto es necesario este flag para que las optimizaciones no eliminen los *stack frames* del comienzo de las funciones.

fno-asynchronous-unwind-tables Como dice en el comentario, la ABI de x86_64 requiere una sección .eh_frames para poder hacer stack unwinding que no necesitamos (al menos por ahora). Reduce el bloat del binario.

fno-stack-protector Este flag generalmente viene activado por default en GCC, pero algunas distros de Linux por seguridad lo parchean y lo dejan desactivado por default. No tenemos protección y somos un kernel así que realmente no necesitamos stack protector.

ffreestanding No linkeamos contra nada, somos un kernel unitario.

mno-red-zone Discutiblemente uno de los mayores dolores de cabeza de este trabajo. Previo a todo el proceso de refactor adicional, una vez terminado de dar soporte ELF/EXT2 con un shell, todo parecía estar en orden con la excepción que el uso prolongado del shell iba corrompiendo de a poco la pantalla hasta tirar una exception. Resulta ser que este flag que debió haber sido agregado desde el comienzo de DeliriOS estaba faltando. La red zone, agregada en la ABI de x86_64, es una zona de 128 bytes por debajo de la posición del stack pointer que permite a las aplicaciones hacer uso de la misma para guardar datos volátiles mientras no se llame a otra función. Pero como nosotros somos un kernel, y tenemos interrupciones, si el compilador asume que tiene esa zona libre cada vez que cae una interrupción se pisan todos esos datos corrompiendo el kernel lentamente hasta su muerte. En este caso, cada vez que se presionaba una tecla, lo cual creaba el efecto antes mencionado. Sin embargo, vale señalar que la interrupción de reloj pudo haber sido la culpable de cualquier crash misterioso que haya ocurrido antes.

[1] Un gran dolor de cabeza en la función mas céntrica del driver de EXT2

2.2.1.2 Flags del Linker

Algunos flags se agregaron en el linker:

```
# Los flags del linker
LDFLAGS = -static --oformat binary -T $(LINKSCRIPT) -Map=$(KERNEL_MAP) --cref
```

static Nada de linkeado dinámico, es un binario crudo.
oformat binary No necesitamos ELF ya que ahora especificamos las secciones con *Multiboot*.
T Le indicamos dónde está el script de linkeo
Map= Le pedimos al linker el mapa de memoria de DeliriOS, muy útil para conocer la estructura exacta del binario resultante.
cref A su vez le pedimos al linker la tabla de referencias entre objetos, útil para generar un grafo de dependencias. Esta tabla se incluye en el archivo de mapa de memoria anterior.

2.2.1.3 Creación de la imagen de disco

La creación de la imagen de disco para levantar en los emuladores es sencilla:

```
# Target para crear la imagen de disco
$(HDD_IMAGE):
        @echo 'Creando imagen de disco con formato EXT2'
        @echo ''
        #Paso 1: Crear el .img
        dd if=/dev/zero bs=1M count=16 of=$(HDD_IMAGE)
        #Paso 2: Formatearlo con EXT2
        /sbin/mkfs -V -t ext2 $(HDD_IMAGE)
        #Paso 3: Copiar los datos de la imagen
        $(HDD_POPULATE) -d $(HDD_ROOT) $(HDD_IMAGE)
```

Se crea un archivo vacío con dd, luego se le da formato EXT2 a la imagen usando mkfs, y a lo último se llena la imagen usando populatefs [2] con los archivos que van en la imagen.

[2] https://github.com/oskarirauta/populatefs

2.2.1.4 Linkeo de DeliriOS

Para linkear DeliriOS, previamente tenemos que generar un archivo de defines para NASM y generar la tabla de símbolos de funciones que usaremos para el calltrace [3].

```
# Target para compilar delirios
$(KERNEL_BIN): $(FULL_DIRS) $(NASM_DEFINES) $(OBJS)
	@echo 'Creando tabla de simbolos'
	@echo ''
	$(SYMBOL_SCRIPT) $(SYMBOL_TABLE) $(INCLUDE_DIR) $(TRACED_OBJS)
	@echo ''
	$(CC) $(CFLAGS) -c -o $(SYMBOL_OBJECT) $(SYMBOL_TABLE)
	@echo 'Linkeando kernel...'
	@echo ''
	$(LD) $(LDFLAGS) -o $@ $(OBJS) $(SYMBOL_OBJECT)
```

Los defines de NASM son necesarios para poder compartir los defines de C, por ejemplo posiciones de memoria u otras constantes. Se toma el archivo defines.h y se reemplazan las lineas que comienzan con un # con un %:

```
# Target para convertir los defines de C a NASM
$(NASM_DEFINES): $(C_DEFINES)
	awk '{if ($$0 ~ /^#/) print "%"substr($$0,2);}' $< > $@
```

Por ultimo necesitamos crear la tabla de símbolos. Esta es una tabla donde cada entrada contiene una dirección y un puntero a un string con el nombre de la función en esa dirección.

Creamos un script de **bash** que toma un nombre para la tabla, la carpeta de headers y los archivos objeto cuyos símbolos de funciones queremos incluir, y nos genera un archivo .c listo para ser compilado y linkeado en DeliriOS:

```
#!/bin/bash
table=$1
include=$2
shift
shift

rm -f $table
for header in $include*.h; do
    printf "#include <%s>\n" $(basename $header) >> $table
done

echo '' >> $table
echo 'const struct symbol_entry SYMBOL_TABLE[] = {' >> $table

# Sacamos los simbolos que estan en .text y que no empiezan con _
for object in "$@"; do
    nm $object | grep ' T ' | awk '{print "\t{(uintptr_t) &"$3", \""$3"\"},"}' >> $table
done

echo "};" >> $table
echo 'const uint64_t SYMBOL_COUNT = sizeof(SYMBOL_TABLE) / sizeof(SYMBOL_TABLE[0]);' >> $table
```

Este incluye todos los headers de DeliriOS y crea una entrada en la tabla por cada función buscando los símbolos de .text de cada objeto usando **nm** y **grep** e imprimiéndolos en el archivo usando **awk**.

[3] Y a futuro, para soportar relocations en ELF

2.3 DeliriOS

Se reescribió casi todo DeliriOS de cero. La motivación es simplificar cualquier tarea de extensión futura y reducir fuertemente su *footprint*.

Se unificó el código del BSP (*Bootstrapping processor*) con el de los APs (*Application processors*). Debido a esto, se modificó la inicialización de los APs para que puedan saltar al mismo punto donde empieza a ejecutar el BSP luego de haber pasado a modo protegido. Además, se dejó de compilar el kernel como un binario ELF y en cambio se le definió un encabezado *Multiboot* para que GRUB lo cargue directamente. Esto es, se han eliminado absolutamente todos los módulos, y ahora DeliriOS es un solo kernel unificado.

2.3.1 Multiboot

Multiboot es un estándar que permite a los bootloaders ser capaces de inicializar cualquier tipo de sistema operativo que lo implemente. Principalmente, especifica la necesidad de un encabezado ubicado en alguna parte del comienzo de la imagen del sistema operativo. Este describe las direcciones de memoria de la imagen del sistema operativo, como así también otras características o requerimientos necesarios para la correcta carga del kernel.

Debido a esto, no es necesario compilar el kernel con un header de formato ELF o a.out, ya que el de formato multiboot es suficiente para describir lo que necesitamos.

Para facilitar el acceso por parte del bootloader, ubicamos el encabezado *Multiboot* al comienzo de la imagen de DeliriOS. Esto no es obligatorio ya que el bootloader simplemente busca la magia en la imagen, pero se recomienda que esté lo más cercano al comienzo posible. Para hacer esto simplemente se define una sección personalizada en ensamblador con el header, y se la ubica al inicio del script de linkeo.

Todos los campos del encabezado son de 32 bits y sucesivos sin padding; se comienza primero por los obligatorios:

magic	Número mágico para que el bootloader pueda ubicar el header, tiene que ser 0x1BADB002
flags	Separados en dos: Flags obligatorios que el bootloader debe cumplir (bits 0 - 15) **Bit 0** Requiere que el bootloader cargue los módulos del SO alineados a 4K. **Bit 1** Requiere conocer la cantidad de memoria disponible. Se ubica en la estructura de información de multiboot, con un puntero a la misma en EBX. **Bit 2** Requiere conocer información del modo de video, habilita la sección de video del header. Flags no obligatorios que el bootloader puede ignorar (bits 16 - 31) **Bit 16** Indica si tiene que acatar la sección de memoria del header.
checksum	Cuando se suma este valor a la suma de los flags y la magia, tiene que dar cero

Nosotros activamos el flag 1 para que nos otorgue información adicional y el 16 para poder indicarle dónde cargar las cosas en memoria y dónde saltar. Dicha última sección opcional viene luego. Esta describe posiciones de memoria de la imagen del sistema operativo. Si bien un bootloader no está obligado a seguirlas, GRUB lo hace y por lo tanto nos sirve:

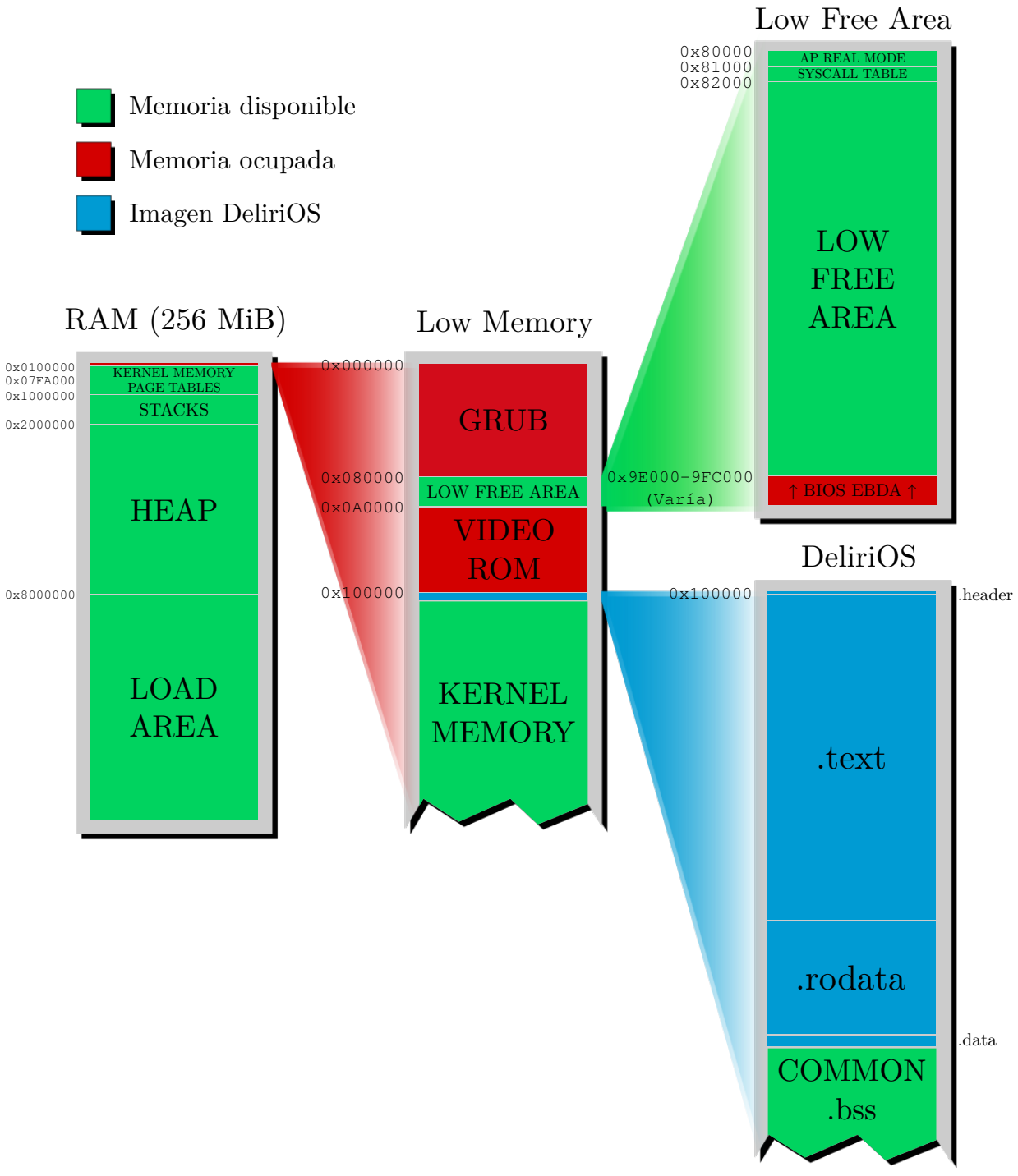
header_addr	La dirección del header, sirve para sincronizar el addressing del kernel con el del bootloader
load_addr	La dirección donde está ubicado el comienzo de los datos a cargar de la imagen
load_end_addr	La dirección donde terminan los datos a cargar de la imagen
bss_end_addr	La dirección donde termina la bss, desde load_end_addr hasta acá se inicializa todo a cero
entry_addr	La dirección donde está el punto de entrada del kernel

Esto nos fuerza a poner la sección bss al final de todo, ya que es la única forma en la que podemos indicarle al bootloader que llene la sección con ceros. Para definir estas direcciones correctamente, creamos símbolos en el script de linkeo correspondientes a las posiciones donde comienza y termina cada sección, que luego son evaluados por el linker y ubicados en el header.

Luego viene la sección de video que no utilizamos y no mostraremos, pero en esta se pueden demandar los modos de video incluyendo gráficos y de texto, así como el tamaño y la profundidad del color siempre que el bit 2 de los flags esté activo.

2.3.2 Mapa de memoria

El mapa de memoria de DeliriOS está definido de la siguiente forma:



Mapa de memoria de DeliriOS

En la referencia, clasificamos la memoria en tres partes, estas son:

2.3.2.1 delirios.bin Como DeliriOS se carga en un solo binario, esta sección es el binario en sí. GRUB lo ubica en memoria exactamente a partir del primer MiB que es lo mínimo que permite.
Dos cosas son importantes:

- El header *Multiboot* debe estar al comienzo para facilitar el acceso por el bootloader.
- La sección COMMON y .bss deben ser inicializadas con ceros, por este motivo deben estar al final para poder especificarlo en el header *Multiboot*. Aunque estas secciones no forman parte del binario en sí, lo son de forma lógica al momento de linkear.

Como DeliriOS no posee protección alguna, los alineamientos y el orden de las otras secciones no son importantes. De todos modos mantuvimos el orden clásico de un binario común y corriente.

2.3.2.2 Memoria Ocupada Estas son las secciones de memoria que no están disponibles, no pertenecientes a DeliriOS luego de que este se cargue en memoria:

Memoria	Tamaño	Descripción
GRUB	**512KiB**	Nuestro bootloader, ocupa la primera mitad del primer MiB de memoria. En realidad, en esta sección de memoria hay también otras cosas importantes como la Bios Data Area a partir del primer KiB, pero en su mayor parte es solo el GRUB.
VIDEO ROM	**384KiB**	En esta sección se encuentran varios dispositivos mapeados a memoria. También varios mapas de video entre las cuales se encuentra la memoria de VGA modo texto a color ubicada en 0xB8000 que usamos en DeliriOS. Hay muchos otros datos read only importantes del BIOS.
BIOS EBDA	**1-8KiB**	Extended Bios Data Area, contiene datos importantes que nos interesan, su tamaño puede variar y siempre está ubicado con su final congruente a la sección de VIDEO y ROM. Generalmente es de 1 KiB pero se han llegado a ver hasta de 8 KiB, por este motivo su ubicación varía y en general se puede encontrar dicha dirección guardada en la BDA. Lo importante es asumir que esta sección no es lo suficientemente grande como para generar problemas y tengamos permitido usar la memoria baja libre sin dañar el GRUB.

El resto de la memoria está disponible para darle un uso en DeliriOS.

2.3.2.3 Memoria Disponible Estas son las secciones de memoria que están libres luego de que el bootloader cargue DeliriOS. Son fácilmente modificables y no deberían considerarse finales en lo absoluto. Sus nombres están dados por el uso que les damos. En la parte alta estas son:

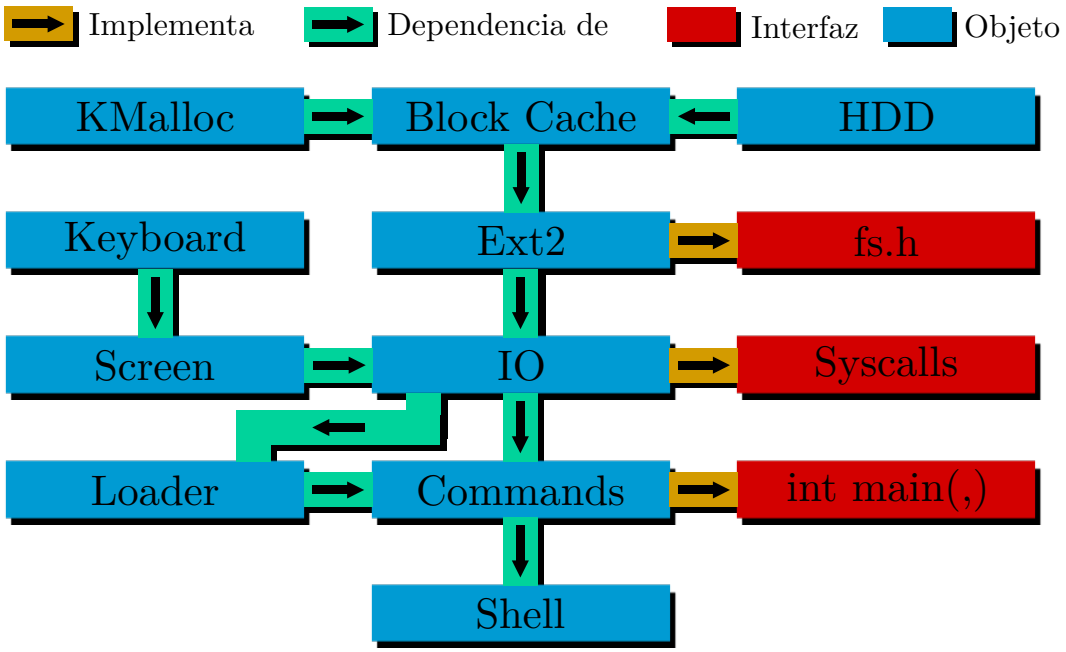
Memoria	Tamaño	Descripción
Kernel Memory	**∼7MiB**	La memoria reservada para la futura expansión de DeliriOS. Actualmente DeliriOS compilado con -O2 no pasa de los 32KiB, y esta sección llega cerca de los 7MiB. De todos modos, también hay que tener en cuenta las secciones que no forman parte del binario, .bss y COMMON, que hacen uso también de este espacio.
Page Tables	**∼8MiB**	Las tablas de página que forman el identity mapping, con 4GiB mapeados. Comienzan en esa posición 0x7FA000 para dar lugar a las tablas de página de mayor nivel a 1, las cuales se ubican consecutivamente hasta la dirección 0x800000 donde comienzan las tablas de nivel 1 que ocupan toda la memoria hasta la dirección 0x1000000 donde comienzan los stacks.
Stacks	**16MiB**	Los stacks de los procesadores, 16 stacks consecutivos de 1MiB cada uno. Esto nos deja un soporte actual de 16 procesadores.
Heap	**96MiB**	El heap de DeliriOS usado por kmalloc para reservar memoria.
Load Area	**128MiB**	Donde el loader de DeliriOS carga los binarios a correr. Para esto es importante que el binario esté linkeado dentro de esta área para evitar pisar datos de alguna de las otras áreas. Además, las aplicaciones pueden considerar la totalidad de la memoria en este área como exclusivamente propia.

En la parte baja son:

Memoria	Tamaño	Descripción
Low Memory Area	**∼120KiB**	Memoria baja no usada por el GRUB, al final de la misma se encuentra la EBDA. Como dijimos antes, no se han visto EBDA más grandes que 8KiB, por lo tanto podemos asumir 120KiB libres. En esta misma reservamos 2 páginas: **AP Real Mode** El código de los AP copiado a memoria baja para saltar a modo protegido. **Syscall Table** La tabla de syscalls copiada a memoria baja temporalmente para mantenerla en una dirección fija para poder ser usada por las aplicaciones, a futuro no va a existir.

2.4 Dependencias y funcionamiento

La idea básica de las dependencias entre las distintas partes de DeliriOS es la siguiente:



Dependencias tentativas en DeliriOS

El funcionamiento, de abajo hacia arriba:

Shell Un shell con una lista determinada de comandos, es la interfaz que expone delirios al usuario.

Commands Utilizan las syscalls provistas por IO para realizar alguna acción determinada, a excepción del Loader, que aún no es una syscall, utilizado por el comando *exec*. El shell los ejecuta de forma genérica gracias a que todos implementan la misma aridad que un main standard, pasándole los parámetros que introdujo el usuario.

Loader Utiliza las syscalls provistas por IO para leer los headers ELF y cargar el binario en memoria.

IO Utiliza la interfaz agnóstica de filesystem provista por la implementación de Ext2 para ofrecer syscalls varias de lectura y escritura de archivos con file descriptors. Además, utiliza Screen para proveer los descriptores *stdin*, *stdout* y *stderr*.

Screen Ofrece acceso de escritura a la pantalla y de lectura del teclado, este último al ser leido es impreso en pantalla.

Keyboard Ofrece acceso a un buffer FIFO cíclico de teclas presionadas, se le puede pedir la próxima tecla presionada.

Ext2 Ofrece acceso de lectura y escritura [4] para Ext2 utilizando la caché de bloques como acceso al disco. Implementa la interfaz mandatoria de `fs.h` para generalizar los filesystems.

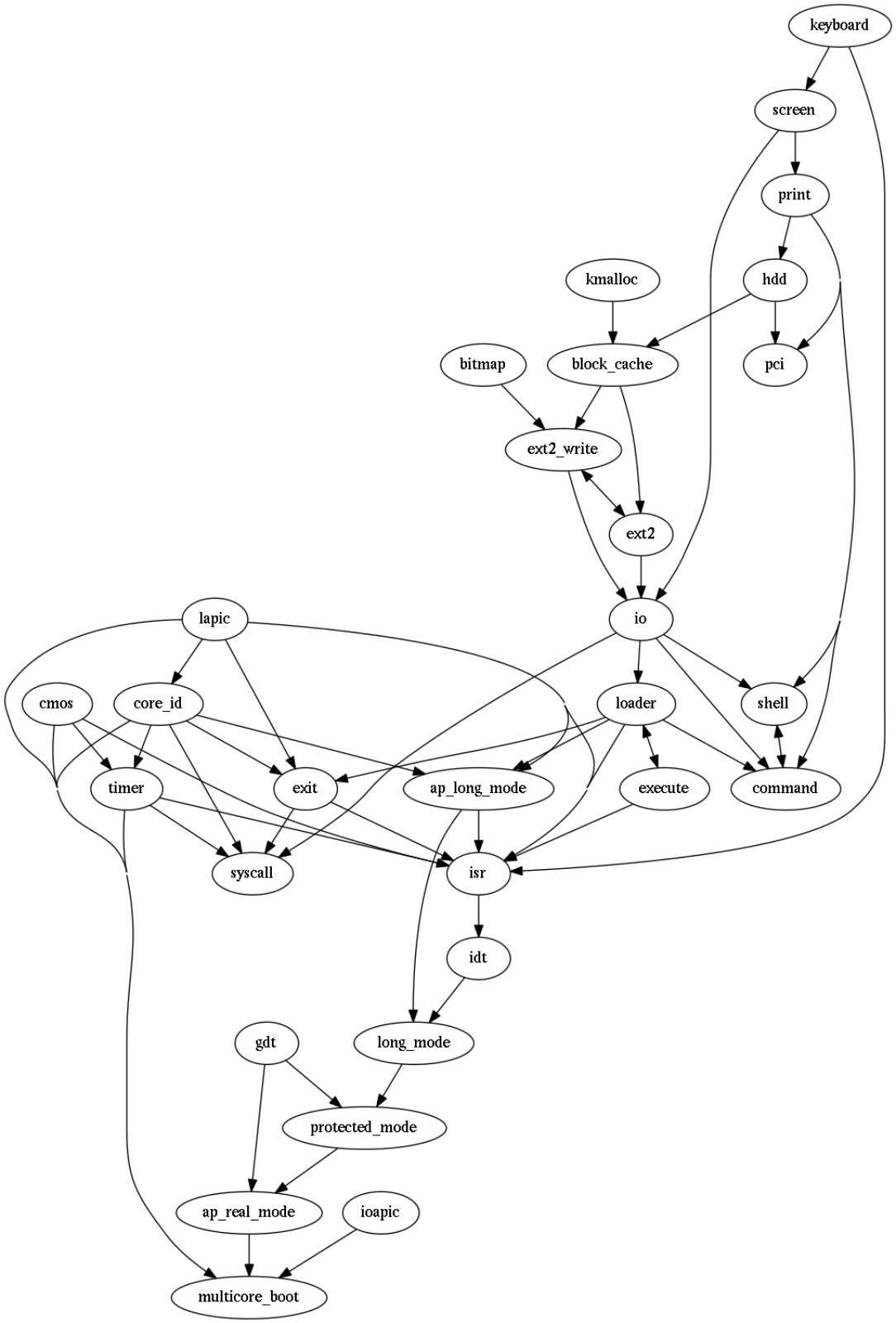
Block Cache Ofrece acceso en memoria a la información del disco utilizando el driver de disco para este fin. Es capaz de crear una caché con un tamaño de bloque predeterminado en sectores como el filesystem prefiera. Utiliza kmalloc para este fin.

Kmalloc Ofrece acceso a memoria dinámica, cabe destacar que debido a los objetivos de DeliriOS, su uso debe ser reducido. Por esto solo es utilizado por la caché de bloques.

HDD Ofrece acceso de lectura y escritura de sectores del disco.

Más adelante se detallarán cada una de estas partes, incluyendo las interfaces. Además se creó un script para parsear las dependencias entre archivos objeto que nos otorga el linker cuando le pasamos el parámetro *–cref*. El output de dicho script es un grafo como el siguiente:

[4] La interfaz de syscalls de lectura y escritura, junto con el driver de Ext2 de lectura fueron implementados en este trabajo. Pero la implementación de escritura de Ext2 fue implementada por Gonzalo Ciruelos en el suyo.



Mapa de dependencias de DeliriOS

Se han removido los objetos kernel, exception y utils; ya que el primero se encarga de inicializar varios componentes y los otros dos son utilizados por varios objetos, lo cual complica la visualización. Se puede notar que las dependencias mostradas anteriormente coinciden tentativamente con las dependencias que muestra el grafo.

2.5 Booteo

El proceso de booteo se ha modificado fuertemente con el fin de simplificarlo. Ahora que el kernel está solo compuesto de un binario, es mucho más sencillo. Para facilitar la comprensión, explicaremos lo que sucede en los tres modos:

2.5.1 Modo Real

Luego de copiar el binario en memoria, GRUB nos deja el BSP en el punto de entrada en modo protegido. Por lo tanto, el Modo Real del BSP ocurre dentro del GRUB. Entre otras cosas GRUB setea la línea A20 por nosotros.

Los APs, en cambio, los levantamos nosotros. Como comienzan en Modo Real es necesario que se despierten en alguna página por debajo del primer MiB de memoria [5]. Es por esto que tenemos que hacer uso de la memoria baja que queda libre luego de que se cargue DeliriOS, que es donde se copia el código en modo real de los APs.

Como DeliriOS es ahora un binario unificado, aprovechamos para cargar la GDT verdadera en el modo real de los APs. El código anterior en cuestión:

```
global _start
BITS 16
section .text
_start:
    jmp mr_ap_start
; data_area
align 4

ap_full_code: dd 0xABBAABBA ; puntero al inicio del codigo de modulo estilo sueco

gdt: dq 0x0
code_s32: dd 0x0000FFFF
          dd 0x00CF9800
data_s:   dd 0x0000FFFF
          dd 0x00CF9200
gdt_desc: dw $ - gdt
          dd gdt

krnPML4T: dd 0x740000 ; segun defines.h

mr_ap_start:
    cli
    ; A20 YA ESTA HABILITADO POR EL BSP
    ; cargar la GDT;
    lgdt [gdt_desc]

    ; setear el bit PE del registro CR0
    mov eax, CR0;levanto registro CR0 para pasar a modo protegido
    or eax, 1;hago un or con una mascara de 0...1 para setear el bit de modo protegido
    mov CR0, eax

    ; pasar a modo protegido
    jmp (1<<3):f_mp_ap_start; saltamos a modo protegido, modificamos el cs con un jump
    ;{index:1 | gdt/ldt: 0 | rpl: 00} => 1000
    ;aca setie el selector de segmento cs al segmento de codigo del kernel

BITS 32
f_mp_ap_start:
    mov ax, 2<<3
    mov ds, ax

    ;apuntar cr3 al PML4
    mov eax, [krnPML4T]
    mov cr3, eax
    jmp [ap_full_code]
```

[5] La verdadera restricción son los 8 bits del APIC para seleccionar la página donde inician, si no, podríamos abusar de la A20

Donde la GDT era una gdt temporal, krnPML4T estaba hardcodeado, y ap_full_code era seteado por un loader previo de DeliriOS.

Se lo cambió por:

```
%include "defines.mac"

global ap_real_mode
global ap_real_mode_end

extern common_code
extern GDT_DESC

BITS 16

section .text

ap_real_mode:
    ; A20 YA ESTA HABILITADO POR EL BSP
    ; Desactivo interrupciones
    cli
    ; Cargar la GDT, levanto el segmento mas cercano al primer MiB
    mov ax, 0xFFFF
    mov ds, ax
    ; Como estamos 16 bytes abajo del primer MiB, le sumamos 16
    mov ebx, GDT_DESC + 16
    ; Tomo solo el offset contando desde el primer MiB
    lgdt [ds:bx]
    ; Setear el bit PE del registro CR0
    mov eax, CR0
    or eax, 1
    mov CR0, eax
    ; Cargo los selectores de segmento
    mov ax, GDT_DESC_DATA_DPL0
    mov ds, ax
    mov es, ax
    mov fs, ax
    mov gs, ax
    mov ss, ax
    ; Saltar a modo protegido
    jmp dword GDT_DESC_CODE_DPL0_32:common_code
ap_real_mode_end:
```

Se diferencia en que:

- No hay datos hardcodeados, la dirección de la GDT verdadera y del punto de entrada de los APs a DeliriOS se le pide al linker. Los valores para los selectores de segmento son tomados de defines.mac, que es un archivo generado a partir de `defines.h` en tiempo de compilación.
- Se carga la GDT verdadera de DeliriOS. En modo real podemos direccionar 64KiB - 16B por encima del primer MiB usando el ultimo segmento 0xFFFF. Abusamos de esto moviendo la dirección en 32 bits + 16 en ebx, los cual nos deja el offset en bx que usamos para cargarla. Recordar que la A20 ya fue levantada por el GRUB.
- Se setean todos los registros de segmento luego de activar modo protegido
- Se salta directamente al código común de modo protegido que ejecutan tanto el BSP como los APs
- La etiqueta ap_real_mode_end nos sirve para saber dónde termina el código. Lo cual es necesario saber para poder copiarlo en la parte baja.

Luego cada AP salta a un punto intermedio de la inicialización de Modo Protegido por parte del BSP, que ocurre luego de que este haya completado las tablas de página.

2.5.2 Modo Protegido

Aquí comienza el BSP luego de su inicialización por GRUB. Todo el código se rehizo de cero para reducir su tamaño, aunque la funcionalidad es similar.

Se carga la GDT, se inicializan los registros de segmento, se limpia la pantalla y se guarda en *ebp* el formato y la posición del cursor en la pantalla, para poder escribir mensajes a medida que se bootea. Se utiliza esta nueva función auxiliar para imprimir en pantalla, en vez de los macros:

```
; Imprime texto en pantalla
; esi: Puntero al texto
; ebp: col|row|format
; clobbers: eax, ecx, dx
print_text:
    xor eax, eax
    mov ecx, ebp
    ; Formato
    mov dx, cx
    shr ecx, 16
    mov al, VIDEO_WIDTH * 2
    ; Fila
    mul cl
    add eax, VIDEO_ADDRESS
    ; Columna
    shr cx, 8
    .loop:
        mov dl, [esi]
        cmp dl, 0
        jz .end
        add esi, 1
        cmp dl, 10
        jz .new_line
        mov [eax + ecx * 2], dx
        add cx, 1
        cmp cx, VIDEO_WIDTH
        jnz .loop
    .new_line:
        ; Fila + 1
        add ebp, 0x10000
        add eax, VIDEO_WIDTH * 2
        xor ecx, ecx
        jmp .loop
    .end:
    ; Restauramos la columna
    and ebp, 0xFFFFFF
    shl ecx, 24
    or ebp, ecx
    ret
```

Basicamente utiliza *ebp* como registro para preservar el estado de la pantalla, esto es la posición del cursor y el formato en 16 bits. Luego de una llamada los datos de ebp se mantienen actualizados acorde. Es capaz también de detectar un salto de linea, ya sea por el caracter 10 ASCII o porque se llegó al final de la pantalla.

Siguiendo con lo anterior, luego se setea la pila (que sabemos que es la primera del arreglo de pilas), se chequea que tanto CPUID como el Modo Largo estén disponibles, y se procede a crear las tablas de página para hacer identity mapping de 4GiB.

Las primeras tres cosas no cambiaron con respecto a la anterior iteración de DeliriOS, pero para paginación se creó una función auxiliar con otros parámetros custom:

```
; Llena una tabla de pagina de cualquier nivel
; Deja los registros bien posicionados para sucesivas llamadas
; edx:eax: La pagina a apuntar por la primera entrada con atributos
; ecx: La cantidad de entradas a llenar
; edi: La tabla a llenar
; clobbers: esi
fill_page_table:
    lea esi, [ecx * 8]
    add esi, edi
    .loop:
        mov [edi], eax
        mov [edi + 4], edx
        ; Incremento 4KiB
        add eax, PAGE_SIZE
        ; Una entrada son 8 bytes
        add edi, 8
        cmp edi, esi
        jnz .loop
    ; Llenamos el resto de la tabla con ceros
    .loop_zero:
        test edi, PAGE_SIZE - 1
        jz .end
        mov dword [edi], 0
        add edi, 4
        jmp .loop_zero
    .end:
    ret
```

Esto nos asegura que se escriben *ecx* entradas en la/s tabla/s apuntada por *edi* utilizando *edx:eax* como valor de la primera entrada de la tabla. El resto de las entradas se llenan con 0s, al finalizar la función *ecx* no cambia, *edi* permanece apuntando a la siguiente página sin rellenar (Lo cual es conveniente, por que es donde estará ubicada la siguiente tabla que tenemos que llenar), y edx:eax apuntando a la próxima tabla sin asignar manteniendo los atributos [6].

[6] Notar que no se incrementa edx ya que estamos por debajo de los 4GiB.

Es por esto que luego para crear el identity mapping, solo necesitamos hacer lo siguiente:

```
; Estoy usando paginacion IA-32e => bits CR0.PG=1 y CR4.PAE=1 y EFER.LME=1
; Mapeo 4GiB con paginas de 4KiB

; Limpiamos edx lo usamos como la parte alta de las entradas
xor edx, edx

; Creamos solo una entrada en la PML4T que apunta a la PDPT
mov edi, KERNEL_PML4T
; La PDPT esta 1 pagina despues y le ponemos los atributos
lea eax, [edi + PAGE_SIZE + PAGE_ATTRIBUTES]
mov ecx, 1
call fill_page_table

; Creamos solo 4 entradas en la PDPT que apuntan a PDTs
; Cada entrada mapea un GiB de memoria, en total 4GiB
shl ecx, 2
call fill_page_table

; Crear 4 * 512 entradas en las PDTs que apuntan a PTTs
; Se mapean 4 Page Directories completos
shl ecx, 9
call fill_page_table

; Crear 4 * 512 * 512 entradas en PT que apuntan a paginas de 4KiB
; Se mapean 4 * 512 Page Tables completas haciendo identity mapping
and eax, PAGE_ATTRIBUTES
shl ecx, 9
call fill_page_table
```

Seteamos *edx* en 0 ya que todas las entradas tendrán limpia la parte alta al estar debajo de los 4GiB de memoria. Apuntamos en *edi* a la posición de la primera tabla, que es la de nivel 4. Luego creamos el valor de la primera entrada en *eax*, apuntando a la tabla de nivel 3 que estará ubicada en la siguiente página, y añadimos los atributos (Present, Read/Write, System).

Luego se llama a la función con distintos valores de *ecx*, siendo estos 1, 4, 4 * 512 y 4 * 512 * 512 cantidad de entradas. Debido a la naturaleza del identity mapping se pueden generar shifteando el registro hacia la izquierda 2, 9 y 9 veces luego de haber sido seteado en 1 inicialmente. Los primeros dos niveles de las tablas no quedan completas, es por esto que es necesario hacer más de un llamado. Los dos niveles más bajos no se pueden unificar tampoco porque difieren en que las tablas de nivel mas bajo deben apuntar desde la dirección 0, por este motivo limpiamos *eax* exceptuando los atributos en el último llamado.

Una vez hecho esto comienza el código que comparte con los APs, o sea, donde saltan luego de Modo Real:

```
; ======== CODIGO COMUN ========
; Aca arrancan los AP y el BSP sigue ejecutando
common_code:
    ; Apunto cr3 al PML4
    mov eax, KERNEL_PML4T
    mov cr3, eax
    ; Prender el bit 5 (Sexto bit) para activar PAE
    mov eax, cr4
    or eax, 1 << 5
    mov cr4, eax
    ; Activamos modo largo
    mov ecx, 0xC0000080 ; Seleccionamos EFER MSR poniendo 0xC0000080 en ECX
    rdmsr               ; Leemos el registro en EDX:EAX.
    or eax, 1 << 8      ; Seteamos el bit de modo largo.
    wrmsr               ; Escribimos nuevamente al registro.
    ; Activamos paginacion
    mov eax, cr0        ; Obtenemos el registro de control 0 actual.
    or eax, 1 << 31     ; Habilitamos el bit de Paginacion.
    mov cr0, eax        ; Escribimos el nuevo valor sobre el registro de control
    ; Estamos en modo ia32e compatibilidad con 32 bits
    ; Ahora saltamos a modo largo, que tiene otra entrada en la GDT
    jmp GDT_DESC_CODE_DPL0_64:long_mode
```

Aquí se setea el CR3 correspondiente, se activa PAE y el Modo Largo, y subsecuentemente se realiza el salto a Modo Largo cambiando el descriptor de código por el de 64 bits.

2.5.3 Modo Largo

En el Modo Largo comienzan tanto los APs como el BSP luego de saltar desde Modo Protegido. Se limpian los registros, se carga la IDT (aún no inicializada) y se activa SSE.

Luego, el BSP debe encargarse de inicializar el Kernel y tiene un flujo y destino distinto al de los APs. Por este motivo, necesitamos una forma de detectar si somos un AP para generar un desglose. Como aún no está inicializado nada con respecto a Multicore, necesitamos una alternativa. El noveno bit del registro MSR 0x1B indica si somos BSP. Podemos entonces crear un macro de NASM fácilmente para saltar a otro lugar si no somos BSP:

```
%macro jump_if_not_bsp 1
    mov ecx, 0x1B
    rdmsr
    test eax, 1 << 8
    jz %1
%endmacro
```

A partir de acá, tenemos dos flujos, el del BSP y el de los APs.

2.5.3.1 Flujo del BSP En el caso en que seamos BSP, se procede a setear la pila de nuevo pero en 64 bits. Para el BSP es la primera del array, la cargamos en el RSP, pusheamos RBP que es 0 dos veces para alinearla, y le asignamos al RBP el RSP para completar el stack frame. Una vez hecho esto saltamos a la función kernel_main en código C, la cual no retorna y se encarga de inicializar lo más grueso del Kernel.

Quitando de lado el escribir en pantalla, en kernel_main se hace en orden (Una por función):

1. Inicializar la IDT, necesario para las interrupciones.
2. Inicializar todo lo relacionado con Multicore, incluyendo el APIC y el IOAPIC.
3. Inicializar el heap, necesario para kmalloc.
4. Inicializar el disco, necesario para poder leer y escribir en él.
5. Inicializar IO, esto incluye el filesystem, en este caso EXT2.
6. Inicializar el shell, el cual permanece en un input loop y no retorna.

El orden es casi mandatorio a excepción del heap y el disco que no dependen entre sí[7].

2.5.3.1.1 Inicializar IDT Inicializamos las 21 excepciones reservadas del CPU, y además:

Indice IDT	Nombre	Descripción
32	Clock	Solía ser la rutina del PIT. Actualmente en desuso, usamos el RTC
33	Keyboard	Interrupción de teclado
34	RTC	Real Time Clock. Lo usamos en su modo periódico para poder implementar Sleep.
44	Exit	Cuando se deja de ejecutar un binario, se realiza un broadcast de esta interrupción. Se encarga de hacer volver a todos los procesadores al Kernel correctamente.
255	Spurious	Para las interrupciones espurias, obligatorio al inicializar IOAPIC. El índice necesariamente tiene que tener los 4 bits menos significativos en 1. Lo más simple es usar 255 (0xFF).

[7] Referirse al grafo de dependencias provisto anteriormente

2.5.3.1.2 Inicializar Multicore Se refactorizó el código con respecto al trabajo original de DeliriOS pero la lógica no cambió mucho salvo algunos añadidos y bugfixes. Se separaron las funciones del LAPIC en un archivo aparte.

Se hace en orden:

1. Se parsea la estructura MPFS para obtener la configuración multicore.
2. Se enmascara el PIC legacy para que no moleste ya que vamos a usar APIC.
3. Se prende el Local APIC del BSP. O sea, se prende el Local APIC del procesador actual, ya que es el BSP el que está corriendo el código. La función es la misma para todos los procesadores, se le indica el índice de la IDT de las interrupciones espurias en los 8 bits menos significativos sumado al noveno bit en 1 que lo enciende.
4. Se arma un array con los LAPIC IDs de los procesadores obtenidos al parsear la estructura MPFS. Empezando por el BSP en el índice 0 y siguiendo por los APs. Necesario para poder diferenciarlos entre ellos y asignarle las pilas a los APs, ya que el LAPIC ID no es obligatoriamente una sucesión de números naturales sucesivos partiendo del 0. En otras palabras, se les da un orden a los procesadores dado sus Lapic ID, accesible por la syscall **core_id()**.
5. Se inicializa el IOAPIC mapeado en la dirección que obtuvimos al parsear la estructura MPFS. Es el equivalente más avanzado a inicializar el viejo PIC. Entre otras cosas se le indica los índices de la IDT para el Teclado y el RTC atendidas solo por el BSP.
6. Inicializamos el RTC mediante el CMOS, necesario para poder utilizar la syscall **sleep()**, la cual necesitamos para encender los APs. Se le activa el modo periódico donde cada interrupción dista un poco menos que un milisegundo respecto a la última. Las interrupciones tienen que estar desactivadas para poder hacerlo, de lo contrario en un peor caso podría quedar en un estado inválido si transcurriese mucho tiempo entre las escrituras del CMOS.
7. Activamos interrupciones, también obviamente necesario para **sleep**, aunque debía suceder en algún momento luego de inicializar el RTC y previo a utilizar interrupciones.
8. Encendemos los APs. Lo único que cambió es la dirección donde comienzan, la cual es a partir del medio MiB de memoria justo luego del GRUB. En esa posición se copia el código de modo real de los APs y luego se los despierta, utilizando **sleep()** para esperar entre los IPIs.

Una vez terminado esto, los APs ya estarían esperando en su posición del Kernel en modo Largo a que un binario se ejecute. Se puede proceder a inicializar el resto del Kernel.

2.5.3.1.3 Inicializar Heap La inicialización del heap es necesaria para poder usar kmalloc. Lo creamos en la posición indicada en el mapa de memoria anteriormente, de 0x2000000 a 0x8000000. Actualmente está solo siendo utilizado por la caché de bloques necesaria para la implementación del filesystem. Referirse a la sección Heap para más detalles.

2.5.3.1.4 Inicializar Disco La inicialización del disco fue tomada de JuampiOS y, como todo el código, fue pulido y adaptado al code style de DeliriOS. El código tenía una inicialización no estándar, lo cual provocaba que DeliriOS no pudiese ser corrido en QEMU o VirtualBox. Actualmente Gonzalo Ciruelos lo cambió a una inicialización estándar usando IDENTIFY, la cual es bastante simple y devuelve un sector de datos lleno de información del disco como indica el estándar ATA.

Lo único importante acá es ser capaz de leer y escribir sectores, utilizando las funciones hdd_read y hdd_write.

2.5.3.1.5 Inicializar IO IO en este contexto implica básicamente la interfaz de acceso a file descriptors y las syscalls que estos conllevan. En este caso tenemos que levantar el filesystem que esté asignado al kernel, en nuestro caso EXT2, y reservar los primeros 3 file descriptors (STDIN, STDOUT, STDERR) para acceder a la pantalla.

Para más detalles remitirse a la sección IO.

2.5.3.1.6 Inicializar Shell El shell es simplemente un input loop, la función de inicialización no retorna. Para más detalles ver la sección Shell.

2.5.3.2 Flujo de los APs Los APs aún no tienen seteada su pila. Para diferenciarlos entre ellos, necesitamos el *Lapic ID*. Como el *Lapic ID* para cada procesador puede ser modificado por el BIOS, y solo está garantizado que son números distintos incrementales, utilizamos el arreglo mencionado anteriormente para obtener el ID correspondiente a cada AP y lo usamos como índice en el arreglo de pilas. Como no se cuenta con una pila aún, esta parte se hace toda *inline* en ensamblador evitando calls y pushes.

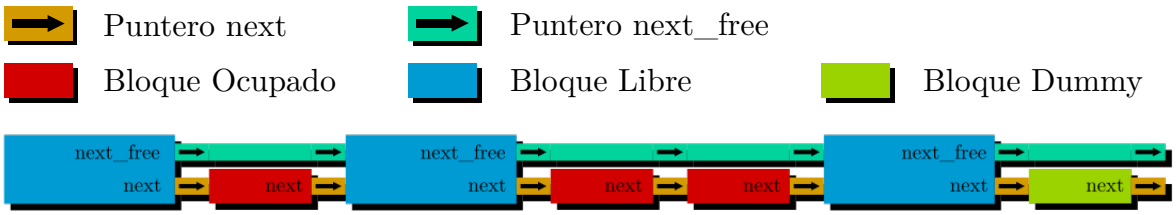
Una vez setada la pila, podemos hacer uso de las funciones de DeliriOS. Prendemos el Local APIC del AP y activamos interrupciones.

3 Heap

El heap es necesario para poder usar kmalloc/kfree. Actualmente solo es usado por la caché de bloques.

3.1 Estructura

Tiene una estructura similar a una estructura de malloc clásica con la diferencia de que los bloques libres también mantienen una lista circular entre ellos además de la que mantienen todos los bloques. Se puede apreciar en la siguiente imagen:



Heap de DeliriOS

Cada bloque ocupado es de al menos 16 bytes que es el tamaño mínimo que puede tener un bloque libre con sus dos punteros. Esto es para que al liberar haya espacio suficiente para transformarlo en uno libre, por lo tanto el alloc mínimo es de 8 bytes. El bloque dummy está ubicado para mantener el siguiente invariante: que *next* de un bloque libre apunta a uno ocupado, este solo ocupa 8 bytes ya que nunca será liberado y apunta al primer bloque del heap.

3.2 Búsqueda

Para ocupar un bloque libre simplemente tenemos que iterar sobre la lista circular de bloques libres. Se usa el criterio *best fit* para encontrar bloques libres al reservar memoria, o dicho de otra forma, se busca el bloque libre donde sobre menos espacio. Para conocer el tamaño de un bloque libre se calcula la diferencia entre la posición del bloque y el siguiente.

Para liberar un bloque ocupado simplemente iteramos la lista de bloques libres hasta que nos pasemos de la dirección que queremos buscar y luego iteramos los ocupados a partir del anterior bloque libre. Una vez que encontramos el bloque ocupado tenemos cinco casos:

Caso 1 El bloque está entre bloques libres, en ese caso nos queda un solo bloque libre que es la fusión de los 3. En ese caso el bloque libre anterior tiene que apuntar a lo mismo que apuntaba el bloque libre posterior.

Caso 2 El bloque tiene un bloque libre solo atrás, tenemos que hacer que dicho bloque libre apunte a nuestro siguiente bloque ocupado.

Caso 3 El bloque tiene un bloque libre solo adelante, entonces este bloque tiene que apuntar a lo mismo que apuntaba dicho bloque libre. Además el anterior libre debe apuntar a nosotros.

Caso 4 El bloque no tiene bloque libres en los extremos, en este caso tenemos que crear el puntero que apunta al siguiente libre. Además el anterior libre debe apuntar a nosotros.

Caso 5 El heap estaba lleno, en este caso no tenemos bloques libres así que tenemos que crear un bloque libre que se apunte a sí mismo.

3.3 Inicialización

Para inicializarlo, simplemente se crea una estructura para identificar al heap, indicando dónde comienza, dónde termina y el bloque libre como punto de entrada, el cual será usado en kmalloc y kfree. Dicha estructura se ubica en el comienzo del heap dado por los parámetros que se otorguen. Luego se crea un bloque libre del tamaño del heap entero que apunte a sí mismo en la lista de bloques libres y al bloque dummy en la lista común, el cual se crea en el final apuntando a este bloque libre que acabamos de crear. Este bloque no es reconocido por el heap como un bloque válido ya que no debe ser liberado. Está ubicado justo en el final dentro de los parámetros otorgados.

4 IO

Con IO nos referimos al manejo de archivos, esto es, las syscalls clásicas POSIX-like para interactuar con archivos. Con estas funciones se puede acceder a la pantalla y al filesystem implementado.

4.1 Filesystem

En el caso de interactuar con el filesystem, la implementación del mismo es agnóstica al kernel, se creó una interfaz de funciones que todo filesystem debe implementar para poder ser utilizado en DeliriOS. Las funciones están definidas en el header `fs.h` y hasta ahora son:

```
// Monta el filesystem
int64_t fs_mount(fs_info* info);
// Desmonta el filesystem
void fs_unmount(fs_info* info);
// Devuelve el directorio raíz del filesystem en dir
fs_dir* fs_root(fs_info* info, fs_dir* dir);
// Dado un directorio src abre el archivo apuntado por él en dest
fs_file* fs_open(fs_info* info, fs_dir* src, fs_file* dest);
// Cierra el archivo file
int64_t fs_close(fs_info* info, fs_file* file);
// Lee de un archivo "size" bytes y los pone en "buf"
uint64_t fs_read(fs_info* info, fs_file* file, void* buf, uint64_t size);
// Escribe en un archivo "size" bytes desde "buf"
uint64_t fs_write(fs_info* info, fs_file* file, const void* buf, uint64_t size);
// Avanza offset en el archivo, puede retroceder también
int64_t fs_seek(fs_info* info, fs_file* file, int64_t offset);
// Indica dónde está parado el archivo
int64_t fs_tell(fs_info* info, fs_file* file);
// Rellena stat con información del archivo
int64_t fs_fstat(fs_info* info, fs_file* file, fs_stat* stat);
// Dado un directorio src abre el directorio apuntado por él en dest
// Si src y dest son el mismo se cierra primero y se abre en su lugar
fs_dir* fs_opendir(fs_info* info, fs_dir* src, fs_dir* dest);
// Cierra el directorio
int64_t fs_closedir(fs_info* info, fs_dir* dir);
// Lee la próxima entrada en el directorio y la pone en su dir_entry
dir_entry* fs_readdir(fs_info* info, fs_dir* dir);
// Retrocede el directorio al comienzo como luego de un fs_opendir
fs_dir* fs_rewinddir(fs_info* info, fs_dir* dir);
// Crea un nuevo archivo
int64_t fs_create(fs_info* info, fs_dir* dir, dir_entry* file_info);
// Trunca un archivo al tamaño length
int64_t fs_truncate(fs_info* info, fs_file* file, uint64_t length);
// Borra el archivo apuntado por dir
int64_t fs_remove(fs_info* info, fs_dir* dir);
```

La explicación de cada una está dada con un comentario arriba. Cualquier filesystem que implemente correctamente estas funciones es automáticamente compatible con DeliriOS.

Por otra parte, el filesystem también se tiene que encargar de definir los siguientes 3 structs:

fs_info Contiene campos con datos que el filesystem debe guardar para su correcto funcionamiento luego de ser montado. Por ejemplo, punteros a caché de bloques, el tamaño de los bloques, o algún otro dato adicional que requiera.

fs_file Contiene campos con datos que un archivo debe guardar para su correcto funcionamiento luego de ser abierto. Por ejemplo, punteros a bloques, el offset donde se está leyendo/escribiendo, etc.

fs_dir Contiene campos con datos que un directorio debe guardar para su correcto funcionamiento luego de ser abierto. Por ejemplo, si los directorios también son archivos, un fs_file podría ser incluido, el nombre de la entrada actual del directorio, etc.

Como se puede notar, se diferencian directorios de archivos ya que podrían diferir en un filesystem, y mejoran la abstracción en el sistema operativo.

Un lector atento se dará cuenta de que esto limita la posibilidad de tener múltiples filesystems de distinto tipo, ya que no puede haber múltiples definiciones del mismo struct. Debido a la naturaleza de DeliriOS, se tomó esta decisión de diseño ya que cambiar el filesystem y recompilar está ubicado en el orden de los segundos. Y además, para el rango de usos que se le puede dar a este sistema operativo, es el mejor compromiso entre simplicidad y flexibilidad.

Sin embargo esto no quita que se pueda soportar montar múltiples filesystems **del mismo tipo**, que si bien actualmente no está soportado, no debería ser difícil añadirlo.

Dicho esto, para inicializar el filesystem, basta con montarlo ofreciéndole un struct fs_info para que guarde sus datos, el cual por ahora guardamos estáticamente dentro de `io.c`. Dentro de esta función el filesystem hará uso de la caché de bloques, la cual tiene acceso al disco, y tratará de parsear su estructura.

4.2 File y Dir descriptors

Cada file descriptor está compuesto por un par fs_file y fs_stat.

Este último lleva la cuenta del estado del archivo correspondiente. Los campos son similares al stat de POSIX, entre ellos se encuentra el tipo. En nuestro caso si el tipo es 0 se considera libre, y si es 255 está reservado. Con 255 asignamos al tipo de STDIN, STDOUT y STDERR para que no sean alterados.

Cada dir descriptor está compuesto por un par fs_dir y un booleano que indica si está libre o no.

Dado ambos, se tiene un arreglo de file descriptors y dir descriptors estático que se utiliza para llevar la cuenta de ambos tipos de descriptores. Se utiliza el índice de este arreglo como interfaz en las syscalls, al igual que POSIX.

4.3 Syscalls

Las syscalls implementadas con respecto a IO son las siguientes:

```
/* SYSCALLS */
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 type);
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);
```

Son bastante auto explicativas. Todas hacen uso de código estático dentro de io.c para manejar paths. A la vez se lleva la cuenta del *current working directory* en un dir descriptor aparte para diferenciar entre paths relativos y paths absolutos y manejarlos correctamente.

5 Block Cache

La caché de bloques es la encargada de cargar datos del disco en memoria. Tiene el concepto de bloques abstraído para que un filesystem sea capaz de cargar bloques de un tamaño determinado en vez de sectores de disco.

5.1 Funcionamiento

Al crear una caché de bloques se puede pedir el tamaño de bloque deseado y la cantidad de entradas inicial de la caché.

Lo primero puede ser cualquier tamaño positivo, que será redondeado hacia arriba en tamaño de sectores. Esto es debido a que es lo mínimo indizable con lo que podemos leer y escribir utilizando hdd_read y hdd_write respectivamente.

Lo segundo se refiere al tamaño inicial del vector que almacenará las entradas de caché. Esto devolverá un puntero a la estructura de la caché que puede ser reusado para hacer operaciones sobre la caché.

Una de estas operaciones es cargar un bloque en memoria dado su índice en el disco, del cual se devuelve un puntero a él. Luego si se quisiera liberar dicho bloque solo se necesita pedir liberar un puntero que esté dentro del rango del bloque. Esto es útil si solo se desea guardar un puntero en el medio del bloque, por ejemplo una tabla.

Además cada bloque puede ser pedido múltiples veces, en ese caso la caché devolverá siempre el mismo y llevará la cuenta de cuántas referencias tiene ese bloque.

También se puede marcar como sucio un bloque, para poder ser escrito a disco al liberarse.

5.2 Implementación

Al crear la caché se crea un struct con los siguientes datos:

- Puntero al vector que contiene las entradas de la caché
- Cantidad de entradas del vector
- Tamaño de un bloque en sectores
- Tamaño de un bloque en bytes

Lo primero es una tabla con entradas que describen bloques. Cada entrada de la caché posee:

- Un puntero al bloque cargado en memoria con malloc
- El número de dicho bloque, o sea si consideramos al disco como un arreglo de bloques, su índice.
- Flags de la entrada, por ahora utilizado para asignar si está sucio o no el bloque para saber si es necesario escribirlo a disco al liberarlo.
- Cantidad de referencias, debido a que un bloque puede ser cargado más de una vez. Se considera libre el bloque si esto es 0.

Se inicializa todo a 0 en dicha tabla y se devuelve un puntero al mismo struct, que puede ser usado para hacer allocs en la caché.

Al hacer un alloc de un bloque primero se busca que no esté actualmente en la tabla, en cuyo caso se aumenta la cantidad de referencias y se devuelve dicho bloque. Caso contrario se busca una entrada libre y se hace un malloc de ser necesario (Podría el malloc estar hecho si había otro bloque antes) y se carga el bloque en esa posición de memoria dejando la cantidad de referencias en 1.

Para marcar como sucio un bloque, se lo busca y se le setea el flag de sucio.

Luego al liberar un bloque, se busca en la tabla si alguno de ellos contiene el puntero peticionado. En caso afirmativo se reduce en uno la cantidad de referencias. Si esta cantidad llega a 0 no se libera el bloque, pero se considera libre si es necesario usar la entrada. También se escribe en disco en caso de estar sucio. Si se llegara a pedir dicho bloque nuevamente y la entrada no fue reemplazada por otra petición, se puede devolver sin necesidad de recargar el bloque.

Una vez hechos múltiples allocs, si se completan todas las entradas del vector y no hay más lugar para un nuevo alloc, se hace un realloc de la tabla manualmente y se duplica su tamaño.

Por último, si se desea liberar la caché solo hay que liberar todos los bloques escribiendo a disco si están sucios, y liberar la memoria tanto del vector como de la caché.

6 Ext2

6.1 Por qué EXT2

Había varios filesystems para elegir. JuampiOS implementa el de MINIX que es a decir verdad un precursor de EXT2. El problema con el filesystem de MINIX es que la documentación y el soporte es inexistente, además de que es un filesystem más limitado. Luego había otras opciones, como FAT o ISO9660, pero las extensiones de FAT están patentadas por Microsoft y es muy limitado e ISO9660 si bien era una buena opción ya que es el filesystem estándar de los CDs, no es booteable por GRUB y tiene poco soporte fuera de los CDs.

En contraste, EXT2 es el filesystem por defecto de Linux, diseñado específicamente para él, y tiene soporte a nivel global tanto por Bootloaders como Sistemas Operativos. EXT2 si bien no tiene una especificación documentada más allá del source code de Linux, tiene documentación no oficial bastante buena en especial una realizada por Dave Poirier[8] que es la más completa y explicativa.

Además, EXT2 tiene múltiples features que pueden ser soportadas a futuro si es necesario, y fue incrementalmente mejorado en EXT3 y EXT4, si es que se llegan a necesitar incluso más features.

6.2 Estructura

EXT2 está basado en el filesystem clásico de Unix. A grandes rasgos:

- Se divide el espacio de memoria a utilizar en **bloques** contiguos de igual tamaño, indizados por números enteros de 32 bits a partir del 0.
- Los bloques se agrupan a su vez en **grupos de bloques** contiguos de igual tamaño.
- Los archivos son representados por el concepto de **inodos** que son estructuras dentro de una tabla en cada grupo, estos apuntan a una determinada cantidad de bloques utilizando sus índices que forman el contenido del mismo. Notar que esto incluye a los directorios, que son un tipo especial de archivo.
- La existencia del **superbloque**, que está en una posición fija dentro del filesystem y contiene los datos más generales del mismo, el cual es utilizado como punto de entrada para detectar su contenido.
- Luego del superbloque se encuentra una tabla con descriptores de grupos de bloque, con datos necesarios para poder leer cada grupo.

Existen dos versiones de EXT2 bien marcadas, la versión *major* 0 y la *major* 1. La actual y más usada de la que vamos a hablar es la versión 1, donde se agregó un poco de flexibilidad y más parámetros al superbloque e inodos.

6.2.1 Tamaños

Para ofrecer una visión más precisa, el tamaño de cada elemento en EXT2 está dado por:

Superbloque Siempre 1KiB

Bloque Parametrizable, está denotado por la fórmula $1KiB * 2^n$, siendo n un número mayor o igual a cero que se almacenará en el superbloque para poder parsear correctamente.

Grupo Depende del tamaño del bloque. Como cada grupo contiene un bloque que define un mapa de bits indicando el estado de sus bloques (Libre u Ocupado), los grupos miden como mucho *[Tamaño de un bloque en bits]* bloques. Por ejemplo, para bloques de 2KiB, miden como mucho 16Ki bloques, este valor está ubicado en el superbloque.

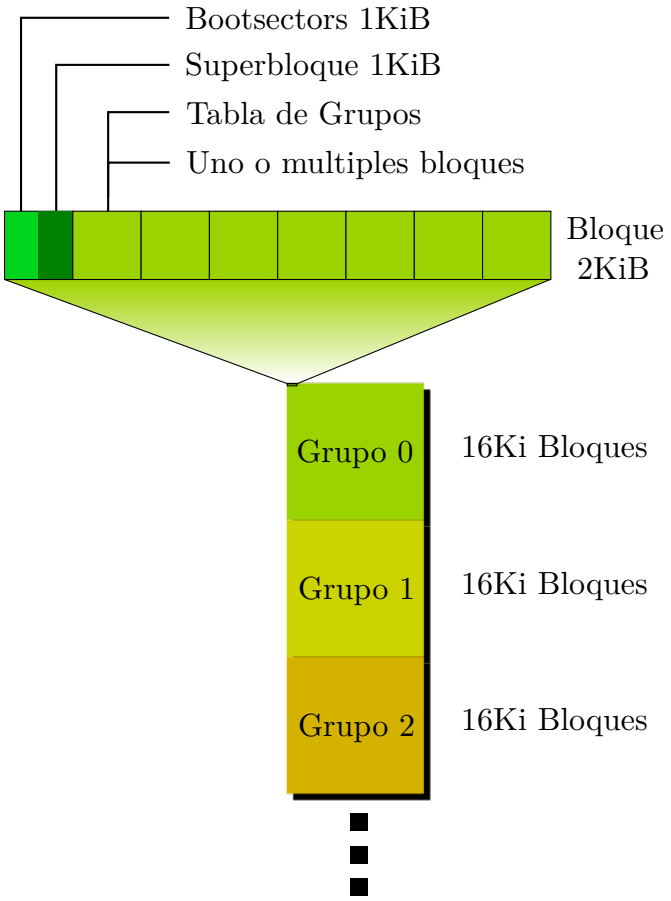
Inodo Originalmente siempre 128 bytes. Se hizo variable en la versión 1, en ese caso el tamaño en bytes está ubicado en el superbloque y debe ser una potencia de 2 mayor a 128 y menor al tamaño de bloque.

Descriptor de grupo Esto es, una entrada de la tabla de grupos, siempre 32 bytes.

[8] http://www.nongnu.org/ext2-doc/ext2.html

6.2.2 Estructura general

Un ejemplo de estructura general de EXT2 es el siguiente:



Estructura general de EXT2 con bloques de 2KiB

- El primer KiB está siempre reservado para los sectores de booteo.
- El segundo KiB contiene siempre al superbloque que mide exactamente 1KiB.
- A partir del bloque siguiente a donde se encuentre el superbloque[9] se encontrará la tabla de grupos de bloque, la cual puede ocupar varios bloques.
- El grupo 0 comienza contando desde el bloque donde se encuentre el superbloque[9].

A su vez, cada grupo contiene lo siguiente:

- Un bitmap de bloques, esto es una serie de bits que indican qué bloques tiene libres (Bit en 0) y qué bloques tiene ocupados (Bit en 1), debe entrar en un bloque.
- Un bitmap de inodos, esto es una serie de bits que indican qué inodos tiene libres (Bit en 0) y qué inodos tiene ocupados (Bit en 1), debe entrar en un bloque.
- Una tabla de inodos, la cantidad de inodos por grupo (Y por lo tanto, por tabla) está definida en el superbloque. La cantidad de inodos está delimitada por el bitmap de inodos y debe ser múltiplo de la cantidad de inodos que entran en un bloque. Todos los inodos del filesystem están indizados absolutamente desde 1, esto es, si un grupo contiene 2048 inodos, el grupo 0 contendrá los inodos 1 a 2048, el grupo 1 los inodos 2049 a 4096, etc.

Utilizando la tabla de descriptores de grupo, obtenemos datos del contenido de cada grupo, esto es en orden:

- El número de bloque absoluto del bitmap de bloques.
- El número de bloque absoluto del bitmap de inodos.
- El número de bloque absoluto del primer bloque de la tabla de inodos.
- La cantidad de bloques libres
- La cantidad de inodos libres
- La cantidad de inodos que son directorios

[9] Como el tamaño de bloque es $1KiB * 2^n$ para algún n, para bloques de 1KiB el superbloque estará en el bloque 1 y para cualquier otro tamaño, en el bloque 0 como en el ejemplo de la imagen. Esto es solo un caso borde que hay que tener en cuenta.

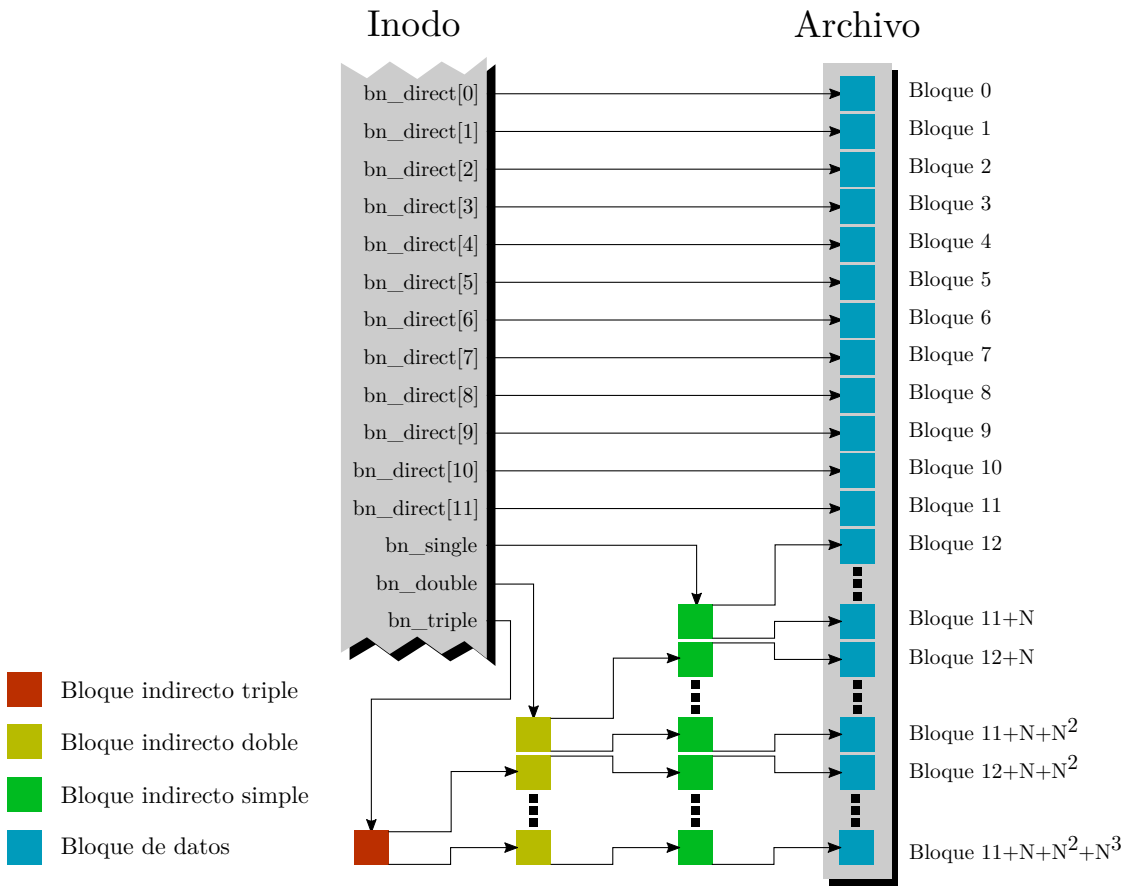
Los bloques y los inodos son siempre indizados globalmente, la división en grupos no cambia este hecho. Solo falta entender el concepto de inodos y cómo se relacionan con los bloques para poder ser capaces de leer correctamente el filesystem.

6.2.3 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 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:



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 = block_size / 4, ya que un número de bloque son 4 bytes

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 $block\_size/4$, 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 $N = block\_size/4$. En total nos quedan $12 + N + N^2 + N^3$ bloques de datos[10]. Para bloques de 1KiB, esto es 16GiB de tamaño máximo de archivo, sin contar el costo de los bloques indirectos.

6.2.3.1 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 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.

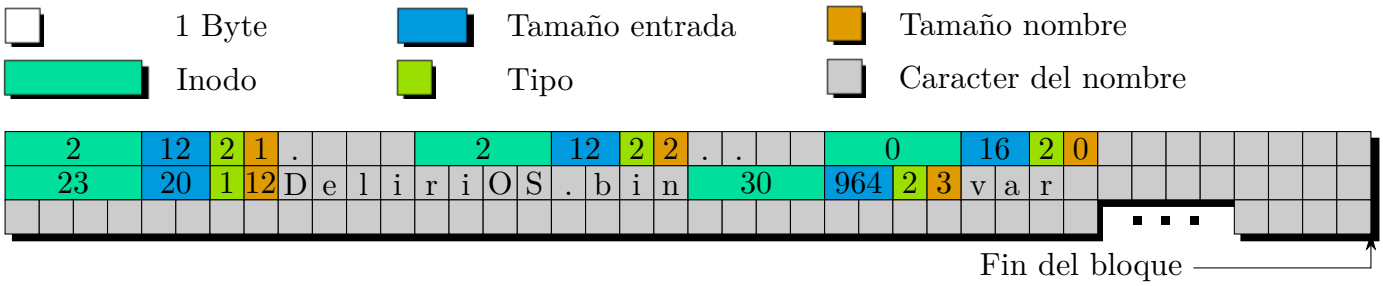
[10] Si tomamos los 12 bloques apuntados por el inodo, más los N bloques apuntados por el bloque indirecto simple, más los N^2 bloques apuntados por los N bloques simples apuntados por el bloque indirecto doble, más los N^3 bloques apuntados por los N^2 bloques simples apuntados por los N bloques dobles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

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:



Estructura de un directorio raíz con bloques de 1KiB

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.

.. 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[11].

Dicho esto, ya tenemos una idea general de cómo se recorre EXT2, al menos para lectura.

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

6.3 Implementación

Como dijimos en secciones anteriores, DeliriOS posee una interfaz de funciones de filesystem que tenemos que cumplir para poder ser accedidos por las syscalls.

6.3.1 FS Structs

Una de las cosas importantes de esta interfaz es que tenemos que definir tres structs nosotros mismos, estos son:

6.3.1.1 fs_info En este struct tenemos que definir los datos globales que queremos guardar del filesystem. En todas las funciones de la interfaz nos pasan un puntero a esta estructura. Definimos los siguientes campos:

```
typedef struct fs_info {
    block_cache* head_cache;   // Cache con el superbloque y la tabla
    block_cache* cache;        // Cache general para cargar bloques
    ext2_superblock* sb;       // Puntero directo al superbloque
    ext2_group_desc* gt;       // Puntero directo a la tabla de grupos
    uint64_t block_size;       // Tamaño de un bloque pero en bytes
    uint64_t block_shift;      // Shift para dividir por el tamaño de un bloque
    uint64_t group_count;      // Cantidad de grupos de bloque
} fs_info;
```

Primero necesitamos dos cachés. Una de ellas la utilizaremos para acceder tanto al superbloque como a la tabla de grupos. La otra la usaremos para acceder al resto del filesystem a medida que se requiera.

La razón por la cual creamos una caché aparte para el superbloque y la tabla de grupos, es porque necesitamos que estos estén en memoria todo el tiempo. A la vez, necesitamos que la tabla de grupos no esté fragmentada entre bloques para poder accederla como un arreglo. Por lo tanto para aprovechar la funcionalidad de la caché, creamos una con tamaño de bloque que abarque desde el comienzo hasta la tabla y cargamos solo el primer bloque.

A partir de ahí podemos crear un puntero directo al superbloque y un puntero directo a la tabla que son los dos siguientes campos.

Luego vienen tres variables que son útiles, calculadas a partir de los datos del superbloque, son utilizadas según convenga en las distintas funciones:

block_size El tamaño en bytes del bloque.
block_shift El logaritmo en base 2 de block_size tal que hacer (n » block_shift) sea equivalente a (n / block_size) para hacer la división más rápido.
group_count La cantidad de grupos en el filesystem.

6.3.1.2 fs_file En este struct tenemos que definir los datos que queremos guardar de un archivo. En todas las funciones de la interfaz que manipulen un archivo nos pasan un puntero a esta estructura.

Un archivo clásico tiene que ser capaz de poder ser recorrido tanto con lecturas, escrituras o búsquedas, moviendo la posición del puntero acorde a cada acción dentro de los límites del archivo.

Para hacer eso definimos los siguientes campos:

```
typedef struct fs_file {
    ext2_inode *inode;         // Puntero al inodo de este archivo
    uint64_t remaining;        // Cantidad de bytes que quedan por leer
    uint64_t block_offset;     // Offset actual dentro del bloque
    uint32_t block_number;     // Número de bloque actual
    uint32_t block_index;      // Indice de puntero a bloque actual
    uint32_t single_index;     // Indice del bloque indirecto simple actual
    uint32_t double_index;     // Indice del bloque indirecto doble actual
    uint32_t *singly_indirect; // Puntero al bloque indirecto simple
    uint32_t *doubly_indirect; // Puntero al bloque indirecto doble
    uint32_t *triply_indirect; // Puntero al bloque indirecto triple
    /* Usado en ext2_write */
    uint32_t inode_number;     // Numero de inodo del archivo
    uint64_t prealloc_count;   // Cantidad de bloques prealloc
    uint32_t prealloc_block;   // Primer bloque prealloc
} fs_file;
```

De primera, se definen los siguientes campos:

inode Puntero al inodo, cuando cargamos el archivo cargamos el bloque que contiene al inodo que lo representa y dejamos este campo apuntando a él. De esta forma podemos acceder a los datos del archivo constantemente.

remaining Cuántos bytes quedan hasta el final del archivo. Esto sería el tamaño del archivo menos la posición en bytes donde estamos ubicados dentro del archivo. Cuando se abre un archivo, esto tiene el tamaño del archivo, ya que empieza en la posición 0. A medida que se hagan operaciones sobre el archivo variará acorde.

block_number El número de bloque absoluto de datos en el que estamos posicionados actualmente. Ni bien se cargue el archivo, este será el indicado por bn_direct[0] en el inodo. Si se avanza lo suficiente como para caer en el siguiente bloque, este tendrá el valor de bn_direct[1]. Si en vez de eso se avanza 12 bloques tendrá el valor del primer número del bloque indirecto simple apuntado por el inodo, y así sucesivamente.

block_index El equivalente a la posición del archivo pero en bloques. Este es el número que le habíamos asignado a los bloques en la imagen de ejemplo. Arranca en 0 al abrirse, y vale exactamente (posición del archivo en bytes / block_size) en cualquier otro momento.

block_offset El offset en bytes dentro del actual bloque. Básicamente es el sobrante en bytes de block_index. Se calcula como (posición del archivo en bytes % block_size).

Es muy importante saber que, en todo momento, se da el siguiente invariante:
$block\_index * block\_size + block\_offset + remaining = file\_size$
Ya que nos deja en claro dónde estamos parados en el archivo exactamente. Luego se define:

single_index El equivalente a block_index pero para los bloques indirectos simples, aunque partiendo desde 1. Si es un número distinto de 0, nos va a indicar qué bloque indirecto simple tenemos cargado en singly_indirect, y lo podemos utilizar para comparar y saber si tenemos que cargar otro al movernos en el archivo. Esto es, vale 0 si no hay bloque simple cargado. Vale 1 si está cargado el bloque indirecto simple apuntado por el inodo, y así sucesivamente con el resto de los indirectos simples apuntados por bloques indirectos dobles.

singly_indirect Puntero al bloque indirecto simple cargado en memoria actualmente si es que hay uno, siempre es coherente con single_index.

double_index Lo mismo que single_index pero para bloques dobles, esto es, vale 0 si no hay nada cargado, 1 si es el bloque indirecto doble apuntado por el inodo y 2 en adelante si es alguno de los que pertenecen al bloque indirecto triple.

doubly_indirect Puntero al bloque doble indirecto cargado si es que hay uno. Siempre es coherente con double_index.

triply_indirect Puntero al bloque triple indirecto cargado si es que está cargado. Ya que hay solo uno, no es necesario un índice para identificarlo; o está cargado y es el del inodo, o no lo está.

Notar que tanto single_index como double_index no necesariamente son coherentes con block_index en la implementación por temas de optimización. Si se avanzara en el archivo lo suficiente, y quedaran cargados los respectivos bloque doble y simple de esa posición, y luego se volviese al principio del archivo ni single_index ni double_index van a cambiar. Permanecerán cargados hasta que sea necesario reemplazarlos por otros.

Lo mismo pasa con el bloque indirecto triple, que una vez cargado permanecerá así hasta que se cierre el archivo.

Los otros campos son usados en ext2_write que queda fuera del alcance de este trabajo.

6.3.1.3 fs_dir

En este struct tenemos que definir los datos que queremos guardar de un directorio. En todas las funciones de la interfaz que manipulen un directorio nos pasan un puntero a esta estructura. Definimos los siguientes campos:

```
typedef struct fs_dir {
    fs_file file;              // Struct del directory file
    dir_entry data;            // Datos de la entrada en la que estamos
    /* Usado en ext2_write */
    uint16_t rec_len;          // Longitud de la entrada actual
    uint32_t rec_inode;        // Inodo de la entrada actual
} fs_dir;
```

Como un directorio es a la vez un archivo en EXT2, simplemente guardamos un archivo y usamos los datos del archivo-directorio correspondiente. A la vez, como en readdir tenemos que devolver un puntero a una entrada de directorio genérica POSIX nos vemos forzados a incluir una en el struct.

Nuevamente, los otros campos son usados en ext2_write que queda fuera del alcance de este trabajo.

6.3.1.4 Superbloque

No tomamos todos los datos del superbloque, pero son relevantes:

ext2_magic Necesitamos que sea 0xEF53, si no, no es un EXT2 válido.

fs_state Necesitamos que esté en CLEAN, que es cuando está en 1, si no, tiene inconsistencias y no podemos leerlo.

req_flags Estos son los flags de features requeridos para poder interactuar con el filesystem. Si el driver de EXT2 no soporta alguno de estos features, no debería ser montado. Nosotros solo soportamos (y requerimos) el flag DIR_TYPE, que como dijimos anteriormente toma la parte alta del tamaño del nombre de las entradas de directorio como el tipo de archivo. Por default, **mkfs** activa siempre dicho flag.

req_write_flags Estos son los flags de features requeridos para poder escribir en el filesystem. En este trabajo no le dimos soporte de escritura al driver así que no nos interesan por ahora.

optional_flags Los flags de features opcionales del filesystem, los cuales son útiles para ganar performance si se acatasen. No es necesario tenerlos en cuenta (al menos en lectura). Por eso los ignoramos.

bn_superblock El número de bloque donde está ubicado el superbloque, esto es 1 para bloques de 1KiB, o 0 para cualquier otro. Necesario para saber en qué bloque comienza la tabla de bloques.

block_size_shift Nos dice el tamaño de bloque como $1024 << block\_size\_shift bytes$. Lo utilizamos para calcular block_size y block_shift en fs_info.

number_blocks y blocks_per_group Cantidad de bloques en el filesystem y cantidad de bloques por grupo respectivamente. Necesario para calcular el group_count en fs_info, como $\lceil number\_blocks / blocks\_per\_group \rceil$.

inode_size e inodes_per_group Tamaño de un inodo y cantidad de inodos por grupo respectivamente. Necesario para poder recorrer las tablas de inodos.

7 Loader

7.1 Introducción

El objetivo para el loader de binarios es soportar la carga estática de binarios en formato ELF64. Cabe destacar que la elección de ELF64 como formato predilecto está basada fundamentalmente en el amplio soporte que tiene. Otra alternativa es inventar un formato propio donde se podría, por ejemplo, especificar más de un punto de entrada al programa para que cada procesador lógico ya se encuentre en su sección de código que le corresponde. Sin embargo, las ventajas que esto pueda ofrecer no son una prioridad obvia. De hecho, existen más herramientas adaptadas al trabajo con modelo de *threads* que el de programas separados para cada procesador.

7.2 Características soportadas

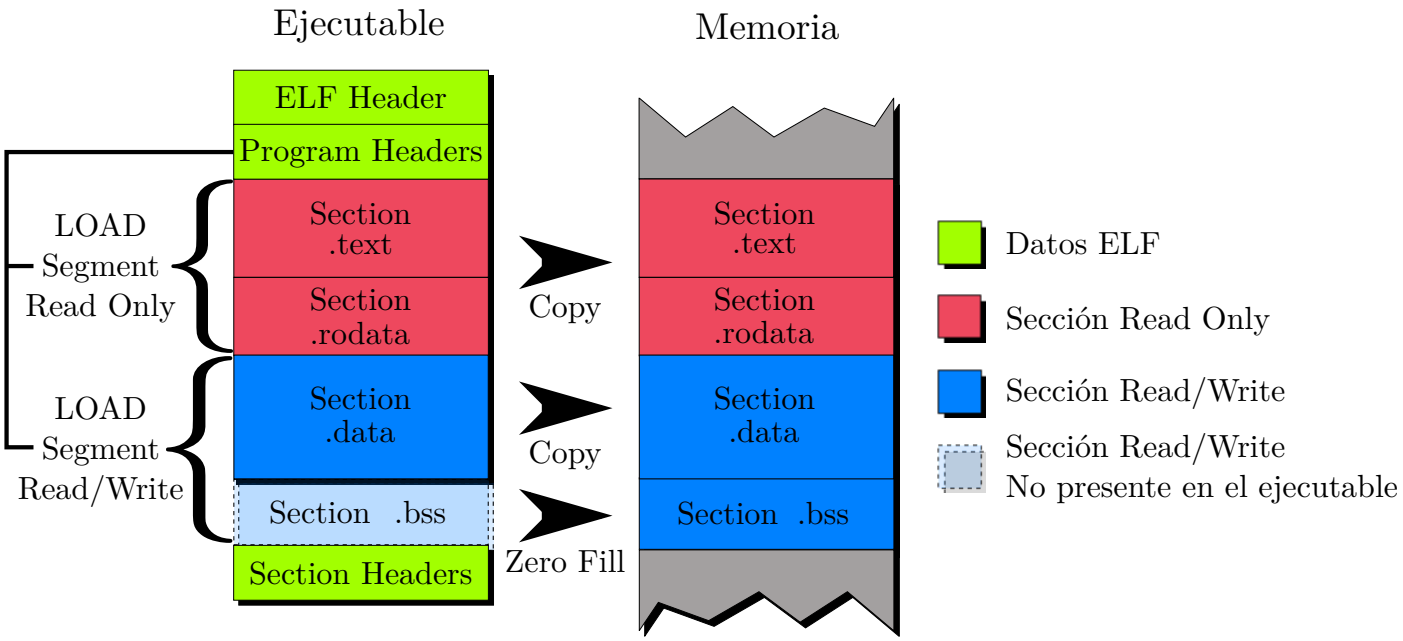
Se puede cargar un ejecutable estático desde un archivo (el cual se lee mediante la interfaz de *input/output*). Al finalizar la ejecución de la aplicación mediante la syscall **exit()**, se restaura el contexto del shell para poder continuar con otras tareas.

Los ejecutables deben ser linkeados con un linker script que los ubique en un área de memoria libre de DeliriOS. Actualmente no existen mecanismos que protejan la memoria del kernel así que es responsabilidad del desarrollador de aplicaciones linkear correctamente para poder restaurar el contexto y asegurar el funcionamiento correcto de las syscalls. En el repositorio se encuentran algunas aplicaciones de ejemplo donde se linkea a la dirección donde comienza el espacio reservado para las aplicaciones, pero se podría linkear la dirección base en cualquier otro punto de este espacio también.

7.3 Implementación

El loader está implementado para leer un archivo del sistema de archivos virtual, cargarlo y ejecutarlo si es un binario válido.

7.3.1 Carga



Segmentos de un binario ELF y su relación con la memoria

Lo primero que hace el loader es leer el encabezado del archivo. Carga los primeros bytes en memoria y los interpreta según esta estructura definida por el formato ELF64:

```
//ELF64 file object header
typedef struct {
    char magic[16];
    uint16_t type;
    uint16_t machine;
    uint32_t version;
    uint64_t entry_point;
    uint64_t ph_offset;
    uint64_t sh_offset;
    uint32_t flags;
    uint16_t header_size;
    uint16_t ph_entry_size;
    uint16_t ph_entry_count;
    uint16_t sh_entry_size;
    uint16_t sh_entry_count;
    uint16_t sh_string_table_index;
} __attribute__((__packed__)) elf_header;
```

magic El lugar donde se encuentra el número mágico de ELF. Si los primeros cuatro bytes no son 0x7f'ELF', rechazamos el archivo como inválido. También revisamos otros bytes para verificar que se trata de un archivo ELF64 y que sus estructuras de datos están organizadas como little endian.
type Indica si es un objeto reubicable, cargable o compartido entre otras posibilidades. Nosotros soportamos únicamente los cargables.
version Indica la versión del formato ELF64. Actualmente tiene un único valor porque no se hicieron revisiones aún.
entry_point La dirección virtual donde se debe comenzar la ejecución en un archivo ejecutable. Si no se trata de un ejecutable, siempre es cero.
ph_offset Es el offset del archivo en bytes donde se encuentra la *program header table*. Esta está presente en todo ejecutable y en ella están descriptos los segmentos a cargar.
sh_offset Es el offset del archivo en bytes donde se encuentra la *section header table*. Esta está presente en todo objeto compartido [12] y/o reubicable [13]. Como no ofrecemos soporte para estas características, ignoramos esta tabla.
flags Se trata de flags característicos para cada arquitectura de procesador. Actualmente tienen que ver con los procesadores que soportan tanto orden *little endian* como *big endian*. Aquí se informa al loader cuál de los dos modos espera el archivo objeto. En nuestro caso lo ignoramos ya que solo soportamos arquitectura AMD64.
header_size El tamaño en bytes de este header. Actualmente lo ignoramos pero puede servir para revisiones futuras del formato donde se agreguen campos al encabezado.
ph_entry_size El tamaño en bytes de cada elemento en la *program header table*.
ph_entry_count La cantidad de elementos en la *program header table*.
sh_entry_size El tamaño en bytes de cada elemento en la *section header table*.
sh_entry_count La cantidad de elementos en la *section header table*.
sh_string_table_index Es el índice de la *section header table* de la sección que contiene la tabla de nombres en strings para todas las secciones.

Al momento de cargar en memoria el binario es necesario buscar la *program header table* e interpretar cada entrada en ella:

```
//ELF64 program header entry
typedef struct {
    uint32_t type;
    uint32_t flags;
    uint64_t offset;
    uint64_t virtual_address;
    uint64_t physical_address;
    uint64_t file_size;
    uint64_t memory_size;
    uint64_t align;
} __attribute__((__packed__)) elf_pheader;
```

type Este atributo identifica el tipo de segmento descripto. El loader busca únicamente aquellos que son para cargar en memoria tal como están. El valor en cuestión está definido como **ELF_LOAD** aunque en el estándar se puede encontrar como **PT_LOAD**.
flags Describe atributos del segmento. Habitualmente se indica aquí que la memoria que albergue los datos del segmento debe ser de solo lectura o permitir la ejecución de su contenido. También tiene 16 bits reservados en la parte superior para uso del sistema operativo (ocho de ellos son características dadas de acuerdo al procesador). Como DeliriOS no provee tipo alguno de protección, ignoramos este campo.
offset Especifica el offset, en bytes, desde el comienzo del archivo donde se encuentra el segmento.
virtual_address Contiene la dirección de memoria virtual del segmento.
physical_address Este atributo está reservado para sistemas con direccionamiento a memoria física.
file_size Contiene el tamaño en bytes de la imagen en archivo del segmento.
memory_size Contiene el tamaño en bytes de la imagen en memoria del segmento.
align Especifica el alineamiento requerido. **virtual_address** y **offset** deben ser congruentes módulo **align**.

Para cargar en memoria lo que se hace es, una vez encontrado un *program header entry* en esta tabla que sea cargable, leer la información básica del segmento (dirección virtual, offset de la imagen en el archivo) y realizar el copiado de la imagen del segmento en el archivo a memoria. Cada segmento cargable puede tener un tamaño en

[12] El cual contiene código independiente de su posición en memoria en tiempo de ejecución.
[13] El cual contiene código dependiente de su posición en memoria que aún tiene etiquetas indefinidas. Estas deben ser resueltas (linkeadas o enlazadas) por el loader para el correcto funcionamiento del objeto.

memoria más grande que el de la imagen en el binario. El loader se encarga de inicializar esta memoria sobrante a cero tal como indica el estándar ELF.

Al terminar la carga se escribe en la siguiente estructura el **entry_point** y el **loader_error**:

```
typedef struct application_data {
    uint64_t entry_point;
    uint64_t running_status; // 1 si se está ejecutando una aplicación.
    uint64_t stack_pointer;  // Stack pointer antes de empezar a ejecutar.
    uint64_t exit_code;      // Código de terminación de la aplicación.
    uint64_t exit_core_id;   // Id del núcleo que efectivamente terminó.
    int64_t loader_error;    // Código de error del loader.
} application_data;
```

entry_point La dirección virtual donde debe comenzar la ejecución el loader. La copiamos tal como está del encabezado ELF64.

running_status Este flag indica si hay una aplicación ejecutándose. Si el loader no cargó una aplicación o si se salió de una aplicación este flag es cero. En caso contrario es uno. Lo utilizamos para hacer un spinlock sobre los procesadores lógicos que intentaron terminar la aplicación justo cuando otro procesador ya estaba iniciando el protocolo de terminación.

stack_pointer Aquí guardamos el puntero del stack antes de saltar al punto de entrada de la aplicación. Con esto podemos restaurar el contexto del kernel previo al lanzamiento de la aplicación.

exit_code Aquí guardamos el valor recibido por parámetro en la syscall **exit()**. La idea es utilizarlo como un código de terminación para proveer feedback al que lanza la aplicación.

exit_core_id Aquí guardamos el identificador del procesador que llamó a la syscall **exit()**. Si bien es posible obtener esta información realizando acrobacias con el código de terminación, preferimos facilitar esta información dado que es excesivamente fácil de obtener durante la ejecución de la syscall.

loader_error Cuando falla el loader en la fase de carga se guarda el número de error aquí para poder imprimir la información pertinente en el shell. Si no falla el loader, se le asigna cero.

Cuando se llega al final de la tabla sin haber encontrado ningún error, el loader finaliza la carga y pasa a ejecutar la aplicación.

7.3.2 Ejecución

Con esto, solo queda transicionar y ejecutar la aplicación en sí. Para hacer esto se guardan los registros del procesador para cumplir con la convención C para AMD64 en la pila. Luego se guarda el puntero del stack en **stack_pointer** de **application_data** y se asigna un uno a **running_status** lo que indica que está corriendo una aplicación en el sistema. A continuación se realiza el salto al punto de entrada. Todo esto es algo que ocurre en el BSP ya que es el que se encarga de ejecutar el shell.

En los APs lo que ocurre es que se encuentran esperando a que **running_status** sea uno. Cuando esto se cumple, saltan a **entry_point**. De esta forma todos los procesadores se encuentran ejecutando la aplicación. En el caso de los APs no resulta necesario guardar el puntero a la pila ya que se encuentra vacía en el momento en que se encuentran esperando. Por lo tanto es solo cuestión de reinicializar el puntero como si se iniciara el procesador por primera vez.

7.3.3 Terminación

La terminación está soportada mediante una syscall. Terminar la aplicación no implica una garantía del correcto funcionamiento del kernel ya que la aplicación corre en privilegio 0. Sin embargo, mientras no se escriba en lugares sensibles de la memoria del kernel o se modifiquen registros de control del procesador, se podrá restablecer el contexto de ejecución del shell.

Cuando un thread[14] llama a la syscall exit, lo primero que se hace es adquirir un lock sobre la terminación de la aplicación. Esto se realiza con un exchange atómico sobre el flag que indica el estado del sistema en **application_data**. Si no se adquirió con éxito el lock, se para el procesador para esperar a que llegue la señal de terminación de otro thread. En caso contrario, se guarda el identificador del procesador lógico que está terminando la aplicación en **exit_core_id** y el código de error recibido en **exit_code** en la estructura **application_data**. Luego, se envía una interrupción a todos los procesadores incluyendo al actual.

La rutina de atención de esta interrupción se encarga de restaurar el contexto del shell. En el caso de los APs, esto significa colocarlos en espera reinicializando el puntero del stack. En el caso del BSP, se resume la ejecución luego del salto al punto de entrada con el mismo puntero del stack que se tenía antes de lanzar la aplicación y se restauran los registros del procesador guardados en el stack para cumplir con la convención C para AMD64.

[14] Aquí, por la naturaleza de DeliriOS, un thread equivale a un procesador lógico en tiempo completo.

8 Syscalls

8.1 Visión general

Si bien no era parte del objetivo encarar el aspecto de las llamadas al sistema de manera orgánica, no pudimos evitar implementar una interfaz liviana para facilitarlas. Los objetivos que nos propusimos para esta interfaz fueron dos: sencillez en implementación y sencillez en adopción.

8.2 Características de la interfaz

Para poder efectuar una llamada al sistema lo que se requiere es entregarle el flujo de ejecución al kernel. Teniendo en cuenta que el kernel de DeliriOS no tiene protección, se hace innecesariamente engorrosa la interfaz a través de interrupciones. En esencia, lo que queremos es llamar a las funciones definidas en el kernel de la manera más directa posible. Lo cual se podría hacer si tuviéramos la dirección donde está ubicada cada función.

Es por esto que armamos una tabla de direcciones a las funciones que nos interesa acceder desde el espacio de una aplicación. Esta se encuentra en una dirección específica de la memoria baja[15]. De esta manera, es fácil utilizar estos punteros para hacer las llamadas en cualquier momento que sea necesario. Naturalmente, esto significa que no se trata de una llamada directa, si no que hay un nivel de indirección.

8.3 Integración

Se escribieron los prototipos y las declaraciones necesarias en un encabezado `syscall.h` para que el uso de una syscall cualquiera sea natural en una aplicación. Sin embargo, además de incluir este encabezado, es necesario pasarle una directiva al linker para definir el símbolo **SYSCALL_TABLE** en la dirección específica antes mencionada. Hecho esto, la aplicación está lista para utilizar todas las syscalls definidas por DeliriOS.

8.4 Syscalls Definidas

Las definidas en IO, y a la vez:

```
void exit(int64_t exit_code);
void sleep(uint64_t ticks);
uint64_t core_id();
uint64_t core_count();
```

exit Termina la ejecución de la aplicación sin importar qué core la haya llamado. Remitirse a la sección Loader para más detalles.

sleep Duerme al core una cantidad de milisegundos.

core_id Devuelve el id del core que la haya llamado. Devuelve 0 si es el BSP o cualquier otro número incremental si es un AP, usando el LapicID como diferenciador. Está garantizado que el id más alto es core_count - 1.

core_count Devuelve la cantidad de cores reconocidos por DeliriOS.

[15] Es decir, antes del kernel. Esto se debe a que reservar espacio por encima del kernel es muy propenso a errores durante las primeras etapas de desarrollo donde el tamaño del código del kernel varía considerablemente entre iteración e iteración.

9 Shell

El shell está compuesto de dos partes, el input loop y los comandos.

9.1 Input Loop

Primero se setea el current working directory a root y luego el loop:

1. Imprime el prompt
2. Lee una línea por entrada estándar
3. Se le pasa la línea al parser. Este borra los espacios sobrantes y deja un caracter nulo al final de cada palabra ingresada. A su vez es capaz de reconocer apóstrofes (') en caso de necesitar ingresar espacios y barras invertidas (\) en caso de tener que ingresar un espacio o un apóstrofe de forma literal.
4. Se toma la primera palabra como nombre de comando y se la busca en una tabla de comandos, que tiene un par nombre/puntero a función.
5. Se ejecuta el comando encontrado. Se le pasa como parámetro el resto de las palabras, mediante la clásica aridad **(int argc, char* argv[])**
6. El comando se encargará de interpretar los parámetros acorde a su función y devolverá un código de error que es impreso en pantalla si no es 0.

Nada más complejo que eso, una vez ejecutado el comando vuelve al comienzo del loop.

9.2 Comandos

Actualmente hay implementados cuatro comandos simples:

cd Cambia el *current working directory* al indicado, tanto el string del prompt como el directorio de IO.
ls Lista el contenido de un directorio.
echo Imprime los parámetros en STDOUT.
exec Ejecuta el binario pasado por parámetro.

En el caso de cd, se crearon unas funciones que se encargan de modificar el current path acorde dependiendo si fue relativo o absoluto el cambio de directorio.

10 Testing y Debugging

Si bien no hemos podido realizar tests exhaustivos, debido a la volatilidad del proyecto y falta de tiempo, pudimos dejar creada la interfaz para testear DeliriOS.

10.1 Linkeo

Para realizar un test sobre algún componente de DeliriOS, solo basta con linkear los archivos objeto de DeliriOS con algún ejecutable standalone capaz de realizar test sobre las funciones. Para el caso de EXT2, hemos creado este makefile:

```
# Compilador que será utilizado
CC=gcc

# flags de compilación
CFLAGS=-g -Wall -std=c11

# flags de linkeo
LDFLAGS=

# Carpeta con codigo
FS_DIR=fs/
FS_OBJECTS=ext2.o ext2_write.o block_cache.o
FS_FULL=$(addprefix $(OBJ_DIR)$(FS_DIR), $(FS_OBJECTS))

UTILS_DIR=utils/
UTILS_OBJECTS=bitmap.o
UTILS_FULL=$(addprefix $(OBJ_DIR)$(UTILS_DIR), $(UTILS_OBJECTS))

OBJ_DIR=../../obj/

OBJECTS= ext2_test.o io_clash.o

# Agrear acá los directorios a incluir en la compilación
INCDIR=../../src/include/ ./

# Nombre del ejecutable a generar
EXECUTABLE=ext2_test

all: $(EXECUTABLE)

$(EXECUTABLE): $(OBJECTS) $(FS_FULL) $(UTILS_FULL)
	$(CC) $(LDFLAGS) $^ -o $@

clean:
	rm -rf $(EXECUTABLE) $(OBJECTS)

io_clash.o: $(OBJ_DIR)syscall/io.o
	objcopy --redefine-syms=io_clash.sym $< $@

.c.o:
	$(CC) $(INCDIR:%=-I%) $(CFLAGS) $< -c -o $@
```

Se toman los objetos necesarios de DeliriOS para poder correr el filesystem.

A su vez debido a que las syscalls de DeliriOS tienen nombres similares a las syscalls de Linux, se les cambia el nombre a las syscalls del objeto `io.o` prefijándole *sys_* para evitar colisiones. Esto es, se usa `objcopy` y se le pasa una tabla de renombres en el parámetro `-redefine-syms`, indicando un renombre por línea.

Luego se puede crear un archivo de test que tenga implementado el resto de las funciones no linkeadas que pueden ser emuladas por fuera:

```
extern fs_info fs;
FILE* fp;
int main(int argc, char *argv[]) {
    if (!(fp = fopen("../../hdd/hdd.img", "rb+"))) {
        printf("hdd/hdd.img not found\n");
        return 1;
    }
    io_init();
    print_fs_data();
    fclose(fp);
    return 0;
}

void* kmalloc(uint64_t size) {
    return malloc(size);
}

void kfree(void* data) {
    free(data);
}

void hdd_read(uint32_t lba_address, uint32_t sectors, void* buffer) {
    fseek(fp, lba_address * ATA_SECTSIZE, SEEK_SET);
    fread(buffer, ATA_SECTSIZE, sectors, fp);
}

void hdd_write(uint32_t lba_address, uint32_t sectors, void* buffer) {
    fseek(fp, lba_address * ATA_SECTSIZE, SEEK_SET);
    fwrite(buffer, ATA_SECTSIZE, sectors, fp);
}

uint64_t screen_write(void* buffer, uint64_t count) {
    return fwrite((char *) buffer, 1, count, stdout);
}

uint64_t screen_read(void* buffer, uint64_t count) {
    return 0;
}

uint64_t screen_write_err(void *buffer, uint64_t count) {
    return fwrite((char *) buffer, 1, count, stderr);
}

void kernel_panic(const char* function,
                  const uint64_t line,
                  const char* file,
                  const char* message, ...) {
    va_list list;
    fprintf(stderr, "[KERNEL_PANIC]: ");
    va_start(list, message);
    vfprintf(stderr, message, list);
    va_end(list);
    fprintf(stderr, "\nFuncion que produjo el error: %s\n", function);
    fprintf(stderr, "(Archivo:Linea) %s:%lu\n", file, line);
    _Exit(1);
}
```

Donde printf_fs_data es una función que imprime datos del filesystem usando fs_info, y hará otros tests necesarios. Con esto somos capaces de tener un binario listo para ser corrido luego de que se hayan compilado los objetos de DeliriOS, capaz de decirnos si hay algún problema con la implementación.

10.2 Calltrace

Para facilitar un poco la parte de debugging agregamos un calltrace a DeliriOS. Este es utilizado ante un Kernel Panic, que puede suceder si se lo llama deliberadamente o se alcanza una excepción por algún motivo. También se lo puede llamar independientemente de un Kernel Panic.

```
[KERNEL PANIC]: Ejemplo de calltrace

Funcion que produjo el error:    fs_open
(Archivo:Linea)    src/fs/ext2.c:287

[CALL TRACE] (Puede haber funciones mal traceadas)
Function                Address
kernel_panic + DD       0x100BA6
fs_open + 2F            0x102EA8
open + 88               0x104A85
load_binary + 31        0x105166
execute_command + 86    0x104EF1
shell_init + 80         0x105044
kernel_main + A5        0x10036E
Unknown Function        0x0
Unknown Function        0x0
```

Ejemplo de calltrace en DeliriOS

Como mostramos anteriormente en la sección Refactor, se crea una tabla de símbolos donde cada entrada es un par con dirección y un puntero a string del nombre de la función. Para generar el calltrace se realiza lo siguiente:

1. Se toma el base pointer. Debido al stack frame[16], esta es la posición en el stack donde está ubicado el base pointer de la función que nos llamó. En la posición siguiente (Esto es siguiente en memoria pero anterior en el stack), se encuentra la dirección de retorno.
2. Tomamos la dirección de retorno y buscamos en la tabla de símbolos cuál es el símbolo más cercano a la función hacia atrás. La dirección de retorno puede estar en cualquier posición intermedia de la función, nosotros necesitamos el símbolo anterior que esté más cerca.
3. Imprimimos el símbolo encontrado junto con los datos de su ubicación, por ahora debido a que hay funciones estáticas este símbolo puede no ser el correcto. En caso de no encontrar ningún símbolo anterior imprimimos "Unknown Function".
4. Tomamos el base pointer de la función anterior y repetimos el proceso.

Hacemos esto una cantidad de veces limitada o hasta que el base pointer encontrado sea 0. Dicha cantidad de veces puede ser pasada como parámetro.

El calltrace es muy útil para descubrir qué sucedió dado un Kernel Panic, y agiliza el debugging.

[16] Ver flags de linkeo en la sección Refactor, es importante que esté activo `-fno-omit-frame-pointer` para que el compilador haga el stack frame incluso con optimizaciones

11 Trabajo a futuro

Hay realmente muchísimas cosas que se pueden hacer a futuro dada la base presentada en este trabajo.

Entre ellas las más inmediatas son:

Implementar una libc: Actualmente hay algunas funciones implementadas de libc, con las aridades correspondientes, entre ellas la mayoría de string.h, printf, etc. Se podría terminar de implementar el resto, o portear partes de otras libc como *musl*.

Agregar soporte de relocations al parser de ELF: Las reubicaciones son muy usadas para insertar módulos en los kernels, por ejemplo drivers. En nuestro caso permitiría eliminar la tabla de syscalls y a la vez podría permitir potencialmente que un binario llame a cualquier función de DeliriOS bajo su propia discreción, por ejemplo las funciones que hay implementadas de libc.

Hacer que exec sea una syscall: Esto permitiría que un binario pueda ejecutar a otro, por ejemplo daría la posibilidad de quitar el shell fuera del kernel. Hay planes de implementar un "stack de binarios", donde cada uno es pegado a continuación del otro en la memoria de carga, y los cores se transfieren entre los niveles.

Implementar más tests: Ya dada la interfaz para linkear los tests, es fácil crear otros tests que checkeen otros elementos de DeliriOS, por ejemplo el heap.

Implementar otros filesystems: Ya dada la interfaz es fácil implementar otros filesystems, como FAT por ejemplo.

Implementar otras interrupciones: Actualmente solo se utiliza la interrupción de teclado y el RTC, potencialmente hay otros tipos de interrupciones que pueden llegar a ser útiles, como por ejemplo los relojes más precisos HPET o APIC timer.

Mejorar el uso de las interrupciones: Las interrupciones las maneja el kernel de forma estática, se podría ofrecer una interfaz para ofrecerle las interrupciones a las aplicaciones que se ejecutan sobre DeliriOS.

Mejorar el manejo de Disco: Por ahora se leen sectores usando ATA PIO lo cual es muy lento y obsoleto, se debería implementar DMA y reconocer más Discos que no sean solo el Master.

Agregar soporte USB: Sin tener la capacidad de leer de dispositivos de almacenamiento masivos sobre USB la utilidad de DeliriOS se ve limitada. Ser capaces de implementar OHCI, UHCI, EHCI y/o XHCI podría aumentar el rango de posibilidades de uso del sistema operativo.

Agregar soporte de otros dispositivos: Por ejemplo soporte de red, uno de los objetivos iniciales de DeliriOS era ser capaces de utilizar máquinas remotas para realizar cálculos de alguna índole. Ser capaces de enviar instrucciones por la red y obtener resultados sería definitivamente un avance importante en DeliriOS.

Mejorar la interfaz Multicore: Actualmente la interfaz Multicore es limitada, y no se puede tener un manejo simple de los cores. Ser capaces de redireccionar cores a otras funciones, o pararlos/esperarlos podría facilitar el desarrollo de aplicaciones en DeliriOS.

Desarrollar un scheduler: Si bien no es el objetivo principal de DeliriOS ser de propósito general, se puede experimentar con otro tipo de Schedulers menos comunes, que en este nivel tan bajo pueden traer beneficios. Específicamente, para extender las capacidades de exokernel que DeliriOS tiene, se podría realizar un framework para definir schedulers propios. Esto implicaría modificar la interrupción de reloj y su entrada correspondiente en la IDT en tiempo de ejecución.

Documentar DeliriOS: Este informe es un buen punto de partida pero hay que documentar el uso de cada función de DeliriOS individualmente. Se puede usar Doxygen para esto.

Agregar soporte para sesiones interactivas de debugging: Esto implica tener el mapa de símbolos de todas las funciones de DeliriOS. Hecho esto, se pueden cargar en una sesión remota de gdb en qemu por ejemplo. Bochs también tiene soporte en el debugger interno para cargar símbolos si fuera necesario. Incluso se lo puede compilar con soporte para conectarse con gdb. DeliriOS por ahora es chico pero podría existir un futuro no muy lejano en el cual sea absolutamente necesario debuguear interactivamente por una cuestión de escalabilidad.