



Escuela
Politécnica
Superior

Desarrollo de un kernel académico para arquitecturas x86-64 en C++



Grado en Ingeniería Informática

Trabajo Fin de Grado

Autor:

Ernesto Martínez García

Tutor:

Antonio Miguel Corbi Bellot

Marzo 2022

Desarrollo de un kernel académico para arquitecturas x86-64 en C++

Implementación del kernel “alma” en C++ para arquitecturas x86-64 junto a su bootloader UEFI en C para ejecutar arranques con qemu

Autor

Ernesto Martínez García

Tutor

Antonio Miguel Corbi Bellot

Departamento de Lenguajes y Sistemas Informáticos



Grado en Ingeniería Informática



Escuela
Politécnica
Superior



Universitat d'Alacant
Universidad de Alicante

ALICANTE, Marzo 2022

Resumen

El siguiente trabajo tiene como objetivo el desarrollo y documentación de un kernel (*núcleo*) en C++ de funcionalidad muy reducida para manejar los recursos hardware de una máquina virtual `qemu` [1] x86-64 con arranque Unified Extensible Firmware Interface (UEFI) proporcionado por el kit de desarrollo para UEFI: EDK II [2]. Teóricamente no existen limitaciones para cargar el kernel desarrollado en hardware real pero se ha acotado al entorno de `qemu` por garantizar el funcionamiento de las *demos*.

En conjunto con el kernel también se ha desarrollado un bootloader capaz de cargarlo con el arranque UEFI de `qemu`. El bootloader se ha escrito en el lenguaje C usando la librería de desarrollo UEFI `posix-uefi` [3], que proporciona una capa de compatibilidad Portable Operating System Interface (POSIX) sobre la librería `gnu-efi` [4].

Puesto que la compilación manual del kernel es una tarea complicada debido a la dificultad de configurar una *toolchain* capaz de compilarlo, se ha diseñado una máquina virtual¹ ligera lista para construir el proyecto. También se ha proporcionado un directorio de descargas² de distintas versiones del kernel precompilado para poder probarlo sin compilarlo y sin necesidad de instalar dependencias.

El objetivo del proyecto no es desarrollar un kernel usable en hardware real ni útil para determinadas tareas, simplemente se busca desarrollarlo con fines académicos de aprendizaje como otros kernels similares desarrollados por otras universidades tales como xv6 [5] OS/161 [6] y SWEB [7].

¹<https://github.com/ecomaikgolf/alma#build>

²<https://github.com/ecomaikgolf/alma#run>

Abstract

Traducir el resumen anterior a inglés

Agradecimientos

Agradecimientos

El truco del cantamañanas es responder generalidades cuando se preguntan detalles y cuestionar trivialidades cuando se preguntan principios.

Ricardo Gali.

Índice general

1. Introducción	1
1.1. Motivación	1
1.2. Objetivos	2
2. Marco Teórico	5
3. Metodología	7
4. Toolchain	9
4.1. Alma Build VM	10
4.2. Dependencias de Ejecución	11
4.3. Dependencias de Construcción	12
4.3.1. GNU Compiler Collection (gcc)	12
4.3.2. GNU Binutils	14
4.3.3. EDK II OVMF	15
4.3.4. posix-uefi	16
5. Construcción del Proyecto	17
6. Bootloader	19
6.1. Introducción	19
6.2. Estado del Arte	22
6.3. Sistema de Construcción	25
6.4. Desarrollo del Bootloader	29
6.4.1. Errores	29
6.4.2. Logs	30
6.4.3. Ficheros	31
6.4.3.1. Obtener el Tamaño de un Fichero	31
6.4.3.2. Cargar un Fichero en Memoria	32
6.4.4. Gráficos	33
6.4.4.1. Graphics Output Protocol	33
6.4.4.2. Framebuffer	34
6.4.4.3. Fuente	36
6.4.5. Mapa de Memoria	39
6.4.6. ACPI	43
6.4.7. ELF	45
6.4.7.1. Cargar fichero ELF	48
6.4.7.2. Obtener y Verificar la Cabecera ELF	48

6.4.7.3. Cargar los Program Headers	50
6.4.7.4. Llamar a los Constructores Globales	51
6.4.8. Función Principal	53
6.5. Conclusiones	56
6.5.1. Errores conocidos	57
6.5.1.1. Carga del ELF	57
6.5.1.2. Salida de los servicios de UEFI	58
7. Desarrollo del Kernel	59
8. Conclusiones	61
9. Auxiliar	63
Bibliografía	67
A. Anexo I	69

Índice de figuras

4.1.	Alma Build VM	10
4.2.	alma 20211012-e906c4c-m build	11
4.3.	Red Zone en el Stack (https://os.phil-opp.com)	13
6.1.	Ejemplos de bootloaders	19
6.2.	Arranque UEFI	20
6.3.	Bootloaders comunes	22
6.4.	Funcionamiento SEEK_SET, SEEK_CUR y SEEK_END [8]	32
6.5.	Formato Píxel GOP	35
6.6.	Fuente PSF1 [9]	36
6.7.	Formato ELF	45
6.8.	Formato Código de Error	49
6.9.	Traza de ejecución del bootloader	55
9.1.	Caption	63
9.2.	Subreferences in L ^A T _E X.	64
9.3.	Directorio de archivos wrapped	64

Índice de tablas

6.1. Bootloaders más comunes [10]	22
---	----

Índice de Códigos

4.1. Submódulos git	9
4.2. Construcción de la toolchain de alma	9
4.3. Importar .ova de alma build VM	10
4.4. Arrancar alma build VM	10
4.5. Instalación en Ubuntu 20.04	11
4.6. Instalación en ArchLinux	11
4.7. Ejecución de build precompilada de alma	11
4.8. Instalación en Ubuntu 20.04	12
4.9. Compilación de <code>gcc 11.x</code>	14
4.10. Compilación de <code>binutils 2.37</code>	15
4.11. Compilación de EDK II OVMF	15
4.12. Compilación de EDK II OVMF	16
6.1. Sintaxis de <code>gnu-efi</code>	24
6.2. Sintaxis de <code>posix-efi</code>	24
6.3. Makefile básico de <code>posix-efi</code>	25
6.4. <code>posix-efi</code> cmake I	25
6.5. <code>posix-efi</code> cmake II	25
6.6. <code>posix-efi</code> cmake III	26
6.7. <code>posix-efi</code> cmake IV	26
6.8. <code>posix-efi</code> cmake V	27
6.9. <code>posix-efi</code> cmake VI	27
6.10. <code>err_values.h</code>	29
6.11. <code>err_values</code> ejemplo	30
6.12. <code>logs/stdout.h</code>	30
6.13. <code>io/file.h</code>	31
6.14. <code>uint64_t file_size(FILE *file)</code>	31
6.15. <code>void *load_file(const char *filename)</code>	32
6.16. <code>gop.h</code>	33
6.17. <code>efi_gop_t *load_gop()</code>	33
6.18. <code>framebuffer.h</code>	34
6.19. <code>framebuffer.h</code>	35
6.20. <code>font.h</code>	36
6.21. <code>PSF1_Font *load_psf1_font(const char* const filename)</code>	37
6.22. <code>PSF1_Header * get_psf1_header(const char *const memory)</code>	38
6.23. <code>uint8_t verify_psf1_header(const PSF1_Header *const header)</code>	38
6.24. <code>void *get_psf1_glyph(const char *const memory)</code>	38

6.25. memory/memory.h	40
6.26. efi_memory_descriptor_t	40
6.27. MapInfo *load_memmap()	40
6.28. void print_memmap(const MapInfo *map)	42
6.29. rsdp_v1	43
6.30. rsdp_v2	43
6.31. rsdp_v2 *load_rsdp()	44
6.32. efi_guid_t	44
6.33. bool compare_guid(efi_guid_t *g1 efi_guid_t *g2)	45
6.34. ELF Header	45
6.35. ELF Program header	46
6.36. ELF Section header	47
6.37. Elf64_Ehdr *load_elf(const char *const filename)	48
6.38. Elf64_Ehdr *get_elf_header(char *memory)	48
6.39. uint8_t verify_elf_headers(const Elf64_Ehdr *const elf_header)	49
6.40. void load_phdrs(const Elf64_Ehdr *const elf_hedaer	50
6.41. Ejemplo de función constructora	51
6.42. Ejemplo de objeto global con constructor	51
6.43. void call_ctors(Elf64_Ehdr *elf)	52
6.44. int main()	53
6.45. Error en void load_phdrs(...)	57
6.46. Error en la salida del los UEFI services	58
9.1. Prototipo tarea cargar kernel	65

1. Introducción

1.1. Motivación

El ámbito de los sistemas operativos y núcleos es una disciplina ampliamente trabajada por desarrolladores de bajo nivel e investigadores teóricos. Al contrario de lo que sucede con las investigaciones teóricas, la bibliografía relacionada con el desarrollo práctico de estos es escueta y antigua. Algunos proyectos *Open Source* modernos carecen de explicaciones profundas sobre su funcionamiento interno, limitándose a proveer documentación para poder continuar con el desarrollo del código, lo que dificulta la intruducción a esta disciplina.

La escasez de documentación práctica centrada en el desarrollo de un núcleo provoca que el poco contenido que hay utilice tecnologías antiguas. La mayoría de materiales encontrados al respecto se alejan del ámbito académico, donde solo ciertas universidades reconocidas proveen asignaturas que abordan la temática. En Harvard encontramos el curso *CS161 (Operating Systems)* donde se parte de un esqueleto de sistema operativo llamado “OS/161” donde los alumnos tienen que implementar funcionalidades tales como paginación, *syscalls*, etc. En el “Institute of Applied Information Processing and Communications” de la Universidad Tecnológica de Graz tienen la asignatura *INP32512UF (Operating Systems)* donde trabajan con *SWEB*, un sistema operativo de la universidad sobre el que los alumnos tienen que trabajar con *mutexes*, *FPU*, paginación, etc. La Universidad de Wisconsin-Madison tiene las asignaturas *CS-537 (Introduction to Operating Systems)* y *CS-736 (Advanced Operating Systems)*, donde la primera utiliza el núcleo académico *xv6* desarrollado por el Instituto Tecnológico de Massachusetts, donde se imparte la asignatura *6.S081 (Operating System Engineering)* para el que se desarrolló el núcleo.

Añadir enlaces

Durante mi estancia en la Universidad de Alicante no he visto que ningún departamento tenga líneas de trabajo en este aspecto, algo que confirmé mediante el repositorio institucional de la universidad realizando búsquedas de palabras clave como “kernel”, “núcleo”, “UEFI”. La universidad oferta la asignatura *21012 (Sistemas Operativos)*, impartida por el departamento de “Tecnología Informática y Computación” en el área de “Arquitectura y Tecnología de Computadores”. Esta difiere en bibliografía de lo que otras universidades ofertan relacionado con los sistemas operativos, en este caso el contenido práctico se basa en el uso de *syscalls* que ofrece el kernel de Linux, sin abarcar el desarrollo o modificación de un sistema a bajo nivel, donde encontramos dificultades y retos nuevos programando.

Mi interés por el área del desarrollo de sistemas operativos y núcleos, junto con mi inclinación por el desarrollo de software de bajo nivel pueden ayudar a otros estudiantes a

introducirse en el área mediante este Trabajo Final de Grado (TFG). Considero que la universidad también puede beneficiarse al incorporar trabajos relacionados con este área.

Finalmente, el desarrollo de un núcleo es una tarea que requiere consultar mucha documentación técnica, a parte de documentar el código es necesario tener esquemas del funcionamiento interno de cada aspecto del núcleo y como se relaciona con las interfaces que nos proporciona la CPU para trabajar con ella directamente. Es por esto que desarrollar un TFG creo que puede ser beneficioso para mí, aumentando mi comprensión de mi propio sistema y teniendo cada detalle plasmado en un único documento, consultable por mí o cualquier desarrollador que se embarque en la misma aventura.

1.2. Objetivos

El objetivo del TFG es la creación de un núcleo académico, de usabilidad muy limitada o nula, con el fin de documentar el proceso de desarrollo y las funcionalidades de CPUs con arquitecturas *x86-64* relacionadas con el desarrollo del núcleo. El desarrollo del núcleo será en el lenguaje *C++* y en una máquina con arranque UEFI mediante un bootloader escrito en *C*. El objetivo de estas dos decisiones es modernizar la escasa documentación que hay, donde se utiliza el lenguaje *C* para el kernel, ensamblador para el *bootloader* y arranque Basic Input/Output System (BIOS).

Al establecer el arranque mediante UEFI obtenemos una “interfaz” y un “ecosistema” preparado por UEFI que nos evita tener que pasar por el modo de compatibilidad de CPUs *16 Bit Real Mode* y realizar llamadas a la BIOS mediante interrupciones. Esta decisión ayuda a centrarme en el objetivo que es el desarrollo y funcionamiento interno del núcleo, mediante la flexibilidad y abstracción de UEFI.

Respecto del uso de bootloaders ya existentes como GRand Unified Bootloader (GRUB) mediante el uso de *Multiboot*, ha quedado descartado de los objetivos del proyecto, puesto que proporcionaría demasiada abstracción sobre el *bootloader* puesto que no hay que desarrollarlo, impidiendo así aprender sobre las funcionalidades básicas de un *bootloader* y su integración con el núcleo.

También se tiene como objetivo aprender e implementar sistemas automatizados de construcción de proyectos tales como *cmake* y *make*, con el fin de construir de forma automática el proyecto y la *toolchain* necesaria para este. Además, se hará uso del Version Control System (VCS) “Git” con el objetivo de aprender buenas prácticas con el mismo y profundizar en su funcionamiento para proyectos más complejos, para el alojamiento el proyecto se configurarán tres repositorios remotos, *Github*, *Gitlab* y un *home server* con el objetivo de aprender a trabajar con distintos tipos de repositorios remotos.

Finalmente, puesto que es necesario gestionar y mantener una base de código relativamente grande de *C/C++* he establecido como otro objetivo del TFG aprender a usar y configurar herramientas de análisis de código como “loc” y herramientas de formateado y estilo de código

Se puede escribir mejor

Definir que es multiboot

como “clang-format”.

2. Marco Teórico

3. Metodología

4. Toolchain

En Ingeniería Informática, la *toolchain* es el conjunto de herramientas de programación usadas para llevar a cabo complejas tareas de desarrollo de software o de construcción de software.

El conjunto de herramientas necesarias para la construcción de alma está en la carpeta `toolchain`, donde podemos encontrar un `Makefile` para su construcción automática a partir de los fuentes. Los fuentes se descargan mediante los submódulos de git:

```
⌚ Código 4.1: Submódulos git
1 [submodule "toolchain posix-uefi"]
2   path = toolchain/posix-uefi
3   url = https://gitlab.com/bztsrc/posix-uefi.git
4   ignore = all
5 [submodule "toolchain edk2"]
6   path = toolchain/edk2
7   url = https://github.com/tianocore/edk2.git
8   ignore = all
9 [submodule "toolchain binutils-gdb"]
10  path = toolchain/binutils-gdb
11  url = git://sourceware.org/git/binutils-gdb.git
12  ignore = all
13 [submodule "toolchain gcc"]
14  path = toolchain/gcc
15  url = https://github.com/gcc-mirror/gcc
16  ignore = all
```

Para construir la toolchain de alma y poder realizar construcciones del kernel es necesario instalar en el sistema operativo las dependencias de construcción de la toolchain (véase sec. 4.3). Una vez se hayan instalado dichas dependencias se puede construir la toolchain de la siguiente manera:

```
⌚ Código 4.2: Construcción de la toolchain de alma
1 make -C toolchain
```

Una vez se haya finalizado la construcción de la toolchain, el sistema de construcción de alma reconocerá las herramientas construidas y las utilizará ningún cambio. La instalación de la toolchain se realiza en `toolchain/build` y no se instala en el sistema¹.

¹Se tomó esta decisión con el fin de mantener el sistema host limpio

4.1. Alma Build VM

Puesto que el proceso de construcción de la toolchain es un proceso lento y costoso, para este proyecto se ha diseñado una máquina virtual específica con toda la toolchain precompilada, la “Alma Build VM”.



Figura 4.1: Alma Build VM

La máquina virtual es una versión modificada de Xubuntu 20.04² con la toolchain de alma precompilada, VSCodium preconfigurado para poder editar y compilar el proyecto y `ccache` [11] a nivel de sistema. La máquina se distribuye en formato Open Virtualization Format (OVA) con un peso de 5.9G.

Se puede descargar desde <https://github.com/ecomaikgolf/alma#virtual-machine> e importar con el siguiente comando en virtualbox:

```
⌚ Código 4.3: Importar .ova de alma build VM
1 vboxmanage import alma.ova
```

Ahora se puede arrancar la máquina con:

```
⌚ Código 4.4: Arrancar alma build VM
1 vboxmanage startvm "alma"
```

²<https://virtual-machines.github.io/Xubuntu-VirtualBox/>

4.2. Dependencias de Ejecución

Para poder ejecutar el proyecto a partir de una build precompilada por otra máquina es necesario el siguiente software. Los paquetes se han probado en **Ubuntu 20.04** y se ha replicado en **Arch Linux**.

- qemu
- qemu-system-x86

Q Código 4.5: Instalación en Ubuntu 20.04

```
1 apt install qemu-system-x86 qemu-system-gui
```

Q Código 4.6: Instalación en ArchLinux

```
1 pacman -S qemu qemu-arch-extra
```

Ahora se puede descargar y ejecutar una build precompilada de alma:

Q Código 4.7: Ejecución de build precompilada de alma

```
1 cd qemu-fs
2 wget https://ls.ecomaikgolf.com/alma/builds/bios.bin
3 wget https://ls.ecomaikgolf.com/alma/builds/20211012-e906c4c-m.tar.gz
4 tar xf 20211012-e906c4c-m.tar.gz
5 make
```

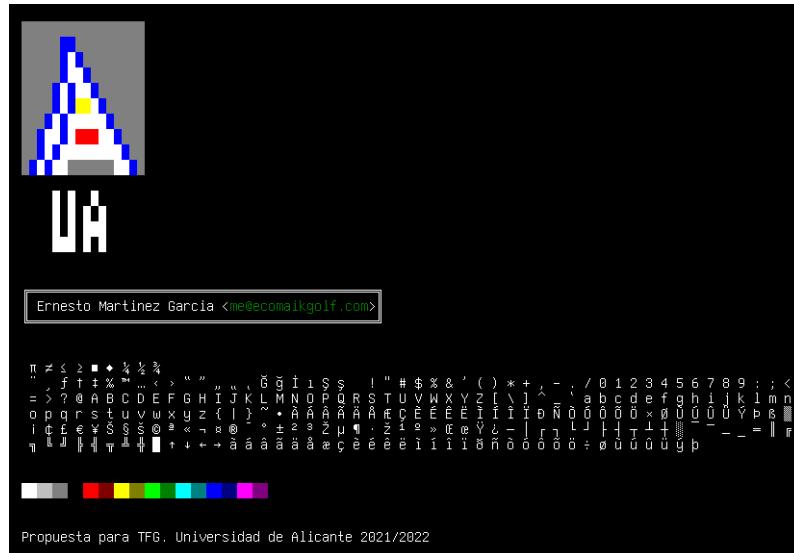


Figura 4.2: alma 20211012-e906c4c-m build

Se pueden encontrar otras builds en <https://ls.ecomaikgolf.com/alma/builds/>, el nombre del fichero corresponde a **fecha-commit-[m].tar.gz**, la **m** indica si ha recibido modificaciones de estilo como prints para que sea más visual.

4.3. Dependencias de Construcción

Para construir y trabajar con alma es necesario tener instalados estos paquetes en el sistema operativo donde se va a desarrollar el kernel. Los paquetes se han probado en **Ubuntu 20.04** y se ha replicado en **Arch Linux**.

- nasm
- iasl
- cmake
- qemu-system-x86
- qemu-system-gui
- uuid-dev
- python
- python3-distutils
- texinfo
- bison
- flex
- mtools

© Código 4.8: Instalación en Ubuntu 20.04

```
1 apt install nasm iasl cmake make qemu-system-x86 qemu-system-gui git uuid-dev ↵
   ↵ python python3-distutils bash texinfo bison flex build-essential mtools
```

Este listado de paquetes lo componen herramientas necesarias para la compilación de alma y herramientas necesarias para la construcción de la toolchain de alma.

4.3.1. GNU Compiler Collection (gcc)

El desarrollo de un kernel empieza por la selección del software que se va a utilizar para compilarlo, esto no es una tarea simple como seleccionar **gcc**, el desarrollo de software a bajo nivel como es un kernel comunmente requiere de un *cross compiler*, un compilador capaz de crear código ejecutable para una plataforma distinta a la que está usando para compilar.

El sistema de construcción de GNU define un concepto denominado “Target Triplets” que describe la plataforma en la que se ejecutará el código que vamos a compilar [12]. El triplete está compuesto por tres³ campos:

- Nombre del modelo/familia de la CPU (eg. `x86_64`)
- Vendor (eg. `linux`)
- Sistema Operativo (eg. `gnu`)

y se suele dar en el siguiente formato `machine-vendor-operatingsystem`. Se puede consultar el triplete de tu compilador GNU ejecutando la instrucción `gcc -dumpmachine`.

En mi caso el triplete es `x86_64-pc-linux-gnu`, por lo que aunque mi kernel se vaya a ejecutar en la misma arquitectura (`x86_64`) el compilador no generaría código correcto, puede darse el caso de que nuestro compilador genere código que solo puede ejecutarse bajo sistemas operativos con kernel linux.

³No siempre es obligatorio que aparezcan los tres

Es evidente que necesitamos de un compilador que tenga un “Target triplet” genérico que no haga asunciones de donde se va a ejecutar el código máquina generado. El triplete que se necesita es `x86_64-elf`, puesto que el formato del binario a generar queremos que sea Executable and Linkable Format (ELF) y la arquitectura `x86_64`, no podemos hacer más asunciones sobre el entorno de ejecución.

Además de los tripletes, el compilador ha de cumplir otro requisito para no encontrarnos con problemas a la hora de desarrollar código a bajo nivel es que tenemos que desactivar el uso de las `red-zone` por parte del compilador.

La `red-zone` es [13] una zona de 128 bytes de longitud situada debajo del stack pointer (véase fig. 4.3a) de libre uso para el compilador, sin necesidad de “notificarlo” a la aplicación, el sistema operativo o a las interrupciones en ejecución. Esto es una funcionalidad especificada en el Application Binary Interface (ABI) de `x86_64` pero que puede romper nuestro kernel.

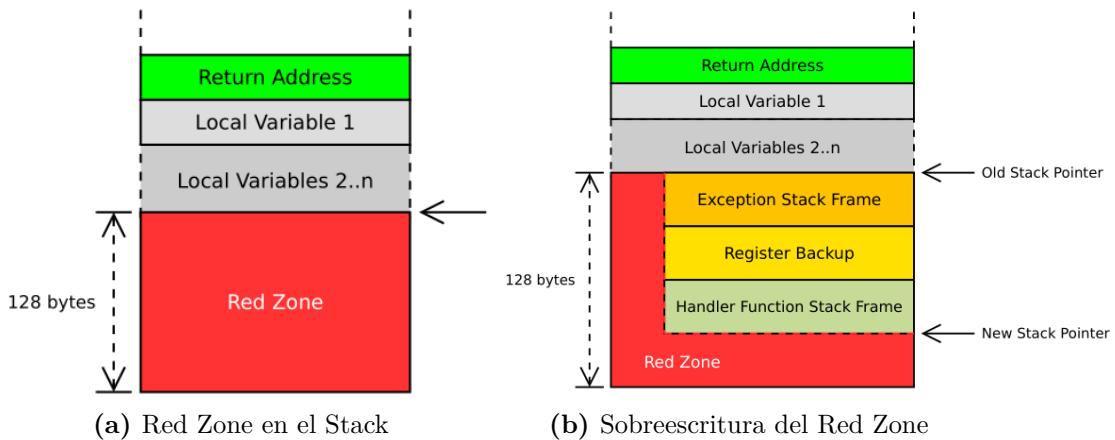


Figura 4.3: Red Zone en el Stack (<https://os.phil-opp.com>)

Para las aplicaciones de usuario no tiene ningún problema puesto que el propio compilador sabe cuando se va a necesitar decrementar el stack pointer para agrandar el stack⁴, pero si el programa en ejecución es sorprendido por una `interrupt routine`⁵ se agrandará el stack obligatoriamente y sin que el compilador lo pueda predecir, por lo que es posible que la red zone se sobreesciba y provoque *undefined behaviour* como podemos ver en la figura 4.3b.

Pese a que el uso de las red zones por parte del compilador lo podemos desactivar con la flag `-mno-red-zone`, el compilador utiliza en nuestro código de forma automática una librería denominada `libgcc` que de normal viene compilada con la red zone activada. Para desactivar la red zone en el código de `libgcc` es necesario compilarla con una flag especial para que si nuestro compilador al compilar el kernel genera llamadas a `libgcc`, no se asuma que hay un espacio de 128 bytes pasado el stack pointer.

⁴El stack en la ABI de System V crece “hacia abajo” [14]

⁵Explicado en

Debido a todos los problemas que presenta un gcc común, hemos de compilarlo nosotros manualmente para cambiar aquellos aspectos que nos imposibilitan el desarrollo de un kernel con él. En `toolchain/makefile` se ha escrito un target `gcc-build` que se encarga de automatizar esta misma tarea.

```
① Código 4.9: Compilación de gcc 11.x
1 gcc-build:
2   $(eval CONTENT = tmake_file="$$\{tmake_file\} i386/t-x86_64-elf")
3   $(eval NUMBER = $($shell grep -F x86_64-*-elf -n $(CURDIR)/gcc/gcc/config.gcc ↵
4     ↵ | cut -f1 -d:))
5   cd gcc; \
6   git checkout $(gcc_v); \
7   ./contrib/download_prerequisites; \
8   echo -e "MULTILIB_OPTIONS += mno-red-zone\nMULTILIB_DIRNAMES += no-red-zone" ↵
9     ↵ > gcc/config/i386/t-x86_64-elf; \
10  sed '$(NUMBER) a $(CONTENT)' gcc/config.gcc; \
11  mkdir build; \
12  cd build; \
13  export PATH=$(BUILD_DIR_TOOLCHAIN)/bin/:$$PATH; \
14  ../configure --target=x86_64-elf --prefix="$(BUILD_DIR_TOOLCHAIN)" --disable-↪
15    ↵ nls --enable-languages=c,c++ --without-headers; \
16  make -j$(THREADS) all-gcc; \
17  make -j$(THREADS) all-target-libgcc; \
18  make install-gcc; \
19  make install-target-libgcc;
```

El script se mete a la carpeta del VCS de `gcc` en la línea 4 y hace un checkout a una versión pre-especificada, concretamente a `releases/gcc-11` por lo que el compilador construido será `gcc 11.x`. Acto seguido descarga los requisitos necesarios para construir `gcc` en la línea 6 y en la 7 cambia la configuración de `libgcc` para que se construya sin soporte para la red-zone.

Finalmente se crea un directorio de construcción y se configura la build de `gcc` con el `configure` generado por `autoconf`, en la línea 12 se especifica el triplete del target, se indica donde queremos que se instale, activamos solo el lenguaje C/C++ y especificamos con `--without-headers` que `gcc` no puede usar la librería estándar de C, la opción `--disable-nls` desactiva el soporte para Internacionalización (I18N).

El compilador construido se instala en `toolchain/build/toolchain` y el sistema de construcción de alma lo usará automáticamente para las construcciones del bootloader y del kernel.

4.3.2. GNU Binutils

Las `binutils` son una colección de herramientas del proyecto GNU necesarias para compilar `gcc`, por lo que antes de compilar `gcc` el makefile de la toolchain de alma descarga y compila las `binutils 2.37`.

Q Código 4.10: Compilación de binutils 2.37

```

1 binutils-build:
2   cd binutils-gdb; \
3   git checkout $(binutils_v); \
4   mkdir build; \
5   cd build; \
6   ../configure --target=$(TARGET_BINUTILS) --enable-targets=all --disable-gdb ↪
      ↪ --disable-libdecnumber --disable-readline --disable-sim --prefix="$(  

      ↪ BUILD_DIR_TOOLCHAIN)" --with-sysroot --disable-nls --disable-werror; \
7   make -j$(THREADS); \
8   make install;

```

Las binutils se configuran con ciertos parámetros para desactivar funcionalidades que no necesitamos a fin de acelerar su proceso de compilación, finalmente se instala en la carpeta `toolchain/build/toolchain` para que el makefile de construcción de la toolchain pueda compilar el resto de dependencias.

4.3.3. EDK II OVMF

Para poder usar UEFI en `qemu` es necesario proporcionarle a `qemu` mediante el parámetro `-bios` un archivo Open Virtual Machine Firmware (OVMF) con el firmware UEFI.

El archivo OVMF es parte de la toolchain de alma y se compila⁶ partiendo del código fuente del EDK II [2].

Q Código 4.11: Compilación de EDK II OVMF

```

1 $(BUILD_DIR_UEFI)/bios.bin:
2   cd edk2; \
3   git checkout $(edk2_v); \
4   make CC=$(COMPILER_EDK2) -j$(THREADS) -C BaseTools; \
5   source ./edksetup.sh; \
6   sed -i -e "s@= EmulatorPkg/EmulatorPkg.dsc@= $(PLATFORM)@" Conf/target.txt; \
7   sed -i -e "s@= DEBUG@= $(TARGET_EDK2)@" Conf/target.txt; \
8   sed -i -e "s@= IA32@= $(ARCH)@" Conf/target.txt; \
9   sed -i -e "s@= VS2015x86@= $(TOOLCHAIN)@" Conf/target.txt; \
10  CC=$(COMPILER_EDK2) build; \
11  cp ./Build/$(PLATFORM_NAME)/$(TARGET_EDK2)_$(TOOLCHAIN)/FV/OVMF.fd $(  

      ↪ BUILD_DIR_UEFI)/$(BIOS_FILE);

```

La compilación utiliza la versión de EDK II `stable/202011`. En primer lugar se construyen las `BaseTools` de EDK II en la línea 4 y en la 5 se entra en el entorno de construcción. En las líneas 6-9 se configura el OVMF a generar, en nuestro caso se construye en modo *Release* y con el target `x86_64-elf`, finalmente se construye el OVMF y se copia a `toolchain/build/uefi`.

⁶<https://github.com/tianocore/edk2/blob/master/OvmfPkg/README>

4.3.4. posix-uefi

Para escribir el bootloader es necesaria la librería `posix-uefi` [3] que como ya se ha explicado, provee una API POSIX sobre la librería `gnu-efi`, lo que nos permite escribir aplicaciones para UEFI usando funciones POSIX.

Q Código 4.12: Compilación de EDK II OVMF

```
1 posix-uefi-build:  
2   cd posix-uefi; \  
3   git checkout ${posix-uefi_v}; \  
4   export PATH=$(BUILD_DIR_TOOLCHAIN)/bin/:$$PATH; \  
5   USE_GCC=$(USE_GCC) make -C uefi; \  
6   ln -s $(CURDIR)/build/uefi $(BUILD_DIR_POSIX_UEFI);
```

El `Makefile` de construcción de la toolchain de alma construye `posix-uefi` en la versión especificada por el commit 1431b... con el compilador `gcc` anteriormente construido. Finalmente sitúa en `toolchain/build/posix-uefi` un enlace simbólico a la carpeta con la librería compilada.

Se ha modificado la forma de la que se utiliza `posix-uefi`, se ha integrado en el sistema de construcción de alma y se ha descartado el `Makefile` que provee el proyecto. Esto mejora el sistema de construcción y reduce las dependencias con makefiles no propios, esto se explicará en

Poner referencia a la construcción del bootloader

5. Construcción del Proyecto

En este capítulo se presentan

6. Bootloader

6.1. Introducción

Como norma general todo sistema operativo es cargado por un bootloader al arrancar la máquina. Cuando arrancamos un disco que tiene Linux instalado lo que ejecutamos inicialmente (quitando el firmware) no es Linux, sino GRUB (Figura 6.1a), el bootloader más famoso. GRUB tiene la responsabilidad de presentarle a Linux un entorno técnico concreto y pasarle cierta información sobre la máquina donde se está ejecutando. En cuanto al usuario, GRUB ofrece funcionalidades como la pantalla de selección, el editado de entradas y el bloqueo por contraseña.



Figura 6.1: Ejemplos de bootloaders

Lo mismo pasa en plataformas antagónicas como puede ser Windows, si queremos arrancar desde un disco externo un sistema operativo Windows 7, al arrancar desde la BIOS ese disco lo primero que se ejecutará será el Windows Boot Manager (Figura 6.1b) el cual dará paso a nuestro Windows 7. Otras como Mac OSX disponen de BootX¹, FreeBSD de “Boot Manager”², etc.

Como podemos observar, plantearnos el desarrollo de un kernel suscita la pregunta de cómo se va arrancar, quién va a ser el responsable de pasar el contenido binario de nuestro kernel

¹<https://web.archive.org/web/20070309142504/http://www.cs.rpi.edu/~gerbal/BootX.pdf>

²<https://people.freebsd.org/~trhodes/doc/handbook/boot-blocks.html>

desde el **.iso**³ a la memoria RAM, y cómo se le va a dar el control de la máquina. Para responder esta pregunta tenemos que conocer primero si vamos a arrancar de un entorno BIOS o EFI (UEFI).

BIOS (Basic Input/Output System) fue creado para ofrecer servicios de bajo nivel a programadores de sistemas, su intención era ocultar al máximo las variaciones entre modelos de ordenadores, con el fin de ofrecer un marco común a los programadores de sistemas. La BIOS se encuentra en la ROM⁴ y la CPU apunta a ella al arrancarse mediante el registro de instrucción IP. En primer lugar la BIOS realiza un checkeo denominado POST (Power-on self-test) para comprobar que el hardware funciona correctamente⁵, si el hardware está bien procede a enumerar los dispositivos instalados a lo largo de nuestra placa base, finalmente busca dispositivos arrancables (offset byte 510 a 0x55 y 511 a 0xAA) y carga sus 512 bytes (tamaño de un sector) en la dirección 0x0:0x7c000⁶.

(U)EFI (Unified Extensible Firmware Interface) es una especificación para plataformas x86, x86-64, ARM y Itanium que define una interfaz entre el sistema operativo y el firmware de la plataforma. EFI se desarrolló originalmente en el 1990 por Intel para plataformas Itanium, el proyecto derivó en 2005 en lo que se conoce como UEFI, conformado por varias empresas tecnológicas como AMD, Apple, Intel y Microsoft. UEFI tiene un proceso de arranque muy complejo (Figura 6.2) pero que se basa en el arranque UEFI mencionado anteriormente, además provee retrocompatibilidad con BIOS. La finalidad de UEFI era ofrecer una interfaz estandarizada y mucho más cómoda a los recursos del sistema. Usar arranques UEFI puede no ser compatible con todos los sistemas vistos, a veces es necesario arrancarlos en modo BIOS o pueden presentar problemas. Podemos establecer una serie de ventajas y desventajas del uso de UEFI respecto de BIOS:

↑ Ventajas	↓ Desventajas
<ul style="list-style-type: none"> Estado preconfigurado Comodidad de desarrollo Estandarización 	<ul style="list-style-type: none"> Menor compatibilidad Menor control de la máquina No tan bajo nivel

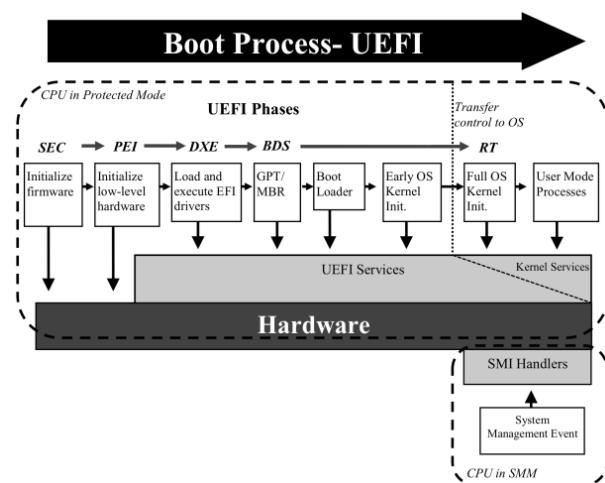


Figura 6.2: Arranque UEFI

³Se ha utilizado como ejemplo, no tiene por que ser un iso

⁴Actualmente se utiliza memoria flash para poder actualizarla.

⁵En caso contrario **suele** emitir pitidos que significan un código de error

⁶Segmento 0, dirección 0x7c000

Si estamos desarrollando un bootloader para nuestro sistema tendremos que tener en cuenta si queremos ejecutarlo bajo el paraguas de U(EFI) o BIOS, aunque es posible dar soporte para ambos es normal encontrarse en la tesitura de elegir un camino inicial. Trabajar con BIOS nos daría la garantía de poder ejecutarlo prácticamente en todas las máquinas que nos encontremos, puesto que si tienen BIOS tendremos la certeza de que lo podremos ejecutar pero si tienen UEFI también, mediante su capa de retrocompatibilidad⁷. Si se trabaja con UEFI tendremos gran parte del trabajo hecho, utilizaremos un entorno más moderno que nos proporciona un estado de la máquina más avanzado (por ejemplo, configura una memoria virtual “identity mapped”⁸) y nos provee de los determinados “Servicios UEFI”, con los que podemos trabajar en estados muy tempranos de arranque de forma muy cómoda. Por otro lado, si usamos UEFI estamos expuestos a que la placa base de la máquina donde queramos ejecutar nuestro sistema disponga solo de BIOS, en ese caso no podríamos arrancar nuestro sistema.

Arranque sin bootloader con EFI

Ciertos sistemas soportan el arranque directo sin necesidad de un bootloader, es el caso de Linux desde la versión 3.3 que bajo el paraguas de EFI puede ser cargado directamente por el firmware.

La técnica que se utiliza se denomina “EFISTUB”, podemos activarlo en linux compilando el kernel con el parámetro `CONFIG_EFI_STUB=y`. Se puede leer más en <https://www.kernel.org/doc/html/latest/admin-guide/efi-stub.html>

Ante la decisión de escribir un bootloader bajo EFI o BIOS encontramos otra solución, no usar ni uno ni otro, no implementar un bootloader. Es una decisión bastante coherente si tu objetivo no era desarrollar un bootloader, existen implementaciones de distintos bootloaders como puede ser GRUB2 o Limine que proveen de una “interfaz” para comunicarte con ellos desde el kernel, mediante esa interfaz nuestro sistema se comunica de una forma estandarizada con el bootloader y de esta manera no tenemos que implementar uno, además podemos usar cualquiera que siga la misma interfaz.

Por ejemplo, si queremos desarrollar un sistema que sea capaz de arrancar mediante GRUB tendremos que seguir el estándar “Multiboot”⁹. Con nuestro sistema implementado siguiendo el estándar Multiboot podremos descargarnos GRUB y arrancarlo. Como estamos siguiendo un estándar podemos utilizar otros bootloaders que también lo sigan como es el caso de SYSLINUX¹⁰, Limine y otros.

⁷Si se ha trasteado con UEFI se puede ver que al arrancar desde un dispositivo te deja hacerlo mediante “Legacy Boot” (BIOS) o UEFI

⁸La dirección virtual y la física es la misma

⁹<https://www.gnu.org/software/grub/manual/multiboot/multiboot.html>

¹⁰<https://en.wikipedia.org/wiki/SYSLINUX>

6.2. Estado del Arte

En la actualidad existen una multitud de bootloaders ya escritos que permiten arrancar nuestros sistemas favoritos, también resultan una alternativa a desarrollar uno propio a la hora de desarrollar un kernel.

Bootloader	Arquitectura	Estándar/Formato
GRUB Legacy	x86 (PC)	Multiboot 1, Linux zImage, Linux bzImage, etc.
GRUB2	x86 (PC, EFI) IA-64, ARM (U-Boot, UEFI), PowerPC	Multiboot, etc.
LILO	x86 (PC)	Linux zImage, Linux bzImage
SYSLINUX	x86 (PC)	Linux zImage, Linux bzImage, Multiboot MBR
Yaboot	PowerPC	Linux ELF image
Das U-Boot	PowerPC, ARM, RISC-V, x86, MIPS	EFI, ELF, U-Boot, Linux zImage
Windows Boot Manager	x86 (PC), ARM (some)	PE
Limine	x86 (PC)	Multiboot 1 y 2, Stivale 1 y 2, Linux zImage and bzImage
NTLDR	x86 (PC)	Windows NT kernel (PE)

Tabla 6.1: Bootloaders más comunes [10]

Dentro de los más comunes encontramos GRUB2, desarrollado por GNU y el sucesor de GRUB 0.9x (Legacy), suele ser la opción por defecto que encontramos en la mayoría de distros Linux. También encontramos el bootloader LILO (Figura 6.3a), que es la opción elegida por la Escuela Politécnica Superior para sus ordenadores como podemos comprobar en los laboratorios al arrancar. Windows Boot Manager no tiene tanto reconocimiento puesto que siempre suele ser reemplazado en términos generales por “Windows”, la mayoría de desarrolladores ajenos al campo técnico de Windows no suelen hacer una diferenciación entre Windows y su bootloader.



Figura 6.3: Bootloaders comunes

Si el desarrollador se decantase por utilizar un bootloader que siga la especificación “stivale” o “stivale2” dispondría de las cabeceras en formato .h especificando las estructuras especificadas. La especificación y las estructuras se pueden encontrar en <https://github.com/stivale/stivale>, solo hay que incluirlas en un proyecto C/C++ o portearlas a otros lenguajes para poder empezar a desarrollar un sistema que siga la especificación de stivale.

Con el sistema desarrollado encontramos varios bootloaders que siguen “stivale2”, pero el principal y el que sirve como guía para la especificación es limine <https://github.com/limine-bootloader/limine>. El desarrollador provee de ramas git donde se publican versiones compiladas del bootloader que podríamos usar directamente <https://github.com/limine-bootloader/limine/tree/v2.0-branch-binary>.

Generación de un .iso

Con los binarios de limine y la herramienta externa xorriso podemos crear fácilmente un archivo .iso arrancable con UEFI y BIOS para ejecutar nuestro sistema.

```
$ xorriso -as mkisofs -b limine-cd.bin \
    -no-emul-boot -boot-load-size 4 -boot-info-table \
    --efi-boot limine-eltorito-efi.bin \
    -efi-boot-part --efi-boot-image --protective-msdos-label \
    iso_root -o image.iso
```

```
$ ./limine/limine-install image.iso
```

Luego podemos flashearlo a un usb mediante:

```
$ dd bs=4M if=image.iso of=/dev/sdX conv=fdatasync status=progress
```

Y ejecutarlo con arranque UEFI o BIOS.

En caso de que se buscasen desarrollos completos del bootloader hay dos grandes librerías que podemos utilizar para desarrollarlo bajo un entorno EFI: gnu-efi y posix-uefi.

gnu-efi (<https://sourceforge.net/projects/gnu-efi/>) es un entorno de desarrollo ligero para aplicaciones UEFI, es una biblioteca que nos permite interactuar con los componentes de UEFI haciendo llamadas a los servicios de UEFI y usando sus estructuras de datos. A pesar de ser “únicamente” una librería, para que el binario resultante pueda ser una aplicación EFI válida tenemos que manejar completamente el proceso de compilación y enlazado, además de modificar el objeto resultante a fin de obtener un binario PE (Portable Executable).

La sintaxis a la hora de trabajar con gnu-efi puede parecer un poco tosca (Código 6.1), es por eso que han surgido alternativas como posix-uefi que sobre esa capa tosca y dura de gnu-efi proporcionan una interfaz POSIX a la que se está más acostumbrado (Código 6.2). Por lo tanto posix-uefi “únicamente” actúa como capa por encima de gnu-efi, internamente son lo mismo y desde posix-uefi se puede acceder a gnu-efi.

© Código 6.1: Sintaxis de gnu-efi

```

1 #include <efi.h>
2 #include <efilib.h>
3
4 EFI_STATUS efi_main(EFI_HANDLE ImageHandle, EFI_SYSTEM_TABLE *SystemTable)
5 {
6     EFI_STATUS Status;
7     EFI_INPUT_KEY Key;
8
9     ST = SystemTable;
10
11    Status = ST->ConOut->OutputString(ST->ConOut, L"Hello World\r\n");
12
13    if (EFI_ERROR(Status))
14        return Status;
15
16    Status = ST->ConIn->Reset(ST->ConIn, FALSE);
17
18    if (EFI_ERROR(Status))
19        return Status;
20
21    while ((Status = ST->ConIn->ReadKeyStroke(ST->ConIn, &Key)) == EFI_NOT_READY) ;
22
23    return Status;
24 }
```

© Código 6.2: Sintaxis de posix-efi

```

1 #include <uefi.h>
2
3 int main (int argc, char **argv)
4 {
5     void *aux = malloc(sizeof(...));
6     FILE *f = fopen("...");  

7     printf("Hello, world!\n");
8     return 0;
9 }
```

Como podemos ver tenemos la posibilidad de desarrollar un bootloader como aplicación EFI mediante varias interfaces según nuestros requisitos o gustos personales. Hay que tener en cuenta que **posix-uefi** no abarca todas las funcionalidades de EFI porque no hay llamadas estándar POSIX para ellas, así que se tendrá que lidiar con las estructuras internas de **gnu-efi** que lleva **posix-uefi**. Un ejemplo de esto puede ser obtener la tabla de configuración de EFI, puesto que no hay llamada POSIX para ello hay que hacer **ST->ConfigurationTable**, usando así la parte de **gnu-efi**.

⚠ Consideraciones de Desarrollo

- Las aplicaciones EFI utilizan unicode.
- Por lo general no puedes estar en ejecución más de 5 minutos sin salirte de los servicios UEFI (o sin desactivar el contador), la máquina se reinicia porque cree que ha habido algún error.
- El bootloader a ejecutar de forma automática será ::/EFI/BOOT/BOOTX64.EFI

6.3. Sistema de Construcción

Compilar y enlazar un archivo con tantos requisitos es una tarea tediosa que requiere de una sección entera para explicarse. Por suerte `posix-uefi` incluye un archivo `Makefile` que permite compilar de forma automática y muy simple (Código 6.3), por desgracia no es el método que vamos a usar puesto que estamos usando `cmake` como herramienta de construcción de proyecto, pero era el que usaba en commits antiguos `b3b0d36`.

© Código 6.3: Makefile básico de `posix-efi`

```

1 ARCH=x86_64
2 TARGET = bootloader.efi
3 USE_GCC = 1
4 CFLAGS=
5 LDFLAGS=
6 LIBS= #static only .a
7
8 BUILDDIR=../build
9
10 bootloader: all
11   mv bootloader.efi $(BUILDDIR)
12   mv bootloader.o $(BUILDDIR)
13
14 include uefi/Makefile

```

En commits más modernos, cuando se migró todo el sistema de construcción a `cmake` se movió también el sistema de construcción del bootloader.

© Código 6.4: `posix-efi cmake I`

```

1 cmake_minimum_required(VERSION 3.16)
2
3 project(alma-bootloader C)
4
5 set(POSIXUEFI_DIR "${CMAKE_CURRENT_SOURCE_DIR}/../toolchain/posix-uefi/uefi")
6
7 find_program(CCACHE_FOUND ccache)
8 if(CCACHE_FOUND)
9   set_property(GLOBAL PROPERTY RULE_LAUNCH_COMPILE ccache)
10  set_property(GLOBAL PROPERTY RULE_LAUNCH_LINK ccache)
11 endif(CCACHE_FOUND)

```

En primer lugar se establece la versión mínima de `cmake` para poder ejecutar el sistema de construcción, se establece a la 3.16 puesto que es la versión por defecto instalada en Ubuntu 20.04 y en los laboratorios de la Escuela. Acto seguido se establece el nombre del proyecto y que es en C. Después se le indica donde están los archivos de `posix-uefi` y finalmente si se encuentra `ccache` en la máquina host se activa el cacheo de compilaciones a fin de acelerar la construcción.

© Código 6.5: `posix-efi cmake II`

```

12 set(CMAKE_C_COMPILER "${TOOLCHAINBIN}/x86_64-elf-gcc")
13 set(CMAKE_CXX_COMPILER "${TOOLCHAINBIN}/x86_64-elf-g++")
14 set(CMAKE_LINKER "${TOOLCHAINBIN}/x86_64-elf-ld")
15 set(CMAKE_OBJCOPY "${TOOLCHAINBIN}/x86_64-elf-objcopy")
16

```

```

17 set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -ffreestanding")
18 set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -fshort-wchar")
19 set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} --ansi")
20 set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -fno-stack-protector")
21 set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -fno-stack-check")
22 set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -fno-strict-aliasing")
23 set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -DHAVE_USE_MS_ABI")
24 set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -D__x86_64__")
25 set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -mno-red-zone")
26 set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -Wno-builtin-declaration-mismatch")
27 set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -fpic")
28 set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -fPIC")
29 set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -Wl,--defsym=_DYNAMIC=0")
30 set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -Wall")
31 set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -Wextra")
32
33 set(CMAKE_C_STANDARD 11)
34 set(CMAKE_C_STANDARD_REQUIRED True)

```

En el Código 6.5 se establecen los compiladores, enlazadores y binarios auxiliares a utilizar. En este caso seleccionamos los compilados por al toolchain con target `x86_64-elf`. Acto seguido establecemos flags de compilación como indicar que estamos en un entorno ffreestanding y no hosteado, que no establezca canarios en el stack y que no tenga red zone (Figura 4.3a). Finalmente establecemos un estándar mínimo de C requerido, si no se puede compilar con C11 por algún motivo no se intentará construir.

--ffreestanding

La opción de compilación `--ffreestanding` indica al compilador que el entorno donde se va a ejecutar el código no tiene un sistema “por detrás” de apoyo. Podemos comprobar el sistema en el que estamos mediante la macro `__STDC_HOSTED__` (1 si está hospedado, 0 si no).

Al estar en un entorno ffreestanding o hosted los requerimientos para ciertos aspectos del lenguaje y de la librería cambian. Por ejemplo en un entorno ffreestanding las funciones de inicio y fin (donde se llama a los constructores globales) son implementation defined, requerir una función main es implementation defined, etc.

Código 6.6: posix-efi cmake III

```

35 set(LINKER_SCRIPT "${POSIXUEFI_DIR}/elf_x86_64_efi.lds")
36
37 SET(CMAKE_EXE_LINKER_FLAGS "${CMAKE_EXE_LINKER_FLAGS} -nostdlib")
38 SET(CMAKE_EXE_LINKER_FLAGS "${CMAKE_EXE_LINKER_FLAGS} -shared")
39 SET(CMAKE_EXE_LINKER_FLAGS "${CMAKE_EXE_LINKER_FLAGS} -Bsymbolic")
40 SET(CMAKE_EXE_LINKER_FLAGS "${CMAKE_EXE_LINKER_FLAGS} -Luefi")
41 set(CMAKE_EXE_LINKER_FLAGS "${CMAKE_EXE_LINKER_FLAGS} -T ${LINKER_SCRIPT}")

```

Aquí vemos como se establecen opciones de enlazado necesarias para enlazar el bootloader, la más notable es el uso de un script de enlazado que nos da `posix-uefi`, también establecemos `-Luefi` para enlazar con la librería. Establecemos `-nostdlib` para indicarle que no tenemos librería estándar y no se genere en ningún momento nada relacionado con esta.

Q Código 6.7: posix-efi cmake IV

```

42 set(UEFI_HEADER_DIR
43   ${POSIXUEFI_DIR}
44 )
45 set(EFI_HEADER_DIR
46   /usr/include
47   /usr/include/efi
48   /usr/include/efi/protocol
49   /usr/include/efi/x86_64
50 )
51 set(BOOTLOADER_HEADER_DIR
52   lib
53 )
54 set(INCLUDE_DIRECTORIES
55   ${UEFI_HEADER_DIR}
56   ${EFI_HEADER_DIR}
57   ${BOOTLOADER_HEADER_DIR}
58 )
59
60 include_directories(${INCLUDE_DIRECTORIES})

```

Con el Código 6.7 establecemos los directorios en los cuales el compilador va a buscar las cabeceras incluidas en los fuentes.

Q Código 6.8: posix-efi cmake V

```

61 set(BOOTLOADER_SOURCES
62   bootloader.c
63   lib/elf/loader.c
64   lib/gop/font.c
65   lib/gop/framebuffer.c
66   lib/gop/gop.c
67   lib/memory/memory.c
68   lib/io/file.c
69   lib/acpi/acpi.c
70 )
71
72 set(POSIXUEFI_LIBS
73   ${POSIXUEFI_DIR}/crt_x86_64.o
74   ${POSIXUEFI_DIR}/libuefi.a
75 )
76
77 set(SOURCES
78   ${BOOTLOADER_SOURCES}
79   ${POSIXUEFI_LIBS}
80 )

```

Se establecen los archivos fuentes a compilar, se ha decidido incluirlos uno a uno para evitar errores futuros, esta suele ser la norma general al trabajar con proyectos grandes¹¹.

Q Código 6.9: posix-efi cmake VI

```

81 add_executable(bootloader ${POSIXUEFI_LIBS} ${BOOTLOADER_SOURCES})
82
83 SET_SOURCE_FILES_PROPERTIES(
84   ${POSIXUEFI_LIBS}
85   PROPERTIES
86   EXTERNAL_OBJECT true
87   GENERATED true

```

¹¹Referencia tomada del proyecto SerenityOS

```

88 )
89
90 set_target_properties(bootloader PROPERTIES
91   SUFFIX ".so"
92 )
93 set_target_properties(bootloader PROPERTIES
94   LINK_DEPENDS ${LINKER_SCRIPT}
95 )
96 add_custom_command(
97   TARGET bootloader
98   POST_BUILD
99   BYPRODUCTS bootloader.efi
100  COMMAND ${CMAKE_OBJCOPY} -j .text -j .sdata -j .data -j .dynamic -j .dynsym -j .rel -j .rela -j ↪
101    ↪ .rel.* -j .rela.* -j .reloc --target efi-app-x86_64 --subsystem=10 "<TARGET_FILE:<↪
102     ↪ bootloader>" "bootloader.efi"
103  VERBATIM
104 )

```

Finalmente en el Código 6.9 creamos el ejecutable final del bootloader. En primer lugar le decimos que compile y cree el bootloader con formato `.so`, también le indicamos que las librerías de `posix-uefi` son objetos generados por una librería externa y no por nosotros. Cuando se acabe de generar el bootloader se ejecutará un comando definido por nosotros, en nuestro caso será `objcopy` tal como nos lo indica `posix-uefi` en su `Makefile` y `gnu-efi` en la documentación.

⚠ Correctitud de cmake

Si no añadimos en `BYPRODUCTS` (Código 6.9) el fichero que genera el comando, `cmake` no sabrá que tiene que borrarlo cuando se ordene que limpie la construcción.

Para configurar y compilar el bootloader:

```

>_ cmake -B build && cd build
-- Configuring done
-- Generating done
-- Build files have been written to: /home/ecomaikgolf/alma/build

```

```

>_ make bootloader
[ 11%] Building C object bootloader/CMakeFiles/bootloader.dir/bootloader.c.o
[ 22%] Building C object bootloader/CMakeFiles/bootloader.dir/lib/elf/loader.c.o
[ 33%] Building C object bootloader/CMakeFiles/bootloader.dir/lib/gop/font.c.o
[ 44%] Building C object bootloader/CMakeFiles/bootloader.dir/lib/gop/framebuffe
[ 55%] Building C object bootloader/CMakeFiles/bootloader.dir/lib/gop/gop.c.o
[ 66%] Building C object bootloader/CMakeFiles/bootloader.dir/lib/memory/memory.
[ 77%] Building C object bootloader/CMakeFiles/bootloader.dir/lib/io/file.c.o
[ 88%] Building C object bootloader/CMakeFiles/bootloader.dir/lib/acpi/acpi.c.o
[100%] Linking C executable bootloader.so
[100%] Built target bootloader

```

6.4. Desarrollo del Bootloader

La funcionalidad y la lógica de ejecución del bootloader se encuentra en la función main, el resto de trabajo se ha ocultado a través de módulos y librerías que podemos encontrar en lib/. Las funciones que podemos encontrar aquí ofrecen funcionalidad de muy alto nivel al bootloader con el fin de mantener el main lo más limpio posible, para que así pueda reflejar de forma más clara las tareas de arranque

Las librerías escritas son las siguientes:

```
>_ ls -R lib
lib/bootparams.h lib/err_values.h
./lib/acpi:
acpi.c acpi.h
./lib/elf:
loader.c loader.h types.h
./lib/gop:
font.c font.h framebuffer.c framebuffer.h gop.c gop.h
./lib/io:
file.c file.h
./lib/log:
stdout.h
./lib/memory:
memory.c memory.h
```

6.4.1. Errores

Se ha definido una serie de códigos de retorno a lo largo del bootloader para identificar posibles errores (y su procedencia) a lo largo de la ejecución:

Código 6.10: err_values.h

```
1 enum
2 {
3     SUCCESS,
4     INCORRECT_ARGC,
5     KERNEL_LOAD_FAILURE,
6     GOP_RETRIEVE_FAILURE,
7     FRAMEBUFFER_FAILURE,
8     PSF1_FAILURE,
9     BOOTARGS_MEM,
10    UEFI_BS,
11};
```

Estos códigos se utilizan en el `main()` cuando se va a salir de este porque el error encontrado es fatal, de esta forma el desarrollador conoce el motivo del fallo, un ejemplo de uso es el siguiente:

© Código 6.11: err_values ejemplo

```
1 if (elf_header == NULL) {
2     error("cannot load the kernel");
3     return KERNEL_LOAD_FAILURE;
4 }
```

6.4.2. Logs

El módulo de logs (`log/`) es un módulo completamente opcional y tiene como función servir al programador de ayuda a la hora de mostrar o registrar información del proceso de arranque del kernel.

Este fichero se encarga de los logs mostrados por pantalla. Como podemos ver hay 4 tipos de logs: información, debug, aviso y error.

© Código 6.12: logs/stdout.h

```
1 /* (I) [bootloader] message */
2 #define info(message, ...) \
3     printf("(I) [bootloader] " message "\n", ##__VA_ARGS__)
4 /* (D) [bootloader] {file:function:line} message */
5 #define debug(message, ...) \
6     printf("(D) [bootloader] {%s:%s:%d} " message "\n", __FILE__, \
7             __PRETTY_FUNCTION__, __LINE__, ##__VA_ARGS__)
8 /* (W) [bootloader] {file:function:line} message */
9 #define warning(message, ...) \
10    printf("(W) [bootloader] {%s:%s:%d} " message "\n", __FILE__, \
11           __PRETTY_FUNCTION__, __LINE__, ##__VA_ARGS__)
12 /* (E) [bootloader] {file:function:line} message */
13 #define error(message, ...) \
14    printf("(E) [bootloader] {%s:%s:%d} " message "\n", __FILE__, \
15           __PRETTY_FUNCTION__, __LINE__, ##__VA_ARGS__)
```

Como se puede observar, son macros con argumentos variádicos, por lo que su funcionamiento es similar al de un `printf`, pudiendo marcar tipos de datos en el string a mostrar y pasándole las variables como argumento a la función de log.

>_ Ejemplo de mensajes de log

```
(I) [bootloader] Log informativo
(D) [bootloader] {bootloader.c:main:33} Log de debug
(W) [bootloader] {bootloader.c:main:34} Log de aviso
(E) [bootloader] {bootloader.c:main:35} Log de error
```

Al enviar un mensaje de log, en caso de que sea de debug, aviso o error, incluimos el fichero fuente donde ha ocurrido, la función y la línea concreta desde donde se ejecuta. De esta forma se facilita mucho las tareas de depuración del bootloader.

6.4.3. Ficheros

La carga de ficheros es un aspecto fundamental a la hora de escribir nuestro bootloader, tenemos que tener en cuenta que tendremos que cargar (como mínimo) el fichero ELF del kernel. Lo que buscamos es, a partir de un string (nombre del fichero), obtener un puntero a una dirección de memoria RAM con el contenido completo del archivo. Para esto tendremos que leer de disco el archivo y copiarlo en un fragmento de memoria.

También tendremos que conocer de antemano el tamaño del fichero, puesto que tendremos que reservar memoria dinámica suficiente para copiar su contenido, por lo que necesitaremos una función que tome un fichero y nos devuelva el tamaño total en bytes que ocupa el fichero.

Las funcionalidades que necesitamos de este módulo/librería se pueden desglosar en dos funciones, especificadas en `file.h`:

Q Código 6.13: `io/file.h`

```
1 uint64_t file_size(FILE *file);
2 void *load_file(const char *const filename);
```

6.4.3.1. Obtener el Tamaño de un Fichero

Q Código 6.14: `uint64_t file_size(FILE *file)`

```
1 uint64_t
2 file_size(FILE *file)
3 {
4     if (file == NULL) {
5         warning("file parameter is NULL");
6         return 0;
7     }
8
9     /* Store file read pointer */
10    uint64_t initial = ftell(file);
11    /* Move the file read pointer to the end */
12    fseek(file, 0, SEEK_END);
13    /* Return current in bytes */
14    uint64_t size = ftell(file);
15    /* Move the file reda pointer to the intial position*/
16    fseek(file, initial, SEEK_SET);
17
18    return size;
19 }
```

En primer lugar lo que haremos será comprobar si el puntero es nulo, en cuyo caso emitimos un warning y devolvemos 0, puesto que el tamaño de un archivo inexistente lo considero como 0 bytes¹².

En la línea 10 se guarda la posición de lectura del archivo con `ftell()`, posteriormente movemos dicho puntero en la línea 12 con `fseek()` al final del archivo, así en la línea 14

¹²Otro desarrollador podría argumentar que debe levantar un error puesto que el fichero no existe.

`fseek()` nos dice donde está el puntero de lectura y podemos asumir que esta lectura será el tamaño en bytes del archivo, puesto que está al final del todo. Finalmente recuperamos el puntero inicial para dejar el archivo como estaba y devolvemos el tamaño.

i Ficheros

Cabe recordar el funcionamiento de los ficheros en Linux. Todo fichero tiene almacenado un “file offset” (comunmente denominado como puntero) que nos indica la localización del fichero donde se va a producir la siguiente operación `read()` o `write()`:

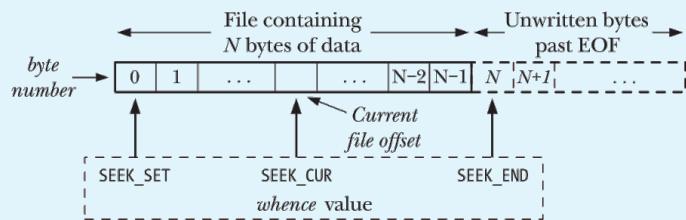


Figura 6.4: Funcionamiento SEEK_SET, SEEK_CUR y SEEK_END [8]

6.4.3.2. Cargar un Fichero en Memoria

```
Q Código 6.15: void *load_file(const char *filename)

1 void *
2 load_file(const char *filename)
3 {
4     info("opening '%s' file", filename);
5
6     FILE *file = fopen(filename, "r");
7
8     if (file == NULL) {
9         error("%s could not be opened", filename);
10        return NULL;
11    }
12
13    info("%s file opened", filename);
14
15    /* get file size to alloc enough space */
16    uint64_t size = file_size(file);
17
18    /* allocate memory for file contents */
19    void *memory = malloc(size);
20
21    if (memory == NULL) {
22        error("couldn't allocate memory for %s", filename);
23        return NULL;
24    }
25
26    info("allocated %d bytes for %s contents starting in 0x%p", size, filename, memory);
27
28    /* Copy file contents to memory */
29    fread((void *)memory, size, 1, file);
30    info("copied %s contents to memory 0x%p (%d bytes)", filename, memory, size);
```

```

31     fclose(file);
32     info("closed %s", filename);
33
34
35     return memory;
36 }
```

En este caso el funcionamiento de la función es bastante común al programar con una interfaz POSIX. Abrimos el fichero en modo lectura con `fopen` y obtenemos un puntero a un tipo `FILE`, comprobamos que no sea nulo (en cuyo caso lanzamos un error) y obtenemos el tamaño del fichero. Finalmente reservamos memoria suficiente con `malloc()`, comprobamos si hemos reservado memoria correctamente y copiamos el archivo desde el fichero a nuestro puntero en memoria¹³.

6.4.4. Gráficos

El módulo de gráficos se encuentra en `gop/` y es el encargado de obtener datos gráficos de EFI y construir una capa de compatibilidad para nuestro kernel. En este módulo se obtendrá en GOP (Graphics Output Protocol), se construirá un framebuffer para nuestro kernel y se cargará una fuente en formato PSF1.

El kernel utilizará el framebuffer y la fuente PSF1 para establecer una interfaz mínima funcional para imprimir caractéres y píxeles arbitrarios en pantalla.

6.4.4.1. Graphics Output Protocol

El Graphics Output Protocol [15, 16] es el estándar utilizado por UEFI que reemplazó el estándar VESA [17] (BIOS) y UGA (EFI 1.0). El protocolo nos provee de servicios runtime para interactuar con el apartado gráfico de nuestro sistema, en nuestro caso lo que queremos hacer será obtener un puntero a una estructura de tipo `efi_gop_t` para construir un framebuffer y pasárselo a nuestro kernel.

La única funcionalidad que queremos en nuestro módulo del GOP es obtener el GOP:

🔗 Código 6.16: `gop.h`

```
1 efi_gop_t *load_gop();
```

🔗 Código 6.17: `efi_gop_t *load_gop()`

```

1 efi_gop_t *
2 load_gop()
3 {
4     efi_gop_t *gop = malloc(sizeof(efi_gop_t));
5
6     if(gop == NULL) {
```

¹³No hace falta llevar el puntero de lectura al principio puesto que hemos abierto nosotros el archivo en este caso, a diferencia de la función anterior

```

7     error("cannot allocate memory for the GOP");
8     return NULL;
9 }
10
11 efi_guid_t gop_guid = EFI_GRAPHICS_OUTPUT_PROTOCOL_GUID;
12 efi_status_t status = BS->LocateProtocol(&gop_guid, NULL, (void **)&gop);
13
14 if (EFI_ERROR(status)) {
15     error("unable to initialise GOP");
16     return NULL;
17 }
18
19 return gop;
20 }
```

Como `posix-uefi` no tiene funciones POSIX (como es obvio) para obtener el GOP lo que tenemos que hacer es hacer llamadas con el runtime de UEFI directamente (aunque realmente usamos un wrapper más cómodo).

Lo que hacemos es usar `LocateProtocol` para que con el GUID que le pasemos nos devuelva el puntero que queremos, el del GOP. Crear una variable para asignarle el GUID se debe a que es un `define`, y `LocateProtocol` tiene que tomar un puntero al GUID.

6.4.4.2. Framebuffer

El framebuffer [18] es una porción de memoria RAM que contiene un bitmap mapeado a la memoria de video, por consiguiente los datos que escribamos a ese bitmap se mostrarán por pantalla. La dirección de memoria y las características del bitmap nos las da el GOP que hemos obtenido anteriormente, en este módulo creamos una estructura framebuffer para pasarsela al kernel.

Para representar el framebuffer definimos una estructura¹⁴ con toda la información relevante del framebuffer:

Código 6.18: `framebuffer.h`

```

1 typedef struct
2 {
3     /** Base address of the framebuffer */
4     char *base;
5     /** Total size */
6     unsigned long long buffer_size;
7     /** Screen width */
8     unsigned int width;
9     /** Screen height */
10    unsigned int height;
11    /** Pixels per scan line */
12    unsigned int ppscl;
13 } Framebuffer;
```

¹⁴Como se verá posteriormente, es importante que tanto el kernel como el bootloader tengan definida la estructura exactamente igual. Se podría definir la misma cabecera tanto para el kernel como para el bootloader.

La estructura consta de una serie de campos como un puntero a la dirección base de memoria que está mapeada a memoria de video, el tamaño total de dicha memoria (cuanto nos podemos extender), un ancho y alto de píxeles (de ventana, como una matriz) y los “pixels per scan line” que son el número de píxeles (visibles y no visibles¹⁵) por línea.

Puesto que estamos en C y no disponemos de constructores, definimos una función auxiliar que nos construya un framebuffer (en memoria dinámica) a partir del GOP obtenido en la sección 6.4.4.1

```
④ Código 6.19: framebuffer.h

1 Framebuffer *
2 create_fb(const efi_gop_t *const gop)
3 {
4     Framebuffer *fb = malloc(sizeof(Framebuffer));
5
6     if (fb == NULL) {
7         error("cannot allocate memory for the framebuffer");
8         return NULL;
9     }
10
11     fb->base = (char *)gop->Mode->FrameBufferBase;
12     fb->buffer_size = gop->Mode->FrameBufferSize;
13     fb->height = gop->Mode->Information->VerticalResolution;
14     fb->width = gop->Mode->Information->HorizontalResolution;
15     fb->ppsc1 = gop->Mode->Information->PixelsPerScanLine;
16
17     info("Window width: %d", fb->width);
18     info("Window height: %d", fb->height);
19
20     return fb;
21 }
```

Como vemos el funcionamiento es bastante simple, creamos un tipo **Framebuffer** en memoria dinámica y rellanamos sus campos con los homólogos del GOP.

➊ Dibujar Píxeles

El campo **base** de **Framebuffer** apunta a la memoria mapeada a vídeo por lo que cada escritura en este bloque de memoria se representará por pantalla.

Cada píxel son 32 bits de los cuales 24 son para valores RGB y 8 reservados [19]:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R	G	B	Reservado																												

Figura 6.5: Formato Píxel GOP

Por lo que para escribir (por ejemplo) el primer píxel en color blanco hacemos:

`*(uint32_t *)base = 0xFFFFFFFF`

¹⁵Los píxeles no visibles son píxeles que no se van a mostrar en pantalla y que se utilizan como padding en la memoria del framebuffer

6.4.4.3. Fuente

Como hemos visto, el framebuffer nos permite dibujar píxeles en pantalla. Si queremos imprimir un carácter tendremos que dibujarlo **pixel a pixel**. Para esto, tendremos que proveer al kernel de una fuente que le indique, para cada carácter, qué píxeles han de ser dibujados.

La fuente elegida se trata de `zap-light16.psf` (se puede encontrar en `toolchain/font`) que es una fuente de formato PSF1 (PC Screen Font 1 [20]). Para poder trabajar con la fuente, tendremos que implementar su especificación.

Una fuente PSF1 tiene el siguiente formato, con 0x36 y 0x04 como números mágicos:

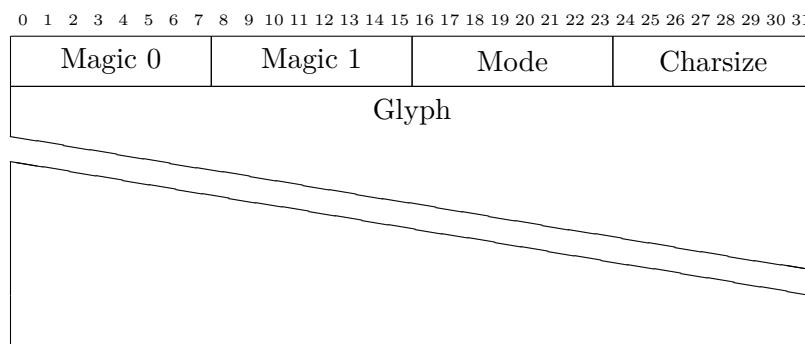


Figura 6.6: Fuente PSF1 [9]

Podemos comprobar la estructura y los valores mágicos abriendo la fuente `zap-light16.psf` en un visor hexadecimal:

```
>_ hexyl zap-light16.psf
00000000 36 04 02 10 00 00 00 3e 63 5d 7d 7b 77 77 7f 77 6•••000>c]}{ww•w
00000010 3e 00 00 00 00 00 00 00 00 7e 24 24 24 24 24 24 >0000000 0~$$$$$$
00000020 22 00 00 00 00 00 00 00 01 02 7f 04 08 10 7f 20 "0000000 *******
00000030 40 00 00 00 00 00 00 00 08 10 20 40 20 10 08 00 @0000000 .. @ ..0
00000040 7c 00 00 00 00 00 00 00 10 08 04 02 04 08 10 00 |0000000 *******0
```

Vemos que los números mágicos coinciden, que el modo es 0x2 y que el tamaño es 0x10 (16).

Conociendo la estructura de la fuente, podemos definirla en la cabecera `font.h` junto a una serie de funciones que nos ayuden a manejar dicha fuente. Necesitaremos una función para cargar la fuente a partir de un bloque de memoria y, en consecuencia, funciones para verificar si la función cargada es, en efecto, una fuente PSF1 (mediante los números mágicos mostrados anteriormente).

© Código 6.20: font.h

```

1 #define PSF1_MAGIC0 0x36
2 #define PSF1_MAGIC1 0x04
3
4 typedef struct
5 {
6     /** Magic number */
7     unsigned char magic[2];
8     /** Mode to manage number of glyphs */
9     unsigned char mode;
10    /** Size of a glyph bitmap */
11    unsigned char charsize;
12 } PSF1_Header;
13
14 typedef struct
15 {
16     PSF1_Header *header;
17     /** Glyph buffer data */
18     void *buffer;
19 } PSF1_Font;
20
21 PSF1_Font *load_psf1_font(const char *const);
22 PSF1_Header *get_psf1_header(const char *const);
23 void *get_psf1_glyph(const char *const);
24 uint8_t verify_psf1_header(const PSF1_Header *const);

```

Se define la fuente PSF1 (tanto cabecera como fuente en si), en la cabecera PSF1 `magic[0]` debe ser `0x36` y `magic[1]` `0x04`. `mode` es un campo especial para indicar el número de glifos y `charsize` nos dice el tamaño que ocupa cada glifo.

Para cargar una fuente `PSF1_Font` definiremos una función con la siguiente estructura, posteriormente se definirán los métodos:

© Código 6.21: `PSF1_Font *load_psf1_font(const char* const filename)`

```

1 PSF1_Font *
2 load_psf1_font(const char *const filename)
3 {
4     /* Copy file contents to memory */
5     char *memory = load_file(filename);
6
7     /* Not enough memory */
8     if (memory == NULL)
9         return NULL;
10
11    /* Get PSF1 header struct */
12    PSF1_Header *header = get_psf1_header(memory);
13
14    /* Verify PSF1 header */
15    uint8_t flags = verify_psf1_header(header);
16
17    /* Incorrect header */
18    if (flags > 0)
19        return NULL;
20
21    void *glyph_buffer = get_psf1_glyph(memory);
22
23    /* Create the font structure */
24    PSF1_Font *font = malloc(sizeof(PSF1_Font));
25
26    /* Not enough memory */

```

```

27     if (font == NULL)
28         return NULL;
29
30     font->header = header;
31     font->buffer = glpyh_buffer;
32
33     return font;
34 }
```

Podemos ver que en primer lugar se carga la fuente en memoria a partir de un fichero especificado mediante su nombre (o path), acto seguido se extraen las cabeceras PSF1 de los datos del fichero, y se verifica si corresponden con una fuente PSF1. Si corresponden con una fuente PSF1 se obtiene el buffer de glifos, se construye el struct final de la fuente y se devuelve.

© Código 6.22: PSF1_Header * get_psf1_header(const char *const memory)

```

1 PSF1_Header *
2 get_psf1_header(const char *const memory)
3 {
4     return (PSF1_Header *)memory;
5 }
```

Como la cabecera PSF1 siempre estará al principio del archivo extraerla es simplemente hacer un cast¹⁶.

© Código 6.23: uint8_t verify_psf1_header(const PSF1_Header *const header)

```

1 uint8_t
2 verify_psf1_header(const PSF1_Header *const header)
3 {
4     /* magic[0] == 0x36, magic[1] == 0x04 */
5     if (header->magic[0] != PSF1_MAGIC0 || header->magic[1] != PSF1_MAGIC1) {
6         error("PSF1 font incorrect magic header");
7         return 1;
8     } else {
9         info("PSF1 font correct magic header");
10    return 0;
11 }
```

Verificar si la cabecera obtenida es en efecto una fuente PSF1 es sencillo, lo único que hacemos es comprobar el primer y segundo byte con unos números mágicos predefinidos: 0x36 y 0x04.

© Código 6.24: void *get_psf1_glyph(const char *const memory)

```

1 void *
2 get_psf1_glyph(const char *const memory)
3 {
4     return (void *)(memory + sizeof(PSF1_Header));
5 }
```

Obtener el buffer de glifos de una fuente PSF1 se consigue mediante aritmética de punteros.

¹⁶Se ha preferido encapsularlo en una función propia por abstracción.

El buffer de glifos estará justo después de la cabecera PSF1 (Figura 6.6), entonces lo que hacemos es tomar la dirección inicial, le sumamos el offset en bytes que ocupa la cabecera PSF1 y el resultado será el puntero al buffer de glifos.

6.4.5. Mapa de Memoria

Al usar EFI no disponemos de una distribución de memoria prefijada que conozcamos con exactitud de antemano. En EFI la distribución de memoria inicial puede variar (por ejemplo se han reservado bloques de memoria por el uso de servicios de EFI) o incluso empezar de forma distinta. Para gestionar la memoria tanto en el bootloader como en el kernel se hace uso del mapa de memoria reportado por EFI.

Podemos consultar cómo es el mapa de memoria al arrancar nuestro ordenador mediante el comando `memmap` de la shell de EFI:

Type	Start	End	Pages	Attributes
(...)				
Available	0000000007E00000-0000000007E9DFFF	0000000000000009E	0000000000000000000F	00000000000000000000
BS_Data	0000000007E9E000-0000000007EBDFFF	00000000000000020	0000000000000000000F	00000000000000000000
BS_Code	0000000007EBE000-0000000007ED6FFF	00000000000000019	0000000000000000000F	00000000000000000000
BS_Data	0000000007ED7000-0000000007EDFFFF	00000000000000009	0000000000000000000F	00000000000000000000
BS_Code	0000000007EE0000-0000000007EF3FFF	00000000000000014	0000000000000000000F	00000000000000000000
RT_Data	0000000007EF4000-0000000007F77FFF	00000000000000084	8000000000000000000F	80000000000000000000
ACPI_NVS	0000000007F78000-0000000007FFFFFF	00000000000000088	0000000000000000000F	00000000000000000000
Reserved :		128 Pages (524,288 Bytes)		
LoaderCode:		236 Pages (966,656 Bytes)		
LoaderData:		0 Pages (0 Bytes)		
BS_Code :		643 Pages (2,633,728 Bytes)		
BS_Data :		6,638 Pages (27,189,248 Bytes)		
RT_Code :		256 Pages (1,048,576 Bytes)		
RT_Data :		388 Pages (1,589,248 Bytes)		
ACPI_Recl :		16 Pages (65,536 Bytes)		
ACPI_NVS :		512 Pages (2,097,152 Bytes)		
MMIO :		0 Pages (0 Bytes)		
MMIO_Port :		0 Pages (0 Bytes)		
PalCode :		0 Pages (0 Bytes)		
Available :		23,855 Pages (97,710,080 Bytes)		
Persistent:		0 Pages (0 Bytes)		
<hr/>				
Total Memory:		127 MB (133,300,224 Bytes)		

Al sumar toda la memoria encontraremos que es una aproximación muy cercana al tamaño de RAM instalado. Es importante recalcar que no todo será utilizable para almacenar datos por parte del kernel.

Para trabajar con el mapa de memoria que nos proporciona EFI definiremos una estructura general para agrupar toda la información del mapa de memoria, una función para construir dicho mapa y una función auxiliar que lo imprima.

© Código 6.25: `memory/memory.h`

```

1  /** num of tries to get correct memory size */
2  static const uint8_t RETRIES = 5;
3
4  typedef struct
5  {
6      /** Array of memory descriptors */
7      efi_memory_descriptor_t *map;
8      uint64_t map_size;
9      uint64_t map_key;
10     uint64_t descriptor_size;
11     uint64_t entries;
12 } MapInfo;
13
14 MapInfo *load_memmap();
15 void print_memmap(const MapInfo *);
```

`MapInfo` es la estructura que actúa como interfaz del mapa de memoria de EFI, tenemos una array de descriptores de memoria que serán bloques de memoria, un tamaño total de mapa, el tamaño de cada descriptor y el número de entradas totales.

El descriptor de memoria de EFI nos lo provee posix-uefi (basándose en gnu-efi, que a su vez sigue el estándar de UEFI [21]) y es el siguiente:

© Código 6.26: `efi_memory_descriptor_t`

```

1  typedef struct efi_memory_descriptor_t
2  {
3      uint32_t type;
4      efi_physical_address PhysicalStart;
5      efi_virtual_address VirtualStart;
6      uint64_t NumberOfPages;
7      uint64_t Attribute;
8 }
```

La función encargada de cargar el mapa de memoria EFI y construir el `MapInfo` que pasaremos a nuestro kernel es `load_memmap`:

© Código 6.27: `MapInfo *load_memmap()`

```

1 MapInfo *
2 load_memmap()
3 {
4     MapInfo *map = calloc(1, sizeof(MapInfo));
5
6     if (map == NULL) {
7         error("can't allocate memory for UEFI memory map info");
8         return NULL;
9     }
10
11    efi_status_t status =
12        BS->GetMemoryMap(&map->map_size, NULL, &map->map_key, &map->descriptor_size, NULL);
13
14    /* status will be EFI_BUFFER_TOO_SMALL as mapsize = 0 and give us the correct size */
```

```

15     if (status != EFI_BUFFER_TOO_SMALL || map->map_size <= 0 || map->descriptor_size <= 0) {
16         /*
17          * Check if map->map_size is a reasonable number
18          * View page 164 of UEFI Specification 2.8
19          */
20         warning("memory map info retrieval error");
21         return NULL;
22     }
23
24     /* Retry to get correct size, not needed but sometimes it randomly faults and this patches it ↵
25      ↵ */
26     for (int i = 0; i < RETRIES; i++) {
27
28         map->map = malloc(map->map_size);
29
30         if (map->map == NULL) {
31             error("can't allocate memory for UEFI memory map info");
32             return NULL;
33         }
34
35         status =
36             BS->GetMemoryMap(&map->map_size, map->map, &map->map_key, &map->descriptor_size, NULL);
37
38         if (status == 0) {
39             break;
40         } else {
41             if (i != 0)
42                 warning("Memory map size try number %d", i);
43             free(map->map);
44         }
45
46     if (status > 0) {
47         /* Possible errors: view page 164 of UEFI Specification 2.8 */
48         error("Status error: %d", status);
49         error("memory map retrieval error");
50         return NULL;
51     }
52
53     map->entries = map->map_size / map->descriptor_size;
54
55     if (map->entries == 0)
56         warning("memory map without entries");
57
58     return map;
59 }
```

En primer lugar se reserva memoria inicializada a 0 suficiente para almacenar el **MapInfo**. Acto seguido se llama al servicio **GetMemoryMap** de EFI que nos devolverá el tamaño del mapa¹⁷. Luego iteramos¹⁸ una serie de veces para obtener el mapa de memoria mediante el mismo servicio **GetMemoryMap** (previamente reservando¹⁹ el tamaño suficiente, obtenido en el paso anterior).

Finalmente calculamos cuantas entradas tendremos en el mapa de memoria (filas del comando **memmap**) dividiendo el tamaño del mapa entre el tamaño de cada descriptor de memoria (entrada en el mapa).

¹⁷La funcionalidad varía dependiendo de los parámetros que le pases

¹⁸No es necesario como tal pero es una recomendación que encontré en una guía de desarrollo.

¹⁹Hacer **malloc()** y **free()** dentro del bucle puede no ser lo más recomendable.

`print_memmap` es una función auxiliar que nos imprime el mapa de memoria actual por pantalla, además transforma el tipo de segmento de memoria a un string legible:

```
© Código 6.28: void print_memmap(const MapInfo *map)

1 char desctypes[] [26] = {
2     "EfiReservedMemoryType",
3     "EfiLoaderCode",
4     "EfiLoaderData",
5     "EfiBootServicesCode",
6     "EfiBootServicesData",
7     "EfiRuntimeServicesCode",
8     "EfiRuntimeServicesData",
9     "EfiConventionalMemory",
10    "EfiUnusableMemory",
11    "EfiACPIReclaimMemory",
12    "EfiACPIMemoryNVS",
13    "EfiMemoryMappedIO",
14    "EfiMemoryMappedIOPortSpace",
15    "EfiPalCode"
16 };
17
18 void
19 print_memmap(const MapInfo *map)
20 {
21     uint8_t *start = (uint8_t *)map->map;
22     for (uint64_t i = 0; i < (map->map_size / map->descriptor_size); i++) {
23         efi_memory_descriptor_t *descriptor =
24             (efi_memory_descriptor_t *)((uint64_t)start + (i * map->descriptor_size));
25         debug("Type %s", desctypes[descriptor->Type]);
26         debug("Phy start %p", descriptor->PhysicalStart);
27         debug("Number of pages %d", descriptor->NumberOfPages);
28         debug("Pad %d\n", descriptor->Pad);
29     }
30 }
```

⚠ Por qué `char [] [28]` y no `const char * []` ?

El hecho de que `desctypes` no sea un array de `const char*` es porque al imprimir el string del tipo de descriptor de memoria con `const char *` tenía problemas. Es algo que en un entorno común de desarrollo en C++ funciona sin problemas pero en mi caso no me imprimía bien el texto, nunca entendí el motivo.

Entiendo que la diferencia principal probablemente sea que la versión con `const char *` se guarda en una zona distinta del ELF (*hardcoded*), pero igual lo están los strings de las líneas 25-27.

Gracias a la función `print_memmap` podremos imprimir el mapa de memoria directamente desde el bootloader, sin necesidad de hacerlo con `memmap` en la shell de EFI. Actualmente si vemos el código del bootloader es una función que está en desuso por defecto, solo se ha implementado para que el desarrollador manualmente modifique²⁰ el código y añada una llamada a esta donde necesite.

²⁰Esto se ha decidido así puesto que el bootloader es un programa sin ningún tipo de entrada interactiva para el usuario, no ofrecemos una línea de comandos ni nada similar, es un proceso secuencial y predefinido.

6.4.6. ACPI

ACPI (Advanced Configuration and Power Interface) es un estándar abierto desarrollado por Intel, Microsoft y Toshiba que los sistemas operativos usan para descubrir y configurar componentes hardware. Para que el kernel pueda trabajar con ACPI tendremos que pasarle el RSDP (Root System Descriptor Pointer) desde el bootloader. El RSDP es una estructura de datos que nos da información sobre la versión de ACPI que implementa la máquina y, lo más importante, nos da acceso a la tabla principal de ACPI, la RSDT.

Arquitectura de ACPI

La arquitectura de ACPI se compone de tres componentes principales: las tablas ACPI, la ACPI BIOS y los registros ACPI. La ACPI BIOS genera las tablas ACPI y las carga en memoria.

Para empezar a trabajar con ACPI es necesario obtener el RSDP (Root System Descriptor Pointer). Dependiendo de la versión de ACPI que tengamos podemos tener una versión de la estructura de datos u otra, pero es importante recalcar que la `rsdp_v2` es compatible con la `rsdp_v1`²¹.

 Código 6.29: `rsdp_v1`

```

1 typedef struct
2 {
3     /* Null terminated string that has to be "RSDP PTR " (see the last space) */
4     char signature[8];
5     /* byte cast of the sum of all bytes must be = 0*/
6     uint8_t checksum;
7     /* OEM string */
8     char oem[6];
9     /* ACPI Version */
10    uint8_t version;
11    /* RSDT physic address*/
12    uint32_t rsdt;
13 } __attribute__((packed)) rsdp_v1;
```

 Código 6.30: `rsdp_v2`

```

1 typedef struct
2 {
3     /* Old header before (new items in v2 are "appended" so it's backward compatible) */
4     rsdp_v1 header_v1;
5     /* table length from 0 to end */
6     uint32_t length;
7     /* XSDT physic address (if ACPI 2.0, use this instead of header_v1->rsdt, even in 32bit) */
8     uint64_t xsdt;
9     /* as rsdp_v1->checksum */
10    uint8_t checksum_v2;
11    /* reserved */
12    uint8_t reserved[3];
13 } __attribute__((packed)) rsdp_v2;
```

²¹Esto lo vemos viendo que la v2 contiene como primer elemento la v1, la v2 es una **extensión** de la v1. Al leer de memoria la estructura lo único que haremos si tenemos la v2 es leer más bytes.

El RSDT [22] y el XSDT [23] (campos de las estructuras) son las tablas principales de ACPI (Root System Descriptor Table), son las que utilizará el kernel para trabajar con ACPI y son el motivo principal de obtener el RSDP.

El RSDP se puede obtener de varias formas, por ejemplo en BIOS el RSDP se puede encontrar en el primer kilobyte de la EBDA (Extended BIOS Data Area) o entre 0x000E0000 y 0x000FFFFF [24]. Como estamos bajo el paraguas de EFI no nos tenemos que preocupar mucho, el RSDP se encuentra en la `EFI_SYSTEM_TABLE`.

Para obtener el RSDP definimos una función en la que iteraremos la `EFI_SYSTEM_TABLE` hasta encontrar una entrada con un GUID determinado: `ACPI_20_TABLE_GUID` (proporcionado por posix-uefi).

```
© Código 6.31: rsdp_v2 *load_rsdp()
1 rsdp_v2 *
2 load_rsdp()
3 {
4     info("Loading RSDP table");
5     efi_configuration_table_t *config_table = ST->ConfigurationTable;
6     rsdp_v2 *rsdp = NULL;
7     efi_guid_t acpi20_table_guid = ACPI_20_TABLE_GUID;
8
9     for (uintn_t i = 0; i < ST->NumberOfTableEntries; i++) {
10         if (compare_guid(&config_table->VendorGuid, &acpi20_table_guid)) {
11             if (strncpy("RSD PTR ", config_table->VendorTable, 8)) {
12                 rsdp = (rsdp_v2 *)config_table->VendorTable;
13                 info("RSDP table found in %p", rsdp);
14                 break;
15             }
16         }
17         config_table++;
18     }
19
20     if (rsdp == NULL)
21         error("RSDP table not found");
22
23     return rsdp;
24 }
```

Al encontrar un GUID que coincida compraremos la cadena identificativa con ‘RSD PTR’²² para ver si de verdad se trata del RSDP²³.

Como se puede observar en la línea 10, se hace uso de una función para comparar GUIDs. Los GUIDs son una estructura de datos definida por posix-uefi de la siguiente forma:

```
© Código 6.32: efi_guid_t
1 typedef struct {
2     uint32_t Data1;
3     uint16_t Data2;
4     uint16_t Data3;
5     uint8_t Data4[8];
6 } efi_guid_t;
```

²²Con el espacio al final y sin terminación null.

²³También puede utilizar para encontrar el RSDP en caso de que no estemos en un entorno EFI.

Por lo que para comparar dos GUIDs debemos crear una función auxiliar de comparación:

```
Q Código 6.33: bool compare_guid(efi_guid_t *g1 efi_guid_t *g2)
1 bool
2 compare_guid(efi_guid_t *g1, efi_guid_t *g2)
3 {
4     return (g1->Data1 == g2->Data1 && g1->Data2 == g2->Data2 && g1->Data3 == g2->Data3 &&
5         *(uint64_t *)g1->Data4 == *(uint64_t *)g2->Data4);
6 }
```

6.4.7. ELF

ELF (Executable and Linkable Format)²⁴, es el formato por excelencia para ejecutables, librerías y código objeto en sistemas Unix. Se publicó inicialmente en la especificación del ABI (Application Binary Interface) de Unix [25] y más tarde en el “Tool Interface Standard” [26].

El kernel desarrollado, al compilarlo, es un archivo ELF. El bootloader tiene la responsabilidad de encontrar su fichero ELF, verificar que el formato del fichero sea correcto, obtener datos relevantes del ejecutable y cargarlo en memoria. El proceso de carga de archivos ELF no es tan simple como la carga en memoria de ficheros planos que hemos visto antes, es un proceso que pese a que en su versión más básica no parece complejo, a la hora de implementar una versión que siga el estándar y que funcionen bien, no es tarea fácil.

Los archivos ELF tienen, al principio del fichero, una cabecera denominada “ELF Header”. La cabecera al igual que otros muchos formatos dispone de una serie de bytes denominados números mágicos que nos sirven para verificar que el fichero en cuestión es en efecto, un fichero de tipo ELF, en este caso son 0x7f, 'E', 'L' y 'F'. La cabecera incluye, a parte de los magic bytes, una serie de flags en sus primeros 16 bytes que indican versiones de la especificación, si el ELF es 32 o 64 bits, el *endianess*²⁵ etc. Acto seguido dispone de una serie de campos que nos proporcionan offsets de otras partes del archivo, dirección de punto de entrada (dónde hay que empezar a ejecutar código del archivo), etc.

```
Q Código 6.34: ELF Header
1 typedef struct elf64_hdr {
2     unsigned char e_ident[EI_NIDENT];
3     Elf64_Half e_type;
4     Elf64_Half e_machine;
5     Elf64_Word e_version;
6     Elf64_Addr e_entry; /* Entry point virtual address */
```

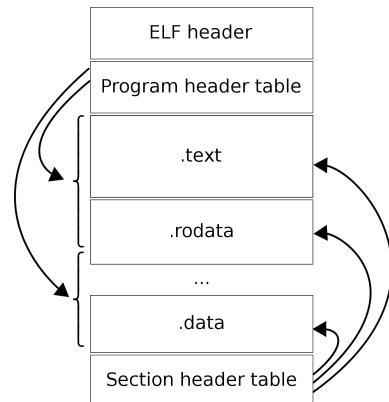


Figura 6.7: Formato ELF

²⁴Anteriormente Extensible Linking Format

²⁵El orden de los bits en el fichero

```

7 Elf64_Off e_phoff; /* Program header table file offset */
8 Elf64_Off e_shoff; /* Section header table file offset */
9 Elf64_Word e_flags;
10 Elf64_Half e_ehsize;
11 Elf64_Half e_phentsize;
12 Elf64_Half e_phnum;
13 Elf64_Half e_shentsize;
14 Elf64_Half e_shnum;
15 Elf64_Half e_shstrndx;
16 } Elf64_Ehdr;

```

A continuación de la cabecera ELF tendremos las “Program Headers”. Estas son cabeceras con el fin de indicar al cargador de ELF’s cómo cargar el fichero en memoria, por ello solo aparecen en ficheros ELF ejecutables. Proporcionan una visión del fichero en “segmentos” a diferencia de “secciones” (Section Header), un segmento está compuesto por 0 o más secciones, agrupándolas en un único bloque.

© Código 6.35: ELF Program header

```

1 typedef struct {
2     uint32_t p_type; /* Segment type */
3     uint32_t p_flags; /* Segment flags */
4     uint64_t p_offset; /* Segment file offset */
5     uint64_t p_vaddr; /* Segment virtual address */
6     uint64_t p_paddr; /* Segment physical address */
7     uint64_t p_filesz; /* Segment size in file */
8     uint64_t p_memsz; /* Segment size in memory */
9     uint64_t p_align; /* Segment alignment */
10 } Elf64_Phdr;

```

Podemos ver las cabeceras de programa mediante la herramienta `readelf`:

```

>_ readelf --program-headers a.out

```

Tipo	Desplazamiento	DirVirtual	DirFísica	Opts	Alineación
	TamFichero	TamMemoria			
PHDR	0x0000000000000040	0x0000000000000040	0x0000000000000040		
	0x00000000000002d8	0x00000000000002d8	R	0x8	
INTERP	0x0000000000000318	0x0000000000000318	0x0000000000000318		
	0x000000000000001c	0x000000000000001c	R	0x1	
[Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]					
LOAD	0x0000000000000000	0x0000000000000000	0x0000000000000000		
	0x00000000000006e8	0x00000000000006e8	R	0x1000	

Luego de las “Program Headers” encontraremos el bloque de secciones. Las secciones son utilizadas tanto por los program headers como por las section headers. Estas secciones son las que realmente contienen datos del ELF, por ejemplo el código compilado se suele encontrar en la sección `.text`, los datos constantes y estáticos en `.rodata`, bloques de memoria sin inicializar en `.bbs`, etc. Como se ha podido apreciar, cada sección suele estar representada por un nombre (string) que podemos encontrar accediendo a una tabla especial de strings mediante un índice de la *section header*.

Es importante tener en cuenta que las secciones no tienen estructura como tal, pueden ser bloques de memoria sin aparente sentido. Es responsabilidad de las cabeceras de las secciones (“Section Headers”) dotarles de un sentido. Es por esto que herramientas como `readelf` hacen lo mismo al ejecutar `--section-headers` y `--sections`.

Finalmente, seguido de las secciones encontraremos sus cabeceras. Las cabeceras, como ya se ha mencionado, dotan de sentido a las secciones. Las secciones son utilizadas por el *linker* y herramientas de análisis estático de binarios.

▲ ELF sin Section Headers

Podemos encontrar ficheros ELF sin section headers, estas cabeceras se utilizan por el enlazador, por lo que si no necesitamos enlazar nuestro programa podemos no incluir esta parte del fichero y construir un ELF sin section headers.

© Código 6.36: ELF Section header

```

1 typedef struct {
2     uint32_t sh_name; /* Section name tbl index */
3     uint32_t sh_type; /* Section type */
4     uint64_t sh_flags; /* Section flags */
5     uint64_t sh_addr; /* virtual addr at execution */
6     uint64_t sh_offset; /* section file offset */
7     uint64_t sh_size; /* section size (bytes) */
8     uint32_t sh_link; /* link to another section */
9     uint32_t sh_info; /* additional information */
10    uint64_t sh_addralign; /* section alignment */
11    uint64_t sh_entsize; /* entry size if section holds a table */
12 } Elf64_Shdr;

```

Podemos ver las cabeceras de programa mediante la herramienta `readelf`:

>_ readelf --sections a.out						
[Nr]	Nombre	Tipo	Dirección	Despl		
	Tamaño	TamEnt	Opts	Enl	Info	Alin
[0]		NULL	0000000000000000	0	0	0
	0000000000000000	0000000000000000				
[1]	.interp	PROGBITS	000000000000318	00000318		
	0000000000000001c	0000000000000000	A	0	0	1
[2]	.note.gnu.pr[...]	NOTE	000000000000338	00000338		
	0000000000000040	0000000000000000	A	0	0	8
[3]	.note.gnu.bu[...]	NOTE	000000000000378	00000378		
	0000000000000024	0000000000000000	A	0	0	4
[4]	.note.ABI-tag	NOTE	000000000000039c	0000039c		
	0000000000000020	0000000000000000	A	0	0	4
[5]	.gnu.hash	GNU_HASH	00000000000003c0	000003c0		
	0000000000000024	0000000000000000	A	6	0	8

6.4.7.1. Cargar fichero ELF

La tarea general que tendremos que resolver en el bootloader en este apartado es cargar un fichero ELF. La función general para realizar esta tarea sería la siguiente:

```
© Código 6.37: Elf64_Ehdr *load_elf(const char *const filename)
1 Elf64_Ehdr *
2 load_elf(const char *const filename)
3 {
4     char *memory = (char *)load_file(filename);
5
6     if (memory == NULL)
7         return NULL;
8
9     Elf64_Ehdr *header = get_elf_header(memory);
10    uint8_t flags = verify_elf_headers(header);
11
12    if (flags > 0) {
13        error("wrong ELF header, status code: %d", flags);
14        return NULL;
15    }
16
17    load_phdrs(header, memory);
18
19    return header;
20 }
```

En primer lugar se encuentra el fichero buscado y se cargan todos los contenidos en memoria, acto seguido de ese bloque de ceros y unos obtenemos la cabecera del archivo ELF. Acto seguido comprobamos si efectivamente se trata de un archivo ELF y, finalmente, cargamos el código en memoria.

6.4.7.2. Obtener y Verificar la Cabecera ELF

Para obtener la cabecera del fichero ELF, como se puede observar en la Figura 6.7, simplemente tenemos que acceder a los primeros bytes del fichero. Esto se resuelve con un simple cast a (`Elf64_Ehdr *`), lo encapsularemos en una función por abstracción.

```
© Código 6.38: Elf64_Ehdr *get_elf_header(char *memory)
1 Elf64_Ehdr *
2 get_elf_header(char *memory)
3 {
4     /* First data of ELF file is the ELF header */
5     return (Elf64_Ehdr *)memory;
6 }
```

Una vez obtenida la cabecera tendremos que verificar si se corresponde con la de un fichero ELF. Esto consiste en comprobar, de su cabecera, si el campo `e_ident`²⁶ contiene unas constantes predefinidas en (el magic number). En el caso de ELF el magic number es `0x7F`,

²⁶Los magic bytes se encuentran primero, por lo que los encontraremos en los 4 primeros bytes del fichero.

'E', 'L', 'F'. También realizamos chequeos adicionales comprobando si la cabecera del fichero corresponde con un ELF ejecutable, si es de 64 bits, si tiene uno o más program headers, etc.

```
Q Código 6.39: uint8_t verify_elf_headers(const Elf64_Ehdr *const elf_header)
1 uint8_t
2 verify_elf_headers(const Elf64_Ehdr *const elf_header)
3 {
4     if (elf_header == NULL) {
5         warning("elf_header parameter is null");
6         return 255;
7     }
8
9     uint8_t elf_parse_flags = 0;
10
11    for (int i = 0; i < NUM_CHECKS; i++) {
12        uint8_t check = 255; /* NUM_CHECKS > actual checks, throw warning */
13        switch (i) {
14            case 0:
15                /* Magic header */
16                check = !(memcmp(elf_header->e_ident, ELFMAG, SELFMAG) == 0);
17                break;
18            case 1:
19                /* Executable ELF */
20                check = !(elf_header->e_type == ET_EXEC);
21                break;
22            case 2:
23                /* ELF Architecture */
24                check = !(elf_header->e_machine == EM_MACH);
25                break;
26            case 3:
27                /* ELF 64 bits target */
28                check = !(elf_header->e_ident[EI_CLASS] == ELFCLASS64);
29                break;
30            case 4:
31                /* ELF non empty program headers */
32                check = !(elf_header->e_phnum > 0);
33                break;
34            default:
35                printf("(W) [bootloader] verify_elf_headers(...) default switch case reached\n");
36        }
37
38        if (check != 255) {
39            /* Mark the bit with 1 to identify the check error */
40            elf_parse_flags |= check << i;
41
42            (check > 0) ? error("%s", errormessages[i]) : info("%s", verifymessages[i]);
43        }
44    }
45    return elf_parse_flags;
46 }
```

El valor devuelto es un byte que activa ciertos bits dependiendo del error:

0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0

Figura 6.8: Formato Código de Error

6.4.7.3. Cargar los Program Headers

Como hemos visto anteriormente, los program headers son la parte del archivo ELF que indica al cargador como crear la imagen del proceso [27]: dónde y qué bloques de código cargar, etc. Si queremos ejecutar el kernel tendremos que cargar los program headers primeros para construir la imagen del proceso.

Para cargar el ELF del kernel iteraremos todas las program headers, esto lo conseguimos mediante aritmética de punteros: `memory` es la dirección inicial del fichero ELF en la memoria del bootloader, `e_phoff` es el offset en bytes desde el inicio del ELF hasta la tabla de program headers. El tamaño de cada program header viene dado por `e_phentsize` por lo que simplemente vamos saltando dicho tamaño en cada iteración.

```
❷ Código 6.40: void load_phdrs(const Elf64_Ehdr *const elf_header
1 void
2 load_phdrs(const Elf64_Ehdr *const elf_header, const char *const memory)
3 {
4     for (int i = 0; i < elf_header->e_phnum; i++) {
5         Elf64_Phdr *prog_hdr =
6             (Elf64_Phdr *) (memory + elf_header->e_phoff + elf_header->e_phentsize * i);
7         switch (prog_hdr->p_type) {
8             /* Loadable */
9             case PT_LOAD: {
10                 info("loading program header %d at: 0x%p", i, prog_hdr->p_vaddr);
11
12                 memcpy((void *)prog_hdr->p_vaddr, memory + prog_hdr->p_offset, prog_hdr->p_filesz);
13                 memset((void *)(prog_hdr->p_vaddr + prog_hdr->p_filesz),
14                         0,
15                         prog_hdr->p_memsz - prog_hdr->p_filesz);
16                 break;
17             }
18             /* Ignore */
19             default:
20                 info("program header %d of type %d ignored", i, prog_hdr->p_type);
21         }
22     }
23 }
```

Para cada iteración comprobaremos el tipo de program header encontrada, si es `PT_LOAD` lo que se nos indica es que esta program header debe ser cargada en memoria. Copiamos `p_filesz` bytes en `p_vaddr`, la dirección virtual donde se nos dice que carguemos la sección.

⚠ Incompletitud del Cargador ELF

Al ignorar el resto de tipos de program headers el cargador es incompleto y no cumple con la especificación. No es el objetivo escribir un cargador de ELFs completo puesto que es una tarea muy compleja.

Finalmente, justo después del bloque de memoria que hemos copiado, establecemos `p_memsz - p_filesz` bytes a 0. Esto se debe a que podemos tener bloques de memoria que deben ser inicializados a 0 y que no tienen por qué ocupar espacio en el ELF, en vez de guardar espacio

para ello en el fichero como tal simplemente se indica en la discrepancia entre tamaño que ocupa en fichero y en memoria la sección.

Con este último paso tendríamos el mínimo funcional para poder cargar el ELF de nuestro kernel en la máquina. Es importante recalcar que esta función desarrollada tiene un fallo conocido, la explicación del error se encuentra en la sección 6.5.1.1.

6.4.7.4. Llamar a los Constructores Globales

Si bien este apartado no corresponde a la carga del ELF como tal, está estrechamente ligado con su ejecución. Como bien sabemos en C/C++ podemos indicar al compilador, mediante atributos, que ciertas funciones deben ejecutarse automáticamente al principio del programa.

Q Código 6.41: Ejemplo de función constructora

```
1 void foo(void) __attribute__((constructor));
```

Por ejemplo, esta función se ejecutará antes de entrar en el `main()` de nuestro programa. También podemos encontrar el mismo caso en objetos globales de C++, por ejemplo este objeto (en scope global) tendrá que llamar a su constructor por defecto antes de que `main()` empiece:

Q Código 6.42: Ejemplo de objeto global con constructor

```
1 class A {
2     A() {
3         ...
4     }
5 }
6 A a;
```

Si compilamos un ejemplo simple con una función constructora veremos (utilizando `radare2` [28]) que se nos incluyen gran cantidad de funciones definidas en el ELF resultante:

```
>_ radare2
[0x00001050]> afl
0x00001050    1 38          entry0
0x00001080    4 41    -> 34  sym.deregister_tm_clones
0x000010b0    4 57    -> 51  sym.register_tm_clones
0x000010f0    5 65    -> 55  sym.__do_global_dtors_aux
0x00001140    1 9           entry.init0
0x00001149    1 22           main
0x00001270    1 13           sym._fini
0x0000115f    1 270          info()
0x00001040    1 6            sym.imp.fwrite
0x00001030    1 6            sym.imp.printf
0x00001000    3 27          sym._init
```

```
>_ readelf --headers a.out

Encabezado ELF:
  Mágico: 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Clase: ELF64
  Datos: complemento a 2, little endian
  Version: 1 (current)
  OS/ABI: UNIX - System V
  Versión ABI: 0
  Tipo: DYN (Position-Independent Executable file)
  Máquina: Advanced Micro Devices X86-64
  Versión: 0x1
  Dirección del punto de entrada: 0x1050
  Inicio de encabezados de programa: 64 (bytes en el fichero)
  Inicio de encabezados de sección: 14144 (bytes en el fichero)
(...)
```

Como podemos ver el ELF tiene como punto de entrada la dirección 0x1050, que coincide si vemos lo que muestra radare2 con la función `entry0` y no con `main()`. Por lo que `main()` no es lo primero que se ejecuta del ELF.

Para ahorrar varias páginas de explicación, lo que se ejecuta **antes** del `main` son las funciones constructoras que hemos visto. Todo el trabajo de llamar a constructores globales y funciones se hace de forma transparente, gracias al `entry0`, antes del `main`.

Puesto que el kernel es en C++ y tendremos objetos globales es bastante importante implementar esta funcionalidad. Hay dos grandes formas de hacerlo, una mediante gcc y modificando los parámetros de enlazado y otra de forma manual. En el trabajo se han implementado las dos formas y, en esta sección, se explicará la última.

© Código 6.43: void call_ctors(Elf64_Ehdr *elf)

```
1 void
2 call_ctors(Elf64_Ehdr *elf)
3 {
4     Elf64_Shdr *str_table_hdr =
5         (Elf64_Shdr *)((char *)elf + elf->e_shoff + (elf->e_shentsize * elf->e_shstrndx));
6     char *str_table = ((char *)elf + str_table_hdr->sh_offset);
7
8     for (int i = 1; i < elf->e_shnum; i++) {
9         Elf64_Shdr *header = (Elf64_Shdr *)((char *)elf + elf->e_shoff + elf->e_shentsize * i);
10        char *section_name = str_table + header->sh_name;
11        if (strcmp(section_name, ".ctors", 7) == 0) {
12            uint64_t *ctors = (uint64_t *)((char *)elf + header->sh_offset);
13            uint64_t num_functions = (header->sh_size / sizeof(uint64_t));
14            for (uint64_t i = 0; i < num_functions; i++) {
15                /* sysv_abi as we are on a PE executable */
16                void (*func)() = ((__attribute__((sysv_abi)) void (*)()) * ctors);
17                func();
18                ctors++;
19            }
20        }
21    }
22}
```

Los constructores globales, al compilar y enlazar, se guardan en una sección concreta denominada `.ctors`. Como cada sección está identificada por un string, tendremos que acceder a la tabla de strings del fichero ELF²⁷ como se hace en la línea 5 y 7. Luego iteramos cada tabla y obtenemos su string identificativo, comparándolo con `.ctors`.

Si encontramos la tabla `.ctors` definiremos un puntero a entero sin signo de 64 bits²⁸, luego vemos cuantos punteros caben en la sección (viendo su tamaño y el tamaño del puntero) y vamos iterando el número de punteros resultante.

Para cada puntero, definimos una función void que no toma parámetros y que está en la dirección de memoria del puntero iterador. Especificamos que la convención de llamada a la función es SYSV porque recordemos que el bootloader es un fichero PE²⁹. Finalmente llamamos a dicha función.

6.4.8. Función Principal

La función principal `main()` del bootloader es a encargada de utilizar todo lo desarrollado anteriormente en el resto de módulos para cumplir los objetivos que le han designado. Esta función actúa como un “director de orquesta” llamando a la capa más alta de abstracción de los módulos desarrollados. También se encarga de detectar errores durante la ejecución del bootloader y notificarlos mediante mensaje y valor de retorno.

Código 6.44: int main()

```

1 int
2 main(void)
3 {
4     info("started bootloader %s function from %s", __PRETTY_FUNCTION__, __FILE__);
5     info("C environment is %s", __STDC_HOSTED__ ? "hosted" : "freestanding");
6     info("Compilation datetime %s %s", __DATE__, __TIME__);
7
8     Elf64_Ehdr *elf_header = load_elf("kernel.elf");
9     if (elf_header == NULL) {
10         error("cannot load the kernel");
11         return KERNEL_LOAD_FAILURE;
12     }
13
14     efi_gop_t *gop = load_gop();
15     if (gop == NULL) {
16         error("cannot retrieve the gop");
17         return GOP_RETRIEVE_FAILURE;
18     }
19
20     Framebuffer *fb = create_fb(gop);
21     if (fb == NULL) {
22         error("cannot construct the framebuffer");
23         return FRAMEBUFFER_FAILURE;
24     }
25
26     MapInfo *map = load_memmap();

```

²⁷Sección que contiene strings del ELF, como los nombres de las secciones.

²⁸Puesto que estamos en sistemas de 64 bits.

²⁹Esto generaría una llamada con una convención distinta que la función a la que se llama.

```

27     rsdp_v2 *rsdp = load_rsdp();
28
29     PSF1_Font *font = load_psf1_font("zap-light16.psf");
30     if (font == NULL) {
31         error("font load failure");
32         return PSF1_FAILURE;
33     }
34
35     if (elf_header->e_entry == 0x0)
36         warning("kernel entry point is 0x0");
37     if (elf_header->e_entry < 0x100)
38         warning("kernel entry point is less than 0x100");
39
40     void (*_start)() = ((__attribute__((sysv_abi)) void (*)(BootArgs *))elf_header->e_entry);
41     info("jumping to kernel code at address: 0x%p", _start);
42
43     info("calling kernel global constructors");
44
45     call_ctors(elf_header);
46
47     info("finished kernel global constructors");
48
49     /* Exit UEFI Boot Services */
50     info("Exiting UEFI Boot Services before the jump");
51     if (exit_bs() > 0) {
52         error("error exiting UEFI boot services");
53         return UEFI_BS;
54     }
55
56     BootArgs args = { fb, font, map, rsdp };
57     _start(&args);
58
59     /* _start shouldn't return */
60     __asm__("hlt")
61     while (1) {
62     }
63
64     return 0;
65 }
```

Las líneas 4 – 6 se encargan de mostrar una cabecera informativa sobre la ejecución del bootloader. Muestran el nombre de la función principal y el fichero donde se encuentra, también muestran el entorno de C en el que están (*hosted* o *ffrestanding*³⁰) y la fecha y hora de la compilación.

Con las líneas 8 – 12 cargamos un archivo ELF en el path proporcionado. Esto, como ya hemos visto, indirectamente realiza la carga del fichero, chequeos de la cabecera y carga de las *phdrs*.

La obtención del puntero del GOP se realiza en las líneas 14 – 18 y posteriormente, en 20 – 24 construimos un framebuffer a partir del GOP.

Las líneas 26 y 27 obtienen tanto el mapa de memoria de EFI como el puntero al RSDP para posteriormente pasárselo al kernel.

Luego, cargamos la fuente elegida en las líneas 29 – 33. Cambiando el path de la fuente

³⁰Se ha explicado la diferencia entre ambos en el cuadro informativo de las sección 6.3

podemos cargar cualquier fuente PSF1 y así cambiar la tipografía que usa el kernel.

En la línea 40 declaramos e inicializamos un puntero a la función `_start` del kernel, que como veremos posteriormente en el sistema de construcción del kernel, está en `elf_header->e_entry`. Acto seguido llamamos a los constructores globales que deban llamarse antes de `_start` en la línea 45.

Añadir referencia

Tal como especifica UEFI, tenemos que salirnos de los UEFI boot services, tarea que realizamos en las líneas 50 – 54. La función a la que se llama la provee `posix-uefi`.

Finalmente, en las líneas 56 – 57, construimos la estructura que se le va a pasar al kernel como parámetro y llamamos a la función del kernel, perdiendo control sobre la máquina (al otorgárselo al bootloader).

La línea 60 actúa como barrera para evitar que el bootloader, por algún motivo, finalice su ejecución. El bootloader nunca puede volver de su función principal³¹.

```
(I) [startup.nsh] bootloader.efi finding started
(I) [startup.nsh] bootloader.efi found in fs0:\bootloader.efi
(I) [bootloader] started bootloader main function from /home/ecomaikgolf/Projects/os-dev-cmake/bootloader/bootloader.c
(I) [bootloader] C environment is freestanding
(I) [bootloader] Compilation datetime Oct 30 2021 02:46:29
(I) [bootloader] opening 'kernel.elf' file
(I) [bootloader] kernel.elf file opened
(I) [bootloader] allocated 26952 bytes for kernel.elf contents starting in 0x0000000000e881018
(I) [bootloader] copied kernel.elf contents to memory 0x0000000000e881018 (26952 bytes)
(I) [bootloader] closed kernel.elf
(I) [bootloader] ELF magic number is correct
(I) [bootloader] ELF is a executable object
(I) [bootloader] ELF target arch is x86_64
(I) [bootloader] ELF target is 64 bits
(I) [bootloader] ELF program header counter is non zero
(I) [bootloader] loading program header 0 at: 0x0000000000001000
(I) [bootloader] loading program header 1 at: 0x00000000000006000
(I) [bootloader] Window width: 800
(I) [bootloader] Window height: 600
(I) [bootloader] opening 'zap-light16.psf' file
(I) [bootloader] zap-light16.psf file opened
(I) [bootloader] allocated 5312 bytes for zap-light16.psf contents starting in 0x0000000000e853018
(I) [bootloader] copied zap-light16.psf contents to memory 0x0000000000e853018 (5312 bytes)
(I) [bootloader] closed zap-light16.psf
(I) [bootloader] PSF1 font correct magic header
(I) [bootloader] jumping to kernel code at address: 0x0000000000001c00
(I) [bootloader] Exiting UEFI Boot Services before the jump
```

Figura 6.9: Traza de ejecución del bootloader

Podemos ver en la Figura 6.9 una traza de la función principal que se ha analizado en esta sección.

³¹Apagar la máquina no hace que las funciones principales devuelvan. Esto no debe pasar nunca.

6.5. Conclusiones

En esta sección se ha presentado el proceso de desarrollo de un bootloader UEFI simple escrito en C. Es importante recalcar que el bootloader funciona únicamente para nuestro kernel, no es capaz de desarrollar otros kernels puesto que no hemos seguido ninguna especificación estándar.

El desarrollo de un bootloader, pese a que enriquecedor, es una tarea que consume mucho tiempo y que puede acabar eclipsando el desarrollo principal del kernel. Si el objetivo no es desarrollar un bootloader sino un kernel o sistema operativo, recomiendo encarecidamente ojear el funcionamiento de los bootloaders pero no entrar en desarrollar uno propio. Considero más beneficioso si se siguen convenios como *stivale2* [29] en el kernel y se utilicen bootloaders ya desarrollados que sigan el mismo convenio.

Como se verá a lo largo del desarrollo del kernel, *alma*, en sus versiones más recientes, no utiliza el bootloader desarrollado en esta sección. Se ha seguido el consejo que he presentado de entender un poco el funcionamiento de los bootloaders (desarrollando uno simple) pero al final el mantenimiento que necesitaba esta pieza del proyecto eclipsaba la principal (*alma*) por lo que decidí portear alma para que siguiese el convenio de *stivale2*.

El uso de *posix-uefi* me parece la mejor decisión a la hora de desarrollar un bootloader para entornos UEFI, la comodidad de una capa posix a la hora de programar es imbatible, sobretodo ante el sistema de llamadas a los servicios de UEFI que proporciona *gnu-efi*. Además la flexibilidad que proporciona *posix-uefi* a la hora de poder usar la capa de *gnu-efi* para usar funciones que no tienen equivalencia posix es idónea.

Integrar *posix-uefi* con el sistema de construcción del proyecto (*cmake*) no fué tarea sencilla pero acabó siendo correcta. Portear el script de construcción que viene con *posix-uefi* a tu propio sistema de construcción permite independizar tu código de los sistemas de construcción de otros proyectos y te permite tener toda la construcción encapsulada en tus propios *CMakeLists.txt* (en caso de usar *cmake* como sistema de construcción).

La simulación de un sistema UEFI mediante *qemu* y *edk2* es muy beneficiosa a la hora de desarrollar un bootloader destinado a hardware real. A lo largo de todo el proceso de desarrollo del bootloader no he encontrado inconsistencias entre *edk2* y la ejecución en hardware real en mi máquina (UEFI Gigabyte). Recomiendo encarecidamente usar este sistema de simulación para probar el código a desarrollar e ir realizando pruebas menos frecuentes en hardware real.

El desarrollo de un bootloader, al utilizar los servicios de UEFI, incurre en pocos errores de programación relacionados con el sistema subyacente, mayoritariamente se deben a errores lógicos de programación (sobretodo de aritmética de punteros). Algunos de los errores, pese a haberse estudiado y depurado, se mantienen en el código actual por distintos motivos, estos se presentan a continuación en la sección 6.5.1.

Todo el código que se ha presentado a lo largo de esta sección puede encontrarse, en su versión más actualizada, en <https://github.com/ecomaikgolf/alma/tree/master/bootloader>

6.5.1. Errores conocidos

En esta sección se presentarán errores o incorrectitudes del código que son conocidas por el desarrollador. En ningún caso se ha de tomar esta sección como algo malo o incorrecto, son errores que se conocen y se han estudiado sus orígenes mediante una exhaustiva depuración pero que su solución o bien es demasiado compleja o bien no entra dentro de las intenciones del proyecto.

6.5.1.1. Carga del ELF

La carga de ficheros ELF tiene el problema de que dependiendo del mapa de memoria de EFI que tengamos y el tamaño del fichero ELF podremos sobreescribir zonas de memoria al cargar las program headers.

Esto se debe a que en la función `load_phdrs` tenemos un `memcpy` en la línea 156 que copia los contenidos a la dirección virtual dada por `p_vaddr`. Como hemos dicho anteriormente, el mapa de memoria de EFI es cambiante y no podemos asegurar que donde vayamos a copiar esté vacío.

La dirección virtual inicial la establecemos en el link script del kernel y, a partir de esa dirección, se establecen las direcciones del resto del fichero ELF. Es por eso que a mayor tamaño del fichero podemos incurrir en que direcciones finales sobreescriban lugares de memoria que eran importantes.

```
Q Código 6.45: Error en void load_phdrs(...)

152 // ...
153 case PT_LOAD: {
154     info("loading program header %d at: 0x%p", i, prog_hdr->p_vaddr);
155
156     memcpy((void *)prog_hdr->p_vaddr, memory + prog_hdr->p_offset, prog_hdr->p_filesz);
157     memset((void *)(prog_hdr->p_vaddr + prog_hdr->p_filesz),
158            0,
159            prog_hdr->p_memsz - prog_hdr->p_filesz);
160     break;
161 }
162 // ...
```

La única solución coherente que hay es, a la hora de cargar el kernel, reservar mediante los servicios de UEFI un bloque de memoria suficiente para cargar las program headers y cargarlas ahí. El problema es que no basta únicamente con esto puesto que romperíamos las referencias a direcciones de memoria que tiene el ELF, tendríamos que, mediante memoria virtual, realizar las reasignaciones de direcciones de memoria pertinentes para que las referencias sigan siendo correctas.

Este error se ha dejado sin solventar porque el arreglo eclipsaba el desarrollo principal que es el del kernel, además, por las mismas fechas encontré la especificación *stivale2*, me pareció una solución tan elegante que preferí continuar con *stivale2*.

6.5.1.2. Salida de los servicios de UEFI

Salir de los servicios de UEFI provoca que algunos trozos de memoria que no se han copiado y “pertenezcan” a EFI puedan ser liberados sin especial cuidado, dejando accesos a zonas de memoria liberadas.

Q Código 6.46: Error en la salida del los UEFI services

```

100 // (...)

101 /* Exit UEFI Boot Services */
102 info("Exiting UEFI Boot Services before the jump");
103 if (exit_bs() > 0) {
104     error("error exiting UEFI boot services");
105     return UEFI_BS;
106 }
107 // (...)
```

Era un error que sucedía en condiciones extrañas, por ejemplo cuando se usaba el `startup.nsh` para arrancar el bootloader y se simulaba un sistema de ficheros FAT con el directorio actual de qemu (versiones antiguas del proyecto) no pasaba. Al dejar de usar `startup.nsh` y crear manualmente el disco desde el que se arrancaba, al dar el salto a `_start`, los valores del buffer de glifos de la fuente pasaban a ser `0xfafafafafafa....`

Me di cuenta que si no me salía de los servicios de UEFI no cambiaban los valores del buffer de glifos, por lo que el error sería que tendría que copiar el buffer de glifos a otro bloque de memoria³² o alguna solución similar.

Este error ha quedado sin solucionar porque se dejó de desarrollar el bootloader para pasar a usar stivale2. Trazar las escrituras que podían ser liberadas al salir de los servicios de UEFI no tiene mucho valor para el proyecto principal.

³²Cosa que no me acaba de encajar, porque al cargar del fichero hago un malloc, que debería de estar gestionado por EFI para no ser liberado (por lógica).

7. Desarrollo del Kernel

8. Conclusiones

9. Auxiliar

Esta sección no sirve de nada, solo es para guardarme snippets de latex.

❗ Error en X

Tengo un bug en el manejador de excepciones porque.
El siguiente código puede causar error:

⚠ Aviso sobre UEFI

Tengo un bug en el manejador de excepciones porque.
El siguiente código puede causar error:

ℹ️ Información sobre UEFI

Tengo un bug en el manejador de excepciones porque.
El siguiente código puede causar error:

```
>_ hexyl zap-light16.psf
00000000  36 04 02 10 00 00 00 3e  63 5d 7d 7b 77 77 7f 77  6...000>c] }{ww•w
00000010  3e 00 00 00 00 00 00 00  00 7e 24 24 24 24 24 24  >0000000 0~$$$$$$
00000020  22 00 00 00 00 00 00 00  01 02 7f 04 08 10 7f 20  "0000000  .....
00000030  40 00 00 00 00 00 00 00  08 10 20 40 20 10 08 00  @0000000  .. @ ..0
00000040  7c 00 00 00 00 00 00 00  10 08 04 02 04 08 10 00  |0000000  .....
00000050  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  0000000 00000000
```

Texto lateral

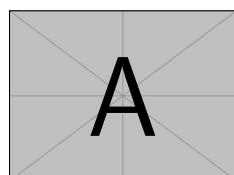
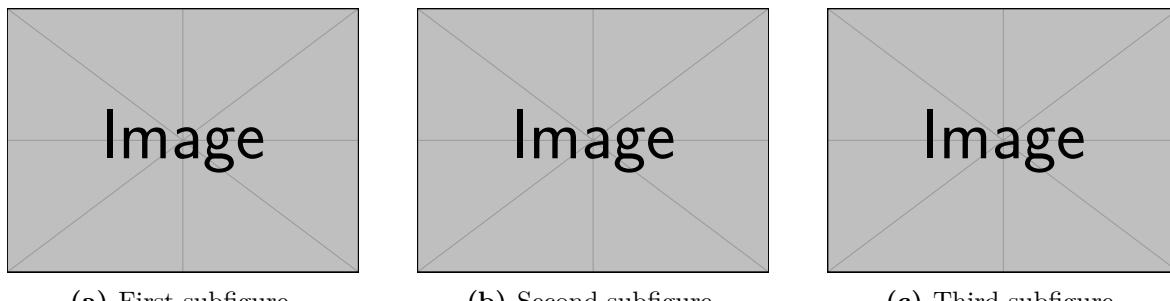


Figura 9.1: Caption



(a) First subfigure. (b) Second subfigure. (c) Third subfigure.

Figura 9.2: Subreferences in L^AT_EX.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa. Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

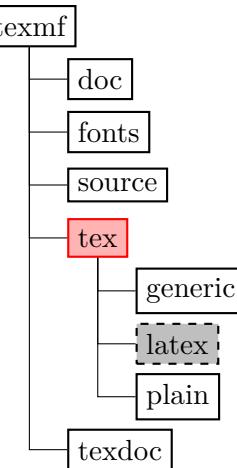
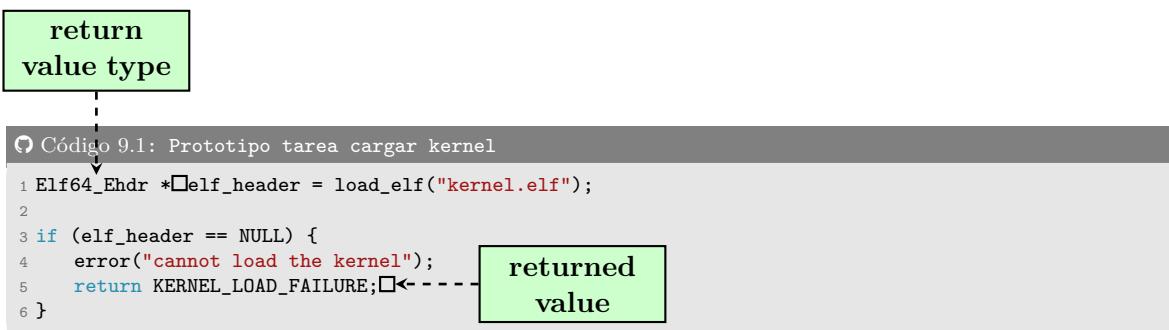


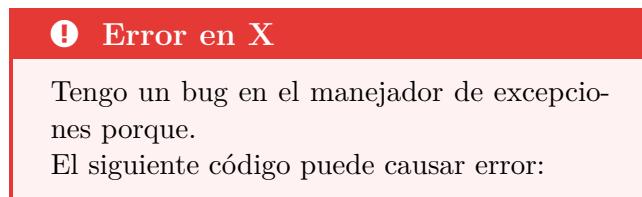
Figura 9.3: Directorio de archivos wrapped

sdakjsdalkf asdfklsadjf sdfhkl

↑ Ventajas	↓ Desventajas
<ul style="list-style-type: none"> • Motivo 1 • Motivo 2 • Motivo 2 	<ul style="list-style-type: none"> • Motivo 1 • Motivo 2 • Motivo 2



Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa. Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.



Short text

Sed feugiat. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Ut pellentesque augue sed urna. Vestibulum diam eros, fringilla et, consectetur eu, nonummy id, sapien. Nullam at lectus. In sagittis ultrices mauris. Curabitur malesuada erat sit amet massa. Fusce blandit. Aliquam erat volutpat. Aliquam euismod. Aenean vel lectus. Nunc imperdiet justo nec dolor.

Bibliografía

- [1] “QEMU,” Software, A generic and open source machine emulator and virtualizer. [Online]. Available: <https://www.qemu.org/>
- [2] “EDK II,” Software, A modern, feature-rich, cross-platform firmware development environment for the UEFI and PI specifications. [Online]. Available: <https://github.com/tianocore/tianocore.github.io/wiki/EDK-II>
- [3] “POSIX UEFI,” Software, Dependency-free POSIX compatibility layer and build environment for UEFI. [Online]. Available: <https://gitlab.com/bztsrc/posix-uefi>
- [4] “GNU EFI,” Software, Library to develop EFI applications for ARM-64, ARM-32, x86_64, IA-64 (IPF), IA-32 (x86), and MIPS platforms using the GNU toolchain and the EFI development environment. [Online]. Available: <https://sourceforge.net/projects/gnu-efi/>
- [5] “xv6,” Software, A teaching operating system developed in the summer of 2006 for MIT’s operating systems course. [Online]. Available: <https://pdos.csail.mit.edu/6.828/2012/xv6.html>
- [6] “OS/161,” Software, A simplified system used for teaching undergraduate operating systems classes. [Online]. Available: <https://www.os161.org/>
- [7] “SWEB,” Software, Educational OS. [Online]. Available: <https://github.com/IAIK/sweb>
- [8] M. Kerrisk, *The Linux programming interface : a Linux and UNIX system programming handbook*. San Francisco: No Starch Press, 2010.
- [9] “Font-formats recognized by the Linux kbd package: PSF fonts — win.tue.nl,” <https://www.win.tue.nl/~aeb/linux/kbd/font-formats-1.html>, [Accessed 24-Mar-2022].
- [10] “Tabla de bootloaders,” Wiki, Tabla comparativa de bootloaders. [Online]. Available: https://en.wikipedia.org/wiki/Comparison_of_boot_loaders
- [11] “ccache,” Software, A compiler cache. [Online]. Available: <https://ccache.dev/>
- [12] “OSDev Target Triplet,” Wiki. [Online]. Available: https://wiki.osdev.org/Target_Triplet
- [13] P. Oppermann.
- [14] “System V ABI,” Wiki. [Online]. Available: https://wiki.osdev.org/System_V_ABI#x86-64

- [15] “Graphics Output Protocol,” Wiki. [Online]. Available: <https://wiki.osdev.org/GOP>
- [16] AMD, “Replacing VGA, GOP implementation for UEFI,” https://uefi.org/sites/default/files/resources/UPFS11_P4_UEFI_GOP_AMD.pdf, 2011, [Accessed 24-Mar-2022].
- [17] “VESA,” Wiki. [Online]. Available: <https://wiki.osdev.org/VESA>
- [18] “Framebuffer - Wikipedia — en.wikipedia.org,” <https://en.wikipedia.org/wiki/Framebuffer>, [Accessed 24-Mar-2022].
- [19] “uefi::proto::console::gop::PixelFormat - Rust — docs.rs,” <https://docs.rs/uefi/0.2.0/i686-unknown-linux-gnu/uefi/proto/console/gop/enum.PixelFormat.html>, [Accessed 24-Mar-2022].
- [20] “PC Screen Font - Wikipedia — en.wikipedia.org,” https://en.wikipedia.org/wiki/PC_Screen_Font, [Accessed 24-Mar-2022].
- [21] U. E. F. I. Forum, “Unified extensible firmware interface (UEFI) specification,” 2021. [Online]. Available: https://uefi.org/sites/default/files/resources/UEFI_Spec_2_9_2021_03_18.pdf
- [22] “RSDT - OSDev Wiki — wiki.osdev.org,” <https://wiki.osdev.org/RSDT>, [Accessed 24-Mar-2022].
- [23] “XSDT - OSDev Wiki — wiki.osdev.org,” <https://wiki.osdev.org/XSDT>, [Accessed 24-Mar-2022].
- [24] “RSDP - OSDev Wiki — wiki.osdev.org,” <https://wiki.osdev.org/RSDP>, [Accessed 24-Mar-2022].
- [25] “Executable and Linkable Format - Wikipedia — en.wikipedia.org,” https://en.wikipedia.org/wiki/Executable_and_Linkable_Format, [Accessed 24-Mar-2022].
- [26] T. I. Standards, “Executable and linkable format (elf),” *Specification, Unix System Laboratories*, 2001. [Online]. Available: <http://refspecs.linuxbase.org/elf/elf.pdf>
- [27] D. Andriesse, *Practical binary analysis : build your own Linux tools for binary instrumentation, analysis, and disassembly*. San Francisco, CA: No Starch Press, Inc, 2019.
- [28] pancake, “radare2,” <https://rada.re/n/>.
- [29] “GitHub - stivale/stivale: The stivale boot protocols’ specifications and headers. — github.com,” <https://github.com/stivale/stivale>, [Accessed 27-Mar-2022].
- [30] “cmake,” Software, Family of tools designed to build, test and package software. [Online]. Available: <https://cmake.org/>
- [31] “OSDev libgcc,” Wiki. [Online]. Available: <https://wiki.osdev.org/Libgcc>

A. Anexo I