



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

Este trabajo no habría sido posible sin el apoyo y el estímulo de mi colega y amigo, Doctor Rudolf Fliesning, bajo cuya supervisión escogí este tema y comencé la tesis. Sr. Quentin Travers, mi consejero en las etapas finales del trabajo, también ha sido generosamente servicial, y me ha ayudado de numerosos modos, incluyendo el resumen del contenido de los documentos que no estaban disponibles para mi examen, y en particular por permitirme leer, en cuanto estuvieron disponibles, las copias de los recientes extractos de los diarios de campaña del Vigilante Rupert Giles y la actual Cazadora la señorita Buffy Summers, que se encontraron con William the Bloody en 1998, y por facilitarme el pleno acceso a los diarios de anteriores Vigilantes relevantes a la carrera de William the Bloody.

También me gustaría agradecerle al Consejo la concesión de Wyndham-Pryce como Compañero, el cual me ha apoyado durante mis dos años de investigación, y la concesión de dos subvenciones de viajes, una para estudiar documentos en los Archivos de Vigilantes sellados en Munich, y otra para la investigación en campaña en Praga. Me gustaría agradecer a Sr. Travers, otra vez, por facilitarme la acreditación de seguridad para el trabajo en los Archivos de Munich, y al Doctor Fliesning por su apoyo colegial y ayuda en ambos viajes de investigación.

No puedo terminar sin agradecer a mi familia, en cuyo estímulo constante y amor he confiado a lo largo de mis años en la Academia. Estoy agradecida también a los ejemplos de mis difuntos hermano, Desmond Chalmers, Vigilante en Entrenamiento, y padre, Albert Chalmers, Vigilante. Su coraje resuelto y convicción siempre me inspirarán, y espero seguir, a mi propio y pequeño modo, la noble misión por la que dieron sus vidas.

Es a ellos a quien dedico este trabajo.

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	30
6.4.3.1. Obtener el Tamaño de un Fichero	31
6.4.3.2. Cargar un Fichero en Memoria	31
6.4.4. Gráficos	32
6.4.4.1. Graphics Output Protocol	32
6.4.4.2. Framebuffer	33
6.4.4.3. Fuente	34
6.4.5. Mapa de Memoria	37
6.4.6. ACPI	40
6.4.7. ELF	40
6.5. Conclusiones	40

7. Desarrollo del Kernel	41
8. Conclusiones	43
9. Auxiliar	45
Bibliografía	49
A. Anexo I	51

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. Fuente PSF1	35
9.1. Caption	45
9.2. Subreferences in L ^A T _E X.	46
9.3. Directorio de archivos wrapped	46

Índice de tablas

6.1. Bootloaders más comunes [8]	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	29
6.12. <code>logs/stdout.h</code>	30
6.13. <code>io/file.h</code>	30
6.14. <code>uint64_t file_size(FILE *file)</code>	31
6.15. <code>void *load_file(const char *filename)</code>	31
6.16. <code>gop.h</code>	32
6.17. <code>efi_gop_t *load_gop()</code>	33
6.18. <code>framebuffer.h</code>	33
6.19. <code>framebuffer.h</code>	34
6.20. <code>font.h</code>	35
6.21. <code>PSF1_Font *load_psf1_font(const char* const filename)</code>	36
6.22. <code>PSF1_Header * get_psf1_header(const char *const memory)</code>	36
6.23. <code>uint8_t verify_psf1_header(const PSF1_Header *const header)</code>	37
6.24. <code>void *get_psf1_glyph(const char *const memory)</code>	37

6.25. memory/memory.h	37
6.26. efi_memory_descriptor_t	38
6.27. MapInfo *load_memmap()	38
6.28. void print_memmap(const MapInfo *map)	39
6.29. Prototipo tarea cargar kernel	40
9.1. Prototipo tarea cargar kernel	47

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` [9] 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
```

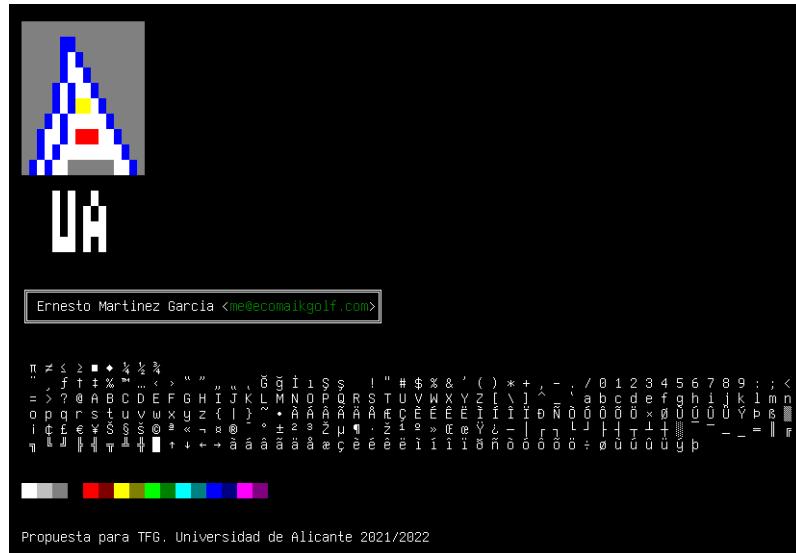


Figura 4.2: alma 20211012-e906c4c-m build

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 [10]. 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 [11] 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.

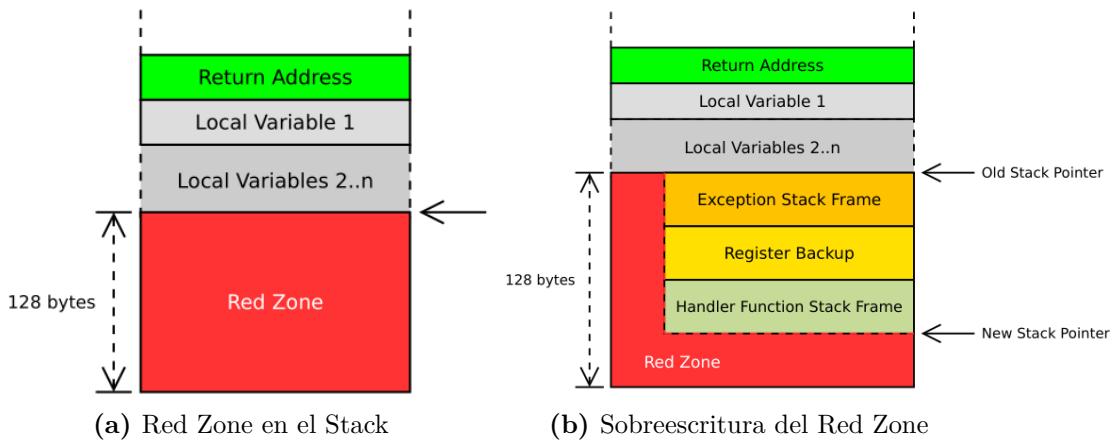


Figura 4.3: Red Zone en el Stack (<https://os.phil-opp.com>)

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 sobreescriba 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” [12]

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`.

© 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].

© 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

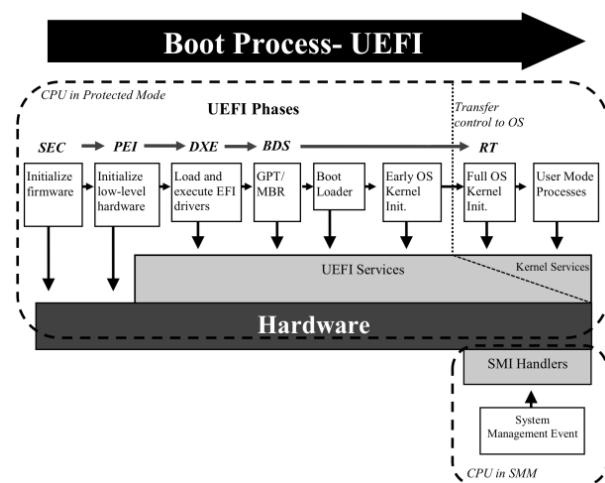


Figura 6.2: Arranque UEFI

³Se ha utilizado como ejemplo, no tiene por qué 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 [8]

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"
101  VERBATIM
102 )

```

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:

Q 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, por ejemplo:

Q 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__)
```

Su uso es similar a un `printf` de C, se han utilizado macros con argumentos variádicos para conseguir este resultado.

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 para poder reservar memoria 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.

 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

© 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 read 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.

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 `ftell()` 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.

6.4.3.2. Cargar un Fichero en Memoria

© 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  /* get file size to alloc enough space */
15  uint64_t size = file_size(file);
16
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
32  fclose(file);
33  info("closed %s", filename);
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), luego obtenemos el tamaño del fichero y reservamos memoria suficiente con `malloc()`, comprobamos si hemos reservado memoria correctamente y finalmente copiamos el archivo desde el fichero a nuestro puntero en memoria.

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.

6.4.4.1. Graphics Output Protocol

El Graphics Output Protocol [13] es el estándar utilizado por UEFI que reemplazó el estándar VESA [14] (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) {
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 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.

```

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;
14
15 Framebuffer *create_fb(const efi_gop_t *const);
```

Declaramos una estructura de tipo framebuffer con una serie de campos como un puntero a la dirección base de memoria que está mapeada a la 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 visibles¹²) por línea.

También definimos una función¹³ que nos cree un framebuffer a partir de un GOP devuelto por EFI.

© 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->bufcer_size = gop->Mode->FrameBufferSize;
13    fb->height = gop->Mode->Information->VerticalResolution;
14    fb->width = gop->Mode->Information->HorizontalResolution;
15    fb->ppscl = 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** definido anteriormente y lo rellenamos con sus homólogos del GOP de EFI.

6.4.4.3. Fuente

Con el framebuffer es posible dibujar píxeles de forma manual con el puntero **fb->base**, pero si se quiere escribir caracteres en la pantalla es necesario una fuente “bitmap”, que nos da un mapa de bits para cada carácter.

En este caso se ha elegido el formato de fuente PSF1¹⁴ que nos proporciona un mapa de

¹²No visibles son píxeles que están en la memoria pero que no se ven, y que están ahí de padding para temas de alineamiento de memoria.

¹³Esto no es C++, no dispongo de constructores

¹⁴<https://www.win.tue.nl/~aeb/linux/kbd/font-formats-1.html>

bits para carácter. La fuente usada se trata de `zap-light16.psf` y se puede encontrar en `toolchain/font`.

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 */
17     PSF1_Header *header;
18     /** Glyph buffer data */
19     void *buffer;
20 } PSF1_Font;
21
22 PSF1_Font *load_psf1_font(const char *const);
23 PSF1_Header *get_psf1_header(const char *const);
24 void *get_psf1_glyph(const char *const);
25 uint8_t verify_psf1_header(const PSF1_Header *const);

```

Se define la fuente 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 el array de glifos.

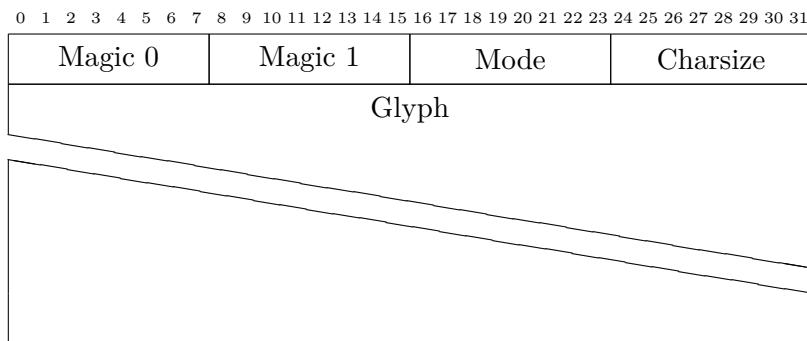


Figura 6.4: Fuente PSF1

Podemos comprobar que esto es cierto abriendo la fuente `.psf` descargada en un visor hexadecimal:

```

1 $ hexyl zap-light16.psf
2 00000000 36 04 02 10 00 00 00 3e 63 5d 7d 7b 77 77 7f 77  ••• 6000>c]••www
3 00000010 3e 00 00 00 00 00 00 00 7e 24 24 24 24 24 24 >00000000~$$$$$$

```

```

4 00000020 22 00 00 00 00 00 00 01 02 7f 04 08 10 7f 20 ....."0000000
5 00000030 40 00 00 00 00 00 00 08 10 20 40 20 10 08 00 ..@0000000 @ •0

```

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

```

❶ 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    /* Get the glyph font data
22     * https://github.com/ecomaikgolf/os-dev/raw/master/toolchain/font/zap-vga16.pdf
23     */
24    void *glyph_buffer = get_psf1_glyph(memory);
25
26    /* Create the font structure */
27    PSF1_Font *font = malloc(sizeof(PSF1_Font));
28
29    /* Not enough memory */
30    if (font == NULL)
31        return NULL;
32
33    font->header = header;
34    font->buffer = glyph_buffer;
35
36    return font;
37 }

```

En esta función en primer lugar se carga la fuente 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.

```

❷ 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 }

```

La función es una abstracción para obtener la cabecera de una fuente PSF1 a partir de memoria arbitraria.

```
④ 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 }
12 }
```

Es una abstracción para comprobar si los números mágicos que son correctos y por ende, si la fuente cargada es correcta.

```
④ 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 }
```

Sobre un bloque de memoria (supuestamente una fuente PSF1), obtiene el buffer de glifos, que se sitúa seguido de la cabecera como vemos en la figura 6.4.

6.4.5. Mapa de Memoria

Al usar EFI no disponemos de un mapa de memoria prefijado que sepamos con exactitud de antemano, nuestro mapa de memoria puede variar (por ejemplo se han reservado bloques de memoria por el uso de servicios de EFI) o iniciar 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:

```
④ 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 una estructura que actúa como interfaz del mapa de memoria de EFI, tenemos una rray 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 se basa en el estándar de UEFI y es similar al 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 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        status =
```

```

35     BS->GetMemoryMap(&map->map_size, map->map, &map->map_key, &map->descriptor_size, NULL);
36
37     if (status == 0) {
38         break;
39     } else {
40         if (i != 0)
41             warning("Memory map size try number %d", i);
42         free(map->map);
43     }
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 para la estructura del MapInfo, acto seguido llamamos a `GetMemoryMap` pero esta solo devolverá el tamaño del mapa (está programada así). En el bucle for pedimos 5 veces el mapa de memoria y comprobamos su status para ver si la lectura ha sido correcta (se recomienda hacerlo al llamar a esta función). Si el mapa es correcto lo devolvemos.

Código 6.28: void print_memmap(const MapInfo *map)

```

1 char descatypes[] [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", descatypes[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 }
```

`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.

También se puede imprimir el mapa de memoria mediante la shell de EFI:

6.4.6. ACPI

6.4.7. ELF

© Código 6.29: Prototipo tarea cargar kernel

```
1 Elf64_Ehdr *elf_header = load_elf("kernel.elf");
2
3 if (elf_header == NULL) {
4     error("cannot load the kernel");
5     return KERNEL_LOAD_FAILURE;
6 }
```

6.5. Conclusiones

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

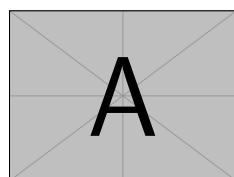
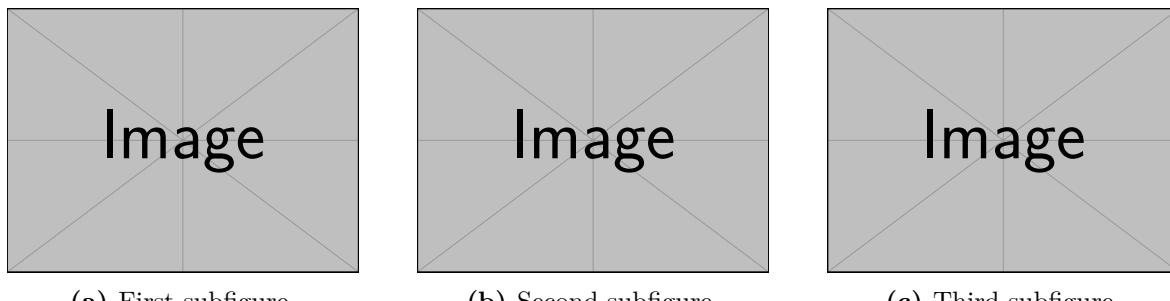


Figura 9.1: Caption



(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.

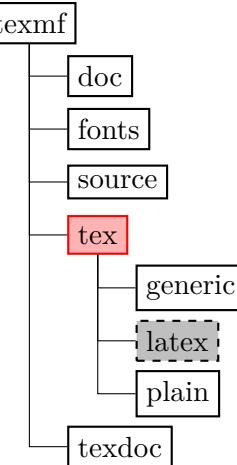
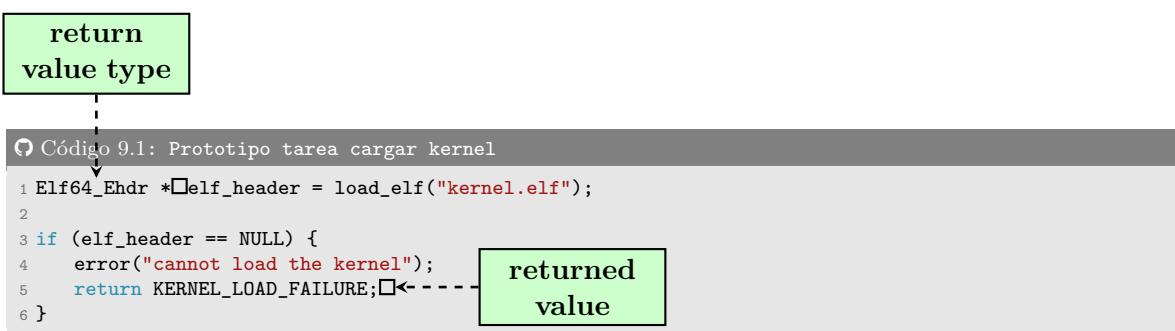


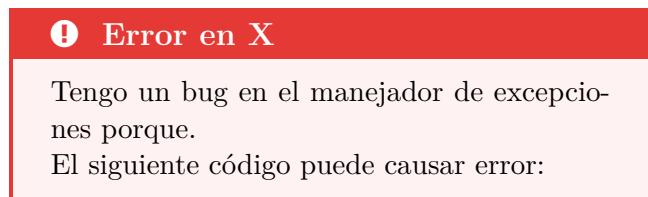
Figura 9.3: Directorio de archivos wrapped

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] “Tabla de bootloaders,” Wiki, Tabla comparativa de bootloaders. [Online]. Available: https://en.wikipedia.org/wiki/Comparison_of_boot_loaders
- [9] “ccache,” Software, A compiler cache. [Online]. Available: <https://ccache.dev/>
- [10] “OSDev Target Triplet,” Wiki. [Online]. Available: https://wiki.osdev.org/Target_Triplet
- [11] P. Oppermann.
- [12] “System V ABI,” Wiki. [Online]. Available: https://wiki.osdev.org/System_V_ABI#x86-64
- [13] “Graphics Output Protocol,” Wiki. [Online]. Available: <https://wiki.osdev.org/GOP>
- [14] “VESA,” Wiki. [Online]. Available: <https://wiki.osdev.org/VESA>

- [15] “cmake,” Software, Family of tools designed to build, test and package software. [Online]. Available: <https://cmake.org/>
- [16] “OSDev libgcc,” Wiki. [Online]. Available: <https://wiki.osdev.org/Libgcc>

A. Anexo I