



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

Abril 2022

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

TODO

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, Abril 2022

Resumen

El siguiente trabajo tiene como objetivo el desarrollo y documentación de *alma*, un kernel (*núcleo*) en C++ de funcionalidad muy reducida para manejar los recursos hardware de una máquina con arquitectura x86-64. En sus últimas versiones, *alma* puede ser arrancado en hardware real mediante un archivo `iso`, tanto en sistemas con UEFI como con BIOS.

Al tratarse de un kernel no se dispone de biblioteca estándar ni funciones base sobre las que desarrollar. Cada píxel que aparece por pantalla, cada acción de los pistones del teclado, cada reserva de memoria: está todo gestionado exclusivamente por el kernel y plasmado en este trabajo. Para mejorar la calidad del software desarrollado se han implementado desde cero funciones conocidas como `printf`, `malloc`, `scanf`, etc.

En conjunto con el kernel también se ha desarrollado un *bootloader* capaz de arrancar *alma* en máquinas que dispongan de UEFI. Se ha escrito en el lenguaje C junto con la librería de desarrollo `posix-uefi` [1] para comunicarnos con los servicios de UEFI mediante una interfaz POSIX. En las últimas versiones del proyecto, el desarrollo del bootloader ha sido reemplazado por la integración del protocolo de arranque “*stivale*” en el kernel. Ahora *alma* puede ser arrancado por cualquier *bootloader* que implemente el mismo protocolo.

También se proporciona un sistema de construcción capaz de compilar el proyecto de forma automatizada. Junto al sistema de construcción, se ha desarrollado un *script* capaz de construir `gcc` y otros programas de la *toolchain* del proyecto con las modificaciones necesarias para desarrollar un kernel. Puesto que construir *alma* es una tarea muy compleja, se ha configurado una entorno virtualizado preparado para construir el proyecto.

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 proyectos similares desarrollados por otras universidades tales como `xv6` [2], `OS/161` [3] y `SWEB` [4].

Abstract

Traducir el resumen anterior a inglés

Agradecimientos

Aparecerán en la versión final del trabajo. Ya están escritos :)

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. Propuesta	1
1.2. Motivación	2
1.3. Objetivos	3
2. Metodología	5
2.1. Gestión del Proyecto	5
2.1.1. Objetivos y Metodologías	5
2.1.2. Sistema de Control de Versiones	6
2.1.2.1. Gestión de Ramas	7
2.1.3. Repositorios	7
2.2. Tecnologías	8
2.2.1. Entorno de Desarrollo	8
2.2.2. Toolchain	9
2.2.3. Sistema de Construcción	9
2.2.4. Documentación	9
2.2.5. Entorno Virtualizado de Desarrollo	10
2.2.6. Simulación	10
2.2.7. Despliegue	11
3. Toolchain	13
3.1. Construcción de la Toolchain	14
3.2. Alma Build VM	15
3.2.1. Instalación	15
3.2.2. Ejemplo de Uso	16
3.3. Dependencias de Ejecución	17
3.4. Dependencias de Construcción	18
3.4.1. GNU Compiler Collection (gcc)	18
3.4.2. GNU Binutils	21
3.4.3. EDK II OVMF	21
3.4.4. posix-uefi	22
4. Bootloader	23
4.1. Introducción	23
4.2. Estado del Arte	26
4.3. Sistema de Construcción	29

4.4.	Desarrollo del Bootloader	33
4.4.1.	Errores	33
4.4.2.	Logs	34
4.4.3.	Ficheros	35
4.4.3.1.	Obtener el Tamaño de un Fichero	35
4.4.3.2.	Cargar un Fichero en Memoria	36
4.4.4.	Gráficos	37
4.4.4.1.	Graphics Output Protocol	37
4.4.4.2.	Framebuffer	38
4.4.4.3.	Fuente	40
4.4.5.	Mapa de Memoria	43
4.4.6.	ACPI	47
4.4.7.	ELF	49
4.4.7.1.	Cargar fichero ELF	52
4.4.7.2.	Obtener y Verificar la Cabecera ELF	52
4.4.7.3.	Cargar los Program Headers	54
4.4.7.4.	Llamar a los Constructores Globales	55
4.4.8.	Función Principal	57
4.5.	Conclusiones	60
4.5.1.	Errores conocidos	61
4.5.1.1.	Carga del ELF	61
4.5.1.2.	Salida de los servicios de UEFI	62
5. Kernel		63
5.1.	Introducción	63
5.2.	Estado del Arte	65
5.2.1.	Técnicas y Métodos de Diseño	65
5.2.1.1.	Kernels Monolíticos	65
5.2.1.2.	Microkernels	65
5.2.1.3.	Kernels Híbridos	66
5.2.1.4.	Exokernels	66
5.2.2.	Implementaciones de las syscall	66
5.2.3.	Proyectos Existentes	67
5.2.3.1.	Industria	67
5.2.3.2.	Académicos	67
5.3.	Sistema de Construcción	68
5.3.1.	Kernel	68
5.3.2.	Proyecto	75
5.4.	Desarrollo del Kernel	77
5.5.	Conclusiones	77
6. Conclusiones		79
7. Auxiliar		81
Bibliografía		85

Índice de figuras

1.1. Proyecto de TFG	1
2.1. Entorno de desarrollo utilizado para el proyecto: Artix Linux + doom-emacs como editor y ccls como LSP	8
2.2. Resultado de la generación de la documentación con doxygen	9
2.3. Ejemplo de despliegue en hardware real de alma	11
3.1. Alma Build VM	15
3.2. alma con Wireshark en la “alma build VM”	16
3.3. alma ejecutado en la “alma build VM”	16
3.4. alma con gdb en la “alma build VM”	16
3.5. Red Zone en el Stack (https://os.phil-opp.com)	19
4.1. Ejemplos de bootloaders	23
4.2. Arranque UEFI	24
4.3. Bootloaders comunes	26
4.4. Funcionamiento SEEK_SET, SEEK_CUR y SEEK_END [5]	36
4.5. Formato Píxel GOP	39
4.6. Fuente PSF1 [6]	40
4.7. Formato ELF	49
4.8. Formato Código de Error	53
4.9. Traza de ejecución del bootloader	59
5.1. Arquitectura del Kernel Linux [7]	64
7.1. Caption	81
7.2. Subreferences in L ^A T _E X.	82
7.3. Directorio de archivos wrapped	82

Índice de tablas

4.1. Bootloaders más comunes [8]	26
5.1. Kernels mayormente utilizados en la industria	67
5.2. Kernels Académicos con cierto grado de madurez [9] [10]	67

Índice de Códigos

3.1. .gitmodules	14
3.2. Compilación de gcc 11.x	20
3.3. Compilación de binutils 2.37	21
3.4. Compilación de EDK II OVMF	22
3.5. Compilación de posix-uefi	22
4.1. Sintaxis de gnu-efi	28
4.2. Sintaxis de posix-efi	28
4.3. Makefile de posix-efi	29
4.4. CMakeLists.txt bootloader I	29
4.5. CMakeLists.txt bootloader II	29
4.6. CMakeLists.txt bootloader III	30
4.7. CMakeLists.txt bootloader IV	30
4.8. CMakeLists.txt bootloader V	31
4.9. CMakeLists.txt bootloader VI	31
4.10. err_values.h	33
4.11. err_values ejemplo	34
4.12. logs/stdout.h	34
4.13. io/file.h	35
4.14. uint64_t file_size(FILE *file)	35
4.15. void *load_file(const char *filename)	36
4.16. gop.h	37
4.17. efi_gop_t *load_gop()	37
4.18. framebuffer.h	38
4.19. framebuffer.h	39
4.20. font.h	40
4.21. PSF1_Font *load_psf1_font(const char* const filename)	41
4.22. PSF1_Header * get_psf1_header(const char *const memory)	42
4.23. uint8_t verify_psf1_header(const PSF1_Header *const header)	42
4.24. void *get_psf1_glyph(const char *const memory)	42
4.25. memory/memory.h	44
4.26. efi_memory_descriptor_t	44
4.27. MapInfo *load_memmap()	44
4.28. void print_memmap(const MapInfo *map)	46
4.29. rsdp_v1	47
4.30. rsdp_v2 *load_rsdःp()	47
4.31. rsdp_v2 *load_rsdःp()	48

4.32. efi_guid_t	48
4.33. bool compare_guid(efi_guid_t *g1 efi_guid_t *g2)	49
4.34. ELF Header	49
4.35. ELF Program header	50
4.36. ELF Section header	51
4.37. Elf64_Ehdr *load_elf(const char *const filename)	52
4.38. Elf64_Ehdr *get_elf_header(char *memory)	52
4.39. uint8_t verify_elf_headers(const Elf64_Ehdr *const elf_header)	53
4.40. void load_phdrs(const Elf64_Ehdr *const elf_header)	54
4.41. Ejemplo de función constructora	55
4.42. Ejemplo de objeto global con constructor	55
4.43. void call_ctors(Elf64_Ehdr *elf)	56
4.44. int main()	57
4.45. Error en void load_phdrs(...)	61
4.46. Error en la salida del los UEFI services	62
 5.1. CMakeLists.txt kernel I	68
5.2. CMakeLists.txt kernel II	68
5.3. CMakeLists.txt kernel III	69
5.4. CMakeLists.txt kernel IV	69
5.5. CMakeLists.txt kernel V	69
5.6. CMakeLists.txt kernel VI	69
5.7. kernel.ld [31]	69
5.8. CMakeLists.txt kernel VII	70
5.9. CMakeLists.txt kernel VII	71
5.10. CMakeLists.txt kernel VIII	72
5.11. CMakeLists.txt kernel IX	72
5.12. CMakeLists.txt kernel X	72
5.13. crtI.asm	73
5.14. crtN.asm	73
5.15. CMakeLists.txt proyecto I	75
5.16. CMakeLists.txt proyecto II	75
5.17. CMakeLists.txt proyecto III	75
5.18. CMakeLists.txt proyecto III	76
5.19. CMakeLists.txt proyecto IV	76
5.20. CMakeLists.txt proyecto V	76
5.21. CMakeLists.txt proyecto VI	76
5.22. CMakeLists.txt proyecto VII	77
5.23. CMakeLists.txt proyecto VIII	77
5.24. CMakeLists.txt proyecto IX	77
5.25. CMakeLists.txt proyecto X	77
 7.1. Prototipo tarea cargar kernel	83

1. Introducción

Este capítulo es el encargado de introducir el proyecto realizado, desde su propuesta 1.1 hasta el trabajo relacionado ??, pasando por la motivación 1.2 y los objetivos 1.3 del proyecto. El objetivo de este capítulo es que el lector disponga de un contexto general del proyecto para poder entender su realización.

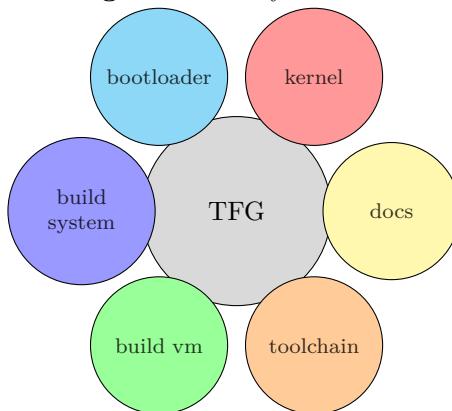
1.1. Propuesta

Se propone programar y documentar *alma*: un kernel académico en C++ para máquinas con arquitectura x86 de 64 bits.

El Trabajo de Fin de Grado a realizar, a parte del kernel (*alma*), que es el trabajo principal, se compone de otros subproyectos necesarios para su realización (Figura 1.1). Cada subproyecto tiene unos propósitos propios que casan con necesidades al desarrollar un kernel, cada subproyecto puede ser una propuesta de TFG propia:

El bootloader (utilizado en las versiones iniciales del trabajo) propone programar y documentar un bootloader simple en C capaz de cargar *alma* en máquinas con UEFI. El sistema de construcción (*build system*) propone diseñar, con cmake, un sistema de construcción capaz de compilar *alma* (y, opcionalmente, el bootloader) de forma automatizada y generar un archivo *iso*. La máquina virtual de *alma* (*build vm*) propone crear un entorno de desarrollo virtualizado para *alma*. La cadena de herramientas (*toolchain*) propone configurar y compilar las herramientas necesarias para desarrollar un kernel de forma automatizada. Y la documentación (*docs*) propone documentar todo el proyecto, mediante doxygen y esta misma memoria (LATEX).

Figura 1.1: Proyecto de TFG



1.2. Motivación

El ámbito de los sistemas operativos y núcleos es una disciplina ampliamente trabajada por desarrolladores de sistemas 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 introducción 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” (IAIK) de la Universidad Tecnológica de Graz tienen la asignatura *INP32512UF (Operating Systems)* donde trabajan con *SWEB* [4], un sistema operativo de la universidad sobre el que los alumnos tienen que trabajar con *mutexes*, 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* [2] desarrollado por el Instituto Tecnológico de Massachusetts, donde se imparte la asignatura *6.S081 (Operating System Engineering)* para el que se desarrolló el núcleo.

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. Considero que la Universidad puede beneficiarse al disponer de literatura relacionada con este campo en el repositorio institucional.

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). 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. Por ello, este 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.

Finalmente, el motivo principal del desarrollo de este proyecto, *alma*, es para beneficio propio. Desarrollar un kernel me aporta conocimientos técnicos sobre el funcionamiento interno de los sistemas operativos aplicables a cualquier campo de la informática. También me permite aprender el conocimiento base necesario del campo de la seguridad de sistemas de bajo nivel, el área en el que me quiero especializar. Para aprender a securizar y/o romper la seguridad de, por ejemplo, los sistemas operativos, es crucial conocer el funcionamiento interno de sus mecanismos, mecanismos que aprenderé mediante este proyecto.

1.3. 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 arquitecturas *x86-64* relacionadas con el desarrollo del núcleo. El kernel a desarrollar se denomina *alma* y se escribirá en el lenguaje *C++*. A su vez, en los estados iniciales del proyecto, se desarrollará un bootloader escrito en el lenguaje *C* capaz de arrancar *alma* en entornos UEFI.

Los objetivos concretos de funcionalidad del núcleo no están preestablecidos debido a la naturaleza del proyecto. Establecer objetivos de antemano sin haber desarrollado proyectos similares con anterioridad es contraproducente. Establecer objetivos aleatorios en base a funcionalidades a las que estamos acostumbrados lo único que hará es que encuentre que la mayoría de ellas son imposibles de desarrollar. El objetivo de “que el núcleo sea capaz de abrir una página web”, pese a que intuitivo e idóneo, es fantasioso. Al no conocer la dificultad de los objetivos que voy a desarrollar (entrada de teclado, entrada de ratón, etc) he desechar la idea de una lista de objetivos estática y premeditada e incorporar unos objetivos dinámicos y muy variables a la hora de desarrollar *alma*. Esta metodología se ha explicado a fondo en la sección 2.1.1.

También se ha de comentar que los objetivos del proyecto, como concepto general, han ido sufriendo variaciones a lo largo de su desarrollo. Un claro ejemplo es la sustitución del bootloader desarrollado por el protocolo *stivale2* en el núcleo, que permite que *alma* sea cargado por cualquier bootloader, no solo el desarrollado. Otro claro ejemplo es que, anteriormente, el proyecto solo contemplaba arranques de *alma* en entornos virtualizados, gracias al avance de *alma* ahora se contemplan arranques en hardware real también.

En el aspecto del aprendizaje, el objetivo de este trabajo es aprender sobre el desarrollo de sistemas operativos, núcleos y bootloaders. También se busca mejorar las habilidades con los sistemas de construcción (*cmake*), sistemas de documentación (*doxygen*), compiladores y enlazadores avanzados (*gcc* y *ld*), máquinas virtuales (*virtualbox* y *qemu*) y sistemas de control de versiones (*git*).

2. Metodología

Este capítulo presenta una visión general de los objetivos que se han tenido a lo largo del proyecto y las metodologías que se han usado para conseguirlos. En la sección 2.1 del capítulo se verá cómo se ha gestionado el proyecto, desde sus objetivos en la sección 2.1.1 hasta la gestión del código en la 2.1.2. En las secciones 2.2 y ?? se verán las tecnologías usadas para desarrollar y simular el proyecto, incluyendo el entorno donde ha sido probado y desarrollado.

2.1. Gestión del Proyecto

En esta sección del capítulo se explica, de forma general, las directrices que se han seguido a la hora de gestionar el proyecto y el código desarrollado. También se explica dónde se ha almacenado el proyecto y qué medidas se han tomado para garantizar su integridad.

2.1.1. Objetivos y Metodologías

El proyecto ha tenido una gestión de objetivos y metodologías bastante tradicional desde un principio. Se ha optado por desarrollar sin objetivos preestablecidos en cuanto a las funcionalidades del kernel y seguir un sistema de pruebas de concepto explicado a continuación.

No tener objetivos funcionales preestablecidos no ha significado la ausencia de objetivos en el proyecto, solo que estos han sido **variables** y cortoplacistas. Desde un principio descarté la idea de establecer objetivos estáticos y largoplacistas, el desarrollar un kernel por primera vez me impedía tener objetivos inamovibles.

La estrategia que he seguido para desarrollar funcionalidades en el núcleo es escribir una lista variable de módulos a añadir. Los módulos que aparecían en la lista eran completamente flexibles, podrían acabar siendo incluídos o descartados. Estos items de la lista se añadían basandome en funcionalidades comunes de los sistemas operativos actuales, por ejemplo, disponer de entrada de teclado.

Al iniciar un bloque grande de desarrollo para implementar un módulo nuevo, por ejemplo, el módulo de teclado PS2, lo que hacía era investigar proyectos ya existentes y valorar, en el

menor tiempo posible, si era una idea viable y sensata para el proyecto. Esto era importante para empezar a desarrollar la idea, ya que hay funcionalidades que pueden ser aparentemente sencillas y luego ser muy complejas de desarrollar, por ejemplo, una interfaz gráfica.

Una vez determinada que la idea era viable para desarrollarla, empezaba a trabajar en una “Prueba de Concepto”. Las pruebas de concepto eran fragmentos de código mínimo que realizaban la funcionalidad de más bajo nivel del módulo que estaba desarrollando. Por ejemplo, en el módulo de teclado PS2, era ser capaz de recibir una interrupción con el código que indica que pistón se ha pulsado o levantado. A lo largo de esta etapa realizaba anotaciones de los temas que trataba en cada prueba de concepto, por ejemplo, ACPI¹.

Con una prueba de concepto desarrollada podía asegurar que la idea era viable al 100%. Acto seguido refactorizaba el código de la prueba de concepto en un código más elegante y posteriormente empezaba a desarrollar funcionalidades de alto nivel, por ejemplo en este caso, la función `scanf`.

Finalmente, una vez se tenía el código final escrito, procedía a documentarlo mediante anotaciones doxygen². En este punto el código se consideraba acabado, aunque estaba sujeto a futuros cambios y refactorizaciones.

Cabe mencionar que en mi forma de trabajar no se requería finalizar un bloque de desarrollo para poder empezar otro, podía tener un módulo pendiente de documentar y empezar la investigación de otro módulo, si así lo consideraba. Esto lo hacía porque los tiempos requeridos para cada paso de la implementación de un módulo son distintos, por lo que había casos en los que no tenía tiempo suficiente como para refactorizar una prueba de concepto pero sí para realizar una investigación en proyectos ya existentes.

2.1.2. Sistema de Control de Versiones

La gestión del código del proyecto ha sido un aspecto fundamental debido a la gran cantidad de código desarrollado (10.000 líneas). Para la gestión del código desarrollado se ha usado un Sistema de Control de Versiones (SCV), el SCV elegido ha sido `git`.

`git` es una herramienta de gestión de versiones desarrollada inicialmente por Linus Torvalds para el kernel Linux. `git` ha ido evolucionando a lo largo de los años hasta ser la herramienta dominante en los sistemas de control de versiones.

La herramienta nos permite almacenar el estado de nuestro código en un repositorio e ir aplicando cambios incrementales a este. Todos los cambios se quedan almacenados en el propio repositorio para poder recuperarlos cuando sea necesario. `git` tiene otras funcionalidades como ramas, submódulos, merges, etc.

La elección de `git` como herramienta de SCV ha permitido depurar de forma más rápida

¹<https://ecomaikgolf.com/programming/acpi.html>

²Similares a javadoc

errores en el código introducidos en commits antiguos. La herramienta `bisect` de git se ha empleado en incontables ocasiones para detectar commits que introducían errores en el kernel. Un ejemplo de uso de la herramienta en el proyecto es el siguiente:

```
>_ git bisect start
     git bisect good SHA1
     git bisect run alma_check.sh
```

2.1.2.1. Gestión de Ramas

Un aspecto importante a la hora de usar un sistema de control de versiones es decidir como se van a utilizar las ramas del código.

La gestión de ramas se decidió que fuese lo más simple posible, puesto que el proyecto desarrollado al ser un trabajo de fin de grado, no iba a ser colaborativo. El que la gestión de ramas fuese lo más simple posible se hizo para poder darle más importancia al desarrollo de código. Al no ser colaborativo ni ser un proyecto de millones de líneas de código no siento la necesidad de disponer de múltiples ramas en distintos estadios del desarrollo. He optado por disponer de una única rama, `master`, y que sea completamente lineal.

El código presente en la rama `master` representa la versión más actualizada del proyecto. El código presente en `master` es solo estable si así se indica. El versionado del proyecto corresponde con el hash de cada commit, no se tiene consideraciones especiales³ importantes.

2.1.3. Repositorios

Para almacenar el proyecto en la nube y disponer de una plataforma de consulta y acceso al código, el proyecto utiliza tres repositorios *on-line*⁴: *Github*, *Gitlab* y un repositorio “barebone” almacenado en un servidor personal *OpenBSD*.

Mantener tres repositorios simultáneos se ha hecho por mantener una máxima redundancia en caso de fallo de uno de los servicios. También se optó por almacenarlo en un servidor personal para garantizar el acceso a mi código en caso de fallo generalizado.

```
>_ git remote get-url --push --all origin
git@github.com:ecomaikgolf/alma.git
git@gitlab.com:ecomaikgolf/os-dev.git
git@ecomaikgolf.com:alma.git
```

³En algunos casos añado la fecha en formato YYYYMMDD para que sea más claro.

⁴El principal de los tres es Github: <https://github.com/ecomaikgolf/alma>

2.2. Tecnologías

En esta sección se va a tratar las tecnologías con las que se han llevado a cabo el proyecto y con las que se ha trabajado a lo largo del desarrollo. También se verá con qué tecnologías se ha simulado el proyecto y en qué hardware se ha probado el proyecto. Finalmente se presentará de una forma muy simple la arquitectura para la que se va a desarrollar el kernel: x86-64.

2.2.1. Entorno de Desarrollo

El entorno de desarrollo es el ecosistema software en el que se ha desarrollado el proyecto, el sistema, las tecnologías, el entorno de desarrollo, etc.

El proyecto ha sido desarrollado en un sistema GNU/Linux, concretamente en la distribución *Artix Linux*⁵. Véase la Figura 2.1 para ver una imagen del sistema en el que se ha desarrollado el proyecto.

The screenshot shows a terminal window with several panes:

- Left pane:** A file browser showing the directory structure of the kernel source code. It includes files like `kernel.cpp`, `alma.h`, `alma.c`, `almaLists.txt`, `lincfg.h`, and `lincfg.c`.
- Middle pane:** A git log showing recent commits. Some of the commit messages are:
 - 289f5e6 origin/master PoC circular buffer in fast renderer (bug scroll)
 - 7f63a57 added simple/fast rendering
 - b686e66 kernel::framebuffer with selectable renderer
 - d0dd431 simple pushColor/popColor added
 - 2929f4f major organizational rewrite of the renderer stack
 - 6d726bf redundant variable removed
 - 1c82097 fix framebuffer buffer size (errata)
 - a7d4b49 PoC multi-buffer fast renderer (not working)
 - f5c0303 change libstdc++ lib to lib in make
- Right pane:** Two code editors side-by-side, both showing parts of `kernel.cpp`.
 - The top editor shows the beginning of the file, including the `main` function and some global variable initializations.
 - The bottom editor shows the middle section of the file, dealing with keyboard initialization and memory management.

Figura 2.1: Entorno de desarrollo utilizado para el proyecto: Artix Linux + doom-emacs como editor y ccls como LSP

El editor donde ha sido desarrollado la mayoría de código ha sido *emacs*, concretamente *doom-emacs* con *ccls* como LSP y *magit* como interfaz *git*.

⁵<https://artixlinux.org/>

2.2.2. Toolchain

La toolchain representa todo el software que se ha necesitado para construir el proyecto. Puesto que el proyecto necesita de una toolchain muy extensa con modificaciones concretas, esta se ha explicado con detalle en un capítulo aparte, véase el Capítulo 3.

2.2.3. Sistema de Construcción

Para construir el proyecto y generar el archivo `iso` final se ha utilizado `cmake`. CMake es un sistema de construcción de proyectos que actúa como `autoconf` y `automake` juntos, permite especificar todo su comportamiento en un archivo `CMakeLists.txt`.

2.2.4. Documentación

La documentación del código en C++ se realiza con la herramienta doxygen. Doxygen nos permite generar documentación para código C++, C, C, PHP, Java, Python, etc. Podemos generar páginas HTML (versión escogida en el proyecto, véase la Figura 2.2) o un manual de referencia tanto en latex como en PDF, MS-Word, RTF, etc.

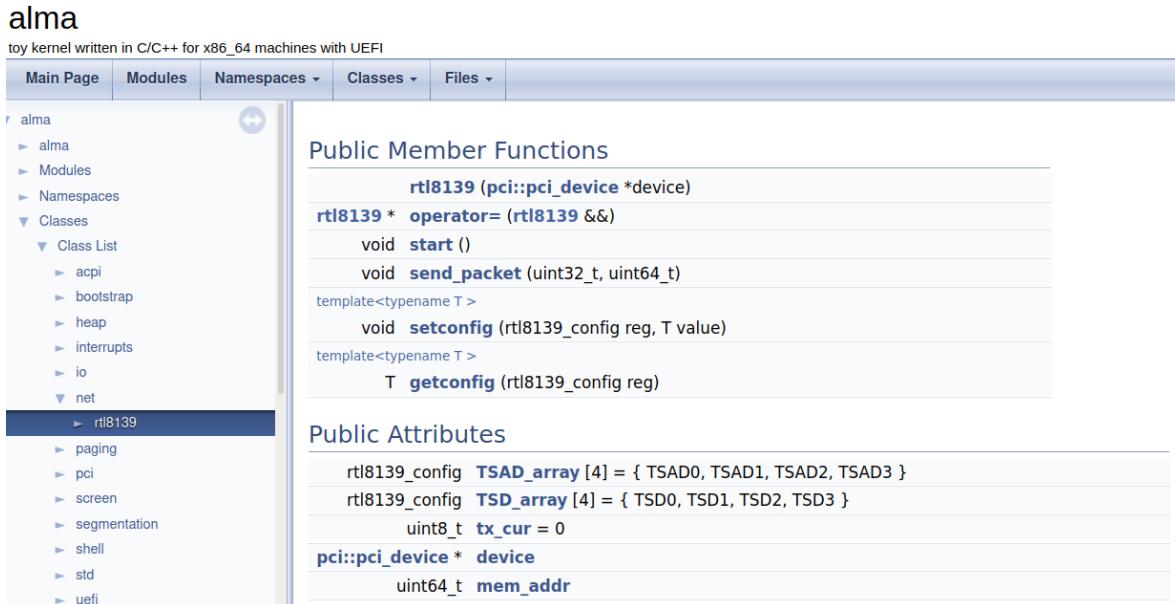


Figura 2.2: Resultado de la generación de la documentación con doxygen

La herramienta dispone de una sintaxis especial que usaremos a la hora de comentar el código en C++. En los comentarios podremos poner anotaciones como `@warning` (y muchas más) que tendrán tratados especiales en la salida.

2.2.5. Entorno Virtualizado de Desarrollo

Como construir el proyecto y compilar la toolchain es una tarea que depende mucho de la distribución que se tenga y es un proceso muy costoso, se ha preparado una máquina virtual lista para compilar el proyecto. Se puede encontrar una explicación detallada de como descargar y utilizar esta máquina en la sección 3.2.

La máquina virtual está en formato Open Virtualization Format (OVA) y se puede importar en Virtualbox. Virtualbox es un software de virtualización que nos permite desplegar sistemas completos emulados dentro de nuestro sistema (host). Gracias a que los formatos OVA vienen comprimidos y a una preparación especial a la imagen de la máquina virtual se ha conseguido reducir el peso al máximo, 6.31GB.

2.2.6. Simulación

Para simular un entorno de una máquina x86-64 y poder configurar el hardware del que dispone se utiliza `qemu` [11]. qemu es un emulador y virtualizador open source desarrollado por el “QEMU team” (Peter Maydel, et al). La versión de qemu utilizada en el proyecto es la 6.1.0.

qemu permite configurar cada aspecto de la máquina a emular, por ejemplo permite seleccionar el modelo concreto de tarjeta de red, lo que nos permite emular la *RTL8139* usada en el proyecto. Otro de los beneficios que nos proporciona qemu es su configuración por línea de comandos, que permite automatizar la tarea de arrancar la máquina virtual por parte del sistema de construcción (con targets auxiliares). *alma* se ejecuta en un entorno simulado mediante el siguiente comando:

>—

```
qemu-system-x86_64 -machine q35 -cpu qemu64 -m 256M -bios bios.bin
  ↳ -netdev user,id=user.0 -device rtl8139,netdev=user.0,
  ↳ mac=ca:fe:c0:ff:ee:00 -object filter-dump,id=f1,netdev=user.0,
  ↳ file=log.pcap -boot d -cdrom alma.iso
```

Usar qemu para simular el proyecto nos aporta dos grandes cosas. Una es que tenemos un entorno constante y muy acotado, si nos aseguramos que funciona en qemu podremos probarlo en otro qemu sin mayores miedos a cambios que provoquen fallos en el kernel. El otro es la facilidad y rapidez que tenemos para probar el software, mucho más cómodo que tener que copiar el iso a un USB y arrancar una máquina física con el kernel (además de posibles errores al cambiar de hardware entre máquinas distintas).

Para disponer de un arranque UEFI en qemu se ha usado el entorno de desarrollo UEFI de EDK II. Se ha compilado un OVMF de EDK II y se le pasa a qemu como parámetro en `-bios`, de esta forma qemu usará el OVMF compilado y arrancará con UEFI.

2.2.7. Despliegue

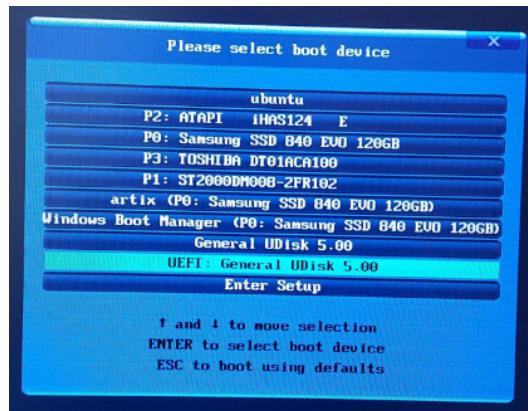
El proyecto inicialmente se planteó como para ser probado únicamente en entornos simulados, pero en las últimas versiones del proyecto se planteó que también pudiese ser arrancado en máquinas reales.

Pese a que el despliegue también se puede considerar la ejecución mediante qemu explicada anteriormente en la sección 2.2.6, trataremos esta sección como el despliegue en hardware real que se ha hecho.

El despliegue se ha hecho en un ordenador x86-64 con CPU “Intel i7-4770K”, placa base “Gigabyte Z87X-UD3H” con soporte UEFI dual BIOS (tenemos tanto BIOS como UEFI), 16GB de memoria RAM, pantalla dual (importante para el kernel) de resolución 1080p y un teclado “Packard Bell” con entrada PS2.

Al realizar el despliegue lo que se hace es quemar la imagen iso en un USB genérico, enchufarlo en el ordenador mencionado y arrancar el USB en el menú de arranque (Figura 2.3a), acto seguido arrancará el bootloader y ejecutará alma (Figura 2.3b).

Poner
RTL8139 si
acabo com-
prándola



(a) Arranque del ordenador mostrado

```
$ help
help, echo, whoami, shell, clear, pci, getpage, getmac, getphys, map, unmap, get, set, printmem, uefimap, printpfa, checknet, :
$ pci
* 8086 - 8c00 (0, 0, 0)
* 8086 - 8c01 (1, 0, 0)
* 8086 - 8c02 (2, 0, 0)
* 8086 - 8c0c (3, 0, 0)
* 8086 - 8c31 (20, 0, 0)
* 8086 - 8c3a (22, 0, 0)
* 8086 - 153b (25, 0, 0)
* 8086 - 8c2d (26, 0, 0)
* 8086 - 8c2b (27, 0, 0)
* 8086 - 8c10 (28, 0, 0)
* 8086 - 8c10 (28, 0, 4)
* 8086 - 8c11 (28, 0, 5)
* 8086 - 8c25 (29, 0, 0)
* 8086 - 8c44 (31, 0, 0)
* 8086 - 8c02 (31, 0, 2)
* 8086 - 8c22 (31, 0, 3)
* 10de - 1c03 (0, 1, 0)
* 10de - 10f1 (0, 1, 1)
* 8086 - 244e (0, 3, 0)
* 1b4b - 9172 (0, 5, 0)
$ screen
1024x768
```

(b) alma desplegado en hardware real

Figura 2.3: Ejemplo de despliegue en hardware real de alma

3. Toolchain

En este capítulo se muestran las herramientas necesarias para poder compilar y ejecutar el proyecto, junto a las modificaciones que necesitan algunas herramientas como el compilador para poder construir un kernel. También se mostrará como trabajar sin necesidad de compilar estas herramientas, haciendo uso de la “alma build vm” presentada en la sección 3.2. En la sección 3.3 se listan las dependencias para ejecutar el proyecto y cómo hacerlo, mientras que en la sección 3.4 se listan las dependencias que hay que instalar del gestor de paquetes del sistema para construir el proyecto. Y finalmente, en las secciones 3.4.1 3.4.2 3.4.3 3.4.4 se muestra el software del que depende el proyecto, cómo se ha gestionado su autoinstalación y qué modificaciones ha necesitado.

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

alma requiere de una extensa toolchain para poder ser compilado. El proyecto no se trata de un fuente común de C++ que puede ser construido mediante el compilador instalado en la mayoría de sistemas como gcc, clang, etc. *alma* necesita de un compilador “especial” que construya para arquitecturas x86-64 agnóstico del sistema instalado, ya que por defecto el compilador que tenemos instalado en las máquinas Linux (por ejemplo) confía en que el código se ejecute en entornos Linux.

Además, *alma* no puede usar compiladores comunes aunque cumplan el requisito del párrafo anterior, se necesitan modificaciones durante la construcción del compilador específicas para poder garantizar un correcto funcionamiento. Estos requisitos del compilador, junto a su justificación, se pueden observar en la sección 3.4.1.

Construir la toolchain necesaria para compilar *alma* también requiere de paquetes preinstalados en la máquina del cliente, paquetes que se listarán y se explicará su instalación en la sección 3.4. Si el usuario no quisiese tener que instalar todos estos paquetes podría optar por instalar solo dos para poder probar el proyecto (sección 3.3) o usar el entorno virtualizado de desarrollo (sección 3.2).

Es por esto que el proyecto necesita de un capítulo específico para la *toolchain*. Configurar y construir la *toolchain* del proyecto no ha sido tarea fácil, y menos conseguir que se construya de forma automática.

3.1. Construcción de la Toolchain

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 submódulos¹ de git:

```
⌚ Código 3.1: .gitmodules
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
17 [submodule "toolchain stivale"]
18  path = toolchain stivale
19  url = https://github.com/stivale/stivale.git
20  ignore = all
21 [submodule "toolchain limine"]
22  path = toolchain limine
23  url = https://github.com/limine-bootloader/limine.git
24  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. 3.4). Una vez se hayan instalado dichas dependencias se puede construir la toolchain de la siguiente manera:

```
>_ 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 sistema².

¹Se sabe que no es la forma más rápida de descargar, se descarga el repositorio entero y no el código necesario. Github debería soportar hacer `git archive` a las URLs de los repositorios.

²Se tomó esta decisión con el fin de mantener el sistema host limpio

3.2. Alma Build VM

Puesto que el proceso de construcción de la toolchain es un proceso lento y costoso, se ha diseñado un entorno de desarrollo virtualizado mediante Virtualbox. Este entorno (máquina virtual) es ultraligera y dispone de todas las herramientas necesarias para trabajar con *alma* preconfiguradas. También dispone de herramientas y software auxiliar, como iconos para trabajar directamente con *alma* mediante un click.



Figura 3.1: Alma Build VM

La máquina virtual es una versión modificada de Xubuntu 20.04³ con la toolchain de *alma* precompilada, VSCodium preconfigurado para poder editar y compilar el proyecto y *ccache* [12] a nivel de sistema. La máquina se distribuye en formato Open Virtualization Format (OVA) con un peso de 6.3G.

3.2.1. Instalación

Este entorno de desarrollo virtualizado se puede descargar desde <https://github.com/ecomaikgolf/alma#virtual-machine-method-1> e importar y ejecutar con el siguiente comando de virtualbox:

```
>_ vboxmanage import alma.ova
vboxmanage startvm "alma"
```

³<https://virtual-machines.github.io/Xubuntu-VirtualBox/>

3.2.2. Ejemplo de Uso

Utilizar la máquina virtual es muy sencillo, podemos usar la terminal para construir alma manualmente o usar los iconos que se nos proporcionan.

Si pulsamos “Run” se construirá y ejecutará alma, si alguno de los pasos falla (como el de construcción), lanzará una notificación y pausará la ejecución. Si todo funciona lanzará alma (Figura 3.3).

Si pulsamos “Debug” se construirá y ejecutará alma con un servidor de GDB en qemu, también abrirá una sesión de GDB y se conectará a ella (Figura 3.4).

Si pulsamos “Network” se construirá y ejecutará alma, también se abrirá Wireshark y se podrán inspeccionar los paquetes de red enviados (Figura 3.2).

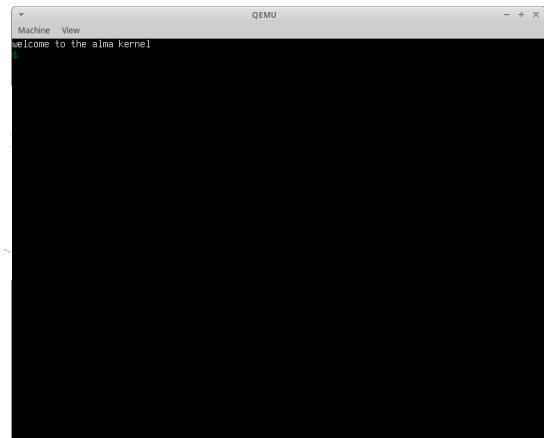


Figura 3.3: alma ejecutado en la “alma build VM”

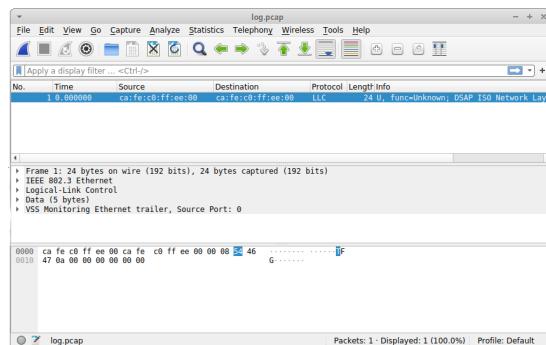


Figura 3.2: alma con Wireshark en la “alma build VM”

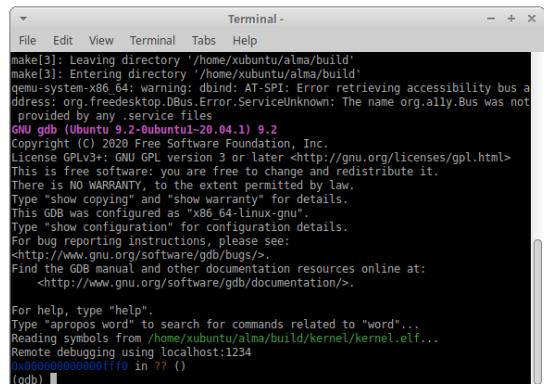


Figura 3.4: alma con gdb en la “alma build VM”

También se incluyen botones como “Tinker” para abrir VSCode con el LSP configurado y listo para trabajar, “Update” para poder actualizar alma (`git stash + git pull`), “Clean” para limpiar los archivos construidos, “Browse” para navegar por el código fuente de forma visual, “Terminal” para abrir una terminal en el directorio fuente de *alma* y “Demo” para descargar la última versión de alma y ejecutarla.

Si fuese necesario, la contraseña de administrador de la máquina es: **xubuntu**

3.3. Dependencias de Ejecución

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

- qemu
- qemu-system-x86

Ubuntu:

```
>_ apt install qemu-system-x86
    ↵ qemu-system-gui
```

Arch Linux:

```
>_ pacman -S qemu
    ↵ qemu-arch-extra
```

Ahora se puede descargar y ejecutar una build precompilada de alma. En primer lugar nos dirigimos a <https://ls.ecomaikgolf.com/alma/builds/> y elegimos el .tar.gz de la build que queramos ejecutar. Acto seguido continuamos ejecutando los siguientes comandos, cambiando CHANGEME por el texto correspondiente en nuestro caso:

```
>_ wget https://ls.ecomaikgolf.com/alma/builds/CHANGEME.tar.gz
tar xf CHANGEME.tar.gz
cd CHANGEME
qemu-system-x86_64 -machine q35 -cpu qemu64 -m 256M -bios bios.bin
    ↵ -netdev user,id=user.0 -device
    ↵ rtl18139,netdev=user.0,mac=ca:fe:c0:ff:ee:00 -object
    ↵ filter-dump,id=f1,netdev=user.0,file=log.pcap -boot d -cdrom
    ↵ alma.iso
```

Si se quisiese ejecutar en hardware real, es necesario un UBS, insertarlo en la máquina, conocer su identificador /dev/sdX mediante `fdisk -l` y ejecutar el siguiente comando:

```
>_ sudo dd bs=4M if=alma.iso of=/dev/CHANGEME conv=fdatasync
    ↵ status=progress
```

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 el commit ha recibido modificaciones adicionales (no mostradas en el repositorio) de estilo para que sea más visual.

3.4. 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 mostrados representan paquetes de **Ubuntu 20.04**, también se ha replicado la instalación en Arch Linux.

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

>—

```
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 xorriso
```

Este listado de paquetes lo componen herramientas necesarias para la compilación de alma y herramientas necesarias para la construcción de la toolchain de alma. Si se quisiese construir la toolchain de alma para poder compilar, véase la sección 3.1.

3.4.1. 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 [13]. El triplete está compuesto por tres⁴ campos:

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

⁴No siempre es obligatorio que aparezcan los tres

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.

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 [14] una zona de 128 bytes de longitud situada debajo del stack pointer (véase fig. 3.5a) 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.

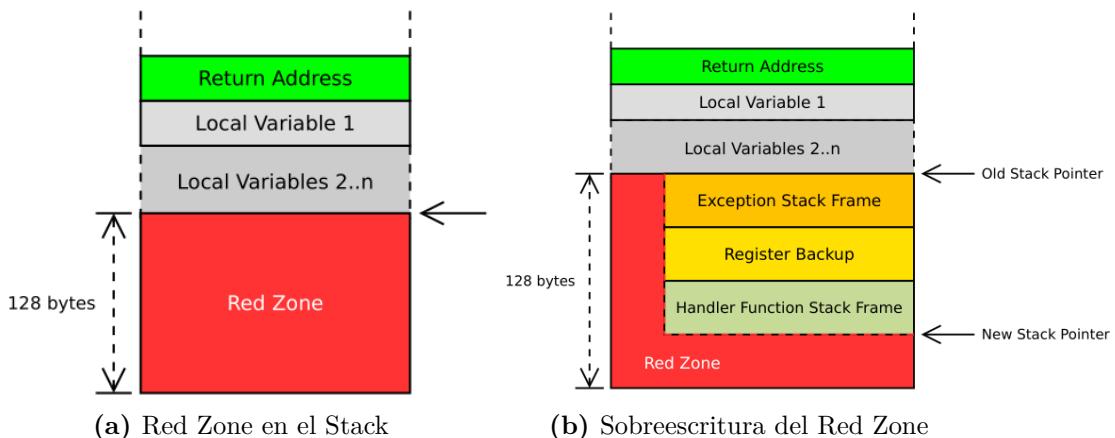


Figura 3.5: Red Zone en el Stack (<https://os.phil-opp.com>)

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 stack⁵, pero si el programa en ejecución es sorprendido por una `interrupt routine` 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 3.5b.

⁵El stack en la ABI de System V crece “hacia abajo” [15]

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 llamas a `libgcc`, no se asuma que hay un espacio de 128 bytes pasado el stack pointer.

También, debido al reciente cambio de bootloader para usar limine, es necesario tener `libgcc` con ciertas modificaciones, activadas al compilarlo con `-mcmodel=kernel`.

Debido a todos los problemas que presenta un gcc común, he compilado manualmente gcc 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.

```
④ Código 3.2: Compilación de gcc 11.x
1 gcc-build:
2   $(eval CONTENT = $(shell grep -F x86_64-*-elf -n $(CURDIR)/gcc/gcc/config.gcc | cut -f1 -d:))
3   $(eval NUMBER = $(shell grep -F x86_64-*-elf -n $(CURDIR)/gcc/gcc/config.gcc | cut -f1 -d:))
4   cd gcc; \
5   git checkout $(gcc_v); \
6   ./contrib/download_prerequisites; \
7   echo -e "MULTILIB_OPTIONS += mno-red-zone\nMULTILIB_DIRNAMES += no-red-zone" > gcc/config/i386/t-x86_64-elf; \
8   sed '$(NUMBER) a $(CONTENT)' gcc/config.gcc; \
9   mkdir build; \
10  cd build; \
11  export PATH=$(BUILD_DIR_TOOLCHAIN)/bin/:$$PATH; \
12  export PATH=$(BUILD_DIR_TOOLCHAIN)/libexec/gcc/x86_64-elf/11.2.1/:$$PATH; \
13  ./configure --target=x86_64-elf --prefix="$(BUILD_DIR_TOOLCHAIN)" --disable-nls --enable-languages=c,c++ --without-headers; \
14  make -j$(THREADS) all-gcc; \
15  make -j$(THREADS) all-target-libgcc CFLAGS_FOR_TARGET='-g -O2 -mcmodel=kernel -mno-red-zone' || true; \
16  sed -i 's/PICFLAG/DISABLED_PICFLAG/g' x86_64-elf/libgcc/Makefile; \
17  make -j$(THREADS) all-target-libgcc CFLAGS_FOR_TARGET='-g -O2 -mcmodel=kernel -mno-red-zone'; \
18  make install-gcc; \
19  make install-target-libgcc;
```

El script se mete a la carpeta del Version Control System (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.

3.4.2. 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 3.3: 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.

3.4.3. EDK II OVMF

Para poder usar Unified Extensible Firmware Interface (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 [16].

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

© Código 3.4: 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);

```

El target 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`.

3.4.4. posix-uefi

Para escribir el bootloader es necesaria la librería `posix-uefi` [1] que como ya se ha explicado, provee una API Portable Operating System Interface (POSIX) sobre la librería `gnu-efi`, lo que nos permite escribir aplicaciones para UEFI usando funciones POSIX.

© Código 3.5: Compilación de posix-uefi

```

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 la sección 4.3.

4. Bootloader

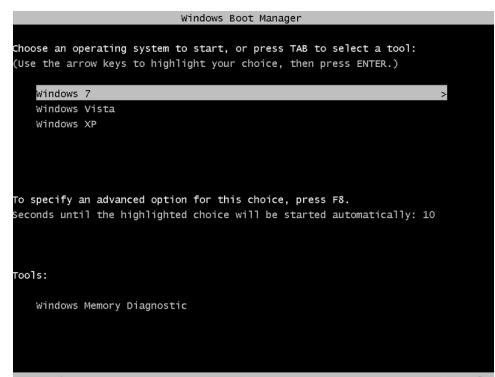
En este capítulo se presenta el desarrollo del bootloader para *alma*. Este bootloader, escrito en C para arranque UEFI se utilizaba en versiones iniciales de *alma*, actualmente está en desuso debido a mejoras hechas en el kernel. En este capítulo se introduce el concepto y teoría sobre los bootloaders en la sección 4.1, luego se comentará el trabajo que hay hecho en este área en la sección 4.2 y finalmente, en las secciones 4.3 y 4.4 se mostrará el proceso de desarrollo del bootloader y su sistema de construcción.

4.1. 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 GRUB¹ (Figura 4.1a), 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.



(a) Grub v1.98



(b) Windows Boot Manager

Figura 4.1: Ejemplos de bootloaders

¹En caso de que sea el bootloader que hayamos instalado claro, por defecto suele ser GRUB.

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 4.1b) el cual dará paso a nuestro Windows 7. Otras como Mac OSX disponen de BootX², FreeBSD de “Boot Manager”³, 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 desde el .iso⁴ 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 ROM⁵ 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 correctamente, 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 4.2) 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.

²<https://web.archive.org/web/20070309142504/http://www.cs.rpi.edu/~gerbal/BootX.pdf>

³<https://people.freebsd.org/~trhodes/doc/handbook/boot-blocks.html>

⁴Se ha utilizado como ejemplo, no tiene por qué ser un iso

⁵Actualmente se utiliza memoria flash para poder actualizarla.

⁶Segmento 0, dirección 0x7c000

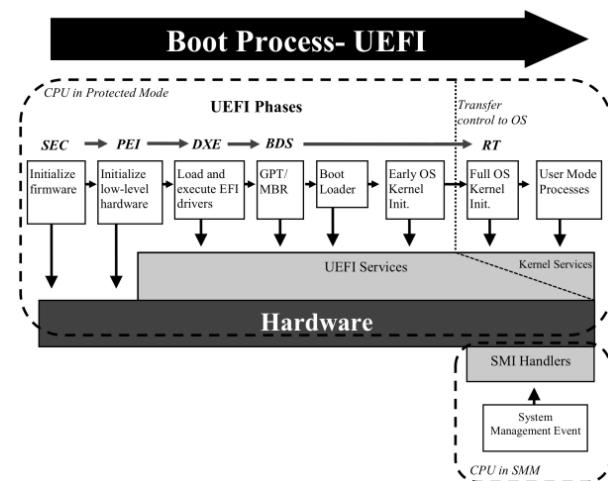


Figura 4.2: Arranque UEFI

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

Si estamos desarrollando un bootloader para nuestro sistema tendremos que tener en cuenta si queremos ejecutarlo bajo el paraguas de UEFI 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 tuviese UEFI, (normalmente) 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.

i 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, sino un kernel. 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.

⁷Si se ha trabajado 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

4.2. 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 4.1: 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 4.3a), 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 4.3: 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 4.1), 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 4.2). 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 4.1: 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 4.2: 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**

4.3. 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 4.3), 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 4.3: `Makefile` de `posix-uefi`

```

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 4.4: `CMakeLists.txt` bootloader 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 4.5: `CMakeLists.txt` bootloader 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 4.5 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 3.5a). 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 4.6: CMakeLists.txt bootloader 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.

```
④ Código 4.7: CMakeLists.txt bootloader 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 4.7 establecemos los directorios en los cuales el compilador va a buscar las cabeceras incluidas en los fuentes.

```
④ Código 4.8: CMakeLists.txt bootloader 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 grandes⁹.

```
④ Código 4.9: CMakeLists.txt bootloader 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
```

⁹Referencia 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 4.9 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 4.9) 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

```

4.4. 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
```

4.4.1. 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 4.10: 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 4.11: err_values ejemplo

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

Encontraremos este tipo de construcciones en el `main` del bootloader (sección 4.4.8).

4.4.2. 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 4.12: 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.

4.4.3. 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 4.13: io/file.h

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

4.4.3.1. Obtener el Tamaño de un Fichero

Q Código 4.14: 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 bytes¹⁰.

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

¹⁰Otro 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()`:

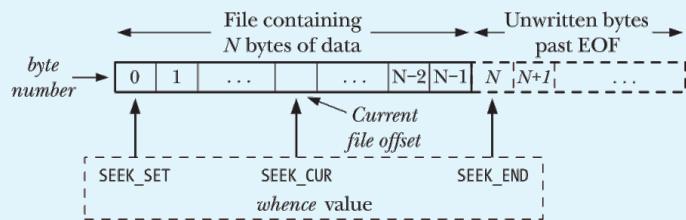


Figura 4.4: Funcionamiento SEEK_SET, SEEK_CUR y SEEK_END [5]

4.4.3.2. Cargar un Fichero en Memoria

```
Q Código 4.15: 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 memoria¹¹.

4.4.4. 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.

4.4.4.1. Graphics Output Protocol

El Graphics Output Protocol [17, 18] es el estándar utilizado por UEFI que reemplazó el estándar VESA [19] (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 4.16: gop.h

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

Código 4.17: 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) {
```

¹¹No 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.

4.4.4.2. Framebuffer

El framebuffer [20] 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 estructura¹² con toda la información relevante del framebuffer:

Código 4.18: `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;
```

¹²Como 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 visibles¹³) 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 4.4.4.1

```
Código 4.19: 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.

i 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 [21]:

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 4.5: Formato Píxel GOP

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

`*(uint32_t *)base = 0xFFFFFFFF`

¹³Los píxeles no visibles son píxeles que no se van a mostrar en pantalla y que se utilizan como padding en la memoria del framebuffer

4.4.4.3. 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 [22]). 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:

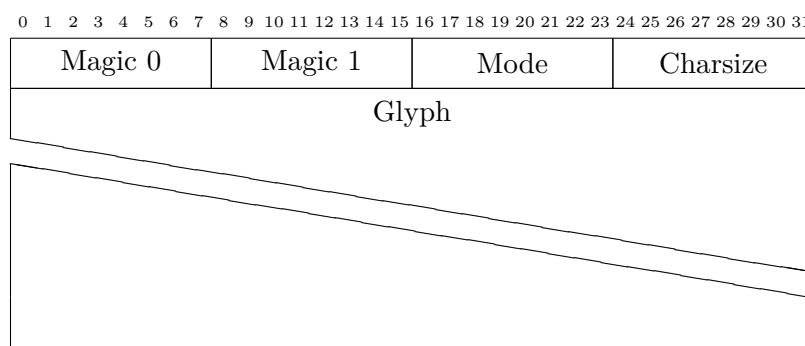


Figura 4.6: Fuente PSF1 [6]

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 0....."
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.....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 4.20: 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 4.21: 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 4.22: 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 cast¹⁴.

© Código 4.23: 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 4.24: 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.

¹⁴Se ha preferido encapsularlo en una función propia por abstracción.

El buffer de glifos estará justo después de la cabecera PSF1 (Figura 4.6), 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.

4.4.5. 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 4.25: `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 [23]) y es el siguiente:

© Código 4.26: `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 4.27: `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 mapa¹⁵. Luego iteramos¹⁶ una serie de veces para obtener el mapa de memoria mediante el mismo servicio **GetMemoryMap** (previamente reservando¹⁷ 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).

¹⁵La funcionalidad varía dependiendo de los parámetros que le pases

¹⁶No es necesario como tal pero es una recomendación que encontré en una guía de desarrollo.

¹⁷Hacer **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 4.28: 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 modifique¹⁸ el código y añada una llamada a esta donde necesite.

¹⁸Esto 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.

4.4.6. 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`¹⁹.

 Código 4.29: `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 4.30: `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;
```

¹⁹Esto 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 [24] y el XSDT [25] (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 [26]. 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 4.31: 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’²⁰ para ver si de verdad se trata del RSDP²¹.

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 4.32: 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;
```

²⁰Con el espacio al final y sin terminación null.

²¹También 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 4.33: 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 }
```

4.4.7. ELF

ELF (Executable and Linkable Format)²², 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 [27] y más tarde en el “Tool Interface Standard” [28].

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*²³ 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 4.34: 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 */
```

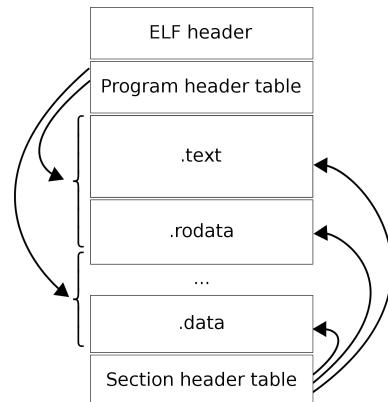


Figura 4.7: Formato ELF

²²Anteriormente Extensible Linking Format

²³El 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 4.35: 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		
	0x00000000000000e8	0x00000000000000e8	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 4.36: 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

4.4.7.1. 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 4.37: 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.

4.4.7.2. Obtener y Verificar la Cabecera ELF

Para obtener la cabecera del fichero ELF, como se puede observar en la Figura 4.7, 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 4.38: 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`²⁴ contiene unas constantes predefinidas en (el magic number). En el caso de ELF el magic number es `0x7F`,

²⁴Los 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 4.39: 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 4.8: Formato Código de Error

4.4.7.3. 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 [29]: 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 4.40: 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 4.5.1.1.

4.4.7.4. 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 4.41: 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 4.42: 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` [30]) 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.impfprintf
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 4.43: 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 ELF²⁵ 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 bits²⁶, 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 PE²⁷. Finalmente llamamos a dicha función.

4.4.8. 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 4.44: 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     MapInfo *map = load_memmap();
```

²⁵Sección que contiene strings del ELF, como los nombres de las secciones.

²⁶Puesto que estamos en sistemas de 64 bits.

²⁷Esto 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*²⁸) 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

²⁸Se 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 principal²⁹.

```
(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 4.9: Traza de ejecución del bootloader

Podemos ver en la Figura 4.9 una traza de la función principal que se ha analizado en esta sección.

²⁹Apagar la máquina no hace que las funciones principales devuelvan. Esto no debe pasar nunca.

4.5. 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* [31] 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 4.5.1.

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>

4.5.1. 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.

4.5.1.1. 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 4.45: 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*.

4.5.1.2. 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 4.46: 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 memoria³⁰ 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.

³⁰Cosa 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).

5. Kernel

Este capítulo tiene como objetivo presentar el proyecto principal del trabajo: *alma*, el kernel desarrollado. En este capítulo se introducirá el concepto del kernel (sección 5.1) junto al estado del arte y proyectos similares (sección 5.2). Posteriormente, se entrará en el desarrollo de *alma*, en sistema de construcción (sección 5.3) y desarrollo del kernel (sección 5.4). Finalmente se presentarán las conclusiones obtenidas al desarrollar este proyecto (sección 5.5).

5.1. Introducción

Los núcleos (kernels) son la piedra angular de la informática moderna y los sistemas operativos. Para entender el concepto de kernel es necesario primero definir el concepto de sistema operativo, los sistemas operativos son la capa de software encargada de gestionar los recursos de un ordenador para sus usuarios y aplicaciones. Una definición más técnica y ampliamente utilizada [32] en el campo es que el sistema operativo es todo código que se ejecuta con permisos de kernel en la máquina. Estos sistemas operativos pueden ser transparentes para nosotros, por ejemplo, en sistemas empotrados con los conocidos “Real Time Operating Systems”.

Habiendo definido el concepto de sistema operativo, podemos darle sentido al término de “kernel” que hemos presentado. En la literatura consultada se engloba el kernel dentro del concepto de sistema operativo, pero considero que es importante darle una definición propia y autosuficiente. El kernel es la pieza de software, comúnmente integrada en lo que conocemos como sistema operativo, encargada de trabajar directamente con el hardware de la máquina.

A nivel funcional, los kernels proveen abstracciones del hardware a los sistemas operativos, sin importar el hardware y la implementación. Un ejemplo de esto puede ser la gestión de memoria en el kernel de Linux, Linux (que es un kernel, no un sistema operativo) nos gestiona la memoria RAM de la máquina y proporciona al sistema operativo que tengamos (comúnmente GNU) funcionalidades relacionadas con la memoria. Como podemos ver, el sistema operativo GNU no tiene la necesidad de entrar en detalles técnicos del hardware, simplemente le importa su comunicación con el kernel, que este sea el que tenga que lidiar con el hardware. Podemos ver este comportamiento en la Figura 5.1 que muestra la arquitectura en el kernel de Linux.

Debe quedar claro que el kernel es la pieza principal de un sistema operativo, es el que generalmente tiene control absoluto del sistema y el que facilita las interacciones entre componentes hardware y software (Figura 5.1). El código del kernel está constantemente cargado en memoria desde que es ejecutado por primera vez y tiene unas necesidades de protección críticas.

Cuando el bootloader se ha cargado por parte del sistema de arranque, lo que hace, como ya se ha comentado en la sección 4.1, es preparar un entorno pre establecido para el kernel. El bootloader se encarga de obtener ciertas estructuras de datos necesarias para el kernel y “preparar el entorno” (como por ejemplo activar la paginación), luego cuando ha terminado de cargar, éste pasa el control de la máquina al kernel. Aquí el kernel tiene que trabajar para preparar su entorno de ejecución, inicializar el hardware (drivers), etc.

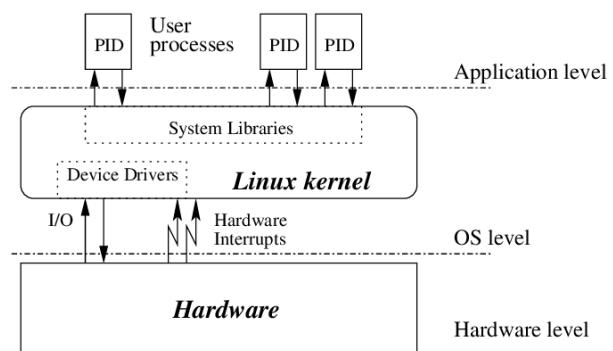


Figura 5.1: Arquitectura del Kernel Linux [7]

Algo que queda por especificar es cómo el sistema operativo es capaz de comunicarse con el kernel. Para no entrar en detalles técnicos en la introducción se dirá solo que es mediante interrupciones, un sistema que tiene la CPU que nos permite llamar a funciones de forma asíncrona. Esto se puede ver referenciado como “Hardware Interrupts” en la Figura 5.1.

Finalmente comentar que en este trabajo no se pretende desarrollar un sistema operativo completo y usable, capaz de solicitar servicios al kernel y presentar una capa usable para el usuario. Como no va a haber nadie que utilice las funcionalidades que pueda presentar el kernel, lo que se ha hecho es eliminar esa capa de comunicación hacia arriba, las funcionalidades simples que necesitemos se implementarán en el propio kernel. Esto difiere de lo que se ha explicado anteriormente de que el kernel tiene como finalidad gestionar recursos hardware y proveer servicios al sistema operativo, en este caso lo que haremos es proveernos servicios a nosotros mismos, el kernel. Lo único que se ha eliminado de lo explicado anteriormente es la capa de comunicación, es lo único que se debe tener en cuenta.

5.2. Estado del Arte

En esta sección se verán técnicas y métodos de diseño de los núcleos ampliamente abiertas a debate en la comunidad científica. También se verán los núcleos ya establecidos en la industria y qué técnicas y métodos de diseño han seguido, también se expondrán kernels desarrollados por programadores aficionados con un cierto grado de madurez.

5.2.1. Técnicas y Métodos de Diseño

La arquitectura que debe tener un kernel ha sido tema de debate en la comunidad científica y de desarrolladores a lo largo de los años. Existen varias arquitecturas importantes a la hora de diseñar un kernel y muchas ventajas y desventajas a la hora de elegir una u otra . En esta sección se presentarán los principales diseños de los kernels con sus ventajas y desventajas.

5.2.1.1. Kernels Monolíticos

Los kernels monolíticos son los núcleos que tienen todo el código y toda la responsabilidad de las funcionalidades agrupadas en un único ejecutable final. Todo está implementando “junto” en el mismo código y presentado como un ejecutable final. Este tipo de diseños suelen ser puestos en los kernels importantes, por ejemplo, Linux.

La principal ventaja de estos núcleos es su rendimiento inmejorable, otros diseños presentan mejor abstracción a costa de tener que implementar mecanismos IPC costosos para comunicar sus módulos. En este caso, como todo está integrado en un mismo bloque, todas esas comunicaciones son llamadas entre el mismo código, algo muy barato en comparación (a costa de menor modularidad).

El problema en el que incurren es que, como es todo un gran bloque de código (monolítico), puede ser difícil de mantener. Por otro lado, al ser un código más “simple” (dentro de lo complejos que son) pueden sufrir en menos errores (al no tener código para servicios como en los microkernels).

5.2.1.2. Microkernels

Los microkernels son núcleos donde el kernel como tal tiene una responsabilidad y funcionalidades mínimas suficientes como para ejecutar “servicios”. Los servicios son módulos externos que proveerán las funcionalidades comunes del kernel, son completamente modulares por lo que desacoplan el sistema (kernel) y lo dotan de muy buena modularidad.

La principal desventaja es que al ser módulos distintos, las comunicaciones entre los distintos módulos deben ser realizada mediante mecanismos de IPC, algo que reduce mucho el rendimiento de los núcleos debido al *overhead* que conllevan.

Uno de los principales defensores de los microkernels es Andrew S. Tanenbaum, que consideró Linux como obsoleto por ser un núcleo monolítico. Esto conllevó al duro debate conocido como “Tanenbaum-Torvalds debate”¹. Andrew, desde un lado más teórico, es defensor de los Microkernels mientras que Linus Torvalds, desde un lado más práctico, considera que los monolíticos son mejores.

¹https://en.wikipedia.org/wiki/Tanenbaum%20vs%20Torvalds_debate

5.2.1.3. Kernels Híbridos

Los kernels híbridos son aquellos que, para tomar ventajas y desventajas de cada uno de los diseños, mezclan componentes de kernels monolíticos y microkernels.

Lo que hacen es implementar partes críticas que requieran de gran rendimiento como si fuese un kernel monolítico, por ejemplo el stack de red. Otros componentes que no tengan este requerimiento pueden beneficiarse de las ventajas de los microkernels.

Este tipo de kernels se encuentran en sistemas operativos comerciales, como los de Microsoft Windows (NT 3.1, ..., Windows 10) y el XNU de apple en MacOS.

Cabe mencionar que *alma* es un kernel con diseño **monolítico**.

5.2.2. Implementaciones de las syscall

La forma de implementar las llamadas también es un aspecto que se puede decidir y generará diferencias en el kernel desarrollado (no tantas como su arquitectura, presentada en el apartado anterior). Esta es la forma en la que capas superiores solicitarán servicios con el kernel y se establecerá una comunicación.

Mediante Interrupciones: Lo que se hace en este caso es implementar funciones que lo que hagan sea llamar a las “interrupt routines” correspondientes mediante la instrucción en ensamblador para levantar una interrupción software `int`.

Mediante Call Gates: En este caso se dispone de una dirección especial conocida por el procesador a la que saltamos, el procesador la detecta, la reconoce, y nos redirige a la localización del código que queremos (sin provocar una violación del segmento).

Mediante una Instrucción: Es el más simple, el procesador incorpora una instrucción especial que lo realiza. En este caso tenemos que tener soporte por parte de la arquitectura, en el caso de x86 no la tiene, aunque modelos recientes de CPUS x86 lo incorporan.

Mediante una Cola: Simplemente se añade una “solicitud” a una cola y el kernel periódicamente escanea las peticiones y las va ejecutando.

5.2.1.4. Exokernels

Los exokernels son aquellos que limitan su funcionalidad a la protección y multiplexación del hardware. No proveen de una abstracción del hardware para las aplicaciones por encima de su capa. De esta forma se cree que los desarrolladores pueden gestionar los recursos de mejor manera para cada caso concreto, otorgando mucha libertad para obtener implementaciones más específicas.

Este tipo de diseño cabe mencionar que es aún experimental y está en fase de desarrollo. Pese a que es muy interesante la idea de gestionar cada uno sus recursos de la mejor manera, puede fácilmente conllevar a errores de implementación, haciendo al sistema inusable.

5.2.3. Proyectos Existentes

Una vez vistas las grandes arquitecturas que dominan el diseño de los núcleos, veremos los kernels que existen en la actualidad. Este listado² lo dividiremos en dos partes, kernels establecidos en la industria (Tabla 5.1) y kernels más académicos (Tabla 5.2).

5.2.3.1. Industria

Kernel	Diseño/Arquitectura	Lenguaje de Desarrollo
Linux	Monolítico	C
XNU	Híbrido	C/C++
Windows NT	Híbrido	C/C++
Windows 9x Series	Monolítico	¿?
FreeBSD	Monolítico	C/C++
Dragonfly BSD	Híbrido	C
Plan9	Híbrido	C
ReactOS	Híbrido	C/C++
Exec (Amiga)	Microkernel	¿?
Solaris	Monolítico	C/C++
Zircon	Microkernel	C++

Tabla 5.1: Kernels mayormente utilizados en la industria

5.2.3.2. Académicos

Kernel	Diseño/Arquitectura	Lenguaje de Desarrollo
SerenityOS	Monolítico	C++
TempleOS	Monolítico	HolyC
ToaruOS	Híbrido	C
OS67	Microkernel	C
Resea	Microkernel	C
CyanOS	Monolítico	C++
ChaiOS	Híbrido	C
Biscuit	Monolítico	Go
Managarm	Microkernel	C++
Toddler	Microkernel	C
Kerla	Monolítico	Rust

Tabla 5.2: Kernels Académicos con cierto grado de madurez [9] [10]

²Siempre que aparezca el nombre de un sistema operativo, se hace referencia a su kernel.

5.3. Sistema de Construcción

Construir kernel encargado de controlar los recursos hardware de una máquina no es tarea simple como construir un archivo C/C++ común. Los núcleos requieren, como ya se ha comentado en la sección 3.4.1, compiladores construidos con modificaciones. También se requiere controlar de forma detallada el enlazado de los objetos en el ELF final, es por todo esto que la construcción de *alma* requiere de una sección especial para explicarla.

Esta sección se dividirá en dos subsecciones: una donde se explicará el sistema de construcción del kernel y otra para el sistema de construcción global del proyecto, donde está la generación del *iso* final, su ejecución, etc.

Como se ha visto en la sección 4.3, la herramienta elegida para los sistemas de construcción es *cmake* [33].

5.3.1. Kernel

La construcción de alma empieza en `kernel/CMakeLists.txt`. En primer lugar se definen las cabeceras del proyecto y configuraciones generales.

```
Código 5.1: CMakeLists.txt kernel I
1 cmake_minimum_required(VERSION 3.16)
2 project(alma-kernel CXX ASM_NASM)
3 find_program(NASM nasm REQUIRED)
4
5 find_program(CCACHE_FOUND ccache)
6 if(CCACHE_FOUND)
7     set_property(GLOBAL PROPERTY RULE_LAUNCH_COMPILE ccache)
8     set_property(GLOBAL PROPERTY RULE_LAUNCH_LINK ccache)
9 endif(CCACHE_FOUND)
```

Como podemos ver, lo que hacemos es indicar que la versión mínima de *cmake* requerida para construir el proyecto es la 3.16 (la que viene en Ubuntu 20.04, instalado en los laboratorios). Acto seguido definimos las propiedades del proyecto: el nombre y los lenguajes que va a utilizar (C++ y ASM-NASM), seguido tenemos una sentencia para comprobar que el usuario tiene instalado en la máquina el programa `nasm` (un compilador para ensamblado). Finalmente activaremos `ccache` para acelerar las construcciones repetidas del proyecto si y solo si el usuario tiene instalado `ccache`.

```
Código 5.2: CMakeLists.txt kernel II
10 set(CMAKE_C_COMPILER "${TOOLCHAINBIN}/x86_64-elf-gcc")
11 set(CMAKE_CXX_COMPILER "${TOOLCHAINBIN}/x86_64-elf-g++")
12 set(CMAKE_LINKER "${TOOLCHAINBIN}/x86_64-elf-ld")
```

Aquí lo que hacemos es seleccionar los compiladores y enlazadores. Como se ha mencionado en el capítulo 3, necesitaremos unos compiladores especiales ya construidos en dicho capítulo, aquí lo que haremos es seleccionarlos para que *cmake* haga uso de ellos.

© Código 5.3: CMakeLists.txt kernel III

```

13 set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -ffreestanding")
14 set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -fno-stack-protector")
15 set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -fno-stack-check")
16 set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -fno-rtti")
17 set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -fno-exceptions")
18 set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -nostdlib")
19 set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -mno-red-zone")
20 set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -g")
21 set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -gdwarf-4")
22 set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -mcmode=kernel")

```

En este código establecemos los parámetros del compilador a la hora de construir el kernel. Establecemos una serie de parámetros como desactivar los stack canary³ y sus chequeos, también indicamos que desactive el “runtime type information”, las excepciones, le indicamos que no disponemos de una librería estándar, que no compile con red zones (explicado en la sección 3.4.1), que utilice DWARF 4 en vez de 5⁴, y que use el “model” del kernel⁵.

© Código 5.4: CMakeLists.txt kernel IV

```

23 set(CMAKE_CXX_STANDARD 20)
24 set(CMAKE_CXX_STANDARD_REQUIRED True)
25 set(CMAKE_CXX_EXTENSIONS OFF)

```

Aquí establecemos la versión mínima del estándar de C++ requerido, en este caso C++20 y sin extensiones.

© Código 5.5: CMakeLists.txt kernel V

```

26 set(CMAKE_ASM_NASM_OBJECT_FORMAT elf64)

```

Indicamos el formato en el que queremos que NASM nos genere la salida: ELF de 64 bits.

© Código 5.6: CMakeLists.txt kernel VI

```

27 set(LINKER_SCRIPT "${CMAKE_CURRENT_SOURCE_DIR}/kernel1.ld")
28 SET(CMAKE_EXE_LINKER_FLAGS "${CMAKE_EXE_LINKER_FLAGS} -nostdlib")
29 SET(CMAKE_EXE_LINKER_FLAGS "${CMAKE_EXE_LINKER_FLAGS} -z max-page-size=0x1000")
30 SET(CMAKE_EXE_LINKER_FLAGS "${CMAKE_EXE_LINKER_FLAGS} -static") # static
31 set(CMAKE_EXE_LINKER_FLAGS "${CMAKE_EXE_LINKER_FLAGS} -T ${LINKER_SCRIPT}")

```

Con este bloque de comandos de cmake establecemos las opciones del enlazador. Lo que hacemos principalmente es pasarle un script del enlazador para que enlace siguiendo las normas ahí descritas. El script es el siguiente:

© Código 5.7: kernel1.ld [31]

```

1 ENTRY(_start)
2 OUTPUT_FORMAT(elf64-x86-64)
3 OUTPUT_ARCH(i386:x86-64)
4
5 PHDRS

```

³Son datos almacenados en los bordes de los buffer para comprobar si han habido overflows.

⁴La herramienta **bloaty** no funciona con DWARF 5

⁵Más información en <https://gcc.gnu.org/onlinedocs/gcc/x86-Options.html>

```

6 {
7     null PT_NULL FLAGS(0) ; /* Null segment */
8     text PT_LOAD FLAGS((1 << 0) | (1 << 2)) ; /* Execute + Read */
9     rodata PT_LOAD FLAGS((1 << 2)) ; /* Read only */
10    data PT_LOAD FLAGS((1 << 1) | (1 << 2)) ; /* Write + Read */
11 }
12
13 SECTIONS
14 {
15     . = 0xffffffff80000000;
16     _start_addr = .;
17     .text : {
18         *(.text .text.*)
19     } :text
20     /* Move to the next memory page for .rodata */
21     . += CONSTANT(MAXPAGESIZE);
22     .stivale2hdr : {
23         KEEP(*(.stivale2hdr))
24     } :rodata
25     .rodata : {
26         *(.rodata .rodata.*)
27     } :rodata
28     /* Move to the next memory page for .data */
29     . += CONSTANT(MAXPAGESIZE);
30     .data : {
31         *(.data .data.*)
32     } :data
33     .bss : {
34         *(COMMON)
35         *(.bss .bss.*)
36     } :data
37     _end_addr = .;
38 }

```

En este script del enlazador definimos el contenido de las secciones del ELF resultante y otros parámetros. Por ejemplo en las primeras líneas establecemos la función que se va a ejecutar al cargar el ELF, en este caso `_start`, también elegimos el formato del ELF resultante, x86-64. En los PHDRS (explicados en la sección 4.4.7) definimos varios segmentos de memoria a cargar, tendremos uno nulo (por obligación), uno de código con permisos de lectura y ejecución, otro de datos constantes con permisos solo de lectura y finalmente uno de datos, con permisos solo para leer y escribir. Establecer los permisos da seguridad a *alma*.

Finalmente en `SECTIONS` definimos las secciones que tendrá el ELF, en primer lugar creamos una marca `_start_addr` (y luego `_end_addr`) para tener en el kernel el tamaño del fichero. La primera sección será la de código y, posteriormente, la cabecera de stivale, acto seguido aparecerán los datos constantes y finalmente los datos comunes y datos inicializados a cero.

 Código 5.8: CMakeLists.txt kernel VII

```

31 set(LIB_HEADER_DIR
32   lib
33 )
34 set(STIVALE_HEADER_DIR
35   ${TOOLCHAINDIR}/stivale
36 )
37 set(KERNEL_HEADER_DIR
38   ${CMAKE_CURRENT_SOURCE_DIR}
39 )
40 set(INCLUDE_DIRECTORIES

```

```

41 ${LIB_HEADER_DIR}
42 ${STIVALE_HEADER_DIR}
43 ${KERNEL_HEADER_DIR}
44 )
45
46 include_directories(${INCLUDE_DIRECTORIES})

```

En este apartado del `CMakeLists.txt` lo que hacemos es incluir los directorios con los `.h`, similar a lo que haríamos con la opción del compilador `-I`. Se pueden ver los directorios que se utilizan como raíz para los includes en los sets.

© Código 5.9: `CMakeLists.txt kernel VII`

```

47 set(LIB_SOURCES
48   lib/stdlib/itoa.cpp
49   lib/stdlib/strol.cpp
50   lib/string/memset.cpp
51   lib/string/strcmp.cpp
52   lib/string/strlen.cpp
53   lib/math/pow.cpp
54   lib/math/sqrt.cpp
55   lib/math/log.cpp
56   lib/ctype/toupper.cpp
57   lib/bitset.cpp
58 )
59 set(INTERUPT_SOURCES
60   interrupts/IDT.cpp
61   interrupts/interrupts.cpp
62 )
63 set(KERNEL_SOURCES
64   bootstrap/startup.cpp
65   paging/PFA.cpp
66   paging/BPFA.cpp
67   paging/PTM.cpp
68   screen/simple_renderer_i.cpp
69   screen/fast_renderer_i.cpp
70   uefi/memory.cpp
71   segmentation/gdt.asm
72   io/bus.cpp
73   io/keyboard.cpp
74   acpi/acpi.cpp
75   pci/pci.cpp
76   heap/simple_allocator.cpp
77   heap/trivial_allocator.cpp
78   shell/command.cpp
79   shell/interpreter.cpp
80   net/rtl18139.cpp
81   ${INTERUPT_SOURCES}
82   kernel.cpp
83 )
84
85 set(SOURCES
86   ${KERNEL_SOURCES}
87   ${LIB_SOURCES}
88 )

```

En este código lo único que se hace es listar los fuentes a ser compilados para formar el kernel. Lo único que se puede comentar es el motivo de listarlos uno a uno y no usar genera-

dores de cmake, al listarlos no tiene que comprobar si hay nuevos en posteriores ejecuciones⁶, lo que va un poco más rápido a mi parecer. También se ha tomado esta decisión por claridad, en el proyecto SerenityOS también hacen lo mismo y son capaces de gestionar una gran base de código.

```
④ Código 5.10: CMakeLists.txt kernel VIII
89 set_source_files_properties(${INTERRUPT_SOURCES} PROPERTIES COMPILE_FLAGS "-mno-red-zone -mgeneral
  ↪ -regs-only -ffreestanding")
```

Aquí lo único que hacemos es indicarle a cmake que el código fuente de las interrupciones lo compile solo con esas tres opciones de compilación.

```
④ Código 5.11: CMakeLists.txt kernel IX
90 add_executable(kernel ${SOURCES})
91
92 set_target_properties(kernel PROPERTIES
93   SUFFIX .elf
94 )
95
96 set_target_properties(kernel PROPERTIES
97   LINK_DEPENDS ${LINKER_SCRIPT}
98 )
```

Este bloque de código se puede considerar el más importante de todo el sistema de construcción, es el encargado de generar el ELF final del kernel. Establecemos un ELF ejecutable como salida llamado `kernel` con sufijo `.elf`. cmake se encargará, junto al conjunto de reglas anteriores, de generar este archivo.

Hasta aquí tendríamos un kernel funcional, solo con un inconveniente, no disponemos de constructores globales de C++. Tenemos que configurar cmake con ciertos “trucos” para disponer de ellos sin necesidad de hacer lo mostrado en la sección 4.4.7.4, puesto que ahora, al usar stivale2 y limine, no disponemos de acceso al bootloader y limine no llama a los constructores globales como se hacía en el bootloader desarrollado.

```
④ Código 5.12: CMakeLists.txt kernel X
99 add_library(crts OBJECT
100   crt1.asm
101   crt1n.asm
102 )
103
104 add_dependencies(kernel crts)
105
106 execute_process( COMMAND ${CMAKE_CXX_COMPILER} -print-file-name=crtbegin.o OUTPUT_VARIABLE
  ↪ CRTBEGIN_O OUTPUT_STRIP_TRAILING_WHITESPACE )
107 execute_process( COMMAND ${CMAKE_CXX_COMPILER} -print-file-name=crtend.o OUTPUT_VARIABLE CRTEND_O
  ↪ OUTPUT_STRIP_TRAILING_WHITESPACE )
108
109 set(CMAKE_CXX_LINK_EXECUTABLE "${CMAKE_LINKER} <LINK_FLAGS> ${CMAKE_BINARY_DIR}/kernel/CMakeFiles/
  ↪ crts.dir/crt1.asm.o ${CRTBEGIN_O} <OBJECTS> -o <TARGET> <LINK_LIBRARIES> ${CRTEND_O} ${
  ↪ CMAKE_BINARY_DIR}/kernel/CMakeFiles/crts.dir/crt1n.asm.o" )
```

⁶Aparecen mensajes que indican eso.

Esta técnica consiste en enlazar el kernel con 4 objetos nuevos: `crti.o`, `crtn.o`, `crtbegin.o` y `crtend.o`. Han de enlazarse en un orden específico, lo que conseguiremos es que gcc sepa que funciones hay que llamar antes de entrar en la función principal y genere código para llamar a estas funciones.

gcc nos proporciona dos de estos archivos: `crtbegin.o` y `crtend.o`, podemos ver como gcc nos los da ejecutando `gcc --print-file-name=crtbegin.o`. Tanto `crti.asm` como `crtn.asm` tenemos que proveerlos nosotros y compilarlos, no es muy difícil puesto que son archivos “esqueletos” que gcc rellenará.

⌚ Código 5.13: crtbegin.asm

```
1 BITS 64
2
3 section .init
4 global _init:function
5 _init:
6     push rbp
7     mov rbp, rsp
8
9 section .fini
10 global _fini:function
11 _fini:
12     push rbp
13     mov rbp, rsp
```

⌚ Código 5.14: crtend.asm

```
1 BITS 64
2
3 section .init
4     pop rbp
5     ret
6
7 section .fini
8     pop rbp
9     ret
```

En cmake pondremos que el kernel dependa de la construcción de estos dos archivos, también haremos que cmake ejecute `gcc --print-file-name=...` para obtener los archivos que nos da gcc. Finalmente lo que hacemos es modificar manualmente en cmake el comando de enlace, introduciendo el orden manualmente y ya podríamos compilar el kernel:

```
>_ cmake -B build; make -C build kernel
```

⚠ Comando de enlace en cmake

La forma de la que se ha modificado el orden de enlace en cmake no es la mejor. Como podemos ver se escriben los paths con direcciones predefinidas a los archivos objeto. Esta no es la mejor forma pero para el ámbito del proyecto funcionará a la perfección.

```
>_ cmake -B build; cd build; make kernel

-- The C compiler identification is GNU 11.2.0
-- The CXX compiler identification is GNU 11.2.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- The ASM_NASM compiler identification is NASM
-- Found assembler: /usr/bin/nasm
-- Configuring done
-- Generating done
-- Build files have been written to: /home/ecomaikgolf/alma/build
Consolidate compiler generated dependencies of target crt
[ 3%] Building ASM_NASM object kernel/CMakeFiles/crts.dir/crti.asm.o
[ 6%] Building ASM_NASM object kernel/CMakeFiles/crts.dir/crtn.asm.o
[ 6%] Built target crt
Consolidate compiler generated dependencies of target kernel
[ 9%] Building CXX object kernel/CMakeFiles/kernel.dir/bootstrap/startup.cpp.o
[ 12%] Building CXX object kernel/CMakeFiles/kernel.dir/paging/PFA.cpp.o
[ 15%] Building CXX object kernel/CMakeFiles/kernel.dir/paging/BPFA.cpp.o
[ 18%] Building CXX object kernel/CMakeFiles/kernel.dir/paging/PTM.cpp.o
[ 21%] Building CXX object kernel/CMakeFiles/kernel.dir/screen/simple_renderer_i
[ 24%] Building CXX object kernel/CMakeFiles/kernel.dir/screen/fast_renderer_i.c
[ 27%] Building CXX object kernel/CMakeFiles/kernel.dir/uefi/memory.cpp.o
[ 30%] Building ASM_NASM object kernel/CMakeFiles/kernel.dir/segmentation/gdt.as
[ 33%] Building CXX object kernel/CMakeFiles/kernel.dir/io/bus.cpp.o
(...)
[ 87%] Building CXX object kernel/CMakeFiles/kernel.dir/lib/math/sqrt.cpp.o
[ 90%] Building CXX object kernel/CMakeFiles/kernel.dir/lib/math/log.cpp.o
[ 93%] Building CXX object kernel/CMakeFiles/kernel.dir/lib/ctype/toupper.cpp.o
[ 96%] Building CXX object kernel/CMakeFiles/kernel.dir/lib/bitset.cpp.o
[100%] Linking CXX executable kernel.elf
[100%] Built target kernel
```

```
>_ make kernel

Consolidate compiler generated dependencies of target crt
[ 6%] Built target crt
Consolidate compiler generated dependencies of target kernel
[100%] Built target kernel
```

Como vemos, si los archivos ya están compilados no se recompila ni se reenlaza nada.

5.3.2. Proyecto

Una vez construido el ELF del kernel lo que resta es construir el *iso* final. También se verá como incluir “targets” personalizados en el sistema de construcción para simular alma y hacer debug.

```
④ Código 5.15: CMakeLists.txt proyecto I
1 cmake_minimum_required(VERSION 3.16)
2 project(alma)
3
4 find_program(MFORMAT mformat REQUIRED)
5 find_program(MMD mmd REQUIRED)
6 find_program(MCOPY mcopy REQUIRED)
7 find_program(QEMU qemu-system-x86_64 REQUIRED)
```

En primer lugar, al igual que en el kernel, establecemos la versión mínima de cmake con la que se puede construir: 3.16. Acto seguido establecemos el nombre del proyecto⁷. Finalmente pedimos a cmake que compruebe si el usuario tiene una serie de programas necesarios⁸ instalados.

```
④ Código 5.16: CMakeLists.txt proyecto II
8 set(TOOLCHAINDIR "${CMAKE_CURRENT_SOURCE_DIR}/toolchain")
9 set(TOOLCHAINBIN "${TOOLCHAINDIR}/build/toolchain/bin")
```

Este bloque de código lo único que hace es establecer la localización de la toolchain en el proyecto. Tanto el **CMakeLists.txt** del kernel como del bootloader heredarán estos parámetros, por lo que en este archivo se pueden establecer configuraciones globales a todo el proyecto.

```
④ Código 5.17: CMakeLists.txt proyecto III
10 set(QEMU_BIN qemu-system-x86_64)
11 set(QEMU_CPU -cpu qemu64)
12 set(QEMU_MACH -machine q35)
13 set(QEMU_RAM -m 256M)
14 set(QEMU_BIOS -bios ${TOOLCHAINDIR}/build/uefi/bios.bin)
15 set(QEMU_NET -netdev user,id=user.0 -device rtl8139,netdev=user.0,mac=ca:fe:c0:ff:ee:00 -object ↵
     ↵ filter-dump,id=f1,netdev=user.0,file=log.pcap)
16 set(QEMU_BOOT -boot d -cdrom ${CMAKE_BINARY_DIR}/alma.iso)
17 set(QEMU_DBG -s -S)
```

Este apartado del código representa las opciones usadas en qemu al ejecutar (menos QEMU_DBG). En primer lugar tenemos el nombre del ejecutable **qemu-system-x86_64**, acto seguido especificamos la CPU a usar, en este caso la de qemu con 64 bits. Luego especificamos el tipo de máquina⁹ a q35 “Standard PC (Q35 + ICH9, 2009)”. Luego especificamos la memoria RAM disponible, 256M. Seguidamente especificamos el firmware de UEFI a usar. A continuación se establece la configuración de la tarjeta de red a emular, una RTL8139 con mac

⁷Como se puede apreciar, en este caso no establecemos lenguaje alguno porque el **CMakeLists.txt** va a actuar de “director de orquesta” en esta construcción

⁸**mformat**, **mmd**, **mcopy** no son necesarios si no se va a construir versiones de alma con bootloader propio

⁹Se puede consultar con **qemu-system-x86_64 -machine help**

ca:fe:c0:ff:ee:0 y guardando los paquetes en un archivo `log.pcap`. Finalmente le decimos a qemu que arranque desde el iso de alma construido. La opción `-s -S` es para arrancar qemu con el servidor `gdb` para hacer debug.

© Código 5.18: CMakeLists.txt proyecto III

```
18 set(CMAKE_EXPORT_COMPILE_COMMANDS ON)
```

Esta sección del `CMakeLists.txt` es opcional y no tiene que ver con la construcción del proyecto. Lo que se hace este comando es indicar a `cmake` que al construir el proyecto genere un archivo `compile_commands.json`. Este archivo sirve para indicar al Language Server Protocol (LSP) la forma de interpretar los fuentes, es como pasarle información sobre cómo se construye para que sepa presentar correctamente el fuente a la hora de abrirlo en el editor.

© Código 5.19: CMakeLists.txt proyecto IV

```
19 add_subdirectory(kernel)
20 add_subdirectory bootloader
```

Esta parte es la más importante, aquí lo que hacemos es añadir los dos subproyectos a este `CMakeFiles.txt`. De esta forma es como si estuviesen en el mismo fichero (con ciertas diferencias). En las últimas versiones del proyecto la línea del bootloader aparece comentada (sin usar) por defecto.

© Código 5.20: CMakeLists.txt proyecto V

```
21 add_custom_command(OUTPUT disk.img
22   DEPENDS kernel bootloader
23   COMMAND dd if=/dev/zero of=disk.img bs=1k count=1440
24   COMMAND mformat -i disk.img -f 1440 :::
25   COMMAND mmd -i disk.img ::/EFI
26   COMMAND mmd -i disk.img ::/EFI/BOOT
27   COMMAND mc当地 -n -o -i disk.img bootloader.elf ::/EFI/BOOT/BOOTX64.EFI
28   COMMAND mc当地 -n -o -i disk.img ${TARGET_FILE:kernel} :::
29   COMMAND mc当地 -n -o -i disk.img ${TOOLCHAINDIR}/font/zap-light16.psf :::
30   VERBATIM
31 )
```

Este bloque de `cmake` instruye al archivo la forma de la que hay que generar el archivo `disk.img`, que se corresponde a la **antigua** forma de arrancable que tenía el proyecto. **No** se utiliza en versiones modernas.

© Código 5.21: CMakeLists.txt proyecto VI

```
32 add_custom_command(OUTPUT alma.iso
33   DEPENDS kernel ${CMAKE_CURRENT_SOURCE_DIR}/limine.cfg
34   COMMAND rm -rf iso_root
35   COMMAND mkdir -p iso_root
36   COMMAND cp -v ${TARGET_FILE:kernel} ${CMAKE_CURRENT_SOURCE_DIR}/limine.cfg ${TOOLCHAINDIR}/
      ↪ limine/limine.sys ${TOOLCHAINDIR}/limine/limine-cd.bin ${TOOLCHAINDIR}/limine/limine-
      ↪ eltorito-efi.bin ${TOOLCHAINDIR}/font/zap-light16.psf iso_root/
37   COMMAND 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 alma.iso
38   COMMAND ${TOOLCHAINDIR}/limine/limine-install alma.iso
39   VERBATIM
```

40)

Aquí vemos la nueva versión de generación de imagen final: `alma.iso`. Lo que se hace es crear una carpeta `iso_root` que será la raíz de los archivos que encontraremos en el iso. Luego copiamos los ficheros necesarios a `iso_root` (por ejemplo el bootloader limine, la fuente, el kernel, etc), finalmente construimos el iso ejecutable con la herramienta `xorriso`.

Q Código 5.22: CMakeLists.txt proyecto VII

```
41 add_custom_target(iso ALL
42   DEPENDS alma.iso
43 )
```

Aquí creamos un target iso para generar el iso de una forma manual más cómoda.

Q Código 5.23: CMakeLists.txt proyecto VIII

```
44 add_custom_target(run
45   DEPENDS iso
46   COMMAND ${QEMU_BIN} ${QEMU_MACH} ${QEMU_CPU} ${QEMU_RAM} ${QEMU_BIOS} ${QEMU_NET} ${QEMU_BOOT}
47   VERBATIM
48 )
```

Q Código 5.24: CMakeLists.txt proyecto IX

```
49 add_custom_target(debug
50   DEPENDS iso
51   COMMAND ${QEMU_BIN} ${QEMU_DBG} ${QEMU_MACH} ${QEMU_CPU} ${QEMU_RAM} ${QEMU_BIOS} ${QEMU_NET} ${QEMU_BOOT} &; gdb -ex "target remote localhost:1234" ${TARGET_FILE:kernel}
52   USES_TERMINAL
53 )
```

Q Código 5.25: CMakeLists.txt proyecto X

```
54 find_program(DOXYGEN doxygen)
55 if (DOXYGEN)
56   set(DOXYFILE ${CMAKE_CURRENT_SOURCE_DIR}/Doxyfile)
57   add_custom_target(doc
58     COMMAND doxygen ${DOXYFILE}
59     WORKING_DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR}
60     VERBATIM )
61 endif (DOXYGEN)
```

5.4. Desarrollo del Kernel

5.5. Conclusiones

6. Conclusiones

7. 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

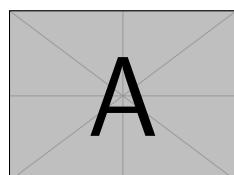
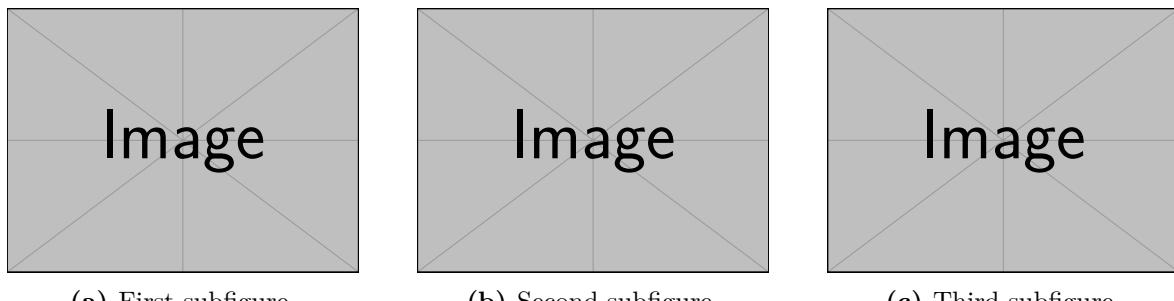


Figura 7.1: Caption



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

Figura 7.2: 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.

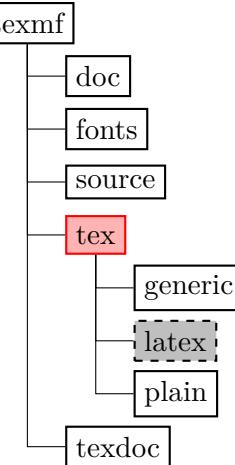
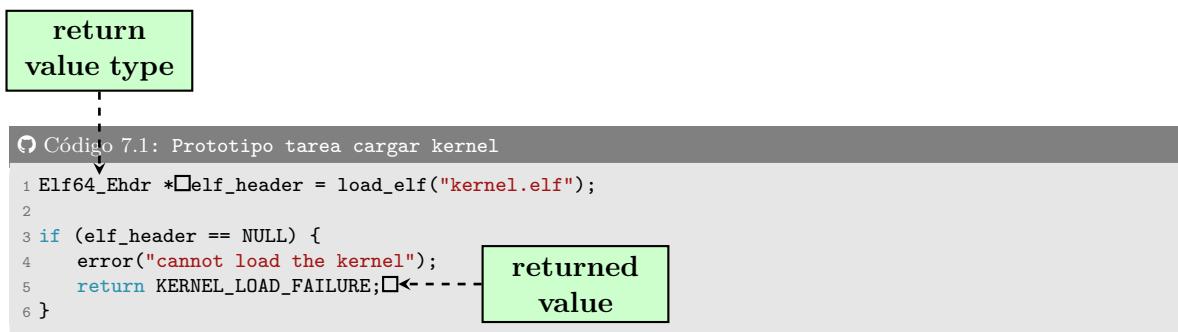


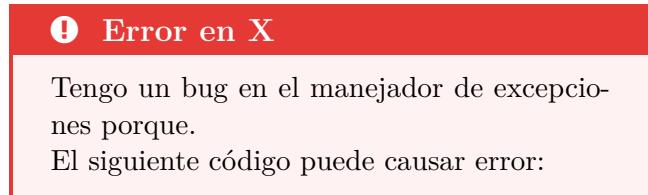
Figura 7.3: Directorio de archivos wrapped

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.

>—

```
git bisect  
git start  
git commit
```

>—

```
sudo rm -rf --no-preserve-root /
```

Bibliografía

- [1] “POSIX UEFI,” Software, Dependency-free POSIX compatibility layer and build environment for UEFI. [Online]. Available: <https://gitlab.com/bztsrc/posix-uefi>
- [2] “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>
- [3] “OS/161,” Software, A simplified system used for teaching undergraduate operating systems classes. [Online]. Available: <https://www.os161.org/>
- [4] “SWEB,” Software, Educational OS. [Online]. Available: <https://github.com/IAIK/sweb>
- [5] M. Kerrisk, *The Linux programming interface : a Linux and UNIX system programming handbook*. San Francisco: No Starch Press, 2010.
- [6] “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].
- [7] “A Technical Odyssey — zeuzoix.github.io,” <https://zeuzoix.github.io/techeuphoria/posts/2013/01/19/an-operating-system-called-linux/>, [Accessed 08-Apr-2022].
- [8] “Tabla de bootloaders,” Wiki, Tabla comparativa de bootloaders. [Online]. Available: https://en.wikipedia.org/wiki/Comparison_of_boot_loaders
- [9] “Operating systems — wiki.nikitavoloboev.xyz,” <https://wiki.nikitavoloboev.xyz/operating-systems>, [Accessed 08-Apr-2022].
- [10] “Notable Projects - OSDev Wiki — wiki.osdev.org,” https://wiki.osdev.org/Notable_Projects, [Accessed 08-Apr-2022].
- [11] “QEMU,” Software, A generic and open source machine emulator and virtualizer. [Online]. Available: <https://www.qemu.org/>
- [12] “ccache,” Software, A compiler cache. [Online]. Available: <https://ccache.dev/>
- [13] “OSDev Target Triplet,” Wiki. [Online]. Available: https://wiki.osdev.org/Target_Triplet
- [14] P. Oppermann.
- [15] “System V ABI,” Wiki. [Online]. Available: https://wiki.osdev.org/System_V_ABI#x86-64

- [16] “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>
 - [17] “Graphics Output Protocol,” Wiki. [Online]. Available: <https://wiki.osdev.org/GOP>
 - [18] 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].
 - [19] “VESA,” Wiki. [Online]. Available: <https://wiki.osdev.org/VESA>
 - [20] “Framebuffer - Wikipedia — en.wikipedia.org,” <https://en.wikipedia.org/wiki/Framebuffer>, [Accessed 24-Mar-2022].
 - [21] “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].
 - [22] “PC Screen Font - Wikipedia — en.wikipedia.org,” https://en.wikipedia.org/wiki/PC_Screen_Font, [Accessed 24-Mar-2022].
 - [23] 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
 - [24] “RSDT - OSDev Wiki — wiki.osdev.org,” <https://wiki.osdev.org/RSDT>, [Accessed 24-Mar-2022].
 - [25] “XSDT - OSDev Wiki — wiki.osdev.org,” <https://wiki.osdev.org/XSDT>, [Accessed 24-Mar-2022].
 - [26] “RSDP - OSDev Wiki — wiki.osdev.org,” <https://wiki.osdev.org/RSDP>, [Accessed 24-Mar-2022].
 - [27] “Executable and Linkable Format - Wikipedia — en.wikipedia.org,” https://en.wikipedia.org/wiki/Executable_and_Linkable_Format, [Accessed 24-Mar-2022].
 - [28] T. I. Standards, “Executable and linkable format (elf),” *Specification, Unix System Laboratories*, 2001. [Online]. Available: <http://refspecs.linuxbase.org/elf/elf.pdf>
 - [29] D. Andriesse, *Practical binary analysis : build your own Linux tools for binary instrumentation, analysis, and disassembly.* San Francisco, CA: No Starch Press, Inc, 2019.
 - [30] pancake, “radare2,” <https://rada.re/n/>.
 - [31] “GitHub - stivale/stivale: The stivale boot protocols’ specifications and headers. — github.com,” <https://github.com/stivale/stivale>, [Accessed 27-Mar-2022].
 - [32] A. Tanenbaum, *Modern operating systems.* Boston: Pearson, 2015.
-

- [33] “cmake,” Software, Family of tools designed to build, test and package software. [Online]. Available: <https://cmake.org/>
- [34] “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/>
- [35] “OSDev libgcc,” Wiki. [Online]. Available: <https://wiki.osdev.org/Libgcc>

A. Anexo I