



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

Mayo 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, Mayo 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	80
5.4.1.	Código y Entorno Principal	81
5.4.2.	Inicialización	85
5.4.2.1.	Reserva de Páginas	86
5.4.2.2.	Memoria Virtual	86
5.4.2.3.	Memoria Dinámica	87
5.4.2.4.	Pantalla	87

5.4.2.5. Segmentación	89
5.4.2.6. Interrupciones	89
5.4.2.7. Teclado PS2	91
5.4.2.8. ACPI	91
5.4.2.9. PCI	92
5.4.2.10. Tarjeta de Red RTL8139	92
5.4.3. Gestión de Memoria	93
5.4.3.1. Reserva de Páginas	94
5.4.3.2. Mejor Reserva de Páginas	99
5.4.4. Memoria Virtual	105
5.4.5. Memoria Dinámica	112
5.4.5.1. Trivial Allocator	112
5.4.5.2. Simple Allocator	113
5.4.6. Pantalla	118
5.4.6.1. Renderer	118
5.4.6.2. Renderer Simple	119
5.4.6.3. Renderer Rápido	125
5.4.6.4. Fuente PSF1	129
5.4.7. Segmentación	131
5.4.8. Bus I/O	133
5.4.9. Interrupciones	135
5.4.9.1. Interrupt Handlers	138
5.4.10. Teclado PS2	140
5.4.11. ACPI	147
5.4.12. PCI Express	151
5.4.13. Tarjeta de Red RTL8139	157
5.4.14. Interfaz	161
5.4.14.1. Intérprete	161
5.4.14.2. Comandos	163
5.4.15. Librerías	173
5.5. Conclusiones	174
6. Conclusiones	175
Bibliografía	177
A. Anexo I	181

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
5.2. Proceso de Inicialización de <i>alma</i>	85
5.3. Dirección Virtual en Intel x86-64 (4-level Mapping)	105
5.4. Proceso de traducción de una dirección virtual (Fuente: Wikimedia)	106
5.5. Prueba de Funcionamiento de la Memoria Virtual	111
5.6. Puertos PS/2	140
5.7. Formato de la tabla RSDP/XSDT	148
5.8. Ejemplo de <code>print_acpi_tables()</code> en <code>qemu</code>	150
5.9. Conexiones PCI y PCIe de un ordenador	151
5.10. Listado de dispositivos PCI en <i>alma</i> (<code>qemu</code>)	156
5.11. Tarjeta de red RTL8139	157
5.12. Funcionamiento del driver RTL8139 en <i>alma</i>	160
5.13. Ejemplo de ejecución de comandos en <i>alma</i>	172

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
5.3. Lista de Interrupciones [11]	135
5.4. Tablas ACPI [12, p. 112]	149

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_hedaer	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 [13]	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. crt.i.asm	73
5.14. crt.n.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
5.26. kernel.cpp I	81
5.27. kernel.cpp II _start(stivale2_struct *)	82
5.28. kernel.h	83
5.29. Código de inicialización de alma	85

5.30. startup.cpp – void allocator(stivale2_struct)	86
5.31. startup.cpp – void translator(stivale2_struct)	86
5.32. startup.cpp – void enable_virtualaddr()	87
5.33. startup.cpp – void heap(size_t)	87
5.34. startup.cpp – void screen(stivale2_struct)	87
5.35. startup.cpp – void gdt()	89
5.36. gdt.h – table	89
5.37. startup.cpp – void interrupts()	90
5.38. startup.cpp – void enable_interrupts()	90
5.39. startup.cpp – void keyboard()	91
5.40. startup.cpp – void acpi()	91
5.41. startup.cpp – void pci()	92
5.42. startup.cpp – void rtl18139()	92
5.43. Tipos de memoria en stivale	93
5.44. PFA.h – class PFA	94
5.45. PFA.cpp – PFA::PFA(stivale2_struct_tag_memmap)	95
5.46. PFA.cpp – PFA::get_largest_segment(stivale2_struct_tag_memmap) . .	95
5.47. PFA.cpp – PFA::zero_bitset()	96
5.48. PFA.cpp – PFA::free_page(void *)	96
5.49. PFA.cpp – PFA::free_pages(void *, uint64_t)	96
5.50. PFA.cpp – PFA::lock_page(void *)	96
5.51. PFA.cpp – PFA::lock_pages(void *, uint64_t)	97
5.52. PFA.cpp – PFA::reserve_page(void *)	97
5.53. PFA.cpp – PFA::reserve_pages(void *, uint64_t)	97
5.54. PFA.cpp – PFA::release_page(void *)	97
5.55. PFA.cpp – PFA::release_pages(void *, uint64_t)	98
5.56. PFA.cpp – PFA::request_page()	98
5.57. BPFA.h – struct BPFA_page	99
5.58. BPFA.h – class BPFA	99
5.59. BPFA.cpp – BPFA::BPFA(stivale2_struct_tag_memmap)	100
5.60. BPFA.cpp – BPFA::lock_page(uint64_t)	101
5.61. BPFA.cpp – BPFA::lock_pages(uint64_t, uint64_t)	102
5.62. BPFA.cpp – BPFA::free_page(uint64_t)	102
5.63. BPFA.cpp – BPFA::free_pages(uint64_t, uint64_t)	103
5.64. BPFA.cpp – BPFA::request_page(void *)	103
5.65. BPFA.cpp – BPFA::new_node()	103
5.66. BPFA.cpp – BPFA_page::remove_node()	104
5.67. BPFA.cpp – BPFA_page::split(uint64_t, BPFA_page *)	104
5.68. BPFA.cpp – BPFA::request_cont_page(uint32_t)	104
5.69. address.h – struct address_t	107
5.70. PTM.h – struct page_global_dir_entry_t	107
5.71. PTM.h – struct page_upper_dir_entry_t	107
5.72. PTM.h – struct page_mid_dir_entry_t	108
5.73. PTM.h – struct page_table_entry_t	108
5.74. PTM.h – struct PGDT_wrapper	108

5.75. PTM.h – class PTM	108
5.76. PTM.cpp – PTM::PTM()	109
5.77. PTM.cpp – PTM::map(uint64_t, uint64_t)	109
5.78. allocator_i.h – class allocator_i	112
5.79. trivial_allocator.h – class trivial_allocator	112
5.80. trivial_allocator.cpp – trivial_allocator::malloc(uint64_t)	112
5.81. trivial_allocator.cpp – trivial_allocator::free(void *)	113
5.82. simple_allocator.h – class simple_allocator	114
5.83. simple_allocator.h – simple_allocator::simple_allocator(uint64_t)	114
5.84. simple_allocator.cpp – simple_allocator::malloc(uint64_t)	115
5.85. simple_allocator.cpp – simple_allocator::free(void *)	116
5.86. simple_allocator.cpp – simple_allocator::expand_heap(uint64_t) . .	116
5.87. simple_allocator.cpp – simple_allocator::heap_header::split(uiint64_t)	117
5.88. simple_allocator.cpp – simple_allocator::combine_forward(heap_header *)	117
5.89. simple_allocator.cpp – simple_allocator::combine_backward(heap_header *)	117
5.90. renderer_i.h – class renderer_i	118
5.91. simple_renderer_i.h – class simple_renderer_i	119
5.92. simple_renderer_i.cpp – simple_renderer_i::simple_renderer_i(...)	119
5.93. simple_renderer_i.cpp – simple_renderer_i::put(const char)	120
5.94. simple_renderer_i.cpp – simple_renderer_i::print(const char *, int64_n)	120
5.95. simple_renderer_i.cpp – simple_renderer_i::println(const char *) .	121
5.96. simple_renderer_i.cpp – simple_renderer_i::clear()	121
5.97. simple_renderer_i.cpp – simple_renderer_i::scroll()	121
5.98. simple_renderer_i.cpp – simple_renderer_i::fmt(const char *) . . .	122
5.99. simple_renderer_i.cpp – simple_renderer_i::pushColor(color_e) . . .	123
5.100. simple_renderer_i.cpp – simple_renderer_i::popColor()	124
5.101. simple_renderer_i.cpp – simple_renderer_i::draw_pixel(uint32_t, uint32_t)	124
5.102. simple_renderer_i.cpp – simple_renderer_i::pushCoords(uint32_t, uint32_t)	124
5.103. simple_renderer_i.cpp – simple_renderer_i::popCoords()	124
5.104. fast_renderer_i.h – class fast_renderer_i	125
5.105. fast_renderer_i.cpp – fast_renderer_i::fast_renderer_i(...)	126
5.106. fast_renderer_i.cpp – fast_renderer_i::clear()	126
5.107. fast_renderer_i.cpp – fast_renderer_i::scroll()	127
5.108. fast_renderer_i.cpp – fast_renderer_i::update_video()	127
5.109. fast_renderer_i.cpp – fast_renderer_i::draw_pixel(uint32_t, uint32_t)	128
5.110. fast_renderer_i.cpp – fast_renderer_i::get_pixel(uint32_t, uint32_t)	128
5.111. psf1.h – struct psf1_header	129
5.112. psf1.h – struct psf1	129
5.113. psf1.h – class psf1	130
5.114. gdt.h – struct gdt_entry	131
5.115. gdt.h – struct gdt_ptr	131
5.116. gdt.h – const gdt_entry table[]	132
5.117. gdt.asm – load_gdt(gdt_ptr *)	132

5.118bus.h	133
5.119bus.cpp – outb(uint16_t, uint8_t)	134
5.120bus.cpp – outb(io::port, uint8_t)	134
5.121bus.cpp – inb(uint16_t)	134
5.122bus.cpp – inb(io::port)	134
5.123bus.cpp – io_wait()	134
5.124IDT.h – struct idt_ptr	136
5.125IDT.h – struct idt_entry	136
5.126IDT.h – struct addr_t	136
5.127IDT.cpp – idt_ptr::remap_pic(uint8_t, uint8_t)	137
5.128IDT.cpp – idt_ptr::add_handle(interrupts::vector_e, void (*) (frame *))	137
5.129IDT.cpp – idt_entry::set_offset(uint64_t)	138
5.130IDT.cpp – idt_entry::get_offset()	138
5.131interrupts.h	139
5.132interrupts.cpp – reserved(frame *)	139
5.133interrupts.cpp – keyboard(frame *)	139
5.134interrupts.cpp – ethernet(frame *)	139
5.135keyboard.h – enum class PS2_State	140
5.136keyboard.h – const char PS2_SCANCODES[] [4]	141
5.137keyboard.h – class PS2	141
5.138keyboard.cpp – PS2::process_scancode(uint8_t)	142
5.139keyboard.cpp – PS2::delete_char(uint16_t)	144
5.140keyboard.cpp – PS2::add_char(char)	144
5.141keyboard.cpp – PS2::update()	144
5.142keyboard.cpp – PS2::scanf(char *, uint32_t)	144
5.143keyboard.cpp – PS2::update_scanf()	145
5.144keyboard.cpp – PS2::enable_keyboard()	146
5.145acpi.h – struct rsdp_v1	147
5.146acpi.h – struct rsdp_v2	147
5.147acpi.h – struct sdt	148
5.148acpi.cpp – rsdp_v2::find_table(const char *)	149
5.149acpi.cpp – sdt::check_signature(const char *)	150
5.150acpi.cpp – rsdp_v2::print_acpi_tables()	150
5.151pci.h – struct device_config	152
5.152pci.h – struct device_header	152
5.153pci.h – struct header_t0	152
5.154pci.h – struct header_t1	153
5.155pci.h – struct header_t2	153
5.156pci.h – struct pci_device	154
5.157pci.h – struct BAR_mem	154
5.158pci.h – struct BAR_io	154
5.159pci.cpp – enum_pci(acpi::sdt *)	154
5.160pci.cpp – enum_bus(uint64_t, uint64_t)	155
5.161pci.cpp – enum_dev(uint64_t, uint64_t)	155

5.162pci.cpp – enum_fun(uint64_t, uint64_t)	156
5.163rtl18139.h – enum rtl18139_config	157
5.164rtl18139.h – class rtl18139	158
5.165rtl18139.cpp – rtl18139::rtl18139(pci::pci_device *)	159
5.166rtl18139.cpp – rtl18139::start()	159
5.167rtl18139.cpp – rtl18139::send_packet(uint32_t, uint64_t)	160
5.168interpreter.h – class interpreter	161
5.169interpreter.cpp – interpreter::process(char *)	162
5.170interpreter.cpp – interpreter::launch_command(char *, int, char **)	162
5.171command.h – struct command	163
5.172command.h – static const command []	163
5.173command.h – help(int, char **)	164
5.174command.h – echo(int, char **)	164
5.175command.h – shell(int, char **)	164
5.176command.h – clear(int, char **)	165
5.177command.h – pci(int, char **)	165
5.178command.h – getpage(int, char **)	165
5.179command.h – getmac(int, char **)	165
5.180command.h – getphys(int, char **)	166
5.181command.h – map(int, char **)	167
5.182command.h – unmap(int, char **)	167
5.183command.h – set(int, char **)	167
5.184command.h – get(int, char **)	168
5.185command.h – printmem(int, char **)	168
5.186command.h – uefimmap(int, char **)	169
5.187command.h – printpfa(int, char **)	170
5.188command.h – checknet(int, char **)	170
5.189command.h – sendpacket(int, char **)	171
5.190command.h – screen(int, char **)	171
5.191command.h – acpi(int, char **)	172

1. Introducción

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

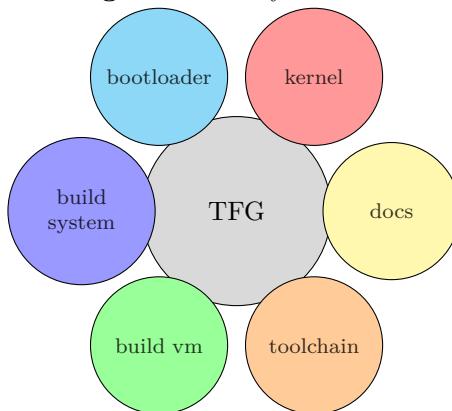
1.1. Propuesta

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 2.2.6 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.

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.

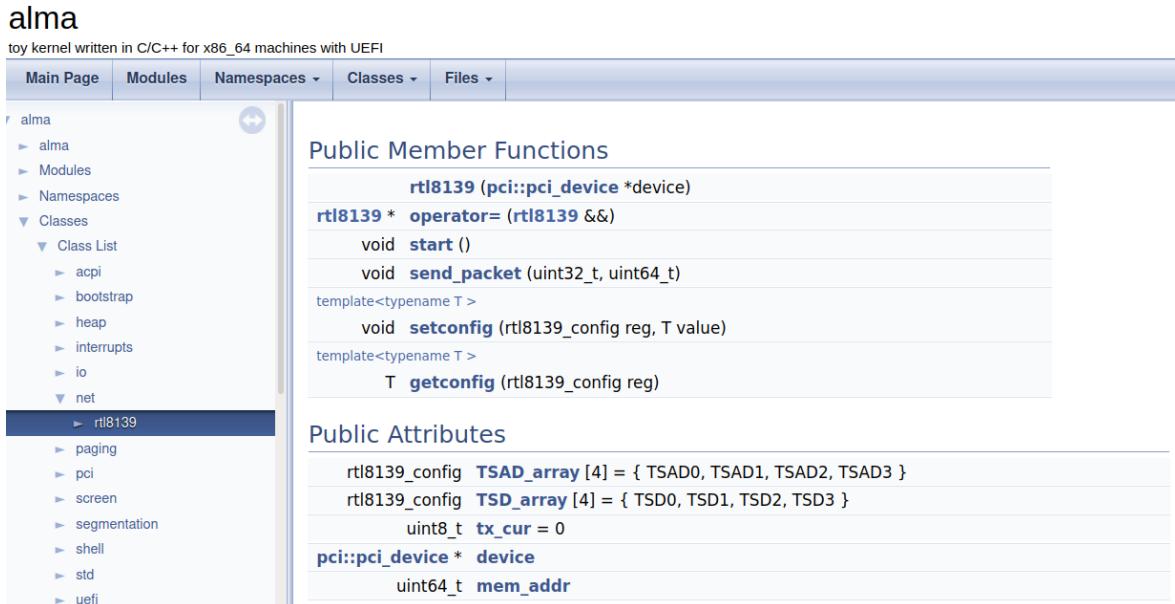


Figura 2.2: Resultado de la generación de la documentación con doxygen

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` [14]. 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

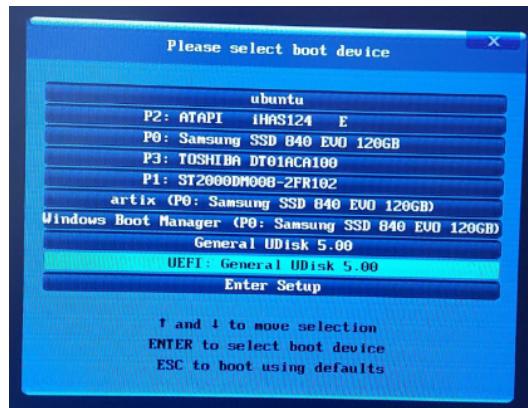
El proyecto inicialmente se planteó como para ser probado únicamente en entornos simulados, pero en las últimas versiones del proyecto se planteó que también pudiese ser arrancado en máquinas reales.

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 - 8c28 (27, 0, 0)
* 8086 - 8c18 (28, 0, 0)
* 8086 - 8c16 (28, 0, 4)
* 8086 - 8c14 (28, 0, 5)
* 8086 - 8c12 (28, 0, 6)
* 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* [15] 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).

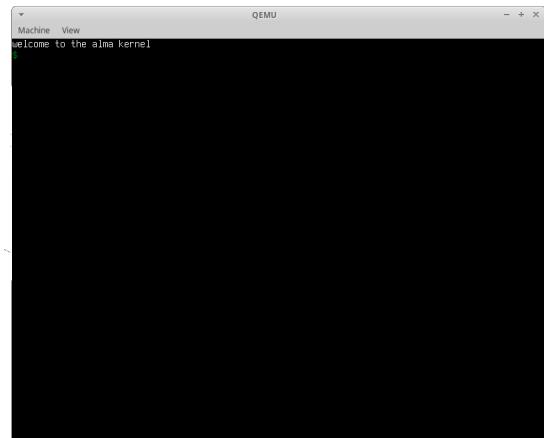


Figura 3.3: alma ejecutado en la “alma build VM”

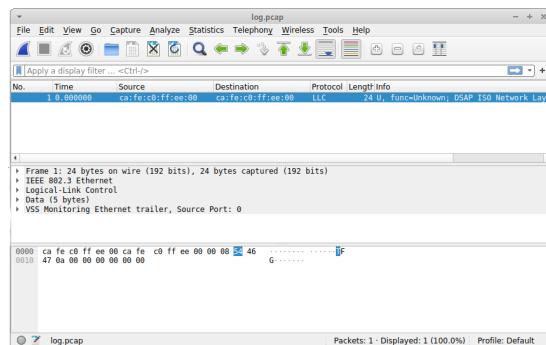


Figura 3.2: alma con Wireshark en la “alma build VM”

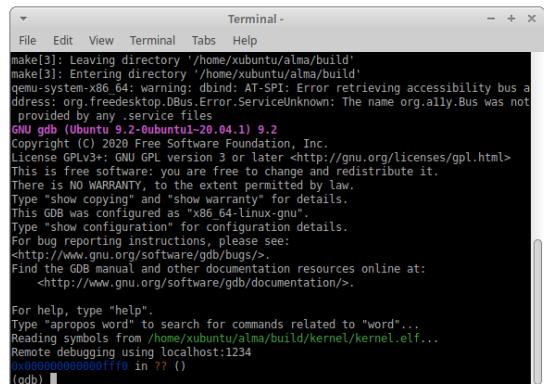


Figura 3.4: alma con gdb en la “alma build VM”

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 [16]. 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 [17] 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.

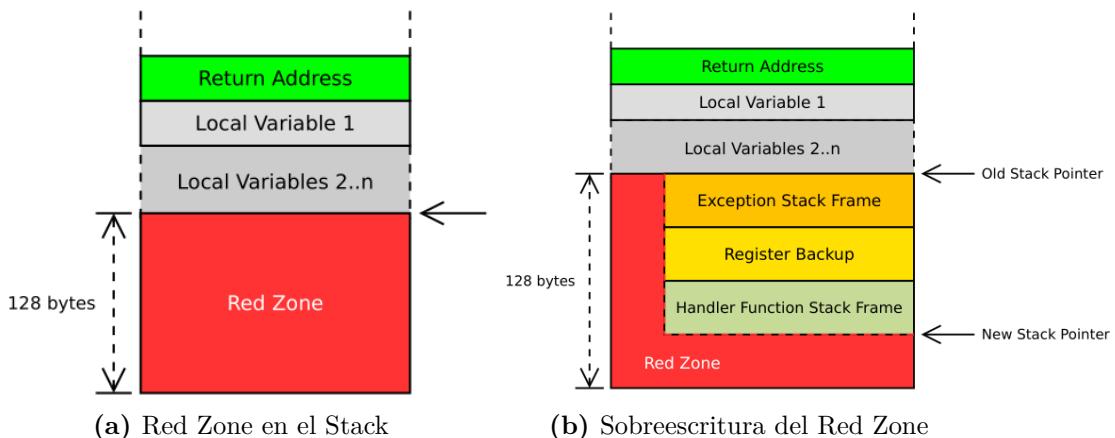


Figura 3.5: Red Zone en el Stack (<https://os.phil-opp.com>)

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 sobreesciba y provoque *undefined behaviour* como podemos ver en la figura 3.5b.

⁵El stack en la ABI de System V crece “hacia abajo” [18]

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.

```
① 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 [19].

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.

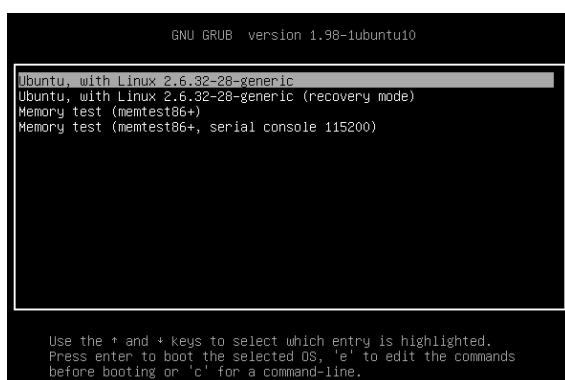
Se ha modificado la forma de la que se utiliza `posix-uefi`, se ha integrado en el sistema de construcción de alma y se ha descartado el `Makefile` que provee el proyecto. Esto mejora el sistema de construcción y reduce las dependencias con makefiles no propios, esto se explicará en la sección 4.3.

4. Bootloader

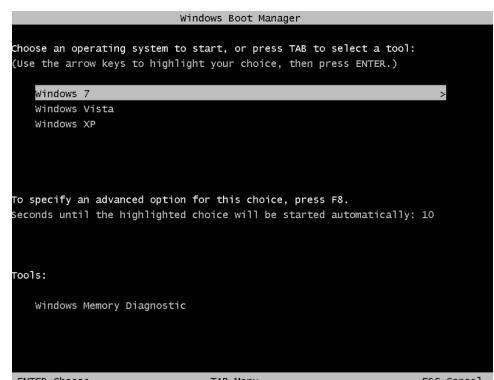
En este capítulo se presenta el desarrollo del bootloader para *alma*. Este bootloader, escrito en C para arranque UEFI se utilizaba en versiones iniciales de *alma*, actualmente está en desuso debido a mejoras hechas en el kernel. En este capítulo se introduce el concepto y teoría sobre los bootloaders 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

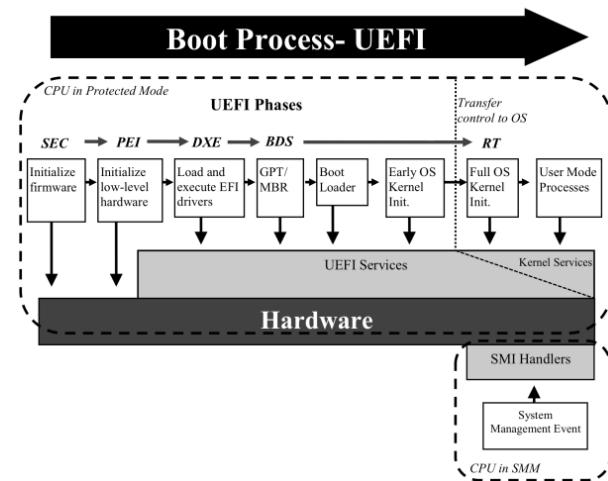


Figura 4.2: Arranque UEFI

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 encontrremos, 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

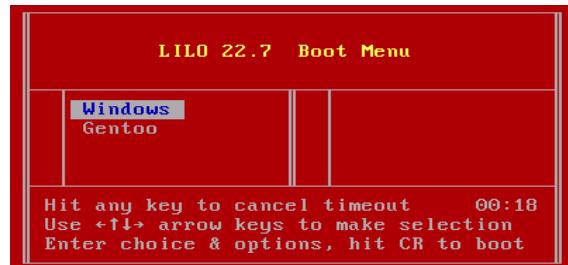
4.2. Estado del Arte

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.

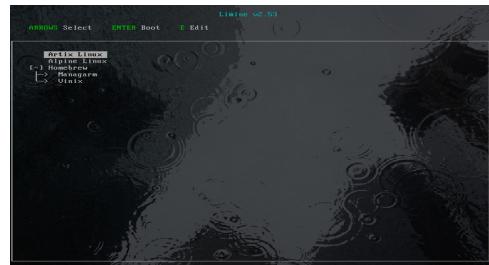
Bootloader	Arquitectura	Estándar/Formato
GRUB Legacy	x86 (PC)	Multiboot 1, Linux zImage, Linux bzImage, etc.
GRUB2	x86 (PC, EFI) IA-64, ARM (U-Boot, UEFI), PowerPC	Multiboot, etc.
LILO	x86 (PC)	Linux zImage, Linux bzImage
SYSLINUX	x86 (PC)	Linux zImage, Linux bzImage, Multiboot MBR
Yaboot	PowerPC	Linux 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.



(a) LILO



(b) Limine

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()`:

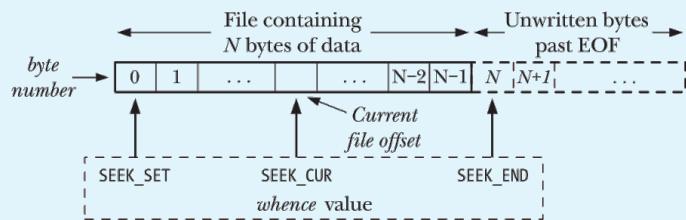


Figura 4.4: Funcionamiento SEEK_SET, SEEK_CUR y SEEK_END [5]

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 [20, 21] es el estándar utilizado por UEFI que reemplazó el estándar VESA [22] (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 [23] 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->ppsc = 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.

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 [24]:

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 [25]). 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:

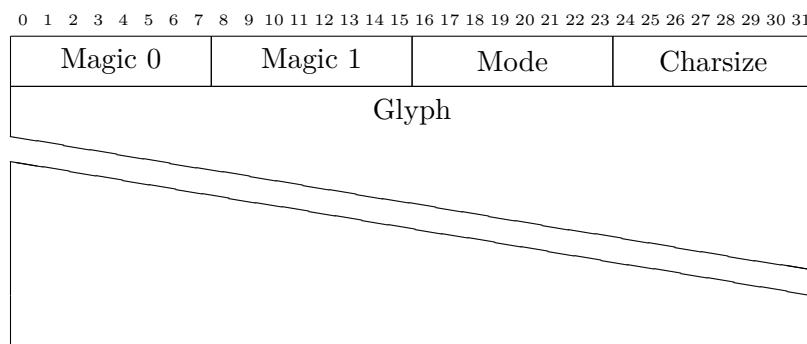


Figura 4.6: Fuente PSF1 [6]

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 *******
00000030 40 00 00 00 00 00 00 00 08 10 20 40 20 10 08 00 @0000000 .. @ ..0
00000040 7c 00 00 00 00 00 00 00 10 08 04 02 04 08 10 00 |0000000 *******0
```

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

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

Código 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 [26]) 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 [27] y el XSDT [28] (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 [29]. 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 [30] y más tarde en el “Tool Interface Standard” [31].

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 */
```

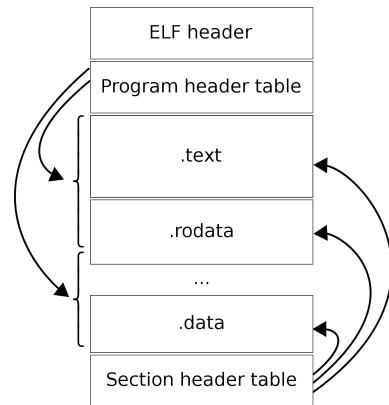


Figura 4.7: Formato ELF

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 [32]: 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` [33]) 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* [13] 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 [34] 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.

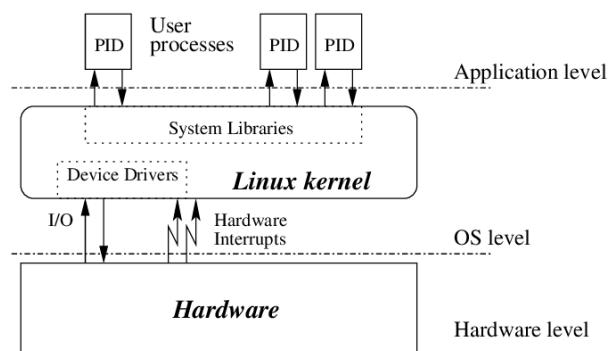


Figura 5.1: Arquitectura del Kernel Linux [7]

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 funcionalidadada 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 un 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* [35].

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 "${CMAKE_CURRENT_SOURCE_DIR}/kernel.1d")
28 SET(CMAKE_EXE_LINKER_FLAGS "${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: kernel.1d [13]

```

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 "-mgeneral-reg-only ${←
  ↪ COMPILE_FLAGS}" )
```

Aquí lo único que hacemos es indicarle a cmake que el código fuente de las interrupciones lo compile con una opción adicional.

```
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 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.

© 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}/
37     ↪ limine/limine.sys ${TOOLCHAINDIR}/limine/limine-cd.bin ${TOOLCHAINDIR}/limine/limine-
38     ↪ eltorito-efi.bin ${TOOLCHAINDIR}/font/zap-light16.psf iso_root/
39   COMMAND xorriso -as mkisofs -b limine-cd.bin -no-emul-boot -boot-load-size 4 -boot-info-table --
39     ↪ efi-boot limine-eltorito-efi.bin -efi-boot-part --efi-boot-image --protective-msdos-
39     ↪ label iso_root -o alma.iso
40   COMMAND ${TOOLCHAINDIR}/limine/limine-install alma.iso
41   VERBATIM
42 )
```

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`.

© 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 forma manual más cómoda, si se ejecuta `make iso` llamará a generar `alma.iso` tal como le hemos especificado anteriormente.

© 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 )
```

Este target representa la ejecución simulada de `alma`, cuando se ejecuta `make -C build run` se construye `alma.iso` y se simula mediante `qemu`.

© 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 )
```

Aquí creamos un target de debug que ejecute `qemu` con un servidor de `gdb` abierto en `localhost:1234` (opción `-s -S` de `qemu`), luego abrimos `gdb` ejecutando el comando `target remote localhost:1234` para conectarnos a dicho servidor, también abrimos el archivo ELF del kernel para poder leer los símbolos de debug.

© 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)
```

Finalmente, lo que hacemos es crear un nuevo target llamado “`doc`” si y solo si el usuario tiene `doxygen` instalado, si se ejecuta `make -C build doc` se construirá la documentación.

```
>_ cd build; make

[ 5%] Built target crtS
[ 97%] Built target kernel
[100%] Generating alma.iso
'/home/ecomaikgolf/alma/build/kernel/kernel.elf' -> 'iso_root/kernel.elf'
'/home/ecomaikgolf/alma/limine.cfg' -> 'iso_root/limine.cfg'
'/home/ecomaikgolf/alma/toolchain/limine/limine.sys' -> 'iso_root/limine.sys'
'/home/ecomaikgolf/alma/toolchain/limine/limine-cd.bin' ->
↳ 'iso_root/limine-cd.bin'
'/home/ecomaikgolf/alma/toolchain/limine/limine-eltorito-efi.bin' ->
↳ 'iso_root/limine-eltorito-efi.bin'
'/home/ecomaikgolf/alma/toolchain/font/zap-light16.psf' ->
↳ 'iso_root/zap-light16.psf'
xorriso 1.5.4 : RockRidge filesystem manipulator, libburnia project.

Drive current: -outdev 'stdio:alma.iso'
Media current: stdio file, overwriteable
Media status : is blank
Media summary: 0 sessions, 0 data blocks, 0 data, 882g free
Added to ISO image: directory '://'='/mnt/HDD/Projects/alma/build/iso_root'
xorriso : UPDATE :       6 files added in 1 seconds
xorriso : UPDATE :       6 files added in 1 seconds
ISO image produced: 1246 sectors
Written to medium : 1246 sectors at LBA 0
Writing to 'stdio:alma.iso' completed successfully.

Physical block size of 512 bytes.
Installing to GPT. Logical block size of 512 bytes.
Secondary header at LBA 0x1377.
Secondary header valid.
GPT partition NOT specified. Attempting GPT embedding.
New maximum count of partition entries: 44.
Stage 2 to be located at 0x1a00 and 0x268800.
Reminder: Remember to copy the limine.sys file in either
          the root or /boot directories of one of the partitions
          on the device, or boot will fail!
Limine installed successfully!
[100%] Built target iso
```

```
>_ cd build; make

[ 5%] Built target crtS
[ 97%] Built target kernel
[100%] Built target iso
```

Como vemos, no vuelve a crear el iso si no es necesario.

```
>_ cd build; make run  
[ 5%] Built target crtS  
[ 97%] Built target kernel  
[100%] Built target iso  
[100%] Built target run  
(qemu launches)
```

```
>_ cd build; make debug  
[ 5%] Built target crtS  
[ 97%] Built target kernel  
[100%] Built target iso  
(qemu launches)  
GNU gdb (GDB) 11.2  
(...)
```

```
>_ cd build; make doc  
Doxygen version used: 1.9.3  
Searching for include files...  
Searching for example files...  
Searching for images...  
Searching for dot files...  
(...)
```

```
>_ cd build; make clean  
(build/ gets cleaned from build artifacts)
```

```
>_ cd build; make help  
The following are some of the valid targets for this Makefile:  
... all (the default if no target is provided)  
... clean  
... depend  
... edit_cache  
... rebuild_cache  
... debug  
... doc  
... iso  
... run  
... crtS  
... kernel
```

5.4. Desarrollo del Kernel

Entramos en la sección más importante del trabajo, el desarrollo del kernel. En esta sección se explicará el código desarrollado de *alma* junto a las explicaciones del funcionamiento de funcionalidades del procesador con las que trabajaremos.

En esta sección se encontrará grandes cantidades de código en C++, se ha decidido incluir el código en la memoria para facilitar la lectura y comprensión del trabajo, igualmente el repositorio de código se puede descargar mediante:

>—

```
git clone https://github.com/ecomaikgolf/alma.git
```

Cabe mencionar que se pueden encontrar diferencias entre el código listado en este documento y el repositorio del proyecto. Siempre es recomendable tomar como referencia el repositorio, ya que podemos encontrar código con arreglos incorporados o nuevas y mejores funcionalidades.

La forma de explicar el código será desde los módulos más abstractos (dentro del ámbito del proyecto) hasta los módulos de más bajo nivel. Es por esto que al principio se ha de ver el código como un esquema, se ha de confiar en la implementación de las llamadas a las funciones que aún no se han visto y que se verán después. Si se encuentran llamadas a `malloc` se ha de asumir el funcionamiento básico de `malloc` (o similar), luego se explicará la implementación en el proyecto.

Si el lector lo considera, puede ir alternando la lectura entre las secciones que considere convenientes a lo largo de la explicación del desarrollo, también se recomienda que acompañe esta lectura con el código fuente que se encuentra en el repositorio. Recordar que el código cuenta con comentarios y documentación mediante `doxygen`.

Para realizar modificaciones el código mostrado o realizar trazas con `gdb`, vaya a la Sección 3.2 para trabajar con el proyecto de forma rápida en un entorno de desarrollo virtualizado.

Finalmente comentar, de cara a un lector exigente, que el código ha sido desarrollado bajo unas fuertes restricciones temporales. Al tratarse de un trabajo de fin de grado, este proyecto ha sido realizado con fecha límite y junto a otras asignaturas a trabajar. Estas limitaciones no han supuesto impedimento para desarrollar un proyecto funcional tan complejo y amplio, pero es necesario comprender que el código no está libre de errores e imperfecciones. Los núcleos suelen ser desarrollados por grandes equipos de programadores experimentados en grandes períodos de tiempo con ingentes cantidades de dinero de por medio, este proyecto ha sido desarrollado por un programador inexperimentado en el ámbito del desarrollo de kernels con grandes restricciones de tiempo y sin ayudas económicas. El código tiene aspectos a mejorar, como el de ciertas complejidades asintóticas o de seguridad y diseño, pero no entran dentro de los objetivos de este proyecto.

5.4.1. Código y Entorno Principal

En este apartado se explicará el corazón del núcleo, su función principal: `_start()`. Esta función es la primera que ejecutamos al salir del bootloader puesto que así lo hemos indicado tanto en el enlazador del `kernel.elf` al poner esta función como entry point y porque así se lo hemos indicado al bootloader en la línea 29 del Código 5.26.

 Código 5.26: kernel.cpp I

```

1 #include "kernel.h"
2 #include "bootstrap/startup.h"
3 #include "bootstrap/stivale_hdrs.h"
4 #include "float.h"
5 #include "interrupts/interrupts.h"
6 #include "lib/stdlib.h"
7 #include "lib/string.h"
8 #include "paging/PFA.h"
9 #include "screen/framebuffer.h"
10 #include "shell/command.h"
11 #include "stivale2.h"
12 #include <stddef.h>
13 #include <stdint.h>
14
15 static uint8_t stack[8192];
16 static struct stivale2_header_tag_terminal terminal_hdr_tag = {
17     .tag = { .identifier = STIVALE2_HEADER_TAG_TERMINAL_ID, .next = 0 },
18     .flags = 0
19 };
20 static struct stivale2_header_tag_framebuffer framebuffer_hdr_tag = {
21     .tag = { .identifier = STIVALE2_HEADER_TAG_FRAMEBUFFER_ID,
22             .next = (uint64_t)&terminal_hdr_tag },
23     /* Best effort pick */
24     .framebuffer_width = 0,
25     .framebuffer_height = 0,
26     .framebuffer_bpp = 0
27 };
28 __attribute__((section(".stivale2hdr"), used)) static struct stivale2_header stivale_hdr = {
29     .entry_point = 0,
30     .stack = (uintptr_t)stack + sizeof(stack),
31     .flags = (1 << 1) | (1 << 2) | (1 << 3) | (1 << 4),
32     .tags = (uintptr_t)&framebuffer_hdr_tag
33 };
34
35 extern "C" void _init();
36 extern "C" void _fini();

```

Este código corresponde con el inicio del archivo `kernel.cpp`. En la línea 15 creamos un espacio vacío para el stack y le decimos a stivale que use este espacio como stack en la línea 30 (podríamos establecer cualquier otro espacio en la memoria más complejo, este es el caso más simple). El bootloader encargado de cargar el código será el responsable de que el stack vaya donde le hemos dicho.

Luego, desde la línea 16 hasta la línea 33 tenemos cabeceras de configuración de stivale. La cabecera de configuración importante es `stivale_hdr` que se guardará en una sección del ELF especial: `.stivale2hdr` para que el bootloader pueda encontrarla y configurar el arranque del kernel. Podemos configurar otra serie de cosas como por ejemplo el tamaño del framebuffer (línea 24), que en este caso se establece a 0 para que el bootloader encuentre la

mejor configuración.

En las líneas 35 y 36 declaramos que tenemos dos funciones externas que se nos incluirá a la hora de enlazar el kernel. Estas funciones son definidas por el compilador en archivos objeto especiales tal como se explica en la sección 5.3.1. Estas funciones llamarán a los constructores y destructores globales.

```
④ Código 5.27: kernel.cpp II _start(stivale2_struct *)  
37 extern "C" [[noreturn]] void  
38 _start(stivale2_struct *stivale2_struct)  
39 {  
40     /* Call global constructors & functions */  
41     _init();  
42  
43     /* Store the stivale header pointer */  
44     kernel::internal::stivalehdr = *stivale2_struct;  
45  
46     /* Bootstrap the kernel (order is important) */  
47     bootstrap::allocator(stivale2_struct);  
48     bootstrap::translator(stivale2_struct);  
49     bootstrap::enable_virtualaddr();  
50     bootstrap::heap(0x10);  
51     bootstrap::screen(stivale2_struct);  
52     bootstrap::gdt();  
53     bootstrap::interrupts();  
54     bootstrap::enable_interrupts();  
55     bootstrap::keyboard();  
56     bootstrap::acpi(stivale2_struct);  
57     bootstrap::pci();  
58     bootstrap::rtl18139();  
59  
60     /* Welcome the user */  
61     kernel::tty.println("welcome to the alma kernel");  
62  
63     /* Start a shell */  
64     shell::commands::shell(0, nullptr);  
65  
66     /* Call global destructors & functions */  
67     _fini();  
68     /* Shouldn't return (poweroff instead) */  
69     __asm__("hlt");  
70     /* To suppress diagnostics */  
71     while (1) {  
72     };  
73 }
```

Este código (5.27) muestra la función principal del kernel. Como ya se ha mencionado anteriormente, es la primera función a ejecutar cuando el bootloader cede el control al kernel.

En la línea 41 llamamos a la función provista por el compilador para llamar a los constructores de los objetos globales que tengamos y funciones constructoras que definamos. La llamada a los destructores, de la misma manera, la podemos ver en la línea 67.

La línea 44 guarda el puntero provisto por el bootloader (**stivale2_struct**) en las variables globales de *alma* (explicado a continuación). Este puntero conforma una lista enlazada de “tags” definidas por stivale que contienen información pasada por el bootloader.

Las líneas 47–58 son las más importantes de la función, estas inicializan el núcleo. Estas funciones son lo que conocemos como el proceso de arranque en cualquier sistema operativo, es una serie de funciones ejecutadas secuencialmente (su orden es importante) que van inicializando partes del kernel. Por ejemplo `bootstrap::rt18139()` es el encargado de “arrancar” la tarjeta de red y dejarla lista para poder enviar paquetes en partes posteriores del código.

En la línea 61 y 64 encontramos la inicialización de la interfaz que provee *alma*. En primer lugar se da la bienvenida al usuario al núcleo y acto seguido se arranca una shell interactiva en la que el usuario puede introducir comandos. Esta shell en un principio no tiene forma de finalizarse.

Las líneas 67–71 se encargan de finalizar el kernel. Esto realmente no es así puesto que un núcleo no puede acabar su función, como hemos visto al definir la función hemos usado `[[noreturn]]` porque los núcleos no pueden volver de la función a ningún lado, la máquina debe apagarse en vez de volver de la función. Es por eso que ponemos que la CPU pare en caso de que se llegue y `while(1)` para suprimir los diagnósticos de las herramientas de análisis estático.

Ahora procederemos a explicar los contenidos de la cabecera `kernel.h` (Código 5.28). La cabecera de `alma` la considero el estado global del kernel, es donde se almacenan las variables con las que trabaja el resto de módulos. Si se quiere solicitar una página de memoria al kernel se ha de acudir a esta cabecera, al igual que si se quiere consultar o modificar constantes¹⁰ del kernel.

© Código 5.28: `kernel.h`

```

1 #pragma once
2
3 #include "acpi/acpi.h"
4 #include "heap/allocator_i.h"
5 #include "heap/simple_allocator.h"
6 #include "heap/trivial_allocator.h"
7 #include "interrupts/IDT.h"
8 #include "io/keyboard.h"
9 #include "net/rt18139.h"
10 #include "paging/BPFA.h"
11 #include "paging/PTM.h"
12 #include "pci/pci.h"
13 #include "screen/fast_renderer_i.h"
14 #include "screen/fonts/psf1.h"
15 #include "screen/simple_renderer_i.h"
16 #include "segmentation/gdt.h"
17 #include "shell/interpreter.h"
18 #include <stdint.h>
19
20 namespace kernel {
21
22 namespace internal {
23
24 /* Set by the linker */
25 extern "C" uint64_t _start_addr;
26 extern "C" uint64_t _end_addr;
27
28 inline stivale2_struct stivalehdr;
```

¹⁰Al editar el código

```

29 } // namespace internal
31
32 /* System Constants */
33 const auto page_size = uefi::page_size;
34
35 /* Variables */
36 inline paging::allocator::BPFA allocator;
37 inline paging::translator::PTM translator __attribute__((aligned(uefi::page_size)));
38 inline screen::fonts::psf1<screen::fast_renderer_i> tty;
39 inline segmentation::gdt_ptr gdt;
40 inline interrupts::idt_ptr idtr;
41 inline io::PS2 keyboard;
42 inline acpi::rsdp_v2 rsdp;
43 inline heap::simple_allocator heap;
44 inline pci::pci_device *devices;
45 inline net::rtl18139 rtl18139;
46
47 /* Kernel Constants */
48 __attribute__((unused)) static void *_start_addr = &internal::_start_addr;
49 __attribute__((unused)) static void *_end_addr = &internal::_end_addr;
50 const uint64_t _size = (size_t)&internal::_end_addr - (size_t)&internal::_start_addr;
51 const uint64_t _size_npages = _size / page_size + 1;
52
53 } // namespace kernel

```

En primer lugar incluimos las cabeceras necesarias de otros módulos para poder tener tipos de datos personalizados pertenecientes a otros módulos.

Acto seguido, en las líneas 25 y 26 definimos dos variables externas que serán definidas por el enlazador. Contendrán las direcciones de inicio y fin del kernel (no en memoria) para poder calcular su tamaño en caso de que fuese necesario. Antes, en el bootloader antiguo, era utilizado para reservar la memoria del kernel en el reservador de páginas de memoria.

En la línea 33 definimos las constantes del sistema, en este caso solo tenemos el tamaño de página, que es el mismo que proporciona uefi, 4096 bytes.

De la línea 36 a 45 tenemos las variables globales del kernel. Estas variables representan los recursos del sistema y presentan una interfaz para poder interactuar con estos. Cualquier módulo o código que quiera trabajar con los recursos del sistema (por ejemplo reservar memoria) usará estos objetos. Como podemos ver, todas estas variables son `inline`.

i Especificador `inline` en variables (C++17)

Las variables, a partir de C++17 pueden tener el especificador `inline`. Las variables `inline` con enlazado externo (no estático) permiten tener más de una definición de la misma, siempre que aparezca en diferentes unidades de compilación. Todas estas variables tendrán la misma dirección (serán la misma variable).

Finalmente, en las líneas 48–51 tenemos las definiciones de las constantes del kernel, como por ejemplo el tamaño total y páginas que ocupa.

5.4.2. Inicialización

En esta sección se explicará el proceso de inicialización que sigue *alma*. Este proceso de inicialización es el encargado de trabajar por primera vez con los recursos hardware para inicializar las estructuras de datos y objetos correspondientes para manejar los recursos.



La figura 5.2 representa el proceso de inicialización de *alma* (de izquierda a derecha y de arriba a abajo). Cada uno de estos bloques representa una de las funciones mostradas en el código 5.29, que se encargan de inicializar el módulo que representan.

Código 5.29: Código de inicialización de *alma*

```

1 // (...)
2 bootstrap::allocator(stivale2_struct);
3 bootstrap::translator(stivale2_struct);
4 bootstrap::enable_virtualaddr();
5 bootstrap::heap(0x10);
6 bootstrap::screen(stivale2_struct);
7 bootstrap::gdt();
8 bootstrap::interrupts();
9 bootstrap::enable_interrupts();
10 bootstrap::keyboard();
11 bootstrap::acpi(stivale2_struct);
12 bootstrap::pci();
13 bootstrap::rtl8139();
14 // (...)
  
```

Todo el código relacionado con el proceso de inicialización de *alma* se encuentra en **bootstrap/**. En **bootstrap/startup.cpp** encontraremos las definiciones de todas y cada una de las funciones listadas anteriormente en el código 5.29.

Esta sección dispondrá de las subsecciones listadas en la figura 5.2, correspondientes con las funciones mostradas en el código 5.29. En cada subsección se verá el contenido de cada función y el objetivo que cumple en el proceso de inicialización del núcleo.

Cabe mencionar que por la forma de la que se está explicando el núcleo, no se entrará en detalle sobre cómo funciona cada detalle de este proceso de inicialización, si no el proceso general que se sigue en cada apartado. Los detalles de funcionamiento interno estarán en el apartado correspondiente de cada módulo.

5.4.2.1. Reserva de Páginas

Corresponde con la función `bootstrap::allocator(stivale2_struct)`. Esta función es la encargada de gestionar el mapa de memoria que nos provee EFI para construir un sistema de reserva de páginas que EFI nos haya indicado que pueden ser utilizadas.

```
© Código 5.30: startup.cpp – void allocator(stivale2_struct)

51 void
52 allocator(stivale2_struct *st)
53 {
54     /* stivale pointer to the memory map */
55     auto *map = (stivale2_struct_tag_memmap *)stivale2_get_tag(st, STIVALE2_STRUCT_TAG_MEMMAP_ID);
56
57     /* Construct the allocator with the UEFI mem map */
58     kernel::allocator = paging::allocator::BPFA(map);
59 }
```

Como podemos ver, obtenemos el puntero al mapa de memoria mediante una función auxiliar de `stivale2` y acto seguido construimos un objeto BPFA (Better Page Frame Allocator). Este objeto (*rvalue*) es movido¹¹ al objeto global `kernel::allocator`, finalizando así su inicialización. `kernel::allocator` ya está listo para reservar páginas físicas de memoria.

5.4.2.2. Memoria Virtual

Corresponde con la función `bootstrap::translator(stivale2_struct)`. Esta función es la encargada de inicializar la estructura de datos necesaria para poder disponer de memoria virtual en el sistema.

```
© Código 5.31: startup.cpp – void translator(stivale2_struct)

73 void
74 translator(stivale2_struct *st)
75 {
76     /*
77      * As I'm using limine now I can't create a PGDT from scratch (as limine maps some important
78      * addresses) so what I have to do is take the limine's one and edit it, instead of creating a
79      * empty one.
80     */
81
82     /* Obtain the actual PGDT addr */
83     uint64_t mapaddr;
84     asm volatile("mov %%cr3, %0" : [Var] "=r"(mapaddr));
85     paging::translator::PGDT_wrapper *newpgdt = (paging::translator::PGDT_wrapper *)mapaddr;
86
87     /* Provide it to the translator */
88     kernel::translator.set_PGDT(newpgdt);
89 }
```

Esta función es la que ha sufrido el mayor cambio al ser movido de un bootloader propio al bootloader limine. Antes se creaba un árbol de 4 niveles y 512 nodos en cada nivel (la

¹¹Movido, no copiado.

estructura de datos necesaria), ahora esto no es posible porque usamos otro bootloader. Lo que hacemos en este caso es obtener un puntero almacenado en la CPU a dicha estructura de datos (registro CR3) establecido por el bootloader y se lo pasamos a nuestro gestor de memoria virtual.

De esta manera, hemos “copiado” la configuración de la que disponía el bootloader para evitar romper su configuración (necesaria para mantener el sistema) y la tenemos disponible en nuestro objeto para realizar modificaciones y cambios adicionales.

© Código 5.32: startup.cpp — void enable_virtualaddr()

```

1 void
2 enable_virtualaddr()
3 {
4     /* Enable virtual addresses */
5     asm("mov %0, %%cr3" : : "r"(kernel::translator.get_PGDT()));
6 }
```

Actualmente la llamada a esta función es redundante puesto que el bootloader nos deja el registro CR3 ya configurado. Lo que hace esta función es “activar” la inicialización de nuestro objeto gestor de memoria virtual en la CPU. Como ya se ha dicho, esta llamada no es estrictamente necesaria con limine, si lo era en el anterior bootloader.

5.4.2.3. Memoria Dinámica

Corresponde con la función `bootstrap::heap(size_t)`. Esta función es la encargada de inicializar un objeto que sea capaz de gestionar reservas y liberaciones de memoria.

© Código 5.33: startup.cpp — void heap(size_t)

```

1 void
2 heap(size_t size)
3 {
4     /* Create the heap */
5     kernel::heap = heap::simple_allocator(size);
6 }
```

Lo único que se hace es construir un objeto gestor del heap de tamaño `size` y asignarselo al objeto global, quedándose este inicializado. Como se verá posteriormente, podemos seleccionar entre varias clases como objeto gestor del heap (simple/trivial allocator).

5.4.2.4. Pantalla

Corresponde con la función `bootstrap::screen(stivale2_struct)`. Esta función es la encargada de inicializar la pantalla para poder escribir texto en ella. Gestiona de forma interna la posición del texto que se va imprimiendo, los saltos de línea cuando no cabe más texto en la línea y el scroll cuando no caben más líneas de texto.

© Código 5.34: startup.cpp – void screen(stivale2_struct)

```

20 void
21 screen(stivale2_struct *st)
22 {
23     using namespace screen;
24
25     /* Stivale pointers to the framebuffer & modules */
26     auto *fb =
27         (stivale2_struct_tag_framebuffer *)stivale2_get_tag(st, STIVALE2_STRUCT_TAG_FRAMEBUFFER_ID);
28     auto *mod = (stivale2_struct_tag_modules *)stivale2_get_tag(st, STIVALE2_STRUCT_TAG_MODULES_ID);
29
30     /* Create a copy of our framebuffer struct from the data provided by stivale (other format) */
31     framebuffer frame;
32     frame.base = (unsigned int *)fb->framebuffer_addr;
33     frame.ppscl = (fb->framebuffer_pitch / sizeof(uint32_t));
34     frame.width = fb->framebuffer_width;
35     frame.height = fb->framebuffer_height;
36     frame.buffer_size = frame.ppscl * frame.height * sizeof(uint32_t);
37
38     /* Get the font module from stivale */
39     fonts::specification::psf1 *font_ptr =
40         (fonts::specification::psf1 *)stivale2_get_mod(mod, "font");
41
42     /* Create the font */
43     fonts::specification::psf1 font;
44     font.header = *(fonts::specification::psf1_header *)font_ptr;
45     font.buffer = (uint8_t *)font_ptr + sizeof(fonts::specification::psf1_header);
46
47     /* Create the tty */
48     kernel::tty = fonts::psf1<screen::fast_renderer_i>(frame, font, 0, 0, screen::color_e::WHITE);
49 }
```

Esta función ha sufrido un cambio drástico al cambiar del bootloader propio a limine. Actualmente tiene como función principal ser una capa de compatibilidad entre la estructura de datos del framebuffer provisionada por limine y el renderer de *alma*, ya que este se escribió basándose en estructuras de datos propias en el otro bootloader.

En las líneas 27 y 28 obtenemos punteros al framebuffer y a los módulos del bootloader. Acto seguido, entre las líneas 31 y 36 transformamos la estructura framebuffer de limine a una versión propia que es capaz de entender nuestro renderer.

En la línea 40 obtenemos un puntero a la fuente PSF1 y luego en las líneas 43–45 creamos la estructura de datos que representa la fuente PSF1 en *alma*.

Finalmente, en la línea 48 creamos el objeto `kernel::tty` encargado de gestionar la pantalla. Como podemos ver especificamos una template con `screen::fast_renderer_i`, eso es porque podemos seleccionar entre distintas formas de renderizar los píxeles. También en el constructor vemos el color por defecto (`screen::color_e::WHITE`) y la posición inicial del “cursor de texto” (0,0).

5.4.2.5. Segmentación

Corresponde con la función `bootstrap::gdt()`. Esta función es la encargada de inicializar un entorno segmentado básico en la máquina creando y cargando una Global Descriptor Table (GDT).

```
Q Código 5.35: startup.cpp - void gdt()
61 void
62 gdt()
63 {
64     using namespace segmentation;
65
66     /* Create an empty GDT */
67     kernel::gdt.size = sizeof(table) - 1;
68     kernel::gdt.offset = (uint64_t)&table;
69     /* Load it (assembly) */
70     load_gdt(&kernel::gdt);
71 }
```

Como podemos ver, en las líneas 67 y 68 se crea un puntero a una GDT para en la línea 70 cargárselo a la CPU. La tabla que se carga es la siguiente:

```
Q Código 5.36: gdt.h - table
84 __attribute__((aligned(uefi::page_size)))
85 const gdt_entry table[] = {
86     // Kernel null
87     { 0, 0, 0, 0, 0, 0 },
88     // Kernel code
89     { 0, 0, 0, 0x9a, 0xa0, 0 },
90     // Kernel data
91     { 0, 0, 0, 0x92, 0xa0, 0 },
92     // User null descriptor
93     { 0, 0, 0, 0, 0, 0 },
94     // User code
95     { 0, 0, 0, 0x9a, 0xa0, 0 },
96     // User data
97     { 0, 0, 0, 0x92, 0xa0, 0 },
98 };
```

Se explicará más a fondo los detalles en su sección, pero no es una segmentación funcional, ya que disponemos de la paginación que la sustituye. Además limine establecerá su propia segmentación (sin uso real), por lo que tampoco es necesario cambiarla.

5.4.2.6. Interrupciones

Corresponde con `bootstrap::interrupts()`. Esta función es la encargada de inicializar el entorno necesario para poder utilizar las interrupciones de la CPU. También se incluye la función `bootstrap::enable_interrupts()`, encargada de habilitar las interrupciones anteriormente inicializadas. Si solo se llama a la primera función las interrupciones no funcionarán, hay que habilitarlas. El momento en el que se habiliten ha de ser posterior a la inicialización, quitando esa restricción da igual cuando se activen.

```
© Código 5.37: startup.cpp – void interrupts()

1 void
2 interrupts()
3 {
4     /* Reserve memory for the interrupt array */
5     kernel::idtr.set_ptr((uint64_t)kernel::allocator.request_page());
6
7     /* Load the interrupt handlers */
8     kernel::idtr.add_handle(interrupts::vector_e::reserved, interrupts::reserved);
9
10    /*
11     * From OSDev:
12     * In protected mode, the IRQs 0 to 7 conflict with the CPU exception which are reserved
13     * by Intel up until 0x1F. It is thus recommended to change the PIC's offsets (also known
14     * as remapping the PIC) so that IRQs use non-reserved vectors. A common choice is to move
15     * them to the beginning of the available range (IRQs 0..0xF -> INT 0x20..0x2F). For that,
16     * we need to set the master PIC's offset to 0x20 and the slave's to 0x28.
17     */
18     kernel::idtr.remap_pic(0x20, 0x28);
19
20     io::outb(io::PIC1_DATA, 0b11111101); // enable IRQ1 from PIC1 (keyboard)
21     io::outb(io::PIC2_DATA, 0b11111111); // mask all lines
22 }
```

En primer lugar reservamos memoria para una estructura de datos interna del gestor de las interrupciones. Acto seguido en la línea 98 registramos nuestra primera interrupción, cuando se levante 0x9 (`interrupts::vector_e::reserved`) se ejecutará la función `interrupts::reserved`.

Acto seguido en la línea 108 reorganizamos los offsets para que no utilicen vectores reservados. Esto se hace por recomendación de un foro de desarrolladores de sistemas operativos (osdev.org).

Finalmente, en las líneas 110 y 111 activamos o desactivamos las IRQs que nos interesen de cada chip de interrupciones. El PIC2 (slave) lo desactivamos entero (todo a 1s) y del primero dejamos el IRQ del teclado.

```
© Código 5.38: startup.cpp – void enable_interrupts()

121 void
122 enable_interrupts()
123 {
124     asm("lidt %0" : : "m"(kernel::idtr));
125
126     /* Enable interrupts */
127     asm("sti");
128 }
```

Esta función es la encargada de activar las interrupciones. Con `lidt`¹² cargamos una interrupt descriptor table en la CPU y con `sti`¹³ establecemos la “interrupt flag”, que activa el reconocimiento de todas las interrupciones hardware.

¹²<https://web.itu.edu.tr/kesgin/mul06/intel/instr/lidt.html>

¹³<https://web.itu.edu.tr/kesgin/mul06/intel/instr/sti.html>

5.4.2.7. Teclado PS2

Corresponde con `bootstrap::keyboard()`. Esta función es la encargada de inicializar el entorno del teclado PS2 para ser capaz de recibir mediante interrupciones las pulsaciones del teclado y almacenarlas en un buffer circular.

```
④ Código 5.39: startup.cpp — void keyboard()

131 void
132 keyboard()
133 {
134     /* Reserve keyboard buffer memory */
135     auto size = KEYBOARD_BUFF_SIZE;
136     auto buffer = kernel::allocator.request_cont_page(size / kernel::page_size + 1);
137
138     /* Bootstrap the keyboard and enable it */
139     kernel::keyboard.set_buffer(static_cast<char *>(buffer));
140     kernel::keyboard.set_maxsize(size);
141     io::PS2::enable_keyboard();
142 }
```

En la línea 135–140 reservamos un buffer de un tamaño determinado para usarlo como almacén de pulsaciones del teclado. Acto seguido llamamos a la función de activación de teclado `io::PS2::enable_keyboard()` para empezar a recibir en el buffer las pulsaciones del teclado.

5.4.2.8. ACPI

Corresponde con la función `bootstrap::acpi(stivale2_struct)`. Esta función es la encargada de obtener el Root System Descriptor Pointer (RSDP) para que los siguientes módulos puedan utilizarlo.

```
④ Código 5.40: startup.cpp — void acpi()

144 void
145 acpi(stivale2_struct *st)
146 {
147     /* Find and set the rsdp */
148     auto *rsdp = (stivale2_struct_tag_rsdp *)stivale2_get_tag(st, STIVALE2_STRUCT_TAG_RSDP_ID);
149     kernel::rsdp = *(acpi::rsdp_v2 *)rsdp->rsdp;
150 }
```

En la línea 148 obtenemos el root system descriptor pointer a partir de la estructura que nos pasa el bootloader, acto seguido en la línea 149 lo hacemos disponible en poniéndolo en la variable global del kernel. De esta forma otros módulos pueden acceder al puntero sin trabajar directamente con stivale.

Se puede encontrar la definición de la estructura de datos del RSDP y su significado y funcionamiento en la sección 4.4.6.

Se puede cambiar la referencia a la del kernel

5.4.2.9. PCI

Corresponde con la función `bootstrap::pci()`. Esta función es la encargada de inicializar la lista enlazada de dispositivos PCI que tiene la máquina.

```
© Código 5.41: startup.cpp - void pci()

152 void
153 pci()
154 {
155     /* Find and set the MCFG table */
156     acpi::sdt *mcfg_ptr = (acpi::sdt *)kernel::rsdp.find_table("MCFG");
157     /* Enumerate all installed PCI devices */
158     pci::enum_pci(mcfg_ptr);
159 }
```

En la línea 156 obtenemos la tabla ACPI con la firma: “MCFG”. Acto seguido en la línea 158 enumeramos todos los dispositivos PCI y a la vez, en la misma función, inicializamos la lista enlazada de dispositivos PCI disponibles. Esta lista se puede acceder desde las variables globales del kernel con `kernel::devices`.

5.4.2.10. Tarjeta de Red RTL8139

Corresponde con la función `boostrap::rtl8139`. Esta función es la encargada de inicializar la tarjeta de red instalada en la máquina (RTL8139).

```
© Código 5.42: startup.cpp - void rtl8139()

168 void
169 rtl8139()
170 {
171     /* From the PCI devices linked list, find the RTL8139 and bootstrap it */
172     for (pci::pci_device *i = kernel::devices; i != nullptr; i = i->next) {
173         if (i->header->id == kernel::rtl8139.PCI_ID && i->header->header_type == 0x0) {
174             kernel::rtl8139 = net::rtl8139(i);
175             kernel::rtl8139.start();
176             return;
177         }
178     }
179 }
```

Lo que hacemos es iterar la lista enlazada de dispositivos PCI disponibles (línea 172). Si encontramos el RTL8139 (línea 173) creamos un objeto `net::rtl8139` (línea 174) y se lo asignamos la variable global del kernel. Finalmente en la línea 175 arrancamos el dispositivo y lo ponemos en funcionamiento.

Cabe mencionar que el bloque de inicialización solo funcionará en caso de que la tarjeta instalada sea una Realtek 8139, si no, no debería de encontrarse en la línea 173, ya que cada uno debe tener un ID distinto (primera condición del `if`). Esto se hace así porque el driver para una tarjeta de red no es válido para otras, por lo que en caso de no encontrarse la tarjeta para la que se ha desarrollado el drive, no se hace nada.

5.4.3. Gestión de Memoria

En esta sección se explicará el código de *alma* relacionado con la gestión de páginas de memoria. Tanto esta sección como la sección 5.4.4 están muy ligadas y se recomienda leer ambas para entender correctamente la gestión de memoria en *alma*.

Lo primero que se hace al inicializar *alma* (sección 5.4.2) es crear un mapa de memoria de la RAM de la máquina. Lo que tenemos que hacer (el objetivo) es dotar a ese mapa de memoria un orden y garantizar que se puede reservar y liberar memoria de tal forma que no se solapen entre si. Si no disponemos de un gestor de memoria tendríamos que usar directamente direcciones de memoria para almacenar datos, lo que podría causar errores si otros módulos deciden usar la misma dirección de memoria para guardar sus datos (se sobreescrivirían).

alma dispone de dos clases que cumplen con este cometido, son clases que actúan como gestores de la memoria. Pese a que no existe protección de memoria y se puede escribir libremente en cualquier dirección de memoria, este gestor garantiza que si ambos módulos solicitan memoria a través de este gestor, la memoria usada por ambos módulos nunca se solapará. Las dos clases encargadas de esta tarea son PFA (*paging/PFA.h*) y BPFA (*paging/BPFA.h*), siendo esta última la versión por defecto utilizada en *alma*.

La gestión de memoria de este apartado trabaja con páginas, las páginas son bloques de memoria de 4096 bytes (en este caso, en otros puede tener distintos tamaños). La memoria RAM de la máquina se divide en estos bloques de memoria. El gestor de memoria debe tratar la gestión a nivel de página, es decir, a nivel de bloques de 4096 bytes y con la dirección inicial alineada también a 4096 bytes.

Cabe mencionar que este apartado no trata de implementar `malloc()` ni `free()`, este apartado trata la implementación de un sistema de más bajo nivel, sobre el que se sostenta todo.

El gestor de memoria debe tener en cuenta que existe alguien que ha ejecutado anteriormente en la CPU y ha hecho cambios en la RAM: el bootloader. El bootloader, en este caso limine, ha hecho reservas de memoria y detectado inicialmente el mapa de memoria (provisto por los servicios de EFI). limine, mediante el protocolo stivale, nos provee de este mapa de memoria para saber que rangos de memoria podemos usar y cuales no. stivale clasifica los rangos de memoria en los siguientes tipos:

```
④ Código 5.43: Tipos de memoria en stivale
0 enum stivale2_mmap_type : uint32_t {
1   USABLE = 1,
2   RESERVED = 2,
3   ACPI_RECLAMABLE = 3,
4   ACPI_NVS = 4,
5   BAD_MEMORY = 5,
6   BOOTLOADER_RECLAMABLE = 0x1000,
7   KERNEL_AND_MODULES = 0x1001,
8   FRAMEBUFFER = 0x1002
9 };
```

5.4.3.1. Reserva de Páginas

Esta primera versión del gestor de memoria crea unbitset en el que cada bit representa una página en memoria, desde el principio hasta el final de la memoria en la máquina.

Cuando se quiere bloquear una página, se pone el bit a 1. Cuando un módulo solicita una página, se busca de forma lineal el primer bit a 0 (libre), se calcula su dirección y se devuelve. También se mantienen estadísticas de memoria usada, memoria reservada y memoria libre.

▲ Page Frame Allocator en desuso

El desarrollo de este gestor de memoria se hizo durante las versiones de *alma* con el bootloader propio. Las versiones que incluyen *limine/stivale* están pensadas para funcionar con BPFA (sección 5.4.3.2) pese a que funcionen con PFA. Es posible que debido al port que se tuvo que hacer para que funcione con el nuevo bootloader, la calidad del código de PFA no sea la mejor. Se recomienda usar BPFA.

© Código 5.44: PFA.h – class PFA

```

30 class PFA
31 {
32     public:
33     PFA(stivale2_struct_tag_memmap *);
34     PFA();
35     void operator=(PFA &&);
36     void free_page(void *);
37     void free_pages(void *, uint64_t);
38     void lock_page(void *);
39     void lock_pages(void *, uint64_t);
40     void *request_page();
41     std::bitset page;
42
43     size_t get_free_mem()
44     {
45         return this->free_mem;
46     }
47     size_t get_reserved_mem()
48     {
49         return this->reserved_mem;
50     }
51     size_t get_used_mem()
52     {
53         return this->used_mem;
54     }
55
56     private:
57     void reserve_page(void *);
58     void reserve_pages(void *, uint64_t);
59     void release_page(void *);
60     void release_pages(void *, uint64_t);
61     void zero_bitset();
62     stivale2_mmap_entry get_largest_segment(stivale2_struct_tag_memmap *);
63     size_t free_mem;
64     size_t reserved_mem;
65     size_t used_mem;
66 };

```

Q Código 5.45: PFA.cpp — PFA::PFA(stivale2_struct_tag_memmap)

```

48 PFA::PFA(stivale2_struct_tag_memmap *map)
49 {
50     if (map == NULL)
51         return;
52
53     auto largest = PFA::get_largest_segment(map);
54
55     size_t memsize = get_memsize(map);
56     this->free_mem = memsize;
57     this->reserved_mem = 0;
58     this->used_mem = 0;
59
60     size_t bitset_size = memsize / kernel::page_size;
61     page.set_size(bitset_size);
62     page.set_buffer((uint8_t *)largest.base);
63
64     PFA::zero_bitset();
65
66     for (uint64_t i = 0; i < map->entries; i++) {
67         if (map->memmap[i].type == 1) {
68             this->free_pages((void *)map->memmap[i].base,
69                               map->memmap[i].length / kernel::page_size);
70         }
71     }
72 }
```

En el constructor lo que se hace es buscar un trozo de memoria inicial para almacenar el gestor de memoria (el bitset de páginas) y posteriormente se inicializa el bitset.

En la línea 53 se busca el segmento más largo que esté libre del mapa de memoria que nos ha pasado el bootloader, el bitset se almacenará en esa región. Posteriormente en la línea 64 se inicializa el bitset todo a unos¹⁴, finalmente en la línea 67 se establecen los bits a cero de páginas que el bootloader nos indique que estén libres, es decir, que no haya usado ni que pertenezcan a otros componentes.

Q Código 5.46: PFA.cpp — PFA::get_largest_segment(stivale2_struct_tag_memmap)

```

73 stivale2_mmap_entry
74 PFA::get_largest_segment(stivale2_struct_tag_memmap *map)
75 {
76     size_t largest_segment_size = 0;
77     uint64_t largest_segment = 0;
78     for (uint64_t i = 0; i < map->entries; i++) {
79         if (map->memmap[i].type == 1 && map->memmap[i].length > largest_segment_size)
80             largest_segment = i;
81     }
82
83     return map->memmap[largest_segment];
84 }
```

Esta función se encarga de iterar el mapa de memoria de EFI para obtener el segmento más grande que pueda ser utilizado. Esto se utiliza en el bootloader para encontrar el sitio donde guardar el bitset.

¹⁴La función se llama `zero_bitset` porque con el bootloader antiguo se establecía a cero, ahora la lógica está negada.

© Código 5.47: PFA.cpp — PFA::zero_bitset()

```
85 void
86 PFA::zero_bitset()
87 {
88     for (size_t i = 0; i < page.get_size() / 8; i++)
89         page.set(i);
90 }
```

Esta función establece todos los bits delbitset del gestor de memoria a uno, es decir, bloquea todas las páginas de memoria para que no puedan ser reservadas. Esto antes, de ahí el nombre de la función, era al revés, se inicializaba a cero elbitset. Se cambió por seguridad, si por lo que sea hay un fallo y no se puede bloquear una página que no debe ser usada, es mejor que se quede una página bloqueada que pueda ser usada.

© Código 5.48: PFA.cpp — PFA::free_page(void *)

```
91 void
92 PFA::free_page(void *addr)
93 {
94     efi_physical_address_t index = (efi_physical_address_t)addr / kernel::page_size;
95
96     /* already free page */
97     if (!this->page[index])
98         return;
99
100    this->page.unset(index);
101
102    this->free_mem += kernel::page_size;
103    this->used_mem -= kernel::page_size;
104 }
```

Esta función establece a cero el bit delbitset a partir de la dirección de memoria de una página. Para obtener el índice que le corresponde a la dirección de memoria en elbitset, como se ve, solo hay que dividir la dirección entre el tamaño de página. Como el resultado es entero, las direcciones que se le pasan no deben estar alineadas a la página, automáticamente se entenderá que es la primera dirección de memoria de la página.

© Código 5.49: PFA.cpp — PFA::free_pages(void *, uint64_t)

```
105 void
106 PFA::free_pages(void *addr, uint64_t count)
107 {
108     for (uint64_t i = 0; i < count; i++)
109         this->free_page((void *)((uint64_t)addr + (i * kernel::page_size)));
110 }
```

Esta función es auxiliar y se utiliza para liberar páginas consecutivas.

© Código 5.50: PFA.cpp — PFA::lock_page(void *)

```
111 void
112 PFA::lock_page(void *addr)
113 {
114
115     uint64_t index = (efi_physical_address_t)addr / kernel::page_size;
116
117     /* already used page */
```

```

118     if (this->page[index])
119         return;
120
121     this->page.set(index);
122
123     this->free_mem -= kernel::page_size;
124     this->used_mem += kernel::page_size;
125 }
```

Es la función antagónica a la liberación de memoria, el único cambio es establecer el bitset a uno en vez de a cero.

¶ Código 5.51: PFA.cpp – PFA::lock_pages(void *, uint64_t)

```

126 void
127 PFA::lock_pages(void *addr, uint64_t count)
128 {
129     for (uint64_t i = 0; i < count; i++)
130         this->lock_page((void *)((uint64_t)addr + (i * kernel::page_size)));
131 }
```

Esta función es auxiliar y se utiliza para bloquear páginas consecutivas.

¶ Código 5.52: PFA.cpp – PFA::reserve_page(void *)

```

132 void
133 PFA::reserve_page(void *addr)
134 {
135     uint64_t index = (efi_physical_address_t)addr / kernel::page_size;
136
137     /* already free page */
138     if (!this->page[index])
139         return;
140
141     this->page.set(index);
142
143     this->free_mem -= kernel::page_size;
144     this->reserved_mem += kernel::page_size;
145 }
```

Esta función acutalmente está en desuso, actúa igual que `lock_page` pero actualiza contadores distintos. Se utilizaba para contabilizar de forma distinta las páginas que vienen bloqueadas por el bootloader (las que no se pueden utilizar).

¶ Código 5.53: PFA.cpp – PFA::reserve_pages(void *, uint64_t)

```

146 void
147 PFA::reserve_pages(void *addr, uint64_t count)
148 {
149     for (uint64_t i = 0; i < count; i++)
150         this->reserve_page((void *)((uint64_t)addr + (i * kernel::page_size)));
151 }
```

Esta función es auxiliar y se utiliza para reservar páginas consecutivas.

¶ Código 5.54: PFA.cpp – PFA::release_page(void *)

```

152 void
153 PFA::release_page(void *addr)
```

```

154 {
155     uint64_t index = (efi_physical_address_t)addr / kernel::page_size;
156
157     /* already used page */
158     if (this->page[index])
159         return;
160
161     this->page.unset(index);
162
163     this->free_mem += kernel::page_size;
164     this->reserved_mem -= kernel::page_size;
165 }
```

Es la función antagónica a `reserve_page`, actualmente está en desuso.

© Código 5.55: PFA.cpp — PFA::release_pages(void *, uint64_t)

```

166 void
167 PFA::release_pages(void *addr, uint64_t count)
168 {
169     for (uint64_t i = 0; i < count; i++)
170         this->release_page((void *)((uint64_t)addr + (i * kernel::page_size)));
171 }
```

Esta función es auxiliar y se utiliza para liberar páginas consecutivas.

© Código 5.56: PFA.cpp — PFA::request_page()

```

172 void *
173 PFA::request_page()
174 {
175     for (uint64_t i = 0; i < this->page.get_size(); i++) {
176         if (this->page[i])
177             continue;
178
179         this->lock_page((void *)(i * kernel::page_size));
180         return (void *)(i * kernel::page_size);
181     }
182
183     /* No memory */
184     return NULL;
185 }
```

Es la interfaz principal del gestor de memoria. Sigue la solicitud que se busque una página libre, se bloquee para que nadie pueda utilizarla y se devuelva su dirección de memoria.

PFA basado en el trabajo de *Absurdponcho*

Pese a que el código haya sido escrito en su totalidad por mi y con grandes cambios para soportar bootloaders que usen stivale, el trabajo está basado en el modelo de gestor de memoria presentado por “Absurdponcho” en su proyecto de sistema operativo.

Puedes encontrar su trabajo en <https://github.com/Absurdponcho/PonchoOS>

5.4.3.2. Mejor Reserva de Páginas

El “*Better Page Frame Allocator*” (BPFA) es el sucesor del “*Page Frame Allocator*” (PFA) anteriormente. Esta versión es la que se utiliza actualmente en *alma* y está basada en un diseño propio usando una lista enlazada.

```
④ Código 5.57: BPFA.h – struct BPFA_page
23 struct BPFA_page
24 {
25     BPFA_page(BPFA_page *next, BPFA_page *prev, uint64_t addr, uint64_t pages, bool free)
26     : next(next)
27     , prev(prev)
28     , addr(addr)
29     , pages(pages)
30     , occupied(free){};
31
32     BPFA_page *next;
33     BPFA_page *prev;
34     uint64_t addr;
35     uint64_t pages;
36     bool occupied = false;
37
38     void remove_node();
39     void split(uint64_t, BPFA_page *);
40 };
```

La estructura `BPFA_page` representa un nodo de la lista enlazada. Cada nodo es el encargado de aportar información sobre el estado (reservado o no) de un bloque de páginas, con dirección de inicio `addr` y un total de páginas de `pages`. El miembro `occupied` sirve para indicar si el nodo está en uso, esto es debido a la forma de la que se representa la lista enlazada.

```
④ Código 5.58: BPFA.h – class BPFA
47 class BPFA
48 {
49     public:
50     BPFA(stivale2_struct_tag_memmap *);
51     BPFA() = default;
52     BPFA &operator=(BPFA &&);
53     bool free_page(uint64_t);
54     bool free_pages(uint64_t, uint64_t);
55     bool lock_page(uint64_t);
56     bool lock_pages(uint64_t, uint64_t);
57     bool free_page(void *);
58     bool free_pages(void *, uint64_t);
59     bool lock_page(void *);
60     bool lock_pages(void *, uint64_t);
61     void *request_page(void *ptr = nullptr);
62     void *request_cont_page(uint32_t);
63
64     BPFA_page *get_first()
65     {
66         return this->list_first;
67     };
68     BPFA_page *get_last()
69     {
70         return this->list_last;
71     };
72 }
```

```

73 private:
74     BPFA_page *buffer_base;
75     BPFA_page *buffer_next;
76     BPFA_page *buffer_limi;
77
78     BPFA_page *list_first;
79     BPFA_page *list_last;
80     BPFA_page *new_node();
81
82     stivale2_mmap_entry get_largest_segment(stivale2_struct_tag_memmap *);
83     uint64_t get_total_pages(stivale2_struct_tag_memmap *);
84 };

```

Como podemos ver, la forma de trabajar con el BPFA es muy similar que la de PFA. Cambia principalmente el funcionamiento interno y la desaparición de los contadores y de la diferenciación de bloquear la memoria no usable de forma distinta, por el resto es igual.

© Código 5.59: BPFA.cpp — BPFA::BPFA(stivale2_struct_tag_memmap)

```

19 BPFA::BPFA(stivale2_struct_tag_memmap *map)
20 {
21     if (map == NULL)
22         return;
23
24     auto largest = BPFA::get_largest_segment(map);
25     /* Pessimistic approximation */
26     auto total_pages = BPFA::get_total_pages(map);
27
28     this->buffer_base = (BPFA_page *)largest.base;
29     this->buffer_next = this->buffer_base;
30     this->buffer_limi = this->buffer_base + total_pages;
31
32     for (uint64_t i = 0; i < map->entries; i++) {
33
34         if (map->memmap[i].type == 1) {
35             *buffer_next = BPFA_page(nullptr,
36                                     nullptr,
37                                     map->memmap[i].base,
38                                     (uint16_t)(map->memmap[i].length / kernel::page_size),
39                                     true);
40
41         if (buffer_next != this->buffer_base) {
42             buffer_next->prev = (buffer_next - 1);
43             (buffer_next - 1)->next = this->buffer_next;
44         }
45
46         this->list_last = buffer_next;
47         buffer_next++;
48     }
49 }
50
51 this->list_first = this->buffer_base;
52
53 this->lock_pages(this->buffer_base, total_pages);
54 }

```

El BPFA se construye obteniendo, al igual que en el PFA, el mayor bloque de memoria libre que nos indique EFI (línea 24). De este bloque de memoria no se usará su totalidad, lo que se hace es pre-reservar únicamente el espacio que creamos que va a ser necesario en el peor caso (línea 26).

i Obtención del peor caso

Como es una lista enlazada de bloques de páginas, asumí que el peor caso espacial sería en el que hubiese una máxima fragmentación, y cada nodo representase solo una página. Debido a técnicas de desfragmentación que se implementan esto no llegaría a pasar, es por eso que es una cota pesimista, si el núcleo fuese desarrollado con fines comerciales esto se podría recalcular y tomar una cota más apropiada.

cto seguido en la línea 34 se empieza a inicializar la lista enlazada. Se itera sobre el mapa de memoria proporcionado por EFI y se crean los bloques de páginas libres. A la vez se va construyendo la lista enlazada. Finalmente en la línea 53, del gestor de memoria recién creado, se reserva la memoria que hemos usado para la lista enlazada, para evitar que se pueda otorgar por este como si se tratase de memoria libre.

```
④ Código 5.60: BPFA.cpp – BPFA::lock_page(uint64_t)

82 bool
83 BPFA::lock_page(uint64_t addr)
84 {
85     auto iter = this->list_first;
86     do {
87         if (addr >= iter->addr && addr < (iter->addr + (iter->pages * kernel::page_size))) {
88             uint64_t diff = addr - iter->addr;
89             uint16_t pg_diff = diff / kernel::page_size;
90             if (diff == 0) {
91                 if (iter->pages == 1) {
92                     iter->remove_node();
93                     if (iter->prev == nullptr)
94                         this->buffer_base = iter->next;
95                 } else {
96                     iter->addr += kernel::page_size;
97                     iter->pages -= 1;
98                 }
99             } else if (pg_diff == iter->pages - 1) {
100                 iter->pages -= 1;
101             } else {
102                 auto newaddr = this->new_node();
103                 if (newaddr == nullptr)
104                     return false;
105                 iter->split(addr, newaddr);
106             }
107             return true;
108         }
109         iter = iter->next;
110     } while (iter != nullptr);
111
112     return false;
113 }
114 }
```

Esta función se encarga de bloquear una página de memoria. Lo que se hace es iterar la lista enlazada (línea 87) hasta encontrar el bloque de páginas de memoria que contenga esta página. En caso de encontrarla tenemos tres casos, que la dirección inicial del bloque sea la misma que la que queremos bloquear (caso de la línea 90), que la dirección esté justo al final (caso de la línea 99) o que esté por enmedio y tengamos que dividir el nodo 101.

© Código 5.61: BPFA.cpp — BPFA::lock_pages(uint64_t, uint64_t)

```

119 bool
120 BPFA::lock_pages(uint64_t addr, uint64_t pages)
121 {
122     auto it = addr;
123     uint64_t done = 0;
124     for (; pages > 0; pages--) {
125         done++;
126         if (!this->lock_page(it)) {
127             this->free_pages(it, done);
128             return false;
129         }
130         it += kernel::page_size;
131     }
132     return true;
133 }
```

Esta función es auxiliar, sirve para bloquear un número de páginas contiguas en memoria y hacer rollback si alguna de estas no se puede reservar. La función devuelve true si se han podido reservar todas las páginas de memoria, false si no se ha podido y se ha tenido que hacer rollback.

© Código 5.62: BPFA.cpp — BPFA::free_page(uint64_t)

```

140 bool
141 BPFA::free_page(uint64_t addr)
142 {
143     auto it = this->get_first();
144     while (it != nullptr) {
145         if ((it->addr + (it->pages * kernel::page_size)) == addr) {
146             it->pages++;
147             return true;
148         } else if ((it->addr - kernel::page_size) == addr) {
149             it->pages++;
150             it->addr = addr;
151             return true;
152         }
153         it = it->next;
154     }
155
156     auto newpage = this->new_node();
157     if (newpage == nullptr)
158         return false;
159     newpage->addr = addr;
160     newpage->pages = 1;
161     newpage->occupied = 1;
162     newpage->prev = this->list_last;
163     newpage->prev->next = newpage;
164     this->list_last = newpage;
165     return true;
166 }
```

Esta función se encarga de liberar una página reservada. Lo que se hace es iterar primero la lista enlazada para ver si podemos añadir la página que se va a liberar a algún nodo existente y así evitar la fragmentación. Esto puede suceder si la dirección que se va a añadir está al final del nodo (caso de la línea 145) o si está al principio (caso de la línea 148). Si no se puede evitar la fragmentación creamos un nuevo nodo (línea 156) y lo añadimos al final de la lista enlazada.

Q Código 5.63: BPFA.cpp – BPFA::free_pages(uint64_t, uint64_t)

```

171 bool
172 BPFA::free_pages(uint64_t addr, uint64_t pages)
173 {
174     for (; pages > 0; pages--) {
175         if (!this->free_page(addr))
176             return false;
177         addr += kernel::page_size;
178     }
179
180     return true;
181 }
```

Esta función es una función auxiliar que libera un número determinado de páginas contiguas. No necesita de una funcionalidad de rollback.

Q Código 5.64: BPFA.cpp – BPFA::request_page(void *)

```

212 void *
213 BPFA::request_page(void *ptr)
214 {
215     if (ptr != nullptr) {
216         if (!this->lock_page(ptr))
217             return nullptr;
218         return ptr;
219     }
220
221     auto iter = this->list_first;
222     if (iter == nullptr)
223         return nullptr;
224     void *retval = (void *)iter->addr;
225     this->lock_page(iter->addr);
226     return retval;
227 }
```

Esta función solicita una página de memoria al gestor. Para obtener una página de memoria, como mantenemos una lista enlazada de bloques de páginas libre, lo único que hacemos es obtener la dirección de la primera página libre y la bloqueamos. Si la lista enlazada está vacía es que no queda memoria libre en el sistema y se devuelve `nullptr`.

Q Código 5.65: BPFA.cpp – BPFA::new_node()

```

263 BPFA_page *
264 BPFA::new_node()
265 {
266     auto it = this->buffer_next;
267     while (true) {
268         if (it == this->buffer_limi)
269             it = this->buffer_base;
270         if (it->occupied == false) {
271             this->buffer_next = it + 1;
272             return it;
273         }
274         if (it == this->buffer_next - 1)
275             return nullptr;
276         it++;
277     }
278 }
```

Esta función crea un nuevo nodo en la lista enlazada. Lo que hace es encontrar un hueco libre en el buffer (array circular de nodos) para poder crear un nuevo nodo.

© Código 5.66: BPFA.cpp — BPFA_page::remove_node()

```
286 void
287 BPFA_page::remove_node()
288 {
289     this->occupied = false;
290
291     if (this->next != nullptr)
292         this->next->prev = this->prev;
293     if (this->prev != nullptr)
294         this->prev->next = this->next;
295 }
```

Esta función elimina el nodo desde el que se llama, además libera la memoria que ocupa almacenar ese nodo en el buffer de nodos del gestor de memoria.

© Código 5.67: BPFA.cpp — BPFA_page::split(uint64_t, BPFA_page *)

```
303 void
304 BPFA_page::split(uint64_t addr, BPFA_page *newaddr)
305 {
306     uint64_t diff = addr - this->addr;
307     uint16_t pg_diff = diff / kernel::page_size;
308
309     newaddr->addr = addr + kernel::page_size;
310     newaddr->pages = this->pages - pg_diff - 1;
311     newaddr->occupied = true;
312
313     this->pages = pg_diff;
314
315     newaddr->prev = this;
316     newaddr->next = this->next;
317     newaddr->next->prev = this;
318     this->next = newaddr;
319 }
```

Esta función divide el nodo desde la que se llama en dos. Divide a partir de la página con dirección **addr** pasada por parámetro.

© Código 5.68: BPFA.cpp — BPFA::request_cont_page(uint32_t)

```
324 void *
325 BPFA::request_cont_page(uint32_t pages)
326 {
327     auto it = this->get_first();
328     while (it != nullptr) {
329         if (it->pages >= pages) {
330             auto ret = it->addr;
331             this->lock_pages(it->addr, pages);
332             return (void *)ret;
333         }
334         it = it->next;
335     }
336     return nullptr;
337 }
```

Esta función es auxiliar y sirve para solicitar páginas contiguas de memoria.

5.4.4. Memoria Virtual

Este apartado trata sobre el código de *alma* relacionado con la memoria virtual. El código que gestiona la memoria virtual se encuentra en `paging/PTM.cpp` y `paging/PTM.h`.

La memoria virtual es una característica que nos permite asociar direcciones virtuales con direcciones físicas. Cuando se utiliza la memoria virtual todos los accesos a direcciones se traducen por la “Memory Management Unit” (MMU) de la CPU a la dirección física asociada.

Esto nos permite que, en núcleos más avanzados como Linux, podamos tener las mismas direcciones de memoria en distintos procesos. Si lanzamos un proceso *A* y un proceso *B* en Linux podremos ver que ambos tienen las mismas direcciones de memoria, algo que a nuestro entendimiento no se podría dar puesto que al escribir sobre estas se solaparían las escrituras entre ambos procesos (se escribe en la misma dirección de la memoria RAM). Esto se consigue arreglar con la memoria virtual, las direcciones que aparecen son direcciones virtuales que tienen asociadas direcciones físicas, esta vez si, distintas. De esta forma aunque tengamos el mismo rango de memoria, las escrituras se realizan en direcciones físicas distintas.

Existen muchos modelos de esta característica, en este caso se explicará el mecanismo denominado como “4-level Page Mapping”.

Las direcciones virtuales están conformadas por 48 bits, los 16 bits superiores no pueden ser utilizados. La dirección tiene el siguiente formato:

47	39	30	21	12	0
PGDT	PUDT	PMDT	PTDT	Offset	

Figura 5.3: Dirección Virtual en Intel x86-64 (4-level Mapping)

Como tenemos páginas de 4096 bytes necesitamos 12 bits para poder direccionar todo el espacio de memoria ($\log_2(4096) = 12$). Tanto PGDT como PUDT, PMDT y PTDT se usan como índices en una tabla. Cada tabla ocupa una página (4096 bytes) y cada entrada son 8bytes (64 bits) por lo que tenemos que direccionar 512 entradas en la tabla ($4096 / 8$). Para direccionar 512 entradas necesitamos $\log_2(512) = 9$ bits.

Ahora entraremos en la forma de indicarle a la CPU las asociaciones entre direcciones virtuales y físicas. El nombre de “4-level Page Mapping” viene de la estructura de datos usada para estas asociaciones: un mapa *k*-ario de 4 niveles. En este caso, como hemos visto anteriormente, las entradas eran de 512 posiciones, por lo que el árbol es $k = 512$, es decir, cada nodo tendrá exactamente 512 hijos.

Al guardar una asociación en el árbol se accederá al nodo raíz mediante los offsets indicados en la figura 5.3, cuando se llega al nodo hoja se almacenará la dirección física. De esta forma la MMU cuando quiera realizar una traducción lo único que tendrá que hacer es volver a

recorrer este mismo camino hasta el nodo hoja (usando la dirección virtual) hasta encontrar la dirección física almacenada en el nodo hoja.

Para no tener que asociar todas las direcciones dispondremos de una entrada especial (**present**) que nos permite poder tener partes inicializadas y sin inicializar del árbol. De esta forma podemos ir construyendo el árbol de forma dinámica cada vez que realicemos asociaciones nuevas.

Si la MMU de la CPU es incapaz de traducir la dirección virtual por cualquier motivo, ocurre un **Pagefault**. Por ejemplo esto puede suceder si no la tenemos asociada a otra dirección. También suele pasar cuando el árbol está mal construido, muy común a la hora de implementar este módulo.

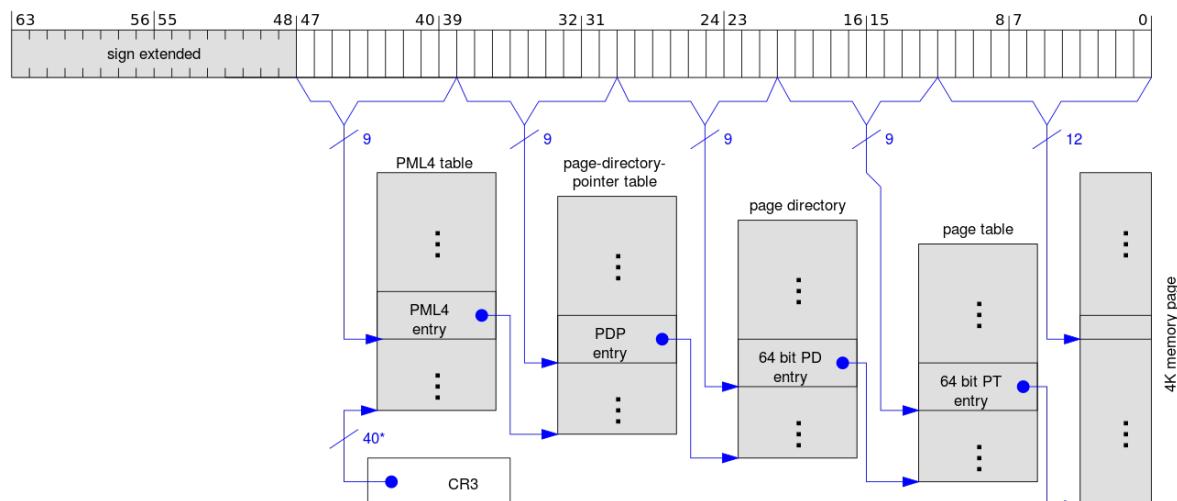


Figura 5.4: Proceso de traducción de una dirección virtual (Fuente: Wikimedia)

⚠ Convención de nombres

A lo largo del trabajo y en *alma* se ha usado una convención distintiva (respecto de otra literatura) de nombres para las tablas que podemos ver en la figura 5.4:

- PGDT = PML4
- PUDT = PDPT
- PMDT = PDT
- PTDT = PT

Como podemos ver, la figura 5.4 se puede ver como recorrer un árbol k -ario de 4 niveles con $k = 512$. El registro *CR3* de la CPU apunta a dicho árbol.

En el directorio `paging/` podemos encontrar el código relacionado con este módulo: `address.h`, `PTM.h` y `PTM.cpp`.

```
④ Código 5.69: address.h — struct address_t
17 struct address_t
18 {
19     uint64_t offset : 12;
20     uint64_t table : 9;
21     uint64_t mid : 9;
22     uint64_t upper : 9;
23     uint64_t global : 9;
24     uint64_t reserved : 16;
25 } __attribute__((__packed__));
```

Esta estructura de datos se utiliza para representar los bits de una dirección virtual (figura 5.3). Luego podemos transformar direcciones en esta estructura mediante `address_t *aux = (address_t *)&dir.`

```
④ Código 5.70: PTM.h — struct page_global_dir_entry_t
37 struct page_global_dir_entry_t
38 {
39     uint64_t present : 1;
40     uint64_t writeable : 1;
41     uint64_t user_access : 1;
42     uint64_t write_through : 1;
43     uint64_t cache_disabled : 1;
44     uint64_t accessed : 1;
45     uint64_t ignored_3 : 1;
46     uint64_t size : 1;
47     uint64_t ignored_2 : 4;
48     uint64_t page_ppn : 28;
49     uint64_t reserved_1 : 12;
50     uint64_t ignored_1 : 11;
51     uint64_t execution_disabled : 1;
52 } __attribute__((__packed__));
```

Esta estructura de datos representa una entrada en la tabla PGDT.

```
④ Código 5.71: PTM.h — struct page_upper_dir_entry_t
56 struct page_upper_dir_entry_t
57 {
58     uint64_t present : 1;
59     uint64_t writeable : 1;
60     uint64_t user_access : 1;
61     uint64_t write_through : 1;
62     uint64_t cache_disabled : 1;
63     uint64_t accessed : 1;
64     uint64_t ignored_3 : 1;
65     uint64_t size : 1;
66     uint64_t ignored_2 : 4;
67     uint64_t page_ppn : 28;
68     uint64_t reserved_1 : 12;
69     uint64_t ignored_1 : 11;
70     uint64_t execution_disabled : 1;
71 } __attribute__((__packed__));
```

Esta estructura de datos representa una entrada en la tabla PUDT.

Q Código 5.72: PTM.h – struct page_mid_dir_entry_t

```

76 struct page_mid_dir_entry_t
77 {
78     uint64_t present : 1;
79     uint64_t writeable : 1;
80     uint64_t user_access : 1;
81     uint64_t write_through : 1;
82     uint64_t cache_disabled : 1;
83     uint64_t accessed : 1;
84     uint64_t ignored_3 : 1;
85     uint64_t size : 1;
86     uint64_t ignored_2 : 4;
87     uint64_t page_ppn : 28;
88     uint64_t reserved_1 : 12;
89     uint64_t ignored_1 : 11;
90     uint64_t execution_disabled : 1;
91 } __attribute__((__packed__));

```

Esta estructura de datos representa una entrada en la tabla PMDT.

Q Código 5.73: PTM.h – struct page_table_entry_t

```

96 struct page_table_entry_t
97 {
98     uint64_t present : 1;
99     uint64_t writeable : 1;
100    uint64_t user_access : 1;
101    uint64_t write_through : 1;
102    uint64_t cache_disabled : 1;
103    uint64_t accessed : 1;
104    uint64_t dirty : 1;
105    uint64_t size : 1;
106    uint64_t global : 1;
107    uint64_t ignored_2 : 3;
108    uint64_t page_ppn : 28;
109    uint64_t reserved_1 : 12;
110    uint64_t ignored_1 : 11;
111    uint64_t execution_disabled : 1;
112 } __attribute__((__packed__));

```

Esta estructura de datos representa una entrada en la tabla PTDT.

Q Código 5.74: PTM.h – struct PGDT_wrapper

```

121 struct PGDT_wrapper
122 {
123     page_global_dir_entry_t PGDT[512];
124 } __attribute__((aligned(uefi::page_size)));

```

PGDT_wrapper es una estructura auxiliar hecha para alinear la estructura de datos de las asociaciones a una página. La MMU de la CPU requiere que esto sea así, si no, no funcionará y el núcleo no arrancará (obtendremos muchos `Pagefault` hasta que se reinicie la máquina). El atributo `__attribute__((aligned(...)))` nos permite alinear la dirección de una estructura a un determinado tamaño, en este caso, a 4096 bytes. De esta forma nos aseguramos que el contenido de la estructura empiece en una página nueva (y por consiguiente que cada tabla empiece en una página nueva).

```
④ Código 5.75: PTM.h — class PTM
129 class PTM
130 {
131     public:
132     PTM();
133     void map(uint64_t, uint64_t);
134     static const uint16_t page_size = 512;
135
136     page_global_dir_entry_t *get_PGDT()
137     {
138         return this->PGD_table->PGDT;
139     }
140
141     void set_PGDT(PGDT_wrapper *PGDT)
142     {
143         this->PGD_table = PGDT;
144     }
145
146     private:
147     PGDT_wrapper *PGD_table;
148 };
```

En PTM.h declaramos la clase del gestor de memoria virtual (PTM). Solo definimos una función, `map(uint64_t, uint64_t)`, que añade el camino necesario en la estructura de traducciones (Figura 5.4) para traducir una memoria virtual a una física, es decir, realiza una asociación entre direcciones.

```
④ Código 5.76: PTM.cpp — PTM::PTM()
23 PTM::PTM()
24 {
25     memset(this->get_PGDT(), 0, uefi::page_size);
26 }
```

El constructor de la clase PTM lo único que hace es inicializar el primer nivel del árbol a 0. El resto no hace falta inicializarlo a 0, se irá haciendo dinámicamente en `PTM::map(uint64_t, uint64_t)`.

ⓘ Almacenamiento e inicialización del árbol

En las últimas versiones de *alma* la estructura no la inicializamos y almacenamos nosotros si no que nos la pasa ya inicializada el bootloader. Utilizamos la estructura que nos otorga el bootloader, puesto que este ha hecho cambios en la estructura que nosotros debemos mantener.

```
④ Código 5.77: PTM.cpp — PTM::map(uint64_t, uint64_t)
47 void
48 PTM::map(uint64_t virt, uint64_t phys)
49 {
50     /* Parse uint64_t bits to a x86-64 virtual address struct */
51     address_t *virtaddr = (address_t *)&virt;
52
53     /** Get the page global entry from it's table */
54     page_global_dir_entry_t *PGD = &this->get_PGDT()[virtaddr->global];
55 }
```

```

56     /** Get the page upper dir table from the PGD or create & link a new one if needed */
57     page_upper_dir_entry_t *PUDT;
58     if (!PGD->present) {
59         PUDT = (page_upper_dir_entry_t *)kernel::allocator.request_page();
60         memset(PUDT, 0, uefi::page_size);
61         PGD->page_ppn = (uint64_t)PUDT >> 12;
62         PGD->present = true;
63         PGD->writeable = true;
64         PGD->user_access = true;
65     } else {
66         PUDT = (page_upper_dir_entry_t *)((uint64_t)PGD->page_ppn << 12);
67     }
68
69     /** Get the page upper entry from it's table */
70     page_upper_dir_entry_t *PUD = &PUDT[virtaddr->upper];
71
72     /** Get the page mid dir table from the PUD or create & link a new one if needed */
73     page_mid_dir_entry_t *PMDT;
74     if (!PUD->present) {
75         PMDT = (page_mid_dir_entry_t *)kernel::allocator.request_page();
76         memset(PMDT, 0, uefi::page_size);
77         PUD->page_ppn = (uint64_t)PMDT >> 12;
78         PUD->present = true;
79         PUD->writeable = true;
80         PUD->user_access = true;
81     } else {
82         PMDT = (page_mid_dir_entry_t *)((uint64_t)PUD->page_ppn << 12);
83     }
84
85     /** Get the page mid entry from it's table */
86     page_mid_dir_entry_t *PMD = &PMMDT[virtaddr->mid];
87
88     /** Get the page table dir table from the PMD or create & link a new one if needed */
89     page_table_entry_t *PTDT;
90     if (!PMD->present) {
91         PTDT = (page_table_entry_t *)kernel::allocator.request_page();
92         memset(PTDT, 0, uefi::page_size);
93         PMD->page_ppn = (uint64_t)PTDT >> 12;
94         PMD->present = true;
95         PMD->writeable = true;
96         PMD->user_access = true;
97     } else {
98         PTDT = (page_table_entry_t *)((uint64_t)PMD->page_ppn << 12);
99     }
100
101    /** Get the page table entry from it's table */
102    page_table_entry_t *PTD = &PTDT[virtaddr->table];
103
104    /** Fill it with the physical address */
105    PTD->page_ppn = (uint64_t)phys >> 12;
106    PTD->present = true;
107    PTD->writeable = true;
108    PTD->user_access = true;
109
110    /** Flush TLB Cache Entries https://www.felixcloutier.com/x86/invlpg */
111    asm("invlpg %0" : : "m"(virt));
112 }

```

En esta función añadimos una asociación entre dirección virtual y física. Como vemos, en la línea 51 transformamos una dirección en la estructura de datos mostrada en la figura 5.3. Acto seguido en la línea 54 obtenemos la primera entrada de la tabla. Esta entrada, en la línea 58 comprobamos si está inicializada, si lo está obtenemos la dirección de la siguiente tabla, si

no la creamos. Así seguimos hasta llegar a las hojas del árbol, donde finalmente almacenamos la dirección física (línea 105) y establecemos que la entrada está presente. Finalmente para acabar la asociación y evitar inconsistencias por la caché, limpiamos la “TLB”.

Translation Lookaside Buffer (TLB)

La TLB (Translation Lookaside Byffer) es una caché de memoria usada para almacenar traducciones recientes entre direcciones físicas y virtuales. La MMU, antes de realizar la búsqueda de la dirección física en el árbol, busca en la TLB si tiene una entrada almacenada, si no, realiza el proceso de traducción.

Podemos invalidar las entradas de la caché de una dirección virtual con la instrucción `invlpg virtmem`.

```
welcome to the alma kernel
$ getpage
0x0000000000002d000
$ getpage
0x0000000000002e000
$ set 20000 true
*(bool *)0x2D000 = true
$ get 2E000
*(bool *)0x2E000 -> false
$ set 20000 false
*(bool *)0x2D000 = false
$ map 2D000 2E000
2D000 -> 2E000
$ set 20000 true
*(bool *)0x2D000 = true
$ get 2E000
*(bool *)0x2E000 -> true
$
```

Figura 5.5: Prueba de Funcionamiento de la Memoria Virtual

En la figura 5.5 podemos observar una demostración de esto. Se obtienen dos direcciones de memoria separadas y, cuando se asocian entre ellas, las escrituras en una dirección se reflejan de la misma forma en la otra.

5.4.5. Memoria Dinámica

Esta sección trata sobre el desarrollo de las funciones `void *malloc(uint64_t)` y `void free(void *)` en *alma*. Tanto `malloc` como `free` hacen uso del gestor de páginas de memoria presentado en la sección 5.4.3. Este gestor, el de la memoria dinámica, ofrece mayor flexibilidad a la hora de reservar ciertos tamaños, mientras que en el gestor de páginas solo nos preocupaba la reserva y liberación de páginas, aquí podremos reservar cantidades variables de memoria.

En *alma*, no existe mucha diferencia entre el gestor de memoria dinámica y el gestor de reserva de páginas (quitando la reserva variable de memoria). Esto se debe a que *alma* no es un núcleo maduro con muchas características, no existen procesos ni el concepto de *scheduling*. Al ser un sistema tan simple no se aprecia la diferencia entre este apartado y el 5.4.3, el gestor de páginas lo podemos tener a nivel de máquina mientras que el gestor de memoria dinámica lo podemos tener a nivel de proceso (un heap para cada proceso).

Existen infinidad de modelos para implementar `malloc` y `free` en la literatura. Podemos encontrar implementaciones como `jemalloc`, `dmalloc`, `ptmalloc`, `mimalloc`, `tcmalloc`, etc. Diseñar un “malloc” desde cero no es el objetivo del proyecto, y mucho menos conseguir una versión eficiente, esto es algo que se escapa de complejidad.

Para poder trabajar con distintas implementaciones de `malloc` y `free` se define en *alma* una interfaz para el gestor de memoria:

© Código 5.78: allocator_i.h – class allocator_i

```
13 class allocator_i
14 {
15     public:
16     virtual void *malloc(uint64_t) = 0;
17     virtual void free(void *) = 0;
18 };
```

5.4.5.1. Trivial Allocator

La primera implementación que se va a presentar es una versión para poder trabajar con una implementación **mínimamente funcional** para empezar a escribir código usando `malloc` y `free`, pensando en posteriormente cambiar la implementación por una de verdad.

© Código 5.79: trivial_allocator.h – class trivial_allocator

```
25 class trivial_allocator : allocator_i
26 {
27     public:
28     trivial_allocator &operator=(const trivial_allocator &) = default;
29
30     void *malloc(uint64_t);
31     void free(void *);
32 };
```

```
④ Código 5.80: trivial_allocator.cpp — trivial_allocator::malloc(uint64_t)
20 void *
21 trivial_allocator::malloc(uint64_t size)
22 {
23     uint32_t pages = size / kernel::page_size + 1;
24
25     return kernel::allocator.request_cont_page(pages);
26 }
```

La función `malloc` de este trivial allocator delega el trabajo al gestor de páginas. El problema que tiene esto principalmente es que siempre reservamos de más: si reservamos 1 byte acabaremos reservando una página. Como se ha dicho esto es solo temporalmente hasta que se desarrolle otro allocator.

```
④ Código 5.81: trivial_allocator.cpp — trivial_allocator::free(void *)
20 void
21 trivial_allocator::free(void *addr)
22 {
23     return;
24 }
```

La función `free` no está implementada en este allocator de prueba. No es necesario para realizar código de prueba. Recordemos que este allocator solo se debe usar como esqueleto en el código hasta que se desarrolle otro.

5.4.5.2. Simple Allocator

Este modelo de allocator es una versión medianamente funcional para poder reservar y liberar memoria. El diseño y modelo es de “absurdponcho” y la implementación está muy basada en su versión.

▲ Correctitud del allocator

Este allocator no se ha probado en profundidad y puede contener casos concretos de fallo. Desarrollar un allocator es una tarea muy compleja que da para un trabajo por si solo, esta implementación es solo una versión mínimamente viable para tener algo funcional.

El allocator consiste en una lista enlazada de bloques de memoria con una cabecera que proporciona enlaces a los siguientes bloques de memoria y, el tamaño del bloque de memoria que sostenta.

Se utiliza como base el gestor de páginas para reservar páginas de memoria, luego estas páginas se usan como memoria base para construir el allocator. Es como si el gestor de páginas nos dejara reservar solo una página y nosotros particionáramos esa página en trozos más pequeños para ofertarlos como memoria libre y gestionarla a nuestra manera.

© Código 5.82: simple_allocator.h – class simple_allocator

```

14 class simple_allocator : allocator_i
15 {
16     public:
17     simple_allocator()
18         : heap_address(nullptr)
19         , heap_lenght(0){};
20
21     simple_allocator(uint64_t);
22
23     simple_allocator &operator=(const simple_allocator &) = default;
24
25     void *malloc(uint64_t);
26     void free(void *);
27
28     private:
29     void *heap_address;
30     uint64_t heap_lenght;
31
32     static const uint32_t ROUND_NUM = 0x10;
33
34     struct heap_header
35     {
36         uint64_t length;
37         heap_header *next;
38         heap_header *last;
39         bool is_free;
40
41         heap_header *split(uint64_t lenght);
42     };
43
44     void *heap_start;
45     void *heap_end;
46     heap_header *last_header;
47
48     void expand_heap(uint64_t);
49     void combine_forward(heap_header *);
50     void combine_backward(heap_header *);
51 };

```

Como podemos ver, de forma pública solo podemos interactuar con el objeto para reservar o liberar memoria. De forma interna tenemos los punteros al inicio y final de la lista enlazada y el tamaño y posición de donde se almacena el heap. También se incluye una constante, que es el tamaño de los bloques de memoria que vamos a reservar, es decir, todos los tamaños de reserva serán múltiplos de ROUND_NUM¹⁵.

© Código 5.83: simple_allocator.h – simple_allocator::simple_allocator(uint64_t)

```

17 simple_allocator::simple_allocator(uint64_t pages)
18 {
19     auto aux = kernel::allocator.request_page();
20     if (aux == nullptr) {
21         kernel::tty.println("fatal error");
22         return;
23     }
24
25     /* for each page, allocate and map them */
26     void *iter = aux;
27     for (uint64_t i = 0; i < pages - 1; i++) {

```

¹⁵De esta forma reducimos la fragmentación

```

28     /* phys addr != virt addr*/
29     kernel::translator.map((uint64_t)iter, (uint64_t)kernel::allocator.request_page());
30     iter = (uint8_t *)iter + kernel::page_size;
31 }
32
33 this->heap_address = aux;
34
35 /* calculate lenght of our heap */
36 uint64_t length = pages * kernel::page_size;
37
38 /* start and end pointer of the heap */
39 this->heap_start = heap_address;
40 this->heap_end = (void *)((uint8_t *)heap_start + length);
41
42 heap_header *start_header = (heap_header *)heap_address;
43 start_header->length = length - sizeof(heap_header);
44 start_header->next = nullptr;
45 start_header->last = nullptr;
46 start_header->is_free = true;
47
48 this->last_header = start_header;
49 }
```

Lo que hacemos en el constructor es inicializar el allocator. Se le pasa por parámetro el número de páginas a ocupar de forma inicial y se reservan tantas como se ha solicitado. Finalmente inicializamos la lista enlazada y creamos el primer (y por ahora único) bloque de memoria.

Código 5.84: simple_allocator.cpp – simple_allocator::malloc(uint64_t)

```

56 void *
57 simple_allocator::malloc(uint64_t size)
58 {
59     if (size == 0)
60         return nullptr;
61
62     /* round up size to ROUND_NUM (reduce fragmentation) */
63     if (size % simple_allocator::ROUND_NUM != 0) {
64         size -= (size % simple_allocator::ROUND_NUM);
65         size += simple_allocator::ROUND_NUM;
66     }
67
68     /* iterate through the heap */
69     heap_header *it = (heap_header *)heap_start;
70     while (true) {
71         /* if the segment is free */
72         if (it->is_free) {
73             /* we can split */
74             if (it->length > size) {
75                 it->split(size);
76                 it->is_free = false;
77                 return (void *)((uint8_t *)it + sizeof(heap_header));
78             }
79             /* exact match*/
80             if (it->length == size) {
81                 it->is_free = false;
82                 return (void *)((uint8_t *)it + sizeof(heap_header));
83             }
84         }
85
86         if (it->next == nullptr)
87             break;
88     }
89 }
```

```

88         it = it->next;
89     }
90     /* no memory for size, we need to expand the heap */
91     this->expand_heap(size);
92     return malloc(size);
93 }
94 }
```

La reserva de memoria con `malloc` itera sobre los nodos de la lista hasta encontrar uno libre. Si el nodo libre tiene más tamaño del que se solicita reservar, este se divide (línea 72) en dos bloques. Otro caso es que la memoria buscada sea exacta (línea 80). En caso de no encontrar ningún bloque disponible el heap se expande y se vuelve a iniciar el proceso de búsqueda de bloque libre (línea 92 y 93).

© Código 5.85: simple_allocator.cpp – simple_allocator::free(void *)

```

99 void
100 simple_allocator::free(void *addr)
101 {
102     /* -1 to get the header addr and not the memory block */
103     heap_header *header = (heap_header *)addr - 1;
104
105     header->is_free = true;
106     this->combine_forward(header);
107     this->combine_backward(header);
108 }
```

Lo que hacemos para liberar un bloque de memoria es acceder a su cabecera (línea 103). Como la dirección que se nos pasa es del inicio de la memoria, la cabecera del nodo estará justo detrás. Finalmente marcamos el nodo como vacío (línea 105) e intentamos juntarlo con los nodos anterior (línea 106) y posterior (107) de la lista para reducir la fragmentación.

© Código 5.86: simple_allocator.cpp – simple_allocator::expand_heap(uint64_t)

```

115 void
116 simple_allocator::expand_heap(uint64_t size)
117 {
118     if (size % simple_allocator::ROUND_NUM != 0) {
119         size -= (size % simple_allocator::ROUND_NUM);
120         size += simple_allocator::ROUND_NUM;
121     }
122
123     auto pages = size / kernel::page_size;
124
125     heap_header *header = (heap_header *)this->heap_end;
126     for (uint64_t i = 0; i < pages; i++) {
127         kernel::translator.map((uint64_t)this->heap_end,
128                               (uint64_t)kernel::allocator.request_page());
129         this->heap_end = (uint8_t *)this->heap_end + kernel::page_size;
130     }
131     header->is_free = true;
132     header->last = this->last_header;
133     this->last_header->next = header;
134     this->last_header = header;
135     header->next = nullptr;
136     header->length = size - sizeof(heap_header);
137     this->combine_backward(header);
138 }
```

Esta función nos permite expandir el heap. Se utiliza cuando no existe más memoria para otorgar o cuando se solicita un tamaño demasiado grande. Es una función muy similar a la del constructor, solo que añade un nodo más en la lista.

```
Q Código 5.87: simple_allocator.cpp - simple_allocator::heap_header::split(uint64_t)
146 simple_allocator::heap_header *
147 simple_allocator::heap_header::split(uint64_t size)
148 {
149     if (size < simple_allocator::ROUND_NUM)
150         return nullptr;
151     int64_t split_length = this->length - size - sizeof(heap_header);
152     if (split_length < simple_allocator::ROUND_NUM)
153         return nullptr;
154
155     heap_header *header = (heap_header *)((uint8_t *)this + size + sizeof(heap_header));
156     this->next->last = header;
157     header->next = this->next;
158     this->next = header;
159     header->last = this;
160     header->length = split_length;
161     header->is_free = true;
162     this->length = size;
163
164     if (this->last == this)
165         this->last = header;
166     return header;
167 }
```

Con `split` dividimos un bloque de memoria en dos, dejando el primero de un tamaño `size` pasado por parámetro.

```
Q Código 5.88: simple_allocator.cpp - simple_allocator::combine_forward(heap_header *)
181 void
182 simple_allocator::combine_forward(heap_header *hdr)
183 {
184     if (hdr->next == nullptr || !hdr->next->is_free)
185         return;
186     if (hdr->next == this->last_header)
187         last_header = hdr;
188     hdr->next = hdr->next->next;
189     hdr->length = hdr->length + hdr->next->length + sizeof(heap_header);
190 }
```

La función `combine_forward` une un bloque de memoria con el siguiente, en caso de que sea posible.

```
Q Código 5.89: simple_allocator.cpp - simple_allocator::combine_backward(heap_header *)
198 void
199 simple_allocator::combine_backward(heap_header *hdr)
200 {
201     if (hdr->last != nullptr && hdr->last->is_free)
202         this->combine_forward(hdr->last);
203 }
```

La función `combine_backward` une un bloque de memoria con el anterior, en caso de que sea posible.

5.4.6. Pantalla

La pantalla es el elemento que nos proporciona información sobre el estado de *alma*. Nos permite interactuar de forma visual con el proyecto ya sea mediante la pantalla de **qemu** (simulado) o mediante una pantalla real (hardware físico).

Para interactuar con la pantalla, el bootloader nos proporciona una estructura (framebuffer) con la información necesaria. El framebuffer, como ya se ha explicado anteriormente, contiene la dirección base de un buffer donde la memoria está mapeada a la memoria de la pantalla, es decir, lo que escribamos en esa memoria se visualizará por pantalla.

Puesto que estamos en el nivel más bajo de un sistema operativo, todo lo que queramos hacer en la pantalla tendrá que ser controlado manualmente. El dibujado de cada letra en píxeles, el desplazamiento cuando no caben más filas de texto, etc.

El código que gestiona la pantalla en alma está dividido en 4 grandes clases: **renderer_i**, **simple_renderer_i**, **fast_renderer_i** y **psf1**. **renderer_i** es una interfaz que define los métodos que han de tener las clases que van a renderizar el texto en la pantalla. **simple_renderer_i** y **fast_renderer_i** son interfaces que extienden a **renderer_i** e implementan los métodos relacionados con el renderizado de píxeles, cada una de una forma. **psf1** es una clase que implementa cualquier renderizador y se encarga de imprimir texto con una fuente PSF1 en pantalla.

5.4.6.1. Renderer

```
© Código 5.90: renderer_i.h – class renderer_i
18 class renderer_i
19 {
20     public:
21     renderer_i() = default;
22     virtual void draw(const char) = 0;
23     virtual void put(const char) = 0;
24     virtual void print(const char *, int64_t n = -1) = 0;
25     virtual void println(const char *) = 0;
26     virtual void fmt(const char *, ...) = 0;
27     virtual void newline() = 0;
28     virtual void clear() = 0;
29     virtual void scroll() = 0;
30     virtual void setColor(color_e) = 0;
31     virtual color_e getColor() = 0;
32     virtual void pushColor(color_e) = 0;
33     virtual void popColor() = 0;
34     virtual void pushCoords(uint32_t, uint32_t) = 0;
35     virtual void popCoords() = 0;
36     virtual uint32_t get_x() = 0;
37     virtual uint32_t get_y() = 0;
38     virtual void set_x(uint32_t) = 0;
39     virtual void set_y(uint32_t) = 0;
40     virtual uint32_t get_width() = 0;
41     virtual uint32_t get_height() = 0;
42 };
```

5.4.6.2. Renderer Simple

Este es el primer renderizador implementado. Se encarga de limpiar la pantalla, imprimir cadenas de texto (no caracteres), hacer scroll, implementar `fmt`, etc.

Este renderizador se usaba en las versiones iniciales de *alma*, sobretodo cuando solo se ejecutaba de forma simulada en `qemu`. Posteriormente se sustituyó por su versión rápida puesto que en máquinas reales era muy lento.

Se puede sustituir el renderizado por defecto de *alma* por este renderizador sin mayor problema, pero no es recomendado usarlo en hardware real.

```
Q Código 5.91: simple_renderer_i.h – class simple_renderer_i
19 class simple_renderer_i : public renderer_i
20 {
21     public:
22     simple_renderer_i(framebuffer, unsigned int, unsigned int, color_e);
23     simple_renderer_i() = default;
24     virtual void draw(const char) = 0;
25     void put(const char);
26     void print(const char *, int64_t n = -1);
27     void println(const char *);
28     void fmt(const char *, ...);
29     void newline();
30     void clear();
31     void scroll();
32     void setColor(color_e);
33     color_e getColor();
34     void pushColor(color_e);
35     void popColor();
36     uint32_t get_x();
37     uint32_t get_y();
38     void set_x(uint32_t);
39     void set_y(uint32_t);
40     void pushCoords(uint32_t, uint32_t);
41     void popCoords();
42     uint32_t get_width();
43     uint32_t get_height();
44     /** renderer glyph x size */
45     virtual unsigned int glyph_x() = 0;
46     /** renderer glyph y size */
47     virtual unsigned int glyph_y() = 0;
48
49 protected:
50     void draw_pixel(uint32_t, uint32_t);
51     /** Framebuffer to use */
52     framebuffer fb;
53     /** x PIXEL offset of next glyph */
54     unsigned int x_offset;
55     unsigned int alt_x_offset; // backup for push/pop
56     /** y PIXEL offset of next glyph */
57     unsigned int y_offset;
58     unsigned int alt_y_offset; // backup for push/pop
59     /** color of next glyph */
60     color_e color;
61     color_e alt_color;
62 };
```

© Código 5.92: simple_renderer_i.cpp – simple_renderer_i::simple_renderer_i(...)

```

12 simple_renderer_i::simple_renderer_i(framebuffer fb,
13                               unsigned int init_x,
14                               unsigned int init_y,
15                               color_e init_color)
16   : fb(fb)
17   , x_offset(init_x)
18   , y_offset(init_y)
19   , color(init_color)
20 {
21   this->clear();
22 }
```

El constructor del renderizado limpia la pantalla por defecto (todo a negro).

© Código 5.93: simple_renderer_i.cpp – simple_renderer_i::put(const char)

```

31 void
32 simple_renderer_i::put(const char character)
33 {
34     char aux[2] = { character, '\0' };
35     this->print(aux);
36 }
```

La función `put(const char)` crea un array de caracteres para pasarselo a `print(const char *)` y así poder imprimir un carácter de forma simple.

© Código 5.94: simple_renderer_i.cpp – simple_renderer_i::print(const char *, int64_t n)

```

45 void
46 simple_renderer_i::print(const char *str, int64_t n)
47 {
48     int i = 0;
49     while (str[i] && n != 0) {
50         switch (str[i]) {
51             case '\n':
52                 this->y_offset += this->glyph_y();
53                 this->x_offset = 0;
54                 break;
55             default:
56                 this->draw(str[i]);
57                 this->x_offset += this->glyph_x();
58                 if (this->x_offset >= this->fb.width)
59                     this->print("\n");
60             }
61         if ((this->y_offset + this->glyph_y()) > this->fb.height)
62             this->scroll();
63         n--;
64         i++;
65     }
66 }
```

Esta función es la encargada de imprimir una cadena de texto, gestiona caracteres especiales como (línea 51), saltos de línea (línea 58 y scrolls (línea 63). Mantiene actualizados las posiciones del siguiente carácter a imprimir (línea 52). El parámetro `n` indica cuantos caracteres queremos procesar.

```
Q Código 5.95: simple_renderer_i.cpp — simple_renderer_i::println(const char *)
```

```
77 void
78 simple_renderer_i::println(const char *str)
79 {
80     this->print(str);
81     this->put('\n');
82 }
```

La función `println` imprime una cadena y luego un salto de línea.

```
Q Código 5.96: simple_renderer_i.cpp — simple_renderer_i::clear()
```

```
111 void
112 simple_renderer_i::clear()
113 {
114     uint32_t jumps = fb.buffer_size / sizeof(uint64_t);
115     uint32_t rest = fb.buffer_size % sizeof(uint64_t);
116
117     /* Fast clear, use big integer movements */
118     unsigned int i;
119     for (i = 0; i < jumps; i++)
120         *((uint64_t *)fb.base + i) = static_cast<uint8_t>(screen::color_e::BLACK);
121
122     /* for sizes < 64 bytes */
123     for (unsigned int j = 0; j < rest; j++)
124         *((uint8_t *)((uint64_t *)fb.base + i) + j) = static_cast<uint8_t>(screen::color_e::BLACK);
125
126     this->x_offset = 0;
127     this->y_offset = 0;
128 }
```

`clear()` limpia la pantalla a negro (todo a 0). Como podemos ver primero se divide el tamaño del framebuffer en bloques de 64 bits para poder realizar asignaciones de 64 bits a 0. Esto hace que se limpie la pantalla mucho más rápido que si li hiciesemos byte a byte o cada 32 bits.

```
Q Código 5.97: simple_renderer_i.cpp — simple_renderer_i::scroll()
```

```
133 void
134 simple_renderer_i::scroll()
135 {
136     uint32_t mrg = fb.ppscl * (this->glyph_y() * 4);
137     uint32_t dif = fb.buffer_size - mrg;
138     uint8_t *src = (uint8_t *)fb.base + mrg;
139     uint8_t *dst = (uint8_t *)fb.base;
140
141     {
142         uint32_t jumps = dif / sizeof(uint64_t);
143         uint32_t rest = dif % sizeof(uint64_t);
144
145         unsigned int i;
146         for (i = 0; i < jumps; i++)
147             *((uint64_t *)dst + i) = *((uint64_t *)src + i);
148
149         for (unsigned int j = 0; j < rest; j++)
150             *((uint8_t *)((uint64_t *)dst + i) + j) = *((uint8_t *)((uint64_t *)src + i) + j);
151     }
152
153     {
154         uint8_t *clr = ((uint8_t *)fb.base + fb.buffer_size) - mrg;
```

```

155     uint32_t jumps = mrg / sizeof(uint64_t);
156     uint32_t rest = mrg % sizeof(uint64_t);
157
158     unsigned int i;
159     for (i = 0; i < jumps; i++)
160         *((uint64_t *)clr + i) = 0;
161
162     for (unsigned int j = 0; j < rest; j++)
163         *((uint8_t *)((uint64_t *)clr + i) + j) = 0;
164     }
165
166     if (this->y_offset >= this->glyph_y())
167         this->y_offset -= this->glyph_y();
168 }
169 }
```

La función `scroll()` es la encargada de desplazar cada píxel el número necesario de píxeles “hacia arriba” en la pantalla de tal forma que todo el texto mostrado se “suba” una columna hacia arriba.

▲ Lentitud de la función scroll

La función `scroll` ha de leer una gran cantidad de información y reescribirla en otra posición de memoria. Pese a que se realiza en bloques de 64 bits para que sea lo más rápido posible, puede ser muy lenta en hardware real.

Esto se debe a que en hardware real, leer un valor de la memoria del framebuffer es mucho más caro que escribir un valor. Esto provoca que las lecturas de `scroll` sean muy costosas y ejecute muy lento. Esto no tiene por qué pasar en ejecuciones simuladas ya que dicho framebuffer es almacenado en la RAM de nuestro ordenador y dibujado por nuestro sistema operativo en pantalla.

`fast_renderer` implementa ciertos mecanismos para evitar estas lecturas lentas, haciendo que sea mucho más rápido el `scroll` en dicha versión.

© Código 5.98: simple_renderer_i.cpp – simple_renderer_i::fmt(const char *)

```

195 void
196 simple_renderer_i::fmt(const char *fmtstr, ...)
197 {
198     va_list args;
199     va_start(args, fmtstr);
200     char buffer[256];
201     uint64_t i = 0;
202     uint64_t literal = 0;
203     bool specialchar = false;
204     while (fmtstr[i] != '\0') {
205         if (fmtstr[i] == '%') {
206             specialchar = true;
207             this->print(&fmtstr[literal], i - literal);
208             literal = i + 2;
209         } else if (specialchar) {
210             specialchar = false;
211             switch (fmtstr[i]) {
212                 case 'i': {
213                     str(va_arg(args, int), buffer);
```

```

214         this->print(buffer);
215         break;
216     }
217     case 's': {
218         this->print(va_arg(args, const char *));
219         break;
220     }
221     case 'p': {
222         hstr(va_arg(args, uint64_t), buffer);
223         this->print("0x");
224         this->print(buffer);
225         break;
226     }
227     case 'd': {
228         str(va_arg(args, double), buffer);
229         this->print(buffer);
230         break;
231     }
232     case 'c': {
233         this->put((char)va_arg(args, int));
234         break;
235     }
236 }
237 }
238 i++;
239 }
240 this->print(&fmtstr[literal]);
241 this->newline();
242 }
```

La función `fmt` es una función auxiliar que implementa las funcionalidades más avanzadas de `printf` como imprimir números, floats, strings, etc. Esto se hace con identificadores especiales en el texto como `%s`, `%i`, etc. Como se puede ver la función utiliza argumentos variádicos para ser usada de la siguiente manera: `fmt("Número: %i %d", num, num2)`, esto nos permite pasarle a la función como argumento las variables que vamos a imprimir, sin importar su número.

Actualmente se soporta imprimir enteros, cadenas de caracteres, punteros, doubles y caracteres.

`fmt`, a pesar de implementar funcionalidades similares a lo que estamos acostumbrados a encontrar en `printf`, se ha decidido implementar como función ajena a `printf` debido a su complejidad. Es importante recordar que *alma* es un proyecto independiente y no tiene por que tener similitudes con otros núcleos o librerías.

Código 5.99: simple_renderer_i.cpp — `simple_renderer_i::pushColor(color_e)`

```

251 void
252 simple_renderer_i::pushColor(color_e color)
253 {
254     this->alt_color = this->color;
255     this->color = color;
256 }
```

La función `pushColor()` actualiza el color en el que se dibujan los píxeles por el introducido en el parámetro, guardando el anterior para poder revertirlo mediante `popColor()`.

© Código 5.100: simple_renderer_i.cpp – simple_renderer_i::popColor()

```
258 void
259 simple_renderer_i::popColor()
260 {
261     this->color = this->alt_color;
262 }
```

Esta función restaura el color anteriormente actualizado por pushColor.

© Código 5.101: simple_renderer_i.cpp – simple_renderer_i::draw_pixel(uint32_t, uint32_t)

```
264 void
265 simple_renderer_i::draw_pixel(uint32_t x, uint32_t y)
266 {
267     *(static_cast<unsigned int *>(this->fb.base + x + (y * this->fb.ppscl))) =
268         static_cast<unsigned int>(this->color);
269 }
```

`draw_pixel` dibuja un pixel de un determinado color en una posición `x` e `y` de la pantalla. Se hace uso de aritmética de punteros para obtener el puntero al pixel a actualizar.

El puntero base del framebuffer es un array lineal de píxeles por lo que el acceso a una determinada fila ha de ser transformado a su versión de acceso lineal.

Es importante recalcar que se utiliza `fb.ppscl` como tamaño de fila y no `fb.width`, esto es porque `ppscl` puede mostrar bits que no pertenecen al tamaño de la pantalla y que se quedan sin usar, a modo de padding. Es decir, el framebuffer puede tener un ancho mayor que el real de la pantalla, esos píxeles no se mostrarán en pantalla pero hemos de tenerlos en cuenta a la hora de acceder al buffer.

© Código 5.102: simple_renderer_i.cpp – simple_renderer_i::pushCoords(uint32_t, uint32_t)

```
271 void
272 simple_renderer_i::pushCoords(uint32_t x, uint32_t y)
273 {
274     this->alt_x_offset = this->x_offset;
275     this->alt_y_offset = this->y_offset;
276     this->x_offset = x;
277     this->y_offset = y;
278 }
```

Esta función actualiza las coordenadas donde se imprime el siguiente carácter o píxel, guardando las anteriores coordenadas para poder revertirlas mediante `popCoords()`.

© Código 5.103: simple_renderer_i.cpp – simple_renderer_i::popCoords()

```
280 void
281 simple_renderer_i::popCoords()
282 {
283     this->x_offset = this->alt_x_offset;
284     this->y_offset = this->alt_y_offset;
285 }
```

Esta función restaura las coordenadas anteriormente actualizadas por `pushCoords`.

5.4.6.3. Renderer Rápido

El renderizado rápido es una derivación del renderizado simple que implementa ciertos mecanismos para evitar lecturas de píxeles en pantalla (comunes en la función `scroll`). Como ya se ha mencionado en la función `scroll` del renderizado simple, la lectura de datos en pantalla es muy lenta en hardware real, por lo que es necesario implementar mecanismos para evitarla, si no el sistema no es usable.

Este renderizado almacena una copia sincronizada de los contenidos del framebuffer en RAM para evitar copias. Los detalles de la implementación se verá en cada función listadas a continuación.

```
④ Código 5.104: fast_renderer_i.h — class fast_renderer_i
19 class fast_renderer_i : public renderer_i
20 {
21     public:
22         fast_renderer_i(framebuffer, unsigned int, unsigned int, color_e);
23         fast_renderer_i() = default;
24         virtual void draw(const char) = 0;
25         void put(const char);
26         void print(const char *, int64_t n = -1);
27         void println(const char *);
28         void fmt(const char *, ...);
29         void newline();
30         void clear();
31         void scroll();
32         void setColor(color_e);
33         color_e getColor();
34         void pushColor(color_e);
35         void popColor();
36         uint32_t get_x();
37         uint32_t get_y();
38         void set_x(uint32_t);
39         void set_y(uint32_t);
40         void pushCoords(uint32_t, uint32_t);
41         void popCoords();
42         uint32_t get_width();
43         uint32_t get_height();
44     /** renderer glyph x size */
45     virtual unsigned int glyph_x() = 0;
46     /** renderer glyph y size */
47     virtual unsigned int glyph_y() = 0;
48
49     protected:
50         void draw_pixel(uint32_t, uint32_t);
51         uint32_t *get_pixel(uint32_t, uint32_t);
52         void update_video();
53         void update_line();
54         framebuffer video_memory;
55     /** Cache (double buffer) */
56         cache_framebuffer video_cache;
57         unsigned int x_offset;
58         unsigned int alt_x_offset; // backup for push/pop
59         unsigned int y_offset;
60         unsigned int alt_y_offset; // backup for push/pop
61         color_e color;
62         color_e alt_color;
63 };
```

▲ Funciones mostradas

En este apartado, debido a la similitud con `simple_renderer` en la mayoría de funciones, solo se van a mostrar las funciones que difieren de `simple_renderer`.

Para ver el resto de funciones puedes acudir al repositorio (`kernel/screen/`) o ver la versión idéntica de `simple_renderer` en la sección 5.4.6.2.

© Código 5.105: `fast_renderer_i.cpp` – `fast_renderer_i::fast_renderer_i(...)`

```

22 fast_renderer_i::fast_renderer_i(framebuffer video_memory,
23                                     unsigned int init_x,
24                                     unsigned int init_y,
25                                     color_e init_color)
26   : video_memory(video_memory)
27   , x_offset(init_x)
28   , y_offset(init_y)
29   , color(init_color)
30 {
31   /* Create the cache buffer */
32   this->video_cache = video_memory;
33   this->video_cache.base = (uint32_t *)kernel::allocator.request_cont_page(
34     this->video_memory.buffer_size / kernel::page_size + 1);
35   this->video_cache.actual = this->video_cache.base;
36   this->video_cache.limit =
37     (unsigned int *)((uint8_t *)this->video_cache.base + this->video_memory.buffer_size);
38
39   /* Clear cache & video memory to 0 */
40   uint64_t steps = this->video_memory.buffer_size / sizeof(uint64_t);
41   uint32_t remainder = this->video_memory.buffer_size % sizeof(uint64_t);
42
43   for (uint64_t i = 0; i < steps; i++)
44     ((uint64_t *)this->video_cache.base)[i] = 0;
45
46   for (uint32_t i = 0; i < remainder; i++)
47     ((uint8_t *)this->video_cache.base)[i] = 0;
48
49   this->update_video();
50 }
```

El constructor de `fast_renderer` crea un buffer que actua como copia de la memoria de video que tenemos. Tanto el buffer como la memoria de video se inicializan a 0 (negro).

© Código 5.106: `fast_renderer_i.cpp` – `fast_renderer_i::clear()`

```

139 void
140 fast_renderer_i::clear()
141 {
142   uint32_t jumps = this->video_memory.buffer_size / sizeof(uint64_t);
143   uint32_t rest = this->video_memory.buffer_size % sizeof(uint64_t);
144
145   /* Fast clear, use big integer movements */
146   unsigned int i;
147   uint64_t *video_64 = (uint64_t *)this->video_memory.base;
148   uint64_t *cache_64 = (uint64_t *)this->video_cache.base;
149   for (i = 0; i < jumps; i++) {
150     *video_64++ = 0;
151     *cache_64++ = 0;
152   }
153 }
```

```

154     uint8_t *video_8 = (uint8_t *)video_64;
155     uint8_t *cache_8 = (uint8_t *)cache_64;
156     /* for sizes < 64 bytes */
157     for (unsigned int j = 0; j < rest; j++) {
158         *video_64++ = 0;
159         *cache_64++ = 0;
160     }
161
162     this->video_cache.actual = this->video_cache.base;
163     this->x_offset = 0;
164     this->y_offset = 0;
165 }
```

La función `clean` limpia tanto la memoria de video como la cache, estableciendo todos los valores a 0 (negro).

© Código 5.107: fast_renderer_i.cpp – fast_renderer_i::scroll()

```

170 void
171 fast_renderer_i::scroll()
172 {
173     uint64_t *ptr_64 = (uint64_t *)this->video_cache.actual;
174
175     this->video_cache.actual += (this->video_cache.ppscl * this->glyph_y());
176     if (this->video_cache.actual >= this->video_cache.limit)
177         this->video_cache.actual = this->video_cache.base;
178
179     uint32_t size = this->video_cache.ppscl * this->glyph_y() * sizeof(uint32_t);
180
181     // It shouldn't happen as it's aligned but safety checks as it can lead to out of bounds ↪
182     // writes
183     if ((uint32_t *)((uint8_t *)ptr_64 + size) > this->video_cache.limit)
184         size = ((uint8_t *)this->video_cache.limit - (uint8_t *)ptr_64);
185
186     uint32_t jumps = size / sizeof(uint64_t);
187     uint32_t rest = size % sizeof(uint64_t);
188
189     for (int i = 0; i < jumps; i++)
190         *ptr_64++ = 0;
191
192     uint8_t *ptr_8 = (uint8_t *)ptr_64;
193
194     for (int i = 0; i < rest; i++)
195         *ptr_8++ = 0;
196
197     if (this->y_offset >= this->glyph_y())
198         this->y_offset -= this->glyph_y();
199
200     this->update_video();
201 }
```

`scroll` copia los valores de la cache en la memoria de video empezando por la segunda fila. Para evitar mover los valores en scroll se implementa una mecánica de buffer circular donde se mueve un puntero que marca el inicio de los datos, evitando así copias innecesarias.

© Código 5.108: fast_renderer_i.cpp – fast_renderer_i::update_video()

```

282 void
283 fast_renderer_i::update_video()
284 {
285     uint64_t *video_64 = (uint64_t *)this->video_memory.base;
```

```

286     uint8_t *video_8;
287     {
288         uint64_t size = (uint8_t *)this->video_cache.limit - (uint8_t *)this->video_cache.actual;
289         uint64_t steps = size / sizeof(uint64_t);
290         uint64_t remainder = size % sizeof(uint64_t);
291
292         uint64_t *cache_64 = (uint64_t *)this->video_cache.actual;
293
294         for (uint64_t i = 0; i < steps; i++)
295             *video_64++ = *cache_64++;
296
297         uint8_t *cache_8 = (uint8_t *)cache_64;
298         video_8 = (uint8_t *)video_64;
299
300         for (uint64_t i = 0; i < remainder; i++)
301             *video_8++ = *cache_8++;
302     }
303
304     {
305         uint64_t size = (uint8_t *)this->video_cache.actual - (uint8_t *)this->video_cache.base;
306         uint64_t steps = size / sizeof(uint64_t);
307         uint64_t remainder = size % sizeof(uint64_t);
308
309         uint64_t *cache_64 = (uint64_t *)this->video_cache.base;
310
311         for (uint64_t i = 0; i < steps; i++)
312             *video_64++ = *cache_64++;
313
314         uint8_t *cache_8 = (uint8_t *)cache_64;
315         video_8 = (uint8_t *)video_64;
316         for (uint64_t i = 0; i < remainder; i++)
317             *video_8++ = *cache_8++;
318     }
319 }
```

La función `update_video` copia los valores de la cache a la memoria de video (se sincronizan). El código es notablemente más complejo que una simple copia debido a que la cache es un buffer circular, por lo que hay que realizar la copia en dos tramos.

 Código 5.109: `fast_renderer_i.cpp` – `fast_renderer_i::draw_pixel(uint32_t, uint32_t)`

```

327 void
328 fast_renderer_i::draw_pixel(uint32_t x, uint32_t y)
329 {
330     *(static_cast<unsigned int *>(this->video_memory.base + x + (y * this->video_memory.ppscl))) =
331         static_cast<unsigned int>(this->color);
332
333     *(this->get_pixel(x, y)) = static_cast<unsigned int>(this->color);
334 }
```

La función de dibujado de un pixel se caracteriza por realizar dos escrituras en vez de una. Una de ellas se hace a la memoria de video para mostrar el pixel de inmediato y otra a la cache, para evitar leer el pixel escrito de la memoria de video cuando se haga `scroll`.

 Código 5.110: `fast_renderer_i.cpp` – `fast_renderer_i::get_pixel(uint32_t, uint32_t)`

```

352 uint32_t *
353 fast_renderer_i::get_pixel(uint32_t x, uint32_t y)
354 {
355     uint32_t *ptr = this->video_cache.actual + x + (y * this->video_cache.ppscl);
```

```

356     if (ptr >= this->video_cache.limit)
357         ptr = this->video_cache.base + (ptr - this->video_cache.limit);
358
359     return ptr;
360 }

```

Obtener un pixel requiere acceder a la cache y no a la memoria de video para poder realizarse de forma rápida. Como la cache es un buffer circular es necesario gestionar su acceso en forma de buffer circular.

5.4.6.4. Fuente PSF1

Las fuentes son las clases que implementan la interfaz del renderizado, son las responsables de implementar la función para dibujar un carácter en pantalla, ya que son las que conocen como se dibuja un carácter.

En primer lugar, definimos la estructura de la fuente PSF1, ya explicada en 4.4.4.3:

© Código 5.111: psf1.h – struct psf1_header

```

17 struct psf1_header
18 {
19     /** 0x36, 0x04 */
20     unsigned char magic[2];
21     /** number of glyphs mode */
22     unsigned char mode;
23     /** size of each glyph in the glyph buffer */
24     unsigned char charsize;
25 };

```

© Código 5.112: psf1.h – struct psf1

```

30 struct psf1
31 {
32     /**
33      * PSF1 Font Header
34      */
35     psf1_header header;
36
37     /** glyph buffer */
38     void *buffer;
39
40     /** Glyph x size in pixels */
41     static const unsigned int glyph_x = 8;
42     /** Glyph y size in pixels */
43     static const unsigned int glyph_y = 16;
44 };

```

Colisión de nombres

No hay colisión de nombres en `psf1` debido a que la estructura que conforma el formato de la fuente y la clase que renderiza la fuente están en espacios de nombres distintos.

© Código 5.113: psf1.h – class psf1

```

50 template<typename T>
51 class psf1 : public T
52 {
53     public:
54         psf1(screen::framebuffer fb,
55             screen::fonts::specification::psf1 font,
56             unsigned int x_offset = 0,
57             unsigned int y_offset = 0,
58             color_e color = color_e::WHITE)
59             : T(fb, x_offset, y_offset, color)
60             , font(font)
61     {
62         this->video_cache.width == this->video_cache.width % 8;
63         this->video_cache.height == this->video_cache.height % 16;
64         this->video_cache.buffer_size =
65             this->video_cache.width * this->video_cache.height * sizeof(uint32_t);
66         this->video_cache.limit =
67             (uint32_t *)((uint8_t *)this->video_cache.base + this->video_cache.buffer_size);
68     }
69     psf1() = default;
70     psf1 &operator= (psf1 &&rhs)
71     {
72         T::operator= (rhs);
73         this->font = rhs.font;
74         return *this;
75     }
76
77     void draw(const char character)
78     {
79         char *chr = static_cast<char*>(this->font.buffer) +
80             (static_cast<unsigned char>(character) * this->font.header.charsize);
81
82         for (unsigned long y = this->y_offset; y < this->y_offset + this->glyph_y(); y++) {
83             for (unsigned long x = this->x_offset; x < this->x_offset + this->glyph_x(); x++) {
84                 if ((*chr & (0b10000000 >> (x - this->x_offset))))
85                     this->draw_pixel(x, y);
86             }
87             chr++;
88         }
89     }
90
91     unsigned int glyph_x()
92     {
93         return fonts::specification::psf1::glyph_x;
94     }
95     unsigned int glyph_y()
96     {
97         return fonts::specification::psf1::glyph_y;
98     }
99
100    private:
101        fonts::specification::psf1 font;
102 };

```

Esta clase hereda de un renderizador cualquiera, especificado mediante el uso de templates¹⁶, por lo que la misma implementación sirve tanto para `simple_renderer` como para `fast_renderer` como para un hipotético nuevo renderizador.

¹⁶Al usar templates tenemos que poner las definiciones de toda la clase en un archivo de cabecera. Existen otros métodos pero tienen inconvenientes.

5.4.7. Segmentación

La segmentación es un mecanismo que implementa la arquitectura x86 para poder direccionar un mayor rango de memoria. Este mecanismo posteriormente derivó en lo que se conoce como “protected mode”, ya que incorporaba soporte para memoria virtual y protección de memoria. La segmentación hace uso de registros especiales como “base” en los accesos de memoria y para comprobar dichos accesos.

En x86-64 se ha desactivado de forma general (no completamente) el soporte para segmentación, siendo este sustituido por la paginación. De igual manera necesitaremos trabajar con la segmentación de forma breve para crear un espacio lineal de memoria. No usaremos mecanismos complejos de segmentación pero si que crearemos un mapa simple, puesto que es necesario para las interrupciones.

Como no tenemos segmentación como tal en x86-64, en esta sección solo se mostrará lo necesario para que *alma* pueda trabajar con las interrupciones: crear un mapa lineal de memoria.

La estructura de datos encargada de almacenar la información sobre los segmentos de memoria es la Global Descriptor Table (GDT). La GDT es una array (tabla) que contiene entradas para cada segmento de memoria. El formato de las entradas es el siguiente:

```
④ Código 5.114: gdt.h – struct gdt_entry
27 struct gdt_entry
28 {
29     /** The lower 16 bits of the limit */
30     uint16_t limit_low;
31     /** The lower 16 bits of the base */
32     uint16_t base_low;
33     /** The next 8 bits of the base */
34     uint8_t base_middle;
35     uint8_t access;
36     uint8_t granularity;
37     /** The last 8 bits of the base */
38     uint8_t base_high;
39 } __attribute__((packed));
```

La estructura que se le debe pasar a la CPU es una estructura con la dirección física de la tabla y su tamaño -1 (última dirección válida). El formato de la estructura es el siguiente:

```
④ Código 5.115: gdt.h – struct gdt_ptr
56 struct gdt_ptr
57 {
58     /**
59      * sizeof(gdt) - 1
60      * @warning Yes, it's -1 always (last valid address)
61      */
62     uint16_t size;
63     /** &gdt */
64     uint64_t offset;
65 } __attribute__((packed));
```

Ahora hay que crear la GDT como tal. La GDT como ya hemos dicho es una array de `gdt_entry`, cada uno representando un segmento de memoria. La GDT usada en *alma* es la siguiente:

Q Código 5.116: `gdt.h` – `const gdt_entry table[]`

```

83 __attribute__((aligned(uefi::page_size)))
84 const gdt_entry table[] = {
85     /** null descriptor */
86     { 0, 0, 0, 0, 0, 0 },
87     /** Kernel code */
88     { 0, 0, 0, 0x9a, 0xa0, 0 },
89     /** Kernel data */
90     { 0, 0, 0, 0x92, 0xa0, 0 },
91     /** User null descriptor */
92     { 0, 0, 0, 0, 0, 0 },
93     /** User code */
94     { 0, 0, 0, 0x9a, 0xa0, 0 },
95     /** User data */
96     { 0, 0, 0, 0x92, 0xa0, 0 },
97 };

```

El primer segmento que creamos, por especificación, debe ser un segmento nulo (todo a 0). Acto seguido creamos el código y datos del kernel y posteriormente hacemos lo mismo con un supuesto segmento para datos de usuario. Estos segmentos dan igual ya que, como ya hemos comentado, en x86-64 la segmentación no se usa.

Q Código 5.117: `gdt.asm` – `load_gdt(gdt_ptr *)`

```

8 [bits 64]
9
10 ;
11 ; void load_gdt(gdt_ptr*)
12 ;
13 load_gdt:
14     lgdt [rdi] ; rdi contains first parameter passed to load_gdt
15     mov ax, 0x10 ; gdt kernel data segment
16     mov ds, ax ; set data segments
17     mov es, ax
18     mov fs, ax
19     mov gs, ax
20     mov ss, ax
21     pop rdi ; get return addr (from stack) to rdi register
22     mov rax, 0x08 ; gdt kernel code segment
23     push rax
24     push rdi
25     retfq ; far 64bit return (pops address and code segment)
26
27 ; make it accessible for other code
28 GLOBAL load_gdt

```

Este código en ensamblador es el encargado de cargar la GDT en la CPU con la instrucción `lgdt`. La función está en ensamblador y en C se incluye en la cabecera de `gdt.h` como `extern` para poder llamar a la función desde *C*. En la función, a parte de insertar la GDT se actualizan los registros de segmento tal como se explica en [36].

5.4.8. Bus I/O

Esta sección muestra las funciones implementadas en *alma* para comunicarse con los periféricos mediante los puertos I/O del procesador.

En x86-64 tenemos dos formas de comunicarnos con los puertos I/O: mediante “Memory Mapped I/O” (MMIO) o mediante “Port I/O” (PIO). En esta sección se verá el mecanismo de “Port I/O” mientras que en la sección 5.4.13 se verá un ejemplo de acceso mediante MMIO.

A estos puertos se accede mediante un rango de direcciones independiente de la máquina denominado “I/O Address Space”. A este rango de direcciones se accede mediante instrucciones especiales de I/O como `outb`, `inb`, etc.

El rango de direcciones que tenemos para los puertos I/O es $[0x0000 - 0xFFFFh]$, lo que nos permite tener 65536 puertos. Este rango de direcciones se puede dividir en dos rangos o tipos de acceso: fijos y variables. Los accesos fijos son puertos que no pueden ser cambiados, que vienen prefijados. Los accesos variables pueden ser reorganizados y también se pueden desactivar.

Los accesos fijos suelen ser puertos que están conectados directamente al una pieza de hardware específica, por ejemplo 0x50-0x52 está conectado al Programmable Interval Timer (PIT), el puerto 0x60 está conectado al controlador de teclado, etc.

Los accesos variables suelen ser para que la BIOS ajuste estos puertos para poder comunicarnos con los dispositivos PCI. Posteriormente, en la sección 5.4.13 veremos como con PCI Express podemos realizar lo mismo que veremos en esta sección pero con simples accesos a memoria.

Para trabajar con los puertos I/O mediante PIO (sin MMIO) definimos en una cabecera una serie de funciones auxiliares que nos permitan escribir y leer datos de los puertos:

```
Código 5.118: bus.h
16 enum class port
17 {
18     PS2 = 0x60,
19 };
20
21 const int PIC1_COMMAND = 0x20;
22 const int PIC1_DATA = 0x21;
23 const int PIC2_COMMAND = 0xA0;
24 const int PIC2_DATA = 0xA1;
25 const int ICW1_INIT = 0x10;
26 const int ICW1_ICW4 = 0x01;
27 const int ICW3_PIC1 = 0b00000100;
28 const int ICW3_PIC2 = 0b00000010;
29 const int ICW4_8086 = 0x01;
30
31 void outb(uint16_t, uint8_t);
32 void outb(io::port, uint8_t);
33 uint8_t inb(uint16_t);
34 uint8_t inb(io::port);
35 void io_wait();
```

Q Código 5.119: bus.cpp – outb(uint16_t, uint8_t)

```
14 void
15 outb(uint16_t port, uint8_t value)
16 {
17     /* value -> %0 */
18     /* port -> %1 */
19     asm volatile("outb %0, %1" : : "a"(value), "Nd"(port));
20 }
```

Esta función nos permite enviar un byte `value` a un puerto `port`.

Q Código 5.120: bus.cpp – outb(io::port, uint8_t)

```
25 void
26 outb(io::port port, uint8_t value)
27 {
28     io::outb(static_cast<uint16_t>(port), value);
29 }
```

Esta función es equivalente a la anterior, está sobrecargada para recibir un `enum` como puerto.

Q Código 5.121: bus.cpp – inb(uint16_t)

```
34 uint8_t
35 inb(uint16_t port)
36 {
37     /* return value -> aux */
38     /* port -> %1 */
39     uint8_t aux;
40     asm volatile("inb %1, %0" : "=a"(aux) : "Nd"(port));
41     return aux;
42 }
```

Esta función nos permite recibir un byte de un puerto `port`.

Q Código 5.122: bus.cpp – inb(io::port)

```
47 uint8_t
48 inb(io::port port)
49 {
50     return io::inb(static_cast<uint16_t>(port));
51 }
```

Esta función es equivalente a la anterior, está sobrecargada para recibir un `enum` como puerto.

Q Código 5.123: bus.cpp – io_wait()

```
56 void
57 io_wait()
58 {
59     /* Send a empty value to the device to "wait" */
60     asm volatile("outb %%al, $0x80" : : "a"(0));
61 }
```

Esta función envía un dato vacío para esperar a que el anterior envío se procese.

5.4.9. Interrupciones

Las interrupciones son eventos que indican que existe una condición específica en el sistema, el procesador o en el programa ejecutándose que requiere la atención del procesador [11]. Esto resulta en una transmisión forzosa de la CPU del programa ejecutándose a una rutina especial para gestionar dicho evento.

Las interrupciones ocurren en respuesta a señales del hardware. El hardware del sistema utiliza las interrupciones para gestionar eventos externos, como puede ser la recepción de un paquete ethernet, el click del ratón, etc. El software también puede generar interrupciones ejecutando la instrucción `int N`.

Cada interrupción tiene asignado un número [0 – 255] denominado “vector”. El vector es el número que identifica la interrupción y el que usaremos posteriormente para asignarla con una función para gestionarla. Aquí podemos ver una tabla con los vectores de las interrupciones:

Vector	Código	Descripción
0	DE	Divide Error
1	DB	Debug Exception
2	-	NMI Interrupt
3	BP	Breakpoint
4	OF	Overflow
5	BR	BOUND Range Exceeded
6	UD	Invalid Opcoded
7	NM	Device Not Available
8	DF	Double Fault
9	-	Coprocessor Segment Overrun
10	TS	Invalid TSS
11	NP	Segment Not present
12	SS	Stack-Segment Fault
13	GP	General Protection
14	PF	Page Fault
15	-	Intel Reserved
16	MF	x87 FPU Floating Point Error
17	AC	Alignment Check
18	MC	Machine Check
19	XM	SIMD Floating Point Exception
20	VE	Virtualization Exception
21	CP	Control Protection Exception
22-31	-	Intel Reserved
32-255	-	User Defined

Tabla 5.3: Lista de Interrupciones [11]

El sistema de interrupciones de x86 necesita de tres elementos clave [37] para funcionar:

- El Programable Interrupt Controller (PIC) ha de estar configurado para recibir solicitudes de interrupción (IRQs) y enviarlas a la CPU
- La CPU ha de estar configurada para recibir IRQs del PIC e invocar el “interrupt handler” correspondiente en base a la “Interrupt Descriptor Table” (IDT)
- El núcleo del sistema operativo debe proveer las “Interrupt Service Routines” (ISR) para gestionar las interrupciones.

Empezaremos explicando el segundo punto mencionado. La “Interrupt Descriptor Table” (IDT) es una tabla que contiene una estructura que nos une un vector de los vistos anteriormente con una dirección. La IDT debe estar apuntada por una estructura especial, al igual que la GDT (son similares), esta estructura es la siguiente:

```
© Código 5.124: IDT.h – struct idt_ptr
38 struct idt_ptr
39 {
40     uint16_t lenght;
41     uint64_t ptr;
42     idt_ptr();
43     void set_ptr(uint64_t);
44     void add_handle(interrupts::vector_e code, void (*handler)(frame *));
45     static void remap_pic(uint8_t, uint8_t);
46 } __attribute__((packed));
```

Cada entrada de la tabla viene definida por la estructura `idt_entry`:

```
© Código 5.125: IDT.h – struct idt_entry
51 struct idt_entry
52 {
53     uint16_t offset_low;
54     uint16_t vector;
55     uint8_t ist;
56     uint8_t type_attr;
57     uint16_t offset_middle;
58     uint32_t offset_high;
59     uint32_t reserved;
60     uint64_t get_offset();
61     void set_offset(uint64_t);
62 } __attribute__((packed));
```

Como tendremos que transformar direcciones de `uint64_t` al formato de `idt_entry`, definimos una estructura auxiliar para poder hacer `addr_t *aux = (addr_t *)&addr;`

```
© Código 5.126: IDT.h – struct addr_t
66 struct addr_t
67 {
68     uint16_t offset_low;
69     uint16_t offset_middle;
70     uint32_t offset_high;
71 } __attribute__((packed));
```

```
④ Código 5.127: IDT.cpp - idt_ptr::remap_pic(uint8_t, uint8_t)

39 void
40 idt_ptr::remap_pic(uint8_t ICW2_PIC1, uint8_t ICW2_PIC2)
41 {
42     /* https://wiki.osdev.org/8259_PIC (Initialisation) */
43     /* Store the PIC masks */
44     uint8_t mask_PIC1, mask_PIC2;
45     mask_PIC1 = io::inb(io::PIC1_DATA);
46     io::io_wait();
47     mask_PIC2 = io::inb(io::PIC2_DATA);
48     io::io_wait();
49
50     /* (ICW1) Init master/slave PIC chip */
51     io::outb(io::PIC1_COMMAND, io::ICW1_INIT | io::ICW1_ICW4);
52     io::io_wait();
53     io::outb(io::PIC2_COMMAND, io::ICW1_INIT | io::ICW1_ICW4);
54     io::io_wait();
55
56     /* (ICW2) Set vector offset */
57     io::outb(io::PIC1_DATA, ICW2_PIC1);
58     io::io_wait();
59     io::outb(io::PIC2_DATA, ICW2_PIC2);
60     io::io_wait();
61
62     /* (ICW3) Tell how master/slave are wired */
63     io::outb(io::PIC1_DATA, io::ICW3_PIC1);
64     io::io_wait();
65     io::outb(io::PIC2_DATA, io::ICW3_PIC2);
66     io::io_wait();
67
68     /* (ICW4) Tell PIC we are on 8086 */
69     io::outb(io::PIC1_DATA, io::ICW4_8086);
70     io::io_wait();
71     io::outb(io::PIC2_DATA, io::ICW4_8086);
72     io::io_wait();
73
74     /* Restore saved masks */
75     io::outb(io::PIC1_DATA, mask_PIC1);
76     io::io_wait();
77     io::outb(io::PIC2_DATA, mask_PIC2);
78     io::io_wait();
79 }
```

Esta función inicializa y reasigna unos vectores por recomendación de [wiki.osdev.org](https://wiki.osdev.org/8259_PIC) ya que supuestamente dan problemas en modo protegido. Esta función se llama al inicializar el módulo de interrupciones en el startup.

```
④ Código 5.128: IDT.cpp - idt_ptr::add_handle(interrupts::vector_e, void (*) (frame *))

87 void
88 idt_ptr::add_handle(interrupts::vector_e code, void (*handler)(frame *))
89 {
90     interrupts::idt_entry *reserved =
91         (interrupts::idt_entry *) (kernel::idtr.ptr +
92             static_cast<int>(code) * sizeof(interrupts::idt_entry));
93
94     reserved->set_offset((uint64_t) handler);
95     reserved->vector = static_cast<uint8_t>(code);
96     reserved->type_attr = static_cast<uint8_t>(interrupts::gate_e::interrupt) |
97         static_cast<uint8_t>(interrupts::status_e::enabled);
98 }
```

`add_handle` es quizá la función más importante de todo el módulo. Esta función asocia un vector con una función de tipo `void (*)(frame *)`, de tal forma que cuando se levante la interrupción del vector pasado, se ejecutará la función asociada.

© Código 5.129: IDT.cpp – `idt_entry::set_offset(uint64_t)`

```

103 void
104 idt_entry::set_offset(uint64_t offset)
105 {
106     helper::addr_t *aux = (helper::addr_t *)&offset;
107
108     this->offset_high = aux->offset_high;
109     this->offset_middle = aux->offset_middle;
110     this->offset_low = aux->offset_low;
111 }
```

`set_offset` establece, a partir de una dirección de memoria en formato `uint64_t`, la dirección de la función gestora de la interrupción en una entrada de la IDT.

© Código 5.130: IDT.cpp – `idt_entry::get_offset()`

```

116 uint64_t
117 idt_entry::get_offset()
118 {
119     helper::addr_t address = { this->offset_low, this->offset_middle, this->offset_high };
120
121     return *((uint64_t *)&address);
122 }
```

Esta función obtiene la dirección de la función que gestiona la interrupción de una determinada entrada en la IDT, en formato `uint64_t`.

5.4.9.1. Interrupt Handlers

Este apartado contiene las funciones que se comportan como gestores de las interrupciones (“Interrupt Handlers”). Estas funciones se encuentran en un archivo distinto de la IDT debido a que requieren flags especiales al compilarlas.

ⓘ Flags extra en el compilador

El compilador exige la flag `-mgeneral-reg-only` extra a la hora de compilar, esto se puede ver reflejado en la sección 5.3:

```
set_source_files_properties(${INTERRUPT_SOURCES} PROPERTIES
    → COMPILE_FLAGS "-mgeneral-reg-only ${COMPILE_FLAGS}")
```

`man gcc`: “Generate code which uses only the general-purpose registers. This will prevent the compiler from using floating-point and Advanced SIMD registers but will not impose any restrictions on the assembler.”

El compilador requiere que las funciones tomen como argumento un puntero a una estructura, creamos una vacía y se lo pasamos como argumento en cada gestor de interrupción:

```
④ Código 5.131: interrupts.h
11 namespace interrupts {
12
13 /** Mandatory in order to compile */
14 struct frame;
15
16 __attribute__((interrupt)) void reserved(frame *);
17 __attribute__((interrupt)) void keyboard(frame *);
18 __attribute__((interrupt)) void ethernet(frame *);
19
20 } // namespace interrupts
```

Ahora se listarán las interrupciones que se han escrito:

```
④ Código 5.132: interrupts.cpp — reserved(frame *)
17 __attribute__((interrupt)) void
18 reserved(frame *)
19 {
20     kernel::tty.println("Hola desde las interrupciones!");
21 }
```

Esta función se utiliza para comprobar que el código de las interrupciones se ha escrito correctamente. Se fuerza que se lance esta interrupción mediante el código en ensamblador `int 0x9`. Esta interrupción se registra en el código “bootstrap” de las interrupciones.

```
④ Código 5.133: interrupts.cpp — keyboard(frame *)
26 __attribute__((interrupt)) void
27 keyboard(frame *)
28 {
29     uint8_t status = io::inb(io::port::PS2);
30     /* Return correct receive */
31     io::outb(io::PIC1_COMMAND, 0x20);
32     kernel::keyboard.process_scancode(status);
33 }
```

La función `keyboard` es llamada cada vez que el teclado PS2 lanza una interrupción indicando que se ha pulsado o soltado una tecla. En esta función se obtiene el código PS2 recibido por parte del teclado y se le confirma que se ha recibido, finalmente se procesa este código en el módulo de teclado para hacer lo que corresponda con dicha tecla.

```
④ Código 5.134: interrupts.cpp — ethernet(frame *)
38 __attribute__((interrupt)) void
39 ethernet(frame *)
40 {}
```

Esta función está preparada para llamarse cada vez que la tarjeta de red RTL8139 reciba un paquete ethernet. Puesto que por ahora no se hace nada la función está vacía para que se pueda compilar, si se quisiese gestionar los paquetes recibidos habría que cambiar el contenido y, por ejemplo, implementar un stack de red capaz de gestionar los paquetes ethernet recibidos.

5.4.10. Teclado PS2

Este apartado tiene como objetivo explicar el módulo de teclado en *alma*. A la hora de desarrollar código para trabajar con un teclado hemos de tener en cuenta si su conexión es PS/2 o USB ya que funcionan de forma distinta. Pese a que el estándar actual es USB en *alma* se ha decidido implementar la versión con conexión PS/2.

Puerto PS/2

El puerto PS2 es un conector de 6 pines usado para conectar teclados y ratones en los ordenadores. Su nombre procede de la serie de ordenadores IBM Personal System/2 introducidos en 1987.



Figura 5.6: Puertos PS/2

El protocolo de comunicación es un canal serial síncrono bidireccional. El canal es ligeramente asimétrico: favorece la transmisión del periférico al ordenador.

El teclado nos enviará un código indicando si determinada tecla ha sido pulsada o levantada a través de una interrupción cuando esto pase. Hemos de tener en cuenta que no nos pasa el carácter que representa, es decir no nos va a devolver un `char 'a'` si se pulsa la tecla '`'a'`', ni tampoco `char 'A'` si se pulsa la tecla con mayúsculas activado. Toda la lógica de qué caracteres representan cada pulsación es tarea de *alma*.

Para empezar a controlar el estado del teclado definimos un `enum` con los distintos estados que queramos gestionar, por ejemplo normal (teclas minúsculas), shifted y mayus (teclas mayúsculas y especiales) y alt (teclas especiales):

```
Q Código 5.135: keyboard.h - enum class PS2_State
52 enum class PS2_State
53 {
54     Normal = 0,
55     Shifted = 1,
56     Alt = 2,
57     Mayus = 3,
58 };
```

Acto seguido construiremos un array de arrays que actuará como el conocido “mapa de teclas” que podemos configurar en los sistemas operativos. Este mapa de teclas puede personalizarse libremente para coincidir con el idioma y tipo de teclado que queramos conectar:

Q Código 5.136: keyboard.h — const char PS2_SCANCODES[] [4]

```

16 const char PS2_SCANCODES[] [4] = {
17     { 0x0, 0x0, 0x0, 0x0 }, { 0x0, 0x0, 0x0, 0x0 }, { '1', '!', '|', '1' },
18     { '2', '"', '@', '2' }, { '3', 0x0, '#', '3' }, { '4', '$', '~', '4' },
19     { '5', '%', 0x0, '5' }, { '6', '&', 0x0, '6' }, { '7', '/', '}' }, { '7' },
20     { '8', '(', '[', '8' }, { '9', ')', ']' }, { '0', '=', '}' }, { '0' },
21     { '\'', '?', '\\', 0x0 }, { 0x0, 0x0, 0x0, 0x0 }, { 0x0, 0x0, 0x0, 0x0 },
22     { 0x0, 0x0, 0x0, 0x0 }, { 'q', 'Q', 'Q' }, { 'w', 'W', 'W' },
23     { 'e', 'E', 'E' }, { 'r', 'R', 'R' }, { 't', 'T', '0x0, 'T' },
24     { 'y', 'Y', 'Y' }, { 'u', 'U', 'U' }, { 'i', 'I', '0x0, 'I' },
25     { 'o', 'O', 'O' }, { 'p', 'P', 'P' }, { '^', '^', '[' }, { '0x0 },
26     { '+', '*', ']' }, { '\n', '\n', '\n' }, { 0x0, 0x0, 0x0, 0x0 },
27     { 'a', 'A', 'A' }, { 's', 'S', 'S' }, { 'd', 'D', '0x0, 'D' },
28     { 'f', 'F', 'F' }, { 'g', 'G', 'G' }, { 'h', 'H', '0x0, 'H' },
29     { 'j', 'J', 'J' }, { 'k', 'K', 'K' }, { 'l', 'L', '0x0, 'L' },
30     { 0x0, 0x0, 0x0, 0x0 }, { 0x0, 0x0, 0x0, 0x0 }, { 0x0, 0x0, 0x0, 0x0 },
31     { 0x0, 0x0, 0x0, 0x0 }, { 0x0, 0x0, 0x0, 0x0 }, { 'z', 'Z', '0x0, 'Z' },
32     { 'x', 'X', 'X' }, { 'c', 'C', 'C' }, { 'v', 'V', '0x0, 'V' },
33     { 'b', 'B', 'B' }, { 'n', 'N', 'N' }, { 'm', 'M', '0x0, 'M' },
34     { ',', ';' }, { 0x0, 0x0 }, { '-' }, { 0x0, 0x0, 0x0 },
35     { 0x0, 0x0, 0x0, 0x0 }, { 0x0, 0x0, 0x0, 0x0 }, { 0x0, 0x0, 0x0, 0x0 },
36     { ' ', ' ', 0x0, 0x0 }, { 0x0, 0x0, 0x0, 0x0 }, { 0x0, 0x0, 0x0, 0x0 },
37     { 0x0, 0x0, 0x0, 0x0 }, { 0x0, 0x0, 0x0, 0x0 }, { 0x0, 0x0, 0x0, 0x0 },
38     { 0x0, 0x0, 0x0, 0x0 }, { 0x0, 0x0, 0x0, 0x0 }, { 0x0, 0x0, 0x0, 0x0 },
39     { 0x0, 0x0, 0x0, 0x0 }, { 0x0, 0x0, 0x0, 0x0 }, { 0x0, 0x0, 0x0, 0x0 },
40     { 0x0, 0x0, 0x0, 0x0 }, { 0x0, 0x0, 0x0, 0x0 }, { 0x0, 0x0, 0x0, 0x0 },
41     { 0x0, 0x0, 0x0, 0x0 }, { 0x0, 0x0, 0x0, 0x0 }, { 0x0, 0x0, 0x0, 0x0 },
42     { 0x0, 0x0, 0x0, 0x0 }, { 0x0, 0x0, 0x0, 0x0 }, { 0x0, 0x0, 0x0, 0x0 },
43     { 0x0, 0x0, 0x0, 0x0 }, { 0x0, 0x0, 0x0, 0x0 }, { 0x0, 0x0, 0x0, 0x0 },
44     { 0x0, 0x0, 0x0, 0x0 }, { 0x0, 0x0, 0x0, 0x0 }, { 0x0, 0x0, 0x0, 0x0 },
45     { 0x0, 0x0, 0x0, 0x0 }, { 0x0, 0x0, 0x0, 0x0 }, { '<', '>', 0x0, 0x0 },
46 };

```

El primer índice en el acceso a PS2_SCANCODES[] [] representa el código PS2 que nos enviará el teclado. El segundo índice será el estado en el que se encuentra el teclado.

Q Código 5.137: keyboard.h — class PS2

```

73 class PS2
74 {
75     public:
76     /** Process a new scancode */
77     void process_scancode(uint8_t);
78     /** Delete n chars from buffer */
79     void delete_char(uint16_t n = 1);
80     /** Add a new char */
81     void add_char(char);
82     /** Add a new mod char */
83     void add_modchar(char);
84     /** True if we need to update keyboard values */
85     bool update();
86     /** Read */
87     void scanf(char *, uint32_t);
88     /** Returns the buffered input text */
89     const char *get_buffer() const
90     {
91         return this->buffer;
92     };
93     /** Sets a new buffer to store text */
94     void set_buffer(char *buffer)
95     {
96         this->buffer = buffer;

```

```

97     };
98     uint16_t get_maxsize() const
99     {
100         return this->buffer_maxsize;
101    };
102    void set_maxsize(uint16_t maxsize)
103    {
104        this->buffer_maxsize = maxsize;
105    };
106    static void enable_keyboard();
107
108 private:
109     const static uint16_t DEFAULT_MAXSIZE = 256;
110     /** Input buffer max size */
111     uint16_t buffer_maxsize = DEFAULT_MAXSIZE;
112     /** Chars in the input buffer */
113     volatile uint16_t buffer_count = 0;
114     /** Input buffer */
115     char *buffer = nullptr;
116     /** Buffer has unrequested changes */
117     bool has_new_key = false;
118     /** Keyboard state */
119     PS2_State state = PS2_State::Normal;
120     enum class buffer_mode
121     {
122         circular,
123         limit
124     };
125     enum class read_mode
126     {
127         kernel,
128         scanf,
129     };
130     buffer_mode buffer_handling;
131     volatile read_mode input_mode = read_mode::kernel;
132
133     void update_scanf();
134 };

```

La clase PS2 será la encargada de gestionar todo lo relacionado con el teclado. Será el objeto almacenado en `kernel.h` para que esté disponible en otros módulos. Lo más remarcable de esta clase es que tenemos dos `enums` indicando si el buffer que tenemos dentro de la clase es circular o con límite, también tenemos un enum que nos indica si estamos en modo “normal” o en `scanf`.

Una vez construido e inicializado el teclado esta clase recibe todas las pulsaciones del teclado, las transforma en el carácter correspondiente y lo almacena en un buffer privado, si el buffer se sobrepasa de tamaño se vuelve a escribir desde el principio. De esta forma podemos tener las últimas pulsaciones del teclado para implementar combinaciones de teclas como en los sistemas operativos.

Si llamamos a la función `scanf` del objeto, este entra en un modo especial en el que el buffer que usamos se cambia por uno introducido por el usuario¹⁷ y cuando se deja de introducir texto se vuelve al modo anterior.

¹⁷Esto se hace para evitar copiar del buffer del kernel al del usuario.

```

19 void
20 PS2::process_scancode(uint8_t keycode)
21 {
22     /* keycode received inside char map */
23     if (keycode < PS2_SCANCODE_SIZE) {
24         char letter = PS2_SCANCODES[keycode] [static_cast<int>(this->state)];
25
26         if (letter != 0x0) {
27             /* User entered a char */
28             this->add_char(letter);
29             /* Special hardwire to scanf buffer to improve performance */
30             if (this->input_mode == PS2::read_mode::scanf)
31                 this->update_scanf();
32             return;
33         } // else unsupported char OR modkey
34     }
35
36     /* Handle modkeys */
37     switch (keycode) {
38         case 0x2a:
39             this->state = PS2_State::Shifted;
40             break;
41         case 0xaa:
42             if (this->state != PS2_State::Mayus)
43                 this->state = PS2_State::Normal;
44             break;
45         case 0x3a:
46             if (this->state == PS2_State::Mayus)
47                 this->state = PS2_State::Normal;
48             else
49                 this->state = PS2_State::Mayus;
50             break;
51         case 0xba:
52             break;
53         /* Del */
54         case 0xe0: {
55             this->delete_char();
56             /* Special hardwire to scanf buffer to improve performance */
57             if (this->input_mode == PS2::read_mode::scanf)
58                 this->update_scanf();
59             break;
60         }
61         case 0x8e:
62             break;
63     }
64 }
```

La función `process_scancode` es quizás la función más importante del módulo. Esta función es la encargada de procesar un código PS2 fuente de una interrupción al pulsarse o levantarse una tecla.

En primer lugar se comprueba si el código está dentro del mapa de teclas que tenemos definido (línea 23). Si está dentro del mapa de teclas se obtiene la tecla y, si no es 0x0 (no definida) se procesa la tecla añadiéndola al buffer. Si estamos en modo `scanf` indicamos que tenemos cambios en el buffer (importante para visualizar lo que escribe el usuario por pantalla). En caso de que no lo tengamos en la tabla significa que son teclas especiales como `Alt`, `Shift`, `Del`, etc.

© Código 5.139: keyboard.cpp — PS2::delete_char(uint16_t)

```
73 void
74 PS2::delete_char(uint16_t n)
75 {
76     if (n > buffer_count)
77         return;
78     this->buffer_count = this->buffer_count - n; // -= n volatile deprecated
79     this->has_new_key = true;
80 }
```

La función `delete_char` elimina n caracteres del buffer que tenga PS2 en ese momento. Por defecto elimina un carácter solo.

© Código 5.140: keyboard.cpp — PS2::add_char(char)

```
85 void
86 PS2::add_char(char letter)
87 {
88     /* Surpass buffer handling*/
89     if (this->buffer_count >= PS2::buffer_maxsize) {
90         switch (this->buffer_handling) {
91             case PS2::buffer_mode::circular:
92                 this->buffer_count = 0;
93                 break;
94             case PS2::buffer_mode::limit:
95                 return;
96         }
97     }
98
99     this->buffer[this->buffer_count] = letter;
100    this->buffer_count = this->buffer_count + 1; // += 1 volatile deprecated
101    this->has_new_key = true;
102 }
```

La función `add_char` añade un carácter al buffer que tenga PS2 en ese momento, haciendo distinción en caso de que estemos en modo normal o modo `scanf`. Si se está en modo normal, al llegar al límite se empieza a llenar desde el principio del buffer mientras que si estamos en modo `scanf` se deja de leer.

Además la función `add_char` como `delete_char` actualizan la variable `has_new_key` para indicar que el buffer ha sufrido modificaciones. Esta variable se utiliza en `update()`.

© Código 5.141: keyboard.cpp — PS2::update()

```
107 bool
108 PS2::update()
109 {
110     auto temp = this->has_new_key;
111     this->has_new_key = false;
112     return temp;
113 }
```

La función `update` se llama para consultar actualizaciones en el buffer del teclado. Su uso común suele ser de la forma `while(true) if (keyboard->update) foo();`. En caso de que consultemos una actualización y la haya, esta se marca como ya “procesada”.

```
④ Código 5.142: keyboard.cpp — PS2::scanf(char *, uint32_t)

120 void
121 PS2::scanf(char *buffer, uint32_t maxsize)
122 {
123     if (buffer == nullptr || maxsize <= 0)
124         return;
125
126     auto buffer_backup = this->buffer;
127     auto count_backup = this->buffer_count;
128     auto maxsize_backup = this->buffer_maxsize;
129     auto has_new_key_backup = this->has_new_key;
130     auto mode_backup = this->buffer_handling;
131
132     this->buffer = buffer;
133     this->buffer_maxsize = maxsize - 1;
134     this->buffer_count = 0;
135     this->has_new_key = false;
136     this->buffer_handling = PS2::buffer_mode::limit;
137     this->input_mode = read_mode::scanf;
138
139     /* Wait until user finished introducing text */
140     while (this->input_mode == read_mode::scanf) {
141
142
143         this->buffer[this->buffer_count] = '\0';
144         kernel::tty.newline();
145
146         /* Restore kernel buffer */
147         this->buffer = buffer_backup;
148         this->buffer_maxsize = maxsize_backup;
149         this->buffer_count = count_backup;
150         this->has_new_key = has_new_key_backup;
151         this->buffer_handling = mode_backup;
152     }
}
```

La función `scanf` acepta como parámetro un buffer para almacenar el texto introducido y su tamaño. `scanf` lo que hace es sustituir el buffer de entrada que tiene el objeto PS2 para que en vez de guardarse las teclas en dicho buffer se guarden en el que se pasa por parámetro, esto evita una copia innecesaria.

Además se pone el objeto PS2 en modo `scanf`, para que funciones como `process_scancode` llamen a la función `update_scanf`. Esta forma es novedosa en *alma* ya que anteriormente en `scanf` funcionaba de una forma más genérica y aparentemente con mejor diseño, pero este mecanismo de poner el objeto PS2 en modo `scanf` y llamar desde las propias funciones a `update_scanf` cuando hay cambios directamente es **mucho** más rápida.

```
④ Código 5.143: keyboard.cpp — PS2::update_scanf()

161 void
162 PS2::update_scanf()
163 {
164     static uint32_t last_text_size = 0;
165     if (this->buffer_count > 0 && this->buffer[this->buffer_count - 1] == '\n') {
166         /* User ends scanf with enter */
167         this->input_mode = read_mode::kernel;
168         last_text_size = 0;
169         return;
170     } else if (this->buffer_count > last_text_size) {
171         /* User enters new character(s) */
```

```

172     auto new_chars = this->buffer_count - last_text_size;
173     buffer[this->buffer_count] = '\0';
174     kernel::tty.print(&this->buffer[last_text_size]);
175 } else if (this->buffer_count < last_text_size) {
176     /* User removes character(s) */
177     auto del_chars = last_text_size - this->buffer_count;
178     int new_x = kernel::tty.get_x() - (del_chars * kernel::tty.glyph_x());
179     if (new_x < 0) [[unlikely]] {
180         kernel::tty.set_x(kernel::tty.get_width() - abs(new_x));
181         kernel::tty.set_y(kernel::tty.get_y() - kernel::tty.glyph_y());
182     } else [[likely]] {
183         kernel::tty.set_x(new_x);
184     }
185     kernel::tty.pushCoords(kernel::tty.get_x(), kernel::tty.get_y());
186     kernel::tty.pushColor(screen::color_e::BLACK);
187     for (uint32_t i = 0; i < del_chars; i++)
188         kernel::tty.put((char)0xdb); // Full color character
189     kernel::tty.popColor();
190     kernel::tty.popCoords();
191 }
192 last_text_size = this->buffer_count;
193 }
```

`update_scanf` es una función que se llama cuando se añade o se elimina un carácter durante `scanf`. Esta función es la encargada de actualizar el texto que aparece en pantalla conforme el contenido del buffer.

Hay maneras mucho más simples de escribirla, por ejemplo redibujando la pantalla entera o la línea entera. El problema es que esto es muy costoso ya que hay que iterar la pantalla para limpiarla (la caché también en caso de usar el renderizado rápido), luego reescribirla, etc. Esta versión es más rápida y solo redibuja los caracteres necesarios, aunque a nivel de código sea un poco más compleja.

Código 5.144: `keyboard.cpp` — `PS2::enable_keyboard()`

```

198 void
199 PS2::enable_keyboard()
200 {
201     kernel::idtr.add_handle(interrupts::vector_e::keyboard, interrupts::keyboard);
202 }
```

Esta función activa el teclado, asociando el vector de las interrupciones correspondiente al teclado PS2 con la función gestora de la interrupción `interrupts::keyboard`. El contenido de la función gestora de la interrupción aparece en el Código 5.133.

Ratón PS/2

Teniendo una implementación para el teclado PS/2 es “sencillo” transformarla en una implementación para el ratón PS/2.

alma no dispone de soporte para ratón por falta de tiempo pero sería un proyecto a futuro muy viable disponer de un ratón simple en *alma*.

5.4.11. ACPI

ACPI (Advanced Configuration and Power Interface) es una especificación desarrollada para establecer una interfaz común en la industria que permita que el sistema operativo pueda configurar dispositivos de la placa base y gestionar las políticas de energía [12].

Para proporcionar flexibilidad de implementación a los fabricantes de hardware, ACPI utiliza tablas para describir la información del sistema, capacidades y métodos para controlar dichas capacidades. Mediante este uso de tablas se consigue controlar los dispositivos del sistema sin necesidad de conocer como están implementados.

Si se quiere acceder a las tablas primero tendremos que encontrar el “Root System Description Pointer” (RSDP). El RSDP es una estructura que encontramos en la memoria del sistema, inicializada por el firmware de la plataforma [12, p. 109]. La estructura es la que encontramos en el Código 5.145 (ACPI 1.0) y 5.146 (ACPI 2.0+).

© Código 5.145: acpi.h – struct rsdp_v1

```
41 struct rsdp_v1
42 {
43     /** Null terminated string that has to be "RSDP PTR " (see the last space) */
44     char signature[8];
45     /** byte cast of the sum of all bytes must be = 0*/
46     uint8_t checksum;
47     /** OEM string */
48     char oem[6];
49     /** ACPI Version */
50     uint8_t version;
51     /** RSDT physic address*/
52     uint32_t rsdt;
53 } __attribute__((packed));
```

© Código 5.146: acpi.h – struct rsdp_v2

```
58 struct rsdp_v2
59 {
60     rsdp_v1 header_v1;
61     uint32_t length;
62     /** XSDT physic address (if ACPI 2.0, use this instead of header_v1->rsdt, even in 32bit) */
63     uint64_t xsdt;
64     uint8_t checksum_v2;
65     uint8_t reserved[3];
66
67     sdt *find_table(const char *);
68     void print_acpi_tables();
69 } __attribute__((packed));
```

ⓘ rsdp_v2 retrocompatible con rsdp_v1

Como podemos ver, rsdp_v2 (usada si tenemos ACPI 2.0+) es retrocompatible con la versión anterior. Esto es debido a que al hacer rsdp *aux = (rsdp *)&memory si estamos con ACPI 1.0 leeremos los bytes desde el principio y leeremos solo la primera parte.

El RSDP nos da acceso a la `rsdt` o `xsdt` mediante una dirección física (32 y 64 bits respectivamente). Debemos usar la `rsdt` en caso de estar en ACPI 1.0 y `xsdt` en caso de estar en ACPI 2.0+, incluso en entornos de 32 bits.

Podemos ver en la línea 44 que disponemos de una “firma” constante para comprobar que la estructura está correctamente formada y que, en efecto, se trata de la RSDP. También se dispone de un *checksum* para garantizar la correctitud de los datos de la estructura.

La RSDT (o XSDT) es la tabla principal de ACPI. Como todo van a ser tablas similares podemos definir la cabecera de las tablas de forma genérica (Código 5.147). La RSDT y XSDT serán estructuras que empiezan con una cabecera `sdt` (Código 5.147) y a continuación un array de direcciones (32 o 64 bits) a otras tablas del sistema (Figura 5.7).

```
© Código 5.147: acpi.h – struct sdt
19 struct sdt
20 {
21     char signature[4];
22     uint32_t length;
23     uint8_t revision;
24     uint8_t checksum;
25     char oem[6];
26     char oem_table[8];
27     uint32_t oem_version;
28     uint32_t creator_id;
29     uint32_t creator_revision;
30     bool check_signature(const char *);
31 } __attribute__((packed));
```

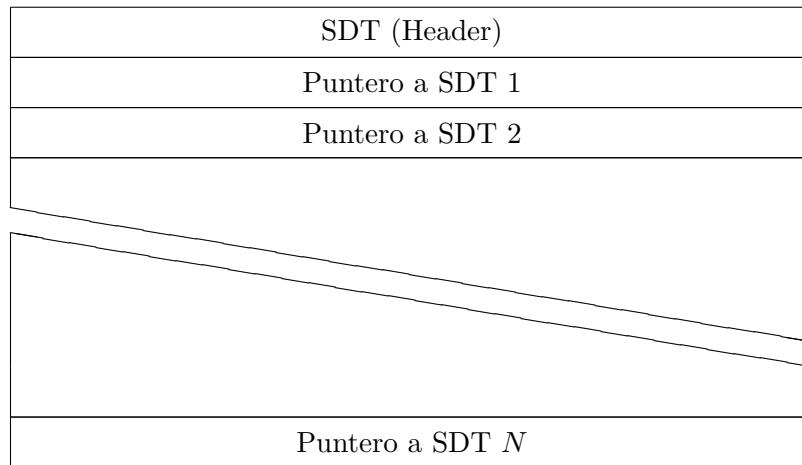


Figura 5.7: Formato de la tabla RSDP/XSDT

El número de tablas (N en la Figura 5.7) que tendremos lo podemos calcular como:

- `sdt.length - sizeof(sdt) / sizeof(uint32_t)` (32 bits)
- `sdt.length - sizeof(sdt) / sizeof(uint64_t)` (64 bits)

Los punteros a otras tablas que encontramos en la RSDT o XSDT pueden ser a muchos tipos de tablas, cada una estará identificada por cuatro caracteres¹⁸ (línea 21). Se ha listado algunas tablas que podemos encontrar en la Tabla 5.4.

Código	Descripción
FADT	Fixed ACPI Description Table
FACS	Firmware ACPI Control Structure
DSDT	Differentiated System Descriptor Table
SSDT	Secondary System Description Table
MADT	Multiple APIC Description Table
SBST	Smart Battery Table
ECDT	Embedded Controller Boot Resources Table
SLIT	System Locality Distance Information Table
SRAT	System Resource Affinity Table
CPEP	Corrected Platform Error Polling Table
MSCT	Maximum System Characteristics Table
RASF	ACPI RAS Feature Table
MPST	Memory Power State Table
PMTT	Platform Memory Topology Table
BGRT	Boot Graphics Resource Table
FPDT	Firmware Performance Data Table
GTDT	Generic Timer Description Table
NFIT	NVDIMM Firmware Interface Table
HMAT	Heterogeneous Memory Attributes
PDTT	Platform Debug Trigger Table
PPTT	Processor Properties Topology Table

Tabla 5.4: Tablas ACPI [12, p. 112]

Podemos encontrar una lista de las “firmas” reservadas para ciertas tablas en la página 120 – 124 en la especificación de ACPI [12].

Ahora pasaremos a definir una serie de funciones que nos ayuden a trabajar con estas tablas, como mínimo necesitaremos una función para buscar una tabla en base a su firma, otra para comprobar que la firma de una tabla es correcta y finalmente incluiremos una función auxiliar para imprimir todas las tablas.

```
Q Código 5.148: acpi.cpp - rsdp_v2::find_table(const char *)
41 sdt *
42 rsdp_v2::find_table(const char *signature)
43 {
44     acpi::sdt *xsdt = (acpi::sdt *)this->xsdt;
45
46     int entries = (xsdt->length - sizeof(acpi::sdt)) / sizeof(uint64_t);
47 }
```

¹⁸Importante, no están terminados con un carácter nulo, no usar `strcmp` o similares.

```

48     for (int i = 0; i < entries; i++) {
49         acpi::sdt *tbl =
50             (acpi::sdt *)*((uint64_t *)((uint64_t)xsdt + sizeof(acpi::sdt) + (i * sizeof(uint64_t))));
51         if (tbl->check_signature(signature));
52             return tbl;
53     }
54     return nullptr;
55 }
```

`find_table(const char *)` nos permite, en base a una firma pasada como array de chars (sin terminación a nulo) encontrar una tabla. La tabla se devuelve como puntero a `sdt` o `nullptr` si no se encuentra.

© Código 5.149: acpi.cpp – `sdt::check_signature(const char *)`

```

65 bool
66 sdt::check_signature(const char *signature)
67 {
68     return (strncmp(this->signature, signature, 4) == 0);
69 }
```

`check_signature(const char *)` nos sirve como abstracción para comprobar, en una `sdt`, si la firma coincide con la pasada como argumento. Como podemos ver se utiliza `strncmp` para comprobar solo 4 caracteres.

© Código 5.150: acpi.cpp – `rsdp_v2::print_acpi_tables()`

```

19 void
20 rsdp_v2::print_acpi_tables()
21 {
22     acpi::sdt *xsdt = (acpi::sdt *)this->xsdt;
23
24     int entries = (xsdt->length - sizeof(acpi::sdt)) / sizeof(uint64_t);
25     for (int i = 0; i < entries; i++) {
26         acpi::sdt *tbl =
27             (acpi::sdt *)*((uint64_t *)((uint64_t)xsdt + sizeof(acpi::sdt) + (i * sizeof(uint64_t))));
28         char str[5] = {
29             tbl->signature[0], tbl->signature[1], tbl->signature[2], tbl->signature[3], '\0'
30         };
31         kernel::tty.println(str);
32     }
33 }
```

Finalmente tenemos la función `print_acpi_tables()`, esta función imprime por pantalla las firmas de todas las tablas que tenemos disponibles en nuestra máquina.

```
welcome to the alma kernel
$ acpi
FACP
APIC
HPET
MCFG
WAET
BGRT
$
```

Figura 5.8: Ejemplo de `print_acpi_tables()` en qemu

5.4.12. PCI Express

PCI (*Peripheral Component Interconnect*) es una especificación desarrollada por Intel para evitar que la industria desarrolle distintas arquitecturas de buses locales¹⁹. La versión 1.0 de la especificación se publicó el 22 de Junio de 1992 [38]. El consorcio industrial que actualmente gestiona las especificaciones de PCI es el denominado “PCI Special Interest Group (SIG)”.

PCI Express (a continuación PCIe) reemplaza los anteriores PCI y PCI-X para ofrecer mayor flexibilidad y velocidad. PCIe introduce un cambio arquitectural manteniendo el núcleo software de PCI. Es importante mencionar que PCIe es retrocompatible con PCI en el aspecto software.

En *alma* trabajaremos con esta especificación para poder configurar y comunicarnos con los dispositivos que conectemos al ordenador, por ejemplo, una tarjeta de red. La mayoría de componentes del ordenador se conectan a ranuras PCIe (Figura 5.9) como ya sabemos. Los componentes que se conectan a las ranuras PCIe podremos configurarlos y acceder a ellos mediante la especificación PCIe.



Figura 5.9: Conexiones PCI y PCIe de un ordenador

A Arquitectura PCIe

En este trabajo no se tiene como objetivo explicar la arquitectura interna de PCIe. No se pretende explicar el funcionamiento de los envíos, conexiones, etc. Nos centraremos en la capa “software” de PCIe (obtener y configurar los dispositivos).

¹⁹A fin de poder tener una versión estandarizada que beneficiase a todos.

Para empezar a trabajar con PCIe nos remontaremos a la sección 5.4.11. En esta sección se introduce el concepto de tablas ACPI, para empezar a trabajar con PCIe utilizaremos la tabla “MCFG”.

La tabla MCFG contiene estructuras que almacenan información sobre los dispositivos PCI. El formato de las estructuras es el siguiente:

© Código 5.151: pci.h – struct device_config

```
16 struct device_config
17 {
18     uint64_t baseaddr;
19     uint16_t pci_seg_group;
20     uint8_t start_bus;
21     uint8_t end_bus;
22     uint32_t reserved;
23 } __attribute__((packed));
```

Ahora definiremos las cabeceras de los dispositivos PCI:

© Código 5.152: pci.h – struct device_header

```
28 struct device_header
29 {
30     /* manufacturer */
31     uint16_t vendor;
32     uint16_t id;
33     uint16_t command;
34     uint16_t status;
35     uint8_t revision;
36     uint8_t program_interface;
37     uint8_t dev_subclass;
38     uint8_t dev_class;
39     uint8_t cache_line_size;
40     uint8_t latency_timer;
41     uint8_t header_type;
42 } __attribute__((packed));
```

A continuación de la cabecera PCI, los dispositivos almacenan otra cabecera que podemos ver como extensión de la anterior, esta cabecera depende del tipo de dispositivo PCI que tengamos (campo header_type en device_header). Dependiendo de si el valor es 0x0, 0x1 o 0x2 podremos encontrar las siguientes cabeceras:

© Código 5.153: pci.h – struct header_t0

```
48 struct header_t0
49 {
50     uint8_t BIST;
51     uint32_t BAR[6];
52     uint32_t cardbus_ptr;
53     uint16_t subsystem_vendor;
54     uint16_t subsystem;
55     uint32_t exp_rom_addr;
56     uint8_t capabilities;
57     uint8_t reserved;
58     uint16_t reserved2;
59     uint32_t reserved3;
60     uint8_t int_line;
61     uint8_t int_pin;
```

```
62     uint8_t min_gran;
63     uint8_t max_lat;
64 } __attribute__((packed));
```

Q Código 5.154: pci.h – struct header_t1

```
69 struct header_t1
70 {
71     uint8_t BIST;
72     uint32_t BAR[2];
73     uint8_t primary_bus;
74     uint8_t secondary_bus;
75     uint8_t subordinate;
76     uint8_t secondary_lat;
77     uint8_t io_base;
78     uint8_t io_limit;
79     uint16_t sec_status;
80     uint16_t memory_base;
81     uint16_t memory_limit;
82     uint16_t prefetch_mem_base;
83     uint16_t prefetch_mem_limit;
84     uint32_t prefetch_upper_base;
85     uint32_t prefetch_upper_limit;
86     uint16_t io_upper_base;
87     uint16_t io_upper_limit;
88     uint8_t capabilities;
89     uint8_t reserved1;
90     uint16_t reserved2;
91     uint32_t expansion_rom_addr;
92     uint8_t int_line;
93     uint8_t int_pin;
94     uint16_t bridge_crtl;
95 } __attribute__((packed));
```

Q Código 5.155: pci.h – struct header_t2

```
100 struct header_t2
101 {
102     uint8_t BIST;
103     uint32_t cardbus_socket;
104     uint8_t capabilities;
105     uint8_t reserved;
106     uint16_t secondary_status;
107     uint8_t pci_bus_num;
108     uint8_t cardbus_bus_num;
109     uint8_t subordinate_bus_num;
110     uint8_t cardbus_lat_timer;
111     uint32_t memory_base_addr_0;
112     uint32_t memory_limit_0;
113     uint32_t memory_base_addr_1;
114     uint32_t memory_limit_1;
115     uint32_t io_base_addr_0;
116     uint32_t io_limit_0;
117     uint32_t io_base_addr_1;
118     uint32_t io_limit_1;
119     uint8_t int_line;
120     uint8_t int_pin;
121     uint16_t bridge_crtl;
122     uint16_t subsystem_vendor;
123     uint16_t subsystem;
124     uint32_t legacy_addr;
125 } __attribute__((packed));
```

Una vez hayamos encontrado y clasificado todas las cabeceras de los dispositivos, los juntaremos en una estructura en forma de nodo de lista enlazada. Esto lo hacemos para poder almacenar en las variables globales del kernel una lista enlazada de dispositivos PCI encontrados y poder recorrerla.

```
© Código 5.156: pci.h - struct pci_device
130 struct pci_device
131 {
132     device_header *header;
133     void *header_ext;
134     uint16_t device;
135     uint16_t bus;
136     uint16_t function;
137     pci_device *prev;
138     pci_device *next;
139 };
```

A continuación definiremos la estructura de dos campos que tienen `header_t0` y `header_t1`. El campo en cuestión es BAR, un array de las siguientes estructuras que conforman direcciones de memoria. Definimos dos estructuras porque los campos pueden ser de una u otra y elegiremos dependiendo de la cabecera PCI.

```
© Código 5.157: pci.h - struct BAR_mem
146 struct BAR_mem
147 {
148     bool mem : 1;
149     unsigned int type : 2;
150     bool prefetchable : 3;
151     unsigned int addr : 26;
152 };
```

```
© Código 5.158: pci.h - struct BAR_io
154 struct BAR_io
155 {
156     bool io : 1;
157     bool reserved : 1;
158     unsigned int addr : 30;
159 };
```

Ahora pasaremos a detallar las funciones encargadas de cargar la lista enlazada de dispositivos PCI en el kernel.

```
© Código 5.159: pci.cpp - enum_pci(acpi::sdt *)
105 void
106 enum_pci(acpi::sdt *mcfg)
107 {
108     if (mcfg == nullptr)
109         return;
110
111     if (!mcfg->check_signature(pci::MCFG_SIGN))
112         return;
113
114     int entries =
115         ((mcfg->length) - (sizeof(acpi::sdt) + sizeof(uint64_t))) / sizeof(pci::device_config);
116 }
```

```

117     for (int i = 0; i < entries; i++) {
118         /* Table content is after the header, that's why +sizeof(acpi::sdt) */
119         pci::device_config *device =
120             (pci::device_config *)((uint64_t)mcfg + (sizeof(acpi::sdt) + sizeof(uint64_t)) +
121             sizeof(pci::device_config) * i);
122
123         startaddr = device->baseaddr;
124         startbus = device->start_bus;
125
126         for (uint64_t bus = device->start_bus; bus < device->end_bus; bus++)
127             enum_bus(device->baseaddr, bus);
128     }
129 }
```

En primer lugar tenemos la función `enum_pci(acpi::sdt *)`, esta función obtiene por parámetro la tabla MCFG y se comprueba que no sea nula y que su firma sea correcta. A continuación itera todas las entradas de la tabla y, para cada entrada, enumera el segmento de bus marcado por la tabla.

© Código 5.160: pci.cpp – `enum_bus(uint64_t, uint64_t)`

```

84 void
85 enum_bus(uint64_t addr, uint64_t bus)
86 {
87     _bus = bus;
88     uint64_t offset = bus << 20;
89
90     uint64_t busaddr = addr + offset;
91
92     pci::device_header *device = (pci::device_header *)busaddr;
93
94     /* not valid */
95     if (device->id == 0 || device->id == 0xffff)
96         return;
97     for (uint64_t dev_i = 0; dev_i < 32; dev_i++)
98         enum_dev(busaddr, dev_i);
99 }
```

`enum_bus(uint64_t, uint64_t)` comprueba si se tiene dispositivo PCI. Si existe, itera sobre los 32 posibles dispositivos²⁰ (`device`) y los enumera.

© Código 5.161: pci.cpp – `enum_dev(uint64_t, uint64_t)`

```

62 void
63 enum_dev(uint64_t addr, uint64_t dev)
64 {
65     _device = dev;
66     uint64_t offset = dev << 15;
67
68     uint64_t devaddr = addr + offset;
69
70     pci::device_header *device = (pci::device_header *)devaddr;
71
72     /* not valid */
73     if (device->id == 0 || device->id == 0xffff)
74         return;
75 }
```

²⁰Para entender correctamente esta parte y el por qué de “buses, dispositivos y funciones” es mejor leer la especificación PCI o similares.

```

76     for (uint64_t fun_i = 0; fun_i < 8; fun_i++) {
77         enum_fun(devaddr, fun_i);
78     }
79 }
```

`enum_dev(uint64_t, uint64_t)` comprueba si se tiene dispositivo PCI. Si existe, itera sobre las 8 posibles funciones y los enumera.

© Código 5.162: pci.cpp – `enum_fun(uint64_t, uint64_t)`

```

23 void
24 enum_fun(uint64_t addr, uint64_t fun)
25 {
26     _function = fun;
27     uint64_t offset = fun << 12;
28     uint64_t funaddr = addr + offset;
29     pci::device_header *device = (pci::device_header *)funaddr;
30     if (device->id == 0 || device->id == 0xffff)
31         return;
32
33     static pci_device *prev = nullptr;
34     pci::pci_device *dev = (pci::pci_device *)kernel::heap.malloc(sizeof(pci::pci_device));
35     dev->header = device;
36     dev->device = _device;
37     dev->function = _function;
38     dev->bus = _bus;
39     dev->prev = prev;
40     dev->header_ext = (void *)((uint8_t *)device + sizeof(pci::device_header));
41
42     if (prev == nullptr)
43         kernel::devices = dev;
44     else
45         prev->next = dev;
46     prev = dev;
47 }
```

`num_fun(uint64_t, uint64_t)` es la función clave de la enumeración PCI. Se comprueba si se tiene dispositivo PCI, si existe se construye un nuevo nodo en la lista enlazada con los valores anteriores de “bus, dispositivo y función” y su cabecera. Una vez se crea el nodo en la lista enlazada ya tenemos el dispositivo PCI encontrado y podemos acceder a él desde las variables globales de *alma*.

```

welcome to the alma kernel
$ pci
* 0x00000000000000000000000000000000 - 0x00000000000000000000000000000000
$
```

Figura 5.10: Listado de dispositivos PCI en alma (qemu)

Podemos ver el resultado de la enumeración en la Figura 5.10. El comando `pci` itera sobre la lista enlazada construida e imprime un dispositivo PCI (Identificador del fabricante – Identificador del dispositivo) por línea.

5.4.13. Tarjeta de Red RTL8139

En este apartado se tiene como objetivo ver el desarrollo de un controlador para la tarjeta de red²¹ RTL8139 (Figura 5.11). Esto permitirá a *alma* enviar y recibir paquetes ethernet para comunicarse con otras máquinas.

Recordando el modelo OSI de capas, en este trabajo programaremos a nivel de enlace de datos, el nivel físico (el único por debajo) está gestionado por la tarjeta de red. Si se quisiera se podría programar niveles superiores del modelo OSI para que el núcleo disponga de conectividad IP, TCP, etc.



Figura 5.11: Tarjeta de red RTL8139

En primer lugar hay que comentar que la tarjeta de red será un dispositivo PCI y, como tal, podremos configurarla mediante dos métodos que soporta PCI Express (PCIe): I/O o MMIO (*Memory Mapped I/O*), en este apartado se explicará MMIO.

Las configuraciones de la tarjeta de red se hacen en un buffer de memoria donde cada offset del buffer es un “registro”. Podemos encontrar una serie de *offsets* y sus características en el manual de la tarjeta de red [39, p. 12]. Con estos offsets crearemos un enum para tenerlos clasificados y facilitar su acceso:

```
④ Código 5.163: rtl8139.h — enum rtl8139_config
18 enum rtl8139_config
19 {
20     MAC0 = 0x0,
21     MAC1 = 0x1,
22     MAC2 = 0x2,
23     MAC3 = 0x3,
24     MAC4 = 0x4,
25     MAC5 = 0x5,
26     CONFIG1 = 0x52,
27     COMMAND = 0x32,
28     RECVBUFF = 0x30,
29     IMR = 0x3c,
30     ISR = 0x3e,
31     RCR = 0x44,
32     CR = 0x37,
```

²¹Tarjetas con chipset RL8139.

```

33     TSAD0 = 0x20,
34     TSAD1 = 0x24,
35     TSAD2 = 0x28,
36     TSAD3 = 0x2c,
37     TSD0 = 0x10,
38     TSD1 = 0x14,
39     TSD2 = 0x18,
40     TSD3 = 0x1c,
41 };

```

Ahora crearemos una clase que represente la tarjeta de red. El constructor tendrá como parámetro el dispositivo PCI de la tarjeta de red. Este dispositivo se obtiene en la función `bootstrap::rtl18139()` explicada en la sección 5.4.2.

© Código 5.164: rtl18139.h – class rtl18139

```

48 class rtl18139
49 {
50     public:
51     rtl18139(pci::pci_device *device);
52     rtl18139() = default;
53     rtl18139 *operator=(rtl18139 &&);
54     void start();
55     void send_packet(uint32_t, uint64_t);

56     // private: (removed to debug better)
57     static const auto PCI_ID = 0x8139;
58     rtl18139_config TSAD_array[4] = { TSAD0, TSAD1, TSAD2, TSAD3 };
59     rtl18139_config TSD_array[4] = { TSD0, TSD1, TSD2, TSD3 };
60     uint8_t tx_cur = 0; // used to cycle TSAD and TSD arrays
61     pci::pci_device *device;
62     uint64_t mem_addr;

63     template<typename T>
64     void setconfig(rtl18139_config reg, T value)
65     {
66         *((T *)((uint8_t *)this->mem_addr + static_cast<int>(reg))) = value;
67     }
68     template<typename T>
69     T getconfig(rtl18139_config reg)
70     {
71         return *((T *)((uint8_t *)this->mem_addr + static_cast<int>(reg)));
72     }
73 }
74 
```

En la clase que acabamos de definir tenemos dos funciones importantes a remarcar: `setconfig` (línea 71) y `getconfig` (línea 66). Estas funciones nos ayudan a leer o establecer una configuración en el dispositivo PCI mediante MMIO. Están escritas con una template para poder leer o escribir tamaños variables de registros.

⚠️ private comentado

Todos los elementos de la clase son públicos debido a que `private` está comentado. Esto se ha hecho para poder imprimir registros de la tarjeta de red mediante comandos en *alma*, en un uso real habría que comentar `private`.

Otros miembros de la clase que cabe remarcar son los dos arrays que aparecen: TSAD y TSD, que son una serie de buffers para poder almacenar el estado de la transmisión y la dirección del buffer de forma circular. Cada array contiene los offsets donde se debe escribir o consultar el registro, debemos usar uno y luego el siguiente (marcado por el miembro `tx_cur`), así sucesivamente.

El miembro estático `PCI_ID` (línea 58) sirve para encontrar el dispositivo PCI.

```
④ Código 5.165: rtl8139.cpp — rtl8139::rtl8139(pci::pci_device *)
```

```
15 rtl8139::rtl8139(pci::pci_device *device)
16 {
17     this->device = device;
18     pci::header_t0 *ext_hdr = (pci::header_t0 *)device->header_ext;
19     this->mem_addr = (ext_hdr->BAR[1] & 0xffffffff);
20     kernel::translator.map(this->mem_addr, this->mem_addr);
21 }
```

El constructor es el encargado de obtener la dirección de memoria base para realizar las operaciones con “offsets” en MMIO. Como podemos ver se utilizan las cabeceras extendidas de PCI y el miembro BAR del que ya se ha hablado en el trabajo.

▲ El constructor no inicializa la tarjeta de red

A diferencia de lo que podemos pensar, al construir la tarjeta de red no he hecho que se “arranque” y esté lista para enviar paquetes. Esta separación se ha hecho para poder ver los registros de la tarjeta antes de arrancarse.

```
④ Código 5.166: rtl8139.cpp — rtl8139::start()
```

```
38 void
39 rtl8139::start()
40 {
41     /** Enable PCI Bus Mastering */
42     this->device->header->command |= (1 << 2);
43     /** Turn on the device */
44     this->setconfig<uint8_t>(rtl8139_config::CONFIG1, 0x0);
45     /** Software reset */
46     this->setconfig<uint8_t>(rtl8139_config::CR, 0x10);
47     while ((this->getconfig<uint8_t>(rtl8139_config::CR) & 0x10) != 0) {
48 }
49     /** Set receive buffer */
50     void *buffer = kernel::allocator.request_cont_page(9);
51     this->setconfig<uintptr_t>(rtl8139_config::RECVBUFF, (uintptr_t)buffer);
52     /** Set IMR + ISR */
53     this->setconfig<uint16_t>(rtl8139_config::IMR, 0b101);
54     this->setconfig<uint16_t>(rtl8139_config::ISR, 0);
55     /** Set RCR */
56     this->setconfig<uint32_t>(rtl8139_config::RCR, 0b10001111);
57     /** Set RE & TE */
58     this->setconfig<uint8_t>(rtl8139_config::CR, 0x0c);
59     this->tx_cur = 0;
60     uint32_t int_line = ((pci::header_t0 *)this->device->header_ext)->int_line;
61     kernel::idtr.add_handle(static_cast<interrupts::vector_e>(32 + int_line), interrupts::etherenet
62 }
```

La función `start()` es la encargada de poner en marcha la tarjeta de red e inicializar todos los registros necesarios para que pueda enviar y recibir paquetes.

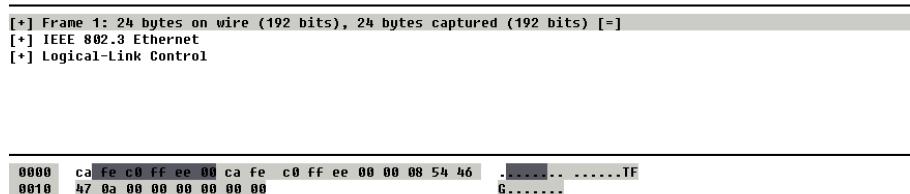
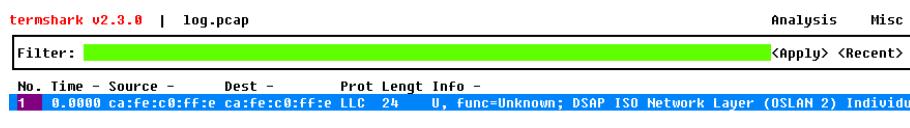
```
© Código 5.167: rtl8139.cpp – rtl8139::send_packet(uint32_t, uint64_t)

76 void
77 rtl8139::send_packet(uint32_t addr, uint64_t size)
78 {
79     this->setconfig<uint32_t>(this->TSAD_array[tx_cur], addr);
80     this->setconfig<uint32_t>(this->TSD_array[tx_cur], size);
81     tx_cur++;
82     if (this->tx_cur > 3)
83         this->tx_cur = 0;
84 }
```

Finalmente tenemos la función más importante: `send_packet(uint32_t, uint32_t)`. Esta función, como su nombre indica, es la encargada de transmitir un conjunto de datos a través de la tarjeta de red.

```
welcome to the alma kernel
$ checknet
BAR1] & 0xf..f0 : 0x00000000c1041000
pci_dev->command : 7
CF1      (0x52) : 12
CR       (0x37) : 13
BUFF     (0x30) : 0x00000000000014000
IMR      (0x3c) : 5
ISR      (0x3e) : 0
RCR      (0x44) : 143
M0:      202
M1:      254
M2:      192
M3:      255
M4:      238
M5:      0
$ sendpacket
> TFG
```

(a) Ejemplo de envío de un paquete ethernet en *alma*



(b) Registro del paquete enviado por *alma* con termshark

Figura 5.12: Funcionamiento del driver RTL8139 en *alma*

5.4.14. Interfaz

Esta sección pretende mostrar la interfaz humano – máquina que presenta *alma*. La interfaz elegida, por simplicidad, ha sido una terminal tal y como estamos acostumbrados. No sed ha elegido implementar una interfaz gráfica debido a su coste de desarrollo y porque no implica desarrollar funcionamiento “clave” del núcleo.

La interfaz desarrollada se ejecuta automáticamente al acabar de arrancar *alma*, esto se hace para poder trabajar y probar el kernel de forma más rápida. En caso de que *alma* se usase como kernel real delegaríamos este trabajo al sistema.

En esta sección desarrollaremos dos submódulos: el intérprete y los comandos. El intérprete es el encargado de, en base a una cadena introducida por el usuario, entender los componentes que la conforman y ejecutar esa orden. Los comandos son las funcionalidades que puede ejecutar el usuario, incluyendo la propia shell.

5.4.14.1. Intérprete

Para desarrollar este intérprete definimos una clase que lo represente²²:

```
Código 5.168: interpreter.h – class interpreter
13 class interpreter
14 {
15     public:
16         interpreter()
17             : commands(nullptr){};
18         interpreter(const command *input)
19             : commands(input){};
20         interpreter &operator=(interpreter &input)
21     {
22             this->commands = input.commands;
23             return *this;
24     }
25
26     char *get_buffer()
27     {
28         return this->input_buffer;
29     }
30
31     int process(char *);
32     int launch_command(int, char **);
33     static const auto BUFFER_SIZE = 256;
34
35     private:
36     /** Buffer for user input (shell input) */
37     char input_buffer[BUFFER_SIZE];
38     /** Array of commands */
39     const command *commands;
40 };
```

²²También se podría haber simplificado a una función con un array global de comandos, comunes a todos los intérpretes o similar.

Como podemos ver esta clase dispone de un array de comandos disponibles y las funciones de procesado de entrada y ejecución de comando.

```
© Código 5.169: interpreter.cpp — interpreter::process(char *)
17 int
18 interpreter::process(char *input)
19 {
20     int argc = 0;
21     char *argv[256];
22
23     uint32_t charcount = 0;
24     bool literal_flag = false;
25     for (uint64_t i = 0; true; i++) {
26         if (input[i] == '"') {
27             literal_flag = !literal_flag;
28             continue;
29         }
30
31         if (!literal_flag && (input[i] == ' ' || input[i] == '\n')) {
32             if (charcount <= 0)
33                 continue;
34
35             char *buffer = (char *)kernel::heap.malloc(sizeof(char) * (charcount + 1));
36
37             int32_t j = i - 1;
38             int32_t k = charcount;
39             buffer[k--] = '\0';
40             while (j >= 0 && k >= 0 && input[j] != ' ') {
41                 if (input[j] == '"')
42                     continue;
43                 buffer[k] = input[j];
44                 j--;
45                 k--;
46             }
47
48             argv[argc++] = buffer;
49             charcount = 0;
50             if (input[i] == ' ')
51                 continue;
52             if (input[i] == '\n')
53                 break;
54         }
55         charcount++;
56     }
57
58     if (argc <= 0)
59         return (argc != 0);
60     return this->launch_command(argc, argv);
61 }
```

`interpreter::process(char *)` es la función más importante del intérprete. Es la encargada de, a partir de una cadena de texto, ejecutar las órdenes que pide el usuario. La forma en la que el usuario solicita estas órdenes tiene un formato específico (muy similar a lo que estamos acostumbrados, como Bash).

Por ejemplo, esta función toma como argumento `echo hola que tal` y pasará a la función `launch_command` los argumentos `argc = 4` y `argv[] = echo, hola, que, tal`. La función es capaz de gestionar texto entrecomillado para agruparlo en un único argumento.

④ Código 5.170: interpreter.cpp – interpreter::launch_command(char *, int, char **)

```

70 int
71 interpreter::launch_command(char *command, int argc, char **argv)
72 {
73     for (uint32_t i = 0; this->commands[i].name != nullptr; i++) {
74         if (strcmp(this->commands[i].name, argv[0]) == 0)
75             return this->commands[i].function(argc, argv);
76     }
77     return 127;
78 }
```

Finalmente en el intérprete tendremos la función `launch_command` que será la encargada de iterar el array de comandos que tengamos y ejecutar el correspondiente.

5.4.14.2. Comandos

En esta sección se listarán los comandos que dispone el usuario para ejecutar en alma. Cabe mencionar que puede no estar actualizado con las últimas versiones de *alma* ya que es un módulo en constante cambio.

④ Código 5.171: command.h – struct command

```

11 struct command
12 {
13     const char *name;
14     int (*function)(int, char **);
15 };
```

Cada comando es un par nombre y función a ejecutar. Si *alma* fuese más avanzado y pudiese leer archivos y cargarlos en memoria podríamos sustituir la ejecución de funciones por lanzar ejecutables (con acceso a servicios del kernel o similar).

④ Código 5.172: command.h – static const command []

```

42 static const command kernel_commands[] = {
43     { "help" , &commands::help},
44     { "echo" , &commands::echo },
45     { "shell" , &commands::shell },
46     { "clear" , &commands::clear },
47     { "pci" , &commands::pci},
48     { "getpage" , &commands::getpage},
49     { "getmac" , &commands::getmac},
50     { "getphys" , &commands::getphys},
51     { "map" , &commands::map},
52     { "unmap" , &commands::unmap},
53     { "get" , &commands::get},
54     { "set" , &commands::set},
55     { "printmem" , &commands::printmem},
56     { "uefimmap" , &commands::uefimmap},
57     { "printpfa" , &commands::printpfa},
58     { "checknet" , &commands::checknet},
59     { "sendpacket" , &commands::sendpacket},
60     { "screen" , &commands::screen},
61     { "acpi" , &commands::acpi},
62     { nullptr , nullptr }};
```

El comando `help` imprime la lista de comandos disponibles, separados por una coma:

```
© Código 5.173: command.h — help(int, char **)

17 int
18 help(int argc, char **argv)
19 {
20     for (uint32_t i = 0; kernel_commands[i].name != nullptr; i++) {
21         kernel::tty.print(kernel_commands[i].name);
22         if (kernel_commands[i + 1].name != nullptr)
23             kernel::tty.print(", ");
24         else
25             kernel::tty.newline();
26     }
27
28     return 0;
29 }
```

El comando `echo` imprime el texto provisto por el usuario como argumento, soportando comillas dobles:

```
© Código 5.174: command.h — echo(int, char **)

31 int
32 echo(int argc, char **argv)
33 {
34     for (int i = 1; i < argc; i++) {
35         kernel::tty.print(argv[i]);
36         if (i + 1 < argc)
37             kernel::tty.print(" ");
38     }
39     kernel::tty.newline();
40
41     return 0;
42 }
```

El comando `shell` es el más importante, se ejecuta automáticamente al arrancar *alma*. Provee de una interfaz estilo “terminal” escaneando el texto introducido por el usuario y procesando el comando:

```
© Código 5.175: command.h — shell(int, char **)

44 int
45 shell(int argc, char **argv)
46 {
47     shell::interpreter inter(shell::kernel_commands);
48     while (true) {
49         kernel::tty.pushColor(screen::color_e::GREEN);
50         kernel::tty.print("$ ");
51         kernel::tty.popColor();
52         kernel::keyboard.scanf(inter.get_buffer(), inter.BUFFER_SIZE);
53         auto ret = inter.process(inter.get_buffer());
54         if (ret == 127) {
55             kernel::tty.pushColor(screen::color_e::RED);
56             kernel::tty.println("Command not found");
57             kernel::tty.popColor();
58         }
59     }
60     return 0;
61 }
```

El comando `clear` limpia la pantalla a negro:

```
⌚ Código 5.176: command.h - clear(int, char **)

65 int
66 clear(int argc, char **argv)
67 {
68     kernel::tty.clear();
69     return 0;
70 }
```

El comando `pci` imprime línea por línea todos los dispositivos PCI encontrados en la máquina (previamente enumerados). Imprime el identificador del fabricante y el ID del dispositivo:

```
⌚ Código 5.177: command.h - pci(int, char **)

72 int
73 pci(int argc, char **argv)
74 {
75     char buffer[256];
76     for (pci::pci_device *i = kernel::devices; i != nullptr; i = i->next) {
77         hstr(i->header->vendor, buffer);
78         kernel::tty(fmt("* %p - %p", i->header->vendor, i->header->id));
79     }
80
81     return 0;
82 }
```

El comando `getpage` solicita una página al gestor de páginas y devuelve su dirección:

```
⌚ Código 5.178: command.h - getpage(int, char **)

90 int
91 getpage(int argc, char **argv)
92 {
93     if (argc >= 2) {
94         uint64_t addr = strol(argv[1], 16);
95         auto ret = kernel::allocator.request_page((void *)addr);
96         if (ret == nullptr) {
97             kernel::tty.println("Not available");
98             return 1;
99         }
100        kernel::tty(fmt("%p", (uint64_t)ret));
101        return 0;
102    }
103
104    kernel::tty(fmt("%p", (uint64_t)kernel::allocator.request_page()));
105    return 0;
106 }
```

El comando `getmac` imprime la MAC de la tarjeta de red en formato humano:

```
⌚ Código 5.179: command.h - getmac(int, char **)

108 int
109 getmac(int argc, char **argv)
110 {
111
112     char mac_addr[32];
113     for (pci::pci_device *i = kernel::devices; i != nullptr; i = i->next) {
114         if (i->header->id == 0x8139 && i->header->header_type == 0x0) {
```

```

115     pci::header_t0 *ext_hdr = (pci::header_t0 *)i->header_ext;
116     uint64_t mmio_addr = (ext_hdr->BAR[1] & 0xfffffffff0);
117
118     char auxstr[16];
119     for (int i = 0; i < 6; i++) {
120         uint8_t aux = *(uint8_t *)mmio_addr + i;
121         hstr(aux, auxstr);
122         uint8_t endstr = strlen(auxstr);
123         mac_addr[i * 3] = auxstr[endstr - 2];
124         mac_addr[i * 3 + 1] = auxstr[endstr - 1];
125         if (i != 5)
126             mac_addr[i * 3 + 2] = ':';
127     }
128     break;
129 }
130 }
131 mac_addr[17] = '\0';
132 kernel::tty.println(mac_addr);
133
134 return 0;
135 }
```

El comando `getphys` imprime, a partir de una dirección virtual pasada por argumento, la dirección física a la que está asociada:

© Código 5.180: command.h – `getphys(int, char **)`

```

138 int
139 getphys(int argc, char **argv)
140 {
141     using namespace paging;
142     using namespace paging::translator;
143
144     if (argc <= 1) {
145         kernel::tty.fmt("Usage: %s virtaddr", argv[0]);
146         return 1;
147     }
148     uint64_t addr = strol(argv[1], 16);
149     address_t *virtaddr = (address_t *)&addr;
150     page_global_dir_entry_t *PGD = &kernel::translator.get_PGDT()[virtaddr->global];
151
152     if (!PGD->present)
153         return 1;
154
155     page_upper_dir_entry_t *PUDT = (page_upper_dir_entry_t *)((uint64_t)PGD->page_ppn << 12);
156     page_upper_dir_entry_t *PUD = &PUDT[virtaddr->upper];
157
158     if (!PUD->present)
159         return 1;
160
161     page_mid_dir_entry_t *PMDT = (page_mid_dir_entry_t *)((uint64_t)PUD->page_ppn << 12);
162     page_mid_dir_entry_t *PMD = &PMDF[virtaddr->mid];
163
164     if (!PMD->present)
165         return 1;
166
167     page_table_entry_t *PTDT = (page_table_entry_t *)((uint64_t)PMD->page_ppn << 12);
168     page_table_entry_t *PTD = &PTDT[virtaddr->table];
169     uint64_t phys = (uint64_t)PTD->page_ppn << 12;
170     kernel::tty.fmt("%p", phys);
171
172 }
```

El comando `map` toma dos argumentos (una dirección virtual y otra física) y asocia la dirección física con la virtual:

```
④ Código 5.181: command.h — map(int, char **)

178 int
179 map(int argc, char **argv)
180 {
181     if (argc <= 2) {
182         kernel::tty(fmt("Usage: %s virtaddr physaddr", argv[0]));
183         return 1;
184     }
185
186     uint64_t virt = strtol(argv[1], 16);
187     uint64_t phys = strtol(argv[2], 16);
188
189     kernel::translator.map(virt, phys);
190
191     kernel::tty(fmt("%s -> %s", argv[1], argv[2]));
192
193     return 0;
194 }
```

El comando `unmap` deshace las asociaciones de una dirección física asociandola con su misma dirección física (por ejemplo `0xff...ff` asociada a la dirección física `0xff..ff`):

```
④ Código 5.182: command.h — unmap(int, char **)

196 int
197 unmap(int argc, char **argv)
198 {
199     if (argc <= 1) {
200         kernel::tty(fmt("Usage: %s virtaddr", argv[0]));
201         return 1;
202     }
203
204     uint64_t virt = strtol(argv[1], 16);
205
206     kernel::translator.map(virt, virt);
207
208     kernel::tty(fmt("%s -> %s", argv[1], argv[1]));
209
210     return 0;
211 }
```

El comando `set` establece un byte de cierta dirección de memoria pasada como argumento a true/false dependiendo del segundo argumento:

```
④ Código 5.183: command.h — set(int, char **)

213 int
214 set(int argc, char **argv)
215 {
216     if (argc <= 2) {
217         kernel::tty(fmt("Usage: %s addr true/false", argv[0]));
218         return 1;
219     }
220
221     bool set = (strncmp(argv[2], "true", 4) == 0);
222
223     uint64_t addr = strtol(argv[1], 16);
```

```

224     bool *data = (bool *)addr;
225     *data = set;
226
227     kernel::tty.print("*(bool *)0x");
228     kernel::tty.print(argv[1]);
229     if (*data)
230         kernel::tty.println(" = true");
231     else
232         kernel::tty.println(" = false");
233
234     return 0;
235 }
```

El comando `get` lee un byte a partir de una dirección de memoria pasada como argumento:

© Código 5.184: command.h — `get(int, char **)`

```

237 int
238 get(int argc, char **argv)
239 {
240     if (argc <= 1) {
241         kernel::tty.fmt("Usage: %s addr", argv[0]);
242         return 1;
243     }
244
245     uint64_t addr = strol(argv[1], 16);
246     bool *data = (bool *)addr;
247     kernel::tty.print("*(bool *)0x");
248     kernel::tty.print(argv[1]);
249     if (*data)
250         kernel::tty.println(" -> true");
251     else
252         kernel::tty.println(" -> false");
253     return 0;
254 }
```

El comando `printmem` imprime un rango de memoria a partir de la dirección proporcionada:

© Código 5.185: command.h — `printmem(int, char **)`

```

256 int
257 printmem(int argc, char **argv)
258 {
259     /** Constants */
260     static const uint32_t BYTES_PER_LINE = 16;
261     static const uint32_t DEFAULT_LINES = kernel::page_size / BYTES_PER_LINE;
262
263     if (argc <= 1) {
264         kernel::tty.fmt("Usage: %s addr [bytes]", argv[0]);
265         return 1;
266     }
267
268     uint16_t nlines = DEFAULT_LINES;
269     if (argc == 3)
270         nlines = strol(argv[2], 10);
271
272     uint64_t addr = strol(argv[1], 16);
273     uint8_t column = 0;
274     uint8_t line = 0;
275     for (uint32_t byte = 0; byte < nlines * BYTES_PER_LINE; byte++) {
276
277         if (column == 0) {
```

```

278     char buff[32];
279     hstr((uint64_t)(uint8_t *)addr + (line * BYTES_PER_LINE), buff);
280     kernel::tty.print(buff);
281     kernel::tty.print(": ");
282 }
283
284     char buff[16];
285     uint8_t *ptr = ((uint8_t *)addr + byte);
286     hstr(*ptr, buff);
287     int wrongthing = strlen(buff); // TODO: could be improved
288     kernel::tty.put(buff[wrongthing - 2]);
289     kernel::tty.put(buff[wrongthing - 1]);
290     kernel::tty.put(' ');
291
292     column++;
293
294     if (column != 0 && column % BYTES_PER_LINE == 0) {
295         line++;
296         column = 0;
297         kernel::tty.newline();
298     }
299 }
300
301 return 0;
302 }
```

El comando `uefimmap` imprime el mapa de memoria presentado por EFI (provisto por el bootloader):

Código 5.186: command.h – uefimmap(int, char **)

```

304 int
305 uefimmap(int argc, char **argv)
306 {
307     auto *map = (stivale2_struct_tag_memmap *)stivale2_get_tag(&kernel::internal::stivalehdr,
308                                                               STIVALE2_STRUCT_TAG_MEMMAP_ID);
309
310     for (uint64_t i = 0; i < map->entries; i++) {
311         auto entry = map->memmap[i];
312         uint64_t init_addr = entry.base;
313         uint64_t fini_addr = entry.base + entry.length;
314         uint64_t kbsize = entry.length / 1024;
315         const char *memtype = nullptr;
316         switch (entry.type) {
317             case 1:
318                 memtype = "Usable ";
319                 break;
320             case 2:
321                 memtype = "Reserved ";
322                 break;
323             case 3:
324                 memtype = "ACPI Reclaimable ";
325                 break;
326             case 4:
327                 memtype = "ACPI NVS ";
328                 break;
329             case 5:
330                 memtype = "Bad Memory ";
331                 break;
332             case 0x1000:
333                 memtype = "Bootloader Reclaimable";
334                 break;
335             case 0x1001:
```

```

336         memtype = "Kernel and Modules ";
337         break;
338     case 0x1002:
339         memtype = "Framebuffer ";
340         break;
341     default:
342         memtype = "ERROR ";
343     }
344     kernel::tty(fmt("%p - %p %s [%i KB]", init_addr, fini_addr, memtype, kbsize);
345 }
346
347 return 0;
348 }
```

El comando `printpfa` imprime los nodos de la lista enlazada del gestor de páginas de memoria:

© Código 5.187: command.h – `printpfa(int, char **)`

```

350 int
351 printpfa(int argc, char **argv)
352 {
353     auto it = kernel::allocator.get_first();
354     while (it != nullptr) {
355         auto limaddr = it->addr + (it->pages * kernel::page_size);
356         kernel::tty(fmt("%p - %p [%i pages]", it->addr, limaddr, it->pages);
357         it = it->next;
358     }
359
360     return 0;
361 }
```

El comando `checknet` imprime los registros más importantes de la tarjeta de red:

© Código 5.188: command.h – `checknet(int, char **)`

```

363 int
364 checknet(int argc, char **argv)
365 {
366
367     auto hdr = kernel::rtl8139.device->header;
368     auto cmd = kernel::rtl8139.device->header->command;
369     auto addr = kernel::rtl8139.mem_addr;
370
371     auto cf1 = kernel::rtl8139.getconfig<uint8_t>(net::rtl8139_config::CONFIG1);
372     auto cr = kernel::rtl8139.getconfig<uint8_t>(net::rtl8139_config::CR);
373     auto buff = kernel::rtl8139.getconfig<uint32_t>(net::rtl8139_config::RECVBUFF);
374     auto imr = kernel::rtl8139.getconfig<uint16_t>(net::rtl8139_config::IMR);
375     auto isr = kernel::rtl8139.getconfig<uint16_t>(net::rtl8139_config::ISR);
376     auto rcr = kernel::rtl8139.getconfig<uint32_t>(net::rtl8139_config::RCR);
377
378     auto m0 = kernel::rtl8139.getconfig<uint8_t>(net::rtl8139_config::MAC0);
379     auto m1 = kernel::rtl8139.getconfig<uint8_t>(net::rtl8139_config::MAC1);
380     auto m2 = kernel::rtl8139.getconfig<uint8_t>(net::rtl8139_config::MAC2);
381     auto m3 = kernel::rtl8139.getconfig<uint8_t>(net::rtl8139_config::MAC3);
382     auto m4 = kernel::rtl8139.getconfig<uint8_t>(net::rtl8139_config::MAC4);
383     auto m5 = kernel::rtl8139.getconfig<uint8_t>(net::rtl8139_config::MAC5);
384
385     kernel::tty(fmt("BAR[1] & 0xf..f0 : %p", addr));
386     kernel::tty(fmt("pci_dev->command : %i", cmd));
387     kernel::tty(fmt("CF1 (0x52): %i", cf1));
388     kernel::tty(fmt("CR (0x37): %i", cr));
```

```

389     kernel::tty.fmt("BUFF (0x30): %p", buff);
390     kernel::tty.fmt("IMR (0x3c): %i", imr);
391     kernel::tty.fmt("ISR (0x3e): %i", isr);
392     kernel::tty.fmt("RCR: (0x44): %i", rcr);
393
394     kernel::tty.fmt("M0: %i", m0);
395     kernel::tty.fmt("M1: %i", m1);
396     kernel::tty.fmt("M2: %i", m2);
397     kernel::tty.fmt("M3: %i", m3);
398     kernel::tty.fmt("M4: %i", m4);
399     kernel::tty.fmt("M5: %i", m5);
400
401     return 0;
402 };

```

El comando `sendpacket` envía un paquete ethernet (mac `caffec0ffe00`) en base al texto introducido por el usuario:

© Código 5.189: command.h – `sendpacket(int, char **)`

```

404 int
405 sendpacket(int argc, char **argv)
406 {
407     struct ethheader
408     {
409         unsigned char dsta[6];
410         unsigned char srca[6];
411         uint16_t type;
412         unsigned char payload[10];
413     } __attribute__((packed));
414
415     auto buffer = (ethheader *)kernel::allocator.request_page();
416
417     kernel::tty.print("> ");
418     char text[256];
419     kernel::keyboard.scanf(text, 256);
420
421     buffer->dsta[0] = 0xca; buffer->dsta[1] = 0xfe; buffer->dsta[2] = 0xc0;
422     buffer->dsta[3] = 0xff; buffer->dsta[4] = 0xee; buffer->dsta[5] = 0x00;
423     buffer->srca[0] = 0xca; buffer->srca[1] = 0xfe; buffer->srca[2] = 0xc0;
424     buffer->srca[3] = 0xff; buffer->srca[4] = 0xee; buffer->srca[5] = 0x00;
425     buffer->type = 0x0800;
426     buffer->payload[0] = text[0]; buffer->payload[1] = text[1]; buffer->payload[2] = text[2];
427     buffer->payload[3] = text[3]; buffer->payload[4] = text[4]; buffer->payload[5] = text[5];
428     buffer->payload[6] = text[6]; buffer->payload[7] = text[7]; buffer->payload[8] = text[8];
429     buffer->payload[9] = text[9];
430
431     kernel::rtl18139.send_packet((uint64_t)buffer, sizeof(ethheader));
432     return 0;
433 }

```

El comando `screen` imprime la resolución actual de la pantalla:

© Código 5.190: command.h – `screen(int, char **)`

```

454 int
455 screen(int argc, char **argv)
456 {
457     kernel::tty.fmt("%ix%i", kernel::tty.get_width(), kernel::tty.get_height());
458     return 0;
459 }

```

El comando `acpi` imprime las tablas ACPI encontradas:

```
461 int
462 acpi(int argc, char **argv)
463 {
464     kernel::rsdp.print_acpi_tables();
465     return 0;
466 }
```

Podemos ver el ejemplo de la ejecución de los comandos `getpage`, `get`, `set`, `pci` y `acpi` en la Figura 5.13. Cabe mencionar que la terminal soporta scroll cuando no cabe más texto en la pantalla.

```
welcome to the alma kernel
$ help
help, echo, shell, clear, pci, getpage, getmac, getphys, map, unmap, get, set, printmem, uefimmap, p
rintpfa, checknet, sendpacket, screen, acpi
$ getpage
0x0000000000001d000
$ getpage
0x0000000000001e000
$ map 1D000 1E000
1D000 -> 1E000
$ get 1E000
*(bool *)0x1E000 -> false
$ set 1D000 true
*(bool *)0x1D000 = true
$ get 1E000
*(bool *)0x1E000 -> true
$ pci
* 0x00000000000008086 - 0x000000000000029c0
* 0x0000000000001234 - 0x00000000000001111
* 0x00000000000010ec - 0x00000000000008139
* 0x00000000000008086 - 0x00000000000002918
* 0x00000000000008086 - 0x00000000000002922
* 0x00000000000008086 - 0x00000000000002930
$ acpi
FACP
APIC
HPET
MCFG
WAET
BGRT
$
```

Figura 5.13: Ejemplo de ejecución de comandos en *alma*

5.4.15. Librerías

5.5. Conclusiones

6. Conclusiones

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] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer’s Manual*, December 2021, no. 325462-O76US.
- [12] T. U. E. Forum, *Advanced Configuration and Power Interface (ACPI) Specification*, 1 2019.
- [13] “GitHub - stivale/stivale: The stivale boot protocols’ specifications and headers. — github.com,” <https://github.com/stivale/stivale>, [Accessed 27-Mar-2022].
- [14] “QEMU,” Software, A generic and open source machine emulator and virtualizer. [Online]. Available: <https://www.qemu.org/>

- [15] “ccache,” Software, A compiler cache. [Online]. Available: <https://ccache.dev/>
 - [16] “OSDev Target Triplet,” Wiki. [Online]. Available: https://wiki.osdev.org/Target_Triplet
 - [17] P. Oppermann.
 - [18] “System V ABI,” Wiki. [Online]. Available: https://wiki.osdev.org/System_V_ABI#x86-64
 - [19] “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>
 - [20] “Graphics Output Protocol,” Wiki. [Online]. Available: <https://wiki.osdev.org/GOP>
 - [21] 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].
 - [22] “VESA,” Wiki. [Online]. Available: <https://wiki.osdev.org/VESA>
 - [23] “Framebuffer - Wikipedia — en.wikipedia.org,” <https://en.wikipedia.org/wiki/Framebuffer>, [Accessed 24-Mar-2022].
 - [24] “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].
 - [25] “PC Screen Font - Wikipedia — en.wikipedia.org,” https://en.wikipedia.org/wiki/PC_Screen_Font, [Accessed 24-Mar-2022].
 - [26] 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
 - [27] “RSDT - OSDev Wiki — wiki.osdev.org,” <https://wiki.osdev.org/RSDT>, [Accessed 24-Mar-2022].
 - [28] “XSDT - OSDev Wiki — wiki.osdev.org,” <https://wiki.osdev.org/XSDT>, [Accessed 24-Mar-2022].
 - [29] “RSDP - OSDev Wiki — wiki.osdev.org,” <https://wiki.osdev.org/RSDP>, [Accessed 24-Mar-2022].
 - [30] “Executable and Linkable Format - Wikipedia — en.wikipedia.org,” https://en.wikipedia.org/wiki/Executable_and_Linkable_Format, [Accessed 24-Mar-2022].
 - [31] T. I. Standards, “Executable and linkable format (elf),” *Specification, Unix System Laboratories*, 2001. [Online]. Available: <http://refspecs.linuxbase.org/elf/elf.pdf>
-

- [32] D. Andriesse, *Practical binary analysis : build your own Linux tools for binary instrumentation, analysis, and disassembly.* San Francisco, CA: No Starch Press, Inc, 2019.
- [33] pancake, “radare2,” <https://rada.re/n/>.
- [34] A. Tanenbaum, *Modern operating systems.* Boston: Pearson, 2015.
- [35] “cmake,” Software, Family of tools designed to build, test and package software. [Online]. Available: <https://cmake.org/>
- [36] “GDT Tutorial - OSDev Wiki — wiki.osdev.org,” https://wiki.osdev.org/GDT_Tutorial, [Accessed 24-Apr-2022].
- [37] “Basic x86 interrupts | There is no magic here — alex.dzyoba.com,” <https://alex.dzyoba.com/blog/os-interrupts/>, [Accessed 25-Apr-2022].
- [38] D. Anderson, T. Shanley, M. Inc, and I. MindShare, *PCI System Architecture*, 05 1999.
- [39] Realtek, *RTL8139C(L) + Datasheet*, 9 2004.
- [40] “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/>
- [41] “OSDev libgcc,” Wiki. [Online]. Available: <https://wiki.osdev.org/Libgcc>
- [42] Realtek, *RTL8139(A/B) Programming Guide*, 1 1999.

A. Anexo I