Assembly-Level Implementation of a Minimalist UEFI Bootloader for Custom Operating Systems

Myint Myat Aung
University of Information Technology
Yangon, Myanmar

***Abstract*—This paper presents the implementation of a minimalist UEFI x64 bootloader and kernel entirely in assembly using the FASM assembler. The implementation is unique in that both the bootloader and kernel executables are created from simple flat binary output from the assembler. The generation of PE32+ and ELF64 executables and the interfaces required to interact with UEFI hardware are implemented from scratch by referring to their ABI specification documents without relying on libraries. This technique eliminates toolchain dependencies, grants full control to the system architect, and facilitates deeper understanding of internal mechanisms. The work serves as both a foundation for the future development of a modern operating system free from legacy design decisions and an educational resource for understanding the esoteric nature of operating systems development.**

***Index Terms*—uefi, bootloader, assembly, operating systems, low-level programming.**

I. Introduction

Building an operating system from scratch remains one of the most effective ways to understand how computers truly work. This paper documents the development of a minimal UEFI bootloader and kernel written entirely in x64 assembly, where every byte of the executable files is crafted by hand. Unlike typical OS projects that rely on compilers and linkers, this implementation constructs both the bootloader (PE32+ format) and kernel (ELF64 format) through direct assembly programming. Doing this offers complete visibility into the system's lowest levels.

Modern computers begin their startup sequence through a complex interaction between firmware and software. While most developers interact with this process through high-level tools, there's unique value in controlling every aspect:

- For learners: Manually creating executable files reveals how programs are structured at the binary level.
- For tinkerers: Handwriting the boot process enables custom behaviors impossible with standard toolchains.
- For future work: This minimal foundation can grow into a full OS without inheriting unnecessary abstractions.

The implementation consists of two core components:

A. A UEFI bootloader

The UEFI bootloader performs the following:

- Prepares the system by calling UEFI functions like ExitBootServices()
- Loads the kernel into memory by parsing its ELF64 headers manually
- Passes critical hardware information (like the framebuffer) to the kernel

B. A demonstration kernel

The demonstration kernel performs the following:

- Takes control from the bootloader
- Renders "HELLOWORLD" to the screen by directly manipulating pixels
- Uses custom bitmap fonts defined in the assembly code

II. Background

A. UEFI Overview

The Unified Extensible Firmware Interface (UEFI) has largely replaced the legacy BIOS as the standard firmware interface for modern computing systems. UEFI provides a more flexible and modular environment for system initialization, offering features such as secure boot, network booting, and support for larger storage devices. Unlike BIOS, which operates in 16-bit real mode, UEFI operates in 32-bit or 64-bit protected mode, enabling direct access to modern hardware capabilities. Bootloaders in UEFI systems are typically implemented as PE32+ executables, which are loaded and executed by the firmware during the boot process. This shift from BIOS to UEFI has introduced new challenges and opportunities in bootloader development, particularly in terms of compatibility. The UEFI boot process is described in Fig. 1. The bootloader to be implemented will serve the purpose of the OS Loader as mentioned in the figure, after the UEFI has loaded all hardware-related devices. This is how UEFI achieves machine-independence.

B. PE32+ Binary Format

The PE32+ (Portable Executable) format is a file format used for executables, object code, and DLLs in Windows and UEFI environments. It includes headers such as the DOS header, PE header, and section headers, which define the executable's structure, entry point, and memory layout. The PE32+ format is essential for UEFI bootloaders, as it ensures compatibility with the firmware's

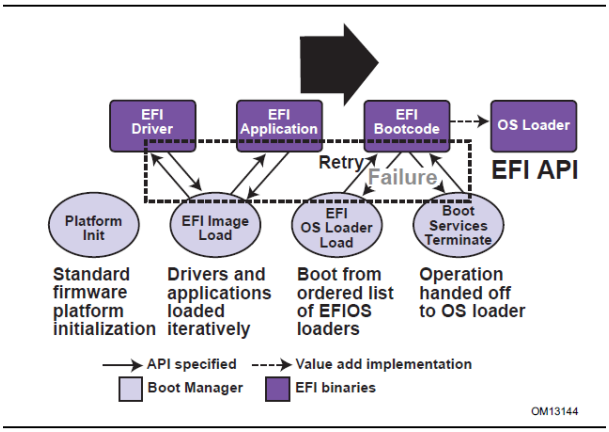

Fig. 1. Typical boot process of a UEFI bootloader

loading mechanism. Similarly, the ELF64 (Executable and Linkable Format) format is widely used for executables in Unix-like operating systems. It includes headers such as the ELF header, program headers, and section headers, which define the executable's memory layout and entry point. The format is further described in Design and Implementation.

C. ELF64 Binary Format

The ELF64 (Executable and Linkable Format) serves as the standard binary format for 64-bit operating systems and applications [1], particularly on AMD64/x86-64 architectures. Defined in the System V ABI specification [2], this format organizes executable code and data into three primary components: (1) the ELF header, which identifies the file type (using the magic sequence 0x7F+'ELF'), specifies the target architecture (0x3E for AMD64), and declares the entry point address where execution begins; (2) program headers that define memory segments with specific permissions (read/write/execute) and loading addresses; and (3) optional section headers for linking and debugging. For AMD64 processors, the format enforces critical architectural requirements including canonical addressing (where upper address bits must be properly sign-extended) and 16-byte stack alignment for function calls. In this project, the kernel's ELF64 structure is manually constructed in assembly, featuring a minimal setup with just two loadable segments: one for executable code (marked with read/execute permissions) loaded at 0x400000, and another for data at 0x600000. This approach provides several advantages over more complex formats like PE32+: it eliminates legacy compatibility fields, allows control over memory layout through straightforward program headers, and maintains portability across Unix-like systems while being simple enough to implement directly in assembly.

III. Related Work

There exists other bootloaders that could have been used instead of manually developing one.

- GNU GRUB bootloader is by far the most widely used bootloader in the Linux ecosystem. [3] It has the ability to also load Windows and other OSes and boot manager. While vastly compatible with almost all hardware and software, it is a large and complex bootloader, holding much legacy code and code for backward compatibility. Multiboot 2 is the boot protocol for GRUB [4], which is used for passing boot-time obtained data to the kernel, but is equally large albeit featureful.
- The Limine Bootloader is a modern, advanced, and multiprotocol bootloader. [5] It supports many architectures including 32-bit x86, x86-64 and arm64 and also supports Multiboot 2. It makes use of the Limine protocol. [6] This has is origins in hobby OS development but later grew to become usable for other users.
- BOOTBOOT bootloader was created to be minimalistic. [7] It's protocol is the BOOTBOOT Protocol [8]. Being minimalistic, it does not include features like Secure Boot and dynamic configuration.

The bootloader developed is most similar to the BOOTBOOT bootloader, but only focusing on modern x86-64 architectures running on UEFI. This allows it to have a simpler architecture and a solid foundation upon which a modern Operating System can be built without unnecessary legacy compatibility.

IV. Design and Implementation

The implementation consists of 5 files:

- bootloader.asm: Main bootloader FASM source file containing flat binary which will output in the UEFI-compatible PE32+ format, specified according to Microsoft's PE32+ specification [9]
- kernel.asm: Source FASM file of the demo kernel containing flat binary which will output in the ELF64 format, specified according to ELF64 specification [1]. The rationale behind choosing ELF64 is due to being widely used in the Linux, being field-tested, and the relatively simple structure of the binary.
- bootloaderstruct.inc: Custom defined structures to use when passing information about the system environment that the bootloader gathered to the kernel. Currently only supports passing graphics information. A standard can be defined later.
- elf.inc: Manually defined ELF64 structures so that the bootloader can parse them. Defined according to ELF [1] and AMD64 processor-specific [2] specification documents
- efi.inc: Manually defined EFI structures for interacting with UEFI firmware to perform tasks like getting memory map, printing to screen, allocating disk

space, etc. Defined according to UEFI specifications [10].

UEFI-specific structures, such as the System Table, were defined based on the UEFI specification. These structures facilitated interaction with UEFI services, including getting memory map, printing to the screen, memory allocation and boot services termination.

The process is as follows:

- The bootloader is loaded into memory as an UEFI application by the UEFI firmware of the machine that it is running on (whether it be virtual or physical).
- The bootloader outputs to the screen its status using the defined UEFI functions from the system table.
- The bootloader loads the kernel file in the file system into memory
- The bootloader parses the kernel file according to the ELF64 format, discovering information about the size of the kernel, etc.
- The bootloader gets the memory map of the system used to change execution environment.
- The bootloader passes arguments using bootloaderstruct.inc, passes execution control to the kernel and jumps to the kernel entry point.
- The kernel code begins execution, performing boot up of the operating system.

The following snippet 1 describes the System Table and Table header structures, amongst others, as defined in the UEFI specs. They are the main functions the bootloader must use to load a kernel.

The flat binary output is produced using code such as in 2, where assembly "define byte" and family directives are used.

UEFI function calls are performed using Microsoft x64 calling convention in assembly as defined in the specifications [11].

The process of loading the kernel file into memory is described in 3. This involves using the UEFI OpenVolume function, a part of BootServices which give full functionality for OS loaders.

The loaded exceutable is parsed using the ELF structure as in 4.

5 marks the important execution transfer point, where the bootloader gathers memory map information, loads it up as arguments to the kernel and passes execution to the kernel. Arguments are passed using the bootloaderstruct structure defined in 6.

7 is where the kernel obtains the information passed to it. and saves the arguments.

This completes the bootloading and kernel execution process.

V. Results

For the purposes of testing, the bootloader and kernel can be copied to into an EFI-compatible image and then transformed into a bootable ISO. The Makefile displayed in 8 is used to automate the process of:

```
struc EFI_TABLE_HEADER
  label .
  .Signature  UINT64
  .Revision   UINT32
  .HeaderSize UINT32
  .CRC32      UINT32
  .Reserved   UINT32
end struc

virtual at 0
  EFI_TABLE_HEADER EFI_TABLE_HEADER
end virtual

struc EFI_SYSTEM_TABLE
  label .
  .Hdr                  EFI_TABLE_HEADER
  .FirmwareVendor       UINTN
  .FirmwareRevision     UINT32
  .ConsoleInHandle      UINTN
  .ConIn                UINTN
  .ConsoleOutHandle     UINTN
  .ConOut               UINTN
  .StandardErrorHandle  UINTN
  .StdErr               UINTN
  .RuntimeServices      UINTN
  .BootServices         UINTN
  .NumberOfTableEntries UINTN
  .ConfigurationTable   UINTN
end struc

virtual at 0
  EFI_TABLE_HEADER EFI_TABLE_HEADER
end virtual
```

Listing 1: EFI system table and table header structures required to interface with UEFI, amongst other structures

- Assembling the bootloader and kernel source
- Creating an empty EFI compatible image
- Creating an EFI compatible ISO from said image using xorriso
- Creating and running the virtual machine with said ISO as the medium

When run using the Linux QEMU emulator, the virtual machine boots up and successfully renders the letters 'HELLOWORLD', as can be seen in 2.

VI. Discussion

The implementation of this UEFI bootloader represents a significant milestone in the ongoing effort to develop a custom operating system. [1]

The reason for the development of the bootloader implementation as described in this paper is to serve as the foundation for a modern operating system which is

```
ehdr:
.EI_MAG0          db 0x7f
.EI_MAG1          db 'E'
.EI_MAG2          db 'L'
.EI_MAG3          db 'F'
.EI_CLASS         db 2
.EI_DATA          db 1
.EI_VERSION       db 1
.EI_OSABI         db 0
.EI_ABIVERSION    db 0
.EI_PAD           rb 6
.EI_NIDENT        db 16

.e_type          dw 2
.e_machine       dw 62
.e_version       dd 1
.e_entry         dq main
.e_phoff         dq phdr - BASE
.e_shoff         dq 0
.e_flags         dd 0
.e_ehsize        dw phdr - ehdr
.e_phentsize     dw endphdr1 - phdr
.e_phnum         dw 2
.e_shentsize     dw 0
.e_shnum         dw 0
.e_shstrndx      dw 0
```

Listing 2: Kernel ELF EHeader defined using define byte primitives according to ELF specification

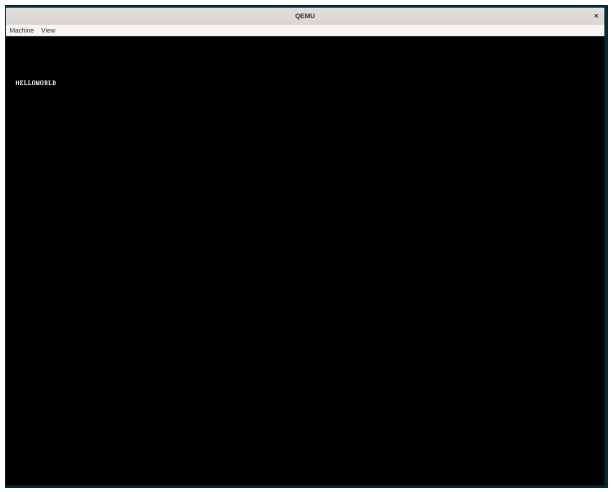

Fig. 2. Bootloader successfully loading the demo kernel inside QEMU

minimal in nature and does not contain ”legacy baggage”. Legacy baggage consists of (but is not limited to) the following:

- Outdated Design Decisions
- Accumulation of legacy code
- The need to maintain backward compatibility

```
mov rax, [SystemTable]
mov rax, [rax+EFI_SYSTEM_TABLE.BootServices]
mov rcx, [ImageHandle]
lea rdx, [efiLoadedImageProtocolGuid]
lea r8, [loadedImage]
call [rax+EFI_BOOT_SERVICES.HandleProtocol]

mov rax, [SystemTable]
mov rax, [rax+EFI_SYSTEM_TABLE.BootServices]
mov rcx, [loadedImage]
mov rcx, [rcx+EFI_LOADED_IMAGE_PROTOCOL.Devic
↪  eHandle]
lea rdx, [efiSimpleFileSystemProtocolGuid]
lea r8, [bootVolume]
call [rax+EFI_BOOT_SERVICES.HandleProtocol]


mov rax, [bootVolume]
mov rcx, [bootVolume]
lea rdx, [rootFile]
call [rax+EFI_SIMPLE_FILE_SYSTEM_PROTOCOL.Ope
↪  nVolume]

mov rax, [rootFile]
mov rcx, [rootFile]
mov rdx, kernelFile
mov r8, fileName
mov r9, EFI_FILE_MODE_READ
mov r10, 0
mov [rsp + 4*8], r10
call [rax+EFI_FILE_PROTOCOL.Open]

mov rax, [kernelFile]
mov rcx, [kernelFile]
mov rdx, readSize
mov r8, KernelAddress
call [rax+EFI_FILE_PROTOCOL.Read]

mov rdx, [KernelAddress+Elf64_Ehdr.e_entry]
mov [KernelEntryPoint], rdx
```

Listing 3: Discovering the volume in which the kernel file is situated and loading it into memory

A. Outdated Design Decisions

Two of the most widely used operating systems currently, Windows and Linux-based distributions, were first released in 1991 and 1985 respectively. The design decisions made during the development of these OSes were based on outdated assumptions. An example is UNIX (which Linux is based on) was initially created for allowing multiple users to simultaneously use a single large mainframe via terminals. For this, UNIX offers multiuser sessions and terminal multiplexing. Linux also sets these

```
struc Elf64_Ehdr
  label .
  .e_ident      rb 16
  .e_type       Elf64_Half
  .e_machine    Elf64_Half
  .e_version    Elf64_Word
  .e_entry      Elf64_Addr
  .e_phoff      Elf64_Off
  .e_shoff      Elf64_Off
  .e_flags      Elf64_Word
  .e_ehsize     Elf64_Half
  .e_phentsize  Elf64_Half
  .e_phnum      Elf64_Half
  .e_shentsize  Elf64_Half
  .e_shnum      Elf64_Half
  .e_shstrndx   Elf64_Half
end struc

virtual at 0
  Elf64_Ehdr Elf64_Ehdr
end virtual

struc Elf64_Phdr
  label .
  .p_type     Elf64_Word
  .p_flags    Elf64_Word
  .p_offset   Elf64_Off
  .p_vaddr    Elf64_Addr
  .p_paddr    Elf64_Addr
  .p_filesz   Elf64_Xword
  .p_memsz    Elf64_Xword
  .p_align    Elf64_Xword
end struc

virtual at 0
  Elf64_Phdr Elf64_Phdr
end virtual
```

Listing 4: ELF64 EHeader and Program Header structures, amongst other structures

virtual terminals up when the computer is booted up, and exposes multiple tty devices. In the modern day, desktop OSes are used mostly by a single user. It would be desirable to simplify this by only using one terminal, or even wholly eliminating the concept of the terminal (terminal as in the device and not the command-line interface). Doing so would bring about a small performance gain but most importantly make the OS's architecture simpler, allowing codebase maintenance to be easier.

B. Accumulation of legacy code

This is the issue of old code accumulating over the course of decades of development. This can amount to millions of lines of code, degrading performance and in-

```
mov rax, [SystemTable]
mov rax, [rax+EFI_SYSTEM_TABLE.BootServices]
mov rcx, MemoryMapSize
mov rdx, MemoryMap
mov r8, MapKey
mov r9, DescriptorSize
mov r10, DescriptorVersion
mov [rsp + 4*8], r10
call [rax+EFI_BOOT_SERVICES.GetMemoryMap]

; Off we go!
mov rax, [SystemTable]
mov rax, [rax+EFI_SYSTEM_TABLE.BootServices]
mov rcx, [ImageHandle]
mov rdx, [MapKey]
call [rax+EFI_BOOT_SERVICES.ExitBootServices]

;mov rcx, [FramebufferAddress]
;mov rdx, [PixelsPerScanLine]
;mov rcx, FramebufferAddress
mov rcx, bootloaderStruct
call [KernelEntryPoint]
```

Listing 5: The entry point: Getting memory map, exiting boot services and passing arguements and passing execution from bootloader to kernel

```
struc BOOTLOADER_STRUCT
  label .
  .framebufferAddress  rq 1
  .pixelsPerScanLine   rq 1
end struc

virtual at 0
  BOOTLOADER_STRUCT BOOTLOADER_STRUCT
end virtual
```

Listing 6: Custom bootloader structure used to pass information from bootloader to kernel

```
startcode:
main:

sub rsp, 4*8

mov r9,
↪  [rcx+BOOTLOADER_STRUCT.framebufferAddress]
mov [FramebufferAddress], r9
mov r9,
↪  [rcx+BOOTLOADER_STRUCT.pixelsPerScanLine]
mov [PixelsPerScanLine], r9
```

Listing 7: Receiving kernel arguments from the bootloader using bootloaderstruct

```
all: bootableiso testqemu

bootableiso: efi.img tkernel tbootloader
        mcopy -o -i efi.img bootloader.efi
        ↪  ::/EFI/BOOT/BOOTX64.efi
        mcopy -o -i efi.img kernel ::/kernel
        mkdir -p isobuild
        cp efi.img isobuild
        xorriso -as mkisofs -R -f -e efi.img
        ↪  -no-emul-boot -o bootable.iso
        ↪  isobuild

efi.img:
        dd if=/dev/zero of=efi.img bs=1k
        ↪  count=1440
        mkfs.msdos -F 12 efi.img
        mmd -i efi.img ::/EFI
        mmd -i efi.img ::/EFI/BOOT

tkernel:
        ./fasm2/fasm2 -n kernel.asm kernel

tbootloader:
        ./fasm2/fasm2 -n bootloader.asm
        ↪  bootloader.efi

testqemu:
        cp OVMF_VARS_ORIGINAL.fd OVMF_VARS.fd
        qemu-system-x86_64 -drive
        ↪  if=pflash,format=raw,readonly=on,
        ↪  file=OVMF_CODE.fd -drive
        ↪  if=pflash,format=raw,file=OVMF_VA
        ↪  RS.fd -cdrom
        ↪  bootable.iso

clean:
        rm -rf kernel bootloader.efi efi.img
        ↪  OVMF_VARS.fd isobuild OVMF_VARS.fd
```

Listing 8: Makefile used to create and run the virtual machine

creasing the rate of failures, a phenomenon often referred to as Software aging. Linux is not an exception and analysis of the Linux kernel's aging codebase has been carried out by others. [12], [13] The sheer size of these codebases require a huge undertaking to refactor.

C. The need to maintain backward compatibility

This refers to the maintaining of code solely due to support old hardware or software technologies. e.g UEFI is replacing BIOS in almost all modern computers. Both Windows and Linux distributions need to support old BIOS plaforms in order that they keep running on legacy hardware. Another example is 32-bit x86 code. x86-64 (the 64-bit architecture) has superceded 32-bit x86, which could only support 2 GB of RAM. An OS focused on modern hardware need not maintain backward compatibility with BIOS and 32-bit architecture. A side effect in addition to simplifying architecture is that security is vastly improved as there is a smaller attack surface.

A possible future operating system could be a multi-kernel approach like the Barrelfish OS described by Baumann et al. in [14]. This kind of OS allows massive parallelization and scalability as it is distributed in the lowest kernel level instead of using a middleware to achieve distributed properties. Future work may be based on such a kernel.

VII. Conclusion

This paper presented the implementation of a minimalist UEFI bootloader developed entirely in assembly. The bootloader UEFI (PE32+) application successfully loads a custom ELF64 kernel that displays 'HELLOWORLD'. The system implements the UEFI, System V ELF ABI and PE32+ executable standards to build interfaces that allow communication with the hardware, eliminating the use of external libraries and thus keeping the source minimal and easy-to-understand.

The project represents a critical first step in the broader effort to develop a custom modern operating system, free from legacy architecture and unnecessary backward compatibilities. Future work will build on this foundation, extending the bootloader and integrating it with other components of the operating system. The minimalistic nature of the implementation allows it to be used as an education resource in Operating Systems curriculums to offer clear insight into the practical, low-level workings of a bootloaders's mechanisms.

The implementation is available on Github at [15].

Acknowledgment

The author expresses deepest gratitude to all current and past Professors of the High-Performance Computing major at the University of Information Technology who have, over the course of two academic years, shared their knowledge and offered their exceptional guidance and encouragement. Their profound ability to deliver concepts during lectures demystified complex concepts and laid the foundation for this work. Special thanks to the Department of Computer Science and the institute for providing the environment and facilities that made this paper possible.

References

[1] *System V Application Binary Interface*, Intel Std., Rev. Latest Draft, 2013. [Online]. Available: https://www.sco.com/developers/gabi/

[2] M. Matz, J. Hubička, A. Jaeger, and M. Mitchell, *AMD64 Architecture Processor Supplement*, System V Application Binary Interface, Rev. 0.99.6, Jul. 2, 2012. [Online]. Available: https://refspecs.linuxfoundation.org/elf/x86_64-abi-0.99.pdf

[3] Gnu grub. Free Software Foundation, Inc. [Online]. Available: https://www.gnu.org/software/grub/

[4] *Multiboot2 Specification*, Free Software Foundation, Inc. Std., 2016. [Online]. Available: https://www.gnu.org/software/grub/manual/multiboot2/multiboot.html
[5] mintsuki. The limine boot loader. [Online]. Available: https://github.com/limine-bootloader/limine
[6] ——. The limine boot protocol. [Online]. Available: https://github.com/limine-bootloader/limine/blob/trunk/PROTOCOL.md
[7] B. Z. Tamás, *The BOOTBOOT Bootloader*, Std. [Online]. Available: https://gitlab.com/bztsrc/bootboot
[8] ——, *The BOOTBOOT Protocol Specification and Manual*, Std., Rev. 1.0, 2017-2021. [Online]. Available: https://gitlab.com/bztsrc/bootboot/-/raw/master/bootboot_spec_1st_ed.pdf
[9] *Microsoft Portable Executable and Common Object File Format Specification*, Microsoft Corporation Std., Rev. 6.0, Feb. 1999. [Online]. Available: http://www.osdever.net/documents/PECOFF.pdf
[10] *Unified Extensible Firmware Interface (UEFI) Specification*, UEFI Forum, Inc. Std., Rev. 2.11, Nov. 2024. [Online]. Available: https://uefi.org/sites/default/files/resources/UEFI_Spec_Final_2.11.pdf
[11] Microsoft. (2025, Mar.) x64 calling convention. [Online]. Available: https://learn.microsoft.com/en-us/cpp/build/x64-calling-convention
[12] D. Cotroneo, R. Natella, R. Pietrantuono, and S. Russo, "Software aging analysis of the Linux Operating System," in *2010 IEEE 21st International Symposium on Software Reliability Engineering*, 2010, pp. 71–80.
[13] H. Xu and R. Tang, "Study and improvements for the real-time performance of linux kernel," in *2010 3rd International Conference on Biomedical Engineering and Informatics*, vol. 7, 2010, pp. 2766–2769.
[14] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The multikernel: a new OS architecture for scalable multicore systems," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 29–44. [Online]. Available: https://doi.org/10.1145/1629575.1629579
[15] M. M. Aung. Test implementation of a uefi x64 bootloader. [Online]. Available: https://github.com/posalusa24/test-uefi-bootloader