An ABI-Level Approach: Implementing a UEFI Bootloader and Kernel from First Principles

Myint Myat Aung
University of Information Technology
Yangon, Myanmar
myintmyataung@uit.edu.mm

Aung Nway Oo
Faculty of Computer Science
University of Information Technology
Yangon, Myanmar
aungnwayoo@uit.edu.mm

Win Win Thant
Faculty of Computer Science
University of Information Technology
Yangon, Myanmar
winwinthant@uit.edu.mm

Thinn Thinn Wai
Faculty of Computer Science
University of Information Technology
Yangon, Myanmar
thinnthinnwai@uit.edu.mm

2025 6th International Conference on Advanced Information Technologies (ICAIT) | 979-8-3315-7272-3/25/$31.00 ©2025 IEEE | DOI: 10.1109/ICAIT68809.2025.11236747

***Abstract*—This paper presents the implementation of a minimalist UEFI x64 bootloader and kernel entirely in FASM assembly. The bootloader loads a simple kernel that displays "HELLOWORLD". The implementation is unique in that both the bootloader and kernel executables are created from simple flat binary output of 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 a 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.**

Keywords—UEFI, bootloader, assembly, operating systems, application binary interface

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 using a "first principles" approach where every byte of the executable files is crafted by hand. Unlike typical OS implementations that rely on compilers and linkers, this implementation constructs both the bootloader (PE32+ format) and kernel (ELF64 format) through direct assembly programming. This approach minimizes dependencies and reduces code complexity.

The implementation consists of two core components:

1. A UEFI bootloader that prepares the system by calling UEFI functions, loads the kernel into memory by parsing its ELF64 headers manually, and passes hardware information (like the framebuffer) to the kernel
2. A demonstration kernel that takes control from the bootloader and render "HELLOWORLD" to the screen by directly manipulating pixel by using custom bitmap fonts defined in the assembly code

The following sections will explain in detail the related work, background concepts, implementation of the bootloader and kernel, the results of the implementation with a comparative analysis of the bootloader, and a discussion of bootloader and OS development focusing on legacy code and code maintainability.

II. Related Work

There exist other bootloaders that could have been used instead of manually developing ones.

- GNU GRUB bootloader is by far the most widely used bootloader in the Linux ecosystem. 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 [1]. Multiboot 2 is the boot protocol for GRUB, which is used for passing boot-time obtained data to the kernel, but is equally large albeit featureful [2].
- The Limine Bootloader is a modern, advanced, and multiprotocol bootloader. It supports many architectures including 32-bit x86, x86-64 and arm64 and also supports Multiboot 2 [3]. It makes use of the Limine protocol [4]. This has its origins in hobby OS development but later grew to become usable for other users.
- BOOTBOOT bootloader was created to be minimalistic [5]. Its protocol is the BOOTBOOT Protocol [6]. Being minimalistic, it does not include features like Secure Boot and dynamic configuration.

However, in accordance with a first-principles approach and a focus on minimizing legacy support, we have decided to create a new bootloader from scratch solely for modern x86-64 architectures running on UEFI.

III. 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

979-8-3315-7272-3/25/$31.00 ©2025 IEEE

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. The UEFI boot process is described in Figure 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.

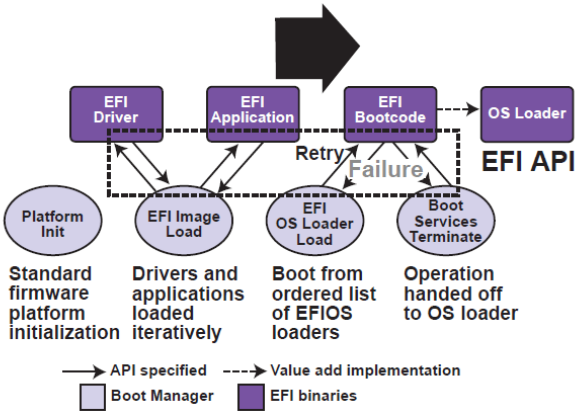


Fig. 1. Typical Boot Process of a UEFI Bootloader

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 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 the Design and Implementation section.

C. *ELF64 Binary Format*

The ELF64 (Executable and Linkable Format) serves as the standard binary format for 64-bit operating systems and applications [7], particularly on AMD64/x86-64 architectures. Defined in the System V ABI specification [7] and its AMD64-specific supplement [8], this format organizes executable code and data into three primary components: (1) the ELF header, which identifies the file type (ELF), specifies the target architecture (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. In this implementation, 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.

IV. Design and Implementation

The implementation consists of five files, all written in FASM assembly. The ".inc" files are "include" files that contain structures, constants and data types common to the ".asm" files. It is crucial to note that these "struct" structures are simply definitions and are not part of the actual bytes generated into the final binary executable. The two ".asm" files are the ones which will generate the corresponding bytes that wholly make up the final binary.

- 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 the ELF64 specification [7]. The rationale behind choosing ELF64 is due to being widely used in 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 custom standard that supports more information can be defined in the future.
- elf.inc: Manually defined ELF64 structures so that the bootloader can parse them. Defined according to ELF [7] and AMD64 processor-specific [8] specification documents
- efi.inc: Manually defined EFI structures for interacting with UEFI firmware to perform tasks like getting a memory map, loading the kernel binary from the file system, printing to the screen, etc. Defined according to UEFI specifications [10].

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

The full execution process flowchart of the bootloader and the kernel can be seen in Figure 2.

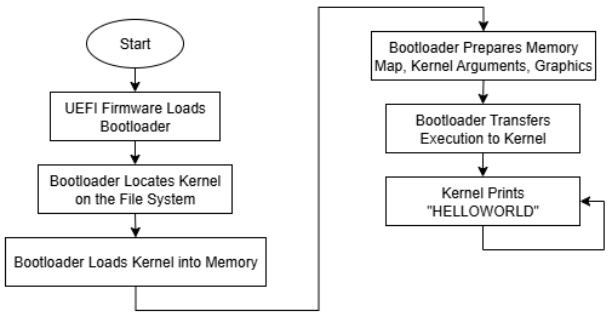


Fig. 2. Execution Process Flowchart of the Bootloader and Kernel

The explanation of the boot process is as follows:

1. The bootloader binary is loaded into memory as a UEFI application by the UEFI firmware of the machine that it is running on (whether it be virtual or physical).

2. The bootloader outputs to the screen its status using the defined UEFI functions from the system table.

3. The bootloader locates the kernel file on the file system.

4. The bootloader loads the kernel file into memory.

5. The bootloader parses the kernel file according to the ELF64 format, discovering information about the size of the kernel, etc.

6. The bootloader prepares graphics structures and gets the memory map of the system, which is then used to exit the UEFI environment and gain complete control over memory.

7. The bootloader passes arguments and environment information to the kernel using the structure in bootloaderstruct.inc, jumps to the kernel entry point.

8. The kernel code begins execution, performing the boot up of the operating system.

A. Explanation of the "efi.inc" file

The bootloader begins running in the UEFI environment. All actions that interact with the hardware are performed via accessing UEFI's data structures and calling UEFI's functions. Specific UEFI structures that contain functions exist in specific memory addresses, which are defined in the UEFI standard [10]. It is therefore first necessary to define the addresses of these structures and the offsets of their function addresses or pointers. Each function pointer has a size such as whether it is 32 bits long or 64 bits long.

Below is a snippet from the efi.inc file which defines the main data structure that contains UEFI's functions: The EFI system table. The table contains a pointer to a table header structure as its first item, followed by pointers to various services. In this way, the rest of the necessary UEFI structures are defined.

The basic datatypes used by UEFI as laid out in the UEFI specification [10] are:

- UINTN: A native (64-bit) unsigned integer, usually a memory address to a function or structure
- UINT64: A 64-bit unsigned integer for data
- UINT32: A 32-bit unsigned integer for data

These datatypes are defined in efi.inc as well.

```
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
```

B. Explanation of the "elf.inc" file

Like "efi.inc", "elf.inc" contains structures defined according to the ELF64 binary structure mentioned in the System V ABI specification [7]. These structures make it simple to "offset into" data stored in the kernel binary, which uses ELF64, and parse information such as the entry point of execution and where in memory to load the kernel.

Below is the ELF64 header structure defined in the "elf.inc" file.

```
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
```

C. Explanation of the "bootloader.asm" file

The bootloader.asm process contains the assembly that produces the flat bootloader binary executable, which is in the PE32+ format. As such, it begins with the required PE32+ data fields defined using FASM, according to the PE32+ standard [9].

The beginning of the PE32+ assembly that starts generating bytes is shown below. The DosStub followed by the PE signature and the COFF File Header's members identify that the binary is indeed a PE32+ binary. Data about

the size of the headers and pointers to other data structures within the binary such as symbol tables are either calculated dynamically with FASM's operators or calculated manually and then set:

```
   DosStub:
   db 0x4d, 0x5a, 0x80, 0x00, 0x01, 0x00, 0x00,
0x00
   db 0x04, 0x00, 0x10, 0x00, 0xff, 0xff, 0x00,
0x00
   db 0x40, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00
   db 0x40, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00
   db 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00
   db 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00
   db 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00
   db 0x00, 0x00, 0x00, 0x00, 0x80, 0x00, 0x00,
0x00
   db 0x0e, 0x1f, 0xba, 0x0e, 0x00, 0xb4, 0x09,
0xcd
   db 0x21, 0xb8, 0x01, 0x4c, 0xcd, 0x21, 0x54,
0x68
   db 0x69, 0x73, 0x20, 0x70, 0x72, 0x6f, 0x67,
0x72
   db 0x61, 0x6d, 0x20, 0x63, 0x61, 0x6e, 0x6e,
0x6f
   db 0x74, 0x20, 0x62, 0x65, 0x20, 0x72, 0x75,
0x6e
   db 0x20, 0x69, 0x6e, 0x20, 0x44, 0x4f, 0x53,
0x20
   db 0x6d, 0x6f, 0x64, 0x65, 0x2e, 0x0d, 0x0a,
0x24
   db 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00

   PeSignature:
   db 'PE', 0, 0

   CoffFileHeader:
   .Machine               dw 0x8664
   .NumberOfSections      dw 2
   .TimeDateStamp         dd %t and 0xffffffff
   .PointerToSymbolTable  dd 0
   .NumberOfSymbols       dd 0
   .SizeOfOptionalHeader  dw SectionTable -
PeOptionalHeader
   .Characteristics       dw 0x0001 or 0x0002 or
0x0020 or 0x0200
```

Next is the actual assembly code which bootstraps the system. In accordance with the first principles approach, all actions required for interaction with hardware are done via the UEFI ABI. UEFI function calls are performed using Microsoft x64 calling convention in assembly as defined in the specifications [11], which states the registers where arguments are to be stored in order to pass them as arguments during a function call.

The process of loading the kernel file into memory is thus described as below. It can be seen how the bootloader.asm file makes use of the offsets to UEFI functions defined previously in "efi.inc":

```
   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.DeviceHandle]
   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.OpenVolume]

   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
```

This involves using the UEFI OpenVolume function, a part of BootServices which gives full functionality for OS loaders.

The code below 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:

```
   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]

   mov rax, [SystemTable]
   mov rax, [rax+EFI_SYSTEM_TABLE.BootServices]
   mov rcx, [ImageHandle]
   mov rdx, [MapKey]
   call [rax+EFI_BOOT_SERVICES.ExitBootServices]

   mov rcx, bootloaderStruct
   call [KernelEntryPoint]
```

Arguments are passed using the bootloaderstruct structure defined as follows in the "bootloader.inc" file. The two items passed are the address of the graphics framebuffer and the pixels count per scanline for rendering purposes.

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

D. Explanation of the "kernel.asm" file

The kernel binary is generated in the same manner as the bootloader binary with flat assembly but this time using the ELF64 format. Of note here are the multitude of data fields that specify where the sections of the ELF64 file begin and their sizes:

```
.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
```

This is where the kernel obtains the information passed to it and saves the arguments:

```
startcode:
main:

sub rsp, 4*8

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

The rest of the code in the kernel is assembly to render the words "HELLOWORLD" using a bitmap font, which is also stored in the binary.

This completes the bootloading and kernel execution process.

V. Results and Analysis

Building the implementation has been performed successfully on Linux. In the directory containing the five assembly files, the bootloader and kernel are assembled by running fasm:

```
fasm2 -n bootloader.asm bootloader.efi
fasm2 -n kernel.asm kernel
```

The bootloader and kernel are then copied into an EFI-compatible image using mtools. From said image, a bootable ISO is created using xorriso.

Finally, QEMU, an emulator, can be used to run the ISO using openly available OVMF files for UEFI support:

```
qemu-system-x86_64 -drive
if=pflash,format=raw,readonly=on,file=OVMF_CODE.fd
-drive if=pflash,format=raw,file=OVMF_VARS.fd -
cdrom bootable.iso
```

The virtual machine boots up and successfully renders the letters 'HELLOWORLD', as can be seen in Figure 3.

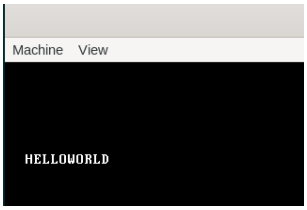


Fig. 3. Bootloader Successfully Loading the Demo Kernel inside QEMU

The exact commands for the previous steps are detailed in a Makefile, and the user can run a single "make" command to build and test the system. The implementation files, along with configuration details and prerequisites can be found in the GitHub repository at [12].

To quantitatively evaluate the benefits of our first-principles, minimal-dependency approach, we conducted a comparative analysis. This analysis does not compare features, but rather foundational metrics of complexity and footprint against Limine, a modern bootloader with a considerable number of users.

A. Build Dependencies

Limine requires eight build dependencies in the form of Linux programs and the GCC or LLVM/Clang toolchain along with their respective binutils [13]. In comparison, our bootloader only requires three build dependencies: coreutils, make, fasm. which can be easily seen by analyzing the Makefile, and, most notably, requires no C/C++ toolchain. In both bootloaders, mtools and xorriso are used to create the final ISO image.

B. Source Code Complexity

Limine supports multiple architectures including the legacy 32-bit x86 architecture, as well as BIOS systems and the MBR partitioning scheme, which are used less in modern PCs. Our bootloader solely focuses on x64 UEFI and will continue to do so in the future. This drastically simplifies complexity of the codebase. Furthermore, Limine uses NASM while our bootloader uses FASM, which is more focused on cleaner syntax and macros for assembly development.

C. Executable File Size

Limine's x64 bootloader EFI executable is ~248KB at its current version (v10.1.1) while our bootloader's final EFI executable is ~132KB, almost half the size of Limine. While our bootloader lacks advanced features, our size advantage comes from eliminating toolchains, and additional features added as assembly will not substantially impact the code size.

VI. Discussion

The implementation of this UEFI bootloader represents a significant milestone in the ongoing effort to develop a custom operating system [7]. 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 minimal in nature and does not contain "legacy baggage". Legacy baggage consists of (but is not limited to) the following.

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. For example, UNIX (on which Linux is based) 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 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 increasing 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 [14], [15]. The sheer size of these codebases requires a huge undertaking to refactor.

C. The need to maintain backward compatibility

This refers to the maintenance of code solely 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 platforms in order that they keep running on legacy hardware. Another example is 32-bit x86 code. x86-64 (the 64-bit architecture) has superseded 32-bit x86, which could only support 4 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.

The main challenges of bootloader and OS complexity are rooted in the aforementioned "legacy baggage", leading to a large codebase, large executable sizes, low maintainability, and reduced extensibility in the future.

Future work may involve developing a multikernel operating system like the Barrelfish OS described by Baumann et al. in [16]. This kind of OS allows massive parallelization and scalability as it is distributed at the lowest kernel level instead of using a middleware to achieve distributed properties. The bootloader and kernel system developed using the first principles approach may be used as a modern, clean basis for such work.

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 approach described in this paper 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 educational resource in Operating Systems curriculums to offer insight into the low-level workings of a bootloader's mechanisms.

References

[1] Free Software Foundation. *GNU GRUB*. [Online]. Available: https://www.gnu.org/software/grub.

[2] Free Software Foundation, "Multiboot2 specification," Version 2.0, 2016. [Online]. Available: https://www.gnu.org/software/grub/manual/multiboot2/multiboot.html.

[3] Mintsuki. *Limine Bootloader*. [Online]. Available: https://github.com/limine-bootloader/limine.

[4] Mintsuki. *Limine Boot Protocol*. [Online]. Available: https://codeberg.org/Limine/limine-protocol/src/branch/trunk/PROTOCOL.md.

[5] B. Z. Tam´as. *BOOTBOOT Bootloader*. [Online]. Available: https://gitlab.com/bztsrc/bootboot.

[6] B. Z. Tam´as, "The BOOTBOOT protocol specification and manual," Revision 1.0, 2017-2021. [Online]. Available: https://gitlab.com/bztsrc/bootboot/-/raw/master/bootboot_spec_1st_ed.pdf.

[7] Intel, "System V application binary interface," 2013. [Online]. Available: https://www.sco.com/developers/gabi/latest/contents.html.

[8] M. Matz, J. Hubička, A. Jaeger, and M. Mitchell, "AMD64 architecture processor supplement," Revision 0.99.6, Jul. 2, 2012. [Online]. Available: https://refspecs.linuxfoundation.org/elf/x86_64-abi-0.99.pdf.

[9] Microsoft, "Microsoft Portable Executable and Common Object File Format specification," Revision 6.0, Feb. 1999. [Online]. Available: http://www.osdever.net/documents/PECOFF.pdf.

[10] UEFI Forum, "Unified Extensible Firmware Interface (UEFI) specification," Version 2.11, Nov. 2024. [Online]. Available: https://uefi.org/sites/default/files/resources/UEFI_Spec_Final_2.11.pdf.

[11] Microsoft, "x64 calling convention," *Microsoft Learn*, Mar. 19, 2025. [Online]. Available: https://learn.microsoft.com/en-us/cpp/build/x64-calling-convention.

[12] M. M. Aung. *UEFI Bootloader and Kernel from First Principles*. [Online]. Available: https://github.com/meaung/uefi-bootloader-and-kernel-from-first-principles.

[13] Mintsuki. *Limine Bootloader Build and Install Instructions*. [Online]. Available: https://github.com/limine-bootloader/limine/blob/v10.x/INSTALL.md.

[14] 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.

[15] H. Xu and R. Tang, "Study and improvements for the realtime performance of linux kernel," in *2010 3rd International Conference on Biomedical Engineering and Informatics,* vol. 7, 2010, pp. 2766–2769.

[16] A. Baumann et al., "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.