OSPP 项目申请书

项目名称：ArceBoot 引导程序设计

项目编号：256590454

个人信息
项目背景与需求
设计与实现
    UEFI Runtime
    Arce OS 与 UEFI Runtime
        内存分配器
        文件系统
    Arm EBBR
    提供接口
    实现难点
时间规划
    时间线
    预期效果

个人信息

一名在读大学生，正在学习操作系统、组成原理等知识，能够使用 Rust、Java、Typescript 等语言进行项目编写，能够独立解决一些实际工程问题，努力参与到开源社区中！

Github：https://github.com/hanbings
Blog：https://blog.hanbings.io
邮箱：hanbings@hanbings.io

项目背景与需求

RISC-V 作为近年兴起的开源指令集架构，因其高度的可定制性和生态开放性，正在逐步扩展在嵌入式、边缘计算乃至服务器等高性能领域的影响力。不同厂商硬件实现差异大，设备树结构复杂，启动方式不一，系统适配难度高。

在传统架构（如 x86 或 ARM64）中，UEFI 已成为广泛采用的引导标准，为系统启动提供了清晰统一的接口。而在 RISC-V 领域，UEFI 尚未成为标准或是事实标准，大部分平台仍采用硬编码跳转、OpenSBI 固定入口、甚至裸机式的启动方式，这在服务器和通用操作系统场景中严重限制了其部署与维护效率。

> **BIOS、*SBI 与 UEFI**
> 传统的 x 86 计算机启动流程依赖于 BIOS（Basic Input/Output System），其主要通过固定地址的固件代码初始化硬件并引导操作系统。然而，BIOS 存在启动流程复杂、扩展能力弱、无法支持大容量磁盘（如 GPT）等局限。
> 在 RISC-V 架构中，目前存在 *SBI、LinuxBoot 和 UEFI 三种固件生态路径，发展程度差异明显。基于 SBI 的方案已经相对成熟，SBI 作为特权级之间的标准化接口，OpenSBI、RustSBI 等实现已被广泛采用；LinuxBoot 固件利用 Linux 驱动体系，也已有一定的成熟产品；然而，UEFI 固件在 RISC-V 生态中明显滞后，虽然在 x86 架构上已非常成熟，但针对 RISC-V 的 UEFI 实现仍缺乏稳定的商用级解决方案和广泛生态支持。
> UEFI（Unified Extensible Firmware Interface）是对 BIOS 的现代替代方案，由 Intel 主导并逐步成为主流架构上的引导标准。UEFI 以模块化设计提供了图形界面、网络启动、安全启动等扩展功能，并使用标准化的 PE/COFF 格式加载操作系统或其他 EFI 应用程序。

此外，EDK2 或 GNU EFI 等项目提供了 UEFI 的开发工具包，它们有项目规模较大、使用 C / C++ 以及它们的工具链、对 RISC-V 架构支持滞后等问题。

在这种背景下，ArceBoot 项目基于 Rust 语言，使用 ArceOS 的组件化系统生态，构建一个轻量级别的 UEFI 运行时环境或是说 UEFI 固件，支持在多种架构中快速运行和调试 UEFI 程序，并尝试解决在 RISC-V 中的 UEFI 生态的问题。

设计与实现

以 x86 为例子，从裸机到启动操作系统之间，有一段用于从 BIOS / UEFI 获取硬件控制权、设置（临时）页表、从存储设备读取内核文件、控制操作系统内核加载和内核映射的代码被称为 BootLoader，在 UEFI 固件下，该 BootLoader 通常为 UEFI 的 EFI 程序。

在 UEFI 中，`EFI_SYSTEM_TABLE`（系统表）、`EFI_BOOT_SERVICES`（引导服务）和 `EFI_RUNTIME_SERVICES`（运行时服务）是三个核心的结构体，分别作为系统服务、引导服务与运行时服务的索引表。它们本质上是由大量**函数指针与数据字段**组成的结构体，提供了调用 UEFI 各类功能的统一接口。开发者在编写 EFI 程序时，通常通过 `SystemTable` 访问 `BootServices` 和 `RuntimeServices`，进而调用如内存管理、设备访问、变量读写等底层功能。因此，要实现一个完整的 UEFI 环境，关键之一是**正确定义并组织这些结构体的内存布局**，确保 EFI 程序在运行时能够准确地读取并调用其中的函数。

那么关键步骤如下：

1. 裸机启动，进入非 UEFI 的裸机固件程序
2. 裸机固件程序扫描硬件后跳入 UEFI 固件程序，也就是本文中的 `UEFI Runtime`
3. UEFI 负责进一步为硬件分类、建立 `UEFI Runtime` 页表、建立内存映射、填充 `SystemTable` 等关键数据结构等工作
4. UEFI 按指定顺序从存储设备扫描符合格式的 EFI 文件
5. 将 EFI 文件加载到内存，并跳入 EFI 程序（UEFI 内抽象为 `Image` 结构）的程序入口函数

UEFI Runtime

基于以上要求，逐步对各个环节进行验证。本节主要描述编写一个 EFI 加载器以构建结构体的方式向 EFI 程序提供接口，将构建一个简单的 EFI 程序并调用输出函数，向屏幕打印字符。

EFI 程序的一个较为典型的入口函数签名为：`int efi_main(void *imageHandle, EfiSystemTable* systemTable)`，其中 `void *imageHandle` 指向正在运行的这个 EFI 程序，`EfiSystemTable* systemTable` 则指向系统表。

https://github.com/AndreVallestero/minimal-efi 是一个不依赖于 EDK2 的最小化 x86 EFI 程序。使用如下指令可以在 Linux 下进行编译：`clang -target x86_64-pc-win32-coff -fno-stack-protector -fshort-wchar -mno-red-zone -c main.c -o main.o && lld-link -filealign:16 -subsystem:efi_application -nodefaultlib -dll -entry:efi_main main.o -out:BOOTX64.EFI && rm main.o BOOTX64.lib`

本节的 EFI 加载器将围绕着这个最小化 x86 EFI 程序展开。

参考 https://github.com/AndreVallestero/minimal-efi/blob/master/main.c 可以得到以下关键的结构：

```
typedef struct EfiSystemTable {
    EfiTableHeader                hdr;
    int16_t*                      firmwareVendor;
    uint32_t                      firmwareRevision;
    void*                         consoleInHandle;
    uint64_t                      conIn;
    void*                         consoleOutHandle;
    EfiSimpleTextOutputProtocol* conOut;
```

```
    void*                       standardErrorHandle;
    uint64_t                    stdErr;
    uint64_t                    runtimeServices;
    uint64_t                    bootServices;
    uint64_t                    numberOfTableEntries;
    uint64_t                    configurationTable;
} EfiSystemTable;
```

对于本节来说，需要使用 `EfiSimpleTextOutputProtocol* conOut;` ，`EfiSimpleTextOutputProtocol` 定义如下：

```
struct EfiSimpleTextOutputProtocol;
typedef uint64_t (*EfiTextString)(struct EfiSimpleTextOutputProtocol* this, int16_t*
string);
typedef struct EfiSimpleTextOutputProtocol {
    uint64_t      reset;
    EfiTextString output_string;
    uint64_t      test_string;
    uint64_t      query_mode;
    uint64_t      set_mode;
    uint64_t      set_attribute;
    uint64_t      clear_screen;
    uint64_t      set_cursor_position;
    uint64_t      enable_cursor;
    uint64_t      mode;
} EfiSimpleTextOutputProtocol;
```

同时观察编译得到的 `BOOTX64.EFI` 文件：

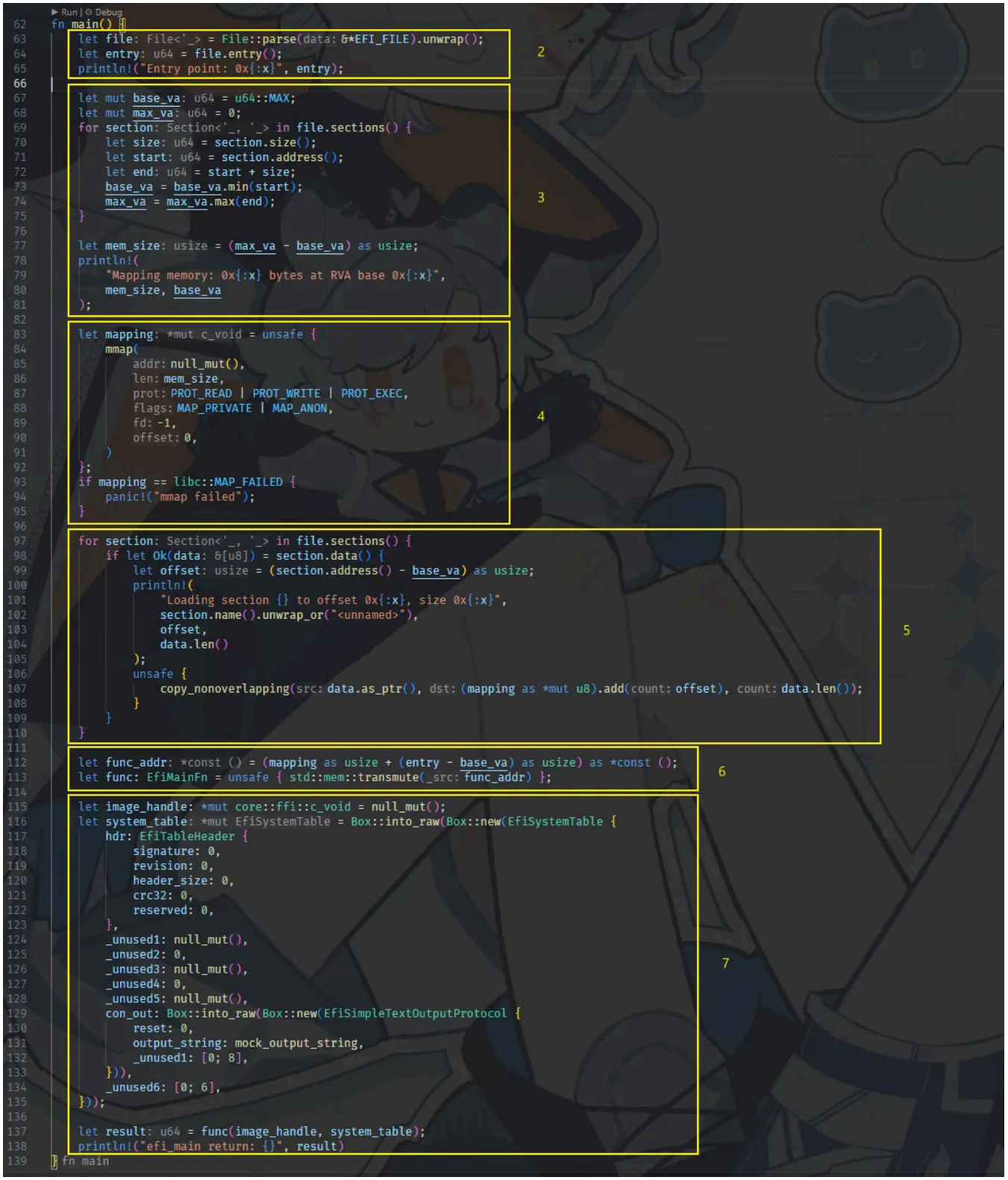
```
➜  arceboot-test file BOOTX64.EFI
BOOTX64.EFI: PE32+ executable (DLL) (EFI application) x86-64, for MS Windows, 3 sections
➜  arceboot-test llvm-readelf -S BOOTX64.EFI

File: BOOTX64.EFI
Format: COFF-x86-64
Arch: x86_64
AddressSize: 64bit
Sections [
  Section {
    Number: 1
    Name: .text (2E 74 65 78 74 00 00 00)
    VirtualSize: 0x34
    VirtualAddress: 0x1000
    RawDataSize: 64
    PointerToRawData: 0x250
    PointerToRelocations: 0x0
    PointerToLineNumbers: 0x0
    RelocationCount: 0
    LineNumberCount: 0
    Characteristics [ (0x60000020)
      IMAGE_SCN_CNT_CODE (0x20)
      IMAGE_SCN_MEM_EXECUTE (0x20000000)
```

```
      IMAGE_SCN_MEM_READ (0x40000000)
    ]
  }
  Section {
    Number: 2
    Name: .rdata (2E 72 64 61 74 61 00 00)
    VirtualSize: 0x24
    VirtualAddress: 0x2000
    RawDataSize: 48
    PointerToRawData: 0x290
    PointerToRelocations: 0x0
    PointerToLineNumbers: 0x0
    RelocationCount: 0
    LineNumberCount: 0
    Characteristics [ (0x40000040)
      IMAGE_SCN_CNT_INITIALIZED_DATA (0x40)
      IMAGE_SCN_MEM_READ (0x40000000)
    ]
  }
  Section {
    Number: 3
    Name: .pdata (2E 70 64 61 74 61 00 00)
    VirtualSize: 0xC
    VirtualAddress: 0x3000
    RawDataSize: 16
    PointerToRawData: 0x2C0
    PointerToRelocations: 0x0
    PointerToLineNumbers: 0x0
    RelocationCount: 0
    LineNumberCount: 0
    Characteristics [ (0x40000040)
      IMAGE_SCN_CNT_INITIALIZED_DATA (0x40)
      IMAGE_SCN_MEM_READ (0x40000000)
    ]
  }
]
```

它是一个 PE 格式的文件，在 Rust 中，我们可以使用 object 库解析这种类型的文件文件。

因此整个 EFI 加载器要做的工作就是：

1. 使用 `include_bytes!` 在最终生成的二进制文件中夹带 EFI 文件
2. 使用 `object` 库解析 EFI（PE） 文件
3. 遍历 `object` 解析出来的 `Section`，计算出能容纳全部 `Section` 所需要的大小
4. 通过 libc 提供的 mmap 向操作系统申请可读可写可执行的内存块
5. 将 EFI 文件原封不动的拷贝到申请出来的内存块里
6. 计算得出 EFI 的执行入口，也就是 `int efi_main(void *imageHandle, EfiSystemTable* systemTable)` 的位置
7. 用自己的 `output_string` 填充 `SystemTable`，并传递给 EFI 的执行入口（此时其实是把它看作一个带参数的函数处理的），并执行这个函数，最后得到返回值



完整代码：

```
use libc::{MAP_ANON, MAP_PRIVATE, PROT_EXEC, PROT_READ, PROT_WRITE, mmap};
use object::{File, Object, ObjectSection};
use std::{
    char::decode_utf16,
    ptr::{copy_nonoverlapping, null_mut},
};

pub static EFI_FILE: &'static [u8] = include_bytes!("../../boot.efi");
```

```rust
pub type EfiTextString =
    extern "efiapi" fn(this: *mut EfiSimpleTextOutputProtocol, string: *const u16) -> u64;
pub type EfiMainFn = extern "efiapi" fn(
    image_handle: *mut core::ffi::c_void,
    system_table: *mut EfiSystemTable,
) -> u64;

#[repr(C)]
pub struct EfiTableHeader {
    pub signature: u64,
    pub revision: u32,
    pub header_size: u32,
    pub crc32: u32,
    pub reserved: u32,
}

#[repr(C)]
pub struct EfiSimpleTextOutputProtocol {
    pub reset: u64,
    pub output_string: EfiTextString,
    pub _unused1: [u64; 8],
}

#[repr(C)]
pub struct EfiSystemTable {
    pub hdr: EfiTableHeader,
    pub _unused1: *mut core::ffi::c_void,
    pub _unused2: u32,
    pub _unused3: *mut core::ffi::c_void,
    pub _unused4: u64,
    pub _unused5: *mut core::ffi::c_void,
    pub con_out: *mut EfiSimpleTextOutputProtocol,
    pub _unused6: [u64; 6],
}

extern "efiapi" fn mock_output_string(
    _this: *mut EfiSimpleTextOutputProtocol,
    string: *const u16,
) -> u64 {
    unsafe {
        let mut len = 0;
        while *string.add(len) != 0 {
            len += 1;
        }
        let message = core::slice::from_raw_parts(string, len as usize).iter();
        let utf16_message = decode_utf16(message.cloned());
        let decoded_message: String = utf16_message.map(|r|
r.unwrap_or('\u{FFFD}')).collect();
        println!("EFI Output: {}", decoded_message);
    }
    0
}
```

```rust
fn main() {
    let file = File::parse(&*EFI_FILE).unwrap();
    let entry = file.entry();
    println!("Entry point: 0x{:x}", entry);

    let mut base_va = u64::MAX;
    let mut max_va = 0;
    for section in file.sections() {
        let size = section.size();
        let start = section.address();
        let end = start + size;
        base_va = base_va.min(start);
        max_va = max_va.max(end);
    }

    let mem_size = (max_va - base_va) as usize;
    println!(
        "Mapping memory: 0x{:x} bytes at RVA base 0x{:x}",
        mem_size, base_va
    );

    let mapping = unsafe {
        mmap(
            null_mut(),
            mem_size,
            PROT_READ | PROT_WRITE | PROT_EXEC,
            MAP_PRIVATE | MAP_ANON,
            -1,
            0,
        )
    };
    if mapping == libc::MAP_FAILED {
        panic!("mmap failed");
    }

    for section in file.sections() {
        if let Ok(data) = section.data() {
            let offset = (section.address() - base_va) as usize;
            println!(
                "Loading section {} to offset 0x{:x}, size 0x{:x}",
                section.name().unwrap_or("<unnamed>"),
                offset,
                data.len()
            );
            unsafe {
                copy_nonoverlapping(data.as_ptr(), (mapping as *mut u8).add(offset),
data.len());
            }
        }
    }

    let func_addr = (mapping as usize + (entry - base_va) as usize) as *const ();
```

```
    let func: EfiMainFn = unsafe { std::mem::transmute(func_addr) };

    let image_handle: *mut core::ffi::c_void = null_mut();
    let system_table = Box::into_raw(Box::new(EfiSystemTable {
        hdr: EfiTableHeader {
            signature: 0,
            revision: 0,
            header_size: 0,
            crc32: 0,
            reserved: 0,
        },
        _unused1: null_mut(),
        _unused2: 0,
        _unused3: null_mut(),
        _unused4: 0,
        _unused5: null_mut(),
        con_out: Box::into_raw(Box::new(EfiSimpleTextOutputProtocol {
            reset: 0,
            output_string: mock_output_string,
            _unused1: [0; 8],
        })),
        _unused6: [0; 6],
    }));

    let result = func(image_handle, system_table);
    println!("efi_main return: {}", result)
}
```

Arce OS 与 UEFI Runtime

现有工作已经完成了通过 Arce OS 组件读取文件以及构建 SBI Payload。

https://github.com/rustsbi/arceboot/pull/2

https://github.com/rustsbi/arceboot/pull/3

根据同样的思路，也可以使用 Arce OS 的内存分配器组件等简化实现。在这一节中，列举（堆）内存分配器和文件系统两个模块。

内存分配器

使用 Arce OS axalloc 组件作为 UEFI Runtime 的堆内存分配器，并简单描述 axalloc 中所实现的分配器。

`axalloc` 主要使用 TLSF、Buddy 和 Slab 分配算法，提供了 rust feature cfg 进行配置：

```
cfg_if::cfg_if! {
    if #[cfg(feature = "slab")] {
        /// The default byte allocator.
        pub type DefaultByteAllocator = allocator::SlabByteAllocator;
    } else if #[cfg(feature = "buddy")] {
        /// The default byte allocator.
        pub type DefaultByteAllocator = allocator::BuddyByteAllocator;
    } else if #[cfg(feature = "tlsf")] {
        /// The default byte allocator.
        pub type DefaultByteAllocator = allocator::TlsfByteAllocator;
    }
}
```

在 `config.toml` 中添加依赖，并声明分配器对应的 `feature` 即可使用：

```
axalloc = { git = "https://github.com/arceos-org/arceos.git", optional = true }
```

Buddy 分配器

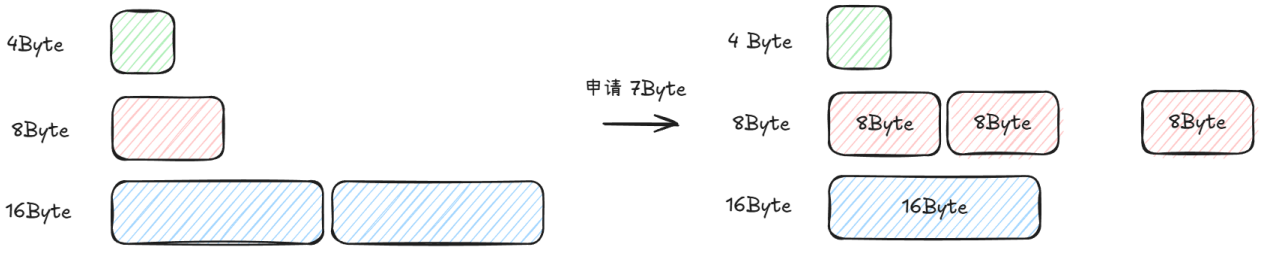
Buddy 分配器将物理内存划分成大小为 2 的幂次方的块。例如 4Byte、8Byte、16Byte（真实操作系统的分配器会有多个层级的内存分配器，从页 / 大页到字符都会有对应的分配器，本节主要讨论字符粒度的分配器）

Buddy 分配器关键的点在于内存块分裂和合并，如内存块 16Byte 可以分裂两个连续内存地址的 8Byte 内存块，同样的 8Byte 也可以分裂为两个 4Byte 块。Buddy 会为每一个大小的块都开辟一个链表，将所有大小相同的块串联到一起，并在运行时动态的分裂和合并块到对应大小的链表中。

当应用申请内存时会经过：

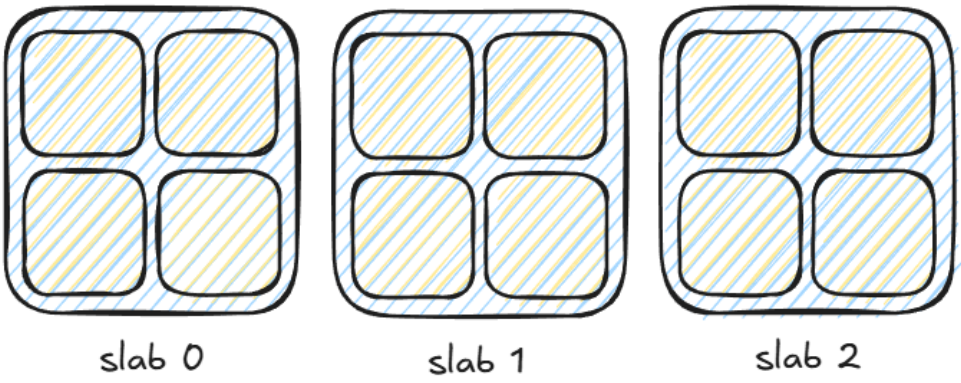
1. 判断需要的内存是否位 2 的幂次方大小
2. 从对应大小的链表中获取足够大小的内存，并将剩余的内存返还给分配器
3. 分配器将剩余内存添加回对应的链表中

如图，如果申请 7Byte 大小的内存块，在分配器认为 8Byte 链表需要补充块时，会从 16Byte 取出一块内存块，并将其中的 8Byte 返回给申请方，然后把剩下的 8Byte 添加到 8Byte 块的链表中，尽可能地避免浪费：



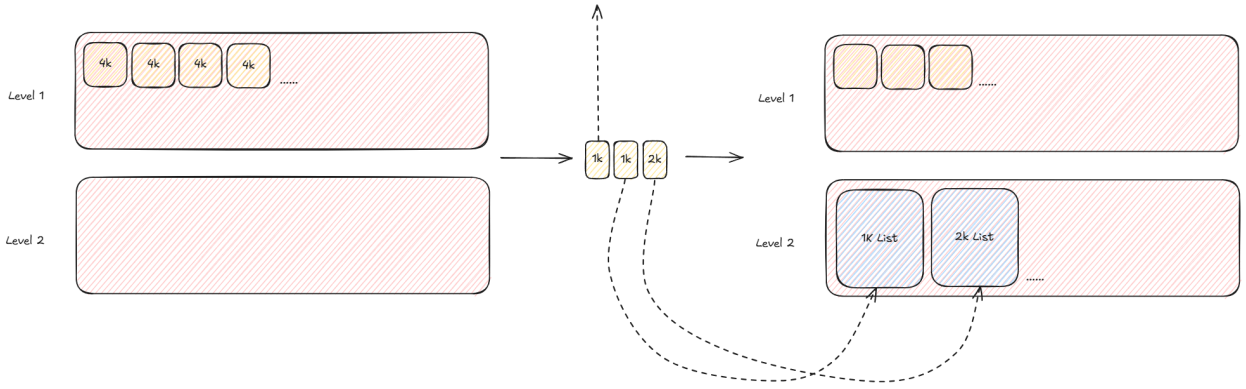
当 Buddy 分配器回收时，分配器会判断当前内存一开始分裂的另一半是否还在链表里，且需要进行合并了，将会合并到上一级大小。

Slab 分配器



Slab 分配器的思路是将较大块内存切分为包含相同大小的内存块的几段，一个段就是一个 Slab，如 1Byte、2Byte、3Byte 和 4Byte，每一次分配尽可能地从合适的大小中分配。通常可以将 Slab 分配器与 Buddy 分配器组合使用，如 Buddy 分配器的例子中，我们 8Byte 的内存块使用了 7Byte，剩余 1Byte 就可以添加到 Slab 的 1Byte 段中，以此管理内存碎片。

TLSF （Two-Level Separable Fit）分配器



TLSF 分配器在初始化时会开辟两层空间，将较大块的内存合理地分配到第一层，并将其连接成链表。当有内存申请时，假如申请的内存块大小较小，例如一个 4k 的内存块仅申请 1k，那么剩余的 3k 会被切割成 1k 和 2k，分别归还到第二层相应的链表中，以供后续其他内存请求使用。通过这种方式，TLSF 能有效地管理内存块，尽量减少碎片化。

文件系统

UEFI 中的 EFI 普遍使用 FAT32 作为文件系统，我们可以引入 Arce OS 的 axfs 作为文件系统读写模块，同样的，在 `config.toml` 声明依赖：

```
axfs = { git = "https://github.com/arceos-org/arceos.git", optional = true }
```

在 axfs/lib.rs 中实现了文件系统的读写功能，且实现了抽象层 vfs ，如下列出了一部分的函数签名：

```
// 新建文件
fn new_file(file: File<'_, Disk, NullTimeProvider, LossyOemCpConverter>) -> Arc<FileWrapper>

// 新建文件夹
fn new_dir(dir: Dir<'_, Disk, NullTimeProvider, LossyOemCpConverter>) -> Arc<DirWrapper>

// 查找文件
fn lookup(self: Arc<Self>, path: &str) -> VfsResult<VfsNodeRef>

...
```

Arm EBBR

Arm EBBR https://github.com/ARM-software/ebbr ， Embedded Base Boot Requirements (EBBR) specification，是一套定义 SoC、硬件平台和固件之间启动协议的规范。是 Arm 平台上的一种 UEFI 最小化规范，以启动顺序为例，EBBR 定义了 `BootOrder`、`BootNext` 和 `BootCurrent` 等变量的管理、在 `BootOrder` 尝试失败后，能够在可移动和固定介质上搜索默认引导应用程序、必须实现 `EFI_SIMPLE_FILE_SYSTEM_PROTOCOL` 加载应用程序，支持 `LoadImage()` 函数，配置看门狗定时器，以及通过 `EFI_LOAD_FILE_PROTOCOL` 引导等强制性要求。

对本项目来说，它能够提供大致的思路和方向实现 UEFI 规范，而不需要翻阅全部的 UEFI 手册内容。

提供接口

UEFI 规范中对 UEFI 运行时提供的接口均匀明确描述，本项目参考于 Arm EBBR 规范，其中对运行时服务如下：

Table 2.8: *EFI_RUNTIME_SERVICES* Implementation Requirements

EFI_RUNTIME_SERVICES function	Before *ExitBootServices()*	After *ExitBootServices()*
GetTime	Required if RTC present.	Optional
SetTime	Required if RTC present.	Optional
GetWakeupTime	Required if wakeup supported.	Optional
SetWakeupTime	Required if wakeup supported.	Optional
SetVirtualAddressMap	N/A	Required
ConvertPointer	N/A	Required
GetVariable	Required	Optional
GetNextVariableName	Required	Optional
SetVariable	Required	Optional
GetNextHighMonotonicCount	N/A	Optional
ResetSystem	Required	Optional
UpdateCapsule	Required for in-band update.	Optional
QueryCapsuleCapabilities	Optional	Optional
QueryVariableInfo	Optional	Optional

对于本项目来说，需要注意的地方有：内存对齐方式、函数调用方式、来自于 Runtime Service 所要求的虚拟地址功能实现、变量存储区设置、高精度时钟实现。

实现难点

- 不同架构的差异

  参考 Arce OS，可以使用 Rust Trait 机制分别对不同架构进行实现。

```
trait ArchInterface {
    fn cpu_init(&self);
    fn mmu_init(&self) -> MemoryMap;
    fn get_firmware_info(&self) -> FirmwareHob;
}

// RISC-V 实现
struct RiscVImpl;
impl ArchInterface for RiscVImpl {
    fn cpu_init(&self) {
        unsafe { riscv::asm::wfi() };
    }
    ...
}

// x86_64 实现
struct X86Impl;
impl ArchInterface for X86Impl {
    fn cpu_init(&self) {
        unsafe { x86::controlregs::Cr0::update(...) };
    }
    ...
}
```

- 满足 https://github.com/ARM-software/ebbr 规范的 UEFI 接口子集完整实现

  通过 https://github.com/ARM-software/bbr-acs 的 UEFI Self Certification Tests (SCT)

  和 Firmware Test Suite (FWTS) 等测试对实现的 UEFI 运行时进行完整性测试。

时间规划

时间线

时间范围	阶段任务	目标产出
07/01 - 07/15	项目预研与技术验证	验证 Rust 环境下 UEFI 运行最小样例，确定引导流程的初始设计与架构、平台的支持目标
07/16 - 07/31	UEFI 应用加载框架搭建	实现基本 UEFI 入口、内存分配、控制台输出，加载 `.efi` 文件并跳转执行的初步实现

时间范围	阶段任务	目标产出
08/01 - 08/15	文件系统与驱动集成	通过 `axdriver` 探测启动设备与使用 `axfs` 加载扫描文件系统，并加载文件形式的 `.efi`
08/16 - 08/31	UEFI 接口的进一步支持、PE/COFF 文件解析与内核加载	实现对 `.efi` 格式内核的解析与内存映射- 搭建跳转执行机制，尝试启动内核测试文件
09/01 - 09/15	模块完善与平台适配	进一步集成 `axdriver` 和 `axfs` ，对多架构进行支持，实现 GOP（图形协议），映射外部 IO 设备，尝试实现安全启动
09/16 - 09/30	测试文档与贡献提交	编写使用文档、构建脚本、测试案例

预期效果

在尽可能多的架构（x86、AArch、RISC-V）中，完成 ArceBoot 的 UEFI 运行时和引导工具实现，支持 UEFI 运行时内分配内存、读写存储空间、正确映射显示区域、正确映射外围 IO 硬件等，支持在QEMU或者真实硬件上加载并启动内核，并尝试引导 Linux 等成熟的操作系统。

提供包括示例配置和跨平台测试指南完整在内的构建与使用文档。