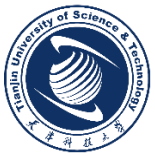


天津科技大学

Tianjin University of Science & Technology

Rust 编写的基于RISC-V64 的多核操作系统 TCore

项目成员：王　炼（组长）

王　涛

陈钰霖

指导老师：梁　琨

目 录

1 概述

TCore为用Rust编写的基于RISC-V的多核操作系统，并且能够运行在实体裸机K210平台

1.1 项目背景及意义

从天津科技大学的计算机系统教学角度来说，不管是在理论的教学还是在实际的实验操作中，都显得较为薄弱。作为计算机专业的四大科目之一的OS，是理解程序运行以及计算机资源管理的重中之重，从教学上来说，正是因为其机制的复杂性，导致了实践上和理论上的障碍。

同时，我们认为天津科技大学的计算机系统人才的关键在于三点：理论课程的贴合性（不在满足于古老的PPT）、课程实验（精巧可行又富有挑战性的实验）、领域前沿探索（有对应领域的指导老师进行深入挖掘）。我们迫切需要一个麻雀虽小五脏俱全的操作系统展示实现原理和细节的同时又不至于让学生陷入到复杂而又巨大的现代操作系统中去

我们着力解决第二点：课程实验。我们以清华大学的教学项目 rCore-Tutorial-V3 为核心，在其代码框架上进行二次开发，以拓展自由课程实验的设计。这样也能够有更大的自由空间和进一步深化了解的空间。

同时我们还希望比赛之外，设计教学，带动本校的学习氛围，构建适合实验操作的代码，进行传承下去。

1.2 项目的主要工作

TCore 致力于开发一个基于RISC-V64的宏内核操作系统，并且能够运行在实体裸机K210平台上。这里需要注意的是，TCore的初心是促成一次教学尝试，也就是提供一个探索的先例，为后续学弟学妹探个底。同时，在比赛过程中，我们也应该选取尽可能经典的方案在系统能力培养中许多高校都取得了一些显著的成果我们希望在借鉴和模仿中加入自己的思路为一个可行的但又是经典的内核迈出第一步。在学习和接触中我们了解到工业界在云存储开发中SPDK和DPDK得到了广泛的应用他凭借着轮询机制极大的提高了I/O因此我们在开发中有意借鉴他的开发思路想要实现一个线程机制提供一个展示和技术培养的氛围。总之，得奖并不是我们的主要目标，他们或许在我们的开发过程中成为中间目标，但 TCore 的初心还是给予大家一个操作系统

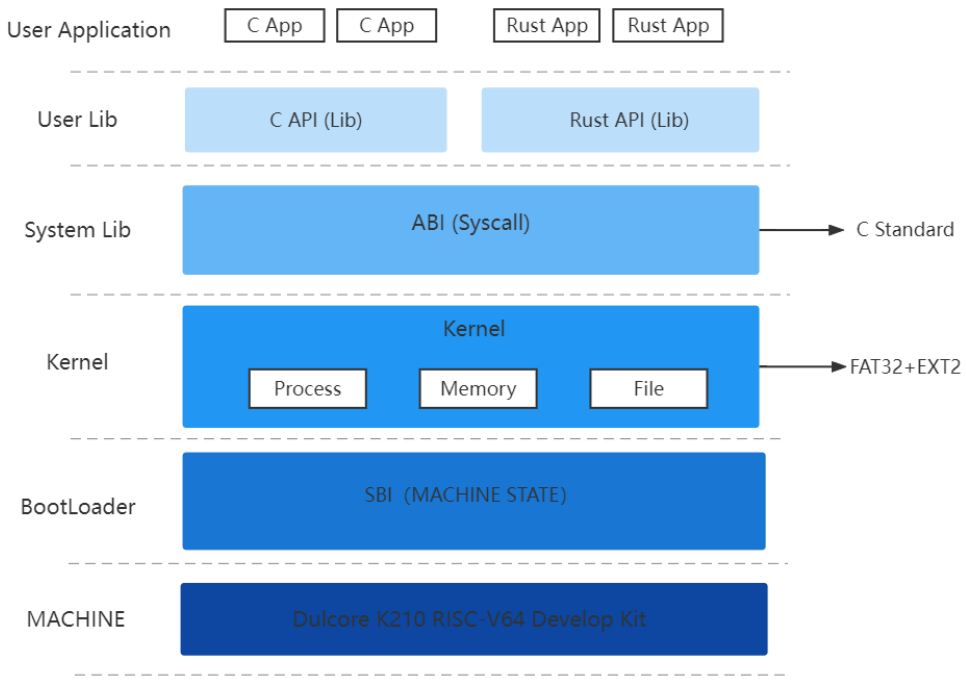
开发的过程借鉴以及架构设计的参考。

为了避免“造轮子”的反工程学开发逻辑，我们使用了清华大学吴亦凡同学的rCoreTutorial-v3代码框架和上一届代表哈尔滨工业大学（深圳）参赛的 UltraOS 队伍的代码框架，并在其上进行开发，使得我们能够快速起步。

现在将目光转向TCore。TCore架构主要包括三个部分：内存管理、进程管理、文件系统。进程管理主要解决初始进程的创建，进程的创建、异常和调度处理。内存管理主要解决页表管理、进程的内存分配和管理、内存布局初始化。文件系统则支持FAT32文件系统，主要解决硬盘驱动访问、文件系统管理问题。同时需要注意到的时，所有的部分都必须支持并发访问的一致性和安全性，也就是支持多核操作。

2 系统设计

2.1 系统整体架构设计



The Deep Sea（Tcore 系统架构图）

TCore 采用了模块化的设计思想。我们将操作系统的构建也分为了三个部分：SBI，内核和标准库，该三个部分分别对应三种特权级运行模式。下面我们简要介绍一下各个功能模块的主要设计思想和架构。

SBI，又称 Supervisor Binary Interface，可以看作为 Machine Mode 提供给 Supervisor Mode 程序的功能接口。

Kernel，称作操作系统内核，是 TCore 的设计核心部分，主要包括有进程管理、内存管理以及文件系统三个部分。

内核不仅需要实现其中断服务机制，还需要系统调用，系统调用组成的接口称作ABI，同时，为了支持C语言和Rust语言的编写，我们同时加入了C和Rust语言的标准库，意味着，我们原生支持跨语言程序编写和运行。

2.2 子模块设计

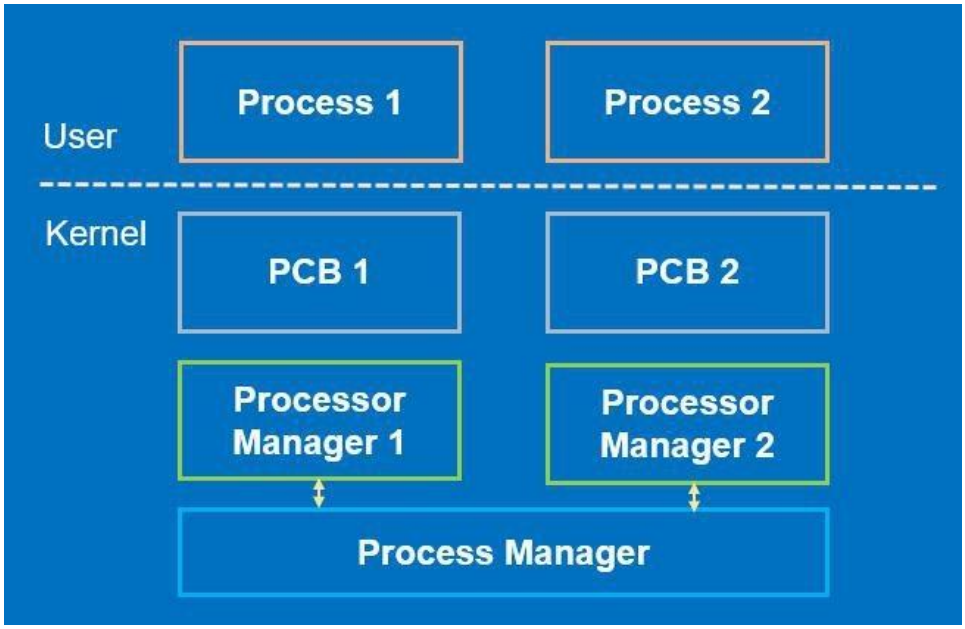
本部分主要介绍三个主要部分：线程管理、内存管理、文件系统。

2.2.1 进程管理

进程管理模块有四大核心模块。我们将进程管理进行层次化抽象区分，这样我们就可以以一个通用的逻辑来对进程实施调度，这种思想将进程管理内不同模块的职责进行了精准的划分。

我们将进程的管理分为四个层次：进程、进程控制块、线程控制块、核调度器、进程管理器。

其示意图如下：



2.2.1.1 进程控制块

进程本身在用户态运行，为对其信息进行抽象，我们采用 Process Control Block（PCB）也就是进程控制块来进行信息管理。进程控制块与进程是一一对应的关系，进一步的，进程控制块内部有着进程所拥有的所有信息与一些关系。

2.2.1.2 核管理器

为了管理进程的信息，并对其进行相应的操作，我们引入了核管理器，或者称为核心管理器，其本质是每一个核都拥有的进程管理器。CPU中每个核心在同一时刻只能运行一个进程，此时核管理器将持有该进程的 PCB 以对其进行管理，与 CPU 核对应。

2.2.1.3 任务管理器

根据进程各个抽象层次的设计，我们遵循其对应关系，也构造了进程控制块、核管理器以及进程管理器三者对应的结构体和方法。但是同时，实际的设计需要一些细节上的变动，以支持完整的实现，于是TCore还构建了PID_ALLOCATOR，该结构用于分配Pid。同时，对于每个结构体的实现，我们不止需要考虑其之间的对应关系和内部的负责区域，还需要考虑到具体的要求，包括进程的复制、替换等逻辑，这些都属于具体实现的内容。

PCB的作用内容大概包括以下几个部分：

- 进程信息：进程上下文、进程之间的关系、进程的标识等。
- 内存信息：进程所占有的内存，包括地址空间，堆，vma等。
- 文件信息：文件描述符表，当前位于的路径等。
- 辅助信息：时钟，资源使用量和限制，定时器，信号等。

2.2.1.4 线程控制块

线程控制块(Task Control Block, TCB)记录线程运行的一些信息，是线程存在的唯一标识。线程是 CPU 进行调度的基本单位，而进程可以被理解为线程的“容器”，不能被直接调度。线程拥有独立的内核栈、用户栈及 Trap 上下文、任务调度上下文，除此以外没有独立的内存空间。

TCB 包括线程的父进程指针、内核栈（这部分不需加锁），TCB 资源结构体、TrapContext 所在物理页帧号、任务上下文、线程状态、退出码。其中，TCB 资源结构体（TaskUserRes）与线程所拥有的的资源（用户栈内存、TrapContext 内存）相绑定：TaskUserRes 创建时，页帧分配器分配内存，PCB 将线程资源内存注册到用户页表中；而 TaskUserRes销毁时，页帧分配器回收内存，用户页表自动删除资源内存的对应表项。

2.2.2 内存管理

内存管理主要分为内核地址空间，用户地址空间以及页表结构。其中，内核用户空间主要负责系统的启动、进程之间的切换以及内核态系统调用过程中的堆栈功

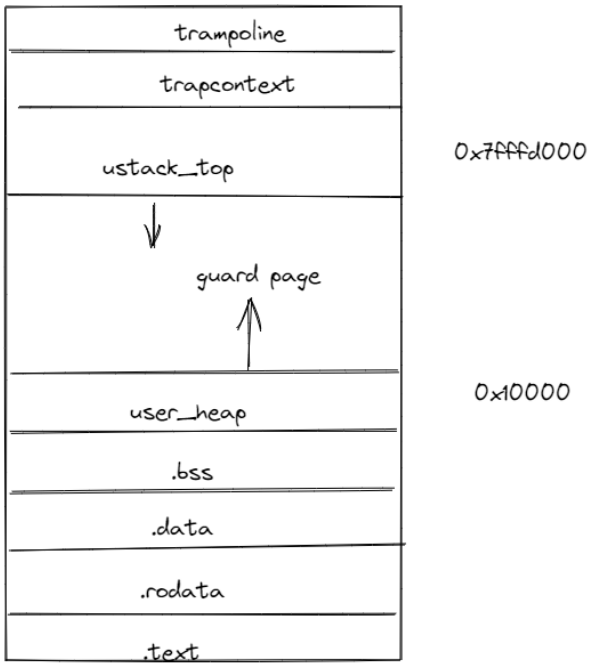
能，而用户栈主要负责对于用户应用程序的载入，运行中的数据存储以及动态的内存分配。

K210平台的最大问题就是内存不足，除去SBI所管理的空间，操作系统内核加上其所管理的空间只有6MB。因此我们在内存管理部分进行了许多优化。

同时，TCore为了达到以上所述的目标，设计了虚拟地址空间vma、页表、地址空间、内核堆、页帧管理器。

- 地址空间：描述了整个进程的所有拥有的地址空间以及其信息。包括了页表、vma以及页表和所拥有的页帧。
- 页表：管理进程的页表以及其中的映射关系。可以管理页表项，也可以利用它做地址的软件翻译。
- 页帧管理器：用来管理未被内核程序所占用的空闲物理页帧，包括页帧的释放、的获取等。
- Vma：主要为mmap而生，管理了线性虚拟地址空间的映射关系。
- 内核堆：管理内核的堆分配和异常处理。

TCore的内存布局如下图所示：

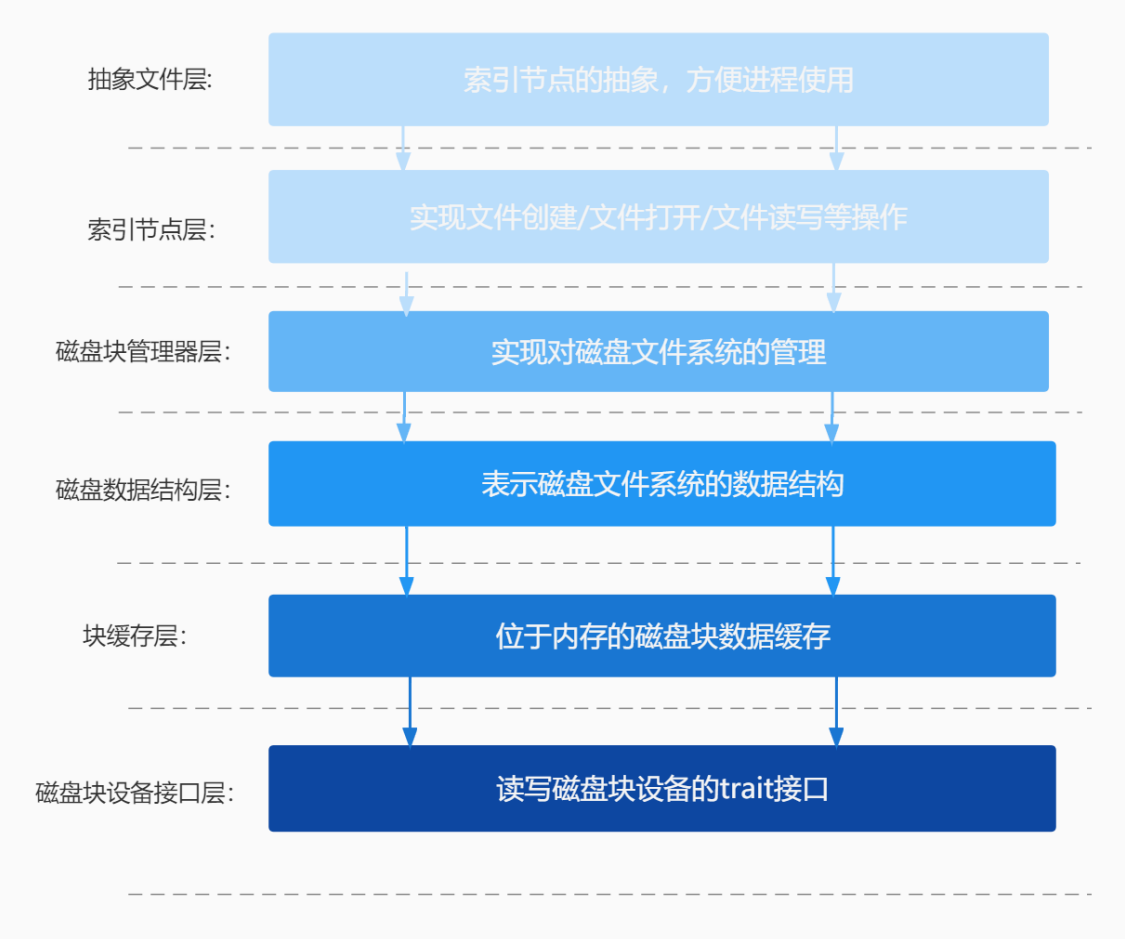


2.2.3 文件系统及设备

TCore贯彻了UNIX的“一切皆文件”的设计理念。我们将文件分为实际文件和抽象文件两大类。前者即真正意义上的文件，后者可以是具备文件功能的系统对象，例如设备、管道等。为此，我们的文件系统分为两个部分：磁盘文件系统以及内核虚拟文件系统。

TCore 支持FAT32文件系统。在内核中，我们对磁盘文件进行了高度抽象，只要文件系统实现了所需的 Trait，都可以与内核对接使用。结构设计

TCore的文件系统的整体结构如下：



磁盘文件系统主要由块设备接口层、块缓存层、磁盘数据结构层、磁盘块管理器层、索引节点层和抽象文件层构成。

2.2.3.1 各模块介绍

(1) 抽象文件层

内核虚拟文件系统统筹了所有类型的文件，抽象出统一的接口，主要面向系统调用。通过这些接口，能提高代码复用率，又具备很强的扩展粘合性。

(2) 设备管理

设备的多样性往往是系统走向的实用关键。因为教学主体的性质，TCore并没有实现丰富的设备驱动。设备管理是内核与设备驱动之间的桥梁。对于内核，其需要提供相应 Trait 使得驱动获取来自用户的控制信息；对于驱动，其需要提供接口以便内核控制和调度。

(3) 块设备接口层

为最底层就是对块设备的访问操作接口。在 Tcore/codes/simple_fat32/src/block_dev.rs 中，可以看到 BlockDevice trait ，它代表了一个抽象块设备的接口，该 trait 仅需求两个函数 read_block 和 write_block ，分别代表将数据从块设备读到内存缓冲区中，或者将数据从内存缓冲区写回到块设备中，数据需要以块为单位进行读写。simple_fat32 库的使用者（如操作系统内核）需要实现块设备驱动程序，并实现 BlockDevice trait 以提供给 simple_fat32 库使用，这样 simple_fat32 库就与一个具体的执行环境对接起来了。至于为什么块设备层位于 simple_fat32 的最底层，那是因为文件系统仅仅是在块设备上存储的稍微复杂一点的数据。无论对文件系统的操作如何复杂，从块设备的角度看，这些操作终究可以被分解成若干次基本的块读写操作。

(4) 块缓存层

I/O是影响性能的关键。为了提升性能，需要利用局部性原理设计缓存以尽可能减少I/O读写次数。

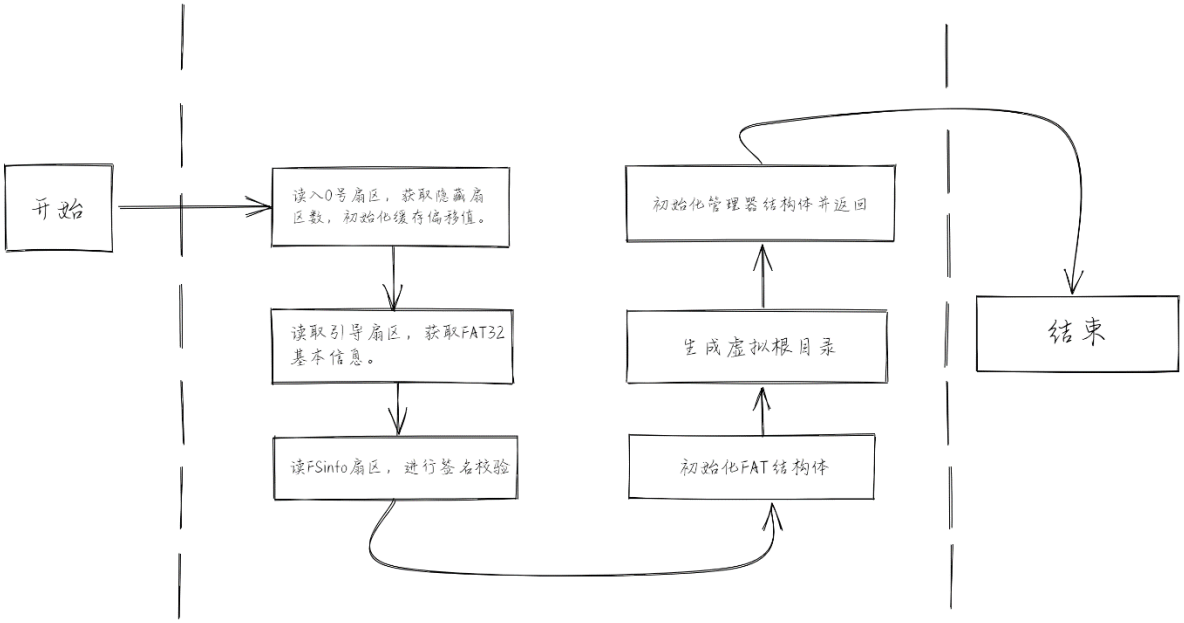
使用磁盘缓存的另一个好处是可以屏蔽具体的块读写细节，以此提升效率。在我们的设计中，上层模块可以直接向缓存索取需要的块，具体的读写、替换过程将不再向下传递而是交由缓存完成。

(5) 磁盘数据结构层

本层真正开始对文件系统进行组织。FAT32有许多重要的磁盘数据结构。他们由不同的字段构成，存储文件系统的信息，部分字段也存在特定的取值。磁盘数据结构层的工作就是组织这些数据结构，并为上层提供便捷的接口以获取或修改信息。

(6) 磁盘块管理层

文件系统管理器层是整个文件系统的核心，其负责文件系统的启动、整体结构的组织、重要信息的维护、簇的分配与回收，以及一些的实用的计算工具。



(7) 索引节点层

索引节点层主要负责为内核提供接口，屏蔽文件系统的内部细节，首要任务就是实现复杂的功能。

```
// 此inode实际被当作文件
pub struct OSInode {
    readable: bool,
    writable: bool,
    //fd_cloexec: bool,
    inner: Mutex<OSInodeInner>,
}
```

3 系统实现

TCore 采用Rust语言进行具体的实现，使用rCoreTutorial-v3作为其主体框架，

RustSBI 作为底层硬件抽象实现。

3.1 进程管理

3.1.1 进程控制器

```
pub struct TaskManager {
    ready_queue: VecDeque<Arc<TaskControlBlock>>,
}
```

在这里，我们可以根据自己的判断实现各种调度算法，但是其必要 Trait 只有三个：Init、Add、Take。进程控制器可以采用其他的算法来进行进程调度优先级。

3.1.2 核管理器

核管理器的进程调度。它将不停的寻找下一个进程，如果获取成功下一进程，则切换栈至对应进程的内核栈，如果没有寻找到可切换进程（没有就绪进程），就返回至原有进程的内核栈。

3.1.3 进程控制块

进程控制块主要负责进程的控制。进程的信息包括有：Trap上下文指针和对应页帧、堆信息、段信息（包括页表）、进程父子关系、进程当前状态以及退出码、进程的文件系统路径和所拥有的文件资源。

其内容可总结为：

- 进程信息：进程上下文、进程之间的关系、进程的标识等。
- 内存信息：进程所占有的内存，包括地址空间，堆，vma等。
- 文件信息：文件描述符表，当前位于的路径等。

```rust
pub struct ProcessControlBlockInner {
    pub is_zombie: bool,
    pub memory_set: MemorySet,
    pub base_size: usize,
    pub heap_start: usize,
    pub heap_pt: usize,
    pub mmap_area: MmapArea,
    pub parent: Option<Weak<ProcessControlBlock>>,
    pub children: Vec<Arc<ProcessControlBlock>>,
    pub exit_code: i32,
    pub fd_table: FdTable,
    pub signals: Signals,
    pub siginfo: SigInfo,
    pub tasks: Vec<Option<Arc<TaskControlBlock>>>,
    pub task_res_allocator: RecycleAllocator,
    pub current_path: String,
}
```

3.1.4 线程控制块

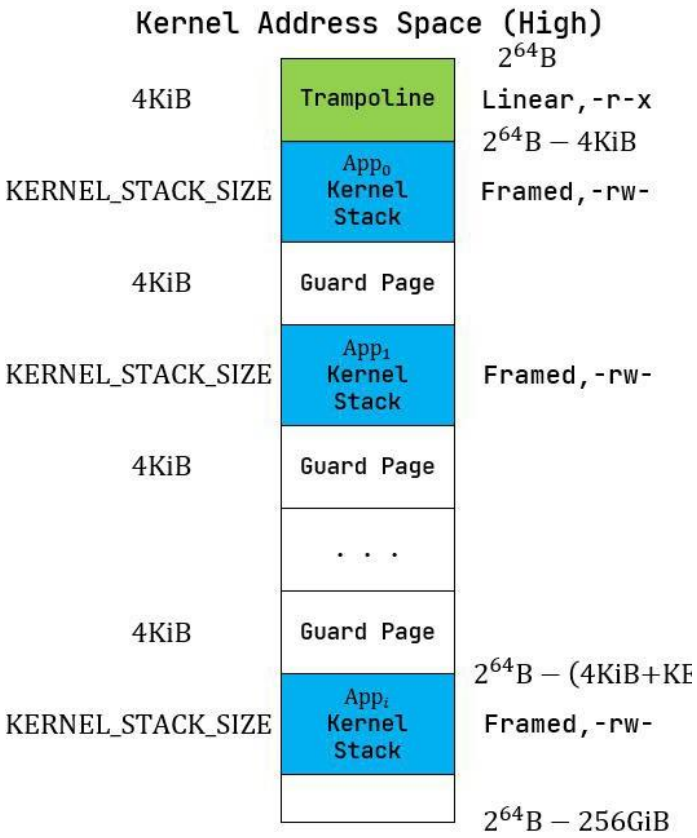
```rust
pub struct TaskControlBlock {
// immutable
pub process: Weak<ProcessControlBlock>,
pub kstack: KernelStack,
// mutable
inner: Mutex<TaskControlBlockInner>,
}

pub struct TaskControlBlockInner {
    pub res: Option<TaskUserRes>,
    pub trap_cx_ppn: PhysPageNum,
    pub task_cx: TaskContext,
    pub task_status: TaskStatus,
    pub exit_code: Option<i32>,

}
```

3.2 内存管理

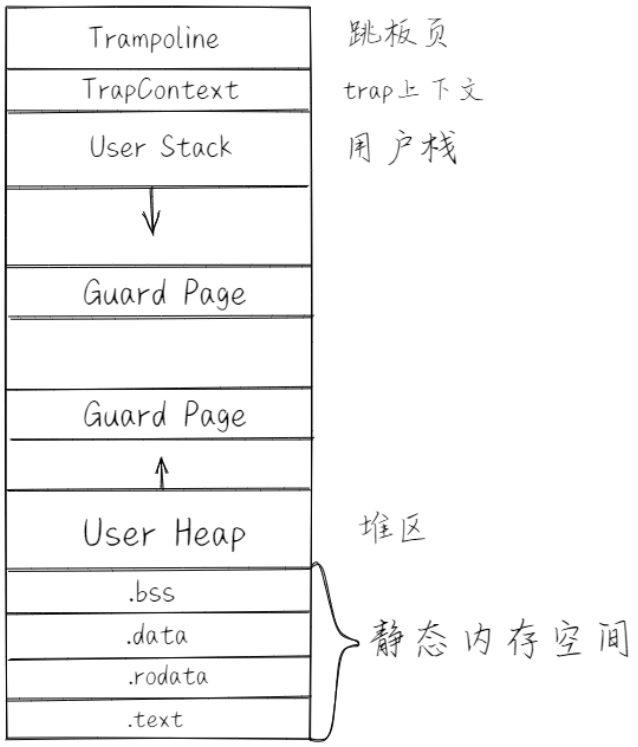
3.2.1 内核地址空间



从操作系统启动开始，内核地址空间就需要载入启动所需要的数据，并且载入0号初始进程 initproc 以及用户命令行进程usershell 。在运行程序的过程中，内核地址空间划分成多个应用程序各自对用的内核栈空间，并且通过 Guard Page 机制来分隔开——两个内核栈之间会预留一个 保护页面 (Guard Page) ，它是内核地址空间中的空洞，多级页表中并不存在与它相关的映射。它的意义在于当内核栈空间不足（如调用层数过多或死递归）的时候，代码会尝试访问空洞区域内的虚拟地址，然而它无法在多级页表中找到映射，便会触发异常，此时控制权会交给内核 trap handler 函数进行异常处理。

3.2.1.1 用户地址空间

右图为用户地址空间分布。



用户地址主要通过 MemorySet 的数据结构来管理，并且通过页表与物理地址空间映射，每次新建一个用户应用程序对应的进程时，都会新建一个专属的 MemorySet，并且通过 from_elf 接口从二进制文件中读取数据。MemorySet 下面包含用户地址中的每一段空间（data,text,heap,stack,…），并且分别用一个单独的数据结构表示，在新建时都会通过 map 函数在页表中建立映射。

```rust
pub struct MemorySet {
    page_table: PageTable,
    areas: Vec<MapArea>,
    chunks: ChunkArea,
    stack_chunks: ChunkArea,
    mmap_chunks: Vec<ChunkArea>,
}
```

跳板在用户地址空间的最顶部，将虚地址和实地址联系起来（跳板的虚地址和实地址对于每一个进程都是相同的），用于在不同的用户进程空间中进行切换，并且可以在trap陷入的时候，将 TrapContext 保存在此处。

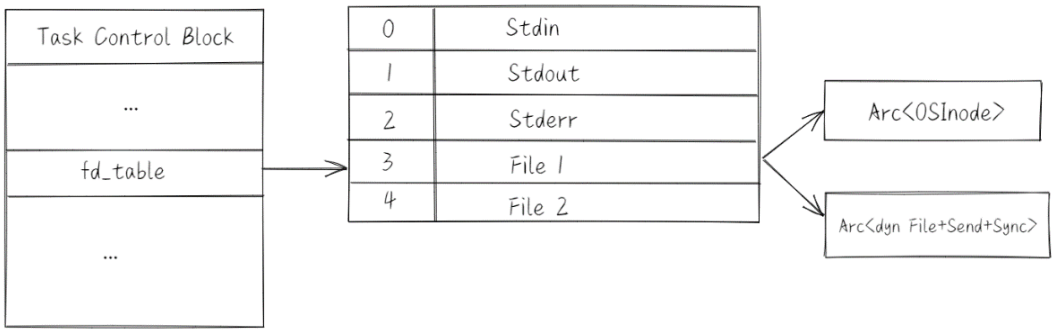
4.3 文件系统及设备

因为文件系统与设备管理系统十分庞大，这里仅介绍各层模块核心功能实现。

4.3.1 内核抽象文件系统

在内核中，我们定义了OSInode结构体来表示文件，

进程维护的文件如下所示：



可以看出，抽象类文件还实现了Send与Sync接口，这是Rust为了保障安全跨线程共享而提供的特性。

4.3.2 设备管理

系统通常使用设备树获取设备信息，其由Bootloader提供。但因为TCore使用的Bootloader未提供设备树，同时考虑到效率问题，我们暂不使用设备树，只调用平台相关的库对需要的硬件实现驱动，使用表的方式检索。

设备管理的目的是为内核与驱动程序提供桥梁。为了可拓展性，TCore希望将一切内核无关的程序解耦，这其中就包括了驱动程序。因此我们要定义统一的接口，便于内核和驱动交互。

通常情况下，用户进程通过sys_ioctl系统调用对IO设备进行控制。sys_ioctl的控制对象为文件，为此我们将所有需要的设备都抽象为存储于内存的虚拟文件。具体实现上，我们为设备实现了File trait，同时为File trait增加 ioctl 方法。设备驱动可以根据需要实现 File trait 定义的部分方法。系统启动时，将各个设备初始化并装入设备表以进行维护。

sys_ioctl 的参数为文件描述符、控制命令，以及控制参数。实现的难点在于，控制参数可能是一个整数，也可能是一个结构体。这就导致，内核不可能针对不同类型的控制区别管理，而应当把一切控制交由目标设备驱动程序进行管理，这也是必须实现解耦的原因。为此，我们的设计目标为：内核不需要了解用户对设备进行什么控制，也不需要了解驱动程序如何实现控制；驱动程序则不需要接触用户的地址

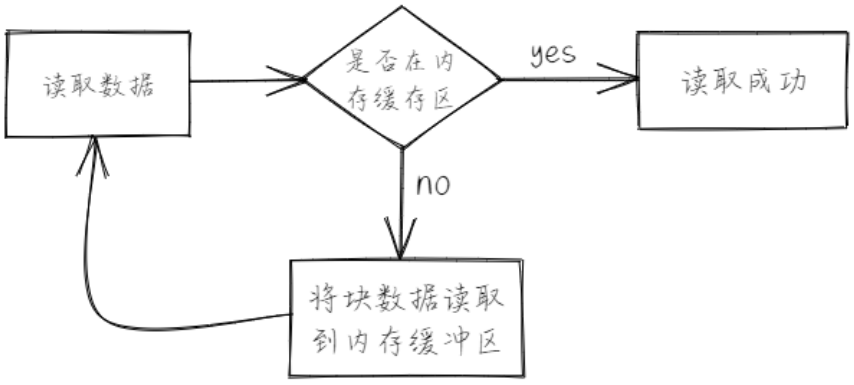
空间，只能得到自身需要的参数。在这种目标下，内核只负责数据转发，用户和设备的行为完全透明。

4.3.3 块设备接口层

块设备接口层定义了块设备驱动需要实现的接口，通过该层，文件系统访问块设备将是透明的。我们使用Rust的Trait定义了块设备的抽象接口，其包含read_block和write_block两个方法。对于设备驱动，为了提高开发的效率，我们使用了开源项目rCore-Tutorial的SD卡驱动。

4.3.4 块缓存层

尽管在操作系统的最底层（即`/drivers/block/virtio_blk.rs`中的块设备驱动程序）已经有了对块设备的读写能力，但从编程方便/正确性和读写性能的角度来看，仅有块读写这么基础的底层接口是不足以实现高效的文件系统。比如，某应用将一个块的内容读到内存缓冲区，对缓冲区进行修改，并尚未写回块设备时，如果另外一个应用再次将该块的内容读到另一个缓冲区，而不是使用已有的缓冲区，这将会造成数据不一致问题。此外还有可能增加很多不必要的块读写次数，大幅降低文件系统的性能。因此，通过程序自动而非程序员手动地对块缓冲区进行统一管理也就很必要了，该机制被我们抽象为 simple_fat32 自底向上的第二层，即块缓存层。在 `simple_fat32/src/block_cache.rs` 中， `BlockCache` 代表一个被我们管理起来的块缓冲区，它包含块数据内容以及块的编号等信息。当它被创建的时候，将触发一次 `read_block` 将数据从块设备读到它的缓冲区中。接下来只要它驻留在内存中，便可保证对于同一个块的所有操作都会直接在它的缓冲区中进行而无需额外的 `read_block` 。块缓存管理器 `BlockManager` 在内存中管理有限个 `BlockCache` 并实现了类似 FIFO 的缓存替换算法，当一个块缓存被换出的时候视情况可能调用 `write_block` 将缓冲区数据写回块设备。总之，块缓存层对上提供 `get_block_cache` 接口来屏蔽掉相关细节，从而可以向上层子模块提供透明读写数据块的服务。



4.3.5 磁盘数据结构层

磁盘数据结构层用于组织FAT32特有的数据结构。需要注意的是，K210要求对齐读写，即某类型变量的地址必须以该类型大小对齐。通过查看FAT32文档可知，引导扇区的一些字段是不对齐的，例如半字类型的字段可能与字节对齐。对于这些字段，我们以字节数组的形式读入，设计接口封装其读写以使上层直接使用字段类型控制。

首先介绍引导扇区和扩展引导扇区。他们给出了块设备的基本信息，例如扇区大小数量、簇大小、版本、校验签名、保留扇区数目、文件信息扇区所在位置等。如果设备进行了分区，引导扇区还包含了第一个分区的起始扇区。引导扇区的信息通常不需要修改，他们只负责在文件系统其动时进行校验并为管理层提供相关的信息。事实上，这两个扇区也仅在文件系统起动时会读入。因为这两个扇区的字段较多，我们定义了完整的结构体，将所有字段作为成员，以与磁盘数据一一对应。使用时需从cache读入对应的块，然后将对块的引用转换为对应结构体的引用。其次是文件系统信息扇区。该扇区有五个重要字段，前两个字段为校验签名，第三个为FS剩余簇数、第四个字段为起始空闲簇、第五个字段为校验签名。显然，第三、四个字段经常需要读写。该扇区的有效字段分布于扇区首尾，中间均为无效字节，为了节约空间，不必建立包含整个扇区字段成员的结构体。我们定义了一个 FSInfo结构体，其只包含扇区号这一个成员，但是实现了丰富的接口，包括签名校验，簇信息字段的读写。只保留扇区号一个成员是合理的，因为扇区的读写经由缓存，而缓存的提供的接口只需要扇区号和偏移信息。

然后是目录项结构。目录项结构复杂，字段较多，因此依然实现了完整的结构体以与磁盘数据一一对应。FAT32的目录项长32字节，包括长文件名目录项和短文件名目录项两种类型，当文件名较长时，采用多个长名目录项加一个短名目录项的

形式存储。文件的所有信息都存储于短名目录项，包括名称、扩展、属性、创建/访问/修改时间、大小、起始簇号等，因此我们将其作为文件的访问入口。对于长短名目录项，我们均设计了丰富的接口以进行封装，这样上层在访问时可以以直观的数据类型读写信息，而不必考虑复杂的内部结构。

短名目录项比较简单，其用11个字节存储文件名，包括8字节名称和3字节扩展名，使用ASCII编码。长目录项则使用分散的26个字节存储文件名，使用Unicode编码。为了屏蔽内部的存储细节，我们实现了get_name接口，其以字符串的形式返回目录项中的文件名，便于其他模块使用。

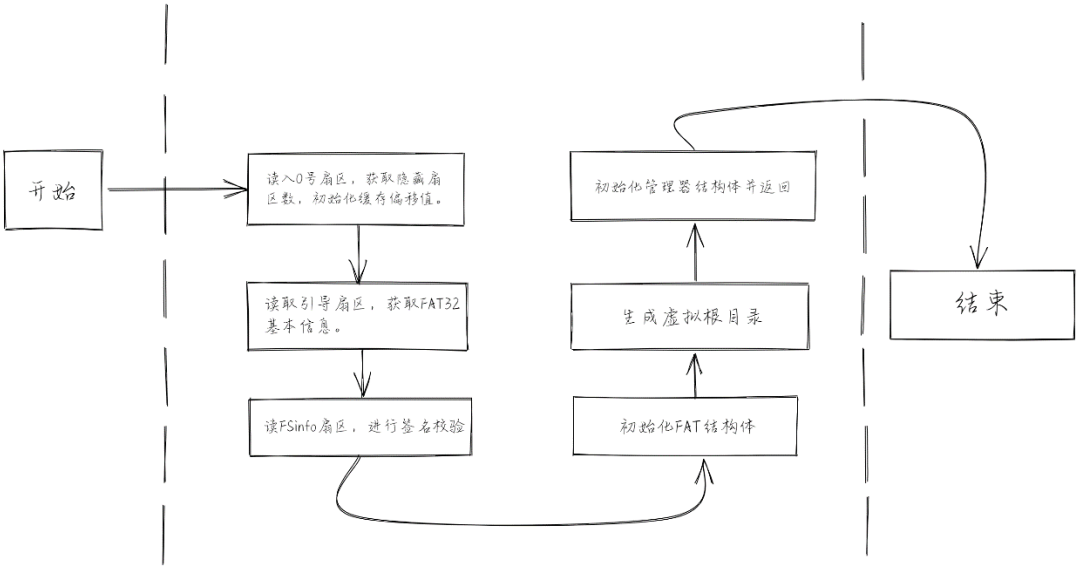
最后是FAT，即文件分配表。FAT是FAT类文件系统的核心。在FAT32文件系统中，文件以簇为单位通过链式结构组织，各簇的下一簇号即存储于FAT中。FAT32拥有2个FAT，FAT2作为FAT1的备份，因此写操作需要同步进行，而读操作发现故障时需要及时换到另一FAT。在实现上，我们定义了一个FAT结构体，其包含两个成员，分别是FAT1和FAT2的起始扇区号。由此，对于FAT的一切操作都直接通过缓存进行。

4.3.6 磁盘块管理层

管理器的结构体定义如下：

FAT32Manager	
成员	描述
block_device	块设备的引用，使用Rust的动态分配以支持不同设备
fsinfo	文件系统信息扇区的引用
sectors_per_cluster	每个簇的扇区数
bytes_per_sector	每个扇区的字节数
fat	FAT的引用
root_sec	更目录所在簇号
vroot_dirent	虚拟根目录项。根目录无目录项，引入以与其他文件一致

管理器负责的首要任务就是初始化文件系统，具体的启动流程如下：



4.3.7 索引节点层

在索引节点层中，我们将文件抽象为虚拟文件，只暴露出提供的接口。

4 总结与展望

4.1 工作总结

(1) SPDK的探索：尽管在开发和尝试中面对这庞大的源码库我们并没有做出许多有意义的代码实例但是我想只要是一次探索哪怕是失败的也知道作为一次经验和教训给后续的人留下些什么。我们在开发中将学习文档和设计的进展进行了汇总方便大家继续探索SPDK的特性最快的入门方式是公众号DPDK和SPDK开源社区。我们希望大家能够收获有益的部分。同时也希望大家探索RUST与SPDK的绑定仓库（我们也做了一个链接）。

(2) 我们在开发中可以说是从零开始团队的成员除了简单的RUST基础和OS课程经验基本都是从零开始这也意味着我们更加适合上手我们也整理了不少好用的“芝士”链接和强大的开发工具希望可以免除入门阶段的一些烦恼。

(3) 虽然受限于有限的时间和有限的能力但是这个过程中我们感受到了系统培养圈子良好的氛围，在我们的实践中rcore-Tutorial-V3的主要开发和维护者吴一凡的细心答疑，哈工深李程浩同学对于多核知识的无私分享和不厌其烦的答疑，王润基对于第二阶段musl-libc的详细启发。希望大家也能够在这个过程中感受OS的魅力。

4.2 未来展望

（1） 比赛之外，设计教学，带动本校的学习氛围，构建适合实验操作的代码，进行传承下去。

（2） 目前无法实时监控内核的进程调度状态，故多核调试上存在一定困难，想在未来实现类似于 Linux下的 Htop 工具

（3） 操作系统多核支持鲁棒性：对于多核操作系统，我们在正常的使用之中以通过测试用例为基准已经能够实现较好的得分，但是我们还是发现，在极端情况下或者说是压力测试下，依然会出现非预期效果，这可能值得我们警惕，同时对 TCore 进行更深入的改进，以支持更稳定的运行。我们也希望在今年更多多核队伍的基础上学习到宝贵经验。

（4） SPDK线程机制的继续探索力争能够实现一个简单的展示机制是的这层抽

象可以便于提供教学和理解。

（5） GUI支持：虽然操作系统的内存较小，但是我们可以利用板载16MB的SSD进行页面换入换出来提升内存的大小。这样我们就有希望支持GUI，能够实现基本的图像库。