哈爾濱工業大學（深圳）
HARBIN INSTITUTE OF TECHNOLOGY

# Rust 编写的基于 RISC-V64 的多核操作系统 UltraOS

项目成员：李程浩（组长）
宫浩辰
任翔宇

指导老师：夏文、江仲鸣

# 目 录

# 1 概述

UltraOS为用Rust编写的基于RISC-V64的多核操作系统，并且能够运行在实体裸机K210平台（包含有400MHZ双核RISC-V处理器）以及qemu模拟器上。

## 1.1 项目背景及意义

从哈尔滨工业大学（深圳）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，都显得相对薄弱。作为计算机专业的三大金刚之一的操作系统，是理解几乎程序运行以及硬件资源管理的核心，从教学上来说，正是因为其管理机制的复杂，操作系统成为了实践上和理论上的“坎”。

横向比较来看，华中科技大学的学生已经创建了属于自己的团队“无相之风”。专注于计算机操作系统方向的研究，包括硬件抽象化、设备驱动、新型操作系统实现和研究等深入探索的项目。再看清华大学，有着rCore，uCore以及zCore三座大山，分别在语言、操作系统基石、微内核上进行了探索，并且也成功将其转化为实验指导以及具体的教学友好型代码。这两所学校的本科操作系统教育不仅在基础的操作系统实验教学上有着自己的独有模式，并且进行了深入的探索。

同时，还注意到而且学校的本科生还在自己感兴趣的方向进行持续的探索，并且组成了一定的研究小团体。综上，我们认为哈尔滨工业大学（深圳）的计算机系统人才的关键在于三点：丰富理论支持（针对课程）、自有实验设计（针对课程实验）、前沿探索（有对应领域的指导老师进行深入挖掘）。

我们着力解决第二点：自有实验设计。于是我们立足于在实机上开发操作系统，以支持自由课程实验的设计，而不是受制于人。这样，在课程任务设计上，能够有更大的自由度，也能够有更大的自由空间和进一步深化了解的空间。

同时我们还希望能够以Github项目组、课程实验、计算机系统小组等形式传承以及迭代下去，这对于人才的培养都是必须的。

## 1.2 国内外研究概况

对于工业或者学术上来说，都会将操作系统从最宽泛的角度分为两方面：宏内核和微内核。UltraOS所采取的设计方式为宏内核，但是实际上，由于其简洁性的

设计理念，以及其功能的精简性，可以在在框架下转化为微内核。下面介绍微内核和宏内核的国内外研究现状。

对于微内核，最为著名的是L4阶段，也就是当前最前沿的阶段。我们先讨论L4之前的阶段。

微内核一开始就是为其出色的开发设计和鲁棒性备受期待，但是在一个硬件性能不出色的时代，并没有能够取得很好的影响力。微内核的本质，是将大部分的内核通信，放在用户态，以将尽可能多的内核部件放在用户程序。这样，内核程序就能够在出现故障的时候被微内核重启，需要交流的时候就使用进程间通信技术。这样看起来很美好，但是面临着两大问题：频繁的进程间通信大大降低了性能，用户程序可能会出现新的安全问题。

微内核L4版本之前，这一直都是一个不小的问题。L4采用了新的架构，将进程间通信，也就是IPC，提升了10至20倍，从此开辟了新纪元。不仅如此，在L4的演变过程中，有专家使用了逻辑推理，证明了操作系统是完全安全的，而在宏内核中，这是不可思议的证明方法。从此，微内核在安全性和性能上，都有了相当的突破，正式向宏内核提出了挑战。

在L4系列的系统之中，产生了不同的应用范围对应的不同微内核版本，有嵌入式的，有为安全性而生的。但是其中相当具有影响力的还是seL4，它也是经过了演绎推理的安全性验证，同时相比同类微内核有着非常突出的性能优势。同时，谷歌近期推出了基于Zircon微内核的Fuchsia操作系统，也在业内取得了较大的影响力。

对于宏内核，最著名的就是开源操作系统Linux。Linux由C语言编写，拥有非常强悍的功能和性能，备受广大程序员喜爱。其问题在与潜在的安全和鲁棒性问题，由于内核代码数量十分庞大，很难系统的对其进行检测，只能通过发现再消除的方式进行避免，无法从设计上规避所有的安全问题，系统也会偶见崩溃状况。

实际上，在工业上进行应用的操作系统大多不是简单的宏内核和微内核，而是两者融合的混合内核。相对于微内核来说，在内核加入了更多重要的内核程序，以保证性能和安全性。但是相对于宏内核来说，又将一些内核级的程序放置在用户态。混合内核的代表为基于XNU混合内核的MacOS操作系统以及Windows系统，前者由于其架构优势，很少受到电脑病毒的侵袭。

## 1.3 项目的主要工作

UltraOS致力于开发一个基于RISC-V64的多核操作系统，并且能够运行在实体裸机K210平台（包含有400MHZ双核RISC-V处理器）上。这里需要注意的是，UltraOS的最终目标是教学向，也就是提供一个探索的先例，使得在之后的操作系统人才的培养上能够出一份力。这意味着我们既需要朝着工业界努力，也要去除一些虽然利于工程上应用但是却不利于理解核心概念的枝干。同时，在比赛的适配过程中，我们也应该选取尽可能具有普遍性的方案。**总之，性能、功能都不是最终的目标，他们或许在我们的开发过程中成为中间目标，但UltraOS的终点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（2021.03.26版本）框架，并在其上进行开发，使得我们能够快速起步，并从上至下逐步完善操作系统功能，从用户程序支持到硬件抽象的至上而下的改进，包括新增功能、鲁棒性增强、提升性能、增强兼容性等，最终实现UltraOS的开发。

通常，对于操作系统的讨论都集中在内核的设计上，但是正式的操作系统需要更多的、更完整的支持。其中最重要的两个部分是：硬件抽象化接口SBI和用户抽象化接口ABI。前者用来实现硬件平台的无关性，包括中断代理、无效指令软处理等操作，这样可以使得操作系统内核无需过度关心硬件平台的差异，同时还能在软件层面扩展硬件层面的功能缺失，延长硬件平台的生命周期。后者则使得用户程序无需过度关心操作系统具体的差异，使得其程序能够在不同的操作系统上运行。

针对该目标，我们需要对其进行具体的设计和实现。针对K210的硬件平台，我们使用了华中科技大学洛佳同学开发的RustSBI项目，来实现硬件的抽象化。同时我们也对其进行了修改，以从底层上更好的实现对操作系统功能的支持。而对于用户支持，我们构建了Rust和C的标准库，使得用户能够在标准库的帮助之下编写对应的程序，并且无需过多关注操作系统的接口。

现在将目光转向内核。内核架构主要包括三个部分：进程管理、内存管理、文件系统。进程管理主要解决初始进程的创建，进程的维护、创建、调度和异常处理。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EXT2-like和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 需求分析

UltraOS致力于实现一个精简的多核操作系统，使得其具有良好的拓展性，同时又能够在重要功能上进行支持。基于此目标，UltraOS将重心放在了功能上，而非性能。

对于功能上，UltraOS的所有功能必须支持K210实机双核运行。同时，为了能够参加第一届全国大学生计算机系统能力培养大赛操作系统赛道中内核实现组，同时兼顾基本架构的系统调用实现，我们最后得出应该实现的系统调用如下：

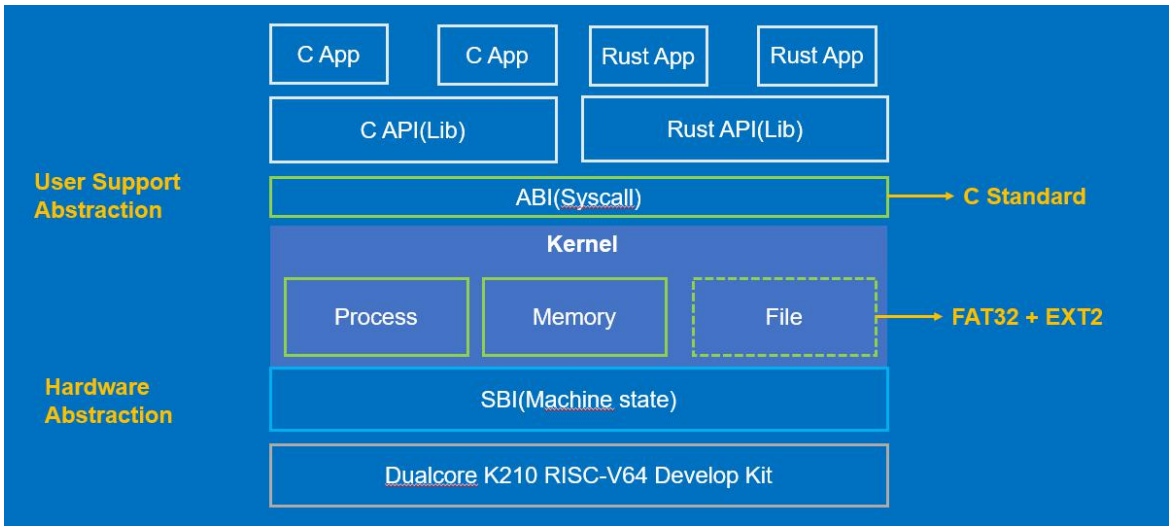
```
const SYSCALL_GETCWD: usize = 17;
const SYSCALL_DUP: usize = 23;
const SYSCALL_DUP3:usize = 24;
const SYSCALL_FCNTL:usize = 25;
const SYSCALL_IOCTL:usize = 29;
const SYSCALL_MKDIRAT: usize = 34;
const SYSCALL_UNLINKAT: usize = 35;
const SYSCALL_LINKAT: usize = 37;
const SYSCALL_UMOUNT2: usize = 39;
const SYSCALL_MOUNT: usize = 40;
const SYSCALL_FACCESSAT: usize = 48;
const SYSCALL_CHDIR: usize = 49;
const SYSCALL_OPENAT: usize = 56;
const SYSCALL_CLOSE: usize = 57;
const SYSCALL_PIPE: usize = 59;
const SYSCALL_GETDENTS64: usize = 61;
const SYSCALL_LSEEK: usize = 62;
const SYSCALL_READ: usize = 63;
const SYSCALL_WRITE: usize = 64;
const SYSCALL_WRITEV: usize = 66;
const SYSCALL_SENDFILE: usize = 71;
const SYSCALL_PSELECT6: usize = 72;
const SYSCALL_READLINKAT: usize = 78;
const SYSCALL_NEW_FSTATAT: usize = 79;
const SYSCALL_FSTAT:usize = 80;
const SYSCALL_FSYNC:usize = 82;
const SYSCALL_UTIMENSAT:usize = 88;
```

```
const SYSCALL_EXIT: usize = 93;
const SYSCALL_EXIT_GRUOP: usize = 94;
const SYSCALL_SET_TID_ADDRESS: usize = 96;
const SYSCALL_NANOSLEEP: usize = 101;
const SYSCALL_GETITIMER: usize = 102;
const SYSCALL_SETITIMER: usize = 103;
const SYSCALL_CLOCK_GETTIME: usize = 113;
const SYSCALL_YIELD: usize = 124;
const SYSCALL_KILL: usize = 129;
const SYSCALL_SIGACTION: usize = 134;
const SYSCALL_SIGRETURN: usize = 139;
const SYSCALL_TIMES: usize = 153;
const SYSCALL_UNAME: usize = 160;
const SYSCALL_GETRUSAGE: usize = 165;
const SYSCALL_GET_TIME_OF_DAY: usize = 169;
const SYSCALL_GETPID: usize = 172;
const SYSCALL_GETPPID: usize = 173;
const SYSCALL_GETUID: usize = 174;
const SYSCALL_GETEUID: usize = 175;
const SYSCALL_GETGID: usize = 176;
const SYSCALL_GETEGID: usize = 177;
const SYSCALL_GETTID: usize = 177;
const SYSCALL_SBRK: usize = 213;
const SYSCALL_BRK: usize = 214;
const SYSCALL_MUNMAP: usize = 215;
const SYSCALL_CLONE: usize = 220;
const SYSCALL_EXEC: usize = 221;
const SYSCALL_MMAP: usize = 222;
const SYSCALL_MPROTECT: usize = 226;
const SYSCALL_WAIT4: usize = 260;
const SYSCALL_PRLIMIT: usize = 261;
const SYSCALL_RENAMEAT2: usize = 276;
```

对于评测系统的支持，我们还需要支持非在线库项目依赖、非外部设备依赖自启动、测评程序自动运行（包括串行和并行）等功能。

# 3 系统设计

## 3.1 系统整体架构设计




UltraOS系统简要架构

UltraOS采用了模块化的设计思想，以保证开发速度以及最终成果的鲁棒性。同时，模块化的开发也考虑到了RISC-V架构本身对于特权级的规定。该指令集通常将程序运行的特权级分为三个部分：Machine Mode， Supervisor Mode and User Mode。对应的程序执行权限从高到低分布。根据这样的特性，我们将操作系统的构建也分为了三个部分：SBI，kernel and Standard Library，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。该接口力图实现硬件抽象化，包括有CPU的指令集，以及计算机所拥有的其他硬件资源，并且抽象化之后保持行业内的规范，将硬件上表现不一致的部分，通过同等抽象，最终可以使得上层软件无需过多关心硬件的不一致性以及硬件的细节，使得上层软件能够拥有很好的逻辑性以及兼容性。在操作系统中，这一层又可以称作HAL(Hardware Abstraction Layer)。RustSBI在该层，主要实现了stdio, hsm(hart state management),ipi(inter-processor interrupt support), reset, timer五个部分。

Kernel，称作操作系统内核，是UltraOS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。进程管理主要实现对于用户进程的控制，包括进

程间的切换，进程状态的控制，进程对内核服务请求的应答，硬件中断的设置与处理等。而内存管理主要实现用户程序的内存隔离，包括用户动态堆分配、页表管理、页帧分配以及用户程序的内存数据读取与搭建等。文件系统则主要面向硬盘等IO设备，我们实现了EXT2-like和FAT32文件系统，以及以内存为介质的虚拟文件系统。前者主要功能为硬盘的增删改查以及硬盘的初始化和文件系统检测，硬盘驱动及抽象化以及文件系统对内核的抽象，文件系统的访问、修改、更新等；后者的功能则为将实体文件、设备、管道等统一抽象为文件，便于内核管理。

内核不仅需要实现其中断服务机制，还需要同步服务机制，也就是系统调用，系统调用组成的接口称作ABI，我们所构建的ABI满足C语言规定调用标准，为用户提供用户服务抽象。但是直接裸露的ABI不便于用户使用，因此我们还构建了用户标准库。同时，为了支持C语言和Rust语言的编写，我们同时加入了C语言和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。

## 3.2 子模块设计

本部分主要介绍三个主要部分：进程管理、内存管理、文件系统。

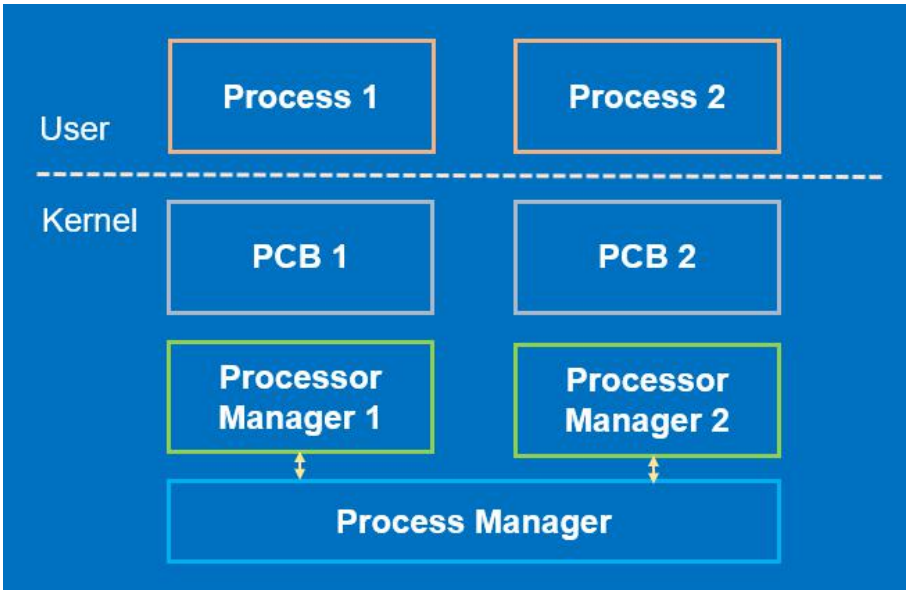
### 3.2.1 进程管理

进程管理模块主要功能有：进程初始化、进程载入和解析、进程切换、进程状态构建模块。

进程管理模块时UltraOS三大核心模块，并且由于涉及到进程的运行，进程管理模块与内核有着非常高的耦合度，同时也是最为错综复杂的部分。我们将进程管理进行层次化区分，这样我们就可以以一个通用的逻辑来对进程进行管理，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。但是值得注意的是，进程管理层次的划分和实现有着非常大的区别，因为层次锁定了各个部分的具体对应关系，但是不关注于具体的实现，甚至不关注具体功能。

我们将进程的管理分为四个层次：进程、进程控制块、核调度器、进程管理器。其层次级别从低到高依次排列。

其示意图如下：




进程的层次结构

3.2.1.1 进程控制块

进程本身在用户态运行，为对其信息进行抽象，我们使用Process Control Block（PCB）也就是进程控制块来管理信息，进程控制块在内核进行管理。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资源信息、进程运行信息、进程间的关系等等。实际上，因为进程是操作系统管理的核心对象，其拥有的资源包括内存以及文件，因此进程控制块内部信息包含了内存管理和文件系统相关的实现和管理。

3.2.1.2 核管理器

为了管理进程的信息，并对其进行相应的操作，我们引入了核调度器，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以表示对其的拥有，与CPU核对应。这样，我们可以实现一一对应关系，方便与进行逻辑上的管理。因为核上进程将会随时间变化进行切换，因此核心管理器需要实现进程间的切换。

3.2.1.3 进程管理器

而进程间的切换则由进程管理器进行管理，该管理器不同于核心管理器，它并非每个核心所独有，而是所有核心所共有的。这么做的原因是，UltraOS将进程管

理放在全局的角度来看，每一个进程都有可能被任意一个核心所调度运行。进程管理器主要的职责是全局进程的调度，包括等待进程的调度实现，可以支持不同的调度算法，而核管理器无需关心调度算法，它只要保证自己能够进行切换进程即可。

#### 3.2.1.4 进程机制

之前我们仅仅关注进程是怎样被组织的，但是是实现上有着相当的差别，同时具体的功能也在内部的掩盖之下。注意到我们使用的编程语言Rust为面向对象语言，在我们的层次设计之下，Rust有着绝佳的实现可能。

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UltraOS还构建了进程id号的分配器，该分配器用于分配Pid，以及相应的内核栈空间。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。

实际上，进程的组织虽然和文件系统和内存管理平行，但是实际上还是处于顶层模块，其与内存和文件系统杂糅在一起，起着统领的作用。而PCB的作用就是集大成的最终接口，其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。

### 3.2.2 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。

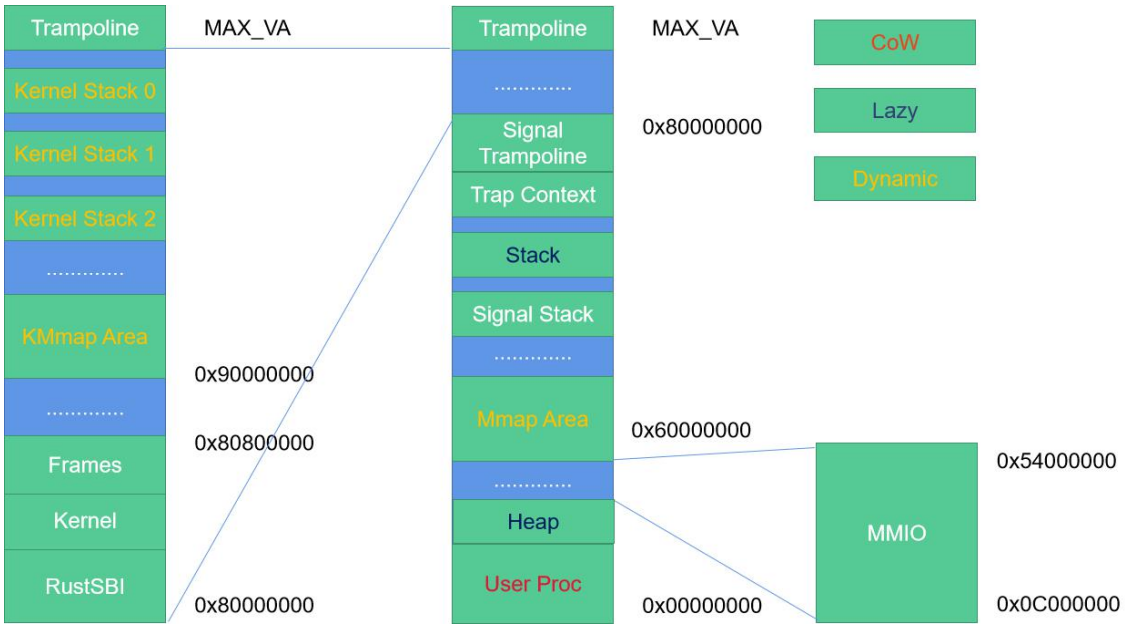
K210平台的最大问题就是内存不足，除去SBI所管理的空间，操作系统内核加上其所管理的空间只有6MB。因此我们在内存管理部分进行了许多优化，优化的原理

大概分为CoW和Lazy两类，具体的做法在系统实现有具体的说明。

同时，UltraOS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。

UltraOS的内存布局如下图所示：




UltraOS将内核和用户的页表融合在一起，这样，我们在陷入内核的时候就不必进行页表的切换。同时，融合的前提条件是内核和用户的虚拟地址空间完全不重叠。

### 3.2.3 文件系统及设备

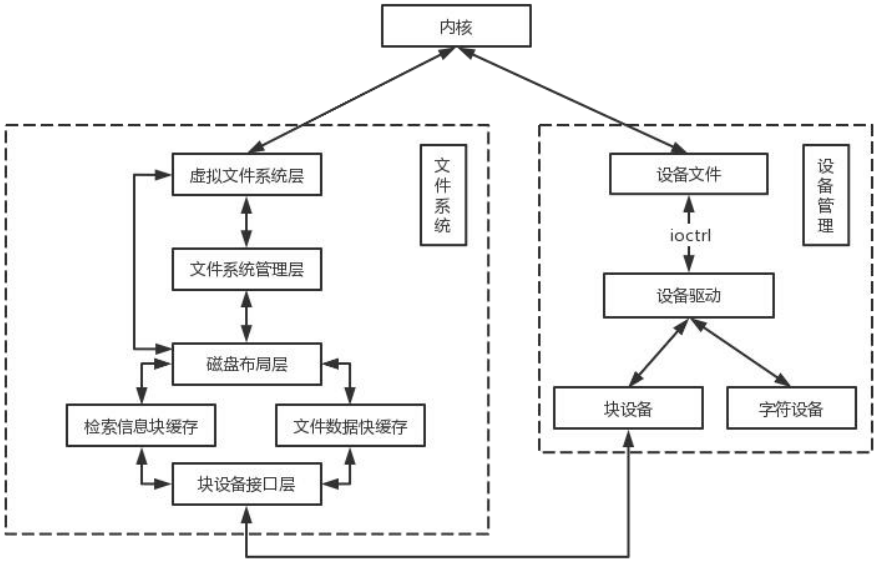
UltraOS继承了UNIX“一切皆文件”的设计哲学。我们将文件分为实际文件和抽象文件两大类。前者即普通的文件，后者可以是任何具备读写功能的系统对象，例如设备、管道等。为此，我们的文件系统分为两个部分：磁盘文件系统以及内核虚拟文件系统。

Ultra OS支持两种磁盘文件系统，分别是参照EXT2的简易文件系统和FAT32文件系统。因为最终提交本版为支持FAT32的文件系统，且二者整体架构相似，所以本文仅介绍FAT32文件系统。在我们的设计中，内核和文件系统是低耦合的，这样有更好的可拓展性。此外，因为UltraOS支持双核，所以文件系统也对并发访问做了相应的设计。

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接口，都可以与内核对接使用。我们也对其他任何具备读写功能的对象进行了抽象，只要实现了文件接口，内核就可以按文件的形式将其管理。

#### 3.2.3.1 结构设计

UltraOS的文件系统的整体结构与数据通路如下：




磁盘文件系统主要由块设备接口层、块缓存层、磁盘布局层、文件系统管理器层、虚拟文件系统层构成。下层模块为上层模块提供服务，文件系统管理层为上下

的多个模块提供服务。设备管理部分则由设备文件、设备驱动、设备实体几个部分组成。

3.2.3.2 功能模块介绍

该部分将对上文提及的五个层级、内核支持，以及并发访问进行简单介绍。

(1) 内核虚拟文件系统

内核虚拟文件系统统筹了所有类型的文件，把不同可读写对象抽象出了统一的接口，主要面向系统调用。通过这些接口，相关系统调用可以按一致的编程模式实现，既能提高代码复用率，又具备很强的可扩展性。

(2) 设备管理

设备的多样性往往是系统走向的实用关键。因为比赛的性质，UltraOS并没有实现丰富的设备驱动，但我们为未来可能的拓展搭好了框架，将设备驱动程序和内核解耦。设备管理是内核与设备驱动之间的桥梁。对于内核，其需要提供接口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

为了在虚拟机和开发板上运行，文件系统必须支持不同的块设备，例如SD卡或磁盘镜像。块设备接口层即用于与不同的块设备对接，同时为文件系统屏蔽不同块设别的差异性。

(4) 块缓存层

I/O设备的读写是影响文件系统性能的关键。为了提升性能，需要利用局部性原理设计缓存以减小I/O设备读写次数。此外，为了避免不同类型的块数据覆盖而造成效率下降，我们设计了双路缓存，分别存储文件数据和检索信息。

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、替换过程交由缓存完成。

(5) 磁盘布局层

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，例如引导扇区、文件系统信息扇区、FAT、目录项等。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。磁盘布局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) 文件系统管理层

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。该层为其他模块提供了FAT32相关的实用接口，其他模块如有任何相关的计算或者处理工作，例如获取/回收簇、计算地址、单位转换、文件名处理等，都可以调用该模块的接口。

(7) 磁盘虚拟文件系统层

虚拟文件系统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。在该层中，我们定义了虚拟文件结构体以对文件进行描述，其与短目录项成对应关系，共同作为访问文件的入口。该层实现了文件系统常见的功能，例如创建、读写、查找、删除等。

# 4 系统实现

UltraOS采用Rust语言进行具体的实现，使用rCoreTutorial-v3（wyf，THU）作为其主体框架，RustSBI（Luojia，HUST）作为底层硬件抽象实现。在此基础上，进行修改以符合以上所述的所有设计。

## 4.1 进程管理

在模块设计的时候，我们将内核对于进程的实现分成了三个部分，接下来我们对其具体实现进行具体讲述。根据由表及里的叙述原则，我们从高层次到低层次依次进行叙述，即：进程控制器、核管理器、进程控制块。

### 4.1.1 进程控制器

目前。进程控制器采用的调度算法为FIFO算法，因此对应的实现为进程就绪等待队列。在队列中排列着所有等待进程的进程控制块引用，每次核调度器需要进行进程切换时都会从该队列中取出队头的进程控制块，作为核下一个运行的进程。

当进程就绪等待的时候，也需要将其放入队列之中，作为候选调度对象。而FIFO

算法就会根据进入队列的先后顺序进行出队。

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

在这里，我们可以自由实现各种调度算法，但是其必要接口只有三个：初始化、新增就绪进程、取出就绪进程。进程控制器可以采用其他的算法来进行进程调度优先级。

注意，由于UltraOS采用了多核的设计，对于共享的进程控制器的访问，我们设计了互斥锁保证同时只有一个核能够访问该结构。

### 4.1.2 核管理器

对于核管理器，其内部存放的是当前运行的正在运行进程，以及独有的内核栈指针，该内核栈指针专用于用于进行进程切换。

Idle栈指针，即刚才所提及的独有内核栈指针，在有效初始化时被设置为内核的初始栈，每个核心独有一个内核初始栈，在进行进程切换的时候，内核持有的是进程独有的内核栈，而进行切换的时候，需要将内核栈切换至Idle栈，并对相应的临时寄存器进行保存。Idle栈实际上就是初始栈，但是初始栈在完成内核的初始化之后就变为了Idle栈，做这样的区分是因为：Idle栈仅为进程调度服务。实际上，Idle栈对应的上下文指示的程序所在地，是一段无尽的进程调度循环，永远在寻找下一个切换的进程，如果成功获取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。

除了Idle栈的使用，其他的使用则都对应于进程控制块的具体使用。首先，核管理器提供基本的使用方式：获得当前正在运行的进程信息、获得当前正在运行的核心信息，获得该核心的Idle栈指针。

当然，最主要的还是核管理器的进程调度。它将不停的寻找下一个进程，如果成功获取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。

### 4.1.3 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。
- 辅助信息：时钟，资源使用量和限制，定时器，信号等。

```
pub struct TaskControlBlockInner {
    // task
    pub trap_cx_ppn: PhysPageNum,
    pub task_cx_ptr: usize,
    pub task_status: TaskStatus,
    pub parent: Option<Weak<TaskControlBlock>>,
    pub children: Vec<Arc<TaskControlBlock>>,
    pub exit_code: i32,
    // memory
    pub memory_set: MemorySet,
    pub base_size: usize,
    pub heap_start: usize,
    pub heap_pt: usize,
    pub mmap_area: MmapArea,
    // file
    pub fd_table: FdTable,
    pub current_path: String,
    // info
    pub address: ProcAddress,
    pub rusage: RUsage,
    pub itimer: ITimerVal, // it_value if not remaining time but the time t
o alarm
    pub siginfo: SigInfo,
    pub trapcx_backup: TrapContext,
    // resource
    pub resource_list: [RLimit;17],
}
```

基于所拥有的信息，进程控制块提供了最基本的方法，包括：获取当前进程的栈顶指针、页表、状态等方法。接下来介绍进程控制块提供的复杂方法。

(1) 新建进程（控制块）：进程控制块必须唯一对应一个进程，因此，其初始化

不能够仅仅填充对应结构体的数据，而是应该从进程的数据出发进行构建。

- 进程控制块将根据程序的内容（程序设定为为ELF文件），构建出用户地址空间并同时分配物理页帧给对应的空间。
- 分配对应的Pid，以及绑定于Pid的内核栈。将内核栈初始化一个空的上下文，以便适应现有的进程调度运行结构设计。
- 创建对应的进程控制块，并且预先分配三个文件描述符（标准输入输出和错误）。

同时我们需要注意到，进程的直接创建有仅有可能是第一个进程initproc的创建，其他的进程都通过fork或者是exec进行创建。

(2) 复制（fork）进程（控制块）：从大体上来说，就是复制一个几乎一模一样的进程（以及进程控制块），因此我们需要构造一个完全相同的用户空间以及分配对应的空间，还要分配Pid和对应的内核栈。但是注意，fork还需要建立父子关系。

(3) 动态堆增长：根据堆的增长（减少），来分配（回收）对应的页帧，并且记录相应的堆信息。

### 4.1.4 辅助信息

辅助信息包括有三大类：

- 定时器：该定时器负责周期性或者一次性根据时间发送时间软中断信号。
- 资源记录仪：记录进程运行过程中的资源使用量，包括时间消耗，陷入内核次数等等。
- 信号处理机：用来存储信号以及处理信号。

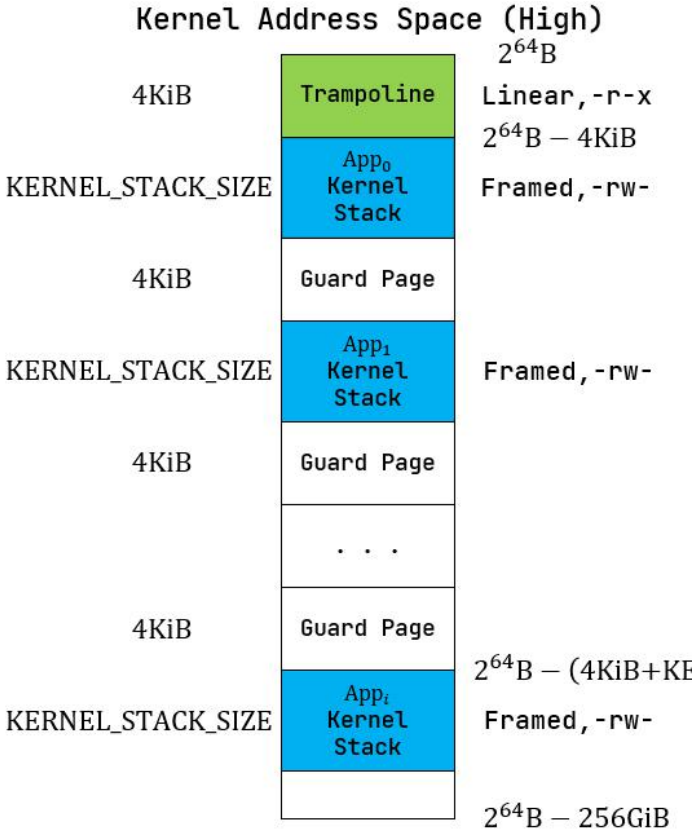
## 4.2 内存管理

内存管理主要设计的是用户以及内核的内存地址空间。

同时，内存管理中并非简单的地址设计问题，UltraOS还需要满足进程和文件系统的独有要求，前者我们以进程切换的trap context的设计为例，后者以mmap的设计为例。

### 4.2.1 内核地址空间

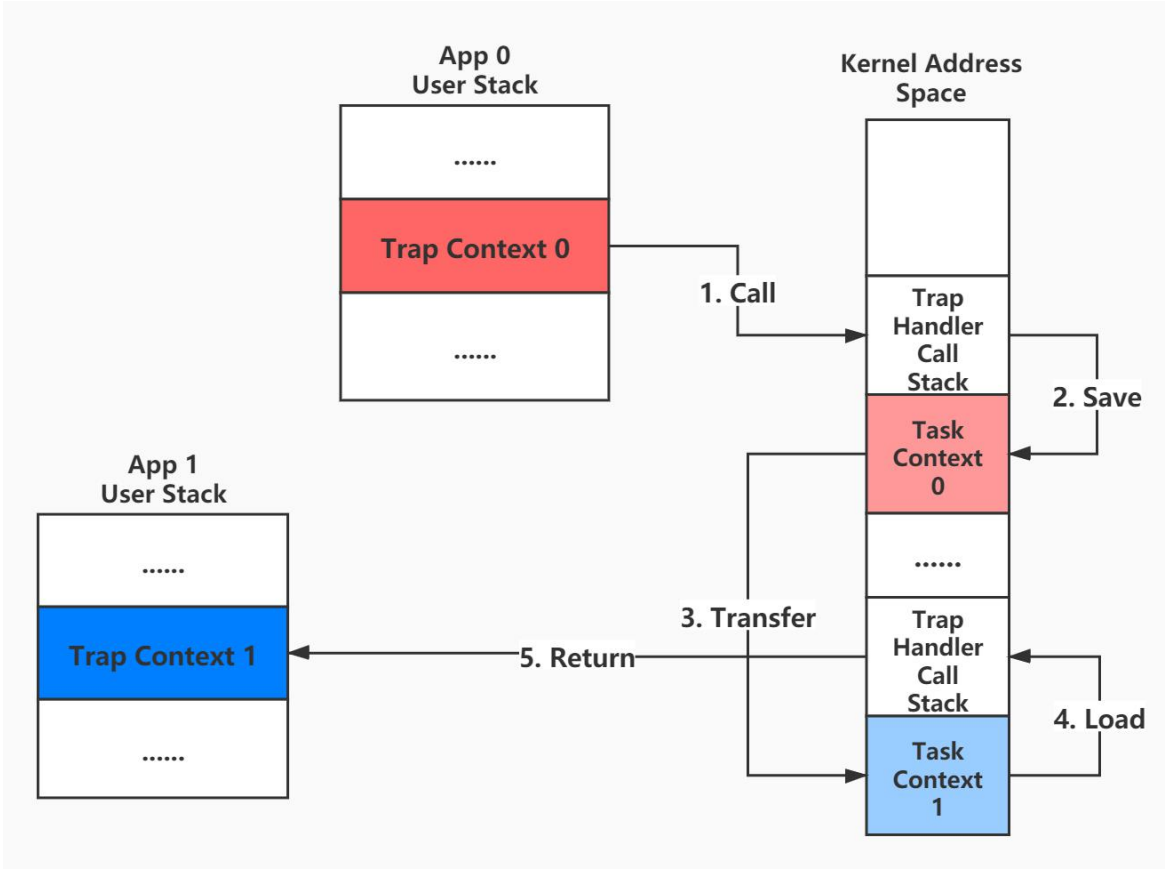
从操作系统启动开始，内核地址空间就需要载入启动所需要的数据，并且载入0号初始进程initproc以及用户命令行进程usershell。在运行程序的过程中，内核地址空间划分成多个应用程序各自对用的内核栈空间，并且通过guardpage机制来分隔开。




### 4.2.2 Trap Context相应内存设计

操作系统在不同的进程之间进行切换，并行执行不同的用户程序时，需要进入内核态来完成进程的调度以及切换任务。

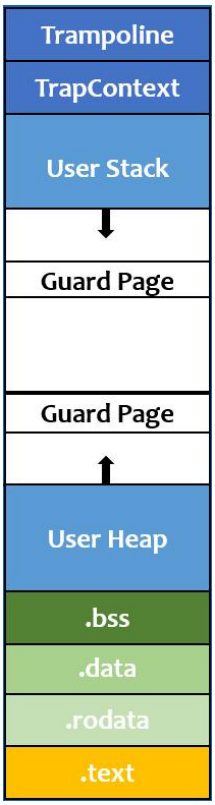
如图，




需要切换进程时，首先需要通过trap的方式进入内核态，因此需要通过陷入时的TrapContext结构来保存当前用户态的寄存器上下文（保存在该用户程序对应的内核地址空间的栈中），并且进入到TrapHandler中，相关的Trap信息也保存在了TrapHandlerCallStack之中。在内核态完成系统调度并且准备切换到下一个用户态程序时，通过TaskContext保存之前进程的信息。这时，可以将栈指针sp切换到另一个内核栈并且载入另一个即将运行的进程的TaskContext，并且通过下一个进程的TrapHandlerCallStack和TrapContext恢复到该进程的现场，并且回到用户态，开始进行下一个用户程序的执行。

### 4.2.3 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。

在进行 fork 或者 clone 的时候，MemorySet 也提供了专门的接口来实现通过这两种系统调用而生成新的进程的

```
pub struct MemorySet {
    page_table: PageTable,
    areas: Vec<MapArea>,
}
```

MemorySet 的方法。

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。

用户地址的栈空间用于函数之间相互调用时，传递参数，在用户栈底下的是该应用程序的预分配的静态内存空间。

而用户地址的堆空间是从（除了跳板和TrapContext以外的）顶部的地址开始向下增长的，是动态的内存空间，通过sys_brk()的系统调用来进行空间的增长，并且把空间给BuddyAllocator管理，用户程序可以通过new()函数来获取。

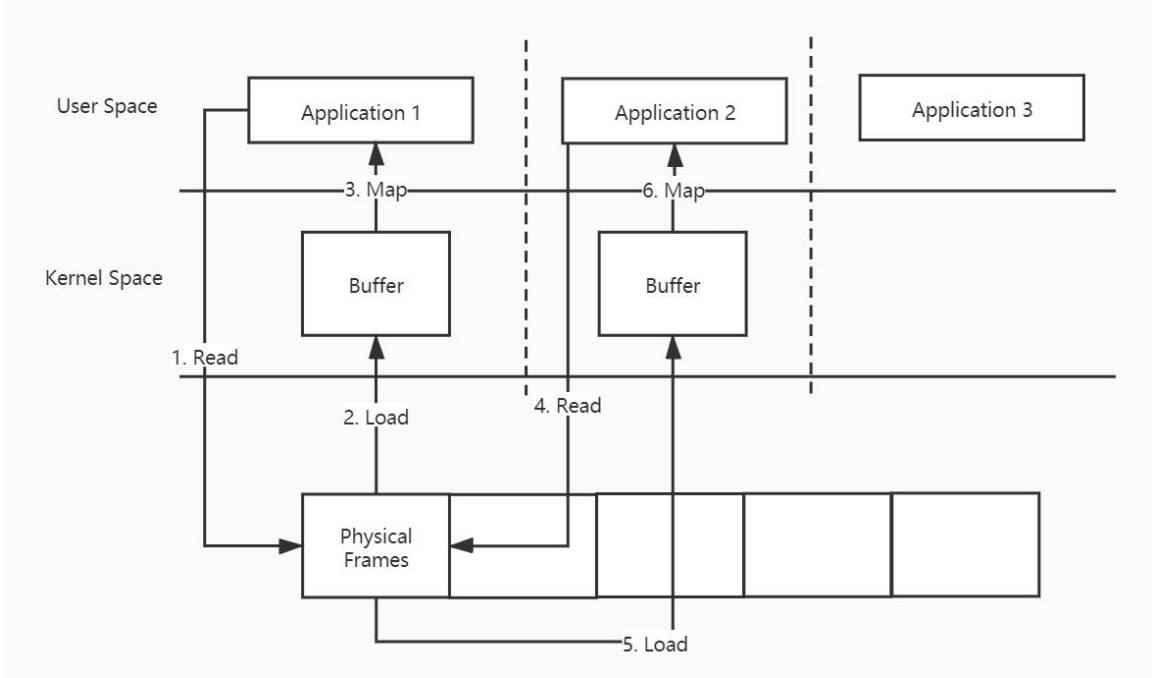
sys_brk 也是几句通过 sbrk 的系统调用实现的。sys_sbrk 是用户堆空间分配动态内存的接口，可以对某个进程的动态内存进行增长和收缩，通过 grow_proc()函数实现。

堆空间也是从属于所在进程的 MemorySet 之中的，在最开始空间大小为 0，并且

堆指针随着空间的增长与收缩而上下移动。通过 grow_proc 函数，用户内存在堆空间中增长响应的大小，并且把这一整段内存地址传给 buddy_allocator，于是通过 BuddyAllocator 的 new()可以在应用程序中获取随即大小的动态内存空间。

### 4.2.4 mmap和munmap设计

通常情况下，如果用户需要访问文件或者I/O接口，需要调用内核的sys_open, sys_read, sys_write这些提供的文件系统的syscall来进行操作。如图1，如果多个进程的各自应用程序需要访问相同的某个磁盘块时，需要分别在自己的用户内存空间中建立响应的文件缓冲区buffer，从而实现文件的读/写操作，并且收到文件锁的限制，不可以同时对相同文件进行写入。

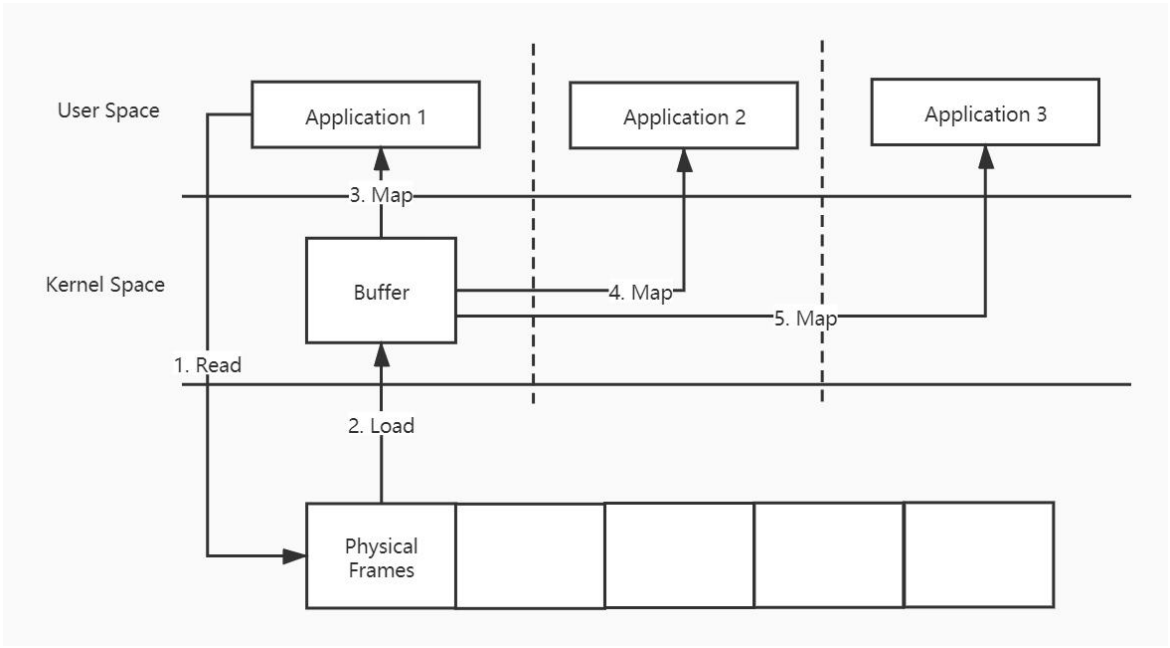



正常文件缓冲区

但是通过sys_mmap和sys_munmap的系统调用，可以在文件磁盘块和内存空间之间建立映射关系，此时就可以将文件块当成内存空间来进行读写的操作。需要某个磁盘块在第一次被进程调用时，在内存地址中开辟对应的空间，并且通过mmap进行映射，将其映射到一段虚拟内存地址中，并且建立页表。在最开始访问这段内存时，

触发页表的invalid异常，此时异常处理中断会通过文件操作读取磁盘块中的数据并且拷贝到内存中；此后每次就可以直接对这段内存数据进行操作了。并且有新的其它进程要访问此文件时，再次调用sys_mmap可以直接共享这段内存空间，并根据它的权限标记来判断是否可以共享读/写这段内存。

当一个进程访问完这个文件块需要释放时，调用sys_munmap, 如果所有进程绑定的都对这段空间进行了munmap, 就会通过系统调用将这段映射解除。




mmap文件缓冲区

mmap 的空间也是在进程建立时，就在 MemorySet 中建立了一个相应的数据结构进行分配的，但是在 MemeorySet 中并不能很灵活地管理 mmap 的虚拟地址空间的分配，因此加入了 MmapArea 和 MmapSpace 的数据结构来专门管理磁盘文件块和对应的物理缓冲区与虚拟内存之间的映射关系以及页表的建立，这两个数据结构本身并不分配内存空间，但是调用页表的接口来建立虚实地址的映射，并且本身记录了 mmap 相关的全部信息，以便动态地建立和调整 mmap 空间。

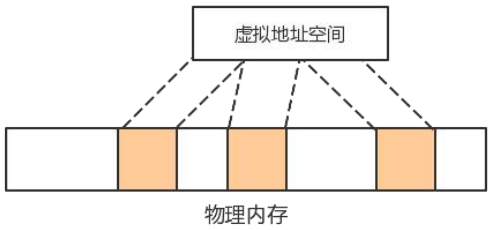
每当建立新的 mmap 时，先向 Mmap 传递将要映射的文件大小，并且获得可以建立页表映射的那段虚地址，然后在 TaskControlBlock 中完成地址的映射。随后进入为这段 mmap 而新建的 MmapSpace，通过 fat32 的文件接口将这段文件内容读出并且写入到对应的物理缓冲区地址中，完成 mmap 系统调用。

### 4.2.5 kmmap设计

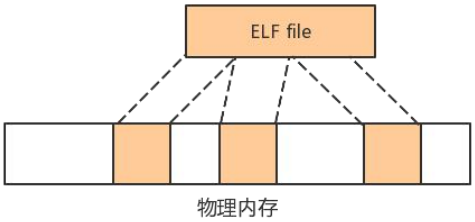
UltraOS起初使用内核堆读入可执行文件，在解析后存入用户空间。内核堆采用了具备分裂性质的伙伴管理系统，这意味着需要极大的额外空间以支持大型可执行文件，例如Busybox需要大于2M的额外空间。但在实际的运行中，我们不需要如此大的内核堆，因此需要及时释放这些分配给可执行文件的页框。

为此我们实现了kmmap。kmmap即为内核服务的mmap，其将文件映射至内存，同时通过非恒等映射为内核提供连续的虚拟地址。通过kmmap，我们可以更加灵活地为内核加载的文件分配和回收内存。例如载入用户程序时，其具备以下流程：

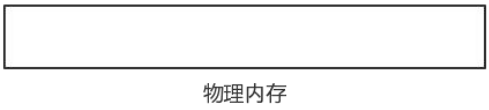
- 根据文件大小，内存管理器分配足够页框，同时在内核页表建立与连续虚拟地址的映射。




- 将可执行文件载入映射区域，进行解析




- 完成解析后，释放页框



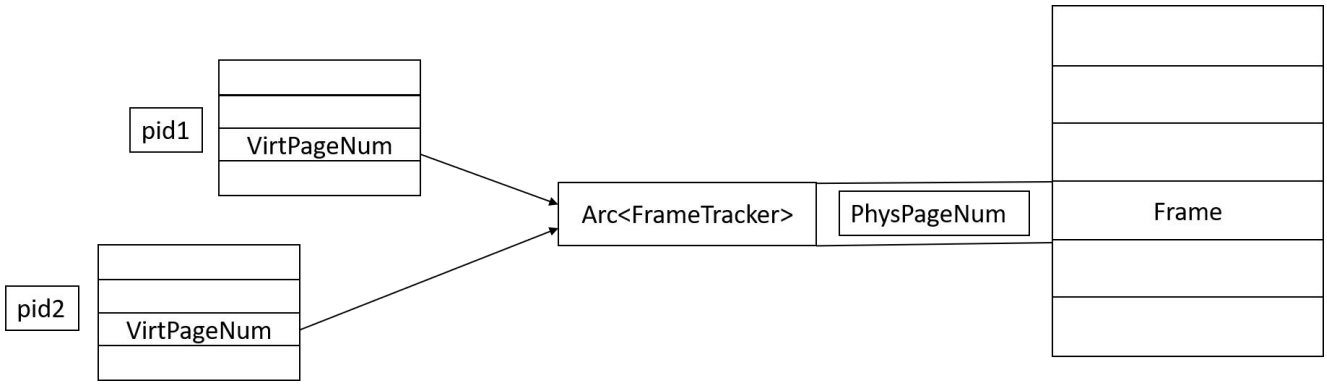

经过我们的测试，当使用堆载入文件时，运行Busybox总共需要的内存大于6.5MB。而当采用kmmap策略时，我们可以将堆减小至512KB，最后只需峰值占用4.38MB即可运行内核和完整的Busybox，减小峰值内存占用32.6%。

### 4.2.6 Copy on Write 优化fork之后占用的内存

CoW策略的具体方法是，当某个进程调用fork()生成新的进程时，需要为新的进程创建一个新的地址空间MemorySet，这个地址空间的内容应该是父进程的地址空间的完全复制，因此内容相同。在原来的情况下，这个过程会执行彻底的复制，但是由于子进程之后可能会马上就调用exec()来载入新用户程序的elf文件，原来的内存彻底清除，就浪费了原来的复制时间。CoW的做法是在创建新子进程的过程中，把子进程的虚拟页对应到父进程原有的物理页上，也就是父子公用同一个物理页。

这样的情况下，父子都只允许读这些空间而不能进行写操作，如果要进行写操作的话，写的那一方会为要写的那个虚拟地址页，自己申请分配一个新的物理页帧，在自己的页表中重新映射这个虚拟地址页，这时就可以独享单个物理页帧，也可以进行写操作。

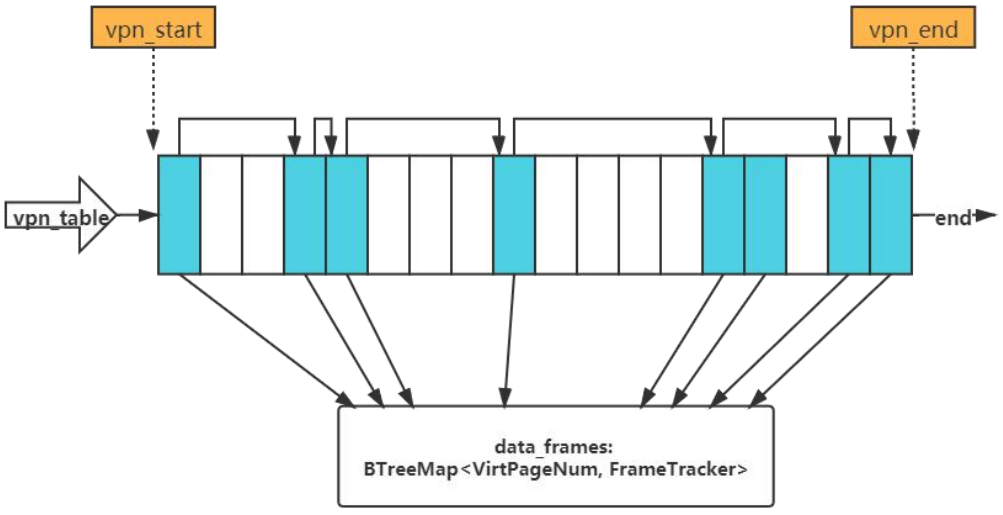
对一个物理页帧的引用次数RefCount决定了它是否是一个CoW策略的物理页。当一个新的进程的虚拟页映射了这个物理页帧的时候，它的RefCount+1，当RefCount>1时，就意味着不止一个进程引用了该物理页帧，也就是使用了CoW；当一个进程取消映射时，RefCount-1。



### 4.2.7 Lazy Stack & Heap: 动态分配的堆栈空间

在一开始的UltraOS中，我们采用预分配与映射的方式来为每一个用户程序预留一段固定大小的堆/栈空间，但实际上这段空间在大多数程序的运行过程中并未别完全使用，或是只是用了非常少的一部分。因此，随着堆的增长（brk）和栈的增长（push）来动态地增长，当需要使用到某个虚拟页的时候再给它分配物理页帧，可以节省足够的空间，以免在程序进行高强度的fork()的时候，可分配的物理内存空间不足。但对于栈空间Stack来说，每个程序至少会用到一定大小的内存，这造成了每个程序栈空间的前几个一定会被分配。在Lazy的策略下，每次都要进入trap去给他们分配，实际上拖慢了系统的执行速度，我们参考Linux的方法，恰好给每个进程提前分配少量固定的堆栈空间，对于堆栈使用强度小的大部分程序，这些提前分配的空间恰好足够他们使用，不触发lazy alloc，对于少部分强度大的程序，后面分配的空间不一定全部都被使用，因此当用到时再触发trap来分配。

不同于elf文件对应的程序段和数据段在内存空间中的连续存放的方式，lazy策略的堆栈段中已经被映射到页表的虚拟页可能是离散不连续的，地址也是分开的，因此不同于maparea数据结构中通过VPNrange来保存内存空间段的方式，我们采用类似链表的数据结构来把lazy分配的零散的堆栈内存虚拟页串在一起。每一个段对应一条虚拟页的链表，当然还需要其它的信息，比如段的起始和结束地址，因此我们把这些内容都封装在chunkarea的数据结构里，并且和maparea一样整合到某个进程下的MemorySet中去。



```rust
pub struct ChunkArea {
    vpn_table: Vec<VirtPageNum>,
    data_frames: BTreeMap<VirtPageNum, FrameTracker>,
    map_type: MapType,
    map_perm: MapPermission,
    mmap_start: VirtAddr,
    mmap_end: VirtAddr,
}
```

在进行了lazy的优化后，我们对它的主要目标：内存空间（物理页帧）的节省效果进行了测试，测试用的程序就是决赛第一、二阶段的Busybox和lmbench。

| | **without lazy alloc** | | | **with lazy alloc** | | | **Compare** |
|---|---|---|---|---|---|---|---|
| **test on program** | before runing program | after | **pages consume** | before runing programafter | after | **pages consume** | **improve** |
| lmbench_all lat_syscall -P 1 null<br>lmbench_all lat_select -n 100 -P 1 file<br>lmbench_all lat_sig -P 1 install<br>lmbench_all lat_sig -P 1 catch | 1107 | 662 | 445 | 1232 | 912 | 320 | 28.02% |
| lmbench_all lat_proc -P 1 fork<br>lmbench_all lat_proc -P 1 exec<br>lmbench_all lat_proc -P 1 shell | 1107 | 566 | 541 | 1232 | 880 | 352 | 34.94% |
| busybox_all | 1103 | 368 | 735 | 1279 | 624 | 655 | 10.88% |

从测试结果中，我们可以看到在fork/exec等lmbench测试程序中，lazy策略将单个程序运行时的内存空间占用优化了大约35%。

### 4.2.8 Lazy Mmap: mmap空间的节约策略

在busybox的测试中，我们频繁地遇到内存不足，物理页帧分配耗尽的情况，部分原因是测试中调用mmap对一个非常大（超出硬件总内存大小）的文件进行内存映射操作，但是实际上只使用了其中的一小部分进行读写。因此在mmap的过程中，很

大一部分文件数据是不需要预先就映射到内存中的，同样采用和堆栈一样的lazy策略，仅通过trap对当前要进行读写的那个页进行页表映射和内存分配。

我们尝试了更加灵活的分配策略，当mmap映射的文件大小只有1个page的时候，大概率随后是要读取/写入这个page的，因此我们在mmap的时候对映射的大小进行判断，如果 len<=PAGE_SIZE，那么进行直接分配内存空间的映射，否则采用lazy策略。

## 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。

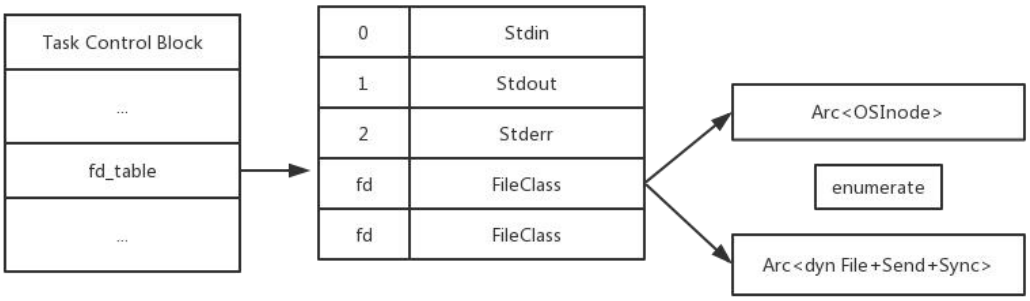
### 4.3.1 内核抽象文件系统

在内核中，我们定义了OSInode结构体来表示文件，该结构体包含读写标签、当前偏移、以及对应虚拟文件的引用。不管是文件还是目录，在内核中都以OSInode的形式存在。除了实际的文件，还有一些可读写的对象，例如部分设备、Pipe，他们都实现了文件的接口File。File包含四个方法：判断可读/可写、读、写。

Stdio通过RustSBI的接口控制UART实现，不具备实体文件，因此这样的抽象文件是无法用OSInode描述的。问题在于，每个进程都要使用文件描述符表(fd_table)维护打开的文件。在rCore-Tutorial的实现中，其使用了Rust的动态分发机制，在fd_table存储实现了File接口的任意结构体的引用。但这样的问题在于，从fd_table取出的文件引用只能调用File包含的方法，OSInode特有的方法，例如查找和创建，都无法使用。

为了解决该问题，我们定义了一个枚举类FileClass，目前其中包含两种类型，分别是抽象文件和实体文件。抽象文件使用动态分配机制对实现了File接口的任意结构体引用，实体文件则是对OSInode的引用。在fd_table中存储FileClass，既能保证维护不同类型的文件，也能保持不同类型文件的方法。

在此设计下，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。

除了维护打开文件外，内核还支持挂载。具体实现为维护一个挂载表，访问目录时首先检索挂载表，如果目录挂载了其他目录，则访问挂载的目录。因为比赛暂不需要挂载不同设备，我们暂时关闭了该功能以提升性能。

### 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UltraOS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。

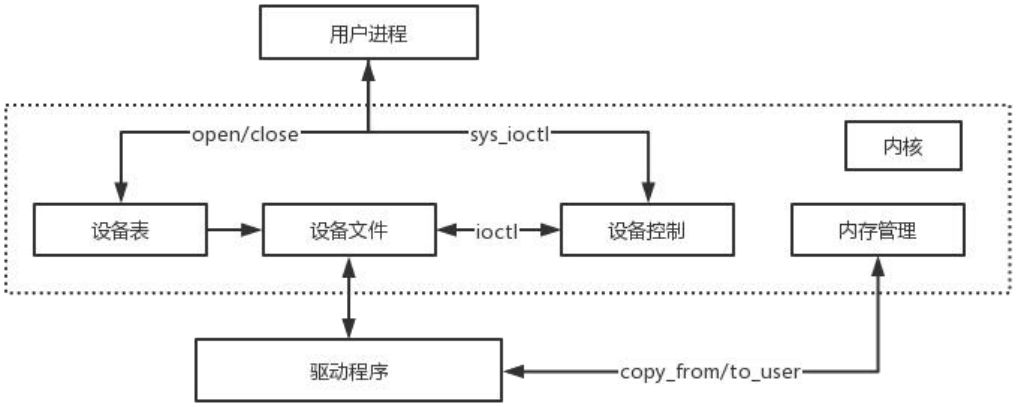
设备管理的目的是为内核与驱动程序提供桥梁。为了可拓展性，UltraOS希望将一切内核无关的程序解耦，这其中就包括了驱动程序。因此我们要定义统一的接口，便于内核和驱动交互。

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。

我们在内核的内存管理模块实现了copy_from_user/copy_to_user两个接口。通过这两个接口，驱动程序可以与用户之间传递任意大小的结构体参数。我们使用了UserBuffer来辅助接口的实现，以应对结构体跨页的情况。

UltraOS的设备管理框架图如下：




### 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。

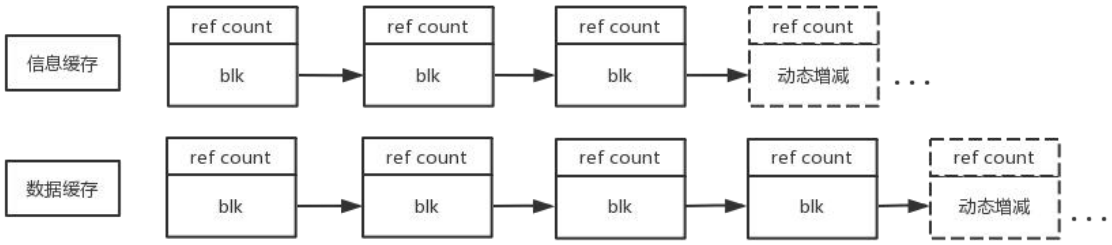
### 4.3.4 块缓存层

文件系统以块(扇区)为基本读写单位，因此缓存的基本单位也是块。参照CPU缓存指令数据分离的设计，我们将块缓存分为信息缓存和数据缓存。信息缓存用于缓存存储检索信息的块，例如文件系统信息扇区、FAT、目录等，数据缓存则用于缓存文件的数据。这样做的好处是显然的：因为开发板内存很小，我们不能设置大的块缓存，如果不采用双缓存，一旦操作比较大的文件，该文件的块会充满整个缓存，覆盖检索信息块，造成目录操作、检索操作、信息修改等操作效率低下，尤其是修改文件时，会频繁访问FAT和磁盘信息扇区。使用两个缓存则可以避免覆盖的问题。

对于缓存本身，我们设置其单个大小为10个块，即5KB，双缓存合计10KB。因为UltraOS暂未实现完善的时钟，所以难以实现严谨的LRU策略，为此我们使用了近似LRU算法——Clock算法。我们使用了一个队列维护所有缓存块，该队列使用了

alloc库提供的VecDequeue，其在堆上动态增减，进而可以灵活地减少内存占用。文件系统上层通过智能指针Arc访问缓存块，其会为引用原子计数。当队列满时，寻找引用量为0的块进行替换即可。

缓存结构如下图所示：




通过缓存，我们屏蔽了扇区读写的具体细节。我们定义了get_block/info_cache以及modify_block/info_cache四个接口，通过这些接口，上层模块不必考虑分区的起始偏移等问题，只需向缓存寻求需要的逻辑块号，缓存就会帮其获取。为了让上层模块可以直接按照需要的数据类型对缓存块操作，这四个接口支持传入闭包，通过闭包可以指定数据类型和处理的过程，进而对块中指定类型的数据进行操作。

UltraOS的文件系统是支持分区磁盘的，这实际需要缓存的支持。在管理层启动文件系统后，其会从0号扇区读入隐藏扇区数（第一个分区的偏移量），并传递给缓存层，之后缓存层每次进行块设备读写时都会加上此偏移量，进而上层模块只需使用逻辑块号。

#### 4.3.5 磁盘布局层

磁盘布局层用于组织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。短名目录项的核心接口如下：

| 接口 | 描述 |
| --- | --- |
| initialize | 以指定的信息初始化目录项 |
| get_pos | 获取文件的扇区和偏移 |
| read_at | 在指定偏移处读文件 |
| write_at | 在指定偏移处写文件，上层模块需保证文件大小足够 |
| as_bytes/as_bytes_mut | 将数据结构转换为字节数组 |
| checksum | 计算短文件名的校验和，用于填充长名目录项的字段 |

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。我们对FAT实现了丰富的接口，核心接口定义如下：

| 接口 | 描述 |
| --- | --- |
| calculate_pos | 计算簇号对应表项所在的扇区和偏移 |
| next_free_cluster | 获取可用簇的簇号 |
| get_next_cluster | 获取当前簇的下一簇 |
| set_next_cluster | 设置当前簇的下一簇 |
| get_cluster_at | 获取某个簇链的第i个簇(i为参数) |
| final_cluster | 获取某个簇链的最后一个簇 |
| get_all_cluster_of | 获得某个簇链从指定簇开始的所有簇 |
| count_cluster_num | 统计某个簇链从指定簇开始到结尾的簇数 |

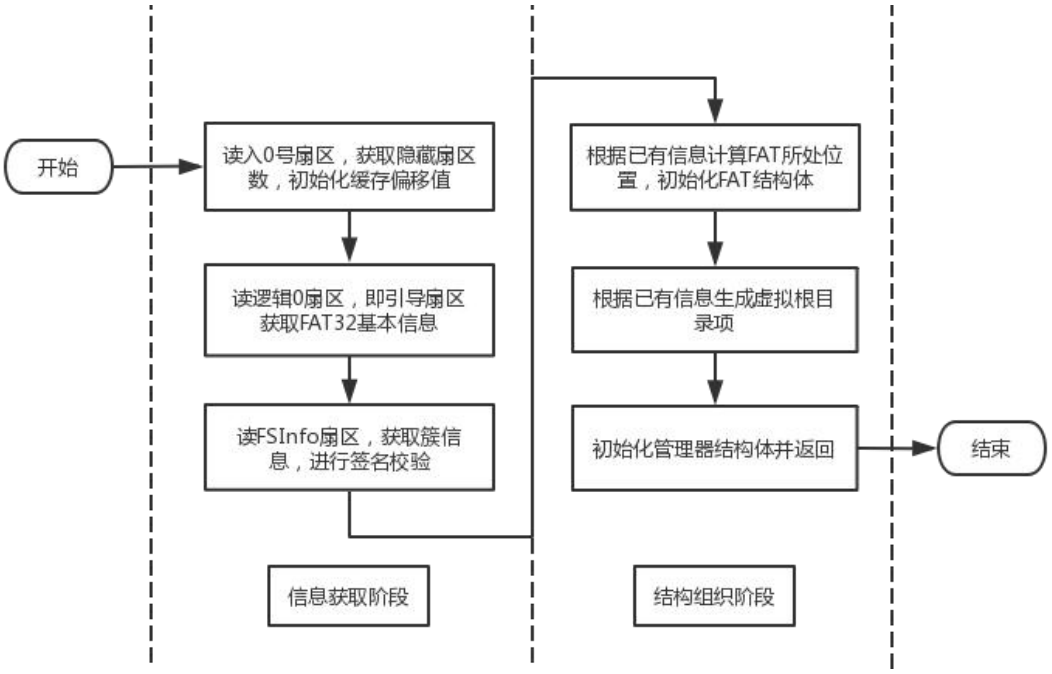
以上接口实现的算法比简单，主要为反复查询FAT，此处不再赘述。但是通过这样的封装，其他模块可以便捷的对FAT操作。例如，管理模块可以通过接口直观地对簇进行分配和回收，虚拟文件层通过接口可以计算文件或者目录占有的簇数等。

### 4.3.6 文件系统管理层

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启动文件系统，其需要读取引导扇区的数据并进行校验，然后对数据结构进行简单组织。文件系统的启动流程如下：




对于其他模块而言，管理层提供了实用的工具。但凡需要FAT32相关的计算，或者文件系统的相关信息，都可以调用管理层的接口。其提供的服务包含计算文件起始扇区、拆分长文件名、文件名格式化、短文件名生成、计算文件大小所需簇、获取可用簇、单位转换等等。其中，文件名格式化指将自字符串转换为存储于目录项的字节数组格式，同时对空缺位置进行补全。FAT32在长文件名的情况下以长名目录项+短名目录项的形式存储目录项，短名目录项中的文件名由长文件名压缩而成，常用的压缩方式为：取文件名的前6位+编号+后缀。短文件名生成接口就用于实现该功能。其余接口的算法都比较简易，此处不赘述。

### 4.3.7 虚拟文件系统层

在虚拟文件系统层中，我们将文件抽象为虚拟文件，设计了VFile结构体，其成员包含常见用的文件信息、文件短目录项所在扇区和偏移、FS管理器和块设备接口的引用。上文提及本文件系统以短目录项为访问入口，因此此处的虚拟文件与短目录项对应。以下是VFile对内核提供的重要接口：

| 接口 | 描述 |
| --- | --- |
| create | 在当前目录的相对路径下创建文件 |
| find_vfile_bypath | 查找当前目录下的文件，支持递归查找 |
| read_at | 在当前文件的指定偏移处读数据 |

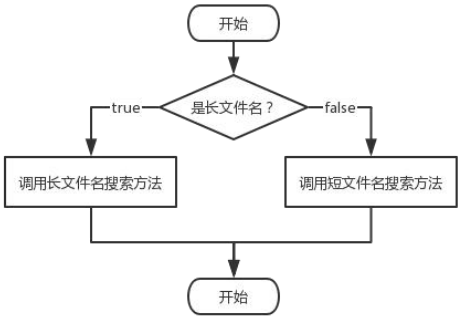
| | |
|---|---|
| write_at | 在当前文件的指定偏移处写数据，会自动完成簇分配 |
| clear | 清除当前文件 |
| ls | 列举当前目录下的所有文件 |
| dirent_info | 获取当前目录下指定目录项的信息 |
| stat | 获取当前文件的信息 |

上述接口主要为供内核使用的接口，一些私有的方法未在此例举。

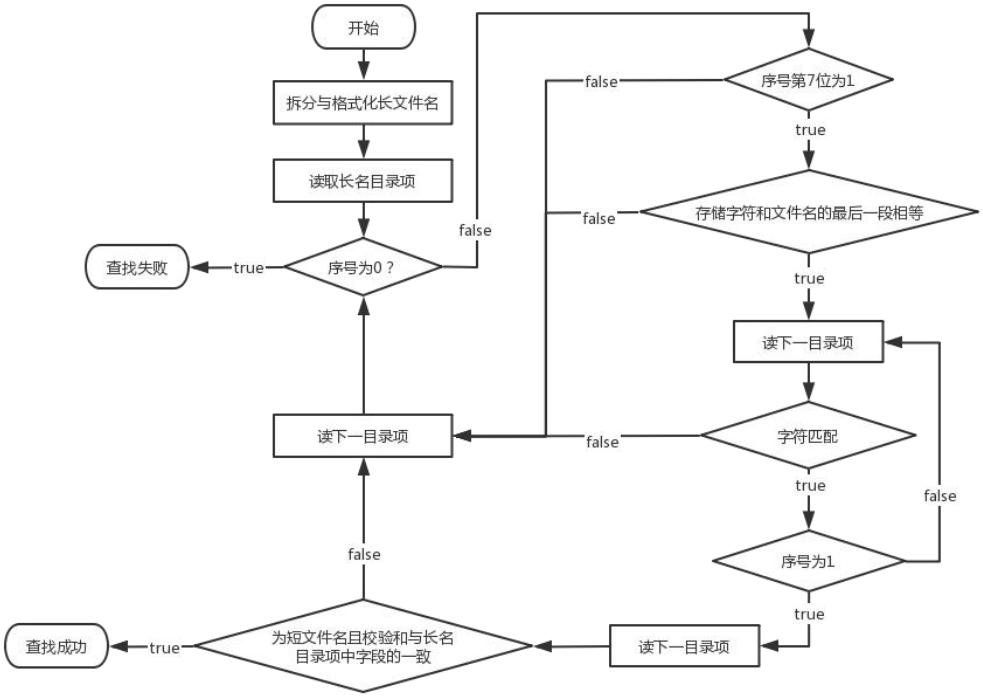
以下介绍一些核心的算法：

FAT类文件系统不同于EXT之处在于，其为链式结构，而非索引结构。实现EXT的难点在于间接索引的逻辑比较复杂，而FAT则不必考虑该问题。FAT文件系统的难点在于其区分了长短文件名，使用不同结构的目录项存储。

首先介绍文件搜索算法，其核心即为目录项检索，整体算法流程如下：




算法先判断文件名长短，然后调用对应的方法，因为短文件名匹配比较简单，这里仅展示长名匹配算法：




长名目录项以“连续长名目录项+对应短名目录项”的形式存储，例如abcdefghijklmn.txt，将以如下形式存储：

| | |
|---|---|
| 低地址 | 2：n.txt |
| ↓ | 1：abcdefghijklm |
| 高地址 | ABCDEF~1 TXT |

因此进行匹配时要从后向前进行匹配。为了实现该过程，我们使用栈来存储拆分后的长文件名：拆分后顺序压入栈，每次匹配一段就弹出一段，以此实现倒序。

然后介绍文件增量算法，主要分为目录和文件两种类型。在VFile中，我们实现了increase_size方法，用于在写之前分配足够的簇。对于文件，其目录项有记录大小的字段，因此在增量时，只需计算新尺寸和旧尺寸所需簇数的差值，然后调用管理层的分配接口即可。对于目录，FAT32不会在目录项记录其大小，因此在计算时，需要使用FAT提供的接口计算当前使用的簇数，再根据新尺寸计算需要的簇数，最后根据差值调用管理层的簇分配接口。

其余算法的核心思想与搜索算法基本一致，例如create和ls，这里不再介绍。

#### 4.3.8 并发访问

UltraOS支持双核，两个核可能同时访问文件系统。对于并发访问的正确性，我们通过读写锁的方式加以保证。使用读写锁的好处在于，如果不同核的两个进程同

时需要读一个资源，则可以同时访问资源，仅在写资源的时候互斥访问。

在本文件系统中，加锁对象有三者：FAT、文件系统管理器、以及缓存块。

对于FAT，其访问多数为查询，获取读锁即可，仅在对文件增删改的时候可能会修改FAT，因此可以达到较好的并发。FAT的修改位置可以看作随机的，因为需要修改的表项是动态变化的。无法保证分配或者回收的簇号，也就无法保证表项在哪个数据块，进而在执行单个事务时无法保证原子性。例如分配簇时，可能需要分配多个簇，而这多个簇对应的表项可能位于不同的数据块，因此仅对某个块加锁并不能保证其他块上的表项不被另一进程篡改。为此，需要对FAT整体加锁。

对于文件系统管理器，其主要负责组织文件系统全局以及为其他层级提供服务，其本身不会被写访问，因此原则上不需要加锁。但为了之后可能需要拓展其功能，在使用时依然加读写锁。缓存块锁是整个文件系统最基本的锁，因为文件系统以块为读写单位。以块为锁的粒度是一种折衷，如果粒度进一步减小，例如按目录项加锁，则因为FAT32不定长目录项的性质，控制机制会格外复杂，甚至影响效率；如果粒度放大，例如锁整个文件系统，则无法进行并发访问，降低性能。相比之下，块级锁实现简易，同时也具备灵活性。块级锁实际上也实现了对整个文件的加锁，因为本文件系统以短名目录项为访问文件的入口，而访问短名目录项需要获得其所在块的锁，因此相当于对整个文件加锁。

# 5 系统测试

## 5.1 测试准备

UltraOS的整体测试支持两种平台，以及两种应用语言的用户程序测试用例。

在模拟器qemu上，UltraOS将使用FAT32文件系统镜像作为qemu搭载的文件系统。而在K210上，则采用SD卡作为文件系统的硬件支持，并且实现对应的驱动控制SD卡的相应读写功能。

在用户程序上，我们运行了Rust和C编写的对应程序，来判定是否对应的程序满足相应的要求。该程序包括各种系统调用的组合，以及不同压力下的测试。

文件系统的测试主要分为两个部分，一部分是文件系统本身的测试，另一部分是相关系统调用的测试。UltraOS的文件系统是与内核低耦合的，因此可以单独对文件系统进行测试。为了确保测试的可靠性，我们直接将SD卡按FAT32格式化，然后在卡上测试。为了提高测试效率，我们将测试划分为三个阶段：首先使用读卡器将SD卡连接虚拟机，在虚拟机上对文件系统进行单独验证；然后继续使用读卡器，将SD卡作为QEMU的虚拟磁盘，在QEMU上运行内核以对内核支持和系统调用进行验证；最后在K210进行验证。三个阶段调试的困难程度依次递增，但前两步就可以发现并解决大多数问题。此外，FAT32是SD卡的常用格式，Windows和Linux都支持该文件系统。为了验证我们文件系统的可靠性，还应保证在对SD操作之后，其他系统依旧可以正常读写该SD卡，并且能识别本文件系统保留的操作。

## 5.2 测试方法

表 5-1 系统调用功能和性能的测试方法

| 序号 | 测试对象 | 测试方法 |
|---|---|---|
| 1、系统基本指标测试 | | |
| （1） | 系统调用能够正常使用 | 在内核启动后进行对应系统调用的功能测试，并且能够达到基本要求。 |
| | | |
| 2、功能测试（用户态测试） | | |
| （1） | 用户程序退出 | 对 for 可 exit 进行测试，看是否 pid 返回值满足对应要求，同时子进程退出状态能够对应父进程获得的状态。 |
| （2） | 多彩字符测试 | 使用多种 CSI 控制序列，观察是否能够在中断打印出特殊字符。 |

| | | |
|---|---|---|
| （3） | 进程复制测试 | 高强度进行进程复制并且退出，观察是否 pid 满足对应关系，同时进程运行时调用 sleep 延长进程生命周期。 |
| （4） | 矩阵计算 | 使用矩阵计算，来测试计算性程序的正常运行情况。 |
| （5） | 管道测试 | 测试管道的数据传输是否出错。 |
| （6） | 主动让出调度测试 | 使进程不停的让步于其他进程，看是否成功进行主动时间片的让出。 |

表 5-2 文件系统功能和性能的测试方法

| 序号 | 测试对象 | 测试方法 |
|---|---|---|
| 1、系统基本指标测试 | | |
| （1） | 文件系统启动测试 | 格式化 SD 卡为 FAT32 并分区，运行文件系统代码，检查文件系统是否可以正确对块设备进行校验并读入 FAT32 的基本信息。 |
| （2） | 兼容性测试 | 格式化 SD 卡为 FAT32，在 Linux 或者 Windows 下写入内容，应确保在本文件系统可以读出。本文件系统向 SD 卡写入内容，应确保在 Linux 或者 Windows 也可以正常访问。 |
| 2、功能测试 | | |
| （1） | 文件创建测试 | 在根目录创建文件。 |
| （2） | 文件读写测试 | 对创建的文件写入随机字符串，然后读出，进行比较。 |
| （3） | 文件清除测试 | 清除写文件内容，再次读取，检测读取长度是否为 0 |
| （4） | 目录创建测试 | 在根目录创建目录，同时创建多级目录 |
| （5） | 目录内文件读写测试 | 在多级目录下创建文件，进行随机字符串读写测试 |
| （6） | 文件搜索测试 | 搜索多级目录下的目录或文件，打开并查看内容是否这正确 |
| （7） | 文件/目录删除测试 | 删除文件，递归删除目录，检查删除后簇变化量是否正确 |
| （8） | 系统调用测试 | 使用赛方提供的系统调用测试程序进行验证 |
| 3、性能测试 | | |
| （1） | 大文件读写测试 | 该测试为了验证文件系统对大文件的操作是否可靠。测试分为 8 轮，随机生成大小为 4 至 5000 个扇区的字符串，向文件内写入，写入后读出，检查是否一致。 |

## 5.3 测试结果

表 5-3 用户程序功能指标测试结果

| 序号 | 测试项目 | 测试结果 |
|---|---|---|
| 1、功能指标测试 | | |
| （1） | 用户程序退出 | pid 返回值满足对应要求，同时子进程退出状态对应父进程获得的状态。 |
| （2） | 多彩字符测试 | 使用多种 CSI 控制序列，字符能够以不同颜色显 |

| | | |
|---|---|---|
| | | 示。 |
| （3） | 进程复制测试 | 高强度进行进程复制并且退出，pid 满足对应关系。 |
| （4） | 矩阵计算 | 使用矩阵计算，矩阵结果正确。 |
| （5） | 管道测试 | 测试管道的数据传输正确。 |
| （6） | 主动让出调度测试 | 使进程不停的让步于其他进程，成功。 |

表 5-4 文件系统的功能指标与性能指标测试结果

| 序号 | 测试对象 | 测试结果 |
|---|---|---|
| 1、系统基本指标测试 | | |
| （1） | 文件系统启动测试 | 文件系统可以正确启动，对 FAT32 格式设备进行组织 |
| （2） | 兼容性测试 | SD 卡在不同环境下使用无异常无差别，系统具备兼容性 |
| 2、功能测试 | | |
| （1） | 文件创建测试 | 文件系统可以正确创建文件 |
| （2） | 文件读写测试 | 文件系统可以正确读写，读写后在其他平台下可以正常访问 |
| （3） | 文件清除测试 | 文件系统可以正确清除文件 |
| （4） | 目录创建测试 | 文件系统可以正确创建目录 |
| （5） | 目录内文件读写测试 | 文件系统可以在多级目录下对文件正确读写 |
| （6） | 文件搜索测试 | 文件系统可以正确搜索存在的文件，在文件不存在时及时报错 |
| （7） | 文件/目录删除测试 | 文件可以正确删除文件目录，删除后可用簇数与创建前一致，且在其他平台下也无法找到删除的文件 |
| （8） | 系统调用测试 | 通过赛方所有文件系统相关的系统调用测试，共计 17 个 |
| 3、性能测试 | | |
| （1） | 大文件读写测试 | 通过大文件读写测试，累计对 6596 个扇区进行读、写、清除操作，耗时 6041ms |

# 6 总结与展望

## 6.1 工作总结

(1) Rust语言

(2) 多核操作系统

(3) 支持59条系统调用

(4) 高性能优化：内存弱一致性优化、lazy、CoW、文件系统双文件块缓存等优化等机制

(5) 信号机制：进程支持进程信号软中断。

(6) 支持C语言程序和Rust语言用户程序编写和运行（提供回归测试基础）

(7) FAT32虚拟文件系统

(8) 混合调试工具：Monitor（结合静态宏打印以及动态gdb特性）

(9) 详细项目文档、开发过程支持以及理解友好型代码构造和注释

## 6.2 未来展望

（1）我们期望能够将UltraOS在内部设计文档之外，设计教学向一步一步构建的操作系统实现文档，以及将对应的代码构建成适合实验操作的代码，并且按照阶段提供不同的代码框架。

（2）目前我们的SBI采用的是Luojia构建的RustSBI，这使得我们在构建的过程之中对于硬件初始化以及中断的设置等K210硬件平台特有的问题不甚了解，这对于之后的调试步骤显得愈发艰难。

（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已经能够做到几乎无错误，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UltraOS进行更深入的改进，以支持更稳定的运行。

（4）内核监视器：我们目前大多数只能从用户层次，或者内核的print信息来观察内核的状态。我们需要根据内核监视器，包括软件上以及硬件上的实现，来监视内核进程运行情况、内核内存情况、文件系统状况等等。

（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。

（6）外部设施支持：我们注意到班上资源有LCD显示屏以及摄像头，并且已经有项目对于外设进行开发，包括外接WIFI实现互联网通信等等，这都是将操作系统衍生至真正实用性的关键挑战。