



哈爾濱工業大學(深圳)

HARBIN INSTITUTE OF TECHNOLOGY, SHENZHEN

Titanix 操作系统

设计文档

参赛队名: Titanix

队伍成员: 曾培鑫、陈佳豪、任秦江

指导老师: 夏文、仇洁婷

全国大学生计算机系统能力大赛

操作系统赛

内核实现赛道

2023 年 05 月

摘要

Titanix(Titanic-nix) 是一个使用 Rust 编写的基于 RiSC-V 架构的**类 Unix 操作系统**，以**异步无栈协程架构**为基础，具有高效的调度效率，支持多核多任务。目前初赛的所有测试用例已经**满分通过**，下图是我们队伍（Titanix）的初赛测试通过情况：

操作系统大赛-内核实现赛道-区域赛

操作系统大赛-内核实现赛道-全国赛阶段1

比赛提交到排行榜更新有20秒左右的延迟

#	用户名	队伍	最后提交时间(ASC)	提交次数(ASC)	rank
1	202310464101015	你说对不队/ 河南科技大学	2023-04-18 14:30:26	22	102.0000
2	202318123101314	Titanix/ 哈尔滨工业大学 (深圳)	2023-05-04 21:50:35	17	102.0000
3	202310698101003	PLNTRY/ 西安交通大学	2023-05-12 01:43:31	9	102.0000

此外，Titanix 能支持 busybox 的部分功能，如 sh、ls、echo 等，以下是我们各模块的完成情况：

表 1 Titanix 的模块完成情况

模块	完成情况
进程管理	实现分时多任务 无栈异步协程调度 ； 实现多线程运行与回收； 实现 多核并行 ；
内存管理	实现虚拟内存，用户和内核共享地址空间； 实现 页缓存 和 块缓存 ，减少 IO 次数； 实现文件映射和匿名映射； 实现 懒分配 与 写时复制 ；
文件系统	实现虚拟文件系统，支持 FAT32 文件系统接入； 实现 Inode 缓存 ，减少 IO 次数； 利用 hash 加速 Inode 查找 ；
信号系统	初步实现信号机制， 包括 sigaction, kill, sigprocmask, sigreturn 等基础系统调用
多核支持及同步	实现 多核启动和调度 ； 实现 futex 和信号量；

目录

一、概述	3
1.1 Titanix 介绍	3
1.2 操作系统整体架构	4
1.3 目录和文件描述	4
二、Titanix 的设计与实现	7
2.1 Titanix 的进程管理	7
2.1.1 概述	7
2.1.2 进程控制块	7
2.1.3 线程控制块	8
2.1.4 线程调度	9
2.1.5 异常与中断	13
2.1.6 多核心管理	15
2.2 Titanix 的内存管理	15
2.2.1 概述	15
2.2.2 地址空间	15
2.2.3 内存映射	18
2.2.4 内存管理相关数据结构	18
2.2.5 懒分配与写时复制	19
2.2.6 用户地址检查	21
2.2.7 页缓存	22
2.3 Titanix 的文件系统	23
2.3.1 概述	23
2.3.2 虚拟文件系统	24
2.3.3 FAT32 文件系统	31
2.4 Titanix 的信号系统	37
2.4.1 相关数据结构	38
2.4.2 信号处理	38

三、总结与展望	40
3.1 工作总结	40
3.2 经验总结	40
3.3 未来计划	40

一、概述

1.1 Titanix 介绍

一个操作系统一般分为三个层级：U 态、S 态和 M 态。U 态即用户态，具有最少的权限，不能读写特权级寄存器，不能修改 S 态及 M 态的内存空间，只能通过系统调用或异常中断等方式与 S 态进行交互，常见的用户程序如 `vim`、`gcc` 等都是运行在 U 态；S 态即内核态，操作系统的关键部分，在这个模式下，内核需要统筹各个进程的调度协作、响应用户的系统调用请求、管理各个进程的内存空间以及控制设备的 IO 操作等；M 态为机器模式，比 S 态更加底层，负责完成硬件相关的初始化等工作，该层级常见的软件有 `open-sbi`、`rust-sbi` 等。

Titanix 的设计目标是利用 Rust 语言在系统级开发得天独厚的优势，构建一个符合 POSIX 规范的高性能可扩展的操作系统，同时尽可能地支持网络功能与现实应用，兼顾功能性与高性能。

Titanix 使用 Rust 语言进行开发，之所以选择 Rust，主要考虑到以下几个原因：

1. Rust 是一门内存安全的语言。对于 C/C++ 这样的手动管理内存的编程语言，我们在分配堆变量的时候需要调用 `malloc/new`，而当该变量使用完毕之后要手动调用 `free/delete` 回收内存，而这这就要求程序员需要关注所有堆变量的生命周期并及时将其释放，否则就会造成内存泄漏的问题，而过早的释放堆变量又可能造成“use-after-free”的问题；对于 Java/Go 这样的具有垃圾回收机制的编程语言，它们虽然可以自动回收不再使用的堆变量而不需要程序员操心，但回收的过程会“stop the world”，造成一定的性能开销；而 Rust 独特的所有权机制和借用检查，让编译器掌管变量的生命周期，使得变量的回收变得可控，同时也杜绝了“use-after-free”的问题，又不至于带来垃圾回收的开销；
2. Rust 优秀的开发体验与丰富的第三方库支持。Rust 优秀的 lsp 插件和合理规范的模块化设计让开发时无论是补全提示还是库间引用都有良好的体验；另外，`crates.io` 上有许许多多的 Rust 第三方库，举个例子，在 `xv6` 里我们可能需要手动实现一个类似空闲链表的内存分配器，但 Rust 里已经有像伙伴分配器这样的第三方库实现了，我们直接引入即可。

在开发 Titanix 之前，我们先完成了 MIT6.S081 XV6 课程实验，初步了解了一个操作系统的运行逻辑以及常见的优化手段（如懒分配与写时拷贝等）；然后又学习 `rCore-tutorial` 教程，了解了如何使用 Rust 进行 OS 内核开发，通过借鉴 `rCore-tutorial` 的启动流程和内存管理，我们已经能够实现简单的操作系统；后来学习到 FTL OS，被其无栈异

步协程调度框架所折服，开始重构代码，将进程调度逻辑完全重构，内存管理也改为用户内核共享地址空间，同时逐渐修改系统调用接口使其符合 POSIX 规范。不同于 FTL OS，我们希望内核更加的简洁，减少不必要的设计，具有相对详细的注释，以便于同学们的阅读与参考；另外，我们希望让 Titanix 更加注重功能性的拓展，如添加网络功能、适配更多的现实应用等，使内核具有更高的实用性和参考价值，目前内核大约有 13000 行左右的代码，实现了初赛要求的所有系统调用。

Titanix 目前以 rust-sbi 代理硬件层，通过 SBI 实现对硬件的控制，如串口输入输出、多核心启动、操作系统的关机等。关于内核部分，Titanix 实现了进程管理、内存管理、文件系统（包括虚拟文件系统和 FAT32 文件系统及其他辅助文件系统）、信号系统以及多核心同步功能；Titanix 以无栈异步协程调度为基础实现整个操作系统的调度运行；内核与用户共享地址空间，使得内核可以高效访问用户态地址；支持多核心并行调度运行；Titanix 实现了虚拟文件系统，为所有文件相关的操作提供了统一的接口，便于不同底层文件系统的扩展，另外，Titanix 设置了许多缓存（页缓存、块缓存等），尽量减少磁盘 IO 次数。

1.2 操作系统整体架构

如图 1 所示。

1.3 目录和文件描述

OS 目录树

```
| build.rs           // 将用户态的 shell 和 initproc 程序链接进内核
| Makefile          // 构建脚本
\---src
|   console.rs      // 控制台输入输出
|   entry.S         // Titanix 入口函数
|   lang_items.rs   // 内核崩溃时的处理
|   linker64.ld     // 内核编译链接脚本
|   main.rs         // Titanix 初始化处
|   sbi.rs          // SBI 相关调用
|   timer.rs        // 硬件时间相关函数
+---boards          // 开发板相关
|   mod.rs
|   qemu.rs         // Qemu 平台配置
+---config          // 各模块的配置参数
+---driver          // 驱动相关
+---executor        // 线程调度执行器
|   mod.rs
+---fs              // 文件系统模块（虚拟文件系统和 FAT32 文件系统）
|   | dirent.rs     // 内核中的文件目录类型，与用户态的 dirent 一致
|   | fd_table.rs
|   | file.rs       // 虚拟文件系统的文件类型
|   | file_system.rs // 虚拟文件系统中的抽象文件系统类型
|   | hash_name.rs  // 对 inode 的名字和父亲 inode 名字进行 hash
```

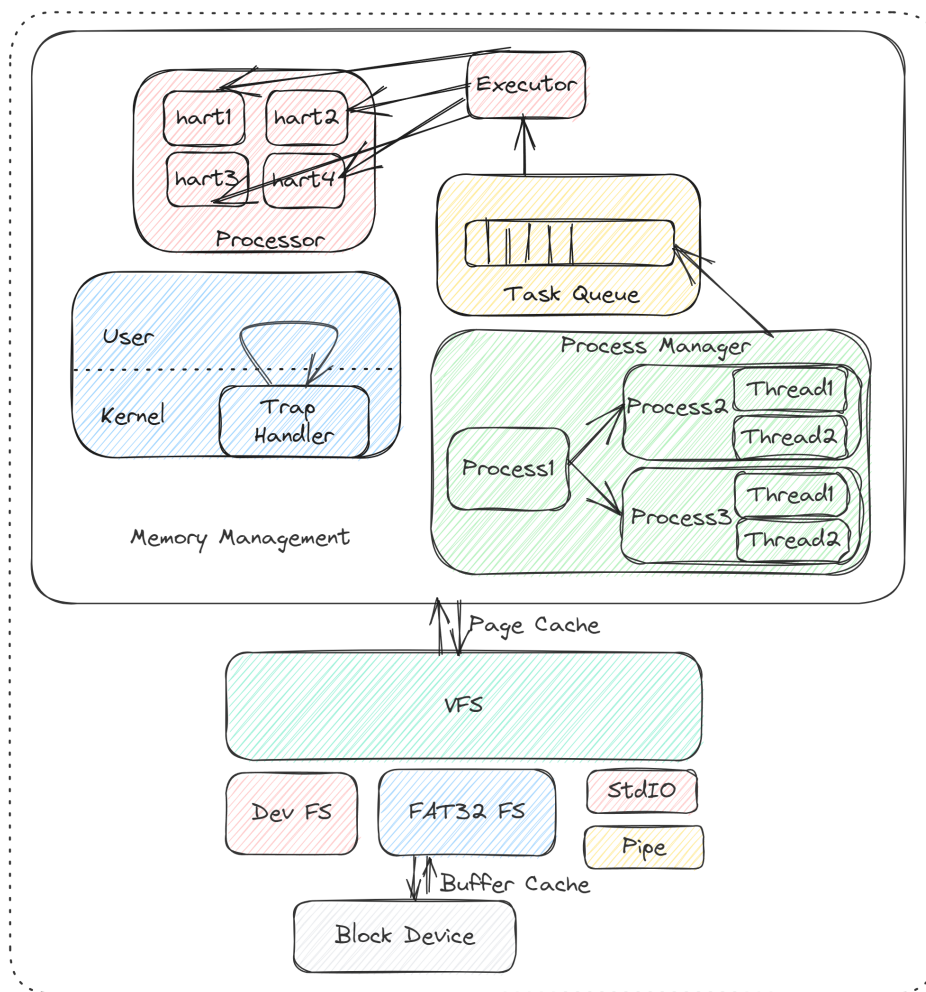


图 1 总体架构

```

| | inode.rs // 虚拟文件系统的 inode
| | stat.rs
| | mod.rs
| | pipe.rs // 管道
| | stdio.rs // 标准输入输出
| | uio.rs
| | utsname.rs
| +---devfs // 设备文件系统
| +---fat32 // FAT32 文件系统
+---mm // 内存管理模块
| | address.rs // 虚拟/物理地址相关
| | frame_allocator.rs // 物理页帧分配器
| | heap_allocator.rs // 堆分配器
| | mod.rs
| | page.rs // 文件页
| | page_cache.rs // 页缓存
| | page_table.rs // 页表
| | radix_tree.rs // 基数树, 用以组织页缓存
| | recycle_allocator.rs // 分配各种可回收 ID
| +---memory_set // 地址空间
| | mod.rs
| | page_fault_handler.rs // 页错误处理函数
| | vm_area.rs // 虚拟地址段

```

```

| \---user_check                // 用户地址检查
+---process                    // 进程管理模块
| | manager.rs                // 进程管理器
| | mod.rs
| | pid.rs                    // 进程 ID 管理
| \---thread                  // 线程管理
| | exit.rs                  // 线程退出处理
| | mod.rs
| | schedule.rs              // 线程调度相关
| | threadloop.rs            // 线程主循环函数
| | thread_resource.rs        // 线程相关资源
| | thread_state.rs           // 线程状态
| | tid.rs                   // 线程 ID 管理
+---processor                  // 处理器模块
| | context.rs
| | env.rs
| | hart.rs                  // 核心管理
| | mod.rs
+---signal                    // 信号处理模块
| | mod.rs
| | signal_context.rs         // 信号处理上下文
| | signal_handler.rs         // 信号处理函数
+---sync                      // 同步模块
| | cond_var.rs              // 条件变量
| | mod.rs
| | \---mutex                // 互斥锁
| | | mod.rs
| | | spin_mutex.rs
+---syscall                   // 系统调用模块
| | dev.rs                   // 设备相关的系统调用
| | fs.rs                    // 文件系统相关的系统调用
| | mm.rs                    // 内存管理相关的系统调用
| | mod.rs
| | process.rs               // 进程相关的系统调用
| | signal.rs                // 信号相关的系统调用
| | sync.rs                  // 同步相关的系统调用
+---trap                      // 异常处理模块
| | context.rs               // 陷阱上下文
| | mod.rs
| | trap.S
\---utils                    // 工具函数
| | error.rs                 // 封装 POSIX 标准错误返回值
| | hash_table.rs            // 哈希表
| | logging.rs               // 日志输出
| | mem.rs
| | mod.rs
| | path.rs                  // 文件路径相关函数
| | string.rs                // 字符处理函数
\---debug                    // debug 工具
| | mod.rs
| | stack_tracker.rs         // 栈追踪器

```


二、Titanix 的设计与实现

2.1 Titanix 的进程管理

2.1.1 概述

进程是操作系统中资源管理的基本单位，而线程是操作系统中调度的基本单位。在 linux 内核代码里，进程和线程都统一用 `task_struct` 结构体表示。不同于 linux 内核，Titanix 中将进程和线程分别用 `Process` 和 `Thread` 结构体表示，这样能更加清晰的处理同一进程内不同线程的创建与回收等过程。同一进程的不同线程共享地址空间（包括内核栈），后文会提到，Titanix 采用无栈协程的调度方式，所有线程（包括不同进程的线程）共享同一个内核栈，调度起来开销比较小。

2.1.2 进程控制块

os/src/process/mod.rs

```
/// Process control block
pub struct Process {
    /// immutable
    pid: PidHandle,
    /// mutable
    inner: SpinNoIrqLock<ProcessInner>,
}
```

我们将进程控制块分为可变和不可变的两个部分，后文会提到，进程控制块的所有权由其父进程和其包含的所有线程持有，在 Rust 中，对于这种有多个所有权的情况，我们需要用一个 `Arc(Atomic reference count)` 将其包裹，而 `Arc` 在 Rust 中默认是不可变的，因此对于可变部分，我们需要用互斥锁将其包裹；由于进程一创建便会分配一个进程 ID 且在其生命周期内不可变，因此我们将其归为不可变部分，对于可变部分，具体成员如下：

os/src/process/mod.rs

```
/// Process control block inner
pub struct ProcessInner {
    /// Whether this process is a zombie process
    pub is_zombie: bool,
    /// The process's address space
    pub memory_set: MemorySet,
    /// Parent process
    pub parent: Option<Weak<Process>>,
    /// Children processes
    pub children: Vec<Arc<Process>>,
    /// File descriptor table
    pub fd_table: FdTable,
    /// Allocate tid
```

```

pub tid_allocator: RecycleAllocator,
/// TODO: use BTreeMap to query and delete more quickly
pub threads: Vec<Weak<Thread>>,
/// Signal handlers for every signal
pub sig_handler: Arc<SpinNoIrqlLock<SigHandlerManager>>,
/// Pending sigs that wait for the prcoess to handle
pub pending_sigs: SigQueue,
/// UStack base of all threads(the lowest bound)
pub ustack_base: usize,
/// Addr -> Condvar map
pub addr_to_condvar_map: BTreeMap<usize, CondVar>,
/// Exit code of the current process
/// Note that we may need to put this member in every thread
pub exit_code: i8,
/// Current Work Directory
/// Maybe change to Dentry later.
pub cwd: String,
}

```

每个成员的作用如注释所述，值得注意的是，进程对所有孩子进程持有强引用（即所有权），而对父进程持有弱引用（没有所有权），这样做能防止循环引用导致内存泄漏；另外进程对其所有的线程也是持弱引用，因为某个线程可以在进程还未退出时先退出并释放内存。

2.1.3 线程控制块

```

os/src/process/thread/mod.rs

/// Thread control block
pub struct Thread {
    /// immutable
    pub tid: TidHandle,
    /// the process this thread belongs to
    pub process: Arc<Process>,
    /// whether the user specify the stack
    /// pub user_specified_stack: bool,
    /// mutable
    pub inner: UnsafeCell<ThreadInner>,
}

```

类似进程控制块，我们也将线程控制块分成可变和不可变部分，不可变部分包括线程 ID 和该线程所属进程的 Arc 指针，对于可变部分，我们采用了 UnsafeCell 而不是像进程控制块一样采用 SpinNoIrqlLock，原因是 UnsafeCell 可以不需要加锁从而修改内部成员，顾名思义，这一行为在编译器看来是不安全的，不安全是因为编译器无法帮我们保证不同线程对该结构体的互斥访问，但对于我们的内核来说，这个成员只会在当前核心进行访问，即不可能出现在某一线程里访问另一线程的这个成员，对于这个成员，其拥有的具体子成员如下：

os/src/process/thread/mod.rs

```

/// Thread inner,
/// This struct can only be visited by the local hart except the `terminated` field
/// which is the reason why it is an atomic variable
pub struct ThreadInner {
    // TODO: add more members
    /// Trap context that saves both kernel and user msg
    pub trap_context: TrapContext,
    /// Used for signal handle
    pub signal_context: Option<SignalContext>,
    /// When invoking `exec`, we need to get the ustack base.
    /// Note that ustack_base is the base of all ustacks
    pub ustack_base: usize,
    /// Thread state.
    /// Note that this may be modified by another thread, which
    /// need to be sync
    pub state: ThreadStateAtomic,
    /// Tid address, which may be modified by `set_tid_address` syscall
    pub tid_addr: Option<TidAddress>,
}

```

每个成员的具体作用如注释所述，值得注意的是，我们可能会出现在主线程里强制杀死别的子线程的情况，这个时候可能需要跨线程访问 ThreadInner 结构体并修改其 state 成员，因此我们将其设为原子变量，防止并发问题。

2.1.4 线程调度

本系统的线程调度模型采用的是无栈异步协程架构，协程即协作式多任务的子例程，协作式指的是协程的调度（挂起或运行）是由协程本身决定而不是被抢占式的；为什么说线程调度模型是协程架构呢？从用户态来说线程显然不符合协作式的概念（需要被内核抢占式调度），但从内核态来看，内核拥有线程的所有权，可以自行决定在某个时候将某个线程挂起或切换（如时钟中断的到来等），因此可以说 OS 中的线程对内核态来说便是协程。以下是无栈协程和有栈协程的区别：

2.1.4.1 有栈协程

每一个协程（即操作系统的线程）拥有自己独立的栈，每次进行协程的切换时需要修改栈指针，手动保存所有上下文信息如通用寄存器并切换为目标协程的通用寄存器，另外协程切换前后需要考虑是否有互斥锁或其他资源尚未释放，否则容易造成死锁。调度过程如图 2 所示。

2.1.4.2 无栈协程

所有的协程共用一个栈，每个协程在堆上维护一个状态机，协程的切换是通过轮询当前的状态，然后根据状态决定是否需要切换，下述代码为简单示例：

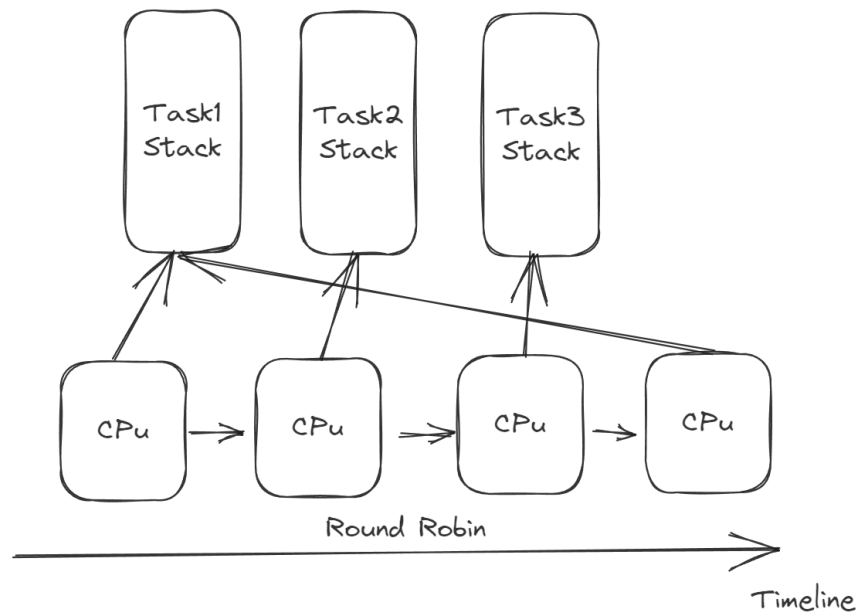


图 2 有栈协程调度

coroutine example

```

enum PollResult {
    Ready,
    Pending,
}

enum State {
    StateA,
    StateB,
    StateC,
}

struct Coroutine {
    state: State,
}

impl Coroutine {
    pub fn poll(&mut self) -> PollResult {
        match self.state {
            State::StateA => {
                /* Do something */
                self.state = State::StateB;
                return PollResult::Pending;
            }
            State::StateB => {
                /* Do something */
                self.state = State::StateC;
                return PollResult::Pending;
            }
            State::StateC => {
                /* Do something */
                return PollResult::Ready;
            }
        }
    }
}

```

上述代码中的 poll 函数即为协程的具体运行函数，不难看出其是一个状态机模型，上层每次调用 poll 时都可能改变其状态，从而推进其运行；总的来说，无栈协程的调度是通过函数返回然后调用另一个函数实现的，而不是像有栈协程那样直接原地更改栈指针，如下图 3 所示。

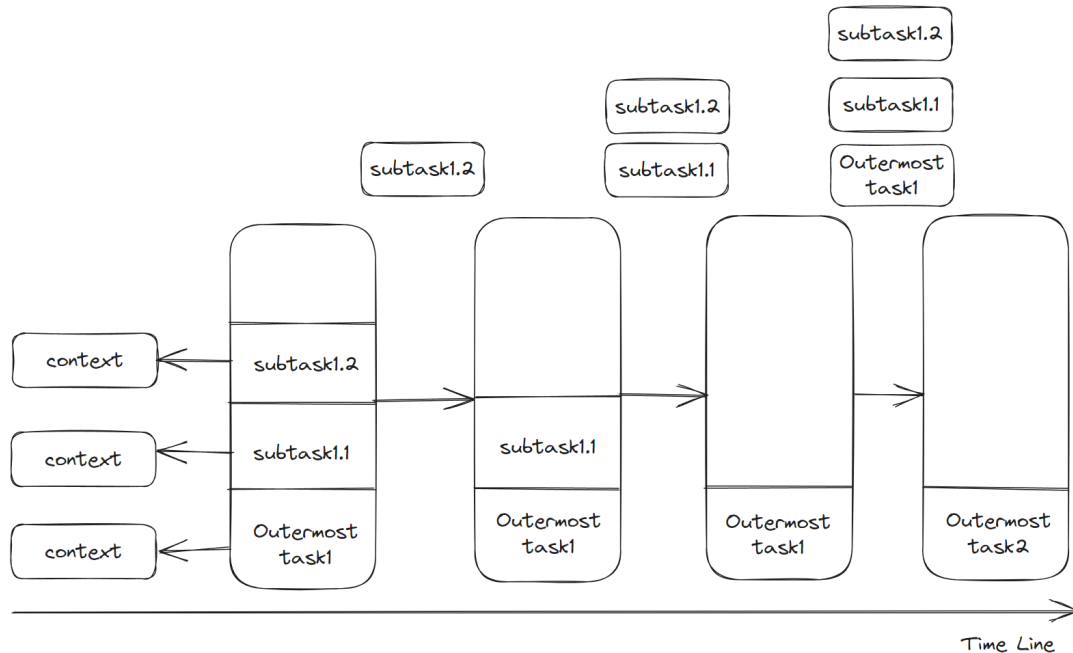


图 3 无栈协程调度

2.1.4.3 Rust 中的 async/await

在 Rust 中，将一个函数定义为 async 或者使用 async move 包裹的代码块即为一个无栈协程，称为一个 future。对于一个 future，Rust 会将其编译成类似上述状态机的形式，然后用户通过 await 使其开始运行（即调用 poll 函数），因此我们可以为每一个线程设置一个 future，大致框架如下：

os/src/process/thread/schedule.rs

```
impl<F: Future + Send + 'static> Future for UserTaskFuture<F> {
    type Output = F::Output;
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output> {
        let this = unsafe { self.get_unchecked_mut() };
        let hart = processor::local_hart();

        hart.push_task(&mut this.task_ctx);
        // run the `threadloop`
        let ret = unsafe { Pin::new_unchecked(&mut this.task_future).poll(cx) };
        hart.pop_task(&mut this.task_ctx);

        ret
    }
}
```

```
}
```

上述代码中的 `UserTaskFuture` 即为每个线程对应的无栈协程运行逻辑，在 `poll` 函数中，我们先获得当前核心（在多核心管理中详细介绍），做一些运行新线程的准备工作（如切换页表等），然后便可以开始轮询具体的线程处理函数，每个线程的运行函数实际上是一个死循环，一直运行直至线程退出，其具体逻辑大体如下：

os/src/process/thread/threadloop.rs

```
pub async fn threadloop(thread: Arc<Thread>) {
    loop {
        let trap_context = unsafe {
            let p = &mut *thread.inner.get();
            &mut p.trap_context
        };
        trap::trap_return(trap_context);
        trap::trap_handler().await;
        if thread.is_zombie() {
            break;
        }
    }
    // When the process becomes zombie, all of its threads should exit too
    handle_exit(&thread);
}
```

如上代码所示，一个线程的生命周期内做的事情就是不断重复“回到用户态执行用户态代码-> 由于中断或异常陷入内核进行处理”这个过程。

2.1.4.4 线程调度

介绍完上述关于无栈协程和 Rust 的 `async/await`，我们还需要一个执行器来调度这些协程 (future)，这里我们采用了第三方库 `async-task`，这个库可以构造一个 future 执行器来调度内核的所有任务。首先我们需要维护一个全局任务队列，然后执行器依次从全局队列中取出任务执行，只要我们的 OS 还有线程在跑（至少 `initproc` 永不消亡），那任务队列里就始终有任务，具体逻辑如下：

os/src/executor/mod.rs

```
/// Return the number of the tasks executed
pub fn run_until_idle() -> usize {
    let mut n = 0;
    loop {
        if let Some(task) = TASK_QUEUE.fetch_task() {
            // info!("fetch a task");
            task.run();
            n += 1;
        } else {
            break;
        }
    }
}
```

```

    }
  }
  n
}

```

2.1.4.5 线程唤醒

Rust 中的 future 在轮询的过程中会传入一个 waker 句柄，可以通过其来唤醒上层任务，我们的内核线程可以通过该句柄将某个线程唤醒，并且重新加入任务队列；

总体来看，线程调度是一个 Round-Robin 的过程，内核维护一个固定的时间片，每次时间片一到，会产生一个时钟中断，用户线程陷入内核，在 trap_handler 函数处 await 出去，然后上层执行器便从任务队列中取出下一个任务，不断重复此过程即可实现分时多任务。

2.1.5 异常与中断

OS 内核控制和调度用户程序以及各个外设硬件的方式就是异常和中断。对于 RISC-V 架构来说，stvec 寄存器会存储中断向量，每次有异常或中断发生时，pc 寄存器会被设置为 stvec 中存储的中断向量的值。因此我们需要做的就是实现一个异常或中断处理函数，然后将其地址存入 stvec 寄存器中。

2.1.5.1 返回用户态

返回用户态这一过程大体上来说会在两个地方用到：当一个进程在内核态被构建出来后首先要做的事情就是返回用户态去执行 elf 文件中的代码，此时要经历该过程；当一个进程因为异常或者中断陷入内核之后，要重新返回用户态，此时要经历该过程。由于 Titanix 的内核和用户共用页表，我们返回用户态的操作也比较简单。具体来说，可以把返回用户态这一步看成一个正常的函数调用，因此只需要手动保存 caller-saved 寄存器即可（其他寄存器会在编译的时候自动保存到内核栈中），代码如下所示：

os/src/trap/mod.rs

```

#[no_mangle]
/// Back to user mode.
/// Note that we don't need to flush TLB since user and
/// kernel use the same pagetable.
pub fn trap_return(trap_context: &mut TrapContext) {
    set_user_trap_entry();
    extern "C" {
        // fn __alltraps();
        fn __return_to_user(cx: *mut TrapContext);
    }
}

```



```
check_signal_for_current_process();
unsafe {
    __return_to_user(trap_context);
}
}
```

`__return_to_user` 是一个汇编函数，所做的事情就是手动保存所有的 caller-saved 寄存器，然后将用户态的所有通用寄存器（在下一部分陷入内核会讲）一并恢复，最后将 `TrapContext` 的地址保存在 `sscratch` 寄存器中。另外，值得注意的是，当一个进程刚刚被构建出来的时候，我们需要在内核态给他的用户态通用寄存器赋初值，默认是 0，但栈指针需要根据我们设定的用户态栈进行赋值。

2.1.5.2 陷入内核

在用户因系统调用、时钟中断、缺页中断等原因发生异常和中断时，会跳转 `stvec` 中存储的中断向量，我们需要保存用户态的上下文，具体包括 32 个通用寄存器和 `sepc` 寄存器，当返回用户态时保存的通用寄存器可以恢复现场，`sepc` 寄存器可以让用户态在发生异常的指令处或者下一条指令处继续执行。我们要想在中断向量中保存这些寄存器，由于此时已经处于内核态了，我们还需要一个内核结构体来保存，因此在陷入内核时需要知道该内核结构体的地址。RISC-V 中有一个特权寄存器 `sscratch` 可以用来存放核心相关的上下文的地址，在前文中有提到，我们再返回用户态时会把 `TrapContext` 的地址存入 `sscratch` 中，因此这个时候便可以用其来保存用户上下文。

2.1.5.3 陷阱处理函数

由于我们的内核是异步的，结合了 Rust 的 `async/await` 机制，因此自然要将陷进处理函数（即 `trap handler`）定义为异步函数，实际上，内核的进程切换点也是在 `trap handler` 中。

具体来说，我们在 `trap handler` 中对不同的中断或异常类型进行相应的处理，对于每种情况，可能的结果有：

1. 正常处理并返回用户态；
2. 发生用户态严重异常，导致进程直接退出，不会再返回用户态；
3. 时钟中断，当前进程切换到下一个进程（严格来说是线程），此时该进程会一直等到下一次调度到自己后才返回用户态；

2.1.6 多核心管理

对于 RISC-V 架构来说，每个核心有自己独立的一套寄存器，因此从总体上看，只需要给每个核心划分好内核栈，便可以开始进行并行调度运行，多核内存管理在下一章节讨论。这里我们重点关注多核运行，前面线程调度中提到，每次取出新的任务时，我们需要拿到当前的核心，而我们将每个核心的核心控制块地址存放在 `tp` 寄存器中，因此可以通过读取 `tp` 寄存器来获取当前核心，核心控制块的定义如下：

os/src/processor/hart.rs

```
pub struct Hart {
    hart_id: usize,
    /// Spare env ctx when in need(e.g. kernel thread or idle thread)
    spare_env_ctx: EnvContext,
    local_ctx: LocalContext,
    /// Every hart has its own kernel stack
    kstack_bottom: usize,
}
```

核心控制块的关键成员在于 `local_ctx` 成员，该成员可以在挂上各种各样的 per-CPU 的成员，目前有栈追踪器（用来记录调用栈）、页表和线程控制块，线程控制块即当前核心正在运行的线程，每次调度新的任务时需要修改 `satp` 值以启用新的页表，同时刷新 TLB。

2.2 Titanix 的内存管理

2.2.1 概述

在 Titanix 中，内核和用户共享地址空间，因此在陷入内核和返回用户态不需要像 rCore-tutorial 那样进行繁琐的处理，不需要 trampoline 页面，只需要正常跳转即可。对于内核的地址空间，我们采用直接映射，对于用户的地址空间，我们采用物理页帧随机映射。

2.2.2 地址空间

在 QEMU 平台上，内核的入口地址是在 `0x8020_0000`，若将 `0x8000_0000` 以下部分作为用户的地址空间，那用户态便只有 2G 的地址空间，若是在 U740 开发板上则无法完全利用其内存，又因为 RISC-V SV39 的规定，内存地址空间只能分布在 `0x0` 至 `0x3f_fff_fff` 和 `0xffff_ffc0_0000_0000` 至 `0xffff_fff_fff_fff`，因此我们将用户地址空间映射到 `0x0` 至 `0x3f_fff_fff`，内核地址空间映射到 `0xffff_ffc0_0000_0000` 至 `0xffff_fff_fff_fff`。

2.2.2.1 内核地址空间

内核地址空间见下图 4 所示，可以看到内核的入口地址位于 `0xffff_ffc0_8020_0000`，我们首先在链接脚本中将链接基址改为 `0x8020_0000`，由于生成的内核 elf 文件不是位置无关代码（或许可以通过编译选项设置为 PIC?），而 QEMU 默认跳转到的内核第一条指令为 `0x8020_0000`，因此我们需要在内核启动的时候就尽快将地址空间映射到高位，否则可能会因为跳转绝对地址导致内核崩溃。

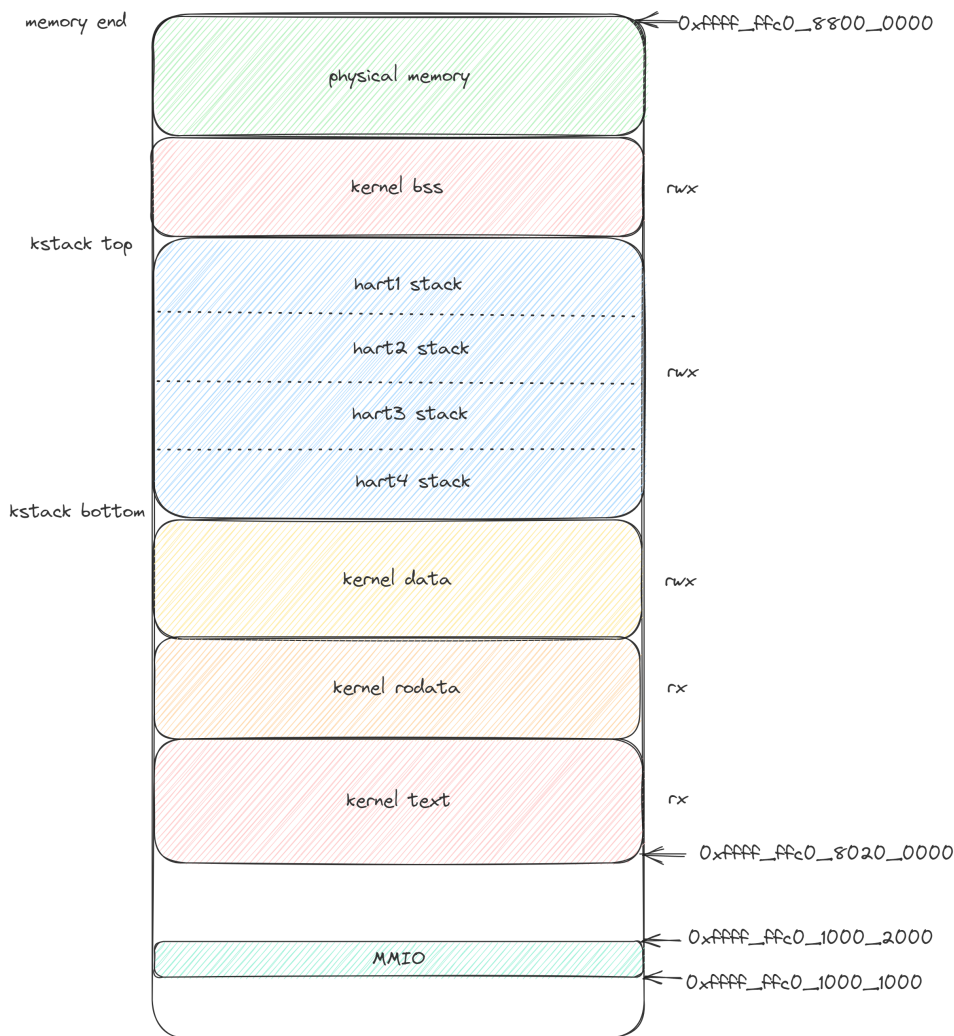


图 4 内核地址空间

2.2.2.2 进入内核

那么如何在启动的时候快速映射到高位呢？我们难道要用汇编手写一个 SV39 三级页表？其实没必要，在 RISC-V SV39 里，MMU 的翻译逻辑是逐级查找，一旦遇到页表项包含了 V 位，便停止查找，并不需要查满三层，这意味着我们可以直接映射一个巨页（即只用一级页表），手写一个根页表并不是一件非常复杂的事情，我们只需要将

0x8000_0000 和 0xffff_ffc0_8000_0000 这两个巨页添加到页表项即可, 具体逻辑如下所示:

os/src/entry.S

```
boot_pagetable:
# we need 2 pte here
# 0x0000_0000_8000_0000 -> 0x0000_0000_8000_0000
# 0xffff_ffc0_8000_0000 -> 0x0000_0000_8000_0000
.quad 0
.quad 0
.quad (0x80000 << 10) | 0xcf # VRWXAD
.zero 8 * 255
.quad (0x80000 << 10) | 0xcf # VRWXAD
.zero 8 * 253
```

另外, 我们将内核的栈段划分为多段作为每个核心的内核栈, 在进入内核的时候根据 sbi 传入的核心 id 计算出对应核心的内核栈的范围, 并将其值赋给 sp 指针即可。

2.2.2.3 用户地址空间

用户地址空间见下图 5 所示, 我们将文件映射及匿名映射的内存地址放置到用户地址空间的最高位, 将堆段放置到栈段之上, 其他的段都是在解析程序 elf 文件时映射的。

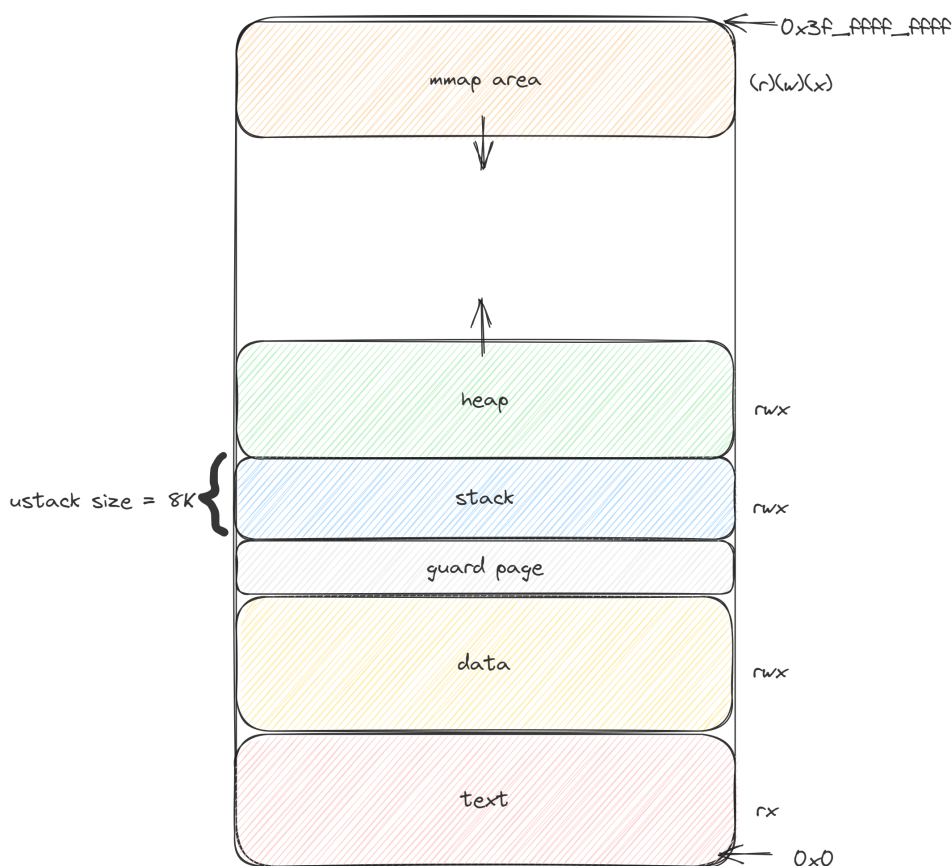


图 5 用户地址空间

2.2.3 内存映射

2.2.3.1 内核空间内存映射

对于内核空间来说，所有的段都是直接映射，这里的直接映射指物理页地址加上一个偏移量（即 `0xffff_ffc0`），包括物理页帧，因此在将物理页帧分配给用户时需要将地址减去偏移量才能得到真正的物理地址。

2.2.3.2 用户空间内存映射

对于用户空间来说，所有的段都是物理页帧随机映射，我们在内核中维护一个物理页帧分配器用以管理所有的物理页帧，对于每个物理页帧，我们采用了 Rust 中的原子引用计数 Arc 来维护，当物理页帧的引用计数为 0 时自动释放，我们通过覆盖物理页帧的析构函数来实现将空闲物理页帧回收至物理页帧分配器，这是一种 RAII 的思想，广泛运用在我们的内核中，同时 Arc 也大大简化了后面我们实现写时复制的步骤。

2.2.4 内存管理相关数据结构

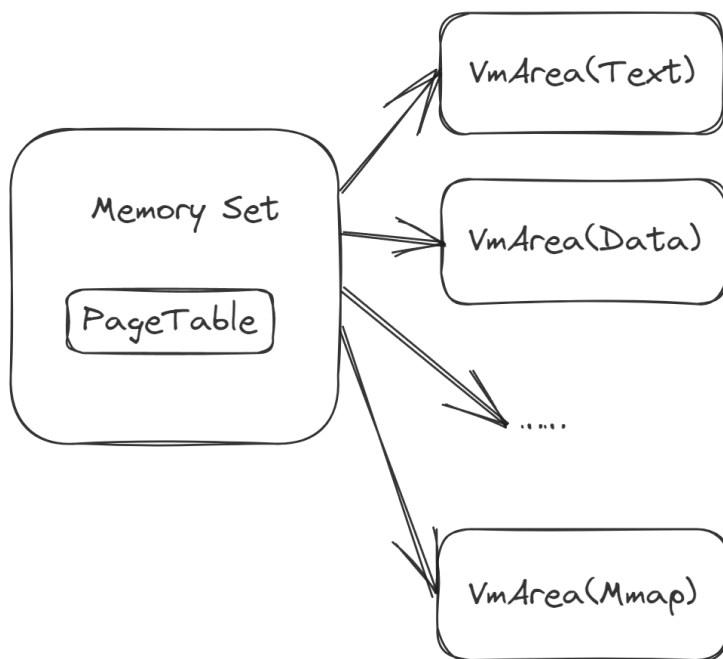


图 6 进程地址空间数据结构

下面介绍内核中内存管理相关的重要数据结构。

2.2.4.1 MemorySet

os/src/entry.S

```

/// memory set structure, controls virtual-memory space
pub struct MemorySet {
    /// we should ensure modifying page table exclusively(e.g. through process_inner's lock)
    /// TODO: optimization: decrease the lock granularity when handling page fault
    pub page_table: Arc<SyncUnsafeCell<PageTable>>,
    /// start vpn -> vm_area
    areas: BTreeMap<VirtPageNum, VmArea>,
    /// heap range
    pub heap_range: Option<HeapRange>,
}

```

如上所示，每个进程会有一个 MemorySet 成员表示其地址空间，该结构体包含一个页表，以及通过 B 树维护对各个内存段的索引，便于根据给定的虚拟地址快速查找、删除或修改其对应的内存段。

2.2.4.2 VmArea

os/src/entry.S

```

/// map area structure, controls a contiguous piece of virtual memory
pub struct VmArea {
    /// Vpn range
    pub vpn_range: VPNRange,
    /// We don't need to use lock because we've locked the process
    /// inner every time we handle page fault
    pub data_frames: UnsafeCell<FrameManager>,
    /// Map type
    pub map_type: MapType,
    /// Map permission
    pub map_perm: MapPermission,
    /// Mmap flags
    pub mmap_flags: Option<MmapFlags>,
    /// Page fault handler that is invoked when page fault
    pub handler: Option<Box<dyn PageFaultHandler>>,
    /// Backup file(only used for mmap)
    pub backup_file: Option<BackupFile>,
}

```

如上所示，VmArea 即代表某个内存段，各个成员的作用如注释所述，值得注意的是，VmArea 有一个 handler 成员，当发生页错误时，我们可以根据发生页错误的地址找出相应的 VmArea，然后再调用其 handler 成员处理这个页错误，从而让页错误的维护工作变得统一且具有可扩展性。

2.2.5 懒分配与写时复制

为了提高性能，我们应当尽可能地减少拷贝和申请内存的次数，目前 Titanix 里采用了懒分配和写时复制的技术来减少开销。

2.2.5.1 懒分配

目前 Titanix 里的懒分配主要有以下三个方面：

1. 用户栈的懒分配：在进程构建出来的时候，我们给用户进程划分一个虚拟地址栈空间但不实际分配物理地址，当用户访问栈空间时再通过缺页中断分配物理页。
2. 用户堆的懒分配：类似与用户栈的懒分配，当用户调用 `brk` 系统调用增长进程空间时，我们只增长虚拟地址空间而不实际分配物理内存，当用户真正读写该堆空间时再通过缺页中断进行物理页分配。
3. `mmap` 内存段的懒读取：当用户进行 `mmap` 系统调用时，我们记录下对应的文件指针以及映射的偏移量范围但不进行实际读取，当用户真正读写到该内存段时再通过缺页中断读取相应文件的相应位置的内容。

2.2.5.2 写时复制

当进行 `fork` 系统调用构造出新的进程时，我们不需要将父进程地址空间的全部内存拷贝一份，而是让子进程与父进程共享物理内存页，这样做的开销就只有修改页表了。注意到 RISC-V 中页表项的标志位中留了一个保留位，我们可以将其设置为 COW 位，在 `fork` 的时候需要将具有写权限的内存段的写标志位去掉，同时添上 COW 位。值得注意的是，在 `copy-on-write` 的时候我们还需要注意某个内存段是否是懒分配的内存段，若是，我们便不需要共享物理页，直接新增一个虚拟地址内存段即可。

另外，前文提到我们对物理页帧使用 Arc（原子引用计数）进行维护，在这里派上了用场，因为在 COW 的时候父子进程会同时持有对同一物理页面的所有权，对于一个物理页面，我们需要等到其引用计数为 0 的时候才可以将其回收进空闲物理页帧中，这里直接利用了 Arc 的特性。

2.2.5.3 页错误处理函数

懒分配和写时复制在后面还需要进行扩展，如 `elf` 文件懒加载等，而这项技术的关键在于通过缺页中断进行真正的分配或者加载，因此我们需要设计一个可扩展的统一的页错误处理方式以便于后期的维护与扩展。通过参考 Linux，我们为每个内存段 (`VmArea`) 设置一个页错误处理函数，利用 Rust 里的动态分发实现多态，从而可以使用统一的接口来存储页错误处理函数，该 `trait` 如下所示：

os/src/mm/memory_set/page_fault_handler.rs

```

/// General page fault handler
pub trait PageFaultHandler: Send + Sync {
    /// Handle the specific virtual page fault
    fn handle_page_fault(
        &self,
        va: VirtAddr,
        vma: &VmArea,
        page_table: &mut PageTable,
    ) -> GeneralRet<>;

    ///
    fn is_legal(&self, _scause: Scause) -> bool {
        todo!();
    }

    /// Used for cloning in `fork`
    fn box_clone(&self) -> Box<dyn PageFaultHandler>;
}

```

2.2.6 用户地址检查

在 Titanix 中，用户和内核共享地址空间，因此在访问用户态的内存时不需要同 Xv6 那样通过软件查询页表，而可以直接解引用访问。但直接解引用会带来一个问题，如果用户态传入的地址是一个不合法的地址，那在内核态直接解引用就会导致内核崩溃，因此对于用户态传入的地址，我们需要谨慎判断。我们采用这样一个方案，首先实现一个内核态异常的处理函数，当想要读写某个用户地址段时，先将修改中断向量的值为以下汇编函数的地址：

os/src/mm/user_check/check.S

```

// if pagefault occurs, return: (a0, a1) <- (1, scause).
.align 6
__try_access_user_error_trap:
    csrw sepc, ra    # ra -> __try_x_user_u8's return addr
    li a0, 1
    csrr a1, scause
    sret

```

设置完中断向量后，我们开始尝试读（写）用户地址，如果发生页错误，则会直接跳转到中断向量，也就是以上函数，该函数的逻辑很简单，只是把 a0 寄存器设置为 1，然后就返回（即 sepc 的值），这样我们就能够根据读（写）某个用户地址时 a0 寄存器是否为 1 来判断是否发生了页错误，若发生了页错误，则调用用户虚拟地址所在的 VmArea 的页错误处理函数进行处理。

os/src/mm/user_check/mod.rs

```

impl UserCheck {
    /// Create a new UserCheck
    pub fn new() -> Self {
        unsafe {
            stvec::write(_try_access_user_error_trap as usize, TrapMode::Direct);
        }
        let ret = Self {
            _sum_guard: SumGuard::new(),
        };
        ret
    }
}

impl Drop for UserCheck {
    fn drop(&mut self) {
        unsafe {
            stvec::write(trap_from_kernel as usize, TrapMode::Direct);
        }
    }
}

```

另外，如上所示，这里我们采用 RAII 的思想，在 UserCheck 构造函数中更改中断向量为新的值，在 UserCheck 析构函数中还原中断向量为旧的值。

2.2.7 页缓存

页缓存主要为文件系统服务，这里我们只介绍页缓存的读写逻辑以及更新方法。

2.2.7.1 文件页

文件页为内核读取文件的内存单位，具体结构如下：

os/src/mm/page.rs

```

/// Note that the process will visit one page through `Arc`
/// which maintains the ref cnt, so we can decide whether
/// one page can be evicted by `Arc::strong_count()`
pub struct Page {
    /// Mutable page inner
    pub inner: Mutex<PageInner>,
}

pub struct PageInner {
    /// Start offset of this page at its related file
    pub file_offset: Option<usize>,
    /// Data
    pub data: [u8; FILE_PAGE_SIZE],
    /// Data block state
    data_states: [DataState; FILE_PAGE_SIZE / BLOCK_SIZE],
    /// Inode that this page related to
    inode: Option<Weak<dyn Inode>>,
}

```

如上所示，每个文件页会按块缓存划分为几个部分，每次读写的时候通过偏移量计

算出对应的块缓存并加载该块缓存即可，从而最小程度上减少了 IO 次数，另外我们会维护每个块缓存跟底层设备的数据新旧关系，只有经过更新的块缓存才需要刷到块设备；值得注意的是，由于 mmap 也是以文件页为单位，所以当读写 mmap 对应的页时需要一次性将所有块缓存全部加载进来。

2.2.7.2 页缓存

os/src/mm/page_cache.rs

```
pub struct PageCache {  
    inode: Option<Weak<dyn Inode>>,  
    pages: RadixTree<Arc<Page>>,  
}
```

如上所示，每一个 inode（在后文文件系统部分会介绍）会维护一个页缓存，我们参考了 Linux，对所有页用基数树（类似于三级页表的思想）进行维护，这样可以高效地某个页进行查找、删除、替换等。

2.3 Titanix 的文件系统

2.3.1 概述

对于文件系统模块，我们的主要目标是使该模块结构合理清晰，实现简单，功能符合大赛要求，最主要的拥有良好的可扩展性。目前 Titanix 的文件系统模块的总体已经完成还有一些细节和 BUG 需要完善。

2.3.1.1 虚拟文件系统

Titanix 的文件系统模块借鉴了 Linux 中 VFS(虚拟文件系统)的设计,使得 Titanix 可以支持多种文件系统我们对文件系统和文件分别定义了统一的接口，open，write，read，mkdir 等系统调用的处理函数可以通过这些接口来对具体的文件系统和文件进行操作，而不需要考虑各个文件系统的实现细节。如果需要对 Titanix 支持新的文件系统，只需要为这个文件系统以及这个文件系统的文件实现统一的接口即可。

文件系统和文件接口是以 trait 的方式定义的一共有两类抽象接口：

1. 文件系统抽象接口：FileSystem trait，每一个文件系统都需要实现该 trait 该接口非常简单，主要的成员只有 root_inode，通过该接口可以获得文件系统的根索引。
2. 文件抽象接口：如果来实现 File trait，必须先实现其依赖的父 trait：Inode trait，而 Inode trait 要由要求实现 InodeDevice，InodeDevice 又要求注册的设备需要满足块设备或者字符设备的抽象，于是就有了虚拟文件系统的层级结构。

2.3.1.2 FAT32 文件系统

Titanix 支持了竞赛要求的 FAT32 文件系统，并将 FAT32 系统接入 VFS 中。Titanix 的 FAT32 文件系统实现了单独的文件分配表缓存，由于在访问一个文件时要访问文件分配表，而文件分配表只占用磁盘的一小块区域，将其缓存可以加速文件的访问；还实现了文件的 FAT 表项缓存，将一个访问过的文件的 FAT 表项按一定步长存储在相关的数组中，使文件可以随机访问，而不需要每次都遍历 FAT 表。

2.3.2 虚拟文件系统

VFS 的功能是负责定义内核和文件系统之间的接口规范，因此我们在实现 FAT32 文件系统之前进行了 VFS 的设计与编写。同时希望我们的 VFS 具有一定的可扩展性，可以使得不同的文件系统在实现了 VFS 规定的接口之后即可与我们的 Titanix 进行对接，并迅速投入使用。此外，我们希望内核代码具有易读性以及良好的可维护性，于是我们在 Linux 的 VFS 基础之上，对其设计和相应的结构体进行了简化。为此我们进行了以下设计：

2.3.2.1 Inode & Dentry

在 FAT32 文件系统当中没有 inode，而一些文件系统当中又有一些特殊的结构体，为了简化设计，VFS 当中只设计 inode 作为与下层实际文件系统相关联的结构体，并在 inode 基础之上抽象出文件类型（file），以及文件系统（filesystem）；在 file 基础上抽象出默认文件（default file）、标准输入输出（stdio）、管道（pipe）和文件树（fd_table），见下图 7。

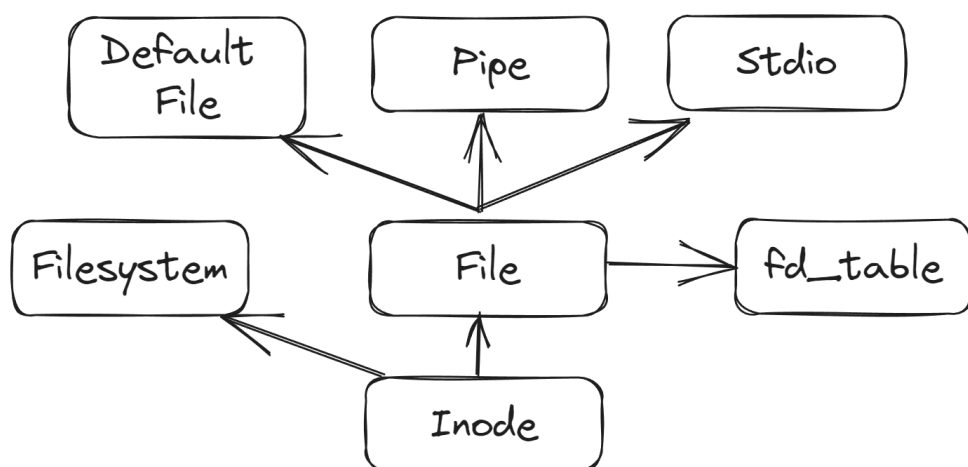


图 7 VFS 框架

对于 Inode，由于 Titanix 中不存在 dentry 的概念，原本由 dentry 的负责的路径转实

实际 inode 的过程 (lookup)，现在应该交由 inode 负责。于是为了后续对 inode 内容的更改的便捷，我们设计了以下 InodeMeta 结构体：

os/src/fs/inode.rs

```
pub struct InodeMeta {
    /// inode number
    pub ino: usize,
    /// data address
    pub data: usize,
    /// type of inode
    pub mode: InodeMode,
    /// device id (only for block device and char device)
    pub rdev: Option<usize>,
    /// inode's device
    pub device: Option<InodeDevice>,
    /// path to this inode
    pub path: String,
    /// name which doesn't have slash
    pub name: String,
    /// a inode's unique id
    pub uid: usize,
    pub inner: Mutex<InodeMetaInner>,
}
```

并将可能需要更改的对象放在了 InodeMetaInner 里面，这里给出 InodeMetaInner 结构体的设计：

os/src/fs/inode.rs

```
pub struct InodeMetaInner {
    // pub offset: usize,
    /// inode's file's size
    pub size: usize,
    /// last access time, need to flush to disk.
    pub st_atim: TimeSpec,
    /// last modification time, need to flush to disk
    pub st_mtim: TimeSpec,
    /// last status change time, need to flush to disk
    pub st_ctim: TimeSpec,
    /// hash name (Note that this doesn't consider the parent uid)
    pub hash_name: HashName,
    /// parent
    pub parent: Option<Weak<dyn Inode>>,
    /// brother list
    pub brothers: BTreeMap<String, Weak<dyn Inode>>,
    /// children list
    pub children: BTreeMap<String, Arc<dyn Inode>>,
    /// page cache of the related file
    pub page_cache: Option<PageCache>,
    /// data len
    pub data_len: usize,
}
```

为什么要这样做呢？这是考虑到 rust 这门语言的特性。rust 对于对象的可变性有着较高的要求，对于可变对象的可变引用，如果想要保证对象在并行线程之间的访问的安全

全性，需要使用智能指针来保证（我们使用 Arc 来将可能需要更改的对象包裹），但是智能指针（Arc）要求对象为不可变，这样的话，如果想要更改对象的值，我们只能通过 unsafe 块来包裹线程不安全的代码，并自己手动保证线程之间的访问安全。我们最开始是这样写的，但是后面发现这样会导致代码难以读懂，并且导致指针之间的关系复杂且曲折。实际上，这种写法是违背 rust 的特性的。rust 这门语言希望你不要随意对 Arc 中的对象进行更改，如果一定需要，请“上锁”。那么有什么办法可以对某一对象在保持 Arc 的引用的同时还可以更改其中的某些值？答案就是将这些可能需要更改的对象抽象出来，作为一个 Inner 对象，并在原来的对象中添加该 Inner 对象，但是需要用所包裹保证更改时的线程安全，这样就有了我们的解决办法。

这样做还有另外一个好处，通过将可变对象和不可变对象分开，使得我们可以在访问可变对象的同时，只对 Inner 上锁，让外部的不可变对象的访问可以并行执行，极大的提高了访问的并行性。

2.3.2.2 Name Hash

由于文件系统的文件组织具有层次化的特点，往往都会存在父目录和子目录这样的关系，所以在 Linux 当中，对于文件查找，使用了哈希表的技术来加快文件名到对应的 inode 的查找。

Titanix 效仿 Linux，同样对 inode 和实际文件名（字符串类型）之间建立一层哈希映射，以便于快速地进行文件查找。为此我们设计了 HashName 结构体，该结构体完成了文件名（str 类型）到 u64 类型的映射，当我们知道文件名之后，就可以调用该结构体实现的 hash_name 方法进行哈希值的计算，并从构建的全局哈希表当中获取该 inode 的 Arc 引用。该结构体如下所示：

```
os/src/fs/hash_name.rs
```

```
#[derive(Clone, PartialEq)]
pub struct HashName {
    pub name_hash: u64,
    pub parent: usize,
    pub name: Arc<str>,
}
```

你可能注意到了，我们的 HashName 结构体中存在 parent 的字段，为什么要设置这个字段呢？这是因为文件系统的层次性让我们可以在文件查询的时候做一些“小手脚”。通常来将，我们查找文件是将以字符串标识的路径，转换成对应的 inode，那么这个过程可以通过从根 inode 开始进行逐层查找，所以一般来讲，我们是通过父目录找到下一级的目录，然后逐级查找到对应的文件（inode），那么我们就可以使用父目录和子目录

的名字来一起做哈希，这样可以使得我们的哈希能够铺得更加平，从而在查找的时候能够在更加小的范围内进行精确匹配，使得查找的速度加快。

为了实现该结构体的一些 hash 函数，我们参考了 Github 开源仓库 rust-hash-table 中对于哈希表的实现。核心方法 hash_name 的实现如下：

os/src/fs/hash_name.rs

```
pub fn hash_name(parent: Option<usize>, name: &str) -> HashName {
    let parent_ptr = match parent {
        Some(p) => p as u64,
        None => 0 as u64,
    };
    HashName {
        name_hash: Self::myhash(parent_ptr, Self::str2num(name)),
        parent: parent_ptr as usize,
        name: Arc::from(name),
    }
}
```

2.3.2.3 Inode Cache

通过 Name Hash 来加速查找可以提高查找的速度，但是频繁的访问磁盘永远不是一个好的决定，于是 Titanix 采用了一贯的用于协调内外存访问速度不匹配问题的解决办法——开辟缓存！

我们通过设置全局对象 INODE_CACHE，来对可能会使用的 inode 进行缓存，由于需要保证线程之间的并行访问安全，我们需要对该对象上锁。对于 INODE_CACHE，他应该完成某一个文件的 inode 的文件名哈希值与 inode 自身的映射管理，于是我们在设计 Titanix 时，将 INODE_CACHE 定义为以下形式：

os/src/fs/inode.rs

```
lazy_static! {
    pub static ref INODE_CACHE: Mutex<HashTable<usize, Arc<dyn Inode>>> =
        ↪ Mutex::new(HashTable::new());
}
```

可以看见，我们在申明该全局对象时，采用了 lazy allocation 的方式以帮助我们更好得实现内存管理，这里使用了 rust 的 lazy_static 模块。

2.3.2.4 路径解析

对于路径解析，Titanix 效仿 Linux 实现 inode 的 lookup 查找，利用文件名哈希值来加速查找，并利用 INODE_CACHE 来减少访问磁盘的次数，我们给出了以下 lookup 方法：

os/src/fs/inode.rs

```

fn lookup(&self, this: Arc<dyn Inode>, name: &str) -> Option<Arc<dyn Inode>> {
    let key = HashName::hash_name(Some(self.metadata().uid), name).name_hash as usize;
    let value = INODE_CACHE.lock().get(&key).cloned();
    match value {
        Some(value) => Some(value.clone()),
        None => {
            debug!(
                "cannot find child dentry, name: {}, try to find in inode",
                name
            );
            let target_inode = self.try_find_and_insert_inode(this, name);
            match target_inode {
                Some(target_inode) => Some(target_inode.clone()),
                None => None,
            }
        }
    }
}

```

从该方法的参数可以看出，这是一个成员函数，如何调用该方法呢？需要一个 inode 对象通过“.”的方式来访问该成员方法。从这里就可以看出我们的 lookup 的逻辑，实际上是父亲 inode 调用 lookup 函数查找对应的孩子 inode 的过程。从上面的代码不难看出，我们会先尝试在 INDOE_CAHCE 当中查找是否存在对应的 inode，然后如果不存在，才开始调用 try_find_and_insert_inode 函数，试图在实际文件系统当中查找该 inode，但是如果查找失败，那么返回值为 None，交由上层判断处理。经过上面的分析，我们给出 lookup 函数的运行示意图如下图 8。

对于 try_find_and_insert_inode 函数，我们将其作为 VFS 与实际文件系统相衔接的部分，这一方法负责调用 VFS 提供给实际文件系统的一些接口来实现从实际文件系统中查找对应的 inode，由于调用该方法往往意味着在内存的 INODE_CACHE 当中没有找到该 inode，所以该方法顺带将找到的 inode 添加到 INODE_CACHE 当中。以下给出该方法的代码：

os/src/fs/inode.rs

```

fn try_find_and_insert_inode(
    &self,
    this: Arc<dyn Inode>,
    child_name: &str,
) -> Option<Arc<dyn Inode>> {
    let key = HashName::hash_name(Some(self.metadata().uid), child_name).name_hash as
    ↪ usize;
    self.load_children(this);
    debug!(
        "children size {}",
        self.metadata().inner.lock().children.len()
    );
    let target_inode = self
        .metadata()

```

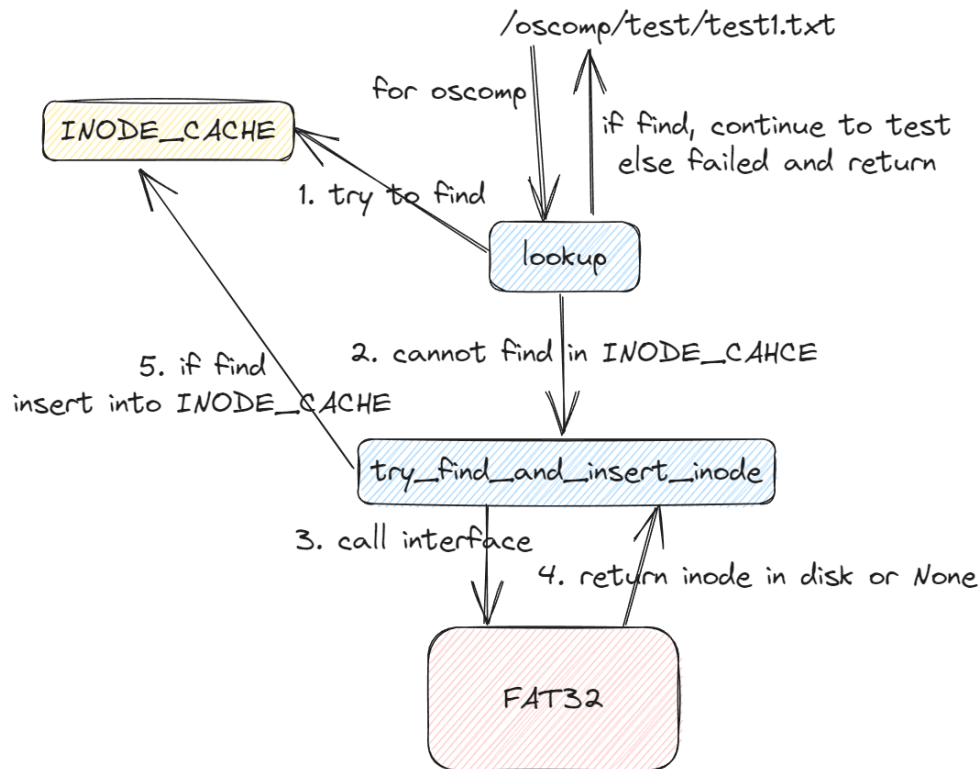


图 8 lookup 函数流程

```

    .inner
    .lock()
    .children
    .get(child_name)
    .cloned();
    match target_inode {
        Some(target_inode) => {
            // find the inode which related to this subentry
            INODE_CACHE.lock().insert(key, target_inode.clone());
            Some(target_inode.clone())
        }
        None => {
            debug!("Cannot find {} in children", child_name);
            None
        }
    }
}

```

这里的 `load_children` 方法就是提供给实际文件系统的接口，用于向内存当中载入给定 inode 的孩子 inode。

在上面单层查找的基础之上，我们实现了从根 inode 开始查找的代码：

os/src/fs/inode.rs

```

/// Look up from root(e.g. "/home/oscomp/workspace")
pub fn lookup_from_root(
    // file_system: Arc<dyn FileSystem>,
    path: &str,
) -> Option<Arc<dyn Inode>> {
    let path_names = Path::path2vec(path);
    let root_fs = FILE_SYSTEM_MANAGER
        .fs_mgr
        .lock()
        .get("/")
        .cloned()
        .expect("No root fs is mounted");
    let mut parent = root_fs.metadata().root_inode.clone().unwrap();
    for name in path_names {
        match parent.lookup(parent.clone(), name) {
            Some(p) => parent = p,
            None => return None,
        }
    }
    Some(parent)
}

```

该方法并不是一个关联方法，而是一个静态方法。抽象出该方法的目的是为了编写内核系统调用时能够更加有逻辑，并将大量重复代码进行复用。

2.3.2.5 文件系统抽象

Titanix 为所有的文件系统建立一层抽象，通过让实际文件系统实现该 `FileSystem` trait 来将自己接入 Titanix。为了在 `mount` 的时候能够区分不同的文件系统类型，Titanix 将自己支持的文件系统类型直接硬编码到了内核当中，我们设计了 `FileSystemType` 这个枚举类型：

os/src/fs/file_system.rs

```

#[derive(Clone)]
pub enum FileSystemType {
    VFAT,
    EXT2,
    NFS,
}

impl FileSystemType {
    pub fn fs_type(ftype: String) -> Option<Self> {
        match ftype {
            vfat => Some(Self::VFAT),
            ext2 => Some(Self::EXT2),
            nfs => Some(Self::NFS),
            _ => None,
        }
    }
}

pub fn new_fs(&self) -> impl FileSystem {
    match self {
        Self::VFAT => {
            // let fs = FAT32FileSystem::new();

```



```
        let fs = TestFs::new();
        fs
    }
    - => {
        todo!()
    }
}
}
```

目前 Titanix 只能支持 FAT32 文件系统，后续还会支持更多类型的文件系统。

所有的文件系统通过实现 Titanix 中的 `FileSystem` 中的方法，可以实现与内核的对接，我们为实际文件系统实现了默认的初始化方法，但是这个只针对 FAT32 文件系统，后续还需要进行修改。实际文件系统通过将 `write_inode`、`sync_fs` 等方法实现来完成 Titanix 需要的文件交互动作。

2.3.3 FAT32 文件系统

Titanix 实现了竞赛要求的 FAT32 文件系统。

FAT32 文件系统是由微软设计的 FAT12、FAT16 文件系统的后继版本，将 FAT 项由之前的 12 位、16 位扩展为 32 位，同时加入了 FSInfo 块等功能，并对磁盘布局作出了优化。其将磁盘中存储数据的区域划分为簇 (Cluster)，使用文件分配表 (File Allocation Table, FAT) 维护一个文件的簇链信息，将文件名等信息存储在目录项 (Directory Entry) 中，而没有单独的 Inode。FAT32 因为使用 32 位表项的文件分配表而得名，虽然每个表项只使用了 28 位存储有效数据。FAT32 的设计中，为了考虑到向前兼容性，维持了 MS-DOS 操作系统中使用的“8DOT3” (8 个字符长的文件名 + 一个点号 + 3 个字符长的扩展名) 的文件名存储格式，并通过定义“长目录项”以支持更长的文件名。不同于类 unix 文件系统提供的较完整的“读、写、执行”权限管理，FAT32 没有提供完善的权限管理，只有只读、隐藏的标记；此外 FAT32 没有提供内置的符号链接和硬链接功能，文件名中支持的字符集也比其它文件系统要小。针对这些特性，Titanix 的 FAT32 采用了一些机制以兼容 FAT32 标准和类 unix 系统上的 VFS。

下面介绍 FAT32 的整体设计和各个主要模块的设计。

2.3.3.1 整体设计

下图 9 为 Titanix FAT32 的整体结构。

一个 FAT32 文件系统对象是一个 `FAT32FileSystem` 结构体，这个结构体定义如下：

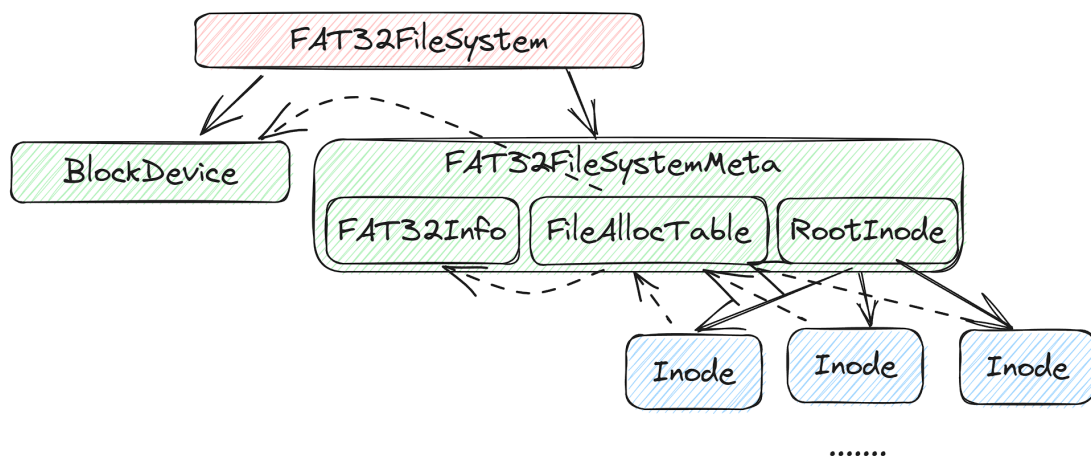


图9 FAT32 整体结构

os/src/fs/fat32/mod.rs

```
pub struct FAT32FileSystemMeta {
    info: Arc<FAT32Info>,
    fat: Arc<FileAllocTable>,
    root_inode: Arc<FAT32Inode>,
}

pub struct FAT32FileSystem {
    block_device: Arc<dyn BlockDevice>,
    meta: Option<FAT32FileSystemMeta>,
}
```

FAT32FileSystem 结构体包含其下层块设备的指针，以及其元信息。构造一个结构体，只需要传入其下层设备的原子引用计数（Arc），此时只是构造了一个结构体对象，没有进行读写、初始化磁盘的操作。在设计中，一个文件系统的挂载、卸载与其结构体的生命周期是分开的，用户需要显式调用 mount 和 unmount 来将其挂载或卸载。挂载时，文件系统将会尝试读取启动扇区的信息，创建一个文件分配表的缓存和根目录的 Inode 以加载文件系统。目录的 Inode 中可以包含其内容的 Inode 的引用。用户通过获取文件系统根目录的 Inode，加载其所有子文件或目录，以递归地获取文件系统的所有文件。

FAT32 文件系统基本信息结构体 FAT32Info 定义如下：

os/src/fs/fat32/fat32info.rs

```
#[derive(Copy, Clone, Default)]
pub struct FAT32Info {
    pub bk_bootsector_id: usize,
    pub fsinfo_sector_id: usize,
    pub fat_start_sector: usize,
    pub fat_sector_count: usize,
    pub fat_count: usize,
    pub data_start_sector: usize,
    pub sector_per_cluster: usize,
}
```

```
pub tot_sector_count: usize,  
pub tot_cluster_count: usize,  
pub root_cluster_id: usize,  
}
```

这些成员分别是：

1. `bk_bootsector_id`: 备份引导扇区号。FAT32 中记录文件系统基本信息的引导扇区位于 0 扇区，并且为了冗余，将其开始的若干扇区备份到另一位置。
2. `fsinfo_sector_id`: FSInfo 扇区号。FAT32 相比 FAT12 和 FAT16，增加了 FSInfo 扇区，用于存储和簇分配有关的信息。
3. `fat_start_sector`: FAT 表初始位置，这在引导扇区记录中对应保留扇区数量。FAT32 将磁盘分为三部分：保留扇区、FAT 表、数据区。
4. `fat_sector_count`: 单个 FAT 表占用扇区数。
5. `fat_count`: FAT 表的个数。FAT32 中也将 FAT 表冗余备份，若干个相同的 FAT 表都在 FAT 表区中。
6. `data_start_sector`: 数据区起始位置。这个字段是挂载过程中算出来的，因为后续在转换簇号和扇区号中经常被用到。
7. `sector_per_cluster`: 一个簇的扇区数。这个数是 2 的幂。FAT32 将连续的几个扇区作为一个簇，以簇为单位分配和释放存储空间。
8. `tot_sector_count`: 磁盘的总扇区数。
9. `tot_cluster_count`: 磁盘的总簇数。这个字段也是计算得出的，因为后续在处理簇号会经常用到。计算数据区的扇区数，除以一个簇的扇区数。
10. `root_cluster_id`: 根目录的簇号。不同于 FAT12 和 FAT16 有单独的区域存储根目录的目录项，它将根目录的目录项也存储到数据区，与其它目录同等对待。

由于这些字段在文件系统格式化后通常是不会改变的，为方便其它成员（FAT 表、Inode 等）使用，这个成员设置为原子引用计数的形式，以让多个结构体共享其中的数据。

2.3.3.2 文件分配表及缓存

Titanix 的 FAT32 在设计过程中，考虑到兼容 FAT32 标准，使用了单独的缓存区缓存文件分配表，并将文件分配表的读写、分配释放簇块封装为一个结构体，即 `FileAllocTable`。文件分配表将文件分配表的读写以及缓存抽象为一层，由这一层完成簇分配和 FAT 读写的所有操作。文件分配表的结构体定义如下：

os/src/fs/fat32/fat.rs

```

#[derive(Clone, Copy, PartialEq, Eq, Debug)]
enum FATSectorBufferState {
    Unassigned, // 未分配
    Assigned,   // 已分配, 未修改
    Dirty,      // 已分配, 已修改未写回
}

struct FATSectorBuffer {
    sector_no: usize,
    data: [u32; FATENTRY_PER_SECTOR],
    state: FATSectorBufferState,
}

pub struct FATMeta {
    free_count: usize,
    nxt_free: usize,
}

struct FATBufferCache {
    data: Mutex<LinkedList<(usize, Arc<Mutex<FATSectorBuffer>>>>>,
    block_device: Arc<dyn BlockDevice>,
    info: Arc<FAT32Info>,
}

pub struct FileAllocTable {
    block_device: Arc<dyn BlockDevice>,
    info: Arc<FAT32Info>,
    fatcache: FATBufferCache,
    fatmeta: Arc<Mutex<FATMeta>>,
}

```

文件分配表保存了块设备、文件系统基本信息的指针，包含一个块缓存的定义，同时包含了文件分配表信息的互斥锁的指针。文件分配表信息 FATMeta 结构体中的 free_count 和 nxt_free 代表空闲簇数以及上一个被分配簇的编号，与 FAT32 中 FSInfo 的两个字段相对应，文件系统挂载时会尝试加载这两个字段。

FATBufferCache 是分配表的缓存结构。FATBufferCache 除了块设备和文件系统基本信息的指针外，还持有一个块缓存的链表。块缓存包含了一个文件分配表扇区的编号（其逻辑地址相对于文件分配表首扇区的逻辑地址的差）、其内容和状态，分别为未分配、已分配未修改、已修改未写回。

我们在块设备中已经实现了一个块缓存，为什么我们还要写一个单独的文件分配表缓存呢？

1. 文件分配表的访问频率比数据区的访问频率高。访问一个文件需要先访问文件分配表再访问数据区，而文件分配表相比数据区来说占用空间很小，因此实现文件分配表的缓存可以加速这一经常性事件。
2. FAT32 中文件分配表有备份。FAT32 的文件分配表通常在磁盘上有两个备份（备份数量由启动扇区中 NumFAT 字段指定，通常这一值都为 2）。而块缓冲没有实现一个

块在磁盘上有两个备份的情况。因此需要单独实现回写的逻辑，以便于文件分配表块被替换出时将所有备份都回写。

3. 便于扩展功能。在文件分配表缓存中，我们集成了簇空间的分配与回收的功能。文件系统为文件和目录分配空间时只需要调用分配功能而不需要在其它位置实现。

2.3.3.3 目录项解析与文件名匹配

为了兼容 8DOT3 文件格式的历史遗留问题，FAT32 设置了长目录项。一个文件在其父目录中的目录项由若干 32 字节的目录项组成，其中最后一个为短目录项，存储文件的基础信息；而前面若干个为长目录项，专用于存储文件名。文件名是以 2 字节的 OEM 名称存储的，如中文以 GBK 编码按 2 字节存储，而在 Titanix 中没有指定专门的编码字符集，因此为了设计简便，在文件名解析中只识别 8 位的编码，对于小于 128 的编码即 ASCII 码按照 FAT32 的规定处理其可用性，对于大于 127 的编码也如同 ASCII 处理。

考虑到访问目录的系统调用，FAT32 一次解析一个目录中所有目录项。FAT32 中一个文件名有一个 8DOT3 短名和一个通用的长名，短名存储在最后的短目录项中，长名存储在长目录项中。FAT32 不允许一个目录中任意一个短名和长名冲突（就像其它文件系统中不允许一个目录中文件名冲突一样），并且比较时不区分大小写。对于查找文件的操作，FAT32 同时匹配短名和长名，这意味着可以通过短名或长名找到一个文件。对于新增文件的操作，如果文件名不符合 8DOT3 规则，则 FAT32 提取其基本文件名并加入数字后缀作为其自动生成的短名。

显然，出现冲突短名是概率很低的事情，而多数情况下一个目录中不会有太多文件，为了设计方便，Titanix 使用朴素匹配维护一个目录下的文件名信息。具体说是将一个目录下所有文件名（含长名、短名）放入一个集合，需要创建新名时在集合中查找是否冲突即可。不足之处在于 FAT32 的文件名不区分大小写，并且使用平衡树维护的集合的复杂度多一个 \log ，因此自己实现一个字典树维护文件名集合可能是更好、更快的做法。

为一个长名创建其对应的短名时，由于短名的长度很短（8DOT3，即 11 个字符长），因此可以暴力枚举其生成的字符后缀（如果需要生成），直到找到不冲突的项。尽管这么做在文件较多时会很慢，但是是最容易想到也容易实现的做法，文件名冲突的概率也很低。

2.3.3.4 Inode

FAT32 中没有索引节点（Inode）的概念，而为了方便上层对文件的访问，我们抽象

出了 Inode 的概念。FAT32 中的文件或目录，由其在父目录中的一系列目录项、其占用的所有簇和这些簇对应的 FAT 项组成；而设计中的 Inode，存储了其父目录中目录项位置信息、其占用簇链的第一个簇。在初始化 Inode 时只存储其第一个簇（如果有），在真正要访问这个文件时再去加载全部簇。

如果修改一个文件的访问时间、文件名等信息，这个信息是存储在父目录的目录项中的。为了方便修改存储在其父目录的数据块中的数据，FAT32 的索引节点保留了对父目录的文件的弱引用计数。在 FAT32 的目录中，目录项也是像文件数据一样按照簇链顺序排列的，因此对 FAT32 目录中目录项的读写可以直接复用对文件读写的模块。

Inode 结构体如下：

os/src/fs/fat32/inode.rs

```
#[derive(Copy, Clone)]
pub enum FAT32FileType {
    Regfile,
    Directory,
}

pub struct FAT32InodeMeta {
    filename: [u8; 256], // 长文件名
    short_filename: [u8; 11], // 短文件名
    file_raw_attr: u8, // 文件（原始）属性
    crt_time: FAT32Timestamp, // 创建时间 s
    acc_time: FAT32Timestamp, // 访问时间
    wrt_time: FAT32Timestamp, // 保存时间
}

pub struct FAT32Inode {
    ftype: FAT32FileType,
    fat: Arc<FileAllocTable>,
    meta: Mutex<FAT32InodeMeta>,
    content: Mutex<FAT32File>,
    father: Option<Weak<FAT32Inode>>,
    child_lname: Mutex<BTreeMap<String, Arc<FAT32Inode>>>>,
    child_sname: Mutex<BTreeMap<String, Arc<FAT32Inode>>>>,
}
```

ftype 表示当前 Inode 表示一个普通文件还是一个目录；fat 表示文件系统文件分配表的指针；文件分配表结构体中也保存了 BlockDevice 和 fatinfo 的指针，可以通过 fat 间接访问这两个元素；FAT32InodeMeta 存储了文件的基本信息：文件名（含短文件名和长文件名）、文件在文件系统中保存的属性、访问时间、修改时间与创建时间等。content 封装了对文件的读写操作，其保存了文件包含的所有簇链。father 是其上层目录的弱引用计数（而根目录没有上层目录，因此使用了 Option）。child_sname 和 child_lname 使用了两个 BTreeMap 索引一个目录下的所有子目录和文件的弱引用计数。一个通过短文件名索引，另一个通过长文件名索引。将长文件名和短文件名分开索引，是为了便于在存储到磁盘时将每个文件只存储一份，因为每个子文件都一定有一个短文件名。

这里本意是使用字典树（Trie）索引子文件，因为 FAT32 匹配文件名不区分大小写。

但考虑到访问字典树时每一次迭代都需要加锁,效率较差,因此先使用简单的 BTreeMap 直接索引 String。

在创建一个 Inode 时,除了子目录之外的内容都会被初始化。对于一个普通文件,我们将 FAT32File 中的读写接口暴露出来,就可以完成读写了。对于目录,我们在需要加载其子文件时,从 FAT32File 中按顺序读取、创建所有子文件、子目录的 Inode 并插入 BTreeMap 的索引项中。保存一个目录内的信息时,先递归保存目录下子目录的信息,然后利用短名 BTreeMap 索引,将其所有子目录写回到其存储空间中。

FAT32File 提供了对文件读写以及存储空间分配的封装,下面是 FAT32File 的定义:

os/src/fs/fat32/file.rs

```
pub struct FAT32File {  
    fat: Arc<FileAllocTable>,  
    clusters: Vec<usize>,  
    size: Option<usize>,  
}
```

其存储文件分配表的指针,使用一个可变数组 (Vector) 存储其所有使用的簇。对于普通文件,保存了文件大小 size。对于目录,因为 FAT32 的目录项中没有记录一个目录使用簇个数的信息,其在加载所有簇编号后会自动计算其占用总空间信息。

FAT32File 提供了调整文件大小、读写文件的接口。调整文件大小时,其先判断是增大还是减小文件,然后计算应该分配或释放多少块,调用文件分配表提供的接口,由 FileAllocTable 结构体修改文件分配表。读写文件时通过文件分配表查询应该读写哪些簇,从而推算出对应磁盘块的编号,调用 BlockDevice 提供的接口读写。

2.4 Titanix 的信号系统

信号 (Signal) 是 Unix 家族中一个古老的通信机制,主要用来通知进程某个特定事件的发生,或者是让进程执行某个特定的处理函数。说它古老,是因为它在第一代 Unix 系统中就已经存在了。信号是进程间的异步通信机制。

2.4.1 相关数据结构

os/src/signal/mod.rs

```
pub struct SigHandlerManager {
    sigactions: [SigActionKernel; SIG_NUM],
}
#[derive(Clone, Copy)]
pub struct SigActionKernel {
    pub sig_action: SigAction,
    pub is_user_defined: bool,
}
#[derive(Clone, Copy)]
#[repr(C)]
pub struct SigAction {
    pub sa_handler: fn(usize),
    pub sa_flags: usize,
    pub sa_restorer: usize,
    pub sa_mask: [SigSet; 1],
}
```

以上几个数据结构就是信号处理的核心，我们在内核里对 POSIX 定义的所有信号都实现相应的信号处理方式。对于 SigAction 结构体，我们需要让他符合 Linux 在 RISC-V 架构下的定义，sa_handler 即用户自定义处理函数，sa_flags 即相应的标志位，sa_restorer 为当用户自定义处理函数运行完毕后跳转的地址，sa_mask 为运行该信号处理函数时需要阻塞的信号。对于 SigActionKernel 结构体，我们添加一个用户自定义的标志来区分是默认处理函数还是用户自定义处理函数。

2.4.2 信号处理

在用户陷入内核态并最终返回用户态之前，我们检查一遍该进程的所有待决信号，并且逐一处理：

os/src/signal/mod.rs

```
fn handle_signal(signo: usize, sig_action: SigActionKernel) {
    debug!("handle signal {}", signo);
    if sig_action.is_user_defined {
        current_trap_cx().sepc = sig_action.sig_action.sa_handler as *const usize as usize;
        // a0
        current_trap_cx().user_x[10] = signo;
        if sig_action.sig_action.sa_restorer != 0 {
            // ra
            current_trap_cx().user_x[1] = sig_action.sig_action.sa_restorer;
        }
    } else {
        // Just in kernel mode
        (sig_action.sig_action.sa_handler)(signo);
    }
}
```

如上所示，我们在对某一个待决信号进行处理时，首先判断它的信号处理函数是默

认信号处理函数还是用户自定义处理函数，若是前者，我们只需要按照正常函数调用的方式直接调用相应的处理函数即可（当然这里应该改为异步函数更为合理，但目前暂未实现）；若是后者，我们需要保存原来的 `TrapContext`（即原来的返回用户态的上下文），然后将 `sepc` 改为用户自定义处理函数的地址，最后返回用户态，之后用户态处理函数结束后会调用 `sigreturn` 系统调用重新返回内核态，这个时候我们可以把之前存的用户态上下文重新恢复回去再返回用户态。

三、总结与展望

3.1 工作总结

1. 实现进程管理和内存管理，以无栈协程的方式高效调度任务；
2. 实现虚拟文件系统，将具体文件系统与内核解耦合；
3. 支持多核调度运行；
4. 基本实现初赛要求的相关系统调用；
5. 实现不同等级的调试日志；
6. 实现栈追踪器，便于内核崩溃时获取调用栈；

3.2 经验总结

1. 每次更新页表之后应当及时刷新 TLB，否则会出现各种诡异的内存 Bug；
2. 在 `exec` 出新的进程的时候，对进程的栈初始化应当符合 POSIX 标准，所有的辅助变量、环境变量及命令行参数在栈中的位置不能随意改变；
3. 在 Debug 模式下多插桩，这样在内核崩溃的时候可以看到更多的调用栈信息；
4. 手写汇编代码需要仔细检查关键寄存器如 `sp` 是否正确赋值，否则会发生难以理解的 Bug；
5. 用户态内存地址需要仔细检查，防止不合法地址导致内核崩溃；

3.3 未来计划

1. 目前内核中的同步锁只有互斥锁，未来需要加入睡眠锁和读写锁，尤其是在文件系统部分读写锁较为重要；
2. 修改 `async-task` 的调度逻辑，让由于 IO 唤醒的线程优先调度；
3. 适配 `busybox` 和 `libc`，完善相关系统调用的功能；
4. 压力测试，发现并解决内核中一些安全性 Bug；
5. 在 HiFive U740 平台上无法使用 `rust-sbi`，因此需要适配 `opensbi`；
6. 提高多核稳定性，发现并解决一些潜在的多核运行带来的 Bug；
7. 增加网卡驱动以及网络协议栈，实现网络功能；