



BTD OS

Bite The Disk

刘卓敏，宋相呈，刘潞

June 7, 2023

Contents

1	概述	3
1.1	比赛准备与调研	3
1.2	需求分析与完成情况	3
1.3	系统设计与整体架构	4
2	进程管理	4
2.1	引导进程的装载	4
2.2	TaskControlBlock(PCB)	8
2.3	进程描述符 (PidHandle)	8
2.3.1	多核 PID 池优化	9
2.4	进程的生成与调度	9
2.4.1	子进程的创建和写时复制 (COW)	9
2.4.2	单/多核调度策略	10
2.5	信号与进程的退出	12
2.6	进程的资源回收	13
2.6.1	进程结束的时机	13
2.6.2	通过 Rust 生命周期实现资源回收	13
3	内存管理	13
3.1	分页系统	13
3.1.1	页表的硬件机制	13
3.1.2	管理多级页表	14
3.2	地址空间	15
3.2.1	地址空间设计	15
3.2.2	内核地址空间与用户程序地址空间	15
3.3	页帧管理器	16
3.4	地址转换	17
3.5	Mmap 管理器	17
3.6	懒加载与懒分配	18
4	文件系统	19
4.1	FAT32 库	19
4.1.1	块设备接口层	20
4.1.2	块缓存层	20
4.1.3	文件系统描述层	22
4.1.4	虚拟文件层	27
4.2	内核中的文件系统模块	27
4.2.1	文件抽象	27
4.2.2	在内核中接入 FAT32	28
4.2.3	内核虚拟文件	28
4.2.4	绝对路径模块	29
4.2.5	其他模块与系统调用说明	29

5 问题与解决 30

5.1 工具链默认不支持乘法指令 30

5.2 VirtIOBlk 物理内存不连续时导致缓存数据丢失 30

5.3 FAT32 文件系统规范问题 30

5.4 多核启动顺序 30

5.5 栈空间过小 31

5.6 页引用计数更新 31

6 未来与展望 31

6.1 页面置换 31

6.2 大页 31

7 尾声 31

1 概述

BTD OS 是使用 Rust 语言开发的宏内核操作系统，可运行于 RISC-V64 平台，并实现了中断与异常处理、进程管理、内存管理以及文件系统等操作系统基本模块。目前，BTD OS 支持在 QEMU 虚拟环境中运行。

BTD OS 的总体目标是设计一款简洁小巧、结构清晰、基本功能完善、可拓展性良好的操作系统，旨在体现操作系统基本功能与核心思想，便于他人提供借鉴学习，以加深对操作系统基本原理与概念的理解。

1.1 比赛准备与调研

在比赛准备之前，我们的团队中有两位同学已经对 Rust 语言有初步的接触。在学习 Rust 的过程中，我们了解到了许多 Rust 特有的语言特性，例如独特的所有权机制、模式匹配与错误处理方式、无需手动内存管理但无 GC、性能比肩 C/C++，以及具有与 C 一样的硬件控制能力，同时也大大强化了安全编程和抽象编程能力。

恰好在学习 Rust 的过程中，我们发现了一个非常好的 Rust 入门项目，即 rCore-Tutorial-v3。于是我们参加了 rCore 的操作系统训练营。在其他学习者和助教老师的帮助下，我们完成了部分实验，进一步加深了对操作系统基本原理以及 rCore 部分实现细节的理解。因此，在准备比赛时，我们毫不犹豫地选择基于 rCore-Tutorial-v3 以 Rust 语言开发内核。

在比赛准备过程中，我们很幸运地在往届优秀作品中发现了本校的参赛获奖作品 RongOS。我们期望能获得上届学长的支持与答疑，因此在实现过程中，我们主要参考了 RongOS¹ 和 rCore-Tutorial-v3 os5-ref 版本²。

1.2 需求分析与完成情况

根据 2023 年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的赛题要求，目前 BTD 支持的系统调用如下：

类型	系统调用名			
进程	SYS_CLONE	220	SYS_EXECVE	221
	SYS_WAIT4	260	SYS_EXIT	93
	SYS_GETPPID	173	SYS_GETPID	172
内存	SYS_BRK	214	SYS_MUNMAP	215
	SYS_MMAP	222		
文件系统	SYS_GETCWD	17	SYS_PIPE2	59
	SYS_DUP	23	SYS_DUP3	24
	SYS_CHDIR	49	SYS_OPENAT	56
	SYS_CLOSE	57	SYS_GETDENTS64	61
	SYS_READ	63	SYS_WRITE	64
	SYS_LINKAT	37	SYS_UNLINKAT	35
	SYS_MKDIRAT	34	SYS_UMOUNT2	39
	SYS_MOUNT	40	SYS_FSTAT	80
其他	SYS_TIMES	153	SYS_UNAME	160
	SYS_SCHED_YIELD	124	SYS_GETTIMEOFDAY	169
	SYS_NANOSLEEP	101		

Table 1: 系统调用表

¹<https://gitlab.eduxiji.net/19061120/oskernel2022-segmentfault>

²<https://github.com/rcore-os/rCore-Tutorial-v3/tree/ch5>

1.3 系统设计与整体架构

RISC-V 架构中一共定义了 4 种特权级, RISC-V 架构中, 只有 M 模式是必须实现的, 剩下的特权级则可以根据跑在 CPU 上应用的实际需求进行调整。BTD OS 一共涉及 M/S/U 三种特权级, 可以根据运行时的特权级分为三个部分:

- 用户/应用模式 (U, User/Application): 用户应用程序运行在 U 态。
- 监督模式 (S, Supervisor): 操作系统的内核运行在 S 态, 是用户程序的运行环境。
- 机器模式 (M, Machine): 监督模式执行环境 (SEE, Supervisor Execution Environment) 运行在 M 模式上, 如在操作系统运行前负责加载操作系统的 Bootloader –RustSBI, 是操作系统内核的运行环境。

在 RISC-V 中, 各模式使用 `ecall` 指令与对应模式下的 `ret` 指令进行特权级切换, 对于用户模式 (即用户态) 切换到监督模式 (即内核态), 主要是使用内核提供的提供了 ABI (Application Binary Interface) 接口, 即 BTD OS 实现的系统调用。对于监督模式切换到机器模式, 我们选择使用 RustSBI³, 它是运行在更底层的 M 模式特权级下的软件, 是操作系统内核的执行环境。

2 进程管理

2.1 引导进程的装载

在系统初始化完成后, 会执行第一个引导任务, 该任务是在内核构建时同步编译, 并装载到内核中 (是内核的一部分) 的:

```
// kernel/src/task/initproc.rs

global_asm!(include_str!("../initproc.S"));
lazy_static! {
    /// 引导 pcb
    pub static ref INITPROC: Arc<TaskControlBlock> = Arc::new({
        extern "C" {
            fn initproc_entry();
            fn initproc_tail();
        }
        let entry = initproc_entry as usize;
        let tail = initproc_tail as usize;
        let siz = tail - entry;
        let initproc = unsafe { core::slice::from_raw_parts(entry as *const u8, siz) };
        let path = AbsolutePath::from_str("/initproc");
        let inode = fs::open(path, OpenFlags::O_CREATE, CreateMode::empty()).expect("initproc create failed!");
        inode.write_all(&initproc.to_owned());
        TaskControlBlock::new(inode)
    });
}
```

需要注意的是, 虽然我们通过 `fs::open` 来打开了 `initproc`, 但其实 `initproc` 本身并不存在于提供的 SD card 中。

³<https://github.com/rustsbi/rustsbi>

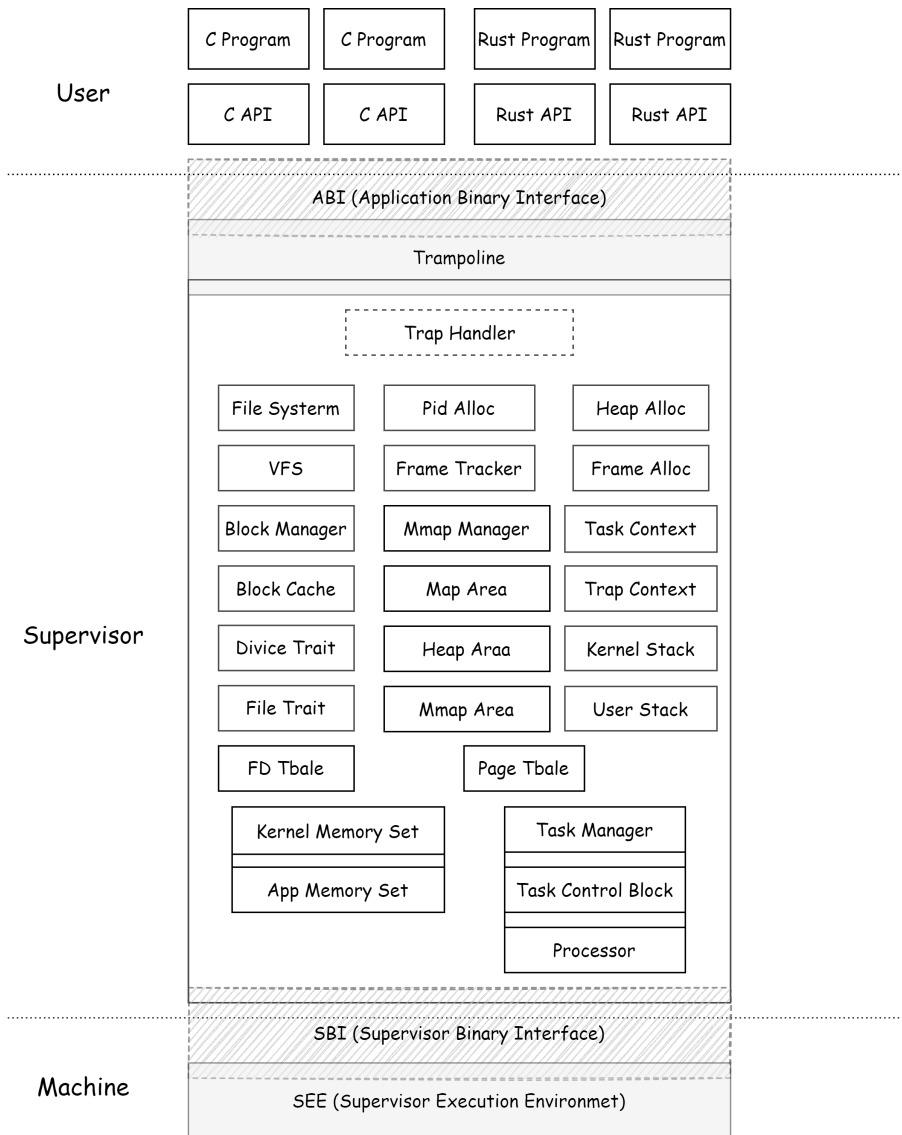


Figure 1: 整体系统架构

```
# kernel/src/initproc.S

.align 3
.section .data
.globl initproc_entry
.globl initproc_tail
initproc_entry:
    .incbin "../misc/tests_booter/bin/tests_booter"
initproc_tail:
```

可以看到，引导程序通过 `.incbin` 伪指令打包进了内核的 `.data` 段，成为了内核的一部分。

```
// kernel/src/task/initporc.rs

lazy_static! {
    pub static ref ROOT_INODE: Arc<VirFile> = {
        let fs = FileSystem::open(BLOCK_DEVICE.clone());
```

```

        // 返回根目录
        Arc::new(root(fs.clone()))
    };
}

```

这里 `FileSystem::open` 会通过一个块设备通过 `VirtIOBlk` 间接地与 SD card 进行交互，而 `FileSystem::open` 后返回的结构体是对 SD card 上文件系统的抽象表示，在这里其本身记录了 SD card 上 FAT32 格式文件系统的相关信息，为后续我们构建内核内存中的文件句柄 `Fat32File` 提供必要的信息和支持。

这里 `fat32::vfs::root` 会通过给定的 `FileSystem` 返回一个磁盘文件在内存中的映射 `VirFile`，之后我们进一步封装 `VirFile`，从中构造出一个具有一系列实用方法的 `Fat32File` (内核内存中的文件句柄)。

到此为止，我们已经获取到 `initproc` 在内核中的句柄，这个句柄本质是 `initproc` 在内核内存中的一种表示形式。

为了保证后续对于 SD card 上文件读写操作的一致性，也是为了方便实现，我们将内核 `.data` 段中的 `initproc` 先写入 SD card 上，之后用时再从 SD card 上进行读取 (和对待其他位于 SD card 上的文件一样，由此保证了对文件操作的一致性)。

```

// kernel/src/task/initproc.rs

let initproc = unsafe { core::slice::from_raw_parts(entry as *const u8, siz) };
let path = AbsolutePath::from_str("/initproc");
let inode = fs::open(path, OpenFlags::O_CREATE, CreateMode::empty()).expect("initproc
    create failed!");
inode.write_all(&initproc.to_owned());

```

下一步操作，就是从 SD card 上加载刚才写入的 `initproc`，将其在内存中实例化为一个 `TaskControlBlock`，并将其加入全局任务管理器 `TASK_MANAGER` 中等待内核调度。

```

// kernel/src/task/initproc.rs

TaskControlBlock::new(inode)

```

目前为止，整个引导进程的建立流程就结束了，该流程大致上可以用下面的流程图来表示：

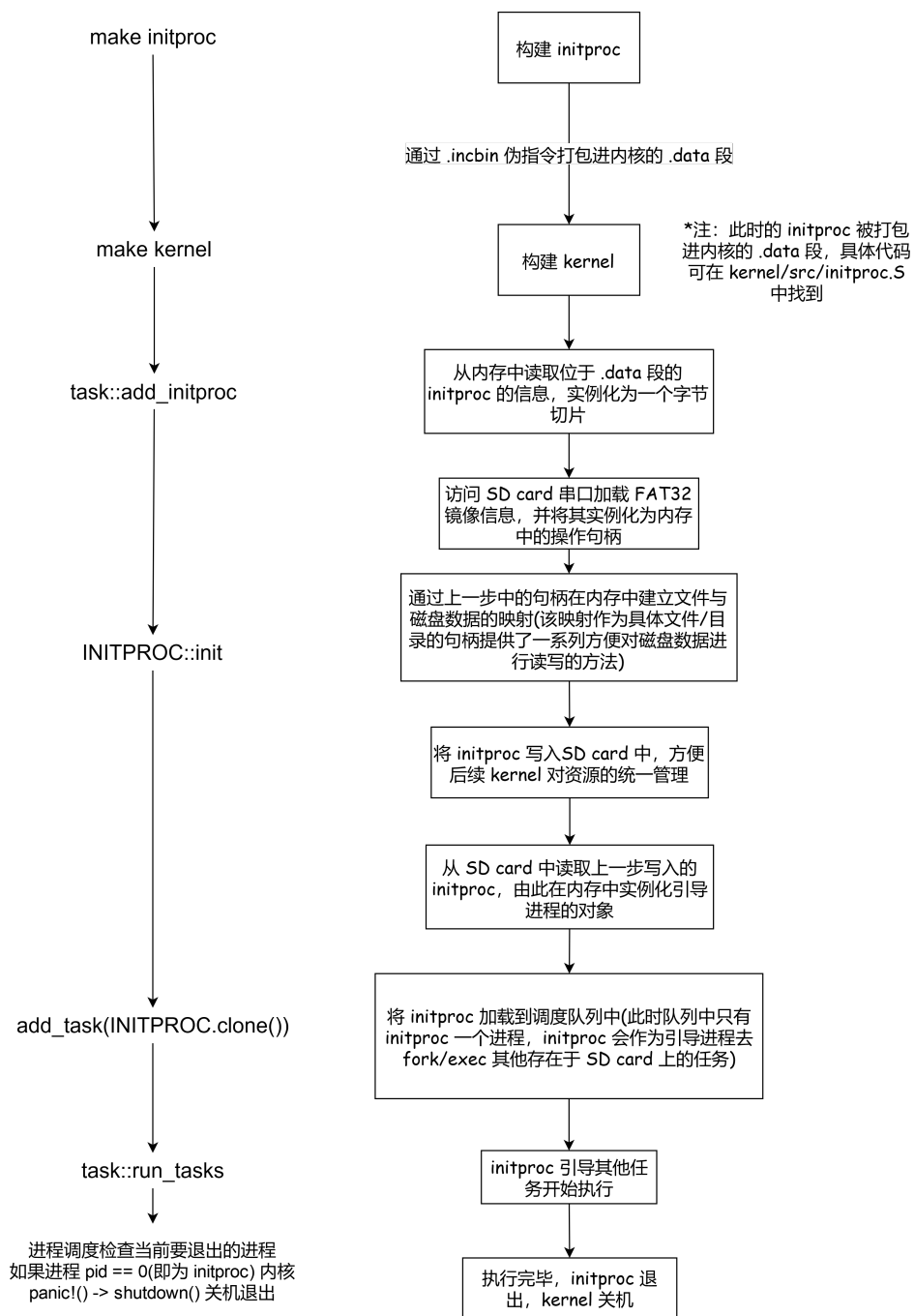


Figure 2: 引导进程 initproc 的创建和加载

2.2 TaskControlBlock(PCB)

任务控制块是进程在内存中的实例，记录了进程当前的所有信息：

```
// kernel/src/task/task.rs

pub struct TaskControlBlock {
    pub pid: PidHandle,
    pub tgid: usize,
    pub kernel_stack: KernelStack,
    inner: Mutex<TaskControlBlockInner>,
}

pub struct TaskControlBlockInner {
    /// 应用地址空间中的 Trap 上下文所在的物理页帧的物理页号
    pub trap_cx_ppn: PhysPageNum,
    pub task_cx: TaskContext,
    pub task_status: TaskStatus,
    /// 指向当前进程的父进程（如果存在的话）
    pub parent: Option<Weak<TaskControlBlock>>,
    /// 当前进程的所有子进程的任务控制块向量
    pub children: Vec<Arc<TaskControlBlock>>,
    pub exit_code: i32,
    pub memory_set: MemorySet,
    pub mmap_manager: MmapManager,
    /// 文件描述符表
    pub fd_table: Vec<Option<Arc<dyn File>>>,
    pub signals: SignalFlags,
    pub current_path: AbsolutePath,
}
```

下面我们将按照以下顺序介绍我们 `TaskControlBlock`（下面称作 `TCB`）的相关实现及相关**优化策略**：

- 进程描述符（进程描述符）
- 进程的用户/内核地址空间布局
- 进程的生成与调度
- 信号与进程的退出
- 进程的资源回收
- 进程的调度
 - 对于 CPU 和 RISC-V 中 `hart`（内核线程）的抽象
 - 多核调度策略

2.3 进程描述符 (PidHandle)

在单核环境下，`pid` 的分配不涉及并发问题，所以我们通过一个简单的栈式 `pid` 即可满足 `TCB` 的相关需求：

```
// kernel/src/task/pid.rs
```

```

/// 栈式进程标识符分配器
struct PidAllocator {
    /// 当前可用的最小PID
    current: usize,
    /// 已回收的 PID
    recycled: Vec<usize>,
}
/// 进程标识符
pub struct PidHandle(pub usize);
impl Drop for PidHandle {
    fn drop(&mut self) {
        PID_ALLOCATOR.lock().dealloc(self.0);
    }
}

```

这里我们利用 Rust 的生命周期，将超出生命周期的 `PidHandle` 交由全局 `pid` 分配器 `PID_ALLOCATOR` 回收，被回收的 `pid` 会被加入到 `PidAllocator` 的 `recycled` 字段中，在下一次分配的时候首先检查是否有已经回收过的 `pid`，如果没有就返回当前的 `current` 字段值，并完成 `current` 的递增。

```

// kernel/src/task/pid.rs

/// 分配一个进程标识符
pub fn alloc(&mut self) -> PidHandle {
    if let Some(pid) = self.recycled.pop() {
        PidHandle(pid)
    } else {
        self.current += 1;
        PidHandle(self.current - 1)
    }
}

```

2.3.1 多核 PID 池优化

而在多核环境中，存在多个 CPU 对全局 `PID_ALLOCATOR` 的并发访问情况，而如果我们只简单的采取竞态加锁的策略来应对，会大大降低多核的效率，无法发挥出多核的性能。所以我们提出了 `pid` 池这个概念：

对于每一个 CPU 抽象，其本身自主维护一个 `pid` 池，在分配 `pid` 时，会先检查当前 CPU `pid` 池 (`pid_pool`) 的情况，若未完全分配，就从本地 (这里的本地指的是当前 CPU) `pid` 池中拿取一个 `pid` 直到 CPU 本地 `pid_pool` 耗尽，才会去全局 `PID_ALLOCATOR` 那里领取一个池大小数量的 `pid`。

注：当前实现比较简陋，比如不能动态调整 CPU `pid_pool` 的大小，可能出现 CPU `pid` 饿死的情况 (一个 CPU 占据很多 `pid`，导致其他 CPU 没有其他 `pid` 可用)，初步设想是利用大小核的特点，对其余大核进行遥测监控，并有针对性的进行调整调度。

2.4 进程的生成与调度

2.4.1 子进程的创建和写时复制 (COW)

除了引导进程 `initproc` 外，其他进程都是通过 `fork/exec` 系统调用产生的。

一次具体的系统调用流程如下：

1. 用户提前将 `syscall` 所需参数写入到指定寄存器。

2. 通过汇编执行 `ecall` 调用 `trap` 到内核态，进入内核态后，PC 寄存器会跳转到提前设置好的 `stvec` 寄存器所指向的函数 `user_trap_handler` 处，由此开始执行 `trap` 的分发，进入 `sys_do_fork` 函数。
3. 在 `sys_do_fork` 函数中，我们会为子进程创建虚拟地址空间，分配 `pid`，将其加入到全局等待队列中。但对于子进程的虚拟地址空间 `MemorySet`，我们并不会立刻对齐分配物理页帧，而是将其指向父进程对应的物理页帧，同时将页表项 `PTEFlags` 中的可写标志位（若存在）置零。这样，当子进程访问到对应的物理页帧时，会引发又一次 `trap`，而在这次 `trap` 的处理中，我们才对子进程分配实际的物理页帧，同时恢复相应页表项的标志位。由此实现地址空间的写时复制（COW）。

2.4.2 单/多核调度策略

我们在全局创建了一个 `TaskManager` 对象用来管理闲置的进程实例，进程的实际运行委托给了 `Processor` 对象：

```
// kernel/src/task/manager.rs

/// 负责管理待运行的进程对象
pub struct TaskManager {
    ready_queue: VecDeque<Arc<TaskControlBlock>>,
}

/// 每个核上的处理器，负责运行一个进程
pub struct Processor {
    /// 当前处理器上正在执行的任务
    current: Option<Arc<TaskControlBlock>>,
    /// 当前处理器上的 idle 控制流的任务上下文
    idle_task_cx: TaskContext,
}
```

在 `kernel/src/task/processor/schedule.rs` 中，我们针对单/多核采取了不同的进程调度策略：

```
// kernel/src/task/processor/schedule.rs

#[cfg(not(feature = "multi_harts"))]
pub fn run_tasks() {
    use crate::task::{manager::fetch_task, processor::acquire_processor, task::TaskStatus};

    loop {
        let mut processor = acquire_processor();
        if let Some(task) = fetch_task() {
            let idle_task_cx_ptr = processor.idle_task_cx_ptr();
            // access coming task TCB exclusively
            let mut task_inner = task.lock();
            let next_task_cx_ptr = &task_inner.task_cx as *const TaskContext;
            task_inner.task_status = TaskStatus::Running;
            drop(task_inner);
            // release coming task TCB manually
            *processor.current_mut() = Some(task);
            // release processor manually
            drop(processor);
            unsafe { __switch(idle_task_cx_ptr, next_task_cx_ptr) }
        }
    }
}
```

```

}
#[cfg(feature = "multi_harts")]
pub fn run_tasks() -> ! {
    use super::acquire_processor;
    use crate::task::{manager::fetch_task, task::TaskStatus};
    loop {
        if let Some(task) = fetch_task() {
            info!("task {} fetched by hart {}", task.pid(), hartid!());
            let mut processor = acquire_processor();
            let idle_task_cx_ptr = processor.idle_task_cx_ptr();
            let mut task_inner = task.lock();
            let next_task_cx_ptr = &task_inner.task_cx as *const TaskContext;
            task_inner.task_status = TaskStatus::Running;
            drop(task_inner);
            *processor.current_mut() = Some(task);
            drop(processor);
            unsafe { __switch(idle_task_cx_ptr, next_task_cx_ptr) }
        }
    }
}

```

当前我们采用**分时调度**策略，当一个进程运行到一定时间或收到相应信号时，会让出 CPU，单核情况中，全局唯一的 `Processor` 会从 `TaskManager` 那里 `fetch` 一个任务，并切换当前任务上下文，开始运行。而在多核中，我们每次是先尝试获取一个任务，再去获取当前 CPU 对应的 `Processor` 来运行该任务。

对于多核上 `Processor` 的获取逻辑如下：

```

// kernel/src/task/processor/processor.rs

#[cfg(feature = "multi_harts")]
pub static mut PROCESSORS: [Cpu; 2] = [Cpu::new(), Cpu::new()];
pub fn acquire_processor<'a>() -> RefMut<'a, Processor> {
    unsafe { PROCESSORS[hartid!()].get_mut() }
}

```

需要点明的是，我们使用了 `RustSBI` 来方便我们的开发，而根据 `RISCV SBI` 的规范可知，每个 CPU(hart) 所对应的 id 会被提前设置到 `a0` 寄存器，所以在 `kernel/src/entry.S` 中，我们先获取了当前的 CPU id，并将其放在了 `tp` 寄存器中，之后我们就可以很方便地通过 `hartid!` 宏来获取当前 CPU 的 id 了：

```

# kernel/src/entry.S

.section .text.entry
.globl _entry
_entry:
    mv tp, a0 # mhartid 由 RustSBI 提前设置到 a0 寄存器中
    la sp, stack0
    li a0, 0x4000
    addi a1, tp, 1
    mul a0, a0, a1
    add sp, sp, a0

    call meow

```

```
// kernel/src/macros/hsm.rs

/// 获取当前正在运行的 CPU(hart) id
macro_rules! hartid {
    () => { {
        let hartid: usize;
        unsafe {core::arch::asm!("mv {}, tp", out(reg) hartid)}
        hartid
    }};
}

```

2.5 信号与进程的退出

我们的信号和 PCB 是一体的，其本质是一个 `u32` 类型的字段，我们使用了 Rust 第三方库 `bit_flags` 来方便我们对具体的二进制位进行操作。

```
// kernel/src/task/signals/signal_flags.rs

bitflags! {
    #[derive(PartialEq, Eq, Debug)]
    pub struct SignalFlags: u32 {
        const SIGINT    = 1 << 2;
        const SIGILL     = 1 << 4;
        const SIGABRT    = 1 << 6;
        const SIGFPE     = 1 << 8;
        const SIGKILL    = 1 << 9;
        const SIGUSR1    = 1 << 10;
        const SIGSEGV    = 1 << 11;
    }
}

```

BTD 信号产生及处理的流程大致如下图所示:

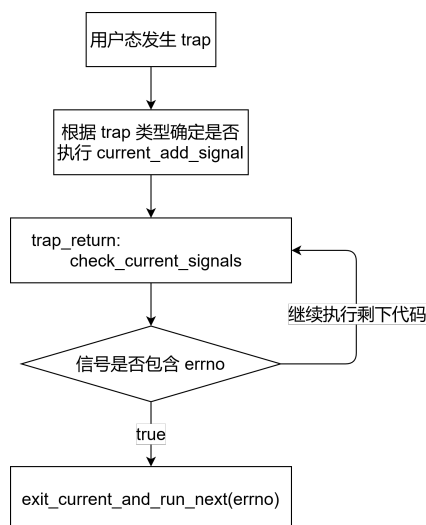


Figure 3: 信号处理流程图

2.6 进程的资源回收

2.6.1 进程结束的时机

当前进程一共有两种结束方式:

1. 进程运行完代码段, 非法访问到 `.rodata` 引发 trap, 在 trap 处理中回收进程对象
2. 进程意外结束, trap 到内核处理, 回收进程对象

2.6.2 通过 Rust 生命周期实现资源回收

进程在其结束后的资源回收是基于 Rust 的生命周期来实现的, Rust 无 GC 的实现是因为编译器会在对象生命周期分析后对超出生命周期的对象自动插入析构代码, 回收其所占用的资源。

这里我们对于进程 (`TaskControlBlock`) 资源的回收, 是通过 `Arc` 原子引用计数来实现的, `Arc` 实际上是一个胖指针, 其将实际的 `TaskControlBlock` 对象保存到内核堆上, 而当自身生命周期结束的同时, 会通过 Rust 编译器在编译期插入的析构代码减少 `TaskControlBlock` 的引用计数, 而当对 `TaskControlBlock` 的引用计数归零时, 其所占用的地址空间, `pid` 等资源会随着其自身的析构而被回收。

具体资源的回收逻辑可以通过 Rust 类型系统中的 `Drop` trait 来实现, 相当于添加了一个钩子 (Hook) 函数。

以 `pid` 为例, 我们通过 `PidHandle` 来实现 RAII:

```
// kernel/src/task/pid.rs

impl Drop for PidHandle {
    fn drop(&mut self) {
        PID_ALLOCATOR.lock().dealloc(self.0);
    }
}
```

3 内存管理

内存管理的主要任务是管理地址空间和页表结构。主要是为了统一内存的分配释放和访问以及保障内存安全。为了增加性能表现我们增加了对虚拟页的懒分配。整体的内存模块依赖于 riscv 的 SV39 分页系统设计, 大多数的模块都是基于这一模块建立的。

3.1 分页系统

与分段式内存管理相比, 我们采用了分页式内存管理方式, 既与 riscv 的 SV39 分页模式实现了统一又减少了内存碎片。在开启虚拟地址后所有的地址地址查询都会经过分页系统, 遵循 SV39 分页模式进行查询, 当 CPU 作出一次访存指令, 页表机构会自动从内存中读取相应的页表信息自动完成地址转换。

3.1.1 页表的硬件机制

要使用 riscv 处理器的 SV39 分页模式, 要在 S 特权级下启用虚拟地址需要修改 `satp` 寄存器, 修改后 S 特权级和 U 特权级的访存操作都会经过 MMU 转换。SV39 页表的实际结构类似一颗字典树。一次地址转换首先需要提取出虚拟地址中的三级页表的索引 `PPN[2], PPN[1], PPN[0]`, 以及页内偏移 `offset`。然后从 `satp` 中的根页表位置开始依次查找下一级页表的 `PPN`, 最后与 `offset` 拼接即可获得对应的物理地址。

⁴MIT 6.828 课程

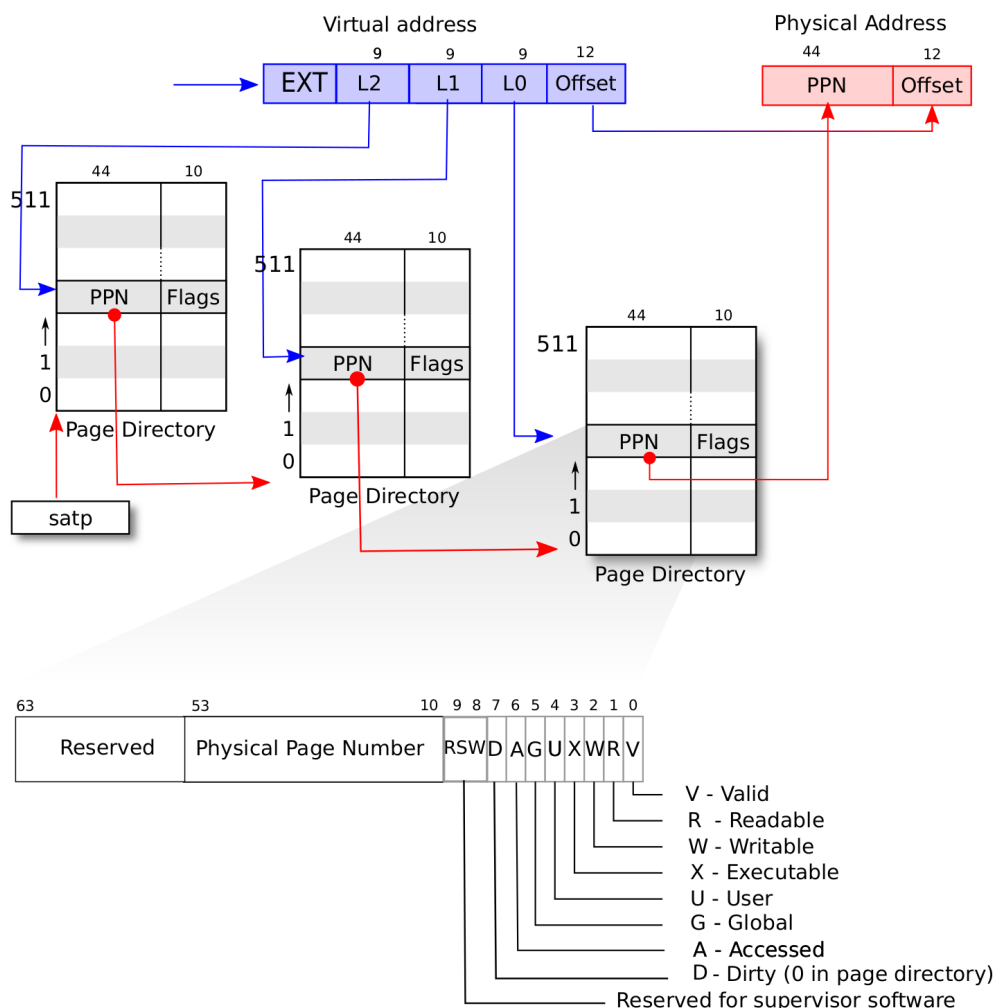


Figure 4: SV39 地址转换过程⁴

SV39 中的每级页表由 512 个页表项组成, 页表项的 [63:54]10 位是保留位不使用, [53:10]44 位是物理页号, [9:8]2 位是供操作系统使用的保留位, [7:0]8 位是标志位, 代表的意义如下:

- D(Dirty): 代表自上一次清零后页表项对应的页有没有被修改过, 该项由处理器自动设置。
- A(Accessed): 代表自上一次清零后页表项对应的页有没有被访问过, 该项由处理器自动设置。
- G(Global): 代表页表项对应的页是一个全局页, 可以在所有地址空间内访问。
- U(User): 代表页表项对应的页是可以在 U 特权级访问。
- X(eXecute), W(Writer), R(Read): 分别代表页表项对应的页是否可执行, 可写, 可读, 当该页是下一级页表时该 3 位都为 0。
- V(Valid): 代表该页表项是否有效, 只有当 V 位为 1 时页表项才合法。

内核中的页表项结构与 SV39 中实际储存的结构相同。

3.1.2 管理多级页表

为了在内核中使用多级页表, 我们设计了与多级页表匹配的数据结构。包括 VirtAddr, VirtPageNum, PhysAddr, PhysPageNum, PageTable, PageTableEntry. 对于 PageTable, 我们通过模拟 SV39 地址转换的过程实现了从虚拟地

址到物理地址的转换。同时还实现了页表和页表项的写入。

3.2 地址空间

地址空间是整个内存管理模块的主体，主要完成了对进程以及内核内存的统一管理。设置地址空间有利于实现用户程序的隔离，可以防止用户程序对系统内核或其他用户程序造成破坏，同时还可以为各个内存位置设置正确的权限。不仅如此，引入地址空间还能分别为每个用户程序创建独立的虚拟地址空间，这使得程序可以更便捷的分配内存。

3.2.1 地址空间设计

每个地址空间都是一个 `MemorySet` 结构体，使用不同的 `PageTable` 来使得彼此的虚拟地址不会发生冲突。`MemorySet` 有两个重要的方法，用于插入逻辑段的 `insert` 方法以及以 COW 的方式复制地址空间的 `from_copy_on_write` 方法。

```
// kernel/src/mm/memory_set/memory_set.rs

pub struct MemorySet {
    pub page_table: PageTable,
    vm_areas: Vec<VmArea>,
    mmap_areas: Vec<VmArea>,
    heap_areas: VmArea,
    pub brk_start: usize,
    pub brk: usize,
}
```

`VmArea` 代表储存在 `MemorySet` 内的已分配内存段。内部主要维护了每个逻辑段的起止位置和权限信息。同时，由于 `mmap` 使用的也是 `VmArea`，`VmArea` 也包装了文件信息以及文件的写回操作。此外 `VmArea` 还保存了所有属于该 `VmArea` 的页的 `FrameTracker`，这使得单个 `VmArea` 被释放或整个 `MemorySet` 被释放时可以自动释放相应的页。

```
// kernel/src/mm/memory_set/vm_area.rs

pub struct VmArea {
    pub vpn_range: VPNRange,
    pub map_type: MapType,
    pub permission: MapPermission,
    pub file: Option<Arc<dyn File>>, // 被映射的文件
    pub file_offset: usize,          // 被映射的文件在文件中的偏移量
    pub frame_map: BTreeMap<VirtPageNum, FrameTracker>, // vpn -> frame_tracker
}
```

3.2.2 内核地址空间与用户程序地址空间

我们将实际的物理地址分为大致 3 个部分：`MMIO`、内核堆、空闲物理页帧区。

内核与用户进程的虚拟地址空间参考 `linux/ELF` 的布局，但与 `linux` 不同的是，内核地址空间 and 用户态进程的地址空间完全享有两个大致上相同，大小相等的虚拟地址空间。

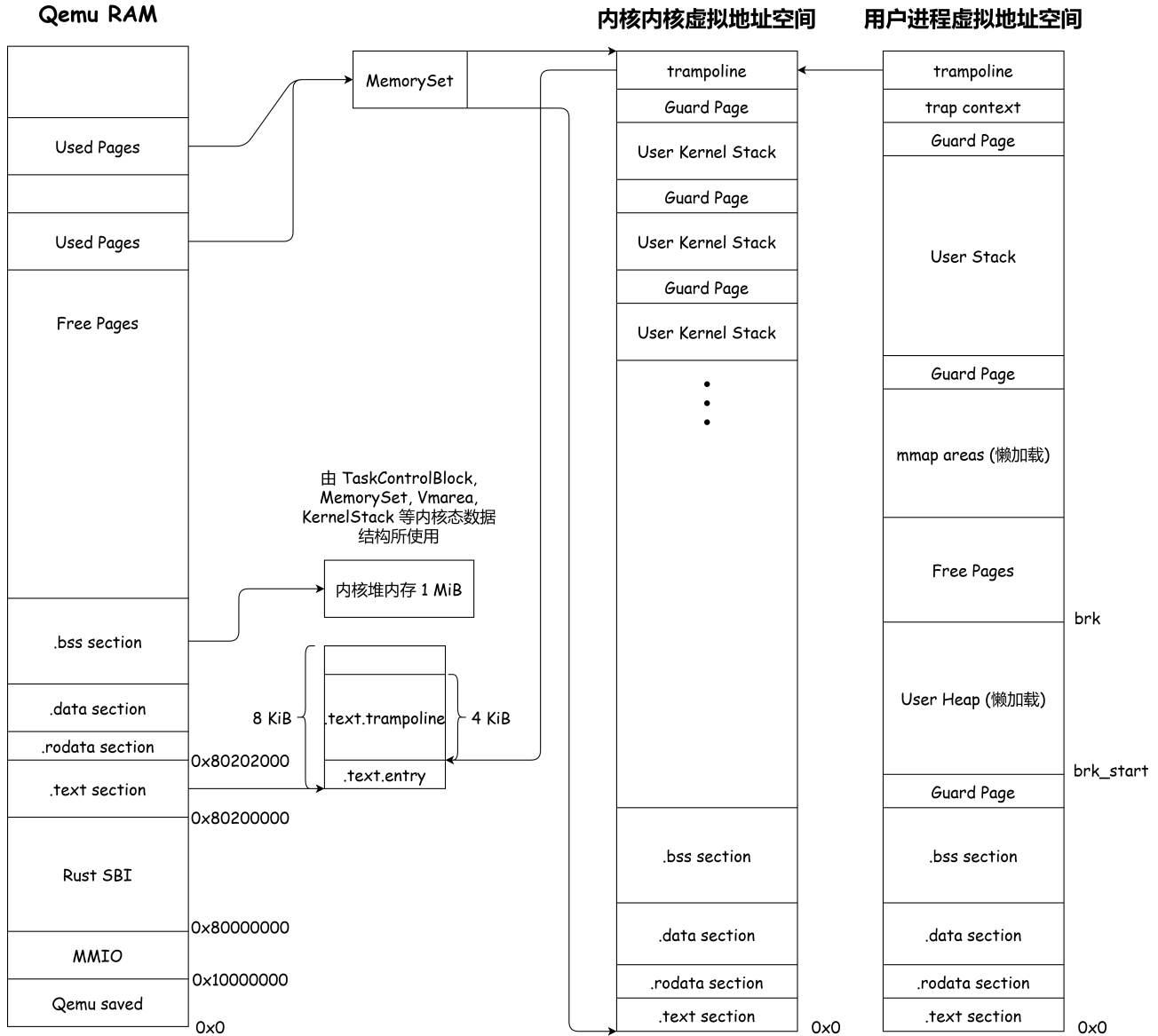


Figure 5: 地址空间布局

对于一个具体的用户进程来说，其所占有的**虚拟**地址空间可分为两部分，一部分位于用户态，另一部分为进程的内核栈，该内核栈可用于保存用户进程陷入 `trap` 时的进程上下文，和用户进程在内核态运行时执行函数调用的内存开销。

3.3 页帧管理器

页帧管理器管理的页帧区域为内核内存空间的结尾到物理内存的结尾，分配方式采用了简单的栈式分配。栈式分配可以在常数时间内完成一个页的分配和回收。页帧分配与回收是十分频繁的操作，目前只存在一个全局页帧分配器，可以为每个核心创建额外的页帧分配器以减小锁的竞争消耗。页帧的管理借鉴了 RAII 的思想，当分配一个页帧时创建一个 `FrameTracker`，在 `FrameTracker` 进行 `drop` 时回收一个页帧。同时页帧管理器还维护了一个引用计数器，可以在 `copy on write` 的情况下直接引用一个已有的页。

3.4 地址转换

每个进程都有独立的虚拟地址空间，同时内核也有自己的虚拟地址空间，彼此不能直接访问。想要在内核内访问进程的虚拟地址需要进行地址转换。目前实现了对用户空间的缓冲区、字符串和其它类型数据的地址转换。其中用户缓冲区的使用最为频繁，我们将其封装成了 UserBuffer 结构体，并为其实现了 read 和 write 方法。

3.5 Mmap 管理器

mmap (Memory Mapped Files) 是一种内存映射文件的机制，可以将一个文件映射到进程的地址空间中，使得进程可以像访问内存一样访问文件，而不需要进行繁琐的文件 I/O 操作。

在使用 mmap 时，操作系统会将文件的一部分或全部映射到进程的虚拟内存空间中，这个虚拟内存空间被称为映射区域。当进程需要读写文件时，可以直接读写映射区域中的数据，而不需要像普通的文件 I/O 一样进行读写操作。操作系统会自动将进程对映射区域的读写操作映射到文件中相应的位置，同时也会自动处理文件和内存之间的同步问题。

使用 mmap 可以提高文件 I/O 的效率，特别是对于大文件的读写操作，能够有效地减少系统调用的次数和数据的复制操作。同时，mmap 还可以用于共享内存，允许不同的进程可以共享同一块内存。

BTD OS 使用 MmapManger 来管理 mmap 空间，该管理器的结构如下：

```
// kernel/src/mm/vma.rs

/// mmap 管理器
pub struct MmapManager {
    pub mmap_start: VirtAddr,    // mmap区域的起始地址
    pub mmap_top: VirtAddr,      // mmap区域的顶部地址，标记下一个可用的地址
    pub mmap_set: Vec<MmapInfo>, // 统计各个 MmapArea 的信息
}
```

MmapManager 提供了 push 和 remove 方法来管理地址空间中的 mmap 区域。其中 MmapInfo 用于记录 mmap 空间的信息，结构如下：

```
// kernel/src/mm/vma.rs

/// MmapArea 空间信息，也是 sys_mmap 参数信息的结构体
pub struct MmapInfo {
    pub oaddr: VirtAddr,        // mmap 空间起始虚拟地址
    pub valid: usize,
    pub length: usize,
    pub prot: usize,            // mmap 空间权限
    pub flags: usize,           // 映射方式
    pub fd: isize,
    pub offset: usize,          // 映射文件偏移地址
}
```

mmap 空间权限与映射方式的定义如下：

```
// kernel/src/mm/vma.rs

bitflags! {
    /// mmap 空间权限
    pub struct MmapProts: usize {
        const PROT_NONE = 0; // 不可访问，用于实现防范攻击的 guard page 等
    }
}
```

```

    const PROT_READ = 1;
    const PROT_WRITE = 1 << 1;
    const PROT_EXEC  = 1 << 2;
}
}
bitflags! {
    pub struct MmapFlags: usize {
        const MAP_FILE = 0;
        const MAP_SHARED= 0x01;
        const MAP_PRIVATE = 0x02;
        const MAP_FIXED = 0x10;
        const MAP_ANONYMOUS = 0x20;
    }
}

```

mmap flags 的含义如下：

- MAP_FILE: 文件映射，使用文件内容初始化内存
- MAP_SHARED: 共享映射，修改对所有进程可见，多进程读写同一个文件需要调用者提供互斥机制
- MAP_PRIVATE: 私有映射，进程 A 的修改对进程 B 不可见的，利用 COW 机制，修改只会存在于内存中，不会同步到外部的磁盘文件上
- MAP_FIXED: 将 mmap 空间放在 addr 指定的内存地址上，若与现有映射页面重叠，则丢弃重叠部分。如果指定的地址不能使用，mmap 将失败。
- MAP_ANONYMOUS: 匿名映射，初始化全为 0 的内存空间

MmapInfo 提供了 lazy_map_page 与 map_file 方法，借助 translated_bytes_buffer 与 check_lazy 方法来实现懒分配与懒映射，具体流程查看图 6。

3.6 懒加载与懒分配

在某些情况下，内核分配的空间在很长一段时间内不会被访问，如果每次都实际分配物理空间将造成极大地性能浪费。为此需要实现懒分配方法：只有当真正需要分配物理空间时才实际分配物理空间。一般的懒分配策略需要配合缺页异常实现。

目前使用了懒分配策略的情况有：mmap 和 fork。mmap 的作用是将文件映射到内存。由于在很多情况下并不需要对文件全部内容进行读写，可以直接构建 VmArea 放入进程内，而不需要实际分配空间读入文件内容。在下次访问时由于没有进行空间分配，会发生缺页异常。此时可以检查进程 VmArea，判断此次缺页是否是一次懒分配，接着可以实际分配内存并从文件中读入真正内容。

fork 会将父进程的地址空间完全复制，这会造成巨大的性能消耗。而且在 fork 的实际使用过程中，有许多内存空间是可以父子进程共同使用的或子进程永远不会使用的。此时也可以使用懒分配策略，也叫 copy on write。在进行 fork 时仅复制父进程的地址空间，不为子进程的逻辑段分配物理内存，而是创建子进程虚拟地址到父进程物理地址的映射，复用父进程的物理页。同时擦除父进程物理页的 Write 许可位并设置 COW 标志位。这样在父进程或子进程对公共页进行写操作时就会触发页错误，进而未子进程或父进程分配新的页，从而实现懒分配。

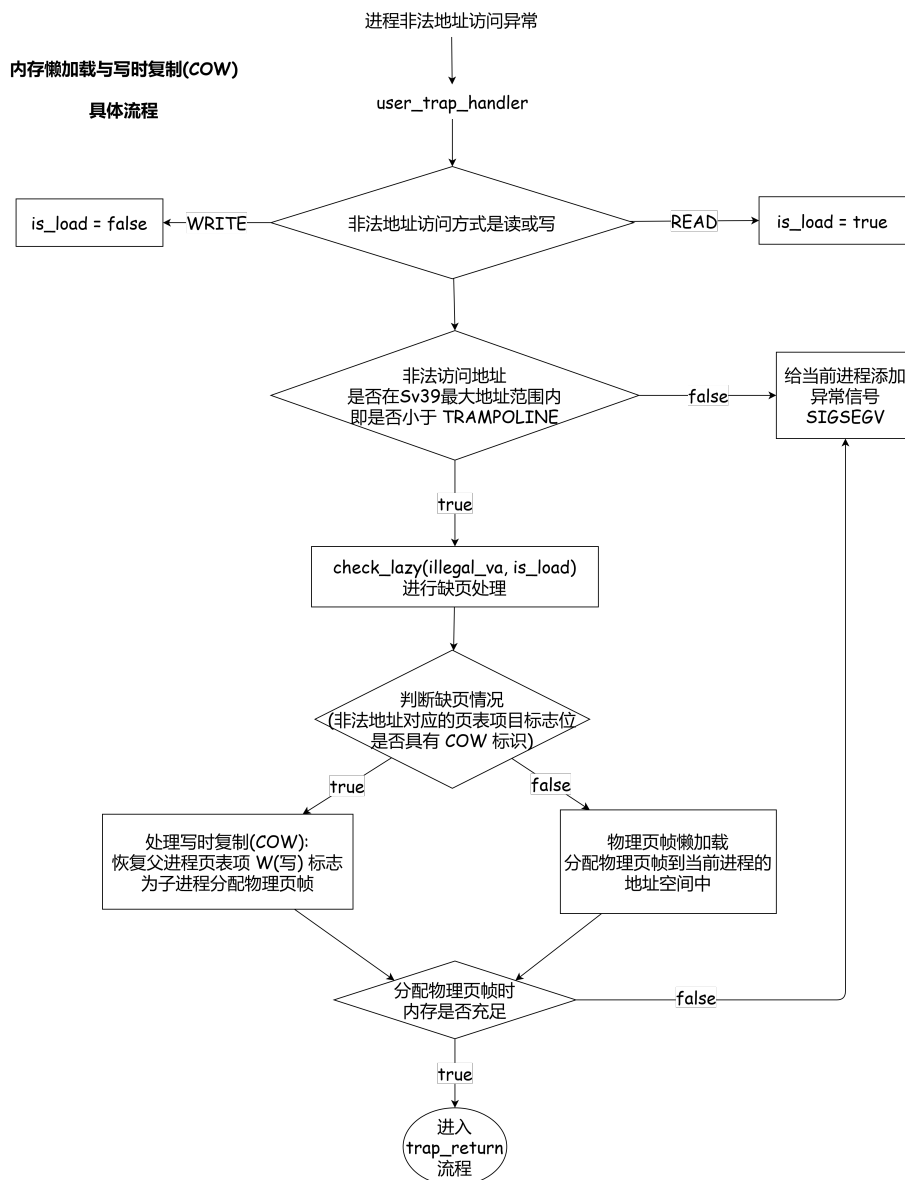


Figure 6: 懒加载与懒分配流程

4 文件系统

常规文件和目录都是实际保存在持久存储设备中的。持久存储设备仅支持以扇区（或块）为单位的随机读写，文件系统负责将逻辑上的目录树结构映射到持久存储设备上，决定设备上的每个扇区应存储哪些内容。反过来，文件系统也可以从持久存储设备还原出逻辑上的目录树结构。

对于文件系统模块，BTD OS 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。

4.1 FAT32 库

FAT32 库根据 Windows 官方标准实现，能够解析出由 Linux 创建的 FAT32 文件系统镜像，但目前 BTD OS 实现的 FAT32 是一个简化版的 FAT32，体现如下：

- FAT32 规定的 BPB, FSInfo, Entry 等字段仅维护目前需要使用的信息，部分信息如：文件访问或修改的任何时间戳等不维护。

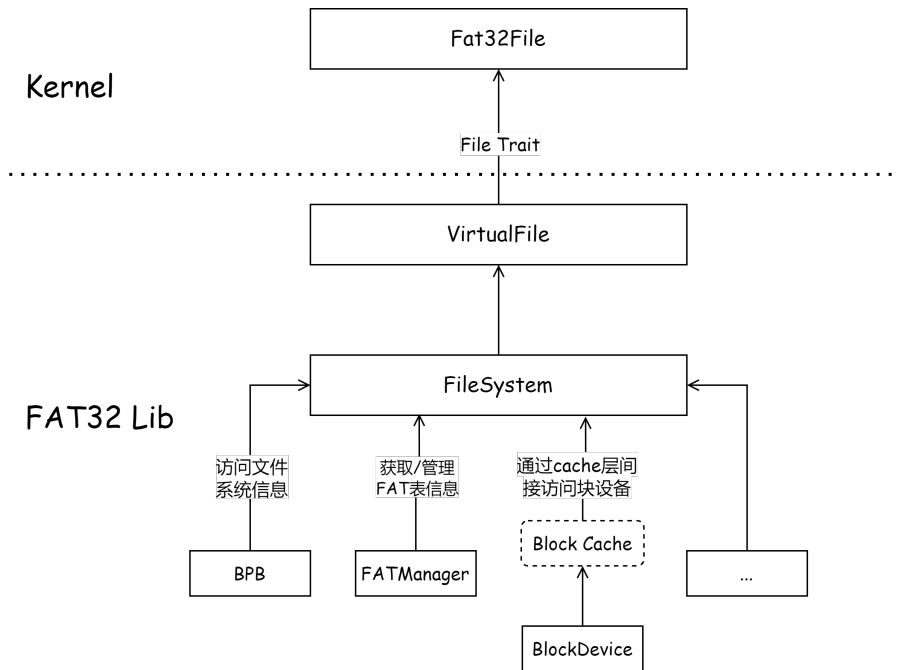


Figure 7: FAT32 架构

- 只使用第一个 FAT 表，不维护其他的 FAT 表。
- 对于文件名的处理不完善：未提供修改文件名的方法、长文件名转短文件名的方法不够完善。

4.1.1 块设备接口层

声明了一个块设备的抽象接口 BlockDevice，声明如下：

```
// fat32/src/device.rs

pub trait BlockDevice: Send + Sync + Any {
    fn read_blocks(&self, buf: &mut [u8], offset: usize, _block_cnt: usize) -> Result<(), Error>;
    fn write_blocks(&self, buf: &[u8], offset: usize, _block_cnt: usize) -> Result<(), Error>;
}
```

该 Trait 定义了两个方法：

- read_blocks: 从偏移 offset 处将数据从块设备中读到内存缓冲区中。
- write_blocks: 将内存缓冲区的数据从偏移 offset 处写回到块设备中。

另外，规定 buf.len() 必须为块大小的倍数，offset 偏移必须为块大小的倍数，需要读取的块的数目 block_cnt = buf.len() / BLOCK_SIZE 缓冲区大小以块为单位。目前支持在 Qemu 模拟器环境下使用 VirtIOBlock 访问 VirtIO 块设备。

4.1.2 块缓存层

由于硬盘读写速度相较于内存是数量级上的差距，操作系统频繁读写速度缓慢的磁盘块会极大降低系统性能，常见的操作是将磁盘块的数据通过 read_blocks 读取到内存缓冲区中，然后在缓冲区中直接进行读写操作，以避免

频繁的磁盘读写。如果缓冲区的内容被修改，需要通过 `write_blocks` 将修改后的内容写回磁盘块。为了提高代码实现鲁棒性和性能，需要对缓冲区进行合理的管理，即尽可能减少实际的块读写次数。这可以通过合并块读写操作来实现，例如可以直接使用已有的缓冲区而不必再次读取，多次修改同一块的内容时只需在所有修改结束后一次性写回磁盘。

对于复杂的磁盘数据结构，合理规划块读取/写入时机很困难，因此可以通过统一管理缓冲区来解决同步性问题。在读写块时，先查看全局管理器中是否已缓存该块，如果已缓存，则所有操作均在同一缓冲区中进行，解决了同步性问题。全局管理器负责处理块实际读写的时机，上层子系统无需关心。全局管理器会尽可能合并更多块操作，并在必要时发起真正的块实际读写。为了提高扩展性和兼容性，我们利用 Rust 中数据和行为分离的设计，使用 `trait` 规定了缓存块的行为，由此作为接口提供了一定的可适配性。

```
// fat32/src/cache.rs

pub trait Cache {
    fn read<T, V>(&self, offset: usize, f: impl FnOnce(&T) -> V) -> V;
    fn modify<T, V>(&mut self, offset: usize, f: impl FnOnce(&mut T) -> V) -> V;
    fn sync(&mut self);
}
```

- `read` 和 `modify` 方法分别在 `Cache` 中获取一个类型为 `T` 的磁盘数据结构的不可变与可变引用，并让它执行传入的闭包 `f` 中所定义的操作。这两个方法返回值和闭包 `f` 的返回值相同。这样，`read` 和 `modify` 方法就为传入的闭包 `f` 提供了一个执行环境，让它能够绑定到一个缓冲区上执行。
- `sync` 方法用于将缓冲区中的数据写回磁盘。

块缓存 `BlockCache` 的定义如下：

```
// fat32/src/cache.rs

pub struct BlockCache {
    cache: Vec<u8>,
    block_id: usize,
    block_device: Arc<dyn BlockDevice>,
    modified: bool,
}
```

其中：

- `cache` 表示位于内存中的缓冲区，这里使用 `Vec<u8>` 表示，创建时 `Vec` 大小与块大小相同；
- `block_id` 保存当前块缓存在物理存储介质的实际块号；
- `block_device` 是一个底层块设备的引用，可通过它进行块读写；
- `modified` 记录这个块从磁盘载入内存缓存之后，它有没有被修改过，若当前块缓存的内容有更改需要写回磁盘，则值为 `True`。

`BlockCache` 的设计体现了 `RAII` 思想，它管理着一个缓冲区的生命周期。当 `BlockCache` 的生命周期结束之后缓冲区也会被从内存中回收，这个时候 `modified` 标记将会决定数据是否需要写回磁盘。

`BlockCache` 实现了 `Cache Trait`，在 `BlockCache` 被 `drop` 的时候，它会首先调用 `sync` 方法，如果自身确实被修改过的话才会将缓冲区的内容写回磁盘。事实上，`sync` 并不是只有在 `drop` 的时候才会被调用。由于目前的实现比较简单，`sync` 仅会在 `BlockCache` 被 `drop` 时才会被调用。

块缓存管理器 `BlockCacheManager` 定义如下：

```
// fat32/src/cache.rs

pub struct BlockCacheManager {
    lru: LruCache<usize, Arc<RwLock<BlockCache>>>,
}

lazy_static! {
    pub static ref BLOCK_CACHE_MANAGER: Mutex<BlockCacheManager> =
        Mutex::new(BlockCacheManager::new());
}

// fat32/src/lib.rs

pub const BLOCK_CACHE_LIMIT: usize = 64;
```

BlockCacheManager 是一个 LRU Cache, 管理的是块编号和块缓存的二元组。为了避免在块缓存上浪费过多内存, 我们需要限制内存中同时能驻留的磁盘块的数目。

块编号的类型为 `usize`, 块缓存的类型是一个 `Arc<RwLock<BlockCache>>`, 它提供共享引用和读写访问。块缓存需要同时在 BlockCacheManager 中保留一个引用, 并以引用的形式返回给请求者, 以便访问块缓存。这里的共享引用就能够实现这一目的。互斥访问提供内部可变性, 以便在单核上通过编译, 在多核环境下避免并发冲突。事实上, 一般情况下我们需要在更上层提供保护措施避免两个线程同时对一个块缓存进行读写, 因此这里只是比较谨慎的留下一层保险。

BlockCacheManager 提供了两个方法:

```
// fat32/src/cache.rs

pub fn get_block_cache(
    &mut self,
    block_id: usize,
    block_device: Arc<dyn BlockDevice>,
) -> Arc<RwLock<BlockCache>>;

pub fn sync_all(&mut self);
```

`get_block_cache` 方法尝试从块缓存管理器中获取一个编号为 `block_id` 的块的块缓存, 如果找不到, 会从磁盘读取到内存中, 还有可能会发生缓存替换:

4.1.3 文件系统描述层

存储布局 存储布局是体现 FAT32 规范的关键, 严格按照 FAT32 的标准进行数据结构的设计, 使用的参考资料为: Microsoft Extensible Firmware Initiative FAT32 File System Specification (Version 1.03, December 6, 2000)

一个 FAT32 文件系统卷 (Volume) 由 3 个基本区域组成, 它们按此顺序排列在 Volume 上:

- 保留扇区 (包括引导扇区)
- FAT 表区域 (文件分配表区)
- 数据 (文件与目录) 区域

磁盘布局如图:

其中:

1. 保留扇区包括引导扇区, 引导扇区包括 BPB 和 FSInfo

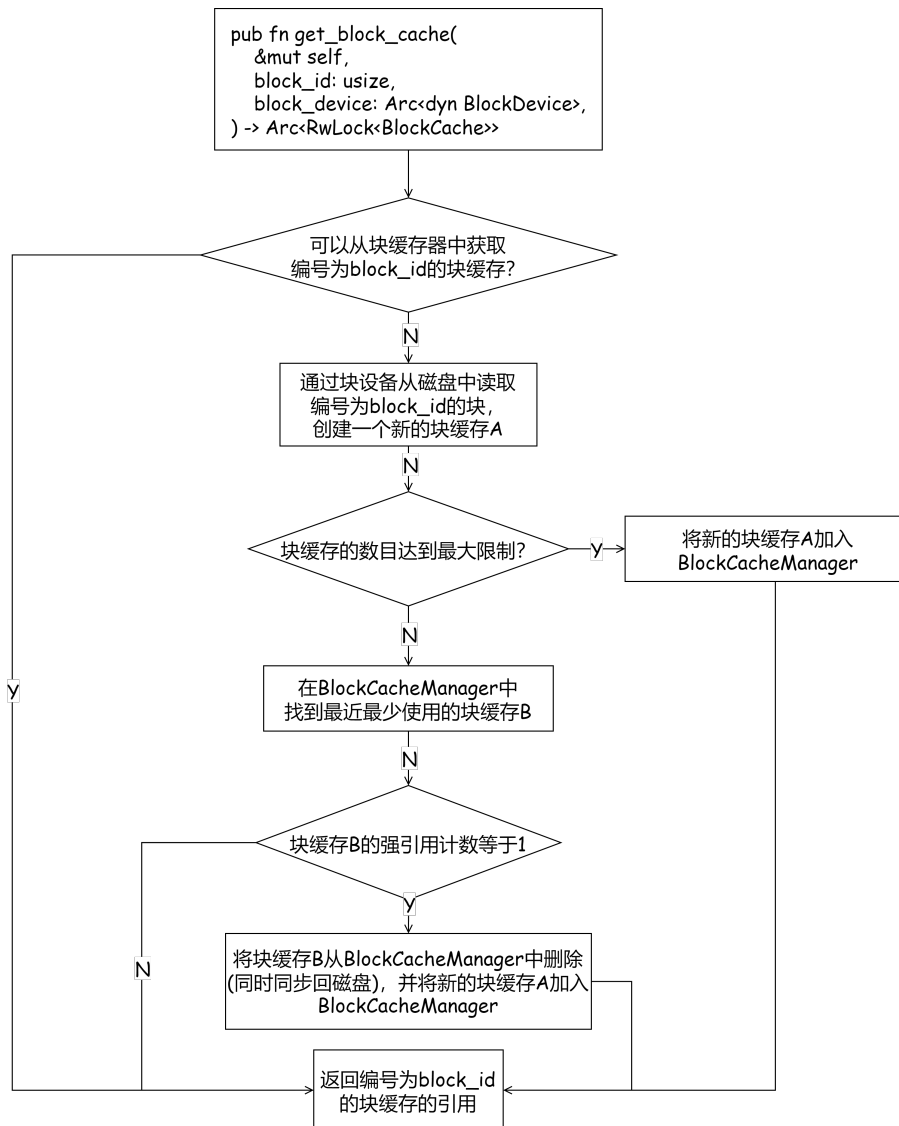


Figure 8: 获取缓存流程图

2. FAT1 起始地址 = 保留扇区数 * 扇区大小

3. 文件分配表区共保存了两个相同的文件分配表, 因为文件所占用的存储空间 (簇链) 及空闲空间的管理都是通过 FAT 实现的, 保存两个以便第一个损坏时, 还有第二个可用。

保留区域的第一个扇区是 BPB(BIOS Parameter Block), 其中较为重要的有:

- BPB_BytsPerSec: 每个扇区所包含的字节数量。
- BPB_SecPerClus: 每个簇所包含的扇区数量。
- BPB_RsvdSecCnt: 保留扇区的数目, 通过它能获得第一个 FAT 所在的扇区, 在 FAT32 中为 32。
- BPB_NumFATs: FAT 文件系统中 FAT 的数量, 通常为 2。

扩展 BPB 字段中较为重要的有:

- BPB_FATSz32: 一个 FAT 所占的扇区数。

BPB	FSInfo	Other Reserved Region	FAT1	FAT2	Root Directory	Other File and Directory Data
-----	--------	-----------------------	------	------	----------------	-------------------------------

Figure 9: FAT 磁盘布局图

- BPB_FSInfo: FsInfo 结构在保留区域中的扇区号。

保留区域中还有一个记录 FsInfo 的扇区，其中较为重要的字段有：

- FSI_Free_Count: FAT32 卷上最近已知的空闲簇计数。

FAT32 lib 实现的 BPB 与 FSInfo 结构均提供了一些较为有用的方法：

```
// fat32/src/bpb.rs

impl BIOSParameterBlock {
    // 返回一个簇在磁盘上的位置
    pub fn offset(&self, cluster: u32) -> usize {...};
    // 返回根目录位置
    pub fn first_data_sector(&self) -> usize {...};
}

impl FSInfo {
    pub fn free_cluster_cnt(&self) -> u32 {...};
    pub fn set_free_clusters(&mut self, free_clusters: u32) {...};
    pub fn next_free_cluster(&self) -> u32 {...};
    pub fn set_next_free_cluster(&mut self, start_cluster: u32) {...};
}
```

目录项 FAT32 的每个目录项大小固定为 32 字节，根据文件名的长度划分长短目录项。Fat32 是一种常用的文件系统，支持长文件名和短文件名两种目录项格式。其中，短目录项是 Fat32 文件系统的基本目录项，它由固定长度的 32 字节组成，用于存储文件的基本信息。下面是 Fat32 短目录项的结构：

```
// fat32/src/entry.rs

pub struct ShortDirEntry {
    name: [u8; 8],           // 文件名
    extension: [u8; 3],      // 文件扩展名
    attr: u8,                // 文件属性
    nt_res: u8,              // 保留字段
    crt_time_tenth: u8,      // 创建时间(十分之一秒计)
    crt_time: u16,           // 创建时间
    crt_date: u16,           // 创建日期
    lst_acc_date: u16,       // 最后访问日期
    fst_clus_hi: u16,        // 文件起始簇号高位
    wrt_time: u16,           // 最后修改时间
    wrt_date: u16,          // 最后修改日期
    fst_clus_lo: u16,        // 文件起始簇号低位
    file_size: u32,          // 文件大小
}
```

其中，文件名和文件扩展名的长度分别为 8 和 3 个字符，文件属性用于指示文件的类型和属性，包括只读、隐藏、系统文件、目录等。创建时间和创建日期记录了文件的创建时间，最后访问日期记录了文件的最后访问时间，最后修改时间和最后修改日期则记录了文件的最后修改时间。文件起始簇号用于指示文件的起始簇号，文件大小记录了文件的大小。

由于短目录项的文件名和扩展名长度限制，它无法支持长文件名。因此，在 Fat32 文件系统中，还引入了长目录项来支持长文件名。长目录项由一系列 32 字节的目录项构成，其中每个目录项都用于存储长文件名的一部分，最后一个目录项存储文件的基本信息。

```
// fat32/src/entry.rs

pub struct LongDirEntry {
    ord: u8,           // 序列号
    name1: [u16; 5],   // 文件名的第 1~5 个字符
    attr: u8,          // 属性标志
    ldir_type: u8,      // 长目录项类型
    chk_sum: u8,        // 校验和
    name2: [u16; 6],   // 文件名的第 6~11 个字符
    fst_clus_lo: u16,   // 长目录项起始簇的低位
    name3: [u16; 2],   // 文件名的第 12~13 个字符
}
```

长短文件名目录项的特点如下：

- 每个长目录项都有对应的短目录项，其短目录项的文件属性（偏移为 0xB）为 0x0F。
- 系统将长文件名以 13 个字符为单位进行切割，每一组占据一个目录项，所以一个文件可能需要多个目录项。
- 若需要多个目录项才能保存下文件名，那么第一个目录项的编号是 0x1，第二个是 0x2，以此类推。最后一个目录项的需要与 0x40 进行异或操作。FAT32 限制文件名的长度不超过 255 字节（小于 20 个目录项）。
- 多个长文件名的各个目录项按倒序排列在目录表中，以防与其他文件名混淆。最后的长目录项（即长文件名的第一部分）紧跟其对应的短文件名。
- 对于目录文件（文件属性为 0x10），短目录中的文件大小字段（偏移为 0x1C）的值为 0。
- 长文件的 checksum 字段（偏移为 0x0D）是用短文件名的 11 个字符通过一种运算方式来得到的。系统根据相应的算法来确定相应的长文件名和短文件名是否匹配。如果通过短文件名计算出来的校验和与长文件名中的 0xD 偏移处数据不相等，那么系统不会将它们进行配对。

由于内核目前对时间支持得不完善，因此 BTD OS 实现的 FAT32 Lib 中没有对文件的各种时间做处理。

文件分配表 文件分配表 (File Allocation Table, FAT) 是存储在存储介质上的表，它指示磁盘上所有数据簇的状态和位置。

BTD OS 使用 FATManager 管理 FAT，FATManager 数据结构如下：

```
// fat32/src/fat.rs

pub struct FATManager {
    device: Arc<dyn BlockDevice>,
    recycled_cluster: VecDeque<u32>,
}
```

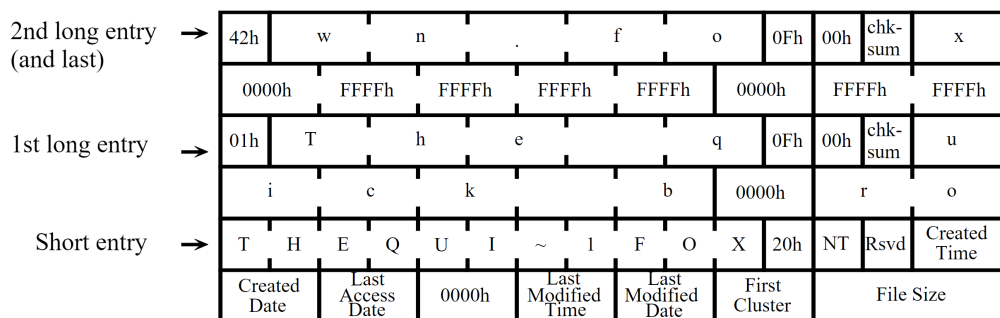


Figure 10: The quick brown.fox 磁盘布局

```
fat1_offset: usize,
}
```

提供给上层的重要方法：

```
// fat32/src/fat.rs

// 给出 FAT 表的下标(cluster_id_in_fat 数据区簇号)，返回这个下标 (fat 表的) 相对于磁盘的扇区
// 数 (block_id) 与扇区内偏移
pub fn cluster_id_pos(&self, index: u32) -> (usize, usize);
// 从 start_from 开始找到 FAT 表中的空闲的簇
pub fn blank_cluster(&mut self, start_from: u32) -> u32;
pub fn get_next_cluster(&self, cluster: u32) -> Option<u32>;
pub fn get_cluster_at(&self, start_cluster: u32, index: u32) -> Option<u32>;
```

此外，我们还设计了 ClusterChain 的数据结构，该结构目前仅用于遍历文件的簇链。未来可能考虑用于 FAT Lib 的优化

```
// fat32/src/fat.rs

pub struct ClusterChain {
    pub(crate) device: Arc<dyn BlockDevice>,
    pub(crate) fat1_offset: usize, // read_only
    pub(crate) start_cluster: u32,
    pub(crate) previous_cluster: Option<u32>,
    pub(crate) current_cluster: u32,
    pub(crate) next_cluster: Option<u32>,
}
```

磁盘块管理器 FileSystem 负责实现整体磁盘布局，将各段区域及上面的磁盘数据结构整合。通过 FileSystem，可以解析真实的 FAT32 文件系统（目前支持解析由 Linux 格式化生成的 FAT32 文件系统镜像），它知道每个布局区域所在的位置，磁盘块的分配和回收也需要经过它才能完成。FileSystem 结构如下：

```
// fat32/src/fs.rs

pub struct FileSystem {
    pub(crate) device: Arc<dyn BlockDevice>,
```

```

pub(crate) free_cluster_cnt: Arc<RwLock<usize>>,
pub(crate) bpb: BIOSParameterBlock, // read only
pub(crate) fat: Arc<RwLock<FATManager>>,
// 虚拟根目录项。根目录无目录项，引入以与其他文件一致
pub(crate) root_dir_entry: Arc<RwLock<ShortDirEntry>>,
}

```

提供的重要方法有：

```

// fat32/src/fs.rs

pub fn first_sector_of_cluster(&self, cluster: u32) -> usize;
// 获得一个簇在磁盘上的偏移
pub fn cluster_offset(&self, cluster: u32) -> usize;
// 从一个设备打开文件系统
pub fn open(device: Arc<dyn BlockDevice>) -> Arc<RwLock<Self>>;
pub fn alloc_cluster(&self, num: usize, start_cluster: u32) -> Option<u32>;
pub fn dealloc_cluster(&self, clusters: Vec<u32>);
// 获得根目录的短目录项
pub fn root_dir_entry(&self) -> Arc<RwLock<ShortDirEntry>>;

```

4.1.4 虚拟文件层

虚拟文件层服务于文件相关系统调用。它通过屏蔽文件系统内部细节，提供一个虚拟文件接口给操作系统内核使用。虚拟文件层使得文件系统的使用者无需关心磁盘布局的实现细节，而是能够直接看到逻辑上的目录树结构中的文件和目录。它主要实现了文件创建、清空、删除、查找、读取和写入等接口，直接提供给操作系统内核调用。

```

// fat32/src/vfs.rs

pub struct VirFile {
    pub(crate) name: String,
    pub(crate) sde_pos: DirEntryPos,
    pub(crate) lde_pos: Vec<DirEntryPos>,
    pub(crate) fs: Arc<RwLock<FileSystem>>,
    pub(crate) device: Arc<dyn BlockDevice>,
    pub(crate) cluster_chain: Arc<RwLock<ClusterChain>>,
    pub(crate) attr: VirFileType,
}

pub struct DirEntryPos {
    pub(crate) cluster: u32,
    pub(crate) offset_in_cluster: usize,
}

```

4.2 内核中的文件系统模块

4.2.1 文件抽象

为了使 BTD OS 有更好的可拓展性，BTD OS 的 FS 模块声明 File Trait 接口，只要实现了接口的对象都可视为 OS 可管理的文件。OS 不关心文件内容，只关心如何对文件按字节流进行读写的机制，有了文件这样的抽象后，

内核就可把能读写并持久存储的数据按文件来进行管理，并把文件分配给进程，让进程以统一抽象接口 File 来读写数据。File Trait 声明如下：

```
// kernel/src/fs/mod.rs

pub trait File: Send + Sync {
    fn readable(&self) -> bool;
    fn writable(&self) -> bool;
    fn available(&self) -> bool;
    fn read(&self, buf: UserBuffer) -> usize;
    fn write(&self, buf: UserBuffer) -> usize;
    fn read_to_vec(&self, offset: isize, len: usize) -> Vec<u8>;
    fn seek(&self, _pos: usize);
    fn name(&self) -> &str;
    fn fstat(&self, _kstat: &mut Kstat);
    fn set_time(&self, _timespec: &Timespec);
    fn dirent(&self, _dirent: &mut Dirent) -> isize;
    fn get_path(&self) -> &str;
    fn offset(&self) -> usize;
    fn set_offset(&self, _offset: usize);
    fn set_flags(&self, _flag: OpenFlags);
    fn set_cloexec(&self);
    fn read_kernel_space(&self) -> Vec<u8>;
    fn write_kernel_space(&self, _data: Vec<u8>) -> usize;
    fn file_size(&self) -> usize;
    fn r_ready(&self) -> bool;
    fn w_ready(&self) -> bool;
}
```

目前，在 BTDOs 中为 pipe, stdio, 以及 FAT32 提供的 VirFile 实现了 File Trait。具体可以查看

4.2.2 在内核中接入 FAT32

为了能够在内核中接入我们的 FAT32 库，首先需要块设备实例实现 FAT32 库声明的块设备接口。这里我们使用 virtio-drivers 库的 VirtIOBlock 作为块设备，为其实现 HallImpl, Send, Sync, BlockDevice Trait，具体实现位于 kernel/src/drivers/block/virtio_blk.rs。其中 HallImpl 接口的实现参考 virtio-drivers 提供的样例，具体实现位于 kernel/src/drivers/block/virtio_impl.rs。

接着利用创建好的块设备实例，在内核中使用 FAT32 库提供的 Open 接口，从块设备 BLOCK_DEVICE 上打开文件系统，并从文件系统中获取根目录文件。实际上 FAT32 并没有根目录文件的目录项，该根目录文件是 FAT32 库虚拟文件在内核中进一步封装的内核虚拟文件 FAT32File。

4.2.3 内核虚拟文件

从用户的角度来看，打开一个文件时可以使用不同的标志，这些标志会影响到文件的访问方式。此外，在连续调用 sys_read/write 函数读写文件时，文件读写的当前偏移量也很重要，因为它随着文件读写的进行而被不断更新。为了实现这些用户视角中的文件系统抽象特征，需要在内核中进行实现，与进程有很大的关系。然而，FAT32 提供的虚拟文件不需要涉及这些与进程紧密结合的属性。因此，我们需要将 FAT32 库提供的 VirFile 加上相关信息，并将其进一步封装为 OS 中的 FAT32File。

```
// kernel/src/fs/fat32/file.rs
```

```
pub struct Fat32File {
    readable: bool, // 该文件是否允许通过 sys_read 进行读
    writable: bool, // 该文件是否允许通过 sys_write 进行写
    pub inner: Mutex<Fat32FileInner>,
    path: AbsolutePath,
    name: String,
}

pub struct Fat32FileInner {
    offset: usize, // 偏移量
    pub inode: Arc<VirFile>,
    flags: OpenFlags,
    available: bool,
}
```

4.2.4 绝对路径模块

为了更方便地进行路径处理，BTD OS 对 Vec<String> 进行抽象，声明为 AbsolutePath，具体如下：

```
// kernel/src/fs/path.rs

pub struct Path {
    pub components: VecDeque<String>,
}
```

AbsolutePath 用于 TCB 的 current_path 字段以及 FAT32File 的 path 字段。基于 AbsolutePath，可以更方面地从 ROOT_INODE 搜寻文件，更方便地实现 FS 相关系统调用。提供了以下较为重要的方法：

```
// kernel/src/fs/path.rs

pub fn from_string(path: String) -> Option<Self>;
pub fn as_string(&self) -> String;
pub fn cd(&self, path: String) -> Option<Self>;
pub fn join_string(&self, path: String) -> Self;
```

4.2.5 其他模块与系统调用说明

(1) 获取文件状态与目录项信息 sys_fstat 和 sys_getdents64 是两个用于获取文件信息的系统调用。它们分别使用在 stat.rs 和 dirent.rs 中定义的结构体和方法来实现。这些结构体和方法使用 #[repr(C)] 宏进行了调整，以支持 C 语言程序的使用。此外，位于内存模块中的 UserBuffer 也为这两个系统调用提供了支持。它是对用户地址空间中一段缓冲区的抽象，通过提供的 write 方法，内核可以方便地在用户地址空间进行写入数据。

(2) 挂载由于目前系统中对挂载的需求不是很大，因此我们只设计了一个挂载表来记录挂载设备，它的定义如下：

```
// kernel/src/fs/mount.rs

pub struct MountTable {
    mnt_list: Vec<(String, String, String)>, // 挂载设备，挂载点，挂载文件系统类型
}
```

5 问题与解决

5.1 工具链默认不支持乘法指令

可以通过伪指令 `.attribute arch, "rv64gc"`⁵来解决。这个问题的解决其实并不直观,除了给出的文档还参考了别人一些类似的情况,不过有的相关链接已经失效了。不过这次给我们的启发是:遇到不好解决的问题,除了别人的经验,可以大胆向官方提问,开 issue。

5.2 VirtIOBlk 物理内存不连续时导致缓存数据丢失

在接入文件系统后的一次测试中,我们发现系统在不同优化等级或打印不同的调试信息时发生错误且错误不同,有时是 ELF 文件读取长度为 0,有时是用户程序发生 Instruction Fault,有时是 Cluster Chain 跳转到 0 号簇。虽然错误类型各不相同但都表现为与文件系统相关的错误。经过进一步的调查,我们发现这些错误都是在对 block cache 进行读写操作时丢失后段信息导致的。一般表现为 `get_block_cache` 返回的 `cache` 为空或后段为空。

我们最初怀疑这是一个与内存相关的错误,可能是栈或者某些内存空间被错误释放或者被重复使用导致数据被污染。为此我们仔细检查了每个内存段的读写权限,并且追踪了所有页的释放。最终我们认为这不是内核内存部分代码引起的错误。

接着我们对内核的各个组件进行了替换测试,在将文件系统替换后错误消失,我们将调查的重点放到了文件系统上。在修复了文件系统的若干 bug 后错误仍未消失。在一次和队友的杂谈中我们了解到打印调试信息使用的 `printf` 可能会影响 rust 的优化行为,进而会影响一个 `buffer` 在栈和堆间的移动。于是我们又将调查的重点放到堆和栈上。

在一次偶然的调试中,我们将页帧分配器由栈分配方式改为简单的连续分配方式,此时消除了错误。我们推断是使用了栈式页帧分配器导致程序的内核栈在物理空间内连续,从而给 VirtIOBlk 传入了不连续的 `buffer`。在查看了 VirtIOBlk 的 issue⁶后验证了我们的猜想,并在 rCore 的代码中找到了解决方法⁷。

5.3 FAT32 文件系统规范问题

没有根目录短目录项实体: 最初我们 FAT32 库为根目录记录存储了实体,但发现没法解析 Linux 格式化出的 FAT32 文件系统镜像,虽然看文档时对 FAT32 的根目录没有目录项有印象,但实际实现过程中为了方面记录了实体,最后查阅文档以及参考上届作品的 FAT32 文件系统实现时发现并解决了这个问题。

目录文件的短目录项的文件大小字段为 0: 最初我们 FAT32 库的读参考了部分 `eazy-fs` 的文件读,其中利用文件大小来判断边界范围等问题,虽然看文档时对目录文件的文件大小为 0 也有映像,但是由于在实现文件读时使用了文件大小作为判断依据,导致没法解析文件系统内容。最后也是通过查阅文档以及参考上届作品的 FAT32 文件系统实现时才发现并解决了这个问题。

5.4 多核启动顺序

虽然我们接触过多线程的并发问题,但没有考虑到多核情况下的并发。资源的初始化需要在单核环境初始化完成后进行,然后其他核才能依次运行。参考 `xv6-riscv`⁸ 解决。

⁵<https://github.com/riscv-non-isa/riscv-asm-manual/blob/master/riscv-asm.md#-attribute>

⁶<https://github.com/rcore-os/virtio-drivers/issues/8>

⁷<https://github.com/rcore-os/rCore-Tutorial-v3/pull/79#issue-1251450181>

⁸<https://github.com/mit-pdos/xv6-riscv/blob/f5b93ef12f7159f74f80f94729ee4faabe42c360/kernel/main.c#L32>

5.5 栈空间过小

在解决缓存数据丢失问题的过程中，使用 gdb 进行单步追踪发现在 vector 的扩容环节发生了错误。经过进一步追踪，发现 linked_list_allocator 库函数中的表达式 `8.is_power_of_two()` (`linked_list_allocator-1.61/src/lib.rs::`

`align_down_size()` 错误的返回了假，产生了**非预期的行为**。根据我们的开发经验，非预期的函数行为一般是缓冲区溢出导致的。加之，在另一次 debug 中，我们发现程序在 **trampoline 位置重复触发 PageFault**。我们确定这时栈分配过小导致的。在若干个断点位置进行分析后，我们将内核栈大小由原来的 8KB 扩展到 16KB，由此解决了问题。

5.6 页引用计数更新

我们最初直接沿用了学长的设计使用了 vector 保存 FrameTracker 来追踪页的引用计数。后来发现在 cow remap 时没有释放原有的 FrameTracker，这导致了页的引用计数不会正确减少，导致 cow 的实现不完整。具体表现为父子进程在访问虚拟地址时都会重新申请一个新物理页，不能恢复具有 COW 标志的原有物理页。我们换用 BTreeMap 保存 VPN 到 FrameTracker 的映射，加快了查找速度，而且在 remap 时只需对 BTreeMap 进行 insert 操作就能自动释放原有 FrameTracker。

6 未来与展望

6.1 页面置换

页面置换用于在物理内存空间不足的情况下，将一些不常用的页面从物理内存中换出，以便为新的页面腾出空间。它可用于解决物理内存不足的情况下如何管理内存的问题。如，通过页面置换机制，可以在物理内存不足的情况下，及时为新进程或应用程序腾出足够的内存空间，从而提高系统的稳定性。引入页面置换可以提高 BTD OS 的鲁棒性，提高其管理内存的能力，为此我们计划引入页面置换技术更新强化我们的系统。

6.2 大页

在 SV39 中，如果使用了一级页索引就停下来，则它可以涵盖虚拟页号的高 21 位为某一固定值的所有虚拟地址，对应于一个 2MB 的大页；如果使用了二级页索引就停下来，则它可以涵盖虚拟页号的高 30 位为某一固定值的所有虚拟地址，对应于一个 1GB 的大页。以同样的视角，如果使用了所有三级页索引才停下来，它可以涵盖虚拟页号的高 39 位为某一个固定值的所有虚拟地址，自然也就对应于一个大小为 4KB 的虚拟页面。使用大页的优点在于，当地址空间的大块连续区域的访问权限均相同时，可以直接映射一个大页，从时间上避免了大量页表项的读写开销，从空间上降低了所需节点的数目。由于小页需要更多的页表项来映射同样大小的虚拟内存空间，因此在使用小页的情况下，系统的页表大小会更大，导致内存开销增加。针对这一问题，我们计划使用大页，期望在一定程度上避免该类问题的发生。

7 尾声

分工和协作

刘卓敏：主要负责文件系统模块以及地址空间的完善。

宋相呈：主要负责进程模块和地址空间的设计与构建、编写测试环境。

刘潞：完善地址空间与文件系统，代码审查与调试，修复内核问题。

指导教师

章复嘉，赵建勇

仓库地址

https://gitlab.eduxiji.net/bite__the__disk/oskernel2023_bitethedisk