



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

0opS 操作系统 设计文档

小组名字: 0opS

小组成员: 张艺枫、李诺舟, 刘嘉琛

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

2022 年 08 月

目录

1 概述	1
1.1 完成情况	1
1.2 OopS 介绍	3
1.3 操作系统整体架构	3
1.4 文档相关	5
2 进程管理	6
2.1 进程	6
2.1.1 进程控制块	6
2.1.2 进程的状态	9
2.2 进程管理器	10
2.3 Processor	11
2.4 进程的阻塞与唤醒	12
2.5 中断和异常处理	13
3 内存管理	14
3.1 内核地址空间	14
3.2 用户程序地址空间	15
3.3 内存分配器	15
3.4 懒分配	16
3.5 内存相关系统调用	17
4 文件系统模块	20
4.1 概述	20
4.1.1 虚拟文件系统	20
4.1.2 总体架构	22
4.2 虚拟文件系统管理器	24
4.2.1 解析路径	24
4.2.2 挂载文件系统	25
4.3 块缓存层	26
4.4 设备管理	27
4.5 FAT32	28
4.5.1 文件一致性	29

4.5.2 空闲簇缓存	30
5 信号处理	31
5.1 信号控制块	32
5.2 信号处理	32
6 总结与未来计划	34
6.1 总结	34
6.2 未来计划	34

1 概述

1.1 完成情况

截至 2022 年 8 月 1 日, OopS 能够支持能够在 QEMU 和 K210 平台上通过决赛第一阶段的全部测试。

我们选择的赛道是 K210 赛道, 下图是我们队伍(OopS/哈尔滨工业大学深圳)的测试通过情况截图 (todo:换成 k210 的截图) :

图 1 测试通过情况

此外, OopS 还能够较好的支持 busybox, 能够运行 busybox 的 sh 以及部分指令, 如 ls、touch、mount 等。

各个部分的具体完成情况如下表所示:

部分	完成情况
进程管理	实现基本的进程与线程管理功能, 能够通过决赛第一阶段测试
内存管理	实现基本的内存管理功能, 能够通过决赛第一阶段测试 实现 mmap, munmap 系统调用 使用 Lazy_allocation 和 Copy_on_write 优化 实现了简单的页面置换 实现了简单的动态链接
文件系统	完成虚拟文件系统 完成 FAT32 文件系统 完成设备设备文件系统
信号机制	完成基础的信号机制
多核支持	在引入线程后不太稳定, 需要完善
设备管理	尚不完善
用户程序	能够支持决赛 musl libc 库, 以及 busybox 的运行

表 1 各部分完成情况

目前 OopS 一共支持以下系统调用：

类别	系统调用	
进程相关	SYS_EXIT 93 SYS_EXIT_GROUP 94 SYS_SET_TID_ADDRESS 96 SYS_FUTEX 98 SYS_SET_ROBUST_LIST 99 SYS_GET_ROBUST_LIST 100 SYS_NANOSLEEP 101 SYS_SCHED_YIELD 124	SYS_GETPID 172 SYS_GETPPID 173 SYS_GETTID 178 SYS_CLONE 220 SYS_EXECVE 221 SYS_WAIT 260 SYS_PRLIMIT 261
内存相关	SYS_BRK 214 SYS_MUNMAP 215	SYS_MPROTECT 226 SYS_MMAP 260
文件相关	SYS_GETCWD 17 SYS_DUP 23 SYS_DUP3 24 SYS_FCNTL 25 SYS_IOCTL 29 SYS_MKDIRAT 34 SYS_UNLINKAT 35 SYS_LINKAT 37 SYS_UMOUNT 39 SYS_MOUNT 40 SYS_STATFS 43 SYS_FACCESSAT 48 SYS_CHDIR 49 SYS_OPENAT 56 SYS_CLOSE 57	SYS_PIPE2 59 SYS_GETDENTS 61 SYS_LSEEK 62 SYS_READ 63 SYS_WRITE 64 SYS_READV 65 SYS_WRITEV 66 SYS_PREAD 67 SYS_PWRITE 68 SYS_SENDFILE 71 SYS_PPOLL 73 SYS_NEWFSTATAT 79 SYS_FSTAT 80 SYS_UTIMENSAT 88

信号相关	SYS_KILL 129	SYS_SIGACTION 134
	SYS_TKILL 130	SYS_SIGPROCMASK 135
	SYS_TGKILL 131	SYS_SIGRETURN 139
其它	SYS_TIMES 153	SYS_GETTIMEOFDAY 169
	SYS_UNAME 160	

表 2 目前支持系统调用

1.2 OopS 介绍

我们的总体目标是设计一个精简，结构清晰，扩展性良好的操作系统，我们的目前主要精力放在了实现更多的功能，即更多的系统调用之上，而不是在性能之上。我们希望能够在决赛第一阶段截止日前，通过测试平台的测试，并在此基础上，尽可能地支持 busybox，特别是 busybox 的 sh，因为如果能够支持 busybox 的 shell，也就意味着该操作系统拥有最基本的交互能力和能够支持最基础的功能。目前我们的操作系统已经基本达成了决赛第一阶段的目标。

此外，我们的操作系统能够在 QEMU、K210、SIFIVE 平台上运行，只需要在编译的时候选择操作系统的目标运行平台，就能够编译出不同平台上的 os.bin 镜像。

从 0 开始写一个操作系统会造成很多不必要的开销，所以我们和上一届参赛作品 UltraOS 一样，是在 rCore Tutorial v3¹ ch5 分支的基础上进行开发该分支已经实现了基本的进程管理、内存管理、中断和异常处理功能我们在开发的过程中，对这部分基础的代码进行了删减和改动此外，我们的操作系统开发的过程中还参考了上一届参赛作品中的 Oshit_kernel²，特别是文件系统部分。

1.3 操作系统整体架构

操作系统可以根据运行时的特权级分为 3 个部分：

¹ <https://github.com/rcore-os/rCore-Tutorial-v3>

² 啊普鲁派哒哒哒哒哒"; DROP DATABASE teams;，https://gitlab.eduxiji.net/willson0v0/oscomp_handin

- **SBI (Supervisor Binary Interface):** SBI 本身运行在 M 态，SBI 实现了对硬件的抽象,是操作系统内核的运行环境为操作系统内核提供了统一的 SBI 接口,这些接口可以实现: 标准输入输出, 进程间中断等功能 SBI 还可以处理来自内核的异常, 模拟开发板不支持的指令集, 比如 `sfence.vma`。
- **内核:** 操作系统的内核运行在 S 态, 是用户程序的运行环境 OopS 的内核一共可以分成 4 个模块: 进程管理模块、内存管理模块、文件系统模块, 进程通信模块 (信号系统)。
- **用户程序:** 用户程序运行在 M 态, 一些操作系统具备有基础的用户程序或程序库, 比如 `libc`、桌面应用程序、浏览器等不过我们并不打算实现任何用户程序或程序库, 我们的计划是让我们的内核支持一些现有的用户程序, 比如 `busybox`。

操作系统的架构如图 2 所示:

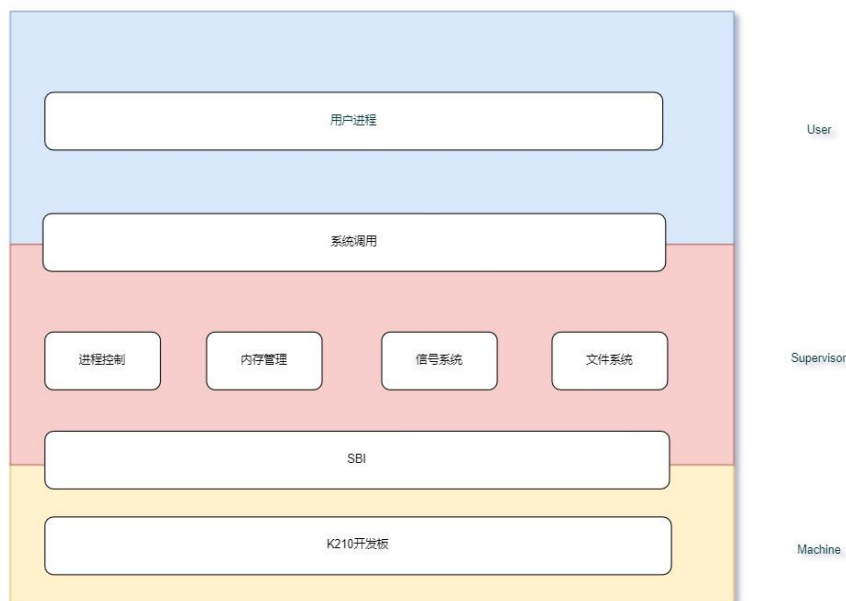


图 2 系统架构图

1.4 文档相关

本文档仅作为初赛的报告文档，而不是 OopS 操作系统的最终文档主要作为本组目前工作的记录与梳理，很多功能的实现并不是最终的方案，语言与排版方面比较随意。

此外，该文档提供的列出的代码与源代码不一样，而是为了可读性删除了一些细节，或者是伪代码。

2 进程管理

由于 OopS 是在 rCore Tutorial v3 的基础上进行开发的, 进程管理模块的基本框架与 rCore 类似。所以在本文档中, 对 rCore Tutorial 包含的部分只作简单的说明, 对 OopS 独有的部分则作较详细的介绍。

OopS 支持进程和线程, 进程和线程都是用任务(Task)统一表示。进程管理模块的顶层是进程管理器 (TaskManager) 和 Processor 结构 (对 CPU 的抽象), 底层是进程控制块(TaskControlBlock)。

在本章中, 将先后介绍:

- 任务控制块: 该结构用来表示一个进程或线程
- 进程管理器: 包含进程的调度算法
- Processor 结构: 涉及调度进程时, 控制流的切换方式
- 进程的阻塞和唤醒
- 中断和异常处理: 涉及进程用户态和内核态之间的切换

2.1 进程和线程

在 OopS 内核中, 进程和线程两者并没有区别, 可以统一地称之为任务(Task), 用数据结构任务控制块(TaskControl)表示。线程可以理解为共享了资源的任务, sys_clone 系统调用时, 能够指定子进程和父进程之间共享哪些资源(包括地址空间、文件描述符表、待处理信号等)。线程模型的具体定义可以由用户库负责。

2.1.1 任务控制块

在 OopS 中代表任务控制块的数据结构是 TaskControlBlock。其中存放着用于描述进程状态的全部信息。任务控制块的包含可以共享的数据结构以及 Task 独享的数据结构。目前 OopS 的任务控制块主要包含以下数据结构(其中**粗体**的是可在任务间共享的):

类别	详细信息
进程相关	进程号 pid, 线程号 tid, 线程组号 tgid 任务状态 TaskStatus

	任务上下文 TaskContext 内核栈 KernelStack 父进程 Parent 线程组组长: leader 退出码 exit_code 阻塞标识 Channel (用于任务的阻塞和唤醒) 资源信息 rlim 子进程集合 child 线程组任务集合 groups
内存地址相关	进程 TrapFrame 的物理页号 用户地址空间布局 memory_set
文件系统相关	文件系统信 fs_info 文件描述符表 fd_table
信号相关	信号屏蔽掩码 sig_mask 线程待处理信号: t_pending 退出信号: exit_signal 信号处理函数: handlers 待处理信号 p_pending
其它	在 U 态和 S 态消耗的时间

表 3 进程控制块成员

进程控制块的代码如下图所示:

```
pub struct TaskControlBlock {
    /* 0: Ready; 1: Running; 2: Stopped; 3: Zombie */
    pub status: AtomicU8,
    pub kernel_stack: KernelStack,
    pub chan: Channel,

    /* id */
    pub pid : i32,
    pub tid : i32,          /* global tid */
    pub private_tid: i32,  /* private_tid */
}
```

```

pub tgid: AtomicI32,

/* exit */
pub exit_code :AtomicI32,
pub exit_signal :AtomicU64,

/* 在睡眠时是否被中断 */
pub interrupted :AtomicBool,

/* 线程处理信号函数时的上下文指针 */
pub signal_context_ptr :Mutex< usize>,
pub task_info :Mutex< TaskInfo>,
pub parent :Mutex< Option<Weak<TaskControlBlock>>>>,
pub leader :Mutex< Option<Weak<TaskControlBlock>>>>,
pub time_info :Mutex< TimeStruct>,
pub sig_mask :Mutex< usize>,
pub t_pending :Mutex< SigPending>,
pub s_trap_cx :Mutex< Vec<TrapContext>>>,
pub robust_list :Mutex< Option<RobustList>>>,

/* 可以共享的成员 */
pub memory :Arc<Mutex< MemorySet>>>,
pub child :Arc<Mutex< Vec<Arc<TaskControlBlock>>>>>>,
pub groups :Arc<Mutex< ThreadGroup>>>,
pub rlim :Arc<Mutex< RlimArr>>>,
pub fs_info :Arc<Mutex< FsStruct>>>,
pub fd_table :Arc<Mutex< FdTable>>>,
pub handlers :Arc<Mutex< SigHandlers>>>,
pub p_pending :Arc<Mutex< SigPending>>>,
pub futex_list :Arc<Mutex< FutexList>>>,
pub buffer_list :Arc<Mutex< BTreeMap<VirtPageNum, BufferFrame>>>>
}

```

在内核代码中，一般通过 Rust 的 Arc 智能指针来访问进程控制块。由于不能直接获得 Arc 指针指向的数据结构的可变引用，所以需要将进程控制块可变的数
据用 Mutex 包装，这样可以利用 Rust 的内部可变性来修改进程控制块的成员。

可以共享的数据结构都用 Arc 包装，这样可以直接利用 Arc 的 clone 方法，
使得两个任务的数据结构指向同一个实例，该过程是在 sys_clone 的时候进行。
如果选择不进行共享，就创建的的数据结构。如下所示

```

/kernel/src/proc/task.rs/copy_process
if clone_flags.contains(CloneFlags::THREAD) {
    parent = self.parent.lock().clone();
    child = self.child.clone();
    leader = self.leader.lock().clone();
    groups = self.groups.clone();
    .....
} else {
    parent = Some(Arc::downgrade(self));
    child = Arc::new(Mutex::new(Vec::new()));
    leader = None; //self
    groups = Arc::new(Mutex::new(ThreadGroup::new()));
    .....
}

```

2.1.2 进程的状态

在 OopS 中，进程有 4 种状态：

- Running: 该进程正在被执行
- Ready: 可以被调度到处理器上执行
- Blocked: 被阻塞，等待被唤醒
- Zombie: 僵尸进程，等待被父进程回收

这些状态的相互转换关系如下图所示，其中箭头表示使转换发生的函数：

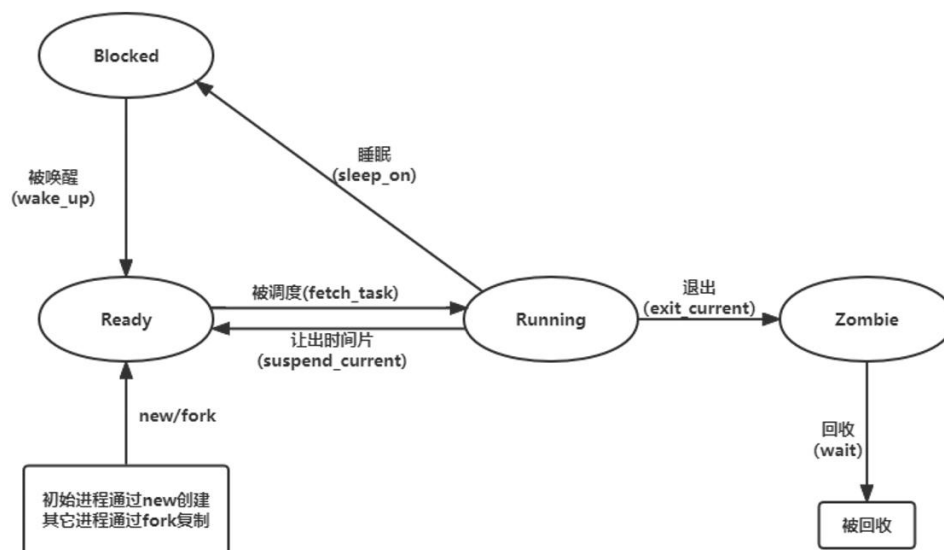


图 3 进程的状态转换图

2.2 进程管理器

进程管理器 (TaskManager) 保存着所有所有进程控制块的 Arc 指针，主要作用是选择被调度的进程，目前只有一个进程调度器，使用的算法是最简单的 FIFO，所有的进程的优先级都相同。

进程管理器提供了 5 个接口：

接口	功能描述
add_task	添加一个进程到进程管理器的就绪进程列表
fetch_task	从进程管理器的就绪进程列表取出一个进程
add_blocked_task	添加一个进程到阻塞进程列表
wake_blocked_task	从阻塞进程列表中唤醒指定进程
get_task_by_tid	获取指定 pid 的进程的指针

图 4 进程管理器接口

为了保持多核情况下的一致性，整个进程管理器被一个自旋互斥锁包装，所以调用这些接口可能会造成阻塞由于这些接口的使用频率较高，所以目前的实现会造成不少的性能损失。

进程管理器有 3 个成员：

- ready_tasks: 保存就绪进程为了实现 FIFO 算法，数据结构为队列
- stopped_tasks: 保存阻塞进程为了唤醒进程时能够更快的索引，数据结构为 B 树表
- running_tasks: 保存正在被运行的进程该结构是一个数组，如果 id 为 i 的 CPU 正在运行进程 P，那么数组第 i 个元素为进程 P 的进程控制块指针

```
pub struct TaskManager {  
    pub ready_tasks: ReadyList,  
    pub stopped_tasks: StoppedList,  
    pub running_tasks: [Option<Arc<TaskControlBlock>>; MAX_CPU_NUM]  
}
```

2.3 Processor

Processor 是对 CPU 的抽象，开发板上的每一个核心都与内核中的一个 Processor 对应 Processor 保存在一个数组之中，可以用核心的 id 作为索引获得对应的 Processor。

Processor 结构保存了：

- 在该 CPU 上运行的进程控制块
- 该 CPU 上进程切换任务控制流的上下文
- 该 CPU 的 ASID 分配器
- 该 CPU 的互斥锁嵌套深度 `n_off`
- 该 CPU 获取互斥锁前的的中断状态 `intena`

每当要切换进程时，都会将当前控制流切换到对应的 Processor 的进程切换任务控制流的上下文在这个控制流中，会执行 `scheduler` 函数，该函数会不断的调用 `fetch_task`，直到从进程管理器中取出一个进程，然后切换到这个进程的上下文中执行。

在内核中，如果进程 A 获取了一个自旋锁的锁，然后进程 A 的控制流由于中断等原因被切换成其它的进程这个时候其它的进程想要获取该自旋锁时就会发生阻塞现象，直到进程 A 的控制流被切换回来后将锁释放，这种阻塞会造成极大的性能浪费所以必须要保证进程在获取了自旋锁之后，其控制流不能够被中断打断为此，我们在 Processor 结构中增加了 `n_off` 和 `intena`。

获取和释放自旋锁的时候，会分别调用 `push_off` 和 `pop_off` 函数，这回分别导致 Processor 上的 `n_off` 加 1 和减 1 如果 `n_off` 由 0 到 1 时，会将当前的中断开启状态记录到 `intena` 中，如果 `n_off` 由 1 到 0 时，会根据 `intena` 设置中断开启状态这样就可以保证进程获取了自旋锁时，中断是关闭的当获取自旋锁被释放之后，中断状态还原成获取自旋锁之前的状态这种实现方式参考的是 xv6。

2.4 进程的阻塞与唤醒

我们目前实现的进程的阻塞与唤醒方法并不完善，主要是因为目前的实现效率低下，而且使用比较不方便，所以还需要改进。

进程的阻塞与唤醒需要使用进程管理器提供的 2 个接口：

- `wake_up(channel)`: 唤醒在 `channel` 上睡眠的进程
- `sleep_on(channel)`: 使该进程在 `channel` 上睡眠

进程的阻塞与唤醒的使用例子如下图所示：

1. 在向管道读数据时，进程的阻塞和唤醒
2. 进程在 `wait` 回收子进程时，进程的阻塞和唤醒

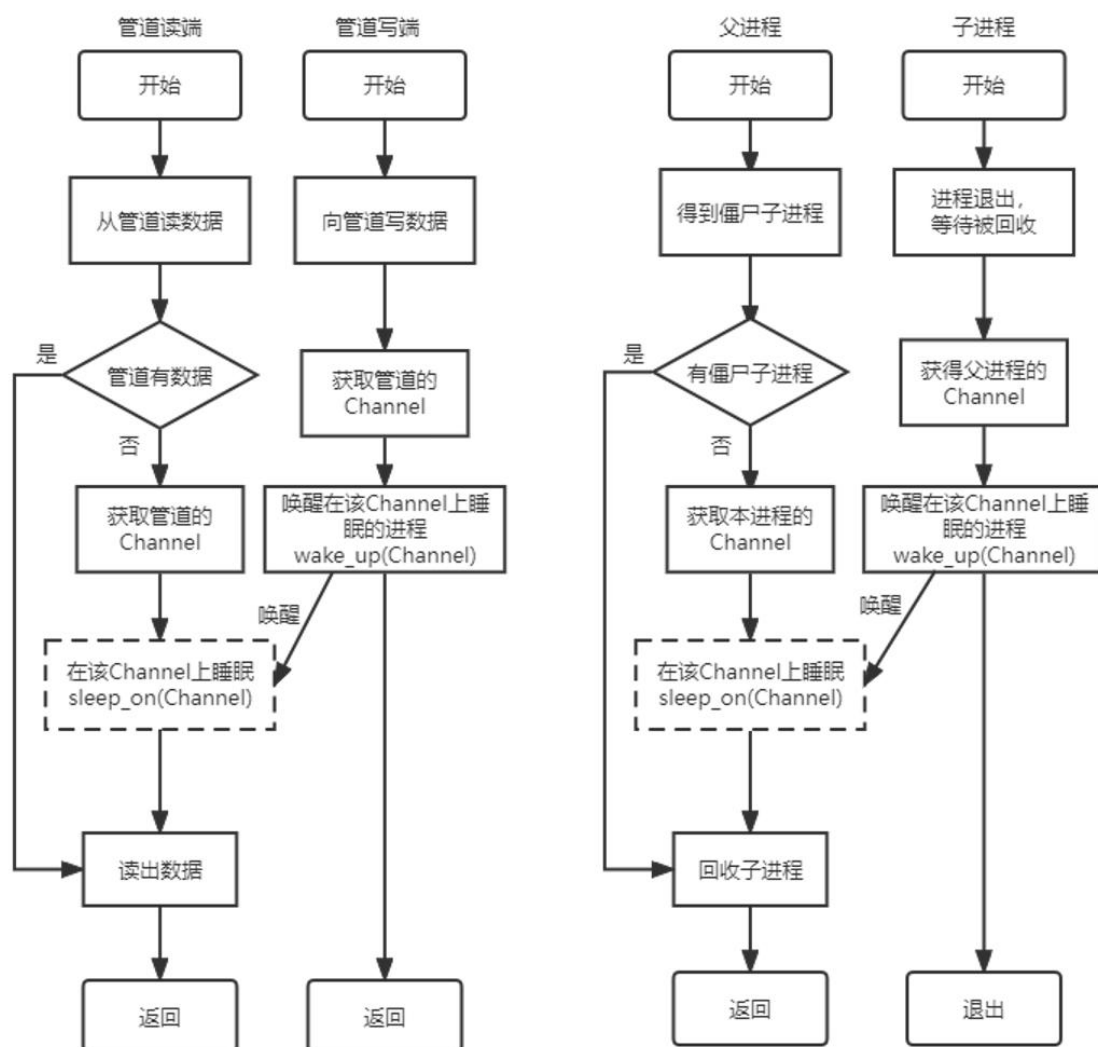


图 5 进程的阻塞和唤醒例子

其中的 `Channel` 本质上是一个标识，只需要保证所有的 `Channel` 之间是可比較的，而且所有的 `Channel` 都不一样即可。`Channel` 本质上是一个 `usize`，由一个全局的分配器分配，可以保证每次分配的数字都不相同。

2.5 中断和异常处理

在 `OopS` 的中断和异常处理和 `rCore` 是类似的。

如果在用户态遇到中断或异常，那么处理流程为：

- 1，根据 RISC-V 标准，`pc` 保存到 `sepc`，由 U 态切换为 S 态，将当前 `pc` 设置为跳板代码 `Trampoline` 的首地址。
- 2，执行 `Trampoline`，将用户态上下文保存到 `TrapFrame` 中，加载用户态上下文。
- 3，进入内核的 `trap` 处理函数 `trap_handler`，处理中断和异常。
- 4，如果成功处理中断和异常，从 `TrapFrame` 中恢复上下文，返回到保存在 `sepc` 里的 `pc`，否则进程终止。

如果在内核态遇到中断和异常，处理流程为：

- 1，`pc` 保存到 `sepc`，将当前 `pc` 设置为 `kernel_vec` 的首地址。
- 2，执行 `kernel_vec`，将上下文保存到内核栈中。
- 3，进入内核的 `kernel_trap_handler`，处理中断和异常。
- 4，从内核栈中恢复上下文，返回到保存在 `sepc` 里的 `pc`。

目前，在如果内核态遇到中断和异常，那么该中断一定是硬件中断，而且为计时器中断如果在内核态遇到异常，那么终止进程。

3 内存管理

内存管理设计主要是地址空间的布局以及懒加载的实现，以及 2 个重要的系统调用：`mmap`、`munmap`。

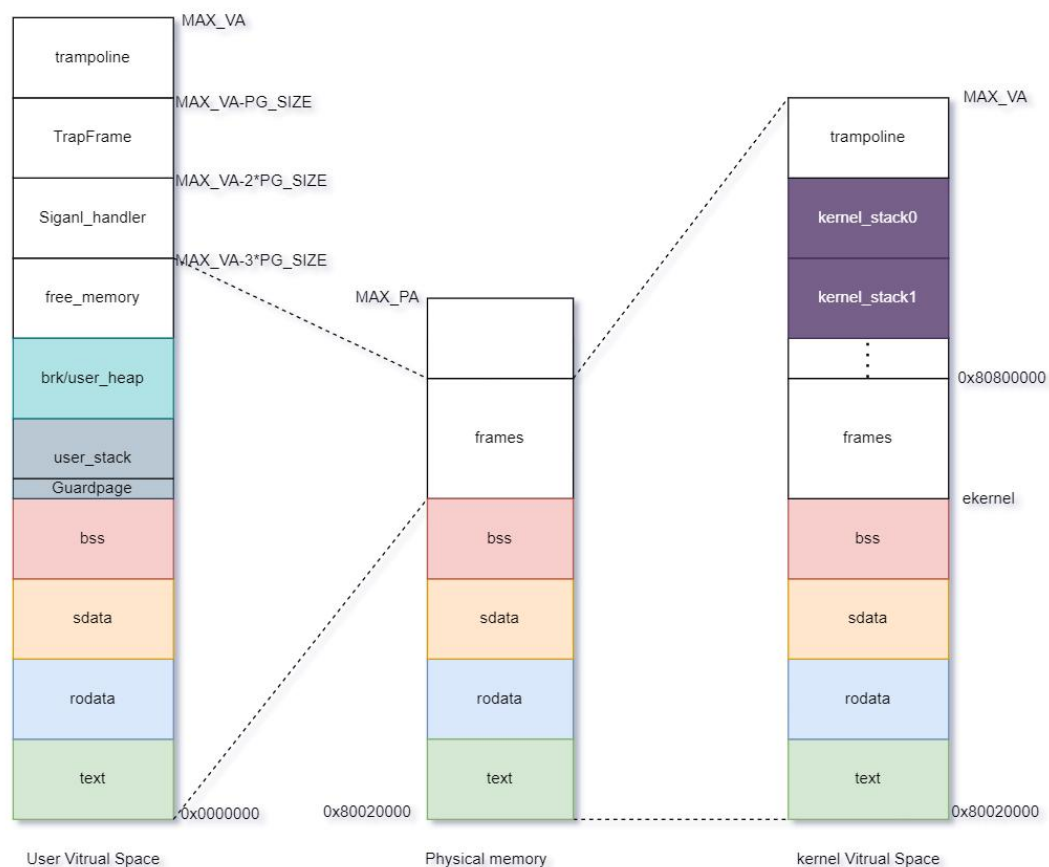


图 6 地址空间

3.1 内核地址空间

内核地址在 `0x80020000` 到 `0x80800000` 的虚拟地址恒等映射到物理内存中内核地址空间的最高页设置了 `trampoline` 的映射， 尽管 `trampoline` 的代码实际存在于物理内存中的 `text` 段中，但为了保证页表切换时虚拟地址的连续性，在此处设置了 `trampoline` 额外的一个映射。

内核地址空间如图 6 右侧所示。

3.2 用户程序地址空间

用户地址空间是根据用户程序的 `elf` 文件建立的在代码数据段的上方是用户栈，然后是堆在用户地址空间顶部有三段：`signal_handler`、`trapframe`、`trampoline`。

`signal_handler` 操作系统中关于信号处理的默认函数的存储区域，映射到物理内存的 `text` 区域中的信号处理函数代码而 `trapframe` 用于保存保存内核态和用户态的寄存器上下文 `trampoline` 映射到内核态和用户态之间的跳板代码。

用户程序地址空间布局如图 6 左侧所示。

3.3 内存分配器

`rCore` 一样，`OopS` 有两个内存相关的分配器：动态内存分配器、物理页帧分配器。

动态内存分配器使得内核代码可以使用 `Vec` 或 `BtreeMap` 等动态数据结构动态内存分配器使用 `buddy_system_allocator` 库来完成在代码中用数组占用了一块内存区域，在初始化分配器时，这块区域会分配给动态内存分配器进行内存分配。

动态内存分配器被一个自旋互斥锁包装进行动态内存分配时首先要获取该自旋锁由于动态内存操作是非常频繁的，所以当进程并发运行时，会有大量的锁竞争消耗如果有办法能够消除或减少这些锁竞争消耗，则能够提高运行效率（目前未实现）。

我们的物理页帧分配器的分配方式为栈式分配³内核代码段结尾到内存空间末尾的区域是物理页帧区，这段区域的内存可以以页为单位，被物理页帧分配器分配给各个进程我们和 `rCore` 一样，使用了 `RAII` 方法，用 `FrameTracker` 数据结构来代表一个物理页帧，通过实现 `FrameTracker` 数据结构的 `drop` 方法能够实现物理页帧的自动回收。

物理页帧分配器也是用互斥锁包装所以和动态内存分配器一样会有锁竞争消耗我们可以为每个核分配一个物理页帧分配器，这样就能够大大的减少锁竞争

³ 详细见 https://wiki.osdev.org/Page_Frame_Allocation 的 Stack/List of pages

（目前未实现）。

3.4 懒分配

在一些情形中，刚分配的物理页可能永远都不会被访问，这些不被访问的物理页是没有必要分配的不必要的分配就会造成内存空间和时间的浪费为了解决这个问题，可以使用懒分配优化方法：只有在访问物理页的时候才分配物理页帧，并对该物理页进行初始化借助缺页异常可以实现懒分配优化策略。

以下目前利用了懒分配优化策略的 3 中情形：`sys_brk`、`mmap`、`fork`。

①`sys_brk` 被用于申请新内存时，只是修改 `brk` 虚拟段中的虚拟页范围，并不进行页表的填写，同时通过一个 `lazy_flag` 枚举变量来表明该段使用了懒分配的机制。当要对 `brk` 新申请的内存进行读写时，会因为页表寻址失败而触发缺页异常。此时操作系统根据发生缺页异常的地址所在段的 `lazy_flag` 来决定它的下一步动作。在这里，`lazy_flag` 被设置为 `NORMAL`，表明该段是一个普通的懒分配段，那么操作系统就会为发生缺页异常的虚拟页分配内存并填写页表，然后重新进行内存的读写。

②`mmap`：`mmap` 不仅需要分配新内存，还需要把文件中的数据写入内存中供操作系统进行读写因此 `mmap` 的懒分配机制需要跟 `brk` 区分开来，`mmap` 段的懒分配标志将被设置为 `MMAP`，表明这是一个需要 `mmap` 懒分配的段，在进行 `mmap` 系统调用时，除了改变 `mmap` 段的虚拟页范围之外，还会将映射的文件对应的 `file` 结构体以及偏移量 `offset` 记录在 `mmap` 段中以便后续进行懒分配。在对 `mmap` 段中的页进行第一次读写时，会因为页表寻址失败而出缺页异常。操作系统发现 `lazy_flag` 为 `MMAP`，便会对发生缺页的虚拟也分配内存，并从对应的文件中取出一页数据写入新分配的物理页中。然后重新进行内存的读写。

③`fork`：`sys_clone` 在克隆子进程时，会涉及到父进程地址空间的复制如果不使用懒加载，那么需要为新的子进程地址空间分配物理内存并将父进程地址空间中的数据写入其中。而父进程中有些数据段并不会被写入导致改变。如果能够让父子进程共享这些数据段，那么会减少很多时间和空间开销。`Rust` 语言为这种共享的需求提供了 `Arc` 数据结构。我们将与物理页绑定的 `Frametracker` 放入 `Arc` 数据结构中，就可以允许不同进程使用同一个物理页了。

有了可以共享的物理页，在进行地址空间复制的时候，操作系统不再对子进程分配新物理页并复制数据，取而代之的是通过 `arc` 的 `clone` 方法实现让子进程的虚拟页映射到父进程的物理页上。同时取消父子进程共享页的写权限，以便在发生写操作的时候可以触发缺页异常进行处理。

与 `brk` 段和 `mmap` 段不同，我们并不需要一个标志变量来指明该段正在与其他进程共享。一是所有数据段都有被共享的可能，二是被共享的物理页可以通过检测该物理页的引用计数被检测到。因此当写物理内存触发缺页中断时，操作系统会检测该页的引用计数，如果大于 1 说明该页正在被共享，那么就会给正在写内存的进程分配新的物理内存。然后重新进行写内存的操作。

3.5 内存相关系统调用

本次初赛涉及到的内存相关系统调用有：`sys_brk`，`sys_mmap`，`sys_munmap`。

①`sys_brk`:当用户进程申请内存时，一般会调用该系统调用。工作机制:从用户进程刚刚初始化完成时的内存布局图中我们可以看到，`user_stack` 上方是尚未分配的的虚拟内存区域。

如图 7 所示，`program_end` 下方的虚拟段一般是不会发生变动的，因此如果要增加内存区域，我们只能向上申请新的虚拟段并分配物理内存。`sys_brk` 系统调用的作用就是增加用户进程可用的物理内存。

根据 `brk` 的特点，该系统调用所申请的物理内存存在虚拟地址上是连续的，因此我们统一用一个虚拟内存段来管理所有 `brk` 申请得到的物理内存。

在这里我们用一个指针标记起始 `brk` 起始的虚拟地址，防止错误释放代码段和用户栈区域的内存，另一个指针标记此时 `brk` 段的最高地址，下次分配获释放就从这里开始。

同时懒加载机制也可以很好地应用到 `brk` 上:申请内存时我们可以只用简单地移动 `current_end` 指针并修改 `brk` 段的结束地址，等到用到的时候再分配;释放内存时如果实际物理内存并没有被使用到，则可以简单地修改指针和 `brk` 段结束地址，这减少了内存分配的次数。

②:`sys_mmap`:`sys_mmap` 系统调用可以映射文件内容到内存上，将读写外存

转换为读写内存，这提高了文件的读写速度。

根据 `mmap` 系统调用的要求，我们需要将文件的部分内容映射到内存中。考虑到文件映射的内存区域往往有着一致的属性，因此每次 `mmap` 我们都会新建一个 `mmap` 虚拟段负责管理这块区域。这就涉及到新的内存分配问题。考虑到可用虚拟地址中，`brk` 系统调用会在低地址分配内存，为了保持 `brk` 分配得到的虚拟地址的连续性和避免冲突，我们决定让 `mmap` 段从高地址向下开始分配。如图 7 所示。

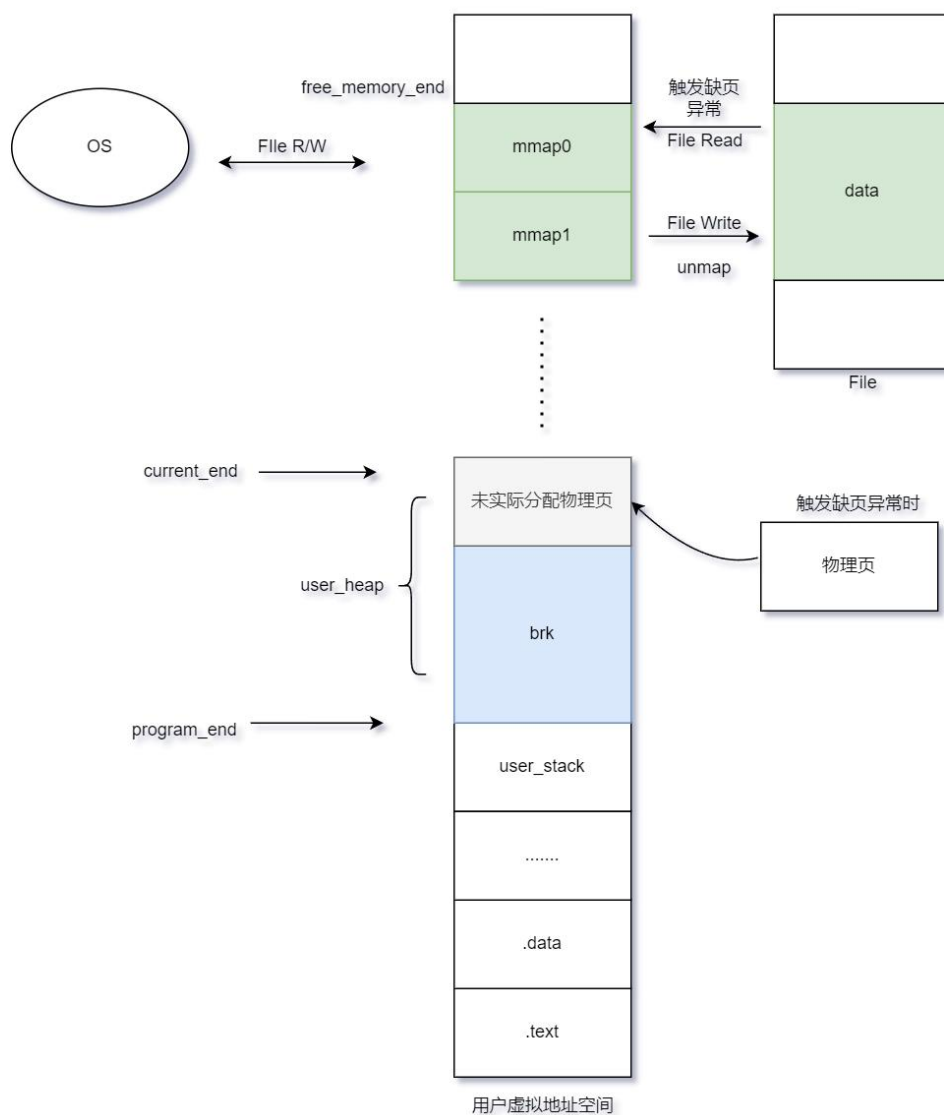


图 7 内存部分系统调用示意图

但是这种做法需要把指定的内容从外存上全部读到内存中，速度很慢，我们可以再次利用懒加载的机制，只是在当前进程地址空间中新建立一个 `mmap` 段，并记录下对应的文件和偏移量，就可以完成我们的系统调用了。后续的工作可以交给缺页处理程序完成：当发生缺页异常并且异常地址处于 `mmap` 段中时，就新分配一页，再从当前段映射的文件中读取一页数据放置到缺页的内存当中。如此就实现了 `sys_mmap` 的大部分要求。

③:`sys_unmap`:该系统调用将 `mmap` 映射的内容写回文件中并解除映射。在当前实现中，`unmap` 仅仅是简单地将一整个 `mmap` 段上的内容写回文件中。后续会实现更精确的 `ummap` 系统调用，能够以页为单位进行映射的解除。

3.6 页面置换

K210 的内存仅有 8MB，难以支持多个较大型的程序同时运行，比如 `busybox` 和决赛测试程序。所以非常有必要实现页面置换算法。目前的页面置换机制是临时实现的，实现方式比较简朴，还需要完善。

当内核需要分配页帧给用户地址空间的时候，如果页帧用完时，就会进行页面置换。内核会选择一个页淘汰出去，目前的选择方式十分简单：选择该进程一个引用计数为 0 的物理页帧，将该物理页帧代表的物理页淘汰出去。淘汰方式为：将该物理页的数据写入缓存文件，并将该页的 PTE 的有效位置 0。

每个进程的任务控制块都维护了一个表，记录了被淘汰出去的页的在缓存文件中的位置。需要重新调回时可以根据记录的位置读取数据。

目前主要需要改善淘汰算法，不足及改善点有如下几点：

- 1，目前只能淘汰引用计数为 1 的页，共享的页不能被淘汰。
- 2，没有利用 PTE 中的 D 和 A 位。
- 3，可以使用更复杂的选择算法，比如 LRU。
- 4，可以一次讲多个页面淘汰出去。

4 文件系统模块

4.1 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 OopS 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。

该模块的设计参考了上一届参赛作品 `oshit_kernel` 的设计。

4.1.1 虚拟文件系统

OopS 的文件系统模块借鉴了 Linux 中 VFS（虚拟文件系统）的设计，使得 OopS 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` 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能让 OopS 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：

- 文件系统抽象接口：VFS `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 `get_root`，通过该接口可以获得文件系统的根文件目录。
- 文件抽象接口：一共有 7 个文件相关的 `trait`，如图 1 所示箭头表示一种类似继承的关系，即如果要实现某个 `trait`，必须先实现其指向的父 `trait` 例如字符设备文件 `CharFile` 需要实现 `CharFile trait`，那么需要首先实现 `File trait` 和 `DirFile trait`。

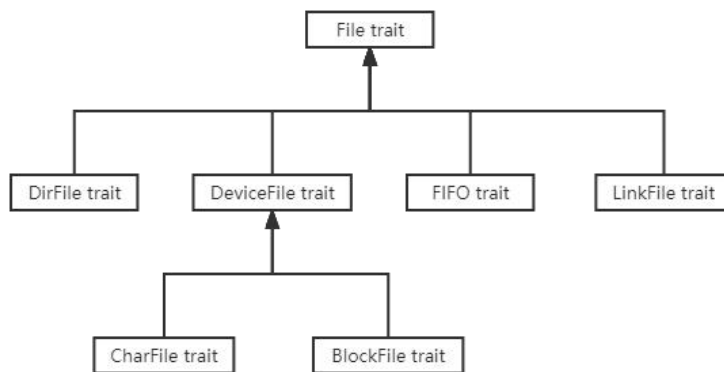


图 8 文件抽象接口继承关系图

OopS 一共有 6 种类型的文件，如下表所示

文件类型	需要实现的 trait
普通文件 (ReguleR File)	File
目录文件 (DirFile)	File + DirFile
管道文件 (FIFOFile)	File + FIFO
链接文件 (LinkFile)	File + LinkFile
字符设备文件 (CharFile)	File + DeviceFile + CharFile
块设备文件 (BlockFile)	File + DeviceFile + BlockFile

表 4 文件类型

为了更好地说明这些文件接口，下面将以表格的形式说明 3 个最重要的接口也可以直接阅读代码 ([/kernel/src/fs/file.rs](#)):

File trait	
接口	描述
read/write	读写文件
readable/writable	文件是否可读/写
read_stat/write_stat	读/修改文件信息
seek	修改文件的 cursor
get_index	获得文件的文件标识 FileIndex
get_size	获得文件的大小

表 5 普通文件接口 File trait

DirFile trait	
接口	描述
openat	打开目录中的文件
mknod	在该目录中创建文件
delete	删除目录中的文件
getdent	获得该目录的目录项

表 6 目录文件接口 DirFile trait

Device trait	
接口	描述
get_id	获得文件的设备 id
ioctl	对设备进行控制

表 7 设备文件接口 Device trait

4.1.2 总体架构

文件系统模块的总体架构如下图所示：

1. 文件系统模块的顶层是虚拟文件系统管理器 (VFS Manager) 虚拟文件系统管理器提供了多个接口，内核的其它模块通过这些接口来访问文件系统模块。文件系统管理器主要负责两个工作：挂载文件系统，解析路径的命令在 4.2 中会对该结构作更详细的说明。
2. 文件系统层：除了根文件系统外，文件系统都在处理 mount 系统调用的时候创建并初始化，并挂载到一个已存在的目录中所有的文件系统都需要实现 VFS trait 目前 OopS 只实现了 2 类文件系统：FAT32 和设备文件系统 DevFS 根文件系统的类型是 FAT32 在 4.5 中会对 FAT32 作更详细的介绍。
3. 块缓存层：文件系统通过块缓存层读写数据，块缓存层会在合适的时间将数据从块缓存中写入设备文件，或从设备文件中读取数据，在 4.3 中会作更详细的介绍。

4. 设备文件层：设备文件是对存储设备的抽象存储设备文件都实现了 `BlockFile trait`，可以通过调用 `read_block`，`write_block` 接口来读写存储设备。
5. 模块最底层是存储设备的驱动，目前使用的是 rCore 开源的 SD 卡驱动

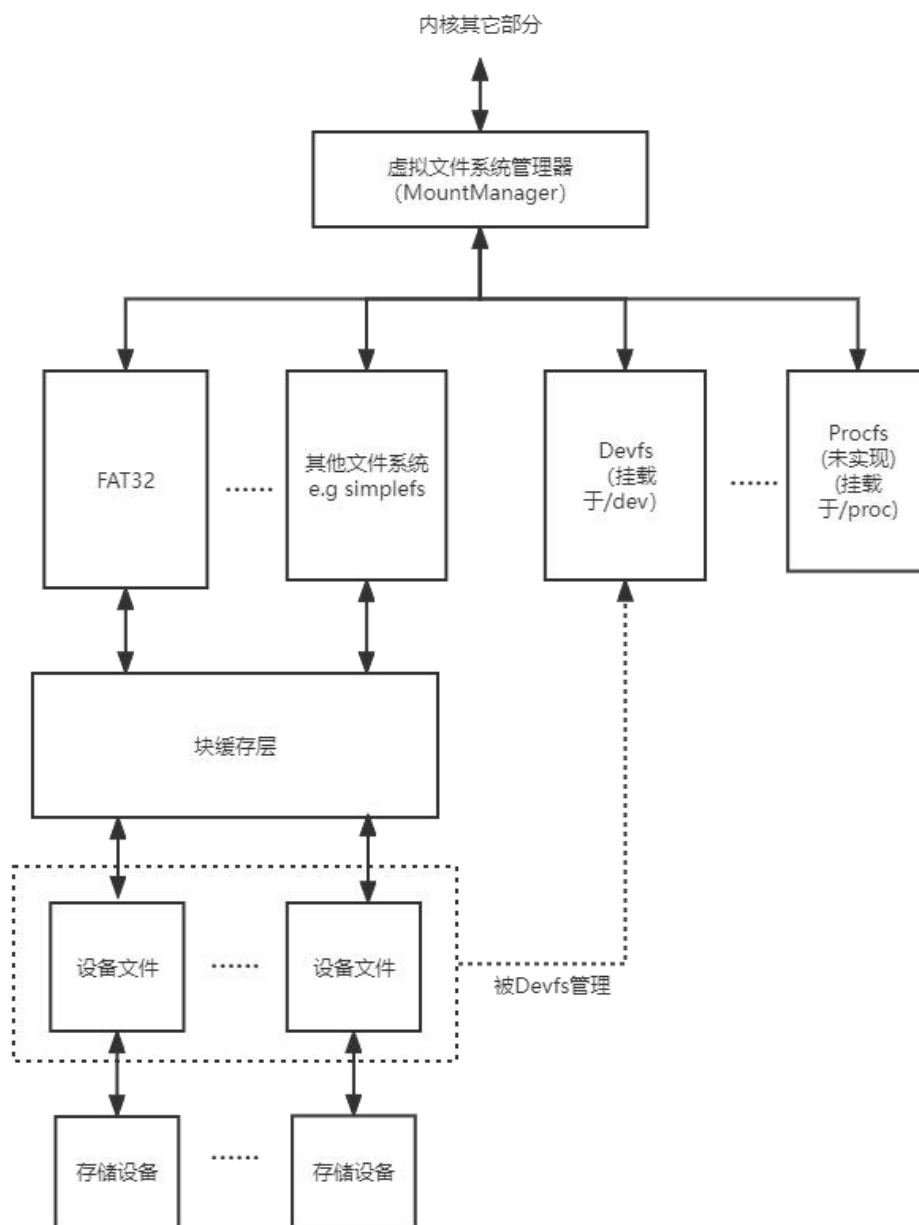


图 9 文件系统架构图

4.2 虚拟文件系统管理器

虚拟文件系统管理器层是文件系统模块的最顶层管理器中包含根文件系统和挂载表主要功能是挂载文件系统以及解析路径。

```
struct VFSManager {  
    root_fs: Arc<dyn VFS>,           //根文件系统  
    map: BTreeMap<FileIndex, Arc<dyn VFS>>, //挂载表  
}
```

该层向内核的其它模块提供以下接口：

接口	功能描述
mount/umount	挂载/卸载文件系统
open/openat	打开文件（绝对路径/相对路径）
mknod/mknodat	创建文件（绝对路径/相对路径）
link	创建链接（绝对路径）
delete	删除文件（绝对路径）

表 8 虚拟文件系统管理器接口

4.2.1 解析路径

解析路径的核心函数是 `open_path`，该函数的函数声明为：

```
fn open_path(  
    &self,  
    current: Arc<dyn File>,  
    path: Path,  
) -> Arc<dyn File>
```

`current` 是一个文件，`open_path` 以 `current` 作为起点，获得路径 `path` 指定的文件解析路径的时候遇到的中间文件不同，可能是目录，也可能是链接文件或挂载点，处理的方式也不同简化版本的 `open_path` 的流程图如图 10 所示。

文件系统管理器的其它的接口的实现都是以 `open_path` 为基础的以 `open` 和 `open_at` 为例：

```
//从根目录开始解析路径  
fn open(&self, path: Path) -> Arc<dyn File> {  
    self.open_path(self.root_fs.root_dir(mode), path)  
}
```

```
//从指定的目录文件 src 开始解析路径
fn open_at(&self, src: Arc<dyn File>, path: Path) -> Arc<dyn File> {
    self.open_path(src, path, mode, 0)
}
```

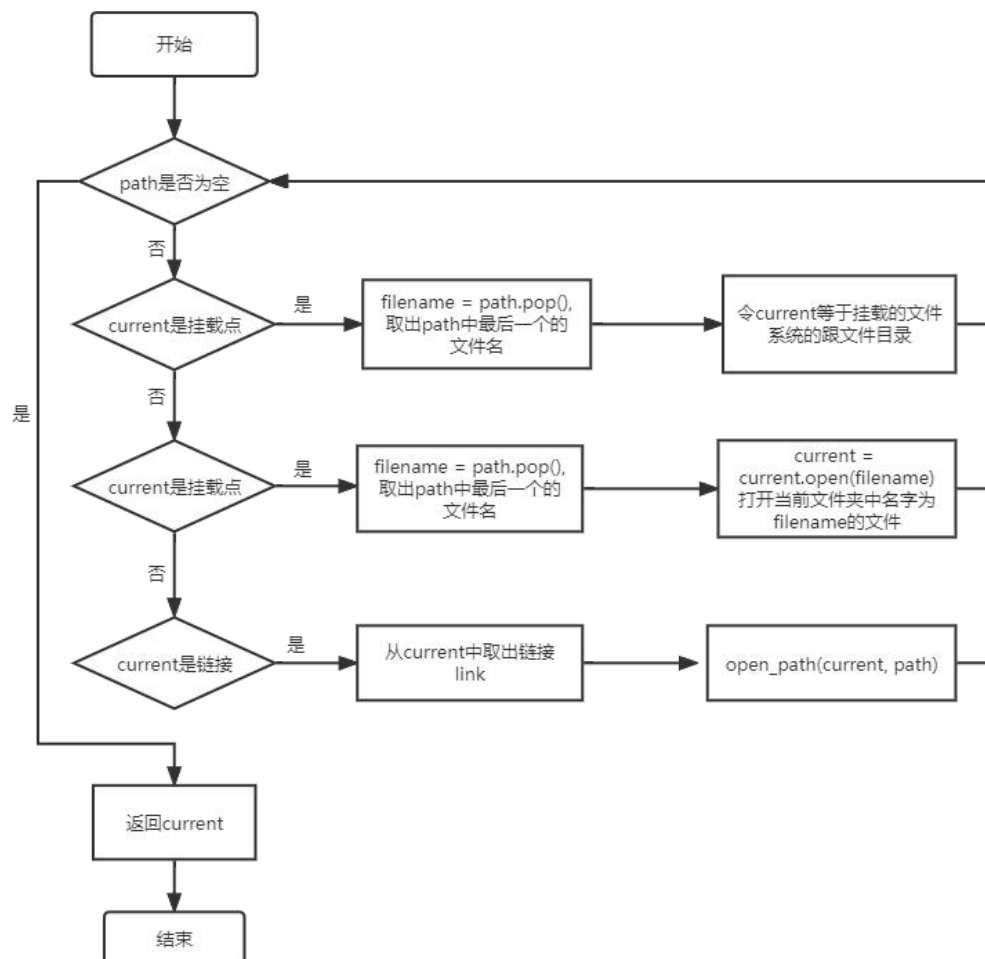


图 10 open_path 流程图

4.2.2 挂载文件系统

实现挂载的重点是如何识别挂载点，在 OopS 中，所有的文件都有一个全局的文件标识 FileIndex。FileIndex 是一个二元组，由文件系统编号和系统内部编号组成。可以通过 File trait 中的 get_index 方法来获得文件的文件标识。

在文件系统被创建的时候，内核会分配给新创建的文件系统一个唯一的文件系统编号。系统内部编号是文件在文件系统内部的编号，只需要保证在该文件系统中，每一个文件的系统内部编号不同即可。比如，在 FAT32 中文件的起始簇号、

ext2 中 inode 号都可以作为系统内部编号由于每个文件的文件系统编号和系统内部编号不可能同时相同，所以可以用 FileIndex 可以唯一的标识一个文件。

虚拟文件系统管理器中有一个挂载表，挂载表的类型是 Map 类型，key 的类型是文件标识 FileIndex，value 的类型是文件系统。

mount 接口会根据 fstype 和 dev 创建一个文件系统然后打开 path 指向的目录，获取这个目录的 FileIndex，将(FileIndex, VFS)作为键值对加入到挂载表上。

如果要判断一个目录是否为挂载点，需要通过 get_index 方法获得目录的 FileIndex，如果挂载表上有键值对的键为该 FileIndex，那么这个目录就是挂载点，挂载在该目录上的文件系统为 FileIndex 对应的 VFS。

在 4.2.1 的 open_path 中，如果在解析路径的时候碰到了挂载点，就会打开挂载的文件系统的根目录访问该挂载点，就是访问挂载的文件系统的根目录，这个目录原本包含的内容被屏蔽了。

umount 会将挂载点从挂载表中删除那么被挂载的目录就会解除屏蔽，能够被正常访问。

4.3 块缓存层

文件系统运行时需要频繁的向存储设备中读写数据一般情况下，直接读写存储设备的时间消耗十分巨大为了避免直接读写存储设备，操作系统可以在内存中划分出一段区域作为块缓存文件系统以读写块缓存的方式间接地读写存储设备块缓存由多个缓存块组成，每个块都对应着存储设备的一个基本存储单元，比如 SD 卡的 sector，所以缓存块的大小需要与存储设备的基本存储单元大小一致在 OopS 中，缓存块的大小为 512B，所以只支持基本存储单元大小为 512B 的块设备。

和 rCore Tutorial 一样，OopS 的缓存块是动态分配的，当需要新增一个块缓存的时候，就向动态内存分配器中获得一块 512B 的内存空间为了避免块缓存占据内存过多，缓存块的数量不能超过缓存块数目最大值 (MAX_CACHE_NUM)。

块缓存层中的核心结构是块缓存管理器 BlockCacheManager 块缓存管理器包含一个队列，队列中存储着存储块块缓存管理器提供 get_block_cache 接口，用来获取指定存储设备中指定块号的缓存块以下是 BlockCacheManager 数据结构

和 `get_block_cache` 接口声明。

```
pub struct BlockCacheManager {
    caches: VecDeque<(Block_id, Device_id, Arc<RwLock<BlockCache>>>>
}

pub fn get_block_cache(
    self,
    block_id: usize,
    block_file: Arc<dyn BlockFile>,
) -> Arc<RwLock<BlockCache>>
```

缓存块数据结构的定义如下：

```
pub struct BlockCache {
    cache: [u8; BLOCK_SIZE],
    block_id: usize,
    block_file: Arc<dyn BlockFile>,
    modified: bool,
}
```

缓存块包含一个长度为 512 的字节数组，用来存储数据因为在某个时刻需要向存储设备中正确的块写回数据，所以需要保存存储设备的引用和块号将字节数组的数据写回到存储设备的时候，只需要调用 `block_file` 的 `write_block` 方法即可。

因为缓存块的个数有限，所以需要将没有被占用的缓存块淘汰，淘汰算法使用的是 LRU 算法判断一个缓存块有没有被占用是通过判断缓存块的 `Arc` 指针的引用计数是否为 1 来实现的。

目前的实现有需要改进的地方目前缓存块存储在一个队列中，通过设备号和块号索引一个缓存块的时候，需要从前向后遍历缓存块队列如果缓存块队列长度过大，遍历操作将会非常的耗时所以可以考虑用 B 树表来实现缓存块管理器，难点在于如何在 B 树上实现 LRU 算法。

4.4 设备管理

每一个设备在 OopS 中都对应着一个对象，这个对象是对驱动程序的抽象创建设备对象的时候会分配一个唯一的设备编号 `DevID`，设备编号能够唯一地标识一个设备对象。

OopS 沿用了 Unix “一切皆文件”的思想，所以 OopS 中有设备文件所以所有的设备文件都需要实现 `File trait` 和 `DeviceFile trait` 在此基础上，字符设备文件需要实现 `CharFile trait`，块设备文件需要实现 `BlockDevice trait` 一般情况下，设备文件都指向一个设备对象通过调用设备文件上的读写接口，可以将读写操作传递给设备对象，即设备的驱动程序，最终传递给硬件设备。

OopS 的设备管理部分还远远没有完成 RustSBI 会获取设备树的信息，并将设备树信息传递给内核但是目前 OopS 还没有从设备树中提取信息。

目前的实现还非常得简陋在 OopS 文件系统初始化的时候，将会创建一个设备文件系统，并将设备文件系统挂载到 `/dev` 上目前设备文件系统中只有 2 个文件：块设备文件 `sda2` 和字符设备文件 `pts` 可以通过 `open` 接口打开设备文件。

设备文件系统还有很多的地方需要完善由于目前对设备树还不太了解，所以这只是一份暂时的计划：内核初始化创建设备文件系统的时候，从设备树中读取设备信息，如果操作系统支持该设备，即实现了该设备的驱动程序那么就创建一个设备对象，将这个文件对象加入到 DevFS 的根目录中。

4.5 FAT32

目前 OopS 实现的 FAT32 是一个简化版的 FAT32，简化的地方包括如下几点：

1. 忽略 `info sector`，不维护 `valid number of free clusters` 等次要信息。
2. 假定文件系统只有 1 个分区。
3. 只使用第一个 FAT 表，并且不维护其它的 FAT 表。
4. 将 BPB 和 EBPB 中一些次要的信息忽略，比如：Volume ID，Version 等。
5. 默认 `bytes_per_sector = 512`，`root_dir_cluster = 2`，否则报错。
6. 目前暂时不支持时间信息（以后会支持）。
7. 目前不支持非 ASCII 码文件名（以后可能会支持）。

FAT32 根据官方标准⁴实现目前实现的 FAT32 还有 BUG 和缺陷，还需要在以后完善本文档不会讲解 FAT32 的具体实现细节而是主要介绍 OopS 实现的 FAT32 的一些特性。

⁴ https://en.wikipedia.org/wiki/Design_of_the_FAT_file_system

4.5.1 文件一致性

文件系统需要保证文件的一致性，否则在特定情况下会造成严重的后果比如：多个进程打开了同一个文件，如果某一个进程删除了该文件，但是另一个进程的文件描述符表中仍然保留有该文件，且不知道该文件已经被删除如果此时向磁盘中的文件中读写数据，就会从原本被删除的文件占有存储单元中读写数据这些数据可能是错误的如果这些存储单元被分配给了新创建的文件，则会造成更严重的错误。

造成以上错误的原因是：内核可以通过多个“入口”来访问和修改同一个存储设备中的文件，比如说上述例子中，进程的文件描述符中有多个文件指向同一个存储设备中的文件为了保证文件的一致性，OopS 的 FAT32 文件系统中有一个文件缓存层，它的核心思想是：**保证存储设备中的文件在内核中最多只有一个“入口”**。这个“入口”就是文件缓存层中的文件缓存文件描述符表中的文件是一个指向文件缓存层中的文件的 Arc 指针可以说，文件缓存为文件描述符表中的文件和存储设备中的文件提供了一个桥梁，如图 11 所示。

文件缓存层是以文件的起始簇号来标识一个文件对文件进行任何访问和修改之前，都需要打开文件在打开文件的时候，先获得这个文件的起始簇号，然后判断这个文件是否在文件缓存层中有一个文件缓存，如果有的话，则返回指向文件缓存的一个指针，否则将文件加入到文件缓存层中，再返回进程的文件描述符指向的是文件缓存层中的文件缓存，而不是直接指向存储设备中的文件这样就能保证存储设备中的文件最多只有一个文件缓存。

由于文件描述符表中的文件是一个指向文件缓存层中的文件的 Arc 指针，所以可以通过判断 Arc 指针的引用计数来判断该文件是否还被其它进程打开如果文件还被其它进程打开，那么就不能删除该文件。

文件缓存层没有限制文件缓存的数量，不过一个文件缓存只包含文件的元信息而进程的数目和进程描述符的大小是有限的，所以文件缓存层并不会占用过多的内存空间。

目前文件缓存层有不少可以改进的地方：文件缓存层是 FAT32 特有的可以让所有的文件系统共享一个文件缓存层，这样就不需要为每一个文件系统实现文件缓存层。

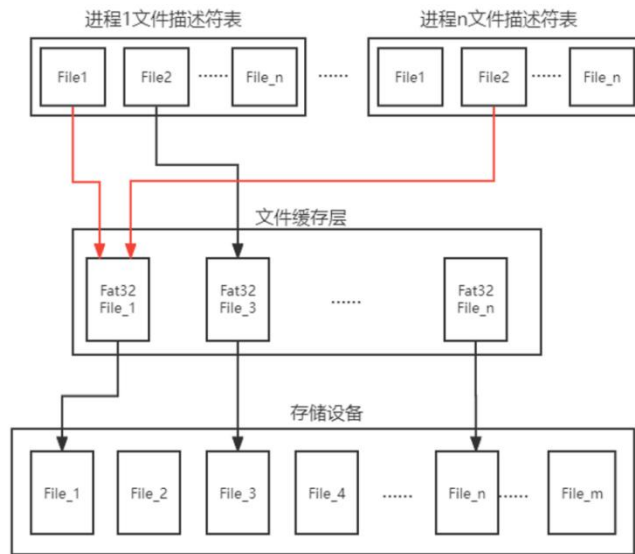


图 11 文件缓存层示意图

4.5.2 空闲簇缓存

在创建文件或增大文件的时候，需要为文件分配空闲簇可以通过遍历 FAT32 表的方式来找到空闲簇不过由于前面的簇一般都被分配出去了，所以需要遍历到 FAT32 表的靠后的位置才能找到空闲簇而遍历 FAT32 表需要读写块缓存，会占用大量的块缓存块，导致大量块缓存块被淘汰出去，造成效率损失。

为了减少遍历 FAT32 表造成的损失，OopS 为 FAT32 实现了空闲簇缓存在初始化的时候，空闲簇缓存会遍历 FAT32 表，把遍历到的空闲簇加入到空闲簇缓存中，直到得到指定数目的空闲簇后则终止遍历。

以后需要获取空闲簇的时候，可以从空闲簇缓存中获取空闲簇，而不用遍历 FAT32 表如果空闲簇缓存中空闲簇用完了，再遍历 FAT32 表，补充空闲簇缓存

目前再将空闲簇添加到空闲簇缓存的时候，并没有在 FAT32 表上进行标记，而是在空闲簇被使用的时候，再在 FAT32 表上进行标记所以每次使用空闲簇的时候，还是需要写一次 FAT32 表其实可以在将空闲簇加入到缓存的时候，就在 FAT32 表中标记簇被占用操作系统关闭的时候，再将缓存中没有使用的空闲簇释放，即将 FAT32 表中的占用标记清除这样可以把多次的写操作集中到一次，节省了时间不过目前内核还没有实现在关闭的时候进行后续操作，所以还没有实现该功能。

5 信号处理

信号系统是进程与进程之间，内核与进程沟通的一种方式，属于进程间通信模块。每个进程的任务控制块中都保存着该进程的信号掩码和信号队列。进程可以向目标进程发送信号，待处理信号会在信号队列中进行排队。每当进程即将从内核态返回用户态时，就会对信号进行处理，直到所有信号处理完毕，进程才会返回到用户态。具体处理过程可以参照图 12 所示流程。

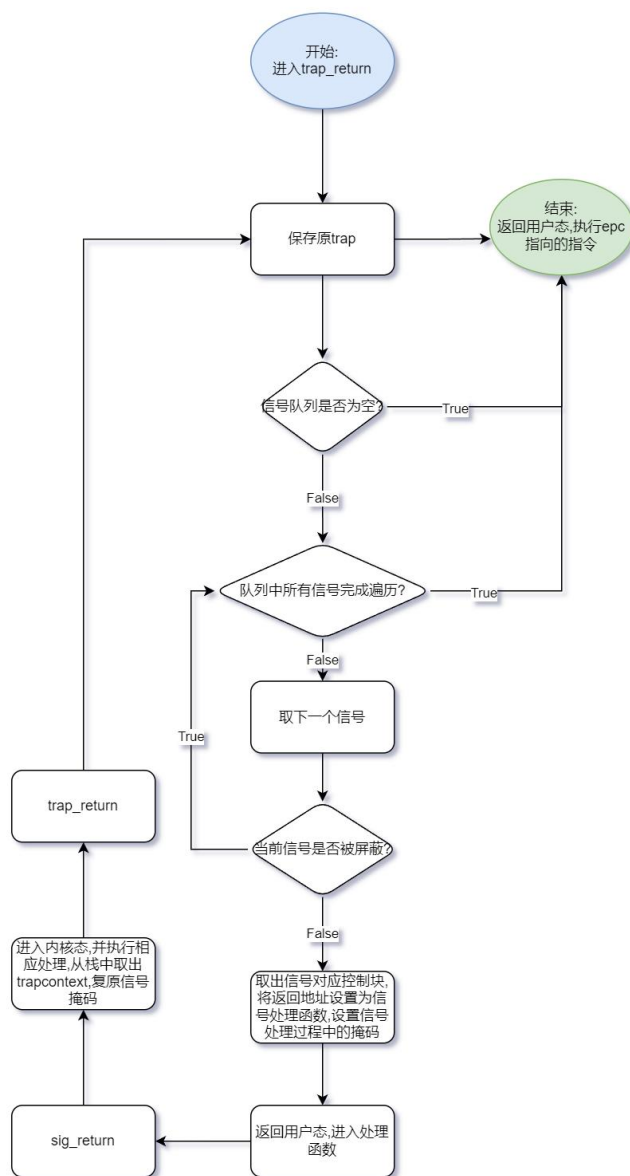


图 12 信号处理流程图

5.1 信号控制块

```
pub struct Sigaction{
    pub sa_handler: usize,      //默认信号处理函数
    pub sa_sigaction: usize,    //自定义信号处理函数
    pub sa_mask: usize,        //信号处理函数执行期间的掩码
    pub sa_flags: u32,          //标志位
    pub sig_info_ptr: usize,
}
```

上图是信号控制块的数据结构，`sa_handler` 是默认信号处理函数的地址，由操作系统负责设置。`sa_sigaction` 则是自定义的信号处理函数地址，由用户程序设置。`sa_mask` 则是信号处理函数执行过程中的进程信号掩码。`sa_flags` 则是有关信号处理行为的标志位，目前只有 `SIG_INFO` 被启用，其作用是表明信号处理函数有参数需要进行入栈处理。`sig_info_ptr` 则是指向参数的指针。

5.2 信号处理

信号处理相当于在正常的退出 `trap` 的过程中插入一段执行用户函数的过程，涉及到对 `Trapframe` 的修改和恢复，因此需要保存原本 `Trapframe` 中的上下文考虑到信号处理函数执行过程中可能需要参数，以及执行过程中处理另一个信号，操作系统设置了信号栈来保存这些上下文。

对信号的处理有两种方式:忽略或执行信号处理函数。当进程准备从内核态返回用户态时，会从信号队列中选择一个未被屏蔽的信号序号，根据该信号的序号得到对应的信号控制块 `Sigaction`。

`Sigaction` 中的 `sa_handler` 指示了该信号将使用何种处理函数。如果 `sa_handler` 的值为 `SIG_DFL` 等操作系统提供的默认处理函数序号，那么操作系统会将 `sepc` 设置为默认处理函数的用户地址;否则会使用 `sa_sigaction` 指示的处理函数地址设置 `sepc`，该地址通常由用户提供设置完成后当进程进入到用户态时就会进入到处理函数之中。

这里会产生一个问题，所有信号处理函数都是在用户态中执行的，操作系统提供的默认处理函数也不例外，因此我们需要使得用户进程能够在用户态下访问

这些默认处理函数。但是默认处理函数部分的代码是跟内核一起编译的，与其他内核代码存在于 `.text` 段中，显然用户进程并不具有该段的访问权限。因此我们需要将这些函数统一放置在一个内存区域中，然后仿照 `trampoline` 的实现方式，在用户页表中将一块特定的虚拟内存地址映射到默认处理函数代码所在的物理内存区域，如图 13 所示。

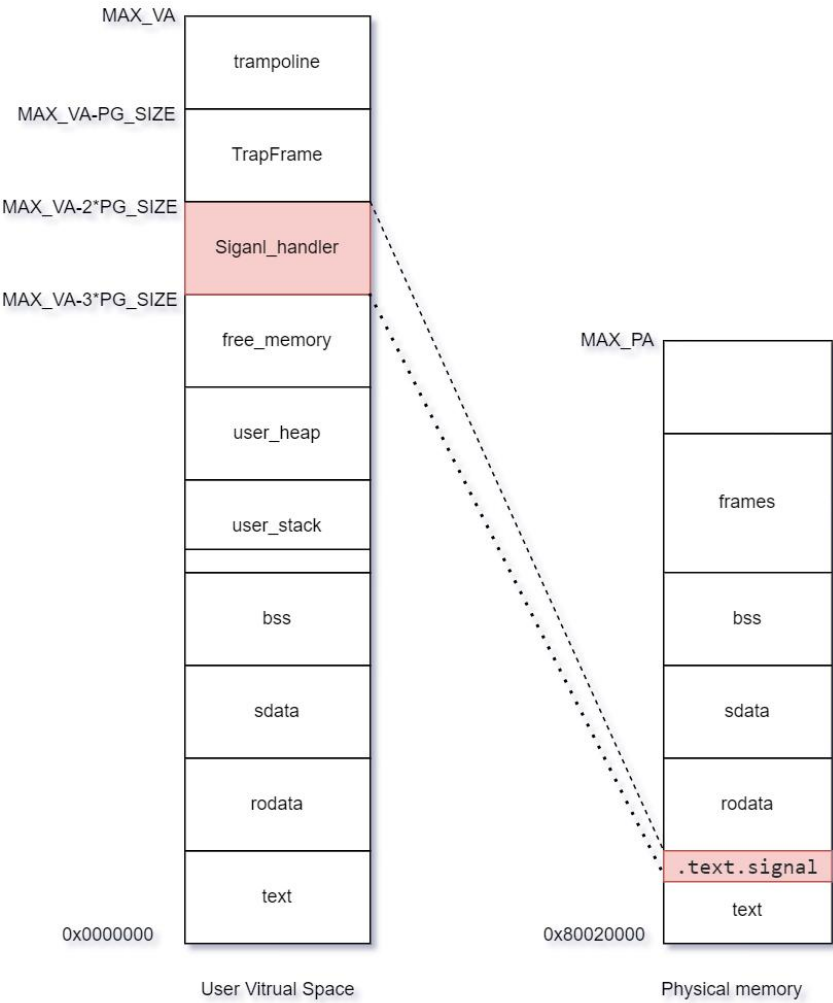


图 13 默认处理函数映射

6 总结与未来计划

6.1 总结

目前我们的进度基本上达到了我们决赛第一阶段的目标：即通过所有测试，较好地支持 busybox。

由于时间关系，还有一些模块还需要完善，特别是页面置换算法和任务的睡眠和唤醒。此外，内核的鲁棒性不足，恶意的用户程序容易导致内核的崩溃。还有一些地方存在未解决的 BUG。需要在接下来的第二阶段进行修补和改善。

6.2 未来计划

目前已经明确的主要计划如下所示：

计划	优先级
支持在多核情况下稳定运行	中
完善页面置换方式	中
重构部分逻辑混乱的代码	高
实现更多的驱动（比如 RTC），充分利用开发板上的硬件设备	低
修改块缓存的实现方式，实现更高效的索引	低
完善任务的阻塞和唤醒流程	高
改进 exec 装载可执行文件到内存的方式	中
检查代码各处 unwrap 的合理性，提高内核的鲁棒性	中

表 9 计划表