



哈爾濱工業大學(深圳)

HARBIN INSTITUTE OF TECHNOLOGY, SHENZHEN

Phoenix

设计文档

参赛队名 Phoenix

队伍成员 陈睿玮、石全、王华杰

指导老师 夏文、仇洁婷

2024 年 5 月 30 日

摘要

Phoenix 是 Rust 编写的基于 RISC-V 的宏内核操作系统，结合 Rust 语言异步机制，采用无栈协程架构，支持多核运行，使用模块化的方式组织内核代码，致力于为用户程序提供完善的系统调用功能和高效的内核性能。

于 2024-05-31 结束初赛阶段时，满分通过所有初赛测试用例。所属 VisionFive 2 赛道的排行榜如下图所示：

介绍

赛题

排行榜

2024年操作系统赛-内核实现全国赛初赛-VisionFive 2

2024年操作系统赛-内核实现全国赛初赛-2K1000

2024年操作系统赛-内核实现全国赛决赛-VisionFive 2

2024年操作系统赛-内核实现全国赛决赛-2K1000

比赛提交到排行榜更新有20秒左右的延迟

#	用户名	队伍	提交次数(ASC)	最后提交时间(ASC)	rank
1	T202410700992507	NPUcore-无所畏惧/ 西安理工大学	2	2024-05-05 11:40:06	102.0000
2	T202410487992456	cabbageOS/ 华中科技大学	3	2024-05-05 20:37:55	102.0000
3	T202418123993075	Phoenix/ 哈尔滨工业大学（深圳）	3	2024-05-08 19:36:27	102.0000
4	T202410359992729	小混子队/ 合肥工业大学	5	2024-05-07 00:03:35	102.0000
5	T202410336992584	Pantheon/ 杭州电子科技大学	6	2024-05-08 11:11:57	102.0000
6	T202410183992516	10183_BOOT/ 吉林大学	7	2024-05-07 17:14:15	102.0000
7	T202410288992519	月光加冕贝拉拉/ 南京理工大学	9	2024-04-14 21:01:15	102.0000
8	T202410214992509	NPUcore-whoami/ 哈尔滨理工大学	9	2024-05-03 21:06:11	102.0000
9	T202410464992703	海底小纵队/ 河南科技大学	10	2024-05-13 05:05:50	102.0000
10	T202410559992716	双层吉士比板烧好吃/ 暨南大学	10	2024-05-15 15:25:48	102.0000

表 0-1: 模块完成情况

模块	完成情况
无栈协程	基于全局队列实现的调度器，完善的辅助 Future 支持
进程管理	统一的进程线程抽象，可以细粒度划分进程共享的资源
内存管理	实现基本的内存管理功能。使用懒分配和 Copy-on-Write 优化策略
文件系统	基于 Linux 设计的虚拟文件系统。实现页缓存加速文件读写，实现 Dentry 缓存加速路径查找。使用开源 <code>rust-fatfs</code> 库提供对 FAT32 文件系统的支持

信号机制	支持用户自定义信号处理例程，有较为完善的信号系统，与内核其他异步设施无缝衔接
------	----------------------------------------

目 录

摘要	I
1 概述	1
1.1 Phoenix 介绍	1
1.2 Phoenix 整体架构	1
2 进程管理	2
2.1 任务调度	2
2.2 任务控制块	9
2.3 中断与异常	14
3 内存管理	19
3.1 物理内存管理	19
3.2 地址空间	20
3.3 缺页异常处理	23
4 文件系统	25
4.1 虚拟文件系统	25
4.2 FAT32 文件系统	35
5 进程间通信	36
5.1 信号机制	36
6 总结与展望	38
6.1 工作总结	38
6.2 经验总结	38
6.3 未来计划	39

1 概述

1.1 Phoenix 介绍

Phoenix 是 Rust 编写的基于 RISC-V 的多核异步宏内核模块化操作系统，使用了以 Future 抽象为代表的无栈协程异步模型，提供统一的线程和进程表示，细粒度的资源共享以及段式地址空间管理，拥有基于 Linux 设计的完善的虚拟文件系统以及基础的信号机制。

Phoenix 使用模块化的方式组织内核代码，将宏内核各个部分独立成不同的模块。通过明确模块之间的接口，减少模块间的相互依赖和干扰，从而使内核代码复用成为可能，目标是为以后参与操作系统竞赛的同学提供功能与性能兼备的模块。

Phoenix 致力于实现高质量的代码，使内核兼具完善的功能和高效的性能。功能方面，Phoenix 通过了初赛所有系统调用测试用例。性能方面，使用懒分配和写时复制机制优化 `execve` 执行速度，使用页缓存加速文件读写，减少磁盘 IO 访问次数。

Phoenix 严格遵循 System Calls Manual 手册进行开发，确保实现的所有系统调用符合 Linux raw syscall 规范，为运行在 Phoenix 的用户态程序提供了完备可靠的系统调用支持。

1.2 Phoenix 整体架构

```
.
├── arch/                # 平台相关的包装函数与启动函数
├── config/              # 配置常量
├── crates/              # 自己编写的功能单一的库
│   ├── async_utils/    # 异步工具
│   └── recycle_allocator/ # ID 分配器
├── docs/                # 文档
├── driver/              # 驱动模块
├── kernel/              # 内核
│   └── src/
│       ├── ipc/         # 进程间通信机制
│       ├── mm/          # 内存管理
│       ├── processor/   # 多核心管理
│       ├── syscall/     # 系统调用
│       ├── task/        # 进程管理
│       ├── trap/        # 异常处理
│       ├── utils/       # 工具
│       ├── boot.rs      # 内核启动通用函数
│       ├── impls.rs     # 模块接口实现
│       ├── link_app.asm
│       ├── loader.rs
│       └── main.rs      # 主函数
```

```
├── panic.rs
├── trampoline.asm # 信号跳板代码
├── build.rs
├── Cargo.toml
├── linker.ld      # 链接脚本
├── Makefile
├── modules/      # 内核各个模块
│   ├── executor/ # 异步调度器
│   ├── fat32/    # FAT32 文件系统支持
│   ├── futex/    # futex 机制
│   ├── logging/  # 日志系统
│   ├── memory/   # 基础内存模块
│   ├── signal/   # 基础信号模块
│   ├── sync/     # 同步原语
│   ├── systype/  # 系统调用类型
│   ├── time/     # 时间模块
│   ├── timer/    # 定时器模块
│   ├── vfs/      # 虚拟文件系统模块
│   └── vfs-core/ # 虚拟文件系统接口
├── testcase/     # 测试用例
├── third-party/  # 第三方库
│   └── vendor/   # Rust 库缓存
├── user/         # 用户程序
├── Cargo.lock
├── Cargo.toml
├── Dockerfile
├── LICENSE
├── Makefile
├── README.md
├── rustfmt.toml
└── rust-toolchain.toml
```

2 进程管理

2.1 任务调度

2.1.1 为什么选择无栈协程

协程是一种比线程更加轻量级的并发单位, 允许在执行过程中挂起并稍后恢复, 从而使得单个线程可以处理多个任务。无栈协程和有栈协程的主要区别在于它们的上下文管理方式。

- **有栈协程 (Stackful Coroutine)** : 每个协程都有自己的栈, 这样可以在协程内部随时进行函数调用和上下文切换, 在大规模并发时会带来较大的内存开销。在 RISC-V 架构的操作系统中, 有栈协程需要在栈中保存 `s0-s11`、`ra`、`sp` 这 14 个寄存器。

- **无栈协程 (Stackless Coroutine)**：协程没有独立的栈，而是依赖于状态机来管理上下文，每次协程挂起时需要保存执行的状态和位置，下次恢复时从该位置继续执行。无栈协程非常轻量，适合大规模并发。但是不适合深度嵌套的函数调用，适合用在状态转移和事件驱动的场景中。

在操作系统中，每一个线程都有其上下文环境，当发生任务调度时，任务切换其实就是上下文切换。任务是由内核来管理和调度的，任务的切换必须发生在内核态。

在 Linux 系统中，一个进程的上下文可以分为以下三个部分：

1. 用户级上下文：包括了进程在用户态下的运行状态和资源。如用户堆栈、全局变量和静态变量等等
2. 寄存器上下文：如通用寄存器、栈指针
3. 系统级上下文：如进程控制块、内核栈、页表

当 Linux 发生进程调度时，必须对上面的全部信息进行切换。通过调用函数的方法 (Linux 内核中 `switch_to` 函数) 进行上下文切换。

但是，Phoenix 使用无栈协程架构，所有任务共享同一个内核栈，任务调度时不需要切换内核栈，因此调度的开销实际上小于 Linux。Phoenix 的任务切换发生在 `await` 处，当 `poll` 轮询返回 `pending` 时不断退出本层 `async` 函数返回到上一层的 `async` 函数，直到回到调度器，接着调度器调度到下一个任务，并根据堆上的信息重新生成新任务的函数调用栈，来到上次该任务 `await` 的地方继续 `poll` 轮询。

我们选择无栈协程，除了其任务切换开销小以外，很大一部分原因也与 Rust 语言本身的异步编程模型有关。

Rust 的所有权系统通过编译时检查保证内存安全，防止数据竞争、空指针和悬挂指针等问题。Rust 的所有权和借用检查，能够更好地保证协程在并发执行时的安全性，确保资源在使用后会被正确释放，更加高效地利用系统资源。

另外，Rust 内置的异步编程模型通过 `async` 和 `await` 关键字支持无栈协程，这种语法糖使得编写和使用无栈协程变得更加简洁和直观。每个 `async` 函数在编译时会被转换为状态机，自动管理状态的保存和恢复。无栈协程符合 Rust 所追求的”零成本抽象”，状态机的转换和上下文切换在编译期确定，运行时开销极小。在敏捷开发过程中，这种语法糖能极大提高操作系统的开发效率。

此外，Rust 活跃的社区和丰富的生态系统也提供了大量用于异步编程的库和工具。Phoenix 在无标准库 `no-std` 的环境下开发，因此可以利用 `async-task` 库提供的关于 `Future` 的抽象实现自己的全局任务队列调度器。

2.1.2 Rust 无栈协程原理

2.1.2.1 将异步函数编译成状态机

2019 年底 Rust 推出了支持异步编程的 `async` 和 `await` 关键字，极大地简化了异步函数的定义和调用，并在无栈协程调度中发挥了重要作用。

- `async` 关键字用于定义异步函数、异步块或异步闭包。当使用 `async` 标记一个函数时，该函数不再同步返回结果，而是返回一个实现了 `Future` trait 的对象。在编译时，带有 `async` 关键字的代码会被转换为一个状态机。这个状态机会在每个 `await` 点保存当前状态，并在未来的某个时刻恢复执行。
- `await` 关键字用于等待一个实现了 `Future` trait 的对象完成，并获取其结果。当执行到 `await` 时，如果 `Future` 尚未完成，当前任务会挂起，并允许其它任务继续执行。每当 `Future` 完成时，使用 `await` 的代码会从挂起点恢复执行，状态机会从之前保存的状态继续运行。

下面这个例子演示了这两个关键字的使用：

```
async fn async_function() {
    println!("First part of the function");
    async_operation().await;
    println!("Second part of the function");
}

async fn async_operation() {
    // 模拟异步操作
    println!("Performing async operation");
}
```

Rust 编译器会将上面这段代码编译成一个实现 `Future` trait 的状态机。这个状态机记录了函数执行的当前位置和需要恢复的状态：

```
enum AsyncState {
    FirstPart,
    AwaitingOperation,
    SecondPart,
    Done,
```



```

}

struct AsyncFunction {
    state: AsyncState,
}

impl AsyncFunction {
    fn new() → Self {
        AsyncFunction {
            state: AsyncState::FirstPart,
        }
    }
}

impl Future for AsyncFunction {
    type Output = ();

    fn poll(mut self: Pin<&mut Self>, _cx: &mut Context<'_>) →
Poll<Self::Output> {
        loop {
            match self.state {
                AsyncState::FirstPart ⇒ {
                    println!("First part of the function");
                    self.state = AsyncState::AwaitingOperation;
                }
                AsyncState::AwaitingOperation ⇒ {
                    println!("Performing async operation");
                    self.state = AsyncState::SecondPart;
                    return Poll::Pending;
                }
                AsyncState::SecondPart ⇒ {
                    println!("Second part of the function");
                    self.state = AsyncState::Done;
                }
                AsyncState::Done ⇒ {
                    return Poll::Ready(());
                }
            }
        }
    }
}

```

上面这段代码涉及到如下几个关键的概念：

- **Future Trait**: 我们可以将 `Future` 理解为一个异步计算的抽象，它的 `poll` 方法会尝试推进计算。`Future` trait 类似于一个任务的描述，告诉你这个任务将在未来完成，并且可以查询其状态。

- **Poll 枚举**: 表示 `Future` 的当前状态（注意 `Poll` 枚举不是 `poll` 方法，两者是两个概念）：
 - `Poll::Pending`: 异步操作尚未完成，需要等待。
 - `Poll::Ready(T)`: 异步操作已经完成，可以返回结果。
- **Pin**: `poll` 方法的 `self` 参数被包裹在 `Pin<&mut Self>` 中，确保 `AsyncFunction` 的内存位置固定，不会被移动，从而保证安全。
- **Context**: 包含了执行 `Future` 所需的上下文信息，例如 `Waker`，它允许 `Future` 在准备好继续执行时通知执行器。`Context` 使得 `Future` 可以与外部世界交互，知道何时需要继续执行。

2.1.2.2 Executor 和 Reactor

Executor（执行器）是一个运行时组件，它负责管理和调度异步任务。Executor 的主要职责包括：

1. **运行 Futures**: Executor 会调用 Futures 的 `poll` 方法来推进它们的执行。
2. **任务队列**: Executor 维护一个任务队列，存放所有准备好执行的任务。
3. **循环调度**: Executor 在循环中轮询任务队列，逐个执行任务。当任务返回 `Poll::Ready` 时，表示任务已经完成；当返回 `Poll::Pending` 时，表示任务尚未完成，需要等待某些条件。

Reactor（反应器）是另一个运行时组件，负责管理和处理异步 I/O 操作或其他需要等待的事件。Reactor 的主要职责包括：

1. **等待事件**: Reactor 等待异步操作的完成，例如网络 I/O、文件 I/O、计时器等。
2. **唤醒任务**: 当异步操作完成时，Reactor 会唤醒相应的任务，并将它们重新放入 Executor 的任务队列，以便再次执行。

2.1.3 任务调度队列与执行器

得益于 `async_task` 对 `Future` 提供的便捷的抽象，Phoenix 自己实现了全局任务队列调度器。

为了存储待执行的协程任务，Phoenix 定义了一个全局的任务队列 `TASK_QUEUE`，各个 CPU 可以从该任务队列中获取任务执行。`TaskQueue` 是一个使用 `SpinNoIrqLock` 保护一个双端队列（`VecDeque`），`Runnable` 为可调度的任务：

```
struct TaskQueue {  
    queue: SpinNoIrqLock<VecDeque<Runnable>>,  
}
```

`spawn` 函数用于创建并启动一个新的异步任务。它接收一个实现了 `Future` 的对象，并将其转换为一个可以调度和执行的任务并添加到任务队列中。

```
pub fn spawn<F>(future: F) → (Runnable, Task<F::Output>)  
where  
    F: Future + Send + 'static,  
    F::Output: Send + 'static,  
{  
    let schedule = move |runnable: Runnable, info: ScheduleInfo| {  
        if info.woken_while_running {  
            // i.e. `yield_now()`  
            TASK_QUEUE.push(runnable);  
        } else {  
            // i.e. woken up by some signal  
            TASK_QUEUE.push_preempt(runnable);  
        }  
    };  
    async_task::spawn(future, WithInfo(schedule))  
}
```

任务的调度策略如下：

- 如果任务在运行时被唤醒（`woken_while_running`），将任务放到队列尾部，按照先入先出的顺序执行。这种情况通常发生在任务调用 `yield_now()` 主动让出处理器时。
- 如果任务在睡眠中被唤醒，将任务放到队列头部，以确保优先执行。这种情况通常发生在任务已经休眠正在等待某个事件，此时等待的事件发生时将该任务唤醒。也可能发生在任务在处于 `Interruptable` 状态，此时收到了一个未阻塞的信号并且触发了 `sighandler` 函数，将任务唤醒。

`run_until_idle` 函数负责从任务队列中取出并执行任务，直到队列为空。Phoenix 支持多核并发，每一个 CPU 核心都在一个永不停止的 loop 循环中持续调用该函数

```
pub fn run_until_idle() {  
    while let Some(task) = TASK_QUEUE.fetch() {  
        task.run();  
    }  
}
```

2.1.4 异步任务上下文切换

每一个任务都有独立的上下文，在进行上下文切换时，Phoenix 需要切换任务控制块、`sum_cnt` 计数器（用来维护内核态对用户内存的访问限制）、页表。

在 RISC-V 中，`SUM` 标志位的作用是控制在内核态（特权级 S 模式）下，是否允许内核访问用户态（U 模式）的内存。Phoenix 在内核态处理系统调用访问用户态指针时需要维护 `sum_cnt` 计数器，当计数器大于 0，`SUM` 标志位为 1，此时内核可以安全地读取或写入用户态缓冲区。内核完成对用户态缓冲区的操作后，`sum_cnt` 计数器减一。当减到 0 时需要恢复 `SUM` 位为 0 以避免意外访问用户态内存。

实际上，上下文的切换可以与 Rust 异步无栈协程完美结合。在每一个用户异步任务的最外层都套了一层 `UserTaskFuture`：

```
pub struct UserTaskFuture<F: Future + Send + 'static> {
    task: Arc<Task>,
    env: EnvContext,
    future: F,
}

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 = hart::local_hart();
        hart.enter_user_task_switch(&mut this.task, &mut this.env);
        let ret = unsafe { Pin::new_unchecked(&mut this.future).poll(cx) };
        hart.leave_user_task_switch(&mut this.env);
        ret
    }
}
```

`UserTaskFuture` 的 `EnvContext` 字段存放了 `sum_cnt`，`task` 字段存储了任务控制块，页表保存在任务控制块内部。`enter_user_task_switch` 函数将当前 CPU 的上下文切换为即将执行的 `task`，执行完毕后，`leave_user_task_switch` 则将页表切换回内核页表，任务调度器将会轮询下一个 `UserTaskFuture`。注意切换任务的时候需要关中断，否则可能产生死锁。

`UserTaskFuture` 中的 `future` 字段对于每一个任务而言都是同一个异步函数 `task_loop`，也就是在 `task` 的生命周期内不断进行返回用户态、陷入内核态、信号处理等过程，直到 `task` 结束，状态变为 `Zombie` 后退出循环，交由 `do_exit` 函数进行处理。

2.2 任务控制块

进程是操作系统中资源分配的基本单位。每个进程都有自己独立的地址空间和资源，如内存、文件描述符等。线程是操作系统中 CPU 调度的基本单位。线程共享所在进程的地址空间和资源，但有独立的执行上下文。

在调研其他队的设计时我们发现，很多队都将进程和线程分开设计，分别使用 `Process` 和 `Thread` 结构体表示。但其实在 Linux 内核中，并没有严格地区分进程和线程，而是通过一组统一的 API 来操作任务，例如 `sys_clone` 系统调用通过不同的 `flags` 组合创建共享不同资源的新任务，因此进程和线程的创建本质上是类似的。统一使用 `Task` 结构体，可以通过标志位决定任务的具体属性，而不需要区分进程结构体和线程结构体。比如，设置 `CLONE_VM` 标志位时，新任务共享父任务的地址空间，这可以通过 `Task` 结构体中的 `memory_space` 字段来实现共享。同样，设置 `CLONE_THREAD` 标志位时，新任务加入父任务的线程组，这可以通过 `Task` 结构体中的 `ThreadGroup` 字段来实现管理。

2.2.1 Task 结构体设计

基于上述分析，我们对进程和线程统一使用 `Task` 结构体表示，其具体设计如下：

```
pub struct Task {
    // Immutable
    /// Task identifier handle.
    tid: TidHandle,
    /// Weak reference to the leader task. `None` if this task is the leader.
    leader: Option<Weak<Task>>,
    /// Indicates if the task is the leader of its thread group.
    is_leader: bool,

    // Mutable
    /// Indicates if the task is a zombie. Protected by a spin lock due to
    /// potential access by other tasks.
    state: SpinNoIrqLock<TaskState>,
    /// The address space of the process.
    memory_space: Shared<MemorySpace>,
    /// Map of start address of shared memory areas to their keys in the
    /// shared memory manager.
    shm_ids: Shared<BTreeMap<VirtAddr, usize>>,
    /// Parent task.
```

```

parent: Shared<Option<Weak<Task>>>,
/// Child tasks.
children: Shared<BTreeMap<Tid, Arc<Task>>>,
/// Exit code of the task.
exit_code: AtomicI32,
/// Trap context for the task.
trap_context: SyncUnsafeCell<TrapContext>,
/// Waker to add the task back to the scheduler.
waker: SyncUnsafeCell<Option<Waker>>,
/// Thread group containing this task.
thread_group: Shared<ThreadGroup>,
/// File descriptor table.
fd_table: Shared<FdTable>,
/// Current working directory.
cwd: Shared<Arc<dyn Dentry>>,
/// Pending signals for the task.
sig_pending: SpinNoIrqLock<SigPending>,
/// Signal handlers. Shared if `CLONE_SIGHAND` flag is set or among
threads in the same group.
sig_handlers: Shared<SigHandlers>,
/// Signal mask indicating which signals are blocked.
sig_mask: SyncUnsafeCell<SigSet>,
/// Optional signal stack for the task, settable via `sys_signalstack`.
sig_stack: SyncUnsafeCell<Option<SignalStack>>,
/// Pointer to the user context for signal handling.
sig_ucontext_ptr: AtomicUsize,
/// Statistics for task execution times.
time_stat: SyncUnsafeCell<TaskTimeStat>,
/// Interval timers for the task.
itimers: Shared<[ITimer; 3]>,
/// Futexes used by the task.
futexes: Shared<Futexes>,
/// Address to store the task identifier.
tid_address: SyncUnsafeCell<TidAddress>,
/// Allowed CPUs for the task.
cpus_allowed: SyncUnsafeCell<CpuMask>,
}

```

下面介绍一下 `Task` 结构体各个字段的含义。

首先 `Task` 结构体被大致划分为不可变（Immutable）和可变（Mutable）两部分：

- **不可变部分**：不可变字段在任务创建后不会改变，因此可以保证数据的一致性。这些字段通常包括任务的基本标识信息，如 `tid`（任务 ID）和 `is_leader`（是否为线程组的主线程）。由于这些字段不会改变，多线程环境下访问这些字段时不需要加锁，可以提高访问效率和安全性。

字段名	含义
-----	----

<code>tid</code>	任务的唯一标识符（任务 ID）
<code>leader</code>	对主线程任务的弱引用。如果该任务是线程组的主线程，则为 <code>None</code>
<code>is_leader</code>	标识任务是否是线程组的主线程

- **可变部分**：可变字段涉及任务的状态和资源管理，如 `state`（任务状态）、`memory_space`（内存空间）和 `children`（子任务）。这些字段在任务的生命周期内可能会改变，需要使用锁机制来保证线程安全。

字段名	含义
<code>state</code>	任务的当前状态（例如，运行中、等待中、僵尸状态等），由自旋锁保护以确保线程安全
<code>memory_space</code>	任务的地址空间
<code>shm_ids</code>	共享内存区域起始地址到共享内存管理器中的键的映射
<code>parent</code>	父任务
<code>children</code>	子任务
<code>exit_code</code>	任务的退出代码
<code>trap_context</code>	任务的陷阱上下文
<code>waker</code>	用于将任务重新添加到调度器的唤醒器
<code>thread_group</code>	包含该任务的线程组
<code>fd_table</code>	文件描述符表
<code>cwd</code>	当前工作目录
<code>sig_pending</code>	任务的挂起信号
<code>sig_handlers</code>	信号处理程序表。如果设置了 <code>CLONE_SIGHAND</code> 标志或者任务在同一线程组中，则这些处理程序是共享的
<code>sig_mask</code>	信号屏蔽，用于指示哪些信号被阻塞
<code>sig_stack</code>	任务的可选信号栈，通过 <code>sys_signalstack</code> 设置

<code>sig_ucontext_ptr</code>	信号处理的用户上下文指针
<code>time_stat</code>	任务执行时间的统计信息
<code>itimers</code>	任务的间隔定时器
<code>futexes</code>	任务使用的轻量级锁
<code>tid_address</code>	存储任务标识符的地址
<code>cpus_allowed</code>	任务允许运行的 CPU 集合

对于 `Task` 的可变部分，我们也进行了巧妙设计，调研其他队时我们发现，很多队都用了 `Inner` 结构体存储可变字段，并在外面加上自旋锁的方式，此种方式虽然能在并发时保证数据的安全性，但是用一把大锁来锁定可变字段并不高效。因此，我们根据可变字段是线程内部独占还是在线程之间共享，分别使用 `SyncUnsafeCell` 和 `Shared<T>` (即 `Arc<SpinNoIrqLock<T>>`) 对可变字段进行包裹，并通过谨慎使用，保证了并发安全性：

- `SyncUnsafeCell<T>`：对于线程独有的字段使用 `SyncUnsafeCell` 包裹。`SyncUnsafeCell` 是一种可以在线程间共享的 `UnsafeCell`。虽然 `UnsafeCell` 本身不实现 `Sync`，`SyncUnsafeCell` 可以在 `T` 实现 `Sync` 的情况下实现 `Sync`。这使得它可以在多线程环境中共享，并且仍然允许内部可变性。
- `Arc<SpinNoIrqLock<T>>`：对同一进程内线程共享的字段使用。`SpinNoIrqLock` 是一种自旋锁，适用于短时间的锁定操作。它不会引发中断，因此适用于操作系统内核中需要快速响应的部分。使用 `SpinNoIrqLock` 包裹的字段可以在并发访问时进行保护，避免数据竞争和不一致。`Arc`（原子引用计数）允许多个所有者共享同一个数据，同时维护共享数据的引用计数，以确保数据在最后一个引用被丢弃时才被释放。

在 `Task` 结构体中使用 `Shared<T>` 和 `SyncUnsafeCell<T>` 包裹可变字段，可以有效地实现内部可变性和细粒度锁定。这种设计允许在多个任务之间安全地共享和修改数据，提高了系统的并发性和性能，同时保证了线程安全性。通过这些机制，操作系统能够更加高效地管理任务和资源。

2.2.2 ThreadGroup 与进程线程的关系

在 Linux 中，线程和进程的关系比一些传统操作系统更加紧密。Linux 使用 `task_struct` 数据结构来表示线程和进程，线程和进程在内核中的表示实际上是一样的。一个线程或进程都是一个 `task_struct` 实例。进程实际上被抽象成了线程组，其主要由线程组的主线程进行管理。线程组是由同一个进程创建的所有线程组成的组。每个线程组有一个主线程（通常是创建线程的进程），其 `TID`（线程标示符）是线程组的 `TGID`（线程组标识符）。从系统角度来看，线程组的主线程表示整个进程，而每个线程的 `TID` 是其在进程中的标识。

Phoenix 效仿 Linux，对进程和线程用 `Task` 结构体进行统一表示，`ThreadGroup` 结构体用于管理属于同一进程的线程组。通过将多个线程归属于一个 `ThreadGroup`，可以方便地管理和调度同一进程中的所有线程。`Task` 结构体和 `ThreadGroup` 结构体的结合，实现了统一的进程和线程管理。

```
pub struct ThreadGroup {
    members: BTreeMap<Tid, Weak<Task>>,
}
```

`ThreadGroup` 结构体包含一个 `BTreeMap`，该映射将线程的唯一标识符（`Tid`）映射到对 `Task` 的弱引用（`Weak<Task>`）。当新创建一个进程时，`ThreadGroup` 仅有进程本身这个成员，`Task` 的 `is_leader` 字段设为 `True`。当 `sys_clone` 设置 `CLONE_THREAD` 标志位时，将新创建的 `Task` 添加入 `ThreadGroup`，新 `Task` 的 `is_leader` 字段设为 `False`。

2.2.3 任务的状态

在调研其他操作系统时，我们发现部分往届作品如 Titanix 只是简单区分了 `Running` 和 `Zombie` 状态，并不支持当 `SIGSTOP` 信号到来时进程暂停执行，此外，任务在阻塞的过程中能否被信号打断也支持得并不是很好，Linux 关于信号的手册中有提到，部分系统调用在阻塞等待的过程中可以被信号打断停止执行，返回 `EINTR` 错误，如果打断系统调用的信号的 `flag` 标志中含有 `SA_RESTART` 可以重启系统调用。基于以上考虑，我们将 `Task` 的状态分为以下五种：

状态	含义
<code>Running</code>	正在运行或准备运行的任务。此状态下，任务占用 CPU，执行其代码。

Zombie	任务已终止，但其进程控制块 (PCB) 仍然存在，以便父进程可以读取其退出状态。
Stopped	任务已停止运行，通常是由于接收到停止信号(如 SIGSTOP)。可以通过特定信号（如 SIGCONT ）恢复运行。
Interruptable	任务处于可中断的等待状态，等待某个事件（如 I/O 操作完成或资源释放）。此状态下，任务可以被信号中断并唤醒。
UnInterruptable	任务处于不可中断的等待状态，等待某个事件的发生。此状态下，任务不会被信号中断，以确保某些关键操作的完整性和原子性。

进程间的状态转换情况如下：

1. **Running ↔ Interruptable**: 当任务需要等待某个事件时，从 **Running** 转换到 **Interruptable** 状态。例如，在实现 `sys_wait4` 系统调用时，**Task** 调用 `suspend_now()` 将自己从任务调度队列中移除，进入 **Interruptable** 状态，即可以被信号中断，如果是等待的子进程退出，返回子进程的 `pid`，如果是被信号中断，返回 `EINTR` 错误。之后从 **Interruptable** 状态恢复为 **Running**
2. **Running ↔ UnInterruptable**: 当任务需要等待某个关键事件且不希望被信号中断时，从 **Running** 转换到 **UnInterruptable** 状态。当等待的事件发生时，恢复到 **Running** 状态
3. **Running → Zombie**: 当任务执行结束并退出时，从 **Running** 转换到 **Zombie** 状态。
4. **Running → Stopped**: 当任务接收到停止信号（如 **SIGSTOP**）时，从 **Running** 转换到 **Stopped** 状态。
5. **Stopped → Running**: 当任务接收到继续信号（如 **SIGCONT**）时，从 **Stopped** 转换回 **Running** 状态。

2.3 中断与异常

在 Linux 操作系统中，Linux 内核使用了 Page Table Isolation (PTI) 机制，内核态和用户态的页表相互独立，当系统运行在用户态时，仅使用用户页表，这个页表只包含极少量必

须的内核数据，如用于进入和退出内核的函数和中断描述符表 (IDT)。当系统进入内核态（通过系统调用、中断或异常）时，切换到完整的内核页表。这种设计的主要优点包括：

1. **提高安全性**：独立的页表可以有效防止侧信道攻击，使得用户态程序无法访问敏感的内核数据。
2. **保护内核空间**：即使攻击者能够突破用户态的防线，也无法直接访问或利用内核地址空间。

然而，这种设计也带来了额外的开销：

1. **内存使用增加**：每个进程需要维护两套页表，增加了内存消耗。
2. **性能开销**：每次从用户态切换到内核态（或反之）时，都需要切换页表，这涉及到刷新 TLB (Translation Lookaside Buffer)，增加了系统调用和中断处理的成本

Phoenix 采用了用户态与内核态共用一个页表的策略，共用一个页表避免了在用户态和内核态之间切换时刷新 TLB 的开销，从而提升了系统性能，并且简化了页表管理的实现和维护。

在 Phoenix 中，`Task` 的生命周期就是在执行下面这个函数：

```
pub async fn task_loop(task: Arc<Task>) {
    *task.waker() = Some(take_waker().await);
    loop {
        trap::user_trap::trap_return(&task);
        match task.state() {
            Zombie => break,
            Stopped => suspend_now().await,
            _ => {}
        }
        trap::user_trap::trap_handler(&task).await;
        match task.state() {
            Zombie => break,
            Stopped => suspend_now().await,
            _ => {}
        }
        task.update_itimers();
        do_signal().expect("do signal error");
    }
    task.do_exit();
}
```

即 `Task` 先创建，然后执行 `trap_return` 返回用户态、陷入内核态由 `trap_handler` 处理中断和异常、`do_signal` 执行信号处理函数，当 `Task` 被设置为 `Zombie` 状态时，回收部分资源，在 `Wait4` 系统调用时彻底回收 `Task` 资源。

在 Phoenix 的内核态用户态切换中，`TrapContext` 是一个关键的数据结构，用于保存从用户态切换到内核态，以及从内核态切换回用户态时需要保存和恢复的上下文信息。这个结构体的设计保证了用户态和内核态之间的切换能够正确地进行，不会丢失任何重要的状态信息。

```
pub struct TrapContext {
    // User to kernel should save:
    pub user_x: [usize; 32],
    pub sstatus: Sstatus, // 32
    pub sepc: usize,      // 33
    // Kernel to user should save:
    pub kernel_sp: usize, // 34
    pub kernel_ra: usize, // 35
    pub kernel_s: [usize; 12], // 36 - 47
    pub kernel_fp: usize, // 48
    pub kernel_tp: usize, // 49
    pub user_fx: UserFloatContext,
}
```

字段名称	说明	用户态到 内核态保 存	内核态到 用户态保 存
<code>user_x</code>	保存用户态的通用寄存器，包含了 RISC-V 架构中的所有 32 个寄存器的值。	是	否
<code>sstatus</code>	保存状态寄存器（Status Register）的值，这个寄存器包含了用户态和内核态的各种状态标志。	是	否
<code>sepc</code>	保存异常程序计数器（Exception Program Counter），即发生异常或中断时的用户程序的 PC 值。	是	否
<code>kernel_sp</code>	保存内核态的栈指针，用于在返回用户态时恢复内核栈的状态。	否	是

<code>kernel_ra</code>	保存内核态的返回地址寄存器 (Return Address Register)，用于在返回用户态时恢复内核的返回地址。	否	是
<code>kernel_s</code>	保存内核态的保存寄存器 (Saved Registers)，这些寄存器在上下文切换时需要保存和恢复。	否	是
<code>kernel_fp</code>	保存内核态的帧指针 (Frame Pointer)，用于在返回用户态时恢复内核的帧指针。	否	是
<code>kernel_tp</code>	保存内核态的硬件线程指针 (Hardware Thread Pointer)，用于在返回用户态时恢复内核的线程指针。	否	是
<code>user_fx</code>	保存用户态的浮点寄存器上下文，包括所有的浮点寄存器以及相关的状态标志。	是	否

2.3.1 内核态 → 用户态

在 Phoenix 中，从内核态切换到用户态发生在以下两种情况：

1. `Task` 从 ELF 文件中构建出来时
2. 因为内核态的系统调用、中断或异常处理完成重新返回用户态时

通常情况下，在返回用户态之前需要执行信号处理函数，不过在第一种情况中，`Task` 刚被创建时还没有收到信号，因此 `loop` 循环第一行是直接 `trap_return`，将 `do_signal` 放在 `loop` 的结尾。

```
pub fn trap_return(task: &Arc<Task>) {
    unsafe {
        disable_interrupt();
        set_user_trap()
    };
    task.time_stat().record_trap_return();
    unsafe {
        __return_to_user(task.trap_context_mut());
    }
    task.time_stat().record_trap();
}
```

在返回用户态之前, 为了保证上下文切换的原子性, 先禁用中断以防止在切换过程中发生新的中断。调用 `set_user_trap()` 函数设置陷阱处理函数, 以确保下一次从用户态陷入内核态时能够正确处理。使用 `__return_to_user` 汇编函数, 将之前存储的用户寄存器恢复。

在返回用户态时, 将 `TrapContext` 的地址存储在 `sscratch` 寄存器中, `a0` 寄存器指向 `TrapContext`。再保存内核态的寄存器, 切换栈指针并恢复状态寄存器、通用寄存器、用户态指针, 最后使用 `sret` 指令返回用户态。

2.3.2 用户态 → 内核态

在 Phoenix 中, 从用户态切换到内核态发生在以下三种情况:

1. **系统调用**: 用户程序请求内核服务。
2. **中断**: 硬件设备需要处理。
3. **异常**: 用户程序执行非法操作。

当从用户态陷入内核态时, 会进入 `__trap_from_user` 标签定义的代码。这段汇编代码的作用是保存用户态的上下文, 并准备好内核态的环境。接着进入 `trap_handler` 函数根据不同的陷阱类型进行处理。

`trap_handler` 函数会获取任务的陷阱上下文、读取 `stval`、`scause`、`sepc` 寄存器的值, 得到引起陷阱的虚拟地址、引起陷阱的原因、引起陷阱的指令地址。如果是系统调用则依据无栈协程架构异步执行系统调用, 如果是页面错误则调用 `handle_page_fault` 函数, 如果是非法指令则终止任务, 如果是定时器中断则让出处理器等待下次调度。

2.3.3 内核态 → 内核态

在 MIT 的 xv6 操作系统与清华大学的 rCore-Tutorial 操作系统中, 并不支持内核态的中断, 这会导致内核对部分中断的响应不及时, 在 rCore-Tutorial 操作系统中甚至会直接 Panic。

Phoenix 允许嵌套中断, 例如内核态在收到时钟中断时, 或者在执行系统调用中遇到访存异常时, 依然会对中断进行处理。

内核态发生中断时, Phoenix 依次执行如下操作:

- **保存调用者保存寄存器**: 在陷阱发生时, 保存调用者保存的寄存器 (`ra`, `t0-t6`, `a0-a7`)。
- **调用陷阱处理函数**: 调用 `kernel_trap_handler` 处理陷阱。
- **恢复调用者保存寄存器**: 在陷阱处理完成后, 恢复之前保存的寄存器。

- **返回指令**: 使用 `sret` 指令返回到陷阱发生前的执行点。

此外, 在执行系统调用时会涉及到用户态指针的读写, Phoenix 采取的是直接解引用用户态指针的方法提高系统调用的执行速度。但是用户态传入的指针可能指向了非法内存, 此时对非法内存的访问也会造成内核态的中断。为此, Phoenix 在读写用户态指针时, 将中断处理函数单独设置为 `__user_rw_trap_vector` 对内存读写产生的中断进行单独处理。

3 内存管理

3.1 物理内存管理

3.1.1 内核动态内存分配器

Phoenix 使用伙伴分配器管理内核所需的动态内存结构, 来自 `crate buddy_system_allocator`。

伙伴分配器 (Buddy Allocator) 是一种内存分配算法, 常用于操作系统内核和高性能应用程序中, 通过分配和管理内存块来满足不同大小的内存请求, 并进行高效的合并和分割操作。其工作原理是将内存块按照 2 的幂次方大小分为多个层级, 当需要分配特定大小的内存时, 从最小适合该请求的层级开始查找。每个内存块都有一个“伙伴”块, 如果块大小是 2^k , 那么它的伙伴块也是 2^k 且紧挨着它。通过检查和计算伙伴块的地址, 可以快速地进行内存分割与合并。当需要分配的内存块小于当前可用最小块时, 将当前块一分为二, 直到找到合适大小的块为止; 当释放一个内存块时, 若其伙伴块也空闲, 则将两个块合并为一个更大的块, 递归进行直到不能再合并为止。伙伴分配器的优点在于分配和释放内存块的操作非常快速, 且通过内存块大小的选择和合并操作, 有效减少了外部碎片。

3.1.2 物理页分配器

内核还需要管理全部的空闲物理内存, Phoenix 为此使用了来自 `rCore` 的仓库的 `bitmap-allocator`。Phoenix 在内核初始化时, 会将所有内核未占用的物理内存加入物理页分配器。

Bitmap allocator 的主要原理是通过一个位图来管理一段连续的内存空间。这个位图中的每一位代表一块内存, 如果该位为 0, 说明对应的内存块空闲; 如果该位为 1, 说明对应的内存块已经被分配出去。当需要分配一个指定大小的内存时, bitmap allocator 首先检查位图中是否有足够的连续空闲内存块可以满足分配请求。如果有, 就将对应的位图标记为已分配,

并返回该内存块的起始地址；如果没有，就返回空指针，表示分配失败。当需要释放已经分配出去的内存时，bitmap allocator 将对应位图标记为未分配。这样，已经释放的内存块就可以被下一次分配请求使用了。

此外，Phoenix 将物理页帧抽象成 `FrameTracker` 结构体，并结合 RAII 的思想，在结构体析构时自动调用 `dealloc_frame` 函数将页帧释放。

```
/// Manage a frame which has the same lifecycle as the tracker.
pub struct FrameTracker {
    /// PPN of the frame.
    pub ppn: PhysPageNum,
}

impl Drop for FrameTracker {
    fn drop(&mut self) {
        dealloc_frame(self.ppn);
    }
}
```

3.2 地址空间

3.2.1 地址空间布局

Phoenix 地址空间的设计如下图所示：

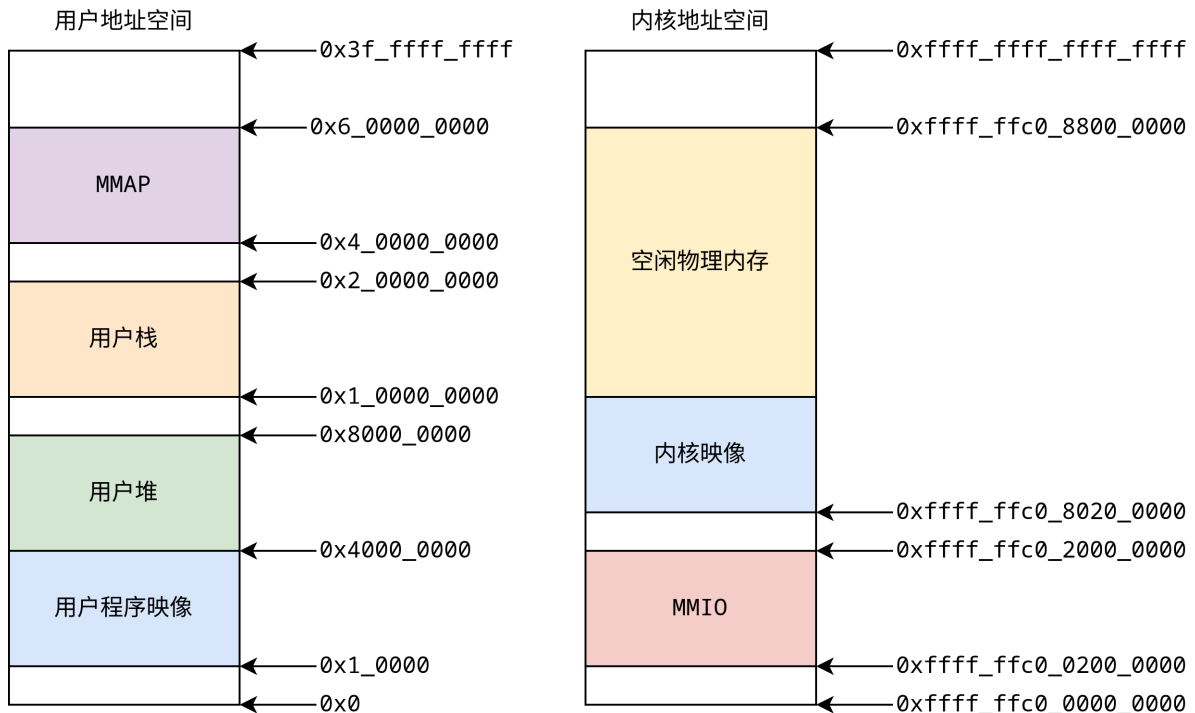


图 3-1: 地址空间

Phoenix 内核态页表保存在全局内核地址空间 `KERNEL_SPACE` 中，用户地址空间共享内核二级页表。

对于内核地址空间，为了方便管理所有物理地址，采用偏移映射的方式将物理地址以加上偏移量的方式映射为虚拟地址，即每一个虚拟地址都为对应物理地址加上 `VIRT_RAM_OFFSET`。

对于用户地址空间，为了利用分页机制的灵活性，消除外部碎片，采用随机映射的方式将需要的页随机映射到空闲物理内存中随机一块页帧。

3.2.2 Boot 阶段高位映射

在 QEMU 平台上，内核的入口地址位于 `0x8020_0000`。在 xv6 与 rCore-Tutorial 中，`0x8020_0000` 以上部分以直接映射的方式作为内核地址空间，`0x8000_0000` 以下部分以随机映射的方式作为用户程序地址空间。因此用户程序最多只有 2G 的地址空间，而考虑到 MMIO 也映射在低位，用户程序地址空间只会更少。因此，Phoenix 充分利用页表的灵活性，将内核地址空间以偏移映射的方式映射到 RISC-V SV39 规定的高位地址空间，即 `0xFFFF_FFC0_0000_0000` 至 `0xFFFF_FFFF_FFFF_FFFF`，每一个虚拟地址对应于物理地址加上一个相同的偏移量，而将用户地址空间映射到 `0x0` 至 `0x3F_FFFF_FFFF`。这样，用户地址空间不足的问题就被解决了。

但是问题也随之而来，OpenSBI 会识别内核 ELF 文件入口地址的加载地址 (LMA)，然后在启动完毕后会跳转到此处，这时 RISC-V 页表机制还未开启，内核会直接访问物理地址。而开启页表时需要保证指令在启动页表前后物理地址连续，这就需要跳板页。Phoenix 希望在内核尽早映射到高位地址空间，这样在调试时能够尽早将代码与地址对应，因此 Phoenix 首先将内核虚拟地址链接到高位地址空间，将加载地址链接到低位，然后结合 RISC-V 巨页的机制，在内核启动阶段，在 Boot 页表中构造了三个巨页，地址分别为 `0x8000_0000`、`0xFFFF_FFC0_0000_0000` 和 `0xFFFF_FFC0_8000_0000`。其中，`0x8000_0000` 所在巨页作为跳板页，保证在打开页表前后 `pc` 寄存器指向连续的物理地址；`0xFFFF_FFC0_0000_0000` 所在巨页保存了 MMIO 的高位映射，以便在 Boot 页表阶段开启打印调试功能；`0xFFFF_FFC0_8000_0000` 所在巨页对应于内核 ELF 文件的虚拟地址高位映射，在 `_start` 函数最后会跳转到此处执行内核代码。在 `entry.rs` 代码执行完毕后，内核便成功执行在高位地址空间了，最后，Boot 页表会在 `main` 函数执行过程中被更加细化的全局内核页表 `KERNEL_SPACE` 所取代。

```
// arch/src/riscv64/entry.rs

#[link_section = ".bss.stack"]
static mut BOOT_STACK: [u8; KERNEL_STACK_SIZE * HART_NUM] = [0u8;
    KERNEL_STACK_SIZE * HART_NUM];

#[repr(C, align(4096))]
struct BootPageTable([u64; PTE_NUM_IN_ONE_PAGE]);

static mut BOOT_PAGE_TABLE: BootPageTable = {
    let mut arr: [u64; 512] = [0; 512];
    arr[2] = (0x80000 << 10) | 0xcf;
    arr[256] = (0x00000 << 10) | 0xcf;
    arr[258] = (0x80000 << 10) | 0xcf;
    BootPageTable(arr)
};

#[naked]
#[no_mangle]
#[link_section = ".text.entry"]
unsafe extern "C" fn _start(hart_id: usize) → ! {
    core::arch::asm!(
        // 1. set boot stack
        // sp = boot_stack + (hartid + 1) * 64KB
        "
            addi    t0, a0, 1
            slli    t0, t0, 16                // t0 = (hart_id + 1) * 64KB
            la      sp, {boot_stack}
            add     sp, sp, t0                // set boot stack
        ",
        // 2. enable sv39 page table
        // satp = (8 << 60) | PPN(page_table)
        "
            la      t0, {page_table}
            srli    t0, t0, 12
            li      t1, 8 << 60
            or      t0, t0, t1
            csrw    satp, t0
            sfence.vma
        ",
        // 3. jump to rust_main
        // add virtual address offset to sp and pc
        "
            li      t2, {virt_ram_offset}
            or      sp, sp, t2
            la      a2, rust_main
            or      a2, a2, t2
            jalr    a2                        // call rust_main
        ",
        boot_stack = sym BOOT_STACK,
        page_table = sym BOOT_PAGE_TABLE,
    );
}
```

```

        virt_ram_offset = const VIRT_RAM_OFFSET,
        options(noreturn),
    )
}

```

3.2.3 地址空间管理

Phoenix 使用 RAII 机制管理地址空间，主要使用 `MemorySpace` 和 `VmArea` 以及 `RangeMap` 数据结构。

```

/// Virtual memory space for kernel and user.
pub struct MemorySpace {
    /// Page table of this memory space.
    page_table: PageTable,
    /// Map of `VmArea`s in this memory space.
    areas: RangeMap<VirtAddr, VmArea>,
}

/// A contiguous virtual memory area.
#[derive(Clone)]
pub struct VmArea {
    /// Aligned `VirtAddr` range for the `VmArea`.
    range_va: Range<VirtAddr>,
    /// Hold pages with RAII.
    pub pages: BTreeMap<VirtPageNum, Arc<Page>>,
    /// Map permission of this area.
    pub map_perm: MapPerm,
    /// Type of this area.
    pub vma_type: VmAreaType,

    // For mmap.
    /// Mmap flags.
    pub mmap_flags: MmapFlags,
    /// The underlying file being mapped.
    pub backed_file: Option<Arc<dyn File>>,
    /// Start offset in the file.
    pub offset: usize,
}

/// A range map that stores range as key.
///
/// # Panic
///
/// Range is clipped or range is empty.
#[derive(Clone)]
pub struct RangeMap<U: Ord + Copy + Add<usize>, V>(BTreeMap<U, Node<U, V>>);

```

3.3 缺页异常处理

Phoenix 目前能够利用缺页异常处理来实现写时复制 (Copy on write)、懒分配 (Lazy page allocation) 以及用户地址检查机制。

当用户程序因缺页异常返回内核时，内核异常处理函数能够从 `stval` 寄存器读取异常发生的地址，并交给 `VmArea::handle_page_fault` 函数进行处理。

```
pub fn handle_page_fault(
    &mut self,
    page_table: &mut PageTable,
    vpn: VirtPageNum,
) → SysResult<()> {
    let page: Page;
    let pte = page_table.find_pte(vpn);
    if let Some(pte) = pte {
        // if PTE is valid, then it must be COW
        let old_page = self.get_page(vpn);
        let mut cnt: usize;
        let cnt = Arc::strong_count(old_page);
        if cnt > 1 {
            // shared now
            // copy the data
            page = Page::new();
            page.copy_data_from_another(&old_page);

            // unmap old page and map new page
            pte_flags.remove(PTEFlags::COW);
            pte_flags.insert(PTEFlags::W);
            page_table.map_force(vpn, page.ppn(), pte_flags);
            // NOTE: track `Page` with great care
            self.pages.insert(vpn, Arc::new(page));
            unsafe { sfence_vma_vaddr(vpn.to_va().into()) };
        } else {
            // not shared
            // set the pte to writable
            pte_flags.remove(PTEFlags::COW);
            pte_flags.insert(PTEFlags::W);
            pte.set_flags(pte_flags);
            unsafe { sfence_vma_vaddr(vpn.to_va().into()) };
        }
    } else {
        match self.vma_type {
            VmAreaType::Heap ⇒ {
                // lazy allcation for heap
                page = Page::new();
                page_table.map(vpn, page.ppn(), self.map_perm.into());
                self.pages.insert(vpn, Arc::new(page));
                unsafe { sfence_vma_vaddr(vpn.to_va().into()) };
            }
            _ ⇒ {}
        }
    }
}
```

```
    }  
    Ok::<(),  
}
```

3.3.1 用户地址检查

在系统调用过程中，内核需要频繁与用户态指针指向的数据进行交互。在 Phoenix 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，而是可以利用硬件页表机制直接解引用用户态指针。

然而，用户态指针并不总是有效的，有可能指向非法内存，出于安全性保证，内核需要能够捕获这种异常。在用户态下，无效指针解引用，或者向只读地址写入数据会触发缺页异常，陷入内核并执行相应缺页异常处理函数，通常，缺页异常处理函数会向程序发送 `SIGSEGV` 信号或者终止进程。而 Phoenix 内核态也需要解引用用户态的指针，因此内核也需要捕获并处理这种页错误。

我们参考了往届 MankorOS 队伍的做法，借助硬件 MMU 部件的帮助实现了高效的指针检查。该做法的基本思路是，先将内核的异常捕捉函数替换为“用户检查模式”下的函数，然后直接尝试向目标地址读取或写入一个字节。若是目标地址发生了缺页异常，则内核将表现得如同用户程序发生了一次异常一般，进入用户缺页异常处理程序进行处理。若处理成功或目标地址访问成功，便可假定当前整个页范围内都是合法的用户地址空间，否则用户指针便不合法。该处理方法相当于直接利用了硬件 MMU 来检查用户指针是否可读或可写，在用户指针正常时速度极快，同时还能完全复用用户缺页异常处理的代码来处理用户指针懒加载/CoW 的情况。

4 文件系统

4.1 虚拟文件系统

虚拟文件系统（Virtual File System，简称 VFS）是内核中负责与各种字符流（如磁盘文件，IO 设备等等）对接，并对外提供操作接口的子系统。它为用户程序提供了一个统一的文件和文件系统操作接口，屏蔽了不同文件系统之间的差异和操作细节。这意味着，用户程序可以使用标准的系统调用，如 `open()`、`read()`、`write()` 来操作文件，而无需关心文件实际存储在何种类型的文件系统或存储介质上。

Phoenix OS 的虚拟文件系统以 Linux 为师，并结合 Rust 语言的特性，从面向对象的角度出发对虚拟文件系统进行了设计和优化。

4.1.1 虚拟文件系统结构

目前虚拟文件系统包含 `SuperBlock`、`Inode`、`Dentry`、`File` 等核心数据结构，也包含 `FdTable`、`Pipe` 等用于实现系统调用的辅助数据结构。

4.1.2 核心数据结构

4.1.2.1 SuperBlock

超级块对象用于存储特定文件系统的信息，通常对应于存放在磁盘特定扇区中的文件系统超级块。超级块是对文件系统的具象，换句话说，一个超级块对应一个文件系统的实例。对于基于磁盘上的文件系统，当文件系统被挂载内核时，内核需要读取文件系统位于磁盘上的超级块，并在内存中构造超级块对象；当文件系统卸载时，需要将超级块对象释放，并将内存中的被修改的数据写回到磁盘。对于并非基于磁盘上的文件系统（如基于内存的文件系统，比如 `sysfs`），就只需要在内存构造独立的超级块。

超级块由 `SuperBlock` trait 定义，如下：

```
pub trait SuperBlock: Send + Sync {
    /// Get metadata of this super block.
    fn meta(&self) → &SuperBlockMeta;

    /// Get filesystem statistics.
    fn stat_fs(&self) → SysResult<StatFs>;

    /// Called when VFS is writing out all dirty data associated with a
    /// superblock.
    fn sync_fs(&self, wait: isize) → SysResult<()>;
}
```

与传统的面向对象编程语言（如 Java 或 C++）不同，Rust 没有内置的类继承机制，而是鼓励使用组合和 trait 来实现代码复用和抽象。如果要模拟继承特性，就需要设计 Meta 结构体来表示对基类的抽象，为了使用继承来简化设计，减少冗余代码，超级块基类对象的设计由 `SuperBlockMeta` 结构体表示。

```
pub struct SuperBlockMeta {
    /// Block device that hold this file system.
    pub device: Option<Arc<dyn BlockDevice>>,
}
```

```

    /// File system type.
    pub fs_type: Weak<dyn FileSystemType>,
    /// Root dentry points to the mount point.
    pub root_dentry: Once<Arc<dyn Dentry>>,
    /// All inodes.
    pub inodes: Mutex<Vec<Arc<dyn Inode>>>,
    /// All dirty inodes.
    pub dirty: Mutex<Vec<Arc<dyn Inode>>>,
}

```

对于具体的文件系统，只需要实现自己的超级块对象，其中包含 `SuperBlockMeta` 的字段，就能完成继承对超级块基类的继承。比如对 FAT32 文件系统，我们只需要构造这样一个 `FatSuperBlock` 对象就能完成对 VFS `SuperBlockMeta` 的继承，同时，只需要为 `FatSuperBlock` 实现 `SuperBlock` trait 就能实现对接口方法的多态行为。这样就能在 Rust 语言中使用面向对象的设计来大大简化具体文件系统与 VFS 层接口对接的代码量。

```

pub struct FatSuperBlock {
    meta: SuperBlockMeta,
    fs: Arc<FatFs>,
}

```

4.1.2.2 Inode

索引节点是对文件系统中文件信息的抽象。对于文件系统中的文件来说，文件名可以随时更改，但是索引节点对文件一定是唯一的，并且随文件的存在而存在。

索引节点由 `Inode` trait 表示，如下：

```

pub trait Inode: Send + Sync + DowncastSync {
    /// Get metadata of this Inode
    fn meta(&self) -> &InodeMeta;

    /// Get attributes of this file
    fn get_attr(&self) -> SysResult<Stat>;
}

```

索引节点对象由 `InodeMeta` 结构体表示，下面给出它的结构和描述：

```

pub struct InodeMeta {
    /// Inode number.
    pub ino: usize,
    /// Mode of inode.
    pub mode: InodeMode,
}

```

```
    /// Super block this inode belongs to
    pub super_block: Weak<dyn SuperBlock>,
    /// Protect mutable data with mutex.
    pub inner: Mutex<InodeMetaInner>,
}

pub struct InodeMetaInner {
    /// Size of a file in bytes.
    pub size: usize,
    /// Last access time.
    pub atime: TimeSpec,
    /// Last modification time.
    pub mtime: TimeSpec,
    /// Last status change time.
    pub ctime: TimeSpec,
    /// State of a file
    pub state: InodeState,
}
```

4.1.2.3 Dentry

目录项是管理文件在目录树中的信息的结构体，是对文件路径的抽象。在文件系统中，以挂载点，即文件系统的根目录为根节点，按照文件夹与下属文件的父子关系逐级向下，形成一个目录树的结构。目录树的每个节点对应一个目录项，每一个目录项都指向一个文件的索引节点。

Dentry 存在的必要性源于 Unix 将文件本身与文件名解耦合的设计，这使得不同的目录项可以指向相同的索引节点（即硬链接）。虽然竞赛规定使用的 FAT32 文件系统在设计上是路径与文件本身耦合的，这也导致其不支持硬链接技术，因而往届很多作品并没有 Dentry 这个结构，而是将路径解析的功能保存在 Inode 结构体中，这样的做法是针对竞赛的简化，然而，这并不符合 Unix 哲学，这种 VFS 设计并不能扩展到其他文件系统上。而 Phoenix 认为遵守 Unix 设计哲学能有更好的扩展性，因此，Phoenix 选择遵守 Unix 设计规范，将路径与文件本身相分离，形成了 Dentry 和 Inode 这两者的抽象。

目录项与索引节点是多对一的映射关系，因此文件系统只需要缓存目录项就能缓存对应的索引节点。而目录项的状态分为两种，一种是被使用的，即正常指向 Inode 的目录项，一种是负状态，即没有对应 Inode 的目录项。负目录项的存在是因为文件系统试图访问不存在的路径，或者文件被删除了。如果没有负目录项，文件系统会到磁盘上遍历目录结构体并检查这个文件的确不存在，这样的失败查找非常浪费资源，为了尽量减少对磁盘的 IO 访问，Phoenix 的文件系统会缓存这些负目录项，以便快速解析这些路径。

目录项的操作由 `Dentry` trait 描述，定义如下：

```
pub trait Dentry: Send + Sync {
    /// Get metadata of this Dentry
    fn meta(&self) → &DentryMeta;

    /// Open a file associated with the inode that this dentry points to.
    fn base_open(self: Arc<Self>) → SysResult<Arc<dyn File>>;

    /// Look up in a directory inode and find file with `name`.
    ///
    /// If the named inode does not exist, a negative dentry will be created
as
    /// a child and returned. Returning an error code from this routine must
    /// only be done on a real error.
    fn base_lookup(self: Arc<Self>, name: &str) → SysResult<Arc<dyn Dentry>>;

    /// Called by the open(2) and creat(2) system calls. Create an inode for
a
    /// dentry in the directory inode.
    ///
    /// If the dentry itself has a negative child with `name`, it will create
an
    /// inode for the negative child and return the child.
    fn base_create(self: Arc<Self>, name: &str, mode: InodeMode) →
SysResult<Arc<dyn Dentry>>;

    /// Called by the unlink(2) system call. Delete a file inode in a
directory
    /// inode.
    fn base_unlink(self: Arc<Self>, name: &str) → SyscallResult;

    /// Called by the rmdir(2) system call. Delete a dir inode in a directory
    /// inode.
    fn base_rmdir(self: Arc<Self>, name: &str) → SyscallResult;
}
```

目录项对象由 `DentryMeta` 结构体表示：

```
pub struct DentryMeta {
    /// Name of this file or directory.
    pub name: String,
    /// Super block this dentry belongs to
    pub super_block: Weak<dyn SuperBlock>,
    /// Parent dentry. `None` if root dentry.
    pub parent: Option<Weak<dyn Dentry>>,
    /// Inode it points to. May be `None`, which is called negative dentry.
    pub inode: Mutex<Option<Arc<dyn Inode>>>,
    /// Children dentries. Key value pair is <name, dentry>.
```

```
pub children: Mutex<BTreeMap<String, Arc<dyn Dentry>>>,>
}
```

4.1.2.4 File

文件对象是进程已打开的文件在内存中的表示。文件对象由系统调用 `open()` 创建, 由系统调用 `close()` 撤销, 所有文件相关的系统调用实际上都是文件对象定义的操作。文件对象与文件系统中的文件并不是一一对应的关系, 因为多个进程可能会同时打开同一个文件, 也就会创建多个文件对象, 但这些文件对象指向的索引节点都是同一个索引节点, 即同一个文件。

文件对象的操作由 `File` 描述, 其形式如下:

```
pub trait File: Send + Sync {
    /// Get metadata of this file
    fn meta(&self) -> &FileMeta;

    /// Called by read(2) and related system calls.
    ///
    /// On success, the number of bytes read is returned (zero indicates end
    of
    /// file), and the file position is advanced by this number.
    async fn read(&self, offset: usize, buf: &mut [u8]) -> SyscallResult;

    /// Called by write(2) and related system calls.
    ///
    /// On success, the number of bytes written is returned, and the file
    offset
    /// is incremented by the number of bytes actually written.
    async fn write(&self, offset: usize, buf: &[u8]) -> SyscallResult;

    /// Read directory entries. This is called by the getdents(2) system
    call.
    ///
    /// For every call, this function will return an valid entry, or an
    error.
    /// If it read to the end of directory, it will return an empty entry.
    fn base_read_dir(&self) -> SysResult<Option<DirEntry>>;

    /// Called by the close(2) system call to flush a file
    fn flush(&self) -> SysResult<usize>;

    /// called by the ioctl(2) system call.
    fn ioctl(&self, cmd: usize, arg: usize) -> SyscallResult;

    /// called when a process wants to check if there is activity on this
```

```

file and (optionally)
    /// go to sleep until there is activity.
    /// Called by the select(2) and poll(2) system calls
    async fn poll(&self, events: PollEvents) → SysResult<PollEvents>;

    /// Called when the VFS needs to move the file position index.
    ///
    /// Return the result offset.
    fn seek(&self, pos: SeekFrom) → SysResult<usize>;
}

```

文件对象的设计由 `FileMeta` 结构体表示，下面给出它的结构和描述：

```

pub struct FileMeta {
    /// Dentry which points to this file.
    pub dentry: Arc<dyn Dentry>,
    /// Inode which points to this file
    pub inode: Arc<dyn Inode>,
    /// Offset position of this file.
    pub pos: AtomicUsize,
    /// File mode
    pub flags: Mutex<OpenFlags>,
}

```

4.1.2.5 FileSystemType

`FileSystemType` 用来描述各种特定文件系统类型的功能和行为，并负责管理每种文件系统下的所有文件系统实例以及对应的超级块。

`FileSystemType` trait 的定义如下：

```

pub trait FileSystemType: Send + Sync {
    fn meta(&self) → &FileSystemTypeMeta;

    /// Call when a new instance of this filesystem should be mounted.
    fn base_mount(
        self: Arc<Self>,
        name: &str,
        parent: Option<Arc<dyn Dentry>>,
        flags: MountFlags,
        dev: Option<Arc<dyn BlockDevice>>,
    ) → SysResult<Arc<dyn Dentry>>;

    /// Call when an instance of this filesystem should be shut down.
    fn kill_sb(&self, sb: Arc<dyn SuperBlock>) → SysResult<()>;
}

```

`FileSystemType` 的设计由 `FileSystemTypeMeta` 结构体表示，下面给出它的结构和描述：

```
pub struct FileSystemTypeMeta {  
    /// Name of this file system type.  
    name: String,  
    /// Super blocks.  
    supers: Mutex<BTreeMap<String, Arc<dyn SuperBlock>>>,  
}
```

4.1.2.6 Path

`Path` 结构体的主要用来实现路径解析，由于我们在 `DentryMeta` 中使用 `BTreeMap` 来对缓存一个文件夹下的所有子目录项，因此我们能够在内存中快速进行路径解析，而无需重复访问磁盘进行耗时的 IO 操作。

```
pub struct Path {  
    /// The root of the file system  
    root: Arc<dyn Dentry>,  
    /// The directory to start searching from  
    start: Arc<dyn Dentry>,  
    /// The path to search for  
    path: String,  
}
```

由于我们已经通过 `Dentry` 实现了对目录树的抽象，路径解析的实现非常简单，只需要判断传入路径为绝对路径或相对路径，然后逐级对目录进行查找即可。

```
impl Path {  
    /// Walk until path has been resolved.  
    pub fn walk(&self) → SysResult<Arc<dyn Dentry>> {  
        let path = self.path.as_str();  
        let mut dentry = if is_absolute_path(path) {  
            self.root.clone()  
        } else {  
            self.start.clone()  
        };  
        for p in split_path(path) {  
            match p {  
                ".." => {  
                    dentry = dentry.parent().ok_or(SysError::ENOENT)?;  
                }  
                name => match dentry.lookup(name) {  
                    Ok(sub_dentry) => {  
                        dentry = sub_dentry  
                    }  
                }  
            }  
        }  
    }  
}
```

```

    }
    Err(e) => {
        return Err(e);
    }
},
}
}
Ok(dentry)
}
}
}

```

4.1.3 其他数据结构

4.1.3.1 FdTable

Unix 设计哲学将文件本身抽象成 Inode，其保存了文件的元数据；将内核打开的文件抽象成 File，其保存了当前读写文件的偏移量以及文件打开的标志；进程只能看见文件描述符，文件描述符由进程结构体中的文件描述符表进行处理。

当一个进程调用 `open()` 系统调用，内核会创建一个文件对象来维护被进程打开的文件的信息，但是内核并不会将这个文件对象返回给进程，而是将一个非负整数返回，即 `open()` 系统调用的返回值是一个非负整数，这个整数称作文件描述符。文件描述符和文件对象一一对应，而维护二者对应关系的数据结构，就是文件描述符表。在实现细节中，文件描述符表本质是一个数组，数组中每一个元素就是文件对象，而元素下标就是文件对象对应的文件描述符。

4.1.3.2 Pipe

管道 Pipe 是一种基本的进程间通信机制。它允许一个进程将数据流输出到另一个进程。文件系统来实现管道通信，实现方式就是创建一个 FIFO 类型的管道文件，文件内容就是一个缓冲区，同时创建两个文件对象和对应的两个文件描述符。两个文件对象都指向这个管道文件，一个文件负责向管道的缓冲区中写入内容，一个负责从管道的缓冲区中读出内容。

管道文件的数据结构由 `PipeInode` 描述：

```

pub struct PipeInode {
    meta: InodeMeta,
    is_closed: Mutex<bool>,
    buf: Mutex<AllocRingBuffer<u8>>,
}

```

`PipeInode` 是对 VFS 中 `Inode` 数据结构的一个实现，包含元数据、缓冲区和管道是否关闭的信息。

对管道进行读写的两个文件对象，`PipeReadFile` 和 `PipeWriteFile`，则是对 VFS 中 `File` 的实现：

```
pub struct PipeWriteFile {
    meta: FileMeta,
}

pub struct PipeReadFile {
    meta: FileMeta,
}
```

`PipeReadFile` 负责从管道文件中读出数据，因此在实现 `File` 的时候，只实现 `read` 方法：

```
impl File for PipeReadFile {
    async fn read(&self, offset: usize, buf: &mut [u8]) →
    systype::SysResult<usize> {
        let pipe = self
            .inode()
            .downcast_arc::<PipeInode>()
            .map_err(|_| SysError::EIO)?;
        let mut pipe_len = pipe.buf.lock().len();
        while pipe_len == 0 {
            yield_now().await;
            pipe_len = pipe.buf.lock().len();
            if *pipe.is_closed.lock() {
                break;
            }
        }
        let mut pipe_buf = pipe.buf.lock();
        let len = core::cmp::min(pipe_buf.len(), buf.len());
        for i in 0..len {
            buf[i] = pipe_buf
                .dequeue()
                .expect("Just checked for len, should not fail");
        }
        Ok(len)
    }
}
```

调用 `read` 方法，当缓冲区没有数据时，读进程会主动让出 CPU 资源，等待异步调度器调度到本进程时再次查看缓冲区是否有数据。上述步骤会一直重复，知道写进程将数据写入缓冲区，之后读进程将数据从缓冲区读出。

`PipeWriteFile` 负责向管道文件写入数据，在实现 `File` 的时候，只实现 `write` 方法：

```
impl File for PipeWriteFile {
    async fn write(&self, offset: usize, buf: &[u8]) →
systype::SysResult<usize> {
    let pipe = self
        .inode()
        .downcast_arc::<PipeInode>()
        .map_err(|_| SysError::EIO)?;
    let mut pipe_buf = pipe.buf.lock();
    let space_left = pipe_buf.capacity() - pipe_buf.len();

    let len = core::cmp::min(space_left, buf.len());
    for i in 0..len {
        pipe_buf.push(buf[i]);
    }
    log::trace!("[Pipe::write] already write buf {buf:?} with data len {len:?}");
    Ok(len)
}
}
```

管道文件在写进程的 `PipeWriteFile` 生命周期结束时关闭，因此在 `PipeWriteFile` 的 `Drop` 中关闭管道文件：

```
impl Drop for PipeWriteFile {
    fn drop(&mut self) {
        let pipe = self
            .inode()
            .downcast_arc::<PipeInode>()
            .map_err(|_| SysError::EIO)
            .unwrap();
        *pipe.is_closed.lock() = true;
    }
}
```

4.2 FAT32 文件系统

FAT32，全称为 File Allocation Table 32，是一种文件系统格式，用于在各种存储设备上存储和管理文件和目录。它是 FAT 文件系统的一个版本，最初由微软在 1996 年引入，主要是为了解决 FAT16 在处理大容量存储设备时的限制问题。FAT32 文件系统在 Windows 操作系统以及许多其他设备和媒体中得到了广泛应用。

FAT32 的主要特点包括：

- 稳定性和兼容性：FAT32 提供了良好的稳定性和兼容性，能够兼容 Win 9X 及以前版本的 Windows 操作系统。
- 簇大小：使用比 FAT16 更小的簇（数据存储单元），从而提高了大容量硬盘上的空间利用率。
- 分区容量：支持的每个分区容量最大可达到 128TB，远大于 FAT16 的限制。
- 文件大小限制：单个文件最大支持 4GB，这对于处理大型文件来说是一个限制。

FAT32 文件系统的结构主要包括三个部分：

- 引导区：包含文件系统的具体信息，如 FAT 表个数、每个 FAT 表的大小、每扇区内的字节数目等。
- 文件分配表区：管理磁盘空间和文件，保存逻辑盘数据区各簇使用情况信息。
- 数据区：存放用户数据，以簇为分配单位来使用。

作为为 Windows 设计的文件系统，FAT32 并没有采取 UNIX 系列文件系统的设计范式。相比于 UNIX 系列的文件系统，FAT32 缺少 UNIX 规定的 `rwX` 权限管理，也没有提供硬链接功能或可以实现硬链接功能的模块。虽然要使内核支持 FAT32，只需实现对应的 VFS 接口，但是具体实现仍需要采取一些特殊机制。

Phoenix 使用了开源的 `rust-fatfs` 库，并在其基础上添加了多核的支持。通过实现 FAT32 的 VFS 层接口完成了 FAT32 的对接。

5 进程间通信

5.1 信号机制

信号是操作系统向进程传递事件通知的一种机制，主要用于通知进程发生了异步事件。Phoenix 与往届作品 Titanix 的信号机制相比，信号机制更加完善。Titanix 的信号队列中只有信号编号，Phoenix 参考了 Linux 的实现，使用 `SigInfo` 结构体代替，除了能表示信号编号外，还能携带更多的信息：

```
pub struct SigInfo {
    pub sig: Sig,
    pub code: i32,
    pub details: SigDetails,
}
```


这种设计与 POSIX 标准中的 `siginfo_t` 结构体相似, 增强了 Phoenix 系统与标准 POSIX 接口的兼容性。并且使得 Phoenix 的信号机制具备了更强的表达能力和灵活性。携带附加信息可以是信号的来源、产生原因和相关的上下文数据。例如, `code` 字段可以用于区分不同类型的信号或事件, `details` 字段则可以包含更详细的上下文信息:

```
pub enum SigDetails {
    None,
    Kill {
        /// sender's pid
        pid: usize,
    },
    CHLD {
        /// which child
        pid: usize,
        /// exit code
        status: i32,
        utime: Duration,
        stime: Duration,
    },
}
```

例如, 当子进程状态变为 `Zombie` 时需要向父进程发送 `SIGCHLD` 信号告知父进程状态改变, 此时将 `SigInfo` 中的 `SigDetails` 字段设为 `CHLD`, 并且告诉父进程自己的 `pid`、状态码 `status`、用户态运行时间 `utime` 和内核态运行时间 `stime`, 这些信息可以有助于父进程在 `Wait4` 系统调用时掌握子进程的数据。

5.1.1 信号处理函数

在任务由内核态返回到用户态之前, 往往需要执行信号处理函数, 检查和处理挂起的信号, 调用适当的信号处理程序或执行默认行为, 确保进程能够正确响应和处理信号。这一机制是操作系统处理异步事件、进程控制和进程间通信的重要组成部分。

这里涉及到一个细节, 就是如果此时信号待处理队列中有多个需要处理的信号同时到来, 那么应该以什么顺序处理。大部分往届作品都是按照信号到来顺序依次处理的, 这种 FIFO(先入先出) 方法虽然简单直接, 但在处理高优先级和紧急事件时可能存在不足。例如, 当一个需要立即响应的紧急信号和一个普通信号同时到达时, 按照到达顺序处理可能导致紧急信号的响应延迟。Phoenix 在设计信号处理机制时, 参考了 Linux 内核的实现, 引入了信号优先级的概念, 这样的好处是紧急信号可以立即得到处理, 减少了响应延迟, 提升了系统对关键事件的响应能力。例如, 当发生非法内存访问时, 会触发 `SIGSEGV` (Segmentation Fault) 信号, 指示程序运行中出现严重问题, 这种信号在 Phoenix 中会优先处理。

用户可以使用 `sigaction` 系统调用为某些信号自定义信号处理函数，操作系统内核会按照如下步骤来调用用户自定义的信号处理函数：

1. 保存上下文：内核会保存当前的进程执行上下文，包括寄存器状态、堆栈指针、程序计数器等，以便在信号处理完成后恢复进程的执行。
2. 切换到用户态：内核切换到用户态，并开始执行用户自定义的信号处理函数
3. 恢复上下文：使用 `sigreturn` 函数返回到内核态并且恢复之前保存的进程上下文，如寄存器状态、堆栈指针、程序计数器等。

这里同样存在一个问题，即切换到用户态需要保存上下文，那么应该将上下文保存在哪里？部分作品如往届一等奖作品 Titanix 将上下文记录在内核态，这当然也可以，但是对于 POSIX 规范中一些标志位就难以支持，例如设置了 `SA_SIGINFO` 标志位的信号处理程序的函数签名会变成如下形式：

```
void handler(int sig, siginfo_t *info, void *ucontext);
```

这里 `ucontext` 是指向 `ucontext_t` 结构体的指针，提供接收信号时进程的上下文信息，如果将进程上下文保存在了内核态，那么用户将有机会访问到内核中的信息，这无疑是非常不安全的。虽然竞赛并未有测试程序需要用到该标志位，但是 Phoenix 的目标是实现符合 POSIX 规范的操作系统，因此依然实现了该标志位。

Phoenix 参考了 Linux 的设计，将信号处理时需要保存的进程上下文保存在了用户栈中，这样 `ucontext` 指向用户栈就非常安全。

6 总结与展望

6.1 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. 实现初赛要求的所有系统调用。
5. 实现不同等级的调试日志，实现 backtrace 机制，在内核崩溃时能够打印函数调用栈。

6.2 经验总结

1. 系统调用的实现多参考 System Calls Manual 手册，支持 Linux raw syscall 规范。
2. 多核调试很困难，尽量依靠完善的日志，运行时查看相关信息。

6.3 未来计划

1. 修改异步任务调度器，将全局队列拆分到各个核心，并增加优先级调度机制。
2. 适配 busybox 和 libc，完善相关系统调用的功能。
3. 适配星光二代板，完善相关驱动。
4. 实现完善的网络功能，通过决赛测例。