2023

UMI 设计文档

西安交通大学 PLNTRY

队员：徐启航 / 指导老师：陈渝

摘要

UMI 是一个使用 Rust 语言编写的 Unix 兼容操作系统，以并发与并行为主要设计目标，通过了初赛的所有测试程序和决赛的大部分测试程序。
以下是该操作系统的各个子系统的当前实现：

子系统	实现
内存管理	在页帧管理实现写时复制策略和通用 IO 缓存，减少 IO 设备访问次数。 在地址空间管理实现懒分配策略，提高内存分配速度。
线程管理与调度	基于无栈协程的任务表示 使用细粒度锁和 Rust 所有权系统管理线程的本地状态和信息 支持软抢占和任务窃取的 SMP 多核调度器 基于有栈协程模式的特权级切换
文件系统	统一的虚拟文件系统接口 支持 debugfs、FAT32、procfs、devfs 等多种存储和内存文件系统
网络协议栈	基于 smoltcp 的多设备接口网络协议栈 TCP 独立的 ACCEPT 队列
驱动程序	网络设备分割接口对象 中断驱动的 Virt IO 设备（Block 和 Net） 中断驱动和 DMA 传输方式的 SD 卡驱动
其他模块	各种同步原语、时间管理、信号处理等

目录

一、概述

1.1 总体介绍

UMI 操作系统目前建立在 RISC-V 平台上，秉持着并发与并行两个主要设计目标，利用 Rust 语言和生态的优势，基于模块化架构的设计思想。

在着手设计该操作系统之前，我们队员曾调研学习过各种系统开发的编程语言，最后确定了以 Rust 语言为开发语言。这其中运用到了 Rust 的两个优势：

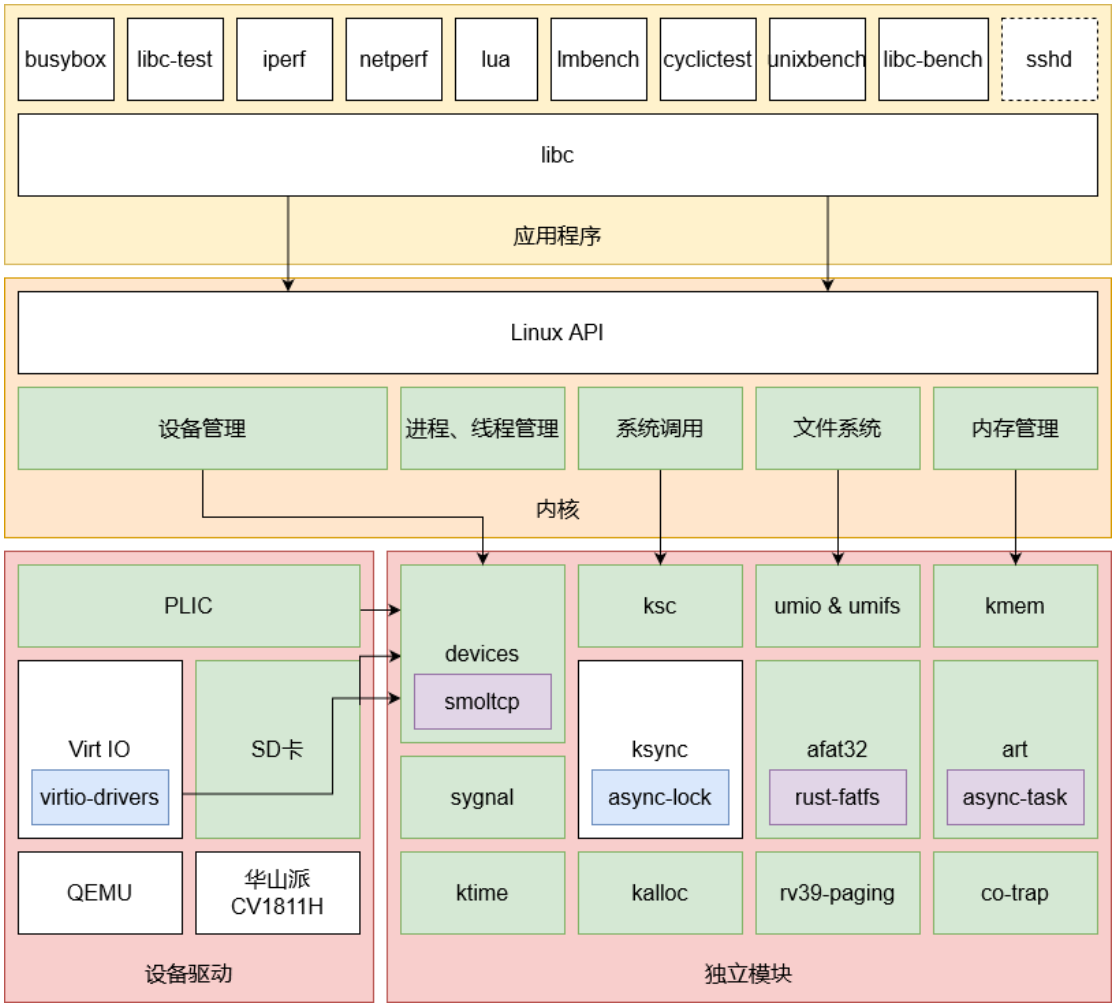
1. Rust 拥有现代化的软件包管理工具以及接近成熟的无依赖和开源友好的包管理生态，避免了大量无意义的重复造轮子，可以有跟为充足的精力去专心实现操作系统的设计；
2. Rust 语言拥有最严谨的所有权系统和借用检查，可以在编译层面最大程度地减少 use-after-free、double-free 等低级的内存安全 bug，从而在保证内存安全的情况下大大加快操作系统的开发效率，减少无意义的调试工作。

也因此，对于该操作系统，我们特别以模块化的方式来设计 UMI 的架构，并且通过构建适当的依赖关系使得一部分模块没有对于自身操作系统内部的依赖，由此保留了对外开放、发布到 crate.io 上来回报 Rust 开源社区的可能。实际上其中一些模块确实从理论上可以发布，只是由于代码完备性和文档注释等原因搁置。

在 UMI 操作系统的开发过程之前，我们已经对 Rust 及其异步子系统有较为精通的掌握，但是在操作系统内核中使用 Rust 的异步子系统还是第一次。在充分了解到 Rust 异步对于全通有栈协程的优势之后，我们确立了并发与并行两个设计目标，并将 Rust 异步作为其实现两个目标的总基础。

1.2 架构简述

UMI 操作系统目前已经实现了一个满足可用性的操作系统需要的各种子系统，诸如任务调度、内存管理、文件系统、网络系统、驱动框架等。在这些主要的子系统中，大部分主要的逻辑代码通过依赖的方式存放于外部模块之中，仅有少部分关联性、整合性代码存放于内核之中。具体的架构图如下：



其中，白色指代简单设计与引用的模块，其中的蓝色指代简单引用的开源库；绿色指代深度设计以及与开源库深度合作开发的模块，其中的紫色指代修改过后适配进该操作系统的开源库。

1.3 代码库简述

除去直接引用的开源库，该操作系统的代码库总共有大约 3 万行代码；再除去修改过的开源库，总共有大约 2.5 万行代码。

代码库的目录树大致如下：

- `.github/workflows` - Github Actions 配置；
- `.vscode` - VS Code 工作区配置；
- `cargo-config` - 根目录 cargo 配置，会在 make 时复制到`.cargo` 目录中；
- `debug` - 调试信息，包括内核文件的反汇编、ELF 元数据还有 QEMU 的输出日志；
- `docs` - 文档；
- `mizu/dev` - 各个设备驱动程序代码；
- `mizu/kernel` - 内核主程序代码；
- `mizu/lib` - 内核各个模块的代码；
- `scripts` - 包含第三方依赖的换源脚本；
- `target` - cargo 的生成目录；

- `third-party/bin` - RustSBI 的 BIOS 二进制文件;
- `third-party/vendor` - 第三方库的依赖;
- `third-party/img` - 初赛评测程序的磁盘映像文件。

二、子系统的设计与实现

2.0 设计目标

在详细讲述各个子系统之前，先简要介绍一下此次的两个设计目标：并发与并行。

并发指的是在宏观层面，由系统内部（包括硬件和软甲）的支持实现多个任务同时运行、多个事件同时被处理的表象；而并行指的是在微观层面，在系统硬件的支持下，多个任务在同时运行、多个事件被同时处理的事实。二者均是任务运行效率在一定程度上的的衡量标准。

并发处于宏观层面，实际在实现的时候可以通过时间片轮转等调度方法，在不支持并行的系统上“近似”使得多个任务在同时运行。在理论上，时间片的粒度越细，同时性的近似度就越好，但是任务间切换的平均开销就越大；反之亦然。

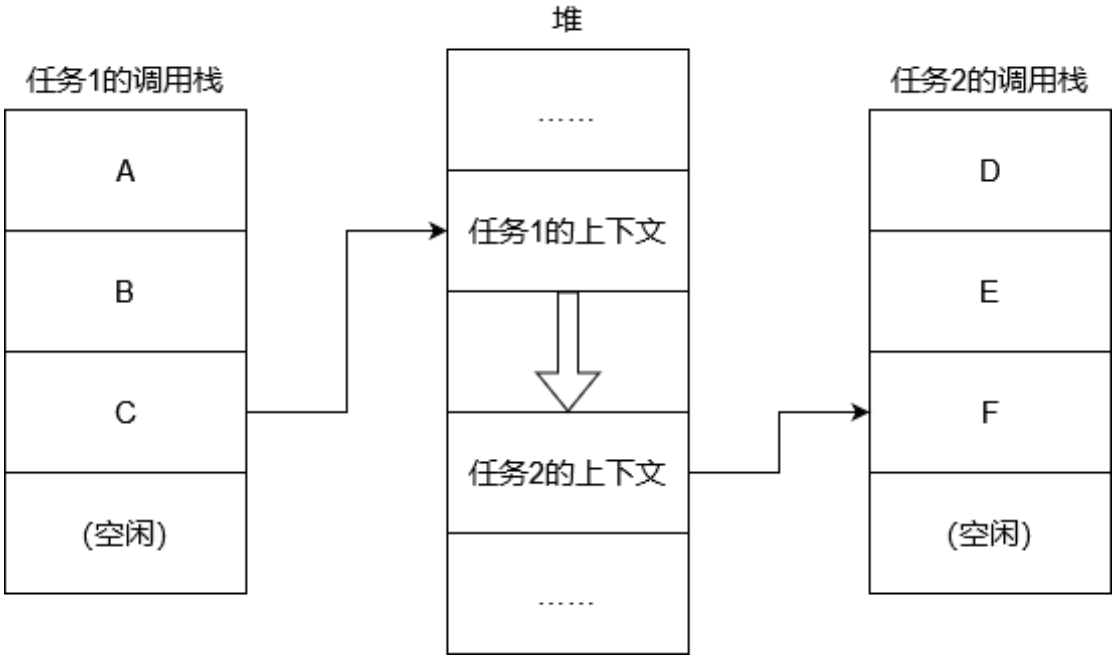
支持高并行与高并发，就要充分运用各种硬件支持和软件算法，通过各个子系统实现的配合来达到高效的任务间切换和调度，达到 CPU 和 IO 资源的最高效利用来提升任务的运行效率。

我们大部分子系统都围绕并发与并行这两个目标进行的特别的设计与实现，接下来将依次展开。

2.1 任务管理和调度

2.1.1 Rust 的异步（无栈协程）简介

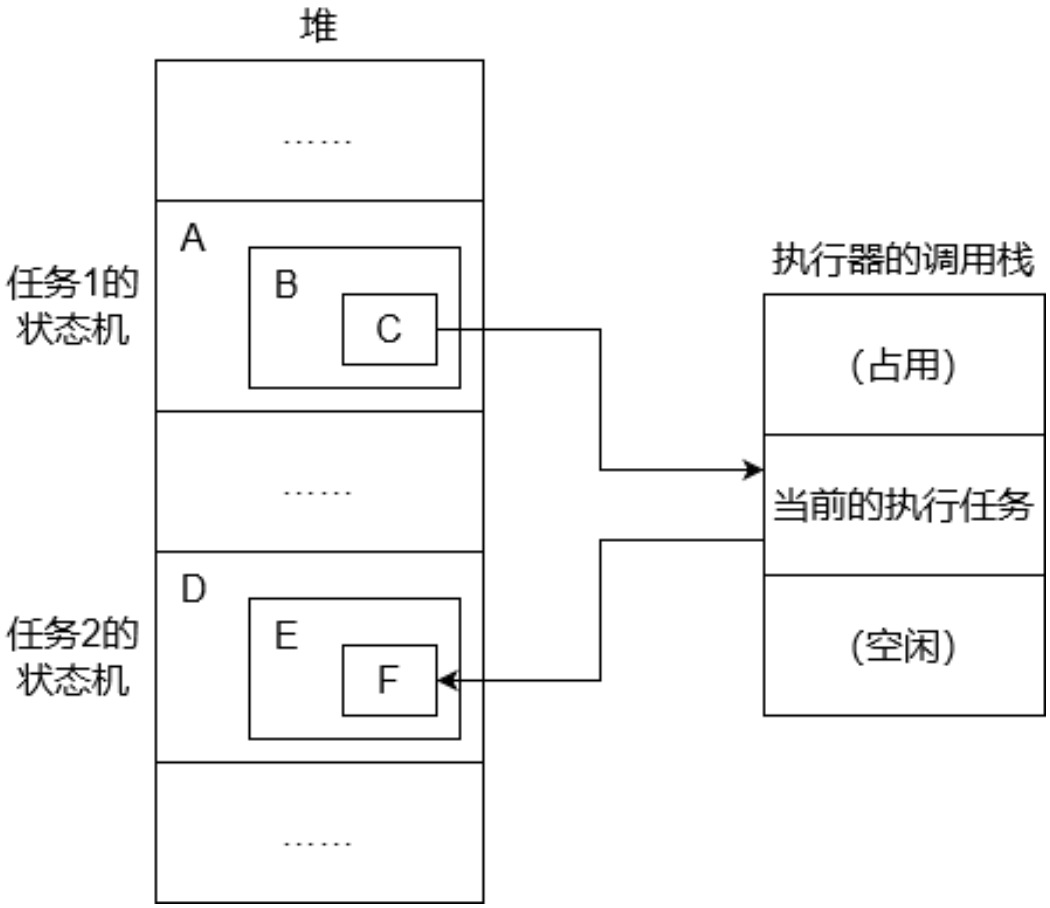
传统的大部分操作系统均使用有栈协程的调度方式，具体的逻辑结构如下：



从图中我们可以发现，有栈协程将任务的调用栈和上下文分开保存，通过汇编代码手动切换函数调用栈来进行任务切换。每个任务的调用栈都会有一定的内存浪费（空闲），并都

会有栈溢出风险。

而无栈协程则将任务的信息同一保存成状态机，统一存放在堆上，由执行器通过更改指针来切换执行的任务。从图中我们可以发现，调用栈仅与每个执行器一一绑定，一定程度上减少了内存浪费、降低栈溢出风险。



2.1.2 任务信息与状态表示

由于比赛对于 POSIX 系统调用的要求，我们并不遵循将线程和进程的结构体分开的设计。但是，这并不意味着保存任务信息和状态的结构体将会上一把全局大锁来控制访问，而是通过 Rust 所有权系统和细粒度锁最大程度地控制对其的高效率访问，达到高并发的效果。

传统的线程和进程分立将进程作为资源管理的基本单位，线程作为调度的基本单位。这样做确实有一定好处，将资源管理和调度正交化，可以减少二者逻辑上的耦合，以此达到减少 bug 的目的。然而，POSIX 的 clone 系统调用的参数并没有完全区分线程和进程，而是将 CLONE_VM、CLONE_FS 和 CLONE_THREAD 等参数独立，从而导致进程不能和文件描述符表或者地址空间等资源进行对等映射，也就无法用线程与进程的分立在理论上完美支持该系统调用。由此可见，我们需要一种新的方式，使之又能完美支持该系统调用，又能达到资源分配与调度代码之间的耦合性。

在我们的设计中，我们将任务结构体中的所有 field 分成两大块——信息部分和状态部分。其中信息部分对外共享，也就是其他任务均可以访问到；状态部分则对内私有，仅该任务本身可以访问。

信息部分的结构体如下：

mizu/kernel/src/task.rs

```
#[derive(Debug)]
pub struct Task {
    // The parent task.
    parent: Weak<Task>,
    // Child tasks, can either be a process or a thread
logically.
    children: spin::Mutex<Vec<Child>>,
    // Task ID.
    tid: usize,
    // Executable file name (arg[0]).
    executable: spin::Mutex<String>,

    // Task time consumptions.
    times: Arc<Times>,

    // The signal struct delivering to itself.
    sig: Signals,
    // The signal struct delivering to its thread group.
    shared_sig: AtomicArsc<Signals>,
    // The broadcaster of the events of the task.
    event: Broadcast<SegQueue<TaskEvent>>,
}
```

而状态部分的结构体如下：

mizu/kernel/src/task.rs

```
pub struct TaskState {
    pub(crate) task: Arc<Task>,
    tgroup: Arsc<(usize, spin::RwLock<Vec<Arc<Task>>>)>,

    counters: [Counter; 3],

    sig_mask: SigSet,
    sig_stack: Option<SigStack>,
    pub(crate) brk: usize,

    pub(crate) virt: Arsc<Virt>,
    pub(crate) futex: Arsc<Futexes>,
    pub(crate) shm: Arsc<Shm>,
    sig_actions: Arsc<ActionSet>,
    pub(crate) files: Files,
    tid_clear: Option<UserPtr<usize, Out>>,
    exit_signal: Option<Sig>,
}
```

可以看到，信息部分的结构体要么不可变，要么每个字段都加了一把锁，要么就是一个专有的结构体；而状态部分的好些则是可变的，比如 `pub(crate) brk: usize` 便是会

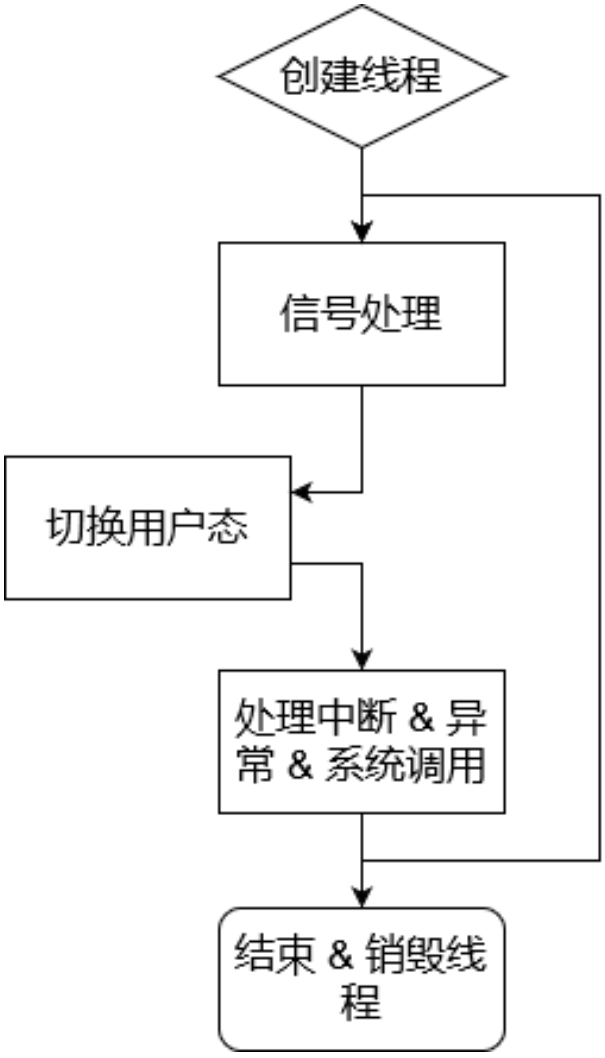
在运行期间变化的一个字段。之所以有一些处于引用计数指针包装下的可能对外共享的字段也被包含在状态部分的结构体内，是因为上述 POSIX 的 clone 系统调用需要控制上述每一个共享字段的深拷贝与否。

在大部分 Rust 实现的线程进程分立的任务管理子系统中，对于线程的可变字段常常使用全局 Mutex 来互斥访问，或者绕过 Rust 所有权机制使用 UnsafeCell 来强行可变访问；这样做要么进行了多余的上锁操作拖慢效率，要么破坏了内存安全，带来潜在的 bug。而在我们的实现中并未见到 UnsafeCell 的身影。那么，我们是如何实现不加 UnsafeCell 就能安全地实现对私有字段的可变访问的呢？请看后文。

在后文的讲述中，为了方便理解，我们仍然将“线程”作为调度单位的称号之一，而包括资源管理的方面则会单独说明。

2.1.3 线程的生命周期

在我们的设计中，一个线程的生命周期可以抽象为如下的流程图。



而在实际的实现中，使用了异步无栈协程之后，上述流程图在代码结构中可以与实际的代码一一对应。对应的简化代码如下：

mizu/kernel/src/task/future.rs

```
pub async fn user_loop(mut ts: TaskState, mut tf: TrapFrame)
{
    …
    let (code, sig) = 'life: loop {
        if let Err((code, sig)) = ts.handle_signals(&mut
tf).await {
            break 'life (code, Some(sig));
        }

        …

        let (scause, fr) = crate::trap::yield_to_user(&mut
tf);

        …

        match handle_scause(scause, &mut ts, &mut tf).await {
            Continue(Some(sig)) => ts.task.sig.push(sig),
            Continue(None) => {}
            Break(code) => break 'life (code, None),
        }

        …

    };
    ts.cleanup(code, sig).await
}
```

从代码实现中我们可以发现，之前提及的 TaskState 并没有保存到某个全局集合里，而是保留全部的所有权作为一个局部变量的函数参数传入了该主函数，并在主函数退出时自动销毁。于是，我们在主函数中便可以如同访问普通的局部变量一般对待该结构体，包括在所有权框架内的安全可变访问，从而摒弃 UnsafeCell、Mutex 等不安全或低效的访问方式。如此，我们便解答了在上一节末尾的疑问。

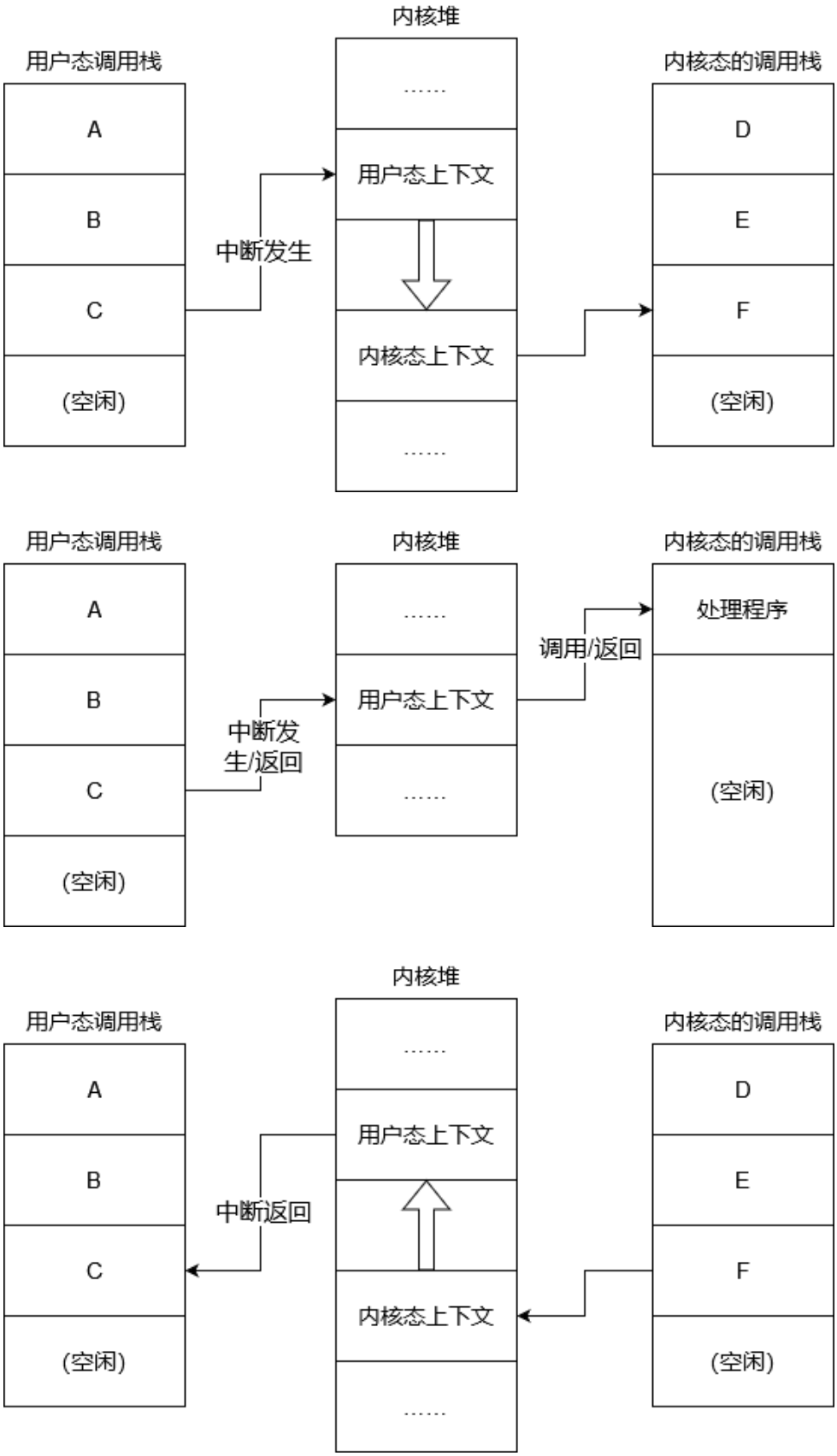
2.1.4 特权级切换

当中断发生的时候，一般的中断处理流程首先切换到内核的调用栈，然后保存好用户态的上下文。在这之后，会有两种方向：

1. 内核不保存上下文，于是直接通过函数调用的形式进入中断处理程序，在调用栈内创建一个暂时的子上下文；
2. 内核有自己的并行上下文，于是恢复上次中断保存的上下文进行中断处理，完成之后则保存内核上下文，二者是并行的上下文，没有调用层级关系。

在我们的观察中，以往大部分操作系统的处理方式都是前者。由于后者的处理方式将内核上下文和用户上下文的关系置于平行关系，相互切换的方式也是对偶的，实际上相互切换是一种有栈协程的方式，于是我们选择了第二种特权级切换方式。处理流程的对比如下图：

第一张图代表内核不保存上下文的切换方式，第二、三张图代表内核保存上下文的切换方式。



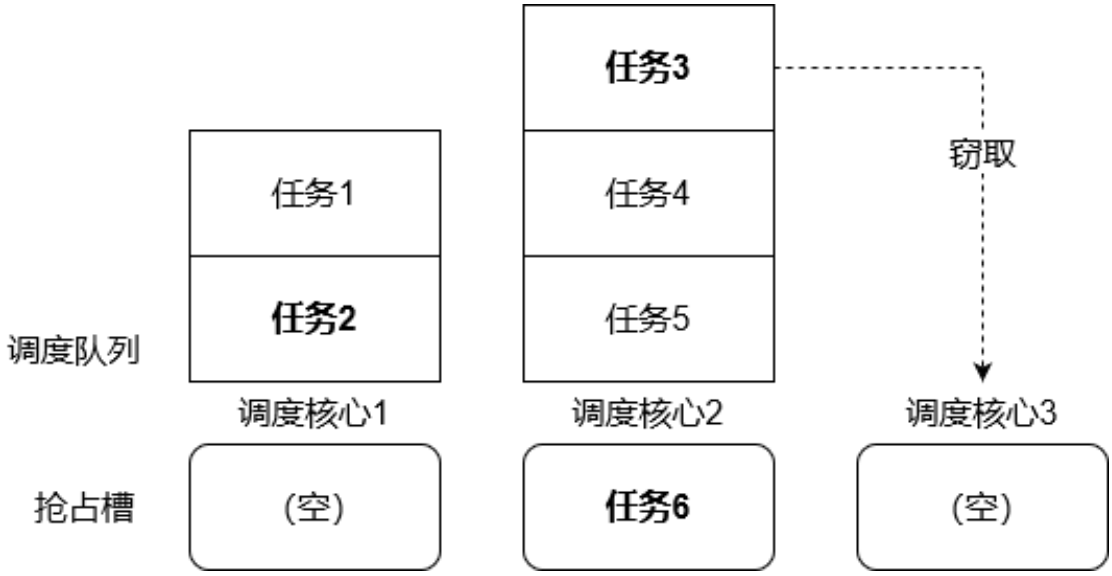
2.1.5 任务调度

在实际编写操作系统之前的调研学习中，我们认识到了 Rust 异步任务的调度核心代码是与特权级无关的。这样我们意识到，即使是操作系统的线程调度也可以借鉴目前市面上的优秀开源的异步运行时库的实现，比如 Tokio 和 async-task 等。于是，我们仿造 Tokio 的调度机制，基于 async-task 的代码，开发出了一套支持多核心、软抢占和任务窃取的任务调度器。

在调度器的实现中，每个调度核心拥有一个经典的 FIFO 任务队列，和一个抢占槽。每个调度核心的任务获取按照如下优先级顺序:

1. 抢占槽
2. 自身调度队列
3. 其他调度队列（窃取）

所有的操作都是无锁的实现。实现软抢占主要考虑了 I/O 事件唤醒和主动 yield 的调度信息差异，来加快 I/O 事件的响应速度；而任务窃取主要保证了多个调度核心之间的负载均衡性。由此来最大化地支持并行度。



具体的结构示例如上图，加粗任务为每个核心接下来要执行的任务。

在软抢占功能实现的过程中，我们发现 async-task 开源库的当时最新版本并不支持获得调度信息（是否在运行时被重复唤醒），于是我们向其合并了一个新的 PR 以支持该功能，目前已经被发布在了 4.4 及其之后的版本。

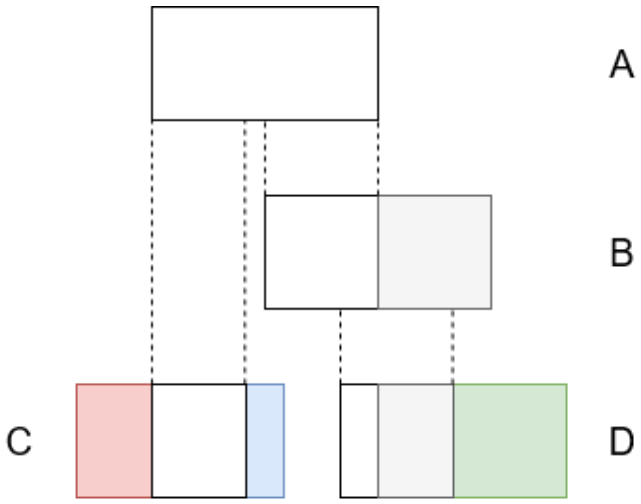
2.2 内存管理

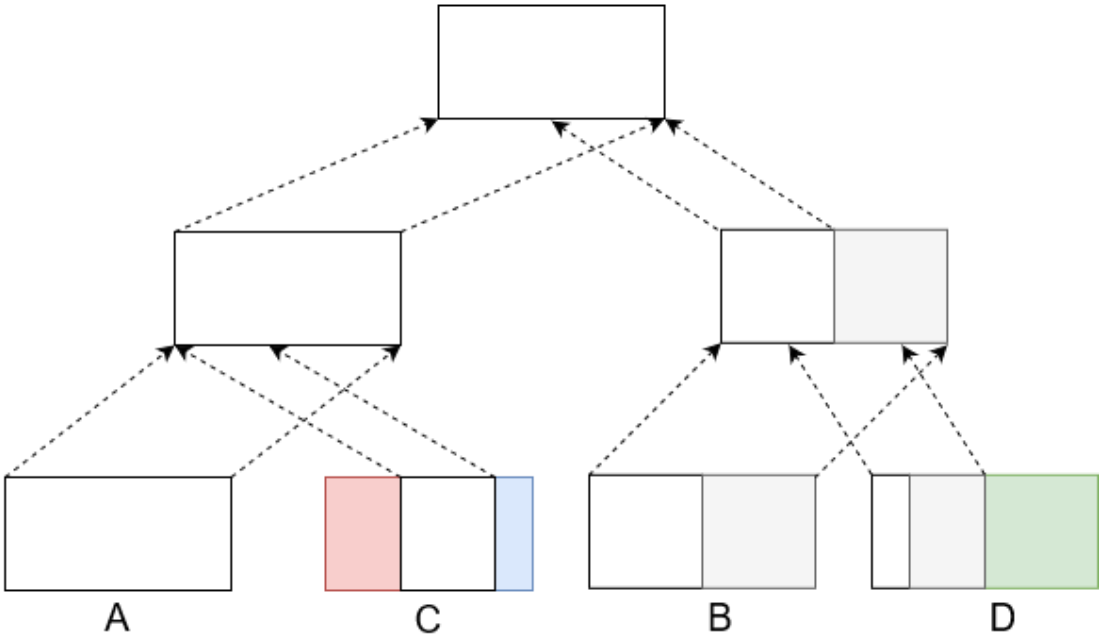
UMI 操作系统将内存管理子系统分成内核堆分配器、物理页帧管理和地址空间管理三部分，其中内核堆分配器是对开源库的简单引用，能够支持 Rust 内部的全局分配器工作，在此不再赘述。接下来将从物理页帧管理和地址空间管理两大块来讲解。

2.2.1 物理页帧管理

在内存管理的设计中，页帧管理一直是其中重要的一环。就好比物质的量之于物质微粒个数，内存页作为对内存操作的主要单元，占据着重要的作用。

在设计UMI的页帧管理模块的时候，我们充分对比了目前开源的主流操作系统的源码，并最终借鉴 Fuchsia 设计了一套 RAII 的基于二叉树形的数据结构。具体的结构如下图：





图中，上图为逻辑关系图；下图为实际存储关系图，箭头代表引用。同个颜色（内容）的页帧只存在一份副本。每一个页帧管理结构包含页帧哈希表、父节点（包含可能的 I/O 后端，比如文件或块设备）、刷新器（背景任务）、以及状态信息。

mizu/lib/kmem/phys.rs
`#[derive(Clone)]` `enum Parent {`

```
    Phys {
        phys: Arsc<Phys>,
        start: usize,
        end: Option<usize>,
    },
    Backend(Arc<dyn Io>),
}

#[derive(Debug)]
struct FrameList {
    parent: Option<Parent>,
    frames: HashMap<usize, FrameInfo, RandomState>,
}

#[derive(Debug)]
pub struct Phys {
    branch: bool,
    list: Mutex<FrameList>,
    position: AtomicUsize,
    cow: bool,
    flusher: Option<Flusher>,
}
```

每个节点逻辑上是其父节点的一个切片，有标志指示是否拥有写时复制（CoW）特性。页帧通过引用计数和缓存状态的更新在树形结构中复制和流动。例如在提交页帧的时候，我们依次从自身的哈希表、父节点的哈希表、I/O 后端读写、新清零页的顺序来依次访问并提交页帧。

通过如上说明我们可以看到，Phys 结构体可以作为任意读写+寻址的后端的页缓存，包括普通文件、块设备等。这样，虽然缺乏了一些特定场景的优化，我们可以避免重复的页缓存代码。并且由于写时复制和懒分配两个特性，任何涉及到内存分配的场景都会获得对应的性能提升。同时，每个节点可以实现同时读写页帧，在不浪费页帧的情况下提升页缓存的并发性。

2.2.2 地址空间管理

由于比赛对于内存大小没有极限的要求，UMI 中仅实现了 RISC-V64 位的 SV39 分页模式。一下均在此情况下讲解。

UMI 的用户态坐落于 39 位地址空间的下半，是按需私有的页表，内部的分配完全按照执行程序的需求进行，将在之后进行详细介绍；而内核空间则位于地址空间上半，采用添加偏移的恒等映射，为了简化分配使用 1GB 的大页，页表均设置有全局标志位。

内核空间启动之时，由于初始状态并没有设置页表，内核需要一小段桩代码来设置好页表和内核执行文件的相关参数并跳转到实际映射地址。并且尽管我们在编译的时候设置了位置无关代码（PIC）的选项，由于不同的代码执行位置引用保存的实际代码偏移也不同，可能会在某处计算偏移的时候产生混乱，因此需要尽快设置好页表跳转。至于内核栈等链接时

的配置，与其他比赛作品相差无几，在此不再赘述。

用户地址空间按照按需分配的原则，若指定地址则按地址分配，若仅指定大小则随机化查找分配，也即实现了用户全局的地址空间随机化（ASLR）功能，并不像 Linux 那般给每个程序映像设置了 mmap 区域，堆区域等等。

在处理页表映射的时候，秉持与页帧管理相同的原则，我们的页表映射也是懒分配、写时复制的。地址空间管理结构如下：

mizu/lib/kmem/virt.rs

```
struct Mapping {
    phys: Arsc<Phys>,
    start_index: usize,
    attr: Attr,
}

pub struct Virt {
    root: Mutex<Frame>,
    map: RwLock<RangeMap<LAddr, Mapping>>,
    cpu_mask: AtomicUsize,

    _marker: PhantomPinned, // Unused
}
```

我们建立映射的操作分为两个部分：创建映射（`map`）和提交映射（`commit`）。在创建映射的时候，我们仅对管理数据结构进行操作，而并不实际提交页表，等到发生页错误的时候才提交映射，而页表依赖的物理页帧提交也是在此时才实际进行。而进行重设页表标志位的时候，我们也仅仅做了取消映射与更改 `Mapping` 的 `attr` 字段，以最极致化懒分配的策略。

而在程序 fork 的时候，我们仅仅将数据结构进行克隆而不对实际的页表进行克隆。这也是为了减少大量加锁操作带来的并发度的减损。

在实际提交映射的时候有一类特殊情况需要考虑，那就是用户指针与用户缓冲区。

1. 用户指针

对于用户指针的读写，我们在提交映射之后，暂时修改页错误的处理函数标志位便可以直接解引用，发生了页错误便直接在原来的函数内返回错误标志即可。具体代码大致如下：

mizu/kernel/src/mem/user.rs

```
async fn checked_op(
    virt: &Virt,
    mut op: impl FnMut() -> usize,
    expect_attr: Attr,
) -> Result<(), Error> {
    extern "C" {
        fn _checked_ua_fault();
    }
    let _checked_ua_fault = _checked_ua_fault as _;
```

```
    let mut last_addr = None;
    loop {
        match UA_FAULT.set(&_checked_ua_fault, &mut op) {
            0 => break Ok(()),
            addr if last_addr == Some(addr) => break
Err(EFAULT),
            addr => {
                virt.commit((*last_addr.insert(addr)).into(),
expect_attr)
                    .await?
            }
        };
    }
}

scoped_thread_local!(pub static UA_FAULT: usize);
```

其中 UA_FAULT 存储的是处理用户指针对应的页错误处理函数，每个 CPU 核心一个，而_checked_ua_fault 便是返回发生页错误的异常地址。

2. 用户缓冲区

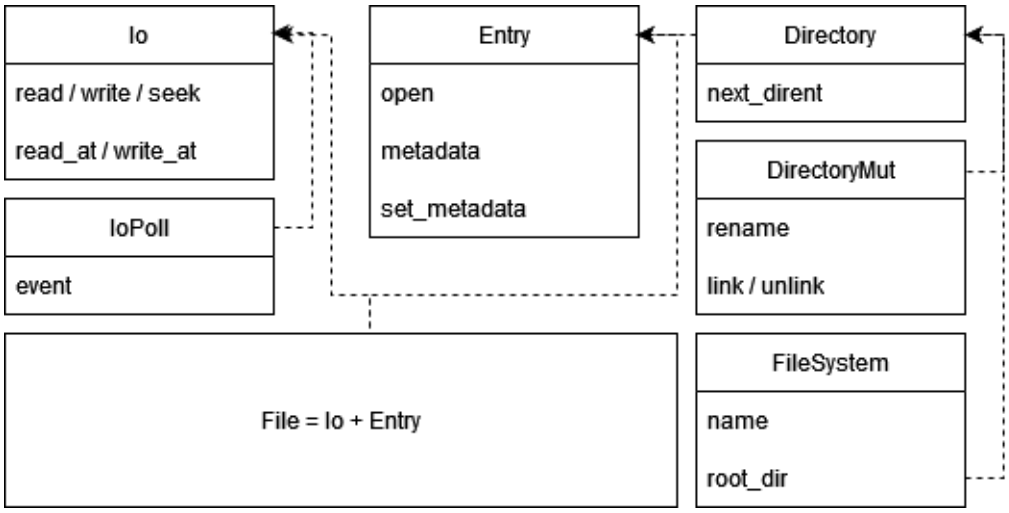
而对于用户缓冲区，有一个更难解决的问题存在：实际操作用户缓冲区的时机可能并不与提交映射的时机相同，有可能在使用时该缓冲区已经被另一个线程取消映射了。造成的页错误发生在内核空间，没有挽回的余地（二阶中断处理不能发生在异步上下文，因此异步的映射函数无法调用，错误的 I/O 操作无法中止）。

对于该问题，我们的解决办法是，在使用用户缓冲区的时候保持持有地址空间的读锁，使得其他线程无法改变地址空间，虽然在一定程度上拖慢了效率，但是增加了缓冲区安全性。

2.3 存储系统

2.3.1 虚拟文件系统

UMI 操作系统的文件系统接口从读写（I/O）接口开始，延展出一个 Rust trait 的集合，基本延续了面向对象的接口设计，并没有做过多的优化。具体关系如下，不再赘述：



2.3.2 FAT32

UMI 的 FAT32 文件系统实现是一个主要参考了 rust-fatfs 实现的异步并发的 FAT32 文件系统。通过将原本代码中的`RefCell`等结构换成了互斥等同步源语和引用计数指针，实现了同时读写不同文件的功能。在此简要介绍一下扩展的内部功能：批量操作 FAT。

FAT32 文件系统的文件分配表（FAT）是其核心结构，控制着簇（块）的分配。原本的实现中，一次只能从设备读取一条 FAT 条目。虽然有`Phys`对块设备页面缓存的加持，基于链表读写操作还是不够快。因此，我们引入了批量的功能，一次可以读取、更新至多 1024 条（1024*4=4096 刚好是一个页面）的条目，极大地加快了簇的分配以及依赖该功能的文件创建和写入功能。

mizu/lib/afat32/src/table.rs

```
impl Fat {
    /// # Safety
    ///
    /// The buf must be written zeros.
    async unsafe fn get_range_raw(
        &self,
        start: u32,
        buf: &mut [MaybeUninit<u32>],
    ) -> Result<usize, Error> {
        let end = (start +
u32::try_from(buf.len())?).min(self.allocable_range().end);
        if start > end {
            return Err(EINVAL);
        }
        if start == end {
            return Ok(0);
        }
        let read_len = (end - start) as usize;
```

```
        let bytes = MaybeUninit::slice_as_bytes_mut(&mut
buf[0..read_len]);

        self.device
            .read_exact_at(self.offset(0, start), unsafe {
                MaybeUninit::slice_assume_init_mut(bytes)
            })
            .await?;

        Ok(read_len)
    }

    pub async fn get_range<'a>(
        &self,
        start: u32,
        buf: &'a mut [u32],
    ) -> Result<impl Iterator<Item = (u32, FatEntry)> + Send
+ Clone + 'a, Error> {
        buf.fill(0);
        // SAFETY: init to uninit is safe.
        let len = unsafe { self.get_range_raw(start,
mem::transmute(&mut *buf)) }.await?;

        let zip = buf[..len].iter().zip(start..);
        Ok(zip.map(|(&raw, cluster)| (cluster,
FatEntry::from_raw(raw, cluster))))
    }
}
```

虽然有着以上的功能，但是不管怎么样批量，都是直接操作设备的数据。因此，一个未来可以优化的点是，直接设计一个内存中的 FAT 管理数据表，对簇的分配直接操作该表，并在合适的时机向设备同步数据。

2.4 网络系统

UMI 的网络系统建立于以开源库 smoltcp 位核心、embassy-net 为代码实现参考的网络协议栈，辅之以高并发度的异步无栈协程和中断驱动的支持，最终以多网络设备接口，支持 UDP、TCP、DNS、IP（v4+v6）的多协议网络协议栈的形态成型。

2.4.1 核心结构

网络协议栈的核心结构包括插座集合、接口封装和设备对象三个部分。具体结构如下：

mizu/lib/devices/src/net/stack.rs

```
pub struct Stack {
    devices: Vec<Arc<RwLock<dyn Net>>>,
    socket: RwLock<SocketStack>,
    states: RwLock<Vec<State>>,
}

#[derive(Debug)]
pub(in crate::net) struct State {
    link_up: bool,
    local_sockets: SocketSet<'static>,

    dhcpv4: Option<SocketHandle>,

    dns_socket: SocketHandle,
    dns_servers_ipv4: heapless::Vec<Ipv4Address, 3>,
    dns_servers_ipv6: heapless::Vec<Ipv6Address, 3>,
    dns_waker: AtomicWaker,
}

pub(in crate::net) struct SocketStack {
    pub sockets: SocketSet<'static>,
    next_local_port: u16,

    pub loopback: Tracer<Loopback>,
    pub ifaces: Vec<Interface>,

    waker: AtomicWaker,
}
```

其中 SocketStack 包含全局插座集合、本地端口分配、回环设备、网络接口封装等字段；State 是与网络设备意义对应的，包含设备本地的链路状态、插座集合、DHCP 配置、DNS 插座等字段；而公开结构类型 Stack 则是上述二者的超集，再包括网络设备对象的集合。

这么分立对象的原因是不同的操作对不同对象所需要的互斥条件不一致。通过精细地控制锁的粒度，可以最大化并发度，达到最好的效果。

2.4.2 配置与运行

如概述所述，UMI 的网络协议栈支持 IPv4+v6 的双版本配置。其中 IPv4 支持静态配置和动态（DHCP）配置，而 IPv6 由于 smoltcp 的实现原因仅支持静态配置。这对 smoltcp 的功能利用达到了最大化。

在比赛的决赛一阶段评测中，不论是华山派还是 QEMU 实际上都不需要实际的网络设备配置。但是为了支持 sshd，我们还是设置了如上配置。目前，QEMU IPv4 的静态和动态配置均已正确运行，遵循着 QEMU 的官方文档（通过配置 user 模式的虚拟网络设备，虚拟子网运行在 10.0.2.0/8 地址段之下）。

具体的配置代码涉及设置网络接口结构、更新 DNS 服务器等逻辑，在此不再单独叙述。

UMI 的网络协议栈运行在单独的异步任务之下。无论是用户端还是设备端，只要有一个操作导致网络协议栈中的某个状态改变，该异步任务就会被唤醒触发，进行一次轮询与状态更新。更新中会对每一个网络设备与其对应的网络接口和状态对象进行如下的操作:

mizu/lib/devices/src/net/stack.rs

```
let mut dev = dev.write();
Stack::update_interface(iface, &*dev);

let mut poller = Tracer::new(dev.with_cx(Some(cx)), |i, p|
writer(1, i, p));
iface.poll(instant, &mut poller, &mut self.sockets);
iface.poll(instant, &mut poller, &mut state.local_sockets);

let old = mem::replace(&mut state.link_up, dev.is_link_up());
if old != state.link_up {
    let s = if state.link_up { "up" } else { "down" };
    log::info!("Net device {:x?} link {s}", dev.address());
}

// Configure DHCPv4 of this interface if any.
```

其中 `Tracer` 是对网络设备对象的一层包装，用来追踪所有该网络设备上传输的数据包，正确设置日志相关的编译环境变量时不会对运行效率产生太大的影响。

对网络协议栈中进行了状态更新后，若对用户端或设备端有状态的改变，例如收包完成或发包开始等等，该异步任务也会唤醒对应的对象进行实际的操作。而在所有操作更新完成之后，该异步任务还会依据内部协议估算下一次进行轮询的时间，然后启动可能的定时器以触发之后的轮询。

UMI 网络协议栈中回环设备与普通的网络设备处于平等地位，对于上述的流程也会对等地执行一次，除了链路状态更新等操作。这使得回环的数据包不需要进行特殊的单独拦截处理，仅需目标插座正确绑定对应的网络接口即可。

2.4.3 网络插座

UMI 的网络插座接口主要参考了 embassy-net 的实现代码，依据 smoltcp 本身对唤醒对象已有的支持，手动实现无栈协程状态机函数，完成对网络插座接口的全异步包装。

DNS 插座的支持并没有与其他插座一般单独包装，而是被包含在网络协议栈的核心实现函数之中，以接口函数的形式访问。其中包括 `dns_start_query`、`dns_get_result`、`dns_cancel` 三个核心状态机函数，在此不再过多赘述，感兴趣的可以直接查看对应源码。

UDP 插座大体上是对 smoltcp 的一层简单包装。其中收包的核心状态机代码如下:

mizu/lib/devices/src/net/socket/udp.rs

```
// Content in `fn poll_receive()`:
if !self.bound.load(SeqCst) {
    self.bind(IpListenEndpoint::default())?;
}
```

```
loop {
    if let Some(res) = self.with_mut(|socket| match
socket.recv_slice(buf) {
        Ok((n, meta)) if Some(meta.endpoint) ==
socket.remote_endpoint() => {
            Some(Poll::Ready(Ok((n, meta.endpoint))))
        }
        Ok(_) => None,
        Err(udp::RecvError::Exhausted) => {
            socket.register_recv_waker(cx.waker());
            Some(Poll::Pending)
        }
    }) {
        break res;
    }
}

// Content in `async fn receive()`:
poll_fn(|cx| self.poll_receive(buf, cx)).await
```

首先判断是否已绑定端口，若无则绑定一个随机分配的端口。之后持续取缓冲区内容，若是收到了对应远程端口的包则直接返回有效，否则继续取包直到缓冲区耗尽，此时则注册唤醒对象并返回等待。

由于 smoltcp 内置的 UDP 插座并没有筛选对应远程端口的功能，也即无法维持伪连接的存在，更是在实际进行 iperf 测试之时发生了手法不均造成大量丢包的问题。因此，我们修改了 smoltcp 的源码，加入了远程端口的字段并添加了对应的处理代码以完成该功能。经过测试 iperf 的丢包率降到了 10%以下。

TCP 插座的实现相对复杂一些。由于 smoltcp 内置的 TCP 插座仅能维持单端口单连接，因此我们在我们的实现代码中添加了单独的 ACCEPT 队列及其背景任务，通过置换内部实例对象达到单端口多连接的功能。并且该功能将建立连接和创建连接插座的文件描述符两个操作分离，也减少了长时间对网络协议栈的锁的持有，增强了并发性。由于代码量过大，在此不做详解，具体可以直接查看对应代码仓库。

TCP 插座的 I/O 传输函数实现跟 UDP 大致相同，不过需要考虑状态的切换来判断是否注册唤醒对象。收包代码如下：

mizu/lib/devices/src/net/socket/tcp.rs

```
// Content in `fn poll_receive()`:
self.inner.with_mut(|_, s| match s.recv_slice(buf) {
    Ok(0) => {
        s.register_recv_waker(cx.waker());
        Poll::Pending
    }
    Ok(n) => Poll::Ready(Ok(n)),
    Err(tcp::RecvError::Finished) => Poll::Ready(Ok(0)),
    Err(tcp::RecvError::InvalidState) =>
Poll::Ready(Err(ECONNREFUSED)),
```

```
})

// Content in `async fn receive()`:
poll_fn(|cx| self.poll_receive(buf, cx)).await
```

需要注意的是，smoltcp 的返回值含义跟 POSIX 系统调用的返回值含义略有不同，实现时需要多加注意。

对于 UDP、TCP 两种插座的 I/O 事件监听，由于仅需要 TCP 状态或缓冲区空闲状态的查询以及唤醒对象的注册，在此不过多赘述。

2.5 驱动程序

由于经济原因，本次参赛我们队伍选择了华山派开发板，其使用的 CPU 是平头哥的 C906。我们在 QEMU 虚拟机和开发板上共实现了 UART、Virt IO 块设备、Virt IO 网络设备、SD 卡、PLIC 等驱动程序。其中 UART 过于简单，在此不做详解。

2.5.1 FDT

不同于静态初始化所有平台的硬件配置，我们采用了动态传入 FDT（Flattened DeviceTree）数据结构的方式来初始化设备。

FDT 是一种二进制格式，保存硬件平台的设备树信息，一般通过 u-boot 等启动软件或虚拟机本身向操作系统提供。在 RISC-V64 架构上，操作系统在执行入口的 a0 寄存器便可收到保存在内存中的 FDT 地址。

我们使用了开源库 fdt 来访问 FDT 相关结构，并通过专门的设备初始化表来执行对应设备初始化的接口，以减少对项目 feature 的控制。

mizu/kernel/src/dev.rs

```
static DEV_INIT: Lazy<Handlers<&str, (&FdtNode, &Fdt), bool>>
= Lazy::new(|| {
    Handlers::new()
        .map("ns16550a", serial::init_ns16550a)
        .map("snps,dw-apb-uart", serial::init_dw_apb_uart)
        .map("riscv,plic0", intr::init_plic)
        .map("virtio,mmio", virtio::init_mmio)
        .map("cvitek,mars-sd", sdmmc::init)
});
```

2.5.2 PLIC

PLIC 是 RISC-V 平台使用最广泛的内存映射的外部中断控制器，支持多核同时访问。PLIC 的中断处理核心编号与 HART ID 并不总是恒等映射，需要查看开发板手册来确定。

PLIC 的功能大致分为中断注册和中断处理两个部分。

中断注册时，需要找到对应中断编号和中断处理核心编号的内存映射寄存器的比特位置

位，还需要顺带设置该中断的优先级（一般 0 表示禁用），以便在对应的中断处理核心上启用该中断。

硬件中断发生的时候，PLIC 会转发中断信号至所有启用该中断的处理核心，每个核心都会进入*tvec 设置好的中断处理程序中。但是只有最先在对应核心的中断响应的寄存器中读出该中断编号的中断处理核心才能处理中断。一旦有核心读出该中断，则后续其他核心的读取操作的结果都将为 0（或其他优先级更低的待处理中断编号）。当中断处理完成之后，处理核心再将该中断编号写入对应的寄存器以更新 PLIC 的状态，完成操作。

PLIC 的标准可以参考处于 Github 上的官方文档（riscv/riscv-spec-plic）。

2.5.3 Virt IO 块设备

Virt IO 块设备的驱动程序实现遵循基本的提交-完成异步流程的状态机编写。用户向驱动提交读写操作时，将缓冲区等参数存放于设备的提交队列中，之后陷入等待。读写操作完成后设备收到中断，将缓冲区交还给用户后唤醒用户，最后用户收到缓冲区完成流程。具体状态机代码如下：

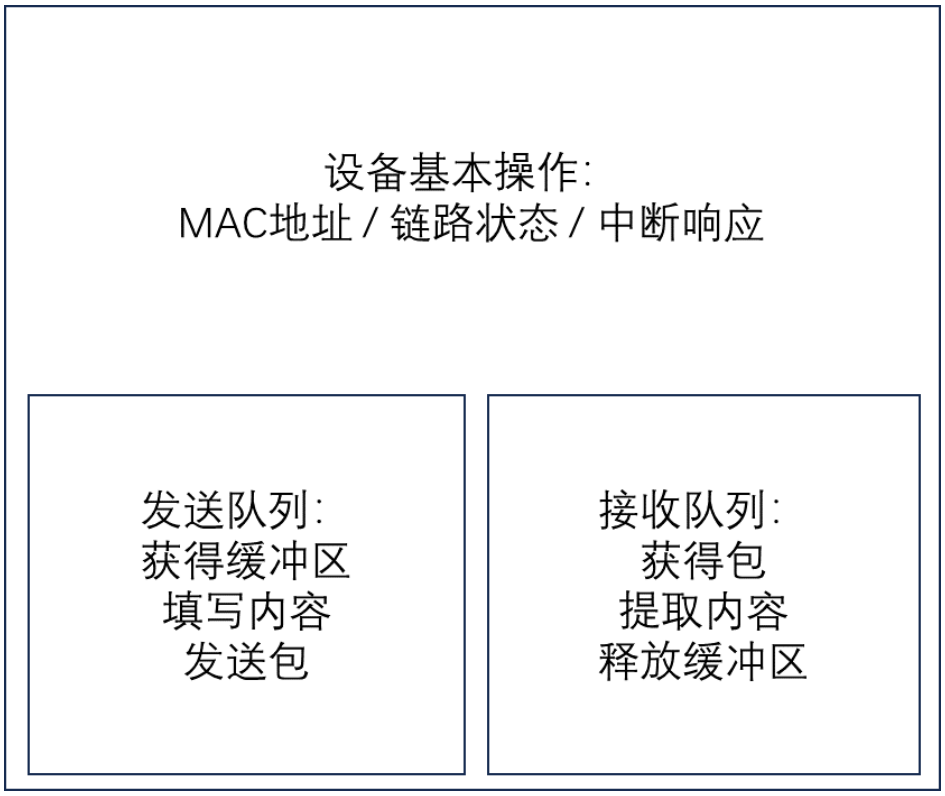
mizu/dev/virtio/src/block.rs

```
loop {
    match &mut this.state {
        ChunkState::Submitting { req, resp, buf, listener }
=> {
            let res = ksync::critical(|| {
                // Submit the request. Returns the receiver
for its completed transfer buffer.
            });

            match res {
                Ok(rx) => this.state = ChunkState::Waiting
{ ret: rx },
                Err(virtio_drivers::Error::QueueFull) => {
                    match listener {
                        Some(l) => {
                            ready!(l.poll_unpin(cx));
                            *listener = None;
                        }
                        None => *listener =
Some(this.device.virt_queue.listen()),
                    }
                    continue;
                }
                Err(err) => break Poll::Ready(Err(err)),
            };
        }
        ChunkState::Waiting { ret } => {
```

```
            let buf = ready!(ret.poll_unpin(cx));

            this.state = ChunkState::Complete;
            break Poll::Ready(buf.map_err(|_|
virtio_drivers::Error::DmaError));
        }
        ChunkState::Complete => unreachable!("polling after
complete"),
    }
}
```

2.5.4 网络设备的接口

网络设备的接口与块设备不同。由于发送端和接收端均不需要等待操作的完成，因此其接口函数并不是异步函数。其接口对象分成三部分抽象，独立成 trait：设备信息、发送队列、接收队列。具体的结构如下：



其中，发送队列和接收队列两个接口对象通过令牌来管理其内部缓冲区。

通过如上抽象，我们可以最大力度地细化网络设备的每一个操作。比如，我们可以同时持有发送队列和接收队列两个对象，并**并行地**对二者进行操作；又或者，我们同时有两个等待发送的数据包，我们便可以**并发地**同时获得两个包的缓冲区，同时填写内容，最后依次将其发送。这样做，我们便取得了高并发的效果。

而对于 Virt IO 网络设备的实现，由于接口函数的操作均是细粒度操作，内部并没有太多逻辑组织的空间，因此在此不再赘述。

2.5.5 SD 卡

华山派 cv1811h 的 SD 卡设备是标准的 MMIO SD Host Controller 再加一点自定义的 PINMUX 控制寄存器。其数据和命令交互的方式除了通过其他一些开发板也有的 SPI，还可以通过 MMIO 寄存器。具体的标准可以查看 SDHCI 的官方文档。

UMI 实现的 SD 卡驱动并不采用 Data Port 寄存器的方式来数据传输，而是采用了 ADMA2 的方式。ADMA 第二版本是 SDHCI 标准规定的一种依靠 DMA 来传输数据的方式。在其实现中，CPU 通过内存中的描述符给设备提交 I/O 操作，设备则在 I/O 操作完成时通过中断通知 CPU。

以下是发送命令的部分函数代码：

mizu/dev/sdmmc/src/imp.rs

```
// Content in `async fn send_data()`.
let block_shift =
data.block_shift.unwrap_or(self.block_shift);
let block_size = u12::new(1 << block_shift).ok_or(EINVAL)?;

let len = ((data.block_count as usize) <<
block_shift).min(DescTable::MAX_LEN);

let buffer = data.buffer.get_mut(..len).ok_or(EINVAL)?;
let filled = unsafe { self.dma_table.fill(buffer,
data.is_read) };

self.data_slot = Some(DataSlot {
    buffer: mem::take(&mut data.buffer),
    bytes_transfered: filled,
    is_read: data.is_read,
    res: None,
});
// Set registers to enable data parameters, ADMA2 and
interrupts.
Ok(())

// Content in `async fn send_cmd()`.
if !self.is_present() {
    return Poll::Ready(Err(ENODEV));
}

let has_data = data.is_some();
let is_busy = cmd.cmd == stop_transmission().cmd;

ready!(self.poll_inhibit(cx, has_data || is_busy));
```

```
let transfer_mode = if let Some(data) = data {
    self.send_data(data)?;
    let regs = &mut self.regs;
    // Returns the set-up transfer mode register for
transferring data.
} else {
    let regs = &mut self.regs;
    map_field!(regs.transfer_mode).read()
& !TransferMode::AUTO_CMD_MASK
}
self.set_timeout(is_busy);

let regs = &mut self.regs;
// Write registers to send the command.
Poll::Ready(Ok(()))
```

通过 ADMA2 的方式，使得提交 I/O 操作和完成 I/O 操作分离，细粒度化操作，可以增强其并发性。

2.6

2.6 其他功能和子系统简述

由于其他子系统并没有过多的特色需要我们介绍，仅是保证基本的功能可用和代码可读性，因此在此我们并不对其作详解，而是简要讲述为止。

2.6.1 信号系统

UMI 的信号系统采用经典的事件处理方式，将每个信号单独建立事件来订阅和通知。定义了四种信号处理程序类型：忽略、杀死当前进程、挂起和用户态处理。

当待处理信号为用户态处理程序时，内核保存当前的用户上下文到当前的信号栈（若无则为用户调用栈 sp）上，并重新初始化信号处理的上下文，其中 ra 设置为一个特殊的在 39 位地址空间之外的非法地址，称为哨兵地址。当用户处理程序范围时，发生该哨兵地址的页错误，被内核识别到后便恢复到正常的信号处理退出流程，恢复原来的用户态上下文。

2.6.2 时间管理

UMI 实现了最基本的定时器及其队列，默认实现了异步接口，可以与所有的异步任务组合使用。详情请查看 futures-util 开源库。

所有定时器被包含在由 Rust 标准库的 B 树映射表实现的全局定时器队列内。当时钟中断发生时，内核调用对应的定时器队列状态更新代码，更新所有可能的定时器状态。

UMI 定时器的用户接口为了避免大量对全局定时器队列的互斥访问，添加了注册批处理机制，默认将请求存放于无锁的请求队列中，当请求队列满则一起清空完成所有待处理请求。

2.6.3 系统调用

UMI 的系统调用表使用专门实现的泛型函数哈希表。部分使用代码如下:

mizu/kernel/src/syscall.rs

```
pub type ScParams<'a> = (&'a mut TaskState, &'a mut
TrapFrame);
pub type ScRet = ControlFlow<i32, Option<SigInfo>>;

pub static SYSCALL: Lazy<AHandlers<Scn, ScParams, ScRet>> =
Lazy::new(|| {
    AHandlers::new()
        .map(BRK, crate::mem::brk)
        .map(FUTEX, crate::mem::futex)
…
        .map(UMASK, dummy_umask)
});
```

可以看到, 该结构与设备树初始化的函数表几乎完全相同。实际上两者也是同一个实现。该函数哈希表可以将任意同步、异步的函数作为存储单元, 并且每个存储单元的函数参数只要满足默认转换条件可以任意选择。

三、开发问题与解决

虽然在开发初期便已经将大部分结构进行了初步设计，但在开发过程中仍然会碰到一些较为繁琐的问题。在此列举一些较为典型的事例：

3.1 UDP 丢包

问题：在使用`smoltcp`时，`iperf`测试的 UDP 的丢包率一直居高不下（20%左右）。经过排查发现是`smoltcp`中的 UDP 收包逻辑较为简单，没有远程端口的参与，而`iperf`使用多个插座连接同一个服务器端口，于是经常出现收发不均衡的现象，造成某些插座缓冲区满而丢包。

解决：在`smoltcp`中加入了伪远程端口字段，使得在 UDP 收包时优先匹配远程端口。经过测试将丢包率降到了个位数。还在`smoltcp`中增加了检测收发缓冲区空闲大小的函数，以支持其 poll 事件功能。

3.2 用户缓冲区

问题：实际操作用户缓冲区的时机可能并不与提交映射的时机相同，有可能在使用时该缓冲区已经被另一个线程取消映射了。造成的页错误发生在内核空间，没有挽回的余地（二阶中断处理不能发生在异步上下文，因此异步的映射函数无法调用，错误的 I/O 操作无法中止）。

解决：在使用用户缓冲区的时候保持持有地址空间的读锁，使得其他线程无法改变地址空间，虽然在一定程度上拖慢了效率，但是增加了缓冲区安全性。

3.3 内核堆内存泄露

问题：在运行测试 clone 系统调用的时延的测例时，出现了内核堆的内存溢出问题。经过查看，发现是因为地址空间管理对象的页表未正确释放。由于 Rust 目前不支持 async drop，只能手动处理相关资源，因此未正确处理才导致了这个问题。

解决：正确处理了地址空间管理对象的资源释放时机，并将其管理的根页表从内核堆分配切换成专门的页帧分配器（也是其他页表默认的分配器），从而解决了该问题。然而，其他结构也可能导致内存泄漏，例如信号系统的事件对象（本身一个事件对象需要几百字节，而信号一共有 64 种），未来还可以继续优化。

四、总结与展望

我们 PLNTRY 队伍经过 5 个月的紧张工作，编写出了一个以并发与并行为主要设计目标，具有异步无栈协程、多核、模块化等诸多特性的 RISC-V64 位平台的实验性操作系统。

在操作系统实现的过程中，我们遇到了许多问题，诸如 UDP 丢包、内存泄漏等等，但是经过不断地思考与尝试，我们最终解决了大部分问题，通过了初赛的所有测试与决赛的大部分测试。

在比赛落下帷幕后，我们还将继续维护这个项目一段时间，其中工作包括但不限于解决遗留的一些 bug、完善 sshd 等服务程序的支持、完善代码注释等等。我们衷心希望这个项目不仅可以提升我们自身的技术水平，还能为广大涉猎操作系统领域的同学们提供学习参考。在未来，这个项目的一些思想可能会被合并进更多更优秀的项目，让我们拭目以待那一天的到来。