



哈爾濱工業大學(深圳)

HARBIN INSTITUTE OF TECHNOLOGY, SHENZHEN

MankorOS

设计文档

参赛队名 MankorOS

队伍成员 满洋、梁韬、苏亦凡

指导老师 夏文、仇洁婷

2023 年 6 月 3 日

摘要

MankorOS 是 Rust 编写的基于 RISC-V 的多核异步宏内核操作系统。

目前（2023.05.27）已满分通过初赛所有测试用例，下图为排行榜截图：

#	用户名	队伍	最后提交时间(ASC)	提交次数(ASC)	rank
1	202310464101015	你说对不队/ 河南科技大学	2023-04-18 14:30:26	22	102.0000
2	202318123101314	Titanix/ 哈尔滨工业大学（深圳）	2023-05-04 21:50:35	17	102.0000
3	202310698101003	PLNTRY/ 西安交通大学	2023-05-12 01:43:31	9	102.0000
4	202314430101195	编写吧!NutOS/ 中国科学院大学	2023-05-19 15:39:44	17	102.0000
5	202318123101332	MoOS/ 哈尔滨工业大学（深圳）	2023-05-19 17:09:27	28	102.0000
6	202310336101112	LostWakeup/ 杭州电子科技大学	2023-05-20 21:25:24	5	102.0000
7	202310336101111	BiteTheDisk/ 杭州电子科技大学	2023-05-22 22:55:01	5	102.0000
8	202310487101114	AVX/ 华中科技大学	2023-05-22 23:21:08	17	102.0000
9	202318123101282	MankorOS/ 哈尔滨工业大学（深圳）	2023-05-23 18:39:43	12	102.0000

图 0-1 初赛排行榜

下表为 MankorOS 各模块的完成情况：

模块	完成情况
无栈协程基建	基于全局队列实现的调度器, 可供异步程序执行
内存管理	实现 mmap/munmap 系统调用, 可对所有内存段进行懒分配或懒加载, 具备写时复制 (CoW) 功能
文件系统	完成虚拟文件系统, 支持 devfs 和管道
进程管理	支持 clone 系统调用, 可以细粒度划分进程共享的资源
信号机制	完成基础的信号机制
用户程序	能通过所有初赛测试样例

表 0-1 模块完成情况

目 录

摘要	I
1 概述	1
1.1 MankorOS 介绍	1
1.2 MankorOS 整体架构	1
1.3 MankorOS 项目结构	2
2 物理内存管理	2
2.1 内核动态内存分配器	2
2.2 物理页分配器	3
2.3 页表管理	3
3 基于无栈协程的异步	5
3.1 基础概念	5
3.2 内核实现要点	11
3.3 上下文切换	14
4 文件系统	15
4.1 VFS 设计与结构	15
4.2 设备文件系统 (devfs)	16
4.3 FAT32 文件系统	16
4.4 Pipe 与 Stdio	16
5 进程设计	17
5.1 进程和线程	17
5.2 地址空间	20
5.3 进程调度	23
6 同步	24
6.1 内存模型和缓存一致性	24
6.2 锁	25
6.3 进程间通信	27
7 系统调用	27
7.1 内存相关系统调用	27

7.2 文件系统相关系统调用	28
7.3 进程相关系统调用	28
7.4 其他系统调用	29
8 总结与展望	30
8.1 初赛测评	30
8.2 实现情况	31
8.3 未来工作	31

1 概述

1.1 MankorOS 介绍

MankorOS 是 Rust 编写的基于 RISC-V 的多核异步宏内核操作系统，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，拥有虚拟文件系统和设备文件系统。

在开发过程中，MankorOS 注重代码质量和规范性，确保每次 commit 都有意义的 commit message，并能够通过编译。这种严格的开发流程使 MankorOS 有高质量的代码，还可以减少 bug 的产生，从而提高项目的稳定性。

为了优化编译的效率，MankorOS 采用了 monorepo 的组织方式。这种方式将所有相关的代码和库都放在同一个仓库中，避免了不同组件之间的版本冲突和依赖问题。这种方式还可以提高代码的可读性和可维护性，可供教学或交流学习。

未来，MankorOS 将会在不牺牲性能的前提下，继续维持高质量的代码，目标成为一个高性能、易学习的操作系统。

1.2 MankorOS 整体架构

```
.
├── src
│   ├── arch          # 汇编与平台相关的包装函数
│   ├── axerrno       # 错误处理
│   ├── consts        # 常量
│   │   └── platform
│   ├── driver        # 驱动，包括块设备和 uart 驱动
│   ├── executor      # 管理 future 执行
│   ├── fs            # 文件系统
│   │   ├── devfs     # 设备文件系统
│   │   └── vfs       # 虚拟文件系统
│   ├── memory        # 内存
│   │   ├── address   # 地址类型
│   │   └── pagetable # 页表
│   ├── process       # 进程
│   │   └── user_space
│   ├── signal        # 信号
│   ├── sync          # 同步，一致性、锁和进程通讯
│   ├── syscall       # POSIX 系统调用
│   ├── timer         # 时钟，时钟中断管理和计时器
│   ├── tools         # 工具类型和函数
│   ├── trap          # 中断
│   └── xdebug        # debug 工具
```

```

├── boot.rs      # 启动
├── lazy_init.rs # 懒加载
├── logging.rs   # 日志
├── main.rs
├── utils.rs
├── vendor
├── Cargo.toml
├── LICENSE
├── linker.ld
├── Makefile
├── README.md
└── rust-toolchain.toml

```

1.3 MankorOS 项目结构

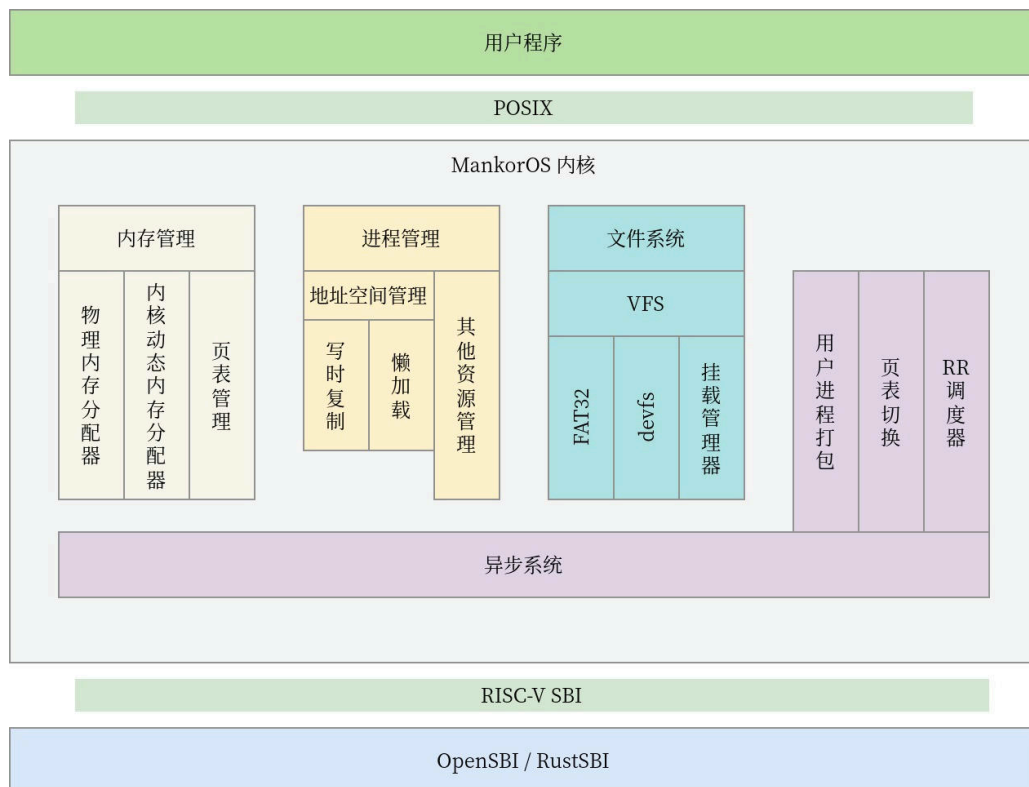


图 1-1 MankorOS 系统结构图

2 物理内存管理

2.1 内核动态内存分配器

为了使用 Rust 中的各种动态内存结构，需要在内核中实现一个动态内存的分配器。MankorOS 的内核动态内存分配器使用了 Buddy allocator，它来自 crate `buddy_system_allocator`。

Buddy allocator 是操作系统中一种常用的物理页分配算法，它的主要原理是将可用的物理内存按照 2 的幂次方进行划分，每个划分成为一个“伙伴块”，并根据伙伴块的大小将它们组织成一棵二叉树。当需要分配一个指定大小的物理内存时，buddy allocator 首先找到最小的 2 的幂次方大小的伙伴块，然后检查该伙伴块是否已经被分配出去。如果该伙伴块已经被分配出去，则继续寻找下一个伙伴块，直到找到空闲的伙伴块。如果找到了一个空闲的伙伴块，那么就将该伙伴块标记为已分配，并把它从可用伙伴列表中移除。接着，将该伙伴块逐级向上合并，直到合并到大于等于分配请求大小的伙伴块为止。这样，最终的合并后的伙伴块就可以满足分配请求。如果在合并过程中发现其中某个伙伴块已经被分配出去，那么就停止合并，将剩余的子伙伴块重新加入可用伙伴列表中。

当需要释放已经分配出去的物理内存时，buddy allocator 会将该内存块标记为未分配并加入可用伙伴列表中。接着，它会检查该内存块所处的伙伴块是否也是未分配状态。如果该伙伴块的另一个子伙伴块也是未分配状态，那么就将这两个伙伴块合并成一个更大的伙伴块，并继续向上检查合并后的伙伴块是否可以再次和其它空闲伙伴块合并。

这样，通过不断地进行伙伴块的合并和分裂，buddy allocator 可以高效地管理可用的物理内存，避免了内存碎片化和空间浪费，提高了内存空间利用率和系统性能。

MankorOS 内核的初始化动态内存区位于一个 `.bss` 段的数据区，在物理内存页分配器未初始化好之前，能够提供有限的动态内存，满足初始化时内核的需求。

2.2 物理页分配器

内核还需要管理全部的空闲物理内存，MankorOS 为此使用了来自 rCore 的仓库的 bitmap allocator。

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。

MankorOS 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。

2.3 页表管理

2.3.1 启动阶段

简单起见，MankorOS 并没有实现内核搬运等功能，而是直接在编译时将内核直接链接到高地址空间。这带来了一个问题，在未配置好地址翻译的时候，不能进入 Rust 执行，也就是需要在汇编语言尽快打开地址翻译。

MankorOS 设计了一个 boot 页表，嵌入在内核映像的 `.data` 段，具体如下：

```
"
.section .data
.align 12
_boot_page_table_sv39:
# 0x00000000_00000000 → 0x00000000 (1G, VRWXAD) for early console
.quad (0x00000 << 10) | 0xcf
.quad 0
# 0x00000000_80000000 → 0x80000000 (1G, VRWXAD)
.quad (0x80000 << 10) | 0xcf
.zero 8 * 507
# 0xffffffff_80000000 → 0x80000000 (1G, VRWXAD)
.quad (0x80000 << 10) | 0xcf
.quad 0
"
```

boot 页表使用了 huge page，直接将内核映像映射到正确的高位地址。

2.3.2 打开分页

使用汇编直接设置页表并打开修改地址翻译模式：

```
unsafe extern "C" fn set_boot_pt(hartid: usize) {
    core::arch::asm!(
        "    la    t0, _boot_page_table_sv39
          srli t0, t0, 12
          li    t1, 8 << 60
          or    t0, t0, t1
          csrw satp, t0
          ret
        ",
        options(noreturn),
    )
}
```

内核初始化结束后，低地址空间中的映射将被删除，留给用户程序使用。

```
pub fn unmap_boot_seg() {
    let boot_pagetable = boot::boot_pagetable();
    boot_pagetable[0] = 0;
    boot_pagetable[2] = 0;
}
```


2.3.3 页表的创建与回收

新建进程的页表时，我们一般希望内核区域的映射能够共享。因此，MankorOS 在新建用户进程的页表时，会直接复制 boot 页表的内核段。由于 boot 页表没有用户段的映射，因此直接复制是安全的。

```
pub fn new_with_kernel_seg() → Self {
    // Allocate 1 page for the root page table
    let root_paddr: PhysAddr = Self::alloc_table();
    let boot_root_paddr: PhysAddr = boot::boot_pagetable_paddr().into();

    // Copy kernel segment
    unsafe {
        root_paddr.as_mut_page_slice().
            copy_from_slice(boot_root_paddr.as_page_slice())
    }

    PageTable {
        root_paddr,
        intrm_tables: vec![root_paddr],
    }
}
```

页表回收时，MankorOS 利用 Rust 的 RAII 机制实现了进程结束的内存自动回收。该页表使用一个 **Vec** 保存所有它映射的物理内存页，当页表离开生命周期时，对应的物理页将会被自动释放 (返还给物理页分配器)。

```
impl Drop for PageTable {
    fn drop(&mut self) {
        // shared kernel segment pagetable is not in intrm_tables
        // so no extra things should be done
        for frame in &self.intrm_tables {
            frame::dealloc_frame((*frame).into());
        }
    }
}
```

3 基于无栈协程的异步

3.1 基础概念

3.1.1 **Future** 之间的组合

无栈协程的核心是 **Future**：

```
pub trait Future {
    type Output;
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) → Poll<Self::Output>;
}
```

Future 是异步函数的抽象。调用 **poll** 方法代表“检查任务结果”，如果任务已经完成，则返回 **Poll::Ready** (其中包含结果数据)，否则返回 **Poll::Pending** 表示任务尚未结束。

```
struct IdFuture {
    result: i32
}
impl Future for IdFuture {
    type Output = i32;
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) → Poll<Self::Output>
    {
        if /* some condition */ {
            Poll::Ready(self.result)
        } else {
            Poll::Pending
        }
    }
}
impl IdFuture {
    pub fn new(result: i32) → Self {
        Self { result }
    }
}
```

Future 之间可以组合。下面的 **Future** 实现了将一个 **Future** 的结果乘以 2 的功能：

```
struct DoubleFuture {
    a: IdFuture,
}
impl Future for DoubleFuture {
    type Output = i32;
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) → Poll<Self::Output>
    {
        let this = unsafe { self.get_unchecked_mut() };

        // 调用 a 的 poll 方法，如果 a 返回 Poll::Pending，则返回 Poll::Pending
        let a = unsafe { Pin::new_unchecked(&mut this.a) };
        let ar = match a.poll(cx) {
            Poll::Ready(x) ⇒ x,
            Poll::Pending ⇒ return Poll::Pending,
        };

        // 如果 a 返回了 Poll::Ready，就返回最终结果 Poll::Ready(ar * 2)
    }
}
```

```

        Poll::Ready(ar * 2)
    }
}
impl DoubleFuture {
    pub fn new(x: i32) → Self {
        Self { a: IdFuture::new(x) }
    }
}

```

可以看到, `Future` 的组合是通过对于 `Future` 的 `poll` 方法结果进行简单的组合得到的: 只要子 `Future` 返回 `Poll::Pending`, 则父 `Future` 也返回 `Poll::Pending`. 于是我们可以让编译器代我们生成上述代码:

```

async fn double(x: i32) → i32 {
    let ar = IdFuture::new(x).await;
    ar * 2
}

```

一个异步函数的上下文可以被保存在具体的 `Future` 结构体中, 从而使得函数可保存其上下文状态并恢复之。比如下面的 `Future` 实现了将两个 `Future` 的结果相加的功能:

```

struct AddFuture {
    status: usize,
    x: i32,
    y: i32,
    a1: IdFuture,
    a2: IdFuture,
}
impl Future for AddFuture {
    type Output = i32;
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) → Poll<Self::Output>
    {
        let this = unsafe { self.get_unchecked_mut() };
        loop {
            // 使用状态机的方式实现
            match this.status {
                AddFuture::STATUS_BEGIN ⇒ {
                    let a1 = unsafe { Pin::new_unchecked(&mut this.a1) };
                    let ar = match a1.poll(cx) {
                        Poll::Ready(x) ⇒ x,
                        Poll::Pending ⇒ return Poll::Pending,
                    };
                    // 保存局部变量
                    this.x = ar;
                    // 修改状态
                    this.status = AddFuture::STATUS_A1;
                }
                AddFuture::STATUS_A1 ⇒ {
                    let a2 = unsafe { Pin::new_unchecked(&mut this.a2) };

```

```

        let ar = match a2.poll(cx) {
            Poll::Ready(x) => x,
            Poll::Pending => return Poll::Pending,
        };
        // 保存局部变量
        this.y = ar;
        // 修改状态
        this.status = AddFuture::STATUS_A2;
    }
    AddFuture::STATUS_A2 => {
        // 返回最终结果
        return Poll::Ready(this.x + this.y);
    }
    _ => unreachable!()
}
}
}
}
const UNINIT: i32 = 0;
impl AddFuture {
    const STATUS_BEGIN: usize = 0;
    const STATUS_A1: usize = 1;
    const STATUS_A2: usize = 2;

    pub fn new(x: i32, y: i32) -> Self {
        Self {
            status: AddFuture::STATUS_BEGIN,
            x: UNINIT,
            y: UNINIT,
            a1: IdFuture::new(x),
            a2: IdFuture::new(y)
        }
    }
}
}

```

注意我们不能直接顺序地调用两个 `Future` 的 `poll` 方法并检查, 因为一个 `Future` 的 `poll` 方法可能会被执行多次。比如第一次 `poll` 时, `a1` 返回了 `Ready` 但 `a2` 返回了 `Pending`, 此时整个 `poll` 也应该返回 `Pending`. 但是当第二次调用 `poll` 时, 我们显然不能再重复去调用 `a1` 的 `poll` 方法, 所以我们需要保存“我们已经执行过 `a1.poll` 了”这个状态, 同时还要保存 `a1.poll` 方法的返回值。于是我们可以总结出, 每次对子 `Future` 的 `poll` 方法调用, 都需要产生一个“保存点”, 并且还需要在这里存下 `poll` 方法的返回值。这种规则也是非常机械的, 我们还是可以交给编译器, 让它在编译时为我们自动生成相似的代码:

```

async fn add(x: i32, y: i32) -> i32 {
    let a1 = IdFuture::new(x).await;

```

```
let a2 = IdFuture::new(y).await;
a1 + a2
}
```

3.1.2 Future 的边界

上面的论述了 `Future` 的组合方式，但要想写出真实可用的异步程序，我们还缺少两个部分：最初和最后的 `Future`。而这两部分都与 `Future::poll` 方法的第二个参数 `Context` 息息相关。我们先来考虑如何在一个普通的 `main` 函数中调用我们刚刚写的 `Future`。假设我们通过了某种魔法操作获得了一个平凡的，不起作用的 `Context` 对象，我们应该会这样使用 `AddFuture`：

```
pub fn main() {
    let mut ctx: Context = /* some magic here */;
    let result = AddFuture::new(1, 2).poll(&mut ctx);
    match result {
        Poll::Ready(x) => println!("result: {}", x),
        Poll::Pending => todo!(),
    }
}
```

那么，当我们调用一个 `Future` 的 `poll` 方法时，如果它返回了 `Pending`，应该怎么办呢？一种直接的解法是直接写一个 `loop` 循环，持续调用 `poll` 方法直到其返回 `Ready` 为止。但这样做有一个问题：我们的 `main` 函数会被阻塞在一个 `Future` 上。初看这似乎不是什么问题，但如果我们需要等待多种资源去完成多个任务时，阻塞在一个 `Future` 上就会变得非常低效。尤其是当我们可以让资源在就绪时调用某个回调函数来通知我们，而不需要我们去轮询它们的就绪状态时，我们会很自然地想到：能不能做一个队列，当一个 `Future` 执行到最后，卡在需要获取某个资源时，我们让它设置好该资源的回调函数，在资源就绪时重新将最开始的 `Future` 放回队列中等待执行，自己则直接返回 `Pending` 放弃本次执行？

而这，就是 `Context` 的作用了。我们深入 `Context` 的实现，会发现它大概长这样（删去了一些无关紧要的成员）：

```
pub struct Context<'a> {
    waker: &'a Waker,
}
pub struct Waker {
    raw: RawWaker,
}
pub struct RawWaker {
    data: *const (),
    vtable: *const RawWakerVTable,
```

```

}
pub struct RawWakerVTable {
    clone: unsafe fn(*const ()) → RawWaker,
    wake: unsafe fn(*const ()),
    wake_by_ref: unsafe fn(*const ()),
    drop: unsafe fn(*const ()),
}

```

那么，只要我们将这个队列的指针和最外层的 `Future` 的指针保存到 `RawWaker::data` 中，并将 `RawWakerVTable` 的 `wake` 方法设置为将保存的最外层 `Future` 放回队列中，我们就能实现上面的想法了。当然，上面的都是非常简化的讨论，实际要在 Rust 中实现上述操作需要考虑很多细节，比如各类数据结构的生命周期和内存位置等问题。好在大部分时候我们不需要手动去实现自己的 `Context`，而是可以使用 `async_task` 库来帮助我们完成这些工作，我们只需要指定当 `wake` 方法被调用时，我们想要干什么就可以了。

在目前版本的 MankorOS 中，我们基本上使用了上述实现。这相当于一个简单的 Round-Robin 调度器，具体细节可以参见 进程调度 一节。

```

// src/executor/task_queue.rs:6
pub struct TaskQueue {
    queue: SpinNoIrqLock<VecDeque<Runnable>>,
}
impl TaskQueue {
    pub const fn new() → Self {
        Self { queue: SpinNoIrqLock::new(VecDeque::new()) }
    }
    pub fn push(&self, task: Runnable) {
        self.queue.lock(here!()).push_back(task);
    }
    pub fn fetch(&self) → Option<Runnable> {
        self.queue.lock(here!()).pop_front()
    }
}

// src/executor/mod.rs:15
lazy_static! {
    static ref TASK_QUEUE: TaskQueue = TaskQueue::new();
}
pub fn spawn<F>(future: F) → (Runnable, Task<F::Output>)
where
    F: Future + Send + 'static,
    F::Output: Send + 'static,
{
    // 在此处指定当 cx.wake_by_ref() 被调用时，我们需要它干什么
    async_task::spawn(future, |runnable| TASK_QUEUE.push(runnable))
}

```

而在 `Future` 栈的另一头，当我们需要等待某个资源时，我们就可以直接将对应的 `waker` 传给资源方，等待回调，同时返回 `Pending` 了。具体细节可参见 包装型 `Future`。

3.2 内核实现要点

倘若是编写普通的异步程序，只需使用 `async/await` 关键字即可。但一个异步内核显然不能仅仅依赖组合已有的 `Future`，还必须实现一些底层或顶层的 `Future`，这些 `Future` 大致可以分为三类：

- 为其他 `Future` “装饰” 的 `Future`
- 为底层回调提供包装的 `Future`
- 一些辅助性的工具 `Future`

3.2.1 装饰型 `Future`

当我们使用 `async/await` 时，编译器会自动为我们生成一个 `Future` 的实现，这个实现会在子 `Future` 返回 `Pending` 时直接返回 `Pending`。

如果我们需要在子 `Future` 返回 `Pending` 时执行一些额外的操作，我们就必须手动编写该 `Future` 的实现。异步内核中用于切换到用户态线程的 `Future` 就是典型的此类 `Future`，无论子 `Future` 返回 `Pending` 还是 `Ready`，它都需要在执行前后完成一些额外的操作：

```
pub struct OutermostFuture<F: Future> {
    lproc: Arc<LightProcess>,
    future: F,
}
impl<F: Future> Future for OutermostFuture<F> {
    type Output = F::Output;
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) → Poll<Self::Output>
    {
        // ... (关闭中断，切换页表，维护当前 hart 状态)
        let ret = unsafe { Pin::new_unchecked(&mut this.future).poll(cx) };
        // ... (开启中断，恢复页表，恢复之前的 hart 状态)
        ret
    }
}
```

随后我们便可以在 `userloop` 外边包裹这个 `Future`，从而其无需再担心进程切换相关的杂事：

```
pub fn spawn_proc(lproc: Arc<LightProcess>) {
    // userloop 为切换到用户态执行的 Future
```

```

let future = OutermostFuture::new(
    lproc.clone(), userloop::userloop(lproc));
let (r, t) = executor::spawn(future);
r.schedule();
t.detach();
}

```

3.2.2 包装型 Future

这种 `Future` 通常位于 `Future` 栈的最底层（最后被调用的那个），用于将底层的回调接口包装成 `Future`。其一般表现为将 `Waker` 传出或将 `cx.waker().wake_by_ref()` 设置为回调函数。异步内核中用于实现异步管道读写操作的 `Future` 就是典型的此类 `Future`：

```

pub struct PipeReadFuture {
    pipe: Arc<Pipe>,
    buf: Arc<[u8]>,
    offset: usize,
}
impl Future for PipeReadFuture {
    type Output = usize;
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) → Poll<Self::Output>
    {
        // ... 各种检查和杂项代码
        if pipe.is_empty() {
            // 如果管道为空，就将当前 waker 存起来。在管道写入数据之后，
            // 会调用 pipe.read_waker.wake_by_ref() 以重新将顶层 Future 唤醒
            pipe.read_waker = Some(cx.waker().clone());
            Poll::Pending
        } else if pipe.is_done() {
            pipe.read_waker = None;
            Poll::Ready(0)
        } else {
            let len = pipe.read(this.buf.as_mut(), this.offset);
            // 如果写入时写满了管道的缓冲区，那么就将写入者的 waker 存起来。
            // 现在再调用。如果写入者已经写完了，则它不会再设置 pipe 的该成员。
            if let Some(write_waker) = pipe.write_waker {
                // 如果管道写入数据之前，已经有一个 waker 等待管道读取数据，
                // 那么就将这个 waker 唤醒
                write_waker.wake_by_ref();
            }
            Poll::Ready(len)
        }
    }
}

```

3.2.3 辅助型 Future

除了上面两大类 `Future` 之外，还有一些工具性质的 `Future`，在开发异步内核时也是非常有用的，现列举一二。

3.2.3.1 YieldFuture

有时候，我们需要当前 `Future` 主动返回一次 `Pending` 以让出控制权，但是并不想让它等待什么，而是直接回到调度器中等待下一次调度。这时候就可以使用 `YieldFuture`：

```
pub struct YieldFuture(bool);
impl Future for YieldFuture {
    type Output = ();
    fn poll(mut self: Pin<&mut Self>, cx: &mut Context<'_>) → Poll<Self::Output>
    {
        if self.0 {
            // 之后再次被 poll 时，它直接返回 Ready，什么事都不干
            return Poll::Ready(());
        } else {
            // 第一次调用时，self.0 为 false，此时它直接调用 wake_by_ref
            // 将自己重新加回调度器中，并返回 Pending 使得所有上层 Future
            // 返回，让出这一轮的调度权
            self.0 = true;
            cx.waker().wake_by_ref();
            Poll::Pending
        }
    }
}

pub fn yield_now() → YieldFuture {
    YieldFuture(false)
}
```

它可以用于实现 `yield` 系统调用，也可以在异步内核实现过程中用来实现某种“spin”式操作：

```
loop {
    let resource_opt = try_get_resouce();
    if let Some(resource) = resource_opt {
        break resource;
    } else {
        // 如果资源不可用，就让出控制权，并期望下次被调用时等待资源可用
        yield_now().await;
    }
}
```

但是，这种写法是不推荐的，它放弃了异步内核的很大一部分优越性。在使用这种写法之前，应该首先尝试将该资源的获取改写为回调式的，使用“包装型 `Future`”的写法实现。`YieldFuture` 也可用于某些系统的最底层实现中，比如搭配定时器中断，使用自旋检查的方法实现内核内的定时任务。

3.2.3.2 WakerFuture

WakerFuture 用于在 **async fn** 中获取当前 **Waker**:

```
struct WakerFuture;
impl Future for WakerFuture {
    type Output = Waker;
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) → Poll<Self::Output>
    {
        Poll::Ready(cx.waker().clone())
    }
}
```

如下使用便可以在 **async fn** 中获取当前 **Waker**:

```
async fn foo() {
    let waker = WakerFuture.await;
    resource.setReadyCallback(|| waker.wake_by_ref());
}
```

使用该 **Future** 时, 可以使很大一部分包装型 **Future** 得以直接使用 **async fn** 来实现, 而不用再手动实现 **Future** trait.

3.3 上下文切换

本节将描述异步内核中用户态 - 内核态上下文切换的过程。

内核态中最接近用户态的是 **userloop** 函数:

```
// src/process/userloop.rs:48
pub async fn userloop(lproc: Arc<LightProcess>) {
    // ...
    // 上下文保存在此
    let context = lproc.context();
    match lproc.status() {
        // ...
        ProcessStatus::READY => {
            // ...
            // run_user 函数便是切换到用户态的函数
            run_user(context);
        }
        // ...

        // 根据 scause 的值来判断是什么原因导致的陷入, 从而进行不同的处理
        let scause = scause::read().cause();
        // ...
        match scause {
            scause::Trap::Exception(e) => match e {
                Exception::UserEnvCall => {
                    // 系统调用
                    is_exit = Syscall::new(context, lproc.clone())
                }
            }
        }
    }
}
```

```

        .syscall().await;
    }
    Exception::InstructionPageFault
    | Exception::LoadPageFault
    | Exception::StorePageFault =>
        { /* 缺页异常, 略去 */ }
    Exception::InstructionFault
    | Exception::IllegalInstruction =>
        { /* 略去 */ }
    _ => todo!(),
},
scause::Trap::Interrupt(i) => match i {
    Interrupt::SupervisorTimer => {
        // 定时器中断, 让出本轮执行权
        if !is_exit {
            yield_now().await;
        }
    }
}
// ...

```

其中 `run_user` 函数只是汇编写成的上下文保存函数的简单包装。于是“进入用户态”这个操作在内核看来不过是一个执行时间稍微久了那么一点的 `Future` 而已。而当用户尝试进行一个需要等待特定资源就绪的系统调用时，它会直接因为 Syscall `Future` 的 `Pending` 而被挂起，直到资源就绪才会重新将顶层 `Future` 返回给调度器。由于页表切换等环境准备都是在 `userloop` 之上的 `OutermostFuture` 中处理的，而只要 `userloop` 中的 `.await` 导致其生成的 `Future` 返回 `Pending`，那么在一层层退出 `Future` 的过程中，环境自然会被切换回去，无需进一步操心。

4 文件系统

4.1 VFS 设计与结构

VFS (Virtual File System) 是指一种抽象层，用于在操作系统中将不同类型的文件系统统一起来。在 VFS 的设计中，所有的 I/O 请求都被发送到 VFS 层，并由 VFS 层进行相应的处理后再传递给具体的文件系统。VFS 的结构通常由以下几个部分组成：

- 虚拟文件系统：代表了整个文件系统树，包含了各种文件系统节点（文件、目录、符号链接等）。
- 文件系统接口：由各个具体文件系统提供的接口，用于实现文件系统的基本操作，例如读写文件、创建删除文件等。

- 虚拟文件系统操作接口：由 VFS 层提供的接口，用于对外提供文件系统操作的统一接口，例如打开文件、关闭文件、读取文件等。

为了支持异步内核 Cooperative Scheduling 的设计，MankorOS 的 VFS 设计中同时包含有异步的接口和同步的接口。同步的接口主要供内核使用，一般是较快的操作，而且保证不会受到其他进程的阻塞。异步的接口主要供系统调用使用，进程对文件的读写可能依赖于其他进程，例如管道的读写，因此对于这些请求，实现异步的读写是有必要的。

4.2 设备文件系统 (devfs)

设备文件系统 (devfs) 是一种特殊的文件系统，用于管理系统中的设备文件。在 Unix-like 操作系统中，所有的硬件设备都被表示为一个文件或文件夹，并挂载在设备文件系统中。通过这种方式，用户可以像访问普通文件一样访问和操作硬件设备。

在 MankorOS 中，设备管理模块通过注册和卸载设备的方式来实现对设备文件系统的管理。当一个设备被注册后，其对应的设备文件会被创建并挂载到 `/dev` 目录下，用户可以通过这个设备文件进行设备的读写等操作。

4.2.1 块设备的挂载

块设备是指按照固定大小划分为若干个扇区的存储介质，例如硬盘、U 盘等。在 MankorOS 中，支持将块设备挂载到文件系统上，并通过 VFS 层提供的标准接口进行操作。为了实现块设备的挂载，MankorOS 在 `Disk` 结构体中实现了 `VFS Trait` (包括 `open`、`read`、`write`、`seek` 等函数)，并在设备管理模块中注册了 `VirtIO` 发现的块设备。

4.3 FAT32 文件系统

FAT32 是一种常见的文件系统格式，广泛应用于 Windows 系统及其他各种设备中，例如移动硬盘、SD 卡等。在 MankorOS 中，文件系统模块实现了对 FAT32 文件系统的支持，用户可以对 FAT32 格式的设备进行挂载和操作。MankorOS 还支持 MBR 格式的分区表，如果块设备上存在有 MBR 分区表，MankorOS 可以解析并挂载上面的多个文件系统。

区域赛中，MankorOS 自动从 `virtio blk` 设备上识别 FAT32 文件系统，并能够自动执行上面的测试程序。

4.4 Pipe 与 Stdio

管道 (pipe) 是一种特殊的文件，主要用于实现进程间通信。在 Unix-like 系统中，管道被视为一种特殊的文件类型，可以像普通文件一样进行读写操作。在 MankorOS 中，支持通过 VFS 层提供的接口创建、打开、关闭、读写管道文件，从而实现进程间通信的功能。

MankorOS 实现了一个管道数据结构，其中包含两个实例，一个是读端，一个是写端。管道的数据保存在一个环形缓冲区中，而这个缓冲区是使用一个 RingBuffer 库来实现的。该环形缓冲区分配于内核堆上，通过 `Arc` 和 `SpinNoIrqLock` 进行并发访问控制。

当写入数据时，管道首先检查是否可写，然后检查是否挂起。如果管道没有挂起，则获取锁以访问管道的数据，并将数据写入环形缓冲区中。如果缓冲区已满，释放锁，并调用 `yield_now()` 函数，将 CPU 切换到其他任务。当有足够的空间时，释放锁并返回写入的字节数。同样地，当读取数据时，管道首先检查是否可读，然后检查是否挂起。如果管道没有挂起，则获取锁以访问管道的数据，并从环形缓冲区中读取数据。如果缓冲区为空，释放锁，并调用 `yield_now()` 函数，将 CPU 切换到其他任务。当有足够的空间时，释放锁并返回读取的字节数。对于管道的其他操作，如 `fsync` 和 `truncate`，MankorOS 会返回不支持的错误。

目前 MankorOS 的管道实现并不高效，高效的实现需要使用到暂未实现的异步睡眠锁，未来 MankorOS 将会对这个部分进行优化。

Stdio (standard input/output) 是指标准输入输出，在 C 语言中主要通过三个标准流 `stdin`、`stdout` 和 `stderr` 来实现。在 MankorOS 中，用户可以通过标准输入输出流来读取或输出数据，并可以将标准输入输出流与文件系统中的文件或管道进行关联，实现灵活的输入输出方式。

5 进程设计

MankorOS 支持进程和线程，进程和线程都是用轻量级线程 (`LightProcess`) 结构统一表示。在本章中，将先后介绍：

- `LightProcess` 结构体
- 用户地址空间管理
- 进程调度

5.1 进程和线程

与 Linux 内核相似，在 MankorOS 内核中，进程和线程两者并没有区别，可以统一地称之为轻量级进程，以 `LightProcess` 结构体表示和管理。线程可以理解为在 `sys_clone` 系统调用时指定了共享资源的进程 (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。

5.1.1 `LightProcess` 结构体

目前 `LightProcess` 结构体主要包含以下数据结构 (其中 **粗体** 的是可以在任务之间共享的)

- 进程基本信息
 - 进程号 `id`
 - 进程状态 `state`
 - 退出码 `exit_code`
- 进程关系信息
 - 父进程 `parent`
 - 子进程数组 `children`
 - 进程组信息 `group` (用于实现线程组)
- 进程资源信息
 - 地址空间 `memory`
 - 文件系统信息 `fsinfo`
 - 文件描述符表 `fdtable`
- 其他
 - 信号处理函数 `signal`

`LightProcess` 代码如下所示:

```
type Shared<T> = Arc<SpinNoIrqLock<T>>;

pub struct LightProcess {
    id: PidHandler,
    parent: Shared<Option<Weak<LightProcess>>>,
    context: SyncUnsafeCell<Box<UKContext, Global>>,

    children: Arc<SpinNoIrqLock<Vec<Arc<LightProcess>>>>,
    status: SpinNoIrqLock<SyncUnsafeCell<ProcessStatus>>,
    exit_code: AtomicI32,
```

```

group: Shared<ThreadGroup>,
memory: Shared<UserSpace>,
fsinfo: Shared<FsInfo>,
fdtable: Shared<FdTable>,
signal: SpinNoIrqLock<signal::SignalSet>,
}

```

在内核代码中，其他部分一般持有 `Arc<LightProcess>` (`LightProcess` 的引用计数智能指针)。这样既可以保证对应进程的信息不会过早被释放，也可以保证当无人持有此进程信息时，此结构体占用的资源可以被回收。`LightProcess` 中可以共享的数据结构都用 `Arc` 包装，在 `sys_clone` 系统调用的实现中，如果需要共享特定资源，则可以直接利用 `Arc::clone` 方法使得两个进程的数据结构指向同一个实例；如果无需共享，则使用具体资源的 `clone` 的方法进行复制：

```

// src/process/lproc.rs:265
if flags.contains(CloneFlags::THREAD) {
    parent = self.parent.clone();
    children = self.children.clone();
    // remember to add the new lproc to group please!
    group = self.group.clone();
} else {
    parent = new_shared(Some(Arc::downgrade(&self)));
    children = new_shared(Vec::new());
    group = new_shared(ThreadGroup::new_empty());
}

```

5.1.2 进程的状态

在 MankorOS 中，进程有 3 种状态：

- **UNINIT**：该进程还未针对第一次运行做好准备（没有为 `main` 函数准备好栈上的内容）
- **READY**：该进程可以被执行
- **ZOMBIE**：已经退出的进程

在异步内核中，不需要维护进程是否被阻塞之类或是否已经被加入准备执行的队列的状态：

- 若进程执行某个耗时久的系统调用（比如 `sleep`），代表它的 `Future` 会直接返回 Pending，从而使它离开调度队列；直到阻塞的条件被满足后，它会被 waker 自动重新加入回调调度队列。因此不需要代表“进程因为缺少某条件不能被调度”的状态。
- 异步编程模型中的 `Task` 抽象会保证一个 `Future` 不会被 wake 多次，从而使得已经在调度队列中的进程不会被重复加入。因此不需要代表“进程已经被调度”的状态

5.2 地址空间

出于性能考虑，MankorOS 的内核与用户程序共用页表，且内核空间占用的二级页表在不同用户程序之间是共享的。

5.2.1 地址空间布局

用户地址空间布局如下图所示：

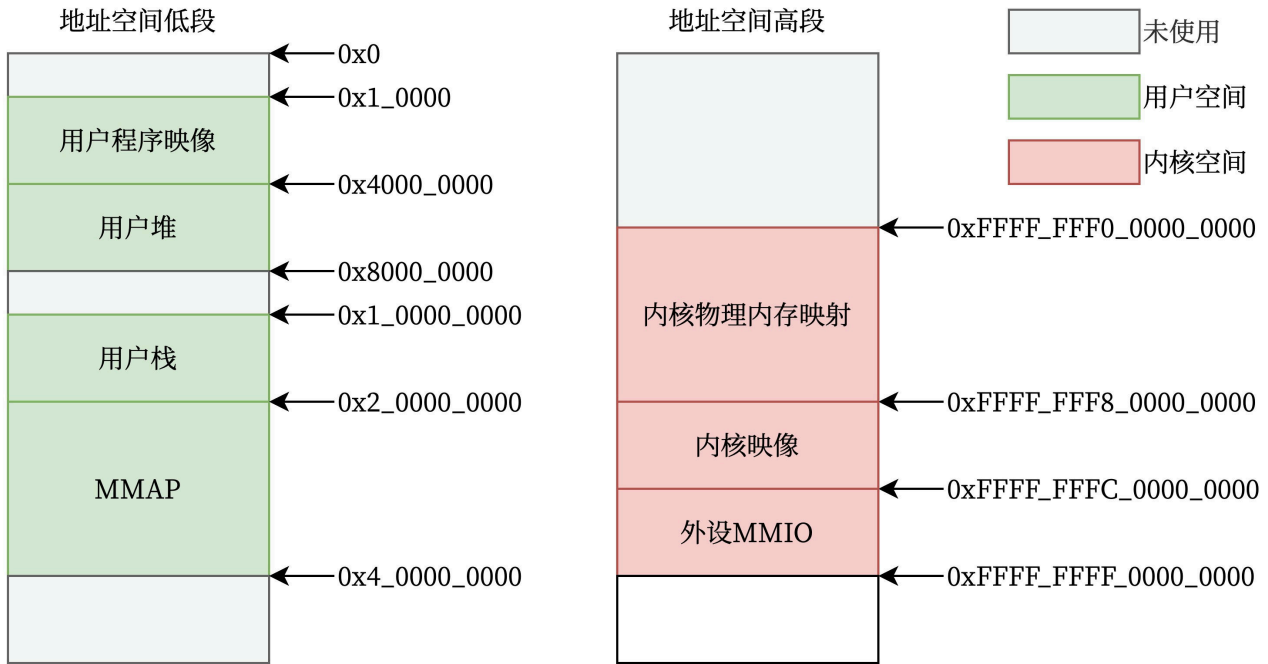


图 5-1 地址空间

5.2.2 地址空间管理

MankorOS 中的地址空间的各类信息由 `UserSpace` 结构体表示：

```
pub struct UserSpace {
    // 根页表
    pub page_table: PageTable,
    // 分段管理
    areas: UserAreaManager,
}
```

其中 `UserAreaManager` 结构体用于管理用户程序的各个段，其组成非常简单：

```
pub struct UserAreaManager {
    map: RangeMap<VirtAddr, UserArea>,
}

pub struct RangeMap<U: Ord + Copy, V>(BTreeMap<U, Node<U, V>>);
```


`RangeMap` 的实现直接借用了 FTL-OS 的实现。但额外增加了原地修改区间长度的 `extend_back` 和 `reduce_back` 方法，以针对堆内存的动态增长和缩减进行优化。对于其他类型的区间长度增减，仍然采用“创建新区间 - 合并”和“分裂旧区间 - 删除”的方式。

`UserArea` 中保存了各个内存段的信息，包括：

```
bitflags! {
    pub struct UserAreaPerm: u8 {
        const READ = 1 << 0;
        const WRITE = 1 << 1;
        const EXECUTE = 1 << 2;
    }
}

enum UserAreaType {
    /// 匿名映射区域
    MmapAnonymous,
    /// 私有映射区域
    MmapPrivate {
        file: Arc<dyn VfsNode>,
        offset: usize,
    },
    // TODO: 共享映射区域
    // MmapShared {
    //     file: Arc<dyn VfsNode>,
    //     offset: usize,
    // },
}

pub struct UserArea {
    kind: UserAreaType,
    perm: UserAreaPerm,
}
```

考虑到地址空间段的类型是基本确定的，此处并没有像 Linux 一样使用函数指针（“虚表”）来抽象各类段的行为，也没有使用本质上相同的 Rust 的 `dyn trait` 方式，而是直接使用枚举类型实现。这样既可以保证处理时的完整地好各类情况，也有一定的性能优势。

MankorOS 中所有段都是懒映射且懒加载的，所有内存数据都会且只会在处理缺页异常时被请求（譬如换入页或读取文件信息）。这同时也带来了 `exec` 系统调用中对 ELF 文件的懒加载能力。该实现还意味着各种不同类型的段只需要在构造或处理缺页异常时进行不同的处理即可。所有 `UserArea` 方法中只有缺页异常处理需要针对不同的段类型进行不同的处理，使得使用枚举区分不同段的方法带来了几乎为零的代码清晰程度开销。正因如此，我们放弃了虚表方法可能带来的代码清晰度的提升与灵活性，而选择了性能更好的枚举实现。

5.2.3 缺页异常处理与 CoW

当 `userloop` 中检测到用户程序因为缺页异常而返回内核时，会从 `stval` 寄存器中读出发生缺页异常的地址，在经过一些包装函数后，会来到 `UserArea::page_fault` 函数中：

```
// src/process/user_space/user_space.rs:193
pub fn page_fault(
    &self,
    page_table: &mut PageTable,
    range_begin: VirtAddr, // Allow unaligned mmap ?
    access_vpn: VirtPageNum,
    access_type: PageFaultAccessType,
) → Result<(), PageFaultErr> {
    if !access_type.can_access(self.perm()) {
        // 权限检查，如果访问权限不符合要求，则直接返回错误
        // 此处"权限" 检查匹配的是 UserArea 中保存的，该区域应有的权限
        // 而非页表中的权限（页表中的权限会因为 CoW/懒加载 等改变）
        return Err(PageFaultErr::PermUnmatch);
    }

    // 分配新物理页
    let frame = alloc_frame()
        .ok_or(PageFaultErr::KernelOOM)?;
    // 遍历页表找到发生缺页异常的页的页表项
    let pte = page_table.get_pte_copied_from_vpn(access_vpn);
    if let Some(pte) = pte && pte.is_valid() {
        // 如果权限正确，且页表有效，那么是因为 CoW 引起的缺页异常
        // 因为 CoW 会将原来的页表项可写的页的权限设置为只读，而此处发生了真实的写入
        let pte_flags = pte.flags();
        // 减少旧物理页的引用计数
        let old_frame = pte.paddr();
        with_shared_frame_mgr(|mgr| mgr.remove_ref(old_frame.into()));
        // 复制旧物理页的内容到新物理页
        unsafe {
            frame.as_mut_page_slice()
                .copy_from_slice(old_frame.as_page_slice());
        }
    } else {
        // 如果页表项无效，说明是懒加载
        match &self.kind {
            // 对匿名映射的段，懒加载并不需要向其写入任何数据
            UserAreaType::MmapAnonymous => {},
            // 对私有文件映射的段，懒加载需要去文件中读取对应范围内的数据
            UserAreaType::MmapPrivate { file, offset } => {
                let access_vaddr: VirtAddr = access_vpn.into();
                let real_offset = offset + (access_vaddr - range_begin);
                let slice = unsafe { frame.as_mut_page_slice() };
                let _read_length =
                    file.sync_read_at(real_offset as u64, slice)
            }
        }
    }
}
```

```

        .expect("read file failed");
    }
}

// 修改页表项
page_table.map_page(access_vpn.into(), frame, self.perm().into());
Ok(())
}

```

当地址空间发生 CoW 复制时，我们一项项遍历原来的页表项，将其修改为只读且将其映射的物理页的使用计数加一：

```

// src/process/user_space/mod.rs:260
pub fn clone_cow(&mut self) → Self {
    Self {
        page_table: self.page_table.copy_table_and_mark_self_cow(|frame_paddr|
        {
            with_shared_frame_mgr(|mgr| mgr.add_ref(frame_paddr.into()));
        }),
        areas: self.areas.clone(),
    }
}

```

5.3 进程调度

MankorOS 是异步内核，其进程调度与同步内核有所不同。显著的一个特点是在内核中并不需要维护一个保存了所有进程信息的数组，而是使 `Arc<LightProcess>` 分散在内核内存中的各个 `Future` 中，直到 waker 将其“唤醒”，调度器才能知晓该进程的存在。

目前 MankorOS 中的进程调度器是一个简单的 FIFO 调度器，其内部维护了一个双端队列，依次从队列头部取出包含待调度进程的 `Future` 并执行。当用户程序因为时间片用完而返回内核时，放弃该轮执行并被重新加入调度队列的尾部。当用户程序因为系统调用而返回内核，并且该系统调用会“阻塞”时，它会返回一个 `Pending` 状态，直到等待到阻塞条件满足后被 waker 唤醒。换言之，异步内核中没有“阻塞”的概念，一切操作要么马上结束，要么放弃执行等待回调。

MankorOS 下一步预计引入更加复杂的调度算法，比如为每个 CPU 核心维护一个优先级队列以更好地利用缓存。这可以通过在生成代表进程的 `Future` 时，向其传入不同的函数来实现。具体而言便是修改此处的实现，将 `|runnable| TASK_QUEUE.push(runnable)` 改为更复杂的“使不同的调度器知晓自身存在”的操作即可。

```
// src/executor/mod.rs:25
pub fn spawn<F>(future: F) → (Runnable, Task<F::Output>)
where
    F: Future + Send + 'static,
    F::Output: Send + 'static,
{
    async_task::spawn(future, |runnable| TASK_QUEUE.push(runnable))
}
```

6 同步

6.1 内存模型和缓存一致性

首先回顾一下内存模型和缓存一致性的发展历史。

最早期的多核处理器使用 Sequential Consistency 内存模型。Sequential Consistency 是一种严格的内存模型，它要求所有处理器核心之间的内存访问操作按照程序中编写的顺序执行。这意味着每个处理器核心看到的共享内存状态都是相同的，从而保证了数据的正确性。然而，这种模型的缺点是性能较低，因为它限制了处理器核心之间的并行度，所以在现代处理器中已经看不到使用了。

现代 x86 的处理器使用的是另一种内存模型 Total Store Ordering，它允许处理器核心之间的内存访问操作可以乱序执行，但要求写入操作必须全局可见，即写入操作必须以与程序中编写的顺序相同的顺序出现在其他处理器核心的读取中。这可以通过在写入操作前插入一个屏障指令来实现。这种模型允许更高的并行度，但需要编译器和程序员使用屏障指令来保证正确性。

与上述两种模型不同，RISC-V 中常使用的 **RVWMO** (RISC-V Weak Memory Model) 内存模型是一种弱同步模型，它允许处理器核心之间的内存访问操作可以乱序执行，并且没有全局的内存访问顺序要求。但是，RVWMO 内存模型规定了一些具体的内存序要求，以确保正确性：

1. 内存屏障（memory barrier）指令必须按照程序中编写的顺序执行，并且必须在读取操作和写入操作之间使用。
2. 写入相同地址的操作必须保持程序顺序。
3. 读取相同地址的操作必须保持程序顺序。
4. 原子操作可以保证多个处理器核心之间的互斥和同步，并且必须按照内存屏障指令的顺序执行。

Rust 中提供了多种不同的内存序用于控制多线程并发访问时的行为，以下是其中几种被抽象出来的内存序：

1. **SeqCst**：全称为 Sequentially Consistent，表示所有的操作都会按照程序中给定的顺序执行，即保持原子性、有序性和一致性。
2. **Acquire**：表示该操作之前的所有读取操作必须先于该操作执行，并且该操作与后续写入操作无序。
3. **Release**：表示该操作与之后的所有写入操作无序，并且该操作之后的所有读取操作必须在该操作执行之后执行。
4. **Relaxed**：表示不需要任何同步或顺序约束。

这些内存序可以通过 Rust 原子类型（如 **AtomicBool**、**AtomicI32** 等）中的方法进行设置和使用。通过适当地选择内存序，我们可以在不同的情况下实现合适的内存同步，以确保多线程代码的正确性和性能。

6.2 锁

6.2.1 自旋锁

在 Rust 的 **no_std** 环境下，一些操作系统提供的同步原语（如互斥锁和条件变量）不可用。此时我们可以使用自旋锁来实现同步。

自旋锁是一种简单的同步机制，它通过忙等待的方式来阻塞线程，直到共享资源可用为止。当一个线程获取到自旋锁时，其他试图获取该锁的线程会进入自旋状态，反复尝试获取锁，直到当前持有锁的线程释放锁为止。

在 Rust 中，自旋锁可以通过原子类型 **AtomicBool** 和 **spin_loop_hint()** 函数来实现。以下是一个简单的自旋锁实现：

```
use core::sync::atomic::{AtomicBool, Ordering};

pub struct SpinLock {
    locked: AtomicBool,
}

impl SpinLock {
    pub const fn new() → Self {
        SpinLock { locked: AtomicBool::new(false) }
    }

    pub fn lock(&self) {
        while self.locked.swap(true, Ordering::Acquire) {}
    }
}
```

```

        // 自旋等待锁
        core::hint::spin_loop();
    }
}

pub fn unlock(&self) {
    self.locked.store(false, Ordering::Release);
}
}

```

上述代码中, 使用 `AtomicBool` 类型的 `locked` 字段表示锁的状态。`lock()` 方法使用 `swap()` 方法来尝试获取锁并将 `locked` 设为 `true`, 同时使用 `Acquire` 内存序来保证前面的读操作和当前的写操作不被重排序。如果 `swap()` 返回的是 `true`, 则表示锁已经被其他线程持有, 此时进入自旋状态直到获取到锁为止。在自旋状态中使用 `spin_loop()` 函数来提示 CPU 循环等待, 以减少 CPU 的消耗。`unlock()` 方法通过调用 `store()` 方法将 `locked` 设为 `false`, 同时使用 `Release` 内存序来保证当前的写操作和后续的读操作不被重排序。

MankorOS 利用 Rust 的 RAII (资源获取即初始化) 来确保在作用域结束时自旋锁会被正确地释放。具体来说, MankorOS 定义了一个包含自旋锁的新类型, 并实现 `Drop` trait 来在该类型的实例离开作用域时释放锁。

代码实现举例如下:

```

use core::sync::atomic::{AtomicBool, Ordering};

pub struct SpinLock {
    locked: AtomicBool,
}

impl SpinLock {
    pub const fn new() → Self {
        SpinLock { locked: AtomicBool::new(false) }
    }

    pub fn lock(&self) → SpinLockGuard {
        while self.locked.swap(true, Ordering::Acquire) {
            // 自旋等待锁
            core::hint::spin_loop();
        }
        SpinLockGuard { spin_lock: self }
    }

    pub fn unlock(&self) {
        self.locked.store(false, Ordering::Release);
    }
}

```

```

}

pub struct SpinLockGuard<'a> {
    spin_lock: &'a SpinLock,
}

impl<'a> Drop for SpinLockGuard<'a> {
    fn drop(&mut self) {
        self.spin_lock.unlock();
    }
}

```

上述代码中，定义了一个 `SpinLockGuard` 结构体来保存 `SpinLock` 的引用，并在其 `Drop` 实现中调用自旋锁的 `unlock()` 方法来释放锁。在 `lock()` 方法中，通过返回一个 `SpinLockGuard` 结构体来获取自旋锁。由于 `SpinLockGuard` 结构体实现了 `Drop` trait，因此当该结构体离开作用域时，会自动调用 `unlock()` 方法来释放锁。

使用 RAII 来管理自旋锁的获取和释放，可以有效避免忘记释放锁导致死锁等问题，在 Rust 中也是一种常见的编程模式。

6.2.2 睡眠锁

异步 Rust 中，睡眠锁的实现涉及 Future 状态的转换，在区域赛阶段，MankorOS 并未实现睡眠锁，未来将会实现。

6.3 进程间通信

MankorOS 在区预赛中仅实现了简单的进程间通信，使用一个 `AtomicBool` 保存信号的 flag。为了支持 wait 系统调用，进程退出时会正常向父进程发送 `SIGCHLD` 信号。

7 系统调用

7.1 内存相关系统调用

MankorOS 实现了三个内存相关的系统调用，分别为 `brk`、`mmap` 和 `munmap`。

`sys_brk`：该系统调用用于更改进程的堆顶地址，并返回当前进程的堆顶地址。当参数 `brk` 为 `0` 时表示查询当前堆顶地址。MankorOS 实现中，通过使用 `LightProcess` 的内存管理器，记录和更新堆顶地址。

`sys_mmap`：该系统调用允许进程在其虚拟地址空间中映射内存区域。支持选项包括指定起始地址、长度、权限（`PROT_READ`、`PROT_WRITE` 和

`PROT_EXEC`) 和标志 (`MAP_SHARED`、`MAP_PRIVATE`、`MAP_FIXED`、`MAP_ANONYMOUS` 和 `MAP_NORESERVE`)。MankorOS 通过 `LightProcess` 的内存管理器来分配和映射物理页框，并将这些页框映射到进程的虚拟地址空间中。

`sys_munmap`：该系统调用用于解除映射的内存区域。MankorOS 会通过内存管理器来取消对映射内存的映射，并释放相应的物理页。

7.2 文件系统相关系统调用

MankorOS 实现了十四个文件系统相关的系统调用。其中与文件读写相关的有七个 `openat`、`close`、`read`、`write`、`dup`、`dup3`、`pipe`；与文件系统相关的有七个 `getcwd`、`fstat`、`mkdir`、`getdents`、`unlinkat`、`mount`、`umount`。

`sys_openat`：该系统调用用于打开文件。支持选项包括指定文件路径（包括基于 `dir_fd` 的相对路径和绝对路径）和文件创建标志 (`O_CREAT`)。

`sys_close`：该系统调用用于关闭文件。

`sys_read`：该系统调用用于读取文件。

`sys_write`：该系统调用用于写入文件。

`sys_dup`：该系统调用用于复制文件描述符。

`sys_dup3`：该系统调用用于复制文件描述符，并指定新的文件描述符的值。

`sys_pipe`：该系统调用用于创建管道。

`sys_getcwd`：该系统调用用于获取进程当前工作目录。

`sys_fstat`：该系统调用用于获取文件的状态。

`sys_mkdir`：该系统调用用于创建目录。支持基于 `dir_fd` 的相对路径和绝对路径。

`sys_getdents`：该系统调用用于获取目录下的文件信息。

`sys_unlinkat`：该系统调用用于删除文件。支持指定文件路径（包括基于 `dir_fd` 的相对路径和绝对路径）和文件删除标志 (`AT_REMOVEDIR`)。

7.3 进程相关系统调用

MankorOS 实现了五个进程相关的系统调用，分别为 `wait`、`clone`、`execve`、`getpid` 和 `getppid`。

`sys_wait`：该系统调用用于等待子进程结束。支持非空的 `wstatus` 参数，可告知调用者子进程的退出状态。

sys_clone: 该系统调用用于创建一个新的进程, 并能详细指定资源共享的情况。支持选项包括指定内存, 文件系统信息, 已打开文件, 信号处理句柄, 父进程与线程组的共享选项 (**CLONE_VM**, **CLONE_FS**, **CLONE_FILES**, **CLONE_SIGHAND**, **CLONE_PARENT** 和 **CLONE_THREAD**), 以及设置新进程的局部储存的选项, 在父进程中获取子进程 PID, 在子进程中返回 0 的选项和在子进程中获取自身 PID 的选项。(分别为 **CLONE_SETTLS**, **CLONE_PARENT_SETTID**, **CLONE_CHILD_CLEARTID** 和 **CLONE_CHILD_SETTID**)

sys_execve: 该系统调用用于加载新的程序。支持指定程序的路径, 参数和环境变量。

sys_getpid: 该系统调用用于获取当前进程的 PID。

sys_getppid: 该系统调用用于获取当前进程的父进程的 PID。

7.4 其他系统调用

除了上述的系统调用外, MankorOS 还实现了四个系统调用, 分别是三个时间相关的 **gettimeofday**、**times** 和 **nanosleep** 以及一个系统信息相关的 **uname**。

sys_gettimeofday: 该系统调用用于获取 **TimeVal** 格式的当前时间。当前时间由系统 **timer** 直接维护。

TimeVal 参考 Linux 实现如下:

```
pub struct TimeVal {
    // seconds
    pub tv_sec: usize,
    // microseconds
    pub tv_usec: usize,
}
```

sys_times: 该系统调用用于获取当前进程的运行时间。包括在内核态的时间和用户态的时间, 以及当前进程的全体子进程的内核态时间和用户态时间。实现上每个进程都单独维护一个计时器, 当进程在用户态和内核态之间切换时更新时间, 当进程被剥夺或者被调度时相应地停止或启动计时。该时间以 **Tms** 的格式返回, 参考 Linux 实现如下:

```
pub struct Tms {
    // user time
    pub tms_utime: usize,
    // system time
    pub tms_stime: usize,
    // user time of children
    pub tms_cutime: usize,
    // system time of children
    pub tms_cstime: usize,
}
```

sys_nanosleep: 该系统调用用于实现纳秒级别的高精度 sleep。用于描述纳秒级的时间间隔的结构体 **TimeSpec** 参考 Linux 实现如下:

```
pub struct TimeSpec {  
    // seconds  
    pub tv_sec: usize,  
    // nanoseconds  
    pub tv_nsec: usize,  
}
```

sys_uname: 该系统调用用于获取系统描述信息。默认的系统描述信息如下:

描述	值
sysname	“MankorOS”
nodename	“MankorOS-VF2”
release	“rolling”
version	“unknown”
machine	“unknown”
domainname	“localhost”

表 7-1 uname 的返回值

8 总结与展望

8.1 初赛测评

目前（2023.05.27）已满分通过初赛所有测试用例，排行榜如图 8-1 所示:

#	用户名	队伍	最后提交时间(ASC)	提交次数(ASC)	rank
1	202310464101015	你说对不队/ 河南科技大学	2023-04-18 14:30:26	22	102.0000
2	202318123101314	Titanix/ 哈尔滨工业大学（深圳）	2023-05-04 21:50:35	17	102.0000
3	202310698101003	PLNTRY/ 西安交通大学	2023-05-12 01:43:31	9	102.0000
4	202314430101195	编写吧!NutOS/ 中国科学院大学	2023-05-19 15:39:44	17	102.0000
5	202318123101332	MoOS/ 哈尔滨工业大学（深圳）	2023-05-19 17:09:27	28	102.0000
6	202310336101112	LostWakeup/ 杭州电子科技大学	2023-05-20 21:25:24	5	102.0000
7	202310336101111	BiteTheDisk/ 杭州电子科技大学	2023-05-22 22:55:01	5	102.0000
8	202310487101114	AVX/ 华中科技大学	2023-05-22 23:21:08	17	102.0000
9	202318123101282	MankorOS/ 哈尔滨工业大学（深圳）	2023-05-23 18:39:43	12	102.0000

图 8-1 初赛排行榜

8.2 实现情况

当前各模块完成情况如 表 8-1 所示：

模块	完成情况
无栈协程基建	基于全局队列实现的调度器，可供异步程序执行
内存管理	实现 <code>mmap</code> / <code>munmap</code> 系统调用，可对所有内存段进行懒分配或懒加载，具备写时复制 (CoW) 功能
文件系统	完成虚拟文件系统，支持 <code>devfs</code> 和管道
进程管理	支持 <code>clone</code> 系统调用，可以细粒度划分进程共享的资源
信号机制	完成基础的信号机制
用户程序	能通过所有初赛测试样例

表 8-1 模块完成情况

8.3 未来工作

- 使内核更加异步化，支持异步内存复制和文件 IO，实现异步文件系统
- 完善进程信号传递机制，实现基于事件总线的进程通讯机制
- 支持动态链接
- 支持 `proc` 文件系统
- 支持 `musl libc` 和 `busybox`，移植更多用户程序