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

基于 RISC-V 64 的
异步多核操作系统 FTL OS

FTL OS 项目组

项目成员：叶自立（组长）
樊博
李羿廷

指导老师：夏文，仇洁婷

2022 年 5 月

目录

1 系统设计

1.1 前言

FTL OS 是 RISC-V 64 平台上的高性能多核异步宏内核操作系统，完全从零开始开发，部分代码借鉴自清华大学的 rCore-tutorial。正如 FTL 超越光速之意，**FTL OS 将性能作为第一目标**，性能与多核安全是 FTL OS 开发的绝对重心。FTL OS 将探索现代操作系统发展的最前沿，在性能、功能、安全、稳定性上都达到现代操作系统的水平，并维持代码的可读性，能够在教学上发挥作用。

FTL OS 开发过程并不是完全循序渐进的迭代开发，而是完成阶段性架构设计后一次性进行完整的代码编写，将重构放在了设计阶段。也正因为如此 FTL OS 在预开发阶段抛弃了 rCore-tutorial，因为尽管 rCore-tutorial 作为一个教学操作系统是非常成功的，但 rCore-tutorial 没有为多核运行作任何的考量，这一点甚至不如 xv6。由于 FTL OS 的多核安全从立项开始就放置在了最高优先级，而为 rCore-tutorial 添加多核的补丁的工作量太大且可能忽略许多细节，这也是 FTL OS 选择从零开发的原因。

1.2 为什么选择 rust

FTL OS 选择了 rust 语言开发内核。这并不是随大流，下面将说明放弃 C/C++的部分原因：

1. C/C++无法使用原始 libc 库，如果要从零开发需要造非常多的轮子，而且性能低下。例如 memcpy 函数，xv6 中的实现是使用 for 循环逐字节复制，但这其实是很低效的，当地址对齐到 8 字节时可以用 ld/sd 指令一次复制 8 个字节，相比原始实现性能提高 8 倍！如果支持向量指令集还可以更快。你费尽心血应用了一大堆复杂的优化，另一个队简单重写了 memcpy 就比你快了！这样的优化还有很多，阅读 libc 的源码可以有更深的理解。

2. C 语言基于 goto 的错误处理机制常人难以驾驭，C++开创性的 RAII 机制改善了资源的回收，但基于栈回退的异常的错误处理性能极差且阻碍优化，而基

于错误码的错误处理非常繁琐。

3. C 语言是彻底的静态语言，类型系统孱弱无比，宏是唯一批量生成重复代码的手段，因此所有数据结构都强制使用危险的侵入式数据类型。C++提供了强大的模板来生成代码，但每个编译单元都需要独立编译，编译速度非常缓慢。

4. 声明与实现分离的分离编译。C 语言的这一设计是早期编译平台性能不足导致的，它可以提高编译速度，但也非常严重地阻碍了编译器内联，需要繁琐的方式来解决它。也正因如此，C/C++之后所有语言采用了模块的方式编译，C++20 也加入了 module。

5. C/C++没有提供 slice 抽象，特别是字符串，全世界的编程语言中字符串通过结束符终止的仅此两家，除了效率低下外也带来了无数安全性的问题。C++20 加入了 span 改善了这一问题，但比赛的编译器不支持 C++20。

还可以列出许多，但这些问题替换为 rust 就可以解决了吗？rust 正式版于 2015 年上线，避免了许多语言的弯路。rust 相比 C/C++可以：

1. 提供了 no_std 下依然可以使用的 core 库，不用手写 memcpy 了。
2. 基于模式匹配的和类型错误处理，处理过程流畅无比。基于所有权的析构函数，所有权转移后不会调用析构函数（C++的析构函数必定执行）。
3. 基于 trait 的泛型机制，强大的类型匹配系统。
4. 基于模块的编译，函数显式声明为 inline 后在所有编译单元可见。
5. 基于 slice 于所有权的内存管理，不使用 unsafe 能够保证绝对内存安全。
6. 完善的 cargo 包管理工具，通过版本号可以保证编译稳定性。

1.3 为什么使用无栈异步架构

首次接触无栈异步架构是在调研 2021 届操作系统大赛时发现的华中科技大学无相之风团队开发的飓风内核，飓风内核是操作系统功能赛道二等奖作品，使用运行在用户态的共享调度器完成调度，上下文切换速度极快。接下来是 rCore，FTL OS 也从此开始确定使用了无栈协程架构。无栈上下文切换具有更高的速度，所有涉及上下文切换的函数都必须显示使用 await 标识，编译器可以看到所有上下文切换的位置。不像有栈协程需要为每个任务分配一个栈空间且必须在浪费空

间和栈溢出二选一，无栈协程最显著的优点就是程序运行时需要的内存大小是编译时确定的，在分配任务时被一次性分配内存，任务运行时绝对不会出现空间不足或空间过剩。目前无栈协程已经被大规模应用在了软件开发中且有了替代有栈协程的趋势，C++20 也历史性地加入了无栈协程的编译器支持。

UltraOS 是 2021 届操作系统大赛内核赛道一等奖作品，它的许多设计是针对 K210 平台运行的，大量的优化针对 K210 6MB 的内存限制，但这样小的内存基本只出现在嵌入式芯片中，严重限制了操作系统的施展。Hifive Unmatched 平台提供了 16GB 的内存与 4 个核心，这为 FTL OS 创造了巨大施展空间。K210 只有 2 个核心，因此多核竞争问题并不严重，但 Hifive Unmatched 提供了 4 个核心，因此如 UltraOS 一样大面积地使用全局锁会限制多核的施展。UltraOS 也缺少唤醒机制，这让全部进程都在调度队列中“排队”，如果存在非常多的“睡眠”进程将显著降低调度速度。

现代操作系统是如何运行的？开源的 Linux 是一个绝佳的老师。但由于 Linux 经过了多年发展已经高度复杂，FTL OS 对 Linux 的借鉴仅限于运行逻辑与优化算法。Linux 的设计确实精妙，每次阅读源码都会被它高效的实现震撼。FTL OS 将尝试将 Linux 内的系统以更高效简洁的方式在 FTL OS 中实现。

1.4 FTL OS 的设计准则

- 坚持性能至上不动摇
- 完善的多核内存安全
- 超大规模并发与高响应速度
- 文件系统支持超大磁盘
- 将锁的作用域缩小至极限
- 不需要的内存绝不浪费

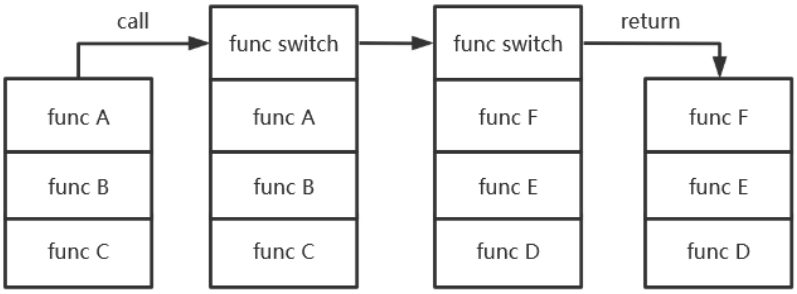
2 系统实现

2.1 进程调度

2.1.1 无栈协程

解释无栈协程之前首先要说明什么是协程。协程是在用户态模拟多线程的一种方式，区别是上下文没有操作系统的介入，上下文切换完全在用户态进行。协程调度器就像在用户态运行一个轻量操作系统，管理一系列协程的运行。但协程不同于线程，协程是用户程序的一部分，完全由开发者控制，因此不像操作系统一样需要抢占式调度，而可以使用更高效的协作式调度，在固定的调度点进行上下文切换。

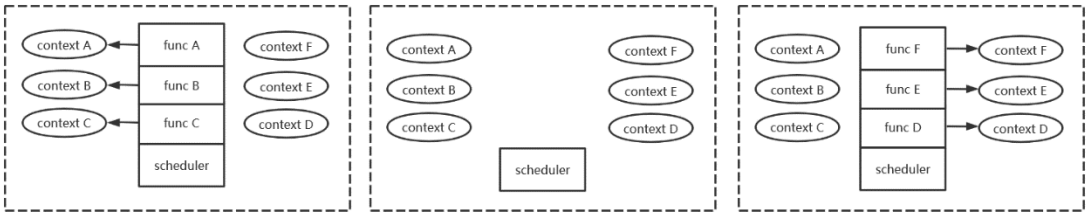
协程作为一个可以调度的对象允许函数在还没有完成之前离开当前的上下文，并在未来恢复并继续执行。离开一个函数只有两种方式：调用一个新函数与退出当前的函数，有栈协程和无栈协程分别对应了两种离开当前函数的方式。有栈协程的切换如下：



有栈协程的切换非常简单，因为调用一个函数不会销毁栈上数据，因此只需要保存现场并切换至待调度上下文即可。对于 RISC-V，上下文切换需要改变 s0-s11、ra、sp 共 14 个寄存器，其他的临时寄存器不需要保存。上下文切换函数与其他函数对语言来说没有本质区别，因此有栈协程的切换函数可以插入至程序的任意位置，因此一段普通程序改写成有栈协程是非常简单的。有栈协程的缺点很明显，每个上下文都需要分配独立的栈空间，而函数的调用深度是无法确定的，因此很容易出现栈空间溢出或浪费空间的情况。另一个问题是上下文切换修改了函数调用栈，因此 CPU 内部的函数调用缓存全部失效，每次函数返回都导致一

次分支预测失败，对流水线较长的高性能 CPU 性能开销非常大。

无栈协程通过函数返回的方式离开当前函数的上下文。无栈协程的切换如下：



func A 切换上下文时会依次退出函数直到回到调度器，但数据依然保存在堆上。调度器执行另一个函数时根据堆上信息重新生成了调用栈，恢复执行 func F。

rust 是如何编译 async 函数的呢？对于一个 async 函数：

```rust
async fn function() {
    async_fn().await;
    async_fn().await;
}
```

rust 编译器将 function 编译成如下的状态机：

```rust
enum Status {
    S0, S1, S2,
}
fn async_fn_poll() -> Poll<()> {
    todo!()
}
```

```rust
fn function_poll(mut s: Status) -> Poll<()> {
    loop {
        match s {
            Status::S0 => match async_fn_impl() {
                Poll::Ready(()) => s = Status::S1,
                Poll::Pending => return Poll::Pending,
            },
            Status::S1 => match async_fn_impl() {
                Poll::Ready(()) => s = Status::S2,
                Poll::Pending => return Poll::Pending,
            },
            Status::S2 => return Poll::Ready(()),
        }
    }
}
```

这里忽略了上下文相关的细节。可以看到，async_fn_poll 的上下文返回的过

程就是普通的函数返回，不需要进行栈的切换，不需要改变栈寄存器，CPU 的函数调用缓存可以正常工作。无栈协程的切换速度远高于有栈协程，速度甚至可以达到 4 倍以上。

无栈协程的另一个优点是作用域跨越`await`的变量将分配在函数自身所属的结构体上，而结构体的大小是在编译时就确定的。因此在编译的时候我们就可以获知整个异步任务需要的空间大小，一次性在堆上分配，完全不需要担心栈溢出的问题。与此同时，无栈任务运行在同一个栈上，因此栈的数量就等于线程数量或 CPU 的数量。如此少的栈意味着可以给他们开出巨大无比的空间，不再需要担心非溢出下的栈溢出的问题。

2.1.2 任务调度器

单线程下的进程调度很简单，只需要维护一个调度队列，OS 不断从调度队列中取出任务运行，进程切换时把当前进程再次加入调度队列即可。多核下的情况就变得复杂起来。锁相关的不再赘述，还需要考虑以下问题：

1. 一个线程被唤醒时需要加入调度队列，如果它被唤醒了两次会发生什么？
2. 如果一个线程在未退出时就被唤醒了会发生什么？(例如 yield)

最简单的解决方案是运行调度器中存在同一个线程的多个副本，并在线程中增加锁保证它不会被两个核心同时运行。这又会带来如下的问题：

如果获取的线程被另外的核心占有了，应该抛弃或死等，还是再次加入调度器？

(1)抛弃策略可能导致线程丢失。核心 A 占有线程 T 时执行了 yield 调用，加入调度队列。核心 B 在核心 A 离开前获取了 T，由于无法获取锁立刻释放，核心 A 离开线程 T 就永远丢失了。

(2)死等将导致极大的竞争，例如将一个线程在睡眠调度器中的同一时刻注册 4 次，当到达此时刻后四个核心都将阻塞在这个线程中，而只有一个核心在实际运行。

(3)再次加入调度器只能缓解死等的开销，当线程已经被占有后调度器中的其他副本事实上处于不可消失的状态，导致调度器的巨大无意义开销。

因此允许调度器中存在同一个线程的多个副本将导致极大的负面影响。要避免这种情况的发生需要考虑以下两种情况：

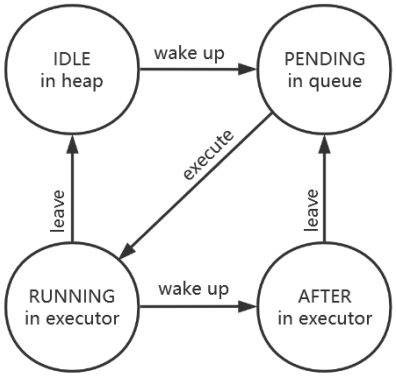
(1)线程未运行时被唤醒，直接加入调度器即可。

(2)线程运行时再次被唤醒，需要等待线程执行完成后再加入调度器。

使用这种方式就不再需要获取线程的锁了，只需要每个线程维护一个原子变量，原子变量具有如下 4 种状态：

状态	描述	唤醒后的状态	进入/离开调度器后的状态
IDLE	线程未运行且不处于调度器中	PENDING	Impossible
PENDING	线程未运行且处于调度器中	PENDING	RUNNING
RUNNING	线程正在运行，结束后不加入调度器	AFTER	IDLE
AFTER	线程正在运行，结束后加入调度器	AFTER	PENDING

这种方式保证了一个线程至多在调度器中出现一次，原子变量的状态改变使用 CAS 或 AMO 操作，不需要关中断。



FTL OS 采用了 async-task 库辅助调度。async-task 提供了工业级代码来完成上述的唤醒流程，提供的 spawn 接口如下：

```
pub fn spawn<F, S>(future: F, schedule: S) -> (Runnable, Task<F::Output>)
where
    F: Future + Send + 'static,
    F::Output: Send + 'static,
    S: Fn(Runnable) + Send + Sync + 'static;
```

FTL OS 中线程不需要返回值，因此 F::Output 是空。F 必须实现 Send，这保证了 FTL OS 自旋锁不能跨越 await。S 是一个参数为 Runnable 的函数，async-

task 保证调度器中只有一个 Runnable 实例。Runnable 是 async-task 内部实现类型，调用 schedule()方法能将自身按上述状态规则调用参数中的 schedule 函数，调度器能够调用 Runnable 的 run()方法来运行对应的 future。async-task 历经多次优化，相比于最简单的调度器，async-task 整个过程中只进行一次内存分配，并且能够在 future 运行结束时立刻析构而不是等待引用计数归零，完全由原子变量无锁实现相关逻辑，具有非常高的性能。

局部调度器与任务窃取

全局唯一的调度队列会导致较大的竞争开销，目前主流的工业调度器都加入了线程独立调度器来降低竞争。每个 CPU 都包含一个局部调度器，CPU 优先从局部调度器获取任务，任务为空时再从全局调度器加载任务；如果全局调度器为空，调度器将尝试从其他 CPU 的局部调度器窃取一些任务运行。使用局部调度器的除了降低锁竞争外，还可以让被唤醒的进程优先加入上一次运行局部调度器，再次利用 cache 资源。

异步上下文切换

每一个线程都有独立的上下文（如页表，关中断计数，调试栈），而这些上下文都需要在进入 future 时被加载，离开 future 时清空。在 xv6 中这也有体现，例如上下文切换前后需要刷表等。

有栈上下文切换通过调用函数的方式切换上下文，因此在上下文切换函数前后增加相关逻辑即可。但无栈上下文切换不同，函数在 await 处发生上下文切换，函数按调用栈一次弹出，这是不是说明无栈上下文切换不能切换更复杂的上下文呢？当然不是！但要进行上下文的切换，首先要理解 rust 对 future 的处理。

rust 无栈协程模型依赖 Future。Future 是标准核心库如下定义的 trait：

```rust
pub enum Poll<T> {
    Ready(T),
    Pending,
}
pub trait Future {
    type Output;
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output>;
}
```

Output 是 Future 的输出类型，poll 函数是这个 Future 的运行逻辑，返回的 Poll 表示 future 是否完成了执行。如果执行完成将返回获取到的值，如果未执行完成则返回 Pending 标志。

对于一个 async 函数，rust 事实上也会编译为一个 Future，根据函数体自行生成对应 poll 函数。生成的 poll 函数对于大部分情况已经足够，但如果我们要对上下文进行更细致的处理，必须手动编写 Future 来完成上下文切换。FTL OS 的进程上下文切换 Future 定义如下：

```rust
// kernel/src/process/userloop.rs

struct OutermostFuture<F: Future + Send + 'static> {
    future: F,
    local_switch: LocalNow,
}
impl<F: Future + Send + 'static> Future for OutermostFuture<F> {
    type Output = F::Output;

    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output> {
        let local = local::hart_local();
        let this = unsafe { self.get_unchecked_mut() };
        local.enter_task_switch(&mut this.local_switch);
        if !USING_ASID {
            sfence::sfence_vma_all_no_global();
        }
        let ret = unsafe { Pin::new_unchecked(&mut this.future).poll(cx) };
        if !USING_ASID {
            sfence::sfence_vma_all_no_global();
        }
        local.leave_task_switch(&mut this.local_switch);
        ret
    }
}
```

local_switch 是任务独立的上下文，使用 enter_task_switch()将当前 CPU 上下

文切换为此进程，并将源上下文保存在 local_switch 的位置。使用 leave_task_switch()函数将当前 CPU 上下文切换为 local_switch 保存的原上下文，并将任务独立的上下文保存回 local_switch。任务上下文处于另外分配的堆空间，因此切换上下文只需要修改指针，速度极快。sfence::sfence_vma_all_no_global()函数刷新用户页表，不会处理包含 G 标志位的内核全局映射。

利用泛型参数 F 可以将 future 成员和 local_switch 成员放入同一块内存，避免一次指针寻址。任务的具体实现在 future 成员中，因此上下文切换完成后将调用 future 的 poll 函数进行具体处理。

2.1.3 浮点数

开启浮点单元的流程概况如下：

- 在 mstatus 中开启浮点单元，此过程需要 SBI 代理。
- 设置 sstatus 中的浮点标志位为 FS::Initial / FS::Clean。
- 正常保存寄存器。

多核下每个核都要启用浮点运算。

通过 benchmark 获知，hifive unmatched 的 CPU 权限切换开销非常小（仅刷新流水线），主要开销为陷阱函数中的上下文保存与恢复。尽管 hifive unmatched 为双发射处理器，但访存端口只有 1 个，而浮点寄存器有 32 个，保存开销很大。如果能减少浮点寄存器保存的次数就能显著提高性能。

启用浮点后 sstatus 标志位中含有三种可能：init, clean, dirty。用户调用浮点运算时标志位会变为 dirty。

FTL OS 内核全局关闭浮点指令集，仅在浮点上下文保存时启用，保证操作系统内核态不使用浮点运算。FTL OS 实现了如下懒优化：

- 浮点寄存器只在线程切换时保存。
- 不进入用户态时，不加载浮点寄存器。

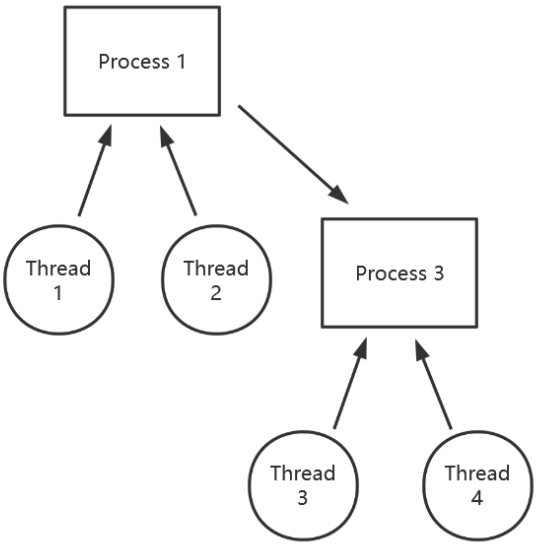
以上的优化针对两种场景，第一种为连续不涉及上下文切换的系统调用，例如反复获取 pid。由于上下文未切换，CPU 浮点寄存器不会发生变化，因此也不

需要保存与恢复。第二种优化的例子有两个进程通过管道传输传输大量数据，在数据传输结束前都不需要再次进入用户态，自然也不需要恢复浮点寄存器。实现第一种优化的方式是将寄存器的保存推迟到上下文切换，实现第二种优化的方式是将寄存器的恢复推迟到进入用户态之前，并通过 need_store 和 need_load 两个标志位来决定是否需要触发寄存器的保存与恢复。当上下文发生切换时，need_load 标志位将变为 1，而离开用户态时 need_store 变为 1，每次浮点寄存器的保存与恢复都会重置对应的标志位。

FTL OS 实现了更进一步的优化，即用户不使用浮点指令时不保存浮点寄存器。实现原理为用户使用浮点指令后 csr 的 fs 标志位将变为 dirty，此时将 need_store 设为 1 即可，否则保持不变。

2.1.4 进程管理

进程是操作系统为任务分配资源的基本单位，而线程是操作系统调度的基本单位。FTL OS 设计上进程可以被多线程所有，所有线程都是对等的，只有所有线程全部退出或执行了`exit`系统调用才会释放进程。这两种途径不会完全释放进程的内存，因为进程退出后需要留下为父进程留下必要的信息，进程还需要被父进程持有。因此进程的所有权树如下：



每个线程被所在的异步 Future 持有，Future 将不断地被调度器执行 poll，直到返回了`Ready`后被释放。线程将不断则进行用户循环：

```
// kernel/src/process/userloop.rs
async fn userloop(thread: Arc<Thread>) {
    // Future从参数捕获了thread
    loop {
        // 进入用户态
        ...
        // 离开用户态处理陷阱
        ...
        if exit {
            break; // 线程在这里离开的主循环
        }
    }
    // 执行到这后poll会返回Ready(()), thread将被释放
}
```

每个进程都具有唯一标识符 pid，目前 FTL OS 不支持进程生成额外的线程，因此 pid 在全局分配并通过 pid 获取线程的 tid，tid 和 pid 相同。但在支持多线程后需要改变获取 pid 的方式，先有线程获取 tid，再从 tid 获取 pid，tid 分配的唯一性保证了 pid 的唯一性。但初始线程 tid 的回收应该在进程释放时进行，因为初始线程释放后进程可能还被其他线程持有，此时如果立刻释放 tid 则下次产生的线程对应的进程会和旧进程拥有一样的 pid 并导致一系列错误。

FTL OS 目前阶段的线程只携带了用户地址空间的上下文信息，定义如下：

```
pub struct Thread {
    // never change
    pub tid: Tid,
    pub process: Arc<Process>,
    // thread local
    inner: UnsafeCell<ThreadInner>,
}
pub struct ThreadInner {
    pub stack_id: StackID,
    pub set_child_tid: UserInOutPtr<u32>,
    pub clear_child_tid: UserInOutPtr<u32>,
    pub signal_mask: SignalSet,
    uk_context: Box<UKContext>,
}
```

ThreadInner 包含了线程的局部信息。这部分信息被定义为只有线程自身才能访问，因此不需要加锁。目前线程局部信息只有 uk_context 被使用，其他字段处于保留状态。uk_context 处于内核态时保存了用户态的上下文，处于用户态时保存了内核态的上下文，是用户态和内核态的上下文交换区。

FTL OS 的进程控制块定义如下：

```
pub struct Process {
    pid: PidHandle,
    pub pgid: AtomicUsize,
    pub event_bus: Arc<EventBus>,
    pub alive: Mutex<Option<AliveProcess>>,
    pub exit_code: AtomicI32,
}
pub struct AliveProcess {
    pub user_space: UserSpace,
    pub cwd: Arc<VfsInode>,
    pub exec_path: String,
    pub envp: Vec<String>,
    pub parent: Option<Weak<Process>>,
    pub children: ChildrenSet,
    pub threads: ThreadGroup,
    pub fd_table: FdTable,
    pub signal_queue: LinkedList<SignalPack>,
}
```

可以看到进程控制块被分为了`Process`与`AliveProcess`两个部分，`Process`包含了进程的不可变信息，获取这部分信息不需要加锁。`AliveProcess`只有在进程退出之前才有效，进程退出时会变为`None`并在析构函数中释放所有持有的句柄。

Process 的各个字段定义如下：

字段	描述
PidHandle	进程 ID 句柄，析构时自动回收
pgid	进程组 ID
event_bus	事件总线，其他进程访问此进程的唯一途径
alive	进程运行信息，只能被所在进程访问
exit_code	进程退出码，只有进程退出后才有效

AliveProcess 的各个字段定义如下：

字段	描述
user_space	进程地址空间管理器，持有进程页表
cwd	当前命令行所在的目录
exec_path	程序执行时的目录
envp	程序执行时的环境变量
parent	此进程的父进程，如果为空说明是初始化进程
children	子进程集合
threads	运行在此进程的全部线程
fd_table	所有打开文件的映射表
signal_queue	信号队列，尚未启用

FTL OS 的事件总线能够从根本上防止死锁并提高效率，但目前还没有实现全部功能，仍然允许其他进程在退出时访问 AliveProcess 字段。但即使只允许当前进程访问自身 AliveProcess 然需要被锁保护，因为进程可以拥有多个线程。

2.1.5 陷阱与上下文切换

用户态陷阱

FTL OS 的用户态与内核态使用相同的页表，进入与离开用户态不需要清空 TLB。但除了这一点外，共享页表为更多的优化方式提供了可能。在 XV 6 与 rCore-tutorial 中，由于使用了用户态独立的页表，用户态与内核态的切换方式通过跳板页（trampoline）实现，用户态的现场保存就在这里进行。trampoline 通常映射在地址空间的最高位置，而在编译时通常放置在其他位置，因此从 trampoline 跳转至内核代码不能通过基于相对地址跳转的 call 进入内核，而必须从陷阱上下文中获取内核入口，通过寄存器跳转的方式进入内核。而当用户空间与内核空间共享页表时，陷阱入口可以直接设置为代码段中的陷阱入口函数，并直接通过 call 进入内核。

FTL OS 采用了无栈协程，因此没有为每个进程分配内核栈，使用了不同的用户态切换方式。XV 6 使用有栈上下文切换，因此一个 CPU 在从调度器进入用户态经过如下的两个过程：

1. 从调度态进入进程内核态，通过_swtch 函数实现，函数将切换所有 callee saved 寄存器，进入进程内核上下文。

2. 从 trap 以函数调用的方式进入用户态，加载用户态的全部寄存器与部分特权寄存器，内核态的寄存器上下文直接丢弃。

从用户态进入调度器也经过两个过程：

1. 从 trap 进入内核，保存所有用户态寄存器，加载内核栈指针等必要上下文，进入内核处理函数，处理函数处于内核栈的栈底。

2. 通过`_swtch`函数切换所有 callee saved 寄存器，返回到调度器函数。

在这种处理方式下，进程内核态的处理就好像是用户在调用一个有状态的函数，尽管这个函数没有运行在用户栈上。

在基于无栈上下文切换的 FTL OS 中这一切都有所不同，从调度器进入用户态经过的两个过程如下：

1. 从调度态进入进程内核态，通过 Future 的 poll 函数实现。这部分实现由编译器完成的函数调用，没有寄存器的保存，在代码看来则是从 await 处开始执行。

2. 从__enter_user 函数进入用户态。保存内核态的全部 callee saved 寄存器，加载用户态的全部寄存器与必要上下文。

从用户态进入调度器的过程如下：

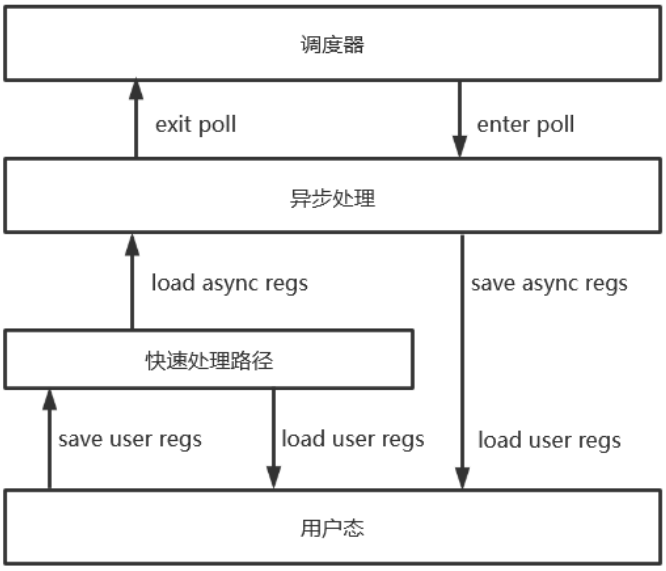
1. 保存所有用户寄存器，加载内核态的 callee saved 寄存器与必要上下文，从__enter_user 函数回到内核态。

2. 在 await 处退出正在执行的 poll 函数，回到调度器。

可以看到在无栈上下文切换中，进入用户态就好像是内核态在调用一个有状态的函数，尽管这个函数没有运行在内核栈上。

内核态处理快速路径

对于无栈上下文切换，进出用户态需要保存在有栈上下文切换时不需要保存的 callee saved 寄存器，就好像在过程 1 中一次性完成了有栈上下文切换的过程 1 与 2，由于需要保存与恢复更多的寄存器，无栈上下文切换的用户态-内核态切换速度比有栈上下文切换慢。对这一速度劣势有一种优化方法，就是结合有栈上下文的切换方式，为每个线程分配一个“快速路径栈”。从异步上下文可以直接进入用户态；但从用户态进入陷阱时会先进入快速处理路径，如果快速处理路径可以完成处理将直接回到用户态，如果无法处理将回到异步上下文。这种方式下保存用户寄存器于保存异步上下文寄存器被拆分到了用户态与快速处理路径的跳转和快速处理路径与异步上下文的跳转，没有额外的寄存器开销。对于如 getpid 等系统调用，快速处理路径可以避免大量不必要的 callee saved 寄存器加载与保存。初赛阶段此优化尚未应用于 FTL OS，将在未来应用于内核。



这种方案需要分配一个内核栈，由于快速处理路径不需要太复杂的逻辑，4 KB 的内核栈已经足够了。

用户地址测试

发生系统调用时内核需要接收指向用户地址的指针，而此地址可能有效也可能无效，内核需要验证此指针指向的地址是否有效才能使用。由于用户地址处于低位地址，内核地址处于高位地址，因此首先需要测试指针是否指向内核高位地址。但指向低位地址并不意味着指针是安全的，因为这个指针可能处于未映射的内存，或者尝试向只读区域写入数据。用户发生页错误是安全的，但一旦在内核态发生页错误很可能直接导致内核崩溃。通常内核为了安全性不会直接用用户指针访问数据，而是在获取用户数据时从用户态复制到新的内存区域，写入数据时在特定函数完成写入，通常这是通过页表直接完成的。

为了提高性能，FTL OS 采用了直接解引用用户指针的方式来传递用户态数据。为了安全地解引用用户态数据，FTL OS 将在操作数据之前在指定函数对地址进行测试。在测试时内核不会查询页表，而是直接对指定页进行单字节的读与写并捕获内核产生的异常，这种方式在数据已经映射的情况下不会产生任何开销，但即便发生了异常，由于访存被限制在了特定区域，产生的异常类型是可以预测的，因此异常处理时不需要保存任何上下文，甚至可以将其看作一个特殊函数调用。

FTL OS 的访存测试函数为如下：

```
# kernel/src/user/check_impl.S
__try_read_user_u8:
    mv a1, a0
    mv a0, zero
    lb a1, 0(a1)
    ret
__try_write_user_u8:
    mv a2, a0
    mv a0, zero
    sb a1, 0(a2)
    ret
```

__try_read_user_u8 与__try_write_user_u8 只有一次字节访存，没有任何栈上操作，当页表正常映射时速度极快。

FTL OS 的异常捕获函数为如下：

```
# kernel/src/user/check_impl.S
try_access_user_error:
    csrr a0, sepc
    addi a0, a0, 4
    csrw sepc, a0
    li   a0, 1
    csrr a1, scause
    sret
```

如果异常没有发生，那么 a0 寄存器会被设置为 0，否则将在异常中被设置为 1。检查每次步长为 4 KB，直接利用 CPU 的 MMU 硬件资源完成地址检查。需要注意的是这种检查方式替换了内核陷阱函数，而在遇到中断时会修改 a0 或 a1 寄存器导致内核不稳定甚至崩溃。一种解决此问题的方法是在检查用户地址时关中断，而这会增加中断响应时间。FTL OS 使用了另一种方式，即使用向量陷阱模式。在向量陷阱模式下如果发生了异常，PC 寄存器会被设置为 stvec.base，而如果发生了中断，PC 寄存器会被设置为 stvec.base + 4 × scause。向量陷阱模式可以让异常和中断进入不同的处理函数，这样就可以保证异常处理函数只会在固定位置执行了。

用户态数据访问

用户地址空间通过测试后 FTL OS 将假定这部分映射永远有效，未来可能会增加锁定逻辑来避免空间在操作完成之前被提前释放。目前 FTL OS 处于用户数据操作状态时会忽略所有由用户地址错误引发的页错误并延迟到读取结束时处理。由于用户态的数据仅涉及数据而不涉及逻辑，忽略页错误只会降低操作系统速度并读出错误的数据，不会导致操作系统崩溃。

操作系统访问用户数据时处于内核态，要直接通过用户态数据指针访问数据需要将 sstatus 寄存器中的 sum 位设为 1。FTL OS 通过 RAII 管理此标志位，即访问用户数据需要持有 AutoSum 句柄来设置 sum 标志位，AutoSum 析构时会释放 sum 标志位。但 sum 标志位的释放不是通过 AutoSum 自身管理的，而是

将相关信息放置在了线程上下文并使用栈的方式管理，这样就能允许 AutoSum 被嵌套使用并在多个异步任务之间互不影响。

上述实现方式的 AutoSum 句柄能够跨越 await，因此 FTL OS 绝大多数的系统调用都不会为用户数据分配内存，而是直接解引用用户态指针，这种方式能够将数据传输速度提高一倍。以通过管道将数据从写者传至读者为例，分配内存的方式中数据经过了用户态-临时内存-管道缓冲区-临时内存-用户态的变化，发生了 4 次复制与两次内存分配，而 FTL OS 中为用户态-管道缓冲区-用户态，只经过了两次复制，没有任何内存分配。

2.2 内存管理

2.2.1 页表映射

页表设计

Hifive Unmatched 开发板文档说明支持 ASID，FTL OS 也实现了基于 ASID 的地址空间切换优化。但实际上板测试中 ASID 无效，因此目前禁用了 ASID 优化并在页表操作前后增加了刷表指令。Hifive Unmatched 开发板相比 K210 使用了更新版本的 RISC-V 指令集，在硬件上支持 sfence.vma 指令，不需要 sbi 代理，因此执行 sfence.vma 只需要刷新处理器流水线(约 10 时钟)，速度非常快。尽管如此开销大幅下降，但 TLB 未命中带来的开销依然较大。

FTL OS 的页表同时映射了用户段和内核段，在用户态和内核态切换时不切换页表，只有在切换页表或页表修改时才需要刷表指令。

Hifive Unmatched 开发板支持 16 GB 的内存，而内核入口地址为 0x80200000。如果将 0x80000000 作为用户数据和内核数据的分界线，用户虚拟地址将只能获得 2 GB 的空间，单个进程无法充分利用 16 GB 的内存。为了充分使用硬件资源，FTL OS 将内核段划分至 0xffff_ffc0_0000_0000 - MAX，用户段划分至 0x0 - 0x40_0000_0000，用户段和内核段都拥有 256GB 的空间，可以进行更进一步的划分。

进入内核

内核段重新映射后不可以从入口地址(0x80200000)直接 call/j 跳入内核入口函数，因为编译器的链接基址为 0xffffffff80200000，在编译器看来运行在这里时 PC=0xffffffff80200000，但实际上 PC=0x80200000！能够正常运行仅仅是因为两个地址对齐了而已。普通的跳转方式使用 auipc 进行相对地址跳转，而从 0x80200000 地址相对跳转只能进入虚空。因此必须先将内核地址映射到高位地址，切换页表后再通过 jr 跳转至内核入口。

如何在 entry 函数中完成完整的高位内核映射？用汇编写一个页表映射函数？事实上 sv39 不仅支持 4 KB 的页映射，还支持 2 MB 和 1 GB 的巨页表映射。利用巨页表即可方便地将物理地址映射到高位地址空间：

```
    # kernel/src/hart/boot/entry64.asm
    .align 12   # page align
boot_page_table_sv39:
    # 0x00000000_80000000 -> 0x80000000 (1G) 2 physical mapping
    # 0xffffffc0_80000000 -> 0x80000000 (1G) 450 kernel direct mapping offset
    # 0xffffffff_80000000 -> 0x80000000 (1G) 510 kernel link
    # -2- <2> -447- <450> -59- <510> -1
    .quad 0
    .quad 0
    .quad (0x80000 << 10) | 0xcf # VRWXAD
    .zero 8 * 447
    .quad (0x80000 << 10) | 0xcf # VRWXAD
    .zero 8 * 59
    .quad (0x80000 << 10) | 0xcf # VRWXAD
    .quad 0
```

boot 页表在映射了高位地址的同时还保持了物理地址的直接映射，利用它就可以正常进入内核入口函数 rust_main 了。显然这个页表的映射太过于粗糙，因此需要在内存初始化中替代为更加精细的页表。

sv39 支持 39 位地址空间，寻址时高 25 位需要和第 39 位相同，否则将产生页错误。利用高位 0/1 划分为用户区和内存区，两个区域的地址空间都是 256 GB。目前 FTL OS 的虚拟内存映射如下：

内核段

描述	起始地址	结束地址
全部映射(256 GB)	0xffff_ffc0_0000_0000	MAX
硬件IO地址(1 GB 未使用)	0xffff_ffff_c000_0000	0xffff_ffff_ffff_f000
内核数据段(1 GB)	0xffff_ffff_8000_0000	0xffff_ffff_c000_0000
直接内存映射(32 GB)	0xffff_fff0_0000_0000	0xffff_fff8_0000_0000
文件映射 (64 GB 未使用)	0xffff_ffd0_0000_0000	0xffff_ffe0_0000_0000
虚拟内存映射(64GB 未使用)	0xffff_ffc0_0000_0000	0xffff_ffd0_0000_0000

用户段

描述	起始地址	结束地址
全部映射(256 GB)	0x0	0x40_0000_0000
链接基地址	0x10000	0x10000
数据段(32 GB)	0x10000	0x8_0000_0000
堆段(32 GB)	0x8_0000_0000	0x10_0000_0000
线程栈段 (64 GB)	0x10_0000_0000	0x20_0000_0000
文件映射 (128 GB)	0x20_0000_0000	0x40_0000_0000

2.2.2 映射管理器

xv6 使用硬编码方式在 usertrap 中根据地址的值进行对应处理，因此每个地址只能用于特定的用途，并极大地增加维护成本。FTL OS 希望能够有一种解耦的方式来管理用户地址空间，增加功能时不再改动代码的其他区域。linux 显然不会和 xv6 一样处理地址映射。linux 对每个进程维护了一个链表，链表保存了用户的每一个数据段，发生页错误时只需要在页表中查询对应的数据段，进行相关处理即可。

FTL OS 使用 rust 编写，相比 C 语言的人工虚表，rust 提供了 trait 作为动态接口，具有更高的开发效率。FTL OS 的 handler trait 定义如下：

```rust
// 仅展示了部分接口，详细实现在 kernel/src/memeory/map_segment/handler/mod.rs
pub trait UserAreaHandler: Send + 'static {
    fn id(&self) -> HandlerID;
    fn perm(&self) -> PTEFlags;
    fn map_perm(&self) -> PTEFlags;
    fn user_area(&self, range: URange);
    fn unique_writable(&self) -> bool;
    fn using_cow(&self) -> bool;
    fn shared_always(&self) -> bool;
    fn may_shared(&self) -> Option<bool>;
    fn executable(&self) -> bool;
    fn init(&mut self, id: HandlerID, pt: &mut PageTable, all: URange) -> Result<(), SysError>;
```

```
    fn max_perm(&self) -> PTEFlags;
    fn new_perm_check(&self, perm: PTEFlags) -> Result<(), ()>;
    fn modify_perm(&mut self, perm: PTEFlags);
    fn map(&self, pt: &mut PageTable, range: URange) -> TryR<(), Box<dyn AsyncHandler>>;
    fn copy_map(&self, src: &mut PageTable, dst: &mut PageTable, r: URange) -> Result<(), SysError>;
    fn page_fault(&self, pt: &mut PageTable, addr: UserAddr4K, access: AccessType)
        -> TryR<DynDropRun<(UserAddr4K, Asid)>, Box<dyn AsyncHandler>>;
    fn unmap(&self, pt: &mut PageTable, range: URange);
    fn unmap_ua(&self, pt: &mut PageTable, addr: UserAddr4K);
    fn split_l(&mut self, _addr: UserAddr4K, _all: URange) -> Box<dyn UserAreaHandler>;
    fn split_r(&mut self, _addr: UserAddr4K, _all: URange) -> Box<dyn UserAreaHandler>;
    fn box_clone(&self) -> Box<dyn UserAreaHandler>;
}
```

上述实现包含了令人困惑的东西，例如 *AsyncHandler* 。*AsyncHandler* 是一个异步句柄，通过它可以进行异步页错误处理，因为 FTL OS 的映射管理器将页错误处理分为了两个阶段，第一部分是同步页错误处理阶段，当同步页错误处理涉及异步调用时将返回异步句柄，再进入异步页错误处理阶段。

为什么要将页错误处理分为两部分，而不是将 page_fault 函数改为异步函数，在一个函数中完成整个处理？

因为获取 handler 的唯一方式是通过映射管理器，而若要操作映射管理器必须先获取映射管理器的锁。为了提高响应速度，映射管理器使用自旋锁保护，而 page_fault 函数持有了一个页表的可变引用，而获取页表可变引用的唯一方式是持有映射管理器的锁。因此如果 handler 是异步调用，当 handler 发生上下文切换时，上下文切换后映射管理器的自旋锁不会释放，导致非常严重的锁竞争与多线程死锁。由于 FTL OS 使用无栈上下文切换，卡在自旋时不可能由上下文切换出让 CPU，此时内核出了重启别无他法。

rust 虽然不能让上边死锁的内核活过来，但可以让写出这种代码的内核无法编译。方法非常简单：异步调度器要求异步函数必须满足 Send 的 trait，但显式撤销自旋锁 guard 的 Send trait。异步函数有确定的切换点 await，当异步函数要求满足 Send 时，所有越过 await 的实体也必须满足 Send trait。当我们持有未实现 Send 的自旋锁跨过 await 时异步函数的 Send trait 也跟着被撤销了，因此整个异步任务再也无法放入异步调度器，内核就无法编译了。

既然异步函数不能持有自旋锁，那 *AsyncHandler* 是如何修改页表的？

AsyncHandler 不在参数中获取可变引用，而是在参数中捕获进程指针，写入阶段才获取锁，不涉及自旋锁跨越 await。

如何防止线程 A 在页错误进行文件读入时，映射被线程 B 修改，之后线程 A 对页表的无效位置进行映射？

考虑增加 map 区域 64 位版本号，每次映射新的区域都会修改版本号。文件读入完成后先检测版本号再修改页表。

映射管理器实现

handler 统一了用户地址映射处理，可以方便地实现地址映射了。不局限于单进程映射，FTL OS 希望用一套机制处理进程间页面共享，无论是只读的还是可写的。但在进一步设计之前需要决定使用段式管理还是页式管理。

段式管理的优点是可以一次性处理大段地址，但分裂较麻烦。页式管理占用更多的内存，如果涉及引用计数还将大大降低吞吐量，但没有分裂的问题。

由于用户使用 mmap 对页表的操作都是段操作，handler 采用段式管理，并提供两侧的分段操作。hander 代表整个段，从处理函数参数获取待处理的地址。

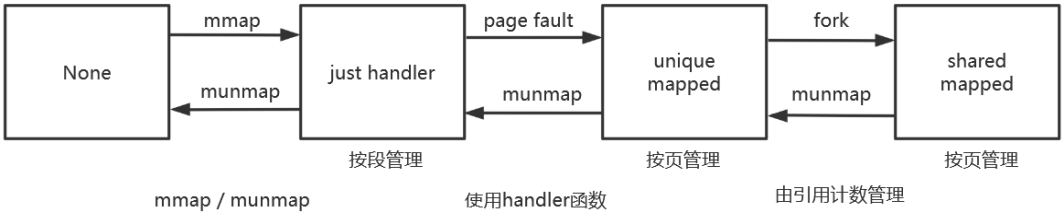
由于页错误都是按页发生的，已经映射的页面采用页式管理。单进程所有的页面由页表管理，共享页表使用引用计数由独立管理器管理。

FTL OS 的页面具有两个状态：

1. 唯一状态。唯一状态的页的只被一个进程持有，来自 handler 页错误/共享状态，可以被 handler 直接回收。
2. 共享状态。共享状态的页可能被多个进程持有，不存在单进程的共享状态，并使用原子引用计数维护，不能被 handler 直接回收，它在页表项中包含共享标志位。
3. 页错误处理成功后将唯一页表项插入页表。如果发生 fork，页面将从唯一状态转换为共享状态，并提交至共享映射管理器，在堆上分配一个引用计数节点。

进程回收或 munmap 将删除映射管理器中的 handler，删除 handler 前将先撤销共享映射，撤销时原子递减引用计数，当引用计数变为 1 时使用 handler 释放

页表。撤销共享映射完成后撤销唯一映射，直接使用 handler 对应释放函数即可。映射管理器代理了绝大多数页表操作，实现了共享映射和映射段的完全解耦，让映射段只需要处理最简单的情况。映射管理器可以方便地引入虚拟内存，在内存不足时将部分页面放入磁盘交换区。



为了提高 fork 的处理速度，FTL OS 使用了页表项中的一个保留位来标识共享页，这样就可以将唯一页和共享页的处理合并至同一个逻辑中，通过共享标志位选择相应的处理方式。handler 也可以禁用共享页表，在这种情况下页表管理将申请新的内存并产生唯一映射页，对部分用户栈使用申请新内存的方式可以减少页错误的数量，提高性能。

FTL OS 支持映射不可访问的页，不可访问的页依然会分配内存。为不可访问页分配内存的原因是操作系统支持直接修改映射页的标志位，不可访问的页可能在未来被修改标志位进入可访问状态。

相对于 linux 使用链表储存 handler，FTL OS 将 BTreeMap 封装为段映射容器，并提供了与段相关的一系列操作。相对于链表，段映射容器提供了 O(log N)的增删查改操作，当映射数量较大时具有明显性能优势，但内存开销更大且需要更多次内存分配。

handler 实例

FTL OS 目前为不同场景实现了如下的 handler：

handler	描述
懒分配	映射时不写入任何数据，初始化段时不进行任何操作
直接映射	映射时不写入任何数据，初始化段时将整个段映射至页表
文件映射	懒分配的文件映射，发生页错误时读取文件并映射至页表，释放时按需写回

2.2.3 内存分配器

常见内核都使用 buddy 伙伴内存分配器，它以 2 幂划分空闲内存至各个集合，分配内存时从对应大小的集合取出一块，如果不能取出则从上级集合分裂出对应大小的内存。释放时放回对应的集合，并判断集合中是否存在可以合并的另一块内存，如果存在则合并这两块空间，加入合并后对应的集合。

rCore-tutorial 采用了第三方库的 buddy_system_allocator 分配器作为全局的堆分配器。此分配器使用链表管理同大小的集合，默认分配器内部的内存块不存在其他的引用，直接在每个内存块头部保存一个指针，利用此指针将各个内存连成链表，为了保存指针，最小的可分配空间为 8 字节。buddy_system_allocator 释放内存的合并策略是扫描整条链表寻找伙伴，因此每次释放操作的时间复杂度是 O(N)，内存碎片越多性能越低。

TCmalloc 是 google 开发的高速内存分配器，使用线程缓存技术在每个线程放置了内存分配缓存，从分配缓存中分配内存不需要获取锁。

FTL OS 内存分配器实现

FTL OS 希望构造一个高速的内存分配系统，希望兼顾性能和安全时尽可能简单。

管理内存需要建立内存页之间的数据结构，数据结构信息有三种放置方式：

1. 被管理的内存内部（buddy_system_allocator 方式）
2. 被管理的内存之前（libc malloc 方式）
3. 放置于其他内存块

第二种方式是绝大多数内存分配器的管理方式，这种用户不小心向被释放的内存写入数据也可以基本保证数据结构不被破坏，释放内存时不需要传入空间的大小。但如果内存是利用伙伴分配器分配的，这种方式会导致极大的实际空间开销，例如内存页要求对齐至 4 KB，为了额外放置元信息的空间，伙伴分配器不得不分配 8 KB 的空间，空间直接损耗了 50%！

FTL OS 相信 rust 的内存安全性，永远相信不会使用被释放后的内存，因此使用了将信息放在被管理内存内部的方式。rust 的内存分配与释放也和 C/C++不同，绝大多数内存都通过 Box 智能指针分配，可以从类型获取内存释放信息，编译器将自动向分配函数与释放函数中填入分配信息。

FTL OS 使用了 TCmalloc 的线程局部分配器优化，CPU 内部都拥有分配缓冲区，cache 空时继续分配则从全局分配器调入内存。cache 满时释放一半内存。buddy_system_allocator 的 O(N)内存释放是不可接受的，是否有更高效的实现方法？

1. 定期回收策略

每次释放内存都遍历链表效率太低，修改定期回收策略。对每个链表，设上一次回收后的长度为 N，取一合适的 k 值（如 2），当释放内存使链表长度到达 kN 且 kN 大于某个常数 C 时进行回收。合并段的方式为先整体排序，临时调入最大长度的连续内存，将整个链表写入这个段后使用随机访问的快速排序，合并完成后写回链表。如果连续内存空间不足可以使用链表上的快速排序，性能相对随机访存的快速排序会明显下降。回收的时间复杂度是排序的 O(NlogN)，平均复杂的降低至 O(logN)。分配内存的复杂度为 O(1)，且不会在大量重复分配释放时出现大量拆分合并，这是 FTL OS 目前采用的优化策略。

2. 使用更快的容器

可以使用时间复杂度更低的容器来替换缓慢的链表。树状数据结构会将释放内存的复杂度降低至 O(logN)，但也将申请内存的时间复杂度提高至 O(logN)。集合的节点需要使用其他动态分配的方式，这会产生更低的内存开销。

2.2.4 RCU 系统

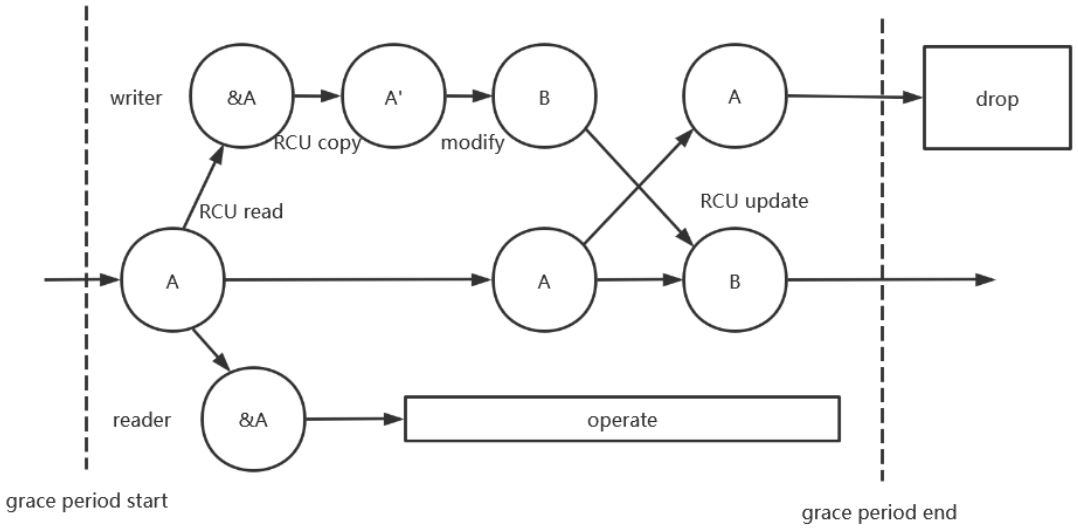
RCU(Read-Copy Update)，顾名思义就是读-拷贝-修改。对于被 RCU 保护的共享数据，读者不需要获得任何锁就可以访问它，但写者在访问它时首先拷贝一个副本，然后对副本进行修改，修改完成后把指向原来数据的指针替换为指向新

数据的指针，最后使用一个回调函数在所有引用该数据的 CPU 都退出对共享数据的访问时释放旧数据的内存。

对于需要频繁读但是不需要频繁写的共享数据 RCU 是一种非常成功的同步机制。它可以实现多线程无阻塞地读取数据，就算有线程在修改数据时读者也不会被阻塞，且读取数据时除了内存屏障外没有额外同步开销，不会读取到不完整的数据，尽管数据可能不是最新的。

RCU 实现的关键是确定释放旧数据时所有读者都必须完成访问。通常把写者开始更新，到所有读者完成访问这段时间叫做宽限期（Grace Period）。如果不对读者作任何限制那么宽限期将是无尽的，但我们可以合理限制读者的最大访问作用域。linux 中 RCU 读者作用域的限制为不能经过抢占点，因此在抢占发生之前所有通过 RCU 解引用的指针都是有效的。

FTL OS 使用了无栈上下文切换，因此抢占只可能在`await`处发生。rust 编译器强大的生命周期分析能力可以在编译时发现代码的错误，和自旋锁相同，FTL OS 将 RCU 读者的 guard 标记为`!Send`，于是通过`rcu_read()`获取的指针都无法越过`await`了，保证了安全性。下图展示了 RCU 的更新过程：



图中读者获取 A 的引用后可以一直使用至宽限期结束，在这期间 A 的内存不会被释放，写者的修改也不会作用于 A。A 被替换为 B 后其他核不再可以访问到旧的 A，但已经持有的 A 的引用可以持续使用，尽管 A 的值并不是最新的版本，在大多数情况下使用旧的数据不会有太大的负面影响。不可变数据结构天然

适用于 RCU 更新，因为它完全没有上图中修改过程，直接在原数据上创建新的数据结构，开销极小。基于引用计数的不可变数据结构会进行大量原子操作降低性能，但不可变数据结构可以在不 STW 的情况下使用图可达性搜索，只要保证当前数据结构只有一个线程在回收即可。

RCU 接口

FTL OS 使用了非常简单且高效的方式来实现 RCU 操作，接口定义如下：

```rust
// ftl-util/src/rcu/mod.rs
pub struct RcuDrop(usize, unsafe fn(usize));

pub trait RcuCollect: Sized + 'static {
    fn rcu_assert();
    fn rcu_read(&self) -> RcuReadGuard<Self>;
    unsafe fn rcu_write(&self, src: Self);
    unsafe fn rcu_transmute(self) -> RcuDrop;
    fn rcu_drop(self);
}
```

这里只展示了 RcuCollect 的接口定义，实际代码中 RcuCollect 提供了完善的默认实现，任何满足 RCU 要求的类型都可以实现 RcuCollect 且不需要重写任何函数。这几个函数的作用如下：

函数名	描述
rcu_assert	判断类型是否满足 RCU 要求，不满足则 panic。
rcu_read	获取一个 RCU 临时对象，此对象不能跨越 await。
rcu_write	将旧数据更新为新数据，旧数据调用 rcu_drop 释放。
rcu_transmute	类型擦除并获取析构函数
rcu_drop	使用处于全局空间的 self::rcu_drop 函数进行 RCU 延迟析构

所有函数在调用之前都会使用 rcu_assert 判断此类型是否满足 RCU 条件，如果不满足将 panic。rcu_assert 保证了此类型的 load 和 store 操作都可以用一条指令完成，即内存读写是原子的。绝大多数 RCU 对象是指针，FTL OS 对 Box、Arc、Weak 这三个 alloc 库提供的智能指针都实现了 RcuCollect，满足绝大多数使用场景。一般来说用户只需要使用 rcu_read 和 rcu_write 两个函数，其他函数主要被 RCU 释放系统使用。

RCU 释放系统

RCU 释放系统追踪每一个宽限期，当每一个 CPU 都离开当前宽限期时释放注册此宽限期上的 RCU 对象。管理系统的定义为：

```rust
// ftl-util/src/rcu/mod.rs
pub struct RcuDrop(usize, unsafe fn(usize));
// ftl-util/src/rcu/manager.rs
pub struct RcuManager<S: MutexSupport> {
    flags: AtomicU64,
    rcu_current: SpinMutex<Vec<RcuDrop>, S>,
    rcu_pending: SpinMutex<Vec<RcuDrop>, S>,
}
```

RcuDrop 封装了元素对象与对应的释放虚表，当管理器释放此对象时对数据应用虚表函数即可。

RCU 管理器将宽限期划分为两部分，当前宽限期和下个宽限期。flags 原子变量将追踪当前宽限期，支持最多 32 个 CPU 同时运行。flags 划分为高 32 位和低 32 位，高 32 位为处于当前宽限期的 CPU 集合，低 32 位为处于当前宽限期或下个宽限期的 CPU 集合。rcu_current 字段是当前宽限期结束时将释放的对象集合，rcu_pending 是下个宽限期结束时将释放的对象集合。第 x 号 CPU 每次进入宽限期都会将 flags 中的下一个宽限期对应位置（第 x 位）原子地设为 1，离开宽限期时将当前宽限期和下个宽限期的对应位（第 x 位和第 x+32 位）同时原子地置为 0。这里需要作一些特殊判断，如果当前宽限期只有当前 CPU 所属的位为 1，flags 会左移 32 位，再设置 x+32 位为 1，保证在当前 CPU 回收完成时不会有另一个 CPU 也进入回收状态。回收时将移动 rcu_pending 至 rcu_current，但取出的待回收对象不会马上回收，而是先将 flags 的 x+32 位置为 0 后再进行回收。这是因为回收状态的持续时间可能很长，在这期间积累的与 RCU 释放系统的对象都是不可回收的。而如果在回收之前将 flags 的 x+32 位置为 0 则允许多个 CPU 同时进行回收，除了提高了速度之外还能防止注册速度超过回收速度导致内存不足。离开宽限期时对 flags 的操作为 CAS 操作，保证了原子性。

第 x 个 CPU 离开宽限期的过程如下：

1. 以 Relaxed 内存序读取`flags`并命名为 v。
2. 将 v 的 x 位和 x+32 位置为 0。
3. 如果此时 v 的高 32 位为 0，v 左移 32 位，将`release`标志设为 true。

4. 无锁读取 rcu_pending 与 rcu_current 的长度，如果任意一方的长度不为 0 则将 need_release 标志设为 true，并将 v 的第 x+32 位置为 1。

5. 使用 CAS 更新 flags，如果更新失败则返回 1。

6. 如果 release 和 need_release 之一为 false，离开宽限期结束，否则进入回收流程。

7. 有锁地取出 rcu_pending 并将其替换至 rcu_current，保存 rcu_current 的旧数据。

8. 将 flags 的第 x+32 位置为 0。

9. 释放 rcu_current 的旧数据，回收结束。

flags 置为 0 时高 32 位可能为 0，此时不会从 rcu_current 继续获取数据，因为严格按时释放没有太大意义，在这期间注册到释放系统的对象可以等待至下一次回收。

RCU 竞争优化

RCU 释放队列是全局竞争的，这会导致并发性能下降。但 RCU 析构对象没有必要立刻提交到 RCU 释放系统中，可以先缓存在 CPU 的局部队列中，等待宽限期结束再统一提交至 RCU 释放系统。RCU 释放系统也没有必要每次发生上下文切换都离开宽限期，可以设置一个阈值，只有队列长度超过阈值时才释放的对象，在队列长度超过阈值之前不关闭宽限期，此优化可以在 RCU 释放队列长度达到阈值之前不产生任何原子开销，但可能会增加内存占用。

RCU 读取的安全性

RCU 读数据分为两个阶段，首先是从内存中取出指针，再从指针中读取数据。看起来这不会遇到任何并发问题，但事实确实如此吗？

部分处理器使用一种极端激进的优化：值预测。使用值预测的处理器会先取出预测指针指向对象的数据，然后再读取指针，最后判断预测指针和实际指针是否相等。值预测是现代 CPU 的优化方向之一，可以直接破除数据依赖导致的性能下降，速度可以提高数倍以上，但也会导致存在数据依赖的访问乱序。为了确保 RCU 读取的绝对安全，取出指针后需要添加 Acquire 屏障再进行数据读取。

添加内存屏障可以保证读取数据的正确性，但从内存取出指针有没有可能出问题呢？以 64 位平台为例，在 C/C++中所有指针都是 8 字节的，可以由一条指令完成访问。rust 则不相同，rust 基于智能指针完成资源回收，通常智能指针也是 8 字节的，但对于包含虚表的动态类型的智能指针，C++的做法是将虚表放在成员内部，智能指针维持 8 字节，而 rust 将虚表放在了智能指针结构体内（胖指针），这样做可以减少一次寻址，但智能指针变成了 16 字节，无法原子地由一条指令完成读取，RCU 读无法保证安全性。不可能用锁来保证 RCU 读的安全性（用了锁还要 RCU 干什么？），有两种方式可以解决大对象的问题：

- 维持单指令读取指针，将大对象用 8 字节的智能指针再封装一层。
- 仿造无锁数据结构的读操作，在内存屏障前后分别读取两次并比较，如果不相等则重试。

FTL OS 使用了第一种方案，因为第二种方案将导致 RCU 操作函数从 8 字节放开至 16 字节，降低速度，而目前尚无 RCU 回收胖指针的需求。第一种方案没有重试操作，处理过程符合正常认知且速度更快。

2.3 同步系统

2.3.1 锁

FTL OS 实现了自旋排他锁，自旋共享锁，睡眠排他锁，睡眠共享锁四种锁，分别应用在不同的场景。

自旋排他锁

FTL OS 的自旋排他锁实现非常简单，核心代码如下：

```
// ftl-uitl/src/sync/spin_mutex.rs
fn obtain_lock(&self) {
    while self.lock.compare_exchange(false, true, Ordering::Acquire, Ordering::Relaxed).is_err() {
        while self.lock.load(Ordering::Relaxed) {
            core::hint::spin_loop();
            ... // deadlock detect
        }
    }
}
fn drop(&mut self) {
    self.mutex.lock.store(false, Ordering::Release);
    S::after_unlock(&mut self.guard);
}
```

obtain_lock 使用 CAS 操作更新原子变量 lock，当值为 false 时原子地修改为 true，这是自旋锁实现的通用方法。drop 函数中将 lock 写入 false，这两处代码是唯一会修改 lock 字段的代码，而当 lock 为 true 时 obtain_lock 无法修改 lock，因此析构函数对 lock 的修改一定会被 obtain_lock 观测到。CAS 操作失败时执行的等待操作为普通的访存，自旋等待时不会如 CAS 操作一样反复广播原子操作导致性能下降。FTL OS 设计上不存在长时间占有的自旋锁，需要长时间占有对象应该由睡眠锁保护。FTL OS 在自旋次数过多时（约 1 秒）将触发 panic，这意味着内核发生了死锁。

自旋读写锁

FTL OS 自旋读写锁是读者优先锁，状态定义如下：

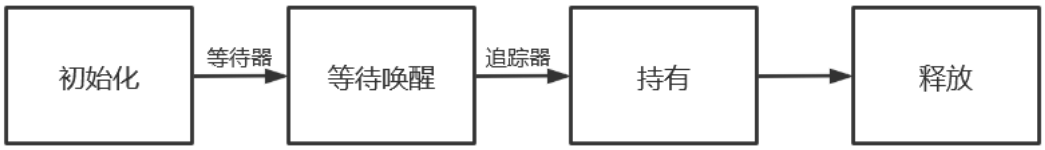
lock 的值	定义
-2^31	非法值
-2^31+1	读写锁被写者占有，没有正在等待的读者
[-2^31+2, -1]	读写锁被写者占有，有正在等待的读者
0	锁未被占有
[1, 2^31-1]	读写锁被读者占有，数量等于 lock 的值

FTL OS 的读者在获取读锁时会直接使用 fetch_add 操作将 lock 的值增加 1，即使锁处于写锁状态也进行操作，然后才等待写者将锁释放。这样的实现相比在写者释放后进行原子操作将原子开销隐藏在了写者占有锁的时间段内。相比于排他锁，读写锁可以为读者提供更高的并行度，但释放锁时也需要一次原子操作，两者开销需要权衡。

睡眠锁

自旋锁可以在多核环境下有效保护数据的安全，但在竞争严重的环境下会导致 CPU 资源的极大浪费。而睡眠锁在获取锁失败时将出让 CPU，可以有效地提高 CPU 资源的利用率。为了深度融合 rust 的 async 架构，FTL OS 自行实现了异步睡眠锁，当睡眠锁结束占用自动唤醒下一个等待的任务，唤醒顺序严格等于提交顺序，且运行时不需要分配任何内存。

睡眠锁包含三个部分：睡眠锁本体，睡眠锁等待器，睡眠锁追踪器。等待器和追踪器对应了睡眠锁上锁的两个阶段：等待阶段和持有阶段。而等待阶段也被划分为了两个部分：初始化阶段和唤醒阶段。在初始化阶段睡眠锁将产生一个等待器，如果在初始化时睡眠锁处于上锁状态，等待器会将自身注册到睡眠锁的等待队列中并出让 CPU。当持有睡眠锁的任务释放它的追踪器时会从睡眠锁等待队列中唤醒一个任务。被唤醒的任务将进入持有阶段，获取睡眠锁追踪器并释放等待器。相比于 Linux 的睡眠锁，FTL OS 在唤醒阶段不需要获取睡眠锁的链表锁，被误唤醒时开销极小。



为了实现不需要额外分配内存的睡眠锁，睡眠锁将等待器放置在“逻辑栈”上。所谓“逻辑栈”是因为等待器跨越了 await，空间被分配在了等待器所属 Future 的堆上，但我们依然可以将它看作在“逻辑栈”上，它所需的内存在所属 Future 分配内存时被一同分配了，没有额外内存分配。

等待器上包含一个侵入式链表节点，初始化时链表节点将会添加到睡眠锁等待队列的链表末尾。等待器上还携带了表示是否允许运行的标志位，这个标志位可能在初始化时被设置为 true 表示睡眠锁为空，不需要阻塞，但如果初始化时被设置为 false，那么它只能被离开睡眠锁的任务设置为 true。为了能够被释放锁的任务唤醒，等待器上还放置了自身的唤醒器，使用唤醒器可以将对于任务加入调度队列。需要注意的是等待器获取锁后就析构了，其他任务再获取它的任何数据

都是极为危险的，而等待器能够析构的标志就是运行标志位，因此所有涉及唤醒任务内存的操作都需要在标志位修改之前进行。因此唤醒器需要在标志位修改之前取出。但任务的唤醒必须在标志为 true 之后，否则由于确实内存序限制，被唤醒任务执行 poll 时可能依然观测到运行标志位为 false 导致唤醒失败，任务丢失，而这又导致此睡眠锁死锁，而睡眠锁是无法用尝试次数的方式来发现死锁的。正确的方式是在标志位修改之前取出唤醒器，之后修改标志位，最后再用唤醒器唤醒对应任务。FTL OS 调度器保证了如果任务被唤醒时处于运行状态，任务结束运行状态时会被再次放入调度器，不会导致任务丢失。

内存序

存粹的原子指令只能保证自身的修改在多核上是原子的，但被两次原子指令包围的起来的访存操作是如何保证在多核可见的呢？这就涉及了内存序。C++20 标准定义了如下内存序：

内存序	RISC-V 实现	描述
relaxed	(NULL)	不对访存作任何约束
acquire	fence r, rw	禁止之后的所有写操作被重排序在此之前
release	fence rw, w	禁止之前的所有写操作被重排序在此之后
consume	fence r, rw	类似 acquire，但只对 release 的数据依赖链生效，不作用于全部访存
acq_rel	fence.tso	同时具有 acquire 和 release 的内存序
seq_cst	fence rw, rw	在满足 acq_rel 内存序的同时使之前的所有写操作在多核间可见

除了 relaxed 内存序外所有内存序都禁止了读操作重排序与 CPU 内部相关数据的乱序执行。需要注意的是单独使用 acquire 或 release 并不能保证数据的安全性，只有 release-acquire 结合使用才能保证数据操作是安全的。

CPU 是如何保证所有写操作在多核间可见的？多核 CPU 采用 MESI 缓存一致性协议，每个缓存块都有无效、独享、修改、共享四种状态。独享状态的数据和内存是一致的，当其他 CPU 读取时会变为共享状态，发生写入时变为修改状态。修改状态缓存的数据与内存是不一致的，当其他核进行读操作时会写回主存

并转变为独享状态。共享状态的数据被多个缓存持有并和主存是一致的，只要任何 CPU 修改了缓存，其他 CPU 对应的缓存都变为无效状态。MESI 保证了任意数据的写入都能够在未来被所有其他的核心观测到，而内存序对应的内存屏障为状态的更新提供了顺序保证。

MESI 协议的操作单位是 cache-line，绝大多数平台上这一大小为 64 字节，这导致了被称为伪共享(false share)的性能下降现象。处于同一个 cache-line 的对象会共享缓存状态，因此当共享对象被其他核修改时整个 cache-line 都会失效，例如一个数组中每个单元都属于各自的 CPU，CPU 在各自单元上的操作理论上的互不影响的，但由于共享了 cache-line 会导致大量的 cache 失效，在高频 CPU 上会导致 5 倍以上的性能损失。更常见的例子是引用计数，由于引用计数是于数据一起放置的，因此每次引用计数的更新都会导致对象 cache 失效。为了避免伪共享，线程局部对象需要对齐至 cache-line。

Hifive Unmatched 使用的 U74 核心只有一个访存端口的顺序处理器，但这不意味着内存序不会出错。包括 U74 在内的多核 CPU 采用了如下的两种方式来提高性能：

- **写缓冲区**（Store Buffer）。

MESI 协议的写操作将等待其他 CPU 的确认信息，在这期间写操作将被阻塞。为了避免阻塞影响到 CPU，所有 store 操作会先提交至写缓冲区，在之后的适当时机才会写入 cache 与内存，而只有写缓冲区满时才会阻塞 CPU。写缓冲区避免了写操作导致的频繁 CPU 暂停，但也导致在其他核心看来这个写操作被延迟了，如果延迟至了某条 load 之后就产生了 store-load 乱序。release 内存序生成的屏障指令可以阻止它的发生，fence r, rw 执行后的任何访存操作会等待写缓冲区清空，保证其他核心的观测顺序。

- **失效队列**（Invalid Queue）。

MESI 协议的读操作需要等待其他 CPU 导致的缓存行被清空后再执行，在这期间读操作将被阻塞。引入失效队列后 CPU 的读操作不再被阻塞，其他 CPU 发送的缓存行无效信息将立刻回复并加入失效队列，在未来的适当时机再刷入 cache。失效队列能有效提高其他 CPU 写操作的速度与当前 CPU 读操作的速度，但也导致了 load-load 与 load-store 乱序。acquire 内存序生成的屏障指令可以阻止

它的发生，fence r, r 之后的访存操作将阻塞等待失效队列处理完成，保证 cache 和内存中的数据是相同的。

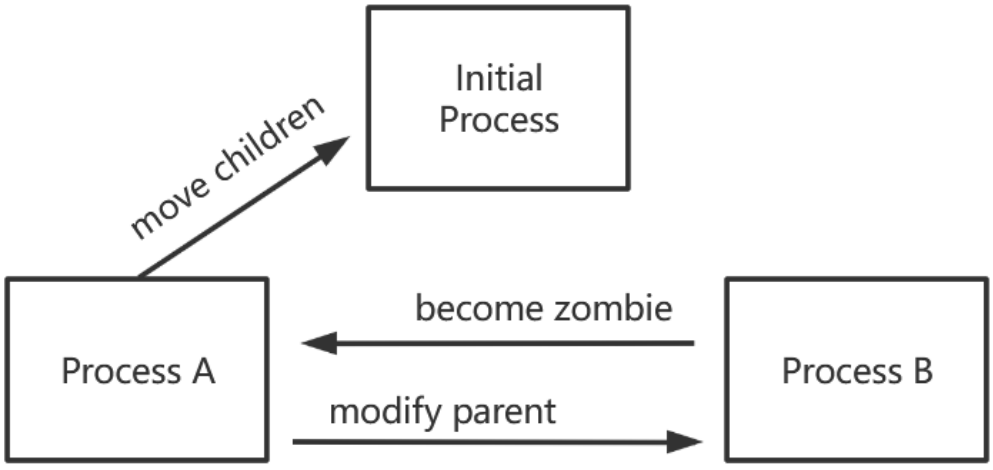
store-store 乱序不会发生在顺序处理器，在乱序处理器中屏障指令可以防止 store 被乱序发射至写缓冲区。原子指令不保证清空写缓冲区和失效队列，不能保证前后的指令内存序正确，因此保护数据必须依赖内存屏障。

2.3.2 进程间通信

事件总线

FTL OS 的进程间通信方案参考了 rCore 的实现。每个进程都持有一个事件总线（`EventBus`），事件总线是其他进程通知本进程的唯一方式并自身有独立的锁保护，向事件总线发送消息不需要获取对应进程的锁。引入事件总线可以有效地解决进程释放时潜在的死锁，对于初始进程 G，某父进程 A 和它的子进程 B，当 A 和 B 同时退出时：

- A 需要通知 B 其父进程修改为 G，并将他放入 G 的子进程集合中。
- B 需要通知 A 自身变为了僵尸进程。



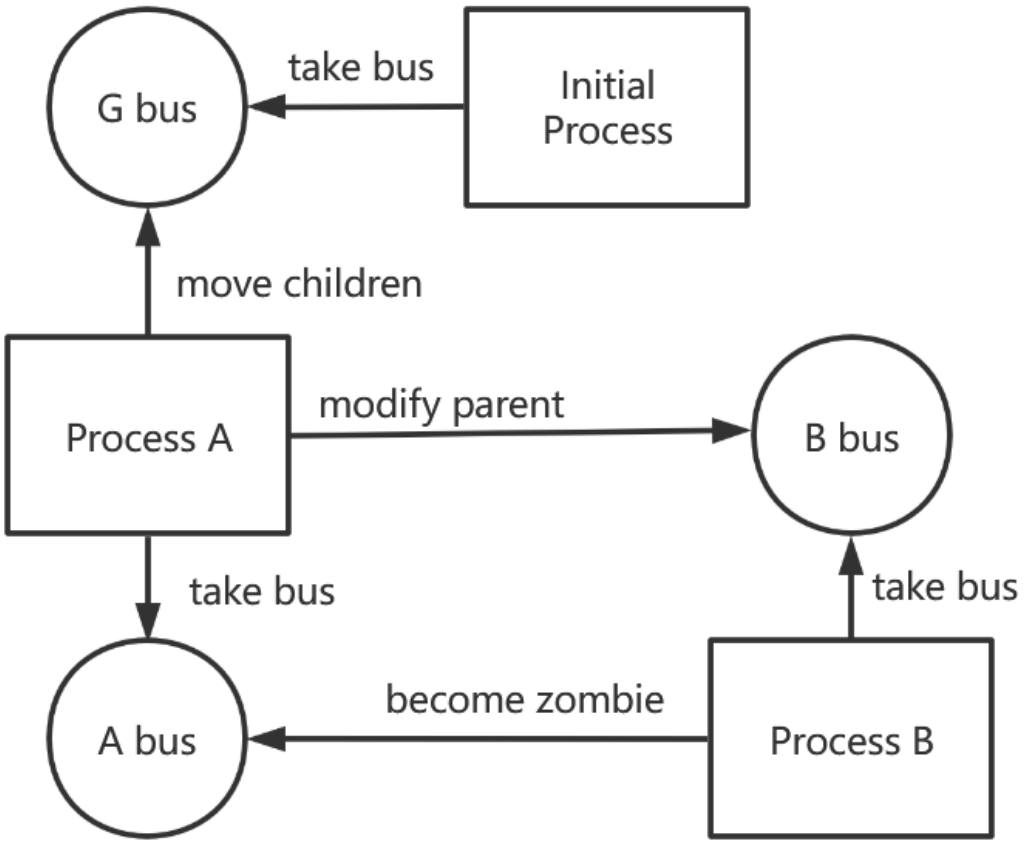
在不使用事件总线时，如果两个事件同时发生则产生了 A 指向 B 和 B 指向 A 的有环图导致死锁，如果想解决这个问题需要加入更加复杂的处理逻辑，而这个问题并不是只在退出进程时会遇到。但使用事件总线则可以一劳永逸地解决死

锁的问题，因为事件总线的锁被封装了，操作事件总线时无法获取另一个锁，因此事件总线永远不会成为环的一部分。

为了防止关闭进程导致的事件丢失，事件总线提供了原子的获取-关闭操作，进程退出之前将关闭事件总线，此时事件总线会返回所有积累的事件，在此之后不再接收任何消息。关闭的事件总线不能再次打开，因此事件总线可以作为僵尸进程的标志。初始化进程 G 被定义为不可退出的，因此永远可以向 G 的事件总线发送消息。因此上述的进程退出竞争将会：

• A 进程向 B 进程发送父进程变更消息，向 G 进程发送 A 变为僵尸的消息与 A 的子进程添加的消息。

• B 向 A 发送自身变为僵尸进程的消息。



使用了事件总线后有环图变为了无环图。由于 A 和 B 处于退出状态，因此事件的发送可能失败，对发送失败的处理为：

• A 向 B 发送父进程变更消息失败，这意味着 B 变为了僵尸进程，改变父进

程毫无意义，放弃发送消息。

• B 向 A 发送自身变为僵尸进程的消息失败，这说明自身变为了 G 的子进程，因此向 G 发送自身变为僵尸的消息。

尽管事件总线能有效避免了死锁，但也让消息的顺序变为了不确定的状态，例如 B 向 G 发送的变为僵尸进程的消息可能先于 A 发送的子进程变更消息到达。FTL OS 引入了“僵化”状态，变为僵尸的进程如果未在子进程集合中找到将处于僵化状态，收到实际的进程后再变为僵尸状态。

引入事件总线后除了数据传输（通过管道）外所有的进程间通信操作都通过事件总线进行。事件总线收到可以处理的事件后将唤醒进程。事件总线的处理单位是以进程为单位，而一个进程可能包含多个线程，因此当前的处理方式是唤醒全部线程，FTL OS 中的异步执行模型保证了误唤醒只会降低增加调度开销而不会导致错误，在未来会追踪每个事件的处理者并只唤醒对于进程。

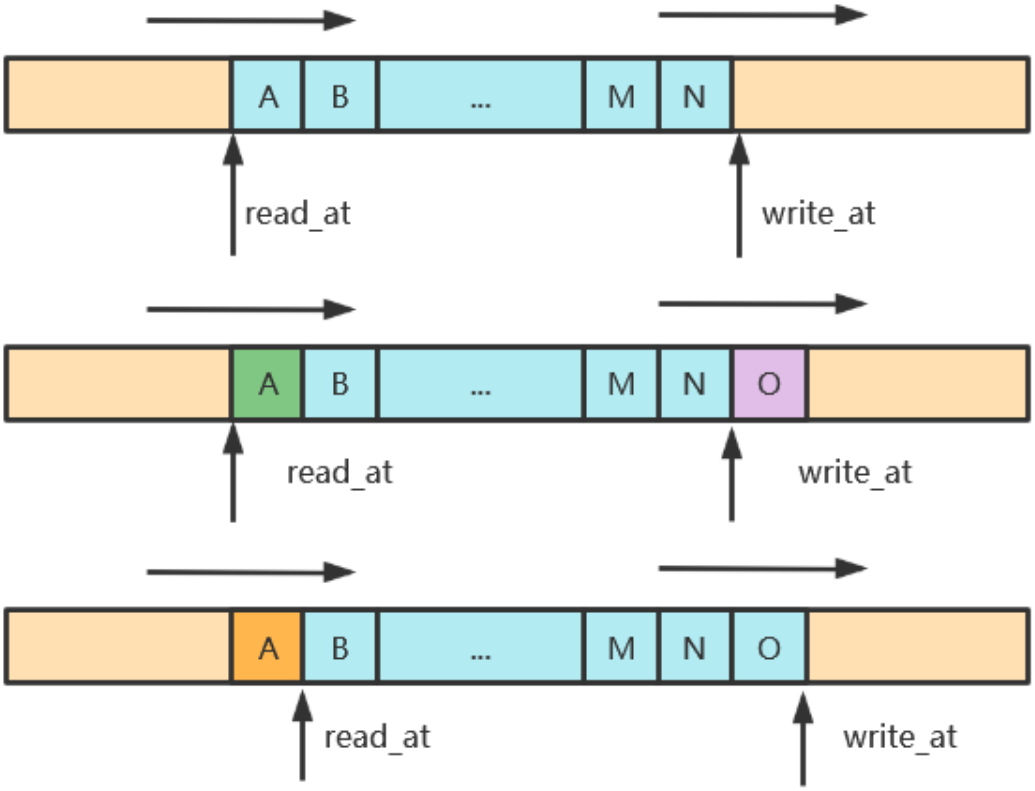
管道

管道是在两个不同进程之间传输数据的常用方式，一个进程向管道写入数据后另一个进程可以从管道读出数据，如果缓冲区空或满都会阻塞相应的操作。常用的管道实现中管道只允许被一个写者或一个读者占用，这种方式非常安全且简单，但 FTL OS 更进一步，允许管道同时有一个读者和一个写者同时存在，由原子操作保证管道的正确运行。

管道缓冲区是 16 KB 的循环 FIFO，由 4 个 4 KB 的内存组成，初始化完成后缓冲区大小不再变化，未来可以通过 RCU 在并发状态下动态地改变缓冲区的占用内存。缓冲区包含两个标志原子变量，分别为 read_at 和 write_at。对于读者来说，从 read_at 到 write_at 的空间是可以读取的，对于写者来说，从 write_at 到 read_at 的空间是可以写入的。读写两侧都会完成数据操作后再更新各自标志位，标志位更新就意味着另一侧可以观测到这部分更新了。由于 FIFO 被分为多片内存，因此数据的最大操作单位为 4 KB，每操作 4 KB 的数据就会更新一次标志位，防止了输入数据过大导致的死锁。

管道运行于异步上下文，因此读写两端都由睡眠锁保护。FTL OS 认为管道

的并行读取和并行写入是没有意义的，因为管道的操作单位是字节而不是数据块，并行的读写只会得到乱序的无效字节，因此没有进一步在缓冲区上应用 wait-free 读写优化。read_at 和 write_at 的更新需要内存屏障。以下展示了管道并行读写时数据和标志的变化：



蓝色的是可以读取的数据，橙色是可以覆盖的无效数据，绿色是已经读取但无法被写入的数据，紫色是已经被写入但无法被读取的数据。

频繁的调度会增加内核开销，FTL OS 将尽可能减少数据传输时的调度次数，即使要传输的数据超过了 16 KB。由于管道缓冲区的读写是由 read_at 和 write_at 保证的，因此只要任意一个发生了变化就意味着另一侧有了数据操作空间。每次数据分片操作完成后管道会重写读入 read_at 和 write_at，如果发生了变化就说明管道获取或被取出了数据，此时将重新开始管道读写。当读取速度和写入速度相同时一次调度就可以完成数据传输，如果速度不匹配则速度较快的一方可以出让 CPU 降低开销，直到被另一端唤醒，而速度较慢的一方将在缓冲区满时出让 CPU。

2.4 文件系统

2.4.1 FAT32 文件系统

FTL OS 的 FAT32 文件系统是异步文件系统，支持完善的多核并发操作，缓存块管理和无阻塞写与异步 IO，严格按照 Unicode 标准处理字符串。

文件系统按模块可划分为：

模块	用途
layout	磁盘数据布局定义
block	缓存块管理模块
fat_list	FAT 簇分配链表管理模块
inode	FAT inode 管理模块
manager	FAT32 文件系统的唯一对外接口

磁盘布局

FAT32 文件系统在磁盘上被划分为如下的多个部分：

MBR(Main Boot Record) 主引导记录，大小为 446 字节。MBR 是计算机启动加载，解释分区结构，包含物理上的 0 扇区。

DBR(DOS Boot Record) DOS 引导记录，DBR 是操作系统访问的第一个扇区，用来解释文件系统。这里的第一个扇区作为文件系统的逻辑 0 扇区。

FAT table。FAT 分区表维护了每一个文件的簇链表。取值如下：

簇号	描述
0	空闲
<0x0FFFFFF7	后继
=0xFFFFFFF7	坏簇
>0x0FFFFFF7	末尾

FAT 表中 0 号位与 1 号位是无效的，0 号位的值永远为 0xF8FFFF0F，1 号位的值为 0xFFFFFFFF/0xFFFFFF0F。FAT32 文件系统可能包含多个 FAT 表，这些 FAT 表互为备份，操作 FAT 表后需要分别写回。

数据段。FAT32 文件系统的数据段放置在 FAT table 之后，并延续到分区的末尾。数据段按簇管理，每个簇包含多个扇区。绝大多数文件系统的簇大小为 4KB，对于 512 字节的扇区，一个簇包含 8 个扇区。不像 ext 文件系统将 inode 和数据块分开管理，FAT32 文件系统中文件和目录的数据都放置在数据段中。

短文件名与长文件名

一个文件名项占用空间为 32 字节，因此一个 4KB 的簇可以放置 128 个项。短文件名格式为 8.3（字节数），文件名部分不可为空，如果扩展名为空则输出的文件名不包含末尾的点，因此短文件名不能表示除了"."与".."之外末尾包含点的文件名。文件名项第一个字节为 0x00 时表示空闲，0xE5 表示文件名项已释放。如果包含 0x40 位且不包含 0xA0 位说明是最后一个长文件名，此时低五位为顺序号。

无扩展的短文件名不能出现小写字母，因此出现小写字母就要使用长文件名表示，对应的短文件名将变为大写字母。windows 支持短文件名中的保留标志位中文件名大小写扩展，全小写文件名可以只用一个短文件名表示。

一个长文件名项可以储存 13 个 utf-16 编码字符。当无法正好填充时，下一个位置将填充 0x0，之后的空闲位置填充 0xffff。

基于 try 的惰性扫描

Haskell 中惰性求值是一项基础语言特性，计算时只会数据结构的必要部分求值，利用惰性求值 Haskell 可以表示出无限链表等在普通语言中难以做到的数据结构。haskell 中 foldr 函数定义如下：

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
```

逻辑上 foldr 是对链表从右向左折叠，但在实际求值中是从左向右计算的。由于 haskell 的惰性求值特性，foldr 中参数 f 可能不需要对 x:xs 求值，此时 foldr 不再需要遍历余下的链表而直接返回。惰性求值让 foldr 可以输入无限列表，并在有限时间内返回结果。

rust 的标准库也提供了惰性求值支持，相对于 Haskell，rust 只能在函数中显式返回，而问号语法为此提供了便利:

```
fn try_fold<B, F, R>(&mut self, init: B, mut f: F) -> R
where
    Self: Sized,
    F: FnMut(B, Self::Item) -> R,
    R: Try<Output = B>,
{
    let mut accum = init;
    while let Some(x) = self.next() {
        accum = f(accum, x)?;
    }
    try { accum }
}
```

这是标准库中 try_fold 的函数定义。在 f(accum, x)?处会对返回值进行判断，如果返回值为 R::Residual，try_fold 将直接返回。它与 haskell 中的 foldr 并不等价，因为折叠的方向不同。

FTL OS 文件系统使用 try_fold 封装对缓存块的操作，并在此基础上继续构造出一系列目录项处理操作。使用 try_fold 而不是迭代器的原因是 try_fold 和迭代器运行的位置不同，try_fold 的参数函数运行在栈的最深处，而迭代器是从函数的返回值中获取值。try_fold 可以作为深层异步函数的一个部分运行，而迭代器除了需要将自身异步化外，每次调用都涉及内部上下文的切换，在文件系统中的表现为 try_fold 只需要获取锁一次，而迭代器每次取出数据都要获取锁，除了开销更大外还导致了潜在的数据不同步问题。

缓存块层

文件系统采用异步驱动，因此缓存块读写也是异步化的。缓存块使用异步睡眠锁保护，当进程阻塞于睡眠锁时不会消耗 CPU 资源。缓存块只能经 inode 层修改，不暴露对外接口。

为了提高性能，文件系统使用无阻塞写策略，除了脏块到达上限外，inode 写操作不会阻塞。为了实现无阻塞写，文件系统构建时将产生一个只运行在内核态的磁盘同步线程。每个磁盘写操作都会将缓存块标记为脏并提交至同步等待集合，磁盘同步线程则取出磁盘同步集合并对每个簇生成一个微线程，这为高性能并行

IO 提供了可能。最后一个微线程执行完成后会唤醒同步线程，同步线程将反馈同步结果至块缓存管理器，所有在同步期间没有被写入的缓存块的状态将变回 clean。

多核环境中无阻塞写必须保证提交至驱动的内存处于不可变状态，即使缓存块在驱动写期间被其他线程修改。为了保证这一点，FTL OS 使用了可以在唯一状态和共享状态自如转变的缓存。定义如下：

```
// fat32/src/block/buffer.rs
pub(crate) enum Buffer {
    Unique(Box<[u8]>),
    Shared(SharedBuffer),
}
#[derive(Clone)]
pub(crate) struct SharedBuffer(pub Arc<Box<[u8]>>);
```

缓存块的大小是可变的，在大多数文件系统中大小为 4KB。Buffer 可以在唯一状态和共享状态之间自由转换，且当引用计数为 1 时转变为唯一状态不需要复制内存，不为 1 时将分配新内存并从共享内存中复制资源。Buffer 保证了唯一状态不存在其他的引用，共享状态的内存不会被修改。缓存块 Cache 使用睡眠读写锁封装 Buffer，保证并发的安全。同步线程只需从可变的 Buffer 获取 SharedBuffer 即可保证数据不变性。

缓存块管理

磁盘的大小可能达到数个 TB，这种情况下缓存整个磁盘是不可能的，而缓存块有限时需要考虑缓存块的替换。无阻塞写策略下缓存块具有 clean 和 dirty 两种状态，而只有 clean 状态的缓存块是可以回收的，因此如果所有缓存块都为 dirty，要么超出最大缓存块限制继续分配，要么阻塞获取操作直到某个 dirty 的缓存块同步完成。这会导致潜在的饥饿，因为如果缓存块在同步完成之前又被写入了，缓存块将依然处于脏状态。FTL OS 使用信号量限制脏块数量保证不会到达缓存块数量上限，每个脏块都持有信号量，只有状态变回 clean 时持有的信号量才会释放。脏块数量限制不能保证可以找到一个可替换的块，因为只有不被持有缓存块才是可替换的。所有缓存块都被持有的概率很低，这种情况下文件系统将操作

失败并返回 ENOBUFS 错误码。

FTL OS 的文件系统使用 RLU 替换策略。缓存块管理器可以原子分配时间戳，缓存块具有一个时间戳字段。每次对缓存块的访问都会更新时间戳字段，但这个字段只会在缓存块替换时被访问。所有缓存块都处于按时间戳排序的有序集合中，但排序使用的排序时间戳仅保证不大于缓存块上的实际时间戳。替换时从时间戳的最小端开始寻找，只有排序时间戳和实际时间戳相等时才进行替换，否则更新排序时间戳并将缓存块放入正确的位置。这种替换方式里每次访问只有一次原子地址递增开销，但缓存块替换时开销较大，被访问的缓存块越多耗时越长。由于时间戳分配器是全局的，分配时间戳的竞争也非常大。

一种有效加速替换且减少竞争的方式是放弃 LRU 替换策略改为时钟替换策略，所有缓存块都放置在一个循环链表构成的时钟轮盘上，每个缓存块维护一个标志位。每次缓存块的访问都会将对应标志位设为 1，而时钟经过缓存块时判断标志位，如果为 0 则直接替换，如果为 1 则改为 0。这种方式在访问缓存块时只有一次额外访存开销，缺点是相对 RLU 会增加缓存块的查找和替换次数。当缓存块数量较多时替换次数将非常少，但减少的一次原子指令能明显提高性能。另一种方式是使用多代 LRU 算法，将所有缓存块放置在多条链表上，每次缓存块访问都会提升缓存块所属级别，但提升至最高级时不再提升，此时不再发生锁竞争。缓存块查找时只会在最低级的链表查找，当链表为空时所有链表都降低一个级别，此方法经过优化后可以在降低级别时不需要修改每一个缓存块上的信息，速度很快。

缓存块索引器

每个缓存块都对应一个磁盘上的簇，因此簇号是缓存块的唯一标识。获取缓存块自然可以在缓存块管理器中获取，但缓存块管理器被睡眠锁保护，竞争和锁开销非常大。为了提高性能，FTL OS 在缓存块管理器的睡眠锁之外加入了缓存块索引器，查找缓存块时优先从索引器查找，当索引器查找失败时再进入缓存块管理器中查找缓存块。索引器定义如下：

```
// fat32/src/block/index.rs
pub(crate) struct CacheIndex(RwSpinMutex<BTreeMap<CID, Weak<Cache>>>);
```

可见索引器就是一个由读写自旋锁保护的映射。由于索引器只是不持有所有权的加速结构，因此映射结果也不持有所有权的弱指针。只有成功通过 upgrade 获取缓存块的强指针才是一次成功的索引器查找，索引器发生未命中则会在缓存块管理器进行缓存块查找，查找结束后将更新索引器，保证未命中至多在每个 CPU 发生一次。索引器在释放管理器睡眠锁后更新来减少缓存块管理器的持续占用，这可能存在由乱序更新导致的轻微内存泄露（长时间存在一个失效缓存块的索引），但不影响程序的安全性。

高并发的 FAT 表

许多文件系统的实现将 FAT 表看作一个对象并使用读写锁保护，例如 2021 届操作系统比赛冠军 Ultra OS 就使用了读写锁来保护 FAT 表。这样实现无可厚非，但 FTL OS 对 FAT 表有更进一步的高并发设计。FAT 表在磁盘上占有大量的簇，将如此大的空间看作一个对象将极大地降低并发度，一个例子是两个互不相关的线程正在分别独立地读与写文件 A 与 B，写文件 B 获取的 FAT 表写锁会阻塞读文件 A，但这是两个不同的文件，拥有的 FAT 链表也是不相同的，为什么要相互干扰降低性能呢？FTL OS 仅仅将 FAT 表看作一块磁盘上的空间，空间上分布了许多各自属于对应的文件的 FAT 链表，对一条链表的操作不会阻塞其他链表的操作。此设计下某文件分配簇的过程完全不会阻塞到其他文件对所属簇的读取，只要它拥有的链表存在于缓存中。

FTL OS 的文件系统支持 TB 级磁盘，不会将整个 FAT 表放入内存。与缓存块不同，FAT 表缓存以扇区为单位而不是以簇为单位，替换策略与缓存块一样使用 LRU 替换策略。FAT 链表分配涉及大量数据修改不可并发，因此 FAT 表的磁盘 IO 和簇分配都在 FAT 表管理器中进行并由读写锁保护。

和缓存块管理器相同，FAT 表采用了无阻塞写策略，管理器初始化时将生成一个只在内核态运行的同步线程。

FAT 表索引器

FAT 表索引器除了提供加速查找作用外，更重要的是它不处于管理器睡眠锁的作用范围，管理器处于占有状态时依然可以查找。由于 FAT 表的使用频率较大，索引器没有使用 BTreeMap，而是使用了数组来实现更快的查找。索引器定义如下：

```rust
// fat32/src/fat_list/index.rs
pub(crate) struct ListIndex {
    weak: Box<[UnsafeCell<Weak<ListUnit>>]>,
    lock: Box<[RwSpinMutex<()>]>,
}

pub(crate) struct ListUnit {
    buffer: UnsafeCell<Buffer>,
    aid: UnsafeCell<AID>,
}
```

索引器并不是对单个项的索引，而是对一个 FAT 表扇区 ListUnit 的索引。在文件系统中 FAT 链表的修改只能在 FAT 表管理器中进行，所有修改都是串行化的，因此这里系统使用了 UnsafeCell 来提高性能。和缓存块索引器一样，索引器的弱指针可能是无效的，此时需要进入 FAT 表管理器再次获取索引。索引器使用读写自旋锁的原因是多核环境下操作智能指针是不安全的，文件系统允许多个线程同时操作 FAT 表的同一个缓存扇区，由于 CPU Cache 的存在，写线程释放了旧的 Weak 指针后可能无法及时通知到读线程的 Cache，此刻读线程将获取到一个已经被析构的弱指针，进行任何操作都会导致内核出错。最简单的解决方法就是上锁：

```rust
// fat32/src/fat_list/index.rs
pub fn get(&self, index: usize) -> Option<Arc<ListUnit>> {
    let _lock = self.lock[index].shared_lock();
    unsafe { (*self.weak[index].get()).upgrade() }
}
pub fn set(&self, index: usize, arc: &Arc<ListUnit>) {
    let _lock = self.lock[index].unique_lock();
    unsafe { *self.weak[index].get() = Arc::downgrade(arc) }
}
```

另一个方式是使用 RCU 释放 Weak 指针，这个方式的读者不需要上锁：

```rust
// fat32/src/fat_list/index.rs
use ftl_util::rcu::RcuCollect;

pub fn get(&self, index: usize) -> Option<Arc<ListUnit>> {
    unsafe { &(*self.weak[index].get()) }.rcu_read().upgrade()
}
pub fn set(&self, index: usize, arc: &Arc<ListUnit>) {
    let _lock = self.lock[index].unique_lock();
    unsafe { &mut (*self.weak[index].get()) }.rcu_write(Arc::downgrade(arc))
}
```

rcu 系统不会立刻析构旧的弱指针，而是等待所有 CPU 离开宽限期才释放弱指针，因此 get 函数实现中的 rcu_read 总是有效的。rcu_read 仅需要一次访存和一次读内存屏障，相对于读写锁的两次原子操作能明显提升性能。

索引器的数组长度与 FAT 表的扇区数量相同，每个 FAT 表扇区需要 8 B 的指针。对于扇区大小为 512 B，簇大小为 4 KB 的 1 TB 磁盘，FAT 表的大小为 1 GB，索引器的空间开销为 16 MB，因此文件系统可以轻易支持 TB 级别的磁盘。

文件 inode

FTL OS 中 inode 是用户操作磁盘数据的唯一方式，一个 inode 对应一个文件，也拥有对应 FAT 链表表的所有权。由于 FAT32 文件系统特性，文件系统中包含两种 inode：目录 inode 和文件 inode。为了提高性能，inode 被分为了两个部分，inode 缓存和 inode 句柄。每个 inode 句柄都对应至一个 inode 缓存，只有虚拟文件系统的 inode 持有 inode 句柄的所有权，因此虚拟文件系统的 inode 引用计数归零析构时也将析构 inode 句柄。但 inode 句柄的析构并不意味着 inode 缓存立刻析构，inode 缓存使用和缓存块相同了 LRU 替换策略，当缓存数量达到上限时将替换已有的 inode 缓存块。inode 缓存不存在需要写回磁盘的数据，如果 inode 缓存的引用只被管理器持有就可以直接替换。

inode 缓存数据	描述与作用
inode 弱指针索引	不经过缓存管理器直接获取缓存对应的 inode
文件名项地址	磁盘上此文件的短文件名位置
FAT 链表缓存	此文件所属文件的 FAT 链表缓存，懒加载
FAT 链表首簇号	此文件 FAT 链表的第一个簇号

FAT 链表的末簇号	此文件 FAT 链表的最后一个簇号，懒加载
文件簇数	簇链表的长度
短文件名项	此文件短文件名项的副本

inode 缓存使用读写自旋锁保护，每次 inode 的 FAT 链表查找都会更新 inode 缓存。inode 自身由读写睡眠锁保护，当涉及文件大小修改或簇的分配与释放时必须获取写睡眠锁。

FTL OS 的文件系统对 inode 的操作进行了更高粒度的划分：

inode 操作	获取的锁
读文件	inode 共享锁
写文件（不改变文件长度）	inode 共享锁
写文件（改变文件长度）	inode 排他锁

inode 需要一个标识来作为索引，防止多个 inode 实例同时存在。FTL OS 使用了短文件名项的磁盘位置来作为标识，根目录的位置设为 0。不使用 inode 首簇号作为标识的原因是 FAT32 文件系统的空文件不会分配簇。不使用索引的方式也是存在的，即由操作系统的虚拟文件系统层来保证 inode 的唯一性，但 FTL OS 未使用此方案。

目录操作

目录的操作包括插入、删除、查找三种操作。增加目录项和删除目录项都需获取 inode 排他锁，查找只需要获取 inode 共享锁。

查找项时输入为 utf-8 标准字符串，由于短文件名扩展等情况的存在同一字符串在 FAT32 文件系统可以有多种表示方式，不能提前判断输入字符串是否为短文件名，只能先根据文件名项生成文件名，再判断当前文件名与目标文件名是否匹配。文件名还需要判断是否包含保留字符，例如'<'、'>'等。

插入新的项需要先判断是否已经有存在的同名文件，并预先计算需要多少项文件名。计算完成后使用 try 方式遍历簇链寻找连续可用空间，如果找不到则在末尾分配新的簇。结束过程后在空闲位置写入即可。

删除项则需要找到文件名的位置，将每个项的首字母标记为 0xE5 即可。删除项分为删除目录和删除文件两种情况，锁定目录后将删除缓存管理器中的缓存，

此后删除文件时将释放整个数据链，如果 inode 缓存存在其他的引用说明文件没有关闭，删除操作将失败。删除目录时只允许删除空目录，由上层操作负责递归删除整个目录。目录的排他锁保证了在删除过程中其他进程无法向删除目录中插入数据。

2.4.2 SD 驱动

Hifive Unmatched 平台缺少必须的 SD 卡驱动，因此我们参考了 K210 的 SD 卡驱动实现了基于 SPI 协议的可以在 Hifive Unmatched 平台的 SD 驱动。

SPI 协议

FTL OS 将 SPI 协议的实现划分为了 3 个模块：

文件	实现
layout.rs	内存映像抽象及通信
mod.rs	SPIActions 接口定义
registers.rs	spi 控制寄存器抽象

Hifive Unmatched 的 SPI 协议的相关寄存器通过内存映射的方式访问。具体的映射如下：

控制寄存器	相关字段及功能
SCKDIV	控制串行时钟的频率
SCKMODE	控制数据采样和切换数据和时钟上升下降沿的关系
CSID	片选寄存器,实现SD卡的选择,这里由于只有一块SD卡,只实现了寄存器的reset
CSDEF	设定片选线
CSMODE	设置片选模式 1.AUTO：使CS生效或者失效在帧的开始或结束阶段 2.HOLD: 保持CS在初始帧之后一直有效 3.OFF：使得硬件失去对CSpin的掌控
DELAY0	cssck字段：控制CS有效和SCK第一次上升沿之间的时延 sckcs字段：控制SCK最后的下降沿和CS失效之间的时延
DELAY1	cssck字段：intercs字段：控制最小CS失效时间 interxfr字段：控制两个连续帧在不丢弃CS的情况下之间的延迟，只在sckmod寄存器是HOLD或者OFF模式时使用
FMT	设置协议，大小端和方向等，传输数据的长度
TXDATA	data字段：存储了要传输的一个字节数据，这个数据是被写入FIFO的，注意大小端 full字段：表示FIFO是否已满，如果已经满了，则忽略写到tx_data的数据这些数据自然也就无法FIFO
RXDATA	data字段：存储了要传输的一个字节数据，这个数据是被写入FIFO的，注意大小端 full字段：表示FIFO是否已满，如果已经满了，则忽略写到tx_data的数据这些数据自然也就无法FIFO
TXMARK	决定传输的FIFO的中断在低于多少阈值下进行触发 txmark字段:当FIFO中的数据少于设置阈值时会触发中断，导致txdata向FIFO中写入数据
RXMARK	决定接收的FIFO的中断在高于多少阈值时触发 rxmark字段：当FIFO中的数据超出阈值时会从FIFO中读取数据
FCTRL	控制memory-mapped和programmed-I/O两种模式的切换
FFMT	定义指令的一些格式例如指令协议，地址长度等等
IE	txwm字段：当FIFO中的数据少于txmark中设定的阈值时，txwm被设置 rxwm字段：当FIFO中的数据多余rxmark中设定的阈值时，rxwm被设置
IP	txwm悬挂字段:当FIFO中有充足的数据被写入并且超过了txmark时，txwm的悬挂位被清除 rxwm悬挂字段:当FIFO中有充足的数据被读出并且少于rxmark时，rxwm的悬挂位被清除

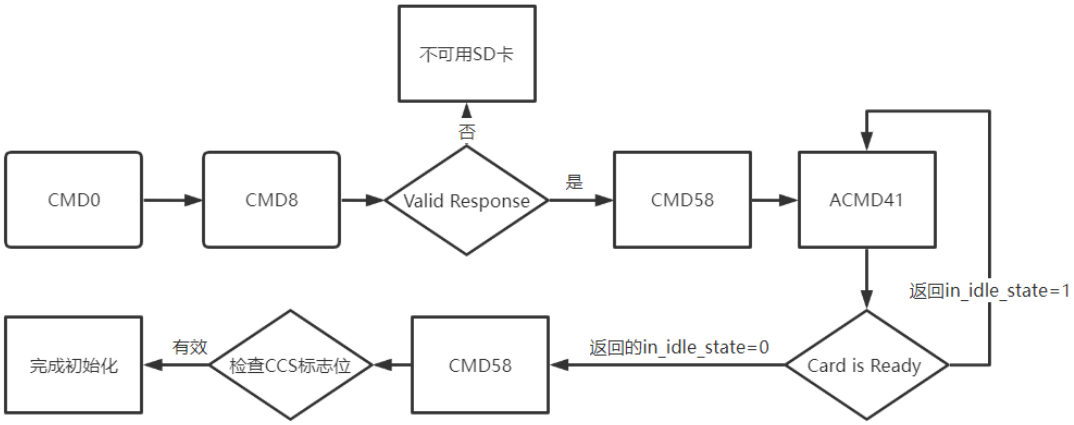
其中 TXMARK 和 RXMARK 寄存器以及 IE 和 IP 寄存器对后面的控制数据传输起着重要的作用，具体表现如下：

1. TXMARK 设置了写中断的界限，设置了 TXMARK 后，若 FIFO 中的数据少于设定的值，就会将 TXDATA 中的数据写入 FIFO 直到满足其中的数据量大于 TXMARK 值，这个中断是由 IE 来判断的，所以我们如果在中断以后，可以依据 IE 中的相关标志位设置循环，然后直到写入 FIFO 数据超过相关阈值。

2. RXMARK 设置了读中断的界限，设置了 RXMARK 后，若 FIFO 中的数据多于设定的值，就会将 FIFO 中的数据读入 RXDATA 中直到 FIFO 中的数据量少于 RXMARK 值，这个中断也是由 IE 来判断，所以在中断以后，就根据标志位设置循环以不停读出 FIFO 中的数据，直到 FIFO 中的数据低于相关阈值。

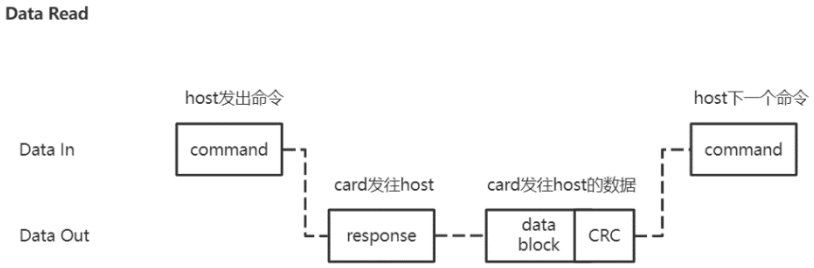
SD 卡驱动实现

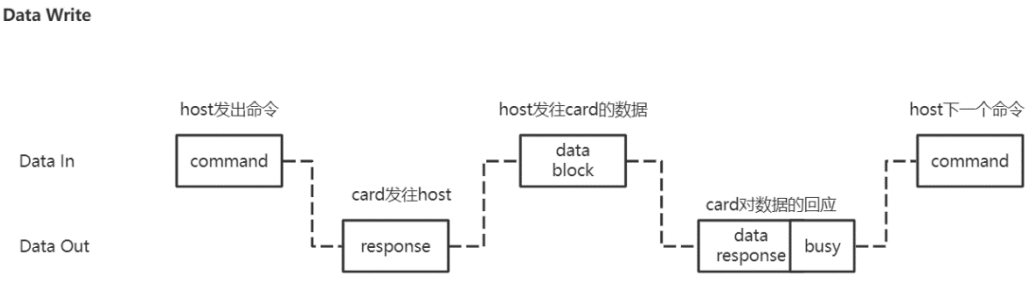
SD 卡进行 SPI 通信的过程实际上是 host 发送指令，SD 卡做出回应这样的方式。SD 卡的初始化过程如下：



这是 SD 卡进入 SPI 模式的一个过程，可以看到，是主机发送 CMD 指令和 SD 卡进行交互，在“一切就绪”（包括电压范围等）后，SD 卡就进入了 SPI 通信的模式。

读写数据块





• 读一个块的过程如下：

1. host 发出读一个块的指令

2. SD 卡 response

3. SD 卡返回一个块加上校验码

4. 结束

• 写一个块的过程如下：

1. host 发出写一个块的指令

2. SD 卡 response

3. host 发出一个带有开始数据标志的数据块

4. SD 卡接收到数据块后发出带有数据回应和 busy 的信息

5. 结束

在读写多个块的情况下大同小异，不同的只有指令号不一样，数据块的数量上以及个别标志位有所差异，具体请参考文档 Technical Commitee SD Card Association 发布的 SD Specifications 的第 7 章。

CRC 纠错

在实际测试中，目前的驱动以 8MHz 读取数据时，约 200 个扇区会发生一次错误，因此 CRC 校验是必须的。扇区数据的校验使用 CRC16 校验方法，FTL OS 采用了和 Linux 相同的校验函数实现，将每个字节对于的 256 种 CRC16 校验状态转换硬编码在代码中，以查表的方式校验数据。一旦发生校验失败就重新传输数据。

3 初赛总结

3.1 初赛评测

FTL OS 已在初赛测试获得满分：

内核实现赛-初赛-K210 | 内核实现赛-初赛-Unmatched | 内核实现赛-决赛阶段1 | 内核实现赛-决赛阶段2

比赛提交到排行榜更新有20秒左右的延迟

#	用户名	队伍	rank
1	19373469	图漏图森破/ 北京航空航天大学	102.0000
2	DarkAngelEX	FTL OS/ 哈尔滨工业大学（深圳）	102.0000
3	dh2zz	健康向上好青年/ 哈尔滨工业大学(深圳)	102.0000

3.2 当前实现

- 无栈异步架构，完善的多核并发设计与睡眠唤醒机制。
- 所有初赛要求的系统调用。
- 完全解耦的进程映射页管理器，一步到位地管理共享映射。
- 完全符合 FAT32 标准的高性能并行异步文件系统。
- 带 CRC 纠错的 SPI 协议 SD 卡驱动。
- 异步上下文运行的栈回退调试系统。

3.3 未来展望

- 完善的信号系统支持
- 支持更多的系统调用，能够正常运行大部分应用程序。
- 网络支持