操作系统内核设计

MinotaurOS

哈尔滨工业大学

2024 年 8 月

目录

第 1 章　内核整体设计

MinotaurOS 整体结构为宏内核，如图 1-1 所示。在监督模式层面，硬件抽象层（HAL）负责与硬件交互，包括中断处理、CPU 核心管理、SBI 调用和设备驱动等。往下细分为 4 个模块，分别为块设备管理模块、字符设备管理模块、网络设备管理模块和虚拟内存管理模块，其中块设备管理器和字符设备管理器组成文件系统。MinotaurOS 将可执行块分为两大基本类，即用户线程和内核线程，两者在协程执行器的管理下统一调度。在用户模式层面，系统调用 Trap 回到内核态后，仍然在协程执行器的控制下。这样的设计使得 MinotaurOS 能够充分利用多核特性，实现内核异步模型。

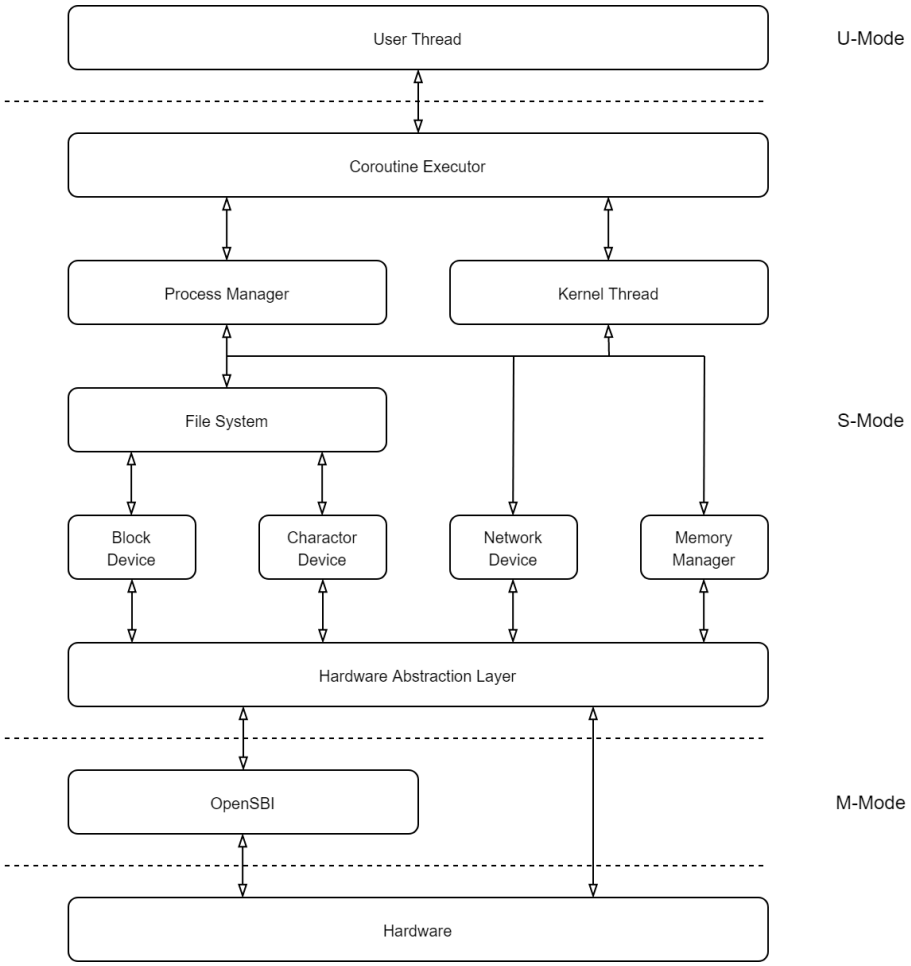

图 1-1　MinotaurOS 系统架构

在硬件操作上，MinotaurOS 一部分使用 RISC-V 的标准 SBI 接口与运行在机器模式下的 OpenSBI 交互；一部分通过设备树信息直接操作 MMIO。MinotaurOS

通过 SBI 接口管理 CPU 核心的启动和暂停；通过设备树直接管理内存、磁盘、时钟和串口等硬件资源。

MinotaurOS 目前可以运行在 QEMU 虚拟机和 Vision Five 2 硬件平台，通过解析设备树实现一套二进制兼容不同硬件。MinotaurOS 支持了 108 个 Linux 系统调用，并且能够通过一系列测试用例，如图 1-2。

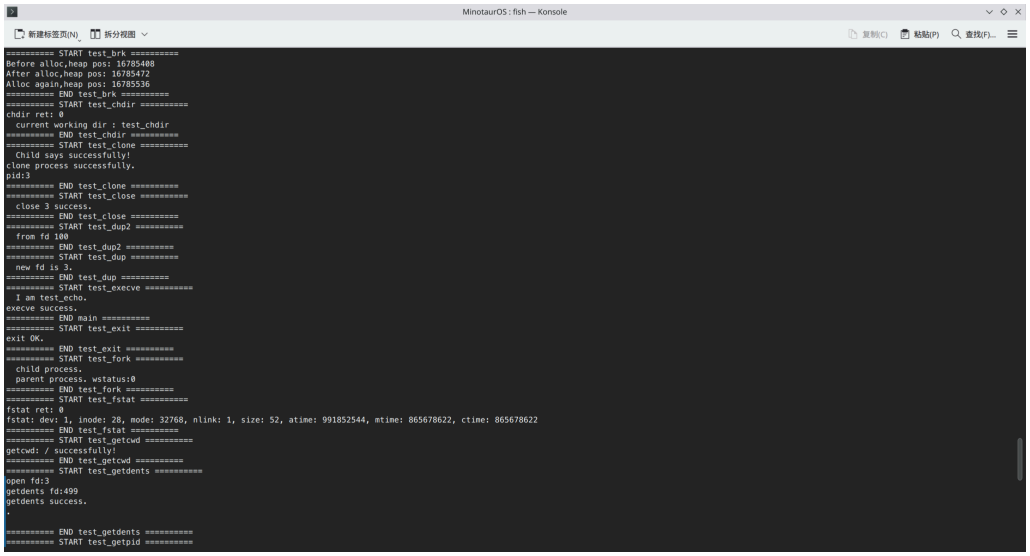

图 1-2　运行 2024 oscomp 初赛测试用例

与此同时，MinotaurOS 还能够支持 iozone、lmbench 等 Linux 程序的运行。截至一阶段结束，MinotaurOS 取得了第三名的成绩，并且具有优秀的性能，如图 1-3 所示。

比赛提交到排行榜更新有20秒左右的延迟

#	用户名	队伍	提交次数 (ASC)	最后提交时间 (ASC)	busybox	cyclictest	iozone	iperf	libcbench	libctest	lmbench	ltp	lua	netperf	unixbench	rank
1	T202410336992584	Pantheon/ 杭州电子科技大学	49	2024-07-31 22:43:41	54.0	0.0	24.421824347289	6.032994923857868	27.48622851500211	220.0	39.819711550482914	394.0	9.0	6.6032925743208875	29.14192591020019	810.5060
2	T202418123993075	Phoenix/ 哈尔滨工业大学（深圳）	65	2024-07-31 21:38:08	54.0	4.0	28.684616768668953	6.0	27.052516328824716	220.0	44.00349712824376	235.0	9.0	5.0	28.22169489475709	660.9623
3	T202410213992712	练习时长两年半/ 哈尔滨工业大学	35	2024-07-31 23:01:35	54.0	0.0	35.70246671175238	1.0	27.201884064011246	219.0	45.71043774462256	91.0	9.0	5.7254334365048685	27.94544901303355	516.2857
4	T202410055992606	萌新/ 南开大学	55	2024-07-31 14:29:36	54.0	0.0	0.0	0.0	27.37918170895098	220.0	21.80486066429678	81.0	9.0	0.0	25.45228741964901	438.6363
5	T202410487992457	RustTrustHuster/ 华中科技大学	46	2024-07-31 14:06:03	54.0	0.0	30.758105091493917	0.0	28.371556342692095	219.0	45.24075590092338	0.0	9.0	0.0	7.236434526413594	393.6069

图 1-3　决赛一阶段成绩

第 2 章　内存管理模块实现

2.1　内存布局

MinotaurOS 使用单一页表结构，内存布局如图 2-1 所示。在本系统中，内核例程与用户例程处于同一页表下：内核区域位于虚拟地址高位，用户区域位于虚拟地址低位。为了方便在内核进行地址转换，从内核镜像开始到可用内存的最大地址的物理内存都按照固定偏移映射到虚拟内存。因此在一个页表结构下，可能存在两个页表项指向同一个物理页帧。这样的设计最大的方便之处在于通过物理页号访问内存时，可以无需经过页表翻译，直接通过偏移得到虚拟地址，利于构造页目录帧和写时复制。

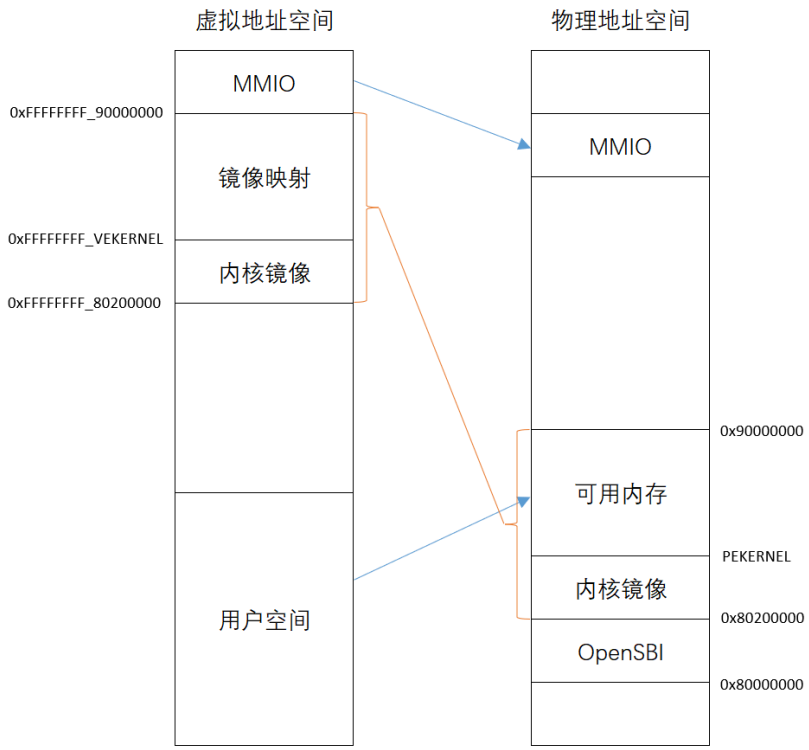

图 2-1　MinotaurOS 内存布局

2.2　堆分配器和用户页帧分配器

MinotaurOS 存在两个内存分配器，分别是堆分配器和用户页帧分配器。堆分配器用于分配内核堆，即内存布局中位于内核镜像区域的内存，其分配的页帧始终位于所有页表的全局映射区域。用户页帧分配器分配内存布局中位于可用内存区域

的内存，其分配的页帧也存在于页表的全局映射区域（可以通过偏移直接访问），但是可能被同时映射到用户空间的虚拟地址。

堆分配器和用户页帧分配器使用相同的数据结构保存（如代码 2-1 所示），两个分配器均使用 Buddy System 算法实现。分配器使用 RAII 方式实现自动回收，即使用帧跟踪器的数据结构跟踪分配的内存。当帧跟踪器被析构时，自动执行回收操作。

```rust
pub struct HeapFrameTracker /* UserFrameTracker */ {
    pub ppn: PhysPageNum,
    pub pages: usize,
}
```

代码 2-1　分配器结构

2.3　虚拟内存数据结构

一个虚拟地址空间由一个 `AddressSpace` 对象描述（如代码 2-2 所示）。`AddressSpace` 对象包含了唯一的地址空间标识、区域映射表和页表帧数组等。其中，地址空间标识用于 TLB 优化，在性能优化一章中会详细介绍；区域映射表是 MinotaurOS 虚拟内存系统的核心数据结构，描述了虚拟地址空间的区域。不同的区域可能有不同的映射方式，例如线性映射、共享内存、写时复制等。页表帧数组用于存储页表帧的跟踪器。

```rust
pub struct AddressSpace {
  /// 地址空间标识
  pub token: usize,
  /// 根页表
  pub root_pt: PageTable,
  /// 地址空间中的区域
  regions: BTreeMap<VirtPageNum, Box<dyn ASRegion>>,
  /// 该地址空间关联的页表帧
  pt_dirs: Vec<HeapFrameTracker>,
  /// System V 共享内存
  sysv_shm: Arc<Mutex<SysVShm>>,
  /// 堆范围
  heap: Range<VirtPageNum>,
  /// 堆最大位置
  heap_max: VirtPageNum,
}
```

代码 2-2　AddressSpace 对象

2.4　虚拟地址空间区域实现

MinotaurOS 的虚拟地址空间区域通过 `ASRegion` 接口定义（如代码 2-3 所示）。`ASRegion` 定义了虚拟地址空间区域的基本操作，包括获取元数据、映射、解映射、复制和错误处理程序。目前为止，MinotaurOS 实现了 `DirectRegion`、`LazyRegion`、`FileRegion`、`SharedRegion` 四种区域类型。其中，`DirectRegion` 用于内核空间的全局映射；`LazyRegion` 用于用户空间的页，实现了写时复制；`FileRegion` 用于文件映射；`SharedRegion` 用于共享内存。

地址空间区域元数据 `ASRegionMeta` 存储了区域的基本信息，包括区域的名称、起始地址、大小和权限。每个 `ASRegion` 对象都需要实现 `metadata` 方法，返回一个 `ASRegionMeta` 对象的引用。为了与整体地址空间解耦，`ASRegion` 及其元数据不存储页表帧和根页表，而是要求映射和解映射等操作所需的根页表在调用时传入，并将映射过程产生的页表帧委托给上层的 `AddressSpace` 管理。因此，虽然一个地址空间区域必然包含在一个地址空间中，但是地址空间区域并不与特定的地址空间绑定。这方便了地址空间区域的复制和移动。

```rust
pub trait ASRegion: Send + Sync {
    fn metadata(&self) -> &ASRegionMeta;
    fn metadata_mut(&mut self) -> &mut ASRegionMeta;
    /// 将区域映射到页表，返回创建的页表帧
    fn map(&self, root_pt: PageTable, overwrite: bool)
        -> Vec<HeapFrameTracker>;
    /// 将区域取消映射到页表
    fn unmap(&self, root_pt: PageTable);
    /// 分割区域
    fn split(&mut self, start: usize, size: usize)
        -> Vec<Box<dyn ASRegion>>;
    /// 扩展区域
    fn extend(&mut self, size: usize);
    /// 拷贝区域
    fn fork(&mut self, parent_pt: PageTable) -> Box<dyn ASRegion>;
    /// 同步区域
    fn sync(&self) {}
    /// 错误处理
    fn fault_handler(&mut self, root_pt: PageTable, vpn: VirtPageNum)
    -> SyscallResult<Vec<HeapFrameTracker>> {
        Err(Errno::EINVAL)
    }
}
```

代码 2-3　ASRegion 接口

2.4.1　直接映射区域

`DirectRegion` 采用线性映射，仅记录对应页的物理页号和权限。在映射到页表时，通过将物理页号加上固定的偏移，直接得到对应的虚拟页号。`DirectRegion` 的

页表权限始终与区域权限保持一致。复制区域时，直接复制记录的字面值即可。同时，无需额外的错误处理程序。

2.4.2 懒惰映射区域

`LazyRegion` 为区域内的每一个虚拟页存储了一个 `PageState` 对象（如代码 2-4 所示）。一个虚拟页可能存在三种状态：未分配、已映射和写时复制。若虚拟页处于未分配状态，则对应的 `PageState` 无需存储额外信息；若虚拟页处于已映射状态，则其独占持有一个用户页帧跟踪器；若虚拟页处于写时复制状态，则与其他区域（可能存在于不同的地址空间）共享持有一个用户页帧跟踪器。

```
enum PageState {
    /// 页面为空，未分配物理页帧
    Free,
    /// 页面已映射
    Framed(UserFrameTracker),
    /// 写时复制
    CopyOnWrite(Arc<UserFrameTracker>),
}
```

代码 2-4　LazyRegion PageState 对象

将 `LazyRegion` 映射到页表时，根据每一个虚拟页的状态不同，映射到页表上的对应页表项的权限也不同。即页表项权限与区域的权限并不一定保持一致。页表项映射规则如算法 2-1 所示：

（1）若虚拟页处于未分配状态，则对应页表项的权限为空；

（2）若虚拟页处于已映射状态，则对应页表项的权限与区域权限保持一致；

（3）若虚拟页处于写时复制状态，则对应页表项的权限为只读。

算法 2-1: LazyRegion 页表映射

```
   input: page
1  if page.state = Free then
2      page.pte ← empty
3  else if page.state = Framed then
4      page.pte ← region.perm
5  else page.state = CopyOnWrite then
6      page.pte ← region.perm - Write
7  end
```

当虚拟页处于写时复制状态时，若发生写操作，会触发 Page Fault。无论 Page Fault 发生在内核态还是用户态，MinotaurOS 都会在 Trap 中调用对应区域的错误处

理程序，如算法 2-2 所示。`LazyRegion` 的错误处理程序会将写时复制页的状态转换为已映射，并分配一个新的用户页帧，将原有的用户页内容复制到新的用户页上，再将页表项权限调整为与区域一致。这样，写时复制的区域就变成了独占持有一个用户页帧跟踪器，不再与其他区域共享。

算法 2-2: LazyRegion 错误处理

```
    input: page
1   if page.state = Free then
2       page.frame ← alloc_frame()
3       page.state ← Framed
4   else if page.state = CopyOnWrite then
5       new_frame ← alloc_frame()
6       copy_frame(new_frame, page.frame)
7       page.frame ← new_frame
8       page.state ← Framed
9   end
10  remap(page)
```

当对整个区域进行复制时，`AddressSpace` 会调用 `fork` 方法，如算法 2-3 所示。`LazyRegion` 会将当前区域已映射的虚拟页和复制区域对应的页状态都设置为写时复制，指向同一个物理页，再将页表项权限调整为只读。这样，原区域与复制出来的区域共享持有一个用户页帧跟踪器，直到下次发生写操作。

算法 2-3: LazyRegion 复制

```
    input: region
    output: new_region
1   for each page in region do
2       if page.state = Free then
3           new_page.frame ← empty
4           new_page.state ← Free
5       else if page.state = Framed then
6           new_page.frame ← page.frame
7           new_page.state ← CopyOnWrite
8           page.state ← CopyOnWrite
9       else page.state = CopyOnWrite then
10          new_page.frame ← page.frame
11          new_page.state ← CopyOnWrite
12          new_region.push(new_page)
13      end
```

14 **remap**(page)
15 **end**

通过上述设计，MinotaurOS 能够在创建进程时实现写时复制，减少复制开销；同时允许了程序申请巨大的内存空间，而不会立即分配物理页帧，提高了内存分配的效率。

2.4.3 共享内存区域

MinotaurOS 支持两种类型的共享内存：System V 共享内存和匿名共享内存。System V 共享内存由 `shmat` 系列系统调用产生，匿名共享内存由 `mmap` 系统调用产生。System V 共享内存通过 `SysVShm` 对象管理，`SysVShm` 对象包含了共享内存的 id 和对应分配的页帧，如代码 2-5 所示。

```
pub struct SysVShm {
    ids: IdAllocator,
    shm: BTreeMap<usize, Vec<Arc<UserFrameTracker>>>,
}
```

代码 2-5 SysVShm 对象

System V 共享内存与匿名共享内存统一使用 `SharedRegion` 数据结构实现。`SharedRegion` 的虚拟页状态有三种，分别是未分配、已映射和引用映射。与 `LazyRegion` 自身持有用户页帧跟踪器不同，`SharedRegion` 的引用映射状态下，地址空间区域对象引用持有在 `SysVShm` 中的用户页帧跟踪器。在帧管理上，System V 共享内存总是在调用时即时分配物理页帧，而匿名共享内存则是在第一次读/写操作时分配物理页帧。

```
enum PageState {
    /// 页面为空，未分配物理页帧
    Free,
    /// 页面已映射
    Framed(UserFrameTracker),
    /// 通过 `shmem` 系统调用映射的页面
    Reffed(Weak<UserFrameTracker>),
}
```

代码 2-6 SharedRegion PageState 对象

第 3 章　进程调度模块

3.1　内核异步模型：多核无栈协程

3.1.1　传统有栈协程设计

传统的有栈协程设计中，每个用户线程都有自己的内核栈，线程的上下文切换需要保存和恢复栈指针。

以 rCore 为例，如图 3-1，上下文切换需要首先保存当前线程的 CPU 寄存器快照到当前线程的栈上，并根据目标线程栈上保存的内容来恢复相应的寄存器，最后切换栈指针。这样的设计在多核环境下存在一些问题。首先，每个线程的内核栈占用了大量内存，并且内核栈的大小很难进行动态调整，或是需要付出很大代价。其次，内核栈的保存和恢复需要大量的指令，影响了性能。最后，上下文切换需要考虑是否有互斥锁等资源未释放，一方面容易造成死锁，另一方面也不太符合 Rust RAII 的原则。

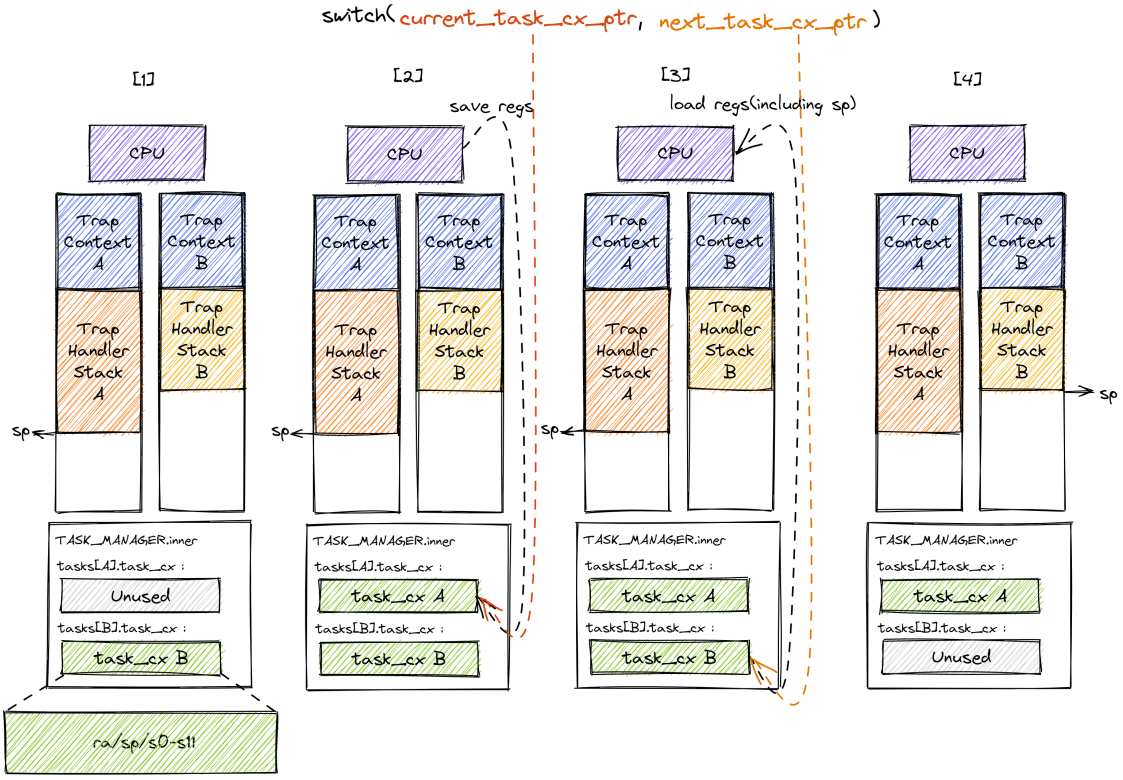

图 3-1　rCore 上下文切换

3.1.2 Rust 异步模型：Async 和 Future

Rust 在语言层面提供了异步编程的支持，通过 `Future` 实现异步模型。`Future` 是一个异步计算的抽象，它代表了一个异步计算的结果。`Future` 在 Rust 中是惰性的，只有在被轮询（poll）时才会运行。`Future` 通过实现 `poll` 方法来定义异步计算的行为。当 `Future` 被轮询时，它会返回一个 `Poll` 枚举，表示异步计算的状态。`Poll` 枚举有两个成员：`Ready` 表示异步计算已经完成，`Pending` 表示异步计算尚未完成。`Future` 通过 `poll` 方法的返回值来告诉调用者异步计算的状态。

Rust 的 `async` 和 `await` 关键字用于简化异步编程。`async` 关键字用于定义异步函数，`await` 关键字用于等待异步计算的结果。`async` 函数返回一个 `Future` 对象，`await` 关键字会将 `Future` 对象的执行挂起，直到异步计算完成。在实现上，`async` 函数会被编译成一个状态机，通过 `Future` 对象的 `poll` 方法来实现异步计算。

Async 在 Rust 中的脱糖是零开销的，这意味着无需分配任何堆内存、也无需任何动态分发。状态机的构建使得切换异步程序不再需要换栈，而是变为函数调用和返回，本文会在后面详细介绍这点。因此，Rust 的异步模型非常适合用于实现高并发的异步编程，尤其是操作系统内核这种存在大量异步 I/O 的程序。

3.1.3 异步运行时

由于 Rust 没有内置异步调用所必需的运行时，而大部分库都不支持裸机环境，且异步组件往往强依赖于运行时，因此 MinotaurOS 需要自行实现异步运行时及其组件，这包括 executor、异步锁和各种类型的 `Future` 。

MinotaurOS 的异步执行器基于 async_task 库实现。async_task 库提供了构建执行器的基本抽象，包括 `Runnable` 和 `Task` 。一个 `Runnable` 对象持有一个 `Future` 句柄，在运行时，会对 `Future` 轮询一次。然后，`Runnable` 会消失，直到 `Future` 被唤醒才再次进入调度。一个 `Task` 对象用于获取 `Future` 的结果，通过 `detatch` 方法将任务移入后台执行。async_task 提供了 `spawn` 方法，传入 `Future` 和调度器，用于创建 `Runnable` 和 `Task` 对象。然后，通过 `runnable.schedule` 方法将 `Runnable` 加入调度序列。显而易见的优点是，I/O 阻塞的任务不会进入调度序列，避免了忙等待。

MinotaurOS 的调度方式为改进的轮转调度（如代码 3-1 所示）。MinotaurOS 维护两个全局的无锁任务队列，一个是 FIFO 队列，一个是优先级队列，保存 `Runnable` 对象。调度开始时，CPU 核心首先尝试从优先级队列中取出一个 `Runnable` 运行，若优先级队列为空，则从 FIFO 队列中取出一个 `Runnable` 运行。在任务调度上，若 `Runnable` 在运行时被唤醒，通常是因为任务执行了 yield，让出时间片，此时将

`Runnable` 放入 FIFO 队列；若 `Runnable` 在其他任务运行时被唤醒，则通常是因为异步 I/O 完成的中断，此时将 `Runnable` 放入优先队列。

```
struct TaskQueue {
  fifo: SegQueue<Runnable>,
  prio: SegQueue<Runnable>,
}

pub fn spawn<F>(future: F) -> (Runnable, Task<F::Output>)
    where
        F: Future + Send + 'static,
        F::Output: Send + 'static,
{
    let schedule = move |runnable: Runnable, info: ScheduleInfo| {
        if info.woken_while_running {
          TASK_QUEUE.push_fifo(runnable);
        } else {
          TASK_QUEUE.push_prio(runnable);
        }
    };
    async_task::spawn(future, WithInfo(schedule))
}
```

代码 3-1　调度器实现

3.1.4　多核调度和无栈上下文切换

MinotaurOS 支持多个 CPU 核心的运行。每个核心可以平等地从调度序列中取出 `Runnable` 运行。为了实现这个目的，每个 CPU 核心都有一个 thread local 的数据结构 `Hart`，包含核心 ID 和核心上下文。`tp` 寄存器保存了当前核心的 ID，通过这个 ID 可以获取到当前核心的 `Hart`。核心上下文 `HartContext` 保存了当前核心正在运行的用户任务，其中包含 `Thread` 对象的引用、页表和地址空间 token。上述数据结构定义如代码 3-2 所示。

```rust
pub struct Hart {
    pub id: usize,
    pub ctx: HartContext,
    pub on_kintr: bool,
    pub on_page_test: bool,
    pub last_page_fault: SyscallResult,
    kintr_rec: usize,
    asid_manager: LateInit<ASIDManager>,
}

pub struct HartContext {
    pub user_task: Option<UserTask>,
    pub last_syscall: SyscallCode,
    pub timer_during_sys: usize,
}

pub struct UserTask {
    pub thread: Arc<Thread>,
    pub token: usize,
    pub root_pt: PageTable,
}
```

代码 3-2　CPU 核心数据结构

内核线程和用户线程使用统一的数据结构 `HartTaskFuture`。该结构包含了核心上下文和具体的异步任务 `Future`，其中对于内核线程，核心上下文中保存的 `UserTask` 为空。`HartTaskFuture` 的 poll 过程会首先将当前核心的上下文与该结构中的上下文交换，然后 poll 保存的 `Future`，执行异步任务的方法体（若就绪），最后再次交换上下文。上下文切换遵循下面的步骤：

（1）若当前任务是用户线程，则更新线程的调度换出时间；

（2）若目标任务是用户线程，则更新线程的调度换入时间；

（3）若目标任务是用户线程，且当前任务是内核线程或当前用户线程与目标用户线程不属于同一进程，则激活目标进程的地址空间；

（4）若目标任务是内核线程，且当前任务是用户线程，则激活内核地址空间；

（5）交换上下文数据。

与有栈协程相比，上下文切换不涉及到栈指针的切换和寄存器的保存与恢复（上下文数据交换只是两个指针的交换），因此开销更小。同时，由于异步任务的执行是在异步执行器的控制下，因此不需要考虑是否有互斥锁等资源未释放，也不需要考虑是否会因为线程过多而导致栈溢出等问题。这样的设计使得 MinotaurOS 能够充分利用多核特性，实现内核异步模型，如图 3-2 所示。

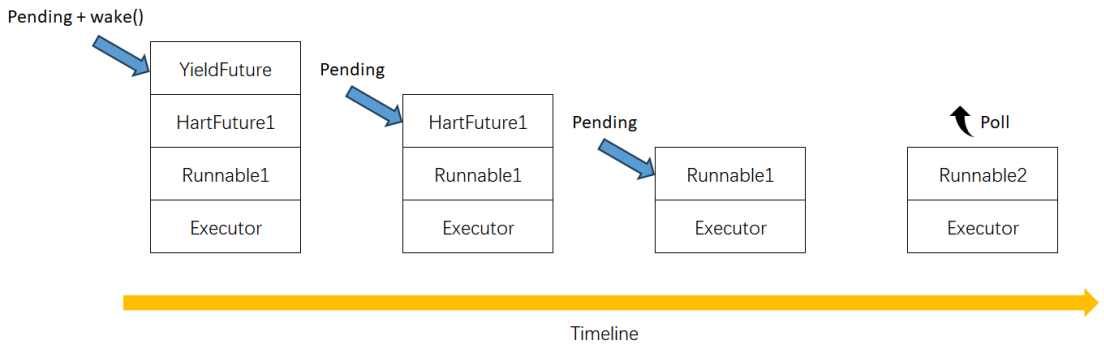

图 3-2　无栈协程调度

3.1.5　线程循环

用户线程的 `Future` 是一个无限循环（如代码 3-3 所示）。线程在创建后，首先调用 `trap_return` 从内核态返回用户态。Trap 发生时（无论是系统调用或是时钟中断），寄存器信息会被保存到线程的 Trap 上下文中，然后从 `trap_return` 返回，接着执行异步函数 `trap_from_user`。如果 Trap 来自 yield 系统调用或是时钟中断，`YieldFuture.await` 会使线程挂起，并将线程放回调度序列尾部，接着完成上述的 `HartTaskFuture` 上下文切换后半部分。下次调度到该线程时，会从挂起的地方继续执行，从而再次通过 `trap_return` 返回用户态。

```rust
async fn thread_loop(thread: Arc<Thread>) {
  loop {
      trap_return();
      trap_from_user().await;
      check_signal();
      if thread.inner().exit_code.is_some() {
          break;
      }
  }
  thread.on_exit();
}
```

代码 3-3　线程循环

3.2　进程控制块设计

在 Linux 源码中，进程和线程是使用统一的结构 `task_struct` 来表示的。这是由于历史原因，Linux 早期没有线程的概念，线程是通过进程来实现的。在 MinotaurOS 中，为了更加清晰地表示进程和线程的关系，以及减少互斥锁的使用，设计了两个不同的结构 `Process` 和 `Thread`（如代码 3-4 和代码 3-5 所示）。`Process` 结构表示一个进程，`Thread` 结构表示一个线程。`Process` 结构包含了进程的基本信

息，包括父子进程、线程组、地址空间等。`Thread` 结构包含了线程的基本信息，包括所属进程、线程上下文等。

```rust
pub struct Process {
    pub pid: Arc<TidTracker>,
    pub inner: IrqReMutex<ProcessInner>,
}

pub struct ProcessInner {
    pub parent: Weak<Process>,                  // 父进程
    pub children: Vec<Weak<Process>>,           // 子进程
    pub pgid: Gid,                              // 进程组
    pub threads: BTreeMap<Tid, Weak<Thread>>,   // 进程的线程组
    pub addr_space: AddressSpace,               // 地址空间
    pub mnt_ns: Arc<MountNamespace>,            // 挂载命名空间
    pub fd_table: FdTable,                      // 文件描述符表
    pub futex_queue: FutexQueue,                // 互斥锁队列
    pub timers: [ITimerVal; 3],                 // 定时器
    pub cwd: String,                            // 工作目录
    pub exe: String,                            // 可执行文件路径
    pub exit_code: Option<i8>,                  // 退出状态
}
```

代码 3-4　进程控制块

```rust
pub struct Thread {
    pub tid: Arc<TidTracker>,
    pub process: Arc<Process>,
    pub signals: SignalController,
    pub event_bus: EventBus,
    pub cpu_set: Mutex<CpuSet>,
    inner: SyncUnsafeCell<ThreadInner>,
}

pub struct ThreadInner {
    pub trap_ctx: TrapContext,
    pub tid_address: TidAddress,
    pub rusage: ResourceUsage,
    pub exit_code: Option<i8>,
}
```

代码 3-5　线程控制块

进程控制块和线程控制块的可变部分使用了不同的可变容器。进程的可变部分需要使用互斥锁进行保护，这是因为可能存在多个 CPU 核心的线程同时访问同一个进程的可变部分。而线程的可变部分只会被当前线程访问，因此使用了 `UnsafeCell` 来避免互斥锁的开销。

进程控制块中对线程组的引用是弱引用，相反线程控制块中对进程的引用是强引用。这是因为线程才是真正运行在 CPU 核心上的实体，而进程是线程的集合。当所有线程都终止并 drop 后，进程会因为所有强引用都消失而被 drop，从而释放进程占用的资源并向父进程发送 SIGCHLD 信号。而之所以进程控制块中对父进程和子进程的引用都是弱引用，也是这个原因。进程和线程的关系如图 3-3 所示。

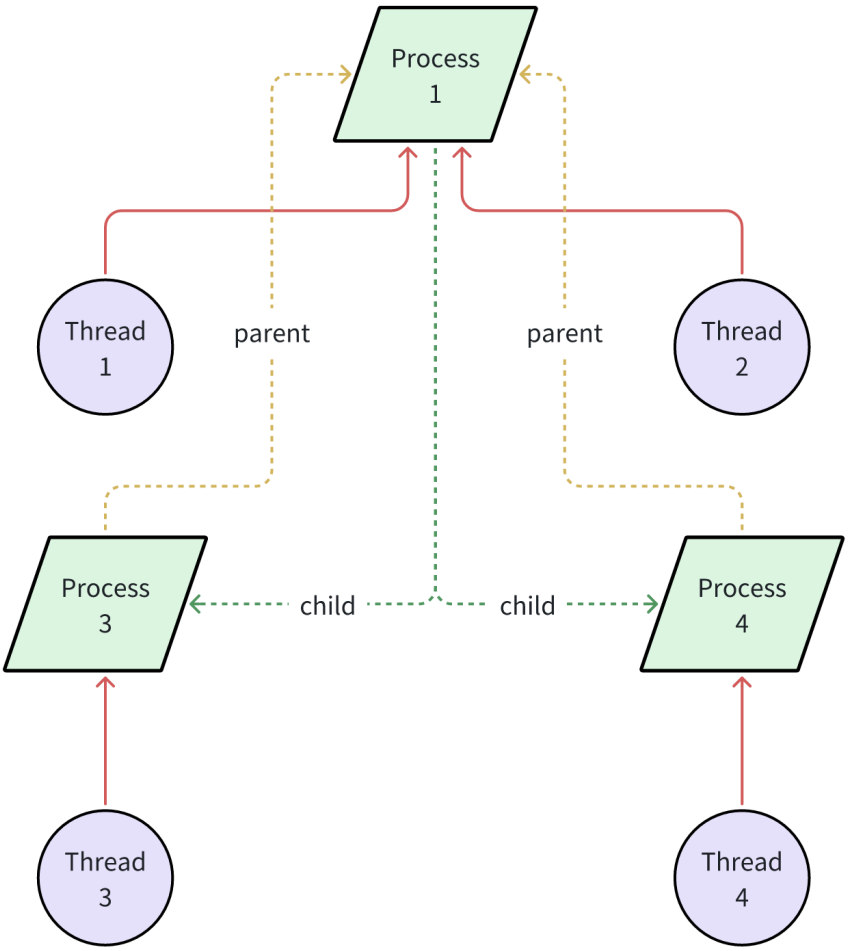

图 3-3　进程和线程关系

第 4 章　设备驱动模块

4.1　设备树解析

MinotaurOS 使用设备树（Device Tree）来描述硬件信息。设备树是一种描述硬件信息的数据结构，通常以 `.dts` 或 `.dtb` 文件的形式存在。设备树的结构是一个树状结构，每个节点表示一个硬件设备，节点之间通过属性和子节点进行关联。

在引导流程中，SBI 在完成硬件初始化后跳转到内核时，会将设备树的地址使用 `a1` 寄存器传递。MinotaurOS 在内核启动时，会解析设备树，根据设备树完成硬件识别。设备树解析的过程如下：

（1）解析 `cpus` 节点，获取时钟频率，CPU 核心个数和每个核心的信息；

（2）解析 `memory` 节点，获取内存的基地址和大小；

（3）解析 `soc` 节点，获取 SoC 的信息，包括串口、时钟、中断控制器等，并记录 MMIO 地址。

通过实现设备树解析功能，MinotaurOS 能够实现一次编译运行在不同的硬件平台上，而无需修改内核代码。

设备树描述的内存布局会保存到 `GlobalMapping` 结构中（如代码 4-1 所示）。`GlobalMapping` 记录了内存区域的物理地址、虚拟地址和大小，其中最重要的两个区域是物理内存和 virtio_mmio / sdcard_mmio：前者包含了内核镜像和用户页帧需要的内存，后者包含了磁盘设备的 MMIO 地址。

```
1  pub struct GlobalMapping {
2      pub name: String,
3      pub phys_start: PhysAddr,
4      pub virt_start: VirtAddr,
5      pub size: usize,
6      pub perms: ASPerms,
7  }
```

代码 4-1　GlobalMapping 结构

解析设备树之前，需要首先初始化堆分配器，保证 `GlobalMapping` 能够构造。在设备树解析完成后，会初始化用户页帧分配器，然后构造内核地址空间和页表。在此之后，会根据 `GlobalMapping` 记录的 MMIO 信息完成设备驱动的初始化。

MinotaurOS 使用 `Device` 结构表示一个硬件设备，如代码 4-2 所示。MinotaurOS 支持三种类型的设备：块设备、字符设备和网络设备。每种设备类型都有对应的接

口。设备驱动程序需要实现对应的接口，并将设备注册到全局驱动表中。每种设备类型都提供了 `metadata` 方法，获取设备的元数据。设备元数据包含设备名和唯一的设备 ID，用于在设备表中查找设备。

```rust
#[derive(Clone)]
pub enum Device {
    Block(Arc<dyn BlockDevice>),
    Character(Arc<dyn CharacterDevice>),
    Network(Arc<dyn NetworkDevice>),
}

pub struct DeviceMeta {
    pub dev_id: usize,
    pub dev_name: String,
}
```

代码 4-2　Device 结构

4.2　块设备驱动

块设备驱动是 MinotaurOS 设备驱动的一种，用于访问块设备。块设备是一种随机访问设备，数据以块为单位进行读写。MinotaurOS 的块设备驱动接口定义如代码 4-3 所示。块设备驱动需要实现 `BlockDevice` 接口，包括获取设备元数据、初始化、读取块数据和写入块数据四个方法。

```rust
pub trait BlockDevice: Send + Sync {
    /// 设备元数据
    fn metadata(&self) -> &DeviceMeta;

    /// MMIO 映射完成后初始化
    fn init(&self);

    /// 从块设备读取数据
    async fn read_block(&self, block_id: usize, buf: &mut [u8])
        -> SyscallResult;

    /// 向块设备写入数据
    async fn write_block(&self, block_id: usize, buf: &[u8])
        -> SyscallResult;
}
```

代码 4-3　块设备接口

4.2.1　VirtIO

MinotaurOS 使用 VirtIO 块设备驱动来访问 QEMU 模拟器的磁盘设备。VirtIO 是一种虚拟化设备标准，用于在虚拟机和宿主机之间传输数据。VirtIO 设备包括

VirtIO 网卡、VirtIO 块设备等。VirtIO 设备通过 MMIO 地址进行访问，MinotaurOS 使用 virtio-drivers 库来实现 VirtIO 设备的操作。但是，由于 virtio-drivers 库对异步的支持十分有限，因此 MinotaurOS 采用轮询的方式进行同步读写，导致了一定程度的性能损失。

4.3 字符设备驱动

字符设备驱动是 MinotaurOS 设备驱动的一种，用于访问字符设备。字符设备是一种流设备，数据以字符为单位进行读写。MinotaurOS 的字符设备驱动接口定义如代码 4-4 所示。字符设备驱动需要实现 `CharacterDevice` 接口，包括获取设备元数据、读取字符和写入字符等方法。与块设备不同，字符设备的读写是串行的，没有块 ID 的概念。

```rust
pub trait CharacterDevice: Send + Sync {
    /// 设备元数据
    fn metadata(&self) -> &DeviceMeta;

    /// MMIO 映射完成后初始化
    fn init(&self);

    /// 是否有数据
    fn has_data(&self) -> bool;

    /// 注册唤醒器
    fn register_waker(&self, waker: Waker);

    /// 从字符设备读取数据
    async fn getchar(&self) -> SyscallResult<u8>;

    /// 向字符设备写入数据
    async fn putchar(&self, ch: u8) -> SyscallResult;
}
```

代码 4-4　字符设备接口

4.3.1 TTY 驱动

TTY 通常用于描述一个文本输入/输出环境。在 UNIX 和 UNIX-like 系统（如 Linux）中，TTY 用来描述任何一种文本终端，无论它是物理设备还是虚拟设备。传统的 TTY 实现方式是使用 SBI 的 DBCN 扩展，但是直接使用 SBI 存在一定的局限性，例如无法实现 ppoll 系统调用和异步读写。因此，MinotaurOS 使用 ns16550a 兼容的 UART 驱动来实现 TTY，通过解析如代码 4-5 所示的设备树串口结点，从对应的 MMIO 寄存器读写和处理外设中断来实现串口的输入和输出。

```
 serial@10000000 {
    interrupts = <0x0a>;
    interrupt-parent = <0x05>;
    clock-frequency = "\08@";
    reg = <0x00 0x10000000 0x00 0x100>;
    compatible = "ns16550a";
};
```

代码 4-5　serial dts

MinotaurOS 通过如下的步骤进行串口读写：

（1）设备结构体保存了一个 `AtomicU8` 类型的缓冲区，当尝试从设备读取字符时，首先检查缓冲区是否为空，如果不为空，则从缓冲区取出字符返回；

（2）如果缓冲区为空，则获取 `LSR` 寄存器信息，判断串口是否有数据待读取，如果有，则从串口取出字符返回；

（3）如果串口没有数据待读取，则注册一个唤醒器，等待串口中断唤醒后再次尝试读取；

（4）当中断发生时，设备从串口读取字符，将字符存入缓冲区，并唤醒等待的任务。

（5）写入字符时，直接以同步方式轮询向串口写入。

```rust
async fn getchar(&self) -> SyscallResult<u8> {
poll_fn(|cx| unsafe {
  // Fast path
  let val = self.buf.swap(0xff, Ordering::Relaxed);
  if val != 0xff {
    return Poll::Ready(Ok(val));
  } else if self.line_status_ptr().read_volatile() & 0x01 == 0x01 {
    return Poll::Ready(Ok(self.rxdata_ptr().read_volatile()));
  }

  self.waker.register(cx.waker());

  // Slow path
  if self.buf.swap(0xff, Ordering::Relaxed) != 0xff {
      Poll::Ready(Ok(self.buf.load(Ordering::Relaxed)))
  } else {
      Poll::Pending
  }
}).await
}

fn handle_irq(&self) {
  let ch = unsafe { self.rxdata_ptr().read_volatile() };
  if ch == CTRL_C {
      DEFAULT_TTY.handle_ctrl_c();
  }
  self.buf.store(ch, Ordering::Relaxed);
  self.waker.wake();
}
```

代码 4-6　TTY 读取字符

4.3.2　零设备和空设备驱动

`/dev/zero` 是一个特殊的字符设备，用于提供无限的空数据流。读取 `/dev/zero` 会返回无限的 0 字节，写入 `/dev/zero` 则会被忽略。`/dev/null` 是一个特殊的字符设备，用于丢弃数据。读取 `/dev/null` 会返回空数据，写入 `/dev/null` 则会被忽略。

4.3.3　随机数设备

MinotaurOS 提供了随机数设备 `/dev/urandom`。目前，MinotaurOS 并未使用专门的熵池来生成随机数，而是直接使用当前的 CPU 时间戳作为种子生成随机数。因此，`/dev/urandom` 并不是真正的随机数设备，而是伪随机数设备。

第 5 章　文件系统模块

5.1　挂载命名空间

MinotaurOS 提供了类似 Linux 的挂载命名空间功能。每个进程包含了一个 `MountNamespace` 引用，`MountNamespace` 对象包含一颗挂载树、挂载树的平坦快照和 Inode 缓存。`MountTree` 对象包含一个文件系统指针和子树的 map，如代码 5-1 所示。

```rust
pub struct MountNamespace {
    pub mnt_ns_id: usize,
    pub caches: [InodeCache; 2],
    inner: Mutex<NSInner>,
}

struct NSInner {
    tree: MountTree,
    snapshot: HashMap<usize, (usize, String)>,
}

struct MountTree {
    mnt_id: usize,
    fs: Arc<dyn FileSystem>,
    /// 子挂载树，`key` 为挂载路径，以 `/` 开头
    sub_trees: BTreeMap<String, MountTree>,
}
```

代码 5-1　挂载命名空间和挂载树

5.2　文件系统

MinotaurOS 实现了类似 Linux 虚拟文件系统功能。为了支持不同类型的文件系统，MinotaurOS 设计了一个通用的接口 `FileSystem`。`FileSystem` trait 包含两个方法 `metadata` 和 `root`，分别用于获取文件系统的元数据和根目录 `Inode`。文件系统元数据包含了文件系统的唯一标识符、设备号、挂载源、文件系统类型和文件系统标志。文件系统接口定义如代码 5-2 所示。

```rust
/// 文件系统元数据
/// 一个文件系统在刚创建时不关联任何挂载点，
/// 通过 `move_mount` 挂载到命名空间。
pub struct FileSystemMeta {
    /// 唯一标识符
    fsid: usize,
    /// 设备号
    pub dev: u64,
    /// 挂载源
    pub source: String,
    /// 文件系统类型
    pub fstype: FileSystemType,
    /// 文件系统标志
    pub flags: VfsFlags,
}

/// 文件系统
pub trait FileSystem: Send + Sync {
    /// 文件系统元数据
    fn metadata(&self) -> &FileSystemMeta;
    /// 根 Inode
    fn root(&self) -> Arc<dyn Inode>;
}
```

代码 5-2　文件系统接口

文件系统与挂载命名空间在结构上是解耦的，文件系统只关心自己内部的目录结构，而挂载命名空间则负责管理全局的路径抽象。这样的设计使得 MinotaurOS 能够支持更多的文件系统类型，同时也使得文件系统的实现更加简单。

5.3　Inode

Inode 是文件系统的核心结构，其唯一地标识了文件系统中的一个文件或目录。在 MinotaurOS 中，Inode 采用了较为复杂的两层接口组成。内层的 `InodeInternal` 接口定义了文件系统的“物理”操作，即直接在块设备上的读写、创建、删除等。外层的 `Inode` 接口定义了文件系统的“逻辑”操作，即包含了元数据、挂载点、和页缓存下的读写操作等。同时，通过隐藏内部接口，防止不经意间绕过了中间缓存结构，MinotaurOS 保证了文件系统的安全性和封装性。`Inode` 和 `InodeInternal` 接口定义如代码 5-3 和代码 5-4 所示。

```rust
#[async_trait]
pub(super) trait InodeInternal {
  /// 从 `offset` 处读取 `buf`，绕过缓存
  async fn read_direct(&self, buf: &mut [u8], offset: isize)
    -> SyscallResult<isize>

  /// 向 `offset` 处写入 `buf`，绕过缓存
  async fn write_direct(&self, buf: &[u8], offset: isize)
    -> SyscallResult<isize>

  /// 设置文件大小，绕过缓存
  async fn truncate_direct(&self, size: isize)
    -> SyscallResult

  /// 查询目录项
  async fn do_lookup_name(self: Arc<Self>, name: &str)
    -> SyscallResult<Arc<dyn Inode>>

  /// 查询目录项
  async fn do_lookup_idx(self: Arc<Self>, idx: usize)
    -> SyscallResult<Arc<dyn Inode>>

  /// 在当前目录下创建文件/目录
  async fn do_create(self: Arc<Self>, mode: InodeMode, name: &str)
    -> SyscallResult<Arc<dyn Inode>>

  /// 在当前目录下创建符号链接
  async fn do_symlink(self: Arc<Self>, name: &str, target: &str)
    -> SyscallResult

  /// 将文件移动到当前目录下
  async fn do_movein(
      self: Arc<Self>,
      name: &str,
      inode: Arc<dyn Inode>,
  ) -> SyscallResult

  /// 在当前目录下删除文件
  async fn do_unlink(self: Arc<Self>,target: Arc<dyn Inode>)
    -> SyscallResult

  /// 读取符号链接
  async fn do_readlink(self: Arc<Self>)
    -> SyscallResult<String>
}
```

代码 5-3　InodeInternal 接口

```rust
#[async_trait]
pub trait Inode: DowncastSync + InodeInternal {
  /// 获取 Inode 元数据
  fn metadata(&self) -> &InodeMeta;

  /// 获取文件系统
  fn file_system(&self) -> Weak<dyn FileSystem>;

  fn ioctl(&self, request: usize, value: usize)
    -> SyscallResult<i32>
}
impl_downcast!(sync Inode);
```

代码 5-4　Inode 接口

为了使所有的文件系统能够拥有相同的数据抽象，MinotaurOS 定义了统一的文件元数据结构 `InodeMeta`（如代码 5-5 所示）。`InodeMeta` 结构包含了文件的基本信息，包括文件的设备号、文件名和路径等。

```rust
pub struct InodeMeta {
    pub key: usize,
    /// 结点编号
    pub ino: usize,
    /// 结点设备
    pub dev: u64,
    /// 结点类型
    pub mode: InodeMode,
    /// 文件名
    pub name: String,
    /// 文件系统路径
    pub path: String,
    /// 父目录
    pub parent: Option<Weak<dyn Inode>>,
    /// 页面缓存
    pub page_cache: Option<Arc<PageCache>>,
    /// 可变数据
    pub inner: Arc<Mutex<InodeMetaInner>>,
}
```

代码 5-5　InodeMeta

会发生变化对象放在了 `InodeMetaInner` 结构内。特别之处在于，整个结构包含在一个 `Arc<Mutex>` 中，这是为了实现移动文件的一致性。在移动文件时，需要创建一个新的 Inode。而如果旧 Inode 上尚有未关闭的文件描述符，通过旧 Inode 仍然可以访问到物理的文件。因此，需要将元数据的可变部分放在一个独立的结构中，以便在移动文件时，只需要修改元数据的指针即可。`InodeMetaInner` 结构定义如代码 5-6 所示。

```rust
pub struct InodeMetaInner {
    /// uid
    pub uid: usize,
    /// gid
    pub gid: usize,
    /// 硬链接数
    pub nlink: usize,
    /// 访问时间
    pub atime: TimeSpec,
    /// 修改时间
    pub mtime: TimeSpec,
    /// 创建时间
    pub ctime: TimeSpec,
    /// 文件大小
    pub size: isize,
    /// 父目录
    pub parent: Weak<dyn Inode>,
    /// 挂载点
    pub mounts: BTreeMap<String, Arc<dyn Inode>>,
}
```

代码 5-6　InodeMetaInner

值得注意的一点是，Inode 的元数据中包含了父节点的弱引用和挂载点。这是为了支持相对路径的解析。在将一个文件系统挂载加入挂载树的同时，MinotaurOS 会将子文件系统的根结点引用加入到挂载点的元数据中。这样，当解析相对路径时，会首先根据挂载点的元数据找到对应的子文件系统，再在子文件系统中查找；当不存在相应挂载点时，再在当前文件系统中查找。除此之外，另一个特点是 `InodeMeta` 的文件系统路径仅是相对于该文件系统根目录的路径，而不是相对整个挂载命名空间的路径。

这样的设计使得文件系统与挂载树解耦，并让 MinotaurOS 能够在未来兼容更多的高级文件系统挂载功能，如 `chroot`、`move_mount` 等系统调用。目前为止，MinotaurOS 实现了基本的文件系统挂载功能，包括 `mount` 和 `umount` 系统调用。

5.4　文件读写接口 File

`File` 接口是对“打开的文件”的抽象。与 `Inode` 不同，`File` 是对文件的操作的抽象，而不是文件对应的物理结点。`File` trait 定义了文件的基本操作，包括读、写、pollin、pollout 等，通常由 `Inode` 的 `open` 方法得到。`File` 的元数据中包含了一个 `Inode` 的引用和打开文件的 Flags。文件的类型包括普通文件、目录文件、设备文件、管道和 socket。每种文件类型都有对应的实现，这些实现在不同的文件系统中是通用的。

5.4.1 普通文件

普通文件结构中存储了文件元数据、读写偏移量，以及 `pread`、`pwrite` 操作的锁，如代码 5-7 所示。普通文件实现了 `seek` 方法，并通过 Inode 的 `read`、`write` 方法进行具体的读写操作。

```
pub struct RegularFile {
    metadata: FileMeta,
    pos: AsyncMutex<isize>,
    prw_lock: AsyncMutex<()>,
}
```

代码 5-7　RegularFile

5.4.2 目录文件

目录文件结构中存储了文件元数据和读写偏移量。目录文件实现了 `readdir` 方法，用于读取目录项，如代码 5-8 所示。值得注意的是，MinotaurOS 不将“.”和“..”作为独立的 Inode 存储，而是将读取目录时的 0 号和 1 号直接指向自身和父目录。这是因为 Inode 是树型结构，需要防止出现循环引用。

```
pub struct DirFile {
    metadata: FileMeta,
    pos: AsyncMutex<usize>,
}

async fn readdir(&self)
  -> SyscallResult<Option<(usize, Arc<dyn Inode>)>> {
    let inode = self.metadata.inode.as_ref().unwrap();
    let mut pos = self.pos.lock().await;
    let inode = match *pos {
        0 => inode.clone(),
        1 => inode.metadata().parent.clone()
                .and_then(|p| p.upgrade())
                .unwrap_or(inode.clone()),
        _ => match inode.clone().lookup_idx(*pos - 2).await {
            Ok(inode) => inode,
            Err(Errno::ENOENT) => return Ok(None),
            Err(e) => return Err(e),
        },
    };
    *pos += 1;
    Ok(Some((*pos - 1, inode)))
}
```

代码 5-8　DirFile

5.4.3 设备文件

字符设备和块设备文件仅包含文件元数据，并将读写和 ioctl 操作委托给对应的设备驱动。

5.4.4 管道

管道（Pipe）是一种特殊的文件，用于进程间通信。管道的读写两端由相同的数据结构构成，如代码 5-9 所示。管道结构中包含了文件元数据、读写标志、另一端的弱引用和内部数据结构。内部数据结构包含了缓冲区、已传输的字节数和读写等待队列。管道的两端共用一个内部数据结构，通过读写标志区分读写操作。

```rust
pub struct Pipe {
    metadata: FileMeta,
    is_reader: bool,
    other: LateInit<Weak<Pipe>>,
    inner: Arc<Mutex<PipeInner>>,
}

#[derive(Default)]
struct PipeInner {
    buf: VecDeque<u8>,
    transfer: usize,
    readers: VecDeque<Waker>,
    writers: VecDeque<Waker>,
}
```

代码 5-9　Pipe

管道没有关联的 Inode，它的读写由专门的管道函数处理。管道的读写操作是异步的，读写两端分别通过 `PipeReadFuture` 和 `PipeWriteFuture` 实现异步读写操作。以 `PipeReadFuture` 为例：该结构中包含了管道对象的引用、系统调用的缓冲区和读取位置。`PipeReadFuture` 实现了 `Future` trait，通过 `poll` 方法实现了管道的异步读取操作。当管道缓冲区中有数据时，将数据读取到系统调用的缓冲区中，并唤醒等待的写者；当缓冲区为空时，将当前读者加入到等待队列中，等待写者唤醒；当管道的另一端被关闭时，返回读取的字节数为 0。

```rust
struct PipeReadFuture<'a> {
    pipe: &'a Pipe,
    buf: &'a mut [u8],
    pos: usize,
}
```

代码 5-10　PipeReadFuture

```rust
impl Future for PipeReadFuture<'_> {
  type Output = SyscallResult<isize>;

  fn poll(mut self: Pin<&mut Self>, cx: &mut Context<'_>)
    -> Poll<Self::Output> {
    let mut inner = self.pipe.inner.lock();
    let read = min(self.buf.len() - self.pos, inner.buf.len());
    if read > 0 {
      for (i, b) in inner.buf.drain(..read).enumerate() {
        self.buf[i] = b;
      }
      self.pos += read;
      inner.transfer += read;
      while let Some(waker) = inner.writers.pop_front() {
        waker.wake();
      }
      Poll::Ready(Ok(read as isize))
    } else {
      if self.pipe.other.strong_count() == 0 {
        return Poll::Ready(Ok(0));
      }
      inner.readers.push_back(cx.waker().clone());
      Poll::Pending
    }
  }
}
```

代码 5-11　PipeReadFuture poll

5.5　Inode 缓存

文件系统的访问存在相当的时间局部性，往往存在于绝对路径、父目录和子目录之间。在 Linux 当中，使用了哈希表来加快 Inode 的查找过程。MinotaurOS 也实现了类似的缓存机制。

在 MinotaurOS 中，我们采用了一套查询零拷贝的缓存机制。`MountNamespace` 中保存了两个 `InodeCache`，分别用于绝对路径和相对路径的解析。`InodeCache` 是一个哈希表，键为 `HashKey`，值为弱引用的 `Inode`。`HashKey` 由父节点的 Inode key 和子路径组成，如代码 5-12 所示。

```rust
#[derive(Eq, Hash, PartialEq, Clone, Debug)]
struct HashKey<'a> {
    pub parent_key: usize,
    pub subpath: Cow<'a, str>,
}

pub struct InodeCache(Mutex<HashMap<
    HashKey<'static>,
    Weak<dyn Inode>,
>>)
```

代码 5-12　HashKey 和 InodeCache

Inode key 是全局自增的，在不同文件系统中，Inode 可能有相同的 ino，但不会有相同的 key。子路径由一个 `Cow` 类型的字符串表示，在 `HashKey` 中，子路径字符串只有在插入缓存时才会被复制构造，而在查询缓存时，子路径字符串作为 `Cow::Borrowed` 类型从调用者借用，避免了不必要的内存拷贝。

```rust
// 插入缓存，复制构造
let hash_key = HashKey::new(parent_key, Cow::Owned(subpath));
// 查询缓存，零拷贝
let hash_key = HashKey::new(parent_key, Cow::Borrowed(subpath));
```

代码 5-13　InodeCache 插入和查询

5.6　路径解析

图 5-1 是一个路径解析的例子。访问绝对路径从根挂载树开始，依次匹配子树的路径前缀，遍历相应子树，直到最后一颗不包含匹配前缀的子树，然后将路径解析委托给相应文件系统。对于文件系统内部而言，所有的路径解析都是相对于该文件系统的根目录进行的。

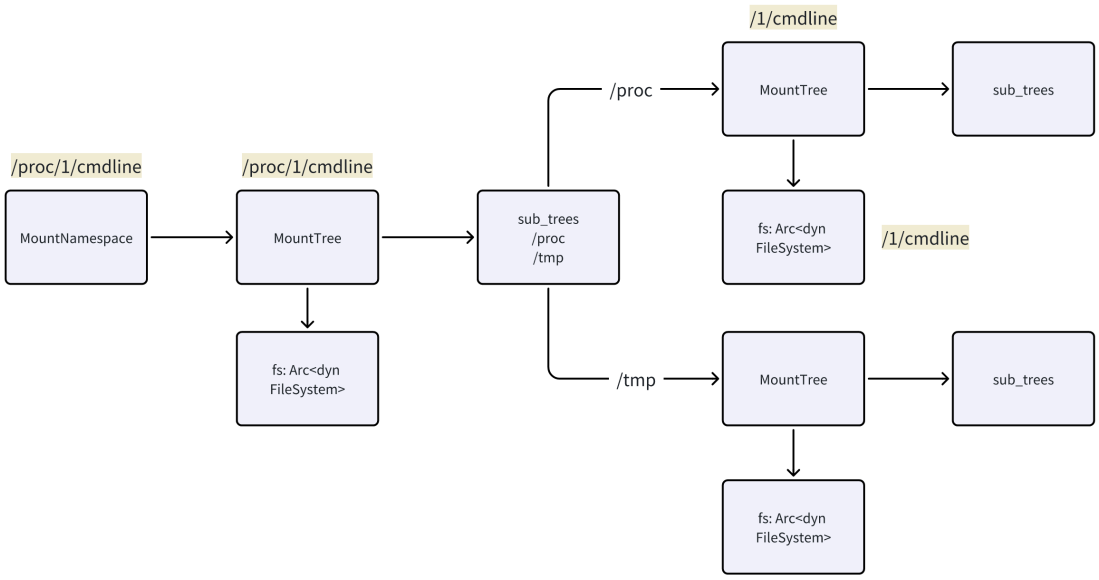
/1/cmdline
/proc
MountTree
sub_trees
/proc/1/cmdline
/proc/1/cmdline
MountNamespace
MountTree
sub_trees
/proc
/tmp
fs: Arc<dyn
FileSystem>
/1/cmdline
fs: Arc<dyn
FileSystem>
/tmp
MountTree
sub_trees
fs: Arc<dyn
FileSystem>

图 5-1　绝对路径解析

第 6 章　中断处理模块

6.1　中断切换

为了方便起见，本文不对中断和异常做额外区分。RISC-V 架构有两种处理中断的方式：直接模式（Direct）和向量模式（Vectored）。在直接模式下，所有的中断都会跳转到 `stvec` 寄存器所设定的基地址进行处理；在向量模式下，不同的中断会根据基地址加中断号确定中断向量表中处理函数的位置。为了方便调试和实现统一的中断处理，MinotaurOS 使用直接模式处理中断。

中断发生时，RISC-V 处理器会将中断号保存到 `scause` 寄存器中，将中断发生时的指令地址保存到 `sepc` 寄存器中。之后，跳转到 `stvec` 寄存器设定的中断处理程序。对于内核态中断和用户态中断，MinotaurOS 使用不同的中断处理函数。

对于内核态中断，MinotaurOS 只需要保存 caller-saved 寄存器，然后调用 `trap_from_kernel` 函数。这是因为内核态中断不会改变内核的栈帧，在控制流角度就像插入了一次普通的函数调用，编译器会自动保存和恢复 callee-saved 寄存器。

对于用户态中断，MinotaurOS 会保存中断上下文（如代码 6-1 所示），并恢复内核 callee-saved 寄存器和切换内核栈，然后执行 `ret` 。这是因为在编译器看来，当前内核栈状态是执行了 `trap_return` 函数，因此需要在返回前恢复 callee-saved 寄存器。此时，内核的控制流会回到协程调度器管理下的 `thread_loop` 函数中（见第 4.1.5 节）。然后，再执行对应的中断处理函数。处理完成后，MinotaurOS 会调用 `trap_return` 函数，恢复之前保存的中断上下文，然后执行 `sret` 将控制流返回到用户态。整体流程如图 6-1 所示。

```rust
pub struct TrapContext {
    /*  0 */ pub user_x: [usize; 32],
    /* 32 */ pub fctx: FloatContext,
    /* 65 */ pub sstatus: Sstatus,
    /* 66 */ pub sepc: usize,
    /* 67 */ pub kernel_tp: usize,
    /* 68 */ pub kernel_fp: usize,
    /* 69 */ pub kernel_sp: usize,
    /* 70 */ pub kernel_ra: usize,
    /* 71 */ pub kernel_s: [usize; 12],
}
```

代码 6-1　中断上下文

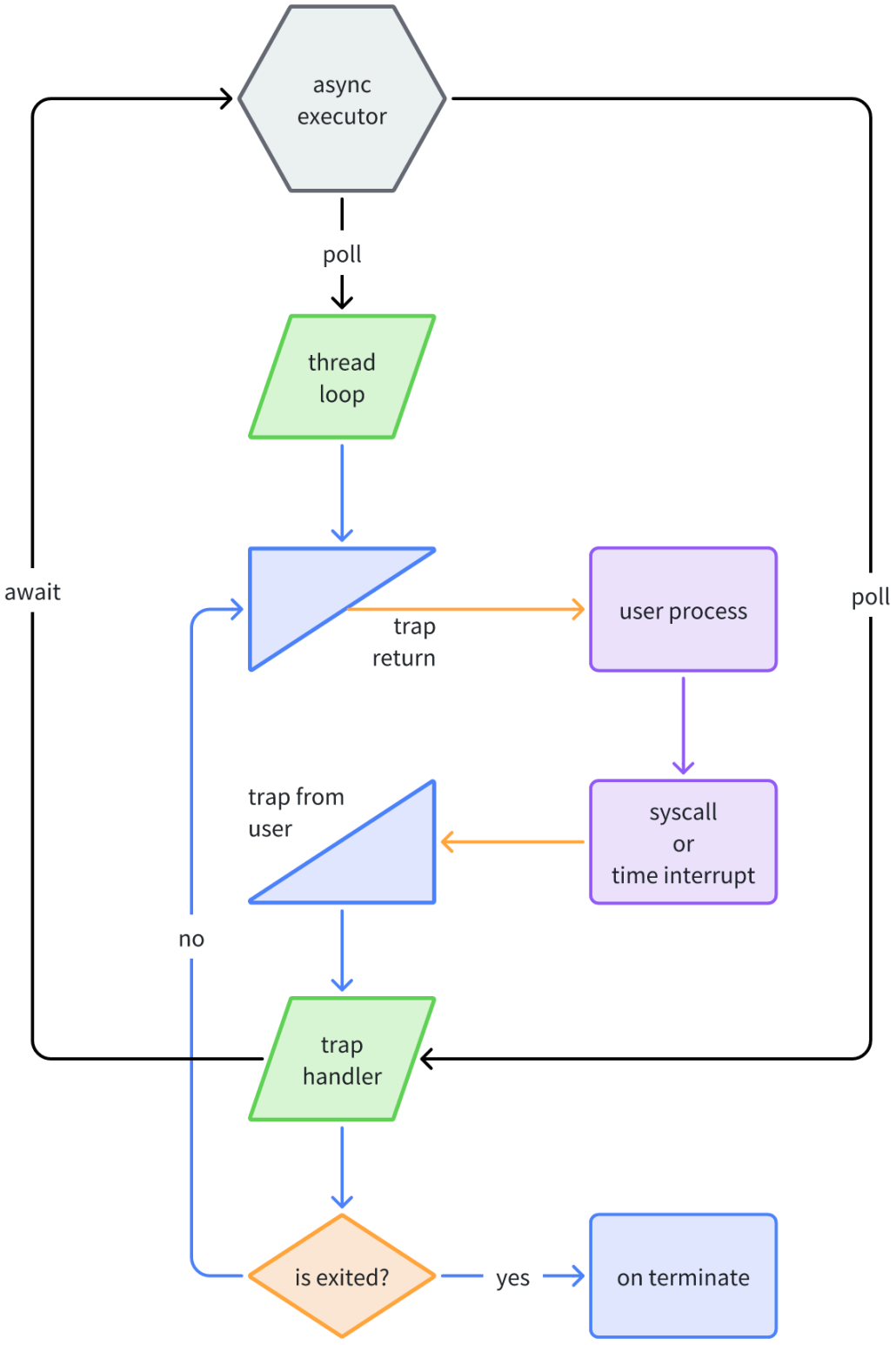

图 6-1　用户态中断处理流程

6.2 中断处理

6.2.1 内核态中断处理

内核中断分为三类，分别是时钟中断、外部中断和缺页异常，如代码 6-2 所示。

```
fn trap_from_kernel() -> bool {
    local_hart().on_kintr = true;
    let stval = stval::read();
    let sepc = sepc::read();
    let trap = scause::read().cause();
    match trap {
        Trap::Exception(Exception::LoadFault)
        | Trap::Exception(Exception::LoadPageFault) => {
            local_hart().last_page_fault =
              handle_page_fault(VirtAddr(stval), ASPerms::R);
        }
        Trap::Exception(Exception::StoreFault)
        | Trap::Exception(Exception::StorePageFault) => {
            local_hart().last_page_fault =
              handle_page_fault(VirtAddr(stval), ASPerms::W);
        }
        Trap::Exception(Exception::InstructionFault)
        | Trap::Exception(Exception::InstructionPageFault) => {
            local_hart().last_page_fault =
              handle_page_fault(VirtAddr(stval), ASPerms::X);
        }
        Trap::Interrupt(Interrupt::SupervisorTimer) => {
            local_hart().ctx.timer_during_sys += 1;
            query_timer();
            set_next_trigger();
        }
        Trap::Interrupt(Interrupt::SupervisorExternal) => {
            BOARD_INFO.plic.handle_irq(local_hart().id);
        }
        _ => {
            panic!("Fatal");
        }
    }
    local_hart().on_kintr = false;
    local_hart().on_page_test
}
```

代码 6-2　内核态中断处理

为了支持缺页异常以外的内核中断，且避免出现嵌套内核中断，我们设计了中断保护机制：`kintr_rec` 记录了当前中断保护的层数，`on_kintr` 标志记录了当前是否在内核中断处理中。当且仅当 `kintr_rec` 为 0 且 `on_kintr` 为 `false` 时，才会启用内核中断。当内核中断发生时，CPU 会自动关闭中断；在处理函数中，`on_kintr`

被设置为 `true`，从而避免出现嵌套。在内核中断外的临界区中，通过 RAII 风格的保护结构 `KIntrGuard` 来控制 `kintr_rec` 的增减，从而保证中断保护的正确性。

```rust
pub fn enable_kintr(&mut self) {
    self.kintr_rec -= 1;
    if self.kintr_rec == 0 && !self.on_kintr {
        arch::enable_kernel_interrupt();
    }
}

pub fn disable_kintr(&mut self) {
    if self.kintr_rec == 0 {
        arch::disable_kernel_interrupt();
    }
    self.kintr_rec += 1;
}

/// RAII 风格守护结构
pub struct KIntrGuard;
```

代码 6-3　中断保护机制

内核缺页异常通常发生在系统调用，内核向用户传入的地址写入数据时，触发写时复制机制。如果异常处理失败，则根据失败类型做不同处理。如果失败原因是内存不足，则终止当前进程；如果失败原因是非法访问，则发送 `SIGSEGV` 信号。处理函数如代码 6-4 所示。

```rust
fn handle_page_fault(addr: VirtAddr, perform: ASPerms) {
    debug!("Kernel page fault at {:?} for {:?}", addr, perform);
    let thread = local_hart()
        .current_thread()
        .expect("Page fault while running kernel thread");
    if thread.process.inner.is_locked() == Some(local_hart().id) {
        warn!("Page fault while holding process lock");
    }
    let mut proc_inner = thread.process.inner.lock();
    match proc_inner.addr_space.handle_page_fault(addr, perform) {
        Ok(()) => debug!("Page fault resolved"),
        Err(Errno::ENOSPC) => {
            error!("Fatal page fault: Out of memory, kill process");
            current_process().terminate(-1);
        }
        Err(e) => {
            error!("Page fault failed: {:?}, send SIGSEGV", e);
            current_thread().signals.recv_signal(Signal::SIGSEGV);
        }
    }
}
```

代码 6-4　内核缺页异常处理

外部中断的触发是由外部设备产生的，例如串口接收到数据、磁盘读写完成等。外部中断通过 PLIC 进行管理。PLIC 是 RISC-V 的外部中断控制器，用于管理外部中断的优先级和掩码。PLIC 的 MMIO 地址在设备树中定义。MinotaurOS 在解析完设备树后，初始化 PLIC。

6.2.2 用户态中断处理

用户态中断分为三类，分别是系统调用、缺页异常和时钟中断。MinotaurOS 的用户态中断的处理程序如代码 6-5 所示。对于系统调用，会先将中断上下文中的 `sepc` 加 4，使得调用完成后能够跳转到下一条指令。然后，调用 `syscall` 函数执行系统调用。系统调用完成后，将返回值写入 `x10` 寄存器。对于缺页异常，处理流程与内核态中断相同。对于时钟中断，会设置下一次触发时间，然后调用 `yield_now` 函数切换到下一个线程。对于未识别的用户态中断，MinotaurOS 不会触发崩溃，但会终止当前进程。

用户态中断处理程序是一个异步函数，其执行过程可能不会一次性完成。协程调度器会在等待异步任务完成或使用 `yield_now` 主动让出时间片时切换到其他协程执行。这样，MinotaurOS 能够实现多任务并发。

```rust
pub async fn trap_from_user() {
    set_kernel_trap_entry();
    let stval = stval::read();
    let sepc = sepc::read();
    let trap = scause::read().cause();
    trace!(
        "Trap {:?} from user at {:#x} for {:#x}",
        trap, sepc, stval,
    );
    match trap {
        Trap::Exception(Exception::UserEnvCall) => {
            let ctx = current_trap_ctx();
            // syscall 完成后，需要跳转到下一条指令
            ctx.sepc += 4;
            let result = syscall(
                ctx.user_x[17],
                ctx.user_x[10..=15],
            ).await;
            ctx.user_x[10] = result
                .unwrap_or_else(|err| -(err as isize) as usize)
        }
        | Trap::Exception(Exception::LoadFault)
        | Trap::Exception(Exception::LoadPageFault) => {
            handle_page_fault(VirtAddr(stval), ASPerms::R);
        }
        Trap::Exception(Exception::StoreFault)
        | Trap::Exception(Exception::StorePageFault) => {
            handle_page_fault(VirtAddr(stval), ASPerms::W);
        }
        Trap::Exception(Exception::InstructionFault)
        | Trap::Exception(Exception::InstructionPageFault) => {
            handle_page_fault(VirtAddr(sepc), ASPerms::X);
        }
        Trap::Interrupt(Interrupt::SupervisorTimer) => {
            set_next_trigger();
            yield_now().await;
        }
        _ => {
            error!("Unhandled trap: {:?}", trap);
            current_thread().terminate(-1);
        }
    }
}
```

代码 6-5　用户态中断处理

第 7 章　信号系统

7.1　事件总线

MinotaurOS 在每个线程中设置了一个事件总线 `EventBus`，用于处理异步事件。事件总线是一个异步回调机制，保存了当前发生的异步事件和对应的唤醒器。事件的定义如代码 7-1 所示。

```
pub struct Event: u32 {
    const CHILD_EXIT = 1 << 0;
    const KILL_PROCESS = 1 << 2;
    const COMMON_SIGNAL = 1 << 3;
}
```

代码 7-1　Event 定义

事件总线提供等待事件、发送事件和注册回调的接口。等待事件会创建 `WaitEventFuture` 将当前协程挂起，直到事件发生。发送事件会触发所有先前注册的等待该事件的唤醒器，实现回调。

```
impl Future for WaitForEventFuture<'_> {
    type Output = Event;

    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>)
        -> Poll<Self::Output> {
        let mut inner = self.event_bus.0.lock();
        let happened = inner.event.intersection(self.event);
        if happened != Event::empty() {
            inner.event.remove(happened);
            Poll::Ready(happened)
        } else {
            let mut event = self.event;
            inner.callbacks.retain(|(e, waker)| {
                if waker.will_wake(cx.waker()) {
                    event |= *e;
                    false
                } else {
                    true
                }
            });
            inner.callbacks.push((event, cx.waker().clone()));
            Poll::Pending
        }
    }
}
```

代码 7-2　WaitEventFuture

7.2 信号管理

信号是 UNIX 系统中的一种进程间通信机制，用于通知进程发生了某种事件。MinotaurOS 实现了类似 Linux 的信号系统，支持常见的信号，如 `SIGKILL`、`SIGSTOP`、`SIGCHLD` 等。MinotaurOS 使用如代码 7-3 所示的 `SignalController` 结构管理信号。信号管理器内部包含信号等待队列、信号屏蔽集和信号处理程序。每个线程都有一个独立的信号控制器。

```
pub struct SignalController(Mutex<SignalControllerInner>);

struct SignalControllerInner {
    pending: SignalQueue,
    blocked: SigSet,
    handlers: [SignalHandler; SIG_MAX],
}
```

代码 7-3　SignalController 结构

线程结构提供了 `recv_signal` 方法来向发送一个信号。对于 `SIGCHLD` 信号和 `SIGKILL` 信号，会始终向事件总线发送对应事件。其余信号会根据屏蔽集决定是否向事件总线发送事件。之后，信号会被压入信号队列，等待线程处理。

```
pub fn recv_signal(&self, signal: Signal) {
    info!("Thread {} receive signal {:?}", self.tid.0, signal);
    match signal {
        Signal::SIGCHLD =>
            self.event_bus.recv_event(Event::CHILD_EXIT),
        Signal::SIGKILL =>
            self.event_bus.recv_event(Event::KILL_PROCESS),
        _ => {
            if !self.signals.get_mask().contains(signal.into()) {
                self.event_bus.recv_event(Event::COMMON_SIGNAL);
            }
        }
    }
    self.signals.push(signal);
}
```

代码 7-4　recv_signal 方法

信号队列提供 `poll` 方法（如代码 7-5 所示），用于从信号队列中取出一个未被屏蔽的信号。用户定义的信号处理器中有一个值 `sa_mask`，用来设置信号处理器的屏蔽集。当信号处理器被调用时，内核需要将当前线程的屏蔽集暂时设置为 `sa_mask`，以防止信号处理器被信号中断。因此，每次查询信号时，需要将查询前

的屏蔽集保存下来，以便在信号处理器执行完毕后恢复，同时将当前的屏蔽集替换为 `sa_mask`。

```rust
pub struct SignalPoll {
    pub signal: Signal,
    pub handler: SignalHandler,
    pub blocked_before: SigSet,
}

pub fn poll(&self) -> Option<SignalPoll> {
    let mut inner = self.0.lock();
    let mut popped = vec![];
    while let Some(signal) = inner.pending.pop() {
        if inner.blocked.contains(signal.into()) {
            debug!("Signal {:?} is blocked", signal);
            popped.push(signal);
            continue;
        }
        let blocked_before = inner.blocked;
        let handler = inner.handlers[signal as usize];
        if let SignalHandler::User(sig_action) = &handler {
            inner.blocked.insert(signal.into());
            inner.blocked |= sig_action.sa_mask;
        }

        inner.pending.extend(popped);
        let poll = SignalPoll { signal, handler, blocked_before };
        return Some(poll);
    }

    inner.pending.extend(popped);
    return None;
}
```

代码 7-5　查询信号

线程循环会在处理完用户态中断后，查询信号队列，如果有未被屏蔽的信号，则会调用默认信号处理器或将用户态的控制流切换到用户定义的信号处理器中（如果有）。

用户定义的信号处理器切换过程分为三步。首先，将当前线程的用户态寄存器保存到用户栈上；然后，将 `epc` 寄存器设置为信号处理器的入口地址，将返回地址设置为默认跳板或 `sa_restorer` 指定的地址，并将 `a0` 和 `a1` 分别设置为发生的信号和寄存器信息地址（在这里是用户栈），下次线程返回用户态时，会从用户定义的信号处理器开始执行。最后，信号处理器函数会返回到跳板或 `sa_restorer` 指定的地址，再通过 `sigreturn` 系统调用恢复信号发生前的用户态寄存器，切换回之前的控制流。

第 8 章　网络系统

8.1　网络协议栈

MinotaurOS 使用的网络协议栈是 smoltcp，能够处理 IPv4 和 IPv6 两类 IP 地址。smoltcp 提供了基本的网络硬件设备和 Socket 抽象，在此基础上，MinotaurOS 实现了网络设备，提供网络设备调用接口。网络接口的定义如代码 8-1 所示。

```rust
pub struct NetInterface {
    pub iface: Interface,
    pub device: Loopback,
    pub sockets: SocketSet<'static>,
    pub port_cx: PortContext,
}
```

代码 8-1　NetInterface 定义

通过调用接口的方法，可以实现收发数据包的功能。

值得一提的是，在 smoltcp 中，没有实现 Udp 一对多发送的功能，所以 MinotaurOS 对 smoltcp 进行修改，使其在适配内核的基础上实现 Udp 一对多发送的功能。

8.2　本地网络设备

MinotaurOS 将网络设备分为两类，一种是本地网络设备，另外一类是虚拟网络设备，之后可能会对于实际的开发板实现具体的物理网络设备。

基于 smoltcp 的 Loopback 定义，MinotaurOS 实现了本地网络的接口，如代码 8-1 所示。

8.3　虚拟网络设备

MinotaurOS 基于 virtio-drivers，实现了虚拟网络设备。可以用于在 QEMU 模拟器上模拟网络。虚拟网络设备定义如代码 8-2 所示。

```
pub struct VirtIONetDevice {
    metadata: DeviceMeta,
    base_addr: VirtAddr,
    dev: LateInit<Arc<Mutex<Net>>>,
}
```

代码 8-2　VirtIONetDevice 定义

8.4　Socket 定义

MinotaurOS 实现网络系统调用的关键是 Socket trait 的定义和实现，通过 Socket，MinotaurOS 抽象出了系统 Socket 具体的功能。Socket trait 如代码 8-3 所示。

```rust
pub trait Socket: File {
    fn bind(&self, addr: SockAddr) -> SyscallResult;

    async fn connect(&self, addr: SockAddr) -> SyscallResult;

    fn listen(&self) -> SyscallResult;

    async fn accept(&self,
          addr: Option<&mut SockAddr>)
    -> SyscallResult<Arc<dyn Socket>>;

    fn set_send_buf_size(&self, size: usize) -> SyscallResult;

    fn set_recv_buf_size(&self, size: usize) -> SyscallResult;

    fn dis_connect(&self, how: u32) -> SyscallResult;

    fn socket_type(&self) -> SocketType;

    fn sock_name(&self) -> SockAddr;

    fn peer_name(&self) -> Option<SockAddr>;

    fn shutdown(&self, how: u32) -> SyscallResult;

    fn recv_buf_size(&self) -> SyscallResult<usize>;

    fn send_buf_size(&self) -> SyscallResult<usize>;

    fn set_keep_alive(&self, enabled: bool) -> SyscallResult;

    fn set_nagle_enabled(&self, enabled: bool) -> SyscallResult;

    async fn recv(&self,
          buf: &mut [u8],
          flags: RecvFromFlags,
          src: Option<&mut SockAddr>,)
    -> SyscallResult<isize>;

    async fn send(&self,
          buf: &[u8],
          flags: RecvFromFlags,
          dest: Option<SockAddr>,)
    -> SyscallResult<isize>;
}
```

代码 8-3　Socket trait 定义

8.5　与文件系统连接

基于 Socket trait 对于 File trait 的限定，一个 Socket 同时是一个文件，每当 MinotaurOS 创建一个 Socket，都会创建一个对应的逻辑上的文件，并返回文件句柄，可以通过这个句柄操作这个 Socket。

由于每个进程都有自己的文件描述符表，所以 MinotaurOS 将每个 Socket 看作一个文件，文件描述符和 Socket 的对应关系存储在进程的 fd_table 中。

Socket 与 File 的关系密不可分，当我们需要将一个 File 转换为 Socket 时，调用如代码 8-4 所示的函数。具体的 Socket 对于这个函数有自己的实现，而其他类型的文件则不会理会。

```
pub trait File: Send + Sync {

    fn as_socket(self: Arc<Self>) -> SyscallResult<Arc<dyn Socket>> {
        Err(Errno::ENOTSOCK)
    }

}
```

代码 8-4　Socket 转化函数

8.6　实现 Tcp 和 Udp

对于具体实现 Socket 的内容，TcpSocket 和 UdpSocket 结构体是 MinotaurOS 对 Socket 的具体操作的对象。这两个结构体内部并不会直接存储 Socket 的实现细节，那是 smoltcp 的工作，而是存储 Socket 的状态信息，供 MinotaurOS 在调用时查找，并决定后续的行为。

TcpSocket 和 UdpSocket 如代码 8-5 所示。

```rust
pub struct TcpSocket {
    metadata: FileMeta,
    inner: Mutex<TcpInner>,
}

struct TcpInner {
    handle: SocketHandle,
    local_endpoint: Option<IpEndpoint>,
    remote_endpoint: Option<IpEndpoint>,
    last_state: tcp::State,
    recv_buf_size: usize,
    send_buf_size: usize,
}

pub struct UdpSocket {
    metadata: FileMeta,
    handle: SocketHandle,
    inner: Mutex<UdpInner>,
}

struct UdpInner {
    local_endpoint: Option<IpEndpoint>,
    remote_endpoint: Option<IpEndpoint>,
    recvbuf_size: usize,
    sendbuf_size: usize,
}
```

代码 8-5　TcpSocket 和 UdpSocket 结构体

MinotaurOS 为这两个结构实现 Socket 和 File 的 trait，从而满足网络系统调用的需求。

第 9 章　总结和展望

9.1　亮点与创新

1. 内存管理：TLB 是用于加速虚拟地址到物理地址转换的硬件缓存。当 CPU 需要访问内存时，首先在 TLB 中查找对应的虚拟地址。如果找到，则直接使用 TLB 中缓存的物理地址，避免了访问页表的开销。如果未命中，则需要访问页表以进行地址转换。找到物理地址后，将该映射缓存到 TLB 中，以便后续访问更快。MinotaurOS 充分利用了 RISC-V 的 ASID 机制，对 TLB 刷新做了优化，减少了刷新带来的开销，提高了效率。
2. 进程管理：MinotaurOS 通过协程执行器管理用户线程和内核线程，实现了用户线程和内核线程的统一调度。
3. 浮点寄存器优化：使用浮点寄存器状态机，精细化管理浮点寄存器的保存和恢复操作。只有在浮点寄存器被修改后才标记为脏，只有在必要时才进行保存和恢复操作。这样可以减少不必要的浮点寄存器操作，降低开销。通过减少浮点寄存器操作的频率，降低了系统的开销，提高了系统的响应速度和整体性能。在处理多线程、多进程以及频繁的用户态和内核态切换时，能够显著提高响应速度和稳定性。

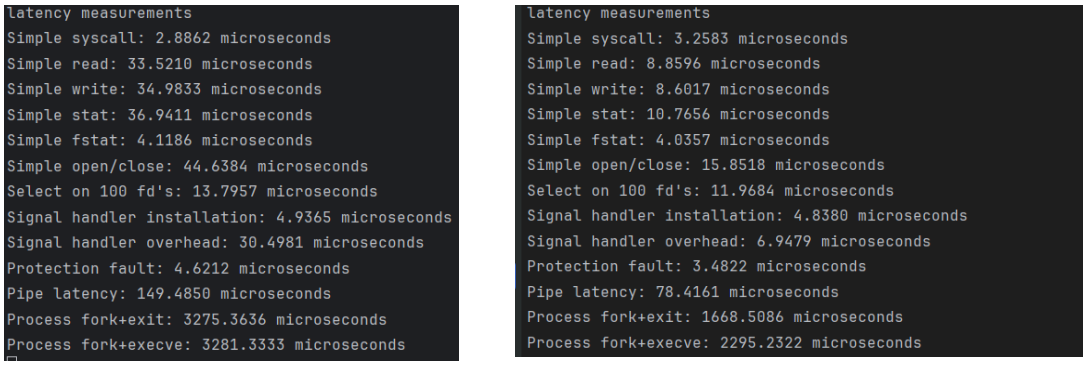

图 9-1　性能优化前后对比

9.2　问题与挑战

目前 MinotaurOS 还存在一些复杂的挑战，例如需要适配 LTP 测试用例、支持更多的 Linux 系统调用、优化文件系统等。在未来的工作中，我们将继续努力，解决这些问题，提高 MinotaurOS 的性能和稳定性。