哈爾濱工業大學(深圳)

HARBIN INSTITUTE OF TECHNOLOGY, SHENZHEN

Del0n1x

设计文档

参赛队名 Del0n1x

队伍成员 姚俊杰、卢家鼎、林顺喆

指导老师 夏文、仇洁婷

2025 年 6 月

目录

第 1 章　概述

1.1　项目介绍

Del0n1x 是一个使用 Rust 语言编写的同时适配 RISC-V64 和 LoongArch64 的宏内核，支持多核运行与无栈协程进程调度，拥有完善的内存管理和信号传递机制。在软硬件交换层面，我们实现了自己的 HAL 层， 统一调用接口，能够同时支持 RISC-V64 和 LoongArch64 指令架构。

Del0n1x 致力于实现高效清晰的代码逻辑，遵守 System Manual 手册，实现 105 个相关系统调用，并且对于其中大部分系统调用做了相对完善的错误检查和处理机制，这为我们后来适配 ltp 带来了便利。

Del0n1x 初赛阶段的内核主要模块和完成情况如下表格：

模块	**完成情况**
HAL 模块	实现自己的 HAL 代码库，支持 RISC-V64 和 LoongArch64 双架构
进程管理	无栈协程调度，支持全局统一的 executor 调度器；实现多线程的资源回收； 统一进程和线程的数据结构
文件系统	实现 dentry 构建目录树； 实现页缓存和 dentry 缓存加快读写
内存管理	实现基本的内存管理功能； 实现 CoW、懒分配内存优化
时钟模块	实现时间轮混合最小堆的数据结构管理方式； 支持定时器唤醒机制
IPC 系统	支持处理用户自定义信号和 sigreturn 机制；实现支持读者写者同步的管道机制； 支持 System V 共享内存
网络模块	初步完成网络模块相关代码，由于时间原因还没有适配通过网络测例

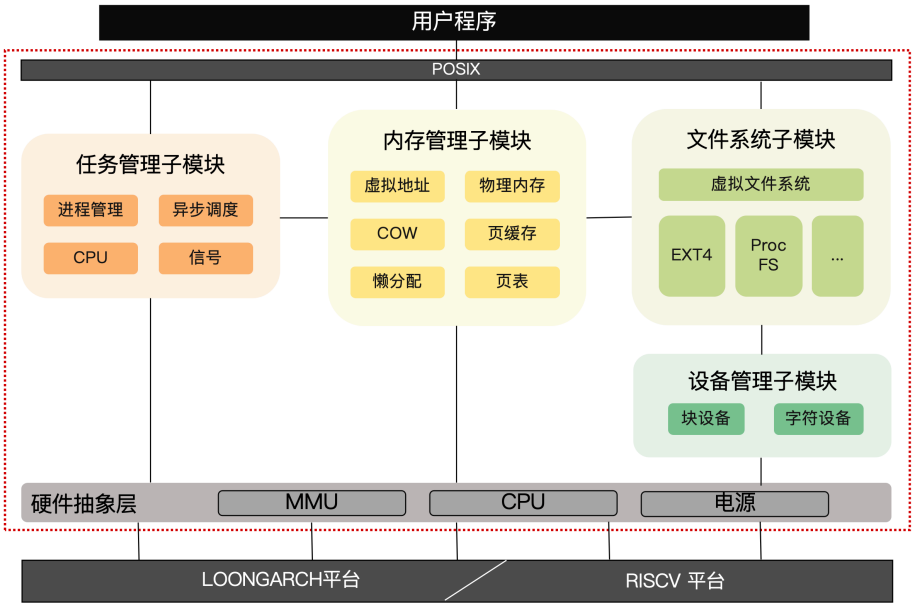

图 1-1　Del0n1x 整体架构图

整个项目的代码结构如下：

```
os
├── linker                  # 程序链接脚本
├── src
│   ├── arch                # 架构相关的汇编
│   ├── driver              # 块设备驱动
│   ├── fuse                # 文件系统
│   ├── hal                 # 架构相关代码
│   ├── ipc                 # 进程间通信相关的部分代码
│   ├── mm                  # 内存页表
│   ├── net                 # 网络模块
│   ├── signal              # 信号模块
│   ├── task                # 任务控制块，任务调度
│   ├── utils               # 一些工具
│   ├── sync                # 同步相关
│   ├── syscall             # 系统调用
│   ├── entry_la.asm        # 龙芯入口初始化汇编函数
│   ├── entry.asm           # riscv 入口初始化函数
│   ├── console.rs
│   ├── lang_items.rs
```

```
│   ├── Makefile
│   └── main.rs
user                        # 用户程序
├── src
│   ├── bin
│   │   ├── autorun.rs      # 自动测试
│   │   ├── gbshell.rs      # glibc 的 busybox shell
│   │   ├── huge_write.rs   # 测试文件系统写入速度
│   │   ├── initproc.rs     # 调用 user_shell，进入自己实现的终端
│   │   ├── mbshell.rs      # musl 的 busybox shell
│   │   └── user_shell.rs
vendor                      # 第三方依赖
report                      # 文档
bootloader                  # 引导加载程序
```

第 2 章　进程管理

2.1　概述

Del0n1x 操作系统采用无栈协程作为核心任务管理模型，该设计基于用户级轻量级线程理念，允许任务在执行过程中挂起并恢复。无栈协程与有栈协程的核心差异体现在上下文管理机制上：在 rcore 和 xv6 中，采用有栈方式进行任务切换，每次任务切换需要调用 `__switch` 函数，每个任务有独立栈空间 并在切换时保存完整栈帧及 `s0-s11`、`ra`、`sp` 等 14 个 RISC-V 寄存器， 而 Del0n1x 的无栈协程不依赖独立栈结构，转而通过状态机管理上下文状态，任务挂起时仅需保存当前执行位置和关键状态变量，显著降低内存开销与切换延迟。

在传统操作系统中，进程调度需切换用户级上下文、寄存器上下文和系统级上下文（含内核栈与页表），这种完整上下文切换必须通过内核态系统调用完成。 但是 Del0n1x 采用共享内核栈架构，所有任务复用同一内核栈空间，任务切换时不涉及内核栈切换，这极大地降低了任务切换的开销。

2.2　任务调度

在无栈协程中，任务调度并不需要在栈上保存任务栈帧，而是将必要的中间信息保存为状态机，放置在堆空间进行保存，这样不仅减少了任务调度开销，同时 提高了调度过程中的安全性，降低栈溢出的风险。

在 Del0n1x 中， 当对异步函数调用 `.await` 方法时， async-task 库会首次调用 Future 的 poll 方法，如果 poll 返回的结果是 Pending，那么该任务将被挂起， await 将控制权交给调度器，以便另一个任务可以继续进行。任务调度如下图：

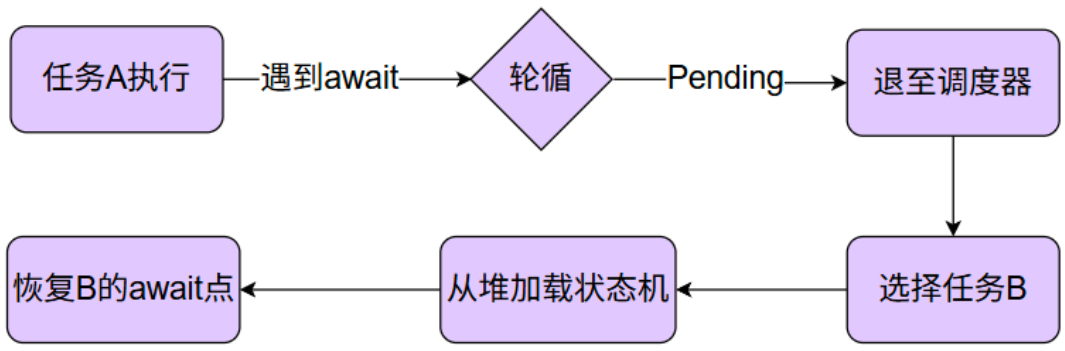

图 2-1　任务调度

2.3 任务调度队列与执行器

在任务调度队列实现中，Del0n1x 将调度队列分为 FIFO 和 PRIO 队列，感谢优秀作品 Plntry 对 async-task 库做出的 Pr， 使得在任务调度过程中可以获取调度信息，通过调度信息，我们可以对任务做出更加精细的控制。如果任务在运行时被唤醒，则将其加入 FIFO 队列，其他的就放入 PRIO 队列进行管理。

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

....
// 任务入队逻辑
if info.woken_while_running {
    queue.push_normal(runnable);
} else {
    queue.push_prior(runnable);
}

....
```

代码 2-1　任务队列结构

在 Del0n1x 中，我们使用统一的 `TaskFuture` 封装了任务。

```
pub enum TaskFuture<F: Future<Output = ()> + Send + 'static> {
    UserTaskFuture {
        task: Arc<TaskControlBlock>,
        future: F,
    },
    KernelTaskFuture {
        future: F,
    },
}
```

代码 2-2　任务 Future 枚举

```rust
fn poll(
    self: Pin<&mut Self>,
    cx: &mut core::task::Context<'_>,
) -> core::task::Poll<Self::Output> {
    let this = unsafe { self.get_unchecked_mut() };

    match this {
        TaskFuture::UserTaskFuture { task, future } => {
            let processor = get_current_cpu();
            processor.user_task_checkin(task);  // 用户任务 checkin
           let ret = unsafe { Pin::new_unchecked(future).poll(cx) };
            processor.user_task_checkout(task); // 用户任务 checkout
            ret
        }
        TaskFuture::KernelTaskFuture { future } => {
            unsafe { Pin::new_unchecked(future).poll(cx) }
        }
    }
}
```

代码 2-3　任务 Future poll 实现

对于用户任务，在 poll 轮循中实现了任务的切换调度。当任务 checkin 时，需要在修改 TCB 的时间戳记录调度时间，然后切换 CPU 中运行任务和切换页表。当任务 checkout 时，需要判断浮点寄存器状态是否为 dirty 以确定是否保存浮点寄存器，然后清空 CPU 当前任务，并记录任务 checkout 时间。 对于内核任务，Del0n1x 并没有设计任务切换，而是让该任务一直 poll，直到任务结束，这类任务主要是 shell 程序。

spawn_user_task 可以设置一个用户任务。Del0n1x 将用户任务的 future 设置为 trap_loop 循环，负责处理任务在用户态和内核态之间的切换，直到任务结束。执行 `executor::spawn(future)` 将任务挂入全局队列中等待被调度。

```rust
pub fn spawn_user_task(user_task: Arc<TaskControlBlock>) {
    let future = TaskFuture::user_task(
        user_task.clone(),
        trap_loop(user_task)
    );
    executor::spawn(future);
}
```

代码 2-4　用户任务生成函数

2.4　多核心 CPU 管理

在 Del0n1x 中，我们将处理器抽象为 CPU，使用内核中 CPU 结构体进行统一管理。current 中保存当前正在运行的任务的 TCB；timer_irq_cnt 记录内核时钟中断

次数，在内核时钟中断处理函数中会增加这个计数器，trap return 时会清零，如果计数器大于阈值，手动对该任务 yield 进行调度，避免任务一直占用 CPU；kernel_trap_ret_value 用于记录 pagafault 返回值。

```
pub struct CPU {
    current: Option<Arc<TaskControlBlock>>,
    timer_irq_cnt: usize,
    hart_id: usize,
    kernel_trap_ret_value: Option<SysResult<()>>,
}
```

代码 2-5　CPU 结构体定义

单个 CPU 被存放在全局的 PROCESSORS 管理器中，并向外暴露接口，通过管理器我们能获取到当前任务的上下文信息和页表 token、CPU id 号等。

```
const PROCESSOR: CPU = CPU::new();
pub static PROCESSORS: SyncProcessors =
    SyncProcessors(UnsafeCell::new([PROCESSOR; HART_NUM]));
```

代码 2-6　全局 CPU 管理器

2.5　任务控制块

进程是操作系统中资源管理的基本单位，而线程是操作系统中调度的基本单位。由于在 Linux 设计理念中，线程是轻量级进程，所以在 Del0n1x 中使用统一的任务控制块（Task Control Block，TCB）来管理进程和线程。同时我们对 TCB 字段进行细粒化的加锁处理，类似 `memory_space` 和 `trap_cx` 等高频访问的字段来说，可以显著减少并发过程中锁的竞争，提高并发效率。

```rust
pub struct TaskControlBlock {
    pub pid: Pid,                   // 任务标识符
    pub tgid: AtomicUsize,          // leader 的 pid 号
    pub pgid: AtomicUsize,          // 进程组 id
    pub task_status: SpinNoIrqLock<TaskStatus>,// 任务状态
    pub thread_group: Shared<ThreadGroup>,      // 线程组
    pub memory_space: Shared<MemorySpace>,      // 地址空间
    pub fd_table: Shared<FdTable>,              // 文件描述表
    pub current_path: Shared<String>,           // 路径
    pub robust_list: Shared<RobustList>,        // 存储线程的信息
    pub futex_list: Shared<FutexBucket>,        // futex 互斥锁队列
    pub itimers: Shared<[ITimerVal; 3]>,        // 任务的内部时钟
    pub fsz_limit: Shared<Option<RLimit64>>,    // 任务的资源限制
    pub shmid_table: Shared<ShmidTable>,        // sysv 进程共享内存表
    pub pending: AtomicBool,                    // 是否有信号待处理
    pub ucontext: AtomicUsize,                  // 信号用户态指针
    pub sig_pending: SpinNoIrqLock<SigPending>,// 信号列表
    pub blocked: SyncUnsafeCell<SigMask>,       // 任务阻塞信号
    pub handler: Shared<SigStruct>,             // 信号处理集合
    pub sig_stack: SyncUnsafeCell<Option<SignalStack>>, // 信号栈
    pub waker: SyncUnsafeCell<Option<Waker>>,   // 任务唤醒句柄
    pub trap_cx: SyncUnsafeCell<TrapContext>,   // 上下文
    pub time_data: SyncUnsafeCell<TimeData>,    // 时间戳
    pub cpuset: SyncUnsafeCell<CpuSet>,         // CPU 亲和性掩码
    pub prio: SyncUnsafeCell<SchedParam>,       // 调度优先级和策略
    pub exit_code: AtomicI32,                   // 退出码
    /// CHILD_CLEARTID 清除地址
    pub clear_child_tid: SyncUnsafeCell<Option<usize>>,
    /// CHILD_SETTID 设置地址
    pub set_child_tid: SyncUnsafeCell<Option<usize>>,
    /// 父进程
    pub parent: Shared<Option<Weak<TaskControlBlock>>>,
    /// 子进程
    pub children: Shared<BTreeMap<usize, Arc<TaskControlBlock>>>,
}
```

代码 2-7　任务控制块结构体

利用 Rust Arc 引用计数和 clone 机制，可以有效的解决进程和线程之间资源共享和隔离问题。对于可以共享的资源，调用 `Arc::clone()` 仅增加引用计数（原子操作），未复制底层数据，父子进程共享同一份数据。如果是可以独立的资源，调用 clone 会递归复制整个结构，生成完全独立的数据副本，父子进程修改互不影响。

2.5.1　进程和线程联系

在 Del0n1x 中，我们使用 `ThreadGroup` 管理线程组。其中选择 `BTreeMap` 作为管理数据结构，其中 key 是 task pid，value 是 TCB 的弱引用。 线程之间需要一个 leader，而该 leader 是一个进程，同样被 `ThreadGroup` 管理。线程组的 leader 可以

用 tgid 表示；利用 tgid 可以通过 `BTreeMap` 快速定位到 leader。线程与线程之间并没有父子关系，他们同属于一个进程创建。

Type	Characteristic	Implementation
Process Leader	tgid == pid	新的内存空间，独立的信号处理程序
Thread	tgid != pid	通过 CLONE_VM 共享内存空间，通过 CLONE_SIGHAND 共享信号处理程序

进程和进程之间是树状结构，通过 `parent` 和 `children` 字段指明父进程和子进程。 Del0n1x 使用 `Manager` 管理任务和进程组，结构设计如下：

```rust
pub struct Manager {
    pub task_manager: SpinNoIrqLock<TaskManager>,
    pub process_group: SpinNoIrqLock<ProcessGroupManager>,
}
/// 存放所有任务的管理器，可以通过 pid 快速找到对应的 Task
pub struct TaskManager(pub HashMap<Pid, Weak<TaskControlBlock>>);
/// 存放进程组的管理器，通过进程组的 leader 的 pid 可以定位到进程组
pub struct ProcessGroupManager(HashMap<PGid, Vec<Pid>>);
```

代码 2-8　任务与进程组管理结构体

2.5.2　任务的状态

在 rCore 的基础上，我们为任务在运行过程中设计了 4 种状态:

- Ready: 任务已准备好执行，等待调度器分配 CPU 时间片;
- Running: 任务正在 CPU 上执行指令;
- Stopped: 任务被暂停执行，但未被终止，收到 SIGSTOP 信号
- Zombie: 任务已终止，但尚未被父进程回收

进程间状态转化如下：

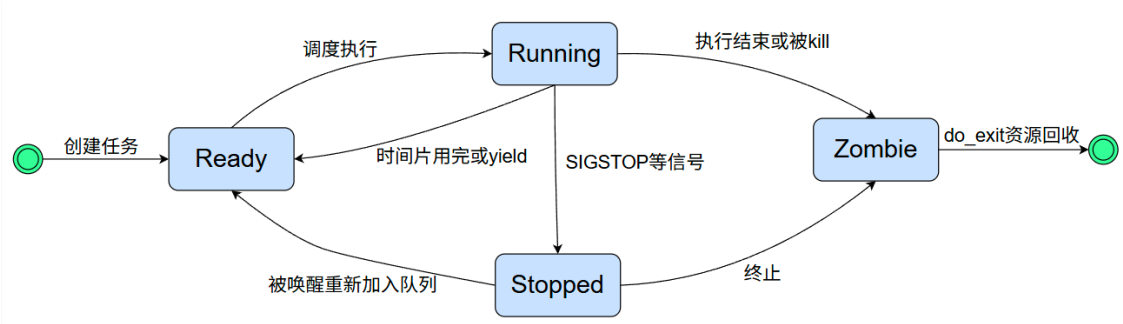

图 2-2　进程状态转化图

第 3 章　中断与异常处理

3.1　特权级切换

中断与异常是用户态与内核态之间切换的中重要机制。在 Del0n1x 中，为了更加清晰的设计模式，将中断异常分为两类：

- 用户中断异常：发生在用户态(U-Mode)的中断或异常
- 内核中断异常：发生在内核态(S-Mode)的中断或异常

当用户态发生中断或异常时，系统需要完成从用户态到内核态的切换。这个过程通过汇编函数 `__trap_from_user` 实现，这是用户态中断处理的入口点，负责保存完整的用户上下文。而 sepc（era）、stval（estat）等寄存器则是由硬件自动完成保存。用户上下文保存在如下结构中：

```
pub struct TrapContext {
    /* 0-31 */ pub user_gp: GPRegs,
    /*  32  */ pub sstatus: Sstatus,
    /*  33  */ pub sepc: usize,
    /*  34  */ pub kernel_sp: usize,
    /*  35  */ pub kernel_ra: usize,
    /* 36-47*/ pub kernel_s: [usize; 12],
    /*  48  */ pub kernel_fp: usize,
    /*  49  */ pub kernel_tp: usize,
    /*  50  */ pub float_regs: UserFloatRegs,
}
```

代码 3-1　TrapContext 结构

需要注意的是，在 RISC-V64 架构下，我们通过检测 fs 寄存器是否被使用决定是否保存浮点寄存器，这样的设计减少上下文切换的开销。但是通过查阅手册，LoongArch64 架构中并没有提供这样的寄存器，所以我们将 LoongArch64 中浮点寄存器的保存处理放置在 `__trap_from_user`，同时我们在 TrapContext 中增加了 `fcsr` 字段用于保存浮点控制状态寄存器。`fcsr` 是浮点运算单元（FPU）的核心控制寄存器，它负责管理浮点运算的异常标志、舍入模式、使能控制等关键功能。

3.2　处理过程

3.2.1　内核中断异常处理

以下均以 RISC-V64 为例。Del0n1x 的内核态中断异常处理目前支持时钟中断和地址缺页异常（对于外部中断放置于比赛下一阶段进行完善），代码如下：

```
Trap::Interrupt(Interrupt::SupervisorTimer) => {
    TIMER_QUEUE.handle_expired();
    get_current_cpu().timer_irq_inc();
    set_next_trigger();
}
Trap::Exception(e) => match e {
    Exception::StorePageFault
    | Exception::InstructionPageFault
    | Exception::LoadPageFault => {
        let access_type = match e {
         Exception::InstructionPageFault => PageFaultAccessType::RX,
            Exception::LoadPageFault => PageFaultAccessType::RO,
            Exception::StorePageFault => PageFaultAccessType::RW,
            _ => unreachable!(),
        };

        let task = current_task().unwrap();
        result = task
            .with_mut_memory_space(|m| {
                m.handle_page_fault(stval.into(), access_type)
            });
        result.is_err().then(|| {
            use crate::hal::arch::current_inst_len;
            sepc::write(sepc + current_inst_len());
        });
    }

set_ktrap_ret(result);
```

代码 3-2　内核中断异常处理关键函数

内核缺页异常通常发生在系统调用中，由于进程虚拟地址空间没有实际分配独立的物理页帧，而是将其指向父进程对应的物理页帧，当内核向用户传入的地址写入数据时，会触发 PageFault 异常跳转至内核处理函数中，在这次 trap 的处理中，我们不仅对子进程分配实际的物理页帧，恢复相应页表项的标志位，同时通过检查 trap 的返回值实现用户地址空间可写性的检查。

3.2.2　用户中断异常处理

在 Del0n1x 中除了对用户态的 PageFault 处理之外，还实现了时钟中断和系统调用处理。对于时钟中断，Del0n1x 检查了全局定时器中是否有超时任务，然后设置下一次时钟中断时间点，最后需要调用 yield 释放当前任务对 CPU 的使用权，调度下一个任务，避免任务长时间占用 CPU 导致其他任务饥饿。对于系统调用处理，会先将中断上下文中的 sepc 加 4，使得从内核态返回到用户态后能够跳转到下一条

指令。然后，调用 syscall 函数系统调用。系统调用完成后，将返回值保存在 x10 寄存器。

```
....
Trap::Exception(Exception::UserEnvCall) => { // 7
  let mut cx = current_trap_cx();
  let old_sepc: usize = cx.get_sepc();
  let syscall_id = cx.user_gp.a7;
  cx.set_sepc(old_sepc + 4);

  let result = syscall(
      syscall_id,
      [cx.user_gp.a0,
      cx.user_gp.a1,
      cx.user_gp.a2,
      cx.user_gp.a3,
      cx.user_gp.a4,
      cx.user_gp.a5]
  ).await;

  // cx is changed during sys_exec, so we have to call it again
  cx = current_trap_cx();

  match result {
      Ok(ret) => {
          cx.user_gp.a0 = ret as usize;
      }
      Err(err) => {
          if (err as isize) < 0 {
              cx.user_gp.a0 = err as usize;
          } else {
              cx.user_gp.a0 = (-(err as isize)) as usize;
          }
      }
  }
}

Trap::Interrupt(Interrupt::SupervisorTimer) => { // 5
    TIMER_QUEUE.handle_expired();
    set_next_trigger();
    yield_now().await;
}
....
```

代码 3-3　用户中断异常处理函数

3.2.3　返回用户态

Del0n1x 中从内核态返回到用户态过程交付予 `user_trap_return` 函数处理。在该函数的处理逻辑中，首先要通过设置 `stvec` 寄存器确保下次用户中断的入口地址正确，然后恢复用户浮点寄存器状态，同时修改进程时间戳记录进程 `trap_out` 时

间。最后通过 `__reture_to_user` 调用 `sret` 实现 S 监督模式到 U 用户模式特权级的切换。

```rust
pub fn user_trap_return() {
    // 重新修改 stvec 设置 user 的 trap handler entry
    set_trap_handler(IndertifyMode::User);

    let trap_cx = current_trap_cx();
    trap_cx.float_regs.trap_out_do_with_freg();
    trap_cx.sstatus.set_fs(FS::Clean);

    get_current_cpu().timer_irq_reset();
    let task = current_task().unwrap();
    task.get_time_data_mut().set_trap_out_time();
    unsafe {
        __return_to_user(trap_cx);
    }
    task.get_time_data_mut().set_trap_in_time();

    trap_cx.float_regs.trap_in_do_with_freg(trap_cx.sstatus);
}
```

代码 3-4　内核->用户切换

第 4 章　内存管理

4.1　地址空间总览

4.1.1　地址空间布局

Del0n1x OS 的地址空间大小为2^{39} B = 512 GB，布局如图所示：

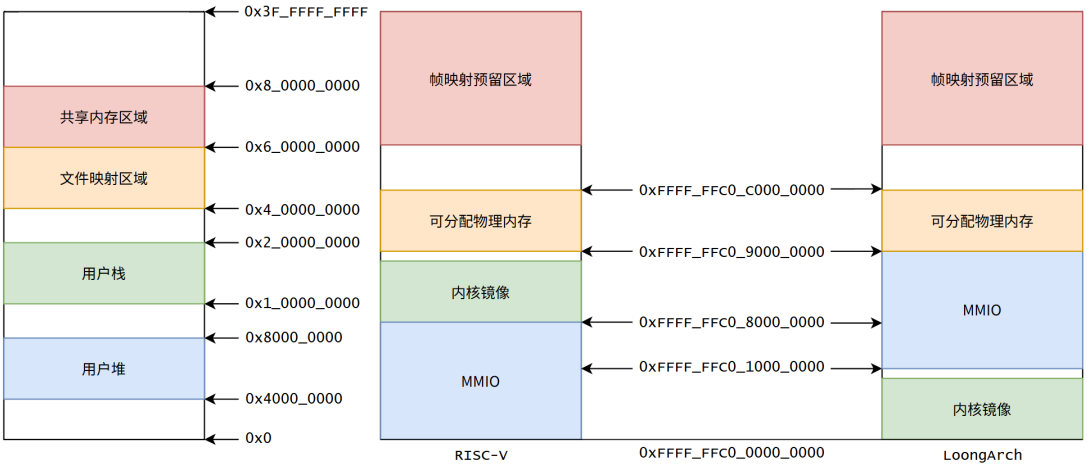

图 4-1　地址空间布局

Del0n1x 的用户程序与内核程序共用同一个地址空间。用户态程序只允许访问用户地址空间，即地址空间的低半部分。内核态程序不仅可以访问用户地址空间，还对内核地址空间，即地址空间高半部分具有访问权。

用户地址空间使用帧映射，即由内核分配对应的物理页帧加以映射，以保证高效的物理内存利用率。而内核地址空间中还存在对物理内存空间的直接映射（偏移映射），将物理地址 `pa` 映射为 `pa + KERNEL_PG_ADDR_BASE`，便于内核直接访问和操作物理内存。RISC-V 架构与 LoongArch 架构的物理内存布局有所不同，故两个架构下地址空间的布局也不尽相同。

4.1.2　地址翻译模式

LoongArch 架构支持“分页地址翻译模式”和“直接映射地址翻译模式”两种地址翻译模式。其中，分页地址翻译模式即为依赖 MMU 遍历页表的地址翻译模式，而直接映射地址翻译模式则是将落在直接映射窗口中的虚拟地址直接转换为对应的物理地址。Del0n1x 设置了 `0x8000_xxxx_xxxx_xxxx` 和 `0x9000_xxxx_xxxx_xxxx` 两

个内核态可用的直接映射窗口，分别用于设备和物理内存的直接访问，窗口内的虚拟地址将被映射到 `0x0000_xxxx_xxxx_xxxx`。

RISC-V 架构只支持分页地址翻译模式。Del0n1x 在 RISC-V 和 LoongArch 两种架构下均使用 SV39 分页翻译模式，页大小为 4096B。内核在初始化时，将设备、物理内存映射到内核地址空间的对应区域。

4.1.3 Boot 阶段的预映射

RISC-V 架构的 QEMU virt 机器上，CPU 默认不开启分页地址翻译模式，而是直接访问物理地址。但是 Del0n1x 内核链接的基地址 `0xffffffc080200000` 位于 SV39 模式下的高半段地址空间，是 Boot 初期不可达的虚拟地址。为了解决这个冲突，Del0n1x 在 Boot 阶段中创建了一个临时页表，并为内核地址空间映射了巨页，这样就可以在执行内核代码之前开启 MMU 的分页地址翻译模式，保证了内核中地址的正确性和有效性。这个临时页表将在内核初始化阶段被淘汰。

```
_start:
    ...

    # satp: 8 << 60 | boot_pagetable （开启页表机制 SV39）
    la t0, boot_pagetable
    li t1, 8 << 60
    srli t0, t0, 12
    or t0, t0, t1
    csrw satp, t0
    sfence.vma

    ...

    boot_pagetable:
    # 这是大页表
    # 里面只需要两个 pte，供我们找到正确的物理地址
    # 0x0000_0000_8000_0000 -> 0x0000_0000_8000_0000

    .quad 0
    .quad 0
    .quad (0x80000 << 10) | 0xcf # VRWXAD
    .zero 8 * 255
    .quad (0x80000 << 10) | 0xcf # VRWXAD
    .zero 8 * 253
```

代码 4-1　RISC-V entry.S 中的预映射

LoongArch 架构下，Del0n1x 内核链接的基地址为 `0x9000000000200000`，这是一个直接映射地址翻译模式下的虚拟地址。Del0n1x 只需在 Boot 阶段设置直接映射窗口，MMU 就能正确解析内核中所有地址。

```
ori         $t0, $zero, 0x1     # CSR_DMW1_PLV0
lu52i.d     $t0, $t0, -2048     # UC, PLV0, 0x8000 xxxx xxxx xxxx
csrwr       $t0, 0x180          # LOONGARCH_CSR_DMWIN0
ori         $t0, $zero, 0x11    # CSR_DMW1_MAT | CSR_DMW1_PLV0
lu52i.d     $t0, $t0, -1792     # CA, PLV0, 0x9000 xxxx xxxx xxxx
csrwr       $t0, 0x181          # LOONGARCH_CSR_DMWIN1
```

代码 4-2　LoongArch 直接映射窗口设置

4.1.4　内核地址转换

Del0n1x 在对 LoongArch 架构的兼容中同时使用了两种地址翻译模式，并且两种模式均建立了一个虚拟地址到物理地址的直接映射，这将导致存在两个虚拟地址映射到同一个物理地址。在使用两种地址翻译模式的边界上，需要对两种虚拟地址进行转换。同时，由于直接映射的存在，内核代码中需要大量地进行物理地址和虚拟地址之间的转换。为了避免在代码中插入过多的加减运算式，Del0n1x 充分发挥 Rust 语言的优势，使用如下 trait 实现内核中地址之间的转换：

```rust
// os/src/mm/address.rs

/// 直接映射地址翻译模式下的地址
pub trait Direct {
    /// 翻译为物理地址
    fn direct_pa(&self) -> PhysAddr;
    /// 转换为分页地址翻译模式下的地址
    fn paged_va(&self) -> VirtAddr;
}

/// 分页地址翻译模式下的地址
pub trait Paged {
    /// 翻译为物理地址
    fn paged_pa(&self) -> PhysAddr;
    /// 转换为直接映射地址翻译模式下的虚拟地址
    fn direct_va(&self) -> VirtAddr;
}

// os/src/hal/[ARCH]/mem/address.rs
impl Direct for VirtAddr {...}
impl Direct for PhysAddr {...}
impl Paged for VirtAddr {...}
impl Paged for PhysAddr {...}
```

代码 4-3　内核地址类型转换接口

藉此，只需要知道源地址和目标地址使用的地址翻译模式，就可以通过简单的调用方法来进行转换。在内核通过直接映射访问物理内存的场景下，这个设计统一了地址翻译和等价虚拟地址转换的接口，极大地优化了代码的可读性。

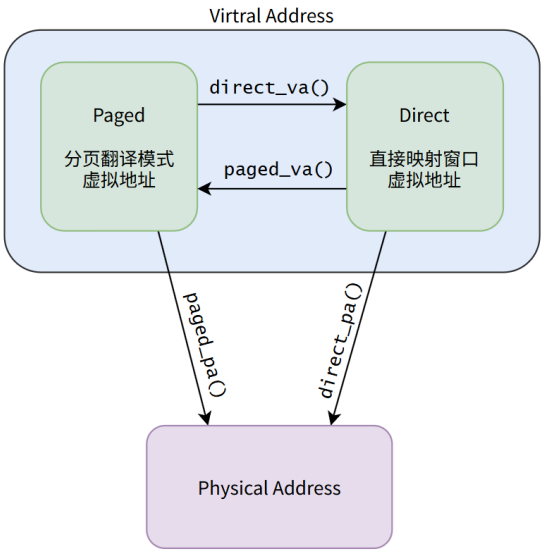

图 4-2　内核地址转换示意

由于 RISC-V 架构下只有一种地址翻译模式，也就只有一个虚拟地址到物理地址的直接映射，因此令两个 trait 的实现完全相同即可。

4.2　物理内存管理

4.2.1　物理页帧分配器

Del0n1x 的内核管理了所有可分配的物理内存。在分页地址翻译模式中，当创建一个新的帧映射时，内核需要为虚拟内存页分配对应的物理页帧；在创建文件页缓存时，内核也需要为缓存的页分配物理页帧。目前的设计下，Del0n1x 继承了 rCore 使用的 StackFrameAllocator 分配器：

```
trait FrameAllocator {
    fn new() -> Self;
    fn alloc(&mut self) -> Option<PhysPageNum>;
    fn dealloc(&mut self, ppn: PhysPageNum);
}
/// an implementation for frame allocator
pub struct StackFrameAllocator {
    current: usize,
    end: usize,
    recycled: Vec<usize>,
}
impl FrameAllocator for StackFrameAllocator {...}
```

代码 4-4　Stack Frame Allocator 分配器

该分配器使用了一种最简单的物理页帧分配策略：记录从未被分配的物理页号区间和已回收的物理页号栈，栈不为空时从栈中分配物理页号对应的页帧，栈中页号用尽时则从未分配区间的起点处分配。当 `dealloc` 方法被调用时，待回收的物理页号将被压入已回收物理页号栈，等待下一次被分配。

4.2.2 物理页帧的生命周期管理

Del0n1x 继承了 rCore 中的 `FrameTracker` 类型，使用 RAII 的思想维护已分配的物理页帧。`FrameTracker` 初始化时自动调用 `FrameAllocator::alloc` 方法分配物理页号，并将对应物理页帧清零；析构时自动调用 `FrameAllocator::dealloc` 方法，回收物理页帧。

```
impl FrameTracker {
    pub fn new() -> Option<Self> {
        FRAME_ALLOCATOR.lock().alloc().map( | ppn | {
            let bytes_array = ppn.get_bytes_array();
            for i in bytes_array {
                *i = 0;
            }
            Self { ppn }
        })
    }
}
impl Drop for FrameTracker {
    fn drop(&mut self) {
        FRAME_ALLOCATOR.lock().dealloc(self.ppn);
    }
}
```

代码 4-5　物理页帧分配器的 RAII 管理

4.3 进程内存管理

4.3.1 虚拟内存区域

在 Del0n1x 中，进程内存管理的基础单元是虚拟内存区域（Virtual Memory Area，VMA）。Del0n1x 将虚拟地址空间上映射的一段连续区域抽象为一个 VMA，这一段区域有统一的映射类型、访问权限、共享标记等属性。在内核中，进程的程序映射区域、栈、堆、一个文件映射区域、一个共享内存区域等等，分别都可以抽象为一个 VMA。Del0n1x 使用 `VmArea` 结构体管理 VMA，并通过 RAII 机制管理相关资源的释放：

```rust
pub struct VmArea {
    /// VMA 虚拟地址范围
    range_va: Range<VirtAddr>,
    /// 使用 btreemap 持有并关联物理页
    pub pages: BTreeMap<VirtPageNum, Arc<Page>>,
    /// VMA 访问权限
    pub map_perm: MapPerm,
    /// VMA 类型
    pub vma_type: VmAreaType,

    /// 文件映射：保存映射标志位
    pub mmap_flags: MmapFlags,
    /// 文件映射：持有映射的文件
    pub mmap_file: Option<Arc<dyn FileTrait>>,
    /// 文件映射：保存映射的文件偏移
    pub mmap_offset: usize,

    /// 是否共享
    pub shared: bool,
}
```

代码 4-6　虚拟内存区域结构体

4.3.2　进程地址空间

Del0n1x 使用 `MemorySpace` 结构体管理每个进程的地址空间。该结构体定义如下：

```rust
pub struct MemorySpace {
    /// 进程页表
    page_table: SyncUnsafeCell<PageTable>,
    /// 持有进程地址空间中的所有虚拟内存区域(VMA)，析构时释放这些 VmArea 结构体
    areas: SyncUnsafeCell<RangeMap<VirtAddr, VmArea>>,
}
```

代码 4-7　进程地址空间结构体

其中，`PageTable` 结构体是分页地址翻译模式下进程页表的抽象。该结构体存储页表根目录的物理页号用于地址空间切换，并持有所有目录页的 `FrameTracker` 以便在析构时回收这些物理页帧：

```rust
pub struct PageTable {
    pub root_ppn: PhysPageNum,
    pub frames: Vec<FrameTracker>,
}
```

代码 4-8　Page Table 结构体

4.4 缺页异常处理

4.4.1 进程缺页异常概述

Del0n1x 的设计中，缺页异常主要来源于以下四种情况：

- 访问了未映射或无权限的虚拟地址
- 访问了未分配的区域
- 尝试写入了写时复制（Copy on Write，CoW）的区域
- 访问了未缓存的区域

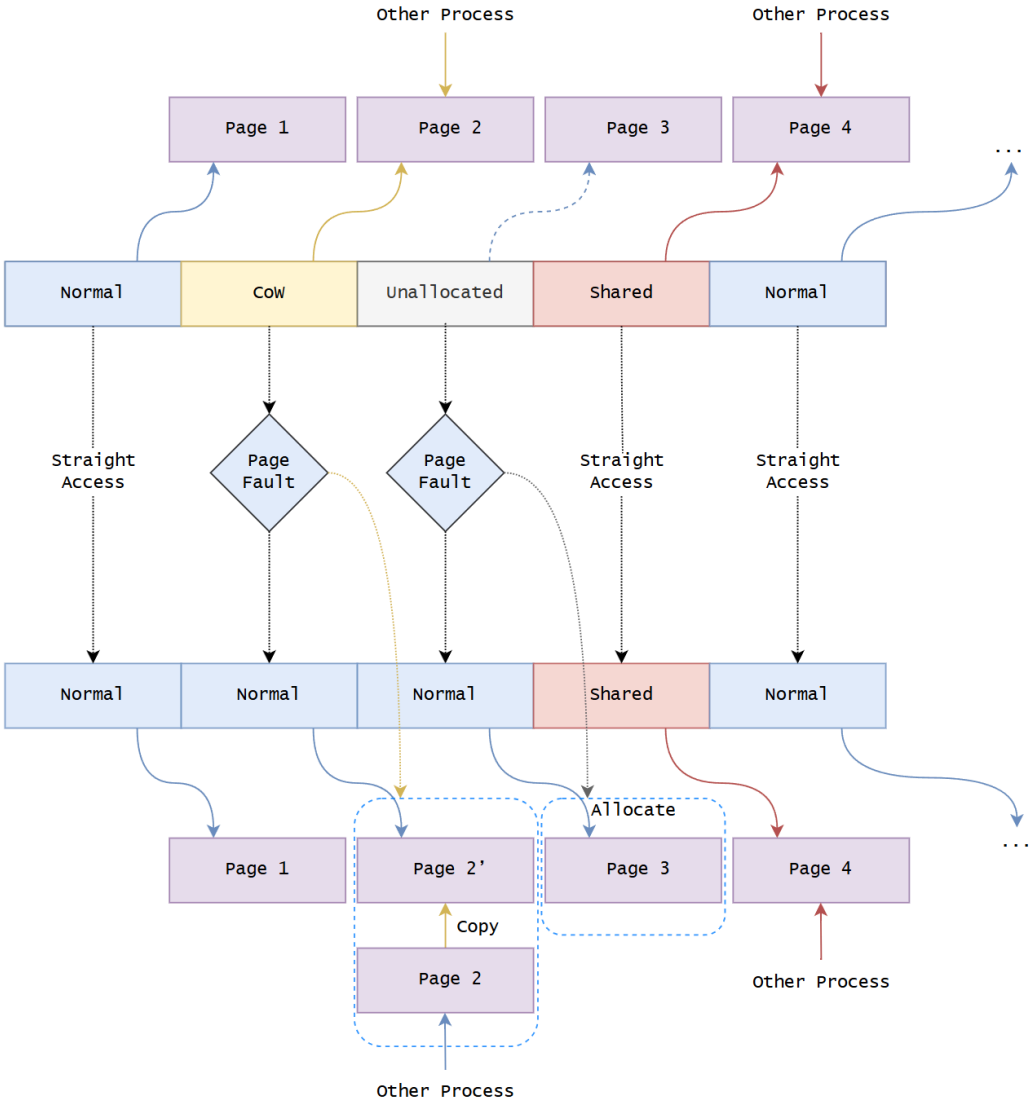

图 4-3　进程缺页异常总览

Del0n1x 的内存管理模块能够正确地对缺页异常进行处理，从而使用写时复制机制、懒分配机制、文件页缓存等提高整体效率。

4.4.2 写时复制机制

当有多个进程可写访问同一内存资源（如文件缓存、物理内存页）但又不希望在进程间共享修改时，写时复制（Copy on Write，CoW）机制能推迟复制操作的发生，直到任意一方尝试进行修改操作。CoW 机制能显著提升操作系统的时空效率。Del0n1x 依赖缺页异常处理实现了对 CoW 机制的支持，并全面使用 CoW 机制以保证进程 `fork`、`mmap` 文件私有映射等功能的高效。

Del0n1x 借用页表项（Page Table Entry，PTE）上未使用的标志位标识 CoW 的内存页。当内存页需要被标记为 CoW 时，内核将遍历页表，将对应 PTE 的 CoW 标志位设置为 1，同时摘除 PTE 上的可写标志位。当进程尝试修改该内存页的内容时，会因为不满足权限触发缺页异常，在异常处理函数中恢复 PTE 的标志位。Del0n1x 依赖 Rust 原子引用计数的强大功能，可以判断缺页异常触发时是否仍然存在内存页的共用，在存在共用时执行复制，在不存在共用时（如 fork 之后父子进程的其中一个已终止或执行了 exec）仅恢复可写标志位，优化了 `fork + exec` 的时间效率。

4.4.3 懒分配机制

懒分配（Lazy Allocation）机制将进程堆栈、mmap 匿名映射等区域分配内存页的时机延迟到访问时。Del0n1x 会为每个进程创建一个 8 MB 大小的用户栈，分配全部 8 MB 的内存页会带来不可忽视的时空开销。应用懒分配机制后，用户栈分配的时间开销分散在运行时，同时未被使用的栈空间将永远不会分配。同理，mmap 匿名映射也应用懒分配机制降低时空开销。

懒分配机制的实现同样依赖于缺页异常处理。当进程访问到未分配内存页的虚拟地址时，会触发缺页异常，Del0n1x 的缺页处理函数将立即为该虚拟页分配对应的物理内存页并映射到页表。

4.5 内核动态内存分配

为了在 Del0n1x 内核中使用 Rust `alloc` 库提供的 `Vec`、`Arc`、`BTreeMap` 等数据结构，避免重复造轮子，Del0n1x 继承了 rCore 中使用的伙伴分配器（Buddy Allocator）用于内核堆的动态分配。伙伴分配器能够针对待分配空间的布局（Layout），在内核堆里寻找连续的未分配区域进行分配。值得注意的是，如果一次性申请分配过大的动态内存空间，可能导致伙伴分配器找不到足够大的连续区域

而出错。为此，Del0n1x 对申请内核堆空间的场景做了预判断，只分配必要的空间，防止此类问题的发生。

4.6 用户地址检查

为了响应用户请求、正确传达信息，内核往往需要频繁访问用户态传入的地址。Del0n1x 采用了用户程序与内核程序共用地址空间的设计，内核可以直接通过 MMU 解引用用户态虚拟地址进行操作。然而，传入的地址中可能存在非法地址。如果不做处理，轻则可能导致内核 `panic`，重则可能导致内核被侵入，用户窃取最高权限。因此，检查传入地址的合法性是内核设计必须考虑的问题。

用户态传入地址不合法主要有以下两种情况：

1. 传入了用户没有访问权限的地址，如内核 `.data` 段、内核堆
2. 传入了未映射或无读写权限的地址

第一种情况可以使用简单的地址数值判断解决。针对第二种情况，Del0n1x 参考了往届队伍的做法，依赖内核 Trap 中的缺页异常处理逻辑解决。Del0n1x 为每一个 CPU 核心定义一个 `ktrap_ret` 变量（Kernel Trap Return-value），用于存储内核 Trap 的执行结果。

```
pub struct CPU{

    ...
    /// None: 没有发生 kernel trap
    /// Some(Ok): 发生 kernel trap, 正常处理（如 CoW、懒分配、文件未缓存）
    /// Some(Err): 发生 kernel trap, 处理异常, 说明存在非法访问
    /// 目前仅支持保存缺页异常的结果
    ktrap_ret: Option<SysResult<()>>,
    ...

}
```

代码 4-9　ktrap_ret 定义

Del0n1x 向待检查地址读/写一个字节，并获取 `ktrap_ret` 的内容，据此判断地址所在的虚拟内存页是否合法。如果需要检查一段连续的地址范围是否合法，则将其拆分为数个内存页分别进行读/写检查即可。

```rust
pub fn try_load_page(addr: VirtAddr) -> SysResult<()> {
    #[cfg(target_arch = "riscv64")]
    unsafe fn try_load_page_inner(addr: usize) {
        asm!(
            "mv t0, a0",
            "lb t0, 0(t0)",
            in("a0") addr,
            out("t0") _,
        );
    }

    #[cfg(target_arch = "loongarch64")]
    unsafe fn try_load_page_inner(addr: usize) {...}

    // get_current_cpu().ktrap_ret.take()
    take_ktrap_ret();
    unsafe {
        try_load_page_inner(addr.0);
    }
    // get_current_cpu().ktrap_ret.take().map_or(Ok(()), |ret| ret)
    take_ktrap_ret().map_or(Ok(()), |ret| ret)
}
```

代码 4-10　用户地址检查

第 5 章　文件系统

5.1　虚拟文件系统

虚拟文件系统（Virtual File System， VFS）是对各种文件系统的抽象，这种抽象屏蔽了各种具体文件系统的细节，为内核提供统一的统一的文件系统接口。在标准的 系统调用中，例如 `open()` 、 `read()` 、 `write()` 中，可以忽略掉文件系统的特性，调用统一的 VFS 接口，从而简洁、安全地实现对文件的各种操作。

我们充分利用 Rust 语言的特性，借鉴 Linux 文件系统设计，以面向 trait 的方式进行编程，提供了方便、安全的文件系统接口供内核的其他模块与用户系统调用使用。

Del0n1x OS 的虚拟文件系统的主要数据结构为 `SuperBlock` 、 `Inode` 、 `Dentry` 、 `File` 。

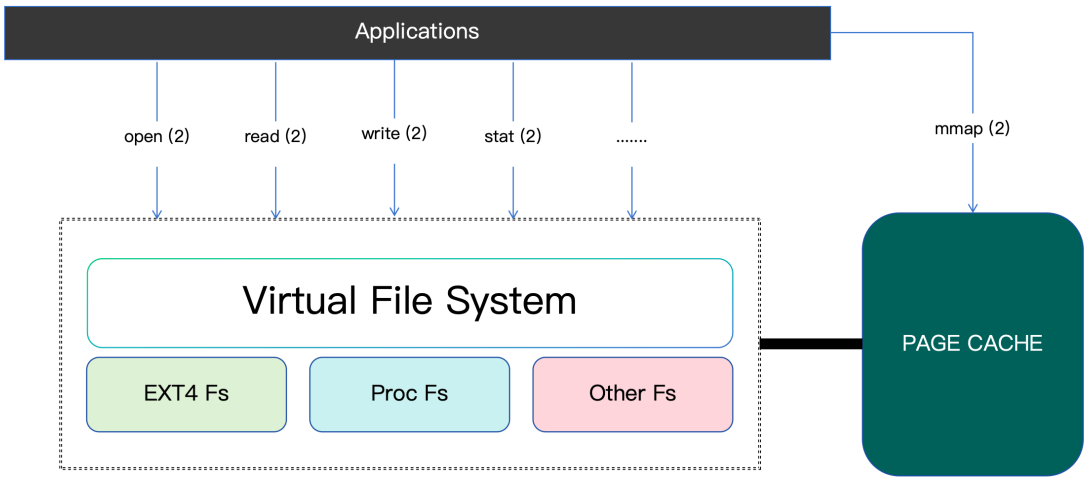

图 5-1　虚拟文件系统

5.1.1　SuperBlock

`SuperBlock trait` 超级块属性是一个具体的文件系统的抽象。每一个具体文件系统都有对 SuperBlock trait 的实现，不同文件系统对同一个方法会有不同的实现。

超级块 `SuperBlock trait` 的定义如下：

```rust
pub trait SuperBlockTrait: Send + Sync {
    /// 获取根节点
    fn root_inode(&self) -> Arc<dyn InodeTrait>;
    /// 将数据写回
    fn sync(&self);
    /// 显示文件系统信息
    fn fs_stat(&self) -> StatFs;
}
```

代码 5-1　文件系统超级块

与 C 语言相比，Rust 提供了足够的抽象机制。对于不同的文件系统，实现 `SuperBlock trait` 后就可以在文件系统中安全和高效地使用，鲜明的方法名称也为编程带来方便。 Rust 同时区别于传统的面向对象语言，抛弃了继承机制的设计，鼓励面向 trait 编程，并使用组合替代继承，使得代码结构更为简单高效。

我们的内核中简化了超级块的提供的功能，仅提供有限的信息。

5.1.2　Inode

索引节点（inode）是文件系统的核心，是磁盘文件的抽象。Linux 中 inode 结构体使用 `struct inode_operations *i_op` 字段作为访问 inode 的函数接口。在 Linux 中，当处理内核文件系统操作或文件系统相关系统调用的时候，会获得对应的 inode 对象，随后去检查并调用 `inode->i_op` 中的函数来实现具体功能。这种面向对象的编程思想极大地提高了内核编码的安全性和便利性，我们无需关心各个文件系统当中如何实现具体的方法的，仅需调用暴露的方法就好了。

Del0n1x 使用 Rust 开发，充分利用了 Rust 原生面向 Trait 编程的优势，定义了索引节点属性 `InodeTrait`。 `InodeTrait` 是对索引节点的抽象，屏蔽了磁盘文件系统的底层技术细节，只提供统一的接口。`InodeTrait` 与索引节点的 `inode_operations` 功能类似，定义了索引节点应当实现的方法，而具体的文件系统的 inode 类型需要实现这些方法（或者使用默认方法），供给上层使用。

`InodeTrait` 的声明如下：

```rust
pub trait InodeTrait: Send + Sync {
    /// inode 的信息
    fn fstat(&self) -> Kstat;
    /// 在文件夹上创建一个子文件
    fn do_create(&self, bare_dentry: Arc<Dentry>, _ty: InodeType)
        -> Option<Arc<dyn InodeTrait>>;
    /// 读文件
    fn read_at(&self, _off: usize, _buf: &mut [u8]) -> usize;
    /// 写文件
    fn write_at(&self, _off: usize, _buf: &[u8]) -> usize;
    /// 截断文件
    fn truncate(&self, _size: usize) -> usize;
    /// unlink 文件
    fn unlink(&self, valid_dentry: Arc<Dentry>) -> SysResult<usize>;
    /// link 文件
    fn link(&self, bare_dentry: Arc<Dentry>) -> SysResult<usize>;
    /// 获得 page cache
    fn get_page_cache(&self) -> Option<Arc<PageCache>>;
    /// 更名
    fn rename(&self, old_path: Arc<Dentry>, new_path: Arc<Dentry>)
        -> SysResult<usize>;
    /// 文件夹读取目录项
    fn read_dents(&self) -> Option<Vec<Dirent>>;
    /// io 操作
    fn ioctl(&self, op: usize, arg: usize) -> SysResult<usize>
}
```

代码 5-2　InodeTrait 接口定义

在文件对象（File）或者目录项对象中会持有数据类型为 `Arc<dyn InodeTrait>` 的 inode 对象。通过 inode 对象实现的 `InodeTrait` 接口我们可以获得文件的各种信息，进行文件的读写、创建、删除等操作。在通过 `dyn InodeTrait` 调用方法时，实际调用的方法将自动分派给对应的文件系统的索引节点（inode）实现。

5.1.3　Dentry

目录项 Dentry 是目录树上节点的抽象。每一个有效的目录项持有对一个合法的 inode 的引用。操作系统使用路径寻得对应的文件，这一过程是由目录项 完成。Del0n1x 通过实现 Dentry，文件系统目录树进行了管理。通过从挂载节点向下搜索，实现对文件的查找。

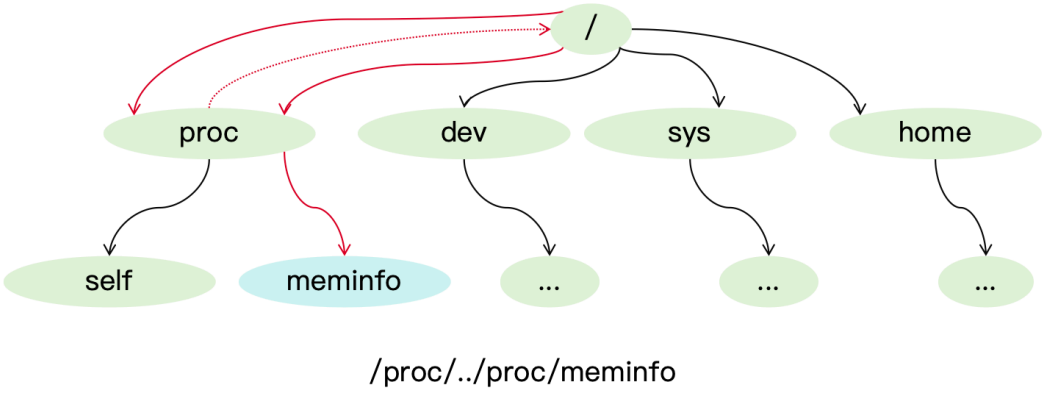

图 5-2　访问目录树

在初始化目录树的过程中，我们需要将具体的文件系统挂载（mount）在目录树上，Del0n1x 的目录树提供了这一功能的实现。操作系统和用户可以把任意的文件系统挂载到目录树上的文件夹节点上，将该目录项持有的 inode 替换为该文件系统的根目录，藉此可以通过对应的路径访问该文件系统下的文件。

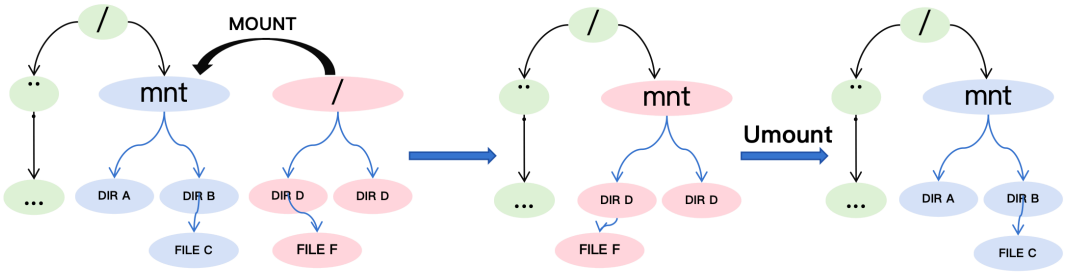

图 5-3　mount and umount

在 Del0n1x 的实现中，当我们挂载了一个新的文件系统到某个目录项上时，原有的目录项上持有的 inode 的引用会隐藏起来，当从这个目录项上卸载（unmount）该文件系统的时候原来持有的 inode 会恢复，进而恢复原有的子树结构。目前相关操作仅支持内核中调用，未来我们希望实现 loop 设备后在用户态也可以使用。

目录项 `Dentry` 的定义如下：

```rust
pub struct Dentry {
    /// 目录项文件名
    name: RwLock<String>,
    /// 对父 dentry 的弱引用
    parent: Weak<Dentry>,
    /// 孩子 dentry 的强引用
    children: RwLock<HashMap<String, Arc<Dentry>>>,
    /// 当前的持有的的 inode 对象
    inode: RwLock<Vec<Arc<dyn InodeTrait>>>,
    /// dentry 的状态
    status: RwLock<DentryStatus>,
}
```

代码 5-3　Dentry 结构

通过 `children` 字段获得当前目录项的子目录项，通过 `parent` 获得当前目录项的双亲目录项，注意到这里使用弱引用（不增加引用计数）防止出现循环引用。`inode` 字段为所持有的索引节点（inode）。

定义目录项状态 `DentryStatus`：

```rust
pub enum DentryStatus {
    /// 这个 dentry 是有效的，并且已经初始化
    Valid,
    /// 这个 dentry 是有效的，但是没有初始化
    Unint,
    /// 这个 dentry 是无效的
    Negtive,
}
```

代码 5-4　DentryStatus 枚举

目录项状态在这三个状态之间转移，当目录项被标记为无效时，会在合适的时机进行回收，并且释放对 inode 对象的引用。当目录项尚未初始化的时候，其会在访问时被初始化。只有当目录项有效时才可以进行访问。

目录项额外使用了一个缓存用于加速从路径到目录项的查找，目录项缓存（DentryCache）使用内核定义的 Cache 泛型容器进行定义。目录项缓存的存在极大地加速了获得 inode 的过程，从而使 Del0n1x 的文件系统获得显著的性能提升。

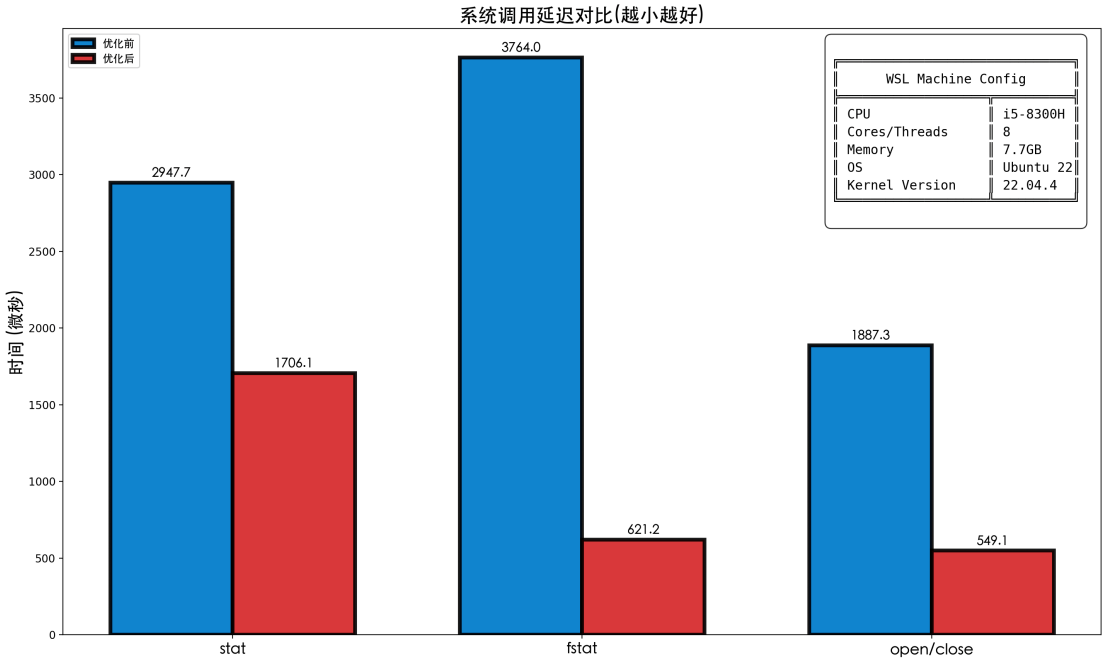

图 5-4　dentry cache 性能对比

5.1.4　File

文件对象（file）是进程中已打开文件在内粗那种的表示。文件由 open 系统调用创建，由 close 系统调用关闭。一个最基本的文件对象由持有的索 inode、文件的偏移和其他状态信息组成。用户态程序调用 open 系统调用去创建文件对象时，首先需要根据路径获得对应的目录项，通过目录项获得索引节 inode，将索引节点包装在文件对象中，并设置其状态，最后将创建好的文件对象注册到文件描述符表。

5.2　磁盘文件系统

磁盘文件系统（Disk File System）是操作系统中用于管理磁盘和其他存储设备的数据存储机制的系统。磁盘文件系统实现了数据以文件和目录的形式存储、命名、访问、更新的具体方式，以及物理存储介质上空间分配的细节。

5.2.1　EXT4 文件系统

Ext4（第四代扩展文件系统）是 Ext3 文件系统的继承者，主要用于 Linux 操作系统。与前代文件系统相比，Ext4 在性能、可靠性和容量方面都有显著改进。Ext4 文件系统对 Unix 操作系统适配性更好，支持硬链接等操作。Del0n1x 使用了开源的 `lwext4-rust` 库，为其实现块设备访问的接口。借助 `lwext4-rust` 库提供的功能，Del0n1x 对 VFS 中的接口进行了实现。

5.3 非磁盘文件系统

非磁盘文件系统（Non-Disk File System）是指不依赖于传统磁盘存储介质的文件系统。在我们的操作系统中提供了若干个非磁盘文件系统供系统使用。

5.3.1 procfs

procfs 是一种特殊的文件系统，它不从磁盘上的文件系统中读取数据，而是从内核中读取数据。procfs 包括：

- `/proc/mounts`：显示当前挂载的文件系统
- `/proc/meminfo`：提供关于系统内存使用情况的信息，包括总内存、可用内存、缓存和缓冲区等详细数据
- `/proc/exe`：当前正在运行的程序
- `/proc/self`：当前正在运行的进程所持有的内容

这个文件系统完整地实现了 VFS 中所有的接口，用户可以**透明地**使用其中的文件。

用户态程序可以很方便地从这些文件中提取相关信息。

5.3.2 devfs

devfs 中的文件代表一些具体的设备，比如终端、硬盘等。devfs 内包含：

- `/dev/zero`：一个无限长的全 0 文件
- `/dev/null`：用于丢弃所有写入的数据，并且读取时会立即返回 EOF（文件结束）
- `/dev/random`：一个伪随机数生成器，提供随机数据流
- `/dev/rtc`：实时时钟设备，提供日期和时间
- `/dev/tty`：终端设备，能支持 ioctl 中的特定命令
- `/dev/loop0`：回环设备，用于虚拟块设备

5.4 页缓存

页缓存（Page Cache）以页为单位缓存文件的内容。当我们需要读文件时，在缓存命中的情况下，就省去了对持久化设备的访问，从而提高性能。同样，当写文件时，也可以暂时写入页缓存，同时标记为脏页，而不需要等待数据真正地被写入到磁盘。脏页由内核进行统一的管理。总而言之，页缓存的设计极大地提高了文件的读写性能。

页缓存同时也是连接文件系统模块和内存模块的桥梁。用户可以调用 mmap 系统调用，将文件映射到用户态地址空间中。当访问 mmap 映射区域内的虚拟页时，对应文件页会被缓存并映射，用户可以通过内存读写实现对文件的安全高效访问。当用户态程序借助共享文件映射进行进程间通信时，页缓存能对进程间数据吞吐性能带来极大地提升。

以下为页缓存的定义：

```
pub struct PageCache {
    pub pages: RwLock<BTreeMap<usize, Arc<Page>>>,
    inode: RwLock<Option<Weak<dyn InodeTrait>>>,
}
```

代码 5-5　PageCache 结构体

在 PageCache 实现中以页对齐的文件偏移（offset）为 key 去获得对应的页。

Del0n1x 在对 Ext4 文件系统的实现中，Ext4 文件系统的 inode 类型通过 RAII 管理页缓存的释放。当 inode 对象被析构的时候，其持有的页缓存也会自动释放，进而释放其占用的所有资源。

5.5　其他数据结构

5.5.1　FdTable

用户态程序使用 open 系统调用打开文件后，会获得文件描述符（File Descriptor）来用于控制文件（File），这需要内核为每一个进程创建一个对应的文件描述符表（Fd Table）用于实现文件描述符到文件的映射。

以下为文件描述符表的实现：

```
pub struct FdTable {
    pub table: Vec<FdInfo>,
    pub rlimit: RLimit64,
    free_bitmap: Vec<u64>,
    next_free: usize,
    freed_stack: Vec<usize>,
}
pub struct FdInfo {
    pub file: Option<Arc<dyn FileTrait>>,
    pub flags: OpenFlags,
}
```

代码 5-6　FdTable 结构

第 6 章　进程间通信

6.1　信号机制

信号是操作系统向进程传递事件通知的一种机制，主要用于通知进程发生了异步事件。在我们的内核中，严格按照 Liunx 中对于信号结构的设计，实现了相对完善且清晰的信号机制。信号相关结构体设计自顶向下为分别为，`SigStruct`（一个包含所有信号处理方法的数组），其中每个元素为 `KSigAction`（内核层信号动作），`SigAction`（信号处理相关配置），三者关系如下：

```rust
#[derive(Clone, Copy)]
pub struct SigStruct {
    pub actions: [KSigAction; MAX_SIGNUM],
}

/// 内核层信号动作
#[derive(Clone, Copy)]
pub struct KSigAction {
    pub sa: SigAction,
    pub sa_type: SigHandlerType,
}

/// 用户层信号处理配
#[derive(Clone, Copy, Debug)]
#[repr(C)]
pub struct SigAction {
    /// 信号处理函数类型，可能是自定义，也可能是默认
    pub sa_handler: usize,
    /// 控制信号处理行为的标志位
    pub sa_flags: SigActionFlag,
    pub sa_restorer: usize,
    /// 在执行信号处理函数期间临时阻塞的信号集合
    /// 信号处理函数执行时，内核会自动将 sa_mask 中的信号添加到进程的阻塞
    /// 信号集
    /// 处理函数返回后，阻塞信号集恢复为原状态
    pub sa_mask: SigMask,
}
```

代码 6-1　信号相关结构体定义

6.2　信号传输

在 Del0n1x 中，用户可以通过 kill 系统调用向进程传送信号，利用参数 pid 可以找到对应的进程或进程组，然后调用 TCB 成员函数接口 `proc_recv_siginfo` 将信号推入对应进程的待处理信号队列中（结构如下）：

```rust
pub struct SigPending {
    /// 检测哪些 sig 已经在队列中,避免重复加入队列
    mask: SigMask,
    /// 普通队列
    fifo: VecDeque<SigInfo>,
    /// 存放 SIGSEGV, SIGBUS, SIGILL, SIGTRAP, SIGFPE, SIGSYS
    prio: VecDeque<SigInfo>,
    /// 如果遇到的信号也在 need_wake 中，那就唤醒 task
    pub need_wake: SigMask,
}
```

代码 6-2　SigPending 结构体

我们将信号处理队列分为普通队列和优先队列，对不同的信号做了优先级处理，这样的数据结构时的 Del0n1x 对于紧急时间和高优先级时的相应延迟更低，提高了内核的实时性。

对于队列中的信号结构体 SigInfo 设计，我们借鉴了 Linux 中的 `siginfo_t` 实现方式，同时对其进行了简化和封装，能够携带更多的数据信息（发送者 pid、子进程 exit code 和信号编码等），这样极大的方便了 Wait4 和 `do_signal` 中对于不同信号的处理分发流程。

6.3　信号处理

进程因系统调用、中断或异常进入内核态，完成内核任务后，在返回用户态前，内核会检查该进程的未决信号。Del0n1x 中信号处理集中在 `do_signal` 函数中，我们会依次遍历 prio 和 fifo 队列，如果该信号没有被阻塞，则根据 siginfo 中信号编码找到对应的信号处理函数 KSigAction，然后对 `KSigAction` 中的 `sa_type` 字段进行模式匹配，对应的动作分别为 Ignore（忽略该信号）、Default（系统默认处理）和 Customized（用户自定义处理函数）。

对于用户自定义函数，内核会下面的流程进行处理（如下图）：

构建用户态栈帧：在内核栈中创建新栈帧，如果用户没有自定义栈帧位置，那么默认为将用户栈 sp 向低地址扩展分配，确保信号处理函数有独立栈空间。

修改返回上下文：将原用户态执行点（如 pc 寄存器）保存到 `UContext` 中，然后复制到用户栈；然后修改当前 `TrapContext` 指向用户处理函数。

切换至用户态：跳转至信号处理函数入口并执行用户自定义函数。在这一过程中，为了避免信号的嵌套处理，需要将原信号加入屏蔽字。

切换到内核态：处理函数结束后调用 sigreturn 系统调用，主动陷入内核态。内核从用户栈恢复原进程上下文，清除信号屏蔽字。

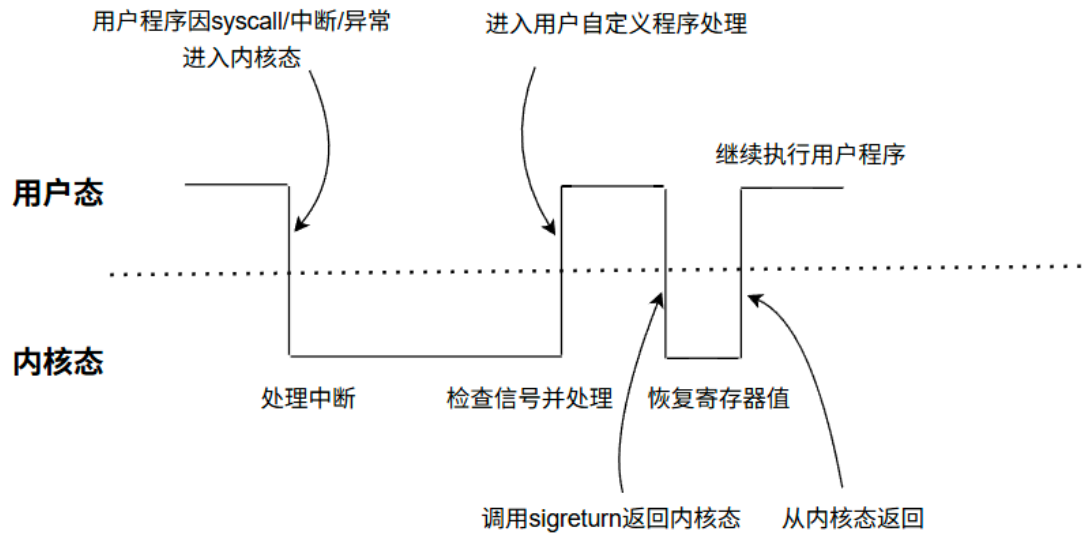

图 6-1　信号处理

6.4　管道（Pipe）

每个进程有自己独立的地址空间，在用户态的情况下，任何一个进程的数据对于其他进程都是不可访问的。为了实现进程之间的数据交换，用户可以请求内核在不同的进程之间开辟一个缓冲区，进程 A 将数据写入缓冲区，然后进程 B 从缓冲区中取走数据，Pipe 就是通过这样的机制实现进程间的通信。

6.4.1　Pipe 设计

从基本功能的分析得知，Pipe 需要维护一个写端和一个读端供进程操作缓冲区，同时，考虑到读写之间并发问题，我们需要对缓冲区加锁，也就是默认当 有一方在读或在写时，另一方需要阻塞；除此之外，为了解决缓冲区并没有数据但是读者持有锁的问题，我们还需要设计读写者唤醒机制，基于异步架构的 调度方式，Del01x 用数组保存进程 waker 句柄来记录有待唤醒的读写者。基于此，Del0n1x 的 Pipe 结构设计如下：

```rust
pub struct Pipe {
    pub flags: OpenFlags,
    pub other: LateInit<Weak<Pipe>>,
    pub is_reader: bool,
    pub buffer: Arc<SpinNoIrqLock<PipeInner>>,
}
pub struct PipeInner {
    pub buf: VecDeque<u8>,
    pub reader_waker: VecDeque<Waker>,
    pub writer_waker: VecDeque<Waker>,
    pub status: RingBufferStatus,
}
```

代码 6-3　Pipe 结构设计

鉴于在 Linux 中 Pipe 是一种文件，Del0n1x 为 Pipe 实现了 `FileTrait`，这样我们可以像操作文件一样建立和操控 Pipe。

6.4.2　读者写者通信

在 Del0n1x 中，我们通过手写 Future 的方式实现管理读者写者之间同步。下面以读者为例，我们为读者实现 `PipeReadFuture` 记录 读者在异步轮循中的关键字段，解释如下。在读者访问缓冲区前，会计算目前可读的长度，如果缓冲区没有数据，那么读者会将自己的唤醒句柄 waker 保存在 pipe 的读者带唤醒数组中，等待写者完成数据写入后唤醒；如果缓冲区有数据，那么读者将该数据段拷贝到 `userbuf` 中，并通知 写者此时 Pipe 缓冲区有空余空间可写。

```rust
struct PipeReadFuture<'a> {
    /// 与写者通信的管道
    pipe: &'a Pipe,
    /// 用户空间指针
    userbuf: &'a mut [u8],
    /// 记录当前用户数据 buf 读取到的位置
    cur: usize,
}

impl Future for PipeReadFuture<'_> {
    type Output = SysResult<usize>;

    fn poll(mut self: Pin<&mut Self>, cx: &mut Context<'_>)
    -> Poll<Self::Output> {
        let this = unsafe { self.get_unchecked_mut() };
        let userbuf_left = this.userbuf.len() - this.cur;
        let read_size = {
            let mut inner = this.pipe.buffer.lock();
            inner.available_read(userbuf_left)
        };

        if read_size > 0 {
            let mut inner = this.pipe.buffer.lock();
            let target =
                &mut this.userbuf[this.cur..this.cur + read_size];
            for (i, byte) in inner.buf
                .drain(..read_size).enumerate() {
                target[i] = byte;
            }
            this.cur += read_size;
            this.pipe.wake_writers(&mut inner);
            Poll::Ready(Ok(read_size))
        } else if !this.pipe.other_alive() {
            return Poll::Ready(Ok(0));
        } else {
            let mut inner = this.pipe.buffer.lock();
            inner.reader_waker.push_back(cx.waker().clone());
            Poll::Pending
        }
    }
}
```

代码 6-4　读者同步

这样的设计看似读者写者之间互相交错唤醒，场面和谐。但试想一下一种极端情况，如果读者或者写者在操作完缓冲区后，因为一些原因导致进程 退出，而此时 Pipe 中还有待唤醒的读者或写者，那这样会造成死等的情况。这样的 bug 是在适配 libcbench 的 pthread 中发现的，为了解决这样的问题， 需要 为 Pipe 实现 `Drop` 方法自动唤醒等待队列中的进程。

6.5 System V IPC 机制

6.5.1 System V IPC 对象

Del0n1x 支持进程间通过 System V IPC 机制进行通信。System V IPC 使用全局唯一的 IPC Key 标识 IPC 对象，知晓 IPC Key 的进程可以调用相关的 ABI 获取 IPC 对象，并使用相关的 ABI 创建通信信道。

System V IPC 对象包括三种类型的对象：

- 消息队列（Message Queue，msg）
- 信号量（Semaphore，sem）
- 共享内存（Shared Memory，shm）

Del0n1x 实现了 `IPCPerm` 结构体用于维护 IPC 对象的所有权和权限信息。在此基础上，Del0n1x 对共享内存对象提供了支持，并为其余两种类型的 IPC 对象预留了可供拓展的接口。

```
#[repr(C)]
pub struct IPCPerm {
    pub key: IPCKey,
    pub uid: u32,
    pub gid: u32,
    pub cuid: u32,
    pub cgid: u32,
    pub mode: IPCPermMode,
    pub seq: u32,
}
```

代码 6-5　IPC Perm 结构体

6.5.2 IPC Key 管理器

Del0n1x 定义了一个全局的 IPC Key 管理器，为每一个 IPC 对象分配唯一的 IPC Key。

```rust
pub struct IPCKey(pub i32);
pub struct IPCKeyAllocator {
    current: i32,
    recycled: BTreeSet<i32>,
}
impl IPCKeyAllocator {
    /// 初始化分配器
    pub fn new() -> Self {...}
    /// 分配 IPC Key
    pub fn alloc(&mut self) -> IPCKey {...}
    /// 释放 IPC Key
    pub fn dealloc(&mut self, key: i32) {...}
}
```

代码 6-6　IPC Key 全局分配器

进程可以通过传入 `IPC_PRIVATE` 调用分配器为创建的 IPC 对象分配 IPC Key，也可以指定对象的 IPC Key，以便从 IPC 对象管理器中获取 IPC Key 对应的 IPC 对象。

```rust
impl IPCKey {
    pub fn new_alloc() -> IPCKey {
        IPC_KEY_ALLOCATOR.lock().alloc()
    }
    pub fn from_user(user_key: i32) -> IPCKey {
        const IPC_PRIVATE: i32 = 0;
        if (user_key == IPC_PRIVATE) {
            Self::new_alloc()
        } else {
            IPCKey(user_key)
        }
    }
}
```

代码 6-7　IPC Key 的创建与获取

6.5.3　System V 共享内存

Del0n1x 实现了 `ShmidDs` 和 `ShmObj` 数据结构，作为操作 System V 共享内存的句柄。用户进程可以使用 `shmget`、`shmctl`、`shmat`、`shmdt` 等 System V 共享内存相关 ABI 创建、访问共享内存 IPC 对象，并通过映射和读写 System V 共享内存实现通信。

```rust
/// System V 共享内存对象元数据
pub struct ShmidDs {
    pub shm_perm: IPCPerm,
    pub shm_segsz: usize,
    pub shm_atime: usize,
    pub shm_dtime: usize,
    pub shm_ctime: usize,
    pub shm_cpid: usize,
    pub shm_lpid: usize,
    pub shm_nattch: usize,
}
/// 维护 System V 共享内存的映射目标
pub struct ShmObject {
    pub shmid_ds: ShmidDs,
    pub pages: Vec<Weak<Page>>,
}
```

代码 6-8　System V 共享内存 IPC 对象

第 7 章　时钟模块

7.1　定时器队列

在操作系统的定时器模块中，我们创造性地实现了一套混合定时器管理系统，将时间轮算法与最小堆优化相结合，相较于 Phoenix 和 Pantheon 使用的最小堆数据结构，我们的混合定时器在时间的处理上更加精细，同时在特定场景下更加的高效。

我们将定时器分为了短期和长期两种，以 600ms 作为阈值，将小于阈值的划分为短期定时器，大于阈值的划分为长期定时器。两种定时器分别存储在时间轮和最小堆中。

```rust
// 定时器队列
pub struct TimerQueue {
    /// 短期定时器（<600ms）
    wheel: SpinNoIrqLock<TimingWheel>,
    /// 长期定时器（最小堆）
    long_term: SpinNoIrqLock<BinaryHeap<TimerEntry>>,
    /// 定时器句柄计数器
    handle_counter: SpinNoIrqLock<u64>,
}
```

代码 7-1　TimerQueue 结构

7.1.1　时间轮设计

时间轮是一种高效的定时器管理数据结构，特别适合处理大量短周期定时器。鉴于我们内核的时钟中断间隔为 10ms，所以将时间轮划分为 60 个槽位，同时时间轮的滴答间隔为 10ms，10ms 自动推进一槽，与硬件时钟中断完美同步。如下图所示，当时间指针指向的槽位为 1 时，代表这次推进将处理 1 槽位中所有的定时器。槽内采用平铺向量存储，插入/删除操作达到 O(1)常数时间，相较于最小堆插入 O(log n)/删除 O(log n)更加高效。

```rust
struct TimingWheel {
    // 时间轮的槽数组，每个槽存储一组定时器条目
    slots: [Vec<TimerEntry>; TIME_WHEEL_SLOTS],
    // 当前指向的槽索引，随着时间推移循环递增
    current_slot: usize,
    // 时间轮当前表示的时间点
    current_time: Duration,
}
```

代码 7-2　TimingWheel 结构体

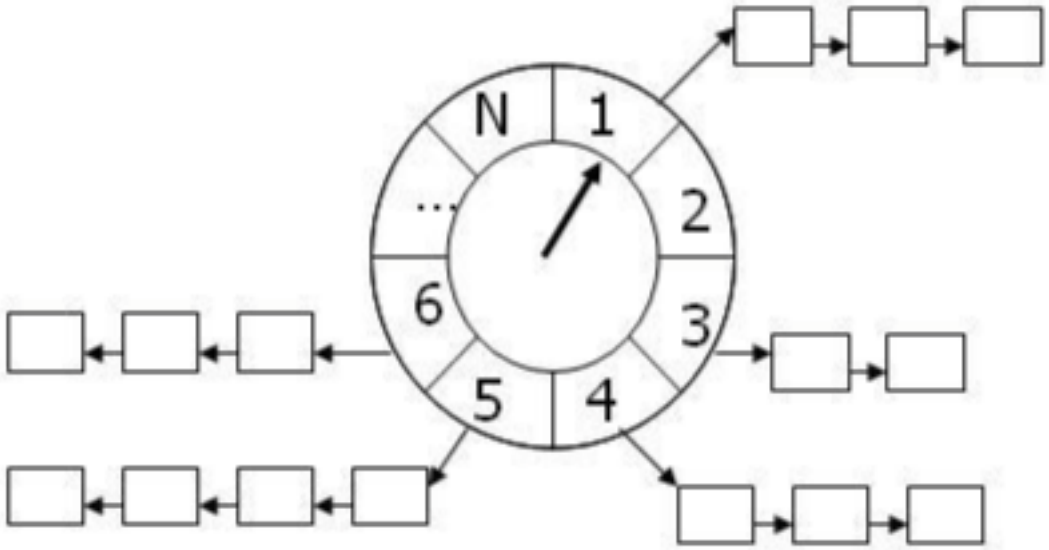

图 7-1　时间轮结构

时间轮推进算法如下所示，通过该函数我们获取到所有超时的定时器 waker，返回给上级调用者用于批量唤醒。

算法 7-1: 时间轮推进算法

function advance_to(target_time: Duration) *to* Vec
wake_list ← empty list
Calculate slots to advance: slots_to_advance = calc_slot(target_time)
for each slot to process **do**
  for each timer entry in current slot **do**
    if entry has expired (expire ≤ target_time) **then**
      Add entry's waker to wake list
      Remove entry from slot
    else
      Keep the entry
  Move pointer to next slot

```
11      Update time wheel's current time
12      if current time exceeds target time then break early
13  return wake_list
```

7.2 定时器

我们使用 `TimerEntry` 数据结构表示定时器，每个定时器都携带专属的 `TimerHandle` ——一个单调递增的唯一标识符。当异步 Future 结束其生命周期时，我们需要 drop 其中剩下的定时器，这时就可以通过对比 `TimerHandle` 找到对应的定时器。系统先在时间轮中闪电扫描，然后扫描二叉堆。为优化堆内搜索，我们设计了临时缓存策略：将非目标项暂存后重新入堆，避免了重建整个堆的昂贵开销。这种双路径检索确保删除操作始终保持高效。

```rust
// 定时器条目
struct TimerEntry {
    /// 到期时间
    expire: Duration,
    /// 唤醒器
    waker: Option<Waker>,
    /// 用于取消的句柄
    handle: TimerHandle,
}
```

代码 7-3　TimerEntry 结构体

为了充分利用异步的优势，我们将需要监测的任务被封装在一个超时 Future 中（如下所示），`deadline` 表示任务的超时期限，`timer_handle` 用于 Drop 机制中确保定时器资源的回收，即使任务提前完成也不会留下幽灵定时器。我们使用 poll 轮循的方式对任务进行推进检测，轮询时执行三重检测：首先尝试推进内部任务，其次检查期限是否届满，最后才注册唤醒器，并且唤醒器只会注册一次。

```rust
pub struct TimeoutFuture<F: Future> {
    inner: F,
    deadline: Duration,
    timer_handle: Option<TimerHandle>,
}

impl<F: Future> Drop for TimeoutFuture<F> {
    fn drop(&mut self) {
        // 确保定时器被删除
        if let Some(handle) = self.timer_handle.take() {
            TIMER_QUEUE.cancel(handle);
        }
    }
}
```

代码 7-4　TimeoutFuture 结构体与 Drop 实现

目前 Del0n1x 已实现较为高效的定时任务管理，但是该优化并没有在初赛的测试用例中体现出来，在初赛测试用例中大部分定时器时间为 1s，其实都会被分配到最小堆结构中。这也反向说明了对于代码的优化也要在特定的场景中才能得到体现，比如时间轮适合处理大量短周期定时任务，最小堆适合处理少量长周期定时任务。

第 8 章　网络模块

在网络模块中，我们使用了官方推荐的 smoltcp 库作为网络协议栈的底层处理，该库具有十分高效的网络协议栈处理，同时通过 Rust 内存安全、事件驱动模型和高度模块化设计，成为资源受限内核场景的理想网络栈。

为了体现 Linux 中"一切皆文件"的思想，我们为网络系统中的关键结构体 `Socket` 实现了 `Filetrait`，统一了网络和文件系统，方便内核设计中对其进行高效管理。在此基础上，我们对网络模块实现分层架构设计，从下至上依次为物理网络驱动层、传输层、Socket 接口层。

目前在初赛阶段我们实现了一个简单的本地回环网络，但是由于时间原因，还未适配通过 netperf、iperf 网络测例，不过这也是我们在决赛阶段的一个目标，努力实现一个简洁高效、易于学习的网络系统模块。

8.1　Socket 套接字

系统核心围绕 `SockMeta` 元数据结构和 `Socket` trait 展开，通过实现 `FileTrait` 将网络套接字无缝集成到文件系统中，实现了 Linux"一切皆文件"的设计哲学。整个架构体现了协议无关性设计，TCP 与 UDP 套接字通过统一的接口向上提供服务，底层差异由各自的具体实现处理。为了利用文件描述符操控套接字，我们将 socket 映射到进程控制块的 `fdtable` 中：

```rust
/// 将一个 socket 加入到 fd 表中
pub fn sock_map_fd(
    socket: Arc<dyn FileTrait>,
    cloexec_enable: bool
) -> SysResult<usize> {
    let mut flag = OpenFlags::O_RDWR;
    let fdInfo = FdInfo::new(socket, flag);
    let new_info = fdInfo.off_Ocloexec(!cloexec_enable);
    let task = current_task().expect("no current task");
    let fd = task.alloc_fd(new_info)?;
    Ok(fd)
}
```

代码 8-1　将 socket 映射到 fd 的函数

`SockMeta` 结构体作为套接字的核心元数据容器，采用 Rust 的强类型系统精确描述套接字状态。其字段设计反映了网络连接的全生命周期：从初始创建时的空端口和端点，到绑定后的本地地址确定，再到连接建立后的远程端点记录。特别是通

过 Option 类型明确区分已初始化和未初始化状态，避免了传统 C 实现中常见的无效值问题。

```rust
pub struct SockMeta {
    pub domain: Sock,
    pub iptype: IpType,
    pub recv_buf_size: usize,
    pub send_buf_size: usize,
    pub port: Option<u16>,
    pub shuthow: Option<ShutHow>,
    pub local_end: Option<IpEndpoint>,
    pub remote_end: Option<IpEndpoint>,
}
```

代码 8-2　SockMeta 结构体

`Socket` trait 定义了完整的套接字操作接口，既包含标准的 bind/connect/listen 等基本操作，也提供了 send_msg/recv_msg 等增强功能。关键接口定义如下：

```rust
#[async_trait]
pub trait Socket: FileTrait {
    /// 异步接受一个传入的连接请求
    async fn accept(&self, flags: OpenFlags)
        -> SysResult<(IpEndpoint, usize)>;
    /// 异步连接到指定的地址
    async fn connect(&self, addr: &SockAddr) -> SysResult<()>;
    /// 异步发送消息到指定地址
    async fn send_msg(&self, buf: &[u8], dest_addr: &SockAddr)
        -> SysResult<usize>;
    /// 异步接收消息
    async fn recv_msg(&self, buf: &mut [u8])
        -> SysResult<(usize, SockAddr)>;
    /// 绑定套接字到本地地址
    fn bind(&self, addr: &SockAddr) -> SysResult<()>;
    /// 开始监听传入连接
    fn listen(&self, backlog: usize) -> SysResult<()>;
    /// 设置接收缓冲区大小
    fn set_recv_buf_size(&self, size: u32) -> SysResult<()>;
    /// 设置发送缓冲区大小
    fn set_send_buf_size(&self, size: u32) -> SysResult<()>;
    /// 获取套接字绑定的本地地址
    fn get_sockname(&self) -> SysResult<SockAddr>;
    /// 获取对端连接的地址
    fn get_peername(&self) -> SysResult<SockAddr>;
    /// 设置 TCP 保持活动选项
    fn set_keep_alive(&self, action: u32) -> SysResult<()>;
}
```

代码 8-3　Socket trait 部分接口定义

该 trait 标记为 `async_trait` 以适应现代异步 IO 需求，同时继承 `FileTrait` 实现了文件描述符的统一管理，可以通过 `Filetrait` 中的 `pollin`、`pollout` 异步检查网络缓冲区是否可读可写。

8.2 Ethernet 设备

Del0n1x 使用 `NetDev` 结构体实现网络设备的封装和抽象，其包含两个关键组件：`device` 字段表示具体的网络设备类型，当前支持环回接口；`iface` 字段维护了访问 smoltcp 协议栈的通道。这种分离设计使得设备驱动与协议栈保持松耦合，未来扩展新设备类型时无需修改上层协议逻辑。

```
pub enum NetDevType {
    Loopback(Loopback),
    Unspec,
}

pub struct NetDev {
    pub device: NetDevType,
    pub iface: Interface,
}
```

代码 8-4　NetDevType 与 NetDev 结构体

◆ 网络轮循

`poll()` 方法是整个网络栈的驱动引擎，实现了事件处理的核心循环，每次 poll 轮循将处理 `SOCKET_SET` 中所有的句柄。

`iface.poll()` 调用实现了三层重要功能：

- 接收处理：从设备读取数据包并递交给相应协议处理程序
- 发送处理：将协议栈待发送数据提交给设备驱动
- 状态更新：维护 TCP 定时器、重传队列等状态机

该方法主要在发送和就收数据的循环中使用，驱动协议栈与设备的异步交互。

```
pub fn poll(&mut self) {
        let instant = Instant::from_millis(get_time_ms() as i64);
        let mut socket = SOCKET_SET.lock();
        let device = match self.device {
            NetDevType::Loopback(ref mut dev) => dev,
            NetDevType::Unspec => panic!("Device not initialized"),
        };
        self.iface.poll(instant, device, &mut socket);
    }
```

代码 8-5　网络轮询 poll 方法

8.3 传输层——UDP 与 TCP

TCP 是一种面向连接的字节流套接字，而 UDP 是一种无连接的报文套接字。他们都继承了 `SockMeta` 中的字段，实现了不同的 `Socket` trait。

```rust
pub struct TcpSocket {
    pub handle: SocketHandle,
    pub flags: OpenFlags,
    pub sockmeta: SpinNoIrqLock<SockMeta>,
    pub state: SpinNoIrqLock<TcpState>,
}

pub struct UdpSocket {
    pub handle: SocketHandle,
    pub flags: OpenFlags,
    pub sockmeta: SpinNoIrqLock<SockMeta>,
}
```

代码 8-6　TcpSocket 与 UdpSocket 结构体

在 Tcp、Udp 数据结构中，都是用 `SocketHandle` 表示自己的句柄。该句柄存放在 smoltcp 提供的全局 `SOCKET_SET` 中，该 set 可以类比为 fdtable，方便快速获取到该套接字的相关句柄，并通过该句柄实现与 smoltcp 底层网络栈功能交互。

```rust
/// 全局 handle 管理器
pub static ref SOCKET_SET: SpinNoIrqLock<SocketSet<'static>> =
        SpinNoIrqLock::new(SocketSet::new(vec![]));

/// 通过闭包快速获取到对应的 handle
pub fn with_socket<F, R>(&self, f: F) -> R
where
    F: FnOnce(&mut udp::Socket<'_>) -> R,
{
    let mut binding = SOCKET_SET.lock();
    let socket = binding.get_mut::<udp::Socket>(self.handle);
    f(socket)
}
```

代码 8-7　SOCKET_SET 及 with_socket 函数

8.4 Port 端口分配

在网络交互过程中，`PortManager` 负责动态管理 TCP/UDP 端口资源的分配与回收。该系统采用多层级管理策略，在保证线程安全的前提下实现高效端口分配。为了记录端口分配清空，Del0n1x 采用了双位图的设计，单个协议仅需 8KB 内存

（65536 位）即可实现 O(1)复杂度的状态查询，相比传统哈希表节省了 90% 以上的内存开销。双位图设计彻底避免了 TCP/UDP 端口冲突。

```rust
pub struct PortManager {
    /// 动态端口范围
    pub start: u16,
    pub end: u16,
    /// 回收端口队列
    pub recycled: VecDeque<u16>,
    /// TCP 端口位图
    pub tcp_used_ports: BitVec,
    /// UDP 端口位图
    pub udp_used_ports: BitVec,
}
```

代码 8-8　PortManager 结构体

◆ 分配算法设计

端口的 `alloc` 方法实现了三级分配策略，如算法所示。先从回收队列获取，用局部性原理提升缓存命中率；然后随机尝试，`PORT_RANGE` 范围内进行有限次随机探测；如果都失败，最后使用顺序扫描，保在极端情况下仍能穷尽搜索。

算法 8-1: 端口分配三级策略

```
    function alloc(domain: Sock) to Result
1   if recycled queue is not empty then
2       port ← recycled.pop_front()
3       mark_used(domain, port)
4       return port
5   chance ← (end - start) - |recycled|
6   for i = 0 to chance-1 do
7       random_port ← start + (random() % PORT_RANGE)
8       if try_mark_used(domain, random_port) then
9           return random_port
10  for port ← start to end do
11      if try_mark_used(domain, port) then
12          return port
13  return error EADDRINUSE
```

第 9 章　设备

9.1　设备管理模块概述

设备的管理在操作系统中起到至关重要的部分。Del0n1x 对设备进行抽象和封装，确保内核对设备进行规范的管理，向其他模块提供简便易用的接口，并为拓展新的设备提供方便。

设备管理模块实现了设备的发现和初始化、驱动程序匹配与加载等功能。操作系统依赖设备管理模块与计算机系统中的硬件部分进行信息的交换，设备管理模块的设计直接影响整个系统的可用性、稳定性与性能。

Del0n1x 中，基本设备抽象为基本设备属性接口 `BaseDriver trait` 。块设备、tty 设备、网络设备等一切设备的实现均拓展于这个接口。例如，Del0n1x 将块设备抽象为块设备属性接口 `BlockDriver trait` ，这个属性继承了 `BaseDevice trait` 。对于一个块设备的具体实现，需要实现块设备属性接口 `BlockDriver trait` ，通过这样的设计，可以对设备进行统一的管理与使用。

9.2　设备树

操作系统通过解析设备树实现对设备的发现。Del0n1x 内核获取到设备树的地址的流程如下：

1. OpenSBI 启动：当系统启动时，OpenSBI 固件首先运行。它完成基础的硬件初始化，如内存控制器设置、I/O 初始化等
2. 传递控制权到内核：OpenSBI 初始化完成后，将控制权传递给内核的入口点，并传递必要的参数。这些参数包括：
   - `hart_id` ：当前硬件线程的 ID。
   - `dtb_addr` ：设备树地址，该地址指向设备树描述符（DTB），描述了系统的硬件布局和配置信息。

对于 LoongArch QEMU virt 平台，我们通过查看 QEMU 源码确定了设备树的地址。

Del0n1x 通过 `probe` 方法，从设备树的根节点进行搜索，获得当前机器上的设备与其地址，当检测到系统支持的设备的时候就调用注册函数，使用对应的驱动程

序实现内核中对该设备的抽象，形成内核中的设备实体，供给其他模块使用。通过设备树解析，我们可以实现同一份内核二进制在不同的硬件上启动。

未来我们希望完善 devtmpfs ，实现从设备到文件的抽象。

第 10 章　硬件抽象层

10.1　硬件抽象层总览

为支持多平台运行与测试，Del0n1x 在内核中实现了功能基本完整的硬件抽象层（Hardware Abstraction Layer，HAL），可以运行于 RISC-V64 和 LoongArch64 两种指令集架构的 QEMU 平台。Del0n1x 的内核代码全部基于硬件抽象层开发，屏蔽了架构细节和平台差异，具有更好的兼容性和可移植性。

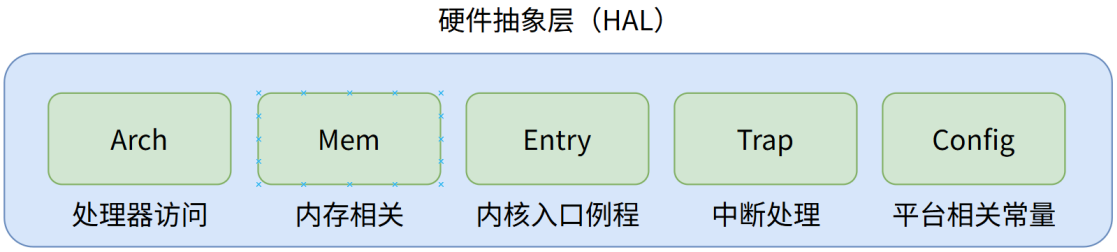

图 10-1　硬件抽象层示意图

10.2　处理器访问接口

Del0n1x 在 `riscv` 和 `loongarch` 外部库的帮助下，对 RISC-V 和 LoongArch 两种架构下的处理器访问实现了统一抽象。在内核代码中，对处理器及相关资源的访问包括以下几种类型：

- 读写通用寄存器
- 访问控制状态寄存器
- 执行特殊的控制指令（如内核刷表指令）

在 Del0n1x 中，这些操作只需要调用统一的接口即可完成，具体实现细节位于 HAL 中，便于构建架构无关的内核代码。

10.3　内核入口例程

RISC-V 和 LoongArch 架构在内核启动上的细节有所异同。LoongArch 架构启动时需要配置直接映射地址翻译模式，需要为分页地址翻译模式设置高半空间、低半空间两个页表 token，还需要设置 TLB 重填异常入口，这些都是 RISC-V 架构下内核启动所不需要的步骤。Del0n1x 的硬件抽象层将初始化时的架构相关部分抽象为 `arch_init()` 例程，供内核初始化时调用，为内核入口的规范化提供支持。

```
// loongarch
pub fn arch_init() {
    mmu_init();
    euen::set_fpe(true);
    tlb_init(tlb_fill as usize);
}
```

代码 10-1　arch_init 例程

10.4　内存管理单元与地址空间

10.4.1　物理内存

RISC-V 和 LoongArch 的 QEMU virt 平台的物理编址方式和布局存在很大区别。以下分别为两个平台的物理地址布局：

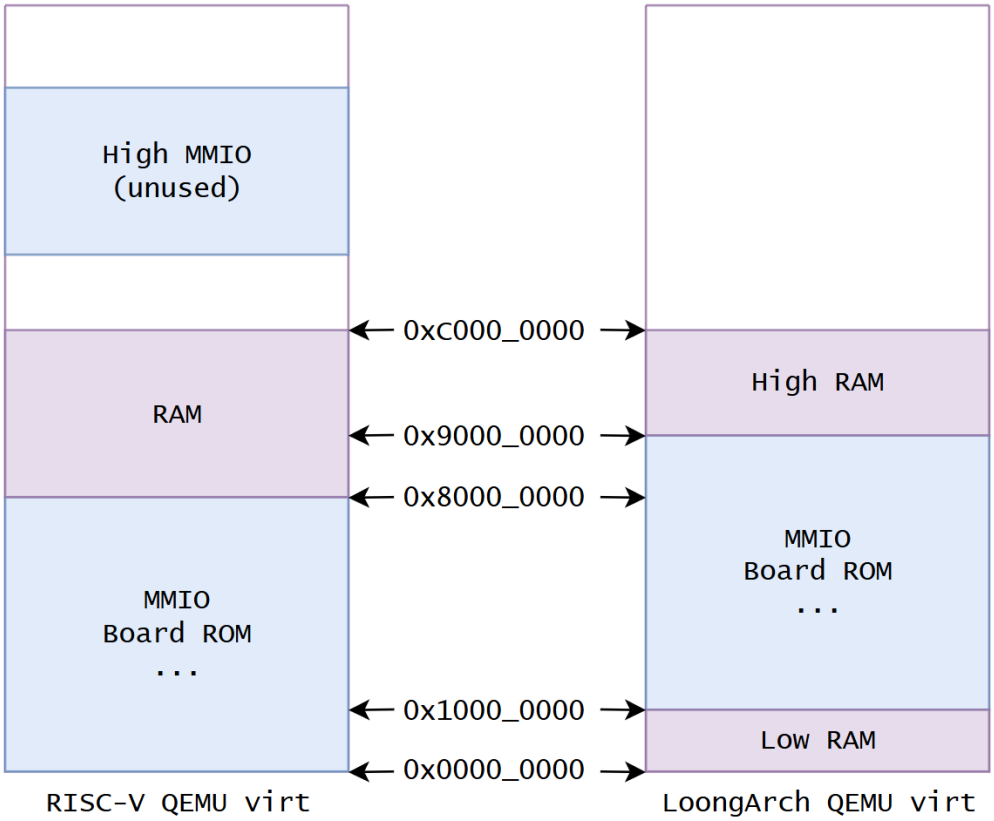

图 10-2　QEMU virt 物理地址布局

RISC-V QEMU virt 平台将 RAM 编码于 `0x8000_0000` 以上的物理地址空间。而 LoongArch QEMU virt 平台则将 RAM 切分为 `lowram` 和 `highram` 两个部分，其中

`lowram` 位于 `0x1000_0000` 以下的物理地址空间，`highram` 位于 `0x9000_0000` 以上的物理地址空间。Del0n1x 将内核镜像加载于 `lowram` 中。

10.4.2 分页地址翻译模式

Del0n1x 使用 SV39 分页地址翻译模式。在不考虑直接映射窗口的前提下，Del0n1x 将完整的 39 位虚拟地址空间分为高半部分和低半部分，高半部分的地址范围为 `0xffff_ffc0_0000_0000~0xffff_ffff_ffff_ffff`，低半部分的地址范围为 `0x0000_0000_0000_0000~0x0000_003f_ffff_ffff`。在分页地址翻译模式中，地址格式不满足 39 位整数的符号拓展形式的，将被视为非法地址。

RISC-V 架构使用控制状态寄存器 SATP 控制分页模式类型（如 SV39）并存储页表根目录的物理页号。Del0n1x 为每一个进程创建一张页表，并为其应用整个虚拟地址空间内的全部映射。当地址空间切换时，通过对 SATP 的修改，即可达到切换页表的目的。

LoongArch 架构的情况有所不同。LoongArch 架构使用 CSR.CRMD 控制状态寄存器的 PG 位开关分页地址翻译模式，通过设置 CSR.PWCL 和 CSR.PWCH 控制状态寄存器中的 `Dir{}_base`、`Dir{}_width` 等部分手动配置分页模式细节。同时，LoongArch 架构下虚拟地址空间低半部分和高半部分的遍历应用不同的页表，其根目录地址分别经由 CSR.PGDL 和 CSR.PGDH 控制状态寄存器设置。这意味着 LoongArch 架构下一个进程需要创建两张页表，分别映射虚拟地址空间的两半。当地址空间切换时，需要按需修改 CSR.PGDL 或 CSR.PGDH，从而灵活保留不需要切换的部分。

Del0n1x 在硬件抽象层中为两种架构实现了分页地址翻译模式的初始化，为 LoongArch 手动配置了 SV39 分页模式。由于 Del0n1x 使用虚拟地址空间的高半部分作为通用的内核地址空间，借助 LoongArch 架构的“双目录”设计，在地址空间切换时只需修改 CSR.PGDL。

```rust
pub fn mmu_init() {
    // 设置页表项长度
    pwcl::set_pte_width(8);
    // 设置页表第三级目录的索引位位置和长度
    pwcl::set_ptbase(PAGE_SIZE_SHIFT);
    pwcl::set_ptwidth(PAGE_SIZE_SHIFT - 3);
    // 设置页表第二级目录的索引位位置和长度
    pwcl::set_dir1_base(PAGE_SIZE_SHIFT + PAGE_SIZE_SHIFT - 3);
    pwcl::set_dir1_width(PAGE_SIZE_SHIFT - 3);
    // 设置页表根目录的索引位位置和长度
    pwch::set_dir3_base(PAGE_SIZE_SHIFT + PAGE_SIZE_SHIFT - 3
      + PAGE_SIZE_SHIFT - 3);
    pwch::set_dir3_width(PAGE_SIZE_SHIFT - 3);
}
```

代码 10-2　LoongArch SV39 分页地址翻译模式初始化

10.4.3　页表

Del0n1x 支持两种架构下的多级页表。在硬件抽象层中，Del0n1x 为两种架构分别实现了对页表项（Page Table Entry，PTE）的封装，可以便捷地访问 PTE 中存储的物理页号和各标志位。Del0n1x 使用 Rust 宏为 PTE 标志位定义了统一的 `checker` 和 `setter` 方法，用于在内核代码中灵活修改 PTE 标志位。

```rust
bitflags! {
    pub struct PTEFlags: usize {
        const V = 1 << 0;
        const R = 1 << 1;
        const W = 1 << 2;
        const X = 1 << 3;
        const U = 1 << 4;
        const G = 1 << 5;
        const A = 1 << 6;
        const D = 1 << 7;
        const COW = 1 << 8;
    }
}
impl PTEFlags {
    // 为标志位 FLAG 实现 is_[FLAG](&self) -> bool 方法
    impl_flag_checker!(
        pub U,
        pub V,
        ... // 篇幅需要，省略
        pub COW
    );
    // 为标志位 FLAG 实现 set_[FLAG](&mut self, bool)-> &mut Self
    // 方法，支持链式调用
    impl_flag_setter!(
        pub U,
        pub V,
        ... // 篇幅需要，省略
    );
}
```

代码 10-3　RISC-V PTEFlags 实现

Del0n1x 的设计中，LoongArch 架构的内核地址空间需要单独使用一张页表，而 RISC-V 架构的内核地址空间则与用户地址空间共用同一张。Del0n1x 在硬件抽象层中为两个架构均映射一张内核页表，区别在于 LoongArch 架构下该页表会被写入 CSR.PGDH 并永不切换，而 RISC-V 架构下该页表只作为一个映射用户地址空间时的模板。当创建新的进程页表时，内核页表中根目录中映射地址空间高半部分的页表项将被复制到新的页表中，供内核态访问。

10.4.4　直接映射窗口

LoongArch 架构支持直接映射地址翻译模式，该模式下允许通过修改 CSR.DMW0 CSR.DMW3 控制状态寄存器配置至多 4 个直接映射窗口。当虚拟地址的高 4 位（在 LoongArch64 中为[63:60]位）恰好与某个直接映射窗口的高 4 位相同时，虚拟地址将被直接映射为其低 PALEN 位的物理地址（PALEN 为机器有效物理地址长度）。使用 `cpucfg` 指令可以查明，LoongArch QEMU virt 平台

下 PALEN=48，故直接映射窗口将 `0xW000_xxxx_xxxx_xxxx` 范围内的所有虚拟地址映射为 `0x0000_xxxx_xxxx_xxxx` 。通过修改 CSR.DMWx 还可配置直接映射窗口的允许访问特权级、存储访问类型。

Del0n1x 使用 `0x8000_xxxx_xxxx_xxxx` 和 `0x9000_xxxx_xxxx_xxxx` 两个直接映射窗口，均限定内核特权级（PLV0）使用，分别用于设备访问和物理内存访问。

10.4.5 TLB 重填

LoongArch 架构使用软件管理 TLB。当发生 TLB 中没有匹配项时，将触发 TLB 重填异常，跳转到内核设置的 TLB 重填入口执行软件重填。现阶段 Del0n1x 使用了往届优秀作品 NPUCore-IMPACT 编写的 TLB 重填代码。

第 11 章　总结与展望

11.1　工作总结

1. 支持 LoongArch64 和 RISC-V64 架构，实现了自己的硬件抽象层。
2. 实现进程管理，以无栈协程的方式高效调度任务。
3. 实现设备树解析
4. 实现 COW、懒分配等内存优化策略。
5. 实现虚拟文件系统，将具体文件系统与内核解耦合，实现了 page cache 和 dentry cache 优化。
6. 支持信号模块，实现进程间信号通信。
7. 时钟模块支持时间轮混合最小堆的时钟管理方式。
8. 支持多核调度运行。
9. 实现 100+条系统调用。
10. 支持运行 llama2.c，执行大模型推理。

11.2　未来计划

1. 完善 net 模块，支持网络上板。
2. 完善 loop 设备，实现功能更加完善的 mount 机制。
3. 适配龙芯板和 riscv 板，完善相关驱动。
4. 支持外设中断。
5. 支持更多 ltp 测例，修复更多内核不稳定的 bug。
6. 支持更多现实应用。

11.3　参考

- Phoenix：无栈协程、内存管理
- Polyhal，NPUCore-IMPACT： 硬件抽象层
- Phoenix、MinotaurOS：设计文档模板