# NPUcore-重生之我是菜狗

# 初赛文档

西北工業大學

NORTHWESTERN POLYTECHNICAL UNIVERSITY

# 目 录

# 图目录

# 代码片段目录

# 第 1 章 概述

## 1.1 NPUcore 介绍

### 1.1.1 NPUcore 的特性

**• 完善的内核功能支持:** NPUcore 根据 Linux Manual 支持或不完全支持的系统调用多达 102 个，且较为完整地实现了信号机制，线程等功能，在初赛阶段针测试用例获得了满分。但我们的目标并不仅仅是通过测试用例，而是希望 NPUcore 具有一定的趣味性和实用性，这就要求 NPUcore 能够运行更多用户程序。因此我们支持了 bash 这个许多 Linux 发行版的默认 shell，结合 busybox 可以获得较好的 CLI 使用体验。

**• 精巧的内存管理:** NPUcore 团队围绕内存管理做了很多工作：懒分配、写时复制、zRAM，虚拟内存等。这些机制使得内存能够得到尽可能的利用，能够支撑更多用户程序。

**• 基于等待队列的阻塞:** 阻塞与非阻塞是资源访问的两种方式。在非阻塞的资源访问下，通常要使用轮询来检查待获取的资源是否就绪，这种反复查询的操作会造成 CPU 资源的浪费。NPUcore 实现了基于等待队列的系统资源访问，futex 等系统调用使用等待队列实现，能够提高内核事件通知和资源访问的效率。

**• 块缓存和页缓存:** NPUcore 团队通过对函数调用的跟踪统计发现文件系统在 I/O 时的等待有极大的开销，为了提高系统 I/O 性能, NPUcore 实现了类似 Linux 的 Buffer Cache(块缓存) 和 Page Cache(页缓存)，Buffer Cache 缓存块设备上的数据块，加速对存储器的访问。Page Cache 缓存文件的逻辑内容, 加速对文件内容的访问。此外，考虑到开发板的内存有限，而 Buffer Cache 和 Page Cache 之间又存在数据冗余，NPUcore 尝试将 Buffer Cache 和 Page Cache 之间重复的数据映射到相同的物理页，并且进行维护，确保不会出现一致性问题。这样能够提升内存利用率，减少 Buffer Cache 和 Page Cache 之间的拷贝。此外，我们还根据 Fat 区和数据区的特性设计了不同的缓存策略，使得文件系统的 I/O 性能大幅上升。

NPUcore 使用较为激进的缓存策略，Page Cache 容量不设上限，所有的内存空间都可以作为缓存使用。发生内存不足时，NPUcore 会根据 LRU 算法清理无用缓存。经过测试发现，在这种缓存策略下运行大多数测例时，对每个文件 NPUcore 只从外存读取一次，之后的读写全部发生在 Cache 中，从而带来极大的性能提升。

**• 高效可扩展的虚拟文件系统:** 在前期开发中，NPUcore 团队注意到文件目录的查找操作每次都需要访问底层文件系统，反复读取 Fat 区和数据区，造成了巨大的耗时。NPUcore 在开发过程中对虚拟文件系统进行了重写，引入了目录树这一数据结构，作为文件目录逻辑结构的缓存，一些文件系统操作如切换目录等在缓存命中时不再需要访问底层文件系统，大幅减少文件 I/O，使文件操作更加高效。

另外，在 Linux 的 VFS 中，同一文件操作可以通过函数指针实现对不同底层文件系统的函数调用，这其实是一种由 C 语言实现的“多态”。我们借鉴这种思想，使用 Rust 的语法将文件操作 trait 化，这样能够让 VFS 具有更高的可扩展性。

• **稳定性:** NPUcore 遵循软件工程的思想，在每次合并新模块后都会进行压力测试和回归测试，能够尽可能发现系统中隐藏的 bug。结合完善的运行日志系统，QEMU 远程调试和 JTAG 硬件调试，可以迅速定位问题并加以解决。

### 1.1.2 目录树结构

NPUcore 的目录树结构如下：

```
.
├── bash
├── buildfs.sh //构建脚本
├── Cargo.lock
├── Cargo.toml
├── clear_doc.sh
├── initproc
├── la_fat
├── la_gdbserver
├── Makefile
├── run_script //运行脚本
├── src
│   ├── arch
│   │   ├── la64
│   │   └── mod.rs
│   ├── console.rs
│   ├── drivers //驱动相关
│   │   ├── block //块设备
│   │   ├── mod.rs
│   │   └── serial //串口
│   ├── fs 文件系统
│   │   ├── cache.rs //缓存
│   │   ├── dev
│   │   ├── directory_tree.rs //目录树结构
│   │   ├── fat32 //文件系统具象
│   │   ├── filesystem.rs
│   │   ├── file_trait.rs
│   │   ├── layout.rs //文件系统框架
│   │   ├── mod.rs
│   │   ├── poll.rs
│   │   └── swap.rs
│   ├── lang_items.rs
│   ├── linker.ld
│   ├── load_img.S
│   ├── main.rs
│   ├── mm 内存管理
│   │   ├── address.rs //地址管理
│   │   ├── frame_allocator.rs //帧分配器
│   │   ├── heap_allocator.rs //堆分配器
│   │   ├── map_area.rs //地址映射
│   │   ├── memory distribution //内存分配
│   │   ├── memory_set.rs //内存集
```

```
43 │   │   ├── mod.rs
44 │   │   ├── page_table.rs //页表结构
45 │   │   └── zram.rs
46 │   ├── net 网络模块
47 │   │   ├── address.rs //网络地址映射
48 │   │   ├── config.rs //设备配置
49 │   │   ├── mod.rs
50 │   │   ├── tcp.rs
51 │   │   ├── udp.rs
52 │   │   └── unix.rs
53 │   ├── preload_app.S
54 │   ├── syscall //各模块系统调用结合
55 │   │   ├── errno.rs
56 │   │   ├── fs.rs //文件系统
57 │   │   ├── mod.rs
58 │   │   ├── net.rs //网络模块
59 │   │   ├── process.rs //进程模块
60 │   │   └── syscall_macro.rs
61 │   ├── task //进程管理
62 │   │   ├── context.rs //上下文信息
63 │   │   ├── elf.rs
64 │   │   ├── manager.rs //进程管理器
65 │   │   ├── mod.rs
66 │   │   ├── pid.rs //进程模块
67 │   │   ├── processor.rs //进程轮询
68 │   │   ├── signal.rs
69 │   │   ├── task.rs //TCB模块
70 │   │   └── threads.rs //线程模块
71 │   ├── timer.rs
72 │   └── utils
73 │       ├── error.rs
74 │       ├── mod.rs
75 │       └── random.rs
```

# 第 2 章　NPUCore 的设计与实现

## 2.1 NPUcore 的进程管理

### 2.1.1 概述

进程指的是在系统中运行的一个程序的实例。而进程的生命周期包括从创建，就绪，阻塞，运行中，退出。在一个进程被创建之后，他会进入 NPUcore 中的就绪队列，在被操作系统调度之后将会进入到运行状态。在运行状态下时间片耗尽或者是主动让出 CPU 的时候，进程会进入到就绪队列中，等待下一次被调度。在运行的时候调用诸如 wait 等系统调用，进程会进入到阻塞队列中，等待被唤醒。当进程执行结束，他会退出，释放系统资源。

NPUcore 进程管理的代码树如下：

```
os
|--src
|--task
|--context.rs //存储上下文信息
|--elf.rs //ELF文件加载和解析
|--manager.rs //任务管理器的结构存储
|--mod.rs//任务操作系统任务管理和调度的实现
|--pid.rs//实现内核栈、进程分配器、内核栈分配器等数据结构及操作
|--processor.rs // 进程轮询的主体和processor的声明与实现
|--signal.rs //
|--threads.rs//实现用户空间多线程快速互斥锁（Futex）
|--task.rs //涉及TCB模块的声明以及方法的实现，以及rusage信号量的声明。
```

### 2.1.2 进程控制块

在 NPUcore 中，我们使用 TCB 这一名字代替 PCB 来同时用作进程控制块和线程控制块。

TCB 作为进程实体的一部分，记录了操作系统所需的，描述进程的当前情况以及管理进程运行的全部信息，是操作系统中最重要的数据结构。整个进程管理模块，正是围绕着 TCB 数据结构的生命周期来展开，通过控制 TCB 模块的生命周期来实现进程的创建，调度和回收。

**作为独立运行基本单位的标志：**当一个程序 (含数据) 配置了 TCB 后，就标志着它已经是一个能在多道程序环境下独立运行、合法的基本单位。系统是通过 TCB 感知进程的存在的，TCB 已成为进程存在于系统的唯一标志。

**能实现间断性运行方式：**当进程因阻塞而暂停运行时，他必须保留运行时的 CPU 现场信息，再次被调度运行时，还需要恢复运行时的 CPU 现场信息。在有了 TCB 后，系统将 CPU 现场信息保存在被中断进程的 TCB 中，供该进程再次被调度时恢复其 CPU 现场信息。由此可知，在多道程序环境下，传统意义上的程序无法保护自身运行环境，无法保证其执行结果的可再现性，失去运行意义。

**提供进程管理所需的信息：**当调度某进程运行时，只能根据该进程 TCB 只能记录的程序和数据的始址指针，来找到相应的程序和数据；在进程运行期间，当需要访问文件系统中的文件或 I/O 设备时，也需要借助 TCB 中的信息。可见，在进程的生命周期中，操作系统总是根据 TCB 中的信息来控制和管理进程。

**提供进程调度所需的信息：**只有处于就绪状态的进程才能被调度执行，而 TCB 就提供了该进程处于何种状态的信息。

**实现与其他进程的同步和通信**

在 NPUcore 中，任务控制块 TCB(Task Control Block) 将直接作为 PCB 进行使用。下图是 NPUcore 中进程控制块的具体组成。

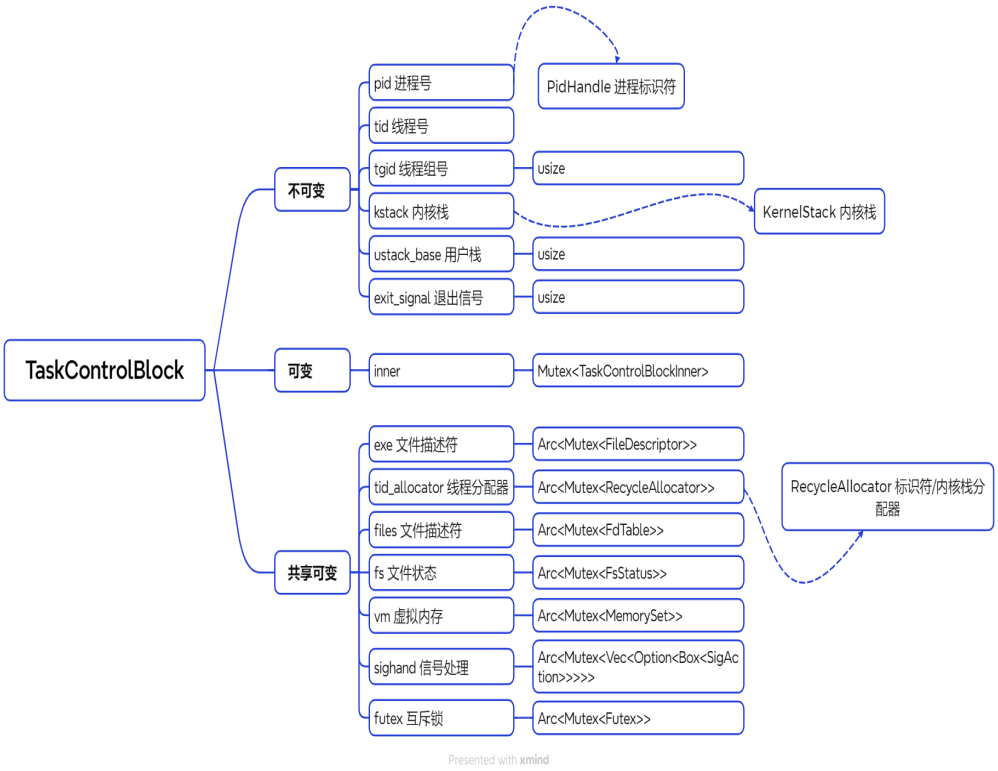



图 2-1　NPUcore 中进程控制块的具体组成

### 2.1.3 进程切换

NPUcore 进程的切换大致可以分为以下几个步骤：

1. 用户态陷入内核态（系统调用或中断）
2. 切换到调度器进程
3. 切换到新进程的内核线程
4. 从内核态返回

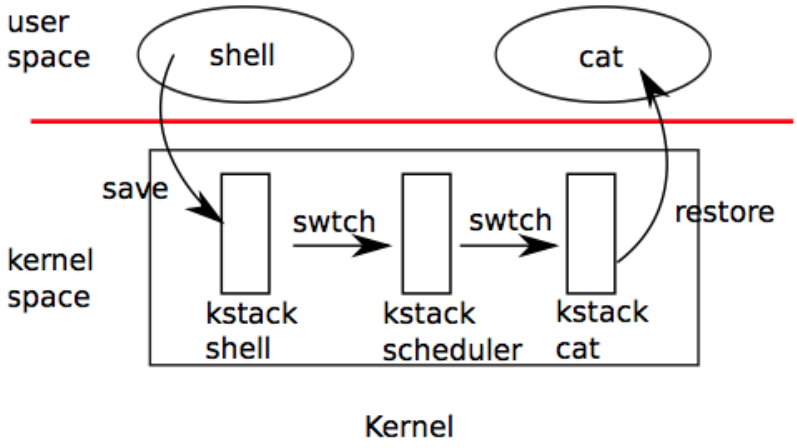



图 2-2　进程切换

为了在进程之间进行切换，NPUcore 需要在内核态执行两次上下文切换：从旧进程的内核线程切换到 CPU 的调度器线程，以及从调度器线程切换到新进程的内核线程。进程切换的核心**__switch** 并不了解线程，它只是简单地保存和恢复寄存器集合，即上下文。上下文是 CPU 保存的当前进程执行的状态。对于 RISC-V，进程上下文包括：**ra** 进程的返回地址、**sp** 进程内核栈指针、**s0-s11** 被调用者保存寄存器。

### 2.1.4 进程调度

NPUcore 所使用的调度策略为时间片轮转（RR），时间片轮转调度的基本思想是让每个线程在就绪队列中的等待时间与占用 cpu 的执行时间成正比例。其大致实现是：

1. 将所有的就绪线程按照 FCFS 原则，排成一个就绪队列。
2. 每次调度时将 cpu 分派（dispatch）给队首进程，让其执行一个时间片。
3. 在时钟中断时，统计比较当前线程时间片是否已经用完。

• 若用完，则调度器暂停当前进程的执行，将其送到就绪队列的队尾，并通过切换执行就绪队列的队首进程。

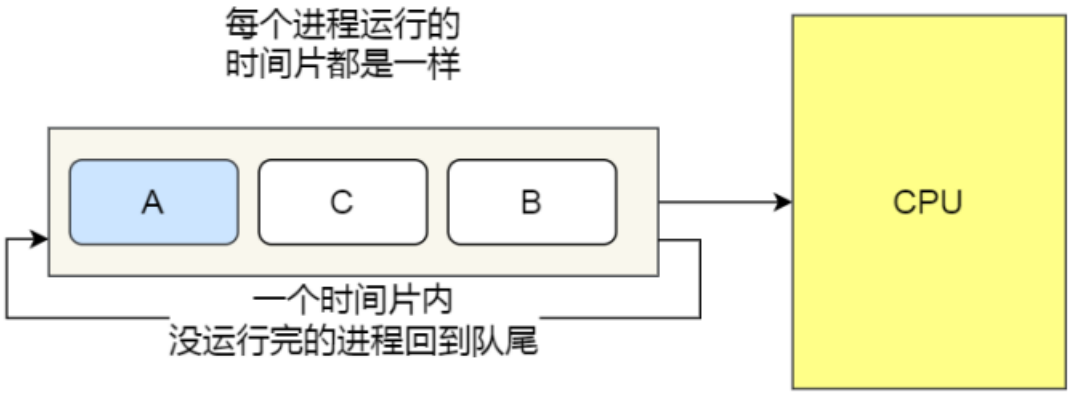
• 若没有用完，则线程继续使用。（2）NPUcore 进程调度的创新：NPUcore 团队在进行性能调优的过程中，发现操作系统运行示例程序时，IO 操作导致 cpu 挂起的性能损失很大，因此团队对调度器进行修改，使其支持阻塞式的进程调度模式。

阻塞式和非阻塞式 IO 是访问设备的两种模式，驱动程序可以灵活的支持两种 IO 模式。

• 阻塞操作指的是在执行设备操作时，如果得不到资源，那么进程就会挂起一直到满足可以操作的条件后再进行操作，被挂起的进程会进入睡眠状态，进入阻塞队列中，直到被唤醒。

• 非阻塞指的是不能进行设备操作时不进行挂起，要么一直等待，要么放弃处理机。

图 2-3　时间片轮转




采用阻塞式进程调度模式的好处是显而易见的，不能获取资源的进程将会被休眠，让出 CPU 供给其他进程，直到得到资源被唤醒，唤醒进程的代码于中断之中，因为在硬件获得资源的同时往往伴随着一个中断。

## 2.2 NPUcore 的内存管理

### 2.2.1 概述

在 NPUcore 中采取 SV39 分页模式，虚拟地址空间和物理地址空间均采用页式管理，且每个页面的大小为 4KiB ($2^{12}$B)。如此一来，一个虚拟页面中的数据正好对应存储在一个物理页帧上，便于管理。根据页面大小的规定可知，每个页面需要使用 12 位字节地址来进行页内索引。

### 2.2.2 虚拟地址与物理地址

#### (1) 地址的数据结构抽象

物理地址共有 56 位，这是由 RISC-V 的硬件设计人员决定的。但在 64 位的架构上，虚拟地址长度确实应该和位宽一致，为 64 位。不过在 SV39 分页模式下，虚拟地址只有低 39 位是有实际意义的。SV39 分页模式规定 64 位虚拟地址的高 25 位必须和第 38 位相同，否则内存管理单元（MMU）会直接认定它是一个不合法的虚拟地址。通过这个检查之后，MMU 再取出低 39 位尝试将其转化为一个 56 位的物理地址。同样，为了易于数据结构的实现，我们也将物理地址以 64 位进行封装。具体的实现如下：

代码片段 2.1　os/src/mm/address.rs

```
1     // Definitions
2     #[repr(C)]
3     #[derive(Copy, Clone, Ord, PartialOrd, Eq, PartialEq)]
4     pub struct PhysAddr(pub usize);
5
6     #[repr(C)]
7     #[derive(Copy, Clone, Ord, PartialOrd, Eq, PartialEq)]
8     pub struct VirtAddr(pub usize);
9
10    #[repr(C)]
```

```
    #[derive(Copy, Clone, Ord, PartialOrd, Eq, PartialEq)]
    pub struct PhysPageNum(pub usize);

    #[repr(C)]
    #[derive(Copy, Clone, Ord, PartialOrd, Eq, PartialEq)]
    pub struct VirtPageNum(pub usize);
```

上面分别给出了物理地址 PA、虚拟地址 VA、物理页号 PPN、虚拟页号 VPN 的类型声明，它们都是元组式结构体，可以看成 usize 的一种简单包装。我们刻意将它们各自抽象出不同的类型而不是都使用与 RISC-V 64 硬件直接对应的 usize 基本类型，是为了在 Rust 编译器的帮助下，通过多种安全且方便的类型转换来构建页表。

实现这些地址信息类型与 usize 类型之间的相互转换，需要使用 From<T> trait (同时实现了 Into<T> trait)。这里我们以 PPN 为例，介绍其与 usize 类型的转换（其余三种地址信息类型 PA、VA、VPN 的实现均一致，仅有类型声明的差异）。对 usize 类型实现以下 trait，使我们可以使用 usize 类型数据生成一个 PhysPageNum 类型的数据：

代码片段 2.2　os/src/mm/address.rs

```
impl From<usize> for PhysPageNum {
    fn from(v: usize) -> Self {
        Self(v)
    }
}
```

反过来，同样对 PhysPageNum 类型实现该 trait，使我们可以使用 PhysPageNum 类型数据生成一个 usize 类型的数据：

代码片段 2.3　os/src/mm/address.rs

```
impl From<PhysPageNum> for usize {
    fn from(v: PhysPageNum) -> Self {
        v.0
    }
}
```

至此，我们实现了地址信息类型与 usize 类型的相互转换。注意到，从地址信息变量（以 PPN 为例）得到它的 usize 类型的更简便方法是直接 ppn.0。

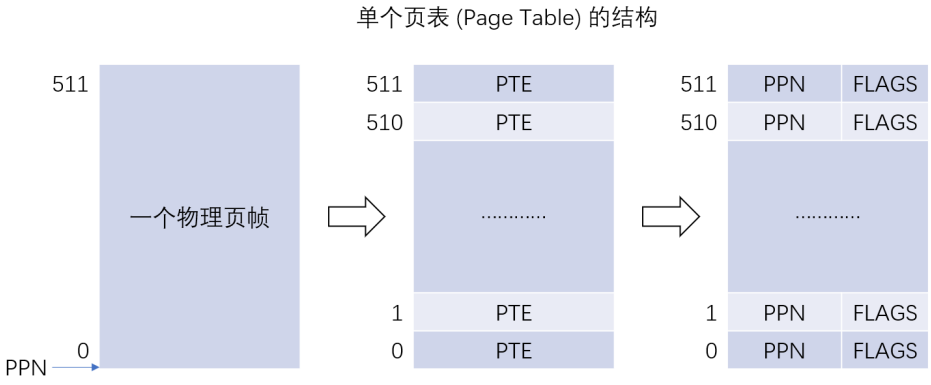
同时，我们也支持地址类型与页号类型的相互转换。需要注意的是，从页号到地址的转换只需左移 12 位即可；而地址转换至页号则必须保证它与页面大小对齐（即页内偏移为 0），若不对齐，则需要先进行取整。

### 2.2.3 页表

#### (1) SV39 三级页表结构

首先，在 SV39 模式下，一张页表正好占据一张物理页帧。由于一个页表项是 8 字节，因此每个页表需要保存 4KiB/8B=512 个页表项，这些页表项线性排列在页表内，如下图：

图 2-4　单个页表结构




NPUcore 中页表以三级的树结构组织在一起。树的根节点被称为一级页表节点，一级页表节点存储着二级页表节点的位置信息；同样二级页表节点存储着三级页表节点的位置信息；而三级页表节点是树的叶子节点，存储着最终虚拟页号所对应的物理页号。

### (2) 页表的数据结构抽象

SV39 模式下，每个页表恰好占据一个物理页帧的空间，因此每个页表可以用一个物理页号来标识。

代码片段 2.4　os/src/mm/page_table.rs

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

impl PageTable {
    pub fn new() -> Self {
        let frame = frame_alloc().unwrap();
        PageTable {
            root_ppn: frame.ppn,
            frames: vec![frame],
        }
    }
}
```

可见，PageTable 类型保存了其所在的物理页帧的页号作为其唯一标识。此外还有一个 frames 字段，该字段主要用于实现页表映射的物理页帧的生命周期与页表同步。

由于页表使用物理页号进行标识，因此我们已经可以方便地使用 PTE 来寻找对应的页表位置，即非叶子节点的页表中的 PTE 存储的 PPN，即为一个页表的 root_ppn。

## 2.2.4 虚实地址的映射过程

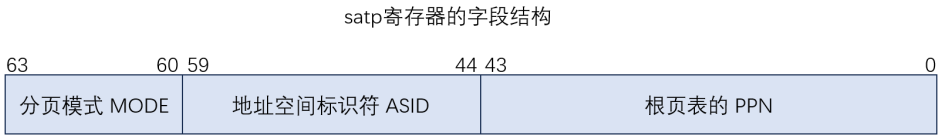
### (1) satp 寄存器

satp 寄存器存储了与分页模式有关的信息。默认情况下，内存管理单元 MMU 未被使能（启用），此时无论 CPU 位于哪个特权级，访存的地址都会作为一个物理地址交给对应的内存控制单元来直接访问。通过修改 S 特权级的 satp 寄存器可以启用分页模式，

在这之后 S 和 U 特权级的访存地址会被视为一个虚拟地址，它需要经过 MMU 的地址转换变为一个物理地址，再通过它来访问物理内存。

下面给出 NPUcore 中 satp 的字段分布，如下图：

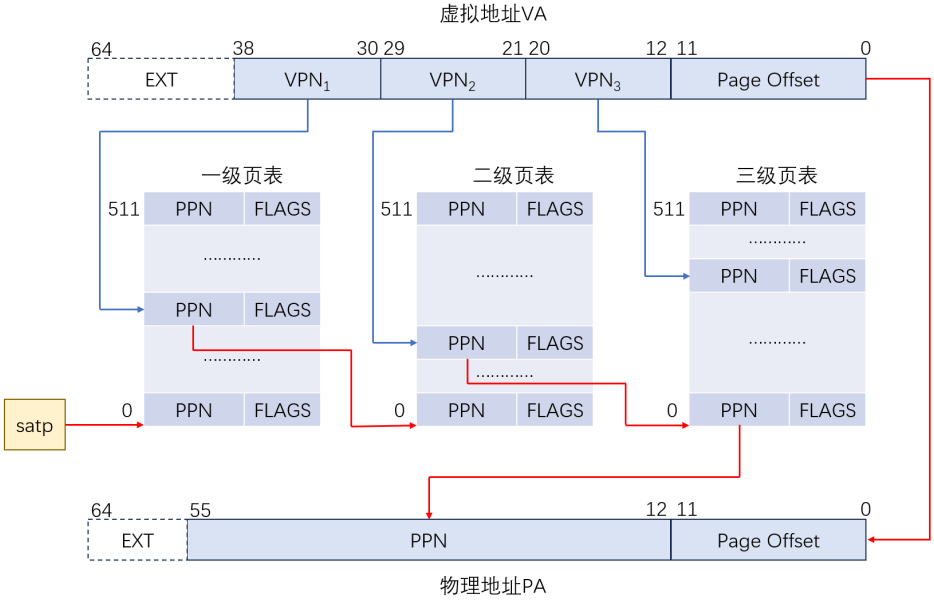
图 2-5　satp 寄存器的字段结构




### (2) 三级页表的使用流程

我们知道，SV39 模式中的虚拟页号为 27 位，而这其实是大有深意的。对于一个虚拟地址来说，将其 [38:12] 这 27 位的虚拟页号分为三个等长的部分，每个部分 9 位，即 [38:30] 为 $VPN_1$,[29:21] 为 $VPN_2$,[20:12] 为 $VPN_3$。这样，每个 $VPN_i$ 均能标识 $2^9$=512 个单位。事实上，每个虚拟页号的三个分段也正是对应着其在三级页表中的索引。所以，一个虚拟地址通过页表转化为物理地址的流程如下：

图 2-6　虚实地址转换流程




### (3) NPUcore 中的方法实现

在 NPUcore 中，我们使用 PageTable 类型的 translate_va 方法来实现虚拟地址到物理地址的转换：

代码片段 2.5　os/src/mm/page_table.rs

```
1    pub fn translate_va(&self, va: VirtAddr) -> Option<PhysAddr> {
2        self.find_pte(va.clone().floor()).map(|pte| {
3            let aligned_pa: PhysAddr = pte.ppn().into();
4            let offset = va.page_offset();
```

```
5               let aligned_pa_usize: usize = aligned_pa.into();
6               (aligned_pa_usize + offset).into()
7           })
8       }
```

该方法的核心实际上是调用了 find_pte 方法，完成从虚拟页号查询到叶子节点的对应 PTE 的过程。第 3 行是从最终查询到的 PTE 获取物理地址的 PPN 段，而 4、5、6 行实际上是完成了一个物理地址的拼接过程。map 是一个泛型闭包，将最后拼接而成的 usize 类型转化为 PhysAddr 类型。我们接下来看 find_pte 方法的实现：

代码片段 2.6　os/src/mm/page_table.rs

```
1       fn find_pte(&self, vpn: VirtPageNum) -> Option<&PageTableEntry> {
2           let idxs = vpn.indexes();
3           let mut ppn = self.root_ppn;
4           let mut result: Option<&PageTableEntry> = None;
5           for i in 0..3 {
6               let pte = &ppn.get_pte_array()[idxs[i]];
7               if !pte.is_valid() {
8                   return None;
9               }
10              if i == 2 {
11                  result = Some(pte);
12                  break;
13              }
14              ppn = pte.ppn();
15          }
16          result
17      }
```

由于 find_pte 是一个页表类型下的方法，其在调用时对应一个页表实例。而由于每个地址空间总是保存着其根页表的位置信息（satp 寄存器），因此该方法的调用者总是根页表，相当于我们已经实现了取得根页表的第一步。因此本方法实际就是实现对三级页表树的查询。

第 2 行，调用 indexes 方法：

代码片段 2.7　os/src/mm/address.rs

```
1       impl VirtPageNum {
2           pub fn indexes(&self) -> [usize; 3] {
3               let mut vpn = self.0;
4               let mut idx = [0usize; 3];
5               for i in (0..3).rev() {
6                   idx[i] = vpn & 511;
7                   vpn >>= 9;
8               }
9               idx
10          }
11      }
```

该方法将 VPN 切分为三份，返回一个长度为 3 的 usize 类型数组，即 $VPN_1$,$VPN_2$,$VPN_3$ 的信息。

第 3 行，我们获取当前页表的 root_ppn，是为了后续调用 get_pte_array 方法来取得页表中存储的 PTE。

从第 5 行开始，我们通过 3 次的 for 循环来进行三级页表的查询，每次循环均通过对应的 VPN 片段在页表中获取 PTE，然后判断该 PTE 的有效性以及其所在的页表是否位于叶子节点：若非叶子节点，则取出该 PTE 的 PPN 定位下一级页表；若为叶子节点，则直接返回当前 PTE。

至此，我们完成了虚实地址转换的实现。

### 2.2.5 内核地址空间

**地址空间（Address Space）**是一层抽象，在内核中建立虚实地址空间的映射机制，给应用程序提供一个基于地址空间的安全虚拟内存环境，让应用程序简单灵活地使用内存。这层抽象需要达成以下设计目标：

- **透明**：应用开发者可以不必了解底层真实物理内存的硬件细节，且在非必要时也不必关心内核的实现策略，最小化他们的心智负担；
- **高效**：这层抽象至少在大多数情况下不应带来过大的额外开销；
- **安全**：这层抽象应该有效检测并阻止应用读写其他应用或内核的代码、数据等一系列恶意行为。

启用分页模式下，内核代码的访存地址也会被视为一个虚拟地址并需要经过 MMU 的地址转换，因此我们也需要为内核对应构造一个地址空间，它除了仍然需要允许内核的各数据段能够被正常访问之后，还需要包含所有应用的内核栈以及一个跳板 (Trampoline)。

#### (1) 内核虚拟地址空间分布

在操作系统中，内核空间通常被划分为多个不同的区域，这些区域用于存储不同的数据结构和代码。以下是一些常见的内核空间划分：

- **代码段 (Code Segment)**：代码段用于存储内核代码，包括系统调用接口、中断处理程序、内核函数等。代码段通常被标记为只读，并且只能在内核编译时进行分配。
- **数据段 (Data Segment)**：数据段用于存储内核数据和变量，包括进程管理数据、内存管理数据、网络数据等。数据段通常可以被内核修改。
- **堆栈段 (Stack Segment)**：堆栈段用于存储进程的栈数据，包括函数调用堆栈、参数传递堆栈等。堆栈段通常被标记为只读，并且只能在内核编译时进行分配。
- **全局变量段 (Global Variable Segment)**：全局变量段用于存储全局变量和静态变量。全局变量段通常可以被内核修改。
- **中断向量表 (Interrupt Vector Table)**：中断向量表用于存储内核中的中断处理程序地址。中断向量表通常被标记为只读，并且只能在内核编译时进行分配。
- **内核引导段 (Kernel Boot Segment)**：内核引导段用于存储内核引导程序，用于启动操作系统。内核引导段通常被标记为只读，并且只能在内核编译时进行分配。

在内核编译时，通常会根据内核的功能和需求进行空间分配，以确保内核能够正确运行。不同的内核版本可能会有不同的空间分配策略，但通常需要考虑内存使用、代码和数据的可读性和可维护性等因素。

不同的内核有不同的地址空间划分，并找不到一种通用的划分，如下图 2-7，我们可以看到 NPUcore 的内核虚拟地址空间分布图：

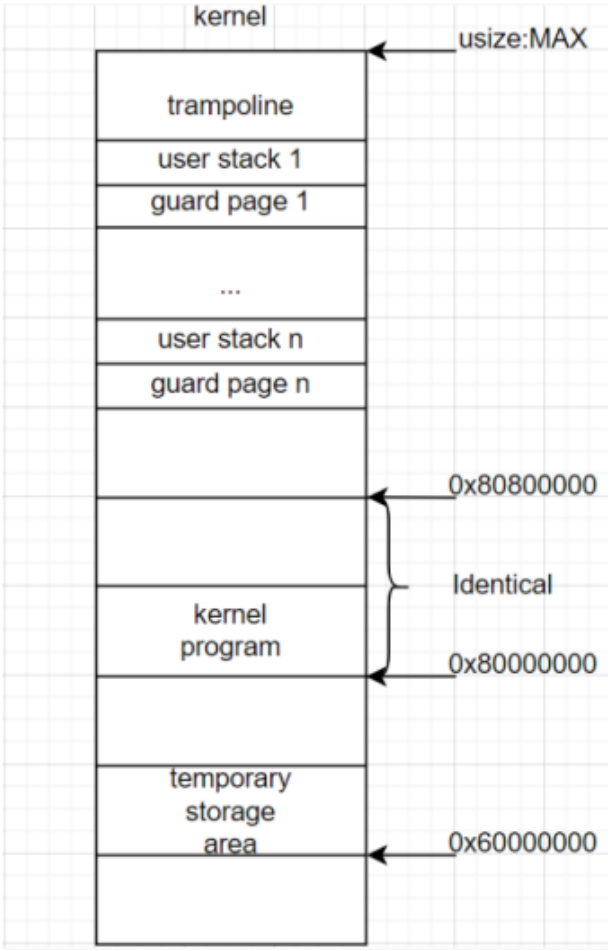



图 2-7　内核虚拟地址空间分布

- **trampoline**：跳板（trampoline），跳板页面的大小是 1 page，这个页面在地址空间的最高虚拟页面上。在内核空间中，跳板唯一的用途就是发生 panic。
- **user stack**：接下来则是从高到低放置用户栈，栈向下增长。每个 user stack 的大小为 1 page。**用户栈 (User Stack)** 是操作系统中的一种数据结构，用于存储用户程序执行期间所需的临时数据和指令。用户栈通常位于系统的虚拟内存中，是一个固定大小的数组，用于存储程序运行期间的临时数据和指令。在程序运行时，操作系统会为程序创建一个用户栈，用于存储程序执行期间的临时数据和指令。这些数据和指令包括程序运行时需要的临时变量、函数调用时的返回地址、参数等。当程序执行完毕后，操作系统会将该栈中的数据清除，以便为新的程序运行提供空间。用户栈的基本概念包括栈帧、栈空间、栈底和栈顶等，这些概念对于理解程序的

执行过程和程序的内存分布非常重要。需要注意的是这里的 userstack 并不专指用户栈，当某个进程处于 S 模式时，内核将使用此区域作为内核堆栈。

- **guard page**：相邻两个内核栈之间会预留一个保护页面 (Guard Page)，页面大小也是 1 page。这些页面位于堆栈页面之下。它们未被分配，这意味着如果内核使用这些区域中的数据，将捕获页面错误。它们保护内核不修改另一个进程的堆栈数据。用户栈的基本概念包括:

  1. 栈帧 (Stack Frame)：栈帧是用户栈中的数据结构，用于存储程序运行时所需的临时数据和指令。每个栈帧都包含一个返回地址 (Return Address) 和一个数据区 (Data Area), 其中数据区用于存储程序运行期间的临时数据和指令。
  2. 栈空间 (Stack Space)：栈空间是用户栈中的数据结构，用于存储程序运行时所需的临时数据和指令。栈空间的大小通常由操作系统分配和回收，程序运行时所需的栈空间大小由程序控制。
  3. 栈底 (Stack Bottom)：栈底是用户栈中的数据结构，用于表示当前栈中存储的数据和指令的起始地址。栈底通常是程序运行时的数据构，用于存储程序的返回地址和参数等。
  4. 栈顶 (Stack Top)：栈顶是用户栈中的数据结构，用于表示当前栈中存储的数据和指令的结束地址。栈顶通常是程序运行时的数据结构，用于存储程序的返回地址和参数等。
  5. 入栈 (Push)：入栈是将数据或指令压入用户栈中的数据结构中。入栈的操作通常用于将数据或指令存储到栈中，以便程序在执行时能够访问这些数据或指令。
  6. 出栈 (Pop)：出栈是将栈中的数据或指令弹出到程序的数据结构中。出栈的操作通常用于将数据或指令从栈中弹出，以便程序在执行时能够访问这些数据或指令。

  由于编译器会对访存顺序和局部变量在栈帧中的位置进行优化，我们难以确定一个已经溢出的栈帧中的哪些位置会先被访问，但总的来说，保护页面大小被设置的越大，我们就能越早捕获到这一可能覆盖其他重要数据的错误异常。由于我们的内核非常简单且内核栈的大小设置比较宽裕，在当前的设计中我们仅将其大小设置为单个页面。
- **kernel program**：这里将加载内核程序。注意到这里有 identical 标识，表明这段空间是恒等映射，之后会在内核物理空间详细介绍。
- **temporary storage area**：临时存储区域，这个区域最多占据 512 pages。这个区域现在用于 exec 系统调用，这意味着文件将首先加载到这里。

### (2) 内核物理地址空间分布

内核物理地址空间分布通常被划分为多个不同的区域，这些区域用于存储不同的数据结构和代码。例如，代码段、数据段、堆栈段、全局变量段等，这些区域的大小和位置都会在内核编译时进行分配。在内核物理地址空间分布中，还有一些特殊的区域，如内核引导段和中断向量表等，这些区域用于存储内核引导程序和中断处理程序的地址。

在 Memoryset 的 new_kernel() 方法中，首先 map 了 trampoline（这里不是恒等映射，trampoline 虽然在虚拟地址的最顶端，但是物理地址并没有到那么高），由 anonymous_-identical_map! 匿名恒等映射宏可知，接下来恒等映射内核的四个逻辑段.text .rodata .data 和.bss。最后恒等映射 physical memory 和 memory-mapped registers。

之所以使用恒等映射到物理内存的方法，是因为这能够使得我们在无需调整内核内存布局 os/src/linker-xxx.ld 的情况下，仍能像启用页表机制之前那样访问内核的各个段。

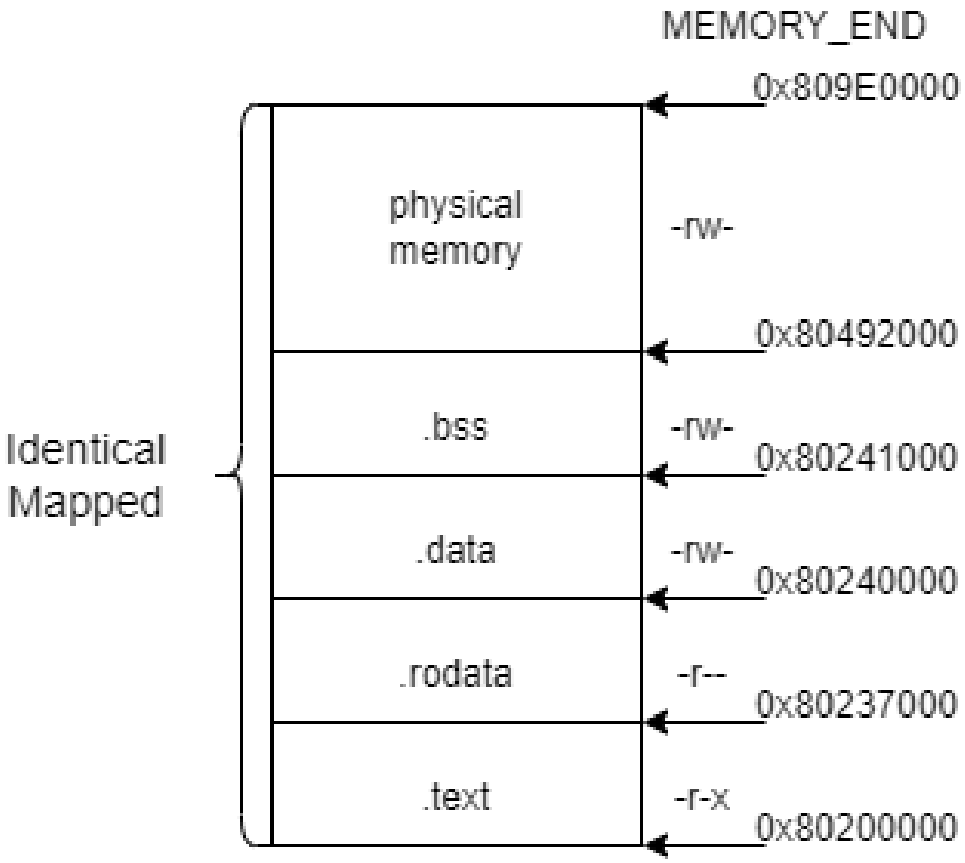



图 2-8　逻辑区域的物理地址空间分布

需要特别说明的是，这里画出的只是在 qemu 平台上的地址空间分布，在 k210 或 fu740 上则不一样。一是因为上　提到 k210 的 BASE_ADDRESS 为 0x80020000，不同于 qemu 的 0x80200000 。二是因为 MEMORY_END 在不同平台上也不相同，具体如下：

代码片段 2.8　config.rs

```
1    //os/src/config.rs
```

```
    #[cfg(all(not(feature="board\_k210"), not(feature="board\_fu740")))]
    pub const MEMORY_END: usize = 0x809e_0000;
    #[cfg(feature = "board_k210")]
    pub const MEMORY_END: usize = 0x8080_0000;
    #[cfg(feature = "board_fu740")]
    pub const MEMORY_END: usize = 0x9000_0000;
```

### (3) 内存映射空间

**MMIO**（Memory-mapped I/O，即内存映射 I/O），通过将外围设备映射到内存空间，便于 CPU 的访问。使用这种处理方式，设备控制寄存器只是内存中的变量，可以和其他变量一样进行寻址，无需特殊的 I/O 指令（如 IN，OUT）来读写设备控制器。而且这种　式可以将设备的读取权限管理同区域读取权限管理结合起来，这个区域没有 U 权限标志，那么就只可以在内核态被访问，无需特殊的保护机制。

### (4) 地址空间实现

首先，从数据结构入手。地址空间由页表和一系列内存映射区域构成。在 NPUcore 中，页表可以直接创建为 PageTable 对象，而内存映射区域需要使用一个 MapArea 的向量（vec）来表示。这里我们主要关注 MapArea 的实现：

代码片段 2.9　MapArea

```
    #[derive(Clone)]
    /// Map area for different segments or a chunk of memory for memory
        mapped file access.
    pub struct MapArea {
        /// Range of the mapped virtual page numbers.
        /// Page aligned.
        /// Map physical page frame tracker to virtual pages for RAII &
            lookup.
        inner: LinearMap,
        /// Direct or framed(virtual) mapping?
        map_type: MapType,
        /// Permissions which are the OR of RWXU, where U stands for user.
        map_perm: MapPermission,
        pub map_file: Option<Arc<dyn File>>,
    }

    #[derive(Copy, Clone, PartialEq, Debug)]
    pub enum MapType {
        Identical,
        Framed,
    }
```

MapPermission 表示映射权限，MapType 表示映射方式，分为直接映射（Identical）和间接映射（Framed）。inner 是一种 LinearMap 类型，表示 MapArea 所映射到的内存区域。

VPNRange 表示虚拟页号的范围，通过 VPNRange 可以了解到该内存区域的大小。frames 是枚举类型 Frame 的向量。

接下来分析几个 MemorySet 接口下常用方法：

这里给出该函数下调用的其他与 MemorySet 相关的函数：

代码片段 2.10　内核创建

```
MemorySet
├── new_bare()// 新建一个空的地址空间
├── new_kernel()// 新建一个空的内核空间，在前面介绍过
├── map_trampoline()// 映射跳板区域
├── push()// 将MapArea push 到MemorySet 中
└── MapArea
├── map_one()// 映射页面前的检查，检查完调用下面的方法
└── map_one_unchecked()// 映射页面的具体实现
```

在整个代码中，对 NPUcore 地址空间的实现进行了清晰的分层和结构化设计，采用了合理的数据结构和接口设计，实现了对虚拟地址空间的灵活管理。

### 2.2.6 内存管理相关数据结构

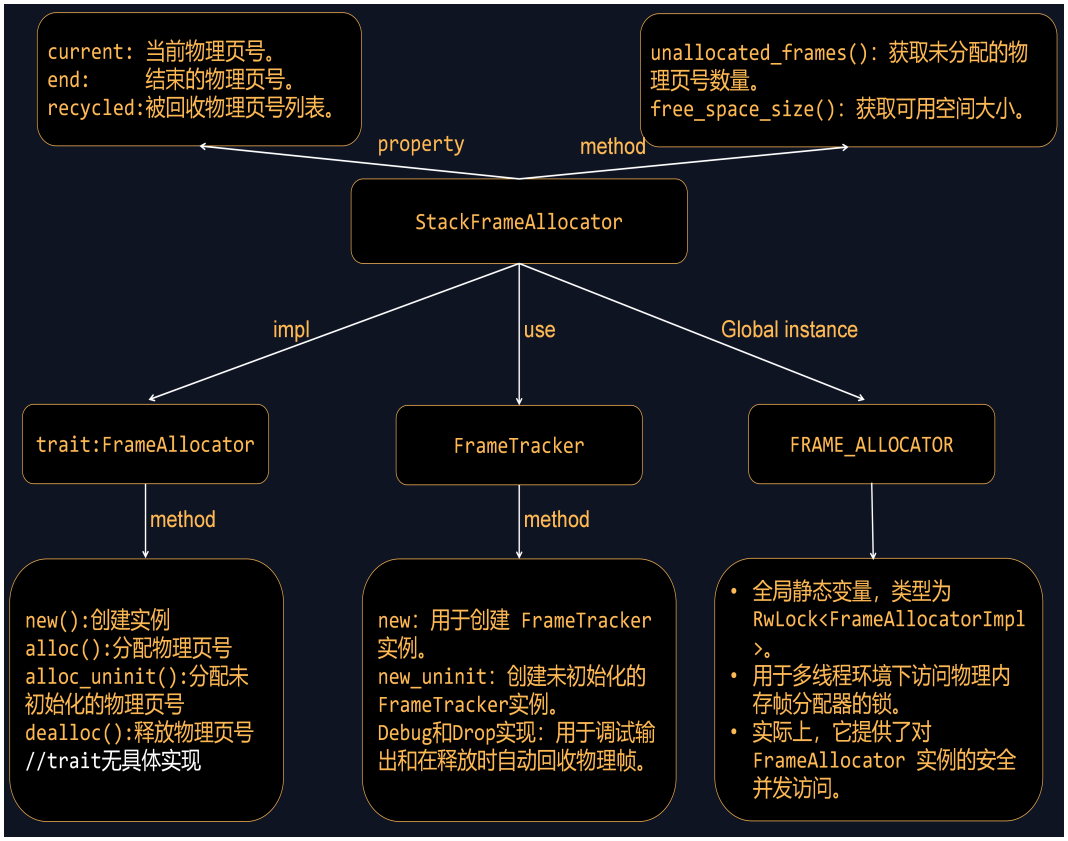



图 2-9　NPUcore 中内存分配涉及的数据结构

**FrameAllocator trait：**这是一个特性，定义了管理物理内存帧的标准行为。它规定了创建、分配、释放物理内存帧的方法。

**FrameTracker 结构体：**用于表示单个物理内存帧的状态，包括物理页号（PhysPageNum）以及一些操作方法。这些方法可能涉及物理内存的初始化、输出调试信息和回收等，用于创建和管理单个物理内存帧的状态。

**FRAME_ALLOCATOR 全局静态变量：**这是用于多线程环境下访问物理内存帧分配器的锁。它提供了对 FrameAllocator 实例的安全并发访问。

**StackFrameAllocator 结构体：**实现了 FrameAllocator 这个 trait，用于管理物理内存帧。它采用基于栈的策略来管理物理内存，实现了创建全局变量 FRAME_A LLOCATOR 的功能。除此之外，这个结构体还涉及到调用 FrameTracker 中的方法，这些方法用于对物理页面进行操作，如初始化、调试输出和回收等。

（1）物理内存空间

要想有效的管理物理内存，首先要清楚我们管理的内存空间范围。在 NPUcore 中，在三个平台上，物理内存的起始物理地址 MEMORY_START 均为 0x80000000，单个页面大小 PAGE_SIZE 均为 0x1000，即 4096 字节。

在 k210 上,我们硬编码整块物理内存的终止物理地址 MEMORY_END 为 0x80800000，这意味着可用内存大小为 8MiB 。

在 fu740 上，MEMORY_END 为 0x9000_0000，可用内存大小为 256MiB 。

如果没有在这两个平台上 (也就是在 qemu 模拟器上)，MEMORY_END 被设置为 0x809e_0000，可用内存大小将近 10MiB 。

（2）物理内存分配器

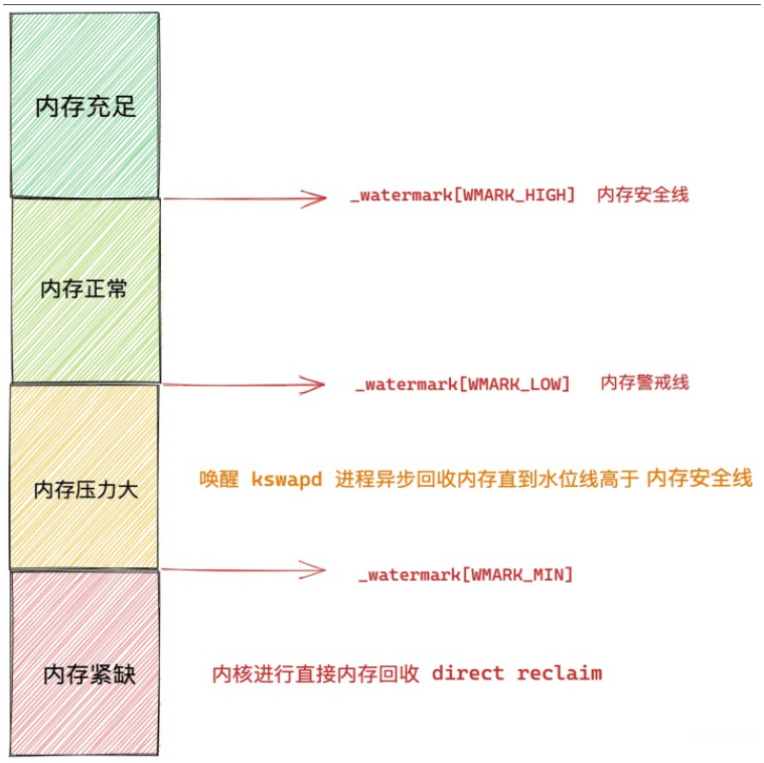



图 2-10　物理内存分配器示意图

物理内存分配器会根据一定的算法进行内存释放，以腾出物理内存资源，然后从内存池中提取所需的物理内存，并将其分配给程序运行所需的页面。

### (3) 物理内存分配器接口

接口描述: 物理内存分配器的接口包括一个自身的 new() 方法，以及实现物理页面的分配和回收。需要注意的是，这里有一个未初始化的页面分配方法 alloc_uninit()，省去初始化操作将会缩短分配时间。

### (4) 全局物理内存分配器

全局物理内存分配器是一种用于分配和释放物理内存的机制，用于解决程序在运行时的内存分配和释放问题。在计算机程序中，内存分配和释放通常是由不同的进程或线程进行的，这可能会导致内存泄漏和其他问题。

NPUcore 通过如下方式实现全局物理内存分配器。

首先创建一下 StackFrameAllocator 的全局实例 FRAME_ALLOCATOR ：

```
// os\src\mm\frame_allocator.rs
type FrameAllocatorImpl = StackFrameAllocator;
lazy_static!{
    pub static ref FRAME_ALLOCATOR:RwLock<FrameAllocatorImpl>=
    RwLock::new(FrameAllocatorImpl::new());
}
```

通过全局物理内存分配器的包装，在内核其他模块的视角下，申请一个物理页面会返回 FrameTracker ， 当 FrameTracker 生命期结束，物理页面也就被自动回收，体现了 RAII 的思想。

### (5) 内存分配机制

在物理内存分配方法中，一种常用并且简单有效的管理策略为栈式内存管理。它不但存取速度快，而且数据存取操作十分简单，表示操作也比较容易，存储空间可用性较大，占用存储空间也小，有很强的数据抽象能力，程序代码表示简洁，工作方便。

栈式管理，数据的读写只能从一端进行，遵循后进先出的原则，与堆放木柴一样，最后放进去的木柴在上面，是下次最先取出的一块木柴。

### (6) 物理页帧分配

核心为物理页帧的分配和回收，先看物理页帧的分配：

在分配 alloc 的时候，首先会检查栈 recycled 内有没有之前回收的物理页号，如果有的话弹出栈顶，使用 into 方法将 usize 转换成了物理页号 PhysPageNum ， 构造 frame_tracker 返回。否则检查 current == end ， 若相等则表示内存耗尽，没有空闲页面，返回 None 。

```
// os\src\mm\frame_allocator.rs
impl FrameAllocator for StackFrameAllocator {
    fn alloc(&mut self) -> Option<FrameTracker> {
        if let Some(ppn) = self.recycled.pop() {
            let frame_tracker = FrameTracker::new(ppn.into());
            log::trace!("[frame_alloc] {:?}",frame_tracker);
            Some(frame_tracker)
        } else if self.current == self.end {
```

```
                None
            } else {
                self.current += 1;
                let frame_tracker = FrameTracker::new((self.current - 1).
                    into());
                log::trace!("[frame_alloc]{:?}",frame_tracker);
                Some(frame_tracker)
            }
        }
    }
```

若不等说明之前从未分配过的物理页号区间还有剩余，可以在 [ current , end ) 上进行分配，我们分配它的左端点 current (即从低地址向高地址分配)，同时将管理器内部维护的 current 加 1 代表 current 已被分配了。同样将 usize 转换成了物理页号，构造 frame_tracker 返回。

之前提到过 FrameAllocator 有一个不初始化分配方法 alloc_uninit，二者的不同其实也就是在 FrameTracker::new 的时候，在 alloc 中使用的是 FrameTracker::new 这个有初始化的方法，在 alloc_uninit 中使用的是 FrameTracker::new_uninit 这个不初始化的方法。

#### (7) 物理页帧回收

物理页帧的回收是整个内存管理中一个重要的环节，它负责释放不再被使用的页面以供后续分配使用。让我们来看一下物理页面的回收机制以及其实现的过程。

```
    // os\src\mm\frame_allocator.rs
    // Deallocate a physical page
    fn dealloc(&mut self,ppn:PhysPageNum){
        log::trace!("[frame_dealloc] {:?}", ppn);
        let ppn = ppn.0;
        // validity check, note that this should be unnecessary for RELEASE
            build and it
        if option_env!("MODE") == Some("debug") && ppn >= self .current
        self .recycled.iter().find(|&v|*v == ppn).is_some()
        {
            panic!("Frame ppn={:#x} has not been allocated!",ppn);
        }
        // recycle
        self.recycled.push(ppn);
    }
```

实际的页面回收机制非常简单，它仅将要回收的页面号 ppn 压入回收页面的 recycled 向量中。这个向量用于存储可供重新分配的物理页面。物理页面的回收是内存管理中重要的一环，通过这个机制，系统可以及时释放不再使用的页面，并通过有效性检查和回收流程来确保内存的正确性和可靠性。

## 2.3 NPUcore 的文件系统

### 2.3.1 概述

对于 Fat32 文件系统模块，我们的主要目标是使该模块结构合理清晰，实现简单，功能符合大赛要求，最主要的拥有良好的可扩展性。目前，NPUcore 的文件系统模块的

总体已经完成还有一些细节和 BUG 需要完善。

(1) 虚拟文件系统及接口

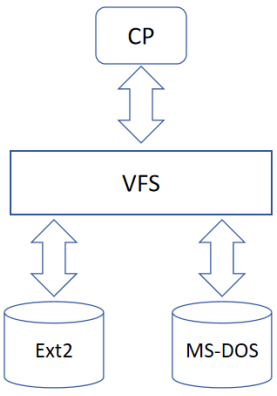
NPUcore 的文件系统模块借鉴了 Linux 中 VFS（虚拟文件系统）的设计，使得 NPUcore 可以支持多种文件系统我们对文件系统和文件分别定义了统一的接口，open，write，read，mkdir 等系统调用的处理函数可以通过这些接口来对具体的文件系统和文件进行操作，而不需要考虑各个文件系统的实现细节。如果需要让 NPUcore 支持新的文件系统，只需要为这个文件系统以及这个文件系统的文件实现统一的接口即可。文件系统和文件接口是以 trait 的方式定义的一共有两类抽象接口：1. 文件系统抽象接口：FileSystem trait，每一个文件系统都需要实现该 trait 该接口非常简单，主要的成员只有 root_inode，通过该接口可以获得文件系统的根索引。2. 文件抽象接口：如果要实现 File trait，必须先实现其依赖的父 trait：Inode trait，而 Inode trait 要由要求实现 InodeDevice，InodeDevice 又要求注册的设备需要满足块设备或者字符设备的抽象，于是就有了虚拟文件系统的层级结构。

(2) Fat32 文件系统

NPUcore 支持了竞赛要求的 FAT32 文件系统，并将 FAT32 系统接入 VFS 中。

### 2.3.2 虚拟文件系统

虚拟文件系统（Virtual File System，简称 VFS）也可称为虚拟文件转换，是一个内核软件层，用来处理与 Unix 标准文件系统相关的所有系统调用。它为用户程序提供文件和文件系统操作的统一接口，屏蔽不同文件系统的差异和操作细节。借助 VFS 可以直接使用 open()、read()、write() 这样的系统调用操作文件，而无须考虑具体的文件系统和实际的存储介质，极大简化了用户访问不同文件系统的过程。另一方面，新的文件系统、新类型的存储介质，可以无须编译的情况下，动态加载到内核中。




```
inf = open("/floppy/TEST", O_RONLY, o);
outf = open("/tmp/test",
O_WRONLY|O_CREAT|O_TRUNC, 0600);
do {
    i = read(inf, buf, 4096);
    write(outf, buf, i);
}while(i);
close(outf);
close(inf);
```

图 2-11　VFS 在文件复制

VFS 的思想是把不同类型文件的共同信息放入内核，具体思路是通过在用户进程和文件系统之间引入了一个抽象层。用户可以通过这个抽象层的接口自由使用不同的文件系统，而新的文件系统只需要支持这些接口就能直接加载到内核中使用。

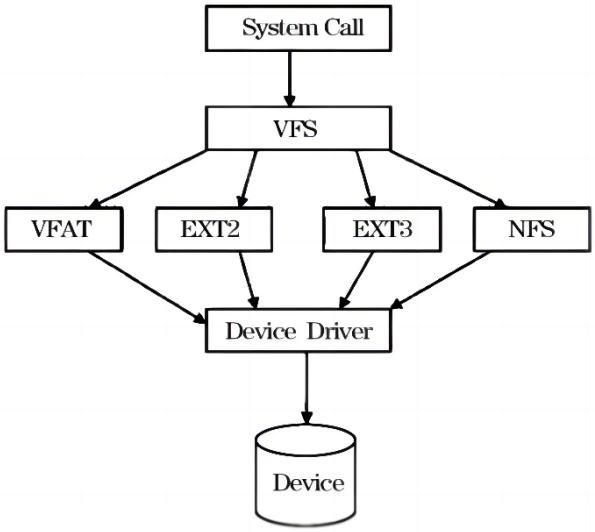



图 2-12　VFS 在 OS 结构中

**(1) 虚拟文件系统的组成**

为了实现对于不同文件系统的抽象，虚拟文件系统则需要通过数据结构完成对于不同文件系统的统一描述。在 linux 中为了实现这一点，定义了以下内容：

- 超级块 (super block) 超级块用于存储已安装的文件系统的相关信息。因此一个超级块可代表一个文件系统。文件系统的任意元数据修改都要修改超级块。超级块对象是常驻内存并被缓存的。该列表以链式方式维护在内存中，为所有进程可见。
- 目录项 目录项模块，管理路径的目录项，存储这个目录下的所有的文件的 inode 号和文件名等信息。其内部是树形结构，操作系统检索一个文件，都是从根目录开始，按层次解析路径中的所有目录，直到定位到文件。
- inode 存放具体文件的一般信息（内核在操作文件或目录时需要的全部信息）。一个索引节点代表文件系统中的一个文件，但是索引节点仅当文件被访问时，才在内存中创建
- 文件对象 它代表由进程打开的文件。存放打开文件与进程之间进行交互的有关信息。这些信息仅当进程访问文件期间存放在内核中。这类信息仅当进程访问期间存在于内核内存中。文件对象（不是物理文件）由相应的 open() 系统调用创建，由 close() 系统调用撤销。

在 NPUcore 中，各部分由相应的数据结构实现。

对于超级块，NPUcore 中在虚拟文件系统中定义了一个 filesystem 结构体，用来存储文件系统的信息。这个结构体较为简单。由于 NPUcore 暂时只支持 FAT32 文件系统，

所以文件系统的类型的枚举类型只有两个值。具体代码如下：

代码片段 2.11　filesystem

```
pub enum FS {
    Null,
    Fat32,
}

pub struct FileSystem {
    pub fs_id: usize,
    pub fs_type: FS,
}
```

对于目录项和 inode，NPUcore 使用了 DirectoryTreeNode 结构体。不难发现，这里既有相关目录的下文件的部分，又有相关文件的信息。定义如下：

代码片段 2.12　DirectoryTreeNode

```
pub struct DirectoryTreeNode {
    /// If this is a directory
    /// 1. cwd
    /// 2. mount point
    /// 3. root node
    /// If this is a file
    /// 1. executed by some processes
    /// This parameter will add 1 when opening
    spe_usage: Mutex<usize>,
    name: String,
    filesystem: Arc<FileSystem>,
    file: Arc<dyn File>,
    selfptr: Mutex<Weak<Self>>,
    father: Mutex<Weak<Self>>,
    children: RwLock<Option<BTreeMap<String, Arc<Self>>>>,
}
```

从结构可以看出，这里实现了一个树形结构，指向子节点并带有路径，同时还有指向文件和文件系统的 Arc 指针。

对于文件对象，使用了文件描述符 file_description。定义如下：

代码片段 2.13　DirectoryTreeNode

```
#[derive(Clone)]
pub struct FileDescriptor {
    cloexec: bool,
    nonblock: bool,
    pub file: Arc<dyn File>,
}
```

### (2) 虚拟文件系统提供的接口

- sys_openat:

```
pub fn sys_openat(dirfd: usize, path: *const u8, flags: u32,
    mode: u32)
```

该接口接收来自用户空间的参数，包括目录描述符 dirfd、路径 path、打开标志位 flags 和文件权限 mode。它会根据传入的目录描述符选择要打开的文件描述符，并通过文件描述符的 open() 方法尝试打开文件。如果打开失败，则返回相应的错误码。然后，函数将新打开的文件描述符插入到当前任务的文件描述符表中，并返回新文件描述符的整数值

- sys_close:

```
pub fn sys_close(fd: usize)
```

该接口会将进程控制块中的文件描述符表对应的一项改为 None ，代表它已经空闲，同时这也会导致内层的引用计数类型 Arc 被销毁，会减少一个文件的引用计数，当引用计数减少到 0 之后文件所占用的资源就会被自动回收。

- sys_read:

```
pub fn sys_read(fd: usize, buf: usize, count: usize)
```

该接口根据文件描述符 fd 在文件描述符表中找到相应的文件描述符对象，使用文件描述符的 read_user() 方法尝试从文件中读取数据到用户空间的缓冲区 buf 中，读取 count 个字节。

- sys_write:

```
pub fn sys_write(fd: usize, buf: usize, count: usize)
```

该接口根据文件描述符 fd 在文件描述符表中找到相应的文件描述符对象，使用文件描述符的 write_user() 方法尝试从文件中写入数据到用户空间的缓冲区 buf 中，共写 count 个字节。

- sys_fstat:

```
pub fn sys_fstat(fd: usize, statbuf: *mut u8)
```

该接口根据文件描述符 fd 在文件描述符表中找到相应的文件描述符对象，并根据文件描述符提供的方法将文件信息写入缓冲区 statbuf 中。

- sys_mount:

```
pub fn sys_mount(source: *const u8,target: *const u8,
    filesystemtype: *const u8,mountflags: usize,data: *const u8
    ,)
```

该接口实现了挂载，参数中 source 为要挂载的文件系统的源路径或标识,target 为文件系统将要挂载到的目标位置,filesystemtype 为要挂载的文件系统的类型,mountflags 是挂载选项和标志,data 为挂载所需的其他数据。

- sys_lseek:

```
pub fn sys_lseek(fd: usize, offset: isize, whence: u32)
```

该接口根据文件描述符 fd 在文件描述符表中找到相应的文件描述符对象, 然后以 whence 为偏移的基准，offset 为偏移量，返回操作后的文件指针位置。

- `sys_mkdirat`:

```
pub fn sys_mkdirat(dirfd: usize, path: *const u8, mode: u32)
```

该接口在指定路径 dirfd 下创建路径为 path 的目录。

### 2.3.3 NPUcore 中目录树的数据结构及具体实现

NPUcore 中具体实现方式是通过文件目录树来实现的，目录树 (Directory Tree) 的有根树数据结构如下所示。

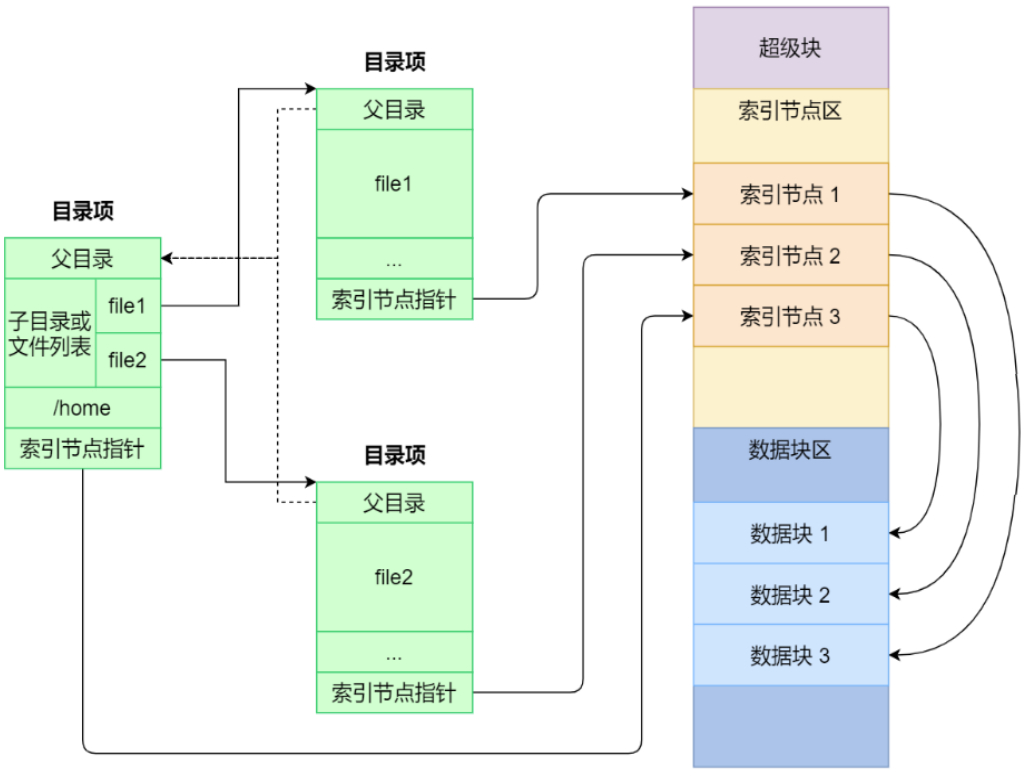



图 2-13　目录树的有根树数据结构

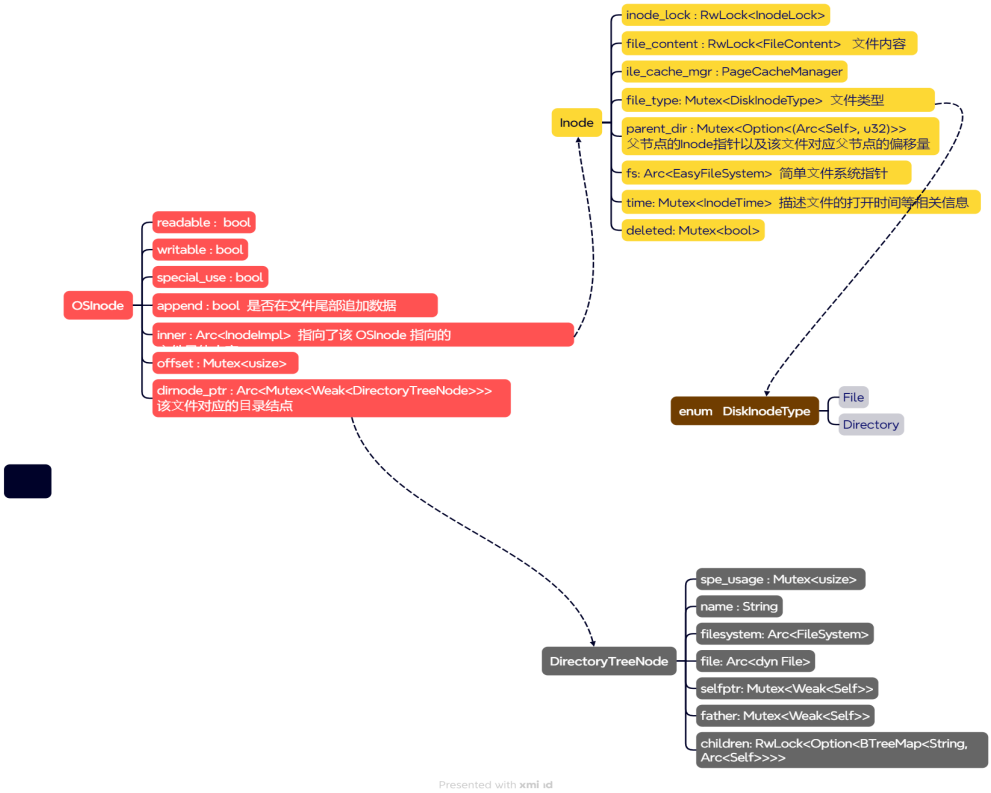



图 2-14　NPUcore 中数据结构之间的联系

在 NPUcore 中，对于目录同样是以 inode 的形式存储在磁盘中，不过为了更方便对文件进行各种操作，我们使用目录树结构来进行对文件的定位。一方面我们可以使用 OSInode 来寻找其所在的目录指针 dirnode_ptr，另一方面可以通过 directoryTreeNode 来找到其对应的文件（夹）file 。

```
// os/src/fs/fat32/directory_tree.rs
pub struct DirectoryTreeNode {
    /// If this is a directory
    /// 1. cwd
    /// 2. mount point
    /// 3. root node
    /// If this is a file
    /// 1. executed by some processes
    /// This parameter will add 1 when
    opening
    ///
    pub nlink_count : Mutex<u32>,
    在通过文件树寻找制定文件的过程中，首先会判断文件的路径前缀
    是否在路径缓存中，这样使大量文件操作效率更高，同时
    NPUcore也对一些默认的路径进行了路径的转化。
    文件描述符层
    spe_usage: Mutex<usize>,
    name: String,
    filesystem: Arc<FileSystem>,
    file: Arc<dyn File>,
    selfptr: Mutex<Weak<Self>>,
    father: Mutex<Weak<Self>>,
```

```
        children: RwLock<Option<BTreeMap<String,
        Arc<Self>>>>,
    }
```

```
    // src/os/fs/fat32/inode.rs
    pub struct OSInode {
        //nlink_count : Mutex<usize>,
        readable: bool,
        writable: bool,
        special_use: bool,
        append: bool,
        inner: Arc<InodeImpl>,
        offset: Mutex<usize>,
        dirnode_ptr:
        Arc<Mutex<Weak<DirectoryTreeNode>>>,
    }
```

内核全局维护了一个全局的目录节点向量 DIRECTORY_VEC，记录当前系统所在的目录：

```
    static ref DIRECTORY_VEC: Mutex<
    (Vec<Weak<DirectoryTreeNode>>, usize)> = Mutex::new((Vec::new(), 0));
```

同时内核记录了根目录节点 ROOT，它的目录是空字符串:

```
    pub static ref ROOT: Arc<DirectoryTreeNode> = {
        let inode = DirectoryTreeNode::new(
        "".to_string(),
        Arc::new(FileSystem::new(FS::Fat32)),
        OSInode::new(InodeImpl::root_inode(&FILE_SYSTEM)),
        Weak::new()
        );
        inode.add_special_use();
        inode
    };
```

在通过文件树寻找制定文件的过程中，首先会判断文件的路径前缀是否在路径缓存中，这样使大量文件操作效率更高，同时 NPUcore 也对一些默认的路径进行了路径的转化。

### 2.3.4 文件描述符层

为简化操作系统设计实现，可以让每个进程都带有一个线性的 **文件描述符表**，记录该进程请求内核打开并读写的那些文件集合。而 **文件描述符** (File Descriptor) 则是一个非负整数，表示**文件描述符表**中一个打开的**文件描述符**所处的位置（可理解为数组下标）。进程通过**文件描述符**，可以在自身的**文件描述符表**中找到对应的文件记录信息，从而也就找到了对应的文件，并对文件进行读写。当打开（open ）或创建（create ）一个文件的时候，一般情况下内核会返回给应用刚刚打开或创建的文件对应的文件描述符；而当应用想关闭（close ）一个文件的时候，也需要向内核提供对应的**文件描述符**，以完成对应文件相关资源的回收操作。

因为 OSInode 也是一种要放到进程**文件描述符表**中文件，并可通过 sys_read/write 系统调用进行读写操作，因此我们也需要为它实现 File Trait：

```
    fn readable(&self) -> bool {
        self.readable
    }
    fn writable(&self) -> bool {
        self.writable
    }
    fn read(\&self, offset: Option<\&mut usize>, buffer: \&mut [u8]) ->
        usize {
        match offset {
            Some(offset) => {
                let len =
                self.inner.read_at_block_cache(*offset,
                buffer);
                *offset += len;
                len
            }
            None => {
                let mut offset =
                self.offset.lock();
                let len =
                self.inner.read_at_block_cache(*offset,
                buffer);
                *offset += len;
                len
            }
        }
    }

    fn write(\&self, offset: Option<\&mut
    usize>, buffer: \&[u8]) -> usize {
        match offset {
            Some(offset) => {
                let len =
                self.inner.write_at_block_cache(*offset,
                buffer);
                *offset += len;
                len
            }
            None => {
                let mut offset =
                self.offset.lock();
                let inode_lock =
                self.inner.write();
                if self.append {
                    *offset =
                    self.inner.get_file_size_wlock(\&inode_lock)
                    as usize;
                }
                let len = self
                .inner
                .write_at_block_cache_lock(\&inode_lock,
                *offset, buffer);
                *offset += len;
                len
            }
```

```
55          }
56      }
```

read 将数据从文件读取到提供的缓冲区中。offset 参数是一个可选的 usize 的可变引用，允许指定开始读取的偏移量。如果提供了 Some(offset)，则从指定的偏移量开始读取，并相应地更新偏移量。如果提供了 None，该方法锁定偏移量，使用当前偏移量从文件中读取数据，并相应地更新偏移量。

write 将提供的缓冲区中的数据写入文件。与 read 方法类似，它接受一个可选的 usize 的可变引用作为偏移量参数。如果提供了 Some(offset)，则从指定的偏移量开始写入，并相应地更新偏移量。如果提供了 None，该方法锁定偏移量，获取文件 inode 的写锁，然后将数据写入文件。如果设置了 append 标志，它在写入之前将偏移量更新为文件的末尾。

同时，在 NPUcore 中，使用文件描述符表 Fdtable 来进行对文件描述符的管理:

```
1   pub struct FdTable {
2       inner: Vec<Option<FileDescriptor>>,
3       recycled: Vec<u8>,
4       soft_limit: usize,
5       hard_limit: usize,
6   }
```

可以理解为文件描述符表内部存在一个向量组，我们通过 open 等操作得到的文件描述符（非负整数），对应的就是该数组的下标。

```
1   pub struct FileDescriptor {
2       cloexec: bool,
3       nonblock: bool,
4       pub file: Arc<dyn File>,
5   }
```

所以在一个进程中，活跃的文件被存放在文件描述符表中，我们通过文件描述符表获得我们需要的文件描述符，从而找到我们需要操作的文件，进行处理。

## 2.4 NPUcore 的网络

NPUcore 为实现网络模块功能，采用组委会的建议，使用 rust 语言的 smoltcp 中的网络协议栈以实现 socket 的相关系统调用。

### 2.4.1 网络接口模块

由于使用 smoltcp 依赖前需要对 device 进行配制，NPUcore 选择构建 config 模块，用于进行网络设备的初始化。

NPUcore 的网络 config 主要用于优化内核与网络设备的交互，通过在不同源代码中对`NetInterfaceInner`方法的不同实现来适配不同协议的 socket 功能，目前的`NetInterfaceInner`只支持本地环回检测。

目前 NPUcore 的网络模块支持 Ipv4 和 Ipv6 两种地址，通过构建网络的 address 模

块实现自定义的 Ipv4，Ipv6 与 smoltcp 协议栈的`IpEndPoint`，`IpListenEndPoint`的相互转化。

### 2.4.2 Socket 模块

NPUcore 的网络通过在 mod 模块中实现 Socket 的特征，并根据`SocketType`的具体类型来分配 Tcp 和 Udp 的不同实例。

代码片段 2.14　net/mod.rs

```
pub trait Socket: File {
fn bind(&self, addr: IpListenEndpoint) -> SyscallRet;
fn listen(&self) -> SyscallRet;
fn connect<'a>(&'a self, addr_buf: &'a [u8]) -> SyscallRet;
fn accept(&self, sockfd: u32, addr: usize, addrlen: usize) ->
    SyscallRet;
fn socket_type(&self) -> SocketType;
fn recv_buf_size(&self) -> usize;
fn send_buf_size(&self) -> usize;
fn set_recv_buf_size(&self, size: usize);
fn set_send_buf_size(&self, size: usize);
fn loacl_endpoint(&self) -> IpListenEndpoint;
fn remote_endpoint(&self) -> Option<IpEndpoint>;
fn shutdown(&self, how: u32) -> GeneralRet<()>;
fn set_nagle_enabled(&self, enabled: bool) -> SyscallRet;
fn set_keep_alive(&self, enabled: bool) -> SyscallRet;
}
```

秉承着“一切皆是文件”的思想，我们让 Scoket 继承了 File 的特征，仿照维护 fd_tableo 的方法，在每个进程中加入 socket_table 以维护该进程对应的 socket，更方便我们按照处理文件的方法对需要 listen，accept，connect 的 socket 进行对应的处理。socket 值提供了 connet，listen，accept 方法的接口，具体的实现由 Tcp 和 Udp 完成。

#### (1) Tcp 模块

NPUcore 的`TcpSocket`结构体保存了创建 socket 时的`SocketHandle`信息，同时在`TcpSocketInner`中保存本地地址，远端地址和上一个状态（保证按序发送和接受数据），以及接受数据流大小和发送数据流大小等。

对于 Tcp 模块中的一些方法，我们重点讨论`accept`方法。Tcp 接受连接时，首先需要获取并暂时保存旧的 socket 信息，之后调用`_accept`方法，通过判断 `TcpScoket`的上一状态来决定是否 accept。如果接受新的连接，则通过远端的地址创建并绑定新的套接字，同时更新地址信息并用新的`TcpSocket`替换旧的 `TcpSocket`，最后将旧的 socket 与新的 fd 绑定并插入到 fd_table。

代码片段 2.15　net/tcp.rs

```
fn accept(&self, sockfd: u32, addr: usize, addrlen: usize) -> crate::utils
    ::error::SyscallRet {
// get old socket
let task = current_task().unwrap();
let mut fd_table = task.files.lock();
let mut socket_table = task.socket_table.lock();
```

```
let old_file = fd_table.get_ref(sockfd as usize).unwrap();
let old_nonblock = old_file.get_nonblock();
let old_cloexec = old_file.get_cloexec();

let peer_addr = self._accept(old_nonblock)?;
log::info!("[Socket::accept] get peer_addr: {:?}", peer_addr);
let local = self.loacl_endpoint();
log::info!("[Socket::accept] new socket try bind to : {:?}", local);
let new_socket = TcpSocket::new();
use core::convert::TryInto;
new_socket.bind(local.try_into().expect("cannot convert to ListenEndpoint")
    )?;
log::info!("[Socket::accept] new socket listen");
new_socket.listen()?;
address::fill_with_endpoint(peer_addr, addr, addrlen)?;
let new_socket = Arc::new(new_socket);
log::debug!("[Socket::accept] take old sock");
// 取出旧的
let old_file = fd_table.take(sockfd as usize).unwrap();
let old_socket: Option<Arc<dyn Socket>> =
socket_table.get_ref(sockfd as usize).cloned();
// 新的替换旧的
log::debug!("[Socket::accept] replace old sock to new");
let _ = fd_table.insert_at(
FileDescriptor::new(old_cloexec, old_nonblock, new_socket.clone()),
sockfd as usize,
);
socket_table
.insert(sockfd as usize, new_socket.clone());
// 旧的插在新的fd上
let fd = fd_table.insert(old_file).unwrap();
socket_table.insert(fd, old_socket.unwrap());
log::info!("[Socket::accept] insert old sock to newfd: {}", fd);
Ok(fd)
}
```

(2) Udp

由于实际的 Udp 并不涉及具体的连接过程，但在实际的测试中 connect 系统调用也会进入到 Udp，所以在 connect 调用进入 Udp 时，对远端的地址和端口信息进行保存，以便在 sendto 时直接使用，这也是与 Tcp 的 connect 的不同之处。

(3) Unix

对于进程间的通信并不需要网络协议栈的支持，所以直接创造一个双向读写的结构体，通过复用 pipe 的 read 和 write 方法以实现进程间的通信。

## 2.5 NPUcore SATA 驱动

### 2.5.1 PCI AHCI 介绍

PCI AHCI 是 PCI Express AHCI 接口的简称，它是一种用于连接计算机主板和存储设备的接口标准。AHCI 代表高级主机控制器接口（Advanced Host Controller Interface），是 SATA（串行 ATA）接口的一种扩展，用于提供更高级的存储功能，如热插拔、原生

命令队列（NCQ）和热备份。

PCI AHCI 接口允许主板通过 PCI Express 总线与 SATA 存储设备进行通信，支持更高的数据传输速率和更复杂的存储管理功能。这种接口在现代计算机系统中非常常见，特别是在需要高性能存储解决方案的场合。

在 NPUcore 中，我们选择接入 Isomorphic Drivers 库，并使用其中已经实现好的 AHCI 接口。

其中，针对部分重要数据结构进行讲解。

```
pub struct AHCI<P: Provider> {
    header: usize,
    size: usize,
    provider: PhantomData<P>,
    ghc: &'static mut AHCIGenericHostControl,
    received_fis: &'static mut AHCIReceivedFIS,
    cmd_list: &'static mut [AHCICommandHeader],
    cmd_table: &'static mut AHCICommandTable,
    data: &'static mut [u8],
    port: &'static mut AHCIPort,
}
```

- AHCI 结构体：它包含了与 AHCI 控制器和 SATA 端口相关的信息和状态。这个结构体包含了一系列成员变量，用于访问 AHCI 寄存器和数据结构。
- AHCIGenericHostControl 结构体：代表了 AHCI 控制器的通用主机控制寄存器。它包含了 AHCI 控制器的能力、全局控制、中断状态等信息。
- AHCIPort 结构体：代表了一个 SATA 端口的寄存器集合，用于与具体的 SATA 设备进行通信。
- AHCIReceivedFIS 结构体：定义了接收到的 FIS（Frame Information Structure）结构，用于从 SATA 设备接收数据。
- AHCICommandHeader 结构体：代表了 AHCI 命令列表中的一个命令头部，用于描述一个 ATA 或 ATAPI 命令，包括命令 FIS、ATAPI 命令缓冲区以及一系列物理区域描述符表（PRDT）。
- SATAFISRegH2D 结构体：代表了一个 SATA 寄存器 FIS（Frame Information Structure）的 Host to Device 格式，用于向 SATA 设备发送命令。
- AHCICommandTable 结构体：定义了 AHCI 命令表的结构，用于描述命令执行的细节，包括命令 FIS、ATAPI 命令以及一系列物理区域描述符表（PRDT）。

通过在 NPUcore 中适配后，我们确定针对 SATA 的 MMIO 起始地址为 0x400E_0000，区域大小为 0x1_0000。具体配置如下：

```
    use crate::config::HIGH_BASE_EIGHT;

    pub const MMIO: &[(usize, usize)] = &[
    (0x400E_0000, 0x1_0000)
    ];

```

```
    pub const BLOCK_SZ: usize = 2048;
    // warning: 不能移除“ + HIGH_BASE_EIGHT”，会导致开发板上地址错误
    pub const UART_BASE: usize = 0x1FE2_0000 + HIGH_BASE_EIGHT;
    pub const ACPI_BASE: usize = 0x1FE2_7000 + HIGH_BASE_EIGHT;
```

### 2.5.2 Sata 驱动程序

#### (1) SataBlock 结构体

SataBlock 结构体用于表示一个 SATA 块设备，封装了 AHCI（高级主机控制器接口）操作。

- `Mutex<AHCI<Provider>>`：在互斥锁中封装的 AHCI 控制器，用于安全的并发访问。
- `new()`:
  - 作用：创建一个新的 SataBlock 实例的构造函数。
  - 实现：利用 `pci_init()` 函数初始化 AHCI 控制器，并将其封装在互斥锁中。
  - 代码：

```
impl SataBlock {
    pub fn new() -> Self {
        Self(Mutex::new(pci_init().expect("AHCI new
            failed")))
    }
}
```

- `read_block()`:
  - 作用：读取指定块的数据到缓冲区中。
  - 实现：根据内核的块大小和 SATA 驱动的块大小进行转换，循环读取数据块。
  - 代码：

```
impl BlockDevice for SataBlock {
    fn read_block(&self, mut block_id: usize, buf: &mut
        [u8]) {
        // 内核BLOCK_SZ为2048，SATA驱动中BLOCK_SIZE为
            512，四倍转化关系
        block_id = block_id * (BLOCK_SZ / BLOCK_SIZE);
        for buf in buf.chunks_mut(BLOCK_SIZE) {
            self.0
            .lock()
            .read_block(block_id, buf);
            block_id += 1;
        }
    }
}
```

- `write_block()`:
  - 作用：将缓冲区中的数据写入指定块。
  - 实现：根据内核的块大小和 SATA 驱动的块大小进行转换，循环写入数据块。
  - 代码：

```
impl BlockDevice for SataBlock {
    fn write_block(&self, mut block_id: usize, buf: &[
        u8]) {
        block_id = block_id * (BLOCK_SZ / BLOCK_SIZE);
        for buf in buf.chunks(BLOCK_SIZE) {
            self.0
            .lock()
            .write_block(block_id, buf);
            block_id += 1;
        }
    }
}
```

(2) Provider 结构体

Provider 结构体实现了 provider::Provider trait，提供了 DMA（直接内存访问）分配和释放的功能。

- PAGE_SIZE：指定页面大小。
- alloc_dma(size: usize) -> (usize, usize):
  - 作用：分配指定大小的 DMA 内存。
  - 实现：根据请求的大小分配页面，并确保页面连续。
  - 代码：

```
impl provider::Provider for Provider {
    const PAGE_SIZE: usize = PAGE_SIZE;
    fn alloc_dma(size: usize) -> (usize, usize) {
        let pages = size / PAGE_SIZE;
        let mut base = 0;
        for i in 0..pages {
            let frame = frame_alloc().unwrap();
            let frame_pa: PhysAddr = frame.ppn.into();
            let frame_pa = frame_pa.into();
            core::mem::forget(frame);
            if i == 0 {
                base = frame_pa;
            }
            assert_eq!(frame_pa, base + i * PAGE_SIZE);
        }
        let base_page = base / PAGE_SIZE;
        info!("virtio_dma_alloc: {:#x} {}", base_page,
            pages);
        (base, base)
    }
```

- dealloc_dma(va: usize, size: usize):
  - 作用：释放指定虚拟地址和大小的 DMA 内存。
  - 实现：根据请求的大小逐页释放内存。
  - 代码：

```
    fn dealloc_dma(va: usize, size: usize) {
        info!("dealloc_dma: {:x} {:x}", va, size);
        let pages = size / PAGE_SIZE;
```

```
                    let mut pa = va;
                    for _ in 0..pages {
                        frame_dealloc(PhysAddr::from(pa).into());
                        pa += PAGE_SIZE;
                    }
                }
            }
```

(3) pci_init() 函数

pci_init() 函数初始化 PCI（外设组件互连）总线并搜索 AHCI 设备。

如果找到 AHCI 控制器则返回可选的 AHCI 控制器，否则返回 None。

初始化步骤如下：

1. 扫描 PCI 总线以寻找设备。
2. 遍历每个设备并打印信息。
3. 检查设备是否兼容 AHCI 的 SATA 控制器。
4. 如果找到则启用设备并初始化 AHCI。

```
pub fn pci_init() -> Option<AHCI<Provider>> {
    for dev in unsafe {
        scan_bus(
        &UnusedPort,
        CSpaceAccessMethod::MemoryMapped,
        PCI_CONFIG_ADDRESS,
        )
    } {
        info!(
        "pci: {:02x}:{:02x}.{} {:#x} {:#x} ({} {}) irq: {}:{:?}",
        dev.loc.bus,
        dev.loc.device,
        dev.loc.function,
        dev.id.vendor_id,
        dev.id.device_id,
        dev.id.class,
        dev.id.subclass,
        dev.pic_interrupt_line,
        dev.interrupt_pin
        );
        dev.bars.iter().enumerate().for_each(|(index, bar)| {
            if let Some(BAR::Memory(pa, len, _, t)) = bar {
                info!("\tbar#{} (MMIO) {:#x} [{:#x}] [{:?}]", index, pa
                    , len, t);
            } else if let Some(BAR::IO(pa, len)) = bar {
                info!("\tbar#{} (IO) {:#x} [{:#x}]", index, pa, len);
            }
        });
        if dev.id.class == 0x01 && dev.id.subclass == 0x06 {
            // Mass storage class, SATA subclass
            if let Some(BAR::Memory(pa, len, _, _)) = dev.bars[0] {
                if pa == 0 {
                    continue;
                }
                info!("Found AHCI device");
                // 检查status的第五位是否为1，如果是，则说明该设备存在
```

```
                    能力链表
                if dev.status | Status::CAPABILITIES_LIST == Status::
                    empty() {
                    info!("\tNo capabilities list");
                    return None;
                }
                unsafe { enable(dev.loc) };
                if let Some(x) = AHCI::new(pa as usize, len as usize) {
                    return Some(x);
                }
            }
        }
    }
    None
}
```

### (4) enable() 函数

enable() 函数通过配置其命令寄存器来启用 PCI 设备。

- loc：PCI 设备的位置。

具体步骤如下：

1. 读取原始命令寄存器值。
2. 设置适当的位以启用总线主控和特殊循环。
3. 将修改后的值写回命令寄存器。

```
unsafe fn enable(loc: Location) {
    let ops = &UnusedPort;
    let am = CSpaceAccessMethod::MemoryMapped;

    let orig = am.read16(ops, loc, PCI_COMMAND);
    // bit0     |bit1       |bit2          |bit3           |bit10
    // IO Space |MEM Space  |Bus Mastering |Special Cycles |PCI
        Interrupt Disable
    am.write32(ops, loc, PCI_COMMAND, (orig | 0x40f) as u32);
    // Use PCI legacy interrupt instead
    // IO Space | MEM Space | Bus Mastering | Special Cycles
    am.write32(ops, loc, PCI_COMMAND, (orig | 0xf) as u32);
}
```

### (5) UnusedPort 结构体

UnusedPort 结构体是一个占位符结构体，实现了 PortOps trait 用于 PCI 端口操作。实现了从 PCI 端口读取和写入的操作。

```
struct UnusedPort;
impl PortOps for UnusedPort {
    unsafe fn read8(&self, _port: u16) -> u8 {0}
    unsafe fn read16(&self, _port: u16) -> u16 {0}
    unsafe fn read32(&self, _port: u16) -> u32 {0}
    unsafe fn write8(&self, _port: u16, _val: u8) {}
    unsafe fn write16(&self, _port: u16, _val: u16) {}
    unsafe fn write32(&self, _port: u16, _val: u32) {}
}
```

(6) PortOps trait

PortOps trait 定义了用于从和向 PCI 端口读取和写入的操作。

- read8, read16, read32：以不同大小从 PCI 端口读取。
- write8, write16, write32：以不同大小向 PCI 端口写入。

# 第 3 章　总结与展望

## 3.1 工作总结

1. 移植到龙芯 2K1000 的 Qemu 模拟器及开发板上；
2. 满分通过初赛测例，实现相关系统调用；
3. 实现龙芯 2K1000 开发板上的 SATA 驱动；
4. 增加网络模块，基本实现 socket 的相关系统调用，支持网络测例；

## 3.2 未来展望

1. 继续完善网络模块相关系统调用，实现网络功能，通过网络相关测例；
2. 实现 Ext4 文件系统，配合 SATA 驱动读取龙芯 2K1000 开发板上的 32GB 固态硬盘；
3. 进一步支持决赛相关测例