NPUcore-IMPACT!!!
设计文档

西北工業大學
NORTHWESTERN POLYTECHNICAL UNIVERSITY

目 录

摘　要

NPUcore-IMPACT 是一个使用 Rust 编写的基于 LoongArch 架构的类 Unix 操作系统。本工作基于原先的 NPUcore(RISCV) 开发迭代并扩展，实现 POSIX 标准系统调用 90 个，支持信号机制及线程。目前 NPUcore 的 OS 家族支持国际开源 RISC-V 指令集、硬件开发板（K210、U740、星光二代）及 Qemu 模拟器，与自主龙芯 LoongArch 指令集、硬件开发板（2K500 / 1000 / 2000、3A5000）及龙芯虚拟机。由于本次龙芯赛道仅需支持 2K1000 开发板，因此我们后文的解释都会基于此开发板进行讲解。同时，我们也会在后文详细介绍我们工作的增量与 OS 特色。目前初赛的所有测试用例已经满分通过，下图是我们队伍（NPUcore-IMPACT!!!）的初赛测试通过情况：

#	用户名	队伍	提交次数(ASC)	最后提交时间(ASC)	rank
1	T202411664992499	重启之我是loader/ 西安邮电大学	29	2024-05-27 20:05:26	102.0000
2	T202410699992496	NPUcore-IMPACT!!!/ 西北工业大学	45	2024-05-03 23:34:44	102.0000
3	T202410699992491	NPUcore-重生之我是菜狗/ 西北工业大学	56	2024-03-07 15:27:46	102.0000
4	T202410460992502	NPUcore-重生之我是秦始皇/ 河南理工大学	43	2024-05-23 18:54:39	89.0000
5	T202410486992576	俺争取不掉队/ 武汉大学	26	2024-05-28 01:38:05	56.0000
6	T202410614992892	HelloWorld/ 电子科技大学	1	2024-04-23 12:28:03	53.0000
7	T202410213992605	Refill/ 哈尔滨工业大学	8	2024-04-02 19:13:24	53.0000
8	T202412802992528	菜鸡不会riscv/ 吉利学院	2	2024-05-04 01:23:42	0.0000

我们的 NPUcore 大致由（如图 0-1）四个模块构成：Syscall（系统调用）、Memory（内存管理）、Process（进程控制）、File System（文件系统）。

关键词： NPUcore；OSKernel；LoongArch

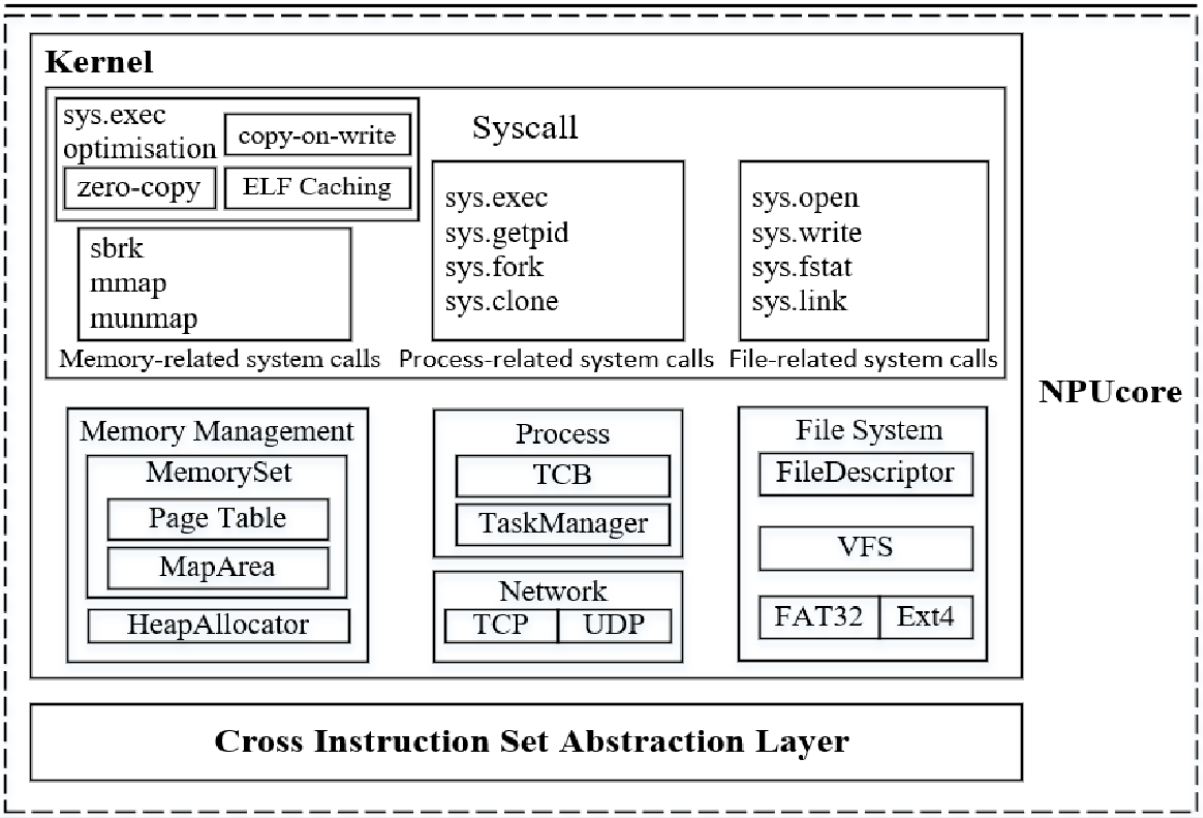


图 0-1　NPUcore 整体内核框架

第 1 章 NPUcore 简介（Introduction）

“NPUcore”是西北工业大学的操作系统内核构建实践型教学操作系统，曾获得 2022 年 OSKernel 大赛内核实现赛道一等奖。NPUcore 致力于使用 Rust 新型编程语言，帮助老师和学生自行研制一个操作系统微型内核，提升操作系统原理的实践体验并探索新型操作系统的设计与实现。原始的 2022 版 NPUcore 具有内存管理、进程管理、文件系统核心系统调用功能，支持 RISCV32/64 指令集，可在对应的 QEMU 模拟器和 SiFive-U740、K210 等嵌入式开发板上运行。该版本基于 rCore-Tutorial 迭代开发，重构 90% 模块以支持 Linux 接口，共实现系统调用 81 个，且满分通过 libc-test，是一个不错的 baseline。虽然该版本有着不错的性能，但却无法支持全部测例，以及国内自主研发的 LoongArch 龙芯架构。不仅如此，该版本不支持网络协议，EXT4 文件系统，以及其它多种多样的外设，因此我们认为，这个版本仍然有很大的优化空间。接下来，我们将分三部分介绍初版 NPUcore，分别是：Rust 特性、目标平台、系统调用。在这些章节中，我们也会提到我们的修改内容。最后我们会给出我们的文件目录树，搭建出新的整体 NPUcore-IMPACT 架构。

1.1 Rust 特性

Rust 是一个“安全、并发、实用”，支持函数式、并发式、过程式以及面向对象的程序设计风格的新型语言。Rust 在完全公开的情况下开发，并且相当欢迎社区的反馈。近些年，Rust 语言在工业应用上的势头越来越猛。基于 Rust 语言的种种特性，我们认为它更适合一些底层应用的开发，尤其是 OSKernel。

1. 我们为什么选择 Rust 作为 OS 编程语言？ Rust 是一门内存安全的语言。对于 C/C++ 这样的手动管理内存的编程语言，我们在分配堆变量的时候需要调用 malloc/new 函数，而当该变量使用完毕之后要手动调用 free/delete 回收内存。这就要求程序员需要关注所有堆变量的生命周期并及时将其释放，否则就会造成内存泄漏的问题，而过早的释放堆变量又可能造成“use-after- free”的问题。而 Rust 独特的所有权机制和借用检查，让编译器掌管变量的生命周期，使得变量的回收变得可控，同时也杜绝了”use-after-free”的问题，又不至于带来垃圾回收的开销。

Rust 还能够推断出类型的大小，然后分配正确的内存大小并将其设置为您要求的值。但这意味着无法分配未初始化的内存：Rust 没有 null 的概念。此外，所有这些检查都是在编译时完成的，因此没有运行时开销，这也是为什么 Rust 被成为是安全的“C”。如果你编写了正确的 C++ 代码，你将编写出与 C++ 代码基本上相同的 Rust 代码。而且由于编译器的帮忙，编写错误的代码版本是不可能的。所以，我们选择 Rust 语言的原因，不仅是因为他安全，还因为其享有和 C 一样的速度，和更丰富的库。

2.unsafe 关键字

几乎每个语言都有 unsafe 关键字，但 Rust 语言使用 unsafe 的原因可能与其它编程语言还有所不同。接下来我们展示一下 unsafe 的特性：

代码片段 1.1　unsafe 展示（r1 是一个裸指针）

```
fn main() {
    let mut num = 5;

    let r1 = &num as *const i32;

    unsafe {
        println!("r1 is: {}", *r1);
    }
}
```

在代码块??中，r1 是一个裸指针 (raw pointer)，由于它具有破坏 Rust 内存安全的潜力，因此只能在 unsafe 代码块中使用，如果你去掉 unsafe{}，编译器会立刻报错。在我们的 NPUcore 中，对于一个 OS 来说，安全是最大的保障，因此 unsafe 在初期 NPUcore 建设中给予了很大帮助。因为，即使做到小心谨慎，依然会有出错的可能性，但是 unsafe 语句块决定了：就算内存访问出错了，你也能立刻意识到，错误是在 unsafe 代码块中，而不花大量时间像无头苍蝇一样去寻找问题所在。unsafe 不安全，但是该用的时候就要用，在一些时候，它能帮助我们大幅降低代码实现的成本。虽然在网上充斥着“千万不要使用 unsafe，因为它不安全”的言论。事实上，我们认为 unsafe 是一个有效且必要的手段，因此我们选择遵循如下规则去使用：

1. 没必要用时，就不用；
2. 当有必要用时，就大胆用，但是要控制好边界；
3. 尽量保证 unsafe 的边界范围最小。

1.2 目标平台（Target）

1、RISC-V

RISC-V 是一个基于精简指令集（RISC）原则的开源指令集架构，其指令集设计简洁、高效，并且具有可扩展性。与大多数指令集相比，RISC-V 指令集可以自由地用于任何目的，允许任何人设计、制造和销售 RISC-V 芯片和软件而不必支付给任何公司专利费。虽然这不是第一个开源指令集，但它具有重要意义，因为其设计使其适用于现代计算设备（如仓库规模云计算机、高端移动电话和微小嵌入式系统）。设计者考虑到了这些用途中的性能与功率效率。该指令集还具有众多支持的软件，这解决了新指令集通常的弱点。RISC-V 指令集的设计考虑了小型、快速、低功耗的现实情况来实做，但并没有对特定的微架构做过度的设计。

先前的比赛只有 RISC-V 一个主赛道，其中包括了 k210，U740，和对应的 QEMU 模拟器三个平台。我们的 NPUcore 不仅支持了三个平台，同时也是 k210 榜单上唯一功能完整 (能够通过所有性能测试项) 的队伍，且 U740 平台上 80% 性能测试项超越基准

而本次比赛还包括了 LoongArch 赛道，这也是我们国产自主的指令集架构，因此更具有里程碑式意义，因此我们果断选择参加了本赛道。

2、LoongArch

2020 年，龙芯中科基于二十年的 CPU 研制和生态建设积累推出了龙架构（LoongArch），包括基础架构部分和向量指令、虚拟化、二进制翻译等扩展部分，近 2000 条指令。

龙架构具有较好的自主性、先进性与兼容性。它从整个架构的顶层规划，到各部分的功能定义，再到细节上每条指令的编码、名称、含义，在架构上进行自主重新设计，具有充分的自主性。龙架构摒弃了传统指令系统中部分不适应当前软硬件设计技术发展趋势的陈旧内容，吸纳了近年来指令系统设计领域诸多先进的技术发展成果。同原有兼容指令系统相比，不仅在硬件方面更易于高性能低功耗设计，而且在软件方面更易于编译优化和操作系统、虚拟机的开发。龙架构在设计时充分考虑兼容生态需求，融合了各国际主流指令系统的主要功能特性，同时依托龙芯团队在二进制翻译方面十余年的技术积累创新，能够实现多种国际主流指令系统的高效二进制翻译。龙芯中科从 2020 年起新研的 CPU 均支持 LoongArch。

今年的龙芯赛道，将基于 2K1000 开发板进行 OSkernel 开发。龙芯 2K1000 的结构如图**??**所示。一级交叉开关连接两个处理器核、两个二级 Cache 以及 IO 子网络（Cache 访问路径）。二级交叉开关连接两个二级 Cache、内存控制器、启动模块（SPI 或者 LIO）以及 IO 子网络（Uncache 访问路径）。IO 子网络连接一级交叉开关，以减少处理器访问延迟。IO 子网络中包括需要 DMA 的模块（PCIE、GMAC、SATA、USB、HDA/I2S、NAND、SDIO、DC、GPU、VPU、CAMERA 和加解密模块）和不需要 DMA 的模块，需要 DMA 的模块可以通过 Cache 或者 Uncache 方式访问内存。更详细的细节，可以参考龙芯官方提供的 2K1000 处理器用户手册。

同时，大赛官方也提供了对应的 Qemu 版本https://gitlab.educg.net/wangmingjian/os-contest-2024-image（似乎有未解决 bug）。我们根据先前在 RISC-V 的经验，写了如下的 runqemu 脚本，则可以启动对应我们 OS 版本的 2K1000-QEMU。

代码片段 1.2 uitl/qemu/runqemu

```
1  #!/bin/bash
2  DISK=/tmp/disk
3  SCRIPTPATH="$( cd -- "$(dirname "$0")" >/dev/null 2>&1 ; pwd -P )"
4  QEMU="$SCRIPTPATH"/bin/qemu-system-loongarch64
5  [ -e $DISK ] || { truncate -s 32M $DISK;echo -e 'n\n\n\n\n\n\nw\nq\n'|
       fdisk /tmp/disk; }
6  SUDO=$(if [ $(whoami) = "root" ];then echo -n "";else echo -n "sudo";fi
       )
7  TFTP_DIR="$SCRIPTPATH"/../../easy-fs-fuse
8
9  ls2k()
10 {
11 BIOS="$SCRIPTPATH"/2k1000/u-boot-with-spl.bin
```

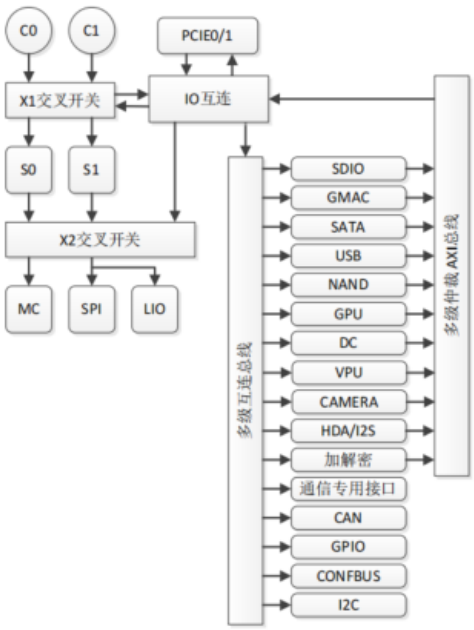


图 1-1　LA2K1000 结构图

```
    DEBUG_UNALIGN=1 DEBUG_GMAC_PHYAD=0 DEBUG_MYNAND=cs=0,id=0x2cda
       DEBUG_MYSPIFLASH=gd25q128 $QEMU \
        -M ls2k \
        -serial stdio \
        -drive if=pflash,file=$BIOS  \
        -m 1024 \
        -device usb-kbd,bus=usb-bus.0 -device usb-tablet,bus=usb-bus.0 \
        -device usb-storage,drive=udisk \
        -vnc :0 -D "$SCRIPTPATH"/qemu.log \
        -drive if=none,id=udisk,file=$DISK \
        -net nic -net user,net=192.168.1.2/24,tftp=$TFTP_DIR \
        -smp threads=1\
        -s -hda "$SCRIPTPATH"/2k1000/2kfs.img \
        -k "$SCRIPTPATH"/share/qemu/keymaps/en-us
    }

    ls2k "$@"
```

为了适配 2K1000 与龙芯架构，我们重构了大部分构建脚本，和底层硬件交互代码（约 4000 行）。这个过程非常曲折，遇到了各种各样繁复的问题，在这里我们已经没有办法一一说明了，这些细节我们都会在后续代码和注释中提到，希望可以对今后的队伍有所帮助。

1.3 系统调用

在初版 NPUcore 中，我们实现了系统调用 81 个，并满分通过了 libc 测试。如今，我们在 NPUcore-IMPACT 中已经增加到 90 个 POSIX 系统调用（如图**??**和**??**），完整版可参考“os/src/arc/la64/syscall_id.rs”。

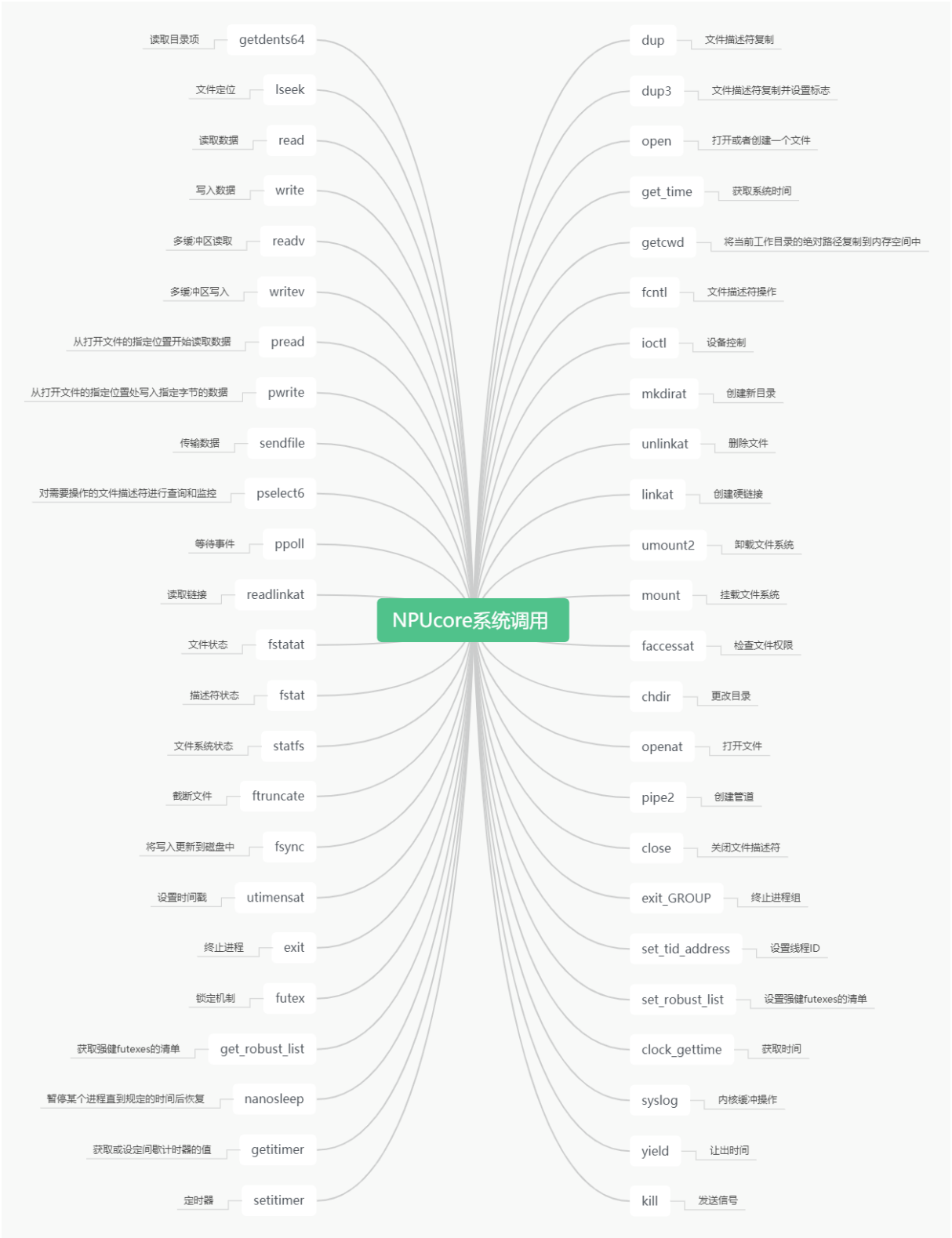


图 1-2　NPUcore-IMPACT 系统调用（其一）

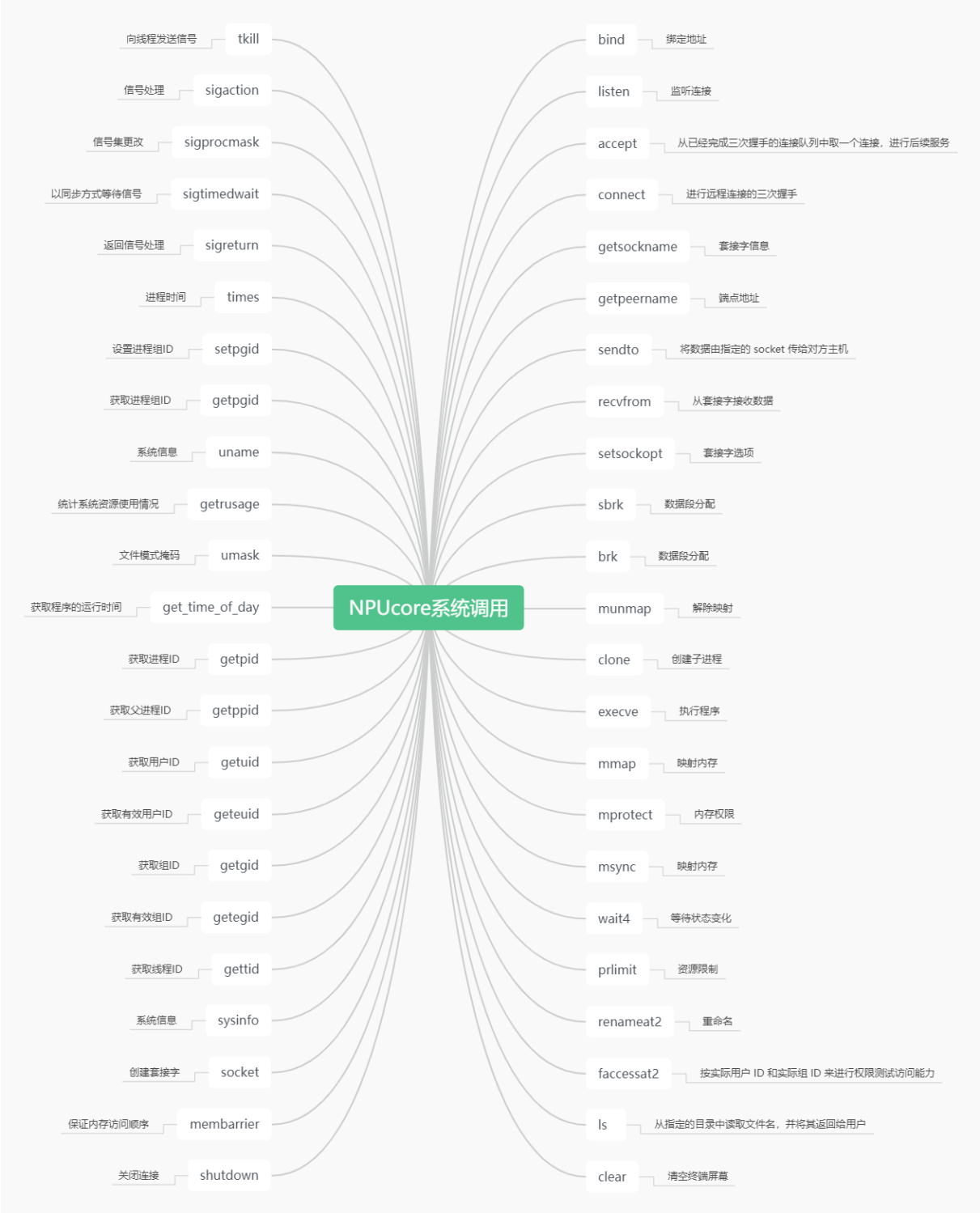


图 1-3　NPUcore-IMPACT 系统调用（其二）

1.4 NPUcore-IMPACT 目录树

代码片段 1.3 目录树

```
./
|-- Makefile            // 根目录makefile，用于链接到os目录
|-- README.md           // Readme
|-- dependency          // 使用的外部包
|   |-- ext4            // EXT4文件系统（初赛不提交该版本）
|   |-- virtio-drivers  // 虚拟设备
|-- os                  // OS内核
|   |-- Cargo.lock
|   |-- Cargo.toml
|   |-- Makefile        // 核心makefile
|   |-- buildfs.sh      // 建立fat32文件系统
|   |-- cpio.log
|   |-- la_fat          // fat32文件系统脚本
|   |-- src             // 核心代码
|   |   |-- arch        // LA2K1000底层交互
|   |   |   |-- la64
|   |   |   |   |-- board
|   |   |   |   |   |-- 2k1000.rs
|   |   |   |   |-- config.rs
|   |   |   |   |-- entry.asm
|   |   |   |   |-- kern_stack.rs
|   |   |   |   |-- la_libc_import.rs
|   |   |   |   |-- laflex.rs
|   |   |   |   |-- load_img.S
|   |   |   |   |-- mod.rs
|   |   |   |   |-- register
|   |   |   |   |   |-- base
|   |   |   |   |   |-- macros.rs
|   |   |   |   |   |-- mmu
|   |   |   |   |   |-- mod.rs
|   |   |   |   |   |-- ras
|   |   |   |   |   |-- timer
|   |   |   |   |-- sbi.rs
|   |   |   |   |-- switch.S
|   |   |   |   |-- switch.rs
|   |   |   |   |-- syscall_id.rs
|   |   |   |   |-- time.rs
|   |   |   |   |-- tlb.rs
|   |   |   |   |-- trap
|   |   |   |   |   |-- context.rs
|   |   |   |   |   |-- mem_access.rs
|   |   |   |   |   |-- mod.rs
|   |   |   |   |   |-- trap.S
|   |   |   |-- mod.rs
|   |   |-- console.rs
|   |   |-- drivers     // 设备块
|   |   |   |-- block
|   |   |   |   |-- block_dev.rs
|   |   |   |   |-- mem_blk.rs
|   |   |   |   |-- mod.rs
|   |   |   |   |-- virtio_blk.rs
|   |   |   |-- mod.rs
|   |   |   |-- serial
|   |   |   |   |-- mod.rs
```

```
|   |   |       |-- ns16550a.rs
|   |   |-- fs          //文件系统
|   |   |   |-- cache.rs
|   |   |   |-- dev
|   |   |   |   |-- hwclock.rs
|   |   |   |   |-- mod.rs
|   |   |   |   |-- null.rs
|   |   |   |   |-- pipe.rs
|   |   |   |   |-- socket.rs
|   |   |   |   |-- tty.rs
|   |   |   |   |-- zero.rs
|   |   |   |-- directory_tree.rs
|   |   |   |-- fat32
|   |   |   |   |-- bitmap.rs
|   |   |   |   |-- dir_iter.rs
|   |   |   |   |-- efs.rs
|   |   |   |   |-- inode.rs
|   |   |   |   |-- layout.rs
|   |   |   |   |-- mod.rs
|   |   |   |   |-- vfs.rs
|   |   |   |-- file_trait.rs
|   |   |   |-- filesystem.rs
|   |   |   |-- layout.rs
|   |   |   |-- mod.rs
|   |   |   |-- poll.rs
|   |   |   |-- swap.rs
|   |   |-- lang_items.rs
|   |   |-- linker.in.ld
|   |   |-- main.rs
|   |   |-- mm          // 访存
|   |   |   |-- address.rs
|   |   |   |-- frame_allocator.rs
|   |   |   |-- heap_allocator.rs
|   |   |   |-- map_area.rs
|   |   |   |-- memory_set.rs
|   |   |   |-- mod.rs
|   |   |   |-- page_table.rs
|   |   |   |-- zram.rs
|   |   |-- syscall     // 系统调用
|   |   |   |-- errno.rs
|   |   |   |-- fs.rs
|   |   |   |-- mod.rs
|   |   |   |-- process.rs
|   |   |   |-- socket.rs
|   |   |-- task        // 线程
|   |   |   |-- context.rs
|   |   |   |-- elf.rs
|   |   |   |-- manager.rs
|   |   |   |-- mod.rs
|   |   |   |-- pid.rs
|   |   |   |-- processor.rs
|   |   |   |-- signal.rs
|   |   |   |-- task.rs
|   |   |   |-- threads.rs
|   |   |-- timer.rs
|   |-- vendor          // 大赛要求的离线包
|-- user                // user测例与用户系统调用
|   |-- Cargo.lock
```

```
    |   |-- Cargo.toml
    |   |-- Makefile
    |   |-- busybox_lua_testsuites
    |   |-- loongarch64
    |   |-- src
    |   |   |-- bin
    |   |   |-- console.rs
    |   |   |-- la_libc_import.rs
    |   |   |-- lang_items.rs
    |   |   |-- lib.rs
    |   |   |-- linker-2k500.ld
    |   |   |-- linker.ld
    |   |   |-- syscall.S
    |   |   |-- syscall.rs
    |   |   |-- usr_call.rs
    |   |-- user_C_program
    |-- util                  // 大赛官方qemu与镜像生成
        |-- mkimage
        |-- qemu
```

第 2 章　相关工作（Related Work）

为了取长补短，我们这里将与 NPUcore-IMPACT 优化相关的工作总结到本章节中。

2.1 NPUcore 系统调用性能分析

我们对比了往年的 OSKernel 大赛优胜，与我们 NPUcore-family 的多种性能，以便从中获取到关键优化信息。首先我们给出 NPUcore，基准程序，与两个其它内核 Titanix 和 PLNTRY 在 libc-bench 和 Unixbench 测例上的耗时与得分（如表**??**和**??**）。

我们可以看到，NPUcore 的耗时均比基准程序短，Unixbench 得分均比基准程序高，这说明 NPUcore 相较于基准程序，我们在时间维度上性能提升许多，但是我们仍有一些测例无法通过。例如对于缓存相关测试，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。即使发生内存不足，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。而因为对于比赛而言，和基准程序比较不能说明太多问题，应与其它参赛队伍比较，这点我们会在下面详细分析。

2.2 其他内核系统调用性能分析

2.2.1 Titanix

首先，从https://gitlab.eduxiji.net/202318123101314/oskernel2023-Titanix地址克隆仓库，切换到 master 分支。根据其 README 说明，进入 kernel 目录并运行 sudo make fs-img, 镜像构建完成后运行 make run 启动内核，在内核中运行 runtestcase 开始运行测例。

通过比较 NPUcore 和 Titanix 的 libc-bench 和 Unix-bench 可以发现, 在 libc-bench 中, NPUcore 的和 Titanix 的所耗时间比较相似，不过 NPUcore 对于进程的创建要表现得更好一些。在 Unix-bench 中，Titanix 的部分性能优于 NPUcore，例如 SYSCALL (lps), 而 NPUcore 在其他方面的得分均略高于 Titanix, 特别是 EXEC test(lps)。

对于 Titanix 性能表现的分析：

- 内存管理：实现了页缓存和块缓存以减少 IO 次数，实现懒分配和写时复制以优化性能。Titanix 的懒分配包括三个方面：（1）用户栈的懒分配，进程构建出来时只分配虚拟地址栈空间，当用户访问栈空间时再通过缺页中断分配物理页（2）用户堆的懒分配，与用户栈的懒分配相似，当用户真正读写该堆空间时再通过缺页中断进行物理页分配。（3）mmap 内存段的懒读取，当用户进行 mmap 系统调用时，记录下对应的文件指针以及映射的偏移量范围但不进行实际读取，当用户真正读写到该内存段时再通过缺页中断读取相应文件的相应位置的内容。Titanix 的写时

测例	NPUcore	基准程序	Titanix	PLNTRY
b_malloc_sparse (0)	-	0.325008007	0.307474000	0.749882000
b_malloc_bubble (0)	-	0.299156383	0.302118000	0.795564000
b_malloc_tiny1 (0)	-	0.011439820	0.010716000	0.020421000
b_malloc_tiny2 (0)	0.005085920	0.008827947	0.008677000	0.013895000
b_malloc_big1 (0)	-	0.089668000	0.089668000	0.205780000
b_malloc_big2 (0)	0.028311200	0.092570898	0.089264000	0.159222000
b_malloc_thread_stress (0)	0.066339520	0.078210295	0.088639000	0.065488000
b_malloc_thread_local (0)	-	0.076371644	0.084808000	0.057541000
b_string_strstr (”abcdefghijklmnopqrstuvwxyz”)	0.011712000	0.014879018	0.014637000	0.014606000
b_string_strstr (”azbycxdwevfugthsirjqkplomn”)	0.017838240	0.022160122	0.022694000	0.022971000
b_string_strstr (”aaaaaaaaaaaaaaccccccccccc”)	0.011262640	0.013371076	0.014105000	0.014078000
b_string_strstr (”aaaaaaaaaaaaaaaaaaaaaaaac”)	0.010942000	0.013494579	0.014031000	0.013603000
b_string_strstr (”aaaaaaaaaaaaaaaaaaaaaaaaaaaaaac”)	0.013523120	0.016474362	0.017411000	0.017628000
b_string_memset (0)	0.009740400	0.010840004	0.012793000	0.012005000
b_string_strchr (0)	0.011392640	0.013300974	0.015052000	0.014630000
b_string_strlen (0)	0.009787360	0.011389320	0.013102000	0.012690000
b_pthread_createjoin_serial1 (0)	0.439336960	1.087431144	1.643873000	1.048143000
b_pthread_createjoin_serial2 (0)	0.426977760	0.861784643	1.791139000	1.020008000
b_pthread_create_serial1 (0)	0.476990480	0.785009610	2.702702000	2.689056000
b_pthread_uselesslock (0)	0.060514480	0.067189695	0.079902000	0.078151000
b_utf8_bigbuf (0)	0.032511520	0.035250394	0.035864000	0.037977000
b_utf8_onebyone (0)	0.091743280	0.114485028	0.116207000	0.117029000
b_stdio_putcgetc (0)	0.439399360	0.764247677	0.621226000	0.361184000
b_stdio_putcgetc_unlocked (0)	0.426097680	0.752266360	0.615693000	0.339361000
b_regex_compile (”(a|b|c)*d*b”)	0.057043200	0.071444121	0.073049000	0.075878000
b_regex_search (”(a|b|c)*d*b”)	-	0.081086193	0.083481000	0.086110000
b_regex_search (”a{25}b”)	-	0.254820207	0.253847000	0.256301000

表 2-1　libc-bench 耗时，“-”代表 NPUcore 无法正常执行。

测例	NPUcore	基准程序	Titanix	PLNTRY
Unixbench DHRY2 test(lps)	61405230	49005063	49932495	48523418
Unixbench WHETSTONE test(MFLOPS)	1269.670	1026.820	997.612	1014.102
Unixbench SYSCALL test(lps)	222615	2416696	1216209	547678
Unixbench CONTEXT test(lps)	168113	80174	54440	42416
Unixbench PIPE test(lps)	413979	330556	162389	802449
Unixbench SPAWN test(lps)	64394	16686	14937	13586
Unixbench EXECL test(lps)	73597	6645	1262	-
Unixbench ARITHOH test(lps)	7032261867	5718673825	5607494736	5541357804
Unixbench SHORT test(lps)	179342297	147678033	141948434	140413305
Unixbench INT test(lps)	179660281	148128134	140942140	140311940
Unixbench LONG test(lps)	179644882	153417287	141540178	140287530
Unixbench FLOAT test(lps)	178725171	148049012	141550560	142030413
Unixbench DOUBLE test(lps)	179034748	148031827	142851862	141484435
Unixbench HANOI test(lps)	677685	544504	545068	536762
Unixbench EXEC test(lps)	39493	945	8561	-

表 2-2　Unixbench 测试分数，“-”代表 PLNTRY 无法正常执行。

复制主要指在进行 fork 系统调用构造出新的进程时，不需要将父进程地址空间的全部内存拷贝一份，而是让子进程与父进程共享物理内存页，这样做的开销就只有修改页表。同时如果某个内存段是懒分配的内存段，便不需要共享物理页，直接新增一个虚拟地址内存段即可。

- 进程管理：Titanix 采用无栈协程的调度方式，所有线程（包括不同进程的线程）共享同一个内核栈，调度起来开销比较小。因为所有协程共用一个栈，所以需要每个协程在堆上维护一个状态机，通过轮询当前的状态进行协程的切换，然后根据状态决定是否需要切换。无栈协程的调度是通过函数返回然后调用另一个函数实现的，而不是像有栈协程那样直接原地更改栈指针。也带来了一定程度的性能优化和安全性保证。
- 文件系统：实现了 Inode 缓存，可以减少 IO 次数：Titanix 通过设置全局对象 INODE_CACHE，来对可能会使用的 inode 进行缓存，Inode 缓存主要用于完成某一个文件的 inode 的文件名哈希值与 inode 自身的映射管理。这样一来，在频繁的访问 Inode 时可以减少一部分因为查找带来的 IO 访问磁盘时延，从而达到优化性能的效果。

  Titanix 通过在 Inode 和实际文件名之间建立哈希映射来实现对文件的快速查找：当传入一个文件名时，调用实现的 hash_name 方法进行哈希值的计算，并从构建的全局哈希表当中获取该 inode 的 Arc 引用，即查找到了对应文件。

2.2.2 PLNTRY

首先，从https://gitlab.eduxiji.net/PLNTRY/OSKernel2023-umi/-/blob/comp3-coverage地址处克隆仓库，切换到 master 分支。将比赛提供的镜像文件复制到 OSKernel2023-umi/third-party/img 文件目录下，命名为 sdcard-comp2.img, 在主文件目录下 make all,make run 即可运行该 kernel。所有的测试结果不直接在终端显示，而在 qemu.log 中显示。

通过比较 NPUcore 和 PLNTRY 的 libc-bench 和 Unix-bench 可以发现，在 libc-bench 中，NPUcore 的和 plntry 的所耗时间比较相似，这与测例的相对简单有比较大的关系。但是在 Unix-bench 中，plntry 的部分性能得分优于 NPUcore。例如 SYSCALL (lps),CONTEXT (lps),PIPE (lps), 当然 NPUcore 也有比 plntry 表现更良好的测试项，比如 DHRY2 test(lps)，SPAWN test(lps) 等。

“syscall”测试是一个基准测试，用于评估系统在执行系统调用（syscalls）时的性能。系统调用是操作系统提供给用户空间程序访问操作系统内核功能的接口。这个测试旨在测量系统在执行各种系统调用时的效率和速度。UnixBench 会执行一系列常见的系统调用，比如文件操作、进程控制、内存管理等，然后测量系统在执行这些调用时所需的时间和性能。“syscall”测试的结果以每秒钟能够执行的系统调用数量（lps-syscalls per second）作为单位，因此其结果值越高表示系统在处理系统调用时的效率越高，执行系统调用的能力也就越强。

UnixBench 中的“CONTEXT”测试是用来评估系统在上下文切换方面的性能。上下文切换是操作系统在多任务环境中切换执行不同进程或线程时所需的过程。“CONTEXT”测试测量系统在进行上下文切换时的效率，它涉及将处理器从一个进程或线程切换到另一个的能力。在多任务系统中，上下文切换是一种常见操作，而系统的性能可能会受到其影响。测试结果以每秒钟能够完成的上下文切换数量（lps-context switches per second）作为单位。因此，较高的数值表示系统在处理上下文切换时更有效率，能够更快地在不同的进程或线程之间进行切换。

”spawn” 测试是一个基准测试，用于评估系统在并发进程创建和销毁方面的性能。该测试模拟了系统同时启动多个进程的情况，然后检查系统在这种高并发情况下的性能表现。”spawn” 测试通常会创建许多子进程，然后立即销毁它们，以测试系统处理这些操作的速度和效率。这个测试可以显示系统在处理并发任务时的能力，因为进程的创建和销毁在某些应用场景下可能是非常常见的操作。UnixBench 中的”spawn” 测试的结果以每秒钟能够创建和销毁的进程数（lps-processes per second）作为单位，因此其结果值越高表示系统在这个方面的性能越好。

性能优秀的原因:

- 内存管理: 在页帧管理实现写时复制策略和通用 IO 缓存，减少 IO 设备访问次数。在地址空间管理实现懒分配策略，提高内存.

  在设计 UMI 的页帧管理模块的部分，借鉴 Fuchsia 设计了一套 RAII 的基于二叉树形的数据结构。每个节点逻辑上是其父节点的一个切片，有标志指示是否拥有写时复制（CoW）特性。页帧通过引用计数和缓存状态的更新在树形结构中复制和流动。例如在提交页帧的时候，依次从自身的哈希表、父节点的哈希表、I/O 后端读写、新清零页的顺序来依次访问并提交页帧。

  通过如上说明可以看到，Phys 结构体可以作为任意读写 + 寻址的后端的页缓存，包括普通文件、块设备等。这样，虽然缺乏了一些特定场景的优化，可以避免重复的页缓存代码。并且由于写时复制和懒分配两个特性，任何涉及到内存分配的场景都会获得对应的性能提升。同时，每个节点可以实现同时读写页帧，在不浪费页帧的情况下提升页缓存的并发性。
- 线程管理与调度: 使用细粒度锁和 Rust 所有权系统管理线程的本地状态和信息支持软抢占和任务窃取的 SMP 多核调度器基于有栈协程模式的特权级切换

  传统的操作系统往往采用有栈协程将任务的调用栈和上下文分开保存，通过汇编代码手动切换函数调用栈来进行任务切换。每个任务的调用栈都会有一定的内存浪费（空闲），并都会有栈溢出风险。

  而无栈协程则将任务的信息同一保存成状态机，统一存放在堆上，由执行器通过更改指针来切换执行的任务。从图中我们可以发现，调用栈仅与每个执行器一一绑定，一定程度上减少了内存浪费、降低栈溢出风险。

- 文件系统: 统一的虚拟文件系统接口支持 debugfs、FAT32、procfs、devfs 等多种存储和内存文件系统。本身并没有很多创新之处。
- 网络协议栈: 基于 smoltcp 的多设备接口网络协议栈 TCP 独立的 ACCEPT 队列

第 3 章　LoongArch2K1000 适配

在 NPUcore 中，我们在 os/arc/arch 里对所有涉及硬件交互的组件均更新为 LoongArch 架构版本。如代码片段**??**所示，我们的硬件设备约束，都放入 arch 中的 la64 文件夹，其中分为 board，register，trap 三个子文件夹。在 board 中，2k1000.rs 用于规定与 2K1000 开发板相关的串口宏（这个串口地址也是我们自己摸出来的，未找到任何官方说明）。在 register 中，我们封装了 base（基本寄存器），mmu（tlb 相关），ras，timer 的所有硬件相关寄存器。在 trap 中，我们主要规定了内核态陷入方面所必须的类型与限制。由于有关内存模块的适配最为艰难，因此，我们后面重点讲述这部分内容，同时也会提到基于 2K1000-QEMU 的适配。

代码片段 3.1　“os/arc/arch”目录树

```
    |-- la64
    |   |-- board
    |   |   |-- 2k1000.rs
    |   |-- config.rs
    |   |-- entry.asm
    |   |-- kern_stack.rs
    |   |-- la_libc_import.rs
    |   |-- laflex.rs
    |   |-- load_img.S
    |   |-- mod.rs
    |   |-- register
    |   |   |-- base
    |   |   |   |-- badi.rs
    |   |   |   |-- badv.rs
    |   |   |   |-- cpuid.rs
    |   |   |   |-- crmd.rs
    |   |   |   |-- ecfg.rs
    |   |   |   |-- eentry.rs
    |   |   |   |-- era.rs
    |   |   |   |-- estat.rs
    |   |   |   |-- euen.rs
    |   |   |   |-- llbctl.rs
    |   |   |   |-- misc.rs
    |   |   |   |-- mod.rs
    |   |   |   |-- prcfg.rs
    |   |   |   |-- prmd.rs
    |   |   |   |-- rvacfg.rs
    |   |   |-- macros.rs
    |   |   |-- mmu
    |   |   |   |-- asid.rs
    |   |   |   |-- dmw.rs
    |   |   |   |-- mod.rs
    |   |   |   |-- pgd.rs
    |   |   |   |-- pwch.rs
    |   |   |   |-- pwcl.rs
    |   |   |   |-- stlbps.rs
    |   |   |   |-- tlbehi.rs
    |   |   |   |-- tlbelo.rs
    |   |   |   |-- tlbidx.rs
```

```
40      |   |   |   |-- tlbrbadv.rs
41      |   |   |   |-- tlbrehi.rs
42      |   |   |   |-- tlbrelo.rs
43      |   |   |   |-- tlbrentry.rs
44      |   |   |   |-- tlbrera.rs
45      |   |   |   |-- tlbrprmd.rs
46      |   |   |   |-- tlbrsave.rs
47      |   |   |-- mod.rs
48      |   |   |-- ras
49      |   |   |   |-- merrctl.rs
50      |   |   |   |-- merrentry.rs
51      |   |   |   |-- merrera.rs
52      |   |   |   |-- merrinfo.rs
53      |   |   |   |-- merrsave.rs
54      |   |   |   |-- mod.rs
55      |   |   |-- timer
56      |   |       |-- cntc.rs
57      |   |       |-- mod.rs
58      |   |       |-- tcfg.rs
59      |   |       |-- ticlr.rs
60      |   |       |-- tid.rs
61      |   |       |-- tval.rs
62      |   |-- sbi.rs
63      |   |-- switch.S
64      |   |-- switch.rs
65      |   |-- syscall_id.rs
66      |   |-- time.rs
67      |   |-- tlb.rs
68      |   |-- trap
69      |       |-- context.rs
70      |       |-- mem_access.rs
71      |       |-- mod.rs
72      |       |-- trap.S
73      |-- mod.rs
```

3.1 LoongArch 内存模块适配

3.1.1 内存地址映射布局

LoongArch 的虚拟映射模式内存机理如下图：

1. 首先，检查是否符合直接映射窗口（通过 DMW0～3 四个 CSR 进行映射），如果其高 4 位相同则认为是符合，则将其他位数截断，作为物理地址访问。
2. 其次，如果不是，则检查是否符号扩展，是则尝试查找 TLB，否则出发异常，在 miss 后，如果是 1 扩展则 PGD 为 PGDH，0 扩展为 PGDL，然后触发 TLB 重填异常，ertn 返回后重新进行 TLB 查找。如果此时没有对应的 TLB，则触发页无效异常。
3. 此外, 对 DMW 的权限是: 0 号和 1 号窗口是 RWX, 2 号和 3 号窗口是 RW(不可执行)。每个窗口可以单独设置自己的缓存一致性类型。

由于 LoongArch 的特殊特性，NPUcore-IMPACT 内存布局采取了和 RISC-V 下不同的方案：

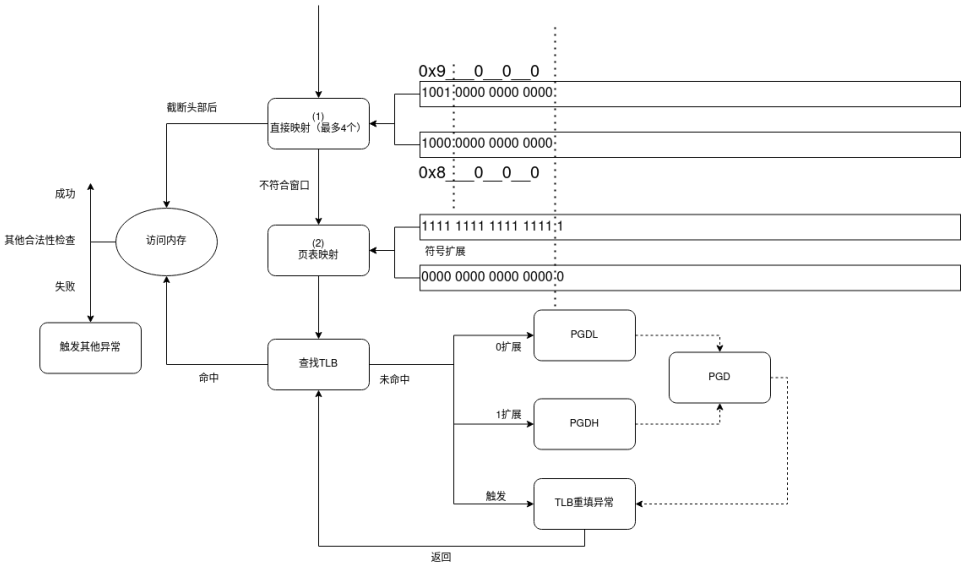


图 3-1　LoongArch 的虚拟映射模式

1. 对用户态，使用 0 扩展地址作为虚拟空间；

2. 对内核态，用 0 扩展地址空间访问物理内存，然后对页表映射使用 1 扩展的地址。

这样做可以用直接继承原版 NPUcore 的部分内存布局，无需为了利用 LA 的直接映射窗口就大量修改代码，在物理地址上按位或大量的 0x9000000090000000，减少代码出错空间。而且，这样可以利用直接映射减小恒等映射的开销，在上下文无需存储页表地址，可以直接切换地址空间。

（1）跳转

从 u-boot 获得控制权时，引导程序提供了几个已经映射好的段: 0x9000 段 (一致可缓存), 和 0x8000 段 (无缓存直接访问内存), 前者使用 DMW1, 后者 DMW0(可能随 u-boot 版本不同而不同)。

由于地址空间中 0 段地址是不被映射的，因此需要某些方法先设置 DMW 映射 0 段，再让 PC 跳转到该地址（NPUcore 内核态的 PC 是在物理地址上的）。为此, 我们的启动代码 la64/entry.asm 需要进行如下设计（如代码片段**??**）。

代码片段 3.2　la64/entry.asm

```
_start:
    pcaddi      $t0,    0x0
    srli.d      $t0,    $t0,    0x30
    slli.d      $t0,    $t0,    0x30    # 位移删去物理地址
    addi.d      $t0,    $t0,    0x11    # 计算当前的窗口CSR值
    csrwr       $t0,    0x181           # 上面代码保证窗口DMW0写入切换后不
        会被覆盖,所以先将DMW1设为当前段
    sub.d       $t0,    $t0,    $t0
    addi.d      $t0,    $t0,    0x11    # 计算0段DMW的值
    csrwr       $t0,    0x180           # 设置0段DMW的值
    pcaddi      $t0,    0x0
    slli.d      $t0,    $t0,    0x10
    srli.d      $t0,    $t0,    0x10
```

```
    jirl        $t0,    $t0,    0x10    # 跳0段的下一条指令
    # The barrier
    sub.d       $t0,    $t0,    $t0
    csrwr       $t0,    0x181           # 写入DMW1，清零DMW1
    sub.d       $t0,    $t0,    $t0     # 加载boot_stack_top
    la.global $sp, boot_stack_top       # 跳转到rust_main
    bl          rust_main
```

注意这里没有使用 $t0=$zero 是因为在 QEMU 虚拟机下, 某些时候似乎 $zero 寄存器会被赋值为 0(GDB 提示), 因此使用手工计算的 $t0=0。这里先设置映射窗口, 而不是计算完 0 段的 DMW 控制寄存器数值后直接覆盖到 DMW0 的原因是：如果当前 PC 的地址处在 DMW0 而非 DMW1 的区段内, 覆盖 DMW0 后 PC 的地址就会成为非法地址。跳转必然发生在映射了新的区段号后，因此要先将当前的区段保存到 DMW1, 然后再进行对 DMW0 的配置, 从而确保至少有一个 DMW 是 PC 当前的地址。

3.1.2 不对齐读写问题

当前开源的 LLVM 的 LoongArch 后端不支持生成严格对齐的代码, 因此也导致依赖 LLVM 后端进行代码生成的 Rustc 无法编译出可以在开发板上正常运行的代码, 这也就要求我们对 NPUcore 的不对齐异常手动处理。

虽然 QEMU 支持不对齐读写, 但是开发板是无法支持不对齐读写的. 如果要在 QEMU 上支持不对齐读写的检测和不对齐读写的报错, 需要开启环境变量 DEBUG_-UNALIGN=1, 否则会忽略所有的不对齐读写异常。具体的启动命令是：DEBUG_-UNALIGN=1 DEBUG_GMAC_PHYAD=0 DEBUG_MYNAND=cs=0,id=0x2cda DEBUG_-MYSPIFLASH=gd25q128 $QEMU。

如果将来 LoongArch 的 LLVM 完成了严格对齐读写的修复, 则应当可以用 target-feature 开启编译器的选项:

代码片段 3.3　la64/entry.asm

```
[target.loongarch64-unknown-linux-gnu]
rustflags = [”-Ctarget-feature=-unaligned-access”,
”-Clink-arg=-Tsrc/linker.ld”, ”-Clink-arg=-nostdlib”, ”-Clink-arg=-static”]
```

这个问题在 C 语言的 GCC 下是不存在的, 因此, 理论上在 Rust 编写的操作系统上, 当前会由于不对齐读写存在性能瓶颈。为了解决这个问题, 我们需要在内核态手动模拟不对齐读写指令的执行。

内核态的不对齐异常只出现在栈上, 因为静态数据和堆上的数据结构是对齐的, 而 NPUcore-LA 的内核堆则是从栈上分配的, 只有在 ELF 加载阶段有部分只读临时的内核页表映射。因此, 只需要保存内容后, 对进行逐字节解读取即可。具体来说, 为了节省内容恢复的空间, 我们直接对 kernel trap 的设计为:

1. 内容保存不需要单独建立一个位置, 直接将寄存器上下文保存在栈上, 因为内核的不对齐读写异常是发生在执行时, 所以直接视为一个单独的函数调用即可。

2. 指令解码使用。

内容保存的代码如下：

代码片段 3.4　la64/trap/trap.S

```
    __kern_trap:
    # Keep the original $sp in SAVE
    csrwr $sp, CSR_SAVE
    csrrd $sp, CSR_SAVE
    # Now move the $sp lower to push the registers
    addi.d $sp, $sp, -256
    # Align the $sp
    srli.d  $sp, $sp, 3
    slli.d  $sp, $sp, 3
    # now sp->*GeneralRegisters in kern space, CSR_SAVE->(the previous $sp)

    SAVE_GP 1 # Save $ra
    SAVE_GP 2 # Save $tp

    # skip r3(sp)
    .set n, 4
    .rept 28
        SAVE_GP %n
        .set n, n+1
    .endr
    .set n, 0
    csrrd $t0, CSR_ERA
    st.d $t0, $sp, 0

    move $a0, $sp
    csrrd $sp, CSR_SAVE
    st.d $sp, $a0, 3*8
    move $sp, $a0

    bl trap_from_kernel

    ld.d  $ra, $sp, 0
    csrwr $ra, CSR_ERA
    LOAD_GP 1
    LOAD_GP 2

    # skip r3(sp)
    .set n, 4
    .rept 28
        LOAD_GP %n
        .set n, n+1
    .endr
    .set n, 0

    csrrd $sp, CSR_SAVE
    ertn
```

读写的关键代码如下:

代码片段 3.5　la64/trap/trap.S

```
if op.is_store() {
                    let mut rd = gr[ins.get_rd_num()];
                    for i in 0..sz {
```

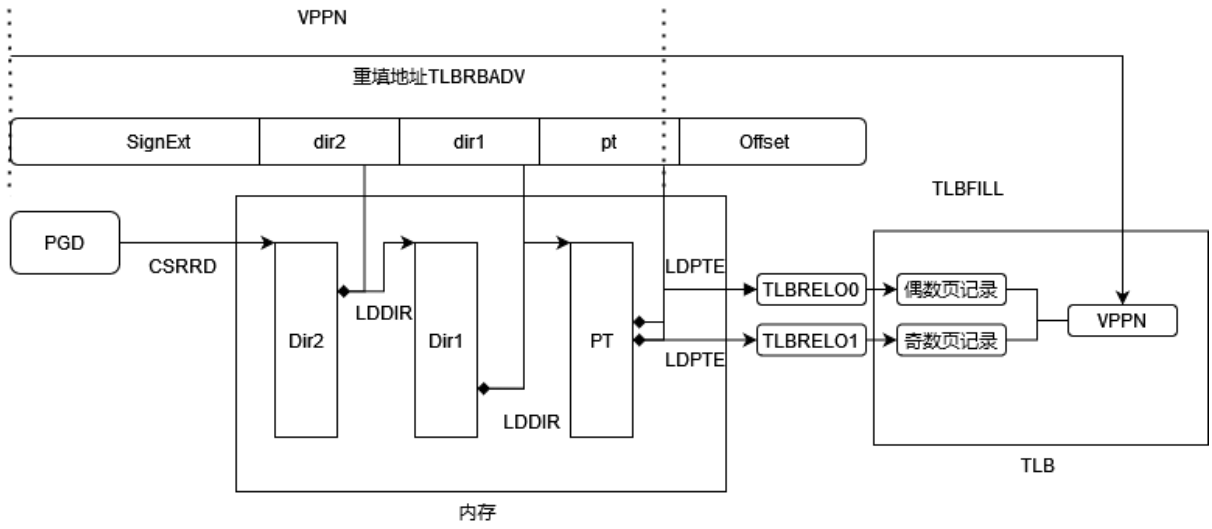


图 3-2　TLB 重填

```
                unsafe { ((addr + i) as *mut u8).write_unaligned(rd
                    as u8) };
                rd >>= 8;
            }
        } else {
            let mut rd = 0;
            for i in (0..sz).rev() {
                rd <<= 8;
                let read_byte =
                    (unsafe { ((addr + i) as *mut u8).
                        read_unaligned() } as usize);
                rd |= read_byte;
                //debug!("{:#x}, {:#x}", rd, read_byte);
            }
            if !op.is_unsigned_ld() {
                match sz {
                    2 => rd = (rd as u16) as i16 as isize as usize,
                    4 => rd = (rd as u32) as i32 as isize as usize,
                    8 => rd = rd,
                    _ => unreachable!(),
                }
            }
            gr[ins.get_rd_num()] = rd;
        }
        gr.pc += 4;
```

注意这里的 rev 是必须的, 因为 LoongArch 是小端机器, 所以高字节的先读才能被或到低位。理论上, 还存在更快的办法, 就是判断是否跨整 2^n 字节, 如果不跨过, 则用汇编选择对应字节的内容, 否则就读入两寄存器 (128bits) 再取各自需要的部分组合成一个寄存器的值. 但由于其实现较为复杂, 所以我们考虑在决赛时采用。

3.1.3 TLB refill 与页表结构

(1) TLB 重填异常

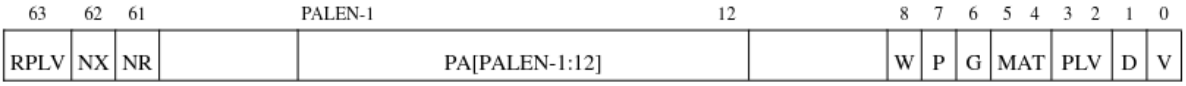
如图**??**，一旦在虚拟地址模式中被判定为页表映射的地址, 发生 TLB miss, 就会触发 TLB 重填异常。LoongArch 目前是手动重填 TLB 的。其重填的一般流程如下：

1. 硬件提供页表地址 PGD。
   (**1**) LoongArch 有两个页表 PGDL 和 PGDH，分别对应符号扩展的全 1 段和 0 段。
   (**2**) 如果触发重填的地址来自 0 段, 则重填的页表 PGD 此时等于 PGDL, 否则 PGD 此时等于 PGDH。这样一来, 页表实际上分为了高地址和低地址页表, 可以用作不同的功能。
2. 硬件触发 TLB 异常，跳转到 TLBRENTRY CSR 所指向的地址。
3. 操作系统会响应异常，通过读取 PGD 状态控制寄存器, 获取最高一级页表目录。
4. 根据虚拟地址计算出对应的物理页框号。然后在主存中查找该物理页框的对应页表项，然后返回该页表项的物理地址. 为了加速页表重填, LoongArch 提供了下列特性:
   (**1**) LoongArch 的 TLB 以双页形式组织，目的是减少重填次数。具体来说，LoongArch 的 TLB 的索引单位是 VPPN（Virtual Page Pair Number），为虚拟页号去掉最第一位。每个 VPPN 对应 VPN 为 VPPN×2+VPN[12] 的一对奇数页和偶数页两页的物理地址。考虑到 TLB 重填的局部性，这样最多可以减少一半的 TLB 重填。
   (**2**) LoongArch 提供 LDDIR 和 LDPTE 两种指令进行页表遍历，通过 TLBRBADV 寄存器提供重填的虚拟地址，并以此为基础获得各级页表的索引号。LDDIR 是用于给定非末级页表起始地址，求取本级页表项（或者说下一级页表起始地址），其格式为 (其中 imm 为页表级数。以 rj 为页表起始地址，rd 为目的寄存器。)：LDDIR rd, rj, imm。
5. 返回页表项后，处理器将该对页表项用 TLBFILL 写入 TLB 中，以便下次使用该虚拟地址时，能够直接从 TLB 中获取物理地址信息，而不需要再次触发 TLB 重填。在 TLB 更新后，处理器会重新执行之前的指令或内存访问操作，这次操作可以直接从 TLB 中获取到物理地址。

(2) LoongArch 的页表结构

官方手册对 LoongArch 的项目并没有清晰的叙述, 其中存在部分不明确的地方。具体来说 (如图**??**), 除了大页之外, LoongArch 并不规定页目录的页表项格式, 所以其实际上是未定义的。而上文提到的 LDDIR 和 LDPTE 并不检查目录项合法性, 且对非法的页表项仍会继续寻址, 因此, 对目录项需要有手工的检测或者非法目录项的处理方法。最简单的方法是直接模拟其他 RISC 架构的页表处理方式, 直接用汇编代码对各层页表项进行处理。首先, 页表是一棵前缀树, 其部分没有被映射的节点是空的, 因此如果直接使用

基本页页表项格式：



大页页表项格式：

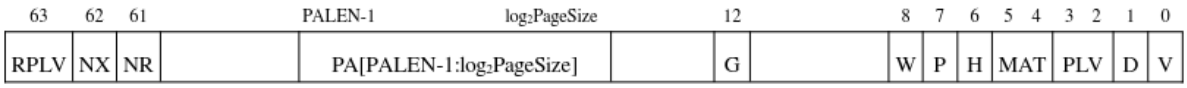


图 3-3　LA 页表格式

上述的 LDPTE 和 LDDIR, 由于其只是单纯的将异常地址的对应段取出作相加, 因此对空的地址会计算出错误的地址, 而不是和其他 RISC 一样触发异常。因此, 非最底层页表的叶结点 (空表项) 必须要手工判断, 并对其错误的表项, 填写无读写权限的 TLB 表项。

代码片段 3.6　TLB refill

```
    csrwr  $t0, 0x8b
    csrrd  $t0, 0x1b
    lddir  $t0, $t0, 3
    andi   $t0, $t0, 1
    beqz   $t0, 1f

    csrrd  $t0, 0x1b
    lddir  $t0, $t0, 3
    addi.d $t0, $t0, -1
    lddir  $t0, $t0, 1
    andi   $t0, $t0, 1
    beqz   $t0, 1f
    csrrd  $t0, 0x1b
    lddir  $t0, $t0, 3
    addi.d $t0, $t0, -1
    lddir  $t0, $t0, 1
    addi.d $t0, $t0, -1

    ldpte  $t0, 0
    ldpte  $t0, 1
    csrrd  $t0, 0x8c
    csrrd  $t0, 0x8d
    csrrd  $t0, 0x0
2:
    tlbfill
    csrrd  $t0, 0x89
    srli.d $t0, $t0, 13
    slli.d $t0, $t0, 13
    csrwr  $t0, 0x11
    tlbsrch
    tlbrd
    csrrd  $t0, 0x12
    csrrd  $t0, 0x13
    csrrd  $t0, 0x8b
    ertn
1:
    csrrd  $t0, 0x8e
    ori    $t0, $t0, 0xC
    csrwr  $t0, 0x8e
```

```
40
41      rotri.d $t0, $t0, 61
42      ori     $t0, $t0, 3
43      rotri.d $t0, $t0, 3
44
45      csrwr   $t0, 0x8c
46      csrrd   $t0, 0x8c
47      csrwr   $t0, 0x8d
48      b       2b
```

代码片段**??**实现了以下功能:

1. 逐层读取页表项, 如果没到最后一层, 检查读取到的页表项的合法性。合法则继续读取下一级页表项；非法则准备填入 0 页表项, 表示该页非法。
2. 填入 0 页表项或者读取最后一层页表项结束后, 将页表大小填写为 4KiB, 然后向 TLB 填入 0 地址, 表示该地址不合法。注意, 这段代码有 2 个需要注意的细节:
   (**1**) 由于该段代码使用的 csrwr 指令在 LoongArch 下的语义是交换 CSR 和寄存器的内容, 而非简单地将寄存器内容写入, 因此在向两个相同结构的 CSR 填入某个相同数值的时候, 我们需要重新读取之前的目的寄存器, 然后重新写入才能正确处理结果。
   (**2**) 之所以需要对 TLB Refill exception Entry HIgh-order bits (TLBREHI)(0x8e) 的状态控制寄存器填入 0, 是因为该地址内包括页长度相关的域, 但该地址原本是由 ldpte 填写, 为了防止手动填写 TLB 项导致未定义行为 (填入错误的 TLB 或虚拟机报错), 我们需要先对该页填写 0 地址。

3.2 LoongArch 的启动步骤

首先是 NPUcore 的主函数:

代码片段 3.7　os/src/main.rs

```rust
#[no_mangle]
pub fn rust_main() -> ! {
    println!("[kernel] NPUcore-IMAPCT!!! ENTER!");
    bootstrap_init();
    mem_clear();
    console::log_init();
    move_to_high_address(); \\ img move in kernel
    println!("[kernel] Console initialized.");
    mm::init();

    machine_init();
    println!("[kernel] Hello, world!");

    //machine independent initialization
    fs::directory_tree::init_fs();
    task::add_initproc();

    // note that in run_tasks(), there is yet *another* pre_start_init(),
    // which is used to turn on interrupts in some archs like LoongArch.
    task::run_tasks();
```

```
    panic!("Unreachable in rust_main!");
}
```

我们可以看到, 除了 bootstrap_init() 和 machine_init(), 其他都是平台无关的, 因此这里主要介绍平台相关的启动流程。其中, machine_init() 的发生在 mm::init() 后。

代码片段 3.8　os/src/arch/la64/mod.rs - bootstrap_init

```
    pub fn bootstrap_init() {
        /* if CPUId::read().get_core_id() != 0 {
         *     loop {}
         * } */
        ECfg::empty()
            .set_line_based_interrupt_vector(LineBasedInterrupt::TIMER)
            .write();
        EUEn::read().set_float_point_stat(true).write();
        // Timer & other Interrupts
        TIClr::read().clear_timer().write();
        TCfg::read().set_enable(false).write();
        CrMd::read()
            .set_watchpoint_enabled(false)
            .set_paging(true)
            .set_ie(false)
            .write();

        // Trap/Exception Hanlder initialization.
        set_kernel_trap_entry();
        set_machine_err_trap_ent();
        TLBREntry::read().set_addr(srfill as usize).write();

        // MMU Setup
        DMW2::read()
            .set_plv0(true)
            .set_plv1(false)
            .set_plv2(false)
            .set_plv3(false)
            .set_vesg(SUC_DMW_VESG)
            .set_mat(MemoryAccessType::StronglyOrderedUnCached)
            .write();
        DMW3::empty().write();
        //DMW1::empty().write();

        STLBPS::read().set_ps(PTE_WIDTH_BITS).write();
        TLBREHi::read().set_page_size(PTE_WIDTH_BITS).write();
        PWCL::read()
            .set_ptbase(PAGE_SIZE_BITS)
            .set_ptwidth(DIR_WIDTH)
            .set_dir1_base(PAGE_SIZE_BITS + DIR_WIDTH)
            .set_dir1_width(DIR_WIDTH) // 512*512*4096 should be enough for
                256MiB of 2k1000.
            .set_dir2_base(0)
            .set_dir2_width(0)
            .set_pte_width(PTE_WIDTH)
            .write();
        PWCH::read()
            .set_dir3_base(PAGE_SIZE_BITS + DIR_WIDTH * 2)
            .set_dir3_width(DIR_WIDTH)
            .set_dir4_base(0)
```

```
            .set_dir4_width(0)
            .write();

        println!("[kernel] UART address: {:#x}", UART_BASE);
        println!("[bootstrap_init] {:?}", PRCfg1::read());
    }
```

我们可以看到, 在上面的启动过程中, 龙芯实际上中断要触发有几个使能层：

1. 中断本身的使能, 来自 ECfg。
2. 中断源的使能 (如果存在), e.g. 时钟中断来自 TCfg。
3. CrMd: 当前模式寄存器的中断使能。
4. 最后是 Trap Handler 相关的设置和内存相关的初始化。

对于机器相关初始化, 主要是初始化时钟相关的内容。

代码片段 3.9　os/src/arch/la64/mod.rs - machine_init

```
pub fn machine_init() {
    // remap_test not supported for lack of DMW read only privilege support
    trap::init();
    get_timer_freq_first_time();
    /* println!(
     *     "[machine_init] VALEN: {}, PALEN: {}",
     *     cfg0.get_valen(),
     *     cfg0.get_palen()
     * ); */
    for i in 0..=6 {
        let j: usize;
        unsafe { core::arch::asm!("cpucfg {0},{1}",out(reg) j,in(reg) i) };
        println!("[CPUCFG {:#x}] {}", i, j);
    }
    for i in 0x10..=0x14 {
        let j: usize;
        unsafe { core::arch::asm!("cpucfg {0},{1}",out(reg) j,in(reg) i) };
        println!("[CPUCFG {:#x}] {}", i, j);
    }
    println!("{:?}", Misc::read());
    println!("{:?}", RVACfg::read());
    println!("[machine_init] MMAP_BASE: {:#x}", MMAP_BASE);
    trap::enable_timer_interrupt();
}
```

之所以将这些内容放到这里, 是因为 LoongArch 下的恒等时钟频率是可以通过指令获取的, 如果因此我们获取后存入静态数据区域, 所以需要在 bss 段被清 0 后才开始处理时钟相关中断。

第 4 章　NPUcore-IMPACT 增量

在上述基础上，我们继续做了许多努力，让 NPUcore-IMPACT 通过了初赛的所有测试用例，以及实验性地初步支持了 EXT4 文件系统。

后文我们会分别详细地介绍这两部分的内容。

4.1 初赛测试用例

我们针对性地对初赛的测试用例进行了调试，将问题归类定位到了如下两点。

然后我们分别对每个问题进行了细致的分析，最终逐个击破，通过了初赛的所有测试用例。

1. statx 系统调用

LoongArch 赛道的初赛测试用例中，mmap 与 munmap 这两个测例涉及到了一个新的系统调用 statx。

代码片段 4.1　statx 手册

```
NAME
       statx - get file status (extended)

LIBRARY
       Standard C library (libc, -lc)

SYNOPSIS
       #define _GNU_SOURCE          /* See feature_test_macros(7) */
       #include <fcntl.h>           /* Definition of AT_* constants */
       #include <sys/stat.h>

       int statx(int dirfd, const char *restrict pathname, int flags,
                 unsigned int mask, struct statx *restrict statxbuf);

STANDARDS
       Linux.

HISTORY
       Linux 4.11, glibc 2.28.
```

如手册 **??** 中所示，这个系统调用涉及了文件信息的获取。

我们为 NPUcore-IMPACT 实现了这个新的系统调用，并与文件系统进行了整合。

代码片段 4.2　statx 系统调用入口

```
let ret = match syscall_id {
    // ...
    SYSCALL_STATX => sys_statx(
        args[0],
        args[1] as *const u8,
        args[2] as u32,
        args[3] as u32,
        args[4] as *mut u8,
    ),
```

```
    // ...
};
```

代码片段 4.3　statx 系统调用实现

```
pub trait File: DowncastSync {
    // ...
    fn get_statx(&self) -> Statx;
    // ...
}
```

随后我们进行了测试，成功通过了 mmap 与 munmap 测试用例。

2. 文件描述符分配

通过对 openat 测试用例进行调试，我们最终发现问题出在操作系统对文件描述符的分配上。

Unix 标准要求操作系统分配文件描述符时，总是分配该进程还未使用的最小的文件描述符；而 NPUcore 回收进程关闭的文件描述符时，使用了一个线性表；操作系统重新分配之前回收的文件描述符时，没有使用表中最小的文件描述符，最终导致出现了问题。

代码片段 4.4　回收文件描述符

```
match self.inner[fd].take() {
    Some(file_descriptor) => {
        self.recycled.push(fd as u8);
        // TODO: maybe replace this with balanced binary tree?
        self.recycled.sort_by(|a, b| b.cmp(a));
        Ok(file_descriptor)
    }
    None => Err(EBADF),
}
```

我们选择了在回收文件描述符后进行一次排序来解决这个问题。

这个方案不一定是性能最佳的方案，我们还有以下方案可选：

1. 回收时不进行排序，重新分配时使用 $O(n)$ 时间寻找最小的文件描述符；
2. 使用二叉平衡树替换线性表，从而在 $O(\log n)$ 时间进行回收与重新分配，但也许会带来内存分配的额外开销。

未来此处成为性能瓶颈时，根据性能测试结果选用最优方案会是更好的选择。

4.2 EXT4 文件系统

EXT4（fourth extended filesystem）是 Linux 内核的一个日志文件系统，是 EXT3 文件系统的继任者。EXT4 文件系统具有许多改进和新特性，使其在性能、可靠性和可扩展性方面优于前代文件系统。

与 NPUcore 先前使用的 FAT32 文件系统相比，EXT4 文件系统不仅允许了更大的文件大小与卷大小，更有着显著的性能和效率提升。EXT4 文件系统使用了延迟分配和多

块分配策略，显著减少了碎片并提高了写入性能；同时它支持 Extents 和更高效的分配策略，提高了文件操作的速度和效率。此外，EXT4 文件系统还支持日志记录，通过记录元数据变化确保系统崩溃时的数据一致性和完整性，检查速度快且更可靠。

我们为 NPUcore-IMPACT 实验性地加入了 EXT4 文件系统支持，使得其可以从 EXT4 文件系统启动，并读写其中的文件。

我们使用了 lwext4 作为 EXT4 文件系统驱动。lwext4 是一个针对嵌入式系统设计的轻量级 EXT4 文件系统实现，它旨在提供 EXT4 文件系统的关键特性，同时保持低资源消耗和高性能，以适应嵌入式系统的限制。

为了让 lwext4 能与 NPUcore-IMPACT 一起工作，我们对 NPUcore-IMPACT 的文件系统设计做出了一定调整。

我们借助 Rust 的 trait 语言特性，设计了一个 File trait，用于表示一个抽象的文件，或者说一个可以对其进行读写的对象。

代码片段 4.5　File trait

```rust
pub trait File: DowncastSync {
    fn deep_clone(&self) -> Arc<dyn File>;
    fn readable(&self) -> bool;
    fn writable(&self) -> bool;
    fn read(&self, offset: Option<&mut usize>, buf: &mut [u8]) -> usize;
    fn write(&self, offset: Option<&mut usize>, buf: &[u8]) -> usize;
    fn r_ready(&self) -> bool;
    fn w_ready(&self) -> bool;
    fn read_user(&self, offset: Option<usize>, buf: UserBuffer) -> usize;
    fn write_user(&self, offset: Option<usize>, buf: UserBuffer) -> usize;
    fn get_size(&self) -> usize;
    fn get_stat(&self) -> Stat;
    fn get_statx(&self) -> Statx;
    fn get_file_type(&self) -> DiskInodeType;
    fn is_dir(&self) -> bool {
        self.get_file_type().is_dir()
        // self.get_file_type() == DiskInodeType::Directory
    }
    fn is_file(&self) -> bool {
        self.get_file_type().is_file()
        // self.get_file_type() == DiskInodeType::File
    }
    fn info_dirtree_node(&self, dirnode_ptr: Weak<DirectoryTreeNode>);
    fn get_dirtree_node(&self) -> Option<Arc<DirectoryTreeNode>>;
    /// open
    fn open(&self, flags: OpenFlags, special_use: bool) -> Arc<dyn File>;
    fn open_subfile(&self) -> Result<Vec<(String, Arc<dyn File>)>, isize>;
    /// create
    fn create(&self, name: &str, file_type: DiskInodeType) -> Result<Arc<
        dyn File>, isize>;
    fn link_child(&self, name: &str, child: &Self) -> Result<(), isize>
    where
        Self: Sized;
    /// delete(unlink)
    fn unlink(&self, delete: bool) -> Result<(), isize>;
    /// dirent
    fn get_dirent(&self, count: usize) -> Vec<Dirent>;
```

```
    /// offset
    fn get_offset(&self) -> usize {
        self.lseek(0, SeekWhence::SEEK_CUR).unwrap()
    }
    fn lseek(&self, offset: isize, whence: SeekWhence) -> Result<usize,
        isize>;
    /// size
    fn modify_size(&self, diff: isize) -> Result<(), isize>;
    fn truncate_size(&self, new_size: usize) -> Result<(), isize>;
    // time
    fn set_timestamp(&self, ctime: Option<usize>, atime: Option<usize>,
        mtime: Option<usize>);
    /// cache
    fn get_single_cache(&self, offset: usize) -> Result<Arc<Mutex<PageCache
        >>, ()>;
    fn get_all_caches(&self) -> Result<Vec<Arc<Mutex<PageCache>>>, ()>;
    /// memory related
    fn oom(&self) -> usize;
    /// poll, select related
    fn hang_up(&self) -> bool;
    /// iotcl
    fn ioctl(&self, _cmd: u32, _argp: usize) -> isize {
        ENOTTY
    }
    /// fcntl
    fn fcntl(&self, cmd: u32, arg: u32) -> isize;
}
```

在此基础上，我们为 lwext4 提供的 ext4_file 类型实现我们的 File trait，让 NPUcore-IMPACT 可以对其进行读写，从而实现 EXT4 文件系统的支持。

由于 lwext4 依赖 libc 进行内存分配，为了让它能工作在没有 libc 的环境下，我们还需要对其做出一定修改。

代码片段 4.6　管理 lwext4 内存

```
#if CONFIG_USE_USER_MALLOC

#define ext4_malloc  ext4_user_malloc
#define ext4_calloc  ext4_user_calloc
#define ext4_realloc ext4_user_realloc
#define ext4_free    ext4_user_free

#else

#define ext4_malloc  malloc
#define ext4_calloc  calloc
#define ext4_realloc realloc
#define ext4_free    free

#endif
```

我们希望让 NPUcore-IMPACT 为 lwext4 管理内存，为此我们实现 ext4_user_malloc、ext4_user_calloc、ext4_user_realloc、ext4_user_free 这四个内存管理函数，并将其与 lwext4 链接，从而让 lwext4 可以使用我们为它分配的内存，并在合适的时候回收这些内存。

代码片段 4.7　NPUcore-IMPACT 为 lwext4 分配内存

```
#[no_mangle]
pub extern "C" fn ext4_user_malloc(size: ::core::ffi::c_size_t) -> *mut ::core::ffi::c_void {
    HEAP_ALLOCATOR
        .lock()
        .alloc(Layout::array::<u8>(size).unwrap())
        .unwrap()
        .as_ptr() as *mut ::core::ffi::c_void
}
```

为了便于调试，我们需要在 lwext4 执行时打印日志，得益于 Rust 与 C 跨语言互操作十分方便，我们直接在 Rust 侧编写了打印日志的工具函数。

代码片段 4.8　在 lwext4 的 C 语言代码中打印日志

```
#[no_mangle]
pub extern "C" fn os_log(str: *const ::core::ffi::c_char) {
    let str = unsafe { CStr::from_ptr(str) };
    log::info!("{str:?}");
}

#[no_mangle]
pub extern "C" fn os_var_log(name: *const ::core::ffi::c_char, value: ::core::ffi::c_int) {
    let name = unsafe { CStr::from_ptr(name) };
    log::info!("{name:?}: {value}");
}
```

使用 #[no_mangle] 可以让编译器不对函数名字进行混淆，使得我们可以在 C 语言侧直接调用 os_log 与 os_var_log 日志函数。

第 5 章　NPUcore-IMPACT 特性

5.1 内存管理方面

5.1.1 页表与地址空间管理

在 NPUcore-IMPACT 中，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 $4KiB(2^{12}B)$。对于页表以及地址的管理可参见表**??**。

5.1.2 地址的数据结构抽象

代码片段 5.1　存储地址数据结构

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

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

#[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.

5.1.3 页表项

其中，对于符号位的详细解释如下：

- V(Valid)：有效位。仅当 V 为 1 时，该页表项合法；
- R(Read)/W(Write)/X(eXecute)：分别表示索引到这个页表项的对应虚拟页面是否允许读/写/执行；
- U(User)：表示索引到这个页表项的对应虚拟页面是否在 CPU 处于 U 特权级的情况下允许访问；

地址	页号	页内偏移	总位数
虚拟地址 VA	VPN-38:12 27bit	Offset-11:0 12bit	39bit
物理地址 PA	PPN-55:12 44bit	Offset-11:0 12bit	56bit

表 5-1　地址详细列表

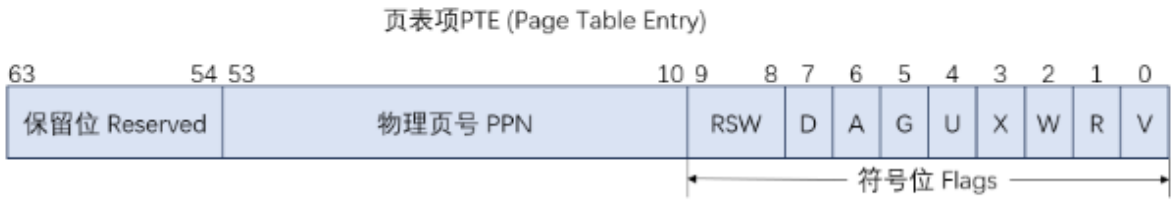


图 5-1　页表项的格式

- G(Global)：全局标志。为 1 时表明该页面为全局页面；
- A(Accessed)：处理器使用此位来记录自页表项上的这一位被清零后，其对应虚拟页面是否被访问过；
- D(Dirty)：处理器使用此位来记录自页表项上的这一位被清零后，其对应虚拟页面是否被修改过；
- RSW(Reserved for Supervisor softWare)：保留位。该部分被处理器忽略，软件可以使用。

5.1.4 地址空间

内核地址空间有四个标志位，具体为 *R* - 读；*W* - 写；*X* - 执行；*U* - 用户。

对于内核程序区域而言：

在 **os/src/linker.in.ld** 下，我们可以找到对应的基址，其值为：**0x0000000090000000**。而对于本次新的开发板 LoongArch-2K1000 而言，其 MMIO 区域并无多少设备，仅有一 UART 寄存器，具体可在 **os/src/arch/la64/board/2k1000.rs** 下找到 UART 寄存器地址：**0x1fe20000**.

目前仅介绍了部分重要地址，由于更换开发板，大部分地址常数已经重写，具体可参见 **os/src/arch/la64/config.rs**。

代码片段 5.2　os/src/arch/la64/config.rs

```rust
pub const MEMORY_SIZE: usize = 0x1000_0000;
pub const USER_STACK_SIZE: usize = PAGE_SIZE * 40;
pub const USER_HEAP_SIZE: usize = PAGE_SIZE * 20;
pub const SYSTEM_TASK_LIMIT: usize = 128;
pub const DEFAULT_FD_LIMIT: usize = 128;
pub const SYSTEM_FD_LIMIT: usize = 256;
pub const PAGE_SIZE: usize = 0x1000;
pub const PAGE_SIZE_BITS: usize = PAGE_SIZE.trailing_zeros() as usize;
pub const PTE_WIDTH: usize = 8;
pub const PTE_WIDTH_BITS: usize = PTE_WIDTH.trailing_zeros() as usize;
pub const DIR_WIDTH: usize = PAGE_SIZE_BITS - PTE_WIDTH_BITS;
pub const DIRTY_WIDTH: usize = 0x100_0000;
#[cfg(debug_assertions)]
pub const KSTACK_PG_NUM_SHIFT: usize = 16usize.trailing_zeros() as usize;
#[cfg(not(debug_assertions))]
pub const KSTACK_PG_NUM_SHIFT: usize = 2usize.trailing_zeros() as usize;

pub const KERNEL_STACK_SIZE: usize = PAGE_SIZE << KSTACK_PG_NUM_SHIFT;
```

```rust
pub const KERNEL_HEAP_SIZE: usize = PAGE_SIZE * 0x3000;

// Addresses
/// Maximum length of a physical address
pub const PALEN: usize = 48;
/// Maximum length of a virtual address
pub const VALEN: usize = 48;
/// Maximum address in virtual address space.
/// May be used to extract virtual address from a segmented address
/// '0'-extension may be performed using this mask.
pub const VA_MASK: usize = (1 << VALEN) - 1;
/// Mask for extracting segment number from usize address.
/// '1'-extension may be performed using this mask.
/// e.g. 'flag' |= 'SEG_MASK'
pub const SEG_MASK: usize = !VA_MASK;
/// Mask for extracting segment number from VPN.
/// All-one for segment field.
/// '1'-extension may be performed using this mask.
/// e.g. 'flag' |= 'SEG_MASK'
pub const VPN_SEG_MASK: usize = SEG_MASK >> PAGE_SIZE_BITS;

pub const HIGH_BASE_EIGHT: usize = 0x8000_0000_0000_0000;
pub const HIGH_BASE_ZERO: usize = 0x0000_0000_0000_0000;

// manually make usable memory space equal
pub const SUC_DMW_VESG: usize = 8;
pub const MEMORY_HIGH_BASE: usize = HIGH_BASE_ZERO;
pub const MEMORY_HIGH_BASE_VPN: usize = MEMORY_HIGH_BASE >> PAGE_SIZE_BITS;
pub const USER_STACK_BASE: usize = TASK_SIZE - PAGE_SIZE | LA_START;
pub const MEMORY_START: usize = 0x0000_0000_9000_0000;
pub const MEMORY_END: usize = MEMORY_SIZE + MEMORY_START;

pub const SV39_SPACE: usize = 1 << 39;
pub const USR_SPACE_LEN: usize = SV39_SPACE >> 2;
pub const LA_START: usize = 0x1_2000_0000;
pub const USR_VIRT_SPACE_END: usize = USR_SPACE_LEN - 1;
pub const TRAMPOLINE: usize = SIGNAL_TRAMPOLINE; // The trampoline is NOT
    mapped in LA.
pub const SIGNAL_TRAMPOLINE: usize = USR_VIRT_SPACE_END - PAGE_SIZE + 1;
pub const TRAP_CONTEXT_BASE: usize = SIGNAL_TRAMPOLINE - PAGE_SIZE;
pub const USR_MMAP_END: usize = TRAP_CONTEXT_BASE - PAGE_SIZE;
pub const USR_MMAP_BASE: usize = USR_MMAP_END - USR_SPACE_LEN / 8 + 0x3000;
pub const TASK_SIZE: usize = USR_MMAP_BASE - USR_SPACE_LEN / 8;
pub const ELF_DYN_BASE: usize = (((TASK_SIZE - LA_START) / 3 * 2) |
    LA_START) \& (!(PAGE_SIZE - 1));

pub const MMAP_BASE: usize = 0xFFFF_FF80_0000_0000;
pub const MMAP_END: usize = 0xFFFF_FFFF_FFFF_0000;
pub const SKIP_NUM: usize = 1;

pub const DISK_IMAGE_BASE: usize = 0x800_0000 + MEMORY_START;
pub const BUFFER_CACHE_NUM: usize = 256 * 1024 * 1024 / 2048 * 4 / 2048;

pub static mut CLOCK_FREQ: usize = 0;
```

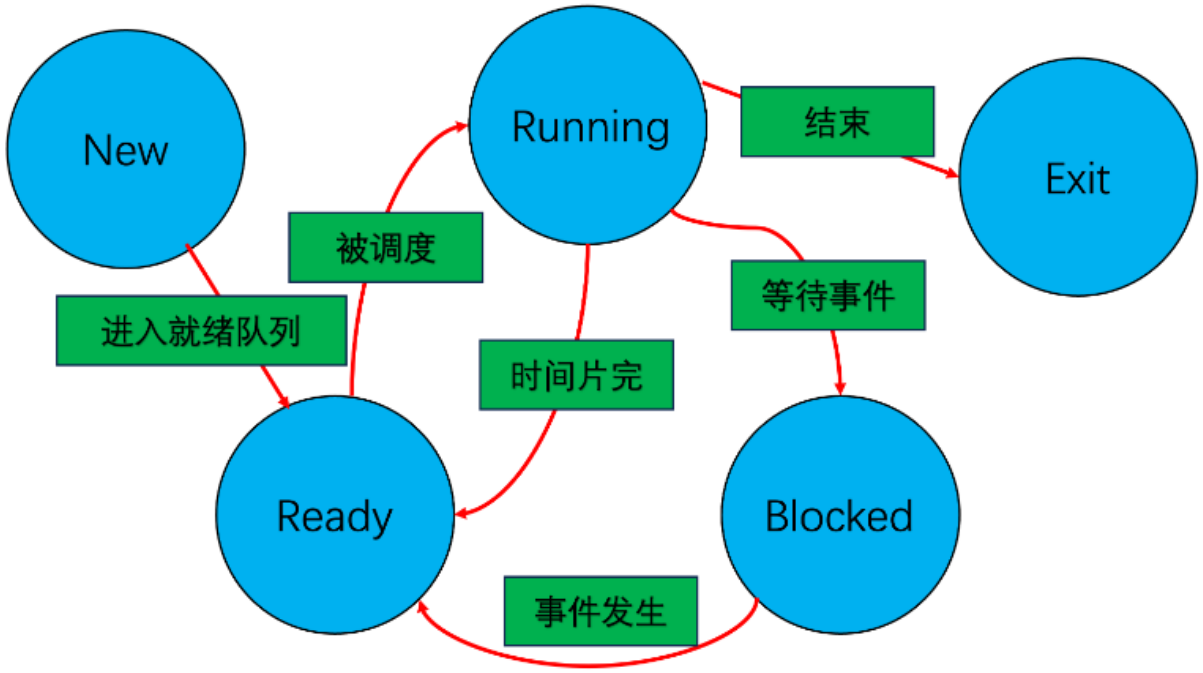


图 5-2　进程生命周期示意图

单核/多核	单核
资源复用思路 资源回收思路	一段所有进程共享的跳板代码 + 一个进程的私有的保存现场的帧 释放一部分资源等待父进程完成剩下资源回收
进程管理方式	TCB*

表 5-2　NPUcore-IMPACT 的复用思路

5.2 进程调度方面

对于进程调度，NPUcore-IMPACT 给出了一系列方法用于提升操作系统的性能。

5.2.1 进程生命周期

对于 NPUcore-IMPACT 的进程生命周期，我们给出**??**以有直观理解团队在进行性能优化时，发现在操作系统运行示例程序时，IO 操作导致 CPU 挂起的性能损失非常之大，因此我们将调度器进行了大改 (如**??**)，使其完全支持了阻塞式的进程调度模式, 即构建挂起队列，使进程不能获得资源时可被挂起，直到满足可操作的条件后再进行操作。。

5.2.2 资源复用

这里引出我们内核特殊的一点，即使用 TCB（TaskControlBlock/任务控制块）（如下面展示的 TCB 结构体代码）来代替传统的 PCB 数据结构，我们将线程视为共享栈的进程。

代码片段 5.3　TCB 结构体

```
1 pub struct TaskControlBlock {
```

```
    // immutable
    pub pid: PidHandle ,
    pub tid: usize,
    pub tgid: usize,
    pub kstack: KernelStack ,
    pub ustack_base: usize,
    pub exit_signal: Signals ,
    // mutable
    inner: Mutex<TaskControlBlockInner >,
    // shareable and mutable
    pub exe: Arc<Mutex<FileDescriptor >>,
    pub tid_allocator: Arc<Mutex<RecycleAllocator >>,
    pub files: Arc<Mutex<FdTable >>,
    pub fs: Arc<Mutex<FsStatus >>,
    pub vm: Arc<Mutex<MemorySet >>,
    pub sighand: Arc<Mutex<Vec<Option<Box<SigAction >>>>>,
    pub futex: Arc<Mutex<Futex>>,
}
```

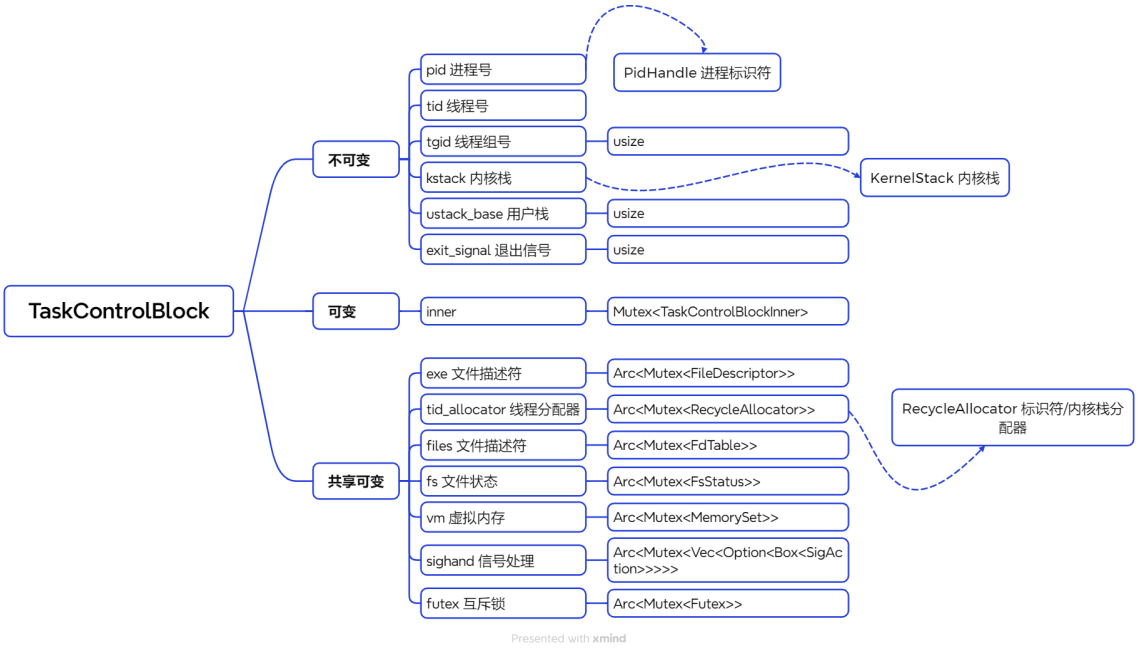


图 5-3　TCB 结构示意图

如上述定义的 TCB，使得线程在内核中的理解，变为了这样一个可以与同线程组 tgid 共享内存空间的进程即为线程。

5.3 文件系统方面

目前来讲，NPUcore-IMPACT 已经完美支持了 fat32 文件系统，但因耦合度较高，目前我们正在进行解耦合操作，且会在后面支持 ext4 文件系统靠拢。对于 fat32 文件系统，是我们先前 NPUcore 的增量，不属于我们团队的贡献，因此这里不再赘述，仅在下面给出 NPUcore-IMPACT 的具体参数。

5.3.1 fat32 具体参数

fat32 文件系统是个较为简单的系统，当然，其系统的易移植特性与其简单耐用的特性使其成为了几乎所有系统在初始阶段的第一选择。对于 NPUcore-IMPACT，我们可以用一个简单的示意图**??**来描述他

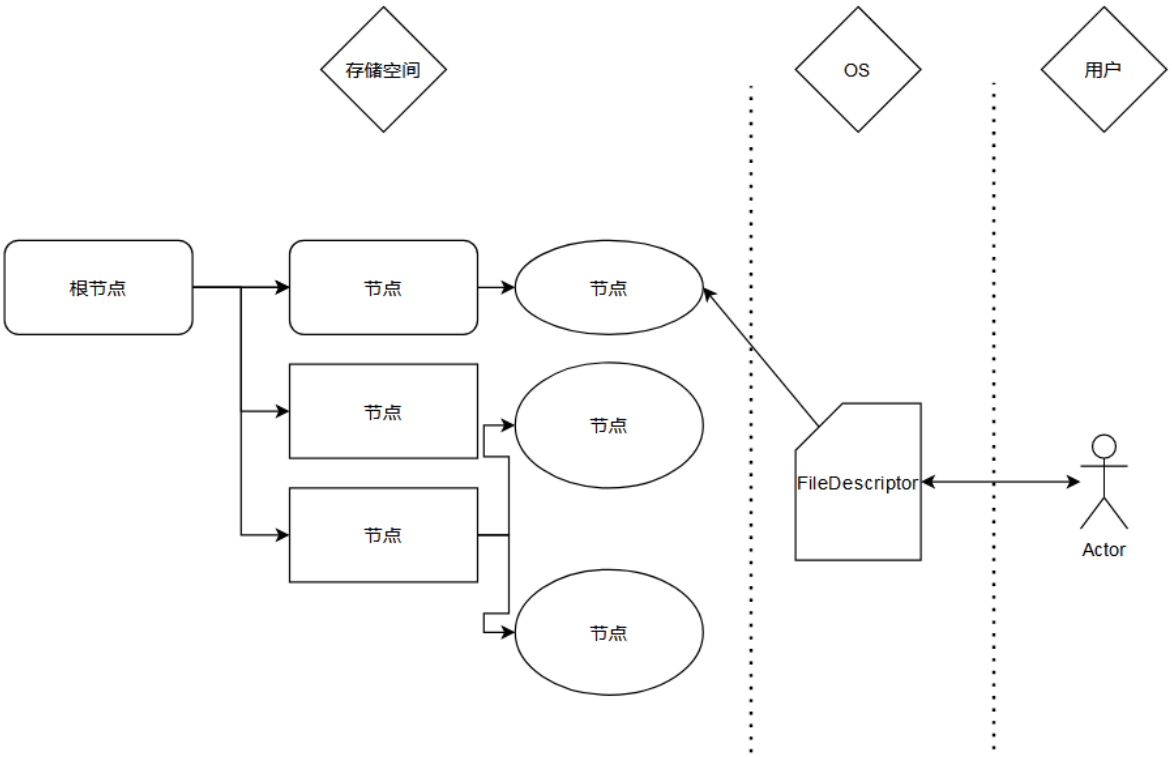


图 5-4　NPUcore-IMPACT 的文件系统示意图

(1) FileDescriptor

如果用简单的描述来简述 NPUcore-IMPACT 的文件系统，那可以将之理解为一栋楼，我们利用一个描述每个房间的 FileDescriptor 类的指针来读取一个文件的信息，可以毫不为过的将其理解为我们的文件类型指针, **FileDescriptor 的具体定义如下所示:**

代码片段 5.4　FileDescriptor

```
1    //fs/mod.rs
2    #[derive(Clone)]
3    pub struct FileDescriptor {
4        cloexec: bool,
5        nonblock: bool,
6        pub file: Arc<dyn File>,
7    }
```

(2) 文件树结构

以下是我们定义文件树节点的结构体：

代码片段 5.5　FileDescriptor

```
    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>>>>,
    }
```

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, 这部分将于后续 5.4 中详细说明。

5.3.2 NPUcore-IMPACT BitFlags 一览

我们遵循原版 POSIX 的 flag 编码，源码如下：

代码片段 5.6　FileDescriptor

```
    bitflags! {
        pub struct OpenFlags: u32 {
            const O_RDONLY      =   0o0;
            const O_WRONLY      =   0o1;
            const O_RDWR        =   0o2;

            const O_CREAT       =   0o100;
            const O_EXCL        =   0o200;
            const O_NOCTTY      =   0o400;
            const O_TRUNC       =   0o1000;

            const O_APPEND      =   0o2000;
            const O_NONBLOCK    =   0o4000;
            const O_DSYNC       =   0o10000;
            const O_SYNC        =   0o4010000;
            const O_RSYNC       =   0o4010000;
            const O_DIRECTORY   =   0o200000;
            const O_NOFOLLOW    =   0o400000;
            const O_CLOEXEC     =   0o2000000;
            const O_ASYNC       =   0o20000;
            const O_DIRECT      =   0o40000;
            const O_LARGEFILE   =   0o100000;
            const O_NOATIME     =   0o1000000;
            const O_PATH        =   0o10000000;
            const O_TMPFILE     =   0o20200000;
        }
    }

    bitflags! {
```

```
30          pub struct SeekWhence: u32 {
31              const SEEK_SET  =   0; /* set to offset bytes.  */
32              const SEEK_CUR  =   1; /* set to its current location plus
                    offset bytes.  */
33              const SEEK_END  =   2; /* set to the size of the file plus
                    offset bytes.  */
34          }
35      }
36
37      bitflags! {
38          pub struct StatMode: u32 {
39              ///bit mask for the file type bit field
40              const S_IFMT    =   0o170000;
41              ///socket
42              const S_IFSOCK  =   0o140000;
43              ///symbolic link
44              const S_IFLNK   =   0o120000;
45              ///regular file
46              const S_IFREG   =   0o100000;
47              ///block device
48              const S_IFBLK   =   0o060000;
49              ///directory
50              const S_IFDIR   =   0o040000;
51              ///character device
52              const S_IFCHR   =   0o020000;
53              ///FIFO
54              const S_IFIFO   =   0o010000;
55
56              ///set-user-ID bit (see execve(2))
57              const S_ISUID   =   0o4000;
58              ///set-group-ID bit (see below)
59              const S_ISGID   =   0o2000;
60              ///sticky bit (see below)
61              const S_ISVTX   =   0o1000;
62
63              ///owner has read, write, and execute permission
64              const S_IRWXU   =   0o0700;
65              ///owner has read permission
66              const S_IRUSR   =   0o0400;
67              ///owner has write permission
68              const S_IWUSR   =   0o0200;
69              ///owner has execute permission
70              const S_IXUSR   =   0o0100;
71
72              ///group has read, write, and execute permission
73              const S_IRWXG   =   0o0070;
74              ///group has read permission
75              const S_IRGRP   =   0o0040;
76              ///group has write permission
77              const S_IWGRP   =   0o0020;
78              ///group has execute permission
79              const S_IXGRP   =   0o0010;
80
81              ///others (not in group) have read, write,and execute
                    permission
82              const S_IRWXO   =   0o0007;
83              ///others have read permission
84              const S_IROTH   =   0o0004;
```

```
85              ///others have write permission
86              const S_IWOTH   =   0o0002;
87              ///others have execute permission
88              const S_IXOTH   =   0o0001;
89          }
90      }
```

5.3.3 NPUcore-IMPACT 的目标与计划

目前我们对于 NPUcore-IMPACT 整体的性能指标感到较为满意，但并不满足于功能较为单一的现状，目前我们正在为 NPUcore-IMPACT 添加更多的功能。由于现在的存储设备技术发展迅速，存储单元的价格已经出现了大幅度下降。同时，随着芯片技术发展，计算机性能变得更加强劲，文件大小也跟着变大，fat32 仅能满足**最大 4GB** 的单文件存储能力已经远远无法支持新一代设备了，所以我们计划为其添加 ext4 文件系统支持。同时由于网络层对于文件系统的依赖耦合程度较高 *(Unix_Like 系统大多依赖 VFS 来对设备进行控制，即把设备当做一个可以进行读写的类管道文件)*，我们暂时放弃了已经写好的 fat32 下的网络支持，计划于 ext4 完成适配再进行添加。最终我们拟定在决赛时提交完整版文件系统。