OS 内核实现赛道——LoongArch 赛道

设计文档

参赛队名：NPUcore-重生之我是秦始皇

队伍成员：包子旭、李佳航、刘家威

指导老师：张羽

2024 年 5 月 日

目录

摘要

本团队是使用 Rust 编写的基于 LoongArch 的的内核操作系统。于 2024-05- 31 结束竞赛第一阶段时，完成大部分功能测试，小部分性能测试的适配。所属 OS 内核实现赛道——LoongArch 赛道的排行榜如图 0- 1 所示：

比赛提交到排行榜更新有20秒左右的延迟

#	用户名	队伍	提交次数(ASC)	最后提交时间(ASC)	rank
1	T202410699992496	NPUcore-IMPACT!!!/ 西北工业大学	45	2024-05-03 23:34:44	102.0000
2	T202410699992491	NPUcore-重生之我是菜狗/ 西北工业大学	56	2024-03-07 15:27:46	102.0000
3	T202410460992502	NPUcore-重生之我是秦始皇/ 河南理工大学	41	2024-05-17 09:42:54	89.0000
4	T202410614992892	HelloWorld/ 电子科技大学	1	2024-04-23 12:28:03	53.0000
5	T202410213992605	Refill/ 哈尔滨工业大学	8	2024-04-02 19:13:24	53.0000
6	T202412802992528	菜鸡不会riscv/ 吉利学院	2	2024-05-04 01:23:42	0.0000
7	T202410486992576	俺争取不掉队/ 武汉大学	6	2024-05-22 19:48:19	0.0000
8	T202411664992499	重启之我是loader/ 西安邮电大学	20	2024-05-22 15:19:54	0.0000

图 0-1 竞赛第一阶段测试排行榜

1.介绍

1.1LoongArch 指令集介绍

LoongArch 是 RISC 中的一个具体实现。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。

龙架构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™。

1.2 项目背景及意义

在河南理工大学的计算机系统教育中，我们发现理论授课和实践操作的平衡还需加强。操作系统是计算机科学的关键基础之一，理解程序运行和硬件资源管理至关重要，但其复杂的管理机制使得在理论教学和实践应用中都存在一定挑战。

河南理工大学的本科学生积极参与学术研究，在各自感兴趣的领域组建研究小组，深入探索。这种学习态度有助于培养学生的研究能力和解决实际问题的能力。

然而，我们认识到在计算机系统专业培养中，河南理工大学还需在以下三个方面进一步提升：强化理论教学，加深学生对专业理论的理解；设计独特的实验，提升学生的实践技能；鼓励导师引领下进行前沿探索，促进学生深入研究，获取新的知识和技能。

我们认为，设计独特的实验是目前需要优先解决的问题。因此，我们计划在实际硬件环境中进行操作系统的开发，为课程实验设计提供更多自由度，让学生有更多空间去理解和探索新的知识。

我们希望这种方式不仅能解决现有问题，还能成为一个可持续的教学模式。因此，我们计划通过 Github 项目组、课程实验以及计算机系统小组等形式来推广和实践这种模式。

我们深信，这对于人才培养至关重要。提高学生的理论知识和实践技能，激发创新思维，培养解决问题的能力，这将为河南理工大学的计算机科学教育领域带来新的活力，推动其不断发展。

总的来说，我们希望通过这些行动，为河南理工大学的计算机科学教育领域注入新的活力，促进创新进步，提供更好的教学环境。

1.3 国内外研究状况

操作系统内核是操作系统的核心部分，负责处理硬件与软件之间的交互，包括内存管理、进程调度和文件系统等。国内外的操作系统内核研究可以从商业系统、开源系统，以及相关研究领域和趋势等多个角度介绍。

首先，商业系统方面，微软和苹果是两大主要玩家。微软的 Windows 操作系统内核是 NT 内核，采用混合内核结构，整合了微内核和宏内核的特点。苹果的 macOS 操作系统使用的是 XNU 内核，也是混合内核，包含了 Mach 微内核和 BSD 宏内核的部分。微软致力于提供高效、安全和稳定的用户体验，而苹果则注重性能提升、安全加强，以及新硬件和技术的支持。

其次，开源系统方面，Linux 是最重要的项目之一。Linux 内核是开源的宏内核，有数万名开发者参与研究和开发。其设计目标是提供通用、稳定和高效的操作系统内核，目前被广泛应用于各种设备和环境。

在国内，包括中科院软件研究所、哈工大等多家研究机构和高校也在进行操作系统内核相关的研究工作。例如，中科院软件研究所的可信鸿蒙操作系统以 Linux 内核为基础，旨在提供高度可信的操作系统；哈工大的麒麟操作系统也以 Linux 内核为基础，主要满足国内的自主可控需求。

未来的操作系统内核研究趋势包括微内核研究、安全性研究和新硬件支持。微内核有助于提高系统的可靠性和安全性，而安全性研究针对网络安全问题，关注内核自我保护、内核隔离和内核加固等技术。同时，随着硬件技术的发展，如多核处理器和非易失性内存，操作系统内核研究也在探索更好地利用这些新硬件的方法。

2.需求分析

本团队致力于实现一个精简的操作系统，使得其具有良好的拓展性，同时，又能够在重要功能上进行支持。实现在操作系统大赛的初赛阶段通过 102 个测试点，完成操作系统的基本功能。在实现测试的系统调用过程中，附带实现了一些其他的系统调用，以方便整个操作系统的实现。

测试结果		
测试样例名	通过测试点	全部测试点
test_sleep	2	2
test_mkdir	3	3
test_pipe	0	4
test_dup2	0	2
test_openat	4	4
test_getdents	5	5
test_clone	4	4
test_wait	4	4
test_times	6	6
test_exit	2	2

测试结果		
测试样例名	通过测试点	全部测试点
test_getcwd	2	2
test_read	3	3
test_umount	5	5
test_unlink	2	2
test_uname	2	2
test_waitpid	4	4
test_mmap	0	3
test_fork	3	3
test_dup	2	2
test_fstat	3	3
test_chdir	3	3
test_yield	4	4
test_execve	3	3
test_gettimeofday	3	3
test_munmap	0	4
test_getppid	2	2
test_close	2	2
test_getpid	3	3
test_open	3	3
test_brk	3	3
test_write	2	2
test_mount	5	5

表 2-1 测试点

为了充分支持评测系统，我们需要实现以下功能：支持非在线库的项目依赖项，避免外部设备的自动启动依赖，并且需要实现自动运行评测程序，包括序列化和并行化的执行等功能。

3. 系统设计与实现

3.1 进程管理

进程管理模块主要功能有：进程初始化、进程载入和解析、进程切换、进程状态构建模块。下面分别介绍。

3.1.1 进程控制块

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。

在 NPUcore+ 中，我们使用 TCB 结构体来描述一个进程，该结构体可以完整地描述一个进程的内容和结构。利用该结构体，我们可以像 linux 一样使用进程来模拟线程。这与 windows 操作系统的设计思路是不同的，其线程的实现依靠单独的一套 thread_control_block 完成。

因此线程在 NPUcore+ 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。

进程控制包括进程的创建、结束和信号处理。

- exit: 结束调用它的进程。
- exit_group: 结束与调用进程相同进程组的所有进程。
- kill: 向指定进程发送信号。
- tkill: 向指定线程发送信号。
- clone: 创建一个新进程，是 Linux 线程创建的基础。
- execve: 在当前进程中加载并运行一个新程序。
- wait4: 等待进程状态发生变化。
- setpgid: 设置进程组 ID。
- getpgid: 获取指定进程的进程组 ID。

管理进程时，NPUcore+ 使用了 TCB 的数据结构，不同于传统的 PCB 数据结构，NPUcore+ 将线程视为共享栈的进程。TCB 的数据结构如下：

```rust
pub struct TaskControlBlock {
    // immutable
    pub pid: PidHandle,         //进程号
    pub tid: usize,             //线程号
    pub tgid: usize,            //线程组号
    pub kstack: KernelStackImpl,//内核栈
    pub ustack_base: usize,     //用户栈
    pub exit_signal: Signals,   //退出信号
    // mutable  可变部分
    inner: Mutex<TaskControlBlockInner>,
    // shareable and mutable  共享可变部分
    pub exe: Arc<Mutex<FileDescriptor>>,            //可执行文件描述符
    pub tid_allocator: Arc<Mutex<RecycleAllocator>>,// 线程 ID 分配器
    pub files: Arc<Mutex<FdTable>>,                 // 文件描述符表
    pub fs: Arc<Mutex<FsStatus>>,                   // 文件系统状态，暂时觉着这个FsStatus有点多余，觉着可以直接换成FileDescriptor
    pub vm: Arc<Mutex<MemorySet<PageTableImpl>>>,   // 虚拟内存映射
    pub sighand: Arc<Mutex<Vec<Option<Box<SigAction>>>>>,// 信号处理器
    pub futex: Arc<Mutex<Futex>>,                   // 用于 futex 系统调用的同步原语
}
```

图 3-1 TCB 数据结构

在上面的定义中，我们可以看到 TCB 中包含了进程的进程号，线程号，线程组号，内核栈，用户栈，退出信号，以及一些共享的资源。后面将具体讲解如何管理这些资源以及进程之间的调度。

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。

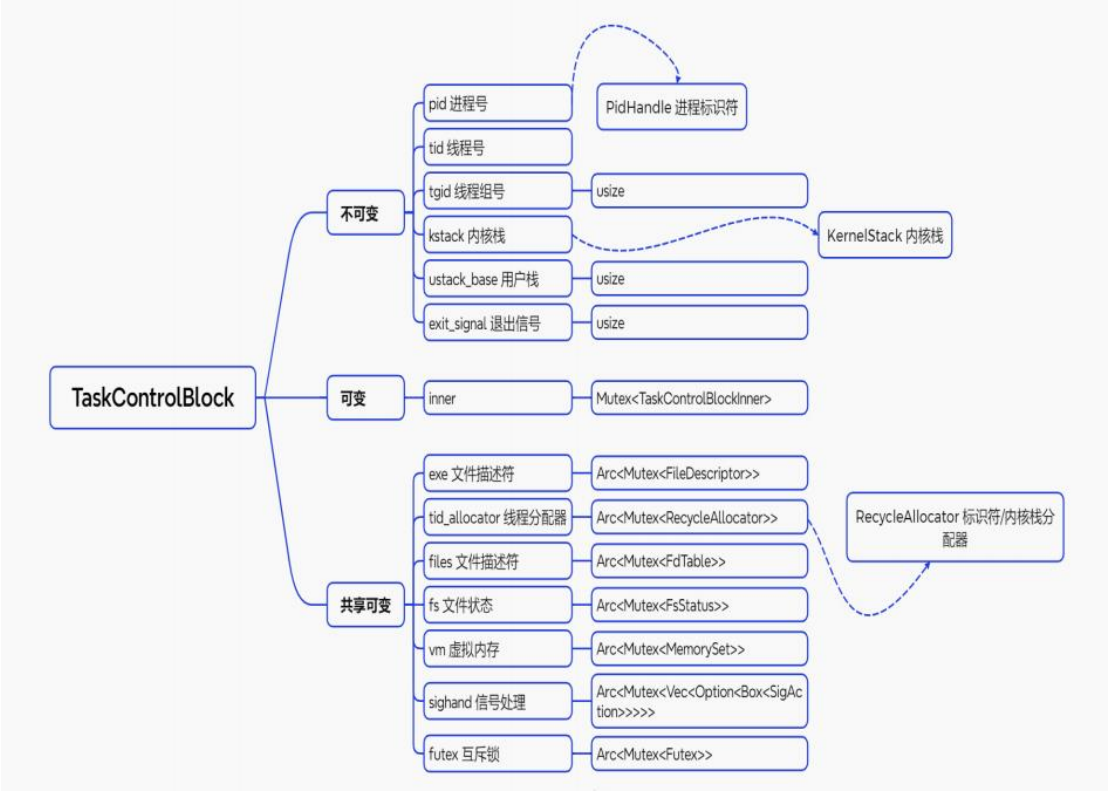


图 3-2NPUcore+ 中进程控制块的具体组成

3.1.2 进程管理重要数据结构

(1) 进程标识符 PidHandle

同一时间存在的所有进程都有一个唯一的进程标识符，将其抽象为 PidHandle 类型。

```
pub struct PidHandle(pub usize);
```

图 3-3 进程标识符

进程标识符是一个 64 位的无符号整数，用来标识进程 ID。进程标识符的分配和回收由标识符分配器 RecycleAllocator 完成。我们一般简称其为 PID。

(2) 内核栈 KernelStack

内核在创建进程的时候，在创建 task_struct 的同时，会为进程创建两个栈，第一个栈也就是上面分析到的进程用户栈，存在于用户空间使用，另外还有一个内核栈，存放在内核空间。

内核栈存在的意义：如系统调用在陷入内核后，系统调用中也是存在函数调用和自动变量，这些都需要栈支持。

每个进程都要有独自的内核栈的必要性：所有进程在运行时，都有可能通过系统调用陷入内核态继续执行，假设第一个进程陷入内核执行的时候，需要等待某个资源，主动 schedule() ，让出 CPU ，第二个进程假设也通过系统调用进入了内核态，如果进程共享内核栈，那么第二个进程在系统调用压栈时会破坏第一个进程栈数据。

每个应用都有自己的内核栈，因此 KernelStack 的内部就是其所属进程的

PID 号。

```rust
pub struct KernelStack(Vec<u8>);
impl KernelStack {
    pub fn new() -> Self {
        Self(alloc::vec![0_u8; KERNEL_STACK_SIZE])
    }
    pub fn get_top(&self) -> usize {
        let (_, kernel_stack_top: usize) = Self::kernel_stack_position(&self.0);
        kernel_stack_top
    }
    /// Return (bottom, top) of a kernel stack in kernel space.
    fn kernel_stack_position(v: &Vec<u8>) -> (usize, usize) {
        /* let top: usize = TRAMPOLINE - kstack_id * (KERNEL_STACK_SIZE + PAGE_SIZE);
        let bottom: usize = &v[0] as *const u8 as usize;
        let top: usize = bottom + KERNEL_STACK_SIZE;
        (bottom, top)
    }
}
```

图 3-4 内核栈

3.2 内存管理

3.2.1 物理内存管理

物理内存管理是操作系统的核心部分之一。为了提高系统对物理内存的动态使用效率，隔离各应用的物理内存空间以保证应用间的安全性，我们对硬件层面的物理内存空间进行了一层抽象，建立了虚拟地址空间到物理内存空间的映射。从此，每个应用程序都享有独属于自己的，且足够庞大（一般来说）的存储空间，而不用与其他应用程序“抢占”资源。而将每个应用的逻辑地址空间分配到实际的物理内存空间这一任务，正是由操作系统来负责。

根据页式管理的知识，我们将虚拟地址和物理地址均分成两部分：它们的低 12 位，即 [11:0] 被称为页内偏移 (Page Offset)，它描述一个地址指向的字节在其所在页面中的相对位置。在 SV39 分页模式下，我们规定虚拟地址一共 39 位，则虚拟地址的高 27 位，即 [38:12] 为它的虚拟页号 VPN (Virtual Page Number)；我们规定物理地址一共 56 位，则物理地址的高 44 位，即 [55:12] 为它的物理页号 PPN (Physical Page Number)。

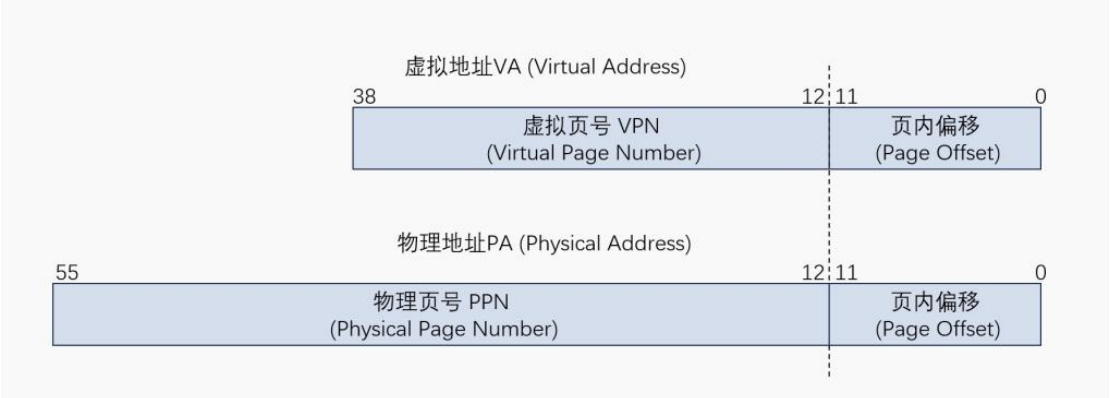


图 3-5 虚拟地址与物理地址的格式

我们实现地址信息类型与 usize 类型的相互转换，我们已经实现了多级页表的分页管理机制，大大简化了物理内存分配的复杂度，每次分配和回收都以一个页面为单位，新建进程时地址空间为空，没有对应的物理页面的。随着进程的不断运行，逐渐申请物理页面，所占的物理内存不断增加。这就要求内核要给用户进程提供物理内存分配的功能。因此，我们需要通过以下几个方面来学习如何实现物理内存页面的分配管理。

1. 划分内核在不同平台的可动态分配的物理地址空间范围。

2. 实现物理页面的 RAII 特性，即生命周期随着页面的申请而分配，随着进程结束而释放。

3. 实现栈式的物理内存分配管理，通过栈的维护来管理空闲的物理页面，实现分配时从栈顶弹出，回收时压栈的效果。

4. 向用户进程提供申请和释放物理页面的接口方法。

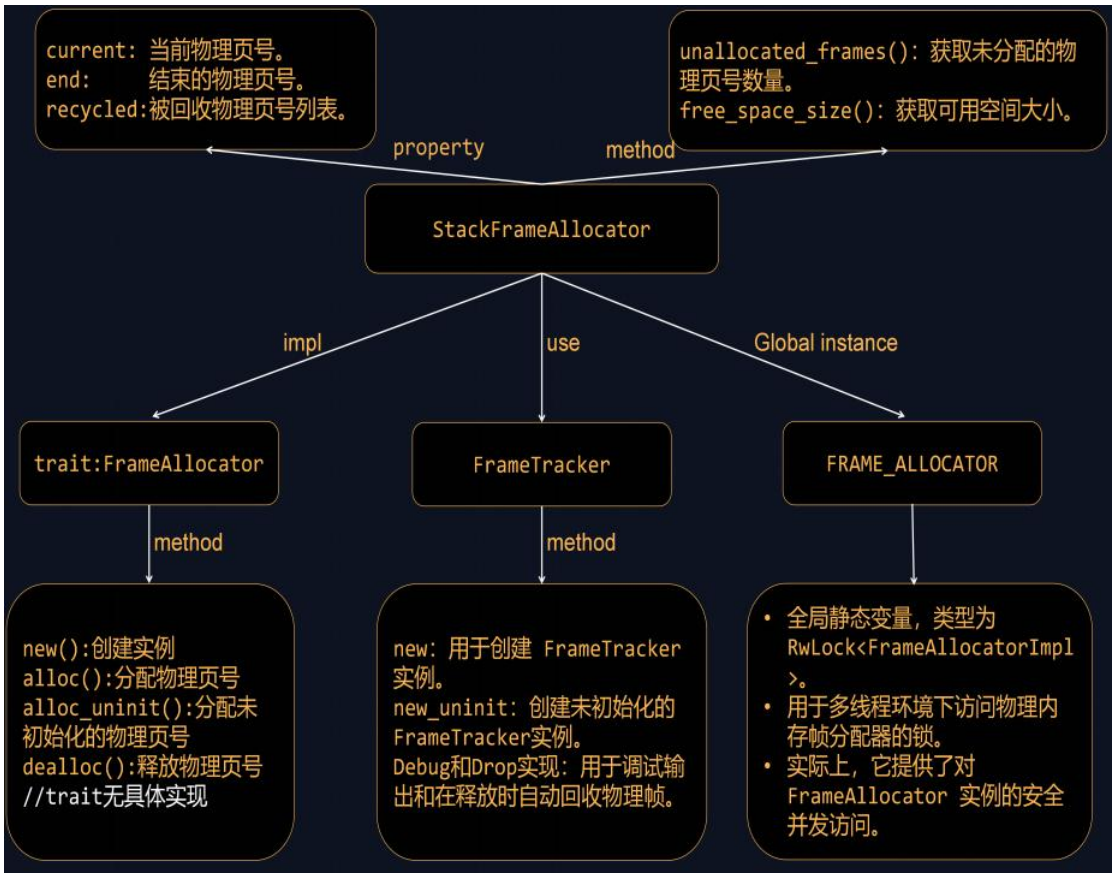


图 3-6NPUcore+ 中物理内存分配涉及的数据结构

3.2.2 虚拟内存管理

内存管理，是指软件运行时对电脑内存资源的分配和使用的技术。其最主要的目的是如何高效、快速的分配，并且在适当的时候释放和回收内存资源。

NPUcore+实现了一种依赖 Page Fault 的优化（CoW）。通过将两个页表项的 W 权限位清零，并且映射到相同的物理页。可以实现页面共享和写时复制（Copy onWrite）。对共享页的写入会产生 Page Fault，此时我们再结合内存描述符以及页表项的情况做出判断和相应处理。

如图 3-6 所示，是我们对相关 Fault 具体的处理方法：

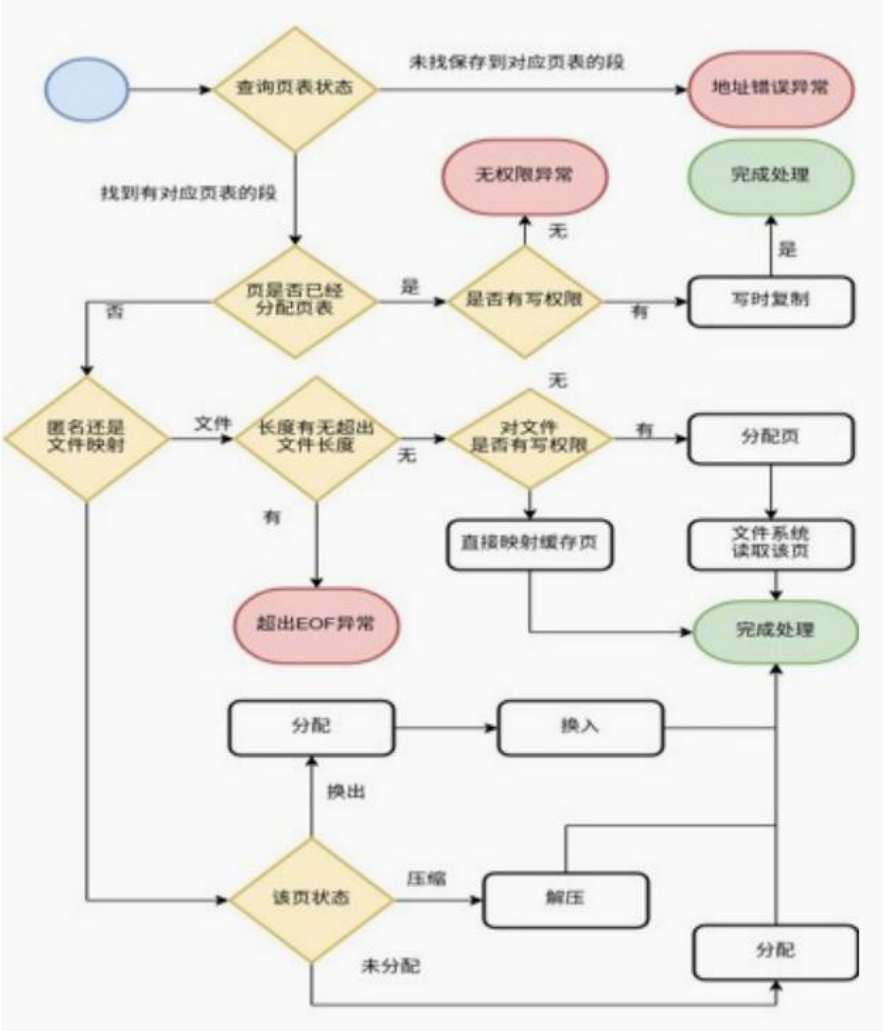


图 3-6 Page Fault 处理方法

其次是 Page Fault 。由于没有权限的情况判断已经由之前的两个路径分支点完成,这里一定是有权限的，只是触发的权限是 Read 还是 Write。进入这个分支的必然是页还没分配的，如果是文件映射，对于 MapArea 中有写入权限的，按照 Linux 的默认行为，是直接写回文件，所以我们直接将页分配给这里,然后读取一页的文件即可。对于文件没有写入权限的，就可以直接映射已经缓存的页，然后完成。之后的其他行为和别的已经映射的没有差异。事实上，Linux 是允许阻断文件映射对源文件的写回，但不是默认行为，我们暂时不考虑。如果是匿名映射，则需要进行判断，如果该页面未被分配，则分配页面；如果是该页面已分配但是已被换出，则重新进行分配，即将该页面重新换入；如果该页面处于压缩内存当中，则将该页面进行解压，得到原来的页面，经过上述三种不同情况的处理，就可完成分配。

NPUcore+ 的“扩容”集成优化方法，效果极其显著。这一套虚拟内存管理方法，让应用程序可以感知到更多的可用内存，我们的实验也验证了 NPUcore+ 内存回收性能的高效。

3.3 文件系统

3.3.1 文件系统概述

文件最早来自于计算机用户需要把数据持久保存在持久存储设备上的需求。由于放在内存中的数据在计算机关机或掉电后就会消失，所以应用程序要把内存中需要保存的数据放到持久存储设备的数据块（比如磁盘的扇区等）中存起来。随着操作系统功能的增强，在操作系统的管理下，应用程序不用理解持久存储设备的硬件细节，而只需对文件这种持久存储数据的抽象进行读写就可以了，由操作系统中的文件系统和存储设备驱动程序一起来完成繁琐的持久存储设备的管理与读写。所以本章要完成的操作系统的第一个核心目标是：让应用能够方便地把数据持久保存起来。

对于应用程序访问持久存储设备的需求，内核需要新增两种文件：常规文件和目录文件，它们均以文件系统所维护的磁盘文件形式被组织并保存在持久存储设备上。

3.3.2 虚拟文件系统及接口

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。

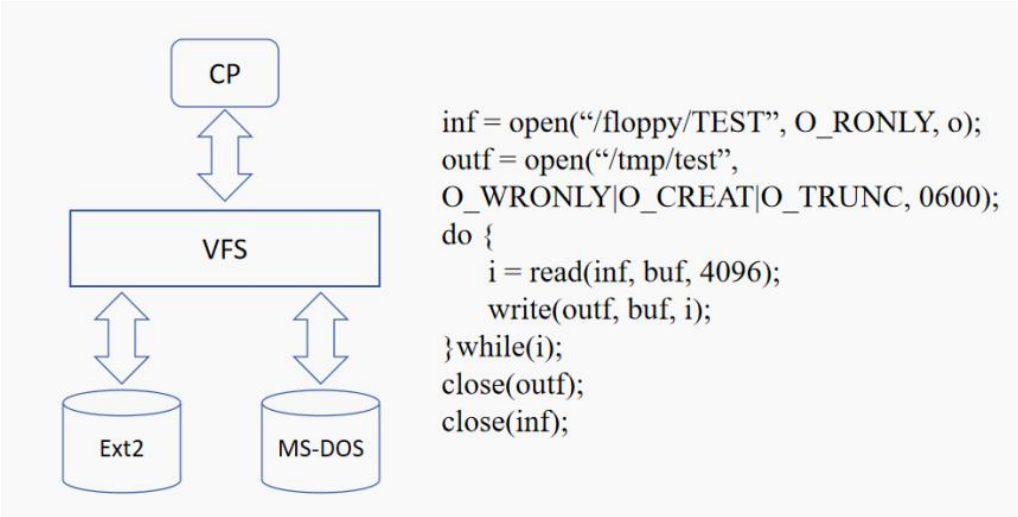


图 3-7VFS 在文件复制

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。

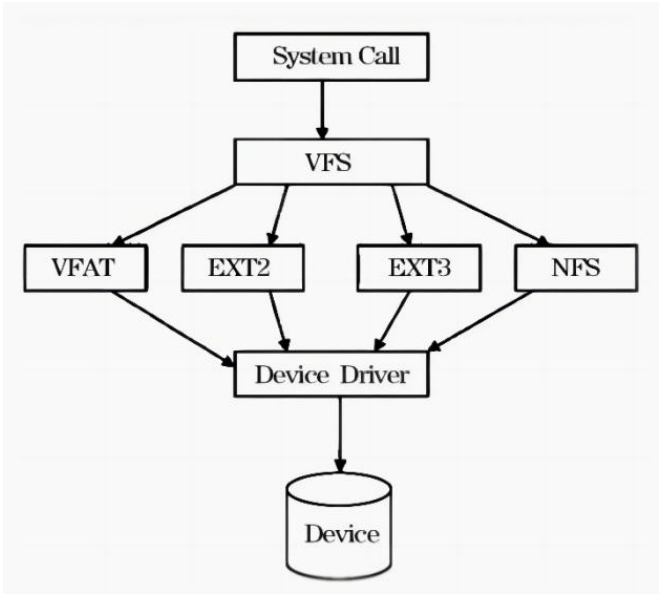


图 3-8VFS 在 OS 结构中

4. 系统调用的设计实现

4.1 系统调用的流程

系统调用就是对操作系统内核中的一组用于实现系统功能的过程的调用。用户程序可以利用系统调用，向操作系统发出服务请求；操作系统通过系统调用为运行于其上的应用程序提供服务。操作系统与运行在用户态软件之间的接口形式就是应用程序二进制接口 (ABI, Application Binary Interface)。操作系统设计了一套安全可靠的二进制接口，我们称为系统调用接口 (System Call Interface)。系统调用接口通常面向应用程序提供了 API 的描述，但在具体实现上，还需要提供 ABI 的接口描述规范。

凡是与资源有关的操作、会影响到其它进程的操作，为了方便管理资源（防止恶意操作）、使进程间隔离，操作系统必须介入，实现统一管理调度。操作系统为上层编程语言提供了一套接口，这套接口就是系统调用。

4.2 一些系统调用的实现

4.2.1sbrk 系统调用

sbrk 系统调用用于改变进程的堆的大小。堆是一段长度可变的连续虚拟内存，始于进程的未初始化数据段末尾，随着内存的分配和释放而增减。通常将堆的当前内存边界称为“program **br**ea**k**”。

sbrk 系统调用的作用是将进程的堆空间扩大或缩小一定的尺寸。要注意不能超过用户空间大小限制，也不能使堆顶指针小于堆底指针。

sbrk 系统调用核心功能的实现由 MemorySet 接口下的 sbrk 函数完成，

```
void *sbrk(intptr_t increment);
```

图 4-1sbrk 系统调用的原型

sbrk 系统调用将堆的大小增加 increment 字节，并返回堆的起始地址。如果 increment 为负数，则堆的大小减少 increment 字节。如果堆的大小超过了进程的地址空间，则 sbrk 系统调用返回-1，并设置 errno 为 ENOMEM。sbrk 系统调用可以为一个进程扩大或者缩小堆的大小，主要的实现是由 os/src/memory_set.rs 中的 sbrk 函数完成。sbrk 函数调用 Memoryset::mmap 或者 Memoryset::munmap 来实现堆的扩大或者缩小。mmap 函数不仅用于 sbrk 系统调用，还用于 mmap 系统调用，用于将文件映射到用户空间和开辟匿名内存映射。

```rust
pub fn sbrk(&mut self, heap_pt: usize, heap_bottom: usize, increment: isize) -> usize {
        // TODO: 在此处填充代码实现 sbrk
        let old_pt = heap_pt;
        let new_pt = (heap_pt as isize + increment) as usize; //修改
        if increment > 0 {
            if new_pt > heap_pt + USER_HEAP_SIZE {
                warn!("increment too big");
                return old_pt;
            }else {
                self.mmap(
                    old_pt,
                    increment as usize,
                    MapPermission::R | MapPermission::W | MapPermission::U,
                    MapFlags::MAP_ANONYMOUS | MapFlags::MAP_FIXED | MapFlags::MAP_PRIVATE,
                    0,
                    0
                );
          //      return new_pt;
            }
        }else if increment <0 {
            if new_pt < heap_bottom{
                warn!("increment too small");
                return old_pt;
            }else {
                self.munmap(old_pt,(-increment) as usize).unwrap(); //修改
               // return new_pt;
            }
        }
        return new_pt;
    }
```

4.2.2mmap 系统调用

首先根据提示实现缺失的两个条件，第一个条件判断区域权限与参数中指定权限相同：area 是对 MapArea 结构体实例的一个引用，可以用来访问和修改 MapArea 实例中的数据，pub struct MapArea { inner: LinearMap,map_type: MapType,map_perm: MapPermission,pub map_file: Option<Arc<dyn File>>,} 可看到其中的 map_prem 字段的类型是 MapPermission，和 mmap 中的 prot 字段的类型相同，所以我们可以写出第一个判断条件为：area.map_perm == prot。

第二个判断条件是判断区域 map_file 为空,由于 map_file 是 Option 类型，所以可以使用 Option 类型的一个方法叫做 is_none()来判断是否为空，即：area.map_file.is_none()。

接下来是实现 mmap 系统调用的 MAP_ANONYMOUS，mmap 匿名映射的作用是在进程空间中创建一段指定大小的虚拟内存区域，该区域不与任何实际文件关联，也不会被写入硬盘。根据上面的提示[mmap] merge with previous area, call expand_to,可知道可以通过调用 expand_to()函数来实现 mmap 的 匿名映射。查看 expand_to 函数：

```rust
//expand_to函数的主要作用是根据传入的新的结束虚拟地址，更新数据结构中的结束虚拟页号，从而实现扩张
pub fn expand_to(&mut self, new_end: VirtAddr) -> Result<(), ()> {
        let new_end_vpn: VirtPageNum = new_end.ceil();
        let old_end_vpn = self.inner.vpn_range.get_end();
        if new_end_vpn < old_end_vpn {
            warn!(
                "[expand_to] new_end_vpn: {:?} is lower than old_end_vpn: {:?}",
                new_end_vpn, old_end_vpn
            );
            return Err(());
        }
        // `set_end` must be done before calling `map_one`
        // because `map_one` will insert frames into `data_frames`
        // if we don't `set_end` in advance, this insertion is out of bound
        self.inner.set_end(new_end_vpn)?;
        Ok(())
    }
```

可知道 expand_to 函数函数用于将数据结构的内存范围扩展到给定的虚拟地址，确保新范围不小于旧范围。所以我们只需要将 mmap 中的扩展后的地址传入 expand_to 函数中即可，首先需要计算出新的扩展地址，首先需要获得当前区域的结束虚拟地址，可以借鉴 expand_to 函数中 self.inner.vpn_range.get_end（）的写法来获取当前区域的结束虚拟页号，然后在使用 into()转换成虚拟地址，即 let end_va : VirtAddr= area.inner.vpn_range.get_end().into(); 接着可通过 VirtAddr::from(end_va.0 + len))得到扩展后的虚拟内存地址，注意这里由于 end_va 是 VirtAddr 类型，len 是 usize 类型，这里没有实现 VirtAddr 和 VirtAddr 相加的 trait，所以只能通过 usize+usize 相加然后在转换成 VirtAddr 实现，这里 end_va.0 就已经是 usize 类型的了，• pub struct VirtAddr(pub usize);由于 VirtAddr 结构体定义中的唯一字段是 usize 类型的公共字段 pub usize，因此在这里 .0 表示访问该结构体实例中的 usize 类型字段。接着通过 area 调用 expand_to 函数，即：area.expand_to(VirtAddr::from(end_va.0 + len)).unwrap(); 因为 expand_to 函数返回类型是 Result<(), ()> 这里通过调用 unwrap 来防止 expand_to 函数中的新的结束地址小于旧的结束地址的错误。由于 mmap 的返回类型是 isize，我们要在调用 expand_to 函数完成扩展后，返回旧的结束地址，即 return end_va.0 as isize。

```rust
    pub fn mmap(
        &mut self,
        start: usize,    //要映射的地址空间区域的开始地址
        len: usize,      //该区域的大小
        prot: MapPermission, //该区域的访问权限
        flags: MapFlags, // 映射的方式 (如读写权限等)
        fd: usize,       //要映射的文件的句柄 (文件描述符)
        offset: usize,   //映射的起始偏移量
    ) -> isize {
        // not aligned on a page boundary
        if start & 0xfff != 0 {
            return EINVAL;
        }
        let len = if len == 0 { PAGE_SIZE } else { len };
        let task = current_task().unwrap();
        let idx = self.last_mmap_area_idx();
        let start_va: VirtAddr = if flags.contains(MapFlags::MAP_FIXED) {
            // unmap if exists
            self.munmap(start, len);
            start.into()
        } else {
            if let Some(idx) = idx {
                let area = &mut self.areas[idx];
                if flags.contains(MapFlags::MAP_PRIVATE | MapFlags::MAP_ANONYMOUS)
                    && area.map_perm == prot // 增加条件，判断区域权限与参数中指定权限相同
                    && area.map_file.is_none()// 增加条件，判断区域 map_file 为空
                {
                    debug!("[mmap] merge with previous area, call expand_to");
                    // ************************************************
                    // TODO: 在此处填充代码，实现 mmap 系统调用的 MAP_ANONYMOUS
                    let end_va: VirtAddr = area.inner.vpn_range.get_end().into();
                    area.expand_to(VirtAddr::from(end_va.0 + len)).unwrap();
                    return end_va.0 as isize;
                    // ************************************************
                }
                area.inner.vpn_range.get_end().into()
            } else {
                MMAP_BASE.into()
            }
        };
        let mut new_area = MapArea::new(
            start_va,
            VirtAddr::from(start_va.0 + len),
            MapType::Framed,
            prot,
            None,
        );
        if !flags.contains(MapFlags::MAP_ANONYMOUS) {
            warn!("[mmap] file-backed map!");
            let fd_table = task.files.lock();
            match fd_table.get_ref(fd) {
                Ok(file_descriptor) => {
                    if !file_descriptor.readable() {
                        return EACCES;
                    }
                    let file = file_descriptor.file.deep_clone();
                    file.lseek(offset as isize, SeekWhence::SEEK_SET).unwrap();
                    new_area.map_file = Some(file);
                }
                Err(errno) => return errno,
            }
        }
        // insert MapArea and keep the order
        let (idx, _) = self
            .areas
            .iter()
            .enumerate()
            .skip_while(|(_, area)| {
                area.inner.vpn_range.get_start() >= VirtAddr::from(MMAP_END).into()
            })
            .find(|(_, area)| area.inner.vpn_range.get_start() >= start_va.into())
            .unwrap();
        self.areas.insert(idx, new_area);
        start_va.0 as isize
    }
```

4.2.3 打印 TCB 信息

本地打印 TCB 信息：

首先在/oskernel2024-npucore/user/src/bin 目录下创建 printf_tcb.rs 文件：

```rust
///oskernel2022-npucore/user/src/bin/printf_tcb.rs
#![no_std]
#![no_main]

#[macro_use]
extern crate user_lib;
use user_lib::print_tcb;

#[no_mangle]
pub fn main() -> i32 {
    let mut message = [0 as usize; 6];
    let mut message_ptr = &mut message as *mut usize;  //指向message地址的可变指针
    print_tcb(message_ptr);
    println!("pid of the TCB is: {} ",message[0]);
    println!("tid of the TCB is: {} ",message[1]);
    println!("tgid of the TCB is: {} ",message[2]);
    println!("kernel stack of the TCB is: {} ",message[3]);
    println!("user stack of the TCB is: {} ",message[4]);
    println!("trap context physics page number of the TCB is: {} ",message[5]);
    0
}
```

在 /oskernel2024-npucore/user/src 目录下修改 usr_call.rs 和 syscall.rs

```
//usr_call.rs添加下面代码
pub fn print_tcb(message: *mut usize) -> isize {
    sys_printtcb(message)
}
```

```
//syscall.rs  添加下面代码
const SYSCALL_PRINT_TCB: usize =1691;

pub fn sys_printtcb(message: *mut usize) -> isize {
    syscall(SYSCALL_PRINT_TCB, [message as usize, 0, 0])
}
```

在 /oskernel2024-npucore/os/src/syscall 目录下修改 mod.rs 和 process.rs

```
//mod.rs 添加下面代码
const SYSCALL_PRINT_TCB: usize =1691;

pub fn syscall_name(id: usize) -> &'static str {
    match id {
        SYSCALL_PRINT_TCB => "print_tcb",
    }
}

SYSCALL_PRINT_TCB => sys_printtcb(args[0] as *mut usize),



//process.rs 添加下面代码

pub fn sys_printtcb(message: *mut usize) -> isize {
    let mut tcb_massage = [0 as usize ; 6];
    let tcb = current_task().unwrap();
    let token = tcb.get_user_token();
    tcb_massage[0] = tcb.pid.0;
    tcb_massage[1] = tcb.tid;
    tcb_massage[2] = tcb.tgid;
    tcb_massage[3] = tcb.kstack.0;
    tcb_massage[4] = tcb.ustack_base;
    tcb_massage[5] = tcb.acquire_inner_lock().trap_cx_ppn.0;
    copy_to_user(token, &tcb_massage, message as *mut [usize; 6]);
    0
}
```

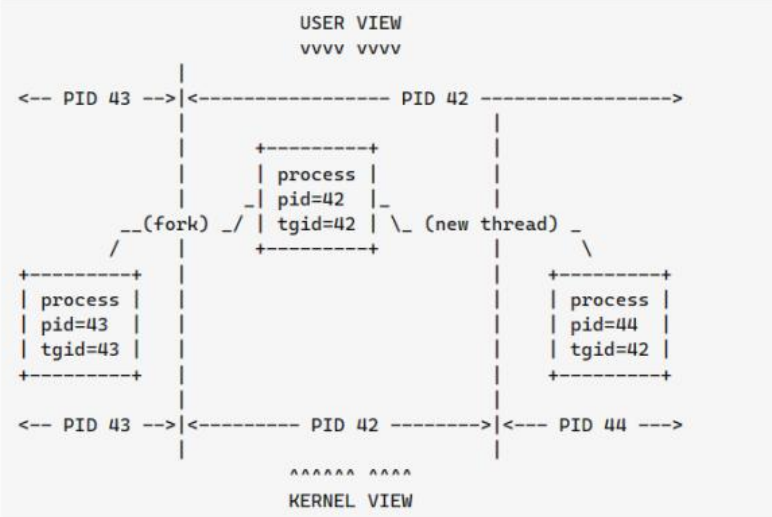
在/oskernel2024-npucore/user 目录下执行：make、make rust-user 编译新编写的 rust 代码，生成可执行文件，接着在/oskernel2024-npucore/os 目录下执行:make fat32，为 NPUcore+重新加载系统镜像。

os 目录下执行 make run，键入 print_tcb 结果如下：

```
NPUCore:/# print_tcb
pid of the TCB is: 2
tid of the TCB is: 0
tgid of the TCB is: 2
kernel stack of the TCB is: 2
user stack of the TCB is: 3221221376
trap context physics page number of the TCB is: 526070
NPUCore:/# print_tcb
pid of the TCB is: 2
tid of the TCB is: 0
tgid of the TCB is: 2
kernel stack of the TCB is: 2
user stack of the TCB is: 3221221376
trap context physics page number of the TCB is: 526144
```

4.2.4 实现 sys_getpid 系统调用

sys_getpid 系统调用的作用是获取自己的进程 ID，所以我们只需要获取到当前的进程，然后将当前进程的 id 返回即可，且由上图 4.1 print_tcb 实验结果可看到 pid 和 tgid 是一样的，一个进程就是线程组，这是因为 NPUcore 在进程管理这一章节只考虑的单线程的情况。




```rust
pub fn sys_getpid() -> isize {
    let pid = current_task().unwrap().tgid;
    pid as isize
}
```

下面编写 getpid.rs 来验证 sys_getpid()系统调用是否正确：

```rust
///oskernel2022-npucore/user/src/bin/getpid.rs
#![no_std]
#![no_main]

#[macro_use]
extern crate user_lib;
use user_lib::getpid;

#[no_mangle]
pub fn main() -> i32 {

    println!(" The current process pid is:{}",getpid());
    0
}
```

在/oskernel2024-npucore/user 目录下执行：make、make rust-user 编译新编写的 rust 代码，生成可执行文件，接着在/oskernel2024-npucore/os 目录下执行:make fat32，为 NPUcore+重新加载系统镜像。

os 目录下执行 make run，键入 getpid 后结果如下：

```
NPUCore:/# getpid
 The current process pid is:2
NPUCore:/#
```

4.2.5 实现 getrusage 系统调用

当用户进程发起 getrusage 系统调用时，内核会收集并记录进程的资源利用情况，同时内核会将相应的资源使用信息传递给用户进程，使用户能够访问和使用这些数据。who：若为 0，表示获取当前进程的信息；若为-1，表示获得子进程的信息（可不实现）。

所以实现 getrusage 系统调用需要先获取到当前进程的信息，然后通过 copy_to_user()函数写入用户空间。

```rust
pub fn sys_getrusage(who: isize, usage: *mut Rusage) -> isize {
    if who != 0 {
        panic!("[sys_getrusage] parameter 'who' is not RUSAGE_SELF.");
    }
    let task = current_task().unwrap();
    let inner = task.acquire_inner_lock();
    let token = task.get_user_token(); //物理页的标识符（token）
    copy_to_user(token, &inner.rusage, usage);
    // info!("[sys_getrusage] who: RUSAGE_SELF, usage: {:?}", inner.rusage);
    SUCCESS
}
```

测试 sys-getrusage 系统调用是否正确：

```rust
///oskernel2022-npucore/user/src/bin/getrusage_test.rs
#![no_std]
#![no_main]

#[macro_use]
extern crate user_lib;
use user_lib::getrusage;

#[no_mangle]
pub fn main() -> i32 {
    let mut rusage = [0 as usize; 18];
    let mut rusage_ptr = &mut rusage as *mut usize;  //指向rusage地址的可变指针
    getrusage(0, rusage_ptr);   //通过rusage_ptr提供的rusage的地址，将内核态获取到的信息写入rusage
    unsafe{
        let utime_sec = *(rusage_ptr.offset(0));
        let utime_nsec= *(rusage_ptr.offset(1));
        let stime_sec = *(rusage_ptr.offset(2));
        let stime_nsec= *(rusage_ptr.offset(3));
        println!("user cpu time:{}ns",utime_sec * 1000000 + utime_nsec);
        println!("system cpu time:{}ns",stime_sec * 1000000 + stime_nsec);
    }
    0
}
```

在 /oskernel2024-npucore/user/src 目录下修改 usr_call.rs 和 syscall.rs

```rust
//usr_call.rs 添加代码
pub fn getrusage(who: isize, rusage:*mut usize) -> isize {
    sys_getrusage(who, rusage)
}
```

```rust
//syscall.rs 添加代码
pub fn sys_getrusage(who: isize, rusage:*mut usize) -> isize {
    syscall(SYSCALL_GETRUSAGE, [who as usize, rusage as usize, 0])
}
```

在/oskernel2024-npucore/user 目录下执行：make、make rust-user 编译新编写的 rust 代码，生成可执行文件，接着在/oskernel2024-npucore/os 目录下执行:make fat32，为 NPUcore+重新加载系统镜像。

os 目录下执行 make run，键入 getrusage_test 结果如下：

```
NPUCore:/# getrusage_test
user cpu time:3432ns
system cpu time:14230ns
```

4.2.6 实现 fork+exec 系统调用

与清华大学开发的 rCore 教学操作系统不同，在 NPUcore+中并没有特意实现 fork 系统调用，fork 就是特定参数下的 clone 系统调用。

```rust
// user/src/syscall.rs
pub fn sys_fork() -> isize {
    const SIGCHLD: usize = 17;
    syscall(SYSCALL_CLONE, [SIGCHLD, 0, 0])
}
```

所以我们只需要在 fork 中调用 sys_clone，并传入相关的参数即可。

sys_clone 系统调用的传参如下：

```rust
pub fn sys_clone(  //真正的sys_clone
    self: &Arc<TaskControlBlock>,
    flags: CloneFlags,
    stack: *const u8,
    tls: usize,   //线程本地存储
    exit_signal: Signals,
) -> Arc<TaskControlBlock> {
```

下面开始编写 sys_fork()：

```rust
fn sys_fork() -> isize {
    let flags = CloneFlags::from_bits(0).unwarp(); //from_bits(0)可能会报panic
    let stack = 0 as *const u8;
    let tls = 0 as usize;
    let parent = current_task().unwrap();
    let exit_signal = Signals::from_bits_truncate(1 << 16); //const SIGCHLD= 1 << (16);
    show_frame_consumption! {
        "clone";
        let child = parent.sys_clone(flags, stack, tls, exit_signal);
    }
    let new_pid = child.pid.0;
    add_task(child);
    new_pid as isize
}
```

测试 fork_test 系统调用是否正确：

```rust
///oskernel2022-npucore/user/src/bin/fork_test.rs
#![no_std]
#![no_main]

#[macro_use]
extern crate user_lib;
use user_lib::{exit, fork, wait};
const MAX_CHILD: usize = 30;

#[no_mangle]
pub fn main() -> i32 {
    for i in 0..MAX_CHILD {
    let pid = fork();
        if pid == 0 {
        println!("I am child {}", i);
        exit(0);
        } else {
            println!("forked child pid = {}", pid);
        }
            assert!(pid > 0);
    }
    let mut exit_code: i32 = 0;
    for _ in 0..MAX_CHILD {
        if wait(&mut exit_code) <= 0 {
            panic!("wait stopped early");
        }
    }
    if wait(&mut exit_code) > 0 {
        panic!("wait got too many");
    }
    println!("forktest pass.");
    0
}
```

在/oskernel2024-npucore/user 目录下执行：make、make rust-user 编译新编写的 rust 代码，生成可执行文件，接着在/oskernel2024-npucore/os 目录下执行:make fat32，为 NPUcore+重新加载系统镜像。

os 目录下执行 make run，键入 fork_test 后结果如下：

```
NPUCore:/# fork_test
forked child pid = 3
forked child pid = 4
forked child pid = 5
forked child pid = 6
forked child pid = 7
forked child pid = 8
forked child pid = 9
forked child pid = 10
forked child pid = 11
forked child pid = 12
forked child pid = 13
forked child pid = 14
forked child pid = 15
forked child pid = 16
forked child pid = I am child 0
I am child 1
I am child 2
I am child 3
I am child 4
I am child 5
I am child 6
I am child 7
I am child 8
I am child 9
I am child 10
I am child 11
I am child 12
I am child 13
I am child 14
17
forked child pid = 18
forked child pid = 19
forked child pid = 20
forked child pid = 21
forked child pid = 22
forked child pid = 23
forked child pid = 24
forked child pid = 25
forked child pid = 26
forked child pid = 27
forked child pid = 28
forked child pid = 29
forked child pid = 30
forked child pid = 31
forked child pid = 32
I am child 15
I am child 16
I am child 17
I am child 18
I am child 19
I am child 20
```

接下来是 exec 系统调用的实现：

sys_execve 系统调用是用于执行新程序的操作。它将一个新程序加载到当前进程的地址空间并执行它。

它接受三个参数：pathname：指向可执行文件名的用户空间指针

argv：参数列表，指向用户空间的参数列表起始地址

envp：环境变量表，环境变量是一系列键值对，字符串类型

当调用 execve 时，操作系统将当前进程替换为指定路径的新程序，并将其参数传递给新程序。这个调用是通过加载新的程序映像来实现的，原来的进程的代码、数据和堆栈等信息都被新的程序所取代，执行权也被移交给新程序。

4.2.7 实现 sys_clone 系统调用

```rust
pub fn sys_clone(
    flags: u32,
    stack: *const u8,
    ptid: *mut u32,
    tls: usize,
    ctid: *mut u32,
) -> isize {}
```

sys_clone 系统调用一共有五个参数：
flags：用于配置新任务的不同属性。前 24 位记录 CloneFlags 信息，后 8 位记录 Signals 信息。
stack：这是一个指向用户空间中新任务堆栈的指针。如果这是 NULL，则子任务将共享父任务的堆栈。
ptid 和 ctid：用于在创建子任务时存储父进程和子进程的线程 ID。
tls：无符号整数，代表新任务的线程本地存储（TLS）地址。TLS 是一种机制，用于在线程内存储线程特定的数据。它通常用于存储线程私有的数据。
clone 系统调用的核心在于调用 task 模块中的 sys_clone 方法，下面我们看下 task 模块中的 sys_clone 的参数：

```rust
//task模块中的sys_clone
pub fn sys_clone(  //真正的sys_clone
        self: &Arc<TaskControlBlock>,
        flags: CloneFlags,
        stack: *const u8,
        tls: usize,    //线程本地存储
        exit_signal: Signals,
    ) -> Arc<TaskControlBlock> {}
```

下面我们只需要传递正确的参数，就可以通过 sys_clone 实现 clone 系统调用：

```rust
pub fn sys_clone(   //封装后的sys_clone
    flags: u32,
    stack: *const u8,
    ptid: *mut u32,
    tls: usize,   //TLS:代表新任务的线程本地存储（TLS）地址
    ctid: *mut u32,
) -> isize {
    let parent = current_task().unwrap();
    // This signal will be sent to its parent when it exits
    // we need to add a field in TCB to support this feature, but not now.
    let exit_signal = match Signals::from_signum((flags & 0xff) as usize) {//通过掩码将flags的后八位转化为Signals类型的exit_signal
        Ok(signal) => signal,
        Err(_) => {
            warn!(
                "[sys_clone] signum of exit_signal is unspecified or invalid: {}",
                (flags & 0xff) as usize
            );
            // This is permitted by standard, but we only support 64 signals
            Signals::empty()
        }
    };
    // Sure to succeed, because all bits are valid   (See `CloneFlags`)
    let flags = CloneFlags::from_bits(flags & !0xff).unwrap();//通过掩码将flags的前24位转化为CloneFlags类型的参数flags
    info!(
        "[sys_clone] flags: {:?}, stack: {:?}, exit_signal: {:?}, ptid: {:?}, tls: {:?}, ctid: {:?}",
        flags, stack, exit_signal, ptid, tls, ctid
    );
    show_frame_consumption! {
        "clone";
        let child = parent.sys_clone(flags, stack, tls, exit_signal); //调用task模块中的sys_clone实现clone
    }

    let new_pid = child.pid.0;

    //根据不同的标志位对相应的内存进行修改
    if flags.contains(CloneFlags::CLONE_PARENT_SETTID) {    //表示子线程的线程标识符将被存储在父进程内存中由 ptid 指向的位置
        match translated_refmut(parent.get_user_token(), ptid) {
            Ok(word) => *word = child.pid.0 as u32,
            Err(errno) => return errno,
        };
    }
    if flags.contains(CloneFlags::CLONE_CHILD_SETTID) {     //表示子线程的线程标识符将被存储在子进程内存中由 ctid 指向的位置
        match translated_refmut(child.get_user_token(), ctid) {
            Ok(word) => *word = child.pid.0 as u32,
            Err(errno) => return errno,
        };
    }
    if flags.contains(CloneFlags::CLONE_CHILD_CLEARTID) {   //在子进程终止时将ctid所指向的内存清零
        child.acquire_inner_lock().clear_child_tid = ctid as usize;
    }
    // add new task to scheduler
    add_task(child);
    new_pid as isize
}
```

由于我们在前面实现 fork 系统调用时已经验证了 sys_clone 的正确性，在此不再进行验证。

4.2.8 实现 mmap 系统调用扩展

```rust
//os/src/mm/memory_set.rs
pub fn mmap(
        &mut self,
        start: usize,
        len: usize,
        prot: MapPermission,
        flags: MapFlags,
        fd: usize,
        offset: usize,
    ) -> isize {
  /**************
  在之前的关卡中已经实现了匿名映射，因此
  只需要在此题中if !flags.contains(MapFlags::MAP_ANONYMOUS)块内，补全非匿名映射，即私有映射时的情况。
  ***************/

//MAP_PRIVATE这个标志最常用于在进程中只读地映射一个文件，并且不想把进程的修改写回到文件中
        //下面是自己实现的内容
        // 以下提供一个实现的大致思路：
        // 通过传入的文件描述符克隆一个文件对象；
        // 使用lseek方法移动文件指针到指定位置；
        // 判断访问权限，文件是否可读；
        // 将文件对象保存到映射区域中，表示这个文件是映射区域的内容来源。
        if !flags.contains(MapFlags::MAP_ANONYMOUS) {
            let fd_table = task.files.lock();
            let file_descriptor = match fd_table.get_ref(fd){
                Ok(file_descriptor) => file_descriptor.clone(),
                Err(errno) => return errno,
            };
            let file = file_descriptor.file.deep_clone();
            file.lseek(offset as isize, SeekWhence::SEEK_SET).unwrap();
            if !file.readable(){
                return EACCES;
            }
            new_area.map_file = Some(file);
        }
}
```

在/oskernel2024-npucore/user 目录下执行：make 编译新编写的 rust 代码，生成可执行文件，接着在/oskernel2024-npucore/os 目录下执行:make fat32，为 NPUcore+重新加载系统镜像。os 目录下执行 make run。

4.2.9 实现 open 系统调用

```rust
//os/src/syscall/fs.rs
//自己实现的sys_openat
//sys_openat接收四个参数，分别为目录描述符（文件所在目录的文件描述符）、路径、打开标志和文件权限模式。
//函数需要按照flags指定的打开模式来查找path指定的文件，并以相应权限打开，将文件描述符插入到请求的task中，若失败需要返回相应错误码。
pub fn sys_openat(dirfd: usize, path: *const u8, flags: u32, mode: u32) -> isize {
    let task = current_task().unwrap();
    let token = task.get_user_token();
    let mut fd_table = task.files.lock();
    let file_descriptor = match dirfd {    //根据dirfd获取文件描述符
        AT_FDCWD => task.fs.lock().working_inode.as_ref().clone(),
        fd => {
            match fd_table.get_ref(fd) {
                Ok(file_descriptor) => file_descriptor.clone(),
                Err(errno) => return errno,
            }
        }
    };
    let flags = match OpenFlags::from_bits(flags) {
        Some(flags) => flags,
        None => {
            warn!("[sys_openat] unknown flags");
            return EINVAL;
        },
    };
    let path = translated_str(token, path).unwrap();
    //open系统调用是一种常用的系统调用，它用于打开文件并返回一个文件描述符，应用程序可以使用该文件描述符执行各种操作，如读取、写入、关闭文件等。
    let new_file_descriptor =match file_descriptor.open(&path, flags, false) {
            Ok(new_file_descriptor) => new_file_descriptor,
            Err(errno) =>return errno,
    };
    let new_fd = match fd_table.insert(new_file_descriptor){
        Ok(new_fd) => new_fd,
        Err(errno) =>return errno,
    };
    new_fd as isize
}
```

编写用户态测试程序

```rust
//user/src/bin/file_test.rs
#![no_std]
#![no_main]

#[macro_use]
extern crate user_lib;

use user_lib::{
    open,
    close,
    read,
    write,
    OpenFlags,
};

#[no_mangle]
pub fn main() -> i32 {
    let test_str = "Hello, world!";
    let filea = "filea\0";
    let fd = open(filea, OpenFlags::CREATE | OpenFlags::WRONLY);
    assert!(fd > 0);
    let fd = fd as usize;
    write(fd, test_str.as_bytes());
    close(fd);

    let fd = open(filea, OpenFlags::RDONLY);
    assert!(fd > 0);
    let fd = fd as usize;
    let mut buffer = [0u8; 100];
    let read_len = read(fd, &mut buffer) as usize;
    close(fd);

    assert_eq!(
        test_str,
        core::str::from_utf8(&buffer[..read_len]).unwrap(),
    );
    println!("file_test passed!");
    0
}
```

在/oskernel2024-npucore/user 目录下执行：make 编译新编写的 rust 代

码，生成可执行文件，接着在/oskernel2024-npucore/os 目录下执行:make fat32，为 NPUcore+重新加载系统镜像。os 目录下执行 make run，进入 NPUcore+后输入：file_test，出现下面报错信息：

```
NPUCore:/# file_test
Panicked at src/bin/file_test.rs:20, assertion failed: fd > 0
```

开启 LOG 结合 GDB 进行调试 os 目录下执行：make run LOG=info

发现和我在 user 目录下编写的测试程序传入的 flags 不一致，测试程序传入的 flags 是 OpenFlags::CREATE | OpenFlags::WRONLY，在打开文件时 CREATE 标志使得如果 filea 原本不存在，文件系统会自动创建一个同名文件，如果已经存在的话则会清空它的内容。而 WRONLY 使得此次只能写入该文件而不能读取。但是而内核这里接收到的却是 O_WRONLY | O_TRUNC，查看内核 OpenFlags（os/src/layout.rs）的定义发现：*const* O_CREAT = 0o100;而在 user 目录下(/user/src/lib.rs)发现:*const* CREATE = 1 << 9;在这里发现两者定义不匹配，0o100（八进制）为 0b1000000（二进制） 为 1<<7，而不是 1<<9。

```rust
//在user/src/lib.rs下将OpenFlags修改成与内核下的一致
bitflags! {
    pub struct OpenFlags: u32 {
        const RDONLY = 0o0;
        const WRONLY = 0o1;
        const RDWR   = 0o2;
        const CREATE = 0o100;  //0b1000000 == 1<<7
        const TRUNC  = 0o1000;
    }
}
```

5. 系统测试

5.1 测试结果

测试结果		
测试样例名	通过测试点	全部测试点
test_sleep	2	2
test_mkdir	3	3
test_pipe	0	4
test_dup2	0	2
test_openat	4	4
test_getdents	5	5
test_clone	4	4
test_wait	4	4
test_times	6	6
test_exit	2	2
test_getcwd	2	2
test_read	3	3

测试结果		
测试样例名	通过测试点	全部测试点
test_umount	5	5
test_unlink	2	2
test_uname	2	2
test_waitpid	4	4
test_mmap	0	3
test_fork	3	3
test_dup	2	2
test_fstat	3	3
test_chdir	3	3
test_yield	4	4
test_execve	3	3
test_gettimeofday	3	3
test_munmap	0	4
test_getppid	2	2
test_close	2	2
test_getpid	3	3
test_open	3	3
test_brk	3	3
test_write	2	2
test_mount	5	5

表 5-1 测试结果

6.总结与展望

6.1 初赛测评

比赛提交到排行榜更新有20秒左右的延迟

#	用户名	队伍	提交次数(ASC)	最后提交时间(ASC)	rank
1	T202410699992496	NPUcore-IMPACT!!!/ 西北工业大学	45	2024-05-03 23:34:44	102.0000
2	T202410699992491	NPUcore-重生之我是菜狗/ 西北工业大学	56	2024-03-07 15:27:46	102.0000
3	T202410460992502	NPUcore-重生之我是秦始皇/ 河南理工大学	41	2024-05-17 09:42:54	89.0000
4	T202410614992892	HelloWorld/ 电子科技大学	1	2024-04-23 12:28:03	53.0000
5	T202410213992605	Refill/ 哈尔滨工业大学	8	2024-04-02 19:13:24	53.0000
6	T202412802992528	菜鸡不会riscv/ 吉利学院	2	2024-05-04 01:23:42	0.0000
7	T202410486992576	俺争取不掉队/ 武汉大学	6	2024-05-22 19:48:19	0.0000
8	T202411664992499	重启之我是loader/ 西安邮电大学	20	2024-05-22 15:19:54	0.0000

图 6-1 初赛测评排行榜

6.2 测试点通过情况

测试结果		
测试样例名	通过测试点	全部测试点
test_sleep	2	2
test_mkdir	3	3
test_pipe	0	4
test_dup2	0	2
test_openat	4	4
test_getdents	5	5
test_clone	4	4
test_wait	4	4
test_times	6	6
test_exit	2	2
test_getcwd	2	2
test_read	3	3
test_umount	5	5
test_unlink	2	2
test_uname	2	2
test_waitpid	4	4
test_mmap	0	3
test_fork	3	3
test_dup	2	2
test_fstat	3	3
test_chdir	3	3
test_yield	4	4
test_execve	3	3
test_gettimeofday	3	3
test_munmap	0	4
test_getppid	2	2
test_close	2	2
test_getpid	3	3
test_open	3	3
test_brk	3	3
test_write	2	2
test_mount	5	5

表 6-1 测试结果

6.3 未来展望

我们期望能够在内部设计文档之外，进一步全面通过所设置的测试点，设计教学向一步一步构建的操作系统实现文档，以及将对应的代码构建成适合实验操作的代码，并且按照阶段提供不同的代码框架。