

# NPUcore 操作系统内核构建实践 基础篇



## 目 录



# 第1章 操作系统内核构建概述

## 1.1 什么是操作系统

### 1.1.1 体系结构

如果我们站在一万米的高空来看操作系统，可以发现操作系统这个系统软件干的事主要有两件：一是向下管理并控制计算机硬件和各种外设，二是向上管理应用软件并提供各种服务。我们可对其进一步定义为：操作系统是一种系统软件，主要功能是向下管理CPU、内存和各种外设等硬件资源，并形成软件执行环境来向上管理和服务应用软件。这样的描述也符合大多数操作系统教材上对操作系统的定义。为了完成这些工作，操作系统需要知道如何与硬件打交道，如何给应用软件提供服务。这就有一系列与操作系统相关的理论、抽象、设计等来支持如何做和做得好这两件事情。如果看看我们的身

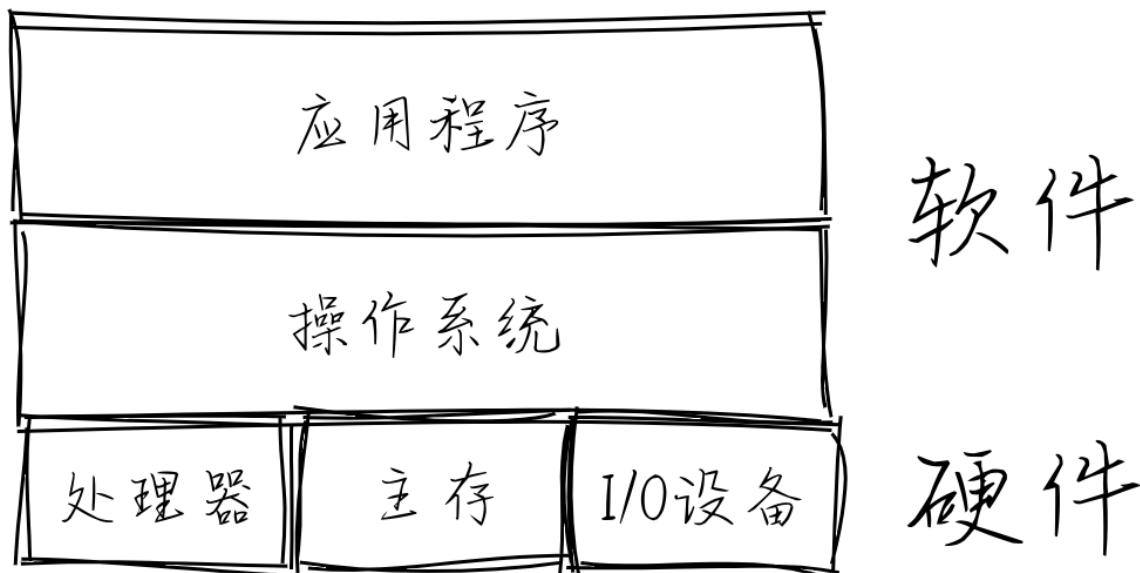


图 1-1 computer-hw-sw

边，Android 应用运行在 ARM 处理器上的 Android 操作系统执行环境中；微软的 Office 应用运行在 x86-64 处理器上的 Windows 操作系统执行环境中；Web Server 应用运行在 x86-64 处理器上的 Linux 操作系统执行环境中；Web app 应用运行在 x86-64 或 ARM 处理器上的 Chrome OS 操作系统执行环境中。而在一些嵌入式环境中，操作系统以运行时库的形式与应用程序紧密结合在一起，形成一个可在嵌入式硬件上执行的嵌入式应用。所以，在不同的应用场景下，操作系统的边界也是不同的，我们可以把运行时库、图形界面支持库等这些可支持不同应用的系统软件 (System Software) 也看成是操作系统的一部分。

那操作系统的组成部分包含哪些内容呢？在一般情况下，操作系统的主要组成包括：

**1、操作系统内核：**操作系统的核心部分，负责控制计算机的硬件资源并为用户和应用程序提供服务。

**2、系统工具和软件库：**为操作系统提供基本功能的软件，包括工具软件和系统软件库等。

**3、用户接口：**是操作系统的外壳，是用户与操作系统交互的方式。用户接口包括图形用户界面（GUI）和命令行界面（CLI）等。

而本书重点讲述的对象是操作系统内核，它的主要组成部分包括：

**进程/线程管理：**内核负责管理系统中的进程或线程，创建、销毁、调度和切换进程或线程。

**内存管理：**内核负责管理系统的内存，分配和回收内存空间，并保证进程之间的内存隔离。

**文件系统：**内核提供文件系统接口，负责管理存储设备上的文件和目录，并允许应用访问文件系统。

**网络通信：**内核提供网络通信接口，负责管理网络连接并允许应用进行网络通信。

**设备驱动：**内核提供设备驱动接口，负责管理硬件设备并允许应用和内核其他部分访问设备。

**同步互斥：**内核负责协调多个进程或线程之间对共享资源的访问。同步功能主要用于解决进程或线程之间的协作问题，互斥功能主要用于解决进程或线程之间的竞争问题。

**系统调用接口：**内核提供给应用程序访问系统服务的入口，应用程序通过系统调用接口调用操作系统提供的服务，如文件系统、网络通信、进程管理等。

### 1.1.2 系统调用

**什么是系统调用**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。**API 与 ABI**站在使用操作系统的角度会比较容易对操作系统内核的功能产生初步的认识。操作系统内核是一个提供各种服务的软件，其服务对象是应用程序，而用户（这里可以理解为一般使用计算机的人）是通过应用程序的服务间接获得操作系统的服务的，因此操作系统内核藏在一般用户看不到的地方。但应用程序需要访问操作系统获得操作系统的服务，这就需要通过操作系统的接口才能完成。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）。

操作系统不能只提供面向单一编程语言的函数库的编程接口（API, Application Programming Interface），它的接口需要考虑对基于各种编程语言的应用支持，以及访问安全等因素，使得应用软件不能像访问函数库一样的直接访问操作系统内部函数，更不能直接读写操作系统内部的地址空间。为此，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。

在现代处理器的安全支持（特权级隔离，内存空间隔离等）下，应用程序就不能直接以函数调用的方式访问操作系统的函数，以及直接读写操作系统的数据变量。不同类型的应用程序可以通过符合操作系统规定的系统调用接口，发出系统调用请求，来获得操作系统的服务。操作系统提供完服务后，返回应用程序继续执行。

### 1.1.3 进程和内存

#### 进程

站在应用程序自身的角度来看，进程 (Process) 的一个经典定义是一个正在运行的程序实例。当程序运行在操作系统中的时候，从程序的视角来看，它会产生一种“幻觉”：即该程序是整个计算机系统中当前运行的唯一的程序，能够独占使用处理器、内存和外设，而且程序中的代码和数据是系统内存中唯一的对象。

然而，这种“幻觉”是操作系统为了便于应用的开发且不损失安全性刻意为应用程序营造出来的，它具体表现为“进程”这个抽象概念。站在计算机系统和操作系统的角度来看，并不存在这种“幻觉”。事实上，在一段时间之内，往往会有多个程序同时或交替在操作系统上运行，因此程序并不能独占整个计算机系统。具体而言，进程是应用程序的一次执行过程。并且在这个执行过程中，由“操作系统”执行环境来管理程序执行过程中的进程上下文—一种控制流上下文。这里的进程上下文是指程序在运行中的各种物理/虚拟资源（寄存器、可访问的内存区域、打开的文件、信号等）的内容，特别是与程序执行相关具体内容：内存中的代码和数据，栈、堆、当前执行的指令位置（程序计数器的内容）、当前执行时刻的各个通用寄存器中的值等。

我们知道，处理器是计算机系统中的硬件资源。为了提高处理器的利用率，操作系统需要让处理器足够忙，即让不同的程序轮流占用处理器来运行。如果一个程序因某个事件而不能运行下去时，就通过进程上下文切换把处理器占用权转交给另一个可运行程序。进程上下文切换如下图所示：

基于上面的介绍，我们可以给进程一个更加准确的定义：一个进程是一个具有一定独立功能的程序在一个数据集合上的一次动态执行过程。操作系统中的进程管理需要采用某种调度策略将处理器资源分配给程序并在适当的时候回收，并且要尽可能充分利用处理器的硬件资源。

**内存** (Memory) 是计算机的重要部件，也称内存储器和主存储器，它用于暂时存放 CPU 中的运算数据，以及与硬盘等外部存储器交换的数据。它是外存与 CPU 进行沟通的桥梁（缓和 CPU 和硬盘之间的速度矛盾），计算机中所有程序的运行都在内存中进行，内存性能的强弱影响计算机整体发挥的水平。只要计算机开始运行，操作系统就会把需要运算的数据从内存调到 CPU 中进行运算，当运算完成，CPU 将结果传送出来。

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

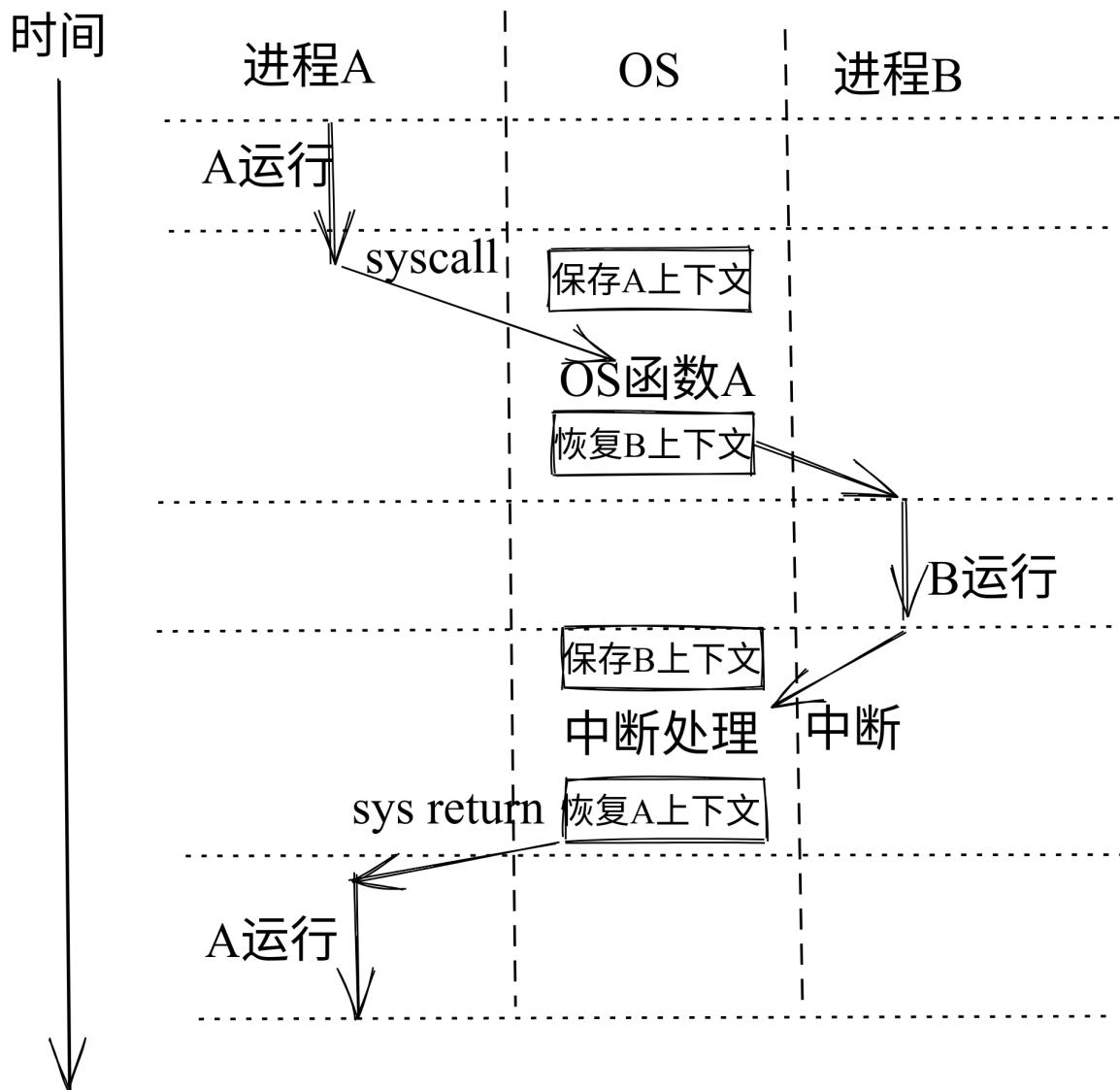


图 1-2 context switch

是如何高效、快速的分配，并且在适当的时候释放和收回内存资源。

一个执行中的程序，譬如网页浏览器在个人电脑或是图灵机（Turing machine）里面，为一个进程将资料转换于真实世界及电脑存储器之间，然后将资料存于电脑存储器内部（在电脑科学，一个程序是一群指令的集合，一个进程是电脑在执行中的程序）。存储器能被实际组织在许多方法里，例如磁带或是磁盘，或是小数组容量的微芯片。从1950年代开始，电脑变的更复杂，它被连线于许多种类的存储器。内存管理的任务也变得复杂，甚至必须要在同一台机器上相同的时间执行多个进程。

在现代操作系统中，对于每个用户层（user-level）的程序，操作系统分配一段虚拟内存空间，当进程开始时，不需要加载磁盘中的用户程序到物理内存中，当用户真正使用到时，才会加载到物理内存中。

### 1.1.4 I/O 与文件系统

#### I/O 设备

在操作系统中，I/O 设备的管理和控制非常重要。操作系统需要控制计算机的输入和输出，需要向设备发送命令，捕捉中断，并处理设备的各种错误，并为应用程序提供一个简单易用、统一的设备访问控制接口完成应用程序与设备的数据交互。同时，操作系统还需要控制设备的输入输出操作，保证数据的安全性和正确性。

I/O 设备的操作通常需要使用一些系统调用来完成。这些系统调用是操作系统提供给应用程序的一组函数，用于进行输入输出操作。例如，系统调用 `read()` 可以用于从设备文件中读取数据，系统调用 `write()` 可以用于向设备文件中写入数据。

在操作系统的 I/O 管理中，通常会采用一些优化技术来提高设备的输入输出效率。例如，缓冲技术可以暂存一部分数据，减少直接对设备的读写次数；异步 I/O 技术可以允许应用程序在等待设备操作完成时继续执行其他任务；直接内存访问技术可以允许应用程序直接对内存进行读写操作，避免通过操作系统的缓冲区进行数据交换。

#### 文件系统

文件系统是用于存储和组织文件的一种机制。文件系统通过特定的数据结构和算法，实现了对文件和目录的管理和控制，提供了方便的文件访问和使用方式。

文件系统的基本功能包括：存储信息、保存信息、共享信息（并发存取）。文件系统通过符号名来标识文件，便于用户使用；在多道程序系统中支持对文件的并发访问和控制；在不同的设备上提供同样的接口，方便用户操作和编程；同时，还提供多种文件访问权限，以及一定的差错恢复能力。

文件系统通过操作系统来进行管理，包括文件的结构、命名、存取、使用、保护和实现方法等。文件系统的架构通常包括设备驱动程序、基本文件系统等层次。设备驱动程序直接与外围设备通信，处理 I/O 请求的完成。基本文件系统又称为物理 I/O 层，是与计算机系统外部环境的基本接口。

文件系统中的文件是一组相似记录的集合，它被用户和应用程序视为一个实体，并可以通过名字访问。文件有一个唯一的文件名，可以被创建或删除。访问控制通常在文件级实施，也就是说，在一个共享系统中，用户和程序被允许或被拒绝访问整个文件。

## 1.2 NPUcore 操作系统

「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。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。目前 NPUcore 具有内存管理、进程管理、文件系统核心功能，支持 RISCV32/64 指令集，可在 QEMU 模拟器和 SiFive-U740、K210 等嵌入式开发板上运行。

以下为 NPUcore 的所有的系统调用：

## 1.3 预备知识及技能

### 1.3.1 RISC-V 和 LoongArch 指令集介绍

#### 1、RISC-V

RISC-V（发音为“risk-five”）是一个基于精简指令集（RISC）原则的开源指令集架构（ISA），简易解释为开源软体运动相对应的一种“开源硬体”。该项目 2010 年始于加州大学柏克莱分校，但许多贡献者是该大学以外的志愿者和行业工作者。

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器、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。新出现的 RISC-V 的核心目标是灵活适应未来的 AIoT 场景，保证基本功能，提供可配置的扩展功能。其开源特征使得学生都可以方便地设计一个 RISC-V CPU。

写面向 RISC-V 的 OS 的代价仅仅是你了解 RISC-V 的 Supervisor 特权模式，知道 OS 在 Supervisor 特权模式下的控制能力。

#### 2、LoongArch

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™。

龙架构已得到国际开源软件界广泛认可与支持，正成为与 X86/ARM 并列的顶层开源生态系统。已向 GNU 组织申请到 ELF Machine 编号（258 号），并获得 Linux、Binutils、GDB、.NET、GCC、LLVM、Go、Chromium/V8、Mozilla / SpiderMonkey、Javascript、FFmpeg、libyuv、libvpx、OpenH264、SRS 等音视频类软件社区、UEFI（UEFI 规范、ACPI

规范) 以及国内龙蜥开源社区、欧拉 openEuler 开源社区的支持。

指令系统是软件生态的起点，只有从指令系统的根源上实现自主，才能打破软件生态发展受制于人的锁链。龙架构的推出，是龙芯中科长期坚持自主研发理念的重要成果体现，是全面转向生态建设历史关头的重大技术跨越。

### 1.3.2 Rust 语言及其主要特性

Rust 是由 Mozilla 主导开发的通用、编译型编程语言。设计准则为“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。

Rust 语言原本是 Mozilla 员工 Graydon Hoare 的个人项目，而 Mozilla 于 2009 年开始赞助这个项目，并且在 2010 年首次公开。也在同一年，其编译器原始码开始由原本的 OCaml 语言转移到用 Rust 语言，进行自我编译工作，称做“rustc”，并于 2011 年实际完成。这个可自我编译的编译器在架构上采用了 LLVM 做为它的后端。

第一个有版本号的 Rust 编译器于 2012 年 1 月发布。Rust1.0 是第一个稳定版本，于 2015 年 5 月 15 日发布。

Rust 在完全公开的情况下开发，并且相当欢迎社区的反馈。在 1.0 稳定版之前，语言设计也因为透过撰写 Servo 网页浏览器排版引擎和 rustc 编译器本身，而有进一步的改善。它虽然由 Mozilla 资助，但其实是一个共有项目，有很大部分的代码是来自于社区的贡献者。

#### 1. 所有权

所有权是 Rust 的核心，也是其更有趣和独特的功能之一。“所有权”是指允许哪部分的代码修改内存。让我们从查看一些 C++ 代码开始：

代码片段 1.1 forktest.rs

```

1 int *dangling(void)
2 {
3     int i = 1234;
4     return &i;
5 }
6
7 int add_one(void)
8 {
9     int *num = dangling();
10    return *num + 1;
11 }
```

dangling 函数在栈上分配了一个整型，然后保存给一个变量 i，最后返回了这个变量 i 的引用。这里有一个问题：当函数返回时栈内存变成失效。意味着在函数 add\_one 第二行，指针 num 指向了垃圾值，我们将无法得到想要的结果。虽然这个一个简单的例子，但是在 C++ 的代码里会经常发生。当堆上的内存使用 malloc (或 new) 分配，然后使用 free (或 delete) 释放时，会出现类似的问题，但是您的代码会尝试使用指向该内存的指针执行某些操作。更现代的 C++ 使用 RAII 和构造函数/析构函数，但它们无法完

全避免“悬空指针”。这个问题被称为“悬空指针”，并且不可能编写出现“悬空指针”的 Rust 代码。我们试试吧：

代码片段 1.2 forktest.rs

```

1 fn dangling() -> &int {
2     let i = 1234;
3     return &i;
4 }
5
6 fn add_one() -> int {
7     let num = dangling();
8     return *num + 1;
9 }
```

当你尝试编译这个程序时，你会得到一个有趣和非常长的错误信息：

代码片段 1.3 forktest.rs

```

1 temp.rs:3:11: 3:13 error: borrowed value does not live long enough
2 temp.rs:3      return &i;
3
4 temp.rs:1:22: 4:1 note: borrowed pointer must be valid for the
5     anonymous lifetime #1 defined on the block at 1:22...
6 temp.rs:1 fn dangling() -> &int {
7     temp.rs:2     let i = 1234;
8     temp.rs:3     return &i;
9     temp.rs:4 }
10
11 temp.rs:1:22: 4:1 note: ...but borrowed value is only valid for the
12     block at 1:22
13 temp.rs:1 fn dangling() -> &int {
14     temp.rs:2     let i = 1234;
15     temp.rs:3     return &i;
error: aborting due to previous error
```

为了完全理解这个错误信息，我们需要谈谈“拥有”某些东西意味着什么。所以现在，让我们接受 Rust 不允许我们用悬空指针编写代码，一旦我们理解了所有权，我们就会回来看这块代码。

让我们先放下编程一会儿，先聊聊书籍。我喜欢读实体书，有时候我真的很喜欢一本书，并告诉我的朋友他们应该阅读它。当我读我的书时，我拥有它：这本书是我所拥有的。当我把书借给别人一段时间，他们向我“借用”这本书。当你借用一本书时，在特定的一段时间它是属于你的，然后你把它还给我，我又拥有它了。对吗？

这个概念也直接应用于 Rust 代码：一些代码“拥有”一个指向内存的特定指针。它是该指针的唯一所有者。它还可以暂时将该内存借给其他代码：代码“借用”它。借用它一段时间，称为“生命周期”。

这是关于所有权的所有。那似乎并不那么难，对吧？让我们回到那条错误信息：error: borrowed value does not live long enough。我们试图使用 Rust 的借用指针`&`，借出一个特定的变量`i`。但 Rust 知道函数返回后该变量无效，因此它告诉我们：

代码片段 1.4 forktest.rs

```

1 borrowed pointer must be valid for the anonymous lifetime #1
2
3 ... but borrowed value is only valid for the block.

```

这是栈内存的一个很好的例子，但堆内存呢？Rust 有第二种指针，一个‘唯一’指针，你可以用 ~ 创建。看看这个：

代码片段 1.5 forktest.rs

```

1 fn dangling() -> ~int {
2     let i = ~1234;
3     return i;
4 }
5
6 fn add_one() -> int {
7     let num = dangling();
8     return *num + 1;
9 }

```

此代码将成功编译。请注意，我们使用指针指向该值而不是将 1234 分配给栈：~1234。你可以大致比较这两行：

代码片段 1.6 forktest.rs

```

1 // rust
2 let i = ~1234;
3 // C++
4 int *i = new int;
5 *i = 1234;

```

Rust 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。这意味着无法分配未初始化的内存：Rust 没有 null 的概念。万岁！Rust 和 C++ 之间还有另外一个区别：Rust 编译器还计算了 i 的生命周期，然后在它无效后插入相应的 free 调用，就像 C++ 中的析构函数一样。您可以获得手动分配堆内存的所有好处，而无需自己完成所有工作。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。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助，编写错误的代码版本是不可能的。你已经看到了一种情况，所有权和生命周期有利于防止在不太严格的语言中通常会出现的危险代码。现在让我们谈谈另一种情况：并发。

## 2. 并发：

并发是当前软件世界中一个令人难以置信的热门话题。对于计算机科学家来说，它一直是一个有趣的研究领域，但随着互联网的使用爆炸式增长，人们正在寻求改善给定的服务可以处理的用户数量。并发是实现这一目标的一种方式。但并发代码有一个很大的缺点：它很难推理，因为它是非确定性的。编写好的并发代码有几种不同的方法，但让我们来谈谈 Rust 的所有权和生命周期的概念如何帮助实现正确并且并发的代码。

首先，让我们回顾一下 Rust 中的简单并发示例。Rust 允许你启动 task，这是轻量级的“绿色”线程。这些任务没有任何共享内存，因此，我们使用“通道”在 task 之间进行通信。像这样：

代码片段 1.7 forktest.rs

```

1 fn main() {
2     let numbers = [1,2,3];
3
4     let (port, chan) = Chan::new();
5     chan.send(numbers);
6
7     do spawn {
8         let numbers = port.recv();
9         println!("{}: {}", numbers[0]);
10    }
11 }
```

在这个例子中，我们创建了一个数字的 vector。然后我们创建一个新的 Chan，这是 Rust 实现通道的包名。这将返回通道的两个不同端：通道 (channel) 和端口 (port)。您将数据发送到通道端 (channel)，它从端口端 (port) 读出。spawn 函数可以启动一个 task。正如你在代码中看到的那样，我们在 task 中调用 port.recv()，我们在外面调用 chan.send()，传入 vector。然后打印 vector 的第一个元素。

这样做是因为 Rust 在通过 channel 发送时 copy 了 vector。这样，如果它是可变的，就不会有竞争条件。但是，如果我们正在启动很多 task，或者我们的数据非常庞大，那么为每个任务都 copy 副本会使我们的内存使用量膨胀而没有任何实际好处。

引入 Arc。Arc 代表“原子引用计数”，它是一种在多个 task 之间共享不可变数据的方法。这是一些代码：

代码片段 1.8 forktest.rs

```

1 extern mod extra;
2 use extra::arc::Arc;
3
4 fn main() {
5     let numbers = [1,2,3];
6
7     let numbers_arc = Arc::new(numbers);
8
9     for num in range(0, 3) {
10        let (port, chan) = Chan::new();
11        chan.send(numbers_arc.clone());
12
13        do spawn {
14            let local_arc = port.recv();
15            let task_numbers = local_arc.get();
16            println!("{}: {}", task_numbers[num]);
17        }
18    }
19 }
```

这与我们之前的代码非常相似，除了现在我们循环三次，启动三个 task，并在它们之间发送一个 Arc。Arc :: new 创建一个新的 Arc，.clone () 返回 Arc 的新的引用，而.get () 从 Arc 中获取该值。因此，我们为每个 task 创建一个新的引用，将该引用发送到通道，然后使用引用打印出一个数字。现在我们不 copy vector。

Arcs 非常适合不可变数据，但可变数据呢？共享可变状态是并发程序的祸根。您可以使用互斥锁 (mutex) 来保护共享的可变状态，但是如果您忘记获取互斥锁 (mutex)，则可能会发生错误。

Rust 为共享可变状态提供了一个工具：RWArc。Arc 的这个变种允许 Arc 的内容发生变异。看看这个：

代码片段 1.9 forktest.rs

```

1  extern mod extra;
2  use extra::arc::RWArc;
3
4  fn main() {
5      let numbers = [1,2,3];
6
7      let numbers_arc = RWArc::new(numbers);
8
9      for num in range(0, 3) {
10          let (port, chan) = Chan::new();
11          chan.send(numbers_arc.clone());
12
13          do spawn {
14              let local_arc = port.recv();
15
16              local_arc.write(|nums| {
17                  nums[num] += 1
18              });
19
20              local_arc.read(|nums| {
21                  println!("{}:{}", num, nums[num]);
22              })
23          }
24      }
25  }

```

我们现在使用 RWArc 包来获取读/写 Arc。RWArc 的 API 与 Arc 略有不同：读和写允许您读取和写入数据。它们都将闭包作为参数，并且在写入的情况下，RWArc 将获取互斥锁，然后将数据传递给此闭包。闭包完成后，互斥锁被释放。

你可以看到在不记得获取锁的情况下是不可能改变状态的。我们获得了共享可变状态的便利，同时保持不允许共享可变状态的安全性。

但是我们不能同时允许和禁止可变状态。是什么赋予了这种能力的？

### 3.unsafe:

因此，Rust 语言不允许共享可变状态，但我刚刚向您展示了一些允许共享可变状态的代码。这怎么可能？答案：unsafe。

你看，虽然 Rust 编译器非常聪明，并且可以避免你通常犯的错误，但它不是人工智能。因为我们比编译器更聪明，有时候，我们需要克服这种安全行为。为此，Rust 有一个 `unsafe` 关键字。在一个 `unsafe` 的代码块里，Rust 关闭了许多安全检查。如果您的程序出现问题，您只需要审核您在不安全范围内所做的事情，而不是整个程序。

如果 Rust 的主要目标之一是安全，为什么要关闭安全？嗯，实际上只有三个主要原因：与外部代码连接，例如将 FFI 写入 C 库，性能（在某些情况下），以及围绕通常不安全的操作提供安全抽象。我们的 `Arcs` 是最后一个目的的一个例子。我们可以安全地分发对 `Arc` 的多个引用，因为我们确信数据是不可变的，因此可以安全地共享。我们可以分发对 `RWArc` 的多个引用，因为我们知道我们已经将数据包装在互斥锁中，因此可以安全地共享。但 Rust 编译器无法知道我们已经做出了这些选择，所以在 `Arcs` 的实现中，我们使用不安全的块来做（通常）危险的事情。但是我们暴露了一个安全的接口，这意味着 `Arcs` 不可能被错误地使用。

这就是 Rust 的类型系统如何让你不会犯一些使并发编程变得困难的错误，同时也能获得像 C++ 等语言一样的效率。

我希望这个对 Rust 的尝试能让您了解 Rust 是否适合您。如果这是真的，我建议您查看完整的教程，以便对 Rust 的语法和概念进行全面，深入的探索。

### 1.3.3 如何查资料

#### 1、查手册

##### 程序自带的文档 (1) README 和 INSTALL

很多程序在编译或者安装过程中都会自带一个 README 和 INSTALL 文件，不要漏掉，否则可能会有重要的信息遗漏并导致某些严重问题。

其中，如果 INSTALL 文件被单独呈现，则其往往是解释软件的安装方式的，有的软件有很特殊的安装要求，如执行脚本的位置必须是在文件夹内或者文件夹外，如果漏掉可能导致软件完全运行不起来。

README 一般介绍软件的使用方式，文档获取位置和帮助信息，有时候也介绍安装方法。

例如，你在 NPUcore 的源代码文件夹中可以找到 README: (2) `help` 参数绝大多数程序会自带一个 `help` 选项，甚至不加任何参数。例如，`man` 命令的 `help` 参数：

代码片段 1.10 forktest.rs

```
1 whatis --help
```

会打印出：那么，如果你直接在终端中敲入 `help` 并回车，会发生什么呢（假设你使用的是 bash）？请自己试一下。

此外，帮助文档的提供软件自身往往也会有自己的帮助文档，显然自产自销是最合适的。

## Welcome to NPUcore 🙋

### 简介

NPUcore 是来自西北工业大学的三位同学基于清华大学 rCore Tutorial 框架，参考借鉴去年内核赛道优秀参赛队伍与 Linux 内核的诸多优秀设计，并结合硬件平台特性进行优化和重构，不断迭代形成的竞赛操作系统，现已支持 qemu、k210、Hifive Unmatched 三平台的运行。

### 特性

- 完善的内核功能支持

图 1-3 readme

```
dragon@dragon-Lenovo-K2450:~/Documents/EduDept/Computer/MyCode/rCore/backup/NPUcore_tutorial$ whatis
--help

用法： whatis [选项...] 关键词...

-d, --debug          输出调试信息
-v, --verbose         输出详细的警告信息
-r, --regex           把每个关键词都当作正则表达式解读
-w, --wildcard        关键词里包含通配符
-l, --long            不要把输出按终端宽度截断
-C, --config-file=文件 使用该用户设置文件
-L, --locale=区域    定义本次搜索所使用的区域设置
-m, --systems=系统   使用来自其它系统的手册页
-M, --manpath=路径   设置搜索手册页的路径为 PATH
-s, --sections=列表, --section=列表
                     仅在这些分区中搜索（冒号分隔）
-?, --help            显示此帮助列表
--usage              显示一份简洁的用法信息
-V, --version         打印程序版本

选项完整形式所必须用的或是可选的参数，在使用选项缩写形式时也是必须的或是可选的。

请向 ciwatson@debian.org 报告错误。
```

图 1-4 help

所以，你不妨试一下 man man，或者进入 man 后有没有能看到的某些 man 自己的帮助文档（仔细找，我这么说就一定有）。

之后的任何帮助套件也可以“自己帮助自己”，所以文献中不再赘述。

### (3) TAB 补全

在 Bash 中，TAB 补全是一种非常有用的功能，它可以让用户更快捷、更准确地输入命令和文件路径。在终端输入命令或文件路径时，如果按下 TAB 键，Bash 会尝试自动补全输入的内容。

下面是关于 Bash 中 TAB 补全常见类型（如果没有特别说明，则 Bash 会列出这些选项供选择）：

- 命令补全：输入一个命令的前几个字母时，补全该命令的名称。如果有多个以该字符串开头的命令，
- 文件路径补全：输入一个文件或目录的路径时，补全路径中的文件或目录名称。如果有多个符合条件的文件或目录，
- 变量名补全：输入一个变量名时，补全该变量的名称。如果有多个符合条件的变量名，
- 命令参数补全：输入命令的参数时，补全该命令所支持的参数选项。如果有多个符合条件的参数选项，
- 目录补全：在输入路径时，如果您只知道路径中的某些部分，可以使用通配符进行

补全。例如，输入”/u/lo\*”，按下 TAB 键可以自动补全为”/usr/local”。

总之，bash 中的 TAB 补全是一种非常方便的功能，可以让用户更快速地输入命令和路径，并且减少输入错误的可能性。

此外，TAB 补全需要程序自身和终端的支持，有时候甚至需要单独配置，(例如 rust 的工具链就需要自行配置对应的 shell 补全选项)

### **man**

在 Linux 操作系统中，man 命令是一个非常重要的命令，它可以帮助用户查看 Linux 系统中各种命令的手册。

使用 man 只需要在终端中输入”man”加上要查看的命令名称，然后按下回车键即可。例如：

代码片段 1.11 forktest.rs

```
1 man help
```

man 命令将会显示出该命令的手册页，可以使用键盘上的箭头键进行滚动，并且可以使用 “/” 加上关键字进行查找。

在手册页中，可以查看该命令的使用方法、参数选项、示例以及其他相关信息。man 软件本身的帮助信息可以在软件中按 “h” 查看。

当不再需要查看手册页时，可以按下 “q” 键退出 man 命令。

**完整手册**多数成体系的大型软件系统会有自己对应的文档，一般称为手册。具体来说，这种文档会出现在官方网站的 Documentation 环节，且往往有在线或者线下 PDF 两种版本。

我们以 GNU GCC 为例，在 <https://gcc.gnu.org/> 中，浏览器搜索（一般快捷键是 Ctrl-F）Documentation，下方的 Manual 就是手册。点进去会有各种格式的手册。

有的成熟的软件或者语言会提供 Tutorial 和 Reference Manual，后者倾向于列举所有的性质，前者则是为入门初学者提供的简单的教程。

一般而言，绝大多数的软件是自身具有自己的手册的，但部分软件的手册是集合型的，或者本身就是其 manpage 的集合。

一个典型的例子是 coreutils，其中包括了 cut, head, tail 等简单工具；另一个是 binutils，包括各种 GCC 的常用工具。这时候需要自行查询其手册的所在之处。

### **教材**

很多的软件都有自己的教材，且层次从入门到精通都有，如果你有需要，可以找买一本合适的书从中学习。一般教材会比官方手册更详细，并提供作者自己的思考。

### **TLDR**

TLDR 是 “Too long, don't read.” 的缩写，如果要最快获得某个命令的简单使用方法，tldr 是一个不错的来源。例如我们输入

代码片段 1.12 forktest.rs

```

1 $ tldr man
2
3 Format and display manual pages. More information: https://www.man7.org/linux/man-pages/man1/man.1.html.
4
5 - Display the man page for a command:
6   man {{command}}
7
8 - Display the man page for a command from section 7:
9   man {{7}} {{command}}
10
11 - List all available sections for a command:
12   man -f {{command}}
13
14 - Display the path searched for manpages:
15   man --path
16
17 - Display the location of a manpage rather than the manpage itself:
18   man -w {{command}}
19
20 - Display the man page using a specific locale:
21   man {{command}} --locale={{locale}}
22
23 - Search for manpages containing a search string:
24   man -k "{{search_string}}"

```

## info

注意，info 是用某个目录作为中心数据库的，所以完全可能存在在一个软件中可以阅读但在另一个软件中读不了的情况。

info 中有大量长篇的完整文档，一般就是上述完整手册。一般各种 IDE 本身也会自带 Info 的阅读器。只是 info 有自己的搜索，历史记录等功能（有的功能是配合 IDE 使用的），这里不再赘述。

此外，GNU 套件几乎所有的工具都有 info 文档。所谓的 GNU 套件 PDF 文档就是用 info 相关的一个软件 texinfo 写的。

我们以 gdb 为例展示其内容。终端输入“info gdb”可以得到：

另外，info 文档的安装方法如下：

对于软件自带文档，一般可以：

代码片段 1.13 info

```

1 sudo apt-get install gdb-doc
2 有时候需要去网站上搜索并下载，就会需要：
3 whereis info # 获得info的安装目录
4 # 注意：有时候下载到的是一个info.tar.gz，这时候需要自行解压，
5 # 但是如果是info.gz，则可以直接跳过这一步，因为gz一般是自动解压的。
6 # 另外，有时候文档是以texi后缀名出现的
7 tar xf <infofilepath> <tmp_path>
8 sudo install-info bison.info <one_of_the_info_paths>/dir

```

## 自动补全与文档显示插件

多数的 IDE 有自己的文档现实和自动补全插件，可以在光标悬停在某个符号一段时间后自动显示特定的文档。

很多 IDE 还会集成之前的所说的这些文档查询方式，从而在内部查询所有的文献。

请自行搜索自己的编辑器和 IDE 的文档寻找配置方式。

## 搜索引擎

如果你遇到什么工具你无法使用或者不会用，可以尝试通过搜索引擎寻找替代品，在线版或者其他能让你用上的方法（比如能够代理你请求的某些接口，软件和网站）。

## 论坛

论坛往往是最后一步，一般来说很少出现别人没有发现过而自己发现的问题，毕竟计算机工业确实过于发达了。但是，在 stackoverflow 和其他论坛上提问仍然有可能可以获得比较好的效果。

如果人家恰好碰到过或者有兴趣帮你解决问题的话是再幸运不过的事，不过在此之前，请先不要往下看，尝试通过之前几步查找可能技术论坛以便解决问题。

然后这是一份作者根据自己回忆写的常见的论坛，你可以试着在上面发帖。此外，一般使用量大的项目都有自己的论坛，你也可以自行查找。

```

Next: Summary, Up: (dir)

Debugging with GDB
***** 

This file describes GDB, the GNU symbolic debugger.

  This is the Tenth Edition, for GDB (Ubuntu 12.1-0ubuntu1~22.04)
Version .

  Copyright (C) 1988-2022 Free Software Foundation, Inc.

  This edition of the GDB manual is dedicated to the memory of Fred
Fish. Fred was a long-standing contributor to GDB and to Free software
in general. We will miss him.

* Menu:

* Summary::          Summary of GDB
* Sample Session::    A sample GDB session

* Invocation::         Getting in and out of GDB
* Commands::          GDB commands
* Running::            Running programs under GDB
* Stopping::           Stopping and continuing
* Reverse Execution::  Running programs backward
* Process Record and Replay:: Recording inferior's execution and replaying it
* Stack::               Examining the stack
* Source::              Examining source files
* Data::                Examining data
* Optimized Code::     Debugging optimized code
* Macros::              Preprocessor Macros
* Tracepoints::         Debugging remote targets non-intrusively
* Overlays::            Debugging programs that use overlays

* Languages::          Using GDB with different languages

* Symbols::             Examining the symbol table
* Altering::            Altering execution
* GDB Files::           GDB files
* Targets::              Specifying a debugging target
* Remote Debugging::    Debugging remote programs
* Configurations::     Configuration-specific information
* Controlling GDB::     Controlling GDB
* Extending GDB::       Extending GDB
* Interpreters::        Command Interpreters
* TUI::                 GDB Text User Interface
* Emacs::                Using GDB under GNU Emacs
* GDB/MI::               GDB's Machine Interface.

-----Info: (gdb)Top, 78 lines --Top-----
Welcome to Info version 6.8. Type H for help, h for tutorial.

```

图 1-5 info

## 第 2 章 内核构建初探

### 2.1 NPUcore 内核运行

#### 2.1.1 本地运行 NPUcore

##### 编译和运行 NPUcore 需要的环境

NPUcore 的编写使用 Rust 语言, 编译需要 RISC-V 的编译器, 同时使用 make 来辅助编译。编译完成后, 可在 QEMU (虚拟机软件) 或是硬件平台 (U740 和 K210) 上运行。

**什么是 make** make 是一个在软件开发中所使用的工具程序 (Utility software), 经由读取 “makefile” 的文件以自动化建构软件。它是一种转化文件形式的工具, 转换的目标称为 “target”; 与此同时, 它也检查文件的依赖关系, 如果需要的话, 它会调用一些外部软件来完成任务。它的依赖关系检查系统非常简单, 主要根据依赖文件的修改时间进行判断。大多数情况下, 它被用来编译源代码, 生成结果代码, 然后把结果代码连接起来生成可执行文件或者库文件。它使用叫做 “makefile” 的文件来确定一个 target 文件的依赖关系, 然后把生成这个 target 的相关命令传给 shell 去执行。

许多现代软件的开发中 (如 Microsoft Visual Studio), 集成开发环境已经取代 make, 但是在 Unix 环境中, 仍然有许多工程师采用 make 来协助软件开发。

**NPUcore 环境配置** 建议首先在 VMware 中配置一个运行 Ubuntu22.04 操作系统的虚拟机, 随后在 Ubuntu 中配置编译和运行环境。首先是在 Ubuntu 中配置 Rust 的编译链工具, 以实现 NPUcore 的编译; 其次是安装 QEMU, 为 NPUcore 的运行提供虚拟的 RISC-V 硬件环境。下面将讲解具体的配置方法。

##### Rust 编译工具安装

代码片段 2.1 forktest.rs

```

1 一些基础编译包的安装:
2 sudo apt-get install git build-essential gdb-multiarch qemu-system-misc
   gcc-riscv64-linux-gnu binutils-riscv64-linux-gnu
3 安装Rust版本管理器rustup和Rust包管理器Cargo:
4 curl -sSf https://sh.rustup.rs | sh
5 安装时注意选择nightly版本, 默认的是stable版本。
6 如果没有curl, 请先安装curl
7 sudo apt-get install curl
8
9 Cargo 是 Rust 的构建系统和包管理器。大多数Rust程序员使用 Cargo 来管理他们的 Rust 项目, 因为它可以为你处理很多任务, 比如构建代码、下载依赖库并编译这些库。(我们把代码所需要的库叫做依赖 (dependencies) )。
10 最简单的 Rust 程序, 没有任何依赖。如果使用 Cargo 来构建 “Hello, world!” 项目, 将只会用到 Cargo 构建代码的那部分功能。在编写更复杂的 Rust 程序时, 你将添加依赖项, 如果使用 Cargo 启动项目, 则添加依赖项将更容易。
11 rustup 是Rust的安装程序。rustup 从官方发布渠道安装Rust编程语言, 使你能够在稳定版、测试版和nightly编译器之间轻松切换并保持更新。它使交叉编

```

译变得更加简单，为普通平台的标准库建立二进制。而且它可以在Rust支持的所有平台上运行。

如果官方的脚本在运行时出现了网络速度较慢的问题，可选地可以通过修改 `rustup` 的镜像地址（修改为清华大学的镜像服务器）来加速：

```
12 export RUSTUP_DIST_SERVER=https://mirrors.tuna.edu.cn/rustup
13 export RUSTUP_UPDATE_ROOT=https://mirrors.tuna.edu.cn/rustup/rustup
14 curl -sSf https://sh.rustup.rs | sh
```

安装完成后，记得重新打开一个终端使新的环境变量生效。在新终端中，输入以下命令来查看安装的`rustup`的版本，以验证是否成功安装：

```
17 rustc --version
```

若出现类似下面的输出，即代表安装成功

```
18 rustc 1.71.0-nightly (7f94b314c 2023-04-23)
```

安装完成后，我们需要重新打开一个终端来让之前设置的环境变量生效。

下面利用上面步骤安装好的`rustup`和`cargo`来安装Rust相关软件包

```
22 rustup target add riscv64gc-unknown-none-elf
```

```
23 cargo install cargo-binutils
```

```
24 rustup component add llvm-tools-preview
```

```
25 rustup component add rust-src
```

至此我们实验所需的Rust编译环境已经配置完成。

至于开发环境，推荐使用Visual Studio Code搭配`rust-analyzer`和RISC-V Support插件。

### QEMU 安装

这里安装QEMU虚拟机是为了在其上运行你将要编写的NPUcore。注意需要使用QEMU7.0.0版本，低版本的QEMU可能无法运行NPUcore。很多Linux发行版的软件包管理器默认软件源中的QEMU版本过低，因此我们需要从源码手动编译安装QEMU模拟器软件。

安装编译QEMU源码所需的依赖包：

代码片段 2.2 forktest.rs

```
1 sudo apt install autoconf automake autotools-dev curl libmpc-dev
2     libmpfr-dev libgmp-dev \
3 gawk build-essential bison flex texinfo gperf libtool patchutils bc \
4 zlib1g-dev libexpat-dev pkg-config libglib2.0-dev libpixman-1-dev
5     libsdl2-dev \
6 git tmux python3 python3-pip ninja-build
```

下载QEMU源代码：

代码片段 2.3 forktest.rs

```
1 wget https://download.qemu.org/qemu-7.0.0.tar.xz
```

解压：

代码片段 2.4 forktest.rs

```
1 tar xvJf qemu-7.0.0.tar.xz
```

编译QEMU源码，生成可运行的QEMU软件：

代码片段 2.5 forktest.rs

```

1 cd qemu-7.0.0 // 打开解压后的文件夹
2 ./configure --target-list=riscv64-softmmu,riscv64-linux-user // 配置编
   译选项
3 make -j$(nproc)

```

配置环境变量，使你能在任意位置直接运行 QEMU：

代码片段 2.6 forktest.rs

```

1 下面提供两种配置方式：
2 直接将编译后的QEMU软件放到/usr/local/bin目录下，但这样可能会引起冲突。
3 sudo make install
4 使用Debian系统提供的包管理工具
5 sudo checkinstall

```

运行上述命令后，会在当前位置生成一个.deb 的文件，随后鼠标右击，选择用包管理器安装即可。这种法安装后的软件卸载起来也会很方便，直接在包管理器中卸载即可。

### QEMU 上运行 NPUcore 内核

虚拟机软件 QEMU 为 NPUcore 提供了一个虚拟的硬件环境，在使用物理机前，一切对于硬件环境的改变（如增加外设等）都要通过操作 QEMU 来实现。本节主要介绍 QEMU 的使用方法，以及 NPUcore 在 QEMU 上的运行过程。

#### 了解 QEMU

QEMU 是一个通用的开源机器模拟处理软件和虚拟机，是最早由法布里斯 · 贝拉 (Fabrice Bellard) 编写的以 GPL 许可证分发源码的模拟处理器软件，在 GNU/Linux 平台上使用广泛。拥有高速、跨平台的特性。当将 QEMU 用作机器模拟处理软件时，QEMU 可以在一类机器（例如 PC 机）上运行为另一类机器（例如 ARM 板）制作的操作系统和程序。通过使用动态翻译机制 (dynamic translation)，QEMU 实现了强大的性能。当将 QEMU 用作虚拟器时，QEMU 通过直接在主机 CPU 上执行代码来实现接近本机的性能。QEMU 在 Xen 管理程序下执行或在 Linux 中使用 KVM 内核模块时支持虚拟化。使用 KVM 时，QEMU 可以虚拟化 x86、服务器和嵌入式 PowerPC、64 位 POWER、S390、32 位和 64 位 ARM 以及 MIPS 客户机。

QEMU 包含多种运作模式，包括：

(1)**User mode:** 用户模式，在这种模式下，QEMU 执行针对不同指令编译的单个 Linux 或 Darwin/macOS 程序。在这种模式下，我们可以实现交叉编译 (cross-compilation) 与交叉调试 (cross-debugging) (2)**System mode:** 系统模式。在这一模式下，QEMU 能模拟整个电脑系统，包括中央处理器及其他周边设备。它使得为跨平台编写的程序进行测试及除错工作变得容易。其亦能用来在一部主机上虚拟数部不同虚拟电脑。(3)**KVM Hosting:** QEMU 在这时处理 KVM 镜像的设定与迁移，并参加硬件的仿真，但是客户端的执行则由 KVM 完成。(4)**Xen Hosting:** 在这种模式下，客户端的执行几乎完全在 Xen 中完成，并且对 QEMU 封锁。QEMU 只提供硬件仿真的支援。

其中，用户模式与系统模式为 QEMU 的主要运作模式。

QEMU 作为应用广泛的虚拟机软件，具有可扩展，可自定义新的指令集，开源，可移植，仿真速度快，默认支持多种架构等诸多优势，但同时 QEMU 对微软 Windows 及某些主机操作系统支持不完善（某些模拟的系统仅能运行），对不常用的架构的支持不完善，除非使用 kqemu 或 KVM 加速器，否则其模拟速度仍不及其他虚拟软件，如 VMware 等，同时安装和使用较为复杂。

## QEMU 的常用命令

代码片段 2.7 forktest.rs

```

1 $check [-f fmt] filename
2
3 对磁盘镜像文件进行一致性检查，查找镜像文件中的错误，目前仅支持对“qcow2”
4   “qed”、“vdi”格式文件的检查。其中，qcow2是QEMU 0.8.3版本引入
5   的镜像文件格式，也是目前使用最广泛的格式。qed (QEMU enhanced disk)
6   是从QEMU 0.14版开始加入的增强磁盘文件格式，为了避免qcow2格式的一些缺
7   点，也为了提高性能，不过目前还不够成熟。而vdi (Virtual Disk Image)
8   是Oracle的VirtualBox虚拟机中的存储格式。参数-f fmt是指定文件的格式，
9   如果不指定格式qemu-img会自动检测，filename是磁盘镜像文件的名称（包括
10  路径）。
11
12 $create [-f fmt] [-o options] filename [size]
13
14 创建一个格式为fmt大小为size文件名为filename的镜像文件。根据文件格式fmt
15  的不同，还可以添加一个或多个选项(options)来附加对该文件的各种功能
16  设置，可以使用“-o ?”来查询某种格式文件支持那些选项，在“-o”选项中
17  各个选项用逗号来分隔。
18
19 如果“-o”选项中使用了backing_file这个选项来指定其后端镜像文件，那么这
20  个创建的镜像文件仅记录与后端镜像文件的差异部分。后端镜像文件不会被修
21  改，除非在QEMU monitor中使用“commit”命令或者使用“qemu-img commit”
22  命令去手动提交这些改动。这种情况下，size参数不是必须需的，其值默认
23  为后端镜像文件的大小。另外，直接使用“-b backfile”参数也与“-o
24  backing_file=backfile”效果相同。
25
26 size选项用于指定镜像文件的大小，其默认单位是字节(bytes)，也可以支持k
27  (或K)、M、G、T来分别表示KB、MB、GB、TB大小。另外，镜像文件的大小(
28  size)也并非必须写在命令的最后，它也可以被写在“-o”选项中作为其中一个
29  选项。
30
31 对create命令的演示如下所示，其中包括查询qcow2格式支持的选项、创建有
32  backing_file的qcow2格式的镜像文件、创建没有backing_file的10GB大小的
33  qcow2格式的镜像文件。
34
35 $commit [-f fmt] [-t cache] filename
36
37 提交filename文件中的更改到后端支持镜像文件（创建时通过backing_file指定
38  的）中去。
39
40 $convert [-c] [-p] [-f fmt] [-t cache] [-O output_fmt] [-o options] [-s
41  snapshot_name] [-S sparse_size] filename [filename2 [...]] output_filename
42
43 将fmt格式的filename镜像文件根据options选项转换为格式为output_fmt的名为
44  output_filename的镜像文件。它支持不同格式的镜像文件之间的转换，比如
45  可以用VMware用的vmdk格式文件转换为qcow2文件，这对从其他虚拟化方案转
46  移到KVM上的用户非常有用。一般来说，输入文件格式fmt由qemu-img工具自动
47  检测到，而输出文件格式output_fmt根据自己需要来指定，默认会被转换为与
48  
```

19        raw文件格式（且默认使用稀疏文件的方式存储以节省存储空间）。  
 其中，“-c”参数是对输出的镜像文件进行压缩，不过只有qcow2和qcow格式的镜像文件才支持压缩，而且这种压缩是只读的，如果压缩的扇区被重写，则会被重写为未压缩的数据。同样可以使用“-o options”来指定各种选项，如：后端镜像、文件大小、是否加密等等。使用backing\_file选项来指定后端镜像，让生成的文件是copy-on-write的增量文件，这时必须让转换命令中指定的后端镜像与输入文件的后端镜像的内容是相同的，尽管它们各自后端镜像的目录、格式可能不同。

20        如果使用qcow2、qcow、cow等作为输出文件格式来转换raw格式的镜像文件（非稀疏文件格式），镜像转换还可以起到将镜像文件转化为更小的镜像，因为它可以将空的扇区删除使之在生成的输出文件中并不存在。

21  
22        \$info [-f fmt] filename  
23  
24        展示filename镜像文件的信息。如果文件是使用稀疏文件的存储方式，也会显示出它的本来分配的大小以及实际已占用的磁盘空间大小。如果文件中存放有客户机快照，快照的信息也会被显示出来。

25  
26        \$snapshot [-l | -a snapshot | -c snapshot | -d snapshot] filename  
27  
28        “-l”选项是查询并列出镜像文件中的所有快照，“-a snapshot”是让镜像文件使用某个快照，“-c snapshot”是创建一个快照，“-d”是删除一个快照。  
 29        \$rebase [-f fmt] [-t cache] [-p] [-u] -b backing\_file [-F backing\_fmt] filename  
30        改变镜像文件的后端镜像文件，只有qcow2和qed格式支持rebase命令。使用“-b backing\_file”中指定的文件作为后端镜像，后端镜像也被转化为“-F backing\_fmt”中指定的后端镜像格式。

31        它可以工作于两种模式之下，一种是安全模式（Safe Mode）也是默认的模式，qemu-img会去比较原来的后端镜像与现在的后端镜像的不同进行合理的处理；另一种是非安全模式（Unsafe Mode），是通过“-u”参数来指定的，这种模式主要用于将后端镜像进行了重命名或者移动了位置之后对前端镜像文件的修复处理，由用户去保证后端镜像的一致性。

32  
33        \$resize filename [+ | -]size  
34  
35        改变镜像文件的大小，使其不同于创建之时的大小。“+”和“-”分别表示增加和减少镜像文件的大小，而size也是支持K、M、G、T等单位的使用。缩小镜像的大小之前，需要在客户机中保证里面的文件系统有空余空间，否则会数据丢失，另外，qcow2格式文件不支持缩小镜像的操作。在增加了镜像文件大小后，也需启动客户机到里面去应用“fdisk”、“parted”等分区工具进行相应的操作才能真正让客户机使用到增加后的镜像空间。不过使用resize命令时需要小心（最好做好备份），如果失败的话，可能会导致镜像文件无法正常使用而造成数据丢失。

## 编译具体操作流程

操作系统并不神秘，无非就是一个更复杂的软件，它的编译流程和普通的helloworld没有区别：

在shell中打开os目录后，执行make命令即可看到系统编译过程的运行截图：

### QEMU上运行NPUCore内核

在shell中输入make run即可在QEMU中启动运行NPUCore。QEMU相关指令如下：

代码片段2.8 在QEMU中启动运行NPUCore指令

```
1 @qemu-system-riscv64 \
```

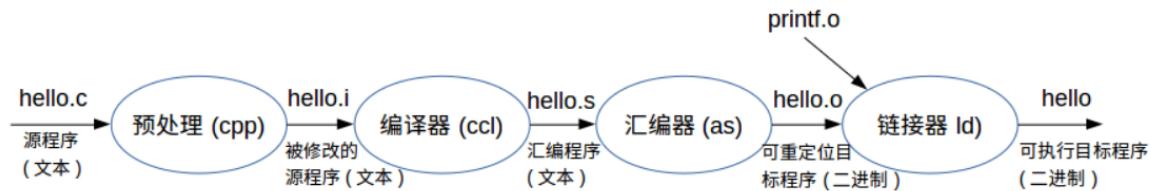


图 2-1 C 语言文件的编译流程  
编译 NPUcore

```

(rustup target list | grep "riscv64gc-unknown-none-elf (installed)") || rustup target add riscv64gc-unknown-none-elf
riscv64gc-unknown-none-elf (installed)
cargo install cargo-binutils
  Updating crates.io index
    Ignored package `cargo-binutils v0.3.6` is already installed, use --force to override
rustup component add rust-src
info: component 'rust-src' is up to date
rustup component add llvm-tools-preview
info: component 'llvm-tools-preview' for target 'x86_64-unknown-linux-gnu' is up to date
(which last-qemu) || (rm -f last-k210 && touch last-qemu && make clean)
make[1]: Entering directory '/home/xm/Desktop/rCore-Tutorial-v3/os'
make[1]: Leaving directory '/home/xm/Desktop/rCore-Tutorial-v3/os'
Platform: qemu
  Compiling memchr v2.3.4
  Compiling regex-syntax v0.6.27
  Compiling lazy_static v1.4.0
  Compiling semver-parser v0.7.0
  Compiling proc-macro2 v1.0.43
  Compiling quote v1.0.21
  Compiling autocfg v1.1.0
  Compiling unicode-ident v1.0.4
  Compiling az v1.2.1
  Compiling log v0.4.17
  Compiling bit_field v0.10.1
  Compiling syn v1.0.100
  Compiling k210-pac v0.2.0 (https://github.com/wyfcyx/k210-pac#8f99cica)
  Compiling cfg-if v1.0.0
  Compiling version_check v0.9.4
  Compiling bitflags v1.3.2
  Compiling nb v1.0.0
  Compiling vcell v0.1.3
  Compiling byteorder v1.4.3
  Compiling spin v0.5.2
  Compiling spin v0.7.1

```

图 2-2 NPUcore 编译过程

```

2   -M 128m \
3   -machine virt \
4   -bios $(BOOTLOADER) \
5   -device loader,file=$(KERNEL_BIN),addr=$(KERNEL_ENTRY_PA) \
6   -drive file=$(FS_IMG),if=none,format=raw,id=x0 \
7   -device virtio-blk-device,drive=x0 \
8   -device virtio-gpu-device \
9   -device virtio-keyboard-device \
10  -device virtio-mouse-device \
11  -serial stdio
12 其中：
13  $(BOOTLOADER) 为引导内核的 BIOS 的二进制文件；
14  $(KERNEL_BIN) 为 NPUcore 的二进制文件；
15  $(KERNEL_ENTRY_PA) 为 NPUcore 的二进制文件的起始执行地址；
16  $(FS_IMG) 为虚拟硬盘（对应现实中的硬盘）；

```

运行结果图：

在上图中，从第一行开始到 “[kernel] Hello, world!” 的上一行均为 OpenSBI 输出的相关信息，感兴趣的同学可以自行搜索 SBI 的相关内容，这里不做展开。

## 2.1.2 在线运行 NPUcore

NPUcore 操作系统内核已部署在头歌平台、Gitee、GitHub 平台。为广大教师和学生学习操作系统内核构建提供线上实训支持。它也为大赛“操作系统设计赛”内核赛道区域赛顺利晋级提供支撑。可以通过在头歌平台上的实验环境在线运行 NPUcore，以下是头歌平台内的链接：

代码片段 2.9 NPUcore 实验在头歌平台上的链接

<https://www.educoder.net/paths/7wnb29j6>

## 2.2 基于 GDB 内核调试

在软件开发过程中，调试是不可避免的。一个程序往往不会一开始就按照程序员预期的方式运行，对操作系统这样的复杂巨系统而言尤其如此。本节主要讲解调试软件 GDB 的使用方法。

### 2.2.1 认识 GDB

GNU 调试器（英语：GNU Debugger，缩写：GDB），是 GNU 软件系统中的标准调试器，此外 GDB 也是个具有移植性的调试器，经过移植需求的调修与重新编译，如今许多的类 UNIX 操作系统上都可以使用 GDB，而现有 GDB 所能支持调试的编程语言有 C、C++、Pascal 以及 FORTRAN。

#### 为什么需要 GDB

当程序出现错误时，开发者需要快速地找出错误的原因，并修复它们。很多人会倾向于使用“人工静态分析”（也就是目测法和冥想法）解决 Bug。

然而，程序的运行时状态往往非常复杂，有时很难在代码中准确地定位错误。具体来说，目测的以下缺点导致开发者往往会百思不得其解进而无功而返：

**1. 无法检测运行时问题：**代码静态分析只能检测静态代码问题，例如语法错误、类型错误等，它无法检测代码的运行时问题。例如，它无法检测到由于代码在特定环境下执行而引起的问题，如内存泄漏、死锁等。

**2. 误报和漏报：**静态分析可能会误会遗漏问题。例如，可能会将某些无害的代码标记为错误，或者忽略某些实际上是错误的代码。这可能会导致开发人员浪费时间和精力来调查错误的根本原因。

**3. 对高质量代码的依赖：**代码静态分析需要高质量的代码才能进行准确的分析。如果代码质量不好，例如缺乏注释、变量名不规范、代码冗余等，那么静态分析可能会产生误报或漏报。

**4. 难以发现复杂的问题：**静态分析工具通常使用各种分析技术来分析代码，但这些技术很难发现复杂的问题，例如多线程问题、分布式系统问题等。这些问题通常需要动态调试或其他更高级的技术来解决。

同样，也有人会尝试插入 LOG 打印部分状态，但是这种方法除了费时费力，且暴露

状态不够精确的问题之外，在 OS 中，某些 LOG 会产生系统状态的改变，进而影响结果，导致 debug 失败。

这时候，调试工具就显得非常 important 了。调试工具可以帮助开发者在运行时监视程序的状态，跟踪代码的执行流程，查看变量的值，以及定位错误的位置。这正是 GDB 的用途。

GDB 最初由 Richard Stallman 在他的 GNU Emacs 系统稳定后于 1986 年编写，并设计作为他的 GNU 系统的一部分。GDB 是根据 GNU 通用公共许可证 (GPL) 发布的免费软件。它是在 Berkeley Unix 发行版附带的 DBX 调试器之后建模的。从 1990 年到 1993 年，它由 John Gilmore 维护。现在由自由软件基金会任命的 GDB 指导委员会维护。

GDB 允许用户查看一个程序在执行时“内部”的执行过程—，或者查看程序在崩溃时的内部状态。这些被调试的程序可以与 GDB 在同一台机器上（本地）、另一台机器（远程）或模拟器上执行。GDB 可以在大多数流行的 UNIX 和 Microsoft Windows 变体以及 Mac OS X 上运行。具体而言，目前 GDB 支持以下程序：Ada、Assembly、C、C++、D、Fortran、Go、Objective-C、OpenCL、Modula-2、Pascal、Rust 等。

GDB 默认只有命令行接口（CLI）可用，而不具备较能亲合上手、直觉操作的图形用户界面（GUI），不过此一弱处也已经有几个前端程序为其补强，例如 DDD、GDBtk / Insight（页面存档备份，存于互联网档案馆）以及 Emacs 中的“GUD 模式”等，有了这些补强后，GDB 在功效使用的便利性上就能够与“集成发展环境中的调试功效使用”相接近。

### GDB 的启动

显然，启动 gdb 有不同的方法，在终端中输入 gdb 是最简单的，但 NPUcore 在 RISC-V 上构建，因此不能直接使用本机的 gdb(除非你使用一台 RISC-V64 计算机)，因此我们推荐安装并使用 gdb-multiarch(这里需要 Ubuntu 环境)：

代码片段 2.10 forktest.rs

```

1 $ sudo apt-get install gdb-multiarch
2 # 然后启动：
3 $ gdb-multiarch

```

注意，这里实际上有“工作目录”的概念，也就是你的当前目录实际上最好在项目或者源代码的路径上，否则会需要手工加载源代码路径（方法见下文）。

## 2.2.2 基于 GDB 的内核调试

### QEMU 虚拟机的相关命令

介绍 GDB 为什么要先介绍虚拟机呢？因为正是 QEMU 与 GDB 合作，才给了我们方便地进行大部分系统软件调试的机会。

作为一款全虚拟化虚拟机，QEMU 能彻底模拟 CPU 的内部状态，包括寄存器和其他部分，因此很适合进行调试。

具体来说, QEMU 配合 GDB 提供了单步执行、断点调试、内存监视、寄存器查看等。

用户可以使用调试功能逐步执行代码, 查看每一步的运行结果和寄存器状态, 同时还可以设置断点, 方便定位问题所在。

利用 QEMU 提供的远程调试功能, 允许用户在另一台计算机上通过网络连接到 QEMU 的调试接口进行调试。这个功能可以方便地在不同的计算机之间进行协作开发和调试。

这里我们只介绍本地的远程调试。

为了方便, 在 os 文件夹中, 使用下列 make 命令直接进行 gdb 调试:

代码片段 2.11 forktest.rs

```
1 make gdb
```

其后端执行实际命令是:

代码片段 2.12 forktest.rs

```
1 gdb:
2 @qemu-system-riscv64 -machine virt -nographic -bios $(BOOTLOADER) -
3   device loader,
4   file=target/riscv64gc-unknown-none-elf/debug/os,addr=0x80200000 -drive
5     \
6   file=$(U_FAT32),if=none,format=raw,id=x0 \
7   -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0 -smp threads=$
8     (CORE_NUM) -S -s
```

和 do-run 的内容相比,

代码片段 2.13 forktest.rs

```
1 do-run:
2 @qemu-system-riscv64 \
3 -machine virt \
4 -nographic \
5 -bios $(BOOTLOADER) \
6 -device loader,file=$(KERNEL_BIN),addr=$(KERNEL_ENTRY_PA) \
7 -drive file=$(U_FAT32),if=none,format=raw,id=x0 \
8 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0 \
9 -smp threads=$(CORE_NUM)
```

不难发现最主要的差别在于后面多出的”-S -s”。前面的 S 代表 STOP, 意思是设置完虚拟机直接挂起, 停止一切执行, 直到接收到外部的 continue 信息为止.

第二个小写 s 等价于”-gdb tcp::1234”, 指的是开启远程调试, 其在 localhost(本机) 的 1234 端口侦听 GDB 的信号, 等待连接。然后, 就是我们的下一个工具 GDB 的任务和工作范畴了。

历史在 GDB 的命令行 (和各种 IDE 的 GDB 控制台) 中, 使用上下左右 (或者 Alt-P 之类的自定义按键) 可以直接显示之前的命令, 这和 Bash 是一致的。

但和 Bash 不同的是, 其重复执行上一条命令可以通过直接在不进行任何输入的时候敲回车实现:

## 代码片段 2.14 forktest.rs

```

1 (gdb) stepi
2 (gdb) (然后这里敲回车)

```

这时候就前进了两条指令。但这个功能也会导致有时候多按了一下回车结果重复执行了某些只应当被执行一次的指令，因此也要注意使用的场景。**补全**

GDB 命令的确数量庞大且内容复杂，但是但作为一款老牌软件，GDB 自然也有解决方案。一方面，GDB 自己提供了强大的命令补全功能，能像在 Bash 中一样 TAB 补全，例如如下所示的键盘输入：输入 b 然后按下 TAB 键，会补全为 break，其他的，如寄存器名称等往往也可以在有了部分提示前缀后进行补全。因此不需要每次都键入完整的命令。

**辅助**

另一方面，多数的 IDE 和编辑器都有辅助 GDB 的功能，例如在某一行代码旁边点击行号附近的位置，会出现一个圆点，表示加入断点。

另外，如果你需要重复某个命令多遍，并不需要一直按着鼠标或者键盘回车键，只需要在命令后面插入重复次数即可：

## 代码片段 2.15 forktest.rs

```

1 (gdb) stepi 4

```

这里就前进了 4 条指令。

**常见命令**

进入 gdb，会看到”(gdb)”提示可以输入命令（有时候无法输入）。

如果在 VSCode 中使用，还需要在之前加上 exec

## 代码片段 2.16 forktest.rs

```

1 exec gdb <...>

```

**设定命令（1）连接**

按照之前的方法启动 QEMU 后，GDB 要通过下列命令连接本地的 QEMU。

## 代码片段 2.17 forktest.rs

```

1 target remote :1234
2
3 <div align=center></
4   div>
5

```

如果你之前指定了自定义的端口，需要将 1234 换成其他的端口号。同时，冒号之前实际上省略了 localhost（也就是本机的“网址”），如果你将来有自定义的地址或者网址，也可以在前面补上。

**（2）加载调试信息**

## 代码片段 2.18 forktest.rs

```

1 (1)file
2 在开始调试之前，你首先需要加载调试信息。
3 file target/riscv64gc-unknown-none-elf/debug/os
4 从而加载os文件作为符号文件。请注意，使用release版本的文件（在make命令
5 中加入“MODE=release”得到）往往不带有任何的调试信息，不适合用于
6 debug，但也不尽然：你可以对着汇编语言调试。当然，就算使用了带有
7 符号文件的版本，这种体验你总会遇到的，因为操作系统总是要涉及某些底层。
8 这里加载的是操作系统的符号，那如果某些过程经过用户程序（作为一个操作系
9 统，你总会遇到这种问题），如何添加用户程序的代码？这就要用到下一个命
10 令了。
11 如果先连接QEMU后加载二进制文件，就会出现上面的“A program is being
12 debugged already.”提示，当然这并不影响使用。
13
14 $add-symbol-file
15
16 $ add-symbol-file bash

```

如果你的当前工作文件夹中具有 bash 文件，就可以直接添加，否则需要自行前往特定的。显然，上面两个命令的顺序可以修改，但注意，file 只能有一个，symbol-file 却可以有很多，且符号文件指代的不一定是带有符号部分的可执行文件，也可能是纯粹的符号文件（考虑到其和主题无关，这里的內容我们不拓展，读者可以进一步查找资料）。这时候可以 info files 显示所有已经添加的符号和二进制文件

## 代码片段 2.19 forktest.rs

```

1 (1)directory
2 设置完符号文件，接着需要设置源代码目录，这样在IDE/GDB中可以自动跳转到
3 函数代码所在处。
4 $dir ~/Downloads/SW/Bash/bash-5.1.16/
5 $dir ~/Downloads/SW/Bash/bash-5.1.16/lib/sh/
6 注意，这里需要你将bash的源代码先提前下载好到某个目录，并将上面的这个
7 路径替换成正确的目录。
8 最终得到：
9 Source directories searched: /home/dragon/Downloads/SW/Bash/bash
10 -5.1.16/lib/sh:/home/dragon/Downloads/SW/Bash/bash-5.1.16:$cdir:$cwd
11 另外，部分的软件目录结构复杂，这时候需要手动用上述命令添加。
12 (2)break
13 break用于设置断点。可以通过断点中断程序的执行并让你进入调试模式。
14 一般常见的断点设定方式有：文件：行号格式和函数(方法)格式
15 例如：(基于特定版本，你的具体地址与行号可能不同)
16 $(gdb) b src/main.rs:50
17 Breakpoint 1 at 0x900000000004f158: file src/main.rs, line 59.
18 $(gdb) b rust_main
19 Breakpoint 2 at 0x900000000004f198: file src/main.rs, line 66.
20
21 注意！函数名方法有时候要指定域，格式类似os::rust_main;
22 如果要删除断点，则可以
23 $delete 1
24 跟上断点号即可。
25 (3)set
26 set可以是多种的，最典型的是让pc强制移动到某个位置，例如：
27 $set $pc=0x0
28 回到最开始的执行点。当然，你也可以用它对别的地址/变量进行强行赋值。

```

## 执行流

(1)continue 很显然, GDB 有两种状态, 停止和执行, 只有在停止的时候, 我们才能对其中的数据进行查看和修改, 对自己的命令进行调整, 而 continue 正是从停止到执行的切换工具。

continue 会继续执行程序直到遇到下一个断点或程序结束。这条命令往往用于 target remote :1234 后继续执行。

例如, 我们一开始执行 make gdb 只有:

代码片段 2.20 forktest1.rs

```
1 $ make gdb
```

一个光标停在原地。

但是, 如果你在 gdb 中输入 continue 并回车, 虚拟机马上就会停止冻结, 开始执行指令(直到撞到某个停止条件为止):

此时, 命令就会停在之前设定的断点上 (2) 暂停或终止运行

在终端和多数 IDE 中, 暂停执行流是通过 gdb 控制台(注意不是虚拟机的终端, 而是 gdb 的控制台) ctrl-C(同时按下 Ctrl 和 C) 实现的, 这可以让程序暂停在当前执行到的位置。

如果你之前在执行 QEMU 时没有加入 “-S” 选项, 那么你可以用这条命令立即暂停执行流(当然其具体停止位置难以保证。)

完成后 debug 后, 可以用 quit 退出。

(3)next, step 和 finish

stepi 前进一条指令.

next 可以单步执行程序, 跳过函数调用, 例如(中间省略几步 continue 和 break):

可以看到这里的几条函数都被跳过了。

step 单步执行程序, 进入函数调用(这里的执行流进入了函数调用):

finish 执行完当前函数并返回到调用函数, 然后又回到了之前的函数:

## 查看命令

在停止状态下, 我们可以 backtrace 显示函数调用栈:

或者 print 打印寄存器/变量/内存地址的值:

如果觉得每次都打印很麻烦, 可以用 display 每次停在断点处时自动打印某个变量的值。一旦不需要, 可以 undisplay 该号码取消(类似 delete 语法), 如下列这段话打印附近前后各 6 条汇编代码(其他的变量也可以打印, 语法类似)

代码片段 2.21 forktest1.rs

```
1 display/12i $pc-6*4
```

也可以 info registers 打印所有的寄存器

## 自定义命令与脚本

在使用 GDB 进行调试时，我们需要多次执行一些常见的指令。为了提高调试效率，我们可以使用 `define` 命令来定义自己的命令，简化重复操作。

下面我们来介绍如何定义一个自定义命令。首先，使用文本编辑器创建一个 `gdb` 脚本文件，例如 `mycommands.gdb`。然后，在该文件中添加以下代码：

代码片段 2.22 forktest1.rs

```

1 # 加载符号文件
2 file target/riscv64gc-unknown-none-elf/debug/os
3 add-symbol-file bash
4 # 井号加入注释
5 define mynext
6 stepi
7 info registers
8 end
9 # 添加bash的源代码
10 dir

```

这个自定义命令名为 `mynext`，执行的操作包括执行下一条指令和显示所有寄存器的值。

接下来，启动 GDB 并加载定义的自定义命令。我们可以通过以下命令将 `mycommands.gdb` 文件加载到 GDB 中：

代码片段 2.23 forktest1.rs

```
1 source mycommands.gdb
```

现在，我们可以在 GDB 中使用 `mynext` 命令来执行下一条指令并显示所有寄存器的值。只需在 GDB 提示符下输入 “`mynext`” 即可。

使用 `define` 自定义命令可以帮助我们快速地执行常见的调试操作，提高调试效率。另外，其他的断点添加，远程调试连接等命令也可以很方便地加入其中从而加速 Debug 过程。

在命令行下启动 GDB 并加载脚本时，可以使用 `-x` 或 `-command` 选项来指定要执行的脚本文件。该选项后跟要执行的脚本文件路径，如下所示：

代码片段 2.24 forktest1.rs

```

1 # -x 选项指定了要执行的脚本文件路径，file_to_debug 则是要调试的目标程序
2 # 的路径。
3 gdb -x /path/to/script file_to_debug
4 # 除了 -x 选项外，还可以使用 --init-command 选项指定要执行的初始化命
# 令，该选项可以多次使用，每次指定一条命令，如下所示：
5 gdb --init-command="set print pretty on" --init-command="set pagination
off" file_to_debug
# 上述命令中，--init-command 选项指定了要执行的初始化命令，可以多次使
用，每次指定一条命令。

```

### 2.2.3 git bisect——快速问题定位

大家一定听过二分查找的算法, 如果我们发现某个 Bug 出现, 其实也可以通过二分查找定位到出错的版本。git 也自带了这个功能。使用 git bisect, 我们可以在 Git 版本控制系统中进行二分查找, 在版本历史中快速定位错误引入的位置。一般步骤我们以这些图文为例给出一个错误的处理

```

1 $ git bisect start
2 //运行 git bisect start 命令来启动一个二分查找会话。
3 $ git bisect bad
4 //用 git bisect bad 命令告诉 Git 当前版本存在问题。
5 $ git bisect good HEAD~10
6 //用 git bisect good 命令告诉 Git 一个知道没有问题的提交, 是这个提交的
7 哈希值或分支名。git 会给出估计的剩余步骤数
8 Bisecting: 4 revisions left to test after this (roughly 2 steps)
9 [f55d253527e3a72f730b06e6fbfe5e64f8594a27] fix: Change ELF related AuxV
   data alignment to repr(C), fixing the LPF.
10 //Git 会自动切换到一个介于上述两个提交之间的提交, 你需要在该提交上运行
11 你的程序, 检查问题是否存在。
12 如果有问题, 使用 git bisect bad 命令告诉 Git, 否则使用 git bisect good
13 命令告诉 Git。
14 你可以使用 Git bisect run 切换提交后自动执行的命令(注意, git bisect run
15 后仍然在原地, 这时候需要 git bisect next 才能进入下一个, 否则会冲突)
16 Git 会根据你的反馈自动切换到下一个介于两个提交之间的提交, 重复上述步
17 骤, 直到找到引入问题的提交。
18 最后, 使用 git bisect reset 命令退出二分查找会话。

```

**log 与冲突回撤**如果发现某个提交被错误标记 (比如 bad 被错误标记为 good), 尝试

```

1 git bisect bad
2 你会得到以下错误:
3 ba246b7c5294eeabcdca705faf8ea1015e6fd6e was both good and bad
4
5 $ git bisect log//导出日志:
6
7 git bisect start
8 # good: [ba246b7c5294eeabcdca705faf8ea1015e6fd6e] add: Expect script
9   and clauses in Makefile for automation
10 git bisect good ba246b7c5294eeabcdca705faf8ea1015e6fd6e
11 # bad: [ba246b7c5294eeabcdca705faf8ea1015e6fd6e] add: Expect script
12   and clauses in Makefile for automation
13 git bisect bad ba246b7c5294eeabcdca705faf8ea1015e6fd6e
14 //重定向到文件
15 $ git bisect log > bis.log
16 //考虑修改错误
17 $ git bisect start
18 会得到以下信息:
19 # bad: [ba246b7c5294eeabcdca705faf8ea1015e6fd6e] add: Expect script
20   and clauses in Makefile for automation
21 $ git bisect bad ba246b7c5294eeabcdca705faf8ea1015e6fd6e
22
23 $ git bisect replay bis.log
24 //回复到之前的状态

```

## 2. 3 NPUcore 内核代码结构及内核构建目标

### 2. 3. 1 NPUcore 内核代码树

下面是 NPUcore 的内核代码树：

### 2. 3. 2 NPUcore 学习路线

下面是我们为你准备的 NPUcore 的学习路线

希望你能渐渐的喜爱上操作系统～

## 2. 4 实验

```

OpenSBI v1.0

Platform Name          : riscv-virtio,qemu
Platform Features       : medeleg
Platform HART Count    : 1
Platform IPI Device    : aclint-mswi
Platform Timer Device   : aclint-mtimer @ 10000000Hz
Platform Console Device: uart8250
Platform HSM Device     : ---
Platform Reboot Device  : sifive_test
Platform Shutdown Device: sifive_test
Firmware Base          : 0x80000000
Firmware Size           : 252 KB
Runtime SBI Version     : 0.3

Domain0 Name            : root
Domain0 Boot HART        : 0
Domain0 HARTS             : 0*
Domain0 Region00         : 0x000000002000000-0x00000000200ffff (I)
Domain0 Region01         : 0x000000008000000-0x000000008003ffff ()
Domain0 Region02         : 0x000000000000000-0xffffffffffff (R,W,X)
Domain0 Next Address      : 0x0000000080200000
Domain0 Next Arg1         : 0x0000000082200000
Domain0 Next Mode          : S-mode
Domain0 SysReset          : yes

Boot HART ID             : 0
Boot HART Domain          : root
Boot HART ISA              : rv64imafdcuh
Boot HART Features         : scounteren,mcounteren,time
Boot HART PMP Count        : 16
Boot HART PMP Granularity : 4
Boot HART PMP Address Bits: 54
Boot HART MHPM Count       : 0
Boot HART MIDELEG          : 0x0000000000001666
Boot HART MEDELEG          : 0x00000000000f0b509
[kernel] Hello, world!
last 1349 Physical Frames.
.text [0x80200000, 0x8023f000)
.rodata [0x8023f000, 0x80249000)
.data [0x80249000, 0x8024a000)
.bss [0x8024a000, 0x8049b000)
mapping .text section
mapping .rodata section
mapping .data section
mapping .bss section
mapping physical memory
mapping memory-mapped registers
NPUCore:/# 

```

图 2-3 NPUcore 正常启动

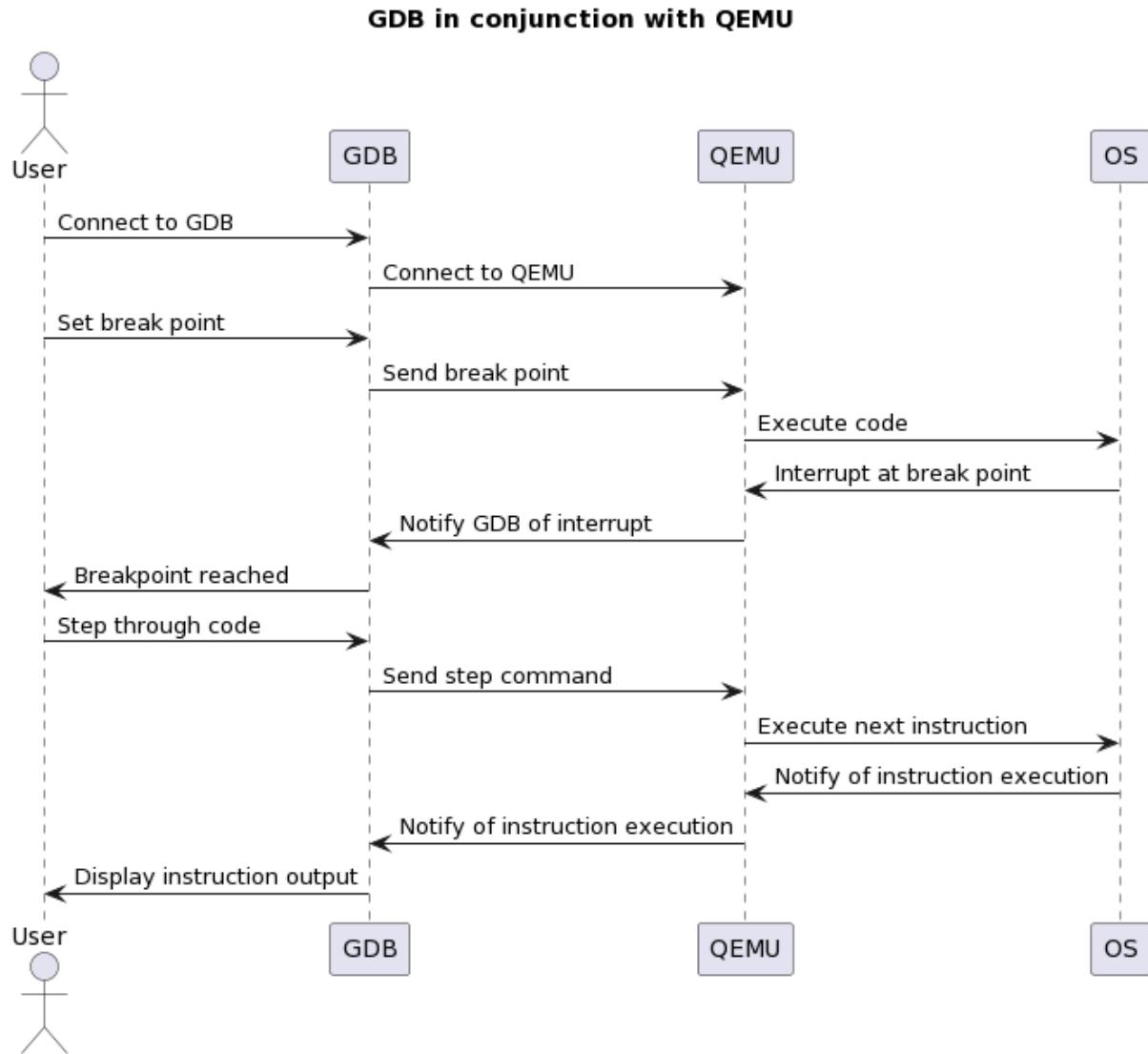


图 2-4 GDB 联调 QEMU 的逻辑流程

```
(gdb) info files
Symbols from "/home/dragon/文档/EduDept/Computer/MyCode/rCore/os_2021/os/target/riscv64gc-unknown-none-elf/debug/os".
Remote target using gdb-specific protocol:
`/home/dragon/文档/EduDept/Computer/MyCode/rCore/os_2021/os/target/riscv64gc-unknown-none-elf/debug/os', file type elf64-littleriscv.
Entry point: 0x80200000
0x00000000000200000 - 0x000000008024598e is .text
0x0000000000080246000 - 0x00000000080251560 is .rodata
0x0000000000080252000 - 0x00000000080252628 is .data
0x0000000000080253000 - 0x000000000804a34f8 is .bss
0x0000000000010120 - 0x0000000000ad6bc is .text in /home/dragon/文档/EduDept/Computer/MyCode/rCore/os_2021/os/bash
0x00000000000ad6c0 - 0x0000000000f1714 is .rodata in /home/dragon/文档/EduDept/Computer/MyCode/rCore/os_2021/os/bash
0x00000000000f1714 - 0x00000000000f1718 is .eh_frame in /home/dragon/文档/EduDept/Computer/MyCode/rCore/os_2021/os/bash
0x00000000000f3480 - 0x00000000000f3488 is .init_array in /home/dragon/文档/EduDept/Computer/MyCode/rCore/os_2021/os/bash
0x00000000000f3488 - 0x00000000000f3490 is .fini_array in /home/dragon/文档/EduDept/Computer/MyCode/rCore/os_2021/os/bash
0x00000000000f3490 - 0x00000000000f6000 is .data.rel.ro in /home/dragon/文档/EduDept/Computer/MyCode/rCore/os_2021/os/bash
0x00000000000f6000 - 0x00000000000fe568 is .data in /home/dragon/文档/EduDept/Computer/MyCode/rCore/os_2021/os/bash
0x00000000000fe568 - 0x00000000000ff6d8 is .got in /home/dragon/文档/EduDept/Computer/MyCode/rCore/os_2021/os/bash
0x00000000000ff6d8 - 0x00000000000ff6e0 is .sdata in /home/dragon/文档/EduDept/Computer/MyCode/rCore/os_2021/os/bash
0x00000000000ff6e0 - 0x0000000000010b970 is .bss in /home/dragon/文档/EduDept/Computer/MyCode/rCore/os_2021/os/bash
While running this, GDB does not access memory from...
Local exec file:
`/home/dragon/文档/EduDept/Computer/MyCode/rCore/os_2021/os/target/riscv64gc-unknown-none-elf/debug/os', file type elf64-littleriscv.
Entry point: 0x80200000
0x00000000000200000 - 0x000000008024598e is .text
0x0000000000080246000 - 0x00000000080251560 is .rodata
0x0000000000080252000 - 0x00000000080252628 is .data
0x0000000000080253000 - 0x000000000804a34f8 is .bss
0x0000000000010120 - 0x0000000000ad6bc is .text in /home/dragon/文档/EduDept/Computer/MyCode/rCore/os_2021/os/bash
0x00000000000ad6c0 - 0x0000000000f1714 is .rodata in /home/dragon/文档/EduDept/Computer/MyCode/rCore/os_2021/os/bash
0x00000000000f1714 - 0x00000000000f1718 is .eh_frame in /home/dragon/文档/EduDept/Computer/MyCode/rCore/os_2021/os/bash
0x00000000000f3480 - 0x00000000000f3488 is .init_array in /home/dragon/文档/EduDept/Computer/MyCode/rCore/os_2021/os/bash
0x00000000000f3488 - 0x00000000000f3490 is .fini_array in /home/dragon/文档/EduDept/Computer/MyCode/rCore/os_2021/os/bash
0x00000000000f3490 - 0x00000000000f6000 is .data.rel.ro in /home/dragon/文档/EduDept/Computer/MyCode/rCore/os_2021/os/bash
0x00000000000f6000 - 0x00000000000fe568 is .data in /home/dragon/文档/EduDept/Computer/MyCode/rCore/os_2021/os/bash
0x00000000000fe568 - 0x00000000000ff6d8 is .got in /home/dragon/文档/EduDept/Computer/MyCode/rCore/os_2021/os/bash
0x00000000000ff6d8 - 0x00000000000ff6e0 is .sdata in /home/dragon/文档/EduDept/Computer/MyCode/rCore/os_2021/os/bash
0x00000000000ff6e0 - 0x0000000000010b970 is .bss in /home/dragon/文档/EduDept/Computer/MyCode/rCore/os_2021/os/bash
(gdb)
```

图 2-5 info files

```
dragon@dragon-Lenovo-K2450:~/Documents/EduDept/Computer/MyCode/rCore/os_2021/os$ make gdb
OpenSBI v1.0
[REDACTED]
Platform Name      : riscv-virtio,qemu
Platform Features  : medeleg
Platform HART Count: 1
Platform IPI Device: aclint-mswi
Platform Timer Device: aclint-mtimer @ 10000000Hz
Platform Console Device: uart8250
Platform HSM Device: ---
Platform Reboot Device: sifive_test
Platform Shutdown Device: sifive_test
Firmware Base      : 0x80000000
Firmware Size       : 252 KB
Runtime SBI Version: 0.3

Domain0 Name        : root
Domain0 Boot HART    : 0
Domain0 HARTs        : 0*
Domain0 Region00     : 0x000000002000000-0x00000000200ffff (I)
Domain0 Region01     : 0x000000008000000-0x000000008003ffff ()
Domain0 Region02     : 0x000000000000000-0xffffffffffff (R,W,X)
Domain0 Next Address: 0x000000008020000
Domain0 Next Arg1   : 0x000000008220000
Domain0 Next Mode    : S-mode
Domain0 SysReset    : yes

Boot HART ID        : 0
Boot HART Domain    : root
Boot HART ISA        : rv64imafdcuh
Boot HART Features   : scounteren,mcounteren,mcountinhibit,time
Boot HART PMP Count  : 16
Boot HART PMP Granularity: 4
Boot HART PMP Address Bits: 54
Boot HART MHPM Count: 16
Boot HART MIDELEG   : 0x0000000000001666
Boot HART MEDELEG   : 0x000000000f0b509
```

图 2-6 设置断点

```
0x0000000000000001000 in ?? ()
(gdb) stepi
0x0000000000000001004 in ?? ()
(gdb)
```

图 2-7 stepi

```
(gdb) next
722     info!()
(gdb) l
717     };
718     let newpath = match translated_str(token, newpath) {
719         Ok(path) => path,
720         Err(errno) => return errno,
721     };
722     info!()
723     "[sys_renameat2] olldirfd: {}, oldpath: {}, newdirfd: {}, newpath: {}, flags: {}",
724     olldirfd as isize, oldpath, newdirfd, newpath, flags
725 );
726
(gdb) next
723     "[sys_renameat2] olldirfd: {}, oldpath: {}, newdirfd: {}, newpath: {}, flags: {}",
(gdb)
727     let old_file_descriptor = match olldirfd {
(gdb)
732         Ok(file_descriptor) => file_descriptor.clone(),
(gdb)
728         AT_FDCWD => task.fs.lock().working_inode.as_ref().clone(),
(gdb)
732         Ok(file_descriptor) => file_descriptor.clone(),
(gdb) ■
```

图 2-8 next

```
(gdb) step
os::task::processor::current_trap_cx () at src/task/processor.rs:74
74      current_task().unwrap().acquire_inner_lock().get_trap_cx()
(gdb) step
os::task::processor::current_task () at src/task/processor.rs:66
66      PROCESSOR.lock().current()
(gdb)
```

图 2-9 step

```
(gdb) finish
Run till exit from #0  os::task::processor::current_task () at src/task/processor.rs:66
os::task::processor::current_trap_cx () at src/task/processor.rs:74
74      current_task().unwrap().acquire_inner_lock().get_trap_cx()
(gdb) ■
```

图 2-10 finish

```
(gdb) backtrace
#0  os::task::processor::current_trap_cx () at src/task/processor.rs:74
#1  0x0000000080220d48 in os::trap::trap_handler () at src/trap/mod.rs:81
#2  0x0000000000095fd4 in open64 ()
(gdb) print
```

图 2-11 backtrace

```
(gdb) print 0x802256da
$2 = 2149734106
(gdb) print $pc
$3 = (*mut fn ()) 0x802256d6 <os::task::processor::current_trap_cx+72>
(gdb) ■
```

图 2-12 print

```
(gdb) display/12i $pc-6*4
1: x/12i $pc-6*4
0x802256bc <_ZN2os4task9processor15current_trap_cx17hd87f65f794493641E+46>: j      0x802256ae
<_ZN2os4task9processor15current_trap_cx17hd87f65f794493641E+32>
0x802256be <_ZN2os4task9processor15current_trap_cx17hd87f65f794493641E+48>: ld    a1,128(a0)
0x802256c0 <_ZN2os4task9processor15current_trap_cx17hd87f65f794493641E+50>: beqz a1,0x802256cc
<_ZN2os4task9processor15current_trap_cx17hd87f65f794493641E+62>
0x802256c2 <_ZN2os4task9processor15current_trap_cx17hd87f65f794493641E+52>: li    a3,1
0x802256c4 <_ZN2os4task9processor15current_trap_cx17hd87f65f794493641E+54>: amoadd.d a3,a3,(a1)
0x802256c8 <_ZN2os4task9processor15current_trap_cx17hd87f65f794493641E+58>: bltz a3,0x80225762
<_ZN2os4task9processor15current_trap_cx17hd87f65f794493641E+212>
0x802256cc <_ZN2os4task9processor15current_trap_cx17hd87f65f794493641E+62>: addi a2,a2,1
0x802256ce <_ZN2os4task9processor15current_trap_cx17hd87f65f794493641E+64>: fence rw,w
0x802256d2 <_ZN2os4task9processor15current_trap_cx17hd87f65f794493641E+68>: sd    a2,8(a0)
=> 0x802256d4 <_ZN2os4task9processor15current_trap_cx17hd87f65f794493641E+70>: beqz a1,0x80225732
<_ZN2os4task9processor15current_trap_cx17hd87f65f794493641E+164>
0x802256d6 <_ZN2os4task9processor15current_trap_cx17hd87f65f794493641E+72>: sd    a1,-32(s0)
0x802256da <_ZN2os4task9processor15current_trap_cx17hd87f65f794493641E+76>: addi a0,a1,16
(gdb) stepi
74      current_task().unwrap().acquire_inner_lock().get_trap_cx()
1: x/12i $pc-6*4
0x802256be <_ZN2os4task9processor15current_trap_cx17hd87f65f794493641E+46>: ld    a1,128(a0)
0x802256c0 <_ZN2os4task9processor15current_trap_cx17hd87f65f794493641E+50>: beqz a1,0x802256cc
<_ZN2os4task9processor15current_trap_cx17hd87f65f794493641E+62>
0x802256c2 <_ZN2os4task9processor15current_trap_cx17hd87f65f794493641E+52>: li    a3,1
0x802256c4 <_ZN2os4task9processor15current_trap_cx17hd87f65f794493641E+54>: amoadd.d a3,a3,(a1)
0x802256c8 <_ZN2os4task9processor15current_trap_cx17hd87f65f794493641E+58>: bltz a3,0x80225762
<_ZN2os4task9processor15current_trap_cx17hd87f65f794493641E+212>
0x802256cc <_ZN2os4task9processor15current_trap_cx17hd87f65f794493641E+62>: addi a2,a2,1
0x802256ce <_ZN2os4task9processor15current_trap_cx17hd87f65f794493641E+64>: fence rw,w
0x802256d2 <_ZN2os4task9processor15current_trap_cx17hd87f65f794493641E+68>: sd    a2,8(a0)
0x802256d4 <_ZN2os4task9processor15current_trap_cx17hd87f65f794493641E+70>: beqz a1,0x80225732
<_ZN2os4task9processor15current_trap_cx17hd87f65f794493641E+164>
=> 0x802256d6 <_ZN2os4task9processor15current_trap_cx17hd87f65f794493641E+72>: sd    a1,-32(s0)
0x802256da <_ZN2os4task9processor15current_trap_cx17hd87f65f794493641E+76>: addi a0,a1,16
0x802256de <_ZN2os4task9processor15current_trap_cx17hd87f65f794493641E+80>: li    a2,1
```

图 2-13 display

```
(gdb) info registers
ra          0x8021cd88      0x8021cd88 <os::syscall::fs::sys_openat+156>
sp          0xffffffffffffbe50      0xffffffffffffbe50
gp          0xffed8  0xffed8
tp          0x10b898 0x10b898
t0          0x0      0
t1          0x80212800      2149656576
t2          0xbffffeb60      3221220192
fp          0xffffffffffffbef0      0xffffffffffffbef0
s1          0x80425b80      2151832448
a0          0x1      1
a1          0x80426018      2151833624
a2          0x80424000      2151825408
a3          0x8      8
a4          0x1      1
a5          0x8024edc0      2149903808
a6          0x1      1
a7          0x0      0
s2          0x80426120      2151833888
s3          0x8802  34818
s4          0x8      8
s5          0xfffffffffffff9c      -100
s6          0xfffffffffffffea      -22
s7          0x1      1
s8          0x0      0
s9          0x0      0
s10         0x0      0
s11         0x0      0
t3          0xfe564  1041764
t4          0x0      0
t5          0x10040  65600
t6          0x0      0
pc          0x8021ce24      0x8021ce24 <os::syscall::fs::sys_openat+312>
(gdb)
```

图 2-14 info registers

```
$ git bisect run echo "do something"
running 'echo' 'do something'
do something
Bisecting: 2 revisions left to test after this (roughly 1 step)
[7f85455a44353dbd579641f60c8a3327905bb87a] * add: LA64 trap and asm * refactor: Move MapArea into a separate file.
running 'echo' 'do something'
do something
Bisecting: 0 revisions left to test after this (roughly 1 step)
[0e825841c4c54b4168ae965feab6a0bc3868f58c] refactor: Refactor to compile user programs for LoongArch
running 'echo' 'do something'
do something
ba246b7c5294eeabcdca705fafe8ea1015e6fd6e is the first bad commit
commit ba246b7c5294eeabcdca705fafe8ea1015e6fd6e
Author: tempdragon <645703113@qq.com>
Date:   Fri Mar 31 19:58:43 2023 +0800

add: Expect script and clauses in Makefile for automation

os/Makefile          | 58 ++++++-----+
os/src/arch/la64/board/2k500.rs |  2 +-+
os/src/arch/la64/mod.rs    | 15 ++++++----+
os/src/arch/la64/sbi.rs    |  7 +----+
os/src/linker-2k500.ld    |  5 +---+
5 files changed, 58 insertions(+), 29 deletions(-)
bisection found first bad commit
$ git bisect next
ba246b7c5294eeabcdca705fafe8ea1015e6fd6e is the first bad commit
commit ba246b7c5294eeabcdca705fafe8ea1015e6fd6e
Author: tempdragon <645703113@qq.com>
Date:   Fri Mar 31 19:58:43 2023 +0800

add: Expect script and clauses in Makefile for automation

os/Makefile          | 58 ++++++-----+
os/src/arch/la64/board/2k500.rs |  2 +-+
os/src/arch/la64/mod.rs    | 15 ++++++----+
os/src/arch/la64/sbi.rs    |  7 +----+
os/src/linker-2k500.ld    |  5 +---+
5 files changed, 58 insertions(+), 29 deletions(-)
```

图 2-15 运行实例

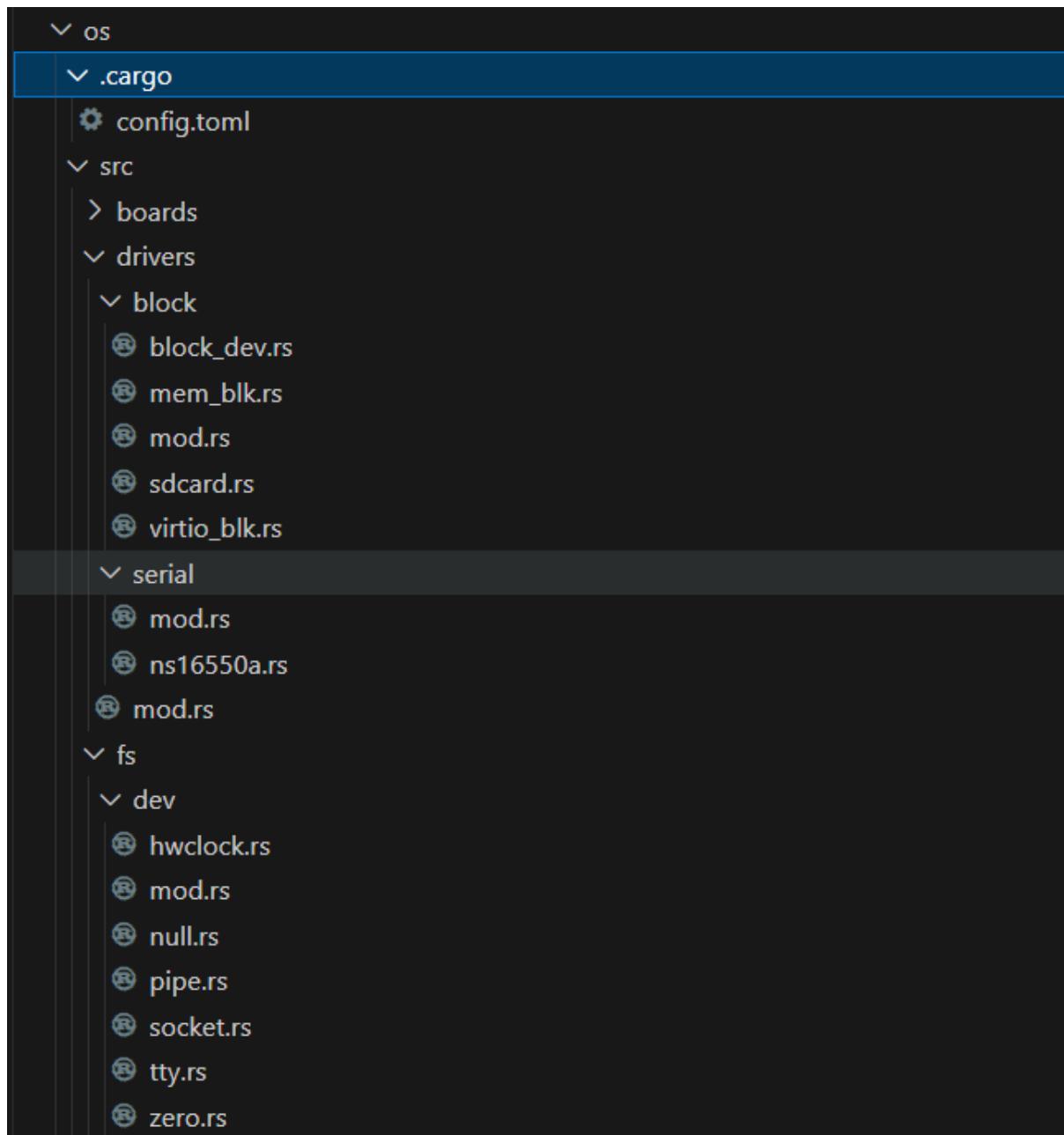


图 2-16 内核代码树 1

```
fat32
├── bitmap.rs
├── dir_iter.rs
├── efs.rs
├── inode.rs
├── layout.rs
├── mod.rs
├── vfs.rs
└── cache.rs
├── directory_tree.rs
└── file_trait.rs
├── filesystem.rs
├── layout.rs
└── mod.rs
├── poll.rs
└── swap.rs
└── mm
    ├── memory distribution
    │   └── memory distribution.drawio
    │   └── memory distribution.md
    │   └── memory distribution.png
    ├── address.rs
    ├── frame_allocator.rs
    ├── heap_allocator.rs
    ├── memory_set.rs
    ├── mod.rs
    ├── page_table.rs
    └── zram.rs
```

图 2-17 内核代码树 2

```
▽ syscall
  ⓘ errno.rs
  ⓘ fs.rs
  ⓘ mod.rs
  ⓘ process.rs
  ⓘ socket.rs
▽ task
  ⓘ context.rs
  ⓘ elf.rs
  ⓘ manager.rs
  ⓘ mod.rs
  ⓘ pid.rs
  ⓘ processor.rs
  ⓘ signal.rs
  ⓘ switch.rs
  ⚡ switch.S
  ⓘ task.rs
  ⓘ threads.rs
```

图 2-18 内核代码树 3

```
└─ trap
    └─ context.rs
    └─ mod.rs
    └─ trap.S
    └─ config.rs
    └─ console.rs
    └─ entry.asm
    └─ lang_items.rs
    └─ linker-fu740.ld
    └─ linker-k210.ld
    └─ linker-qemu.ld
    └─ main.rs
    └─ sbi.rs
    └─ timer.rs
    └─ build.rs
    └─ buildfs.sh
    └─ Cargo.toml
    └─ clear_doc.sh
    └─ Makefile
```

图 2-19 内核代码树 4



图 2-20 NPUcore 学习路线

## 第 3 章 编写第一个系统调用

什么是系统调用？在我们使用 C 语言编程时，使用过库函数提供的一些基本的函数，例如：控制台输出、文件读写。我们使用库函数完成基本的操作，库函数是对操作系统提供的系统调用的进一步封装，并隐藏掉了一些操作。系统调用工作在最底层，使用 POSIX 提供的接口，直接操作硬件，完成最基本的操作。

为什么需要借助于操作系统，用户不能直接操作硬件呢？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。用户库封装系统调用为库函数还有以下优点：

**1. 简化用户程序的编写：**通过封装系统调用，用户程序可以使用更为简单和直观的接口来完成复杂的系统操作，无需了解系统调用的底层细节，减少程序员的开发难度。

**2. 提高代码的可维护性：**通过库函数的封装，程序员可以对库函数进行多层封装，使程序的可读性、可维护性更高。当需要进行修改时，只需修改库函数的实现，无需修改应用程序的代码，降低了代码的耦合度，减少了代码维护的成本。

**3. 提高程序的移植性：**不同的操作系统和硬件平台实现系统调用的方式可能略有不同。使用库函数来封装系统调用可以提高程序的移植性。如果需要在不同操作系统下运行程序，只需更改库函数的实现即可。

**4. 方便进行错误处理：**库函数可以对系统调用返回值进行处理，根据返回值不同的情况进行错误处理。在使用系统调用时，程序员需要手动进行错误处理，使用库函数可以减轻程序员的工作量。

总之，封装系统调用为库函数可以使得程序更加简单、稳定、易维护、易移植，并且可以提高程序员的开发效率。

为了区分一个操作是用户完成的，还是依赖于操作系统完成的，每种指令集体体系结构都对此做出了区分。以 RISC-V 架构为例，CPU 的工作状态分为用户态、内核态等。执行用户程序指令时的状态为用户态，需要发起系统调用时，库函数中 `ecall` 指令会使 CPU 发生陷入提高特权级，到达内核态。内核态完成操作后，使用 `ret` 指令降低特权级，回到用户态。

本章将首先以三个系统调用为例子，讲解在用户态程序中如何使用系统调用，之后使用调试工具跟踪观察系统调用的实现，最后将对系统调用以及用户态、内核态等展开详细介绍。

### 3.1 使用系统调用

本节将以三个常见系统调用为例，简要介绍在用户态用户进程是如何使用系统调用的。

### 3.1.1 fork

fork 是一种用于克隆进程的全部内存空间的系统调用，是 Linux 系统中创建一个新进程的重要方法，除了第一个进程，所有的进程都是由 fork 创建的。

这是一个 Rust 语言编写的程序，主要目的是展示如何使用 fork 函数创建子进程，如??所示。

代码片段 3.1 forktest.rs

```

1 #![no_std]
2 #![no_main]
3
4 #[macro_use]
5 extern crate user_lib;
6
7 use user_lib::{exit, fork, wait};
8
9 const MAX_CHILD: usize = 30;
10
11 #[no_mangle]
12 pub fn main() -> i32 {
13     for i in 0..MAX_CHILD {
14         let pid = fork();
15         if pid == 0 {
16                 println!("I am child {}", i);
17                 exit(0);
18         } else {
19                 println!("forked child pid = {}", pid);
20         }
21         assert!(pid > 0);
22     }
23     let mut exit_code: i32 = 0;
24     for _ in 0..MAX_CHILD {
25         if wait(&mut exit_code) <= 0 {
26                 panic!("wait stopped early");
27         }
28     }
29     if wait(&mut exit_code) > 0 {
30             panic!("wait got too many");
31     }
32     println!("forktest pass.");
33     0
34 }
```

程序首先定义了一个常量 MAX\_CHILD，表示最大的子进程数量。然后程序进入一个循环，每次循环调用 fork 函数创建一个子进程，如果返回值为 0 则表示当前进程是子进程，打印一条信息并通过 exit 函数退出程序。如果返回值大于 0 则表示当前进程是父进程，打印一条信息并继续循环，创建下一个子进程。在父进程中通过 wait 函数等待所有子进程结束，并检查所有子进程的退出码是否为 0。

最后程序输出一条语句 “forktest pass.”，表示测试成功。

该程序主要目的是演示如何使用 fork 函数创建子进程，也在一定程度上展示了进程的并发和协作。每次调用 fork 函数都会创建一个新的进程，并且该进程和父进程是独

立的并发执行。使用 `wait` 函数等待子进程完成执行，防止子进程成为僵尸进程，并且可以获取子进程的退出码。通过这些操作，可以实现进程间的协作和并发执行。

在用户库定义了 `fork()` 函数的原型，使得用户可以在应用程序中调用这个函数。在 C 标准库中，`fork()` 函数的实现方式一般是通过调用一个名为 `_clone()` 的内部函数来完成的，该函数将 `fork()` 的输入参数转换为 `clone()` 系统调用所需要的参数，然后再进行调用。

代码片段 3.2 sys fork

```

1 pub fn sys_fork() -> isize {
2     const SIGCHLD: usize = 17;
3     syscall(SYSCLONE, [SIGCHLD, 0, 0])
4 }
```

在调用 `fork()` 函数时，用户程序通过系统调用将控制权转交给操作系统，在操作系统内部，会复制当前进程的资源到一个新的进程中，并将新进程的 PID 返回给用户。

此外，由于新进程是在当前进程的上下文中创建的，因此新进程会完全继承当前进程的代码、数据和堆栈等资源，但是它的状态是独立的，因此它可以有自己的寄存器、栈等状态。如果希望新进程运行不同于原进程的程序，则可以从 `fork()` 函数调用的返回处开始运行新的程序代码。

总之，`fork()` 函数封装了 `clone()` 系统调用原本极度复杂的进程状态复制过程，使得用户可以在应用程序中方便地创建新的进程。

### 3.1.2 exec

`exec` 系统调用是 Unix 和 Linux 操作系统中的一种常见的系统调用，它的作用是在当前进程的上下文中执行另一个程序。

`??` 是一个 Rust 语言编写的程序，主要目的是演示如何使用 `exec` 系统调用。

代码片段 3.3 exec test

```

1 #![no_std]
2 #![no_main]
3 use user_lib::{exit, exec, fork, wait, yield_};
4
5 #[no_mangle]
6 #[link_section = ".text.entry"]
7 pub extern "C" fn _start() -> ! {
8     exit(main());
9 }
10
11 #[no_mangle]
12 fn main() -> i32 {
13     let path = "/bin/bash\0";
14     let environ = [
15         "SHELL=/bash\0".as_ptr(),
16         "PWD=\0".as_ptr(),
17         "LOGNAME=root\0".as_ptr(),
18         "MOTD_SHOWN=pam\0".as_ptr(),
19         "HOME=/root\0".as_ptr(),
```

```
20     "LANG=C.UTF-8\0".as_ptr(),
21     "TERM=vt220\0".as_ptr(),
22     "USER=root\0".as_ptr(),
23     "SHLVL=0\0".as_ptr(),
24     "OLDPWD=/root\0".as_ptr(),
25     "PS1=\x1b[1m\x1b[32mNPUCore\x1b[0m:\x1b[1m\x1b[34m\\w\x1b[0m\$ \0"
26         .as_ptr(),
27     "/bin/bash\0".as_ptr(),
28     "PATH=/bin\0".as_ptr(),
29     "LD_LIBRARY_PATH=/\0".as_ptr(),
30     core::ptr::null(),
31 ];
32 if fork() == 0 {
33     exec(path, &[path.as_ptr() as *const u8, core::ptr::null()], &
34         environ);
35 } else {
36     loop {
37         let mut exit_code: i32 = 0;
38         let pid = wait(&mut exit_code);
39         // ECHLD is -10
40         if pid == -10 {
41             yield_();
42             continue;
43         }
44         user_lib::println!(
45             "[initproc] Released a zombie process, pid={}, exit_code={}"
46             ,
47             pid,
48             exit_code,
49         );
50     }
51 }
```

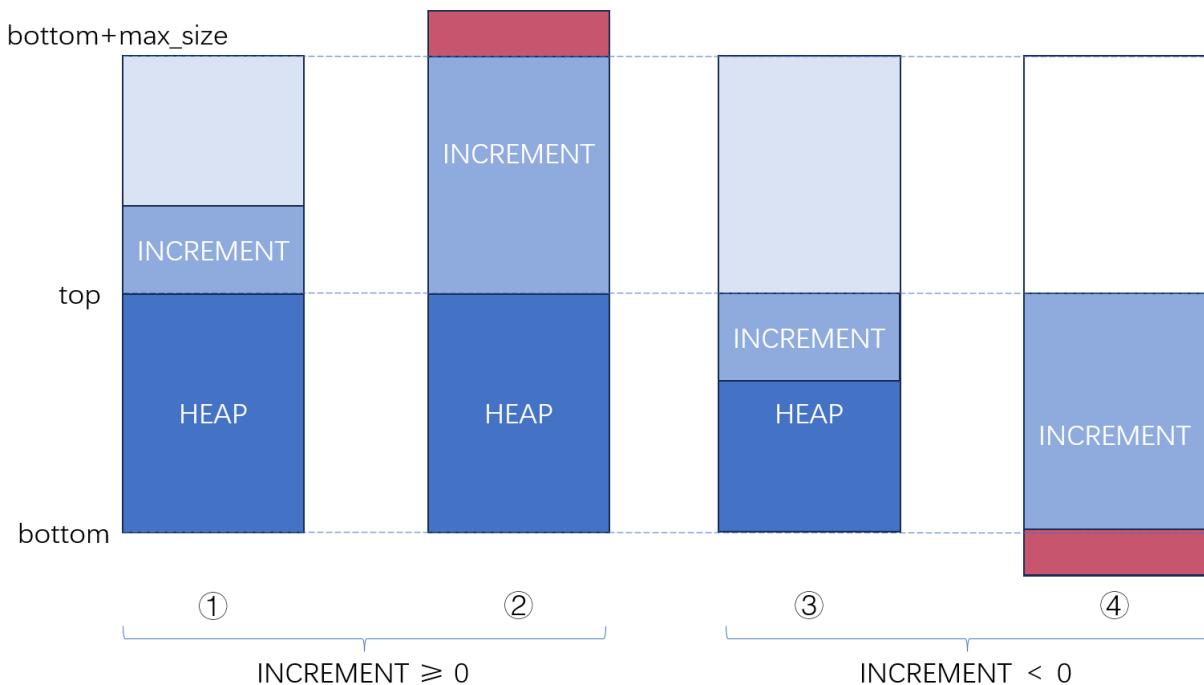
根据代码注释和函数调用，可以判断这个程序是一个操作系统的 init 进程，起到启动和监控子进程的作用。

程序首先调用了 `exit` 函数，该函数会在程序执行结束时将返回码返回给操作系统。然后程序定义了一个 `main` 函数，该函数会调用 `fork` 函数创建子进程并返回子进程的 `PID`。如果当前进程是子进程，则调用 `exec` 函数启动一个 `/bin/bash` 进程，并使用 `environ` 数组来设置进程的环境变量。如果当前进程是父进程，则进入一个死循环，调用 `wait` 函数等待子进程的退出并处理僵尸进程。

从代码中可以看出，`initproc` 的主要任务是启动和监控子进程，保证系统进程的正常运行。通过监控子进程，系统能够自动处理僵尸进程，释放系统资源。

### 3.1.3 sbrk

sbrk 函数的功能是改变进程的堆的大小，包括扩大与缩小。一个堆空间可以用堆底指针与堆顶指针来确定，而堆底指针一般是固定的，因此我们通过移动堆顶指针来实现堆空间大小的改变。如??，共有四种情况：

图 3-1 使用 `sbrk` 四种情况

前两种情况代表堆空间的扩大，后两种情况代表堆空间的缩小。有两种可能遇到的问题：一是堆空间扩大时超过用户空间大小限制；二是堆空间缩小时堆顶指针小于堆底指针。这两种情况都是不合理的请求时，我们来编写用户态程序测试一下，见??。

代码片段 3.4 `sbrk test`

```

1 user_lib::println!("old_heap_pt:{:08x}", sbrk(0));
2 user_lib::println!("increment:8192,      new_heap_pt:{:08x}", sbrk(8192));
3 user_lib::println!("increment:-4096,    new_heap_pt:{:08x}", sbrk(-4096));
4 user_lib::println!("increment:99999999,   new_heap_pt:{:08x}", sbrk
                  (99999999));
5 user_lib::println!("increment:-8192,    new_heap_pt:{:08x}", sbrk(-8192));

```

运行结果如??:

```

old_heap_pt:0000f000
increment:8192,      new_heap_pt:00011000
increment:-4096,    new_heap_pt:00010000
increment:99999999,   new_heap_pt:00010000
increment:-8192,    new_heap_pt:00010000

```

图 3-2 使用 `sbrk` 四种情况运行截图

进程的堆空间初始时堆顶指针为 `0x0f00`，语句二调用了 `sbrk` 函数，为堆空间增大了

8KiB 空间，因此堆顶指针为  $0x0f00+0x0200=0x1100$ 。同理，语句三缩小了堆空间，也能够正常执行。第三、四条语句展示了申请的堆空间过大/过小的情况，可见面对这两种错误，`sbrk` 函数选择不改变原来的堆空间。

### 3.2 利用 GDB 跟踪 `getpid` 系统调用

NPUcore 的系统调用是基于中断来实现的，大致会经历以下步骤，如??：（用户态，内核态由 CPU 特定寄存器中的几位来表示）

表 3-1 系统调用通用过程

内核态	用户态
	执行 <code>ecall</code>
硬件断点保存	
OS 手动断点保存	
中断处理	
中断返回，OS 手动断点恢复	
<code>ret</code> 硬件断点恢复	
	用户程序继续执行

下面请你自己动手，使用调试软件跟踪一遍系统调用。

启动调试后，在 `sys_getpid` 函数处打断点，使用命令 `b sys_getpid`，如??。

```
(gdb) b sys_getpid
Breakpoint 2 at 0x802419ac: file src/syscall/process.rs, line 264.
```

图 3-3 在 `sys_getpid` 函数处打断点

输入命令 `c`，运行程序至断点处，如??。

```
(gdb) c
Continuing.

Breakpoint 2, os::syscall::process::sys_getpid () at src/syscall/process.rs:264
264         let pid = current_task().unwrap().tgid;
```

图 3-4 运行程序至断点处

此时，程序会陷入内核，在 `sys_getpid` 函数处停下。内核的输出如??。

输入命令 `c`，继续让内核运行。因为在 `bash` 应用程序启动中会多次调用 `sys_getpid`，所以会多次停留在 `sys_getpid`，如??。

最终 `bash` 应用完全启动，界面如??。

### 3.3 系统调用机制与中断

中断是一种异步事件，它可以打断正在执行的程序并转移到处理中断的程序（中断处理程序）。中断可以来自外部设备（如硬件中断）或软件（如系统调用）。中断机制是操作系统用来响应和处理中断的一种机制，其中涉及中断向量表、中断控制器、中断处理程序等概念。

```
[kernel] Hello, world!
last 1336 Physical Frames.
.text [0x80200000, 0x8024a000)
.rodata [0x8024a000, 0x80256000)
.data [0x80256000, 0x80257000)
.bss [0x80257000, 0x804a8000)
mapping .text section
mapping .rodata section
mapping .data section
mapping .bss section
mapping physical memory
mapping memory-mapped registers
```

图 3-5 断点处内核输出

```
(gdb) c
Continuing.

Breakpoint 2, os::syscall::process::sys_getpid () at src/syscall/process.rs:264
264      let pid = current_task().unwrap().tgid;
(gdb) c
Continuing.

Breakpoint 2, os::syscall::process::sys_getpid () at src/syscall/process.rs:264
264      let pid = current_task().unwrap().tgid;
(gdb) c
Continuing.

Breakpoint 2, os::syscall::process::sys_getpid () at src/syscall/process.rs:264
264      let pid = current_task().unwrap().tgid;
(gdb) c
Continuing.
```

图 3-6 多次停在 sys\_getpid

系统调用是中断的一种，通常情况下，U态的中断包括了系统调用，系统调用陷入内核态后，将会调用 SBI call 来执行具体的内容。下面的??清晰地展示了这两种中断的区别及联系。

系统调用是应用程序向操作系统请求服务的一种机制。应用程序无法直接访问操作系统内核的代码和数据结构，因此需要通过系统调用请求操作系统的服务。例如，在 Linux 操作系统中，应用程序可以通过系统调用请求创建新的进程、读写文件、网络通信等操作。

同时，RISC-V 使用 SBI (Supervisor Binary Interface) 作为系统调用的接口，提供了一组标准的系统调用函数，包括控制台输出、内存分配、时钟等服务。

从系统调用以及中断的角度考虑，RISC-V 使用 TVEC (Trap Vector Base Address) 寄存器来指定中断向量表的基地址，中断向量表存储了中断处理程序的入口地址。当发生中断时，CPU 会自动跳转到相应的中断处理程序，并在处理完成后返回到中断前的指令位置。RISC-V 还提供了一些相关的指令和寄存器，用于中断的使能、屏蔽和处理。

??是 riscv 架构下对系统调用的处理流程。

从本节开始，我们将正式开始学习 NPUCore 对于系统调用的真正处理和实现。为了确保知识的合理递进，我们将其中最重要的内容分为了下面几个小节：

- Trap 及中断使能

```
[kernel] Hello, world!
last 1336 Physical Frames.
.text [0x80200000, 0x8024a000)
.rodata [0x8024a000, 0x80256000)
.data [0x80256000, 0x80257000)
.bss [0x80257000, 0x804a8000)
mapping .text section
mapping .rodata section
mapping .data section
mapping .bss section
mapping physical memory
mapping memory-mapped registers
NPUCore:/#
```

图 3-7 bash 界面

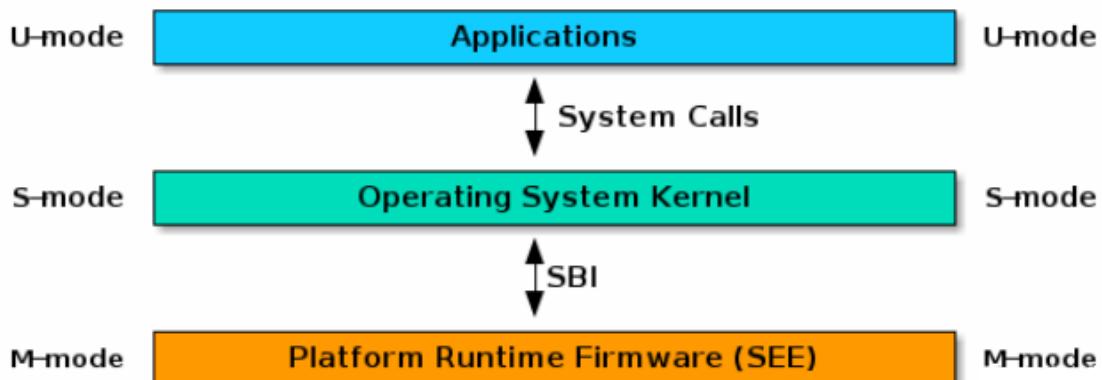


图 3-8 syscall 和 SBI 区别

- 系统调用与 ecall 指令
- 设置 stvec 寄存器及编写 trap\_handle 函数
- 利用汇编实现上下文保存与恢复
- RustSBI 简介及调用 RustSBI

### 3.3.1 Trap 及中断使能

#### (1) Trap 的概念及其作用

在上文中我们提到，系统调用是 trap 的一种，因此我们要了解系统调用，必须先了解 trap 是什么。

trap 的 3 种类型：

1. 主动的陷入：system call
2. 外设中断处理：鼠标、键盘响应
3. 运行时的意外：error、溢出、除 0 等

每个 RISC-V CPU 都有一组控制寄存器，内核通过向这些寄存器写入内容来告诉

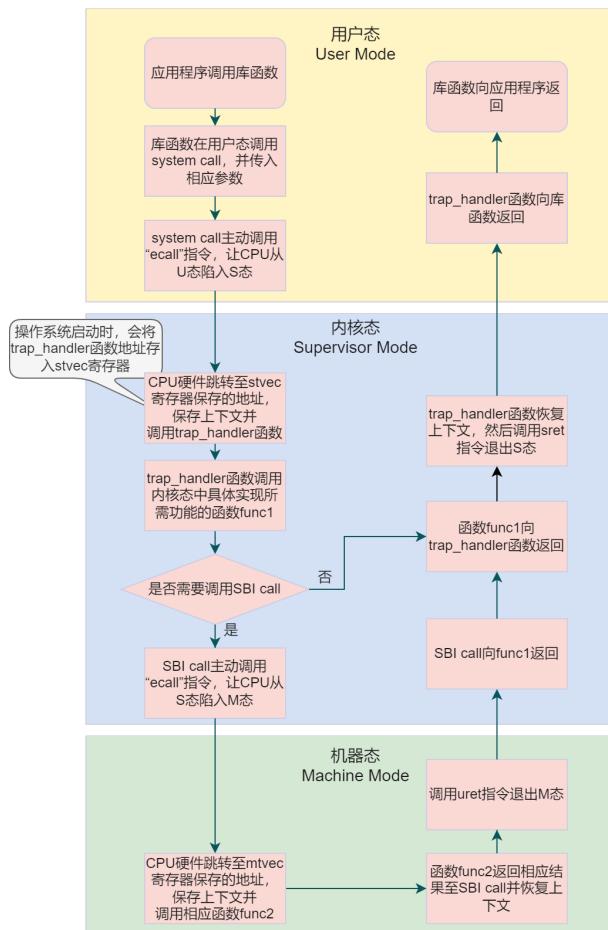


图 3-9 RISC-V 的系统调用处理流程图

CPU 如何处理陷阱，内核可以读取这些寄存器来明确已经发生的陷阱。RISC-V 文档包含了完整的内容。riscv.h(kernel/riscv.h:1) 包含在 NPUcore 中使用到的内容的定义。??是最重要的些寄存器概述：

## (2) 使能中断

使能中断指的是在 CPU 中打开中断处理的能力。如果中断被禁用，即使有中断请求发生，CPU 也不会执行中断处理程序，而是继续执行当前的任务，直到中断被启用为止。一旦中断被启用，当有中断请求发生时，CPU 会在适当的时候挂起当前的任务，跳转到中断处理程序，执行完毕后再返回到原来的任务。

在 RISC-V 架构中，使能中断可以通过设置 sstatus 寄存器下的 SIE 位来打开或关闭的中断使能。当相应的寄存器被置为 1 时，对应的中断将被使能。

在 NPUcore 中，为了确保中断与系统调用可用，我们利用 RustSBI 进行了如下的操作：

```

1 unsafe {
2     riscv::register::sstatus::set_sie();
3 }
```

该操作实际上是对 sstatus 寄存器的低 1 位赋值为 1，这样便打开了中断使能。

表 3-2 重要寄存器概述

寄存器名称	功能
stvec	内核在这里写入其陷阱处理程序的地址, RISC-V 跳转到这里处理陷阱
sepc	当发生陷阱时, RISC-V 会在这里保存程序计数器 pc (因为 pc 会被 stvec 覆盖)
sret	(从陷阱返回) 指令会将 sepc 复制到 pc, 内核可以写入 sepc 来控制 sret 的去向
scause	RISC-V 在这里放置一个描述陷阱原因的数字
sscratch	内核在这里放置了一个值, 这个值在陷阱处理程序一开始就会派上用场
sstatus	其中的 SIE 位控制设备中断是否启用。如果内核清空 SIE, RISC-V 将推迟设备中断, 直到内核重新设置 SIE。SPP 位指示陷阱是来自用户模式还是管理模式, 并控制 sret 返回的模式

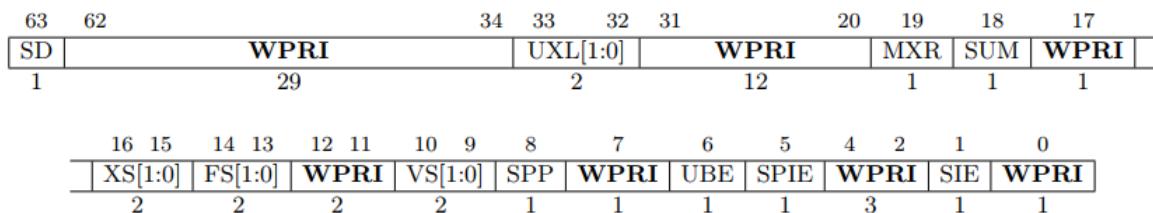


图 3-10 SXLEN=64 时 S-mode 下的状态寄存器 (sstatus)

### 3.3.2 系统调用与 ecall 指令

在 RISC-V 架构下, 系统调用是通过 ecall 指令来触发的。ecall 指令是一条特殊的指令, 当我们在 U 态执行 ecall 指令时, 会跳转到 STVEC 寄存器中函数地址指定的中断处理程序, 即系统调用处理程序 (若是在 S 态触发 ecall, 则会跳转到 mtvec 存储的函数地址), 进行相应的操作。ecall 指令一般用来触发一些特殊的操作, 比如系统调用、中断、异常等。

当我们在 U 态利用系统调用出发 ecall 指令后, 需要进行相应的操作, 比如将参数传递到内核态, 由内核态进行相应的操作, 然后再将结果返回给用户态。在系统调用发生时, 需要将当前用户态的上下文保存起来, 这样在系统调用完成后, CPU 就可以从保存的上下文中恢复用户态的执行状态。

接下来, 我们通过 NPUCore 具体的代码实例来学习如何利用 ecall 指令来完成系统调用。

我们以 write 系统调用为例, 来详细描述系统调用的具体流程。

#### (1) 用户态进程调用 syscall

- 应用程序调用位于 user/src/usr\_call.rs 中被包装好的 write 函数;
- write 函数调用位于 user/src/usr\_call.rs 中的 sys\_write 函数;

应用程序如何想使用操作系统为其提供的服务, 最直接的办法就是调用操作系统提

供的库函数。

NPUcore 的设计正是如此，我们将 syswrite 函数进行了一次包装，放在 user/src/usr\_call.rs 中，模拟库函数，见??。

我们对 sys\_write 传递两个参数，分别是文件描述符 fd 以及缓冲区指针 buf。

代码片段 3.5 usr\_call.rs:write

```
1 pub fn write(fd: usize, buf: &[u8]) -> isize {
2     sys_write(fd, buf)
3 }
```

- sys\_write 函数调用位于 user/src/usr\_call.rs 中的 syscall 函数；

见??，我们的包装类似于 lib 库函数，利用 sys\_write 函数来调用 syscall。

代码片段 3.6 usr\_call.rs:sys\_write

```
1 pub fn sys_write(fd: usize, buffer: &[u8]) -> isize {
2     syscall(SYS_WRITE, [fd, buffer.as_ptr() as usize, buffer.len()])
3 }
```

## (2) syscall 调用 ecall 指令陷入内核态

syscall 函数中内联的汇编函数执行了 ecall，陷入内核态。每一个 syscall 都有 4 个参数，第一个参数代表着系统调用的 id，NPUcore 的系统调用标识完全遵循 linux 标准，因此可以方便地兼容 linux 操作系统的应用程序。其余三个参数供每个系统调用进行灵活选择。

代码片段 3.7 syscall

```
1 fn syscall(id: usize, args: [usize; 3]) -> isize {
2     let mut ret: isize;
3     unsafe {
4         asm!(
5             "ecall",
6             inlateout("x10") args[0] => ret,
7             in("x11") args[1], // 分别把参数放进 a1、a2、a7
8             in("x12") args[2],
9             in("x17") id
10        );
11    }
12    ret
13 }
```

见??，在第五行，第一条指令就是 ecall 指令，我们在此调用 ecall 指令触发 trap，使得程序陷入内核态第六行的 inlateout 指令有两个作用：一是把 args[0] 放进 a0 寄存器；二是在程序返回后 x10 寄存器会作为返回值。接下来三行就是参数的传递。

至此，我们已经完成了系统调用在用户空间中应该进行的操作。

### 3.3.3 设置 stvec 寄存器与编写 trap\_handle 函数

在用户态进程调用 ecall 进入内核态后，CPU 将会自动进行如下操作：

- sstatus 的 SPP 字段会被修改为 CPU 当前的特权级（U/S）。

- sepc 会被修改为 Trap 处理完成后默认会执行的下一条指令的地址。
- scause/stval 分别会被修改成这次 Trap 的原因以及相关的附加信息。
- CPU 会跳转到 stvec 所设置的 Trap 处理入口地址，并将当前特权级设置为 S，然后从 Trap 处理入口地址处开始执行。

在这些操作中，我们最需要关心的是第四点。

尽管上文已经提到了 stvec 寄存器的具体功能，但仍然在这强调一下该寄存器的具体作用：在 RISC-V 架构中，中断控制器寄存器写入的是中断向量表的基址（Base Address）。stvec 寄存器用于存储中断向量表的基址，并指定了中断向量表的模式（Direct Mode 或 Vectored Mode）。当中断处理器接收到一个中断信号时，会根据 stvec 寄存器的值，自动跳转到中断向量表中对应的中断处理程序的入口地址。

因为硬件实现了自动跳转到 stvec 寄存器的功能，因此接下来我们只需要考虑如何往 stvec 寄存器中写入内容，以及写入什么内容。这两个问题的回答正是我们这一小节的主题——设置 stvec 寄存器及编写 trap\_handle 函数。

### (1) 设置 stvec 寄存器

我们应该向 stvec 写入什么呢？答案是显而易见的——中断处理函数。

其实，在操作系统初始化的时候，就已经修改了 stvec 寄存器来指向正确的 Trap 处理入口点，见??。

代码片段 3.8 set\_user\_trap\_entry

```

1 fn set_user_trap_entry() {
2     unsafe {
3         stvec::write(TRAMPOLINE as usize, TrapMode::Direct);
4     }
5 }
```

这里引入了一个外部符号 TRAMPOLINE（意为跳板），并将 stvec 设置为 Direct 模式指向它的地址。在 os/src/trap/trap.S 中实现 Trap 上下文保存/恢复的汇编代码，分别用外部符号 \_\_alltraps 和 \_\_restore 标记为函数，并通过 global\_asm! 宏将 trap.S 这段汇编代码插入进来。

为什么我们不直接将 \_\_alltraps 符号写入 stvec 寄存器，而是要建立一个跳板？使用跳板是为了解决这样一个问题：在开启分页模式之后 \_\_alltraps 的实际程序入口并不是一个特定值，而是取决于所处平台的不同（qemu/k210/u740）以及在编译器/汇编器/链接器进行后端代码生成和链接形成最终机器码时设置好的此指令的地址。

从上文中我们可以知道，在编写 trap.S 中的整段汇编代码时，我们将这段代码放置在 .text.trampoline 段。

该符号在 os/src/linker-k210.ld、os/src/linker-fu740.ld 及 os/src/qemu.ld 被连接，并在调整内存布局的时候将它对齐到代码段的一个页面中，见??。

代码片段 3.9 linker-xxx-.ld

```

1 stext = .;
2 .text : {
3     *(.text.entry)
4     . = ALIGN(4K);
5     strampoline = .;
6     *(.text_trampoline);
7     . = ALIGN(4K);
8     ssignaltrampoline = .;
9     KEEP(*(.text.signaltrampoline));
10    . = ALIGN(4K);
11    *(.text .text.*)
12 }

```

这样，这段汇编代码放在一个物理页帧中，且 `_alltraps` 恰好位于这个物理页帧的开头，其物理地址被外部符号 `strampoline` 标记。在开启分页模式之后，内核和应用代码都只能看到各自的虚拟地址空间，而在它们的视角中，这段汇编代码都被放在它们各自地址空间的最高虚拟页面上，由于这段汇编代码在执行的时候涉及到地址空间切换，故而被称为跳板页面。

这样就可以解释为何在 `_alltraps` 中需要借助寄存器 `jr` 而不能直接 `call trap_handler` 了。因为在内存布局中，这条 `.text_trampoline` 段中的跳转指令和 `trap_handler` 都在代码段之内，汇编器（Assembler）和链接器（Linker）会根据 `linker-qemu/k210.ld` 的地址布局描述，设定跳转指令的地址，并计算二者地址偏移量，让跳转指令的实际效果为当前 `pc` 自增这个偏移量。但实际上由于我们设计的缘故，这条跳转指令在被执行的时候，它的虚拟地址被操作系统内核设置在地址空间中的最高页面之内，所以加上这个偏移量并不能正确的得到 `trap_handler` 的入口地址。

在完成了这一步骤后，接下来将正式开始在 S 状态中的中断处理流程。

## (2) 编写 `trap_handler` 函数

在完成上文所述的设置 `stvec` 寄存器跳转地址后，我们将完成后续步骤：

1. 通过 `_alltraps` 将 Trap 上下文保存在内核栈上。
2. 跳转到使用 Rust 编写的 `trap_handler` 函数完成 Trap 分发及处理。
3. 当 `trap_handler` 返回之后，使用 `_restore` 从保存在内核栈上的 Trap 上下文恢复寄存器。
4. 最后通过一条 `sret` 指令回到应用程序执行。

本节重点讲解其中的第二点，追踪 NPUCore 中 `trap_handler` 函数的具体实现流程并了解其核心思想。由于该函数过长，这里只选取其中有关系统调用的部分进行讲解，见??。

代码片段 3.10 `trap_handler`

```

1 pub fn trap_handler() -> ! {
2     set_kernel_trap_entry();
3     //...
4     let scause = scause::read();

```

```

5  match scause.cause() {
6      Trap::Exception(Exception::UserEnvCall) => {
7          let mut cx = current_trap_cx();
8          cx_gp_pc += 4;
9          let result = syscall(
10              cx_gp_a7,
11              [cx_gp_a0, cx_gp_a1, cx_gp_a2, cx_gp_a3, cx_gp_a4, cx_gp_a5
12              ],
13          );
14          cx = current_trap_cx();
15          cx_gp_a0 = result as usize;
16      }
17 //...
18 trap_return();
}

```

在函数的开始，代码第 2 行，调用了 `set_kernel_trap_entry` 函数，在内核态的陷入是什么情况呢？报错！这个函数定义了如果在内核态再发生 Trap，会直接 panic 报错，是为了防止不正常的 S->S 状态 Trap 的发生。

在第 5 行的 `match` 匹配，这个处理系统调用的子块函数做了三件事：第一步，让 `pc+4`，是为了跳转后能正确执行下一条指令。第二步，把 `scause` 寄存器的值取出，调用 `os/src/syscall/mod.rs` 里面的 `syscall` 函数。第三步，将返回值存入返回值寄存器 `a0` 中。

为什么在用户空间中有一个 `syscall` 函数，在内核空间中又会出现一个同名的 `syscall` 函数呢？其实只有用户空间那个 `syscall` 才是我们平常意义上所说的系统调用，因为只有该 `syscall` 才调用了 `ecall` 指令，完成了 `trap` 操作。而在内核空间中出现的 `syscall` 可以认为是对系统调用所需要完成的不同功能进行分发处理。

内核空间中出现的 `syscall` 在 `os/src/syscall/mod.rs` 中。由于该函数过长，只选取其中少数几个的 `match` 项，见`??`。示例该函数只是起一个根据系统调用号对操作进行分发的作用。

代码片段 3.11 `syscall`

```

1 pub fn syscall(syscall_id: usize, args: [usize; 6]) -> isize {
2     //log
3     let ret = match syscall_id {
4         SYSCALL_GETCWD => sys_getcwd(args[0], args[1]),
5         SYSCALL_DUP => sys_dup(args[0]),
6         SYSCALL_DUP3 => sys_dup3(args[0], args[1], args[2] as u32),
7         SYSCALL_FCNTL => sys_fcntl(args[0], args[1] as u32, args[2]),
8         SYSCALL_IOCTL => sys_ioctl(args[0], args[1] as u32, args[2]),
9         //...still more
10    };
11    ret
12 }

```

最后，在 `trap_handler` 完成 `Trap` 处理之后，调用 `trap_return` 返回用户态，见`??`。

代码片段 3.12 `trap_return`

```

1 fn set_user_trap_entry() {

```

```

2 unsafe {
3     stvec::write(TRAMPOLINE as usize, TrapMode::Direct);
4 }
5 }
6 #[no_mangle]
7 pub fn trap_return() -> ! {
8     set_user_trap_entry();
9     let trap_cx_ptr = TRAP_CONTEXT;
10    let user_satp = current_user_token();
11    extern "C" {
12        fn __alltraps();
13        fn __restore();
14    }
15    let restore_va = __restore as usize - __alltraps as usize + TRAMPOLINE;
16    unsafe {
17        asm!(
18            "fence.i",
19            "jr {restore_va}",
20            restore_va = in(reg) restore_va,
21            in("a0") trap_cx_ptr,
22            in("a1") user_satp,
23            options(noreturn)
24        );
25    }
26    panic!("Unreachable in back_to_user!");
27 }
28 }

```

- 第 11 行，在 trap\_return 的开始处调用 set\_user\_trap\_entry，来让应用 Trap 到 S 的时候可以跳转到 \_\_alltraps。注意，需要把 stvec 设置为内核和应用地址空间共享的跳板页面的起始地址 TRAMPOLINE，而不是编译器在链接时看到的 \_\_alltraps 的地址。这是因为启用分页模式之后，内核只能通过跳板页面上的虚拟地址来实际取得 \_\_alltraps 和 \_\_restore 的汇编代码。
- 第 18 行，展示了计算 \_\_restore 虚地址的过程：由于 \_\_alltraps 是对齐到地址空间跳板页面的起始地址 TRAMPOLINE 上的，则 \_\_restore 的虚拟地址只需在 TRAMPOLINE 基础上加上 \_\_restore 相对于 \_\_alltraps 的偏移量即可。这里 \_\_alltraps 和 \_\_restore 都是指编译器在链接时看到的内核内存布局中的地址。
- 第 20-27 行，首先需要使用 fence.i 指令清空指令缓存 i-cache。因为在内核中进行的一些操作，可能导致一些原先存放某个应用代码的物理页帧，如今用来存放数据或者是其他应用的代码。i-cache 中可能还保存着该物理页帧的错误快照。因此我们直接将整个 i-cache 清空避免错误。接着使用 jr 指令完成了跳转到 \_\_restore 的任务。

### 3.3.4 利用汇编实现上下文保存与恢复

再回顾一下操作系统在 S 状态对于系统调用的处理会经历的步骤：

1. 通过 \_\_alltraps 将 Trap 上下文保存在内核栈上。
2. 跳转到使用 Rust 编写的 trap\_handler 函数完成 Trap 分发及处理。

3. 当 trap\_handler 返回之后，使用 \_\_restore 从保存在内核栈上的 Trap 上下文恢复寄存器。
4. 最后通过一条 sret 指令回到应用程序执行。

上一小节介绍了第二步，这一小节来介绍剩余三步，均是通过汇编代码来实现的，故先介绍一下 RISC-V 汇编的基础知识。

### (1) RISC-V 汇编基础知识

首先看一下 RISC-V 汇编常用指令，见??。其中 CSR 寄存器，即 Control and Status Register，控制与状态寄存器。

表 3-3 RISC-V 汇编常用指令

名称	格式	功能
sd	sd rs2, offset(rs1)	把寄存器 rs2 的值存入地址为寄存器 rs1 的值加 offset 的主存中
ld	ld rd, offset(rs1)	从地址为寄存器 rs1 的值加 offset 的主存中读取并存入 rd
csrr	csrr rd, csr	读取 CSR 寄存器的值
csrwr	csrwr csr, rs	写入 CSR 寄存器
csrrw	csrrw rd, csr, rs1	读取 CSR 的旧值，写入 rd 寄存器，同时将 rs1 的值写入 CSR
jr	jr rs	跳转到 rs 寄存器中的地址，并且不带返回值

为了更好的调用汇编指令，宏操作是一个很有效的手段。.MACRO 和.ENDM 伪指令可以用来组成一个宏。.MACRO 伪指令的格式如下

```
1 .macro macname macargs ...
```

.MACRO 伪指令后面依次是宏名称与宏的参数。在宏里使用参数，需要添加前缀\。

在正式编写函数前，声明了四个宏，作用分别是：保存整数寄存器，恢复整数寄存器，保存浮点寄存器，恢复浮点寄存器。见??。

代码片段 3.13 RISC-V 宏

```
1 .altmacro
2 .macro SAVE_GP n
3     sd x\n, \n*8(sp)
4 .endm
5 .macro LOAD_GP n
6     ld x\n, \n*8(sp)
7 .endm
8 .macro SAVE_FP n, m
9     fsd f\n, \m*8(sp)
10 .endm
11 .macro LOAD_FP n, m
12     fld f\n, \m*8(sp)
13 .endm
```

有了上述 RISC-V 汇编的基础知识，接下来正式开始讲解 \_\_alltraps 与 \_\_restore。

## (2) 保存上下文: \_\_alltraps

代码片段 3.14 \_\_alltraps

```

1  __alltraps:
2      csrrw sp, sscratch, sp
3      # now sp->*TrapContext in user space, sscratch->user stack
4      sd x1, 1*8(sp)
5      # skip sp(x2), we will save it later
6      .set n, 3
7      .rept 29
8          SAVE_GP %n
9          .set n, n+1
10     .endr
11     .set n, 0
12     .set m, FP_START
13     .rept 32
14         SAVE_FP %n, %m
15         .set n, n+1
16         .set m, m+1
17     .endr
18     # we can use t0/t1/t2 freely, because they have been saved in
19     # TrapContext
20     csrr t0, fcsr
21     sd t0, 64*8(sp)
22     # save other general purpose registers
23     sd a0, 65*8(sp)
24     csrr t0, sstatus
25     csrr t1, sepc
26     sd t0, 66*8(sp)
27     sd t1, 0(sp)
28     # read user stack from sscratch and save it in TrapContext
29     csrr t2, sscratch
30     sd t2, 2*8(sp)
31     # load kernel_satp into t0
32     ld t0, 67*8(sp)
33     # load trap_handler into t1
34     ld t1, 68*8(sp)
35     # move to kernel_sp
36     ld sp, 69*8(sp)
37     # switch to kernel space
38     csrw satp, t0
39     sfence.vma
40     # jump to trap_handler
41     jr t1

```

第 2 行的 csrrw 原型是 csrrw rd, csr, rs' 可以将 CSR 当前的值读到通用寄存器 rd 中，然后将通用寄存器 rs 的值写入该 CSR。因此这里起到的是交换 sscratch 和 sp 的效果。在这一行之前 sp 指向用户栈，sscratch 指向内核栈，现在 sp 指向内核栈，sscratch 指向用户栈。

第 4 行，保存 Trap 上下文的通用寄存器 x1。

在这里不保存 sp(x2)，因为要基于它来找到每个寄存器应该被保存到的正确的位置。实际上，在栈帧分配之后，我们可用于保存 Trap 上下文的地址区间为 [sp,sp+8×34)

，按照 TrapContext 结构体的内存布局，基于内核栈的位置（sp 所指地址）来从低地址到高地址分别按顺序放置 x0-x31 这些通用寄存器，最后是 sstatus 和 sepc。

第 6-10 行，为了简化代码，x3 x31 这 29 个通用寄存器通过类似循环的.rept 每次使用 SAVE\_GP 宏来保存，其实质是相同的。注意我们需要在 trap.S 开头加上.altmacro 才能正常使用.rept 命令。

第 11-17 行，用类似的方法，保存浮点寄存器。

第 23-26 行，将 CSR sstatus 和 sepc 的值分别读到寄存器 t0 和 t1 中然后保存到内核栈对应的位置上。指令 csrr rd, csr 的功能就是将 CSR 的值读到寄存器 rd 中。这里不用担心 t0 和 t1 被覆盖，因为它们刚刚已经被保存了。

第 28-29 行专门处理 sp 的问题。首先将 sscratch 的值读到寄存器 t2 并保存到内核栈上，注意：sscratch 的值是进入 Trap 之前的 sp 的值，指向用户栈。而现在的 sp 则指向内核栈。

31-40 行，切换到内核地址空间并跳转到 trap handler。

- 第 31 行将内核地址空间的 token 载入到 t0 寄存器中；
  - 第 33 行将 trap handler 入口点的虚拟地址载入到 t1 寄存器中；
  - 第 35 行直接将 sp 修改为应用内核栈顶的地址；
- 注：这三条信息均是内核在初始化该应用的时候就已经设置好的。
- 第 37 38 行将 satp 修改为内核地址空间的 token 并使用 sfence.vma 刷新快表，这就切换到了内核地址空间；
  - 第 40 行最后通过 jr 指令跳转到 t1 寄存器所保存的 trap handler 入口点的地址。

### (3) 恢复上下文：\_\_restore

代码片段 3.15 \_\_restore

```

1 __restore:
2     # a0: *TrapContext in user space(Constant); a1: user space token
3     # switch to user space
4     csrw satp, a1
5     sfence.vma
6     csrw sscratch, a0
7     mv sp, a0
8     # now sp points to TrapContext in user space, start restoring based on
9     # it
10    # restore sstatus/sepc
11    ld t0, 32*8(sp)
12    ld t1, 33*8(sp)
13    csrw sstatus, t0
14    csrw sepc, t1
15    # restore general purpose registers except x0/sp/tp
16    ld x1, 1*8(sp)
17    ld x3, 3*8(sp)
18    .set n, 5
19    .rept 27
20        LOAD_GP %n
        .set n, n+1

```

```

21 .endr
22 # back to user stack
23 ld sp, 2*8(sp)
24 sret

```

当内核将 Trap 处理完毕准备返回用户态的时候会调用 `_restore`，它有两个参数：第一个是 Trap 上下文在应用地址空间中的位置，这个对于所有的应用来说都是相同的，在 `a0` 寄存器中传递；第二个则是即将回到的应用的地址空间的 `token`，在 `a1` 寄存器中传递。

第 45 行先切换回应用地址空间（注：Trap 上下文是保存在应用地址空间中）；

第 6 行将传入的 Trap 上下文位置保存在 `sscratch` 寄存器中，这样 `_alltraps` 中才能基于它将 Trap 上下文保存到正确的位置；

第 7 行将 `sp` 修改为 Trap 上下文的位置，后面基于它恢复各通用寄存器和 CSR；

第 24 行最后通过 `sret` 指令返回用户态。完成了一次完整的系统调用。

### 3.3.5 RustSBI 简介与调用 RustSBI

#### (1) RustSBI 简介

经过系统调用陷入 S 态后，如果 S 态的内核程序无法独立地完成 U 态应用程序所需的功能，则它必须要再调用 SBI call，以请求 M 态程序 SBI 的服务。

RISC-V 指令集的 SBI 标准规定了类 Unix 操作系统之下的运行环境规范。这个规范拥有多种实现，RustSBI 是它的一种实现。

RISC-V 架构中，存在着定义于操作系统之下的运行环境。这个运行环境不仅将引导启动 RISC-V 下的操作系统，还将常驻后台，为操作系统提供一系列二进制接口，以便其获取和操作硬件信息。RISC-V 给出了此类环境和二进制接口的规范，称为“操作系统二进制接口”，即“SBI”。SBI 的实现是在 M 模式下运行的特定于平台的固件，它将管理 S、U 等特权上的程序或通用的操作系统。

RustSBI 是 RISC-V 下 SBI 标准的实现，旨在为裸机平台、虚拟化和模拟器软件提供良好的 SBI 接口支持。它有机结合了 Rust 嵌入式生态与 RISC-V 系统软件，加快开发速度的同时，保证 Rust 语言具备的良好安全性和运行性能。本次 0.3.0 版本主要包括增加了实例化的 SBI 接口支持及相关的构造器结构，可以在 stable Rust 编译，去除了对堆内存和全局变量的依赖，完善了相关文档，以及若干的小修复。0.3.0 版本更新将为 Rust 编写的 RISC-V 虚拟化软件和 RISC-V 模拟器提供良好的支持，并进一步完善裸机 RISC-V 开发的实用性，可以启动 Linux 等在内的成熟操作系统和 zCore 等在内的科研操作系统。

若想了解更多有关 RustSBI 的知识和具体实现，请前往开发者洛佳的 github 仓库：  
<https://github.com/rustsbi/rustsbi>

## (2) 在 NPUcore 中调用 RustSBI

与我们在 U 态使用 ecall 指令调用 syscall 并陷入内核态那样，我们在内核态同样也可以通过 ecall 指令调用 RustSBI 并进一步陷入 M 态。

正如我们会给 syscall 进行系统调用编号那样，我们也可以通过寄存器传递给 OpenSBI 一个“调用编号”。

SBI 的规范中为 SBI call 定义了 9 个编号，见??

代码片段 3.16 SBI call 调用编号

```

1 const SBI_SET_TIMER: usize = 0;
2 const SBI_CONSOLE_putchar: usize = 1;
3 const SBI_CONSOLE_getchar: usize = 2;
4 const SBI_CLEAR_IPI: usize = 3;
5 const SBI_SEND_IPI: usize = 4;
6 const SBI_REMOTE_FENCE_I: usize = 5;
7 const SBI_REMOTE_SFENCE_VMA: usize = 6;
8 const SBI_REMOTE_SFENCE_VMA_ASID: usize = 7;
9 const SBI_SHUTDOWN: usize = 8;

```

在 S 态调用 SBI call 的方法与 U 态调用 syscall 的方法类似，通过??来具体观察。

代码片段 3.17 syscall 与 SBI call 对比

```

1 fn syscall(id: usize, args: [usize; 3]) -> isize {
2     let mut ret: isize;
3     unsafe {
4         asm!(
5             "ecall",
6             inlateout("x10") args[0] => ret,
7             in("x11") args[1],
8             in("x12") args[2],
9             in("x17") id
10        );
11    }
12    ret
13 }
14 fn sbi_call(which: usize, arg0: usize, arg1: usize, arg2: usize) -> usize {
15     let mut ret;
16     unsafe {
17         asm!(
18             "ecall",
19             inlateout("x10") arg0 => ret,
20             in("x11") arg1,
21             in("x12") arg2,
22             in("x17") which,
23        );
24    }
25    ret
26 }

```

可以发现二者几乎相同。这个现象引出来一个问题：既然我们在 U 态进行系统调用时，保存与恢复用户态寄存器上下文的内容是由内核程序完成的（见上述章节），那么我们现在在内核态进行 SBI call，保存与恢复内核态寄存器的上下文是不是应该由 SBI

程序完成呢？

查看 SBI 的实现规范文档得知，以上猜测是正确的。此外，与内核程序中需要设置 stvec 寄存器以正确跳转至 trap\_handler 函数一样，RustSBI 也同样设置了 mtvec 寄存器。

因此，在编写系统调用乃至整个 NPUCore 的过程中，由于 RustSBI 的存在，机器级硬件的操控基本被屏蔽了，可以把心思专心放在内核态程序的编写中来。

### 3.4 实验

## 第4章 内存管理

### 4.1 RISC-V 页表硬件

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。

在分页内存管理中，操作系统通过“页表”来实现虚实内存映射机制，这是我们本章介绍的重点内容。同时，我们也可以使用页表来实现许多“有趣”的功能，例如将不同的地址空间映射至同一块物理内存空间，以实现共享内存；或是使用未映射的页面来保护内核和用户栈等等。

#### 4.1.1 虚拟地址与物理地址

##### (1) 地址的格式及关系

如之前提到，实现虚拟地址到物理地址的映射，也就是页表的实现是我们本章的重点。不过在具体介绍页表之前，我们先来介绍我们所要维护的对象——虚拟地址与物理地址。

在 NpuCore 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)。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。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)。因此，地址的格式如图 4-1 所示：

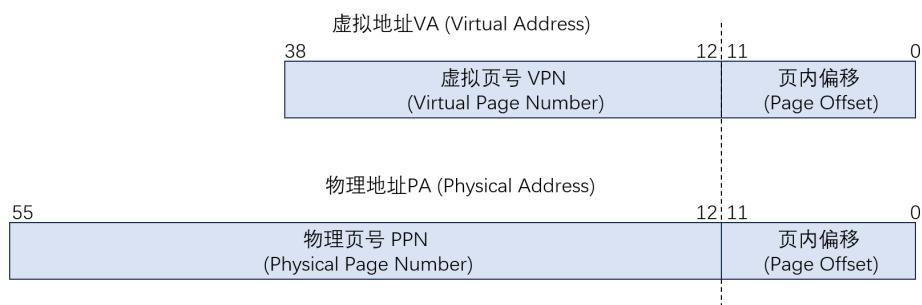


图 4-1 虚拟地址与物理地址的格式

回到我们的重点——地址转换。页式管理下，地址的转换是以页为单位进行的。也就是说，地址转换前后地址的页内偏移部分是不变的。因此，我们实际上要完成的操作是：从虚拟地址中取出 27 位虚拟页号，在页表中查询其对应的物理页号（若存在），最后将这 27 位虚拟页号映射得到的 44 位的物理页号与虚拟地址的 12 位页内偏移按序拼接到一起，就得到了 56 位的物理地址，完成了地址转换的过程。

## (2) 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 4.1 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)]
11 #[derive(Copy, Clone, Ord, PartialOrd, Eq, PartialEq)]
12 pub struct PhysPageNum(pub usize);
13
14 #[repr(C)]
15 #[derive(Copy, Clone, Ord, PartialOrd, Eq, PartialEq)]
16 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 4.2 os/src/mm/address.rs

```

1 impl From<usize> for PhysPageNum {
2     fn from(v: usize) -> Self {
3         Self(v)
4     }

```

5 }

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 4.3 os/src/mm/address.rs

```
1 impl From<PhysPageNum> for usize {
2     fn from(v: PhysPageNum) -> Self {
3         v.0
4     }
5 }
```

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。接下来以物理地址与物理页号的转换为例：

首先介绍地址的取整：

代码片段 4.4 os/src/mm/address.rs

```
1 impl PhysAddr {
2     pub fn floor(&self) -> PhysPageNum {
3         PhysPageNum(self.0 / PAGE_SIZE)
4     }
5     pub fn ceil(&self) -> PhysPageNum {
6         PhysPageNum((self.0 + PAGE_SIZE - 1) / PAGE_SIZE)
7     }
8 }
```

floor 为下取整方法，而 ceil 为上取整方法。其中，PAGE\_SIZE 为 4096，表示每个页面的大小。

接下来介绍地址与页号的转换，我们同样是实现 From<T> trait：

代码片段 4.5 os/src/mm/address.rs

```
1 impl PhysAddr {
2     pub fn page_offset(&self) -> usize { self.0 & (PAGE_SIZE - 1) }
3 }
4
5 impl From<PhysAddr> for PhysPageNum {
6     fn from(v: PhysAddr) -> Self {
7         assert_eq!(v.page_offset(), 0);
8         v.floor()
9     }
10 }
11
12 impl From<PhysPageNum> for PhysAddr {
13     fn from(v: PhysPageNum) -> Self { Self(v.0 << PAGE_SIZE_BITS) }
14 }
```

其中，PAGE\_SIZE\_BITS 为 12，表示页内偏移的位宽。

至此，我们对地址信息类型的实现已经有了基本的掌握。

#### 4.1.2 页表项

##### (1) 页表项格式及含义

在上面的内容中其实我们已经认识到，虚实地址转换的流程核心就是：使用虚拟页号作为索引在页表中查询到对应的物理页号。若把页表比作一个货架，则虚拟页号就是商品的编号，我们通过这个编号寻找到对应的商品，而这个商品就存储着我们需要的物理地址信息。“这个商品”指的就是页表项。

SV39 分页模式下，页表项是一个 8 字节的比特序列，其结构如图 4-2 所示：

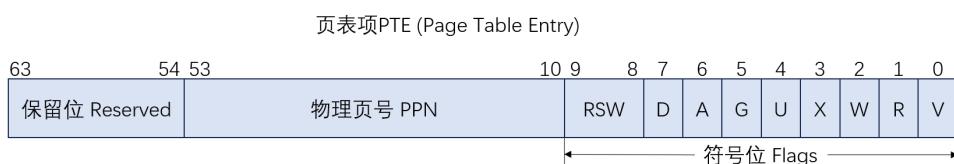


图 4-2 页表项的格式

可见，[63:54] 这 10 位是保留位，被忽略；[53:10] 这 44 位是物理页号；而最低的 10 位 [9:0] 则是标志位。标志位实际上控制了应用对其地址空间中每个虚拟页面的访问权限，他们的具体含义如下：

- 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；
- 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。

总之，页表项不仅存储了物理地址信息，还存储了一组标志位用于对虚拟页面的权限控制。

##### (2) 页表项的数据结构抽象

首先我们对标志位使用 bitflags! 宏进行包装：

代码片段 4.6 os/src/mm/page\_table.rs

```

1 use bitflags::*;
2 bitflags! {
3     pub struct PTEFlags: u8 {
4         const V = 1 << 0;
5         const R = 1 << 1;
6         const W = 1 << 2;
7         const X = 1 << 3;
8         const U = 1 << 4;
9         const G = 1 << 5;
10        const A = 1 << 6;
11        const D = 1 << 7;
12    }
13 }

```

可见，我们将一个 `u8` 类型封装成了一个标志位的集合类型 `PTEFlags`，使其支持一些常见的集合运算，且使用时易于理解。

接下来我们实现页表项 `PageTableEntry`：

代码片段 4.7 os/src/mm/page\_table.rs

```

1 #[derive(Copy, Clone)]
2 #[repr(C)]
3 pub struct PageTableEntry {
4     pub bits: usize,
5 }
6
7 impl PageTableEntry {
8     pub fn new(ppn: PhysPageNum, flags: PTEFlags) -> Self {
9         PageTableEntry {
10             bits: ppn.0 << 10 | flags.bits as usize,
11         }
12     }
13 }

```

可见，`PageTableEntry` 类型实际上也是对 `usize` 类型的一层简单包装。`new` 方法使得我们可以从一个 `PhysPageNum` 类型的物理页号和一个 `PTEFlags` 类型的页表项标志位生成一个页表项实例。当然，我们也提供了一些简单的方法，用于取出或直接使用页表项中的信息，例如：

代码片段 4.8 os/src/mm/page\_table.rs

```

1 impl PageTableEntry {
2     pub fn ppn(&self) -> PhysPageNum {
3         (self.bits >> 10 & ((1usize << 44) - 1)).into()
4     }
5     pub fn flags(&self) -> PTEFlags {
6         PTEFlags::from_bits(self.bits as u8).unwrap()
7     }
8     pub fn is_valid(&self) -> bool {
9         (self.flags() & PTEFlags::V) != PTEFlags::empty()
10    }
11 }

```

前两个方法与 new 方法相对，可以从一个页表项实例中取出物理页号或标志位信息。最后一个方法可以快速判断当前页表项是否合法。当然，还有许多相似的辅助函数在此没有介绍。

至此，我们对页表项的结构以及使用也有了一个基本的了解，下面我们终于可以介绍页表了。

#### 4.1.3 页表

##### (1) SV39 三级页表结构

在上一节中，我们将页表比作了一个货架，而货架上装的是页表项 PTE 这一商品。事实上，页表的确可以看作是一组页表项的集合，且这种集合的组织形式是最简单的线性表，当然这是对于一个页表而言。实际上，如果我们只维护一张页表来将高达 512GiB 的虚拟地址空间进行映射，页表本身就会变得相当庞大，远超出我们的实际物理内存。因此，SV39 模式使用了三级页表结构，将原本一张巨大的页表拆分成许许多多的小页表，再将这些页表通过树状结构组织起来，以此提高信息的利用率，从而节省空间。注意：这里的树状结构指的是页表与页表之间的关系，而一个页表本身仍为一份线性表，里面存储了若干页表项。接下来我们将详细介绍这一结构，介绍完结构后，我们会阐明这么做的原因。

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  $4\text{KiB}/8\text{B}=512$  个页表项，这些页表项线性排列在页表内，如图 4-3：

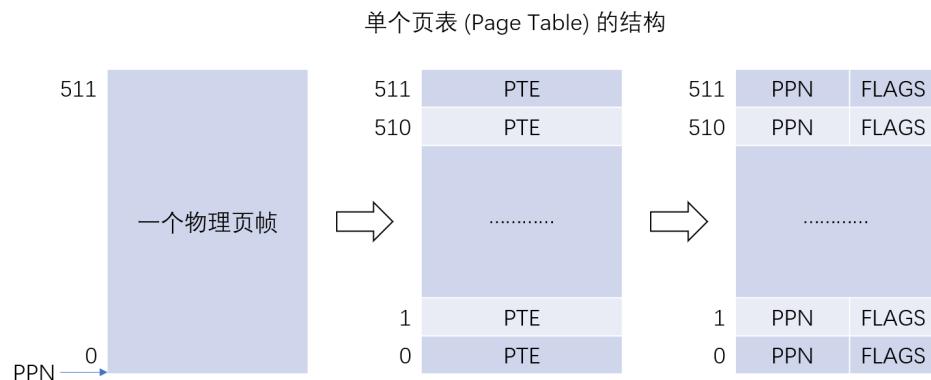


图 4-3 单个页表的结构

上图逐层对页表的结构进行了刻画。请留意这张图，我们在对页表进行数据结构抽象时还会用到。接下来我们来介绍页表间的树状结构：

我们多次强调，树状结构指的是页表间的结构，也就是说，我们要将一个页表视为一个节点，并将这些节点以树状形式相连。换句话说，我们要从一个页表出发，对应到若干个子页表，这要怎么做到呢？

回想两个已知的事实：①一张页表恰好位于一个物理页帧上，而一个物理页帧由一

一个物理页号标识。换句话说，一张页表由其所在物理页帧的物理页号唯一标识。②页表中存储的是若干条 PTE，而每个 PTE 存储着一个物理页号以及若干标志位。

至此，方法已水到渠成：我们可以使用 PTE 来记录页表的位置，从而实现页表至页表间的关联。在上一节中我们说，PTE 存储的是虚拟页号所对应的物理页号。而现在我们要将一部分 PTE 进行改造，使其存储的物理页号信息不再与虚拟页号直接对应，而是与页表所在的物理页帧对应。注意这种改造并非改变 PTE 的结构，仅是改变 PTE 的逻辑含义，改造方法将会在本小节最后给出。

因此，现在我们具有两种不同的 PTE：一种 PTE 存储的是我们最终需要的、虚拟页号所直接对应的物理页号，这种 PTE 存储于位于叶子节点的页表中；第二种 PTE 是经过我们改造的，存储指向一个页表的物理页号，这种 PTE 存储于非叶子节点的页表中。至此，我们已经构建起了页表与页表间的联系，将他们按树状结构组织，如图 4-4：

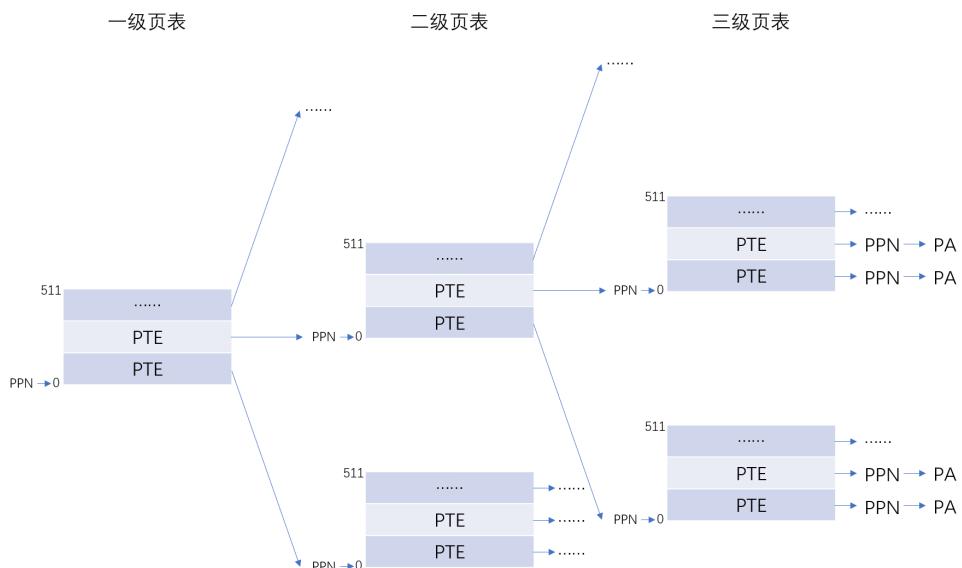


图 4-4 三级页表结构

如图，SV39 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。

下面我们来谈谈这种页表组织形式的好处：

前面我们提到，之所以不维护一个记录全部映射关系的页表，是因为它保存了所有虚拟页号对应的页表项，远超我们内存空间的上限。实际上，高达 512GiB 的虚拟地址空间真正会被使用到的只是其中极小的一部分，也就是说这种页表的绝大部分空间都是被浪费掉的。因此，我们需要“按需分配”，使得页表中存储的是真正有意义的、会被使用到的页表项。

一开始每个应用的地址空间都是空的，此时的页表也应为空。而当内核决定好了一个应用的各逻辑段存放位置后，MMU 就从零开始，以虚拟页面为单位来让该应用的地

址空间的某些部分与实际物理内存空间相对应，反映在本应用的页表上就是一对对映射顺次被插入进来。

因此，我们需要一个动态的、支持页表所占据的内存大小随映射数量增加而增加的页表结构，最终我们选择了树状结构。实际操作中，每个应用最初只有一个一级页表，仅占据一个物理页帧大小。而当映射开始插入后，页表从根节点开始“发芽抽枝”，逐步增加二级页表与三级页表，增加的每个页表也仅占一个物理页帧空间。就这样，这种三级页表结构以低粒度的形式保证了页表大小的动态变化，解决了单一页表占据空间过大的问题。

最后，我们解释如何赋予 PTE 两种不同的含义：存储的 PPN 指向一个页表或是指向我们最终所需的物理页号。其实这十分容易，我们正是通过操作 PTE 的符号位来实现这一区分，在此我们直接给出 SV39 中 PTE 符号位 R/W/X 组合的含义，通过这三个符号的组合我们可以了解该 PTE 的具体属性：

表 4-1 PTE 的 R/W/X 符号位编码

X	W	R	含义
0	0	0	本 PTE 存储的 PPN 指向下一级页表
0	0	1	指向本 PTE 的虚拟页面为只读页
0	1	0	保留，暂无用
0	1	1	指向本 PTE 的虚拟页面为可读写页
1	0	0	指向本 PTE 的虚拟页面为只可执行页
1	0	1	指向本 PTE 的虚拟页面为可读可执行页
1	1	0	保留，暂无用
1	1	1	指向本 PTE 的虚拟页面为可读写可执行页

注意，上表成立的前提是 PTE 的 V (Valid) 标志位为 1。当 V 为 0 的时候，代表当前 PTE 是无效的。

## (2) 页表的数据结构抽象

我们在之前已经提到，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 4.9 os/src/mm/page\_table.rs

```

1 pub struct PageTable {
2     root_ppn: PhysPageNum,
3     frames: Vec<FrameTracker>,
4 }
5
6 impl PageTable {
7     pub fn new() -> Self {
8         let frame = frame_alloc().unwrap();
9         PageTable {
10             root_ppn: frame.ppn,
11             frames: vec![frame],
12         }
13     }
14 }
```

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，其具体功能实现我们将在物理页帧管理模块再继续介绍。

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`。那么，我们怎么通过页表来取得其存储的 PTE 呢？需要用到以下方法：

代码片段 4.10 os/src/mm/page\_table.rs

```

1 impl PhysPageNum {
2     pub fn get_pte_array(&self) -> &'static mut [PageTableEntry] {
3         let pa: PhysAddr = self.clone().into();
4         unsafe {
5             core::slice::from_raw_parts_mut(pa.0 as *mut PageTableEntry,
6                                             512)
7         }
8     }

```

该方法构造可变引用来直接访问一个物理页号对应的物理页帧，然后正如图 4-3 的第一个箭头那样，将页表所在的物理页帧切分成 512 份，每一份正是对应一项 PTE，最终该方法返回一个页表项类型的定长数组（长度为 512）的可变引用，代表了多级页表中的一个节点。至此，我们也实现了从一个页表获取其页表项的方法。万事俱备，现在我们可以介绍虚实地址的映射过程了。

#### 4.1.4 虚实地址的映射过程

##### (1) satp 寄存器

首先我们介绍一个寄存器——satp 寄存器。

该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 RISC-V 64 架构下 satp 的字段分布，如图 4-5：



图 4-5 satp 寄存器的字段结构

各字段含义如下：

- MODE: 控制 CPU 使用何种分页模式。当 MODE 设置为 0 的时候，代表所有访存都被视为物理地址；而设置为 8 的时候，SV39 分页机制被启用；
- ASID: 表示地址空间标识符，与进程管理有关，此处先不介绍；
- PPN: 该 PPN 是当前地址空间的根页表所在的物理页号。这样，给定一个虚拟页号，CPU 就可以从三级页表的根页表开始一步步的将其映射到一个物理页号，也就是说，页表的使用入口存储于此。

## (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。看到这里读者是否想起些什么？没错，我们每个页表所存储的 PTE 数组长度正好是 512，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：

- ① 通过 satp 中的 PPN 字段找到一级页表（根页表）；
- ② 在一级页表中，将  $VPN_1$  作为索引查询到对应的 PTE，通过该 PTE 中的 PPN 字段找到二级页表；
- ③ 在二级页表中，将  $VPN_2$  作为索引查询到对应的 PTE，通过该 PTE 中的 PPN 字段找到三级页表；
- ④ 在三级页表中，将  $VPN_3$  作为索引查询到对应的 PTE，通过该 PTE 中的 PPN 字段找到本虚拟页号所对应的物理页号；
- ⑤ 将得到的物理页号与本虚拟地址的低 12 位偏移量拼凑在一起，即获得了最终对应的物理地址。

该流程的图示如图 4-6:

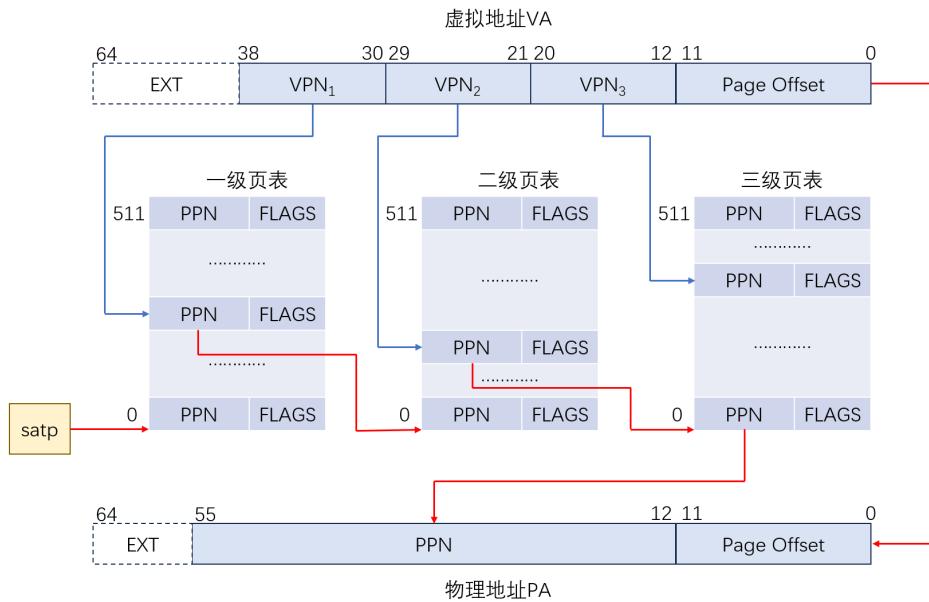


图 4-6 虚实地址转换流程

### (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 4.11 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 4.12 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        }
13    }
14    result
15 }
```

```

12         break;
13     }
14     pte = 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 4.13 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，我们按照图 4-6 完成了虚实地址转换的实现。

#### 4.1.5 TLB

我们知道，物理内存的访问速度要比 CPU 的运行速度慢很多。如果我们按照上面介绍的三级页表机制循规蹈矩地查询，将一个虚拟地址转化为物理地址需要访问 3 次物理内存，得到物理地址后还需要再访问一次物理内存，才能完成全部访存操作，这无疑很大程度上降低了系统执行效率。

我们在计算机组成原理课程中学习过 Cache 的设计理念，即利用地址访问过程的时间局部性和空间局部性特点，将内存上的部分数据拉取至缓存中，加快 CPU 的访问速度。实际上，我们对于页表也是这么做的，我们使用一个速度比较快的缓存 TLB

(Translation Lookaside Buffer), 也称为快表，将页表中最近使用的 PTE 缓存下来。

从本质上来说，TLB 就是页表的 Cache。但是 TLB 不同于一般的 Cache，它只有时间相关性，也就是说，现在访问的页，很有可能在以后继续被访问。至于空间相关性，TLB 并没有明显的规律，因为在一个页内有很多情况，都可能使程序跳转到其他不相邻的页中取指令或数据，也就是说，虽然当前在访问一个页，但未必会访问它相邻的页。正因为如此，Cache 设计中很多的优化方法，例如预取 (prefetching)，是没有办法应用于 TLB 中的。

对于 TLB 的写入、缺失处理、控制等相关知识可在计算机组成原理课程中进一步学习。

## 4.2 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。

地址空间某种程度上讲，可以将它看成一块巨大但并不一定真实存在的内存。在每个应用程序的视角里，操作系统分配给应用程序一个地址范围受限（容量很大），独占的连续地址空间（其中有些地方被操作系统限制不能访问，如内核本身占用的虚地址空间等），因此应用程序可以在划分给它的地址空间中随意规划内存布局，它的各个段也就可以分别放置在地址空间中它希望的位置（当然是操作系统允许应用访问的地址）。

由于每个应用独占一个地址空间，里面只含有自己的各个段，于是它可以随意规划属于它自己的各个段的分布而无需考虑和其他应用冲突；同时鉴于应用只能通过虚拟地址读写它自己的地址空间，它完全无法窃取或者破坏其他应用的数据，毕竟那些段在其他应用的地址空间内，这是它没有能力去访问的。这是地址空间抽象和具体硬件机制对应用程序执行的安全性和稳定性的一种保障。

在本章之前，内核和应用代码的访存地址都被视为一个物理地址，并直接访问物理内存，而在分页模式开启之后，CPU 先拿到虚存地址，需要通过 MMU 的地址转换变成物理地址，再交给 CPU 的访存单元去访问物理内存。地址空间抽象的重要意义在于 **隔离 (Isolation)**，当内核让应用执行前，内核需要控制 MMU 使用这个应用的多级页表进行地址转换。由于每个应用地址空间在创建的时候也顺带设置好了多级页表，使得只有那些存放了它的代码和数据的物理页帧能够通过该多级页表被映射到，这样它就只能访问自己的代码和数据而无法触及其他应用或内核的内容。

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)。

#### 4.2.1 内核虚拟地址空间分布

内核虚拟地址空间分布是计算机操作系统中的一个重要概念。在操作系统中，虚拟地址空间是指用户程序所访问的内存空间，它与物理内存空间不同，它是一个虚拟的地址空间。内核虚拟地址空间分布是指内核中的进程管理、内存管理和文件系统等模块所使用的虚拟地址空间的布局和结构。

在计算机系统中，内核虚拟地址空间分布的合理安排可以提高系统的性能。例如，当用户程序需要访问内存中的数据时，如果内存空间不足，系统就需要将数据复制到更多的内存中，这会增加系统的开销。但如果内核虚拟地址空间分布合理，可以将部分数据缓存在磁盘上，从而减少内存空间的占用，提高系统的性能。

因此，了解内核虚拟地址空间分布的基本知识对于我们来讲是非常重要的。本文将介绍内核虚拟地址空间分布的基本概念、组织结构和使用方法，以便读者更好地理解内核虚拟地址空间分布的重要性和作用。

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 ??，我们可以看到 NPUCore 的内核虚拟地址空间分布图：

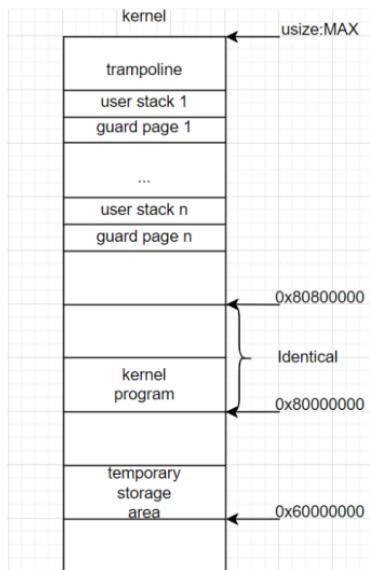


图 4-7 内核虚拟地址空间分布

- **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结构，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容，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。

#### 4.2.2 内核物理地址空间分布

在计算机系统中，内核物理地址空间分布是一个非常重要的概念。它决定了计算机系统如何访问内存和其他硬件资源，并且是操作系统和内核中的一个重要概念。下面将介绍内核物理地址空间分布的基本概念和原理，并提供一些常见的应用场景。

在计算机系统中，物理地址空间是一个固定的物理内存地址范围，用于存储操作系统和内核中的各种数据结构和代码。虚拟地址空间是一个虚拟的内存地址范围，它被操作系统和内核映射到物理地址空间中。内核物理地址空间分布是指在虚拟地址空间中，内核所占用的物理内存地址范围。

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。

内核物理地址空间分布的基本概念和原理对于操作系统和内核的设计和实现常重要。了解内核物理地址空间分布的基本概念和原理可以帮助我们更好地设计和实现操

作系统和内核，从而提高系统的可靠性和性能。此外，内核物理地址空间分布的基本概念和原理也可以应用于其他计算机系统的设计和实现中。

### (1) 内核程序区域

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。

在内核运行时会输出如下内容：

```

1  .text [0x80200000, 0x80237000)
2  .rodata [0x80237000, 0x80240000)
3  .data [0x80240000, 0x80241000)
4  .bss [0x80241000, 0x80492000)
5  mapping .text section
6  mapping .rodata section
7  mapping .data section
8  mapping .bss section
9  mapping physical memory
10 mapping memory-mapped registers

```

上述输出的内容对应创建内核地址空间的方法 new\_kernel 的代码。不同字段的作用如下：

- .text

代码段，存放需要执行的代码，该空间拥有读权限和执行权限。

- .rodata

只读数据段，存放只读数据，该空间拥有读权限。

- .data

数据段，该空间拥有读权限和写权限。要想辨别其与.rodata 的区别，下面的 C 语言示例可以帮助我们更好地理解

```

1  int bss_value1=5; //全局的被初始化的变量，存放于data段
2  const int cbss_value1=5; //加上const变为只读的常量，存放于rodata
   段

```

- .bss

来存储一些未被初始化的全局变量和静态变量的内存区域，一般在初始化时.bss 段部分将会清零。不包含任何数据，只是简单的维护开始和结束的地址，以便内存区能在运行时被有效地清零。可以被读写，但不允许从它上面取指执行。

不同区域读写执行权限不同是通过 MapPermission 映射区域的权限来控制的，标志

如下：

```

1 //os/src/mm/memory_set.rs
2
3 bitflags! {
4     pub struct MapPermission: u8 {
5         const R = 1 << 1; // 读
6         const W = 1 << 2; // 写
7         const X = 1 << 3; // 执
8         const U = 1 << 4; // 用户
9     }
10 }
```

字段地址确定首先要通过外部符号的引用，引入这些标记是为了在下面恒等映射中引用，确定恒等映射的地址区域。下面是 `extern "C"` 的引用：

代码片段 4.14 `extern "C"`

```

1 //os/src/mm/memory_set.rs
2 extern "C" {
3     fn stext();
4     fn etext();
5     fn srodata();
6     fn erodata();
7     fn sdata();
8     fn edata();
9     fn sbss_with_stack();
10    fn ebss();
11    fn ekernel();
12    fn strampoline();
13    fn ssignaltrampoline();
14 }
```

从 `os/src/linker-xxx.ld` 中引用了很多表示各个段位置的符号。可以看到在 `os/src` 目录下有三个 `ld` 文件：`linker-qemu.ld`，`linker-k210.ld`，`linker-fu740.ld`，分别对应 `qemu`，`k210` 和 `fu740` 三个平台。其实三者只有第三行的 `BASE_ADDRESS` 不同，`k210` 为 `0x80020000`，`qemu` 与 `fu740` 均为 `0x80200000`。

此外还有 **physical memory**（物理内存），内核地址空间中需要存在一个恒等映射到内核数据段之外的可用物理页帧的逻辑段，这样才能在启用页表机制之后，内核仍能以纯软件的方式读写这些物理页帧。

有了以上的知识，可以画出这五段逻辑区域的物理地址空间分布如下图 ??：

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 4.15 config.rs

```

1 //os/src/config.rs
2 #[cfg(all(not(feature="board\k210"), not(feature="board\fu740")))]
3 pub const MEMORY_END: usize = 0x809e_0000;
4 #[cfg(feature = "board_k210")]
5 pub const MEMORY_END: usize = 0x8080_0000;
```

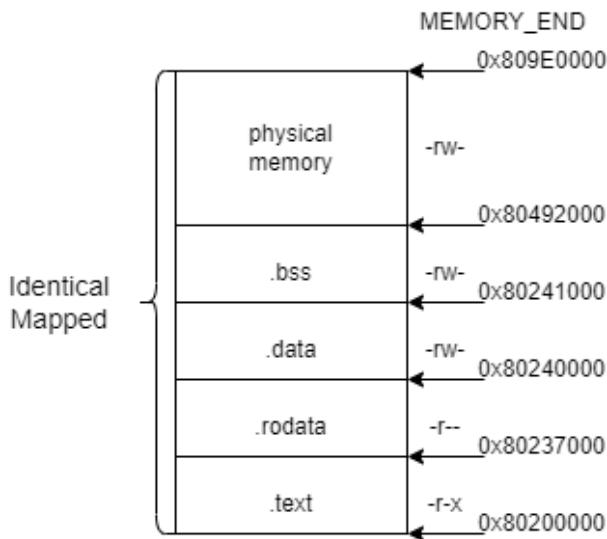


图 4-8 逻辑区域的物理地址空间分布

```

6 #[cfg(feature = "board_fu740")]
7 pub const MEMORY_END: usize = 0x9000_0000;

```

## (2) 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。而且这种方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。

代码片段 4.16 //os/board

```

1 //os/src/boards/qemu.rs
2 pub const MMIO: &[(usize, usize)] = &[
3     (0x1000_0000, 0x1000),
4     (0x1000_1000, 0x1000),
5     (0xC00_0000, 0x40_0000),
6 ];
7 //os/src/boards/fu740.rs
8 pub const MMIO: &[(usize, usize)] = &[
9     (0x1000_0000, 0x1000), // PRCI
10    (DISK_IMAGE_BASE, 0x800_0000), // disk image
11 ];
12 //os/src/boards/k210.rs
13 pub const MMIO: &[(usize, usize)] = &[
14     // we don't need clint in S priv when running
15     // we only need claim/complete for target0 after initializing
16     (0x0C00_0000, 0x3000), /* PLIC */
17     (0x0C20_0000, 0x1000), /* PLIC */
18     (0x3800_0000, 0x1000), /* UARHS */
19     (0x3800_1000, 0x1000), /* GPIOHS */
20     (0x5020_0000, 0x1000), /* GPIO */
21     (0x5024_0000, 0x1000), /* SPI_SLAVE */
22     (0x502B_0000, 0x1000), /* FPIOA */

```

```

23 (0x502D_0000, 0x1000), /* TIMER0 */
24 (0x502E_0000, 0x1000), /* TIMER1 */
25 (0x502F_0000, 0x1000), /* TIMER2 */
26 (0x5044_0000, 0x1000), /* SYSCTL */
27 (0x5200_0000, 0x1000), /* SPI0 */
28 (0x5300_0000, 0x1000), /* SPI1 */
29 (0x5400_0000, 0x1000), /* SPI2 */
30 ];

```

其中每个元组的第一个数为起始地址，第二个数为区域大小。据此来分配 MMIO，权限均为可读可写。在 new\_kernel() 方法中对应的映射代码为：

代码片段 4.17 //os/mm/memory\_set.rs

```

1 macro_rules! anonymous_identical_map {
2     ($begin:expr,$end:expr,$permission:expr) => {
3         memory_set
4             .push(
5                 MapArea::new(
6                     ($begin as usize).into(),
7                     ($end as usize).into(),
8                     MapType::Identical,
9                     $permission,
10                    None,
11                    ),
12                    None,
13                    )
14             .unwrap();
15     };
16     ($name:literal,$begin:expr,$end:expr,$permission:expr) => {
17         println!("mapping {}", $name);
18         anonymous_identical_map!($begin, $end, $permission);
19     };
20 }
21
22 println!("mapping memory-mapped registers");
23 for pair in MMIO {
24     anonymous_identical_map!(
25         (*pair).0,
26         ((*pair).0 + (*pair).1),
27         MapPermission::R | MapPermission::W
28     );
29 }

```

该函数用于将 MMIO 内存区域恒等映射到物理内存中。MMIO 内存区域通常于存储设备驱动程序的缓冲区，这些缓冲区需要被映射到进程的虚拟内存地址空间中，以便进程可以访问它们。

在这段代码中，MMIO 是一个枚举类型，它定义了 MMIO 内存区域的权限和起始地址。`anonymous_identical_map!` 是一个匿名函数，用于做恒等内存映射。该函数接受三个参数：一个起始地址、一个终止地址、以及一个映射权限。在本例中，`(*pair).0` 和 `(*pair).1` 分别表示 MMIO 内存区域的起始地址和长度，`MapPermission::R | MapPermission::W` 表示读取和写入权限。函数的循环遍历了 MMIO 枚举类型中的所

有内存区域，并为每个区域做恒等映射。在创建每个映射时，函数使用了 `anonymous_identical_map!` 函数，以便为每个区域做恒等映射，这些映射空间具有相同的权限。

### (3) 地址空间实现

在前面，我们介绍了地址空间的概念层面，描述了地址空间的分布和映射关系。接下来，我们将深入 NPUcore 内部，探讨地址空间这一抽象概念在 NPUcore 中的实现。

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。具体代码如下：

代码片段 4.18 MemorySet

```

1 pub struct MemorySet {
2     page_table: PageTable,
3     /// The mapped area.
4     /// Segments are implemented using this mechanism. In other words,
5     /// they may be considered a subset of MapArea.
6     /// Yet, other purposes may exist in this struct, such as file
7     /// mapping.
8     areas: Vec<MapArea>,
9 }

```

有关多级页表的具体实现将在后续章节详细讲解。这里我们主要关注 `MapArea` 的实现：

代码片段 4.19 MapArea

```

1 #[derive(Clone)]
2 /// Map area for different segments or a chunk of memory for memory
3     /// mapped file access.
4 pub struct MapArea {
5     /// Range of the mapped virtual page numbers.
6     /// Page aligned.
7     /// Map physical page frame tracker to virtual pages for RAI &
8     /// lookup.
9     inner: LinearMap,
10    /// Direct or framed(virtual) mapping?
11    map_type: MapType,
12    /// Permissions which are the OR of RWXU, where U stands for user.
13    map_perm: MapPermission,
14    pub map_file: Option<Arc<dyn File>>,
15 }
16
17 #[derive(Copy, Clone, PartialEq, Debug)]
18 pub enum MapType {
19     Identical,
20     Framed,
21 }

```

`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，其实现如下：

代码片段 4.20 LinearMap

```

1 #[cfg(not(feature = "oom_handler"))]
2 #[derive(Clone)]
3 pub struct LinearMap {
4     vpn_range: VPNRange,
5     frames: Vec<Frame>,
6 }

```

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。关于 Frame 的枚举类型的具体代码如下：

代码片段 4.21 Frame

```

1 pub enum Frame {
2     InMemory(Arc<FrameTracker>),
3     Unallocated,
4 }
5
6 pub struct FrameTracker {
7     pub ppn: PhysPageNum,
8 }

```

这里利用枚举类型对 FrameTracker 做了一个封装，FrameTracker 绑定每个物理页面，作为物理页面跟踪器。将物理页追踪器映射到虚拟页面有利于 RAII 和进行页面查找。

RAII (Resource Acquisition Is Initialization)，也称为“资源获取就是初始化”，简单地说，RAII 的做法是使用一个对象，在其构造时获取资源，在对象生命周期内控制对资源的访问，使之始终保持有效，最后在对象析构的时候释放资源。

在这里就是使 FrameTracker 与物理页面有相同的生命周期，当获取物理页面构建 FrameTracker 时就将物理页面初始化，在释放 FrameTracker 时也将自动回收物理页面。具体见以下代码：

代码片段 4.22 FrameTracker

```

1 impl FrameTracker {
2     pub fn new(ppn: PhysPageNum) -> Self {
3         // page cleaning
4         let dwords_array = ppn.get_dwords_array();
5         for i in dwords_array {
6             *i = 0;
7         }
8         Self { ppn }
9     }
10
11    pub unsafe fn new_uninit(ppn: PhysPageNum) -> Self {
12        Self { ppn }
13    }
14
15    impl Drop for FrameTracker {
16        // Automatically recycle the physical frame when
17        fn drop(&mut self) {
18            // println!("do drop at {}", self.ppn.0);
19            frame_dealloc(self.ppn);
20        }
21    }

```

```

21     }
22 }
```

需要说明的是 FrameTracker 的生命周期也被一层层绑定到它所在的逻辑段 MapArea 下，当逻辑段被回收之后这些之前分配的物理页帧也会自动地同时被回收。同理，MapArea 的生命周期也随着 MemorySet，这样就做到了当一个地址空间 MemorySet 生命周期结束后，这些物理页帧都会被回收。接下来分析几个 MemorySet 接口下常用方法：

创建内核地址空间的方法 new\_kernel。具体创建方式在前面章节已经给出。这里给出该函数下调用的其他与 MemorySet 相关的函数：

代码片段 4.23 内核创建

```

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

首先调用的是 new\_bare 函数，其作用是创建一个空的页表和空的 areas 向量，具体实现如下：

代码片段 4.24 new\_bare

```

1 impl MemorySet {
2     // Create a new struct with no information at all.
3     pub fn new_bare() -> Self {
4         Self {
5             page_table: PageTable::new(),
6             areas: Vec::with_capacity(16),
7         }
8     }
9 }
```

之后首先映射跳板区域，注意这个跳板区域并没有被包括在 areas 里。

代码片段 4.25 trampoline

```

1 fn map_trampoline(&mut self) {
2     self.page_table.map(
3         VirtAddr::from(TRAMPOLINE).into(),
4         PhysAddr::from(strampoline as usize).into(),
5         PTEFlags::R | PTEFlags::X,
6     );
7 }
```

物理地址是通过 extern "C" 引用的外部符号 strampoline，TRAMPOLINE 被设置为最高虚拟页。跳板页面存放的是可执行的代码，权限是读与执行。

其余区域的映射方式大致相同，NPUCore 定义了一个 anonymous\_identical\_map 宏（匿名恒等映射），来减少重复代码。具体代码如下：

## 代码片段 4.26 anonymous\_identical\_map

```

1 macro_rules! anonymous_identical_map {
2     ($begin:expr,$end:expr,$permission:expr) => {
3         memory_set
4         .push(
5             MapArea::new(
6                 ($begin as usize).into(),
7                 ($end as usize).into(),
8                 MapType::Identical,
9                 $permission,
10                None,
11            ),
12            None,
13        )
14        .unwrap();
15    };
16    ($name:literal,$begin:expr,$end:expr,$permission:expr) => {
17        println!("mapping {}", $name);
18        anonymous_identical_map!($begin, $end, $permission);
19    };
20 }

```

这个宏有两个分支。四个参数时，第一个参数为被 map 的区域名称，会被打印一下再调用三个参数的分支。三个参数时，分别为起始地址，终止地址和映射区域的权限。注意在 MapArea::new 中有 MapType::Identical，这个决定了调用这个宏的映射都是恒等映射。这个宏主体部分调用了 MemorySet 的 push 方法：

## 代码片段 4.27 push

```

1 fn push(&mut self, mut map_area: MapArea, data: Option<&[u8]>) ->
2     Result<(), MemoryError> {
3     match data {
4         Some(data) => {
5             let mut start = 0;
6             let len = data.len();
7             for vpn in map_area.inner.vpn_range {
8                 let ppn = map_area.map_one(&mut self.page_table, vpn)?;
9                 let end = start + PAGE_SIZE;
10                let src = &data[start..len.min(end)];
11                ppn.get_bytes_array()[..src.len()].copy_from_slice(src)
12                ;
13                start = end;
14            }
15        }
16        None => {
17            for vpn in map_area.inner.vpn_range {
18                map_area.map_one(&mut self.page_table, vpn)?;
19            }
20        }
21        self.areas.push(map_area);
22        Ok(())
23    }

```

方法的功能是，将尚未映射的 MapArea push 到当前 MemorySet 中，并为映射分配

所需的内存，如果没有数据，那么直接遍历 map\_area.inner.vpn\_range（虚拟页面页号），使用 MapArea 的 map\_one 方法来映射每个页面。如果有数据的话，在上述操作之外还需添加将数据拷贝映射后的物理页面的操作。下面看一下 map\_one 方法的实现：

代码片段 4.28 map\_one

```

1  pub fn map_one(
2      &mut self,
3      page_table: &mut PageTable,
4      vpn: VirtPageNum,
5  ) -> Result<PhysPageNum, MemoryError> {
6      if !page_table.is_mapped(vpn) {
7          //if not mapped
8          Ok(self.map_one_unchecked(page_table, vpn))
9      } else {
10          //mapped
11          Err(MemoryError::AlreadyMapped)
12      }
13  }
14
15  pub fn map_one_unchecked(
16      &mut self,
17      page_table: &mut PageTable,
18      vpn: VirtPageNum,
19  ) -> PhysPageNum {
20      let ppn: PhysPageNum;
21      match self.map_type {
22          MapType::Identical => {
23              ppn = PhysPageNum(vpn.0);
24          }
25          MapType::Framed => {
26              let frame = unsafe { frame_alloc_uninit().unwrap() };
27              ppn = frame.ppn;
28              self.inner.alloc_in_memory(vpn, frame);
29          }
30      }
31      let pte_flags = PTEFlags::from_bits(self.map_perm.bits).unwrap();
32      page_table.map(vpn, ppn, pte_flags);
33      ppn
34  }

```

map\_one 首先会检查传入的 vpn 是否已经被分配，如已经分配会报错 MemoryError::AlreadyMapped，未分配移交给 map\_one\_unchecked 来分配。分配时会根据分配类型来分别处理，如为 Identical 则物理地址与虚拟地址恒等映射，如为 Framed 则会申请一个空闲物理页面 (frame\_alloc\_uninit) 与之映射 (page\_table.map)。

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。

## 4.3 物理内存分配器

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。

接下来我们先来看本节涉及到的数据结构关系：

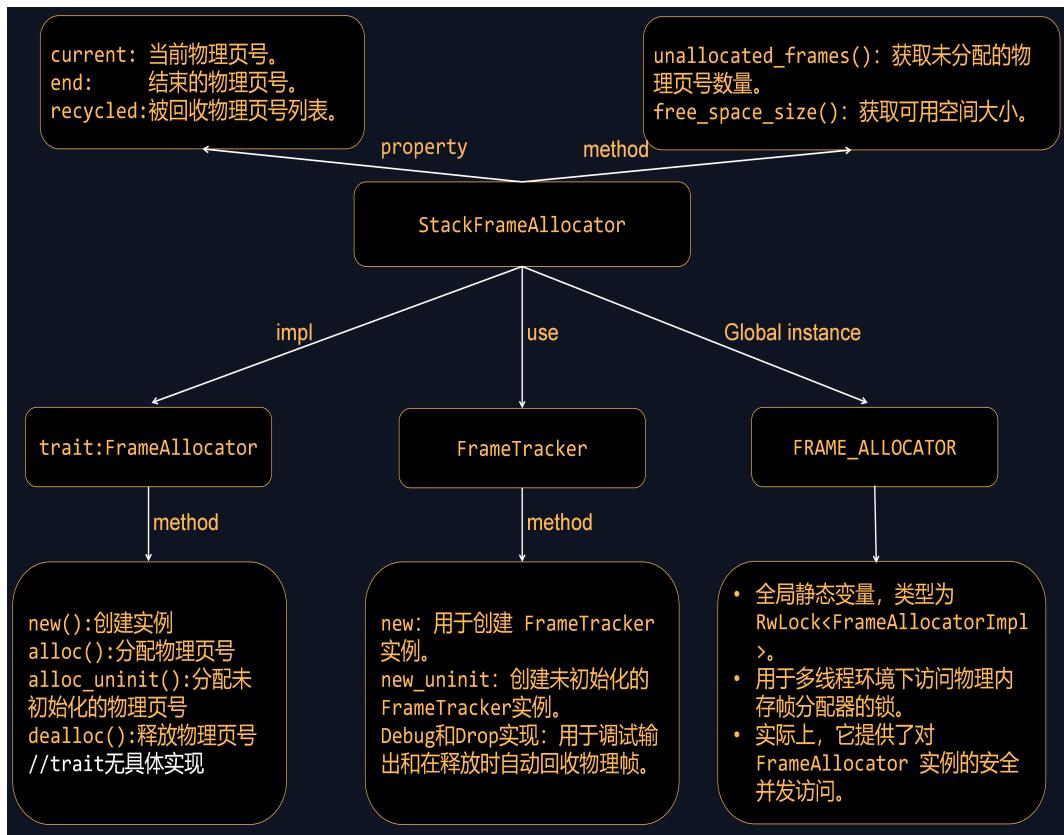


图 4-9 NPUCore 中物理内存分配涉及的数据结构

**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\_ALLOCATOR 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。

综上所述，StackFrameAllocator 结构体实现了 FrameAllocator 这个特性，它在管理物理内存帧的同时，通过调用 FrameTracker 中的方法对物理页面进行操作。同时，它创建了一个全局变量 FRAME\_ALLOCATOR，并提供了线程安全的访问物理内存分配器的机制。

物理内存分配器在计算机操作系统中扮演着非常重要的角色，它负责动态地分配和释放物理内存资源，以确保程序在运行时能够获得足够的物理内存资源，并尽可能地提高系统的稳定性和性能。

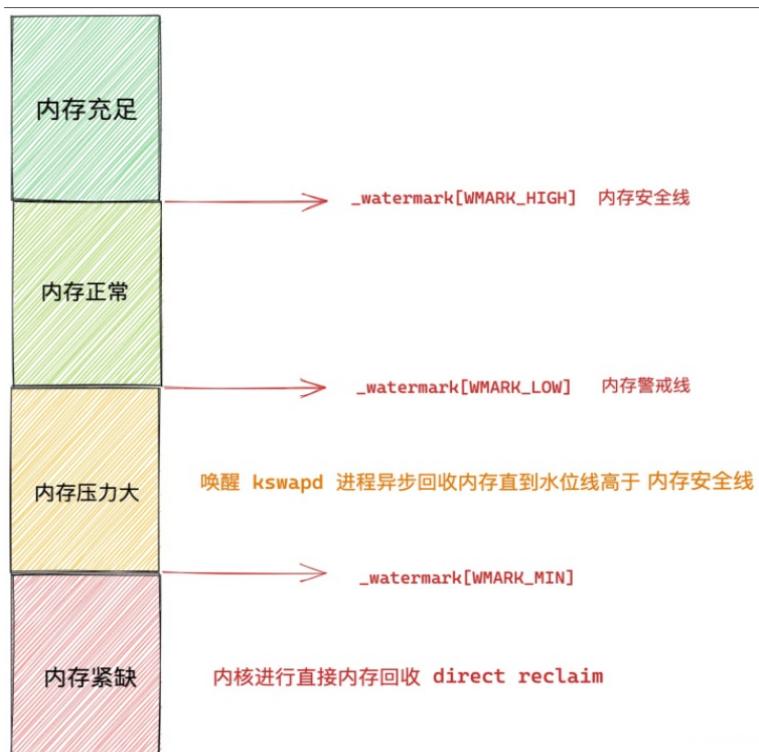


图 4-10 物理内存分配器示意图

物理内存分配器通常包括一个页面置换算法和一个内存映射表。页面置换算法用于在内存池中选择要替换的页面，以腾出空间来分配新的页面；内存映射表用于记录页面在内存中的位置和大小，以及对应的虚拟地址。在分配物理内存时，物理内存分配器会根据页面大小和页面置换算法来决定哪些页面需要被替换，并将新的页面分配到内存池中。在程序运行过程中，如果程序需要的物理内存超过了当前可用的物理内存，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。

#### 4.3.1 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，不同平台上的物理内存空间范围如下：

```

1 // os\src\config.rs
2 pub const MEMORY_START:usize = 0x8000_0000;
3 #[cfg(all(not(feature = "board_k210"), not(feature = "board_fu740")))]
4 pub const MEMORY_END:usize = 0x809e_0000;
5 #[cfg(feature = "board_k210")]
6 pub const MEMORY_END:usize = 0x8080_0000;
7 #[cfg(feature = "board_fu740")]
8 pub const MEMORY_END:usize = 0x9000_0000;
9 pub const PAGE_SIZE:usize = 0x1000;

```

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。

#### 4.3.2 物理内存分配器初始化

在编写程序时，我们需要将内存分配给自己编写的代码和各种外部库，如果在此之前未进行正确的内存初始化，程序可能会出现各种问题，例如内存泄漏、程序崩溃等。内存初始化是计算机编程中非常重要的一个步骤，用以确保程序在运行时能够正确地分配和释放内存，避免不必要的内存浪费和错误。它关系到程序能否正确运行以及运行的稳定性和可靠性。

内存的初始化中包含物理内存分配器的初始化，在这一步骤中，使用上文提到的物理内存空间来初始化内存分配器，让内存分配器清楚待分配的物理空间范围。

```

1 // os\src\mm\frame_allocator.rs
2 pub fn init_frame_allocator(){
3     extern "C" {
4         fn ekernel();
5     }
6     FRAME_ALLOCATOR.write().init(
7         PhysAddr::from(ekernel as usize).ceil(),
8         PhysAddr::from(MEMORY_END).floor(),
9     );
10 }

```

ekernel 即为内核空间的代码和数据存放的末尾，此地址即为 MEMORY\_START，从 MEMORY\_START 到 MEMORY\_END，剩下的空间都将被 frame\_allocator 分配使用。

### 4.3.3 物理内存分配器接口

在了解了所分配的物理空间范围并完成物理内存分配器的初始化后，我们可以详细了解物理内存分配器提供的接口，以及基本的物理内存分配器需要实现哪些功能。

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。

```
1 //os\src\mm\frame_allocator.rs
2 trait FrameAllocator {
3     fn new() -> Self;
4     fn alloc(&mut self) -> Option<FrameTracker>;
5     unsafe fn alloc_uninit(&mut self) -> Option<FrameTracker>;
6     fn dealloc(&mut self, ppn: PhysPageNum);
7 }
```

RAII 与 FrameTracker 在前文介绍地址空间时提到，FrameTracker 绑定了每个物理页面作为物理页面追踪器，将物理页面追踪器映射到虚拟页面有利于 RAII 和页面查找；RAII 指的是 FrameTracker 与物理页面具有相同的生命周期，在获取物理页面构建 FrameTracker 时会将物理页面初始化，在 drop FrameTracker 时也会自动回收物理页面。

```
1 //os\src\mm\frame_allocator.rs
2 pub struct FrameTracker {
3     pub ppn: PhysPageNum,
4 }
5 ///RAII phantom for physical pages
```

new 方法具体实现在 new() 方法中，获取物理页面号后，通过 get\_dwords\_array() 获取 64 字节的数组，逐个元素赋零以实现页面的初始化（清零）。

```
1 pub fn new(ppn: PhysPageNum) -> Self {
2     // page cleaning
3     let dwords_array = ppn.get_dwords_array();
4     for i in dwords_array {
5         *i = 0;
6     }
7     Self {ppn}
8 }
```

new\_uninit() 提高性能原因：循环赋零会产生时间开销。为了应对一些情况下不需要对页面进行清零操作的情况（比如上一节介绍的 COW 处理，在申请到新页面时直接进行完全拷贝），提供了 new\_uninit() 方法，它不执行清零操作，直接返回由物理页面号构造的 FrameTracker 实例。

```
1 pub unsafe fn new_uninit(ppn: PhysPageNum) -> Self {
2     Self {ppn}
3 }
```

正是在 FrameTracker 提供了 new\_uninit 的方法，才在上层的 FrameAllocator 中支持了 alloc\_uninit。

#### 4.3.4 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。

如果没有全局物理内存分配器，每个进程或线程都需要自己管理内存分配和释放，那么可能会出现以下问题：

**内存泄漏：**当程序在分配内存时，如果未能正确释放内存，则可能会导致内存泄漏。这会导致程序占用的内存不断增加，最终导致程序崩溃或拒绝服务。

**内存分配异常：**如果进程或线程需要分配的内存大小超过了系统可用的内存大小，则可能会导致内存分配异常。这可能会导致程序崩溃或拒绝服务。

**内存分配和释放的不稳定性：**每个进程或线程都需要自己管理内存分配和释放，这可能会导致内存分配和释放的不稳定性。例如，某个进程或线程可能会在释放内存后不久就重新分配内存，从而导致内存泄漏或其他问题。

为了解决这些问题，我们需要一个全局的物理内存分配器来负责管理程序运行时的物理内存分配和释放。全局物理内存分配器可以确保内存分配和释放的稳定性和可靠性，减少内存泄漏和其他问题的发生。此外，全局物理内存分配器还可以提高程序的性能和吞吐量，因为内存分配和释放不再由不同的进程或线程单独管理，而是由一个统一的内存管理机制来负责。

接下来我们深入 NPUCore 内部，探究一下 NPUCore 是如何实现全局物理内存分配器的。

首先创建一下 StackFrameAllocator 的全局实例 FRAME\_ALLOCATOR：

```

1 // os\src\mm\frame_allocator.rs
2 type FrameAllocatorImpl = StackFrameAllocator;
3 lazy_static!{
4     pub static ref FRAME_ALLOCATOR:RwLock<FrameAllocatorImpl>=
5     RwLock::new(FrameAllocatorImpl::new());
6 }
```

这里我们使用 `RwLock<T>` 来包裹栈式物理页帧分配器，加上一层读写锁。每次对该分配器进行操作之前，我们都需要先通过 `FRAME_ALLOCATOR.write()` 拿到分配器的写权限，以保证排他性。

公开给其他内核模块调用的分配/回收物理页帧的接口为：

```

1 // os\src\mm\frame_allocator.rs
2 pub fn frame_alloc() -> Option<Arc<FrameTracker>> {
3     FRAME_ALLOCATOR
4         .write().alloc().map(|frame_tracker|Arc::new(frame_tracker))
5 }
6 pub unsafe fn frame_alloc_uninit() -> Option<Arc<FrameTracker>> {
7     FRAME_ALLOCATOR
8         .write().alloc_uninit().map(|frame_tracker|Arc::new(frame_tracker))
9 }
```

```

10 pub fn frame_dealloc(ppn:PhysPageNum) {
11     FRAME_ALLOCATOR.write().dealloc(ppn);
12 }

```

alloc 申请来的 frame\_tracker 用 Arc 包裹实现自动引用计数再返回。在 FrameTracker 被 Drop 的时候，会调用 frame\_dealloc 方法。

```

1 // os\src\mm\frame_allocator.rs
2 impl Drop for FrameTracker{
3     //Automatically recycle the physical frame when
4     fn drop(&mut self) {
5         // println!("do drop at {}", self.ppn.0);
6         frame_dealloc(self.ppn);
7     }
8 }

```

这样就做到了当一个 FrameTracker 实例被回收的时候，它的 drop 方法会自动被编译器调用，通过之前实现的 frame\_dealloc，进一步调用 StackFrameAllocator 的 dealloc，即可将它控制的物理页帧回收以供后续使用。

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。

## 4.4 内存分配办法

全局物理内存分配器提供给内核其他模块申请物理内存的统一方法，而物理内存分配方法的具体实现在内存分配器中。接下来让我们深入物理内存分配器内部，来探究一下在物理内存分配器内部是如何实现物理内存分配的。

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。

NPUcore 使用的正是栈式的内存管理策略，下面我们将以 NPUcore 的具体实现为例，来介绍物理内存分配方法。

### 4.4.1 栈式内存管理

首先用一个结构体来记录空闲的物理页面：

```

1 pub struct StackFrameAllocator {
2     current:usize,
3     end:usize,
4     recycled:Vec<usize>,}

```

为这个 StackFrameAllocator 实现 FrameAllocator 接口，new 方法将空闲内存区间两端均设为 0，然后创建一个新的向量。

```

1 // os\src\mm\frame_allocator.rs
2 impl FrameAllocator for StackFrameAllocator {
3     fn new() -> Self {
4         Self {
5             current: 0,
6             end: 0,
7             recycled: Vec::new(),
8         }
9     }
10}

```

在它真正被使用起来之前，需要调用 StackFrameAllocator 的 init 方法将自身的 [current,end) 初始化为可用物理页号区间。

用结束物理页号减去起始物理页号得到剩余有多少物理页面。在内核运行时输出的 last Physical Frames，就是起源于这里，输出还有多少空闲页面。这里还用了向量的 reserve 方法来预留出足够的空间，避免在运行中不断的重新分配。该方法的方法签名为“pub fn reserve(&mut self, additional: usize)”，用于为 Vec <T> 预留至少 additional 个元素的容量。调用 reserve 后，Vec 的容量将大于或等于 self.len() + additional。如果当前容量已经足够，该方法则不执行任何操作。

```

1 // os\src\mm\frame_allocator.rs
2 impl StackFrameAllocator {
3     pub fn init(&mut self, l: PhysPageNum, r: PhysPageNum) {
4         self.current = l;
5         self.end = r;
6         let last_frames = self.end - self.current;
7         self.recycled.reserve(last_frames);
8         println!("last {} Physical Frames.", last_frames);
9     }
10    pub fn unallocated_frames(&self) -> usize {
11        self.recycled.len() + self.end - self.current
12    }
13    pub fn free_space_size(&self) -> usize {
14        self.unallocated_frames() * PAGE_SIZE
15    }
16}

```

unallocated\_frames 用于计算未被分配的页面数量，为 recycled 中的空闲页面和整块的空闲内存页面数量之和。free\_space\_size 计算空闲空间的大小，只需空闲页面的数量乘页面大小。

#### 4.4.2 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。

```

1 // os\src\mm\frame_allocator.rs
2 impl FrameAllocator for StackFrameAllocator {
3     fn alloc(&mut self) -> Option<FrameTracker> {
4         if let Some(ppn) = self.recycled.pop() {
5             let frame_tracker = FrameTracker::new(ppn.into());
6             log::trace!("[frame_alloc] {:?}", frame_tracker);
7             Some(frame_tracker)
8         } else if self.current == self.end {
9             None
10        } else {
11            self.current += 1;
12            let frame_tracker = FrameTracker::new((self.current - 1).into());
13            log::trace!("[frame_alloc] {:?}", frame_tracker);
14            Some(frame_tracker)
15        }
16    }
17 }

```

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，我们看一下二者的不同：

<pre> 1 fn alloc(&amp;mut self) -&gt; Option&lt;FrameTracker&gt; { 2     if let Some(ppn) = self.recycled.pop() { 3         let frame_tracker = FrameTracker::new(ppn.into()); 4         log::trace!("[frame_alloc] {:?}", frame_tracker); 5         Some(frame_tracker) 6     } else if self.current == self.end { 7         None 8     } else { 9         self.current += 1; 10        let frame_tracker = FrameTracker::new((self.current - 1).into()); 11        log::trace!("[frame_alloc] {:?}", frame_tracker); 12        Some(frame_tracker) 13    } 14 } </pre>	<pre> 1 unsafe fn alloc_uninit(&amp;mut self) -&gt; Option&lt;FrameTracker&gt; { 2     if let Some(ppn) = self.recycled.pop() { 3         let frame_tracker = FrameTracker::new_uninit(ppn.into()); 4         log::trace!("[frame_alloc_uninit] {:?}", frame_tracker); 5         Some(frame_tracker) 6     } else if self.current == self.end { 7         None 8     } else { 9         self.current += 1; 10        let frame_tracker = FrameTracker::new_uninit((self.current - 1).into()); 11        log::trace!("[frame_alloc_uninit] {:?}", frame_tracker); 12        Some(frame_tracker) 13    } 14 } </pre>
--	--

图 4-11 alloc 与 alloc\_uninit 对比图

其实也就是在 FrameTracker::new 的时候不同，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。

#### 4.4.3 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。

```

1 // os\src\mm\frame_allocator.rs
2 // Deallocate a physical page
3 fn deallocate(&mut self, ppn: PhysPageNum) {
4     log::trace!("[frame_dealloc] {:?}", ppn);
5     let ppn = ppn.0;
6     // validity check, note that this should be unnecessary for RELEASE
7     // build and it
8     if option_env!("MODE") == Some("debug") && ppn >= self.current

```

```

8     self.recycled.iter().find(|&v| *v == ppn).is_some()
9     {
10         panic!("Frame ppn={:#x} has not been allocated!", ppn);
11     }
12     // recycle
13     self.recycled.push(ppn);
14 }

```

在这个回收的过程中，首先记录了将要回收的物理页面 ppn，然后在调试模式下进行了有效性检查，以确保被释放的物理页面确实已经被分配过，防止程序在运行时因未分配的页面进行回收而出现问题。

这个有效性检查使用了 recycled.iter().find() 方法，耗时较多，这也是为何在构建 RELEASE 版本时不必执行该检查的原因。对于发布版本来说，已经经过充分测试，假设不会出现未分配页面回收的情况。

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。

## 4.5 用户地址空间

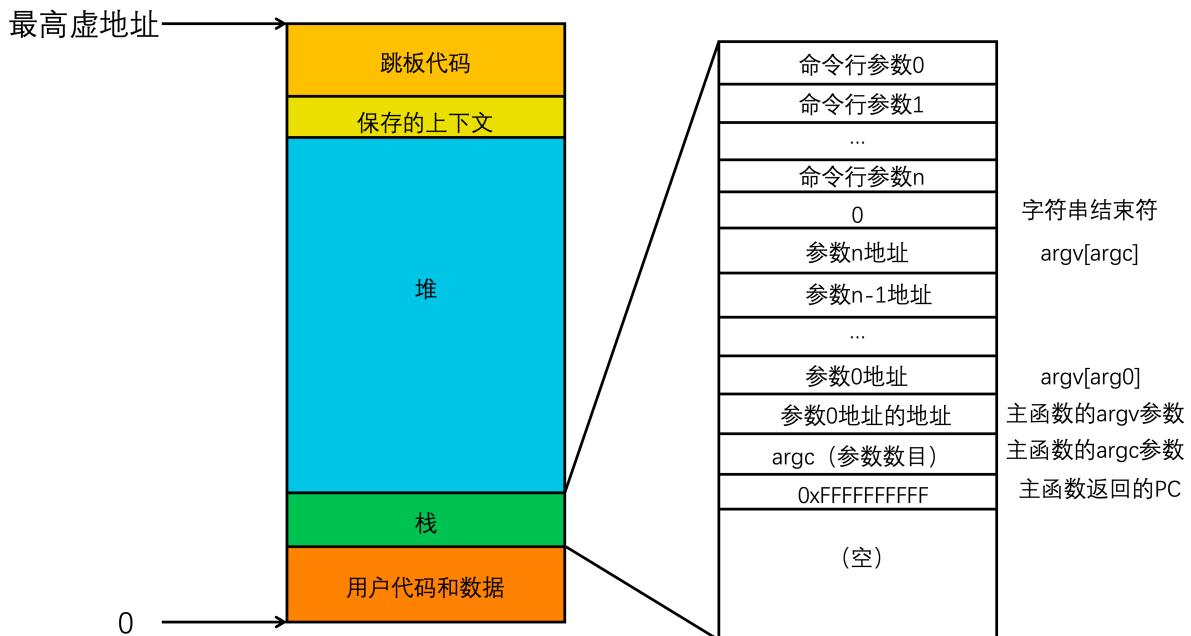


图 4-12 用户虚拟地址空间示意

每一个进程都有一个独立的页表，当 npucore 实现进程切换的时候，对应的页表也会切换。

当一个用户进程通过系统调用向操作系统请求更多的用户空间时，npucore 首先在 os/src/frame\_allocator.rs 中的 StackFrameAllocator 的基于栈的数据结构实现空闲物理

页的分配，然后将物理页的映射加入到用户的页表当中。`npucore` 将设置 PTEflags::R, PTEflags::W, PTEflags::U, PTEflags::X 以及 PTEflags::V 标志位到对应的页表项目，使得用户可以对分配的页面进行读写操作。大多数的用户进程并不能完全利用所有的虚拟空间，对于没有用到的空间，它对应的页表项 PTEflags::V 标志位始终为 0。

页表的设计有很多的好处。首先，首先不同的进程使用不同的页表，相同的虚拟地址映射到不同的物理地址，因此每一个进程可以拥有自己独立的内存空间。其次，用户的虚拟地址是连续的，对应的物理地址不一定是连续的，这样可以有效的避免内存碎片。最后，内核将所有的用户的跳板代码都映射到了同一段虚地址，可以有效的实现上下文切换。

如图??所示，用户的虚拟地址空间被分为了三个部分，分别是用户代码段，用户数据段以及用户堆栈段。用户代码段用于存放用户的代码，用户数据段用于存放用户的数据，用户堆栈段用于存放用户的堆栈。其中，用户栈的初始内容如图中所示，由 `execve` 函数完成初始化，其中包含了用户的命令行参数和返回地址，紧接着就是 `main` 函数使用的栈空间。堆是动态内存分配的区域，用于存储在运行时分配的数据。在 `npucore` 中，堆的起始地址通常在数据段结束后，并根据需要进行扩展。通过系统调用（如 `mmap` 和 `sbrk`），用户程序可以动态地管理堆内存。栈是用于存储函数调用和局部变量的区域。在 `npucore` 中，栈从高地址向低地址生长，通常位于用户地址空间的顶部。栈的大小可以通过操作系统配置或在运行时动态调整。。

代码段是存储进程执行指令的区域。在 `npucore` 中，代码段通常从虚拟地址 0 开始，并包含可执行程序的指令。这个区域对应于用户程序的文本部分。数据段包含了全局变量和静态变量等数据。`npucore` 将数据段安排在代码段之后，从一个虚拟地址开始，并在运行时动态地分配和使用。

用户在代码段执行用户程序，将 `elf` 指定的数据段内容存放在相应的数据段位置。因为我们对与内核的映射是直接映射，所以操作系统对于用户的地址空间是直接可见的。同时，堆栈段的大小是动态变化的，当用户程序需要更多的堆栈空间时，会通过系统调用向操作系统请求更多的堆栈空间，操作系统会在用户的地址空间中分配更多的堆栈空间，并将堆栈空间映射到用户的地址空间中

`npucore` 实现了 `execve` 来将 `elf` 文件加载到内存的进程地址空间中，实现了 `sbrk` 来动态的分配用户空间，实现了 `mmap` 来将文件映射到用户空间，实现了 `munmap` 来取消文件的映射。`execve` 将 `elf` 文件中程序所规定的代码和数据加载到内存中，然后操作系统需要建议对应的映射，从而实现了用户地址空间的建立。

#### 4.5.1 sbrk 系统调用

`sbrk` 系统调用是早期的 Unix 系统中的一个系统调用，用于动态的分配用户空间。`sbrk` 系统调用的原型如下：

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

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。

代码片段 4.29 sbrk

```

1  pub fn sbrk(&mut self, heap_pt: usize, heap_bottom: usize, increment: 
2    isize) -> usize {
3    let old_pt: usize = heap_pt;
4    let new_pt: usize = old_pt + increment as usize;
5    // 判断扩大堆还是缩小堆
6    if increment > 0 {
7      let limit = heap_bottom + USER_HEAP_SIZE;
8      if new_pt > limit {
9        return old_pt;
10     } else {
11       self.mmap(
12         old_pt,
13         increment as usize,
14         MapPermission::R | MapPermission::W | MapPermission::U,
15         MapFlags::MAP_ANONYMOUS | MapFlags::MAP_FIXED |
16           MapFlags::MAP_PRIVATE,
17         1usize.wrapping_neg(),
18         0,
19       );
20       trace!("[sbrk] heap area expanded to {:X}", new_pt);
21     }
22   } else if increment < 0 {
23     // 如果缩小后的堆地址小于堆底地址，则不进行缩小
24     if new_pt <= heap_bottom {
25       return old_pt;
26     } else {
27       self.munmap(old_pt, increment as usize).unwrap();
28     }
29   }
30   new_pt
31 }
```

sbrk 的实现依赖于函数 mmap 的实现，将在接下来的章节介绍，mmap 是比 sbrk 更加灵活的系统调用，能处理更复杂的内存分配和使用的情况。

#### 4.5.2 mmap

代码片段 4.30 sys\_mmap

```

1  pub fn sys_mmap(
2    start: usize,
3    len: usize,
```

```

4     prot: usize,
5     flags: usize,
6     fd: usize,
7     offset: usize,
8 ) -> isize

```

参数 start: 指向欲映射的内存起始地址, 通常设为 NULL, 代表让系统自动选定地址, 映射成功后返回该地址。

- 参数 len: 代表将文件中多大的部分映射到内存。
- 参数 prot: 映射区域的保护方式。
- 参数 flags: 影响映射区域的各种特性。在调用 mmap() 时必须要指定 MAP\_SHARED 或 MAP\_PRIVATE。MAP\_FIXED 如果参数 start 所指的地址无法成功建立映射时, 则放弃映射, 不对地址做修正。通常不鼓励用此旗标。MAP\_SHARED 对映射区域的写入数据会复制回文件内, 而且允许其他映射该文件的进程共享。MAP\_PRIVATE 对映射区域的写入操作会产生一个映射文件的复制, 即私人的“写入时复制”(copy on write) 对此区域作的任何修改都不会写回原来的文件内容。MAP\_ANONYMOUS 建立匿名映射。此时会忽略参数 fd, 不涉及文件, 而且映射区域无法和其他进程共享。MAP\_DENYWRITE 只允许对映射区域的写入操作, 其他对文件直接写入的操作将会被拒绝。MAP\_LOCKED 将映射区域锁定住, 这表示该区域不会被置换(swap)。
- 参数 fd: 要映射到内存中的文件描述符。如果使用匿名内存映射时, 即 flags 中设置了 MAP\_ANONYMOUS, fd 设为-1。
- 参数 offset: 文件映射的偏移量, 通常设置为 0, 代表从文件最前方开始对应, offset 必须是分页大小的整数倍。
- 返回值: 若映射成功则返回映射区的内存起始地址, 否则返回-1。

mmap 用于把文件映射到用户空间中, 简单说 mmap 就是把一个文件的内容在内存里面做一个映像。映射成功后, 用户对这段内存区域的修改可以直接反映到内核空间, 同样, 内核空间对这段区域的修改也直接反映用户空间。那么对于内核空间<—>用户空间两者之间需要大量数据传输等操作的话效率是非常高的。进程可以像读写内存一样对普通文件的操作。mmap 系统调用使得进程之间通过映射同一个普通文件实现共享内存。UNIX 网络编程第二卷进程间通信对 mmap 函数进行了说明。该函数主要用途有三个: 1、将一个普通文件映射到内存中, 通常在需要对文件进行频繁读写时使用, 这样用内存读写取代 I/O 读写, 以获得较高的性能; 2、将特殊文件进行匿名内存映射, 可以为关联进程提供共享内存空间; 3、为无关联的进程提供共享内存空间, 一般也是将一个普通文件映射到内存中。第一步: 找到最后一次 mmap 映射区域

```

1 let idx = self.last_mmap_area_idx();

```

每个进程所申请的用户地址空间是用 vector 存储的 vm\_area\_struct 的结构体, 其中 start

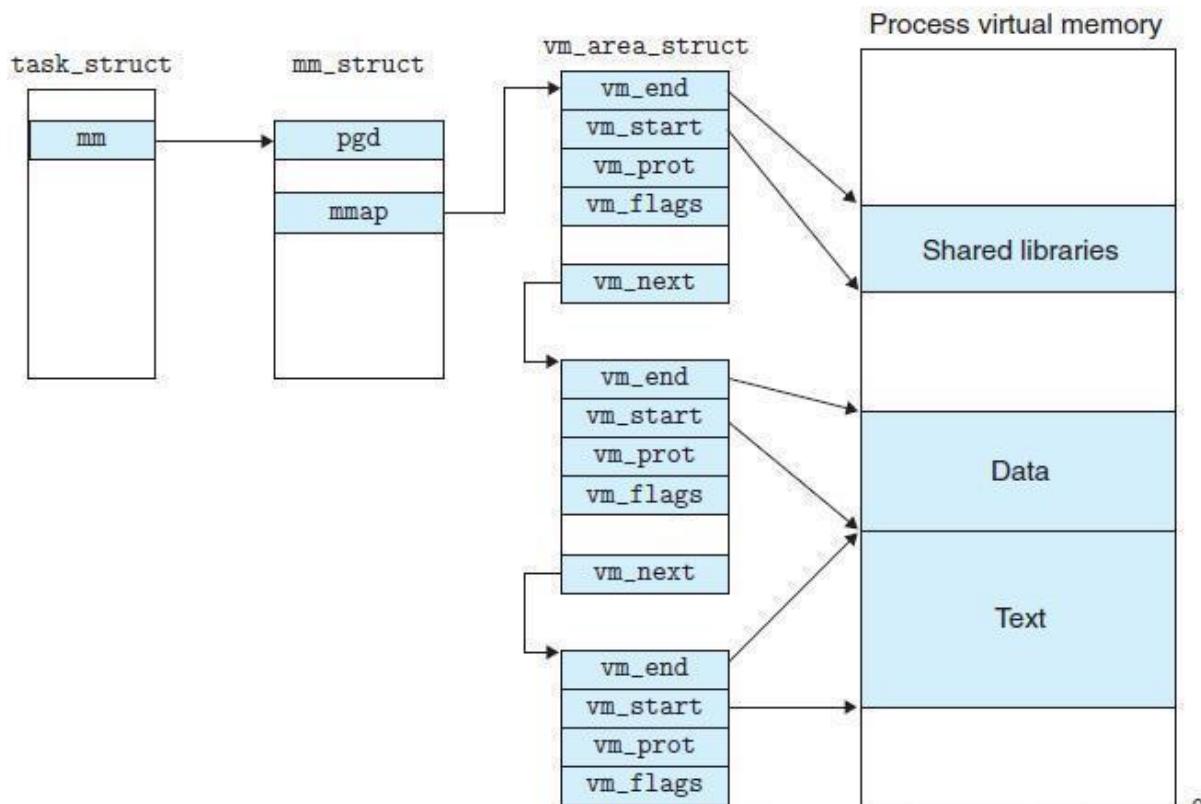


图 4-13 内存地址空间

和 `end` 就是代表这一个 area 虚拟地址的开始和结束，而我们的结构体在 vector 中是按照虚拟地址升序排序的，也就是说我们会找到最后一个符合要求的 vector 下标

第二步：`mmap` 获取映射区域的起始地址，如果设置了 `MAP_FIXED`，则会直接使用用户的参数 `start`，不对地址做修正。

代码片段 4.31 mmap

```

1 let start_va: VirtAddr = if flags.contains(MapFlags::MAP_FIXED) {
2     // unmap if exists
3     self.munmap(start, len);
4     start.into()
5 }
```

如果没有设置，`mmap` 就会拿到先前得到的最后一次 `mmap` 映射区域。如果设置了 `MAP_PRIVATE` 或者 `MAP_ANONYMOUS`，就直接将要新映射的区域 `new_area` 与最后一次 `mmap` 映射区域合并并返回，没有设置就会将最后一次 `mmap` 映射区域的末尾作为新映射的区域的起始。而新映射的区域的末尾则根据用户参数 `len` 设置。

代码片段 4.32 mmap

```

1 else {
2     if let Some(idx) = idx {
3         let area = &mut self.areas[idx];
4         if flags.contains(MapFlags::MAP_PRIVATE | MapFlags::MAP_ANONYMOUS)
5             && prot == area.map_perm
6             && area.map_file.is_none()
7     }
```

```

8     let end_va: VirtAddr = area.inner.vpn_range.get_end().into();
9     area.expand_to(VirtAddr::from(end_va.0 + len)).unwrap();
10    return end_va.0 as isize;
11 }
12 area.inner.vpn_range.get_end().into()
13 } else {
14     MMAP_BASE.into()
15 }
16 };
17 let mut new_area = MapArea::new(
18     start_va,
19     VirtAddr::from(start_va.0 + len),
20     MapType::Framed,
21     prot,
22     None,
23 );

```

第三步：mmap 会判断是否是文件映射，即是否包含 MAP\_ANONYMOUS。如果没有包含，就会先获取 fd\_table，并根据用户参数 fd 从 fd\_table 找到对应的文件描述符，这里文件相关内容不作赘述。然后设置文件偏移量 offset 并置入 new\_area 中

代码片段 4.33 mmap

```

1 if !flags.contains(MapFlags::MAP_ANONYMOUS) {
2     let fd_table = task.files.lock();
3     match fd_table.get_ref(fd) {
4         Ok(file_descriptor) => {
5             if !file_descriptor.readable() {
6                 return EACCES;
7             }
8             let file = file_descriptor.file.deep_clone();
9             file.lseek(offset as isize, SeekWhence::SEEK_SET).unwrap();
10            new_area.map_file = Some(file);
11        }
12        Err(errno) => return errno,
13    }
14 }

```

第四步：mmap 将 new\_area 加入到先前的 vector 中，而且需要保持原 vector 的有序性，这里使用了 rust 语言的特性。

代码片段 4.34 mmap

```

1 let (idx, _) = self
2 .areas
3 .iter()
4 .enumerate()
5 .skip_while(|(_, area)| {
6     area.inner.vpn_range.get_start() >= VirtAddr::from(MMAP_END).into()
7 })
8 .find(|(_, area)| area.inner.vpn_range.get_start() >= start_va.into())
9 .unwrap();
10 self.areas.insert(idx, new_area);

```

#### 4.5.3 execve

应用程序自身的角度来看，进程 (Process) 的一个经典定义是一个正在运行的程序实例。当程序运行在操作系统中的时候，从程序的视角来看，它会产生一种“幻觉”：即该程序是整个计算机系统中当前运行的唯一的程序，能够独占使用处理器、内存和外设，而且程序中的代码和数据是系统内存中唯一的对象。体现表现为“进程”这个抽象概念。站在计算机系统和操作系统的角度来看，并不存在这种“幻觉”。事实上，在一段时间之内，往往会有多个程序同时或交替在操作系统上运行，因此程序并不能独占整个计算机系统。具体而言，进程是应用程序的一次执行过程。并且在这个执行过程中，由“操作系统”执行环境来管理程序执行过程中的 \*\* 进程上下文 \*\* -一种控制流上下文。这里的进程上下文是指程序在运行中的各种物理/虚拟资源（寄存器、可访问的内存区域、打开的文件、信号等）的内容，特别是与程序执行相关的内容：内存中的代码和数据，栈、堆、当前执行的指令位置（程序计数器的内容）、当前执行时刻的各个通用寄存器中的值等。

exec 用一个新的程序来代替当前进程的内存和寄存器，但是其文件描述符、进程 id 和父进程都是不变的。它根据文件系统中保存的某个文件来初始化用户部分。‘exec‘通过 ‘open()‘ 打开二进制文件。然后，它读取 ELF 头。应用程序以通行的 ELF 格式来描述。一个 ELF 二进制文件包括了一个 ELF 头，然后是连续几个程序段的头。exec 会检查文件是否包含 ELF 二进制代码。一个 ELF 二进制文件是以 4 个“魔法数字”开头的，即 0x7F，“E”，“L”，“F”。如果 ELF 头中包含正确的魔法数字，‘exec‘ 就会认为该二进制文件的结构是正确的。

代码片段 4.35 execve

```

1 pub fn sys_execve(
2     pathname: *const u8,
3     mut argv: *const *const u8,
4     mut envp: *const *const u8,
5 ) -> isize

```

execve() 的前置条件 fork() 复制某一个进程，为 execve() 的执行，准备条件。execve() 的核心工作：参数转换，将 argv,envp 转为 Vec<String>。open() 打开 path 路径下的可执行文件，对其进行检查。load\_elf() 将其加载到当前的空间中。load\_elf 的核心工作：将 elf 文件映射到内核空间的 ‘MMAP\_BASE‘ 地址处。在 map\_elf() 中，把 elf 文件的内容拷贝到用户的 MemorySet 中

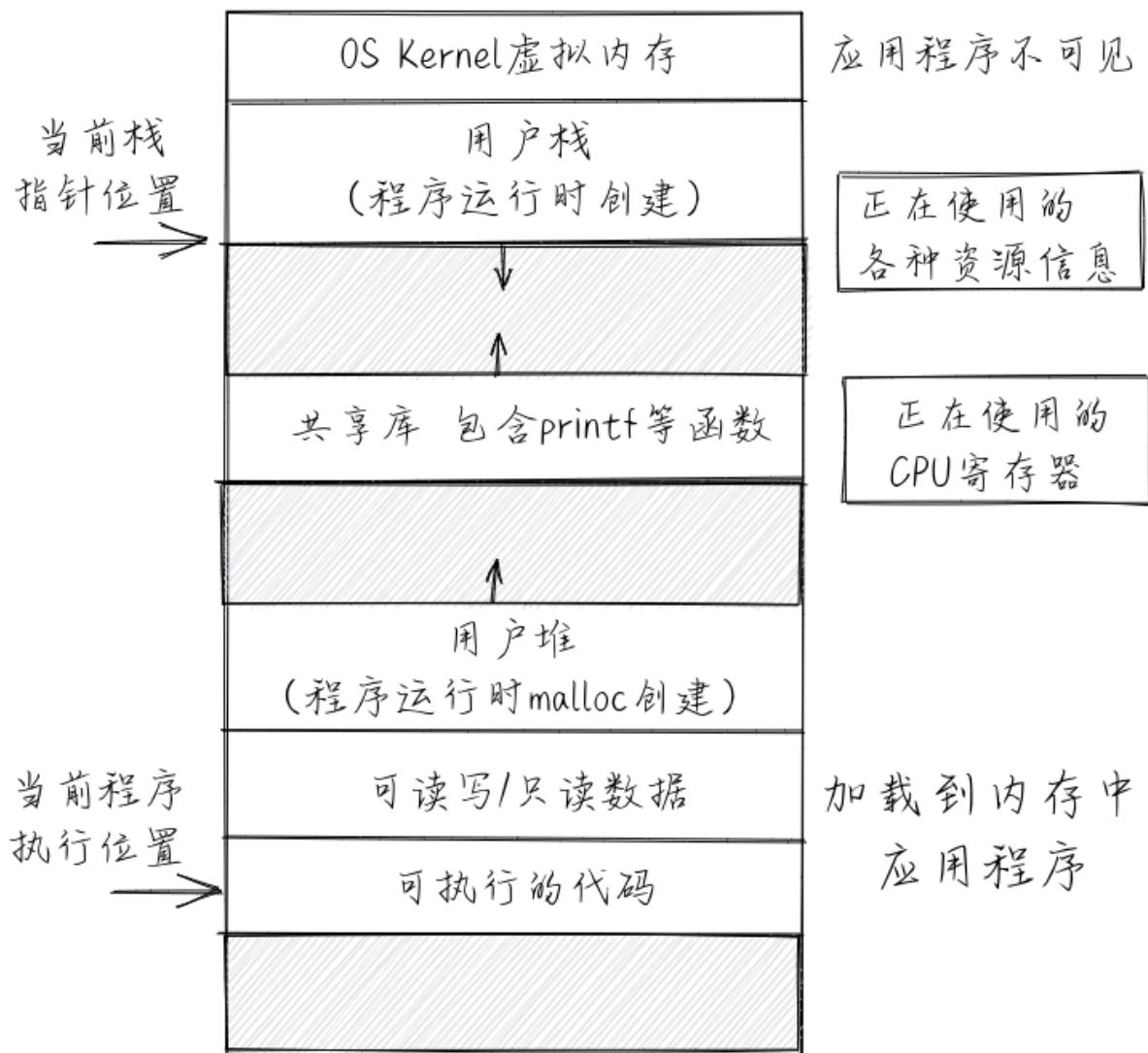


图 4-14 进程的地址空间

## 第 5 章 进程管理

### 5.1 进程生命周期和资源复用

#### 5.1.1 进程生命周期

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

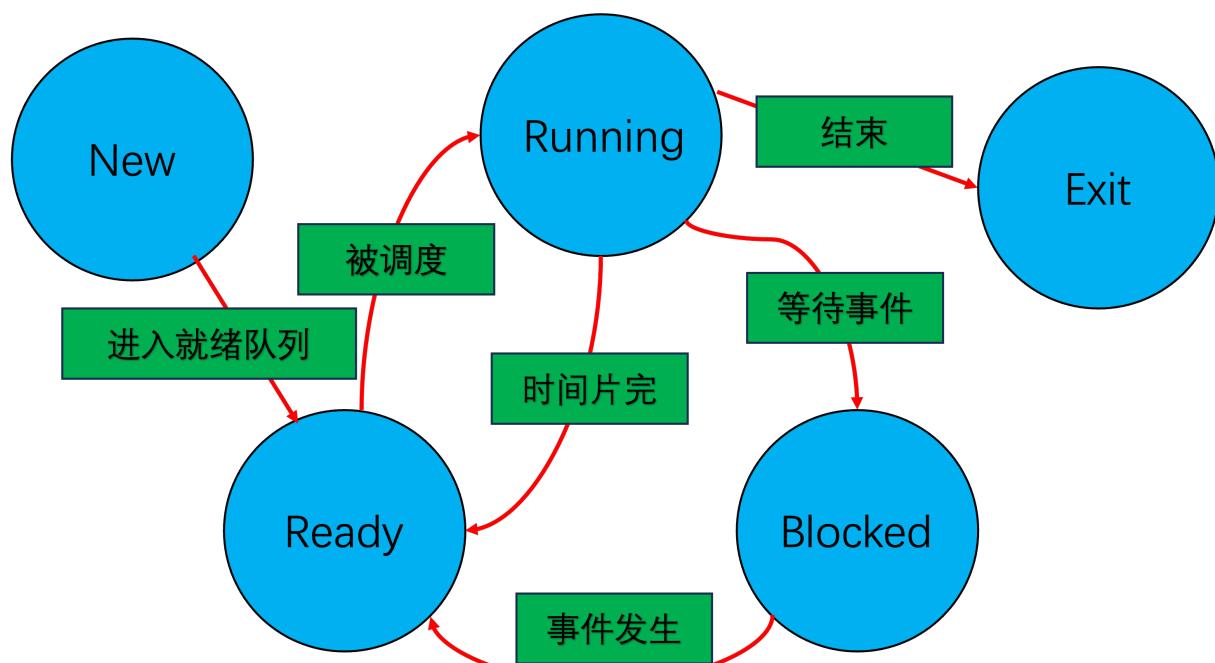


图 5-1 进程生命周期示意图

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 团队在进行性能调优之时，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，使其完全支持了阻塞式的进程调度模式。阻塞和非阻塞 IO 是访问设备的两种模式，驱动程序可以灵活的支持者两种用户空间对设备的访问方式。阻塞操作是指在执行设备操作时，若不能获得资源，则挂起进程直到满足可操作的条件后再进行操作。被挂起的进程进入睡眠状态，被调度器的运行队列移走，直到等待的条件被满足。非阻塞是指在不能进行设备操作时，并不挂起，他要么放弃，要么不停地查询，直到可以进行操作为止。在阻塞访问时，不能获取资源的进程将进入休眠，它将会让出 CPU，因为阻塞的进程会进入休眠状态，所以必须要有一个动作能唤醒该进程，唤醒进程的地方最大的可能发生在中断里面，因为

在硬件资源获得的同时往往伴随着一个中断。而非阻塞的进程则不断的尝试，直到可以进行 IO。与 linux 操作系统的设计类似，npucore 采用等待队列的方式实现阻塞式调度器，将在后续章节介绍。为了可以在 npucore 上同时运行多个进程，npucore 实现了进程的创建，在内核中加载进程到内存，同时为其分配系统资源。npucore 为每一个进程分配系统资源，包括内存、文件描述符、CPU 等，而实现系统资源的分配的方式是通过系统调用 fork。npucore 中，fork 系统调用用于创建一个新的进程，新的进程称为子进程，原来的进程称为父进程。而所有的其他进程都是 initproc 的子进程，他们通过 fork 得到。initproc 是需要在内核启动过程中创建的第一个进程。对应的代码如下：

```

1 lazy_static! {
2     pub static ref INITPROC: Arc<TaskControlBlock> = Arc::new({
3         let elf = ROOT_FD.open("initproc", OpenFlags::O_RDONLY, true).
4             unwrap();
5         TaskControlBlock::new(elf)
6     });
}

```

当创建一个新的进程时，用户进程通过 fork 得到一个原本进程的副本，为其分配系统资源，然后调用 execve 来讲 elf 文件加载到内存以创建一个新的进程。每个进程都有自己的内存空间、代码和数据，它们是系统中资源的分配单位。他们的创建是由 elf 文件指定的。

为了保证所有的进程都能够被调度，从而避免进程饥饿的发生，npucore 实现了进程的调度机制，从而实现阻塞和唤醒。当一个进程主动放弃 CPU 或者被动的被剥夺 CPU 的使用权时，它会让出 CPU，变成等待状态，这个过程称为阻塞。在进程的视角看来，他会有一个自己独占 CPU 的“幻觉”，因为每一个阻塞和唤醒的时候进程的状态总是保持不变的。这样可以保证进程执行的正确性。而 npucore 让一个被阻塞的进程重新开始执行的行为叫做唤醒。唤醒的同时会恢复进程的现场，包括阻塞时的寄存器状态。

而当一个进程执行结束，它就会退出，将它所占有的系统资源释放。进程的退出保证了 npucore 避免出现资源的永久占用的情况。上述过程就是一个进程从“生”到“死”，保证了 npucore 可以正确且高效的执行对应的程序。

### 5.1.2 资源复用

为了实现多进程同时运行，操作系统需要对 CPU，内存，外设等资源进行复用。复用在资源有限的情况下是一个常用且实际的思想。围绕着复用的思想，我们可以提出以下几个问题：

如何实现上下文切换？

虽然保存现场思想是简单的，但是实际的实现却不是那么显然。在 npucore 中我们使用了一段所有进程共享的跳板代码和一个进程的私有的保存现场的帧来实现。

如何让进程如何实现透明调度，也就是用户进程不知道自己被调度了？

npucore 实现了内置的计时器，当计时器中断发生时，内核会调用 schedule 函数，从

而实现进程的调度。

进程的资源回收不能由进程自己来完成，否则进程退出时会出现资源泄露的情况，如何实现进程的资源回收？

npucore 在 exit 之后，会释放一部分资源，但是不会释放所有的资源，从而进入僵尸状态，父进程来完成剩下的进程资源的回收。

如何在并发的情况下不会错过对进程的唤醒？

npucore 中是一个单核的操作系统，在进入关键代码的时候会保证 CPU 不会调度其他进程，从而保证了进程的唤醒不会被错过。

此外，在内存方面，npucore 采用了虚拟内存的方式，将物理内存映射到虚拟内存，从而实现了内存的复用。对于用户的 elf 程序，程序的入口总是相同的，从某种程度上讲，程序将会“共享”同一个地址。但是实际上，物理地址并不能被共享，所以虚拟内存就派上了用场。在上一章我们已经介绍了虚拟内存的实现，这里不再赘述。I/O 设备的复用将在 I/O 章节详细介绍。在这一章中，我们将介绍进程所拥有的各种内核的资源，诸如文件描述符等。

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：

```

1 pub struct TaskControlBlock {
2     // immutable
3     pub pid: PidHandle,
4     pub tid: usize,
5     pub tgid: usize,
6     pub kstack: KernelStack,
7     pub ustack_base: usize,
8     pub exit_signal: Signals,
9     // mutable
10    inner: Mutex<TaskControlBlockInner>,
11    // shareable and mutable
12    pub exe: Arc<Mutex<FileDescriptor>>,
13    pub tid_allocator: Arc<Mutex<RecycleAllocator>>,
14    pub files: Arc<Mutex<FdTable>>,
15    pub fs: Arc<Mutex<FsStatus>>,
16    pub vm: Arc<Mutex<MemorySet>>,
17    pub sighand: Arc<Mutex<Vec<Option<Box<SigAction>>>>>,
18    pub futex: Arc<Mutex<Futex>>,
19 }
```

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。

## 5.2 进程状态控制

### 5.2.1 基本概念

在操作系统课程的学习中我们知道，进程其实是一个“执行中的程序”。程序是一个没有生命的实体，只有在操作系统执行静态的、放在磁盘中的代码段时，它才能成为一个活动的实体，我们称其为进程。

在运行过程中，进程是一个实体。每一个进程都有它自己的地址空间，一般情况下，包括文本区域 (text region)、数据区域 (data region) 和堆栈 (stack region)。文本区域存储处理器执行的代码；数据区域存储变量和进程执行期间使用的动态分配的内存；堆栈区域存储着活动过程调用的指令和本地变量。

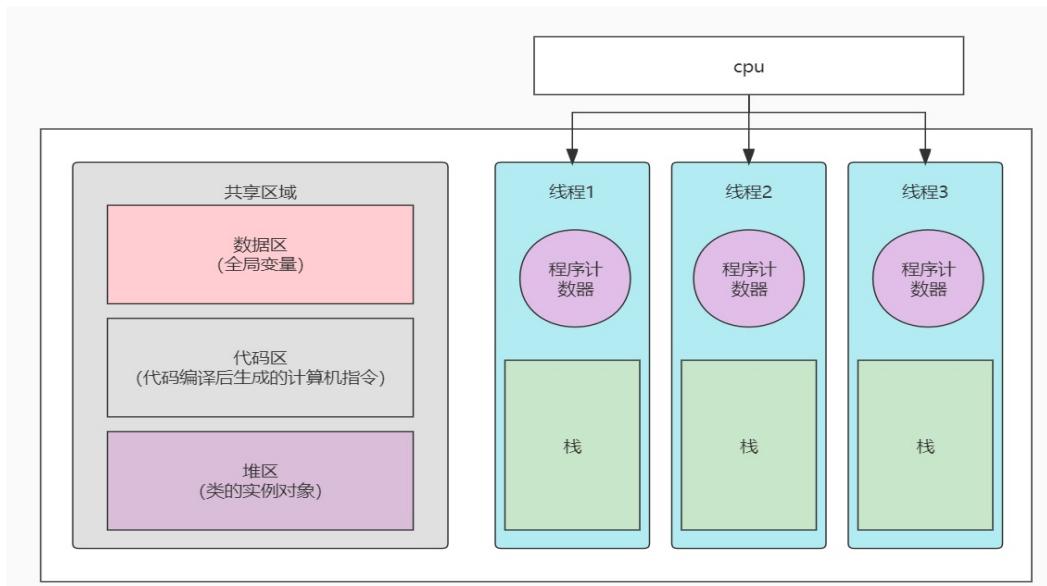


图 5-2 进程的数据结构

进程的创建、销毁与切换存在着较大的时空开销，一种轻型的进程技术线程用来减少开销。线程被设计成进程的一个执行路径，同一个进程中的线程共享进程的资源，因此系统对线程的调度所需的成本远远小于进程。

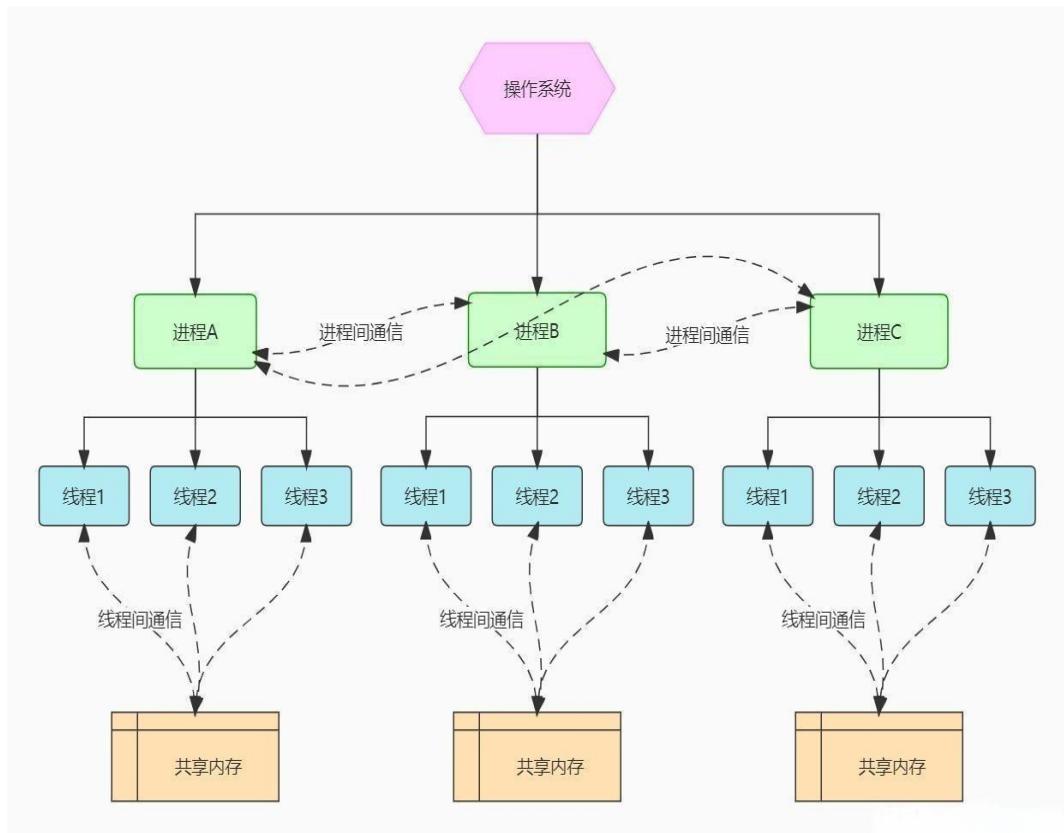


图 5-3 线程与进程间的关系

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。

NPUCore 进程管理的代码树如下：

```

1  os
2  |--src
3  |--task
4  |--context.rs //存储上下文信息
5  |--elf.rs //ELF文件加载和解析
6  |--manager.rs //任务管理器的结构存储
7  |--mod.rs//任务操作系统任务管理和调度的实现
8  |--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
9  |--processor.rs // 进程轮询的主体和processor的声明与实现
10 |--signal.rs //
11 |--switch.rs //调用switch汇编函数的接口,实现任务上下文切换
12 |--switch.S //任务上下文切换的汇编实现
13 |--threads.rs//实现用户空间多线程快速互斥锁(Futex)
14 |--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。

```

## 5.2.2 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。

```
1 //os/src/task/pid.rs
2 pub struct PidHandle(pub usize);
```

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。

```
1 //os/src/task/pid.rs
2 pub struct KernelStack(pub usize);
```

它提供了以下方法：

1. 返回当前内核栈在内核空间中的栈顶和栈底位置：

```
1 pub fn kernel_stack_position(kstack_id: usize) -> (usize, usize) {
2     let top = TRAMPOLINE - kstack_id * (KERNEL_STACK_SIZE + PAGE_SIZE);
3     let bottom = top - KERNEL_STACK_SIZE;
4     (bottom, top)
5 }
```

2. 内核栈分配器：

```
1 pub fn kstack_alloc() -> KernelStack {
2     let kstack_id = KSTACK_ALLOCATOR.lock().alloc();
3     let (kstack_bottom, kstack_top) = kernel_stack_position(kstack_id);
4     KERNEL_SPACE.lock().insert_framed_area(
5         kstack_bottom.into(),
6         kstack_top.into(),
7         MapPermission::R | MapPermission::W, );
8     KernelStack(kstack_id)
9 }
```

该分配器先从全局实例化的 KSTACK\_ALLOCATOR 中获取内核栈编号。它调用了

kernel\_stack\_position 函数来根据进程标识符计算内核栈在内核地址空间中的位置，随即  
将一个逻辑段插入内核地址空间 KERNEL\_SPACE 中（详情见内存管理章节）。

### 3. 将内核栈从内核空间中移除：

```

1  impl Drop for KernelStack {
2      fn drop(&mut self) {
3          let (kernel_stack_bottom, _) = kernel_stack_position(self.0);
4          let kernel_stack_bottom_va: VirtAddr = kernel_stack_bottom.into()
5              ();
6          KERNEL_SPACE
7              .lock()
8              .remove_area_with_start_vpn(kernel_stack_bottom_va.into())
9              .unwrap();
10             KSTACK_ALLOCATOR.lock().dealloc(self.0)
11     }
12 }
```

### 4.push\_on\_top 方法：

```

1  impl KernelStack{
2      #[allow(unused)]
3      pub fn push_on_top<T>(&self,value:T) -> *mut T
4      where
5          T:Sized,
6      {
7          let kernel_stack_top = self.get_top();
8          let ptr_mut = (kernel_stack_top - core::mem::size_of::<T>()) as
9              *mut T;
10             unsafe {
11                 *ptr_mut = value;
12             }
13             ptr_mut
14     }
15     pub fn get_top(&self) -> usize {
16         let (_,kernel_stack_top) = kernel_stack_position(self.0);
17         kernel_stack_top
18     }
19 }
```

该方法可以将一个类型为 T 的变量压入内核栈顶并返回其裸指针，这也是一个泛型函数。它在实现的时候用到了 get\_top 方法来获取当前内核栈顶在内核地址空间中的地址。

## (3) 进程控制块 PCB

为了使并发执行的程序独立运行，描述进程的基本情况和活动过程，进而管理和控  
制进程，操作系统专门配置了一个数据结构——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。但是在本章中，为了简  
化逻辑，我们尽量只讨论单线程的情况，也就是仅仅把 TCB 看作是进程控制块。

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。

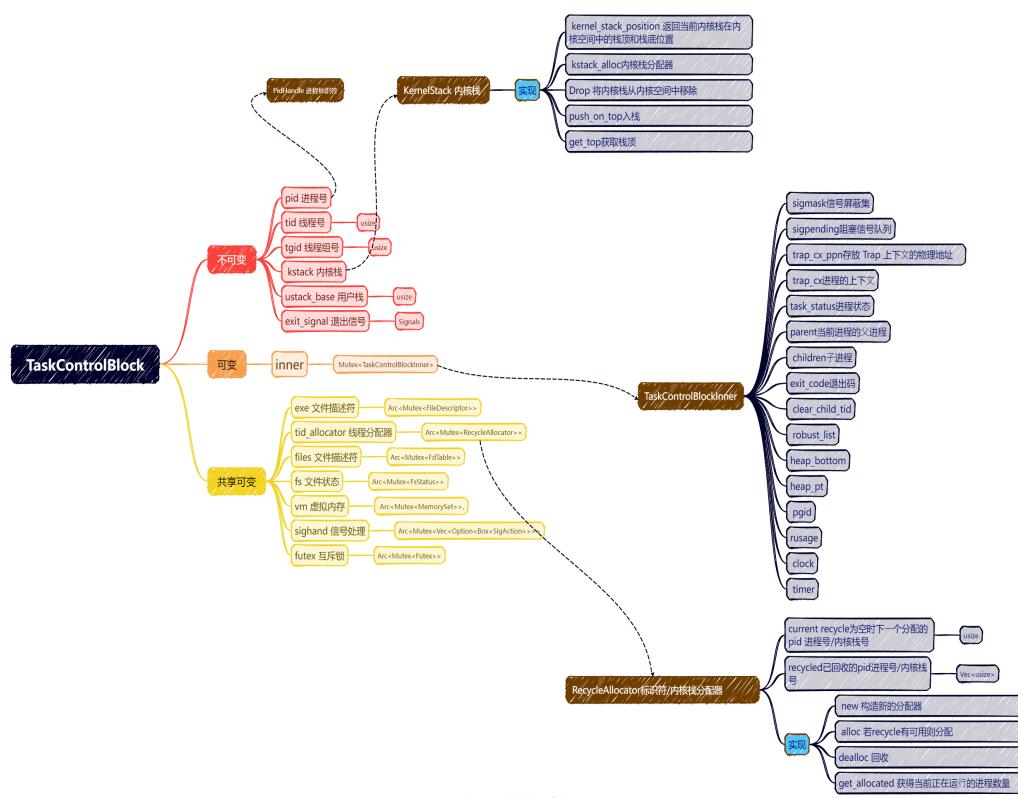


图 5-4 NPUcore 中进程控制块的具体组成

由于该数据结构的成员过多，且很多成员变量都是与信号及多线程的锁相关的，因此这里只挑选部分成员变量进行讲解。

```

1 //os/src/task/task.rs
2 pub struct TaskControlBlock {
3     // immutable
4     pub pid:PidHandle,
5     pub tid:usize,
6     pub tgid:usize,
7     pub kstack:KernelStack,
8     pub ustack_base:usize,
9     pub exit_signal:Signals,
10    // mutable
11    inner:Mutex<TaskControlBlockInner>,
12    // shareable and mutable
13    pub exe:Arc<Mutex<FileDescriptor>>,
14    pub tid_allocator:Arc<Mutex<RecycleAllocator>>,
15    pub files:Arc<Mutex<FdTable>>,
16    pub fs:Arc<Mutex<FsStatus>>,
17    pub vm:Arc<Mutex<MemorySet>>,
18    pub sighand:Arc<Mutex<Vec<Option<Box<SigAction>>>>>,
19    pub futex:Arc<Mutex<Futex>>,
20 }
21 pub struct TaskControlBlockInner {
22     pub sigmask:Signals,
23     pub sigpending:Signals,
24     pub trap_cx_ppn:PhysPageNum,
25     pub task_cx:TaskContext,
26     pub task_status:TaskStatus,
27     pub parent:Option<Weak<TaskControlBlock>>,
28     pub children:Vec<Arc<TaskControlBlock>>,
29     pub exit_code:u32,
30     pub clear_child_tid:usize,
31     pub robust_list:RobustList,
32     pub heap_bottom:usize,
33     pub heap_pt:usize,
34     pub pgid:usize,
35     pub rusage:Rusage,
36     pub clock:ProcClock,
37     pub timer:[ITimerVal;3],
38 }
```

TCB 由可变部分、不可变部分及可共享且可变部分组成。

### 1. 不可变部分：

(1) pid：进程号在进程创建后就是唯一的身份标识，所以肯定不变。它的类型是

PidHandle

(2) kstack：这是每个应用的内核栈。

(3) ustack base: 用户栈的基地址

(4) exit signal: 退出信号

### 2. 可变部分：inner : Mutex<TaskControlBlockInner>

TaskControlBlockInner 结构体的定义如下：

```

1 pub struct TaskControlBlockInner {
2     pub sigmask: Signals,
3     pub sigpending: Signals,
4     pub trap_cx_ppn: PhysPageNum,
5     pub task_cx: TaskContext,
6     pub task_status: TaskStatus,
7     pub parent: Option<Weak<TaskControlBlock>>,
8     pub children: Vec<Arc<TaskControlBlock>>,
9     pub exit_code: u32,
10    pub clear_child_tid: usize,
11    pub robust_list: RobustList,
12    pub heap_bottom: usize,
13    pub heap_pt: usize,
14    pub pgid: usize,
15    pub rusage: Rusage,
16    pub clock: ProcClock,
17    pub timer: [ITimerVal; 3],
18 }

```

同上，我们也不介绍信号与锁相关的成员变量。

`trap_cx_ppn`：这个东西的作用是把存放 Trap 上下文的物理地址拿出来，从而便于使用

```

1 let trap_cx_ppn = memory_set
2 .translate(VirtAddr::from(TRAP_CONTEXT).into())
3 .unwrap()
4 .ppn();

```

`task_cx`: 类型是 `TaskContext` 是进程的上下文

```

1 pub struct TaskContext {
2     /// return address (e.g. __restore ) of __switch ASM function
3     ra: usize,
4     /// kernel stack pointer of app
5     sp: usize,
6     /// s0-11 register, callee saved
7     s: [usize; 12],
8 }

```

`TaskManager`: 进程管理模块，维护两个循环队列，一个是就绪态队列，一个是阻塞态任务队列，队列中的内容都是 TCB 模块的指针。

```

1 pub struct TaskManager {
2     pub ready_queue:
3         VecDeque<Arc<TaskControlBlock>>,
4     pub interruptible_queue:
5         VecDeque<Arc<TaskControlBlock>>,
6 }

```

`ready_queue`: 就绪态队列

`interruptible_queue`: 阻塞态任务队列

`Processor`: 用于保存当前任务的 arc 指针和空闲的任务上下文（用于进程切换）。

```

1 pub struct Processor {

```

```

1 current:Option<Arc<TaskControlBlock>>,
2   idle_task_cx:TaskContext,
3 }
4

```

current 表示在当前处理器上正在执行的任务 idle\_task\_cx 表示当前处理器上的 idle 控制流的任务上下文

**TaskStatus:** 基本的任务状态枚举，包括就绪态，运行态，僵尸态（等待回收），阻塞态（等待激活以进入就绪态）。

```

1 pub enum TaskStatus {
2   Ready,
3   Running,
4   Zombie,
5   Interruptible,
6 }

```

说明进程有四种状态：就绪、正在运行、“僵尸”状态(表示该进程即将被回收)以及可中断状态。

parent 是 TCB 的一个弱引用，他表示当前进程的父进程，同理 children 表示子进程 exit\_code，当进程调用 exit 系统调用主动退出或者执行出错由内核终止的时候，它的退出码 exit\_code 会被内核保存在它的任务控制块中，并等待它的父进程通过 waitpid 回收它的资源的同时也收集它的 PID 以及退出码。

#### (4) 标识符/内核分配器 RecycleAllocator

对于进程标识符和内核栈，NPUCore 都使用 RecycleAllocator 分配器来分配和回收。

```

1 //os/src/task/pid.rs
2 pub struct RecycleAllocator {
3   current:usize,
4   recycled:Vec<usize>,
5 }

```

current 为若 recycled 数组为空时下一个分配的 pid 进程号/内核栈号，每次分配后都会 +1；recycled 数组保存了当前已回收的 pid 进程号/内核栈号。

该分配器实现了以下四个方法：

```

1 impl RecycleAllocator {
2   pub fn new() -> Self {
3     RecycleAllocator {
4       current:0,
5       recycled:Vec::new(),
6     }
7   }
8   pub fn alloc(&mut self) -> usize {
9     if let Some(id) = self.recycled.pop() {
10       id
11     } else {
12       self.current += 1;
13       self.current - 1
14     }
}

```

```

15 }
16 pub fn deallocate(&mut self, id: usize) {
17     assert!(id < self.current);
18     assert!(
19         !self.recycled.iter().any(|i| *i == id),
20         "id {} has been deallocated!",
21         id
22     );
23     self.recycled.push(id);
24 }
25 pub fn get_allocated(&self) -> usize {
26     self.current - self.recycled.len()
27 }

```

**new()**: 构造方法。只会在第一次实例化时调用。

**alloc(&mut self)**: 分配器。若 recycled 数组中有可使用的 pid 号/内核栈号，则直接分配，否则生成一个新的 pid 号。

**deallocate(&mut self, id: usize)**: 回收器。需要检查合法性才会回收。

**get\_allocated(&self)**: 获得当前正在运行的进程数量。

有了分配器后，我们可以将进程分配器分别实例化，

实例化进程标识符分配器和内核栈分配器：

```

1 lazy_static!{
2     static ref PID_ALLOCATOR: Mutex<RecycleAllocator> = Mutex::new(
3         RecycleAllocator::new()
4     )
5 }

```

注意，这里的结构体 RecycleAllocator 并不仅仅是 PID 分配器，还可以做内核栈分配器。

分配器将会分配出去一个 PidHandle，将其包装为一个全局分配进程标识符的接口 pid\_alloc 提供给内核的其他子模块，并为 PidHandle 实现 Drop Trait 来允许编译器进行自动的资源回收：

```

1 pub fn pid_alloc() -> PidHandle {
2     PidHandle(PID_ALLOCATOR.lock().alloc())
3 }
4 impl Drop for PidHandle {
5     fn drop(&mut self) {
6         PID_ALLOCATOR.lock().deallocate(self.0);
7     }
8 }

```

### 5.2.3 进程创建

execve 被用于替换当前进程的地址空间和上下文为新程序的，新程序将取代原程序执行。如果 execve 执行成功，原程序的代码和数据将被替换为新程序的代码和数据，并开始执行新程序。filename: 要执行的新程序的文件路径。argv: 参数数组，用于传递给

新程序的命令行参数。envp: 环境变量数组，用于设置新程序的环境变量。

#### 代码片段 5.1 参数说明

```

1 pub fn sys_execve(
2     pathname: *const u8,
3     mut argv: *const *const u8,
4     mut envp: *const *const u8,
5 ) -> isize {}

```

在 NPUCore 中，若需要创建一个新的进程，总体上的过程和方法如下所示：

- (1) 从系统文件中找到内核加载第一个初始进程的 elf 文件，获取代码的数据和内容。然后调用 TCB 中的 new() 方法创建内核的第一个进程 initproc。
- (2) 其余所有的进程均由初始进程 initproc 进行 fork 而来，初始进程 initproc 是所有进程的父进程。因此创建新进程的第二步便是调用 fork 系统调用
- (3) 在调用 fork 后，新的进程还需要加载独立的程序代码和数据文件，这时便需要用到 exec 系统调用。

exec() 的作用：

fork 通常只创建现有进程的拷贝 exec 可以在现有 fork 的基础上加载一个新应用的 ELF 可执行文件中的代码和数据替换原有的应用地址空间中的内容，并开始执行。

#### 1. 首先看函数的输入参数与返回值

#### 代码片段 5.2 函数的输入参数与返回值

```

1 pub fn sys_execve(
2     pathname: *const u8,
3     mut argv: *const *const u8,
4     mut envp: *const *const u8,
5 ) -> isize {
6     //load_elf 成功
7     SUCCESS
8     //失败
9     Err
10 }

```

2. 将 argv 和 envp 变量字符串向量，通过循环，将 \*const \*const u8 转为 Vec<String> 数据类型。为后面的执行做准备。

#### 代码片段 5.3 类型转换

```

1 let mut argv_vec: Vec<String> = Vec::with_capacity(16);
2 let mut envp_vec: Vec<String> = Vec::with_capacity(16);
3 if !argv.is_null() {
4     loop {
5         let arg_ptr = match translated_ref(token, argv) {
6             Ok(argv) => *argv,
7             Err(errno) => return errno,
8         };
9         if arg_ptr.is_null() {
10             break;
11         }
12         argv_vec.push(match translated_str(token, arg_ptr) {

```

```

13         Ok(arg) => arg,
14         Err(errno) => return errno,
15     );
16     unsafe {
17         argv = argv.add(1);
18     }
19 }
20 }
21 if !envp.is_null() {
22     loop {
23         let env_ptr = match translated_ref(token, envp) {
24             Ok(envp) => *envp,
25             Err(errno) => return errno,
26         };
27         if env_ptr.is_null() {
28             break;
29         }
30         envp_vec.push(match translated_str(token, env_ptr) {
31             Ok(env) => env,
32             Err(errno) => return errno,
33         });
34         unsafe {
35             envp = envp.add(1);
36         }
37     }
38 }

```

### 3.debug 层信息输出

代码片段 5.4 debug 信息

```

1 debug!(
2     "[exec] argv: {:+?} /* {} vars */, envp: {:+?} /* {} vars */",
3     argv_vec,
4     argv_vec.len(),
5     envp_vec,
6     envp_vec.len()
7 );

```

4. 准备将 ELF 文件的内容载入当前进程要对读取的文件做以下检查 1、文件大小大于等于 42、同时，ELF 文件以 4 位魔数\x7fELF开头，因此应检查文件首是否有此魔数。

代码片段 5.5 文件检查

```

1 match working_inode.open(&path, OpenFlags::O_RDONLY, false) {
2     Ok(file) => {
3         if file.get_size() < 4 {
4             return ENOEXEC;
5         }
6         let mut magic_number = Box::<[u8; 4]>::new([0; 4]);
7         // this operation may be expensive... I'm not sure
8         file.read(Some(&mut 0usize), magic_number.as_mut_slice());
9         let elf = match magic_number.as_slice() {
10             b"\x7fELF" => file,
11             b"#!" => {
12                 let shell_file = working_inode
13                     .open(DEFAULT_SHELL, OpenFlags::O_RDONLY, false
14                         )
15             }
16         };
17     }
18 }

```

```

14         .unwrap();
15         argv_vec.insert(0, DEFAULT_SHELL.to_string());
16         shell_file
17     }
18     _ => return ENOEXEC,
19 };

```

5. 真正的载入过程。loaf\_elf 将当前的 elf 文件内容覆盖当前的进程。show\_frame\_consumption! 宏输出对应的信息

代码片段 5.6 文件载入

```

1 let task = current_task().unwrap();
2 show_frame_consumption! {
3     "load_elf";
4     if let Err(errno) = task.load_elf(elf, &argv_vec, &envp_vec) {
5         return errno;
6     };
7 }
8 // should return 0 in success
9 SUCCESS
10 }
11 Err(errno) => errno,
12 }

```

clone() 从 linux 2.3.3 开始，glibc 的 ‘fork()‘封装作为 NPTL(Native POSIX Threads Library) 线程实现的一部分。直接调用 ‘fork()‘等效于调用 [clone(2)](<https://man7.org/linux/man-pages/man2/clone.2.html>) 时仅指定 ‘flags‘为 ‘SIGCHLD‘(共享信号句柄表)。

创建线程的函数 ‘pthread\_create‘内部使用的也是 clone 函数。在 glibc 的 ‘/sysdeps/unix/sysv/linux/createthread.c‘源码中可以看到，创建线程的函数 ‘create\_thread‘中使用了 clone 函数，并指定了相关的 ‘flags‘：

代码片段 5.7 FLAGS 设置

```

1 const int clone_flags = (CLONE_VM | CLONE_FS
2 | CLONE_FILES | CLONE_SYSVSEM
3 | CLONE_SIGHAND | CLONE_THREAD
4 | CLONE_SETTLS | CLONE_PARENT_SETTID
5 | CLONE_CHILD_CLEARTID | 0);

```

同样，npucore 中的 ‘clone‘也使用了这种模式，以这两种方式来创建进程和线程

### clone 的使用

代码片段 5.8 clone

```

1 pub fn sys_clone(
2     flags: u32,
3     stack: *const u8,
4     ptid: *mut u32,
5     tls: usize,
6     ctid: *mut u32,
7 ) -> isize

```

描述该系统调用用于创建一个新的子进程，类似 fork(2)。与 fork(2) 相比，它可以更精确地控制调用进程和子进程之间的执行上下文细节。例如，使用这些系统调用，调用者可以控制两个进程之间是否共享虚拟地址空间，文件描述符表以及信号句柄表等。也可以通过这些系统调用将子进程放到不同的命名空间中。参数 flags: 包括 CloneFlags 和 exit\_signal 两部分：子进程结束信号 exit\_signal 当子进程退出时，会像父进程发送一个信号。退出信号在 clone() 的 flags 的低字节中指定如果该信号不是 SIGCHLD，那么父进程在使用 wait(2) 等待子进程退出时必须指定 \_\_WALL 或 WCLONE 选项。如果没有指定任何信号（即，0），则在子进程退出后不会向父进程发送任何信号。以下是 npucore 中如何从 flags 中获取子进程结束信号的代码块

代码片段 5.9 获取子进程结束信号的代码块

```

1 let exit_signal = match Signals::from_signum((flags & 0xff) as usize) {
2     Ok(signal) => signal,
3     Err(_) => {
4         warn!(
5             "[sys_clone] signum of exit_signal is unspecified or
6             invalid: {}",
7             (flags & 0xff) as usize
8         );
9         // This is permitted by standard, but we only support 64
10        signals
11        Signals::empty()
12    }
13 };

```

CloneFlags CloneFlags 用来指定 clone 系统调用的行为，npucore 的 CloneFlags 中有如下几种值

代码片段 5.10 CloneFlags

```

1 bitflags! {
2     pub struct CloneFlags: u32 {
3         //const CLONE_NEWTIME      = 0x00000080;
4         const CLONE_VM           = 0x000000100;
5         const CLONE_FS            = 0x000000200;
6         const CLONE_FILES          = 0x000000400;
7         const CLONE_SIGHAND        = 0x000000800;
8         const CLONE_PIDFD          = 0x00001000;
9         const CLONE_PTRACE          = 0x00002000;
10        const CLONE_VFORK           = 0x00004000;
11        const CLONE_PARENT          = 0x00008000;
12        const CLONE_THREAD          = 0x00010000;
13        const CLONE_NEWNS           = 0x00020000;
14        const CLONE_SYSVSEM          = 0x00040000;
15        const CLONE_SETTLS          = 0x00080000;
16        const CLONE_PARENT_SETTID      = 0x00100000;
17        const CLONE_CHILD_CLEARTID      = 0x00200000;
18        const CLONE_DETACHED          = 0x00400000;
19        const CLONE_UNTRACED          = 0x00800000;
20        const CLONE_CHILD_SETTID      = 0x01000000;
21        const CLONE_NEWCGROUP          = 0x02000000;
22        const CLONE_NEWUTS           = 0x04000000;

```

```

23     const CLONE_NEWIPC          = 0x08000000;
24     const CLONE_NEWUSER         = 0x10000000;
25     const CLONE_NEWPID          = 0x20000000;
26     const CLONE_NEWWNET         = 0x40000000;
27     const CLONE_IO              = 0x80000000;
28 }
29 }
```

下文将介绍几个和传入参数有关的 cloneflags stack: stack 参数指定了子进程使用的栈的位置。由于子进程和调用进程可能会共享内存，因此不能在调用进程的栈中运行子进程。调用进程必须为子进程的栈配置内存空间，并向 clone() 传入一个执行该空间的指针。运行的所有处理器的栈都是向下生长的，因此 stack 通常指向为子进程栈设置的内存空间的最顶端地址。注意，clone() 没有为调用者提供一种可以将堆栈区域的大小通知内核的方法。

ptid, ctid: 与 CloneFlags 中的 CLONE\_CHILD\_SETTID 和 CLONE\_PARENT\_SETTID 有关 CLONE\_CHILD\_SETTID 在 ctid 的位置上保存线程 ID。保存操作会在 clone 调用返回控制到子进程的用户空间前完成。(注意，在 clone 调用返回父进程前，保存操作可能是未完成的，它与是否引入 CLONE\_VM 标志相关) CLONE\_PARENT\_SETTID 在父进程的 ptid 中保存子线程 ID。在 Linux 2.5.32-2.5.48 版本中，有一个标志 CLONE\_SETTID 做了同样的事情。保存操作会在 clone 调用将控制返回给用户空间前完成。tid: 与 CloneFlags 中的 CLONE\_SETTLS 有关 CLONE\_SETTLS 将 TLS(Thread Local Storage) 保存到 tls 字段中。npucore 中 clone 的具体实现

代码片段 5.11 clone 的具体实现

```
1 let child = parent.sys_clone(flags, stack, tls, exit_signal);
```

npucore 中 clone 的具体实现实际是实现在 TCB 结构体的方法 sys\_clone 里，这里就是调用了 parent (也就是 clone 的调用者) 的 sys\_clone，它返回新创建的进程的 TCB 结构体首先，clone 会检查 CloneFlags 中是否有 CLONE\_VM 和 CLONE\_THREAD 如果有 CLONE\_VM，则调用进程和子进程会运行在系统的同一个内存空间中。调用进程或子进程对内存的写操作都可以被对方看到。如果没有设置 CLONE\_VM，则子进程会运行在执行 clone 时的调用进程的一份内存空间的拷贝中。如果有 CLONE\_THREAD，子线程会放到与调用进程相同的线程组中。当一个 clone 调用没有指定 CLONE\_THREAD 时，生成的线程会放到一个新的线程组中，其 TGID 等于该线程的 TID

代码片段 5.12 CLONE\_THREAD 的实现

```

1 let memory_set = if flags.contains(CloneFlags::CLONE_VM) {
2     self.vm.clone()
3 } else {
4     crate::mm::frame_reserve(16);
5     Arc::new(Mutex::new(MemorySet::from_existing_user(
6         &mut self.vm.lock(),
7     )));
}
```

```

8   };
9
10  let tid_allocator = if flags.contains(CloneFlags::CLONE_THREAD) {
11      self.tid_allocator.clone()
12  } else {
13      Arc::new(Mutex::new(RecycleAllocator::new()))
14  };
15
16 // 这里改变了源代码的顺序，此处的tid实为下文声明的
17 if flags.contains(CloneFlags::CLONE_THREAD) {
18     memory_set.lock().alloc_user_res(tid, stack.is_null());
19 }

```

接着 clone 会像操作系统申请 pid, tid, tgid 和内核栈, 因为 npucore 模仿了 linux 的“以轻量级进程代替线程”, 所以这里的 pid 与 tid 相同, 为线程号, 而 tgid 代表该进程的进程号

代码片段 5.13 线程号设置

```

1 // alloc a pid and a kernel stack in kernel space
2 let pid_handle = pid_alloc();
3 let tid = tid_allocator.lock().alloc();
4 let tgid = if flags.contains(CloneFlags::CLONE_THREAD) {
5     self.tgid
6 } else {
7     pid_handle.0
8 };
9 let kstack = kstack_alloc();
10 let kstack_top = kstack.get_top();

```

然后 clone 会进行新进程 TCB 的构造

代码片段 5.14 进程 TCB 的构造

```

1 let task_control_block = Arc::new(TaskControlBlock {
2     pid: pid_handle,
3     tid,
4     tgid,
5     kstack,
6     ustack_base: if !stack.is_null() {
7         stack as usize
8     } else {
9         ustack_bottom_from_tid(tid)
10    },
11    exit_signal,
12    exe: self.exe.clone(),
13    tid_allocator,
14    files: if flags.contains(CloneFlags::CLONE_FILES) {
15        self.files.clone()
16    } else {
17        Arc::new(Mutex::new(self.files.lock().clone()))
18    },
19    fs: if flags.contains(CloneFlags::CLONE_FS) {
20        self.fs.clone()
21    } else {
22        Arc::new(Mutex::new(self.fs.lock().clone()))
23    },

```

```

24     vm: memory_set,
25     sighand: if flags.contains(CloneFlags::CLONE_SIGHAND) {
26         self.sighand.clone()
27     } else {
28         Arc::new(Mutex::new(self.sighand.lock().clone()))
29     },
30     futex: if flags.contains(CloneFlags::CLONE_SYSVSEM) {
31         self.futex.clone()
32     } else {
33         // maybe should do clone here?
34         Arc::new(Mutex::new(Futex::new()))
35     },
36     inner: Mutex::new(TaskControlBlockInner {
37         // inherited
38         pgid: parent_inner.pgid,
39         heap_bottom: parent_inner.heap_bottom,
40         heap_pt: parent_inner.heap_pt,
41         // clone
42         sigpending: parent_inner.sigpending.clone(),
43         // new
44         children: Vec::new(),
45         rusage: Rusage::new(),
46         clock: ProcClock::new(),
47         clear_child_tid: 0,
48         robust_list: RobustList::default(),
49         timer: [ITimerVal::new(); 3],
50         sigmask: Signals::empty(),
51         // compute
52         trap_cx_ppn,
53         task_cx: TaskContext::goto_trap_return(kstack_top),
54         parent: if flags.contains(CloneFlags::CLONE_PARENT)
55             | flags.contains(CloneFlags::CLONE_THREAD)
56         {
57             parent_inner.parent.clone()
58         } else {
59             Some(Arc::downgrade(self))
60         },
61         // constants
62         task_status: TaskStatus::Ready,
63         exit_code: 0,
64     }),
65 });

```

这里对相关 CloneFlags 进行说明 CLONE\_FILES 子进程与父进程共享相同的文件描述符 (file descriptor) 表 CLONE\_FS 子进程与父进程共享相同的文件系统，包括 root、当前目录、umask CLONE\_SIGHAND 子进程与父进程共享相同的信号处理 (signal handler) 表 CLONE\_SYSVSEM 如果设置了该标志，则子进程和调用进程会共享一组 System V semaphore adjustment (semadj) 值 (参见 semop(2))。这种情况下，共享列表会在共享该列表的所有进程之间累加 semadj 值，并且仅当共享列表的最后一个进程终止 (或使用 unshare(2) 停止共享列表) 时才会执行 semaphore adjustments。如果没有设置该标志，则子进程会有一个独立的 semadj 列表，且初始为空。与信号量操作有关。

#### 代码片段 5.15 信号量操作

```

1 files: if flags.contains(CloneFlags::CLONE_FILES) {
2     self.files.clone()
3 } else {
4     Arc::new(Mutex::new(self.files.lock().clone()))
5 },
6
7 fs: if flags.contains(CloneFlags::CLONE_FS) {
8     self.fs.clone()
9 } else {
10    Arc::new(Mutex::new(self.fs.lock().clone()))
11 },
12
13
14 sighand: if flags.contains(CloneFlags::CLONE_SIGHAND) {
15     self.sighand.clone()
16 } else {
17     Arc::new(Mutex::new(self.sighand.lock().clone()))
18 },
19
20
21 futex: if flags.contains(CloneFlags::CLONE_SYSVSEM) {
22     self.futex.clone()
23 } else {
24     // maybe should do clone here?
25     Arc::new(Mutex::new(Futex::new()))
26 },
27

```

这里会修改父进程的子进程表，如果设置了 CLONE\_PARENT 或 CLONE\_THREAD 则代表新建进程与调用进程是兄弟关系，反之则调用进程是新建进程的父进程

代码片段 5.16 修改父进程的子进程表

```

1 // add child
2 if flags.contains(CloneFlags::CLONE_PARENT) || flags.contains(
3     CloneFlags::CLONE_THREAD) {
4     if let Some(grandparent) = &parent_inner.parent {
5         grandparent
6             .upgrade()
7             .unwrap()
8             .acquire_inner_lock()
9             .children
10            .push(task_control_block.clone());
11     }
12 } else {
13     parent_inner.children.push(task_control_block.clone());
14 }

```

最后，clone 会设置新进程的上下文以及 tls 并返回

代码片段 5.17 设置新进程的上下文

```

1 let trap_cx = task_control_block.acquire_inner_lock().get_trap_cx();
2 if flags.contains(CloneFlags::CLONE_THREAD) {
3     *trap_cx = *parent_inner.get_trap_cx();
4 }

```

```

5 // we also do not need to prepare parameters on stack, musl has done it
6   for us
7 if !stack.is_null() {
8     trap_cx.gp.sp = stack as usize;
9 }
10 // set tp
11 if flags.contains(CloneFlags::CLONE_SETTLS) {
12   trap_cx.gp.tp = tls;
13 }
14 // for child process, fork returns 0
15 trap_cx.gp.a0 = 0;
16 // modify kernel_sp in trap_cx
17 trap_cx.kernel_sp = kstack_top;
18 // return
19 task_control_block

```

### 5.2.4 进程切换

进程切换是操作系统中的一项核心功能，其作用是在多任务操作系统中实现进程间的切换，使得每个进程都可以独立地运行并且感觉到自己在独占 CPU。

实现进程的切换具有以下几个重要作用：

1. 提高系统的并发性：进程切换可以使得多个进程在同一时刻并发运行，从而提高系统的效率和资源利用率。
2. 实现多任务操作系统的功能：进程切换是多任务操作系统的功能之一，它使得每个进程都独立运行，并且感觉到自己在独占 CPU。

总之，进程切换是操作系统中一项重要的功能，它可以提高系统的并发性和效率，支持多任务和多用户系统，并且是操作系统实现任务调度、资源分配等核心功能的基础。

#### (1) 进程切换的基本流程

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

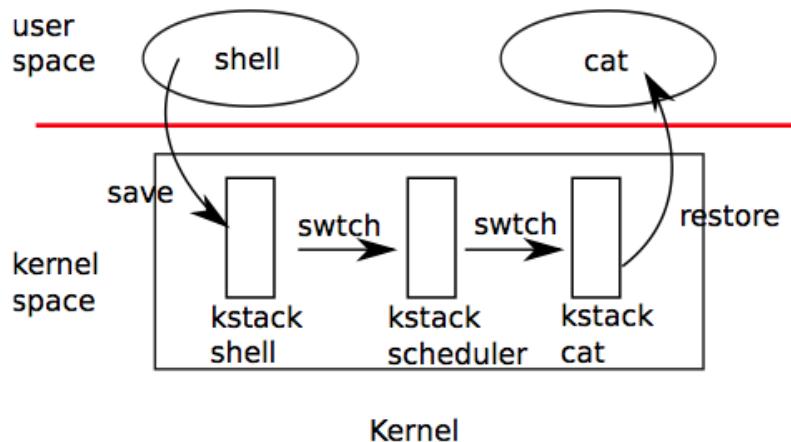


图 5-5 进程切换

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。

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5-7	t0-2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10-11	a0-1	Function arguments/return values	Caller
x12-17	a2-7	Function arguments	Caller
x18-27	s2-11	Saved registers	Callee
x28-31	t3-6	Temporaries	Caller
f0-7	ft0-7	FP temporaries	Caller
f8-9	fs0-1	FP saved registers	Callee
f10-11	fa0-1	FP arguments/return values	Caller
f12-17	fa2-7	FP arguments	Caller
f18-27	fs2-11	FP saved registers	Callee
f28-31	ft8-11	FP temporaries	Caller

表 5-1 RISC-V calling convention register usage.

## (2) NPUcore 实现进程切换的方法

进程的切换依靠 os/src/task/mod.rs 中的 suspend\_current\_and\_run\_next 函数实现。该函数的应用主要有如下两个场景：

- 应用程序手动使用 sys\_yield 系统调用让出当前进程的占用权。

代码片段 5.18 syscall yield

```

1  pub fn sys_yield() -> isize {
2      suspend_current_and_run_next();
3      SUCCESS
4  }

```

- 内核触发定时中断

代码片段 5.19 Timer Interrupt

```

1  Trap::Interrupt(Interrupt::SupervisorTimer) => {
2      do_wake_expired();
3      set_next_trigger();
4      suspend_current_and_run_next();
5  }

```

NPUcore 中， suspend\_current\_and\_run\_next 函数实现如下：

代码片段 5.20 suspend current and run next

```

1  pub fn suspend_current_and_run_next() {
2      // There must be an application running.
3      let task = take_current_task().unwrap();
4
5      // ---- hold current PCB lock
6      let mut task_inner = task.acquire_inner_lock();
7      let task_cx_ptr = &mut task_inner.task_cx as *mut TaskContext;
8      // Change status to Ready
9      task_inner.task_status = TaskStatus::Ready;
10     drop(task_inner);
11     // ---- release current PCB lock
12
13     // push back to ready queue.
14     add_task(task);
15     // jump to scheduling cycle
16     schedule(task_cx_ptr);
17 }

```

函数首先获取了一个当前正在执行进程的 PCB ，命名为 task 。然后从刚才得到的 PCB 里面的可变部分（类型为使用 MutexGuard 锁保护住的一个泛型结构体 MutexGuard<TaskControlBlockInner> ），获取当前进程的 task\_cx (类型是一个结构体 TaskContext ，保存着当前进程的上下文信息)，然后把他转化成一个可变的裸指针。再当前进程的状态从“正在运行”改为“就绪”，也就是停止当前进程。接着手动调用 drop 让 task\_inner 的引用计数值减一，并调用了 add\_task 方法将进程加入就绪队列。最后调用 schedule 函数完成进程的切换。

TaskContext 的定义如下。ra 为调用后需要返回位置的 pc 值，它记录了 \_\_switch 函数返回之后应该跳转到哪里继续执行，从而在任务切换完成并 ret 之后能到正确的位置；sp 为当前程序用户栈的栈指针；s0 s11 是需要保存的 12 个寄存器。

代码片段 5.21 TaskContext

```

1 pub struct TaskContext {
2     ra: usize,
3     sp: usize,
4     s: [usize; 12],
5 }
```

add\_task 方法是将当前的线程加入到懒分配的全局变量 TASK\_MANAGER 的就绪队列中。

代码片段 5.22 add task

```

1 lazy_static! {
2     pub static ref TASK_MANAGER: Mutex<TaskManager> = Mutex::new(
3         TaskManager::new());
4 }
5
6 pub fn add_task(task: Arc<TaskControlBlock>) {
7     TASK_MANAGER.lock().add(task);
8 }
9 impl TaskManager {
10     pub fn add(&mut self, task: Arc<TaskControlBlock>) {
11         self.ready_queue.push_back(task);
12     }
13 }
```

schedule 函数负责将当前的进程切换到处理器调度进程 (idle task)。处理器调度进程由懒分配的全局变量 PROCESSOR 管理，切换过程通过汇编代码 \_\_switch 实现。

代码片段 5.23 TaskContext

```

1 pub struct Processor {
2     current: Option<Arc<TaskControlBlock>>,
3     idle_task_cx: TaskContext,
4 }
5 lazy_static! {
6     pub static ref PROCESSOR: Mutex<Processor> = Mutex::new(Processor::
7         new());
8 }
9 pub fn schedule(switted_task_cx_ptr: *mut TaskContext) {
10     let idle_task_cx_ptr = PROCESSOR.lock().get_idle_task_cx_ptr();
11     unsafe {
12         __switch(switted_task_cx_ptr, idle_task_cx_ptr);
13     }
14 }
```

\_\_switch 函数接受两个参数，旧进程的内核栈指针和新进程的内核栈指针。它将旧进程的上下文 (sp,ra,s0-s11) 保存到内存中，然后从内存中取出新进程的上下文到 CPU 的寄存器中。

代码片段 5.24 \_\_switch

```

1 .altmacro
2 .macro SAVE_SN n
3 sd s\n, (\n+2)*8(a0)
4 .endm
5 .macro LOAD_SN n
6 ld s\n, (\n+2)*8(a1)
7 .endm
8 .section .text
9 .globl __switch
10 __switch:
11 # __switch(
12 #     current_task_cx_ptr: *mut TaskContext,
13 #     next_task_cx_ptr: *const TaskContext
14 # )
15 # save kernel stack of current task
16 sd sp, 8(a0)
17 # save ra & s0~s11 of current execution
18 sd ra, 0(a0)
19 .set n, 0
20 .rept 12
21 SAVE_SN %n
22 .set n, n + 1
23 .endr
24 # restore ra & s0~s11 of next execution
25 ld ra, 0(a1)
26 .set n, 0
27 .rept 12
28 LOAD_SN %n
29 .set n, n + 1
30 .endr
31 # restore kernel stack of next task
32 ld sp, 8(a1)
33 ret

```

## 5.2.5 进程调度

导言：计算机在运行的过程中，经常会出现这样一种情况，那就是内存中的可执行程序（进程）个数大于处理器的个数，为了使得这些程序都能完成自己的任务，处理器可以共享给这些程序使用。而操作系统在这其中起到的作用就是让这些进程高效合理的利用处理器资源，而这个过程我们称之为进程调度（scheduling）。进程调度是进程管理的重要组成部分。

我们的生活中处处存在调度，比如说食堂打饭，一个窗口的阿姨如何满足众多的食客，又比如一家工厂里的一台机床如何处理众多等待加工的零件。调度就是在一定的约束条件下，将有限的资源合理的分配给若干个任务，使得这些任务满足一些指标。回到我们要讲的进程调度上，也就是在有限的cpu资源下，操作系统通过某种调度策略，使得各进程在时间上能够得到处理器完成自己的任务，并且满足一定的性能指标。

然而，NPUCore 是如何完成这个看似简单的操作呢？接下来我们将阐述 npucore 的进程调度这个核心的问题。

### 5.2.4.1 调度策略

我们先大体介绍一下几种常用的调度策略，并详细说明 npucore 采用的调度策略。

(1) 先来先服务：先来先服务 (first-come first-served, FIFO, 先进先出) 调度策略的基本思想是按照进程请求处理器的先后顺序使用处理器。操作系统会创建两个队列，一个称之为就绪队列，一个称之为阻塞队列。

(2) 最短作业优先 (SJF)：在作业调度中，该算法每次从后备作业队列中挑选估计服务时间最短的一个或几个作业，将他们调入内存，分配必要的资源，创建进程并放入就绪队列。

(3) 基于时间片的轮转：如果操作系统给每个运行的进程的运行时间是一个足够小的时间片 (time slice)，时间片一到，就抢占当前进程并切换到另外一个进程进行执行。这样进程以时间片为单位轮流占用处理器执行。对于交互式进程而言，就有比较大的机会在较短时间内执行，从而有助于减少响应时间。这种调度策略称为时间片轮转 (Round-Robin, RR) 调度，基本思路即从就绪队列头取出一个进程，让他运行一个时间片，然后把它放回队列尾，再从队列头取下一个进程执行，周而复始。

(4) 多级反馈队列调度：操作系统根据进程过去一段的执行特征来预测进程未来一段时间里的执行情况，并以此假设为依据来动态的设置进程的优先级，调度选择优先级最高的进程执行。在该调度中，为进程添加了一个名为优先级的属性，并且优先级是可以根据过去的行为反馈来动态的调整。这其中可以细分为“固定优先级的多级无反馈队列”，“可降低优先级的多级反馈队列”，“可提升/降低优先级的多级反馈队列”

除了以上的几种调度策略外，根据计算机系统的不同，还有许多种的进程调度策略，读者可以自行在网上查询资料了解。接下来，我们将详细介绍 npucore 所使用的进程调度策略。

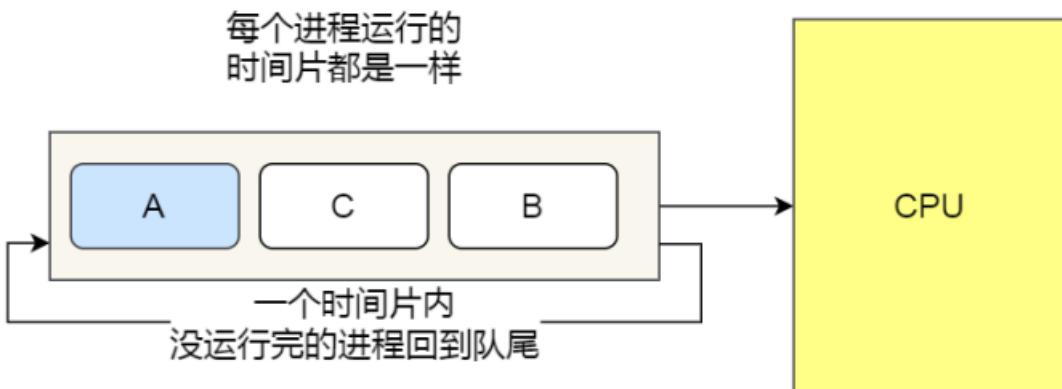
### 5.2.4.2 npucore 的调度策略

(1) 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

图 5-6 时间片轮转



模式。

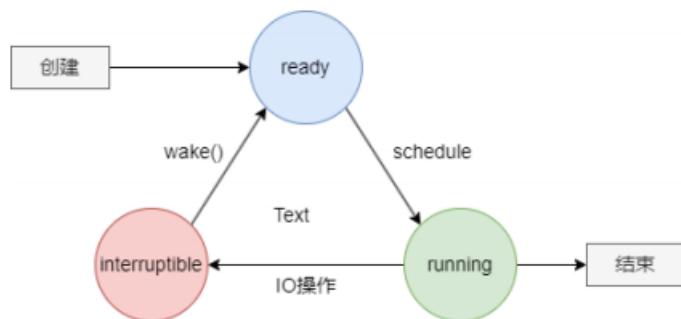
- 阻塞操作指的是在执行设备操作时，如果得不到资源，那么进程就会挂起一直到满足可以操作的条件后再进行操作，被挂起的进程会进入睡眠状态，进入阻塞队列中，直到被唤醒。

• 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。

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。

最后还需要讲的是，npucore 中进程的状态，其分为就绪，正在执行，阻塞（interruptible）状态：

图 5-7 进程的状态



#### 5.2.4.3 进程调度相关代码

前文中我们简要的介绍了何为进程调度以及常见的几种进程调度，还有 npucore 使用了什么调度策略，接下来我们将根据具体的代码来讲解进程调度。

1. 进程调度的容器：任务管理器任务管理器 TaskManager 是进程调度的容器，其包

含一个就绪队列和一个可中断的睡眠状态队列。我们知道，进程由数据结构 TCB 来描述，而目前正在运行的 TCB 处于 processor（处理器）之中管理，而一台计算机不只有一个进程在工作，那些没有在 processor 上运行的进程 TCB 则处于任务管理器中的两个队列之中，一部分是可以准备运行的就绪 TCB，一部分是处于休眠状态的 TCB，该休眠状态可以被中断程序唤醒，因此是可中断的队列。

根据有没有 oom\_handler 特性对于 TaskManger 有两种数据结构，当有时候，会包含 ActiveTracker 这个数据结构，这个的作用主要是维护一个位图来跟踪进程的活动状态，可以检查和设置活动状态

代码片段 5.25 数据结构 ActiveTracker

```

1 pub struct ActiveTracker {
2     bitmap: Vec<u64>,
3 }
4
5 #[cfg(feature = "oom_handler")]
6 #[allow(unused)]
7 impl ActiveTracker {
8     pub const DEFAULT_SIZE: usize = SYSTEM_TASK_LIMIT;
9     pub fn new() -> Self {
10         let len = (Self::DEFAULT_SIZE + 63) / 64;
11         let mut bitmap = Vec::with_capacity(len);
12         bitmap.resize(len, 0);
13         Self { bitmap }
14     }
15     pub fn check_active(&self, pid: usize) -> bool {
16         (self.bitmap[pid / 64] & (1 << (pid % 64))) != 0
17     }
18     pub fn check_inactive(&self, pid: usize) -> bool {
19         (self.bitmap[pid / 64] & (1 << (pid % 64))) == 0
20     }
21     pub fn mark_active(&mut self, pid: usize) {
22         self.bitmap[pid / 64] |= 1 << (pid % 64)
23     }
24     pub fn mark_inactive(&mut self, pid: usize) {
25         self.bitmap[pid / 64] &= !(1 << (pid % 64))
26     }
27 }
```

当没有 oom\_handler 特性时 TaskManger 的数据结构如下：

代码片段 5.26 数据结构 TaskManager

```

1 pub struct TaskManager {
2     pub ready_queue: VecDeque<Arc<TaskControlBlock>>,
3     pub interruptible_queue: VecDeque<Arc<TaskControlBlock>>,
4 }
```

包含了就绪队列和一个可中断的等待队列构成，都是存储进程控制块的队列，其中进程控制块使用原子引用计数智能指针进行引用计数。这有助于有效地管理进程的生命周期和共享。其中就绪队列 ready\_queue 中存储已准备好运行的进程。

进程在这个队列中等待被调度器选中并执行。当一个进程处于等待 CPU 资源的状

态，但是已经准备好执行时，它会被添加到这个队列。

可中断等待队列 interruptible\_queue 专门用于存放处于可中断的睡眠状态的进程，这些进程可以被中断唤醒以继续执行。

关于 TaskManager 的具体实现如下：

代码片段 5.27 TaskManager 下的方法

```

1  impl TaskManager {
2      #[cfg(feature = "oom_handler")]
3      pub fn new() -> Self {
4          Self {
5              ready_queue: VecDeque::new(),
6              interruptible_queue: VecDeque::new(),
7              active_tracker: ActiveTracker::new(),
8          }
9      }
10     #[cfg(not(feature = "oom_handler"))]
11     pub fn new() -> Self {
12         Self {
13             ready_queue: VecDeque::new(),
14             interruptible_queue: VecDeque::new(),
15         }
16     }
17     pub fn add(&mut self, task: Arc<TaskControlBlock>) {
18         self.ready_queue.push_back(task);
19     }
20     #[cfg(feature = "oom_handler")]
21     pub fn fetch(&mut self) -> Option<Arc<TaskControlBlock>> {
22         match self.ready_queue.pop_front() {
23             Some(task) => {
24                 self.active_tracker.mark_active(task.pid.0);
25                 Some(task)
26             }
27             None => None,
28         }
29     }
30     #[cfg(not(feature = "oom_handler"))]
31     pub fn fetch(&mut self) -> Option<Arc<TaskControlBlock>> {
32         self.ready_queue.pop_front()
33     }
34     pub fn add_interruptible(&mut self, task: Arc<TaskControlBlock>) {
35         self.interruptible_queue.push_back(task);
36     }
37     pub fn drop_interruptible(&mut self, task: &Arc<TaskControlBlock>) {
38         self.interruptible_queue
39             .retain(|task_in_queue| Arc::as_ptr(task_in_queue) != Arc::
40             as_ptr(task));
41     }
42     pub fn find_by_pid(&self, pid: usize) -> Option<Arc<TaskControlBlock>>
43     {
44         self.ready_queue
45             .iter()
46             .chain(self.interruptible_queue.iter())
47             .find(|task| task.pid.0 == pid)
48             .cloned()
49     }
50     pub fn find_by_tgid(&self, tgid: usize) -> Option<Arc<TaskControlBlock>>
51     {
52         self.ready_queue
53             .iter()
54             .chain(self.interruptible_queue.iter())
55             .find(|task| task.tgid == tgid)
56             .cloned()
57     }
58 }
```

```

49     >> {
50         self.ready_queue
51             .iter()
52             .chain(self.interruptible_queue.iter())
53             .find(|task| task.tgid == tgid)
54             .cloned()
55     }
56     pub fn ready_count(&self) -> u16 {
57         self.ready_queue.len() as u16
58     }
59     pub fn interruptible_count(&self) -> u16 {
60         self.interruptible_queue.len() as u16
61     }
62     pub fn wake_interruptible(&mut self, task: Arc<TaskControlBlock>) {
63         match self.try_wake_interruptible(task) {
64             Ok(_) => {}
65             Err(_) => {
66                 log::trace!("[wake_interruptible] already waken");
67             }
68         }
69     }
70     pub fn try_wake_interruptible(
71         &mut self,
72         task: Arc<TaskControlBlock>,
73     ) -> Result<(), WaitQueueError> {
74         self.drop_interruptible(&task);
75         if self.find_by_pid(task.pid.0).is_none() {
76             self.add(task);
77             Ok(())
78         } else {
79             Err(WaitQueueError::AlreadyWaken)
80         }
81     }
82     #[allow(unused)]
83     // debug use only
84     pub fn show_ready(&self) {
85         self.ready_queue.iter().for_each(|task| {
86             log::error!("[show_ready] pid: {}", task.pid.0);
87         })
88     }
89     #[allow(unused)]
90     // debug use only
91     pub fn show_interruptible(&self) {
92         self.interruptible_queue.iter().for_each(|task| {
93             log::error!("[show_interruptible] pid: {}", task.pid.0);
94         })
95     }

```

表 5-2 TaskManager 下方法的介绍

方法名	目的	参数	操作	返回值
new	初始化一个 TaskManager 实例。	无	无	返回一个包含两个空队列的 TaskManager 实例。
add	将进程添加到就绪队列。	task: Child(代表一个进程)。	无	无
fetch	从就绪队列中获取一个进程。	无	ready_queue 头部弹出一个进程，并返回它，如果有 oom_handler 特性，则将弹出进程标记为活跃状态。	返回 Option<Child> 表示可能获取到的进程。
add_interruptible	将可中断的进程添加到可中断队列。	task: Child(代表一个进程)。	将 task 添加到 interruptible_queue 尾部。	无
drop_interruptible	从可中断队列中移除指定的可中断进程。	task:&Child(代表一个进程)。	保留不等于 task 的进程，从而移除了匹配的进程。	无
find_by_pid	根据进程 ID 在队列中查找进程	pid: usize(进程 ID)。	无	返回 Option<Child>，表示找到的进程。
find_by_tgid	根据线程组 ID 在队列中查找进程。	tgid: usize(线程组 ID)。	无	返回 Option<Child>，表示找到的进程。
ready_count	获取就绪队列中进程的数量	无	无	返回 u16 类型的进程数量
interruptible_count	获取可中断队列中进程的数量	无	无	返回 u16 类型的进程数量
wake_interruptible	将已唤醒的可中断进程移动到就绪队列中	task: Child(代表一个已唤醒的进程)	调用 try_wake_interruptible 方法，忽略可能的错误	无
try_wake_interruptible	尝试将已唤醒的可中断进程移动到就绪队列中	task: Child(代表一个已唤醒的进程)	如果进程不存在于就绪队列中，将其添加到就绪队列中，否则返回错误	返回 Result<(), WaitQueueError>，表示操作成功或已经唤醒
show_ready	仅用于调试，打印就绪队列的进程 PID。这些方法通过入队出队，查找等方法实现了进程的调度。	无	无	无
show_interruptible	仅用于调试，打印可中断队列的进程 PID。这些方法通过入队出队，查找等方法实现了进程的调度	无	无	无

这是最主要的进程调度的数据结构 TaskManger 的主要内容，Npucore 除了 TaskManger 提供的两个队列之外还建立了 Waitqueue,TimeoutWaitQueue 两个队列分别为等待队列和超时等待队列，这两个队列的结构体为：

代码片段 5.28 数据结构 WaitQueue 与 TimeoutWaitQueue

```
1 pub struct WaitQueue {
```

```

2     inner: VecDeque<Weak<TaskControlBlock>>,
3 }
4 pub struct TimeoutWaitQueue {
5     inner: BinaryHeap<TimeoutWaiter>,
6 }

```

都接收了 TCB 的弱引用作为队列元素，在 Npucore 中，TaskManger 包含的方法实现了最主要的进程调度，WaitQueue 和 TimeoutWaitQueue 则是在进程/线程同步方面 Futex 的视线中发挥作用

以下是关于 WaitQueue 的实现：

代码片段 5.29 数据结构 WaitQueue 下的方法

```

1 impl WaitQueue {
2     pub fn new() -> Self {
3         Self {
4             inner: VecDeque::new(),
5         }
6     }
7     pub fn add_task(&mut self, task: Weak<TaskControlBlock>) {
8         self.inner.push_back(task);
9     }
10    pub fn pop_task(&mut self) -> Option<Weak<TaskControlBlock>> {
11        self.inner.pop_front()
12    }
13    pub fn contains(&self, task: &Weak<TaskControlBlock>) -> bool {
14        self.inner
15            .iter()
16            .any(|task_in_queue| Weak::as_ptr(task_in_queue) == Weak::as_ptr(task))
17    }
18    pub fn is_empty(&self) -> bool {
19        self.inner.is_empty()
20    }
21    pub fn wake_all(&mut self) -> usize {
22        self.wake_at_most(usize::MAX)
23    }
24    pub fn wake_at_most(&mut self, limit: usize) -> usize {
25        if limit == 0 {
26            return 0;
27        }
28        let mut manager = TASK_MANAGER.lock();
29        let mut cnt = 0;
30        while let Some(task) = self.inner.pop_front() {
31            match task.upgrade() {
32                Some(task) => {
33                    let mut inner = task.acquire_inner_lock();
34                    match inner.task_status {
35                        super::TaskStatus::Interruptible => {
36                            inner.task_status = super::task::TaskStatus::
37                                Ready
38                        }
39                        _ => continue,
40                    }
41                    drop(inner);
42                    if manager.try_wake_interruptible(task).is_ok() {

```

```

42             cnt += 1;
43         }
44         if cnt == limit {
45             break;
46         }
47     }
48
49     None => continue,
50 }
51 }
52 cnt
53 }
54 }

```

表 5-3 WaitQueue 下方法的介绍

方法名	目的	参数	操作	返回值
new	创建一个新的 WaitQueue 实例	无	初始化内部使用 VecDeque 存储的任务队列	新创建的 WaitQueue 实例
add_task	将一个任务添加到 WaitQueue 中	自身引用, task - 用 Weak 包装的任务	将任务添加到队列的末尾	无
pop_task	从 WaitQueue 中弹出一个任务	无	从队列的前端弹出一个任务	返回一个 Option, 可能是弹出的任务, 如果队列为空则为 None
contains	判断 WaitQueue 中是否包含与给定任务相等的元素	自身引用, task - 要判断的任务	比较任务的指针是否相等	如果队列中包含相等的任务则返回 true, 否则返回 false
is_empty	判断 WaitQueue 是否为空	自身引用	检查内部队列是否为空	如果队列为空则返回 true, 否则返回 false
wake_all	唤醒 WaitQueue 中的所有任务	自身可变引用	将所有任务的状态更改为 Ready	返回实际唤醒的任务数量
wake_at_most	唤醒 WaitQueue 中不超过 limit 数量的任务	自身可变引用, limit - 最大唤醒数量	将不超过 limit 数量的任务状态更改为 Ready	返回实际唤醒的任务数量

以下是关于 TimeoutWaitQueue 的实现:

代码片段 5.30 TimeoutWaitQueue 下的方法

```

1 impl TimeoutWaitQueue {
2     pub fn new() -> Self {
3         Self {
4             inner: BinaryHeap::new(),
5         }
6     }
7     pub fn add_task(&mut self, task: Weak<TaskControlBlock>, timeout: TimeSpec) {
8         self.inner.push(TimeoutWaiter { task, timeout });
9     }
10    pub fn wake_expired(&mut self, now: TimeSpec) {
11        let mut manager = TASK_MANAGER.lock();
12        while let Some(waiter) = self.inner.pop() {
13            // the remaining tasks in heap haven't reach their timeout

```

```

14     if waiter.timeout > now {
15         log::trace!(
16             "[wake_expired] no more expired, next pending task
17             timeout: {:?}, now: {:?}",
18             waiter.timeout,
19             now
20         );
21         self.inner.push(waiter);
22         break;
23     } else {
24         match waiter.task.upgrade() {
25             Some(task) => {
26                 let mut inner = task.acquire_inner_lock();
27                 match inner.task_status {
28                     super::TaskStatus::Interruptible => {
29                         inner.task_status = super::task::TaskStatus
30                         ::Ready
31                         }
32                         -> continue,
33                     drop(inner);
34                     log::trace!(
35                         "[wake_expired] pid: {}, timeout: {:?}",
36                         task.pid.0,
37                         waiter.timeout
38                     );
39                     manager.wake_interruptible(task);
40                 }
41                 None => continue,
42             }
43         }
44     }
45 #[allow(unused)]
46 // debug use only
47 pub fn show_waiter(&self) {
48     for waiter in self.inner.iter() {
49         log::error!("[show_waiter] timeout: {:?}", waiter.timeout);
50     }
51 }
52 }
53 }
```

表 5-4 TimeoutWaitQueue 下的方法的介绍

方法名	目的	参数	操作	返回值
new	创建一个新的 TimeoutWaitQueue 实例	无	初始化内部使用 BinaryHeap 存储的超时等待队列	新创建的 TimeoutWaitQueue 实例
add_task	将一个带有超时时间的任务添加到 TimeoutWaitQueue 中	自身可变引用, task - 用 Weak 包装的任务。timeout - 任务的超时时间。	将任务包装成 TimeoutWaiter 结构，并将其按照超时时间顺序插入二进制堆中	无
wake_expired	唤醒 TimeoutWaitQueue 中已经超时的任务	自身可变引用, now - 当前时间	弹出二进制堆顶部的任务，检查其超时时间是否已经到达。如果任务还未超时，将其重新插入堆中并结束。如果任务已经超时，将任务的状态更改为 Ready，并唤醒对应的任务。	无
show_waiter	用于调试，打印当前 TimeoutWaitQueue 中的等待者（任务）的超时时间	自身引用	打印每个等待者的超时时间	无

## 2. 阻塞式进程调度的实现：

### 多路复用：

多路复用的实现如下：当一个进程等待磁盘请求时，OS 使之进入睡眠状态，然后调度执行另一个进程。另外，当一个进程耗尽了它在处理器上运行的时间片后，OS 使用时钟中断强制它停止运行，这样调度器才能调度运行其他进程。这样的多路复用机制为进程提供了独占处理器的假象，类似于 OS 使用内存分配器和页表硬件为进程提供了独占内存的假象。

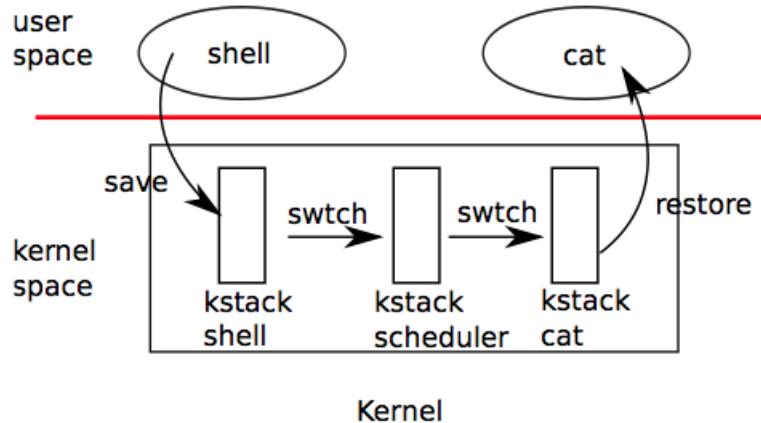
实现多路复用有几个难点。首先，应该如何从运行中的一个进程切换到另一个进程？OS 采用了普通的上下文切换机制；虽然这里的思路是非常简洁明了的，但是其代码实现是操作系统中最晦涩难懂的一部分。第二，如何让上下文切换透明化？OS 只是简单地使用时钟中断处理程序来驱动上下文切换。第三，可能出现多个 CPU 同时切换进程的情况，那么我们必须使用一个带锁的方案来避免竞争。第四，进程退出时必须释放其占用内存与资源，但由于它本身在使用自己的资源（譬如其内核栈），所以不能由该进程本身释放其占有的所有资源。

OS 必须为进程提供互相协作的方法。譬如，父进程需要等待子进程结束，以及读取管道数据的进程需要等待其他进程向管道中写入数据。与其让这些等待中的进程消耗 CPU 资源，不如让它们暂时放弃 CPU，进入睡眠状态来等待其他进程发出事件来唤醒它们。但我们要小心设计以防睡眠进程遗漏事件通知。

### 上下文切换：

如下图所示，OS 在低层次中实现了两种上下文切换：从进程的内核线程切换到当前 CPU 的调度器线程，从调度器线程到进程的内核线程。OS 永远不会直接从用户态进程切换到另一个用户态进程；这种切换是通过用户态-内核态切换（系统调用或中断）、切换到调度器、切换到新进程的内核线程、最后这个陷入返回实现的。我们将以一个简单的例子，详细介绍了这一过程。

图 5-8 上下文切换



设有两个用户态进程 A 和 B，它们在操作系统内核的管理下运行。CPU 当前正在执行进程 A。

首先是用户态到内核态的切换。当进程 A 需要执行一个系统调用（例如，读写文件、申请内存等）或者发生一个中断事件（例如，时钟中断、硬件中断等），CPU 会触发一个异常或中断，将控制权转移到操作系统内核。

**触发异常或中断：**比如，进程 A 执行了一个系统调用指令或硬件设备触发了中断。

**保存用户态上下文：**操作系统内核保存进程 A 的用户态上下文，包括通用寄存器、程序计数器、堆栈指针等。

**切换到内核态：**CPU 转到内核态执行，此时操作系统内核可以访问更多的特权指令和数据结构。

其次是在内核态的处理过程。在内核态，操作系统会执行一些必要的操作，比如：

- **系统调用处理：**根据系统调用的类型，执行相应的内核代码完成用户请求的操作。

- **调度器的工作：**决定下一个要执行的进程。这可能涉及到选择一个就绪队列中的新进程。

**内核线程切换：**如果需要切换到一个新的用户态进程（假设是进程 B），操作系统可能会将控制权切换到该进程的内核线程。

之后为内核态到用户态的切换：

- **恢复用户态上下文：**如果切换到了新的用户态进程 B，操作系统会从进程 B 的

内核线程中恢复用户态上下文。

- 切换到用户态：CPU 会从内核态切换回用户态，开始执行进程 B 的用户态代码。

最后，进程 B 开始执行：

- 加载用户态上下文：进程 B 的用户态上下文被加载到 CPU 寄存器中。
- 继续执行：CPU 开始执行进程 B 的用户态代码，从上次中断或系统调用的位置继续执行。

整个过程中，关键的步骤包括从用户态到内核态的切换，内核态的处理过程，以及从内核态回到用户态的切换。这个过程确保了不同进程之间的无缝切换，使得操作系统能够有效地进行多任务调度。

任务切换：

当一台计算机开始运行时，操作系统首先会通过懒分配创建一个初始的进程，该进程是一个全局的进程，接下来的进程都由它创建。初始进程创建以后，根据计算机的需要或者用户的操作，操作系统开始由 `clone` 函数来创建一个个新的进程并加入到就绪队列中。按照时间片轮转的想法，操作系统给正在处理器上运行的（也就是 `processor` 所拥有的 TCB）进程分配了一个时间片，在自然运行到时间耗尽后，操作系统就会使用 `sys_yield()` 函数让出当前进程对处理器的占有权，又或者内核出现错误造成 trap，程序也会使当前进程让出处理器，其中的核心函数便是 `suspend_current_and_run_next`，该函数将当前进程的上下文进行保存，并将进程的状态改成 `ready`，表示进程准备执行，然后将他放回到 `ready` 队列中的队尾进行排队，等待下一次轮转到它。

`suspend_current_and_run_next` 函数会使得当前进程从 `current_task` 切换到中转进程 `idle`，而调度器如何将 `idle` 切换为 `next_task` 呢？

它将使用 `run_tasks()` 函数完成从 `idle` 到 `next_task` 的切换。`run_tasks()` 函数位于 `os/src/task/processor.rs` 下。让我们看 `os/src/task/processor.rs` 下的 `run_task` 函数：

代码片段 5.31 run\_task 函数

```

1 pub fn run_tasks() {
2     loop {
3         let mut processor = PROCESSOR.lock();
4         if let Some(task) = fetch_task() {
5             let idle_task_cx_ptr = processor.get_idle_task_cx_ptr();
6             // access coming task TCB exclusively
7             let next_task_cx_ptr = {
8                 let mut task_inner = task.acquire_inner_lock();
9                 task_inner.task_status = TaskStatus::Running;
10                &task_inner.task_cx as *const TaskContext
11            };
12            processor.current = Some(task);
13            // release processor manually
14            drop(processor);
15            unsafe {
16                __switch(idle_task_cx_ptr, next_task_cx_ptr);
17            }
18        } else {

```

```

19         drop(processor);
20         // we have no ready tasks, try to wake some...
21         do_wake_expired();
22     }
23 }
24 }
```

可以看到，操作系统采“轮询机制”，在操作系统运行的任一时刻都在尝试从 idle 流切换到下一个进程（采用 loop 死循环），接着我们分析一下具体的过程。

fetch\_task() 的作用是选取一个将要执行的进程

代码片段 5.32 fetch\_task 函数

```

1 pub fn fetch_task() -> Option<Arc<TaskControlBlock>> {
2 TASK_MANAGER.lock().fetch()
3 }
4 impl TaskManager {
5 /**
6 pub fn fetch(&mut self) -> Option<Arc<TaskControlBlock>> {
7 self.ready_queue.pop_front()
8 }
9 /**
10 }
```

可以看到，调度器的选择是将“就绪任务”的队列进行出队操作。因此我们可以知道，NPUcore 中对进程的调度实际上就是对 ready\_queue 进行管理。

等待队列：

进程的调度过程中，有一些进程是需要等待某种资源，例如 IO 输入这样的资源，因此才陷入阻塞过程中，他们不得到自己所需的资源前，便无法被唤醒加入就绪队列中（因此它不同于 interruptible\_queue 中的进程，在可中断睡眠队列中的进程，需要等待中断操作就可以被唤醒），因此这涉及到了 waitqueue 这个重要的数据结构。

等待队列（wait\_queue）是用于管理等待特定资源或时间的进程或线程，它是一种先进先出的数据结构。该队列通常用于解决并发编程中的同步和互斥问题。当多个进程或者线程需要访问共享资源时，如果资源已经被占用，那么需要将正在等待的进程放入等待队列，以便在资源可用时依次获得访问权限。

当某资源得到释放，是的等待队列中的进程可被唤醒时，操作系统会调用 wake\_at\_most 这个方法。

代码片段 5.33 wake\_at\_most 方法

```

1 pub fn wake_at_most(&mut self, limit: usize) -> usize {
2     if limit == 0 {
3         return 0;
4     }
5     let mut manager = TASK_MANAGER.lock();
6     let mut cnt = 0;
7     while let Some(task) = self.inner.pop_front() {
8         match task.upgrade() {
9             Some(task) => {
```

```

10     let mut inner = task.acquire_inner_lock();
11     match inner.task_status {
12         super::TaskStatus::Interruptible => {
13             inner.task_status = super::task::TaskStatus::Ready
14         }
15         _ => continue,
16     }
17     drop(inner);
18     if manager.try_wake_interruptible(task).is_ok() {
19         cnt += 1;
20     }
21     if cnt == limit {
22         break;
23     }
24     None => continue,
25 }
26 }
27 cnt
28 }
29 }
```

该函数的功能为：从等待队列中唤醒最多 limit 个任务，并返回实际唤醒的任务数量。

首先，函数检查 limit 是否为 0，如果为 0，则表示没有唤醒任何任务。

然后，函数获取全局的任务管理器的锁，确保任务管理器的线程安全性。

接下来，函数使用 cnt 记录唤醒的任务的数量，初始为 0.

然后，函数进入循环，从等待队列（self.inner）的头部弹出任务，并进行处理。

在处理吃钱，函数会尝试将任务引用升级为 ARC 类型的对象。如果该对象有效，则执行以下操作：

- 1) 获取任务对象的内部锁（acquire\_inner\_lock）
- 2) 根据任务的状态进行不同处理，interruptible 的任务，将其状态设置为 ready，如果为其他状态，则不需要唤醒，继续处理下一个任务。
- 3) 释放任务对象的内部锁（drop（inner））
- 4) 尝试使用 try\_wake\_interruptible 方法唤醒任务
- 5) 检查 cnt 计数器是否达到 limit 限制，达到则跳出循环。

如果任务对象引用升级为 None，表示任务已经被销毁了，函数会处理下一个任务。

最后返回唤醒的任务数量 cnt。

经过该函数的处理，那些得到了资源的进程，可以被唤醒进入 ready 队列中，等待时间片的轮转。而该函数中又涉及到了一个关键的函数 try\_wake\_interruptible:

代码片段 5.34 try\_wake\_interruptible 方法

```

1 pub fn try_wake_interruptible(
2     &mut self,
3     task: Arc<TaskControlBlock>,
4 ) -> Result<(), WaitQueueError> {
```

```

5     self.drop_interruptible(&task);
6     if self.find_by_pid(task.pid.0).is_none() {
7         self.add(task);
8         Ok(())
9     } else {
10        Err(WaitQueueError::AlreadyWaken)
11    }
12 }
```

该函数接受一个任务对象 task 作为参数，并尝试唤醒该任务，返回一个 result 类型，表示唤醒操作的结果。

首先，函数调用 self.drop\_interruptible 方法，该方法会从等待队列中移除具有相同 pid 的任务。该步骤确保在唤醒任务之前先将其从等待队列中移除，避免重复唤醒。

接下来，函数使用 self.find\_by\_pid 方法来检查 ready 队列中是否具有相同 pid 的任务，如果返回 None，表示不存在。

在这种情况下，函数调用 self.add 方法将任务添加到准备队列中，并返回 ok 结果。

如果 ready 队列中具有相同 pid 的任务，那么函数表示唤醒失败，已经唤醒。

除了等待队列以外，还有一个也很重要的数据结构被用在进程的调度中，那就是 TimeoutWaitQueue（超时等待池）。

超时等待池用于处理等待超时的情况。在某些场景下，等待某个资源或者事件的进程或线程可能需要在一定的时间内得到相应，如果超过指定时间仍然没有得到响应，那么就需要采取相应的措施。超时等待池允许进程或者线程设置一个超时时间，如果超过时间却未能唤醒，那么就会触发超时逻辑。

超时等待池的存在是为了处理等待时间的限制，以避免进程或者线程在等待资源或时间上长时间阻塞。它提供了一种在等待超时后继续执行的机制，从而增加了系统的可靠性和响应性。

npucore 依靠定时器在一定的时间间隔下进入 trap，自动唤醒超时等待池中的进程。

在 os/src/trap/mod.rs 源文件中的 trap\_handle 函数中有对 SupervisorTimer 中断的处理方式：

代码片段 5.35 trap\_handler 中对 SupervisorTimer 中断的处理

```

1 Trap::Interrupt(Interrupt::SupervisorTimer) => {
2     do_wake_expired();
3     set_next_trigger();
4     suspend_current_and_run_next();
5 }
```

该函数首先调用了 do\_wake\_expired 方法：

代码片段 5.36 do\_wake\_expired 方法

```

1 pub fn do_wake_expired() {
2     TIMEOUT_WAITQUEUE
```

```

3     .lock()
4     .wake_expired(crate::timer::TimeSpec::now());
5 }

```

其中又调用了 TimeoutWaitQueue 的 wake\_expired 方法:

代码片段 5.37 wake\_expired 方法

```

1  pub fn wake_expired(&mut self, now: TimeSpec) {
2      let mut manager = TASK_MANAGER.lock();
3      while let Some(waiter) = self.inner.pop() {
4          // the remaining tasks in heap haven't reach their timeout
5          if waiter.timeout > now {
6              log::trace!(
7                  "[wake_expired] no more expired, next pending task
8                  timeout: {:?}, now: {:?}",
9                  waiter.timeout,
10                 now
11             );
12             self.inner.push(waiter);
13             break;
14         } else {
15             match waiter.task.upgrade() {
16                 Some(task) => {
17                     let mut inner = task.acquire_inner_lock();
18                     match inner.task_status {
19                         super::TaskStatus::Interruptible => {
20                             inner.task_status = super::task::TaskStatus
21                             ::Ready
22                             }
23                             -=> continue,
24                         }
25                     drop(inner);
26                     log::trace!(
27                         "[wake_expired] pid: {}, timeout: {:?}",
28                         task.pid.0,
29                         waiter.timeout
30                     );
31                     manager.wake_interruptible(task);
32                 }
33             None => continue,
34         }
35     }
36 }

```

该方法获取一个对 self 的可变引用，以及一个类型为 timespec 的 now 参数，没有指定返回类型。该方法的作用是从任务列表中唤醒已经过期的任务，并将其状态改为 ready。

首先代码通过获取 task\_manager 对象上的锁来创建一个 manager 变量，并获取一个互斥锁的保护。

接着，代码进入循环，不断的从 self.inner 弹出一个元素来遍历任务。

在循环的每一次迭代中，检查当前任务（waiter）是否已经过期，如果没有过期，代码

会将任务放回 self.inner，并中断循环。

如果任务已经过期，代码会尝试将任务唤醒。它通过调用 waiter.task.upgrade() 获取任务的强引用。如果任务仍然存在，那么获取内部锁，检查任务状态。

如果状态为 interruptible，则改为 ready，表示就绪。

最后，代码释放内部锁，调用 wake\_interruptible(task) 来唤醒任务。

如果任务的强引用已经无效，则说明任务已经被销毁，代码忽略此任务，继续处理下一个任务。

由此，我们就讲完了 npucore 中进程调度相关的内容，本节我们只对重要的地方做了比较详细的介绍，实际上进程的调度是个复杂的问题，光靠这些短短的文字是讲不清楚的，读者可以在闲暇之余，查询网上资料加深自己对进程调度的了解。

## 5.2.6 进程退出

当一个进程完成自己的工作后，就需要调用 exit 进行“退出”以结束自己的生命周期。在 xv6 中，当一个子进程退出时它并不是直接死掉，而是将状态转变为 Zombie。此后，当父进程调用 wait 时，将发现子进程可以退出，并由父进程负责释放子进程相关的内存空间。倘若父进程在子进程之前退出了，则由初始进程 initproc 接收子进程并负责它们退出。下面我们结合代码来阐述这一过程：

首先来看 sys\_exit 系统调用：

代码片段 5.38 os/src/syscall/process.rs

```

1 pub fn sys_exit(exit_code: i32) -> ! {
2     exit_current_and_run_next(exit_code);
3     panic!("Unreachable in sys_exit!");
4 }
```

事实上，当应用调用 sys\_exit 系统调用主动退出，或是执行出错由内核终止之后，内核中都将调用 exit\_current\_and\_run\_next 函数退出当前进程并切换到下一个进程。

exit\_current\_and\_run\_next 函数以一个退出码作为参数。当在 sys\_exit 系统调用中正常退出时，退出码由应用传到内核中；而对于出错退出的情况（例如访存错误、非法指令异常等），则是由内核指定一个特定的退出码（具体可在 trap\_handler 函数中查看）。最终，这个退出码会写入当前进程的 TCB 中，具体如下：

代码片段 5.39 os/src/task/mod.rs

```

1 pub fn exit_current_and_run_next(exit_code: i32) {
2     let task = take_current_task().unwrap();
3     let mut inner = task.inner_exclusive_access();
4     inner.task_status = TaskStatus::Zombie;
5     inner.exit_code = exit_code;
6     // ++++++ access initproc TCB exclusively
7     {
8         let mut initproc_inner = INITPROC.inner_exclusive_access();
```

```

9   for child in inner.children.iter() {
10     child.inner_exclusive_access().parent = Some(Arc::downgrade(&
11       INITPROC));
12     initproc_inner.children.push(child.clone());
13   }
14   // ++++++ stop exclusively accessing parent PCB
15   inner.children.clear();
16   inner.memory_set.recycle_data_pages();
17   drop(inner);
18   // **** stop exclusively accessing current PCB
19   drop(task);
20   let mut _unused = TaskContext::zero_init();
21   schedule(&mut _unused as *mut_);
22 }

```

第 2 行，我们从处理器监控 PROCESSOR 中取出而不是获取一个拷贝，这是为了正确维护 TCB 的引用计数。

第 4 行，我们将 TCB 中的状态改为为 TaskStatus::Zombie 即僵尸进程，令其后续被父进程在 waitpid 系统调用时进行回收。

第 5 行，我们将传入的退出码 exit\_code 写入 TCB 中，使后续父进程可以收集该退出码。

第 7~13 行，我们将当前进程的所有子进程挂在初始进程 initproc 下面。做法是遍历当前进程的每个子进程，修改其父进程为初始进程，并把它们加入初始进程的子进程向量中。最后在第 15 行，我们将当前进程的子进程向量清空。

第 16 行，我们对当前进程占用的资源进行早期回收。我们调用了 recycle\_data\_pages 函数：

代码片段 5.40 os/src/mm/memory\_set.rs

```

1 impl MemorySet {
2   pub fn recycle_data_pages(&mut self) {
3     self.areas.clear();
4   }
5 }

```

注意该函数只是将地址空间中的逻辑段列表 areas 清空，从而使应用的地址空间被回收（即进程的数据段和代码段对应的物理页帧被回收）。然而用来存放页表的那些物理页帧此时还不会被回收，需要由父进程最后来回收子进程剩余的占用资源。

第 21 行，我们最后调用了 schedule 触发调度与任务切换，由于我们再也不会回到该进程的执行过程中，因此无需进行任务上下文的保存。

以上是子进程进行退出的过程，下面介绍父进程通过 sys\_wait4 系统调用来自回收子进程资源的实现：

代码片段 5.41 os/src/syscall/process.rs

```

1 pub fn sys_wait4(pid: isize, status: *mut u32, option: u32, ru: *mut Rusage
) -> isize {

```

```

2 let option = WaitOption::from_bits(option).unwrap();
3 info!("[sys_wait4] pid: {}, option: {:?}", pid, option);
4 let task = current_task().unwrap();
5 let token = task.get_user_token();
6 loop {
7     // find a child process
8
9     // ---- hold current PCB lock
10    let mut inner = task.acquire_inner_lock();
11    if inner
12        .children
13        .iter()
14        .find(|p| pid == -1 || pid as usize == p.getpid())
15        .is_none()
16    {
17        return ECHILD;
18        // ---- release current PCB lock
19    }
20    inner
21        .children
22        .iter()
23        .filter(|p| pid == -1 || pid as usize == p.getpid())
24        .for_each(|p| {
25            trace!(
26                "[sys_wait4] found child pid: {}, status: {:?}", p.pid.0,
27                p.acquire_inner_lock().task_status
28            )
29        });
30    let pair = inner.children.iter().enumerate().find(|(_, p)| {
31        // ++++ temporarily hold child PCB lock
32        p.acquire_inner_lock().is_zombie() && (pid == -1 || pid as
33            usize == p.getpid())
34        // ++++ release child PCB lock
35    });
36    if let Some((idx, _)) = pair {
37        // drop last TCB of child
38        let child = inner.children.remove(idx);
39        trace!("[wait4] release zombie task, pid: {}", child.pid.0);
40        // confirm that child will be deallocated after being removed
41        // from children list
42        assert_eq!(Arc::strong_count(&child), 1);
43        // if main thread exit
44        if child.pid.0 == child.tgid {
45            let found_pid = child.getpid();
46            // ++++ temporarily hold child lock
47            let exit_code = child.acquire_inner_lock().exit_code;
48            // ++++ release child PCB lock
49            if !status.is_null() {
50                // this may NULL!!!
51                match translated_refmut(token, status) {
52                    Ok(word) => *word = exit_code,
53                    Err(errno) => return errno,
54                };
55            }
56            return found_pid as isize;
57        }
58    } else {
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
589
590
591
592
593
594
595
596
597
598
599
599
600
601
602
603
604
605
606
607
608
609
609
610
611
612
613
614
615
616
617
618
619
619
620
621
622
623
624
625
626
627
627
628
629
629
630
631
632
633
634
635
636
637
637
638
639
639
640
641
642
643
644
645
645
646
647
647
648
649
649
650
651
652
653
653
654
655
655
656
656
657
657
658
658
659
659
660
660
661
661
662
662
663
663
664
664
665
665
666
666
667
667
668
668
669
669
670
670
671
671
672
672
673
673
674
674
675
675
676
676
677
677
678
678
679
679
680
680
681
681
682
682
683
683
684
684
685
685
686
686
687
687
688
688
689
689
690
690
691
691
692
692
693
693
694
694
695
695
696
696
697
697
698
698
699
699
700
700
701
701
702
702
703
703
704
704
705
705
706
706
707
707
708
708
709
709
710
710
711
711
712
712
713
713
714
714
715
715
716
716
717
717
718
718
719
719
720
720
721
721
722
722
723
723
724
724
725
725
726
726
727
727
728
728
729
729
730
730
731
731
732
732
733
733
734
734
735
735
736
736
737
737
738
738
739
739
740
740
741
741
742
742
743
743
744
744
745
745
746
746
747
747
748
748
749
749
750
750
751
751
752
752
753
753
754
754
755
755
756
756
757
757
758
758
759
759
760
760
761
761
762
762
763
763
764
764
765
765
766
766
767
767
768
768
769
769
770
770
771
771
772
772
773
773
774
774
775
775
776
776
777
777
778
778
779
779
780
780
781
781
782
782
783
783
784
784
785
785
786
786
787
787
788
788
789
789
790
790
791
791
792
792
793
793
794
794
795
795
796
796
797
797
798
798
799
799
800
800
801
801
802
802
803
803
804
804
805
805
806
806
807
807
808
808
809
809
810
810
811
811
812
812
813
813
814
814
815
815
816
816
817
817
818
818
819
819
820
820
821
821
822
822
823
823
824
824
825
825
826
826
827
827
828
828
829
829
830
830
831
831
832
832
833
833
834
834
835
835
836
836
837
837
838
838
839
839
840
840
841
841
842
842
843
843
844
844
845
845
846
846
847
847
848
848
849
849
850
850
851
851
852
852
853
853
854
854
855
855
856
856
857
857
858
858
859
859
860
860
861
861
862
862
863
863
864
864
865
865
866
866
867
867
868
868
869
869
870
870
871
871
872
872
873
873
874
874
875
875
876
876
877
877
878
878
879
879
880
880
881
881
882
882
883
883
884
884
885
885
886
886
887
887
888
888
889
889
890
890
891
891
892
892
893
893
894
894
895
895
896
896
897
897
898
898
899
899
900
900
901
901
902
902
903
903
904
904
905
905
906
906
907
907
908
908
909
909
910
910
911
911
912
912
913
913
914
914
915
915
916
916
917
917
918
918
919
919
920
920
921
921
922
922
923
923
924
924
925
925
926
926
927
927
928
928
929
929
930
930
931
931
932
932
933
933
934
934
935
935
936
936
937
937
938
938
939
939
940
940
941
941
942
942
943
943
944
944
945
945
946
946
947
947
948
948
949
949
950
950
951
951
952
952
953
953
954
954
955
955
956
956
957
957
958
958
959
959
960
960
961
961
962
962
963
963
964
964
965
965
966
966
967
967
968
968
969
969
970
970
971
971
972
972
973
973
974
974
975
975
976
976
977
977
978
978
979
979
980
980
981
981
982
982
983
983
984
984
985
985
986
986
987
987
988
988
989
989
990
990
991
991
992
992
993
993
994
994
995
995
996
996
997
997
998
998
999
999
1000
1000
1001
1001
1002
1002
1003
1003
1004
1004
1005
1005
1006
1006
1007
1007
1008
1008
1009
1009
1010
1010
1011
1011
1012
1012
1013
1013
1014
1014
1015
1015
1016
1016
1017
1017
1018
1018
1019
1019
1020
1020
1021
1021
1022
1022
1023
1023
1024
1024
1025
1025
1026
1026
1027
1027
1028
1028
1029
1029
1030
1030
1031
1031
1032
1032
1033
1033
1034
1034
1035
1035
1036
1036
1037
1037
1038
1038
1039
1039
1040
1040
1041
1041
1042
1042
1043
1043
1044
1044
1045
1045
1046
1046
1047
1047
1048
1048
1049
1049
1050
1050
1051
1051
1052
1052
1053
1053
1054
1054
1055
1055
1056
1056
1057
1057
1058
1058
1059
1059
1060
1060
1061
1061
1062
1062
1063
1063
1064
1064
1065
1065
1066
1066
1067
1067
1068
1068
1069
1069
1070
1070
1071
1071
1072
1072
1073
1073
1074
1074
1075
1075
1076
1076
1077
1077
1078
1078
1079
1079
1080
1080
1081
1081
1082
1082
1083
1083
1084
1084
1085
1085
1086
1086
1087
1087
1088
1088
1089
1089
1090
1090
1091
1091
1092
1092
1093
1093
1094
1094
1095
1095
1096
1096
1097
1097
1098
1098
1099
1099
1100
1100
1101
1101
1102
1102
1103
1103
1104
1104
1105
1105
1106
1106
1107
1107
1108
1108
1109
1109
1110
1110
1111
1111
1112
1112
1113
1113
1114
1114
1115
1115
1116
1116
1117
1117
1118
1118
1119
1119
1120
1120
1121
1121
1122
1122
1123
1123
1124
1124
1125
1125
1126
1126
1127
1127
1128
1128
1129
1129
1130
1130
1131
1131
1132
1132
1133
1133
1134
1134
1135
1135
1136
1136
1137
1137
1138
1138
1139
1139
1140
1140
1141
1141
1142
1142
1143
1143
1144
1144
1145
1145
1146
1146
1147
1147
1148
1148
1149
1149
1150
1150
1151
1151
1152
1152
1153
1153
1154
1154
1155
1155
1156
1156
1157
1157
1158
1158
1159
1159
1160
1160
1161
1161
1162
1162
1163
1163
1164
1164
1165
1165
1166
1166
1167
1167
1168
1168
1169
1169
1170
1170
1171
1171
1172
1172
1173
1173
1174
1174
1175
1175
1176
1176
1177
1177
1178
1178
1179
1179
1180
1180
1181
1181
1182
1182
1183
1183
1184
1184
1185
1185
1186
1186
1187
1187
1188
1188
1189
1189
1190
1190
1191
1191
1192
1192
1193
1193
1194
1194
1195
1195
1196
1196
1197
1197
1198
1198
1199
1199
1200
1200
1201
1201
1202
1202
1203
1203
1204
1204
1205
1205
1206
1206
1207
1207
1208
1208
1209
1209
1210
1210
1211
1211
1212
1212
1213
1213
1214
1214
1215
1215
1216
1216
1217
1217
1218
1218
1219
1219
1220
1220
1221
1221
1222
1222
1223
1223
1224
1224
1225
1225
1226
1226
1227
1227
1228
1228
1229
1229
1230
1230
1231
1231
1232
1232
1233
1233
1234
1234
1235
1235
1236
1236
1237
1237
1238
1238
1239
1239
1240
1240
1241
1241
1242
1242
1243
1243
1244
1244
1245
1245
1246
1246
1247
1247
1248
1248
1249
1249
1250
1250
1251
1251
1252
1252
1253
1253
1254
1254
1255
1255
1256
1256
1257
1257
1258
1258
1259
1259
1260
1260
1261
1261
1262
1262
1263
1263
1264
1264
1265
1265
1266
1266
1267
1267
1268
1268
1269
1269
1270
1270
1271
1271
1272
1272
1273
1273
1274
1274
1275
1275
1276
1276
1277
1277
1278
1278
1279
1279
1280
1280
1281
1281
1282
1282
1283
1283
1284
1284
1285
1285
1286
1286
1287
1287
1288
1288
1289
1289
1290
1290
1291
1291
1292
1292
1293
1293
1294
1294
1295
1295
1296
1296
1297
1297
1298
1298
1299
1299
1300
1300
1301
1301
1302
1302
1303
1303
1304
1304
1305
1305
1306
1306
1307
1307
1308
1308
1309
1309
1310
1310
1311
1311
1312
1312
1313
1313
1314
1314
1315
1315
1316
1316
1317
1317
1318
1318
1319
1319
1320
1320
1321
1321
1322
1322
1323
1323
1324
1324
1325
1325
1326
1326
1327
1327
1328
1328
1329
1329
1330
1330
1331
1331
1332
1332
1333
1333
1334
1334
1335
1335
1336
1336
1337
1337
1338
1338
1339
1339
1340
1340
1341
1341
1342
1342
1343
1343
1344
1344
1345
1345
1346
1346
1347
1347
1348
1348
1349
1349
1350
1350
1351
1351
1352
1352
1353
1353
1354
1354
1355
1355
1356
1356
1357
1357
1358
1358
1359
1359
1360
1360
1361
1361
1362
1362
1363
1363
1364
1364
1365
1365
1366
1366
1367
1367
1368
1368
1369
1369
1370
1370
1371
1371
1372
1372
1373
1373
1374
1374
1375
1375
1376
1376
1377
1377
1378
1378
1379
1379
1380
1380
1381
1381
1382
1382
1383
1383
1384
1384
1385
1385
1386
1386
1387
1387
1388
1388
1389
1389
1390
1390
1391
1391
1392
1392
1393
1393
1394
1394
1395
1395
1396
1396
1397
1397
1398
1398
1399
1399
1400
1400
1401
1401
1402
1402
1403
1403
1404
1404
1405
1405
1406
1406
1407
1407
1408
1408
1409
1409
1410
1410
1411
1411
1412
1412
1413
1413
1414
1414
1415
1415
1416
1416
1417
1417
1418
1418
1419
1419
1420
1420
1421
1421
1422
1422
1423
1423
1424
1424
1425
1425
1426
1426
1427
1427
1428
1428
1429
1429
1430
1430
1431
1431
1432
1432
1433
1433
1434
1434
1435
1435
1436
1436
1437
1437
1438
1438
1439
1439
1440
1440
1441
1441
1442
1442
1443
1443
1444
1444
1445
1445
1446
1446
1447
1447
1448
1448
1449
1449
1450
1450
1451
1451
1452
1452
1453
1453
1454
1454
1455
1455
1456
1456
1457
1457
1458
1458
1459
1459
1460
1460
1461
1461
1462
1462
1463
1463
1464
1464
1465
1465
1466
1466
1467
1467
1468
1468
1469
1469
1470
1470
1471
1471
1472
1472
1473
1473
1474
1474
1475
1475
1476
1476
1477
1477
1478
1478
1479
1479
1480
1480
1481
1481
1482
1482
1483
1483
1484
1484
1485
1485
1486
1486
1487
1487
1488
1488
1489
1489
1490
1490
1491
1491
1492
1492
1493
1493
1494
1494
1495
1495
1496
1496
1497
1497
1498
1498
1499
1499
1500
1500
1501
1501
1502
1502
1503
1503
1504
1504
1505
1505
1506
1506
1507
1507
1508
1508
1509
1509
1510
1510
1511
1511
1512
1512
1513
1513
1514
1514
1515
1515
1516
1516
1517
1517
1518
1518
1519
1519
1520
1520
1521
1521
1522
1522
1523
1523
1524
1524
1525
1525
1526
1526
1527
1527
1528
1528
1529
1529
1530
1530
1531
1531
1532
1532
1533
1533
1534
1534
1535
1535
1536
1536
1537
1537
1538
1538
1539
1539
1540
1540
1541
1541
1542
1542
1543
1543
1544
1544
1545
1545
1546
1546
1547
1547
1548
1548
1549
1549
1550
1550
1551
1551
1552
1552
1553
1553
1554
1554
1555
1555
1556
1556
1557
1557
1558
1558
1559
1559
1560
1560
1561
1561
1562
1562
1563
1563
1564
1564
1565
1565
1566
1566
1567
1567
1568
1568
1569
1569
1570
1570
1571
1571
1572
1572
1573
1573
1574
1574
1575
1575
1576
1576
1577
1577
1578
1578
1579
1579
1580
1580
1581
1581
1582
1582
1583
1583
1584
1584
1585
1585
1586
1586
1587
1587
1588
1588
1589
1589
1590
1590
1591
1591
1592
1592
1593
1593
1594
1594
1595
1595
1596
1596
1597
1597
1598
1598
1599
1599
1600
1600
1601
1601
1602
1602
1603
1603
1604
1604
1605
1605
1606
1606
1607
1607
1608
1608
1609
1609
1610
1610
1611
1611
1612
1612
1613
1613
1614
1614
1615
1615
1616
1616
1617
1617
1618
1618
1619
1619
1620
1620
1621
1621
1622
1622
1623
1623
1624
1624
1625
1625
1626
1626
1627
1627
1628
1628
1629
1629
1630
1630
1631
1631
1632
1632
1633
1633
1634
1634
1635
1635
1636
1636
1637
1637
1638
1638
1639
1639
1640
1640
1641
1641
1642
1642
1643
1643
1644
1644
1645
1645
1646
1646
1647
1647
1648
1648
1649
1649
1650
1650
1651
1651
1652
1652
1653
1653
1654
1654
1655
1655
1656
1656
1657
1657
1658
1658
1659
1659
1660
1660
1661
1661
1662
1
```

```

58     drop(inner);
59     if option.contains(WaitOption::WNOHANG) {
60         return SUCCESS;
61     } else {
62         block_current_and_run_next();
63         debug!("[sys_wait4] --resumed--");
64     }
65 }
66 }
67 }
```

sys\_wait4 函数的参数为指定进程的 pid、可存储退出码的用户空间区域的指针 status、函数执行方式的选项 option、以及可存储进程资源占用信息的 ru 指针。

第 11~19 行，我们判断当前进程是否具有符合要求的子进程。当传入的 pid 为 -1 时，任何一个子进程都算是符合要求；但传入的 pid 不为 -1 的时候，则只有子进程的 PID 恰好与传入的 pid 相同时，才算符合条件。若没有符合条件的子进程，则函数直接返回 ECHILD。

第 20~36 行，我们判断符合要求的子进程中是否有僵尸进程，若有，记录它在当前 TCB 子进程向量中的下标；若无，则进入第 57~65 行的处理。

第 38 行，我们将子进程从向量中移除并置于当前上下文中。

第 41 行，我们确认这是对于该子进程控制块的唯一一次强引用，即它不会出现在某个进程的子进程向量中，更不会出现在处理器监控器或者任务管理器中。当它所在的代码块结束，这次引用变量的生命周期结束，将导致该子进程进程控制块的引用计数变为 0，彻底回收掉它占用的所有资源，包括其内核栈、PID 还有应用地址空间存放页表的那些物理页帧等。

第 43~56 行，我们将收集的子进程信息进行返回，包括退出码等，最后以回收的子进程 PID 作为函数返回值返回。

第 57~65 行，是对于没有可回收的子进程的情况的处理。倘若函数选项参数 option 具有 WNOHANG 标志，则说明调用该函数的进程不会被挂起，返回 SUCCESS 继续执行；若无该标志，则会切换至下一个进程执行。

至此，父进程对子进程的回收机制也介绍完毕。下面我们最后再介绍一个进程的退出机制：kill。

如果说 exit 机制是一个进程的“自杀”，则 kill 机制使得一个进程可以“杀死”其他进程。我们来看具体实现：

代码片段 5.42 os/src/syscall/process.rs

```

1 pub fn sys_kill(pid: usize, sig: usize) -> isize {
2     let signal = match Signals::from_signum(sig) {
3         Ok(signal) => signal,
4         Err(_) => return EINVAL,
5     };
6     if pid > 0 {
7         if let Some(task) = find_task_by_tgid(pid) {
```

```

8     if !signal.is_empty() {
9         let mut inner = task.acquire_inner_lock();
10        inner.add_signal(signal);
11        // wake up target process if it is sleeping
12        if inner.task_status == TaskStatus::Interruptible {
13            inner.task_status = TaskStatus::Ready;
14            drop(inner);
15            wake_interruptible(task);
16        }
17    }
18    SUCCESS
19 } else {
20     ESRCH
21 }
22 } else if pid == 0 {
23     todo!()
24 } else if (pid as isize) == -1 {
25     todo!()
26 } else { // (pid as isize) < -1
27     todo!()
28 }
29 }
```

事实上，sys\_kill 系统调用的实现非常简单。该函数有两个参数，一个是目的进程的 pid，一个是将给目的进程赋予的信号 sig。kill 机制的实现实际上就是找出要“杀死”的进程，并给它赋予一个信号量，该信号可以为“自杀”的信号，然后再将该进程唤醒，让其自我终结。

第 6~21 行，就是先使用 find\_task\_by\_tgid 函数来找出目的进程，若有，则为其赋予一个指定的信号，然后再将该进程从阻塞态唤醒，然后返回 SUCCESS；若无，则返回 ESRCH 代表未找到目的进程。

如你所见，本系统调用仅是 kill 机制的初步实现，还未最终完善。

## 第 6 章 初识块设备

### 6.1 驱动程序

驱动程序是一种软件组件，是操作系统与机外设之间的接口，可让操作系统和设备彼此通信。从操作系统架构上看，驱动程序与 I/O 设备靠的更近，离应用程序更远，这使得驱动程序需要站在协助所有进程的全局角度来处理各种 I/O 操作。这也就意味着在驱动程序的设计实现中，尽量不要与单个进程建立直接的联系，而是在全局角度对 I/O 设备进行统一处理。

上面只是介绍了 CPU 和 I/O 设备之间的交互手段。如果从操作系统角度来看，我们还需要对特定设备编写驱动程序。它一般需包括如下一些操作：

- 定义设备相关的数据结构，包括设备信息、设备状态、设备操作标识等
- 设备初始化，即完成对设备的初始配置，分配 I/O 操作所需的内存，设置好中断处理例程
- 如果设备会产生中断，需要有处理这个设备中断的中断处理例程（Interrupt Handler）
- 根据操作系统上层模块（如文件系统）的要求（如读磁盘数据），给 I/O 设备发出命令，检测和处理设备出现的错误
- 与操作系统上层模块或应用进行交互，完成上层模块或应用的要求（如上传读出的磁盘数据）

从驱动程序 I/O 操作的执行模式上看，主要有两种模式的 I/O 操作：异步和同步。同步模式下的处理逻辑类似函数调用，从应用程序发出 I/O 请求，通过同步的系统调用传递到操作系统内核中，操作系统内核的各个层级进行相应处理，并最终把相关的 I/O 操作命令转给了驱动程序。一般情况下，驱动程序完成相应的 I/O 操作会比较慢（相对于 CPU 而言），所以操作系统会让代表应用程序的进程进入等待状态，进行进程切换。但相应的 I/O 操作执行完毕后（操作系统通过轮询或中断方式感知），操作系统会在合适的时机唤醒等待的进程，从而进程能够继续执行。

异步 I/O 操作是一个效率更高的执行模式，即应用程序发出 I/O 请求后，并不会等待此 I/O 操作完成，而是继续处理应用程序的其它任务（这个任务切换会通过运行时库或操作系统来完成）。调用异步 I/O 操作的应用程序需要通过某种方式（比如某种异步通知机制）来确定 I/O 操作何时完成。

## 6.2 SD 卡的驱动框架

### 6.2.1 SD/MMC 卡介绍

#### (1) 什么是 MMC 卡

MMC 是 MultiMediaCard 的缩写，即多媒体卡。它是一种非易失性存储器件，体积小巧 (24mm\*32mm\*1.4mm)，容量大，耗电量低，传输速度快，广泛应用于消费类电子产品中。

#### (2) 什么是 SD 卡

SD 卡为 Secure Digital Memory Card，即安全数码卡。它在 MMC 的基础上发展而来，增加了两个主要特色：SD 卡强调数据的安全性，可以设定所储存的使用权限，防止数据被他人复制；另外一个特色就是传输速度比 2.11 版的 MMC 卡快。在数据传输和物理规范上，SD 卡 (24mm\*32mm\*2.1mm，比 MMC 卡更厚一点)，向前兼容了 MMC 卡。所有支持 SD 卡的设备也支持 MMC 卡。SD 卡和 2.11 版的 MMC 卡完全兼容。SD 卡内部结构如图所示：

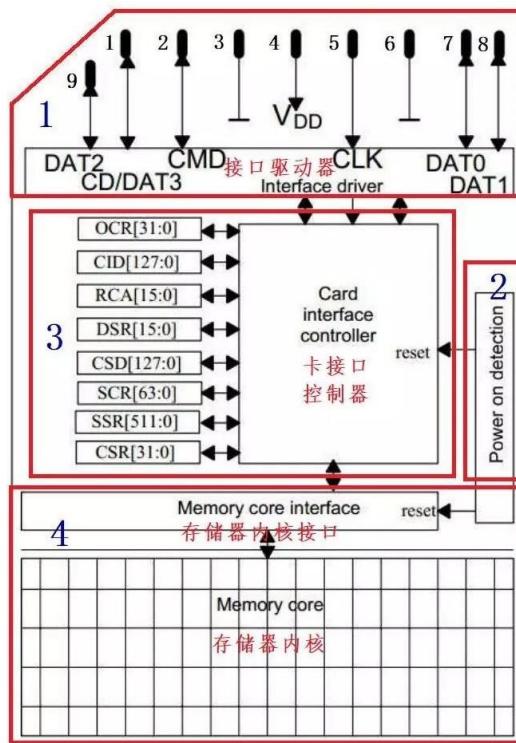


图 6-1 SD 卡内部结构

#### (3) 什么是 SDIO

SDIO 是在 SD 标准上定义了一种外设接口，它和 SD 卡规范间的一个重要区别是增加了低速标准。在 SDIO 卡只需要 SPI 和 1 位 SD 传输模式。低速卡的目标应用是以最小的硬件开销支持低速 IO 能力。

#### (4) 什么是 MCI

MCI 是 Multimedia Card Interface 的简称，即多媒体卡接口。上述的 MMC,SD,SDI 卡定义的接口都属于 MCI 接口。MCI 这个术语在驱动程序中经常使用，很多文件，函数名字都包括“mci”。

#### (5) MMC/SD/SDIO 卡的区别

卡属性	MMC 卡	SD 卡	SDIO 卡
引脚个数	7	9	9
宽度	24mm	24mm	24mm
长度	32mm	32mm	32mm+
厚度	1.4mm	2.1mm	2.1mm
SPI 传输模式	可选	支持	支持
一位传输模式	是	是	是
四位传输模式	否	可选	可选
时钟频率	0-20MHz	0-25MHz	0-25MHz
最高传输率	20Mbit/s	100Mbit/s	100Mbit/s
最高 SPI 传输率	20Mbit/s	25Mbit/s	25Mbit/s

表 6-1 卡对比表

SD 卡内部有 7 个寄存器，如表 6-2 所示。其中 OCR,CID,CSD 和 SCR 寄存器保存卡的配置信息;RCA 寄存器保存着通信过程中卡当前暂时分配的地址(只适合 SD 模式);卡状态(Card Status)和 SD 状态(SD Status)寄存器保存着卡的状态(例如，是否写成功，通信的 CRC 校验是否正确等)，这两个寄存器的内容与通信模式(SD 模式或 SPI 模式)相关。MMC 卡没有 SCR 和 SD Status 寄存器。

名字	宽度	描述
CID	128	卡识别号，每张卡都有的唯一识别号
RCA	16	发布卡的地址，卡的局部系统地址，在初始化过程中，由主机和卡动态支持
DSR	16	驱动级寄存器，配置卡的驱动输出
CSD	128	卡的协议数据，关于卡的操作状态数据
SCR	64	卡配置寄存器，关于卡特性容量的信息
OCR	32	操作状态寄存器
SSR	512	SD 状态，有关卡拥有的特性信息
CSR	32	卡状态，有关卡状态的信息

表 6-2 卡寄存器表

OCR 寄存器保存着 SD/MMC 卡的供电电允许范围。如表 6-3 所示：如果 OCR 寄存器的某位为 1，表示卡支持该位对应的电压。最后一位表示卡上电后的状态(是否处于“忙状态”)，如果该位为 0，表示忙，如果为 1，表示处于空闲状态(MMC/SD 协议 P60)。

OCR bit position	VDD votage window
0-3	Reserved
4	1.6-1.7
5	1.7-1.8
6	1.8-1.9
7	1.9-2.0
8	2.0-2.1
9	2.1-2.2
10	2.2-2.3
11	2.3-2.4
12	2.4-2.5
13	2.5-2.6
14	2.6-2.7
15	2.7-2.8
16	2.8-2.9
17	2.9-3.0
18	3.0-3.1
19	3.1-3.2
20	3.2-3.3
21	3.3-3.4
22	3.4-3.5
23	3.5-3.6
24-30	Reserved
31	Card power up status bit (busy)1

表 6-3 电平-比特位表

CID 为一个 16 个字节的寄存器，该寄存器包含一个独特的卡标识号。

Name	Field	Width	CID-slice
Manufacturer ID	MID	8	[127:120]
OEM/Application ID	OID	18	[119:104]
Product name	PNM	40	[103:64]
Product Revision	PRV	8	[83:56]
Product serial number	PSN	32	[55:24]
Reserved	...	4	[23:20]
Manufacturing Date	MDT	12	[19:8]
CRC7 Checksum	CRC	7	[7:1]
not used, always "T"	-	1	[0:0]

表 6-4 卡识别号表

SD/SDIO 有以下几种传输模式：

- SPI mode，独立序列输入和独立序列输出，使 CS、DI、SCLK 与 DO（SD 卡的片选、数据输入、时钟与数据输出）四根信号线进行数据传输。
- 1-bit mode，独立指令和数据通道，只支持 1 位宽的数据传输。

- 4-bit mode，使用额外的针脚以及某些重新设置的针脚。支持四位宽的并行传输。

### 6.2.2 K210 联动 SD 卡

K210 裸机使用 SD 卡，下图是 SD 卡对应接口：

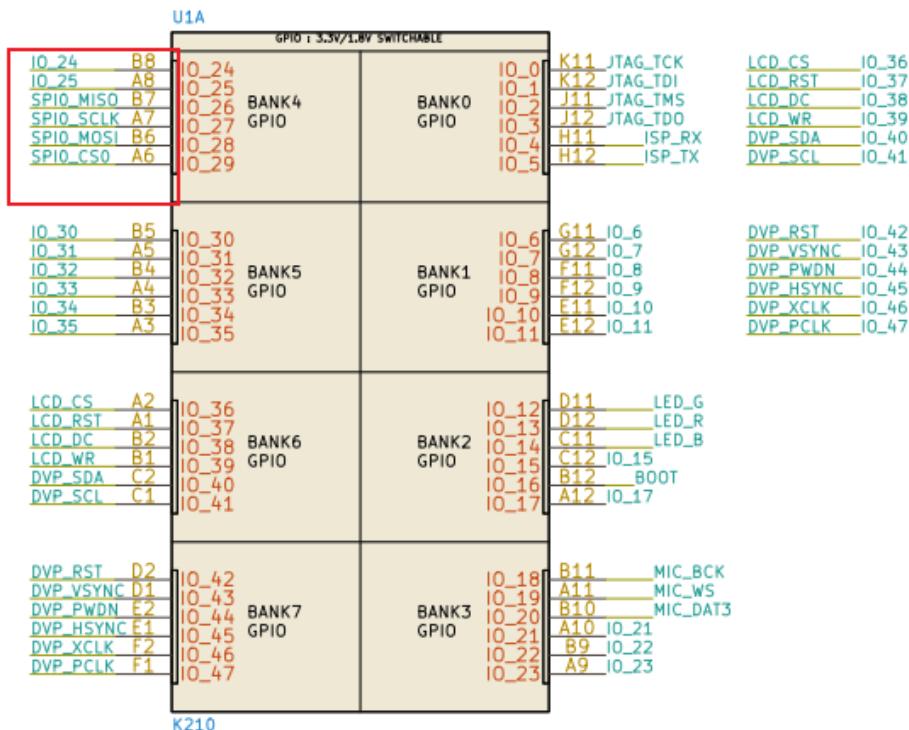


图 6-2 SD 卡与 K210 接口

为了协同软硬接口，我们需要在代码中定义相关常量，使软硬接口保持一致，这即为遵守 SD 卡协议。

代码片段 6.1 SD 协议

```

1 const MMIO =[
2     (0x0C00_0000, 0x3000), /* PLIC */
3     (0x0C20_0000, 0x1000), /* PLIC */
4     (0x3800_0000, 0x1000), /* UARTHS */
5     (0x3800_1000, 0x1000), /* GPIOHS */
6     (0x5020_0000, 0x1000), /* GPIO */
7     (0x5024_0000, 0x1000), /* SPI_SLAVE */
8     (0x502B_0000, 0x1000), /* FPIOA */
9     (0x502D_0000, 0x1000), /* TIMER0 */
10    (0x502E_0000, 0x1000), /* TIMER1 */
11    (0x502F_0000, 0x1000), /* TIMER2 */
12    (0x5044_0000, 0x1000), /* SYSCTL */
13    (0x5200_0000, 0x1000), /* SPI0 */
14    (0x5300_0000, 0x1000), /* SPI1 */
15    (0x5400_0000, 0x1000), /* SPI2 */
16];

```

应用程序通过文件系统接口如 open()、read()、write()、close() 等访问文件系统，根据文件系统 inode 节点，接着找到文件在 SD 卡驱动上的块号。文件系统通过块设备驱

动层与 SD 卡协议层对接，块设备驱动层定义了抽象块设备的接口，主要包括对块设备的读写接口。SD 卡协议层主要负责按照 SD 卡标准规范向 SD 卡发送指令或者接收响应数据；硬件接口层则负责按照硬件板卡的引脚定义操作 GPIO 或 SPI 引脚实现与 SD 卡的数据交互。

### 6.2.3 SD 卡的命令

SD 卡命令共分为 12 类，分别为 class0 到 Class11（注：完整细节请自行参考 SD 卡协议官方文档）。

Class0：卡的识别、初始化等基本命令集

- CMD0: 复位 SD 卡。
- CMD1: 读 OCR 寄存器。
- CMD9: 读 CSD 寄存器。
- CMD10: 读 CID 寄存器。
- CMD12: 停止读多块时的数据传输。
- CMD13: 读 Card\_Status 寄存器。

Class2：读卡命令集

- CMD16: 设置块的长度。
- CMD17: 读单块。
- CMD18: 读多块，直至主机发送 CMD12 为止。

Class4：写卡命令集

- CMD24: 写单块。
- CMD25: 写多块。
- CMD27: 写 CSD 寄存器。

Class5：擦除卡命令集

- CMD32: 设置擦除块的起始地址。
- CMD33: 设置擦除块的终止地址。
- CMD38: 擦除所选择的块。

Class6：写保护命令集

- CMD28: 设置写保护块的地址。
- CMD29: 擦除写保护块的地址。
- CMD30: Ask the card for the status of the write protection bits

Class7：卡的锁定，解锁功能命令集。

Class8：申请特定命令集。

Class10 - 11：保留。

#### 6.2.4 SD 卡工作流程

SD 卡工作流程大致可以分为 3 个大的步骤：初始化 SD 卡、写 SD 卡、读 SD 卡。在 SPI 模式下，SD 卡工作模式分为卡识别模式和数据传输模式。如下图为卡在识别模式下的命令流程。

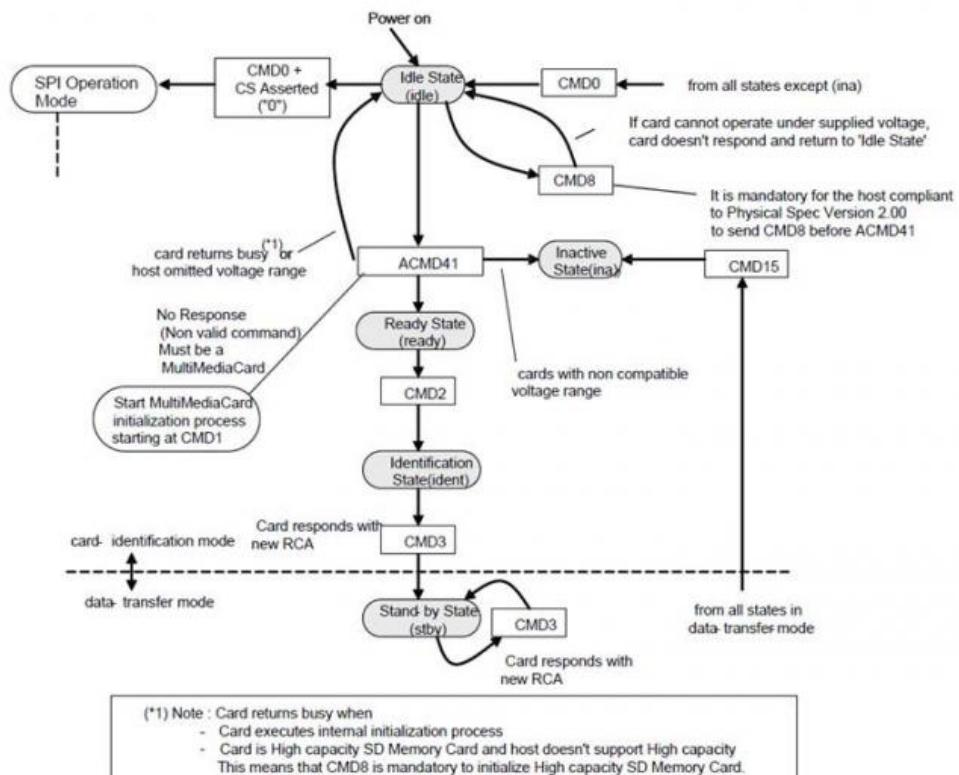


图 6-3 SD 卡识别模式命令流程

在复位后，查找总线上的新卡的时候，主机会处于“卡识别模式”。卡在复位后会处于识别模式。不同的 SD 卡可能支持不同版本的协议或者不同的 工作电压。因此，作为主机，在与 SD 卡进行交互之初，主机需要获取卡的工作电压范围和卡的类型。在卡识别期间，时钟频率应该保持在 100~400kHz 之间。

1. 在主机和 SD 卡进行任何通信之前，主机不知道 SD 卡支持的工作电压范围，卡也不知道是否支持主机当前提供的电压。因此主机首先使用默认的电压发送一条 reset 指令 (CMD0)。
  2. 为了验证 SD 卡的接口操作状态，主机发送 SEND\_IF\_COND(CMD8)，用于取得 SD 卡支持工作的电压范围数据。SD 卡通过检测 CMD8 的参数部分来检查主机使用的工作电压，主机通过分析回传的 CMD8 参数部分来校验 SD 卡是否可以在所给电压下工作，如果 SD 卡可以在指定电压下工作，则它回送 CMD8 的命令响应字。如果不支持所给电压，则 SD 卡不会给出任何响应信息，并继续处于 IDLE 状态。
  3. 在发送 ACMD41 命令初始化高容量的 SD 卡前，需要强制发送 CMD8 命令。强制低电压主机在发送 CMD8 前发送 ACMD41，万一双重电压 SD 卡没有收到 CMD8

命令且工作在高电压状态，在这种情况下，低电压主机不能不发送 CMD8 命令给卡，则收到 ACMD41 后进入无活动状态。

4. SD\_SEND\_OP\_COND(ACMD) 命令是为 SD 卡主机识别卡或者电压不匹配时拒绝卡的机制设计的。主机发送命令操作数代表要求的电压窗口大小。如果 SD 卡在所给的范围内不能实现数据传输，将放弃下一步的总线操作而进入无活动。操作状态寄存器也将被定义。
5. 在主机发出复位命令 (CMD0) 后，主机将先发送 CMD8 再发送 ACMD41 命令重新初始化 SD 卡。

当总线被激合后，主机就开始卡的初始化和识别 3 处理。初始化处理设置它的操作状态和是设置 OCR 中的 HCS 比特命令 SD\_SEND\_OP\_COND(ACMD41) 开始。HCS 比特位被设置为 1 表示主机支持高容量 SD 卡。HCS 被设置为 0 表示主机不支持高容量 SD 卡。卡的初始化和识别流程见下图。

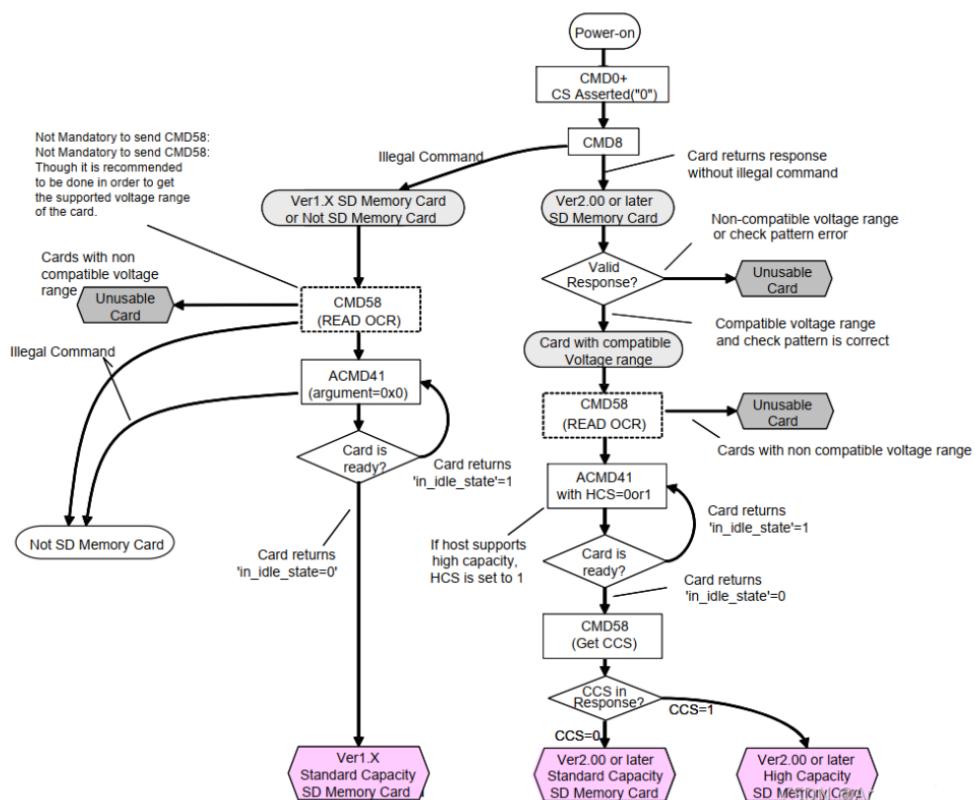


图 6-4 卡初始化和识别流程

(注：这里不推荐使用 NPUCore 或 rCore 作为代码范式来讲解，应直接看 SD 卡协议文档)

卡在识别模式结束后，主机时钟 fpp(数据传输时钟频率) 将保存为 fod(卡识别模式下的时钟)，由于有些卡对操作时钟有限制。主机必须发送 SEND\_CSD(CMD9) 来获得卡规格数据积存器内容，如块大小，卡容量。广播命令 SET\_DSR(CMD4) 配置所有识别

卡的驱动阶段。它对 DSR 积存器进行编程以适应应用总线布局，总线上的卡数目和数据传输频率。SD 卡数据传输模式的流程图如下图。

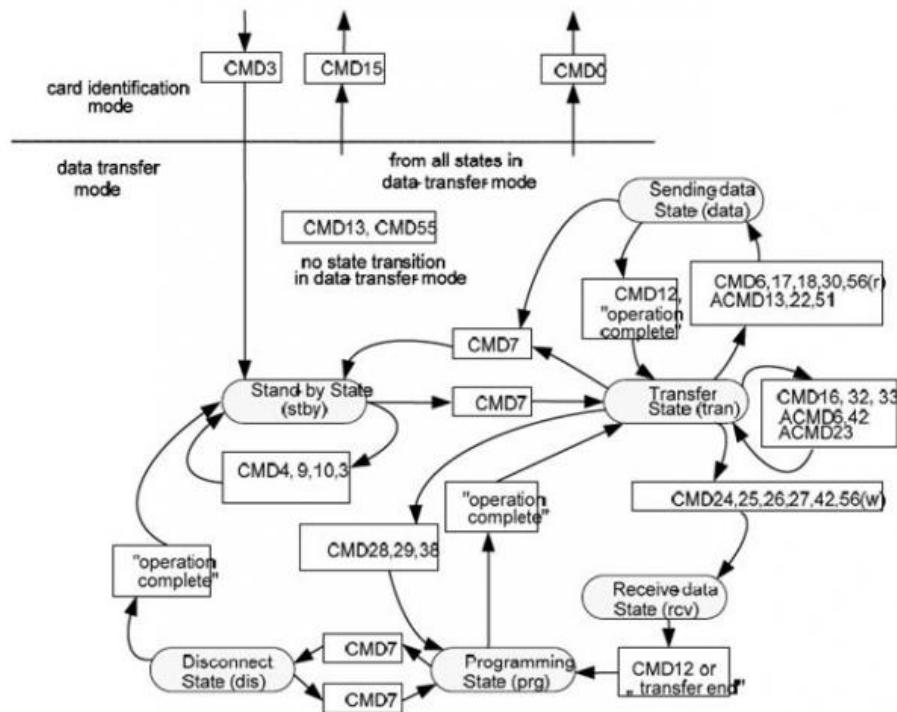


图 6-5 SD 卡数据传输模式流程

1. CMD7 命令用来选择某个 SD 卡，使其进入 Transfer 状态，在指定时间段内，只有一个卡能处于 Transfer 状态。当某个先前被选中的处于 Transfer 状态的 SD 卡接收到 CMD7 之后，会释放与控制器的连接，并进入 Stand-by 状态。当 CMD7 使用保留地址 0x0000 时，所有的 SD 卡都会进入 Stand-by 状态。
2. 所有的数据读命令都可以被停止命令 (CMD12) 在任意时刻终止。数据传输会终止，SD 卡返回 Transfer 状态。读命令有：块读操作 (CMD17)、多块读操作 (CMD18)、发送写保护 (CMD30)、发送 scr(ACMD51) 以及读模式下的普通命令 (CMD56)。
3. 所有的数据写命令都可以被停止命令 (CMD12) 在任意时刻终止。写命令也会在取消选择命令 (CMD7) 之前停止。写命令有：块写操作 (CMD24, CMD25)、编程命令 (CMD27)、锁定/解锁命令 (CMD42) 以及写模式下的普通命令 (CMD56)。
4. 数据传输一旦完成，SD 卡会退出数据写状态，进入 Programming 状态 (传输成功) 或者 Transfer 状态 (传输失败)。

### 6.3 QEMU 下 SD 卡的驱动理解

本节介绍在非实物环境下，使用 qemu 模拟 SD 卡块设备条件下的 SD 卡驱动。在 qemu 环境下，块设备驱动程序主要依赖 virtio，本节介绍 virtio 标准相关内容及 virtio 下 host 和 guest 之间设备数据交互过程，最后使用实现该标准的第三方依赖库介绍 npucore 中 QEMU 下块设备驱动的实现。

随着互联网和云计算的兴起，物理服务器上通过虚拟机技术运行多个虚拟机，并在虚拟机中运行 guest 操作系统的模式成为主流。在传统的全虚拟化解决方案中，guest OS 要使用 host 资源，需要 Hypervisor 来截获所有的请求指令，然后模拟出这些指令的行为，半虚拟化通过底层硬件辅助的方式，将部分没必要虚拟化的指令通过硬件来完成，Hypervisor 只负责完成部分指令的虚拟化，guest 完成不同设备的前端驱动程序，Hypervisor 配合 guest 完成相应的后端驱动程序，两者之间通过某种交互机制实现虚拟化过程。由于不同 guest 前端设备其工作逻辑大同小异，单独为每个设备定义一套接口实属没有必要，而且还要考虑扩平台的兼容性问题，另外，不同后端 Hypervisor 的实现方式也类似（如 KVM、Xen 等）。所以，需要一套通用框架和标准接口（协议）来完成两者之间的交互过程。

《virtio: towards a de-facto standard for virtual I/O devices》首次提出了 Virtio 半虚拟化 Io 模型，它是一套通用 I/O 设备虚拟化的程序，是对半虚拟化 Hypervisor 中的一组通用 I/O 设备的抽象。提供了一套上层应 与各 Hypervisor 虚拟化设备（KVM，Xen，VMware 等）之间的通信框架和编程接口，减少跨平台所带来的兼容性问题，大大提高驱动程序开发效率。

#### 6.3.1 virtio 架构

VirtIO 由 Rusty Russell 开发，对准虚拟化 hypervisor 中的一组通用模拟设备 IO 的抽象。Virtio 是一种前后端架构，包括前端驱动（Guest 内部）、后端设备（QEMU 设备）、传输协议（vring）。框架如下图所示：

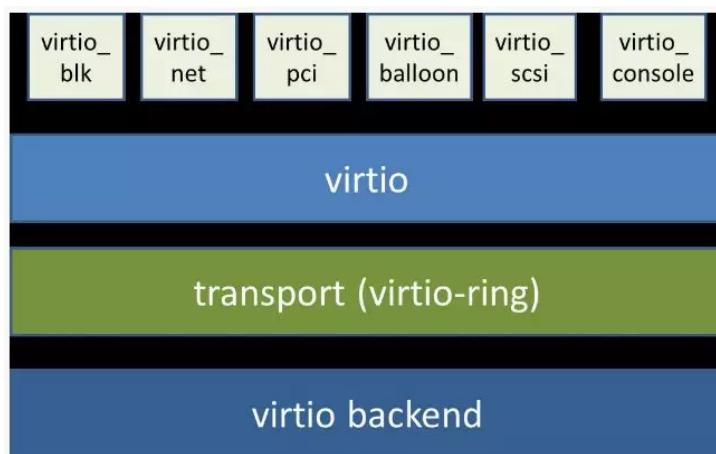


图 6-6 virtio 架构

- 前端驱动：虚拟机内部的 virtio 模拟设备对应的驱动。作用为接收用户态的请求，然后按照传输协议对请求进行封装，再写 I/O 操作，发送通知到 QEMU 后端设备。
- 后端设备：在 QEMU 中创建，用来接收前端驱动发送的 I/O 请求，然后按照传输协议进行解析，在对物理设备进行操作，之后通过终端机制通知前端设备。
- 传输协议：使用 virtio 队列（virtio queue, virtqueue）完成。设备有若干个队列，每个队列处理不同的数据传输（如 virtio-balloon 包含 ivq、dvq、svq 三个）。virtqueue 通过 vring 实现。Vring 是虚拟机和 QEMU 之间共享的一段环形缓冲区，QEMU 和前端设备都可以从 vring 中读取数据和放入数据。

QEMU 如何感知虚拟机的操作的？虚拟机内调用 kick 函数实现通知之后，会产生 KVM\_EXIT。Host 端的 kvm 模块捕获到这个 EXIT 之后，根据它退出的原因来做处理。如果是一个 IO\_EXIT，kvm 会将这个退出交给用户态的 QEMU 程序来完成 IO 操作。QEMU 为 kvm 虚拟机模拟了 virtio 设备，因此后端的 virtio-pci 设备也是在 QEMU 进程中模拟生成的。QEMU 对模拟的 PCI 设备的配置空间注册了回调函数，当虚拟机产生 IO\_EXIT，就调用这些函数来处理事件。

virtio 设备通过特定于总线的方法发现和识别，每个设备由以下几个要素组成：

- 设备状态字：设备状态字指示了在驱动程序初始化设备期间的设备状态，定义以下几个状态：

状态	含义
ACKNOWLEDGE(0x00000001)	guest OS 已找到设备，并将其识别为有效的 virtio 设备。
DRIVER(0x00000010)	guest OS 知道如何驱动设备
FAILED(0x10000000)	设备出现问题
FEATURES_OK(0x00000100)	指示驱动程序以取人其所有功能
DRIVER_OK(0x00000100)	指示驱动程序已设置并准备好驱动
DEVICE_NEEDS_RESET(0x01000000)	指示设备需要重启

表 6-5 设备状态字

- 特征位：特征位用于表示 virtio 设备具有的各种特性和功能，其中 bit0 - 23 是特定设备可以使用的 feature bits，bit24 - 37 预给队列和 feature 协商机制，bit38 以上保留给未来其他用途。驱动程序与设备对设备特性进行协商，形成一致的共识，这样才能正确的管理设备。
- 通知：通知是指设备和驱动程序必须通知对方，它们有数据需要对方处理，有三种类型的通知：配置更改通知，设备-> 驱动程序，指示设备配置空间已更改；可用缓冲区通知，驱动程序-> 设备，指示缓冲区可用；已使用缓冲区通知，设备-> 驱动程序，指示缓冲区已经使用。
- 设备配置空间：设备配置空间通常用于配置不常变动的设备参数（属性），或者初始化阶段需要设置的设备参数。设备的特征位中包含表示配置空间是否存在 bit 位，并可通过在特征位的末尾添加新的 bit 位来扩展配置空间。设备驱动程序在初

初始化 virtio 设备时，需要根据 virtio 设备的特征位和配置空间来了解设备的特征，并对设备进行初始化。

- **virtqueue 虚拟队列：**在 virtio 设备上进行批量数据传输的机制被称为虚拟队列（virtqueue），每个设备可以有零个或多个虚拟队列。设备驱动程序向该队列中添加缓冲区描述符并向设备发送可用缓冲区通知，通知设备从队列中读取请求；设备处理完请求后将使用的缓冲区标记为已使用并添加到队列中，向设备驱动程序发送已用缓冲区同通知，设备驱动程序从已用缓冲区中读取数据。

### 6.3.2 virtio 块驱动整体流程

从代码上看，virtio 的代码主要分两个部分：QEMU 和内核驱动程序。Virtio 设备的模拟就是通过 QEMU 完成的，QEMU 代码在虚拟机启动之前，创建虚拟设备。虚拟机启动后检测到设备，调用内核的 virtio 设备驱动程序来加载这个 virtio 设备。

对于 KVM 虚拟机，都是通过 QEMU 这个用户空间程序创建的，每个 KVM 虚拟机都是一个 QEMU 进程，虚拟机的 virtio 设备是 QEMU 进程模拟的，虚拟机的内存也是从 QEMU 进程的地址空间内分配的。

VRING 是由虚拟机 virtio 设备驱动创建的用于数据传输的共享内存，QEMU 进程通过这块共享内存获取前端设备递交的 IO 请求。如下图所示，虚拟机 IO 请求的整个流程：

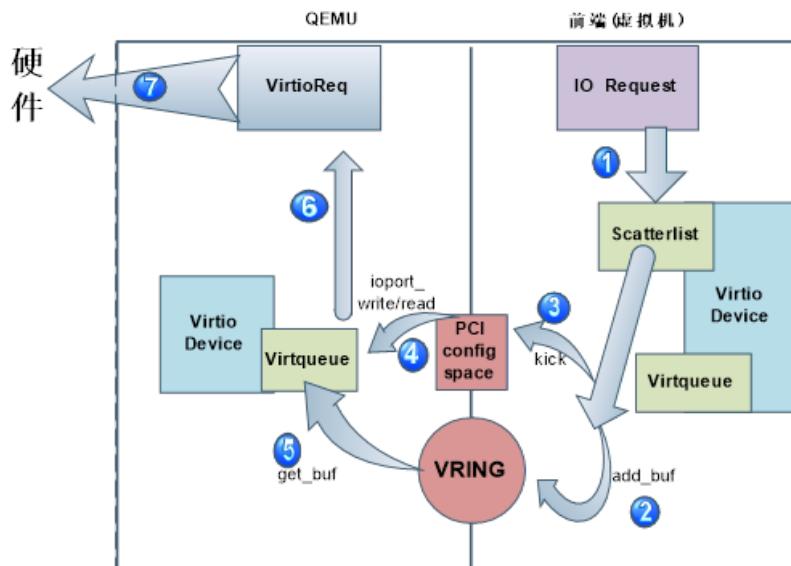


图 6-7 虚拟机 IO 请求流程

1. 虚拟机产生的 IO 请求会被前端的 virtio 设备接收，并存放在 virtio 设备散列表 scatterlist 里；
2. Virtio 设备的 virtqueue 提供 add\_buf 将散列表中的数据映射至前后端数据共享区域 Vring 中；
3. Virtqueue 通过 kick 函数来通知后端 qemu 进程。Kick 通过写 pci 配置空间的寄存

器产生 kvm\_exit;

4. Qemu 端注册 ioport\_write/read 函数监听 PCI 配置空间的改变，获取前端的通知消息；
5. Qemu 端维护的 virtqueue 队列从数据共享区 vring 中获取数据；
6. Qemu 将数据封装成 virtioreq；
7. Qemu 进程将请求发送至硬件层。

前端主要通过 PCI 配置空间的寄存器完成前后端的通信，而 IO 请求的数据地址则存在 vring 中，并通过共享 vring 这个区域来实现 IO 请求数据的共享。

从上图中可以看到，Virtio 设备的驱动分为前端与后端：前端是虚拟机的设备驱动程序，后端是 host 上的 QEMU 用户态程序。为了实现虚拟机中的 IO 请求从前端设备驱动传递到后端 QEMU 进程中，Virtio 框架提供了两个核心机制：前后端消息通知机制和数据共享机制。

- 消息通知机制：前端驱动设备产生 IO 请求后，可以通知后端 QEMU 进程去获取这些 IO 请求，递交给硬件。
- 数据共享机制：前端驱动设备在虚拟机内申请一块内存区域，将这个内存区域共享给后端 QEMU 进程，前端的 IO 请求数据就放入这块共享内存区域，QEMU 接收到通知消息后，直接从共享内存取数据。由于 KVM 虚拟机就是一个 QEMU 进程，虚拟机的内存都是 QEMU 申请和分配的，属于 QEMU 进程的线性地址的一部分，因此虚拟机只需将这块内存共享区域的地址传递给 QEMU 进程，QEMU 就能直接从共享区域存取数据。下图为数据共享机制可视化图。

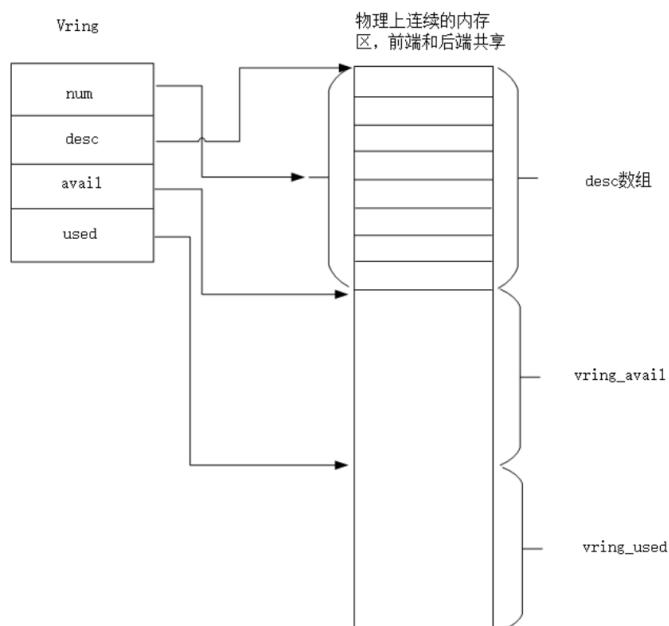


图 6-8 数据共享机制可视化

# 第 7 章 理解文件系统

## 7.1 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。

这里简要介绍一下在内核中添加文件系统的大致开发过程。

### 第一步：是能够写出与文件访问相关的应用

这里是参考了 Linux 的创建/打开/读写/关闭文件的系统调用接口，力图实现一个简化版的文件系统模型。在用户态我们只需要遵从相关系统调用的接口约定，在用户库里完成对应的封装即可。

### 第二步：实现 easyfs 文件系统

由于 Rust 语言的特点，我们可以在用户态实现文件系统，并在用户态完成文件系统功能的基本测试并基本验证其实现正确性之后，就可以放心的将该模块嵌入到操作系统内核中。当然，有了文件系统的具体实现，还需要对上一章的操作系统内核进行扩展，实现文件系统对接的接口，这样才可以让操作系统拥有一个简单可用的文件系统。这样内核就可以支持具有文件读写功能的复杂应用。当内核进一步支持应用的命令行参数后，就可以进一步提升应用程序的灵活性，让应用的开发和调试变得更为轻松。文件系统的整体架构自下而上可分为五层：

- 磁盘块设备接口层：读写磁盘块设备的 trait 接口
- 块缓存层：位于内存的磁盘块数据缓存
- 磁盘数据结构层：表示磁盘文件系统的数据结构
- 磁盘块管理器层：实现对磁盘文件系统的管理
- 索引节点层：实现文件创建/文件打开/文件读写等操作

它的最底层就是对块设备的访问操作接口。两个函数 `read_block` 和 `write_block`，分别代表将数据从块设备读到内存缓冲区中，或者将数据从内存缓冲区写回到块设备中，数据需要以块为单位进行读写。

尽管在操作系统的最底层（即块设备驱动程序）已经有了对块设备的读写能力，但

从编程方便/正确性和读写性能的角度来看，仅有块读写这么基础的底层接口是不足以实现高效的文件系统。比如，某应用将一个块的内容读到内存缓冲区，对缓冲区进行修改，并尚未写回块设备时，如果另外一个应用再次将该块的内容读到另一个缓冲区，而不是使用已有的缓冲区，这将会造成数据不一致问题。此外还有可能增加很多不必要的块读写次数，大幅降低文件系统的性能。因此，通过程序自动而非程序员手动地对块缓冲区进行统一管理也就很必要了，该机制被我们抽象为第二层，即块缓存层。

有了块缓存，我们就可以在内存中方便地处理文件系统在磁盘上的各种数据了，这就是第三层文件系统的磁盘数据结构。

文件系统中的所有需要持久保存的数据都会放到磁盘上，这包括了管理这个文件系统的超级块 (Super Block)，管理空闲磁盘块的索引节点位图区和数据块位图区，以及管理文件的索引节点区和放置文件数据的数据块区组成。文件系统中管理这些磁盘数据的控制逻辑主要集中在磁盘块管理器中，这是文件系统的第四层。

对于单个文件的管理和读写的控制逻辑主要是索引节点（文件控制块）来完成，这是文件系统的第五层。

### 第三步：把 easyfs 文件系统加入内核中

这还需要做两件事情，第一件是在 Qemu 模拟的 virtio 块设备上实现块设备驱动程序。由于我们可以直接使用 virtio-drivers crate 中的块设备驱动，所以只要提供这个块设备驱动所需要的内存申请与释放以及虚实地址转换的 4 个函数就可以了。而我们之前操作系统中的虚存管理实现中，已经有这些函数，这使得块设备驱动程序很简单，且具体实现细节都被 virtio-drivers crate 封装好了。

第二件事情是把文件访问相关的系统调用与 easyfs 文件系统连接起来。在 easfs 文件系统中是没有进程的概念的。而进程是程序运行过程中访问资源的管理实体，而之前的进程没有管理文件这种资源。为此我们需要扩展进程的管理范围，把文件也纳入到进程的管理之中。由于我们希望多个进程都能访问文件，这意味着文件有着共享的天然属性，这样自然就有了 open/close/read/write 这样的系统调用，便于进程通过互斥或共享方式访问文件。

内核中的进程看到的文件应该是一个便于访问的 Inode，这就要对 Inode 结构进一步封装，形成 OSInode 结构，以表示进程中一个打开的常规文件。而进程为了进一步管理多个文件，需要扩展文件描述符表。这样进程通过系统调用打开一个文件后，会将文件加入到自身的文件描述符表中，并进一步通过文件描述符（也就是某个特定文件在自身文件描述符表中的下标）来读写该文件。对于应用程序而言，它理解的磁盘数据是常规的文件和目录，不是 OSInode 这样相对复杂的结构。其实常规文件对应的 OSInode 是操作系统内核中的文件控制块数据结构的实例，它实现了 File Trait 定义的函数接口。这些 OSInode 实例会放入到进程文件描述符表中，并通过 sys\_read/write 系统调用来完成读写文件的服务。这样就建立了文件与 OSInode 的对应关系，通过上面描述的三个开发

步骤将形成包含文件系统的操作系统内核，可给应用提供基于文件的系统调用服务。

## 7.2 块设备接口层

块设备接口层为文件系统提供了对块设备进行读写的操作接口，其代码在 easy-fs/src/block\_dev.rs 中。作为 easy-fs 库的最底层，其声明了一个块设备的抽象接口 BlockDevice，该 trait 定义了设备驱动所需要实现的读写接口，如下：

代码片段 7.1 easy-fs/src/block\_dev.rs

```

1 pub trait BlockDevice: Send + Sync + Any {
2     /// Read block from BlockDevice
3     /// The function panics when the size of 'buf' is not a multiple of
4     /// BLOCK_SZ
5     fn read_block(&self, block_id: usize, buf: &mut [u8]);
6
7     /// Write block into the file system.
8     /// The function panics when the size of 'buf' is not a multiple of
9     /// BLOCK_SZ
10    fn write_block(&self, block_id: usize, buf: &[u8]);
11
12    /// We should rewrite the API for K210 since it supports NATIVE multi-
13    /// block clearing
14    fn clear_block(&self, block_id: usize, num: u8) {
15        self.write_block(block_id, &[num; BLOCK_SZ]);
16    }
17
18    /// We should rewrite the API for K210 if it supports NATIVE multi-
19    /// block clearing
20    fn clear_mult_block(&self, block_id: usize, cnt: usize, num: u8) {
21        for i in block_id..block_id + cnt {
22            self.write_block(i, &[num; BLOCK_SZ]);
23        }
24    }
25 }
```

该 trait 主要实现两个抽象方法：

- `read_block`: 将编号为 `block_id` 的块从磁盘读入内存中的缓冲区 `buf`；
- `write_block`: 将内存中的缓冲区 `buf` 中的数据写入磁盘编号为 `block_id` 的块。

在 easy-fs 中并没有一个实现了 BlockDevice trait 的具体类型，因为块设备仅支持以块为单位进行随机读写，而这两个方法需要由具体的块设备驱动来实现。实际上，这是需要由文件系统的使用者（比如操作系统内核或直接测试 easy-fs 文件系统的 easy-fs-fuse 应用程序）提供并接入到 easy-fs 库的。

块设备接口层及其以下的设备驱动层实现后，向上为接下来要介绍的 easy-fs 库的块缓存层提供这两个方法调用，进行块缓存的管理。换句话说，easy-fs 可以访问实现了 BlockDevice trait 的块设备驱动程序。

同时，NpuCore 为了实现 K210 提供的关于用给定字符清空单个或若干个连续磁盘块的 API，还对该 trait 的 `write_block` 方法进行了进一步封装，增加了 `clear_block` 与 `clear_mult_block` 方法，实现了按照 fat32 文件系统的磁盘块大小，写入给定字符的功能。

至于更复杂的功能，则需要由上层层次进行实现。

Npucore 目前实现了对 SD 卡的驱动实现。在 os/src/drivers/block/sdcard.rs 中对上述 trait 进行了实现，通过 K210 依赖中提供的 GPIO 端口 SPI 串口底层支持，可以对 SD 卡直接进行操作。具体代码如下：

代码片段 7.2 os/src/drivers/block/sdcard.rs

```

1 impl BlockDevice for SDCardWrapper {
2     fn read_block(&self, block_id: usize, buf: &mut [u8]) {
3         let lock = self.0.lock();
4         let mut result = lock.read_sector(buf, block_id as u32);
5         let mut cont_cnt = 0;
6         while result.is_err() {
7             if cont_cnt >= 0 {
8                 log::error!("[sdcard] read_sector(buf, {}) error. Retrying",
9                         "...", block_id);
10                result = lock.read_sector(buf, block_id as u32);
11            }
12            cont_cnt += 1;
13            if cont_cnt >= 5 {
14                log::error!(
15                    "[sdcard] read_sector(buf[{}], {}) error exceeded",
16                    "continuous retry count, waiting...", buf.len(),
17                    block_id
18                );
19                Self::wait_for_one_sec();
20                if lock.read_sector(buf, block_id as u32).is_err() {
21                    lock.init();
22                    Self::wait_for_one_sec();
23                } else {
24                    break;
25                }
26                cont_cnt = 0;
27            }
28        }
29        fn write_block(&self, block_id: usize, buf: &[u8]) {
30            let lock = self.0.lock();
31            let mut result = lock.write_sector(buf, block_id as u32);
32            let mut cont_cnt = 0;
33            while result.is_err() {
34                if cont_cnt >= 0 {
35                    log::error!(
36                        "[sdcard] write_sector(buf, {}) error. Retrying...",
37                        block_id
38                    );
39                    result = lock.write_sector(buf, block_id as u32);
40                }
41                cont_cnt += 1;
42                if cont_cnt >= 5 {
43                    log::error!(
44                        "[sdcard] write_sector(buf[{}], {}) error exceeded",
45                        "continuous retry count, waiting...", buf.len(),
46                        block_id
47                    );
48                }
49            }
50        }
51    }
52}
```

```

48     Self::wait_for_one_sec();
49     if lock.write_sector(buf, block_id as u32).is_err() {
50         lock.init();
51         Self::wait_for_one_sec();
52     } else {
53         break;
54     }
55     cont_cnt = 0;
56 }
57 }
58 }
59 }

```

这两个方法实现了对块号为 block\_id 的 SD 存储区进行直接读写。注意，还设置了在 I/O 故障的情况的读写最大重试次数为 5。

### 7.3 buffer cache 层

文件系统中的块缓存层（Buffer Cache）扮演着非常重要的角色，主要目的是提高文件系统性能和效率。以下是块缓存层的几个关键作用：

1. 减少磁盘 I/O 操作：块缓存层通过在内存中缓存最近或频繁访问的磁盘块来减少对磁盘的直接访问。这是因为内存访问速度远快于磁盘。
2. 加速数据访问：当应用程序请求数据时，操作系统首先在块缓存中查找。如果找到所需的数据块，就可以直接从内存中读取，而无需等待磁盘的慢速读取。
3. 合并写操作：写操作经常被先写入到块缓存中，并在稍后的某个时刻一起写入磁盘。这种方式可以合并多个小的写操作为一个大的磁盘 I/O 操作，提高写入效率。
4. 减少重复数据读取：如果多个应用或进程读取相同的数据块，这些数据块只需要从磁盘读取一次，之后可以从块缓存中获取，减少了重复的磁盘访问。
5. 提供一致性和同步机制：块缓存可以保证文件系统中数据的一致性。例如，在崩溃恢复期间，它可以帮助恢复未完成的写操作，确保文件系统的完整性。
6. 支持异步写操作：数据可以先缓存在块缓存中，然后异步地写入磁盘。这允许程序继续执行而不必等待磁盘 I/O 操作完成。
7. 减轻磁盘的负载：通过减少对磁盘的访问次数，块缓存有助于降低磁盘的工作负载，延长其使用寿命。

NpuCore 中对于文件系统同样实现了 buff cache 层，其主要定义于 easy-fs/src/block-cache.rs 文件中。

首先是对于整个块缓存的定义：

代码片段 7.3 Cache 缓存对象基本操作

```

1 pub trait Cache {
2
3     fn read<T, V>(&self, offset: usize, f: impl FnOnce(&T) -> V) -> V;

```

```

4
5     fn modify<T, V>(&mut self, offset: usize, f: impl FnOnce(&&mut T) -> V)
6         -> V;
7
8     fn sync(&self, _block_ids: Vec<usize>, _block_device: &Arc<dyn
9      BlockDevice>) {}
10
11
12 }
```

这个 trait 定义了一个缓存对象的基本操作。

`read<T, V>(&self, offset: usize, f: impl FnOnce(&T) -> V) -> V`: 允许以只读方式访问缓存。它接受一个偏移量和一个闭包 f, 闭包作用于缓存中的数据，并返回一个结果。这是一个泛型函数，支持不同类型的数据。

`modify<T, V>(&mut self, offset: usize, f: impl FnOnce(&mut T) -> V) -> V`: 类似于 `read`, 但用于可变地修改缓存。它也接受一个偏移量和一个闭包，但闭包可以修改数据。

`sync(&self, _block_ids: Vec<usize>, _block_device: &Arc<dyn BlockDevice>)`: 将缓存中的数据写回到磁盘。接受一个块 ID 列表和一个指向块设备的引用。

为了更好的管理缓存空间，我们需要实现一个 `CacheManager` 对于缓存进行更好的管理，第一点是可以统一管理不同类型的缓存，如块缓存和页表缓存，其次可以实现资源管理和性能优化，`npucore` 中对 oom 的具体实现的是综合了引用计数和优先级两种策略的方法，这有助于提高系统的整体性能，减少磁盘 I/O 操作。

代码片段 7.4 CacheManager 的实现

```

1 pub trait CacheManager {
2     /// The constant to mark the cache size.
3     const CACHE_SZ: usize;
4
5     type CacheType: Cache;
6
7     /// Constructor to the struct.
8     fn new() -> Self
9     where
10        Self: Sized;
11        /// Tell cache manager to write back cache and release memory
12        /// Argument:
13        /// 'neighbor': A closure to get block ids when cache miss.
14        /// 'block_device': The pointer to the block_device.
15        /// Return Value:
16        /// Number of caches freed
17     fn oom<FUNC>(
18         &self,
19         _neighbor: FUNC,
20         _block_device: &Arc<dyn BlockDevice>
21     ) -> usize
22     where
23        FUNC: Fn(usize) -> Vec<usize>
24     {
25         unreachable!()
26     }
27 }
```

```

27  /// When file size changed, we should notify cache manager to drop some
28  /// cache
29  /// Argument:
30  /// 'new_size': File's new size
31  fn notify_new_size(
32      &self,
33      _new_size: usize
34  ) {
35      unreachable!()
36 }

```

这个 trait 定义了缓存管理器的基本功能。

`new() -> Self`: 静态方法，用于创建一个新的缓存管理器实例。

`oom<FUNC>(&self, _neighbor: FUNC, _block_device: &Arc<dyn BlockDevice>) -> usize`: 当系统内存不足时，释放一些缓存。接受一个邻居闭包来获取块 ID，以及一个指向块设备的引用。

`notify_new_size(&self, _new_size: usize)`: 当文件大小变化时，通知缓存管理器释放一些缓存块

然后就是对于缓存管理的分别具体实现，因为我们有块缓存和页缓存两种缓存块，所以我们这里分别对 CacheManager 进一步封装实现了 BlockCacheManager 和 PageManager:

代码片段 7.5 BCM 和 CM

```

1  pub trait BlockCacheManager: CacheManager {
2      /// Try to get the block cache and return None' if not found.    ///
3      /// Argument:      /// 'block_id': The demanded block id(for block cache).
4
5      /// 'inner_cache_id': The ordinal number of the cache inside the file(
6      /// for page cache).
7      /// Return Value:
8      /// If found, return Some(pointer to cache)
9      /// otherwise, return None
10     fn try_get_block_cache(
11         &self,
12         block_id: usize,
13     ) -> Option<Arc<Mutex<Self::CacheType>>>;
14
15     /// Attempt to get block cache from the cache.
16     /// If failed, the manager should try to copy the block from sdcard.
17     /// Argument:
18     /// 'block_id': The demanded block id(for block cache).
19     /// 'inner_cache_id': The ordinal number of the cache inside the file(
20     /// for page cache).
21     /// 'neighbor': A closure to get block ids when cache miss.
22     /// 'block_device': The pointer to the block_device.
23     /// Return Value:
24     /// Pointer to cache

```

```

23 fn get_block_cache<FUNC>(
24     &self,
25     block_id: usize,
26     block_device: &Arc<dyn BlockDevice>,
27 ) -> Arc<Mutex<Self::CacheType>>;
28 }

```

`try_get_block_cache` 该函数尝试获取传入参数 `block_id` (磁盘编号) 和 `inner_cache_id` (管理器内部编号) 对应的 Cache。

`get_block_cache` 该函数获取传入参数 `block_id` (磁盘编号) 和 `inner_cache_id` (管理器内部编号) 对应的 Cache。如果内存中没有，则会通过 `block_device` 从块设备中读取对应的内容：

代码片段 7.6 PageCacheManager

```

1 pub trait PageCacheManager: CacheManager {
2     /// Try to get the block cache and return None' if not found.    ///
3     Argument:      /// block_id': The demanded block id(for block cache).
4
5     /// 'inner_cache_id': The ordinal number of the cache inside the file(
6     for page cache).
7     /// Return Value:
8     /// If found, return Some(pointer to cache)
9     /// otherwise, return None
10    fn try_get_page_cache(
11        &self,
12        inner_cache_id: usize,
13    ) -> Option<Arc<Mutex<Self::CacheType>>>;
14
15    /// Attempt to get block cache from the cache.
16    /// If failed, the manager should try to copy the block from sdcard.
17    /// Argument:
18    /// 'block_id': The demanded block id(for block cache).
19    /// 'inner_cache_id': The ordinal number of the cache inside the file(
20    for page cache).
21    /// 'neighbor': A closure to get block ids when cache miss.
22    /// 'block_device': The pointer to the block_device.
23    /// Return Value:
24    /// Pointer to cache
25    fn get_page_cache<FUNC>(
26        &self,
27        inner_id: usize,
28        neighbor: FUNC,
29        block_device: &Arc<dyn BlockDevice>,
30    ) -> Arc<Mutex<Self::CacheType>>
31    where
32        FUNC: Fn() -> Vec<usize>;
33 }

```

其中两个方法和 `BlockCacheManager` 中两个方法的实现是一样的，唯一区别在于两个管理的 cache (大小) 不同。

## 7.4 索引节点层

我们使用 `ls -l` 时除了看到文件名，还看到了文件元数据

```
1 [root@localhost linux]# ls -l
2 总用量 12
3 -rwxr-xr-x. 1 root root 7438 "9月 13 14:56" a.out
4 -rw-r--r--. 1 root root 654 "9月 13 14:56" test.c
```

为了能解释清楚 inode 我们先简单了解一下 Linux 的文件系统

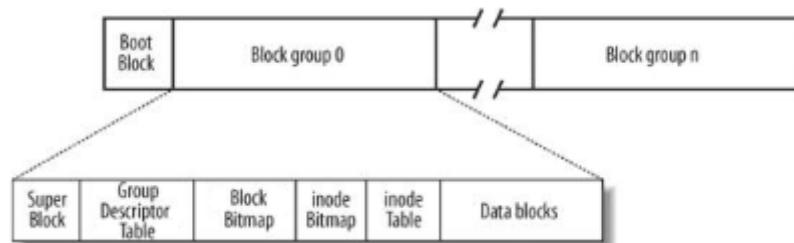


图 7-1 磁盘文件系统图

Linux ext2 文件系统，上图为磁盘文件系统图（内核内存映像肯定有所不同），磁盘是典型的块设备，硬盘分区被划分为一个个的 block。一个 block 的大小是由格式化的时候确定的，并且不可以更改。

一个 block 在磁盘上的布局如下：



图 7-2 block 布局

由上图可知，文件系统的开头通常是由一个磁盘扇区所组成的引导块，该部分的主要目的是用于对操作系统的引导。一般只在启动操作系统时使用。随后是超级块，超级块主要存放了该物理磁盘中文件系统结构的相关信息，并且对各个部分的大小进行说明。inode 块位图，data 块位图记录了在 inode 块和 data 块中哪些块已被使用，以便找到空闲的位置 inode 块和 data 块则是实际存放 inode 和数据的地方，data 存放文件的内容，inode 存放文件元数据并指向所属的 data 的地址

Linux 为了对超级块，inode 块，数据块这三部分进行高效的管理，Linux 创建了几种不同的数据结构，分别是文件系统类型、inode、dentry 等几种。超级块反映了文件系统整体的控制信息。dentry 是反映出某个文件系统对象在全局文件系统树中的位置。inode 则反映了文件系统对象中的一般元数据信息。

**我们可以得出一个结论：inode 存放了文件元数据信息，并指向文件的内容** 所以上文的 '`ls -l`' 实际上就是使用了 inode 的文件元数据而 npucore 中虽然使用了 fat32 文件系统，但在代码层面上同样引入了 inode 这个概念，服务于文件相关系统调用的索引节点

层的代码在 ‘vfs.rs‘ 中。

简单文件系统层实现了磁盘布局并能够将磁盘块有效的管理起来。但是对于文件系统的使用者而言，他们往往不关心磁盘布局是如何实现的，而是更希望能够直接看到目录树结构中逻辑上的文件和目录。为此需要设计索引节点 ‘Inode‘，它屏蔽了简单文件系统层对磁盘的操作，让文件系统的使用者能够直接对文件和目录进行操作。

代码片段 7.7 easy-fs/src/fs/fat32/vfs.rs

```

1 pub struct Inode {
2     /// Inode lock: for normal operation
3     inode_lock: RwLock<InodeLock>,
4     /// File Content
5     file_content: RwLock<FileContent>,
6     /// File cache manager corresponding to this inode.
7     file_cache_mgr: PageCacheManager,
8     /// File type
9     file_type: Mutex<DiskInodeType>,
10    /// The parent directory of this inode
11    parent_dir: Mutex<Option<(Arc<Self>, u32)>>,
12    /// file system
13    fs: Arc<EasyFileSystem>,
14    /// Struct to hold time related information
15    time: Mutex<InodeTime>,
16    /// Info Inode to delete file content
17    deleted: Mutex<bool>,
18 }
```

inode\_lock: 该 inode 的读写锁  
file\_content: 指向文件的内容  
file\_cache\_mgr: 指向管理该 inode 的 PageCacheManager  
file\_type: 文件类型  
parent\_dir: 指向父目录的 inode 以及该 inode 在父目录的位置  
fs: 指向该 inode 所属的简单文件系统  
time: 该文件的相关时间属性  
delete: 一个标识，控制在该 inode 执行析构函数时是否释放文件内容的区域  
利用这个 inode 结构体，文件系统的使用者就可以对文件系统进行一些常用操作：

代码片段 7.8 easy-fs/src/fs/fat32/vfs.rs

```

1 // 创建目录项
2 fn create_dir_ent(
3     &self,
4     inode_lock: &RwLockWriteGuard<InodeLock>,
5     short_ent: FATShortDirEnt,
6     long_ents: Vec<FATLongDirEnt>,
7 ) -> Result<u32, ()>
8
9 // 删除目录项
10 fn delete_dir_ent(
11     &self,
12     inode_lock: &RwLockWriteGuard<InodeLock>,
13     offset: u32,
14 ) -> Result<(), ()>
15
16 // 创建硬链接 (link)
17 impl Inode {
18     pub fn link_par_lock(
19         &self,
```

```

20     inode_lock: &RwLockWriteGuard<InodeLock>,
21     parent_dir: &Arc<Self>,
22     parent_inode_lock: &RwLockWriteGuard<InodeLock>,
23     name: String,
24 ) -> Result<(), ()>
25
26 // 删除硬链接 (unlink)
27 pub fn unlink_lock(
28     &self,
29     _inode_lock: &RwLockWriteGuard<InodeLock>,
30     delete: bool,
31 ) -> Result<(), isize>
32
33 // ls
34 pub fn ls_lock(
35     &self,
36     inode_lock: &RwLockWriteGuard<InodeLock>,
37 ) -> Result<Vec<(String, FATShortDirEnt)>, ()>
38
39 // 获取文件时间相关信息
40 pub fn time(&self) -> MutexGuard<InodeTime>
41
42 // ...

```

## 7.5 目录和路径名

### 7.5.1 目录

最早的文件系统设计简单，只使用文件名来标识文件。这种方法带来了管理上的挑战，因为文件名的唯一性限制了归档和检索的灵活性。随着时间的推移，我们逐渐采用了按功能、属性分类文件的方式，将文件整理到不同层级的目录中。这种层级结构使得我们能够更轻松地逐级定位所需文件。

目录的引入不仅使得文件组织更加清晰，还改善了文件访问权限的管理。通过目录，我们可以间接地控制用户或用户组对特定目录下所有文件的访问权限。这简化了操作系统对多用户环境下文件访问的安全管理。

同时，通过使用 Linux 系统中的 stat 工具，我们可以获取目录的相关信息。这些信息包括目录的权限、所有者、大小以及创建或修改时间等。这些数据对于了解和管理文件系统的结构和属性至关重要，有助于维护和控制文件系统的安全性和可靠性。stat 工具使用实例如下：

```

1 $ stat oskernel2022-npucore
2 File: oskernel2022-npucore
3 Size: 4096          Blocks: 8          IO Block: 4096   directory
4 Device: 805h/2053d  Inode: 589840      Links: 15
5 Access: (0775/drwxrwxr-x)  Uid:(1000/ubuntu)  Gid: (1000/ubuntu)
6 Access: 2023-12-14 21:24:40.479999513 +0800
7 Modify: 2023-12-02 23:54:03.232637849 +0800
8 Change: 2023-12-02 23:54:03.232637849 +0800
9 Birth: -

```

**File** 表明它的文件名为 oskernel2022-npucore。

**Size** 表明它的字节大小为 4096 字节。

**Blocks** 表明它占据 8 个块 (Block) 来存储。在文件系统中，文件的数据以块为单位进行存储。在 IO Block 可以看出，在 Linux 操作系统中的 Ext4 文件系统的每个块的大小为 4096 字节。

**directory** 表明 oskernel2022-npucore 是一个目录，该位置为 **regular file** 时表明这个文件是一个常规文件。事实上，其他类型的文件也可以通过文件名来进行访问。

当文件是一个特殊文件（如块设备文件或者字符设备文件）的时候，Device 将指出该特殊文件的 major/minor ID。

**Inode** 表示文件的底层编号。在文件系统的底层实现中，并不是直接通过文件名来索引文件，而是首先需要将文件名转化为文件的底层编号，再根据这个编号去索引文件。目前我们无需关心这一信息。

**Links** 给出文件的硬链接数。同一个文件系统中如果两个文件（目录也是文件）具有相同的 inode 号码，那么就称它们是“硬链接”关系。这样 links 的值其实是一个文件的不同文件名的数量。

**Uid** 给出该文件的所属的用户 ID，**Gid** 给出该文件所属的用户组 ID。**Access** 的其中一种表示是一个长度为 10 的字符串（这里是 drwxrwxr-x），其中第 1 位给出该文件的类型，这个文件是一个目录，因此这第 1 位为 d。后面的 9 位可以分为三组，分别表示该文件的所有者/在该文件所属的用户组内的其他用户以及剩下的所有用户能够读取/写入/将该文件作为一个可执行文件来执行。

**Access/Modify** 分别给出该文件的最近一次访问/最近一次修改时间。

实际上目录也可以看作一种文件，它也有属于自己的底层编号，它的内容中保存着若干 **目录项** (Dirent, Directory Entry)，可以看成一组映射，根据它下面的文件名或子目录名能够查到文件和子目录在文件系统中的底层编号，即 Inode 编号。但是与常规文件不同的是，用户无法 **直接** 修改目录的内容，只能通过创建/删除它下面的文件或子目录才能间接做到这一点。

## 7.5.2 路径

有了目录之后，我们就可以将所有的文件和目录组织为一种被称为 **目录树** (Directory Tree) 的有根树结构（不考虑软链接）。树中的每个节点都是一个文件或目录，一个目录下面的所有文件和子目录都是它的孩子。可以看出所有的文件都是目录树的叶子节点。目录树的根节点也是一个目录，它被称为 **根目录** (Root Directory)。目录树中的每个目录和文件都可以用它的 **绝对路径** (Absolute Path) 来进行索引和定位。绝对路径是目录树上的根节点到待索引的目录和文件所在的节点之间自上而下的路径。此路径上的所有节点（文件或目录）两两之间加上路径分隔符拼接就可得到绝对路径名。例如，在 Linux 上，根目录的绝对路径是 “/”，路径分隔符也是 “/”，因此：

一般情况下，绝对路径都很长，用起来颇为不便。而且，在日常使用中，我们通常固定在一个工作目录下而不会频繁切换目录。因此更为常用的是 **相对路径** (Relative Path) 而非绝对路径。每个进程都会记录自己当前所在的工作目录 (Current Working Directory, CWD)，当它在索引文件或目录的时候，如果传给它的路径并未以 / 开头，则会被内核认为是一个相对于进程当前工作目录的相对路径。这个路径会被拼接在进程当前路径的后面组成一个绝对路径，实际索引的是这个绝对路径对应的文件或目录。其中，./ 表示当前目录，而../ 表示当前目录的父目录，这在通过相对路径进行索引的时候非常实用。在使用终端的时候，执行 `pwd` 命令可以打印终端进程当前所在的目录，而通过 `cd` 可以切换终端进程的工作目录。

一旦引入目录之后，我们就不再单纯的通过文件名来索引文件，而是通过路径（绝对或相对）进行索引。在文件系统的底层实现中，也是对应的先将路径转化为一个文件或目录的底层编号，然后再通过这个编号具体索引文件或目录。将路径转化为底层编号的过程是逐级进行的，对于绝对路径的情况，需要从根目录出发，每次根据当前目录底层编号获取到它的内容，根据下一级子目录的目录名查到该子目录的底层编号，然后从该子目录继续向下遍历，依此类推。在这个过程中目录的权限控制位将会起到保护作用，阻止无权限用户进行访问。

### 7.5.3 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。

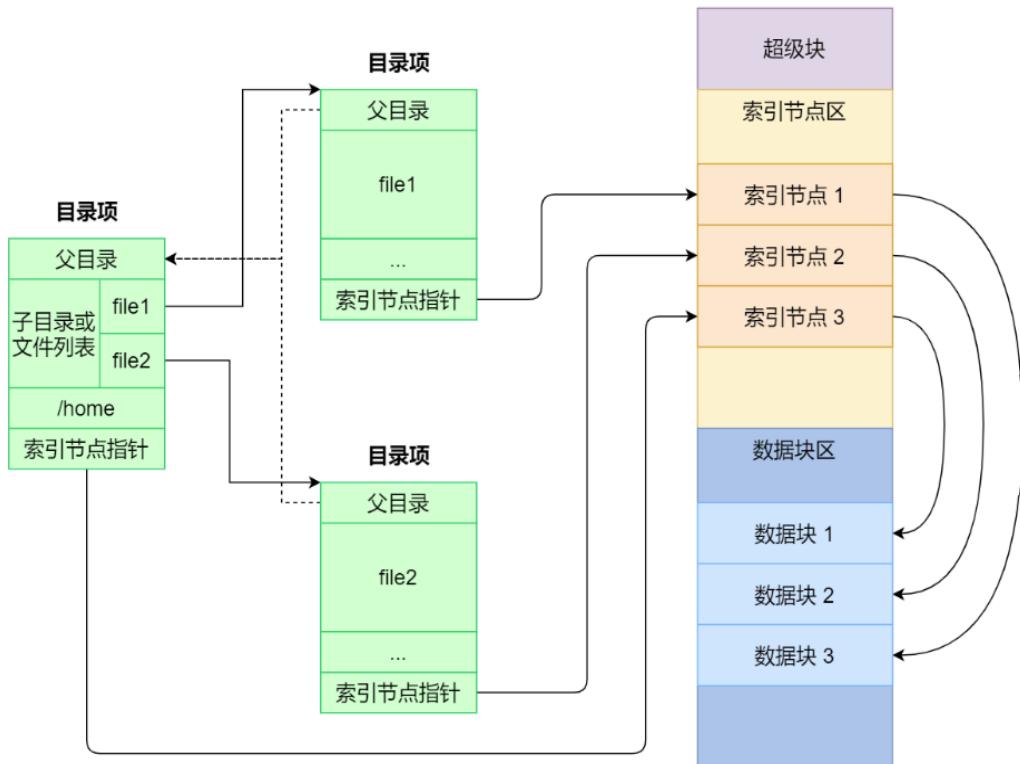
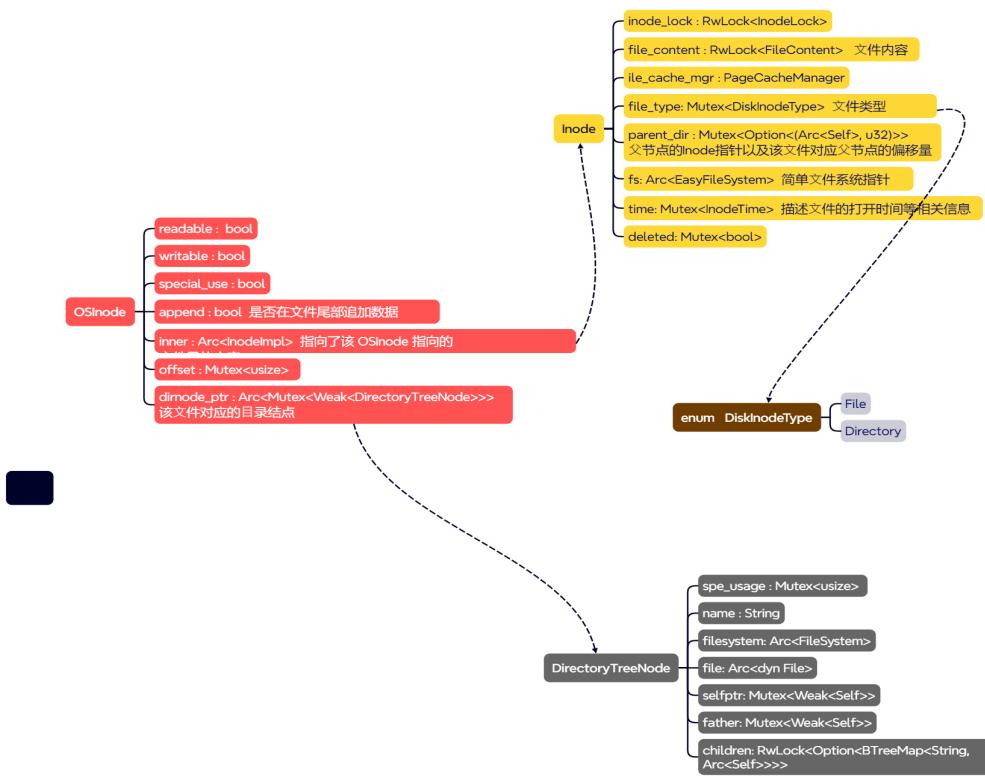


图 7-3 目录树的有根树数据结构



Presented with xml id

图 7-4 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。

```

1 // os/src/fs/fat32/directory_tree.rs
2 pub struct DirectoryTreeNode {
3     /// If this is a directory
4     /// 1. cwd
5     /// 2. mount point
6     /// 3. root node
7     /// If this is a file
8     /// 1. executed by some processes
9     /// This parameter will add 1 when
10    opening
11    ///
12    pub nlink_count : Mutex<u32>,
13    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
14    是否在路径缓存中，这样使大量文件操作效率更高，同时
15    NPUcore也对一些默认的路径进行了路径的转化。
16    文件描述符层
17    spe_usage: Mutex<usize>,
18    name: String,
19    filesystem: Arc<FileSystem>,
20    file: Arc<dyn File>,
21    selfptr: Mutex<Weak<Self>>,
22    father: Mutex<Weak<Self>>,
23    children: RwLock<Option<BTreesMap<String,
24    Arc<Self>>>,
25 }

```

```

1 // src/os/fs/fat32/inode.rs
2 pub struct OSInode {
3     //nlink_count : Mutex<usize>,
4     readable: bool,
5     writable: bool,
6     special_use: bool,
7     append: bool,
8     inner: Arc<InodeImpl>,
9     offset: Mutex<usize>,
10    dirnode_ptr:
11    Arc<Mutex<Weak<DirectoryTreeNode>>>,
12 }

```

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：

```

1 static ref DIRECTORY_VEC: Mutex<
2     (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：

```

1 pub static ref ROOT: Arc<DirectoryTreeNode> = {
2     let inode = DirectoryTreeNode::new(
3         "" .to_string(),

```

```

4     Arc::new(FileSystem::new(FS::Fat32)),
5     OSInode::new(InodeImpl::root_inode(&FILE_SYSTEM)),
6     Weak::new()
7   );
8   inode.add_special_use();
9   inode
10 };

```

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。

## 7.6 文件描述符层

一个进程可以访问的多个文件，所以在操作系统中需要有一个管理进程访问的多个文件的结构，这就是 **文件描述符表** (File Descriptor Table)，其中的每个 **文件描述符** (File Descriptor) 代表了一个特定读写属性的 I/O 资源。

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：

```

1 fn readable(&self) -> bool {
2     self.readable
3 }
4 fn writable(&self) -> bool {
5     self.writable
6 }
7 fn read(\&self, offset: Option<\&mut usize>, buffer: \&mut [u8]) -> usize {
8     match offset {
9         Some(offset) => {
10             let len =
11                 self.inner.read_at_block_cache(*offset,
12                     buffer);
13             *offset += len;
14             len
15         }
16         None => {
17             let mut offset =
18                 self.offset.lock();
19             let len =
20                 self.inner.read_at_block_cache(*offset,
21                     buffer);
22         }
23     }
24 }

```

```

22     *offset += len;
23     len
24   }
25 }
26 }
27
28 fn write(\&self, offset: Option<\&mut
29 usize>, buffer: \&[u8]) -> usize {
30   match offset {
31     Some(offset) => {
32       let len =
33         self.inner.write_at_block_cache(*offset,
34         buffer);
35       *offset += len;
36       len
37     }
38     None => {
39       let mut offset =
40         self.offset.lock();
41       let inode_lock =
42         self.inner.write();
43       if self.append {
44         *offset =
45           self.inner.get_file_size_wlock(\&inode_lock)
46           as usize;
47       }
48       let len = self
49         .inner
50         .write_at_block_cache_lock(\&inode_lock,
51         *offset, buffer);
52       *offset += len;
53       len
54     }
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。

## 7.7 文件相关系统调用

npucore 实现了许多 POSIX 规定的系统调用，本章主要介绍几个与文件相关的系统调用，包括文件的创建、打开、关闭、读写、删除等。

### 7.7.1 write

对于文件的读是最常使用的系统调用之一，包括输入输出都是基于终端来实现的（终端本身也可以当作一个文件）。下面我们来介绍 npucore 中 write 的实现。

write 系统调用的函数签名如下：

```
1 int write(int fd, const void *buf, size_t count);
```

write 系统调用将 buf 中的 count 个字节写入文件描述符 fd 所指向的文件中。成功时返回写入的字节数，失败时返回-1。

NPUCore 中的代码实现如下：

```
1 pub fn sys_write(fd: usize, buf: usize, count: usize) -> isize {
2     let task = current_task().unwrap();
3     let fd_table = task.files.lock();
4     let file_descriptor = match fd_table.get_ref(fd) {
5         Ok(file_descriptor) => file_descriptor,
6         Err(errno) => return errno,
7     };
8     if !file_descriptor.writable() {
9         return EBADF;
10    }
11    let token = task.get_user_token();
12    file_descriptor.write_user(
13        None,
14        UserBuffer::new({
15            match translated_byte_buffer(token, buf as *const u8, count) {
16                Ok(buffer) => buffer,
17                Err(errno) => return errno,
18            }
19        }),
20    ) as isize
21 }
```

其逐代码块的解释如下:`current_task().unwrap()` 获取当前任务(线程)的引用。`task.files.lock()` 获取当前任务的文件表的锁, 以确保并发访问的同步。通过文件描述符 `fd` 从文件表中获取文件描述符的引用, 使用 `fd_table.get_ref(fd)`。如果获取失败, 则返回相应的错误号。检查文件是否可写, 如果不可写则返回 `EBADF` 错误号。获取当前任务的用户令牌, 通常用于在用户空间和内核空间之间传递数据。将用户空间缓冲区的数据写入文件描述符。这里使用了 `translated_byte_buffer` 函数来将用户空间的字节缓冲区翻译成内核地址空间, 确保正确的内存访问。最后, 将写入的字节数作为 `isize` 类型返回。

如果从代码的逻辑层次来分析, 其调用过程如下:

`sys_writet`。获取当前进程的引用, 然后获取文件描述符表, 然后根据该表与参数 `fd` 获取对相应文件描述符的引用。这之后检查文件是否可写并获取 `user_token`, 这些操作都完成之后, 以 `buffer` (缓冲区) 与 `count` (缓冲区长度) 两个参数转换为 `buffer`, 传入 `FileDescriptor::write_user` 中。

`FileDescriptor::write_user`。直接调用 `<dyn File>` 的 `write_user` 函数

`OSInode::write_user`。如果参数中 `offset` 为 `none` (比如由 `write` 系统调用时就为 `none`), 则先获取文件偏移的锁、`inode` (`OSInode` 中 `inner`) 的锁, 然后看是否设置了追加标志, 如果是, 则将文件偏移放到最后 (即从文件末尾开始写入)。然后调用 `OSInode.inner` 的 `write_at_block_cache_lock` (即 `Inode::write_at_block_cache_lock`)。完成之后, 修改记录的文件偏移, 返回写入的字节数。

`Inode::write_at_block_cache_lock`。内部有一个 `loop`, 计算块的末尾、写入并更新大小、移动到下一个块。到这里, 写入过程就结束了。然而此时数据仍然留在内存中, 还没有被持久化。被持久化的过程是另一个独立的过程, 将在 IO 设备章节详细介绍。在这里做一个简要讲解: 当满足某个条件时 (比如缓冲区接近满溢), 应当调用 `OSInode` 的 `oom` 函数, 将满足条件的缓冲块落盘。

### 7.7.2 read

`read` 函数用于从文件描述符中读取数据, 并将数据存储到缓冲区中。这个系统调用在应用程序与操作系统内核之间搭建了一座桥梁, 允许应用程序从指定的文件描述符中读取数据。以下是 `read` 系统调用的工作流程。

**调用请求:** 应用程序通过执行 `read` 系统调用, 并传入相应的文件描述符、缓冲区地址和要读取的字节数, 发起读取请求。

**参数验证:** 内核首先验证参数的有效性, 包括文件描述符的合法性和缓冲区地址的可访问性。

**读取操作:** 内核根据文件描述符找到相应的文件或数据流, 并从中读取指定数量的字节。这个过程可能涉及文件系统的访问、网络操作或设备交互。

**数据传输:** 读取的数据被存储在内核缓冲区中, 然后被复制到用户提供的缓冲区。这个过程涉及从内核空间到用户空间的数据传输。

返回结果：read 调用最终返回读取的字节数。如果到达文件末尾，则返回 0。如果发生错误，则返回一个负值，并设置相应的错误码。

阻塞与非阻塞模式：read 操作可以在阻塞或非阻塞模式下进行。在阻塞模式下，如果没有可用数据，read 调用会阻塞调用进程，直到有数据可读。在非阻塞模式下，如果没有数据可读，read 会立即返回，通常是一个错误码。

接下来是在 NPUCore 中 read 的具体实现：

```

1 pub fn sys_read(fd: usize, buf: &mut [u8], count: usize) -> isize {
2     let task = current_task().unwrap();
3     let fd_table = task.files.lock();
4     let file_descriptor = match fd_table.get_ref(fd) {
5         Ok(file_descriptor) => file_descriptor,
6         Err(errno) => return errno,
7     };
8     // fd is not open for reading
9     if !file_descriptor.readable() {
10         return EBADF;
11     }
12     let token = task.get_user_token();
13     file_descriptor.read_user(
14         None,
15         UserBuffer::new({
16             match translated_byte_buffer(token, buf as *const u8, count) {
17                 Ok(buffer) => buffer,
18                 Err(errno) => return errno,
19             }
20         }),
21     ) as isize
22 }
```

获取当前任务：函数首先获取当前正在执行的任务或进程的上下文。

获取文件描述符表：锁定当前任务的文件描述符表，并尝试从中获取与传入的文件描述符 fd 相对应的文件描述符对象。

错误处理：如果指定的文件描述符不存在或其他错误发生，函数将返回相应的错误码。

检查读取权限：确认获取到的文件描述符是否具有读取权限。如果没有，返回 EBADF 错误码，表示文件描述符不合法或不支持读取操作。

获取用户令牌：获取与当前任务关联的用户令牌，用于后续的权限验证和内存安全检查。

处理用户缓冲区：使用 `translated_byte_buffer` 函数将用户空间的缓冲区地址和长度转换为内核可以操作的缓冲区对象 `UserBuffer`。这一步骤涉及内存地址转换和错误检查。

执行读取操作：调用文件描述符的 `read_user` 方法，从文件或数据流中读取数据到用户提供的缓冲区中。

返回结果：返回从文件描述符中实际读取的字节数。如果读取过程中发生错误，返

回相应的错误码。

### 7.7.3 open

首先，我们分析一下 NPUcore 文件系统提供给应用的接口，即用户态的 sys\_open 系统调用。

在读写一个常规文件之前，应用首先需要通过内核提供的 sys\_open 系统调用让该文件在进程的文件描述符表中占一项，并得到操作系统的返回值——文件描述符，即文件关联的表项在文件描述表中的索引值：

```

1 // user/src/syscall.rs
2 /// syscall ID: 56
3 pub fn sys_open(path: &str, flags: u32) -> isize {
4     syscall(SYS_OPEN, [path.as_ptr() as usize, flags as usize, 0])
5 }
```

如上所示 sys\_open 的功能为打开一个常规文件，并返回可以访问它的文件描述符。参数 path 描述要打开的文件的文件名（简单起见，文件系统不需要支持目录，所有的文件都放在根目录 / 下），flags 描述打开文件的标志，具体含义下面给出。至于返回值，如果出现了错误则返回 -1，否则返回打开常规文件的文件描述符。可能的错误原因是：文件不存在。

然后我们讲解一下 flags，目前我们的内核支持以下几种标志（多种不同标志可能共存）：

- 如果 flags 为 0，则表示以只读模式 RONLY 打开；
- 如果 flags 第 0 位被设置 (0x001)，表示以只写模式 WRONLY 打开；
- 如果 flags 第 1 位被设置 (0x002)，表示既可读又可写 RDWR；
- 如果 flags 第 9 位被设置 (0x200)，表示允许创建文件 CREATE，在找不到该文件的时候应创建文件；如果该文件已经存在则应该将该文件的大小归零；
- 如果 flags 第 10 位被设置 (0x400)，则在打开文件的时候应该清空文件的内容并将该文件的大小归零，也即 TRUNC。

注意 flags 里面的权限设置只能控制进程对本次打开的文件的访问。一般情况下，在打开文件的时候首先需要经过文件系统的权限检查，比如一个文件自身不允许写入，那么进程自然也就不能以 WRONLY 或 RDWR 标志打开文件。但在我们简化版的文件系统中文件不进行权限设置，这一步就可以绕过。

在用户库 user\_lib 中，我们将该系统调用封装为 open 接口：

```

1 // user/src/lib.rs
2
3 bitflags! {
4     pub struct OpenFlags: u32 {
5         const RONLY = 0;
6         const WRONLY = 1 << 0;
7         const RDWR = 1 << 1;
8         const CREATE = 1 << 9;
9     }
10 }
```

```

9         const TRUNC = 1 << 10;
10    }
11 }
12
13 pub fn open(path: &str, flags: OpenFlags) -> isize {
14     sys_open(path, flags.bits)
15 }
```

如上，借助 bitflags! 宏我们将一个 u32 的 flags 包装为一个 OpenFlags 结构体更易使用，它的 bits 字段可以将自身转回 u32，它也会被传给 sys\_open。

```

1 // user/src/syscall.rs
2 /// syscall ID: 56
3 pub fn sys_open(path: &str, flags: u32) -> isize {
4     syscall(SYSCALL_OPEN, [path.as_ptr() as usize, flags as usize, 0])
5 }
```

如上，sys\_open 传给内核的参数只有待打开文件的文件名字符串的起始地址（和之前一样，我们需要保证该字符串以 \0 结尾）还有标志位。由于每个通用寄存器为 64 位，我们需要先将 u32 的 flags 转换为 usize。

接下来，我们分析一下 sys\_open 在内核中的实现。

#### 7.7.4 close

首先，我们分析一下 NPUcore 文件系统提供给应用的接口，即用户态的 sys\_close 系统调用。

在打开文件，对文件完成了读写操作后，还需要关闭文件，这样才让进程释放被这个文件占用的内核资源。close 的调用参数是文件描述符，当文件被关闭后，该文件在内核中的资源会被释放，文件描述符会被回收。这样，进程就不能继续使用该文件描述符进行文件读写了。

```

1 // user/src/lib.rs
2 pub fn close(fd: usize) -> isize { sys_close(fd) }
3
4 // user/src/syscall.rs
5 const SYSCALL_CLOSE: usize = 57;
6
7 pub fn sys_close(fd: usize) -> isize {
8     syscall(SYSCALL_CLOSE, [fd, 0, 0])
9 }
```

如上，sys\_close 的功能是当前进程关闭一个文件，参数 fd 表示要关闭的文件的文件描述符，如果成功关闭则返回 0，否则返回 -1。可能的出错原因：传入的文件描述符并不对应一个打开的文件。

接下来，我们分析一下 sys\_open 在内核中的实现。

关闭文件的系统调用 sys\_close 实现非常简单，我们只需将进程控制块中的文件描述符表对应的一项改为 None 代表它已经空闲即可，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。

```

1  pub fn sys_close(fd: usize) -> isize {
2      info!("[sys_close] fd: {}", fd);
3      let task = current_task().unwrap();
4      let mut fd_table = task.files.lock();
5      match fd_table.remove(fd) {
6          Ok(_) => SUCCESS,
7          Err(errno) => errno,
8      }
9  }
```

### 7.7.5 fstat, fstatat

NPUcore 为用户提供了 fstat 和 fstatat 两个系统调用用于获取文件信息。相比之下，fstatat 的功能更加完善和丰富，实现也更加复杂，这里先从 fstatat 开始分析。

fstatat 和 fstat 的函数原型如下：

代码片段 7.9 fstat,fstatat

```

1  pub fn sys_fstatat(dirfd: usize, path: *const u8, buf: *mut u8, flags:
2      u32) -> isize {}
3  pub fn sys_fstat(fd: usize, statbuf: *mut u8) -> isize {}
```

从参数可以看出，statat 可以按照路径访问文件信息，而 fstat 则只能通过进程的文件描述符表访问。这是两者的主要区别。对于 fstat，首先进行的是获取当前进程信息以及解析路径：

代码片段 7.10 fstatat\_part1

```

1  let token = current_user_token();
2  let path = match translated_str(token, path) {
3      Ok(path) => path,
4      Err(errno) => return errno,
5  };
6  let flags = match FstatatFlags::from_bits(flags) {
7      Some(flags) => flags,
8      None => {
9          warn!("[sys_fstatat] unknown flags");
10         return EINVAL;
11     }
12 };
13
14 info!(
15 "[sys_fstatat] dirfd: {}, path: {:?}", path, flags,
16 dirfd as isize, path, flags,
17 );
18
19 let task = current_task().unwrap();
```

flags 这里只是进行了解析，但是在之后实现中没有涉及，这里也不过多叙述。在这里获取了当前运行的进程，并且将切片类型的 path 映射到当前地址空间，并且转换成 string 类型。

## 代码片段 7.11 fstatat\_part2

```

1 let file_descriptor = match dirfd {
2     AT_FDCWD => task.fs.lock().working_inode.as_ref().clone(),
3     fd => {
4         let fd_table = task.files.lock();
5         match fd_table.get_ref(fd) {
6             Ok(file_descriptor) => file_descriptor.clone(),
7             Err(errno) => return errno,
8         }
9     }
10 };
11
12 match file_descriptor.open(&path, OpenFlags::O_RDONLY, false) {
13     Ok(file_descriptor) => {
14         copy_to_user(token, &file_descriptor.get_stat(), buf as *mut Stat);
15         SUCCESS
16     }
17     Err(errno) => errno,
18 }

```

这里先获取当前目录的文件描述符，之后调用 open 函数通过路径打开文件。如果文件打开成功，则调用 copy\_to\_user 函数将文件信息内容拷贝到 buf 中，之后函数返回。

对于文件信息的内容，NPUCore 中使用结构体 Stat 表示，具体代码如下：

## 代码片段 7.12 Stat

```

1 #[derive(Clone, Copy, Debug)]
2 #[repr(C)]
3 /// Store the file attributes from a supported file.
4 pub struct Stat {
5     /// ID of device containing file
6     st_dev: u64,
7     /// Inode number
8     st_ino: u64,
9     /// File type and mode
10    st_mode: u32,
11    /// Number of hard links
12    st_nlink: u32,
13    /// User ID of the file's owner.
14    st_uid: u32,
15    /// Group ID of the file's group.
16    st_gid: u32,
17    /// Device ID (if special file)
18    st_rdev: u64,
19    __pad: u64,
20    /// Size of file, in bytes.
21    st_size: i64,
22    /// Optimal block size for I/O.
23    st_blksize: u32,
24    __pad2: i32,
25    /// Number 512-byte blocks allocated.
26    st_blocks: u64,
27    /// Backward compatibility. Used for time of last access.
28    st_atime: TimeSpec,
29    /// Time of last modification.
30    st_mtime: TimeSpec,
31    /// Time of last status change.

```

```

32     st_ctime: TimeSpec,
33     __unused: u64,
34 }

```

结构体每一个字段的含义已经给出。

对于系统调用 fstat，其功能相较于 fstatat 更为简单。仅仅只有通过进程的文件描述符表获取文件描述符和相关信息的拷贝。具体代码如下：

代码片段 7.13 fstat

```

1 pub fn sys_fstat(fd: usize, statbuf: *mut u8) -> isize {
2     let task = current_task().unwrap();
3     let token = task.get_user_token();
4
5     info!("[sys_fstat] fd: {}", fd);
6     let file_descriptor = match fd {
7         AT_FDCWD => task.fs.lock().working_inode.as_ref().clone(),
8         fd => {
9             let fd_table = task.files.lock();
10            match fd_table.get_ref(fd) {
11                Ok(file_descriptor) => file_descriptor.clone(),
12                Err(errno) => return errno,
13            }
14        }
15    };
16    copy_to_user(token, &file_descriptor.get_stat(), statbuf as *mut Stat);
17    SUCCESS
18 }

```

由于其本身传入参数的限制，fstat 函数只能获取当前进程的文件信息。

## 7.8 虚拟文件系统及接口

### 7.8.1 VFS 简介

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。

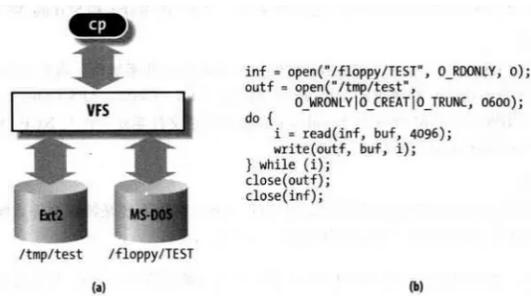


图 7-5 VFS 在文件复制

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。

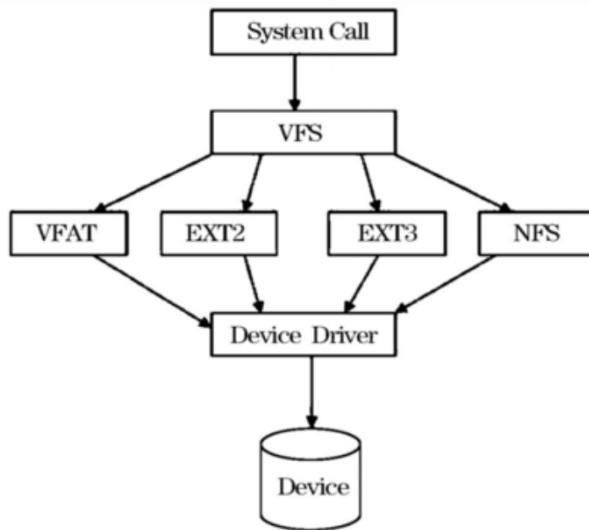


图 7-6 VFS 在 OS 结构中

### 7.8.2 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 7.14 filesystem

```

1 pub enum FS {
2     Null,
3     Fat32,
4 }
5
6 pub struct FileSystem {
7     pub fs_id: usize,
8     pub fs_type: FS,
9 }
```

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 7.15 DirectoryTreeNode

```

1 pub struct DirectoryTreeNode {
2     /// If this is a directory
3     /// 1. cwd
4     /// 2. mount point
5     /// 3. root node
6     /// If this is a file
7     /// 1. executed by some processes
8     /// This parameter will add 1 when opening
9     spe_usage: Mutex<usize>,
10    name: String,
11    filesystem: Arc<FileSystem>,
12    file: Arc<dyn File>,
13    selfptr: Mutex<Weak<Self>>,
14    father: Mutex<Weak<Self>>,
15    children: RwLock<Option<BTrieMap<String, Arc<Self>>>>,
16 }
```

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 7.16 DirectoryTreeNode

```

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

从表面上看这些和多文件系统共存没有关系，但是以下代码完成了该项功能：

代码片段 7.17 file\_trait

```

1 pub trait File: DowncastSync {
2     fn deep_clone(&self) -> Arc<dyn File>;
3     fn readable(&self) -> bool;
4     fn writable(&self) -> bool;
```

```

5   fn read(&self, offset: Option<&mut usize>, buf: &mut [u8]) -> usize
6       ;
7   fn write(&self, offset: Option<&mut usize>, buf: &[u8]) -> usize;
8   fn r_ready(&self) -> bool;
9   fn w_ready(&self) -> bool;
10  fn read_user(&self, offset: Option<usize>, buf: UserBuffer) ->
11      usize;
12  fn write_user(&self, offset: Option<usize>, buf: UserBuffer) ->
13      usize;
14  fn get_size(&self) -> usize;
15  fn get_stat(&self) -> Stat;
16  fn get_file_type(&self) -> DiskInodeType;
17  fn is_dir(&self) -> bool {
18      self.get_file_type() == DiskInodeType::Directory
19  }
20  fn is_file(&self) -> bool {
21      self.get_file_type() == DiskInodeType::File
22  }
23  fn info_dirtree_node(&self, dirnode_ptr: Weak<DirectoryTreeNode>);
24  fn get_dirtree_node(&self) -> Option<Arc<DirectoryTreeNode>>;
25  /// open
26  fn open(&self, flags: OpenFlags, special_use: bool) -> Arc<dyn File
27      >;
28  fn open_subfile(&self) -> Result<Vec<(String, Arc<dyn File>)>,
29      isize>;
30  /// create
31  fn create(&self, name: &str, file_type: DiskInodeType) -> Result<
32      Arc<dyn File>, isize>;
33  fn link_child(&self, name: &str, child: &Self) -> Result<(), isize>
34  where
35      Self: Sized;
36  /// delete(unlink)
37  fn unlink(&self, delete: bool) -> Result<(), isize>;
38  /// dirent
39  fn get_dirent(&self, count: usize) -> Vec<Dirent>;
40  /// offset
41  fn get_offset(&self) -> usize {
42      self.lseek(0, SeekWhence::SEEK_CUR).unwrap()
43  }
44  fn lseek(&self, offset: isize, whence: SeekWhence) -> Result<usize,
45      isize>;
46  /// size
47  fn modify_size(&self, diff: isize) -> Result<(), isize>;
48  fn truncate_size(&self, new_size: usize) -> Result<(), isize>;
49  // time
50  fn set_timestamp(&self, ctime: Option<usize>, atime: Option<usize>,
51      mtime: Option<usize>);
52  /// cache
53  fn get_single_cache(&self, offset: usize) -> Result<Arc<Mutex<
      PageCache>>, ()>;
54  fn get_all_caches(&self) -> Result<Vec<Arc<Mutex<PageCache>>>,
55      ()>;
56  /// memory related
57  fn oom(&self) -> usize;
58  /// poll, select related
59  fn hang_up(&self) -> bool;
60  /// ioctl
61  fn ioctl(&self, _cmd: u32, _argp: usize) -> isize {
62      ENOTTY
63  }

```

```

54 }
55 // fn fcntl
56 fn fcntl(&self, cmd: u32, arg: u32) -> isize;
57 }
```

这在 rust 语法中属于特性，可以实现泛型。无论文件系统的文件结构如何定义，只要实现了这些特性，虚拟文件系统就可以识别并进行操作，很好地完成了这一点。这里不得不感叹 rust 语法的巧妙所在。

### 7.8.3 虚拟文件系统提供的接口

- `sys_openat`:

```

1 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`:

```

1 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`:

```

1 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`:

```

1 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`:

```

1 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`:

```
1 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`:

```
1 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`:

```
1 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。

## 7.9 文件共享与 PIPE

文件共享和管道是操作系统中关键的概念，它们为进程之间的通信和协同提供了有效的手段。在 npucore 中，这些机制是构建多任务、多进程应用程序的基础。本章将深入探讨文件共享和管道的原理、应用和在 npucore 中的具体实现。

文件共享是多个进程之间共享数据的一种机制。我们将深入研究硬链接和软链接，它们分别提供了对同一文件的不同访问方式。

硬链接是文件系统中的重要概念，允许一个文件存在于多个位置，节省存储空间并提供数据的共享。与硬链接相比，软链接提供了更灵活的方式来共享文件。

文件描述符是在进程和文件之间建立联系的关键。文件共享不仅仅是为了在同一进程内部使用，还可以用于不同进程之间的通信。在实际的运用场景中，我们常常需要在父子进程之间通信，POSIX 提供了一种特殊的文件描述符，可以用于实现这种通信，叫做管道。管道是一种强大的进程通信工具，使得不同进程之间能够直接传递数据。我们将深入研究管道的工作原理，以及在 xv6 中如何创建、使用管道。在了解了管道的基本概念后，我们将深入研究在 npucore 中如何创建和使用管道。这涉及到管道的创建、连接和断开的具体步骤。管道不仅仅是用于数据传输，还促进了进程之间的通信。我们将说明在 npucore 中如何利用管道实现进程间通信，以及这种通信的应用场景。通过深入研究这些主题，读者将能够全面了解在 npucore 操作系统中如何利用文件共享和管道机制来构建灵活、高效的多任务应用程序。

### 7.9.1 软链接与硬链接

连接分为两种类型：硬链接（hard link）和软链接（symbolic link）。本文将详细介绍这两种类型的连接的特点、用法和区别。硬链接是指在同一个文件系统中，将一个文件名关联到一个已经存在的文件上，使得该文件名也可以访问该文件。硬链接与原文件共享 inode，即它们有相同的 inode 号和相同的 device 号。因此，对于硬链接和原文件来说，它们的访问权限、所有者、大小等属性都是相同的。软链接（也称符号链接）是指在不同的文件系统之间，将一个文件名关联到另一个文件上，使得该文件名也可以访问该文件。软链接与原文件不共享 inode，它们有不同的 inode 号和 device 号。因此，对于软链接和原文件来说，它们的访问权限、所有者、大小等属性可能不同。

#### (1) 硬链接

硬链接是指一个文件系统中的多个文件名指向同一个数据块 (inode) 的情况。也就是说，硬链接是同一个文件的不同别名，它们共享相同的内容、属性和权限。硬链接只能在同一个分区创建，不能跨越不同的文件系统。硬链接是通过在文件系统中创建一个新的目录项来实现的，这个目录项指向同一个 inode，即原始文件的 inode。在文件系统中，每个文件和目录都有一个 inode，它包含了文件的元数据信息和数据块的位置信息。

当创建一个硬链接时，文件系统会在相应的目录中创建一个新的目录项，这个目录项与原始文件的 inode 相关联，因此它们实际上是同一个文件，只是有不同的文件名和目录路径。

由于硬链接与原始文件共享相同的 inode，它们实际上是同一个文件，因此它们具有相同的权限、所有者、所属组、创建时间、修改时间等。这意味着，如果您修改硬链接文件的内容，那么原始文件的内容也会被修改，因为它们指向相同的 inode。

当删除一个硬链接时，它只是将相应的目录项从文件系统中删除，但是原始文件的 inode 仍然存在，并且只有当所有的硬链接都被删除时，该 inode 才会被释放，文件才会被删除。

总之，硬链接的实现原理是通过在文件系统中创建一个新的目录项，并将其与原始文件的 inode 号和文件内容进行关联，从而实现多个文件名指向同一个文件的效果。

需要注意的是，硬链接只能在同一个文件系统中创建，因为它们都指向相同的 inode。如果您尝试在不同的文件系统中创建硬链接，那么会创建一个新的文件副本，而不是一个硬链接。

以下代码将具体讲解在 NPUCore 中 Link 的实现方式：

```

1 pub fn sys_linkat(
2     old_dirfd: usize,
3     old_path: *const u8,
4     new_dirfd: usize,
5     new_path: *const u8,
6     flags: u32,
7 ) -> isize {
8     let task = current_task().unwrap();

```

```

9  let token = task.get_user_token();
10 let old_path = match translated_str(token, old_path) {
11     Ok(path) => path,
12     Err(errno) => return errno,
13 };
14 let new_path = match translated_str(token, new_path) {
15     Ok(path) => path,
16     Err(errno) => return errno,
17 };
18 info!(
19     "[sys_linkat] old_dirfd: {}, old_path: {}, new_dirfd: {}, new_path:
20         {}, flags: {:?}", 
21     old_dirfd as isize, old_path, new_dirfd as isize, new_path, flags
22 );
23
24 let old_file_descriptor = match old_dirfd {
25     AT_FDCWD => task.fs.lock().working_inode.as_ref().clone(),
26     fd => {
27         let fd_table = task.files.lock();
28         match fd_table.get_ref(fd as usize) {
29             Ok(file_descriptor) => file_descriptor.clone(),
30             Err(errno) => return errno,
31         }
32     }
33 };
34 // let old_inode = old_file_descriptor.file.deep_clone();
35
36 let new_file_descriptor = match new_dirfd {
37     AT_FDCWD => task.fs.lock().working_inode.as_ref().clone(),
38     fd => {
39         let fd_table = task.files.lock();
40         match fd_table.get_ref(fd as usize) {
41             Ok(file_descriptor) => file_descriptor.clone(),
42             Err(errno) => return errno,
43         }
44     }
45 };
46
47 match FileDescriptor::linkat(
48     &old_file_descriptor,
49     &old_path,
50     &new_file_descriptor,
51     &new_path,
52 ) {
53     Ok(_) => SUCCESS,
54     Err(errno) => errno,
55 }
56

```

`old_dirfd` 和 `new_dirfd`: 这两个参数是文件目录的文件描述符，分别指向旧文件和新链接文件所在的目录。`old_path` 和 `new_path`: 这是指向旧文件和新链接文件路径的指针。`flags`: 这是一个控制函数行为的标志位，用于提供额外的操作信息。

获取当前任务上下文: 函数首先获取当前正在执行的任务（或进程）的上下文，这对于后续的文件操作是必要的。

用户令牌获取：为了确保操作的安全性，函数获取了与当前任务关联的用户令牌。

路径转换：将 `old_path` 和 `new_path` 从原始的指针形式转换为 Rust 语言中的字符串形式。这一步骤涉及内存安全和权限的考虑。

日志记录：为了便于调试和跟踪，函数记录了关键的操作信息。

处理文件描述符：函数根据传入的 `old_dirfd` 和 `new_dirfd` 获取相应的文件描述符。这里的处理包括判断文件描述符是否代表当前工作目录 (`AT_FDCWD`)，以及从文件描述符表中获取相应的文件描述符对象。

链接创建：最后，函数调用特定的方法 (`FileDescriptor`) 来在文件系统中创建链接。这涉及到检查文件权限、管理文件系统的 `inode`，以及确保链接创建的正确性。

错误处理和返回值：整个函数在执行过程中，会对可能出现的错误情况进行处理。成功执行后返回特定的成功值（如 0），失败则返回相应的错误码。

```

1 pub fn linkat(
2     old_fd: &Self,
3     old_path: &str,
4     new_fd: &Self,
5     new_path: &str,
6 ) -> Result<(), isize> {
7     if old_fd.file.is_file() && !old_path.starts_with('/') {
8         return Err(ENOTDIR);
9     }
10    if new_fd.file.is_file() && !new_path.starts_with('/') {
11        return Err(ENOTDIR);
12    }
13    let old_inode = old_fd.file.get_dirtree_node();
14    let old_inode = match old_inode {
15        Some(inode) => inode,
16        None => return Err(ENOENT),
17    };
18    let new_inode = new_fd.file.get_dirtree_node();
19    let new_inode = match new_inode {
20        Some(inode) => inode,
21        None => return Err(ENOENT),
22    };
23
24    let mut old_abs_path = [old_inode.get_cwd(), old_path.to_string()].join("/");
25    if old_path.starts_with('/') {
26        old_abs_path = old_path.to_string();
27    } else {
28        old_abs_path = [old_inode.get_cwd(), old_path.to_string()].join("/");
29    }
30    let mut new_abs_path = [new_inode.get_cwd(), new_path.to_string()].join("/");
31    if new_path.starts_with('/') {
32        new_abs_path = new_path.to_string();
33    } else {
34        new_abs_path = [new_inode.get_cwd(), new_path.to_string()].join("/");
35    }
36    //println!("link_path: {} -> {}", old_path, new_path);

```

```

37     //println!("link_abs: {} -> {}", old_abs_path, new_abs_path);
38     DirectoryTreeNode::linkat(&old_abs_path, &new_abs_path)
39 }
```

参数验证：首先检查旧文件描述符和新文件描述符。函数验证这两个描述符是否指向有效的文件（而非目录），同时检查提供的路径是否以斜杠（/）开头。如果任一路径不符合条件，函数将返回错误。

获取 inode 节点：对于旧文件和新文件的路径，函数获取它们各自的 inode 节点。inode 是文件系统中用于存储文件属性的数据结构。如果无法获取 inode 节点，函数返回错误。

路径处理：函数根据旧文件和新文件的 inode 节点以及提供的路径，生成完整的绝对路径。如果提供的路径已经是绝对路径（即以斜杠开头），则直接使用该路径。否则，将 inode 节点的当前工作目录与提供的路径结合生成绝对路径。

创建链接：最后，函数尝试在文件系统中创建一个从旧文件到新文件位置的链接。这是通过调用特定的方法实现的。

结果返回：如果链接创建成功，函数返回成功的结果；如果失败，则返回相应的错误码。

```

1 pub fn linkat(old_path: &str, new_path: &str) -> Result<(), isize> {
2     assert!(old_path.starts_with('/'));
3     assert!(new_path.starts_with('/'));
4
5     let mut old_comps = Self::parse_dir_path(old_path);
6     let mut new_comps = Self::parse_dir_path(new_path);
7
8     // We guarantee that last component isn't empty
9     let old_last_comp = old_comps.pop().unwrap();
10    let new_last_comp = new_comps.pop().unwrap();
11
12    let old_par_inode = match ROOT.cd_comp(&old_comps) {
13        Ok(inode) => inode,
14        Err(errno) => return Err(errno),
15    };
16    let new_par_inode = match ROOT.cd_comp(&new_comps) {
17        Ok(inode) => inode,
18        Err(errno) => return Err(errno),
19    };
20    type ChildLockType<'a> =
21        RwLockWriteGuard<'a, Option<BTreeMap<String, Arc<
22            DirectoryTreeNode>>>;
23
24    let old_lock: Arc<Mutex<ChildLockType<'_>>>;
25    let new_lock: Arc<Mutex<ChildLockType<'_>>>;
26
27    // Be careful about the lock ordering
28    if old_comps == new_comps {
29        old_lock = Arc::new(Mutex::new(old_par_inode.children.write()))
30        ;
31        new_lock = old_lock.clone();
32    } else if old_comps < new_comps {
```

```

31     old_lock = Arc::new(Mutex::new(old_par_inode.children.write()))
32         ;
33     new_lock = Arc::new(Mutex::new(new_par_inode.children.write()))
34         ;
35 } else {
36     new_lock = Arc::new(Mutex::new(new_par_inode.children.write()))
37         ;
38     old_lock = Arc::new(Mutex::new(old_par_inode.children.write()))
39         ;
40 }
41
42 let old_inode =
43     match old_par_inode.try_to_open_subfile(old_last_comp, &mut (*
44         old_lock.lock())) {
45         Ok(inode) => inode,
46         Err(errno) => return Err(errno),
47     };
48
49 if *old_inode.spe_usage.lock() > 0 {
50     return Err(EBUSY);
51 }
52
53 if old_inode.filesystem.fs_id != new_par_inode.filesystem.fs_id {
54     return Err(EXDEV);
55 }
56
57 let old_key = old_last_comp.to_string();
58 let new_key = new_last_comp.to_string();
59 match new_par_inode.try_to_open_subfile(new_last_comp, &mut (*
60         new_lock.lock())) {
61     Ok(new_inode) => {
62         if new_inode.file.is_dir() && !old_inode.file.is_dir() {
63             return Err(EISDIR);
64         }
65         if old_inode.file.is_dir() && !new_inode.file.is_dir() {
66             return Err(ENOTDIR);
67         }
68         if *new_inode.spe_usage.lock() > 0 {
69             return Err(EBUSY);
70         }
71         // delete
72         return Err(EEXIST);
73     }
74     Err(ENOENT) => {}
75     Err(errno) => return Err(errno),
76 }
77
78 match old_inode.filesystem.fs_type {
79     FS::Fat32 => {
80         let old_file = old_inode.file.downcast_ref::<OSInode>().
81             unwrap();
82         let new_par_file = new_par_inode.file.downcast_ref::<
83             OSInode>().unwrap();
84         new_par_file.link_child(new_last_comp, old_file)?;
85     }
86     FS::Null => return Err(EACCES),
87 }
88
89 let value = old_lock.lock().as_mut().unwrap().get(&old_key).unwrap

```

```

81     () .clone ();
82     let new_value = DirectoryTreeNode::new(
83         new_key.clone (),
84         new_par_inode.filesystem.clone (),
85         value.file.deep_clone (),
86         Arc::downgrade(&new_par_inode.get_arc())),
87     );
87     new_lock.lock().as_mut().unwrap().insert(new_key, new_value);
88     Ok(())
89 }
```

路径验证：首先检查 *old\_path* 和 *new\_path* 是否都是以斜杠 (/) 开头的绝对路径。这是通过断言 (assert!) 来完成的。

解析路径：函数将旧路径和新路径解析为它们的组成部分。

获取父 inode：接下来，函数查找旧路径和新路径的父 inode（索引节点），这是文件系统中存储文件元数据的结构。

锁定处理：为了线程安全，函数对父 inode 的子节点进行锁定。如果旧路径和新路径的父节点相同，它们将共享同一个锁；否则，会分别获得各自的锁。

打开子文件：函数尝试打开旧路径的子文件，并对特定的使用情况进行检查，例如是否正被使用 (EBUSY 错误)。

文件系统一致性检查：检查旧路径和新路径是否位于同一个文件系统上。如果不是，返回错误 (EXDEV)。

链接创建：根据文件系统的类型（如 FS::Fat32），执行特定的链接创建操作。如果是不支持的文件系统类型，则返回错误 (EACCES)。

处理新路径下的文件：如果新路径下已存在文件，根据文件类型（目录或普通文件）进行错误检查，并处理可能的冲突。

更新子节点：最后，更新新路径的父 inode 的子节点映射，以包含新创建的链接。

返回结果：根据上述步骤的执行情况，函数返回成功或相应的错误码。

讲完上面硬链接的实现，我们现在来讲一下如何删除一个链接。在 NPUCore 中删除一个链接，我们直接删除新链接的文件即可。下面是在 NPUCore 中具体实现：

```

1 pub fn sys_unlinkat(dirfd: usize, path: *const u8, flags: u32) -> isize {
2     let task = current_task().unwrap();
3     let token = task.get_user_token();
4     let path = match translated_str(token, path) {
5         Ok(path) => path,
6         Err(errno) => return errno,
7     };
8     let flags = match UnlinkatFlags::from_bits(flags) {
9         Some(flags) => flags,
10        None => {
11            warn!("[sys_unlinkat] unknown flags");
12            return EINVAL;
13        }
14    };
15    info!(
```

```

16     "[sys_unlinkat] dirfd: {}, path: {}, flags: {:+?}]",
17     dirfd as isize, path, flags
18 );
19
20 let file_descriptor = match dirfd {
21   AT_FDCWD => task.fs.lock().working_inode.as_ref().clone(),
22   fd => {
23     let fd_table = task.files.lock();
24     match fd_table.get_ref(fd) {
25       Ok(file_descriptor) => file_descriptor.clone(),
26       Err(errno) => return errno,
27     }
28   }
29 };
30 match file_descriptor.delete(&path, flags.contains(UnlinkatFlags::
31   AT_REMOVEDIR)) {
32   Ok(_) => SUCCESS,
33   Err(errno) => errno,
34 }
}

```

获取当前任务：函数首先获取当前执行的任务或进程的上下文。

用户令牌获取：获取与当前任务关联的用户令牌，这对于接下来的文件操作是必要的。

路径转换：将 path 参数从原始指针转换为 Rust 字符串。这一步骤涉及权限验证和内存安全。

标志位处理：解析传入的 flags 参数，将其转换为适当的标志位集合。如果标志位未知，记录警告信息并返回错误。

日志记录：记录操作的关键信息，包括目录文件描述符、路径和标志位。

处理文件描述符：根据传入的 dirfd 获取相应的文件描述符。如果 dirfd 是特殊值（如 *AT\_FDCWD*，表示当前工作目录），则使用当前任务的工作 inode；否则，从文件描述符表中获取对应的文件描述符。

执行删除操作：调用文件描述符的删除方法，根据路径和标志位执行删除操作。如果标志位包含特定值（如 *AT\_REMOVEDIR*），则执行目录的删除；否则，删除文件。

返回结果：根据删除操作的结果返回相应的状态码。成功执行返回特定的成功值（如 0），失败则返回相应的错误码。

```

1 pub fn delete(&self, path: &str, delete_directory: bool) -> Result<(),
2   isize> {
3     if self.file.is_file() && !path.starts_with('/') {
4       return Err(ENOTDIR);
5     }
6     let inode = self.file.get_dirtree_node();
7     let inode = match inode {
8       Some(inode) => inode,
9       None => return Err(ENOENT),
10    };
11    inode.delete(path, delete_directory)
}

```

路径验证：首先检查是否当前对象（self）代表的是一个文件，并且传入的路径（path）是否是绝对路径（以/开头）。如果不是绝对路径，且当前对象是文件，则返回 ENOTDIR 错误，表示路径不是一个目录。

获取 inode 节点：函数尝试获取当前文件对象的 inode 节点。inode 是文件系统中用于存储文件元数据的结构。如果无法获取 inode 节点，函数返回 ENOENT 错误，表示文件或目录不存在。

调用删除操作：通过 inode 节点，函数调用删除操作。这个操作将根据 path 参数和 delete\_directory 参数来决定具体的删除行为。如果 delete\_directory 为 true，则尝试删除目录；否则，删除文件。

返回结果：根据删除操作的结果，函数返回成功或错误状态。如果操作成功执行，返回 Ok(); 如果有错误发生，返回对应的错误码。

```

1 pub fn delete(&self, path: &str, delete_directory: bool) -> Result<(),  
2     isize> {  
3     if path.split('/').last().map_or(true, |x| x == ".") {  
4         return Err(EINVAL);  
5     }  
6  
6     let inode = if path.starts_with("//") {  
7         &**ROOT  
8     } else {  
9         &self  
10    };  
11  
12    let components = Self::parse_dir_path(path);  
13    let last_comp = *components.last().unwrap();  
14    let inode = match inode.cd_comp(&components) {  
15        Ok(inode) => inode,  
16        Err(errno) => return Err(errno),  
17    };  
18  
19    if *inode.spe_usage.lock() > 0 {  
20        return Err(EBUSY);  
21    }  
22  
23    if !delete_directory && inode.file.is_dir() {  
24        return Err(EISDIR);  
25    }  
26  
27    if delete_directory && !inode.file.is_dir() {  
28        return Err(ENOTDIR);  
29    }  
30  
31    match inode.father.lock().upgrade() {  
32        Some(par_inode) => {  
33            let mut lock = par_inode.children.write();  
34            match inode.file.unlink(true) {  
35                Ok(_) => {  
36                    let key = last_comp.to_string();  
37                    lock.as_mut().unwrap().remove(&key);  
38                }  
39            }  
40        }  
41    }  
42  
43    Ok(())

```

```

38             }
39         Err(errno) => return Err(errno),
40     }
41     None => return Err(EACCES),
42 }
43 Ok(())
44
45 }

```

**路径有效性检查：**首先检查 path 的最后一个组件是否为.，如果是，则返回 EINVAL 错误，因为. 代表当前目录，不应被删除。

**确定操作的 inode：**判断 path 是否为绝对路径（以/开头）。如果是绝对路径，使用文件系统的根 inode (ROOT)；如果不是，使用当前对象 (self) 作为操作的起点。

**解析路径：**将 path 解析为其组成部分，并找到路径的最后一个组件。

**导航到目标 inode：**使用 *cd\_comp* 方法基于解析出的路径组件导航到目标 inode。如果导航失败，返回相应的错误。

**检查特殊用途锁：**检查目标 inode 是否被特殊用途锁定（例如正在被使用中），如果是，则返回 EBUSY 错误。

**类型匹配检查：**如果 *delete\_directory* 为 false 且目标 inode 是目录，则返回 EISDIR 错误；如果 *delete\_directory* 为 true 且目标 inode 不是目录，则返回 ENOTDIR 错误。

**删除操作：**尝试删除目标 inode。这涉及获取其父 inode 的锁，并在父 inode 的子节点映射中移除目标 inode 的引用。如果删除过程中出现错误，返回相应的错误码。

**返回结果：**如果所有步骤均成功执行，返回 Ok() 表示成功；如果在任何步骤中出现错误，则返回包含错误码的 Err。

## (2) 软链接

软链接（也称为符号链接或 symlink）是指一个特殊类型的文件，它包含了另一个文件或目录的路径信息。也就是说，软链接是一个指向另一个对象的快捷方式，它不共享相同的内容，属性和权限。软链接可以跨越不同的分区和文件系统创建。

与硬链接不同（硬链接指向文件的 inode 节点），软链接并不是指向文件数据块的物理链接，而是指向文件或目录的路径名称的链接。这使得它们可以跨越不同的文件系统，并且可以链接到不存在的目标。

软链接是一个独立的文件，这个文件的类型是符号链接文件，文件的内容是另一个文件的路径。软链接文件的创建，重命名、删除操作对指向的文件没有任何改动，即不会操作指向的文件。而软链接文件的打开，读写则是在操作指向的文件。这是因为软链接文件是一个独立的文件有自己的 inode，其创建，重命名、删除操作是对其自己的 inode 操作。不会操作指向文件的 inode。

鉴于 NPUCore 的文件系统是 FAT32，并不支持实现软链接，所以并无代码展示。

FAT32 文件系统不支持软链接（也称为符号链接）的原因与其设计和历史背景有关。

**历史背景和设计目标:** FAT32 是一种较早的文件系统，起源于 1980 年代。它是为了在早期的个人计算机上提供简单、可靠的文件存储而设计的。那时，计算机资源有限，操作系统较为简单，因此 FAT32 的设计着重于简单性和兼容性，而非高级功能。

**文件系统结构:** FAT32 的结构相对基础。它使用文件分配表（File Allocation Table, FAT）来跟踪磁盘上文件的存储位置。这种结构为了简化设计，牺牲了支持更复杂特性的能力，例如文件权限、加密或软链接。

**软链接的特性:** 软链接是一种文件系统特性，它允许一个文件或目录在多个位置以不同的名字存在。这需要文件系统能够处理复杂的引用和权限管理。但 FAT32 由于其设计的简洁性，并未包含处理这种复杂关系所需的机制。

??展示了硬链接和软链接的比较，其中最大的区别就是链接的指向是否是磁盘空间。

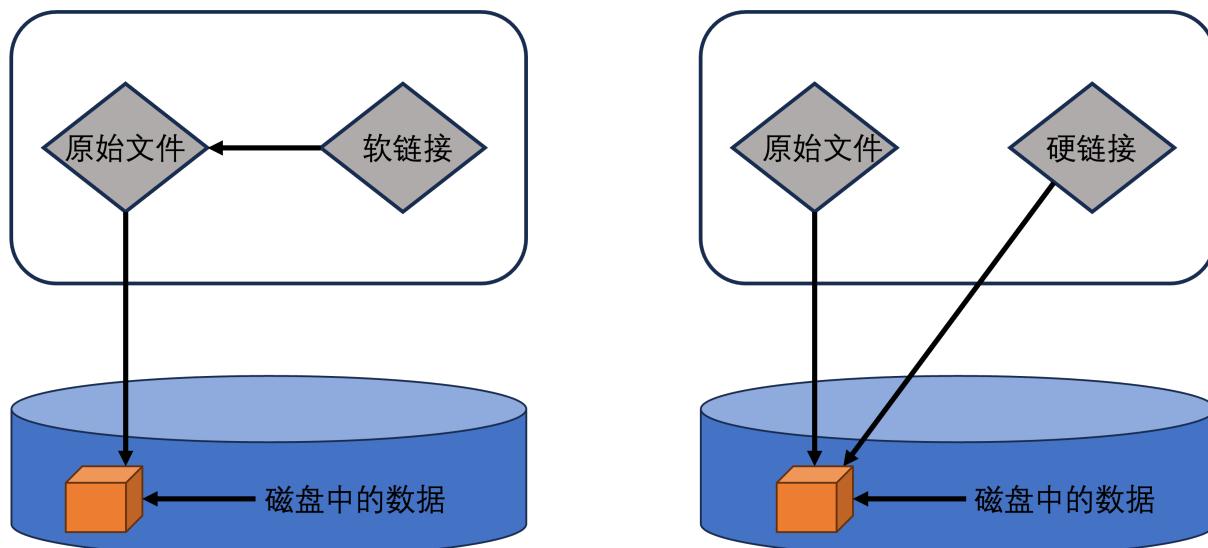


图 7-7 软链接与硬链接示意图

### 7.9.2 管道机制

管道是一种最基本的进程间通信机制，作用于有血缘关系的进程之间，完成数据传递。其本质是一个伪文件，也就是一个小的内核缓冲区，作为一对文件描述符暴露给进程，一个用于读取，一个用于写入。将数据写入管道的一端使得该数据可用于从管道的另一端读取。

与管道相关的系统调用有 `pipe`、`write`、`read` 等，功能分别为创建、写入、读取管道。后两个系统调用已在其他章节详细解释，在这里着重介绍 `pipe` 系统调用。`pipe` 系统调用的定义如下：

1 `sys_pipe2(pipefd: usize, flags: u32) -> isize`

`pipeline()` 创建一个管道，一个可用于进程间通信的单向数据通道。数组 `pipefd` 用于返回两个引用管道末端的文件描述符。`pipefd[0]` 指的是管道的读取端。`pipefd[1]` 指的

是管道的写端。写入管道写入端的数据由内核缓冲，直到从管道读取端读取。

`flags` 可以包含如下标志位：

`O_CLOEXEC`。设置两个读写文件描述符的 `close-on-exec` 标志。正常情况下，当一个进程执行一个新的程序时，新程序会继承其父进程的文件描述符。文件描述符是用于访问文件、套接字和其他 I/O 资源的抽象。`O_CLOEXEC` 标志则用于指定文件描述符在执行新程序时是否应该被关闭。当设置过此标志后，执行一个新程序时，该文件描述符会被自动关闭。也就是说，新程序将不再继承父进程中的这个文件描述符。

`O_DIRECT`。创建一个以“分组”模式进行 I/O 的管道。对管道的每次 `write` 都被视为一个单独的分组，而从管道读取数据时，`read` 系统调用将一次读取一个分组。需要注意的点有：大于 `PIPE_BUF` 字节的写操作将被拆分成多个分组。如果 `read` 指定的缓冲区大小小于下一个分组的大小，则将读取请求的字节数，并丢弃分组中的多余字节。指定缓冲区大小为 `PIPE_BUF` 足以读取可能的最大分组。不支持零长度的分组，`read` 指定缓冲区大小为零的操作是无操作的，并返回 0。

`O_NONBLOCK`。在新文件描述符所引用的打开文件描述符上设置 `O_NONBLOCK` 文件状态标志，指示文件描述符应该以非阻塞模式打开。在非阻塞模式下，文件描述符的 I/O 操作不会阻塞（即不会导致调用进程挂起等待），而是会立即返回，无论操作是否完成。这种模式在需要异步操作或需要避免长时间阻塞的场景下非常有用。

当前，在 NPUCore 中，只支持 `O_CLOEXEC` 标志位。

在内核中，`pipe` 的数据结构本质上是一个环形队列，数据从写端流入管道，从读端流出，这样就实现了进程间通信。对应 NPUCore 的代码如下。队列被初始化为空 (`head=tail=0`)，且没有写端的引 计数。

```

1  impl PipeRingBuffer {
2      fn new() -> Self {
3          // let mut vec = Vec::<u8>::with_capacity(RING_DEFAULT_BUFFER_SIZE)
4          ;
5          // unsafe {
6          //     vec.set_len(RING_DEFAULT_BUFFER_SIZE);
7          // }
8          Self {
9              arr: Box::new([0u8; RING_DEFAULT_BUFFER_SIZE]),
10             head: 0,
11             tail: 0,
12             status: RingBufferStatus::EMPTY,
13             write_end: None,
14             read_end: None,
15         }
16     }
}

```

初始化这个队列之后，实例化两个 `Pipe` 两端，并分别定义为写端与读端，也就是设置其对应的标志位：

```

1  impl Pipe {

```

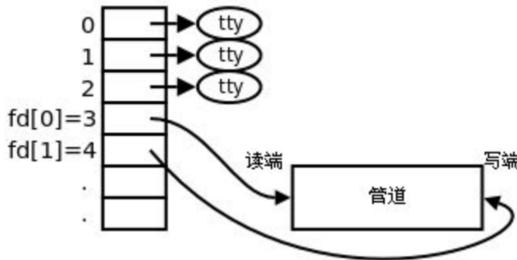
```

2  pub fn read_end_with_buffer(buffer: Arc<Mutex<PipeRingBuffer>>) -> Self
3  {
4      Self {
5          readable: true,
6          writable: false,
7          buffer,
8      }
9  }
10 pub fn write_end_with_buffer(buffer: Arc<Mutex<PipeRingBuffer>>) ->
11     Self {
12     Self {
13         readable: false,
14         writable: true,
15         buffer,
16     }
}

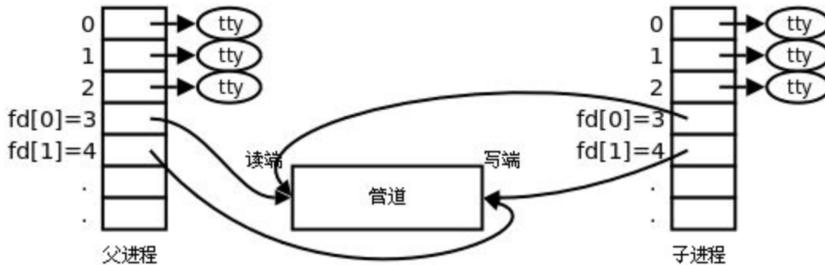
```

上述代码已经完成了 sys\_pipe2 系统调用的主体过程。

#### 1. 父进程创建管道



#### 2. 父进程 fork 出子进程



#### 3. 父进程关闭 fd[0]，子进程关闭 fd[1]

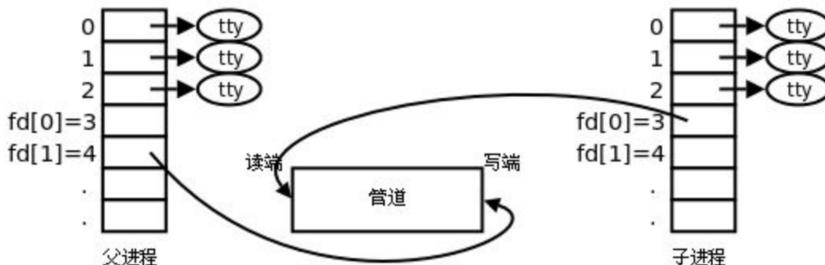


图 7-8 管道操作示意图

下面介绍 pipe 的使用过程。一般情况下，管道创建成功以后，创建该管道的进程

(父进程) 同时掌握着管道的读端和写端。要实现父子进程间通信，通常可以采用如??所示的步骤：

1. 父进程调用 pipe 系统调用创建管道，得到两个文件描述符 pipefd[0]、pipefd[1] 指向管道的读端和写端。
2. 父进程调用 fork 创建子进程，那么子进程也有两个文件描述符指向同一管道。
3. 父进程关闭管道读端，子进程关闭管道写端。父进程可以向管道中写入数据，子进程将管道中的数据读出。

使用管道时，存在以下 4 种特殊情况（假设都是阻塞 I/O 操作，没有设置 O\_NONBLOCK 标志）：

1. 如果所有指向管道写端的文件描述符都关闭了（管道写端引用计数为 0），而仍然有进程从管道的读端读数据，那么管道中剩余的数据都被读取后，再次 read 会返回 0，就像读到文件末尾一样。
2. 如果有指向管道写端的文件描述符没关闭（管道写端引用计数大于 0），而持有管道写端的进程也没有向管道中写数据，这时有进程从管道读端读数据，那么管道中剩余的数据都被读取后，再次 read 会阻塞，直到管道中有数据可读了才读取数据并返回。
3. 如果所有指向管道读端的文件描述符都关闭了（管道读端引用计数为 0），这时有进程向管道的写端 write，那么该进程会收到信号 SIGPIPE，通常会导致进程异常终止。当然也可以对 SIGPIPE 信号实施捕捉，不终止进程。具体方法信号章节详细介绍。
4. 如果有指向管道读端的文件描述符没关闭（管道读端引用计数大于 0），而持有管道读端的进程也没有从管道中读数据，这时有进程向管道写端写数据，那么在管道被写满时再次 write 会阻塞，直到管道中有空位置了才写入数据并返回。

最后介绍 pipe 的关闭过程。由于系统调 sys\_pipe 为参数 pipe 写了写端和读端两个指向进程控制块的文件描述符表的相关位置，通过系统调 sys\_close 可以关闭文件的相同的式关闭管道的端。sys\_close 将文件描述符标记为空，会使得对应的 Arc 引被销毁。写端和读端都被关闭后，buffer 的引对象，即管道的引计数减少到 0，导致管道被销毁，管道也得以关闭。

## 第 8 章 嵌入式硬件平台简介及内核运行

### 8.1 基于嵌入式硬件的操作系统开发流程

随着计算机处理器制造技术的日新月异和集成电路的蓬勃发展，小小的芯片上就可以搭载一个强大的处理器，这也就为嵌入式系统营造了一片良好的土壤，为嵌入式开发创造了一片蓝海。日常生活中，嵌入式系统无处不在，我们使用的手机，平板电脑，空调，冰箱等各种各样的物件中，都存在嵌入式系统的影子。

嵌入式系统的开发与传统的 PC 程序开发是不同的。其设计了硬件和软件的开发，是一个协同工作的统一体。

那么什么是嵌入式系统？

嵌入式系统的英文名称是 **Embedded System**。**embedded** 意为嵌入... 之中的，**system** 意为系统。嵌入式系统有一些显著的特点，它的形式多样、体积比较小，可以灵活地适应各种设备的需求，能够根据用户需求灵活地裁剪软硬件模块。

嵌入式操作系统是计算机发展的一个方向，是一个体积小型化，功能多样化的方向，而另一个方向则是体积大型化，处理能力超强的大型计算机。计算机大型化适用于那些需要进行大量数据存储和大量数据处理的环境中，例如银行需要的是计算机的处理能力和稳定性，计算机的功耗和占用空间反而是其次需要考虑的。计算机小型化适用于响应时间和数据吞吐要求不高，但对功耗和空间占用有要求的场景，像是手机，PC 电脑，电冰箱，空调等设备。

嵌入式操作系统种类繁多，按照系统硬件的核心处理器来区分，可以分为嵌入式微控制器和嵌入式微处理器。嵌入式微控制器就是传统意义上的单片机。单片机就是把一个计算机的主要功能集成到了一个芯片上，其通常包含了运算处理单元、ARM、flash 存储器，以及一些 IO 接口。嵌入式微处理器与单片机相比，其处理器的处理能力更强，目前主流的嵌入式处理器为 32 位，单片机多是 8 位和 16 位。嵌入式微处理器在一个芯片上集成了复杂的功能，其有运算单元，指令存储器，数据存储器，缓冲区，总线，一些 IO 接口。在这样一个功能完善的处理器上，就可以搭载操作系统，帮助嵌入式开发。下图为嵌入式微处理器的基本结构。

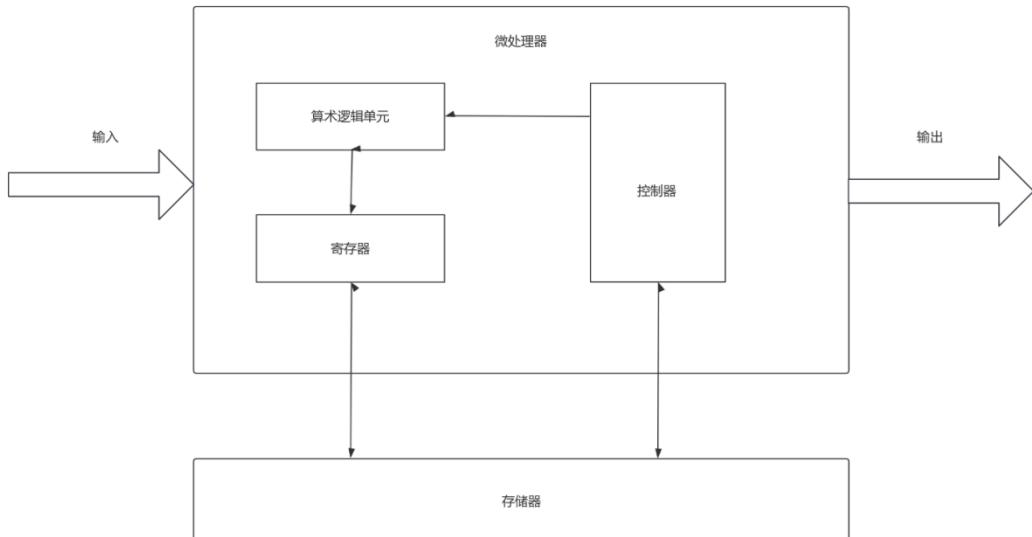


图 8-1 嵌入式微处理器的基本结构

嵌入式系统按层次来分，与传统 pc 一样，仍然分为硬件和软件层，硬件层包括嵌入式微处理器和各种外部设备，根据应用需求的不同，各种外部设备又各不相同；软件层又可以分为操作系统和应用软件两层，操作系统作为软硬件接口，负责管理系统资源，这与传统 pc 是一样的，用户仍然与应用软件打交道，看不到下层。

一般情况下，我们在一台计算机上编写了代码，便直接用该计算机上的编译器，编译后的可执行代码，程序便可以执行，像这样的情况，我们就称之为“本地编译”。然而，有的时候我们并不能本地编译，我们需要在一种平台上编译出能运行在体系结构不同的另一种平台上能够运行的程序，也就是需要“交叉编译”。嵌入式开发过程中，目的平台常常还没有操作系统，谈不上运行编译器，平台运行至少需要两个东西，bootloader（启动引导代码）以及操作系统核心。这种情况下，我们只能求助于交叉编译。

在一个平台上编译成功后，我们便可以将操作系统烧写入开发板（目的平台）中，但在烧写之前，我们还涉及到了一个计算机与开发板的通信问题，这也就涉及到了串口这个概念，我们通过串口，实现计算机与外部设备的通信，来解决把操作系统烧写入开发板的问题。

### 8.1.1 串口简介

#### (1) 什么是串口

串口是串行接口（Serial Port）的简称，是一种常用的计算机接口。由于连线少、通信控制简单而得到广泛的使用。串口有几种标准，常见的一种称为 RS232 接口的标准是在 1970 年由美国电子工业协会（EIA）和几家计算机厂商共同制定的。RS232 标准应用广泛，其全称是“数据终端设备（DTE）和数据通讯设备（DCE）串行二进制数据交换接口”，该标准定义了串口的电气接口特性和各种信号电平等。

标准串口协议支持的最高数据传输率是 115Kbps。一些改进的串口控制器支持更高甚至 460Kbps 的数据传输率，如增强型串口 ESP（Enhanced Serial Port）和超级增强型串口 Super ESP。

RS232 串口使用 D 型数据接口，最初有 9 针和 25 针两种连接方式。随着计算机技术的不断进步，25 针的串口连接方式已经被淘汰，目前所有的 RS232 串口都使用 9 针连接方式。

## (2) 串口工作原理

串口通过直接连接在两台设备之间的线发送和接收数据，两台设备通信最少需要三根线（发送数据、接收数据和接地）才可以通信。以最常见的 RS232 串口为例，通信距离较近时 (<12m)，可以用电缆线直接连接标准 RS232 端口。如果传输距离远，可以通过调制解调器（MODEM）传输。因为串口设备工作频率低且容易受到干扰，远距离传输会造成数据丢失。

针号	功能说明	缩写
1	数据载波检测	DCD
2	接收数据	RXD
3	发送数据	TXD
4	数据终端准备	DTR
5	信号地	GND
6	数据设备准备好	DSR
7	请求发送	RTS
8	清除发送	CTS
9	振铃指示	BELL

表 8-1 DB9 接口的 RS232 串口数据线定义

表 ?? 是常见的 9 针接口串口各条线定义，RS232 标准的串口不仅提供了数据发送和接收的功能，同时可以进行数据流控制。对于普通应用来说，连接好两个数据线和地线就可以通信。

## (3) Windows 系统下的串口工具

MobaXterm 又名 MobaXVT，是一款增强型终端、X 服务器和 Unix 命令集（GNU/Cygwin）工具箱。可以开启多个终端视窗，以最新的 X 服务器为基础的 X.Org，可以轻松地来试用 Unix/Linux 上的 GNU Unix 命令。这样一来，我们可以不用安装虚拟机来试用虚拟环境，然后只要通过 MobaXterm 就可以使用大多数的 linux 命令。MobaXterm 还有很强的扩展能力，可以集成插件来运行 Emacs、Fontforge、Gcc, G++ and development tools、MPlayer、Perl、Curl、Corkscrew、Tcl / Tk / Expect、Screen、Png2Ico、NEdit Midnight Commander 等程序。

MobaXterm 的下载较为简单，进入官网直接下载即可。需要注意的是 MobaXterm 分免费家庭版和收费专业版：

- 家庭版（Home）：家庭版又分便捷版和安装版。便捷版不需要安装，下载压缩包后解压即可使用。安装版则需一步步安装后才能使用。
- 专业版（Professional）：专业版会在 Session 数、SSH tunnels 数和其他一些定制化配置进行限制。

MobaXterm 的界面结构为：

- 主页：打开 MobaXterm 后，绝大部分内容会被主页占据，主页有快捷按钮“start local terminal”以及最近的 Session 记录，可以方便打开终端，进行命令行操作。
- 菜单栏：MobaXterm 的菜单栏如下，分为 menu bar 和 buttons bar 两行。用户可通过 menu bar 设置 MobaXterm。buttons bar 则为用户提供了一系列功能，如用户可通过 Session 启动远程会话，选择创建 SSH、Telnet、Rlogin、RDP、VNC、XDMCP、FTP、SFTP 或串行会话。开始的每个会话都会自动保存并显示在左侧边栏中。
- 侧边栏：侧边栏分为三个视图，分别为 Sessions，Tools，Macros。Sessions 负责管理使用过的 Session 配置，可以在任意 Session 选项上右击进行编辑。Tools 分为四部分 Terminal games、System、Office、NetWork 的工具集合。使用非常的方便。Macros 可以录制操作过程，下次使用时直接回放即可。

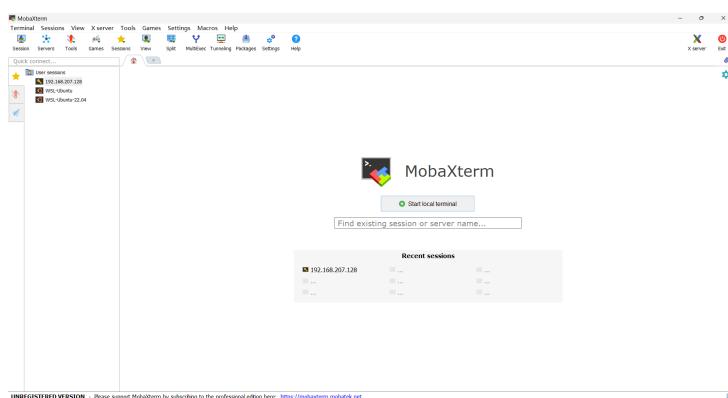


图 8-2 mobaXterm 界面

MobaXterm 是一款功能强大的综合性工具，支持 SSH、RDP、FTP、Serial 等多种通信协议。在本节中，我们将重点介绍其支持的 Serial 协议功能。MobaXterm 可用作串口终端，与串口调试助手等工具相比，它提供了更强大的功能。在使用串口调试助手等工具时，虽然可以用于打印调试信息，但无法实现终端使用，即不能输入命令。

在 MobaXterm 中，通过点击“Session”，选择“Serial”，打开串口设置窗口。首先，选择要配置的串口号，并使用串口线将开发板连接到计算机上。然后，设置波特率，MobaXterm 软件能够自动识别串口，用户只需从下拉菜单中选择相应的波特率。此外，还需进行其他串口功能的设置。点击“Advanced Serial settings”选项卡，可配置串口的其他功能，包括串口号、数据位、停止位、奇偶校验和硬件流控等。若需配置与终端相关的功能，则可点击“Terminal settings”，进行相关设置，如终端字体以及字体大小等。完成设置后，点击窗口下方的“OK”按钮即可。

#### (4) Linux 系统下的串口工具

串口是嵌入式开发使用最多的通信方式。Linux 系统提供了一个串口工具 minicom，可以完成复杂的串口通信工作。若将 minicom 移植到板卡上，我们就可以借助 minicom 对串口进行读写操作。

本节介绍 minicom 的使用。首先是安装 minicom，在 Ubuntu Linux 系统 shell 下输入“`sudo apt-get install minicom`”，按回车键后即可安装 minicom 软件。软件安装好后，第一次使用之前需要配置 minicom。

- 在 shell 中输入 `sudo minicom -s`，出现 minicom 配置界面，下图 ?? 所示。minicom 配置菜单在屏幕中央，每个菜单项都包括了一组配置。



图 8-3 minicom 配置界面

- 用光标键移动高亮条到 **Serial Port setup** 菜单项，按回车键后进入串口参数配置界面，如下图 ?? 所示。串口配置界面列出了串口的配置，每个配置前都有一个英文字母，代表进入配置项的快捷键。首先配置端口，输入小写字母 `a`，光标移动到了`/dev/tty8` 字符串最后，并且进入到编辑模式。以笔者机器为例，修改为`/dev/tty0`，代表连接到系统的第一个串口。

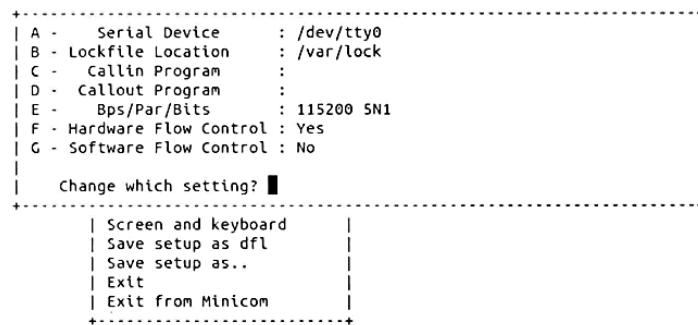


图 8-4 minicom 配置端口界面

- 配置好串口设备后按回车键，保存参数并且回到提示界面。输入小写字母 `e`，进入串口参数配置界面，如下图 ?? 所示。串口参数界面可以配置串口波特率、数据位、停止位等信息。一般只需要配置波特率，如在笔者机器上需要配置波特率是 38400，输入小写字母 `d`，屏幕上方 `current` 字符串后的波特率改变为 38400。

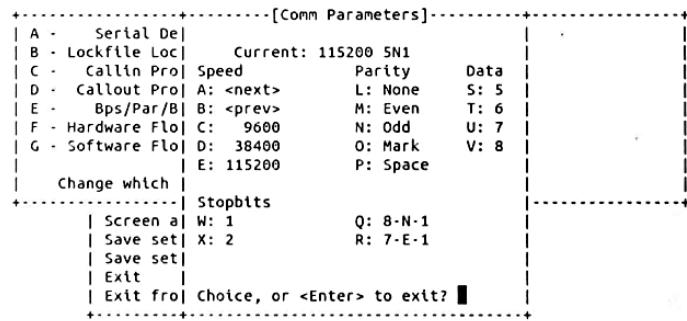


图 8-5 minicom 配置串口参数

4. 设置好波特率后按回车键，保存退出，回到串口配置界面，如下图 ?? 所示。串口被设置为 `tty0`，波特率是 38400，其他配置使用默认设置。如果保存配置，直接按回车键退出。选择 `Save setup as dfl` 选项后按回车键，配置信息被保存为默认配置文件，下次启动的时候会自动加载。保存默认配置后，选择 `Exit` 选项后按回车键，退出配置界面，minicom 自动进入终端界面。在终端界面会自动连接到串口，如果串口没有连接任何设备，屏幕右下角的状态提示为 Offline。

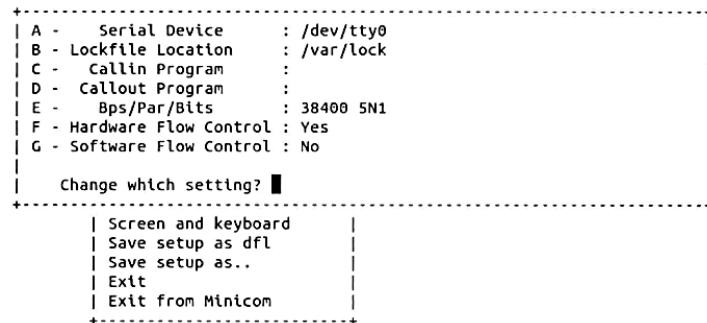


图 8-6 minicom 配置端口结束

5. 退出 minicom，使用 `Ctrl+a` 键，然后输入字母 `z`，出现 minicom 的命令菜单，如下图 ?? 所示。

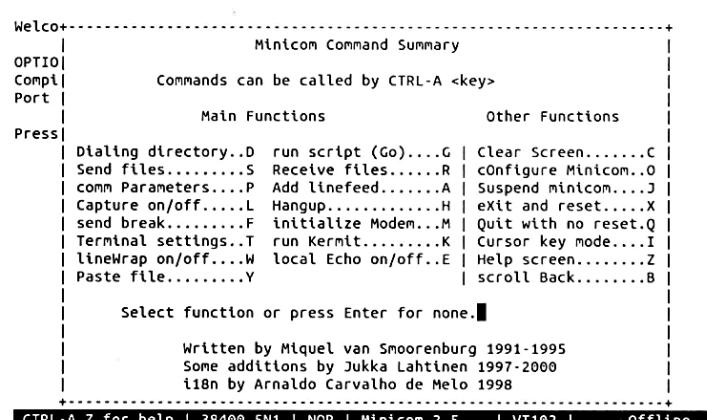


图 8-7 minicom 命令界面

Minicom 提供了丰富的功能，是 Linux 中串口通信和远程管理的重要工具：

- **调试和配置串口设备：**Minicom 可以连接和调试各种串口设备，如调制解调器、路由器、交换机等。用户可以通过 Minicom 查看设备的输出信息，发送指令进行配置和调试。
- **远程终端连接：**Minicom 可以作为一个终端仿真器，用于远程连接到其他计算机或设备。通过串口连接，用户可以在本地计算机上操作远程设备，进行远程管理和维护。
- **数据传输和文件传输：**Minicom 支持通过串口进行数据传输，可用于传输文件、备份数据等。用户可以通过 Minicom 将文件发送到远程设备或从远程设备接收文件。
- **系统调试和故障排查：**Minicom 可用于调试和排查系统故障。通过串口连接到系统控制台，用户可以查看系统的启动信息、错误日志等，帮助定位和解决问题。
- **嵌入式开发和调试：**对于嵌入式系统开发者来说，Minicom 是一个重要的工具。它可用于与嵌入式设备进行通信，进行程序下载、调试和测试。

### 8.1.2 U-Boot 启动介绍

Bootloader 这个名词在嵌入式系统中应用广泛，中文意思可以理解为“启动加载器”。从字面意思来看，Bootloader 就是在系统启动时运行的软件。由于系统启动涉及到硬件和软件的协同操作，因此 Bootloader 成为连接硬件和软件的关键系统软件之一。在我们的后续实验中，将会使用到一个广泛应用的开源 Bootloader 软件——U-Boot。

因此，在这一部分，我们将以 U-Boot 作为示例，介绍 Bootloader 的功能、工作流程、命令以及菜单选项的含义。接着，会详细讲解 U-Boot 中一种常用的启动方式——tftpboot，以及与 tftpboot 相关的 tftp 协议。最后，会分别探讨在 Windows 系统中使用软件 Mobaxterm 时 tftp 的位置以及如何配置使用，在 Linux 系统上如何下载安装 tftp 以及如何配置使用。

#### (1) U-Boot Bootloader 的功能和工作流程

**U-Boot 的功能：**U-Boot 作为一款强大的 Bootloader 软件，拥有多项关键功能。它管理系统启动过程、加载操作系统和初始化硬件，实现了引导加载、固件更新和系统维护。此外，U-Boot 提供了丰富的命令集，允许用户与系统进行交互，进行存储器操作、网络传输以及系统配置，为嵌入式系统的开发和调试带来了灵活性和便利性。虽然 U-Boot 支持多种处理器和操作系统，但其在 PowerPC 系列处理器和 Linux 操作系统上的支持最为完善。它涵盖了许多嵌入式开发中常用的功能，例如查看、修改内存，以及通过网络下载操作系统镜像等，为开发者提供了良好的支持。U-Boot 项目更新迅速，支持的目标板众多，是学习底层开发的极佳工具。

**U-Boot 的工作流程：**大多数 bootloader 的启动流程都分为 stage1 和 stage2 两部分，U-Boot 也不例外。依赖于 CPU 体系结构的代码（如设备初始化代码等）通常都放在 stage1 且可以用汇编语言来实现，而 stage2 则通常用 C 语言来实现，这样可以实现复杂

的功能，而且有更好的可读性和移植性。

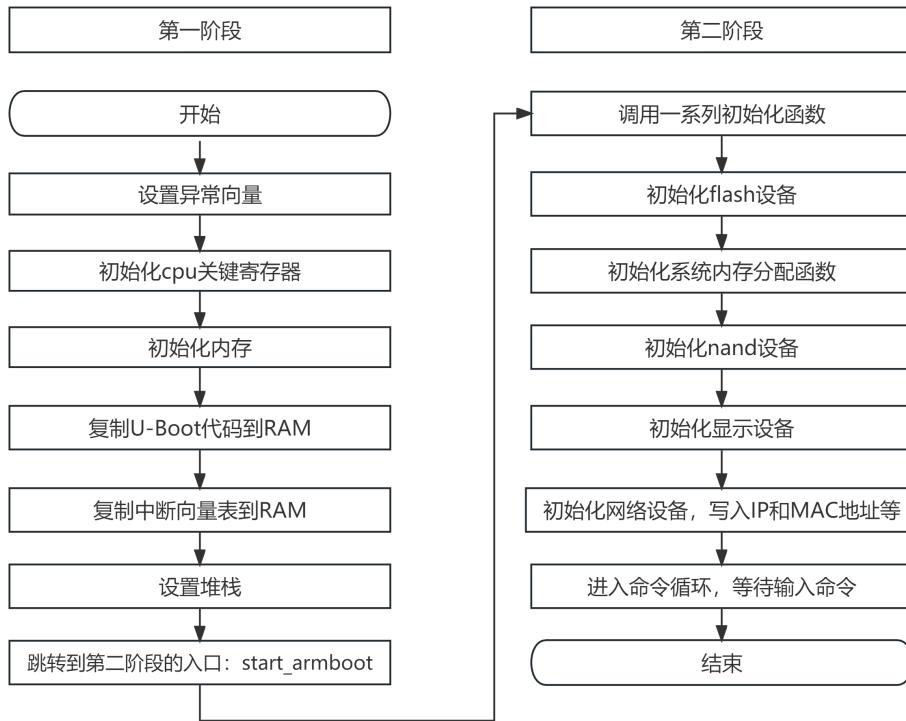


图 8-8 U-Boot 的工作流程

图 8-8 是 U-Boot 的整体工作流程。Stage1 的代码都是与平台相关的，使用汇编语言编写占用空间小而且执行速度快。以 ARM920 为例，Stage1 阶段主要是设置各模式程序异常向量表，初始化处理器相关的关键寄存器以及系统内存。Stage1 负责建立 Stage2 阶段使用的堆栈和代码段，然后复制 Stage2 阶段的代码到内存。

Stage2 阶段一般包括初始化 Flash 器件、检测系统内存映射、初始化网络设备、进入命令循环，接收用户从串口发送的命令然后进行相应的处理。Stage2 使用 C 语言编写，用于加载操作系统内核，该阶段主要是 board.c 中的 start\_armboot() 函数实现。

#### **U-Boot 的命令：**

进入 U-Boot 的命令行模式以后输入 “help” 或者 “?”，然后按下回车即可查看当前 U-Boot 所支持的命令。我们输入 “help(或?) 命令名” 可以查看命令的详细用法。如图 8-9 所示，以 “setenv” 这个命令为例，我们输入如下命令即可查看 “setenv” 这个命令的用法：

**U-Boot 的菜单选项：** uboot 菜单其实就是一个 uboot 中的命令，和其他的命令没有什么差别。如图 8-9 所示，uboot 启动时，如果进入 uboot 命令模式，先运行这个命令，就会打印出一个菜单界面。在 uboot 的命令模式，通过键入 “menu” 命令，同样可以调出这个界面。

这些 U-Boot 中的 menu 选项包含了各种针对系统启动和配置的操作，各选项具体

类别	命令	功能说明
信号查询	bdinfo	查看开发板信息
信号查询	printenv	输出环境变量信息
信号查询	version	查看 U-Boot 的版本号
环境变量	setenv	设置或者修改环境变量的值
环境变量	saveenv	保存修改后的环境变量
内存操作	md	显示内存值
内存操作	nm	修改指定地址内存值, 地址不会自增
内存操作	mm	修改指定地址内存值, 地址会自增
内存操作	mw	使用一个指定的数据填充一段内存
内存操作	cp	用于将 DRAM 中的数据从一段内存拷贝到另一段内存中
内存操作	cmp	用于比较两段内存的数据是否相等
网络操作	ping	检测网络的连通情况
网络操作	dhcp	从路由器获取 IP 地址
网络操作	nfs	通过 nfs 可以在计算机之间通过网络来分享资源
网络操作	tftp	使用 TFTP 协议通过网络下载东西到 DRAM 中
EMMC 和 SD 卡操作	mmc info	输出 MMC 设备信息
EMMC 和 SD 卡操作	mmc read	读取 MMC 中的数据
EMMC 和 SD 卡操作	mmc write	向 MMC 设备写入数据
EMMC 和 SD 卡操作	mmc dev	切换 MMC 设备
FAT 格式文件操作	fatinfo	查询指定 MMC 设置指定分区的文件系统信息
FAT 格式文件操作	fatls	查询 FAT 格式设备的目录和文件信息
FAT 格式文件操作	fstype	查看 MMC 设备某个分区的文件系统格式
FAT 格式文件操作	fatload	命令用于将指定的文件读取到 DRAM 中
BOOT 操作	bootz	自动 zImage 镜像文件
BOOT 操作	bootm	启动 uImage 镜像文件
BOOT 操作	boot	读取环境变量 bootcmd 来启动 Linux 系统
其他常用操作	reset	复位重启
其他常用操作	go	跳到指定的地址处执行应用
其他常用操作	run	运行环境变量中定义的命令
其他常用操作	mtest	简单的内存读写测试命令

表 8-2 U-Boot 的命令表

功能分析如下：

[1] **Download u-boot.bin to Nand Flash:** 用于将 u-boot.bin 文件下载到 Nand Flash 中，是更新或安装 U-Boot 引导程序的选项。

[2] **Download Eboot (eboot.nb0) to Nand Flash:** 将 Eboot (eboot.nb0) 下载到 Nand Flash，可能是针对特定引导程序或固件的更新。

[3] **Download Linux Kernel(zImage.bin) to Nand Flash:** 将 Linux 内核 (zImage.bin) 下载到 Nand Flash 中，用于更新或安装 Linux 内核。

[4] **Download stepldr.nb1 to Nand Flash:** 将 stepldr.nb1 下载到 Nand Flash，可能是针对特定引导程序或固件的更新。

[5] **Set TFTP parameters (PCIP, TQ2440 IP, Mask IP...):** 用于设置 TFTP 参数，包括

```
=> help setenv
setenv - set environment variables

Usage:
setenv [-f] name value ...
  - [forcibly] set environment variable 'name' to 'value ...'
setenv [-f] name
  - [forcibly] delete environment variable 'name'
```

图 8-9 U-Boot 命令行 help 使用实例

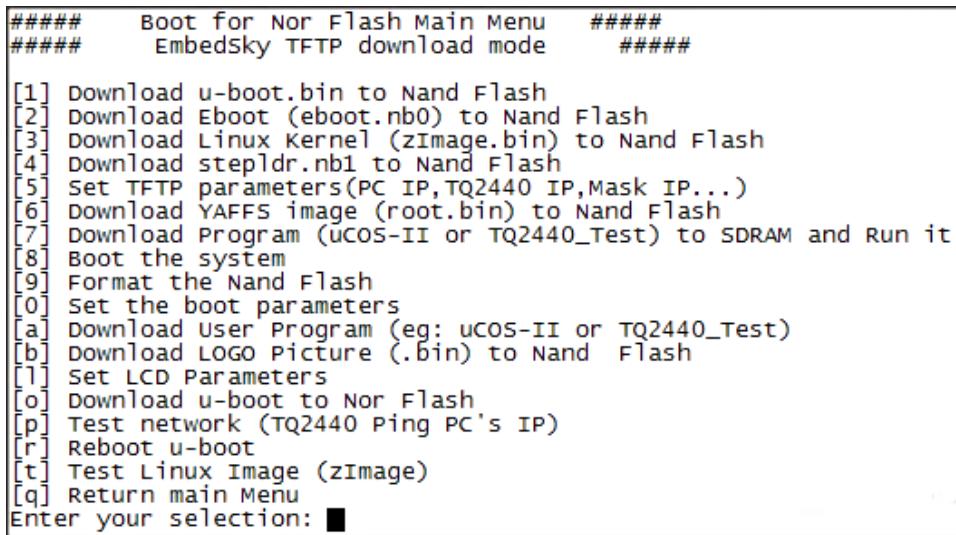


图 8-10 U-Boot 的菜单选项

PC 的 IP 地址、TQ2440 的 IP 地址、子网掩码等。

**[6]Download YAFFS image (root.bin) to Nand Flash:** 将 YAFFS 镜像 (root.bin) 下载到 Nand Flash 中，可能是针对系统根目录的更新或安装。

**[7]Download Program (ucos-II or TQ2440\_Test) to SDRAM and Run it:** 将程序 (ucos-II 或 TQ2440\_Test) 下载到 SDRAM 并运行，用于测试或运行特定程序。

**[8]Boot the system:** 启动系统，触发系统的启动过程。

**[9]Format the Nand Flash:** 格式化 Nand Flash，清除其上的数据并准备好用于存储。

**[0]Set the boot parameters:** 设置引导参数，包括启动时的配置参数和选项。

**[a]Download user's Program (eg: ucos-II or TQ2440\_Test):** 下载用户程序（例如 ucos-II 或 TQ2440\_Test），用于定制或特定应用程序的更新。

**[b]Download Logo Picture (.bin) to Nand Flash:** 将 Logo 图片 (.bin) 下载到 Nand Flash，可能是更新系统启动时显示的 Logo。

**[l]Set LCD Parameters:** 设置液晶显示屏 (LCD) 参数，包括分辨率、刷新率等。

**[o]Download u-boot to Nor Flash:** 将 U-Boot 下载到 Nor Flash，用于更新或安装 U-Boot 引导程序。

**[p]Test network (TQ2440 Ping PC's IP):** 对网络进行测试，Ping TQ2440 到 PC 的 IP 地址，用于网络连接的调试和测试。

**[r]Reboot u-boot:** 重启 U-Boot 引导程序。

**[t]Test Linux image (zImage):** 测试 Linux 镜像 (zImage)，可能是验证镜像完整性或针对错误的测试。

**[q]Return main Menu:** 返回主菜单，退出当前菜单并回到主界面。

## (2) tftpboot 与 tftp 的关系及使用

tftpboot 是 U-Boot 中一种常见的启动方式，它通过 TFTP 协议从网络中加载启动所需的镜像文件。

TFTP (Trivial File Transfer Protocol, 简单文件传输协议) 是 TCP/IP 协议族中的一个用来在客户机与服务器之间进行简单文件传输的协议，提供不复杂、开销不大的文件传输服务，端口号为 69。TFTP 协议的设计目的主要是为了进行小文件传输，因此它不具备通常的 FTP 的许多功能，例如，它只能从文件服务器上获得或写入文件，不能列出目录，不进行认证。

TFTP 代码所占的内存较小，这对于较小的计算机或者某些特殊用途的设备来说是很重要的，这些设备不需要硬盘，只需要固化了 TFTP、UDP 和 IP 的小容量只读存储器即可。因此，随着嵌入式设备在网络设备中所占的比例的不断提升，TFTP 协议被越来越广泛的使用。

下面分别介绍 windows 系统以及 linux 系统中如何下载安装 tftp 和使用：

**1.windows 系统:** 在 windows 系统中我们使用软件 mobaxterm 来支持 tftp 协议。首先在刚刚新建串口的界面点击 “”Servers”window->TFTP server”

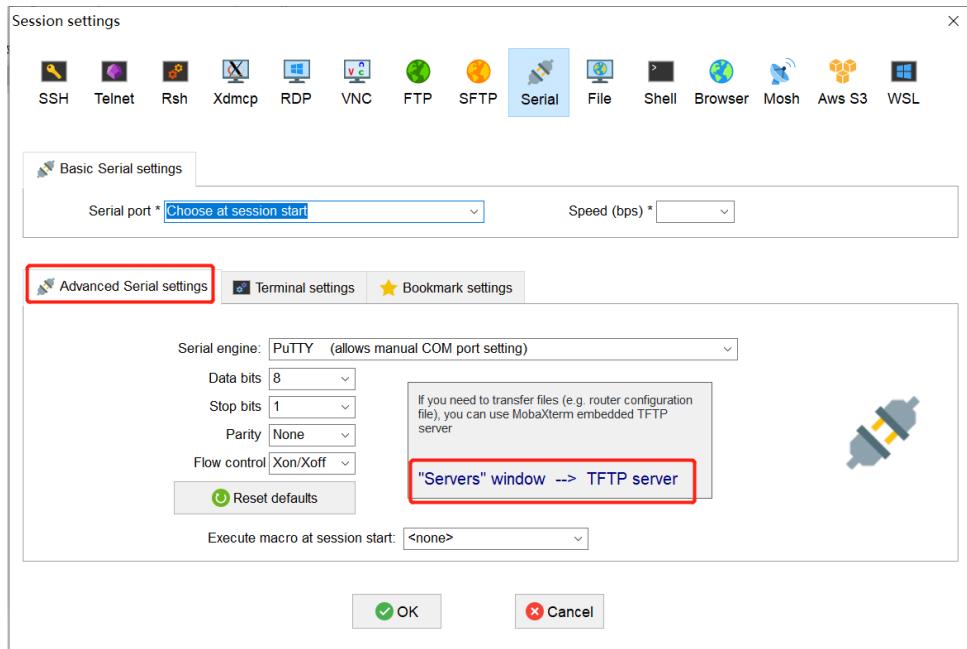


图 8-11 mobaXterm 教程 1

然后在右方“Root directory”可以设置 tftp 服务器根地址，也就是开发板可以访问到的目录路径（包含编译出来的 os.bin 文件）。左上角点击运行按钮可以运行 tftp 服务：

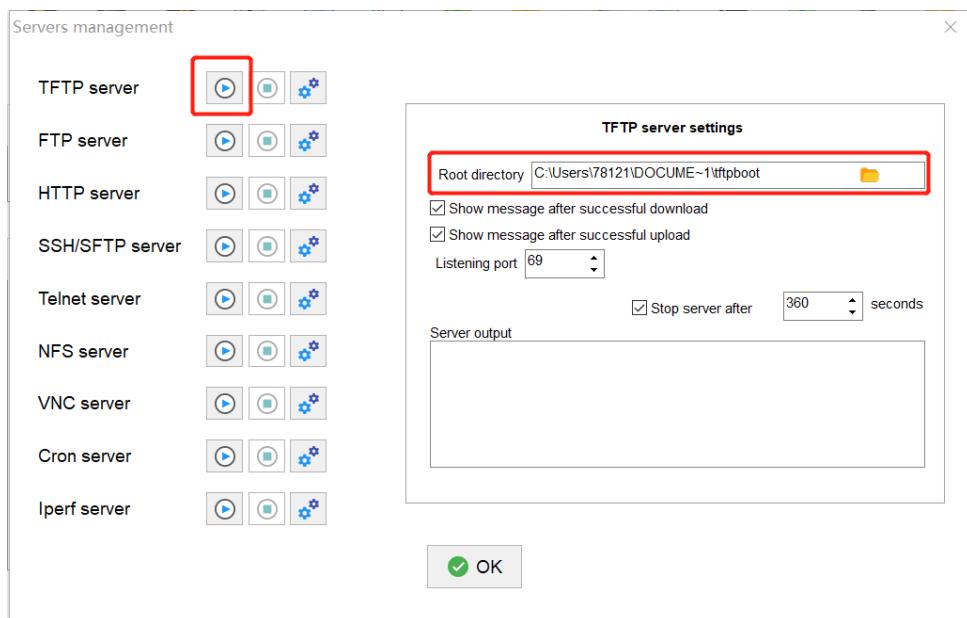


图 8-12 mobaXterm 教程 2

运行按钮旁边的方框按钮为停止服务，右边显示“355”是服务剩余时间 355 秒，每次运行服务只可持续 360 秒，这是 MobaXterm 免费版的限制，服务到期后再通过同样的步骤启动服务。

运行成功如下图：

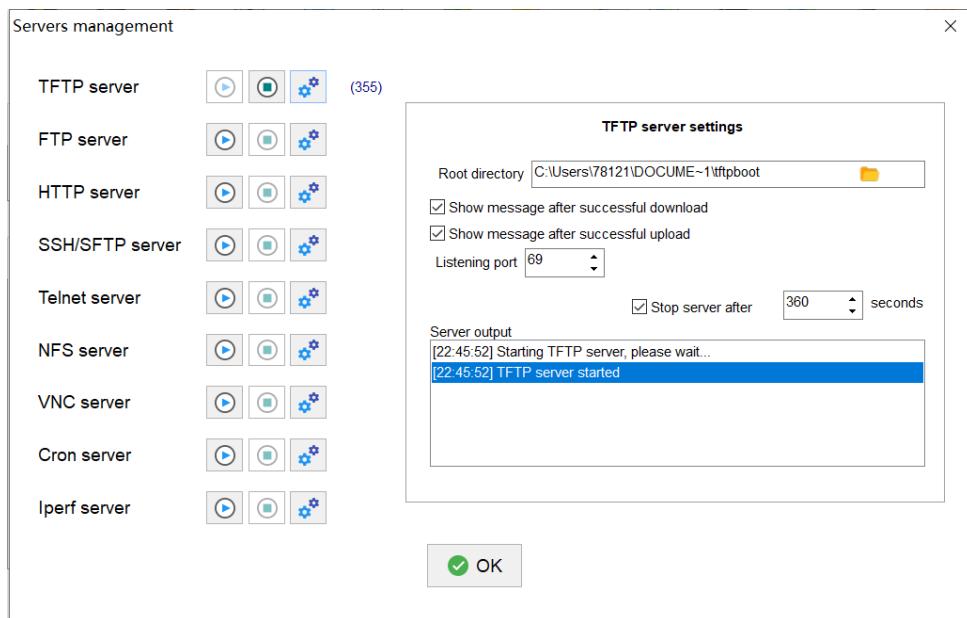


图 8-13 mobaXterm 教程 3

**2.linux 系统：**这里就来介绍 tftp 方式从 linux 主机下载文件到开发板里运行；需要在主机 linux 系统里安装 tftp 服务器。在 Ubuntu 中安装 tftp 服务器的方法如下：

(1) 安装 TFTP 服务器软件：

```
1 sudo apt-get update
2 sudo apt-get install tftpd-hpa
```

(2) 安装 TFTP 客户端程序：

```
1 sudo apt-get update
2 sudo apt-get install tftp
```

(3) 配置 TFTP 服务器：

配置文件通常位于 /etc/default/tftpd-hpa。编辑配置文件并确保以下内容：

```
1 TFTP_USERNAME="tftp"
2 TFTP_DIRECTORY="/work/tftpboot" # TFTP 服务器的根目录，你可以选择其他
3 目录
4 TFTP_ADDRESS="0.0.0.0:69"
5 TFTP_OPTIONS="--secure --create"
```

(4) 创建 TFTP 服务器目录并设置权限：

```
1 sudo mkdir -p /work/tftpboot # 创建 TFTP 服务器的根目录（同(3)中的目录）
2 sudo chmod -R 777 /work/tftpboot # 设置目录权限（同(3)中的目录）
3 sudo chown -R nobody:nogroup /work/tftpboot # 设置目录所有者（同(3)中
4 的目录）
```

(5) 启动 TFTP 服务器服务：

```

1 sudo systemctl restart tftpd-hpa # 启动或重启TFTP服务器服务
2 sudo systemctl enable tftpd-hpa   # 设置TFTP服务器随系统启动而自动启动
    (可选)

```

#### (6) 验证 TFTP 服务器是否运行：

可以使用 netstat 命令检查 TFTP 服务器是否正在监听端口 69:

```
1 sudo netstat -anu | grep :69
```

若出现以下结果，则说明 tftp 服务器正在运行：

1	udp	0	0 0.0.0.0:69	0.0.0.0:*
---	-----	---	--------------	-----------

#### (7) 测试 TFTP 服务器：

使用 tftp 命令测试 TFTP 服务器是否正常工作。例如，可以尝试从 TFTP 服务器下载文件或上传文件：

```

1 tftp 服务器地址
2 tftp> get filename # 从TFTP服务器下载文件到本地
3 tftp> put filename # 从本地上传文件到TFTP服务器

```

设置 TFTP 服务器后，确保防火墙允许 TFTP 流量通过，并根据需要配置相关安全设置。

## 8.2 基于 RISC-V64 的 NPUcore 内核运行

### 8.2.1 昉·星光二代开发板简介

#### 昉·星光 2 简介

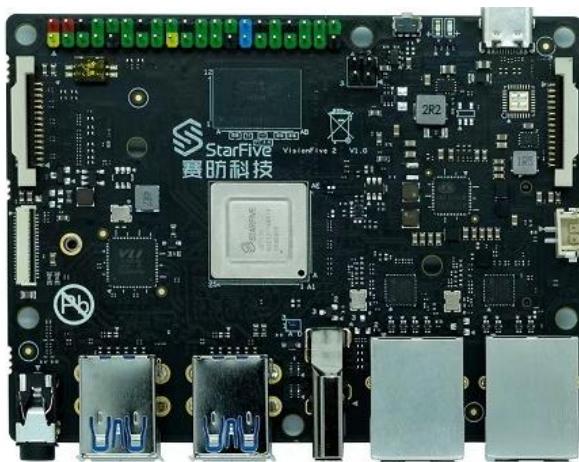


图 8-14 昝·星光 2

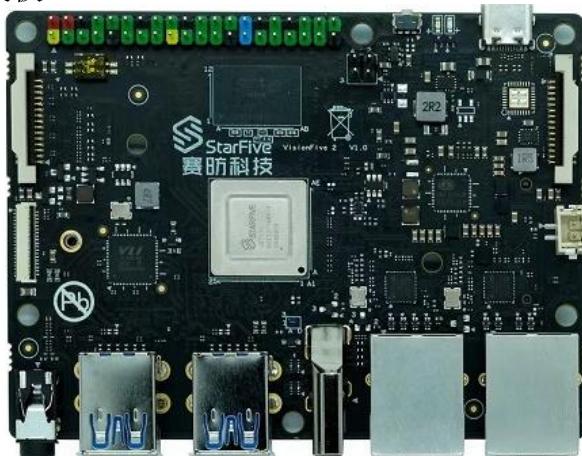
昉·星光 2 是一款集成了 GPU 的高性能 RISC-V 单板计算机。它搭载四核 64 位 RV64GC ISA 的芯片平台（SoC），工作频率最高可达 1.5 GHz，开源的昉·星光 2 具有强大的软件适配性，官方适配 Debian 操作系统，同时通过社区合作适配各种 Linux 发

行版，包括 Ubuntu、OpenSUSE、OpenKylin、OpenEuler、Deepin 等，及在这些操作系统上运行的各类软件。接下来我们的上板实验就在昉 · 星光 2 上进行。

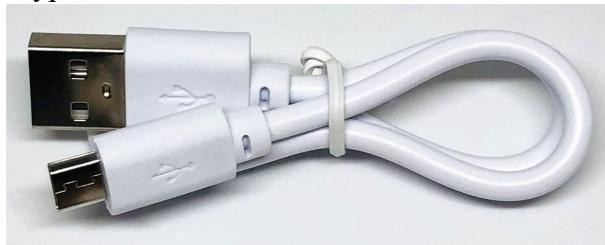
### 硬件准备

在进行上板实验前，首先需要准备以下硬件设备：

- ① 眺 · 星光 2 开发板



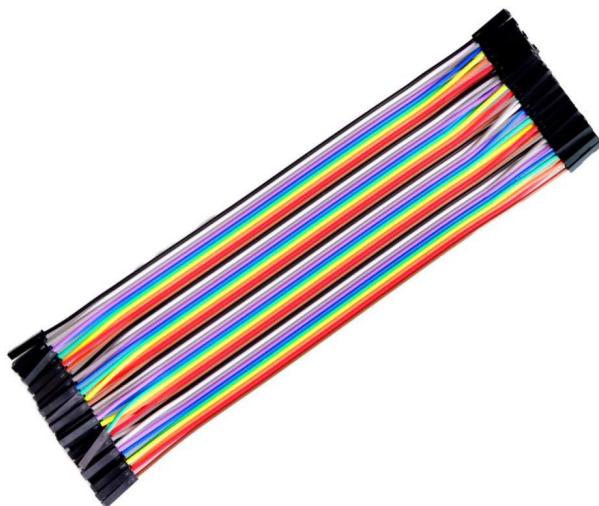
- ② 开发板充电线（type-c 充电线）



- ③ USB TO TTL 串口通信转化器



- ④ 母对母杜邦线



⑤ RJ45 网线



⑥ 拓展坞



拓展坞需要带至少一个网线接口，两个 USB 接口。如果笔记本电脑自带网线接口，并且电脑上的两个 USB 接口都空闲则不需要该拓展坞。

### 昉 • 星光 2 引脚介绍

查阅星光二代官网手册，得到星光二代引脚分布如图所示，

我们只需要进行串口连接，因此只需要关注红框内的 GND，UART TX，UART RX 串口即可，即图中的 6、8、10 号接口。

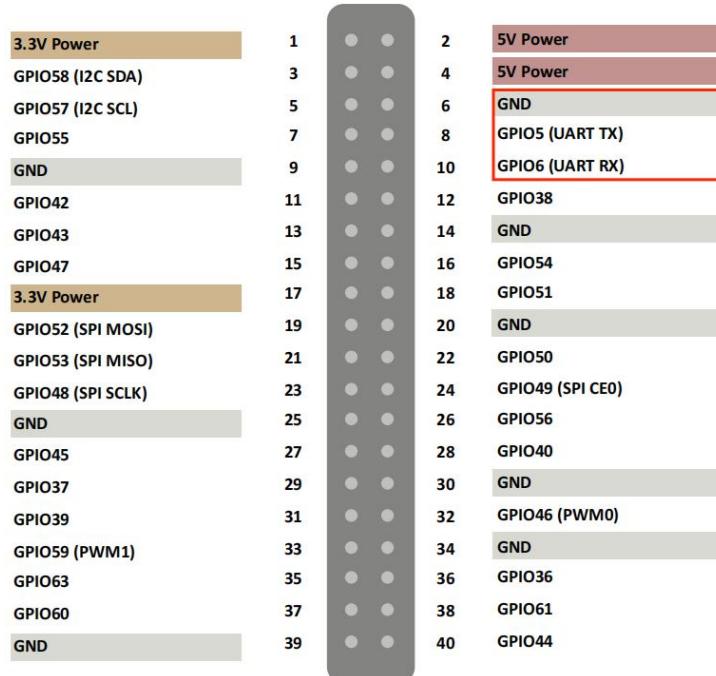


图 8-15 星光二代 pin 图

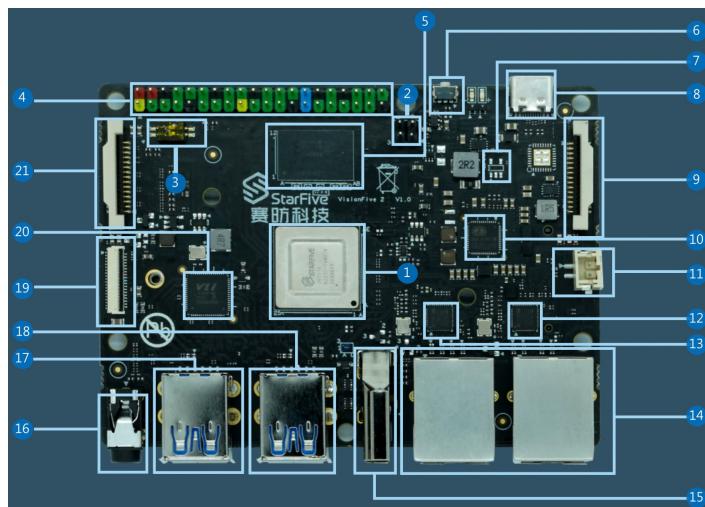
## 硬件连接

### ① 串口连接

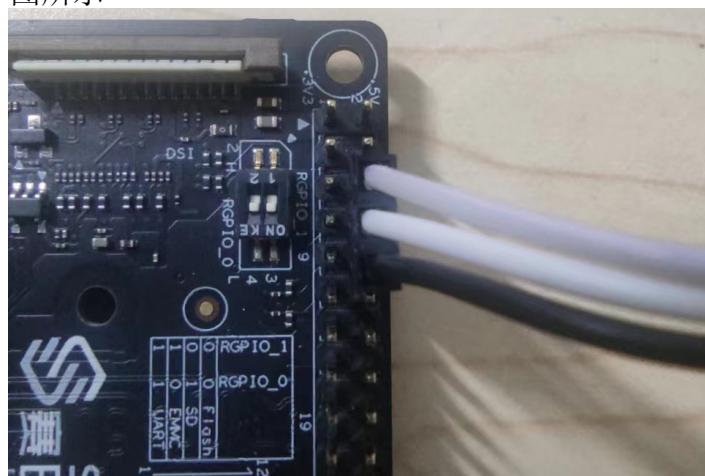
首先，使用杜邦线连接串口通信转换器的 TXD，RXD，GND 接口，仅连接这三个接口即可，剩余悬空，如下图所示。



然后，使用杜邦线连接昉 · 星光 2 单板计算机，连接到图中序号④的引脚处。

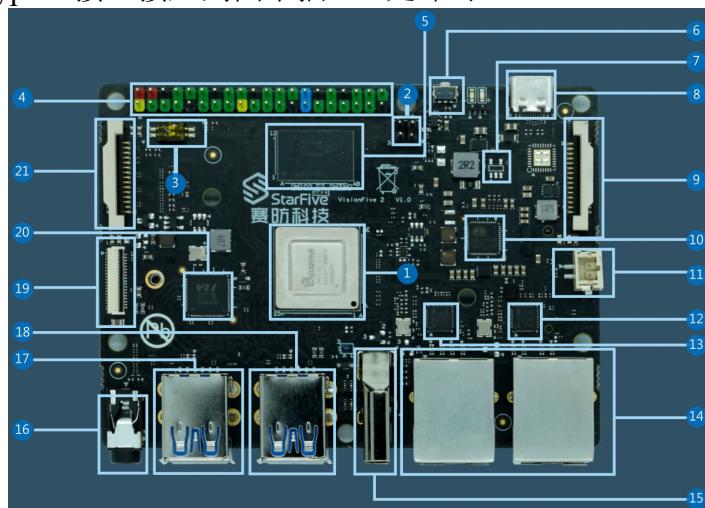


连接好后如下图所示

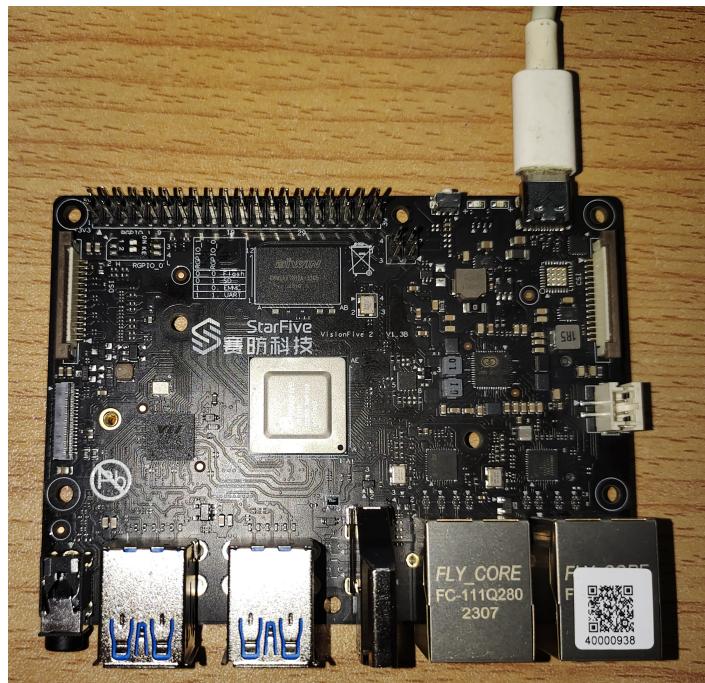


## ② 电源连接

将电源线的 type-c 接口接入到图中插口⑧处即可

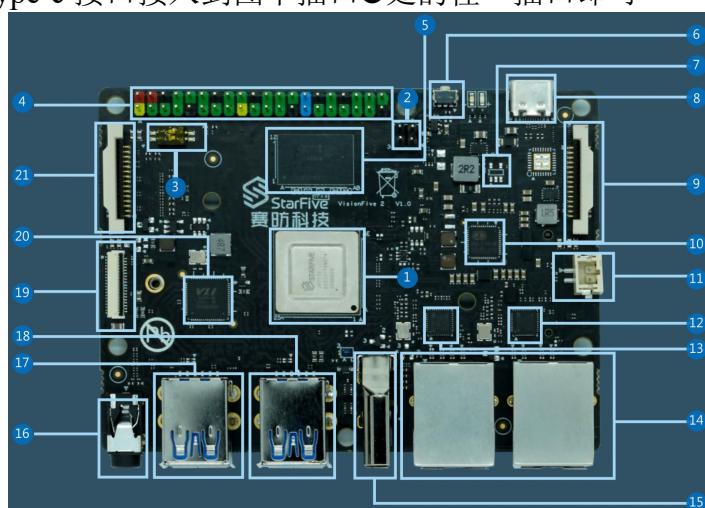


连接好后如下图所示

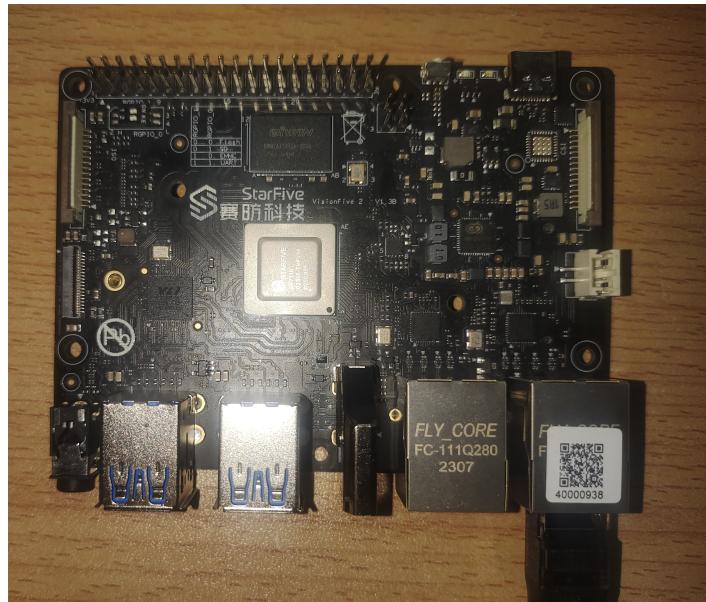


① 网线连接

将电源线的 type-c 接口接入到图中插口④处的任一插口即可



连接好后如下图所示



至此连接完毕，接下来只需将拓展坞连接到电脑即可将开发板连接到电脑，实现插拔方便快捷。



图 8-16 连接完毕全貌

### 8.2.2 面向 RISC-V64 的开发环境搭建与内核运行

#### 1. 编译操作系统内核

熟悉了开发板的相关结构与操作之后，我们开始在开发板上运行 NPUcore。用于开发板上的 NPUcore 源码与在 Qemu 上运行的源码有所不同，需要重新获取。以下时获取方式：

```
git clone https://gitlab.eduxiji.net/202310699101073/oskernel2023-npucore-plus.git
```

以上获取的仓库有多个分支，分别用于不同开发板。本书只介绍星光二号开发板上的运行过程，其他开发板上的运行方式类似，读者可以自行探索。这里获取的

代码默认是 master 分支，需要切换分支，使用下列命令：

```
1 git switch comp-2-starfive
```

在仓库根目录下使用命令可以编译内核：

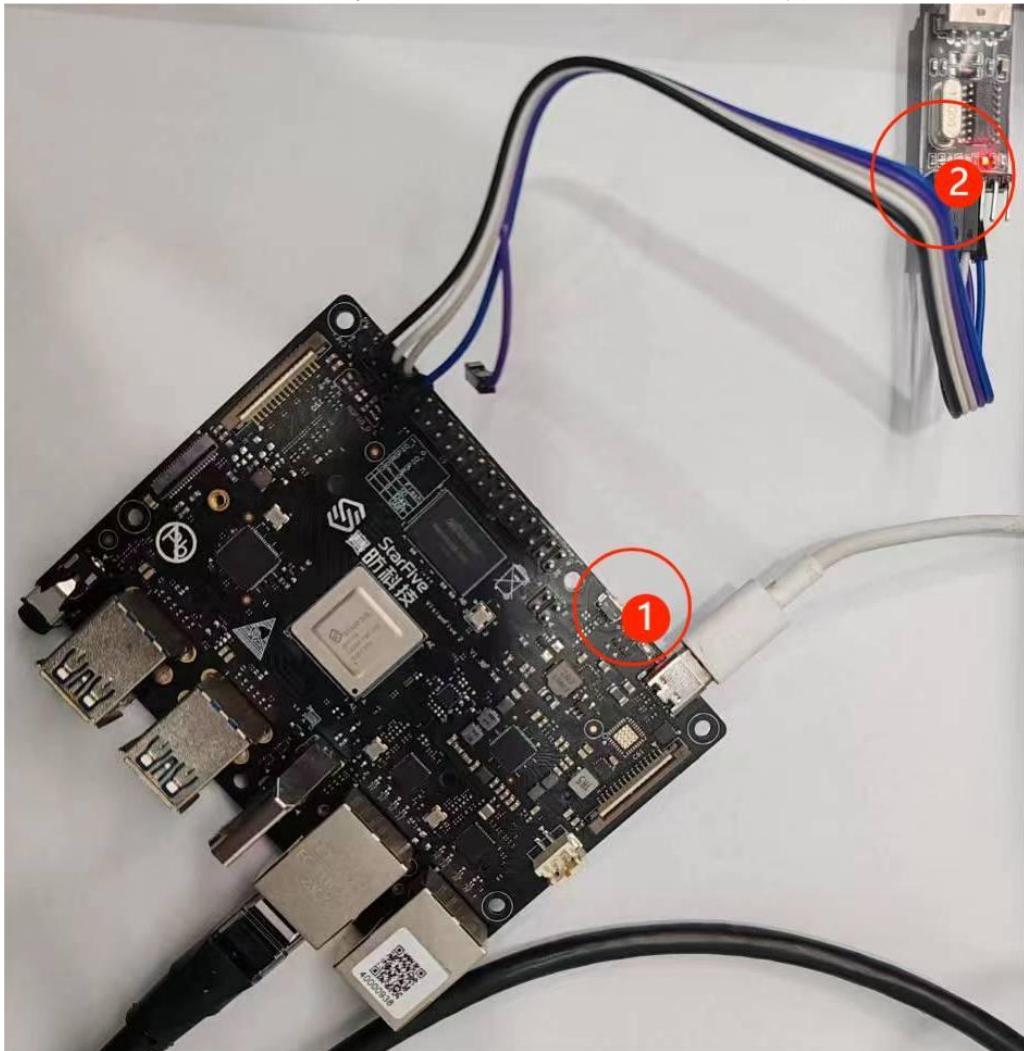
```
1 make
```

编译完成后，会生成 os.bin 文件，这个便是烧入开发板的内核镜像。由于 NPUCore 直接使用的是 risc-v64 的交叉编译工具，该操作可以直接生成 risc-v64 内核。

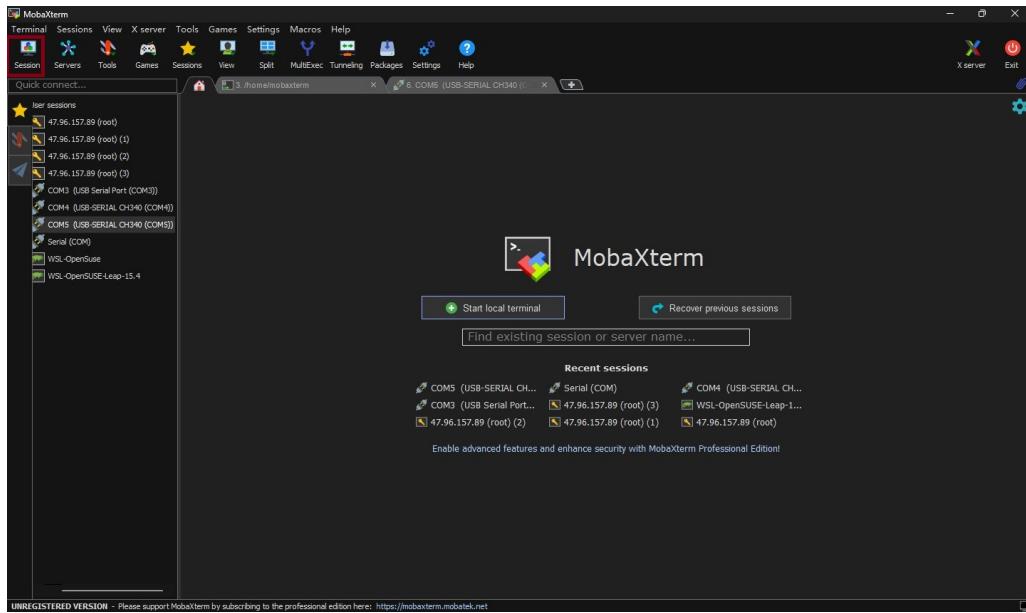
## 2. 串口连接开发板

目前使用的星光二号开发板需要使用串口实现交互。可使用的串口工具种类较多，这里以 windows 操作系统为例，linux 系统的相关串口操作已经在第 8 章第一节“基于嵌入式硬件的操作系统开发流程”中给出，读者可以自行查看，这里使用 MobaXterm 工具连接串口。

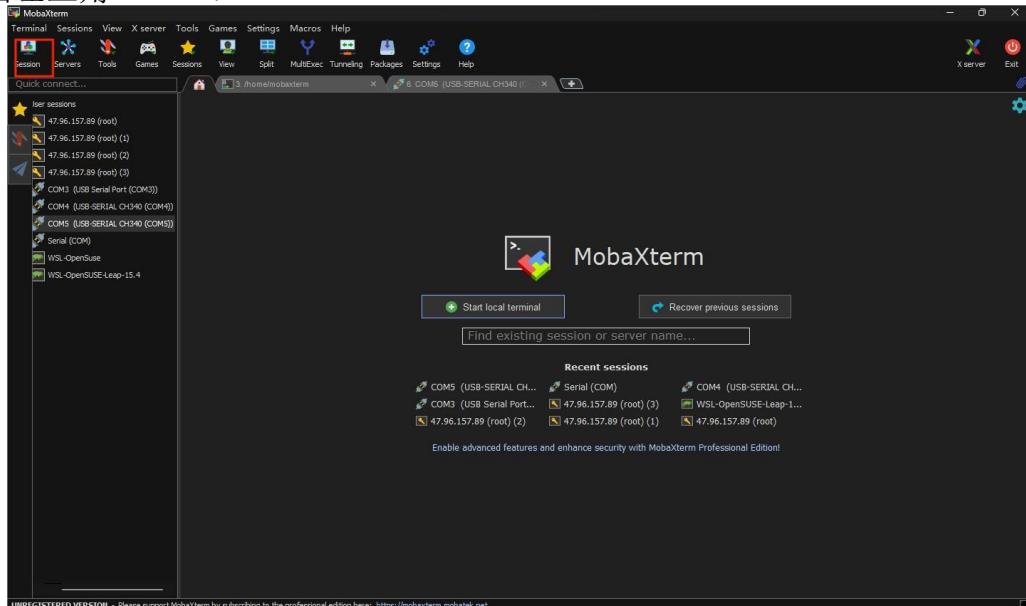
如下图所示，首先将连接好的开发板插入计算机，开发板红色指示灯亮起（①处），串口转化器指示灯红色常亮，有蓝色灯闪烁（②处），说明连接正常。



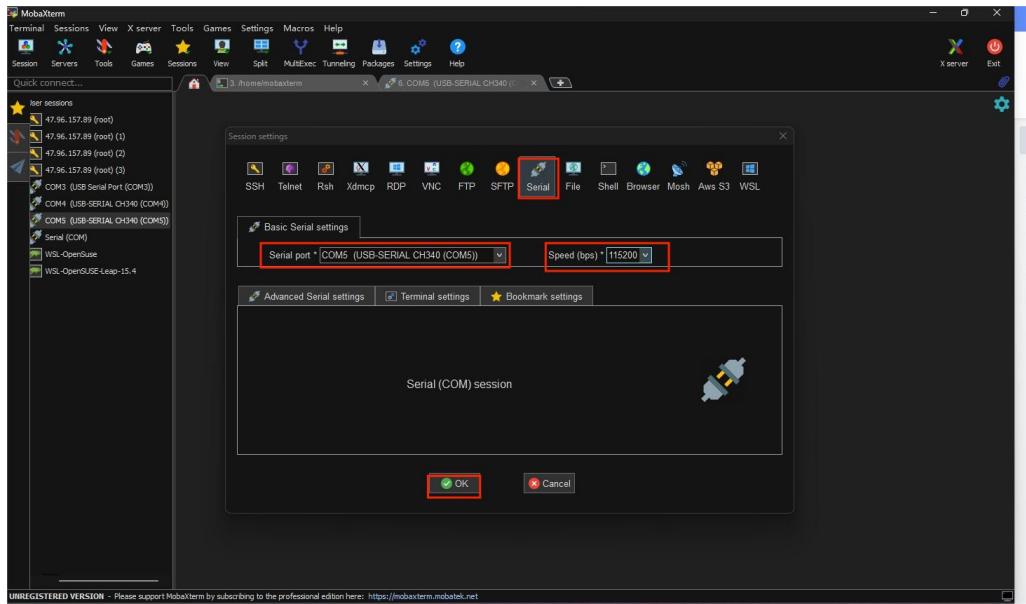
打开 MobaXterm 软件，显示界面如下图所示：



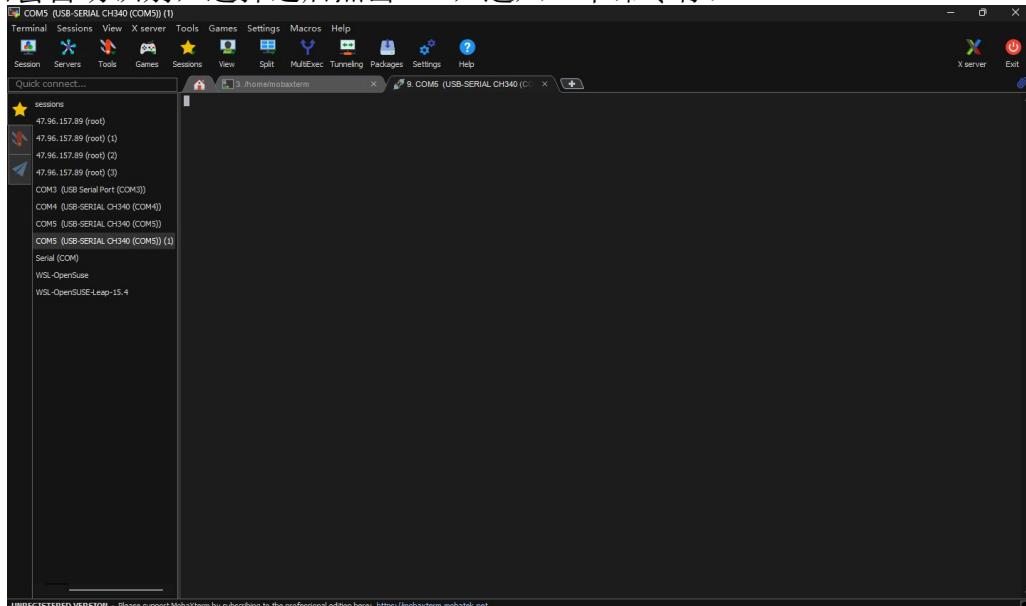
点击左上角 session:



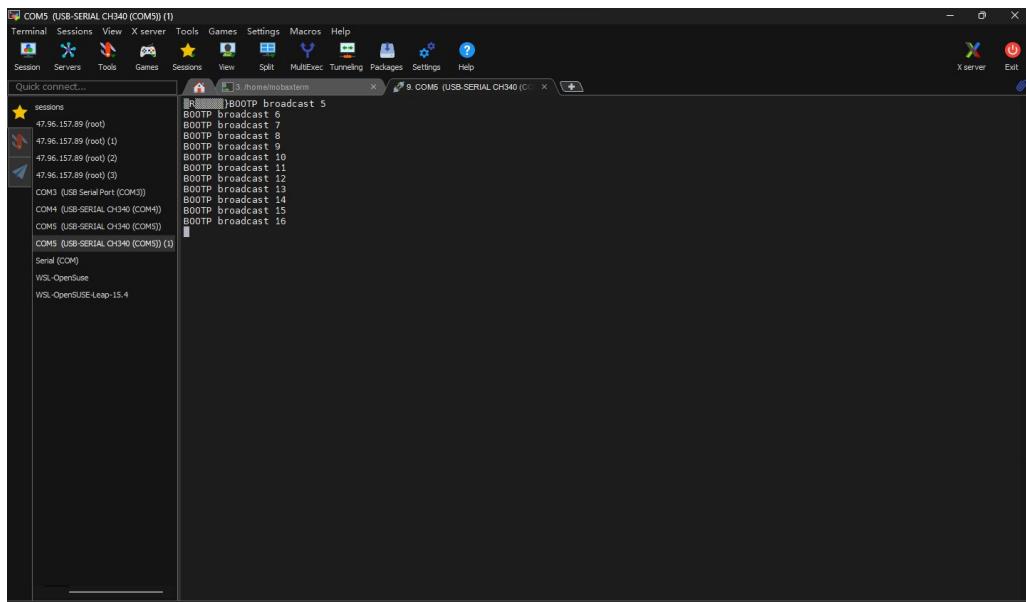
弹出的窗口中选择“serial”，表示使用新建一个串口会话：



需要设置串口端口和波特率。这里波特率选择 115200bps，端口号在开发板正确连接后会自动识别，选择之后点击 OK，进入一个命令行：



输入任意字符回车，观察到开发板终端的输出：

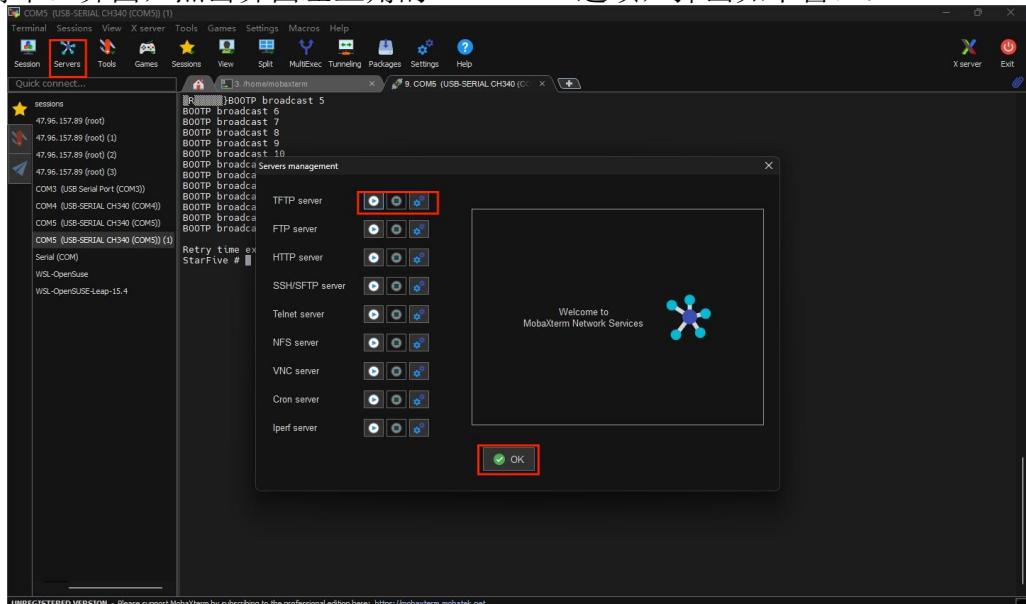


此时开发板和终端已通过串口连接，终端可以使用命令与开发板进行交互。

### 3. 使用 tftp 协议传输内核镜像

由于串口用于交互，所以需要通过 tftp 网络传输内核到开发板。同样 tftp 网络可以使用多种软件搭建，这里由于 MobaXterm 支持 tftp 功能，故用 MobaXterm 举例如何搭建 tftp 服务器以及如何传输内核文件。

保持串口界面，点击界面左上角的“Server”选项，弹出如下窗口：



在右方“Root directory”可以设置 tftp 服务器根地址，也就是开发板可以访问到的目录路径，左上角点击运行按钮可以运行 tftp 服务。

这时 tftp 服务器搭建成功，但是需要设置 IP 地址，将开发板和服务器设置为相同子网，才可以使用 tftp 服务。

如果终端使用的是自动获取 IP 地址，则可以通过以下命令查看 IP 地址：

```
1 ifconfig
```

如果使用的是手动设置的 IP 地址，以上方法也可以查看。如果需要检查连通性，可使用 ping 命令，例如如果查到终端 IP 地址为 168.254.242.234，则可以使用如下命令：

```
1 ping 168.254.242.234
```

在“StarFive #”终端提示符下，输入以下命令设置开发板 IP 地址：

```
1 setenv ipaddr 168.254.242.233
```

此处需要和终端 IP 地址不同但是需要和终端 IP 在同一个局域网中。

执行以下命令设置服务器 IP 地址：

```
1 setenv serverip 168.254.242.234
```

确认 os.bin 已经在 tftp 服务器根目录下，并且 tftp 服务是打开状态，输入下面命令：

```
1 tftpboot 0x80200000 os.bin
```

便可以将内核镜像下载到开发板上，并设置起始地址为 0x80200000。这个地址说明开发板上将从 0x80200000 开始取值。

如果终端无法自动获取 IP 地址，则需要读者自行设置终端 IP 地址。在 windows 设置中选择“网络和 Internet”->“以太网”，如下图：



打开终端网口对应的网络设置。



找到“IP 设置”部分，点击“编辑”，按照如图所示打开 IPv4，并填写 IP 地址，子网前缀长度，网关。这样终端的 IP 地址就设置为相应的 IP 地址了。

这里涉及到了部分 IP 地址相关的知识，再次强调读者要保证开发板和终端的地址处在同一局域网中，也就是网络号相同。这样才能搭建 tftp 服务器。

#### 4. 运行内核

上述过程已经将内核镜像下载到开发板上，而且设置了起始地址为 0x80200000。

接下来输入以下命令便可以开始运行内核：

```
1 go 0x80200000
```

运行结果如下：

```
Bytes transferred = 33837624 (2045238 hex)
StarFive # go 0x80200000
## StarFive application at 0x80200000 ...
[kernel] Console initialized.
.last 48553 Physical Frames.
.text [0x80200000, 0x8023a000]
.rodata [0x8023a000, 0x80245000]
.data [0x80245000, 0x80246000]
.bss [0x802246000, 0x802257000]
mapping .text section
mapping .rodata section
mapping .data section
mapping .bss section
mapping physical memory
mapping memory mapped registers
memory mapped registers
[0x80200000, end at 0x82245000, total: 2000000]
[0x80200000, mm init finish
[0x80200000, test finish
[0x80200000, init world
[0x80200000, dev start
[0x80200000, mkdir dev finish
[0x80200000, init_fs finish
[0x80200000, add_initproc finish
[0x80200000, done]
```

至此内核启动完成。

### 8.3 基于 LoongArch 的 NPUcore 内核运行

## 第 9 章 编写面向 POSIX 标准的系统调用

### 9.1 POSIX 标准简介

#### 9.1.1 什么是 POSIX 标准

可移植操作系统接口（英语：Portable Operating System Interface，缩写为 POSIX）是 IEEE 为要在各种 UNIX 操作系统上运行软件，而定义 API 的一系列互相关联的标准的总称，其正式称呼为 IEEE Std 1003，而国际标准名称为 ISO/IEC 9945。此标准源于一个大约开始于 1985 年的项目。POSIX 这个名称是由理查德·斯托曼（RMS）应 IEEE 的要求而提议的一个易于记忆的名称。它基本上是 Portable Operating System Interface（可移植操作系统接口）的缩写，而 X 则表明其对 Unix API 的传承。

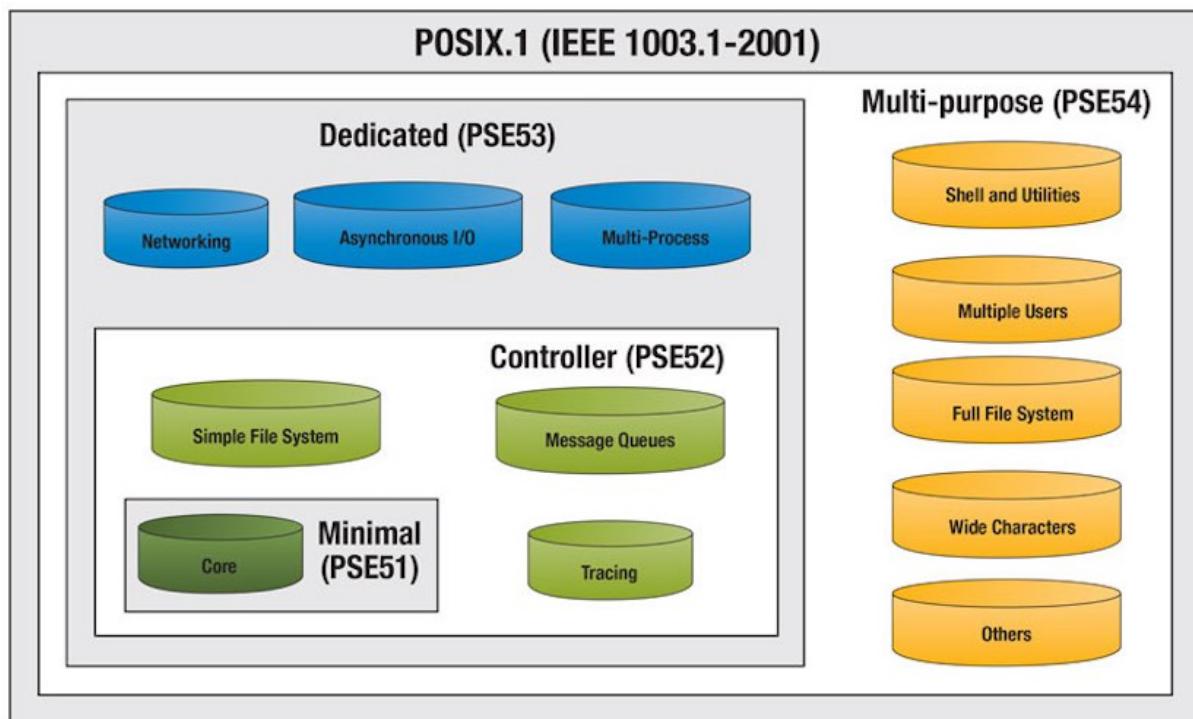


图 9-1 POSIX.1

POSIX 标准的起源可以追溯到 1984 年，当时由于 UNIX 系统的多样化和分化，导致了应用程序的可移植性和兼容性问题。为了解决这个问题，IEEE 开始了一个项目，旨在统一 UNIX 系统的核心编程接口、命令行 shell 和实用工具接口等方面。POSIX 标准的名称是由 Richard Stallman（自由软件运动的创始人之一）建议的，因为他认为这个名字比原来的 IEEE-IX 更容易发音和记忆。

#### 9.1.2 谁遵守 POSIX 标准

常见的支持 POSIX 标准的操作系统有 Linux、BSD、Solaris、macOS 等。这些操作系统都实现了 POSIX 标准中规定的系统调用、库函数、命令和工具等，从而使得开发

者可以在不同的平台上使用相同或类似的 API 编写程序。Windows 操作系统也有部分支持 POSIX 标准，主要是为了吸引 UNIX 用户和提高与 Linux 的竞争力。

### 9.1.3 为什么提出 POSIX 标准

POSIX 标准的优点主要是提高了操作系统和应用程序的可移植性、兼容性和互操作性，促进了软件的开发和交流，降低了开发成本和维护难度，增强了系统的稳定性和安全性。

因为 POSIX 是一系列 API 标准的总称，所以对于不同的操作系统，只要支持一个统一接口，基于这些接口的软件就能在不同操作系统运行。例如：系统 A 实现 fork 的系统调用是 A\_fork，系统 B 实现 fork 的系统调用是 B\_fork，给系统 A 编写的程序如果要移植到系统 B 上，需要修改每一处调用 A\_fork 的代码。如果系统 A 和 B 都遵循标准 POSIX，把自己的 fork 封装到一个通用的 POSIX\_fork 调用里，然后把这样遵循标准的函数都集中到 unistd.h 头文件里，那么应用程序只需要用这个头文件里的函数就可以实现不同系统之间的移植。

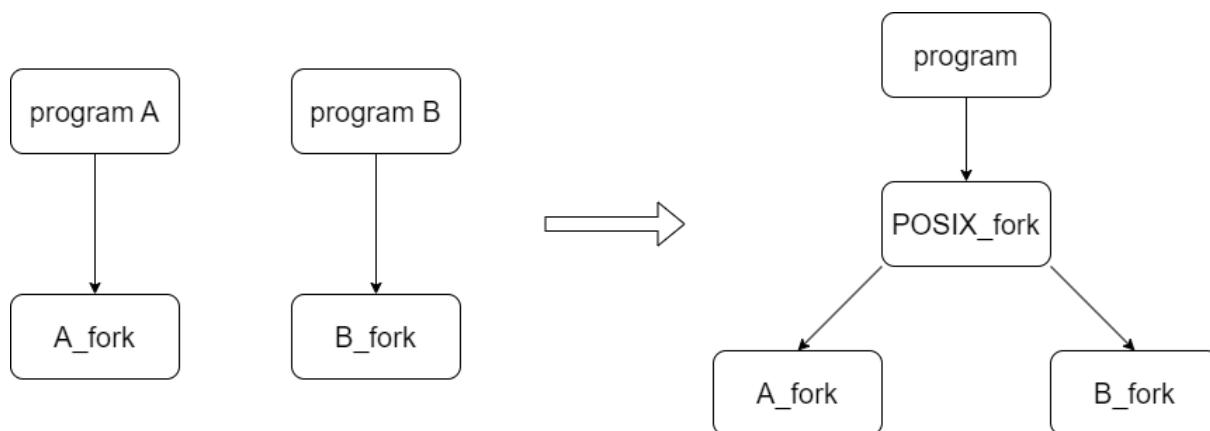


图 9-2 POSIX in Program

## 9.2 Busybox 简介

### 9.2.1 简单介绍

表 9-1 Busybox 源码目录结构

序号	目录名称	功能说明
1	applets	实现 applets 框架的文件，目录中包含了几个 main0 的文件。
2	applets sh	此目录包含了几何作为 shell 脚本实现的 applet 示例。
3	arch	包含用于不同体系架构的 makefile 文件。
4	archival	与压缩相关命令的实现源文件。
5	configs	Busybox 自带的默认配置文件。
6	console-tools	与控制台相关的一些命令。
7	coreutils	常用的一些核心命令。例如 chgrp、m 等。
8	debianutils	针对 Debian 的套件。
9	e2fsprogs	针对 Linux Ext2 FS prog 的命令，例 chattr、Isattr。
10	editors	常用的编辑命令，例如 diff、vi 等。
11	findutils sh	用于查找的命令。
12	include	Busybox 项目的头文件。
13	init	init 进程的实现源码目录。
14	klibc-utils	klibc 命令套件。
15	libbb	与 Busybox 实现相关的库文件。
16	libpwdgrp	libpwdgrp 相关的命令。
17	loginutils	与用户管理相关的命令。
18	mailutils	与 mail 相关的命令套件。
19	miscutils	该文件下是一些杂项命令，针对特定应用场景。
20	modutils	与模块相关的命令。
21	networking	与网络相关的命令，例 arp。
22	printutils	Print 相关的命令。
23	procps	与内存、进程相关的命令。
24	runit	与 Runit 实现相关的命令。
25	shell	与 shell 相关的命令。
26	sysklogd	系统日志记录工具相关的命令。
27	util-linux	Linux 下常用的命令，主要与文件系统操作相关的命令。

Busybox 是一个针对嵌入式系统的轻量级工具集合，旨在通过整合常用的 Linux 命令和服务程序，将它们合并为一个单一的可执行文件。这个项目最初于 1996 年诞生，当时嵌入式系统并不像今天这般普及。Busybox 的最初目的是为软盘系统设计的，因为在那个时候，可移动存储介质的容量十分有限，软盘是主要的存储媒介之一。

Busybox 的设计理念非常巧妙。相较于单独存放每个命令所需的存储空间，Busybox 通过将不同命令的共享部分整合到一起，极大地减小了可执行文件的体积。举例来说，诸如 grep 和 find 这样的命令，尽管功能有所差异，但它们都需要从文件系统中搜索文件，Busybox 将这部分代码进行共享，从而节省了空间。

Busybox 的核心特点在于其高度紧凑的特性。它可以包含最基本的系统命令，例如文件列表显示命令 ls 和文件内容查看命令 cat，同时也能整合更复杂的程序，如文本搜

索命令 grep 和文件查找命令 find，甚至还能将 HTTP 服务器整合进同一个软件包中。

对于嵌入式系统来说，存储空间十分宝贵，而 Busybox 的存在则为这些系统提供了解决方案。通常情况下，Busybox 的可执行文件大小仅约 1MB 左右，相比于分散存放各个命令所需的存储空间，这是一个相当节省空间的选择。用户可以通过建立链接的方式，与传统的命令一样使用 Busybox，只不过它将多个功能整合到一个文件中，从而在嵌入式系统中占用更少的存储空间。

Busybox 源码目录结构图如上，方便以后对 Busybox 做裁剪的时候参考。

### 9.2.2 工作原理

Busybox 利用了 shell 传递给 C 语言 main() 函数的参数，回想一下 C 语言 main() 函数的定义：int main(int argc,char \*argv[])

在 main() 函数的定义中 argc 是传递进来的参数个数，argv 是一个字符串数组，数据的每一项都是一个参数内容。其中，argv[0] 是从命令行调用的程序名。下面是一个简单的程序，使用 argv[0] 确定调用来自哪个程序。

```

1 //test.c
2 #include <stdio.h>
3 /*定义主函数*/
4 int main(int argc, char *argv[])
5 {
6     int i;
7     for(i=0;i<argc ;i++){
8         //for循环语句
9         printf("argv[%d]=%s\n",i,argv[i]);//打印程序参数内容
10    }
11    return 0;
12 }
```

调用这个程序会显示所调用的第一个参数是该程序的名字。可以对这个可执行程序重新进行命名，此时再调用就会得到该程序的新名字。另外，可以创建一个到可执行程序的符号链接，在执行这个符号链接时，就可以看到这个符号链接的名字。

```

1 $ gcc -Wall -o test test.c
2 $ ./test arg1 arg2
3 argv[0]=./test
4 argv[1]=arg1
5 argv[2]=arg2
6
7 $ mv test newtest
8 $ ./newtest arg1
9 argv[0]=./newtest
10 argv[1]=arg1
11
12 $ ln -s newtest linktest
13 $ ./linktest arg
14 argv[0]=./linktest
15 argv[1]=arg
```

Busybox 使用符号链接屏蔽了程序调用细节。从用户的角度看，使用 Busybox 与使

用传统的命令效果是相同的。Busybox 为其包含的每个系统程序都建立了类似的符号链接。当用户使用符号链接调用 Busybox 的时候，Busybox 通过 argv[0] 参数调用对应的功能函数。

### 9.2.3 使用方法

Busybox 的编译过程与 Linux 内核的编译类似。

Busybox 的使用有三种方式：

Busybox 后直接跟命令，如 Busybox ls。

直接将 Busybox 重命名，如 cp Busybox tar。

创建符号链接，如 ln -s Busybox rm。

以上方法中，第三种方法最方便，但为 Busybox 中每个命令都创建一个软链接，相当费事，Busybox 提供自动方法：Busybox 编译成功后，执行 make install，则会产生一个 \_install 目录，其中包含了 Busybox 及每个命令的软链接 Busybox 的使用方法与传统的 Unix 工具类似，通常的语法格式为：Busybox [选项] [命令] [参数]。

Busybox 的命令和参数根据具体的工具而定，可以通过以下方式获取帮助信息：

Busybox –help

#### (1) Busybox 安装

首先进入 Busybox 官网 <https://www.busybox.net/>，选择所需版本下载。

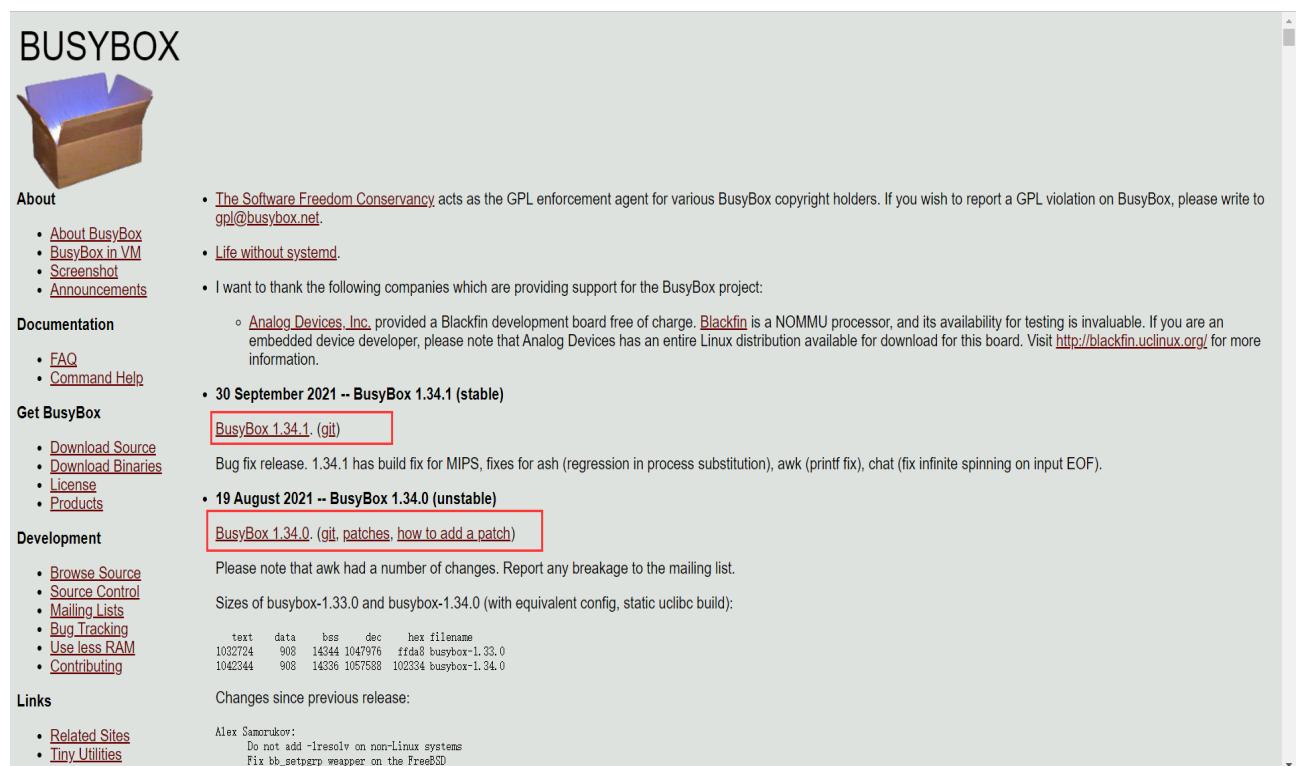


图 9-3 09-02-Busybox 官网界面

接下来将右击解压也可以使用命令行解压。

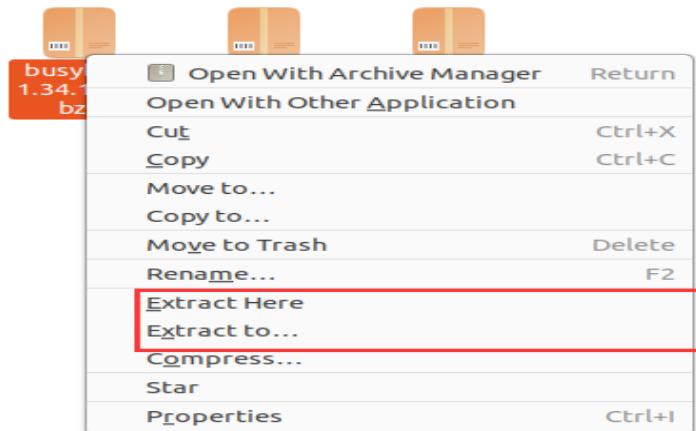


图 9-4 Busybox 安装包解压

### 进入解压后的目录

- 1 make defconfig //使用默认配置，让Busybox包含常用命令和工具
- 2 make menuconfig //在上述基础上，自己更改配置

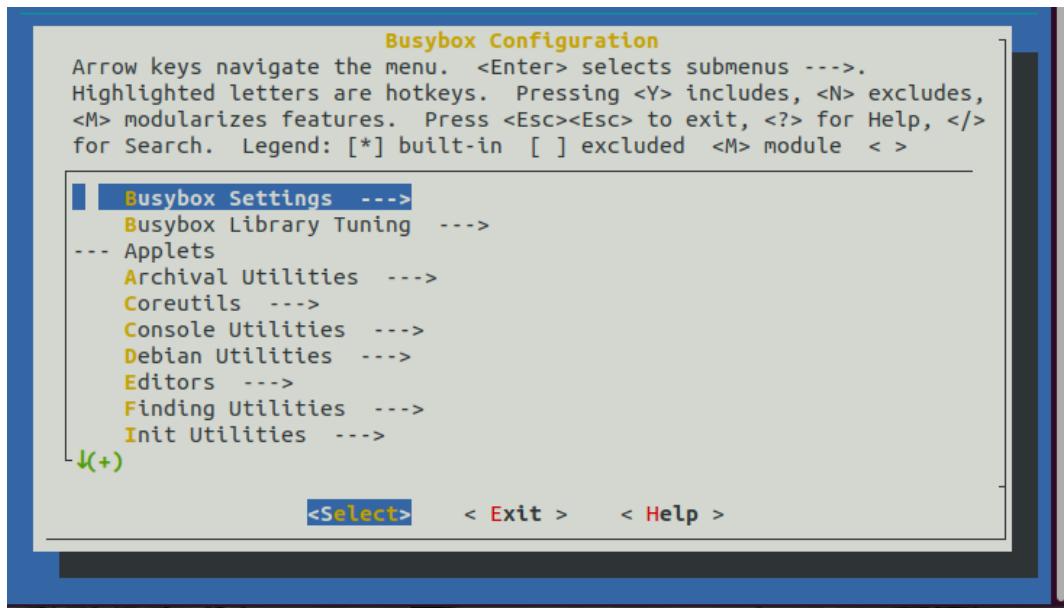


图 9-5 Busybox 配置界面

BusyBox Setting->Build Options->[选]Build Busybox as a static binary (no shared libs)

Shells->choose your default shell(ash):

BusyBox Setting->[\*]Don't use/usr (否则 Busybox 会安装到 ubuntu 的/usr 下，会覆盖原系统原有的命令)

Coreutils—>sync

Linux System Utilities—>nsenter

Linux System Utilities—>Support mounting NFS filesystems(网络文件系统)

Networking Utilities—>inetd (超级服务器)

Busybox settings ->build options ->build with large file support

编译和安装 Busybox

```
1 make
2 make install
```

当出现下图所示时就表明已经安装完成了。

```
CC      util-linux/volume_id/udf.o
CC      util-linux/volume_id/udf.o
CC      util-linux/volume_id/util.o
CC      util-linux/volume_id/volume_id.o
CC      util-linux/volume_id/xfs.o
AR      util-linux/volume_id/lib.a
LINK   busybox_unstripped
Static linking against glibc, can't use --gc-sections
Trying libraries: crypt m resolv rt
Library crypt is not needed, excluding it
Library m is needed, can't exclude it (yet)
Library resolv is needed, can't exclude it (yet)
Library rt is not needed, excluding it
Library m is needed, can't exclude it (yet)
Library resolv is needed, can't exclude it (yet)
Final link with: m resolv
DOC    busybox.pod
DOC    BusyBox.txt
DOC    busybox.1
DOC    BusyBox.html
```

图 9-6 Busybox 安装成功

执行完成后会发现多了一个 \_install 目录。

## 9.2.4 常用的命令

### (1) 安装和登录命令

#### **reboot:**

作用:reboot 命令的作用是重新启动计算机，它的使用权限是系统管理者。

格式:reboot [− n] [− w] [− d] [− f] [− i]

主要参数:

− n: 在重开机前不做将记忆体资料写回硬盘的动作。

− w: 并不会真的重开机，只是把记录写到/var/log/wtmp 文件里。

− d: 不把记录写到/var/log/wtmp 文件里 (− n 这个参数包含了 − d)。

− i: 在重开机之前先把所有与网络相关的装置停止。

#### **mount:**

作用:mount 命令的作用是加载文件系统，它的用权限是超级用户或/etc/fstab 中允许的使用者。

格式:mount − a [− fv] [− t vfstype] [− n] [− rw][− F] device dir

主要参数:

- v: 显示信息，通常和 -f 用来除错。
- a: 将 /etc/fstab 中定义的所有文件系统挂上。
- F: 这个命令通常和 -a 一起使用，它会为每一个 mount 的动作产生一个行程负责执行。在系统需要挂上大量 NFS 文件系统时可以加快加载的速度。

**exit:**

作用:exit 命令的作用是退出系统，它的使用权限是所有用户。

格式:exit

## (2) 文件处理命令

**mkdir:**

作用:mkdir 命令的作用是建立名称为 dirname 的子目录，与 MS DOS 下的 md 命令类似，它的使用权限是所有用户。

格式:mkdir [options] 目录名

主要参数:

- m, -- mode= 模式: 设定权限，与 chmod 类似。
- p, -- parents: 需要时创建上层目录；如果目录早已存在，则不当作错误。
- v, -- verbose: 每次创建新目录都显示信息。

**grep:**

作用:grep 命令可以指定文件中搜索特定的内容，并将含有这些内容的行标准输出。grep 全称是 Global Regular ExpressionPrint，表示全局正则表达式版本，它的使用权限是所有用户。

格式:grep [options]

主要参数:

- c: 只输出匹配行的计数。
- I: 不区分大小写（只适用于单字符）。
- h: 查询多文件时不显示文件名。

**find:**

作用:find 命令的作用是在目录中搜索文件，它的使用权限是所有用户。

格式:find [path][options][expression]

path 指定目录路径，系统从这里开始沿着目录树向下查找文件。它是一个路径列表，相互用空格分离，如果不写 path，那么默认为当前目录。

主要参数:

- depth: 使用深度级别的查找过程方式，在某层指定目录中优先查找文件内容。
- maxdepth levels: 表示至多查找到开始目录的第 level 层子目录。level 是一个非负数，如果 level 是 0 的话表示仅在当前目录中查找。
- mindepth levels: 表示至少查找到开始目录的第 level 层子目录。

### (3) 系统管理命令

#### **reboot:**

作用:df 命令用来检查文件系统的磁盘空间占用情况，使用权限是所有用户。

格式:df [options]

主要参数:

— s: 对每个 Names 参数只给出占用的数据块总数。

— a: 递归地显示指定目录中各文件及子目录中各文件占用的数据块数。若既不指定— s，也不指定— a，则只显示 Names 中的每一个目录及其中的各子目录所占的磁盘块数。

#### **top:**

作用:top 命令用来显示执行中的程序进程，使用权限是所有用户。

格式:top [—] [d delay] [q] [c] [S] [n]

主要参数:

d: 指定更新的间隔，以秒计算。

q: 没有任何延迟的更新。如果使用者有超级用户，则 top 命令将会以最高的优先序执行。

c: 显示进程完整的路径与名称。

#### **free:**

作用:free 命令用来显示内存的使用情况，使用权限是所有用户。

格式:free [— b| — k| — m] [— o] [— s delay] [— t] [— V]

主要参数:

— b — k — m: 分别以字节 (KB、MB) 为单位显示内存使用情况。

### (4) 网络操作命令

#### **ifconfig:**

作用:ifconfig 用于查看和更改网络接口的地址和参数，包括 IP 地址、网络掩码、广播地址，使用权限是超级用户。

格式:ifconfig -interface [options] address

主要参数:

-interface: 指定的网络接口名，如 eth0 和 eth1。

up: 激活指定的网络接口卡。

down: 关闭指定的网络接口。

#### **ip:**

作用:ip 是 iproute2 软件包里面的一个强大的网络配置工具，它能够替代一些传统的网络管理工具，例如 ifconfig、route 等，使用权限为超级用户。几乎所有的 Linux 发行版本都支持该命令。

格式:`ip [OPTIONS] OBJECT [COMMAND [ARGUMENTS]]`

主要参数:

`-V,-Version` 打印 ip 的版本并退出。

`-s,-stats,-statistics` 输出更为详尽的信息。如果这个选项出现两次或多次，则输出的信息将更为详尽。

`-f,-family` 这个选项后面接协议种类，包括 `inet`、`inet6` 或 `link`，强调使用的协议种类。如果没有足够的信息告诉 ip 使用的协议种类，ip 就会使用默认值 `inet` 或 `any`。`link` 比较特殊，它表示不涉及任何网络协议。

## (5) 系统安全相关命令

**su:**

作用:`su` 的作用是变更为其它使用者的身份，超级用户除外，需要键入该使用者的密码。

格式:`su [选项]... [-] [USER [ARG]...]`

主要参数:

`-f, -fast`: 不必读启动文件（如 `csh.cshrc` 等），仅用于 `csh` 或 `tcsh` 两种 Shell。

`-l, -login`: 加了这个参数之后，就好像是重新登陆为该使用者一样，大部分环境变量（例如 `HOME`、`SHELL` 和 `USER` 等）都是以该使用者（USER）为主，并且工作目录也会改变。如果没有指定 `USER`，缺省情况是 `root`。

`-m, -p, -preserve-environment`: 执行 `su` 时不改变环境变数。

**umask:**

作用:`umask` 设置用户文件和目录的文件创建缺省屏蔽值，若将此命令放入 `profile` 文件，就可控制该用户后续所建文件的存取许可。它告诉系统在创建文件时不给谁存取许可。使用权限是所有用户。

格式:`umask [-p] [-S] [mode]`

主要参数:

`-S`: 确定当前的 `umask` 设置。

`-p`: 修改 `umask` 设置。

**chmod:**

作用:`chmod` 命令是非常重要的，用于改变文件或目录的访问权限，用户可以用它控制文件或目录的访问权限，使用权限是超级用户。

格式:`chmod` 命令有两种用法。一种是包含字母和操作符表达式的字符设定法（相对权限设定）`chmod [who] [+|-|=] [mode] 文件名`；另一种是包含数字的数字设定法（绝对权限设定）`chmod [mode] 文件名`。

主要参数:

对字符设定法而言：

操作对象 who 可以是下述字母中的任一个或它们的组合

u: 表示用户，即文件或目录的所有者。

g: 表示同组用户，即与文件属主有相同组 ID 的所有用户。

## (6) 其他命令

**tar:**

作用:tar 命令是 Unix/Linux 系统中备份文件的可靠方法，几乎可以工作于任何环境中，它的使用权限是所有用户。

格式:tar [主选项 + 辅选项] 文件或目录

主要参数:

-c 创建新的档案文件。如果用户想备份一个目录或是一些文件，就要选择这个选项。

-r 把要存档的文件追加到档案文件的末尾。例如用户已经做好备份文件，又发现还有一个目录或是一些文件忘记备份了，这时可以使用该选项，将忘记的目录或文件追加到备份文件中。

-t 列出档案文件的内容，查看已经备份了哪些文件。

## 9.3 支持 BusyBox 所需系统调用

### 9.3.1 从二进制文件中提取系统调用

要从一个二进制文件中提取出相关的系统调用，需要经历如??所示的步骤:

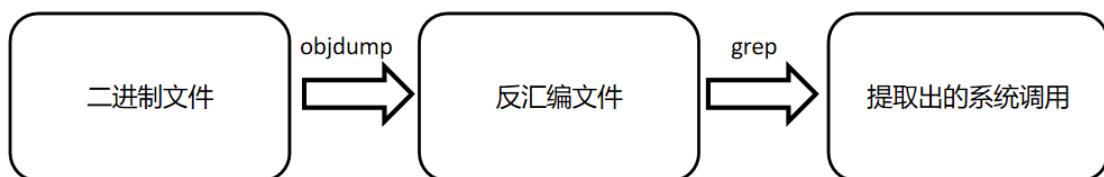


图 9-7 从二进制文件中提取系统调用的步骤

这些指令可以用一行命令完成。

```

1 # 从二进制文件中获取系统调用号，将objfile改为提取的目标二进制文件
2 objdump -d objfile | grep -B 9 ecall | grep "li.a7" | tee syscall.txt
3
4 # 如果报错: objdump: can't disassemble for architecture UNKNOWN! , 是由于当前的objdump并非RISC-V架构, 尝试
5 riscv64-linux-gnu-objdump -d busybox | grep -B 9 ecall | grep "li.a7" |
     tee syscall.txt
  
```

objdump -d objfile: 这部分使用 objdump 工具对目标文件 (objfile) 进行反汇编，显示其汇编代码。

grep -B 9 ecall: 这部分使用 grep 命令在反汇编的输出中查找包含字符串“ecall”的行，并显示该行及其前面的 9 行 (-B 9 选项)。

grep “li.a7”: 在前一步的输出中，再次使用 grep 命令查找包含字符串“li.a7”的行。

tee syscall.txt: 最后，将前两个 grep 命令的输出保存到一个名为 syscall.txt 的文件中，并在屏幕上显示这些输出。

上述三条命令通过管道连接，传递结果，加工完毕后存储到 syscall.txt 文本文件中。而要深入理解这些代码，我们首先要分析各个格式文件的具体内容。

二进制文件的汇编不再进行讲解。下面给出一个反汇编出的程序片段：

```

1 1062c: 0b953c27      fsd fs9,184(a0)
2 10630: 0da53027      fsd fs10,192(a0)
3 10634: 0db53427      fsd fs11,200(a0)
4 10638: 4d3c506f      j 0xd630a
5 1063c: 08100893      li a7,129
6 10640: 00000073      ecall
7 10644: 78fd          lui a7,0xffff
8 10646: 00a8e363      bltu a7,a0,0x1064c
9 1064a: 8082          ret
10 1064c: 5880006f     j 0x10bd4
11 10650: 8082          ret
12 10652: 0000          unimp
13 10654: 0a000893      li a7,160
14 10658: 00000073      ecall

```

让我们结合汇编代码分析调用系统调用的一般结构。上面的代码中有两条 ecall 指令，ecall 指令结合上面一行的 li a7, [系统调用号] 指令，即可实现特定系统调用的调用。在调用之前，还可能存在若干条指令，用于存入系统调用需要的参数。比如，在调用 64 号 write 系统调用时，可能存在若干条 li 指令，将文件描述符、缓冲区指针、缓冲区长度分别写入 a0、a1、a2 寄存器中。

因此，当我们反汇编出如上格式的汇编代码时，为了提取出相关的系统调用，我们首先就要执行 grep -B 9 ecall 指令，将系统调用及前面的若干条指令提取出来。为什么不直接提取 li a7, [系统调用号]？这里的问题在于，a7 并不是仅用于系统调用，有时也作为临时变量参与运算，因此，直接提取是不恰当的，将造成提取出的结果增多。所以我们首先利用 grep -B 9 ecall 指令，把系统调用筛选出来，再获取其系统调用号。

当我们执行了 grep -B 9 ecall 后，获得到的汇编指令集合如下所示。

```

1 --
2 10a4d2: 26a73703      ld a4,618(a4) # 0x1bd738
3 10a4d6: 40a006bb      negw a3,a0
4 10a4da: 547d          li s0,-1
5 10a4dc: 9712          add a4,a4,tp
6 10a4de: c314          sw a3,0(a4)
7 10a4e0: b7e9          j 0x10a4aa
8 10a4e2: 00b50b63      beq a0,a1,0x10a4f8
9 10a4e6: 48e1          li a7,24
10 10a4e8: 4601          li a2,0
11 10a4ea: 00000073      ecall

```

```

12  --
13  10a51a: 000b3797      auipc   a5,0xb3
14  10a51e: 21e7b783      ld      a5,542(a5) # 0x1bd738
15  10a522: 40a0073b      negw    a4,a0
16  10a526: 557d          li      a0,-1
17  10a528: 9792          add    a5,a5,tp
18  10a52a: c398          sw     a4,0(a5)
19  10a52c: 8082          ret
20  10a52e: 03b00893      li      a7,59
21  10a532: 4581          li      a1,0
22  10a534: 00000073      ecall
23  --
24  10a5d8: 7dee          ld      s11,248(sp)
25  10a5da: 8552          mv      a0,s4
26  10a5dc: 7a52          ld      s4,304(sp)
27  10a5de: 6135          addi   sp,sp,352
28  10a5e0: 8082          ret
29  10a5e2: 8a2a          mv      s4,a0
30  10a5e4: d161          beqz   a0,0x10a5a4
31  10a5e6: 48c5          li      a7,17
32  10a5e8: 8552          mv      a0,s4
33  10a5ea: 00000073      ecall
34  --

```

如此，就基本确保 `li a7` 立即数指令为系统调用时使用。此时再进行 `grep "li.a7"`，就可以提取出所有系统调用号。最后的结果使用 `tee syscall.txt` 存入到 `syscall.txt` 中。至此提取系统调用的命令的全过程也就清晰了。

### 9.3.2 BusyBox 系统调用功能分类

在本节中，我们将深入探讨 BusyBox 和 NPUCore 操作系统内核之间的系统调用交集。

首先，我们从 BusyBox 的系统调用列表开始，这个列表包含 BusyBox 运行所需的**各种系统调用**。然后与 NPUCore 中的系统调用进行对照得到了一共 71 个系统调用。我们在进行对照的基础之上，对我们所得到的系统调用进行了分类，我们将其分为：

- 文件系统操作
- 进程控制
- 信号处理
- 网络通信
- 文件描述符操作
- 系统信息与时间
- 用户和权限管理
- 内存管理
- 其他功能

接下来，我们将分类对每一类的部分系统调用进行介绍，在最后我们将给出所有系统调用与其功能的对应表。

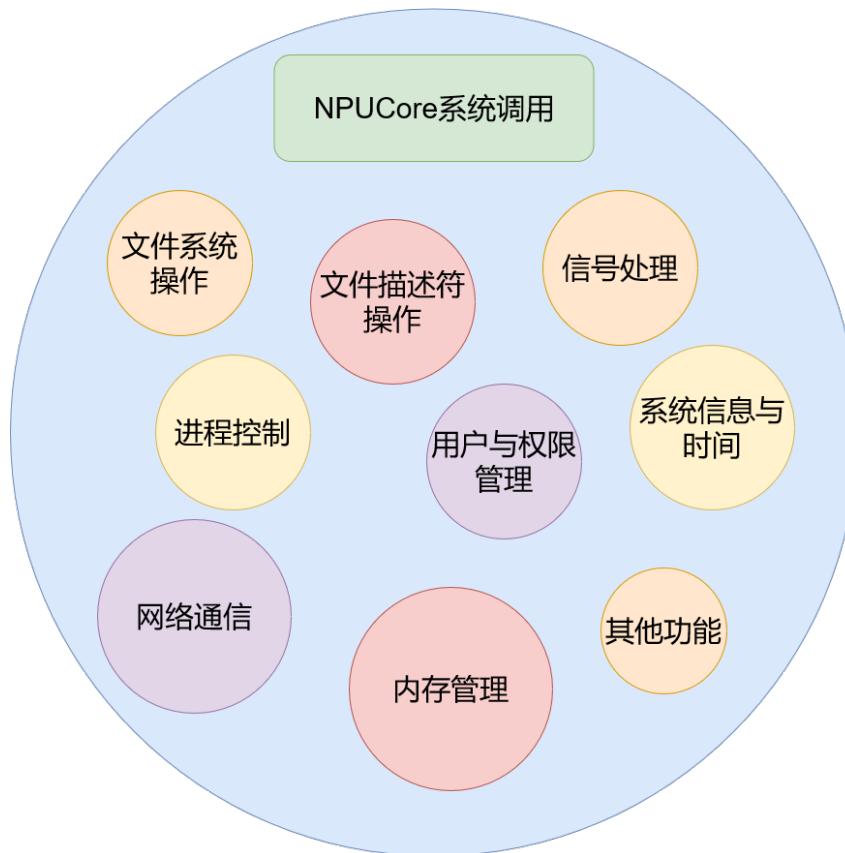


图 9-8 BusyBox 系统调用

### 9.3.3 系统调用分类功能介绍

#### 文件系统操作

文件系统操作涉及创建、打开、读取、写入、移动和删除文件或目录。

- **openat**: 在指定目录下打开一个文件。
- **close**: 关闭一个已打开的文件描述符。
- **read**: 从打开的文件中读取数据。
- **write**: 向打开的文件写入数据。
- **lseek**: 在打开的文件中移动读写位置。
- **unlinkat**: 删除一个文件或目录链接。
- **mkdirat**: 在指定目录下创建一个新目录。
- **renameat2**: 重命名或移动一个文件或目录。

#### 进程控制

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。

#### (1) 信号处理

信号处理包括设置信号的处理方式、控制信号的屏蔽以及从信号处理程序返回。

- **sigaction**: 设置信号的处理函数。
- **sigprocmask**: 用于阻塞或解除阻塞信号。
- **sigreturn**: 从信号处理程序返回。

#### (2) 网络通信

网络通信涉及套接字的创建、管理、绑定、监听及关闭。

- **socket**: 创建一个新的套接字。
- **bind**: 将套接字绑定到一个地址。
- **listen**: 监听套接字连接。
- **getsockname**: 获取套接字的本地地址信息。
- **getpeername**: 获取远程连接端的地址信息。
- **setsockopt**: 设置套接字选项。
- **shutdown**: 关闭套接字的一部分或全部连接。

#### (3) 文件描述符操作

文件描述符操作涉及复制文件描述符、修改其属性及创建管道。

- **dup**: 复制一个文件描述符。
- **dup3**: 复制文件描述符，提供额外的控制。
- **fcntl**: 对文件描述符进行各种控制操作。
- **pipe2**: 创建一个管道。

#### (4) 系统信息与时间

系统信息与时间包括获取系统和进程时间信息及生成系统日志。

- **uname**: 获取系统信息。
- **sysinfo**: 获取系统统计信息。
- **clock\_gettime**: 获取当前时间。
- **times**: 获取进程时间信息。
- **syslog**: 生成系统日志消息。

#### (5) 用户和权限管理

用户和权限管理涉及获取用户/组 ID、设置文件模式掩码及访问权限。

- **getuid**: 获取用户 ID。
- **geteuid**: 获取有效用户 ID。
- **getgid**: 获取组 ID。
- **getegid**: 获取有效组 ID。
- **umask**: 设置文件模式创建掩码。
- **faccessat**: 检查文件的访问权限。

## (6) 内存管理

内存管理包括数据段大小的调整、内存映射及内存保护。

- **brk**: 改变数据段的大小。
- **munmap**: 取消内存映射。
- **mmap**: 映射文件或设备到内存。
- **mprotect**: 改变内存区域的保护权限。

## (7) 其他功能

其他功能包括设备特定操作、线程管理等。

- **ioctl**: 设备特定的输入/输出操作。
- **setsockopt**: 设置套接字选项。
- **futex**: 用户空间线程的快速锁机制。
- **yield**: 使当前线程放弃处理器。
- **set\_tid\_address**: 设置线程 ID 地址。
- **prlimit**: 获取和设置资源限制。

## (8) 系统调用功能对照表

由于系统调用过多，我们在该部分仅用表格的形式将系统调用与其对应关系进行介绍。具体想要得到各个系统调用的细节，可以参考下一节中利用 Linux man page 命令得到。

### 9.3.4 获得系统调用相关信息

Posix 规范规定了一系列的系统调用，linux 提供了查看系统调用相关信息的命令，如下所示：

**man 2 open**

可以查看 open 系统调用的相关信息，如下所示：

调用	描述	调用	描述	调用	描述
dup	文件描述符复制	dup3	类似 dup 但可设置标志	getcwd	获取当前目录路径
fcntl	文件描述符操作	ioctl	设备控制	mkdirat	创建新目录
unlinkat	删除文件	linkat	创建硬链接	umount2	卸载文件系统
mount	挂载文件系统	faccessat	检查文件权限	chdir	更改目录
openat	打开文件	close	关闭文件描述符	pipe2	创建管道
getdents64	读取目录项	lseek	文件定位	read	读取数据
write	写入数据	readv	多缓冲区读取	writev	多缓冲区写入
sendfile	传输数据	ppoll	等待事件	readlinkat	读取链接
fstatat	文件状态	fstat	描述符状态	statfs	文件系统状态
ftruncate	截断文件	utimensat	设置时间戳	exit	终止进程
exit_group	终止进程组	set_tid_address	设置线程 ID	futex	锁定机制
setitimer	定时器	clock_gettime	获取时间	syslog	内核缓冲操作
yield	让出时间	kill	发送信号	tkill	向线程发送信号
sigaction	信号处理	sigprocmask	信号集更改	sigreturn	返回信号处理
times	进程时间	setpgid	设置进程组 ID	getpgid	获取进程组 ID
uname	系统信息	umask	文件模式掩码	getpid	获取进程 ID
getppid	获取父进程 ID	getuid	获取用户 ID	geteuid	获取有效用户 ID
getgid	获取组 ID	getegid	获取有效组 ID	gettid	获取线程 ID
sysinfo	系统信息	socket	创建套接字	bind	绑定地址
listen	监听连接	getsockname	套接字信息	getpeername	端点地址
setsockopt	套接字选项	brk	数据段分配	munmap	解除映射
clone	创建子进程	execve	执行程序	mmap	映射内存
mprotect	内存权限	wait4	等待状态变化	prlimit	资源限制
renameat2	重命名	shutdown	关闭连接		

表 9-2 NPUCore 系统调用及其功能

```

OPEN(2)                               System Calls Manual                  OPEN(2)

NAME
    open, openat - open or create a file for reading or writing

SYNOPSIS
    #include <fcntl.h>

    int
    open(const char *path, int oflag, ...);

    int
    openat(int fd, const char *path, int oflag, ...);

DESCRIPTION
    The file name specified by path is opened for reading and/or writing, as
    specified by the argument oflag; the file descriptor is returned to the
    calling process.

    The oflag argument may indicate that the file is to be created if it
    does not exist (by specifying the O_CREAT flag). In this case, open()
    and openat() require an additional argument mode_t mode; the file is
    created with mode mode as described in chmod(2) and modified by the

```

图 9-9 open 系统调用相关信息

每个 man 指令的手册页面都有类似的结构，由以下部分组成

- NAME: 命令的名称和简短描述

- SYNOPSIS: 命令的语法和选项初步描述了命令的基本用法
- DESCRIPTION: 命令的详细描述和用法
- OPTIONS: 命令的选项和参数
- EXAMPLES: 使用命令的示例
- FILES: 命令可能使用的文件
- SEE ALSO: 相关命令和手册页面
- BUGS: 已知的命令错误和限制
- AUTHOR: 命令的作者和贡献者

man 是系统的分页 (page) 手册，指定的 man 的页选项通常是指程序工具或函数名，程序将显示每一项找得到的相关手册页。如果指定了章节 (section)，man 将在指定章节中搜索。默认将按照预定的顺序查找所有可用的章节，并且只显示第一个页，即使在多个章节中都存在这个页。可以在??看到，在 man 页面的左上角，有一个 MAN(2)，这里的 2 就是章节，比较常用的章节号是 1、5、8，分别代表用户命令、配置文件和格式、系统管理命令。

通过在文档中找到我们想要的规范信息，我们可以知道实现系统调用的细节规范，如报错类型，返回类型，接口参数等。

接下来，我们以 open 系统调用为例，来看看如何实现系统调用。第一步，需要从 linux man page 中找到 open 系统调用的系统调用号和函数签名，并将其如??所示：

```
SYNOPSIS
    #include <fcntl.h>

    int
    open(const char *path, int oflag, ...);

    int
    openat(int fd, const char *path, int oflag, ...);
```

图 9-10 open 系统调用号和函数签名

第二步，分析该系统调用设计的模块，具体而言是需要实现的功能与内核的哪一个部分关联，例如内存映射、进程管理、进程通信、线程相关等。之后，分析具体实现的框架，也就是做那些操作，我们需要的函数接口是哪些，从而可以得出一个实现的伪代码。在 open 的例子中，我们可以得到如下的伪代码：

```
1 int open(const char *pathname, int flags, mode_t mode)
2 {
3     // 1. 检查参数是否合法
4     // 2. 从文件系统中找到文件
5     // 3. 根据flags和mode参数，检查文件是否可以打开
6     // 4. 为文件分配一个文件描述符
7     // 5. 返回文件描述符
8 }
```

最后，在代码实现上，关注手册中的细节，例如可能出现的错误信息，对于特殊情况的处理。如在 `open` 的例子中，可能出现的错误类型有很多，例如：

- `EACCES`: 文件不可访问
- `EEXIST`: 文件已经存在
- `EFAULT`: pathname 指向的内存空间不可访问

如此，我们就完整的实现了一个系统调用。类似的对于其他的系统调用，我们都可以类似的实现。

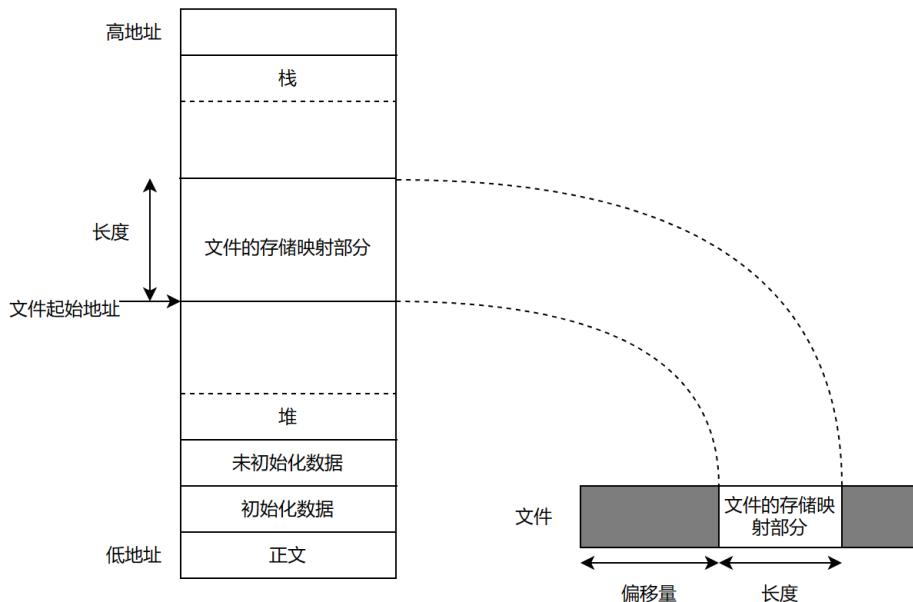
## 9.4 重要系统调用实现

### 9.4.1 mmap

#### (1) 内存映射

内存映射，简而言之就是将用户空间的一段内存区域映射到内核空间，映射成功后，用户对这段内存区域的修改可以直接反映到内核空间，同样，内核空间对这段区域的修改也直接反映用户空间。以下是一个把普遍文件映射到用户空间的内存区域的示意图。

图 9-11 内存映射



#### (2) mmap 主要用途

##### 传统的读写文件

一般来说，修改一个文件的内容需要如下 3 个步骤：把文件内容读入到内存中。修改内存中的内容。把内存的数据写入到文件中。

从传统读写文件的过程中，我们可以发现有个地方可以优化：如果可以直接在用户空间读写页缓存，那么就可以免去将页缓存的数据复制到用户空间缓冲区的过程。那么，有没有这样的技术能实现上面所说的方式呢？答案是肯定的，就是 `mmap`。

## 使用 mmap 读写文件

`mmap` 用于把文件映射到用户空间中，简单说 `mmap` 就是把一个文件的内容在内存里面做一个映像。那么对于内核空间与用户空间两者之间需要大量数据传输等操作的效率是非常高的。进程可以像读写内存一样对普通文件的操作。`mmap` 系统调用也可以使得进程之间通过映射同一个普通文件实现共享内存。

该函数主要用途有三个：

1. 将一个普通文件映射到内存中，通常在需要对文件进行频繁读写时使用，这样用内存读写取代 I/O 读写，以获得较高的性能；
2. 将特殊文件进行匿名内存映射，可以为关联进程提供共享内存空间；
3. 为无关联的进程提供共享内存空间，一般也是将一个普通文件映射到内存中。

### (3) `mmap` 实现

下面将结合 NPUcore 的代码来介绍 `mmap` 的实现

从上文可知，要实现 `mmap`，就要实现用户空间到内核空间的内存映射。所以 `mmap` 会分别确定要映射的用户空间和内存空间。

#### 用户空间

`mmap` 会申请一块适合的虚拟内存作为待映射的用户空间

代码片段 9.1 os/src/mm/memory\_set.rs

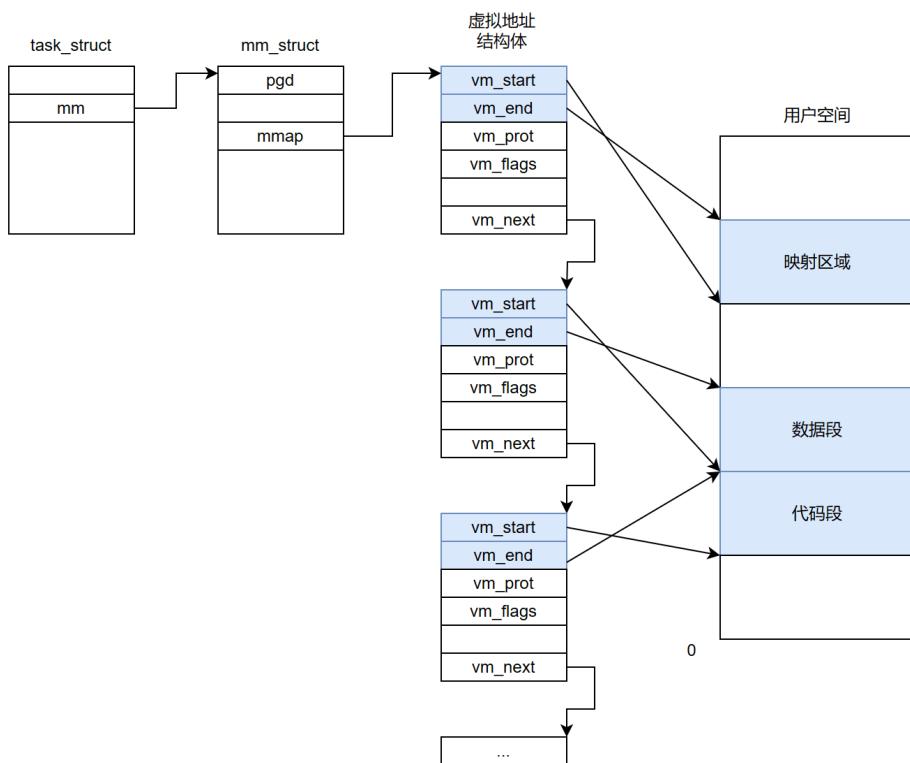
```

1 let area: &mut MapArea = &mut self.areas[idx];
2 let start_va = area.inner.vpn_range.get_end()
3 let end_va = start_va + len;
4
5 let mut new_area: MapArea = MapArea::new(
6     start_va,
7     end_va,
8     MapType::Framed,
9     map_perm,
10    map_file,
11 );

```

`MapArea` 是一个描述虚拟地址的结构体，它指向某块用户空间的首尾，`mmap` 在获取该结构体后会将其链入其进程管理的所有虚拟地址中。

图 9-12 虚拟地址结构体



## 内核空间

`mmap` 获取待映射的内核空间是通过进程打开的文件描述符获取的

代码片段 9.2 os/src/mm/memory\_set.rs

```

1 let fd_table = task.files.lock();
2 match fd_table.get_ref(fd) {
3     Ok(file_descriptor) => {
4         if !file_descriptor.readable() {
5             return EACCES;
6         }
7         let file = file_descriptor.file.deep_clone();
8         file.lseek(offset as isize, SeekWhence::SEEK_SET).unwrap();
9         new_area.map_file = Some(file);
10    }
11    Err(errno) => return errno,
12 }
```

NPUCore 中将文件对应的内核空间 `file` 存入了 `new_area.map_file` 中这样既确定了要映射的用户空间和内存空间，也完成了它们之间的映射。

## 其他

`mmap` 支持文件映射和匿名映射，如果是文件映射，则会通过上述的确定内核空间的过程，如果是匿名映射，那么该进程只是申请了一块空间，这片空间不与任何文件关联。

### 9.4.2 munmap

`munmap` 执行与 `mmap` 相反的操作，也就是删除用户空间与内核空间的映射。该函数会通过用户传入的地址信息删除页表中的有关映射，并将在 `mmap` 中新插入的 Maparea 删除。因大部分内容已在 `mmap` 介绍过，这里不做赘述。

### 9.4.3 execve

#### (1) execve 函数整体介绍

`execve` 用一个新的程序来代替当前进程的内存和寄存器，但是其文件描述符、进程 id 和父进程都是不变的。

它根据文件系统中保存的某个文件来初始化用户部分。`execve` 通过 `open()` 打开二进制文件。然后，它读取 ELF 头。应用程序以通行的 ELF 格式来描述。一个 ELF 二进制文件包括了一个 ELF 头，然后是连续几个程序段的头。

`exec` 会检查文件是否包含 ELF 二进制代码。一个 ELF 二进制文件是以 4 个“魔法数字”开头的，即 0x7F，“E”，“L”，“F”。如果 ELF 头中包含正确的魔法数字，`exec` 就会认为该二进制文件的结构是正确的。

代码片段 9.3 execve 输入参数与返回值

```

1 pub fn sys_execve(
2     pathname: *const u8,
3     mut argv: *const *const u8,
4     mut envp: *const *const u8,
5 ) -> isize

```

- `pathname` 要执行的新程序的文件路径。
- `argc` 参数数组，用于传递给新程序的命令行参数。
- `envp` 环境变量数组，用于设置新程序的环境变量。
- 返回值成功，返回 `SUCCESS`，否则，返回对应的错误码

#### (2) 整体流程

整个 `execve` 函数的整体流程图展示如下：

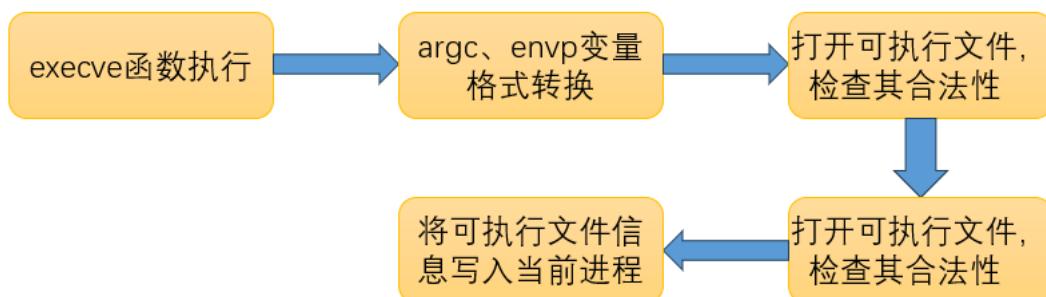


图 9-13 execve 流程图

### (3) 代码详析

下面将以 NPUCore 为例，讲述 execve 的实现。

1、函数的输入参数与返回值情况，已经在前面讲过，在此不过多赘述。

代码片段 9.4 execve 输入参数与返回值

```

1 pub fn sys_execve(
2     pathname: *const u8,
3     mut argv: *const *const u8,
4     mut envp: *const *const u8,
5 ) -> isize {
6     //load_elf 成功
7     SUCCESS
8     //失败
9     Err
10 }
```

2、将 argv 和 envp 变量字符串向量，通过循环，将 \*const \*const u8 转为 Vec<String> 数据类型。为后面的执行做准备。

代码片段 9.5 execve 参数转换

```

1 let mut argv_vec: Vec<String> = Vec::with_capacity(16);
2 let mut envp_vec: Vec<String> = Vec::with_capacity(16);
3 if !argv.is_null() {
4     loop {
5         let arg_ptr = match translated_ref(token, argv) {
6             Ok(argv) => *argv,
7             Err(errno) => return errno,
8         };
9         if arg_ptr.is_null() {
10             break;
11         }
12         argv_vec.push(match translated_str(token, arg_ptr) {
13             Ok(arg) => arg,
14             Err(errno) => return errno,
15         });
16         unsafe {
17             argv = argv.add(1);
18         }
19     }
20 }
21 if !envp.is_null() {
22     loop {
23         let env_ptr = match translated_ref(token, envp) {
24             Ok(envp) => *envp,
25             Err(errno) => return errno,
26         };
27         if env_ptr.is_null() {
28             break;
29         }
30         envp_vec.push(match translated_str(token, env_ptr) {
31             Ok(env) => env,
32             Err(errno) => return errno,
33         });
34         unsafe {
35             envp = envp.add(1);
36         }
37     }
38 }
```

```

36         }
37     }
38 }
```

### 3、debug 层信息输出

代码片段 9.6 execve 的 debug 信息

```

1 debug!(
2     "[exec] argv: {:+?} /* {} vars */, envp: {:+?} /* {} vars */",
3     argv_vec,
4     argv_vec.len(),
5     envp_vec,
6     envp_vec.len()
7 );
```

### 4、准备将 ELF 文件的内容载入当前进程，要对读取的文件做以下检查：

- 文件大小大于等于 4
- 同时，ELF 文件以 4 位魔数”7fELF”开头，因此应检查文件首是否有此魔数。

代码片段 9.7 对 elf 文件检查

```

1 match working_inode.open(&path, OpenFlags::O_RDONLY, false) {
2     Ok(file) => {
3         if file.get_size() < 4 {
4             return ENOEXEC;
5         }
6         let mut magic_number = Box::<[u8; 4]>::new([0; 4]);
7         // this operation may be expensive... I'm not sure
8         file.read(Some(&mut 0usize), magic_number.as_mut_slice());
9         let elf = match magic_number.as_slice() {
10             b"\x7fELF" => file,
11             b"#!" => {
12                 let shell_file = working_inode
13                     .open(DEFAULT_SHELL, OpenFlags::O_RDONLY, false)
14                     .unwrap();
15                 argv_vec.insert(0, DEFAULT_SHELL.to_string());
16                 shell_file
17             }
18             _ => return ENOEXEC,
19         };
20     }
21 }
```

### 5、真正的载入过程。loaf\_elf 将当前的 elf 文件内容覆盖当前的进程。show\_frame\_consumption! 宏输出对应的信息。

代码片段 9.8 execve 载入可执行文件

```

1 let task = current_task().unwrap();
2     show_frame_consumption! {
3         "load_elf";
4         if let Err(errno) = task.load_elf(elf, &argv_vec, &envp_vec) {
5             return errno;
6         };
7     }
8     // should return 0 in success
9     SUCCESS
```

```

10     }
11     Err(errno) => errno,
12 }

```

## 6、深入 load\_elf() 函数，了解 execve 的核心实现

可以很直观的发现,execve 调用的 elf 载入函数 load\_elf() 与 os::task::task::TaskControlBlock pub fn new(elf: FileDescriptor) -> Self 第一个进程初始化函数的整体内容是比较相似的,都包括

- 将 elf 文件映射到内存地址空间中
- 分配用户态资源
- 设置 tcb 信息

这些核心步骤。读者学习的时候不妨比照来阅读学习。

与 new 函数不同的是, load\_elf 会在原有的函数基础之上, 进行部分信息的覆写。首先, load\_elf 会覆写新进程的 trap\_cx 对应的内容, 并将其地址写入新进程的 trap\_cx\_ppn 中。

代码片段 9.9 load\_elf 信息覆写-1

```

1 // initialize trap_cx
2 let trap_cx = TrapContext::app_init_context(
3     if let Some(interp_entry) = elf_info.interp_entry {
4         interp_entry
5     } else {
6         elf_info.entry
7     },
8     user_sp,
9     KERNEL_SPACE.lock().token(),
10    self.kstack.get_top(),
11    trap_handler as usize,
12 );
13 // **** hold current PCB lock
14 let mut inner = self.acquire_inner_lock();
15 // update trap_cx_ppn
16 inner.trap_cx_ppn = (&memory_set)
17     .translate(VirtAddr::from(self.trap_cx_user_va()).into())
18     .unwrap()
19     .ppn();
20 *inner.get_trap_cx() = trap_cx;

```

## 第二, 刷新子进程 tid, 堆指针等信息

代码片段 9.10 load\_elf 信息覆写-2

```

1 // clear clear_child_tid
2 inner.clear_child_tid = 0;
3 // clear robust_list
4 inner.robust_list = RobustList::default();
5 // update heap pointers
6 inner.heap_bottom = program_break;
7 inner.heap_pt = program_break;

```

最后，刷新 tcb 的 elf、fd 表、memory\_set、futex 等信息，并返回载入成功的标识。

代码片段 9.11 load\_elf 信息覆写-3

```

1 // track the change of ELF file
2     *self.exe.lock() = elf;
3 // flush cloexec fd
4 self.files.lock().iter_mut().for_each(|fd| match fd {
5     Some(file) => {
6         if file.get_cloexec() {
7             *fd = None;
8         }
9     }
10    None => (),
11 });
12 // substitute memory_set
13 *self.vm.lock() = memory_set;
14 // flush signal handler
15 for sigact in self.sighand.lock().iter_mut() {
16     *sigact = None;
17 }
18 // flush futex
19 self.futex.lock().clear();
20 if self.tid_allocator.lock().get_allocated() > 1 {
21     let mut manager = TASK_MANAGER.lock();
22     // destroy all other threads
23     manager
24         .ready_queue
25         .retain(|task| (*task).tgid != (*self).tgid);
26     manager
27         .interruptible_queue
28         .retain(|task| (*task).tgid != (*self).tgid);
29 };
30 Ok(())

```

#### 9.4.4 openat

##### (1) openat 函数整体介绍

openat 函数用于打开文件。是 Linux 系统中用于打开文件的系统调用之一。它与 open 函数类似，但提供了更灵活和安全的文件路径处理方式。openat() 在很多方面类似于 open()，但它提供了更加灵活和安全的文件路径处理方式：

- 指定目录文件描述符：openat() 可以使用一个目录文件描述符作为起始位置，使得可以在指定目录下打开文件，而不受当前工作目录的影响。
- 相对路径打开：可以使用相对于指定目录文件描述符的相对路径打开文件，避免了对当前工作目录的依赖。

openat 的参数与返回值展示如下：

代码片段 9.12 openat 的参数与返回值

```

1 pub fn sys_openat(dirfd: usize, path: *const u8, flags: u32, mode: u32) ->
    isize

```

解释如下：

- dirfd 是一个打开的目录文件描述符，用于指定相对路径的起始位置。可以是当前工作目录的文件描述符，或者是先前通过 open() 或 openat() 打开的目录文件描述符。
- path 是要打开的文件的路径名，可以是绝对路径或相对于 dirfd 的相对路径。
- flags 包含文件打开的标志，比如 O\_RDONLY、O\_WRONLY、O\_CREAT、O\_TRUNC 等，控制文件的打开方式。
- mode 是文件的权限设置，仅当 O\_CREAT 被设置时才会生效，用于新建文件的权限设置。

### (2) 整体流程

流程图展示如下：

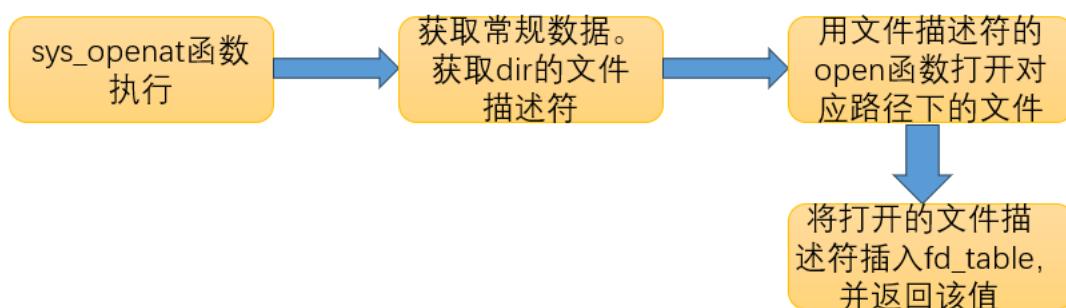


图 9-14 execve 流程图

### (3) 代码详析

1、函数输入参数展示如下

代码片段 9.13 openat 的参数与返回值

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

2、常规操作，获取 tcb 信息，token，path、mode 参数形式转换，获取 fd\_table

代码片段 9.14 openat 常规操作，获取 tcb 信息、token、path 参数形式转换

```
1 let task = current_task().unwrap();
2 let token = task.get_user_token();
3 let path = match translated_str(token, path) {
4     Ok(path) => path,
5     Err(errno) => return errno,
6 };
7 let flags = match OpenFlags::from_bits(flags) {
8     Some(flags) => flags,
9     None => {
10         warn!("[sys_openat] unknown flags");
11         return EINVAL;
12     }
13};
```

```

14 let mode = StatMode::from_bits(mode);
15 info!(
16     "[sys_openat] dirfd: {}, path: {}, flags: {:?}", mode,
17     dirfd as isize, path, flags, mode
18 );
19 let mut fd_table = task.files.lock();

```

3、接着，函数尝试从当前任务的文件表（fd\_table）中获取文件描述符。根据传入的 dirfd 参数，它会判断是否为特殊值 AT\_FDCWD（表示当前工作目录），若不是则尝试从文件表中获取对应的文件描述符。

代码片段 9.15 openat 获取目录文件描述符

```

1 let file_descriptor = match dirfd {
2     AT_FDCWD => task.fs.lock().working_inode.as_ref().clone(),
3     fd => {
4         let fd_table = task.files.lock();
5         match fd_table.get_ref(fd) {
6             Ok(file_descriptor) => file_descriptor.clone(),
7             Err(errno) => return errno,
8         }
9     }
10 };

```

4、根据获取到的文件描述符，调用其 open 方法打开传入的 path 路径的文件，使用传入的 flags 和 mode。如果打开成功，则将新的文件描述符加入到任务的文件表中，并返回该文件描述符作为结果。

代码片段 9.16 openat-打开文件，返回值设置为文件描述符的

```

1 let new_file_descriptor = match file_descriptor.open(&path, flags, false) {
2     Ok(file_descriptor) => file_descriptor,
3     Err(errno) => return errno,
4 };
5
6 let new_fd = match fd_table.insert(new_file_descriptor) {
7     Ok(fd) => fd,
8     Err(errno) => return errno,
9 };
10 new_fd as isize

```

5、深入文件描述符的 open 函数。总体来说，这段代码是文件描述符对象的 open 方法实现，用于处理文件系统中文件的打开操作。它检查路径是否有效，根据文件的类型进行不同的处理，最终尝试打开指定路径的文件或目录，并返回一个新的文件描述符对象或相应的错误信息。首先，该方法接受三个参数：path（文件路径）、flags（打开文件的标志）、special\_use（特殊用途标志）。如果传入的路径 path 为空字符串”，则直接返回当前文件描述符的克隆，即 Ok(self.clone())。接着，如果当前文件描述符对应的文件是一个文件（不是目录），且传入的 path 不是以斜杠 / 开头（即相对路径），则返回错误 ENOTDIR（表示不是一个目录）。之后，方法尝试从当前文件描述符对应的节点（即文件或目录）获取节点 inode。如果成功获取到 inode，则调用 inode 的 open 方法，尝试以

给定的 flags 和 special\_use 打开传入的路径 path。如果打开成功，则获取到一个新的文件描述符 file。根据传入的 flags 是否包含 O\_CLOEXEC 标志，设置 cloexec 变量，用于指示是否在 file 上设置了 O\_CLOEXEC (close-on-exec) 标志。最后，通过 Self::new() 创建一个新的文件描述符对象，返回一个包含新文件描述符的 Result。

代码片段 9.17 文件描述符的 open 函数

```

1 pub fn open(&self, path: &str, flags: OpenFlags, special_use: bool) ->
2     Result<Self, isize> {
3     if path == "" {
4         return Ok(self.clone());
5     }
6     if self.file.is_file() && !path.starts_with('/') {
7         return Err(ENOTDIR);
8     }
9     let inode = self.file.get_dirtree_node();
10    let inode = match inode {
11        Some(inode) => inode,
12        None => return Err(ENOENT),
13    };
14    let file = match inode.open(path, flags, special_use) {
15        Ok(file) => file,
16        Err(errno) => return Err(errno),
17    };
18    let cloexec = flags.contains(OpenFlags::O_CLOEXEC);
19    Ok(Self::new(cloexec, false, file))
}

```

补充: O\_CLOEXEC 是在打开文件描述符时设置的一个标志，它表示在执行 execve() 调用时，该文件描述符将被关闭，避免子进程继承该文件描述符。

具体来说，当一个进程（父进程）打开一个文件描述符并设置了 O\_CLOEXEC 标志后，如果在这个进程中调用 execve() 或 exec() 等系统调用来执行另一个程序，那么在新程序执行时，该文件描述符将会被自动关闭。这样可以确保新程序不会继承或意外使用父进程中打开的文件描述符，从而提高了安全性和可预测性。

这种行为通常用于父进程打开的文件描述符并不需要在子进程中保持打开的情况下。通过设置 O\_CLOEXEC 标志，可以避免不必要的文件描述符传递到子进程中，减少了资源泄漏和意外的文件操作可能性。

6、文件目录树节点的 open 方法详析这段 Rust 代码实现了 inode 对象的 open 方法，用于在文件系统中打开文件或目录。让我们逐步解析这段代码：

#### 日志记录和路径重定向：

代码开始处记录了调试日志，包括当前工作目录和传入的路径。接着定义了一些特殊路径和相应的重定向规则，例如将特定路径重定向到 BUSYBOX\_PATH 或 LIBC\_PATH 等。经过一系列的条件判断，根据特定规则将传入的 path 重新赋值，以便后续处理。

#### 定打开的起始节点 (inode)：

根据传入的 path 是否以 / 开头来确定打开文件的起始节点 inode，如果以 / 开头则从根节点（ROOT）开始，否则从当前节点（self）开始。

#### 路径缓存和文件系统操作：

- 使用互斥锁 PATH\_CACHE.lock() 获取路径缓存，如果存在缓存且路径匹配，则直接使用缓存的 inode。
- 否则，解析路径并逐级切换目录，获取到最终的目标 inode。
- 如果打开的文件不存在，根据标志位创建新的文件。
- 在文件存在的情况下，根据传入的标志位进行相应的操作，比如截断文件或检查文件状态。

#### 文件打开及特殊处理：

- 在文件打开之前，进行了一系列的检查，例如判断是否是目录、文件是否正在被使用、是否需要截断文件等。
- 根据 special\_use 参数进行特殊处理，如果需要特殊处理，则增加 spe\_usage 的计数值。
- 如果路径以 / 开头且与路径缓存不同，则更新路径缓存。

#### 返回结果：

最后调用文件对象的 open 方法打开文件，根据操作结果返回成功打开的文件或相应的错误码。

### 7、深入 OSInode 的 open 方法

#### 创建文件对象：

方法接受两个参数：flags（打开文件的标志）和 special\_use（特殊用途标志）。根据传入的标志位，设置文件对象的一些属性，比如 readable 表示文件是否可读、writable 表示文件是否可写、append 表示是否在末尾追加写入数据等。

#### 构建新的文件对象：

使用 Arc::new() 创建一个新的文件对象 Arc<dyn File>。在构建文件对象时，将文件对象的各种属性设置为对应的标志位状态或传入的参数状态，比如可读性、可写性、追加写入等。将原始文件对象的内部状态 inner 进行克隆，offset 采用 Mutex 进行多线程安全的偏移量管理。

最后将创建的对象返回

代码片段 9.18 OSInode 的 open 方法

```

1 fn open(&self, flags: OpenFlags, special_use: bool) -> Arc<dyn File> {
2     Arc::new(Self {
3         readable: flags.contains(OpenFlags::O_RDONLY) || flags.contains(
4             OpenFlags::O_RDWR),
5         writable: flags.contains(OpenFlags::O_WRONLY) || flags.contains(
6             OpenFlags::O_RDWR),
7         special_use,
8         append: flags.contains(OpenFlags::O_APPEND),
9     })
10 }
```

```

7     inner: self.inner.clone(),
8     offset: Mutex::new(0),
9     dirnode_ptr: self.dirnode_ptr.clone(),
10    })
11 }

```

### 9.4.5 fstat

**fstat** 主要功能在文件系统章节已经介绍，用于由文件描述符，获取文件当前状态。所以本小节将从 fstat 所用到的 **stat** 结构体，与利用 **fd** 查找文件两个角度出发，来解释 fstat 函数在操作系统文件部分中的实现和应用。

#### (1) 系统调用原型

代码片段 9.19 os/src/syscall/fs.rs

```

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

```

参数解释：

**fd**: 文件描述符 (file descriptor)，是一个非负整数，用于唯一标识打开的文件。  
**statbuf**: 一个指向 **stat** 结构的指针，用于接收文件详细信息 (指向内容如代码片段 9.20)。  
**返回值**: 执行成功则返回 **SUCCESS**，失败返回 **errno** 并跳入。成功和失败的判定是在于有没有通过 **fd** 找到文件，且返回相关的 **stat** 信息。

#### (2) fstat 流程

根据 fstat 系统调用的功能，我们可以写出 fstat 函数的流程与伪代码。需要注意的是，传入的 **fd** 可能本身不存在，所以为了提升鲁棒性，我们还需要判断 **fd** 是否为正，否则直接跳入 **errno**。例外：**fd** 为 **AT\_FDCWD** (-100)，表明当 **filename** 为相对路径的情况下，将当前进程的工作目录设置为起始路径。此时我们则需要对当前 **inode** 进行克隆，防止死锁。

**算法 9.1 fstat****Input:** fd, statbuf**Output:** SUCCESS or ERRNO

```

1: Let task = current task.
2: Let token = user token.
3: if fd < 0 or fd unmatched then
4:   return ERRNO
5: else
6:   if fd equal AT_FDCWD then
7:     lock and clone, copy stat to buf
8:   else
9:     copy stat to buf
10:  end if
11:  return SUCCESS
12: end if

```

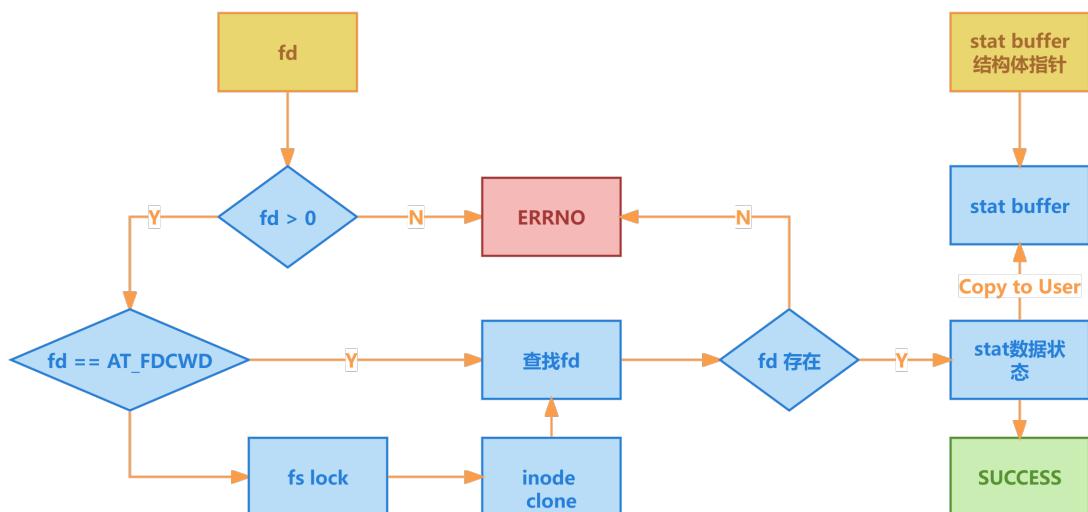


图 9-15 fstat 流程框图，黄色框代表输入，红色与绿色框分别是两种不同的输出，蓝色框为流程

这里我们也给出 stat 的结构体供大家参考。

代码片段 9.20 stat 示意结构体

```

1 struct stat {
2     dev_t      st_dev;          // 文件所在设备的设备号
3     ino_t      st_ino;          // 文件的i节点号
4     mode_t     st_mode;         // 文件的访问权限
5     nlink_t    st_nlink;        // 文件的硬链接数
6     uid_t      st_uid;          // 文件所有者的用户ID
7     gid_t      st_gid;          // 文件所有者的组ID
8     dev_t      st_rdev;         // 如果文件是特殊字符设备或块设备, 保存设备号
9     off_t      st_size;         // 文件的大小(常以字节为单位)
10    blksize_t  st_blksize;       // 文件I/O缓冲区大小
11    blkcnt_t   st_blocks;        // 分配给文件的块数
12    time_t     st_atime;         // 文件的最近访问时间
13    time_t     st_mtime;         // 文件的最近修改时间
14    time_t     st_ctime;         // 文件的最近更改时间(包括权限和归属人等)
15 };

```

### (3) fstat 代码详解

第一步，获取当前线程，老生常谈，不多介绍。

代码片段 9.21 获取线程

```
1 let task = current_task().unwrap();
2 let token = task.get_user_token();
```

第二步，搜索 fd 是否存在。本处省略了对 fd 大于 0 的判断。1. 首先，通过 match 语句对文件描述符进行匹配。2. 如果匹配到的文件描述符是 AT\_FDCWD，代表当前工作目录的文件描述符，那么将通过 task.fs.lock().working\_inode.as\_ref().clone() 获取到当前任务(task)的文件系统(fs)的锁，然后获取到工作 inode(working inode)的引用，并进行克隆(clone)得到一个新的 file\_descriptor。3. 如果匹配到的文件描述符不是 AT\_FDCWD，则会执行下面的代码块。4. 在下面的代码块中，首先获取到当前任务的文件表(fd\_table)的锁，并使用 get\_ref(fd) 方法获取到指定文件描述符的引用。5. 如果获取成功(Ok)，则克隆(clone)该文件描述符，并将克隆得到的新 file\_descriptor 赋值给 file\_descriptor 变量。如果获取失败(Err)，则会返回一个表示错误码(errno)的值。

代码片段 9.22 fd 查找

```
1 let file_descriptor = match fd {
2     AT_FDCWD => task.fs.lock().working_inode.as_ref().clone(),
3     fd => {
4         let fd_table = task.files.lock();
5         match fd_table.get_ref(fd) {
6             Ok(file_descriptor) => file_descriptor.clone(),
7             Err(errno) => return errno,
8         }
9     }
10};
```

第三步，将 fd 搜索到的 stat 信息，写入用户定义的 buffer 中。我们直接使用 copy\_to\_user 函数即可。函数结束，顺利返回 SUCCESS。

代码片段 9.23 赋值 stat buffer

```
1 copy_to_user(token, &file_descriptor.get_stat(), statbuf as *mut Stat);
2 return SUCCESS;
```

## 9.4.6 fstatat

前面的 stat 是通过 fd 去获得 stat 状态，而相比于 fstat，fstatat 可以通过路径名来定位文件，而不仅仅依赖文件描述符来获取文件信息。本小节将重点介绍 **fstatat** 系统调用相较于 **fstat** 不同的地方，因此会忽略部分与 fstat 相同或类似的算法部分。

### (1) 系统调用原型

代码片段 9.24 os/src/syscall/fs.rs

```
1 pub fn sys_fstatat(dirfd: usize, path: *const u8, buf: *mut u8, flags: u32);
```

参数解释：

dirfd：用于定位文件的目录文件描述符（file descriptor）。它可以是当前工作目录的文件描述符，也可以是其它目录的文件描述符。

path：文件的路径名。它可以是绝对路径或相对路径。

buf：一个指向 stat 结构的指针，用于接收文件详细信息。

flags：一些标记选项，控制 fstatat 的行为。

在 NPUcore 中，flags 有如下几种类型：

代码片段 9.25 NPUcore 中的 fstatat flags 选项

```
1 pub struct FstatatFlags: u32 {
2     const AT_EMPTY_PATH = 0x1000; \\ 允许空路径名（仅用于目录文件）
3     const AT_NO_AUTOMOUNT = 0x800; \\ 禁止系统自动挂载文件系统
4     const AT_SYMLINK_NOFOLLOW = 0x100; \\ 不跟随符号链接
5 }
```

## (2) fstatat 流程

fstatat 函数相较于 fstat 多了两个阶段。第一是根据 Flags 查找并获取当前的 fstatat 标记选项，第二是以只读模式打开 dirfd 所指向的 file\_descriptor，如果可以正常打开，再拷贝 stat 状态至 buffer。为此我们需要在 fstat 的流程图中添加部分流程框。

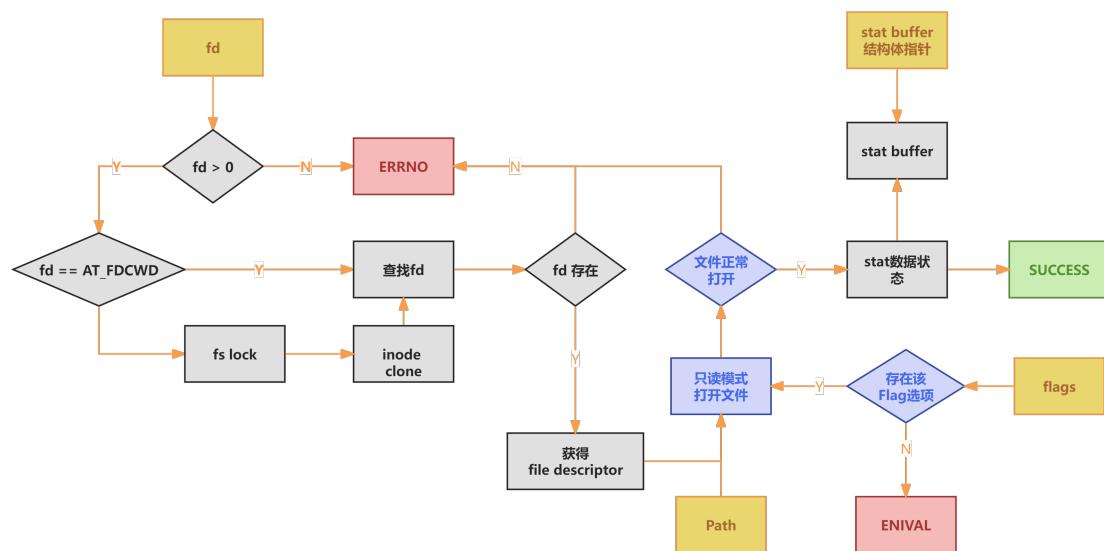


图 9-16 fstatat 流程框图，灰色框代表与 fstat 重复部分，黄色框代表输入，红色与绿色框分别是两种不同的输出，靛蓝色框代表与 fstat 不同的流程。

## (3) fstatat 代码详解

由于 fstatat 与 fstat 耦合代码较多，这里只涉及 fstatat 独有的部分代码。第一个不同是获取 Flag 部分。1. 首先，通过 match 语句将 flags 值转换为 FstatatFlags 类型的枚举

值。2. 将 FstatatFlags 的 flags 值转换为 FstatatFlags 枚举类型的标志位。如果转换成功 (Some(flags)), 则将转换后的枚举值赋给 flags 变量。如果转换失败 (None), 则执行下面的代码块。3. 在代码块中, 打印一个警告信息 (warn!("[sys\_fstatat] unknown flags")) 表示传入的 flags 值无法识别。最后返回一个表示无效参数 (EINVAL) 的错误码。

代码片段 9.26 FstatatFlag 判断

```

1 let flags = match FstatatFlags::from_bits(flags) {
2     Some(flags) => flags,
3     None => {
4         warn!("[sys_fstatat] unknown flags");
5         return EINVAL;
6     }
7 };

```

第二个不同是文件存在性验证部分。1. 首先, 通过 match 语句对 file\_descriptor 对象的 open 方法进行匹配。该方法使用给定的 path 路径、打开标志 (OpenFlags::O\_RDONLY 表示只读打开), 且不需要创建新文件。如果文件打开成功 (Ok), 则执行下面的代码块。2. 在代码块中, 首先通过 file\_descriptor.get\_stat() 方法获取到文件描述符对应文件的统计信息 (Stat 类型), 然后调用 copy\_to\_user 函数将 stat 状态复制到用户提供的 buf 指针指向的内存中, 最后返回一个表示成功的值 (SUCCESS)。3. 如果文件打开失败 (Err), 则执行 errno 代码块, 并返回一个表示错误码 (errno) 的值。

代码片段 9.27 存在性验证与 stat 拷贝

```

1 match file_descriptor.open(&path, OpenFlags::O_RDONLY, false) {
2     Ok(file_descriptor) => {
3         copy_to_user(token, &file_descriptor.get_stat(), buf as *mut
4             Stat);
5         SUCCESS
6     }
7     Err(errno) => errno,
}

```

## 9.4.7 write

### (1) write 函数整体介绍

write 函数通常用于将数据从用户空间写入文件或其他的输出设备, 如果文件不可写, 则返回相应的错误类型。write 的参数与返回值展示如下:

代码片段 9.28 write 的参数与返回值

```

1 pub fn sys_openat(dirfd: usize, path: *const u8, flags: u32, mode: u32) ->
    isize

```

解释如下:

- fd 表示文件描述符, 一般通过 open 函数的返回值获得, 对当前任务队列的活跃文件进行唯一标识。
- buf 表示写入数据的缓冲区的指针。

- count 表示要写入的字节数, 和 buf 参数一起可以表示写入数据的全部内容。

## (2) 整体流程

流程图展示如下:

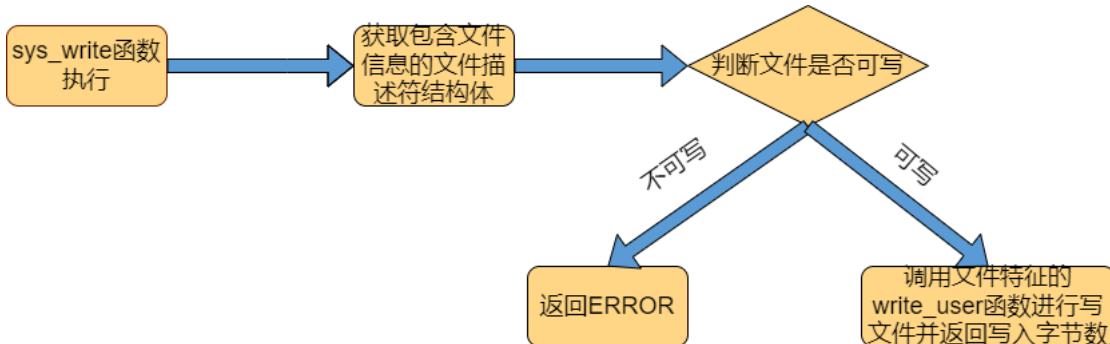


图 9-17 wtire 流程图

## (3) 代码详析

1、函数输入参数展示如下:

代码片段 9.29 write 的参数与返回值

```
1 pub fn sys_write(fd: usize, buf: usize, count: usize) -> isize
```

2、获取 TCB 信息并通过 get\_ref 函数和得到的文件描述符表 fd\_table, 将参数 fd 转化成包含文件信息的文件描述符结构体指针 file\_descriptor。

代码片段 9.30 含有文件信息文件描述符指针的获取

```
1 let task = current_task().unwrap();
2 let fd_table = task.files.lock();
3 let file_descriptor = match fd_table.get_ref(fd) {
4     Ok(file_descriptor) => file_descriptor,
5     Err(errno) => return errno,
6 };

```

3、判断文件是否可写, 如果不可写则返回 EBADF。(Bad file number, 值为-9)

代码片段 9.31 文件权限判断

```
1 if !file_descriptor.writable() {
2     return EBADF;
3 }
```

4、获取用户空间令牌, 用于下文缓冲区数据的获取。

代码片段 9.32 获取用户空间令牌

```
1 let token = task.get_user_token();
```

5、通过调用 write\_user 函数, 将用户空间数据写入文件描述符对应的文件中。 write\_user 有两个参数:

## 代码片段 9.33 write\_user 参数

```
1 fn write_user(&self, offset: Option<usize>, buf: UserBuffer) -> usize
```

- offset 表示用户指定的固定偏移量（相对于文件开始位置的偏移），不受文件指针偏移的影响。
- buf 表示一个 UserBuffer 结构体，其内部信息通过 translated\_byte\_buffer 函数获得。UserBuffer 结构体内容如下，包括一个表示数据具体内容的 Vec<&mut[u8]> 和表示长度的 len。

## 代码片段 9.34 UserBuffer 结构体

```
1 pub struct UserBuffer {
2     /// The segmented array, or, a "vector of vectors".
3     /// # Design Information
4     /// In Rust, reference lifetime is a must for this template.
5     /// The lifetime of buffers is static' because the buffer
6     /// USES A' instead of 'HAS A'
7     pub buffers: Vec<&'static mut [u8]>,
8     /// The total size of the Userbuffer.
9     pub len: usize,
10 }
```

translated\_byte\_buffer 函数接受三个参数，用户空间令牌 token，缓冲区指针 ptr 和缓冲区长度 len，返回值为一个 Result 包裹的 Vec<&mut[u8]> 变量。

## 代码片段 9.35 translated\_byte\_buffer 函数的参数和返回值

```
1 pub fn translated_byte_buffer(
2     token: usize,
3     ptr: *const u8,
4     len: usize,
5 ) -> Result<Vec<&'static mut [u8]>, isize>
```

6、深入文件描述符的 write\_user 函数，我们以 OSInode 的实现为例进行讲解。首先初始化写入总字节数变量 total\_write\_size 并获取文件的写锁。接下来分两种情况进行处理，如果偏移量不为 None，则通过循环遍历用户缓冲区，调用 self.inner.write\_at\_block\_cache\_lock 函数分次将数据写入文件，同时确保实际写入的字节数与数据片段长度相等，并维护偏移量 offset（指向下一个位置）和写入总字节数 total\_write\_size。

## 代码片段 9.36 偏移量不为 None

```
1 Some(mut offset) => {
2     let mut offset = &mut offset;
3     for slice in buf.buffers.iter() {
4         let write_size =
5             self.inner
6                 .write_at_block_cache_lock(&inode_lock, *offset, *slice);
7         assert_eq!(write_size, slice.len());
8         *offset += write_size;
9         total_write_size += write_size;
10    }
11 }
```

如果偏移量为 None，则需额外对 offset 进行处理，如果不是追加模式则设置 offset 为文件描述符的偏移量，如果为追加模式则将 offset 设置为文件的当前大小，其余的操作与 offset 不为 None 的情况相同。

代码片段 9.37 偏移量为 None 的情况

```

1 None => {
2     let mut offset = self.offset.lock();
3     if self.append {
4         *offset = self.inner.get_file_size_wlock(&inode_lock) as usize;
5     }
6     for slice in buf.buffers.iter() {
7         let write_size =
8             self.inner
9                 .write_at_block_cache_lock(&inode_lock, *offset, *slice);
10            assert_eq!(write_size, slice.len());
11            *offset += write_size;
12            total_write_size += write_size;
13     }
14 }
```

7、我们继续深入，来分析 write 的块级实现 write\_at\_block\_cache\_lock，它接受三个参数：

- inode\_lock 表示文件的写锁。
- offset 表示写数据位置的偏移量，即写数据位置相对文件开始位置的偏移。
- buf 表示具体要写入文件的数据，类型为 &[u8]。

write\_at\_block\_cache\_lock 函数首先通过获得的偏移量的信息来判断是否需要进行空间的扩展，如果 diff\_len 大于 0，则调用 modify\_size\_lock 扩展 diff\_len 大小的空间。(分配新簇)

代码片段 9.38 判断是否进行空间扩展分配

```

1 let mut start = offset;
2     let old_size = self.get_file_size() as usize;
3     let diff_len = buf.len() as  isize + offset as  isize - old_size as
4          isize;
5     if diff_len > 0 as  isize {
6         // allocate as many clusters as possible.
7         self.modify_size_lock(inode_lock, diff_len, false);
}
```

接着计算写入结束位置 end 并确保 start 小于 end，同时初始化写入的缓存块索引 start\_cache 和写入的总字节数 write\_size。

代码片段 9.39 其他变量初始化

```

1 let end = (offset + buf.len()).min(self.get_file_size() as usize);
2
3 debug_assert!(start <= end);
4
5 let mut start_cache = start / PageCacheManager::CACHE_SZ;
6 let mut write_size = 0;
```

最后循环处理并写入数据块缓存，每次循环维护当前数据块的结束位置 end\_current\_block，并通过 get\_cache 方法获取数据块的缓存，调用 modify 将数据从用户提供的缓冲区 buf 复制到数据块缓存，同时维护写入字节数和数据块的写入位置。

代码片段 9.40 循环处理数据块并写入

```

1 loop {
2     // calculate end of current block
3     let mut end_current_block =
4         (start / PageCacheManager::CACHE_SZ + 1) * PageCacheManager::
5             CACHE_SZ;
6     end_current_block = end_current_block.min(end);
7     // write and update write size
8     let lock = self.file_content.read();
9     let block_write_size = end_current_block - start;
10    self.file_cache_mgr
11        .get_cache(
12            start_cache,
13            || -> Vec<u8> { self.get_neighboring_sec(&lock.clus_list,
14                start_cache) },
15            &self.fs.block_device,
16        )
17        .lock()
18        // I know hardcoding 4096 in is bad, but I can't get around Rust's
19        // syntax checking...
20        .modify(0, |data_block: &mut [u8; 4096]| {
21            let src = &buf[write_size..write_size + block_write_size];
22            let dst = &mut data_block[start % PageCacheManager::CACHE_SZ
23                ..start % PageCacheManager::CACHE_SZ + block_write_size];
24            dst.copy_from_slice(src);
25        });
26    drop(lock);
27    write_size += block_write_size;
28    // move to next block
29    if end_current_block == end {
30        break;
31    }
32    start_cache += 1;
33    start = end_current_block;
34}

```

#### 9.4.8 read

read 函数的实现逻辑基本与 write 函数相同，只在对块缓存的处理上有不同之处，所以在这里对重复的内容不在赘述，只进行不同之处的讲解。

在 read 函数的块级操作中，与 write 的区别主要体现在 read\_at\_block\_cache\_rlock 方法的循环处理中。write 中循环处理是将用户缓存区写入块缓存中，但是在 read 中调用的是块级的.get\_cache().lock().read() 方法，是将块缓存的内容 (src) copy 到用户到的 buf (dst) 中，最后返回给用户以达到读文件的效果。除此之外，read 与 write 的另一不同之处在于 read 并没有追加模式。(write 有对于追加写的判断)

代码片段 9.41 read 与 write 实现的不同之处

```
1|.get_cache(
2|    start_cache,
3|    || -> Vec<usize> { self.get_neighboring_sec(&lock.clus_list,
4|        start_cache) },
5|    &self.fs.block_device,
6|)
7// I know hardcoding 4096 in is bad, but I can't get around Rust's syntax
8| // checking...
9|.read(0, |data_block: &[u8; 4096]| {
10|     let dst = &mut buf[read_size..read_size + block_read_size];
11|     let src = &data_block[start % PageCacheManager::CACHE_SZ
12|         ..start % PageCacheManager::CACHE_SZ + block_read_size];
13|     dst.copy_from_slice(src);
14});
```

## 第 10 章 内核性能评估

### 10.1 Lmbench 简介

Lmbench 是一个轻量级、可移植且符合 ANSI/C 标准的微型性能评估工具，专为 UNIX/POSIX 系统设计，旨在帮助系统开发者深入了解关键操作的基础成本。Lmbench 的主要目标是衡量系统性能的两个关键特征：反应时间和带宽。

**带宽方面的**测试主要涵盖了以下几个方面：

- 读取缓存文件 (`bw_file_rd`)

这项测试用于评估从硬盘到内存的文件读取速度，从而衡量文件读取时的性能。通过这个测试可以了解系统在读取缓存文件时的表现。

- 拷贝内存、读/写内存 (`bw_mem_*`, `bw_mmap_rd`)

`bw_mem_cp` (memory copy): 评估进程 CPU 与内存之间数据传输的速度，主要衡量内存之间数据拷贝的效率。

`bw_mem_rd` (memory reading and summing): 测试从内存读取数据的速度，通过读取数据并进行求和操作来评估内存读取的性能。

`bw_mem_wr` (memory writing): 评估向内存写入数据的速度，了解系统的内存写入性能。

`bw_mmap_rd` (memory mapping reading): 通过从硬盘到内存的映射并测试其读取速度，衡量内存映射文件读取的效率。

- 管道、TCP (`bw_pipe`, `bw_tcp`)

`bw_pipe`: 测试管道数据传输的速度，通常指定了传输数据的量和管道的大小。这个测试可以衡量管道通信的效率。

`bw_tcp`: 通过 TCP/IP 双向读取数据来评估系统的 TCP 通信性能。该测试可以提供关于 TCP 网络通信速度的信息。

**延时方面的**测试主要涵盖了以下几个方面：

- 上下文切换

上下文切换指的是在多任务系统中，从一个任务（或进程）切换到另一个任务时，保存和恢复进程状态所需的时间。Lmbench 能够测量不同条件下的上下文切换时间，比如 2p/0k、2p/16k 等参数。这些测试条件表示在不同的进程大小和数量下进行的上下文切换。通过这些测试，可以了解系统在不同负载下切换上下文所需的时间，即便无任务执行 (0k)，也能评估出纯粹的上下文切换成本。

- 网络

Lmbench 涵盖了不同类型的网络通信方式的评估，包括 TCP、UDP、RPC/TCP、RPC/UDP 等。这些测试项目评估了通过不同协议进行网络通信时所需的时间。

TCP 和 UDP 代表了常见的传输层协议，而 RPC 则是远程过程调用。这些测试提供了在不同网络通信方式下的延迟情况，有助于评估网络性能。

- 文件系统建立和删除

文件系统操作评估了文件操作的延迟，包括 0K/10K File Create/Delete。这些测试项目评估了在创建或删除不同大小的文件时所需的时间。通过这些测试，可以了解系统在文件系统操作上的性能表现，特别是针对文件的创建和删除操作。

- 进程创建

Lmbench 评估了创建新进程所需的时间，包括 fork proc、exec proc、sh proc 等测试项目。这些测试项目评估了创建新进程、执行新命令、使用系统 shell 来运行新程序等操作所需的时间。这些评估可帮助了解系统在处理进程创建时的效率。

- 信号处理

对信号的处理效率也进行了评估，比如 sig inst、sig hndl。这些测试评估了信号的安装和处理所需的时间。这对于了解系统处理信号及其响应的效率非常重要，尤其是在面对频繁信号触发的情况下。

- 上层系统调用

这部分测试涉及系统调用和相关操作的性能评估，比如 null call、null I/O、stat、open clos 等测试项目。这些测试项目评估了对系统调用和 IO 操作的响应速度。了解系统在这些操作上的性能可以帮助优化 IO 密集型任务或高频率系统调用的应用。

- 内存读入反应时间

这一部分评估了内存操作的延迟和性能，包括 lat\_pagefault、lat\_mem\_rd 等测试项目。这些测试评估了硬盘与内存之间的分页错误、CPU 与内存之间的读取延迟等。通过这些测试，可以了解系统在不同内存操作方面的性能表现。

**lmbench** 常见命令及其作用如下表所示：

针号	命令	作用
1	bw_file_rd	测试从硬盘到内存的文件读取速度
2	bw_mem_*	测试进程 CPU 与内存之间的传输速度（包括内存拷贝、读写、映射等）
3	lat_pagefault	测试硬盘与内存之间的分页错误延迟
4	lat_ctx	测试上下文切换所需的时间
5	lat_proc	测试进程创建的延迟
6	lat_unix	测试本地通信时延
7	lat_tcp/lat_udp/lat_unix_connect	测试网络连接时延
8	lat_fs/lat_select	测试文件系统操作的延迟
9	lat_syscall	测试系统调用的延迟
10	lat_mem_rd/lat_mmap	测试内存读取延迟和内存到硬盘的映射延迟

表 10-1 lmbench 常见命令及其作用

## 10.2 支持运行 Lmbench

方法	运行环境	对象	结果	原理
objdump+grep	Linux	二进制文件	测例的所有系统调用	参见 chapter09-03
strace	Debian+RISCV64	运行的进程	单个测例的系统调用	参见 chapter10-02

表 10-2 提取系统调用的方法

要从 lmbench 二进制文件中提取出相关的系统调用，过程基本与提取 busybox 的系统调用类似。仍然可以用一行命令完成。

```

1 # 从二进制文件中获取系统调用号，将objfile改为提取的目标二进制文件
2 objdump -d objfile | grep -B 9 ecall | grep "li.a7" | tee syscall.txt
3
4 # 如果报错：objdump: can't disassemble for architecture UNKNOWN!，是由
5 # 于当前的objdump并非RISC-V架构，尝试
6 riscv64-linux-gnu-objdump -d busybox | grep -B 9 ecall | grep "li.a7" |
7 tee syscall.txt

```

上述命令的详细解释，请参考第九章第三节，在此不再赘述。

而在实际情况中，还可能出现需要查看某条命令调用系统调用的情况，而反汇编并提取系统调用的方式，粒度过大，无法满足我们的要求。以 lmbench 为例，给出两种可能的查看系统调用的方法。

第一种，阅读 lmbench 的源代码。lmbench 的入口在 /testsuits-for-oskernel/lmbench/src 文件夹下的 lmbench\_all.c 中。这种方法极其繁琐复杂，不推荐。

我们推荐使用第二种方法，也就是利用 strace 工具捕捉程序运行时调用的系统调用。strace 是一个用于 Linux 的诊断、调试和指导用户空间实用程序。它用于监视和篡改进程与 Linux 内核之间的交互，包括系统调用、信号传递和进程状态的更改。

```

1 # 提取出一条指令使用的系统调用，并以摘要形式给出系统调用列表。注意执行
2 # 环境为QEMU (RISCV64+Debian)
3 strace -c -f -e trace=all -o trace_output.txt <命令>

```

例如，对于 lmbench 检测 write 系统调用性能的命令 lmbench\_all lat\_syscall -P 1 write，可以使用如下指令。

```

1 strace -c -f -e trace=all -o trace_output.txt ./lmbench_all
2 lat_syscall -P 1 write

```

执行结果如下，会给出系统调用的列表。

```

1 root@debian:~/20231228# cat trace_output.txt
2 % time    seconds   usecs/call      calls      errors   syscall
3 -----
4   58.11    0.044731     44731          1           0   wait4
5   28.03    0.021578       47         452           0   write
6    5.78    0.004450       13        332           0 getusage
7    3.72    0.002864     2864          1           0 execve
8    0.73    0.000562       43         13           0   close
9    0.72    0.000558       558          1           0 readlinkat

```

10	0.67	0.000515	85	6	pselect6
11	0.64	0.000492	30	16	getppid
12	0.42	0.000320	53	6	read
13	0.34	0.000258	32	8	rt_sigaction
14	0.26	0.000200	50	4	brk
15	0.19	0.000150	150	1	openat
16	0.18	0.000140	70	2	setitimer
17	0.16	0.000121	121	1	mprotect
18	0.05	0.000040	40	1	uname
19	0.00	0.000000	0	4	pipe2
20	0.00	0.000000	0	1	clone
21					-----
22	100.00	0.076979	90	850	total

针对无法编译得到 Lmbench 可执行文件的问题，我们可以利用大赛发布的 SD 卡镜像，将其挂载到我们的目录下，在此给出相关命令。

```

1  mkdir mnt
2  # 挂载到mnt下
3  sudo mount -o loop sdcard.img mnt
4  # 使用完之后，取消挂载
5  umount mnt

```

在本节中，我们通过比较得到 Lmbench 中存在 **17** 条完全不同于 Busybox 中的系统调用。

指令	系统调用	系统调用号
li a7,101	__NR_nanosleep	101
li a7,114	__NR_clock_getres	114
li a7,118	__NR_sched_setparam	118
li a7,119	__NR_sched_setscheduler	119
li a7,120	__NR_sched_getscheduler	120
li a7,121	__NR_sched_getparam	121
li a7,131	__NR_tkill	131
li a7,165	__NR_getrusage	165
li a7,202	__NR_accept	202
li a7,203	__NR_connect	203
li a7,206	__NR_sendto	206
li a7,207	__NR_recvfrom	207
li a7,211	__NR_sendmsg	211
li a7,212	__NR_recvmsg	212
li a7,227	__NR_msync	227
li a7,72	__NR_pselect6	72
li a7,82	__NR_fsync	82

表 10-3 Lmbench 系统调用

针对这些系统调用，我们将其分为了以下 **7** 种类型：

- 时间和定时器相关系统调用
- 进程调度和管理系统调用

- 信号和线程管理系统调用
- 资源和性能监测系统调用
- 网络编程相关系统调用
- 文件操作和数据同步系统调用
- 事件等待和选择系统调用

接下来，我们将分类为大家讲解各个类别的系统调用功能。

- 时间和定时器相关系统调用

- `_NR_nanosleep (101)`: 用于导致当前进程休眠指定的时间，通常用于实现定时器功能。
- `_NR_clock_getres (114)`: 用于获取指定时钟的分辨率（精度），可以用于测量时间间隔。

- 进程调度和管理系统调用

- `_NR_sched_setparam (118)`: 用于设置指定进程的调度参数，例如设置进程的优先级等。
- `_NR_sched_setscheduler (119)`: 用于设置指定进程的调度策略，例如设置为实时调度或普通调度。
- `_NR_sched_getscheduler (120)`: 用于获取指定进程的当前调度策略。
- `_NR_sched_getparam (121)`: 用于获取指定进程的调度参数，例如获取进程的优先级等。

- 信号和线程管理系统调用

- `_NR_tgkill (131)`: 用于向指定线程组（thread group）中的线程发送信号。

- 资源和性能监测系统调用

- `_NR_getrusage (165)`: 用于获取进程或子进程的资源使用情况，如 CPU 时间、内存使用等信息。

- 网络编程相关系统调用

- `_NR_accept (202)`: 用于在网络编程中接受来自客户端的连接请求。
- `_NR_connect (203)`: 用于在网络编程中建立与远程服务器的连接。
- `_NR_sendto (206)`: 用于在网络编程中向指定目标发送数据。
- `_NR_recvfrom (207)`: 用于在网络编程中从指定源接收数据。
- `_NR_sendmsg (211)`: 用于发送消息，通常用于进程间通信（IPC）。
- `_NR_recvmsg (212)`: 用于接收消息，通常用于进程间通信（IPC）。

- 文件操作和数据同步系统调用

- `_NR_msync (227)`: 用于将内存中的数据同步到文件中，通常用于文件映射。
- `_NR_fsync (82)`: 用于将指定文件的数据和元数据刷新到磁盘上，以确保数据持久化。

- 事件等待和选择系统调用

- `__NR_pselect6` (72): 用于在多个文件描述符上等待事件，并在事件发生时返回。

系统调用类别	系统调用	系统调用号
时间和定时器相关	<code>__NR_nanosleep</code> <code>__NR_clock_getres</code>	101 114
进程调度和管理	<code>__NR_sched_setparam</code> <code>__NR_sched_setscheduler</code> <code>__NR_sched_getscheduler</code> <code>__NR_sched_getparam</code>	118 119 120 121
信号和线程管理	<code>__NR_tgkill</code>	131
资源和性能监测	<code>__NR_getrusage</code>	165
网络编程相关	<code>__NR_accept</code> <code>__NR_connect</code> <code>__NR_sendto</code> <code>__NR_recvfrom</code> <code>__NR_sendmsg</code> <code>__NR_recvmsg</code>	202 203 206 207 211 212
文件操作和数据同步	<code>__NR_msync</code> <code>__NR_fsync</code>	227 82
事件等待和选择	<code>__NR_pselect6</code>	72

表 10-4 Lmbench 分类系统调用

## 10.3 多种性能指标测试评估

### 10.3.1 NPUcore 系统调用性能分析

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:

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 ("azbycxdwefugthsirjqkplomn")	0.017838240	0.022160122	0.022694000	0.022971000
b_string strstr ("aaaaaaaaaaaaaaaaaaaaacccccccccc")	0.011262640	0.013371076	0.014105000	0.014078000
b_string strstr ("aaaaaaaaaaaaaaaaaaaaaaaaaaac")	0.010942000	0.013494579	0.014031000	0.013603000
b_string strstr ("aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaac")	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 10-5 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 10-6 Unixbench 测试分数，“-”代表 PLNTRY 无法正常执行。

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，这点我们会在下一章详细分析。

### 10.3.2 其他内核系统调用性能分析

#### (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) 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。注意，所有的测试结果不直接在终端显示，而在 debug/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 RAIU 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统一保存成状态机，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

其结构图如下（引用于其文档）：

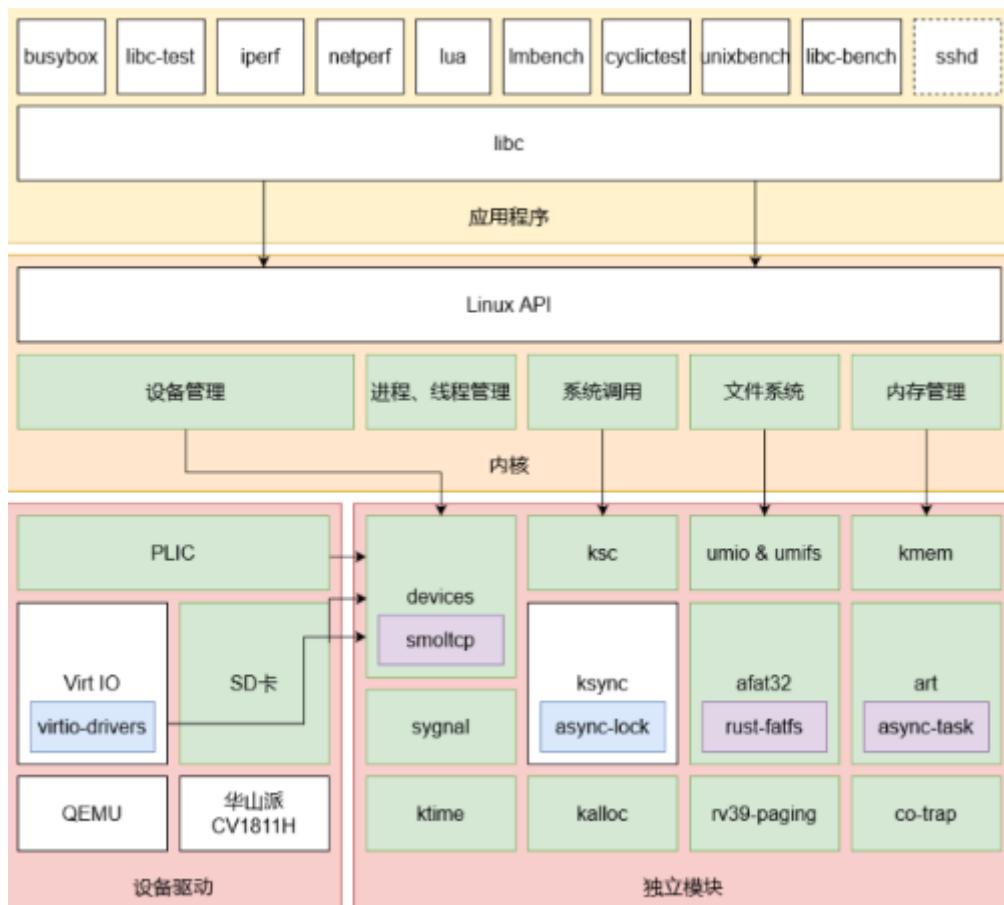


图 10-1 plntry 结构

## 10.4 内核优化与比较

### 10.4.1 时空优化 1——exec 系统调用优化

在前面的章节中，我们已经介绍了 exec 系统调用，其作用是替换当前进程的地址空间和上下文，使得该进程执行新程序。这个系统调用结合 fork 系统调用，可以创建新进程。大多数参赛队伍均实现了该功能，并且可以通过大赛的测试，首届大赛冠军 Ultra OS 便是例子。

笔者进行了 Ultra OS 的 exec 系统调用分析实验，主要方法是使用 exec 调用 10 次 busybox，在系统内插入采集时延的观测点，利用日志输出相应时延数据。最后得到图??的结果。

不难发现，读入 ELF 文件的时延占整个系统调用时长的 96.45%，已然成为 exec 最长的关键路径。如果不能减少该部分的时间开销，将会很大程度上影响该系统调用的性能。

笔者进一步分析 Ultra OS 中 exec 系统调用加载 elf 文件的过程，得到如图??的结果。

不难发现 elf 文件加载时延较大的原因是复制冗余，主要体现在：

- **一次调用多次复制。** elf 文件内容需要从块缓存先拷贝到内核栈中的 Elf buffer，再从 Elf Buffer 拷贝到用户空间，无形中增大了时延。

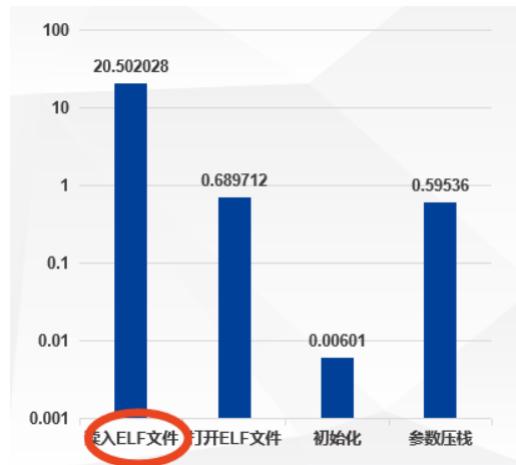


图 10-2 exec 系统调用分析实验结果

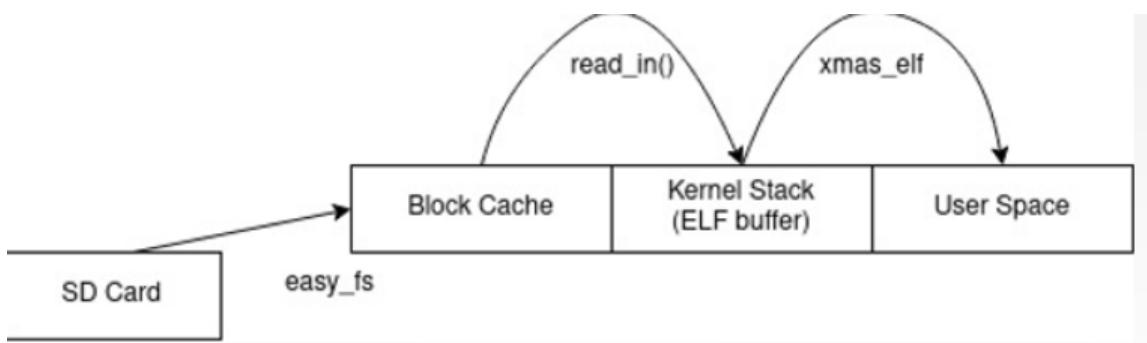


图 10-3 exec 系统调用加载 elf 文件过程

- **重复调用多次复制。**每次读取同一 elf 文件时，没有缓存，需要重新完场图中的过程，进一步增大时延。

为了解决该问题，NPUcore 使用了写时复制、elf caching、零拷贝的方法，成功降低了复制冗余的问题。在与 Ultra OS 的对比实验中，NPUcore 的时延均显著降低。

接下来详细介绍 NPUcore 对于 exec 系统调用的优化过程，读者可以通过图??辅助理解。

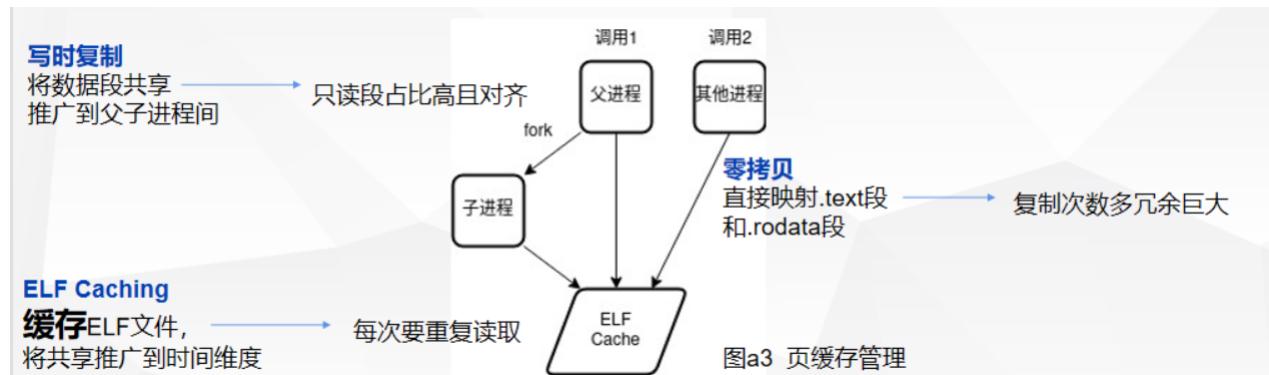


图 10-4 NPUcore 对于 exec 系统调用的创新处理

**优化目标** 由于 exec 系统调用的主要延迟在于冗余拷贝，NPUcore 的优化目标是减少拷贝，甚至做到多次变 0 次 Copy 或者只读文件。

**优化方法** 利用 RISC-V 页表共享和缺页中断的特性，可以实现“需要时”直接使用 ELF Cache 内的 ELF 文件内容，而不需要再多次或重复 Copy。需要时，指多次执行同一 elf 文件和父进程调用 folk 创建子进程时。

在以上优化思想的指导下，NPUcore 添加了以下功能：

**写时复制** 考虑到只读段内容占比高且对齐，NPUcore 将数据段共享推广到父子进程间。

**elf caching** 考虑到每次使用同一个 elf 文件时都要重复读取，NPUcore 通过实现缓存 ELF 文件，将共享推广到时间维度。

**零拷贝** 考虑到.text 段和.rodata 段映射过程中需要多次复制且冗余较大，NPUcore 在.text 段和.rodata 段使用的是直接映射的方式。

为了检验以上优化的效果，笔者设计实验比较时延性能对比。分别设置实现零拷贝，未实现零拷贝，Ultra OS 三组，分别运行 shell，exec，folk 三个程序，记录时延，得到图??的结果。

可以看出零拷贝的性能比不实现零拷贝的性能高出 1 倍还多，而 Ultra OS 的性能由于冗余拷贝的存在，性能远不及前两组。

#### 10.4.2 时空优化 2——文件系统缓存优化

在文件系统章节，笔者介绍过 PageCache 和 BufferCache，这些都是文件系统块缓存的部分，目的是为了缩小持久化设备和内存的读写速度差异。

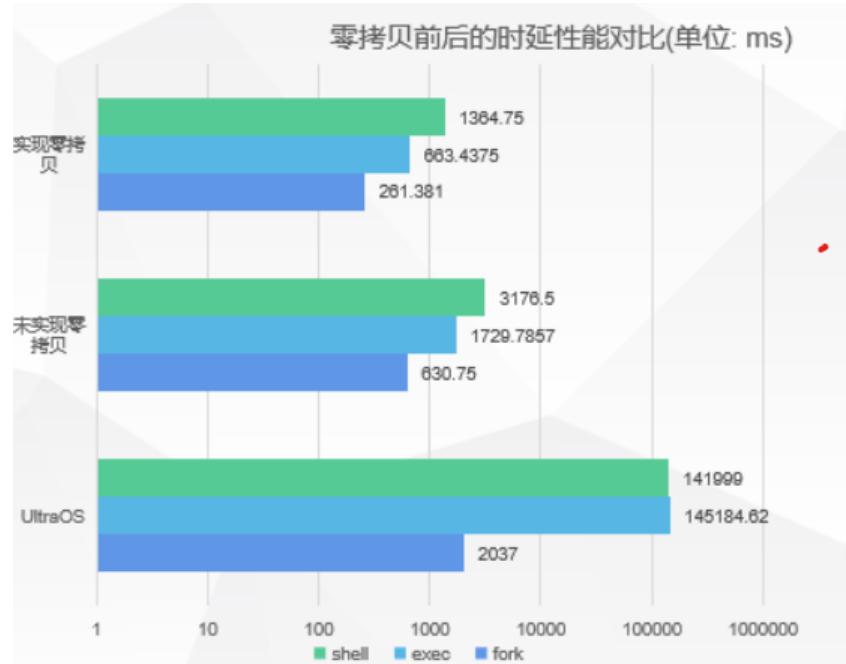


图 10-5 exec 系统调用优化效果

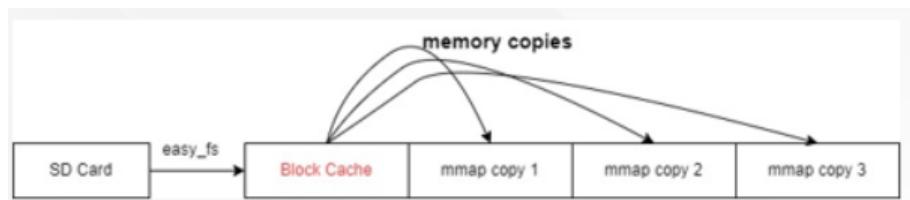


图 10-6 Ultra OS 中的文件缓存过程

事实上，许多操作系统，包括清华大学的 rCore 以及哈工大的 Ultra OS 均实现了该功能。但是，以 Ultra OS 为例，其块缓存 BlockCache 设计中存在一个问题：在 mmap 系统调用时，多个进程调用同一个文件时，需要将文件内容分别拷贝到相应的进程空间，这就导致了多次数据拷贝的问题，造成了时延和空间上的冗余。图?? 可以说明这一点。

NPUCore 基于此进行优化，调整了块缓存的读写策略，从而省去了拷贝的过程，从而减少了时延和空间冗余，详细过程如下。

**优化目标** 减少不必要的拷贝（多次拷贝变一次拷贝）

**优化思想** 仍是利用 RISC-V 页表共享和缺页中断的特性，多个用户程序可以仅通过 mmap 系统调用建立的映射关系直接读写 PageCache 中的数据。

简而言之，就是用地址映射的方式，使得进程可以直接访问 PageCache，从而省略了从 PageCache 拷贝数据的过程，图解释了该过程。

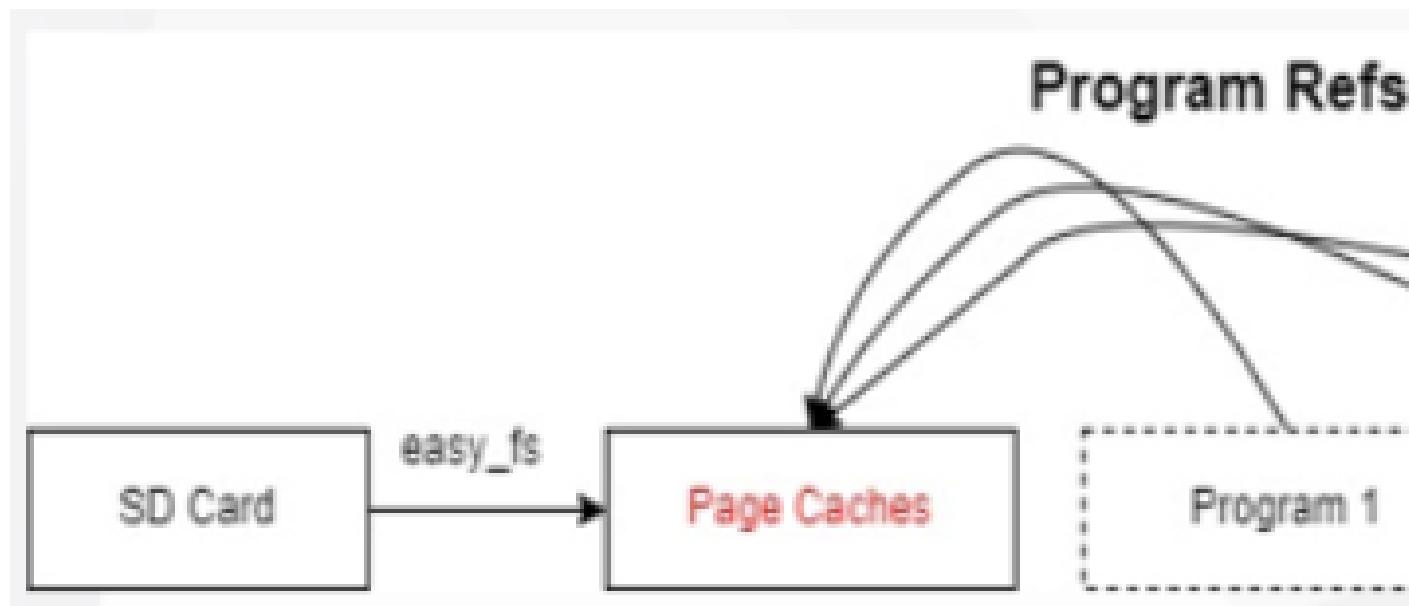


图 10-7 NPUCore 的文件缓存过程

**懒分配与写时复制** 图?? 解释了两种写入情况，写时复制和直接写入。NPUCore 使用了写入复制，当进程写入数据时复制 PageCache，将数据写入复制后的 PageCache。

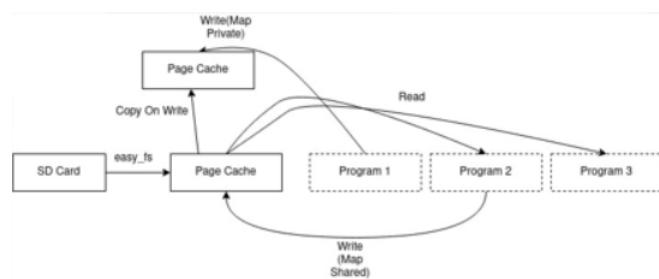


图 10-8 写时复制和直接写入

**页为单位** 块设备层的单位为块，而内存管理使用的单位为页。如果使用地址映射

的方法访问 cache 则会出现单位问题，这也是为什么 Ultra OS 只能拷贝的原因。NPUCore 这里设计了两个 Cache，一个 BufferCache 用于缓存块设备中的数据，而 PageCache 实现块向页的转化。此过程可参考图??。

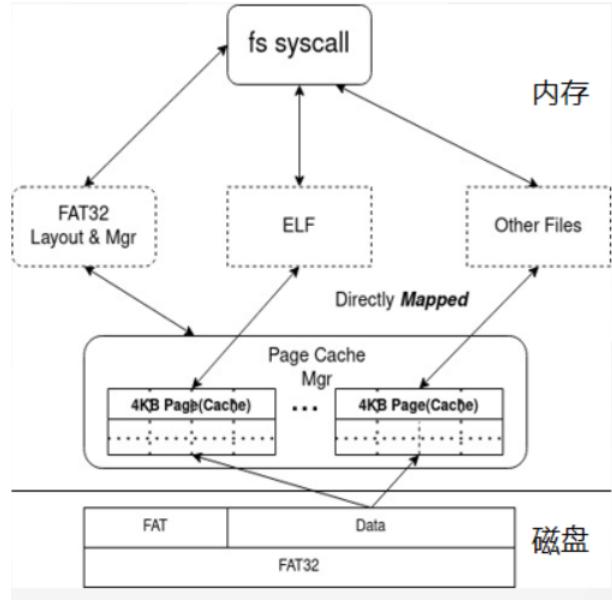


图 10-9 NPUCore 的 PageCache

**缓存与回收** 此部分与 Cache 相关。在文件系统中提到，整个内存地址空间均可以作为 Cache 的空间，所以只要空间足够就可以缓存数据，当内存已满时使用替换策略回收。

在实现了上述功能之后，笔者设计页缓存管理实验评估优化效果，分别使用旧缓存体系和新缓存体系分别测量 mmap 时延，分为 file\_rd io\_read, file\_rd open\_to\_close 和 mmap\_rd io\_rd.

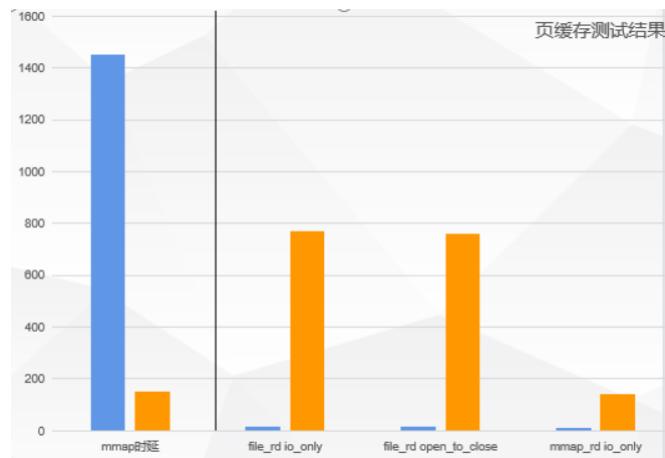


图 10-10 页缓存管理实验评估优化效果

实验结果如图??所示。从图中可以看出，mmap 的时延新缓存体系与旧缓存体系相比有了较大的降低，其他部分的数据却是旧缓存体系占优。由于多了一层 cache，所以

其他部分延迟的增加也是在意料之中。

#### 10.4.3 内存管理空间优化

##### 问题提出与分析

在多道程序运行时，存在内存空间不足的问题。具体而言如下：

1、如图??所示，测例本身的用户程序（ELF）占用的空间较大。

用户程序(ELF)大小		
bash	libc-test (含动态库)	lmbench
1.09MB	~1.5MB	1.04MB

图 10-11 测例 ELF 大小

2、为了考验 os 对于内存管理的性能，部分测例在运行时，需要占用大量内存，比如在运行 libc-test 中的 `sscanf_long` 测例，需要申请 8M 的空间逐字节读写；在执行 lmbench 工具集中的 `lat_ctx` 测试时，需要进行 32, 64, 96 进程上下文切换的测试，不言而喻，这个过程需要占用大量的内存。

3、与此同时，还要受限于 k210 并不宽裕的内存大小，如图??所示，k210 的内存空间已经捉襟见肘。

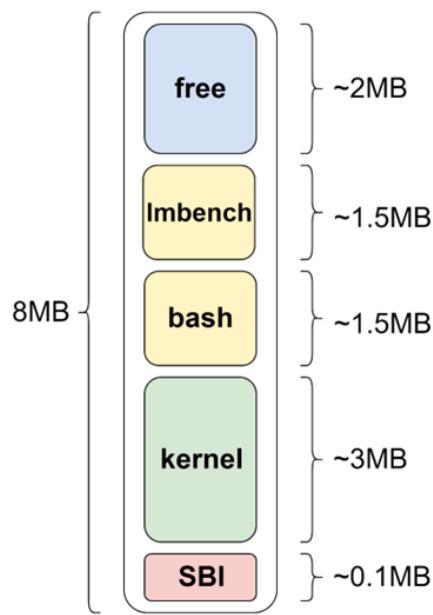


图 10-12 k210 内存使用情况

4、在前文所述的困难下，我们急需通过换出来腾空内存，从而实现内存的“扩容”。此时增加“换入换出”的功能似乎成为不二之选。但是在 NPUCore 之前，UltraOS 并没有实现这一功能（UltraOS 只有不完全的缺页处理和懒分配机制）。

5、如果采用换入换出的策略，又会遇到 IO 瓶颈，k210 上 SD 卡峰值读写速度约 1MB/s，而 libc-test 每个测例限时 5s，如果不对内存管理进行优化，在运行测例时很容易就出现超时。

在以上种种限制之下，迫切需要一套集成优化方法来支持内存“扩容”。

### 优化方法与评估

NPUcore 内存采用了一套“扩容”集成优化方法，包括以下策略：

- 懒分配（分配阶段）
- 缓存回收与写回（回收）
- zRAM 压缩内存（压缩）
- 虚拟内存换入换出（交换）

具体而言，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 on Write）。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??所示，是我们对相关 Fault 具体的处理方法：

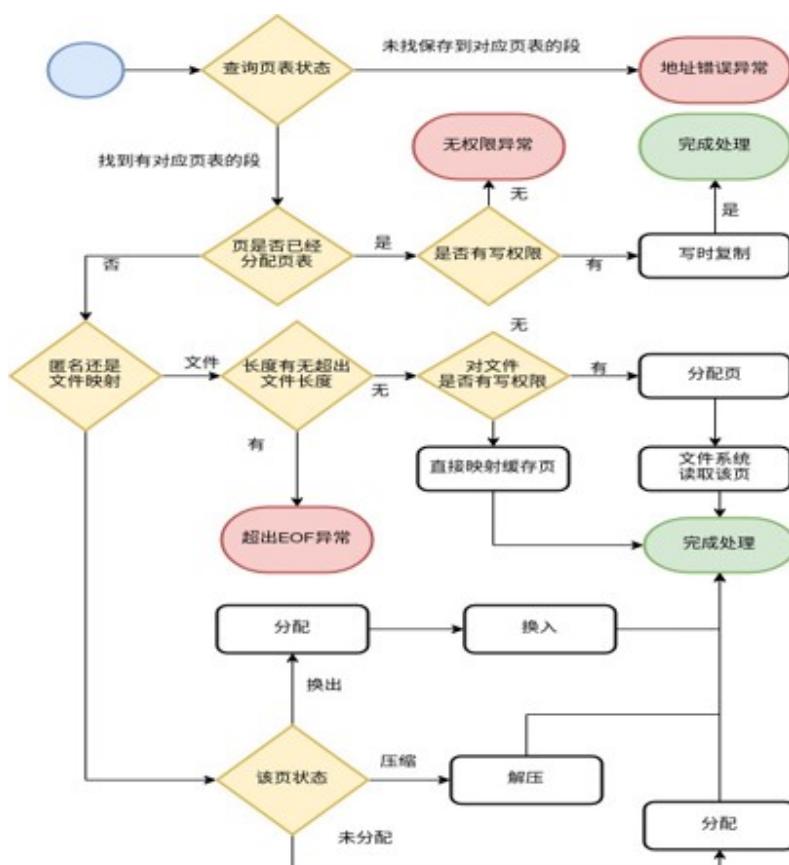


图 10-13 Page Fault 处理方法

首先是 Segment fault，对于原本就没有权限的和没有映射的地址，会在页表状态（实际通过内存中的“MapArea”（段）结构储存）中查找失败，然后直接进入无权异常/错地址异

常(上方的两个红终止框)。

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。如图??所示, 我们增加的内存管理方法, 极大提升内存回收峰值, 提高了操作系统的内存回收性能。

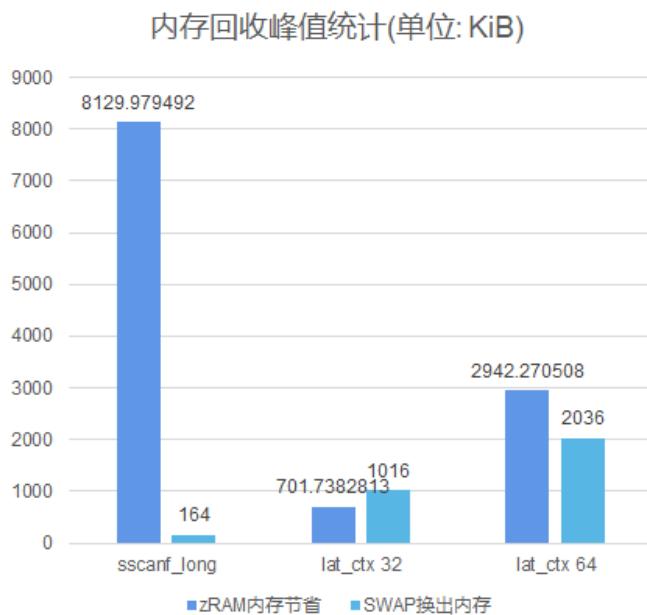


图 10-14 NPUCore 内存回收性能比较

#### 10.4.4 可靠性优化: SD 卡驱动优化

##### 、问题提出与分析

在运行 libc-test 测试, k210 板 SD 卡驱动在运行多个用户程序时, 内存占用率过高时会出现不稳定。如图??所示, rCore Tutorial 系统在压力测试下 SD 卡驱动读写异常。

我们经过初步分析, 发现是因为测例在运行时访问了 KPU 内存, 也就是高 2M 物

```

pid 3: [sdcard] read_sector(buf, 256032) error. Retrying...
pid 3: [sdcard] read_sector(buf[4096], 256032) error exceeded continuous retry count, waiting...
pid 3: [sdcard] read_sector(buf, 256032) error. Retrying...
pid 3: [sdcard] read_sector(buf[4096], 256032) error exceeded continuous retry count, waiting...
pid 3: [sdcard] read_sector(buf, 256032) error. Retrying...
pid 3: [sdcard] read_sector(buf[4096], 256032) error exceeded continuous retry count, waiting...
pid 3: [sdcard] read_sector(buf, 256032) error. Retrying...

```

图 10-15 压力测试下 SD 卡驱动读写异常

理内存。进一步分析，我们发现问题出在当内存占用率高时，CPU 会频繁访问 KPU 内存，由于 KPU 与 CPU 的时钟频率不同，因此会造成访存不稳定，进而影响 SD 卡读写的稳定性。

### 优化方法与评估

为了让 NPUcore 在读写 SD 失败后具有错误恢复的能力，我们对 SD 卡驱动程序进行了完善，以提高 NPUcore 的稳定性。整个 SD 卡驱动恢复过程的伪代码如下所示。

正如伪代码所示，在具体的实现上，NPUcore 的 SD 卡驱动恢复方法为：

首先会进行错误判断，如果 SD 卡驱动读写没完成则会导致失败（SD 卡驱动崩溃后驱动函数会返回错误信息）。

至于失败后的恢复过程，多数情况下，会进行 5 次尝试读写，如果全部失败（返回错误信息），则等待 1 秒，再 1 次尝试读写；如果再失败，则直接重新初始化整个驱动，再重复整个恢复过程，直到成功。

NPUcore 的 SD 卡驱动恢复方法，成效显著。为了验证其效果，使用下列命令：

```

1  while true; do
2      ./run-all.sh;
3  done

```

进行压力测试，如图??所示。最终，NPUcore 可以在 k210 开发板上连续正确运行 6 个小时！这项改进也被其他参赛队伍引用

---

**算法 10.1 SD 卡驱动恢复方法**

---

**Input:** block\_id, buf**Output:** None or SUCCESS

```

1: let result = self.write_sector(buf, block_id)
2: let cont_cnt = 0
3: while result is error do
4:   if cont_cnt >= 0 then
5:     log_error("[sdcard] write_sector(buf, block_id) error. Retrying...")
6:     result = self.write_sector(buf, block_id)
7:   end if
8:   cont_cnt = cont_cnt + 1
9:   if cont_cnt >= 5 then
10:    log_error("[sdcard] write_sector(buf[length(buf)], block_id) error exceeded continuous retry count, waiting...")
11:    self.wait_for_one_second
12:    if self.write_sector(buf, block_id) is error then
13:      self.init
14:      self.wait_for_one_second
15:    else
16:      break
17:    end if
18:    cont_cnt = 0
19:  end if
20: end while

```

---

```

pid 3: [sdcard] read_sector(buf, 256032) error. Retrying...
pid 3: [sdcard] read_sector(buf[4096], 256032) error exceeded continuous retry count, waiting...
pid 3: Waiting...
pid 3: Done.
pid 3: Waiting...
pid 3: Done.
pid 3: Waiting...
pid 3: [sdcard] read_sector(buf, 256032) error. Retrying...
pid 3: [sdcard] read_sector(buf[4096], 256032) error exceeded continuous retry count, waiting...
pid 3: Waiting...
pid 3: Done.
pid 3: Waiting...
pid 3: Done.
pid 3: Waiting...
pid 3: [sdcard] read_sector(buf, 256032) error. Retrying...
pid 3: [syscall] execve(221) -> ENOEXEC
pid 2: [page fault] pid: 2, type: Exception(StoreFault)
pid 2: [do_page_fault] addr: VA:0x6900e7cd, solution: copy on write
pid 2: [frame alloc] PPN:0x884b7
pid 2: [copy on write] copy occurred
pid 2: [sys sigprocmask] how: 2, set: BFFFE840, oldset: 0
pid 2: [sys sigprocmask] how: Some(SIG_SETMASK), *set: (empty)
pid 2: [sys sigprocmask] how: 0, set: BFFFEBC0, oldset: BFFFEE40
pid 2: [sys sigprocmask] *oldset: (empty)
pid 2: [sys sigprocmask] how: Some(SIG_BLOCK), *set: (SIGCHLD)
pid 3: [syscall] openat(56) args: [FFFFFFFFFFFFF9C, 6000C580, 8000, 0, 0, 0]
pid 3: [sys openat] dirfd: -100, path: ./run-static.sh, flags: O_LARGEFILE, mode: Some(empty)
pid 3: [open] cwd: /, path: ./run-static.sh
pid 3: [syscall] openat(56) -> 3
pid 3: [syscall] read(63) args: [3, BFFFE8E0, 80, 0, 0, 0]
pid 3: [frame alloc] PPN:0x884b8
pid 3: [copy on write] copy occurred

```

图 10-16 SD 卡驱动恢复方法效果实验评估