
rCore-Tutorial-Book-v3

发布 *3.6.0-alpha.1*

Yu Chen, Yifan Wu

2024 年 10 月 27 日

1 第零章：操作系统概述	1
1.1 引言	1
1.2 什么是操作系统	7
1.3 操作系统的系统调用接口	15
1.4 操作系统抽象	19
1.5 操作系统的特征	30
1.6 实验环境配置	32
1.7 练习	41
1.8 练习参考答案	42
2 第一章：应用程序与基本执行环境	47
2.1 引言	47
2.2 应用程序执行环境与平台支持	51
2.3 移除标准库依赖	57
2.4 内核第一条指令（基础篇）	62
2.5 内核第一条指令（实践篇）	68
2.6 为内核支持函数调用	75
2.7 基于 SBI 服务完成输出和关机	84
2.8 练习	88
2.9 练习参考答案	93
3 第二章：批处理系统	99
3.1 引言	99
3.2 特权级机制	105
3.3 实现应用程序	110
3.4 实现批处理操作系统	116
3.5 实现特权级的切换	122
3.6 练习	132
3.7 练习参考答案	135
4 第三章：多道程序与分时多任务	143
4.1 引言	143
4.2 多道程序放置与加载	152
4.3 任务切换	155
4.4 多道程序与协作式调度	160
4.5 分时多任务系统与抢占式调度	170
4.6 练习	175

4.7 练习参考答案	180
5 第四章：地址空间	195
5.1 引言	195
5.2 Rust 中的动态内存分配	201
5.3 地址空间	207
5.4 SV39 多级页表的硬件机制	213
5.5 管理 SV39 多级页表	224
5.6 内核与应用的地址空间	234
5.7 基于地址空间的分时多任务	247
5.8 超越物理内存的地址空间	261
5.9 练习	274
5.10 练习参考答案	277
6 第五章：进程	291
6.1 引言	291
6.2 进程概念及重要系统调用	299
6.3 进程管理的核心数据结构	308
6.4 进程管理机制的设计实现	319
6.5 进程调度	332
6.6 练习	349
6.7 练习参考答案	355
7 第六章：文件系统	361
7.1 引言	361
7.2 文件系统接口	369
7.3 简易文件系统 easy-fs	376
7.4 在内核中接入 easy-fs	407
7.5 练习	420
7.6 练习参考答案	424
8 第七章：进程间通信与 I/O 重定向	443
8.1 引言	443
8.2 基于文件的标准输入/输出	451
8.3 管道	455
8.4 命令行参数与标准 I/O 重定向	462
8.5 信号	472
8.6 练习	487
8.7 练习参考答案	490
9 第八章：并发	499
9.1 引言	499
9.2 用户态的线程管理	513
9.3 内核态的线程管理	520
9.4 互斥锁	543
9.5 信号量机制	571
9.6 条件变量机制	580
9.7 并发中的问题	589
9.8 练习	595
9.9 练习参考答案	599
10 第九章：I/O 设备管理	607
10.1 引言	607
10.2 I/O 设备	620
10.3 外设平台	634

10.4 串口驱动程序	642
10.5 virtio 设备驱动程序	646
10.6 virtio_blk 块设备驱动程序	659
10.7 virtio_gpu 设备驱动程序	667
10.8 练习	679
10.9 练习参考答案	681
11 综合练习	683
11.1 基础作业	683
11.2 拓展作业(可选)	685
12 附录 A: Rust 系统编程入门	689
12.1 Rust 编程相关	689
12.2 Rust 系统编程 pattern	689
13 附录 B: 常见工具的使用方法	691
13.1 调试工具的使用	691
13.2 分析可执行文件	693
13.3 qemu 平台上可执行文件和二进制镜像的生成流程	697
13.4 k210 平台上可执行文件和二进制镜像的生成流程	698
13.5 其他工具和文件格式说明的参考	699
14 附录 C: 深入机器模式: RustSBI	701
15 附录 D: RISC-V 相关信息	703
15.1 RISCV 汇编相关	703
15.2 RISCV 硬件相关	705
16 附录 E: 操作系统进一步介绍	707
16.1 openEuler 操作系统	707
17 附录 F: 类 Peterson 算法的局限性和内存顺序	709
17.1 参考文献	709
18 术语中英文对照表	711
18.1 第一章: RV64 裸机应用	711
18.2 第二章: 批处理系统	712
18.3 第三章: 多道程序与分时多任务	713
18.4 第四章: 地址空间	714
19 修改和构建本项目	717
20 reStructuredText 基本语法	719
21 更新日志	723
21.1 2022-12-14	723
21.2 2022-10-08	723
21.3 2022-10-02	723
21.4 2022-01-02	723
21.5 2021-11-20	724
21.6 2021-10-20	724
21.7 2021-03-15	724
21.8 2021-03-09	724
21.9 2021-03-07	724
21.10 2021-03-06	724
21.11 2021-03-05	724

21.12 2021-03-03	725
21.13 2021-02-28	725
21.14 2021-02-27	725
21.15 2021-02-20	725
21.16 2021-02-16	725
21.17 2021-02-13	726
21.18 2021-02-09	726
21.19 2021-02-08	726
21.20 2021-02-07	726
22 项目简介	727
23 导读	729
24 项目协作	731
25 本项目与其他系列项目的联系	733
26 项目进度	735

第零章：操作系统概述

1.1 引言

1.1.1 本章导读

本章主要解释了在已经有一系列优秀操作系统教材的情况下，为何要写本书。所以本章一开始就是分析学生目前学习操作系统碰到的困难和问题，并介绍如何参考操作系统历史，结合操作系统的完整实验来设计本书的各个章节来编写本书。接下来将从非常高层次的角度和计算机以及操作系统的发展史来进一步描述了什么是操作系统、操作系统的访问接口、操作系统的抽象、操作系统特征，让同学能够对操作系统有一个大致的整体把握。最后介绍了本书关联的操作系统实验环境（包括在线实验和本地实验等）的搭建过程，为后续开展各个操作系统实验打好基础。

1.1.2 为什么要写这本操作系统书

在目前的操作系统教学中，已有一系列优秀操作系统教材，例如 William Stallings 的《Operating Systems Internals and Design Principles》，Avi Silberschatz、Peter Baer Galvin 和 Greg Gagne 的《Operating System Concepts》，Remzi H. Arpaci-Dusseau 和 Andrea C. Arpaci-Dusseau 的《Operating Systems: Three Easy Pieces》等。

有待思考的问题

然而，从我们自 2000 年以来的教学实践来看，某些经典教材对操作系统的概念和原理很重视，但还有如下一些问题有待进一步思考：

- 原理与实践脱节：与操作系统的具体实现而言，操作系统的原理与概念相对过于抽象。目前的一些教材缺乏在“操作系统的原理与概念”和“操作系统的概念与实现”之间建立关联关系的桥梁，使得二者之间存在较大的鸿沟。这导致学生即使知道了操作系统的概念，还只能停留在“纸上谈兵”的阶段，依然不知如何实现一个操作系统。另外，学生在完成设计与实现操作系统的实验过程中，容易“一叶障目，不见泰山”，陷入到硬件规范、汇编代码、数据结构、编程优化等细节中，不知这些细节与操作系统概念的关系，缺少全局观和系统思维，难以与课堂上老师讲解的操作系统基本概念对应起来。

- 缺少历史发展的脉络：以史为鉴，可以知兴替。操作系统的概念和原理是从实际操作系统设计与实现的历史发展过程中，随着计算机硬件和应用需求的变化，从无到有逐步演进而产生的，有其发展的历史渊源和规律。但目前的大部分教材只提及当前主流操作系统的概念和原理，有“凭空出现”的感觉，学生并不知道这些内容出现的前因后果，只知道“**How**”，而不知道“**Why**”。而且操作系统发展史上的很多设计思路和实践方法起起伏伏，不断演进，它们并没有过时，而是以新的形态出现。如操作系统远古阶段的 **LibOS** 设计思路在当前云计算时代重新焕发青春，成为学术机构和各大互联网企业探索的新热点。
- 忽视硬件细节或用复杂硬件：很多教材忽视或抽象硬件细节，使得操作系统概念难以落地，学生了解不到软硬件是如何具体协同运行的。部分教材把复杂的 x86 处理器作为操作系统实验的硬件参考平台，缺乏对当前快速发展的 RISC-V 等精简体系结构的实验支持，使得学生在操作系统实验中可能需要花较大代价了解相对繁杂的 x86 硬件细节，编程容易产生缺陷 (bug)，影响操作系统实验的效果，以及对操作系统核心概念的掌握。

解决问题的思路

这些现存问题增加了学生学习和掌握操作系统的难度。我们尝试通过如下方法来解决上面三个问题，达到缓解学生的学习压力，提升学习兴趣，能在一个学期内比较好地掌握操作系统的目

具体而言，为应对“原理与实践脱节”的问题，我们强调 **实践先行，实践引领原理**的教学理念。MIT 教授 Frans Kaashoek 等师生设计实现了基于 UNIX v6 的 xv6 教学操作系统用于每年的本科操作系统课的实验中，并在课程讲解中把原理和实验结合起来，在国际上得到了广泛的认可，也给了我们很好的启发。经过十多年的操作系统教学工作，我们认为：对一位计算机专业的本科生而言，设计实现一个操作系统（包括 CPU）有挑战但可行，前提是这样的操作系统要简洁小巧，能体现操作系统中最基本的核心思想，并能把操作系统各主要部分的原理与概念关联起来，形成一个整体。而且还需要丰富的配套资源，比如对操作系统的整体框架、核心算法、关键组件之间的联系等的分析文档、配套的图示和视频讲解、能够自动测试操作系统功能的测试用例和测试环境、能展现操作系统逐步编写过程的在线源代码版本管理环境，以及逐步递进的综合性在线实验环境等，这样就能够让学生很方便地通过实践来加深对操作系统原理和概念的理解，并能让操作系统原理和概念落地。

为应对“缺少历史发展的脉络”的问题，我们重新设计操作系统实验和教学内容，按照操作系统的歷史发展过程来设立每一章的内容，每一章会围绕操作系统支持应用的某个核心目标来展开，形成相应的软硬件基本知识点和具体实践内容。同时建立与每章配套的多个逐步递进且相对独立的小实验，每个实验会形成一个独立的操作系统，体现了操作系统的一个微缩的发展历史，并可从中归纳总结出操作系统相关的概念与原理。这样可以在教学中引导学生理解操作系统的这些概念和原理是如何一步一步演进的。表面上看，这样会要求同学了解多个不同的操作系统，增加了同学的学习负担。但其实每个实验中的操作系统都是在前一个实验的操作系统上的渐进式扩展，同学只需理解差异的部分即可。而且学生通过分析不同操作系统对应用支持能力和对应实现上的差异，可以更加深入地理解相关操作系统概念与原理出现的前因后果。也许有同学认为讲解历史上的操作系统太过时了。但我们认为：技术可以过时，思想值得传承。

为应对“忽视硬件细节或用复杂硬件”的问题，我们在硬件（x86, ARM, MIPS, RISC-V 等）和编程语言（C, C++, Go, Rust 等）选择方面进行了多年尝试。在 2017 年把复杂 x86 架构换为简洁 RISC-V 架构，作为操作系统实验的硬件环境，降低了学生学习硬件细节的负担。在 2018 年引入 Rust 编程语言作为开发操作系统的可选编程语言之一，减少了用 C 语言编程出现较多运行时缺陷的情况。使得学生以相对较小的开发和调试代价进行操作系统实验。同时，我们把操作系统的概念和原理直接对应到程序代码、硬件规范和操作系统的实际执行中，加强学生对操作系统内核的实际体验和感受。

1.1.3 如何基于本书学习操作系统

前期准备

学习操作系统需要有一些前期准备，主要包括计算机科学基础知识，比如计算机组成原理、数据结构与算法、编程语言、软件开发环境等。具体而言，需要了解计算机的基本原理，特别是 RISC-V 处理器的指令集和部分特权操作；还有就是需要掌握基本的数据结构和算法，毕竟操作系统也是一种软件，需要通过多种数据结构和算法解决问题；在了解操作系统的设计并进行操作系统实验的过程中，需要掌握系统级的高级编程语言和汇编语言，比如 C 或 Rust 编程语言，RISC-V 汇编语言，这样才能深入理解操作系统的实现细节和设计思想；最后还需掌握操作系统的开发与实验环境，本书的主要涉及的开发与实验环境是 Linux，所以同学们需要能够通过 Linux 的命令行界面使用各种开发工具和辅助工具，而掌握基于图形界面或字符界面的 IDE 集成开发环境，如 VSCode、Vim、Emacs 等，可以提高分析操作系统源码，简化操作系统的开发与调试过程。

目标与步骤

所以本书的目标是以简洁的 RISC-V 基本架构为底层硬件基础，根据上层应用从小到大的需求，按 OS 发展的历史脉络，逐步讲解如何设计实现能满足“从简单到复杂”应用需求的多个“小”操作系统。并且在设计实现操作系统的过程中，逐步解析操作系统各种概念与原理的知识点，做到有“理”可循和有“码”可查，最终让同学通过操作系统设计与实现来深入地掌握操作系统的概念与原理。

在本书中，第零章是对操作系统的一个概述，让同学对操作系统的历史、定义、特征等概念上有一个大致的了解。后面的每个章节体现了操作系统的一个微缩的历史发展过程，即从对应用由简到繁的支持角度出发，每章会讲解如何设计一个可运行应用的操作系统，满足应用的阶段性需求。从而同学可以通过配套的操作系统设计实验，了解如何从一个微不足道的“小”操作系统，根据应用需求，添加或增强操作系统功能，逐步形成一个类似 UNIX 的相对完善的“小”操作系统。每一步都小到足以让人感觉到易于掌控。而在每一步结束时，你都能运行一个支持不同应用执行的“小”操作系统。

本书提供了哪些“小”操作系统？

我们按照操作系统的发展历史，设计了如下一些逐步进化的“小”操作系统

- LibOS: 让 APP 与 HW 隔离，简化应用访问硬件的难度和复杂性
 - BatchOS: 让 APP 与 OS 隔离，加强系统安全，提高执行效率
 - Multiprog & Timesharing OS: 让 APP 共享 CPU 资源
 - Address Space OS: 隔离 APP 访问的内存地址空间，限制 APP 之间的互相干涉，提高安全性
 - Process OS: 支持 APP 动态创建新进程，增强进程管理和资源管理能力
 - Filesystem OS: 支持 APP 对数据的持久保存
 - IPC OS: 支持多个 APP 进程间数据交互与事件通知
 - Thread & Coroutine OS: 支持线程和协程 APP，简化切换与数据共享
 - SyncMutex OS: 在多线程 APP 中支持对共享资源的同步互斥访问
 - Device OS: 提高 APP 的 I/O 效率和人机交互能力，支持基于外设中断的串口/块设备/键盘/鼠标/显示设备
-

另外，通过足够详尽的测试程序和自动测试框架，可以随时验证同学实现的操作系统在每次更新后是否正常工作。由于实验的代码规模和实现复杂度在一个逐步递增的可控范围内，同学可以结合对应操作系统实验的原理/概念分析，来建立操作系统概念原理和实际实现的对应关系，从而能够通过操作系统实验的实践过程来加强对理论概念的理解，并通过理论概念来进一步指导操作系统实验的实现与改进。

如何学习操作系统？

这取决于你想学习操作系统的目标，这里主要分为两类：

- 掌握基本原理为主，了解具体实现为辅（一般学习）
 - 理解式学习方式：逐章阅读与实践，阅读分析应用，并通过分析应用与 OS 的动态执行过程，掌握 OS 原理。
 - 掌握操作系统实现和原理为主（深入学习）
 - 构造式学习：在理解式学习方式基础上，进一步分析源码，逐步深入了解每个 OS 的内部增量实现，并且参考并基于这些小 OS，扩展部分 OS 功能，通过测试用例，从而同时掌握操作系统实现和原理。
-

编程语言与硬件环境

在你开始阅读与实践本书讲解的内容之前，你需要决定用什么编程语言来完成操作系统实验。你可以选择你喜欢的编程语言和在你喜欢的 CPU 上来实现操作系统。我们推荐的编程语言和架构分别是 Rust 和 RISC-V。

编程语言与指令集选择

目前常见的操作系统内核都是基于 C 语言的，为何要推荐 Rust 语言？

- 事实上，C 语言就是为写 UNIX 而诞生的。Dennis Ritchie 和 Ken Thompson 没有期望设计一种新语言能帮助高效地开发复杂与并发的操作系统逻辑（面向未来），而是希望用一种简洁的方式来代替难以使用的汇编语言抽象出计算机的行为，便于编写控制计算机硬件的操作系统（符合当时实际情况）。
- C 语言的指针既是天使又是魔鬼。它灵活且易于使用，但语言本身几乎不保证安全性，且缺少有效的并发支持。这导致内存和并发漏洞成为当前基于 C 语言的主流操作系统的噩梦。
- Rust 语言具有与 C 一样的硬件控制能力，且大大强化了安全编程和抽象编程能力。从某种角度上看，新出现的 Rust 语言的核心目标是解决 C 的短板，取代 C。所以用 Rust 写 OS 具有很好的开发和运行体验。
- 用 Rust 写 OS 的代价仅仅是学会用 Rust 编程。

目前常见的指令集架构是 x86 和 ARM，为何要推荐 RISC-V？

- 目前为止最常见的指令集架构是 x86 和 ARM，它们已广泛应用在服务器、台式机、移动终端和很多嵌入式系统中。由于它们的通用性和向后兼容性需求，需要支持非常多（包括几十年前实现）的软件系统和应用需求，导致这些指令集架构越来越复杂。
 - x86 后向兼容的策略确保了它在桌面和服务器领域的江湖地位，但导致其丢不掉很多已经比较过时的硬件设计，让操作系统通过冗余的代码来适配各种新老硬件特征。
 - x86 和 ARM 在商业上都很成功，其广泛使用使得其 CPU 硬件逻辑越来越复杂，且不够开放，不能改变，不是开源的，难以让感兴趣探索硬件的学生了解硬件细节，在某种程度上让 CPU 成为了一个黑盒子，并使得操作系统与硬件的交互变得不那么透明，增加了学习操作系统的负担。
 - 从某种角度上看，新出现的 RISC-V 的核心目标是灵活适应未来的 AIoT（人工智能物联网, AI + IoT）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深入 CPU 的运行细节，甚至可以方便地设计一个 RISC-V CPU。从而可帮助学生深入了解操作系统与硬件的协同执行过程。
 - 编写面向 RISC-V 的 OS 的硬件学习代价仅仅是了解 RISC-V 的 Supervisor 特权模式，知道 OS 在 Supervisor 特权模式下的控制能力。
-

1.1.4 本书章节导引

本书由 0~9 共 10 章组成，其中第 0 章是本书的总览，介绍了为何写本书，概述了操作系统的简要发展历史，操作系统的定义，系统调用接口，操作系统的抽象表示和特征等，以及如何基于本书来学习操作系统。

第 1 章主要讲解了如何通过操作系统来解决应用和硬件隔离达到简化应用编程的问题。并详细讲述了如何设计和实现建立在裸机上的执行环境，如何编写可在裸机执行环境上运行的显示“Hello World”的应用程序。最终形成可运行在裸机上的寒武纪“三叶虫”操作系统—LibOS。这样学生能对应用程序和它所依赖的执行环境的抽象概念与具体实现有一个全面和深入的理解。

第 2 章主要讲解了如何通过操作系统来保障系统安全和多应用支持这两个核心问题。并详细讲述了应该如何设计应用程序，如何通过批处理方式支持多个程序的自动加载和运行，如何实现应用程序与操作系统在执行特权上的隔离。最终形成可运行多个应用程序的泥盆纪“邓氏鱼”操作系统—BatchOS。这样学生可以看到系统调用、特权级、批处理等概念在操作系统上的具体实现，并了解如何通过批处理方式提高系统的整体性能，如何通过特许权隔离来保护操作系统，如何实现跨特权级的系统调用等操作系统核心技术。

第 3 章主要讲解了如何在提高多程序运行的整体性能并保证多个程序运行的公平性这两个核心问题。并详细讲述了如何通过提前加载应用程序到内存来减少应用程序切换开销，如何通过应用程序之间的协作机制来支持程序主动放弃处理器并提高系统整体性能，如何通过基于硬件中断的抢占机制支持程序被动放弃处理器来保证不同程序对处理器资源使用的公平性，也进一步提高了应用对 I/O 事件的响应效率。最终形成了支持多道程序的二叠纪“锯齿螈”操作系统—MultiprogOS，支持协作机制的三叠纪“始初龙”操作系统—CoopOS，支持分时多任务的三叠纪“腔骨龙”操作系统—TimesharingOS。这样学生可以通过分析这些操作系统的设计与实现，提炼出任务、任务切换等操作系统的根本概念，对计算机硬件的中断处理机制、操作系统的分时共享等机制有更深入的理解。

第 4 章主要讲解了内存的安全隔离问题和高效使用问题。有限的物理内存是操作系统需要管理的一个重要资源，如何让运行在一台计算机上的多个应用程序得到无限大的内存空间，如何能够隔离运行应用能访问的内存空间并保证不同应用之间的内存安全是本章要重点解决的问题。为此需要了解计算机硬件中的页表和 TLB 机制，并通过操作系统在内存中构建面向自身和不同应用的页表，形成应用与应用之间、应用与操作系统之间的内存隔离，从而解决内存安全隔离问题。通过缺页异常和动态修改页表等技术，让当前运行的应用正在或即将访问的数据位于内存中，不常用的数据缓存放到存储设备（如硬盘等），形成分时复用内存的操作系统能力，即“虚存”能力。最终形成支持内存隔离的侏罗纪“头甲龙”操作系统—Address Space OS。学生通过分析操作系统的根本设计与实现，可以把地址空间这样的抽象概念和页表的具体设计建立起联系，掌握如何通过页表机制来实现地址空间。对任务切换中增加的地址空间切换机制也会有更深入的了解。能够理解虚存机制中的各种页面置换策略能否有效实现，以及如何具体实现。

第 5 章主要讲解了如何提高应用程序动态执行的灵活性和交互性的问题，即让开发者能够及时控制程序的创建、运行和退出的管理问题。在第 5 章之前，在操作系统整个执行过程中，应用程序是被动地被操作系统加载运行，开发者与操作系统之间没有交互，开发者与应用程序之间没有交互，应用程序不能控制其它应用的执行。这使得用户不能灵活地选择执行某个程序。这需要给用户提供一个灵活的应用程序（俗称 shell），形成用户与操作系统进行交互的命令行界面（Command Line Interface）。用户可以在一个 shell 程序中输入命令即可启动或杀死应用，或者监控系统的运行状况，使得开发者可以更加灵活地控制系统。这种新的用户需求需要重构操作系统的功能，让操作系统提供支持应用程序动态创建/销毁/等待/暂停等服务。这就在已有的任务抽象的基础上进一步新抽象：进程，用于表示和管理应用程序的整个执行过程。这样最终形成具备灵活强大的进程管理功能的白垩纪“伤齿龙”操作系统—Process OS。学生通过分析操作系统的根本设计与实现，可以把进程、进程调度、进程切换、进程状态、进程生命周期这样的抽象概念与操作系统实现中的进程控制块数据结构、进程相关系统调用功能、进程调度与进程切换函数的具体设计建立其联系，能够更加深入掌握进程这一操作系统的根本概念。

第 6 章主要讲解了如何让程序方便地访问存储设备上的数据的问题。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长久保存的数据放到存储设备上存起来，并在需要的时候能读到内存中进行处理。文件和文件系统的出现极大地简化了应用程序访问存储设备上数据的操作。第 6 章将设计并实现操作系统和核心模块，即一个简单的文件系统—easyfs，向上给应用程序提供了常规文件和目录文件两种抽象，并提供 `open`、`close`、`read`、`write` 四个系统调用用来读写文件中的数据，向下通过存储设备驱动程序对存储设备这种 I/O 外设物理资源进行管理。这样就形成了支持文件访问的“霸王龙”操作系统—Filesystem OS。学生通过分析操作系统的根本设计与实现，可以看到文件、文件系统这样的操作系统抽象如何通

过一个具体的文件系统—easyfs 来体现的。并可以看到并理解文件系统与进程管理、内存管理之间的紧密联系，从而支持应用程序便捷地对存储设备上的数据进行访问。

第 7 章主要讲解如何让不同的应用进行数据共享与合作的问题。在第 7 章之前，进程之间被操作系统彻底隔离了，导致进程之间无法方便地分享数据，不能一起协作。如果能让不同进程实现数据共享与交互，就能把不同程序的功能组合在一起，实现更加强大和灵活的复杂功能。第 7 章的核心目标就是让不同应用通过进程间通信的方式组合在一起运行。为此，将引入新的操作系统概念—管道（pipe），以支持进程间的 I/O 重定向功能，即让一个进程的输出成为另外一个进程的输入，从而让进程间能够有效地合作起来。这样管道其实也可以看成是一种特殊的内存文件，并可基于文件的操作来实现进程间的内存数据共享。除了数据共享机制，进程间也需要快捷的通知机制，这就引出了信号（Signal）事件通知机制，让进程能够及时的获得并处理来自其他进程或操作系统发的紧急通知。这样最终形成支持多个 APP 进程间数据交互与事件通知功能的白垩纪“迅猛龙”操作系统—IPC OS。学生通过分析操作系统的概念与实现，可以看到进程间的隔离和共享是可以同时做到的，并可进一步了解在进程的基础上如何通过管道机制来打破进程间建立的地址空间隔离，实现数据共享，以及如何通过信号机制打断进程的正常执行来及时响应相对紧急的事件，从而掌握多应用共享协同的操作系统机制。

第 8 章主要讲解如何提高多个应用并发执行的效率和如何保证能多个应用正确访问共享资源的问题。进程的地址空间隔离会带来管理上的运行时开销，比如 TLB 刷新、页表切换等。如果把一个进程内的多个可并行执行的任务通过一种更细粒度的方式让操作系统进行调度，那么就可以在进程内实现并发执行，且由于这些任务在进程内的地址空间中，不会带来页表切换等运行时开销。这里的任务就是线程（Thread）。线程间共享地址空间，使得它们访问共享资源更加方便，但如果处理不当，就可能出现资源访问冲突和竞争的问题。这就需要通过同步机制来协调进程或线程的执行顺序，并通过互斥机制来保证在同一时刻只有一个进程或线程可以访问共享资源，从而避免了资源冲突和竞争的问题。第 8 章在进程管理的基础上进行重构，设计实现了线程管理机制，形成了支持多线程 app 的达科塔盗龙 OS—ThreadOS；并进一步设计了支持线程同步互斥访问共享资源的锁机制、信号量机制和条件变量机制，最终形成了支持多线程 APP 同步互斥访问共享资源的白垩纪“慈母龙”操作系统—SyncMutex OS。学生通过分析操作系统的概念与实现，可以理解线程和进程的关系与区别，理解同步互斥机制的不同特征和运行机理，从而能够深入理解支持并发访问共享资源的同步互斥机制的原理和实现。

第 9 章主要讲解如何让应用便捷访问 I/O 设备并让应用有更多感知与交互能力的问题。计算机中的外设特征各异，如显卡、触摸屏、键盘、鼠标、网卡、声卡等。在第 9 章之前，同学们已经接触到了串口、时钟、和磁盘设备，使得应用程序能通过操作系统输入输出字符、访问时间、读写在磁盘上的数据，并通过时钟中断让操作系统具有了抢占式分时多任务调度的能力，但这仅仅覆盖了很小的一部分外设，而且在实践上对操作系统与外设的交互细节也涉及不多。操作系统需要对外设有更多的深入理解，才能有效地管理和访问外设，给应用提供丰富的感知与交互能力。在原理与概念方面，第 9 章简要分析了外设的发展历程，外设的数据传输方式。并进一步阐述操作系统如何对外设建立不同层次的抽象和不同 I/O 执行模型，以便于操作系统对外设的内部管理，应用程序对外设的高效便捷访问。在实践上，第 9 章分析了操作系统如何通过设备树（Device Tree）来解析出计算机中的外设信息，并重新设计了基于中断方式的串口驱动程序，涉及串口设备初始化和串口数据输入输出，以及改进进程/线程的调度机制，让等待串口输入或输出完成的进程/线程进入阻塞状态，从而提高系统整体执行效率。在第 9 章还进一步介绍了 QEMU 模拟的 virtio 设备架构，以及 virtio 设备驱动程序的主要功能；并对 virtio-blk 设备及其驱动程序，virtio-gpu 设备及其驱动程序进行了比较深入的分析。这样最终形成支持图形游戏 APP 并具备高效外设中断响应的侏罗纪侏罗猎龙操作系统—Device。学生通过分析操作系统的概念与实现，可以深入了解不同外设的特征，外设的 I/O 传输方式，不同层次的外设抽象概念和 I/O 执行模型，从而对操作系统如何有效管理不同类型的外设有一个相关完整的理解。

百闻不如一见，如果同学们通过读书和阅读代码能逐步地明确每一章要解决的应用需求和问题，渐进地了解每章操作系统中内核模块的组成，并掌握内核模块的功能，以及不同内核模块之间的关系，能归纳总结出操作系统的概念与实现，就能达到了解操作系统的层次。百见不如一干，仅仅看还是不够的，本书的重要目标是希望能推动同学们能够通过编程来掌握操作系统。如果同学们还能通过课后习题和编程实验来完成操作系统的功能，发现编程中的 bug 并修复 bug，通过测试用例，实现你自己编写的操作系统，那将达到掌握操作系统的更高层次。希望同学们能够完整走完整个操作系统的概念与实现的过程，当你完成整个过程后，再回首看，能够发现原来操作系统还可以这样有趣和有用。

1.2 什么是操作系统

1.2.1 站在一米的代码空间维度看

一个操作系统（OS）是一个软件，它帮助用户和应用程序使用和管理计算机的资源。操作系统可能对最终用户不可见，但它控制着嵌入式设备、更通用的系统（如智能手机、台式计算机和服务器）以及巨型机等各种计算机系统。

今天，我们很难想象在没有操作系统的情况下使用计算机，它塑造和构建了我们访问计算机及其外围设备的交互方式。当第一批电子计算机在第二次世界大战后被开发出来时，还没有操作系统这种软件。在计算机诞生十年后，才首次出现某种操作系统的尝试。又过了十年，操作系统才被大家广泛接受。

我们的讨论将集中在通用操作系统上，因为它们需要的技术是嵌入式系统所需技术的超集，对操作系统原理、概念和技术的覆盖更加全面。现在的通用操作系统是一个复杂的系统软件，比如 Linux 操作系统达到了千万行的 C 源码量级。在学习操作系统的初期，如果去分析了解这样大规模的软件，要付出巨大的代价，因此我们对其进行简化，只讨论最基本的功能。

系统软件

系统软件是为计算机系统提供基本功能，并在计算机系统范围内使用的软件，其作用可涉及到整个计算机系统。系统软件包括操作系统内核、驱动程序、工具软件、用户界面、软件库等。操作系统内核是系统软件的核心部分，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。驱动程序是操作系统用于控制硬件设备的软件，如显卡驱动、声卡驱动和打印机驱动等。一般情况下，驱动程序是操作系统内核的一部分。

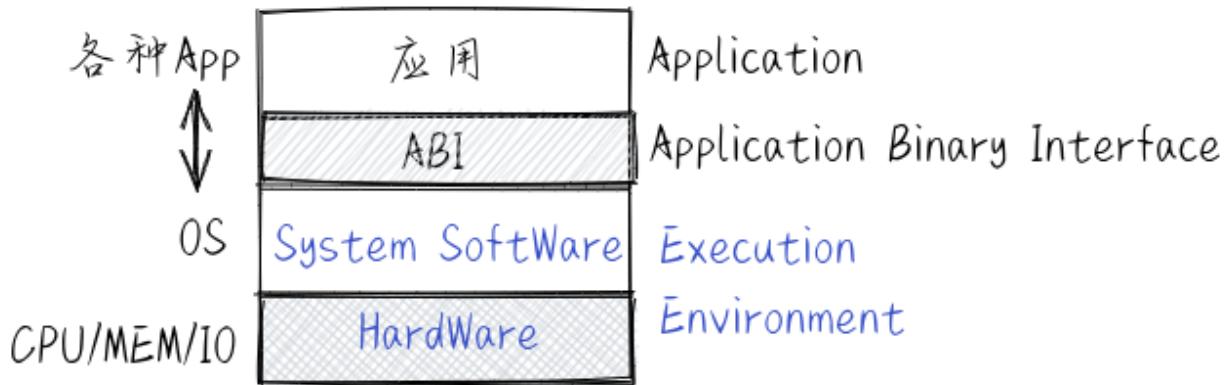
工具软件是操作系统提供的用于维护、调试和优化计算机系统的软件，如磁盘碎片整理工具、系统信息工具和病毒查杀工具等。用户界面可以是图形用户界面（GUI）或命令行界面（CLI）。图形用户界面是操作系统的一种常见用户界面，它使用图形元素（如图标、菜单和按钮）来帮助用户使用操作系统。通常，图形用户界面提供了一个桌面环境，其中包含可以打开和关闭的窗口，用户可以在其中运行应用程序和执行其他操作。

C 语言标准库 `libc`（类似的有 Rust 标准库等）提供了与 OS 交互的系统调用接口，其功能覆盖了整个计算机系统，会被许多不同的软件访问和调用。

从这个角度来看，操作系统是一种系统软件。

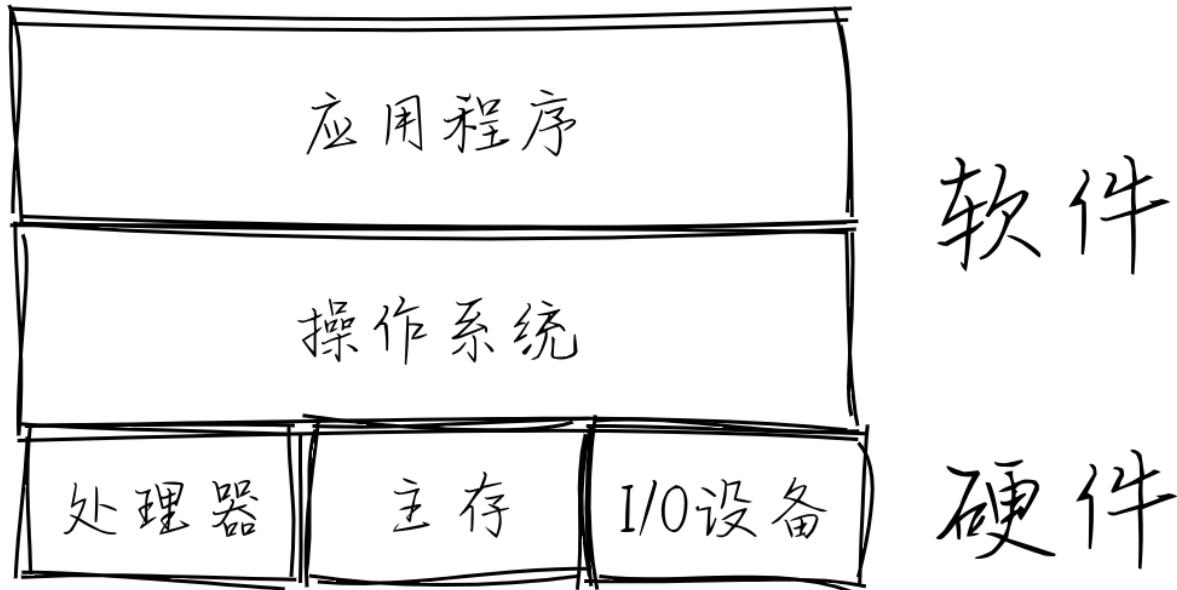
执行环境

站在应用程序的角度来看，我们可以发现常见的应用程序其实是运行在由硬件、操作系统内核、运行时库、图形界面支持库等所包起来的一个执行环境（*Execution Environment*）中，如下图所示。执行环境提供了运行应用软件所需的运行时服务，包括内存管理、文件系统访问、网络连接等，这些服务大部分是由操作系统来提供的。应用程序只需根据与系统软件约定好的应用程序二进制接口（ABI, Application Binary Interface）来请求执行环境提供的各种服务或功能，从而完成应用程序自己的功能。基于这样的观察，我们可以把操作系统的定义简化为：**应用程序的软件执行环境**。这种概括性描述可以适用于操作系统发展的不同历史时期。从这个角度出发，操作系统可以包括运行时库、图形界面支持库等系统软件。在后续小节“执行环境”中会对执行环境相关含义进行进一步的阐述。



操作系统的定义与组成

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：

1. 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。
2. 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。
3. 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。
4. 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。
5. 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。
6. 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。
7. 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。

下图就是一个典型的 UNIX 操作系统的组成示意图：



1.2.2 站在计算机发展的时间尺度看

虽然电子计算机的出现距今才仅仅七十年左右，但计算机技术和操作系统已经发生了巨大的变化。从计算机发展的短暂的历史角度看，操作系统也是从计算机诞生大约十年后，从无到有地逐步发展起来的。操作系统主要完成硬件控制和为应用程序提供服务这些必不可少的功能，它的历史与计算机的发展史密不可分。操作系统的内涵和功能随着历史的发展也在一直变化、改进中。如今在二十一世纪初期的大众眼中，操作系统就是他们的手机/终端上的软件系统，包括各种应用程序集合，图形界面和网络浏览器是其中重要的组成部分。

其实，操作系统的内涵和外延随着历史的发展也一直在变化，并没有类似于“1+1=2”这样的明确定义。参考地球生物的进化史，我们也给操作系统的进化历史做一个简单的概述，从中可以看到操作系统在各个时间段上包含什么，具有什么样的特征。但无论操作系统的内在实现和具体目标如何变化，其管理计算机硬件，给应用提供服务的核心定位没有变化。

寒武纪生物大爆发时代¹

在 1946 年出现电子计算机的时候，只有人类操作员 (Operator) 来管理和操作机器，还没有操作系统 (Operating System) 这种事物。启动，扳开关，装卡片/纸带等比较辛苦的工作都是计算机操作员或者用户自己完成。操作员/用户带着记录有程序和数据的卡片 (Punch Card) 或打孔纸带去操作计算机。装好卡片/纸带后，启动卡片/纸带阅读器，把程序和数据读入计算机内存中之后，计算机就开始工作，并把结果也输出到卡片/纸带或显示屏上，最后程序停止。

随着程序设计语言和编译技术的进步，推动了程序员开发翻译符号程序（即编译器）来自动把代码转换成机器代码，代替了以前低效的手工机器编码的方式，提高了程序开发的效率。但程序执行的效率还很低。而且随着计算机和 I/O 设备变得更强大，程序运行的时间减少了，相比之下，让计算机运行的准备时间变得更长了，使得计算机的整体执行效率很低。

一般情况下，五十年代初期的计算机每次只能执行一个任务，CPU 大部分时间都在等待人类操作员的缓慢操作。由于过低的人工操作效率浪费了计算机的宝贵机时，所以就引入监控程序 (Monitor) 辅助完成输入、输出、加载、运行程序等工作，从而提高了使用计算机的效率。监控程序就是操作系统最开始的雏形，类似寒武纪生物大爆发中的著名生物—“三叶虫”。在 1951-1954 年前后，Swinnerton-Dwyer 等在 EDSAC (Electronic Delay Storage Automatic Calculator) 计算机上研制了监控程序，这也许是最早的最早操作系统雏形⁹。这种类似监控程序的初级“辅助操作”过程一直持续到 20 世纪 50 年代中后期。

在中国，计算机与操作系统相关技术的发展也很快。1958 年 8 月 1 日，在苏联专家的指导下，中科院计算所、第四机械工业部（电子工业部）等单位，仿制苏联 M-3 小型数字电子计算机，设计完成了中国第一台以电子管为主的小型数字电子计算机—103 型（即 DJS-1 型）计算机，这个计算机没有监控程序。1959 年 10 月 1 日，中科院计算所、四机部等单位合作，仿照苏联 BЭCM-II 大型机，制成了中国第一台以电子管为主的大规模通用数字电子计算机—104 机。在 104 机上的软件已经比较丰富了，包括具有前期操作系统雏形的自检程序、标准子程序库、自动更换地址程序，以及面向应用的算法语言与编译系统。

注解：历史的缩影—寒武纪“三叶虫”操作系统—LibOS

可以在[本书第一章](#)看到初级的“三叶虫”操作系统—LibOS 其实就是一个给应用提供各种服务（比如输出字符串）的库，方便了单一应用程序的开发与运行。

¹ 5 亿年前的寒武纪期间生物种类突然丰富起来，呈爆炸式的增加，期间的典型生物是三叶虫。

⁹ Peter H.Salus, A Quarter Century of UNIX, Addison-Wesley Publishing, 1997

泥盆纪² 鱼类时代和二叠纪³ 两栖动物时代

在 20 世纪 50~60 年代，计算机发展到大型机阶段，计算能力和 I/O 处理能力进一步加强，存储空间进一步扩展，应用领域逐步扩大，这使得所对应的的各种早期操作系统具有多样化、专用化的特点。计算机生产厂商设计出针对各自硬件的专用操作系统，且大部分用汇编语言编写。这导致操作系统的开发效率不高，不具有跨硬件平台的可移植性，进化比较缓慢。而且当时的处理器时间很宝贵，将计算机系统的一次使用过程限制在一个用户范围内是一种很大的浪费。例如，在早期计算机系统中，当用户通过监控程序加载应用程序时，处理器保持空闲状态，所有其他用户的程序都不能执行。为提高计算机执行效率，操作系统进化在持续进行，从“手工操作”和“监控程序”阶段进化到了“批处理 Batch”和“多道程序 Multiprogramming”阶段。

批处理是指把一批作业（英文：Job，古老的术语，可理解为现在的应用程序）以脱机方式（offline mode）输入到磁带上，并使这批作业能一个接一个地连续处理，流程如下：

1. 将磁带上的一个作业装入内存；
2. 操作系统把运行控制权交给该作业；
3. 当该作业处理完成后，控制权被交还给操作系统；
4. 重复 1-3 的步骤处理下一个作业直到所有作业处理完毕。

注解：脱机方式（offline mode）

基于脱机方式的操作是指没有与计算机进行关联所完成的操作。比如把包含程序的卡片放到卡片机上，把打印纸安装到打印机上等。与此相反的是联机方式（online mode）的操作，即通过与计算机相联所完成的操作，比如计算机把正在运行的应用程序所计算出来的结果通过打印机打印到纸上。

现在，脱机方式的操作更多的是指断网下进行的操作，联机方式是指在联网下进行的操作。

这样能充分地利用计算机系统：即尽量使该系统连续运行，减少 CPU 的空闲时间。批处理操作系统分为单道批处理系统和多道批处理系统。单道批处理操作系统只能管理内存中的一个（道）作业，无法充分利用大型计算机系统中的所有资源，致使系统整体性能较差。这就像泥盆纪² 的史前鱼类-邓氏鱼，有着坚硬的头部铠甲，很强壮，但运动缓慢，灵敏度低，离不开水。

注解：历史的缩影-泥盆纪“邓氏鱼”操作系统-BatchOS

可以在本书第二章看到“邓氏鱼”操作系统-BatchOS，虽然每次只能加载运行一个应用，但 BatchOS 通过硬件隔离机制让 APP 与 OS 隔离，加强了系统安全，并在一定程度上提高了执行效率。

1956 年，鲍勃·帕特里克（Bob Patrick）在美国通用汽车的系统监督程序（system monitor）的基础上，为美国通用汽车和北美航空公司在 IBM 704 计算机上设计了最早的批处理操作系统-GM-NAA I/O¹¹。这个最早的操作系统已经具有了单道批处理系统的基本功能。

1964 年，IBM 公司开发了面向 System/360 系列大型计算机的统一可兼容的操作系统——OS/360¹²，它是一种多道批处理操作系统。多道批处理操作系统能管理内存中的多个（道）作业，可比较充分地利用计算机系统中的所有资源，提升系统整体性能。这里的“多道”是指多个程序，即允许同时把多个程序放入内存，并允许它们交替在 CPU 中运行，它们共享系统中的各种硬、软件资源。当一个程序因 I/O 请求而暂停运行时，CPU 便立即转去运行另一个程序。这就像二叠纪³ 的两栖动物，当水中暂时有危险或食物不多的时候，可以离开水面到陆地上来，并享用陆地上的动植物资源。

然而，处理器共享产生了程序间隔离的需求，以避免出现一个程序中的运行错误使其他程序崩溃或损坏的情况。为此计算机中进一步添加了硬件级的内存保护机制，限制了一个程序在运行时能访问的地址空间，提高

² 4 亿年前的泥盆纪期间鱼类空前繁荣，并在晚期出现了两栖动物。

³ 3 亿年前的二叠纪期间是一个承上启下的阶段，两栖类动物最繁盛，爬行动物逐渐繁荣。

¹¹ Maarten Bullynck, What is an Operating System? A historical investigation (1954–1964), <https://halshs.archives-ouvertes.fr/halshs-01541602/document>

¹² 布鲁克斯 (Brooks, F. P.), 人月神话 (40 周年中文纪念版), 2015

了故障隔离的能力，并减少了隔离的开销。另外，虽然批处理操作系统提高了系统的执行效率，但其缺点是人机交互性差，比如，批处理计算的一个实际挑战是如何调试应用程序和操作系统本身。如果程序员的代码出现错误，必须重新编码，上传内存，再执行。这需要花费以小时和天为单位的时间开销，使得程序员修改和调试程序很不方便。实质上是将计算机重新变成单用户系统。

1965 年 2 月，哈尔滨军事工程学院（国防科技大学前身）慈云桂教授等成功研制我国第一台晶体管通用电子计算机 441B-I。441B-I 是中国第二代计算机，具有批处理功能的操作系统雏形、汇编语言、FORTRAN 语言及标准程序库等丰富的软件。

注解：历史的缩影—二叠纪“锯齿螈”、三叠纪“始初龙”和三叠纪“腔骨龙”操作系统

在本书第三章可以看到二叠纪“锯齿螈”操作系统支持在内存中驻留多个应用，形成多道程序操作系统—Multiprog OS；三叠纪“始初龙”操作系统—Coop OS 进一步进化，支持协作式多道程序，即支持应用程序主动放弃 CPU 并切换到另一个应用继续执行，从而提高系统整体执行效率；三叠纪“腔骨龙”操作系统—Timesharing OS 则可以抢占应用的执行，从而可以公平和高效地分时执行多个应用，提高系统的整体效率。

侏罗纪⁴ 与白垩纪⁵ 的恐龙时代

1965 年，时任仙童半导体公司研发总监 Gordon Moore 在被要求预测未来半导体产业发展趋势时提出：“当价格不变时，集成电路上可容纳的晶体管数目每年便会增加一倍”，这被称为摩尔定律。1975 年，Gordon Moore 的同事 David House 认为根据摩尔定律，计算机芯片的性能也将每 18 个月提升一倍。这种累积效应使得计算成本逐渐下降，使得软件开发者不必将全部精力用于提高处理器利用率，而是可以开始努力提升用户的使用体验。例如，UNIX 是在 70 年代初在当时没有人使用的备用计算机上开发的。20 世纪 50 年代末，提高人机交互方式的分时操作系统越来越崭露头角。分时是指多个用户和多个程序以很小的时间间隔来共享使用同一台计算机上的 CPU 和其他硬件资源。1961 年，麻省理工学院的 Fernando Corbató 带领团队成功研发了在 IBM 709 计算机上的 CTSS（Compatible Time-Sharing System，兼容时间共享系统）操作系统⁶，它拥有分时系统必须有的特征：支持多个用户分享使用同一台计算机，即宏观上的同一时间段内能完成多个人机交互工作。

在 CTSS 的鼓舞下，1964 年，麻省理工学院、贝尔实验室及美国通用电气公司共同研发一个目标远大的操作系统：MULTICS（MULTIplexed Information and Computing System），它是一套安装在大型主机上、支持多人多任务的操作系统。MULTICS 以兼容分时系统（CTSS）做基础，建置在美国通用电力公司的大型机 GE-645，目标是连接 1000 部终端机，支持 300 位用户同时上线。因 MULTICS 的目标太宏大，而研发工作进度过于缓慢，1969 年 AT&T 的 Bell 实验室从 MULTICS 研发中撤出。CTSS 和 MULTICS 这就像侏罗纪时期体型庞大的食肉恐龙—霸王龙，称霸一时，但进化缓慢，最终灭绝。

但贝尔实验室的两位软件工程师 Ken Thompson 与 Dennis Ritchie 借鉴了一些重要的 MULTICS 设计思想和理念，以 C 语言为基础发展出小巧灵活的 UNIX 操作系统¹⁰。UNIX 操作系统的早期版本是完全免费的，可以轻易获得并随意修改，所以它得到了广泛的接受。后来，它成为开发小型机操作系统的起点。由于早期的广泛应用，它已经成为分时操作系统的典范。这好像一种生活在侏罗纪晚期的小型恐龙—始祖鸟，它可能是鸟类的祖先，最终进化为可以展翅高飞的飞鸟。

注解：Ken Thompson 与 Dennis Ritchie 于 1974 年 7 月在 the Communications of the ACM 期刊上发表 “The UNIX Time Sharing System”，引起了学术界的广泛兴趣并向他们要源码进行分析和学习，所以 Unix v5 以“仅用于教育目的”的开放协议，提供给各个大学作为操作系统教学之用，成为当时操作系统课程中的重要学习资料，而且各大学和公司开始进一步研究 Unix，并对 Unix 进行改进和扩展，从而使得 Unix 在世界上广泛流行。

1973 年，南京大学徐家福、中科院软件所仲萃豪、北京大学杨英清合作，研制了系统程序设计语言 XCY。XCY 由徐（Xu）家福、仲萃（Cui）豪、杨（Yang）英清的姓名汉语拼音各取一个字母组成。南京大学的开发小

⁴ 2 亿年前的侏罗纪期间温暖潮湿，爬行类动物的代表—恐龙成为当时的统治者，哺乳动物开始发展。

⁵ 1 亿年前的白垩纪期间温暖干旱，恐龙经历了从鼎盛到灭绝的巨大变化，哺乳动物兴起。

¹⁰ Brian W. Kernighan, UNIX: A History and a Memoir, Independently published, 2020

组还用 XCY 语言编写开发了 240 机的 DJS200/XT I、DJS200/XT II、XW 等具有分时和进程管理能力的通用操作系统。

注解：历史的缩影—侏罗纪和白垩纪的操作系统

侏罗纪和白垩纪是恐龙称霸地球的最后繁荣时期。侏罗纪“头甲龙”操作系统、白垩纪“伤齿龙”操作系统、白垩纪“霸王龙”操作系统、白垩纪“迅猛龙”操作系统、白垩纪“达科塔盗龙”操作系统、白垩纪“慈母龙”操作系统可以在[本书第四章](#)到[本书第八章](#)中看到。

[第四章](#)的侏罗纪“头甲龙”操作系统—“Address Space OS”主要解决多个应用所在内存隔离的问题，从而确保应用之间不会相互破坏各自的内存空间，并进一步发展出地址空间、虚拟内存等操作系统核心概念。

[第五章](#)的白垩纪“伤齿龙”操作系统—“Process OS”主要解决多个应用灵活创建与执行的问题，并进一步发展出进程、调度等操作系统核心概念。

[第六章](#)的白垩纪“霸王龙”操作系统—“Filesystem OS”主要解决数据持久保存的应用需求，并进一步提出文件的概念，并且把文件与地址空间作为进程的资源来进行管理，综合了操作系统三大核心概念。

[第七章](#)的白垩纪“迅猛龙”操作系统—“IPC OS”主要解决进程间的数据交换与信息通知的应用需求，并把部分 IPC 功能建立在文件的抽象之下，简化了应用开发的复杂性。

[第八章](#)的白垩纪“达科塔盗龙”操作系统—“Thread OS”支持线程抽象，把线程作为处理器的调度单位，并成为进程资源管理的一部分内容。线程间通过共享进程的地址空间建立了线程间共享数据和进行数据交换的基础。而白垩纪“慈母龙”操作系统—“SyncMutex OS”通过建立同步互斥机制来让线程间能够有序、互斥地访问共享资源，从而让基于多线程的应用能够高效正确地完成任务。

古近纪⁶ 哺乳动物时代

20 世纪 70 年代中后期，微型处理器的快速发展使计算机的应用普及至中小企业及个人爱好者，推动了 PC (Personal Computer, 个人计算机) 的发展，也进一步推动了面向一般大众使用的操作系统的出现。其代表是由微软公司在 20 世纪 80 年代为个人计算机开发（实际是购买）的 DOS 操作系统，其特点是简单易用。后来又开发了有图形用户界面 (GUI) 的操作系统—MS Windows，极大地简化了一般用户使用计算机的难度，使得个人计算机得到了快速的普及和广泛的使用。这里需要注意的是，第一个带 GUI 界面的个人计算机原型起源于伟大却又让人扼腕叹息的施乐帕洛阿图研究中心 (PARC, Palo Alto Research Center)，PARC 研发出的带有图标、弹出式菜单和重叠窗口的图形交互界面 (GUI, Graphical User Interface)，可利用鼠标的点击动作来进行操控，这是当今我们所使用的 GUI 系统的基础。支持便捷的图形交互界面也成为自 20 世纪 70 年代以来操作系统的主要特征之一。这就像古近纪⁶的哺乳动物，能在陆上跑，空中飞和水里游，有很强的适应性和生存能力。

1980 年左右，用于远望号测量船的多机实时操作系统 GX-73 诞生了。1983 年，国防科技大学研制了中国第一台亿次巨型机—银河-I，并研制了配套的多道批处理操作系统—YHOS。同年，电子工业部第六研究所设计实现了面向微型计算机的操作系统 CCDOS (汉字磁盘操作系统，英语：Chinese Characters Disk Operation System)，这是一个基于微软公司 DOS 微机操作系统的汉化版本。80 年代到 90 年代，微软的 DOS 操作系统和后继的 Windows 操作系统和 Intel 公司的 x86 处理器相互支持，形成了 Wintel 联盟，逐渐在桌面计算机市场上被广泛使用。中国的操作系统主要集中在使用范围较小的科研领域。1989 - 1997 年这段时间，中国计算机服务总公司与中国软件技术公司联合国内高校和科研院所，共同承担了开发与 UNIX 兼容的 COSIX 操作系统的科技攻关计划。COSIX 取得了很多科研成果，但由于没基于新的硬件进行研发等多种因素，最终没有形成广泛的操作系统产业生态。

注解：历史的缩影—“侏罗猎龙”操作系统

在哺乳动物之前已经出现了具有灵巧感知能力的生物，这就是晚侏罗纪的侏罗猎龙。它体型小巧，有着适应微光环境的大眼睛和具备压力感应能力的神奇鳞片，能够发现外部环境的微小变化，这种“小巧灵敏”的特征为后续哺乳动物的生物进化开辟了一个方向。[第九章](#)的“侏罗猎龙”操作系统—“Device OS”支持与新增

⁶ 0.6 亿年前的古近纪时期，哺乳动物迅速发展，且形态多样化，逐渐统治了地面。

的键盘、鼠标、GPU 等多种 Virtio 外设的交互，可以形成初级的人机交互 GUI 能力，而且支持在内核中响应中断来降低 I/O 响应的总体延迟。这在某种程度上体现了当前广泛使用的有图形界面的桌面/移动终端操作系统的基本特征。

第四纪智人时代⁷

21 世纪以来，Internet 和移动互联网的迅猛发展，使得在服务器领域和个人终端的应用与需求大增，数据中心和个人终端已经进入了人们的日常生活中。现在我们拥有种类繁多的计算设备，在这些设备上运行着许多不同的操作系统，操作系统设计者面临的功能权衡取决于硬件的物理能力以及应用程序和用户需求。下面是一些目前常见类型的操作系统：

- 面向服务器的操作系统：当前大家常用的主流搜索引擎、新媒体、电子商务网站和大数据处理系统等一般都托管在数据中心的计算机上。每台计算机都是强大的服务器，运行着服务器操作系统，典型的例子是 Linux。通常每台服务器只运行一个应用服务程序，例如数据库服务器或 Web 服务器等，用于处理成千上万个用户传入的网络服务请求，所以吞吐量（每秒处理的请求数量）是一个关键的优化目标。而安全和可靠也是服务器操作性系统需要重点关注的目标。服务器操作系统的一种形态是虚拟机（Virtual Machine Monitor, VMM），它可在一台物理机上虚拟出多台虚拟计算机，可以像运行应用程序一样运行另一个操作系统。通过虚拟机可以充分利用数据中心中资源利用率不高的物理服务器，提高整个数据中心的运行效率。典型的服务器操作系统有：FreeBSD、微软的 Windows Server、基于 Linux 系的 RHL、Ubuntu、openEuler、龙蜥操作系统、麒麟服务器操作系统等。
- 面向台式机/笔记本电脑和上网本的操作系统：典型的例子是：Windows、Mac OS X、Linux、Chrome OS 等。这些操作系统主要面向单个用户，运行许多应用程序，并具有各种 I/O 设备。有人可能认为只有一个用户，就没有必要将系统设计为支持多用户共享，这会使得操作系统设计更加简单，但安全性相对弱一些。
- 面向智能移动终端（手机/平板）的操作系统：智能移动终端是一种带有强大处理器的手机或平板，能够运行第三方应用程序。智能移动终端操作系统的典型例子包括两个霸主 iOS、Android，和曾经辉煌过的 Symbian，还有几乎快消失的 WebOS、Blackberry OS 和 Windows Phone 等。智能移动终端一般只有一个用户，对交互响应能力、长效的电池使用时间和各种应用的支持有着迫切的需求。
- 嵌入式操作系统：随着物联网的发展，处理器芯片可以集成到各种的消费设备中，从机顶盒、手表到机器人等，形成各种物联网中的嵌入式设备。这些嵌入式设备的功能相对单一，通常运行嵌入式操作系统。典型的嵌入式操作系统有嵌入式 Linux、VxWorks、FreeRTOS、RT-Thread、SylinxOS 等。预计在下一个十年，通过面向物联网设备的操作系统，可以把各种嵌入式设备连接在一起，实现灵活多变的功能协同与组合，形成万物互联的新应用场景和生态。

从上面的简介，我们可以看到，iOS 和 Android 操作系统是 21 世纪个人终端操作系统的代表，Linux 在巨型机到数据中心服务器操作系统中占据了统治地位。以 Android 系统为例，Android 操作系统是一个包括 Linux 操作系统内核、基于 Java 的中间件、用户界面和关键应用软件的移动设备软件栈集合。这里介绍一下广泛用在服务器领域、智能移动终端和嵌入式系统中的操作系统内核—Linux 操作系统内核。1991 年 8 月，芬兰学生 Linus Torvalds（林纳斯·托瓦兹）在 comp.os.minix 新闻组贴上了以下这段话：

" 你好，所有使用 minix 的人 - 我正在为 386 (486) AT 做一个免费的操作系统（只是为了爱好）
..."

而他所说的“爱好”成为了大家都知道的 Linux 操作系统内核。这个时代的操作系统的特征是联网，提高网络的吞吐量，并降低传输延迟是这个时代的网络操作系统追求的目标。Linux 就像是第四纪出现的智人，横扫陆地上的各种强大生物，出现在生物界的顶端，统治了整个地球。

中国对 Linux 系统的引进源于在芬兰读博士的宫敏。1994 年，他回国休假，随手带了 20 张磁盘、存储了 80GB 的自由软件，其中就有 Linux。由于 Linux 基于 GPL 协议开放源代码，Linux 在国内的高校中被小范围传播。从 1999 年起，国内出现了很多基于 Linux 的操作系统公司，出现了 Xteam、蓝点、中科红旗、银河麒麟、中软 Linux 等几十种发行版。但这些发行版大多数基于 Fedora/CentOS/Debian/Ubuntu 进行二次开发，并没有形成桌面计算机的应用生态，在二十年左右的时间内，大部分发行版都退出了。目前（2019 年之后）在

⁷ 尤瓦尔·赫拉利所著的“人类简史”书中提到的智人遍布地球，可类比现在的 Linux。

桌面计算机领域，麒麟操作系统和统信 UOS 操作系统目前有比较好的应用发展趋势。在服务器领域，华为的 openEuler 操作系统和阿里的龙蜥操作系统借助于云计算的快速发展，形成了较好的云应用生态。在嵌入式操作系统领域，国内有不少有技术特色的操作系统，主要代表是 RT-Thread、SylinxOS、LiteOS 等。

二十一世纪神人时代⁸

当前，大数据、人工智能、机器学习、高速移动互联网络、AR/VR 对操作系统等系统软件带来了新的挑战。如何有效支持和利用这些技术是未来操作系统的方向。我们看到了华为逐步推出的 OpenHarmony 系统；小米也推出了物联网软件平台小米 Vela；阿里推出了 AliOS Thing；腾讯推出了 Tencent OS；苹果公司接连推出 A14、M1 等基于 ARM 的 CPU，逐步开始淘汰 X86 CPU；微软推出 Windows 10 IoT，Google 推出 Fuchsia OS。大家都在做着各种位于云、边、端操作系统的技术调整和创新，构建多种形态的网络基础设施。可以发现操作系统的外延在放大，位于云、边、端的操作系统通过多种形态的网络基础设施，跳出了传统单机为主的运行模式，支持应用程序在分布式环境下的互联、互通以及互操作，从而进一步延伸为分布式操作系统。

另外，随着人工智能和机器学习的快速发展，下一个与人工智能充分融合并带有分布式特征的操作系统即将到来，并试图通过这种操作系统带来的连贯用户体验，打通从数据中心、服务器、桌面、移动端、边缘设备等的整个 AI 和物联网 (IoT, Internet of Things) 的生态。也许这种未来操作系统与之前的操作系统相比，其最大的不同是具有了人工智能的属性，跳出了单个设备节点，通过多种网络从不同维度来管理多个设备。这种操作系统也许就是尤瓦尔·赫拉利所著的《未来简史》⁸ 中描述的“无所不能”的神人操作系统。

目前支持 AIoT 的操作系统设计实现在本书中还没有对应的章节，不过我们的同学也设计了 zCore 操作系统，欢迎看完本书的同学能够尝试参与或独立设计面向未来的操作系统。

1.3 操作系统的系统调用接口

1.3.1 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上一节提到的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。

注解：API 与 ABI 的区别

应用程序二进制接口 ABI 是不同二进制代码片段的连接纽带。ABI 定义了二进制机器代码级别的规则，主要包括基本数据类型、通用寄存器的使用、参数的传递规则、以及堆栈的使用等等。ABI 与处理器和内存地址等硬件架构相关，是用来约束链接器 (Linker) 和汇编器 (Assembler) 的。在同一处理器下，基于不同高级语言编写的应用程序、库和操作系统，如果遵循同样的 ABI 定义，那么它们就能正确链接和执行。

应用程序编程接口 API 是不同源代码片段的连接纽带。API 定义了一个源码级（如 C 语言）函数的参数，参数的类型，函数的返回值等。因此 API 是用来约束编译器 (Compiler) 的：一个 API 是给编译器的一些指令，

⁸ 尤瓦尔·赫拉利所著的“未来简史”书中描述的神人可类比于未来支持 AI 的分布式操作系统。

它规定了源代码可以做以及不可以做哪些事。API 与编程语言相关，如 libc 是基于 C 语言编写的标准库，那么基于 C 的应用程序就可以通过编译器建立与 libc 的联系，并能在运行中正确访问 libc 中的函数。

1.3.2 系统调用接口与功能

对于通用的应用程序，一般需要关注如下问题，并希望得到操作系统的支持：

- 一个运行的程序如何能输出字符信息？如何能获得输入字符信息？
- 一个运行的程序可以要求更多（或更少）的内存空间吗？
- 一个运行的程序如何持久地存储用户数据？
- 一个运行的程序如何与连接到计算机的设备通信并通过它们与物理世界通信？
- 多个运行的程序如何同步互斥地对共享资源进行访问？
- 一个运行的程序可以创建另一个程序的实例吗？需要等待另外一个程序执行完成吗？一个运行的程序能暂停或恢复另一个正在运行的程序吗？

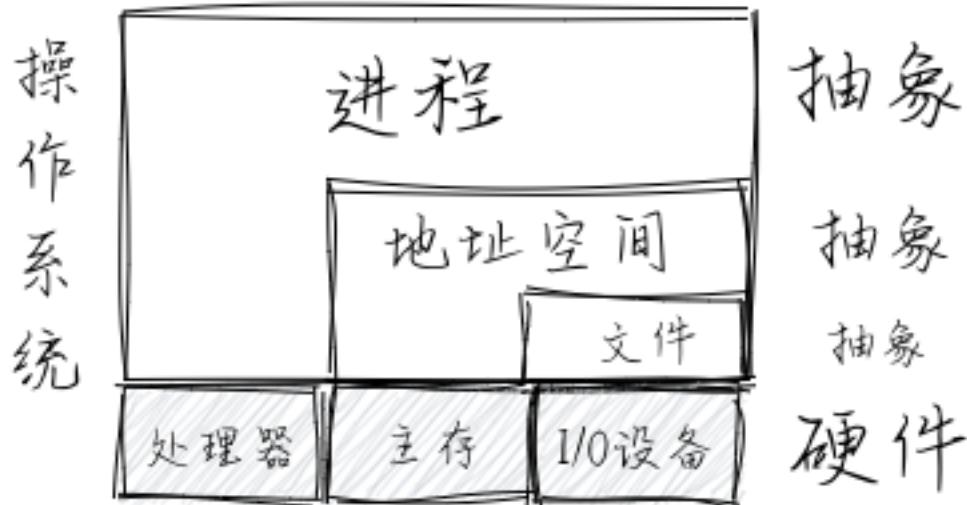
操作系统主要通过基于 ABI 的系统调用接口来给应用程序提供上述服务，以支持应用程序的各种需求。对于实际操作系统而言，有多少操作系统，就有多少种不同类型的系统调用接口。通用操作系统为支持各种应用的服务需求，需要有相对多的系统调用服务接口，比如目前 Linux 有超过三百个的系统调用接口。下面列出了一些相对比较重要的操作系统接口或抽象，以及它们的大致功能：

- 进程（即程序运行过程）管理：复制创建进程 fork、退出进程 exit、执行进程 exec 等。
- 线程管理：线程（即程序的一个执行流）的创建、执行、调度切换等。
- 线程同步互斥的并发控制：互斥锁 mutex、信号量 semaphore、管程 monitor、条件变量 condition variable 等。
- 进程间通信：管道 pipe、信号 signal、事件 event 等。
- 虚存管理：内存空间映射 mmap、改变数据段地址空间大小 sbrk、共享内存 shm 等。
- 文件 I/O 操作：对存储设备中的文件进行读 read、写 write、打开 open、关闭 close 等操作。
- 外设 I/O 操作：外设包括键盘、显示器、串口、磁盘、时钟…，主要采用文件 I/O 操作接口。

注解：上述表述在某种程度上说明了操作系统对计算机硬件重要组成的抽象和虚拟化，这样会有助于应用程序开发。应用程序员只需访问统一的抽象概念（如文件、进程等），就可以使用各种复杂的计算机物理资源（处理器、内存、外设等）：

- 文件 (File) 是外设的一种抽象和虚拟化。特别对于存储外设而言，文件是持久存储的抽象。
- 地址空间 (Address Space) 是对内存的抽象和虚拟化。
- 进程 (Process) 是对计算机资源的抽象和虚拟化。而其中最核心的部分是对 CPU 的抽象与虚拟化。

运行应用程序



有了这些系统调用接口，简单的应用程序就不用考虑底层硬件细节，可以在操作系统的服务支持和管理下简洁地完成其应用功能了。在现阶段，也许大家对进程、文件、地址空间等抽象概念还不了解，在接下来的章节会对这些概念有进一步的介绍。值得注意的是，我们设计的各种操作系统总共只用到三十个左右系统调用功能接口（如下表所示），就可以支持应用需要的上述功能。而且这些调用与最初的 UNIX 的系统调用接口类似，几乎没有变化。尽管 UNIX 的系统调用最早是在 1970 年左右设计和实现的，但这些调用中的大多数仍然在今天的系统中广泛使用。

编号	系统调用	所在章节	功能描述
1	sys_exit	2	结束执行
2	sys_write	2/6	(2) 输出字符串/(6) 写文件
3	sys_yield	3	暂时放弃执行
4	sys_get_time	3	获取当前时间
5	sys_getpid	5	获取进程 id
6	sys_fork	5	创建子进程
7	sys_exec	5	执行新程序
8	sys_waitpid	5	等待子进程结束
9	sys_read	5/6	(5) 读取字符串/(6) 读文件
10	sys_open	6	打开/创建文件
11	sys_close	6	关闭文件
12	sys_dup	7	复制文件描述符
13	sys_pipe	7	创建管道
14	sys_kill	7	发送信号给某进程
15	sys_sigaction	7	设立信号处理例程
16	sys_sigprocmask	7	设置要阻止的信号
17	sys_sigreturn	7	从信号处理例程返回
18	sys_sleep	8	进程休眠一段时间
19	sys_thread_create	8	创建线程
20	sys_gettid	8	获取线程 id

下页继续

表 1 - 续上页

编号	系统调用	所在章节	功能描述
21	sys_waitid	8	等待线程结束
22	sys_mutex_create	8	创建锁
23	sys_mutex_lock	8	获取锁
24	sys_mutex_unlock	8	释放锁
25	sys_semaphore_create	8	创建信号量
26	sys_semaphore_up	8	减少信号量的计数
27	sys_semaphore_down	8	增加信号量的计数
28	sys_condvar_create	8	创建条件变量
29	sys_condvar_signal	8	唤醒阻塞在条件变量上的线程
30	sys_condvar_wait	8	阻塞与此条件变量关联的当前线程

1.3.3 系统调用接口举例

我们以 rCore-Tutorial 中的例子，一个应用程序显示一个字符串，来看看系统调用的具体内容。应用程序的代码如下：

```

1 // user/src/bin/hello_world.rs
2 ...
3 pub fn main() -> i32 {
4     println!("Hello world from user mode program!");
5     0
6 }
```

这个程序的功能就是显示一行字符串（重点看第 4 行的代码）。注意，这里的 `println!` 一个宏。而进一步跟踪源代码（位于 `user/src/console.rs`），可以看到 `println!` 会进一步展开为 `write` 函数：

```

1 // user/src/console.rs
2 ...
3 impl Write for Stdout {
4     fn write_str(&mut self, s: &str) -> fmt::Result {
5         write(STDOUT, s.as_bytes());
6         Ok(())
7     }
8 }
```

这个 `write` 函数就是对系统调用 `sys_write` 的封装：

```

1 // user/src/lib.rs
2 ...
3 pub fn write(fd: usize, buf: &[u8]) -> isize {
4     sys_write(fd, buf)
5 }
6
7 // user/src/syscall.rs
8 ...
9 pub fn sys_write(fd: usize, buffer: &[u8]) -> isize {
10     syscall(SYS_WRITE, [fd, buffer.as_ptr() as usize, buffer.len()])
11 }
```

`sys_write` 用户库函数封装了 `sys_write` 系统调用的 API 接口，这个系统调用 API 的参数和返回值的含义如下：

- `SYSCALL_WRITE` 表示 `sys_write` 的系统调用号
- `fd` 表示待写入文件的文件描述符；

- *buf* 表示内存中缓冲区的起始地址；
- *len* 表示内存中缓冲区的长度；
- 返回值：返回成功写入的长度或错误值

而 `sys_write` 系统调用的 ABI 接口描述了具体用哪些寄存器来保存参数和返回值：

```

1 // user/src/syscall.rs
2 ...
3 fn syscall(id: usize, args: [usize; 3]) -> isize {
4     let mut ret: isize;
5     unsafe {
6         asm!(
7             "ecall",
8             inlateout("x10") args[0] => ret,
9             in("x11") args[1],
10            in("x12") args[2],
11            in("x17") id
12         );
13     }
14     ret
15 }
```

这里我们看到，API 中的各个参数和返回值分别被 RISC-V 通用寄存器 *x17*（即存放系统调用号）、*x10*（存放 *fd*，也保存返回值）、*x11*（存放 *buf*）和 *x12*（存放 *len*）保存。

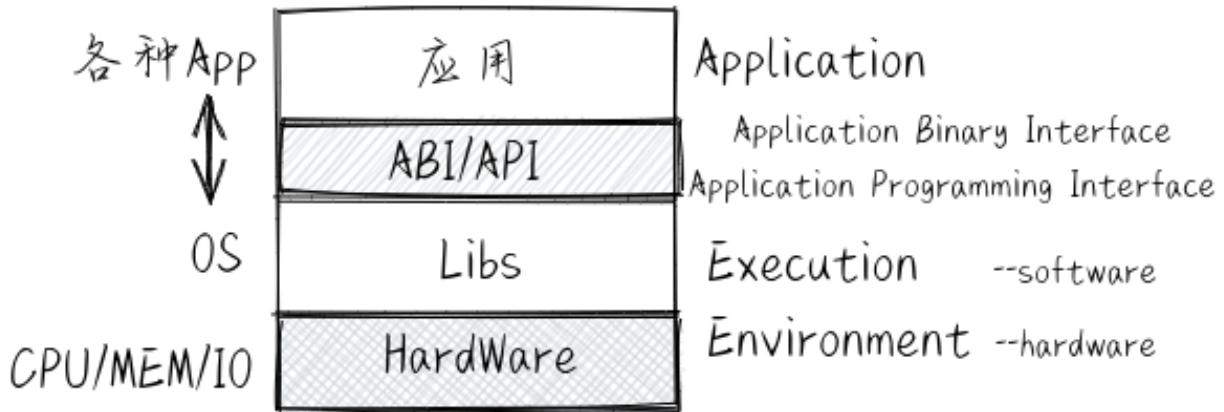
1.4 操作系统抽象

接下来同学可站在操作系统实现的角度来看操作系统。操作系统为了能够更好地管理计算机系统并为应用程序提供便捷的服务，在计算机和操作系统的技术研究和发展的过程中，形成了一系列的核心（抽象）概念：执行环境、进程、地址空间和文件，这些奠定了操作系统内核设计与实现的基础。

注解：在本书中，关于执行环境、进程、地址空间和文件的抽象表示不会仅仅就是一个文字描述，还会在后续章节关于具体操作系统设计与运行的讲述中，以具体化的静态数据结构、动态执行导致计算机中物理/虚拟资源的改变来展示。从而让同学能够建立操作系统抽象概念与操作系统具体实现之间的内在联系。

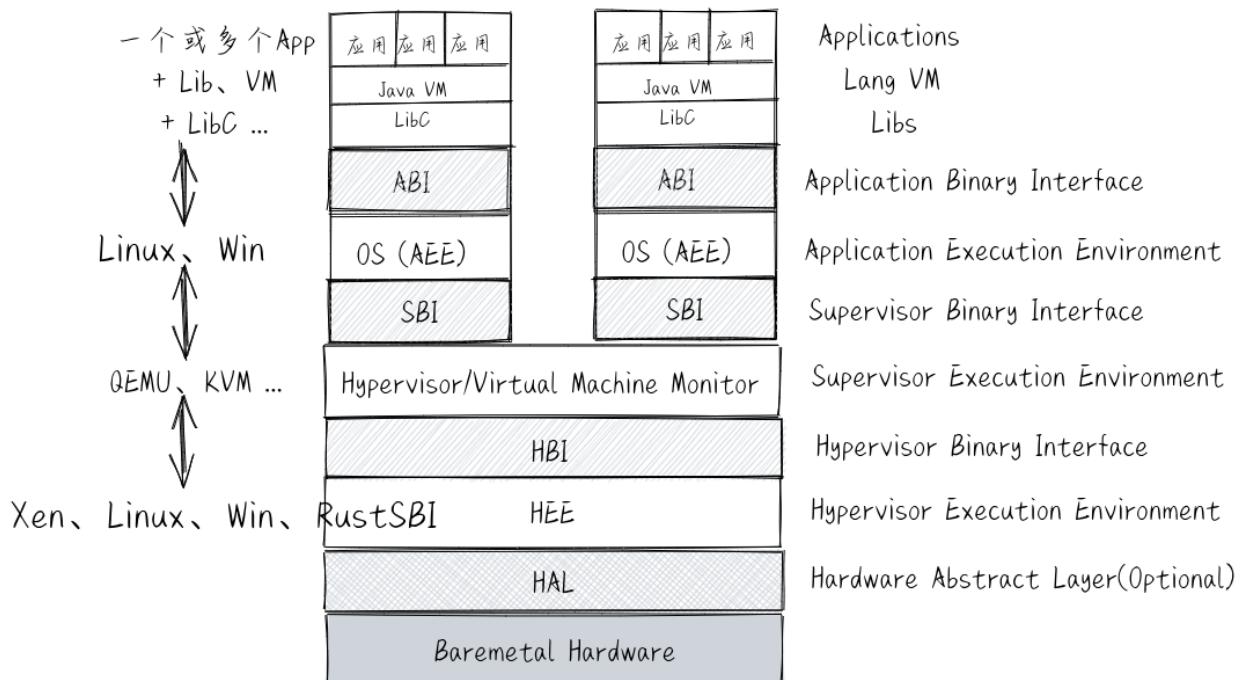
1.4.1 执行环境

执行环境 (Execution Environment) 是一个内涵很丰富且有一定变化的术语，它主要负责给在其上执行的软件提供相应的功能与资源，并可在计算机系统中形成多层次的执行环境。对于现在直接运行在裸机硬件 (Bare-Metal) 上的操作系统，其执行环境是 计算机的硬件。计算机刚刚诞生时，还没有操作系统的概念，对于直接运行在裸机硬件上的应用程序而言，其执行环境也是 计算机的硬件。随着计算机技术的发展，应用程序下面形成了一层比较通用的函数库，这使得应用程序不需要直接访问硬件了，它所需要的功能（比如显示字符串）和资源（比如一块内存）都可以通过函数库的函数来帮助完成。在第二个阶段，应用程序的执行环境就变成了 函数库 -> 计算机硬件，而这时函数库的执行环境就是计算机的硬件。



再进一步，操作系统取代了函数库来访问硬件，函数库通过访问操作系统的系统调用服务来进一步给应用程序提供丰富的功能和资源。在第三个阶段，应用程序的执行环境就变成了 **函数库 -> 操作系统内核 -> 计算机硬件**。在后面又出现了基于 Java 语言的应用程序，在函数库和操作系统之间，多了一层 Java 虚拟机，此时 Java 应用程序的执行环境就变成了 **函数库 -> Java 虚拟机 -> 操作系统内核 -> 计算机硬件**。在云计算时代，在传统操作系统与计算机硬件之间多了一层 Hypervisor/VMM，此时应用程序的执行环境变成了 **函数库 -> Java 虚拟机 -> 操作系统内核 -> Hypervisor/VMM -> 计算机硬件**。这里可以看到，随着软件需求的多样化和复杂化，**执行环境的层次**也越来越多。

另外，CPU 在执行过程中，可以在不同层次的执行环境之间切换，这称为 **执行环境切换**。执行环境切换主要是通过特定的 API 或 ABI 来完成的，这样不同执行环境的软件就能实现数据交换与互操作，而且还保证了彼此之间有清晰的隔离。



对于应用程序的执行环境而言，应用程序只能看到执行环境直接提供给它的接口（API 或 ABI），这使得应用程序所能得到的服务取决于执行环境提供给它的访问接口。所以，操作系统可以看成是应用程序执行环境，其形态可以是一个库，也可以是一个虚拟机等，或者它们的某种组合形式。比如，手机上的 **Android 操作系统**是 Android 应用程序的执行环境，它包括了库形态的 Framework 层，执行 Java 程序的虚拟机层，与操作系统交互的系统类 C 库和 Linux kernel。运行在服务器上的 **麒麟服务器操作系统**、**openEuler 服务器操作系统**、**龙蜥服务器操作系统**、**RHEL-RedHat 服务器操作系统**是各种服务器应用软件，如 Web Server 应用、数据库

等的执行环境，这些操作系统都包含了与操作系统交互的系统类 C 库、Java 虚拟机、容器系统、VMM 虚拟机系统等和 Linux kernel。

基于上面的介绍，我们可以给应用程序的执行环境一个基本的定义：执行环境是应用程序正确运行所需的服务与管理环境，用来完成应用程序在运行时的数据与资源管理、应用程序的生存期等方面的处理，它定义了应用程序有权访问的其他数据或资源，并决定了应用程序的行为限制范围。

注解：在本书中，我们将主要侧重设计与实现类似 Linux kernel 这样的 **操作系统内核形态的执行环境**。

普通控制流

各种应用程序在执行环境中执行其功能，而具体如何执行，取决于 **程序的控制流**。回顾一下编译原理课上的知识，程序的控制流 (Flow of Control or Control Flow) 是指以一个程序的指令、语句或基本块为单位的执行序列。再回顾一下计算机组成原理课上的知识，处理器的控制流是指处理器中程序计数器的控制转移序列。最简单的一种控制流（没有异常或中断产生的前提下）是一个“平滑的”序列，其中每个要执行的指令地址在内存中都是相邻的。如果站在程序员的角度来看控制流，会发现控制流是程序员编写的程序的执行序列，这些序列是程序员预设好的。程序运行时能以多种简单的控制流（顺序、分支、循环结构和多层嵌套函数调用）组合的方式，来一行一行的执行源代码（以编程语言级的视角），也是一条一条的执行汇编指令（以汇编语言级的视角）。对于上述的不同描述，我们可以统称其为 **普通控制流** (CCF, Common Control Flow, 简称控制流)。在应用程序视角下，它只能接触到它所在的执行环境，不会跳到其他执行环境，所以应用程序执行基本上是以普通控制流的形式完成整个运行的过程。

异常控制流

应用程序在执行过程中，如果发出系统调用请求，或出现外设中断、CPU 异常等情况，处理器执行的前一条指令和后一条指令将会位于两个完全不同的位置，即不同的执行环境。比如，前一条指令还在应用程序的代码段中，后一条指令就跑到操作系统的代码段中去了，这就是一种控制流的“突变”，即控制流脱离了其所在的执行环境，并产生**执行环境的切换**。我们把这种“突变”的控制流称为 **异常控制流** (ECF, Exceptional Control Flow)。

应用程序感知不到这种异常的控制流情况，这主要是由于操作系统把这种情况透明地进行了执行环境的切换和对各种异常情况的处理，让应用程序从始至终地认为没有这些异常控制流的产生。

简单地说，**异常控制流**是处理器在执行过程中的突变，其主要作用是通过硬件和操作系统的协同工作来响应处理器状态中的特殊变化。比如当应用程序正在执行时，产生了时钟外设中断，导致操作系统打断当前应用程序的执行，转而进入**操作系统执行环境**去处理时钟外设中断。处理完毕后，再回到应用程序中被打断的地方继续执行。

注解：本书是从操作系统的角度来给出的异常控制流的定义。

在《深入理解计算机系统 (CSAPP)¹》中，对异常控制流也给出了相关定义：系统必须能对系统状态的变化做出反应，这些系统状态不是被内部程序变量捕获，也不一定和程序的执行相关。现代系统通过使控制流发生突变对这些情况做出反应。我们称这种突变为异常控制流 (Exceptional Control Flow, ECF)

我们这里的异常控制流不涉及 C++/Java 等编程语言级的 exception 机制。

¹ 兰德尔 E. 布莱恩特 (Randal E. Bryant) 著，龚奕利，贺莲译, Computer Systems: A Programmer's Perspective (3rd Edition), 深入理解计算机系统 (原书第 3 版), 机械工业出版社, 2016

控制流上下文（执行环境的状态）

站在硬件的角度来看普通控制流或异常控制流的具体执行过程，我们会发现从控制流起始的某条指令执行开始，指令可访问的所有物理资源的内容，包括自带的所有通用寄存器、特权级相关特殊寄存器、以及指令访问的内存等，会随着指令的执行而逐渐发生变化。

这里我们把控制流在执行完某指令时的物理资源内容，即确保下一时刻能继续正确执行控制流指令的物理资源内容称为控制流的 **上下文 (Context)**，也可称为控制流所在执行环境的状态。

我们这里说的控制流的上下文是指仅会影响控制流正确执行的有限的物理/虚拟资源内容。这里需要理解程序中控制流的上下文对程序正确执行的影响。如果在某时刻，由于某种有意或无意的原因，控制流的上下文发生了变化（比如某个寄存器的值变了），但并不是由于程序的控制流本身的指令导致的，这就会使得接下来的程序指令执行出现偏差，并最终导致执行过程或执行结果不符合预期，这种情形称为 **程序执行错误**。而操作系统有责任来保护应用程序中控制流的上下文，以让应用程序得以正确执行。

注解：

- 物理资源：即计算机硬件资源，如 CPU 的寄存器、可访问的物理内存等。
 - 虚拟资源：即操作系统提供的资源，如文件，网络端口号，网络地址，信号等。
-

如果一个控制流属于某个函数，那么这个控制流的上下文简称为函数调用上下文。如果一个控制流属于某个应用程序，那么这个控制流的上下文简称为应用程序上下文。如果把某 [进程](#) 看做是运行的应用程序，那么这个属于某个应用程序的控制流可简称为某进程上下文。如果一个控制流属于操作系统，那么这个控制流的上下文简称为操作系统上下文。如果一个控制流是属于操作系统中处理中断/异常/陷入的那段代码，那么这个控制流的上下文简称为中断/异常/陷入的上下文。

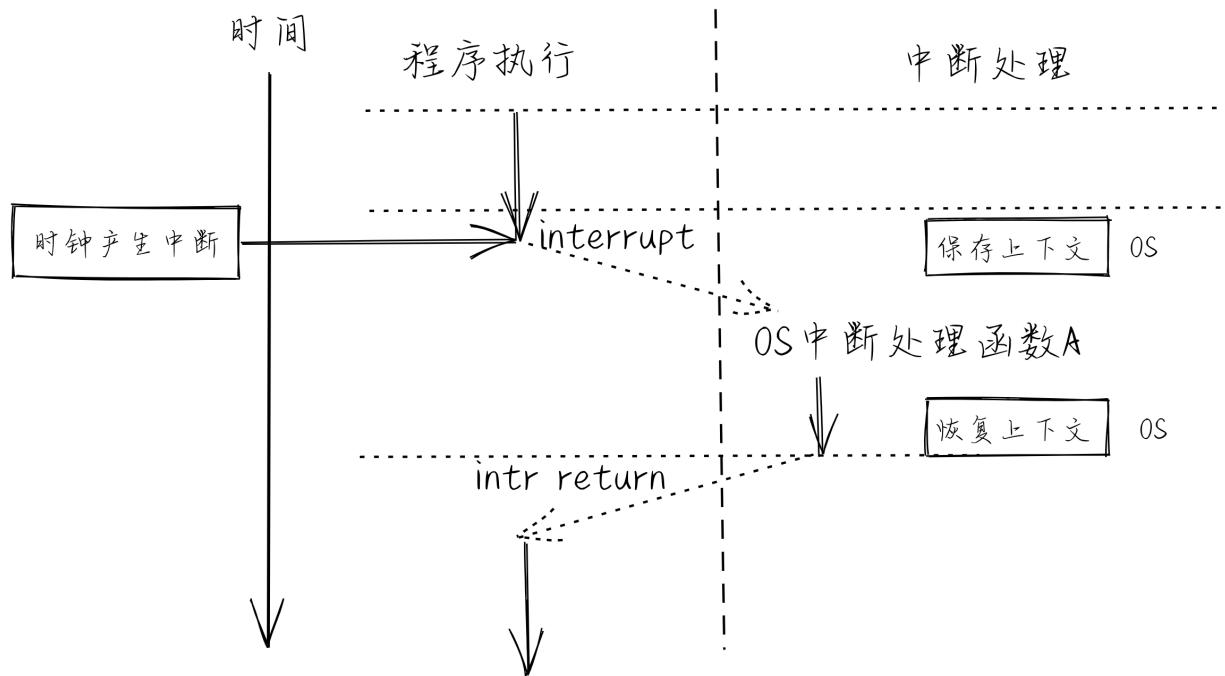
在 CPU 不断执行指令的过程中，各种前缀的上下文（执行环境的状态）会不断地变化。如果出现了处理器在执行过程中的突变（即异常控制流）或转移（如多层函数调用），需要由维持执行环境的软硬件协同起来，保存发生突变或转移前的控制流上下文，即当前执行环境的状态（比如突变或函数调用前一刻的指令寄存器，栈寄存器和其他一些通用寄存器等内容），并在完成突变处理或被调用函数执行完毕后，恢复突变或转移前的控制流上下文。这是由于完成与突变相关的执行会破坏突变前的控制流上下文（比如上述各种寄存器的内容），导致如果不保存之前的控制流上下文，就无法恢复到突变前正确的执行环境，继续正常的普通控制流的执行。

对于异常控制流的上下文保存与恢复，主要是通过 CPU 和操作系统（手动编写在栈上保存与恢复寄存器的指令）来协同完成；对于函数转移控制流的上下文保存与恢复，主要是通过编译器（自动生成在栈上保存与恢复寄存器的指令）来帮助完成的。

在操作系统中，需要处理三类异常控制流：外设中断 (Device Interrupt)、陷入 (Trap) 和异常 (Exception，也称 Fault Interrupt)。

异常控制流：中断

外设 **中断 (Interrupt)** 是指由外部设备引起的外部 I/O 事件，如时钟中断、控制台中断等。外设中断是异步产生的，与处理器的执行无关。产生中断后，操作系统需要进行中断处理来响应中断请求，这会破坏被打断前应用程序的控制流上下文，所以操作系统要保存与恢复被打断前应用程序的控制流上下文。

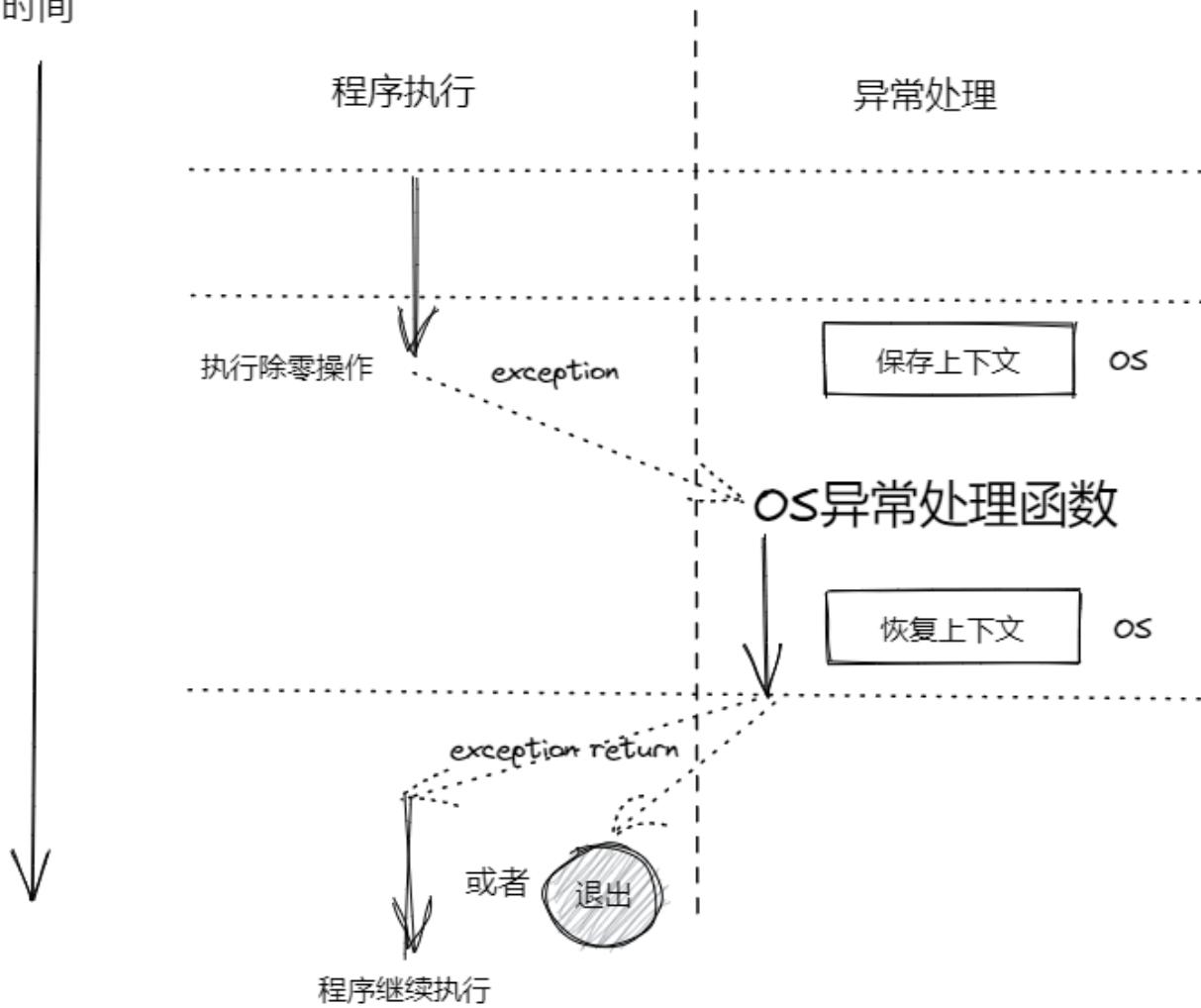


异常控制流: 异常

异常 (Exception) 是在处理器执行指令期间检测到不正常的或非法的内部事件 (如 x86 平台上的除零错、地址访问越界)。产生异常后, 操作系统需要进行异常处理, 这会破坏被打断前应用程序的控制流上下文, 所以操作系统要保存与恢复被打断前应用程序的控制流上下文。

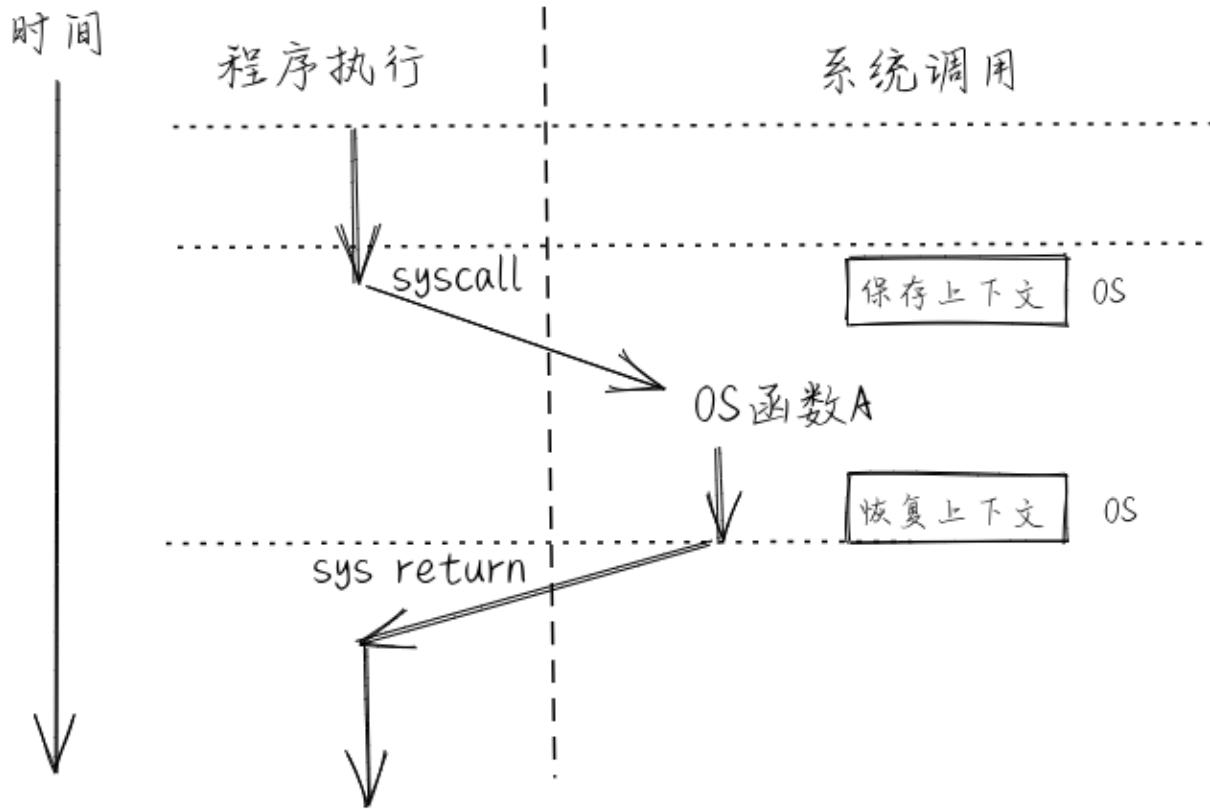
注解: 如果是应用程序产生的不可恢复的异常, 操作系统有权直接终止该应用程序的执行。

时间



异常控制流：陷入

陷入 (Trap) 是程序在执行过程中由于要通过系统调用请求操作系统服务而有意引发的事件。产生陷入后，操作系统需要执行系统调用服务来响应系统调用请求，这会破坏陷入前应用程序的控制流上下文，所以操作系统要保存与恢复陷入前应用程序的控制流上下文。



在后面的叙述中，如果没有特别指出，我们将用简称中断、陷入、异常来区分这三种异常控制流。

注解：本书是从操作系统的角度来给出的中断 (Interrupt)、陷入 (Trap) 和异常 (Exception) 的定义。

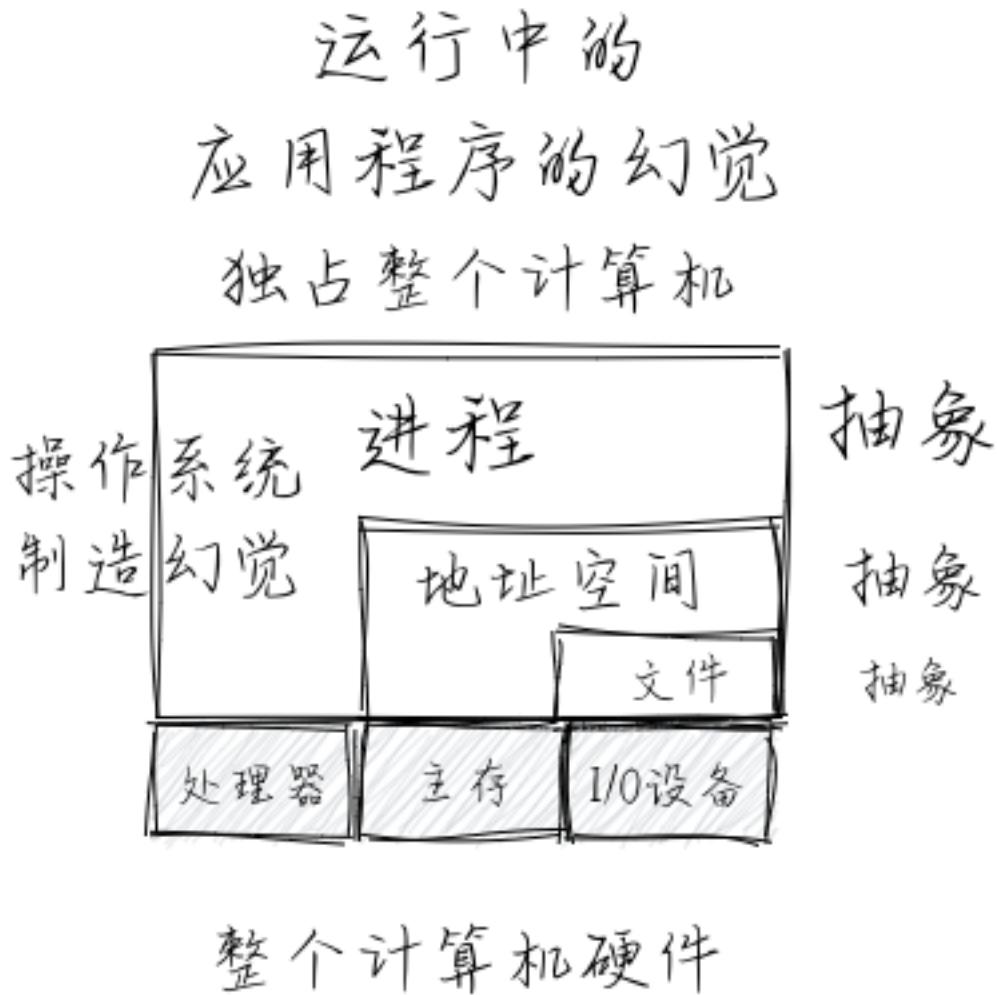
在不同的书籍中，对于中断、陷入和异常的定义会有一些差别。有的书籍把中断、陷入和异常都统一为一种中断，表示程序的当前控制流被打断了，要去执行不属于这个控制流的另外一个没有程序逻辑先后关系的控制流；也有书籍把这三者统一为一种异常，表示相对于程序的正常控制流而言，出现了一种没有程序逻辑先后关系的异常控制流。甚至也有书籍把这三者统一为一种陷入，表示相对于程序的正常控制流而言，CPU 会陷入到操作系统内核中去执行。

在 RISC-V 的特权级规范文档中，异常指的是由于 CPU 当前指令执行而产生的异常控制流，中断指的是与 CPU 当前指令执行无关的异常控制流，中断和异常统称为陷入。当中断或异常触发时，我们首先进行统一的陷入处理流程，随即根据 mcause/scause 等寄存器的内容判定目前触发的是中断还是异常，再对应进行处理。在操作系统意义上的陷入，在 RISC-V 的语境下属于异常的一部分。另外，在 x86 架构下的“软件中断”（也即指令 int 0x80）可以理解为操作系统意义上的陷入，但在 RISC-V 语境下软件中断表示一种特殊的处理核间中断。

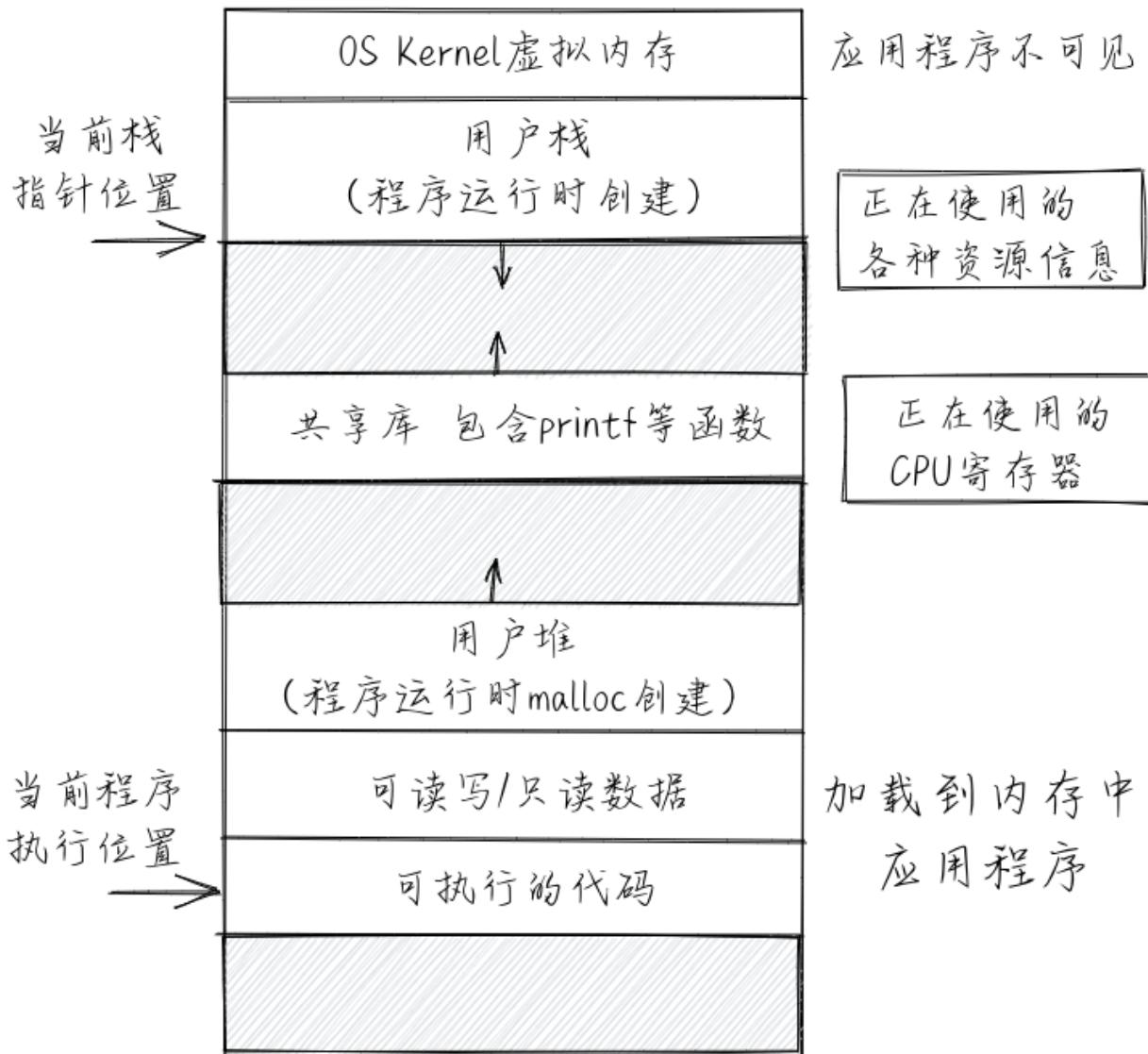
这些都是从不同的视角来阐释中断、陷入和异常，并没有一个唯一精确的解释。对于同学而言，重点是了解这些术语在后续章节的操作系统设计实现中所表示的具体含义和特征。

1.4.2 进程

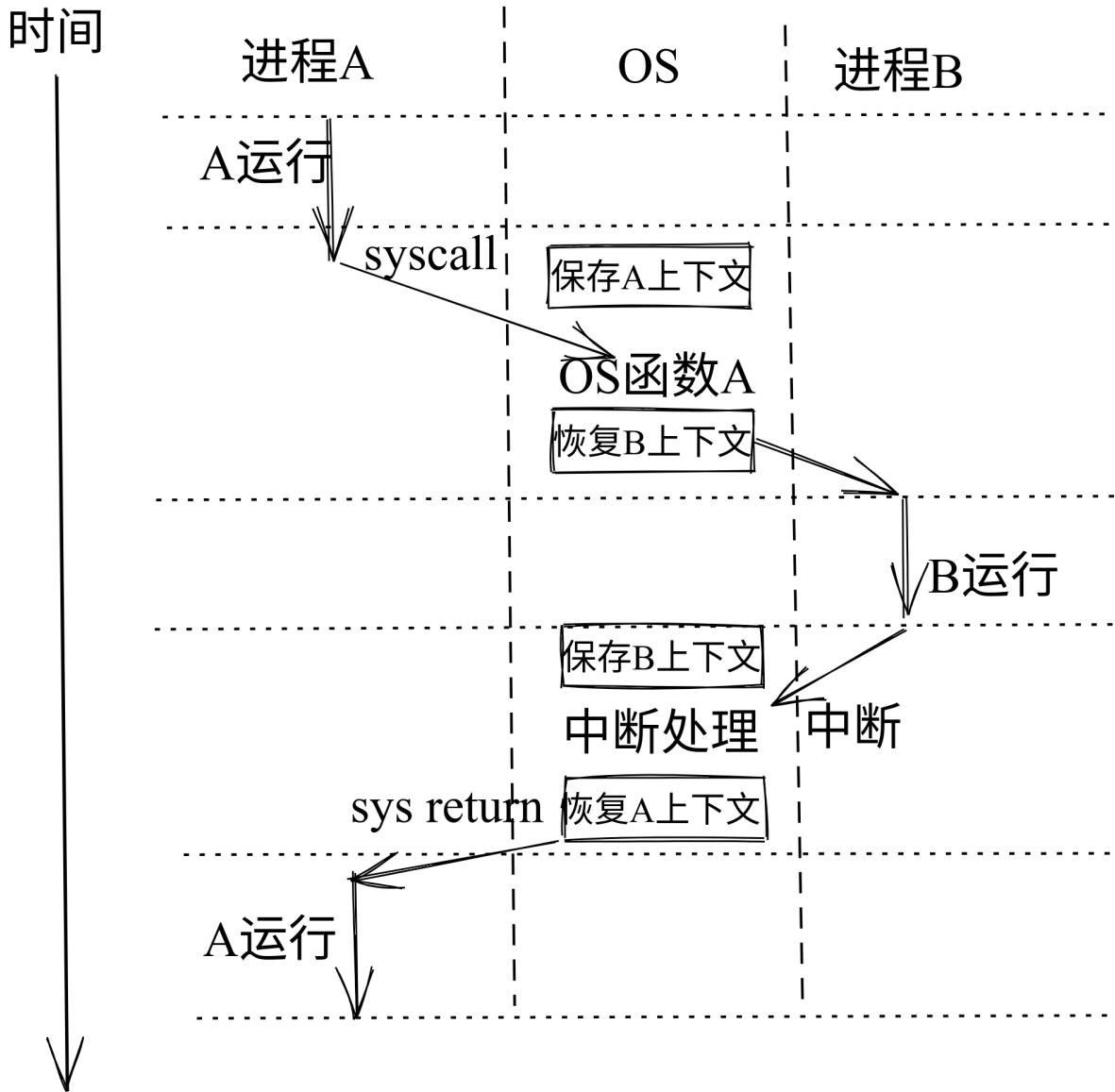
站在应用程序自身的角度来看，进程 (Process) 的一个经典定义是一个正在运行的程序实例。当程序运行在操作系统中的时候，从程序的视角来看，它会产生一种“幻觉”：即该程序是整个计算机系统中当前运行的唯一的程序，能够独占使用处理器、内存和外设，而且程序中的代码和数据是系统内存中唯一的对象。



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。进程上下文如下图所示：



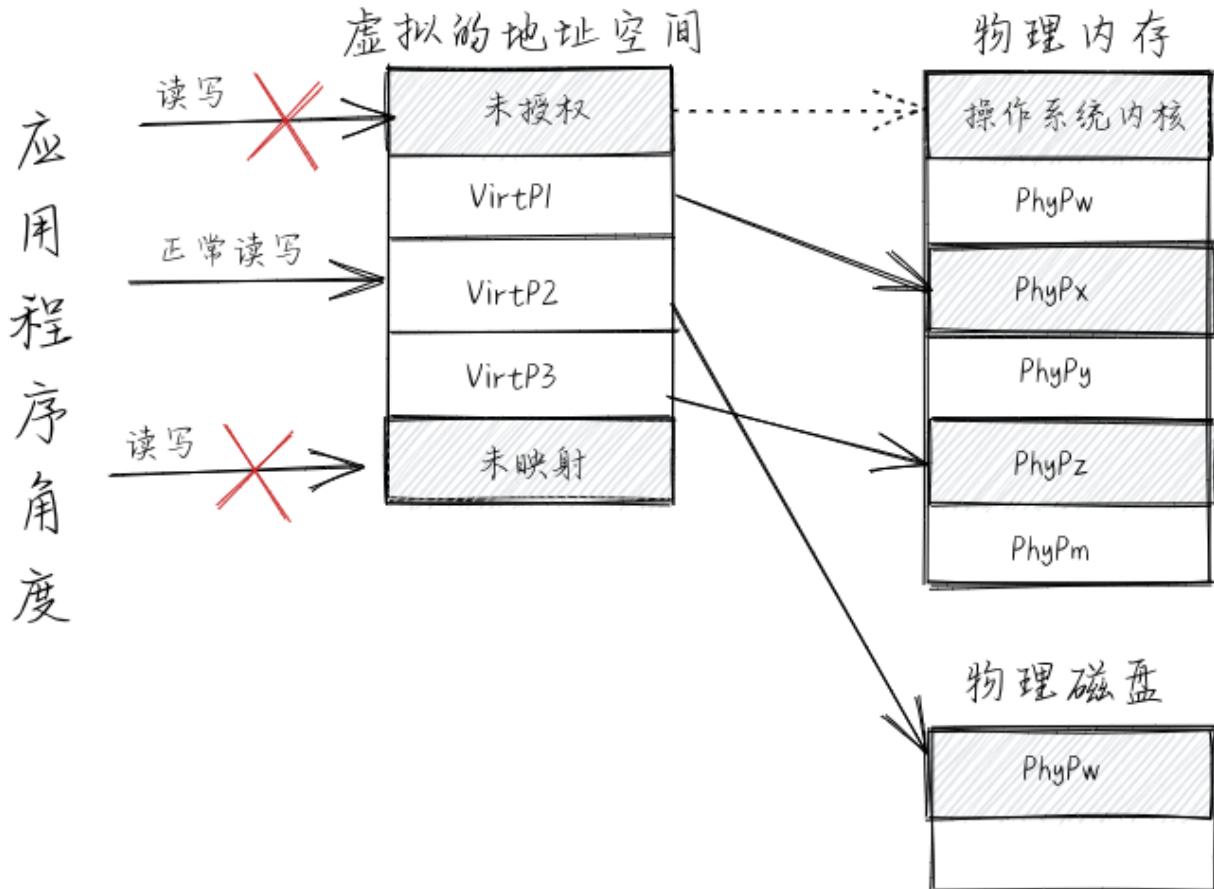
我们知道，处理器是计算机系统中的硬件资源。为了提高处理器的利用率，操作系统需要让处理器足够忙，即让不同的程序轮流占用处理器来运行。如果一个程序因某个事件而不能运行下去时，就通过进程上下文切换把处理器占用权转交给另一个可运行程序。进程上下文切换如下图所示：



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。

1.4.3 地址空间

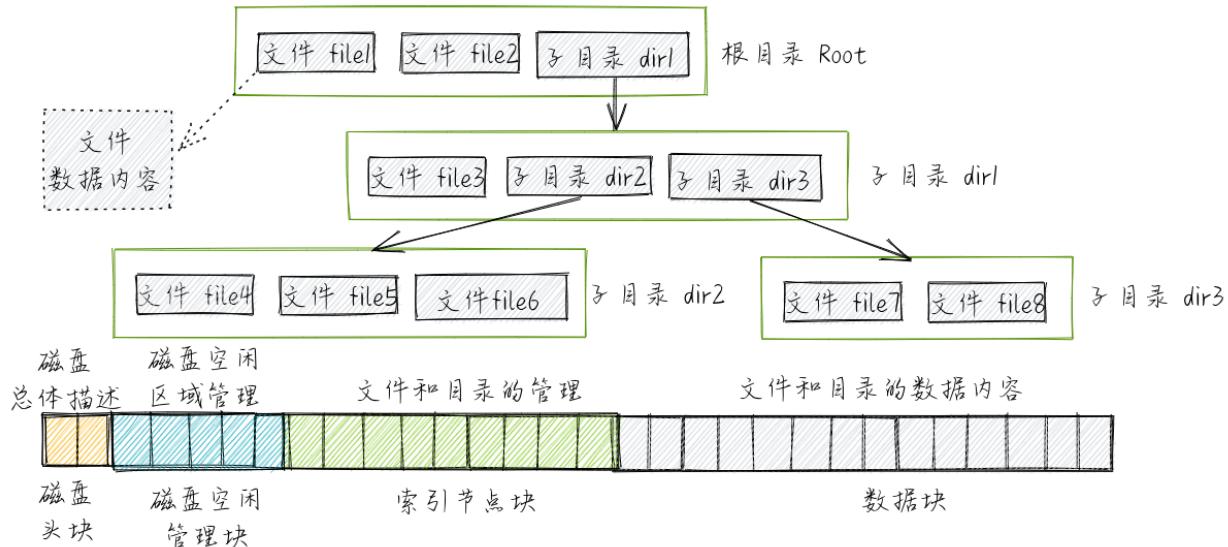
地址空间 (Address Space) 是对物理内存的虚拟化和抽象，也称虚存 (Virtual Memory)。它就是操作系统通过处理器中的内存管理单元 (MMU, Memory Management Unit) 硬件的支持而给应用程序和用户提供一个大的（可能超过计算机中的物理内存容量）、连续的（连续的地址空间编址）、私有的（其他应用程序无法破坏）的存储空间。这需要操作系统将内存和外存（即持久存储，硬盘是一种典型的外存）结合起来管理，为用户提供一个容量比实际内存大得多的虚拟存储器，并且需要操作系统为应用程序分配内存空间，使用户存放在内存中的程序和数据彼此隔离、互不侵扰。操作系统中的虚存管理与处理器的 MMU 密切相关，在启动虚存机制后，软件通过 CPU 访问的每个虚拟地址都需要通过 CPU 中的 MMU 转换为一个物理地址来进行访问。下面是虚拟的地址空间与物理内存和物理磁盘映射的图示：



1.4.4 文件

文件 (File) 主要用于对持久存储的抽象，并进一步扩展到为外设的抽象。具体而言，文件可理解为存放在持久存储介质（比如硬盘、光盘、U 盘等）上，方便应用程序和用户读写的数据。以磁盘为代表的持久存储介质的数据访问单位是一个扇区或一个块，而在内存中的数据访问单位是一个字节或一个字。这就需要操作系统通过文件来屏蔽磁盘与内存差异，尽量以内存的读写方式来处理持久存储的数据。当处理器需要访问文件中的数据时，可通过操作系统把它们装入内存。文件管理的任务是有效地支持文件的存储、检索和修改等操作。

下面是文件对磁盘的抽象映射图示：



从一个更高和更广泛的层次上看，各种外设虽然差异很大，但也有基本的读写操作，可以通过文件来进行统一的抽象，并在操作系统内部实现中来隐藏对外设的具体访问过程，从而让用户可以以统一的文件操作来访问各种外设。这样就可以把文件看成是对外设的一种统一抽象，应用程序通过基本的读写操作来完成对外设的访问。

1.5 操作系统的特征

基于操作系统的四个抽象，我们可以看出，从总体上看，操作系统具有五个方面的特征：虚拟化（Virtualization）、并发性（Concurrency）、异步性、共享性和持久性（Persistency）。操作系统的虚拟化可以理解为它对内存、CPU的抽象和处理；并发性和共享性可以理解为操作系统支持多个应用程序“同时”运行；异步性可以从操作系统调度、中断处理对应用程序执行造成的影响等几个方面来理解；持久性则可以从操作系统中的文件系统支持把数据方便地从磁盘等存储介质上存入和取出来理解。

1.5.1 虚拟性

内存虚拟化

首先来看看内存虚拟化。程序员在写应用程序的时候，不用考虑其程序的起始内存地址要放到计算机内存的具体某个位置，而是用字符串符号定义了各种变量和函数，直接在代码中便捷地使用这些符号就行了。这是由于操作系统建立了一个地址空间，空间巨大的虚拟内存给应用程序来运行，这是 **内存虚拟化**。内存虚拟化的核心问题是：采用什么样的方式让虚拟地址和物理地址对应起来，也就是如何将虚拟地址“翻译”成物理地址。

内存虚拟化其实是一种“空间虚拟化”，可进一步细分为 **内存地址虚拟化** 和 **内存大小虚拟化**。这里的每个符号在运行时是要对应到具体的内存地址的。这些内存地址的具体数值是什么？程序员不用关心。为什么？因为编译器会自动帮我们把这些符号翻译成地址，形成可执行程序。程序使用的内存是否占得太大了？在一般情况下，程序员也不用关心。

注解：还记得虚拟地址（逻辑地址）的描述吗？

实际上，编译器（Compiler，比如 gcc）和链接器（linker，比如 ld）也不知道程序每个符号对应的地址应该放在未来程序运行时的哪个物理内存地址中。所以，编译器的一个简单处理办法就是，设定一个固定地址（比如 0x10000）作为起始地址开始存放代码，代码之后是数据，所有变量和函数的符号都在这个起始地址之后的

某个固定偏移位置。假定程序每次运行都是位于一个不会变化的起始地址。这里的变量指的是全局变量，其地址在编译链接后会确定不变。但局部变量是放在堆栈中的，会随着堆栈大小的动态变化而变化。这里编译器产生的地址就是虚拟地址。

这里，编译器和链接器图省事，找了一个适合它们的解决办法。当程序要运行的时候，这个符号所对应的虚拟内存地址到计算机的物理内存地址的映射必须要解决了，这自然就推到了操作系统身上。操作系统会把编译器和链接器生成的执行代码和数据放到空闲的物理内存中，并建立虚拟地址到物理地址的映射关系。由于物理内存中的空闲区域是动态变化的，这导致虚拟地址到物理地址的映射关系也是动态变化的，需要操作系统来维护好可变的映射关系，确保编译器“固定起始地址”的假设成立。只有操作系统维护好了这个映射关系，才能让程序员只需写一些易于人理解的字符串符号来代表一个内存空间地址。这样，编译器只需确定一个固定地址作为程序的起始地址，就可以不用考虑将来这个程序要在哪个物理地址空间运行的问题，从而实现了**内存地址虚拟化**。

应用程序在运行时不用考虑当前物理内存是否够用。如果应用程序需要一定空间的内存，但由于在某些情况下，物理内存的空闲空间可能不多了，这时操作系统通过把物理内存中最近没使用的空间（不是空闲的，只是最近用得少）换出（就是“挪地”）到硬盘上暂时缓存起来，这样空闲空间就大了，就可以满足应用程序的运行时内存需求了，从而实现了**内存大小虚拟化**。

CPU 虚拟化

再来看 CPU 虚拟化。不同的应用程序可以在内存中并发运行，相同的应用程序也可有多个拷贝在内存中并发运行。而每个程序都“认为”自己完全独占了 CPU 在运行，这是**“CPU 虚拟化”**，也是一种**“时间虚拟化”**。操作系统给了运行的应用程序一个幻象，即操作系统把时间分成小段，每个应用程序占用其中一小段时间片运行，用完这一时间片后，操作系统会切换到另外一个应用程序，让它运行。由于时间片很短，操作系统的切换开销也很小，应用程序或使用应用程序的用户基本上是看不出的，反而感觉到多个程序各自在独立“并行”执行，从而实现了**CPU 虚拟化**。

1.5.2 并发性

操作系统为了能够让 CPU 充分地忙起来，并充分利用各种资源，就需要有多种不同的应用程序在执行。这些应用程序是分时执行的，并由操作系统来完成各个应用在运行时的任务切换。并发性虽然能有效改善系统资源的利用率，但也带来了对共享资源的争夺问题，即同步互斥问题。还会带来执行时间的不确定性问题，即并发程序在执行中是走走停停，断续推进的，使得应用程序的完成时间是不确定的。并发性对操作系统的设计也带来了很多挑战，一不小心就会出现程序执行结果不确定，程序死锁等很难调试和重现的问题。

注解:

- 并行 (Parallel) 是指两个或者多个事件在同一时刻发生；
- 并发 (Concurrent) 是指两个或多个事件在同一时间间隔内发生。

对于基于单 CPU 的计算机而言，各个“同时”运行的程序其实是串行分时复用一个 CPU，任一个时刻点上只有一个程序在 CPU 上运行。这些虚拟性的特征给应用程序的开发和执行提供了非常方便的执行环境，但也给操作系统的设计与实现提出了很多挑战。

1.5.3 异步性

在这里，异步是指由于操作系统的调度和中断等，会不时地暂停或打断当前正在运行的程序，使得程序的整个运行过程走走停停。在应用程序运行的表现上，特别体现在它的执行完成时间是不可预测的。但需要注意，只要应用程序的输入是一致的，那么它的输出结果应该是符合预期的。

1.5.4 共享性

共享是指多个应用并发运行时，宏观上体现出它们可同时访问同一个资源，即这个资源可被共享。但其实在微观上，操作系统在硬件等的支持下要确保应用程序互斥访问这个共享的资源。比如，在单核处理器下，对于两个应用同时访问同一个内存单元的情况，从宏观的应用层面上看，二者都能正确地读出同一个内存单元的内容；而在微观上，操作系统会调度应用程序的先后执行顺序，确保在任何一个时刻，只有一个应用去访问存储单元。在多核处理器下，多个 CPU 核可能同时访问同一内存单元，在这种多核场景下的共享性不仅仅由 OS 来保证，还需硬件级的 Cache 一致性保证。

1.5.5 持久性

操作系统提供了文件系统来从可持久保存的存储介质（磁盘，SSD 等，以后以硬盘来代表）中取数据和代码到内存中，并可以把内存中的数据写回到硬盘上。硬盘在这里是外设，具有持久性，以文件系统的形式呈现给应用程序。

注解：文件系统也可看成是操作系统对存储外设（如硬盘、SSD 等）的虚拟化。这种持久性的特征进一步带来了共享属性，即在文件系统中的文件可以被多个运行的程序所访问，从而给应用程序之间实现数据共享提供了方便。即使掉电，存储外设上的数据还不会丢失，可以在下一次机器加电后提供给运行的程序使用。持久性对操作系统的执行效率提出了挑战，如何让数据在高速的内存和慢速的硬盘间高效流动是需要操作系统考虑的问题。

1.6 实验环境配置

本节我们将完成环境配置并成功运行 rCore-Tutorial-v3。整个流程分为下面几个部分：

- 系统环境配置
- C/Rust 开发环境配置
- QEMU 模拟器安装
- 其他工具安装
- 运行 rCore-Tutorial-v3

1.6.1 在线开发环境配置

Github Classroom 方式进行在线 OS 环境配置

注：这种方式目前主要用于 2022 年开源操作系统训练营

注解：基于 github classroom 的在线开发方式

基于 github classroom，可方便建立开发用的 git repository，并可基于 github 的 codespace（在线版 ubuntu +vscode）在线开发使用。整个开发环境仅仅需要一个网络浏览器。

1. 在网络浏览器中用自己的 github id 登录 github.com
2. 接收 [第一个实验练习 setup-env-run-os1](#) 的 github classroom 在线邀请，根据提示一路选择 OK 即可。
3. 完成第二步后，你的第一个实验练习 [setup-env-run-os1](#) 的 github repository 会被自动建立好，点击此 github repository 的链接，就可看到你要完成的第一个实验了。
4. 在你的第一个实验练习的网页的中上部可以看到一个醒目的 *code* 绿色按钮，点击后，可以进一步看到 *codespace* 标签和醒目的 *create codespace on main* 绿色按钮。请点击这个绿色按钮，就可以进入到在线的 ubuntu +vscode 环境中
5. 再按照下面的环境安装提示在 vscode 的 *console* 中安装配置开发环境：rustc, qemu 等工具。注：也可在 vscode 的 *console* 中执行 `make codespaces_setenv` 来自动安装配置开发环境（执行 `sudo` 需要 root 权限，仅需要执行一次）。
6. **重要：**在 vscode 的 *console* 中执行 `make setupclassroom_testX`（该命令仅执行一次，X 的范围为 1-8）配置 github classroom 自动评分功能。
7. 然后就可以基于在线 vscode 进行开发、运行、提交等完整的实验过程了。

上述的 3, 4, 5 步不是必须的，你也可以仅仅基于 Github Classroom 生成 git repository，并进行本地开发。

1.6.2 本地操作系统开发环境配置

目前实验内容可支持在 [Ubuntu](#) 操作系统、[openEuler](#) 操作系统、[龙蜥操作系统](#) 等上面进行操作。对于 Windows10/11 和 macOS 上的用户，可以使用 WSL2、VMware Workstation 或 VirtualBox 等相关软件，通过虚拟机方式安装 Ubuntu18.04 / 20.04、openEuler 操作系统、龙蜥操作系统等，并在上面进行实验。

Windows 的 WSL2 方式建立 Linux 环境

对于 Windows10/11 的用户可以通过系统内置的 WSL2 虚拟机（请不要使用 WSL1）来安装 Ubuntu 18.04 / 20.04。步骤如下：

- 升级 Windows 10/11 到最新版（Windows 10 版本 18917 或以后的内部版本）。注意，如果不是 Windows 10/11 专业版，可能需要手动更新，在微软官网上下载。升级之后，可以在 PowerShell 中输入 `winver` 命令来查看内部版本号。
- 「Windows 设置 > 更新和安全 > Windows 预览体验计划」处选择加入“Dev 开发者模式”。
- 以管理员身份打开 PowerShell 终端并输入以下命令：

```
# 启用 Windows 功能: “适用于 Linux 的 Windows 子系统”
>> dism.exe /online /enable-feature /featurename:Microsoft-Windows-Subsystem-Linux /all /norestart

# 启用 Windows 功能: “已安装的系统虚拟机平台”
>> dism.exe /online /enable-feature /featurename:VirtualMachinePlatform /all /
  /norestart

# <Distro> 改为对应从微软应用商店安装的 Linux 版本名, 比如: `wsl --set-version Ubuntu 2` 
# 如果你没有提前从微软应用商店安装任何 Linux 版本, 请跳过此步骤
>> wsl --set-version <Distro> 2

# 设置默认为 WSL 2, 如果 Windows 版本不够, 这条命令会出错
>> wsl --set-default-version 2
```

- 下载 Linux 内核安装包
- 在微软商店 (Microsoft Store) 中搜索并安装 Ubuntu18.04 / 20.04。

VMware 虚拟机方式进行本地 OS 开发环境配置

如果你打算使用 VMware 安装虚拟机的话, 我们已经配置好了一个能直接运行 rCore-Tutorial-v3 的 Ubuntu22.04 镜像, 它是一个 vmdk 格式的虚拟磁盘文件, 只需要在 VMware 中新建一台虚拟机, 在设置虚拟磁盘的时候选择它即可。[百度网盘链接](#) 或者 [清华云盘链接](#) (目前是旧版的 Ubuntu18.04+QEMU5.0 的镜像)。已经创建好用户 oslab, 密码为一个空格。它已经安装了中文输入法和作为 Rust 集成开发环境的 Visual Studio Code, 能够更容易完成实验并撰写实验报告。如果想要使用 VMWare 安装 openEuler 虚拟机的话, 可以在 [openEuler 官网](#) 下载 ISO 自行安装, 接着需要参考网络上的一些教程配置网络和安装图形界面。

Docker 方式进行本地 OS 开发环境配置

注解: Docker 开发环境

感谢 qobilidop, dinghao188 和张汉东老师帮忙配置好的 Docker 开发环境, 进入 Docker 开发环境之后不需要任何软件工具链的安装和配置, 可以直接将 tutorial 运行起来, 目前应该仅支持将 tutorial 运行在 QEMU 模拟器上。

使用方法如下 (以 Ubuntu18.04 为例):

1. 通过 `su` 切换到管理员账户 `root` (注: 如果此前并未设置 `root` 账户的密码需要先通过 `sudo passwd` 进行设置), 在 `rCore-Tutorial-v3` 根目录下, 执行 `make build_docker`, 来建立基于 docker 的开发环境;
 2. 在 `rCore-Tutorial-v3` 根目录下, 执行 `make docker` 进入到 Docker 环境;
 3. 进入 Docker 之后, 会发现当前处于根目录 `/`, 我们通过 `cd mnt` 将当前工作路径切换到 `/mnt` 目录;
 4. 通过 `ls` 可以发现 `/mnt` 目录下的内容和 `rCore-Tutorial-v3` 目录下的内容完全相同, 接下来就可以在这个环境下运行 `tutorial` 了。例如 `cd os && make run`。
-

你也可以在 Windows10/11 或 macOS 原生系统或者其他 Linux 发行版上进行实验, 基本上不会出现太大的问题。不过由于时间问题我们主要在 Ubuntu18.04 on x86-64 上进行了测试, 后面的配置也都是基于它的。如果遇到了问题的话, 请在本节的讨论区中留言, 我们会尽量帮助解决。

手动进行本地 OS 开发环境配置

基于 RISC-V 硬件环境的配置

目前已经出现了可以在 RISC-V 64 (简称 RV64) 的硬件模拟环境 (比如 QEMU with RV64) 和真实物理环境 (如全志哪吒 D1 开发板、SiFive U740 开发板) 的 Linux 系统环境。但 Linux RV64 相对于 Linux x86-64 而言, 虽然挺新颖的, 但还不够成熟, 已经适配和预编译好的应用软件包相对少一些, 适合 hacker 进行尝试。如果同学有兴趣, 我们也给出多种相应的硬件模拟环境和真实物理环境的 Linux for RV64 发行版, 以便于这类同学进行实验:

- Ubuntu for RV64 的 QEMU 和 SiFive U740 开发板系统镜像
- OpenEuler for RV64 的 QEMU 系统镜像
- Debian for RV64 的 D1 哪吒开发板系统镜像

注: 后续的配置主要基于 Linux for x86-64 系统环境, 如果同学采用 Linux for RV64 环境, 需要自己配置。不过在同学比较熟悉的情况下, 配置方法类似且更加简单。可能存在的主要问题是, 面向 Linux for RV64 的相关软件包可能不全, 这样需要同学从源码直接编译出缺失的软件包。

C 开发环境配置

在实验或练习过程中, 也会涉及部分基于 C 语言的开发, 可以安装基本的本机开发环境和交叉开发环境。下面是以 Ubuntu 20.04 为例, 需要安装的 C 开发环境涉及的软件:

```
$ sudo apt-get update && sudo apt-get upgrade
$ sudo apt-get install git build-essential gdb-multiarch qemu-system-misc gcc-riscv64-  
-linux-gnu binutils-riscv64-linux-gnu
```

注: 上述软件不是 Rust 开发环境所必须的。且 ubuntu 20.04 的 QEMU 软件版本低, 而本书实验需要安装 7.0 以上版本的 QEMU。

Rust 开发环境配置

首先安装 Rust 版本管理器 rustup 和 Rust 包管理器 cargo, 这里我们用官方的安装脚本来安装:

```
curl https://sh.rustup.rs -sSf | sh
```

如果通过官方的脚本下载失败了, 可以在浏览器的地址栏中输入 <https://sh.rustup.rs> 来下载脚本, 在本地运行即可。

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 (修改为中国科学技术大学的镜像服务器) 来加速:

```
export RUSTUP_DIST_SERVER=https://mirrors.ustc.edu.cn/rust-static
export RUSTUP_UPDATE_ROOT=https://mirrors.ustc.edu.cn/rust-static/rustup
curl https://sh.rustup.rs -sSf | sh
```

或者使用 tuna 源来加速 参见 rustup 帮助:

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

或者也可以通过在运行前设置命令行中的科学上网代理来实现:

```
# e.g. Shadowsocks 代理, 请根据自身配置灵活调整下面的链接
export https_proxy=http://127.0.0.1:1080
export http_proxy=http://127.0.0.1:1080
export ftp_proxy=http://127.0.0.1:1080
```

安装完成后，我们可以重新打开一个终端来让之前设置的环境变量生效。我们也可以手动将环境变量设置应用到当前终端，只需要输入以下命令：

```
source $HOME/.cargo/env
```

接下来，我们可以确认一下我们正确安装了 Rust 工具链：

```
rustc --version
```

可以看到当前安装的工具链的版本。

```
rustc 1.62.0-nightly (1f7fb6413 2022-04-10)
```

警告：目前用于操作系统实验开发的 rustc 编译器的版本不局限在 1.46.0 这样的数字上，你可以选择更新版本的 rustc 编译器。但注意只能用 rustc 的 nightly 类型的版本。

可通过如下命令安装 rustc 的 nightly 版本，并把该版本设置为 rustc 的缺省版本。

```
rustup install nightly
rustup default nightly
```

我们最好把软件包管理器 cargo 所用的软件包镜像地址 crates.io 也换成中国科学技术大学的镜像服务器来加速三方库的下载。我们打开（如果没有就新建）`~/.cargo/config` 文件，并把内容修改为：

```
[source.crates-io]
registry = "https://github.com/rust-lang/crates.io-index"
replace-with = 'ustc'
[source.ustc]
registry = "git://mirrors.ustc.edu.cn/crates.io-index"
```

同样，也可以使用 tuna 源 参见 crates.io 帮助：

```
[source.crates-io]
replace-with = 'tuna'

[source.tuna]
registry = "https://mirrors.tuna.tsinghua.edu.cn/git/crates.io-index.git"
```

接下来安装一些 Rust 相关的软件包

```
rustup target add riscv64gc-unknown-none-elf
cargo install cargo-binutils
rustup component add llvm-tools-preview
rustup component add rust-src
```

警告：如果你换了另外一个 rustc 编译器（必须是 nightly 版的），需要重新安装上述 rustc 所需软件包。rCore-Tutorial 仓库中的 `Makefile` 包含了这些工具的安装，如果你使用 `make run` 也可以不手动安装。

至于 Rust 开发环境, 推荐 JetBrains Clion + Rust 插件或者 Visual Studio Code 搭配 rust-analyzer 和 RISC-V Support 插件。

注解:

- JetBrains Clion 是付费商业软件, 但对于学生和教师, 只要在 JetBrains 网站注册账号, 可以享受一定期限 (半年左右) 的免费使用的福利。
 - Visual Studio Code 是开源软件, 不用付费就可使用。
 - 当然, 采用 VIM, Emacs 等传统的编辑器也是没有问题的。
-

QEMU 模拟器安装

我们需要使用 QEMU 7.0 版本进行实验, 低版本的 QEMU 可能导致框架代码不能正常运行。而很多 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。下面以 Ubuntu 18.04/20.04 上的安装流程为例进行说明:

首先我们安装依赖包, 获取 QEMU 源代码并手动编译:

```
# 安装编译所需的依赖包
sudo apt install autoconf automake autotools-dev curl libmpc-dev libmpfr-dev libgmp-dev \
  gawk build-essential bison flex texinfo gperf libtool patchutils bc \
  zlib1g-dev libexpat-dev pkg-config libglib2.0-dev libpixman-1-dev \
  libSDL2-dev libslirp-dev \
  git tmux python3 python3-pip ninja-build

# 下载源码包
# 如果下载速度过慢可以使用我们提供的百度网盘链接: https://pan.baidu.com/s/1dykndFzY73ngkPLQXs32Q
# 提取码: jmc
wget https://download.qemu.org/qemu-7.0.0.tar.xz
# 解压
tar xvJf qemu-7.0.0.tar.xz
# 编译安装并配置 RISC-V 支持
cd qemu-7.0.0
./configure --target-list=riscv64-softmmu,riscv64-linux-user #_
  在第九章的实验中, 可以有图形界面和网络。如果要支持图形界面, 可添加 "--enable-sdl"_
  参数; 如果要支持网络, 可添加 "--enable-slirp" 参数
make -j$(nproc)
```

注解: 注意, 上面的依赖包可能并不完全, 比如在 Ubuntu 18.04 上:

- 出现 ERROR: pkg-config binary 'pkg-config' not found 时, 可以安装 pkg-config 包;
- 出现 ERROR: glib-2.48 gthread-2.0 is required to compile QEMU 时, 可以安装 libglib2.0-dev 包;
- 出现 ERROR: pixman >= 0.21.8 not present 时, 可以安装 libpixman-1-dev 包。

另外一些 Linux 发行版编译 QEMU 的依赖包可以从 [这里](#) 找到。

之后我们可以在同目录下 sudo make install 将 QEMU 安装到 /usr/local/bin 目录下, 但这样经常会引起冲突。个人来说更习惯的做法是, 编辑 ~/.bashrc 文件 (如果使用的是默认的 bash 终端), 在文件的末尾加入几行:

```
# 请注意, qemu-7.0.0 的父目录可以随着你的实际安装位置灵活调整
export PATH=$PATH:/path/to/qemu-7.0.0/build
```

随后即可在当前终端 source ~/.bashrc 更新系统路径, 或者直接重启一个新的终端。

此时我们可以确认 QEMU 的版本:

```
qemu-system-riscv64 --version
qemu-riscv64 --version
```

在其他缺少预编译 QEMU with RV64 软件包的 Linux x86-64 环境 (如 openEuler 操作系统) 上, 首先需要从 openEuler 社区维护的 QEMU 的 riscv 分支 下载 QEMU 源码, 并直接通过 rpmbuild 进行构建。

警告: 请尽量不要安装 qemu-kvm, 这可能会导致我们的框架无法正常运行。如果已经安装, 可以考虑换用 Docker。

另外, 我们仅在 Qemu 7.0.0 版本上进行了测试, 请尽量不要切换到其他版本。

K210 真机串口通信

为了能在 K210 真机上运行 Tutorial, 我们还需要安装基于 Python 的串口通信库和简易的串口终端。

```
pip3 install pyserial
sudo apt install python3-serial
```

GDB 调试支持

目前我们仅支持基于 QEMU 模拟器进行调试。在 os 目录下 make debug 可以调试我们的内核, 这需要安装终端复用工具 tmux, 还需要支持 riscv64 指令集的 gdb 调试器 riscv64-unknown-elf-gdb。该调试器包含在 riscv64 gcc 工具链中, 工具链的预编译版本可以在如下链接处下载:

- Ubuntu 环境
- macOS 环境
- Windows 环境
- CentOS 环境

最新版的工具链可以在 sifive 官方的 repo 中 找到。

解压后在 bin 目录下即可找到 riscv64-unknown-elf-gdb 以及另外一些常用工具 objcopy/objdump/readelf 等。

在其他缺少预编译 riscv64 gcc 工具链的 Linux x86-64 环境 (如 openEuler 操作系统、龙蜥操作系统等) 上, 则需要 clone riscv 工具链仓库 并参考其说明手动构建。

出于某些原因, 我们全程使用 release 模式进行构建。为了正常进行调试, 请确认各项目 (如 os, user 和 easy-fs) 的 Cargo.toml 中包含如下配置:

```
[profile.release]
debug = true
```

此外, 参考 os/Makefile, 还可以先打开一个终端页面 make gdbserver 启动 QEMU, 此后另开一个终端页面在同目录下 make gdbclient 将 GDB 客户端连接到 QEMU 进行调试。我们推荐使用 gdb-dashboard 插件, 可以大大提升调试体验。在本节的评论区已有同学提供了基于各种 IDE 的调试方法, 也可参考。

1.6.3 运行 rCore-Tutorial-v3

在 QEMU 模拟器上运行

如果是在 QEMU 模拟器上运行，只需在 os 目录下 make run 即可。在内核加载完毕之后，可以看到目前可以用的应用程序。usertests 打包了其中的很大一部分，所以我们可以运行它，只需输入在终端中输入它的名字即可。

```
(base) shinbokuow@shinbokuow-virtual-machine:~/workspace/v3/rCore-Tutorial-v3$ cd os
(base) shinbokuow@shinbokuow-virtual-machine:~/workspace/v3/rCore-Tutorial-v3/os$ make run
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
cargo install cargo-binutils
  Updating crates.io index
    Ignored package `cargo-binutils v0.3.3` is already installed, use --force to override
rustup target add riscv64gc-unknown-none-elf
info: component 'rust-std' for target 'riscv64gc-unknown-none-elf' is up to date
  Finished release [optimized] target(s) in 0.03s
```

之后，可以先按下 **Ctrl+a** (即：先按下 **Ctrl** 不松开，再按下小写字母 **a** 不放，随后同时将两个键松开)，再按下 **x** 来退出 QEMU。

在 K210 平台上运行

如果是在 K210 平台上运行则略显复杂。

首先，我们需要将 MicroSD 插入 PC 来将文件系统镜像拷贝上去。

```
(base) shinbokuow@shinbokuow-virtual-machine:~/workspace/v3/rCore-Tutorial-v3$ cd os
(base) shinbokuow@shinbokuow-virtual-machine:~/workspace/v3/rCore-Tutorial-v3/os$ make sdcard
[sudo] password for shinbokuow:
16+0 records in
16+0 records out
16777216 bytes (17 MB, 16 MiB) copied, 1.71198 s, 9.8 MB/s
3192+0 records in
3192+0 records out
4194304 bytes (4.2 MB, 4.0 MiB) copied, 3.3986 s, 1.2 MB/s
(base) shinbokuow@shinbokuow-virtual-machine:~/workspace/v3/rCore-Tutorial-v3/os$
```

警告: 在 os/Makefile 中我们默认设置 MicroSD 在当前操作系统中可以用设备 SDCARD=/dev/sdb 访问。你可以使用 df -hT 命令来确认在你的环境中 MicroSD 是哪个设备，并在 make sdcard 之前对 os/Makefile 的 SDCARD 配置做出适当的修改。不然，这有可能导致设备 /dev/sdb 上数据丢失！

随后，我们将 MicroSD 插入 K210 开发板，将 K210 开发板连接到 PC，然后进入 os 目录 make run BOARD=k210 在 K210 开发板上跑 rCore Tutorial。

```
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
cargo install cargo-binutils
  Updating crates.io index
    Ignored package `cargo-binutils v0.3.3` is already installed, use --force to override
rustup target add riscv64gc-unknown-none-elf
info: component 'rust-std' for target 'riscv64gc-unknown-none-elf' is up to date
  Compiling os v0.1.0 (/home/shinbokuw/workspace/v3/rCore-Tutorial-v3/os)
    Finished release [optimized] target(s) in 4.06s
0+1 records in
0+1 records out
128520 bytes (129 kB, 126 KiB) copied, 0.000448575 s, 287 MB/s
python3 ./tools/kflash.py -p /dev/ttyUSB0 -b 1500000 target/riscv64gc-unknown-none-elf/release/os.bin
[INFO] COM Port Selected Manually: /dev/ttyUSB0
[INFO] Default baudrate is 115200, later it may be changed to the value you set.
[INFO] Trying to Enter the ISP Mode...
.
[INFO] Automatically detected dan/bit/trainer

[INFO] Greeting Message Detected, Start Downloading ISP
Downloading ISP: |=====-----| 7.7%
```

之后，可以按下 **Ctrl+]** 来退出串口终端。

由于教程的 ch1~ch5 分支还没有文件系统，在 K210 上运行这些分支无需 MicroSD 卡也不需要进行文件系统镜像烧写工作，直接切换到 os 目录下 make run BOARD=k210 即可。

到这里，恭喜你完成了实验环境的配置，可以开始阅读教程的正文部分了！

1.6.4 Q & A

当代码跑不起来的时候，可以尝试：

- 分支是否与 rCore-Tutorial-v3 原版仓库（而非 fork 出来的仓库）的对应分支同步。如不同步的话考虑通过 git pull 进行更新。注：这是因为 Rust 的版本更迭较快，如不及时更新的话曾经能正常运行的代码也会无法运行。
- 项目根目录下的 rust-toolchain 非常重要，它代表整个项目采用的 Rust 工具链版本。请务必保持其与原版仓库对应分支一致。
- 项目根目录下是否存在放置 RustSBI 的 bootloader/ 目录。如不存在的话可从原版仓库的各分支上获取。
- 通过 make clean 或者 cargo clean 删除 os 或 user 目录下的构建产物，并重新 make run。注：出现这样的问题通常说明框架的构建脚本存在 bug，可以提 issue。

如果怀疑遇到了网络问题，可以检查：

- 请按照本节说明进行 Rust 安装和 crates.io 镜像配置。通常情况下能够解决 Rust 工具链更新和下载已发布到 crates.io 上库的问题。
- 如果发现在试图从 github 上下载下述几个库的时候卡死，可以修改 os 和 user 目录下的 Cargo.toml 替换为 gitee 上的镜像。例如，将：

```

riscv = { git = "https://github.com/rcore-os/riscv", features = ["inline-"
    ↵asm" ] }
virtio-drivers = { git = "https://github.com/rcore-os/virtio-drivers" }
k210-pac = { git = "https://github.com/wyfcyx/k210-pac" }
k210-hal = { git = "https://github.com/wyfcyx/k210-hal" }
k210-soc = { git = "https://github.com/wyfcyx/k210-soc" }

```

替换为：

```

riscv = { git = "https://gitee.com/rcore-os/riscv", features = ["inline-"
    ↵asm" ] }
virtio-drivers = { git = "https://gitee.com/rcore-os/virtio-drivers" }
k210-pac = { git = "https://gitee.com/wyfcyx/k210-pac" }
k210-hal = { git = "https://gitee.com/wyfcyx/k210-hal" }
k210-soc = { git = "https://gitee.com/wyfcyx/k210-soc" }

```

1.7 练习

1.7.1 课后练习

课后练习和实验练习中的题目难度表示：

- * 容易
- ** 有一定工作量
- *** 有难度

编程题

1. * 在你日常使用的操作系统环境中安装并配置好实验环境。简要说明你碰到的问题/困难和解决方法。
2. * 在 Linux 环境下编写一个会产生异常的应用程序，并简要解释操作系统的处理结果。
3. ** 在 Linux 环境下编写一个可以睡眠 5 秒后打印出一个字符串，并把字符串内容存入一个文件中的应用程序 A。(基于 C 或 Rust 语言)
4. *** 在 Linux 环境下编写一个应用程序 B，简要说明此程序能够体现操作系统的并发性、异步性、共享性和持久性。(基于 C 或 Rust 语言)

注：在类 Linux 环境下编写尝试用 GDB 等调试工具调试应用程序 A，能够设置断点，单步执行，显示变量信息。

问答题

1. * 什么是操作系统？操作系统的主要目标是什么？
2. * 面向服务器的操作系统与面向手机的操作系统在功能上有何异同？
3. * 对于目前的手机或桌面操作系统而言，操作系统是否应该包括网络浏览器？请说明理由。
4. * 操作系统的核心抽象有哪些？它们应对的对象是啥？
5. * 操作系统与应用程序之间通过什么来进行互操作和数据交换？

6. * 操作系统的特征是什么？请结合你日常使用的操作系统的具体运行情况来进一步说明操作系统的特征。
7. * 请说明基于 C 语言应用的执行环境与基于 Java 语言应用的执行环境的异同。
8. ** 请简要列举操作系统的系统调用的作用，以及简要说明与程序执行、内存分配、文件读写相关的 Linux 系统调用的大致接口和含义。
9. ** 以你编写的可以睡眠 5 秒后打印出一个字符串的应用程序 A 为例，说明什么是控制流？什么是异常控制流？什么是进程、地址空间和文件？并简要描述操作系统是如何支持这个应用程序完成其工作并结束的。
10. * 请简要描述支持单个应用的 OS、批处理 OS、多道程序 OS、分时共享 OS 的特点。

1.8 练习参考答案

1.8.1 编程题

1. * 在你日常使用的操作系统环境中安装并配置好实验环境。简要说明你碰到的问题/困难和解决方法。
2. * 在 Linux 环境下编写一个会产生异常的应用程序，并简要解释操作系统的处理结果。

例如，对于这样一段 C 程序，其中包含一个除以零的操作：

```

1 #include <stdio.h>
2
3 int main() {
4     printf("1 / 0 = %d", 1 / 0);
5     return 0;
6 }
```

在基于 x86-64 Linux 的环境下编译运行结果如下：

```

$ gcc divzero
$ ./divzero
Floating point exception (core dumped)
```

程序接到了一个“浮点数异常”而异常终止。使用 strace 可以看到更详细的信号信息：

```

$ strace ./divzero
[... 此处省略系统调用跟踪输出]
--- SIGFPE {si_signo=SIGFPE, si_code=FPE_INTDIV, si_addr=0x401131} ---
+++ killed by SIGFPE (core dumped) +++
```

用 gdb 的 disassemble 命令可以看到发生异常的指令

```

$ gdb divzero
Reading symbols from divzero...
(gdb) r
Starting program: [...]/divzero

Program received signal SIGFPE, Arithmetic exception.
0x0000000000401131 in main ()
(gdb) disassemble
Dump of assembler code for function main:
0x0000000000401122 <+0>:    push    %rbp
0x0000000000401123 <+1>:    mov     %rsp,%rbp
```

(下页继续)

(续上页)

```

0x00000000000401126 <+4>:    mov    $0x1,%eax
0x0000000000040112b <+9>:    mov    $0x0,%ecx
0x00000000000401130 <+14>:   cltd
=> 0x00000000000401131 <+15>:  idiv   %ecx
0x00000000000401133 <+17>:   mov    %eax,%esi
0x00000000000401135 <+19>:   mov    $0x402004,%edi
0x0000000000040113a <+24>:   mov    $0x0,%eax
0x0000000000040113f <+29>:   call   0x401030 <printf@plt>
0x00000000000401144 <+34>:   mov    $0x0,%eax
0x00000000000401149 <+39>:   pop    %rbp
0x0000000000040114a <+40>:   ret

```

可以看出，应用程序在执行 *idiv* 指令（有符号除法指令）时发生了除以零异常，跳转至操作系统处理。操作系统把它转换为一个信号 *SIGFPE*，使用信号处理机制处理这个异常。该程序收到 *SIGFPE* 时应发生的行为是异常终止，于是操作系统将其终止，并将异常退出的信息报告给 shell 进程。

需要注意的是，异常的处理和与信号的对应是与架构相关的。例如，RISC-V 架构下除以零不是异常，而是有个确定的结果。此外，不同架构下具体异常和信号的对应关系也是不同的，甚至有些混乱（例如，这里明明是整数除以零错误，却报告了“浮点数”异常）。

- ** 在 Linux 环境下编写一个可以睡眠 5 秒后打印出一个字符串，并把字符串内容存入一个文件中的应用程序 A。（基于 C 或 Rust 语言）

样例实现如下（未包含错误处理）

```

#include <stdio.h>
#include <unistd.h>

int main() {
    sleep(5);

    const char* hello_string = "Hello Linux!\n";

    printf(hello_string);

    FILE *output_file = fopen("output.txt", "w");
    fputs(hello_string, output_file);
    fclose(output_file);

    return 0;
}

```

编译运行，查看程序的输出和文件，结果如下：

```

$ gcc -o program-a program-a.c
$ ./program-a
Hello Linux!          [这一行等待 5s 后才输出]
$ cat output.txt
Hello Linux!

```

- *** 在 Linux 环境下编写一个应用程序 B，简要说明此程序能够体现操作系统的并发性、异步性、共享性和持久性。（基于 C 或 Rust 语言）

注：在类 Linux 环境下编写尝试用 GDB 等调试工具调试应用程序 A，能够设置断点，单步执行，显示变量信息。

1.8.2 问答题

1. * 什么是操作系统？操作系统的主要目标是什么？
2. * 面向服务器的操作系统与面向手机的操作系统在功能上有何异同？
 - 相同部分举例：
 - 都需要进行进程的管理
 - 都需要管理 CPU, 内存这些基本资源
 - 都需要隔离保护等安全机制
 - (服务器和手机的操作系统共通之处很多，甚至可以共用最基础的部分，比如都用 Linux 内核)
 - 不同部分举例：
 - 支持的外设不同：
 - * 服务器：高性能网卡（如光纤）、存储设备（如 HBA）等
 - * 手机：屏幕，音响和话筒，基带（移动网络）、WiFi 等
 - 服务器和手机都有的资源，管理起来也有区别：
 - * 服务器：内存很多，CPU 核数很多，可能需要支持更复杂的硬件拓扑，如 NUMA 架构
 - * 手机：对低功耗的要求，需要操作系统在调度时考虑到电源管理，平衡省电和性能
 - 应用不同，导致对安全性要求不同：
 - * 服务器：上千用户，少数几个应用，需要各种阻止用户利用漏洞操控服务器上的程序的功能
 - * 手机：一个或少数几个用户，几十个应用，需要隐私保护功能
3. * 对于目前的手机或桌面操作系统而言，操作系统是否应该包括网络浏览器？请说明理由。
4. * 操作系统的核心抽象有哪些？它们应对的对象是啥？
 - 进程/线程-CPU 时间
 - 地址空间-内存
 - 执行环境-CPU 上复杂的环境（有中断异常等），和操作系统提供的功能（如系统调用）
 - 文件和文件描述符-存储和输入输出设备
5. * 操作系统与应用程序之间通过什么来进行互操作和数据交换？
 - 互操作的方式：
 - 应用程序调用系统调用主动让操作系统进行操作
 - 操作系统在中断异常发生时强制暂停应用程序进行相关操作
 - 数据交换的方式：
 - 系统调用时根据 ABI 规定在（比如）寄存器中传递参数
 - 复制数据：在内核占用的空间和用户占用的空间之间互相复制数据，如读写文件的时候从应用程序给出的缓冲区复制写的数据，或者复制读的数据到缓冲区
 - (共享内存空间：如 *io_uring*)

6. *操作系统的特征是什么？请结合你日常使用的操作系统的具体运行情况来进一步说明操作系统的特征。

以在普通的桌面上运行基于 Linux 的平台上做操作系统实验时可能发生的事情为例：

- 虚拟性：如果物理内存相对不足，Linux 的 swap 机制会将不常用的内存内容转存到硬盘上，优先保证活跃的进程可以高速访问物理内存。
- 并发性：正在运行的程序数量（包括并发运行的 *gcc* 或 *rustc*，和其它程序）可以超过 CPU 核心数，由操作系统来调度分配使用 CPU 时间。
- 异步性：并行运行的编译器太多，可能会影响写代码用的文本编辑器性能，因为操作系统安排了更多时间运行编译器而不是文本编辑器进程。
- 共享性：写代码和编译可以共享同一个文件系统，在同一块硬盘上读写，操作系统的文件系统相关模块（VFS）会调用文件系统实现和硬盘驱动安排这些读写操作。
- 持久性：操作系统实验用的文件保存在文件系统中，晚上宿舍断电关机，明天早上起来可以从昨天保存的文件的状态继续工作。

7. *请说明基于 C 语言应用的执行环境与基于 Java 语言应用的执行环境的异同。

8. **请简要列举操作系统的系统调用的作用，以及简要说明与程序执行、内存分配、文件读写相关的 Linux 系统调用的大致接口和含义。

- 系统调用的作用：
 - 将操作系统实现和用户程序调用方式分开，作为一个抽象层方便开发和使用
 - 在让用户程序没法直接访问别的用户程序的地址空间和操作系统地址空间的情况下，作为一个统一的应用程序请求操作系统服务的入口，方便安全检查和审计。
 - 让用户程序能有限访问其不能直接访问的计算机资源和操作系统对象
- 在 Linux 中：
 - *clone*‘（曾用的有 *fork*）创建进程；*execve* 传入文件路径、命令行参数、环境变量，加载新的程序替换当前进程。
 - *brk* 修改或获取当前的堆顶的位置，增加堆顶地址分配空间，减小堆底地址释放空间；*mmap* 用于映射地址空间，也可以分配物理内存（*MAP_ANONYMOUS*），*munmap* 释放对应的地址空间和对应的内存。
 - 文件读写相关有 *openat*‘（曾用的有 *open*）打开文件，*read* 读文件，*write* 写文件等。

9. **以你编写的可以睡眠 5 秒后打印出一个字符串的应用程序 A 为例，说明什么是控制流？什么是异常控制流？什么是进程、地址空间和文件？并简要描述操作系统是如何支持这个应用程序完成其工作并结束的。

应用程序 A 中体现的操作系统相关概念：

- 控制流：*main* 函数内一行一行往下执行，是一个顺序控制流；其中调用的标准库函数，是函数调用的控制流。
- 异常控制流：在 *sleep* 的时候，执行系统调用 *clock_nanosleep*，此时控制流跳出了该程序，进入了操作系统的系统调用实现代码，之后操作系统将应用程序 A 进入睡眠状态，转而运行别的进程，这里体现了异常控制流。在执行其它系统调用的时候也会有类似的情况。
- 进程：整个应用程序 A 是在一个新的进程中运行的，与启动它的 shell。
- 地址空间：字符串 *hello_string* 所在的地址，是在这个进程自己的地址空间内有效的，和别的进程无关。
- 文件：应用程序 A 打开文件名为 *output.txt* 的文件，向其中写入了一个字符串。此外，*printf* 是向标准输出写入，标准输出在此时是一个对应当前终端的文件。

操作系统支持应用程序 A 运行的流程:

- 操作系统加载程序和 C 语言标准库到内存中
- 操作系统设置一些初始的虚拟内存配置和一些数据, 然后切换到用户态跳转到程序的入口点开始运行。
- 应用程序调用 C 语言标准库, 库再进行系统调用, 进入内核, 内核处理相关的系统调用然后恢复应用程序运行, 具体来说:
 - 对于 `sleep` 来说, 操作系统在 5 秒时间内切换到别的任务, 不运行应用程序 A, 5 秒过后再继续运行
 - 对于文件来说, 操作系统在文件系统内找到对应的文件写入, 或者找到对应的终端将写入的内容发送过去
- 最后 C 语言标准库会调用 `exit_group` 系统调用退出进程。操作系统接收这个系统调用后, 不再回到应用程序, 释放该进程相关资源。

10. * 请简要描述支持单个应用的 OS、批处理 OS、多道程序 OS、分时共享 OS 的特点。

第一章：应用程序与基本执行环境

2.1 引言

2.1.1 本章导读

本章展现了操作系统的一个基本目标：让应用与硬件隔离，简化了应用访问硬件的难度和复杂性。这也是远古操作系统雏形和现代的一些简单嵌入式操作系统的主要功能。具有这样功能的操作系统形态就是一个函数库，可以被应用访问，并通过函数库的函数来访问硬件。

大多数程序员的第一行代码都从 `Hello, world!` 开始，当我们满怀着好奇心在编辑器内键入仅仅数个字节，再经过几行命令编译（靠的是编译器）、运行（靠的是操作系统），终于在黑洞洞的终端窗口中看到期望中的结果的时候，一扇通往编程世界的大门已经打开。在本章第一节[应用程序执行环境与平台支持](#)中，可以看到用 Rust 语言编写的非常简单的“`Hello, world`”应用程序是如何被进一步拆解和分析的。

不过我们能够隐约意识到编程工作能够如此方便简洁并不是理所当然的，实际上有着多层硬件和软件工具和支撑环境隐藏在它背后，才让我们不必付出那么多努力就能够创造出功能强大的应用程序。生成应用程序二进制执行代码所依赖的是以 **编译器** 为主的 **开发环境**；运行应用程序执行码所依赖的是以 **操作系统** 为主的 **执行环境**。

本章主要是讲解如何设计和实现建立在裸机上的执行环境，并让应用程序能够在这样的执行环境中运行。从而让同学能够对应用程序和它所依赖的执行环境有一个全面和深入的理解。

本章的目标仍然只是让应用程序输出 `Hello, world!` 字符串，但这一次，我们将离开舒适区，基于一个几乎空无一物的硬件平台从零开始搭建我们自己的软件高楼大厦，而不是仅仅通过一行语句就完成任务。所以，在接下来的内容中，我们将描述如何让 `Hello, world!` 应用程序逐步脱离对编译器、运行时库和操作系统的现有复杂依赖，最终以最小的依赖需求能在裸机上运行。这时，我们也可把这个能在裸机上运行的 `Hello, world!` 应用程序所依赖的软件库称为一种支持输出字符串的非常初级的寒武纪“三叶虫”操作系统—LibOS。LibOS 其实就是一个给应用提供各种服务（比如输出字符串）的库，方便了单一应用程序在裸机上的开发与运行。输出字符串的功能好比是三叶虫的眼睛功能，有了它，我们就有了对软件的最基本的动态分析与调试功能，即通过在代码中的不同位置插入特定内容的输出语句来实现对应用程序和操作系统运行状态的分析与调试。

注解：最早的操作系统雏形是计算工资单的程序库

操作系统需要给程序员提供支持：高效便捷地开发应用和执行应用。远古时期的计算机硬件昂贵笨重，能力弱，单靠硬件还不能高效地执行应用，能够减少程序员的开发成本就已经很不错了。

程序库一般由一些子程序（函数）组成。通过调用程序库中的子程序，应用程序可以更加方便的实现其应用功能。但在早期的软件开发中，还缺少便捷有效的子程序调用机制。

根据维基百科的操作系统时间线¹ 上的记录，1949-1951 年，英国 J. Lyons and Co. 公司（一家包括连锁餐厅和食品制造的大型集团公司）开创性地引入并使用剑桥大学的 EDSAC 计算机，联合设计实现了 LEO I ‘Lyons Electronic Office’ 软硬件系统，利用计算机的高速度（按当时的标准）来高效地计算薪资，以及组织蛋糕和其他易腐烂的商品的分配等。这样计算机就成为了一个高效的专用事务处理系统。但软件开发还是一个很困难的事情，需要减少软件编程人员的开发负担。而通过函数库来重用软件功能并简化应用的编程是当时自然的想法。但在软件编程中，由于硬件的局限性（缺少索引寄存器、保存函数返回地址的寄存器、栈寄存器、硬件栈等），早期的程序员不得不使用在程序中修改自身代码的方式来访问数组或调用函数。从现在的视角看来，这样具有自修改能力的程序是一种黑科技。

参与 EDSAC 项目的 David Wheeler 发明了子程序的概念—**Wheeler Jump**。Wheeler 的方法是在子程序的最后一行添加 “**jump to this address**” 指令，并在指令后跟一个内存空间，这个内存空间通常被设置为 0，在子程序被调用后，这个内存空间的值会被修改为返回地址。当调用子程序时，调用者（Caller）的地址将被放置在累加寄存器中，然后代码将跳转到子程序的入口。子程序的第一条指令将根据累加寄存器中的值计算返回地址，通常是调用指令的下一条指令所在的内存位置，然后将计算出的返回地址写入先前预留的内存空间中。当子程序继续执行，自然会到达子程序的末尾，即 “**jump to this address**” 指令处，这条指令读取位于它之后的内存单元，获得返回地址，就可以正常返回了。

在有了便捷有效的子程序概念和子程序调用机制后，软件开发人员在 EDSAC 计算机开发了大量的子程序库，其中就包括了检查计算机系统，加载应用软件，写数据到持久性存储设备中，打印数据等硬件系统相关功能的系统子程序库。这样程序员就可以方便开发应用程序来使用计算机了。这也是为何维基百科的操作系统时间线 Page 10, 1 一文中，把 LEO I ‘Lyons Electronic Office’ 软件系统（其实就是硬件系统相关的子程序库）定位为最早（1951 年）的操作系统的起因。这样的计算机系统只支持一个应用的运行，可以称为专用计算机系统。1951 年 9 月 5 日，计算机首次执行了一个名为 *Bakeries Valuations* 的应用程序，并在后续承担计算工资单这一必须按时执行的任务，因为必须向员工按时支付周薪。计算员工薪酬的任务需要一位经验丰富的文员 8 分钟内完成，而 LEO I 在 1.5 秒内完成了这项工作，快了 320 倍，这在当时英国社会上引起了轰动。

即使到了现在，以子程序库形式存在的简单嵌入式操作系统大量存在，运行在很多基于微控制单元（Micro-controller Unit，简称 MCU）的单片机中，并支持简单应用甚至是单一应用，在智能仪表、玩具、游戏机、小家电等领域广泛存在。

2.1.2 实践体验

本章设计实现了一个支持显示字符串应用的简单操作系统—“三叶虫”操作系统—LibOS，它的形态就是一个函数库，给应用程序提供了显示字符串的函数。

获取本章代码：

```
$ git clone https://github.com/rcore-os/rCore-Tutorial-v3.git
$ cd rCore-Tutorial-v3
$ git checkout ch1
```

在 Qemu 模拟器上运行本章代码，看看一个小应用程序是如何在 Qemu 模拟的计算机上运行的：

```
$ cd os
$ LOG=TRACE make run
```

LOG=TRACE 是指定 LOG 的级别为 TRACE，可以查看重要程度不低于 TRACE 的输出日志。目前 TRACE 的重要程度最低，因此这样能够看到全部日志。

¹ https://en.wikipedia.org/wiki/Timeline_of_operating_systems

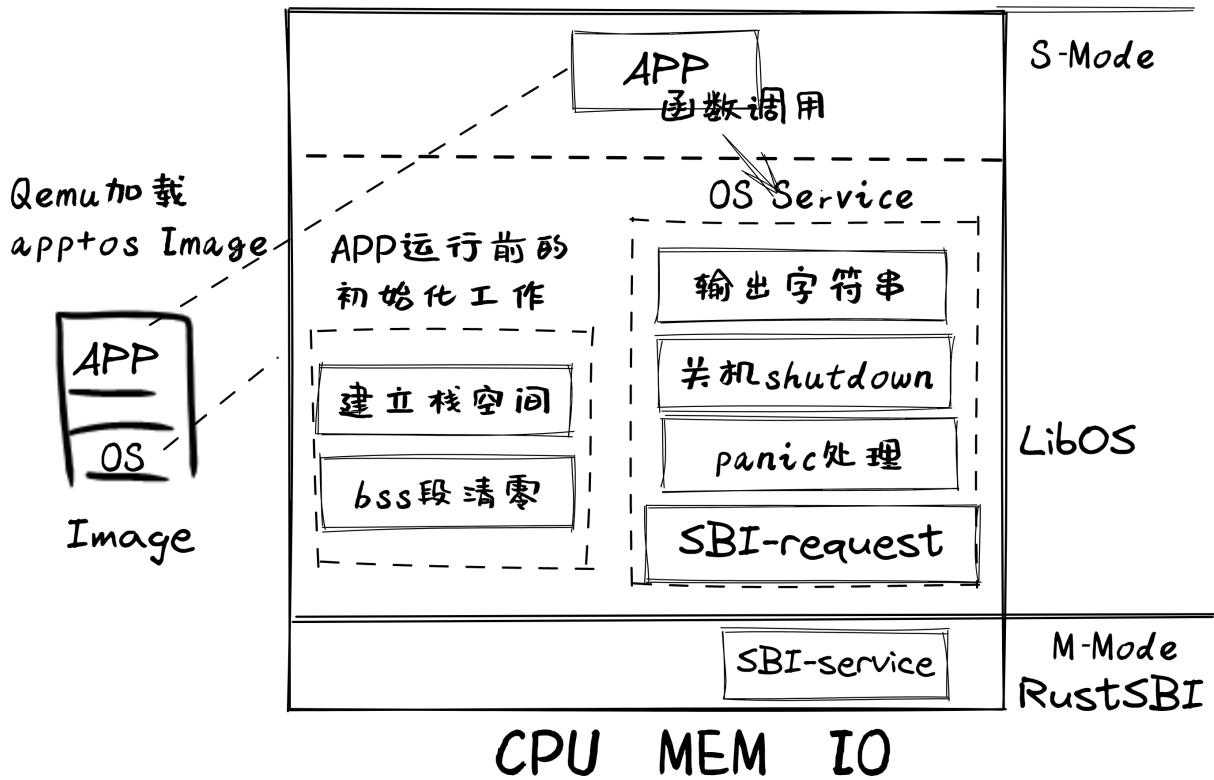
如果顺利的话, 以 Qemu 平台为例, 将输出:

```
[RustSBI output]
[rustsbi] RustSBI version 0.3.1, adapting to RISC-V SBI v1.0.0
[rustsbi] Implementation      : RustSBI-QEMU Version 0.2.0-alpha.2
[rustsbi] Platform Name       : riscv-virtio,qemu
[rustsbi] Platform SMP        : 1
[rustsbi] Platform Memory     : 0x80000000..0x88000000
[rustsbi] Boot HART          : 0
[rustsbi] Device Tree Region : 0x87000000..0x87000ef2
[rustsbi] Firmware Address    : 0x80000000
[rustsbi] Supervisor Address : 0x80200000
[rustsbi] pmp01: 0x00000000..0x80000000 (-wr)
[rustsbi] pmp02: 0x80000000..0x80200000 (---)
[rustsbi] pmp03: 0x80200000..0x88000000 (xwr)
[rustsbi] pmp04: 0x88000000..0x00000000 (-wr)
[kernel] Hello, world!
[TRACE] [kernel] .text [0x80200000, 0x80203000)
[DEBUG] [kernel] .rodata [0x80203000, 0x80205000)
[INFO] [kernel] .data [0x80205000, 0x80206000)
[WARN] [kernel] boot_stack top=bottom=0x80216000, lower_bound=0x80206000
[ERROR] [kernel] .bss [0x80216000, 0x80217000)
```

Hello, world! 前后有一些额外的动态运行信息, 最后是一系列 kernel 的输出日志。

2.1.3 本章代码树

三叶虫 LibOS 操作系统的总体结构如下图所示:



通过上图，大致可以看出 Qemu 把包含 app 和三叶虫 LibOS 的 image 镜像加载到内存中，RustSBI (bootloader) 完成基本的硬件初始化后，跳转到三叶虫 LibOS 起始位置，三叶虫 LibOS 首先进行 app 执行前的初始化工作，即建立栈空间和清零 bss 段，然后跳转到 app 去执行。app 在执行过程中，会通过函数调用的方式得到三叶虫 LibOS 提供的 OS 服务，如输出字符串等，避免了 app 与硬件直接交互的繁琐过程。

注：图中的 S-Mode 和 M-Mode 是 RISC-V 处理器架构中的两种特权级别。S-Mode 指的是 Supervisor 模式，是操作系统使用的特权级别，可执行特权指令等。M-Mode 是 Machine 模式，其特权级别比 S-Mode 还高，可以访问 RISC-V 处理器中的所有系统资源。关于特权级的进一步描述可以看第二章的[特权级机制](#) 中的详细说明。

位于 ch1 分支上的三叶虫 LibOS 操作系统的源代码如下所示：

```

./os/src
Rust      4 Files    119 Lines
Assembly  1 Files     11 Lines

├── bootloader(内核依赖的运行在 M 特权级的 SBI 实现，本项目中我们使用 RustSBI)
│   └── rustsbi-qemu.bin(可运行在 qemu 虚拟机上的预编译二进制版本)
├── LICENSE
├── os(我们的内核实现放在 os 目录下)
│   ├── Cargo.toml(内核实现的一些配置文件)
│   ├── Makefile
│   └── src(所有内核的源代码放在 os/src 目录下)
│       ├── console.rs(将打印字符的 SBI 接口进一步封装实现更加强大的格式化输出)
│       ├── entry.asm(设置内核执行环境的一段汇编代码)
│       ├── lang_items.rs(需要我们提供给 Rust 编译器的一些语义项，目前包含内核 panic_
│       ↪ 时的处理逻辑)
│       └── linker-qemu.ld(控制内核内存布局的链接脚本以使内核运行在 qemu 虚拟机上)
│           └── main.rs(内核主函数)

```

(下页继续)

(续上页)

```

|   └── sbi.rs(调用底层 SBI 实现提供的 SBI 接口)
|   └── README.md
└── rust-toolchain(控制整个项目的工具链版本)

```

注解: 附录 C: 深入机器模式: *RustSBI* 中可以找到关于 RustSBI 的更多信息。

2.1.4 本章代码导读

LibOS 操作系统虽然是软件, 但它不是运行在通用操作系统 (如 Linux) 上的一般应用软件, 而是运行在裸机执行环境中的系统软件。如果采用通常的应用编程方法和编译手段, 无法开发出这样的操作系统。其中一个重要的原因是: 编译器 (Rust 编译器和 C 编译器等) 编译出的应用软件在缺省情况下是要链接标准库, 而标准库是依赖于操作系统 (如 Linux、Windows 等) 的, 但 LibOS 操作系统不依赖其他操作系统。所以, 本章主要是让同学能够脱离常规应用软件开发的思路, 理解如何开发没有操作系统支持的操作系统内核。

为了做到这一步, 首先需要写出不需要标准库的软件并通过编译。为此, 先把一般应用所需要的标准库的组件给去掉, 这会导致编译失败。然后再逐步添加不需要操作系统的极少的运行时支持代码, 让编译器能够正常编译出不需要标准库的正常程序。但此时的程序没有显示输出, 更没有输入等, 但可以正常通过编译, 这样就打下 **可正常编译 OS** 的前期开发基础。具体可看[移除标准库依赖](#)一节的内容。

LibOS 内核主要在 Qemu 模拟器上运行, 它可以模拟一台 64 位 RISC-V 计算机。为了让 LibOS 内核能够正确对接到 Qemu 模拟器上, 需要了解 Qemu 模拟器的启动流程, 还需要一些程序内存布局和编译流程 (特别是链接) 相关知识, 这样才能将 LibOS 内核加载到正确的内存位置上, 并使得它能够在 Qemu 上正常运行。为了确认内核被加载到正确的内存位置, 我们会在 LibOS 内核中手写一条汇编指令, 并使用 GDB 工具监控 Qemu 的执行流程确认这条指令被正确执行。具体可以参考[内核第一条指令 \(基础篇\)](#) 和[内核第一条指令 \(实践篇\)](#) 两节。

我们想用 Rust 语言来实现内核的大多数功能, 因此我们需要进一步将控制权从第一条指令转交给 Rust 入口函数。在 Rust 代码中, 函数调用是不可或缺的基本控制流, 为了使得函数调用能够正常进行, 我们在跳转到 Rust 入口函数前还需要进行栈的初始化工作。为此我们详细介绍了函数调用和栈的相关背景知识, 具体内容可参考[为内核支持函数调用](#) 一节。最终, 我们调用软件栈中相比内核更低一层的软件——也即 RustSBI 提供的服务来实现格式化输出和遇到致命错误时的关机功能, 形成了 LibOS 的核心功能, 详情参考[基于 SBI 服务完成输出和关机](#) 一节。至此, 应用程序可以直接调用 LibOS 提供的字符串输出函数或关机函数, 达到让应用与硬件隔离的操作系统目标。

2.2 应用程序执行环境与平台支持

2.2.1 本节导读

本节我们从一个最简单的 Rust 应用程序入手, 深入地挖掘它下面的多层执行环境, 分析编译器和操作系统为应用程序的开发和运行提供了怎样的便利条件。需要注意的是, LibOS 操作系统也是一层执行环境, 因此它需要为上层的应用程序提供服务。我们会接触到计算机科学中最核心的思想-抽象, 同时以现实中不同需求层级的应用为例分析如何进行合理的抽象。最后, 我们还会介绍软硬件平台 (包括 RISC-V 架构) 的一些基础知识。

2.2.2 执行应用程序

我们先在 Linux 上开发并运行一个简单的“Hello, world”应用程序，看看一个简单应用程序从开发到执行的全过程。作为一切的开始，让我们使用 Cargo 工具来创建一个 Rust 项目。它看上去没有任何特别之处：

```
$ cargo new os --bin
```

我们加上了 `--bin` 选项来告诉 Cargo 我们创建一个可执行程序项目而不是函数库项目。此时，项目的文件结构如下：

```
$ tree os
os
├── Cargo.toml
└── src
    └── main.rs

1 directory, 2 files
```

其中 `Cargo.toml` 中保存着项目的配置，包括作者的信息、联系方式以及库依赖等等。显而易见源代码保存在 `src` 目录下，目前为止只有 `main.rs` 一个文件，让我们看一下里面的内容：

列表 1: 最简单的 Rust 应用

```
1 fn main() {
2     println!("Hello, world!");
3 }
```

进入 `os` 项目根目录下，利用 Cargo 工具即可一条命令实现构建并运行项目：

```
$ cargo run
Compiling os v0.1.0 (/home/shinbokuow/workspace/v3/rCore-Tutorial-v3/os)
  Finished dev [unoptimized + debuginfo] target(s) in 1.15s
    Running `target/debug/os`
Hello, world!
```

如我们预想的一样，我们在屏幕上看到了一行 `Hello, world!`。但是，需要注意到我们所享受到的编程和执行程序的方便性并不是理所当然的，背后有着从硬件到软件多种机制的支持。特别是对于应用程序的运行，需要有一个强大的执行环境来帮助。接下来，我们就要看看有操作系统加持的强大的执行环境。

2.2.3 应用程序执行环境

如下图所示，现在通用操作系统（如 Linux 等）上的应用程序运行需要下面多层次的执行环境栈的支持，图中的白色块自上而下（越往下则越靠近底层，下层作为上层的执行环境支持上层代码的运行）表示各级执行环境，黑色块则表示相邻两层执行环境之间的接口。

我们的应用位于最上层，它可以通过调用编程语言提供的标准库或者其他三方库对外提供的函数接口，使得仅需少量的源代码就能完成复杂的功能。但是这些库的功能不仅限于此，事实上它们属于应用程序 **执行环境** (Execution Environment) 的一部分。在我们通常不会注意到的地方，这些软件库还会在执行应用之前完成一些初始化工作，并在应用程序执行的时候对它进行监控。我们在打印 `Hello, world!` 时使用的 `println!` 宏正是由 Rust 标准库 `std` 提供的。

从操作系统内核的角度看来，它上面的一切都属于用户态软件，而它自身属于内核态软件。无论用户态应用如何编写，是手写汇编代码，还是基于某种高级编程语言调用其标准库或三方库，某些功能总要直接或间接的通过操作系统内核提供的 **系统调用** (System Call) 来实现。因此系统调用充当了用户和内核之间的边界。内核作为用户态软件的执行环境，它不仅要提供系统调用接口，还需要对用户态软件的执行进行监控和管理。

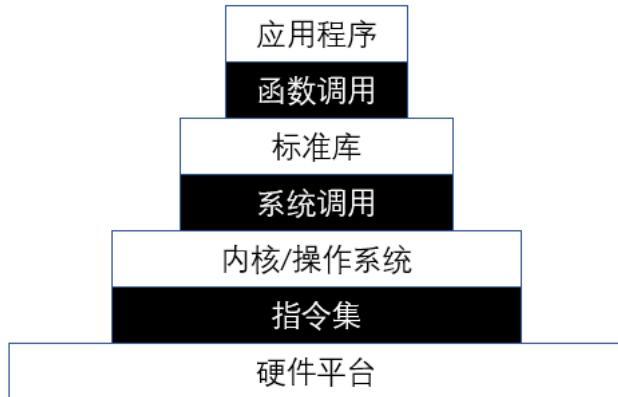


图 1: 应用程序执行环境栈

注解: Hello, world! 用到了哪些系统调用 ?

从之前的 `cargo run` 的输出可以看出之前构建的可执行文件是在 `target/debug` 目录下的 `os`。在 Ubuntu 系统上，可以通过 `strace` 工具来运行一个程序并输出程序运行过程当中向内核请求的所有的系统调用及其返回值。我们只需输入 `strace target/debug/os` 即可看到一长串的各种系统调用。

其中，容易看出与 `Hello, world!` 应用实际执行相关的只有两个系统调用：

```
# 输出字符串
write(1, "Hello, world!\n", 14)          = 14
# 程序退出执行
exit_group(0)
```

其参数的具体含义我们暂且不在这里进行解释。

其余的系统调用基本上分别用于函数库和内核两层执行环境的初始化工作和对上层应用的运行进行监控和管理。之后，随着应用场景的复杂化，操作系统需要提供更强的硬件抽象能力和资源管理能力，并实现相关的一些系统调用。

从硬件的角度来看，它上面的一切都属于软件。硬件可以分为三种：处理器 (Processor，也称 CPU)，内存 (Memory) 还有 I/O 设备。其中处理器无疑是其中最复杂，同时也最关键的一个。它与软件约定一套 **指令集体系结构** (ISA, Instruction Set Architecture)，使得软件可以通过 ISA 中提供的机器指令来访问各种硬件资源。软件当然也需要知道处理器会如何执行这些指令，以及指令执行后的结果。当然，实际的情况远比这个要复杂得多，为了适应现代应用程序的场景，处理器还需要提供很多额外的机制（如特权级、页表、TLB、异常/中断响应等）来管理应用程序的执行过程，而不仅仅是让数据在 CPU 寄存器、内存和 I/O 设备三者之间流动。

注解: 计算机科学中遇到的所有问题都可通过增加一层抽象来解决。

All problems in computer science can be solved by another level of indirection.

—计算机科学家 David Wheeler

注解: 多层执行环境都是必需的吗 ?

除了最上层的应用程序和最下层的硬件平台必须存在之外，作为中间层的函数库和操作系统内核并不是必须存在的：它们都是对下层资源进行了 **抽象** (Abstraction/Indirection)，并为上层提供了一个执行环境（也可理解为一些服务功能）。抽象的优点在于它让上层以较小的代价获得所需的功能，并同时可以提供一些保护。但

抽象同时也是一种限制，会丧失一些应有的灵活性。比如，当你在考虑在项目中应该使用哪个函数库的时候，就常常需要这方面的权衡：过多的抽象和过少的抽象自然都是不合适的。理解应用的需求也很重要。一个能合理满足应用需求的操作系统设计是操作系统设计者需要深入考虑的问题。这也是一种权衡，过多的服务功能和过少的服务功能自然都是不合适的。

实际上，我们通过应用程序的特征和需求来判断操作系统需要什么程度的抽象和功能。

- 如果函数库和操作系统内核都不存在，那么我们就需要手写汇编代码来控制硬件，这种方式具有最高的灵活性，抽象能力则最低，基本等同于编写汇编代码来直接控制硬件。我们通常用这种方式来实现一些架构相关且仅通过高级编程语言无法描述的小模块或者代码片段。
- 如果仅存在函数库而不存在操作系统内核，意味着我们不需要操作系统内核提供过于通用的抽象。在那些功能单一的嵌入式场景就常常会出现这种情况。嵌入式设备虽然也包含处理器、内存和 I/O 设备，但是它上面通常只会同时运行一个或几个功能非常简单的小应用程序，比如定时显示、实时采集数据、人脸识别打卡系统等。常见的解决方案是仅使用函数库构建单独的应用程序或是用专为应用场景特别裁减过的轻量级函数库管理少数应用程序。这就只需要一层函数库形态的执行环境。
- 如果存在函数库和操作系统内核，这意味着应用需求比较多样，会需要并发执行。常见的通用操作系统如 Windows/Linux/macOS 等都支持并发运行多个不同的应用程序。为此需要更加强大的操作系统抽象和功能，也就会需要多层执行环境。

注解：“用力过猛”的现代操作系统

对于如下更简单的小应用程序，我们可以看到“用力过猛”的现代操作系统提供的执行环境支持：

```

1 //ch1/donothing.rs
2 fn main() {
3     //do nothing
4 }
```

对于这个程序，在它的执行过程中，几乎感知不到它存在感。在编译后运行，可以看到的情况是：

```

$ rustc donothing.rs
$ ./donothing
$ (无输出)
```

与我们预计一样，程序执行后，看不到有何明显的输出信息，程序非常快地就结束了。但如果通过监视程序系统调用请求的工具 *strace* 来重新执行一下这个程序，就可以看到程序执行过程中的大量动态信息：

```

$ strace ./donothing
(多达 93 行的输出，表明 donothing 向 Linux
操作系統内核发出了93次各种各样的系统调用)
execve("./donothing", [".donothing"], 0x7ffe02c9ca10 /* 67 vars */) = 0
brk(NULL) = 0x563ba0532000
arch_prctl(0x3001 /* ARCH_??? */, 0x7fff2da54360) = -1 EINVAL (无效的参数)
.....
```

这说明了现在的操作系统，如 *Linux*，为了通用性，而实现了大量的功能。但对于非常简单的程序而言，有很多的功能是多余的。

2.2.4 目标平台与目标三元组

注解: 现代编译器工具集（以 C 或 Rust 编译器为例）的主要工作流程如下：

1. 源代码 (source code) -> 预处理器 (preprocessor) -> 宏展开的源代码
2. 宏展开的源代码 -> 编译器 (compiler) -> 汇编程序
3. 汇编程序 -> 汇编器 (assembler) -> 目标代码 (object code)
4. 目标代码 -> 链接器 (linker) -> 可执行文件 (executables)

对于一份用某种编程语言实现的应用程序源代码而言，编译器在将其通过编译、链接得到可执行文件的时候需要知道程序要在哪个 **平台** (Platform) 上运行。这里平台主要是指 CPU 类型、操作系统类型和标准运行时库的组合。从上面给出的[应用程序执行环境栈](#)可以看出：

- 如果用户态基于的内核不同，会导致系统调用接口不同或者语义不一致；
- 如果底层硬件不同，对于硬件资源的访问方式会有差异。特别是如果 ISA 不同，则向软件提供的指令集和寄存器都不同。

它们都会导致最终生成的可执行文件有很大不同。需要指出的是，某些编译器支持同一份源代码无需修改就可编译到多个不同的目标平台并在上面运行。这种情况下，源代码是**跨平台的**。而另一些编译器则已经预设好了一个固定的目标平台。

Rust 编译器通过**目标三元组** (Target Triplet) 来描述一个软件运行的目标平台。它一般包括 CPU、操作系统和运行时库等信息，从而控制 Rust 编译器可执行代码生成。比如，我们可以尝试看一下之前的 `Hello, world!` 的目标平台是什么。这可以通过打印编译器 `rustc` 的默认配置信息：

```
$ rustc --version --verbose
rustc 1.57.0-nightly (e1e9319d9 2021-10-14)
binary: rustc
commit-hash: e1e9319d93aea755c444c8f8ff863b0936d7a4b6
commit-date: 2021-10-14
host: x86_64-unknown-linux-gnu
release: 1.57.0-nightly
LLVM version: 13.0.0
```

从其中的 `host` 一项可以看出默认的目标平台是 `x86_64-unknown-linux-gnu`，其中 CPU 架构是 `x86_64`，CPU 厂商是 `unknown`，操作系统是 `linux`，运行时库是 `GNU libc`（封装了 Linux 系统调用，并提供 POSIX 接口为主的函数库）。这种无论编译器还是其生成的可执行文件都在我们当前所处的平台运行是一种最简单也最普遍的情况。但是很快我们就将遇到另外一种情况。

讲了这么多，终于该介绍我们的主线任务了。我们希望能够在另一个硬件平台上运行 `Hello, world!`，而与之前的默认平台不同的地方在于，我们将 CPU 架构从 `x86_64` 换成 `RISC-V`。

可以看一下目前 Rust 编译器支持哪些基于 `RISC-V` 的目标平台：

```
$ rustc --print target-list | grep riscv
riscv32gc-unknown-linux-gnu
riscv32i-unknown-none-elf
riscv32imac-unknown-none-elf
riscv32imc-unknown-none-elf
riscv64gc-unknown-linux-gnu
riscv64gc-unknown-none-elf
riscv64imac-unknown-none-elf
```

这里我们选择 `riscv64gc-unknown-none-elf` 目标平台。这其中的 CPU 架构是 `riscv64gc`，CPU 厂商是 `unknown`，操作系统是 `none`，`elf` 表示没有标准的运行时库（表明没有任何系统调用的封装支

持），但可以生成 ELF 格式的执行程序。这里我们之所以不选择有 linux-gnu 系统调用支持的目标平台 riscv64gc-unknown-linux-gnu，是因为我们只是想跑一个在裸机环境上运行的 Hello, world! 应用程序，没有必要使用 Linux 操作系统提供的那么高级的抽象和多余的操作系统服务。而且我们很清楚后续我们要开发的是一个操作系统内核，它必须直面底层物理硬件 (bare-metal) 来提供精简的操作系统服务功能，通用操作系统（如 Linux）提供的很多系统调用服务对这个内核而言是多余的。

注解: RISC-V 指令集拓展

由于基于 RISC-V 架构的处理器可能用于嵌入式场景或是通用计算场景，因此指令集规范将指令集划分为最基本的 RV32/64I 以及若干标准指令集拓展。每款处理器只需按照其实际应用场景按需实现指令集拓展即可。

- RV32/64I: 每款处理器都必须实现的基本整数指令集。在 RV32I 中，每个通用寄存器的位宽为 32 位；在 RV64I 中则为 64 位。它可以用来模拟绝大多数标准指令集拓展中的指令，除了比较特殊的 A 拓展，因为它需要特别的硬件支持。
- M 拓展: 提供整数乘除法相关指令。
- A 拓展: 提供原子指令和一些相关的内存同步机制，这个后面会展开。
- F/D 拓展: 提供单/双精度浮点数运算支持。
- C 拓展: 提供压缩指令拓展。

G 拓展是基本整数指令 I 再加上标准指令集拓展 MAFD 的总称，因此 riscv64gc 也就等同于 riscv64imafdc。我们剩下的内容都基于该处理器架构完成。除此之外 RISC-V 架构还有很多标准指令集拓展，有一些还在持续更新中尚未稳定，有兴趣的同学可以浏览最新版的 RISC-V 指令集规范。

2.2.5 Rust 标准库与核心库

我们尝试一下将当前的 Hello, world! 程序的目标平台换成 riscv64gc-unknown-none-elf 看看会发生什么事情：

```
$ cargo run --target riscv64gc-unknown-none-elf
Compiling os v0.1.0 (/home/shinbokuow/workspace/v3/rCore-Tutorial-v3/os)
error[E0463]: can't find crate for `std'
|
= note: the `riscv64gc-unknown-none-elf` target may not be installed
```

在之前的开发环境配置中，我们已经在 rustup 工具链中安装了这个目标平台支持，因此并不是该目标平台未安装的问题。这个问题只是单纯的表示在这个目标平台上找不到 Rust 标准库 std。我们之前曾经提到过，编程语言的标准库或三方库的某些功能会直接或间接的用到操作系统提供的系统调用。但目前我们所选的目标平台不存在任何操作系统支持，于是 Rust 并没有为这个目标平台支持完整的标准库 std。类似这样的平台通常被我们称为 裸机平台 (bare-metal)。这意味着在裸机平台上的软件没有传统操作系统支持。

注解: Rust Tips: Rust 语言标准库 std 和核心库 core

Rust 语言标准库-std 是让 Rust 语言开发的软件具备可移植性的基础，类似于 C 语言的 LibC 标准库。它是一组小巧的、经过实践检验的共享抽象，适用于更广泛的 Rust 生态系统开发。它提供了核心类型，如 Vec 和 Option、类库定义的语言原语操作、标准宏、I/O 和多线程等。默认情况下，我们可以使用 Rust 语言标准库来支持 Rust 应用程序的开发。但 Rust 语言标准库的一个限制是，它需要有操作系统的支持。所以，如果你要实现的软件是运行在裸机上的操作系统，就不能直接用 Rust 语言标准库了。

幸运的是，Rust 有一个对 Rust 语言标准库-std 裁剪过后的 Rust 语言核心库 core。core 库是不需要任何操作系统支持的，它的功能也比较受限，但是也包含了 Rust 语言相当一部分的核心机制，可以满足我们的大部分功能需求。Rust 语言是一种面向系统（包括操作系统）开发的语言，所以在 Rust 语言生态中，有很多三方库

也不依赖标准库 `std` 而仅仅依赖核心库 `core`。对它们的使用可以很大程度上减轻我们的编程负担。它们是我们能够在裸机平台挣扎求生的主要倚仗，也是大部分运行在没有操作系统支持的 Rust 嵌入式软件的必备。于是，我们知道在裸机平台上我们要将对于标准库 `std` 的引用换成核心库 `core`。但是实际做起来其实还要有一些琐碎的事情需要解决。

2.3 移除标准库依赖

2.3.1 本节导读

本章的目标是构建一个内核最小执行环境使得它能在 RV64GC（即实现了 IMAFDC 规范的 RISC-V 64 位 CPU）裸机上运行，在功能上它则像上一节最简单的 Rust 应用程序一样能够打印 `Hello, world!`，这将会为我们的后续章节提供很多调试上的方便，我们将其称为“三叶虫”操作系统。本节我们来进行第一个步骤：即对上一节最简单的 Rust 应用程序进行改造使得它能够被编译到 RV64GC 裸机平台上，为此我们需要移除它对于 Rust `std` 标准库的依赖，因为 Rust `std` 标准库自己就需要操作系统内核的支持。这样我们需要添加能够支持应用的裸机级别的库操作系统（LibOS）。

注解：库操作系统（Library OS, LibOS）

LibOS 以函数库的形式存在，为应用程序提供操作系统的基本功能。它最早来源于 MIT PDOS 研究小组在 1996 年左右的 Exokernel（外核）操作系统结构研究。Frans Kaashoek 教授的博士生 Dawson Engler 提出了一种与以往操作系统架构大相径庭的 Exokernel（外核）架构设计¹，即把传统的单体内核分为两部分，一部分以库操作系统的形式（即 LibOS）与应用程序紧耦合以实现传统的操作系统抽象，并进行面向应用的裁剪与优化；另外一部分（即外核）仅专注在最基本的安全复用物理硬件的机制上，来给 LibOS 提供基本的硬件访问服务。这样的设计思路可以针对应用程序的特征定制 LibOS，达到高性能的目标。

这种操作系统架构的设计思路比较超前，对原型系统的测试显示了很好的性能提升。但最终没有被工业界采用，其中最重要的原因是针对特定应用定制一个 LibOS 的工作量大，难以重复使用。人力成本因素导致了它不太被工业界认可。

2.3.2 移除 `println!` 宏

`println!` 宏所在的 Rust 标准库 `std` 需要通过系统调用获得操作系统的服务，而如果要构建运行在裸机上的操作系统，就不能再依赖标准库了。所以我们第一步要尝试移除 `println!` 宏及其所在的标准库。

由于后续实验需要 `rustc` 编译器缺省生成 RISC-V 64 的目标代码，所以我们首先要给 `rustc` 添加一个 `target: riscv64gc-unknown-none-elf`。这可通过如下命令来完成：

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

然后在 `os` 目录下新建 `.cargo` 目录，并在这个目录下创建 `config` 文件，并在里面输入如下内容：

```
# os/.cargo/config
[build]
target = "riscv64gc-unknown-none-elf"
```

¹

D. R. Engler, M. F. Kaashoek, and J. O' Toole. 1995. Exokernel: an operating system architecture for application-level resource management. In Proceedings of the fifteenth ACM symposium on Operating systems principles (SOSP '95). Association for Computing Machinery, New York, NY, USA, 251–266.

这会对于 Cargo 工具在 os 目录下的行为进行调整：现在默认会使用 riscv64gc 作为目标平台而不是原先的默认 x86_64-unknown-linux-gnu。事实上，这是一种编译器运行的开发平台 (x86_64) 与可执行文件运行的目标平台 (riscv-64) 不同的情况。我们把这种情况称为 **交叉编译** (Cross Compile)。

注解：本地编译与交叉编译

下面指的 **平台**主要由 CPU 硬件和操作系统这两个要素组成。

本地编译，即在当前开发平台下编译出来的程序，也只是放到这个平台下运行。如在 Linux x86-64 平台上编写代码并编译成可在 Linux x86-64 同样平台上执行的程序。

交叉编译，是一个与本地编译相对应的概念，即在一种平台上编译出在另一种平台上运行的程序。程序编译的环境与程序运行的环境不一样。如我们后续会讲到，在 Linux x86-64 开发平台上，编写代码并编译成可在 rCore Tutorial (这是我们要编写的操作系统内核) 和 riscv64gc (这是 CPU 硬件) 构成的目标平台上执行的程序。

当然，这只是使得我们之后在 cargo build 的时候不必再加上 --target 参数的一个小 trick。如果我们现在执行 cargo build，还是会和上一小节一样出现找不到标准库 std 的错误。于是我们需要在着手移除标准库的过程中一步一步地解决这些错误。

我们在 main.rs 的开头加上一行 #![no_std] 来告诉 Rust 编译器不使用 Rust 标准库 std 转而使用核心库 core (core 库不需要操作系统的支持)。编译器报出如下错误：

错误：

```
$ cargo build
Compiling os v0.1.0 (/home/shinbokuow/workspace/v3/rCore-Tutorial-v3/os)
error: cannot find macro `println` in this scope
--> src/main.rs:4:5
  |
4 |     println!("Hello, world!");
  |     ^^^^^^
```

我们之前提到过，println! 宏是由标准库 std 提供的，且会使用到一个名为 write 的系统调用。现在我们的代码功能还不足以自己实现一个 println! 宏。由于程序使用了系统调用，但不能在核心库 core 中找到它，所以我们目前先通过将 println! 宏注释掉的简单粗暴方式，来暂时绕过这个问题。

2.3.3 提供 panic_handler 功能应对致命错误

我们重新编译简单的 os 程序，之前的 println 宏缺失的错误消失了，但又出现了如下新的编译错误：

错误：

```
$ cargo build
Compiling os v0.1.0 (/home/shinbokuow/workspace/v3/rCore-Tutorial-v3/os)
error: `#[panic_handler]` function required, but not found
```

在使用 Rust 编写应用程序的时候，我们常常在遇到了一些无法恢复的致命错误 (panic)，导致程序无法继续向下运行。这时手动或自动调用 panic! 宏来打印出错的位置，让软件能够意识到它的存在，并进行一些后续处理。panic! 宏最典型的应用场景包括断言宏 assert! 失败或者对 Option::None/Result::Err 进行 unwrap 操作。所以 Rust 编译器在编译程序时，从安全性考虑，需要有 panic! 宏的具体实现。

在标准库 std 中提供了关于 panic! 宏的具体实现，其大致功能是打印出错位置和原因并杀死当前应用。但本章要实现的操作系统不能使用还需依赖操作系统的标准库 std，而更底层的核心库 core 中只有一个 panic! 宏的空壳，并没有提供 panic! 宏的精简实现。因此我们需要自己先实现一个简陋的 panic 处理函数，这样才能让“三叶虫”操作系统-LibOS 的编译通过。

注解: #[panic_handler]

#[panic_handler] 是一种编译指导属性，用于标记核心库 core 中的 panic! 宏要对接的函数（该函数实现对致命错误的具体处理）。该编译指导属性所标记的函数需要具有 fn(&PanicInfo) -> ! 函数签名，函数可通过 PanicInfo 数据结构获取致命错误的相关信息。这样 Rust 编译器就可以把核心库 core 中的 panic! 宏定义与 #[panic_handler] 指向的 panic 函数实现合并在一起，使得 no_std 程序具有类似 std 库的应对致命错误的功能。

我们创建一个新的子模块 lang_items.rs 实现 panic 函数，并通过 #[panic_handler] 属性通知编译器用 panic 函数来对接 panic! 宏。为了将该子模块添加到项目中，我们还需要在 main.rs 的 #![no_std] 的下方加上 mod lang_items;，相关知识可参考 [Rust 模块编程](#)：

```

1 // os/src/lang_items.rs
2 use core::panic::PanicInfo;
3
4 #[panic_handler]
5 fn panic(_info: &PanicInfo) -> ! {
6     loop {}
7 }
```

在把 panic_handler 配置在单独的文件 os/src/lang_items.rs 后，需要在 os/src/main.rs 文件中添加以下内容才能正常编译整个软件：

```

1 // os/src/main.rs
2 #![no_std]
3 mod lang_items;
4 // ... other code
```

注意，panic 处理函数的函数签名需要一个 PanicInfo 的不可变借用作为输入参数，它在核心库中得以保留，这也是我们第一次与核心库打交道。之后我们会从 PanicInfo 解析出错位置并打印出来，然后杀死应用程序。但目前我们什么都不做只是在原地 loop。

2.3.4 移除 main 函数

我们再次重新编译简单的 os 程序，之前的 #[panic_handler] 函数缺失的错误消失了，但又出现了如下新的编译错误：.. error:

```

.. code-block::

$ cargo build
    Compiling os v0.1.0 (/home/shinbokuow/workspace/v3/rCore-Tutorial-v3/os)
error: requires `start` lang_item
```

编译器提醒我们缺少一个名为 start 的语义项。我们回忆一下，之前提到语言标准库和三方库作为应用程序的执行环境，需要负责在执行应用程序之前进行一些初始化工作，然后才跳转到应用程序的入口点（也就是跳转到我们编写的 main 函数）开始执行。事实上 start 语义项代表了标准库 std 在执行应用程序之前需要进行的一些初始化工作。由于我们禁用了标准库，编译器也就找不到这项功能的实现了。

最简单的解决方案就是压根不让编译器使用这项功能。我们在 main.rs 的开头加入设置 #![no_main] 告诉编译器我们没有一般意义上的 main 函数，并将原来的 main 函数删除。在失去了 main 函数的情况下，

编译器也就不需要完成所谓的初始化工作了。

至此，我们成功移除了标准库的依赖，并完成了构建裸机平台上的“三叶虫”操作系统的第一步工作—通过编译器检查并生成执行码。

```
$ cargo build
Compiling os v0.1.0 (/home/shinbokuow/workspace/v3/rCore-Tutorial-v3/os)
  Finished dev [unoptimized + debuginfo] target(s) in 0.06s
```

目前的主要代码包括 `main.rs` 和 `lang_items.rs`，大致内容如下：

```
1 // os/src/main.rs
2 #[no_main]
3 #[no_std]
4 mod lang_items;
5 // ... other code
6
7
8 // os/src/lang_items.rs
9 use core::panic::PanicInfo;
10
11 #[panic_handler]
12 fn panic(_info: &PanicInfo) -> ! {
13     loop {}
14 }
```

本小节我们固然脱离了标准库，通过了编译器的检验，但也是伤筋动骨，将原有的很多功能弱化甚至直接删除，看起来距离在 RV64GC 平台上打印 `Hello world!` 相去甚远了（我们甚至连 `println!` 和 `main` 函数都删除了）。不要着急，接下来我们会以自己的方式来重塑这些基本功能，并最终完成我们的目标。

注解: Rust Tips: Rust 模块化编程

将一个软件工程项目划分为多个子模块分别进行实现是一种被广泛应用的编程技巧，它有助于促进复用代码，并显著提升代码的可读性和可维护性。因此，众多编程语言均对模块化编程提供了支持，Rust 语言也不例外。

每个通过 Cargo 工具创建的 Rust 项目均是一个模块，取决于 Rust 项目类型的不同，模块的根所在的位置也不同。当使用 `--bin` 创建一个可执行的 Rust 项目时，模块的根是 `src/main.rs` 文件；而当使用 `--lib` 创建一个 Rust 库项目时，模块的根是 `src/lib.rs` 文件。在模块的根文件中，我们需要声明所有可能会用到的子模块。如果不声明的话，即使子模块对应的文件存在，Rust 编译器也不会用到它们。如上面的代码片段中，我们就在根文件 `src/main.rs` 中通过 `mod lang_items;` 声明了子模块 `lang_items`，该子模块实现在文件 `src/lang_item.rs` 中，我们将项目中所有的语义项放在该模块中。

当一个子模块比较复杂的时候，它往往不会被放在一个独立的文件中，而是放在一个 `src` 目录下与子模块同名的子目录之下，在后面的章节中我们常会用到这种方法。例如第二章代码（参见代码仓库的 `ch2` 分支）中的 `syscall` 子模块就放在 `src/syscall` 目录下。对于这样的子模块，其所在目录下的 `mod.rs` 为该模块的根，其中可以进而声明它的子模块。同样，这些子模块既可以放在一个文件中，也可以放在一个目录下。

每个模块可能会对其他模块公开一些变量、类型或函数，而该模块的其他内容则是对其他模块不可见的，也即其他模块不允许引用或访问这些内容。在模块内，仅有被显式声明为 `pub` 的内容才会对其他模块公开。Rust 类内部声明的属性域和方法也可以对其他类公开或是不对其他类公开，这取决于它们是否被声明为 `pub`。我们在 C++/Java 语言中能够找到相同功能的关键字：即 `public/private`。提供上述可见性机制的原因在于让其他类/模块能够访问当前类/模块公开提供的内容而无需关心它们是如何实现的，它们实际上也无法看到这些具体实现，因为这些具体实现并未向它们公开。编译器会对可见性进行检查，例如，当一个类/模块试图访问其他类/模块未公开的方法时，将无法通过编译。

我们可以使用绝对路径或相对路径来引用其他模块或当前模块的内容。参考上面的 `use core::panic::PanicInfo;`，类似 C++，我们将模块的名字按照层级由浅到深排列，并在相邻层级

之间使用分隔符 `::` 进行分隔。路径的最后一级（如 `PanicInfo`）则表示我们具体要引用或访问的内容，可能是变量、类型或者方法名。当通过绝对路径进行引用时，路径最开头可能是项目依赖的一个外部库的名字，或者是 `crate` 表示项目自身的根模块。在后面的章节中，我们会多次用到它们。

2.3.5 分析被移除标准库的程序

对于上面这个被移除标准库的应用程序，通过了 Rust 编译器的检查和编译，形成了二进制代码。但这个二进制代码的内容是什么，它能否在 RISC-V 64 计算机上正常执行呢？为了分析这个二进制可执行程序，首先需要安装 `cargo-binutils` 工具集：

```
$ cargo install cargo-binutils
$ rustup component add llvm-tools-preview
```

这样我们可以通过各种工具来分析目前的程序：

```
# 文件格式
$ file target/riscv64gc-unknown-none-elf/debug/os
target/riscv64gc-unknown-none-elf/debug/os: ELF 64-bit LSB executable, UCB RISC-V, ...
←...

# 文件头信息
$ rust-readobj -h target/riscv64gc-unknown-none-elf/debug/os
File: target/riscv64gc-unknown-none-elf/debug/os
Format: elf64-littleriscv
Arch: riscv64
AddressSize: 64bit
.....
Type: Executable (0x2)
Machine: EM_RISCV (0xF3)
Version: 1
Entry: 0x0
.....
}

# 反汇编导出汇编程序
$ rust-objdump -S target/riscv64gc-unknown-none-elf/debug/os
target/riscv64gc-unknown-none-elf/debug/os:           file format elf64-littleriscv
```

通过 `file` 工具对二进制程序 `os` 的分析可以看到它好像是一个合法的 RISC-V 64 可执行程序，但通过 `rust-readobj` 工具进一步分析，发现它的入口地址 `Entry` 是 0，从 C/C++ 等语言中得来的经验告诉我们，0 一般表示 NULL 或空指针，因此等于 0 的入口地址看上去无法对应到任何指令。再通过 `rust-objdump` 工具把它反汇编，可以看到没有生成汇编代码。所以，我们可以断定，这个二进制程序虽然合法，但它是一个空程序。产生该现象的原因是：目前我们的程序（参考上面的源代码）没有进行任何有意义的工作，由于我们移除了 `main` 函数并将项目设置为 `#![no_main]`，它甚至没有一个传统意义上的入口点（即程序首条被执行的指令所在的位置），因此 Rust 编译器会生成一个空程序。

在下面几节，我们将建立有支持显示字符串的最小执行环境。

注解：在 x86_64 平台上移除标准库依赖

有兴趣的同学可以将目标平台换回之前默认的 `x86_64-unknown-linux-gnu` 并重复本小节所做的事情，比较两个平台从 ISA 到操作系统的差异。可以参考 `BlogOS` 的相关内容²。

² <https://os.phil-opp.com/freestanding-rust-binary/>

注解：本节内容部分参考自 [BlogOS](#) 的相关章节。

2.4 内核第一条指令（基础篇）

2.4.1 本节导读

接下来两节我们将进行构建“三叶虫”操作系统的第二步，即将我们的内核正确加载到 Qemu 模拟器上，使得 Qemu 模拟器可以成功执行内核的第一条指令。本节我们将介绍一些相关基础知识，首先介绍计算机的各个硬件组成部分，特别是 CPU 和物理内存。其次介绍 Qemu 模拟器的抽象模型、使用方法以及启动流程。作为知识补充，我们还介绍了一般计算机的启动流程。最后我们介绍了程序内存布局和编译流程，特别是链接的相关知识。这些基本知识可以帮助我们更好的理解下一节的实践操作背后的动机和目的。

2.4.2 计算机组装基础

当编写应用程序的时候，大多数情况下我们只需调用库函数即可在操作系统的支持下实现各项功能，而无需关心操作系统如何调度管理各类软硬件资源。操作系统提供了一些监控工具（如 Windows 上的任务管理器或 Linux 上的 `ps` 工具），这些工具可以帮助我们统计 CPU、内存、硬盘、网络等资源的占用情况，从而让我们大致上了解这些资源的使用情况，并帮助我们更好地开发或部署应用程序。然而，在实际编写操作系统的时候，我们就必须直面这些硬件资源，将它们管理起来并为应用程序提供高效易用的抽象。为此，我们必须增进对于这些硬件的了解。

计算机主要由处理器（Processor，也即中央处理器，CPU，Central Processing Unit），物理内存和 I/O 外设三部分组成。在前八章我们主要用到 CPU 和物理内存，第九章则开始与丰富多彩的外设打交道。处理器的主要功能是从物理内存中读取指令、译码并执行，在此过程中还要与物理内存和 I/O 外设打交道。物理内存则是计算机体系结构中一个重要的组成部分。在存储方面，CPU 唯一能够直接访问的只有物理内存中的数据，它可以通过访存指令来达到这一目的。从 CPU 的视角看来，可以将物理内存看成一个大字节数组，而物理地址则对应于一个能够用来访问数组中某个元素的下标。与我们日常编程习惯不同的是，该下标通常不以 0 开头，而通常以一个常数，如 `0x80000000` 开头。简言之，CPU 可以通过物理地址来寻址，并逐字节地访问物理内存中保存的数据。

值得一提的是，当 CPU 以多个字节（比如 2/4/8 或更多）为单位访问物理内存（事实上并不局限于物理内存，也包括 I/O 外设的数据空间）中的数据时，就有可能会引入端序（也称字节顺序）和内存地址对齐的问题。由于这并不是重点，我们在这里不展开说明，如读者有兴趣可以参考下面的补充说明。

注解：端序或尾序

端序或尾序（Endianness），又称字节顺序。在计算机科学领域中，指电脑内存中或在数字通信链路中，多字节组成的字（Word）的字节（Byte）的排列顺序。字节的排列方式有两个通用规则。例如，将一个多位数的低位放在较小的地址处，高位放在较大的地址处，则称小端序（little-endian）；反之则称大端序（big-endian）。常见的 x86、RISC-V 等架构采用的是小端序。

注解：内存地址对齐

内存地址对齐是内存中的数据排列，以及 CPU 访问内存数据的方式，包含了基本数据对齐和结构体数据对齐的两部分。CPU 在内存中读写数据是按字节块进行操作，理论上任意类型的变量访问可以从内存的任何地址开始，但在计算机系统中，CPU 访问内存是通过数据总线（决定了每次读取的数据位数）和地址总线（决定了寻址范围）来进行的，基于计算机的物理组成和性能需求，CPU 一般会要求访问内存数据的首地址的值为 4 或 8 的整数倍。

基本类型数据对齐是指数据在内存中的偏移地址必须为一个字的整数倍，这种存储数据的方式，可以提升系统在读取数据时的性能。结构体数据对齐，是指在结构体中的上一个数据域结束和下一个数据域开始的地方填充一些无用的字节，以保证每个数据域（假定是基本类型数据）都能够对齐（即按基本类型数据对齐）。

对于 RISC-V 处理器而言，load/store 指令进行数据访存时，数据在内存中的地址应该对齐。如果访存 32 位数据，内存地址应当按 32 位（4 字节）对齐。如果数据的地址没有对齐，执行访存操作将产生异常。这也是在学习内核编程中经常碰到的一种 bug。

2.4.3 了解 Qemu 模拟器

我们编写的内核将主要在 Qemu 模拟器上运行来检验其正确性。这样做主要是为了方便快捷，只需在命令行输入一行命令即可让内核跑起来。为了让我们的内核能够正确对接到 Qemu 模拟器上，我们首先要对 Qemu 模拟器有一定的了解。在本书中，我们使用软件 `qemu-system-riscv64` 来模拟一台 64 位 RISC-V 架构的计算机，它包含 CPU、物理内存以及若干 I/O 外设。它的具体配置（比如 CPU 的核数或是物理内存的大小）均可由用户通过 Qemu 的执行参数选项来调整。作为模拟器，在宿主机看来它只是一个用户程序，因此上面提到的资源都是它利用宿主机（即 Qemu 运行所在的平台，如 Linux/Windows/macOS）提供给它的资源模拟出来的。在 Qemu 上模拟出来的某些硬件资源性能很高，甚至接近宿主机上原生资源的性能；而另一些硬件资源的模拟开销较大，从而导致 Qemu 模拟的整体硬件性能相对较慢。但对于本书所实践的各种操作系统而言，在当前 x86-64 处理器上的 Qemu 所模拟的硬件性能已经足够快了。

接下来我们来看如何启动 Qemu。从各章节代码中的 `os/Makefile` 可以看到，我们使用如下命令来启动 Qemu 并运行我们的内核：

```

1 $ qemu-system-riscv64 \
2   -machine virt \
3   -nographic \
4   -bios ..//bootloader/rustsbi-qemu.bin \
5   -device loader,file=target/riscv64gc-unknown-none-elf/release/os.bin,
  ↳addr=0x80200000

```

其中各个执行参数选项的含义如下：

- `-machine virt` 表示将模拟的 64 位 RISC-V 计算机设置为名为 `virt` 的虚拟计算机。我们知道，即使同属同一种指令集架构，也会有很多种不同的计算机配置，比如 CPU 的生产厂商和型号不同，支持的 I/O 外设种类也不同。关于 `virt` 平台的更多信息可以参考¹。Qemu 还支持模拟其他 RISC-V 计算机，其中包括由 SiFive 公司生产的著名的 HiFive Unleashed 开发板。
- `-nographic` 表示模拟器不需要提供图形界面，而只需要对外输出字符流。
- 通过 `-bios` 可以设置 Qemu 模拟器开机时用来初始化的引导加载程序（bootloader），这里我们使用预编译好的 `rustsbi-qemu.bin`，它需要被放在与 `os` 同级的 `bootloader` 目录下，该目录可以从每一章的代码分支中获得。
- 通过虚拟设备 `-device` 中的 `loader` 属性可以在 Qemu 模拟器开机之前将一个宿主机上的文件载入到 Qemu 的物理内存的指定位置中，`file` 和 `addr` 属性分别可以设置待载入文件的路径以及将文件载入到的 Qemu 物理内存上的物理地址。这里我们载入的 `os.bin` 被称为 **内核镜像**，它会被载入到 Qemu 模拟器内存的 `0x80200000` 地址处。那么内核镜像 `os.bin` 是怎么来的呢？上一节中我们移除标准库依赖后会得到一个内核可执行文件 `os`，将其进一步处理就能得到 `os.bin`，具体处理流程我们会在后面深入讨论。

¹ <https://www.qemu.org/docs/master/system/riscv/virt.html>

Qemu 启动流程

在 Qemu 模拟的 virt 硬件平台上，物理内存的起始物理地址为 0x80000000，物理内存的默认大小为 128MiB，它可以通过 -m 选项进行配置。如果使用默认配置的 128MiB 物理内存则对应的物理地址区间为 [0x80000000, 0x88000000)。如果使用上面给出的命令启动 Qemu，那么在 Qemu 开始执行任何指令之前，首先把两个文件加载到 Qemu 的物理内存中：即把作为 bootloader 的 rustsbi-qemu.bin 加载到物理内存以物理地址 0x80000000 开头的区域上，同时把内核镜像 os.bin 加载到以物理地址 0x80200000 开头的区域上。

为什么加载到这两个位置呢？这与 Qemu 模拟计算机加电启动后的运行流程有关。一般来说，计算机加电之后的启动流程可以分成若干个阶段，每个阶段均由一层软件或固件负责，每一层软件或固件的功能是进行它应当承担的初始化工作，并在此之后跳转到下一层软件或固件的入口地址，也就是将计算机的控制权移交给了下一层软件或固件。Qemu 模拟的启动流程则可以分为三个阶段：第一个阶段由固化在 Qemu 内的一小段汇编程序负责；第二个阶段由 bootloader 负责；第三个阶段则由内核镜像负责。

- 第一阶段：将必要的文件载入到 Qemu 物理内存之后，Qemu CPU 的程序计数器 (PC, Program Counter) 会被初始化为 0x1000，因此 Qemu 实际执行的第一条指令位于物理地址 0x1000，接下来它将执行寥寥数条指令并跳转到物理地址 0x80000000 对应的指令处并进入第二阶段。从后面的调试过程可以看出，该地址 0x80000000 被固化在 Qemu 中，作为 Qemu 的使用者，我们在不触及 Qemu 源代码的情况下无法进行更改。
- 第二阶段：由于 Qemu 的第一阶段固定跳转到 0x80000000，我们需要将负责第二阶段的 bootloader rustsbi-qemu.bin 放在以物理地址 0x80000000 开头的物理内存中，这样就能保证 0x80000000 处正好保存 bootloader 的第一条指令。在这一阶段，bootloader 负责对计算机进行一些初始化工作，并跳转到下一阶段软件的入口，在 Qemu 上即可实现将计算机控制权移交给我们的内核镜像 os.bin。这里需要注意的是，对于不同的 bootloader 而言，下一阶段软件的入口不一定相同，而且获取这一信息的方式和时间点也不同：入口地址可能是一个预先约定好的固定的值，也有可能是在 bootloader 运行期间才动态获取到的值。我们选用的 RustSBI 则是将下一阶段的入口地址预先约定为固定的 0x80200000，在 RustSBI 的初始化工作完成之后，它会跳转到该地址并将计算机控制权移交给下一阶段的软件——也即我们的内核镜像。
- 第三阶段：为了正确地和上一阶段的 RustSBI 对接，我们需要保证内核的第一条指令位于物理地址 0x80200000 处。为此，我们需要将内核镜像预先加载到 Qemu 物理内存以地址 0x80200000 开头的区域上。一旦 CPU 开始执行内核的第一条指令，证明计算机的控制权已经被移交给我们的内核，也就达到了本节的目标。

注解：真实计算机的加电启动流程

真实计算机的启动流程大致上也可以分为三个阶段：

- 第一阶段：加电后 CPU 的 PC 寄存器被设置为计算机内部只读存储器 (ROM, Read-only Memory) 的物理地址，随后 CPU 开始运行 ROM 内的软件。我们一般将该软件称为固件 (Firmware)，它的功能是对 CPU 进行一些初始化操作，将后续阶段的 bootloader 的代码、数据从硬盘载入到物理内存，最后跳转到适当的地址将计算机控制权转移给 bootloader。它大致对应于 Qemu 启动的第一阶段，即在物理地址 0x1000 处放置的若干条指令。可以看到 Qemu 上的固件非常简单，因为它并不需要负责将 bootloader 从硬盘加载到物理内存中，这个任务此前已经由 Qemu 自身完成了。
- 第二阶段：bootloader 同样完成一些 CPU 的初始化工作，将操作系统镜像从硬盘加载到物理内存中，最后跳转到适当地址将控制权转移给操作系统。可以看到一般情况下 bootloader 需要完成一些数据加载工作，这也就是它名字中 loader 的来源。它对应于 Qemu 启动的第二阶段。在 Qemu 中，我们使用的 RustSBI 功能较弱，它并没有能力完成加载的工作，内核镜像实际上是由 bootloader 一起在 Qemu 启动之前加载到物理内存中的。
- 第三阶段：控制权被转移给操作系统。由于篇幅所限后面我们就不再赘述了。

值得一提的是，为了让计算机的启动更加灵活，bootloader 目前可能非常复杂：它可能也分为多个阶段，并且能管理一些硬件资源，从复杂性上它已接近一个传统意义上的操作系统。

基于上面对 Qemu 启动流程的介绍，我们可以知道为了让我们的内核镜像能够正确对接到 Qemu 和 RustSBI 上，我们提交给 Qemu 的内核镜像文件必须满足：该文件的开头即为内核待执行的第一条指令。但后面会讲到，在上一节中我们通过移除标准库依赖得到的可执行文件实际上并不满足该条件。因此，我们还需要对可执行文件进行一些操作才能得到可提交给 Qemu 的内核镜像。为了说明这些条件，首先我们需要了解一些关于程序内存布局和编译流程的知识。

2.4.4 程序内存布局与编译流程

程序内存布局

在我们将源代码编译为可执行文件之后，它就会变成一个看似充满了杂乱无章的字节的一个文件。但我们知道这些字节至少可以分成代码和数据两部分，在程序运行起来的时候它们的功能并不相同：代码部分由一条条可以被 CPU 解码并执行的指令组成，而数据部分只是被 CPU 视作可读写的内存空间。事实上我们还可以根据其功能进一步把两个部分划分为更小的单位：段 (Section)。不同的段会被编译器放置在内存不同的位置上，这构成了程序的 内存布局 (Memory Layout)。一种典型的程序相对内存布局如下所示：

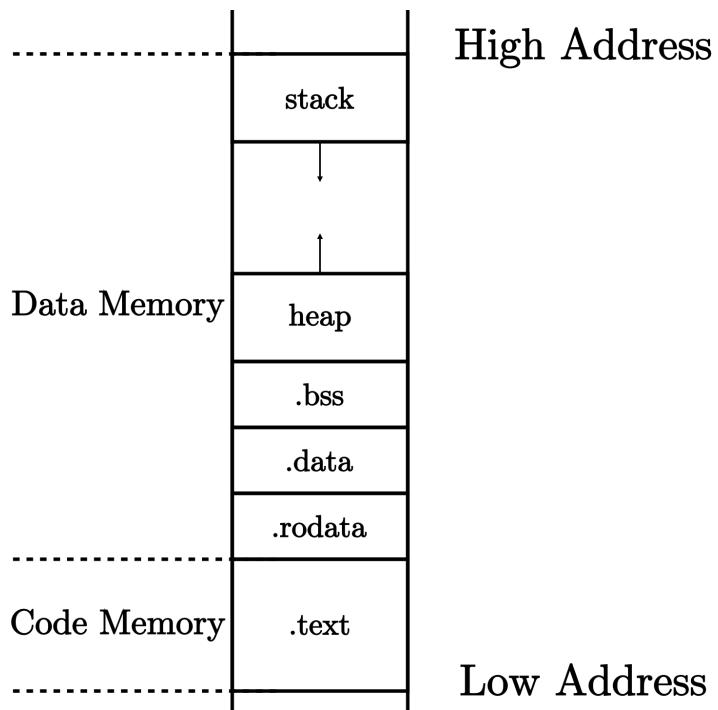


图 2: 一种典型的程序相对内存布局

在上图中可以看到，代码部分只有代码段 `.text` 一个段，存放程序的所有汇编代码。而数据部分则还可以继续细化：

- 已初始化数据段保存程序中那些已初始化的全局数据，分为 `.rodata` 和 `.data` 两部分。前者存放只读的全局数据，通常是一些常数或者是常量字符串等；而后者存放可修改的全局数据。
- 未初始化数据段 `.bss` 保存程序中那些未初始化的全局数据，通常由程序的加载者代为进行零初始化，即将这块区域逐字节清零；
- 堆 (heap) 区域用来存放程序运行时动态分配的数据，如 C/C++ 中的 `malloc/new` 分配到的数据本体就放在堆区域，它向高地址增长；

- **栈 (stack)** 区域不仅用作函数调用上下文的保存与恢复，每个函数作用域内的局部变量也被编译器放在它的栈帧内，它向低地址增长。

注解：局部变量与全局变量

在一个函数的视角中，它能够访问的变量包括以下几种：

- 函数的输入参数和局部变量：保存在一些寄存器或是该函数的栈帧里面，如果是在栈帧里面的话是基于当前栈指针加上一个偏移量来访问的；
- 全局变量：保存在数据段 `.data` 和 `.bss` 中，某些情况下 `gp(x3)` 寄存器保存两个数据段中间的一个位置，于是全局变量是基于 `gp` 加上一个偏移量来访问的。
- 堆上的动态变量：本体被保存在堆上，大小在运行时才能确定。而我们只能直接访问栈上或者全局数据段中的 **编译期确定大小** 的变量。因此我们需要通过一个运行时分配内存得到的一个指向堆上数据的指针来访问它，指针的位宽确实在编译期就能够确定。该指针即可以作为局部变量放在栈帧里面，也可以作为全局变量放在全局数据段中。

编译流程

从源代码得到可执行文件的编译流程可被细化为多个阶段（虽然输入一条命令便可将它们全部完成）：

1. **编译器 (Compiler)** 将每个源文件从某门高级编程语言转化为汇编语言，注意此时源文件仍然是一个 ASCII 或其他编码的文本文件；
2. **汇编器 (Assembler)** 将上一步的每个源文件中的文本格式的指令转化为机器码，得到一个二进制的 **目标文件 (Object File)**；
3. **链接器 (Linker)** 将上一步得到的所有目标文件以及一些可能的外部目标文件链接在一起形成一个完整的可执行文件。

汇编器输出的每个目标文件都有一个独立的程序内存布局，它描述了目标文件内各段所在的位置。而链接器所做的事情是将所有输入的目标文件整合成一个整体的内存布局。在此期间链接器主要完成两件事情：

- 第一件事情是将来自不同目标文件的段在目标内存布局中重新排布。如下图所示，在链接过程中，分别来自于目标文件 `1.o` 和 `2.o` 段被按照段的功能进行分类，相同功能的段被排在一起放在拼装后的目标文件 `output.o` 中。注意到，目标文件 `1.o` 和 `2.o` 的内存布局是存在冲突的，同一个地址在不同的内存布局中存放不同的内容。而在合并后的内存布局中，这些冲突被消除。
- 第二件事情是将符号替换为具体地址。这里的符号指什么呢？我们知道，在我们进行模块化编程的时候，每个模块都会提供一些向其他模块公开的全局变量、函数等供其他模块访问，也会访问其他模块向它公开的内容。要访问一个变量或者调用一个函数，在源代码级别我们只需知道它们的名字即可，这些名字被我们称为符号。取决于符号来自于模块内部还是其他模块，我们还可以进一步将符号分成内部符号和外部符号。然而，在机器码级别（也即在目标文件或可执行文件中）我们并不是通过符号来找到索引我们想要访问的变量或函数，而是直接通过变量或函数的地址。例如，如果想调用一个函数，那么在指令的机器码中我们可以找到函数入口的绝对地址或者相对于当前 PC 的相对地址。

那么，符号何时被替换为具体地址呢？因为符号对应的变量或函数都是放在某个段里面的固定位置（如全局变量往往放在 `.bss` 或者 `.data` 段中，而函数则放在 `.text` 段中），所以我们需要等待符号所在的段确定了它们在内存布局中的位置之后才能知道它们确切的地址。当一个模块被转化为目标文件之后，它的内部符号就已经在目标文件中被转化为具体的地址了，因为目标文件给出了模块的内存布局，也就意味着模块内的各个段的位置已经被确定了。然而，此时模块所用到的外部符号的地址无法确定。我们需要将这些外部符号记录下来，放在目标文件一个名为符号表（Symbol table）的区域内。由于后续可能还需要重定位，内部符号也同样需要被记录在符号表中。

外部符号需要等到链接的时候才能被转化为具体地址。假设模块 1 用到了模块 2 提供的内容，当两个模块的目标文件链接到一起的时候，它们的内存布局会被合并，也就意味着两个模块的各个段的位置均被确定下来。此时，模块 1 用到的来自模块 2 的外部符号可以被转化为具体地址。同时我们还需要注

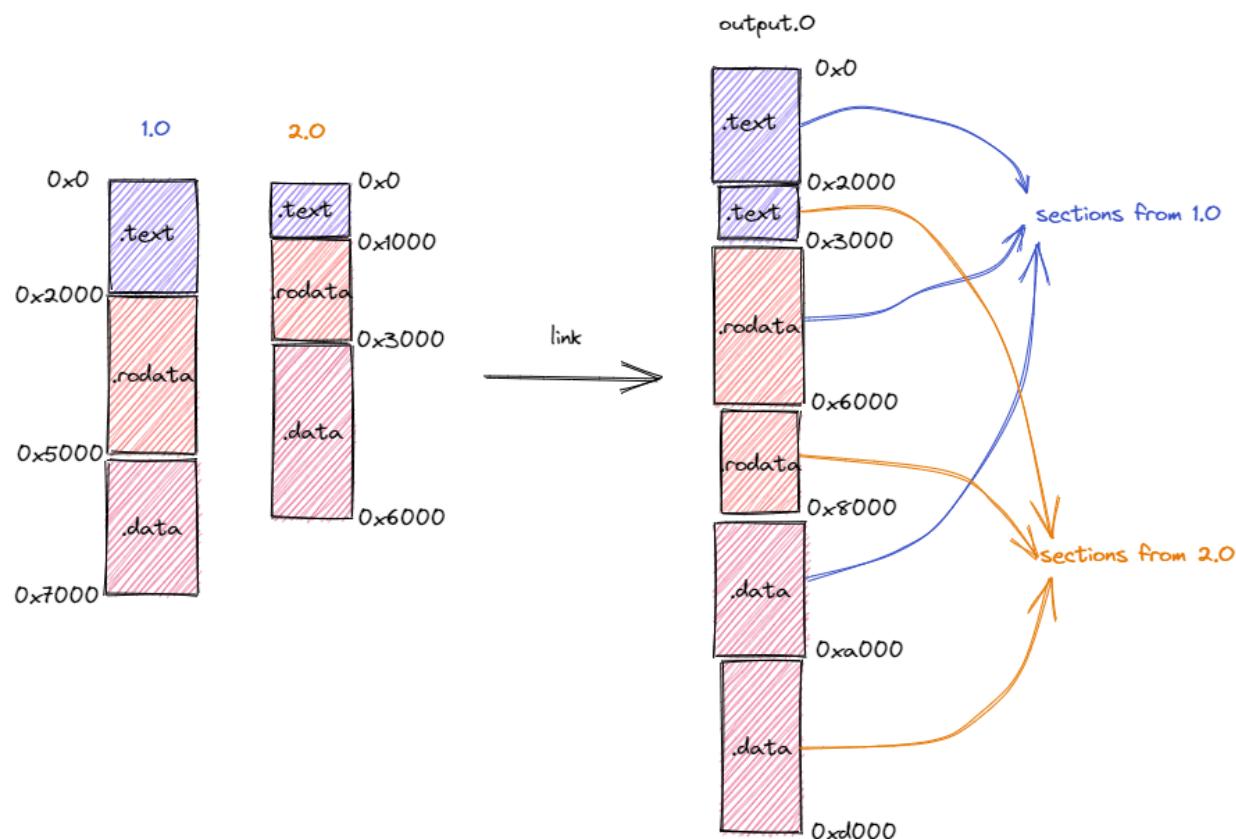


图 3: 来自不同目标文件的段的重新排布

意：两个模块的段在合并后的内存布局中被重新排布，其最终的位置有可能和它们在模块自身的局部内存布局中的位置相比已经发生了变化。因此，每个模块的内部符号的地址也有可能会发生变化，我们还需要进行修正。上面的过程被称为重定位（Relocation），这个过程形象一些来说很像拼图：由于模块 1 用到了模块 2 的内容，因此二者分别相当于一块凹进和凸出一部分的拼图，正因如此我们可以将它们无缝地拼接到一起。

上面我们简单介绍了程序内存布局和编译流程特别是链接过程的相关知识。那么如何得到一个能够在 Qemu 上成功运行的内核镜像呢？首先我们需要通过链接脚本调整内核可执行文件的内存布局，使得内核被执行的第一条指令位于地址 0x80200000 处，同时代码段所在的地址应低于其他段。这是因为 Qemu 物理内存中低于 0x80200000 的区域并未分配给内核，而是主要由 RustSBI 使用。其次，我们需要将内核可执行文件中的元数据丢掉得到内核镜像，此内核镜像仅包含实际会用到的代码和数据。这则是因为 Qemu 的加载功能过于简单直接，它直接将输入的文件逐字节拷贝到物理内存中，因此也可以说这一步是我们在帮助 Qemu 手动将可执行文件加载到物理内存中。下一节我们将成功生成内核镜像并在 Qemu 上验证控制权被转移到内核。

2.5 内核第一条指令（实践篇）

2.5.1 本节导读

承接上一节，本节我们将实践在 Qemu 上执行内核的第一条指令。首先我们编写内核第一条指令并嵌入到我们的内核项目中，接着指定内核的内存布局使得我们的内核可以正确对接到 Qemu 中。由于 Qemu 的文件加载功能过于简单，它不支持完整的可执行文件，因此我们从内核可执行文件中剥离多余的元数据得到内核镜像并提供给 Qemu。最后，我们使用 GDB 来跟踪 Qemu 的整个启动流程并验证内核的第一条指令被正确执行。

提示：在进入本节之前请参考 [实验环境配置](#) 安装配置 Rust 相关软件包、Qemu 软件和 GDB 调试工具等。

2.5.2 编写内核第一条指令

首先，我们需要编写进入内核后的第一条指令，这样更方便我们验证我们的内核镜像是否正确对接到 Qemu 上。

```

1 # os/src/entry.asm
2     .section .text.entry
3     .globl _start
4     _start:
5     li x1, 100

```

实际的指令位于第 5 行，也即 `li x1, 100`。`li` 是 Load Immediate 的缩写，也即将一个立即数加载到某个寄存器，因此这条指令可以看做将寄存器 `x1` 赋值为 100。第 4 行我们声明了一个符号 `_start`，该符号指向紧跟在符号后面的内容——也就是位于第 5 行的指令，因此符号 `_start` 的地址即为第 5 行的指令所在的地址。第 3 行我们告知编译器 `_start` 是一个全局符号，因此可以被其他目标文件使用。第 2 行表明我们希望将第 2 行后面的内容全部放到一个名为 `.text.entry` 的段中。一般情况下，所有的代码都被放到一个名为 `.text` 的代码段中，这里我们将其命名为 `.text.entry` 从而区别于其他 `.text` 的目的在于我们想要确保该段被放置在相比任何其他代码段更低的地址上。这样，作为内核的入口点，这段指令才能被最先执行。

接着，我们在 `main.rs` 中嵌入这段汇编代码，这样 Rust 编译器才能够注意到它，不然编译器会认为它是一个与项目无关的文件：

```

1 // os/src/main.rs
2 #[no_std]
3 #[no_main]
4
5 mod lang_items;

```

(下页继续)

(续上页)

```

6
7 use core::arch::global_asm;
8 global_asm!(include_str!("entry.asm"));

```

第 8 行，我们通过 `include_str!` 宏将同目录下的汇编代码 `entry.asm` 转化为字符串并通过 `global_asm!` 宏嵌入到代码中。

2.5.3 调整内核的内存布局

由于链接器默认的内存布局并不能符合我们的要求，为了实现与 Qemu 正确对接，我们可以通过 **链接脚本 (Linker Script)** 调整链接器的行为，使得最终生成的可执行文件的内存布局符合 Qemu 的预期，即内核第一条指令的地址应该位于 `0x80200000`。我们修改 Cargo 的配置文件来使用我们自己的链接脚本 `os/src/linker.ld` 而非使用默认的内存布局：

```

1 // os/.cargo/config
2 [build]
3 target = "riscv64gc-unknown-none-elf"
4
5 [target.riscv64gc-unknown-none-elf]
6 rustflags = [
7     "-Clink-arg=-Tsrc/linker.ld", "-Cforce-frame-pointers=yes"
8 ]

```

链接脚本 `os/src/linker.ld` 如下：

```

1 OUTPUT_ARCH(riscv)
2 ENTRY(_start)
3 BASE_ADDRESS = 0x80200000;
4
5 SECTIONS
6 {
7     . = BASE_ADDRESS;
8     skernel = .;
9
10    stext = .;
11    .text : {
12        *(.text.entry)
13        *(.text .text.*)
14    }
15
16    . = ALIGN(4K);
17    etext = .;
18    srodata = .;
19    .rodata : {
20        *(.rodata .rodata.*)
21        *(.srodata .srodata.*)
22    }
23
24    . = ALIGN(4K);
25    erodata = .;
26    sdata = .;
27    .data : {
28        *(.data .data.*)
29        *(.sdata .sdata.*)
30    }

```

(下页继续)

(续上页)

```

31     . = ALIGN(4K);
32     edata = .;
33     .bss : {
34         *(.bss.stack)
35         sbss = .;
36         *(.bss .bss.*)
37         *(.sbss .sbss.*)
38     }
39
40     . = ALIGN(4K);
41     ebss = .;
42     ekernel = .;
43
44     /DISCARD/ : {
45         *(.eh_frame)
46     }
47 }
48 }
```

第 1 行我们设置了目标平台为 riscv；第 2 行我们设置了整个程序的入口点为之前定义的全局符号 _start；第 3 行定义了一个常量 BASE_ADDRESS 为 0x80200000，也就是我们之前提到内核的初始化代码被放置的地址；

从第 5 行开始体现了链接过程中对输入的目标文件的段的合并。其中 . 表示当前地址，也就是链接器会从它指向的位置开始往下放置从输入的目标文件中收集来的段。我们可以对 . 进行赋值来调整接下来的段放在哪里，也可以创建一些全局符号赋值为 . 从而记录这一时刻的位置。我们还能够看到这样的格式：

```

.rodata : {
    *(.rodata)
}
```

冒号前面表示最终生成的可执行文件的一个段的名字，花括号内按照放置顺序描述将所有输入目标文件的哪些段放在这个段中，每一行格式为 <ObjectFile>(<SectionName>)，表示目标文件 ObjectFile 的名为 SectionName 的段需要被放进去。我们也可以使用通配符来书写 <ObjectFile> 和 <SectionName> 分别表示可能的输入目标文件和段名。因此，最终的合并结果是，在最终可执行文件中各个常见的段 .text, .rodata, .data, .bss 从低地址到高地址按顺序放置，每个段里面都包括了所有输入目标文件的同名段，且每个段都有两个全局符号给出了它的开始和结束地址（比如 .text 段的开始和结束地址分别是 stext 和 etext）。

第 12 行我们将包含内核第一条指令的 .text.entry 段放在最终的 .text 段的最开头，同时注意到在最终内存布局中代码段 .text 又是先于任何其他段的。因为所有的段都从 BASE_ADDRESS 也即 0x80200000 开始放置，这就能够保证内核的第一条指令正好放在 0x80200000 从而能够正确对接到 Qemu 上。

此后我们便可以生成内核可执行文件，切换到 os 目录下并进行以下操作：

```

$ cargo build --release
Finished release [optimized] target(s) in 0.10s
$ file target/riscv64gc-unknown-none-elf/release/os
target/riscv64gc-unknown-none-elf/release/os: ELF 64-bit LSB executable, UCB RISC-V, ↴
  version 1 (SYSV), statically linked, not stripped
```

我们以 release 模式生成了内核可执行文件，它的位置在 os/target/riscv64gc.../release/os。接着我们通过 file 工具查看它的属性，可以看到它是一个运行在 64 位 RISC-V 架构计算机上的可执行文件，它是静态链接得到的。

注解：思考：0x80200000 可否改为其他地址？

首先需要区分绝对地址和相对地址。在对编译器进行某些设置的情况下，在访问变量或函数时，可以通过它们所在地址与当前某个寄存器（如 PC）的相对地址而非它们位于的绝对地址来访问这些变量或函数。比如，在一个起始地址（即上面提到的 `BASE_ADDRESS`）固定为 `0x80200000` 的内存布局中，某个函数入口位于 `0x80201111` 处，那么我们可以使用其绝对地址 `0x80201111` 来访问它。但是，如果一条位于 `0x80200111` 指令会调用该函数，那么这条指令也不一定要用到绝对地址 `0x80201111`，而是用函数入口地址相对于当前指令地址 `0x80200111` 的相对地址 `0x1000`（计算方式为函数入口地址与当前指令地址之差值）来找到并调用该函数。

如果一个程序全程都使用相对地址而不依赖任何绝对地址，那么只要保持好各段之间的相对位置不发生变化，将程序整体加载到内存中的任意位置程序均可正常运行。在这种情况下，`BASE_ADDRESS` 可以为任意值，我们可以将程序在内存中随意平移。这种程序被称为 **位置无关可执行文件 (PIE, Position-independent Executable)**。相对的，如果程序依赖绝对地址，那么它一定有一个确定的内存布局，而且该程序必须被加载到与其内存布局一致的位置才能正常运行。由于我们的内核并不是位置无关的，所以我们必须将内存布局的起始地址设置为 `0x80200000`，与之匹配我们也必须将内核加载到这一地址。

注解：静态链接与动态链接

静态链接是指程序在编译时就将所有用到的函数库的目标文件链接到可执行文件中，这样会导致可执行文件容量较大，占用硬盘空间；而动态链接是指程序在编译时仅在可执行文件中记录用到哪些函数库和在这些函数库中用到了哪些符号，在操作系统执行该程序准备将可执行文件加载到内存时，操作系统会检查这些被记录的信息，将用到的函数库的代码和数据和程序一并加载到内存，并进行一些重定位工作，即对装入内存的目标程序中的指令或数据的内存地址进行修改，确保程序运行时能正确找到相关函数或数据。使用动态链接可以显著缩减可执行文件的容量，并使得程序不必在函数库更新后重新链接依然可用。

根据以往的经验，Qemu 模拟的计算机不支持在加载时动态链接，因此我们的内核采用静态链接进行编译。

2.5.4 手动加载内核可执行文件

上面得到的内核可执行文件完全符合我们对于内存布局的要求，但是我们不能将其直接提交给 Qemu，因为它除了实际会被用到的代码和数据段之外还有一些多余的元数据，这些元数据无法被 Qemu 在加载文件时利用，且会使代码和数据段被加载到错误的位置。如下图所示：

图中，红色的区域表示内核可执行文件中的元数据，深蓝色的区域表示各个段（包括代码段和数据段），而浅蓝色区域则表示内核被执行的第一条指令，它位于深蓝色区域的开头。图示的上半部分中，我们直接将内核可执行文件 `os` 提交给 Qemu，而 Qemu 会将整个可执行文件不加处理的加载到 Qemu 内存的 `0x80200000` 处，由于内核可执行文件的开头是一段元数据，这会导致 Qemu 内存 `0x80200000` 处无法找到内核第一条指令，也就意味着 RustSBI 无法正常将计算机控制权转交给内核。相反，图示的下半部分中，将元数据丢弃得到的内核镜像 `os.bin` 被加载到 Qemu 之后，则可以在 `0x80200000` 处正确找到内核第一条指令。如果想要深入了解这些元数据的内容，可以参考附录 B：常见工具的使用方法。

使用如下命令可以丢弃内核可执行文件中的元数据得到内核镜像：

```
$ rust-objcopy --strip-all target/riscv64gc-unknown-none-elf/release/os -O binary
→ target/riscv64gc-unknown-none-elf/release/os.bin
```

我们可以使用 `stat` 工具来比较内核可执行文件和内核镜像的大小：

```
$ stat target/riscv64gc-unknown-none-elf/release/os
File: target/riscv64gc-unknown-none-elf/release/os
Size: 1016          Blocks: 8          IO Block: 4096   regular file
...
$ stat target/riscv64gc-unknown-none-elf/release/os.bin
File: target/riscv64gc-unknown-none-elf/release/os.bin
```

(下页继续)

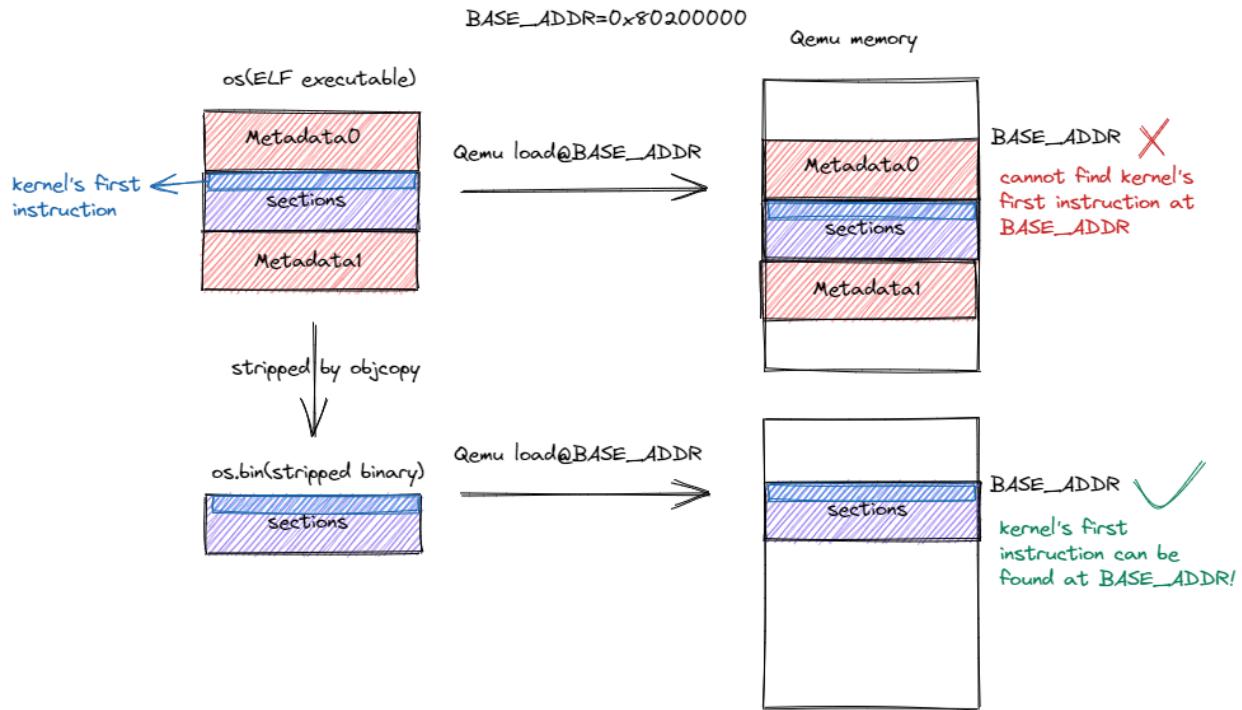


图 4: 丢弃元数据前后的内核可执行文件被加载到 Qemu 上的情形

(续上页)

Size: 4	Blocks: 8	IO Block: 4096	regular file
...			

可以看到，内核镜像的大小仅有 4 字节，这是因为它里面仅包含我们在 `entry.asm` 中编写的一条指令。一般情况下 RISC-V 架构的一条指令位宽即为 4 字节。而内核可执行文件由于包含了两部分元数据，其大小达到了 1016 字节。这些元数据能够帮助我们更加灵活地加载并使用可执行文件，比如在加载时完成一些重定位工作或者动态链接。不过由于 Qemu 的加载功能过于简单，我们只能将这些元数据丢弃再交给 Qemu。从某种意义上可以理解为我们手动帮助 Qemu 完成了可执行文件的加载。

注解: 新版 Qemu 支持直接加载 ELF

经过我们的实验，至少在 Qemu 7.0.0 版本后，我们可以直接将内核可执行文件 `os` 提交给 Qemu 而不必进行任何元数据的裁剪工作，这种情况下我们的内核也能正常运行。其具体做法为：将 Qemu 的参数替换为 `-device loader, file=path/to/os`。但是，我们仍推荐大家了解并在代码框架和文档中保留这一流程，原因在于这种做法更加通用，对环境和工具的依赖程度更低。

2.5.5 基于 GDB 验证启动流程

在 os 目录下通过以下命令启动 Qemu 并加载 RustSBI 和内核镜像：

```

1 $ qemu-system-riscv64 \
2   -machine virt \
3   -nographic \
4   -bios ../bootloader/rustsbi-qemu.bin \
5   -device loader,file=target/riscv64gc-unknown-none-elf/release/os.bin,
6   ↪addr=0x80200000 \
7   -s -S

```

-s 可以使 Qemu 监听本地 TCP 端口 1234 等待 GDB 客户端连接，而 -S 可以使 Qemu 在收到 GDB 的请求后再开始运行。因此，Qemu 暂时没有任何输出。注意，如果不通过 GDB 对于 Qemu 进行调试而是直接运行 Qemu 的话，则要删掉最后一行的 -s -S。

打开另一个终端，启动一个 GDB 客户端连接到 Qemu：

```

$ riscv64-unknown-elf-gdb \
  -ex 'file target/riscv64gc-unknown-none-elf/release/os' \
  -ex 'set arch riscv:rv64' \
  -ex 'target remote localhost:1234'
[GDB output]
0x0000000000001000 in ?? ()

```

可以看到，正如我们在上一节提到的那样，Qemu 启动后 PC 被初始化为 0x1000。我们可以检查一下 Qemu 的启动固件的内容：

```

$ (gdb) x/10i $pc
=> 0x1000: auipc  t0,0x0
0x1004: addi   a1,t0,32
0x1008: csrr   a0,mhartid
0x100c: ld     t0,24(t0)
0x1010: jr     t0
0x1014: unimp
0x1016: unimp
0x1018: unimp
0x101a: 0x8000
0x101c: unimp

```

这里 x/10i \$pc 的含义是从当前 PC 值的位置开始，在内存中反汇编 10 条指令。不过可以看到 Qemu 的固件仅包含 5 条指令，从 0x1014 开始都是数据，当数据为 0 的时候则会被反汇编为 unimp 指令。0x101a 处的数据 0x8000 是能够跳转到 0x80000000 进入启动下一阶段的关键。有兴趣的读者可以自行探究位于 0x1000 和 0x100c 两条指令的含义。总之，在执行位于 0x1010 的指令之前，寄存器 t0 的值恰好为 0x80000000，随后通过 jr t0 便可以跳转到该地址。我们可以通过单步调试来复盘这个过程：

```

$ (gdb) si
0x0000000000001004 in ?? ()
$ (gdb) si
0x0000000000001008 in ?? ()
$ (gdb) si
0x000000000000100c in ?? ()
$ (gdb) si
0x0000000000001010 in ?? ()
$ (gdb) p/x $t0
$1 = 0x80000000
$ (gdb) si
0x0000000080000000 in ?? ()

```

其中, `si` 可以让 Qemu 每次向下执行一条指令, 之后屏幕会打印出待执行的下一条指令的地址。`p/x $t0` 以 16 进制打印寄存器 `t0` 的值, 注意当我们要打印寄存器的时候需要在寄存器的名字前面加上 `$`。可以看到, 当位于 `0x1010` 的指令执行完毕后, 下一条待执行的指令位于 RustSBI 的入口, 也即 `0x80000000`, 这意味着我们即将把控制权转交给 RustSBI。

```
$ (gdb) x/10i $pc
=> 0x80000000:    auipc    sp,0x28
0x80000004: mv    sp,sp
0x80000008: lui    t0,0x4
0x8000000a: addi   t1,a0,1
0x8000000e: add    sp,sp,t0
0x80000010: addi   t1,t1,-1
0x80000012: bnez   t1,0x8000000e
0x80000016: j     0x8001125a
0x8000001a: unimp
0x8000001c: addi   sp,sp,-48
$ (gdb) si
0x0000000080000004 in ?? ()
$ (gdb) si
0x0000000080000008 in ?? ()
$ (gdb) si
0x000000008000000a in ?? ()
$ (gdb) si
0x000000008000000e in ?? ()
```

我们可以用同样的方式反汇编 RustSBI 最初的几条指令并单步调试。不过由于 RustSBI 超出了本书的范围, 我们这里并不打算进行深入。接下来我们检查控制权能否被移交给我们的内核:

```
$ (gdb) b *0x80200000
Breakpoint 1 at 0x80200000
$ (gdb) c
Continuing.

Breakpoint 1, 0x0000000080200000 in ?? ()
```

我们在内核的入口点, 也即地址 `0x80200000` 处打一个断点。需要注意, 当需要在一个特定的地址打断点时, 需要在地址前面加上 `*`。接下来通过 `c` 命令 (Continue 的缩写) 让 Qemu 向下运行直到遇到一个断点。可以看到, 我们成功停在了 `0x80200000` 处。随后, 可以检查内核第一条指令是否被正确执行:

```
$ (gdb) x/5i $pc
=> 0x80200000:    li      ra,100
0x80200004: unimp
0x80200006: unimp
0x80200008: unimp
0x8020000a: unimp
$ (gdb) si
0x0000000080200004 in ?? ()
$ (gdb) p/d $x1
$2 = 100
$ (gdb) p/x $sp
$3 = 0x0
```

可以看到我们在 `entry.asm` 中编写的第一条指令可以在 `0x80200000` 处找到。这里 `ra` 是寄存器 `x1` 的别名, `p/d $x1` 可以以十进制打印寄存器 `x1` 的值, 它的结果正确。最后, 作为下一节的铺垫, 我们可以检查此时栈指针 `sp` 的值, 可以发现它目前是 0。下一节我们将设置好栈空间, 使得内核代码可以正常进行函数调用, 随后将控制权转交给 Rust 代码。

2.6 为内核支持函数调用

2.6.1 本节导读

上一节我们成功在 Qemu 上执行了内核的第一条指令，它是在 `entry.asm` 中手写汇编代码得到的。然而，我们无论如何也不想仅靠手写汇编代码的方式编写我们的内核，绝大部分功能我们都想使用 Rust 语言来实现。不过为了将控制权转交给使用 Rust 语言编写的内核入口函数，我们确实需要手写若干行汇编代码进行一定的初始化工作。和之前一样，这些汇编代码放在 `entry.asm` 中，并在控制权被转交给内核相关函数前最先被执行，但它们的功能会更加复杂。首先需要设置栈空间，来在内核内使能函数调用，随后直接调用使用 Rust 编写的内核入口函数，从而控制权便被移交给 Rust 代码。这就是构建“三叶虫”操作系统的第三个步骤。

在具体操作之前，我们首先会介绍很多函数调用和栈的背景知识。这些知识很重要，而且有一些思想会一直延续到后面的章节。但同时这些知识相对比较基础，因此我们在正式开始介绍之前给出了一个知识点清单，有一定基础的读者可以参照此清单进行选读。

注解：本节知识清单

请尝试回答以下问题，如果对于自己的答案足够自信的话则可以直接进入本节的实践部分。

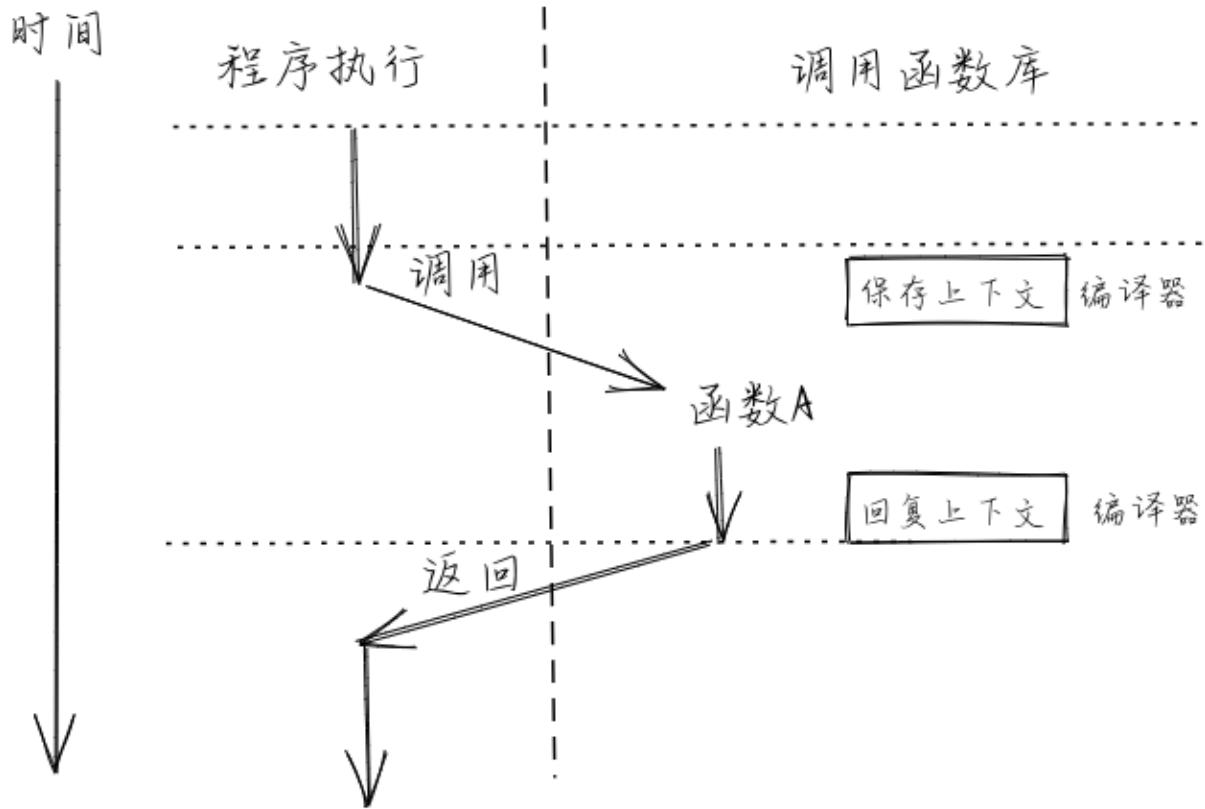
- 如何使得函数返回时能够跳转到调用该函数的下一条指令，即使该函数在代码中的多个位置被调用？
 - 对于一个函数而言，保证它调用某个子函数之前，以及该子函数返回到它之后（某些）通用寄存器的值保持不变有何意义？
 - 调用者函数和被调用者函数如何合作保证调用子函数前后寄存器内容保持不变？调用者保存和被调用者保存寄存器的保存与恢复各自由谁负责？它们暂时被保存在什么位置？它们于何时被保存和恢复（如函数的开场白/退场白）？
 - 在 RISC-V 架构上，调用者保存和被调用者保存寄存器如何划分的？
 - `sp` 和 `ra` 是调用者还是被调用者保存寄存器，为什么这样约定？
 - 如何使用寄存器传递函数调用的参数和返回值？如果寄存器数量不够用了，如何传递函数调用的参数？
-

2.6.2 函数调用与栈

从汇编指令的级别看待一段程序的执行，假如 CPU 依次执行的指令的物理地址序列为 $\{a_n\}$ ，那么这个序列会符合怎样的模式呢？

其中最简单的无疑就是 CPU 一条条连续向下执行指令，也即满足递推公式 $a_{n+1} = a_n + L$ ，这里我们假设该平台的指令是定长的且均为 L 字节（常见情况为 2/4 字节）。但是执行序列并不总是符合这种模式，当位于物理地址 a_n 的指令是一条跳转指令的时候，该模式就有可能被破坏。跳转指令对应于我们在程序中构造的 **控制流** (Control Flow) 的多种不同结构，比如分支结构（如 `if/switch` 语句）和循环结构（如 `for/while` 语句）。用来实现上述两种结构的跳转指令，只需实现跳转功能，也就是将 `pc` 寄存器设置到一个指定的地址即可。

另一种控制流结构则显得更为复杂：**函数调用** (Function Call)。我们大概清楚调用函数整个过程中代码执行的顺序，如果是从源代码级的视角来看，我们会去执行被调用函数的代码，等到它返回之后，我们会回到调用函数对应语句的下一行继续执行。那么我们如何用汇编指令来实现这一过程？首先在调用的时候，需要有一条指令跳转到被调用函数的位置，这个看起来和其他控制结构没什么不同；但是在被调用函数返回的时候，我们却需要返回那条跳转过来的指令的下一条继续执行。这次用来返回的跳转究竟跳转到何处，在对应的函数调用发生之前是不知道的。比如，我们在两个不同的地方调用同一个函数，显然函数返回之后会回到不同的地址。这是一个很大的不同：其他控制流都只需要跳转到一个 编译期固定下来的地址，而函数调用的返回跳转是跳转到一个运行时确定（确切地说是在函数调用发生的时候）的地址。



对此，指令集必须给用于函数调用的跳转指令一些额外的能力，而不只是单纯的跳转。在 RISC-V 架构上，有两条指令即符合这样的特征：

表 1: RISC-V 函数调用跳转指令

指令	指令功能
jal rd, imm[20 : 1]	$rd \leftarrow pc + 4$ $pc \leftarrow pc + imm$
jalr rd, (imm[11 : 0])rs	$rd \leftarrow pc + 4$ $pc \leftarrow rs + imm$

注解: RISC-V 指令各部分含义

在大多数只与通用寄存器打交道的指令中，rs 表示 **源寄存器** (Source Register)，imm 表示 **立即数** (Immediate)，是一个常数，二者构成了指令的输入部分；而 rd 表示 **目标寄存器** (Destination Register)，它是指令的输出部分。rs 和 rd 可以在 32 个通用寄存器 x0~x31 中选取。但是这三个部分都不是必须的，某些指令只有一种输入类型，另一些指令则没有输出部分。

从中可以看出，这两条指令在设置 pc 寄存器完成跳转功能之前，还将当前跳转指令的下一条指令地址保存在 rd 寄存器中，即 $rd \leftarrow pc + 4$ 这条指令的含义。（这里假设所有指令的长度均为 4 字节）在 RISC-V 架构中，通常使用 ra 寄存器（即 x1 寄存器）作为其中的 rd 对应的具体寄存器，因此在函数返回的时候，只需跳转回 ra 所保存的地址即可。事实上在函数返回的时候我们常常使用一条 **汇编伪指令** (Pseudo Instruction) 跳转回调用之前的位置：ret。它会被汇编器翻译为 `jalr x0, 0(x1)`，含义为跳转到寄存器 ra 保存的物理地址，由于 x0 是一个恒为 0 的寄存器，在 rd 中保存这一步被省略。

总结一下，在进行函数调用的时候，我们通过 jalr 指令保存返回地址并实现跳转；而在函数即将返回的时候，则通过 ret 伪指令回到跳转之前的下一条指令继续执行。这样，RISC-V 的这两条指令就实现了函数调

用流程的核心机制。

由于我们是在 `ra` 寄存器中保存返回地址的，我们要保证它在函数执行的全程不发生变化，不然在 `ret` 之后就会跳转到错误的位置。事实上编译器除了函数调用的相关指令之外确实基本上不使用 `ra` 寄存器。也就是说，如果在函数中没有调用其他函数，那 `ra` 的值不会变化，函数调用流程能够正常工作。但遗憾的是，在实际编写代码的时候我们常常会遇到函数 **多层嵌套调用** 的情形。我们很容易想象，如果函数不支持嵌套调用，那么编程将会变得多么复杂。如果我们试图在一个函数 `f` 中调用一个子函数，在跳转到子函数 `g` 的同时，`ra` 会被覆盖成这条跳转指令的下一条的地址，而 `ra` 之前所保存的函数 `f` 的返回地址将会永久丢失。

因此，若想正确实现嵌套函数调用的控制流，我们必须通过某种方式保证：在一个函数调用子函数的前后，`ra` 寄存器的值不能发生变化。但实际上，这并不仅仅局限于 `ra` 一个寄存器，而是作用于所有的通用寄存器。这是因为，编译器是独立编译每个函数的，因此一个函数并不能知道它所调用的子函数修改了哪些寄存器。而站在一个函数的视角，在调用子函数的过程中某些寄存器的值被覆盖的确会对它接下来的执行产生影响。因此这是必要的。我们将由于函数调用，在控制流转移前后需要保持不变的寄存器集合称之为 **函数调用上下文** (Function Call Context)。

由于每个 CPU 只有一套寄存器，我们若想在子函数调用前后保持函数调用上下文不变，就需要物理内存的帮助。确切的说，在调用子函数之前，我们需要在物理内存中的一个区域 **保存** (Save) 函数调用上下文中的寄存器；而在函数执行完毕后，我们会从内存中同样的区域读取并 **恢复** (Restore) 函数调用上下文中的寄存器。实际上，这一工作是由子函数的调用者和被调用者（也就是子函数自身）合作完成。函数调用上下文中的寄存器被分为如下两类：

- **被调用者保存 (Callee-Saved) 寄存器**：被调用的函数可能会覆盖这些寄存器，需要被调用的函数来保存的寄存器，即由被调用的函数来保证在调用前后，这些寄存器保持不变；
- **调用者保存 (Caller-Saved) 寄存器**：被调用的函数可能会覆盖这些寄存器，需要发起调用的函数来保存的寄存器，即由发起调用的函数来保证在调用前后，这些寄存器保持不变。

从名字中可以看出，函数调用上下文由调用者和被调用者分别保存，其具体过程分别如下：

- 调用函数：首先保存不希望在函数调用过程中发生变化的 **调用者保存寄存器**，然后通过 `jal/jalr` 指令调用子函数，返回之后恢复这些寄存器。
- 被调用函数：在被调用函数的起始，先保存函数执行过程中被用到的 **被调用者保存寄存器**，然后执行函数，最后在函数退出之前恢复这些寄存器。

我们发现无论是调用函数还是被调用函数，都会因调用行为而需要两段匹配的保存和恢复寄存器的汇编代码，可以分别将其称为 **开场 (Prologue)** 和 **结尾 (Epilogue)**，它们会由编译器帮我们自动插入，来完成相关寄存器的保存与恢复。一个函数既有可能作为调用者调用其他函数，也有可能作为被调用者被其他函数调用。

注解：寄存器保存与编译器优化

这里值得说明的是，调用者和被调用者实际上只需分别按需保存调用者保存寄存器和被调用者保存寄存器的一个子集。对于调用函数而言，在调用子函数的时候，即使子函数修改了调用者保存寄存器，编译器在调用函数中插入的代码会恢复这些寄存器；而对于被调用函数而言，在其执行过程中没有使用到的被调用者保存寄存器也无需保存。编译器在进行后端代码生成时，知道在这两个场景中分别有哪些值得保存的寄存器。从这一角度也可以理解为何要将函数调用上下文分成两类：可以让编译器尽可能早地优化掉一些无用的寄存器保存与恢复操作，提高程序的执行性能。

2.6.3 调用规范

调用规范 (Calling Convention) 约定在某个指令集架构上，某种编程语言的函数调用如何实现。它包括了以下内容：

1. 函数的输入参数和返回值如何传递；
2. 函数调用上下文中调用者/被调用者保存寄存器的划分；
3. 其他的在函数调用流程中对于寄存器的使用方法。

调用规范是对于一种确定的编程语言来说的，因为一般意义上的函数调用只会在编程语言的内部进行。当一种语言想要调用用另一门编程语言编写的函数接口时，编译器就需要同时清楚两门语言的调用规范，并对寄存器的使用做出调整。

注解：RISC-V 架构上的 C 语言调用规范

RISC-V 架构上的 C 语言调用规范可以在 [这里](#) 找到。它对通用寄存器的使用做出了如下约定：

表 2: RISC-V 寄存器功能分类

寄存器组	保存者	功能
a0~a7 (x10~x17)	调用者保存	用来传递输入参数。其中的 a0 和 a1 还用来保存返回值。
t0~t6(x5~x7, x28~x31)	调用者保存	作为临时寄存器使用，在被调函数中可以随意使用无需保存。
s0~s11(x8~x9, x18~x27)	被调用者保存	作为临时寄存器使用，被调函数保存后才能在被调函数中使用。

剩下的 5 个通用寄存器情况如下：

- zero(x0) 之前提到过，它恒为零，函数调用不会对它产生影响；
- ra(x1) 是被调用者保存的。被调用者函数可能也会调用函数，在调用之前就需要修改 ra 使得这次调用能正确返回。因此，每个函数都需要在开头保存 ra 到自己的栈帧中，并在结尾使用 ret 返回之前将其恢复。栈帧是当前执行函数用于存储局部变量和函数返回信息的内存结构。
- sp(x2) 是被调用者保存的。这个是之后就会提到的栈指针 (Stack Pointer) 寄存器，它指向下一个将要被存储的栈顶位置。
- fp(x0)，它既可作为 s0 临时寄存器，也可作为栈帧指针 (Frame Pointer) 寄存器，表示当前栈帧的起始位置，是一个被调用者保存寄存器。fp 指向的栈帧起始位置和 sp 指向的栈帧的当前栈顶位置形成了所对应函数栈帧的空间范围。
- gp(x3) 和 tp(x4) 在一个程序运行期间都不会变化，因此不必放在函数调用上下文中。它们的用途在后面的章节会提到。

更加详细的内容可以参考 Cornell 大学的 [CS 3410: Computer System Organization and Programming](#) 课件内容。

之前我们讨论了函数调用上下文的保存/恢复时机以及寄存器的选择，但我们并没有详细说明这些寄存器保存在哪里，只是用“内存中的一块区域”草草带过。实际上，它更确切的名字是 **栈** (Stack)。sp 寄存器常用来保存 **栈指针** (Stack Pointer)，它指向内存中栈顶地址。在 RISC-V 架构中，栈是从高地址向低地址增长的。在一个函数中，作为起始的开场代码负责分配一块新的栈空间，即将 sp 的值减小相应的字节数即可，于是物理地址区间 [新 sp, 旧 sp) 对应的物理内存的一部分便可以被这个函数用来进行函数调用上下文的保存/恢复，这块物理内存被称为这个函数的 **栈帧** (Stack Frame)。同理，函数中的结尾代码负责将开场代码分配的栈帧回收，这也仅仅需要将 sp 的值增加相同的字节数回到分配之前的状态。这也解释为什么 sp 是一个被调用者保存寄存器。

注解：栈帧 stack frame

我们知道程序在执行函数调用时，调用者函数和被调用函数使用的是同一个栈。在通常的情况下，我们并不需要区分调用者函数和被调用函数分别使用了栈的哪个部分。但是，当我们需要在执行过程中对函数调用进行调试或 backtrace 的时候，这一信息就很重要了。简单的说，栈帧 (stack frame) 就是一个函数所使用的栈的一部分区域，所有函数的栈帧串起来就组成了一个完整的函数调用栈。一般而言，当前执行函数的栈帧的两个边界分别由栈指针 (Stack Pointer) 寄存器和栈帧指针 (frame pointer) 寄存器来限定。

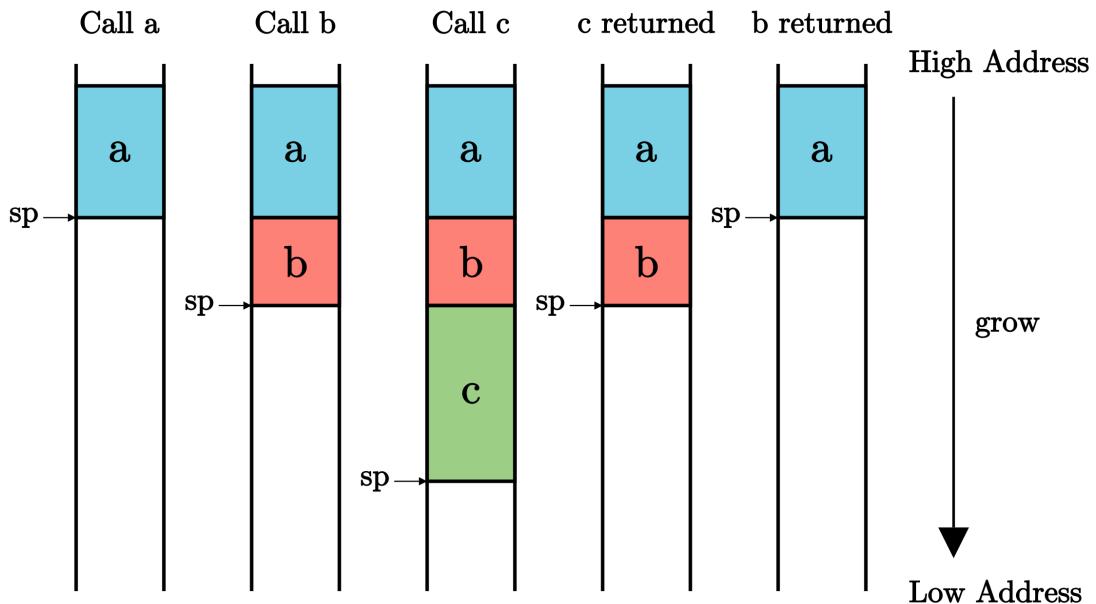


图 5: 函数调用与栈帧：如图所示，我们能够看到在程序依次调用 a、调用 b、调用 c、c 返回、b 返回整个过程中栈帧的分配/回收以及 sp 寄存器的变化。图中标有 a/b/c 的块分别代表函数 a/b/c 的栈帧。

注解：数据结构中的栈与实现函数调用所需要的栈

从数据结构的角度来看，栈是一个 **后入先出** (Last In First Out, LIFO) 的线性表，支持向栈顶压入一个元素以及从栈顶弹出一个元素两种操作，分别被称为 push 和 pop。从它提供的接口来看，它只支持访问栈顶附近的元素。因此在实现的时候需要维护一个指向栈顶的指针来表示栈当前的状态。

我们这里的栈与数据结构中的栈原理相同，在很多方面可以一一对应。栈指针 sp 可以对应到指向栈顶的指针，对于栈帧的分配/回收可以分别对应到 push / pop 操作。如果将我们的栈看成一个内存分配器，它之所以可以这么简单，是因为它回收的内存一定是最近一次分配的内存，从而只需要类似 push / pop 的两种操作即可。

在合适的编译选项设置之下，一个函数的栈帧内容可能如下图所示：

它的开头和结尾分别在 sp(x2) 和 fp(s0) 所指向的地址。按照地址从高到低分别有以下内容，它们都是通过 sp 加上一个偏移量来访问的：

- ra 寄存器保存其返回之后的跳转地址，是一个被调用者保存寄存器；
- 父亲栈帧的结束地址 fp，是一个被调用者保存寄存器；

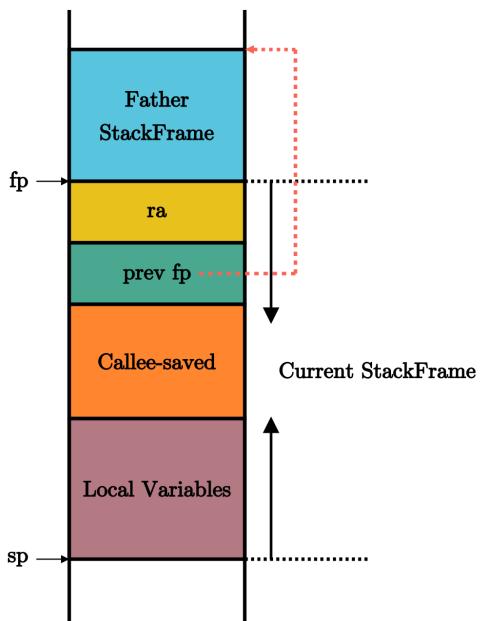


图 6: 函数栈帧中的内容

- 其他被调用者保存寄存器 $s_1 \sim s_{11}$ ；
- 函数所使用到的局部变量。

因此，栈上多个 fp 信息实际上保存了一条完整的函数调用链，通过适当的方式我们可以实现对函数调用关系的跟踪。

ra 、 sp 和 fp 是和函数调用紧密相关的寄存器，我们用一个例子来展示真实编译器生成的汇编代码会如何使用这些寄存器。首先，无论对于内核本身还是第二章后出现的应用程序，我们修改 `.cargo/config`：

```
// .cargo/config

[build]
target = "riscv64gc-unknown-none-elf"

[target.riscv64gc-unknown-none-elf]
rustflags = [
    "-Clink-args=-Tsrc/linker.ld", "-Cforce-frame-pointers=yes"
]
```

这可以设置我们的默认编译目标，同时调整编译选项，设置链接脚本以及强制打开 fp 选项，这样才会避免 fp 相关指令被编译器优化掉。随后，我们可以使用 `rust-objdump` 工具反汇编内核或者应用程序可执行文件，并找到某个函数的入口。然后，我们能够看到在函数的开场和结尾阶段，编译器会生成类似的汇编代码：

```
# 开场
# 为当前函数分配 64 字节的栈帧
addi    sp, sp, -64
# 将 ra 和 fp 压栈保存
sd    ra, 56(sp)
sd    s0, 48(sp)
# 更新 fp 为当前函数栈帧顶端地址
```

(下页继续)

(续上页)

```

1 addi      s0, sp, 64
2
3 # 函数执行
4 # 中间如果再调用了其他函数会修改 ra
5
6 # 结尾
7 # 恢复 ra 和 fp
8 ld ra, 56(sp)
9 ld s0, 48(sp)
10 # 退栈
11 addi      sp, sp, 64
12 # 返回, 使用 ret 指令或其他等价的实现方式
13 ret

```

至此, 我们基本上说明了函数调用是如何基于栈来实现的。不过我们可以暂时先忽略掉这些细节, 因为我们现在只是需要在初始化阶段完成栈的设置, 也就是设置好栈指针 sp 寄存器, 编译器会帮我们自动完成后面的函数调用相关机制的代码生成。麻烦的是, sp 的值也不能随便设置, 至少我们需要保证它指向合法的物理内存, 而且不能与程序的其他代码、数据段相交, 因为在函数调用的过程中, 栈区域里面的内容会被修改。如何保证这一点呢?

2.6.4 分配并使用启动栈

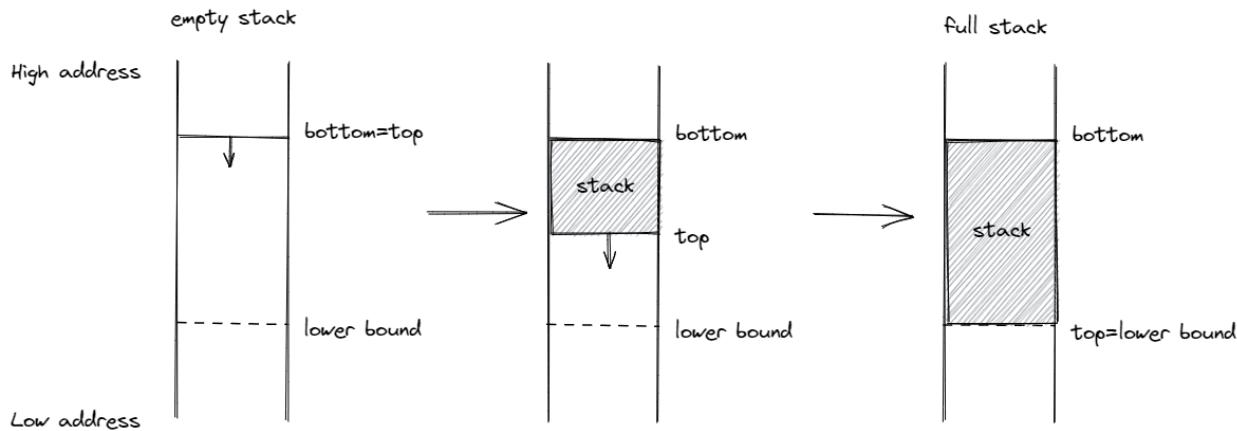
我们在 entry.asm 中分配启动栈空间, 并在控制权被转交给 Rust 入口之前将栈指针 sp 设置为栈顶的位置。

```

1 # os/src/entry.asm
2     .section .text.entry
3     .globl _start
4
5 _start:
6     la sp, boot_stack_top
7     call rust_main
8
9     .section .bss.stack
10    .globl boot_stack_lower_bound
11 boot_stack_lower_bound:
12    .space 4096 * 16
13    .globl boot_stack_top
boot_stack_top:

```

我们在第 11 行在内核的内存布局中预留了一块大小为 $4096 * 16$ 字节也就是 64KiB 的空间用作接下来要运行的程序的栈空间。在 RISC-V 架构上, 栈是从高地址向低地址增长。因此, 最开始的时候栈为空, 栈顶和栈底位于相同的位置, 我们用更高地址的符号 boot_stack_top 来标识栈顶的位置。同时, 我们用更低地址的符号 boot_stack_lower_bound 来标识栈能够增长到的下限位置, 它们都被设置为全局符号供其他目标文件使用。如下图所示:



第 8 行可以看到我们将这块空间放置在一个名为 `.bss.stack` 的段中，在链接脚本 `linker.ld` 中可以看到 `.bss.stack` 段最终会被汇集到 `.bss` 段中：

```
.bss : {
    * (.bss.stack)
    sbss = .;
    * (.bss .bss.*)
    * (.sbss .sbss.*)
}
ebss = .;
```

前面我们提到过 `.bss` 段一般放置需要被初始化为零的数据。然而栈并不需要在使用前被初始化为零，因为在函数调用的时候我们会插入栈帧覆盖已有的数据。我们尝试将其放置到全局数据 `.data` 段中但最后未能成功，因此才决定将其放置到 `.bss` 段中。全局符号 `sbss` 和 `ebss` 分别指向 `.bss` 段除 `.bss.stack` 以外的起始和终止地址，我们在使用这部分数据之前需要将它们初始化为零，这个过程将在下一节进行。

回到 `entry.asm`，可以发现在控制权转交给 Rust 入口之前会执行两条指令，它们分别位于 `entry.asm` 的第 5、6 行。第 5 行我们将栈指针 `sp` 设置为先前分配的启动栈栈顶地址，这样 Rust 代码在进行函数调用和返回的时候就可以正常在启动栈上分配和回收栈帧了。在我们设计好的内存布局中，这块启动栈所用的内存并不会和内核的其他代码、数据段产生冲突，它们是从物理上隔离的。然而如果启动栈溢出（比如在内核代码中出现了太多的函数调用），那么分配的栈帧将有可能覆盖内核其他部分的代码、数据从而出现十分诡异的错误。目前我们只能尽量避免栈溢出的情况发生，到了第四章，借助地址空间抽象和 MMU 硬件的帮助，我们可以做到完全禁止栈溢出。第 6 行我们通过伪指令 `call` 调用 Rust 编写的内核入口点 `rust_main` 将控制权转交给 Rust 代码，该入口点在 `main.rs` 中实现：

```
// os/src/main.rs
#[no_mangle]
pub fn rust_main() -> ! {
    loop {}
}
```

这里需要注意的是需要通过宏将 `rust_main` 标记为 `#[no_mangle]` 以避免编译器对它的名字进行混淆，不然在链接的时候，`entry.asm` 将找不到 `main.rs` 提供的外部符号 `rust_main` 从而导致链接失败。在 `rust_main` 函数的开场白中，我们将第一次在栈上分配栈帧并保存函数调用上下文，它也是内核运行全程中最底层的栈帧。

在内核初始化中，需要先完成对 `.bss` 段的清零。这是内核很重要的一部分初始化工作，在使用任何被分配到 `.bss` 段的全局变量之前我们需要确保 `.bss` 段已被清零。我们就在 `rust_main` 的开头完成这一工作，由于控制权已经被转交给 Rust，我们终于不用手写汇编代码而是可以用 Rust 来实现这一功能了：

```

1 // os/src/main.rs
2 #[no_mangle]
3 pub fn rust_main() -> ! {
4     clear_bss();
5     loop {}
6 }
7
8 fn clear_bss() {
9     extern "C" {
10         fn sbss();
11         fn ebss();
12     }
13     (sbss as usize..ebss as usize).for_each(|a| {
14         unsafe { (a as *mut u8).write_volatile(0) }
15     });
16 }

```

在函数 `clear_bss` 中，我们会尝试从其他地方找到全局符号 `sbss` 和 `ebss`，它们由链接脚本 `linker.ld` 给出，并分别指出需要被清零的 `.bss` 段的起始和终止地址。接下来我们只需遍历该地址区间并逐字节进行清零即可。

注解: Rust Tips: 外部符号引用

`extern "C"` 可以引用一个外部的 C 函数接口（这意味着调用它的时候要遵从目标平台的 C 语言调用规范）。但我们这里只是引用位置标志并将其转成 `usize` 获取它的地址。由此可以知道 `.bss` 段两端的地址。

注解: Rust Tips: 迭代器与闭包

代码第 13 行用到了 Rust 的迭代器与闭包的语法，它们在很多情况下能够提高开发效率。如读者感兴趣的话也可以将其改写为等价的 `for` 循环实现。

警告: Rust Tips: Unsafe

代码第 14 行，我们将 `.bss` 段内的一个地址转化为一个 **裸指针** (Raw Pointer)，并将它指向的值修改为 0。这在 C 语言中是一种司空见惯的操作，但在 Rust 中我们需要将它包裹在 `unsafe` 块中。这是因为，Rust 认为对于裸指针的 **解引用** (Dereference) 是一种 `unsafe` 行为。

相比 C 语言，Rust 进行了更多的语义约束来保证安全性（内存安全/类型安全/并发安全），这在编译期和运行期都有所体现。但在某些时候，尤其是与底层硬件打交道的时候，在 Rust 的语义约束之内没法满足我们的需求，这个时候我们就需要将超出了 Rust 语义约束的行为包裹在 `unsafe` 块中，告知编译器不需要对它进行完整的约束检查，而是由程序员自己负责保证它的安全性。当代码不能正常运行的时候，我们往往也是最先去检查 `unsafe` 块中的代码，因为它没有受到编译器的保护，出错的概率更大。

C 语言中的指针相当于 Rust 中的裸指针，它无所不能但又太过于灵活，程序员对其不谨慎的使用常常会引起很多内存不安全问题，最常见的如悬垂指针和多次回收的问题，Rust 编译器没法确认程序员对它的使用是否安全，因此将其划到 unsafe Rust 的领域。在 safe Rust 中，我们有引用 `&/&mut` 以及各种功能各异的智能指针 `Box<T>/RefCell<T>/Rc<T>` 可以使用，只要按照 Rust 的规则来使用它们便可借助编译器在编译期就解决很多潜在的内存不安全问题。

本节我们介绍了函数调用和栈的背景知识，通过分配栈空间并正确设置栈指针在内核中使能了函数调用并成功将控制权转交给 Rust 代码，从此我们终于可以利用强大的 Rust 语言来编写内核的各项功能了。下一节中我们将进行构建“三叶虫”操作系统的最后一个步骤：即基于 RustSBI 提供的服务成功在屏幕上打印 `Hello, world!`。

2.7 基于 SBI 服务完成输出和关机

2.7.1 本节导读

本节我们将进行构建“三叶虫”操作系统的最后一个步骤，即基于 RustSBI 提供的服务完成在屏幕上打印 `Hello world!` 和关机操作。事实上，作为对我们之前提到的[应用程序执行环境](#)的细化，RustSBI 介于底层硬件和内核之间，是我们内核的底层执行环境。本节将会提到执行环境除了为上层应用进行初始化的第二种职责：即在上层应用运行时提供服务。

2.7.2 使用 RustSBI 提供的服务

之前我们对 RustSBI 的了解仅限于它会在计算机启动时进行它所负责的环境初始化工作，并将计算机控制权移交给内核。但实际上作为内核的执行环境，它还有另一项职责：即在内核运行时响应内核的请求为内核提供服务。当内核发出请求时，计算机会转由 RustSBI 控制来响应内核的请求，待请求处理完毕后，计算机控制权会被交还给内核。从内存布局的角度来思考，每一层执行环境（或称软件栈）都对应到内存中的一段代码和数据，这里的控制权转移指的是 CPU 从执行一层软件的代码到执行另一层软件的代码的过程。这个过程与我们使用高级语言编程时调用库函数比较类似。

这里展开介绍一些相关术语：从第二章将要讲到的[RISC-V 特权级架构](#)的视角来看，我们编写的 OS 内核位于 Supervisor 特权级，而 RustSBI 位于 Machine 特权级，也是最高的特权级。类似 RustSBI 这样运行在 Machine 特权级的软件被称为 Supervisor Execution Environment(SEE)，即 Supervisor 执行环境。两层软件之间的接口被称为 Supervisor Binary Interface(SBI)，即 Supervisor 二进制接口。[SBI Specification](#)（简称 SBI spec）规定了 SBI 接口层要包含哪些功能，该标准由 RISC-V 开源社区维护。RustSBI 按照 SBI spec 标准实现了需要支持的大多数功能，但 RustSBI 并不是 SBI 标准的唯一一种实现，除此之外还有社区中的前辈 OpenSBI 等等。

目前，SBI spec 已经发布了 v2.0-rc8 版本，但本教程基于 2023 年 3 月份发布的 [v1.0.0 版本](#)。我们可以来看看里面约定了 SEE 要向 OS 内核提供哪些功能，并寻找我们本节所需的打印到屏幕和关机的接口。可以看到从 Chapter 4 开始，每一章包含了一个 SBI 拓展（Chapter 5 包含多个 Legacy Extension），代表一类功能接口，这有点像 RISC-V 指令集的 IMAFD 等拓展。每个 SBI 拓展还包含若干子功能。其中：

- Chapter 5 列出了若干 SBI 遗留接口，其中包括串口的写入（正是我们本节所需要的）和读取接口，分别位于 5.2 和 5.3 小节。在教程第九章我们自己实现串口外设驱动之前，与串口的交互都是通过这两个接口来进行的。顺带一提，第三章开始还会用到 5.1 小节介绍的 set timer 接口。
- Chapter 10 包含了若干系统重启相关的接口，我们本节所需的关机接口也在其中。

内核应该如何调用 RustSBI 提供的服务呢？通过函数调用是行不通的，因为内核并没有和 RustSBI 链接到一起，我们仅仅使用 RustSBI 构建后的可执行文件，因此内核无从得知 RustSBI 中的符号或地址。幸而，RustSBI 开源社区的 `sbi_rt` 封装了调用 SBI 服务的接口，我们直接使用即可。首先，我们在 `Cargo.toml` 中引入 `sbi_rt` 依赖：

```
1 // os/Cargo.toml
2 [dependencies]
3 sbi-rt = { version = "0.0.2", features = ["legacy"] }
```

这里需要带上 `legacy` 的 feature，因为我们需要用到的串口读写接口都属于 SBI 的遗留接口。

我们将内核与 RustSBI 通信的相关功能实现在子模块 `sbi` 中，因此我们需要在 `main.rs` 中加入 `mod sbi` 将该子模块加入我们的项目。在 `os/src/sbi.rs` 中，我们直接调用 `sbi_rt` 提供的接口来将输出字符：

```
1 // os/src/sbi.rs
2 pub fn console_putchar(c: usize) {
3     #[allow(deprecated)]
4     sbi_rt::legacy::console_putchar(c);
5 }
```

注意我们为了简单起见并未用到 `sbi_call` 的返回值，有兴趣的同学可以在 SBI spec 中查阅 SBI 服务返回值的含义。到这里，同学们可以试着在 `rust_main` 中调用 `console_putchar` 来在屏幕上输出 OK。接着在 Qemu 上运行一下，我们便可看到由我们自己输出的第一条 log 了。

同样，我们再来实现关机功能：

```

1 // os/src/sbi.rs
2 pub fn shutdown(failure: bool) -> ! {
3     use sbi_rt::system_reset, NoReason, Shutdown, SystemFailure;
4     if !failure {
5         system_reset(Shutdown, NoReason);
6     } else {
7         system_reset(Shutdown, SystemFailure);
8     }
9     unreachable!()
10 }
```

这里的参数 `failure` 表示系统是否正常退出，这会影响 Qemu 模拟器进程退出之后的返回值，我们则会依此判断系统的执行是否正常。更多内容可以参阅 SBI spec 的 Chapter 10。

注解：sbi_rt 是如何调用 SBI 服务的

SBI spec 的 Chapter 3 介绍了服务的调用方法：只需将要调用功能的拓展 ID 和功能 ID 分别放在 `a7` 和 `a6` 寄存器中，并按照 RISC-V 调用规范将参数放置在其他寄存器中，随后执行 `ecall` 指令即可。这会将控制权转交给 RustSBI 并由 RustSBI 来处理请求，处理完成后会将控制权交还给内核。返回值会被保存在 `a0` 和 `a1` 寄存器中。在本书的第二章中，我们会手动编写汇编代码来实现类似的过程。

2.7.3 实现格式化输出

`console_putchar` 的功能过于受限，如果想打印一行 `Hello world!` 的话需要进行多次调用。能否像本章第一节那样使用 `println!` 宏一行就完成输出呢？因此我们尝试自己编写基于 `console_putchar` 的 `println!` 宏。

```

1 // os/src/main.rs
2 #[macro_use]
3 mod console;
4
5 // os/src/console.rs
6 use crate::sbi::console_putchar;
7 use core::fmt::{self, Write};
8
9 struct Stdout;
10
11 impl Write for Stdout {
12     fn write_str(&mut self, s: &str) -> fmt::Result {
13         for c in s.chars() {
14             console_putchar(c as usize);
15         }
16         Ok(())
17     }
18 }
19
20 pub fn print(args: fmt::Arguments) {
21     Stdout.write_fmt(args).unwrap();
22 }
```

(下页继续)

(续上页)

```

23
24 #[macro_export]
25 macro_rules! print {
26     ($fmt: literal $(), $($arg: tt)+)? => {
27         $crate::console::print(format_args!($fmt $(), $($arg)+)?));
28     }
29 }
30
31 #[macro_export]
32 macro_rules! println {
33     ($fmt: literal $(), $($arg: tt)+)? => {
34         $crate::console::print(format_args!(concat!($fmt, "\n") $(), $($arg)+)?));
35     }
36 }

```

我们在 `console` 子模块中编写 `println!` 宏。结构体 `Stdout` 不包含任何字段，因此它被称为类单元结构体 (Unit-like structs，请参考¹)。`core::fmt::Write` trait 包含一个用来实现 `println!` 宏很好用的 `write_fmt` 方法，为此我们准备为结构体 `Stdout` 实现 `Write` trait。在 `Write` trait 中，`write_str` 方法必须实现，因此我们需要为 `Stdout` 实现这一方法，它并不难实现，只需遍历传入的 `&str` 中的每个字符并调用 `console_putchar` 就能将传入的整个字符串打印到屏幕上。

在此之后 `Stdout` 便可调用 `Write` trait 提供的 `write_fmt` 方法并进而实现 `print` 函数。在声明宏 (Declarative macros，参考²) `print!` 和 `println!` 中会调用 `print` 函数完成输出。

现在我们可以在 `rust_main` 中使用 `print!` 和 `println!` 宏进行格式化输出了，如有兴趣的话可以输出 `Hello, world!` 试一下。

注解: Rust Tips: Rust Trait

在 Rust 语言中，trait (中文翻译：特质、特征) 是一种类型，用于描述一组方法的集合。trait 可以用来定义接口 (interface)，并可以被其他类型实现。举个例子，假设我们有一个简单的 Rust 程序，其中有一个名为 `Shape` 的 trait，用于描述形状：

```

1 trait Shape {
2     fn area(&self) -> f64;
3 }

```

我们可以使用这个 trait 来定义一个圆形类型：

```

1 struct Circle {
2     radius: f64,
3 }
4
5 impl Shape for Circle {
6     fn area(&self) -> f64 {
7         3.14 * self.radius * self.radius
8     }
9 }

```

这样，我们就可以使用 `Circle` 类型的实例调用 `area` 方法了。

```

1 let c = Circle { radius: 1.0 };
2 println!("Circle area: {}", c.area()); // 输出: Circle area: 3.14

```

¹ <https://doc.rust-lang.org/book/ch05-01-defining-structs.html#unit-like-structs-without-any-fields>

² https://doc.rust-lang.org/book/ch19-06-macros.html#declarative-macros-with-macro_rules-for-general-metaprogramming

2.7.4 处理致命错误

错误处理是编程的重要一环，它能够保证程序的可靠性和可用性，使得程序能够从容应对更多突发状况而不至于过早崩溃。不同于 C 的返回错误编号 `errno` 模型和 C++/Java 的 `try-catch` 异常捕获模型，Rust 将错误分为可恢复和不可恢复错误两大类。这里我们主要关心不可恢复错误。和 C++/Java 中一个异常被抛出后始终得不到处理一样，在 Rust 中遇到不可恢复错误，程序会直接报错退出。例如，使用 `panic!` 宏便会直接触发一个不可恢复错误并使程序退出。不过在我们的内核中，目前不可恢复错误的处理机制还不完善：

```

1 // os/src/lang_items.rs
2 use core::panic::PanicInfo;
3
4 #[panic_handler]
5 fn panic(_info: &PanicInfo) -> ! {
6     loop {}
7 }
```

可以看到，在目前的实现中，当遇到不可恢复错误的时候，被标记为语义项 `#[panic_handler]` 的 `panic` 函数将会被调用，然而其中只是一个死循环，会使得计算机卡在这里。借助前面实现的 `println!` 宏和 `shutdown` 函数，我们可以在 `panic` 函数中打印错误信息并关机：

```

1 // os/src/main.rs
2 #![feature(panic_info_message)]
3
4 // os/src/lang_item.rs
5 use crate::sbi::shutdown;
6 use core::panic::PanicInfo;
7
8 #[panic_handler]
9 fn panic(info: &PanicInfo) -> ! {
10     if let Some(location) = info.location() {
11         println!(
12             "Panicked at {}:{}:{}",
13             location.file(),
14             location.line(),
15             info.message().unwrap()
16         );
17     } else {
18         println!("Panicked: {}", info.message().unwrap());
19     }
20     shutdown(true)
21 }
```

我们尝试打印更加详细的信息，包括 `panic` 所在的源文件和代码行数。我们尝试从传入的 `PanicInfo` 中解析这些信息，如果解析成功的话，就和 `panic` 的报错信息一起打印出来。我们需要在 `main.rs` 开头加上 `#![feature(panic_info_message)]` 才能通过 `PanicInfo::message` 获取报错信息。当打印完毕之后，我们直接调用 `shutdown` 函数关机，由于系统是异常 `panic` 关机的，参数 `failure` 应为 `true`。

为了测试我们的实现是否正确，我们将 `rust_main` 改为：

```

1 // os/src/main.rs
2 #[no_mangle]
3 pub fn rust_main() -> ! {
4     clear_bss();
5     println!("Hello, world!");
6     panic!("Shutdown machine!");
7 }
```

使用 Qemu 运行我们的内核，运行结果为：

```
[RustSBI output]
Hello, world!
Panicked at src/main.rs:26 Shutdown machine!
```

可以看到, panic 所在的源文件和代码行数被正确报告, 这将为我们后续章节的开发和调试带来很大方便。到这里, 我们就实现了一个可以在 Qemu 模拟的计算机上运行的裸机应用程序, 其具体内容就是上述的 `rust_main` 函数, 而其他部分, 如 `entry.asm`、`lang_items.rs`、`console.rs`、`sbi.rs` 则形成了支持裸机应用程序的寒武纪“三叶虫”操作系统—LibOS。

注解: Rust Tips: Rust 可恢复错误

在有可能出现错误时, Rust 函数的返回值可以属于一种特殊的类型, 该类型可以涵盖两种情况: 要么函数正常退出, 则函数返回正常的返回值; 要么函数执行过程中出错, 则函数返回出错的类型。Rust 的类型系统保证这种返回值不会在程序员无意识的情况下被滥用, 即程序员必须显式对其进行分支判断或者强制排除出错的情况。如果不进行任何处理, 那么无法从中得到有意义的结果供后续使用或是无法通过编译。这样, 就杜绝了很大一部分因程序员的疏忽产生的错误 (如不加判断地使用某函数返回的空指针)。

在 Rust 中有两种这样的特殊类型, 它们都属于枚举结构:

- `Option<T>` 既可以有值 `Option::Some<T>`, 也有可能没有值 `Option::None`;
- `Result<T, E>` 既可以保存某个操作的返回值 `Result::Ok<T>`, 也可以表明操作过程中出现了错误 `Result::Err<E>`。

我们可以使用 `Option/Result` 来保存一个不能确定存在/不存在或是成功/失败的值。之后可以通过匹配 `if let` 或是在能够确定的场合直接通过 `unwrap` 将里面的值取出。详细的内容可以参考 Rust 官方文档³。

2.8 练习

2.8.1 课后练习

编程题

1. * 实现一个 linux 应用程序 A, 显示当前目录下的文件名。(用 C 或 Rust 编程)
2. *** 实现一个 linux 应用程序 B, 能打印出调用栈链信息。(用 C 或 Rust 编程)
3. ** 实现一个基于 rcore/ucore tutorial 的应用程序 C, 用 sleep 系统调用睡眠 5 秒 (in rcore/ucore tutorial v3: Branch ch1)

注: 尝试用 GDB 等调试工具和输出字符串的等方式来调试上述程序, 能设置断点, 单步执行和显示变量, 理解汇编代码和源程序之间的对应关系。

³ <https://doc.rust-lang.org/book/ch09-02-recoverable-errors-with-result.html>

问答题

1. * 应用程序在执行过程中，会占用哪些计算机资源？
2. * 请用相关工具软件分析并给出应用程序 A 的代码段/数据段/堆/栈的地址空间范围。
3. * 请简要说明应用程序与操作系统的异同之处。
4. ** 请基于 QEMU 模拟 RISC-V 的执行过程和 QEMU 源代码，说明 RISC-V 硬件加电后的几条指令在哪里？完成了哪些功能？
5. * RISC-V 中的 SBI 的含义和功能是啥？
6. ** 为了让应用程序能在计算机上执行，操作系统与编译器之间需要达成哪些协议？
7. ** 请简要说明从 QEMU 模拟的 RISC-V 计算机加电开始运行到执行应用程序的第一条指令这个阶段的执行过程。
8. ** 为何应用程序员编写应用时不需要建立栈空间和指定地址空间？
9. *** 现代的很多编译器生成的代码，默认情况下不再严格保存/恢复栈帧指针。在这个情况下，我们只要编译器提供足够的信息，也可以完成对调用栈的恢复。

我们可以手动阅读汇编代码和栈上的数据，体验一下这个过程。例如，对如下两个互相递归调用的函数：

```
void flip(unsigned n) {
    if ((n & 1) == 0) {
        flip(n >> 1);
    } else if ((n & 1) == 1) {
        flap(n >> 1);
    }
}

void flap(unsigned n) {
    if ((n & 1) == 0) {
        flip(n >> 1);
    } else if ((n & 1) == 1) {
        flap(n >> 1);
    }
}
```

在某种编译环境下，编译器产生的代码不包括保存和恢复栈帧指针 fp 的代码。以下是 GDB 输出的本次运行的时候，这两个函数所在的地址和对应地址指令的反汇编，为了方便阅读节选了重要的控制流和栈操作（省略部分不含栈操作）：

```
(gdb) disassemble flap
Dump of assembler code for function flap:
0x00000000000010730 <+0>:    addi    sp,sp,-16    // 唯一入口
0x00000000000010732 <+2>:    sd      ra,8(sp)
...
0x00000000000010742 <+18>:    ld      ra,8(sp)
0x00000000000010744 <+20>:    addi    sp,sp,16
0x00000000000010746 <+22>:    ret
                                // 唯一出口
...
0x00000000000010750 <+32>:    j       0x10742 <flap+18>

(gdb) disassemble flip
Dump of assembler code for function flip:
0x00000000000010752 <+0>:    addi    sp,sp,-16    // 唯一入口
```

(下页继续)

(续上页)

```

0x00000000000010754 <+2>:    sd      ra, 8 (sp)
...
0x00000000000010764 <+18>:   ld      ra, 8 (sp)
0x00000000000010766 <+20>:   addi    sp, sp, 16
0x00000000000010768 <+22>:   ret     // 唯一出口
...
0x00000000000010772 <+32>:   j       0x10764 <flip+18>
End of assembler dump.

```

启动这个程序，在运行的时候的某个状态将其打断。此时的 pc, sp, ra 寄存器的值如下所示。此外，下面还给出了栈顶的部分内容。（为阅读方便，栈上的一些未初始化的垃圾数据用 ??? 代替。）

```

(gdb) p $pc
$1 = (void (*)()) 0x10752 <flip>

(gdb) p $sp
$2 = (void *) 0x40007f1310

(gdb) p $ra
$3 = (void (*)()) 0x10742 <flap+18>

(gdb) x/6a $sp
0x40007f1310: ??? 0x10750 <flap+32>
0x40007f1320: ??? 0x10772 <flip+32>
0x40007f1330: ??? 0x10764 <flip+18>

```

根据给出这些信息，调试器可以如何复原出最顶层的几个调用栈信息？假设调试器可以理解编译器生成的汇编代码¹。

2.8.2 实验练习

实验练习包括实践作业和问答作业两部分。

实践作业

彩色化 LOG

lab1 的工作使得我们从硬件世界跳入了软件世界，当看到自己的小 os 可以在裸机硬件上输出 hello world 是不是很高兴呢？但是为了后续的一步开发，更好的调试环境也是必不可少的，第一章的练习要求大家实现更加炫酷的彩色 log。

详细的原理不多说，感兴趣的同學可以参考 [ANSI 转义序列](#)，现在执行如下这条命令试试

```
$ echo -e "\x1b[31mhello world\x1b[0m"
```

如果你明白了我们是如何利用串口实现输出，那么要实现彩色输出就十分容易了，只需要用需要输出的字符串替换上一条命令中的 hello world，用期望颜色替换 31（代表红色）即可。

警告：以下内容仅为推荐实现，不是练习要求，有时间和兴趣的同學可以尝试。

¹ 对编译器如何向调试器提供生成的代码的信息，有兴趣可以参阅 [DWARF 规范](#)

我们推荐实现如下几个等级的输出，输出优先级依次降低：

表 3: log 等级推荐

名称	颜色	用途
ERROR	红色 (31)	表示发生严重错误，很可能或者已经导致程序崩溃
WARN	黄色 (93)	表示发生不常见情况，但是并不一定导致系统错误
INFO	蓝色 (34)	比较中庸的选项，输出比较重要的信息，比较常用
DEBUG	绿色 (32)	输出信息较多，在 debug 时使用
TRACE	灰色 (90)	最详细的输出，跟踪了每一步关键路径的执行

我们可以输出比设定输出等级以及更高输出等级的信息，如设置 `LOG = INFO`，则输出 `ERROR`、`WARN`、`INFO` 等级的信息。简单 `demo` 如下，输出等级为 `INFO`：

为了方便使用彩色输出，我们要求同学们实现彩色输出的宏或者函数，用以代替 `print` 完成输出内核信息的功能，它们有着和 `printf` 十分相似的使用格式，要求支持可变参数解析，形如：

```
// 这段代码输出了 os 内存空间布局，这到这些信息对于编写 os 十分重要

info!(".text [{:#x}, {:#x})", s_text as usize, e_text as usize);
debug!(".rodata [{:#x}, {:#x})", s_rodata as usize, e_rodata as usize);
error!(".data [{:#x}, {:#x})", s_data as usize, e_data as usize);
```

```
info("load range : [%d, %d] start = %d\n", s, e, start);
```

在以后，我们还可以在 log 信息中增加线程、CPU 等信息（只是一个推荐，不做要求），这些信息将极大的方便你的代码调试。

实验要求

- 实现分支: ch1
- 完成实验指导书中的内容并在裸机上实现 hello world 输出。
- 实现彩色输出宏 (只要求可以彩色输出, 不要求 log 等级控制, 不要求多种颜色)
- 隐形要求

可以关闭内核所有输出。从 lab2 开始要求关闭内核所有输出 (如果实现了 log 等级控制, 那么这一点自然就实现了)。

- 利用彩色输出宏输出 os 内存空间布局

输出 .text、.data、.rodata、.bss 各段位置, 输出等级为 INFO。

challenge: 支持多核, 实现多个核的 boot。

实验检查

- 实验目录要求 (Rust)

```

├── os (内核实现)
│   ├── Cargo.toml (配置文件)
│   ├── Makefile (要求 make run LOG=xxx 可以正确执行, 可以不实现对 LOG)
│   ↪ 这一属性的支持, 设置默认输出等级为 INFO
│   └── src (所有内核的源代码放在 os/src 目录下)
│       ├── main.rs (内核主函数)
│       └── ...
├── reports
│   ├── lab1.md/pdf
│   └── ...
└── README.md (其他必要的说明)
    ...

```

报告命名 labx.md/pdf, 统一放在 reports 目录下。每个实验新增一个报告, 为了方便修改, 检查报告是以最新分支的所有报告为准。

- 检查

```

$ cd os
$ git checkout ch1
$ make run LOG=INFO

```

可以正确执行 (可以不支持 LOG 参数, 只有要彩色输出就好), 可以看到正确的内存布局输出, 根据实现不同数值可能有差异, 但应该位于 linker.ld 中指示 BASE_ADDRESS 后一段内存, 输出之后关机。

tips

- 对于 Rust, 可以使用 crate `log` , 推荐参考 [rCore](#)
- 对于 C, 可以实现不同的函数 (注意不推荐多层可变参数解析, 有时会出现不稳定情况), 也可以参考 `linux printf` 使用宏实现代码重用。
- 两种语言都可以使用 `extern` 关键字获得在其他文件中定义的符号。

问答作业

1. 请学习 `gdb` 调试工具的使用 (这对后续调试很重要), 并通过 `gdb` 简单跟踪从机器加电到跳转到 `0x80200000` 的简单过程。只需要描述重要的跳转即可, 只需要描述在 `qemu` 上的情况。
2. tips:
 - 事实上进入 `rustsbi` 之后就不需要使用 `gdb` 调试了。可以直接阅读代码。[rustsbi 起始代码](#)。
 - 可以使用示例代码 `Makefile` 中的 `make debug` 指令。
 - **一些可能用到的 `gdb` 指令:**
 - `x/10i 0x80000000`: 显示 `0x80000000` 处的 10 条汇编指令。
 - `x/10i $pc`: 显示即将执行的 10 条汇编指令。
 - `x/10xw 0x80000000`: 显示 `0x80000000` 处的 10 条数据, 格式为 16 进制 32bit。
 - `info register`: 显示当前所有寄存器信息。
 - `info r t0`: 显示 `t0` 寄存器的值。
 - `break funcname`: 在目标函数第一条指令处设置断点。
 - `break *0x80200000`: 在 `0x80200000` 处设置断点。
 - `continue`: 执行直到碰到断点。
 - `si`: 单步执行一条汇编指令。

实验练习的提交报告要求

- 简单总结本次实验你编程的内容。(控制在 5 行以内, 不要贴代码)
- 由于彩色输出不好自动测试, 请附正确运行后的截图。
- 完成问答问题。
- (optional) 你对本次实验设计及难度/工作量的看法, 以及有哪些需要改进的地方, 欢迎畅所欲言。

2.9 练习参考答案**2.9.1 课后练习****编程题**

1. * 实现一个 `linux` 应用程序 A, 显示当前目录下的文件名。(用 C 或 Rust 编程)

参考实现:

```
#include <dirent.h>
#include <stdio.h>

int main() {
    DIR *dir = opendir(".");
    struct dirent *entry;

    while ((entry = readdir(dir))) {
        printf("%s\n", entry->d_name);
    }

    return 0;
}
```

可能的输出:

```
$ ./ls
.
..
.git
.dockerignore
Dockerfile
LICENSE
Makefile
[...]
```

2. *** 实现一个 linux 应用程序 B, 能打印出调用栈链信息。(用 C 或 Rust 编程)

以使用 GCC 编译的 C 语言程序为例, 使用编译参数 `-fno-omit-frame-pointer` 的情况下, 会保存栈帧指针 `fp`。

`fp` 指向的栈位置的负偏移量处保存了两个值:

- $-8(fp)$ 是保存的 `ra`
- $-16(fp)$ 是保存的上一个 `fp`

因此我们可以像链表一样, 从当前的 `fp` 寄存器的值开始, 每次找到上一个 `fp`, 逐帧恢复我们的调用栈:

```
#include <inttypes.h>
#include <stdint.h>
#include <stdio.h>

// Compile with -fno-omit-frame-pointer
void print_stack_trace_fp_chain() {
    printf("== Stack trace from fp chain ==\n");

    uintptr_t *fp;
    asm("mv %0, fp" : "=r"(fp) : : );

    // When should this stop?
    while (fp) {
        printf("Return address: 0x%016" PRIxPTR "\n", fp[-1]);
        printf("Old stack pointer: 0x%016" PRIxPTR "\n", fp[-2]);
        printf("\n");

        fp = (uintptr_t *) fp[-2];
    }
}
```

(下页继续)

(续上页)

```

    }
    printf("==== End ====\n\n");
}

```

但是这里会遇到一个问题，因为我们的标准库并没有保存栈帧指针，所以找到调用栈到标准的库时候会打破我们对栈帧格式的假设，出现异常。

我们也可以不做关于栈帧保存方式的假设，而是明确让编译器告诉我们每个指令处的调用栈如何恢复。在编译的时候加入 `-funwind-tables` 会开启这个功能，将调用栈恢复的信息存入可执行文件中。

有一个叫做 `libunwind` 的库可以帮我们读取这些信息生成调用栈信息，而且它可以正确发现某些栈帧不知道怎么恢复，避免异常退出。

正确安装 `libunwind` 之后，我们也可以用这样的方式生成调用栈信息：

```

#include <inttypes.h>
#include <stdint.h>
#include <stdio.h>

#define UNW_LOCAL_ONLY
#include <libunwind.h>

// Compile with -funwind-tables -lunwind
void print_stack_trace_libunwind() {
    printf("==== Stack trace from libunwind ====\n");

    unw_cursor_t cursor; unw_context_t uc;
    unw_word_t pc, sp;

    unw_getcontext(&uc);
    unw_init_local(&cursor, &uc);

    while (unw_step(&cursor) > 0) {
        unw_get_reg(&cursor, UNW_REG_IP, &pc);
        unw_get_reg(&cursor, UNW_REG_SP, &sp);

        printf("Program counter: 0x%016" PRIxPTR "\n", (uintptr_t) pc);
        printf("Stack pointer: 0x%016" PRIxPTR "\n", (uintptr_t) sp);
        printf("\n");
    }
    printf("==== End ====\n\n");
}

```

3. ** 实现一个基于 rcore/ucore tutorial 的应用程序 C，用 sleep 系统调用睡眠 5 秒 (in rcore/ucore tutorial v3: Branch ch1)

注：尝试用 GDB 等调试工具和输出字符串的等方式来调试上述程序，能设置断点，单步执行和显示变量，理解汇编代码和源程序之间的对应关系。

问答题

1. * 应用程序在执行过程中，会占用哪些计算机资源？

占用 CPU 计算资源 (CPU 流水线, 缓存等), 内存 (内存不够还会占用外存) 等

2. * 请用相关工具软件分析并给出应用程序 A 的代码段/数据段/堆/栈的地址空间范围。

简便起见，我们静态编译该程序生成可执行文件。使用 `readelf` 工具查看地址空间：

数据段 (.data) 和代码段 (.text) 的起止地址可以从输出信息中看出。

应用程序的堆栈是由内核为其动态分配的，需要在运行时查看。将 A 程序置于后台执行，通过查看 `/proc/[pid]/maps` 得到堆栈空间的分布：

3. * 请简要说明应用程序与操作系统的异同之处。

这个问题相信大家完成了实验的学习后一定会有更深的理解。

4. ** 请基于 QEMU 模拟 RISC-V 的执行过程和 QEMU 源代码，说明 RISC-V 硬件加电后的几条指令在哪里？完成了哪些功能？

在 QEMU 源码¹ 中可以找到“上电”的时候刚执行的几条指令，如下：

```
uint32_t reset_vec[10] = {
    0x00000297,           /* 1: auipc t0, %pcrel_hi(fw_dyn) */
    0x02828613,           /*      addi a2, t0, %pcrel_lo(1b) */
    0xf1402573,           /*      csrr a0, mhartid */
#if defined(TARGET_RISCV32)
    0x0202a583,           /*      lw    a1, 32(t0) */
    0x0182a283,           /*      lw    t0, 24(t0) */
#elif defined(TARGET_RISCV64)
    0x0202b583,           /*      ld    a1, 32(t0) */
    0x0182b283,           /*      ld    t0, 24(t0) */
#endif
    0x00028067,           /*      jr    t0 */
    start_addr,           /*      start: .dword */
    start_addr_hi32,       /*      fdt_load_addr: .dword */
    fdt_load_addr,         /*      fw_dyn: */
    0x00000000,
};
```

完成的工作是：

- 读取当前的 Hart ID CSR `mhartid` 写入寄存器 `a0`
- (我们还没有用到：将 FDT (Flatten device tree) 在物理内存中的地址写入 `a1`)
- 跳转到 `start_addr`，在我们实验中是 RustSBI 的地址

5. * RISC-V 中的 SBI 的含义和功能是啥？

详情见 [SBI 官方文档](#)

6. ** 为了让应用程序能在计算机上执行，操作系统与编译器之间需要达成哪些协议？

编译器依赖操作系统提供的程序库，操作系统执行应用程序需要编译器提供段位置、符号表、依赖库等信息。[ELF](#) 就是比较常见的一种文件格式。

7. ** 请简要说明从 QEMU 模拟的 RISC-V 计算机加电开始运行到执行应用程序的第一条指令这个阶段的执行过程。

¹ <https://github.com/qemu/qemu/blob/0ebf76aae58324b8f7bf6af798696687f5f4c2a9/hw/riscv/boot.c#L300>

接第 5 题, 跳转到 RustSBI 后, SBI 会对部分硬件例如串口等进行初始化, 然后通过 mret 跳转到 payload 也就是 kernel 所在的起始地址。kernel 进行一系列的初始化后 (内存管理, 虚存管理, 线程 (进程) 初始化等), 通过 sret 跳转到应用程序的第一条指令开始执行。

8. ** 为何应用程序编写应用时不需要建立栈空间和指定地址空间?

应用程序对内存的访问需要通过 MMU 的地址翻译完成, 应用程序运行时看到的地址和实际位于内存中的地址是不同的, 栈空间和地址空间需要内核进行管理和分配。应用程序的栈指针在 trap return 过程中初始化。此外, 应用程序可能需要动态加载某些库的内容, 也需要内核完成映射。

9. *** 现代的很多编译器生成的代码, 默认情况下不再严格保存/恢复栈帧指针。在这个情况下, 我们只要编译器提供足够的信息, 也可以完成对调用栈的恢复。(题目剩余部分省略)

- 首先, 我们当前的 pc 在 flip 函数的开头, 这是我们正在运行的函数。返回给调用者处的地址在 ra 寄存器里, 是 0x10742。因为我们还没有开始操作栈指针, 所以调用处的 sp 与我们相同, 都是 0x40007f1310。
- 0x10742 在 flap 函数内。根据 flap 函数的开头可知, 这个函数的栈帧大小是 16 个字节, 所以调用者处的栈指针应该是 $sp + 16 = 0x40007f1320$ 。调用 flap 的调用者返回地址保存在栈上 8 (sp), 可以读出来是 0x10750, 还在 flap 函数内。
- 依次类推, 只要能理解已知地址对应的函数代码, 就可以完成恢复操作。

2.9.2 实验练习

问答作业

1. 请学习 gdb 调试工具的使用 (这对后续调试很重要), 并通过 gdb 简单跟踪从机器加电到跳转到 0x80200000 的简单过程。只需要描述重要的跳转即可, 只需要描述在 qemu 上的情况。

第二章：批处理系统

3.1 引言

3.1.1 本章导读

保障系统安全和多应用支持是操作系统的两个核心目标，本章从这两个目标出发，思考如何设计应用程序，并进一步展现了操作系统的一系列新功能：

- 构造包含操作系统内核和多个应用程序的单一执行程序
- 通过批处理支持多个程序的自动加载和运行
- 操作系统利用硬件特权级机制，实现对操作系统自身的保护
- 实现特权级的穿越
- 支持跨特权级的系统调用功能

上一章，我们在 RISC-V 64 裸机平台上成功运行起来了 `Hello, world!`。看起来这个过程非常顺利，只需要一条命令就能全部完成。但实际上，在那个计算机刚刚诞生的年代，很多事情并不像我们想象的那么简单。当时，程序被记录在打孔的卡片上，使用汇编语言甚至机器语言来编写。而稀缺且昂贵的计算机由专业的管理员负责操作，就和我们在上一章所做的事情一样，他们手动将卡片输入计算机，等待程序运行结束或者终止程序的运行。最后，他们从计算机的输出端——也就是打印机中取出程序的输出并交给正在休息室等待的程序提交者。

实际上，这样做是一种对于珍贵的计算资源的浪费。因为当时（二十世纪 60 年代）的大型计算机和今天的个人计算机不同，它的体积极其庞大，能够占满一整个空调房间，像巨大的史前生物。当时用户（程序员）将程序输入到穿孔卡片上。用户将一批这些编程的卡片交给系统操作员，然后系统操作员将它们输入计算机。系统管理员在房间的各个地方跑来跑去、或是等待打印机的输出的这些时间段，计算机都并没有在工作。于是，人们希望计算机能够不间断的工作且专注于计算任务本身。**批处理系统** (Batch System) 应运而生，它可用来管理无需或仅需少量用户交互即可运行的程序，在资源允许的情况下它可以自动安排程序的执行，这被称为“批处理作业”，这个名词源自二十世纪 60 年代的大型机时代。批处理系统的核心思想是：将多个程序打包到一起输入计算机。而当一个程序运行结束后，计算机会自动加载下一个程序到内存并开始执行。当软件有了代替操作员的管理和操作能力后，便开始形成真正意义上的操作系统了。

注解：来自汽车生产线灵感的 GM-NAA I/O System 批处理操作系统

操作系统历史上最伟大的想象力飞跃之一是计算机可能通过软件来安排自己的工作负荷的想法，这体现在早期的批处理操作系统的设计与实现中。

在 2006 年计算机历史博物馆对 Robert L. Patrick 的一次采访中，Patrick 回顾了在 1954-1956 年前后他在通用汽车（General Motors，简称 GM）公司设计实现的 GM-NAA I/O 操作系统的有趣开发历史。当时（1954 年），通用汽车公司购置的 IBM 701 大型计算机使用效率极低，大约 2/3 的时间处于浪费的闲置状态，而浪费的计算机时间的成本是每月近 15 万美元，这给公司带来了巨大的经济开销。计算机的用途是程序开发和执行，而开发程序、编译程序、测试程序、运行程序、操作计算机运行等事务大多都由程序员来完成，编写好的程序源码会被程序员手工按顺序放到磁带（磁带只能串行顺序读写代码和数据）上，再串行加载到计算机上被编译器编译成可执行程序，再加载可执行程序运行，最后打印输出执行结果。当时的程序多是机器码程序或汇编程序等，也有处于试验阶段的早期 FORTRAN 语言编写的程序，很容易出错。如果当前正在执行的程序测试运行崩溃或提前终止，其他程序只能等待，整个机器就会闲置。程序员的大量时间是等待机器能运行到他提交的程序。

Patrick 采用了提高并行处理流程的汽车生产线设计中的一些分析技术来设计面向下一代 704 计算机的操作系统（当时的名字还是 Monitor，监控器），而这些想法起源于 Henry Laurence Gantt，他在 1910 年发明了甘特图，这是一种条状图，可显示项目、进度以及其他与时间相关的系统进展的内在关系随着时间进展的情况。然后 Patrick 和来自北美航空公司的 Owen Mock 合作，带领开发团队一起设计了 GM-NAA I/O System（General Motors - North America Aviation Input-Output System）操作系统。

GM-NAA I/O System 操作系统完成对计算机的管理与控制，形成了标准化的输入和输出程序以及作业控制语言。以前由程序员承担的计算机操作工作，如把程序导入磁带，加载程序，转储程序出错信息并继续执行下一程序等各种任务，现在都由操作系统来按相互依赖关系分阶段进行编排，并自动完成。在原有硬件和程序员工资的情况下，计算机的使用效率提高了 5 倍以上，程序员没有那么多空闲的时间用来聊天了。

应用程序总是难免会出现错误，如果一个程序的执行错误导致其它程序或者整个计算机系统都无法运行就太糟糕了。人们希望一个应用程序的错误不要影响到其它应用程序、操作系统和整个计算机系统。这就需要操作系统能够终止出错的应用程序，转而运行下一个应用程序。这种保护计算机系统不受有意或无意出错的程序破坏的机制被称为 **特权级**（Privilege）机制，它让应用程序运行在用户态，而操作系统运行在内核态，且实现用户态和内核态的隔离，这需要计算机软件和硬件的共同努力。

注解：想法超前且用力过猛的 MULTICS 操作系统

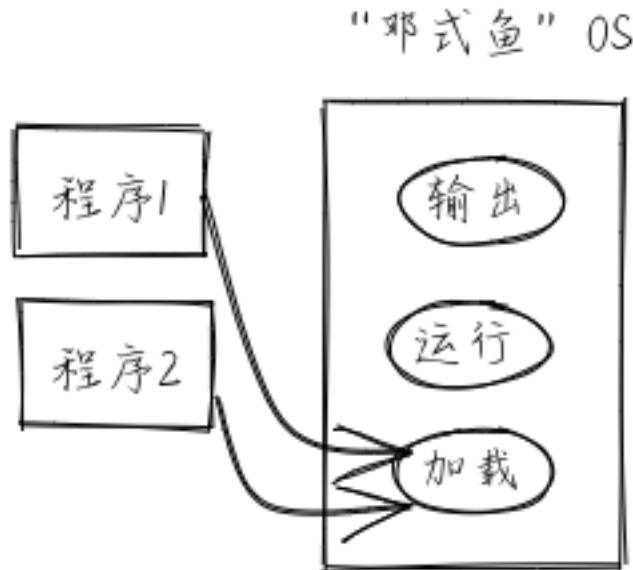
1961-1963 年，MIT 的 Fernando Corbató 教授带领的研究小组在 IBM 7090 上演示和进一步实现了 CTSS（Compatible Time-Sharing System）操作系统，当时 CTSS 被认为是一个提供给广泛和不同用户群的大规模分时系统，受到广泛好评。于是他们在 1965 年计划设计新一代 MULTICS 操作系统，其总体设计目标是创建一个能够全面满足大型计算机服务设施的几乎所有当前和未来要求的计算系统，这是一个更为野心勃勃的目标。在 DARPA 的资助下，MIT 联合了强大的 GE 公司（负责提供 GE 645 大型机）和贝尔实验室（负责提供高水平的软件工程师），开始设计实现 MULTICS 操作系统。MULTICS 操作系统与 GE 645 硬件在同一时间段进行软硬件协同设计，MULTICS 操作系统的前期设计是在 GE 645 模拟器（运行在 GE 635 计算机上）上进行的。

MULTICS 操作系统是一开始就为安全系统而设计的通用操作系统，多重保护环（Multiple rings of protection，也称分级保护域）是 MULTICS 操作系统引入的最具革命性的概念之一。尽管 GE 645 计算机有 8 级硬件支持的保护环，它仍不足以满足 MULTICS 的安全需求，因此 MULTICS 通过软件方式支持多达 64 级保护环，这样在不同安全级别的保护环，可以运行不同的管理程序或应用程序。

这种设计思想具有典型的第二系统效应问题，即在一个简单的第一个系统（这里指 CTSS）成功之后，又有一个更复杂的第二个系统（这里指 MULTICS）的诱惑，但其实在当时情况下难以顺利完成。8 级硬件多重保护环设计仅存在于 GE 645 中，这使得 MULTICS 操作系统不可移植，无法向更便宜、更开放的硬件系统移植。而且后续成功的计算机系统表明，多级保护环不是必须的，一个简单两级保护环（用户态和内核态）再加上分页机制，就足以实现绝大多数的安全隔离需求了。

本章主要是设计和实现建立支持批处理系统的泥盆纪“邓氏鱼”¹ 操作系统，从而对可支持运行一批应用程序的执行环境有一个全面和深入的理解。

本章我们的目标让泥盆纪“邓氏鱼”操作系统能够感知多个应用程序的存在，并一个接一个地运行这些应用程序，当一个应用程序执行完毕后，会启动下一个应用程序，直到所有的应用程序都执行完毕。



3.1.2 实践体验

本章我们的批处理系统将连续运行三个应用程序，放在 `user/src/bin` 目录下。

获取本章代码：

```
$ git clone https://github.com/rcore-os/rCore-Tutorial-v3.git
$ cd rCore-Tutorial-v3
$ git checkout ch2
```

在 `qemu` 模拟器上运行本章代码：

```
$ cd os
$ make run
```

如果顺利的话，我们可以看到批处理系统自动加载并运行所有的程序并且正确在程序出错的情况下保护了自身：

```
[RustSBI output]
[kernel] Hello, world!
[kernel] num_app = 5
[kernel] app_0 [0x8020a038, 0x8020af90)
[kernel] app_1 [0x8020af90, 0x8020bf80)
[kernel] app_2 [0x8020bf80, 0x8020d108)
[kernel] app_3 [0x8020d108, 0x8020e0e0)
[kernel] app_4 [0x8020e0e0, 0x8020f0b8)
```

(下页继续)

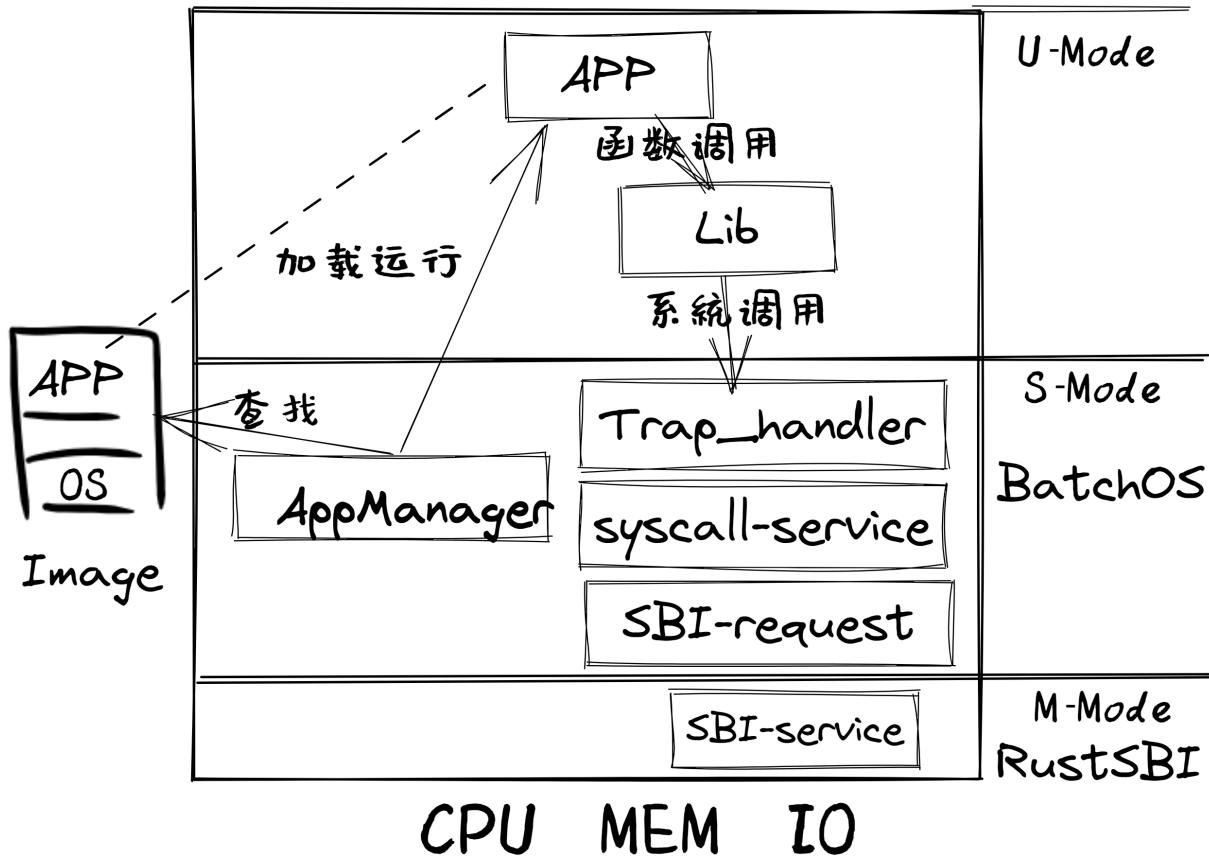
¹ 邓氏鱼是一种晚泥盆纪（距今约 3.82 亿至 3.59 亿年前）的盾皮鱼，其中最大种类体长可达 8.79 米，重量可达 4 吨，是当时最大的海洋掠食者，但巨大而沉重的身躯极大地影响了它的运动速度和灵敏度。

(续上页)

```
[kernel] Loading app_0
Hello, world!
[kernel] Application exited with code 0
[kernel] Loading app_1
Into Test store_fault, we will insert an invalid store operation...
Kernel should kill this application!
[kernel] PageFault in application, kernel killed it.
[kernel] Loading app_2
3^10000=5079 (MOD 10007)
3^20000=8202 (MOD 10007)
3^30000=8824 (MOD 10007)
3^40000=5750 (MOD 10007)
3^50000=3824 (MOD 10007)
3^60000=8516 (MOD 10007)
3^70000=2510 (MOD 10007)
3^80000=9379 (MOD 10007)
3^90000=2621 (MOD 10007)
3^100000=2749 (MOD 10007)
Test power OK!
[kernel] Application exited with code 0
[kernel] Loading app_3
Try to execute privileged instruction in U Mode
Kernel should kill this application!
[kernel] IllegalInstruction in application, kernel killed it.
[kernel] Loading app_4
Try to access privileged CSR in U Mode
Kernel should kill this application!
[kernel] IllegalInstruction in application, kernel killed it.
[kernel] Panicked at src/batch.rs:58 All applications completed!
```

3.1.3 本章代码树

邓式鱼 BatchOS 操作系统的总体结构如下图所示：



通过上图，大致可以看出 Qemu 把包含多个 app 的列表和 BatchOS 的 image 镜像加载到内存中，RustSBI (bootloader) 完成基本的硬件初始化后，跳转到邓式鱼 BatchOS 起始位置，邓式鱼 BatchOS 首先进行正常运行前的初始化工作，即建立栈空间和清零 bss 段，然后通过 AppManager 内核模块从 app 列表中依次加载各个 app 到指定的内存中在用户态执行。app 在执行过程中，会通过系统调用的方式得到邓式鱼 BatchOS 提供的 OS 服务，如输出字符串等。

位于 ch2 分支上的邓式鱼 BatchOS 操作系统的源代码如下所示：

```
./os/src
Rust      13 Files  372 Lines
Assembly   2 Files   58 Lines

├── bootloader
│   └── rustsbi-qemu.bin
├── LICENSE
└── os
    ├── build.rs (新增：生成 link_app.S 将应用作为一个数据段链接到内核)
    ├── Cargo.toml
    ├── Makefile (修改：构建内核之前先构建应用)
    └── src
        ├── batch.rs (新增：实现了一个简单的批处理系统)
        ├── console.rs
        ├── entry.asm
        ├── lang_items.rs
        ├── link_app.S (构建产物，由 os/build.rs 输出)
        └── linker-qemu.ld
```

(下页继续)

(续上页)

```

|   main.rs(修改：主函数中需要初始化 Trap 处理并加载和执行应用)
|   sbi.rs
|   sync(新增：同步子模块 sync，目前唯一功能是提供 UPSafeCell)
|   |   mod.rs
|   |   up.rs(包含 UPSafeCell，它可以帮助我们以更 Rust 的方式使用全局变量)
|   |   syscall(新增：系统调用子模块 syscall)
|   |   |   fs.rs(包含文件 I/O 相关的 syscall)
|   |   |   mod.rs(提供 syscall 方法根据 syscall ID 进行分发处理)
|   |   |   process.rs(包含任务处理相关的 syscall)
|   |   trap(新增：Trap 相关子模块 trap)
|   |   |   context.rs(包含 Trap 上下文 TrapContext)
|   |   |   mod.rs(包含 Trap 处理入口 trap_handler)
|   |   |   trap.S(包含 Trap 上下文保存与恢复的汇编代码)
|   README.md
|   rust-toolchain
└── user(新增：应用测例保存在 user 目录下)
    ├── Cargo.toml
    ├── Makefile
    └── src
        ├── bin(基于用户库 user_lib 开发的应用，每个应用放在一个源文件中)
        |   ├── 00hello_world.rs
        |   ├── 01store_fault.rs
        |   ├── 02power.rs
        |   ├── 03priv_inst.rs
        |   └── 04priv_csr.rs
        ├── console.rs
        ├── lang_items.rs
        ├── lib.rs(用户库 user_lib)
        ├── linker.ld(应用的链接脚本)
        └── syscall.rs(包含 syscall 方法生成实际用于系统调用的汇编指令，  
各个具体的 syscall 都是通过 syscall 来实现的)

```

3.1.4 本章代码导读

相比于上一章的两个简单操作系统，本章的操作系统有两个最大的不同之处，一个是操作系统自身运行在内核态，且支持应用程序在用户态运行，且能完成应用程序发出的系统调用；另一个是能够一个接一个地自动运行不同的应用程序。所以，我们需要对操作系统和应用程序进行修改，也需要对应用程序的编译生成过程进行修改。

首先改进应用程序，让它能够在用户态执行，并能发出系统调用。具体而言，编写多个应用小程序，修改编译应用所需的 `linker.ld` 文件来 [调整程序的内存布局](#)，让操作系统能够把应用加载到指定内存地址，然后顺利启动并运行应用程序。

在应用程序的运行过程中，操作系统要支持应用程序的输出功能，并还能支持应用程序退出。这需要实现跨特权级的系统调用接口，以及 `sys_write` 和 `sys_exit` 等具体的系统调用功能。在具体设计实现上，涉及到内联汇编的编写，以及应用与操作系统内核之间系统调用的参数传递的约定。为了让应用程序在还没实现邓氏鱼操作系统之前就能在 Linux for RISC-V 64 上进行运行测试，我们采用了 Linux on RISC-V64 的系统调用参数约定。具体实现可参看 [系统调用](#) 小节中的内容。这样写完应用小例子后，就可以通过 `qemu-riscv64` 模拟器进行测试了。

写完应用程序后，还需实现支持多个应用程序轮流启动运行的操作系统。这里首先能把本来相对松散的应用程序执行代码和操作系统执行代码连接在一起，便于 `qemu-system-riscv64` 模拟器一次性地加载二者到内存中，并让操作系统能够找到应用程序的位置。为把二者连在一起，需要对生成的应用程序进行改造，首先是把应用程序执行文件从 ELF 执行文件格式变成 Binary 格式（通过 `rust-objcopy` 可以轻松完成）；然后这些 Binary 格式的文件通过编译器辅助脚本 `os/build.rs` 转变成 `os/src/link_app.S` 这个汇编文

件的一部分，并生成各个 Binary 应用的辅助信息，便于操作系统能够找到应用的位置。编译器会把操作系统的源码和 `os/src/link_app.S` 合在一起，编译出操作系统 +Binary 应用的 ELF 执行文件，并进一步转变成 Binary 格式。

为了定位 Binary 应用在被加载后的内存位置，操作系统本身需要完成对 Binary 应用的位置查找，找到后（通过 `os/src/link_app.S` 中的变量和标号信息完成），会把 Binary 应用从加载位置拷贝到 `user/src/linker.ld` 指定的物理内存位置（OS 的加载应用功能）。在一个应用执行完毕后，操作系统还能加载另外一个应用，这主要是通过 `AppManagerInner` 数据结构和对应的函数 `load_app` 和 `run_next_app` 等来完成对应用的一系列管理功能。这主要在实现批处理操作系统 小节中讲解。

为了让 Binary 应用能够启动和运行，操作系统还需给 Binary 应用分配好对应执行环境所需一系列的资源。这主要包括设置好用户栈和内核栈（在用户态的应用程序与在内核态的操作系统内核需要有各自的栈，避免应用程序破坏内核的执行），实现 Trap 上下文的保存与恢复（让应用能够在发出系统调用到内核态后，还能回到用户态继续执行），完成 Trap 分发与处理等工作。由于系统调用和中断处理等内核代码实现涉及用户态与内核态之间的特权级切换细节的汇编代码，与硬件细节联系紧密，所以这部分内容是本章中理解比较困难的地方。如果要了解清楚，需要对涉及到的 RISC-V CSR 寄存器的功能有明确认识。这就需要查看 [RISC-V 手册](#) 的第十章或更加详细的 RISC-V 的特权级规范文档了。有了上面的实现后，就剩下最后一步，实现 [执行应用程序](#) 的操作系统功能，其主要实现在 `run_next_app` 内核函数中。完成所有这些功能的实现，“邓式鱼”^{Page 10, 1} 操作系统就可以正常运行，并能管理多个应用按批处理方式在用户态一个接一个地执行了。

3.2 特权级机制

3.2.1 本节导读

为了保护我们的批处理操作系统不受到出错应用程序的影响并全程稳定工作，单凭软件实现是很难做到的，而是需要 CPU 提供一种特权级隔离机制，使 CPU 在执行应用程序和操作系统内核的指令时处于不同的特权级。本节主要介绍了特权级机制的软硬件设计思路，以及 RISC-V 的特权级架构，包括特权指令的描述。

3.2.2 特权级的软硬件协同设计

实现特权级机制的根本原因是应用程序运行的安全性不可充分信任。在上一章里，操作系统以库的形式和应用紧密连接在一起，构成一个整体来执行。随着应用需求的增加，操作系统的体积也越来越大；同时应用自身也会越来越复杂。由于操作系统会被频繁访问，来给多个应用提供服务，所以它可能的错误会比较快地被发现。但应用自身的错误可能就不会很快发现。由于二者通过编译器形成一个单一执行程序来执行，导致即使是应用程序本身的问题，也会让操作系统受到连累，从而可能导致整个计算机系统都不可用了。

所以，计算机科学家和工程师就想到一个方法，让相对安全可靠的操作系统运行在一个硬件保护的安全执行环境中，不受到应用程序的破坏；而让应用程序运行在另外一个无法破坏操作系统的受限执行环境中。

为确保操作系统的安全，对应用程序而言，需要限制的主要有两个方面：

- 应用程序不能访问任意的地址空间（这个在第四章会进一步讲解，本章不会涉及）
- 应用程序不能执行某些可能破坏计算机系统的指令（本章的重点）

假设有了这样的限制，我们还需要确保应用程序能够得到操作系统的服务，即应用程序和操作系统还需要有交互的手段。使得低特权级软件只能做高特权级软件允许它做的，且超出低特权级软件能力的功能必须寻求高特权级软件的帮助。这样，高特权级软件（操作系统）就成为低特权级软件（一般应用）的软件执行环境的重要组成部分。

为了实现这样的特权级机制，需要进行软硬件协同设计。一个比较简洁的方法就是，处理器设置两个不同安全等级的执行环境：用户态特权级的执行环境和内核态特权级的执行环境。且明确指出可能破坏计算机系统的内核态特权级指令子集，规定内核态特权级指令子集中的指令只能在内核态特权级的执行环境中执行。处理器在执行指令前会进行特权级安全检查，如果在用户态执行环境中执行这些内核态特权级指令，会产生异常。

为了让应用程序获得操作系统的函数服务，采用传统的函数调用方式（即通常的 `call` 和 `ret` 指令或指令组合）将会直接绕过硬件的特权级保护检查。为了解决这个问题，RISC-V 提供了新的机器指令：执行环境调用指令（Execution Environment Call，简称 `ecall`）和一类执行环境返回（Execution Environment Return，简称 `eret`）指令。其中：

- `ecall` 具有用户态到内核态的执行环境切换能力的函数调用指令；
- `sret`：具有内核态到用户态的执行环境切换能力的函数返回指令。

注解：`sret` 与 `eret` 的联系与区别

`eret` 代表一类执行环境返回指令，而 `sret` 特指从 Supervisor 模式的执行环境（即 OS 内核）返回的那条指令，也是本书中主要用到的指令。除了 `sret` 之外，`mret` 也属于执行环境返回指令，当从 Machine 模式的执行环境返回时使用，RustSBI 会用到这条指令。

硬件具有了这样的机制后，还需要操作系统的配合才能最终完成对操作系统自身的保护。首先，操作系统需要提供相应的功能代码，能在执行 `sret` 前准备和恢复用户态执行应用程序的上下文。其次，在应用程序调用 `ecall` 指令后，能够检查应用程序的系统调用参数，确保参数不会破坏操作系统。

注解：一般来说，`ecall` 这条指令和 `eret` 这类指令分别可以用来让 CPU 从当前特权级切换到比当前高一级的特权级和切换到不高于当前的特权级，因此上面提到的两条指令的功能仅是其中一种用法。在本书中，大多数情况我们只需考虑这种用法即可。

读者可能会好奇一共有多少种不同的特权级，在不同的指令集体体系结构中特权级的数量也是不同的。x86 和 RISC-V 设计了多达 4 种特权级，而对于一般的操作系统而言，其实只要两种特权级就够了。

3.2.3 RISC-V 特权级架构

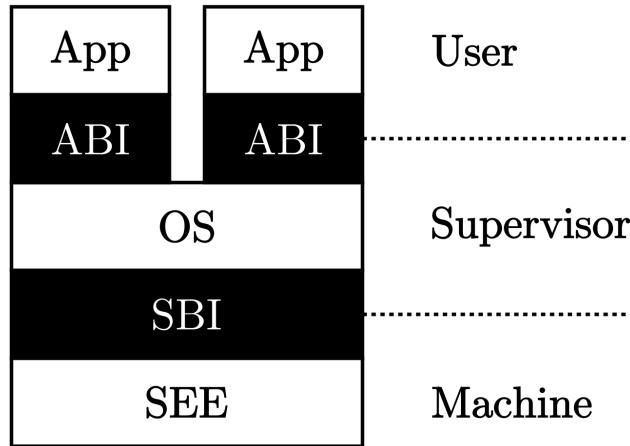
RISC-V 架构中一共定义了 4 种特权级：

表 1: RISC-V 特权级

级别	编码	名称
0	00	用户/应用模式 (U, User/Application)
1	01	监督模式 (S, Supervisor)
2	10	虚拟监督模式 (H, Hypervisor)
3	11	机器模式 (M, Machine)

其中，级别的数值越大，特权级越高，掌控硬件的能力越强。从表中可以看出，M 模式处在最高的特权级，而 U 模式处于最低的特权级。在 CPU 硬件层面，除了 M 模式必须存在外，其它模式可以不存在。

之前我们给出过支持应用程序运行的一套执行环境栈，现在我们站在特权级架构的角度去重新看待它：



和之前一样，白色块表示一层执行环境，黑色块表示相邻两层执行环境之间的接口。这张图片给出了能够支持运行 Unix 这类复杂系统的软件栈。其中操作系统内核代码运行在 S 模式上；应用程序运行在 U 模式上。运行在 M 模式上的软件被称为 **监督模式执行环境 (SEE, Supervisor Execution Environment)**，如在操作系统运行前负责加载操作系统的 Bootloader - RustSBI。站在运行在 S 模式上的软件视角来看，它的下面也需要一层执行环境支撑，因此被命名为 SEE，它需要在相比 S 模式更高的特权级下运行，一般情况下 SEE 在 M 模式上运行。

注解：按需实现 RISC-V 特权级

RISC-V 架构中，只有 M 模式是必须实现的，剩下的特权级则可以根据跑在 CPU 上应用的实际需求进行调整：

- 简单的嵌入式应用只需要实现 M 模式；
- 带有一定保护能力的嵌入式系统需要实现 M/U 模式；
- 复杂的多任务系统则需要实现 M/S/U 模式。
- 到目前为止，(Hypervisor, H) 模式的特权规范还没完全制定好，所以本书不会涉及。

之前我们提到过，执行环境的功能之一是在执行它支持的上层软件之前进行一些初始化工作。我们之前提到的引导加载程序会在加电后对整个系统进行初始化，它实际上是 SEE 功能的一部分，也就是说在 RISC-V 架构上的引导加载程序一般运行在 M 模式上。此外，编程语言相关的标准库也会在执行应用程序员编写的应用程序之前进行一些初始化工作。但在这张图中我们并没有将应用程序的执行环境详细展开，而是统一归类到 U 模式软件，也就是应用程序中。

回顾第一章，当时只是实现了简单的支持单个裸机应用的库级别的“三叶虫”操作系统，它和应用程序全程运行在 S 模式下，应用程序很容易破坏没有任何保护的执行环境—操作系统。而在后续的章节中，我们会涉及到 RISC-V 的 M/S/U 三种特权级：其中应用程序和用户态支持库运行在 U 模式的最低特权级；操作系统内核运行在 S 模式特权级（在本章表现为一个简单的批处理系统），形成支撑应用程序和用户态支持库的执行环境；而第一章提到的预编译的 bootloader - RustSBI 实际上是运行在更底层的 M 模式特权级下的软件，是操作系统内核的执行环境。整个软件系统就由这三层运行在不同特权级下的不同软件组成。

在特权级相关机制方面，本书正文中我们重点关心 RISC-V 的 S/U 特权级，M 特权级的机制细节则是作为可

选内容在附录 C: 深入机器模式: *RustSBI* 中讲解, 有兴趣的同学可以参考。

执行环境的另一种功能是对上层软件的执行进行监控管理。监控管理可以理解为, 当上层软件执行的时候出现了一些异常或特殊情况, 导致需要用到执行环境中提供的功能, 因此需要暂停上层软件的执行, 转而运行执行环境的代码。由于上层软件和执行环境被设计为运行在不同的特权级, 这个过程也往往 (而不一定) 伴随着 CPU 的 特权级切换。当执行环境的代码运行结束后, 我们需要回到上层软件暂停的位置继续执行。在 RISC-V 架构中, 这种与常规控制流 (顺序、循环、分支、函数调用) 不同的 异常控制流 (ECF, Exception Control Flow) 被称为 异常 (Exception), 是 RISC-V 语境下的 Trap 种类之一。

用户态应用直接触发从用户态到内核态的异常的原因总体上可以分为两种: 其一是用户态软件为获得内核态操作系统的服务功能而执行特殊指令; 其二是在执行某条指令期间产生了错误 (如执行了用户态不允许执行的指令或者其他错误) 并被 CPU 检测到。下表中我们给出了 RISC-V 特权级规范定义的会可能导致从低特权级到高特权级的各种 异常:

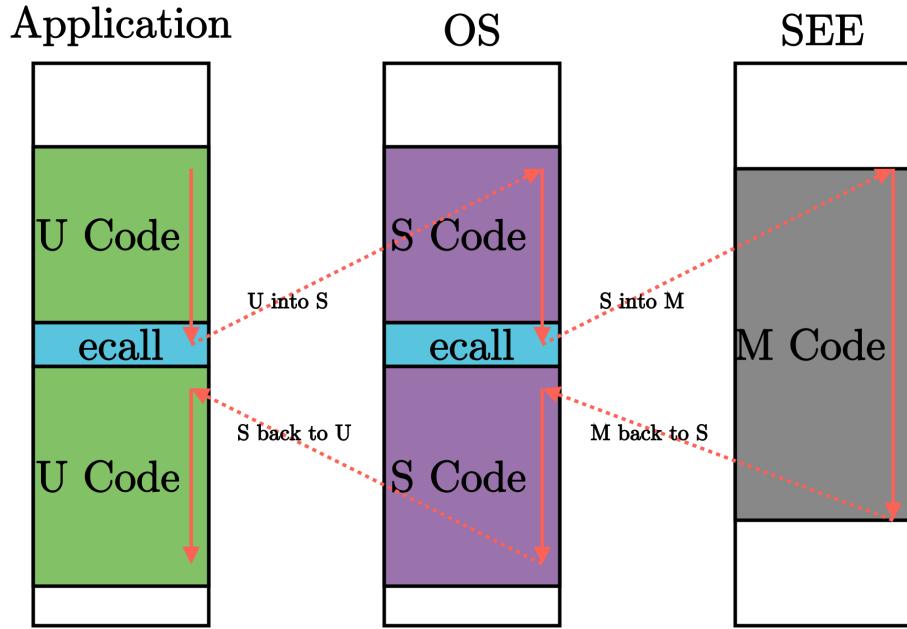
表 2: RISC-V 异常一览表

Interrupt	Exception Code	Description
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	15	Store/AMO page fault

其中 断点 (Breakpoint) 和 执行环境调用 (Environment call) 两种异常 (为了与其他非有意为之的异常区分, 会把这种有意为之的指令称为 陷入或 trap 类指令, 此处的陷入为操作系统中传统概念) 是通过在上层软件中执行一条特定的指令触发的: 执行 `ebreak` 这条指令之后就会触发断点陷入异常; 而执行 `ecall` 这条指令时候则会随着 CPU 当前所处特权级而触发不同的异常。从表中可以看出, 当 CPU 分别处于 M/S/U 三种特权级时执行 `ecall` 这条指令会触发三种异常 (分别参考上表 Exception Code 为 11/9/8 对应的行)。

在这里我们需要说明一下执行环境调用 `ecall`, 这是一种很特殊的 陷入类的指令, 上图中相邻两特权级软件之间的接口正是基于这种陷入机制实现的。M 模式软件 SEE 和 S 模式的内核之间的接口被称为 监督模式二进制接口 (Supervisor Binary Interface, SBI), 而内核和 U 模式的应用程序之间的接口被称为 应用程序二进制接口 (Application Binary Interface, ABI), 当然它有一个更加通俗的名字——系统调用 (syscall, System Call)。而之所以叫做二进制接口, 是因为它与高级编程语言的内部调用接口不同, 是机器/汇编指令级的一种接口。事实上 M/S/U 三个特权级的软件可分别由不同的编程语言实现, 即使是用同一种编程语言实现的, 其调用也并不是普通的函数调用控制流, 而是 陷入异常控制流, 在该过程中会切换 CPU 特权级。因此只有将接口下降到机器/汇编指令级才能够满足其跨高级语言的通用性和灵活性。

可以看到, 在这样的架构之下, 每层特权级的软件都只能做高特权级软件允许它做的、且不会产生什么撼动高特权级软件的事情, 一旦低特权级软件的要求超出了其能力范围, 就必须寻求高特权级软件的帮助, 否则就是一种异常行为了。因此, 在软件 (应用、操作系统等) 执行过程中我们经常能够看到特权级切换。如下图所示:



其他的异常则一般是在执行某一条指令的时候发生了某种错误（如除零、无效地址访问、无效指令等），或处理器认为处于当前特权级下执行的当前指令是高特权级指令或会访问不应该访问的高特权级的资源（可能危害系统）。碰到这些情况，就需要将控制转交给高特权级的软件（如操作系统）来处理。当错误/异常恢复后，则可重新回到低优先级软件去执行；如果不能恢复错误/异常，那高特权级软件可以杀死和清除低特权级软件，避免破坏整个执行环境。

RISC-V 的特权指令

与特权级无关的一般的指令和通用寄存器 $x0 \sim x31$ 在任何特权级都可以执行。而每个特权级都对应一些特殊指令和 **控制状态寄存器 (CSR, Control and Status Register)**，来控制该特权级的某些行为并描述其状态。当然特权指令不仅具有读写 CSR 的指令，还有其他功能的特权指令。

如果处于低特权级状态的处理器执行了高特权级的指令，会产生非法指令错误的异常。这样，位于高特权级的执行环境能够得知低特权级的软件出现了错误，这个错误一般是不可恢复的，此时执行环境会将低特权级的软件终止。这在某种程度上体现了特权级保护机制的作用。

在 RISC-V 中，会有两类属于高特权级 S 模式的特权指令：

- 指令本身属于高特权级的指令，如 `sret` 指令（表示从 S 模式返回到 U 模式）。
- 指令访问了 S 模式特权级下才能访问的寄存器或内存，如表示 S 模式系统状态的 **控制状态寄存器 `sstatus`** 等。

表 3: RISC-V S 模式特权指令

指令	含义
<code>sret</code>	从 S 模式返回 U 模式：在 U 模式下执行会产生非法指令异常
<code>wfi</code>	处理器在空闲时进入低功耗状态等待中断：在 U 模式下执行会产生非法指令异常
<code>sfence.vma</code>	刷新 TLB 缓存：在 U 模式下执行会产生非法指令异常
访问 S 模式 CSR 的指令	通过访问 <code>sepc</code> / <code>stvec</code> / <code>scause</code> / <code>sscratch</code> / <code>stval</code> / <code>sstatus</code> / <code>satp</code> 等 CSR 来改变系统状态：在 U 模式下执行会产生非法指令异常

在下一节中，我们将看到在 *U* 模式下运行的用户态应用程序，如果执行上述 *S* 模式特权指令，将会产生非法指令异常，从而看出 RISC-V 的特权模式提供了对操作系统一定程度的保护。

3.3 实现应用程序

3.3.1 本节导读

本节主要讲解如何设计实现被批处理系统逐个加载并运行的应用程序。这有个前提，即应用程序假定在用户态（*U* 特权级模式）下运行。实际上，如果应用程序的代码都符合用户态特权级的约束，那它完全可以正常在用户态中运行；但如果应用程序执行特权指令或非法操作（如执行非法指令，访问一个非法的地址等），那会产生异常，并导致程序退出。保证应用程序的代码在用户态能正常运行是将要实现的批处理系统的关键任务之一。应用程序的设计实现要点是：

- 应用程序的内存布局
- 应用程序发出的系统调用

从某种程度上讲，这里设计的应用程序与第一章中的最小用户态执行环境有很多相同的地方。即设计一个应用程序和基本的支持功能库，这样应用程序在用户态通过操作系统提供的服务完成自身的任务。

3.3.2 应用程序设计

应用程序、用户库（包括入口函数、初始化函数、I/O 函数和系统调用接口等多个 *rs* 文件组成）放在项目根目录的 *user* 目录下，它和第一章的裸机应用不同之处主要在项目的目录文件结构和内存布局上：

- *user/src/bin/*.rs*：各个应用程序
- *user/src/*.rs*：用户库（包括入口函数、初始化函数、I/O 函数和系统调用接口等）
- *user/src/linker.ld*：应用程序的内存布局说明。

项目结构

我们看到 *user/src* 目录下面多出了一个 *bin* 目录。*bin* 里面有很多文件，目前里面至少有三个程序（一个文件是一个应用程序），分别是：

- *hello_world*：在屏幕上打印一行 *Hello world from user mode program!*
- *store_fault*：访问一个非法的物理地址，测试批处理系统是否会被该错误影响
- *power*：不断在计算操作和打印字符串操作之间进行特权级切换

批处理系统会按照文件名开头的数字编号从小到大的顺序加载并运行它们。

每个应用程序的实现都在对应的单个文件中。打开其中一个文件，会看到里面只有一个 *main* 函数和若干相关的函数所形成的整个应用程序逻辑。

我们还能够看到代码中尝试引入了外部库：

```
# [macro_use]
extern crate user_lib;
```

这个外部库其实就是 *user* 目录下的 *lib.rs* 以及它引用的若干子模块中。至于这个外部库为何叫 *user_lib* 而不叫 *lib.rs* 所在的目录的名字 *user*，是因为在 *user/Cargo.toml* 中我们对于库的名字进行了设置： *name = "user_lib"*。它作为 *bin* 目录下的源程序所依赖的用户库，等价于其他编程语言提供的标准库。

在 lib.rs 中我们定义了用户库的入口点 _start :

```

1 #[no_mangle]
2 #[link_section = ".text.entry"]
3 pub extern "C" fn _start() -> ! {
4     clear_bss();
5     exit(main());
6     panic!("unreachable after sys_exit!");
7 }
```

第 2 行使用 Rust 的宏将 _start 这段代码编译后的汇编代码中放在一个名为 .text.entry 的代码段中，方便我们在后续链接的时候调整它的位置使得它能够作为用户库的入口。

从第 4 行开始，进入用户库入口之后，首先和第一章一样，手动清空需要零初始化的 .bss 段（很遗憾到目前为止底层的批处理系统还没有这个能力，所以我们只能在用户库中完成）；然后调用 main 函数得到一个类型为 i32 的返回值，最后调用用户库提供的 exit 接口退出应用程序，并将 main 函数的返回值告知批处理系统。

我们还在 lib.rs 中看到了另一个 main :

```

1 #[linkage = "weak"]
2 #[no_mangle]
3 fn main() -> i32 {
4     panic!("Cannot find main!");
5 }
```

第 1 行，我们使用 Rust 的宏将其函数符号 main 标志为弱链接。这样在最后链接的时候，虽然在 lib.rs 和 bin 目录下的某个应用程序都有 main 符号，但由于 lib.rs 中的 main 符号是弱链接，链接器会使用 bin 目录下的应用主逻辑作为 main。这里我们主要是进行某种程度上的保护，如果在 bin 目录下找不到任何 main，那么编译也能够通过，但会在运行时报错。

为了支持上述这些链接操作，我们需要在 lib.rs 的开头加入：

```
#![feature(linkage)]
```

内存布局

在 user/.cargo/config 中，我们和第一章一样设置链接时使用链接脚本 user/src/linker.ld。在其中我们做的重要的事情是：

- 将程序的起始物理地址调整为 0x80400000，三个应用程序都会被加载到这个物理地址上运行；
- 将 _start 所在的 .text.entry 放在整个程序的开头，也就是说批处理系统只要在加载之后跳转到 0x80400000 就已经进入了用户库的入口点，并会在初始化之后跳转到应用程序主逻辑；
- 提供了最终生成可执行文件的 .bss 段的起始和终止地址，方便 clear_bss 函数使用。

其余的部分和第一章基本相同。

系统调用

在子模块 `syscall` 中，应用程序通过 `ecall` 调用批处理系统提供的接口，由于应用程序运行在用户态（即 U 模式），`ecall` 指令会触发名为 *Environment call from U-mode* 的异常，并 Trap 进入 S 模式执行批处理系统针对这个异常特别提供的服务代码。由于这个接口处于 S 模式的批处理系统和 U 模式应用程序之间，从上一节我们可以知道，这个接口可以被称为 ABI 或者系统调用。现在我们不关心底层的批处理系统如何提供应用程序所需的功能，只是站在应用程序的角度去使用即可。

在本章中，应用程序和批处理系统之间按照 API 的结构，约定如下两个系统调用：

列表 1: 第二章新增系统调用

```
/// 功能：将内存中缓冲区中的数据写入文件。
/// 参数：`fd` 表示待写入文件的文件描述符；
///        `buf` 表示内存中缓冲区的起始地址；
///        `len` 表示内存中缓冲区的长度。
/// 返回值：返回成功写入的长度。
/// syscall ID: 64
fn sys_write(fd: usize, buf: *const u8, len: usize) -> isize;

/// 功能：退出应用程序并将返回值告知批处理系统。
/// 参数：`exit_code` 表示应用程序的返回值。
/// 返回值：该系统调用不应该返回。
/// syscall ID: 93
fn sys_exit(exit_code: usize) -> !;
```

我们知道系统调用实际上是汇编指令级的二进制接口，因此这里给出的只是使用 Rust 语言描述的 API 版本。在实际调用的时候，我们需要按照 RISC-V 调用规范（即 ABI 格式）在合适的寄存器中放置系统调用的参数，然后执行 `ecall` 指令触发 Trap。在 Trap 回到 U 模式应用程序代码之后，会从 `ecall` 的下一条指令继续执行，同时我们能够按照调用规范在合适的寄存器中读取返回值。

注解：RISC-V 寄存器编号和别名

RISC-V 寄存器编号从 0~31，表示为 x0~x31。其中：

- x10~x17：对应 a0~a7
- x1：对应 ra

在 RISC-V 调用规范中，和函数调用的 ABI 情形类似，约定寄存器 a0~a6 保存系统调用的参数，a0 保存系统调用的返回值。有些许不同的是寄存器 a7 用来传递 `syscall` ID，这是因为所有的 `syscall` 都是通过 `ecall` 指令触发的，除了各输入参数之外我们还额外需要一个寄存器来保存要请求哪个系统调用。由于这超出了 Rust 语言的表达能力，我们需要在代码中使用内嵌汇编来完成参数/返回值绑定和 `ecall` 指令的插入：

```
1 // user/src/syscall.rs
2 use core::arch::asm;
3 fn syscall(id: usize, args: [usize; 3]) -> isize {
4     let mut ret: isize;
5     unsafe {
6         asm!(
7             "ecall",
8             inlateout("x10") args[0] => ret,
9             in("x11") args[1],
10            in("x12") args[2],
11            in("x17") id
12         );
13     }
14 }
```

(下页继续)

(续上页)

```

14     ret
15 }
```

第 3 行，我们将所有的系统调用都封装成 `syscall` 函数，可以看到它支持传入 `syscall` ID 和 3 个参数。

`syscall` 中使用从第 5 行开始的 `asm!` 宏嵌入 `ecall` 指令来触发系统调用。在第一章中，我们曾经使用 `global_asm!` 宏来嵌入全局汇编代码，而这里的 `asm!` 宏可以将汇编代码嵌入到局部的函数上下文中。相比 `global_asm!`，`asm!` 宏可以获取上下文中的变量信息并允许嵌入的汇编代码对这些变量进行操作。由于编译器的能力不足以判定插入汇编代码这个行为的安全性，所以我们需要将其包裹在 `unsafe` 块中自己来对它负责。

从 RISC-V 调用规范来看，就像函数有着输入参数和返回值一样，`ecall` 指令同样有着输入和输出寄存器：`a0~a2` 和 `a7` 作为输入寄存器分别表示系统调用参数和系统调用 ID，而当系统调用返回后，`a0` 作为输出寄存器保存系统调用的返回值。在函数上下文中，输入参数数组 `args` 和变量 `id` 保存系统调用参数和系统调用 ID，而变量 `ret` 保存系统调用返回值，它也是函数 `syscall` 的输出/返回值。这些输入/输出变量可以和 `ecall` 指令的输入/输出寄存器一一对应。如果完全由我们自己编写汇编代码，那么如何将变量绑定到寄存器则成了一个难题：比如，在 `ecall` 指令被执行之前，我们需要将寄存器 `a7` 的值设置为变量 `id` 的值，那么我们首先需要知道目前变量 `id` 的值保存在哪里，它可能在栈上也有可能在某个寄存器中。

作为程序员我们并不知道这些只有编译器才知道的信息，因此我们只能在编译器的帮助下完成变量到寄存器的绑定。现在来看 `asm!` 宏的格式：首先在第 6 行是我们要插入的汇编代码段本身，这里我们只插入一行 `ecall` 指令，不过它可以支持同时插入多条指令。从第 7 行开始我们在编译器的帮助下将输入/输出变量绑定到寄存器。比如第 8 行的 `in("x11") args[1]` 则表示将输入参数 `args[1]` 绑定到 `ecall` 的输入寄存器 `x11` 即 `a1` 中，编译器自动插入相关指令并保证在 `ecall` 指令被执行之前寄存器 `a1` 的值与 `args[1]` 相同。以同样的方式我们可以将输入参数 `args[2]` 和 `id` 分别绑定到输入寄存器 `a2` 和 `a7` 中。这里比较特殊的是 `a0` 寄存器，它同时作为输入和输出，因此我们将 `in` 改成 `inlateout`，并在行末的变量部分使用 `{in_var} => {out_var}` 的格式，其中 `{in_var}` 和 `{out_var}` 分别表示上下文中的输入变量和输出变量。

有些时候不必将变量绑定到固定的寄存器，此时 `asm!` 宏可以自动完成寄存器分配。某些汇编代码段还会带来一些编译器无法预知的副作用，这种情况下需要在 `asm!` 中通过 `options` 告知编译器这些可能的副作用，这样可以帮助编译器在避免出错更加高效分配寄存器。事实上，`asm!` 宏远比我们这里介绍的更加强大易用，详情参考 Rust 相关 RFC 文档¹。

上面这一段汇编代码的含义和内容与第一章中的 *RustSBI* 输出到屏幕的 *SBI* 调用汇编代码涉及的汇编指令一样，但传递参数的寄存器的含义是不同的。有兴趣的同学可以回顾第一章的 `console.rs` 和 `sbi.rs`。

于是 `sys_write` 和 `sys_exit` 只需将 `syscall` 进行包装：

```

1 // user/src/syscall.rs
2
3 const SYSCALL_WRITE: usize = 64;
4 const SYSCALL_EXIT: usize = 93;
5
6 pub fn sys_write(fd: usize, buffer: &[u8]) -> isize {
7     syscall(SYSCALL_WRITE, [fd, buffer.as_ptr() as usize, buffer.len()])
8 }
9
10 pub fn sys_exit(xstate: i32) -> isize {
11     syscall(SYSCALL_EXIT, [xstate as usize, 0, 0])
12 }
```

注意 `sys_write` 使用一个 `&[u8]` 切片类型来描述缓冲区，这是一个 **胖指针** (Fat Pointer)，里面既包含缓冲区的起始地址，还包含缓冲区的长度。我们可以分别通过 `as_ptr` 和 `len` 方法取出它们并独立地作为实际的系统调用参数。

¹ <https://doc.rust-lang.org/reference/inline-assembly.html>

我们将上述两个系统调用在用户库 `user_lib` 中进一步封装，从而更加接近在 Linux 等平台的实际系统调用接口：

```

1 // user/src/lib.rs
2 use syscall::*;
3
4 pub fn write(fd: usize, buf: &[u8]) -> isize { sys_write(fd, buf) }
5 pub fn exit(exit_code: i32) -> isize { sys_exit(exit_code) }

```

我们把 `console` 子模块中 `Stdout::write_str` 改成基于 `write` 的实现，且传入的 `fd` 参数设置为 1，它代表标准输出，也就是输出到屏幕。目前我们不需要考虑其他的 `fd` 选取情况。这样，应用程序的 `println!` 宏借助系统调用变得可用。参考下面的代码片段：

```

1 // user/src/console.rs
2 const STDOUT: usize = 1;
3
4 impl Write for Stdout {
5     fn write_str(&mut self, s: &str) -> fmt::Result {
6         write(STDOUT, s.as_bytes());
7         Ok(())
8     }
9 }

```

`exit` 接口则在用户库中的 `_start` 内使用，当应用程序主逻辑 `main` 返回之后，使用它退出应用程序并将返回值告知底层的批处理系统。

3.3.3 编译生成应用程序二进制码

这里简要介绍一下应用程序的自动构建。只需要在 `user` 目录下 `make build` 即可：

1. 对于 `src/bin` 下的每个应用程序，在 `target/riscv64gc-unknown-none-elf/release` 目录下生成一个同名的 ELF 可执行文件；
2. 使用 `objcopy` 二进制工具将上一步中生成的 ELF 文件删除所有 ELF header 和符号得到 `.bin` 后缀的纯二进制镜像文件。它们将被链接进内核并由内核在合适的时机加载到内存。

3.3.4 实现操作系统前执行应用程序

我们还没有实现操作系统，能提前执行或测试应用程序吗？可以！这是因为我们除了一个能模拟一台 RISC-V 64 计算机的全系统模拟器 `qemu-system-riscv64` 外，还有一个直接支持运行 RISC-V 64 用户程序的半系统模拟器 `qemu-riscv64`。不过需要注意的是，如果想让用户态应用程序在 `qemu-riscv64` 模拟器（实际上是一个 RISC-V 架构下的 Linux 操作系统）上和在我们自己写的 OS 上执行效果一样，需要做到二者的系统调用的接口是一样的（包括系统调用编号，参数约定的具体的寄存器和栈等）。

注解：Qemu 的用户态模拟和系统级模拟

Qemu 有两种运行模式：用户态模拟（User mode）和系统级模拟（System mode）。在 RISC-V 架构中，用户态模拟可使用 `qemu-riscv64` 模拟器，它可以模拟一台预装了 Linux 操作系统的 RISC-V 计算机。但是一般情况下我们并不通过输入命令来与之交互（就像我们正常使用 Linux 操作系统一样），它仅支持载入并执行单个可执行文件。具体来说，它可以解析基于 RISC-V 的应用级 ELF 可执行文件，加载到内存并跳转到入口点开始执行。在翻译并执行指令时，如果碰到是系统调用相关的汇编指令，它会把不同处理器（如 RISC-V）的 Linux 系统调用转换为本机处理器（如 x86-64）上的 Linux 系统调用，这样就可以让本机 Linux 完成系统调用，并返回结果（再转换成 RISC-V 能识别的数据）给这些应用。相对的，我们使用 `qemu-system-riscv64` 模拟器来系统级模拟一台 RISC-V 64 裸机，它包含处理器、内存及其他外部设备，支持运行完整的操作系统。

假定我们已经完成了编译并生成了 ELF 可执行文件格式的应用程序，我们就可以来试试。首先看看应用程序执行 RV64 的 S 模式特权指令 会出现什么情况，对应的应用程序可以在 user/src/bin 目录下找到。

```
// user/src/bin/03priv_inst.rs
use core::arch::asm;
#[no_mangle]
fn main() -> i32 {
    println!("Try to execute privileged instruction in U Mode");
    println!("Kernel should kill this application!");
    unsafe {
        asm!("sret");
    }
    0
}

// user/src/bin/04priv_csr.rs
use riscv::register::sstatus::{self, SPP};
#[no_mangle]
fn main() -> i32 {
    println!("Try to access privileged CSR in U Mode");
    println!("Kernel should kill this application!");
    unsafe {
        sstatus::set_spp(SPP::User);
    }
    0
}
```

在上述代码中，两个应用都会打印提示信息，随后应用 03priv_inst 会尝试在用户态执行内核态的特权指令 sret，而应用 04priv_csr 则会试图在用户态修改内核态 CSR sstatus。

接下来，我们尝试在用户态模拟器 qemu-riscv64 执行这两个应用：

```
$ cd user
$ make build
$ cd target/riscv64gc-unknown-none-elf/release/
# 确认待执行的应用为 ELF 格式
$ file 03priv_inst
03priv_inst: ELF 64-bit LSB executable, UCB RISC-V, version 1 (SYSV), statically linked, not stripped
# 执行特权指令出错
$ qemu-riscv64 ./03priv_inst
Try to execute privileged instruction in U Mode
Kernel should kill this application!
Illegal instruction (core dumped)
# 执行访问特权级 CSR 的指令出错
$ qemu-riscv64 ./04priv_csr
Try to access privileged CSR in U Mode
Kernel should kill this application!
Illegal instruction (core dumped)
```

看来 RV64 的特权级机制确实有用。那对于一般的用户态应用程序，在 qemu-riscv64 模拟器下能正确执行吗？

```
$ cd user/target/riscv64gc-unknown-none-elf/release/
$ qemu-riscv64 ./00hello_world
Hello, world!
# 正确显示了字符串
$ qemu-riscv64 ./01store_fault
```

(下页继续)

(续上页)

```
Into Test store_fault, we will insert an invalid store operation...
Kernel should kill this application!
Segmentation fault (core dumped)
# 故意访问了一个非法地址, 导致应用和 qemu-riscv64 被 Linux 内核杀死
$ qemu-riscv64 ./02power
3^10000=Segmentation fault (core dumped)
# 由于 Qemu 和 Rust 编译器版本不匹配, 无法正常运行
```

可以看到, 除了 02power 之外, 其余两个应用程序都能够执行并顺利结束。这是由于它们在运行时得到了操作系统 Linux for RISC-V 64 的支持。而 02power 的例子也说明我们应用的兼容性比较受限, 当应用用到较多特性时很可能就不再兼容 Qemu 了。我们期望在下一节开始实现的泥盆纪“邓式鱼”操作系统也能够正确加载和执行这些应用程序。

3.4 实现批处理操作系统

3.4.1 本节导读

从本节开始我们将着手实现批处理操作系统——即泥盆纪“邓式鱼”操作系统。在批处理操作系统中, 每当一个应用执行完毕, 我们都需要将下一个要执行的应用的代码和数据加载到内存。在具体实现其批处理执行应用程序功能之前, 本节我们首先实现该应用加载机制, 也即: 在操作系统和应用程序需要被放置到同一个可执行文件的前提下, 设计一种尽量简洁的应用放置和加载方式, 使得操作系统容易找到应用被放置到的位置, 从而在批处理操作系统和应用程序之间建立起联系的纽带。具体而言, 应用放置采用“静态绑定”的方式, 而操作系统加载应用则采用“动态加载”的方式:

- 静态绑定: 通过一定的编程技巧, 把多个应用程序代码和批处理操作系统代码“绑定”在一起。
- 动态加载: 基于静态编码留下的“绑定”信息, 操作系统可以找到每个应用程序文件二进制代码的起始地址和长度, 并能加载到内存中运行。

这里与硬件相关且比较困难的地方是如何让在内核态的批处理操作系统启动应用程序, 且能让应用程序在用户态正常执行。本节会讲大致过程, 而具体细节将放到下一节具体讲解。

3.4.2 将应用程序链接到内核

在本章中, 我们把应用程序的二进制镜像文件 (从 ELF 格式可执行文件剥离元数据, 参考前面章节) 作为内核的数据段链接到内核里面, 因此内核需要知道内含的应用程序的数量和它们的位置, 这样才能够在运行时对它们进行管理并能够加载到物理内存。

在 `os/src/main.rs` 中能够找到这样一行:

```
global_asm!(include_str!("link_app.S"));
```

这里我们引入了一段汇编代码 `link_app.S`, 它一开始并不存在, 而是在构建操作系统时自动生成的。当我们使用 `make run` 让系统运行的过程中, 这个汇编代码 `link_app.S` 就生成了。我们可以先来看一看 `link_app.S` 里面的内容:

```
1 # os/src/link_app.S
2
3     .align 3
4     .section .data
5     .global _num_app
6 _num_app:
```

(下页继续)

(续上页)

```

7     .quad 5
8     .quad app_0_start
9     .quad app_1_start
10    .quad app_2_start
11    .quad app_3_start
12    .quad app_4_start
13    .quad app_4_end
14
15    .section .data
16    .global app_0_start
17    .global app_0_end
18 app_0_start:
19     .incbin "../user/target/riscv64gc-unknown-none-elf/release/00hello_world.bin"
20 app_0_end:
21
22    .section .data
23    .global app_1_start
24    .global app_1_end
25 app_1_start:
26     .incbin "../user/target/riscv64gc-unknown-none-elf/release/01store_fault.bin"
27 app_1_end:
28
29    .section .data
30    .global app_2_start
31    .global app_2_end
32 app_2_start:
33     .incbin "../user/target/riscv64gc-unknown-none-elf/release/02power.bin"
34 app_2_end:
35
36    .section .data
37    .global app_3_start
38    .global app_3_end
39 app_3_start:
40     .incbin "../user/target/riscv64gc-unknown-none-elf/release/03priv_inst.bin"
41 app_3_end:
42
43    .section .data
44    .global app_4_start
45    .global app_4_end
46 app_4_start:
47     .incbin "../user/target/riscv64gc-unknown-none-elf/release/04priv_csr.bin"
48 app_4_end:

```

可以看到第 15 行开始的五个数据段分别插入了五个应用程序的二进制镜像，并且各自有一对全局符号 `app_*_start`, `app_*_end` 指示它们的开始和结束位置。而第 3 行开始的另一个数据段相当于一个 64 位整数数组。数组中的第一个元素表示应用程序的数量，后面则按照顺序放置每个应用程序的起始地址，最后一个元素放置最后一个应用程序的结束位置。这样每个应用程序的位置都能从该数组中相邻两个元素中得知。这个数组所在的位置同样也由全局符号 `_num_app` 所指示。

这个文件是在 `cargo build` 的时候，由脚本 `os/build.rs` 控制生成的。有兴趣的同学可以参考其代码。

3.4.3 找到并加载应用程序二进制码

能够找到并加载应用程序二进制码的应用管理器 AppManager 是“邓式鱼”操作系统的核组件。我们在 os 的 batch 子模块中实现一个应用管理器，它的主要功能是：

- 保存应用数量和各自的位置信息，以及当前执行到第几个应用了。
- 根据应用程序位置信息，初始化好应用所需内存空间，并加载应用执行。

应用管理器 AppManager 结构体定义如下：

```
// os/src/batch.rs

struct AppManager {
    num_app: usize,
    current_app: usize,
    app_start: [usize; MAX_APP_NUM + 1],
}
```

这里我们可以看出，上面提到的应用管理器需要保存和维护的信息都在 AppManager 里面。这样设计的原因在于：我们希望将 AppManager 实例化为一个全局变量，使得任何函数都可以直接访问。但是里面的 current_app 字段表示当前执行的是第几个应用，它是一个可修改的变量，会在系统运行期间发生变化。因此在声明全局变量的时候，采用 static mut 是一种比较简单自然的方法。但是在 Rust 中，任何对于 static mut 变量的访问控制都是 unsafe 的，而我们要在编程中尽量避免使用 unsafe，这样才能让编译器负责更多的安全性检查。因此，我们需要考虑如何在尽量避免触及 unsafe 的情况下仍能声明并使用可变的全局变量。

注解：Rust Tips：Rust 所有权模型和借用检查

我们这里简单介绍一下 Rust 的所有权模型。它可以用一句话来概括：值（Value）在同一时间只能被绑定到一个变量（Variable）上。这里，“值”指的是储存在内存中固定位置，且格式属于某种特定类型的数据；而变量就是我们在 Rust 代码中通过 let 声明的局部变量或者函数的参数等，变量的类型与值的类型相匹配。在这种情况下，我们称值的 所有权（Ownership）属于它被绑定到的变量，且变量可以作为访问/控制绑定到它上面的值的一个媒介。变量可以将它拥有的值的所有权转移给其他变量，或者当变量退出其作用域之后，它拥有的值也会被销毁，这意味着值占用的内存或其他资源会被回收。

有些场景下，特别是在函数调用的时候，我们并不希望将当前上下文中的值的所有权转移到其他上下文中，因此类似于 C/C++ 中的按引用传参，Rust 可以使用 & 或 &mut 后面加上值被绑定到的变量的名字来分别生成值的不可变引用和可变引用，我们称这些引用分别不可变/可变 借用（Borrow）它们引用的值。顾名思义，我们可以通过可变引用修改它借用的值，但通过不可变引用则只能读取而不能修改。这些引用同样是需要被绑定到变量上的值，只是它们的类型是引用类型。在 Rust 中，引用类型的使用需要被编译器检查，但在数据表达上，和 C 的指针一样它只记录它借用的值所在的地址，因此在内存中它随平台不同仅会占据 4 字节或 8 字节空间。

无论值的类型是否是引用类型，我们都定义值的 生存期（Lifetime）为代码执行期间该值必须持续合法的代码区域集合（见¹），大概可以理解为该值在代码中的哪些地方被用到了：简单情况下，它可能等同于拥有它的变量的作用域，也有可能是从它被绑定开始直到它的拥有者变量最后一次出现或是它被解绑。

当我们使用 & 和 &mut 来借用值的时候，则我们编写的代码必须满足某些约束条件，不然无法通过编译：

- 不可变/可变引用的生存期不能 超出（Outlive）它们借用的值的生存期，也即：前者必须是后者的子集；
- 同一时间，借用同一个值的不可变和可变引用不能共存；
- 同一时间，借用同一个值的不可变引用可以存在多个，但可变引用只能存在一个。

这是为了 Rust 内存安全而设计的重要约束条件。第一条很好理解，如果值的生存期未能完全覆盖借用它的引用的生存期，就会在某一时刻发生值已被销毁而我们仍然尝试通过引用访问该值的情形。反过来说，显

¹ <https://doc.rust-lang.org/nomicon/lifetimes.html>

然当值合法时引用才有意义。最典型的例子是 **悬垂指针** (Dangling Pointer) 问题：即我们尝试在一个函数中返回函数中声明的局部变量的引用，并在调用者函数中试图通过该引用访问已被销毁的局部变量，这会产生未定义行为并导致错误。第二、三条的主要目的则是为了避免通过多个引用对同一个值进行的读写操作产生冲突。例如，当对同一个值的读操作和写操作在时间上相互交错时（即不可变/可变引用的生存期部分重叠），读操作便有可能读到被修改到一半的值，通常这会是一个不合法的值从而导致程序无法正确运行。这可能是由于我们在编程上的疏忽，使得我们在读取一个值的时候忘记它目前正处在被修改到一半的状态，一个可能的例子是在 C++ 中正对容器进行迭代访问的时候修改了容器本身。也有可能被归结为 **别名** (Aliasing) 问题，例如在 C 函数中有两个指针参数，如果它们指向相同的地址且编译器没有注意到这一点就进行过激的优化，将会使得编译结果偏离我们期望的语义。

上述约束条件要求借用同一个值的不可变引用和不可变/可变引用的生存期相互隔离，从而能够解决这些问题。Rust 编译器会在编译时使用 **借用检查器** (Borrow Checker) 检查这些约束条件是否被满足：其具体做法是尽可能精确的估计引用和值的生存期并将它们进行比较。随着 Rust 语言的愈发完善，其估计的精确度也会越来越高，使得程序员能够更容易通过借用检查。引用相关的借用检查发生在编译期，因此我们可以称其为编译期借用检查。

相对的，对值的借用方式运行时可变的情况下，我们可以使用 Rust 内置的数据结构将借用检查推迟到运行时，这可以称为运行时借用检查，它的约束条件和编译期借用检查一致。当我们想要发起借用或终止借用时，只需调用对应数据结构提供的接口即可。值的借用状态会占用一部分额外内存，运行时还会有额外的代码对借用合法性进行检查，这是为满足借用方式的灵活性产生的必要开销。当无法通过借用检查时，将会产生一个不可恢复错误，导致程序打印错误信息并立即退出。具体来说，我们通常使用 **RefCell** 包裹可被借用的值，随后调用 `borrow` 和 `borrow_mut` 便可发起借用并获得一个对值的不可变/可变借用的标志，它们可以像引用一样使用。为了终止借用，我们只需手动销毁这些标志或者等待它们被自动销毁。**RefCell** 的详细用法请参考²。

如果单独使用 `static` 而去掉 `mut` 的话，我们可以声明一个初始化之后就不可变的全局变量，但是我们需要 `AppManager` 里面的内容在运行时发生变化。这涉及到 Rust 中的 **内部可变性** (Interior Mutability)，也即在变量自身不可变或仅在不可变借用的情况下仍能修改绑定到变量上的值。我们可以通过用上面提到的 `RefCell` 来包裹 `AppManager`，这样 `RefCell` 无需被声明为 `mut`，同时被包裹的 `AppManager` 也能被修改。但是，我们能否将 `RefCell` 声明为一个全局变量呢？让我们写一小段代码试一试：

```
// https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&
→gist=18b0f956b83e6a8a408215edcfcb6d01
use std::cell::RefCell;
static A: RefCell<i32> = RefCell::new(3);
fn main() {
    *A.borrow_mut() = 4;
    println!("{}", A.borrow());
}
```

这段代码无法通过编译，其错误是：

```
error[E0277]: `RefCell<i32>` cannot be shared between threads safely
--> src/main.rs:2:1
|
2 | static A: RefCell<i32> = RefCell::new(3);
| ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ `RefCell<i32>` cannot be shared between_
→threads safely
|
= help: the trait `Sync` is not implemented for `RefCell<i32>`
= note: shared static variables must have a type that implements `Sync`

For more information about this error, try `rustc --explain E0277`.
```

Rust 编译器提示我们 `RefCell<i32>` 未被标记为 `Sync`，因此 Rust 编译器认为它不能被安全的在线程间共

² <https://doc.rust-lang.org/stable/std/cell/struct.RefCell.html>

享，也就不能作为全局变量使用。这可能会令人迷惑，这只是一个单线程程序，因此它不会有任何线程间共享数据的行为，为什么不能通过编译呢？事实上，Rust 对于并发安全的检查较为粗糙，当声明一个全局变量的时候，编译器会默认程序员会在多线程上使用它，而并不会检查程序员是否真的这样做。如果一个变量实际上仅会在单线程上使用，那 Rust 会期待我们将变量分配在栈上作为局部变量而不是全局变量。目前我们的内核仅支持单核，也就意味着只有单线程，那么我们可不可以使用局部变量来绕过这个错误呢？

很可惜，在这里和后面章节的很多场景中，有些变量无法作为局部变量使用。这是因为后面内核会并发执行多条控制流，这些控制流都会用到这些变量。如果我们最初将变量分配在某条控制流的栈上，那么我们就需要考虑如何将变量传递到其他控制流上，由于控制流的切换等操作并非常规的函数调用，我们很难将变量传递出去。因此最方便的做法是使用全局变量，这意味着在程序的任何地方均可随意访问它们，自然也包括这些控制流。

除了 Sync 的问题之外，看起来 RefCell 已经非常接近我们的需求了，因此我们在 RefCell 的基础上再封装一个 UPSafeCell，它名字的含义是：允许我们在单核上安全使用可变全局变量。

```
// os/src/sync/up.rs

pub struct UPSafeCell<T> {
    /// inner data
    inner: RefCell<T>,
}

unsafe impl<T> Sync for UPSafeCell<T> {}

impl<T> UPSafeCell<T> {
    /// User is responsible to guarantee that inner struct is only used in
    /// uniprocessor.
    pub unsafe fn new(value: T) -> Self {
        Self { inner: RefCell::new(value) }
    }
    /// Panic if the data has been borrowed.
    pub fn exclusive_access(&self) -> RefMut<'_, T> {
        self.inner.borrow_mut()
    }
}
```

UPSafeCell 对于 RefCell 简单进行封装，它和 RefCell 一样提供内部可变性和运行时借用检查，只是更加严格：调用 exclusive_access 可以得到它包裹的数据的独占访问权。因此当我们访问数据时（无论读还是写），需要首先调用 exclusive_access 获得数据的可变借用标记，通过它可以完成数据的读写，在操作完成之后我们需要销毁这个标记，此后才能开始对该数据的下一次访问。相比 RefCell 它不再允许多个读操作同时存在。

这段代码里面出现了两个 unsafe：

- 首先 new 被声明为一个 unsafe 函数，是因为我们希望使用者在创建一个 UPSafeCell 的时候保证在访问 UPSafeCell 内包裹的数据的时候始终不违背上述模式：即访问之前调用 exclusive_access，访问之后销毁借用标记再进行下一次访问。这只能依靠使用者自己来保证，但我们提供了一个保底措施：当使用者违背了上述模式，比如访问之后忘记销毁就开启下一次访问时，程序会 panic 并退出。
- 另一方面，我们将 UPSafeCell 标记为 Sync 使得它可以作为一个全局变量。这是 unsafe 行为，因为编译器无法确定我们的 UPSafeCell 能否安全的在多线程间共享。而我们能够向编译器做出保证，第一个原因是目前我们内核仅运行在单核上，因此无需在意任何多核引发的数据竞争/同步问题；第二个原因则是它基于 RefCell 提供了运行时借用检查功能，从而满足了 Rust 对于借用的基本约束进而保证了内存安全。

这样，我们就以尽量少的 unsafe code 来初始化 AppManager 的全局实例 APP_MANAGER：

```
// os/src/batch.rs

lazy_static! {
    static ref APP_MANAGER: UPSafeCell<AppManager> = unsafe { UPSafeCell::new({
        extern "C" { fn _num_app(); }
        let num_app_ptr = _num_app as *const u8;
        let num_app = num_app_ptr.read_volatile();
        let mut app_start: [u8; MAX_APP_NUM + 1] = [0; MAX_APP_NUM + 1];
        let app_start_raw: &[u8] = core::slice::from_raw_parts(
            num_app_ptr.add(1), num_app + 1
        );
        app_start[..=num_app].copy_from_slice(app_start_raw);
        AppManager {
            num_app,
            current_app: 0,
            app_start,
        }
    })};
}
```

初始化的逻辑很简单，就是找到 `link_app.S` 中提供的符号 `_num_app`，并从这里开始解析出应用数量以及各个应用的起始地址。注意其中对于切片类型的使用能够很大程度上简化编程。

这里我们使用了外部库 `lazy_static` 提供的 `lazy_static!` 宏。要引入这个外部库，我们需要加入依赖：

```
# os/Cargo.toml

[dependencies]
lazy_static = { version = "1.4.0", features = ["spin_no_std"] }
```

`lazy_static!` 宏提供了全局变量的运行时初始化功能。一般情况下，全局变量必须在编译期设置一个初始值，但是有些全局变量依赖于运行期间才能得到的数据作为初始值。这导致这些全局变量需要在运行时发生变化，即需要重新设置初始值之后才能使用。如果我们手动实现的话有诸多不便之处，比如需要把这种全局变量声明为 `static mut` 并衍生出很多 `unsafe` 代码。这种情况下我们可以使用 `lazy_static!` 宏来帮助我们解决这个问题。这里我们借助 `lazy_static!` 声明了一个 `AppManager` 结构的名为 `APP_MANAGER` 的全局实例，且只有在它第一次被使用到的时候，才会进行实际的初始化工作。

因此，借助我们设计的 `UPSafeCell<T>` 和外部库 `lazy_static!`，我们就能使用尽量少的 `unsafe` 代码完成可变全局变量的声明和初始化，且一旦初始化完成，在后续的使用过程中便不再触及 `unsafe` 代码。

`AppManager` 的方法中，`print_app_info/get_current_app/move_to_next_app` 都相当简单直接，需要说明的是 `load_app`：

```
1 unsafe fn load_app(&self, app_id: u8) {
2     if app_id >= self.num_app {
3         panic!("All applications completed!");
4     }
5     println!("[kernel] Loading app_{}", app_id);
6     // clear app area
7     core::slice::from_raw_parts_mut(
8         APP_BASE_ADDRESS as *mut u8,
9         APP_SIZE_LIMIT
10    ).fill(0);
11    let app_src = core::slice::from_raw_parts(
12        self.app_start[app_id] as *const u8,
13        self.app_start[app_id + 1] - self.app_start[app_id]
14    );
15    let app_dst = core::slice::from_raw_parts_mut(
```

(下页继续)

(续上页)

```

16     APP_BASE_ADDRESS as *mut u8,
17     app_src.len()
18 );
19 app_dst.copy_from_slice(app_src);
20 // memory fence about fetching the instruction memory
21 asm! ("fence.i");
22 }
```

这个方法负责将参数 `app_id` 对应的应用程序的二进制镜像加载到物理内存以 `0x80400000` 起始的位置，这个位置是批处理操作系统和应用程序之间约定的常数地址，回忆上一小节中，我们也调整应用程序的内存布局以同一个地址开头。第 7 行开始，我们首先将一块内存清空，然后找到待加载应用二进制镜像的位置，并将它复制到正确的位置。它本质上是把数据从一块内存复制到另一块内存，从批处理操作系统的角度来看，是将操作系统数据段的一部分数据（实际上是应用程序）复制到了一个可以执行代码的内存区域。在这一点上也体现了冯诺依曼计算机的 **代码即数据** 的特征。

注意在第 21 行我们在加载完应用代码之后插入了一条奇怪的汇编指令 `fence.i`，它起到什么作用呢？我们知道缓存是存储层级结构中提高访存速度的很重要一环。而 CPU 对物理内存所做的缓存又分成 **数据缓存** (`d-cache`) 和 **指令缓存** (`i-cache`) 两部分，分别在 CPU 访存和取指的时候使用。在取指的时候，对于一个指令地址，CPU 会先去 `i-cache` 里面看一下它是否在某个已缓存的缓存行内，如果在的话它就会直接从高速缓存中拿到指令而不是通过总线访问内存。通常情况下，CPU 会认为程序的代码段不会发生变化，因此 `i-cache` 是一种只读缓存。但在这里，OS 将修改会被 CPU 取指的内存区域，这会使得 `i-cache` 中含有与内存中不一致的内容。因此，OS 在这里必须使用取指屏障指令 `fence.i`，它的功能是保证 **在它之后的取指过程必须能够看到在它之前的所有对于取指内存区域的修改**，这样才能保证 CPU 访问的应用代码是最新的而不是 `i-cache` 中过时的内容。至于硬件是如何实现 `fence.i` 这条指令的，这一点每个硬件的具体实现方式都可能不同，比如直接清空 `i-cache` 中所有内容或者标记其中某些内容不合法等等。

警告：模拟器与真机的不同之处

至少在 Qemu 模拟器的默认配置下，各类缓存如 `i-cache/d-cache/TLB` 都处于机制不完全甚至完全不存在的状态。目前在 Qemu 平台上，即使我们不加上刷新 `i-cache` 的指令，大概率也是能够正常运行的。但在 K210 物理计算机上，如果没有执行汇编指令 `fence.i`，就会产生由于指令缓存的内容与对应内存中指令不一致导致的错误异常。

`batch` 子模块对外暴露出如下接口：

- `init`：调用 `print_app_info` 的时候第一次用到了全局变量 `APP_MANAGER`，它也是在这个时候完成初始化；
- `run_next_app`：批处理操作系统的核心操作，即加载并运行下一个应用程序。当批处理操作系统完成初始化或者一个应用程序运行结束或出错之后会调用该函数。我们下节再介绍其具体实现。

3.5 实现特权级的切换

3.5.1 本节导读

由于处理器具有硬件级的特权级机制，应用程序在用户态特权级运行时，是无法直接通过函数调用访问处于内核态特权级的批处理操作系统内核中的函数。但应用程序又需要得到操作系统提供的服务，所以应用程序与操作系统需要通过某种合作机制完成特权级之间的切换，使得用户态应用程序可以得到内核态操作系统函数的服务。本节将讲解在 RISC-V 64 处理器提供的 U/S 特权级下，批处理操作系统和应用程序如何相互配合，完成特权级切换的。

3.5.2 RISC-V 特权级切换

特权级切换的起因

我们知道，批处理操作系统被设计为运行在内核态特权级（RISC-V 的 S 模式），这是作为 SEE（Supervisor Execution Environment）的 RustSBI 所保证的。而应用程序被设计为运行在用户态特权级（RISC-V 的 U 模式），被操作系统为核心的执行环境监管起来。在本章中，这个应用程序的执行环境即是由“邓式鱼”批处理操作系统提供的 AEE（Application Execution Environment）。批处理操作系统为了建立好应用程序的执行环境，需要在执行应用程序之前进行一些初始化工作，并监控应用程序的执行，具体体现在：

- 当启动应用程序的时候，需要初始化应用程序的用户态上下文，并能切换到用户态执行应用程序；
- 当应用程序发起系统调用（即发出 Trap）之后，需要到批处理操作系统中进行处理；
- 当应用程序执行出错的时候，需要到批处理操作系统中杀死该应用并加载运行下一个应用；
- 当应用程序执行结束的时候，需要到批处理操作系统中加载运行下一个应用（实际上也是通过系统调用 `sys_exit` 来实现的）。

这些处理都涉及到特权级切换，因此需要应用程序、操作系统和硬件一起协同，完成特权级切换机制。

特权级切换相关的控制状态寄存器

当从一般意义上讨论 RISC-V 架构的 Trap 机制时，通常需要注意两点：

- 在触发 Trap 之前 CPU 运行在哪个特权级；
- CPU 需要切换到哪个特权级来处理该 Trap，并在处理完成之后返回原特权级。

但本章中我们仅考虑如下流程：当 CPU 在用户态特权级（RISC-V 的 U 模式）运行应用程序，执行到 Trap，切换到内核态特权级（RISC-V 的 S 模式），批处理操作系统的对应代码响应 Trap，并执行系统调用服务，处理完毕后，从内核态返回到用户态应用程序继续执行后续指令。

在 RISC-V 架构中，关于 Trap 有一条重要的规则：在 Trap 前的特权级不会高于 Trap 后的特权级。因此如果触发 Trap 之后切换到 S 特权级（下称 Trap 到 S），说明 Trap 发生之前 CPU 只能运行在 S/U 特权级。但无论如何，只要是 Trap 到 S 特权级，操作系统就会使用 S 特权级中与 Trap 相关的 **控制状态寄存器** (CSR, Control and Status Register) 来辅助 Trap 处理。我们在编写运行在 S 特权级的批处理操作系统中的 Trap 处理相关代码的时候，就需要使用如下所示的 S 模式的 CSR 寄存器。

表 4: 进入 S 特权级 Trap 的相关 CSR

CSR 名	该 CSR 与 Trap 相关的功能
sstatus	SPP 等字段给出 Trap 发生之前 CPU 处在哪个特权级 (S/U) 等信息
sepc	当 Trap 是一个异常的时候，记录 Trap 发生之前执行的最后一条指令的地址
scause	描述 Trap 的原因
stval	给出 Trap 附加信息
stvec	控制 Trap 处理代码的入口地址

注解: S 模式下最重要的 sstatus 寄存器

注意 `sstatus` 是 S 特权级最重要的 CSR，可以从多个方面控制 S 特权级的 CPU 行为和执行状态。

特权级切换

当执行一条 Trap 类指令（如 `ecall` 时），CPU 发现触发了一个异常并需要进行特殊处理，这涉及到 [执行环境切换](#)。具体而言，用户态执行环境中的应用程序通过 `ecall` 指令向内核态执行环境中的操作系统请求某项服务功能，那么处理器和操作系统会完成到内核态执行环境的切换，并在操作系统完成服务后，再次切换回用户态执行环境，然后应用程序会紧接着 `ecall` 指令的后一条指令位置处继续执行，参考图示。

应用程序被切换回来之后需要从发出系统调用请求的执行位置恢复应用程序上下文并继续执行，这需要在切换前后维持应用程序的上下文保持不变。应用程序的上下文包括通用寄存器和栈两个主要部分。由于 CPU 在不同特权级下共享一套通用寄存器，所以在运行操作系统的 Trap 处理过程中，操作系统也会用到这些寄存器，这会改变应用程序的上下文。因此，与函数调用需要保存函数调用上下文/活动记录一样，在执行操作系统的 Trap 处理过程（会修改通用寄存器）之前，我们需要在某个地方（某内存块或内核的栈）保存这些寄存器并在 Trap 处理结束后恢复这些寄存器。

除了通用寄存器之外还有一些可能在处理 Trap 过程中会被修改的 CSR，比如 CPU 所在的特权级。我们要保证它们的变化在我们的预期之内。比如，对于特权级转换而言，应该是 Trap 之前在 U 特权级，处理 Trap 的时候在 S 特权级，返回之后又需要回到 U 特权级。而对于栈问题则相对简单，只要两个应用程序执行过程中用来记录执行历史的栈所对应的内存区域不相交，就不会产生令我们头痛的覆盖问题或数据破坏问题，也就无需进行保存/恢复。

特权级切换的具体过程一部分由硬件直接完成，另一部分则需要由操作系统来实现。

3.5.3 特权级切换的硬件控制机制

当 CPU 执行完一条指令（如 `ecall`）并准备从用户特权级陷入（Trap）到 S 特权级的时候，硬件会自动完成如下这些事情：

- `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。

注解：`stvec` 相关细节

在 RV64 中，`stvec` 是一个 64 位的 CSR，在中断使能的情况下，保存了中断处理的入口地址。它有两个字段：

- MODE 位于 [1:0]，长度为 2 bits；
- BASE 位于 [63:2]，长度为 62 bits。

当 MODE 字段为 0 的时候，`stvec` 被设置为 Direct 模式，此时进入 S 模式的 Trap 无论原因如何，处理 Trap 的入口地址都是 $BASE \ll 2$ ，CPU 会跳转到这个地方进行异常处理。本书中我们只会将 `stvec` 设置为 Direct 模式。而 `stvec` 还可以被设置为 Vectored 模式，有兴趣的同学可以自行参考 RISC-V 指令集特权级规范。

而当 CPU 完成 Trap 处理准备返回的时候，需要通过一条 S 特权级的特权指令 `sret` 来完成，这一条指令具体完成以下功能：

- CPU 会将当前的特权级按照 `sstatus` 的 SPP 字段设置为 U 或者 S；
- CPU 会跳转到 `sepc` 寄存器指向的那条指令，然后继续执行。

这些基本上都是硬件不得不完成的事情，还有一些剩下的收尾工作可以都交给软件，让操作系统能有更大的灵活性。

3.5.4 用户栈与内核栈

在 Trap 触发的一瞬间，CPU 就会切换到 S 特权级并跳转到 stvec 所指示的位置。但是在正式进入 S 特权级的 Trap 处理之前，上面提到过我们必须保存原控制流的寄存器状态，这一般通过内核栈来保存。注意，我们需要用专门为操作系统准备的内核栈，而不是应用程序运行时用到的用户栈。

使用两个不同的栈主要是为了安全性：如果两个控制流（即应用程序的控制流和内核的控制流）使用同一个栈，在返回之后应用程序就能读到 Trap 控制流的历史信息，比如内核一些函数的地址，这样会带来安全隐患。于是，我们要做的是，在批处理操作系统中添加一段汇编代码，实现从用户栈切换到内核栈，并在内核栈上保存应用程序控制流的寄存器状态。

我们声明两个类型 KernelStack 和 UserStack 分别表示内核栈和用户栈，它们都只是字节数组的简单包装：

```

1 // os/src/batch.rs
2
3 const USER_STACK_SIZE: usize = 4096 * 2;
4 const KERNEL_STACK_SIZE: usize = 4096 * 2;
5
6 #[repr(align(4096))]
7 struct KernelStack {
8     data: [u8; KERNEL_STACK_SIZE],
9 }
10
11 #[repr(align(4096))]
12 struct UserStack {
13     data: [u8; USER_STACK_SIZE],
14 }
15
16 static KERNEL_STACK: KernelStack = KernelStack { data: [0; KERNEL_STACK_SIZE] };
17 static USER_STACK: UserStack = UserStack { data: [0; USER_STACK_SIZE] };

```

常数 USER_STACK_SIZE 和 KERNEL_STACK_SIZE 指出用户栈和内核栈的大小分别为 8KiB。两个类型是以全局变量的形式实例化在批处理操作系统的 .bss 段中的。

我们为两个类型实现了 get_sp 方法来获取栈顶地址。由于在 RISC-V 中栈是向下增长的，我们只需返回包裹的数组的结尾地址，以用户栈类型 UserStack 为例：

```

1 impl UserStack {
2     fn get_sp(&self) -> usize {
3         self.data.as_ptr() as usize + USER_STACK_SIZE
4     }
5 }

```

于是换栈是非常简单的，只需将 sp 寄存器的值修改为 get_sp 的返回值即可。

接下来是 Trap 上下文（即数据结构 TrapContext），类似前面提到的函数调用上下文，即在 Trap 发生时需要保存的物理资源内容，并将其一起放在一个名为 TrapContext 的类型中，定义如下：

```

1 // os/src/trap/context.rs
2
3 #[repr(C)]
4 pub struct TrapContext {
5     pub x: [usize; 32],
6     pub sstatus: Sstatus,
7     pub sepc: usize,
8 }

```

可以看到里面包含所有的通用寄存器 x0~x31，还有 sstatus 和 sepc。那么为什么需要保存它们呢？

- 对于通用寄存器而言，两条控制流（应用程序控制流和内核控制流）运行在不同的特权级，所属的软件也可能由不同的编程语言编写，虽然在 Trap 控制流中只是会执行 Trap 处理相关的代码，但依然可能直接或间接调用很多模块，因此很难甚至不可能找出哪些寄存器无需保存。既然如此我们就只能全部保存了。但这里也有一些例外，如 `x0` 被硬编码为 0，它自然不会有变化；还有 `tp(x4)` 寄存器，除非我们手动出于一些特殊用途使用它，否则一般也不会被用到。虽然它们无需保存，但我们仍然在 `TrapContext` 中为它们预留空间，主要是为了后续的实现方便。
- 对于 CSR 而言，我们知道进入 Trap 的时候，硬件会立即覆盖掉 `scause/stval/sstatus/sepc` 的全部或是其中一部分。`scause/stval` 的情况是：它总是在 Trap 处理的第一时间就被使用或者是在其他地方保存下来了，因此它没有被修改并造成不良影响的风险。而对于 `sstatus/sepc` 而言，它们会在 Trap 处理的全程有意义（在 Trap 控制流最后 `sret` 的时候还用到了它们），而且确实会出现 Trap 嵌套的情况使得它们的值被覆盖掉。所以我们需要将它们也一起保存下来，并在 `sret` 之前恢复原样。

3.5.5 Trap 管理

特权级切换的核心是对 Trap 的管理。这主要涉及到如下一些内容：

- 应用程序通过 `ecall` 进入到内核状态时，操作系统保存被打断的应用程序的 Trap 上下文；
- 操作系统根据 Trap 相关的 CSR 寄存器内容，完成系统调用服务的分发与处理；
- 操作系统完成系统调用服务后，需要恢复被打断的应用程序的 Trap 上下文，并通过 `sret` 让应用程序继续执行。

接下来我们具体介绍上述内容。

Trap 上下文的保存与恢复

首先是具体实现 Trap 上下文保存和恢复的汇编代码。

在批处理操作系统初始化的时候，我们需要修改 `stvec` 寄存器来指向正确的 Trap 处理入口点。

```

1 // os/src/trap/mod.rs
2
3 global_asm!(include_str!("trap.S"));
4
5 pub fn init() {
6     extern "C" { fn __alltraps(); }
7     unsafe {
8         stvec::write(__alltraps as usize, TrapMode::Direct);
9     }
10 }
```

这里我们引入了一个外部符号 `__alltraps`，并将 `stvec` 设置为 `Direct` 模式指向它的地址。我们在 `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。

Trap 处理的总体流程如下：首先通过 `__alltraps` 将 Trap 上下文保存在内核栈上，然后跳转到使用 Rust 编写的 `trap_handler` 函数完成 Trap 分发及处理。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。最后通过一条 `sret` 指令回到应用程序执行。

首先是保存 Trap 上下文的 `__alltraps` 的实现：

```

1 # os/src/trap/trap.S
2
3 .macro SAVE_GP n
4     sd x\n, \n*8(sp)
```

(下页继续)

(续上页)

```

5 .endm
6
7 .align 2
8 __alltraps:
9   csrrw sp, sscratch, sp
10  # now sp->kernel stack, sscratch->user stack
11  # allocate a TrapContext on kernel stack
12  addi sp, sp, -34*8
13  # save general-purpose registers
14  sd x1, 1*8(sp)
15  # skip sp(x2), we will save it later
16  sd x3, 3*8(sp)
17  # skip tp(x4), application does not use it
18  # save x5~x31
19  .set n, 5
20  .rept 27
21    SAVE_GP %n
22    .set n, n+1
23  .endr
24  # we can use t0/t1/t2 freely, because they were saved on kernel stack
25  csrr t0, sstatus
26  csrr t1, sepc
27  sd t0, 32*8(sp)
28  sd t1, 33*8(sp)
29  # read user stack from sscratch and save it on the kernel stack
30  csrr t2, sscratch
31  sd t2, 2*8(sp)
32  # set input argument of trap_handler(cx: &mut TrapContext)
33  mv a0, sp
34  call trap_handler

```

- 第 7 行我们使用 .align 将 __alltraps 的地址 4 字节对齐，这是 RISC-V 特权级规范的要求；
- 第 9 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（原因稍后说明），现在 sp 指向内核栈，sscratch 指向用户栈。
- 第 12 行，我们准备在内核栈上保存 Trap 上下文，于是预先分配 34×8 字节的栈帧，这里改动的是 sp，说明确实是内核栈上。
- 第 13-24 行，保存 Trap 上下文的通用寄存器 x0~x31，跳过 x0 和 tp(x4)，原因之前已经说明。我们在这里也不保存 sp(x2)，因为我们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 $[sp, sp + 8 \times 34]$ ，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。因此通用寄存器 xn 应该被保存在地址区间 $[sp + 8n, sp + 8(n + 1)]$ 。

为了简化代码，x5~x31 这 27 个通用寄存器我们通过类似循环的 .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 25-28 行，我们将 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。这里我们不用担心 t0 和 t1 被覆盖，因为它们刚刚已经被保存了。
- 第 30-31 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。
- 第 33 行令 $a_0 \leftarrow sp$ ，让寄存器 a0 指向内核栈的栈指针也就是我们刚刚保存的 Trap 上下文的地址，这是由于我们接下来要调用 trap_handler 进行 Trap 处理，它的第一个参数 cx 由调用规范要从 a0 中获取。而 Trap 处理函数 trap_handler 需要 Trap 上下文的原因在于：它需要知道其中某些寄存器的

值，比如在系统调用的时候应用程序传过来的 syscall ID 和对应参数。我们不能直接使用这些寄存器现在的值，因为它们可能已经被修改了，因此要去内核栈上找已经被保存下来的值。

注解：CSR 相关原子指令

RISC-V 中读写 CSR 的指令是一类能不会被打断地完成多个读写操作的指令。这种不会被打断地完成多个操作的指令被称为 **原子指令** (Atomic Instruction)。这里的 **原子** 的含义是“不可分割的最小个体”，也就是说指令的多个操作要么都不完成，要么全部完成，而不会处于某种中间状态。

另外，RISC-V 架构中常规的数据处理和访存类指令只能操作通用寄存器而不能操作 CSR。因此，当想要对 CSR 进行操作时，需要先使用读取 CSR 的指令将 CSR 读到一个通用寄存器中，而后操作该通用寄存器，最后再使用写入 CSR 的指令将该通用寄存器的值写入到 CSR 中。

当 `trap_handler` 返回之后会从调用 `trap_handler` 的下一条指令开始执行，也就是从栈上的 Trap 上下文恢复的 `__restore`：

```

1  # os/src/trap/trap.S

2
3  .macro LOAD_GP n
4      ld x\n, \n*8(sp)
5  .endm

6
7  __restore:
8      # case1: start running app by __restore
9      # case2: back to U after handling trap
10     mv sp, a0
11     # now sp->kernel stack(after allocated), sscratch->user stack
12     # restore sstatus/sepc
13     ld t0, 32*8(sp)
14     ld t1, 33*8(sp)
15     ld t2, 2*8(sp)
16     csrw sstatus, t0
17     csrw sepc, t1
18     csrw sscratch, t2
19     # restore general-purpose registers except sp/tp
20     ld x1, 1*8(sp)
21     ld x3, 3*8(sp)
22     .set n, 5
23     .rept 27
24         LOAD_GP %n
25         .set n, n+1
26     .endr
27     # release TrapContext on kernel stack
28     addi sp, sp, 34*8
29     # now sp->kernel stack, sscratch->user stack
30     csrrw sp, sscratch, sp
31     sret

```

- 第 10 行比较奇怪我们暂且不管，假设它从未发生，那么 sp 仍然指向内核栈的栈顶。
- 第 13~26 行负责从内核栈顶的 Trap 上下文恢复通用寄存器和 CSR。注意我们要先恢复 CSR 再恢复通用寄存器，这样我们使用的三个临时寄存器才能被正确恢复。
- 在第 28 行之前，sp 指向保存了 Trap 上下文之后的内核栈栈顶，sscratch 指向用户栈栈顶。我们在第 28 行在内核栈上回收 Trap 上下文所占用的内存，回归进入 Trap 之前的内核栈栈顶。第 30 行，再次交换 sscratch 和 sp，现在 sp 重新指向用户栈栈顶，sscratch 也依然保存进入 Trap 之前的状态并指向内核栈栈顶。

- 在应用程序控制流状态被还原之后，第 31 行我们使用 `sret` 指令回到 U 特权级继续运行应用程序控制流。

注解: `sscratch` CSR 的用途

在特权级切换的时候，我们需要将 Trap 上下文保存在内核栈上，因此需要一个寄存器暂存内核栈地址，并以它作为地址指针来依次保存 Trap 上下文的内容。但是所有的通用寄存器都不能够用作地址指针，因为它们都需要被保存，如果覆盖掉它们，就会影响后续应用控制流的执行。

事实上我们缺少了一个重要的中转寄存器，而 `sscratch` CSR 正是为此而生。从上面的汇编代码中可以看出，在保存 Trap 上下文的时候，它起到了两个作用：首先是保存了内核栈的地址，其次它可作为一个中转站让 `sp`（目前指向的用户栈的地址）的值可以暂时保存在 `sscratch`。这样仅需一条 `csrrw sp, sscratch, sp` 指令（交换对 `sp` 和 `sscratch` 两个寄存器内容）就完成了从用户栈到内核栈的切换，这是一种极其精巧的实现。

Trap 分发与处理

Trap 在使用 Rust 实现的 `trap_handler` 函数中完成分发和处理：

```

1 // os/src/trap/mod.rs
2
3 #[no_mangle]
4 pub fn trap_handler(cx: &mut TrapContext) -> &mut TrapContext {
5     let scause = scause::read();
6     let stval = stval::read();
7     match scause.cause() {
8         Trap::Exception(Exception::UserEnvCall) => {
9             cx.sepc += 4;
10            cx.x[10] = syscall(cx.x[17], [cx.x[10], cx.x[11], cx.x[12]]) as usize;
11        }
12        Trap::Exception(Exception::StoreFault) |
13        Trap::Exception(Exception::StorePageFault) => {
14            println!("[kernel] PageFault in application, kernel killed it.");
15            run_next_app();
16        }
17        Trap::Exception(Exception::IllegalInstruction) => {
18            println!("[kernel] IllegalInstruction in application, kernel killed it.");
19            run_next_app();
20        }
21        _ => {
22            panic!("Unsupported trap {:?}", stval = {:#x}!, scause.cause(), stval);
23        }
24    }
25    cx
26 }
```

- 第 4 行声明返回值为 `&mut TrapContext` 并在第 25 行实际将传入的 Trap 上下文 `cx` 原样返回，因此在 `__restore` 的时候 `a0` 寄存器在调用 `trap_handler` 前后并没有发生变化，仍然指向分配 Trap 上下文之后的内核栈栈顶，和此时 `sp` 的值相同，这里的 `sp ← a0` 并不会有问题是；
- 第 7 行根据 `scause` 寄存器所保存的 Trap 的原因进行分发处理。这里我们无需手动操作这些 CSR，而是使用 Rust 的 riscv 库来更加方便的做这些事情。要引入 riscv 库，我们需要：

```
# os/Cargo.toml
```

(下页继续)

(续上页)

[dependencies]

```
riscv = { git = "https://github.com/rcore-os/riscv", features = ["inline-asm"] }
```

- 第 8~11 行, 发现触发 Trap 的原因是来自 U 特权级的 Environment Call, 也就是系统调用。这里我们首先修改保存在内核栈上的 Trap 上下文里面 sepc, 让其增加 4。这是因为我们知道这是一个由 ecall 指令触发的系统调用, 在进入 Trap 的时候, 硬件会将 sepc 设置为这条 ecall 指令所在的地址 (因为它是进入 Trap 之前最后一条执行的指令)。而在 Trap 返回之后, 我们希望应用程序控制流从 ecall 的下一条指令开始执行。因此我们只需修改 Trap 上下文里面的 sepc, 让它增加 ecall 指令的码长, 也即 4 字节。这样在 __restore 的时候 sepc 在恢复之后就会指向 ecall 的下一条指令, 并在 sret 之后从那里开始执行。

用来保存系统调用返回值的 a0 寄存器也会同样发生变化。我们从 Trap 上下文取出作为 syscall ID 的 a7 和系统调用的三个参数 a0~a2 传给 syscall 函数并获取返回值。syscall 函数是在 syscall 子模块中实现的。这段代码是处理正常系统调用的控制逻辑。

- 第 12~20 行, 分别处理应用程序出现访存错误和非法指令错误的情形。此时需要打印错误信息并调用 run_next_app 直接切换并运行下一个应用程序。
- 第 21 行开始, 当遇到目前还不支持的 Trap 类型的时候, “邓式鱼” 批处理操作系统整个 panic 报错退出。

实现系统调用功能

对于系统调用而言, syscall 函数并不会实际处理系统调用, 而只是根据 syscall ID 分发到具体的处理函数:

```
1 // os/src/syscall/mod.rs
2
3 pub fn syscall(syscall_id: usize, args: [usize; 3]) -> isize {
4     match syscall_id {
5         SYSCALL_WRITE => sys_write(args[0], args[1] as *const u8, args[2]),
6         SYSCALL_EXIT => sys_exit(args[0] as i32),
7         _ => panic!("Unsupported syscall_id: {}", syscall_id),
8     }
9 }
```

这里我们会将传进来的参数 args 转化成能够被具体的系统调用处理函数接受的类型。它们的实现都非常简单:

```
1 // os/src/syscall/fs.rs
2
3 const FD_STDOUT: usize = 1;
4
5 pub fn sys_write(fd: usize, buf: *const u8, len: usize) -> isize {
6     match fd {
7         FD_STDOUT => {
8             let slice = unsafe { core::slice::from_raw_parts(buf, len) };
9             let str = core::str::from_utf8(slice).unwrap();
10            print!("{}", str);
11            len as isize
12        },
13        _ => {
14            panic!("Unsupported fd in sys_write!");
15        }
16    }
17 }
```

(下页继续)

(续上页)

```

18 // os/src/syscall/process.rs
19
20 pub fn sys_exit(xstate: i32) -> ! {
21     println!("[kernel] Application exited with code {}", xstate);
22     run_next_app()
23 }
24

```

- `sys_write` 我们将传入的位于应用程序内的缓冲区的开始地址和长度转化为一个字符串 `&str`，然后使用批处理操作系统已经实现的 `print!` 宏打印出来。注意这里我们并没有检查传入参数的安全性，即使会在出错严重的时候 `panic`，还是会存在安全隐患。这里我们出于实现方便暂且不做修补。
- `sys_exit` 打印退出的应用程序的返回值并同样调用 `run_next_app` 切换到下一个应用程序。

3.5.6 执行应用程序

当批处理操作系统初始化完成，或者是某个应用程序运行结束或出错的时候，我们要调用 `run_next_app` 函数切换到下一个应用程序。此时 CPU 运行在 S 特权级，而它希望能够切换到 U 特权级。在 RISC-V 架构中，唯一一种能够使得 CPU 特权级下降的方法就是执行 Trap 返回的特权指令，如 `sret`、`mret` 等。事实上，在从操作系统内核返回到运行应用程序之前，要完成如下这些工作：

- 构造应用程序开始执行所需的 Trap 上下文；
- 通过 `__restore` 函数，从刚构造的 Trap 上下文中，恢复应用程序执行的部分寄存器；
- 设置 `sepc` CSR 的内容为应用程序入口点 `0x80400000`；
- 切换 `scratch` 和 `sp` 寄存器，设置 `sp` 指向应用程序用户栈；
- 执行 `sret` 从 S 特权级切换到 U 特权级。

它们可以通过复用 `__restore` 的代码来更容易的实现上述工作。我们只需要在内核栈上压入一个为启动应用程序而特殊构造的 Trap 上下文，再通过 `__restore` 函数，就能让这些寄存器到达启动应用程序所需要的上下文状态。

```

1 // os/src/trap/context.rs
2
3 impl TrapContext {
4     pub fn set_sp(&mut self, sp: usize) { self.x[2] = sp; }
5     pub fn app_init_context(entry: usize, sp: usize) -> Self {
6         let mut sstatus = sstatus::read();
7         sstatus.set_spp(SPP::User);
8         let mut cx = Self {
9             x: [0; 32],
10            sstatus,
11            sepc: entry,
12        };
13        cx.set_sp(sp);
14        cx
15    }
16 }

```

为 `TrapContext` 实现 `app_init_context` 方法，修改其中的 `sepc` 寄存器为应用程序入口点 `entry`，`sp` 寄存器为我们设定的一个栈指针，并将 `sstatus` 寄存器的 `SPP` 字段设置为 `User`。

在 `run_next_app` 函数中我们能够看到：

```

1 // os/src/batch.rs
2
3 pub fn run_next_app() -> ! {
4     let mut app_manager = APP_MANAGER.exclusive_access();
5     let current_app = app_manager.get_current_app();
6     unsafe {
7         app_manager.load_app(current_app);
8     }
9     app_manager.move_to_next_app();
10    drop(app_manager);
11    // before this we have to drop local variables related to resources manually
12    // and release the resources
13    extern "C" { fn __restore(cx_addr: usize); }
14    unsafe {
15        __restore(KERNEL_STACK.push_context(
16            TrapContext::app_init_context(APP_BASE_ADDRESS, USER_STACK.get_sp())
17        ) as *const _ as usize);
18    }
19    panic!("Unreachable in batch::run_current_app!");
20}

```

在高亮行所做的事是在内核栈上压入一个 Trap 上下文，其 `sepc` 是应用程序入口地址 `0x80400000`，其 `sp` 寄存器指向用户栈，其 `sstatus` 的 `SPP` 字段被设置为 `User`。`push_context` 的返回值是内核栈压入 Trap 上下文之后的栈顶，它会被作为 `__restore` 的参数（回看 [__restore 代码](#)，这时我们可以理解为何 `__restore` 函数的起始部分会完成 $sp \leftarrow a_0$ ），这使得在 `__restore` 函数中 `sp` 仍然可以指向内核栈的栈顶。这之后，就和执行一次普通的 `__restore` 函数调用一样了。

注解：有兴趣的同学可以思考：`sscratch` 是何时被设置为内核栈顶的？

3.6 练习

3.6.1 课后练习

编程题

1. *** 实现一个裸机应用程序 A，能打印调用栈。
2. ** 扩展内核，实现新系统调用 `get_taskinfo`，能显示当前 task 的 id 和 task name；实现一个裸机应用程序 B，能访问 `get_taskinfo` 系统调用。
3. ** 扩展内核，能够统计多个应用的执行过程中系统调用编号和访问此系统调用的次数。
4. ** 扩展内核，能够统计每个应用执行后的完成时间。
5. *** 扩展内核，统计执行异常的程序的异常情况（主要是各种特权级涉及的异常），能够打印异常程序的出错的地址和指令等信息。

注：上述编程基于 rcore/ucore tutorial v3: Branch ch2

问答题

1. * 函数调用与系统调用有何区别?
2. ** 为了方便操作系统处理, M态软件会将 S态异常/中断委托给 S态软件, 请指出有哪些寄存器记录了委托信息, rustsbi 委托了哪些异常/中断? (也可以直接给出寄存器的值)
3. ** 如果操作系统以应用程序库的形式存在, 应用程序可以通过哪些方式破坏操作系统?
4. ** 编译器/操作系统/处理器如何合作, 可采用哪些方法来保护操作系统不受应用程序的破坏?
5. ** RISC-V 处理器的 S态特权指令有哪些, 其大致含义是什么, 有啥作用?
6. ** RISC-V 处理器在用户态执行特权指令后的硬件层面的处理过程是什么?
7. ** 操作系统在完成用户态 <-> 内核态双向切换中的一般处理过程是什么?
8. ** 程序陷入内核的原因有中断、异常和陷入 (系统调用), 请问 riscv64 支持哪些中断 / 异常? 如何判断进入内核是由于中断还是异常? 描述陷入内核时的几个重要寄存器及其值。
9. * 在哪些情况下会出现特权级切换: 用户态-> 内核态, 以及内核态-> 用户态?
10. ** Trap 上下文的含义是啥? 在本章的操作系统中, Trap 上下文的具体内容是啥? 如果不进行 Trap 上下文的保存于恢复, 会出现什么情况?

3.6.2 实验练习

实验练习包括实践作业和问答作业两部分。

实践作业

sys_write 安全检查

ch2 中, 我们实现了第一个系统调用 `sys_write`, 这使得我们可以在用户态输出信息。但是 os 在提供服务的同时, 还有保护 os 本身以及其他用户程序不受错误或者恶意程序破坏的功能。

由于还没有实现虚拟内存, 我们可以在用户程序中指定一个属于其他程序字符串, 并将它输出, 这显然是不合理的, 因此我们要对 `sys_write` 做检查:

- `sys_write` 仅能输出位于程序本身内存空间内的数据, 否则报错。

实验要求

- 实现分支: ch2-lab
- 目录要求不变
- 为 `sys_write` 增加安全检查

在 os 目录下执行 `make run TEST=1` 测试 `sys_write` 安全检查的实现, 正确执行目标用户测例, 并得到预期输出 (详见测例注释)。

注意: 如果设置默认 log 等级, 从 lab2 开始关闭所有 log 输出。

challenge: 支持多核, 实现多个核运行用户程序。

实验约定

在第二章的测试中，我们对于内核有如下仅仅为了测试方便的要求，请调整你的内核代码来符合这些要求。

- 用户栈大小必须为 4096，且按照 4096 字节对齐。这一规定可以在实验 4 开始删除，仅仅为通过 lab2/3 测例设置。

注解：如何快速继承上一章练习题的修改

从这一章开始，在完成本章习题之前，首先要做的就是将上一章框架的修改继承到本章的框架代码。出于各种原因，实际上通过 git merge 并不是很方便，这里给出一种打 patch 的方法，希望能够有所帮助。

1. 切换到上一章的分支，通过 git log 找到你在此分支上的第一次 commit 的前一个 commit 的 ID，复制其前 8 位，记作 base-commit。假设分支上最新的一次 commit ID 是 last-commit。
2. 确保你位于项目根目录 rCore-Tutorial-v3 下。通过 git diff <base-commit> <last-commit> > <patch-path> 即可在 patch-path 路径位置（比如 ~/Desktop/chx.patch）生成一个描述你对于上一章分支进行的全部修改的一个补丁文件。打开看一下，它给出了每个被修改的文件中涉及了哪些块的修改，还附加了块前后的若干行代码。如果想更加灵活进行合并的话，可以通过 git format-patch <base-commit> 命令在当前目录下生成一组补丁，它会对于 base-commit 后面的每一次 commit 均按照顺序生成一个补丁。
3. 切换到本章分支，通过 git apply --reject <patch-path> 来将一个补丁打到当前章节上。它的大概原理是对于补丁中的每个被修改文件中的每个修改块，尝试通过块的前后若干行代码来定位它在当前分支上的位置并进行替换。有一些块可能无法匹配，此时会生成与这些块所在的文件同名的 *.rej 文件，描述了哪些块替换失败了。在项目根目录 rCore-Tutorial-v3 下，可以通过 find . -name *.rej 来找到所有相关的 *.rej 文件并手动完成替换。
4. 在处理完所有 *.rej 之后，将它们删除并 commit 一下。现在就可以开始本章的实验了。

问答作业

1. 正确进入 U 态后，程序的特征还应有：使用 S 态特权指令，访问 S 态寄存器后会报错。请自行测试这些内容（运行 Rust 三个 bad 测例），描述程序出错行为，注明你使用的 sbi 及其版本。
2. 请结合用例理解 trap.S 中两个函数 __alltraps 和 __restore 的作用，并回答如下几个问题：
 1. L40：刚进入 __restore 时，a0 代表了什么值。请指出 __restore 的两种使用情景。
 2. L46-L51：这几行汇编代码特殊处理了哪些寄存器？这些寄存器的值对于进入用户态有何意义？请分别解释。

```
ld t0, 32*8(sp)
ld t1, 33*8(sp)
ld t2, 2*8(sp)
csrw sstatus, t0
csrw sepc, t1
csrw sscratch, t2
```

3. L53-L59：为何跳过了 x2 和 x4？

```
ld x1, 1*8(sp)
ld x3, 3*8(sp)
.set n, 5
.rept 27
    LOAD_GP %n
```

(下页继续)

(续上页)

```
.set n, n+1
.endr
```

4. L63: 该指令之后, sp 和 sscratch 中的值分别有什么意义?

```
csrrw sp, sscratch, sp
```

5. __restore: 中发生状态切换在哪一条指令? 为何该指令执行之后会进入用户态?

6. L13: 该指令之后, sp 和 sscratch 中的值分别有什么意义?

```
csrrw sp, sscratch, sp
```

7. 从 U 状态进入 S 状态是哪一条指令发生的?

3. 对于任何中断, __alltraps 中都需要保存所有寄存器吗? 你有没有想到一些加速 __alltraps 的方法? 简单描述你的想法。

实验练习的提交报告要求

- 简单总结与上次实验相比本次实验你增加的东西 (控制在 5 行以内, 不要贴代码)。
- 完成问答问题。
- (optional) 你对本次实验设计及难度/工作量的看法, 以及有哪些需要改进的地方, 欢迎畅所欲言。

3.7 练习参考答案

3.7.1 课后练习

编程题

1. *** 实现一个裸机应用程序 A, 能打印调用栈。

以 rCore tutorial ch2 代码为例, 在编译选项中我们已经让编译器对所有函数调用都保存栈指针 (参考 os/.cargo/config), 因此我们可以直接从 fp 寄存器追溯调用栈:

列表 2: os/src/stack_trace.rs

```
use core::{arch::asm, ptr};

pub unsafe fn print_stack_trace() -> () {
    let mut fp: *const usize;
    asm!("mv {}, fp", out(reg) fp);

    println!("== Begin stack trace ==");
    while fp != ptr::null() {
        let saved_ra = *fp.sub(1);
        let saved_fp = *fp.sub(2);

        println!("0x{:016x}, fp = 0x{:016x}", saved_ra, saved_fp);
        fp = saved_fp as *const usize;
    }
}
```

(下页继续)

(续上页)

```
    println!("== End stack trace ==");
}
```

之后我们将其加入 `main.rs` 作为一个子模块:

列表 3: 加入 `os/src/main.rs`

```
// ...
mod syscall;
mod trap;
mod stack_trace;
// ...
```

作为一个示例, 我们可以将打印调用栈的代码加入 `panic` handler 中, 在每次 `panic` 的时候打印调用栈:

列表 4: `os/lang_items.rs`

```
use crate::sbi::shutdown;
use core::panic::PanicInfo;
use crate::stack_trace::print_stack_trace;

#[panic_handler]
fn panic(info: &PanicInfo) -> ! {
    // ...

    unsafe { print_stack_trace(); }

    shutdown()
}
```

现在, `panic` 的时候输入的信息变成了这样:

```
Panicked at src/batch.rs:68 All applications completed!
== Begin stack trace ==
0x0000000080200e12, fp = 0x0000000080205cf0
0x0000000080201bfa, fp = 0x0000000080205dd0
0x0000000080200308, fp = 0x0000000080205e00
0x0000000080201228, fp = 0x0000000080205e60
0x00000000802005b4, fp = 0x0000000080205ef0
0x0000000080200424, fp = 0x0000000000000000
== End stack trace ==
```

这里打印的两个数字, 第一个是栈帧上保存的返回地址, 第二个是保存的上一个 frame pointer。

2. ** 扩展内核, 实现新系统调用 `get_taskinfo`, 能显示当前 task 的 id 和 task name; 实现一个裸机应用程序 B, 能访问 `get_taskinfo` 系统调用。
3. ** 扩展内核, 能够统计多个应用的执行过程中系统调用编号和访问此系统调用的次数。
4. ** 扩展内核, 能够统计每个应用执行后的完成时间。
5. *** 扩展内核, 统计执行异常的程序的异常情况 (主要是各种特权级涉及的异常), 能够打印异常程序的出错的地址和指令等信息。

在 `trap.c` 中添加相关异常情况的处理:

列表 5: os/trap.c

```

void usertrap()
{
    set_kerneltrap();
    struct trapframe *trapframe = curr_proc()->trapframe;

    if ((r_sstatus() & SSTATUS_SPP) != 0)
        panic("usertrap: not from user mode");

    uint64 cause = r_scause();
    if (cause & (1ULL << 63)) {
        cause &= ~(1ULL << 63);
        switch (cause) {
            case SupervisorTimer:
                tracef("time interrupt!\n");
                set_next_timer();
                yield();
                break;
            default:
                unknown_trap();
                break;
        }
    } else {
        switch (cause) {
            case UserEnvCall:
                trapframe->epc += 4;
                syscall();
                break;
            case StoreMisaligned:
            case StorePageFault:
            case InstructionMisaligned:
            case InstructionPageFault:
            case LoadMisaligned:
            case LoadPageFault:
                printf("%d in application, bad addr = %p, bad instruction = %p,
                ↵"
                "core dumped.\n",
                cause, r_stval(), trapframe->epc);
                exit(-2);
                break;
            case IllegalInstruction:
                printf("IllegalInstruction in application, core dumped.\n");
                exit(-3);
                break;
            default:
                unknown_trap();
                break;
        }
    }
    usertrapret();
}

```

注: 上述编程基于 rcore/ucore tutorial v3: Branch ch2

问答题

1. * 函数调用与系统调用有何区别?

- 函数调用用普通的控制流指令, 不涉及特权级的切换; 系统调用使用专门的指令 (如 RISC-V 上的 *ecall*), 会切换到内核特权级。
- 函数调用可以随意指定调用目标; 系统调用只能将控制流切换给调用操作系统内核给定的目标。

2. ** 为了方便操作系统处理, M态软件会将 S态异常/中断委托给 S态软件, 请指出有哪些寄存器记录了委托信息, rustsbi 委托了哪些异常/中断? (也可以直接给出寄存器的值)

- 两个寄存器记录了委托信息: mideleg (中断委托) 和 medeleg (异常委托)
- 参考 RustSBI 输出

```
[rustsbi] mideleg: ssoft, stimer, sext (0x222)
[rustsbi] medeleg: ima, ia, bkpt, la, sa, uecall, ipage, lpage, spage (0xb1ab)
```

可知委托了中断:

- ssoft : S-mode 软件中断
- stimer : S-mode 时钟中断
- sext : S-mode 外部中断

委托了异常:

- ima : 指令未对齐
- ia : 取指访问异常
- bkpt : 断点
- la : 读异常
- sa : 写异常
- uecall : U-mode 系统调用
- ipage : 取指 page fault
- lpage : 读 page fault
- spage : 写 page fault

3. ** 如果操作系统以应用程序库的形式存在, 应用程序可以通过哪些方式破坏操作系统?

如果操作系统以应用程序库的形式存在, 那么编译器在链接 OS 库时会把应用程序跟 OS 库链接成一个可执行文件, 两者处于同一地址空间, 这也是 LibOS (Unikernel) 架构, 此时存在如下几个破坏操作系统的方式:

- 缓冲区溢出: 应用程序可以覆盖写其合法内存边界之外的部分, 这可能会危及 OS;
- 整数溢出: 当对整数值的运算产生的值超出整数数据类型可以表示的范围时, 就会发生整数溢出, 这可能会导致 OS 出现意外行为和安全漏洞。例如, 如果允许应用程序分配大量内存, 攻击者可能会在内存分配例程中触发整数溢出, 从而可能导致缓冲区溢出或其他安全漏洞;
- 系统调用拦截: 应用程序可能会拦截或重定向系统调用, 从而可能损害 OS 的行为。例如, 攻击者可能会拦截读取敏感文件的系统调用并将其重定向到他们选择的文件, 从而可能危及 unikernel 的安全性。
- 资源耗尽: 应用程序可能会消耗内存或网络带宽等资源, 可能导致拒绝服务或其他安全漏洞。

4. ** 编译器/操作系统/处理器如何合作，可采用哪些方法来保护操作系统不受应用程序的破坏？

硬件操作系统运行在一个硬件保护的安全执行环境中，不受到应用程序的破坏；应用程序运行在另外一个无法破坏操作系统的受限执行环境中。现代 CPU 提供了很多硬件机制来保护操作系统免受恶意应用程序的破坏，包括如下几个：

- 特权级模式：处理器能够设置不同安全等级的执行环境，即用户态执行环境和内核态特权级的执行环境。处理器在执行指令前会进行特权级安全检查，如果在用户态执行环境中执行内核态特权级指令，会产生异常阻止当前非法指令的执行。
- TEE（可信执行环境）：CPU 的 TEE 能够构建一个可信的执行环境，用于抵御恶意软件或攻击，能够确保处理敏感数据的应用程序（例如移动银行和支付应用程序）的安全。
- ASLR（地址空间布局随机化）：ASLR 是 CPU 的一种随机化进程地址空间布局的安全功能，其能够随机生成进程地址空间，例如栈、共享库等关键部分的起始地址，使攻击者预测特定数据或代码的位置。

5. ** RISC-V 处理器的 S 状态特权指令有哪些，其大致含义是什么，有啥作用？

RISC-V 处理器的 S 状态特权指令有两类：指令本身属于高特权级的指令，如 sret 指令（表示从 S 模式返回到 U 模式）。指令访问了 S 模式特权级下才能访问的寄存器或内存，如表示 S 模式系统状态的控制状态寄存器 sstatus 等。如下所示：

- sret：从 S 模式返回 U 模式。如可以让位于 S 模式的驱动程序返回 U 模式。
- wfi：让 CPU 在空闲时进入等待状态，以降低 CPU 功耗。
- sfence.vma：刷新 TLB 缓存，在 U 模式下执行会尝试非法指令异常。
- 访问 S 模式 CSR 的指令：通过访问 spc/stvec/scause/sscarch/stval/sstatus/satp 等 CSR 来改变系统状态。

6. ** RISC-V 处理器在用户态执行特权指令后的硬件层面的处理过程是什么？

CPU 执行完一条指令（如 ecall）并准备从用户特权级陷入（Trap）到 S 特权级的时候，硬件会自动完成如下这些事情：

- 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。

CPU 完成 Trap 处理准备返回的时候，需要通过一条 S 特权级的特权指令 sret 来完成，这一条指令具体完成以下功能：* CPU 会将当前的特权级按照 sstatus 的 SPP 字段设置为 U 或者 S；* CPU 会跳转到 sepc 寄存器指向的那条指令，然后继续执行。

7. ** 操作系统在完成用户态 <-> 内核态双向切换中的一般处理过程是什么？

当 CPU 在用户态特权级（RISC-V 的 U 模式）运行应用程序，执行到 Trap，切换到内核态特权级（RISC-V 的 S 模式），批处理操作系统的对应代码响应 Trap，并执行系统调用服务，处理完毕后，从内核态返回到用户态应用程序继续执行后续指令。

8. ** 程序陷入内核的原因有中断、异常和陷入（系统调用），请问 riscv64 支持哪些中断 / 异常？如何判断进入内核是由于中断还是异常？描述陷入内核时的几个重要寄存器及其值。

- 具体支持的异常和中断，参见 RISC-V 特权集规范 *The RISC-V Instruction Set Manual Volume II: Privileged Architecture*。其它很多问题在这里也有答案。
- scause 的最高位，为 1 表示中断，为 0 表示异常
- 重要的寄存器：

- *scause* : 发生了具体哪个异常或中断
 - *sstatus* : 其中的一些控制为标志发生异常时的处理器状态, 如 *sstatus.SPP* 表示发生异常时处理器在哪个特权级。
 - *sepc* : 发生异常或中断的时候, 将要执行但未成功执行的指令地址
 - *sval* : 值与具体异常相关, 可能是发生异常的地址, 指令等
9. * 在哪些情况下会出现特权级切换: 用户态→内核态, 以及内核态→用户态?
- 用户态→内核态: 应用程序发起系统调用; 应用程序执行出错, 需要到批处理操作系统中杀死该应用并加载运行下一个应用; 应用程序执行结束, 需要到批处理操作系统中加载运行下一个应用。
 - 内核态→用户态: 启动应用程序需要初始化应用程序的用户态上下文时; 应用程序发起的系统调用执行完毕返回应用程序时。
10. ** Trap 上下文的含义是啥? 在本章的操作系统中, Trap 上下文的具体内容是啥? 如果不进行 Trap 上下文的保存与恢复, 会出现什么情况?
- Trap 上下文的主要有两部分含义:
- 在触发 Trap 之前 CPU 运行在哪个特权级;
 - CPU 需要切换到哪个特权级来处理该 Trap, 并在处理完成之后返回原特权级。在本章的实际操作系统中, Trap 上下文的具体内容主要包括通用寄存器和栈两部分。如果不进行 Trap 的上下文保存与恢复, CPU 就无法在处理完成之后, 返回原特权级。

3.7.2 实验练习

问答作业

1. 正确进入 U 态后, 程序的特征还应有: 使用 S 态特权指令, 访问 S 态寄存器后会报错。请自行测试这些内容(运行 Rust 三个 bad 测例), 描述程序出错行为, 注明你使用的 sbi 及其版本。
2. 请结合用例理解 `trap.S` 中两个函数 `__alltraps` 和 `__restore` 的作用, 并回答如下几个问题:
 1. L40: 刚进入 `__restore` 时, `a0` 代表了什么值。请指出 `__restore` 的两种使用情景。
 2. L46-L51: 这几行汇编代码特殊处理了哪些寄存器? 这些寄存器的值对于进入用户态有何意义? 请分别解释。

```
ld t0, 32*8(sp)
ld t1, 33*8(sp)
ld t2, 2*8(sp)
csrw sstatus, t0
csrw sepc, t1
csrw sscratch, t2
```

3. L53-L59: 为何跳过了 `x2` 和 `x4`?

```
ld x1, 1*8(sp)
ld x3, 3*8(sp)
.set n, 5
.rept 27
  LOAD_GP %n
  .set n, n+1
.endr
```

4. L63: 该指令之后, `sp` 和 `sscratch` 中的值分别有什么意义?

```
csrrw sp, sscratch, sp
```

5. __restore: 中发生状态切换在哪一条指令? 为何该指令执行之后会进入用户态?
 6. L13: 该指令之后, sp 和 sscratch 中的值分别有什么意义?
- ```
csrrw sp, sscratch, sp
```
7. 从 U 状态进入 S 状态是哪一条指令发生的?
  3. 对于任何中断, \_\_alltraps 中都需要保存所有寄存器吗? 你有没有想到一些加速 \_\_alltraps 的方法? 简单描述你的想法。



## 第三章：多道程序与分时多任务

---

### 4.1 引言

#### 4.1.1 本章导读

提高系统的性能和效率是操作系统的核心目标之一，本章展现了操作系统在性能上的一系列功能改进：

- 通过提前加载应用程序到内存，减少应用程序切换开销
- 通过协作机制支持程序主动放弃处理器，提高系统执行效率
- 通过抢占机制支持程序被动放弃处理器，保证不同程序对处理器资源使用的公平性，也进一步提高了应用对 I/O 事件的响应效率

上一章，我们实现了一个安全的“邓式鱼”批处理操作系统。首先，它能够自动按照顺序加载并运行序列中的每一个应用，当一个应用运行结束之后无需操作员的手动替换；另一方面，在硬件级特权隔离机制的帮助下，运行在更高特权级的操作系统不会受到有意或者无意出错的应用的影响；在硬件异常触发机制的帮助下，可以全方位监控运行在用户态低特权级的应用执行，一旦应用越过了特权级界限或主动申请获得操作系统的服务，就会触发 Trap 并进入到批处理系统中进行处理。无论原因是应用出错或是应用声明自己执行完毕，批处理系统都只需要加载应用序列中的下一个应用并让其执行。可以看到批处理系统的特性是：在内存中同一时间最多只需驻留一个应用。这是因为只有当一个应用出错或退出之后，批处理系统才会去将另一个应用加载到相同的一块内存区域。

而计算机硬件在快速发展，内存容量在逐渐增大，处理器的速度也在增加，外设 I/O 性能方面的进展不大。这就使得以往内存只能放下一个程序的情况得到很大改善，但处理器的空闲程度加大了。于是科学家就开始考虑在内存中尽量同时驻留多个应用，这样处理器的利用率就会提高。但只有一个程序执行完毕后或主动放弃执行，处理器才能执行另外一个程序。我们把这种运行方式称为 **多道程序 (Multiprogramming)**。

## 协作式操作系统

早期的计算机系统大部分是单处理器计算机系统，由于计算机系统很昂贵，所以是多人共用一台计算机。当处理器进一步发展后，它与 I/O 的速度差距也进一步拉大。这时计算机科学家发现，在 **多道程序**运行方式下，一个程序如果不让出处理器，其他程序是无法执行的。如果一个应用由于 I/O 操作让处理器空闲下来或让处理器忙等，那其他需要处理器资源进行计算的应用还是没法使用空闲的处理器资源。于是就想到，让应用在执行 I/O 操作或空闲时，可以主动 **释放处理器**，让其他应用继续执行。当然执行 **放弃处理器的操作**算是一种对处理器资源的直接管理，所以应用程序可以发出这样的系统调用，让操作系统来具体完成。这样的操作系统就是支持 **多道程序**或 **协作式多任务**的协作式操作系统。

## 抢占式操作系统

计算机科学家很快发现，编写应用程序的科学家（简称应用程序员）来自不同的领域，他们不一定有友好互助的意识，也不了解其它程序的执行情况，很难（也没必要）有提高整个系统利用率上的大局观。在他们的脑海里，整个计算机就应该是为他们自己的应用准备的，不用考虑其他程序的运行。这导致应用程序员在编写程序时，无法做到在程序的合适位置放置 **放弃处理器的系统调用请求**，这样系统的整体利用率还是无法提高。

所以，站在系统的层面，还是需要有一种办法能强制打断应用程序的执行，来提高整个系统的效率，让在整个系统中执行的多个程序之间占用计算机资源的情况相对公平一些。根据计算机系统的硬件设计，为提高 I/O 效率，外设可以通过硬件中断机制来与处理机进行 I/O 交互操作。这种硬件中断机制可随时打断应用程序的执行，并让操作系统来完成对外设的 I/O 响应。

而操作系统可进一步利用某种以固定时长为时间间隔的外设中断（比如时钟中断）来强制打断一个程序的执行，这样一个程序只能运行一段时间（可以简称为一个时间片，Time Slice）就一定会让出处理器，且操作系统可以在处理外设的 I/O 响应后，让不同应用程序分时占用处理器执行，并可通过统计程序占用处理器的总执行时间，来评估运行的程序对处理器资源的消耗。我们把这种运行方式称为 **分时共享（Time Sharing）** 或 **抢占式多任务（Multitasking）**，也可合并在一起称为 **分时多任务**。

我们可以把一个程序的一次完整执行过程称为一次 **任务（Task）**，把一个程序在一个时间片（Time Slice）上占用处理器执行的过程称为一个 **任务片（Task Slice）**。操作系统对不同程序的执行过程中的 **任务片**进行调度和管理，即通过平衡各个程序在整个时间段上的任务片数量，就能达到一定程度的系统公平和高效的系统效率。在一个包含多个时间片的时间段上，会有属于不同程序的多个任务片在轮流占用处理器执行，这样的操作系统就是支持 **分时多任务**或 **抢占式多任务**的抢占式操作系统。

### 注解：支持多道程序的各种操作系统

支持多道程序的第一台计算机是 1961 年的英国 Leo III 计算机，支持在计算机内存中加载几个不同的程序，并从第一个开始依次运行。当第一个程序到达等待外围设备的指令时，该程序的执行上下文被保存下来，内存中的第二个程序就有机会运行。该过程一直持续到所有程序运行完毕。但 1960 年以后，美国人已经占领了英国计算机市场的大部分，他们的机器设计得更好，更可靠，最重要的是更便宜。随着时间的推移，英国的计算机制造逐渐消失。

## 协作式操作系统

对于计算机硬件而言，中断机制很早就出现了，但利用中断机制实现抢占式多任务，会涉及到不少并发的问题，这对操作系统设计是一个比较大的挑战。所以一些相对简单的操作系统采用协作式的任务调度管理方式，如微软早期（1990~1998 年）的 Windows 3.x/9x 操作系统和苹果早期（1987~2001 年）的 Mac OS 操作系统中的 MultiFinder 扩展功能。Windows 和 Mac OS 都是有着漂亮的 GUI 界面的操作系统，它们都采用了协作多任务处理模型，这意味着只有在前台应用程序主动让出控制权时（比如，在等待键盘或鼠标输入时），才将 CPU 时间分配给后台应用程序执行。由于具有便于没有计算机知识的人上手的 GUI 界面，尽管早期 Windows 操作系统任务管理能力较弱，它仍迅速占领 PC 市场，统治了我们的桌面。

### 分时共享（Time Sharing）的本质

1959 年 1 月 1 日，John McCarthy 在 MIT 的一份未发表的备忘录中提出了分时共享最初想法：“我想提出一个在 IBM 709 计算机上的操作系统的设计思路，它将大大减少用户/程序员在机器上求解问题的时间…”。能

够让计算机以可承受的成本提供快速响应的唯一方法是通过分时共享。也就是说，当一个用户对某些输出进行处理时，计算机必须去照顾到其他用户。我认为该提案指出了未来所有计算机的操作方式，我们有机会在计算机使用方式上开拓出一大步。”

在 1961 年的春天，John McCarthy 进一步解释了他的远见卓识：“我所说的分时计算机系统是指通过一些远程控制台与许多用户同步互动的计算机系统。这样一个系统对每个用户来说就像一台大型的私人电脑。当用户需要服务时，他只需开始输入请求服务的信息。计算机随时准备关注他可能敲击的任何键。因为程序可能…在人与人之间的互动中只做相对较短的工作，所以必须不断地在二级存储中来回穿梭，这是不经济的做法。因此，需要有一个大的主存储器。…最后的要求是二级存储要足够大，以维持用户的文件，这样用户就不需要有单独的卡或磁带输入输出单元。”

从 John McCarthy 的文章中，我们可以看到，分时共享的核心目标是让多个用户共享使用一台计算机，且每个用户感觉自己是在独占这台强大的计算机。为达到这一目标，需要计算机硬件和操作系统充分协作，达到两个要素：及时响应和快速执行。首先，计算机要有硬件中断机制，这样在必要的时候（用户敲键盘或时间片到了），硬件能通过中断打断当前执行的应用程序，把控制权交给操作系统，让操作系统切换到需要及时响应用户需求的应用程序。其次，计算机要有足够大的主存，这样应用程序都能放到主存中，这是实现快速保存/恢复程序执行状态并能快速持续执行程序的重要条件之一。

### 抢占式操作系统：支持现代云计算的古老 Compatible Time-Sharing System (CTSS) 操作系统

抢占式操作系统的典型代表是支持分时多任务的操作系统。CTSS 操作系统是 MIT 的 Fernando J. Corbató 带领团队开发的第一个分时操作系统。CTSS 于 1961 年 11 月首次在 MIT 的 IBM 709 上演示，在 1963-1973 年期间，为麻省理工学院用户提供服务。CTSS 的“兼容性”是指兼容以往的批处理运行方式，即在同一台计算机上运行后台作业，这通常比分时功能使用更多的计算资源。

CTSS 操作系统包括一个位于处理器保护模式（内核态）中的内核，内核中的监管功能只能通过软件中断（即系统调用接口）来调用，这与运行在内核态的现代操作系统一样。处理器分配调度算法是多级反馈队列算法，调度的时间片单位为 200 ms。硬件产生时钟中断后会跳转到 CTSS 操作系统内核中进行进一步处理（比如进行任务切换）。

CTSS 操作系统的实现允许一台昂贵的大型机被大量不同用户群分时并发使用，且每个用户感觉上是在独占这台强大的机器，这在当时引起了广泛的关注，并影响了后续一系列操作系统的发展，现在的操作系统基本上都支持分时共享的基础功能。CTSS 是操作系统又一个革命性的突破。

本章所介绍的多道程序和分时多任务系统都有一些共同的特点：在内存中同一时间可以驻留多个应用，而且所有的应用都是在系统启动的时候分别加载到内存的不同区域中。由于目前计算机系统中只有一个处理器核，所以同一时间最多只有一个应用在执行（即处于运行状态），剩下的应用处于就绪状态或等待状态，需要内核将处理器分配给它们才能开始执行。一旦应用开始执行，它就处于运行状态了。

本章主要是设计和实现建立支持 **多道程序** 的二叠纪“锯齿螈”<sup>1</sup> 初级操作系统、支持 **多道程序** 的三叠纪“始初龙”<sup>2</sup> 协作式操作系统和支持 **分时多任务** 的三叠纪“腔骨龙”<sup>3</sup> 抢占式操作系统，从而对可支持运行一批应用程序的多种执行环境有一个全面和深入的理解，并可归纳抽象出 **任务**、**任务切换** 等操作系统的概念。

**提示：**同学也许会有疑问：由于只有一个处理器，即使这样做，同一时间最多还是只能运行一个应用，还浪费了更多的内存来把所有的应用都加载进来。那么这样做有什么意义呢？

同学可以带着这个问题继续看下去。后面我们会介绍这样做到底能够解决什么问题。

---

### 提示：批处理与多道程序的区别是什么？

对于批处理系统而言，它在一段时间内可以处理一批程序，但内存中只放一个程序，处理器一次只能运行一个程序，只有在一个程序运行完毕后再把另外一个程序调入内存，并执行。即批处理系统不能交错执行多个程序。

<sup>1</sup> 锯齿螈身长可达 9 米，是迄今出现过的最大的两栖动物，是二叠纪时期江河湖泊和沼泽中的顶级掠食者。

<sup>2</sup> 始初龙（也称始盗龙）是后三叠纪时期的两足食肉动物，也是目前所知最早的恐龙，它们只有一米长，却代表着恐龙的黎明。

<sup>3</sup> 腔骨龙（也称虚形龙）最早出现于三叠纪晚期，它体形纤细，善于奔跑，以小型动物为食。

对于支持多道程序的系统而言，它在一段时间内也可以处理一批程序，但内存中可以放多个程序，一个程序在执行过程中，可以主动（协作式）或被动（抢占式）地放弃自己的执行，让另外一个程序执行。即支持多道程序的系统可以交错地执行多个程序，这样系统的利用率会更高。

### 4.1.2 实践体验

**多道程序** (Multiprogramming) 和 **分时多任务** (Time-Sharing Multitasking) 对于应用的要求是不同的，因此我们分别为它们编写了不同的应用，代码也被放在两个不同的分支上。对于它们更加深入的讲解请参考本章正文，我们在引言中仅给出运行代码的方法。

获取多道程序的代码：

```
$ git clone https://github.com/rcore-os/rCore-Tutorial-v3.git
$ cd rCore-Tutorial-v3
$ git checkout ch3-coop
```

获取分时多任务系统的代码：

```
$ git clone https://github.com/rcore-os/rCore-Tutorial-v3.git
$ cd rCore-Tutorial-v3
$ git checkout ch3
```

在 qemu 模拟器上运行本章代码：

```
$ cd os
$ make run
```

多道程序的应用分别会输出一个不同的字母矩阵。当他们交替执行的时候，我们将看到字母行的交错输出：

```
[RustSBI output]
[kernel] Hello, world!
AAAAAAA [1/5]
BBBBBBBBB [1/2]
CCCCCCCCC [1/3]
AAAAAAA [2/5]
BBBBBBBBB [2/2]
CCCCCCCCC [2/3]
AAAAAAA [3/5]
Test write_b OK!
[kernel] Application exited with code 0
CCCCCCCCC [3/3]
AAAAAAA [4/5]
Test write_c OK!
[kernel] Application exited with code 0
AAAAAAA [5/5]
Test write_a OK!
[kernel] Application exited with code 0
[kernel] Panicked at src/task/mod.rs:106 All applications completed!
```

分时多任务系统应用分为两种。编号为 00/01/02 的应用分别会计算质数 3/5/7 的幂次对一个大质数取模的余数，并会将结果阶段性输出。编号为 03 的应用则会等待三秒钟之后再退出。以 k210 平台为例，我们将会看到 00/01/02 三个应用分段完成它们的计算任务，而应用 03 由于等待时间过长总是最后一个结束执行。

```
[RustSBI output]
[kernel] Hello, world!
```

(下页继续)

(续上页)

```

power_3 [10000/200000]
power_3 [20000/200000]
power_3 [30000/200000]power_5 [10000/140000]
power_5 [20000/140000]
power_5 [30000/140000]power_7 [10000/160000]
power_7 [20000/160000]
power_7 [30000/160000]
]
power_3 [40000/200000]
power_3 [50000/200000]
power_3 [60000/200000]
power_5 [40000/140000]
power_5 [50000/140000]
power_5 [60000/140000]power_7 [40000/160000]
power_7 [50000/160000]
power_7 [60000/160000]
]
power_3 [70000/200000]
power_3 [80000/200000]
power_3 [90000/200000]
power_5 [70000/140000]
power_5 [80000/140000]
power_5 [90000/140000]power_7 [70000/160000]
power_7 [80000/160000]
power_7 [90000/160000]
]
power_3 [100000/200000]
power_3 [110000/200000]
power_3 [120000/]
power_5 [100000/140000]
power_5 [110000/140000]
power_5 [120000/power_7 [100000/160000]
power_7 [110000/160000]
power_7 [120000/160000]200000]
power_3 [130000/200000]
power_3 [140000/200000]
power_3 [150000/140000]
power_5 [130000/140000]
power_5 [140000/140000]
5^140000 = 386471875]
power_7 [130000/160000]
power_7 [140000/160000]
power_7 [150000/160000/200000]
power_3 [160000/200000]
power_3 [170000/200000]
power_3 [
Test power_5 OK!
[kernel] Application exited with code 0
]
power_7 [160000/160000]
7180000/200000]
power_3 [190000/200000]
power_3 [200000/200000]
3^200000 = 871008973^160000 = 667897727
Test power_7 OK!
[kernel] Application exited with code 0

```

(下页继续)

(续上页)

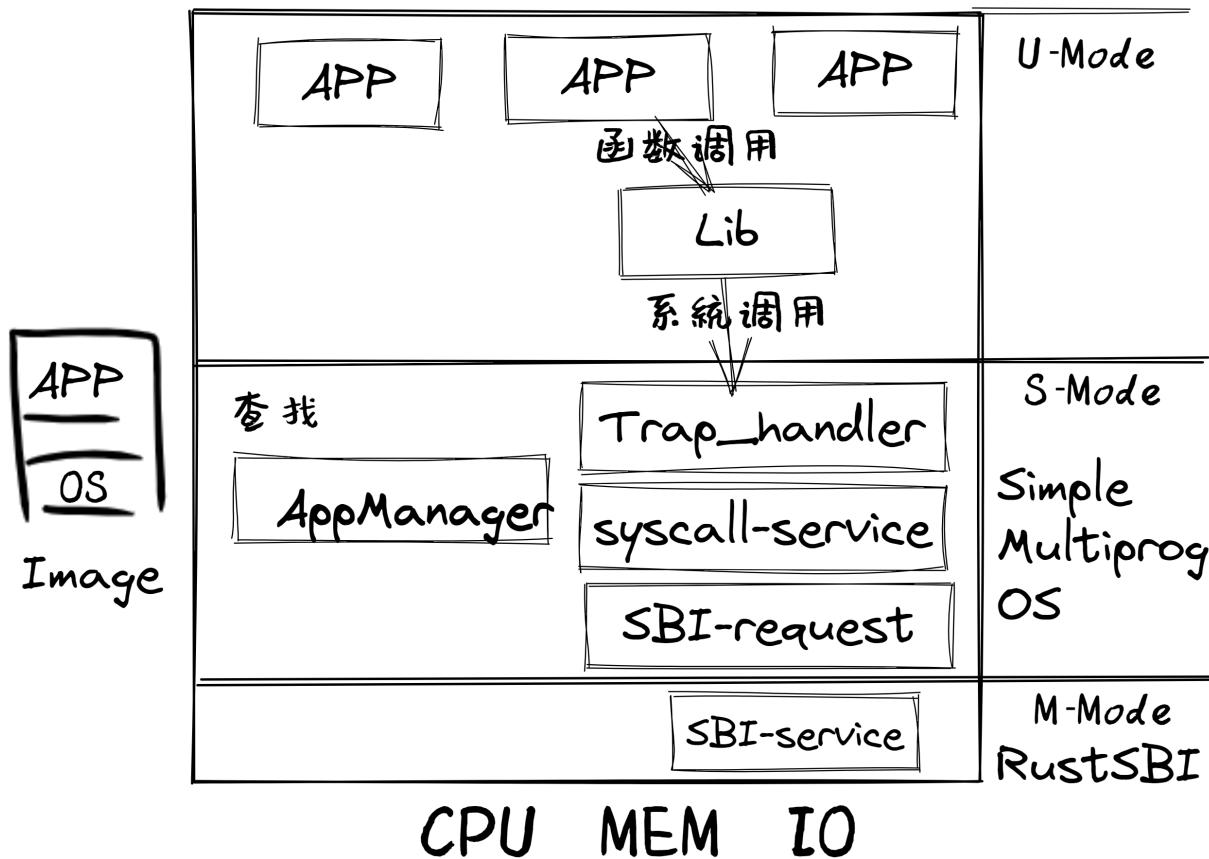
```
Test power_3 OK!
[kernel] Application exited with code 0
Test sleep OK!
[kernel] Application exited with code 0
[kernel] Panicked at src/task/mod.rs:98 All applications completed!
[rustsbi] reset triggered! todo: shutdown all harts on k210; program halt. Type: 0, ↵
 ↵reason: 0
```

输出结果看上去有一些混乱，原因是用户程序的每个 `println!` 往往会被拆分成多个 `sys_write` 系统调用提交给内核。有兴趣的同学可以参考 `println!` 宏的实现。

另外需要说明的一点是：与上一章不同，应用的编号不再决定其被加载运行的先后顺序，而仅仅能够改变应用被加载到内存中的位置。

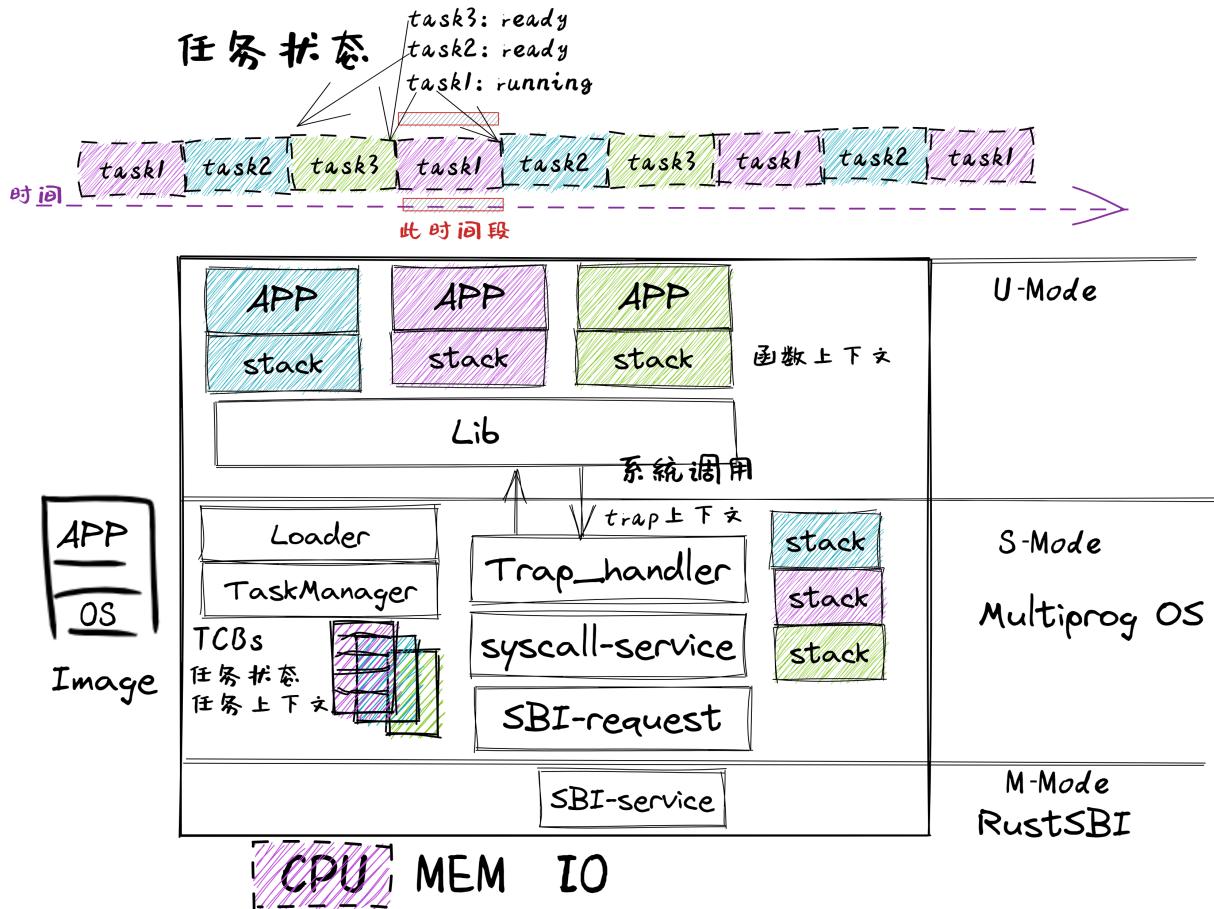
### 4.1.3 本章代码树

锯齿螺多道程序操作系统 - Multiprog OS 的总体结构如下图所示：



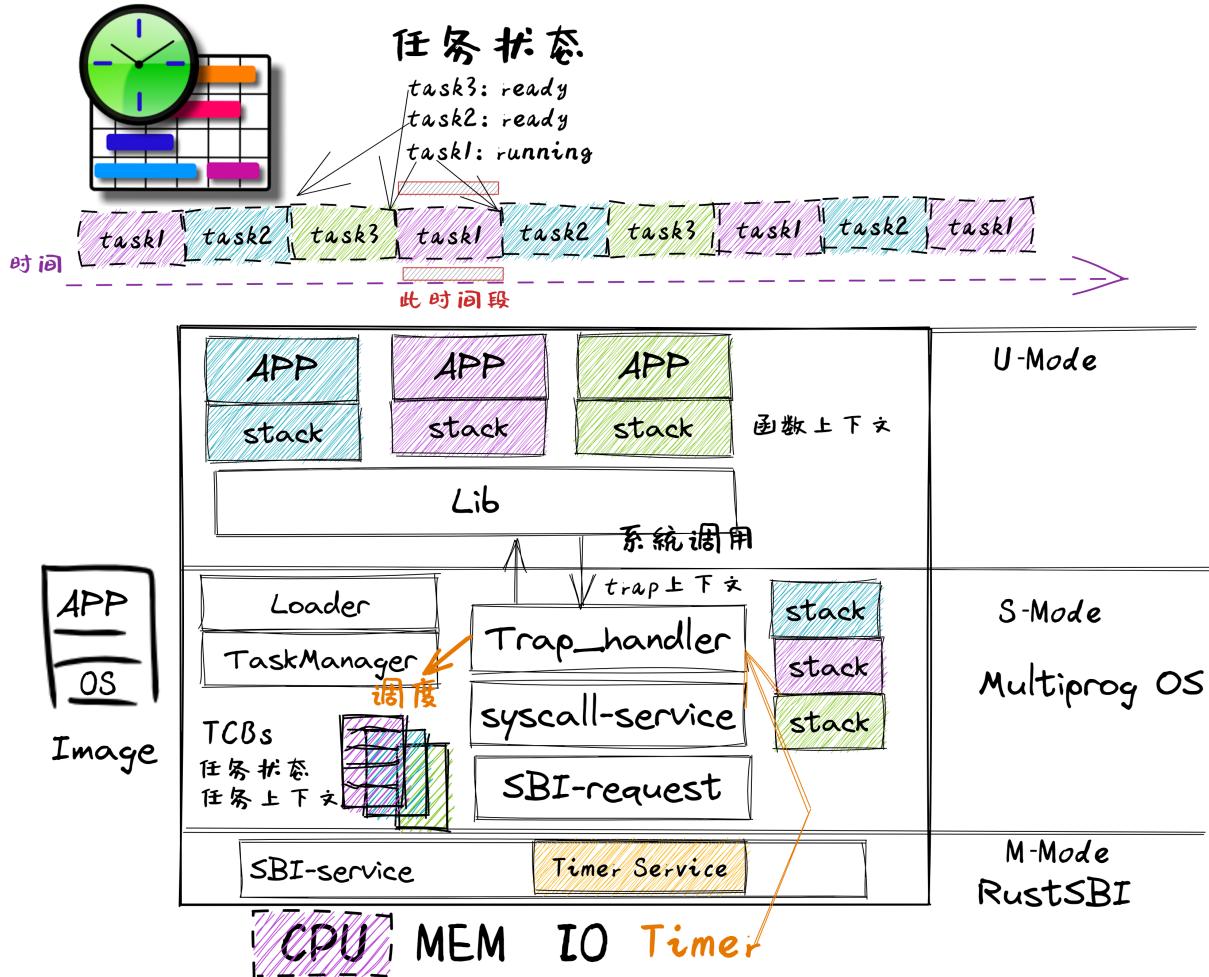
通过上图，大致可以看出 Qemu 把包含多个 app 的列表和 MultiprogOS 的 image 镜像加载到内存中，RustSBI (bootloader) 完成基本的硬件初始化后，跳转到 MultiprogOS 起始位置，MultiprogOS 首先进行正常运行前的初始化工作，即建立栈空间和清零 bss 段，然后通过改进的 AppManager 内核模块从 app 列表中把所有 app 都加载到内存中，并按指定顺序让 app 在用户态一个接一个地执行。app 在执行过程中，会通过系统调用的方式得到 MultiprogOS 提供的 OS 服务，如输出字符串等。

始初龙协作式多道程序操作系统—CoopOS 的总体结构如下图所示：



通过上图，大致可以看出相对于 MultiprogOS，CoopOS 进一步改进了 AppManager 内核模块，把它拆分为负责加载应用的 Loader 内核模块和管理应用运行过程的 TaskManager 内核模块。TaskManager 通过 task 任务控制块来管理应用程序的执行过程，支持应用程序主动放弃 CPU 并切换到另一个应用继续执行，从而提高系统整体执行效率。应用程序在运行时有自己所在的内存空间和栈，确保被切换时相关信息不会被其他应用破坏。如果当前应用程序正在运行，则该应用对应的任务处于运行（Running）状态；如果该应用主动放弃处理器，则该应用对应的任务处于就绪（Ready）状态。操作系统进行任务切换时，需要把要暂停任务的上下文（即任务用到的通用寄存器）保存起来，把要继续执行的任务的上下文恢复为暂停前的内容，这样就能让不同的应用协同使用处理器了。

腔骨龙分时多任务操作系统—TimesharingOS 的总体结构如下图所示：



通过上图，大致可以看出相对于 CoopOS，TimesharingOS 最大的变化是改进了 *Trap\_handler* 内核模块，支持时钟中断，从而可以抢占应用的执行。并通过进一步改进 *TaskManager* 内核模块，提供任务调度功能，这样可以在收到时钟中断后统计任务的使用时间片，如果任务的时间片用完后，则切换任务。从而可以公平和高效地分时执行多个应用，提高系统的整体效率。

位于 ch3 分支上的腔骨龙分时多任务操作系统—TimesharingOS 的源代码如下所示：

这里

```

1 ./os/src
2 Rust 18 Files 511 Lines
3 Assembly 3 Files 82 Lines
4
5 ├── bootloader
6 │ └── rustsbi-qemu.bin
7 └── LICENSE
8
9 └── os
10 ├── build.rs
11 ├── Cargo.toml
12 ├── Makefile
13 └── src
14 ├── batch.rs (移除：功能分别拆分到 loader 和 task 两个子模块)
15 └── config.rs (新增：保存内核的一些配置)

```

(下页继续)

(续上页)

```

15 console.rs
16 entry.asm
17 lang_items.rs
18 link_app.S
19 linker-qemu.ld
20 loader.rs(新增: 将应用加载到内存并进行管理)
21 main.rs(修改: 主函数进行了修改)
22 sbi.rs(修改: 引入新的 sbi call set_timer)
23 sync
24 mod.rs
25 up.rs
26 syscall(修改: 新增若干 syscall)
27 fs.rs
28 mod.rs
29 process.rs
30 task(新增: task 子模块, 主要负责任务管理)
31 context.rs(引入 Task 上下文 TaskContext)
32 mod.rs(全局任务管理器和提供给其他模块的接口)
33 switch.rs(将任务切换的汇编代码解释为 Rust 接口 __switch)
34 switch.S(任务切换的汇编代码)
35 task.rs(任务控制块 TaskControlBlock 和任务状态 TaskStatus 的定义)
36 timer.rs(新增: 计时器相关)
37 trap
38 context.rs
39 mod.rs(修改: 时钟中断相应处理)
40 trap.S
41 README.md
42 rust-toolchain
43 user
44 build.py(新增: 使用 build.py 构建应用使得它们占用的物理地址区间不相交)
45 Cargo.toml
46 Makefile(修改: 使用 build.py 构建应用)
47 src
48 bin(修改: 换成第三章测例)
49 00power_3.rs
50 01power_5.rs
51 02power_7.rs
52 03sleep.rs
53 console.rs
54 lang_items.rs
55 lib.rs
56 linker.ld
57 syscall.rs

```

#### 4.1.4 本章代码导读

本章的重点是实现对应用之间的协作式和抢占式任务切换的操作系统支持。与上一章的操作系统实现相比，有如下一些不同的情况导致实现上也有差异：

- 多个应用同时放在内存中，所以他们的起始地址是不同的，且地址范围不能重叠
- 应用在整个执行过程中会暂停或被抢占，即会有主动或被动的任务切换

这些实现上差异主要集中在对应用程序执行过程的管理、支持应用程序暂停的系统调用和主动切换应用程序所需的时钟中断机制的管理。

对于第一个不同情况，需要对应用程序的地址空间布局进行调整，每个应用的地址空间都不相同，且不

能重叠。这并不修改应用程序本身，而是通过一个脚本 `build.py` 来针对每个应用程序修改链接脚本 `linker.ld` 中的 `BASE_ADDRESS`，让编译器在编译不同应用时用到的 `BASE_ADDRESS` 都不同，且有足够的地址间隔。这样就可以让每个应用所在的内存空间是不同的。

对于第二个不同情况，需要实现任务切换，这就需要在上一章的 Trap 上下文切换的基础上，再加上一个 Task 上下文切换，才能完成完整的任务切换。这里面的关键数据结构是表示应用执行上下文的 `TaskContext` 数据结构和具体完成上下文切换的汇编语言编写的 `__switch` 函数。一个应用的执行需要被操作系统管理起来，这是通过 `TaskControlBlock` 数据结构来表示应用执行上下文的动态执行过程和状态（运行态、就绪态等）。而为了做好应用程序第一次执行的前期初始化准备，`TaskManager` 数据结构的全局变量实例 `TASK_MANAGER` 描述了应用程序初始化所需的数据，而对 `TASK_MANAGER` 的初始化赋值过程是实现这个准备的关键步骤。

应用程序可以在用户态执行中主动暂停，这需要有新的系统调用 `sys_yield` 的实现来支持；为了支持抢占应用执行的抢占式切换，还要添加对时钟中断的处理。有了时钟中断，就可以在确定时间间隔内打断应用的执行，并主动切换到另外一个应用，这部分主要是通过对 `trap_handler` 函数中进行扩展，来完成在时钟中断产生时可能进行的任务切换。`TaskManager` 数据结构的成员函数 `run_next_task` 来具体实现基于任务控制块的任务切换，并会具体调用 `__switch` 函数完成硬件相关部分的任务上下文切换。

如果理解了上面的数据结构和相关函数的关系和相互调用的情况，那么就比较容易理解本章改进的“锯齿螈”、“始初龙”和“腔骨龙”操作系统了。

## 4.2 多道程序放置与加载

### 4.2.1 本节导读

本节我们将实现可以把多个应用放置到内存中的二叠纪“锯齿螈”<sup>1</sup> 操作系统，“锯齿螈”能够上陆了！能实现二叠纪“锯齿螈”操作系统的一个重要前提是计算机中物理内存容量增加了，足以容纳多个应用程序的内容。在计算机的发展史上，我们也确实看到，随着集成电路的快速发展，计算机的内存容量也越来越大了。

在本章的引言中我们提到每个应用都需要按照它的编号被分别放置并加载到内存中不同的位置。本节我们就来介绍多应用的内存放置是如何实现的。通过具体实现，可以看到多个应用程序被一次性地加载到内存中，这样在切换到另外一个应用程序执行会很快，不像前一章介绍的操作系统，还要有清空前一个应用，然后加载当前应用的过程开销。

但我们也会了解到，每个应用程序需要知道自己运行时在内存中的不同位置，这对应用程序的编写带来了一定的麻烦。而且操作系统也要知道每个应用程序运行时的位置，不能任意移动应用程序所在的内存空间，即不能在运行时根据内存空间的动态空闲情况，把应用程序调整到合适的空闲空间中。这是“锯齿螈”<sup>Page 10, 1</sup> 操作系统在动态内存管理上的不足之处。

### 4.2.2 多道程序放置

与第二章相同，所有应用的 ELF 格式执行文件都经过 `objcopy` 工具丢掉所有 ELF header 和符号变为二进制镜像文件，随后以同样的格式通过在操作系统内核中嵌入 `link_user.s` 文件，在编译时直接把应用链接到内核的数据段中。不同的是，我们对相关模块进行了调整：在第二章中应用的加载和执行进度控制都交给 `batch` 子模块，而在第三章中我们将应用的加载这部分功能分离出来在 `loader` 子模块中实现，应用的执行和切换功能则交给 `task` 子模块。

注意，我们需要调整每个应用被构建时使用的链接脚本 `linker.ld` 中的起始地址 `BASE_ADDRESS`，这个地址是应用被内核加载到内存中的起始地址。也就是要做到：应用知道自己会被加载到某个地址运行，而内核也确实能做到将应用加载到它指定的那个地址。这算是应用和内核在某种意义上达成的一种协议。之所以要有这么苛刻的条件，是因为目前的操作系统内核的能力还是比较弱的，对应用程序通用性的支持也不够（比如不支持加载应用到内存中的任意地址运行），这也进一步导致了应用程序编程上不够方便和通用（应用

<sup>1</sup> 锯齿螈身长可达 9 米，是迄今出现过的最大的两栖动物，是二叠纪时期江河湖泊和沼泽中的顶级掠食者。

需要指定自己运行的内存地址)。事实上, 目前应用程序的编址方式是基于绝对位置的, 并没做到与位置无关, 内核也没有提供相应的地址重定位机制。

**注解:** 对于编址方式, 需要再回顾一下编译原理课讲解的后端代码生成技术, 以及计算机组成原理课的指令寻址方式的内容。可以在[这里](#)找到更多有关位置无关和重定位的说明。

由于每个应用被加载到的位置都不同, 也就导致它们的链接脚本 `linker.ld` 中的 `BASE_ADDRESS` 都是不同的。实际上, 我们不是直接用 `cargo build` 构建应用的链接脚本, 而是写了一个脚本定制工具 `build.py`, 为每个应用定制了各自的链接脚本:

```

1 # user/build.py
2
3 import os
4
5 base_address = 0x80400000
6 step = 0x20000
7 linker = 'src/linker.ld'
8
9 app_id = 0
10 apps = os.listdir('src/bin')
11 apps.sort()
12 for app in apps:
13 app = app[:app.find('.')]
14 lines = []
15 lines_before = []
16 with open(linker, 'r') as f:
17 for line in f.readlines():
18 lines_before.append(line)
19 line = line.replace(hex(base_address), hex(base_address+step*app_id))
20 lines.append(line)
21 with open(linker, 'w+') as f:
22 f.writelines(lines)
23 os.system('cargo build --bin %s --release' % app)
24 print('[build.py] application %s start with address %s' % (app, hex(base_
25 ↪address+step*app_id)))
26 with open(linker, 'w+') as f:
27 f.writelines(lines_before)
app_id = app_id + 1

```

它的思路很简单, 在遍历 `app` 的大循环里面只做了这样几件事情:

- 第 16-22 行, 找到 `src/linker.ld` 中的 `BASE_ADDRESS = 0x80400000`; 这一行, 并将后面的地位替换为和当前应用对应的一个地址;
- 第 23 行, 使用 `cargo build` 构建当前的应用, 注意我们可以使用 `--bin` 参数来只构建某一个应用;
- 第 25-26 行, 将 `src/linker.ld` 还原。

### 4.2.3 多道程序加载

应用的加载方式也和上一章的有所不同。上一章中讲解的加载方法是让所有应用都共享同一个固定的加载物理地址。也是因为这个原因，内存中同时最多只能驻留一个应用，当它运行完毕或者出错退出的时候由操作系统的 batch 子模块加载一个新的应用来替换掉它。本章中，所有的应用在内核初始化的时候就一并被加载到内存中。为了避免覆盖，它们自然需要被加载到不同的物理地址。这是通过调用 loader 子模块的 load\_apps 函数实现的：

```

1 // os/src/loader.rs
2
3 pub fn load_apps() {
4 extern "C" { fn _num_app(); }
5 let num_app_ptr = _num_app as usize as *const usize;
6 let num_app = get_num_app();
7 let app_start = unsafe {
8 core::slice::from_raw_parts(num_app_ptr.add(1), num_app + 1)
9 };
10 // load apps
11 for i in 0..num_app {
12 let base_i = get_base_i(i);
13 // clear region
14 (base_i..base_i + APP_SIZE_LIMIT).for_each(|addr| unsafe {
15 (addr as *mut u8).write_volatile(0)
16 });
17 // load app from data section to memory
18 let src = unsafe {
19 core::slice::from_raw_parts(
20 app_start[i] as *const u8,
21 app_start[i + 1] - app_start[i]
22)
23 };
24 let dst = unsafe {
25 core::slice::from_raw_parts_mut(base_i as *mut u8, src.len())
26 };
27 dst.copy_from_slice(src);
28 }
29 unsafe {
30 asm!("fence.i");
31 }
32}

```

可以看出，第  $i$  个应用被加载到以物理地址  $base_i$  开头的一段物理内存上，而  $base_i$  的计算方式如下：

```

1 // os/src/loader.rs
2
3 fn get_base_i(app_id: usize) -> usize {
4 APP_BASE_ADDRESS + app_id * APP_SIZE_LIMIT
5 }

```

我们可以在 config 子模块中找到这两个常数。从这一章开始，config 子模块用来存放内核中所有的常数。看到 APP\_BASE\_ADDRESS 被设置为 0x80400000，而 APP\_SIZE\_LIMIT 和上一章一样被设置为 0x20000，也就是每个应用二进制镜像的大小限制。因此，应用的内存布局就很明朗了——就是从 APP\_BASE\_ADDRESS 开始依次为每个应用预留一段空间。这样，我们就说清楚了多个应用是如何被构建和加载的。

## 4.2.4 执行应用程序

当多道程序的初始化放置工作完成，或者是某个应用程序运行结束或出错的时候，我们要调用 `run_next_app` 函数切换到下一个应用程序。此时 CPU 运行在 S 特权级的操作系统中，而操作系统希望能够切换到 U 特权级去运行应用程序。这一过程与上章的 [执行应用程序](#) 一节的描述类似。相对不同的是，操作系统知道每个应用程序预先加载在内存中的位置，这就需要设置应用程序返回的不同 Trap 上下文（Trap 上下文中保存了放置程序起始地址的 `epc` 寄存器内容）：

- 跳转到应用程序（编号  $i$ ）的入口点  $entry_i$
- 将使用的栈切换到用户栈  $stack_i$

我们的“锯齿螈”初级多道程序操作系统就算是实现完毕了。它支持把多个应用的代码和数据放置到内存中，并能够依次执行每个应用，提高了应用切换的效率，这就达到了本章对操作系统的初级需求。但“锯齿螈”操作系统在任务调度的灵活性上还有很大的改进空间，下一节我们将开始改进这方面的问题。

## 4.3 任务切换

### 4.3.1 本节导读

在上一节实现的二叠纪“锯齿螈”操作系统还是比较原始，一个应用会独占 CPU 直到它出错或主动退出。操作系统还是以程序的一次执行过程（从开始到结束）作为处理器切换程序的时间段。为了提高效率，我们需要引入新的操作系统概念 **任务**、**任务切换**、**任务上下文**。为此，我们需要实现从“螈”到“恐龙”的进化，实现“始初龙”操作系统。

如果把应用程序执行的整个过程进行进一步分析，可以看到，当程序访问 I/O 外设或睡眠时，其实是不需要占用处理器的，于是我们可以把应用程序在不同时间段的执行过程分为两类，占用处理器执行有效任务的计算阶段和不必占用处理器的等待阶段。这些阶段就形成了一个我们熟悉的“暂停-继续…”组合的控制流或执行历史。从应用程序开始执行到结束的整个控制流就是应用程序的整个执行过程。

本节的重点是操作系统的核心机制——**任务切换**，在内核中这种机制是在 `__switch` 函数中实现的。任务切换支持的场景是：一个应用在运行途中便会主动或被动交出 CPU 的使用权，此时它只能暂停执行，等到内核重新给它分配处理器资源之后才能恢复并继续执行。有了任务切换的能力，“螈”级的操作系统才能跳出水坑，进入陆地，才有能力进化到“恐龙”级的操作系统。

### 4.3.2 任务的概念形成

如果操作系统能够在某个应用程序处于等待阶段的时候，把处理器转给另外一个处于计算阶段的应用程序，那么只要转换的开销不大，那么处理器的执行效率就会大大提高。当然，这需要应用程序在运行途中能主动交出 CPU 的使用权，此时它处于等待阶段，等到操作系统让它再次执行后，那它就可以继续执行了。

到这里，我们就把应用程序的一次执行过程（也是一段控制流）称为一个 **任务**，把应用执行过程中的一段时间片段上的执行片段或空闲片段称为“**计算任务片**”或“**空闲任务片**”。当应用程序的所有任务片都完成后，应用程序的一次任务也就完成了。从一个程序的任务切换到另外一个程序的任务称为 **任务切换**。为了确保切换后的任务能够正确继续执行，操作系统需要支持让任务的执行“暂停”和“继续”。

我们又看到了熟悉的“暂停-继续”组合。一旦一条控制流需要支持“暂停-继续”，就需要提供一种控制流切换的机制，而且需要保证程序执行的控制流被切换出去之前和切换回来之后，能够继续正确执行。这需要让程序执行的状态（也称上下文），即在执行过程中同步变化的资源（如寄存器、栈等）保持不变，或者变化在它的预期之内。不是所有的资源都需要被保存，事实上只有那些对于程序接下来的正确执行仍然有用，且在它被切换出去的时候有被覆盖风险的那些资源才有被保存的价值。这些需要保存与恢复的资源被称为 **任务上下文 (Task Context)**。

---

提示：抽象与具体

注意：同学会在具体的操作系统设计实现过程中接触到一些抽象的概念，其实这些概念都是具体代码的结构和代码动态执行过程的文字表述而已。

### 4.3.3 不同类型的上下文与切换

在控制流切换过程中，我们需要结合硬件机制和软件实现来保存和恢复任务上下文。任务的一次切换涉及到被换出和即将被换入的两条控制流（分属两个应用的不同任务），通常它们都需要共同遵循某些约定来合作完成这一过程。在前两章，我们已经看到了两种上下文保存/恢复的实例。让我们再来看看它们：

- 第一章“应用程序与基本执行环境”中，我们介绍了[函数调用与栈](#)。当时提到过，为了支持嵌套函数调用，不仅需要硬件平台提供特殊的跳转指令，还需要保存和恢复[函数调用上下文](#)。注意在上述定义中，函数调用包含在普通控制流（与异常控制流相对）之内，且始终用一个固定的栈来保存执行的历史记录，因此函数调用并不涉及控制流的特权级切换。但是我们依然可以将其看成调用者和被调用者两个执行过程的“切换”，二者的协作体现在它们都遵循调用规范，分别保存一部分通用寄存器，这样的好处是编译器能够有足够的信息来尽可能减少需要保存的寄存器的数目。虽然当时用了很大的篇幅来说明，但其实整个过程都是编译器负责完成的，我们只需设置好栈就行了。
- 第二章“批处理系统”中第一次涉及到了某种异常（Trap）控制流，即两条控制流的特权级切换，需要保存和恢复[系统调用（Trap）上下文](#)。当时，为了让内核能够完全掌控应用的执行，且不会被应用破坏整个系统，我们必须利用硬件提供的特权级机制，让应用和内核运行在不同的特权级。应用运行在 U 特权级，它所被允许的操作进一步受限，处处被内核监督管理；而内核运行在 S 特权级，有能力处理应用执行过程中提出的请求或遇到的状况。

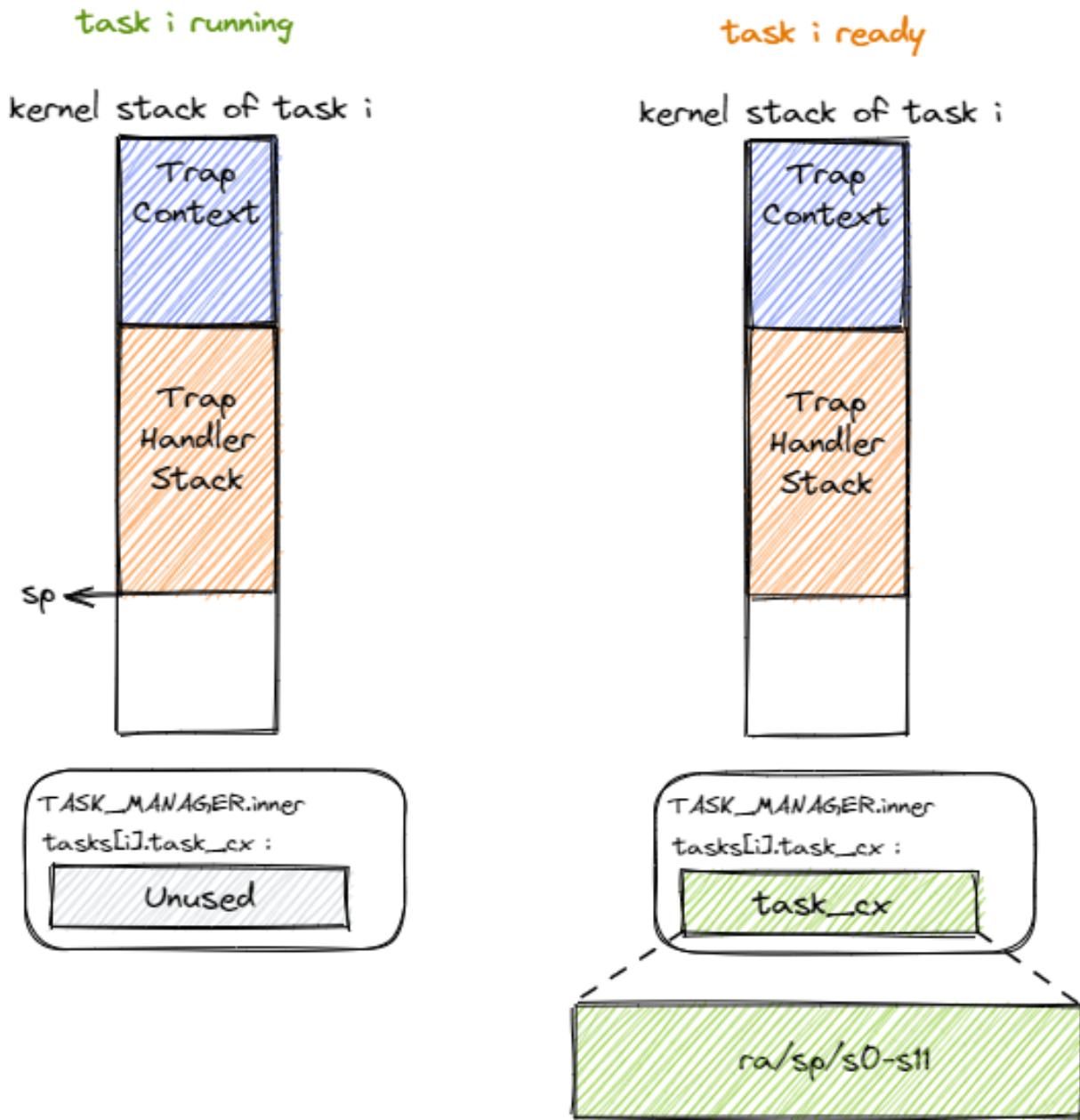
应用程序与操作系统打交道的核心在于硬件提供的 Trap 机制，也就是在 U 特权级运行的应用控制流和在 S 特权级运行的 Trap 控制流（操作系统的陷入处理部分）之间的切换。Trap 控制流是在 Trap 触发的一瞬间生成的，它和原应用控制流有着很密切的联系，因为它几乎唯一的目标就是处理 Trap 并恢复到原应用控制流。而且，由于 Trap 机制对于应用来说几乎是透明的，所以基本上都是 Trap 控制流在“负重前行”。Trap 控制流需要把 Trap 上下文（即几乎所有的通用寄存器）保存在自己的内核栈上，因为在 Trap 处理过程中所有的通用寄存器都可能被用到。可以回看[Trap 上下文保存与恢复](#)小节。

### 4.3.4 任务切换的设计与实现

本节所讲的任务切换是第二章提及的 Trap 控制流切换之外的另一种异常控制流，都是描述两条控制流之间的切换，如果将它和 Trap 切换进行比较，会有如下异同：

- 与 Trap 切换不同，它不涉及特权级切换；
- 与 Trap 切换不同，它的一部分是由编译器帮忙完成的；
- 与 Trap 切换相同，它对应用是透明的。

事实上，任务切换是来自两个不同应用在内核中的 Trap 控制流之间的切换。当一个应用 Trap 到 S 模式的操作系统内核中进行进一步处理（即进入了操作系统的 Trap 控制流）的时候，其 Trap 控制流可以调用一个特殊的 `__switch` 函数。这个函数表面上就是一个普通的函数调用：在 `__switch` 返回之后，将继续从调用该函数的位置继续向下执行。但是其间却隐藏着复杂的控制流切换过程。具体来说，调用 `__switch` 之后直到它返回前的这段时间，原 Trap 控制流 A 会先被暂停并被切换出去，CPU 转而运行另一个应用在内核中的 Trap 控制流 B。然后在某个合适的时机，原 Trap 控制流 A 才会从某一条 Trap 控制流 C（很有可能不是它之前切换到的 B）切换回来继续执行并最终返回。不过，从实现的角度讲，`__switch` 函数和一个普通的函数之间的核心差别仅仅是它会 **换栈**。



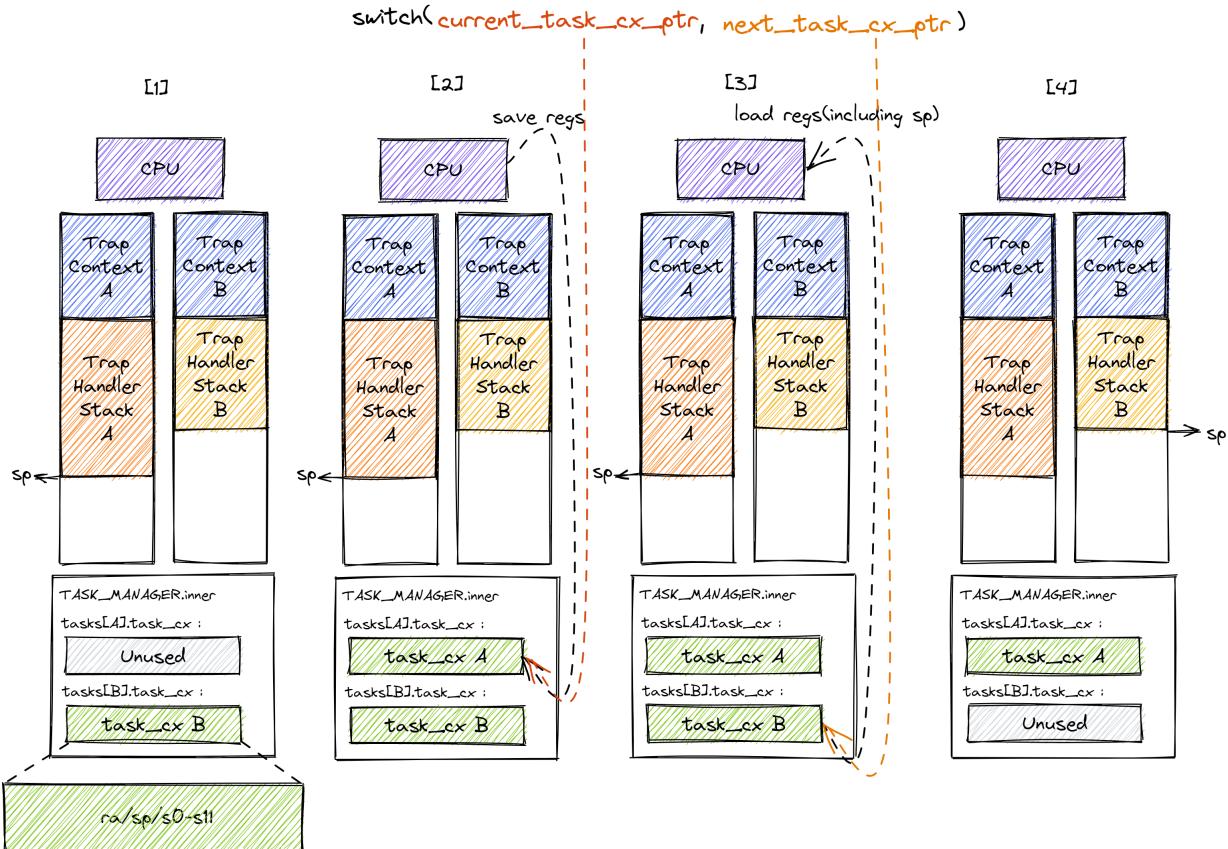
当 Trap 控制流准备调用 `__switch` 函数使任务从运行状态进入暂停状态的时候，让我们考察一下它内核栈上的情况。如上图左侧所示，在准备调用 `__switch` 函数之前，内核栈上从栈底到栈顶分别是保存了应用执行状态的 Trap 上下文以及内核在对 Trap 处理的过程中留下的调用栈信息。由于之后还要恢复回来执行，我们必须保存 CPU 当前的某些寄存器，我们称它们为 **任务上下文** (Task Context)。我们会在稍后介绍里面需要包含哪些寄存器。至于上下文保存的位置，下一节在我们会介绍任务管理器 TaskManager，在里面能找到一个数组 `tasks`，其中的每一项都是一个任务控制块即 `TaskControlBlock`，它负责保存一个任务的状态，而任务上下文 `TaskContext` 被保存在任务控制块中。在内核运行时我们会初始化 `TaskManager` 的全局实例 `TASK_MANAGER`，因此所有任务上下文实际保存在 `TASK_MANAGER` 中，从内存布局来看则是放在内核的全局数据 `.data` 段中。当我们将任务上下文保存完毕之后则转化为下图右侧的状态。当要从其他任务切换回来继续执行这个任务的时候，CPU 会读取同样的位置并从中恢复任务上下文。

对于当前正在执行的任务的 Trap 控制流，我们用一个名为 `current_task_cx_ptr` 的变量来保存放置当前任务上下文的地址；而用 `next_task_cx_ptr` 的变量来保存放置下一个要执行任务的上下文的地址。利

用 C 语言的引用来描述的话就是：

```
TaskContext *current_task_cx_ptr = &tasks[current].task_cx;
TaskContext *next_task_cx_ptr = &tasks[next].task_cx;
```

接下来我们同样从栈上内容的角度来看 `_switch` 的整体流程：



Trap 控制流在调用 `_switch` 之前就需要明确知道即将切换到哪一条目前正处于暂停状态的 Trap 控制流，因此 `_switch` 有两个参数，第一个参数代表它自己，第二个参数则代表即将切换到的那条 Trap 控制流。这里我们用上面提到过的 `current_task_cx_ptr` 和 `next_task_cx_ptr` 作为代表。在上图中我们假设某次 `_switch` 调用要从 Trap 控制流 A 切换到 B，一共可以分为四个阶段，在每个阶段中我们都给出了 A 和 B 内核栈上的内容。

- 阶段 [1]: 在 Trap 控制流 A 调用 `_switch` 之前，A 的内核栈上只有 Trap 上下文和 Trap 处理函数的调用栈信息，而 B 是之前被切换出去的；
- 阶段 [2]: A 在 A 任务上下文空间在里面保存 CPU 当前的寄存器快照；
- 阶段 [3]: 这一步极为关键，读取 `next_task_cx_ptr` 指向的 B 任务上下文，根据 B 任务上下文保存的内容来恢复 ra 寄存器、s0~s11 寄存器以及 sp 寄存器。只有这一步做完后，`_switch` 才能在一个函数跨两条控制流执行，即通过换栈也就实现了控制流的切换。
- 阶段 [4]: 上一步寄存器恢复完成后，可以看到通过恢复 sp 寄存器换到了任务 B 的内核栈上，进而实现了控制流的切换。这就是为什么 `_switch` 能做到一个函数跨两条控制流执行。此后，当 CPU 执行 `ret` 汇编伪指令完成 `_switch` 函数返回后，任务 B 可以从调用 `_switch` 的位置继续向下执行。

从结果来看，我们看到 A 控制流和 B 控制流的状态发生了互换，A 在保存任务上下文之后进入暂停状态，而 B 则恢复了上下文并在 CPU 上继续执行。

下面我们给出 `_switch` 的实现：

```

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

```

我们手写汇编代码来实现 `__switch`。在阶段 [1] 可以看到它的函数原型中的两个参数分别是当前 A 任务上下文指针 `current_task_cx_ptr` 和即将被切换到的 B 任务上下文指针 `next_task_cx_ptr`，从 [RISC-V 调用规范](#) 可以知道它们分别通过寄存器 `a0/a1` 传入。阶段 [2] 体现在第 19~27 行，即将当前 CPU 状态（包括 `ra` 寄存器、`s0~s11` 寄存器以及 `sp` 寄存器）保存到 A 任务上下文。相对的，阶段 [3] 体现在第 29~37 行，即根据 B 任务上下文保存的内容来恢复上述 CPU 状态。从中我们也能够看出 `TaskContext` 里面究竟包含哪些寄存器：

```

1 // os/src/task/context.rs
2
3 pub struct TaskContext {
4 ra: usize,
5 sp: usize,
6 s: [usize; 12],
7 }

```

保存 `ra` 很重要，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。对于一般的函数而言，Rust/C 编译器会在函数的起始位置自动生成代码来保存

`s0~s11` 这些被调用者保存的寄存器。但 `__switch` 是一个用汇编代码写的特殊函数，它不会被 Rust/C 编译器处理，所以我们需要在 `__switch` 中手动编写保存 `s0~s11` 的汇编代码。不用保存其它寄存器是因为：其它寄存器中，属于调用者保存的寄存器是由编译器在高级语言编写的调用函数中自动生成的代码来完成保存的；还有一些寄存器属于临时寄存器，不需要保存和恢复。

我们会将这段汇编代码中的全局符号 `__switch` 解释为一个 Rust 函数：

```

1 // os/src/task/switch.rs
2
3 global_asm!(include_str!("switch.S"));
4
5 use super::TaskContext;
6
7 extern "C" {
8 pub fn __switch(
9 current_task_cx_ptr: *mut TaskContext,
10 next_task_cx_ptr: *const TaskContext
11);
12 }
```

我们会调用该函数来完成切换功能而不是直接跳转到符号 `__switch` 的地址。因此在调用前后 Rust 编译器会自动帮助我们插入保存/恢复调用者保存寄存器的汇编代码。

仔细观察的话可以发现 `TaskContext` 很像一个普通函数栈帧中的内容。正如之前所说，`__switch` 的实现除了换栈之外几乎就是一个普通函数，也能在这里得到体现。尽管如此，二者的内涵却有着很大的不同。

同学可以自行对照注释看看图示中的后面几个阶段各是如何实现的。另外，当内核仅运行单个应用的时候，无论该任务主动/被动交出 CPU 资源最终都会交还给自己，这将导致传给 `__switch` 的两个参数相同，也就是某个 Trap 控制流自己切换到自己的情形，请同学对照图示思考目前的实现能否对它进行正确处理。

## 4.4 多道程序与协作式调度

### 4.4.1 本节导读

上一节我们已经介绍了任务切换是如何实现的，最终我们将其封装为一个函数 `__switch`。但是在实际使用的时候，我们需要知道何时调用该函数，以及如何确定传入函数的两个参数——分别代表待换出和即将被换入的两条 Trap 控制流。本节我们就来介绍任务切换的第一种实际应用场景：多道程序，并设计实现三叠纪“始初龙”协作式操作系统<sup>1</sup>。

本节的一个重点是展示进一步增强的操作系统管理能力和对处理器资源的相对高效利用。为此，对 **任务** 的概念进行进一步扩展和延伸：形成了

- 任务运行状态：任务从开始到结束执行过程中所处的不同运行状态：未初始化、准备执行、正在执行、已退出
- 任务控制块：管理程序的执行过程的任务上下文，控制程序的执行与暂停
- 任务相关系统调用：应用程序和操作系统之间的接口，用于程序主动暂停 `sys_yield` 和主动退出 `sys_exit`

这些都是三叠纪“始初龙”协作式操作系统<sup>Page 10, 1</sup> 需要具有的功能。本节的代码可以在 `ch3-coop` 分支上找到。

<sup>1</sup> 始初龙（也称始盗龙）是后三叠纪时期的两足食肉动物，也是目前所知最早的恐龙，它们只有一米长，却代表着恐龙的黎明。

## 4.4.2 多道程序背景与 `yield` 系统调用

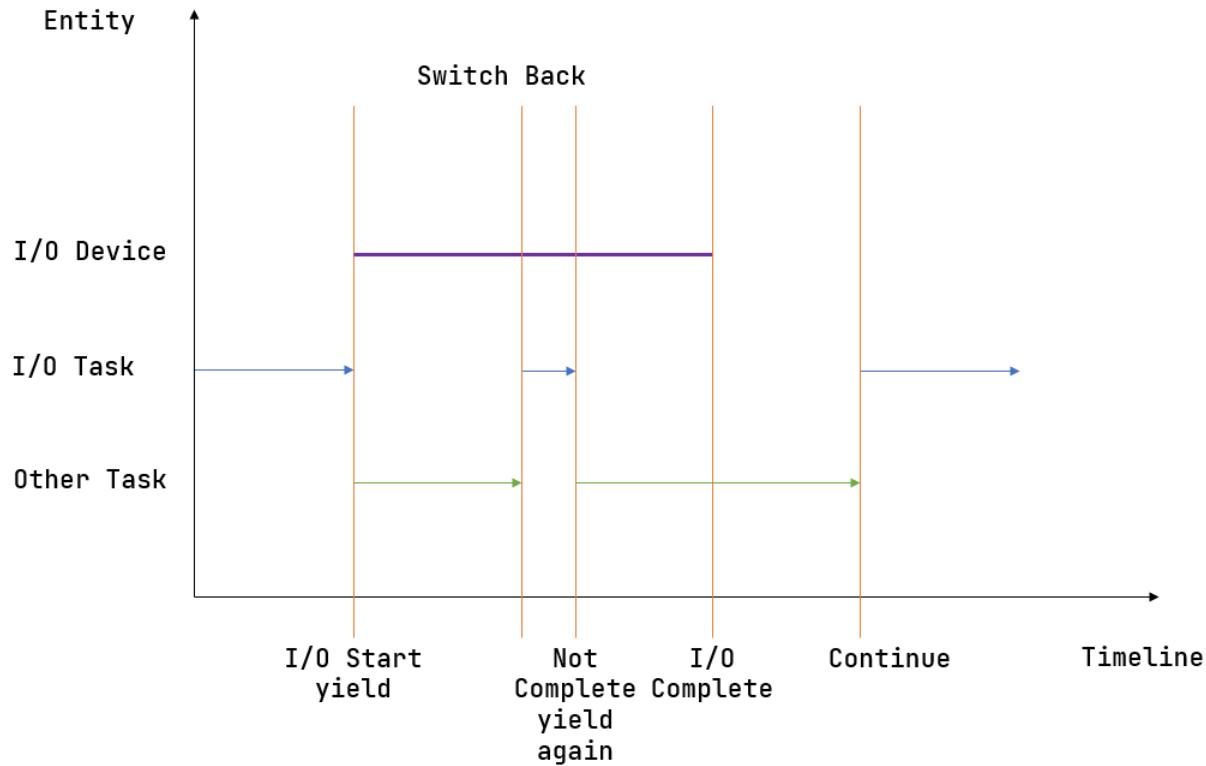
还记得第二章中介绍的批处理系统的设计初衷吗？它是注意到 CPU 并没有一直在执行应用程序，在一个应用程序运行结束直到下一个应用程序开始运行的这段时间，可能需要操作员取出上一个程序的执行结果并手动进行穿孔卡片（卡片记录了程序内容）的替换，这段空档期对于宝贵的 CPU 计算资源是一种巨大的浪费。于是批处理系统横空出世，它可以自动连续完成应用的加载和运行，并将一些本不需要 CPU 完成的简单任务交给廉价的外围设备，从而让 CPU 能够更加专注于计算任务本身，大大提高了 CPU 的利用率。

尽管 CPU 可以一直在跑应用了，但是其利用率仍有上升的空间。随着应用需求的不断复杂，有的时候会在内核的监督下访问一些外设，它们也是计算机系统的另一个非常重要的组成部分，即 **输入/输出 (I/O, Input/Output)**。CPU 会把 I/O 请求传递给外设，待外设处理完毕之后，CPU 便可以从外设读到其发出的 I/O 请求的处理结果。比如在从作为外部存储的磁盘上读取数据的时候，CPU 将要读取的扇区的编号以及放置读数据的内存物理地址传给磁盘，在磁盘把扇区数据拷贝到物理内存的事务完成之后，CPU 就能在物理内存中看到要读取的数据了。

在 CPU 对外设发出了 I/O 请求之后，由于 CPU 速度远快于外设速度，使得 CPU 不能立即继续执行，而是要等待（忙等或睡眠等）外设将请求处理完毕并拿到完整的处理结果之后才能继续。那么如何知道外设是否已经完成了请求呢？通常外设会提供一个可读的寄存器记录它目前的工作状态，于是 CPU 需要不断原地循环读取它直到它的结果显示设备已经将请求处理完毕了，才能继续执行（这就是 **忙等** 的含义）。然而，外设的计算速度和 CPU 相比可能慢了几个数量级，这就导致 CPU 有大量时间浪费在等待外设这件事情上，这段时间它几乎没有做任何事情，也在一定程度上造成了 CPU 的利用率不够理想。

我们暂时考虑 CPU 只能单向地通过读取外设提供的寄存器信息来获取外设处理 I/O 的完成状态。多道程序的思想在于：内核同时管理多个应用。如果外设处理 I/O 的时间足够长，那我们可以先进行任务切换去执行其他应用；在某次切换回来之后，应用再次读取设备寄存器，发现 I/O 请求已经处理完毕了，那么就可以根据返回的 I/O 结果继续向下执行了。这样的话，只要同时存在的应用足够多，就能一定程度上隐藏 I/O 外设处理相对于 CPU 的延迟，保证 CPU 不必浪费时间在等待外设上，而是几乎一直在进行计算。这种任务切换，是让应用 **主动** 调用 `sys_yield` 系统调用来实现的，这意味着应用主动交出 CPU 的使用权给其他应用。

这正是本节标题的后半部分“协作式”的含义。一个应用会持续运行下去，直到它主动调用 `sys_yield` 系统调用来交出 CPU 使用权。内核将很大的权力下放到应用，让所有的应用互相协作来最终达成最大化 CPU 利用率，充分利用计算资源这一终极目标。在计算机发展的早期，由于应用基本上都是一些简单的计算任务，且程序员都比较遵守规则，因此内核可以信赖应用，这样协作式的方案是没有问题的。



上图描述了一种多道程序执行的典型情况。其中横轴为时间线，纵轴为正在执行的实体。开始时，某个应用（蓝色）向外设提交了一个请求，随即可以看到对应的外设（紫色）开始工作。但是它要工作相当长的一段时间，因此应用（蓝色）不会去等待它结束而是会调用 `sys_yield` 主动交出 CPU 使用权来切换到另一个应用（绿色）。另一个应用（绿色）在执行了一段时间之后调用了 `sys_yield`，此时内核决定让应用（蓝色）继续执行。它检查了一下外设的工作状态，发现请求尚未处理完，于是再次调用 `sys_yield`。然后另一个应用（绿色）执行了一段时间之后 `sys_yield` 再次切换回这个应用（蓝色），这次的不同是它发现外设已经处理完请求了，于是它终于可以向下执行了。

上面我们是通过“避免无谓的外设等待来提高 CPU 利用率”这一切入点来引入 `sys_yield`。但其实调用 `sys_yield` 不一定与外设有关。随着内核功能的逐渐复杂，我们还会遇到其他需要等待的事件，我们都可以立即调用 `sys_yield` 来避免等待过程造成的浪费。

### 注解: `sys_yield` 的缺点

请同学思考一下，`sys_yield` 存在哪些缺点？

当应用调用它主动交出 CPU 使用权之后，它下一次再被允许使用 CPU 的时间点与内核的调度策略与当前的总体应用执行情况有关，很有可能远远迟于该应用等待的事件（如外设处理完请求）达成的时间点。这就会造成该应用的响应延迟不稳定或者很长。比如，设想一下，敲击键盘之后隔了数分钟之后才能在屏幕上看到字符，这已经超出了人类所能忍受的范畴。但也请不要担心，我们后面会有更加优雅的解决方案。

我们给出 `sys_yield` 的标准接口：

列表 1: 第三章新增系统调用（一）

```
/// 功能：应用主动交出 CPU 所有权并切换到其他应用。
/// 返回值：总是返回 0。
/// syscall ID: 124
fn sys_yield() -> isize;
```

然后是用户库对应的实现和封装：

```
// user/src/syscall.rs

pub fn sys_yield() -> isize {
 syscall(SYSCALL_YIELD, [0, 0, 0])
}

// user/src/lib.rs

pub fn yield_() -> isize { sys_yield() }
```

注意：yield 是 Rust 的关键字，因此我们只能将应用直接调用的接口命名为 yield\_。

接下来我们介绍内核应如何实现该系统调用。

#### 4.4.3 任务控制块与任务运行状态

在第二章批处理系统中我们只需知道目前执行到第几个应用就行了，因为在一段时间内，内核只管理一个应用，当它出错或退出之后内核会将其替换为另一个。然而，一旦引入了任务切换机制就没有那么简单了。在一段时间内，内核需要管理多个未完成的应用，而且我们不能对应用完成的顺序做任何假定，并不是先加入的应用就一定会先完成。这种情况下，我们必须在内核中对每个应用分别维护它的运行状态，目前有如下几种：

```
1 // os/src/task/task.rs
2
3 #[derive(Copy, Clone, PartialEq)]
4 pub enum TaskStatus {
5 UnInit, // 未初始化
6 Ready, // 准备运行
7 Running, // 正在运行
8 Exited, // 已退出
9 }
```

##### 注解：Rust Tips: #[derive]

通过 #[derive(...)] 可以让编译器为你的类型提供一些 Trait 的默认实现。

- 实现了 Clone Trait 之后就可以调用 clone 函数完成拷贝；
- 实现了 PartialEq Trait 之后就可以使用 == 运算符比较该类型的两个实例，从逻辑上说只有两个相等的应用执行状态才会被判为相等，而事实上也确实如此。
- Copy 是一个标记 Trait，决定该类型在按值传参/赋值的时候采用移动语义还是复制语义。

仅仅有这个是不够的，内核还需要保存一个应用的更多信息，我们将它们都保存在一个名为 **任务控制块** (Task Control Block) 的数据结构中：

```
1 // os/src/task/task.rs
2
3 #[derive(Copy, Clone)]
4 pub struct TaskControlBlock {
5 pub task_status: TaskStatus,
6 pub task(cx: TaskContext,
7 }
```

可以看到我们还在 `task_cx` 字段中维护了上一小节中提到的任务上下文。任务控制块非常重要，它是内核管理应用的核心数据结构。在后面的章节我们还会不断向里面添加更多内容，从而实现内核对应用更全面的管理。

#### 4.4.4 任务管理器

我们还需要一个全局的任务管理器来管理这些用任务控制块描述的应用：

```
// os/src/task/mod.rs

pub struct TaskManager {
 num_app: usize,
 inner: UPSafeCell<TaskManagerInner>,
}

struct TaskManagerInner {
 tasks: [TaskControlBlock; MAX_APP_NUM],
 current_task: usize,
}
```

其中仍然使用到了变量与常量分离的编程风格：字段 `num_app` 仍然表示任务管理器管理的应用的数目，它在 `TaskManager` 初始化之后就不会发生变化；而包裹在 `TaskManagerInner` 内的任务控制块数组 `tasks` 以及表示 CPU 正在执行的应用编号 `current_task` 会在执行应用的过程中发生变化：每个应用的运行状态都会发生变化，而 CPU 执行的应用也在不断切换。因此我们需要将 `TaskManagerInner` 包裹在 `UPSafeCell` 内以获取其内部可变性以及单核上安全的运行时借用检查能力。

再次强调，这里的 `current_task` 与第二章批处理系统中的含义不同。在批处理系统中，它除了表示 CPU 正在执行哪个应用外，表示一个既定的应用序列中的执行进度，可推测出在该应用之前的应用都已经执行完毕，之后的应用都没有执行；而在本章，我们只能通过它知道 CPU 正在执行哪个应用，而不能推测出其他应用的任何信息。

我们可重用并扩展之前初始化 `TaskManager` 的全局实例 `TASK_MANAGER`：

```
1 // os/src/task/mod.rs
2
3 lazy_static! {
4 pub static ref TASK_MANAGER: TaskManager = {
5 let num_app = get_num_app();
6 let mut tasks = [
7 TaskControlBlock {
8 task_cx: TaskContext::zero_init(),
9 task_status: TaskStatus::UnInit
10 },
11 MAX_APP_NUM
12];
13 for i in 0..num_app {
14 tasks[i].task_cx = TaskContext::goto_restore(init_app_cx(i));
15 tasks[i].task_status = TaskStatus::Ready;
16 }
17 TaskManager {
18 num_app,
19 inner: unsafe { UPSafeCell::new(TaskManagerInner {
20 tasks,
21 current_task: 0,
22 }) },
23 }
24 }
}
```

(下页继续)

(续上页)

```
24 };
25 }
```

- 第 5 行：调用 loader 子模块提供的 `get_num_app` 接口获取链接到内核的应用总数，后面会用到；
- 第 6~12 行：创建一个初始化的 `tasks` 数组，其中的每个任务控制块的运行状态都是 `UnInit`：表示尚未初始化；
- 第 13~16 行：依次对每个任务控制块进行初始化，将其运行状态设置为 `Ready`：表示可以运行，并初始化它的任务上下文；
- 从第 17 行开始：创建 `TaskManager` 实例并返回。

注意我们无需和第二章一样将 `TaskManager` 标记为 `Sync`，因为编译器可以根据 `TaskManager` 字段的情况自动推导出 `TaskManager` 是 `Sync` 的。

#### 4.4.5 实现 `sys_yield` 和 `sys_exit` 系统调用

`sys_yield` 表示应用自己暂时放弃对 CPU 的当前使用权，进入 `Ready` 状态。其实现用到了 `task` 子模块提供的 `suspend_current_and_run_next` 接口：

```
// os/src/syscall/process.rs

use crate::task::suspend_current_and_run_next;

pub fn sys_yield() -> isize {
 suspend_current_and_run_next();
 0
}
```

这个接口如字面含义，就是暂停当前的应用并切换到下个应用。

`sys_exit` 表示应用退出执行。它同样也改成基于 `task` 子模块提供的 `exit_current_and_run_next` 接口：

```
// os/src/syscall/process.rs

use crate::task::exit_current_and_run_next;

pub fn sys_exit(exit_code: i32) -> ! {
 println!("[kernel] Application exited with code {}", exit_code);
 exit_current_and_run_next();
 panic!("Unreachable in sys_exit!");
}
```

它的含义是退出当前的应用并切换到下个应用。在调用它之前我们打印应用的退出信息并输出它的退出码。如果是应用出错也应该调用该接口，不过我们这里并没有实现，有兴趣的同学可以尝试。

那么 `suspend_current_and_run_next` 和 `exit_current_and_run_next` 各是如何实现的呢？

```
// os/src/task/mod.rs

pub fn suspend_current_and_run_next() {
 mark_current_suspended();
 run_next_task();
}
```

(下页继续)

(续上页)

```
pub fn exit_current_and_run_next() {
 mark_current_exited();
 run_next_task();
}
```

它们都是先修改当前应用的运行状态，然后尝试切换到下一个应用。修改运行状态比较简单，实现如下：

```
1 // os/src/task/mod.rs
2
3 fn mark_current_suspended() {
4 TASK_MANAGER.mark_current_suspended();
5 }
6
7 fn mark_current_exited() {
8 TASK_MANAGER.mark_current_exited();
9 }
10
11 impl TaskManager {
12 fn mark_current_suspended(&self) {
13 let mut inner = self.inner.borrow_mut();
14 let current = inner.current_task;
15 inner.tasks[current].task_status = TaskStatus::Ready;
16 }
17
18 fn mark_current_exited(&self) {
19 let mut inner = self.inner.borrow_mut();
20 let current = inner.current_task;
21 inner.tasks[current].task_status = TaskStatus::Exited;
22 }
23 }
```

以 `mark_current_suspended` 为例。它调用了全局任务管理器 `TASK_MANAGER` 的 `mark_current_suspended` 方法。其中，首先获得里层 `TaskManagerInner` 的可变引用，然后根据其中记录的当前正在执行的应用 ID 对应在任务控制块数组 `tasks` 中修改状态。

接下来看看 `run_next_task` 的实现：

```
1 // os/src/task/mod.rs
2
3 fn run_next_task() {
4 TASK_MANAGER.run_next_task();
5 }
6
7 impl TaskManager {
8 fn run_next_task(&self) {
9 if let Some(next) = self.find_next_task() {
10 let mut inner = self.inner.exclusive_access();
11 let current = inner.current_task;
12 inner.tasks[next].task_status = TaskStatus::Running;
13 inner.current_task = next;
14 let current_task_cx_ptr = &mut inner.tasks[current].task_cx as *mut_
15 ~TaskContext;
16 let next_task_cx_ptr = &inner.tasks[next].task_cx as *const TaskContext;
17 drop(inner);
18 // before this, we should drop local variables that must be dropped_
19 ~manually
20 }
21 }
22 }
```

(下页继续)

(续上页)

```

18 unsafe {
19 __switch(
20 current_task_cx_ptr,
21 next_task_cx_ptr,
22);
23 }
24 // go back to user mode
25 } else {
26 panic!("All applications completed!");
27 }
28 }
29 }
```

run\_next\_task 使用任务管理器的全局实例 TASK\_MANAGER 的 run\_next\_task 方法。它会调用 find\_next\_task 方法尝试寻找一个运行状态为 Ready 的应用并返回其 ID。注意到其返回的类型是 Option<usize>，也就是说不一定能够找到，当所有的应用都退出并将自身状态修改为 Exited 就会出现这种情况，此时 find\_next\_task 应该返回 None。如果能够找到下一个可运行的应用的话，我们就可以分别拿到当前应用 current\_task\_cx\_ptr 和即将被切换到的应用 next\_task\_cx\_ptr 的任务上下文指针，然后调用 \_\_switch 接口进行切换。如果找不到的话，说明所有的应用都运行完毕了，我们可以直接 panic 退出内核。

注意：（第 16 行代码）在实际切换之前我们需要手动 drop 掉我们获取到的 TaskManagerInner 的来自 UPSafeCell 的借用标记。因为一般情况下它是在函数退出之后才会被自动释放，从而 TASK\_MANAGER 的 inner 字段得以回归到未被借用的状态，之后可以再借用。如果不手动 drop 的话，编译器会在 \_\_switch 返回时，也就是当前应用被切换回来的时候才 drop，这期间我们都不能修改 TaskManagerInner，甚至不能读（因为之前是可变借用），会导致内核 panic 报错退出。正因如此，我们需要在 \_\_switch 前提早手动 drop 掉 inner。

方法 find\_next\_task 又是如何实现的呢？

```

1 // os/src/task/mod.rs
2
3 impl TaskManager {
4 fn find_next_task(&self) -> Option<usize> {
5 let inner = self.inner.exclusive_access();
6 let current = inner.current_task;
7 (current + 1..current + self.num_app + 1)
8 .map(|id| id % self.num_app)
9 .find(|id| {
10 inner.tasks[*id].task_status == TaskStatus::Ready
11 })
12 }
13 }
```

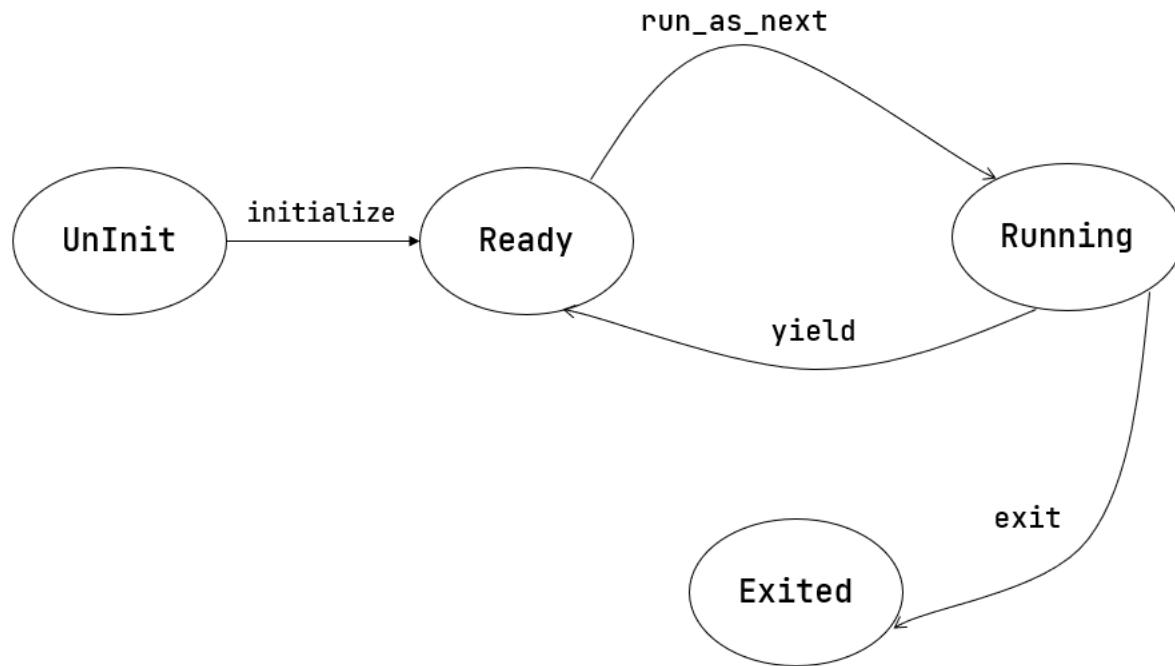
TaskManagerInner 的 tasks 是一个固定的任务控制块组成的表，长度为 num\_app，可以用下标 0~num\_app-1 来访问得到每个应用的控制状态。我们的任务就是找到 current\_task 后面第一个状态为 Ready 的应用。因此从 current\_task + 1 开始循环一圈，需要首先对 num\_app 取模得到实际的下标，然后检查它的运行状态。

### 注解：Rust 语法卡片：迭代器

a..b 实际上表示左闭右开区间  $[a, b)$ ，在 Rust 中，它会被表示为类型 core::ops::Range，标准库中为它实现好了 Iterator trait，因此它也是一个迭代器。

关于迭代器的使用方法如 map/find 等，请参考 Rust 官方文档。

我们可以总结一下应用的运行状态变化图：



#### 4.4.6 第一次进入用户态

在应用真正跑起来之前，需要 CPU 第一次从内核态进入用户态。我们在第二章批处理系统中也介绍过实现方法，只需在内核栈上压入构造好的 Trap 上下文，然后 `__restore` 即可。本章的思路大致相同，但是有一些变化。

当被任务切换出去的应用即将再次运行的时候，它实际上是通过 `__switch` 函数又完成一次任务切换，只是这次是被切换进来，取得了 CPU 的使用权。如果该应用是之前被切换出去的，那么它需要有任务上下文和内核栈上的 Trap 上下文，让切换机制可以正常工作。但是如果应用是第一次被执行，那内核应该怎么办呢？类似构造 Trap 上下文的方法，内核需要在应用的任务控制块上构造一个用于第一次执行的任务上下文。我们是在创建 `TaskManager` 的全局实例 `TASK_MANAGER` 的时候来进行这个初始化的。

```

// os/src/task/mod.rs

for i in 0..num_app {
 tasks[i].task_cx = TaskContext::goto_restore(init_app_cx(i));
 tasks[i].task_status = TaskStatus::Ready;
}

// os/src/task/context.rs

impl TaskContext {
 pub fn goto_restore(kstack_ptr: usize) -> Self {
 extern "C" { fn __restore(); }
 Self {
 ra: __restore as usize,
 sp: kstack_ptr,
 s: [0; 12],
 }
 }
}

```

(下页继续)

(续上页)

```

 }
}

// os/src/loader.rs

pub fn init_app(cx(app_id: usize) -> usize {
 KERNEL_STACK[app_id].push_context(
 TrapContext::app_init_context(get_base_i(app_id), USER_STACK[app_id].get_
 sp()),
)
}

```

对于每个任务，我们先调用 `init_app` 构造该任务的 Trap 上下文（包括应用入口地址和用户栈指针）并将其压入到内核栈顶。接着调用 `TaskContext::goto_restore` 来构造每个任务保存在任务控制块中的任务上下文。它设置任务上下文中的内核栈指针将任务上下文的 `ra` 寄存器设置为 `__restore` 的入口地址。这样，在 `__switch` 从它上面恢复并返回之后就会直接跳转到 `__restore`，此时栈顶是一个我们构造出来第一次进入用户态执行的 Trap 上下文，就和第二章的情况一样了。

需要注意的是，`__restore` 的实现需要做出变化：它 不再需要在开头 `mv sp, a0` 了。因为在 `__switch` 之后，`sp` 就已经正确指向了我们需要的 Trap 上下文地址。

在 `rust_main` 中我们调用 `task::run_first_task` 来开始应用的执行：

```

1 // os/src/task/mod.rs
2
3 impl TaskManager {
4 fn run_first_task(&self) -> ! {
5 let mut inner = self.inner.exclusive_access();
6 let task0 = &mut inner.tasks[0];
7 task0.task_status = TaskStatus::Running;
8 let next_task_cx_ptr = &task0.task_cx as *const TaskContext;
9 drop(inner);
10 let mut _unused = TaskContext::zero_init();
11 // before this, we should drop local variables that must be dropped manually
12 unsafe {
13 __switch(
14 &mut _unused as *mut TaskContext,
15 next_task_cx_ptr,
16);
17 }
18 panic!("unreachable in run_first_task!");
19 }
20
21 pub fn run_first_task() {
22 TASK_MANAGER.run_first_task();
23 }

```

这里我们取出即将最先执行的编号为 0 的应用的任务上下文指针 `next_task_cx_ptr` 并希望能够切换过去。注意 `__switch` 有两个参数分别表示当前应用和即将切换到的应用的任务上下文指针，其第一个参数存在的意义是记录当前应用的任务上下文被保存在哪里，也就是当前应用内核栈的栈顶，这样之后才能继续执行该应用。但在 `run_first_task` 的时候，我们并没有执行任何应用，`__switch` 前半部分的保存仅仅是在启动栈上保存了一些之后不会用到的数据，自然也无需记录启动栈栈顶的位置。

因此，我们显式在启动栈上分配了一个名为 `_unused` 的任务上下文，并将它的地址作为第一个参数传给 `__switch`，这样保存一些寄存器之后的启动栈栈顶的位置将会保存在此变量中。然而无论是此变量还是启动栈我们之后均不会涉及到，一旦应用开始运行，我们就开始在应用的用户栈和内核栈之间开始切换了。这里声明此变量的意义仅仅是为了避免覆盖到其他数据。

我们的“始初龙”协作式操作系统就算是实现完毕了。它支持把多个应用的代码和数据放置到内存中；并能够执行每个应用；在应用程序发出 `sys_yield` 系统调用时，能切换应用，从而让 CPU 尽可能忙于每个应用的计算任务，提高了任务调度的灵活性和 CPU 的使用效率。但“始初龙”协作式操作系统中任务调度的主动权在于应用程序的“自觉性”上，操作系统自身缺少强制的任务调度的手段，下一节我们将开始改进这方面的问题。

## 4.5 分时多任务系统与抢占式调度

### 4.5.1 本节导读

上一节我们介绍了“始初龙”协作式操作系统，依靠应用程序之间的协作来完成任务调度和切换。本节的重点是操作系统对中断的处理和对应用程序的抢占，并设计实现更加公平和高效交互的三叠纪“腔骨龙”<sup>1</sup> 抢占式操作系统。为此，我们需要对 **任务** 的概念进行进一步扩展和延伸：

- 分时多任务：操作系统管理每个应用程序，以时间片为单位来分时占用处理器运行应用。
- 时间片轮转调度：操作系统在一个程序用完其时间片后，就抢占当前程序并调用下一个程序执行，周而复始，形成对应用程序在任务级别上的时间片轮转调度。

#### 分时多任务系统的背景

上一节我们介绍了多道程序，它是一种允许应用在等待外设时主动切换到其他应用来达到总体 CPU 利用率最高的设计。在多道程序盛行的时候，计算机应用的开发者和计算机管理者是两类人，开发者不能管理计算机，对于应用的运行过程没有太多控制权，而计算机管理者的目标是总体 CPU 利用率最高，可以换成一个等价的指标：**吞吐量 (Throughput)**。大概可以理解为在某个时间点将一组应用放进去，要求在一段固定的时间之内执行完毕的应用最多，或者是总进度百分比最大。因此，所有的应用和编写应用的程序员都有这样的共识：只要 CPU 一直在做实际的工作就好。

从现在的眼光来看，当时的应用更多是一种 **后台应用 (Background Application)**，在将它加入执行队列之后我们只需定期确认它的运行状态。而随着技术的发展，涌现了越来越多的 **交互式应用 (Interactive Application)**，它们要达成的一个重要目标就是提高用户（应用的使用者和开发者）操作的响应速度，减少 **延迟 (Latency)**，这样才能优化应用的使用体验和开发体验。对于这些应用而言，即使需要等待外设或某些事件，它们也不会倾向于主动 `yield` 交出 CPU 使用权，因为这样可能会带来无法接受的延迟。也就是说，应用之间更多的是互相竞争宝贵的硬件资源，而不是相互合作。

如果应用自己很少 `yield`，操作系统内核就要开始收回之前下放的权力，由它自己对 CPU 资源进行集中管理并合理分配给各应用，这就是内核需要提供的任务调度能力。我们可以将多道程序的调度机制分类成 **协作式调度 (Cooperative Scheduling)**，因为它的特征是：只要一个应用不主动 `yield` 交出 CPU 使用权，它就会一直执行下去。与之相对，**抢占式调度 (Preemptive Scheduling)** 则是应用随时都有被内核切换出去的可能。

现代的任务调度算法基本都是抢占式的，它要求每个应用只能连续执行一段时间，然后内核就会将它强制性切换出去。一般将 **时间片 (Time Slice)** 作为应用连续执行时长的度量单位，每个时间片可能在毫秒量级。调度算法需要考虑：每次在换出之前给一个应用多少时间片去执行，以及要换入哪个应用。可以从性能（主要是吞吐量和延迟两个指标）和 **公平性 (Fairness)** 两个维度来评价调度算法，后者要求多个应用分到的时间片占比不应差距过大。

<sup>1</sup> 腔骨龙（也称虚形龙）最早出现于三叠纪晚期，它体形纤细，善于奔跑，以小型动物为食。

## 时间片轮转调度

简单起见，本书中我们使用 **时间片轮转算法** (RR, Round-Robin) 来对应用进行调度，只要对它进行少许拓展就能完全满足我们的需求。本章中我们仅需要最原始的 RR 算法，用文字描述的话就是维护一个任务队列，每次从队头取出一个应用执行一个时间片，然后把它丢到队尾，再继续从队头取出一个应用，以此类推直到所有的应用执行完毕。

本节的代码可以在 ch3 分支上找到。

## 4.5.2 RISC-V 架构中的中断

时间片轮转调度的核心机制就在于计时。操作系统的计时功能是依靠硬件提供的时钟中断来实现的。在介绍时钟中断之前，我们先简单介绍一下中断。

在 RISC-V 架构语境下，**中断** (Interrupt) 和我们第二章中介绍的异常（包括程序错误导致或执行 Trap 类指令如用于系统调用的 `ecall`）一样都是一种 Trap，但是它们被触发的原因却是不同的。对于某个处理器核而言，异常与当前 CPU 的指令执行是 **同步** (Synchronous) 的，异常被触发的原因一定能够追溯到某条指令的执行；而中断则 **异步** (Asynchronous) 于当前正在进行的指令，也就是说中断来自于哪个外设以及中断如何触发完全与处理器正在执行的当前指令无关。

---

### 注解：从底层硬件的角度区分同步和异步

从底层硬件的角度可能更容易理解这里所提到的同步和异步。以一个处理器的五级流水线设计而言，里面有取指、译码、算术、访存、寄存器等单元，都属于执行指令所需的硬件资源。那么假如某条指令的执行出现了问题，一定能被其中某个单元看到并反馈给流水线控制单元，从而它会在执行预定的下一条指令之前先进入异常处理流程。也就是说，异常在这些单元内部即可被发现并解决。

而对于中断，可以理解为发起中断的是一套与处理器执行指令无关的电路（从时钟中断来看就是简单的计数和比较器），这套电路仅通过一根导线接入处理器。当外设想要触发中断的时候则输入一个高电平或正边沿，处理器会在每执行完一条指令之后检查一下这根线，看情况决定是继续执行接下来的指令还是进入中断处理流程。也就是说，大多数情况下，指令执行的相关硬件单元和可能发起中断的电路是完全独立 **并行** (Parallel) 运行的，它们中间只有一根导线相连。

---

在不考虑指令集拓展的情况下，RISC-V 架构中定义了如下中断：

表 1: RISC-V 中断一览表

| Interrupt | Exception Code | Description                   |
|-----------|----------------|-------------------------------|
| 1         | 1              | Supervisor software interrupt |
| 1         | 3              | Machine software interrupt    |
| 1         | 5              | Supervisor timer interrupt    |
| 1         | 7              | Machine timer interrupt       |
| 1         | 9              | Supervisor external interrupt |
| 1         | 11             | Machine external interrupt    |

RISC-V 的中断可以分成三类：

- **软件中断** (Software Interrupt): 由软件控制发出的中断
- **时钟中断** (Timer Interrupt): 由时钟电路发出的中断
- **外部中断** (External Interrupt): 由外设发出的中断

另外，相比于异常，中断和特权级之间的联系更为紧密，可以看到这三种中断每一个都有 M/S 特权级两个版本。中断的特权级可以决定该中断是否会被屏蔽，以及需要 Trap 到 CPU 的哪个特权级进行处理。

在判断中断是否会被屏蔽的时候，有以下规则：

- 如果中断的特权级低于 CPU 当前的特权级，则该中断会被屏蔽，不会被处理；
- 如果中断的特权级高于与 CPU 当前的特权级或相同，则需要通过相应的 CSR 判断该中断是否会被屏蔽。

以内核所在的 S 特权级为例，中断屏蔽相应的 CSR 有 `sstatus` 和 `sie`。`sstatus` 的 `sie` 为 S 特权级的中断使能，能够同时控制三种中断，如果将其清零则会将它们全部屏蔽。即使 `sstatus.sie` 置 1，还要看 `sie` 这个 CSR，它的三个字段 `ssie/stie/seie` 分别控制 S 特权级的软件中断、时钟中断和外部中断的中断使能。比如对于 S 状态时钟中断来说，如果 CPU 不高于 S 特权级，需要 `sstatus.sie` 和 `sie.stie` 均为 1 该中断才不会被屏蔽；如果 CPU 当前特权级高于 S 特权级，则该中断一定会被屏蔽。

如果中断没有被屏蔽，那么接下来就需要软件进行处理，而具体到哪个特权级进行处理与一些中断代理 CSR 的设置有关。默认情况下，所有的中断都需要到 M 特权级处理。而通过软件设置这些中断代理 CSR 之后，就可以到低特权级处理，但是 Trap 到的特权级不能低于中断的特权级。事实上所有的中断/异常默认也都是到 M 特权级处理的。

我们会在[附录 C: 深入机器模式: RustSBI](#) 中再深入介绍中断/异常代理。在本书中我们只需要了解：

- U 特权级的应用程序发出系统调用或产生错误异常都会跳转到 S 特权级的操作系统内核来处理；
- S 特权级的时钟/软件/外部中断产生后，都会跳转到 S 特权级的操作系统内核来处理。

这里我们还需要对第二章介绍的系统调用和异常发生时的硬件机制做一下与中断相关的补充。默认情况下，当中断产生并进入某个特权级之后，在中断处理的过程中同特权级的中断都会被屏蔽。中断产生后，硬件会完成如下事务：

- 当中断发生时，`sstatus.sie` 字段会被保存在 `sstatus.spie` 字段中，同时把 `sstatus.sie` 字段置零，这样软件在进行后续的中断处理过程中，所有 S 特权级的中断都会被屏蔽；
- 当软件执行中断处理完毕后，会执行 `sret` 指令返回到被中断打断的地方继续执行，硬件会把 `sstatus.sie` 字段恢复为 `sstatus.spie` 字段内的值。

也就是说，如果不手动设置 `sstatus` CSR，在只考虑 S 特权级中断的情况下，是不会出现 **嵌套中断** (Nested Interrupt) 的。嵌套中断是指在处理一个中断的过程中再一次触发了中断。由于默认情况下，在软件开始响应中断前，硬件会自动禁用所有同特权级中断，自然也就不会再次触发中断导致嵌套中断了。

### 注解：嵌套中断与嵌套 Trap

嵌套中断可以分为两部分：在处理一个中断的过程中又被同特权级/高特权级中断所打断。默认情况下硬件会避免同特权级再次发生，但高特权级中断则是不可避免的会再次发生。

嵌套 Trap 则是指处理一个 Trap (可能是中断或异常) 的过程中又再次发生 Trap，嵌套中断是嵌套 Trap 的一个特例。在内核开发时我们需要仔细权衡哪些嵌套 Trap 应当被允许存在，哪些嵌套 Trap 又应该被禁止，这会关系到内核的执行模型。

### 4.5.3 时钟中断与计时器

由于软件 (特别是操作系统) 需要一种计时机制，RISC-V 架构要求处理器要有一个内置时钟，其频率一般低于 CPU 主频。此外，还有一个计数器用来统计处理器自上电以来经过了多少个内置时钟的时钟周期。在 RISC-V 64 架构上，该计数器保存在一个 64 位的 CSR `mtime` 中，我们无需担心它的溢出问题，在内核运行全程可以认为它是一直递增的。这个计数器被设计成在所有的特权级均可以通过一条 `rdttime` 的伪指令访问 (可以参考 RISC-V 规范的“Zicntr”拓展相关章节)。riscv 库已经封装了这个功能，我们直接调用相应接口，在 `timer` 子模块的 `get_time` 函数中取得计数器的值：

```
// os/src/timer.rs

use riscv::register::time;
```

(下页继续)

(续上页)

```
1 pub fn get_time() -> usize {
2 time::read()
3 }
```

另外一个 64 位的 CSR `mtimecmp` 的作用是：一旦计数器 `mtime` 的值超过了 `mtimecmp`，就会触发一次时钟中断。这使得我们可以方便的通过设置 `mtimecmp` 的值来决定下一次时钟中断何时触发。运行在 M 特权级的 SEE（这里是 RustSBI）预留了相关接口来实现计时器的控制：

```
1 // os/src/sbi.rs
2
3 const SBI_SET_TIMER: usize = 0;
4
5 pub fn set_timer(timer: usize) {
6 sbi_call(SBI_SET_TIMER, timer, 0, 0);
7 }
8
9 // os/src/timer.rs
10
11 use crate::config::CLOCK_FREQ;
12 const TICKS_PER_SEC: usize = 100;
13
14 pub fn set_next_trigger() {
15 set_timer(get_time() + CLOCK_FREQ / TICKS_PER_SEC);
16 }
```

- 代码片段第 5 行，`sbi` 子模块有一个 `set_timer` 调用，是一个由 SEE 提供的标准 SBI 接口函数，它可以用来自设置 `mtimecmp` 的值。
- 代码片段第 14 行，`timer` 子模块的 `set_next_trigger` 函数对 `set_timer` 进行了封装，它首先读取当前 `mtime` 的值，然后计算出 10ms 之内计数器的增量，再将 `mtimecmp` 设置为二者的和。这样，10ms 之后一个 S 特权级时钟中断就会被触发。

至于增量的计算方式，常数 `CLOCK_FREQ` 是一个预先获取到的各平台不同的时钟频率，单位为赫兹，也就是一秒钟之内计数器的增量。它可以在 `config` 子模块中找到。`CLOCK_FREQ` 除以常数 `TICKS_PER_SEC` 即是下一次时钟中断的计数器增量值。

后面可能还有一些计时的操作，比如统计一个应用的运行时长，我们再设计一个函数：

```
1 // os/src/timer.rs
2
3 const MICRO_PER_SEC: usize = 1_000_000;
4
5 pub fn get_time_us() -> usize {
6 time::read() / (CLOCK_FREQ / MICRO_PER_SEC)
7 }
```

`timer` 子模块的 `get_time_us` 以微秒为单位返回当前计数器的值，这让我们终于能对时间有一个具体概念了。实现原理就不再赘述。

新增一个系统调用，方便应用获取当前的时间：

列表 2: 第三章新增系统调用（二）

```
1 /// 功能：获取当前的时间，保存在 TimeVal 结构体 ts 中，_tz 在我们的实现中忽略
2 /// 返回值：返回是否执行成功，成功则返回 0
3 /// syscall ID: 169
```

(下页继续)

(续上页)

```
fn sys_get_time(ts: *mut TimeVal, _tz: usize) -> isize;

#[repr(C)]
pub struct TimeVal {
 pub sec: usize,
 pub usec: usize,
}
```

它在内核中的实现只需调用 `get_time_us` 函数即可。

#### 4.5.4 抢占式调度

有了时钟中断和计时器，抢占式调度就很容易实现了：

```
// os/src/trap/mod.rs

match scause.cause() {
 Trap::Interrupt(Interrupt::SupervisorTimer) => {
 set_next_trigger();
 suspend_current_and_run_next();
 }
}
```

我们只需在 `trap_handler` 函数下新增一个条件分支跳转，当发现触发了一个 S 特权级时钟中断的时候，首先重新设置一个 10ms 的计时器，然后调用上一小节提到的 `suspend_current_and_run_next` 函数暂停当前应用并切换到下一个。

为了避免 S 特权级时钟中断被屏蔽，我们需要在执行第一个应用之前进行一些初始化设置：

```
1 // os/src/main.rs
2
3 #[no_mangle]
4 pub fn rust_main() -> ! {
5 clear_bss();
6 println!("[kernel] Hello, world!");
7 trap::init();
8 loader::load_apps();
9 trap::enable_timer_interrupt();
10 timer::set_next_trigger();
11 task::run_first_task();
12 panic!("Unreachable in rust_main!");
13 }
14
15 // os/src/trap/mod.rs
16
17 use riscv::register::sie;
18
19 pub fn enable_timer_interrupt() {
20 unsafe { sie::set_stimer(); }
21 }
```

- 第 9 行设置了 `sie.stie` 使得 S 特权级时钟中断不会被屏蔽；
- 第 10 行则是设置第一个 10ms 的计时器。

这样，当一个应用运行了 10ms 之后，一个 S 特权级时钟中断就会被触发。由于应用运行在 U 特权级，且 `sie` 寄存器被正确设置，该中断不会被屏蔽，而是跳转到 S 特权级内的我们的 `trap_handler` 里面进行处

理，并顺利切换到下一个应用。这便是我们所期望的抢占式调度机制。从应用运行的结果也可以看出，三个 power 系列应用并没有进行 `yield`，而是由内核负责公平分配它们执行的时间片。

有同学可能会注意到，我们并没有将应用初始 Trap 上下文中的 `sstatus` 中的 `SPIE` 位置为 1。这将意味着 CPU 在用户态执行应用的时候 `sstatus` 的 `SIE` 为 0，根据定义来说，此时的 CPU 会屏蔽 S 状态所有中断，自然也包括 S 特权级时钟中断。但是可以观察到我们的应用在用尽一个时间片之后能够正常被打断。这是因为当 CPU 在 U 状态接收到一个 S 状态时钟中断时会被抢占，这时无论 `SIE` 位是否被设置都会进入 Trap 处理流程。

目前在等待某些事件的时候仍然需要 `yield`，其中一个原因是为了节约 CPU 计算资源，另一个原因是当事件依赖于其他的应用的时候，由于只有一个 CPU，当前应用的等待可能永远不会结束。这种情况下需要先将它切换出去，使得其他的应用到达它所期待的状态并满足事件的生成条件，再切换回来。

这里我们先通过 `yield` 来优化 轮询 (Busy Loop) 过程带来的 CPU 资源浪费。在 `03sleep` 这个应用中：

```
// user/src/bin/03sleep.rs

#[no_mangle]
fn main() -> i32 {
 let current_timer = get_time();
 let wait_for = current_timer + 3000;
 while get_time() < wait_for {
 yield_();
 }
 println!("Test sleep OK!");
 0
}
```

它的功能是等待 3000ms 然后退出。可以看出，我们会在循环里面 `yield_` 来主动交出 CPU 而不是无意义的忙等。其实，现在的抢占式调度会在它循环 10ms 之后切换到其他应用，这样能让内核给其他应用分配更多的 CPU 资源并让它们更早运行结束。

我们的“腔骨龙”协作式操作系统就算是实现完毕了。它支持把多个应用的代码和数据放置到内存中；并能够执行每个应用；在应用程序发出 `sys_yield` 系统调用时，协作式地切换应用；并通过时钟中断来实现抢占式调度并强行切换应用，从而提高了应用执行的灵活性、公平性和交互效率。

---

### 注解：内核代码执行是否会被中断打断？

目前为了简单起见，我们的内核不会被 S 特权级中断所打断，这是因为 CPU 在 S 特权级时，`sstatus.sie` 总为 0。但这会造成内核对部分中断的响应不及时，因此一种较为合理的设计是允许内核在处理系统调用的时候被打断优先处理某些中断，这是一种允许 Trap 嵌套的设计。从第四章可以看到，我们目前的设计不允许 Trap 嵌套，当通过 Trap 进入内核再次遇到 Trap 的时候，内核会直接 panic。

---

## 4.6 练习

### 4.6.1 课后练习

#### 编程题

1. \* 扩展内核，能够显示操作系统切换任务的过程。
2. \*\* 扩展内核，能够统计每个应用执行后的完成时间：用户态完成时间和内核态完成时间。
3. \*\* 编写浮点应用程序 A，并扩展内核，支持面向浮点应用的正常切换与抢占。
4. \*\* 编写应用程序或扩展内核，能够统计任务切换的大致开销。

5. \*\*\* 扩展内核，支持在内核态响应中断。
6. \*\*\* 扩展内核，支持在内核运行的任务（简称内核任务），并支持内核任务的抢占式切换。

注：上述扩展内核的编程基于 rcore/ucore tutorial v3: Branch ch3

## 问答题

1. \* 协作式调度与抢占式调度的区别是什么？
2. \* 中断、异常和系统调用有何异同之处？
3. \* RISC-V 支持哪些中断/异常？
4. \* 如何判断进入操作系统内核的起因是由于中断还是异常？
5. \*\* 在 RISC-V 中断机制中，PLIC 和 CLINT 各起到了什么作用？
6. \*\* 基于 RISC-V 的操作系统支持中断嵌套？请给出进一步的解释说明。
7. \*\* 本章提出的任务的概念与前面提到的进程的概念之间有何区别与联系？
8. \* 简单描述一下任务的地址空间中有哪些类型的数据和代码。
9. \* 任务控制块保存哪些内容？
10. \* 任务上下文切换需要保存与恢复哪些内容？
11. \* 特权级上下文和任务上下文有何异同？
12. \* 上下文切换为什么需要用汇编语言实现？
13. \* 有哪些可能的时机导致任务切换？
14. \*\* 在设计任务控制块时，为何采用分离的内核栈和用户栈，而不用一个栈？
15. \*\*\* 我们已经在 rCore 里实现了不少操作系统的功能：特权级、上下文切换、系统调用……为了让大家对相关代码更熟悉，我们来以另一个操作系统为例，比较一下功能的实现。看看换一段代码，你还认不认识操作系统。

阅读 Linux 源代码，特别是 riscv 架构相关的代码，回答以下问题：

1. Linux 正常运行的时候，stvec 指向哪个函数？是哪段代码设置的 stvec 的值？
2. Linux 里进行上下文切换的函数叫什么？（对应 rCore 的 \_\_switch）
3. Linux 里，和 rCore 中的 TrapContext 和 TaskContext 这两个类型大致对应的结构体叫什么？
4. Linux 在内核态运行的时候，tp 寄存器的值有什么含义？sscratch 的值是什么？
5. Linux 在用户态运行的时候，sscratch 的值有什么含义？
6. Linux 在切换到内核态的时候，保存了和用户态程序相关的什么状态？
7. Linux 在内核态的时候，被打断的用户态程序的寄存器值存在哪里？在 C 代码里如何访问？
8. Linux 是如何根据系统调用编号找到对应的函数的？（对应 rCore 的 syscall::syscall() 函数的功能）
9. Linux 用户程序调用 ecall 的参数是怎么传给系统调用的实现的？系统调用的返回值是怎样返回给用户态的？

阅读代码的时候，可以重点关注一下如下几个文件，尤其是第一个 entry.S，当然也可能会需要读到其它代码：

- arch/riscv/kernel/entry.S (与 rCore 的 switch.S 对比)
- arch/riscv/include/asm/current.h

- arch/riscv/include/asm/processor.h
- arch/riscv/include/asm/switch\_to.h
- arch/riscv/kernel/process.c
- arch/riscv/kernel/syscall\_table.c
- arch/riscv/kernel/traps.c
- include/linux/sched.h

此外，推荐使用 <https://elixir.bootlin.com> 阅读 Linux 源码，方便查找各个函数、类型、变量的定义及引用情况。

一些提示：

- Linux 支持各种架构，查找架构相关的代码的时候，请认准文件名中的 arch/riscv。
- 为了同时兼容 RV32 和 RV64，Linux 在汇编代码中用了几个宏定义。例如，REG\_L 在 RV32 上是 lw，而在 RV64 上是 ld。同理，REG\_S 在 RV32 上是 sw，而在 RV64 上是 sd。
- 如果看到 #ifdef CONFIG\_ 相关的预处理指令，是 Linux 根据编译时的配置启用不同的代码。一般阅读代码时，要么比较容易判断出这些宏有没有被定义，要么其实无关紧要。比如，Linux 内核确实应该和 rCore 一样，是在 S-mode 运行的，所以 CONFIG\_RISCV\_M\_MODE 应该是没有启用的。
- 汇编代码中可能会看到有些 TASK\_ 和 PT\_ 开头的常量，找不到定义。这些常量并没有直接写在源码里，而是自动生成的。

在汇编语言中需要用到的很多 struct 里偏移量的常量定义可以在 arch/riscv/kernel/asm-offsets.c 文件里找到。其中，OFFSET(NAME, struct\_name, field) 指的是 NAME 的值定义为 field 这一项在 struct\_name 结构体里，距离结构体开头的偏移量。最终这些代码会生成 asm/asm-offsets.h 供汇编代码使用。

- #include <asm/unistd.h> 在 arch/riscv/include/uapi/asm/unistd.h, #include <asm-generic/unistd.h> 在 include/uapi/asm-generic/unistd.h。

## 4.6.2 实验练习

实验练习包括实践作业和问答作业两部分。

### 实践作业

#### 获取任务信息

ch3 中，我们的系统已经能够支持多个任务分时轮流运行，我们希望引入一个新的系统调用 sys\_task\_info 以获取任务的信息，定义如下：

```
fn sys_task_info(id: usize, ts: *mut TaskInfo) -> isize
```

- syscall ID: 410
- 根据任务 ID 查询任务信息，任务信息包括任务 ID、任务控制块相关信息（任务状态）、任务使用的系统调用及调用次数、任务总运行时长。

```
struct TaskInfo {
 id: usize,
 status: TaskStatus,
 call: [SyscallInfo; MAX_SYSCALL_NUM],
```

(下页继续)

(续上页)

```
 time: usize
}
```

- 系统调用信息采用数组形式对每个系统调用的次数进行统计，相关结构定义如下：

```
struct SyscallInfo {
 id: usize,
 times: usize
}
```

- **参数：**
  - id: 待查询任务 id
  - ts: 待查询任务信息
- **返回值：** 执行成功返回 0，错误返回-1
- **说明：**
  - 相关结构已在框架中给出，只需添加逻辑实现功能需求即可。
- **提示：**
  - 大胆修改已有框架！除了配置文件，你几乎可以随意修改已有框架的内容。
  - 程序运行时间可以通过调用 `get_time()` 获取。
  - 系统调用次数可以考虑在进入内核态系统调用异常处理函数之后，进入具体系统调用函数之前维护。
  - 阅读 `TaskManager` 的实现，思考如何维护内核控制块信息（可以在控制块可变部分加入其他需要的信息）

## 打印调用堆栈（选做）

我们在调试程序时，除了正在执行的函数外，往往还需要知道当前的调用堆栈。这样的功能通常由调试器、运行环境、IDE 或操作系统等提供，但现在我们只能靠自己了。最基本的实现只需打印出调用链上的函数地址，更丰富的功能包括打印出函数名、函数定义、传递的参数等等。

本实验我们不提供新的测例，仅提供参考实现，各位同学可以通过对照 GDB、参考实现或自行构造调用链等方式检验自己的实现是否正确。

**提示：** 可以参考《编译原理》课程中关于函数调用栈帧的内容。

## 实验要求

- 完成分支: ch3-lab
- 实验目录要求

```
├── os(内核实现)
│ ├── Cargo.toml(配置文件)
│ └── src(所有内核的源代码放在 os/src 目录下)
│ └── main.rs(内核主函数)
```

(下页继续)

(续上页)

```

| └ ...
| reports (不是 report)
| └ lab3.md/pdf
| └ ...
| ...

```

- 通过所有已有的测例：

CI 使用的测例与本地相同，测试中，user 文件夹及其它与构建相关的文件将被替换，请不要试图依靠硬编码通过测试。

**注解：**你的实现只需且必须通过测例，建议读者感到困惑时先检查测例。

## 实验约定

### 问答作业

1. 正确进入 U 状态后，程序的特征还应有：使用 S 状态特权指令，访问 S 状态寄存器后会报错。请同学们可以自行测试这些内容（运行 Rust 两个 bad 测例（ch2b\_bad\_\*.rs）），描述程序出错行为，同时注意注明你使用的 sbi 及其版本。
2. 请通过 gdb 跟踪或阅读源代码了解机器从加电到跳转到 0x80200000 的过程，并描述重要的跳转。回答内核是如何进入 S 状态的？
  - 事实上进入 rustsbi (0x80000000) 之后就不需要使用 gdb 调试了。可以直接阅读 [代码](#)。
  - 可以使用 Makefile 中的 make debug 指令。
  - **一些可能用到的 gdb 指令：**
    - x/10i 0x80000000 : 显示 0x80000000 处的 10 条汇编指令。
    - x/10i \$pc : 显示即将执行的 10 条汇编指令。
    - x/10xw 0x80000000 : 显示 0x80000000 处的 10 条数据，格式为 16 进制 32bit。
    - info register: 显示当前所有寄存器信息。
    - info r t0: 显示 t0 寄存器的值。
    - break funcname: 在目标函数第一条指令处设置断点。
    - break \*0x80200000: 在 0x80200000 处设置断点。
    - continue: 执行直到碰到断点。
    - si: 单步执行一条汇编指令。

## 实验练习的提交报告要求

- 简单总结与上次实验相比本次实验你增加的东西（控制在 5 行以内，不要贴代码）。
- 完成问答问题。
- (optional) 你对本次实验设计及难度/工作量的看法，以及有哪些需要改进的地方，欢迎畅所欲言。

## 4.7 练习参考答案

### 4.7.1 课后练习

#### 编程题

##### 1. 扩展内核，能够显示操作系统切换任务的过程。

切换任务是在 `task/mod.rs` 中的 `TASK_MANAGER` 中完成的，主要靠 `mark_current_suspended()` 和 `mark_current_exited()` 标记任务运行结束，`run_next_task` 标记任务开始运行，在其中插入输出即可显示切换任务的过程（下面代码中的加入的三行 `println!`）

小心：03sleep.rs 测例会导致频繁切换任务，使得输出过多。可以修改测例的等待时间来减少输出。

```
/// Change the status of current `Running` task into `Ready`.
fn mark_current_suspended(&self) {
 let mut inner = self.inner.exclusive_access();
 let current = inner.current_task;
 println!("task {} suspended", current);
 inner.tasks[current].task_status = TaskStatus::Ready;
}

/// Change the status of current `Running` task into `Exited`.
fn mark_current_exited(&self) {
 let mut inner = self.inner.exclusive_access();
 let current = inner.current_task;
 println!("task {} exited", current);
 inner.tasks[current].task_status = TaskStatus::Exited;
}

/// Switch current `Running` task to the task we have found,
/// or there is no `Ready` task and we can exit with all applications completed
fn run_next_task(&self) {
 if let Some(next) = self.find_next_task() {
 let mut inner = self.inner.exclusive_access();
 let current = inner.current_task;
 println!("task {} start", current);
 inner.tasks[next].task_status = TaskStatus::Running;
 inner.current_task = next;
 let current_task_cx_ptr = &mut inner.tasks[current].task_cx as *mut_
 TaskContext;
 let next_task_cx_ptr = &inner.tasks[next].task_cx as *const TaskContext;
 drop(inner);
 // before this, we should drop local variables that must be dropped manually
 }
}
```

(下页继续)

(续上页)

```

unsafe {
 __switch(current_task_cx_ptr, next_task_cx_ptr);
}
// go back to user mode
} else {
 println!("All applications completed!");
 use crate::board::QEMUExit;
 crate::board::QEMU_EXIT_HANDLE.exit_success();
}
}
}

```

## 2. 扩展内核，能够统计每个应用执行后的完成时间：用户态完成时间和内核态完成时间。

**注解：**如果先做了第一题，记得删掉上面的输出，并且最好复原 03sleep.rs (如果有修改)。

### 2.1. 如何计算时间

首先给每个 TaskControlBlock 加上用户时间和内核时间两个属性：

```

#[derive(Copy, Clone)]
pub struct TaskControlBlock {
 pub task_status: TaskStatus,
 pub task_cx: TaskContext,
 pub user_time: usize,
 pub kernel_time: usize,
}

```

然后在 TaskManager 中设置一个“停表”变量 stop\_watch，并通过以下方法来“掐表”：

```

/// Inner of Task Manager
pub struct TaskManagerInner {
 /// task list
 tasks: [TaskControlBlock; MAX_APP_NUM],
 /// id of current `Running` task
 current_task: usize,
 /// 停表
 stop_watch: usize,
}

```

```

impl TaskManagerInner {
 fn refresh_stop_watch(&mut self) -> usize {
 let start_time = self.stop_watch;
 self.stop_watch = get_time_ms();
 self.stop_watch - start_time
 }
}

```

可以看到，这个方法每次会返回从当前到上一次“掐表”的时间间隔，然后刷新为当前时间。之后就可以利用它来统计应用执行时间了。

## 2.2. 统计内核态时间

切换内核态任务时，需要记录上一个任务的时间，并重新开始统计下一个任务的时间。

在 TaskManager 的三个方法中插入上面的 refresh\_stop\_watch() 即可（注意中文注释的位置）

```

fn run_first_task(&self) -> ! {
 let mut inner = self.inner.exclusive_access();
 let task0 = &mut inner.tasks[0];
 task0.task_status = TaskStatus::Running;
 let next_task_cx_ptr = &task0.task_cx as *const TaskContext;
 /// 开始记录时间
 inner.refresh_stop_watch();
 drop(inner);
 let mut _unused = TaskContext::zero_init();
 // before this, we should drop local variables that must be dropped manually
 unsafe {
 __switch(&mut _unused as *mut TaskContext, next_task_cx_ptr);
 }
 panic!("unreachable in run_first_task!");
}

/// Change the status of current `Running` task into `Ready`.
fn mark_current_suspended(&self) {
 let mut inner = self.inner.exclusive_access();
 let current = inner.current_task;
 // 统计内核时间
 inner.tasks[current].kernel_time += inner.refresh_stop_watch();
 inner.tasks[current].task_status = TaskStatus::Ready;
}

/// Change the status of current `Running` task into `Exited`.
fn mark_current_exited(&self) {
 let mut inner = self.inner.exclusive_access();
 let current = inner.current_task;
 // 统计内核时间并输出
 inner.tasks[current].kernel_time += inner.refresh_stop_watch();
 println!("[task {} exited. user_time: {} ms, kernel_time: {} ms.", current, inner.
 tasks[current].user_time, inner.tasks[current].kernel_time);
 inner.tasks[current].task_status = TaskStatus::Exited;
}

```

## 2.3. 统计用户态时间

在 trap 进入退出用户态的时候，可以统计用户态的运行时间，在 trap\_handler 的开头结尾添加上函数 user\_time\_end user\_time\_start:

```

/// handle an interrupt, exception, or system call from user space
pub fn trap_handler(cx: &mut TrapContext) -> &mut TrapContext {
 crate::task::user_time_end();
 let scause = scause::read(); // get trap cause
 let stval = stval::read(); // get extra value
 match scause.cause() {
 Trap::Exception(Exception::UserEnvCall) => {
 cx.sepc += 4;
 cx.x[10] = syscall(cx.x[17], [cx.x[10], cx.x[11], cx.x[12]]) as usize;
 }
 }
}

```

(下页继续)

(续上页)

```

 }
 Trap::Exception(Exception::StoreFault) |_
Trap::Exception(Exception::StorePageFault) => {
 println!("[kernel] PageFault in application, bad addr = {:#x}, bad_"
 instruction = {:#x}, kernel killed it.", stval, cx.sepc);
 exit_current_and_run_next();
}
Trap::Exception(Exception::IllegalInstruction) => {
 println!("[kernel] IllegalInstruction in application, kernel killed it.");
 exit_current_and_run_next();
}
Trap::Interrupt(Interrupt::SupervisorTimer) => {
 set_next_trigger();
 suspend_current_and_run_next();
}
_ => {
 panic!(
 "Unsupported trap {:?}, stval = {:#x}!",
 scause.cause(),
 stval
);
}
}
crate::task::user_time_start();
cx
}

```

这两个函数的实现也很简单，在 TaskManager 的实现中加入：

```

/// 统计内核时间，从现在开始算的是用户时间
fn user_time_start(&self) {
 let mut inner = self.inner.exclusive_access();
 let current = inner.current_task;
 inner.tasks[current].kernel_time += inner.refresh_stop_watch();
}

/// 统计用户时间，从现在开始算的是内核时间
fn user_time_end(&self) {
 let mut inner = self.inner.exclusive_access();
 let current = inner.current_task;
 inner.tasks[current].user_time += inner.refresh_stop_watch();
}

```

再在同文件 task/mod.rs 中包装成

```

/// 统计内核时间，从现在开始算的是用户时间
pub fn user_time_start() {
 TASK_MANAGER.user_time_start()
}

/// 统计用户时间，从现在开始算的是内核时间
pub fn user_time_end() {
 TASK_MANAGER.user_time_end()
}

```

就是全部的实现了。

## 2.4. 这个实现正确吗？(重要)

上面的实现其实隐含了所有的切换情况。请考虑以下的场景：

i. 第一个任务开始运行时：

- 先在 `run_first_task` 中刷新了停表(不统计)，随后通过 `__switch` 跳转到 `__restore` 进入用户态执行。
- 从用户态回来后，遇到函数 `user_time_end()`，此时刷新停表并统计用户态时间。

ii. 正常 syscall

- 从用户态回来后，遇到函数 `user_time_end()`，此时刷新停表并统计用户态时间。
- 在 `trap` 结尾处遇到函数 `user_time_start()`，刷新停表，并统计内核态时间

iii. 通过时钟中断切换任务：

- 前一个任务从用户态回来后，遇到函数 `user_time_end()`，此时刷新停表并统计用户态时间。
- 在 `mark_current_suspended` 中又刷新停表，并统计内核态时间。
- 切换到新任务后，在 `trap` 结尾处遇到函数 `user_time_start()`，刷新停表，并统计新任务的内核态时间

iv. 任务结束

- 从用户态回来后，遇到函数 `user_time_end()`，此时刷新停表并统计用户态时间。
- 在 `mark_current_exited` 中又刷新停表，并统计内核态时间，然后输出

## 3. 编写浮点应用程序 A，并扩展内核，支持面向浮点应用的正常切换与抢占。

需要在 `trap.s` 中加入浮点寄存器组的保存和恢复指令（见中文注释处）：

```
__alltraps:
 csrrw sp, sscratch, sp
 # now sp->kernel stack, sscratch->user stack
 # allocate a TrapContext on kernel stack
 addi sp, sp, -34*8
 # save general-purpose registers
 sd x1, 1*8(sp)
 # skip sp(x2), we will save it later
 sd x3, 3*8(sp)
 # skip tp(x4), application does not use it
 # save x5~x31
 .set n, 5
 .rept 27
 SAVE_GP %n
 .set n, n+1
 .endr
 # we can use t0/t1/t2 freely, because they were saved on kernel stack
 csrr t0, sstatus
 csrr t1, sepc
 sd t0, 32*8(sp)
 sd t1, 33*8(sp)
 # read user stack from sscratch and save it on the kernel stack
 csrr t2, sscratch
 # 浮点寄存器
```

(下页继续)

(续上页)

```

fsd fs0, 34*8(sp)
fsd fs1, 35*8(sp)
.....
sd t2, 2*8(sp)
set input argument of trap_handler(cx: &mut TrapContext)
mv a0, sp
call trap_handler

__restore:
now sp->kernel stack(after allocated), sscratch->user stack
restore sstatus/sepc
ld t0, 32*8(sp)
ld t1, 33*8(sp)
ld t2, 2*8(sp)
csrw sstatus, t0
csrw sepc, t1
csrw sscratch, t2
restore general-purpose registers except sp/tp
ld x1, 1*8(sp)
ld x3, 3*8(sp)
.set n, 5
.rept 27
 LOAD_GP %n
 .set n, n+1
.endr
浮点寄存器
fld fs0, 34*8(sp)
fld fs1, 35*8(sp)
.....
release TrapContext on kernel stack
addi sp, sp, 34*8
now sp->kernel stack, sscratch->user stack
csrrw sp, sscratch, sp
sret

```

此外，支持浮点指令可能还需要(包括但不限于)以下条件：

- 机器本身支持浮点指令
- Rust 编译目标包含浮点指令
  - 在 os/Makefile 中的 TARGET := riscv64gc-unknown-none-elf 支持浮点指令，而对应的 riscv64imac 则不支持。
  - 如果机器本身支持但使用 riscv64imac 作为编译目标，仍然可以通过强行插入指令的方式来支持浮点，如 fld fs0, 280(sp) 在 RISCV 指令集中表示为机器码 0x2472，就可以在上面的 trap.S 中插入

```
.short 0x2472 # fld fs0, 280(sp)
```

来支持浮点指令

- 需要通过控制浮点控制状态寄存器(如 fcsr)来检查 FPU 状态。详见 <https://five-embeddev.com/riscv-isa-manual/latest/machine.html#machine-trap-vector-base-address-register-mtvec>

#### 4. 编写应用程序或扩展内核，能够统计任务切换的大致开销。

所有任务切换都通过 `__switch`，可以包装一下这个函数，统计它运行的开销。首先删除 `task/mod.rs` 中的 `use switch::__switch`，然后加入以下函数来代替 `__switch`：

```
/// 切换的开始时间
static mut SWITCH_TIME_START: usize = 0;
/// 切换的总时间
static mut SWITCH_TIME_COUNT: usize = 0;

unsafe fn __switch(current_task_cx_ptr: *mut TaskContext, next_task_cx_ptr: *const_
 ~TaskContext) {
 SWITCH_TIME_START = get_time_us();
 switch::__switch(current_task_cx_ptr, next_task_cx_ptr);
 SWITCH_TIME_COUNT += get_time_us() - SWITCH_TIME_START;
}

fn get_switch_time_count() -> usize {
 unsafe { SWITCH_TIME_COUNT }
}
```

**小心：**这里统计时间使用了一个 `get_time_us`，即计算当前的微秒数。这是因为任务切换的时间比较短，不好用毫秒来计数。对应的实现在 `timer.rs` 中：

```
const USEC_PER_SEC: usize = 1000000;

/// get current time in milliseconds
pub fn get_time_us() -> usize {
 time::rad() / (CLOCK_FREQ / USEC_PER_SEC)
}
```

最后，在 `run_next_task` 中所有程序退出后，增加一条输出语句即可：

```
.....
} else {
 println!("All applications completed!");
 // 统计任务切换时间
 println!("task switch time: {} us", get_switch_time_count());
 use crate::board::QEMUExit;
 crate::board::QEMU_EXIT_HANDLE.exit_success();
}
```

#### 5. 扩展内核，支持在内核态响应中断。

内核开关中断的控制主要涉及 `sstatus` 寄存器的两个位：

- 在 `sie` 位可开关中断，如使用

```
use riscv::register::sstatus;
unsafe { sstatus::set_sie(); // 打开内核态中断
unsafe { sstatus::clear_sie(); // 关闭内核态中断
}
```

- 在 `spp` 位可以分辨中断的来源。现在将原本的“`trap/mod.rs:trap_handler()`”改名为“`user_trap_handler()`”，并增加一个新的“`trap_handler()`”函数：

```
#[no_mangle]
pub fn trap_handler(cx: &mut TrapContext) -> &mut TrapContext {
 match sstatus::read().spp() {
 sstatus::SPP::Supervisor => kernel_trap_handler(cx),
 sstatus::SPP::User => user_trap_handler(cx),
 }
}
```

从上面的函数可以看出，我们还需要额外在 trap/mod.rs 写一个 kernel\_trap\_handler 来处理内核中断：

```
/// 处理内核异常和中断
pub fn kernel_trap_handler(cx: &mut TrapContext) -> &mut TrapContext {
 let scause = scause::read();
 let stval = stval::read();
 match scause.cause() {
 Trap::Interrupt(Interrupt::SupervisorTimer) => {
 // 内核中断来自一个时钟中断
 println!("kernel interrupt: from timer");
 // 标记一下触发了中断
 mark_kernel_interrupt();
 set_next_trigger();
 }
 Trap::Exception(Exception::StoreFault) |_
 Trap::Exception(Exception::StorePageFault) => {
 panic!("[kernel] PageFault in kernel, bad addr = {:#x}, bad instruction = {:#x}, kernel killed it.", stval, cx.sepc);
 }
 _ => {
 // 其他的内核异常/中断
 panic!("unknown kernel exception or interrupt");
 }
 }
 cx
}
```

其中和用户态的中断实现大致相同，但异常和中断没有写全，可以后续补充。值得注意的有以下几点：

- 在函数中不使用 clear\_sie / set\_sie 来开关中断，这是因为虽然我们在 main.rs 的测试中打开了中断，但 RISC-V 会自动在中断触发时关闭 sstatus.SIE，在 sret 返回时打开 sstatus.SIE。在内核中需要小心打开中断的时机。例如触发中断时，内核正在拿着一些 mutex 锁，那么它在 trap\_handler 中处理时一旦尝试拿锁，就可能自己跟自己造成死锁。
- 在收到时钟中断时，输出了内容并调用了 mark\_kernel\_interrupt。这和中断机制无关，只是在 trap/mod.rs 中增加了一个全局变量，用于检查是否成功触发了内核中断

```
static mut KERNEL_INTERRUPT_TRIGGERED: bool = false;

/// 检查内核中断是否触发
pub fn check_kernel_interrupt() -> bool {
 unsafe { (&mut KERNEL_INTERRUPT_TRIGGERED as *mut bool).read_volatile() }
}

/// 标记内核中断已触发
pub fn trigger_kernel_interrupt() {
 unsafe {
 (&mut KERNEL_INTERRUPT_TRIGGERED as *mut bool).write_volatile(true);
 }
}
```

(下页继续)

(续上页)

```

 }
}
```

相对应的，在 main.rs 中，我们在 timer::set\_next\_trigger() 之后，在开始用户程序的 task::run\_first\_task(); 之前加了一段测试程序：

```

pub fn rust_main() -> ! {
 clear_bss();
 println!("[kernel] Hello, world!");
 trap::init();
 loader::load_apps();
 trap::enable_timer_interrupt();
 timer::set_next_trigger();

 use riscv::register::sstatus;
 unsafe { sstatus::set_sie() }; // 打开内核态中断
 loop {
 if trap::check_kernel_interrupt() {
 println!("kernel interrupt returned.");
 break;
 }
 }
 unsafe { sstatus::clear_sie() }; // 关闭内核态中断
 task::run_first_task();
 panic!("Unreachable in rust_main!");
}
```

但修改上面的代码后还无法通过编译，因为真正的“中断入口处理”是在“trap.S“中的汇编代码，如果它不做任何处理：

- 一个内核中断进入原本的 trap.S 代码时，它会首先遇到 csrrw sp, sscratch, sp
  - 按上面的测试，此时还没有进入用户程序，那么交换后 sp 为 0。

trap.S 下面的代码是增长栈并尝试保存寄存器：

```

__alltraps:
 csrrw sp, sscratch, sp
 # now sp->kernel stack, sscratch->user stack
 # allocate a TrapContext on kernel stack
 addi sp, sp, -34*8
 # save general-purpose registers
 sd x1, 1*8(sp)
```

在 addi 指令后，sp 会被写成 0xfffffffffffff8；在 sd 指令时，因为写入地址不合法（整个内核都在 0x80200000 附近）会触发 StorePageFault

- – 由于触发 StorePageFault，此时又会进入 \_\_alltraps，于是内核在这几条语句上死循环了。如果进入用户程序后，在用户程序的内核态触发内核中断，那么此时会把用户态的栈交换到 sp 寄存器，并在用户态的栈里写内核信息。这也是不可接受的错误。

因此，我们需要在“trap.S“内陷入和返回时做出判断，如果是内核中断异常则不换栈，如果是用户中断异常则通过“csrrw sp, sscratch, sp“换栈

将 \_\_alltrap 的开头改为：

```

__alltraps:
 csrr tp, sstatus
 andi tp, tp, 0x100
 beqz tp, __user_trap_start
 j __real_trap_entry

__user_trap_start:
 csrrw sp, sscratch, sp

__real_trap_entry:
 # now sp->kernel stack, sscratch->user stack
 # allocate a TrapContext on kernel stack
 addi sp, sp, -34*8


```

这段代码等价于取出 sstatus 寄存器的 spp 位做判断，如果为 0 则是用户态中断，否则是内核态中断。类似地，在中断返回时也要做一次判断。将 \_\_restore 改为：

```

__restore:
 # now sp->kernel stack (after allocated), sscratch->user stack
 # restore sstatus/sepc
 ld t0, 32*8(sp)
 ld t1, 33*8(sp)
 ld t2, 2*8(sp)
 csrw sstatus, t0
 csrw sepc, t1
 csrw sscratch, t2
 # get SPP
 andi t0, t0, 0x100
 bnez t0, __kernel_trap_end

__user_trap_end:
 # restore general-purpose registers except sp/tp
 ld x1, 1*8(sp)
 ld x3, 3*8(sp)
 .set n, 5
 .rept 27
 LOAD_GP %n
 .set n, n+1
 .endr
 # release TrapContext on kernel stack
 addi sp, sp, 34*8
 # now sp->kernel stack, sscratch->user stack
 csrrw sp, sscratch, sp
 sret

__kernel_trap_end:
 # restore general-purpose registers except sp/tp
 ld x1, 1*8(sp)
 ld x3, 3*8(sp)
 .set n, 5
 .rept 27
 LOAD_GP %n
 .set n, n+1
 .endr
 # release TrapContext on kernel stack
 addi sp, sp, 34*8
 sret

```

**小心：**注意返回时需要先获取中断来源信息再恢复寄存器。

以上就是内核中断的一个基本实现了，但还不具备实用价值，真正利用内核中断还需要扩展 `kernel_trap_handler` 函数，支持更有意义的中断类型。

### 如何在不破坏寄存器的情况下检查中断来源

在上面的代码实现中，在陷入 `__alltrap` 时利用了 `tp` 寄存器来检查 `sstatus` 的 SPP 信息，这样原本 `tp` 寄存器的信息就丢失了。但后续也有很多方法可以避免这个问题：

- 在 rCore-Tutorial 本章节中，内核栈和用户栈都直接写在 `loader.rs` 中。但在实际的内核中，用户空间往往在低地址（0x0000.....），而内核空间在高地址（0xfffff.....）。

可以利用这一点，把 `sp` 看作有符号整数，如果它是负数则是内核栈地址，代表内核态发的中断；如果是正数则是用户栈地址，代表用户态发的中断。即：

```
__alltraps:
 bgtz sp, __user_trap_start
 j __real_trap_entry

__user_trap_start:
 csrrw sp, sscratch, sp

__real_trap_entry:
 # now sp->kernel stack, sscratch->user stack
 # allocate a TrapContext on kernel stack
 addi sp, sp, -34*8

```

这样就可以规避目前代码中丢失 `tp` 寄存器的问题。

- 另一种方法是扩展 `sscratch` 的定义。目前 `sscratch` 只用于用户栈和内核栈的交换，可以使它表示一个专门的页或者中间栈或者处理函数，在其中完成寄存器的保存，再安全地用寄存器检查中断来源。

注：上述扩展内核的编程基于 rcore/ucore tutorial v3: Branch ch3

### 问答题

1. \* 协作式调度与抢占式调度的区别是什么？

协作式调度中，进程主动放弃 (yield) 执行资源，暂停运行，将占用的资源让给其它进程；抢占式调度中，进程会被强制打断暂停，释放资源让给别的进程。

2. \* 中断、异常和系统调用有何异同之处？

- 相同点

- 都会从通常的控制流中跳出，进入 trap handler 进行处理。

- 不同点

- 中断的来源是异步的外部事件，由外设、时钟、别的 hart 等外部来源，与 CPU 正在做什么没关系。

- 异常是 CPU 正在执行的指令遇到问题无法正常进行而产生的。

- 系统调用是程序有意想让操作系统帮忙执行一些操作，用专门的指令（如 `ecall`）触发的。

3. \* RISC-V 支持哪些中断/异常?

见下图

## : RISC-V Privileged Architectures V20190608-Priv-MSU-Ratified

| Interrupt | Exception Code | Description                             |
|-----------|----------------|-----------------------------------------|
| 1         | 0              | User software interrupt                 |
| 1         | 1              | Supervisor software interrupt           |
| 1         | 2              | <i>Reserved for future standard use</i> |
| 1         | 3              | Machine software interrupt              |
| 1         | 4              | User timer interrupt                    |
| 1         | 5              | Supervisor timer interrupt              |
| 1         | 6              | <i>Reserved for future standard use</i> |
| 1         | 7              | Machine timer interrupt                 |
| 1         | 8              | User external interrupt                 |
| 1         | 9              | Supervisor external interrupt           |
| 1         | 10             | <i>Reserved for future standard use</i> |
| 1         | 11             | Machine external interrupt              |
| 1         | 12–15          | <i>Reserved for future standard use</i> |
| 1         | $\geq 16$      | <i>Reserved for platform use</i>        |
| 0         | 0              | Instruction address misaligned          |
| 0         | 1              | Instruction access fault                |
| 0         | 2              | Illegal instruction                     |
| 0         | 3              | Breakpoint                              |
| 0         | 4              | Load address misaligned                 |
| 0         | 5              | Load access fault                       |
| 0         | 6              | Store/AMO address misaligned            |
| 0         | 7              | Store/AMO access fault                  |
| 0         | 8              | Environment call from U-mode            |
| 0         | 9              | Environment call from S-mode            |
| 0         | 10             | <i>Reserved</i>                         |
| 0         | 11             | Environment call from M-mode            |
| 0         | 12             | Instruction page fault                  |
| 0         | 13             | Load page fault                         |
| 0         | 14             | <i>Reserved for future standard use</i> |
| 0         | 15             | Store/AMO page fault                    |
| 0         | 16–23          | <i>Reserved for future standard use</i> |
| 0         | 24–31          | <i>Reserved for custom use</i>          |
| 0         | 32–47          | <i>Reserved for future standard use</i> |
| 0         | 48–63          | <i>Reserved for custom use</i>          |
| 0         | $\geq 64$      | <i>Reserved for future standard use</i> |

## 4. \* 如何判断进入操作系统内核的起因是由于中断还是异常?

检查 mcause 寄存器的最高位, 1 表示中断, 0 表示异常。

当然在 Rust 中也可以直接利用 riscv 库提供的接口判断:

```
let scause = scause::read();
if scause.is_interrupt() {
 do_something
}
if scause.is_exception() {
 do_something
}
```

又或者, 可以按照 trap/mod.rs:trap\_handler() 中的写法, 用 match scause.cause() 来判断。

## 5. \*\* 在 RISC-V 中断机制中, PLIC 和 CLINT 各起到了什么作用?

CLINT 处理时钟中断 (MTI) 和核间的软件中断 (MSI); PLIC 处理外部来源的中断 (MEI)。

PLIC 的规范文档: <https://github.com/riscv/riscv-plic-spec>

## 6. \*\* 基于 RISC-V 的操作系统支持中断嵌套? 请给出进一步的解释说明。

RISC-V 原生不支持中断嵌套。(在 S 状态的内核中) 只有 sstatus 的 SIE 位为 1 时, 才会开启中断, 再由 sie 寄存器控制哪些中断可以触发。触发中断时, sstatus.SPIE 置为 sstatus.SIE, 而 sstatus.SIE 置为 0; 当执行 sret 时, sstatus.SIE 置为 sstatus.SPIE, 而 sstatus.SPIE 置为 1。这意味着触发中断时, 因为 sstatus.SIE 为 0, 所以无法再次触发中断。

## 7. \*\* 本章提出的任务的概念与前面提到的进程的概念之间有何区别与联系?

- 联系: 任务和进程都有自己独立的栈、上下文信息, 任务是进程的“原始版本”, 在第五章会将目前的用户程序从任务升级为进程。
- 区别: 任务之间没有地址空间隔离, 实际上是能相互访问到的; 进程之间有地址空间隔离, 一个进程无法访问到另一个进程的地址。

## 8. \* 简单描述一下任务的地址空间中有哪些类型的数据和代码。

可参照 user/src/linker.ld:

- .text: 任务的代码段, 其中开头的 .text.entry 段包含任务的入口地址
- .rodata: 只读数据, 包含字符串常量, 如测例中的 `println!("Test power_3 OK!");` 实际打印的字符串存在这里
- .data: 需要初始化的全局变量
- .bss: 未初始化或初始为 0 的全局变量。
- 在之后第四章的 user/src/bin/00power\_3.rs 中, 会把第三章中在用户栈上定义的数组移到全局变量中 `static mut S: [u64; LEN] = [0u64; LEN];`
- 在第五章的 user/lib.rs 中, 会在 bss 段构造一个用户堆 `static mut HEAP_SPACE: [u8; USER_HEAP_SIZE] = [0; USER_HEAP_SIZE];`

除此之外, 在内核中为每个任务构造的用户栈 os/src/loader.rs:USER\_STACK 也属于各自任务的地址。

## 9. \* 任务控制块保存哪些内容?

在本章中, 任务控制块即 os/src/task/task.rs:TaskControlBlock 保存任务目前的执行状态 task\_status 和任务上下文 task(cx)。

## 10. \* 任务上下文切换需要保存与恢复哪些内容?

需要保存通用寄存器的值, PC; 恢复的时候除了保存的内容以外还要恢复特权级到用户态。

11. \* 特权级上下文和任务上下文有何异同？

- 相同点：特权级上下文和任务上下文都保留了一组寄存器，都代表一个“执行流”
- 不同点：
  - 特权级上下文切换可以发生在中断异常时，所以它不符合函数调用约定，需要保存所有通用寄存器。同时它又涉及特权级切换，所以还额外保留了一些 CSR，在切换时还会涉及更多的 CSR。
  - 任务上下文由内核手动触发，它包装在 `os/src/task/switch.rs`:`__switch()` 里，所以除了“返回函数与调用函数不同”之外，它符合函数调用约定，只需要保存通用寄存器中 `callee` 类型的寄存器。为了满足切换执行流时“返回函数与调用函数不同”的要求，它还额外保存 `ra`。

12. \* 上下文切换为什么需要用汇编语言实现？

上下文切换过程中，需要我们直接控制所有的寄存器。C 和 Rust 编译器在编译代码的时候都会“自作主张”使用通用寄存器，以及我们不知道的情况下访问栈，这是我们需要避免的。

切换到内核的时候，保存好用户态状态之后，我们将栈指针指向内核栈，相当于构建好一个高级语言可以正常运行的环境，这时候就可以由高级语言接管了。

13. \* 有哪些可能的时机导致任务切换？

系统调用（包括进程结束执行）、时钟中断。

14. \*\* 在设计任务控制块时，为何采用分离的内核栈和用户栈，而不用一个栈？

用户程序可以任意修改栈指针，将其指向任意位置，而内核在运行的时候总希望在某一个合法的栈上，所以需要用分开的两个栈。

此外，利用后面的章节的知识可以保护内核和用户栈，让用户无法读写内核栈上的内容，保证安全。

15. \*\*\*（以下答案以 Linux 5.17 为准）

1. `arch/riscv/kernel/entry.S` 里的 `handle_exception`；`arch/riscv/kernel/head.S` 里的 `setup_trap_vector`
2. `arch/riscv/kernel/entry.S` 里的 `__switch_to`
3. `TrapContext` 对应 `pt_regs`；`TaskContext` 对应 `task_struct`（在 `task_struct` 中也包含一些其它的和调度相关的信息）
4. `tp` 指向当前被打断的任务的 `task_struct`（参见 `arch/riscv/include/asm/current.h` 里的宏 `current`）；`sscratch` 是 0
5. `sscratch` 指向当前正在运行的任务的 `task_struct`，这样设计可以用来区分异常来自用户态还是内核态。
6. 所有通用寄存器，`sstatus`, `sepc`, `scause`
7. 内核栈底；`arch/riscv/include/asm/processor.h` 里的 `task_pt_regs` 宏
8. `arch/riscv/kernel/syscall_table.c` 里的 `sys_call_table` 作为跳转表，根据系统调用编号调用。
9. 从保存的 `pt_regs` 中读保存的 `a0` 到 `a7` 到机器寄存器里，这样系统调用实现的 C 函数就会作为参数接收到这些值，返回值是将返回的 `a0` 写入保存的 `pt_regs`，然后切换回用户态的代码负责将其“恢复”到 `a0`

---

## 第四章：地址空间

---

### 5.1 引言

#### 5.1.1 本章导读

物理内存是操作系统需要管理的一个重要资源，让运行在一台机器上的多个应用程序不用“争抢”，都能随时得到想要的任意多的内存，是操作系统的想要达到的理想目标。提高系统物理内存的动态使用效率，通过隔离应用的物理内存空间保证应用间的安全性，把“有限”物理内存变成“无限”虚拟内存，是操作系统的一系列重要的目标，本章展现了操作系统为实现“理想”而要扩展的一系列功能：

- 通过动态内存分配，提高了应用程序对内存的动态使用效率
- 通过页表的虚实内存映射机制，简化了编译器对应用的地址空间设置
- 通过页表的虚实内存映射机制，加强了应用之间，应用与内核之间的内存隔离，增强了系统安全
- 通过页表的虚实内存映射机制，可以实现空分复用（提出，但没有实现）

本章将进一步设计与实现具有上述大部分功能的侏罗纪“头甲龙”<sup>1</sup> 操作系统，让应用开发更加简单，应用程序更加通用，且让应用和操作系统都有强大的地址空间隔离的安全保护。

上一章，我们分别实现了多道程序和分时多任务系统，它们的核心机制都是任务切换。由于多道程序和分时多任务系统的设计初衷不同，它们在任务切换的时机和策略也不同。有趣的一点是，任务切换机制对于应用是完全 透明 (Transparent) 的，应用可以不对内核实现该机制的策略做任何假定（除非要进行某些针对性优化），甚至可以完全不知道这机制的存在。

在大多数应用（也就是应用开发者）的视角中，它们会独占一个 CPU 和特定（连续或不连续）的内存空间。当然，通过上一章的学习，我们知道在现代操作系统中，出于公平性的考虑，我们极少会让独占 CPU 这种情况发生。所以应用自认为的独占 CPU 只是内核想让应用看到的一种 幻象 (Illusion)，而 CPU 计算资源被 时分复用 (TDM, Time-Division Multiplexing) 的实质被内核通过恰当的抽象隐藏了起来，对应用不可见。

与之相对，我们目前还没有对内存管理功能进行进一步拓展，仅仅是把程序放到某处的物理内存中。在内存访问方面，所有的应用都直接通过物理地址访问物理内存，这使得应用开发者需要了解繁琐的物理地址空间布局，访问内存也很不方便。在上一章中，出于任务切换的需要，所有的应用都在初始化阶段被加载到内存

---

<sup>1</sup> 头甲龙最早出现在 1.8 亿年以前的侏罗纪中期，是身披重甲的食素恐龙，尾巴末端的尾锤，是防身武器。

中并同时驻留下去直到它们全部运行结束。而且，所有的应用都直接通过物理地址访问物理内存。这会带来以下问题：

- 首先，内核提供给应用的内存访问接口不够透明，也不好用。由于应用直接访问物理内存，这需要它在构建的时候就清楚所运行计算机的物理内存空间布局，还需规划自己需要被加载到哪个地址运行。为了避免冲突可能还需要应用的开发者们对此进行协商，这显然是一件在今天看来不够通用且极端麻烦的事情。
- 其次，内核并没有对应用的访存行为进行任何保护措施，每个应用都有计算机系统中整个物理内存的读写权力。即使应用被限制在 U 特权级下运行，它还是能够造成很多麻烦：比如它可以读写其他应用的数据来窃取信息或者破坏其它应用的正常运行；甚至它还可以修改内核的代码段来替换掉原本的 `trap_handler` 函数，来挟持内核执行恶意代码。总之，这造成系统既不安全、也不稳定。
- 再次，目前应用的内存使用空间在其运行前已经限定死了，内核不能灵活地给应用程序提供的运行时动态可用内存空间。比如一个应用结束后，这个应用所占的空间就被释放了，但这块空间无法动态地给其它还在运行的应用使用。

因此，为了简化应用开发，防止应用胡作非为，本章将更好地管理物理内存，并提供给应用一个抽象出来的更加透明易用、也更加安全的访存接口，这就是基于分页机制的虚拟内存。站在应用程序运行的角度看，就是存在一个从“0”地址开始的非常大的可读/可写/可执行的地址空间（Address Space），而站在操作系统的角度看，每个应用被局限在分配给它的物理内存空间中运行，无法读写其它应用和操作系统所在的内存空间。

实现地址空间的第一步就是实现分页机制，建立好虚拟内存和物理内存的页映射关系。此过程需要硬件支持，硬件细节与具体 CPU 相关，涉及地址映射机制等，相对比较复杂。总体而言，我们需要思考如下问题：

- 硬件中物理内存的范围是什么？
- 哪些物理内存空间需要建立页映射关系？
- 如何建立页表使能分页机制？
- 如何确保 OS 能够在分页机制使能前后的不同时间段中都能正常寻址和执行代码？
- 页目录表（一级）的起始地址设置在哪里？
- 二级/三级等页表的起始地址设置在哪里，需要多大空间？
- 如何设置页目录表项/页表项的内容？
- 如果要让每个任务有自己的地址空间，那每个任务是否要有自己的页表？
- 代表应用程序的任务和操作系统需要有各自的页表吗？
- 在有了页表之后，任务和操作系统之间应该如何传递数据？

如果能解决上述问题，我们就能设计实现具有超强防护能力的侏罗纪“头甲龙”操作系统。并可更好地理解地址空间，虚拟地址等操作系统的抽象概念与操作系统的虚存具体实现之间的联系。

---

#### 注解：提供巨大虚拟内存空间的 Atlas Supervisor 操作系统

两级存储系统在 1940 年就已经存在。1950-1960 年期间，计算机的主存（今天称为 RAM）通常是容量小的磁芯，而辅助存储器通常是容量大的磁鼓。处理器只能对主存寻址来读写数据或执行代码。1960 年前后，位于计算机内存中的应用程序数量和单个程序的体积都在迅速增加，物理内存的容量跟不上应用对内存的需求。应用程序员的一个主要工作是在程序中编写在主存和辅助存储之间移动数据的代码，来扩大应用程序访问的数据量。计算机专家开始考虑能否让计算机自动地移动数据来减轻程序员的编程负担？

虚拟内存（Virtual memory）技术概念首次由德国的柏林工业大学（Technische Universität Berlin）博士生 Fritz-Rudolf Güntsch 提出。在他的博士论文中设想了一台计算机，其内存地址空间大小为  $10^5$  个字，可精确映射到作为二级存储的磁鼓（大小也为  $10^5$  个字）上，应用程序读写的数据的实际位置由硬件和监控器（即操作系统）来管理和控制，并在物理主存（RAM）和辅存（二级存储）之间按需搬移数据。即主存中只放置应用程序最近访问的数据，而应用程序最近不访问的数据会搬到辅存中，在应用程序需要时再搬回内存中。这个搬移过程对应用程序是透明的。

虚拟内存的设想在 1959 年变成了现实。英国曼彻斯特大学的 Tom Kilburn 教授领导的团队于 1959 年展示了他们设计的 Atlas 计算机和 Atlas Supervisor 操作系统，开创了在今天仍然普遍使用的操作系统技术：分页 (paging) 技术和虚拟内存 (virtual memory，当时称为 one-level storage system)。他们的核心思想中的根本性创新是区分了“地址 (address)”和“内存位置 (memory location)”，并因此创造了三项发明：

1. 地址转换：硬件自动将处理器生成的每个地址转换为其当前内存位置。
2. 按需分页 (demand paging)：由硬件地址转换触发缺页中断后，由操作系统将缺失的数据页移动到主存储器中，并形成正确的地址转换映射。
3. 页面置换算法：检查最无用 (least useful) 的页，并将其移回二级存储中，这样可以让经常访问的数据驻留在主存中。

计算机科学家对 Atlas Supervisor 操作系统给予高度的评价。Brinch Hansen 认为它是操作系统史上最重大的突破。Simon Lavington 认为它是第一个可识别的现代操作系统。

## 5.1.2 实践体验

本章的应用和上一章相同，只不过由于内核提供给应用的访存接口被替换，应用的构建方式发生了变化，这方面在下面会深入介绍。因此应用运行起来的效果与上一章是一致的。

获取本章代码：

```
$ git clone https://github.com/rcore-os/rCore-Tutorial-v3.git
$ cd rCore-Tutorial-v3
$ git checkout ch4
```

在 qemu 模拟器上运行本章代码：

```
$ cd os
$ make run
```

如果顺利的话，我们将看到和上一章相同的运行结果（以 K210 平台为例）：

```
[RustSBI output]
[kernel] back to world!
remap_test passed!
init TASK_MANAGER
num_app = 4
power_3 [10000/300000power_5 [10000/210000]
power_5 [20000/210000]
power_5 [30000/210000]

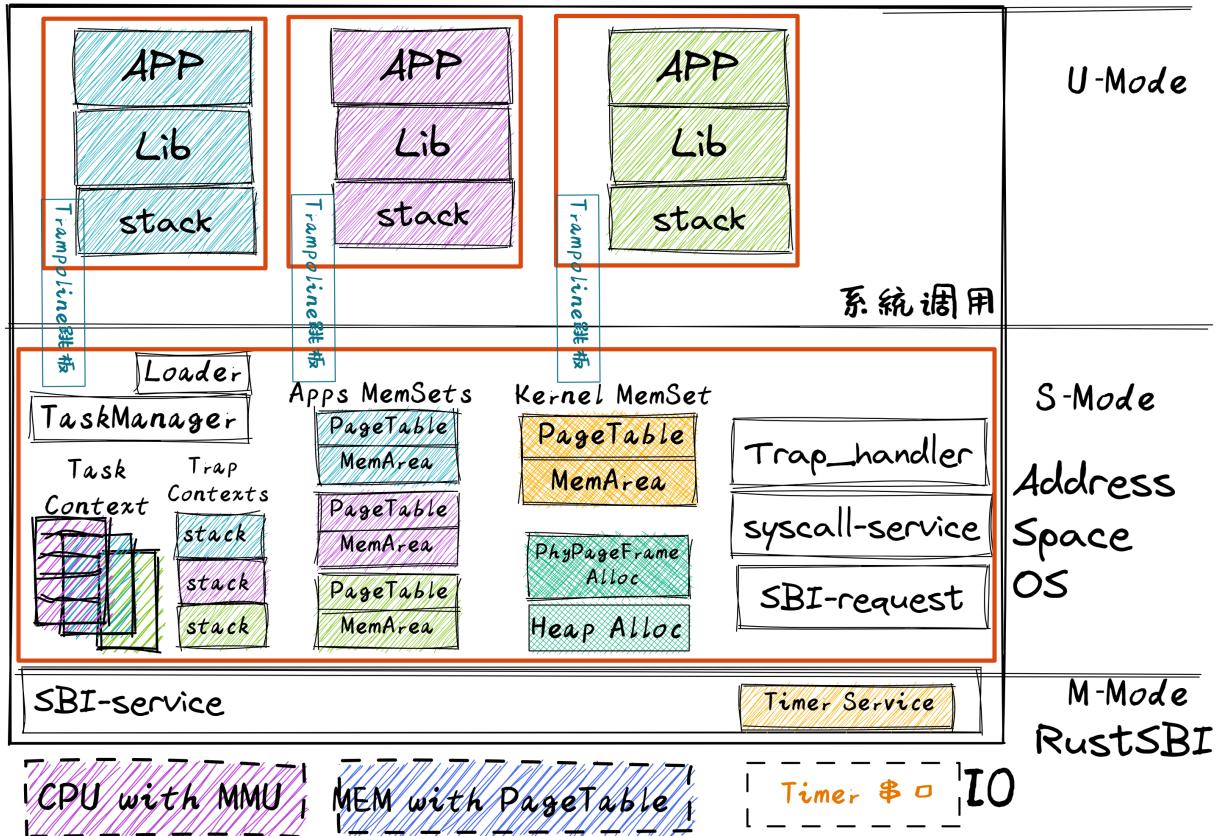
...
(mod 998244353)
Test power_7 OK!
[kernel] Application exited with code 0
power_3 [290000/300000]
power_3 [300000/300000]
3^300000 = 612461288(mod 998244353)
Test power_3 OK!
[kernel] Application exited with code 0
Test sleep OK!
[kernel] Application exited with code 0
[kernel] Panicked at src/task/mod.rs:112 All applications completed!
[rustsbi] reset triggered! todo: shutdown all harts on k210; program halt. Type: 0,_
→ reason: 0
```

(下页继续)

(续上页)

### 5.1.3 本章代码树

头甲龙操作系统 - Address Space OS 的总体结构如下图所示：



通过上图，大致可以看出头甲龙操作系统 - Address Space OS 为了提高操作系统和应用程序执行的安全性，增强了内存管理能力，提供了地址空间隔离机制，给 APP 的内存地址空间划界，不能越界访问 OS 和其他 APP。在具体实现上，扩展了 *TaskManager* 的管理范围，每个 *Task* 的上下文 *Task Context* 还包括该任务的地址空间，在切换任务时，也要切换任务的地址空间。新增的内存管理模块主要包括与内核中动态内存分配相关的页帧分配、堆分配，以及表示应用地址空间的 *Apps MemSets* 类型和内核自身地址空间的 *Kernel MemSet* 类型。*MemSet* 类型所包含的页表 *PageTable* 建立了虚实地址映射关系，而另外一个 *MemArea* 表示任务的合法空间范围。

位于 ch4 分支上的头甲龙操作系统 - Address Space OS 的源代码如下所示：

```

1 ./os/src
2 Rust 25 Files 1415 Lines
3 Assembly 3 Files 88 Lines
4
5 └── bootloader
6 └── rustsbi-k210.bin
7 └── rustsbi-qemu.bin
8 └── LICENSE
9 └── os

```

(下页继续)

(续上页)

```

10 └── build.rs
11 └── Cargo.lock
12 └── Cargo.toml
13 └── Makefile
14 └── src
15 ├── config.rs(修改: 新增一些内存管理的相关配置)
16 ├── console.rs
17 ├── entry.asm
18 ├── lang_items.rs
19 ├── link_app.S
20 ├── linker-k210.ld(修改: 将跳板页引入内存布局)
21 ├── linker-qemu.ld(修改: 将跳板页引入内存布局)
22 ├── loader.rs(修改: 仅保留获取应用数量和数据的功能)
23 ├── main.rs(修改)
24 │ └── mm(新增: 内存管理的 mm 子模块)
25 ├── address.rs(物理/虚拟地址/页号的 Rust 抽象)
26 ├── frame_allocator.rs(物理页帧分配器)
27 ├── heap_allocator.rs(内核动态内存分配器)
28 ├── memory_set.rs(引入地址空间 MemorySet 及逻辑段 MemoryArea 等)
29 ├── mod.rs(定义了 mm 模块初始化方法 init)
30 └── page_table.rs(多级页表抽象 PageTable 以及其他内容)
31 ├── sbi.rs
32 ├── sync
33 ├── mod.rs
34 └── up.rs
35 ├── syscall
36 ├── fs.rs(修改: 基于地址空间的 sys_write 实现)
37 ├── mod.rs
38 └── process.rs
39 ├── task
40 ├── context.rs(修改: 构造一个跳转到不同位置的初始任务上下文)
41 ├── mod.rs(修改, 详见文档)
42 ├── switch.rs
43 ├── switch.S
44 └── task.rs(修改, 详见文档)
45 ├── timer.rs
46 └── trap
47 ├── context.rs(修改: 在 Trap 上下文中加入了更多内容)
48 ├── mod.rs(修改: 基于地址空间修改了 Trap 机制, 详见文档)
49 └── trap.S(修改: 基于地址空间修改了 Trap 上下文保存与恢复汇编代码)
50 └── README.md
51 └── rust-toolchain
52 └── tools
53 ├── kflash.py
54 ├── LICENSE
55 ├── package.json
56 ├── README.rst
57 └── setup.py
58 └── user
59 ├── build.py(移除)
60 ├── Cargo.toml
61 ├── Makefile
62 └── src
63 ├── bin
64 │ ├── 00power_3.rs
65 │ └── 01power_5.rs

```

(下页继续)

(续上页)

```

66 | └── 02power_7.rs
67 | └── 03sleep.rs
68 └── console.rs
69 └── lang_items.rs
70 └── lib.rs
71 └── linker.ld(修改: 将所有应用放在各自地址空间中固定的位置)
72 └── syscall.rs

```

### 5.1.4 本章代码导读

本章涉及的代码量相对多了起来，也许同学们不知如何从哪里看起或从哪里开始尝试实验。这里简要介绍一下“头甲龙”操作系统的开发过程。

我们先从简单的地方入手，那当然就是先改进应用程序了。具体而言，主要就是把 `linker.ld` 中应用程序的起始地址都改为 `0x10000`，这是假定我们操作系统能够通过分页机制把不同应用的相同虚地址映射到不同的物理地址中。这样我们写应用就不用考虑应用的物理地址布局的问题，能够以一种更加统一的方式编写应用程序，可以忽略掉一些不必要的细节。

为了能够在内核中动态分配内存，我们的第二步需要在内核增加连续内存分配的功能，具体实现主要集中在 `os/src/mm/heap_allocator.rs` 中。完成这一步后，我们就可以在内核中用到 Rust 的堆数据结构了，如 `Vec`、`Box` 等，这样内核编程就更加灵活了。

操作系统如果要建立页表（构建虚实地址映射关系），首先要能管理整个系统的物理内存，这就需要知道整个计算机系统的物理内存空间的范围，物理内存中哪些区域是空闲可用的，哪些区域放置内核/应用的代码和数据。操作系统内核能够以物理页帧为单位分配和回收物理内存，具体实现主要集中在 `os/src/mm/frame_allocator.rs` 中；也能在虚拟内存中以各种粒度大小来动态分配内存资源，具体实现主要集中在 `os/src/mm/heap_allocator.rs` 中。

页表中的页表项的索引其实是虚拟地址中的虚拟页号，页表项的重要内容是物理地址的物理页帧号。为了能够灵活地在虚拟地址、物理地址、虚拟页号、物理页号之间进行各种转换，在 `os/src/mm/address.rs` 中实现了各种转换函数。

完成上述工作后，基本上就做好了建立页表的前期准备。我们就可以开始建立页表，这主要涉及到页表项的数据结构表示，以及多级页表的起始物理页帧位置和整个所占用的物理页帧的记录。具体实现主要集中在 `os/src/mm/page_table.rs` 中。

一旦使能分页机制，CPU 访问到的地址都是虚拟地址了，那么内核中也将基于虚地址进行虚存访问。所以在给应用添加虚拟地址空间前，内核自己也会建立一个页表，把整块物理内存通过简单的恒等映射（即虚地址映射到对等的物理地址）映射到内核虚拟地址空间中。后续的应用在执行前，也需要操作系统帮助它建立一个虚拟地址空间。这意味着第三章的初级 `task` 将进化到第四章的拥有独立页表的 `task`。虚拟地址空间需要有一个数据结构管理起来，这就是 `MemorySet`，即地址空间这个抽象概念所对应的具象体现。在一个虚拟地址空间中，有代码段，数据段等不同属性且不一定连续的子空间，它们通过一个重要的数据结构 `MapArea` 来表示和管理。围绕 `MemorySet` 等一系列的数据结构和相关操作的实现，主要集中在 `os/src/mm/memory_set.rs` 中。比如内核的页表和虚拟空间的建立在如下代码中：

```

1 // os/src/mm/memory_set.rs
2
3 lazy_static! {
4 pub static ref KERNEL_SPACE: Arc<Mutex<MemorySet>> = Arc::new(Mutex::new(
5 MemorySet::new_kernel()
6));
7 }

```

完成到这里，我们就可以使能分页机制了。且我们应该有更加方便的机制来支持应用运行。在本章之前，都是把应用程序的所有元数据丢弃从而转换成二进制格式来执行，这其实把编译器生成的 ELF 执行文件中大量

有用的信息给去掉了，比如代码段、数据段的各种属性，程序的入口地址等。既然有了给应用运行提供虚拟地址空间的能力，我们就可以利用 ELF 执行文件中的各种信息来灵活构建应用运行所需要的虚拟地址空间。在 `os/src/loader.rs` 中可以看到如何获取一个应用的 ELF 执行文件数据，而在 `os/src/mm/memory_set` 中的 `MemorySet::from_elf` 可以看到如何通过解析 ELF 来创建一个应用地址空间。

为此，操作系统需要扩展任务控制块 `TaskControlBlock` 的管理范围，使得操作系统能管理拥有独立页表和单一虚拟地址空间的应用程序的运行。相关主要的改动集中在 `os/src/task/task.rs` 中。

由于代表应用程序运行的任务和管理应用的操作系统各自有独立的页表和虚拟地址空间，所以在操作系统的设计实现上需要考虑两个挑战。第一个挑战是 **页表切换**。由于系统调用、中断或异常导致的应用程序和操作系统之间的上下文切换不像以前那么简单了，因为在这些处理过程中需要切换页表，相关改进可参看 `os/src/trap/trap.s`。还有就是需要对来自用户态和内核态的异常/中断分别进行处理，相关改进可参看 `os/src/trap/mod.rs` 和 [跳板的实现](#) 中的讲解。

第二个挑战是 **查页表以访问不同地址空间的数据**。在内核地址空间中执行的内核代码常常需要读写应用的地址空间中的数据，这无法简单的通过一次访存来解决，而是需要手动查用户态应用的地址空间的页表，知道用户态应用的虚地址对应的物理地址后，转换成对应的内核态的虚地址，才能访问应用地址空间中的数据。如果访问应用地址空间中的数据跨了多个页，还需要注意处理地址的边界条件。具体可以参考 `os/src/syscall/fs.rs`、`os/src/mm/page_table.rs` 中的 `translated_byte_buffer` 函数的实现。

实现到这，本章的“头甲龙”操作系统应该就可以给应用程序运行提供一个方便且安全的虚拟地址空间了。

## 5.2 Rust 中的动态内存分配

### 5.2.1 本节导读

到目前为止，如果将当前的操作系统内核也看成一个应用，那么其中所有的变量都是被静态分配在内存中的，这样在对空闲内存的动态使用方面缺少灵活性。我们希望能在操作系统中提供动态申请和释放内存的能力，这样就可以加强操作系统对各种以内存为基础的资源分配与管理。

在应用程序的视角中，动态内存分配中的内存，其实就是操作系统管理的“堆（Heap）”。但现在要实现操作系统，那么就需要操作系统自身能提供动态内存分配的能力。如果要实现动态内存分配的能力，需要操作系统提供如下功能：

- 初始时能提供一块大内存空间作为初始的“堆”。在没有分页机制情况下，这块空间是物理内存空间，否则就是虚拟内存空间。
- 提供在堆上分配和释放内存的函数接口。这样函数调用方通过分配内存函数接口得到地址连续的空闲内存块进行读写，也能通过释放内存函数接口回收内存，以备后续的内存分配请求。
- 提供空闲空间管理的连续内存分配算法。相关算法能动态地维护一系列空闲和已分配的内存块，从而有效地管理空闲块。
- （可选）提供建立在堆上的数据结构和操作。有了上述基本的内存分配与释放函数接口，就可以实现类似动态数组、动态字典等空间灵活可变的堆数据结构，提高编程的灵活性。

考虑到我们是用 Rust 来编写操作系统，为了在接下来的一些操作系统的实现功能中进一步释放 Rust 语言的强表达能力来减轻我们的编码负担，本节我们尝试在内核中支持动态内存分配，以能使用 Rust 核心库中各种灵活的动态数据结构，如 `Vec`、`HashMap` 等，且不用考虑这些数据结构的动态内存释放的繁琐操作，充分利用 Rust 语言保证的内存安全能力。

## 5.2.2 静态与动态内存分配

### 静态分配

若在某一时间点观察一个应用的地址空间，可以看到若干个内存块，每一块都对应于一个生命周期尚未结束的变量。这个变量可能是一个局部变量，它来自于正在执行的当前函数调用栈上，即它是被分配在栈上；这个变量也可能是一个全局变量，它一般被分配在数据段中。它们有一个共同点：在编译器编译程序时已经知道这些变量所占的字节大小，于是给它们分配一块固定的内存将它们存储其中，这样变量在栈帧/数据段中的位置就被固定了下来。

这些变量是被 **静态分配** (Static Allocation) 的，这一过程来源于我们在程序中对变量的声明，在编译期由编译器完成。如果应用仅使用静态分配，它可以应付一部分的需求，但是对于其它情况，如某些数据结构需求的内存大小取决于程序的实际运行情况，就不够灵活了。比如，需要将一个文件读到内存进行处理，而且必须将文件一次性完整读进来处理。我们可以选择声明一个栈上的数组（局部变量）或者数据段中的数组（全局变量），作为缓冲区来暂存文件的内容。但我们在编程的时候并不知道待处理的文件大小，只能根据经验将缓冲区的大小设置为某一固定常数。在程序真正运行的时候，如果待处理的文件很小，那么缓冲区多出的部分被浪费掉了；如果待处理的文件很大，应用则无法正常运行。

### 动态分配

如果使用 **动态分配** (Dynamic Allocation) 则可以解决上述问题。应用另外放置了一个大小可以随着应用的运行动态增减的内存空间—堆 (Heap)。同时，应用还要能够将这个堆管理起来，即支持在运行的时候从里面分配一块空间来存放变量，而在变量的生命周期结束之后，这块空间需要被回收以待后面的使用。如果堆的大小固定，那么这其实就是一个连续内存分配问题，同学们可以使用操作系统课上所介绍到的各种连续内存分配算法。一般情况下，应用所依赖的基础系统库（如 Linux 中的 glibc 库等）会直接通过系统调用（如类 Unix 内核提供的 sbrk 系统调用）来向内核请求增加/缩减应用地址空间内堆的大小，之后应用就可以基于基础系统库提供的内存分配/释放函数来获取和释放内存了。应用进行多次不同大小的内存分配和释放操作后，会产生内存空间的浪费，即存在无法被应用使用的空闲内存碎片。

#### 注解：内存碎片

内存碎片是指无法被分配和使用的空闲内存空间。可进一步细分为内碎片和外碎片：

- 内碎片：已被分配出去（属于某个在运行的应用）内存区域，占有这些区域的应用并不使用这块区域，操作系统也无法利用这块区域。
- 外碎片：还没被分配出去（不属于任何在运行的应用）内存空闲区域，由于太小而无法分配给提出申请内存空间的应用。

为何应用开发者在编程中“看不到”内存碎片？这是因为动态内存管理是由更底层的系统标准库来完成的，它能看到并进行管理。而应用开发者只需调用系统标准库提供的内存申请/释放函数接口即可。

鉴于动态分配是一项非常基础的功能，很多高级语言的系统标准库中都实现了它。以 C 语言为例，C 标准库中提供了如下两个动态分配的接口函数：

```
void* malloc (size_t size);
void free (void* ptr);
```

其中，`malloc` 的作用是从堆中分配一块大小为 `size` 字节的空间，并返回一个指向它的指针。而后续不用的时候，将这个指针传给 `free` 即可在堆中回收这块空间。我们通过返回的指针变量来 **间接** 访问堆空间上的数据。事实上，我们在程序中能够 **直接** 看到的变量都是被静态分配在栈或者全局数据段上的，它们大小在编译期已知。比如我们可以把固定大小的指针放到栈（局部变量）或数据段（全局变量）上，然后通过指针来指向运行时才确定的堆空间上的数据，并进行访问。这样就可以通过确定大小的指针来实现对编译时大小不确定的堆数据的访问。

除了可以灵活利用内存之外，动态分配还允许我们以尽可能小的代价灵活调整变量的生命周期。一个局部变量被静态分配在它所在函数的栈帧中，一旦函数返回，这个局部变量的生命周期也就结束了；而静态分配在数据段中的全局变量则是在应用的整个运行期间均存在。动态分配允许我们构造另一种并不一直存在也不绑定于函数调用的变量生命周期：以 C 语言为例，可以说自 `malloc` 拿到指向一个变量的指针到 `free` 将它回收之前的这段时间，这个变量在堆上存在。由于需要跨越函数调用，我们需要作为堆上数据代表的变量在函数间以参数或返回值的形式进行传递，而这些变量一般都很小（如一个指针），其拷贝开销可以忽略。

而动态内存分配的缺点在于：它背后运行着连续内存分配算法，相比静态分配会带来一些额外的开销。如果动态分配非常频繁，可能会产生很多无法使用的内存碎片，甚至可能会成为应用的性能瓶颈。

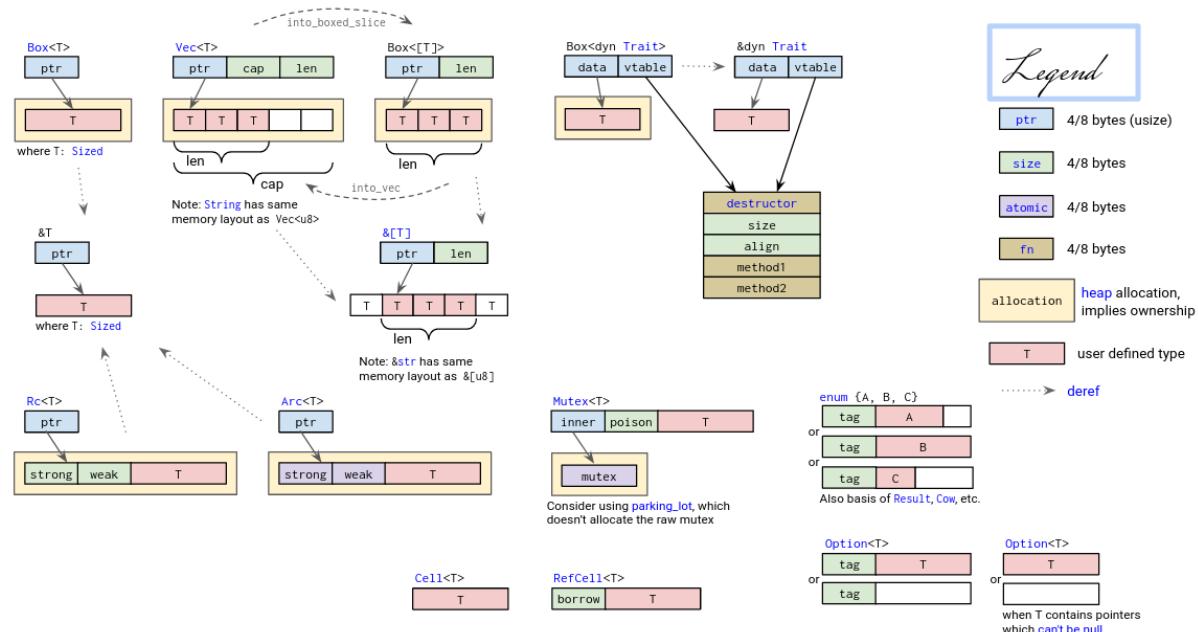
### 5.2.3 Rust 中的堆数据结构

Rust 的标准库中提供了很多开箱即用的堆数据结构，利用它们能够大大提升我们的开发效率。

首先是一类 **智能指针** (Smart Pointer)。智能指针和 Rust 中的其他两类指针：裸指针 `*const T/*mut T` 和引用 `&T/&mut T` 一样，都指向地址空间中的另一个区域并包含它的位置信息。但它们携带的信息数量不等，需要经过编译器不同等级的安全检查，所以它们在可靠性和灵活程度也有所不同。

- 裸指针 `*const T/*mut T` 基本等价于 C/C++ 里面的普通指针 `T*`，它自身的内容仅仅是一个地址。它最为灵活，但是也最不安全。编译器只能对它进行最基本的可变性检查（只读的数据不能写），[第一章](#) 曾经提到，通过裸指针解引用来访问数据的行为是 `unsafe` 行为，需要被包裹在 `unsafe` 块中。
- 引用 `&T/&mut T` 实质上只是一个地址范围，但是 Rust 编译器会在编译的时候进行比较严格的 **借用检查** (Borrow Check)，来确保在编译期就解决掉很多内存不安全问题。详细内容可以参考[Rust 所有权模型](#)。
- 智能指针不仅包含它指向区域的地址范围，还含有一些额外的信息。从用途上看，它不仅可以作为一个媒介来访问它指向的数据，还能在这个过程中起到管理和控制的功能。智能指针的大小通常大于裸指针，这被称作 **胖指针** (Fat Pointer)。如果智能指针仅用堆维护元信息（如针对 `Sized` 类型的 `Arc` 与 `Rc`），那么它们“胖”在堆上，指针本身仍然是 8 字节的；反之，如果指针本身维护元信息（如 `Mutex` 和 `Vec`），指针本身就会大于 8 字节。

具体可以参考下面这张 Rust 智能指针/容器及其他类型的内存布局的 [经典图示](#)。



Rust container cheat sheet, by Raph Levien, Copyright 2017 Google Inc., released under Creative Commons BY, 2017-04-21, version 0.0.4

在 Rust 中，与动态内存分配相关的智能指针主要有如下这些：

- `Box<T>` 在创建时会在堆上分配一个类型为 `T` 的变量，它自身也只保存在堆上的那个变量的位置。而和裸指针或引用不同的是，当 `Box<T>` 被回收的时候，它指向的那个变量（位于堆上）也会被回收。`Box<T>` 可以对标 C++ 的 `std::unique_ptr`。
- `Rc<T>` 是一个单线程上使用的引用计数类型，它提供了多所有权支持，即可同时存在多个智能指针指向同一个堆上变量的 `Rc<T>`，它们都可以拿到指向变量的不可变引用来访问这同一个变量。而它同时也是一个引用计数，事实上在堆上的另一个位置维护了这个变量目前被引用的次数 `N`，即存在 `N` 个 `Rc<T>` 智能指针。这个计数会随着 `Rc<T>` 智能指针的创建或复制而增加，并在 `Rc<T>` 智能指针生命周期结束时减少。当这个计数变为零之后，这个智能指针变量本身以及被引用的变量都会被回收。`Arc<T>` 与 `Rc<T>` 功能相同，只是 `Arc<T>` 可以在多线程上使用。`Arc<T>` 类似于 C++ 的 `std::shared_ptr`。
- `RefCell<T>` 与 `Box<T>` 等智能指针不同，其 **借用检查**在运行时进行。对于 `RefCell<T>`，如果违反借用规则，程序会编译通过，但会在运行时 `panic` 并退出。使用 `RefCell<T>` 的好处是，可在其自身是不可变的情况下修改其内部的值。在 Rust 语言中，在不可变值内部改变值是一种**内部可变性**的设计模式。
- `Mutex<T>` 是一个互斥锁，在多线程中使用。它可以保护里层的堆上的变量同一时间只有一个线程能对它进行操作，从而避免数据竞争，这是并发安全的问题，会在后面详细说明。同时，它也能够提供**内部可变性**。`Mutex<T>` 时常和 `Arc<T>` 配套使用，因为它是用来保护多线程（线程概念在后面会讲，这里可简单理解为运行程序）可同时访问的数据，其前提就是多个线程都拿到指向同一块堆上数据的 `Mutex<T>`。于是，要么这个 `Mutex<T>` 作为全局变量被分配到数据段上，要么将 `Mutex<T>` 包裹上一层多所有权 `Arc`，变成 `Arc<Mutex<T>>` 这种经典组合结构，让最里层基于泛型 `T` 数据结构的变量可以在线程间安全传递。

在讲解**同步互斥**之前我们通过 `RefCell<T>` 来获得内部可变性。可以将 `Mutex<T>` 看成 `RefCell<T>` 的多线程版本，因为 `RefCell<T>` 是只能在单线程上使用的。而且 `RefCell<T>` 并不会在堆上分配内存，它仅用于基于数据段的静态内存分配。

基于上述智能指针，可形成更强大的**集合 (Collection)** 或称**容器 (Container)** 类型，它们负责管理一组数目可变的元素，这些元素的类型相同或是有着一些同样的特征。在 C++/Python/Java 等高级语言中我们已经对它们的使用方法非常熟悉了，对于 Rust 而言，我们可以直接使用以下容器：

- 向量 `Vec<T>` 类似于 C++ 中的 `std::vector`；
- 键值对容器 `BTreeMap<K, V>` 类似于 C++ 中的 `std::map`；
- 有序集合 `BTreeSet<T>` 类似于 C++ 中的 `std::set`；
- 链表 `LinkedList<T>` 类似于 C++ 中的 `std::list`；
- 双端队列 `VecDeque<T>` 类似于 C++ 中的 `std::deque`。
- 变长字符串 `String` 类似于 C++ 中的 `std::string`。

有对比才有更深入的理解，让我们先来看其它一些语言使用动态内存的方式：

- C 语言仅支持 `malloc/free` 这一对操作，它们必须恰好成对使用，否则就会出现各种内存错误。比如分配了之后没有回收，则会导致内存泄漏；回收之后再次 `free` 相同的指针，则会造成 Double-Free 问题；又如回收之后再尝试通过指针访问它指向的区域，这属于 Use-After-Free 问题。总之，这样的内存安全问题层出不穷，毕竟人总是会犯错的。
- Python/Java 通过**引用计数 (Reference Counting)** 对所有的对象进行运行时的动态管理，一套**垃圾回收 (GC, Garbage Collection)** 机制会被自动定期触发，每次都会检查所有的对象，如果其引用计数为零则可以将该对象占用的内存从堆上回收以待后续其他的对象使用。这样做完全杜绝了内存安全问题，但是性能开销则很大，而且 GC 触发的时机和每次 GC 的耗时都是无法预测的，还使得软件的执行性能不够确定。
- C++ 的智能指针 (`shared_ptr`、`unique_ptr`、`weak_ptr`、`auto_ptr` 等) 和**资源获取即初始化 (RAII, Resource Acquisition Is Initialization)**，指将一个使用前必须获取的资源的生命周期绑定到一个变量上，变量释放

时，对应的资源也一并释放。) 风格都是致力于解决内存安全问题。但这些编程方式是“建议”而不是“强制”。

可以发现，在动态内存分配方面，Rust 和 C++ 很像，事实上 Rust 有意从 C++ 借鉴了这部分优秀特性，并强制 Rust 编程人员遵守 **借用规则**。以 `Box<T>` 为例，在它被创建的时候，会在堆上分配一块空间保存它指向的数据；而在 `Box<T>` 生命周期结束被回收的时候，堆上的那块空间也会立即被一并回收。这也就是说，我们无需手动回收资源，它和绑定的变量会被自动回收；同时，由于编译器清楚每个变量的生命周期，则变量对应的资源何时被回收是完全可预测的，回收操作的开销也是确定的。在 Rust 中，不限于堆内存，将某种资源的生命周期与一个变量绑定的这种 RAII 的思想无处不在，甚至这种资源可能只是另外一种类型的变量。

## 5.2.4 在内核中支持动态内存分配

如果要在操作系统内核中支持动态内存分配，则需要实现在本节开始介绍的一系列功能：初始化堆、分配/释放内存块的函数接口、连续内存分配算法。相对于 C 语言而言，Rust 语言在 `alloc` crate 中设定了一套简洁规范的接口，只要实现了这套接口，内核就可以很方便地支持动态内存分配了。

上述与堆相关的智能指针或容器都可以在 Rust 自带的 `alloc` crate 中找到。当我们使用 Rust 标准库 `std` 的时候可以不用关心这个 `crate`，因为标准库内已经实现了一套堆管理算法，并将 `alloc` 的内容包含在 `std` 名字空间之下让开发者可以直接使用。然而操作系统内核运行在禁用标准库（即 `no_std`）的裸机平台上，核心库 `core` 也并没有动态内存分配的功能，这个时候就要考虑利用 `alloc` 库定义的接口来实现基本的动态内存分配器。

`alloc` 库需要我们提供给它一个全局的动态内存分配器，它会利用该分配器来管理堆空间，从而使得与堆相关的智能指针或容器数据结构可以正常工作。具体而言，我们的动态内存分配器需要实现它提供的 `GlobalAlloc` Trait，这个 Trait 有两个必须实现的抽象接口：

```
// alloc::alloc::GlobalAlloc

pub unsafe fn alloc(&self, layout: Layout) -> *mut u8;
pub unsafe fn dealloc(&self, ptr: *mut u8, layout: Layout);
```

可以看到，它们类似 C 语言中的 `malloc/free`，分别代表堆空间的分配和回收，也同样使用一个裸指针（也就是地址）作为分配的返回值和回收的参数。两个接口中都有一个 `alloc::alloc::Layout` 类型的参数，它指出了分配的需求，分为两部分，分别是所需空间的大小 `size`，以及返回地址的对齐要求 `align`。这个对齐要求必须是一个 2 的幂次，单位为字节数，限制返回的地址必须是 `align` 的倍数。

### 注解：为何 C 语言 `malloc` 的时候不需要提供对齐需求？

在 C 语言中，所有对齐要求的最大值是一个平台相关的常数（比如 8 bytes），消耗少量内存即可使得每一次分配都符合这个最大的对齐要求。因此也就不再需要区分不同分配的对齐要求了。而在 Rust 中，某些分配的对齐要求的值可能很大，就只能采用更加复杂的方法。

然后只需将我们的动态内存分配器类型实例化为一个全局变量，并使用 `#[global_allocator]` 语义项标记即可。由于该分配器的实现比较复杂，我们这里直接使用一个已有的伙伴分配器实现。首先添加 `crate` 依赖：

```
os/Cargo.toml

buddy_system_allocator = "0.6"
```

接着，需要引入 `alloc` 库的依赖，由于它算是 Rust 内置的 `crate`，我们并不是在 `Cargo.toml` 中进行引入，而是在 `main.rs` 中声明即可：

```
// os/src/main.rs

extern crate alloc;
```

然后，根据 alloc 留好的接口提供全局动态内存分配器：

```
1 // os/src/mm/heap_allocator.rs
2
3 use buddy_system_allocator::LockedHeap;
4 use crate::config::KERNEL_HEAP_SIZE;
5
6 #[global_allocator]
7 static HEAP_ALLOCATOR: LockedHeap = LockedHeap::empty();
8
9 static mut HEAP_SPACE: [u8; KERNEL_HEAP_SIZE] = [0; KERNEL_HEAP_SIZE];
10
11 pub fn init_heap() {
12 unsafe {
13 HEAP_ALLOCATOR
14 .lock()
15 .init(HEAP_SPACE.as_ptr() as usize, KERNEL_HEAP_SIZE);
16 }
17 }
```

- 第 7 行，我们直接将 `buddy_system_allocator` 中提供的 `LockedHeap` 实例化成一个全局变量，并使用 `alloc` 要求的 `#[global_allocator]` 语义项进行标记。注意 `LockedHeap` 已经实现了 `GlobalAlloc` 要求的抽象接口了。
- 第 11 行，在使用任何 `alloc` 中提供的堆数据结构之前，我们需要先调用 `init_heap` 函数来给我们的全局分配器一块内存用于分配。在第 9 行可以看到，这块内存是一个 `static mut` 且被零初始化的字节数组，位于内核的 `.bss` 段中。`LockedHeap` 也是一个被互斥锁 `Mutex<T>` 保护的类型，在对它任何进行任何操作之前都要先获取锁以避免其他线程同时对它进行操作导致数据竞争。然后，调用 `init` 方法告知它能够用来分配的空间的起始地址和大小即可。

我们还需要处理动态内存分配失败的情形，在这种情况下我们直接 `panic`：

```
// os/src/main.rs

#[feature(alloc_error_handler)]

// os/src/mm/heap_allocator.rs

#[alloc_error_handler]
pub fn handle_alloc_error(layout: core::alloc::Layout) -> ! {
 panic!("Heap allocation error, layout = {:?}", layout);
}
```

最后，让我们尝试一下动态内存分配吧！感兴趣的同学可以在 `rust_main` 中尝试调用下面的 `heap_test` 函数（调用 `heap_test()` 前要记得先调用 `init_heap()`）。

```
1 // os/src/mm/heap_allocator.rs
2
3 #[allow(unused)]
4 pub fn heap_test() {
5 use alloc::boxed::Box;
6 use alloc::vec::Vec;
7 extern "C" {
```

(下页继续)

(续上页)

```

8 fn sbss();
9 fn ebss();
10 }
11 let bss_range = sbss as usize..ebss as usize;
12 let a = Box::new(5);
13 assert_eq!(*a, 5);
14 assert!(bss_range.contains(&(a.as_ref() as *const _ as usize)));
15 drop(a);
16 let mut v: Vec<usize> = Vec::new();
17 for i in 0..500 {
18 v.push(i);
19 }
20 for i in 0..500 {
21 assert_eq!(v[i], i);
22 }
23 assert!(bss_range.contains(&(v.as_ptr() as usize)));
24 drop(v);
25 println!("heap_test passed!");
26}

```

其中分别使用智能指针 `Box<T>` 和向量 `Vec<T>` 在堆上分配数据并管理它们，通过 `as_ref` 和 `as_ptr` 方法可以分别看到它们指向的数据的位置，能够确认它们的确在位于 `.bss` 段的堆上。

---

注解：本节部分内容参考自 [BlogOS](#) 的相关章节。

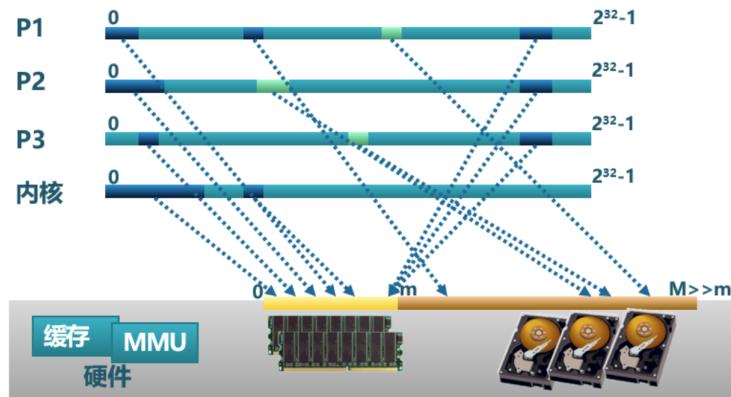
---

## 5.3 地址空间

### 5.3.1 本节导读

直到现在，我们的操作系统给应用看到的是一个非常原始的物理内存空间，可以简单地理解为一个可以随便访问的大数组。为了限制应用访问内存空间的范围并给操作系统提供内存管理的灵活性，计算机硬件引入了各种内存保护/映射/地址转换硬件机制，如 RISC-V 的基址-边界翻译和保护机制、x86 的分段机制、RISC-V/x86/ARM 都有的分页机制。如果在地址转换过程中，无法找到物理地址或访问权限有误，则处理器产生非法访问内存的异常错误。

为了发挥上述硬件机制的能力，操作系统也需要升级自己的能力，更好地管理物理内存和虚拟内存，并给应用程序提供统一的虚拟内存访问接口。计算机科学家观察到这些不同硬件中的共同之处，即 CPU 访问数据和指令的内存地址是虚地址，通过硬件机制（比如 MMU + 页表查询）进行地址转换，找到对应的物理地址。为此，计算机科学家提出了 **地址空间（Address Space）** 抽象，并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。



本节将结合操作系统的发展历程回顾来介绍地址空间抽象的实现策略是如何变化的。

### 5.3.2 虚拟地址与地址空间

#### 地址虚拟化出现之前

我们之前介绍过，在远古计算机时代，整个硬件资源只用来执行单个裸机应用的时候，并不存在真正意义上的操作系统，而只能算是一种应用函数库。那个时候，物理内存的一部分用来保存函数库的代码和数据，余下的部分都交给应用来使用。从功能上可以将应用占据的内存分成几个段：代码段、全局数据段、堆和栈等。当然，由于就只有这一个应用，它想如何调整布局都是它自己的事情。从内存使用的角度来看，批处理系统和裸机应用很相似：批处理系统的每个应用也都是独占内核之外的全部内存空间，只不过当一个应用出错或退出之后，它所占据的内存区域会被清空，而应用序列中的下一个应用将自己的代码和数据放置进来。这个时期，内核提供给应用的访存视角是一致的，因为它们确实在运行过程中始终独占一块固定的内存区域，每个应用开发者都基于这一认知来规划程序的内存布局。

后来，为了降低等待 I/O 带来的无意义的 CPU 资源损耗，多道程序出现了。而为了提升用户的交互式体验，提高生产力，分时多任务系统诞生了。它们的特点在于：应用开始多出了一种“暂停”状态，这可能来源于它主动 yield 交出 CPU 资源，或是在执行了足够长时间之后被内核强制性放弃处理器。当应用处于暂停状态的时候，它驻留在内存中的代码、数据该何去何从呢？曾经有一种省内存的做法是每个应用仍然和在批处理系统中一样独占内核之外的整块内存，当暂停的时候，内核负责将它的代码、数据保存在外存（如硬盘）中，然后把即将换入的应用在外存上的代码、数据恢复到内存，这些都做完之后才能开始执行新的应用。

不过，由于这种做法需要大量读写外部存储设备，而它们的速度都比 CPU 慢上几个数量级，这导致任务切换的开销过大，甚至完全不能接受。既然如此，就只能像我们在第三章中的做法一样，限制每个应用的最大可用内存空间小于物理内存的容量，这样就可以同时把多个应用的数据驻留在内存中。在任务切换的时候只需完成任务上下文保存与恢复即可，这只是在内存的帮助下保存、恢复少量通用寄存器，甚至无需访问外存，这从很大程度上降低了任务切换的开销。

在本章的引言中介绍过第三章中操作系统的做法对应用程序开发带了一定的困难。从应用开发的角度看，需要应用程序决定自己会被加载到哪个物理地址运行，需要直接访问真实的物理内存。这就要求应用开发者对于硬件的特性和使用方法有更多了解，产生额外的学习成本，也会为应用的开发和调试带来不便。从内核的角度来看，将直接访问物理内存的权力下放到应用会使得它难以对应用程序的访存行为进行有效管理，已有的特权级机制亦无法阻止很多来自应用程序的恶意行为。

## 加一层抽象加强内存管理

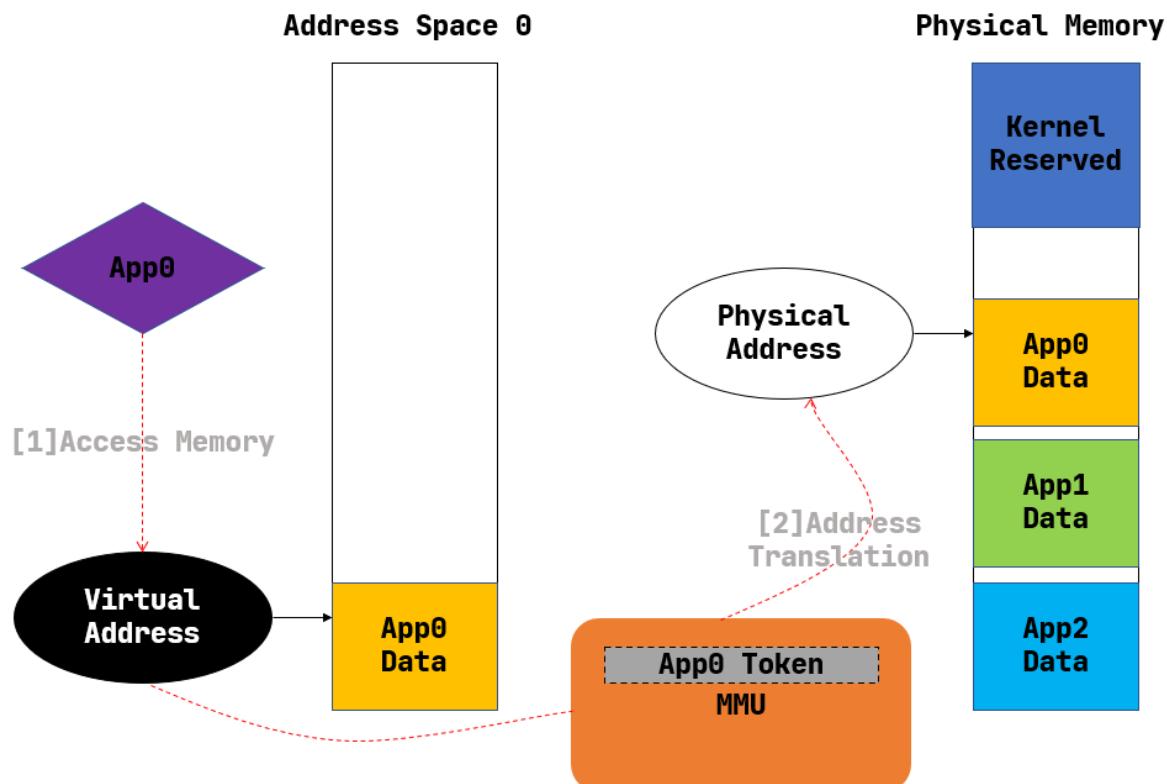
为了解决这种困境，抽象仍然是最重要的指导思想。在这里，抽象意味着内核要负责将物理内存管理起来，并为上面的应用提供一层抽象接口，从之前的失败经验学习，这层抽象需要达成下面的设计目标：

- 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。

最终，到目前为止仍被操作系统内核广泛使用的抽象被称为 **地址空间** (Address Space)。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）。应用同样可以使用一个地址作为索引来读写自己地址空间的数据，就像用物理地址作为索引来读写物理内存上的数据一样。这种地址被称为 **虚拟地址** (Virtual Address)。当然，操作系统要达到地址空间抽象的设计目标，需要有计算机硬件的支持，这就是计算机组成原理课上讲到的 MMU 和 TLB 等硬件机制。

从此，应用能够直接看到并访问的内存就只有操作系统提供的地址空间，且它的任何一次访存使用的地址都是虚拟地址，无论取指令来执行还是读写栈、堆或是全局数据段都是如此。事实上，特权级机制被拓展，使得应用不再具有直接访问物理内存的能力。应用所处的执行环境在安全方面被进一步强化，形成了用户态特权级和地址空间的二维安全措施。

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。



我们知道应用的数据终归还是存在物理内存中的，那么虚拟地址如何形成地址空间，虚拟地址空间如何转

换为物理内存呢？操作系统可以设计巧妙的数据结构来表示地址空间。但如果完全由操作系统来完成转换每次处理器地址访问所需的虚实地址转换，那开销就太大了。这就需要扩展硬件功能来加速地址转换过程（回忆计算机组成原理课上讲的 MMU 和 TLB）。

### 增加硬件加速虚实地址转换

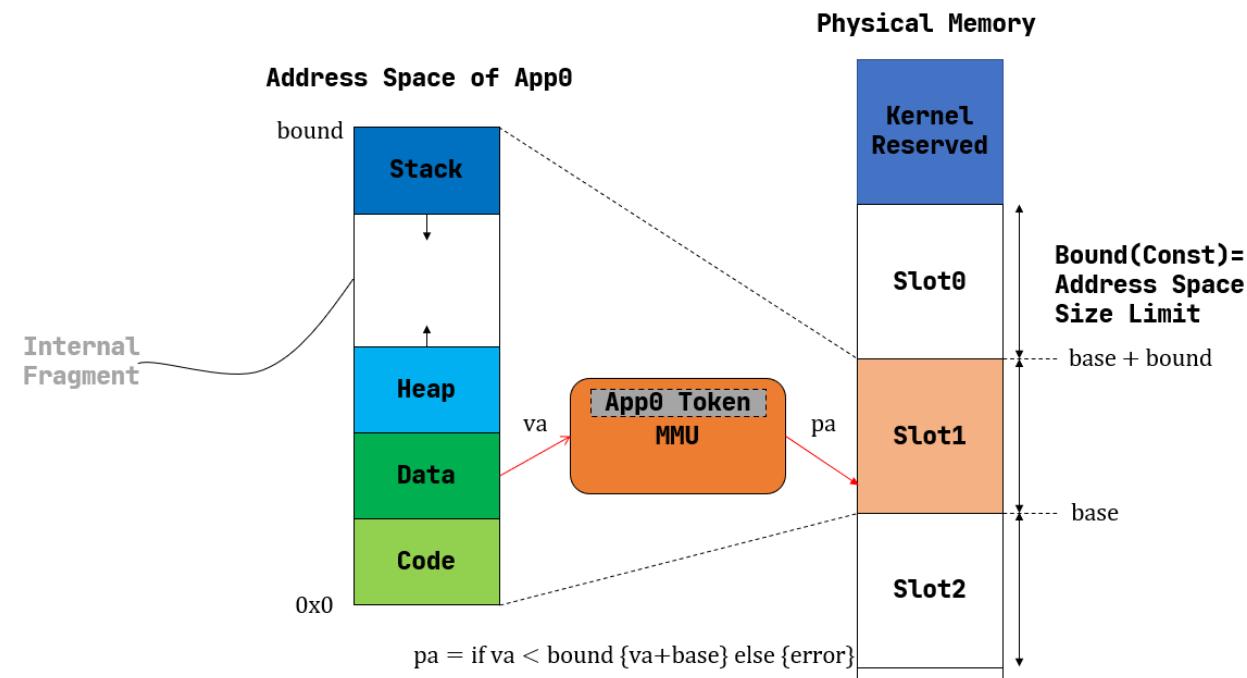
我们回顾一下 **计算机组成原理** 课，如上图所示，当 CPU 取指令或者执行一条访存指令的时候，它都是基于虚拟地址访问属于当前正在运行的应用的地址空间。此时，CPU 中的 **内存管理单元** (MMU, Memory Management Unit) 自动将这个虚拟地址进行 **地址转换** (Address Translation) 变为一个物理地址，即这个应用的数据/指令的物理内存位置。也就是说，在 MMU 的帮助下，应用对自己虚拟地址空间的读写才能被实际转化为对于物理内存的访问。

事实上，每个应用的地址空间都存在一个从虚拟地址到物理地址的映射关系。可以想象对于不同的应用来说，该映射可能是不同的，即 MMU 可能会将来自不同两个应用地址空间的相同虚拟地址转换成不同的物理地址。要做到这一点，就需要硬件提供一些寄存器，软件可以对它进行设置来控制 MMU 按照哪个应用的地址映射关系进行地址转换。于是，将应用的代码/数据放到物理内存并进行管理，建立好应用的地址映射关系，在任务切换时控制 MMU 选用应用的地址映射关系，则是作为软件部分的内核需要完成的重要工作。

回过头来，在介绍内核对于 CPU 资源的抽象——时分复用的时候，我们曾经提到它为应用制造了一种每个应用独占整个 CPU 的幻象，而隐藏了多个应用分时共享 CPU 的实质。而地址空间也是如此，应用只需、也只能看到它独占整个地址空间的幻象，而藏在背后的实质仍然是多个应用共享物理内存，它们的数据分别存放在内存的不同位置。

地址空间只是一层抽象接口，它有很多种具体的实现策略。对于不同的实现策略来说，操作系统内核如何规划应用数据放在物理内存的位置，而 MMU 又如何进行地址转换也都是不同的。下面我们简要介绍几种曾经被使用的策略，并探讨它们的优劣。

### 5.3.3 分段内存管理



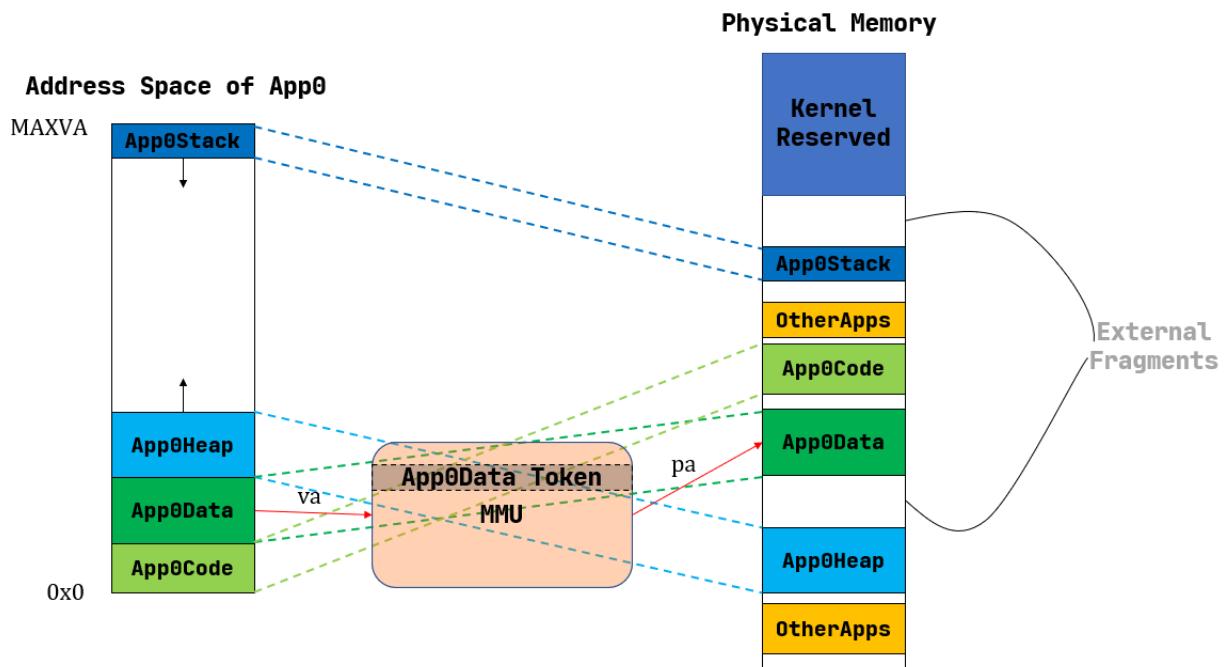
曾经的一种做法如上图所示：每个应用的地址空间大小限制为一个固定的常数 `bound`，也即每个应用的可用虚拟地址区间均为  $[0, bound]$ 。随后，就可以以这个大小为单位，将物理内存除了内核预留空间之外的部

划分分为若干个大小相同的 **插槽** (Slot)，每个应用的所有数据都被内核放置在其中一个插槽中，对应于物理内存上的一段连续物理地址区间，假设其起始物理地址为 `base`，则由于二者大小相同，这个区间实际为  $[base, base + bound]$ 。因此地址转换很容易完成，只需检查一下虚拟地址不超过地址空间的大小限制（此时需要借助特权级机制通过异常来进行处理），然后做一个线性映射，将虚拟地址加上 `base` 就得到了数据实际所在的物理地址。

可以看出，这种实现极其简单：MMU 只需要 `base, bound` 两个寄存器，在地址转换进行比较或加法运算即可；而内核只需要在任务切换时完成切换 `base` 寄存器。在对一个应用的内存管理方面，只需考虑一组插槽的占用状态，可以用一个 **位图** (Bitmap) 来表示，随着应用的新增和退出对应置位或清空。

然而，它的问题在于：可能浪费的内存资源过多。注意到应用地址空间预留了一部分，它是用来让栈得以向低地址增长，同时允许堆往高地址增长（支持应用运行时进行动态内存分配）。每个应用的情况都不同，内核只能按照在它能力范围之内的消耗内存最多的应用的情况来统一指定地址空间的大小，而其他内存需求较低的应用根本无法充分利用内核给他们分配的这部分空间。但这部分空间又是一个完整的插槽的一部分，也不能再交给其他应用使用。这种在已分配/使用的地址空间内部无法被充分利用的空间就是 **内碎片** (Internal Fragment)，它限制了系统同时共存的应用数目。如果应用的需求足够多样化，那么内核无论如何设置应用地址空间的大小限制也不能得到满意的结果。这就是固定参数的弊端：虽然实现简单，但不够灵活。

为了解决这个问题，一种分段管理的策略开始被使用，如下图所示：



注意到内核开始以更细的粒度，也就是应用地址空间中的一个逻辑段作为单位来安排应用的数据在物理内存中的布局。对于每个段来说，从它在某个应用地址空间中的虚拟地址到它被实际存放在内存中的物理地址中间都要经过一个不同的线性映射，于是 MMU 需要用一对不同的 `base/bound` 进行区分。这里由于每个段的大小都是不同的，我们也不再能仅仅使用一个 `bound` 进行简化。当任务切换的时候，这些对寄存器也需要被切换。

简单起见，我们这里忽略一些不必要的细节。比如应用在以虚拟地址为索引访问地址空间的时候，它如何知道该地址属于哪个段，从而硬件可以使用正确的一对 `base/bound` 寄存器进行合法性检查和完成实际的地址转换。这里只关注分段管理是否解决了内碎片带来的内存浪费问题。注意到每个段都只会在内存中占据一块与它实际所用到的大小相等的空间。堆的情况可能比较特殊，它的大小可能会在运行时增长，但是那需要应用通过系统调用向内核请求。也就是说这是一种按需分配，而不再是内核在开始时就给每个应用分配一大块很可能用不完的内存。由此，不再有内碎片了。

尽管内碎片被消除了，但内存浪费问题并没有完全解决。这是因为每个段的大小都是不同的（它们可能来自不同的应用，功能也不同），内核就需要使用更加通用、也更加复杂的连续内存分配算法来进行内存管理，而

不能像之前的插槽那样以一个比特为单位。顾名思义，连续内存分配算法就是每次需要分配一块连续内存来存放一个段的数据。随着一段时间的分配和回收，物理内存还剩下一些相互不连续的较小的可用连续块，其中有一些只是两个已分配内存块之间的很小的间隙，它们自己可能由于空间较小，已经无法被用于分配，这就是 **外碎片** (External Fragment)。

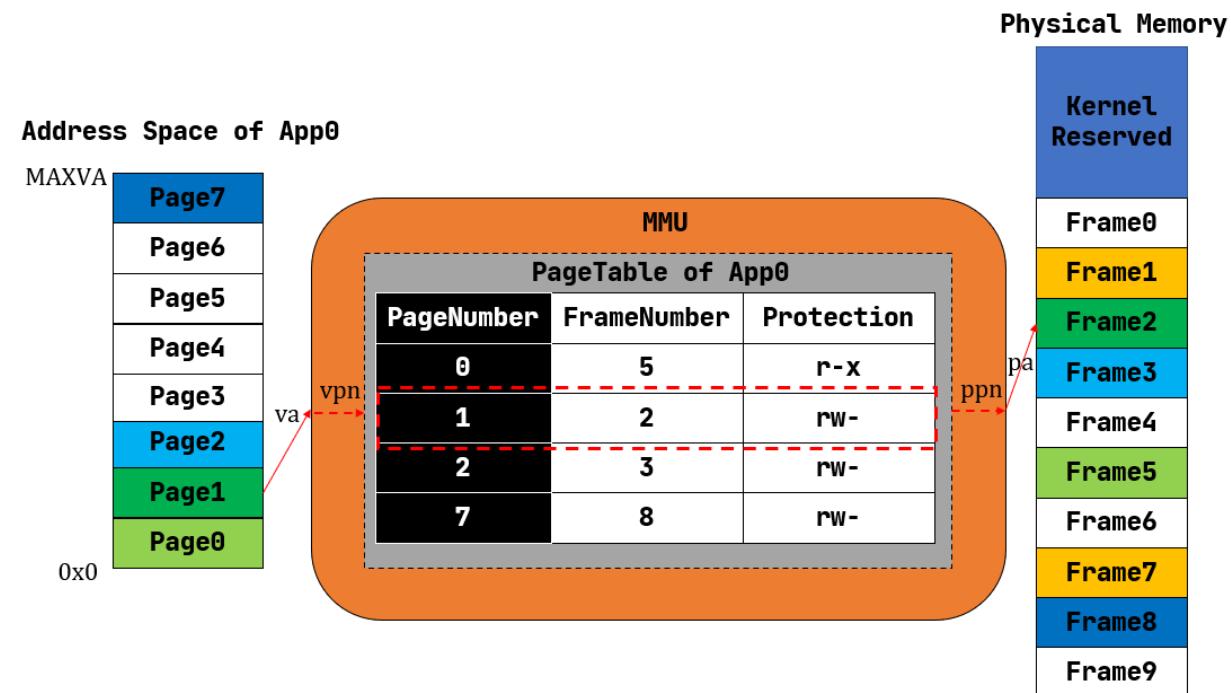
如果这时再想分配一个比较大的块，就需要将这些不连续的外碎片“拼起来”，形成一个大的连续块。然而这是一件开销很大的事情，涉及到极大的内存读写开销。具体而言，这需要移动和调整一些已分配内存块在物理内存上的位置，才能让那些小的外碎片能够合在一起，形成一个大的空闲块。如果连续内存分配算法选取得当，可以尽可能减少这种操作。操作系统课上所讲到的那些算法，包括 first-fit/worst-fit/best-fit 或是 buddy system，其具体表现取决于实际的应用需求，各有优劣。

那么，分段内存管理带来的外碎片和连续内存分配算法比较复杂的问题可否被解决呢？

### 5.3.4 分页内存管理

仔细分析一下可以发现，段的大小不一是外碎片产生的根本原因。之前我们把应用的整个地址空间连续放置在物理内存中，在每个应用的地址空间大小均相同的情况下，只需利用类似位图的数据结构维护一组插槽的占用状态，从逻辑上分配和回收都是以一个固定的比特为单位，自然也就不会存在外碎片了。但是这样粒度过大，不够灵活，又在地址空间内部产生了内碎片。

若要结合二者的优点的话，就需要内核始终以一个同样大小的单位来在物理内存上放置应用地址空间中的数据，这样内核就可以使用简单的插槽式内存管理，使得内存分配算法比较简单且不会产生外碎片；同时，这个单位的大小要足够小，从而其内部没有被用到的内碎片的大小也足够小，尽可能提高内存利用率。这便是我们将要介绍的分页内存管理。



如上图所示，内核以页为单位进行物理内存管理。每个应用的地址空间可以被分成若干个（虚拟）页面 (Page)，而可用的物理内存也同样可以被分成若干个（物理）页帧 (Frame)，虚拟页面和物理页帧的大小相同。每个虚拟页面中的数据实际上都存储在某个物理页帧上。相比分段内存管理，分页内存管理的粒度更小且大小固定，应用地址空间中的每个逻辑段都由多个虚拟页面组成。而且每个虚拟页面在地址转换的过程中都使用与运行的应用绑定的不同的线性映射，而不像分段内存管理那样每个逻辑段都使用一个相同的线性映射。

为了方便实现虚拟页面到物理页帧的地址转换，我们给每个虚拟页面和物理页帧一个编号，分别称为 **虚拟页号** (VPN, Virtual Page Number) 和 **物理页号** (PPN, Physical Page Number)。每个应用都有一个表示地址映射关系的 **页表** (Page Table)，里面记录了该应用地址空间中的每个虚拟页面映射到物理内存中的哪个物理页帧，即数据实际被内核放在哪里。我们可以用页号来代表二者，因此如果将页表看成一个键值对，其键的类型为虚拟页号，值的类型则为物理页号。当 MMU 进行地址转换的时候，虚拟地址会分为两部分（虚拟页号，页内偏移），MMU 首先找到虚拟地址所在虚拟页面的页号，然后查当前应用的页表，根据虚拟页号找到物理页号；最后按照虚拟地址的页内偏移，给物理页号对应的物理页帧的起始地址加上一个偏移量，这就得到了实际访问的物理地址。

在页表中，还针对虚拟页号设置了一组保护位，它限制了应用对转换得到的物理地址对应的内存的使用方式。最典型的如 `rwx`，`r` 表示当前应用可以读该内存；`w` 表示当前应用可以写该内存；`x` 则表示当前应用可以从该内存取指令用来执行。一旦违反了这种限制则会触发异常，并被内核捕获到。通过适当的设置，可以检查一些应用在运行时的明显错误：比如应用修改只读的代码段，或者从数据段取指令来执行。

当一个应用的地址空间比较大的时候，页表中的项数会很多（事实上每个虚拟页面都应该对应页表中的一项，上图中我们已经省略掉了那些未被使用的虚拟页面），导致它的容量极速膨胀，已经不再是像之前那样数个寄存器便可存下来的了，CPU 内也没有足够的硬件资源能够将它存下来。因此它只能作为一种被内核管理的数据结构放在内存中，但是 CPU 也会直接访问它来查页表，这也就需要内核和硬件之间关于页表的内存布局达成一致。

由于分页内存管理既简单又灵活，它逐渐成为了主流的内存管理机制，RISC-V 架构也使用了这种机制。后面我们会基于这种机制，自己动手从物理内存抽象出应用的地址空间来。

---

**注解：**本节部分内容参考自 *Operating Systems: Three Easy Pieces* 教材的 13~16 小节。

---

## 5.4 SV39 多级页表的硬件机制

### 5.4.1 本节导读

在上一小节中我们已经简单介绍了分页的内存管理策略，现在我们尝试在 RISC-V 64 架构提供的 SV39 分页硬件机制的基础上完成内核中的软件对应实现。由于内容过多，我们将分成两个小节进行讲解。本节主要讲解在 RISC-V 64 架构下的虚拟地址与物理地址的访问属性（可读，可写，可执行等），组成结构（页号，帧号，偏移量等），访问的空间范围，硬件实现地址转换的多级页表访问过程等；以及如何用 Rust 语言来设计有类型的页表项。

### 5.4.2 虚拟地址和物理地址

#### 内存控制相关的 CSR 寄存器

默认情况下 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物理内存。我们可以通过修改 S 特权级的一个名为 `satp` 的 CSR 来启用分页模式，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；而 M 特权级的访存地址，我们可设定是内存的物理地址。

---

**注解：** M 特权级的访存地址被视为一个物理地址还是一个需要经历和 S/U 特权级相同的地址转换的虚拟地址取决于硬件配置，在这里我们不会进一步探讨。

---

|         |                            |                           |   |
|---------|----------------------------|---------------------------|---|
| 63<br>4 | 60 59<br>ASID (WARL)<br>16 | 44 43<br>PPN (WARL)<br>44 | 0 |
|---------|----------------------------|---------------------------|---|

Figure 4.14: RV64 Supervisor address translation and protection register `satp`, for MODE values Bare, Sv39, and Sv48.

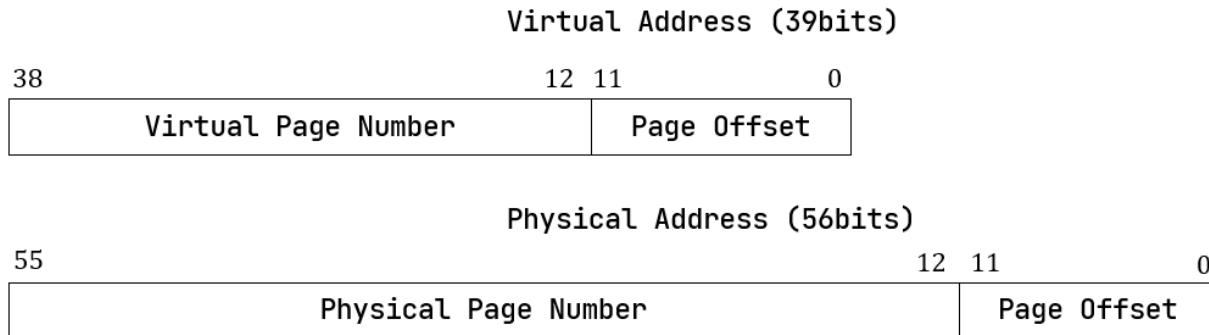
上图是 RISC-V 64 架构下 `satp` 的字段分布, 含义如下:

- MODE 控制 CPU 使用哪种页表实现;
- ASID 表示地址空间标识符, 这里还没有涉及到进程的概念, 我们不需要管这个地方;
- PPN 存的是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一步的将其映射到一个物理页号。

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, 所有 S/U 特权级的访存被视为一个 39 位的虚拟地址, 它们需要先经过 MMU 的地址转换流程, 如果顺利的话, 则会变成一个 56 位的物理地址来访问物理内存; 否则则会触发异常, 这体现了分页机制的内存保护能力。

39 位的虚拟地址可以用来访问理论上最大 512GiB 的地址空间, 而 56 位的物理地址在理论上甚至可以访问一块大小比这个地址空间的还高出几个数量级的物理内存。但是实际上无论是虚拟地址还是物理地址, 真正有意义、能够通过 MMU 的地址转换或是 CPU 内存控制单元的检查的地址仅占其中的很小一部分, 因此它们的理论容量上限在目前都没有实际意义。

## 地址格式与组成



我们采用分页管理, 单个页面的大小设置为 4KiB, 每个虚拟页面和物理页帧都对齐到这个页面大小, 也就是说虚拟/物理地址区间 [0, 4KiB) 为第 0 个虚拟页面/物理页帧, 而 [4KiB, 8KiB) 为第 1 个, 以此类推。4KiB 需要用 12 位字节地址来表示, 因此虚拟地址和物理地址都被分成两部分: 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它所在页面中的相对位置。而虚拟地址的高 27 位, 即 [38 : 12] 为它的虚拟页号 VPN, 同理物理地址的高 44 位, 即 [55 : 12] 为它的物理页号 PPN, 页号可以用来定位一个虚拟/物理地址属于哪一个虚拟页面/物理页帧。

地址转换是以页为单位进行的, 在地址转换的前后地址的页内偏移部分不变。可以认为 MMU 只是从虚拟地址中取出 27 位虚拟页号, 在页表中查到其对应的物理页号 (如果存在的话), 最后将得到的 44 位的物理页号与虚拟地址的 12 位页内偏移依序拼接到一起就变成了 56 位的物理地址。

### 注解: RISC-V 64 架构中虚拟地址为何只有 39 位?

在 64 位架构上虚拟地址长度确实应该和位宽一致为 64 位, 但是在启用 SV39 分页模式下, 只有低 39 位是真正有意义的。SV39 分页模式规定 64 位虚拟地址的 [63 : 39] 这 25 位必须和第 38 位相同, 否则 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。

也就是说，所有  $2^{64}$  个虚拟地址中，只有最低的 256GiB（当第 38 位为 0 时）以及最高的 256GiB（当第 38 位为 1 时）是可能通过 MMU 检查的。当我们写软件代码的时候，一个地址的位宽毋庸置疑就是 64 位，我们要清楚可用的只有最高和最低这两部分，尽管它们已经巨大的超乎想象了；而本节中我们专注于介绍 MMU 的机制，强调 MMU 看到的真正用来地址转换的虚拟地址只有 39 位。

## 地址相关的数据结构抽象与类型定义

正如本章第一小节所说，在分页内存管理中，地址转换的核心任务在于如何维护虚拟页号到物理页号的映射——也就是页表。不过在具体实现它之前，我们先将地址和页号的概念抽象为 Rust 中的类型，借助 Rust 的类型安全特性来确保它们被正确实现。

首先是这些类型的定义：

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

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

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

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

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

上面分别给出了物理地址、虚拟地址、物理页号、虚拟页号的 Rust 类型声明，它们都是 Rust 的元组式结构体，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，就是为了在 Rust 编译器的帮助下，通过多种方便且安全的 **类型转换** (Type Conversion) 来构建页表。

首先，这些类型本身可以和 `usize` 之间互相转换，以物理地址 `PhysAddr` 为例，我们需要：

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

const PA_WIDTH_SV39: usize = 56;
const PPN_WIDTH_SV39: usize = PA_WIDTH_SV39 - PAGE_SIZE_BITS;

impl From<usize> for PhysAddr {
 fn from(v: usize) -> Self { Self(v & ((1 << PA_WIDTH_SV39) - 1)) }
}

impl From<usize> for PhysPageNum {
 fn from(v: usize) -> Self { Self(v & ((1 << PPN_WIDTH_SV39) - 1)) }
}

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

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

前者允许我们从一个 `usize` 来生成 `PhysAddr`，即 `PhysAddr::from(_: usize)` 将得到一个 `PhysAddr`。注意 SV39 支持的物理地址位宽为 56 位，因此在生成 `PhysAddr` 的时候我们仅使用 `usize` 较低的 56 位。同理在生成虚拟地址 `VirtAddr` 的时候仅使用 `usize` 较低的 39 位。反过来，从 `PhysAddr` 等类型也很容

易得到对应的 `usize`。其实由于我们在声明结构体的时候将字段公开了出来，从物理地址变量 `pa` 得到它的 `usize` 表示的更简便方法是直接 `pa.0`。

### 注解: Rust Tips: 类型转换之 From 和 Into

一般而言，当我们为类型 `U` 实现了 `From<T>` Trait 之后，可以使用 `U::from(_: T)` 来从一个 `T` 类型的实例来构造一个 `U` 类型的实例；而当我们为类型 `U` 实现了 `Into<T>` Trait 之后，对于一个 `U` 类型的实例 `u`，可以使用 `u.into()` 来将其转化为一个类型为 `T` 的实例。

当我们为 `U` 实现了 `From<T>` 之后，Rust 会自动为 `T` 实现 `Into<U>` Trait，因为它们两个本来就是在做相同的事情。因此我们只需相互实现 `From` 就可以相互 `From/Into` 了。

需要注意的是，当我们使用 `From` Trait 的 `from` 方法来构造一个转换后类型的实例的时候，`from` 的参数已经指明了转换前的类型，因而 Rust 编译器知道该使用哪个实现；而使用 `Into` Trait 的 `into` 方法来将当前类型转化为另一种类型的时候，它并没有参数，因而函数签名中并没有指出要转化为哪一个类型，则我们必须在其它地方 显式指出目标类型。比如，当我们要将 `u.into()` 绑定到一个新变量 `t` 的时候，必须通过 `let t: T` 显式声明 `t` 的类型；又或是将 `u.into()` 的结果作为参数传给某一个函数，那么由于这个函数的函数签名中指出了传入位置的参数的类型，所以 Rust 编译器也就明确知道转换的类型。

请注意，解引用 `Deref` Trait 是 Rust 编译器允许的一种隐式类型转换，而对于大部分类型转换，我们必须手动调用类型转化方法或者是显式给出转换前后的类型。这体现了 Rust 的类型安全特性，在 C/C++ 中并不是如此，比如两个不同的整数/浮点数类型进行二元运算的时候，编译器经常要先进行隐式类型转换使两个操作数类型相同，而后再进行运算，导致了很多数值溢出或精度损失问题。Rust 不会进行这种隐式类型转换，它会在编译期直接报错，提示两个操作数类型不匹配。

其次，地址和页号之间可以相互转换。我们这里仍以物理地址和物理页号之间的转换为例：

```

1 // os/src/mm/address.rs
2
3 impl PhysAddr {
4 pub fn page_offset(&self) -> usize { self.0 & (PAGE_SIZE - 1) }
5 }
6
7 impl From<PhysAddr> for PhysPageNum {
8 fn from(v: PhysAddr) -> Self {
9 assert_eq!(v.page_offset(), 0);
10 v.floor()
11 }
12 }
13
14 impl From<PhysPageNum> for PhysAddr {
15 fn from(v: PhysPageNum) -> Self { Self(v.0 << PAGE_SIZE_BITS) }
16 }
```

其中 `PAGE_SIZE` 为 4096，`PAGE_SIZE_BITS` 为 12，它们均定义在 `config` 子模块中，分别表示每个页面的大小和页内偏移的位宽。从物理页号到物理地址的转换只需左移 12 位即可，但是物理地址需要保证它与页面大小对齐才能通过右移转换为物理页号。

对于不对齐的情况，物理地址不能通过 `From/Into` 转换为物理页号，而是需要通过它自己的 `floor` 或 `ceil` 方法来进行下取整或上取整的转换。

```

// os/src/mm/address.rs
1
2 impl PhysAddr {
3 pub fn floor(&self) -> PhysPageNum { PhysPageNum(self.0 / PAGE_SIZE) }
4 pub fn ceil(&self) -> PhysPageNum { PhysPageNum((self.0 + PAGE_SIZE - 1) / PAGE_
5 SIZE) }
```

(下页继续)

(续上页)

}

我们暂时先介绍这两种最简单的类型转换。

### 5.4.3 页表项的数据结构抽象与类型定义

第一小节中我们提到，在页表中以虚拟页号作为索引不仅能够查到物理页号，还能查到一组保护位，它控制了应用对地址空间每个虚拟页面的访问权限。但实际上还有更多的标志位，物理页号和全部的标志位以某种固定的格式保存在一个结构体中，它被称为 **页表项** (PTE, Page Table Entry)，是利用虚拟页号在页表中查到的结果。

|    |          |        |        |        |     |   |   |   |   |   |   |   |   |
|----|----------|--------|--------|--------|-----|---|---|---|---|---|---|---|---|
| 63 | 54 53    | 28 27  | 19 18  | 10 9   | 8   | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|    | Reserved | PPN[2] | PPN[1] | PPN[0] | RSW | D | A | G | U | X | W | R | V |
| 10 |          | 26     | 9      | 9      | 2   | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

上图为 SV39 分页模式下的页表项，其中 [53 : 10] 这 44 位是物理页号，最低的 8 位 [7 : 0] 则是标志位，它们的含义如下（请注意，为方便说明，下文我们用 **页表项的对应虚拟页面** 来表示索引到一个页表项的虚拟页号对应的虚拟页面）：

- V(Valid): 仅当位 V 为 1 时，页表项才是合法的；
- R(Read)/W(Write)/X(eXecute): 分别控制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): 控制索引到这个页表项的对应虚拟页面是否在 CPU 处于 U 特权级的情况下是否被允许访问；
- G: 暂且不理会；
- A(Accessed): 处理器记录自从页表项上的这一位被清零之后，页表项的对应虚拟页面是否被访问过；
- D(Dirty): 处理器记录自从页表项上的这一位被清零之后，页表项的对应虚拟页面是否被修改过。

除了 G 外的上述位可以被操作系统设置，只有 A 位和 D 位会被处理器动态地直接设置为 1，表示对应的页被访问过或修改过（注：A 位和 D 位能否被处理器硬件直接修改，取决于处理器的具体实现）。让我们先来实现页表项中的标志位 PTEFlags：

```
// os/src/main.rs

#[macro_use]
extern crate bitflags;

// os/src/mm/page_table.rs

use bitflags::*;

bitflags! {
 pub struct PTEFlags: u8 {
 const V = 1 << 0;
 const R = 1 << 1;
 const W = 1 << 2;
 const X = 1 << 3;
 const U = 1 << 4;
 const G = 1 << 5;
 const A = 1 << 6;
 const D = 1 << 7;
 }
}
```

`bitflags` 是一个 Rust 中常用来比特标志位的 crate。它提供了一个 `bitflags!` 宏，如上面的代码段所展示的那样，可以将一个 `u8` 封装成一个标志位的集合类型，支持一些常见的集合运算。它的一些使用细节这里不展开，请同学自行参考它的官方文档。注意，在使用之前我们需要引入该 crate 的依赖：

```
os/Cargo.toml

[dependencies]
bitflags = "1.2.1"
```

接下来我们实现页表项 `PageTableEntry`：

```
// os/src/mm/page_table.rs

1 #[derive(Copy, Clone)]
2 #[repr(C)]
3 pub struct PageTableEntry {
4 pub bits: usize,
5 }
6
7
8
9 impl PageTableEntry {
10 pub fn new(ppn: PhysPageNum, flags: PTEFlags) -> Self {
11 PageTableEntry {
12 bits: ppn.0 << 10 | flags.bits as usize,
13 }
14 }
15
16 pub fn empty() -> Self {
17 PageTableEntry {
18 bits: 0,
19 }
20 }
21 pub fn ppn(&self) -> PhysPageNum {
22 (self.bits >> 10 & ((1usize << 44) - 1)).into()
23 }
24 pub fn flags(&self) -> PTEFlags {
25 PTEFlags::from_bits(self.bits as u8).unwrap()
26 }
}
```

- 第 3 行我们让编译器自动为 `PageTableEntry` 实现 `Copy/Clone Trait`，来让这个类型以值语义赋值/传参的时候不会发生所有权转移，而是拷贝一份新的副本。从这一点来说 `PageTableEntry` 就和 `usize` 一样，因为它也只是后者的一层简单包装，并解释了 `usize` 各个比特段的含义。
- 第 10 行使得我们可以从一个物理页号 `PhysPageNum` 和一个页表项标志位 `PTEFlags` 生成一个页表项 `PageTableEntry` 实例；而第 20 行和第 23 行则实现了分别可以从一个页表项将它们两个取出的方法。
- 第 15 行中，我们也可以通过 `empty` 方法生成一个全零的页表项，注意这隐含着该页表项的 V 标志位为 0，因此它是不合法的。

后面我们还为 `PageTableEntry` 实现了一些辅助函数 (Helper Function)，可以快速判断一个页表项的 V/R/W/X 标志位是否为 1 以 V 标志位的判断为例：

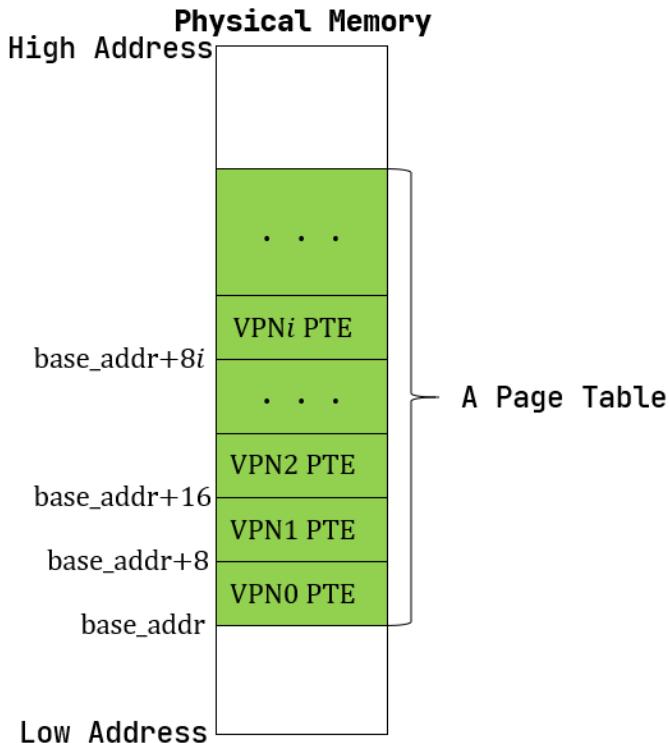
```
// os/src/mm/page_table.rs

impl PageTableEntry {
 pub fn is_valid(&self) -> bool {
 (self.flags() & PTEFlags::V) != PTEFlags::empty()
 }
}
```

这里相当于判断两个集合的交集是否为空集，部分说明了 `bitflags` crate 的使用方法。

#### 5.4.4 多级页表

页表的一种最简单的实现是线性表，也就是按照地址从低到高、输入的虚拟页号从 0 开始递增的顺序依次在内存中（我们之前提到过页表的容量过大无法保存在 CPU 中）放置每个虚拟页号对应的页表项。由于每个页表项的大小是 8 字节，我们只要知道第一个页表项（对应虚拟页号 0）被放在的物理地址 `base_addr`，就能直接计算出每个输入的虚拟页号对应的页表项所在的位置。如下图所示：



事实上，对于虚拟页号  $i$ ，如果页表（每个应用都有一个页表，这里指其中某一个）的起始地址为 `base_addr`，则这个虚拟页号对应的页表项可以在物理地址 `base_addr + 8i` 处找到。这使得 MMU 的实现和内核的软件控制都变得非常简单。然而遗憾的是，这远远超出了我们的物理内存限制。由于虚拟页号有  $2^{27}$  种，每个虚拟页号对应一个 8 字节的页表项，则每个页表都需要消耗掉 1GiB 内存！应用的数据还需要保存在内存的其他位置，这就使得每个应用要吃掉 1GiB 以上的内存。作为对比，我们的 K210 开发板目前只有 8MiB 的内存，因此从空间占用角度来说，这种线性表实现是完全不可行的。

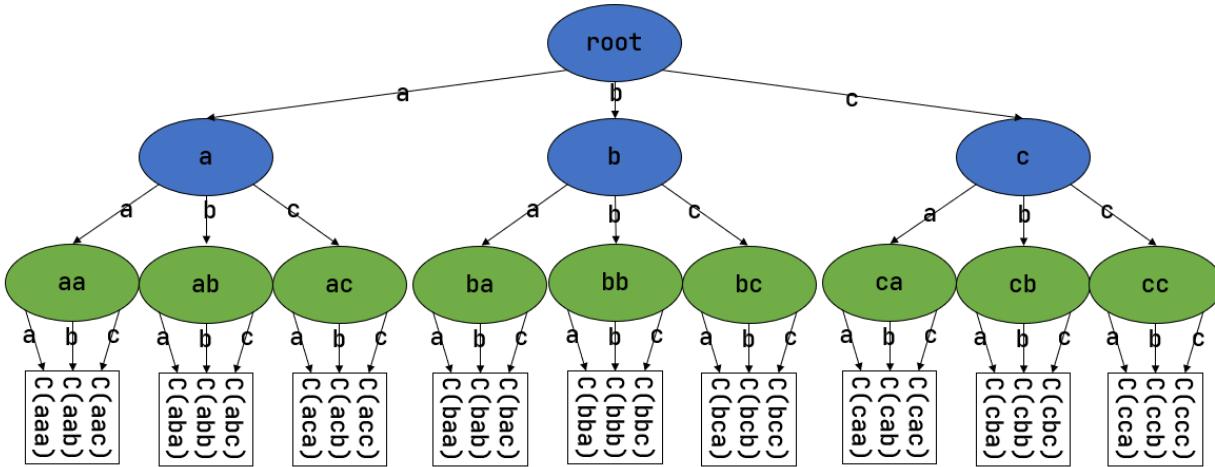
线性表的问题在于：它保存了所有虚拟页号对应的页表项，但是高达 512GiB 的地址空间中真正会被应用使用到的只是其中极小的一个子集（本教程中的应用内存使用量约在数十~数百 KiB 量级），也就导致有意义并能在页表中查到实际的物理页号的虚拟页号在  $2^{27}$  中也只是很小的一部分。由此线性表的绝大部分空间其实都是被浪费掉的。

那么如何进行优化呢？核心思想就在于 **按需分配**，也就是说：有多少合法的虚拟页号，我们就维护一个多大的映射，并为此使用多大的内存用来保存映射。这是因为，每个应用的地址空间最开始都是空的，或者说所有的虚拟页号均不合法，那么这样的页表自然不需要占用任何内存，MMU 在地址转换的时候无需关心页表的内容而是将所有的虚拟页号均判为不合法即可。而在后面，内核已经决定好了一个应用的各逻辑段存放位置之后，它就需要负责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变得合法，反映在该应用的页表上也就是一对对映射顺次被插入进来，自然页表所占据的内存大小也就逐渐增加。

这种 **按需分配** 思想在计算机科学中得到了广泛应用：为了方便接下来的说明，我们可以举一道数据结构的

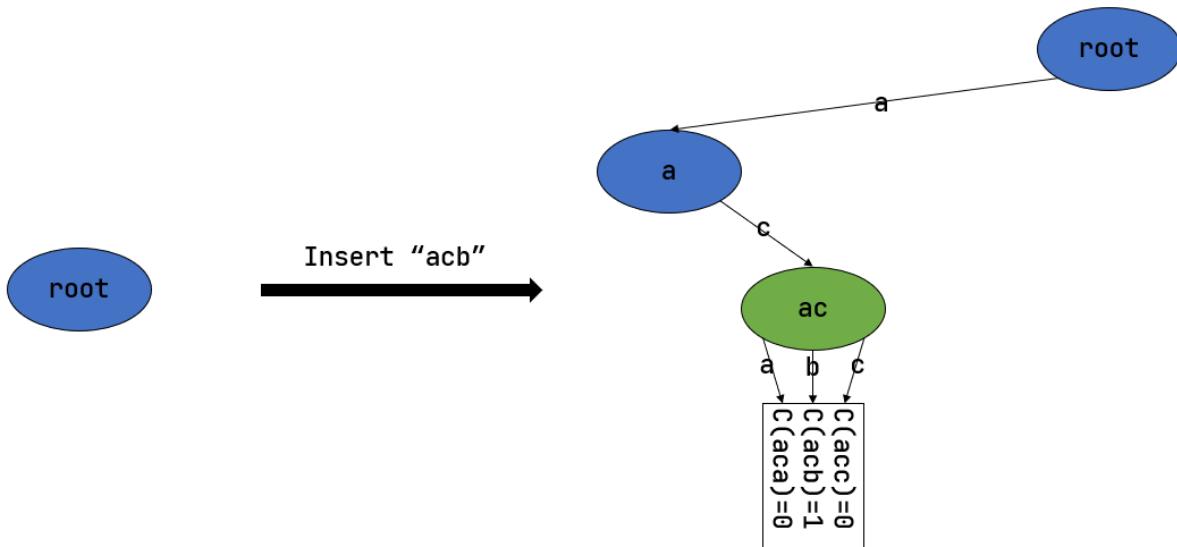
题目作为例子。设想我们要维护一个字符串的多重集，集合中所有的字符串的字符集均为  $\alpha = \{a, b, c\}$ ，长度均为一个给定的常数  $n$ 。该字符串集合一开始为空集。我们要支持两种操作，第一种是将一个字符串插入集合，第二种是查询一个字符串在当前的集合中出现了多少次。

简单起见，假设  $n = 3$ 。那么我们可能会建立这样一颗字典树 (Trie)：



字典树由若干个节点（图中用椭圆形来表示）组成，从逻辑上而言每个节点代表一个可能的字符串前缀。每个节点的存储内容都只有三个指针，对于蓝色的非叶节点来说，它的三个指针各自指向一个子节点；而对于绿色的叶子节点来说，它的三个指针不再指向任何节点，而是具体保存一种可能的长度为  $n$  的字符串的计数。这样，对于题目要求的两种操作，我们只需根据输入的字符串中的每个字符在字典树上自上而下对应走出一步，最终就能够找到字典树中维护的它的计数。之后我们可以将其直接返回或者加一。

注意到如果某些字符串自始至终没有被插入，那么一些节点没有必要。反过来说一些节点是由于我们插入了一个以它对应的字符串为前缀的字符串才被分配出来的。如下图所示：



一开始仅存在一个根节点。在我们插入字符串 acb 的过程中，我们只需要分配 a 和 ac 两个节点。注意 ac 是一个叶节点，它的 b 指针不再指向另外两个节点而是保存字符串 acb 的计数。此时我们无法访问到其他未分配的节点，如根节点的 b/c 或是 a 节点的 a/b 均为空指针。如果后续再插入一个字符串，那么至多分配两个新节点，因为如果走的路径上有节点已经存在，就无需重复分配了。这可以说明，字典树中节点的数

目（或者说字典树消耗的内存）是随着插入字符串的数目逐渐线性增加的。

同学可能很好奇，为何在这里要用相当一部分篇幅来介绍字典树呢？事实上 SV39 分页机制等价于一颗字典树。27 位的虚拟页号可以看成一个长度  $n = 3$  的字符串，字符集为  $\alpha = \{0, 1, 2, \dots, 511\}$ ，因为每一位字符都由 9 个比特组成。而我们也不再维护所谓字符串的计数，而是要找到字符串（虚拟页号）对应的页表项。因此，每个叶节点都需要保存 512 个 8 字节的页表项，一共正好 4KiB，可以直接放在一个物理页帧内。而对于非叶节点来说，从功能上它只需要保存 512 个指向子节点的指针即可，不过我们就像叶节点那样也保存 512 个页表项，这样每个节点都可以被放在一个物理页帧内，节点的位置可以用它所在物理页帧的物理页号来代替。当想从一个非叶节点向下走时，只需找到当前字符对应的页表项的物理页号字段，它就指向了下一级节点的位置，这样非叶节点中转的功能也就实现了。每个节点的内部是一个线性表，也就是将这个节点起始物理地址加上字符对应的偏移量就找到了指向下一级节点的页表项（对于非叶节点）或是能够直接用来地址转换的页表项（对于叶节点）。

这种页表实现被称为 **多级页表** (Multi-Level Page-Table)。由于 SV39 中虚拟页号被分为三级 **页索引** (Page Index)，因此这是一种三级页表。在这种三级页表的树结构中，自上而下分为三种不同的节点：一级/二级/三级页表节点。树的根节点被称为一级页表节点；一级页表节点可以通过一级页索引找到二级页表节点；二级页表节点可以通过二级页索引找到三级页表节点；三级页表节点是树的叶节点，通过三级页索引可以找到一个页表项。

---

**注解：**注意本书将多级页表的根节点称为一级页表，在其他地方则可能以相反的顺序将根节点称为三级页表，这只是表述的习惯不同。

---

非叶节点（页目录表，非末级页表）的表项标志位含义和叶节点（页表，末级页表）相比有一些不同：

- 当  $V$  为 0 的时候，代表当前指针是一个空指针，无法走向下一级节点，即该页表项对应的虚拟地址范围是无效的；
- 只有当  $V$  为 1 且  $R/W/X$  均为 0 时，表示是一个合法的页目录表项，其包含的指针会指向下一级的页表；
- 注意：当  $V$  为 1 且  $R/W/X$  不全为 0 时，表示是一个合法的页表项，其包含了虚地址对应的物理页号。

在这里我们给出 SV39 中的  $R/W/X$  组合的含义：

| X | W | R | Meaning                              |
|---|---|---|--------------------------------------|
| 0 | 0 | 0 | Pointer to next level of page table. |
| 0 | 0 | 1 | Read-only page.                      |
| 0 | 1 | 0 | <i>Reserved for future use.</i>      |
| 0 | 1 | 1 | Read-write page.                     |
| 1 | 0 | 0 | Execute-only page.                   |
| 1 | 0 | 1 | Read-execute page.                   |
| 1 | 1 | 0 | <i>Reserved for future use.</i>      |
| 1 | 1 | 1 | Read-write-execute page.             |

Table 4.4: Encoding of PTE R/W/X fields.

---

**注解：大页 (Huge Page)**

所谓大页就是某些页的大小（如 2MiB, 1GiB）大于常规缺省的页大小（如 4KiB）。本教程中并没有用到大页的知识，这里只是作为拓展，不感兴趣的同學可以跳过。

RISC-V 64 处理器在地址转换过程中，只要表项中的  $V$  为 1 且  $R/W/X$  不全为 0 就会直接从当前的页表项中取出物理页号，再接上页内偏移，就完成最终的地址转换。注意这个过程可以发生在多级页表的任意一级。如

果这一过程并没有发生在多级页表的最深层，那么在地址转换的时候，物理页号对应的物理页帧的起始物理地址的位数与页内偏移的位数都和按缺省页处理时的情况不同了。我们需要按 **大页**的地址转换方式来处理。

这里需要进一步理解将物理页号和页内偏移“接起来”这一行为，它的本质是将物理页号对应的物理页帧的起始物理地址和页内偏移进行求和，物理页帧的起始物理地址是将物理页号左移上页内偏移的位数得到，因此看上去恰好就是将物理页号和页内偏移接在一起。如果在从多级页表往下走的中途停止，未用到的页索引会和虚拟地址的 12 位缺省页内偏移一起形成一个位数更多的 **大页页内偏移**。即对应于一个大页，在转换物理地址的时候，其算法仍是上述二者求和，只是物理页帧的起始物理地址和页内偏移的位数不同了。

在 SV39 中，如果使用了一级页索引就停下来，则它可以涵盖虚拟页号的高 9 位为某一固定值的所有虚拟地址，对应于一个 1GiB 的大页；如果使用了二级页索引就停下来，则它可以涵盖虚拟页号的高 18 位为某一固定值的所有虚拟地址，对应于一个 2MiB 的大页。以同样的视角，如果使用了所有三级页索引才停下来，它可以涵盖虚拟页号的高 27 位为某一个固定值的所有虚拟地址，自然也就对应于一个大小为 4KiB 的虚拟页面。

使用大页的优点在于，当地址空间的大块连续区域的访问权限均相同的时候，可以直接映射一个大页，从时间上避免了大量页表项的读写开销，从空间上降低了所需页表节点的数目。更为重要的是，使用大页可以显著减轻 TLB 的压力，提升 TLB 命中率，因为现在 TLB 中一个表项可以覆盖更大的内存空间了。这可以从整体上提高访存指令的执行速度，进而提升整体的 IPC。但是，从内存分配算法的角度，这需要内核支持从物理内存上分配三种不同大小的连续区域（4KiB 或是另外两种大页），便不能使用更为简单的插槽式管理。权衡利弊之后，本书全程只会以 4KiB 为单位进行页表映射而不会使用大页特性。

那么 SV39 多级页表相比线性表到底能节省多少内存呢？这里直接给出结论：设某个应用地址空间实际用到的区域总大小为  $S$  字节，则地址空间对应的多级页表消耗内存为  $\frac{S}{512}$  左右。下面给出了详细分析，对此不感兴趣的同学可以直接跳过。

### 注解：分析 SV39 多级页表的内存占用

我们知道，多级页表的总内存消耗取决于节点的数目，每个节点需要存放在一个大小为 4KiB 物理页帧中。考虑一个地址空间，除了根节点（即一级页表）占用的一个物理页帧之外，在映射地址空间中的一个实际用到的大小为  $T$  字节的连续区间的时候，最多需要额外分配  $\lceil \frac{T}{1GiB} \rceil$  个二级页表节点和  $\lceil \frac{T}{2MiB} \rceil$  个三级页表节点。这是因为，每个三级页表节点管理地址空间中一块大小为  $4KiB \times 512 = 2MiB$  的区域，每个二级页表节点管理地址空间中一块大小为  $4KiB \times 512^2 = 1GiB$  的区域。因此，每连续映射 1GiB 需要在多级页表中分配一个二级页表节点；每连续映射 2MiB 需要在多级页表中分配一个三级页表节点。无论二级/三级页表都需要占用一个 4KiB 的物理页帧，因此映射  $T$  字节的连续区间需要的额外内存为  $4KiB \times (\lceil \frac{T}{2MiB} \rceil + \lceil \frac{T}{1GiB} \rceil)$ 。相比三级页表，我们可以将二级页表的内存消耗忽略，即省略掉括号中的第二项，最后得到的结果接近于  $\frac{T}{512}$ 。而一般情况下我们对于地址空间的使用方法都是在其中放置少数几个连续的逻辑段，因此当一个地址空间实际使用的区域大小总和为  $S$  字节的时候，我们可以认为多级页表消耗的内存为  $\frac{S}{512}$  左右。相比线性表固定消耗 1GiB 的内存，这已经相当可以接受了。

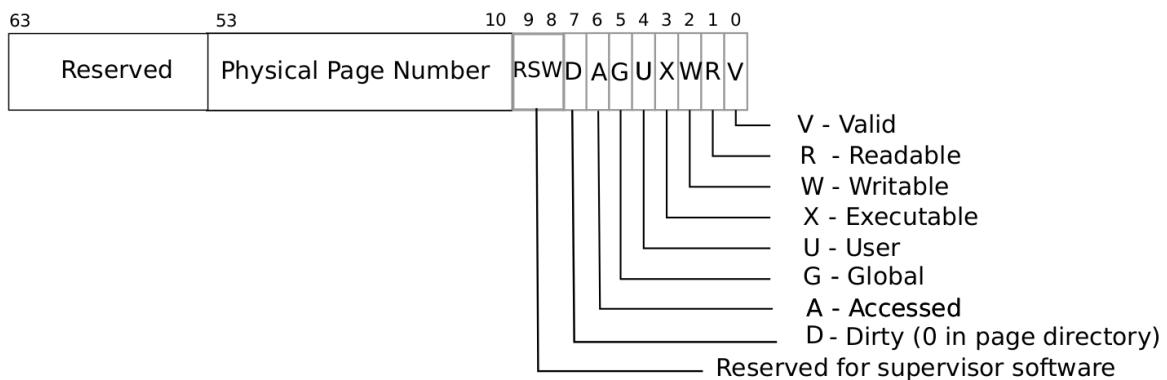
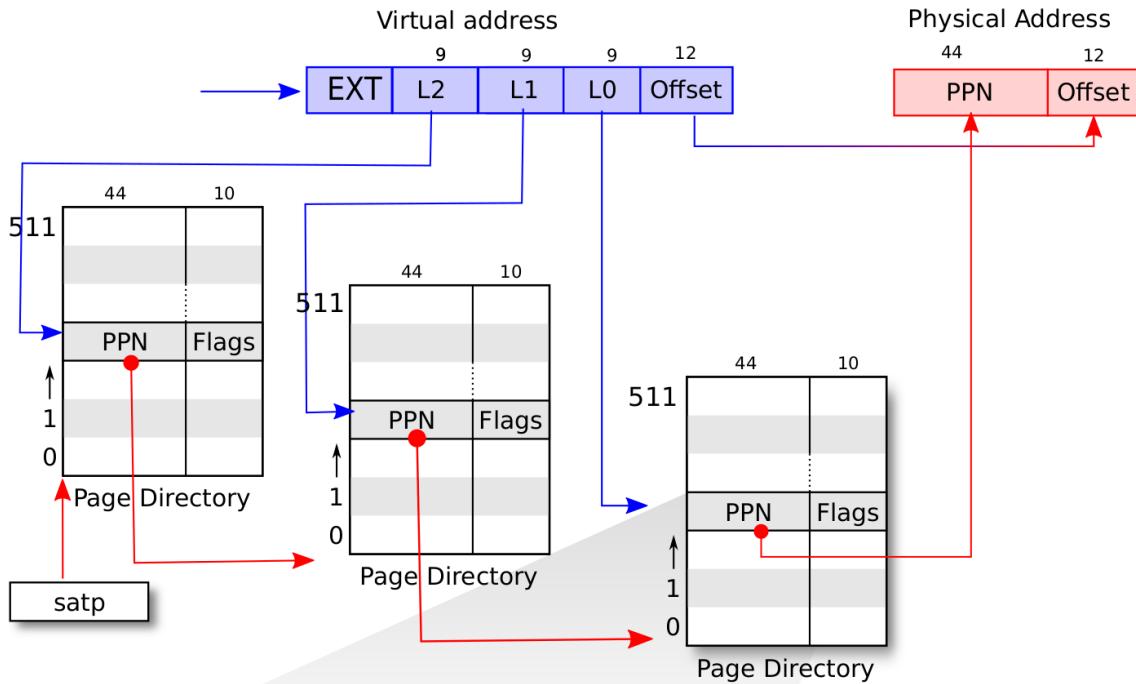
然而，从理论上来说，不妨设某个应用地址空间中的实际用到的总空间大小为  $S$  字节，则对于这个应用的 SV39 多级页表所需的内存量，有两个更加严格的上限：

- 假设目前多级页表中已经分配了一个一级页表节点。接下来，每映射一个 4KiB 的虚拟页面，至多沿着路径额外分配一个二级页表和一个三级页表。之所以是“最多”，是因为路径上经过的二级页表和三级页表可能已经被分配了从而无需重复分配。因此，当总共映射  $S$  字节时，多级页表的总节点数不超过  $1 + 2 \frac{S}{4KiB}$ ，故总消耗内存不超过  $4KiB \times (1 + 2 \frac{S}{4KiB}) = 4KiB + 2S$ ；
- 考虑已经映射了很多虚拟页面，使得根节点（一级页表节点）指向的 512 个二级页表节点都已经被分配的情况（事实上它们不一定都被分配），此时最坏的情况是每次映射都需要分配一个不同的三级页表节点。与上面同理可得：此时消耗内存不超过  $4KiB \times (1 + 512 + \frac{S}{4KiB}) = 4KiB + 2MiB + S$ 。

虽然这两个上限值都可以通过刻意构造一种地址空间的使用来达到，但是它们看起来很不合理，因为它们均大于  $S$ ，也就是元数据比数据还大。其实，真实环境中一般不会有如此极端的使用方式，一般情况下我们知道多级页表消耗内存为  $\frac{S}{512}$  左右就行了。

### 5.4.5 SV39 地址转换过程

接下来，我们给出 SV39 地址转换的全过程图示（来源于 MIT 6.828 课程）来结束多级页表原理的介绍：



在 SV39 模式中我们采用三级页表，即将 27 位的虚拟页号分为三个等长的部分，第 26-18 位为一级页索引  $VPN_0$ ，第 17-9 位为二级页索引  $VPN_1$ ，第 8-0 位为三级页索引  $VPN_2$ 。

我们也将页表分为一级页表（多级页表的根节点），二级页表，三级页表（多级页表的叶节点）。每个页表都用 9 位索引，因此有  $2^9 = 512$  个页表项，而每个页表项都是 8 字节，因此每个页表大小都为  $512 \times 8 = 4\text{KiB}$ 。正好是一个物理页的大小。我们可以把一个页表放到一个物理页中，并用一个物理页号来描述它。事实上，一级页表的每个页表项中的物理页号可描述一个二级页表；二级页表的每个页表项中的物理页号可描述一个三级页表；三级页表中的页表项内容则和我们刚才提到的页表项一样，其内容包含物理页号，即描述一个要映射到的物理页。

具体来说，假设我们有虚拟地址  $(VPN_0, VPN_1, VPN_2, offset)$ ：

- 我们首先会记录装载「当前所用的一级页表的物理页」的页号到 *satp* 寄存器中；

- 把  $VPN_0$  作为偏移在一级页表的物理页中找到二级页表的物理页号；
- 把  $VPN_1$  作为偏移在二级页表的物理页中找到三级页表的物理页号；
- 把  $VPN_2$  作为偏移在三级页表的物理页中找到要访问位置的物理页号；
- 物理页号对应的物理页基址（即物理页号左移 12 位）加上 offset 就是虚拟地址对应的物理地址。

这样处理器通过这种多次转换，终于从虚拟页号找到了一级页表项，从而得出了物理页号和虚拟地址所对应的物理地址。刚才我们提到若页表项满足  $R, W, X$  都为 0，表明这个页表项指向下一级页表。在这里一级和二级页表项的  $R, W, X$  为 0 应该成立，因为它们指向了下一级页表。

## 快表 (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。如果我们按照页表机制循规蹈矩的一步步走，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，才能完成访存。这无疑很大程度上降低了系统执行效率。

实践表明绝大部分应用程序的虚拟地址访问过程具有时间局部性和空间局部性的特点。因此，在 CPU 内部，我们使用 MMU 中的 **快表 (TLB, Translation Lookaside Buffer)** 来作为虚拟页号到物理页号的映射的页表缓存。这部分知识在计算机组成原理课程中有所体现，当我们进行一个地址转换时，会有很大可能对应的地址映射在近期已被完成过，所以我们可以先到 TLB 缓存里面去查一下，如果说有的话我们就可以直接完成映射，而不用访问那么多次内存了。

上面主要是对单个应用的多级页表进行了介绍。在一个多任务系统中，可能同时存在多个任务处于运行/就绪状态，它们各自的多级页表在内存中共存，那么 MMU 应该如何知道当前做地址转换的时候要查哪一个页表呢？回到 `satp CSR` 的布局，其中的 `PPN` 字段指的就是多级页表根节点所在的物理页号。因此，每个应用的地址空间就可以用包含了它多级页表根节点所在物理页号的 `satp CSR` 代表。在我们切换任务的时候，`satp` 也必须被同时切换。

但如果修改了 `satp` 寄存器，说明内核切换到了一个与先前映射方式完全不同的页表。此时快表里面存储的映射已经失效了，这种情况下内核要在修改 `satp` 的指令后面马上使用 `sfence.vma` 指令刷新清空整个 TLB。

同样，我们手动修改一个页表项之后，也修改了映射，但 TLB 并不会自动刷新清空，我们还需要使用 `sfence.vma` 指令刷新整个 TLB。注：可以在 `sfence.vma` 指令后面加上一个虚拟地址，这样 `sfence.vma` 只会刷新 TLB 中关于这个虚拟地址的单个映射项。

## 5.5 管理 SV39 多级页表

### 5.5.1 本节导读

上一节更多的是站在硬件的角度来分析 SV39 多级页表的硬件机制，本节我们主要讲解基于 SV39 多级页表机制的操作系统内存管理。这还需进一步管理计算机系统中当前已经使用的或空闲的物理页帧，这样操作系统才能给应用程序动态分配或回收物理地址空间。有了有效的物理内存空间的管理，操作系统就能在物理内存空间中建立多级页表（页表占用物理内存），为应用程序和操作系统自身建立虚实地址映射关系，从而实现虚拟内存空间，即给应用“看到”的地址空间。

## 5.5.2 物理页帧管理

从前面的介绍可以看出物理页帧的重要性：它既可以用来实际存放应用/内核的数据/代码，也能够用来存储应用/内核的多级页表。当 Bootloader 把操作系统内核加载到物理内存中后，物理内存上已经有一部分用于放置内核的代码和数据。我们需要将剩下的空闲内存以单个物理页帧为单位管理起来，当需要存放应用数据或扩展应用的多级页表时分配空闲的物理页帧，并在应用出错或退出的时候回收应用占有的所有物理页帧。

### 可用物理页的分配与回收

首先，我们需要知道物理内存的哪一部分是可用的。在 `os/src/linker.ld` 中，我们用符号 `ekernel` 指明了内核数据的终止物理地址，在它之后的物理内存都是可用的。而在 `config` 子模块中：

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

pub const MEMORY_END: usize = 0x80800000;
```

我们硬编码整块物理内存的终止物理地址为 `0x80800000`。而之前提到过物理内存的起始物理地址为 `0x80000000`，这意味着我们将可用内存大小设置为 `8MiB`。实际上在 Qemu 模拟器上可以通过设置使用更大的物理内存，但这里我们希望它和真实硬件 K210 的配置保持一致，因此设置为仅使用 `8MiB`。我们用一个左闭右开的物理页号区间来表示可用的物理内存，则：

- 区间的左端点应该是 `ekernel` 的物理地址以上取整方式转化成的物理页号；
- 区间的右端点应该是 `MEMORY_END` 以下取整方式转化成的物理页号。

这个区间将被传给我们后面实现的物理页帧管理器用于初始化。

我们声明一个 `FrameAllocator` Trait 来描述一个物理页帧管理器需要提供哪些功能：

```
// os/src/mm/frame_allocator.rs

trait FrameAllocator {
 fn new() -> Self;
 fn alloc(&mut self) -> Option<PhysPageNum>;
 fn dealloc(&mut self, ppn: PhysPageNum);
}
```

即创建一个物理页帧管理器的实例，以及以物理页号为单位进行物理页帧的分配和回收。

我们实现一种最简单的栈式物理页帧管理策略 `StackFrameAllocator`：

```
// os/src/mm/frame_allocator.rs

pub struct StackFrameAllocator {
 current: usize, // 空闲内存的起始物理页号
 end: usize, // 空闲内存的结束物理页号
 recycled: Vec<usize>,
}
```

其中各字段的含义是：物理页号区间 `[current, end)` 此前均从未被分配出去过，而向量 `recycled` 以后入先出的方式保存了被回收的物理页号（注：我们已经自然的将内核堆用起来了）。

初始化非常简单。在通过 `FrameAllocator` 的 `new` 方法创建实例的时候，只需将区间两端均设为 0，然后创建一个新的向量；而在它真正被使用起来之前，需要调用 `init` 方法将自身的 `[current, end)` 初始化为可用物理页号区间：

```
// os/src/mm/frame_allocator.rs

impl FrameAllocator for StackFrameAllocator {
 fn new() -> Self {
 Self {
 current: 0,
 end: 0,
 recycled: Vec::new(),
 }
 }
}

impl StackFrameAllocator {
 pub fn init(&mut self, l: PhysPageNum, r: PhysPageNum) {
 self.current = l.0;
 self.end = r.0;
 }
}
```

接下来我们来看核心的物理页帧分配和回收如何实现：

```
// os/src/mm/frame_allocator.rs

impl FrameAllocator for StackFrameAllocator {
 fn alloc(&mut self) -> Option<PhysPageNum> {
 if let Some(ppn) = self.recycled.pop() {
 Some(ppn.into())
 } else {
 if self.current == self.end {
 None
 } else {
 self.current += 1;
 Some((self.current - 1).into())
 }
 }
 }
 fn deallocate(&mut self, ppn: PhysPageNum) {
 let ppn = ppn.0;
 // validity check
 if ppn >= self.current || self.recycled
 .iter()
 .find(|&v| *v == ppn)
 .is_some() {
 panic!("Frame ppn={:#x} has not been allocated!", ppn);
 }
 // recycle
 self.recycled.push(ppn);
 }
}
```

- 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，如果说有的话直接弹出栈顶并返回；否则的话我们只能从之前从未分配过的物理页号区间 [current, end) 上进行分配，我们分配它的左端点 current，同时将管理器内部维护的 current 加 1 代表 current 已被分配了。在即将返回的时候，我们使用 into 方法将 usize 转换成了物理页号 PhysPageNum。

注意极端情况下可能出现内存耗尽分配失败的情况：即 recycled 为空且 current == end。为了涵盖这种情况，alloc 的返回值被 Option 包裹，我们返回 None 即可。

- 在回收 `dealloc` 的时候，我们需要检查回收页面的合法性，然后将其压入 `recycled` 栈中。回收页面合法有两个条件：

- 该页面之前一定被分配出去过，因此它的物理页号一定  $< \text{current}$ ；
- 该页面没有正处在回收状态，即它的物理页号不能在栈 `recycled` 中找到。

我们通过 `recycled.iter()` 获取栈上内容的迭代器，然后通过迭代器的 `find` 方法试图寻找一个与输入物理页号相同的元素。其返回值是一个 `Option`，如果找到了就会是一个 `Option::Some`，这种情况说明我们内核其他部分实现有误，直接报错退出。

下面我们来创建 `StackFrameAllocator` 的全局实例 `FRAME_ALLOCATOR`：

```
// os/src/mm/frame_allocator.rs

use crate::sync::UPSafeCell;
type FrameAllocatorImpl = StackFrameAllocator;
lazy_static! {
 pub static ref FRAME_ALLOCATOR: UPSafeCell<FrameAllocatorImpl> = unsafe {
 UPSafeCell::new(FrameAllocatorImpl::new())
 };
}
```

这里我们使用 `UPSafeCell<T>` 来包裹栈式物理页帧分配器。每次对该分配器进行操作之前，我们都需要先通过 `FRAME_ALLOCATOR.exclusive_access()` 拿到分配器的可变借用。

在正式分配物理页帧之前，我们需要将物理页帧全局管理器 `FRAME_ALLOCATOR` 初始化：

```
// os/src/mm/frame_allocator.rs

pub fn init_frame_allocator() {
 extern "C" {
 fn ekernel();
 }
 FRAME_ALLOCATOR
 .exclusive_access()
 .init(PhysAddr::from(ekernel as usize).ceil(), PhysAddr::from(MEMORY_END).
 floor());
}
```

这里我们调用物理地址 `PhysAddr` 的 `floor/ceil` 方法分别下/上取整获得可用的物理页号区间。

## 分配/回收物理页帧的接口

然后是公开给其他内核模块调用的分配/回收物理页帧的接口：

```
// os/src/mm/frame_allocator.rs

pub fn frame_alloc() -> Option<FrameTracker> {
 FRAME_ALLOCATOR
 .exclusive_access()
 .alloc()
 .map(|ppn| FrameTracker::new(ppn))
}

fn frame_dealloc(ppn: PhysPageNum) {
 FRAME_ALLOCATOR
 .exclusive_access()
```

(下页继续)

(续上页)

```

 .dealloc(ppn);
}

```

可以发现, `frame_alloc` 的返回值类型并不是 `FrameAllocator` 要求的物理页号 `PhysPageNum`, 而是将其进一步包装为一个 `FrameTracker`。这里借用了 RAII 的思想, 将一个物理页帧的生命周期绑定到一个 `FrameTracker` 变量上, 当一个 `FrameTracker` 被创建的时候, 我们需要从 `FRAME_ALLOCATOR` 中分配一个物理页帧:

```

// os/src/mm/frame_allocator.rs

pub struct FrameTracker {
 pub ppn: PhysPageNum,
}

impl FrameTracker {
 pub fn new(ppn: PhysPageNum) -> Self {
 // page cleaning
 let bytes_array = ppn.get_bytes_array();
 for i in bytes_array {
 *i = 0;
 }
 Self { ppn }
 }
}

```

我们将分配来的物理页帧的物理页号作为参数传给 `FrameTracker` 的 `new` 方法来创建一个 `FrameTracker` 实例。由于这个物理页帧之前可能被分配过并用做其他用途, 我们在这里直接将这个物理页帧上的所有字节清零。这一过程并不那么显然, 我们后面再详细介绍。

当一个 `FrameTracker` 生命周期结束被编译器回收的时候, 我们需要将它控制的物理页帧回收到 `FRAME_ALLOCATOR` 中:

```

// os/src/mm/frame_allocator.rs

impl Drop for FrameTracker {
 fn drop(&mut self) {
 frame_dealloc(self.ppn);
 }
}

```

这里我们只需为 `FrameTracker` 实现 `Drop Trait` 即可。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` 我们就将它控制的物理页帧回收以供后续使用了。

### 注解: Rust Tips: Drop Trait

Rust 中的 `Drop Trait` 是它的 RAII 内存管理风格可以被有效实践的关键。之前介绍的多种在堆上分配的 Rust 数据结构便都是通过实现 `Drop Trait` 来进行被绑定资源的自动回收的。例如:

- `Box<T>` 的 `drop` 方法会回收它控制的分配在堆上的那个变量;
- `Rc<T>` 的 `drop` 方法会减少分配在堆上的那个引用计数, 一旦变为零则分配在堆上的那个被计数的变量自身也会被回收;
- `UPSafeCell<T>` 的 `exclusive_access` 方法会获取内部数据结构的独占借用权并返回一个 `RefMut<'a, T>` (实际上来自 `RefCell<T>`), 它可以被当做一个 `&mut T` 来使用; 而 `RefMut<'a, T>` 的 `drop` 方法会将独占借用权交出, 从而允许内核内的其他控制流后续对数据结构进行访问。

FrameTracker 的设计也是基于同样的思想，有了它之后我们就不必手动回收物理页帧了，这在编译期就解决了很多潜在的问题。

最后做一个小结：从其他内核模块的视角看来，物理页帧分配的接口是调用 `frame_alloc` 函数得到一个 `FrameTracker`（如果物理内存还有剩余），它就代表了一个物理页帧，当它的生命周期结束之后它所控制的物理页帧将被自动回收。下面是一段演示该接口使用方法的测试程序：

```

1 // os/src/mm/frame_allocator.rs
2
3 #[allow(unused)]
4 pub fn frame_allocator_test() {
5 let mut v: Vec<FrameTracker> = Vec::new();
6 for i in 0..5 {
7 let frame = frame_alloc().unwrap();
8 println!("{:?}", frame);
9 v.push(frame);
10 }
11 v.clear();
12 for i in 0..5 {
13 let frame = frame_alloc().unwrap();
14 println!("{:?}", frame);
15 v.push(frame);
16 }
17 drop(v);
18 println!("frame_allocator_test passed!");
19}

```

如果我们将第 9 行删去，则第一轮分配的 5 个物理页帧都是分配之后在循环末尾就被立即回收，因为循环作用域的临时变量 `frame` 的生命周期在那时结束了。然而，如果我们将它们 `move` 到一个向量中，它们的生命周期便被延长了——直到第 11 行向量被清空的时候，这些 `FrameTracker` 的生命周期才结束，它们控制的 5 个物理页帧才被回收。这种思想我们立即就会用到。

### 5.5.3 多级页表管理

#### 页表基本数据结构与访问接口

我们知道，SV39 多级页表是以节点为单位进行管理的。每个节点恰好存储在一个物理页帧中，它的位置可以用一个物理页号来表示。

```

1 // os/src/mm/page_table.rs
2
3 pub struct PageTable {
4 root_ppn: PhysPageNum,
5 frames: Vec<FrameTracker>,
6 }
7
8 impl PageTable {
9 pub fn new() -> Self {
10 let frame = frame_alloc().unwrap();
11 PageTable {
12 root_ppn: frame.ppn,
13 frames: vec![frame],
14 }
15 }
16 }

```

每个应用的地址空间都对应一个不同的多级页表，这也就意味这不同页表的起始地址（即页表根节点的地址）是不一样的。因此 PageTable 要保存它根节点的物理页号 root\_ppn 作为页表唯一的区分标志。此外，向量 frames 以 FrameTracker 的形式保存了页表所有的节点（包括根节点）所在的物理页帧。这与物理页帧管理模块的测试程序是一个思路，即将这些 FrameTracker 的生命周期进一步绑定到 PageTable 下面。当 PageTable 生命周期结束后，向量 frames 里面的那些 FrameTracker 也会被回收，也就意味着存放多级页表节点的那些物理页帧被回收了。

当我们通过 new 方法新建一个 PageTable 的时候，它只需有一个根节点。为此我们需要分配一个物理页帧 FrameTracker 并挂在向量 frames 下，然后更新根节点的物理页号 root\_ppn。

多级页表并不是被创建出来之后就不再变化的，为了 MMU 能够通过地址转换正确找到应用地址空间中的数据实际被内核放在内存中位置，操作系统需要动态维护一个虚拟页号到页表项的映射，支持插入/删除键值对，其方法签名如下：

```
// os/src/mm/page_table.rs

impl PageTable {
 pub fn map(&mut self, vpn: VirtPageNum, ppn: PhysPageNum, flags: PTEFlags);
 pub fn unmap(&mut self, vpn: VirtPageNum);
}
```

- 通过 map 方法来在多级页表中插入一个键值对，注意这里将物理页号 ppn 和页表项标志位 flags 作为不同的参数传入；
- 通过 unmap 方法来删除一个键值对，在调用时仅需给出作为索引的虚拟页号即可。

在上述操作的过程中，内核需要能访问或修改多级页表节点的内容。即在操作某个多级页表或管理物理页帧的时候，操作系统要能够读写与一个给定的物理页号对应的物理页帧上的数据。这是因为，在多级页表的架构中，每个节点都被保存在一个物理页帧中，一个节点所在物理页帧的物理页号其实是指向该节点的“指针”。

在尚未启用分页模式之前，内核和应用的代码都可以通过物理地址直接访问内存。而在打开分页模式之后，运行在 S 特权级的内核与运行在 U 特权级的应用在访存上都会受到影响，它们的访存地址会被视为一个当前地址空间（satp CSR 给出当前多级页表根节点的物理页号）中的一个虚拟地址，需要 MMU 查相应的多级页表完成地址转换变为物理地址，即地址空间中虚拟地址指向的数据真正被内核放在的物理内存中的位置，然后才能访问相应的数据。此时，如果想要访问一个特定的物理地址 pa 所指向的内存上的数据，就需要构造对应的一个虚拟地址 va，使得当前地址空间的页表存在映射  $va \rightarrow pa$ ，且页表项中的保护位允许这种访问方式。于是，在代码中我们只需访问地址 va，它便会被 MMU 通过地址转换变成 pa，这样我们就做到了在启用分页模式的情况下也能正常访问内存。

这就需要提前扩充多级页表维护的映射，让每个物理页帧的物理页号 ppn，均存在一个对应的虚拟页号 vpn，这需要建立一种映射关系。这里我们采用一种最简单的 **恒等映射** (Identical Mapping)，即对于物理内存上的每个物理页帧，我们都在多级页表中用一个与其物理页号相等的虚拟页号来映射。

### 注解：其他的映射方式

为了达到这一目的还存在其他不同的映射方式，例如比较著名的 **页表自映射** (Recursive Mapping) 等。有兴趣的同学可以进一步参考 [BlogOS 中的相关介绍](#)。

这里需要说明的是，在下一节中我们可以看到，应用和内核的地址空间是隔离的。而直接访问物理页帧的操作只会在内核中进行，应用无法看到物理页帧管理器和多级页表等内核数据结构。因此，上述的恒等映射只需被附加到内核地址空间即可。

## 内核中访问物理页帧的方法

于是，我们来看看在内核中应如何访问一个特定的物理页帧：

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

impl PhysPageNum {
 pub fn get_pte_array(&self) -> &'static mut [PageTableEntry] {
 let pa: PhysAddr = self.clone().into();
 unsafe {
 core::slice::from_raw_parts_mut(pa.0 as *mut PageTableEntry, 512)
 }
 }
 pub fn get_bytes_array(&self) -> &'static mut [u8] {
 let pa: PhysAddr = self.clone().into();
 unsafe {
 core::slice::from_raw_parts_mut(pa.0 as *mut u8, 4096)
 }
 }
 pub fn get_mut<T>(&self) -> &'static mut T {
 let pa: PhysAddr = self.clone().into();
 unsafe {
 (pa.0 as *mut T).as_mut().unwrap()
 }
 }
}
```

我们构造可变引用来直接访问一个物理页号 PhysPageNum 对应的物理页帧，不同的引用类型对应于物理页帧上的一种不同的内存布局，如 get\_pte\_array 返回的是一个页表项定长数组的可变引用，代表多级页表中的一个节点；而 get\_bytes\_array 返回的是一个字节数组的可变引用，可以以字节为粒度对物理页帧上的数据进行访问，前面进行数据清零就用到了这个方法；get\_mut 是个泛型函数，可以获取一个恰好放在一个物理页帧开头的类型为 T 的数据的可变引用。

在实现方面，都是先把物理页号转为物理地址 PhysAddr，然后再转成 usize 形式的物理地址。接着，我们直接将它转为裸指针用来访问物理地址指向的物理内存。在分页机制开启前，这样做自然成立；而开启之后，虽然裸指针被视为一个虚拟地址，但是上面已经提到，基于恒等映射，虚拟地址会映射到一个相同的物理地址，因此在也是成立的。注意，我们在返回值类型上附加了静态生命周期泛型 'static，这是为了绕过 Rust 编译器的借用检查，实质上可以将返回的类型也看成一个裸指针，因为它也只是标识数据存放的位置以及类型。但与裸指针不同的是，无需通过 unsafe 的解引用访问它指向的数据，而是可以像一个正常的可变引用一样直接访问。

---

### 注解：unsafe 真的就是“不安全”吗？

下面是笔者关于 unsafe 一点较为深入的讨论，不感兴趣的可以跳过。

当我们在 Rust 中使用 unsafe 的时候，并不仅仅是为了绕过编译器检查，更是为了告知编译器和其他看到这段代码的程序员：“**我保证这样做是安全的**”。尽管，严格的 Rust 编译器暂时还不能确信这一点。从规范 Rust 代码编写的角度，我们需要尽可能绕过 unsafe，因为如果 Rust 编译器或者一些已有的接口就可以提供安全性，我们当然倾向于利用它们让我们实现的功能仍然是安全的，可以避免一些无谓的心智负担；反之，就只能使用 unsafe，同时最好说明如何保证这项功能是安全的。

这里简要从内存安全的角度来分析一下 PhysPageNum 的 get\_\* 系列方法的实现中 unsafe 的使用。首先需要指出的是，当需要访问一个物理页帧的时候，我们需要从它被绑定到的 FrameTracker 中获得其物理页号 PhysPageNum 随后再调用 get\_\* 系列方法才能访问物理页帧。因此，PhysPageNum 介于 FrameTracker 和物理页帧之间，也可以看做拥有部分物理页帧的所有权。由于 get\_\* 返回的是引用，我们可以尝试检查引用引发的常见问题：第一个问题是 use-after-free 的问题，即是否存在 get\_\* 返回的引用存在期间被引用的物理页帧已被回收的情形；第二个问题则是注意到 get\_\* 返回的是可变引用，那么就需

要考虑对物理页帧的访问读写冲突的问题。

为了解决这些问题，我们在编写代码的时候需要额外当心。对于每一段 unsafe 代码，我们都需要认真考虑它会对其他无论是 unsafe 还是 safe 的代码造成的潜在影响。比如为了避免第一个问题，我们需要保证当完成物理页帧访问之后便立即回收掉 get\_\* 返回的引用，至少使它不能超出 FrameTracker 的生命周期；考虑第二个问题，目前每个 FrameTracker 仅会出现一次（在它所属的进程中），因此它只会出现在一个上下文中，也就不会产生冲突。但是当内核态打开（允许）中断时，或内核支持在单进程中存在多个线程时，情况也许又会发生变化。

当编译器不能介入的时候，我们很难完美的解决这些问题。因此重新设计数据结构和接口，特别是考虑数据的所有权关系，将建模进行转换，使得 Rust 有能力检查我们的设计会是一种更明智的选择。这也可以说为什么要尽量避免使用 unsafe。事实上，我们目前 PhysPageNum::get\_\* 接口并非一个好的设计，如果同学有兴趣可以试着对设计进行改良，让 Rust 编译器帮助我们解决上述与引用相关的问题。

## 建立和拆除虚实地址映射关系

接下来介绍建立和拆除虚实地址映射关系的 map 和 unmap 方法是如何实现的。它们都依赖于一个很重要的过程，即在多级页表中找到一个虚拟地址对应的页表项。找到之后，只要修改页表项的内容即可完成键值对的插入和删除。在寻找页表项的时候，可能出现页表的中间级节点还未被创建的情况，这个时候我们需要手动分配一个物理页帧来存放这个节点，并将这个节点接入到当前的多级页表的某级中。

```

1 // os/src/mm/address.rs
2
3 impl VirtPageNum {
4 pub fn indexes(&self) -> [u32; 3] {
5 let mut vpn = self.0;
6 let mut idx = [0u32; 3];
7 for i in (0..3).rev() {
8 idx[i] = vpn & 511;
9 vpn >>= 9;
10 }
11 idx
12 }
13 }
14
15 // os/src/mm/page_table.rs
16
17 impl PageTable {
18 fn find_pte_create(&mut self, vpn: VirtPageNum) -> Option<&mut PageTableEntry> {
19 let idxs = vpn.indexes();
20 let mut ppn = self.root_ppn;
21 let mut result: Option<&mut PageTableEntry> = None;
22 for i in 0..3 {
23 let pte = &mut ppn.get_pte_array()[idxs[i]];
24 if i == 2 {
25 result = Some(pte);
26 break;
27 }
28 if !pte.is_valid() {
29 let frame = frame_alloc().unwrap();
30 *pte = PageTableEntry::new(frame.ppn, PTEFlags::V);
31 self.frames.push(frame);
32 }
33 ppn = pte.ppn();
34 }
35 }
36 }
```

(下页继续)

(续上页)

```

35 result
36 }
37 fn find_pte(&self, vpn: VirtPageNum) -> Option<&mut PageTableEntry> {
38 let idxs = vpn.indexes();
39 let mut ppn = self.root_ppn;
40 let mut result: Option<&mut PageTableEntry> = None;
41 for i in 0..3 {
42 let pte = &mut ppn.get_pte_array()[idxs[i]];
43 if i == 2 {
44 result = Some(pte);
45 break;
46 }
47 if !pte.is_valid() {
48 return None;
49 }
50 ppn = pte.ppn();
51 }
52 result
53 }
54 }
```

- **VirtPageNum** 的 `indexes` 可以取出虚拟页号的三级页索引，并按照从高到低的顺序返回。注意它里面包裹的 `usize` 可能有 27 位，也有可能有  $64 - 12 = 52$  位，但这里我们是用来在多级页表上进行遍历，因此只取出低 27 位。
- `PageTable::find_pte_create` 在多级页表找到一个虚拟页号对应的页表项的可变引用。如果在遍历的过程中发现有节点尚未创建则会新建一个节点。

变量 `ppn` 表示当前节点的物理页号，最开始指向多级页表的根节点。随后每次循环通过 `get_pte_array` 将取出当前节点的页表项数组，并根据当前级页索引找到对应的页表项。如果当前节点是一个叶节点，那么直接返回这个页表项的可变引用；否则尝试向下走。走不下去的话就新建一个节点，更新作为下级节点指针的页表项，并将新分配的物理页帧移动到向量 `frames` 中方便后续的自动回收。注意在更新页表项的时候，不仅要更新物理页号，还要将标志位 `V` 置 1，不然硬件在查多级页表的时候，会认为这个页表项不合法，从而触发 `Page Fault` 而不能向下走。

- `PageTable::find_pte` 与 `find_pte_create` 的不同在于当找不到合法叶子节点的时候不会新建叶子节点而是直接返回 `None` 即查找失败。因此，它不会尝试对页表本身进行修改，但是注意它返回的参数类型是页表项的可变引用，也即它允许我们修改页表项。从 `find_pte` 的实现还可以看出，即使找到的页表项不合法，还是会将其返回回去而不是返回 `None`。这说明在目前的实现中，页表和页表项是相对解耦合的。

于是，`map/unmap` 就非常容易实现了：

```

// os/src/mm/page_table.rs

impl PageTable {
 pub fn map(&mut self, vpn: VirtPageNum, ppn: PhysPageNum, flags: PTEFlags) {
 let pte = self.find_pte_create(vpn).unwrap();
 assert!(!pte.is_valid(), "vpn {:?} is mapped before mapping", vpn);
 *pte = PageTableEntry::new(ppn, flags | PTEFlags::V);
 }
 pub fn unmap(&mut self, vpn: VirtPageNum) {
 let pte = self.find_pte(vpn).unwrap();
 assert!(pte.is_valid(), "vpn {:?} is invalid before unmapping", vpn);
 *pte = PageTableEntry::empty();
 }
}
```

只需根据虚拟页号找到页表项，然后修改或者直接清空其内容即可。

**警告：**目前的实现方式并不打算对物理页帧耗尽的情形做任何处理而是直接 panic 退出。因此在前面的代码中能够看到很多 unwrap，这种使用方式并不为 Rust 所推荐，只是由于简单起见暂且这样做。

为了方便后面的实现，我们还需要 PageTable 提供一种类似 MMU 操作的手动查页表的方法：

```

1 // os/src/mm/page_table.rs
2
3 impl PageTable {
4 /// Temporarily used to get arguments from user space.
5 pub fn from_token(satp: usize) -> Self {
6 Self {
7 root_ppn: PhysPageNum::from(satp & ((1usize << 44) - 1)),
8 frames: Vec::new(),
9 }
10 }
11 pub fn translate(&self, vpn: VirtPageNum) -> Option<PageTableEntry> {
12 self.find_pte(vpn)
13 .map(|pte| { pte.clone() })
14 }
15 }

```

- 第 5 行的 `from_token` 可以临时创建一个专用来手动查页表的 `PageTable`，它仅有一个从传入的 `satp token` 中得到的多级页表根节点的物理页号，它的 `frames` 字段为空，也即不实际控制任何资源；
- 第 11 行的 `translate` 调用 `find_pte` 来实现，如果能够找到页表项，那么它会将页表项拷贝一份并返回，否则就返回一个 `None`。

之后，当遇到需要查一个特定页表（非当前正处在的地址空间的页表时），便可先通过 `PageTable::from_token` 新建一个页表，再调用它的 `translate` 方法查页表。

小结一下，上一节和本节讲解了如何基于 RISC-V64 的 SV39 分页机制建立多级页表，并实现基于虚存地址空间的内存使用环境。这样，一旦启用分页机制，操作系统和应用都只能在虚拟地址空间中访问数据了，只是操作系统可以通过页表机制来限制应用访问的实际物理内存范围。这就要在后续小节中，进一步看看操作系统内核和应用程序是如何在虚拟地址空间中进行代码和数据访问的。

## 5.6 内核与应用的地址空间

### 5.6.1 本节导读

页表 `PageTable` 只能以页为单位帮助我们维护一个虚拟内存到物理内存的地址转换关系，它本身对于计算机系统的整个虚拟/物理内存空间并没有一个全局的描述和掌控。操作系统通过对不同页表的管理，来完成对不同应用和操作系统自身所在的虚拟内存，以及虚拟内存与物理内存映射关系的全面管理。这种管理是建立在 **地址空间** 的抽象上，用来表明正在运行的应用或内核自身所在执行环境中的可访问的内存空间。本节我们就在内核中通过基于页表的各种数据结构实现地址空间的抽象，并介绍内核和应用的虚拟和物理地址空间中各需要包含哪些内容。

## 5.6.2 实现地址空间抽象

### 逻辑段：一段连续地址的虚拟内存

我们以逻辑段 MapArea 为单位描述一段连续地址的虚拟内存。所谓逻辑段，就是指地址区间中的一段实际可用（即 MMU 通过查多级页表可以正确完成地址转换）的地址连续的虚拟地址区间，该区间内包含的所有虚拟页面都以一种相同的方式映射到物理页帧，具有可读/可写/可执行等属性。

```
// os/src/mm/memory_set.rs

pub struct MapArea {
 vpn_range: VPNRange,
 data_frames: BTreeMap<VirtPageNum, FrameTracker>,
 map_type: MapType,
 map_perm: MapPermission,
}
```

其中 VPNRange 描述一段虚拟页号的连续区间，表示该逻辑段在地址区间中的位置和长度。它是一个迭代器，可以使用 Rust 的语法糖 for-loop 进行迭代。有兴趣的同学可以参考 os/src/mm/address.rs 中它的实现。

### 注解：Rust Tips：迭代器 Iterator

Rust 编程的迭代器模式允许你对一个序列的项进行某些处理。迭代器 (iterator) 是负责遍历序列中的每一项和决定序列何时结束的控制逻辑。对于如何使用迭代器处理元素序列和如何实现 Iterator trait 来创建自定义迭代器的内容，可以参考 [Rust 程序设计语言-中文版第十三章第二节](#)

MapType 描述该逻辑段内的所有虚拟页面映射到物理页帧的同一种方式，它是一个枚举类型，在内核当前的实现中支持两种方式：

```
// os/src/mm/memory_set.rs

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

其中 Identical 表示上一节提到的恒等映射方式；而 Framed 则表示对于每个虚拟页面都有一个新分配的物理页帧与之对应，虚地址与物理地址的映射关系是相对随机的。恒等映射方式主要是用在启用多级页表之后，内核仍能够在虚存地址空间中访问一个特定的物理地址指向的物理内存。

当逻辑段采用 MapType::Framed 方式映射到物理内存的时候，data\_frames 是一个保存了该逻辑段内的每个虚拟页面和它被映射到的物理页帧 FrameTracker 的一个键值对容器 BTreeMap 中，这些物理页帧被用来存放实际内存数据而不是作为多级页表中的中间节点。和之前的 PageTable 一样，这也用到了 RAII 的思想，将这些物理页帧的生命周期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。

MapPermission 表示控制该逻辑段的访问方式，它是页表项标志位 PTEFlags 的一个子集，仅保留 U/R/W/X 四个标志位，因为其他的标志位仅与硬件的地址转换机制细节相关，这样的设计能避免引入错误的标志位。

```
// os/src/mm/memory_set.rs

bitflags! {
```

(下页继续)

(续上页)

```

pub struct MapPermission: u8 {
 const R = 1 << 1;
 const W = 1 << 2;
 const X = 1 << 3;
 const U = 1 << 4;
}
}

```

## 地址空间：一系列有关联的逻辑段

地址空间是一系列有关联的不一定连续的逻辑段，这种关联一般是指这些逻辑段组成的虚拟内存空间与一个运行的程序（目前把一个运行的程序称为任务，后续会称为进程）绑定，即这个运行的程序对代码和数据的直接访问范围限制在它关联的虚拟地址空间之内。这样我们就有任务的地址空间，内核的地址空间等说法了。地址空间使用 `MemorySet` 类型来表示：

```

// os/src/mm/memory_set.rs

pub struct MemorySet {
 page_table: PageTable,
 areas: Vec<MapArea>,
}

```

它包含了该地址空间的多级页表 `page_table` 和一个逻辑段 `MapArea` 的向量 `areas`。注意 `PageTable` 下挂着所有多级页表的节点所在的物理页帧，而每个 `MapArea` 下则挂着对应逻辑段中的数据所在的物理页帧，这两部分合在一起构成了一个地址空间所需的所有物理页帧。这同样是一种 RAI<sup>I</sup> 风格，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。

地址空间 `MemorySet` 的方法如下：

```

1 // os/src/mm/memory_set.rs
2
3 impl MemorySet {
4 pub fn new_bare() -> Self {
5 Self {
6 page_table: PageTable::new(),
7 areas: Vec::new(),
8 }
9 }
10 fn push(&mut self, mut map_area: MapArea, data: Option<&[u8]>) {
11 map_area.map(&mut self.page_table);
12 if let Some(data) = data {
13 map_area.copy_data(&self.page_table, data);
14 }
15 self.areas.push(map_area);
16 }
17 /// Assume that no conflicts.
18 pub fn insert_framed_area(
19 &mut self,
20 start_va: VirtAddr, end_va: VirtAddr, permission: MapPermission
21) {
22 self.push(MapArea::new(
23 start_va,
24 end_va,
25 MapType::Framed,
26 permission,

```

(下页继续)

(续上页)

```

27), None);
28 }
29 pub fn new_kernel() -> Self;
30 /// Include sections in elf and trampoline and TrapContext and user stack,
31 /// also returns user_sp and entry point.
32 pub fn from_elf(elf_data: &[u8]) -> (Self, usize, usize);
33 }

```

- 第 4 行, new\_bare 方法可以新建一个空的地址空间;
- 第 10 行, push 方法可以在当前地址空间插入一个新的逻辑段 map\_area, 如果它是以 Framed 方式映射到物理内存, 还可以可选地在那些被映射到的物理页帧上写入一些初始化数据 data;
- 第 18 行, insert\_framed\_area 方法调用 push, 可以在当前地址空间插入一个 Framed 方式映射到物理内存的逻辑段。注意该方法的调用者要保证同一地址空间内的任意两个逻辑段不能存在交集, 从后面即将分别介绍的内核和应用的地址空间布局可以看出这一要求得到了保证;
- 第 29 行, new\_kernel 可以生成内核的地址空间; 具体实现将在后面讨论;
- 第 32 行, from\_elf 分析应用的 ELF 文件格式的内容, 解析出各数据段并生成对应的地址空间; 具体实现将在后面讨论。

在实现 push 方法在地址空间中插入一个逻辑段 MapArea 的时候, 需要同时维护地址空间的多级页表 page\_table 记录的虚拟页号到页表项的映射关系, 也需要用到这个映射关系来找到向哪些物理页帧上拷贝初始数据。这用到了 MapArea 提供的另外几个方法:

```

// os/src/mm/memory_set.rs

3 impl MapArea {
4 pub fn new(
5 start_va: VirtAddr,
6 end_va: VirtAddr,
7 map_type: MapType,
8 map_perm: MapPermission
9) -> Self {
10 let start_vpn: VirtPageNum = start_va.floor();
11 let end_vpn: VirtPageNum = end_va.ceil();
12 Self {
13 vpn_range: VPNRange::new(start_vpn, end_vpn),
14 data_frames: BTreeMap::new(),
15 map_type,
16 map_perm,
17 }
18 }
19 pub fn map(&mut self, page_table: &mut PageTable) {
20 for vpn in self.vpn_range {
21 self.map_one(page_table, vpn);
22 }
23 }
24 pub fn unmap(&mut self, page_table: &mut PageTable) {
25 for vpn in self.vpn_range {
26 self.unmap_one(page_table, vpn);
27 }
28 }
29 /// data: start-aligned but maybe with shorter length
30 /// assume that all frames were cleared before
31 pub fn copy_data(&mut self, page_table: &PageTable, data: &[u8]) {
32 assert_eq!(self.map_type, MapType::Framed);

```

(下页继续)

(续上页)

```

33 let mut start: usize = 0;
34 let mut current_vpn = self.vpn_range.get_start();
35 let len = data.len();
36 loop {
37 let src = &data[start..len.min(start + PAGE_SIZE)];
38 let dst = &mut page_table
39 .translate(current_vpn)
40 .unwrap()
41 .ppn()
42 .get_bytes_array()[..src.len()];
43 dst.copy_from_slice(src);
44 start += PAGE_SIZE;
45 if start >= len {
46 break;
47 }
48 current_vpn.step();
49 }
50 }
51 }
```

- 第 4 行的 new 方法可以新建一个逻辑段结构体，注意传入的起始/终止虚拟地址会分别被下取整/上取整为虚拟页号并传入迭代器 vpn\_range 中；
- 第 19 行的 map 和第 24 行的 unmap 可以将当前逻辑段到物理内存的映射从传入的该逻辑段所属的地址空间的多级页表中加入或删除。可以看到它们的实现是遍历逻辑段中的所有虚拟页面，并以每个虚拟页面为单位依次在多级页表中进行键值对的插入或删除，分别对应 MapArea 的 map\_one 和 unmap\_one 方法，我们后面将介绍它们的实现；
- 第 31 行的 copy\_data 方法将切片 data 中的数据拷贝到当前逻辑段实际被内核放置在的各物理页帧上，从而在地址空间中通过该逻辑段就能访问这些数据。调用它的时候需要满足：切片 data 中的数据大小不超过当前逻辑段的总大小，且切片中的数据会被对齐到逻辑段的开头，然后逐页拷贝到实际的物理页帧。

从第 36 行开始的循环会遍历每一个需要拷贝数据的虚拟页面，在数据拷贝完成后会在第 48 行通过调用 step 方法，该方法来自于 os/src/mm/address.rs 中为 VirtPageNum 实现的 StepOne Trait，感兴趣的同学可以阅读代码确认其实现。

每个页面的数据拷贝需要确定源 src 和目标 dst 两个切片并直接使用 copy\_from\_slice 完成复制。当确定目标切片 dst 的时候，第 39 行从传入的当前逻辑段所属的地址空间的多级页表中，手动查找迭代到的虚拟页号被映射到的物理页帧，并通过 get\_bytes\_array 方法获取该物理页帧的字节数组型可变引用，最后再获取它的切片用于数据拷贝。

接下来介绍对逻辑段中的单个虚拟页面进行映射/解映射的方法 map\_one 和 unmap\_one。显然它们的实现取决于当前逻辑段被映射到物理内存的方式：

```

1 // os/src/mm/memory_set.rs
2
3 impl MapArea {
4 pub fn map_one(&mut self, page_table: &mut PageTable, vpn: VirtPageNum) {
5 let ppn: PhysPageNum;
6 match self.map_type {
7 MapType::Identical => {
8 ppn = PhysPageNum(vpn.0);
9 }
10 MapType::Framed => {
11 let frame = frame_alloc().unwrap();
12 ppn = frame.ppn;
13 }
14 }
15 }
16 }
```

(下页继续)

(续上页)

```

13 self.data_frames.insert(vpn, frame);
14 }
15 }
16 let pte_flags = PTEFlags::from_bits(self.map_perm.bits).unwrap();
17 page_table.map(vpn, ppn, pte_flags);
18 }
19 pub fn unmap_one(&mut self, page_table: &mut PageTable, vpn: VirtPageNum) {
20 match self.map_type {
21 MapType::Framed => {
22 self.data_frames.remove(&vpn);
23 }
24 _ => {}
25 }
26 page_table.unmap(vpn);
27 }
28 }
```

- 对于第 4 行的 `map_one` 来说，在虚拟页号 `vpn` 已经确定的情况下，它需要知道要将一个怎么样的页表项插入多级页表。页表项的标志位来源于当前逻辑段的类型为 `MapPermission` 的统一配置，只需将其转换为 `PTEFlags`；而页表项的物理页号则取决于当前逻辑段映射到物理内存的方式：
  - 当以恒等映射 `Identical` 方式映射的时候，物理页号就等于虚拟页号；
  - 当以 `Framed` 方式映射时，需要分配一个物理页帧让当前的虚拟页面可以映射过去，此时页表项中的物理页号自然就是这个被分配的物理页帧的物理页号。此时还需要将这个物理页帧挂在逻辑段的 `data_frames` 字段下。

当确定了页表项的标志位和物理页号之后，即可调用多级页表 `PageTable` 的 `map` 接口来插入键值对。

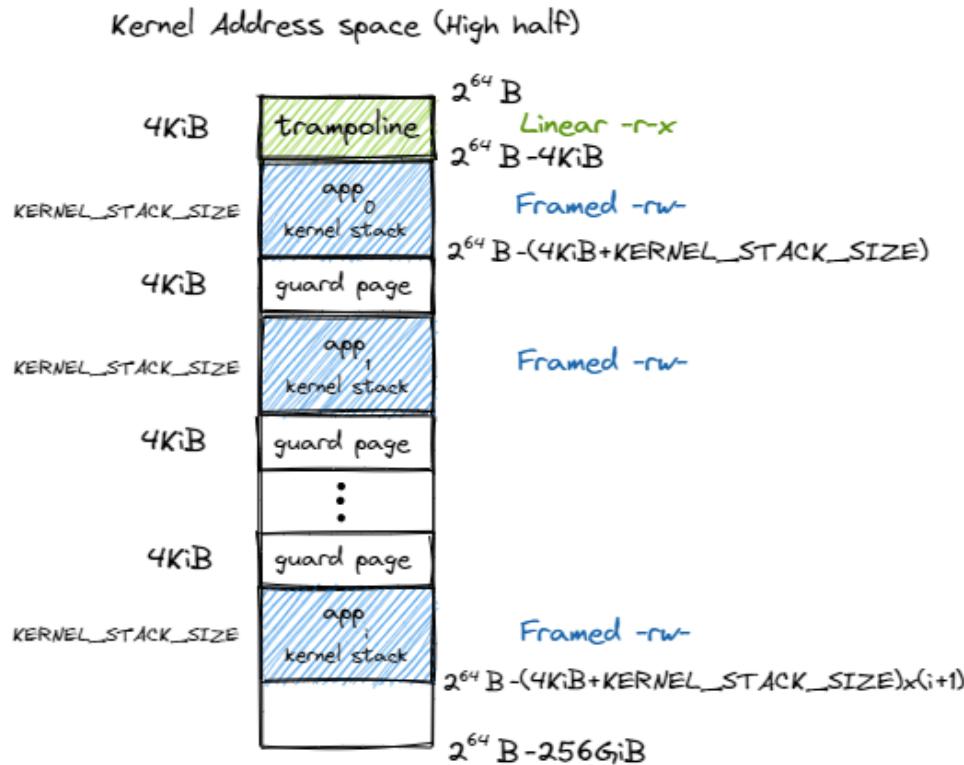
- 对于第 19 行的 `unmap_one` 来说，基本上就是调用 `PageTable` 的 `unmap` 接口删除以传入的虚拟页号为键的键值对即可。然而，当以 `Framed` 映射的时候，不要忘记同时将虚拟页面被映射到的物理页帧 `FrameTracker` 从 `data_frames` 中移除，这样这个物理页帧才能立即被回收以备后续分配。

### 5.6.3 内核地址空间

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)。我们会在本章的后续部分再深入介绍 [跳板的实现](#)。

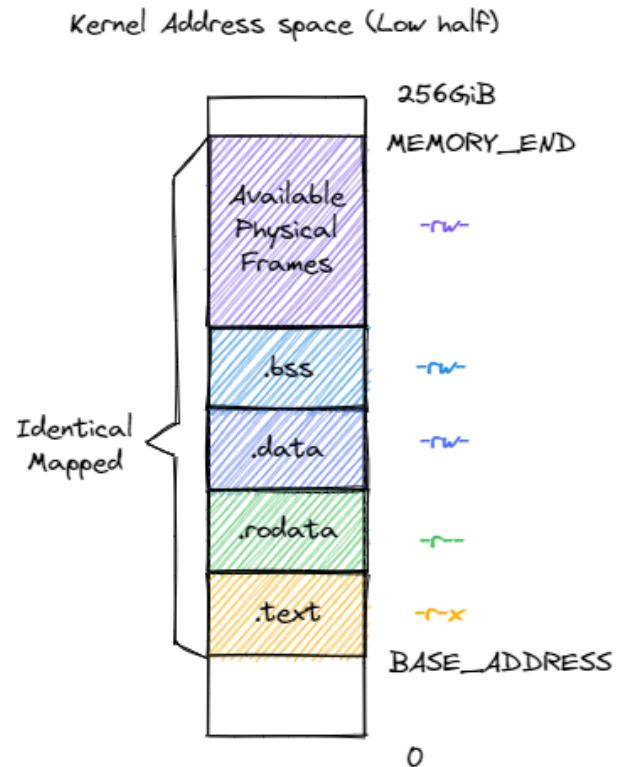
下图是软件看到的 64 位地址空间在 SV39 分页模式下实际可能通过 MMU 检查的最高 256GiB (之前在[这里](#)中解释过最高和最低 256GiB 的问题)：



可以看到，跳板放在最高的一个虚拟页面中。接下来则是从高到低放置每个应用的内核栈，内核栈的大小由 config 子模块的 KERNEL\_STACK\_SIZE 给出。它们的映射方式为 MapPermission 中的 rw 两个标志位，意味着这个逻辑段仅允许 CPU 处于内核态访问，且只能读或写。

注意相邻两个内核栈之间会预留一个 **保护页面 (Guard Page)**，它是内核地址空间中的空洞，多级页表中并不存在与它相关的映射。它的意义在于当内核栈空间不足（如调用层数过多或死递归）的时候，代码会尝试访问空洞区域内的虚拟地址，然而它无法在多级页表中找到映射，便会触发异常，此时控制权会交给内核 trap handler 函数进行异常处理。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，空洞区域被设置的越大，我们就能越早捕获到这一可能覆盖其他重要数据的错误异常。由于我们的内核非常简单且内核栈的大小设置比较宽裕，在当前的设计中我们仅将空洞区域的大小设置为单个页面。

下面则给出了内核地址空间的低 256GiB 的布局：



内核的四个逻辑段 `.text/.rodata/.data/.bss` 被恒等映射到物理内存，这使得我们在无需调整内核内存布局 `os/src/linker.ld` 的情况下就仍能象启用页表机制之前那样访问内核的各个段。注意我们借用页表机制对这些逻辑段的访问方式做出了限制，这都是为了在硬件的帮助下能够尽可能发现内核中的 bug，在这里：

- 四个逻辑段的 U 标志位均未被设置，使得 CPU 只能在处于 S 特权级（或以上）时访问它们；
- 代码段 `.text` 不允许被修改；
- 只读数据段 `.rodata` 不允许被修改，也不允许从它上面取指执行；
- `.data/.bss` 均允许被读写，但是不允许从它上面取指执行。

此外，之前提到过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。它们的标志位仅包含 `rw`，意味着该逻辑段只能在 S 特权级以上访问，并且只能读写。

下面我们给出创建内核地址空间的方法 `new_kernel`：

```

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

```

(下页继续)

(续上页)

```

14 }
15
16 impl MemorySet {
17 /// Without kernel stacks.
18 pub fn new_kernel() -> Self {
19 let mut memory_set = Self::new_bare();
20 // map trampoline
21 memory_set.map_trampoline();
22 // map kernel sections
23 println!(".text [{:#x}, {:#x})", stext as usize, etext as usize);
24 println!(".rodata [{:#x}, {:#x})", srodata as usize, erodata as usize);
25 println!(".data [{:#x}, {:#x})", sdata as usize, edata as usize);
26 println!(".bss [{:#x}, {:#x})", sbss_with_stack as usize, ebss as usize);
27 println!("mapping .text section");
28 memory_set.push(MapArea::new(
29 (stext as usize).into(),
30 (etext as usize).into(),
31 MapType::Identical,
32 MapPermission::R | MapPermission::X,
33), None);
34 println!("mapping .rodata section");
35 memory_set.push(MapArea::new(
36 (srodata as usize).into(),
37 (erodata as usize).into(),
38 MapType::Identical,
39 MapPermission::R,
40), None);
41 println!("mapping .data section");
42 memory_set.push(MapArea::new(
43 (sdata as usize).into(),
44 (edata as usize).into(),
45 MapType::Identical,
46 MapPermission::R | MapPermission::W,
47), None);
48 println!("mapping .bss section");
49 memory_set.push(MapArea::new(
50 (sbss_with_stack as usize).into(),
51 (ebss as usize).into(),
52 MapType::Identical,
53 MapPermission::R | MapPermission::W,
54), None);
55 println!("mapping physical memory");
56 memory_set.push(MapArea::new(
57 (ekernel as usize).into(),
58 MEMORY_END.into(),
59 MapType::Identical,
60 MapPermission::R | MapPermission::W,
61), None);
62 memory_set
63 }
64 }

```

new\_kernel 将映射跳板和地址空间中最低 256GiB 中的内核逻辑段。第 3 行开始，我们从 os/src/linker.ld 中引用了很多表示各个段位置的符号，而后在 new\_kernel 中，我们从低地址到高地址依次创建 5 个逻辑段并通过 push 方法将它们插入到内核地址空间中，上面我们已经详细介绍过这 5 个逻辑段。跳板是通过 map\_trampoline 方法来映射的，我们也将在这章最后一节进行讲解。

## 5.6.4 应用地址空间

现在我们来介绍如何创建应用的地址空间。在前面的章节中，我们直接将丢弃了所有符号信息的应用二进制镜像链接到内核，在初始化的时候内核仅需将他们加载到正确的初始物理地址就能使它们正确执行。但本章中，我们希望效仿内核地址空间的设计，同样借助页表机制使得应用地址空间的各个逻辑段也可以有不同的访问方式限制，这样可以提早检测出应用的错误并及时将其终止以最小化它对系统带来的恶劣影响。

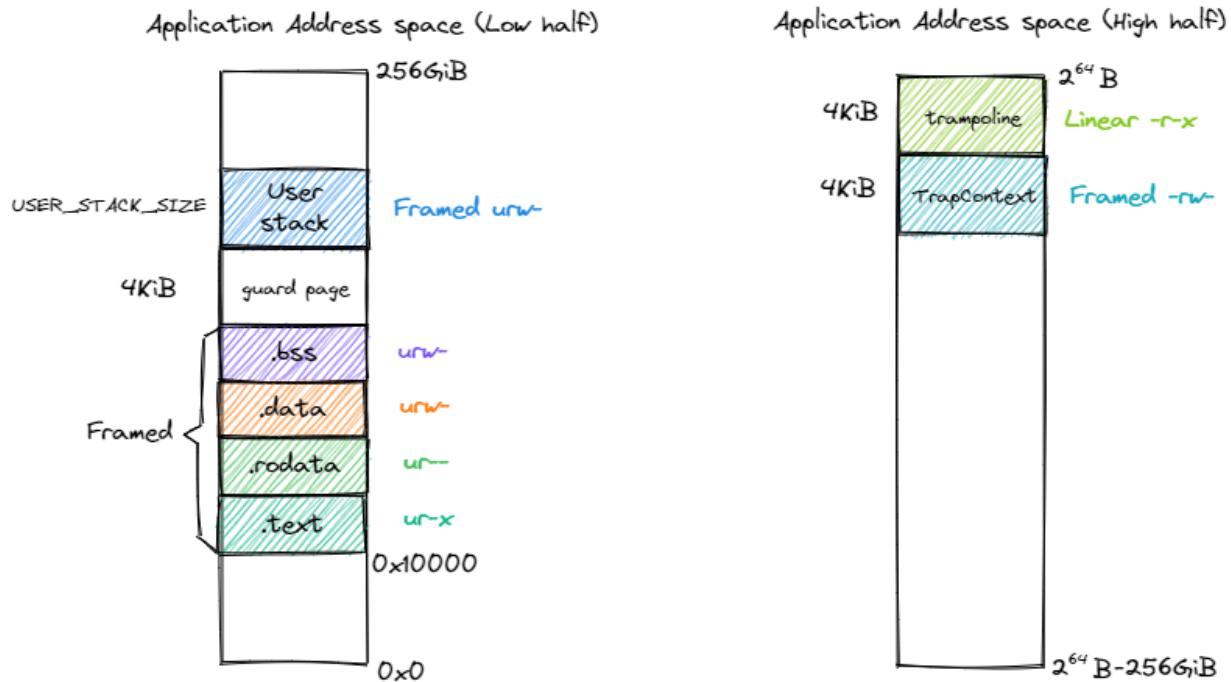
在第三章中，每个应用链接脚本中的起始地址被要求是不同的，这样它们的代码和数据存放的位置才不会产生冲突。但这是一种对于应用开发者很不方便的设计。现在，借助地址空间的抽象，我们终于可以让所有应用程序都使用同样的起始地址，这也意味着所有应用可以使用同一个链接脚本了：

```

1 /* user/src/linker.ld */
2
3 OUTPUT_ARCH(riscv)
4 ENTRY(_start)
5
6 BASE_ADDRESS = 0x10000;
7
8 SECTIONS
9 {
10 . = BASE_ADDRESS;
11 .text : {
12 *(.text.entry)
13 *(.text .text.*)
14 }
15 . = ALIGN(4K);
16 .rodata : {
17 *(.rodata .rodata.*)
18 }
19 . = ALIGN(4K);
20 .data : {
21 *(.data .data.*)
22 }
23 .bss : {
24 *(.bss .bss.*)
25 }
26 /DISCARD/ : {
27 *(.eh_frame)
28 *(.debug*)
29 }
30 }
```

我们将起始地址 `BASE_ADDRESS` 设置为 `0x10000`（我们这里并不设置为 `0x0`，因为它一般代表空指针），显然它只能是一个地址空间中的虚拟地址而非物理地址。事实上由于我们将入口汇编代码段放在最低的地方，这也是整个应用的入口点。我们只需清楚这一事实即可，而无需像之前一样将其硬编码到代码中。此外，在 `.text` 和 `.rodata` 中间以及 `.rodata` 和 `.data` 中间我们进行了页面对齐，因为前后两个逻辑段的访问方式限制是不同的，由于我们只能以页为单位对这个限制进行设置，因此就只能将下一个逻辑段对齐到下一个页面开始放置。而 `.data` 和 `.bss` 两个逻辑段由于访问限制相同（可读写），它们中间则无需进行页面对齐。

下图展示了应用地址空间的布局：



左侧给出了应用地址空间最低 256GiB 的布局：从 0x10000 开始向高地址放置应用内存布局中的各个逻辑段，最后放置带有一个保护页面的用户栈。这些逻辑段都是以 Framed 方式映射到物理内存的，从访问方式上来说都加上了 U 标志位代表 CPU 可以在 U 特权级也就是执行应用代码的时候访问它们。右侧则给出了最高的 256GiB，可以看出它只是和内核地址空间一样将跳板放置在最高页，还将 Trap 上下文放置在次高页中。这两个虚拟页面虽然位于应用地址空间，但是它们并不包含 U 标志位，事实上它们在地址空间切换的时候才会发挥作用，请同样参考本章的最后一节。

在 `os/src/build.rs` 中，我们不再将丢弃了所有符号的应用二进制镜像链接进内核，因为在应用二进制镜像中，内存布局中各个逻辑段的位置和访问限制等信息都被裁剪掉了。我们直接使用保存了逻辑段信息的 ELF 格式的应用可执行文件。这样 `loader` 子模块的设计实现也变得精简：

```
// os/src/loader.rs

pub fn get_num_app() -> usize {
 extern "C" { fn _num_app(); }
 unsafe { (_num_app as usize as *const usize).read_volatile() }
}

pub fn get_app_data(app_id: usize) -> &'static [u8] {
 extern "C" { fn _num_app(); }
 let num_app_ptr = _num_app as usize as *const usize;
 let num_app = get_num_app();
 let app_start = unsafe {
 core::slice::from_raw_parts(num_app_ptr.add(1), num_app + 1)
 };
 assert!(app_id < num_app);
 unsafe {
 core::slice::from_raw_parts(
 app_start[app_id] as *const u8,
 app_start[app_id + 1] - app_start[app_id]
)
 }
}
```

它仅需要提供两个函数: `get_num_app` 获取链接到内核内的应用的数目, 而 `get_app_data` 则根据传入的应用编号取出对应应用的 ELF 格式可执行文件数据。它们和之前一样仍是基于 `build.rs` 生成的 `link_app.S` 给出的符号来确定其位置, 并实际放在内核的数据段中。`loader` 模块中原有的内核和用户栈则分别作为逻辑段放在内核和用户地址空间中, 我们无需再去专门为它们定义一种类型。

在创建应用地址空间的时候, 我们需要对 `get_app_data` 得到的 ELF 格式数据进行解析, 找到各个逻辑段所在位置和访问限制并插入进来, 最终得到一个完整的应用地址空间:

```

1 // os/src/mm/memory_set.rs
2
3 impl MemorySet {
4 /// Include sections in elf and trampoline and TrapContext and user stack,
5 /// also returns user_sp and entry point.
6 pub fn from_elf(elf_data: &[u8]) -> (Self, usize, usize) {
7 let mut memory_set = Self::new_bare();
8 // map trampoline
9 memory_set.map_trampoline();
10 // map program headers of elf, with U flag
11 let elf = xmas_elf::ElfFile::new(elf_data).unwrap();
12 let elf_header = elf.header;
13 let magic = elf_header.pt1.magic;
14 assert_eq!(magic, [0x7f, 0x45, 0x4c, 0x46], "invalid elf!");
15 let ph_count = elf_header.pt2.ph_count();
16 let mut max_end_vpn = VirtPageNum(0);
17 for i in 0..ph_count {
18 let ph = elf.program_header(i).unwrap();
19 if ph.get_type().unwrap() == xmas_elf::program::Type::Load {
20 let start_va: VirtAddr = (ph.virtual_addr() as usize).into();
21 let end_va: VirtAddr = ((ph.virtual_addr() + ph.mem_size()) as usize).
22 into();
23 let mut map_perm = MapPermission::U;
24 let ph_flags = ph.flags();
25 if ph_flags.is_read() { map_perm |= MapPermission::R; }
26 if ph_flags.is_write() { map_perm |= MapPermission::W; }
27 if ph_flags.is_execute() { map_perm |= MapPermission::X; }
28 let map_area = MapArea::new(
29 start_va,
30 end_va,
31 MapType::Framed,
32 map_perm,
33);
34 max_end_vpn = map_area.vpn_range.get_end();
35 memory_set.push(
36 map_area,
37 Some(&elf.input[ph.offset() as usize..(ph.offset() + ph.file_
38 .size()) as usize])
39);
40 }
41 // map user stack with U flags
42 let max_end_va: VirtAddr = max_end_vpn.into();
43 let mut user_stack_bottom: usize = max_end_va.into();
44 // guard page
45 user_stack_bottom += PAGE_SIZE;
46 let user_stack_top = user_stack_bottom + USER_STACK_SIZE;
47 memory_set.push(MapArea::new(
48 user_stack_bottom.into(),
 user_stack_top.into(),
)
 }
 }
}

```

(下页继续)

(续上页)

```

49 MapType::Framed,
50 MapPermission::R | MapPermission::W | MapPermission::U,
51), None);
52 // map TrapContext
53 memory_set.push(MapArea::new(
54 TRAP_CONTEXT.into(),
55 TRAMPOLINE.into(),
56 MapType::Framed,
57 MapPermission::R | MapPermission::W,
58), None);
59 (memory_set, user_stack_top, elf.header.pt2.entry_point() as usize)
60 }
61 }
```

- 第 9 行，我们将跳板插入到应用地址空间；
- 第 11 行，我们使用外部 crate xmas\_elf 来解析传入的应用 ELF 数据并可以轻松取出各个部分。此前我们简要介绍过 ELF 格式的布局。第 14 行，我们取出 ELF 的魔数来判断它是不是一个合法的 ELF。

第 15 行，我们可以直接得到 program header 的数目，然后遍历所有的 program header 并将合适的区域加入到应用地址空间中。这一过程的主体在第 17~39 行之间。第 19 行我们确认 program header 的类型是 LOAD，这表明它有被内核加载的必要，此时不必理会其他类型的 program header。接着通过 ph.virtual\_addr() 和 ph.mem\_size() 来计算这一区域在应用地址空间中的位置，通过 ph.flags() 来确认这一区域访问方式的限制并将其转换为 MapPermission 类型（注意它默认包含 U 标志位）。最后我们在第 27 行创建逻辑段 map\_area 并在第 34 行 push 到应用地址空间。在 push 的时候我们需要完成数据拷贝，当前 program header 数据被存放的位置可以通过 ph.offset() 和 ph.file\_size() 来找到。注意当存在一部分零初始化的时候，ph.file\_size() 将会小于 ph.mem\_size()，因为这些零出于缩减可执行文件大小的原因不应该实际出现在 ELF 数据中。

- 我们从第 40 行开始处理用户栈。注意在前面加载各个 program header 的时候，我们就已经维护了 max\_end\_vpn 记录目前涉及到的最大的虚拟页号，只需紧接着在它上面再放置一个保护页面和用户栈即可。
- 第 53 行则在应用地址空间中映射次高页面来存放 Trap 上下文。
- 第 59 行返回的时候，我们不仅返回应用地址空间 memory\_set，也同时返回用户栈虚拟地址 user\_stack\_top 以及从解析 ELF 得到的该应用入口点地址，它们将被我们用来创建应用的任务控制块。

小结一下，本节讲解了 地址空间 这一抽象概念的含义与对应的具体数据结构设计与实现，并进一步介绍了在分页机制的帮助下，内核和应用各自的地址空间的基本组成和创建这两种地址空间的基本方法。接下来，需要考虑如何把地址空间与之前的分时多任务结合起来，实现一个更加安全和强大的内核，这还需要进一步拓展内核功能—建立具体的内核虚拟地址空间和应用虚拟地址空间、实现不同地址空间的切换，即能切换不同应用之间的地址空间，以及应用与内核之间的地址空间。在下一节，我们将讲解如何构建基于地址空间的分时多任务操作系统—“头甲龙”。

#### 提示：内核如何访问应用的数据？

应用应该不能直接访问内核的数据，但内核可以访问应用的数据，这是如何做的？由于内核要管理应用，所以它负责构建自身和其他应用的多级页表。如果内核获得了一个应用数据的虚地址，内核就可以通过查询应用的页表来把应用的虚地址转换为物理地址，内核直接访问这个地址（注：内核自身的虚实映射是恒等映射），就可以获得应用数据的内容了。

## 5.7 基于地址空间的分时多任务

### 5.7.1 本节导读

本节我们介绍如何基于地址空间抽象而不是对于物理内存的直接访问来实现支持地址空间隔离的分时多任务系统—“头甲龙”<sup>1</sup> 操作系统。这样，我们的应用编写会更加方便，应用与操作系统内核的空间隔离性增强了，应用程序和操作系统自身的安全性也得到了加强。为此，需要对现有的操作系统进行如下的功能扩展：

- 创建内核页表，使能分页机制，建立内核的虚拟地址空间；
- 扩展 Trap 上下文，在保存与恢复 Trap 上下文的过程中切换页表（即切换虚拟地址空间）；
- 建立用于内核地址空间与应用地址空间相互切换所需的跳板空间；
- 扩展任务控制块包括虚拟内存相关信息，并在加载执行创建基于某应用的任务时，建立应用的虚拟地址空间；
- 改进 Trap 处理过程和 sys\_write 等系统调用的实现以支持分离的应用地址空间和内核地址空间。

在扩展了上述功能后，应用与应用之间，应用与操作系统内核之间通过硬件分页机制实现了内存空间隔离，且应用和内核之间还是能有效地进行相互访问，而且应用程序的编写也会更加简单通用。

### 5.7.2 建立并开启基于分页模式的虚拟地址空间

当 SBI 实现（本项目中基于 RustSBI）初始化完成后，CPU 将跳转到内核入口点并在 S 特权级上执行，此时还并没有开启分页模式，内核的每次访存是直接的物理内存访问。而在开启分页模式之后，内核代码在访存时只能看到内核地址空间，此时每次访存需要通过 MMU 的地址转换。这两种模式之间的过渡在内核初始化期间完成。

#### 创建内核地址空间

我们创建内核地址空间的全局实例：

```
// os/src/mm/memory_set.rs

lazy_static! {
 pub static ref KERNEL_SPACE: Arc<UPSafeCell<MemorySet>> = Arc::new(unsafe {
 UPSafeCell::new(MemorySet::new_kernel())
 });
}
```

从之前对于 `lazy_static!` 宏的介绍可知，`KERNEL_SPACE` 在运行期间它第一次被用到时才会实际进行初始化，而它所占据的空间则是编译期被放在全局数据段中。这里使用 `Arc<UPSafeCell<T>>` 组合是因为我们既需要 `Arc<T>` 提供的共享引用，也需要 `UPSafeCell<T>` 提供的内部可变引用访问。

在 `rust_main` 函数中，我们首先调用 `mm::init` 进行内存管理子系统的初始化：

```
// os/src/mm/mod.rs

pub use memory_set::KERNEL_SPACE;

pub fn init() {
 heap_allocator::init_heap();
 frame_allocator::init_frame_allocator();
```

(下页继续)

<sup>1</sup> 头甲龙最早出现在 1.8 亿年以前的侏罗纪中期，是身披重甲的食素恐龙，尾巴末端的尾锤，是防身武器。

(续上页)

```
 KERNEL_SPACE.exclusive_access().activate();
}
```

可以看到，我们最先进行了全局动态内存分配器的初始化，因为接下来马上就要用到 Rust 的堆数据结构。接下来我们初始化物理页帧管理器（内含堆数据结构 `Vec<T>`）使能可用物理页帧的分配和回收能力。最后我们创建内核地址空间并让 CPU 开启分页模式，MMU 在地址转换的时候使用内核的多级页表，这一切均在一行之内做到：

- 首先，我们引用 `KERNEL_SPACE`，这是它第一次被使用，就在此时它会被初始化，调用 `MemorySet::new_kernel` 创建一个内核地址空间并使用 `Arc<UPSafeCell<T>>` 包裹起来；
- 接着使用 `.exclusive_access()` 获取一个可变引用 `&mut MemorySet`。需要注意的是这里发生了两次隐式类型转换：
  - 我们知道 `exclusive_access` 是 `UPSafeCell<T>` 的方法而不是 `Arc<T>` 的方法，由于 `Arc<T>` 实现了 `Deref Trait`，当 `exclusive_access` 需要一个 `&UPSafeCell<T>` 类型的参数的时候，编译器会自动将传入的 `Arc<UPSafeCell<T>>` 转换为 `&UPSafeCell<T>` 这样就实现了类型匹配；
  - 事实上 `UPSafeCell<T>::exclusive_access` 返回的是一个 `RefMut<'_, T>`，这同样是 `RAII` 的思想，当这个类型生命周期结束后互斥锁就会被释放。而该类型实现了 `DerefMut Trait`，因此当一个函数接受类型为 `&mut T` 的参数却被传入一个类型为 `&mut RefMut<'_, T>` 的参数的时候，编译器会自动进行类型转换使参数匹配。
- 最后，我们调用 `MemorySet::activate`：

```
1 // os/src/mm/page_table.rs
2
3 pub fn token(&self) -> usize {
4 8usize << 60 | self.root_ppn.0
5 }
6
7 // os/src/mm/memory_set.rs
8
9 impl MemorySet {
10 pub fn activate(&self) {
11 let satp = self.page_table.token();
12 unsafe {
13 satp:::write(satp);
14 asm!("sfence.vma");
15 }
16 }
17 }
```

`PageTable:::token` 会按照 `satp CSR` 格式要求 构造一个无符号 64 位无符号整数，使得其分页模式为 `SV39`，且将当前多级页表的根节点所在的物理页号填充进去。在 `activate` 中，我们将这个值写入当前 CPU 的 `satp CSR`，从这一刻开始 `SV39` 分页模式就被启用了，而且 MMU 会使用内核地址空间的多级页表进行地址转换。

我们必须注意切换 `satp CSR` 是否是一个平滑的过渡：其含义是指，切换 `satp` 的指令及其下一条指令这两条相邻的指令的虚拟地址是相邻的（由于切换 `satp` 的指令并不是一条跳转指令，`pc` 只是简单的自增当前指令的字长），而它们所在的物理地址一般情况下也是相邻的，但是它们所经过的地址转换流程却是不同的——切换 `satp` 导致 MMU 查的多级页表是不同的。这就要求前后两个地址空间在切换 `satp` 的指令附近的映射满足某种意义上的连续性。

幸运的是，我们做到了这一点。这条写入 `satp` 的指令及其下一条指令都在内核内存布局的代码段中，在切换之后是一个恒等映射，而在切换之前是视为物理地址直接取指，也可以将其看成一个恒等映射。这完全符合我们的期待：即使切换了地址空间，指令仍应该能够被连续的执行。

注意到在 `activate` 的最后，我们插入了一条汇编指令 `sfence.vma`，它又起到什么作用呢？

让我们再来看看多级页表：它相比线性表虽然大量节约了内存占用，但是却需要 MMU 进行更多的隐式访存。如果是一个线性表，MMU 仅需单次访存就能找到页表项并完成地址转换，而多级页表（以 SV39 为例，不考虑大页）最顺利的情况下也需要三次访存。这些额外的访存和真正访问数据的那些访存在空间上并不相邻，加大了多级缓存的压力，一旦缓存缺失将带来巨大的性能惩罚。如果采用多级页表实现，这个问题会变得更为严重，使得地址空间抽象的性能开销过大。

为了解决性能问题，一种常见的做法是在 CPU 中利用部分硬件资源额外加入一个 **快表** (TLB, Translation Lookaside Buffer)，它维护了部分虚拟页号到页表项的键值对。当 MMU 进行地址转换的时候，首先会到快表中看看是否匹配，如果匹配的话直接取出页表项完成地址转换而无需访存；否则再去查页表并将键值对保存在快表中。一旦我们修改 `satp` 就会切换地址空间，快表中的键值对就会失效（因为快表保存着老地址空间的映射关系，切换到新地址空间后，老的映射关系就沒用了）。为了确保 MMU 的地址转换能够及时与 `satp` 的修改同步，我们需要立即使用 `sfence.vma` 指令将快表清空，这样 MMU 就不会看到快表中已经过期的键值对了。

---

#### 注解：`sfence.vma` 是一个屏障 (Barrier)

对于一种含有快表的 RISC-V CPU 实现来说，我们可以认为 `sfence.vma` 的作用就是清空快表。事实上它在特权级规范中被定义为一种含义更加丰富的内存屏障，具体来说：`sfence.vma` 可以使得所有发生在它后面的地址转换都能够看到所有排在它前面的写入操作。在不同的硬件配置上这条指令要做的具体事务是有差异的。这条指令还可以被精细配置来减少同步开销，详情请参考 RISC-V 特权级规范。

---

### 检查内核地址空间的多级页表设置

调用 `mm::init` 之后我们就使能了内核动态内存分配、物理页帧管理，还启用了分页模式进入了内核地址空间。之后我们可以通过 `mm::remap_test` 来检查内核地址空间的多级页表是否被正确设置：

```
// os/src/mm/memory_set.rs

pub fn remap_test() {
 let mut kernel_space = KERNEL_SPACE.exclusive_access();
 let mid_text: VirtAddr = ((stext as usize + etext as usize) / 2).into();
 let mid_rodata: VirtAddr = ((srodata as usize + erodata as usize) / 2).into();
 let mid_data: VirtAddr = ((sdata as usize + edata as usize) / 2).into();
 assert_eq!(
 kernel_space.page_table.translate(mid_text.floor()).unwrap().writable(),
 false
);
 assert_eq!(
 kernel_space.page_table.translate(mid_rodata.floor()).unwrap().writable(),
 false,
);
 assert_eq!(
 kernel_space.page_table.translate(mid_data.floor()).unwrap().executable(),
 false,
);
 println!("remap_test passed!");
}
```

在上述函数的实现中，分别通过手动查内核多级页表的方式验证代码段和只读数据段不允许被写入，同时不允许从数据段上取指执行。

### 5.7.3 跳板机制的实现

上一小节我们看到无论是内核还是应用的地址空间，最高的虚拟页面都是一个跳板。同时应用地址空间的次高虚拟页面还被设置为用来存放应用的 Trap 上下文。那么跳板究竟起什么作用呢？为何不直接把 Trap 上下文仍放到应用的内核栈中呢？

回忆曾在第二章介绍过的[Trap 上下文保存与恢复](#)。当一个应用 Trap 到内核时，`sscratch` 已指向该应用的内核栈栈顶，我们用一条指令即可从用户栈切换到内核栈，然后直接将 Trap 上下文压入内核栈栈顶。当 Trap 处理完毕返回用户态的时候，将 Trap 上下文中的内容恢复到寄存器上，最后将保存着应用用户栈顶的 `sscratch` 与 `sp` 进行交换，也就从内核栈切换回了用户栈。在这个过程中，`sscratch` 起到了非常关键的作用，它使得我们可以在不破坏任何通用寄存器的情况下，完成用户栈与内核栈的切换，以及位于内核栈顶的 Trap 上下文的保存与恢复。

然而，一旦使能了分页机制，一切就并没有这么简单了，我们必须在这个过程中同时完成地址空间的切换。具体来说，当 `__alltraps` 保存 Trap 上下文的时候，我们必须通过修改 `satp` 从应用地址空间切换到内核地址空间，因为 `trap handler` 只有在内核地址空间中才能访问；同理，在 `__restore` 恢复 Trap 上下文的时候，我们也必须从内核地址空间切换回应用地址空间，因为应用的代码和数据只能在它自己的地址空间中才能访问，应用是看不到内核地址空间的。这样就要求地址空间的切换不能影响指令的连续执行，即要求应用和内核地址空间在切换地址空间指令附近是平滑的。

---

#### 注解：内核与应用地址空间的隔离

目前我们的设计思路 A 是：对内核建立唯一的内核地址空间存放内核的代码、数据，同时对于每个应用维护一个它们自己的用户地址空间，因此在 Trap 的时候就需要进行地址空间切换，而在任务切换的时候无需进行（因为这个过程全程在内核内完成）。

另外的一种设计思路 B 是：让每个应用都有一个包含应用和内核的地址空间，并将其中的逻辑段分为内核和用户两部分，分别映射到内核/用户的代码和数据，且分别在 CPU 处于 S/U 特权级时访问。此设计中并不存在一个单独的内核地址空间。

设计方式 B 的优点在于：Trap 的时候无需切换地址空间，而在任务切换的时候才需要切换地址空间。相对而言，设计方式 B 比设计方式 A 更容易实现，在应用高频进行系统调用的时候，采用设计方式 B 能够避免频繁地址空间切换的开销，这通常源于快表或 cache 的失效问题。但是设计方式 B 也有缺点：即内核的逻辑段需要在每个应用的地址空间内都映射一次，这会带来一些无法忽略的内存占用开销，并显著限制了嵌入式平台（如我们所采用的 K210）的任务并发数。此外，设计方式 B 无法防御针对处理器电路设计缺陷的侧信道攻击（如[熔断 \(Meltdown\) 漏洞](#)），使得恶意应用能够以某种方式间接“看到”内核地址空间中的数据，使得用户隐私数据有可能被泄露。将内核与地址空间隔离便是修复此漏洞的一种方法。

经过权衡，在本教程中我们参考 MIT 的教学 OS [xv6](#)，采用内核和应用地址空间隔离的设计。

---

我们为何将应用的 Trap 上下文放到应用地址空间的次高页面而不是内核地址空间中的内核栈中呢？原因在于，在保存 Trap 上下文到内核栈中之前，我们必须完成两项工作：1) 必须先切换到内核地址空间，这就需要将内核地址空间的 `token` 写入 `satp` 寄存器；2) 之后还需要保存应用的内核栈栈顶的位置，这样才能以它为基址保存 Trap 上下文。这两步需要用寄存器作为临时周转，然而我们无法在不破坏任何一个通用寄存器的情况下做到这一点。因为事实上我们需要用到内核的两条信息：内核地址空间的 `token`，以及应用的内核栈栈顶的位置，RISC-V 却只提供一个 `sscratch` 寄存器可用来进行周转。所以，我们不得不将 Trap 上下文保存在应用地址空间的一个虚拟页面中，而不是切换到内核地址空间去保存。

## 扩展 Trap 上下文

为了方便实现，我们在 Trap 上下文中包含更多内容（和我们关于上下文的定义有些不同，它们在初始化之后便只会被读取而不会被写入，并不是每次都需要保存/恢复）：

```

1 // os/src/trap/context.rs
2
3 #[repr(C)]
4 pub struct TrapContext {
5 pub x: [usize; 32],
6 pub sstatus: Sstatus,
7 pub sepc: usize,
8 pub kernel_satp: usize,
9 pub kernel_sp: usize,
10 pub trap_handler: usize,
11 }

```

在多出的三个字段中：

- kernel\_satp 表示内核地址空间的 token，即内核页表的起始物理地址；
- kernel\_sp 表示当前应用在内核地址空间中的内核栈栈顶的虚拟地址；
- trap\_handler 表示内核中 trap handler 入口点的虚拟地址。

它们在应用初始化的时候由内核写入应用地址空间中的 TrapContext 的相应位置，此后就不再被修改。

## 切换地址空间

让我们来看一下现在的 `__alltraps` 和 `__restore` 各是如何在保存和恢复 Trap 上下文的同时也切换地址空间的：

```

1 # os/src/trap/trap.S
2
3 .section .text.trampoline
4 .globl __alltraps
5 .globl __restore
6 .align 2
7
8 __alltraps:
9 csrrw sp, sscratch, sp
10 # now sp->*TrapContext in user space, sscratch->user stack
11 # save other general purpose registers
12 sd x1, 1*8(sp)
13 # skip sp(x2), we will save it later
14 sd x3, 3*8(sp)
15 # skip tp(x4), application does not use it
16 # save x5~x31
17 .set n, 5
18 .rept 27
19 SAVE_GP %n
20 .set n, n+1
21 .endr
22 # we can use t0/t1/t2 freely, because they have been saved in TrapContext
23 csrr t0, sstatus
24 csrr t1, sepc
25 sd t0, 32*8(sp)
26 sd t1, 33*8(sp)
27 # read user stack from sscratch and save it in TrapContext

```

(下页继续)

(续上页)

```

27 csrr t2, sscratch
28 sd t2, 2*8(sp)
29 # load kernel_satp into t0
30 ld t0, 34*8(sp)
31 # load trap_handler into t1
32 ld t1, 36*8(sp)
33 # move to kernel_sp
34 ld sp, 35*8(sp)
35 # switch to kernel space
36 csrw satp, t0
37 sfence.vma
38 # jump to trap_handler
39 jr t1

40
41 __restore:
42 # a0: *TrapContext in user space(Constant); a1: user space token
43 # switch to user space
44 csrw satp, a1
45 sfence.vma
46 csrw sscratch, a0
47 mv sp, a0
48 # now sp points to TrapContext in user space, start restoring based on it
49 # restore sstatus/sepc
50 ld t0, 32*8(sp)
51 ld t1, 33*8(sp)
52 csrw sstatus, t0
53 csrw sepc, t1
54 # restore general purpose registers except x0/sp/tp
55 ld x1, 1*8(sp)
56 ld x3, 3*8(sp)
57 .set n, 5
58 .rept 27
59 LOAD_GP %n
60 .set n, n+1
61 .endr
62 # back to user stack
63 ld sp, 2*8(sp)
64 sret

```

- 当应用 Trap 进入内核的时候，硬件会设置一些 CSR 并在 S 特权级下跳转到 \_\_alltraps 保存 Trap 上下文。此时 sp 寄存器仍指向用户栈，但 sscratch 则被设置为指向应用地址空间中存放 Trap 上下文的位置（实际在次高页面）。随后，就像之前一样，我们 csrrw 交换 sp 和 sscratch，并基于指向 Trap 上下文位置的 sp 开始保存通用寄存器和一些 CSR，这个过程在第 28 行结束。到这里，我们就全程在应用地址空间中完成了保存 Trap 上下文的工作。

- 接下来该考虑切换到内核地址空间并跳转到 trap handler 了。

- 第 30 行将内核地址空间的 token 载入到 t0 寄存器中；
- 第 32 行将 trap handler 入口点的虚拟地址载入到 t1 寄存器中；
- 第 34 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 36~37 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 39 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。

注：这里我们不能像之前的章节那样直接 `call trap_handler`，原因稍后解释。

- 当内核将 Trap 处理完毕准备返回用户态的时候会调用 `__restore`（符合 RISC-V 函数调用规范），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 44~45 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 46 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 47 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 64 行最后通过 `sret` 指令返回用户态。

## 建立跳板页面

接下来还需要考虑切换地址空间前后指令能否仍能连续执行。可以看到我们将 `trap.S` 中的整段汇编代码放置在 `.text.trampoline` 段，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：

```

1 # os/src/linker.ld
2
3 stext = .;
4 .text : {
5 *(.text.entry)
6 . = ALIGN(4K);
7 strampoline = .;
8 *(.text_trampoline);
9 . = ALIGN(4K);
10 *(.text .text.*)
11 }

```

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。

在产生 trap 前后的一小段时间内会有一个比较 **极端** 的情况，即刚产生 trap 时，CPU 已经进入了内核态（即 Supervisor Mode），但此时执行代码和访问数据还是在应用程序所处的用户态虚拟地址空间中，而不是我们通常理解的内核虚拟地址空间。在这段特殊的时间内，CPU 指令为什么能够被连续执行呢？这里需要注意：无论是内核还是应用的地址空间，跳板的虚拟页均位于同样位置，且它们也将会映射到同一个实际存放这段汇编代码的物理页帧。也就是说，在执行 `__alltraps` 或 `__restore` 函数进行地址空间切换的时候，应用的用户态虚拟地址空间和操作系统内核的内核态虚拟地址空间对切换地址空间的指令所在页的映射方式均是相同的，这就说明了这段切换地址空间的指令控制流仍是可以连续执行的。

现在可以说明我们在创建用户/内核地址空间中用到的 `map_trampoline` 是如何实现的了：

```

1 // os/src/config.rs
2
3 pub const TRAMPOLINE: usize = usize::MAX - PAGE_SIZE + 1;
4
5 // os/src/mm/memory_set.rs
6
7 impl MemorySet {
8 /// Mention that trampoline is not collected by areas.
9 fn map_trampoline(&mut self) {
10 self.page_table.map(
11 VirtAddr::from(TRAMPOLINE).into(),

```

(下页继续)

(续上页)

```

12 PhysAddr::from(strampoline as usize).into(),
13 PTEFlags::R | PTEFlags::X,
14);
15 }
16 }
```

这里我们为了实现方便并没有新增逻辑段 MemoryArea 而是直接在多级页表中插入一个从地址空间的最高虚拟页面映射到跳板汇编代码所在的物理页帧的键值对，访问权限与代码段相同，即 RX（可读可执行）。

最后可以解释为何我们在 \_\_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/k210.1d 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。

问题的本质可以概括为：跳转指令实际被执行时的虚拟地址和在编译器/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此指令的地址是不同的。

## 5.7.4 加载和执行应用程序

### 扩展任务控制块

为了让应用在运行时有一个安全隔离且符合编译器给应用设定的地址空间布局的虚拟地址空间，操作系统需要对任务进行更多的管理，所以任务控制块相比第三章也包含了更多内容：

```

1 // os/src/task/task.rs
2
3 pub struct TaskControlBlock {
4 pub task_cx: TaskContext,
5 pub task_status: TaskStatus,
6 pub memory_set: MemorySet,
7 pub trap_cx_ppn: PhysPageNum,
8 pub base_size: usize,
9 }
```

除了应用的地址空间 memory\_set 之外，还有位于应用地址空间次高页的 Trap 上下文被实际存放在物理页帧的物理页号 trap\_cx\_ppn，它能够方便我们对于 Trap 上下文进行访问。此外，base\_size 统计了应用数据的大小，也就是在应用地址空间中从 0x0 开始到用户栈结束一共包含多少字节。它后续还应该包含用于应用动态内存分配的堆空间的大小，但目前暂不支持。

### 更新对任务控制块的管理

下面是任务控制块的创建：

```

1 // os/src/config.rs
2
3 /// Return (bottom, top) of a kernel stack in kernel space.
4 pub fn kernel_stack_position(app_id: usize) -> (usize, usize) {
5 let top = TRAMPOLINE - app_id * (KERNEL_STACK_SIZE + PAGE_SIZE);
6 let bottom = top - KERNEL_STACK_SIZE;
7 (bottom, top)
8 }
```

(下页继续)

(续上页)

```

9
10 // os/src/task/task.rs
11
12 impl TaskControlBlock {
13 pub fn new(elf_data: &[u8], app_id: usize) -> Self {
14 // memory_set with elf program headers/trampoline/trap context/user stack
15 let (memory_set, user_sp, entry_point) = MemorySet::from_elf(elf_data);
16 let trap_cx_ppn = memory_set
17 .translate(VirtAddr::from(TRAP_CONTEXT).into())
18 .unwrap()
19 .ppn();
20 let task_status = TaskStatus::Ready;
21 // map a kernel-stack in kernel space
22 let (kernel_stack_bottom, kernel_stack_top) = kernel_stack_position(app_id);
23 KERNEL_SPACE
24 .exclusive_access()
25 .insert_framed_area(
26 kernel_stack_bottom.into(),
27 kernel_stack_top.into(),
28 MapPermission::R | MapPermission::W,
29);
30 let task_control_block = Self {
31 task_status,
32 task_cx: TaskContext::goto_trap_return(kernel_stack_top),
33 memory_set,
34 trap_cx_ppn,
35 base_size: user_sp,
36 };
37 // prepare TrapContext in user space
38 let trap_cx = task_control_block.get_trap_cx();
39 *trap_cx = TrapContext::app_init_context(
40 entry_point,
41 user_sp,
42 KERNEL_SPACE.exclusive_access().token(),
43 kernel_stack_top,
44 trap_handler as usize,
45);
46 task_control_block
47 }
48 }

```

- 第 15 行, 解析传入的 ELF 格式数据构造应用的地址空间 memory\_set 并获得其他信息;
- 第 16 行, 从地址空间 memory\_set 中查多级页表找到应用地址空间中的 Trap 上下文实际被放在哪个物理页帧;
- 第 22 行, 根据传入的应用 ID app\_id 调用在 config 子模块中定义的 kernel\_stack\_position 找到应用的内核栈预计放在内核地址空间 KERNEL\_SPACE 中的哪个位置, 并通过 insert\_framed\_area 实际将这个逻辑段加入到内核地址空间中;
- 第 30-32 行, 在应用的内核栈顶压入一个跳转到 trap\_return 而不是 \_\_restore 的任务上下文, 这主要是为了能够支持对该应用的启动并顺利切换到用户地址空间执行。在构造方式上, 只是将 ra 寄存器的值设置为 trap\_return 的地址。trap\_return 是后面要介绍的新版的 Trap 处理的一部分。
- 这里对裸指针解引用成立的原因在于: 当前已经进入了内核地址空间, 而要操作的内核栈也是在内核地址空间中的;
- 第 33~36 行, 用上面的信息来创建并返回任务控制块实例 task\_control\_block;

- 第 38 行, 查找该应用的 Trap 上下文的内核虚地址。由于应用的 Trap 上下文是在应用地址空间而不是在内核地址空间中, 我们只能手动查页表找到 Trap 上下文实际被放在的物理页帧, 然后通过之前介绍的在内核地址空间读写特定物理页帧的能力 获得在用户空间的 Trap 上下文的可变引用用于初始化:

```
// os/src/task/task.rs

impl TaskControlBlock {
 pub fn get_trap_cx(&self) -> &'static mut TrapContext {
 self.trap_cx_ppn.get_mut()
 }
}
```

此处需要说明的是, 返回 'static 的可变引用和之前一样可以看成一个绕过 unsafe 的裸指针; 而 PhysPageNum::get\_mut 是一个泛型函数, 由于我们已经声明了总体返回 TrapContext 的可变引用, 则 Rust 编译器会给 get\_mut 泛型函数针对具体类型 TrapContext 的情况生成一个特定版本的 get\_mut 函数实现。在 get\_trap\_cx 函数中则会静态调用 get\_mut 泛型函数的特定版本实现。

- 第 39~45 行, 调用 TrapContext::app\_init\_context 函数, 通过应用的 Trap 上下文的可变引用对其进行初始化。具体初始化过程如下所示:

```
// os/src/trap/context.rs

impl TrapContext {
 pub fn set_sp(&mut self, sp: usize) { self.x[2] = sp; }
 pub fn app_init_context(
 entry: usize,
 sp: usize,
 kernel_satp: usize,
 kernel_sp: usize,
 trap_handler: usize,
) -> Self {
 let mut sstatus = sstatus::read();
 sstatus.set_spp(SPP::User);
 let mut cx = Self {
 x: [0; 32],
 sstatus,
 sepc: entry,
 kernel_satp,
 kernel_sp,
 trap_handler,
 };
 cx.set_sp(sp);
 cx
 }
}
```

和之前实现相比, TrapContext::app\_init\_context 需要补充上让应用在 \_\_alltraps 能够顺利进入到内核地址空间并跳转到 trap handler 入口点的相关信息。

在内核初始化的时候, 需要将所有的应用加载到全局应用管理器中:

```
// os/src/task/mod.rs

struct TaskManagerInner {
 tasks: Vec<TaskControlBlock>,
 current_task: usize,
}
```

(下页继续)

(续上页)

```

8 lazy_static! {
9 pub static ref TASK_MANAGER: TaskManager = {
10 println!("init TASK_MANAGER");
11 let num_app = get_num_app();
12 println!("num_app = {}", num_app);
13 let mut tasks: Vec<TaskControlBlock> = Vec::new();
14 for i in 0..num_app {
15 tasks.push(TaskControlBlock::new(
16 get_app_data(i),
17 i,
18));
19 }
20 TaskManager {
21 num_app,
22 inner: RefCell::new(TaskManagerInner {
23 tasks,
24 current_task: 0,
25 }),
26 }
27 };
28 }

```

可以看到，在 TaskManagerInner 中我们使用向量 Vec 来保存任务控制块。在全局任务管理器 TASK\_MANAGER 初始化的时候，只需使用 loader 子模块提供的 get\_num\_app 和 get\_app\_data 分别获取链接到内核的应用数量和每个应用的 ELF 文件格式的数据，然后依次给每个应用创建任务控制块并加入到向量中即可。将 current\_task 设置为 0，表示内核将从第 0 个应用开始执行。

回过头来介绍一下应用构建器 os/build.rs 的改动：

- 首先，我们在 .incbin 中不再插入清除全部符号的应用二进制镜像 \*.bin，而是将应用的 ELF 执行文件直接链接进来；
- 其次，在链接每个 ELF 执行文件之前我们都加入一行 .align 3 来确保它们对齐到 8 字节，这是由于如果不这样做，xmas-elf crate 可能在解析 ELF 的时候进行不对齐的内存读写，例如使用 ld 指令从内存的一个没有对齐到 8 字节的地址加载一个 64 位的值到一个通用寄存器。而在 k210 平台上，由于其硬件限制，这种情况会触发一个内存读写不对齐的异常，导致解析无法正常完成。

为了方便后续的实现，全局任务管理器还需要提供关于当前应用与地址空间有关的一些信息：

```

1 // os/src/task/mod.rs
2
3 impl TaskManager {
4 fn get_current_token(&self) -> usize {
5 let inner = self.inner.borrow();
6 let current = inner.current_task;
7 inner.tasks[current].get_user_token()
8 }
9
10 fn get_current_trap_cx(&self) -> &mut TrapContext {
11 let inner = self.inner.borrow();
12 let current = inner.current_task;
13 inner.tasks[current].get_trap_cx()
14 }
15 }
16
17 pub fn current_user_token() -> usize {
18 TASK_MANAGER.get_current_token()
19 }

```

(下页继续)

(续上页)

```

19 }
20
21 pub fn current_trap_cx() -> &'static mut TrapContext {
22 TASK_MANAGER.get_current_trap_cx()
23 }
```

通过 `current_user_token` 可以获得当前正在执行的应用的地址空间的 `token`。同时，该应用地址空间中的 Trap 上下文很关键，内核需要访问它来拿到应用进行系统调用的参数并将系统调用返回值写回，通过 `current_trap_cx` 内核可以拿到它访问这个 Trap 上下文的可变引用并进行读写。

## 5.7.5 改进 Trap 处理的实现

让我们来看现在 `trap_handler` 的改进实现：

```

1 // os/src/trap/mod.rs
2
3 fn set_kernel_trap_entry() {
4 unsafe {
5 stvec::write(trap_from_kernel as usize, TrapMode::Direct);
6 }
7 }
8
9 #[no_mangle]
10 pub fn trap_from_kernel() -> ! {
11 panic!("a trap from kernel!");
12 }
13
14 #[no_mangle]
15 pub fn trap_handler() -> ! {
16 set_kernel_trap_entry();
17 let cx = current_trap_cx();
18 let scause = scause::read();
19 let stval = stval::read();
20 match scause.cause() {
21 ...
22 }
23 trap_return();
24 }
```

由于应用的 Trap 上下文不在内核地址空间，因此我们调用 `current_trap_cx` 来获取当前应用的 Trap 上下文的可变引用而不是像之前那样作为参数传入 `trap_handler`。至于 Trap 处理的过程则没有发生什么变化。

注意到，在 `trap_handler` 的开头还调用 `set_kernel_trap_entry` 将 `stvec` 修改为同模块下另一个函数 `trap_from_kernel` 的地址。这就是说，一旦进入内核后再次触发到 S 状态 Trap，则硬件在设置一些 CSR 寄存器之后，会跳过对通用寄存器的保存过程，直接跳转到 `trap_from_kernel` 函数，在这里直接 `panic` 退出。这是因为内核和应用的地址空间分离之后，U 状态 -> S 状态与 S 状态 -> S 状态的 Trap 上下文保存与恢复实现方式/Trap 处理逻辑有很大差别。这里为了简单起见，弱化了 S 状态 -> S 状态的 Trap 处理过程：直接 `panic`。

在 `trap_handler` 完成 Trap 处理之后，我们需要调用 `trap_return` 返回用户态：

```

1 // os/src/trap/mod.rs
2
3 fn set_user_trap_entry() {
4 unsafe {
```

(下页继续)

(续上页)

```

5 stvec::write(TRAMPOLINE as usize, TrapMode::Direct);
6 }
7 }
8
9 #[no_mangle]
10 pub fn trap_return() -> ! {
11 set_user_trap_entry();
12 let trap_cx_ptr = TRAP_CONTEXT;
13 let user_satp = current_user_token();
14 extern "C" {
15 fn __alltraps();
16 fn __restore();
17 }
18 let restore_va = __restore as usize - __alltraps as usize + TRAMPOLINE;
19 unsafe {
20 asm! (
21 "fence.i",
22 "jr {restore_va}",
23 restore_va = in(reg) restore_va,
24 in("a0") trap_cx_ptr,
25 in("a1") user_satp,
26 options(noreturn)
27);
28 }
29 panic!("Unreachable in back_to_user!");
30 }

```

- 第 11 行, 在 trap\_return 的开始处就调用 set\_user\_trap\_entry, 来让应用 Trap 到 S 的时候可以跳转到 \_\_alltraps。注: 我们把 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 12-13 行, 准备好 \_\_restore 需要两个参数: 分别是 Trap 上下文在应用地址空间中的虚拟地址和要继续执行的应用地址空间的 token。

最后我们需要跳转到 \_\_restore, 以执行: 切换到应用地址空间、从 Trap 上下文中恢复通用寄存器、sret 继续执行应用。它的关键在于如何找到 \_\_restore 在内核/应用地址空间中共同的虚拟地址。

- 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。这是因为, 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。

当每个应用第一次获得 CPU 使用权即将进入用户态执行的时候, 它的内核栈顶放置着我们在内核加载应用的时候构造的一个任务上下文:

```

// os/src/task/context.rs

impl TaskContext {
 pub fn goto_trap_return() -> Self {
 Self {
 ra: trap_return as usize,

```

(下页继续)

(续上页)

```
 s: [0; 12],
 }
 }
}
```

在 `__switch` 切换到该应用的任务上下文的时候，内核将会跳转到 `trap_return` 并返回用户态开始该应用的启动执行。

## 5.7.6 改进 sys\_write 的实现

类似 Trap 处理的改进，由于内核和应用地址空间的隔离，`sys_write` 不再能够直接访问位于应用空间中的数据，而需要手动查页表才能知道那些数据被放置在哪些物理页帧上并进行访问。

为此，页表模块 `page_table` 提供了将应用地址空间中一个缓冲区转化为在内核空间中能够直接访问的形式的辅助函数：

```
// os/src/mm/page_table.rs

1 pub fn translated_byte_buffer(
2 token: usize,
3 ptr: *const u8,
4 len: usize
5) -> Vec<&'static [u8]> {
6 let page_table = PageTable::from_token(token);
7 let mut start = ptr as usize;
8 let end = start + len;
9 let mut v = Vec::new();
10 while start < end {
11 let start_va = VirtAddr::from(start);
12 let mut vpn = start_va.floor();
13 let ppn = page_table
14 .translate(vpn)
15 .unwrap()
16 .ppn();
17 vpn.step();
18 let mut end_va: VirtAddr = vpn.into();
19 end_va = end_va.min(VirtAddr::from(end));
20 if end_va.page_offset() == 0 {
21 v.push(&mut ppn.get_bytes_array() [start_va.page_offset() ..]);
22 } else {
23 v.push(&mut ppn.get_bytes_array() [start_va.page_offset() .. end_va.page_
24 .offset()]);
25 }
26 start = end_va.into();
27 }
28 v
29 }
30 }
```

参数中的 `token` 是某个应用地址空间的 `token`，`ptr` 和 `len` 则分别表示该地址空间中的一段缓冲区的起始地址和长度（注：这个缓冲区的应用虚拟地址范围是连续的）。`translated_byte_buffer` 会以向量的形式返回一组可以在内核空间中直接访问的字节数组切片（注：这个缓冲区的内核虚拟地址范围有可能是不连续的），具体实现在这里不再赘述。

进而我们可以完成对 `sys_write` 系统调用的改造：

```
// os/src/syscall/fs.rs

pub fn sys_write(fd: usize, buf: *const u8, len: usize) -> isize {
 match fd {
 FD_STDOUT => {
 let buffers = translated_byte_buffer(current_user_token(), buf, len);
 for buffer in buffers {
 print!("{}", core::str::from_utf8(buffer).unwrap());
 }
 len as isize
 },
 _ => {
 panic!("Unsupported fd in sys_write!");
 }
 }
}
```

上述函数尝试将按应用的虚地址指向的缓冲区转换为一组按内核虚地址指向的字节数组切片构成的向量，然后把每个字节数组切片转化为字符串“&str”然后输出即可。

### 5.7.7 小结

这一章内容很多，讲解了 **地址空间**这一抽象概念是如何在一个具体的“头甲龙”操作系统中实现的。这里面的核心内容是如何建立基于页表机制的虚拟地址空间。为此，操作系统需要知道并管理整个系统中的物理内存；需要建立虚拟地址到物理地址映射关系的页表；并基于页表给操作系统自身和每个应用提供一个虚拟地址空间；并需要对管理应用的任务控制块进行扩展，确保能对应用的地址空间进行管理；由于应用和内核的地址空间是隔离的，需要有一个跳板来帮助完成应用与内核之间的切换执行；并导致了对异常、中断、系统调用的相应更改。这一系列的改进，最终的效果是编写应用更加简单了，且应用的执行或错误不会影响到内核和其他应用的正常工作。为了得到这些好处，我们需要比较费劲地进化我们的操作系统。如果同学结合阅读代码，编译并运行应用 + 内核，读懂了上面的文档，那完成本章的实验就有了一个坚实的基础。

如果同学能想明白如何插入/删除页表；如何在 `trap_handler` 下处理 `LoadPageFault`；以及 `sys_get_time` 在使能页机制下如何实现，那就会发现下一节的实验练习也许就和 **lab1** 一样。

## 5.8 超越物理内存的地址空间

### 5.8.1 本节导读

#### 有限的物理内存

到目前为止，在面向批处理系统的操作系统中，正在运行的任务只有一个，可以访问计算机的整个物理内存空间。如果物理内存空间不够，任务需要比较小心翼翼的申请和释放内存，来确保在当前时间段中内存够用。在面向多道程序的协作式操作系统或面向分时多任务的抢占式操作系统中，由于在内存中会有多个任务来共享整个物理内存，且任务之间的物理内存空间是隔离的，导致任务越多，每个任务可用的物理内存空间会越少。这样，在提高 CPU 利用率和任务间的隔离安全性的同时，一个比较严重的问题出现了：**物理内存不够用**。

---

**注解：**为什么要为应用程序支持巨大的地址空间？

在计算机发展的早期，计算机系统上的应用程序一般不需要巨大的地址空间，更需要的是强劲的 CPU 处理能力。但从二十世纪七十年代末开始的个人计算机一路发展至今，存在着某种动力使得人们如此频繁地更新计算机设备，这就是：

### 安迪比尔定律：比尔要拿走安迪所给的（What Andy gives, Bill takes away）

Intel 处理器的速度每十八个月翻一番，计算机内存和硬盘的容量以更快的速度在增长。过去的 MS DOS 操作系统和 DOS 应用程序在内存容量为 640KB 的计算机上顺畅地运行。而现在的 Windows 11 和 Windows 应用程序在内存容量 8GB 的计算机上仅勉强够用，而建议的内存配置是 16GB（是 640KB 的 25600 倍）。虽然新的软件功能比以前的版本强了一些，但其增加的功能和它的大小不成比例。

那大内存带来的好处是啥？**方便！**一般用户可以在计算机上同时开启多个应用：文字编辑、上网浏览、视频会议等，享受方便的数字生活。应用程序开发者不必担心要处理的数据是否有足够空间存储，只需编写程序，根据数据所需空间分配内存即可，甚至都不需要考虑释放内存的操作。

注：安迪是 Intel 公司 CEO 安迪·格鲁夫（Andy Grove），比尔是微软公司创始人比尔·盖茨（Bill Gates）。

## 超越物理内存的方法

我们需要突破物理内存有限的限制，这需要通过应用程序的软件编程技巧或操作系统与硬件结合的内存管理机制来解决。首先，我们可以看看基于应用程序本身的方法，即分时复用内存：即应用程序/运行时库动态地申请和释放内存，让不同的代码和数据在不同时间段内共享同一块内存空间。

另外一种方法是基于操作系统的方法，即把远大于且远慢于物理内存的存储设备（如硬盘、SSD 等）利用起来作为内存的一部分。如果把正在运行的任务所没用到的物理内存空间，比如处于等待状态的任务所使用的物理内存空间，甚至是正在运行的任务会较晚访问的数据所占用的内存空间，移出并暂存在存储设备中，那么应用程序可以访问的内存空间（虚拟的）就包括了存储设备的巨大容量了。

当然还有其他一些针对特定使用场景下的不太常用的方法，比如内存压缩方法。对于需要处理大数据的应用而言，内存中存放的主要是各种数据，而很多数据（如具有大量重复的值）是可以被压缩的。所以采用一定的压缩数据的结构和数据压缩方法，可以有效地减少数据占用的内存空间。

这些给应用程序带来好处的方法，也会引入各种运行时的开销，如果处理不当，会让任务本身，甚至整个系统的执行效率大大下降。所以，我们希望通过设计有效的机制和策略，能在扩大虚拟的内存容量的同时，保证应用程序和系统能够高效地运行。

### 5.8.2 分时复用内存

考虑到应用程序在其运行中的不同时间段内，会使用不同的数据，所以可以让这些数据在不同的时间段内共享同一内存空间。这就是分时复用内存的基本思路。当应用程序从操作系统中只能获得一块固定大小的有限内存空间（称为：空闲空间）后，应用程序一般会采用动态分配内存的方式来合理使用有限的物理内存。在本章的 [Rust 中的动态内存分配](#) 一节中，讲述了动态内存分配的基本概念。如何动态地管理空闲内存以提高内存的使用效率，是一个需要进一步讨论的问题。我们将讲述两种方法：

- 动态内存分配：由应用程序主动发出申请或释放内存的动态请求，由运行时库或应用程序本身通过一定的策略来管理空闲空间。
- 覆盖技术：一个应用程序中存在若干个功能上相对独立的程序段（函数集合），它们不会同时执行，所以它们可以共享同一块内存空间。这需要应用程序开发者根据需要手动移入或移出内存中的代码或数据。

## 动态内存分配

动态分配内存的目标是处理快速和浪费的空闲空间碎片少。这里我们会进一步分析动态分配内存的策略。对于不同的分配需求的前提情况，会有不同的分配策略。比如，每次分配的空间大小是固定的，那么就可以把空闲空间按固定大小的单元组织为一个列表。当需要分配时，从列表中的第一个单元取出即可；但需要回收时，把回收单元放到列表的末尾即可。

但应用程序需要的内存空间一般是大小不一的，这就需要相对复杂一些的策略来管理不同大小的空闲单元了。

### 最先匹配 (first fit) 策略

首次匹配策略的思路很简单：尽快找到满足用户需求的空闲块。大致处理流程就是把空闲空间基于地址顺序组织为大小不一的空闲块列表。当需求方需要大小为  $n$  的空闲块时，搜索空闲块列表，找到第一个足够大的空闲块；如果此空闲块大小刚好为  $n$ ，则把此空闲块返回给需求方；如果此空闲块大小大于  $n$ ，则把空闲块分割为大小为  $n$  的前面部分和剩余部分，大小为  $n$  的前面部分返回给需求方，而剩余部分会回归给空闲块列表管理，留给后续请求。

最先匹配策略不需要遍历空闲列表查找所有空闲块，所以有速度上的优势，但释放的时间有一定的随机性，这可能会让空闲列表开头的部分有很多小块。

### 最优匹配 (best fit) 策略

最优匹配 (best fit) 策略的思路也很简单：选择最接近需求方请求大小的块，从而尽量避免空间浪费。也是把空闲空间基于地址顺序组织为大小不一的空闲块列表。当需求方需要大小为  $n$  的空闲块时，遍历整个空闲列表，找到足够大且二者空间大小差距最小的空闲块。其代价是时间，即需要遍历一次空闲列表，找到最合适的空闲块。而且还有可能造成空间的浪费，因为可能产生很多小的空闲碎片，虽然碎片容量的总和比较大，但无法满足内存分配请求。

### 最差匹配 (worst fit) 策略

最差匹配 (worst fit) 策略的思路与最优匹配策略相反，它尝试找最大的空闲块，这也需要遍历一次空闲列表。当对找到的空闲块进行分割并满足用户需求后，剩余的空闲部分相对于最优匹配或最先匹配策略剩余的空闲部分相比，在大小上要大不少，可能被再次分配出去的概率要大。但最差匹配策略同样需要遍历整个空闲列表，带来较大的性能开销；且在应用频繁申请和释放的随机情况下，也可能会产生大量的碎片。

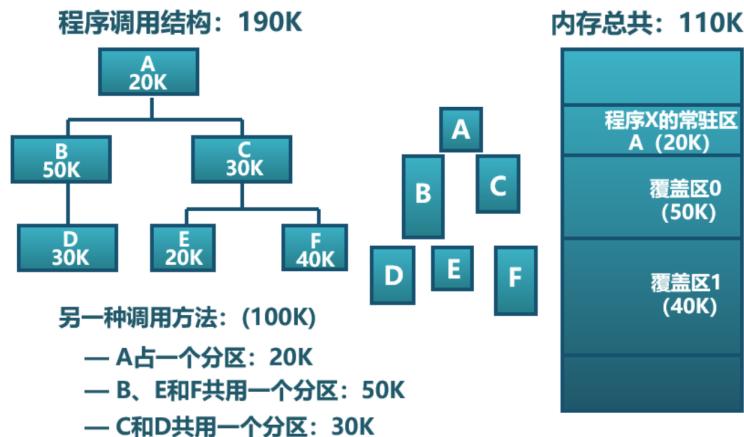
## 减少碎片

其实，采用上面的三种策略，都会由于对空闲块进行分割而产生碎片。所以减少碎片是一个需要考虑的重要问题。减少碎片有两种方法，第一种方法是合并操作，即在内存释放操作中，把在地址连续的多个空闲块合并为一个大的空闲块，这样可以满足更多的内存分配请求。

第二种方法是比较激进的紧致 (compaction，也称紧凑) 操作，即把地址不连续的多个空闲块移动在一个连续的地址空间中，形成一个大的空闲块。在移动空闲块的过程中，也需要移动已经分配的内存块。这除了带来性能上的影响外，还会带来数据寻址问题，即应用程序中数据所在的内存地址会发生改变，应用程序要能够感知这种变化，才能正确访问更新后的数据地址。这需要应用程序具有在紧致操作后的数据地址更新功能，这对应用程序开发者提出了很高的要求，不具有通用性。

## 覆盖 (Overlay) 技术

覆盖技术是指一个应用程序在执行过程中，其若干程序段分时共享同一块内存空间。覆盖技术的大致思路是编写一个应用程序，把它分为若干个功能上相对独立的程序段（函数集合），按照其逻辑执行结构，这些程序段不会同时执行。未执行的程序段先保存在存储设备上，当有关程序段的前一部分执行结束后，把后续程序段和数据调入内存，覆盖前面的程序段和数据。



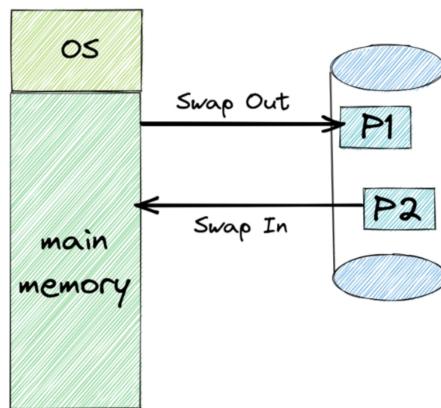
覆盖技术需要程序员在编写应用程序时，来手动控制和管理何时进行覆盖，不同程序段的覆盖顺序，以及覆盖的具体空间位置。这虽然减少了应用程序在执行过程中对内存空间的需求，当对应用程序开发者的编程水平提出了很高的要求，而且通用性比较差。

### 5.8.3 内存交换

既然我们有了操作系统，就应该在给应用更大内存空间的同时，降低应用程序开发者的开发难度。这样的代价就是增加操作系统内核的设计和实现复杂度。操作系统有两种扩大内存的方法。第一种方法是交换 (swapping) 技术，基本思路相对简单，即把一个应用程序从存储设备完整调入内存执行，该应用运行一段时间后，再把它暂停，并把它所占的内存空间（包括代码和数据）存回存储设备，从而释放出内存空间。另一种方法是虚拟内存 (virtual memory) 技术，基本思路是使应用程序在只有一部分代码和数据被调入内存的情况下运行。

## 交换技术

我们目前的操作系统还没实现这种技术，但不妨碍我们思考一下这种技术面临的问题和大致解决方法。在我们之前设计实现的批处理操作系统中，每次执行新应用时，需要把刚执行完毕的老应用所占内存空间给释放了，再把新应用的代码和数据换入到内存中。这是一种比较原始的应用之间交换地址空间（Swap Address Space）的方法。这种交换技术的方法也可进一步扩展用到多任务操作系统中，即把不常用或暂停的任务在内存中的代码和数据换出（Swap Out）存储设备上去。



首先是交换的时机，即何时需要发生交换？当一个任务在创建或执行过程中，如果内存不够用了，操作系统就需要执行内存交换操作。

第二个需要考虑的是交换的对象，即交换哪个任务？如果是一个刚创建的任务，在创建过程中内存不够用了，我们可以基于某种选择策略（如选择占用内存最多的那个任务）选择一个任务，把它所占用的内存换出到存储设备中，形成更多的空闲空间，直到被创建任务有足够的空间开始执行。如果是一个正在执行的当前任务动态需要更多的内存空间，操作系统有两种选择，一个选择与刚才的处理方式类似，找一个其他任务，并把其所占空间换出到存储设备中，从而满足当前任务的需要。另一个选择是把当前任务给暂停了（甚至有可能把它所占空间换出到存储设备上），然后让其他任务继续执行，并在执行完毕后，空出所占用的内存，并切换到下一个新任务执行，如果新任务在存储设备上，还需把新任务的代码和数据换入到内存中来；如果空闲内存能够满足被暂停的任务的内存分配需求，则唤醒该任务继续执行。

第三个需要考虑的是代码/数据的寻址，即任务被换出并被再次换入后，任务能否正确地执行？由于再次换入后，整个地址空间一般不可能正好被放入到同样的位置，这可以采用前述减少内存碎片的紧致技术中的地址更新方法，但实现复杂且不具有通用性。如果操作系统更新任务的页表内容，形成新的物理地址和虚拟地址映射关系。这样虽然任务换入后所在的物理地址是不同的，但可以让任务采用换出前同样的虚拟地址来执行和访问数据。

这里需要注意的是，即使解决了上述三个问题，但由于每次换入换出的数据量大（当然是由于任务的代码量和所访问的数据占用的内存空间大的原因），导致任务的执行会卡顿。这就需要以页为单位的内存换入换出技术—虚拟内存技术了。

## 虚拟内存技术

由于有了基于硬件的分页机制和 MMU/TLB 机制，操作系统可以通过页表为正在运行的任务提供一个虚拟的地址空间，这就是虚拟内存技术的基础之一。但仅有这个基础还不足以让应用程序访问的地址空间超越物理内存空间。我们还需要解决的一个关键问题：操作系统如何利用更大且更慢的存储设备，来 透明地给应用程序提供远超物理内存空间的一个虚拟的地址空间？

## 页面置换机制

为解决这个关键问题，虚拟内存技术其实还需要一系列的页面置换机制来支持：

- 作为交换区的存储设备：用于存放换出的部分物理内存中的代码和数据。
- 页表项存在位：当应用程序（也可称为任务）访问被换出到交换区的物理内存（对应页表项的存在位 V 为“0”）时，处理器会产生访存异常，让操作系统完成内存换入操作。
- 内存访问异常处理：操作系统判断任务的访存异常的地址是否属于任务地址空间，如果是属于，则把交换区对应物理内存页换入内存，使得任务可以继续正常运行。

## 作为交换区的存储设备

为了能够支持对物理内存的换入和换出操作，需要在存储设备上分配一个特定的分区或者文件，作为存放物理内存的交换区（swap space）。一般情况下，存储设备（如磁盘）的扇区大小为 N，而一个页大小为  $k \times N$ ，（k 和 N 为正整数）。为了支持以页为单位进行物理内存与交换区的换入换出操作，需要建立任务的页虚拟地址与多个磁盘扇区地址映射关系。这个映射关系可以建立在任务所在页表中对应虚地址的页表项中，即页表项中的内容是存储设备的扇区地址。这样操作系统在内存访问异常处理中，根据产生异常的虚拟地址，可快速查找到对应的页表项，并取出存储设备的扇区地址，从而可以准确地把存储在扇区中的物理内存换入到物理内存中。

另外，需要在存储设备上建立足够大小的交换区空间。这里的足够大小的值在一般情况下是一个经验值，比如是物理内存大小的 1.5~2 倍。这个值决定了整个系统中的所有应用程序能够使用的最大虚拟内存空间。

### 注解：按需分页（Demand Paging）

还有一点需要注意，交换区并不是应用程序的页面换入换出的唯一区域。由于应用程序的二进制代码文件一般存放在存储设备的文件系统中，当程序一开始执行时，该程序的代码段可以都不在物理内存中，这样当程序对应的任务创建后，一开始执行就会产生异常。操作系统在收到这样的异常后，会把代码段对应的内存一页一页地加载到内存中。则其实是一种内存访问的性能优化技术，称为“按需分页”（Demand Paging），即只有在任务确实需要的时候，才把所需数据/代码以页为单位逐步从存储设备换入到内存中。

## 页表项（Page Table Entry）

页表中的页表项记录了物理页号和对应页的各种属性信息，处理器根据虚拟地址中的虚页号（Virtual Page Number, VPN）为页表索引，可最终查找到虚拟地址所在的物理页位置。这是页表项的基本功能。当我们需要提供远大于物理地址空间的虚拟地址空间时，页表项中的内容能发挥新的作用。

我们重新梳理一下某任务让处理器访问被换出到存储设备上的数据所经历的过程。在处理器访问某数据之前，操作系统已把包含该数据的物理内存换出到了存储设备上，并需要提供关键的关联信息，便于操作系统后续的换入工作：

- 该数据的虚拟地址是属于某任务的地址空间：可在任务控制块中包含任务的合法空间范围的记录
- 该数据的页虚拟地址所对应的存储设备的扇区地址：可在页虚拟地址对应的页表项中包含存储设备的扇区地址的记录
- 该数据的虚拟页没有对应的物理页：在页虚拟地址对应的页表项中的存在位（Present Bit）置“0”，表示物理页不存在

在后续某时刻，该任务让处理器访问该数据时，首先处理器根据虚拟地址获得虚页号，然后检查 MMU 中的 TLB 中是否由匹配的项目，如果 TLB 未命中，则会进一步根据页表基址寄存器信息，查找内存中的页表，并根据 VPN 找到虚拟页对应的页表项。硬件会进一步查找该页表项的存在位，由于已经被操作系统设置为

“0”，表示该页不在物理内存中，处理器会产生“Page Fault”异常，并把控制权交给操作系统的“Page Fault”异常处理例程进行进一步处理。

## 内存访问异常处理

操作系统的“Page Fault”异常处理例程首先会判断，该数据的虚拟地址是否是该任务的合法地址？根据之前操作系统的设置，答案是“Yes”；然后取出保存在对应页表项中的扇区地址，把存储设备上交换区上对应的虚拟页内容读入到某空闲的物理内存页中；接着更新页表项内容，即把对应的物理页号写入页表项，把页表项中的存在位（Present Bit）置“1”；最后是返回该任务，让该任务重新执行访问数据的指令。这次处理器再次执行这条指令时，TLB 还是会没有命中，但由于对应页表项内容合法，所以 TLB 会缓存该页表项，并完成虚拟地址到物理地址的转换，完成访问数据指令的执行。

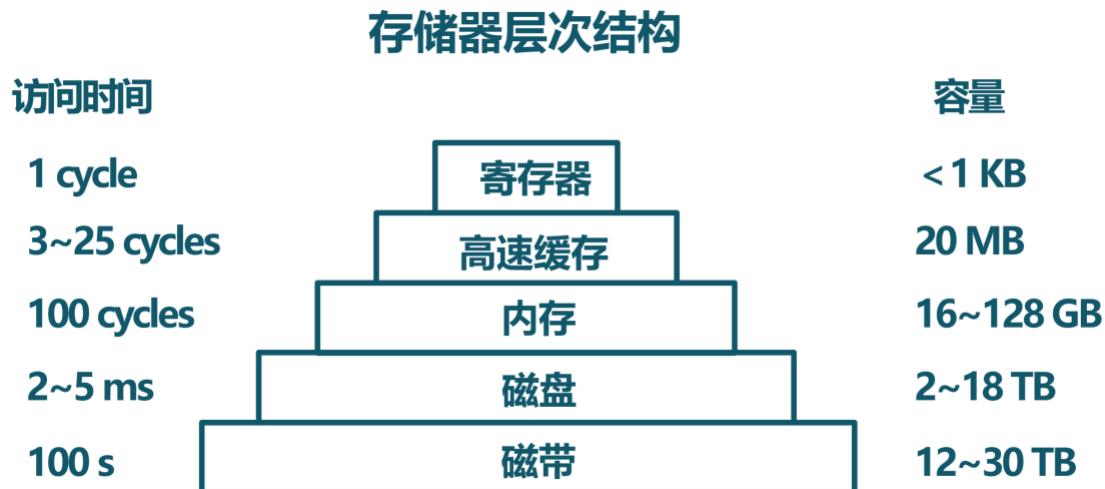
相对于内存访问，交换区的扇区访问要慢很多。为了进一步提高系统的执行效率，当操作系统在让存储设备进行 I/O 访问时，可将当前任务设置为阻塞状态，并切换其他可以运行的任务继续执行。通过这种任务调度方式，可以充分发挥多道程序和分时多任务的整体执行效率。

## 页面置换策略

当物理内存不够用的时候，我们还需要考虑把哪些内存换出去的问题，这就是下面会深入讨论的页面置换策略。置换策略通常会根据程序的访存行为、访存效率的评价指标和一些通用原则来进行设计。

## 内存层次结构与局部性原理

同学们在学习计算机组成原理和编程优化方法时，或多或少地都了解内存/存储的层次结构和局部性原理。



上图是一个典型的内存/存储的层次结构，上层比下层速度快但容量小，每一层都可以放置程序用到的代码和数据。这样的存储器层次结构设计是一种典型的工程思维，因为在成本等工程约束下，我们无法在构造出速度最快且容量最大的“理想”内存。我们希望基于这种层次结构，加上一定的软硬件优化手段，能够接近“理想”内存的目标。要达到这样的目标，需要理解局部性原理。

程序的局部性原理是指程序在执行时呈现出时空局部性的规律，即在一段时间内（时间特性），程序执行仅限于代码的某一部分，且执行所访问的内存空间也局限于某局部内存区域（空间特性）。

局部性可细分为时间局部性和空间局部性。时间局部性是指程序执行的某条指令在不久之后可能会再次被执行；程序访问的某数据在不久之后可能再次被访问。空间局部性是指当前程序访问某个内存单元，在不久之后可能会继续访问邻近的其他内存单元。

如果一个程序具有良好的局部性，那么处理器中的 Cache 就可以缓存程序常用的局部数据和代码，而相对不常用的数据和代码可以放在内存中，从而加快程序的运行效率。同理，操作系统也可以在内存中以页为单位来缓存程序常用的局部数据和代码。但物理内存有限，把哪些数据和代码换入到内存中或换出到存储设备的交换区，是一个值得探索的页面置换策略问题。

## 评价指标

页面置换策略可以由多种，这就需要对各种策略的优劣进行评价。我们希望程序访问的数据 快，最好是数据都在物理内存中。但这仅仅是理想，一旦数据不在物理内存，即访存未命中，就会产生“Page Fault”异常，并让操作系统从存储设备的交换区把数据缓慢地读入到内存中。

如何量化程序访问的速度呢？如果能知道程序执行过程中的访存命中次数和未命中次数，就可以计算出程序的平均内存访问时间：

$$AvgTime = (T_m * P_{hit} + T_s * P_{miss}) / (P_{hit} + P_{miss})$$

这里  $T_m$  是访问内存单元的时间， $T_s$  是访问存储设备的时间， $P_{hit}$  是访存命中次数， $P_{miss}$  是访存未命中次数。访存命中率是：

$$HitRatio = P_{hit} / (P_{hit} + P_{miss})$$

由于访问内存比访问存储设备快 2~3 个数量级，所以性能瓶颈是对存储设备读写次数，即访存未命中次数。只有减少访存未命中次数，才能提高访存命中率，并加快平均内存访问速度。所以页面置换策略的评价指标就是访存未命中次数或访存命中率。

## 策略范畴

对于页面置换策略，还需考虑是否会动态调整任务拥有的物理页数量，如果可以动态调整，这会对其他任务产生影响。如果操作系统给每个任务分配固定大小的物理页，在任务的执行过程中，不会动态调整任务拥有的物理页数量，那么对任务进行页面置换，不会影响到其他任务拥有的物理内存。这种情况下策略属于局部页面置换策略范畴，因为它只需考虑单个任务的内存访问情况。

如果操作系统通过某种页面置换策略可动态调整某任务拥有的物理内存大小，由于总体的物理内存容量是固定的，那就会影响到其他任务拥有的物理内存大小。这样的策略属于全局页面置换策略范畴。

全局页面置换策略可以在任务间动态地调整物理内存大小，通常比局部页面置换策略要效果好一些。而对于某些具体的页面置换策略，可既适用于局部页面置换策略范畴，也适用于全局页面置换策略范畴。

## 最优置换策略

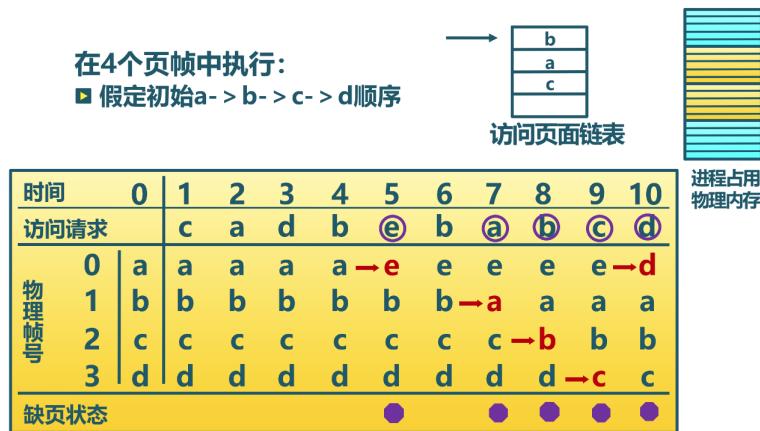
操作系统如果能够预先知道应用程序在执行过程中每次内存访问的虚拟地址序列，那就可以设计出一个最优置换策略，即可以让程序的未命中数量最小。其基本思路是选择一个应用程序在随后最长时间内不会被访问的虚拟页进行换出。当然，最长时间可以是无限长时间，表示随后不会再访问。

但操作系统怎么知道一个虚拟页随后多长时间不会被访问呢？在一般情况下当然不知道，所以最优置换策略是一种无法实际实现的策略。最优置换策略一般可作为标杆而存在，即我们可以把最优页面置换策略于其他可实现的策略进行性能比较，从而对各种策略的效果有一个相对的认识。

## FIFO 置换策略

早期操作系统为了避免尝试达到最优的计算复杂性，采用了非常简单的替换策略，如 FIFO（先入先出）置换策略。其基本思路是：由操作系统维护一个所有当前在内存中的虚拟页的链表，从交换区最新换入的虚拟页放在表尾，最久换入的虚拟页放在表头。当发生缺页中断时，淘汰/换出表头的虚拟页并把从交换区新换入的虚拟页加到表尾。

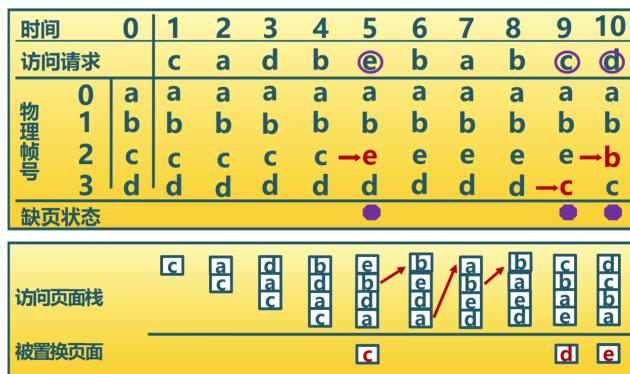
FIFO 置换策略虽然在实现上简单，但它对页访问的局部性感知不够，及时某页被多次访问，也可能由于它较早进入内存而被置换出去。



## LRU 置换策略

LRU (Least Recently Used, 最近使用最少) 置换策略是一种基于历史来预测未来的置换策略，其基本思路是：操作系统替换的是最近最少使用的虚拟页。“最近”是一个时间维度的量，表示过去的一段时间。精确地说，过去的这段时间是从程序开始执行到当前程序产生“Page Fault”异常的时间片段。虚拟页被访问的近期性 (Recency) 表示页被访问到目前产生异常这段时间的长短。这段时间越长，页被访问的近期性越弱，最弱即最近使用最少；反之，近期性就强。

访存的近期性是一种历史信息，也是程序局部性的一种体现，因为越近被访问过的页，也许在不久的将来再次被访问的可能性也就越大。所以 LRU 置换策略是一种以历史来预测未来的启发式方法。在一般情况下，对于具有访存局部性的应用程序，LRU 置换策略的效果接近最优置换策略的效果。



为了实现 LRU 置换策略，操作系统需要记录应用程序执行过程中的访存历史信息，这在实际实现上是一个挑战。LRU 置换策略需要记录应用访问的每页的最后一次被访问的时间。这样，当一个页必须被替换时，该

策略就选择最长时间没有被使用的页。这里的难点是如何准确记录访存的时间信息。我们可以想象一下，通过扩展硬件和软件功能的软硬结合的方式来记录：

### 硬件计数器方法

处理器添加一个计数器，且操作系统为每个页表条目添加一个使用时间字段。对于处理器的每次内存访问，计数器都会递增。每当对某虚拟页进行访问时，计数器的内容都会被处理器复制到该页对应的页表条目中的使用时间字段。这样，我们总有最后一次使用每页的“相对时间”。当尝试“Page Fault”异常并需要替换虚拟页时，操作系统搜索页表，找到具有最小的使用时间的页表项，并把该页表项对应的虚拟页替换出去。

这个方法对硬件资源要求很高，且执行开销高昂。为保证记录的使用时间有足够的表示范围，时计数器位数应该比较大，比如 64 位，如果太小，计数器容易溢出。这样每个页表项也会扩大一倍（假设原页表项占 64 bit）。在性能上的影响更大，处理器需要不时地对位于内存中的页表项进行写操作，且在替换时，还需执行一个开销很大的最小使用时间的页表项查找过程。这将大大影响计算机的造价和实际的性能，所以，该方法也就停留在理论的可行性上了。

### stack 方法

stack 方法是一种基于栈组织结构的算法。即把表示页（有虚拟页号即可）的项目按访存先后组织为一个栈式单链表，表头指向最近访问的页项目，表尾指向最久访问的页项目。每当应用访问一个页时，处理器就会从链表中找到该页对应项目，移除并放在链表头。这样，最近使用的页总是在栈式单链表的顶部，而最近最少使用的页总是在链表的底部。因为必须从栈的中间移除项目，所以一般通过链表组织来实现。

这个方法消耗更多内存，且执行开销很大。由于链表的容量，使得它只能放在内存中。如果由处理器硬件来处理，则对硬件资源要求很高，对于每次访问页，处理器都需要查找链表，并更新链表。如果由软件来处理，那操作系统需要感知每次的访问页情况，则需要处理器通过某种异常机制来通知，并在随后的异常处理过程中，完成对链表的查找和更新工作。这两种方法的执行时间开销都很大，所以 stack 方法也停留在理论的可行性上了。

问题：我们是否真的需要找到准确的最旧页来替换？在大多数情况下，找到差不多最旧的页是否也能获得预期的效果？

## Clock 置换策略

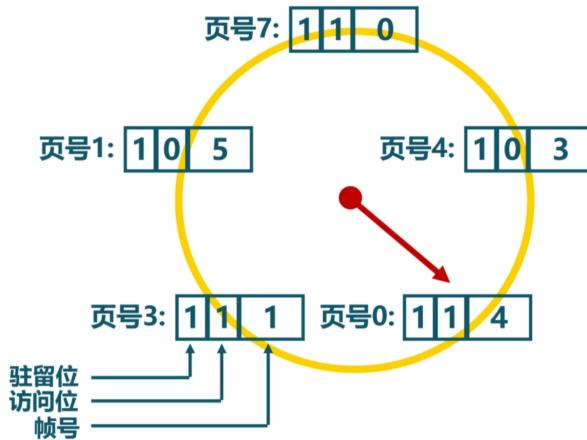
从计算机系统的工程设计（考虑成本）和操作系统设计（考虑执行开销）的角度来看，近似 LRU 策略更为可行。近似 LRU 策略的关键问题是：如何以尽量小的硬件开销和执行代价快速找到差不多最旧的页？

根据之前的[页面置换机制](#)中处理器访问数据的过程，我们可以了解到在访问数据前，想要查找位于 TLB 或内存中的页表项，所以我们可以对页表项中的某些位进行一定的扩展，用来表示应用访问内存的情况。如果要表示应用访问某页的情况，可以用一位就够了，“1”表示近期访问了某页，“0”表示近期没有访问某页。在最小硬件成本的前提下，我们就可在页表项中的一个 bit (use bit 使用位, 也称 reference bit 引用位) 来表示页表项对应的虚拟页的访问情况。

那么如何访问这个特别的 bit 呢？这里处理器和操作系统之间达成了一个访问约定：每当页被引用（即读或写）时，处理器硬件将把对应该页的页表项的使用位设置为 1。但是，硬件不会清除该位（即将其设置为 0），而这是由操作系统来负责的。

有了关于使用位的硬件设置和访问约定，我们就可以设计出近似 LRU 置换策略了。想象一下，应用所在地址空间中的所有有效页（页表项的存在位为 1）都放在一个环形循环列表中，一个指针像时钟的指针一样，会围绕环形列表旋转。该指针开始时指向某个特定的页。当必须进行页替换时，操作系统会查找指针当前指向的页对应的页表项，并检查该页表项的使用位是 1 还是 0。如果是 0，则意味着该页最近没被使用，适合被替换，指针递增到链表的下一项。如果是 1，则意味着该页最近被使用，因此不适合被替换，然后把使用位设置为 0，时钟指针递增到链表的下一页，一直持续到找到一个使用位为 0 的页。

这种近似 LRU 策略类似时钟旋转的过程，所以也称为 Clock (时钟，也称 Second-Chance 二次机会) 置换策略。虽然 Clock 置换策略不如 LRU 置换策略的效果好，但它比不考虑历史访问的方法要好，且在一般情况下，与 LRU 策略的结果对比相差不大。



时钟置换策略的一个小改进，是进一步额外关注内存中的页的修改情况。这样做的原因是：如果页已被修改 (modified，也称 dirty)，称为脏页，则在释放它之前须将它的更新内容写回交换区，这又增加了一次甚至多次缓慢的 I/O 写回操作。但如果它没有被修改 (clean)，就称为干净页，可以直接释放它，没有额外的 I/O 写回操作。因此，操作系统更倾向于先处理干净页，而不是脏页。

为了支持这种改进，页表项还应该扩展一个修改位 (modified bit，又名脏位，dirty bit)。处理器在写入页时，会设置对应页表项的此位为 1，操作系统会在合适的时机清除该位 (即将其设置为 0)。因此可以将该修改位作为该页近期是否被写的信息源。这样，改进的时钟置换策略，制定出新的优先级，即优先查找并清除未使用且干净的页 (第一类)；如无法找到第一类页，再查找并清除已使用且干净的页或未使用且脏的页 (第二类)；如果无法找到第二类的页，再查找并清除已使用且脏的页 (第三类)。

## 工作集置换策略

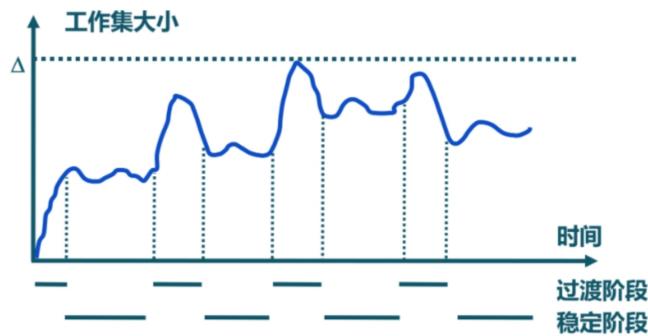
上述置换策略没有涉及动态调整某任务拥有的物理内存大小 (也称页帧数，Frame Number)。如果置换策略能动态调整任务拥有的物理内存大小，则可以在系统层面对其他任务拥有的物理内存产生影响。下面介绍的置换策略就具有这样的特征。

计算机科学家 Denning 在二十世纪六十年代就发现，大多数程序都不是均匀地访问它们的地址空间，而不同时间段的访问往往是集中在不同的小部分页面中。在程序执行的任一时刻，都存在一个动态变化的页面集合，它包含所有最近内存访问所访问过的页面。这个集合其实就是工作集 (working set)。

理解工作集置换策略的前提是先理解工作集的定义。一个任务当前正在使用的页面集合称为它的工作集 (working set，也称驻留集合)。如果整个工作集都被装入到了内存中，那么任务在运行到下一运行阶段之前，一般不会产生很多缺页中断。若内存太小而无法容纳下整个工作集，那么任务在后续运行过程中可能会产生大量的缺页中断，导致执行变慢。

注意，上述工作集的概念和对缺页中断的推断，其实是建立在程序执行具有局部性这个基础上的，也是一种根据历史来推测未来的启发式方法。为了基于工作集的特征来设计置换策略，我们需要量化工作集的概念。在  $t$  时刻，任务最近  $n$  次内存访问均发生在  $m$  个页面集合上，那么这个页面集合就是任务在  $t$  时刻最近  $k$  次内存访问下的工作集，用  $w(k, t)$  来表示。工作集中页面数量用  $|w(k, t)|$  表示。随着任务的执行，工作集中的页面会发生变化，其数量也会发生变化。如果一个任务占用的页面数与其工作集大小相等或超过工作集，则该任务可在一段时间内不会发生缺页异常。如果其在内存的页面数小于工作集，则发生缺页中断的频率将增加。

所以，工作集置换策略的目标就是动态调整工作集的内容和大小，一个任务占用的页面数接近其工作集大小，减少缺页异常次数。



实现工作集置换策略的基本思路是，操作系统能及时跟踪任务执行中位于工作集中页面，然后在发生缺页异常或过了一段时间间隔时，淘汰一个不在工作集中的页面，如果缺页异常就换入位于交换区中的页。

实现工作集置换策略的调整是及时获取工作集信息。有了工作集的定义并不意味着存在一种工程上可接受的有效方法，能够在程序运行期间及时准确地计算出工作集。

如果暂时不考虑硬件上的工程成本，我们可以在处理器上添加一个长度为  $k$  的移位寄存器，当前处理器每进行一次内存访问就把寄存器左移一位，然后在最右端插入刚才所访问过的页面号。这样把移位寄存器中的  $k$  个页面号按访问时间排序，并去除老的重复页面，形成的集合就是工作集。然而，维护移位寄存器并在缺页中断时处理它所需的开销很大，因此该技术仅仅停留在理论上。

其实，我们也可以参考近似 LRU 策略，提出近似工作集置换策略。一种可行的近似方法是，不向后找最近  $k$  次的内存访问，而改为查找一段固定时间被访问的页面。比如，工作集即是一个任务在过去  $\tau$  时间段中的内存访问所用到的页面集合。在一般情况下，对于过去  $\tau$  时间段的具体值是基于经验设置的，如 10ms。这里，我们可以根据页面对应页表项的访问位来判断该页在过去  $\tau$  时间段中是否被访问，如果该位为“1”，表示该页被访问，属于工作集，否则就不属于工作集。当然这种近似方法对工作集的跟踪比较粗略，我们可以建立每个访问页的时间戳链表，得到更准确的工作集信息，但这样的执行开销和空间成本就大了。

当缺页中断发生后，需要扫描整个页表才能确定被淘汰的页面，因此基本的工作集置换策略是比较费时的。Carr 和 Hennessey 在 1981 提出了一种基于时钟信息的改进的工作集置换策略，称为 WSClock (工作集时钟) 置换策略。由于它实现简单，性能较好，所以在实际工作中得到了广泛应用。该策略首先要建立一个以页框为元素的空的循环表。当操作系统加载第一个任务访问的页面后，把对应的页面信息加到该表中。随着更多页面的加入，它们形成一个更大容量的循环表，其中每个表项包含来自基本工作集置换策略中记录的上次使用时间和访问位信息（由处理器硬件置一，表示访问该页）。

每次缺页中断时，WSClock 置换策略首先检查指针指向的页面，如果指针指向的页面的存在位为 1，表示该页面在当前时钟滴答中就被使用过，那么该页面就不适合被淘汰。然后把该页面的存在位置为 0，并指针指向下一个页面，重复扫描循环表。如果指针指向的页面存在位为 0，再进一步检查该页面的生存时间大于  $\tau$ ，那它就不在工作集中，是属于被淘汰的页。

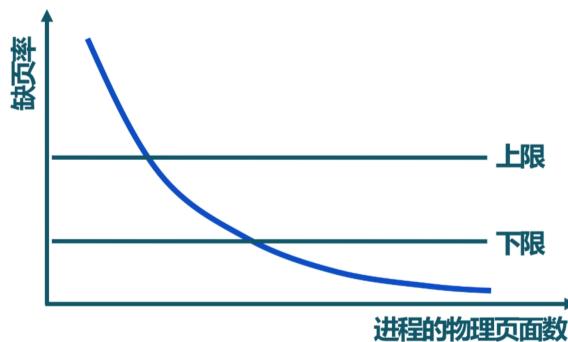
## 缺页率置换策略

在上面的各种置换策略中，或多或少涉及到对页面访问时间的记录和查找，排序等操作，开销很大。而置换策略的目标是减少缺页次数或缺页率。缺页率置换策略就是一种直接根据缺页率的变化来动态调整任务的物理内存大小的方法。如果缺页率高了，就增加任务占用的物理内存，如果缺页率低了，就减少任务占用的物理内存。任务占用的物理内存也称常驻集，即当前时刻，任务实际驻留在内存中的页面集合。

那如何计算缺页率呢？缺页率的定义如下：

$$\text{缺页率 (page fault rate)} = \text{缺页次数} / \text{访存次数}$$

要得到缺页率的精确值比较困难，主要是访存次数难以精确统计。我们可以采用一种近似的方法来表示缺页率。从上次缺页异常时间  $T_{last}$  到现在缺页异常时间  $T_{current}$  的时间间隔作为缺页率的当前指标。并用一个经验值  $T_s$  表示适中的缺页率。



这样，缺页率置换策略的基本思路就是：在任务访存出现缺页时，首先计算从上次缺页异常时间  $T_{last}$  到现在缺页异常时间  $T_{current}$  的时间间隔。然后判断，如果  $T_{current} - T_{last} > T_s$ ，则置换出在  $[T_{last}, T_{current}]$  时间内没有被引用的页，并增加缺失页到工作集中；如果  $T_{current} - T_{last} \leq T_s$ ，则只增加缺失页到工作集中。

在上述思路描述中，

- $T_{current} - T_{last} > T_s$  表示缺页率低了，通过置换出在  $[T_{last}, T_{current}]$  时间内没有被引用的页，来减少任务的常驻集。
- $T_{current} - T_{last} \leq T_s$  表示缺页率高了，需要增加任务的常驻集。
- 页是否被引用是根据任务访问的内存页对应的页表项的存在位信息来判断的。

### Belady 异常现象

计算机科学家 Belady 及其同事在 1969 年左右在研究 FIFO 置换策略时，发现了一个有趣的现象，对于一个内存访问序列：1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5，当物理页帧数从 3 增加为 4 时，访存命中率反而下降了。

我们通常理解，当一个任务的物理页帧数量变大时，访存的命中率是会提高的。但在上面的例子中，命中率反而下降了。这种异常现象被后人称为 Belady 异常 (Belady's Anomaly)。

而其他一些策略，比如 LRU 置换，最优置换等，就不会遇到这个问题。其原因是，LRU 等具有栈特性 (stack property)，即数量为  $m+1$  的物理页帧一定包括数量为  $m$  的物理页帧的数据内容。因此，当增加物理页帧数量时，访存命中率至少保证不变，且有可能提高。而 FIFO 策略、时钟策略等没有栈特性，因此可能出现异常行为。

### 5.8.4 小结

本节的内容比较多，我们介绍了超越物理内存的地址空间，涉及动态内存分配、虚拟内存等概念、机制和策略。如果由应用程序自己来做，会给应用程序员很大的开发负担。如果让操作系统来做，细节对应用程序都是透明的，应用程序员会很开心。但负担转移到了处理器硬件和操作系统上。我们需要对硬件进行扩展，如在页表项中增加 1bit 的存在位、访问位和修改位等。操作系统页错误处理例程 (page-fault handler) 会将需要的页从存储设备的交换区读取到内存，并可能还需要先换出内存中的一些页，为即将换入的页腾出空间。

我们还介绍了一些页置换策略，这些策略有些只存在理论中，并不实用，但其中的思路可以借鉴和发展，比如设计新的近似策略。在当前情况下，考虑到内存的成本在下降，购买更多的内存也是一种很实际的方法。

## 5.9 练习

### 5.9.1 课后练习

#### 编程题

1. \*\* 使用 sbrk, mmap, munmap, mprotect 内存相关系统调用的 linux 应用程序。
2. \*\*\* 修改本章操作系统内核，实现任务和操作系统内核共用同一张页表的单页表机制。
3. \*\*\* 扩展内核，支持基于缺页异常机制，具有 Lazy 策略的按需分页机制。
4. \*\*\* 扩展内核，支持基于缺页异常的 COW 机制。（初始时，两个任务共享一个只读物理页。当一个任务执行写操作后，两个任务拥有各自的可写物理页）
5. \*\*\* 扩展内核，实现 swap in/out 机制，并实现 Clock 置换算法或二次机会置换算法。
6. \*\*\* 扩展内核，实现自映射机制。

#### 问答题

1. \* 在使用高级语言编写用户程序的时候，手动用嵌入汇编的方法随机访问一个不在当前程序逻辑地址范围内的地址，比如向该地址读/写数据。该用户程序执行的时候可能会生什么？
2. \* 用户程序在运行的过程中，看到的地址是逻辑地址还是物理地址？从用户程序访问某一个地址，到实际内存中的对应单元被读/写，会经过什么样的过程，这个过程中操作系统有什么作用？（站在学过计算机组成原理的角度）
3. \* 覆盖、交换和虚拟存储有何异同，虚拟存储的优势和挑战体现在什么地方？
4. \* 什么是局部性原理？为何很多程序具有局部性？局部性原理总是正确的吗？为何局部性原理为虚拟存储提供了性能的理论保证？
5. \*\* 一条 load 指令，最多导致多少次页访问异常？尝试考虑较多情况。
6. \*\* 如果在页访问异常中断服务例程执行时，再次出现页访问异常，这时计算机系统（软件或硬件）会如何处理？这种情况可能出现吗？
7. \* 全局和局部置换算法有何不同？分别有哪些算法？
8. \* 简单描述 OPT、FIFO、LRU、Clock、LFU 的工作过程和特点（不用写太多字，简明扼要即可）
9. \*\* 综合考虑置换算法的收益和开销，综合评判在何种程序执行环境下使用何种算法比较合适？
10. \*\* Clock 算法仅仅能够记录近期是否访问过这一信息，对于访问的频度几乎没有记录，如何改进这一点？
11. \*\*\* 哪些算法有 belady 现象？思考 belady 现象的成因，尝试给出说明 OPT 和 LRU 等为何没有 belady 现象。
12. \* 什么是工作集？什么是常驻集？简单描述工作集算法的工作过程。
13. \* 请列举 SV39 页 \*\* 页表项的组成，结合课堂内容，描述其中的标志位有何作用 / 潜在作用？
14. \*\* 请问一个任务处理 10G 连续的内存页面，需要操作的页表实际大致占用多少内存（给出数量级即可）？
15. \*\* 缺页指的是进程访问页面时页面不在页表中或在页表中无效的现象，此时 MMU 将会返回一个中断，告知操作系统：该进程内存访问出了问题。然后操作系统可选择填补页表并重新执行异常指令或者杀死进程。操作系统基于缺页异常进行优化的两个常见策略中，其一是 Lazy 策略，也就是直到内存页面被访问才实际进行页表操作。比如，一个程序被执行时，进程的代码段理论上需要从磁盘加载到内存。但是操作系统并不会马上这样做，而是会保存.text 段在磁盘的位置信息，在这些代码第一次被执行时

才完成从磁盘的加载操作。另一个常见策略是 swap 页置换策略，也就是内存页面可能被换到磁盘上了，导致对应页面失效，操作系统在任务访问到该页产生异常时，再把数据从磁盘加载到内存。

- 哪些异常可能是缺页导致的？发生缺页时，描述与缺页相关的 CSR 寄存器的值及其含义。
  - Lazy 策略有哪些好处？请描述大致如何实现 Lazy 策略？
  - swap 页置换策略有哪些好处？此时页面失效如何表现在页表项 (PTE) 上？请描述大致如何实现 swap 策略？
16. \*\*为了防范侧信道攻击，本章的操作系统使用了双页表。但是传统的操作系统设计一般采用单页表，也就是说，任务和操作系统内核共用同一张页表，只不过内核对应的地址只允许在内核态访问。(备注：这里的单/双的说法仅为自创的通俗说法，并无这个名词概念，详情见 [KPTI](#))
- 单页表情况下，如何控制用户态无法访问内核页面？
  - 相对于双页表，单页表有何优势？
  - 请描述：在单页表和双页表模式下，分别在哪个时机，如何切换页表？

## 5.9.2 实验练习

实验练习包括实践作业和问答作业两部分。

### 实践作业

#### 重写 sys\_get\_time

引入虚存机制后，原来内核的 sys\_get\_time 函数实现就无效了。请你重写这个函数，恢复其正常功能。

#### mmap 和 munmap 匿名映射

mmap 在 Linux 中主要用于在内存中映射文件，本次实验简化它的功能，仅用于申请内存。

请实现 mmap 和 munmap 系统调用，mmap 定义如下：

```
fn sys_mmap(start: usize, len: usize, prot: usize) -> isize
```

- syscall ID: 222
- 申请长度为 len 字节的物理内存（不要求实际物理内存位置，可以随便找一块），将其映射到 start 开始的虚存，内存页属性为 prot
- **参数：**
  - start 需要映射的虚存起始地址，要求按页对齐
  - len 映射字节长度，可以为 0
  - prot：第 0 位表示是否可读，第 1 位表示是否可写，第 2 位表示是否可执行。其他位无效且必须为 0
- **返回值：**执行成功则返回 0，错误返回 -1
- **说明：**
  - 为了简单，目标虚存区间要求按页对齐，len 可直接按页向上取整，不考虑分配失败时的页回收。
- **可能的错误：**

- start 没有按页大小对齐
- prot & !0x7 != 0 (prot 其余位必须为 0)
- prot & 0x7 = 0 (这样的内存无意义)
- [start, start + len) 中存在已经被映射的页
- 物理内存不足

munmap 定义如下：

```
fn sys_munmap(start: usize, len: usize) -> isize
```

- syscall ID: 215
- 取消到 [start, start + len) 虚存的映射
- 参数和返回值请参考 mmap
- 说明：
  - 为了简单，参数错误时不考虑内存的恢复和回收。
- 可能的错误：
  - [start, start + len) 中存在未被映射的虚存。

TIPS：注意 prot 参数的语义，它与内核定义的 MapPermission 有明显不同！

## 实验要求

- 实现分支：ch4-lab
- 实验目录要求不变
- 通过所有测例

在 os 目录下 make run TEST=1 测试 sys\_get\_time，make run TEST=2 测试 map 和 unmap。

challenge: 支持多核。

## 问答作业

无

## 实验练习的提交报告要求

- 简单总结本次实验与上个实验相比你增加的东西。（控制在 5 行以内，不要贴代码）
- 完成问答问题。
- (optional) 你对本次实验设计及难度的看法。

## 5.10 练习参考答案

### 5.10.1 课后练习

#### 编程题

- \*\* 使用 sbrk, mmap, munmap, mprotect 内存相关系统调用的 linux 应用程序。

可以编写使用 sbrk 系统调用的应用程序，具体代码如下：

```
//user/src/ch4_sbrk.c
int main()
{
 printf("Test sbrk start.\n");
 uint64 PAGE_SIZE = 0x1000;
 uint64 origin_brk = sbrk(0);
 printf("origin break point = %p\n", origin_brk);
 uint64 brk = sbrk(PAGE_SIZE);
 if(brk != origin_brk) {
 return -1;
 }
 brk = sbrk(0);
 printf("one page allocated, break point = %p\n", brk);
 printf("try write to allocated page\n");
 char *new_page = (char *)origin_brk;
 for(uint64 i = 0; i < PAGE_SIZE; i++) {
 new_page[i] = 1;
 }
 printf("write ok\n");
 sbrk(PAGE_SIZE * 10);
 brk = sbrk(0);
 printf("10 page allocated, break point = %p\n", brk);
 sbrk(PAGE_SIZE * -11);
 brk = sbrk(0);
 printf("11 page DEALLOCATED, break point = %p\n", brk);
 printf("try DEALLOCATED more one page, should be failed.\n");
 uint64 ret = sbrk(PAGE_SIZE * -1);
 if(ret != -1) {
 printf("Test sbrk failed!\n");
 return -1;
 }
 printf("Test sbrk almost OK!\n");
 printf("now write to deallocated page, should cause page fault.\n");
 for(uint64 i = 0; i < PAGE_SIZE; i++) {
 new_page[i] = 2;
 }
 return 0;
}
```

使用 mmap、unmap 系统调用的应用代码可参考测例中的 ch4\_mmap0.c、ch4\_unmap0.c 等代码。

- \*\*\* 修改本章操作系统内核，实现任务和操作系统内核共用同一张页表的单页表机制。

要实现任务和操作系统内核通用一张页表，需要了解清楚内核地址空间和任务地址空间的布局，然后为每个任务在内核地址空间中单独分配一定的地址空间。

在描述任务的 struct proc 中添加新的成员 “kpgtbl”、“trapframe\_base”，前者用户保存内核页表，后者用于保存任务的 TRAPFRAME 虚地址。并增加获取内核页表的函数 “get\_kernel\_pagetable()”。

```

//os/proc.h
struct proc {
 enum procstate state; // Process state
 int pid; // Process ID
 pagetable_t pagetable; // User page table
 uint64 ustack;
 uint64 kstack; // Virtual address of kernel stack
 struct trapframe *trapframe; // data page for trampoline.S
 struct context context; // swtch() here to run process
 uint64 max_page;
 uint64 program_brk;
 uint64 heap_bottom;
 pagetable_t kpgtbl; // 增加kpgtbl, 用于保存内核页表
 uint64 trapframe_base; // 增加trapframe, 用于保存任务自己的trapframe
}
//os/vm.c
//增加get_kernel_pagetable函数, 返回内核页表
pagetable_t get_kernel_pagetable(){
 return kernel_pagetable;
}

```

让任务使用内核页表, 在内核地址空间中为每个任务分配一定的地址空间, 在 bin\_loader() 函数中修改任务的内存布局。

```

//os/loader.c
//修改任务的地址空间
pagetable_t bin_loader(uint64 start, uint64 end, struct proc *p, int num)
{
 //pagetable_t pg = uvmcreate(); //任务不创建自己的页表
 pagetable_t pg = get_kernel_pagetable(); //获取内核页表
 uint64 trapframe = TRAPFRAME - (num + 1) * PAGE_SIZE; //_
 →为每个任务依次指定TRAPFRAME
 if (mappages(pg, trapframe, PGSIZE, (uint64)p->trapframe,
 PTE_R | PTE_W) < 0) {
 panic("mappages fail");
 }
 if (!PGALIGNED(start)) {
 panic("user program not aligned, start = %p", start);
 }
 if (!PGALIGNED(end)) {
 // Fix in ch5
 warnf("Some kernel data maybe mapped to user, start = %p, end = %p",
 start, end);
 }
 end = PGROUNDUP(end);
 uint64 length = end - start;
 uint64 base_address = BASE_ADDRESS + (num * (p->max_page + 100)) * PAGE_SIZE; //_
 →设置任务的起始地址, 并为任务保留100个页用做堆内存
 if (mappages(pg, base_address, length, start,
 PTE_U | PTE_R | PTE_W | PTE_X) != 0) {
 panic("mappages fail");
 }
 p->pagetable = pg;
 uint64 ustack_bottom_vaddr = base_address + length + PAGE_SIZE;
 if (USTACK_SIZE != PAGE_SIZE) {
 // Fix in ch5
 panic("Unsupported");
 }
}

```

(下页继续)

(续上页)

```

 }
 mappages(pg, ustack_bottom_vaddr, USTACK_SIZE, (uint64)kalloc(),
 PTE_U | PTE_R | PTE_W | PTE_X);
 p->ustack = ustack_bottom_vaddr;
 p->trapframe->epc = base_address;
 p->trapframe->sp = p->ustack + USTACK_SIZE;
 p->max_page = PGROUNDDUP(p->ustack + USTACK_SIZE - 1) / PAGE_SIZE;
 p->program_brk = p->ustack + USTACK_SIZE;
 p->heap_bottom = p->ustack + USTACK_SIZE;
 p->trapframe_base = trapframe; //任务保存自己的 TRAPFRAME
 return pg;
}

```

在内核返回任务中使用任务自己的 TRAPFRAME。

```

//os/trap.c
void usertrapret()
{
 set_usertrap();
 struct trapframe *trapframe = curr_proc()->trapframe;
 trapframe->kernel_satp = r_satp(); // kernel page table
 trapframe->kernel_sp =
 curr_proc()->kstack + KSTACK_SIZE; // process's kernel stack
 trapframe->kernel_trap = (uint64)usertrap;
 trapframe->kernel_hartid = r_tp(); // unuesd
 w_sepc(trapframe->epc);
 // set up the registers that trampoline.S's sret will use
 // to get to user space.
 // set S Previous Privilege mode to User.
 uint64 x = r_sstatus();
 x &= ~SSTATUS_SPP; // clear SPP to 0 for user mode
 x |= SSTATUS_SPIE; // enable interrupts in user mode
 w_sstatus(x);
 // tell trampoline.S the user page table to switch to.
 uint64 satp = MAKE_SATP(curr_proc()->pagetable);
 uint64 fn = TRAMPOLINE + (userret - trampoline);
 tracef("return to user @ %p", trapframe->epc);
 ((void (*)(uint64, uint64))fn)(curr_proc()->trapframe_base, satp); // 使用任务自己的 TRAPFRAME
 //((void (*)(uint64, uint64))fn)(TRAPFRAME, satp);
}

```

3. \*\*\* 扩展内核，支持基于缺页异常机制，具有 Lazy 策略的按需分页机制。

在页面懒分配 (Lazy allocation of pages) 技术中，内存分配并不会立即发生，而是在需要使用内存时才分配，这样可以节省系统的资源并提高程序的性能。

实现页面懒分配的思路是：当调用 sbrk 时不分配实际的页面，而是仅仅增大堆的大小，当实际访问页面时，就会触发缺页异常，此时再申请一个页面并映射到页表中，这时再次执行触发缺页异常的代码就可以正常读写内存了。

注释掉 growproc() 函数，增加堆的 size，但不实际分配内存：

```

//os/syscall.c
uint64 sys_sbrk(int n)
{
 uint64 addr;

```

(下页继续)

(续上页)

```

struct proc *p = curr_proc();
addr = p->program_brk;
int heap_size = addr + n - p->heap_bottom;
if(heap_size < 0){
 errorf("out of heap_bottom\n");
 return -1;
}
else{
 p->program_brk += n; //增加堆的size, 但不实际分配内存
 if(n < 0){
 printf("uvmdealloc\n");
 uvmdealloc(p->pagetable, addr, addr + n); // ←如果减少内存则调用内存释放函数
 }
 //if(growproc(n) < 0) //注释掉growproc()函数, 不实际分配内存
 // return -1;
 return addr;
}

```

因为没有给虚拟地址实际分配内存, 所以当对相应的虚拟地址的内存进行读写的时候会触发缺页错误, 这时再实际分配内存:

```

//os/loader.c
void usertrap()
{
 set_kerneltrap();
 struct trapframe *trapframe = curr_proc()->trapframe;
 tracef("trap from user epc = %p", trapframe->epc);
 if ((r_sstatus() & SSTATUS_SPP) != 0)
 panic("usertrap: not from user mode");
 uint64 cause = r_scause();
 if (cause & (1ULL << 63)) {
 cause &= ~(1ULL << 63);
 switch (cause) {
 case SupervisorTimer:
 tracef("time interrupt!");
 set_next_timer();
 yield();
 break;
 default:
 unknown_trap();
 break;
 }
 } else {
 switch (cause) {
 case UserEnvCall:
 trapframe->epc += 4;
 syscall();
 break;
 case StorePageFault: // 读缺页错误
 case LoadPageFault: // 写缺页错误
 {
 uint64 addr = r_stval(); // 获取发生缺页错误的地址
 if(lazy_alloc(addr) < 0){ // 调用页面懒分配函数
 errorf("lazy_aalloc() failed!\n");
 }
 }
 }
 }
}

```

(下页继续)

(续上页)

```

 exit(-2);
 }
 break;
}
case StoreMisaligned:
case InstructionMisaligned:
case InstructionPageFault:
case LoadMisaligned:
 errorf("%d in application, bad addr = %p, bad instruction = %p, "
 "core dumped.",
 cause, r_stval(), trapframe->epc);
 exit(-2);
 break;
case IllegalInstruction:
 errorf("IllegalInstruction in application, core dumped.");
 exit(-3);
 break;
default:
 unknown_trap();
 break;
}
}
usertrapret();
}

```

实现页面懒分配函数，首先判断地址是否在堆的范围内，然后分配实际的内存，最后在页面中建立映射：

```

//os/trap.c
int lazy_alloc(uint64 addr){
 struct proc *p = curr_proc();
 // 通过两个if判断发生缺页错误的地址是否在堆的范围内，不在则返回
 if (addr >= p->program_brk) {
 errorf("lazy_alloc: access invalid address");
 return -1;
 }
 if (addr < p->heap_bottom) {
 errorf("lazy_alloc: access address below stack");
 return -2;
 }
 uint64 va = PGROUNDDOWN(addr);
 char* mem = kalloc(); // 调用kalloc()实际分配页面
 if (mem == 0) {
 errorf("lazy_alloc: kalloc failed");
 return -3;
 }
 memset(mem, 0, PGSIZE);
 if(mappages(p->pagetable, va, PGSIZE, (uint64)mem, PTE_W|PTE_X|PTE_R|PTE_U) != 0)
 →{ // 将新分配的页面和虚拟地址在页表中建立映射
 kfree(mem);
 return -4;
 }
 return 0;
}

```

4. \*\*\* 扩展内核，支持基于缺页异常的 COW 机制。(初始时，两个任务共享一个只读物理页。当一个任务执行写操作后，两个任务拥有各自的可写物理页)

COW (Copy on Write) 是指当需要在内存中创建一个新的副本时，COW 技术会推迟复制操作，直到数

据被修改为止。从而减少不必要的内存拷贝，提升性能。

实现 COW 的思路是：在创建内存副本时，在内存中创建一个指向原始数据的指针或引用，而不是创建原始数据的完整副本。如果原始数据没有被修改，新副本将继续共享原始数据的指针或引用，以节省内存。当某个程序试图修改数据时，COW 技术会在新副本中复制原始数据，使得每个程序都有自己的独立副本，从而避免数据之间的干扰。

增加一个当做计数器的数据结构用于记录每个物理页面被多少变量引用，当页面初始被分配时计数器设置为 1，其后如果产生副本则计数器加 1。当页面被释放的时候则计数器减 1，如果计数器不为 0，说明还有其他引用在使用该页面，此时不执行实际的释放操作，最后计数器变为 0 时才真正释放页面：

```
//os/kalloc.c
uint64 page_ref[(PHYSTOP - KERNBASE) / PAGE_SIZE] = {0}; //_
// 定义用来记录页面引用的计数器，并将其值初始化为0
// 新增修改页面计数器的函数
void page_ref_add(uint64 pa, int n){ // 增加页面计数
 page_ref[(PGRONDOWN(pa) - KERNBASE) / PGSIZE] += n;
}
void page_ref_reduce(uint64 pa, int n){ // 减少页面计数
 page_ref[(PGRONDOWN(pa) - KERNBASE) / PGSIZE] -= n;
}
uint64 page_ref_get(uint64 pa){ // 返回页面计数
 return page_ref[(PGRONDOWN(pa) - KERNBASE) / PGSIZE];
}
void *kalloc()
{
 struct linklist *l;
 l = kmem.freelist;
 if (l) {
 kmem.freelist = l->next;
 memset((char *)l, 5, PGSIZE); // fill with junk
 page_ref_add((uint64)l, 1); // 在页面分配的时候设置计数器为1
 }
 return (void *)l;
}
void kfree(void *pa)
{
 struct linklist *l;
 if (((uint64)pa % PGSIZE) != 0 || (char *)pa < ekernel ||
 (uint64)pa >= PHYSTOP)
 panic("kfree");
 if (page_ref_get((uint64)pa) > 1){ //_
// 判断计数器的值，如果大于1说明还有其他引用，计数器减1后直接返回
 page_ref_reduce((uint64)pa, 1);
 return;
 }
 // Fill with junk to catch dangling refs.
 memset(pa, 1, PGSIZE);
 l = (struct linklist *)pa;
 l->next = kmem.freelist;
 kmem.freelist = l;
}
```

修改内存复制函数 umcopy()，其实不进行实际的内存复制，只是增加新的引用到需要复制的内存上：

```
//os/vm.c
int uvmcopy(pagetable_t old, pagetable_t new, uint64 max_page)
{
```

(下页继续)

(续上页)

```

pte_t *pte;
uint64 pa, i;
uint flags;
//char *mem;
for (i = 0; i < max_page * PAGE_SIZE; i += PGSIZE) {
 if ((pte = walk(old, i, 0)) == 0)
 continue;
 if ((*pte & PTE_V) == 0)
 continue;
 pa = PTE2PA(*pte);
 flags = PTE_FLAGS(*pte);
 *pte = ((*pte) & (~PTE_W)) | PTE_COW; //←
 ↪虽然不进行内存页的复制，但是需要修改内存页的操作权限，取消页的写操作权限，同时增加COW权限
 /*if ((mem = kalloc()) == 0) //注释掉分配内存的函数
 goto err;
 memmove(mem, (char *)pa, PGSIZE);
 if (mappages(new, i, PGSIZE, (uint64)mem, flags) != 0) {*/
 if (mappages(new, i, PGSIZE, (uint64)pa, (flags & (~PTE_W)) | PTE_COW) !=
 0) { //让另一页表中的虚拟地址指向原来页表中的物理地址
 //kfree(mem);
 goto err;
 }
 page_ref_add(pa, 1);
}
return 0;
err:
uvmunmap(new, 0, i / PGSIZE, 1);
return -1;
}

```

因为没有实际地进行内存复制，且取消了页面的写权限，所以当对相应的虚拟地址的内存进行写操作的时候会触发缺页错误，这时再调用 cowcopy() 函数实际分配页或修改页的写权限：

```

//os/trap.c
void usertrap()
{
 set_kerneltrap();
 struct trapframe *trapframe = curr_proc()->trapframe;
 tracef("trap from user epc = %p", trapframe->epc);
 if ((r_sstatus() & SSTATUS_SPP) != 0)
 panic("usertrap: not from user mode");
 uint64 cause = r_scause();
 if (cause & (1ULL << 63)) {
 cause &= ~(1ULL << 63);
 switch (cause) {
 case SupervisorTimer:
 tracef("time interrupt!");
 set_next_timer();
 yield();
 break;
 default:
 unknown_trap();
 break;
 }
 } else {
 switch (cause) {

```

(下页继续)

(续上页)

```

case UserEnvCall:
 trapframe->epc += 4;
 syscall();
 break;
case StorePageFault: { // 写缺页错误
 uint64 va = r_stval(); // 获取发生缺页错误的虚拟地址
 if (cowcopy(va) == -1) { // ←
 // 当发生写缺页错误的时候，调用COW函数，进行实际的内存复制
 errorf("Copy on Write Failed!\n");
 exit(-2);
 }
 break;
}
case StoreMisaligned:
case InstructionMisaligned:
case InstructionPageFault:
case LoadMisaligned:
case LoadPageFault:
 errorf("%d in application, bad addr = %p, bad instruction = %p, "
 "core dumped.",
 cause, r_stval(), trapframe->epc);
 exit(-2);
 break;
case IllegalInstruction:
 errorf("IllegalInstruction in application, core dumped.");
 exit(-3);
 break;
default:
 unknown_trap();
 break;
}
}
usertrapret();
}

```

实现 cowcopy() 分配函数，首先判断地址是否在堆的范围内，然后分配实际的内存，最后在页面中建立映射：

```

//os/vm.c
int cowcopy(uint64 va) {
 va = PGROUNDDOWN(va);
 pagetable_t p = curr_proc()->pagetable;
 pte_t* pte = walk(p, va, 0);
 uint64 pa = PTE2PA(*pte);
 uint flags = PTE_FLAGS(*pte); // 获取页面的操作权限
 if (!(flags & PTE_COW)) {
 printf("not cow\n");
 return -2; // not cow page
 }
 uint ref = page_ref_get(pa); // 获取页面的被引用的次数
 if (ref > 1) { // 若果大于1则说明有多个引用，这时需要重新分配页面
 // ref > 1, alloc a new page
 char* mem = kalloc();
 if (mem == 0) {
 errorf("kalloc failed!\n");
 return -1;
 }
 }
}

```

(下页继续)

(续上页)

```

memmove(mem, (char*)pa, PGSIZE); // 复制页中的内容到新的页
if(mappages(p, va, PGSIZE, (uint64)mem, (flags & (~PTE_COW)) | PTE_W) !=_
↪0) {
 errorf("mappage failed!\n");
 kfree(mem);
 return -1;
}
page_ref_reduce(pa, 1);
} else {
 // ref = 1, use this page directly
 *pte = ((*pte) & (~PTE_COW)) | PTE_W; //_
↪如果没有其他引用则修改页面操作权限, 使得该页面可以进行写操作
}
return 0;
}

```

5. \*\*\* 扩展内核, 实现 swap in/out 机制, 并实现 Clock 置换算法或二次机会置换算法。
6. \*\*\* 扩展内核, 实现自映射机制。

## 问答题

1. \* 在使用高级语言编写用户程序的时候, 手动用嵌入汇编的方法随机访问一个不在当前程序逻辑地址范围内的地址, 比如向该地址读/写数据。该用户程序执行的时候可能会生什么?

可能会报出缺页异常.

2. \* 用户程序在运行的过程中, 看到的地址是逻辑地址还是物理地址? 从用户程序访问某一个地址, 到实际内存中的对应单元被读/写, 会经过什么样的过程, 这个过程中操作系统有什么作用? (站在学过计算机组成原理的角度)

逻辑地址。这个过程需要经过页表的转换, 操作系统会负责建立页表映射。实际程序执行时的具体 VA 到 PA 的转换是在 CPU 的 MMU 之中进行的。

3. \* 覆盖、交换和虚拟存储有何异同, 虚拟存储的优势和挑战体现在什么地方?

它们都是采取层次存储的思路, 将暂时不用的内存放到外存中去, 以此来缓解内存不足的问题。

不同之处: 覆盖是程序级的, 需要程序员自行处理。交换则不同, 由 OS 控制交换程序段。虚拟内存也由 OS 和 CPU 来负责处理, 可以实现内存交换到外存的过程。

虚拟存储的优势: 1. 与段/页式存储完美契合, 方便非连续内存分配。2. 粒度合适, 比较灵活。兼顾了覆盖和交换的好处: 可以在较小粒度上置换; 自动化程度高, 编程简单, 受程序本身影响很小。(覆盖的粒度受限于程序模块的大小, 对编程技巧要求很高。交换粒度较大, 受限于程序所需内存。尤其页式虚拟存储, 几乎不受程序影响, 一般情况下, 只要置换算法合适, 表现稳定、高效) 3. 页式虚拟存储还可以同时消除内存外碎片并将内碎片限制在一个页面大小以内, 提高空间利用率。

虚拟存储的挑战: 1. 依赖于置换算法的性能。2. 相比于覆盖和交换, 需要比较高的硬件支持。3. 较小的粒度在面临大规模的置换时会发生多次较小规模置换, 降低效率。典型情况是程序第一次执行时的大量 page fault, 可配合预取技术缓解这一问题。

4. \* 什么是局部性原理? 为何很多程序具有局部性? 局部性原理总是正确的吗? 为何局部性原理为虚拟存储提供了性能的理论保证?

局部性分时间局部性和空间局部性 (以及分支局部性)。局部性的原理是程序经常对一块相近的地址进行访问或者是对一个范围内的指令进行操作。局部性原理不一定是一直正确的。虚拟存储以页为单位, 局部性使得数据和指令的访存局限在几页之中, 可以避免页的频繁换入换出的开销, 同时也符合 TLB 和 cache 的工作机制。

5. \*\*一条 load 指令，最多导致多少次页访问异常？尝试考虑较多情况。

考虑多级页表的情况。首先指令和数据读取都可能缺页。因此指令会有 3 次访存，之后的数据读取除了页表页缺失的 3 次访存外，最后一次还可以出现地址不对齐的异常，因此可以有 7 次异常。若考更加极端的情况，也就是页表的每一级都是不对齐的地址并且处在两页的交界处 (straddle)，此时一次访存会触发 2 次读取页面，如果这两页都缺页的话，会有更多的异常次数。

6. \*\*如果在页访问异常中断服务例程执行时，再次出现页访问异常，这时计算机系统（软件或硬件）会如何处理？这种情况可能出现吗？

我们实验的 os 在此时不支持内核的异常中断，因此此时会直接 panic 掉，并且这种情况在我们的 os 中这种情况不可能出现。像 linux 系统，也不会出现嵌套的 page fault。

7. \* 全局和局部置换算法有何不同？分别有哪些算法？

全局页面置换算法：可动态调整某任务拥有的物理内存大小；影响其他任务拥有的物理内存大小。例如：工作集置换算法，缺页率置换算法。

局部页面置换算法：每个任务分配固定大小的物理页，不会动态调整任务拥有的物理页数量；只考虑单个任务的内存访问情况，不影响其他任务拥有的物理内存。例如：最优置换算法、FIFO 置换算法、LRU 置换算法、Clock 置换算法。

8. \* 简单描述 OPT、FIFO、LRU、Clock、LFU 的工作过程和特点 (不用写太多字，简明扼要即可)

OPT：选择一个应用程序在随后最长时间内不会被访问的虚拟页进行换出。性能最佳但无法实现。

FIFO：由操作系统维护一个所有当前在内存中的虚拟页的链表，从交换区最新换入的虚拟页放在表尾，最久换入的虚拟页放在表头。当发生缺页中断时，淘汰/换出表头的虚拟页并把从交换区新换入的虚拟页加到表尾。实现简单，对页访问的局部性感知不够。

LRU：替换的是最近最少使用的虚拟页。实现相对复杂，但考虑了访存的局部性，效果接近最优置换算法。

Clock：将所有有效页放在一个环形循环列表中，指针根据页表项的使用位（0 或 1）寻找被替换的页面。考虑历史访问，性能略差于但接近 LRU。

LFU：当发生缺页中断时，替换访问次数最少的页面。只考虑访问频率，不考虑程序动态运行。

9. \*\*综合考虑置换算法的收益和开销，综合评判在何种程序执行环境下使用何种算法比较合适？

FIFO 算法：在内存较小的系统中，FIFO 算法可能是一个不错的选择，因为它的实现简单，开销较小，但是会存在 Belady 异常。

LRU 算法：在内存容量较大、应用程序具有较强的局部性时，LRU 算法可能是更好的选择，因为它可以充分利用页面的访问局部性，且具有较好的性能。

Clock 算法：当应用程序中存在一些特殊的内存访问模式时，例如存在循环引用或者访问模式具有周期性时，Clock 算法可能会比较适用，因为它能够处理页面的访问频率。

LFU 算法：对于一些需要对内存访问进行优先级调度的应用程序，例如多媒体应用程序，LFU 算法可能是更好的选择，因为它可以充分考虑页面的访问频率，对重要性较高的页面进行保护，但是实现比较复杂。

10. \*\*Clock 算法仅仅能够记录近期是否访问过这一信息，对于访问的频度几乎没有记录，如何改进这一点？

如果想要改进这一点，可以将 Clock 算法和计数器结合使用。具体做法是为每个页面设置一个计数器，记录页面在一段时间内的访问次数，然后在置换页面时，既考虑页面最近的访问时间，也考虑其访问频度。当待缓存对象在缓存中时，把其计数器的值加 1。同时，指针指向该对象的下一个对象。若不在缓存中时，检查指针指向对象的计数器。如果是 0，则用待缓存对象替换该对象；否则，把计数器的值减 1，指针指向下一个对象。如此直到淘汰一个对象为止。由于计数器的值允许大于 1，所以指针可能循环多遍才淘汰一个对象。

11. \*\*\* 哪些算法有 belady 现象？思考 belady 现象的成因，尝试给出说明 OPT 和 LRU 等为何没有 belady 现象。

FIFO 算法、Clock 算法。

页面调度算法可分为堆栈式和非堆栈式，LRU、LFU、OPT 均为堆栈类算法，FIFO、Clock 为非堆栈类算法，只有非堆栈类才会出现 Belady 现象。

12. \* 什么是工作集？什么是常驻集？简单描述工作集算法的工作过程。

工作集为一个进程当前正在使用的逻辑页面集合，可表示为二元函数  $W(t, \Delta)$ ， $t$  为执行时刻， $\Delta$  称为工作集窗口，即一个定长的页面访问时间窗口， $W(t, \Delta)$  是指在当前时刻  $t$  前的  $\Delta$  时间窗口中的所有访问页面所组成的集合， $|W(t, \Delta)|$  为工作集的大小，即页面数目。

13. \* 请列举 SV39 页 \*\* 页表项的组成，结合课堂内容，描述其中的标志位有何作用 / 潜在作用？

[63:54] 为保留项，[53:10] 为 44 位物理页号，最低的 8 位 [7:0] 为标志位。

- V(Valid): 仅当位 V 为 1 时，页表项才是合法的；
- R(Read)/W(Write)/X(eXecute): 分别控制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): 控制索引到这个页表项的对应虚拟页面是否在 CPU 处于 U 特权级的情况下是否被允许访问；
- A(Accessed): 处理器记录自从页表项上的这一位被清零之后，页表项的对应虚拟页面是否被访问过；
- D(Dirty): 处理器记录自从页表项上的这一位被清零之后，页表项的对应虚拟页面是否被修改过。

14. \*\* 请问一个任务处理 10G 连续的内存页面，需要操作的页表实际大致占用多少内存(给出数量级即可)？  
大致占用 ‘10G/512=20M’ 内存。

15. \*\* 缺页指的是进程访问页面时页面不在页表中或在页表中无效的现象，此时 MMU 将会返回一个中断，告知操作系统：该进程内存访问出了问题。然后操作系统可选择填补页表并重新执行异常指令或者杀死进程。操作系统基于缺页异常进行优化的两个常见策略中，其一是 Lazy 策略，也就是直到内存页面被访问才实际进行页表操作。比如，一个程序被执行时，进程的代码段理论上需要从磁盘加载到内存。但是操作系统并不会马上这样做，而是会保存.text 段在磁盘的位置信息，在这些代码第一次被执行时才完成从磁盘的加载操作。另一个常见策略是 swap 页置换策略，也就是内存页面可能被换到磁盘上了，导致对应页面失效，操作系统在任务访问到该页产生异常时，再把数据从磁盘加载到内存。

1. 哪些异常可能是缺页导致的？发生缺页时，描述与缺页相关的 CSR 寄存器的值及其含义。

- 答案：*mcause* 寄存器中会保存发生中断异常的原因，其中 *Exception Code* 为 12 时发生指令缺页异常，为 15 时发生 *store/AMO* 缺页异常，为 13 时发生 *load* 缺页异常。

CSR 寄存器：

- *scause*: 中断/异常发生时，CSR 寄存器 *scause* 中会记录其信息，*Interrupt* 位记录是中断还是异常，*Exception Code* 记录中断/异常的种类。
- *sstatus*: 记录处理器当前状态，其中 *SPP* 段记录当前特权等级。
- *stvec*: 记录处理 *trap* 的入口地址，现有两种模式 *Direct* 和 *Vectorized*。
- *sscratch*: 其中的值是指向 *hart* 相关的 S 态上下文的指针，比如内核栈的指针。
- *sepc*: *trap* 发生时会将当前指令的下一条指令地址写入其中，用于 *trap* 处理完成后返回。
- *stval*: *trap* 发生进入 S 态时会将异常信息写入，用于帮助处理 *trap*，其中会保存导致缺页异常的虚拟地址。

2. Lazy 策略有哪些好处？请描述大致如何实现 Lazy 策略？
- 答案：Lazy 策略一定不会比直接加载策略慢，并且可能会提升性能，因为可能会有些页面被加载后并没有进行访问就被释放或替代了，这样可以避免很多无用的加载。分配内存时暂时不进行分配，只是将记录下来，访问缺页时会触发缺页异常，在‘trap handler’中处理相应的异常，在此时将内存加载或分配即可。
3. swap 页置换策略有哪些好处？此时页面失效如何表现在页表项 (PTE) 上？请描述大致如何实现 swap 策略？
- 答案：可以为用户程序提供比实际物理内存更大的内存空间。页面失效会将标志位‘V’置为‘0’。将置换出的物理页面保存在磁盘中，在之后访问再次触发缺页异常时将该页面写入内存。
16. \*\* 为了防范侧信道攻击，本章的操作系统使用了双页表。但是传统的操作系统设计一般采用单页表，也就是说，任务和操作系统内核共用同一张页表，只不过内核对应的地址只允许在内核态访问。(备注：这里的单/双的说法仅为自创的通俗说法，并无这个名词概念，详情见 [KPTI](#))
1. 单页表情况下，如何控制用户态无法访问内核页面？
  - 答案：将内核页面的 pte 的‘U’标志位设置为 0。
  2. 相对于双页表，单页表有何优势？
  - 答案：在内核和用户态之间转换时不需要更换页表，也就不需要跳板，可以像之前一样直接切换上下文。
  3. 请描述：在单页表和双页表模式下，分别在哪个时机，如何切换页表？
  - 答案：双页表实现下用户程序和内核转换时、用户程序转换时都需要更换页表，而对于单页表操作系统，不同用户线程切换时需要更换页表。

## 5.10.2 实验练习

实验练习包括实践作业和问答作业两部分。

### 实践作业

#### 重写 sys\_get\_time

引入虚存机制后，原来内核的 sys\_get\_time 函数实现就无效了。请你重写这个函数，恢复其正常功能。

#### mmap 和 munmap 匿名映射

`mmap` 在 Linux 中主要用于在内存中映射文件，本次实验简化它的功能，仅用于申请内存。

请实现 mmap 和 munmap 系统调用，mmap 定义如下：

```
fn sys_mmap(start: usize, len: usize, prot: usize) -> isize
```

- syscall ID: 222
- 申请长度为 len 字节的物理内存（不要求实际物理内存位置，可以随便找一块），将其映射到 start 开始的虚存，内存页属性为 prot
- 参数：
  - start 需要映射的虚存起始地址，要求按页对齐
  - len 映射字节长度，可以为 0

- prot: 第 0 位表示是否可读, 第 1 位表示是否可写, 第 2 位表示是否可执行。其他位无效且必须为 0
- 返回值: 执行成功则返回 0, 错误返回 -1
- 说明:
  - 为了简单, 目标虚存区间要求按页对齐, len 可直接按页向上取整, 不考虑分配失败时的页回收。
- 可能的错误:
  - start 没有按页大小对齐
  - prot & !0x7 != 0 (prot 其余位必须为 0)
  - prot & 0x7 = 0 (这样的内存无意义)
  - [start, start + len) 中存在已经被映射的页
  - 物理内存不足

munmap 定义如下:

```
fn sys_munmap(start: usize, len: usize) -> isize
```

- syscall ID: 215
- 取消到 [start, start + len) 虚存的映射
- 参数和返回值请参考 mmap
- 说明:
  - 为了简单, 参数错误时不考虑内存的恢复和回收。
- 可能的错误:
  - [start, start + len) 中存在未被映射的虚存。

TIPS: 注意 prot 参数的语义, 它与内核定义的 MapPermission 有明显不同!

## 实验要求

- 实现分支: ch4-lab
- 实验目录要求不变
- 通过所有测例

在 os 目录下 make run TEST=1 测试 sys\_get\_time, make run TEST=2 测试 map 和 unmap。

challenge: 支持多核。

## 问答作业

无

## 实验练习的提交报告要求

- 简单总结本次实验与上个实验相比你增加的东西。(控制在 5 行以内, 不要贴代码)
- 完成问答问题。
- (optional) 你对本次实验设计及难度的看法。

## 第五章：进程

## 6.1 引言

## 6.1.1 本章导读

在正式开始这一章的介绍之前，我们可以看到：在前面的章节中基本涵盖了一个功能相对完善的操作系统内核所需的核心硬件机制：中断与异常、特权级、页表，而且一个一个逐步进化的远古生物操作系统内核让应用程序在开发和运行方面也越来越便捷和安全了。但开发者的需求是无穷的，开发者希望能够在计算机上有更多的动态交互和控制能力，比如在操作系统启动后，能灵活选择执行某个程序。但我们目前实现的这些操作系统还无法做到，这说明操作系统还缺少对应用程序动态执行的灵活性和支持！

到目前为止，操作系统启动后，能运行完它管理所有的应用程序。但在整个执行过程中，应用程序是被动地被操作系统加载运行，开发者与操作系统之间没有交互，开发者与应用程序之间没有交互，应用程序不能控制其它应用的执行。这使得开发者不能灵活地选择执行某个程序。为了方便开发者灵活执行程序，本章要完成的操作系统的核心目标是：**让开发者能够控制程序的运行**。

在前面的章节中，随着应用的需求逐渐变得复杂，作为其执行环境的操作系统内核也需要在硬件提供的相关机制的支持之下努力为应用提供更多强大、易用且安全的抽象。让我们先来简单回顾一下：

- 第一章《RV64 裸机应用》中，由于我们从始至终只需运行一个应用，这时我们的内核看起来只是一个**函数库**，它会对应用的执行环境进行初始化，使得应用能够正确接入计算机的启动流程，同时我们还设置好函数调用栈使得应用可以正常进行 Rust 函数调用。此外，它还将 SBI 接口函数进行了封装使得应用更容易使用这些功能。
- 第二章《批处理系统》中，我们需要自动加载并执行一个固定序列内的多个应用，当一个应用出错或者正常退出之后则切换到下一个。为了让这个流程能够稳定进行而不至于被某个应用的错误所破坏，内核需要借助硬件提供的**特权级机制**将应用代码放在 U 特权级执行，并对它的行为进行限制，从而实现了内核的安全核心机制—**控制隔离**。一旦应用出现错误或者请求一些只有内核才能提供的服务时，控制权会移交给内核并对该**Trap** 进行处理。
- 第三章《多道程序与分时多任务》中，出于对提高计算机系统总体性能的需求，操作系统在一个应用执行一段时间之后，会暂停这个应用并切换另外一个应用去执行，等到以后的某个时刻，操作系统再切换回之前的应用继续执行。这样就实现了内核的核心机制—**任务切换**。对于每个应用来说，它会认为自己始终独占一个 CPU，不过这只是内核对 CPU 资源的恰当抽象给它带来的一种幻象。

- 第四章《地址空间》中，我们利用硬件的分页机制，实现了内核的安全核心机制-**内存隔离**，建立了一种经典的抽象-**地址空间**，让应用程序在操作系统管控的内存空间中执行，代替了先前应用程序对于物理内存的直接访问方式。这样做使得每个应用独占一个访存空间并与其他应用隔离起来，这是由内核通过设定应用的页表来保证不同应用的数据（应用间的共享数据除外）所在在物理内存区域互不相交。于是开发者在开发应用的时候无需顾及其他应用，整个系统的安全性也得到了一定保证。

目前为止，所有的应用都是在内核初始化阶段被一并加载到内存中的，之后也无法对应用的执行进行动态增删。从一般用户的角度来看，第四章和第二章的批处理系统似乎并没有什么不同。事实上，由于我们还没有充分发掘这些硬件机制和抽象概念的能力，应用的开发和使用仍然比较受限，且用户在应用运行过程中的动态控制能力不够强。其实用户可以与操作系统之间可以建立一个交互界面，在应用程序的执行过程中，让用户可以通过这个界面主动给操作系统发出请求，来创建并执行新的应用程序，暂停或停止应用程序的执行等。

---

### 注解：UNIX shell 的起源

“shell”的名字和概念是从 UNIX 的前身 MULTICS 发展和继承过来的，应用程序可以通过 shell 程序来进行调用并被操作系统执行。Thompson shell 是历史上第一个 UNIX shell，在 1971 年由肯·汤普逊 (Ken Thompson) 写出了第一版并加入 UNIX 之中。Thompson shell 按照极简主义设计，语法非常简单，是一个简单的命令行解释器。它的许多特征影响了以后的操作系统命令行界面的发展。至 Version 7 Unix 之后，被 Bourne shell 取代。

---

---

### 注解：描述未来的 MULTICS 操作系统

在取得了 CTSS 操作系统的成功后，MIT 与 ARPA 在 1963 年 MAC 项目，其目标之一是设计和实现 CTSS 的后继操作系统。经过前期准备，在 1965 年，MIT 的 Fernando J. Corbató 教授联合贝尔实验室和通用电气公司联合启动了雄心勃勃的 MULTICS 操作系统项目。MULTICS 的目标是：改变人们使用计算机和计算机编程的方式，让人们能像使用电力或电话一样来方便地使用计算机的计算能力。类比于电力基础设施 (Electric utility)，MIT 的科学家想通过 MULTICS 构建未来的计算基础设施 (Computer utility)。

为此开发小组对 GE645 计算机系统和 MULTICS 操作系统提出了一系列的非常先进的设计思路。同学们如果阅读了 “Introduction and Overview of the Multics System”<sup>2</sup> 和 “Structure of the Multics Supervisor”<sup>3</sup> 这两篇论文，可以发现 MULTICS 操作系统的设计思路即使放到二十一世纪的今天也不算过时。但相对较弱和进展缓慢的硬件，用于编写操作系统的 PL/I 高级语言的编译器严重滞后，操作系统各种功能带来的大型软件复杂性导致了 MULTICS 操作系统的开发困难重重。不过最终在 1969 年，MULTICS 操作系统开始提供服务，并一直持续到 2000 年，算得上是很长寿了。

这里我们只讲述 MULTICS 操作系统中与进程 (Process) 相关的一些设计思路。MULTICS 操作系统中的进程是指一个程序/作业的执行过程，如编译一个程序、产生一个文件等。每个进程在执行过程中所占的内存空间范围由 GE645 计算机中处理器指定的段 (硬件机制) 来描述和限制。操作系统通过处理器调度算法和调度分派机制来让不同的进程分时使用处理器，这样进程会有正在运行的运行态、准备运行的就绪态和等待条件满足的阻塞态这样不同的执行状态。在进程管理方面，有动态创建进程、阻塞进程和终止进程等不同的操作。每个子进程都是从某个进程 (父进程) 通过系统调用产生出来的。子进程可以共享父进程拥有的内存空间。用户进程通过系统调用获得操作系统的服务，不能直接访问操作系统的数据和代码，确保了操作系统的安全。

### 成为未来基石的 UNIX 操作系统

Ken Thompson 和 Dennis Ritchie 这一对贝尔实验室的黄金搭档，在 1969 年退出 MULTICS 操作系统开发工作后，并没有放弃操作系统的研发，而是决定重新开始。Ken Thompson 从小处着手，从一台老旧的 DEC PDP-7 计算机开始，将 MULTICS 操作系统的设计想法进行简化，并一个一个地实现，完成了第一版 UNIX 操作系统内核，并带有汇编器、编辑器和 shell 应用程序。这时的操作系统只是一个简单的单任务操作系统。它的

<sup>2</sup> Fernando J. Corbató. “Introduction and overview of the MULTICS system” In Proc AFIPS I965 Fall Joint Computer Conf, Part I, Spartan Books, New York, 185-196.

<sup>3</sup>

V. A. Vyssotsky. “Structure of the Multics supervisor” In AFIPS Conf Proc 27 1965, Spartan Books Washington D C 1965 pp 203-212

UNIX 取名是对 MULTICS 的一种玩笑回应。Dennis Ritchie 具有 MULTICS 项目中的高级语言 PL/I 编译器方面的经验，他创建了小巧灵活的 C 语言和 C 编译器，UNIX 后续版本用 C 语言进行了重写。然后 C 语言和 UNIX 操作系统联手，影响了后续几乎所有的计算机和操作系统（Linux、MacOS、Windows…），成为了未来的基石。

这里我们关注 UNIX 操作系统中与进程（Process）相关的一些设计实现思路。简单地说，UNIX 操作系统中的进程实现充分吸取了 MULTICS 中关于进程的设计思想，实现了 `fork` `exec` `wait` `exit` 四个精巧的系统调用来支持对进程的灵活管理。父进程通过 `fork` 系统调用创建自身的副本（子进程）；称为“子进程”的副本可调用 `exec` 系统调用用另一个程序覆盖其内存空间，这样就可以执行新程序了；子进程执行完毕后，可通过调用 `exit` 系统调用退出并通知父进程；父进程通过调用 `wait` 系统调用等待子进程的退出。

一句话小结：MULTICS 操作系统的思想造就了 UNIX 操作系统，而 UNIX 操作系统引导了操作系统的发展历程，Linux 操作系统统治了当今世界。

于是，本章我们会开发一个用户 **终端**（Terminal）程序或称 **命令行应用**（Command Line Application，俗称 **shell**），形成用户与操作系统进行交互的命令行界面（Command Line Interface），它就和我们今天常用的 OS 中的命令行应用（如 Linux 中的 `bash`，Windows 中的 `CMD` 等）没有什么不同：只需在其中输入命令即可启动或杀死应用，或者监控系统的运行状况。这自然是现代 OS 中不可缺少的一部分，并大大增加了系统的 **可交互性**，使得用户可以更加灵活地控制系统。

为了在用户态就可以借助操作系统的服务动态灵活地管理和控制应用的执行，我们需要在已有的 **任务**抽象的基础上进一步扩展，形成新的抽象：**进程**，并实现若干基于 **进程**的强大系统调用。

- **创建**（Create）：父进程创建新的子进程。用户在 shell 中键入命令或用鼠标双击应用程序图标（这需要 GUI 界面，目前我们还没有实现）时，会调用操作系统服务来创建新进程，运行指定的程序。
- **销毁**（Destroy）：进程退出。进程会在运行完成后可自行退出，但还需要其他进程（如创建这些进程的父进程）来回收这些进程最后的资源，并销毁这些进程。
- **等待**（Wait）：父进程等待子进程退出。父进程等待子进程停止是很有用的，比如上面提到的收集子进程的退出信息，回收退出的子进程占用的剩余资源等。
- **信息**（Info）：获取进程的状态信息：操作系统也可提供有关进程的身份和状态等进程信息，例如进程的 ID，进程的运行状态，进程的优先级等。
- **其他**（Other）：其他的进程控制服务。例如，让一个进程能够杀死另外一个进程，暂停进程（停止运行一段时间），恢复进程（继续运行）等。

有了上述灵活强大的进程管理功能，就可以进化出本章的白垩纪“伤齿龙”<sup>1</sup> 操作系统了。

### 注解：任务和进程的关系与区别

第三章提到的 **任务** 和这里提到的 **进程** 有何关系和区别？这需要从二者对资源的占用和执行的过程这两个方面来进行分析。

- 相同点：站在一般用户和应用程序的角度看，任务和进程都表示运行的程序。站在操作系统的角度看，任务和进程都表示为一个程序的执行过程。二者都能够被操作系统打断并通过切换来分时占用 CPU 资源；都需要 **地址空间** 来放置代码和数据；都有从开始到结束运行这样的生命周期。
- 不同点：第三/四章提到的 **任务** 是这里提到的 **进程** 的初级阶段，任务还没进化到拥有更强大的动态变化功能：进程可以在运行的过程中，**创建子进程**、用新的 **程序内容** 覆盖已有的 **程序内容**。这种动态变化的功能可让程序在运行过程中动态使用更多的物理或虚拟的 **资源**。

<sup>1</sup> 伤齿龙是一种灵活的小型恐龙，生存于 7500 万年前的晚白垩纪，伤齿龙的脑袋与身体的比例是恐龙中最大之一，因此伤齿龙被认为是最有智能的恐龙之一。

## 6.1.2 实践体验

获取本章代码：

```
$ git clone https://github.com/rcore-os/rCore-Tutorial-v3.git
$ cd rCore-Tutorial-v3
$ git checkout ch5
```

在 qemu 模拟器上运行本章代码：

```
$ cd os
$ make run
```

待内核初始化完毕之后，将在屏幕上打印可用的应用列表并进入 shell 程序（以 K210 平台为例）：

```
[RustSBI output]
[kernel] Hello, world!
last 808 Physical Frames.
.text [0x80020000, 0x8002e000)
.rodata [0x8002e000, 0x80032000)
.data [0x80032000, 0x800c7000)
.bss [0x800c7000, 0x802d8000)
mapping .text section
mapping .rodata section
mapping .data section
mapping .bss section
mapping physical memory
remap_test passed!
after initproc!
/**** APPS ****
exit
fantastic_text
forktest
forktest2
forktest_simple
forktree
hello_world
initproc
matrix
sleep
sleep_simple
stack_overflow
user_shell
usertests
yield

Rust user shell
>>
```

其中 usertests 打包了很多应用，只要执行它就能够自动执行一系列应用。

只需输入应用的名称并回车即可在系统中执行该应用。如果输入错误的话可以使用退格键 (Backspace)。以应用 exit 为例：

```
>> exit
I am the parent. Forking the child...
I am the child.
I am parent, fork a child pid 3
```

(下页继续)

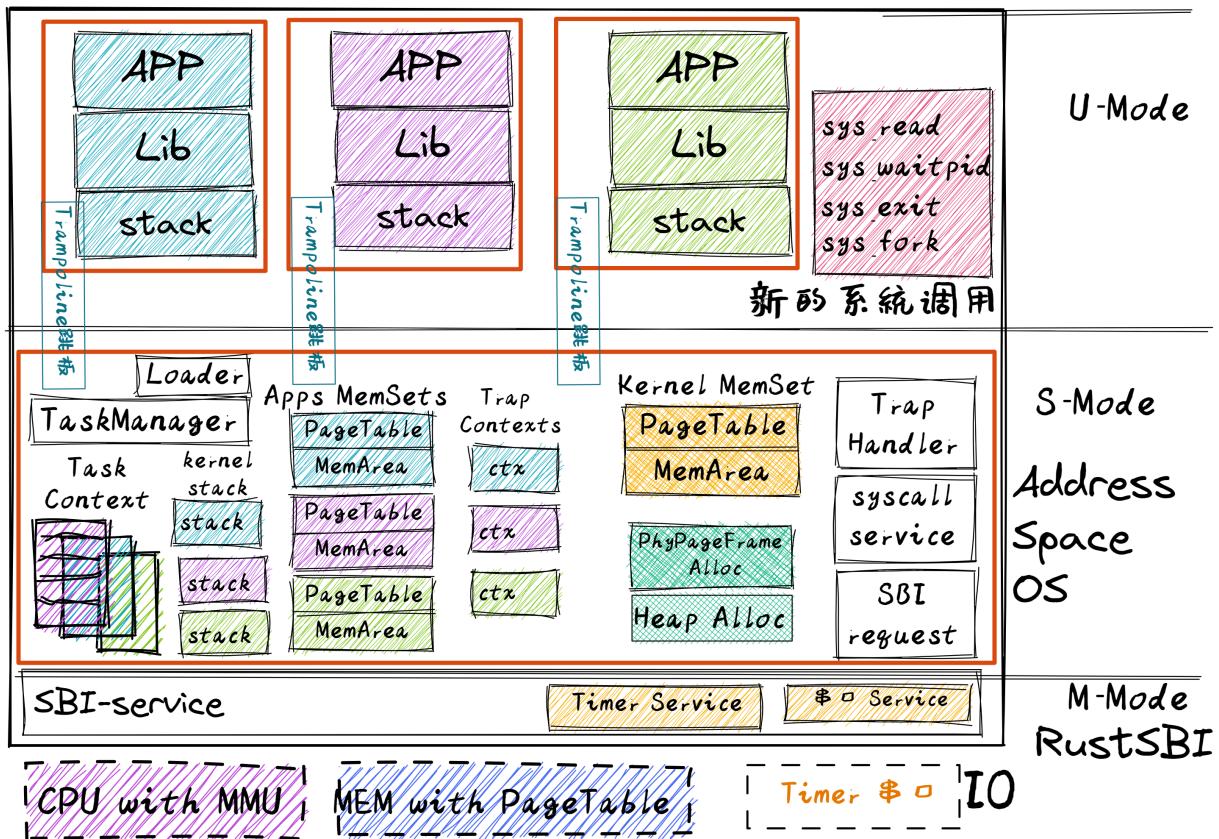
(续上页)

```
I am the parent, waiting now..
waitpid 3 ok.
exit pass.
Shell: Process 2 exited with code 0
>>
```

当应用执行完毕后，将继续回到 shell 程序的命令输入模式。

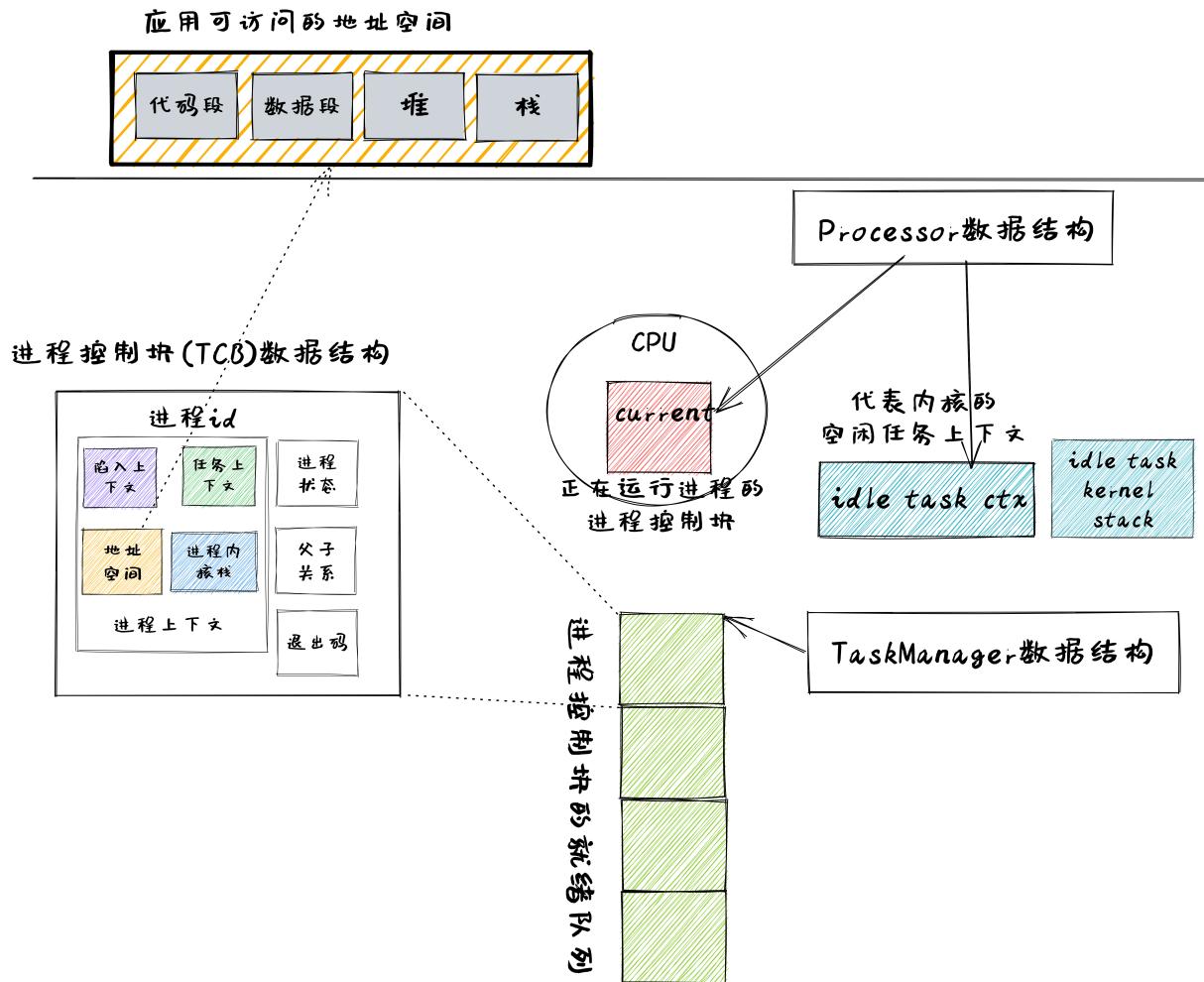
### 6.1.3 本章代码树

伤齿龙操作系统-ProcessOS 的总体结构如下图所示：



通过上图，大致可以看出伤齿龙操作系统-ProcessOS 在内部结构上没有特别大的改动，但把任务抽象进化成了进程抽象，其主要改动集中在进程管理的功能上，即通过提供新的系统调用服务：sys\_fork(创建子进程)、sys\_waitpid(等待子进程结束并回收子进程资源)、sys\_exec (用新的应用内容覆盖当前进程，即达到执行新应用的目的)。为了让用户能够输入命令或执行程序的名字，ProcessOS 还增加了一个 read 系统调用服务，这样用户通过操作系统的命令行接口-新添加的 shell 应用程序发出命令，来动态地执行各种新的应用，提高了用户与操作系统之间的交互能力。

而由于有了进程的新抽象，需要对已有任务控制块进行重构，ProcessOS 中与进程相关的核心数据结构如下图所示：



从上图可知，进程控制块 TaskControlBlock 包含与进程运行/切换/调度/地址空间相关的各种资源和信息。以前的任务管理器 TaskManager 分离为处理器管理结构 Processor 和新的 TaskManager。Processor 负责管理 CPU 上正在执行的任务和一些相关信息；而新的任务管理器 TaskManager 仅负责管理没在执行的所有任务，以及各种新的进程管理相关的系统调用服务。

位于 ch5 分支上的伤齿龙操作系统 - ProcessOS 的源代码如下所示：

```

1 ./os/src
2 Rust 28 Files 1848 Lines
3 Assembly 3 Files 86 Lines
4
5 |- bootloader
6 | |- rustsbi-qemu.bin
7 |- LICENSE
8 |- os
9 | |- build.rs (修改：基于应用名的应用构建器)
10 | |- Cargo.toml
11 | |- Makefile
12 | |- src
13 | |- config.rs
14 | |- console.rs
15 | |- entry.asm
16 | |- lang_items.rs

```

(下页继续)

(续上页)

```

17 link_app.S
18 linker-qemu.ld
19 loader.rs(修改: 基于应用名的应用加载器)
20 main.rs(修改)
21 mm(修改: 为了支持本章的系统调用对此模块做若干增强)
22 address.rs
23 frame_allocator.rs
24 heap_allocator.rs
25 memory_set.rs
26 mod.rs
27 page_table.rs
28 sbi.rs
29 sync
30 mod.rs
31 up.rs
32 syscall
33 fs.rs(修改: 新增 sys_read)
34 mod.rs(修改: 新的系统调用的分发处理)
35 process.rs (修改: 新增 sys_getpid/fork/exec/waitpid)
36 task
37 context.rs
38 manager.rs(新增: 任务管理器, 为上一章任务管理器功能的一部分)
39 mod.rs(修改: 调整原来的接口实现以支持进程)
40 pid.rs(新增: 进程标识符和内核栈的 Rust 抽象)
41 processor.rs(新增: 处理器管理结构 ``Processor``)
42 ↵, 为上一章任务管理器功能的一部分)
43 switch.rs
44 switch.S
45 task.rs(修改: 支持进程管理机制的任务控制块)
46 timer.rs
47 trap
48 context.rs
49 mod.rs(修改: 对于系统调用的实现进行修改以支持进程系统调用)
50 trap.S
51 README.md
52 rust-toolchain
53 user(对于用户库 user_lib 进行修改, 替换了一套新的测例)
54 Cargo.toml
55 Makefile
56 src
57 bin
58 exit.rs
59 fantastic_text.rs
60 forktest2.rs
61 forktest.rs
62 forktest_simple.rs
63 forktree.rs
64 hello_world.rs
65 initproc.rs
66 matrix.rs
67 sleep.rs
68 sleep_simple.rs
69 stack_overflow.rs
70 user_shell.rs
71 usertests.rs
72 yield.rs

```

(下页继续)

(续上页)

```

72 ├── console.rs
73 ├── lang_items.rs
74 ├── lib.rs
75 ├── linker.ld
76 └── syscall.rs

```

### 6.1.4 本章代码导读

本章的第一小节[进程概念及重要系统调用](#)介绍了操作系统中经典的进程概念，并描述我们将要实现的参考自 UNIX 系内核并经过简化的精简版进程模型。在该模型下，若想对进程进行管理，实现创建、退出等操作，核心就在于 `fork/exec/waitpid` 三个系统调用。

首先我们修改运行在应用态的应用软件，它们均放置在 `user` 目录下。在新增系统调用的时候，需要在 `user/src/lib.rs` 中新增一个 `sys_*` 的函数，它的作用是将对应的系统调用按照与内核约定的 ABI 在 `syscall` 中转化为一条用于触发系统调用的 `ecall` 的指令；还需要在用户库 `user/lib` 将 `sys_*` 进一步封装成一个应用可以直接调用的与系统调用同名的函数。通过这种方式我们新增三个进程模型中核心的系统调用 `fork/exec/waitpid`，一个查看进程 PID 的系统调用 `getpid`，还有一个允许应用程序获取用户键盘输入的 `read` 系统调用。

基于进程模型，我们在 `user/src/bin` 目录下重新实现了一组应用程序。其中有两个特殊的应用程序：用户初始程序 `initproc.rs` 和 shell 程序 `user_shell.rs`，可以认为它们位于内核和其他应用程序之间的中间层提供一些基础功能，但是它们仍处于用户态的应用层。前者会被内核唯一自动加载、也是最早加载并执行，后者则负责从键盘接收用户输入的应用名并执行对应的应用。剩下的应用从不同层面测试了我们内核实现的正确性，同学可以自行参考。值得一提的是，`usertests` 可以按照顺序执行绝大部分应用，会在测试操作系统功能和正确性上为我们提供很多方便。

接下来就需要在内核中实现简化版的进程管理机制并支持新增的系统调用。在本章第二小节[进程管理的核心数据结构](#)中我们对一些进程管理机制相关的数据结构进行了重构或者修改：

- 为了支持基于应用名而不是应用 ID 来查找应用 ELF 可执行文件，从而实现灵活的应用加载，在 `os/build.rs` 以及 `os/src/loader.rs` 中更新了 `link_app.s` 的格式使得它包含每个应用的名字，另外提供 `get_app_data_by_name` 接口获取应用的 ELF 数据。
- 在本章之前，任务管理器 `TaskManager` 不仅负责管理所有的任务状态，还维护着 CPU 当前正在执行的任务。这种设计耦合度较高，我们将后一个功能分离到 `os/src/task/processor.rs` 中的处理器管理结构 `Processor` 中，它负责管理 CPU 上执行的任务和一些其他信息；而 `os/src/task/manager.rs` 中的任务管理器 `TaskManager` 仅负责管理所有任务。
- 针对新的进程模型，我们复用前面章节的任务控制块 `TaskControlBlock` 作为进程控制块来保存进程的一些信息，相比前面章节还要新增 PID、内核栈、应用数据大小、父子进程、退出码等信息。它声明在 `os/src/task/task.rs` 中。
- 从本章开始，进程的 PID 将作为查找进程控制块的索引，这样就可以通过进程的 PID 来查找到进程的内核栈等各种进程相关信息。同时我们还面向进程控制块提供相应的资源自动回收机制。具体实现可以参考 `os/src/task/pid.rs`。

有了这些数据结构的支撑，我们在本章第三小节[进程管理机制的设计实现](#)实现进程管理机制。它可以分成如下几个方面：

- 初始进程的创建：在内核初始化的时候需要调用 `os/src/task/mod.rs` 中的 `add_initproc` 函数，它会调用 `TaskControlBlock::new` 读取并解析初始应用 `initproc` 的 ELF 文件数据并创建初始进程 `INITPROC`，随后会将它加入到全局任务管理器 `TASK_MANAGER` 中参与调度。
- 进程切换机制：当一个进程退出或者是主动/被动交出 CPU 使用权之后，需要由内核将 CPU 使用权交给其他进程。在本章中我们沿用 `os/src/task/mod.rs` 中的 `suspend_current_and_run_next` 和 `exit_current_and_run_next` 两个接口来实现进程切换功能，但是需要适当调整它们的实现。我们需要调用 `os/src/task/task.rs` 中的 `schedule` 函数进行进程切换，它会首先切换到处理器

的 idle 控制流（即 `os/src/task/processor` 的 `Processor::run` 方法），然后在里面选取要切换到的进程并切换过去。

- 进程调度机制：在进程切换的时候我们需要选取一个进程切换过去。选取进程逻辑可以参考 `os/src/task/manager.rs` 中的 `TaskManager::fetch_task` 方法。
- 进程生成机制：这主要是指 `fork/exec` 两个系统调用。它们的实现分别可以在 `os/src/syscall/process.rs` 中找到，分别基于 `os/src/process/task.rs` 中的 `TaskControlBlock::fork/exec`。
- 进程资源回收机制：当一个进程主动退出或出错退出的时候，在 `exit_current_and_run_next` 中会立即回收一部分资源并在进程控制块中保存退出码；而需要等到它的父进程通过 `waitpid` 系统调用（与 `fork/exec` 两个系统调用放在相同位置）捕获到它的退出码之后，它的进程控制块才会被回收，从而该进程的所有资源都被回收。
- 进程的 I/O 输入机制：为了支持用户终端 `user_shell` 读取用户键盘输入的功能，还需要实现 `read` 系统调用，它可以在 `os/src/syscall/fs.rs` 中找到。

## 6.2 进程概念及重要系统调用

### 6.2.1 本节导读

本节的内容有：

- 介绍进程的概念以及它和一些其他相近的概念的比较；
- 从应用开发者或是用户的角度介绍我们实现的一种简单类 Unix 进程模型；
- 介绍三个最重要的进程相关系统调用并给出一些用例。

### 6.2.2 进程概念

在本章的引言中，出于方便应用开发和使得应用功能更加强大的目标，我们引入了进程的概念。在本章之前，我们有 **任务** 的概念，即 **正在执行的程序**，主角是程序。而相比于 **任务**，**进程** (Process) 的含义是**在操作系统管理下的程序的一次执行过程**。这里的“程序的”成了形容词，而执行过程成为了主角，这充分体现了动态变化的执行特点。尽管这说起来很容易，但事实上进程是一个内涵相当丰富且深刻、难以从单个角度解释清楚的抽象概念。我们可以先试着从动态和静态的角度来进行初步的思考。我们知道，当一个应用的源程序被编译器成功构建之后，它会从源代码变为某种格式的可执行文件，如果将其展开，可以在它的内存布局中看到若干个功能迥异的逻辑段。但如果仅是这样，它也就只是某种格式特殊的、被**静态**归档到存储器上的一个文件而已。

然而，可执行文件与其他类型文件的根本性不同在于它可以被内核加载并执行。这一执行过程自然是不能凭空进行的，而是需要占据某些真实的硬件资源。例如，可执行文件一定需要被加载到物理内存的某些区域中才能执行，另外还可能需要预留一些可执行文件内存布局中未规划的区域（比如堆和栈），这就会消耗掉部分内存空间；在执行的时候需要占据一个 CPU 的全部硬件资源，我们之前介绍过的有通用寄存器（其中程序计数器 `pc` 和栈指针 `sp` 两个意义尤其重大）、`CSR`、各级 `cache`、`TLB` 等。

打一个比方，可执行文件本身可以看成一张编译器解析源代码之后总结出的一张记载如何利用各种硬件资源进行一轮生产流程的**蓝图**。而内核的一大功能便是作为一个硬件资源管理器，它可以随时启动一轮生产流程（即执行任意一个应用），这需要选中一张蓝图（此时确定执行哪个可执行文件），接下来就需要内核按照蓝图上所记载的对资源的需求来对应的将各类资源分配给它，让这轮生产流程得以顺利进行。当按照蓝图上的记载生产流程完成（应用退出）之后，内核还需要将对应的硬件资源回收以便后续的重复利用。

因此，进程就是操作系统选取某个可执行文件并对其进行一次动态执行的过程。相比可执行文件，它的动态性主要体现在：

1. 它是一个过程，从时间上来看有开始也有结束；
2. 在该过程中对于可执行文件中给出的需求要相对应 **硬件/虚拟资源** 进行 **动态绑定和解绑**。

这里需要指出的是，两个进程可以选择同一个可执行文件执行，然而它们却是截然不同的进程：它们的启动时间、占据的硬件资源、输入数据均有可能是不同的，这些条件均会导致它们是不一样的执行过程。在某些情况下，我们可以看到它们的输出是不同的——这是其中一种可能的直观表象。

在内核中，需要有一个进程管理器，在其中记录每个进程对资源的占用情况，这是内核作为一个硬件资源管理器所必须要做到的。进程管理器通常需要管理多个进程，因为如果同一时间只有一个进程的话，就可以简单的将所有的硬件资源都交给该进程，同时内核也会像第一章《RV64 裸机应用》那样退化成一个函数库。

本节接下来主要站在应用开发者和用户的角度来介绍如何理解进程概念并基于它编写应用程序。

---

#### 注解：为什么要在这里才引入进程

根据我们多年来的 OS 课程经验，学生对 **进程** 的简单定义“**正在执行的程序**”比较容易理解。但对于多个运行的程序之间如何切换，会带来哪些并发问题，进程创建与虚拟内存的关系等问题很难一下子理解清楚，也不清楚试图解决这些问题的原因。这主要是由于学生对进程的理解是站在应用程序角度来看的。

如果变化一下，让学生站在操作系统的角度来看，那么在进程这个定义背后，有特权级切换、异常处理，程序执行的上下文切换、地址映射、地址空间、虚存管理等一系列的知识支撑，才能理解清楚操作系统对进程的整个管理过程。所以，我们在前面几章对上述知识进行了铺垫。并以此为基础，更加全面地来分析操作系统是如何管理进程的。

---

#### 注解：进程，线程和协程

进程，线程和协程是操作系统中经常出现的名词，它们都是操作系统中的抽象概念，有联系和共同的地方，但也有区别。计算机的核心是 CPU，它承担了基本上所有的计算任务；而操作系统是计算机的管理者，它可以以进程，线程和协程为基本的管理和调度单位来使用 CPU 执行具体的程序逻辑。

从历史角度上看，它们依次出现的顺序是进程、线程和协程。在还没有进程抽象的早期操作系统中，计算机科学家把程序在计算机上的一次执行过程称为一个任务（Task）或一个工作（Job），其特点是任务和工作在其整个的执行过程中，不会被切换。这样其他任务必须等待一个任务结束后，才能执行，这样系统的效率会比较低。

在引入面向 CPU 的分时切换机制和面向内存的虚拟内存机制后，进程的概念就被提出了，进程成为 CPU（也称处理器）调度（Scheduling）和分派（Switch）的对象，各个进程间以时间片为单位轮流使用 CPU，且每个进程有各自独立的一块内存，使得各个进程之间内存地址相互隔离。这时，操作系统通过进程这个抽象来完成对应用程序在 CPU 和内存使用上的管理。

随着计算机的发展，对计算机系统性能的要求越来越高，而进程之间的切换开销相对较大，于是计算机科学家就提出了线程。线程是程序执行中一个单一的顺序控制流程，线程是进程的一部分，一个进程可以包含一个或多个线程。各个线程之间共享进程的地址空间，但线程要有自己独立的栈（用于函数访问，局部变量等）和独立的控制流。且线程是处理器调度和分派的基本单位。对于线程的调度和管理，可以在操作系统层面完成，也可以在用户态的线程库中完成。用户态线程也称为绿色线程（GreenThread）。如果是在用户态的线程库中完成，操作系统是“看不到”这样的线程的，也就谈不上对这样线程的管理了。

协程（Coroutines，也称纤程（Fiber）），也是程序执行中一个单一的顺序控制流程，建立在线程之上（即一个线程上可以有多个协程），但又是比线程更加轻量级的处理器调度对象。协程一般是由用户态的协程管理库来进行管理和调度，这样操作系统是看不到协程的。而且多个协程共享同一线程的栈，这样协程在时间和空间的管理开销上，相对于线程又有很大的改善。在具体实现上，协程可以在用户态运行时库这一层面通过函数调用来实现；也可在语言级支持协程，比如 Rust 借鉴自其他语言的的 `async`、`await` 关键字等，通过编译器和运行时库二者配合来简化程序员编程的负担并提高整体的性能。

### 6.2.3 进程模型与重要系统调用

目前，我们只介绍本章实现的内核中采用的一种非常简单的进程模型。这个进程模型有三个运行状态：就绪态、运行态和等待态；有基于独立页表的地址空间；可被操作系统调度来分时占用 CPU 执行；可以动态创建和退出；可通过系统调用获得操作系统的服务。

前面我们并没有给出进程需要使用哪些类型的资源，这其实取决于内核提供给应用的系统调用接口以及内核的具体实现。我们实现的进程模型建立在地址空间抽象之上：每个进程都需要一个地址空间，它涵盖了它选择的可执行文件的内存布局，还包含一些其他的逻辑段。且进程模型需要操作系统支持一些重要的系统调用：创建进程、执行新程序、等待进程结束等，来达到应用程序执行的动态灵活性。接下来会介绍这些系统调用的基本功能和设计思路。

#### fork 系统调用

系统中同一时间存在的每个进程都被一个不同的 **进程标识符 (PID, Process Identifier)** 所标识。在内核初始化完毕之后会创建一个进程——即 **用户初始进程 (Initial Process)**，它是目前在内核中以硬编码方式创建的唯一一个进程。其他所有的进程都是通过一个名为 `fork` 的系统调用创建的。

```
/// 功能：当前进程 fork 出来一个子进程。
/// 返回值：对于子进程返回 0，对于当前进程则返回子进程的 PID。
/// syscall ID: 220
pub fn sys_fork() -> isize;
```

进程 A 调用 `fork` 系统调用之后，内核会创建一个新进程 B，这个进程 B 和调用 `fork` 的进程 A 在它们分别返回用户态那一瞬间几乎处于相同的状态：这意味着它们包含的用户态的代码段、堆栈段及其他数据段的内容完全相同，但是它们是被放在两个独立的地址空间中的。因此新进程的地址空间需要从原有进程的地址空间完整拷贝一份。两个进程通用寄存器也几乎完全相同。例如，`pc` 相同意味着两个进程会从同一位置的一条相同指令（我们知道其上一条指令一定是用于系统调用的 `ecall` 指令）开始向下执行，`sp` 相同则意味着两个进程的用户栈在各自的地址空间中的位置相同。其余的寄存器相同则确保了二者回到了相同的控制流状态。

但是唯有用来保存 `fork` 系统调用返回值的 `a0` 寄存器（这是 RISC-V 64 的函数调用规范规定的函数返回值所用的寄存器）的值是不同的。这区分了两个进程：原进程的返回值为它新创建进程的 `PID`，而新创建进程的返回值为 0。由于新的进程是原进程主动调用 `fork` 衍生出来的，我们称新进程为原进程的 **子进程 (Child Process)**，相对的原进程则被称为新进程的 **父进程 (Parent Process)**。这样二者就建立了一种父子关系。注意到每个进程可能有多个子进程，但最多只能有一个父进程，于是所有进程可以被组织成一颗树，其根节点正是代表用户初始程序——`initproc`，也即第一个用户态的初始进程。

相比创建一个进程，`fork` 的另一个重要功能是建立一对新的父子关系。在我们的进程模型中，父进程和子进程之间的联系更为紧密，它们更容易进行合作或通信，而且一些重要的机制（如第七章会涉及的进程间通信机制）也需要在它们之间才能展开。

#### waitpid 系统调用

当一个进程通过 `exit` 系统调用退出之后，它所占用的资源并不能够立即全部回收。比如该进程的内核栈目前就正用来进行系统调用处理，如果将放置它的物理页帧回收的话，可能会导致系统调用不能正常处理。对于这种问题，一种典型的做法是当进程退出的时候内核立即回收一部分资源并将该进程标记为 **僵尸进程 (Zombie Process)**。之后，由该进程的父进程通过一个名为 `waitpid` 的系统调用收集该进程的返回状态并回收掉它所占据的全部资源，这样这个进程才被彻底销毁。系统调用 `waitpid` 的原型如下：

```
/// 功能：当前进程等待一个子进程变为僵尸进程，回收其全部资源并收集其返回值。
/// 参数：pid 表示要等待的子进程的进程 ID，如果为 -1 的话表示等待任意一个子进程；
/// exit_code 表示保存子进程返回值的地址，如果这个地址为 0 的话表示不必保存。
/// 返回值：如果要等待的子进程不存在则返回 -1；否则如果要等待的子进程均未结束则返回 -2；
```

(下页继续)

(续上页)

```
/// 否则返回结束的子进程的进程 ID。
/// syscall ID: 260
pub fn sys_waitpid(pid: isize, exit_code: *mut i32) -> isize;
```

一般情况下一个进程要负责通过 `waitpid` 系统调用来等待它 `fork` 出来的子进程结束并回收掉它们占据的资源，这也是父子进程间的一种同步手段。但这并不是必须的。如果一个进程先于它的子进程结束，在它退出的时候，它的所有子进程将成为进程树的根节点——用户初始进程的子进程，同时这些子进程的父进程也会转成用户初始进程。这之后，这些子进程的资源就由用户初始进程负责回收了，这也是用户初始进程很重要的一个用途。后面我们会介绍用户初始进程是如何实现的。

## exec 系统调用

如果仅有 `fork` 的话，那么所有的进程都只能和用户初始进程一样执行同样的代码段，这显然是远远不够的。于是我们还需要引入 `exec` 系统调用来执行不同的可执行文件：

```
///_
// 功能：将当前进程的地址空间清空并加载一个特定的可执行文件，返回用户态后开始它的执行。
// 参数：path 给出了要加载的可执行文件的名字；
// 返回值：如果出错的话（如找不到名字相符的可执行文件）则返回 -1，否则不应该返回。
// syscall ID: 221
pub fn sys_exec(path: &str) -> isize;
```

注意，我们知道 `path` 作为 `&str` 类型是一个胖指针，既有起始地址又包含长度信息。在实际进行系统调用的时候，我们只会将起始地址传给内核（对标 C 语言仅会传入一个 `char*`）。这就需要应用负责在传入的字符串的末尾加上一个 `\0`，这样内核才能知道字符串的长度。下面给出了用户库 `user_lib` 中的调用方式：

```
// user/src/exec.rs

pub fn sys_exec(path: &str) -> isize {
 syscall(SYSCALL_EXEC, [path.as_ptr() as usize, 0, 0])
}
```

这样，利用 `fork` 和 `exec` 的组合，我们很容易在一个进程内 `fork` 出一个子进程并执行一个特定的可执行文件。

### 注解：为何创建进程要通过两个系统调用而不是一个？

同学可能会有疑问，对于要达成执行不同应用的目标，我们为什么不设计一个系统调用接口同时实现创建一个新进程并加载给定的可执行文件两种功能？如果使用 `fork` 和 `exec` 的组合，那么 `fork` 出来的进程仅仅是为了 `exec` 一个新应用提供空间。而执行 `fork` 中对父进程的地址空间拷贝没有用处，还浪费了时间，且在后续清空地址空间的时候还会产生一些资源回收的额外开销。这样的设计来源于早期的 `MULTICS1` 和 `UNIX` 操作系统<sup>2</sup>，在当时是经过实践考验的，事实上 `fork` 和 `exec` 是一种灵活的系统调用组合，在当时内存空间比较小的情况下，可以支持更快的进程创建，且上述的开销能够通过一些结合虚存的技术方法（如 `Copy on write` 等）来缓解。而且拆分为两个系统调用后，可以灵活地支持 `重定向 (Redirection)` 等功能。上述方法是 `UNIX` 类操作系统的典型做法。

这一点与 `Windows` 操作系统不一样。在 `Windows` 中，`CreateProcess` 函数用来创建一个新的进程和它的主线程，通过这个新进程运行指定的可执行文件。虽然是一个函数，但这个函数的参数十个之多，使得这个函数很复杂，且没有 `fork` 和 `exec` 的组合的灵活性。而基于 `POSIX` 标准的 `posix_spawn` 系统调用则类似

<sup>1</sup> 1965 年，MIT、通用电气公司、贝尔实验室联合开发 `MULTICS` 操作系统，开发不够成功，但产生了很多新的设计思想，并催生了 `UNIX` 操作系统。

<sup>2</sup> 1969 年，贝尔实验室的 Ken Thompson 和 Dennis Ritchie 在退出 `MULTICS` 操作系统研发后，吸收其好的想法，设计实现了 `UNIX` 操作系统和 C 语言，并开始广泛推广。

Windows 的 CreateProcess 函数，不过对参数进行了简化，更适合现在的计算机系统（有更大的物理内存空间）和类 UNIX 应用程序（更加复杂的软件）。

## 6.2.4 应用程序示例

我们刚刚介绍了 fork/waitpid/exec 三个重要系统调用，我们可以借助它们和之前实现的系统调用开发出功能更为强大的应用程序。下面我们通过描述两个重要的应用程序：**用户初始程序-init** 和 **shell 程序-user\_shell** 的开发过程，来展示这些重要系统调用的使用方法。

### 系统调用封装

同学可以在 user/src/syscall.rs 中看到以 sys\_\* 开头的系统调用的函数原型，它们后续还会在 user/src/lib.rs 中被封装成方便应用程序使用的形式。如 sys\_fork 被封装成 fork，而 sys\_exec 被封装成 exec。这里值得一提的是 sys\_waitpid 被封装成两个不同的 API：

```

1 // user/src/lib.rs
2
3 pub fn wait(exit_code: &mut i32) -> isize {
4 loop {
5 match sys_waitpid(-1, exit_code as *mut _) {
6 -2 => { yield_(); }
7 // -1 or a real pid
8 exit_pid => return exit_pid,
9 }
10 }
11 }
12
13 pub fn waitpid(pid: usize, exit_code: &mut i32) -> isize {
14 loop {
15 match sys_waitpid(pid as isize, exit_code as *mut _) {
16 -2 => { yield_(); }
17 // -1 or a real pid
18 exit_pid => return exit_pid,
19 }
20 }
21 }
```

其中 wait 表示等待任意一个子进程结束，根据 sys\_waitpid 的约定它需要传的 pid 参数为 -1；而 waitpid 则等待一个进程标识符的值为 pid 的子进程结束。在具体实现方面，我们看到当 sys\_waitpid 返回值为 -2，即要等待的子进程存在但它却尚未退出的时候，我们调用 yield\_ 主动交出 CPU 使用权，待下次 CPU 使用权被内核交还给它的时候再次调用 sys\_waitpid 查看要等待的子进程是否退出。这样做可以减小 CPU 资源的浪费。

目前的实现风格是尽可能简化内核，因此 sys\_waitpid 是立即返回的，即它的返回值只能给出返回这一时刻的状态。如果这一时刻要等待的子进程还尚未结束，那么也只能如实向应用报告这一结果。于是用户库 user/src/lib.rs 就需要负责对返回状态进行持续的监控，因此它里面便需要进行循环检查。在后续的实现中，我们会将 sys\_waitpid 的内核实现设计为 **阻塞的**，即直到得到一个确切的结果之前，其对应的进程暂停（不再继续执行）在内核内；如果 sys\_waitpid 需要的值能够得到，则它对应的进程会被内核唤醒继续执行，且内核返回给应用的结果可以直接使用。那时 wait 和 waitpid 两个 API 的实现便会更加简单。

## 用户初始程序 initproc

我们首先来看用户初始程序 initproc 是如何实现的：

```

1 // user/src/bin/initproc.rs
2
3 #![no_std]
4 #![no_main]
5
6 #[macro_use]
7 extern crate user_lib;
8
9 use user_lib::{
10 fork,
11 wait,
12 exec,
13 yield_,
14 };
15
16 #[no_mangle]
17 fn main() -> i32 {
18 if fork() == 0 {
19 exec("user_shell\0");
20 } else {
21 loop {
22 let mut exit_code: i32 = 0;
23 let pid = wait(&mut exit_code);
24 if pid == -1 {
25 yield_();
26 continue;
27 }
28 println!(
29 "[initproc] Released a zombie process, pid={}, exit_code={}",
30 pid,
31 exit_code,
32);
33 }
34 }
35 0
36 }

```

- 第 19 行为 `fork` 返回值为 0 的分支，表示子进程，此行直接通过 `exec` 执行 shell 程序 `user_shell`，注意我们需要在字符串末尾手动加入 `\0`，因为 Rust 在将这些字符串连接到只读数据段的时候不会插入 `\0`。
- 第 21 行开始则为返回值不为 0 的分支，表示调用 `fork` 的用户初始程序 `initproc` 自身。可以看到它在不断循环调用 `wait` 来等待那些被移交到它下面的子进程并回收它们占据的资源。如果回收成功的话则会打印一条报告信息给出被回收子进程的 `pid` 值和返回值；否则就 `yield_` 交出 CPU 资源并在下次轮到它执行的时候再回收看看。这也可以看出，用户初始程序 `initproc` 对于资源的回收并不算及时，但是对于已经退出的僵尸进程，用户初始程序 `initproc` 最终总能够成功回收它们的资源。

## shell 程序 user\_shell

由于 shell 程序 user\_shell 需要捕获我们的输入并进行解析处理，我们需要加入一个新的用于输入的系统调用：

```
/// 功能：从文件中读取一段内容到缓冲区。
/// 参数：fd 是待读取文件的文件描述符，切片 buffer 则给出缓冲区。
/// 返回值：如果出现了错误则返回 -1，否则返回实际读到的字节数。
/// syscall ID: 63
pub fn sys_read(fd: usize, buffer: &mut [u8]) -> isize;
```

在实际调用的时候我们必须要同时向内核提供缓冲区的起始地址及长度：

```
// user/src/syscall.rs

pub fn sys_read(fd: usize, buffer: &mut [u8]) -> isize {
 syscall(SYSCALL_READ, [fd, buffer.as_mut_ptr() as usize, buffer.len()])
}
```

我们在用户库中将其进一步封装成每次能够从 **标准输入** 中获取一个字符的 `getchar` 函数：

```
// user/src/lib.rs

pub fn read(fd: usize, buf: &mut [u8]) -> isize { sys_read(fd, buf) }

// user/src/console.rs

const STDIN: usize = 0;

pub fn getchar() -> u8 {
 let mut c = [0u8; 1];
 read(STDIN, &mut c);
 c[0]
}
```

其中，我们每次临时声明一个长度为 1 的缓冲区。

接下来就可以介绍 shell 程序 user\_shell 是如何实现的了：

```
1 // user/src/bin/user_shell.rs
2
3 #![no_std]
4 #![no_main]
5
6 extern crate alloc;
7
8 #[macro_use]
9 extern crate user_lib;
10
11 const LF: u8 = 0x0au8;
12 const CR: u8 = 0x0du8;
13 const DL: u8 = 0x7fu8;
14 const BS: u8 = 0x08u8;
15
16 use alloc::string::String;
17 use user_lib::{fork, exec, waitpid, yield_};
18 use user_lib::console::getchar;
19
20 #[no_mangle]
```

(下页继续)

(续上页)

```

21 pub fn main() -> i32 {
22 println!("Rust user shell");
23 let mut line: String = String::new();
24 print!(">> ");
25 loop {
26 let c = getchar();
27 match c {
28 LF | CR => {
29 println!("");
30 if !line.is_empty() {
31 line.push('\0');
32 let pid = fork();
33 if pid == 0 {
34 // child process
35 if exec(line.as_str()) == -1 {
36 println!("Error when executing!");
37 return -4;
38 }
39 unreachable!();
40 } else {
41 let mut exit_code: i32 = 0;
42 let exit_pid = waitpid(pid as usize, &mut exit_code);
43 assert_eq!(pid, exit_pid);
44 println!(
45 "Shell: Process {} exited with code {}",
46 pid, exit_code
47);
48 }
49 line.clear();
50 }
51 print!(">> ");
52 }
53 BS | DL => {
54 if !line.is_empty() {
55 print!("{} ", BS as char);
56 print!(" ");
57 print!("{} ", BS as char);
58 line.pop();
59 }
60 }
61 _ => {
62 print!("{} ", c as char);
63 line.push(c as char);
64 }
65 }
66 }
67 }

```

可以看到，在以第 25 行开头的主循环中，每次都是调用 `getchar` 获取一个用户输入的字符，并根据它相应进行一些动作。第 23 行声明的字符串 `line` 则维护着用户当前输入的命令内容，它也在不断发生变化。

### 注解：在应用中使能动态内存分配

我们知道，在 Rust 中可变长字符串类型 `String` 是基于动态内存分配的。因此本章我们还要在用户库 `user_lib` 中支持动态内存分配，与第四章的做法相同，只需加入以下内容即可：

```

use buddy_system_allocator::LockedHeap;

const USER_HEAP_SIZE: usize = 16384;

static mut HEAP_SPACE: [u8; USER_HEAP_SIZE] = [0; USER_HEAP_SIZE];

#[global_allocator]
static HEAP: LockedHeap = LockedHeap::empty();

#[alloc_error_handler]
pub fn handle_alloc_error(layout: core::alloc::Layout) -> ! {
 panic!("Heap allocation error, layout = {:?}", layout);
}

#[no_mangle]
#[link_section = ".text.entry"]
pub extern "C" fn _start() -> ! {
 unsafe {
 HEAP.lock()
 .init(HEAP_SPACE.as_ptr() as usize, USER_HEAP_SIZE);
 }
 exit(main());
}

```

- 如果用户输入回车键（第 28 行），那么 `user_shell` 会 `fork` 出一个子进程（第 34 行开始）并试图通过 `exec` 系统调用执行一个应用，应用的名字在字符串 `line` 中给出。这里我们需要注意的是，由于子进程是从 `user_shell` 进程中 `fork` 出来的，它们除了 `fork` 的返回值不同之外均相同，自然也可以看到一个和 `user_shell` 进程维护的版本相同的字符串 `line`。第 35 行对 `exec` 的返回值进行了判断，如果返回值为 -1 则说明在应用管理器中找不到名字相同的应用，此时子进程就直接打印错误信息并退出；反之 `exec` 则根本不会返回，而是开始执行目标应用。

`fork` 之后的 `user_shell` 进程自己的逻辑可以在第 41 行找到。可以看出它只是在等待 `fork` 出来的子进程结束并回收掉它的资源，还会顺带收集子进程的退出状态并打印出来。

- 如果用户输入退格键（第 53 行），首先我们需要将屏幕上当前行的最后一个字符用空格替换掉，这可以通过输入一个特殊的退格字节 `BS` 来实现。其次，`user_shell` 进程内维护的 `line` 也需要弹出最后一个字符。
- 如果用户输入了一个其他字符（第 61 行），它将会被视为用户的正常输入，我们直接将它打印在屏幕上并加入到 `line` 中。

当内核初始化完毕之后，它会从可执行文件 `initproc` 中加载并执行用户初始程序 `initproc`，而用户初始程序 `initproc` 中又会 `fork` 并 `exec` 来运行 shell 程序 `user_shell`。这两个应用虽然都是在 CPU 的 U 特权级执行的，但是相比其他应用，它们要更加底层和基础。原则上应该将它们作为一个组件打包在操作系统中。但这里为了实现更加简单，我们并不将它们和其他应用进行区分。

除此之外，我们还从 `μCore`<sup>3</sup> 中借鉴了很多应用测例。它们可以做到同一时间 并发多个进程并能够有效检验我们内核实现的正确性。感兴趣的同学可以参考 `matrix` 和 `forktree` 等应用。

<sup>3</sup> `uCore OS` 是用于清华大学计算机系本科操作系统课程的 OS 教学试验内容。`uCore OS` 起源于 MIT CSAIL PDOS 课题组开发的 `xv6&jos`、哈佛大学开发的 `OS161` 教学操作系统、以及 `Linux-2.4` 内核。目前 `rCore/uCore Tutorial OS` 逐步代替 `uCore OS` 成为新的教学 OS。

## 6.3 进程管理的核心数据结构

### 6.3.1 本节导读

为了更好实现进程管理，同时也使得操作系统整体架构更加灵活，能够满足后续的一些需求，我们需要重新设计一些数据结构包含的内容及接口。本节将按照如下顺序来进行介绍：

- 基于应用名的应用链接：在编译阶段的链接过程中，生成包含多个应用和应用位置信息的 `link_app.S` 文件。
- 基于应用名的加载器：根据应用名字来加载应用的 ELF 文件中代码段和数据段到内存中，为创建一个新进程做准备。
- 进程标识符 `PidHandle` 以及内核栈 `KernelStack`：进程控制块的重要组成部分。
- 任务控制块 `TaskControlBlock`：表示进程的核心数据结构。
- 任务管理器 `TaskManager`：管理进程集合的核心数据结构。
- 处理器管理结构 `Processor`：用于进程调度，维护进程的处理器状态。

### 6.3.2 应用的链接与加载支持

#### 基于应用名的应用链接

在实现 `exec` 系统调用的时候，我们需要根据应用的名字而不仅仅是一个编号来获取应用的 ELF 格式数据。因此原有的链接和加载接口需要做出如下变更：

在 Rust 编译 & 链接辅助程序 `os/build.rs` 中，会读取位于 `user/src/bin` 中应用程序对应的执行文件，并生成 `link_app.S`，按顺序保存链接进来的每个应用的名字：

```

1 // os/build.rs
2
3 for i in 0..apps.len() {
4 writeln!(f, r#".quad app_{i}_start"#, i)?;
5 }
6 writeln!(f, r#".quad app_{apps.len() - 1}_end"#, apps.len() - 1)?;
7
8 writeln!(f, r#"
9 .global _app_names
10 _app_names:#)?;
11 for app in apps.iter() {
12 writeln!(f, r#".string \"{}\"#, app)?;
13 }
14
15 for (idx, app) in apps.iter().enumerate() {
16 ...
17 }
```

第 8~13 行，我们按照顺序将各个应用的名字通过 `.string` 伪指令放到数据段中，注意链接器会自动在每个字符串的结尾加入分隔符 `\0`，它们的位置则由全局符号 `_app_names` 指出。这样在编译操作系统的过程中，会生成如下的 `link_app.S` 文件：

```

1 .section .data
2 .global _num_app
3 _num_app:
```

(下页继续)

(续上页)

```

4 .quad 15
5 .quad app_0_start
6 .quad app_1_start
7
8 .global _app_names
9 _app_names:
10 .string "exit"
11 .string "fantastic_text"
12
13 .section .data
14 .global app_0_start
15 .global app_0_end
16 .align 3
17 app_0_start:
18 .incbin ".../user/target/riscv64gc-unknown-none-elf/release/exit"
19 app_0_end:
20

```

在这个文件中，可以看到应用代码和表示应用的元数据信息都放在数据段。第 10 行是第一个应用的名字 exit，第 13~14 行是第一个应用 exit 在 OS 镜像文件中的开始和结束位置；第 18 行是第一个应用 exit 的 ELF 格式执行文件的内容，

### 基于应用名的应用加载器

而在加载器 `loader.rs` 中，我们会分析 `link_app.s` 中的内容，并用一个全局可见的只读向量 `APP_NAMES` 来按照顺序将所有应用的名字保存在内存中：

```

// os/src/loader.rs

lazy_static! {
 static ref APP_NAMES: Vec<&'static str> = {
 let num_app = get_num_app();
 extern "C" { fn _app_names(); }
 let mut start = _app_names as usize as *const u8;
 let mut v = Vec::new();
 unsafe {
 for _ in 0..num_app {
 let mut end = start;
 while end.read_volatile() != '\0' as u8 {
 end = end.add(1);
 }
 let slice = core::slice::from_raw_parts(start, end as usize - start
 ↵as usize);
 let str = core::str::from_utf8(slice).unwrap();
 v.push(str);
 start = end.add(1);
 }
 }
 v
 };
}

```

使用 `get_app_data_by_name` 可以按照应用的名字来查找获得应用的 ELF 数据，而 `list_apps` 在内核初始化时被调用，它可以打印出所有可用的应用的名字。

```
// os/src/loader.rs

pub fn get_app_data_by_name(name: &str) -> Option<&'static [u8]> {
 let num_app = get_num_app();
 (0..num_app)
 .find(|&i| APP_NAMES[i] == name)
 .map(|i| get_app_data(i))
}

pub fn list_apps() {
 println!("/***** APPS *****");
 for app in APP_NAMES.iter() {
 println!("{}", app);
 }
 println!("*****");
}
```

这样，操作系统就可以读取并加载某个应用的执行文件到内存中了，这就为通过 `exec` 系统调用创建新进程做好了前期准备。

### 6.3.3 进程标识符和内核栈

#### 进程标识符

同一时间存在的所有进程都有一个唯一的进程标识符，它们是互不相同的整数，这样才能表示表示进程的唯一性。这里我们使用 RAII 的思想，将其抽象为一个 `PidHandle` 类型，当它的生命周期结束后对应的整数会被编译器自动回收：

```
// os/src/task/pid.rs

pub struct PidHandle(pub usize);
```

类似之前的物理页帧分配器 `FrameAllocator`，我们实现一个同样使用简单栈式分配策略的进程标识符分配器 `PidAllocator`，并将其全局实例化为 `PID_ALLOCATOR`：

```
// os/src/task/pid.rs

struct PidAllocator {
 current: usize,
 recycled: Vec<usize>,
}

impl PidAllocator {
 pub fn new() -> Self {
 PidAllocator {
 current: 0,
 recycled: Vec::new(),
 }
 }
 pub fn alloc(&mut self) -> PidHandle {
 if let Some(pid) = self.recycled.pop() {
 PidHandle(pid)
 } else {
 self.current += 1;
 PidHandle(self.current - 1)
 }
 }
}
```

(下页继续)

(续上页)

```

 }
}

pub fn deallocate(&mut self, pid: usize) {
 assert!(pid < self.current);
 assert!(
 self.recycled.iter().find(|ppid| **ppid == pid).is_none(),
 "pid {} has been deallocated!", pid
);
 self.recycled.push(pid);
}

lazy_static! {
 static ref PID_ALLOCATOR : UPSafeCell<PidAllocator> = unsafe {
 UPSafeCell::new(PidAllocator::new())
 };
}

```

PidAllocator::alloc 将会分配出去一个将 usize 包装之后的 PidHandle。我们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：

```

// os/src/task/pid.rs

pub fn pid_alloc() -> PidHandle {
 PID_ALLOCATOR.exclusive_access().alloc()
}

```

同时我们也需要为 PidHandle 实现 Drop Trait 来允许编译器进行自动的资源回收：

```

// os/src/task/pid.rs

impl Drop for PidHandle {
 fn drop(&mut self) {
 PID_ALLOCATOR.exclusive_access().deallocate(self.0);
 }
}

```

## 内核栈

在前面的章节中我们介绍过内核地址空间布局，当时我们将每个应用的内核栈按照应用编号从小到大的顺序将它们作为逻辑段从高地址到低地址放在内核地址空间中，且两两之间保留一个守护页面使得我们能够尽可能早的发现内核栈溢出问题。从本章开始，我们将应用编号替换为进程标识符。我们可以在内核栈 KernelStack 中保存着它所属进程的 PID：

```

// os/src/task/pid.rs

pub struct KernelStack {
 pid: usize,
}

```

它提供以下方法：

```

1 // os/src/task/pid.rs
2

```

(下页继续)

(续上页)

```

3 /// Return (bottom, top) of a kernel stack in kernel space.
4 pub fn kernel_stack_position(app_id: usize) -> (usize, usize) {
5 let top = TRAMPOLINE - app_id * (KERNEL_STACK_SIZE + PAGE_SIZE);
6 let bottom = top - KERNEL_STACK_SIZE;
7 (bottom, top)
8 }
9
10 impl KernelStack {
11 pub fn new(pid_handle: &PidHandle) -> Self {
12 let pid = pid_handle.0;
13 let (kernel_stack_bottom, kernel_stack_top) = kernel_stack_position(pid);
14 KERNEL_SPACE
15 .exclusive_access()
16 .insert_framed_area(
17 kernel_stack_bottom.into(),
18 kernel_stack_top.into(),
19 MapPermission::R | MapPermission::W,
20);
21 KernelStack {
22 pid: pid_handle.0,
23 }
24 }
25 pub fn push_on_top<T>(&self, value: T) -> *mut T where
26 T: Sized, {
27 let kernel_stack_top = self.get_top();
28 let ptr_mut = (kernel_stack_top - core::mem::size_of::<T>()) as *mut T;
29 unsafe { *ptr_mut = value; }
30 ptr_mut
31 }
32 pub fn get_top(&self) -> usize {
33 let (_, kernel_stack_top) = kernel_stack_position(self.pid);
34 kernel_stack_top
35 }
36 }

```

- 第 11 行, new 方法可以从一个 PidHandle , 也就是一个已分配的进程标识符中对应生成一个内核栈 KernelStack 。它调用了第 4 行声明的 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, 随即在第 14 行将一个逻辑段插入内核地址空间 KERNEL\_SPACE 中。
- 第 25 行的 push\_on\_top 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第 32 行的 get\_top 方法来获取当前内核栈顶在内核地址空间中的地址。

内核栈 KernelStack 也用到了 RAII 的思想, 具体来说, 实际保存它的物理页帧的生命周期与它绑定在一起, 当 KernelStack 生命周期结束后, 这些物理页帧也将被编译器自动回收:

```

// os/src/task/pid.rs

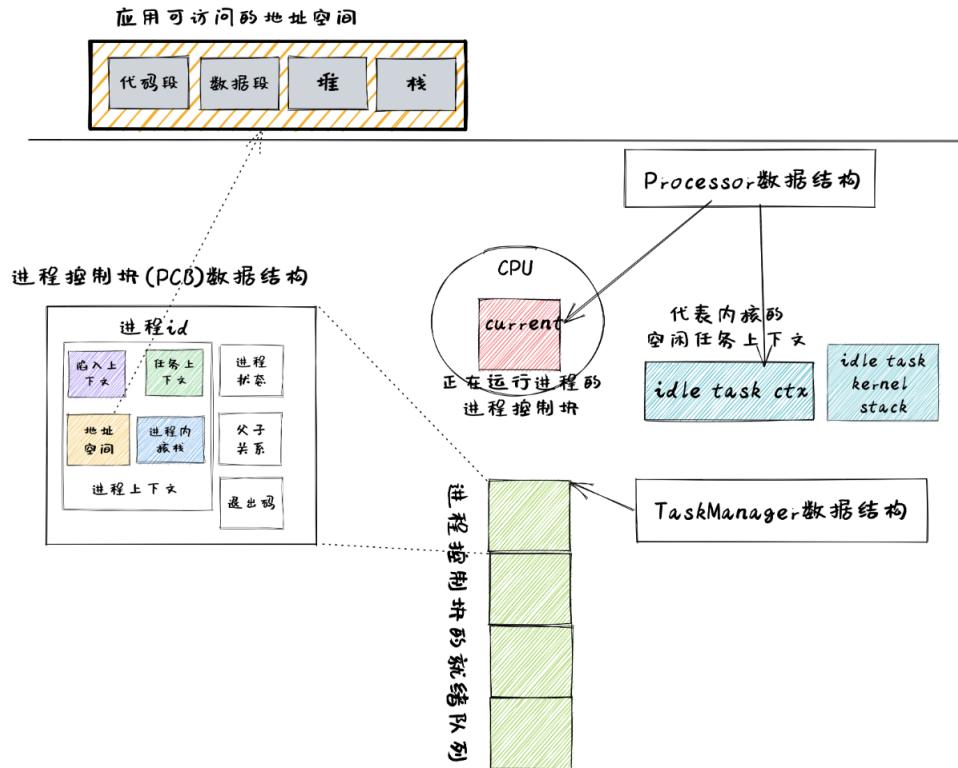
impl Drop for KernelStack {
 fn drop(&mut self) {
 let (kernel_stack_bottom, _) = kernel_stack_position(self.pid);
 let kernel_stack_bottom_va: VirtAddr = kernel_stack_bottom.into();
 KERNEL_SPACE
 .exclusive_access()
 .remove_area_with_start_vpn(kernel_stack_bottom_va.into());
 }
}

```

这仅需要为 `KernelStack` 实现 `Drop Trait`, 一旦它的生命周期结束则在内核地址空间中将对应的逻辑段删除 (为此在 `MemorySet` 中新增了一个名为 `remove_area_with_start_vpn` 的方法, 感兴趣的同学可以参考其实现), 由前面章节的介绍我们知道这也就意味着那些物理页帧被同时回收掉了。

### 6.3.4 进程控制块

在内核中, 每个进程的执行状态、资源控制等元数据均保存在一个被称为 **进程控制块** (PCB, Process Control Block) 的结构中, 它是内核对进程进行管理的单位, 故而是一种极其关键的内核数据结构。在内核看来, 它就等价于一个进程。



承接前面的章节, 我们仅需对任务控制块 `TaskControlBlock` 进行若干改动并让它直接承担进程控制块的功能:

```

1 // os/src/task/task.rs
2
3 pub struct TaskControlBlock {
4 // immutable
5 pub pid: PidHandle,
6 pub kernel_stack: KernelStack,
7 // mutable
8 inner: UPSafeCell<TaskControlBlockInner>,
9 }
10
11 pub struct TaskControlBlockInner {
12 pub trap_cx_ppn: PhysPageNum,
13 pub base_size: usize,
14 pub task_cx: TaskContext,
15 pub task_status: TaskStatus,
16 }
```

(下页继续)

(续上页)

```

16 pub memory_set: MemorySet,
17 pub parent: Option<Weak<TaskControlBlock>>,
18 pub children: Vec<Arc<TaskControlBlock>>,
19 pub exit_code: i32,
20 }
```

任务控制块中包含两部分：

- 在初始化之后就不再变化的元数据：直接放在任务控制块中。这里将进程标识符 `PidHandle` 和内核栈 `KernelStack` 放在其中；
- 在运行过程中可能发生变化的元数据：则放在 `TaskControlBlockInner` 中，将它再包裹上一层 `UPSafeCell<T>` 放在任务控制块中。这是因为在我们的设计中外层只能获取任务控制块的不可变引用，若想修改里面的部分内容的话这需要 `UPSafeCell<T>` 所提供的内部可变性。

`TaskControlBlockInner` 中则包含下面这些内容：

- `trap_cx_ppn` 指出了应用地址空间中的 Trap 上下文（详见第四章）被放在的物理页帧的物理页号。
- `base_size` 的含义是：应用数据仅有可能出现在应用地址空间低于 `base_size` 字节的区域中。借助它我们可以清楚的知道应用有多少数据驻留在内存中。
- `task_cx` 将暂停的任务的任务上下文保存在任务控制块中。
- `task_status` 维护当前进程的执行状态。
- `memory_set` 表示应用地址空间。
- `parent` 指向当前进程的父进程（如果存在的话）。注意我们使用 `Weak` 而非 `Arc` 来包裹另一个任务控制块，因此这个智能指针将不会影响父进程的引用计数。
- `children` 则将当前进程的所有子进程的任务控制块以 `Arc` 智能指针的形式保存在一个向量中，这样才能够更方便的找到它们。
- 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。

注意我们在维护父子进程关系的时候大量用到了引用计数 `Arc/Weak`。进程控制块的本体是被放到内核堆上面的，对于它的一切访问都是通过智能指针 `Arc/Weak` 来进行的，这样是便于建立父子进程的双向链接关系（避免仅基于 `Arc` 形成环状链接关系）。当且仅当智能指针 `Arc` 的引用计数变为 0 的时候，进程控制块以及被绑定到它上面的各类资源才会被回收。子进程的进程控制块并不会被直接放到父进程控制块中，因为子进程完全有可能在父进程退出后仍然存在。

`TaskControlBlockInner` 提供的方法主要是对于它内部的字段的快捷访问：

```

// os/src/task/task.rs

impl TaskControlBlockInner {
 pub fn get_trap_cx(&self) -> &static mut TrapContext {
 self.trap_cx_ppn.get_mut()
 }
 pub fn get_user_token(&self) -> usize {
 self.memory_set.token()
 }
 fn get_status(&self) -> TaskStatus {
 self.task_status
 }
 pub fn is_zombie(&self) -> bool {
 self.get_status() == TaskStatus::Zombie
 }
}
```

(下页继续)

(续上页)

```

 }
}

```

而任务控制块 TaskControlBlock 目前提供以下方法：

```

// os/src/task/task.rs

impl TaskControlBlock {
 pub fn inner_exclusive_access(&self) -> RefMut<'_, TaskControlBlockInner> {
 self.inner.exclusive_access()
 }
 pub fn getpid(&self) -> usize {
 self.pid.0
 }
 pub fn new(elf_data: &[u8]) -> Self {...}
 pub fn exec(&self, elf_data: &[u8]) {...}
 pub fn fork(self: &Arc<TaskControlBlock>) -> Arc<TaskControlBlock> {...}
}

```

- inner\_exclusive\_access 通过 UPSafeCell< T >.exclusive\_access() 来得到一个 RefMut< '\_ , TaskControlBlockInner >，它可以被看成一个内层 TaskControlBlockInner 的可变引用并可以对它指向的内容进行修改。
- getpid 以 usize 的形式返回当前进程的进程标识符。
- new 用来创建一个新的进程，目前仅用于内核中手动创建唯一一个初始进程 initproc。
- exec 用来实现 exec 系统调用，即当前进程加载并执行另一个 ELF 格式可执行文件。
- fork 用来实现 fork 系统调用，即当前进程 fork 出来一个与之几乎相同的子进程。

new/exec/fork 的实现我们将在下一小节再介绍。

### 6.3.5 任务管理器

在前面的章节中，任务管理器 TaskManager 不仅负责管理所有的任务，还维护着 CPU 当前在执行哪个任务。由于这种设计不够灵活，不能拓展到后续的多核环境，我们需要将任务管理器对于 CPU 的监控职能拆分到下面即将介绍的处理器管理结构 Processor 中去，任务管理器自身仅负责管理所有任务。在这里，任务指的就是进程。

```

1 // os/src/task/manager.rs
2
3 pub struct TaskManager {
4 ready_queue: VecDeque<Arc<TaskControlBlock>>,
5 }
6
7 /// A simple FIFO scheduler.
8 impl TaskManager {
9 pub fn new() -> Self {
10 Self { ready_queue: VecDeque::new(), }
11 }
12 pub fn add(&mut self, task: Arc<TaskControlBlock>) {
13 self.ready_queue.push_back(task);
14 }
15 pub fn fetch(&mut self) -> Option<Arc<TaskControlBlock>> {
16 self.ready_queue.pop_front()

```

(下页继续)

(续上页)

```

17 }
18 }
19
20 lazy_static! {
21 pub static ref TASK_MANAGER: UPSafeCell<TaskManager> = unsafe {
22 UPSafeCell::new(TaskManager::new())
23 };
24 }
25
26 pub fn add_task(task: Arc<TaskControlBlock>) {
27 TASK_MANAGER.exclusive_access().add(task);
28 }
29
30 pub fn fetch_task() -> Option<Arc<TaskControlBlock>> {
31 TASK_MANAGER.exclusive_access().fetch()
32 }

```

TaskManager 将所有的任务控制块用引用计数 Arc 智能指针包裹后放在一个双端队列 VecDeque 中。正如之前介绍的那样，我们并不直接将任务控制块放到 TaskManager 里面，而是将它们放在内核堆上，在任务管理器中仅存放他们的引用计数智能指针，这也是任务管理器的操作单位。这样做的原因在于，任务控制块经常需要被放入/取出，如果直接移动任务控制块自身将会带来大量的数据拷贝开销，而对于智能指针进行移动则没有多少开销。其次，允许任务控制块的共享引用在某些情况下能够让我们的实现更加方便。

TaskManager 提供 add/fetch 两个操作，前者表示将一个任务加入队尾，后者则表示从队头中取出一个任务来执行。从调度算法来看，这里用到的就是最简单的 RR 算法。全局实例 TASK\_MANAGER 则提供给内核的其他子模块 add\_task/fetch\_task 两个函数。

### 6.3.6 处理器管理结构

处理器管理结构 Processor 负责从任务管理器 TaskManager 中分出去的维护 CPU 状态的职责：

```

// os/src/task/processor.rs

pub struct Processor {
 current: Option<Arc<TaskControlBlock>>,
 idle_task_cx: TaskContext,
}

impl Processor {
 pub fn new() -> Self {
 Self {
 current: None,
 idle_task_cx: TaskContext::zero_init(),
 }
 }
}

```

在 Processor 中存放所有在运行过程中可能变化的内容，目前包括：

- 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。

Processor 是描述 CPU 执行状态的数据结构。在单核 CPU 环境下，我们仅创建单个 Processor 的全局实例 PROCESSOR：

```
// os/src/task/processor.rs

lazy_static! {
 pub static ref PROCESSOR: UPSafeCell<Processor> = unsafe {
 UPSafeCell::new(Processor::new())
 };
}
```

## 正在执行的任务

在抢占式调度模型中，在一个处理器上执行的任务常常被换入或换出，因此我们需要维护在一个处理器上正在执行的任务，可以查看它的信息或是对它进行替换：

```
// os/src/task/processor.rs

impl Processor {
 pub fn take_current(&mut self) -> Option<Arc<TaskControlBlock>> {
 self.current.take()
 }
 pub fn current(&self) -> Option<Arc<TaskControlBlock>> {
 self.current.as_ref().map(|task| Arc::clone(task))
 }
}

pub fn take_current_task() -> Option<Arc<TaskControlBlock>> {
 PROCESSOR.exclusive_access().take_current()
}

pub fn current_task() -> Option<Arc<TaskControlBlock>> {
 PROCESSOR.exclusive_access().current()
}

pub fn current_user_token() -> usize {
 let task = current_task().unwrap();
 let token = task.inner_exclusive_access().get_user_token();
 token
}

pub fn current_trap_cx() -> &'static mut TrapContext {
 current_task().unwrap().inner_exclusive_access().get_trap_cx()
}
```

- 第 4 行的 `Processor::take_current` 可以取出当前正在执行的任务。
- 第 7 行的 `Processor::current` 返回当前执行的任务的一份拷贝。
- 第 12 行的 `take_current_task` 以及第 16 行的 `current_task` 是对 `Processor::take_current/current` 进行封装并提供给内核其他子模块的接口。
- 第 20 行的 `current_user_token` 和第 26 行的 `current_trap_cx` 基于 `current_task` 实现，可以提供当前正在执行的任务的更多信息。

## 任务调度的 idle 控制流

Processor 有一个不同的 idle 控制流，它运行在这个 CPU 核的启动栈上，功能是尝试从任务管理器中选出一个任务来在当前 CPU 核上执行。在内核初始化完毕之后，会通过调用 run\_tasks 函数来进入 idle 控制流：

```

1 // os/src/task/processor.rs
2
3 pub fn run_tasks() {
4 loop {
5 let mut processor = PROCESSOR.exclusive_access();
6 if let Some(task) = fetch_task() {
7 let idle_task_cx_ptr = processor.get_idle_task_cx_ptr();
8 // access coming task TCB exclusively
9 let mut task_inner = task.inner_exclusive_access();
10 let next_task_cx_ptr = &task_inner.task_cx as *const TaskContext;
11 task_inner.task_status = TaskStatus::Running;
12 // stop exclusively accessing coming task TCB manually
13 drop(task_inner);
14 processor.current = Some(task);
15 // stop exclusively accessing processor manually
16 drop(processor);
17 unsafe {
18 __switch(
19 idle_task_cx_ptr,
20 next_task_cx_ptr,
21);
22 }
23 }
24 }
25 }
26
27 impl Processor {
28 fn get_idle_task_cx_ptr(&mut self) -> *mut TaskContext {
29 &mut self.idle_task_cx as *mut _
30 }
31 }
```

可以看到，调度功能的主体是 run\_tasks()。它循环调用 fetch\_task 直到顺利从任务管理器中取出一个任务，随后便准备通过任务切换的方式来执行：

- 第 7 行得到 \_\_switch 的第一个参数，也就是当前 idle 控制流的 task\_cx\_ptr，这调用了第 25 行的 Processor.get\_idle\_task\_cx\_ptr 方法。
- 第 9~11 行需要先获取从任务管理器中取出对应的任务控制块，并获取任务块内部的 next\_task\_cx\_ptr 作为 \_\_switch 的第二个参数，然后修改任务的状态为 Running。
- 第 13 行需要手动回收对即将执行任务的任务控制块的借用标记，使得后续我们仍可以访问该任务控制块。这里我们不能依赖编译器在 if let 块结尾时的自动回收，因为中间我们会在自动回收之前调用 \_\_switch，这将导致我们在实际上已经结束访问却没有进行回收的情况下切换到下一个任务，最终可能违反 UPSafeCell 的借用约定而使得内核报错退出。同理在第 16 行我们手动回收 PROCESSOR 的借用标记。
- 第 14 行我们修改当前 Processor 正在执行的任务为我们取出的任务。注意这里相当于 Arc<TaskControlBlock> 形式的任务从任务管理器流动到了处理器管理结构中。也就是说，在稳定的情况下，每个尚未结束的进程的任务控制块都只能被引用一次，要么在任务管理器中，要么则是在代表 CPU 处理器的 Processor 中。
- 第 18 行我们调用 \_\_switch 来从当前的 idle 控制流切换到接下来要执行的任务。

上面介绍了从 idle 控制流通过任务调度切换到某个任务开始执行的过程。而反过来，当一个应用用尽了内核本轮分配给它的时间片或者它主动调用 `yield` 系统调用交出 CPU 使用权之后，内核会调用 `schedule` 函数来切换到 idle 控制流并开启新一轮的任务调度。

```
// os/src/task/processor.rs

pub fn schedule(switted_task_cx_ptr: *mut TaskContext) {
 let mut processor = PROCESSOR.exclusive_access();
 let idle_task_cx_ptr = processor.get_idle_task_cx_ptr();
 drop(processor);
 unsafe {
 __switch(
 switted_task_cx_ptr,
 idle_task_cx_ptr,
);
 }
}
```

这里，我们需要传入即将被切换出去的任务的 `task_cx_ptr` 来在合适的位置保存任务上下文，之后就可以通过 `__switch` 来切换到 idle 控制流。从源代码来看，切换回去之后，内核将跳转到 `run_tasks` 中 `__switch` 返回之后的位置，也即开启了下一轮的调度循环。

## 6.4 进程管理机制的设计实现

### 6.4.1 本节导读

有了上节的数据结构和相关基本方法的介绍后，我们还需完成进程管理关键功能的实现，从而构造出一个完整的白垩纪“伤齿龙”操作系统。本节将从如下四个方面介绍如何基于上一节设计的内核数据结构来实现进程管理：

- 创建初始进程：创建第一个用户态进程 `initproc`；
- 进程调度机制：当进程主动调用 `sys_yield` 交出 CPU 使用权或者内核把本轮分配的时间片用尽的进程换出且换入下一个进程；
- 进程生成机制：介绍进程相关的两个重要系统调用 `sys_fork`/`sys_exec` 的实现；
- 进程资源回收机制：当进程调用 `sys_exit` 正常退出或者出错被内核终止之后如何保存其退出码，其父进程通过 `sys_waitpid` 系统调用收集该进程的信息并回收其资源。
- 字符输入机制：为了支持 `shell` 程序 `user_shell` 获得字符输入，介绍 `sys_read` 系统调用的实现；

### 6.4.2 初始进程的创建

内核初始化完毕之后即会调用 `task` 子模块提供的 `add_initproc` 函数来将初始进程 `initproc` 加入任务管理器，但在这之前我们需要初始化初始进程的进程控制块 `INITPROC`，这个过程基于 `lazy_static` 在运行时完成。

```
// os/src/task/mod.rs

use crate::loader::get_app_data_by_name;
use manager::add_task;

lazy_static! {
```

(下页继续)

(续上页)

```

1 pub static ref INITPROC: Arc<TaskControlBlock> = Arc::new(
2 TaskControlBlock::new(get_app_data_by_name("initproc").unwrap())
3);
4
5
6 pub fn add_initproc() {
7 add_task(INITPROC.clone());
8 }

```

我们调用 `TaskControlBlock::new` 来创建一个进程控制块，它需要传入 ELF 可执行文件的数据切片作为参数，这可以通过加载器 `loader` 子模块提供的 `get_app_data_by_name` 接口查找 `initproc` 的 ELF 执行文件数据来获得。在初始化 `INITPROC` 之后，就可以在 `add_initproc` 中调用 `task` 的任务管理器 `manager` 子模块提供的 `add_task` 接口，将其加入到任务管理器。

接下来介绍 `TaskControlBlock::new` 是如何实现的：

```

1 // os/src/task/task.rs
2
3 use super::{PidHandle, pid_alloc, KernelStack};
4 use super::TaskContext;
5 use crate::config::TRAP_CONTEXT;
6 use crate::trap::TrapContext;
7
8 // impl TaskControlBlock
9 pub fn new(elf_data: &[u8]) -> Self {
10 // memory_set with elf program headers/trampoline/trap context/user stack
11 let (memory_set, user_sp, entry_point) = MemorySet::from_elf(elf_data);
12 let trap_cx_ppn = memory_set
13 .translate(VirtAddr::from(TRAP_CONTEXT).into())
14 .unwrap()
15 .ppn();
16 // alloc a pid and a kernel stack in kernel space
17 let pid_handle = pid_alloc();
18 let kernel_stack = KernelStack::new(&pid_handle);
19 let kernel_stack_top = kernel_stack.get_top();
20 // push a task context which goes to trap_return to the top of kernel stack
21 let task_control_block = Self {
22 pid: pid_handle,
23 kernel_stack,
24 inner: unsafe { UPSafeCell::new(TaskControlBlockInner {
25 trap_cx_ppn,
26 base_size: user_sp,
27 task_cx: TaskContext::goto_trap_return(kernel_stack_top),
28 task_status: TaskStatus::Ready,
29 memory_set,
30 parent: None,
31 children: Vec::new(),
32 exit_code: 0,
33 }) },
34 };
35 // prepare TrapContext in user space
36 let trap_cx = task_control_block.inner_exclusive_access().get_trap_cx();
37 *trap_cx = TrapContext::app_init_context(
38 entry_point,
39 user_sp,
40 KERNEL_SPACE.exclusive_access().token(),
41 kernel_stack_top,

```

(下页继续)

(续上页)

```

42 trap_handler as usize,
43);
44 task_control_block
45 }
```

- 第 11 行我们解析应用的 ELF 执行文件得到应用地址空间 `memory_set`，用户栈在应用地址空间中的位置 `user_sp` 以及应用的入口点 `entry_point`。
- 第 12 行我们手动查页表找到位于应用地址空间中新创建的 Trap 上下文被实际放在哪个物理页帧上，用来做后续的初始化。
- 第 16~19 行我们为该进程分配 PID 以及内核栈，并记录下内核栈在内核地址空间的位置 `kernel_stack_top`。
- 第 20 行我们在该进程的内核栈上压入初始化的任务上下文，使得第一次任务切换到它的时候可以跳转到 `trap_return` 并进入用户态开始执行。
- 第 21 行我们整合之前的部分信息创建进程控制块 `task_control_block`。
- 第 37 行我们初始化位于该进程应用地址空间中的 Trap 上下文，使得第一次进入用户态的时候时候能正确跳转到应用入口点并设置好用户栈，同时也保证在 Trap 的时候用户态能正确进入内核态。
- 第 44 行将 `task_control_block` 返回。

### 6.4.3 进程调度机制

通过调用 task 子模块提供的 `suspend_current_and_run_next` 函数可以暂停当前任务并切换到下一个任务，当应用调用 `sys_yield` 主动交出使用权、本轮时间片用尽或者由于某些原因内核中的处理无法继续的时候，就会在内核中调用此函数触发调度机制并进行任务切换。下面给出了两种典型的使用情况：

```

1 // os/src/syscall/process.rs
2
3 pub fn sys_yield() -> isize {
4 suspend_current_and_run_next();
5 0
6 }
7
8 // os/src/trap/mod.rs
9
10 #[no_mangle]
11 pub fn trap_handler() -> ! {
12 set_kernel_trap_entry();
13 let scause = scause::read();
14 let stval = stval::read();
15 match scause.cause() {
16 Trap::Interrupt(Interrupt::SupervisorTimer) => {
17 set_next_trigger();
18 suspend_current_and_run_next();
19 }
20 ...
21 }
22 trap_return();
23 }
```

随着进程概念的引入，`suspend_current_and_run_next` 的实现也需要发生变化：

```

1 // os/src/task/mod.rs
2
3 use processor::{task_current_task, schedule};
4 use manager::add_task;
5
6 pub fn suspend_current_and_run_next() {
7 // There must be an application running.
8 let task = take_current_task().unwrap();
9
10 // ---- access current TCB exclusively
11 let mut task_inner = task.inner_exclusive_access();
12 let task_cx_ptr = &mut task_inner.task_cx as *mut TaskContext;
13 // Change status to Ready
14 task_inner.task_status = TaskStatus::Ready;
15 drop(task_inner);
16 // ---- stop exclusively accessing current PCB
17
18 // push back to ready queue.
19 add_task(task);
20 // jump to scheduling cycle
21 schedule(task_cx_ptr);
22}

```

首先通过 `take_current_task` 来取出当前正在执行的任务，修改其进程控制块内的状态，随后将这个任务放入任务管理器的队尾。接着调用 `schedule` 函数来触发调度并切换任务。注意，当仅有一个任务的时候，`suspend_current_and_run_next` 的效果是会继续执行这个任务。

#### 6.4.4 进程的生成机制

在内核中手动生成的进程只有初始进程 `initproc`，余下所有的进程都是它直接或间接 `fork` 出来的。当一个子进程被 `fork` 出来之后，它可以调用 `exec` 系统调用来加载并执行另一个可执行文件。因此，`fork/exec` 两个系统调用提供了进程的生成机制。下面我们分别来介绍二者的实现。

#### fork 系统调用的实现

在实现 `fork` 的时候，最为关键且困难的是为子进程创建一个和父进程几乎完全相同的应用地址空间。我们的实现如下：

```

1 // os/src/mm/memory_set.rs
2
3 impl MapArea {
4 pub fn from_another(another: &MapArea) -> Self {
5 Self {
6 vpn_range: VPNRange::new(
7 another.vpn_range.get_start(),
8 another.vpn_range.get_end()
9),
10 data_frames: BTreeMap::new(),
11 map_type: another.map_type,
12 map_perm: another.map_perm,
13 }
14 }
15 }
16

```

(下页继续)

(续上页)

```

17 impl MemorySet {
18 pub fn from_existed_user(user_space: &MemorySet) -> MemorySet {
19 let mut memory_set = Self::new_bare();
20 // map trampoline
21 memory_set.map_trampoline();
22 // copy data sections/trap_context/user_stack
23 for area in user_space.areas.iter() {
24 let new_area = MapArea::from_another(area);
25 memory_set.push(new_area, None);
26 // copy data from another space
27 for vpn in area.vpn_range {
28 let src_ppn = user_space.translate(vpn).unwrap().ppn();
29 let dst_ppn = memory_set.translate(vpn).unwrap().ppn();
30 dst_ppn.get_bytes_array().copy_from_slice(src_ppn.get_bytes_array());
31 }
32 }
33 memory_set
34 }
35 }

```

这需要对内存管理子模块 mm 做一些拓展：

- 第 4 行的 `MapArea::from_another` 可以从一个逻辑段复制得到一个虚拟地址区间、映射方式和权限控制均相同的逻辑段，不同的是由于它还没有真正被映射到物理页帧上，所以 `data_frames` 字段为空。
- 第 18 行的 `MemorySet::from_existed_user` 可以复制一个完全相同的地址空间。首先在第 19 行，我们通过 `new_bare` 新创建一个空的地址空间，并在第 21 行通过 `map_trampoline` 为这个地址空间映射上跳板页面，这是因为我们解析 ELF 创建地址空间的时候，并没有将跳板页作为一个单独的逻辑段插入到地址空间的逻辑段向量 `areas` 中，所以这里需要单独映射上。

剩下的逻辑段都包含在 `areas` 中。我们遍历原地址空间中的所有逻辑段，将复制之后的逻辑段插入新的地址空间，在插入的时候就已经实际分配了物理页帧了。接着我们遍历逻辑段中的每个虚拟页面，对应完成数据复制，这只需要找出两个地址空间中的虚拟页面各被映射到哪个物理页帧，就可转化为将数据从物理内存中的一个位置复制到另一个位置，使用 `copy_from_slice` 即可轻松实现。

接着，我们实现 `TaskControlBlock::fork` 来从父进程的进程控制块创建一份子进程的控制块：

```

1 // os/src/task/task.rs
2
3 impl TaskControlBlock {
4 pub fn fork(self: &Arc<TaskControlBlock>) -> Arc<TaskControlBlock> {
5 // ---- access parent PCB exclusively
6 let mut parent_inner = self.inner_exclusive_access();
7 // copy user space(include trap context)
8 let memory_set = MemorySet::from_existed_user(
9 &parent_inner.memory_set
10);
11 let trap_cx_ppn = memory_set
12 .translate(VirtAddr::from(TRAP_CONTEXT).into())
13 .unwrap()
14 .ppn();
15 // alloc a pid and a kernel stack in kernel space
16 let pid_handle = pid_alloc();
17 let kernel_stack = KernelStack::new(&pid_handle);
18 let kernel_stack_top = kernel_stack.get_top();
19 let task_control_block = Arc::new(TaskControlBlock {

```

(下页继续)

(续上页)

```

20 pid: pid_handle,
21 kernel_stack,
22 inner: unsafe { UPSafeCell::new(TaskControlBlockInner {
23 trap_cx_ppn,
24 base_size: parent_inner.base_size,
25 task_cx: TaskContext::goto_trap_return(kernel_stack_top),
26 task_status: TaskStatus::Ready,
27 memory_set,
28 parent: Some(Arc::downgrade(self)),
29 children: Vec::new(),
30 exit_code: 0,
31 }) },
32 });
33 // add child
34 parent_inner.children.push(task_control_block.clone());
35 // modify kernel_sp in trap_cx
36 // **** access children PCB exclusively
37 let trap_cx = task_control_block.inner_exclusive_access().get_trap_cx();
38 trap_cx.kernel_sp = kernel_stack_top;
39 // return
40 task_control_block
41 // ---- stop exclusively accessing parent/children PCB automatically
42 }
43 }

```

它基本上和新建进程控制块的 TaskControlBlock::new 是相同的，但要注意以下几点：

- 子进程的地址空间不是通过解析 ELF 文件，而是通过在第 8 行调用 MemorySet::from\_existed\_user 复制父进程地址空间得到的；
- 第 24 行，我们让子进程和父进程的 base\_size，也即应用数据的大小保持一致；
- 在 fork 的时候需要注意父子进程关系的维护。第 28 行我们将父进程的弱引用计数放到子进程的进程控制块中，而在第 33 行我们将子进程插入到父进程的孩子向量 children 中。

我们在子进程内核栈上压入一个初始化的任务上下文，使得内核一旦通过任务切换到该进程，就会跳转到 trap\_return 来进入用户态。而在复制地址空间的时候，子进程的 Trap 上下文也是完全从父进程复制过来的，这可以保证子进程进入用户态和其父进程回到用户态的那一瞬间 CPU 的状态是完全相同的（后面我们会让它们的返回值不同从而区分两个进程）。而两个进程的应用数据由于地址空间复制的原因也是完全相同的，这是 fork 语义要求做到的。

在具体实现 sys\_fork 的时候，我们需要特别注意如何体现父子进程的差异：

```

1 // os/src/syscall/process.rs
2
3 pub fn sys_fork() -> isize {
4 let current_task = current_task().unwrap();
5 let new_task = current_task.fork();
6 let new_pid = new_task.pid.0;
7 // modify trap context of new_task, because it returns immediately after switching
8 let trap_cx = new_task.inner_exclusive_access().get_trap_cx();
9 // we do not have to move to next instruction since we have done it before
10 // for child process, fork returns 0
11 trap_cx.x[10] = 0; // x[10] is a0 reg
12 // add new task to scheduler
13 add_task(new_task);
14 new_pid as isize
15 }

```

(下页继续)

(续上页)

```

16 // os/src/trap/mod.rs
17
18 #[no_mangle]
19 pub fn trap_handler() -> ! {
20 set_kernel_trap_entry();
21 let scause = scause::read();
22 let stval = stval::read();
23 match scause.cause() {
24 Trap::Exception(Exception::UserEnvCall) => {
25 // jump to next instruction anyway
26 let mut cx = current_trap_cx();
27 cx.sepc += 4;
28 // get system call return value
29 let result = syscall(cx.x[17], [cx.x[10], cx.x[11], cx.x[12]]);
30 // cx is changed during sys_exec, so we have to call it again
31 cx = current_trap_cx();
32 cx.x[10] = result as usize;
33 }
34 }
35 ...
36 }

```

在调用 `syscall` 进行系统调用分发并具体调用 `sys_fork` 之前，第 28 行，`trap_handler` 已经将当前进程 Trap 上下文中的 `sepc` 向后移动了 4 字节，使得它回到用户态之后，会从发出系统调用的 `ecall` 指令的下一条指令开始执行。之后当我们复制地址空间的时候，子进程地址空间 Trap 上下文的 `sepc` 也是移动之后的值，我们无需再进行修改。

父子进程回到用户态的瞬间都处于刚刚从一次系统调用返回的状态，但二者的返回值不同。第 8~11 行我们将子进程的 Trap 上下文中用来存放系统调用返回值的 `a0` 寄存器修改为 0；第 33 行，而父进程系统调用的返回值会在 `trap_handler` 中 `syscall` 返回之后再设置为 `sys_fork` 的返回值，这里我们返回子进程的 PID。这就做到了父进程 `fork` 的返回值为子进程的 PID，而子进程的返回值则为 0。通过返回值是否为 0 可以区分父子进程。

另外，不要忘记在第 13 行，我们将生成的子进程通过 `add_task` 加入到任务管理器中。

## exec 系统调用的实现

`exec` 系统调用使得一个进程能够加载一个新应用的 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。我们先从进程控制块的层面进行修改：

```

1 // os/src/task/task.rs
2
3 impl TaskControlBlock {
4 pub fn exec(&self, elf_data: &[u8]) {
5 // memory_set with elf program headers/trampoline/trap context/user stack
6 let (memory_set, user_sp, entry_point) = MemorySet::from_elf(elf_data);
7 let trap_cx_ppn = memory_set
8 .translate(VirtAddr::from(TRAP_CONTEXT).into())
9 .unwrap()
10 .ppn();
11
12 // **** access inner exclusively
13 let mut inner = self.inner_exclusive_access();
14 // substitute memory_set
15 inner.memory_set = memory_set;

```

(下页继续)

(续上页)

```

16 // update trap_cx_ppn
17 inner.trap_cx_ppn = trap_cx_ppn;
18 // initialize trap_cx
19 let trap_cx = inner.get_trap_cx();
20 *trap_cx = TrapContext::app_init_context(
21 entry_point,
22 user_sp,
23 KERNEL_SPACE.exclusive_access().token(),
24 self.kernel_stack.get_top(),
25 trap_handler as usize,
26);
27 // **** stop exclusively accessing inner automatically
28 }
29 }
```

它在解析传入的 ELF 格式数据之后只做了两件事情：

- 首先是从 ELF 文件生成一个全新的地址空间并直接替换进来（第 15 行），这将导致原有的地址空间生命周期结束，里面包含的全部物理页帧都会被回收；
- 然后是修改新的地址空间中的 Trap 上下文，将解析得到的应用入口点、用户栈位置以及一些内核的信息进行初始化，这样才能正常实现 Trap 机制。

这里无需对任务上下文进行处理，因为这个进程本身已经在执行了，而只有被暂停的应用才需要在内核栈上保留一个任务上下文。

有了 exec 函数后，sys\_exec 就很容易实现了：

```

1 // os/src/mm/page_table.rs
2
3 pub fn translated_str(token: usize, ptr: *const u8) -> String {
4 let page_table = PageTable::from_token(token);
5 let mut string = String::new();
6 let mut va = ptr as usize;
7 loop {
8 let ch: u8 = *(page_table.translate_va(VirtAddr::from(va)).unwrap().get_
9 ->mut());
10 if ch == 0 {
11 break;
12 } else {
13 string.push(ch as char);
14 va += 1;
15 }
16 }
17 string
18 }
19 // os/src/syscall/process.rs
20
21 pub fn sys_exec(path: *const u8) -> isize {
22 let token = current_user_token();
23 let path = translated_str(token, path);
24 if let Some(data) = get_app_data_by_name(path.as_str()) {
25 let task = current_task().unwrap();
26 task.exec(data);
27 0
28 } else {
29 -1
30 }
31 }
```

(下页继续)

(续上页)

```
30 }
31 }
```

应用在 `sys_exec` 系统调用中传递给内核的只有一个要执行的应用名字符串在当前应用地址空间中的起始地址，如果想在内核中具体获得字符串的话就需要手动查页表。第 3 行的 `translated_str` 便可以从内核地址空间之外的某个应用的用户态地址空间中拿到一个字符串，其原理就是针对应用的字符串中字符的用户态虚拟地址，查页表，找到对应的内核虚拟地址，逐字节地构造字符串，直到发现一个 `\0` 为止（第 7~15 行）。

回到 `sys_exec` 的实现，它调用 `translated_str` 找到要执行的应用名并试图在应用加载器提供的 `get_app_data_by_name` 接口中找到对应的 ELF 格式的数据。如果找到，就调用 `TaskControlBlock::exec` 替换掉地址空间并返回 0。这个返回值其实并没有意义，因为我们在替换地址空间的时候本来就对 Trap 上下文重新进行了初始化。如果没有找到，就不做任何事情并返回 -1。在 `shell` 程序 `-user_shell` 中我们也正是通过这个返回值来判断要执行的应用是否存在。

## 系统调用后重新获取 Trap 上下文

过去的 `trap_handler` 实现是这样处理系统调用的：

```
1 // os/src/trap/mod.rs
2
3 #[no_mangle]
4 pub fn trap_handler() -> ! {
5 set_kernel_trap_entry();
6 let cx = current_trap_cx();
7 let scause = scause::read();
8 let stval = stval::read();
9 match scause.cause() {
10 Trap::Exception(Exception::UserEnvCall) => {
11 cx.sepc += 4;
12 cx.x[10] = syscall(cx.x[17], [cx.x[10], cx.x[11], cx.x[12]]) as usize;
13 }
14 ...
15 }
16 trap_return();
17 }
```

这里的 `cx` 是当前应用的 Trap 上下文的可变引用，我们需要通过查页表找到它具体被放在哪个物理页帧上，并构造相同的虚拟地址来在内核中访问它。对于系统调用 `sys_exec` 来说，一旦调用它之后，我们会发现 `trap_handler` 原来上下文中的 `cx` 失效了——因为它是用来访问之前地址空间中 Trap 上下文被保存在的那个物理页帧的，而现在它已经被回收掉了。因此，为了能够处理类似的这种情况，我们在 `syscall` 分发函数返回之后需要重新获取 `cx`，目前的实现如下：

```
1 // os/src/trap/mod.rs
2
3 #[no_mangle]
4 pub fn trap_handler() -> ! {
5 set_kernel_trap_entry();
6 let scause = scause::read();
7 let stval = stval::read();
8 match scause.cause() {
9 Trap::Exception(Exception::UserEnvCall) => {
10 // jump to next instruction anyway
11 let mut cx = current_trap_cx();
12 cx.sepc += 4;
13 }
14 }
15 }
```

(下页继续)

(续上页)

```

13 // get system call return value
14 let result = syscall(cx.x[17], [cx.x[10], cx.x[11], cx.x[12]]);
15 // cx is changed during sys_exec, so we have to call it again
16 cx = current_trap_cx();
17 cx.x[10] = result as usize;
18 }
19 ...
20 }
21 trap_return();
22 }
```

#### 6.4.5 shell 程序 user\_shell 的输入机制

为了实现 shell 程序 user\_shell 的输入机制，我们需要实现 sys\_read 系统调用使得应用能够取得用户的键盘输入。

```

1 // os/src/syscall/fs.rs
2
3 use crate::sbi::console_getchar;
4
5 const FD_STDIN: usize = 0;
6
7 pub fn sys_read(fd: usize, buf: *const u8, len: usize) -> isize {
8 match fd {
9 FD_STDIN => {
10 assert_eq!(len, 1, "Only support len = 1 in sys_read!");
11 let mut c: usize;
12 loop {
13 c = console_getchar();
14 if c == 0 {
15 suspend_current_and_run_next();
16 continue;
17 } else {
18 break;
19 }
20 }
21 let ch = c as u8;
22 let mut buffers = translated_byte_buffer(current_user_token(), buf, len);
23 unsafe { buffers[0].as_mut_ptr().write_volatile(ch); }
24 1
25 }
26 _ => {
27 panic!("Unsupported fd in sys_read!");
28 }
29 }
30 }
```

目前我们仅支持从标准输入 FD\_STDIN 即文件描述符 0 读入，且单次读入的长度限制为 1，即每次只能读入一个字符。我们调用 sbi 子模块提供的从键盘获取输入的接口 console\_getchar，如果返回 0 则说明还没有输入，我们调用 suspend\_current\_and\_run\_next 暂时切换到其他进程，等下次切换回来的时候再看看是否有输入了。获取到输入之后，我们退出循环并手动查页表将输入的字符正确的写入到应用地址空间。

注：我们这里还没有涉及 文件 的概念，在后续章节中有具体的介绍。

## 6.4.6 进程资源回收机制

### 进程的退出

当应用调用 `sys_exit` 系统调用主动退出或者出错由内核终止之后，会在内核中调用 `exit_current_and_run_next` 函数退出当前进程并切换到下一个进程。使用方法如下：

```

1 // os/src/syscall/process.rs
2
3 pub fn sys_exit(exit_code: i32) -> ! {
4 exit_current_and_run_next(exit_code);
5 panic!("Unreachable in sys_exit!");
6 }
7
8 // os/src/trap/mod.rs
9
10 #[no_mangle]
11 pub fn trap_handler() -> ! {
12 set_kernel_trap_entry();
13 let scause = scause::read();
14 let stval = stval::read();
15 match scause.cause() {
16 Trap::Exception(Exception::StoreFault) |
17 Trap::Exception(Exception::StorePageFault) |
18 Trap::Exception(Exception::InstructionFault) |
19 Trap::Exception(Exception::InstructionPageFault) |
20 Trap::Exception(Exception::LoadFault) |
21 Trap::Exception(Exception::LoadPageFault) => {
22 println!(
23 "[kernel] {:?} in application, bad addr = {:#x}, bad instruction = {:#x}, core dumped.",
24 scause.cause(),
25 stval,
26 current_trap_cx().sepc,
27);
28 // page fault exit code
29 exit_current_and_run_next(-2);
30 }
31 Trap::Exception(Exception::IllegalInstruction) => {
32 println!("[kernel] IllegalInstruction in application, core dumped.");
33 // illegal instruction exit code
34 exit_current_and_run_next(-3);
35 }
36 ...
37 }
38 trap_return();
39 }
```

相比前面的章节，`exit_current_and_run_next` 带有一个退出码作为参数。当在 `sys_exit` 正常退出的时候，退出码由应用传到内核中；而出错退出的情况（如第 29 行的访存错误或第 34 行的非法指令异常）则是由内核指定一个特定的退出码。这个退出码会在 `exit_current_and_run_next` 写入当前进程的进程控制块中：

```

1 // os/src/mm/memory_set.rs
2
3 impl MemorySet {
4 pub fn recycle_data_pages(&mut self) {
```

(下页继续)

(续上页)

```

5 self.areas.clear();
6 }
7 }
8
9 // os/src/task/mod.rs
10
11 pub fn exit_current_and_run_next(exit_code: i32) {
12 // take from Processor
13 let task = take_current_task().unwrap();
14 // **** access current TCB exclusively
15 let mut inner = task.inner_exclusive_access();
16 // Change status to Zombie
17 inner.task_status = TaskStatus::Zombie;
18 // Record exit code
19 inner.exit_code = exit_code;
20 // do not move to its parent but under initproc
21
22 // ++++++ access initproc TCB exclusively
23 {
24 let mut initproc_inner = INITPROC.inner_exclusive_access();
25 for child in inner.children.iter() {
26 child.inner_exclusive_access().parent = Some(Arc::downgrade(&INITPROC));
27 initproc_inner.children.push(child.clone());
28 }
29 }
30 // ++++++ stop exclusively accessing parent PCB
31
32 inner.children.clear();
33 // deallocate user space
34 inner.memory_set.recycle_data_pages();
35 drop(inner);
36 // **** stop exclusively accessing current PCB
37 // drop task manually to maintain rc correctly
38 drop(task);
39 // we do not have to save task context
40 let mut _unused = TaskContext::zero_init();
41 schedule(&mut _unused as *mut_);
42 }

```

- 第 13 行我们调用 `take_current_task` 来将当前进程控制块从处理器监控 `PROCESSOR` 中取出而不是得到一份拷贝，这是为了正确维护进程控制块的引用计数；
- 第 17 行我们将进程控制块中的状态修改为 `TaskStatus::Zombie` 即僵尸进程，这样它后续才能被父进程在 `waitpid` 系统调用的时候回收；
- 第 19 行我们将传入的退出码 `exit_code` 写入进程控制块中，后续父进程在 `waitpid` 的时候可以收集；
- 第 24-26 行所做的事情是将当前进程的所有子进程挂在初始进程 `initproc` 下面，其做法是遍历每个子进程，修改其父进程为初始进程，并加入初始进程的孩子向量中。第 32 行将当前进程的孩子向量清空。
- 第 34 行对于当前进程占用的资源进行早期回收。在第 4 行可以看出，`MemorySet::recycle_data_pages` 只是将地址空间中的逻辑段列表 `areas` 清空（即执行 `Vec` 向量清空），这将导致应用地址空间被回收（即进程的数据和代码对应的物理页帧都被回收），但用来存放页表的那些物理页帧此时还不会被回收（会由父进程最后回收子进程剩余的占用资源）。
- 最后在第 41 行我们调用 `schedule` 触发调度及任务切换，由于我们再也不会回到该进程的执行过程

中，因此无需关心任务上下文的保存。

## 父进程回收子进程资源

父进程通过 `sys_waitpid` 系统调用来回收子进程的资源并收集它的一些信息：

```

1 // os/src/syscall/process.rs
2
3 /// If there is not a child process whose pid is same as given, return -1.
4 /// Else if there is a child process but it is still running, return -2.
5 pub fn sys_waitpid(pid: isize, exit_code_ptr: *mut i32) -> isize {
6 let task = current_task().unwrap();
7 // find a child process
8
9 // ---- access current TCB exclusively
10 let mut inner = task.inner_exclusive_access();
11 if inner.children
12 .iter()
13 .find(|p| {pid == -1 || pid as usize == p.getpid()})
14 .is_none() {
15 return -1;
16 // ---- stop exclusively accessing current PCB
17 }
18 let pair = inner.children
19 .iter()
20 .enumerate()
21 .find(|(_ , p)| {
22 // ++++ temporarily access child PCB exclusively
23 p.inner_exclusive_access().is_zombie() && (pid == -1 || pid as usize == p.
24 getpid())
25 // ++++ stop exclusively accessing child PCB
26 });
27 if let Some((idx, _)) = pair {
28 let child = inner.children.remove(idx);
29 // confirm that child will be deallocated after removing from children list
30 assert_eq!(Arc::strong_count(&child), 1);
31 let found_pid = child.getpid();
32 // ++++ temporarily access child TCB exclusively
33 let exit_code = child.inner_exclusive_access().exit_code;
34 // ++++ stop exclusively accessing child PCB
35 *translated_refmut(inner.memory_set.token(), exit_code_ptr) = exit_code;
36 found_pid as isize
37 } else {
38 -2
39 }
40 // ---- stop exclusively accessing current PCB automatically
41 }
42
43 // user/src/lib.rs
44
45 pub fn wait(exit_code: &mut i32) -> isize {
46 loop {
47 match sys_waitpid(-1, exit_code as *mut _) {
48 -2 => { yield_(); }
49 // -1 or a real pid
50 exit_pid => return exit_pid,
51 }
52 }
53 }
```

(下页继续)

(续上页)

```
51 }
52 }
```

sys\_waitpid 是一个立即返回的系统调用，它的返回值语义是：如果当前的进程不存在一个进程 ID 为 pid (pid== -1 或 pid > 0) 的子进程，则返回 -1；如果存在一个进程 ID 为 pid 的僵尸子进程，则正常回收并返回子进程的 pid，并更新系统调用的退出码参数为 exit\_code。这里还有一个 -2 的返回值，它的含义是子进程还没退出，通知用户库 user\_lib (是实际发出系统调用的地方)，这样用户库看到是 -2 后，就进一步调用 sys\_yield 系统调用 (第 46 行)，让当前父进程进入等待状态。

注：在编写应用的开发者看来，位于用户库 user\_lib 中的 wait/waitpid 两个辅助函数都必定能够返回一个有意义的结果，要么是 -1，要么是一个正数 PID，是不存在 -2 这种通过等待即可消除的中间结果的。让调用 wait/waitpid 两个辅助函数的进程等待正是在用户库 user\_lib 中完成。

第 11~17 行判断 sys\_waitpid 是否会返回 -1，这取决于当前进程是否有一个符合要求的子进程。当传入的 pid 为 -1 的时候，任何一个子进程都算是符合要求；但 pid 不为 -1 的时候，则只有 PID 恰好与 pid 相同的子进程才算符合条件。我们简单通过迭代器即可完成判断。

第 18~26 行判断符合要求的子进程中是否有僵尸进程，如果说有的话还需要同时找出它在当前进程控制块子进程向量中的下标。如果找不到的话直接返回 -2，否则进入第 27~35 行的处理：

- 第 27 行我们将子进程从向量中移除并置于当前上下文中；
- 第 29 行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：内核栈和它的 PID 还有它的应用地址空间存放页表的那些物理页帧等等。

剩下主要是将收集的子进程信息返回。

- 第 30 行得到子进程的 PID 并会在最终返回；
- 第 32 行得到了子进程的退出码；
- 第 34 行写入到当前进程的应用地址空间中。由于应用传递给内核的仅仅是一个指向应用地址空间中保存子进程返回值的内存区域的指针，我们还需要在 translated\_refmut 中手动查页表找到应该写入到物理内存中的哪个位置，这样才能把子进程的退出码 exit\_code 返回给父进程。其实现可以在 os/src/mm/page\_table.rs 中找到，比较简单，在这里不再赘述。

到这里，“伤齿龙”操作系统就算完成了。它在启动后，会加载执行用户态的 shell 程序，并可以通过 shell 程序提供的命令行交互界面，让使用者敲入要执行的应用程序名字，就可以创建一个子进程来执行这个应用程序，实现了灵活的人机交互和进程管理的动态灵活性。

## 6.5 进程调度

### 6.5.1 本节导读

计算机内存中可执行的程序个数大于处理器个数时，这些程序可通过共享处理器来完成各自的任务。而操作系统负责让它们能够高效合理地共享处理器资源，这就引入了调度 (scheduling) 这个概念。进程调度 (也称处理器调度) 是进程管理的重要组成部分。

在计算机出现之前，调度这个概念就出现在人类的生活与工作环境中了，如在商店排队购买商品、汽车装配线调度、工厂作业车间调度等。调度的一般定义是：在一定的约束条件下，把有限的资源在时间上分配给若干个任务，以满足或优化一个或多个性能指标。对于计算机系统而言，就是在一台计算机中运行了多个进程，操作系统把有限的处理器在时间上分配给各个进程，以满足或优化进程执行的性能指标。

所以本节要阐述的核心问题是：操作系统如何通过进程调度来提高进程和系统的性能。我们可以把这个问题进一步细化为操作系统的一系列关键子问题：

- 运行进程的约束条件是啥？
- 有哪些调度策略和算法？
- 调度的性能指标是啥？
- 如何评价调度策略和算法？

## 6.5.2 回顾历史

### 本章之前的操作系统实例

本章之前已经实现了多个操作系统实例，它们相对比较简单，重点体现各种操作系统核心知识点的设计与实现。对于 **处理器调度** 这个核心知识点，从面向批处理的“邓式鱼”操作系统开始，就有非常简单的设计与实现。

最早的“三叶虫”操作系统以库的形式支持单个裸机应用程序，单个应用程序独占整个计算机系统，这时还没有调度的必要性。在批处理系统中，需要支持多个程序运行，“邓式鱼”操作系统把要执行的应用按名字排名的先后顺序，按一次加载一个应用的方式把应用放入内存中。当一个应用执行完毕后，再加载并执行下一个应用。这就是一种简单的“排名”调度策略。

面向多道程序的“锯齿螈”操作系统可以把多个要执行的程序放到内存中，其调度策略与“邓式鱼”操作系统一样，也是“排名”调度策略。当进化到“始初龙”操作系统后，我们把运行的程序称为任务，并且任务可以主动放弃处理器。操作系统会把各个任务放到一个调度队列中，从队列头选择一个任务执行。当任务主动放弃处理器或任务执行完毕后，操作系统从队列头选择下一个任务执行，并把主动放弃处理器的任务安置在队列尾。这是一种先来先服务的调度策略，实现起来非常简单。

进一步进化的面向分时多任务的“腔骨龙”操作系统继承了“始初龙”操作系统的调度策略和组织方式，也会把各个任务放到一个调度队列中，并从队列头选择一个任务执行。但它还考虑了应用间的公平性和系统的执行效率。为此，它给每个任务分配了一个固定的时间片。当正在执行的任务消耗完分配给它的时间片后，操作系统就可以通过时钟中断抢占正在执行的任务，把处理器分配给其他任务执行，而被抢占的任务将放置到队列尾。这是一种基于时间片的轮转调度策略。后续的支持地址空间的“头甲龙”操作系统和支持进程的“伤齿龙”操作系统都采用了这种轮转调度策略。

### 计算机发展历史中的调度

早期应用主要面向计算密集型的科学计算，用户对与计算机交互的需求还不强。在早期以纸带、卡片或磁带作为程序/数据输入的批处理系统时代，操作系统的调度很简单，只需依次加载并运行每一个作业（应用以作业的形式存在）即可。操作系统不需要考虑多用户，分时复用等情况，能让程序正常运行就很不错了。

到了多道程序系统时代，内存中存在多个应用，而应用是属于不同用户的，处理器是用户都想占用的宝贵资源。操作系统需要尽量加快应用的执行，减少用户等待应用执行结果的时间，从而提高用户的满意度。这时的调度策略主要考虑如何尽可能地让处理器一直忙起来，减少完成应用的时间。

随着用户对与计算机交互的需求越来越强烈，导致批处理任务（计算密集型应用）和交互任务（I/O 密集型应用）都需要在计算机系统上执行。计算机科学家不得不花费更大心思在操作系统的调度上，以应对不同性能指标的任务要求。这一阶段的计算机系统还是以昂贵的大型机/小型机为主，服务的用户主要来源于科学计算和商业处理等科研机构和公司，他们希望把计算机系统的性能榨干，能及时与计算机进行交互，这样才对得起他们付出的大量金钱。

当发展到了个人计算机时代，计算机的价格大幅下降，个人计算机上的多数应用相对简单，对处理器的性能要求不高，但需要通过键盘/鼠标/图形显示等 I/O 设备来展示其丰富多彩的功能。这样造成的结果是，早期操作系统面向多用户的调度功能在相对简单的单用户个人计算机上作用并不显著。

随着网络和计算机技术的发展，支持并行的多核处理器已经成为处理器的主流，以数据中心为代表的规模网络服务器集群系统改变了我们的生活。各种日常应用（搜索、网络社交、网络游戏等）会消耗数据中心中

大量的处理器资源和网络/存储资源，多个后端服务应用经常会竞争处理器，因此操作系统的调度功能再一次变得至关重要，且要应对更加复杂多样的应用需求和硬件环境。

当移动互联网成为基础设施，移动终端越来越普及，大家几乎人手一台智能手机、智能平板或智能手表等，人们关注的除了流畅地执行各种应用外，还希望这些移动终端能够长时间使用。这使得除了增加电池容量外，操作系统还能在应用不必运行时，让它们尽量休眠，并通过关闭可暂时不用的外设，来减少电量的消耗。可以看到，随着计算机系统的发展和应用需求的变化，操作系统的调度功能也会有新的变化。

虽然各个实际操作系统的调度策略比较复杂，但其基本的设计思路是可以分析清楚的。接下来，我们将针对不同计算机系统特点，简化其中的应用执行过程，形成在该系统下应用执行的约束条件；并进一步分析其对应的性能指标，提出有针对性的设计思路，阐述各种可行的调度策略。

### 6.5.3 批处理系统的调度

在设计具体的调度策略之前，需要了解计算机系统和应用的运行环境，对应用的特点和它期望的性能指标也要清楚。我们先看看批处理系统下的应用的特点和约束条件。在批处理系统下，应用以科学计算为主，I/O 操作较少，且 I/O 操作主要集中在应用开始和结束的一小段时间，应用的执行时间主要消耗在占用处理器进行计算上，且应用的大致执行时间一般可以预估到。

#### 约束条件

批处理系统中的进程有如下一些约束/前提条件：

1. 每个进程同时到达。
2. 每个进程的执行时间相同。
3. 进程的执行时间是已知的。
4. 进程在整个执行过程期间很少执行 I/O 操作。
5. 进程在执行过程中不会被抢占。

对于条件 4，可理解为在操作系统调度过程中，可以忽略进程执行 I/O 操作的开销。我们这里设定的各种条件是可以调整的，即可以进一步简化或更加贴近实际情况，这样可以简化或加强对调度策略的设计。比如，我们可以把条件 2 改变一下：

2. 每个进程的执行时间不同。

#### 性能指标

我们还需给出性能指标，用于衡量，比较和评价不同的调度策略。对于批处理系统中的一般应用而言，可以只有一个性能指标：周转时间 (turn around)，即进程完成时间 (completion) 与进程到达时间 (arrival) 的差值：

$$T_{\text{turn around}} = T_{\text{completion}} - T_{\text{arrival}}$$

由于前提条件 1 明确指出所有进程在同一时间到达，那么  $T_{\text{arrival}} = 0$ ，因此  $T_{\text{turn around}} = T_{\text{completion}}$ 。除了总的周转时间，我们还需要关注平均周转时间 (average turnaround) 这样的统计值：

$$T_{\text{average turnaround}} = T_{\text{turn around}} / \text{number of ready processes}$$

对于单个进程而言，平均周转时间是一个更值得关注的性能指标。

## 先来先服务

先来先服务 (first-come first-served, 也称 First-in first-out, 先进先出) 调度策略的基本思路就是按进程请求处理器的先后顺序来使用处理器。在具体实现上，操作系统首先会建立一个就绪调度队列（简称就绪队列）和一个等待队列（也称阻塞队列）。大致的调度过程如下：

- 操作系统每次执行调度时，都是从就绪队列的队头取出一个进程来执行；
- 当一个应用被加载到内存，并创建对应的进程，设置进程为就绪进程，按进程到达的先后顺序，把进程放入就绪调度队列的队尾；
- 当正在运行的进程主动放弃处理器时，操作系统会把该进程放到就绪队列末尾，并从就绪队列头取出一个进程执行；
- 当正在运行的进程执行完毕时，操作系统会回收该进程所在资源，并从就绪队列头取出一个进程执行；
- 当正在运行的进程需要等待某个事件或资源时，操作系统会把该进程从就绪队列中移出，放到等待队列中（此时这个进程从就绪进程变成等待进程），并从就绪队列头取出下一个进程执行；
- 当等待进程所等待的某个事件出现或等待的资源得到满足时，操作系统会把该进程转为就绪进程，并会把该进程从等待队列中移出，并放到就绪队列末尾。



该调度策略的优点是简单，容易实现。对于满足 1~5 约束条件的执行环境，用这个调度策略的平均周转时间性能指标也很好。如果在一个在较长的时间段内，每个进程都能结束，那么公平性这个指标也是能得到保证的。

操作系统不会主动打断进程的运行。

## 最短作业优先

满足 1~5 的约束条件的执行环境太简化和理想化了，在实际系统中，每个应用的执行时间很可能不同，所以约束条件 2 “每个进程的执行时间相同” 就不合适了。如果把约束条件 2 改为 “每个进程的执行时间不同”，那么在采用先来先服务调度策略的系统中，可能就会出现短进程不得不等长进程结束后才能运行的现象，导致短进程的等待时间太长，且系统的平均周转时间也变长了。

假设有两个进程 PA、PB，它们大致同时到达，但 PA 稍微快一点，进程 PA 执行时间为 100，进程 PB 的执行时间为 20。如果操作系统采用先来先服务的调度策略，进程的平均周转时间为：

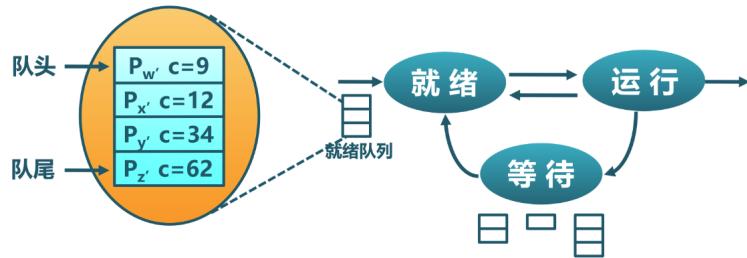
$$(100+120)/2 = 110$$

但如果操作系统先调度进程 PB，那么进程的平均周转时间为：

$$(20+120)/2 = 70$$

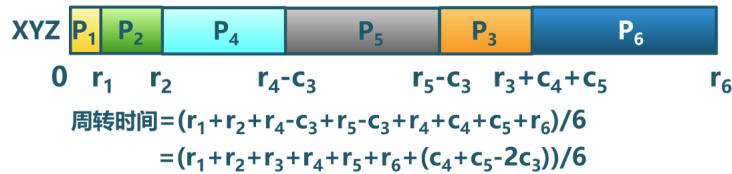
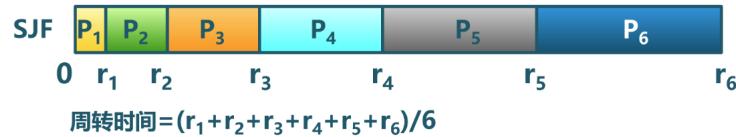
可以看到，如果采用先来先服务调度策略，执行时间短的进程（简称短进程）可被排在执行时间长的进程（长进程）后面，导致进程的平均周转时间变长。

为应对短进程不得不等长进程结束后才能运行的问题，我们可以想到一个调度的方法：优先让短进程执行。这就是最短作业优先（Shortest Job First，简称 SJF）调度策略。其实上面让 PB 先执行的调度方法，就是采用了最短作业优先策略。



在更新约束条件 2 的前提下，如果我们把平均周转时间作为唯一的性能指标，那么 SJF 是一个最优调度算法。这可以用数学方法进行证明。如果有同学感兴趣，可以试试。

#### SJF 算法中一组进程的平均周转时间



虽然 SJF 调度策略在理论上表现很不错，但在具体实现中，需要对处于就绪队列上的进程按执行时间进行排序，这会引入一定的调度执行开销。而且如果进一步放宽约束，贴近实际情况，SJF 就会显现出它的不足。如果我们放宽约束条件 1：

1. 每个进程可以在不同时间到达。

那么可能会发生一种情况，当前正在运行的进程还需 k 执行时间才能完成，这时来了一个执行时间为 h 的进程，且 h < K，但根据约束条件 5，操作系统不能强制切换正在运行的进程。所以，在这种情况下，最短作业优先的含义就不是那么确切了，而且在理论上，SJF 也就不是最优调度算法了。

例如，操作系统采用 SJF 调度策略（不支持抢占进程），有两个进程，PA 在时间 0 到达，执行时间为 100，PB 在时间 20 到达，执行时间为 20，那么周转时间为

$$(100 - 0) + (120 - 20) = 200$$

平均周转时间为 100。

## 6.5.4 交互式系统的调度

交互式系统是指支持人机交互和各种 I/O 交互的计算机系统。可抢占任务执行的分时多任务操作系统对人机交互性和 I/O 及时响应更加友好，对进程特征的约束条件进一步放宽，进程的 **可抢占特性**需要我们重新思考如何调度。

### 约束条件

交互式系统中的进程有如下一些约束/前提条件：

1. 每个进程可不同时间到达。
2. 每个进程的执行时间不同。
3. 进程的执行时间是已知的。
4. 进程在整个执行过程期间会执行 I/O 操作。
5. 进程在执行过程中会被抢占。

相对于批处理操作系统，约束条件 4 发生了变化，这意味着在进程执行过程中，操作系统不能忽视其 I/O 操作。约束条件 5 也发生了改变，即进程可以被操作系统随时打断和抢占。

### 性能指标

操作系统支持任务/进程被抢占的一个重要目标是提高用户的交互性体验和减少 I/O 响应时间。用户希望计算机系统能及时响应他发出的 I/O 请求（如键盘、鼠标等），但平均周转时间这个性能指标不足以反映人机交互或 I/O 响应的性能。所以，我们需要定义新的性能指标—响应时间（response time）：

$$T_{\text{response time}} = T_{\text{first execution}} - T_{\text{arrival}}$$

而对应的平均响应时间（average response time）是：

$$T_{\text{average response time}} = T_{\text{response time}} / \text{number of ready processes}$$

例如，操作系统采用 SJF 调度策略（不支持抢占进程），有两个进程，PA 在时间 0 到达，执行时间为 100，PB 在时间 20 到达，执行时间为 20，那么 PA 的响应时间为 0，PB 为 80，平均响应时间为 40。

### 最短完成时间优先（STCF）

由于约束条件 5 表示了操作系统允许抢占，那么我们就可以实现一种支持进程抢占的改进型 SJF 调度策略，即最短完成时间优先（Shortest Time to Complet First）调度策略。

基于前述的例子，操作系统采用 STCF 调度策略，有两个进程，PA 在时间 0 到达，执行时间为 100，PB 在时间 20 到达，执行时间为 20，那么周转时间为

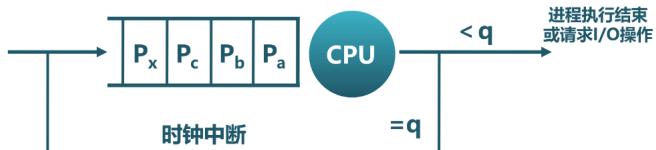
$$(120 - 0) + (40 - 20) = 140$$

平均周转时间为 70。可以看到，如果采用 STCF 调度策略，相比于 SJF 调度策略，在周转时间这个性能指标上得到了改善。

但对于响应时间而言，可能就不这么好了。考虑一个例子，有两个用户发出了执行两个进程的请求，且两个进程大约同时到达，PA 和 PB 的执行时间都为 20。我们发现，无论操作系统采用 FIFO/SJF/STCF 中的哪一种调度策略，某一个用户不得不等待 20 个时间单位后，才能让他的进程开始执行，这是一个非常不好的交互体验。从性能指标上看，响应时间比较差。这就引入了新的问题：操作系统如何支持看重响应时间这一指标的应用程序？

## 基于时间片的轮转

如果操作系统分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, quantum)，时间片一到，就抢占当前进程并切换到另外一个进程执行。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的时间内执行，从而有助于减少响应时间。这种调度策略称为轮转 (Round-Robin，简称 RR) 调度，即基本思路就是从就绪队列头取出一个进程，让它运行一个时间片，然后把它放回到队列尾，再从队列头取下一个进程执行，周而复始。



在具体实现上，需要考虑时间片的大小，一般时间片的大小会设置为时钟中断的时间间隔的整数倍。比如，时钟中断间隔为 1ms，时间片可设置为 10ms，两个用户发出了执行两个进程的请求，且两个进程大约同时到达，PA 和 PB 的执行时间都为 20s(即 20,000ms)。如果采用轮转调度，那么进程的响应时间为：

$$0+10 = 10\text{ms}$$

平均响应时间为：

$$(0+10)/2 = 5\text{ms}$$

这两个值都远小于采用之前介绍的三种调度策略的结果。这看起来不错，而且，直观上可以进一步发现，如果我们进一步减少时间片的大小，那么采用轮转调度策略会得到更好的响应时间。但其实这是有潜在问题的，因为每次进程切换是有切换代价的，参考之前介绍的进程切换的实现，可以看到，进程切换涉及多个寄存器的保存和恢复操作，页表的切换操作等。如果进程切换的时间开销是 0.5ms，时间片设置为 1ms，那么会有大约 50% 的时间用于进程切换，这样进程实际的整体执行时间就大大减少了。所以，我们需要通过在响应时间和进程切换开销之间进行权衡。不能把时间片设置得太小，且让响应时间在用户可以接受的范围内。

看来轮转调度对于响应时间这个指标很友好。但如果用户也要考虑周转时间这个指标，那轮转调度就变得不行了。还是上面的例子，我们可以看到，PA 和 PB 两个进程几乎都在 40s 左右才结束，这意味着平均周转时间为：

$$(40+40)/2 = 40\text{s}$$

这大于基于 SJF 的平均周转时间：

$$((20-0) + (40-0))/2 = 30\text{s}$$

如果活跃进程的数量增加，我们会发现轮转调度的平均周转时间会进一步加强。也许有同学会说，那我们可以通过调整时间片，把时间片拉长，这样就会减少平均周转时间了。但这样又会把响应时间也给增大了。而且如果把时间片无限拉长，轮转调度就变成了 FCFS 调度了。

到目前为止，我们看到以 SJF 为代表的调度策略对周转时间这个性能指标很友好，而以轮转调度为代表的调度策略对响应时间这个性能指标很友好。但鱼和熊掌难以兼得。

## 6.5.5 通用计算机系统的调度

个人计算机和互联网的发展推动了计算机的广泛使用，并出现了新的特点，内存越来越大，各种 I/O 设备成为计算机系统的基本配置，一般用户经常和随时使用交互式应用（如字处理、上网等），驻留在内存中的应用越来越多，应用的启动时间和执行时间无法提前知道。而且很多情况下，处理器大部分时间处于空闲状态，在等待用户或其它各种外设的输入输出操作。

### 约束条件

这样，我们的约束条件也随之发生了新的变化：

1. 每个进程可不同时间到达。
2. 每个进程的执行时间不同。
3. 进程的启动时间和执行时间是未知的。
4. 进程在整个执行过程期间会执行 I/O 操作。
5. 进程在执行过程中会被抢占。

可以看到，其中的第 3 点改变了，导致进程的特点也发生了变化。有些进程为 I/O 密集型的进程，大多数时间用于等待外设 I/O 操作的完成，需要进程能及时响应。有些进程是 CPU 密集型的，大部分时间占用处理器进行各种计算，不需要及时响应。还有一类混合型特点的进程，它在不同的执行阶段有 I/O 密集型或 CPU 密集型的特点。这使得我们的调度策略需要能够根据进程的动态运行状态进行调整，以应对各种复杂的情况。

### 性能指标

如果把各个进程运行时间的公平性考虑也作为性能指标，那么我们就需要定义何为公平。我们先给出一个公平的描述性定义：在一个时间段内，操作系统对每个处于就绪状态的进程均匀分配占用处理器的时间。

这里需要注意，为了提高一个性能指标，可能会以牺牲其他性能指标作为代价。所以，调度策略需要综合考虑和权衡各性能指标。在其中找到一个折衷或者平衡。

### 多级反馈队列调度

在无法提前知道进程执行时间的前提下，如何设计一个能同时减少响应时间和周转时间的调度策略是一个挑战。不过计算机科学家早就对此进行深入分析并提出了解决方案。在 1962 年，MIT 的计算机系教授 Fernando Jose Corbato(1990 年图灵奖获得者)首次提出多级反馈队列 (Multi-level Feedback Queue，简称 MLFQ) 调度策略，并用于当时的 CTSS (兼容时分共享系统) 操作系统中。

Corbato 教授的思路很巧妙，用四个字来总结，就是 **以史为鉴**。即根据进程过去一段的执行特征来预测其未来一段时间的执行情况，并以此假设为依据来动态设置进程的优先级，调度子系统选择优先级最高的进程执行。这里可以看出，进程有了优先级的属性，而且进程的优先级是可以根据过去行为的反馈来动态调整的，不同优先级的进程位于不同的就绪队列中。

接下来，我们逐步深入分析多级反馈队列调度的设计思想。

## 固定优先级的多级无反馈队列

MLFQ 调度策略的关键在于如何设置优先级。一旦设置好进程的优先级，MLFQ 总是优先执行位于高优先级就绪队列中的进程。对于挂在同一优先级就绪队列中的进程，采用轮转调度策略。

先考虑简单情况下，如果我们提前知道某些进程是 I/O 密集型的，某些进程是 CPU 密集型的，那么我们可以给 I/O 密集型设置高优先级，而 CPU 密集型进程设置低优先级。这样就绪队列就变成了两个，一个包含 I/O 密集型进程的高优先级队列，一个是处理器密集型的低优先级队列。

那我们如何调度呢？MLFQ 调度策略是先查看高优先级队列中是否有就绪进程，如果有，就执行它，然后基于时间片进行轮转。由于位于此高优先级队列中的进程都是 I/O 密集型进程，所以它们很快就会处于阻塞状态，等待 I/O 设备的操作完成，这就会导致高优先级队列中没有就绪进程。

在高优先级队列没有就绪进程的情况下，MLFQ 调度策略就会从低优先级队列中选择 CPU 密集型就绪进程，同样按照时间片轮转的方式进行调度。如果在 CPU 密集型进程执行过程中，某个 I/O 密集型进程所等待的 I/O 设备的操作完成了，那么操作系统会打断 CPU 密集型进程的执行，以及时响应中断，并让此 I/O 密集型进程从阻塞状态变成就绪态，重新接入到高优先级队列的尾部。这时调度子系统会优先选择高优先级队列中的进程执行，从而抢占了 CPU 密集型进程的执行。

这样，我们就得到了 MLFQ 的基本设计规则：

1. 如果进程 PA 的优先级  $>$  PB 的优先级，抢占并运行 PA。
2. 如果进程 PA 的优先级  $=$  PB 的优先级，轮转运行 PA 和 PB。

但还是有些假设过于简单化了，比如：

1. 通常情况下，操作系统并不能提前知道进程是 I/O 密集型还是 CPU 密集型的。
2. I/O 密集型进程的密集程度不一定一样，所以把它们放在一个高优先级队列中体现不出差异。
3. 进程在不同的执行阶段会有不同的特征，可能前一阶段是 I/O 密集型，后一阶段又变成了 CPU 密集型。

而在进程执行过程中固定进程的优先级，将难以应对上述情况。

## 可降低优先级的多级反馈队列

改进的 MLFQ 调度策略需要感知进程的过去执行特征，并根据这种特征来预测进程的未来特征。简单地说，就是如果进程在过去一段时间是 I/O 密集型特征，就调高进程的优先级；如果进程在过去一段时间是 CPU 密集型特征，就降低进程的优先级。由于会动态调整进程的优先级，所以，操作系统首先需要以优先级的数量来建立多个队列。当然这个数量是一个经验值，比如 Linux 操作系统设置了 140 个优先级。

那如何动态调整进程的优先级呢？首先，我们假设新创建的进程是 I/O 密集型的，可以把它设置为最高优先级。接下来根据它的执行表现来调整其优先级。如果在分配给它的时间配额内，它睡眠或等待 I/O 事件完成而主动放弃了处理器，操作系统预测它接下来的时间配额阶段很大可能还是具有 I/O 密集型特征，所以就保持其优先级不变。如果进程用完了分配给它的时间配额，操作系统预测它接下来有很大可能还是具有 CPU 密集型特征，就会降低其优先级。这里的时间配额的具体值是一个经验值，一般是时间片的整数倍。

这样，如果一个进程的执行时间小于分配给它的一个或几个时间配额，我们把这样的进程称为短进程。那么这个短进程会以比较高的优先级迅速地结束。而如果一个进程有大量的 I/O 操作，那么一般情况下，它会在时间配额结束前主动放弃处理器，进入等待状态，一旦被唤醒，会以原有的高优先级继续执行。如果一个进程的执行时间远大于几个时间配额，我们把这样的进程称为长进程。那么这个长进程经过一段时间后，会处于优先级最底部的队列，只有在没有高优先级进程就绪的情况下，它才会继续执行，从而不会影响交互式进程的响应时间。

这样，我们进一步扩展了 MLFQ 的基本规则：

3. 创建进程并让进程首次进入就绪队列时，设置进程的优先级为最高优先级。
4. 进程用完其时间配额后，就会降低其优先级。

虽然这样的调度看起来对短进程、I/O 密集型进程或长进程的支持都还不错。但这样的调度只有降低优先级的操作，对于某些情况还是会应对不足。比如：

1. 一个进程先执行了一段比较长时间的 CPU 密集型任务，导致它到了底部优先级队列，然后它在下一阶段执行 I/O 密集型任务，但被其他高优先级任务阻挡了，难以减少响应时间。
2. 在计算机系统中有大量的交互型进程，虽然每个进程执行时间短，但它们还是会持续地占用处理器，追导致位于低优先级的长进程一直无法执行，出现饥饿（starvation）现象。

这主要是调度策略还缺少提升优先级的灵活规则。

### 可提升/降低优先级的多级反馈队列

对于可降低优先级的多级反馈队列调度策略难以解决的上述情况 1 和 2，我们需要考虑如何提升某些进程的优先级。一个可以简单实现的优化思路是，每过一段时间，周期性地把所有进程的优先级都设置为最高优先级。这样长进程不会饿死；而被降到最低优先级的进程，如果当前处于 I/O 密集型任务，至少在一段时间后，会重新减少其响应时间。不过这个“一段时间”的具体值如何设置？看起来又是一个经验值。这样，我们又扩展了 MLFQ 的基本规则。

5. 经过一段时间，把所有就绪进程重新加入最高优先级队列。

但这样就彻底解决问题了吗？其实还不够，比如对于优先级低且处于 I/O 密集型任务的进程，必须等待一段时间后，才能重新加入到最高优先级，才能减少响应时间。难道这样的进程不能不用等待一段时间吗？

而对于长进程，如果有不少长进程位于最低优先级，一下子把它们都提升为最高优先级，就可能影响本来处于最高优先级的交互式进程的响应时间。看来，第 5 条规则还有进一步改进的空间，提升优先级的方法可以更灵活一些。

先看长进程，可以发现，所谓长进程“饥饿”，是指它有很长一段时间没有得到执行了。如果我们能够统计其在就绪态没有被执行的等待时间长度，就可以基于这个动态变量来逐步提升其优先级。比如每过一段时间，查看就绪进程的等待时间（进程在就绪态的等待时间）长度，让其等待时间长度与其优先级成反比，从而能够逐步第动态提升长进程的优先级。

再看优先级低且处于 I/O 密集型任务的进程，可以发现，它也有很长一段时间没有得到执行的特点，这可以通过上面的逐步提升优先级的方法获得执行的机会，并在执行 I/O 操作并处于等待状态，但此时的优先级还不够高。但操作系统在 I/O 操作完成的中断处理过程中，统计其 I/O 等待时间（进程在阻塞态下的等待时间），该进程的 I/O 等待时间越长，那么其优先级的提升度就越高，这可以使其尽快到达最高优先级。

这样根据就绪等待时间和阻塞等待时间来提升进程的优先级，可以比较好地应对上面的问题。我们可以改进第 5 条规则：

5. 定期统计进程在就绪态/阻塞态的等待时间，等待时间越长，其优先级的提升度就越高。

对于就绪态等待时间对应的优先级提升度一般时小于阻塞态等待时间对应的优先级提升度，从而让调度策略优先调度当前具有 I/O 密集型任务的进程。

经过我们总结出来的 MLFQ 调度规则，使得操作系统不需要对进程的运行方式有先验知识，而是通过观测和统计进程的运行特征来给出对应的优先级，使得操作系统能灵活支持各种运行特征的应用在计算机系统中高效执行。

## 公平份额调度

在大公司的数据中心中有着大量的计算机服务器，给互联网上的人们提供各种各样的服务。在这样的服务器中，有着相对个人计算机而言更加巨大的内存和强大的计算处理能力，给不同用户提供服务的各种进程的数量也越来越多。这个时候，面向用户或进程相对的公平性就是不得不考虑的一个问题，甚至时要优先考虑的性能指标。比如，在提供云主机的数据中心中，用户可能会希望分配 20% 的处理器时间给 Windows 虚拟机，80% 的处理器时间给 Linux 系统，如果采用公平份额调度的方式可以更简单高效。

从某种程度上看，MLFQ 调度策略总提到的优先级就是对公平性的一种划分方式，有些进程优先级高，会更快地得到处理器执行，所分配到的处理器时间也多一些。但 MLFQ 并不是把公平性放在第一位。如果把公平性放在首位，我们就可以设计出另外一类调度策略—公平份额（Fair Share，又称为比例份额，Proportional Share）调度。其基本思路是基于每个进程的重要性（即优先级）的比例关系，分配给该进程同比例的处理器执行时间。

在 1993~1994 年，MIT 的计算机系博士生 Carl A. Waldspurger 和他的导师 William E. Weihl 提出了与众不同的调度策略：彩票调度（Lottery Scheduling）和步长调度（Stride Scheduling）。它们都属于公平份额调度策略。彩票调度很有意思，它是从经济学的彩票行为中吸取营养，模拟了购买彩票和中奖的随机性，给每个进程发彩票，进程优先级越高，所得到的彩票就越多；然后每隔一段时间（如，一个时间片），举行一次彩票抽奖，抽出来的号属于哪个进程，哪个进程就能运行。

例如，计算机系统中有两个进程 PA 和 PB，优先级分别为 2 和 8，这样它们分别拥有 2 张（编号为 0-1）和 8 张彩票（编号为 2-9），按照彩票调度策略，操作系统会分配 PA 大约 20% 的处理器时间，而 PB 会分配到大约 80% 的处理器时间。

其具体实现过程是，在每个时间片到时，操作系统就抽取彩票，由于操作系统已知总彩票数有 10 张，所以操作系统产生一个从 0 和 9 之间随机数作为获奖彩票号，拥有这个彩票号的进程中奖，并获得下一次处理器执行机会。通过在一段较长的时间内不断地抽彩票，基于统计学，可以保证两个两个进程可以获得与优先级等比例的处理器执行时间。

这个彩票调度的优势有两点，第一点是可以解决饥饿问题，即使某个低优先级进程获得的彩票比较少，但经过比较长的时间，按照概率，会有获得处理器执行的时间片。第二点是调度策略的实现开销小，因为它不像之前的调度策略，还需要记录、统计、排序、查找历史信息（如统计就绪态等待时间等），彩票调度几乎不需要记录任何历史信息，只需生产一个随机数，然后查找该随机数应该属于那个进程即可。

但彩票调度虽然想法新颖，但有一个问题：如何为进程分配彩票？如果创建进程的用户清楚进程的优先级，并给进程分配对应比例的彩票，那么看起来这个问题就解决了。但彩票调度是在运行时的某个时刻产生一个随机值，并看这个随机值属于当前正在运行中的进程集合中的哪一个进程。而用户无法预知，未来的这个时刻，他创建的进程与当时的那些进程之间的优先级相对关系，这会导致公平性不一定能得到保证。

另外一个问题，基于概率的操作方法的随机性会带来不确定性，特别是在一个比较短的时间段里面，进程间的优先级比例关系与它们获得的处理器执行时间的比例关系之间有比较大的偏差，只有在执行时间很长的情况下，它们得到的处理器执行时间比例会比较接近优先级比例。

### 注解：能否用彩票来表示各种计算机资源的份额？

彩票调度中的彩票表示了进程所占处理器时间的相对比例，那么能否用彩票来表示进程占用内存或其他资源的相对比例？

为了解决彩票调度策略中的偶然出现不准确的进程执行时间比例的问题。Waldspurger 等进一步提出了步长调度（Stride Scheduling）。这是一个确定性的公平配额调度策略。其基本思路是：每个进程有一个步长（Stride）属性值，这个值与进程优先级成反比，操作系统会定期记录每个进程的总步长，即行程（pass），并选择拥有最小行程值的进程运行。

例如，计算机系统中有两个进程 PA 和 PB 几乎同时到达，优先级分别为 2 和 8，用一个预设的大整数（如 1000）去除以优先级，就可获得对应的步长，这样它们的步长分别是 500 和 125 在具体执行时，先选择 PA 执行，它在执行了一个时间片后，其行程为 500；在接下来的 4 个时间片，将选择执行行程少的 PB 执行，它在

连续执行执行 4 个时间片后，其形成也达到了 500；并这样周而复始地执行下去，直到进程执行结束。按照步长调度策略，操作系统会分配 PA 大约 20% 的处理器时间，而 PB 会分配到大约 80% 的处理器时间。

比较一下这两种调度策略，可以看出彩票调度算法只能在一段比较长的时间后，基于概率上实现优先级等比的时间分配，而步长调度算法可以在每个调度周期后做到准确的优先级等比的时间分配。但彩票算法的优势是几乎不需要全局信息，这在合理处理新加入的进程时很精炼。比如一个新进程开始执行时，按照步长调度策略，其行程值为 0，那么该进程将在一段比较长的时间内一直占用处理器执行，这就有点不公平了。如果要设置一个合理的进程值，就需要全局地统计每个进程的行程值，这就带来了比较大的执行开销。但彩票调度策略不需要统计每个进程的彩票数，只需用新进程的票数更新全局的总票数即可。

## 6.5.6 实时计算机系统的调度

计算机系统的应用领域非常广泛，如机器人、物联网、军事、工业控制等。在这些领域中，要求计算机系统能够实时响应，如果采用上述调度方式，不能满足这些需求，这对操作系统提出了新的挑战。

这里，我们首先需要理解实时的含义。实时计算机系统通常可以分为硬实时 (Hard Real Time) 和软实时 (Soft Real Time) 两类，硬实时是指任务完成时间必须在绝对的截止时间内，如果超过意味着错误和失败，可能导致严重后果。软实时是指任务完成时间尽量在绝对的截止时间内，偶尔超过可以接受。

实时的任务是由一组进程来实现，其中每个进程的行为是可预测和提前确定的。这些进程称为实时进程，它们的执行时间一般较短。支持实时任务的操作系统称为实时操作系统。



### 约束条件

实时计算机系统是一种以确定的时间范围起到主导作用的计算机系统，一旦外设发给计算机一个事件（如时钟中断、网络包到达等），计算机必须在一个确定时间范围内做出响应。

实时计算机系统中的事件可以按照响应方式进一步分类为周期性（以规则的时间间隔发生）事件或非周期性（发生时间不可预知）事件。一个系统可能要响应多个周期性事件流。根据每个事件需要处理时间的长短，系统甚至有可能无法处理完所有的事件。

这样，实时计算机系统的约束条件也随之发生了新的变化：

1. 每个进程可不同时间到达。
2. 每个进程的执行时间不同。
3. 进程的启动时间和执行时间是未知的。
4. 进程在整个执行过程期间会执行 I/O 操作。
5. 进程在执行过程中会被抢占。
6. 进程的行为是可预测和提前确定的，即进程在独占处理器的情况下，执行时间的上限是可以提前确定的。
7. 触发进程运行的事件需要进程实时响应，即进程要在指定的绝对截止时间内完成对各种事件的处理。

这里主要增加了第 6 和 7 点。第 6 点说明了实时进程的特点，第 7 点说明了操作系统调度的特点。

## 性能指标

对于实时计算机系统而言，进程的周转时间快和响应时间低这样的性能指标并不是最主要的，进程要在指定的绝对的截止时间内完成是第一要务。这里首先需要理解实时计算机系统的可调度性。如果有  $m$  个周期事件，事件  $i$  以周期时间  $P_i$  发生，并需要  $C_i$  时间处理一个事件，那么计算机系统可以处理任务量（也称负载）的条件是：

$$\text{SUM}(C_i/P_i) \leq 1$$

能满足这个条件的实时计算机系统是可实时调度的。

满足这个条件的实时系统称为是可调度的。例如，一个具有两个周期性事件的计算机系统，其事件周期分别是 20ms、80ms。如果这些事件分别需要 10ms、20ms 来进行处理，那么该计算机系统是可实时调度的，因为

$$(10/20) + (20/80) = 0.75 < 1$$

如果再增加第三个周期事件，其周期是 100ms，需要 50ms 的时间来处理，我们可以看到：

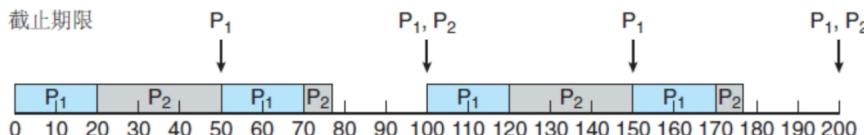
$$(10/20) + (20/80) + (50/100) = 1.25 > 1$$

这说明该计算机系统是不可实时调度的。

实时计算机系统的调度策略/算法可以是静态或动态的。静态调度在进程开始运行之前就作出调度决策；而动态调度要在运行过程中进行调度决策。只有在预知进程要所完成的工作时间上限以及必须满足的截止时间等全部信息时，静态调度才能工作；而动态调度则不需要这些前提条件。

## 速率单调调度

速率单调调度 (Rate Monotonic Scheduling, RMS) 算法是由刘炯朗 (Chung Laung Liu) 教授和 James W. Layland 在 1973 年提出的。该算法的基本思想是根据进程响应事件的执行周期的长短来设定进程的优先级，即执行周期越短的进程优先级越高。操作系统在调度过程中，选择优先级最高的就绪进程执行，高优先级的进程会抢占低优先级的进程。



该调度算法有如下的前提假设：

1. 每个周期性进程必须在其执行周期内完成，以完成对周期性事件的响应。
2. 进程执行不依赖于任何其他进程。
3. 进程的优先级在执行前就被确定，执行期间不变。
4. 进程可被抢占。

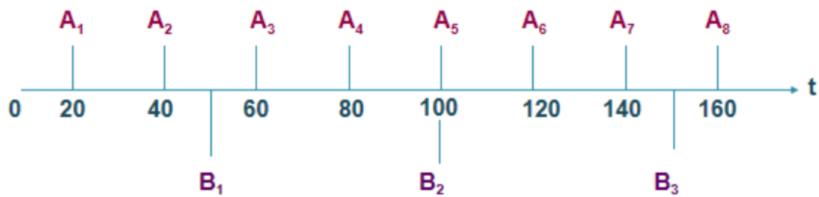
可以看出，RMS 调度算法在每个进程执行前就分配给进程一个固定的优先级，优先级等比于进程所响应的事件发生的周期频率，即进程优先级与进程执行的速率（单位时间内运行进程的次数）成线性关系，这正是为什么将其称为速率单调的原因。例如，必须每 20ms 运行一次（每秒要执行 50 次）的进程的优先级为 50，必须每 50ms 运行一次（每秒 20 次）的进程的优先级为 20。Liu 和 Layland 证明了在静态实时调度算法中，RMS 是最优的。

任务执行中间既不接收新的进程，也不进行优先级的调整或进行 CPU 抢占。因此这种算法的优点是系统消耗小，缺点是不灵活。一旦该系统的任务决定了，就不能再接收新的任务。

采用抢占的、静态优先级的策略，调度周期性任务。

## EDF 调度

另一个典型的实时调度算法是最早截止时间优先 (Earliest Deadline First, EDF) 算法, 其基本思想是根据进程的截止时间来确定任务的优先级。截止时间越早, 其优先级就越高。如果进程的截止期相同, 则处理时间短的进程优先级高。操作系统在调度过程中, 选择优先级最高的就绪进程执行, 高优先级的进程会抢占低优先级的进程。



该调度算法有如下的前提假设:

1. 进程可以是周期性或非周期性的。
2. 进程执行不依赖于任何其他进程。
3. 进程的优先级在执行过程中会基于进程的截止期动态变化。
4. 进程可被抢占。



EDF 调度算法按照进程的截止时间的早晚来分配优先级, 截止时间越近的进程优先级越高。操作系统在进行进程调度时, 会根据各个进程的截止时间重新计算进程优先级, 并选择优先级最高的进程执行, 即操作系统总是优先运行最紧迫的进程。在不同时刻, 两个周期性进程的截止时间的早晚关系可能会变化, 所以 EDF 调度算法是一种动态优先级调度算法。

## 实时调度实例

系统中有三个周期性进程 PA、PB 和 PC, 它们在一开就处于就绪状态, 它们的执行周期分别是 20ms、50ms 和 100ms, 它们响应事件的处理时间分别为 5ms、20ms 和 25ms。操作系统需要考虑如何调度 PA、PB 和 PC, 以确保它们在周期性的截止时间 (最终时限, 即当前执行周期的绝对时间) 到来前都能完成各自的任务。

我们先分析一下系统的可调度性:

$$(5/20) + (20/50) + (25/100) = 0.25 + 0.4 + 0.25 = 0.9 < 1$$

可以看到处理器在理论上有 10% 的空闲时间, 不会被超额执行, 所以找到一个合理的调度应该是可能的。我们首先看看 RMS 调度算法, 由于进程的优先级只与进程的执行周期成线性关系, 所以三个进程的优先级分别为 50、20 和 10。对于 RMS 调度算法而言, 具有如下的调度执行过程:

- t=0: 在 t=0 时刻, 优先级最高的 PA 先执行 (PA 的第一个周期开始), 并在 5ms 时完成;
- t=5: 在 PA 完成后, PB 接着执行;
- t=20: 在执行到 20ms 时 (PA 的第二个周期开始), PA 抢占 PB 并再次执行, 直到 25m 时结束;

- $t=25$ : 然后被打断的 PB 继续执行, 直到 30ms 时结束;
- $t=30$ : 接着 PC 开始执行 (PC 的第一个周期开始);
- $t=40$ : 在执行到 40ms 时 (PA 的第三个周期开始), PA 抢占 PC 并再次执行, 直到 45ms 结束;
- $t=45$ : 然后被打断的 PC 继续执行;
- $t=50$ : 然后在 50ms 时 (PB 的第二个周期), PB 抢占 PC 并再次执行;
- $t=60$ : 然后在 60ms 时 (PA 的第四个周期开始), PA 抢占 PB 并再次执行, 直到 65ms 时结束;
- $t=65$ : 接着 PB 继续执行, 并在 80ms 时结束;
- $t=80$ : 接着 PA 继续抢占 PC (PA 的第五个周期开始), 在 85ms 时结束;
- $t=85$ : 然后 PC 再次执行, 在 90ms 时结束。

这样, 在 100ms 的时间内, PA 执行了 5 个周期任务, PB 执行了 2 个周期任务, PC 执行了 1 个周期任务。在下一个 100ms 的时间内, 上述过程再次重复。

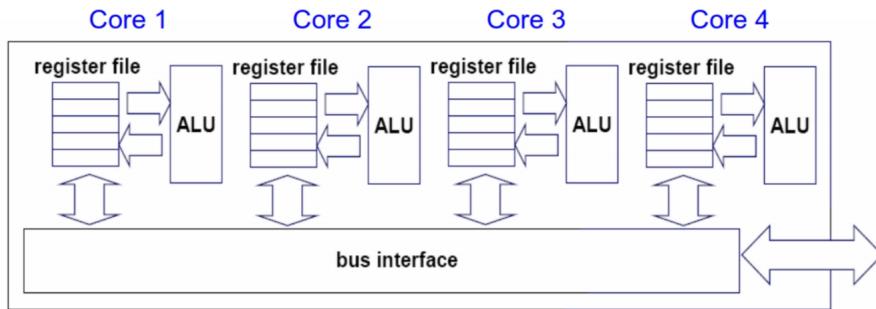
对于 EDF 调度算法而言, 具有如下的调度执行过程:

- $t=0$ : 首先选择截止时间最短的 PA, 所以它先执行 (PA 的第一个周期开始), 并在 5ms 时完成;
- $t=5$ : 在 PA 完成后, 截止时间第二的 PB 接着执行;
- $t=20$ : 在执行到 20ms 时 (PA 的第二个周期开始), PA 截止时间 40ms 小于 PB 截止时间 50ms, 所以抢占 PB 并再次执行, 直到 25ms 时结束;
- $t=25$ : 然后被打断的 PB 继续执行, 直到 30ms 时结束;
- $t=30$ : 接着 PC 开始执行 (PC 的第一个周期开始);
- $t=40$ : 在执行到 40ms 时 (PA 的第三个周期开始), PA 截止时间 40ms 小于 PC 截止时间 100ms, PA 抢占 PC 并再次执行, 直到 45ms 结束;
- $t=45$ : 然后被打断的 PC 继续执行;
- $t=50$ : 然后在 50ms 时 (PB 的第二个周期), PB 截止时间 100ms 小于等于 PC 截止时间 100ms, PB 抢占 PC 并再次执行;
- $t=60$ : 然后在 60ms 时 (PA 的第四个周期开始), PA 截止时间 80ms 小于 PB 截止时间 100ms, PA 抢占 PB 并再次执行, 直到 65ms 时结束;
- $t=65$ : 接着 PB 继续执行, 并在 80ms 时结束;
- $t=80$ : 接着 PA 截止时间 100ms 小于等于 PC 截止时间 100ms, PA 继续抢占 PC (PA 的第五个周期开始), 在 85ms 时结束;
- $t=85$ : 然后 PC 再次执行, 在 90ms 时结束。

上述例子的一个有趣的现象是, 虽然 RMS 调度算法与 EDF 的调度策略不同, 但它们的调度过程是一样的。注意, 这不是普遍现象, 也有一些例子会出现二者调度过程不同的情况, 甚至 RMS 调度无法满足进程的时限要求, 而 EDF 能满足进程的时限要求。同学们能举出这样的例子吗?

## 6.5.7 多处理器计算机系统的调度

在 2000 年前，多处理器计算机的典型代表是少见的高端服务器和超级计算机，但到了 2000 年后，单靠提高处理器的频率越来越困难，而芯片的集成度还在进一步提升，所以在一个芯片上集成多个处理器核成为一种自然的选择。到目前位置，在个人计算机、以手机为代表的移动终端上，多核处理器（Multi Core）已成为一种普遍的现象，多个处理器核能够并行执行，且可以共享 Cache 和内存。



之前提到的调度策略/算法都是面向单处理器的，如果把这些策略和算法扩展到多处理器环境下，是否需要解决新问题？

**注解：**并行处理需要了解更多的硬件并行架构问题和软件的同步互斥等技术，而深入的硬件并行架构目前不在本书的范畴之内，同步互斥等技术在后续章节才介绍。按道理需要先学习这些内容才能真正和深入理解本小节的内容，但本小节的内容在逻辑上都属于进程调度的范畴，所以就放在这里了。建议可以先大致学习本小节内容，在掌握了进程间通信、同步互斥等技术后，再回头重新学习一些本小节内容。

### 约束条件

为了理解多处理器调度需要解决的新问题，我们需要理解单处理器计算机与多处理器计算机的基本区别。对于多处理器计算机而言，每个处理器核心会有共享的 Cache，也会有它们私有的 Cache，而各自的私有 Cache 中的数据有硬件来保证数据的 Cache 一致性（也称缓存一致性）。

简单地说，位于不同私有 Cache 中的有效数据（是某一内存单元的值）要保证是相同的，这样处理器才能取得正确的数据，保证计算的正确性，这就是 Cache 一致性的基本含义。保证一致性的控制逻辑是由硬件来完成的，对操作系统和应用程序而言，是透明的。

在共享 Cache 和内存层面，由于多个处理器可以并行访问位于共享 Cache 和内存中的共享数据，所以需要有后面章节讲解的同步互斥机制来保证程序执行的正确性。这里，我们仅仅介绍一下简单的思路。

以给创建的新子进程设置进程号为例。在单处理器情况下，操作系统用一个整型全局变量保存当前可用进程号，初始值为 0。给新进程设置新进程号的过程很简单：

1. 新进程号 = 当前可用进程号；
2. 当前可用进程号 = 当前可用进程号 + 1；

在多处理器情况下，假设两个位于不同处理器上的进程都发起了创建子进程的系统调用请求，操作系统可以并行地执行创建两个子进程，而且需要给子进程设置一个新的进程号。如果没有一些同步互斥的手段，那么可能出现如下的情况：

$t0: ID-PA = CurID$   $ID-PB = CurID$   $t1: CurID = CurID+1$   $CurID = CurID + 1$

这样两个新进程的进程号就是一样的了，这就会在后续的执行中出现各种问题。为了正确处理共享变量，就需要用类似互斥锁（Mutex）的方法，让在不同处理器上执行的控制流互斥地访问共享变量，这样就能解决正确性问题。

所以，对于多处理器下运行的进程而言，新增加了如下的假设条件：

- 运行在不同处理器上的多个进程可用并行执行，但对于共享资源/变量的处理，需要有同步互斥等机制的正确性保证。

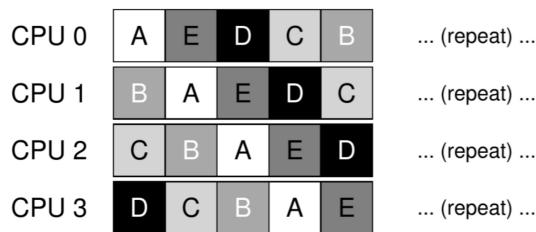
## 性能指标

这里的性能指标与之前描述的基于单处理器的通用计算机系统一样，主要是周转时间、响应时间和公平性。

## 单队列调度

对于多处理器系统而言，两个进程数量多于处理器个数，我们希望每个处理器都执行进程。这一点是之前单处理器调度不会碰到的情况。单处理器的调度只需不断地解答：“接下来应该运行哪个进程？”，而在多处理器中，调度还需解答一个问题：“要运行的进程在哪一个 CPU 上运行？”。这就增加了调度的复杂性。

如果我们直接使用单处理器调度的数据结构，其中的重点就是放置就绪进程的就绪队列或其他与调度相关的数据结构。那么这些数据结构就是需要互斥访问的共享数据。为简化分析过程，我们以轮转调度采用的单就绪队列为例，面向多处理器的单队列调度的策略逻辑没有改变，只是在读写/修改就绪队列等共享数据时，需要用同步互斥的一些操作保护起来，确保对这些共享数据访问的正确性。



采用单队列调度的一个好处是，它支持自动负载平衡，因为决不会出现一个 CPU 空闲而其他 CPU 过载的情况。

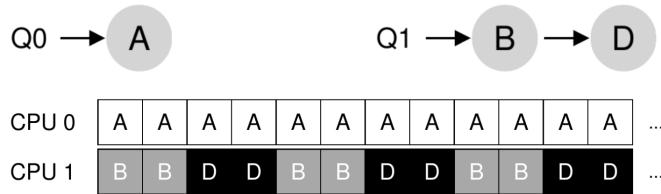
## 处理器亲和性

另外，还需考虑的一个性能问题是调度中的处理器亲和性（也称缓存亲和性、调度亲和性）问题。其基本思想是，尽量使一个进程在它前一次运行过的同一个 CPU 上运行。其原因是，现代的处理器都有私有 Cache，基于局部性的考虑，如果操作系统在下次调度时要给该进程选择处理器，会优先选择该进程上次执行所在的处理器，从而使得 Cache 中缓存的数据可重用，提高了进程执行的局部性。

## 多队列调度

如果处理器的个数较多，频繁对共享数据执行同步互斥操作的开销会很大。为此，能想到的一个方法是，还是保持单处理器调度策略的基本逻辑，但把就绪队列或和他与调度相关的数据结构按处理器个数复制多份，这样操作系统在绝大多数情况下，只需访问本处理器绑定的调度相关数据结构，就可用完成调度操作。这样在一个调度控制框架下就包含多个调度队列。当要把一个新进程或被唤醒的进程放入就绪队列时，操作系统可根据一些启发式方法（如随机选择某个处理器上的就绪队列或选择就绪进程数量最少的就绪队列）来放置进程到某个就绪队列。操作系统通过访问本处理器上的调度相关数据结构，就可以选择出要执行的进程，这样就避免了开销大的同步互斥操作。

多队列调度比单队列调度具有更好的可扩展性，多队列的数量会随着处理器的增加而增加，也具有良好的缓存亲和度。当多队列调度也有它自己的问题：负载均衡（Load Balance）问题。



考虑如下的例子，在一个有 4 个进程，两个处理器的计算机系统中，有两个就绪队列，PA 和 PB 在就绪队列 Q1，PC 和 PD 在就绪队列 Q2，如果采用基于轮转调度的多队列调度，那么两个处理器可以均匀地让 4 个进程分时使用处理器。这是一种理想的情况。如果进程 PB 结束，而调度不进行进一步的干预，那么就会出现 PA 独占处理器 1，PC 和 PD 分时共享处理器 2。如果 PA 也结束了，而调度还不进行进一步的干预，那么（Load Imbalance）就会出现处理器 1 空闲，而处理器 2 繁忙的情况，这就是典型的负载不均衡（Load Imbalance）的现象了。这就没有达到轮转调度的意图。

所以多队列调度需要解决负载不均衡的问题。一个简单的思路就是允许进程根据处理器的负载情况从一个处理器迁移到另外一个处理器上。对于上面的例子，如果是处理器 1 空闲，处理器 2 繁忙的情况，操作系统只需把处理器 2 上的进程分一半，迁移到处理器 1 即可。当如果是处理器 1 上运行了 PA，处理器 2 上运行了 PC 和 PD，这就需要统计每个进程的执行时间，根据进程的执行时间，让进程在两个处理器间不停的迁移，达到在一定时间段内，每个进程所占用的处理器时间大致相同，这就达到了轮转调度的意图，并达到了负载均衡。具体处理有多种方式，比如如下方式调度：

处理器 1: A A C A A C … 处理器 2: C D D C D D …

或者是：

处理器 1: A C A A C A … 处理器 2: C D D C D D …

当然，这个例子是一种简单的理想情况，实际的多处理器计算机系统中运行的进程行为会很复杂，除了并行执行，还有同步互斥执行、各种 I/O 操作等，这些都会对调度策略产生影响。

## 6.5.8 小结

本节对多种类型的计算机系统和不同类型的应用特征进行了分析，并给出了不同的性能指标，这些都是设计调度策略/算法的前提条件。我们给出了从简单到复杂的多种调度策略和算法，这些策略和方法相对于实际的操作系统而言，还很简单，不够实用，但其中的基本思想是一致的。如果同学们需要了解实际的操作系统调度策略和算法，建议阅读关于 UNIX、Linux、Windows 和各种 RTOS 等操作系统内核的书籍和文章，其中有关于这些操作系统的调度策略和算法的深入讲解。

## 6.6 练习

### 6.6.1 课后练习

#### 编程题

1. \* 实现一个使用 nice,fork,exec,spawn 等与进程管理相关的系统调用的 linux 应用程序。
2. \* 扩展操作系统内核，能够显示操作系统切换进程的过程。
3. \* 请阅读下列代码，分析程序的输出 A 的数量：(已知 && 的优先级比 || 高)

```
int main() {
 fork() && fork() && fork() || fork() && fork() || fork() && fork();
 printf("A");
}
```

(下页继续)

(续上页)

```

 return 0;
}

```

如果给出一个 `&& ||` 的序列，如何设计一个程序来得到答案？

4. \*\* 在本章操作系统中实现本章提出的某一种调度算法 (RR 调度除外)。
5. \*\*\* 扩展操作系统内核，支持多核处理器。
6. \*\*\* 扩展操作系统内核，支持在内核态响应并处理中断。

## 问答题

1. \* 如何查看 Linux 操作系统中的进程？
2. \* 简单描述一下进程的地址空间中有哪些数据和代码。
3. \* 进程控制块保存哪些内容？
4. \* 进程上下文切换需要保存哪些内容？
5. \*\* fork 为什么需要在父进程和子进程提供不同的返回值？
6. \*\* fork + exec 的一个比较大的问题是 fork 之后的内存页/文件等资源完全没有使用就废弃了，针对这一点，有什么改进策略？
7. \*\* 其实使用了 6 的策略之后，fork + exec 所带来的无效资源的问题已经基本被解决了，但是近年来 fork 还是在被不断的批判，那么到底是什么正在“杀死” fork？可以参考 [论文](#)。
8. \*\* 请阅读下列代码，并分析程序的输出，假定不发生运行错误，不考虑行缓冲，不考虑中断：

```

int main() {
 int val = 2;

 printf("%d", 0);
 int pid = fork();
 if (pid == 0) {
 val++;
 printf("%d", val);
 } else {
 val--;
 printf("%d", val);
 wait(NULL);
 }
 val++;
 printf("%d", val);
 return 0;
}

```

如果 fork() 之后主程序先运行，则结果如何？如果 fork() 之后 child 先运行，则结果如何？

9. \*\* 为什么子进程退出后需要父进程对它进行 wait，它才能被完全回收？
10. \*\* 有哪些可能的时机导致进程切换？
11. \*\* 请描述在本章操作系统中实现本章提出的某一种调度算法 (RR 调度除外) 的简要实现步骤。
12. \* 非抢占式的调度算法，以及抢占式的调度算法，他们的优点各是什么？
13. \*\* 假设我们简单的将进程分为两种：前台交互（要求短时延）、后台计算（计算量大）。下列进程/进程组分别是前台还是后台？a) make 编译 linux; b) vim 光标移动; c) firefox 下载影片; d) 某游戏处理玩家

点击鼠标开枪; e) 播放交响乐歌曲; f) 转码一个电影视频。除此以外, 想想你日常应用程序的运行, 它们哪些是前台, 哪些是后台的?

14. \*\*RR 算法的时间片长短对系统性能指标有什么影响?
15. \*\*MLFQ 算法并不公平, 恶意的用户程序可以愚弄 MLFQ 算法, 大幅挤占其他进程的时间。(MLFQ 的规则: “如果一个进程, 时间片用完了它还在执行用户计算, 那么 MLFQ 下调它的优先级”) 你能举出一个例子, 使得你的用户程序能够挤占其他进程的时间吗?
16. \*\*\*多核执行和调度引入了哪些新的问题和挑战?

## 6.6.2 实验练习 1

实验练习包括实践作业和问答作业两部分。实验练习 1 和实验练习 2 可以选一个完成。

### 实践作业

#### 进程创建

大家一定好奇过为啥进程创建要用 `fork + execve` 这么一个奇怪的系统调用, 就不能直接搞一个新进程吗? 思而不学则殆, 我们就来试一试! 这章的编程练习请大家实现一个完全 DIY 的系统调用 `spawn`, 用以创建一个新进程。

`spawn` 系统调用定义 (标准 `spawn` 看这里):

```
fn sys_spawn(path: *const u8) -> isize
```

- syscall ID: 400
- 功能: 新建子进程, 使其执行目标程序。
- 说明: 成功返回子进程 id, 否则返回 -1。
- 可能的错误:
  - 无效的文件名。
  - 进程池满/内存不足等资源错误。

TIPS: 虽然测例很简单, 但提醒读者 `spawn` 不必像 `fork` 一样复制父进程的地址空间。

### 实验要求

- 实现分支: ch5-lab
- 实验目录要求不变
- 通过所有测例

在 `os` 目录下 `make run TEST=1` 加载所有测例, `test_usertest` 打包了所有你需要通过的测例, 你也可以通过修改这个文件调整本地测试的内容。

challenge: 支持多核。

## 问答作业

- (1) fork + exec 的一个比较大的问题是 fork 之后的内存页/文件等资源完全没有使用就废弃了，针对这一点，有什么改进策略？
- (2) [选做，不占分] 其实使用了题(1)的策略之后，fork + exec 所带来的无效资源的问题已经基本被解决了，但是近年来 fork 还是在被不断的批判，那么到底是什么正在“杀死” fork？可以参考[论文](#)。
- (3) 请阅读下列代码，并分析程序的输出，假定不发生运行错误，不考虑行缓冲：

```
int main() {
 int val = 2;

 printf("%d", 0);
 int pid = fork();
 if (pid == 0) {
 val++;
 printf("%d", val);
 } else {
 val--;
 printf("%d", val);
 wait(NULL);
 }
 val++;
 printf("%d", val);
 return 0;
}
```

如果 fork() 之后主程序先运行，则结果如何？如果 fork() 之后 child 先运行，则结果如何？

- (4) 请阅读下列代码，分析程序的输出 A 的数量：(已知 && 的优先级比 || 高)

```
int main() {
 fork() && fork() && fork() || fork() && fork() || fork() && fork();
 printf("A");
 return 0;
}
```

[选做，不占分] 更进一步，如果给出一个 && || 的序列，如何设计一个程序来得到答案？

## 实验练习的提交报告要求

- 简单总结本次实验与上个实验相比你增加的东西。(控制在 5 行以内, 不要贴代码)
- 完成问答问题
- (optional) 你对本次实验设计及难度的看法。

### 6.6.3 实验练习 2

#### 实践作业

##### stride 调度算法

ch3 中我们实现的调度算法十分简单。现在我们要为我们的 os 实现一种带优先级的调度算法: stride 调度算法。

算法描述如下:

- (1) 为每个进程设置一个当前 stride, 表示该进程当前已经运行的“长度”。另外设置其对应的 pass 值 (只与进程的优先权有关系), 表示对应进程在调度后, stride 需要进行的累加值。
- (2) 每次需要调度时, 从当前 runnable 态的进程中选择 stride 最小的进程调度。对于获得调度的进程 P, 将对应的 stride 加上其对应的步长 pass。
- (3) 一个时间片后, 回到上一步骤, 重新调度当前 stride 最小的进程。

可以证明, 如果令  $P.pass = BigStride / P.priority$  其中  $P.priority$  表示进程的优先权 (大于 1), 而  $BigStride$  表示一个预先定义的大常数, 则该调度方案为每个进程分配的时间将与其优先级成正比。证明过程我们在这里略去, 有兴趣的同学可以在网上查找相关资料。

其他实验细节:

- stride 调度要求进程优先级  $\geq 2$ , 所以设定进程优先级  $\leq 1$  会导致错误。
- 进程初始 stride 设置为 0 即可。
- 进程初始优先级设置为 16。

为了实现该调度算法, 内核还要增加 set\_prio 系统调用

```
// syscall ID: 140
// 设置当前进程优先级为 prio
// 参数: prio 进程优先级, 要求 prio >= 2
// 返回值: 如果输入合法则返回 prio, 否则返回 -1
fn sys_set_priority(prio: isize) -> isize;
```

tips: 可以使用优先级队列比较方便的实现 stride 算法, 但是我们的实验不考察效率, 所以手写一个简单粗暴的也完全没问题。

## 实验要求

- 完成分支: ch3-lab
- 实验目录要求不变
- 通过所有测例

lab3 有 3 类测例, 在 os 目录下执行 make run TEST=1 检查基本 sys\_write 安全检查的实现, make run TEST=2 检查 set\_priority 语义的正确性, make run TEST=3 检查 stride 调度算法是否满足公平性要求, 六个子程序运行的次数应该大致与其优先级呈正比, 测试通过标准是  $\max_{prio} \frac{runtimes}{runtimes} / \min_{prio} \frac{runtimes}{runtimes} < 1.5$ .

challenge: 实现多核, 可以并行调度。

## 实验约定

在第三章的测试中, 我们对于内核有如下仅仅为了测试方便的要求, 请调整你的内核代码来符合这些要求:

- 用户栈大小必须为 4096, 且按照 4096 字节对齐。这一规定可以在实验 4 开始删除, 仅仅为通过 lab2 测例设置。

## 问答作业

### stride 算法深入

stride 算法原理非常简单, 但是有一个比较大的问题。例如两个 pass = 10 的进程, 使用 8bit 无符号整形储存 stride, p1.stride = 255, p2.stride = 250, 在 p2 执行一个时间片后, 理论上下一次应该 p1 执行。

- 实际情况是轮到 p1 执行吗? 为什么?

我们之前要求进程优先级  $\geq 2$  其实就是为了解决这个问题。可以证明, 在不考虑溢出的情况下, 在进程优先级全部  $\geq 2$  的情况下, 如果严格按照算法执行, 那么  $\text{STRIDE\_MAX} - \text{STRIDE\_MIN} \leq \text{BigStride} / 2$ 。

- 为什么? 尝试简单说明 (传达思想即可, 不要求严格证明)。

已知以上结论, 考虑溢出的情况下, 我们可以通过设计 Stride 的比较接口, 结合 BinaryHeap 的 pop 接口可以很容易的找到真正最小的 Stride。

- 请补全如下 partial\_cmp 函数 (假设永远不会相等)。

```
use core::cmp::Ordering;

struct Stride(u64);

impl PartialOrd for Stride {
 fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
 // ...
 }
}

impl PartialEq for Stride {
 fn eq(&self, other: &Self) -> bool {
 false
 }
}
```

例如使用 8 bits 存储 stride, BigStride = 255, 则:

- $(125 < 255) == \text{false}$
- $(129 < 255) == \text{true}$

### 实验练习的提交报告要求

- 简单总结与上次实验相比本次实验你增加的东西（控制在 5 行以内，不要贴代码）。
- 完成问答问题。
- (optional) 你对本次实验设计及难度/工作量的看法，以及有哪些需要改进的地方，欢迎畅所欲言。

### 参考信息

如果有兴趣进一步了解 stride 调度相关内容，可以尝试看看：

- 作者 Carl A. Waldspurger 写这个调度算法的原论文
- 作者 Carl A. Waldspurger 的博士生答辩 slide
- 南开大学实验指导中对 Stride 算法的部分介绍
- NYU OS 课关于 Stride Scheduling 的 Slide

如果有兴趣进一步了解用户态线程实现的相关内容，可以尝试看看：

- user-multitask in rv64
- 绿色线程 in x86
- x86 版绿色线程的设计实现
- 用户级多线程的切换原理

## 6.7 练习参考答案

### 6.7.1 课后练习

#### 编程题

1. \* 实现一个使用 nice,fork,exec,spawn 等与进程管理相关的系统调用的 linux 应用程序。

参考实现：

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
 int childpid;
 int i;

 if (fork() == 0) {
 //child process
 char * execv_str[] = {"echo", "child process, executed by execv", NULL};
 }
}
```

(下页继续)

(续上页)

```

if (execv("/usr/bin/echo", execv_str) < 0) {
 perror("error on exec\n");
 exit(0);
}
else{
 //parent process
 wait(&childpid);
 printf("parent process, execv done\n");
}
return 0;
}

```

2. \* 扩展操作系统内核，能够显示操作系统切换进程的过程。

体现调度的过程十分简单，只需要在调度器部分，在寻找或运行下一任务的函数中加入一些输出调试信息就可以看到效果了，但切换可能会比较频繁，因此输出会很多。

3. \* 请阅读下列代码，分析程序的输出 A 的数量：(已知 && 的优先级比 || 高)

```

int main() {
 fork() && fork() && fork() || fork() && fork() || fork() && fork();
 printf("A");
 return 0;
}

```

如果给出一个 && || 的序列，如何设计一个程序来得到答案？

22 个。&& 优先级高于 ||，根据 fork 子进程返回值为 0 父进程返回 pid 和逻辑运算符的短路现象（&& 左边为 F 即短路，|| 左边为 T 即短路），可以按 || 分割来进行判断，共  $1+1*3+3*2+3*2*2=22$ 。

```

def count_fork(seq):
 counts = [1] + [i.count(" && ") + 1 for i in seq.split(" || ")]
 total = sum([np.prod(counts[:i + 1]) for i in range(len(counts))])
 return total

```

4. \*\* 在本章操作系统中实现本章提出的某一种调度算法 (RR 调度除外)。

先来先服务调度算法 FCFS：它与 RR 调度的区别在于没有时钟中断导致的任务切换，其他细节上相似。因此基于已有的 RR 调度，删除对 Timer 中断的相关处理即可得到一个 FCFS 调度。

以 ucore 本章节代码为例，一种处理方式是将 trap.c 的 usertrap 函数中 case SupervisorTimer 部分的 yield(); 一句删除即可去掉 RR 特性，得到一个 FCFS 调度。

代码略。

5. \*\*\* 扩展操作系统内核，支持多核处理器。

题目编程内容过于复杂，不建议作为练习题。

6. \*\*\* 扩展操作系统内核，支持在内核态响应并处理中断。

题目编程内容过于复杂，不建议作为练习题。

## 问答题

1. \* 如何查看 Linux 操作系统中的进程?

使用 ps 命令, 常用方法:

```
$ ps aux
```

2. \* 简单描述一下进程的地址空间中有哪些数据和代码。

代码 (text) 段, 数据 (data) 段: 已初始化的全局变量的内存映射, bss 段: 未初始化或默认初始化为 0 的全局变量, 堆 (heap), 用户栈 (stack), 共享内存段

3. \* 进程控制块保存哪些内容?

进程标识符、进程调度信息 (进程状态, 进程的优先级, 进程调度所需的其它信息)、进程间通信信息、内存管理信息 (基地址、页表或段表等存储空间结构)、进程所用资源 (I/O 设备列表、打开文件列表等)、处理机信息 (通用寄存器、指令计数器、用户的栈指针)

4. \* 进程上下文切换需要保存哪些内容?

页全局目录、部分寄存器、内核栈、当前运行位置

5. \*\* fork 为什么需要在父进程和子进程提供不同的返回值?

可以根据返回值区分父子进程, 明确进程之间的关系, 方便用户为不同进程执行不同的操作。

6. \*\* fork + exec 的一个比较大的问题是 fork 之后的内存页/文件等资源完全没有使用就废弃了, 针对这一点, 有什么改进策略?

采用 COW(copy on write), 或使用使用 vfork 等。

7. \*\* 其实使用了 6 的策略之后, fork + exec 所带来的无效资源的问题已经基本被解决了, 但是近年来 fork 还是在被不断的批判, 那么到底是什么正在“杀死” fork? 可以参考 [论文](#)。

fork 和其他的操作不正交, 也就是 os 每增加一个功能, 都要改 fork, 这导致新功能开发困难, 设计受限. 有些和硬件相关的甚至根本无法支持 fork.

fork 得到的父子进程可能产生共享资源的冲突;

子进程继承父进程, 如果父进程处理不当, 子进程可以找到父进程的安全漏洞进而威胁父进程;

还有比如 fork 必须要虚存, SAS 无法支持等等.

8. \*\* 请阅读下列代码, 并分析程序的输出, 假定不发生运行错误, 不考虑行缓冲, 不考虑中断:

```
int main(){
 int val = 2;

 printf("%d", 0);
 int pid = fork();
 if (pid == 0) {
 val++;
 printf("%d", val);
 } else {
 val--;
 printf("%d", val);
 wait(NULL);
 }
 val++;
 printf("%d", val);
```

(下页继续)

(续上页)

```

 return 0;
}

```

如果 fork() 之后主程序先运行, 则结果如何? 如果 fork() 之后 child 先运行, 则结果如何?

01342 03412

9. \*\* 为什么子进程退出后需要父进程对它进行 wait, 它才能被完全回收?

当一个进程通过 exit 系统调用退出之后, 它所占用的资源并不能够立即全部回收, 需要由该进程的父进程通过 wait 收集该进程的返回状态并回收掉它所占据的全部资源, 防止子进程变为僵尸进程造成内存泄漏。同时父进程通过 wait 可以获取子进程执行结果, 判断运行是否达到预期, 进行管理。

10. \*\* 有哪些可能的时机导致进程切换?

进程主动放弃 cpu: 运行结束、调用 yield/sleep 等、运行发生异常中断

进程被动失去 cpu: 时间片用完、新进程到达、发生 I/O 中断等

11. \*\* 请描述在本章操作系统中实现本章提出的某一种调度算法 (RR 调度除外) 的简要实现步骤。

可降低优先级的 MLFQ: 将 manager 的进程就绪队列变为数个, 初始进程进入第一队列, 调度器每次选择第一队列的队首进程执行, 当一个进程用完时间片而未执行完, 就在将它重新添加至就绪队列时添加到下一队列, 直到进程位于底部队列。

12. \* 非抢占式的调度算法, 以及抢占式的调度算法, 他们的优点各是什么?

非抢占式: 中断响应性能好、进程执行连续, 便于分析管理

抢占式: 任务级响应时间最优, 更能满足紧迫作业要求

13. \*\* 假设我们简单的将进程分为两种: 前台交互 (要求短时延)、后台计算 (计算量大)。下列进程/或进程组分别是前台还是后台? a) make 编译 linux; b) vim 光标移动; c) firefox 下载影片; d) 某游戏处理玩家点击鼠标开枪; e) 播放交响乐歌曲; f) 转码一个电影视频。除此以外, 想想你日常应用程序的运行, 它们哪些是前台, 哪些是后台的?

前台: b,d,e

后台: a,c,f

14. \*\* RR 算法的时间片长短对系统性能指标有什么影响?

时间片太大, 可以让每个任务都在时间片内完成, 但进程平均周转时间会比较长, 极限情况下甚至退化为 FCFS;

时间片过小, 反应迅速, 响应时间会比较短, 可以提高批量短任务的完成速度。但产生大量上下文切换开销, 使进程的实际执行时间受到挤占。

因此需要在响应时间和进程切换开销之间进行权衡, 合理设定时间片大小。

15. \*\* MLFQ 算法并不公平, 恶意的用户程序可以愚弄 MLFQ 算法, 大幅挤占其他进程的时间。(MLFQ 的规则: “如果一个进程, 时间片用完了它还在执行用户计算, 那么 MLFQ 下调它的优先级”) 你能举出一个例子, 使得你的用户程序能够挤占其他进程的时间吗?

每次连续执行只进行大半个时间片长度即通过执行一个 IO 操作等让出 cpu, 这样优先级不会下降, 仍能很快得到下一次调度。

16. \*\*\* 多核执行和调度引入了哪些新的问题和挑战?

多处理机之间的负载不均问题: 在调度时, 如何保证每一个处理机的就绪队列保证优先级、性能指标的同时负载均衡

数据在不同处理机之间的共享与同步问题：除了 Cache 一致性的问题，在不同处理机上同时运行的进程可能对共享的数据区域产生相同的数据要求，这时就需要避免数据冲突，采用同步互斥机制处理资源竞争；

线程化问题：如何将单个进程分为多线程放在多个处理机上

Cache 一致性问题：由于各个处理机有自己的私有 Cache，需要保证不同处理机下的 Cache 之中的数据一致性

处理器亲和性问题：在单一处理机上运行的进程可以利用 Cache 实现内存访问的优化与加速，这就需要我们规划调度策略，尽量使一个进程在它前一次运行过的同一个 CPU 上运行，也即满足处理器亲和性。

通信问题：类似同步问题，如何降低核间的通信代价



---

## 第六章：文件系统

---

### 7.1 引言

#### 7.1.1 本章导读

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核心目标是：**让应用能够方便地把数据持久保存起来**。

大家不要被 **持久存储设备** 这个词给吓住了，这就是指计算机远古时代的卡片、纸带、磁芯、磁鼓、汞延迟线存储器等，以及到现在还在用的磁带、磁盘、硬盘、光盘、闪存、固态硬盘 (SSD, Solid-State Drive) 等存储设备。我们可以把这些设备叫做 **外存**。在本章之前，我们仅使用一种存储，就是内存（或称 RAM），内存是一种易失性存储。相比内存，外存的读写速度较慢，容量较大。但内存掉电后信息会丢失，而外存在掉电之后并不会丢失数据。因此，将需要持久保存的数据从内存写入到外存，或是从外存读入到内存是应用的一种重要需求。

---

**注解：**文件系统在 UNIX 操作系统有着特殊的地位，根据史料《UNIX: A History and a Memoir》记载，1969 年，Ken Thompson (UNIX 的作者) 在贝尔实验室比较闲，写了 PDP-7 计算机的磁盘调度算法来提高磁盘的吞吐量。为了测试这个算法，他本来想写一个批量读写数据的测试程序。但写着写着，他在某一时刻发现，这个测试程序再扩展一下，就是一个文件，再扩展一下，就是一个操作系统了。他的直觉告诉他，他离实现一个操作系统仅有 **三周之遥**。一周：写代码编辑器；一周：写汇编器；一周写 shell 程序，在写这些程序的同时，需要添加操作系统的功能（如 exec 等系统调用）以支持这些应用。结果三周后，为测试磁盘调度算法性能的 UNIX 雏形诞生了。

---

**注解：**指明方向的舵手：Multics 文件系统

计算机的第一种存储方式是图灵设计的图灵机中的纸带。在计算机最早出现的年代，纸质的穿孔卡成为了第一代的数据物理存储介质。随着各种应用对持久存储大容量数据的需求，纸带和穿孔卡很快就被放弃，在计

计算机发展历史依次出现了磁带、磁盘、光盘、闪存等各种各样的外部存储器（也称外存、辅助存储器、辅存等）。与处理器可直接寻址访问的主存（也称内存）相比，处理器不能直接访问，速度慢 1~2 个数量级，容量多两个数量级以上，且便宜。应用软件访问这些存储设备上的数据很繁琐，效率也低，于是文件系统就登场了。这里介绍一下顺序存储介质（以磁带为代表）的文件系统和随机存储介质（以磁盘为代表）的文件系统。

磁带是一种顺序存储介质，磁带的历史早于计算机，它始于 1928 年，当时它被开发用于音频存储（就是录音带）。在 1951 年，磁带首次用于在 UNIVAC I 计算机上存储数据。磁带的串行顺序访问特征对通用文件系统的创建和高效管理提出了挑战，磁带需要线性运动来缠绕和展开可能很长的介质卷轴。磁带的这种顺序运动可能需要几秒钟到几分钟才能将读/写磁头从磁带的一端移动到另一端。磁带文件系统用于存储在磁带上的文件目录和文件，为提高效率，它通常允许将文件目录与文件数据分布在一起，因此不需要耗时且重复的磁带往返线性运动来写入新数据。由于磁带容量很大，保存方便，且很便宜（磁带的成本比磁盘低一个数量级），所以到现在为止，磁带文件系统还在被需要存储大量数据的单位（如数据中心）使用。

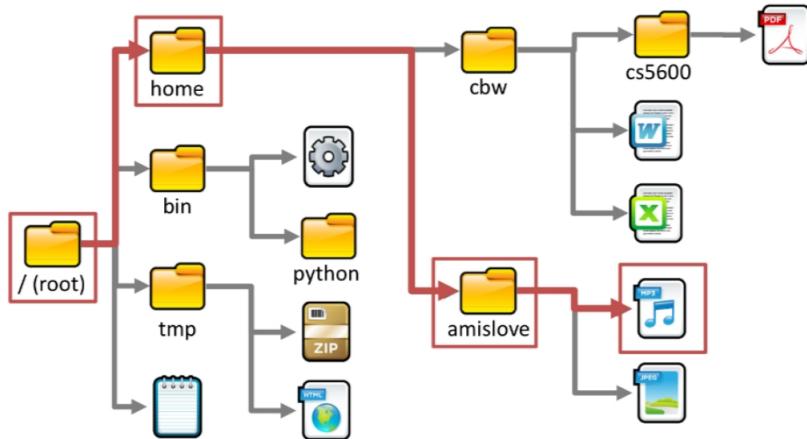
1956 年，IBM 发布了第一款硬盘驱动器，硬盘高速随机访问数据的能力使得它成为替代磁带的合理选择。在 Multics 之前，大多数操作系统一般提供特殊且复杂的文件系统来存储信息。这里的特殊和复杂性主要体现在操作系统对面向不同应用的文件数据格式的直接支持上。与当时的其他文件系统相比，Multics 文件系统不需要支持各种具体的文件数据格式，而是把文件数据看成是一个无格式的字节流，这样在一定程度上就简化了文件系统的设计。Multics 操作系统的存储管理主要面向磁盘这种辅助存储器，文件只是一个字节序列。Multics 操作系统第一次引入了层次文件系统的概念，即文件系统中的目录可以包含其他子目录，从而在理论和概念上描述了无限大的文件系统，并使得所有用户能够访问私人和共享文件。用户通过文件名来寻址文件并访问文件内容，这使得文件系统的基本结构独立于物理存储介质。文件系统可以动态加载和卸载，以便于数据存储备份等操作。可以说，Multics 的这些设计理念（提出这些设计理念的论文出现在 Multics 操作系统完成的四年前）为 UNIX 和后续操作系统中基于文件的存储管理指明了方向。

### 眼中一切皆文件的 UNIX 文件系统

而 Ken Thompson 在 UNIX 文件系统的设计和实现方面，采纳了 Multics 文件系统中的很多设计理念。UNIX 文件只是一个字节序列。文件内容的任何结构或组织仅由处理它的程序决定。UNIX 文件系统本身并不关心文件的具体内容，这意味着任何程序都可以读写任何文件。这样就避免了操作系统对各种文件内容的解析，极大地简化了操作系统的设计与实现。同时 UNIX 提出了“一切皆文件”的设计理念，这使得你几乎可以想到的各种操作系统组件都可以通过文件系统中的文件来命名。除了文件自身外，设备、管道、甚至网络、进程、内存空间都可以用文件来表示和访问。这种命名的一致性简化了操作系统的概念模型，使操作系统对外的接口组织更简单、更模块化。基本的文件访问操作包括 `open`, `read`, `write`, `close`，表示了访问一个文件最核心和基础的操作：打开文件、读文件内容、写文件内容和关闭文件。直到今天，原始 UNIX 文件系统中文件访问操作的语义几乎没有变化。

本章我们将实现一个简单的文件系统—easyfs，能够对 **持久存储设备** (Persistent Storage) 这种 I/O 资源进行管理。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。这样，就形成了具有强大 UNIX 操作系统基本功能的“霸王龙”<sup>1</sup> 操作系统。

<sup>1</sup> 霸王龙是最广为人知的恐龙，生存于约 6850 万年到 6500 万年的白垩纪最末期，位于白垩纪晚期的食物链顶端。



## 7.1.2 实践体验

获取本章代码：

```
$ git clone https://github.com/rcore-os/rCore-Tutorial-v3.git
$ cd rCore-Tutorial-v3
$ git checkout ch6
```

在 qemu 模拟器上运行本章代码：

```
$ cd os
$ make run # 编译后，最终执行如下命令模拟 rv64 virt 计算机运行：
.....
$ qemu-system-riscv64 \
-machine virt \
-nographic \
-bios ../bootloader/rustsbi-qemu.bin \
-device loader,file=target/riscv64gc-unknown-none-elf/release/os.bin,addr=0x80200000 \
-drive file=../user/target/riscv64gc-unknown-none-elf/release/fs.img,if=none,
	format=raw,id=x0 \
-device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0
```

在执行 `qemu-system-riscv64` 的参数中，`../user/target/riscv64gc-unknown-none-elf/release/fs.img` 是包含应用程序集合的文件系统镜像，这个镜像是放在虚拟硬盘块设备 `virtio-blk-device`（在下一章会进一步介绍这种存储设备）中的。

内核初始化完成之后就会进入 shell 程序，在这里我们运行一下本章的测例 `filetest_simple`：

```
>> filetest_simple
file_test passed!
Shell: Process 2 exited with code 0
>>
```

它会将 `Hello, world!` 输出到另一个文件 `filea`，并读取里面的内容确认输出正确。我们也可以通过命令行工具 `cat_filea` 来更直观的查看 `filea` 中的内容：

```
>> cat_filea
Hello, world!
```

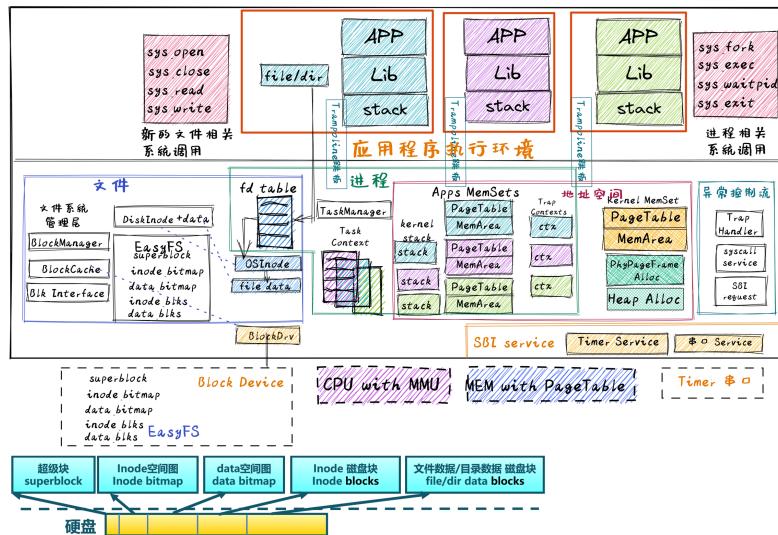
(下页继续)

(续上页)

```
Shell: Process 2 exited with code 0
>>
```

### 7.1.3 本章代码树

霸王龙操作系统-FilesystemOS 的总体结构如下图所示：



通过上图，大致可以看出霸王龙操作系统-FilesystemOS 增加了对文件系统的支持，并对应用程序提供了文件访问相关的系统调用服务。在进程管理上，进一步扩展资源管理的范围，把打开的文件相关信息放到 *fd table* 数据结构中，纳入进程的管辖中，并以此为基础，提供 *sys\_open*、*sys\_close*、*sys\_read*、*sys\_write* 与访问文件相关的系统调用服务。在设备管理层面，增加了块设备驱动-*BlockDrv*，通过访问块设备数据来读写文件系统与文件的各种数据。文件系统-EasyFS 成为 FilesystemOS 的核心内核模块，完成文件与存储块之间的数据/地址映射关系，通过块设备驱动 *BlockDrv* 进行基于存储块的读写。其核心数据结构包括：Superblock（表示整个文件系统结构）、inode bitmap（表示存放 inode 磁盘块空闲情况的位图）、data bitmap（表示存放文件数据磁盘块空闲情况的位图）、inode blks（存放文件元数据的磁盘块）和 data blks（存放文件数据的磁盘块）。EasyFS 中的块缓存管理器 *BlockManager* 在内存中管理有限个 *BlockCache* 磁盘块缓存，并通过 *Blk Interface*（与块设备驱动对接的读写操作接口）与 *BlockDrv* 块设备驱动程序进行互操作。

位于 ch6 分支上的霸王龙操作系统 - FilesystemOS 的源代码如下所示：

```

1 ./os/src
2 Rust 32 Files 2893 Lines
3 Assembly 3 Files 88 Lines
4 ./easyfs/src
5 Rust 7 Files 908 Lines
6 └── bootloader
7 └── rustsbi-qemu.bin
8 └── Dockerfile
9 └── easy-fs(新增：从内核中独立出来的一个简单的文件系统 EasyFileSystem 的实现)
10 ├── Cargo.toml
11 └── src
12 ├── bitmap.rs(位图抽象)
13 ├── block_cache.rs(块缓存层，将块设备中的部分块缓存在内存中)
14 ├── block_dev.rs(声明块设备抽象接口 BlockDevice，需要库的使用者提供其实现)
15 └── efs.rs(实现整个 EasyFileSystem 的磁盘布局)

```

(下页继续)

(续上页)

```

16 layout.rs(一些保存在磁盘上的数据结构的内存布局)
17 lib.rs
18 vfs.rs(提供虚拟文件系统的核心抽象，即索引节点 Inode)
19 easy-fs-fuse(新增：将当前 OS 上的应用可执行文件按照 easy-fs 的格式进行打包)
20 Cargo.toml
21 src
22 main.rs
23 LICENSE
24 Makefile
25 os
26 build.rs
27 Cargo.toml(修改：新增 Qemu 和 K210 两个平台的块设备驱动依赖 crate)
28 Makefile(修改：新增文件系统的构建流程)
29 src
30 config.rs(修改：新增访问块设备所需的一些 MMIO 配置)
31 console.rs
32 drivers(修改：新增 Qemu 和 K210 两个平台的块设备驱动)
33 block
34 mod.rs(将不同平台上的块设备全局实例化为 BLOCK_DEVICE)
35 ↪ 提供给其他模块使用)
36 sdcards.rs(K210 平台上的 microSD 块设备，Qemu 不会用)
37 virtio_blk.rs(Qemu 平台的 virtio-blk 块设备)
38 mod.rs
39 entry.asm
40 fs(修改：在文件系统中新增常规文件的支持)
41 inode.rs(新增：将 easy-fs 提供的 Inode 抽象封装为内核看到的 OSInode
42 并实现 fs 子模块的 File Trait)
43 mod.rs
44 pipe.rs
45 stdio.rs
46 lang_items.rs
47 link_app.S
48 linker-qemu.ld
49 loader.rs(移除：应用加载器 loader 子模块，本章开始从文件系统中加载应用)
50 main.rs
51 mm
52 address.rs
53 frame_allocator.rs
54 heap_allocator.rs
55 memory_set.rs(修改：在创建地址空间的时候插入 MMIO 虚拟页面)
56 mod.rs
57 ↪ 及其迭代器实现)
58 page_table.rs(新增：应用地址空间的缓冲区抽象 UserBuffer)
59 sbi.rs
60 syscall
61 fs.rs(修改：新增 sys_open)
62 mod.rs
63 process.rs(修改：sys_exec 改为从文件系统中加载 ELF，并支持命令行参数)
64 task
65 context.rs
66 manager.rs
67 mod.rs(修改初始进程 INITPROC 的初始化)
68 pid.rs
69 processor.rs
70 switch.rs
71 switch.S

```

(下页继续)

(续上页)

```

70 | | └── task.rs
71 | └── timer.rs
72 | └── trap
73 | ├── context.rs
74 | ├── mod.rs
75 | └── trap.S
76 └── README.md
77 └── rust-toolchain
78 └── user
79 ├── Cargo.lock
80 ├── Cargo.toml
81 ├── Makefile
82 └── src
83 ├── bin
84 | ├── cat_filea.rs (新增：显示文件filea的内容)
85 | ├── cmdline_args.rs (新增)
86 | ├── exit.rs
87 | ├── fantastic_text.rs
88 | ├── filetest_simple.rs (新增：创建文件filea并读取它的内容)
89 | ├── forktest2.rs
90 | ├── forktest.rs
91 | ├── forktest_simple.rs
92 | ├── forktree.rs
93 | ├── hello_world.rs
94 | ├── initproc.rs
95 | ├── matrix.rs
96 | ├── pipe_large_test.rs
97 | ├── pipetest.rs
98 | ├── run_pipe_test.rs
99 | ├── sleep.rs
100 | ├── sleep_simple.rs
101 | ├── stack_overflow.rs
102 | ├── user_shell.rs
103 | ├── usertests.rs
104 | └── yield.rs
105 ├── console.rs
106 ├── lang_items.rs
107 ├── lib.rs (修改：支持命令行参数解析)
108 ├── linker.ld
109 └── syscall.rs (修改：新增 sys_open)

```

## 7.1.4 本章代码导读

本章涉及的代码量相对较多，且与进程执行相关的管理还有直接的关系。其实我们是参考经典的 UNIX 基于索引结构的文件系统，设计了一个简化的有一级目录并支持 `open`, `read`, `write`, `close`，即创建/打开/读写/关闭文件一系列操作的文件系统。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。这一过程我们在前面的章节中已经重复过多次，同学应当对其比较熟悉。其中最为关键的是系统调用可以参考[sys\\_open 语义介绍](#)，此外我们还给出了测例代码解读。

### 第二步：就是要实现 easyfs 文件系统

由于 Rust 语言的特点，我们可以在用户态实现 easyfs 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，实现与 easyfs 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。

easyfs 文件系统的整体架构自下而上可分为五层：

1. 磁盘块设备接口层：读写磁盘块设备的 trait 接口
2. 块缓存层：位于内存的磁盘块数据缓存
3. 磁盘数据结构层：表示磁盘文件系统的数据结构
4. 磁盘块管理器层：实现对磁盘文件系统的管理
5. 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。在 `easy-fs/src/block_dev.rs` 中，可以看到 `BlockDevice` trait，它代表了一个抽象块设备的接口，该 trait 仅需求两个函数 `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。easy-fs 库的使用者（如操作系统内核）需要实现块设备驱动程序，并实现 `BlockDevice` trait 以提供给 easy-fs 库使用，这样 easy-fs 库就与一个具体的执行环境对接起来了。至于为什么块设备层位于 easy-fs 的最底层，那是因为文件系统仅仅是在块设备上存储的稍微复杂一点的数据。无论对文件系统的操作如何复杂，从块设备的角度看，这些操作终究可以被分解成若干次基本的块读写操作。

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 easy-fs 自底向上的第二层，即块缓存层。在 `easy-fs/src/block_cache.rs` 中，`BlockCache` 代表一个被我们管理起来的块缓冲区，它包含块数据内容以及块的编号等信息。当它被创建的时候，将触发一次 `read_block` 将数据从块设备读到它的缓冲区中。接下来只要它驻留在内存中，便可保证对于同一个块的所有操作都会直接在它的缓冲区中进行而无需额外的 `read_block`。块缓存管理器 `BlockManager` 在内存中管理有限个 `BlockCache` 并实现了类似 FIFO 的缓存替换算法，当一个块缓存被换出的时候视情况可能调用 `write_block` 将缓冲区数据写回块设备。总之，块缓存层对上提供 `get_block_cache` 接口来屏蔽掉相关细节，从而可以向上层子模块提供透明读写数据块的服务。

有了块缓存，我们就可以在内存中方便地处理 easyfs 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。easyfs 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。

easyfs 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。对于文件系统管理而言，其核心是 `EasyFileSystem` 数据结构及其关键成员函数：

- `EasyFileSystem.create`: 创建文件系统
- `EasyFileSystem.open`: 打开文件系统
- `EasyFileSystem.alloc_inode`: 分配 inode ( `dealloc_inode` 未实现，所以还不能删除文件)
- `EasyFileSystem.alloc_data`: 分配数据块
- `EasyFileSystem.dealloc_data`: 回收数据块

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，其核心是 `Inode` 数据结构及其关键成员函数：

- `Inode.new`: 在磁盘上的文件系统中创建一个 inode
- `Inode.find`: 根据文件名查找对应的磁盘上的 inode

- Inode.create: 在根目录下创建一个文件
- Inode.read\_at: 根据 inode 找到文件数据所在的磁盘数据块, 并读到内存中
- Inode.write\_at: 根据 inode 找到文件数据所在的磁盘数据块, 把内存中数据写入到磁盘数据块中

上述五层就构成了 easyfs 文件系统的整个内容。我们可以把 easyfs 文件系统看成是一个库, 被应用程序调用。而 easy-fs-fuse 这个应用就通过调用 easyfs 文件系统库中各种函数, 并作用在用 Linux 上的文件模拟的一个虚拟块设备, 就可以在这个虚拟块设备上进行各种文件操作和文件系统操作, 从而创建一个 easyfs 文件系统。

### 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 `os/src/drivers/block/virtio_blk.rs`。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。当然, 我们也可把 easyfs 文件系统烧写到 K210 开发板的存储卡中。

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 easyfs 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 easy-fs crate 提供的 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。文件的抽象 Trait File 声明在 `os/src/fs/mod.rs` 中, 它提供了 `read/write` 两个接口, 可以将数据写入应用缓冲区抽象 `UserBuffer`, 或者从应用缓冲区读取数据。应用缓冲区抽象类型 `UserBuffer` 来自 `os/src/mm/page_table.rs` 中, 它将 `translated_byte_buffer` 得到的 `Vec<&'static mut [u8]>` 进一步包装, 不仅保留了原有的分段读写能力, 还可以将其转化为一个迭代器逐字节进行读写。

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 (即 OSInode 结构)。

在具体实现上, 在进程控制块 `TaskControlBlock` 中需要加入文件描述符表字段 `fd_table`, 可以看到它是一个向量, 里面保存了若干实现了 `File` Trait 的文件, 由于采用 Rust 的 Trait Object 动态分发, 文件的类型可能各不相同。`os/src/syscall/fs.rs` 的 `sys_read/write` 两个读写文件的系统调用需要访问当前进程的文件描述符表, 用应用传入内核的文件描述符来索引对应的已打开文件, 并调用 `File` Trait 的 `read/write` 接口; `sys_close` 这可以关闭一个文件。调用 `TaskControlBlock` 的 `alloc_fd` 方法可以在文件描述符表中分配一个文件描述符。进程控制块的其他操作也需要考虑到新增的文件描述符表字段的影响, 如 `TaskControlBlock::new` 的时候需要对 `fd_table` 进行初始化, `TaskControlBlock::fork` 中则需要将父进程的 `fd_table` 复制一份给子进程。

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 文件系统接口

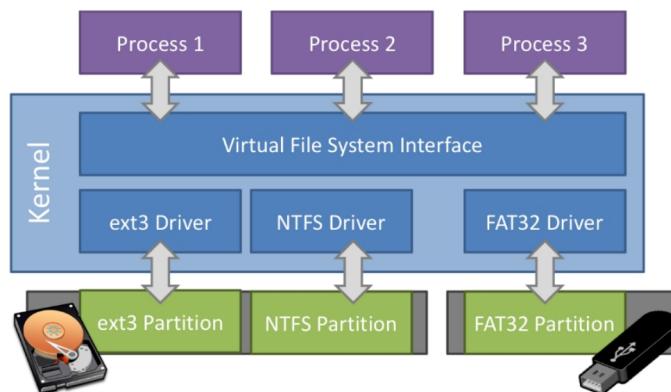
### 7.2.1 本节导读

本节首先以 Linux 上的常规文件和目录为例，站在访问文件的应用的角度，介绍文件中值得注意的地方及文件使用方法。由于 Linux 上的文件系统模型还是比较复杂，在内核实现中对它进行了很大程度的简化，我们会对简化的具体情形进行介绍。最后介绍内核上应用的开发者应该如何使用我们简化后的文件系统和一些相关知识。

### 7.2.2 文件和目录

#### 常规文件

在操作系统的用户看来，常规文件是保存在持久存储设备上的一个字节序列，每个常规文件都有一个 **文件名 (Filename)**，用户需要通过它来区分不同的常规文件。方便起见，在下面的描述中，“文件”有可能指的是常规文件、目录，也可能是之前提到的若干种进程可以读写的标准输出、标准输入、管道等 I/O 资源，请同学自行根据上下文判断取哪种含义。



在 Linux 系统上，stat 工具可以获取文件的一些信息。下面以我们项目中的一个源代码文件 os/src/main.rs 为例：

```
$ cd os/src/
$ stat main.rs
File: main.rs
Size: 940 Blocks: 8 IO Block: 4096 regular file
Device: 801h/2049d Inode: 4975 Links: 1
Access: (0644/-rw-r--r--) Uid: (1000/ oslab) Gid: (1000/ oslab)
Access: 2021-02-28 23:32:50.289925450 +0800
Modify: 2021-02-28 23:32:50.133927136 +0800
Change: 2021-02-28 23:32:50.133927136 +0800
Birth: -
```

stat 工具展示了 main.rs 的如下信息：

- File 表明它的文件名为 main.rs。
- Size 表明它的字节大小为 940 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。

- regular file 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。对于一个常规文件，我们无需关心它。
- 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。（本章的练习需要你在文件系统中实现硬链接！）
- 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 -rw-r--r--），其中第 1 位给出该文件的类型，这个文件是一个常规文件，因此这第 1 位为 -。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。

如果我们使用 stat 工具查看一个能在我们内核上执行的 ELF 可执行文件：

```
$ cd user/target/riscv64gc-unknown-none-elf/release/
$ stat user_shell
File: user_shell
Size: 85712 Blocks: 168 IO Block: 4096 regular file
Device: 801h/2049d Inode: 1460936 Links: 2
Access: (0755/-rwxr-xr-x) Uid: (1000/ oslab) Gid: (1000/ oslab)
Access: 2021-03-01 11:21:34.785309066 +0800
Modify: 2021-03-01 11:21:32.829332116 +0800
Change: 2021-03-01 11:21:32.833332069 +0800
Birth: -
```

从中可以看出我们构建的应用体积大概在数十 KiB 量级。它的 Access 指出所有用户均可将其作为一个可执行文件在当前 OS 中加载并执行。然而这仅仅是能够通过权限检查而已，这个应用只有在我们自己的内核上才能真正被加载运行。

用户常常通过文件的 **拓展名** (Filename extension) 来推断该文件的用途，如 main.rs 的拓展名是 .rs，我们由此知道它是一个 Rust 源代码文件。但从内核的角度来看，它会将所有文件无差别的看成一个字节序列，文件内容的结构和含义则是交给对应的应用进行解析。

## 目录

最早的文件系统仅仅通过文件名来区分文件，但是这会造成一些归档和管理上的困难。如今我们的使用习惯是将文件根据功能、属性的不同分类归档到不同层级的目录之下。这样我们就很容易逐级找到想要的文件。结合用户和用户组的概念，目录的存在也使得文件访问权限控制更加容易，只需要对于目录进行设置就可以间接设置用户/用户组对该目录下所有文件的访问权限，这使得操作系统能够更加安全的支持多用户情况下对不同文件的访问。

同样可以通过 stat 工具获取目录的一些信息：

```
$ stat os
File: os
Size: 4096 Blocks: 8 IO Block: 4096 directory
Device: 801h/2049d Inode: 4982 Links: 5
Access: (0755/drwxr-xr-x) Uid: (1000/ oslab) Gid: (1000/ oslab)
Access: 2021-02-28 23:32:50.133927136 +0800
Modify: 2021-02-28 23:32:50.129927180 +0800
Change: 2021-02-28 23:32:50.129927180 +0800
Birth: -
```

directory 表明 os 是一个目录，从 Access 字符串的首位 d 也可以看出这一点。对于目录而言，Access 的 rwx 含义有所不同：

- r 表示是否允许获取该目录下有哪些文件和子目录；
- w 表示是否允许在该目录下创建/删除文件和子目录；
- x 表示是否允许“通过”该目录。

Blocks 给出 os 目录也占用 8 个块进行存储。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。

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）两者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：

- main.rs 的绝对路径是 /home/oslab/workspace/v3/rCore-Tutorial-v3/os/src/main.rs；
- os 目录的绝对路径则是 /home/oslab/workspace/v3/rCore-Tutorial-v3/os/。

上面的绝对路径因具体环境而异。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。

---

### 注解：目录是否有必要存在

基于路径的索引难以并行或分布式化，因为我们总是需要查到一级目录的底层编号才能查到下一级，这是一个天然串行的过程。在一些性能需求极高的环境中，可以考虑弱化目录的权限控制职能，将目录树结构扁平化，将文件系统的磁盘布局变为类键值对存储。

---

## 文件系统

常规文件和目录都是实际保存在持久存储设备中的。持久存储设备仅支持以扇区（或块）为单位的随机读写，这和上面介绍的通过路径即可索引到文件并以字节流进行读写的用户视角有很大的不同。负责中间转换的便是 **文件系统** (File System)。具体而言，文件系统负责将逻辑上的目录树结构（包括其中每个文件或目录的数据和其他信息）映射到持久存储设备上，决定设备上的每个扇区应存储哪些内容。反过来，文件系统也可以从持久存储设备还原出逻辑上的目录树结构。

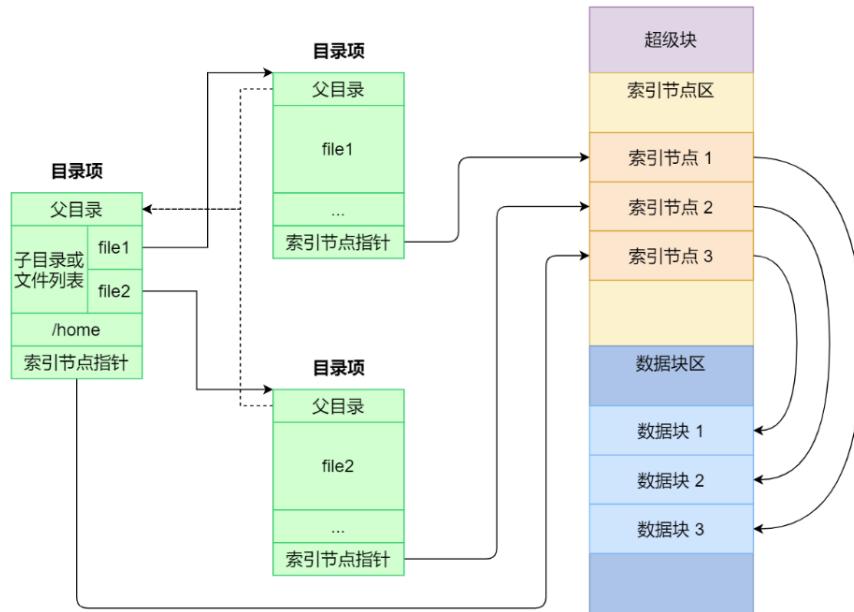
文件系统有很多种不同的实现，每一种都能将同一个逻辑上目录树结构转化为一个不同的持久存储设备上的扇区布局。最著名的文件系统有 Windows 上的 FAT/NTFS 和 Linux 上的 Ext3/Ext4/Btrfs 等。

在一个计算机系统中，可以同时包含多个持久存储设备，它们上面的数据可能是以不同文件系统格式存储的。为了能够对它们进行统一管理，在内核中有一层 **虚拟文件系统** (VFS, Virtual File System)，它规定了逻辑上目录树结构的通用格式及相关操作的抽象接口，只要不同的底层文件系统均实现虚拟文件系统要求的那些抽象接口，再加上 **挂载** (Mount) 等方式，这些持久存储设备上的不同文件系统便可以用一个统一的逻辑目录树结构一并进行管理。

### 7.2.3 简化的文件与目录抽象

我们的内核实现对于目录树结构进行了很大程度上的简化，这样做的目的是为了能够完整地展示文件系统的工作原理，但代码量又不至于太多。我们进行的简化如下：

- 扁平化：仅存在根目录 / 一个目录，剩下所有的文件都放在根目录内。在索引一个文件的时候，我们直接使用文件的文件名而不是它含有 / 的绝对路径。
- 权限控制：我们不设置用户和用户组概念，全程只有单用户。同时根目录和其他文件也都没有权限控制位，即完全不限制文件的访问方式，不会区分文件是否可执行。
- 不记录文件访问/修改的任何时间戳。
- 不支持软硬链接。
- 除了下面即将介绍的系统调用之外，其他的很多文件系统相关系统调用均未实现。

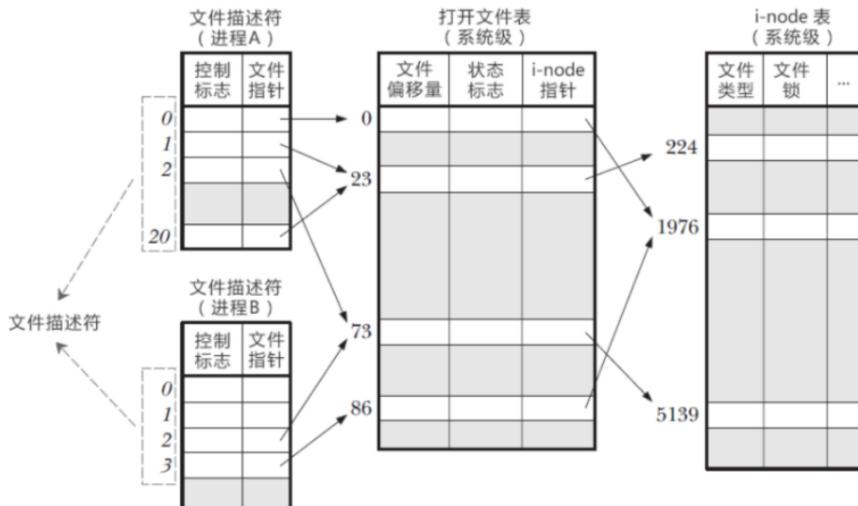


## 7.2.4 打开、关闭与读写文件的系统调用

### 文件打开

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：

```
/// 功能：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。
/// 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。
/// syscall ID: 56
fn sys_open(path: &str, flags: u32) -> isize
```



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：

```
// user/src/lib.rs

bitflags! {
 pub struct OpenFlags: u32 {
 const RDONLY = 0;
 const WRONLY = 1 << 0;
 const RDWR = 1 << 1;
 const CREATE = 1 << 9;
 const TRUNC = 1 << 10;
 }
}

pub fn open(path: &str, flags: OpenFlags) -> isize {
 sys_open(path, flags.bits)
}
```

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。

```
// user/src/syscall.rs

const SYSCALL_OPEN: usize = 56;

pub fn sys_open(path: &str, flags: u32) -> isize {
 syscall(SYSCALL_OPEN, [path.as_ptr() as usize, flags as usize, 0])
}
```

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。

## 文件关闭

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。

```
/// 功能：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。
/// 返回值：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。

// user/src/lib.rs
pub fn close(fd: usize) -> isize { sys_close(fd) }

// user/src/syscall.rs
const SYSCALL_CLOSE: usize = 57;

pub fn sys_close(fd: usize) -> isize {
 syscall(SYSCALL_CLOSE, [fd, 0, 0])
}
```

## 文件的顺序读写

在打开一个文件之后，我们就可以用之前的 `sys_read`/`sys_write` 两个系统调用来对它进行读写了。需要注意的是，常规文件的读写模式和之前介绍过的几种文件有所不同。标准输入输出和匿名管道都属于一种流式读写，而常规文件则是顺序读写和随机读写的结合。由于常规文件可以看成一段字节序列，我们应该能够随意读写它的任一段区间的数据，即随机读写。然而用户仅仅通过 `sys_read`/`sys_write` 两个系统调用不能做到这一点。

事实上，进程为每个它打开的常规文件维护了一个偏移量，在刚打开时初始值一般为 0 字节。当 `sys_read`/`sys_write` 的时候，将会从文件字节序列偏移量的位置开始 顺序 把数据读到应用缓冲区/从应用缓冲区写入数据。操作完成之后，偏移量向后移动读取/写入的实际字节数。这意味着，下次 `sys_read`/`sys_write` 将会从刚刚读取/写入之后的位置继续。如果仅使用 `sys_read`/`sys_write` 的话，则只能从头到尾顺序对文件进行读写。当我们需要从头开始重新写入或读取的话，只能通过 `sys_close` 关闭并重新打开文件来将偏移量重置为 0。为了解决这种问题，有另一个系统调用 `sys_lseek` 可以调整进程打开的一个常规文件的偏移量，这样便能对文件进行随机读写。在本教程中并未实现这个系统调用，因为对于目前实验中的应用例子，顺序文件读写功能就已经足够满足需求了。顺带一提，在文件系统的底层实现中都是对文件进行随机读写的。

下面我们从本章的测试用例 `filetest_simple` 来介绍文件系统接口的使用方法：

```

1 // user/src/bin/filetest_simple.rs
2
3 #[no_std]
4 #[no_main]
5
6 #[macro_use]
7 extern crate user_lib;
8
9 use user_lib::{
10 open,
11 close,
12 read,
13 write,
14 OpenFlags,
15 };
16
17 #[no_mangle]
18 pub fn main() -> i32 {
19 let test_str = "Hello, world!";
20 let filea = "filea\0";
21 let fd = open(filea, OpenFlags::CREATE | OpenFlags::WRONLY);
22 assert!(fd > 0);
23 let fd = fd as usize;
24 write(fd, test_str.as_bytes());
25 close(fd);
26
27 let fd = open(filea, OpenFlags::RDONLY);
28 assert!(fd > 0);
29 let fd = fd as usize;
30 let mut buffer = [0u8; 100];
31 let read_len = read(fd, &mut buffer) as usize;
32 close(fd);
33
34 assert_eq!(
35 test_str,
36 core::str::from_utf8(&buffer[..read_len]).unwrap(),
37);
}

```

(下页继续)

(续上页)

```

38 println!("file_test passed!");
39 0
40 }
```

- 第 20~25 行，我们打开文件 `filea`，向其中写入字符串 `Hello, world!` 而后关闭文件。这里需要注意的是我们需要为字符串字面量手动加上 `\0` 作为结尾。在打开文件时 `CREATE` 标志使得如果 `filea` 原本不存在，文件系统会自动创建一个同名文件，如果已经存在的话则会清空它的内容。而 `WRONLY` 使得此次只能写入该文件而不能读取。
- 第 27~32 行，我们以只读 `RONLY` 的方式将文件 `filea` 的内容读取到缓冲区 `buffer` 中。注意我们很清楚 `filea` 的总大小不超过缓冲区的大小，因此通过单次 `read` 即可将 `filea` 的内容全部读取出来。而更常见的情况是需要进行多次 `read` 直到它的返回值为 0 才能确认文件的内容已被读取完毕了。
- 最后的第 34~38 行我们确认从 `filea` 读取到的内容和之前写入的一致，则测试通过。

## 7.3 简易文件系统 easy-fs

### 7.3.1 本节导读

本节我们介绍一个简易文件系统的实现 `easy-fs`。作为一个文件系统而言，它的磁盘布局（为了叙述方便，我们用磁盘来指代一系列持久存储设备）体现在磁盘上各扇区的内容上，而它解析磁盘布局得到的逻辑目录树结构则是通过内存上的数据结构来访问的，这意味着它要同时涉及到对磁盘和对内存的访问。它们的访问方式是不同的，对于内存直接通过一条指令即可直接读写内存相应的位置，而磁盘的话需要用软件的方式向磁盘发出请求来间接进行读写。因此，我们也要特别注意哪些数据结构是存储在磁盘上，哪些数据结构是存储在内存中的，这样在实现的时候才不会引起混乱。

### 7.3.2 松耦合模块化设计思路

大家可以看到，本章的内核功能越来越多，代码量也越来越大（但仅仅是 Linux 代码量的万分之一左右）。为了减少同学学习内核的分析理解成本，我们需要让内核的各个部分之间尽量松耦合，所以 `easy-fs` 被从内核中分离出来，它的实现分成两个不同的 `crate`：

- `easy-fs` 为简易文件系统的核心部分，它是一个库形式 `crate`，实现一种简单的文件系统磁盘布局；
- `easy-fs-fuse` 是一个能在开发环境（如 Ubuntu）中运行的应用程序，它可以对 `easy-fs` 进行测试，或者将为我们内核开发的应用打包为一个 `easy-fs` 格式的文件系统镜像。

这样，整个 `easy-fs` 文件系统的设计开发可以按照应用程序库的开发过程来完成。而且在开发完毕后，可直接放到内核中，形成有文件系统支持的新内核。

能做到这一点，是由于我们在 `easy-fs` 设计上，采用了松耦合模块化设计思路。`easy-fs` 与底层设备驱动之间通过抽象接口 `BlockDevice` 来连接，避免了与设备驱动的绑定。`easy-fs` 通过 Rust 提供的 `alloc` `crate` 来隔离了操作系统内核的内存管理，避免了直接调用内存管理的内核函数。在底层驱动上，采用的是轮询的方式访问 `virtio_blk` 虚拟磁盘设备，从而避免了访问外设中断的相关内核函数。`easy-fs` 在设计中避免了直接访问进程相关的数据和函数，从而隔离了操作系统内核的进程管理。

同时，`easy-fs` 本身也划分成不同的层次，形成层次化和模块化的设计架构。`easy-fs` `crate` 自下而上大致可以分成五个不同的层次：

1. 磁盘块设备接口层：定义了以块大小为单位对磁盘块设备进行读写的 `trait` 接口
2. 块缓存层：在内存中缓存磁盘块的数据，避免频繁读写磁盘
3. 磁盘数据结构层：磁盘上的超级块、位图、索引节点、数据块、目录项等核心数据结构和相关处理

4. 磁盘块管理器层：合并了上述核心数据结构和磁盘布局所形成的磁盘文件系统数据结构，以及基于这些结构的创建/打开文件系统的相关处理和磁盘块的分配和回收处理
5. 索引节点层：管理索引节点（即文件控制块）数据结构，并实现文件创建/文件打开/文件读写等成员函数来向上支持文件操作相关的系统调用

大家也许觉得有五层架构的文件系统是一个很复杂的软件。其实，相对于面向 Qemu 模拟器的操作系统内核源码所占的 2400 行左右代码，它只有 900 行左右的代码，占总代码量的 27%。且由于其代码逻辑其实是一种自下而上的线性思维，属于传统的常规编程。相对于异常/中断/系统调用的特权级切换，进程管理中的进程上下文切换，内存管理中的页表地址映射等涉及异常控制流和硬件访问的非常规编程，文件系统的设计实现其实更容易理解。

### 7.3.3 块设备接口层

定义设备驱动需要实现的块读写接口 `BlockDevice` trait 的块设备接口层代码在 `block_dev.rs` 中。

在 `easy-fs` 库的最底层声明了一个块设备的抽象接口 `BlockDevice`：

```
// easy-fs/src/block_dev.rs

pub trait BlockDevice : Send + Sync + Any {
 fn read_block(&self, block_id: usize, buf: &mut [u8]);
 fn write_block(&self, block_id: usize, buf: &[u8]);
}
```

它需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，所以需要由具体的块设备驱动来实现这两个方法，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。`easy-fs` 库的块缓存层会调用这两个方法，进行块缓存的管理。这也体现了 `easy-fs` 的泛用性：它可以访问实现了 `BlockDevice` Trait 的块设备驱动程序。

---

#### 注解：块与扇区

实际上，块和扇区是两个不同的概念。扇区 (Sector) 是块设备随机读写的数据单位，通常每个扇区为 512 字节。而块是文件系统存储文件时的数据单位，每个块的大小等同于一个或多个扇区。之前提到过 Linux 的 Ext4 文件系统的单个块大小默认为 4096 字节。在我们的 `easy-fs` 实现中一个块和一个扇区同为 512 字节，因此在后面的讲解中我们不再区分扇区和块的概念。

---

### 7.3.4 块缓存层

实现磁盘块缓存功能的块缓存层的代码在 `block_cache.rs` 中。

由于操作系统频繁读写速度缓慢的磁盘块会极大降低系统性能，因此常见的手段是先通过 `read_block` 将一个块上的数据从磁盘读到内存中的一个缓冲区中，这个缓冲区中的内容是可以直接读写的，那么后续对这个数据块的大部分访问就可以在内存中完成了。如果缓冲区中的内容被修改了，那么后续还需要通过 `write_block` 将缓冲区中的内容写回到磁盘块中。

事实上，无论站在代码实现鲁棒性还是性能的角度，将这些缓冲区合理的管理起来都是很有必要的。一种完全不进行任何管理的模式可能是：每当要对一个磁盘块进行读写的时候，都通过 `read_block` 将块数据读取到一个临时创建的缓冲区，并在进行一些操作之后（可选地）将缓冲区的内容写回到磁盘块。从性能上考

虑，我们需要尽可能降低实际块读写（即 `read/write_block`）的次数，因为每一次调用它们都会产生大量开销。要做到这一点，关键就在于对块读写操作进行 **合并**。例如，如果一个块已经被读到缓冲区中了，那么我们就没有必要再读一遍，直接用已有的缓冲区就行了；同时，对于缓冲区中的同一个块的多次修改没有必要每次都写回磁盘，只需等所有的修改都结束之后统一写回磁盘即可。

当磁盘上的数据结构比较复杂的时候，很难通过应用来合理地规划块读取/写入的时机。这不仅可能涉及到复杂的参数传递，稍有不慎还有可能引入同步性问题（目前可以暂时忽略）：即一个块缓冲区修改后的内容在后续的同一个块读操作中不可见，这很致命但又难以调试。

因此，我们的做法是将缓冲区统一管理起来。当我们要读写一个块的时候，首先就是去全局管理器中查看这个块是否已被缓存到内存缓冲区中。如果是这样，则在一段连续时间内对于一个块进行的所有操作均是在同一个固定的缓冲区中进行的，这解决了同步性问题。此外，通过 `read/write_block` 进行块实际读写的时机完全交给块缓存层的全局管理器处理，上层子系统无需操心。全局管理器会尽可能将更多的块操作合并起来，并在必要的时机发起真正的块实际读写。

## 块缓存

块缓存 `BlockCache` 的定义如下：

```
// easy-fs/src/lib.rs

pub const BLOCK_SZ: usize = 512;

// easy-fs/src/block_cache.rs

pub struct BlockCache {
 cache: [u8; BLOCK_SZ],
 block_id: usize,
 block_device: Arc<dyn BlockDevice>,
 modified: bool,
}
```

其中：

- `cache` 是一个 512 字节的数组，表示位于内存中的缓冲区；
- `block_id` 记录了这个块缓存来自于磁盘中的块的编号；
- `block_device` 是一个底层块设备的引用，可通过它进行块读写；
- `modified` 记录这个块从磁盘载入内存缓存之后，它有没有被修改过。

当我们创建一个 `BlockCache` 的时候，这将触发一次 `read_block` 将一个块上的数据从磁盘读到缓冲区 `cache`：

```
// easy-fs/src/block_cache.rs

impl BlockCache {
 /// Load a new BlockCache from disk.
 pub fn new(
 block_id: usize,
 block_device: Arc<dyn BlockDevice>
) -> Self {
 let mut cache = [0u8; BLOCK_SZ];
 block_device.read_block(block_id, &mut cache);
 Self {
 cache,
 block_id,
```

(下页继续)

(续上页)

```
 block_device,
 modified: false,
 }
}
}
```

一旦磁盘块已经存在于内存缓存中，CPU 就可以直接访问磁盘块数据了：

```
// easy-fs/src/block_cache.rs

impl BlockCache {
 fn addr_of_offset(&self, offset: usize) -> usize {
 &self.cache[offset] as *const _ as usize
 }

 pub fn get_ref<T>(&self, offset: usize) -> &T where T: Sized {
 let type_size = core::mem::size_of::<T>();
 assert!(offset + type_size <= BLOCK_SZ);
 let addr = self.addr_of_offset(offset);
 unsafe { &*(addr as *const T) }
 }

 pub fn get_mut<T>(&mut self, offset: usize) -> &mut T where T: Sized {
 let type_size = core::mem::size_of::<T>();
 assert!(offset + type_size <= BLOCK_SZ);
 self.modified = true;
 let addr = self.addr_of_offset(offset);
 unsafe { &mut *(addr as *mut T) }
 }
}
```

- `addr_of_offset` 可以得到一个 `BlockCache` 内部的缓冲区中指定偏移量 `offset` 的字节地址；
  - `get_ref` 是一个泛型方法，它可以获取缓冲区中的位于偏移量 `offset` 的一个类型为 `T` 的磁盘上数据结构的不可变引用。该泛型方法的 Trait Bound 限制类型 `T` 必须是一个编译时已知大小的类型，我们通过 `core::mem::size_of::<T>()` 在编译时获取类型 `T` 的大小，并确认该数据结构被整个包含在磁盘块及其缓冲区之内。这里编译器会自动进行生命周期标注，约束返回的引用的生命周期不超过 `BlockCache` 自身，在使用的时候我们会保证这一点。
  - `get_mut` 与 `get_ref` 的不同之处在于，`get_mut` 会获取磁盘上数据结构的可变引用，由此可以对数据结构进行修改。由于这些数据结构目前位于内存中的缓冲区中，我们需要将 `BlockCache` 的 `modified` 标记为 `true` 表示该缓冲区已经被修改，之后需要将数据写回磁盘块才能真正将修改同步到磁盘。

BlockCache 的设计也体现了 RAII 思想，它管理着一个缓冲区的生命周期。当 BlockCache 的生命周期结束之后缓冲区也会被从内存中回收，这个时候 modified 标记将会决定数据是否需要写回磁盘：

```
// easy-fs/src/block_cache.rs

impl BlockCache {
 pub fn sync(&mut self) {
 if self.modified {
 self.modified = false;
 self.block_device.write_block(self.block_id, &self.cache);
 }
 }
}
```

(下页继续)

(续上页)

```
impl Drop for BlockCache {
 fn drop(&mut self) {
 self.sync()
 }
}
```

在 BlockCache 被 drop 的时候，它会首先调用 sync 方法，如果自身确实被修改过的话才会将缓冲区的内容写回磁盘。事实上，sync 并不是只有在 drop 的时候才会被调用。在 Linux 中，通常有一个后台进程负责定期将内存中缓冲区的内容写回磁盘。另外有一个 sys\_fsync 系统调用可以让应用主动通知内核将一个文件的修改同步回磁盘。由于我们的实现比较简单，sync 仅会在 BlockCache 被 drop 时才会被调用。

我们可以将 get\_ref/get\_mut 进一步封装为更为易用的形式：

```
// easy-fs/src/block_cache.rs

impl BlockCache {
 pub fn read<T, V>(&self, offset: usize, f: impl FnOnce(&T) -> V) -> V {
 f(self.get_ref(offset))
 }

 pub fn modify<T, V>(&mut self, offset: usize, f: impl FnOnce(&mut T) -> V) -> V {
 f(self.get_mut(offset))
 }
}
```

它们的含义是：在 BlockCache 缓冲区偏移量为 offset 的位置获取一个类型为 T 的磁盘上数据结构的不可变/可变引用（分别对应 read/modify），并让它执行传入的闭包 f 中所定义的操作。注意 read/modify 的返回值是和传入闭包的返回值相同的，因此相当于 read/modify 构成了传入闭包 f 的一层执行环境，让它能够绑定到一个缓冲区上执行。

这里我们传入闭包的类型为 FnOnce，这是因为闭包里面的变量被捕获的方式涵盖了不可变引用/可变引用/和 move 三种可能性，故而我们需要选取范围最广的 FnOnce。参数中的 impl 关键字体现了一种类似泛型的静态分发功能。

我们很快将展示 read/modify 接口如何在后续的开发中提供便利。

## 块缓存全局管理器

为了避免在块缓存上浪费过多内存，我们希望内存中同时只能驻留有限个磁盘块的缓冲区：

```
// easy-fs/src/block_cache.rs

const BLOCK_CACHE_SIZE: usize = 16;
```

块缓存全局管理器的功能是：当我们要对一个磁盘块进行读写时，首先看它是否已经被载入到内存缓存中了，如果已经被载入的话则直接返回，否则需要先读取磁盘块的数据到内存缓存中。此时，如果内存中驻留的磁盘块缓冲区的数量已满，则需要遵循某种缓存替换算法将某个块的缓存从内存中移除，再将刚刚读到的块数据加入到内存缓存中。我们这里使用一种类 FIFO 的简单缓存替换算法，因此在管理器中只需维护一个队列：

```
// easy-fs/src/block_cache.rs

use alloc::collections::VecDeque;

pub struct BlockCacheManager {
```

(下页继续)

(续上页)

```

 queue: VecDeque<(usize, Arc<Mutex<BlockCache>>)>,
}

impl BlockCacheManager {
 pub fn new() -> Self {
 Self { queue: VecDeque::new() }
 }
}

```

队列 queue 中管理的是块编号和块缓存的二元组。块编号的类型为 `usize`，而块缓存的类型则是一个 `Arc<Mutex<BlockCache>>`。这是一个此前频频提及到的 Rust 中的经典组合，它可以同时提供共享引用和互斥访问。这里的共享引用意义在于块缓存既需要在管理器 `BlockCacheManager` 保留一个引用，还需要以引用的形式返回给块缓存的请求者让它可以对块缓存进行访问。而互斥访问在单核上的意义在于提供内部可变性通过编译，在多核环境下则可以帮助我们避免可能的并发冲突。事实上，一般情况下我们需要在更上层提供保护措施避免两个线程同时对一个块缓存进行读写，因此这里只是比较谨慎的留下一层保险。

**警告:** Rust Pattern 卡片: `Arc<Mutex<?>>`

先看下 `Arc` 和 `Mutex` 的正确配合可以达到支持多线程安全读写数据对象。如果需要多线程共享所有权的数据对象，则只用 `Arc` 即可。如果需要修改 `T` 类型中某些成员变量 `member`，那直接采用 `Arc<Mutex<T>>`，并在修改的时候通过 `obj.lock().unwrap().member = xxx` 的方式是可行的，但这种编程模式的同步互斥的粒度太大，可能对互斥性能的影响比较大。为了减少互斥性能开销，其实只需要在 `T` 类型中的需要被修改的成员变量上加 `Mutex<_>` 即可。如果成员变量也是一个数据结构，还包含更深层次的成员变量，那应该继续下推到最终需要修改的成员变量上去添加 `Mutex`。

`get_block_cache` 方法尝试从块缓存管理器中获取一个编号为 `block_id` 的块的块缓存，如果找不到，会从磁盘读取到内存中，还有可能会发生缓存替换：

```

1 // easy-fs/src/block_cache.rs
2
3 impl BlockCacheManager {
4 pub fn get_block_cache(
5 &mut self,
6 block_id: usize,
7 block_device: Arc<dyn BlockDevice>,
8) -> Arc<Mutex<BlockCache>> {
9 if let Some(pair) = self.queue
10 .iter()
11 .find(|pair| pair.0 == block_id) {
12 Arc::clone(&pair.1)
13 } else {
14 // substitute
15 if self.queue.len() == BLOCK_CACHE_SIZE {
16 // from front to tail
17 if let Some((idx, _)) = self.queue
18 .iter()
19 .enumerate()
20 .find(|(_, pair)| Arc::strong_count(&pair.1) == 1) {
21 self.queue.drain(idx..=idx);
22 } else {
23 panic!("Run out of BlockCache!");
24 }
25 }
26 // load block into mem and push back

```

(下页继续)

(续上页)

```

27 let block_cache = Arc::new(Mutex::new(
28 BlockCache::new(block_id, Arc::clone(&block_device)))
29)));
30 self.queue.push_back((block_id, Arc::clone(&block_cache)));
31 block_cache
32 }
33 }
34 }
```

- 第 9 行会遍历整个队列试图找到一个编号相同的块缓存，如果找到了，会将块缓存管理器中保存的块缓存的引用复制一份并返回；
- 第 13 行对应找不到的情况，此时必须将块从磁盘读入内存中的缓冲区。在实际读取之前，需要判断管理器保存的块缓存数量是否已经达到了上限。如果达到了上限（第 15 行）才需要执行缓存替换算法，丢掉某个块缓存并空出一个空位。这里使用一种类 FIFO 算法：每加入一个块缓存时要从队尾加入；要替换时则从队头弹出。但此时队头对应的块缓存可能仍在使用：判断的标志是其强引用计数  $\geq 2$ ，即除了块缓存管理器保留的一份副本之外，在外面还有若干份副本正在使用。因此，我们的做法是从队头遍历到队尾找到第一个强引用计数恰好为 1 的块缓存并将其替换出去。

那么是否有可能出现队列已满且其中所有的块缓存都正在使用的情形呢？事实上，只要我们的上限 BLOCK\_CACHE\_SIZE 设置的足够大，超过所有应用同时访问的块总数上限，那么这种情况永远不会发生。但是，如果我们的上限设置不足，内核将 panic（基于简单内核设计的思路）。

- 第 27 行开始我们创建一个新的块缓存（会触发 read\_block 进行块读取）并加入到队尾，最后返回给请求者。

接下来需要创建 BlockCacheManager 的全局实例：

```

// easy-fs/src/block_cache.rs

lazy_static! {
 pub static ref BLOCK_CACHE_MANAGER: Mutex<BlockCacheManager> = Mutex::new(
 BlockCacheManager::new()
);
}

pub fn get_block_cache(
 block_id: usize,
 block_device: Arc<dyn BlockDevice>
) -> Arc<Mutex<BlockCache>> {
 BLOCK_CACHE_MANAGER.lock().get_block_cache(block_id, block_device)
}
```

这样对于其他模块而言，就可以直接通过 get\_block\_cache 方法来请求块缓存了。这里需要指出的是，它返回的是一个 `Arc<Mutex<BlockCache>>`，调用者需要通过 `.lock()` 获取里层互斥锁 `Mutex` 才能对最里面的 `BlockCache` 进行操作，比如通过 `read/modify` 访问缓冲区里面的磁盘数据结构。

### 7.3.5 磁盘布局及磁盘上数据结构

磁盘数据结构层的代码在 `layout.rs` 和 `bitmap.rs` 中。

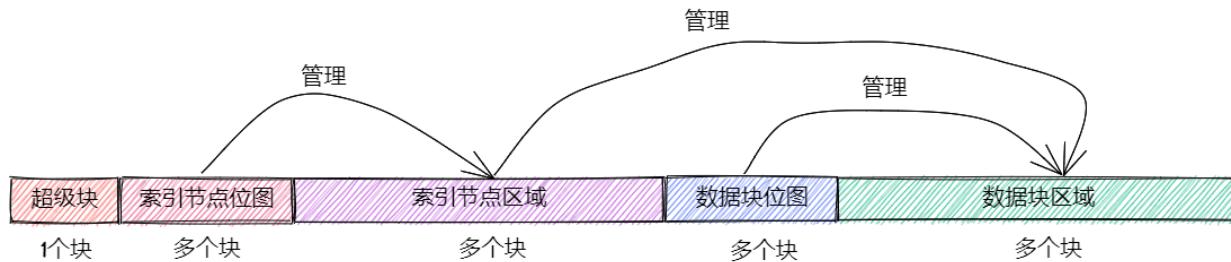
对于一个文件系统而言，最重要的功能是如何将一个逻辑上的文件目录树结构映射到磁盘上，决定磁盘上的每个块应该存储文件相关的哪些数据。为了更容易进行管理和更新，我们需要将磁盘上的数据组织为若干种不同的磁盘上数据结构，并合理安排它们在磁盘中的位置。

#### easy-fs 磁盘布局概述

在 easy-fs 磁盘布局中，按照块编号从小到大顺序地分成 5 个不同属性的连续区域：

- 最开始的区域的长度为一个块，其内容是 easy-fs **超级块** (Super Block)。超级块内以魔数的形式提供了文件系统合法性检查功能，同时还可以定位其他连续区域的位置。
- 第二个区域是一个索引节点位图，长度为若干个块。它记录了后面的索引节点区域中有哪些索引节点已经被分配出去使用了，而哪些还尚未被分配出去。
- 第三个区域是索引节点区域，长度为若干个块。其中的每个块都存储了若干个索引节点。
- 第四个区域是一个数据块位图，长度为若干个块。它记录了后面的数据块区域中有哪些数据块已经被分配出去使用了，而哪些还尚未被分配出去。
- 最后的区域则是数据块区域，顾名思义，其中的每一个已经分配出去的块保存了文件或目录中的具体数据内容。

easy-fs 的磁盘布局如下图所示：



**索引节点 (Inode, Index Node)** 是文件系统中的一种重要数据结构。逻辑目录树结构中的每个文件和目录都对应一个 inode，我们前面提到的文件系统实现中，文件/目录的底层编号实际上就是指 inode 编号。在 inode 中不仅包含了我们通过 `stat` 工具能够看到的文件/目录的元数据（大小/访问权限/类型等信息），还包含实际保存对应文件/目录数据的数据块（位于最后的数据块区域中）的索引信息，从而能够找到文件/目录的数据被保存在磁盘的哪些块中。从索引方式上看，同时支持直接索引和间接索引。

每个区域中均存储着不同的磁盘数据结构，easy-fs 文件系统能够对磁盘中的数据进行解释并将其结构化。下面我们分别对它们进行介绍。

#### easy-fs 超级块

超级块 `SuperBlock` 的内容如下：

```
// easy-fs/src/layout.rs

#[repr(C)]
pub struct SuperBlock {
 magic: u32,
 pub total_blocks: u32,
```

(下页继续)

(续上页)

```

pub inode_bitmap_blocks: u32,
pub inode_area_blocks: u32,
pub data_bitmap_blocks: u32,
pub data_area_blocks: u32,
}

```

其中, `magic` 是一个用于文件系统合法性验证的魔数, `total_blocks` 给出文件系统的总块数。注意这并不等同于所在磁盘的总块数, 因为文件系统很可能并没有占据整个磁盘。后面的四个字段则分别给出 easy-fs 布局中后四个连续区域的长度各为多少个块。

下面是它实现的方法:

```

// easy-fs/src/layout.rs

impl SuperBlock {
 pub fn initialize(
 &mut self,
 total_blocks: u32,
 inode_bitmap_blocks: u32,
 inode_area_blocks: u32,
 data_bitmap_blocks: u32,
 data_area_blocks: u32,
) {
 *self = Self {
 magic: EFS_MAGIC,
 total_blocks,
 inode_bitmap_blocks,
 inode_area_blocks,
 data_bitmap_blocks,
 data_area_blocks,
 }
 }
 pub fn is_valid(&self) -> bool {
 self.magic == EFS_MAGIC
 }
}

```

- `initialize` 可以在创建一个 easy-fs 的时候对超级块进行初始化, 注意各个区域的块数是以参数的形式传入进来的, 它们的划分是更上层的磁盘块管理器需要完成的工作。
- `is_valid` 则可以通过魔数判断超级块所在的文件系统是否合法。

`SuperBlock` 是一个磁盘上数据结构, 它就存放在磁盘上编号为 0 的块的起始处。

## 位图

在 easy-fs 布局中存在两类不同的位图, 分别对索引节点和数据块进行管理。每个位图都由若干个块组成, 每个块大小为 512 bytes, 即 4096 bits。每个 bit 都代表一个索引节点/数据块的分配状态, 0 意味着未分配, 而 1 则意味着已经分配出去。位图所要做的事情是通过基于 bit 为单位的分配 (寻找一个为 0 的 bit 位并设置为 1) 和回收 (将 bit 位清零) 来进行索引节点/数据块的分配和回收。

```

// easy-fs/src/bitmap.rs

pub struct Bitmap {
 start_block_id: usize,
 blocks: usize,
}

```

(下页继续)

(续上页)

```
}

impl Bitmap {
 pub fn new(start_block_id: usize, blocks: usize) -> Self {
 Self {
 start_block_id,
 blocks,
 }
 }
}
```

位图 Bitmap 中仅保存了它所在区域的起始块编号以及区域的长度为多少个块。通过 new 方法可以新建一个位图。注意 Bitmap 自身是驻留在内存中的，但是它能够表示索引节点/数据块区域中的那些磁盘块的分配情况。磁盘块上位图区域的数据则是要以磁盘数据结构 BitmapBlock 的格式进行操作：

```
// easy-fs/src(bitmap.rs
```

BitmapBlock 是一个磁盘数据结构，它将位图区域中的一个磁盘块解释为长度为 64 的一个 u64 数组，每个 u64 打包了一组 64 bits，于是整个数组包含  $64 \times 64 = 4096$  bits，且可以以组为单位进行操作。

首先来看 Bitmap 如何分配一个 bit:

```
1 // easy-fs/src(bitmap.rs
2
3 const BLOCK_BITS: usize = BLOCK_SZ * 8;
4
5 impl Bitmap {
6 pub fn alloc(&self, block_device: &Arc<dyn BlockDevice>) -> Option<usize> {
7 for block_id in 0..self.blocks {
8 let pos = get_block_cache(
9 block_id + self.start_block_id as usize,
10 Arc::clone(block_device),
11)
12 .lock()
13 .modify(0, |bitmap_block: &mut BitmapBlock| {
14 if let Some((bits64_pos, inner_pos)) = bitmap_block
15 .iter()
16 .enumerate()
17 .find(|(_, bits64)| **bits64 != u64::MAX)
18 .map(|(bits64_pos, bits64)| {
19 (bits64_pos, bits64.trailing_ones() as usize)
20 })
21 // modify cache
22 bitmap_block[bits64_pos] |= 1u64 << inner_pos;
23 Some(block_id * BLOCK_BITS + bits64_pos * 64 + inner_pos as usize)
24 } else {
25 None
26 }
27 });
28 if pos.is_some() {
29 return pos;
30 }
31 }
32 None
33 }
34}
```

(下页继续)

(续上页)

34

其主要思路是遍历区域中的每个块，再在每个块中以 bit 组（每组 64 bits）为单位进行遍历，找到一个尚未被全部分配出去的组，最后在里面分配一个 bit。它将会返回分配的 bit 所在的位置，等同于索引节点/数据块的编号。如果所有 bit 均已经被分配出去了，则返回 None。

第 7 行枚举区域中的每个块（编号为 block\_id），在循环内部我们需要读写这个块，在块内尝试找到一个空闲的 bit 并置 1。一旦涉及到块的读写，就需要用到块缓存层提供的接口：

- 第 8 行我们调用 get\_block\_cache 获取块缓存，注意我们传入的块编号是区域起始块编号 start\_block\_id 加上区域内的块编号 block\_id 得到的块设备上的块编号。
- 第 12 行我们通过 .lock() 获取块缓存的互斥锁从而可以对块缓存进行访问。
- 第 13 行我们使用到了 BlockCache::modify 接口。它传入的偏移量 offset 为 0，这是因为整个块上只有一个 BitmapBlock，它的大小恰好为 512 字节。因此我们需要从块的开头开始才能访问到完整的 BitmapBlock。同时，传给它的闭包需要显式声明参数类型为 &mut BitmapBlock，不然的话，BlockCache 的泛型方法 modify/get\_mut 无法得知应该用哪个类型来解析块上的数据。在声明之后，编译器才能在这里将两个方法中的泛型 T 实例化为具体类型 BitmapBlock。

总结一下，这里 modify 的含义就是：从缓冲区偏移量为 0 的位置开始将一段连续的数据（数据的长度随具体类型而定）解析为一个 BitmapBlock 并要对该数据结构进行修改。在闭包内部，我们可以使用这个 BitmapBlock 的可变引用 bitmap\_block 对它进行访问。read/get\_ref 的用法完全相同，后面将不再赘述。

- 闭包的主体位于第 14~26 行。它尝试在 bitmap\_block 中找到一个空闲的 bit 并返回其位置，如果不存在的话则返回 None。它的思路是，遍历每 64 bits 构成的组（一个 u64），如果它并没有达到 u64::MAX（即  $2^{64} - 1$ ），则通过 u64::trailing\_ones 找到最低的一个 0 并置为 1。如果能够找到的话，bit 组的编号将保存在变量 bits64\_pos 中，而分配的 bit 在组内的位置将保存在变量 inner\_pos 中。在返回分配的 bit 编号的时候，它的计算方式是 block\_id \* BLOCK\_BITS + bits64\_pos \* 64 + inner\_pos。注意闭包中的 block\_id 并不在闭包的参数列表中，因此它是从外部环境（即自增 block\_id 的循环）中捕获到的。

我们一旦在某个块中找到一个空闲的 bit 并成功分配，就不再考虑后续的块。第 28 行体现了提前返回的思路。

### 警告：Rust 语法卡片：闭包

闭包是持有外部环境变量的函数。所谓外部环境，就是指创建闭包时所在的词法作用域。Rust 中定义的闭包，按照对外部环境变量的使用方式（借用、复制、转移所有权），分为三个类型：Fn、FnMut、FnOnce。Fn 类型的闭包会在闭包内部以共享借用的方式使用环境变量；FnMut 类型的闭包会在闭包内部以独占借用的方式使用环境变量；而 FnOnce 类型的闭包会在闭包内部以所有者的身份使用环境变量。由此可见，根据闭包内使用环境变量的方式，即可判断创建出来的闭包的类型。

接下来看 Bitmap 如何回收一个 bit：

```
// easy-fs/src/bitmap.rs

/// Return (block_pos, bits64_pos, inner_pos)
fn decomposition(mut bit: usize) -> (usize, usize, usize) {
 let block_pos = bit / BLOCK_BITS;
 bit = bit % BLOCK_BITS;
 (block_pos, bit / 64, bit % 64)
}

impl Bitmap {
 pub fn deallocate(&self, block_device: &Arc<dyn BlockDevice>, bit: usize) {
```

(下页继续)

(续上页)

```
 let (block_pos, bits64_pos, inner_pos) = decomposition(bit);
 get_block_cache(
 block_pos + self.start_block_id,
 Arc::clone(block_device)
).lock().modify(0, |bitmap_block: &mut BitmapBlock| {
 assert!(bitmap_block[bits64_pos] & (1u64 << inner_pos) > 0);
 bitmap_block[bits64_pos] -= 1u64 << inner_pos;
 });
}
```

`dealloc` 方法首先调用 `decomposition` 函数将 bit 编号 `bit` 分解为区域中的块编号 `block_pos`、块内的组编号 `bits64_pos` 以及组内编号 `inner_pos` 的三元组，这样就能精确定位待回收的 bit，随后将其清零即可。

## 磁盘上索引节点

在磁盘上的索引节点区域，每个块上都保存着若干个索引节点 DiskInode：

```
// easy-fs/src/layout.rs

const INODE_DIRECT_COUNT: usize = 28;

#[repr(C)]
pub struct DiskInode {
 pub size: u32,
 pub direct: [u32; INODE_DIRECT_COUNT],
 pub indirect1: u32,
 pub indirect2: u32,
 pub type_: DiskInodeType,
}

#[derive(PartialEq)]
pub enum DiskInodeType {
 File,
 Directory,
}
```

每个文件/目录在磁盘上均以一个 DiskInode 的形式存储。其中包含文件/目录的元数据: size 表示文件/目录内容的字节数, type\_ 表示索引节点的类型 DiskInodeType , 目前仅支持文件 File 和目录 Directory 两种类型。其余的 direct/indirect1/indirect2 都是存储文件内容/目录内容的数据块的索引, 这也是索引节点名字的由来。

为了尽可能节约空间，在进行索引的时候，块的编号用一个 `u32` 存储。索引方式分成直接索引和间接索引两种：

- 当文件很小的时候，只需用到直接索引，`direct` 数组中最多可以指向 `INODE_DIRECT_COUNT` 个数据块，当取值为 28 的时候，通过直接索引可以找到 14KiB 的内容。
  - 当文件比较大的时候，不仅直接索引的 `direct` 数组装满，还需要用到一级间接索引 `indirect1`。它指向一个一级索引块，这个块也位于磁盘布局的数据块区域中。这个一级索引块中的每个 `u32` 都用来指向数据块区域中一个保存该文件内容的数据块，因此，最多能够索引  $\frac{512}{4} = 128$  个数据块，对应 64KiB 的内容。
  - 当文件大小超过直接索引和一级索引支持的容量上限 78KiB 的时候，就需要用到二级间接索引 `indirect2`。它指向一个位于数据块区域中的二级索引块。二级索引块中的每个 `u32` 指向一个不

同的一级索引块，这些一级索引块也位于数据块区域中。因此，通过二级间接索引最多能够索引  $128 \times 64\text{KiB} = 8\text{MiB}$  的内容。

为了充分利用空间，我们将 DiskInode 的大小设置为 128 字节，每个块正好能够容纳 4 个 DiskInode。在后续需要支持更多类型的元数据的时候，可以适当缩减直接索引 direct 的块数，并将节约出来的空间用来存放其他元数据，仍可保证 DiskInode 的总大小为 128 字节。

通过 initialize 方法可以初始化一个 DiskInode 为一个文件或目录：

```
// easy-fs/src/layout.rs

impl DiskInode {
 /// indirect1 and indirect2 block are allocated only when they are needed.
 pub fn initialize(&mut self, type_: DiskInodeType) {
 self.size = 0;
 self.direct.iter_mut().for_each(|v| *v = 0);
 self.indirect1 = 0;
 self.indirect2 = 0;
 self.type_ = type_;
 }
}
```

需要注意的是，indirect1/2 均被初始化为 0。因为最开始文件内容的大小为 0 字节，并不会用到一级/二级索引。为了节约空间，内核会按需分配一级/二级索引块。此外，直接索引 direct 也被清零。

is\_file 和 is\_dir 两个方法可以用来确认 DiskInode 的类型为文件还是目录：

```
// easy-fs/src/layout.rs

impl DiskInode {
 pub fn is_dir(&self) -> bool {
 self.type_ == DiskInodeType::Directory
 }
 pub fn is_file(&self) -> bool {
 self.type_ == DiskInodeType::File
 }
}
```

get\_block\_id 方法体现了 DiskInode 最重要的数据块索引功能，它可以从索引中查到它自身用于保存文件内容的第 block\_id 个数据块的块编号，这样后续才能对这个数据块进行访问：

```
1 // easy-fs/src/layout.rs
2
3 const INODE_INDIRECT1_COUNT: usize = BLOCK_SZ / 4;
4 const INDIRECT1_BOUND: usize = DIRECT_BOUND + INODE_INDIRECT1_COUNT;
5 type IndirectBlock = [u32; BLOCK_SZ / 4];
6
7 impl DiskInode {
8 pub fn get_block_id(&self, inner_id: u32, block_device: &Arc<dyn BlockDevice>) -> u32 {
9 let inner_id = inner_id as usize;
10 if inner_id < INODE_DIRECT_COUNT {
11 self.direct[inner_id]
12 } else if inner_id < INDIRECT1_BOUND {
13 get_block_cache(self.indirect1 as usize, Arc::clone(block_device))
14 .lock()
15 .read(0, |indirect_block: &IndirectBlock| {
16 indirect_block[inner_id - INODE_DIRECT_COUNT]
17 })
18 }
19}
```

(下页继续)

(续上页)

```

18 } else {
19 let last = inner_id - INDIRECT1_BOUND;
20 let indirect1 = get_block_cache(
21 self.indirect2 as usize,
22 Arc::clone(block_device)
23)
24 .lock()
25 .read(0, |indirect2: &IndirectBlock| {
26 indirect2[last / INODE_INDIRECT1_COUNT]
27 });
28 get_block_cache(
29 indirect1 as usize,
30 Arc::clone(block_device)
31)
32 .lock()
33 .read(0, |indirect1: &IndirectBlock| {
34 indirect1[last % INODE_INDIRECT1_COUNT]
35 })
36 }
37 }
38 }
```

这里需要说明的是：

- 第 10/12/18 行分别利用直接索引/一级索引和二级索引，具体选用哪种索引方式取决于 block\_id 所在的区间。
- 在对一个索引块进行操作的时候，我们将其解析为磁盘数据结构 IndirectBlock，实质上就是一个 u32 数组，每个都指向一个下一级索引块或者数据块。
- 对于二级索引的情况，需要先查二级索引块找到挂在它下面的一级索引块，再通过一级索引块找到数据块。

在对文件/目录初始化之后，它的 size 均为 0，此时并不会索引到任何数据块。它需要通过 increase\_size 方法逐步扩充容量。在扩充的时候，自然需要一些新的数据块来作为索引块或是保存内容的数据块。我们需要先编写一些辅助方法来确定在容量扩充的时候额外需要多少块：

```

// easy-fs/src/layout.rs

impl DiskInode {
 /// Return block number correspond to size.
 pub fn data_blocks(&self) -> u32 {
 Self::_data_blocks(self.size)
 }
 fn _data_blocks(size: u32) -> u32 {
 (size + BLOCK_SZ as u32 - 1) / BLOCK_SZ as u32
 }
 /// Return number of blocks needed include indirect1/2.
 pub fn total_blocks(size: u32) -> u32 {
 let data_blocks = Self::_data_blocks(size) as usize;
 let mut total = data_blocks as usize;
 // indirect1
 if data_blocks > INODE_DIRECT_COUNT {
 total += 1;
 }
 // indirect2
 if data_blocks > INDIRECT1_BOUND {
```

(下页继续)

(续上页)

```

 total += 1;
 // sub indirect1
 total += (data_blocks - INDIRECT1_BOUND + INODE_INDIRECT1_COUNT - 1) /_
 ↪INODE_INDIRECT1_COUNT;
 }
 total as u32
}
pub fn blocks_num_needed(&self, new_size: u32) -> u32 {
 assert!(new_size >= self.size);
 Self::total_blocks(new_size) - Self::total_blocks(self.size)
}
}

```

data\_blocks 方法可以计算为了容纳自身 size 字节的内容需要多少个数据块。计算的过程只需用 size 除以每个块的大小 BLOCK\_SZ 并向上取整。而 total\_blocks 不仅包含数据块，还需要统计索引块。计算的方法也很简单，先调用 data\_blocks 得到需要多少数据块，再根据数据块数目所处的区间统计索引块即可。blocks\_num\_needed 可以计算将一个 DiskInode 的 size 扩容到 new\_size 需要额外多少个数据和索引块。这只需要调用两次 total\_blocks 作差即可。

下面给出 increase\_size 方法的接口：

```

// easy-fs/src/layout.rs

impl DiskInode {
 pub fn increase_size(
 &mut self,
 new_size: u32,
 new_blocks: Vec<u32>,
 block_device: &Arc<dyn BlockDevice>,
);
}

```

其中 new\_size 表示容量扩充之后的文件大小；new\_blocks 是一个保存了本次容量扩充所需块编号的向量，这些块都是由上层的磁盘块管理器负责分配的。increase\_size 的实现有些复杂，在这里不详细介绍。大致的思路是按照直接索引、一级索引再到二级索引的顺序进行扩充。

有些时候我们还需要清空文件的内容并回收所有数据和索引块。这是通过 clear\_size 方法来实现的：

```

// easy-fs/src/layout.rs

impl DiskInode {
 /// Clear size to zero and return blocks that should be deallocated.
 ///
 /// We will clear the block contents to zero later.
 pub fn clear_size(&mut self, block_device: &Arc<dyn BlockDevice>) -> Vec<u32>;
}

```

它会将回收的所有块的编号保存在一个向量中返回给磁盘块管理器。它的实现原理和 increase\_size 一样也分为多个阶段，在这里不展开。

接下来需要考虑通过 DiskInode 来读写它索引的那些数据块中的数据。这些数据可以被视为一个字节序列，而每次都是选取其中的一段连续区间进行操作，以 read\_at 为例：

```

1 // easy-fs/src/layout.rs
2
3 type DataBlock = [u8; BLOCK_SZ];
4

```

(下页继续)

(续上页)

```

5 impl DiskInode {
6 pub fn read_at(
7 &self,
8 offset: usize,
9 buf: &mut [u8],
10 block_device: &Arc<dyn BlockDevice>,
11) -> usize {
12 let mut start = offset;
13 let end = (offset + buf.len()).min(self.size as usize);
14 if start >= end {
15 return 0;
16 }
17 let mut start_block = start / BLOCK_SZ;
18 let mut read_size = 0usize;
19 loop {
20 // calculate end of current block
21 let mut end_current_block = (start / BLOCK_SZ + 1) * BLOCK_SZ;
22 end_current_block = end_current_block.min(end);
23 // read and update read size
24 let block_read_size = end_current_block - start;
25 let dst = &mut buf[read_size..read_size + block_read_size];
26 get_block_cache(
27 self.get_block_id(start_block as u32, block_device) as usize,
28 Arc::clone(block_device),
29)
30 .lock()
31 .read(0, |data_block: &DataBlock| {
32 let src = &data_block[start % BLOCK_SZ..start % BLOCK_SZ + block_read_
33 -size];
34 dst.copy_from_slice(src);
35 });
36 read_size += block_read_size;
37 // move to next block
38 if end_current_block == end { break; }
39 start_block += 1;
40 start = end_current_block;
41 }
42 read_size
43 }
}

```

它的含义是：将文件内容从 offset 字节开始的部分读到内存中的缓冲区 buf 中，并返回实际读到的字节数。如果文件剩下的内容还足够多，那么缓冲区会被填满；否则文件剩下的全部内容都会被读到缓冲区中。具体实现上有很多细节，但大致的思路是遍历位于字节区间 start, end 之间的那些块，将它们视为一个 DataBlock (也就是一个字节数组)，并将其中的部分内容复制到缓冲区 buf 中适当的区域。start\_block 维护着目前是文件内部第多少个数据块，需要首先调用 get\_block\_id 从索引中查到这个数据块在块设备中的块编号，随后才能传入 get\_block\_cache 中将正确的数据块缓存到内存中进行访问。

在第 14 行进行了简单的边界条件判断，如果要读取的内容超出了文件的范围，那么直接返回 0，表示读取不到任何内容。

write\_at 的实现思路基本上和 read\_at 完全相同。但不同的是 write\_at 不会出现失败的情况；只要 Inode 管理的数据块的大小足够，传入的整个缓冲区的数据都必定会被写入到文件中。当从 offset 开始的区间超出了文件范围的时候，就需要调用者在调用 write\_at 之前提前调用 increase\_size，将文件大小扩充到区间的右端，保证写入的完整性。

## 数据块与目录项

作为一个文件而言，它的内容在文件系统看来没有任何既定的格式，都只是一个字节序列。因此每个保存内容的数据块都只是一个字节数组：

```
// easy-fs/src/layout.rs

type DataBlock = [u8; BLOCK_SZ];
```

然而，目录的内容却需要遵从一种特殊的格式。在我们的实现中，它可以看成一个目录项的序列，每个目录项都是一个二元组，二元组的首个元素是目录下面的一个文件（或子目录）的文件名（或目录名），另一个元素则是文件（或子目录）所在的索引节点编号。目录项相当于目录树结构上的子树节点，我们需要通过它来一级一级的找到实际要访问的文件或目录。目录项 DirEntry 的定义如下：

```
// easy-fs/src/layout.rs

const NAME_LENGTH_LIMIT: usize = 27;

#[repr(C)]
pub struct DirEntry {
 name: [u8; NAME_LENGTH_LIMIT + 1],
 inode_number: u32,
}

pub const DIRENT_SZ: usize = 32;
```

目录项 Dirent 最大允许保存长度为 27 的文件/目录名（数组 name 中最末的一个字节留给 \0），且它自身占据空间 32 字节，每个数据块可以存储 16 个目录项。我们可以通过 empty 和 new 分别生成一个空的目录项或是一个合法的目录项：

```
// easy-fs/src/layout.rs

impl DirEntry {
 pub fn empty() -> Self {
 Self {
 name: [0u8; NAME_LENGTH_LIMIT + 1],
 inode_number: 0,
 }
 }

 pub fn new(name: &str, inode_number: u32) -> Self {
 let mut bytes = [0u8; NAME_LENGTH_LIMIT + 1];
 &mut bytes[..name.len()].copy_from_slice(name.as_bytes());
 Self {
 name: bytes,
 inode_number,
 }
 }
}
```

在从目录的内容中读取目录项或者是将目录项写入目录的时候，我们需要将目录项转化为缓冲区（即字节切片）的形式来符合索引节点 Inode 数据结构中的 read\_at 或 write\_at 方法接口的要求：

```
// easy-fs/src/layout.rs

impl DirEntry {
 pub fn as_bytes(&self) -> &[u8] {
 unsafe {
```

(下页继续)

(续上页)

```

 core::slice::from_raw_parts(
 self as *const _ as usize as *const u8,
 DIRENT_SZ,
)
 }
}
pub fn as_bytes_mut(&mut self) -> &mut [u8] {
 unsafe {
 core::slice::from_raw_parts_mut(
 self as *mut _ as usize as *mut u8,
 DIRENT_SZ,
)
 }
}
}
}

```

此外，通过 name 和 inode\_number 方法可以取出目录项中的内容：

```

// easy-fs/src/layout.rs

impl DirEntry {
 pub fn name(&self) -> &str {
 let len = (0usize..).find(|i| self.name[*i] == 0).unwrap();
 core::str::from_utf8(&self.name[..len]).unwrap()
 }
 pub fn inode_number(&self) -> u32 {
 self.inode_number
 }
}

```

### 7.3.6 磁盘块管理器

本层的代码在 efs.rs 中。上面介绍了 easy-fs 的磁盘布局设计以及数据的组织方式—即各类磁盘数据结构。但是它们都是以比较零散的形式分开介绍的，并没有体现出磁盘布局上各个区域是如何划分的。实现 easy-fs 的整体磁盘布局，将各段区域及上面的磁盘数据结构整合起来就是简易文件系统 EasyFileSystem 的职责。它知道每个布局区域所在的位置，磁盘块的分配和回收也需要经过它才能完成，因此某种意义上讲它还可以看成一个磁盘块管理器。

注意从这一层开始，所有的数据结构就都放在内存上了。

```

// easy-fs/src/efs.rs

pub struct EasyFileSystem {
 pub block_device: Arc<dyn BlockDevice>,
 pub inode_bitmap: Bitmap,
 pub data_bitmap: Bitmap,
 pub inode_area_start_block: u32,
 pub data_area_start_block: u32,
}

```

EasyFileSystem 包含索引节点和数据块的两个位图 inode\_bitmap 和 data\_bitmap，还记录下索引节点区域和数据块区域起始块编号方便确定每个索引节点和数据块在磁盘上的具体位置。我们还要在其中保留块设备的一个指针 block\_device，在进行后续操作的时候，该指针会被拷贝并传递给下层的数据结构，让它们也能够直接访问块设备。

通过 create 方法可以在块设备上创建并初始化一个 easy-fs 文件系统：

```

1 // easy-fs/src/efs.rs
2
3 impl EasyFileSystem {
4 pub fn create(
5 block_device: Arc<dyn BlockDevice>,
6 total_blocks: u32,
7 inode_bitmap_blocks: u32,
8) -> Arc<Mutex<Self>> {
9 // calculate block size of areas & create bitmaps
10 let inode_bitmap = Bitmap::new(1, inode_bitmap_blocks as usize);
11 let inode_num = inode_bitmap.maximum();
12 let inode_area_blocks =
13 ((inode_num * core::mem::size_of::<DiskInode>() + BLOCK_SZ - 1) / BLOCK_
14 ↵SZ) as u32;
15 let inode_total_blocks = inode_bitmap_blocks + inode_area_blocks;
16 let data_total_blocks = total_blocks - 1 - inode_total_blocks;
17 let data_bitmap_blocks = (data_total_blocks + 4096) / 4097;
18 let data_area_blocks = data_total_blocks - data_bitmap_blocks;
19 let data_bitmap = Bitmap::new(
20 (1 + inode_bitmap_blocks + inode_area_blocks) as usize,
21 data_bitmap_blocks as usize,
22);
23 let mut efs = Self {
24 block_device: Arc::clone(&block_device),
25 inode_bitmap,
26 data_bitmap,
27 inode_area_start_block: 1 + inode_bitmap_blocks,
28 data_area_start_block: 1 + inode_total_blocks + data_bitmap_blocks,
29 };
30 // clear all blocks
31 for i in 0..total_blocks {
32 get_block_cache(
33 i as usize,
34 Arc::clone(&block_device)
35)
36 .lock()
37 .modify(0, |data_block: &mut DataBlock| {
38 for byte in data_block.iter_mut() { *byte = 0; }
39 });
40 }
41 // initialize SuperBlock
42 get_block_cache(0, Arc::clone(&block_device))
43 .lock()
44 .modify(0, |super_block: &mut SuperBlock| {
45 super_block.initialize(
46 total_blocks,
47 inode_bitmap_blocks,
48 inode_area_blocks,
49 data_bitmap_blocks,
50 data_area_blocks,
51);
52 });
53 // write back immediately
54 // create a inode for root node "/"
55 assert_eq!(efs.alloc_inode(), 0);
56 let (root_inode_block_id, root_inode_offset) = efs.get_disk_inode_pos(0);
57 get_block_cache(

```

(下页继续)

(续上页)

```

57 root_inode_block_id as usize,
58 Arc::clone(&block_device)
59)
60 .lock()
61 .modify(root_inode_offset, |disk_inode: &mut Disk_inode| {
62 disk_inode.initialize(Disk_inode_type::Directory);
63 });
64 Arc::new(Mutex::new(efs))
65 }
66 }

```

- 第 10~21 行根据传入的参数计算每个区域各应该包含多少块。根据 inode 位图的大小计算 inode 区域至少需要多少个块才能够使得 inode 位图中的每个 bit 都能够有一个实际的 inode 可以对应，这样就确定了 inode 位图区域和 inode 区域的大小。剩下的块都分配给数据块位图区域和数据块区域。我们希望数据块位图中的每个 bit 仍然能够对应到一个数据块，但是数据块位图又不能过小，不然会造成某些数据块永远不会被使用。因此数据块位图区域最合理的大小是剩余的块数除以 4097 再上取整，因为位图中的每个块能够对应 4096 个数据块。其余的块就都作为数据块使用。
- 第 22 行创建 EasyFileSystem 实例 efs。
- 第 30 行首先将块设备的前 total\_blocks 个块清零，因为 easy-fs 要用到它们，这也是为初始化做准备。
- 第 41 行将位于块设备编号为 0 块上的超级块进行初始化，只需传入之前计算得到的每个区域的块数就行了。
- 第 54~63 行创建根目录 /。首先需要调用 alloc\_inode 在 inode 位图中分配一个 inode，由于这是第一次分配，它的编号固定是 0。接下来需要将分配到的 inode 初始化为 easy-fs 中的唯一一个目录，故需要调用 get\_disk\_inode\_pos 来根据 inode 编号获取该 inode 所在的块的编号以及块内偏移，之后就可以将它们传给 get\_block\_cache 和 modify 了。

通过 open 方法可以从一个已写入了 easy-fs 镜像的块设备上打开我们的 easy-fs：

```

// easy-fs/src/efs.rs

impl EasyFileSystem {
 pub fn open(block_device: Arc<dyn BlockDevice>) -> Arc<Mutex<Self>> {
 // read SuperBlock
 get_block_cache(0, Arc::clone(&block_device))
 .lock()
 .read(0, &super_block: &SuperBlock) {
 assert!(super_block.is_valid(), "Error loading EFS!");
 let inode_total_blocks =
 super_block.inode_bitmap_blocks + super_block.inode_area_blocks;
 let efs = Self {
 block_device,
 inode_bitmap: Bitmap::new(
 1,
 super_block.inode_bitmap_blocks as usize
),
 data_bitmap: Bitmap::new(
 (1 + inode_total_blocks) as usize,
 super_block.data_bitmap_blocks as usize,
),
 inode_area_start_block: 1 + super_block.inode_bitmap_blocks,
 data_area_start_block: 1 + inode_total_blocks + super_block.data_
 ↵bitmap_blocks,
 };
 Arc::new(Mutex::new(efs))
 }
 }
}

```

(下页继续)

(续上页)

```
 };
 Arc::new(Mutex::new(efs))
)
}
}
```

它只需将块设备编号为 0 的块作为超级块读取进来，就可以从中知道 easy-fs 的磁盘布局，由此可以构造 efs 实例。

EasyFileSystem 知道整个磁盘布局，即可以从 inode 位图或数据块位图上分配的 bit 编号，来算出各个存储 inode 和数据块的磁盘块在磁盘上的实际位置。

```
// easy-fs/src/efs.rs

impl EasyFileSystem {
 pub fn get_disk_inode_pos(&self, inode_id: u32) -> (u32, usize) {
 let inode_size = core::mem::size_of::<DiskInode>();
 let inodes_per_block = (BLOCK_SZ / inode_size) as u32;
 let block_id = self.inode_area_start_block + inode_id / inodes_per_block;
 (block_id, (inode_id % inodes_per_block) as usize * inode_size)
 }

 pub fn get_data_block_id(&self, data_block_id: u32) -> u32 {
 self.data_area_start_block + data_block_id
 }
}
```

inode 和数据块的分配/回收也由 EasyFileSystem 负责：

```
// easy-fs/src/efs.rs

impl EasyFileSystem {
 pub fn alloc_inode(&mut self) -> u32 {
 self.inode_bitmap.alloc(&self.block_device).unwrap() as u32
 }

 /// Return a block ID not ID in the data area.
 pub fn alloc_data(&mut self) -> u32 {
 self.data_bitmap.alloc(&self.block_device).unwrap() as u32 + self.data_area_
 ↵ start_block
 }

 pub fn deallocate(&mut self, block_id: u32) {
 get_block_cache(
 block_id as usize,
 Arc::clone(&self.block_device)
)
 .lock()
 .modify(0, |data_block: &mut DataBlock| {
 data_block.iter_mut().for_each(|p| { *p = 0; })
 });
 self.data_bitmap.deallocate(
 &self.block_device,
 (block_id - self.data_area_start_block) as usize
)
 }
}
```

注意：

- `alloc_data` 和 `dealloc_data` 分配/回收数据块传入/返回的参数都表示数据块在块设备上的编号，而不是在数据块位图中分配的 bit 编号；
- `dealloc_inode` 未实现，因为现在还不支持文件删除。

### 7.3.7 索引节点

服务于文件相关系统调用的索引节点层的代码在 `vfs.rs` 中。

`EasyFileSystem` 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` 暴露给文件系统的使用者，让他们能够直接对文件和目录进行操作。`Inode` 和 `DiskInode` 的区别从它们的名字中就可以看出：`DiskInode` 放在磁盘块中比较固定的位置，而 `Inode` 是放在内存中的记录文件索引节点信息的数据结构。

```
// easy-fs/src/vfs.rs

pub struct Inode {
 block_id: usize,
 block_offset: usize,
 fs: Arc<Mutex<EasyFileSystem>>,
 block_device: Arc<dyn BlockDevice>,
}
```

`block_id` 和 `block_offset` 记录该 `Inode` 对应的 `DiskInode` 保存在磁盘上的具体位置方便我们后续对它进行访问。`fs` 是指向 `EasyFileSystem` 的一个指针，因为对 `Inode` 的种种操作实际上都是要通过底层的文件系统来完成。

仿照 `BlockCache::read/modify`，我们可以设计两个方法来简化对于 `Inode` 对应的磁盘上的 `DiskInode` 的访问流程，而不是每次都需要 `get_block_cache.lock.read/modify`：

```
// easy-fs/src/vfs.rs

impl Inode {
 fn read_disk_inode<V>(&self, f: impl FnOnce(&DiskInode) -> V) -> V {
 get_block_cache(
 self.block_id,
 Arc::clone(&self.block_device)
).lock().read(self.block_offset, f)
 }

 fn modify_disk_inode<V>(&self, f: impl FnOnce(&mut DiskInode) -> V) -> V {
 get_block_cache(
 self.block_id,
 Arc::clone(&self.block_device)
).lock().modify(self.block_offset, f)
 }
}
```

下面分别介绍文件系统的使用者对于文件系统的一些常用操作：

## 获取根目录的 inode

文件系统的使用者在通过 `EasyFileSystem::open` 从装载了 easy-fs 镜像的块设备上打开 easy-fs 之后，要做的第一件事情就是获取根目录的 `Inode`。因为 `EasyFileSystem` 目前仅支持绝对路径，对于任何文件/目录的索引都必须从根目录开始向下逐级进行。等到索引完成之后，`EasyFileSystem` 才能对文件/目录进行操作。事实上 `EasyFileSystem` 提供了另一个名为 `root_inode` 的方法来获取根目录的 `Inode`：

```
// easy-fs/src/efs.rs

impl EasyFileSystem {
 pub fn root_inode(efs: &Arc<Mutex<Self>>) -> Inode {
 let block_device = Arc::clone(&efs.lock().block_device);
 // acquire efs lock temporarily
 let (block_id, block_offset) = efs.lock().get_disk_inode_pos(0);
 // release efs lock
 Inode::new(
 block_id,
 block_offset,
 Arc::clone(efs),
 block_device,
)
 }
}

// easy-fs/src/vfs.rs

impl Inode {
 /// We should not acquire efs lock here.
 pub fn new(
 block_id: u32,
 block_offset: usize,
 fs: Arc<Mutex<EasyFileSystem>>,
 block_device: Arc<dyn BlockDevice>,
) -> Self {
 Self {
 block_id: block_id as usize,
 block_offset,
 fs,
 block_device,
 }
 }
}
```

对于 `root_inode` 的初始化，是在调用 `Inode::new` 时将传入的 `inode_id` 设置为 0，因为根目录对应于文件系统中第一个分配的 `inode`，因此它的 `inode_id` 总会是 0。不会在调用 `Inode::new` 过程中尝试获取整个 `EasyFileSystem` 的锁来查询 `inode` 在块设备中的位置，而是在调用它之前预先查询并作为参数传过去。

## 文件索引

前面提到过，为了尽可能简化文件系统设计，EasyFileSystem是一个扁平化的文件系统，即在目录树上仅有一个目录——那就是作为根节点的根目录。所有的文件都在根目录下面。于是，我们不必实现目录索引。文件索引的查找比较简单，仅需在根目录的目录项中根据文件名找到文件的 inode 编号即可。由于没有子目录的存在，这个过程只会进行一次。

```
// easy-fs/src/vfs.rs

impl Inode {
 pub fn find(&self, name: &str) -> Option<Arc<Inode>> {
 let fs = self.fs.lock();
 self.read_disk_inode(|disk_inode| {
 self.find_inode_id(name, disk_inode)
 .map(|inode_id| {
 let (block_id, block_offset) = fs.get_disk_inode_pos(inode_id);
 Arc::new(Self::new(
 block_id,
 block_offset,
 self.fs.clone(),
 self.block_device.clone(),
)));
 })
 })
 }

 fn find_inode_id(
 &self,
 name: &str,
 disk_inode: &DiskInode,
) -> Option<u32> {
 // assert it is a directory
 assert!(disk_inode.is_dir());
 let file_count = (disk_inode.size as usize) / DIRENT_SZ;
 let mut dirent = DirEntry::empty();
 for i in 0..file_count {
 assert_eq!(
 disk_inode.read_at(
 DIRENT_SZ * i,
 dirent.as_bytes_mut(),
 &self.block_device,
),
 DIRENT_SZ,
);
 if dirent.name() == name {
 return Some(dirent.inode_number() as u32);
 }
 }
 None
 }
}
```

find 方法只会被根目录 Inode 调用，文件系统中其他文件的 Inode 不会调用这个方法。它首先调用 find\_inode\_id 方法，尝试从根目录的 DiskInode 上找到要索引的文件名对应的 inode 编号。这就需要将根目录内容中的所有目录项都读到内存进行逐个比对。如果能够找到，则 find 方法会根据查到 inode 编号，对应生成一个 Inode 用于后续对文件的访问。

这里需要注意，包括 find 在内，所有暴露给文件系统的使用者的文件系统操作（还包括接下来将要介绍的几

种), 全程均需持有 EasyFileSystem 的互斥锁 (相对而言, 文件系统内部的操作, 如之前的 Inode::new 或是上面的 find\_inode\_id, 都是假定在已持有 efs 锁的情况下才被调用的, 因此它们不应尝试获取锁)。这能够保证在多核情况下, 同时最多只能有一个核在进行文件系统相关操作。这样也许会带来一些不必要的性能损失, 但我们目前暂时先这样做。如果我们在那里加锁的话, 其实就能够保证块缓存的互斥访问了。

## 文件列举

ls 方法可以收集根目录下的所有文件的文件名并以向量的形式返回, 这个方法只有根目录的 Inode 才会调用:

```
// easy-fs/src/vfs.rs

impl Inode {
 pub fn ls(&self) -> Vec<String> {
 let _fs = self.fs.lock();
 self.read_disk_inode(|disk_inode| {
 let file_count = (disk_inode.size as usize) / DIRENT_SZ;
 let mut v: Vec<String> = Vec::new();
 for i in 0..file_count {
 let mut dirent = DirEntry::empty();
 assert_eq!(dirent.read_at(
 i * DIRENT_SZ,
 dirent.as_bytes_mut(),
 &self.block_device,
), DIRENT_SZ);
 v.push(String::from(dirent.name()));
 }
 })
 }
}
```

### 注解: Rust 语法卡片: \_ 在匹配中的使用方法

可以看到在 ls 操作中, 我们虽然获取了 efs 锁, 但是这里并不会直接访问 EasyFileSystem 实例, 其目的仅仅是锁住该实例避免其他核在同时间的访问造成并发冲突。因此, 我们将其绑定到以 \_ 开头的变量 \_fs 中, 这样即使我们在其作用域中并没有使用它, 编译器也不会报警告。然而, 我们不能将其绑定到变量 \_ 上。因为从匹配规则可以知道这意味着该操作会被编译器丢弃, 从而无法达到获取锁的效果。

## 文件创建

create 方法可以在根目录下创建一个文件, 该方法只有根目录的 Inode 会调用:

```
1 // easy-fs/src/vfs.rs
2
3 impl Inode {
4 pub fn create(&self, name: &str) -> Option<Arc<Inode>> {
5 let mut fs = self.fs.lock();
6 if self.modify_disk_inode(|root_inode| {
7 // assert it is a directory
8 })
9 }
10 }
```

(下页继续)

(续上页)

```

8 assert!(root_inode.is_dir());
9 // has the file been created?
10 self.find_inode_id(name, root_inode)
11 }).is_some() {
12 return None;
13 }
14 // create a new file
15 // alloc a inode with an indirect block
16 let new_inode_id = fs.alloc_inode();
17 // initialize inode
18 let (new_inode_block_id, new_inode_block_offset)
19 = fs.get_disk_inode_pos(new_inode_id);
20 get_block_cache(
21 new_inode_block_id as usize,
22 Arc::clone(&self.block_device)
23).lock().modify(new_inode_block_offset, |new_inode: &mut DiskInode| {
24 new_inode.initialize(DiskInodeType::File);
25 });
26 self.modify_disk_inode(|root_inode| {
27 // append file in the dirent
28 let file_count = (root_inode.size as usize) / DIRENT_SZ;
29 let new_size = (file_count + 1) * DIRENT_SZ;
30 // increase size
31 self.increase_size(new_size as u32, root_inode, &mut fs);
32 // write dirent
33 let dirent = DirEntry::new(name, new_inode_id);
34 root_inode.write_at(
35 file_count * DIRENT_SZ,
36 dirent.as_bytes(),
37 &self.block_device,
38);
39 });
40
41 let (block_id, block_offset) = fs.get_disk_inode_pos(new_inode_id);
42 // return inode
43 Some(Arc::new(Self::new(
44 block_id,
45 block_offset,
46 self.fs.clone(),
47 self.block_device.clone(),
48)));
49 // release efs lock automatically by compiler
50 }
51 }
```

- 第 6~13 行, 检查文件是否已经在根目录下, 如果找到的话返回 None ;
- 第 14~25 行, 为待创建文件分配一个新的 inode 并进行初始化;
- 第 26~39 行, 将待创建文件的目录项插入到根目录的内容中, 使得之后可以索引到。

## 文件清空

在以某些标志位打开文件（例如带有 *CREATE* 标志打开一个已经存在的文件）的时候，需要首先将文件清空。在索引到文件的 `Inode` 之后，可以调用 `clear` 方法：

```
// easy-fs/src/vfs.rs

impl Inode {
 pub fn clear(&self) {
 let mut fs = self.fs.lock();
 self.modify_disk_inode(|disk_inode| {
 let size = disk_inode.size;
 let data_blocks_dealloc = disk_inode.clear_size(&self.block_device);
 assert!(data_blocks_dealloc.len() == DiskInode::total_blocks(size) as
→ usize);
 for data_block in data_blocks_dealloc.into_iter() {
 fs.dealloc_data(data_block);
 }
 });
 }
}
```

这会将该文件占据的索引块和数据块回收。

## 文件读写

从根目录索引到一个文件之后，可以对它进行读写。注意：和 `DiskInode` 一样，这里的读写作用在字节序列的一段区间上：

```
// easy-fs/src/vfs.rs

impl Inode {
 pub fn read_at(&self, offset: usize, buf: &mut [u8]) -> usize {
 let _fs = self.fs.lock();
 self.read_disk_inode(|disk_inode| {
 disk_inode.read_at(offset, buf, &self.block_device)
 })
 }

 pub fn write_at(&self, offset: usize, buf: &[u8]) -> usize {
 let mut fs = self.fs.lock();
 self.modify_disk_inode(|disk_inode| {
 self.increase_size((offset + buf.len()) as u32, disk_inode, &mut fs);
 disk_inode.write_at(offset, buf, &self.block_device)
 })
 }
}
```

具体实现比较简单，需要注意在执行 `DiskInode::write_at` 之前先调用 `increase_size` 对自身进行扩容：

```
// easy-fs/src/vfs.rs

impl Inode {
 fn increase_size(
 &self,
```

(下页继续)

(续上页)

```

new_size: u32,
disk_inode: &mut DiskInode,
fs: &mut MutexGuard<EasyFileSystem>,
) {
 if new_size < disk_inode.size {
 return;
 }
 let blocks_needed = disk_inode.blocks_num_needed(new_size);
 let mut v: Vec<u32> = Vec::new();
 for _ in 0..blocks_needed {
 v.push(fs.alloc_data());
 }
 disk_inode.increase_size(new_size, v, &self.block_device);
}
}

```

这里会从 EasyFileSystem 中分配一些用于扩容的数据块并传给 DiskInode::increase\_size。

### 7.3.8 在用户态测试 easy-fs 的功能

easy-fs 架构设计的一个优点在于它可以在 Rust 应用开发环境（Windows/macOS/Ubuntu）中，按照应用程序库的开发方式来进行测试，不必过早的放到内核中测试运行。众所周知，内核运行在裸机环境上，对其进行调试很困难。而面向应用的开发环境对于调试的支持更为完善，从基于命令行的 GDB 到 IDE 提供的图形化调试界面都能给文件系统的开发带来很大帮助。另外一点是，由于 easy-fs 需要放到在裸机上运行的内核中，使得 easy-fs 只能使用 no\_std 模式，不能在 easy-fs 中调用标准库 std。但是在把 easy-fs 作为一个应用的库运行的时候，可以暂时让使用它的应用程序调用标准库 std，这也会在开发调试上带来一些方便。

easy-fs 的测试放在另一个名为 easy-fs-fuse 的应用程序中，不同于 easy-fs，它是一个可以调用标准库 std 的应用程序，能够在 Rust 应用开发环境上运行并很容易调试。

#### 在 Rust 应用开发环境中模拟块设备

从文件系统的使用者角度来看，它仅需要提供一个实现了 BlockDevice Trait 的块设备用来装载文件系统，之后就可以使用 Inode 来方便地进行文件系统操作了。但是在开发环境上，我们如何来提供这样一个块设备呢？答案是用 Linux（当然也可以是 Windows/MacOS 等其它通用操作系统）上的一个文件模拟一个块设备。

```

// easy-fs-fuse/src/main.rs

use std::fs::File;
use easy-fs::BlockDevice;

const BLOCK_SZ: usize = 512;

struct BlockFile(Mutex<File>);

impl BlockDevice for BlockFile {
 fn read_block(&self, block_id: usize, buf: &mut [u8]) {
 let mut file = self.0.lock().unwrap();
 file.seek(SeekFrom::Start((block_id * BLOCK_SZ) as u64))
 .expect("Error when seeking!");
 assert_eq!(file.read(buf).unwrap(), BLOCK_SZ, "Not a complete block!");
 }
}

```

(下页继续)

(续上页)

```

 }

 fn write_block(&self, block_id: usize, buf: &[u8]) {
 let mut file = self.0.lock().unwrap();
 file.seek(SeekFrom::Start((block_id * BLOCK_SZ) as u64))
 .expect("Error when seeking!");
 assert_eq!(file.write(buf).unwrap(), BLOCK_SZ, "Not a complete block!");
 }
}

```

std::file::File 由 Rust 标准库 std 提供, 可以访问 Linux 上的一个文件。我们将它包装成 BlockFile 类型来模拟一块磁盘, 为它实现 BlockDevice 接口。注意 File 本身仅通过 read/write 接口是不能实现随机读写的, 在访问一个特定的块的时候, 我们必须先 seek 到这个块的开头位置。

测试主函数为 easy-fs-fuse/src/main.rs 中的 efs\_test 函数中, 我们只需在 easy-fs-fuse 目录下 cargo test 即可执行该测试:

```

running 1 test
test efs_test ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished
in 1.27s

```

看到上面的内容就说明测试通过了。

efs\_test 展示了 easy-fs 库的使用方法, 大致分成以下几个步骤:

## 打开块设备

```

let block_file = Arc::new(BlockFile(Mutex::new({
 let f = OpenOptions::new()
 .read(true)
 .write(true)
 .create(true)
 .open("target/fs.img")?;
 f.set_len(8192 * 512).unwrap();
 f
})));
EasyFileSystem::create(
 block_file.clone(),
 4096,
 1,
);

```

第一步我们需要打开虚拟块设备。这里我们在 Linux 上创建文件 easy-fs-fuse/target/fs.img 来新建一个虚拟块设备, 并将它的容量设置为 8192 个块即 4MiB。在创建的时候需要将它的访问权限设置为可读可写。

由于我们在进行测试, 需要初始化测试环境, 因此在虚拟块设备 block\_file 上初始化 easy-fs 文件系统, 这会将 block\_file 用于放置 easy-fs 镜像的前 4096 个块上的数据覆盖, 然后变成仅有一个根目录的初始文件系统。如果块设备上已经放置了一个合法的 easy-fs 镜像, 则我们不必这样做。

## 从块设备上打开文件系统

```
1 let efs = EasyFileSystem::open(block_file.clone());
```

这是通常进行的第二个步骤。

## 获取根目录的 Inode

```
1 let root_inode = EasyFileSystem::root_inode(&efs);
```

这是通常进行的第三个步骤。

## 进行各种文件操作

拿到根目录 `root_inode` 之后，可以通过它进行各种文件操作，目前支持以下几种：

- 通过 `create` 创建文件。
- 通过 `ls` 列举根目录下的文件。
- 通过 `find` 根据文件名索引文件。

当通过索引获取根目录下的一个文件的 `inode` 之后则可以进行如下操作：

- 通过 `clear` 将文件内容清空。
- 通过 `read/write_at` 读写文件，注意我们需要将读写在文件中开始的位置 `offset` 作为一个参数传递进去。

测试方法在这里不详细介绍，大概是每次清空文件 `filea` 的内容，向其中写入一个不同长度的随机数字字符串，然后再全部读取出来，验证和写入的内容一致。其中有一个细节是：用来生成随机字符串的 `rand crate` 并不支持 `no_std`，因此只有在用户态我们才能更容易进行测试。

## 7.3.9 将应用打包为 easy-fs 镜像

在第六章中我们需要将所有的应用都链接到内核中，随后在应用管理器中通过应用名进行索引来找到应用的 ELF 数据。这样做有一个缺点，就是会造成内核体积过度膨胀。在 k210 平台上可以很明显的感觉到从第五章开始随着应用数量的增加，向开发板上烧写内核镜像的耗时显著增长。同时这也会浪费内存资源，因为未被执行的应用也占据了内存空间。在实现了 easy-fs 文件系统之后，终于可以将这些应用打包到 easy-fs 镜像中放到磁盘中，当我们要执行应用的时候只需从文件系统中取出 ELF 执行文件格式的应用并加载到内存中执行即可，这样就避免了前面章节的存储开销等问题。

`easy-fs-fuse` 的主体 `easy-fs-pack` 函数就实现了这个功能：

```
1 // easy-fs-fuse/src/main.rs
2
3 use clap::{Arg, App};
4
5 fn easy_fs_pack() -> std::io::Result<()> {
6 let matches = App::new("EasyFileSystem packer")
7 .arg(Arg::with_name("source")
8 .short("s")
9 .long("source")
10 .takes_value(true)
11 .help("Executable source dir (with backslash)"))
```

(下页继续)

(续上页)

```

12
13 .arg(Arg::with_name("target")
14 .short("t")
15 .long("target")
16 .takes_value(true)
17 .help("Executable target dir (with backslash)"))
18)
19 .get_matches();
20 let src_path = matches.value_of("source").unwrap();
21 let target_path = matches.value_of("target").unwrap();
22 println!("src_path = {}\ntarget_path = {}", src_path, target_path);
23 let block_file = Arc::new(BlockFile(Mutex::new({
24 let f = OpenOptions::new()
25 .read(true)
26 .write(true)
27 .create(true)
28 .open(format!("{}{}", target_path, "fs.img"))?;
29 f.set_len(8192 * 512).unwrap();
30 f
31 })));
32 // 4MiB, at most 4095 files
33 let efs = EasyFileSystem::create(
34 block_file.clone(),
35 8192,
36 1,
37);
38 let root_inode = Arc::new(EasyFileSystem::root_inode(&efs));
39 let apps: Vec<_> = read_dir(src_path)
40 .unwrap()
41 .into_iter()
42 .map(|dir_entry| {
43 let mut name_with_ext = dir_entry.unwrap().file_name().into_string().
44 unwrap();
45 name_with_ext.drain(name_with_ext.find('.').unwrap()..name_with_ext.
46 len());
47 name_with_ext
48 })
49 .collect();
50 for app in apps {
51 // load app data from host file system
52 let mut host_file = File::open(format!("{}{}", target_path, app)).unwrap();
53 let mut all_data: Vec<u8> = Vec::new();
54 host_file.read_to_end(&mut all_data).unwrap();
55 // create a file in easy-fs
56 let inode = root_inode.create(app.as_str()).unwrap();
57 // write data to easy-fs
58 inode.write_at(0, all_data.as_slice());
59 }
60 // list apps
61 for app in root_inode.ls() {
62 println!("{}", app);
63 }
64 Ok(())
}

```

- 为了实现 easy-fs-fuse 和 os/user 的解耦，第 6~21 行使用 clap crate 进行命令行参数解析，需要通过 -s 和 -t 分别指定应用的源代码目录和保存应用 ELF 的目录，而不是在 easy-fs-fuse 中硬

编码。如果解析成功的话它们会分别被保存在变量 `src_path` 和 `target_path` 中。

- 第 23~38 行依次完成：创建 4MiB 的 easy-fs 镜像文件、进行 easy-fs 初始化、获取根目录 inode。
- 第 39 行获取源码目录中的每个应用的源代码文件并去掉后缀名，收集到向量 `apps` 中。
- 第 48 行开始，枚举 `apps` 中的每个应用，从放置应用执行程序的目录中找到对应应用的 ELF 文件（这是一个 Linux 上的文件），并将数据读入内存。接着需要在 easy-fs 中创建一个同名文件并将 ELF 数据写入到这个文件中。这个过程相当于将 Linux 上的文件系统中的一个文件复制到我们的 easy-fs 中。

尽管没有进行任何同步写回磁盘的操作，我们也不用担心块缓存中的修改没有写回磁盘。因为在 `easy-fs-fuse` 这个应用正常退出的过程中，块缓存因生命周期结束会被回收，届时如果块缓存的 `modified` 标志为 `true`，就会将其修改写回磁盘。

## 7.4 在内核中接入 easy-fs

### 7.4.1 本节导读

上节实现了 easy-fs 文件系统，并能在用户态来进行测试，但还没有放入到内核中来。本节我们介绍如何将 easy-fs 文件系统接入内核中从而在内核中支持常规文件和目录。为此，在操作系统内核中需要有对接 easy-fs 文件系统的各种结构，它们自下而上可以分成这几个层次：

- 块设备驱动层：针对内核所要运行在的 qemu 或 k210 平台，我们需要将平台上的块设备驱动起来并实现 easy-fs 所需的 `BlockDevice Trait`，这样 easy-fs 才能将该块设备用作 easy-fs 镜像的载体。
- easy-fs 层：我们在上一节已经介绍了 easy-fs 文件系统内部的层次划分。这里是站在内核的角度，只需知道它接受一个块设备 `BlockDevice`，并可以在上面打开文件系统 `EasyFileSystem`，进而获取 `Inode` 核心数据结构，进行各种文件系统操作即可。
- 内核索引节点层：在内核中需要将 easy-fs 提供的 `Inode` 进一步封装成 `OSInode`，以表示进程中一个打开的常规文件。由于有很多种不同的打开方式，因此在 `OSInode` 中要维护一些额外的信息。
- 文件描述符层：常规文件对应的 `OSInode` 是文件的内核内部表示，因此需要为它实现 `File Trait` 从而能够可以将它放入到进程文件描述符表中并通过 `sys_read/write` 系统调用进行读写。
- 系统调用层：由于引入了常规文件这种文件类型，导致一些系统调用以及相关的内核机制需要进行一定的修改。

### 7.4.2 文件简介

应用程序看到并被操作系统管理的 **文件 (File)** 就是一系列的字节组合。操作系统不关心文件内容，只关心如何对文件按字节流进行读写的机制，这就意味着任何程序可以读写任何文件（即字节流），对文件具体内容的解析是应用程序的任务，操作系统对此不做任何干涉。例如，一个 Rust 编译器可以读取一个 C 语言源程序并进行编译，操作系统并不会阻止这样的事情发生。

有了文件这样的抽象后，操作系统内核就可把能读写并持久存储的数据按文件来进行管理，并把文件分配给进程，让进程以很简洁的统一抽象接口 `File` 来读写数据：

```
// os/src/fs/mod.rs

pub trait File : Send + Sync {
 fn read(&self, buf: UserBuffer) -> usize;
 fn write(&self, buf: UserBuffer) -> usize;
}
```

这个接口在内存和存储设备之间建立了数据交换的通道。其中 `UserBuffer` 是我们在 `mm` 子模块中定义的应用地址空间中的一段缓冲区（即内存）的抽象。它的具体实现本质上其实只是一个 `&[u8]`，位于应用地址空间中，内核无法直接通过用户地址空间的虚拟地址来访问，因此需要进行封装。然而，在理解抽象接口 `File` 的各方法时，我们仍可以将 `UserBuffer` 看成一个 `&[u8]` 切片，它是一个同时给出了缓冲区起始地址和长度的胖指针。

`read` 指的是从文件中读取数据放到缓冲区中，最多将缓冲区填满（即读取缓冲区的长度那么多字节），并返回实际读取的字节数；而 `write` 指的是将缓冲区中的数据写入文件，最多将缓冲区中的数据全部写入，并返回直接写入的字节数。至于 `read` 和 `write` 的实现则与文件具体是哪种类型有关，它决定了数据如何被读取和写入。

回过头来再看一下用户缓冲区的抽象 `UserBuffer`，它的声明如下：

```
// os/src/mm/page_table.rs

pub fn translated_byte_buffer(
 token: usize,
 ptr: *const u8,
 len: usize
) -> Vec<&'static mut [u8]>;

pub struct UserBuffer {
 pub buffers: Vec<&'static mut [u8]>,
}

impl UserBuffer {
 pub fn new(buffers: Vec<&'static mut [u8]>) -> Self {
 Self { buffers }
 }
 pub fn len(&self) -> usize {
 let mut total: usize = 0;
 for b in self.buffers.iter() {
 total += b.len();
 }
 total
 }
}
```

它只是将我们调用 `translated_byte_buffer` 获得的包含多个切片的 `Vec` 进一步包装起来，通过 `len` 方法可以得到缓冲区的长度。此外，我们还让它作为一个迭代器可以逐字节进行读写。有兴趣的同学可以参考类型 `UserBufferIterator` 还有 `IntoIterator` 和 `Iterator` 两个 Trait 的使用方法。

### 7.4.3 块设备驱动层

在 `drivers` 子模块中的 `block/mod.rs` 中，我们可以找到内核访问的块设备实例 `BLOCK_DEVICE`：

```
// os/drivers/block/mod.rs

#[cfg(feature = "board_qemu")]
type BlockDeviceImpl = virtio_blk::VirtIOBlock;

#[cfg(feature = "board_k210")]
type BlockDeviceImpl = sdcard::SDCardWrapper;

lazy_static! {
```

(下页继续)

(续上页)

```

1 pub static ref BLOCK_DEVICE: Arc<dyn BlockDevice> =_
2 ↪Arc::new(BlockDeviceImpl::new());
3

```

qemu 和 k210 平台上的块设备是不同的。在 qemu 上，我们使用 VirtIOBlock 访问 VirtIO 块设备；而在 k210 上，我们使用 SDCardWrapper 来访问插入 k210 开发板上真实的 microSD 卡，它们都实现了 easy-fs 要求的 BlockDevice Trait。通过 #[cfg(feature)] 可以在编译的时候根据编译参数调整 BlockDeviceImpl 具体为哪个块设备，之后将它全局实例化为 BLOCK\_DEVICE，使得内核的其他模块可以访问。

## Qemu 模拟器平台

在启动 Qemu 模拟器的时候，我们可以配置参数来添加一块 VirtIO 块设备：

```

1 # os/Makefile
2
3 FS_IMG := ../user/target/$ (TARGET)/$ (MODE)/fs.img
4
5 run-inner: build
6 ifeq ($ (BOARD), qemu)
7 @qemu-system-riscv64 \
8 -machine virt \
9 -nographic \
10 -bios $ (BOOTLOADER) \
11 -device loader,file=$ (KERNEL_BIN),addr=$ (KERNEL_ENTRY_PA) \
12 -drive file=$ (FS_IMG),if=none,format=raw,id=x0 \
13 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

```

- 第 12 行，我们为虚拟机添加一块虚拟硬盘，内容为我们之前通过 easy-fs-fuse 工具打包的包含应用 ELF 的 easy-fs 镜像，并命名为 x0。
- 第 13 行，我们将硬盘 x0 作为一个 VirtIO 总线中的一个块设备接入到虚拟机系统中。virtio-mmio-bus.0 表示 VirtIO 总线通过 MMIO 进行控制，且该块设备在总线中的编号为 0。

**内存映射 I/O (MMIO, Memory-Mapped I/O)** 指的是外设的设备寄存器可以通过特定的物理内存地址来访问，每个外设的设备寄存器都分布在没有交集的一个或数个物理地址区间中，不同外设的设备寄存器所占的物理地址空间也不会产生交集，且这些外设物理地址区间也不会和 RAM 的物理内存所在的区间存在交集（注：在后续的外设相关章节有更深入的讲解）。从 Qemu for RISC-V 64 平台的 [源码](#) 中可以找到 VirtIO 外设总线的 MMIO 物理地址区间为从 0x10001000 开头的 4KiB。为了能够在内核中访问 VirtIO 外设总线，我们就必须在内核地址空间中对特定内存区域提前进行映射：

```

// os/src/config.rs

#[cfg(feature = "board_qemu")]
pub const MMIO: &[(&u32, &u32)] = &[
 (0x10001000, 0x1000),
];

```

如上面一段代码所示，在 config 子模块中我们硬编码 Qemu 上的 VirtIO 总线的 MMIO 地址区间（起始地址，长度）。在创建内核地址空间的时候需要建立页表映射：

```

// os/src/mm/memory_set.rs

use crate::config::MMIO;

impl MemorySet {

```

(下页继续)

(续上页)

```

/// Without kernel stacks.
pub fn new_kernel() -> Self {
 ...
 println!("mapping memory-mapped registers");
 for pair in MMIO {
 memory_set.push(MapArea::new(
 (*pair).0.into(),
 ((*pair).0 + (*pair).1).into(),
 MapType::Identical,
 MapPermission::R | MapPermission::W,
), None);
 }
 memory_set
}
}

```

这里我们进行的是透明的恒等映射，从而让内核可以兼容于直接访问物理地址的设备驱动库。

由于设备驱动的开发过程比较琐碎，我们这里直接使用已有的 `virtio-drivers` crate，它已经支持 VirtIO 总线架构下的块设备、网络设备、GPU 等设备。注：关于 VirtIO 相关驱动的内容，在后续的外设相关章节有更深入的讲解。

```

// os/src/drivers/block/virtio_blk.rs

use virtio_drivers::{VirtIOBlk, VirtIOHeader};
const VIRTIO0: usize = 0x10001000;

pub struct VirtIOBlock(Mutex<VirtIOBlk<'static>>);

impl VirtIOBlock {
 pub fn new() -> Self {
 Self(Mutex::new(VirtIOBlk::new(
 unsafe { &mut *(VIRTIO0 as *mut VirtIOHeader) }
).unwrap()))
 }
}

impl BlockDevice for VirtIOBlock {
 fn read_block(&self, block_id: usize, buf: &mut [u8]) {
 self.0.lock().read_block(block_id, buf).expect("Error when reading VirtIOBlk");
 }
 fn write_block(&self, block_id: usize, buf: &[u8]) {
 self.0.lock().write_block(block_id, buf).expect("Error when writing VirtIOBlk");
 }
}

```

上面的代码中，我们将 `virtio-drivers` crate 提供的 VirtIO 块设备抽象 `VirtIOBlk` 包装为我们自己的 `VirtIOBlock`，实质上只是加上了一层互斥锁，生成一个新的类型来实现 `easy-fs` 需要的 `BlockDevice Trait`。注意在 `VirtIOBlk::new` 的时候需要传入一个 `&mut VirtIOHeader` 的参数，`VirtIOHeader` 实际上就代表以 MMIO 方式访问 VirtIO 设备所需的一组设备寄存器。因此我们从 `qemu-system-riscv64` 平台上的 Virtio MMIO 区间左端 `VIRTIO0` 开始转化为一个 `&mut VirtIOHeader` 就可以在该平台上访问这些设备寄存器了。

很容易为 `VirtIOBlock` 实现 `BlockDevice Trait`，因为它内部来自 `virtio-drivers` crate 的 `VirtIOBlk` 类型已经实现了 `read/write_block` 方法，我们进行转发即可。

VirtIO 设备需要占用部分内存作为一个公共区域从而更好的和 CPU 进行合作。这就像 MMU 需要在内存中保存多级页表才能和 CPU 共同实现分页机制一样。在 VirtIO 架构下，需要在公共区域中放置一种叫做 VirtQueue 的环形队列，CPU 可以向此环形队列中向 VirtIO 设备提交请求，也可以从队列中取得请求的结果，详情可以参考 [virtio 文档](#)。对于 VirtQueue 的使用涉及到物理内存的分配和回收，但这并不在 VirtIO 驱动 `virtio-drivers` 的职责范围之内，因此它声明了数个相关的接口，需要库的使用者自己来实现：

```
// https://github.com/rcore-os/virtio-drivers/blob/master/src/hal.rs#L57

extern "C" {
 fn virtio_dma_alloc(pages: usize) -> PhysAddr;
 fn virtio_dma_dealloc(paddr: PhysAddr, pages: usize) -> i32;
 fn virtio_phys_to_virt(paddr: PhysAddr) -> VirtAddr;
 fn virtio_virt_to_phys(vaddr: VirtAddr) -> PhysAddr;
}
```

由于我们已经实现了基于分页内存管理的地址空间，实现这些功能自然不在话下：

```
// os/src/drivers/block/virtio_blk.rs

lazy_static! {
 static ref QUEUE_FRAMES: Mutex<Vec<FrameTracker>> = Mutex::new(Vec::new());
}

#[no_mangle]
pub extern "C" fn virtio_dma_alloc(pages: usize) -> PhysAddr {
 let mut ppn_base = PhysPageNum(0);
 for i in 0..pages {
 let frame = frame_alloc().unwrap();
 if i == 0 { ppn_base = frame.ppn; }
 assert_eq!(frame.ppn.0, ppn_base.0 + i);
 QUEUE_FRAMES.lock().push(frame);
 }
 ppn_base.into()
}

#[no_mangle]
pub extern "C" fn virtio_dma_dealloc(pa: PhysAddr, pages: usize) -> i32 {
 let mut ppn_base: PhysPageNum = pa.into();
 for _ in 0..pages {
 frame_dealloc(ppn_base);
 ppn_base.step();
 }
 0
}

#[no_mangle]
pub extern "C" fn virtio_phys_to_virt(paddr: PhysAddr) -> VirtAddr {
 VirtAddr(paddr.0)
}

#[no_mangle]
pub extern "C" fn virtio_virt_to_phys(vaddr: VirtAddr) -> PhysAddr {
 PageTable::from_token(kernel_token()).translate_va(vaddr).unwrap()
}
```

这里有一些细节需要注意：

- `virtio_dma_alloc/dealloc` 需要分配/回收数个连续的物理页帧，而我们的 `frame_alloc` 是逐个分配，严格来说并不保证分配的连续性。幸运的是，这个过程只会发生在内核初始化阶段，因此能

够保证连续性。

- 在 `virtio_dma_alloc` 中通过 `frame_alloc` 得到的那些物理页帧 `FrameTracker` 都会被保存在全局的向量 `QUEUE_FRAMES` 以延长它们的生命周期，避免提前被回收。

## K210 真实硬件平台

在 K210 开发板上，我们可以插入 microSD 卡并将其作为块设备。相比 VirtIO 块设备来说，想要将 microSD 驱动起来是一件比较困难的事情。microSD 自身的通信规范比较复杂，且还需考虑在 K210 中 microSD 挂在 **串行外设接口 (SPI, Serial Peripheral Interface)** 总线上的情况。此外还需要正确设置 GPIO 的管脚映射并调整各锁相环的频率。实际上，在一块小小的芯片中除了 K210 CPU 之外，还集成了很多不同种类的外设和控制模块，它们内在的关联比较紧密，不能像 VirtIO 设备那样容易地从系统中独立出来。

好在目前 Rust 嵌入式的生态正高速发展，针对 K210 平台也有比较成熟的封装了各类外设接口的库可以用来开发上层应用。但是其功能往往分散为多个 `crate`，在使用的时候需要开发者根据需求自行进行组装。这属于 Rust 的特点之一，和 C 语言提供一个一站式的板级开发包风格有很大的不同。在开发的时候，笔者就从社区中选择了一些 `crate` 并进行了微量修改最终变成 `k210-hal/k210-pac/k210-soc` 三个能够运行在 S 特权级（它们的原身仅支持运行在 M 特权级）的 `crate`，它们可以更加便捷的实现 microSD 的驱动。关于 microSD 的驱动 `SDCardWrapper` 的实现，有兴趣的同学可以参考 `os/src/drivers/block/socard.rs`。

### 注解：感谢相关 `crate` 的原身

- `k210-hal`
- `k210-pac`
- `k210-sdk-stuff`

要在 K210 上启用 microSD，执行的时候无需任何改动，只需在 `make run` 之前将 microSD 插入 PC 再通过 `make sdcard` 将 `easy-fs` 镜像烧写进去即可。而后，将 microSD 插入 K210 开发板，连接到 PC 再 `make run`。

在对 microSD 进行操作的时候，会涉及到 K210 内置的各种外设，正所谓“牵一发而动全身”。因此 K210 平台上的 MMIO 包含很多区间：

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

#[cfg(feature = "board_k210")]
pub const MMIO: &[(<code>usize, <code>usize)] = &[
 // we don't need clint in S priv when running
 // we only need claim/complete for target0 after initializing
 (0x0C00_0000, 0x3000), /* PLIC */
 (0x0C20_0000, 0x1000), /* PLIC */
 (0x3800_0000, 0x1000), /* UARTHS */
 (0x3800_1000, 0x1000), /* GPIOHS */
 (0x5020_0000, 0x1000), /* GPIO */
 (0x5024_0000, 0x1000), /* SPI_SLAVE */
 (0x502B_0000, 0x1000), /* FPIOA */
 (0x502D_0000, 0x1000), /* TIMER0 */
 (0x502E_0000, 0x1000), /* TIMER1 */
 (0x502F_0000, 0x1000), /* TIMER2 */
 (0x5044_0000, 0x1000), /* SYSCTL */
 (0x5200_0000, 0x1000), /* SPI0 */
 (0x5300_0000, 0x1000), /* SPI1 */
 (0x5400_0000, 0x1000), /* SPI2 */
];
```

## 7.4.4 内核索引节点层

在本章的第一小节我们介绍过，站在用户的角度看来，在一个进程中可以使用多种不同的标志来打开一个文件，这会影响到打开的这个文件可以用何种方式被访问。此外，在连续调用 `sys_read/write` 读写一个文件的时候，我们知道进程中也存在着一个文件读写的当前偏移量，它也随着文件读写的进行而被不断更新。这些用户视角中的文件系统抽象特征需要内核来实现，与进程有很大的关系，而 `easy-fs` 文件系统不必涉及这些与进程结合紧密的属性。因此，我们需要将 `easy-fs` 提供的 `Inode` 加上上述信息，进一步封装为 OS 中的索引节点 `OSInode`：

```
// os/src/fs/inode.rs

pub struct OSInode {
 readable: bool,
 writable: bool,
 inner: Mutex<OSInodeInner>,
}

pub struct OSInodeInner {
 offset: usize,
 inode: Arc<Inode>,
}

impl OSInode {
 pub fn new(
 readable: bool,
 writable: bool,
 inode: Arc<Inode>,
) -> Self {
 Self {
 readable,
 writable,
 inner: Mutex::new(OSInodeInner {
 offset: 0,
 inode,
 }),
 }
 }
}
```

`OSInode` 就表示进程中一个被打开的常规文件或目录。`readable/writable` 分别表明该文件是否允许通过 `sys_read/write` 进行读写。至于在 `sys_read/write` 期间被维护偏移量 `offset` 和它在 `easy-fs` 中的 `Inode` 则加上一把互斥锁丢到 `OSInodeInner` 中。这在提供内部可变性的同时，也可以简单应对多个进程同时读写一个文件的情况。

## 7.4.5 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：

```
// os/src/fs/inode.rs

impl File for OSInode {
 fn readable(&self) -> bool { self.readable }
 fn writable(&self) -> bool { self.writable }
 fn read(&self, mut buf: UserBuffer) -> usize {
 let mut inner = self.inner.lock();
 let mut total_read_size = 0usize;
 for slice in buf.buffers.iter_mut() {
 let read_size = inner.inode.read_at(inner.offset, *slice);
 if read_size == 0 {
 break;
 }
 inner.offset += read_size;
 total_read_size += read_size;
 }
 total_read_size
 }
 fn write(&self, buf: UserBuffer) -> usize {
 let mut inner = self.inner.lock();
 let mut total_write_size = 0usize;
 for slice in buf.buffers.iter() {
 let write_size = inner.inode.write_at(inner.offset, *slice);
 assert_eq!(write_size, slice.len());
 inner.offset += write_size;
 total_write_size += write_size;
 }
 total_write_size
 }
}
```

本章我们为 File Trait 新增了 readable/writable 两个抽象接口从而在 sys\_read/sys\_write 的时候进行简单的访问权限检查。read/write 的实现也比较简单，只需遍历 UserBuffer 中的每个缓冲区片段，调用 Inode 写好的 read/write\_at 接口就好了。注意 read/write\_at 的起始位置是在 OSInode 中维护的 offset，这个 offset 也随着遍历的进行被持续更新。在 read/write 的全程需要获取 OSInode 的互斥锁，保证两个进程无法同时访问同个文件。

## 7.4.6 文件描述符表

为了支持进程对文件的管理，我们需要在进程控制块中加入文件描述符表的相应字段：

```
1 // os/src/task/task.rs
2
3 pub struct TaskControlBlockInner {
4 pub trap_cx_ppn: PhysPageNum,
5 pub base_size: usize,
6 pub task_cx_ptr: usize,
7 pub task_status: TaskStatus,
8 pub memory_set: MemorySet,
9 pub parent: Option<Weak<TaskControlBlock>>,
10 pub children: Vec<Arc<TaskControlBlock>>,
11 pub exit_code: i32,
```

(下页继续)

(续上页)

```

12 pub fd_table: Vec<Option<Arc<dyn File + Send + Sync>>>,
13 }
```

可以看到 `fd_table` 的类型包含多层嵌套，我们从外到里分别说明：

- `Vec` 的动态长度特性使得我们无需设置一个固定的文件描述符数量上限，我们可以更加灵活的使用内存，而不必操心内存管理问题；
- `Option` 使得我们可以区分一个文件描述符当前是否空闲，当它是 `None` 的时候是空闲的，而 `Some` 则代表它已被占用；
- `Arc` 首先提供了共享引用能力。后面我们会提到，可能会有多个进程共享同一个文件对它进行读写。此外被它包裹的内容会被放到内核堆而不是栈上，于是它便不需要在编译期有着确定的大小；
- `dyn` 关键字表明 `Arc` 里面的类型实现了 `File`/`Send`/`Sync` 三个 Trait，但是编译期无法知道它具体是哪个类型（可能是任何实现了 `File` Trait 的类型如 `Stdin`/`Stdout`，故而它所占的空间大小自然也无法确定），需要等到运行时才能知道它的具体类型，对于一些抽象方法的调用也是在那个时候才能找到该类型实现的方法并跳转过去。

#### 注解：Rust 语法卡片：Rust 中的多态

在编程语言中，**多态** (Polymorphism) 指的是在同一段代码中可以隐含多种不同类型的特征。在 Rust 中主要通过泛型和 Trait 来实现多态。

泛型是一种 **编译期多态** (Static Polymorphism)，在编译一个泛型函数的时候，编译器会对于所有可能用到的类型进行实例化并对应生成一个版本的汇编代码，在编译期就能知道选取哪个版本并确定函数地址，这可能会导致生成的二进制文件体积较大；而 Trait 对象（也即上面提到的 `dyn` 语法）是一种 **运行时多态** (Dynamic Polymorphism)，需要在运行时查一种类似于 C++ 中的 **虚表** (Virtual Table) 才能找到实际类型对于抽象接口实现的函数地址并进行调用，这样会带来一定的运行时开销，但是更省空间且灵活。

### 7.4.7 应用访问文件的内核机制实现

应用程序在访问文件之前，首先需要完成对文件系统的初始化和加载。这可以通过操作系统来完成，也可以让应用程序发出文件系统相关的系统调用（如 `mount` 等）来完成。我们这里的选择是让操作系统直接完成。

应用程序如果要基于文件进行 I/O 访问，大致就会涉及如下一些系统调用：

- 打开文件—`sys_open`：进程只有打开文件，操作系统才能返回一个可进行读写的文件描述符给进程，进程才能基于这个值来进行对应文件的读写。
- 关闭文件—`sys_close`：进程基于文件描述符关闭文件后，就不能再对文件进行读写操作了，这样可以在一定程度上保证对文件的合法访问。
- 读文件—`sys_read`：进程可以基于文件描述符来读文件内容到相应内存中。
- 写文件—`sys_write`：进程可以基于文件描述符来把相应内存内容写到文件中。

## 文件系统初始化

在上一小节我们介绍过，为了使用 easy-fs 提供的抽象和服务，我们需要进行一些初始化操作才能成功将 easy-fs 接入到我们的内核中。按照前面总结的步骤：

1. 打开块设备。从本节前面可以看出，我们已经打开并可以访问装载有 easy-fs 文件系统镜像的块设备 BLOCK\_DEVICE；
2. 从块设备 BLOCK\_DEVICE 上打开文件系统；
3. 从文件系统中获取根目录的 inode。

2-3 步我们在这里完成：

```
// os/src/fs/inode.rs

lazy_static! {
 pub static ref ROOT_INODE: Arc<Inode> = {
 let efs = EasyFileSystem::open(BLOCK_DEVICE.clone());
 Arc::new(EasyFileSystem::root_inode(&efs))
 };
}
```

这之后就可以使用根目录的 inode ROOT\_INODE，在内核中进行各种 easy-fs 的相关操作了。例如，在文件系统初始化完毕之后，在内核主函数 rust\_main 中调用 list\_apps 函数来列举文件系统中可用的应用的文件名：

```
// os/src/fs/inode.rs

pub fn list_apps() {
 println!("/**** APPS ****/");
 for app in ROOT_INODE.ls() {
 println!("{}", app);
 }
 println!("*****");
}
```

## 打开与关闭文件

我们需要在内核中也定义一份打开文件的标志 OpenFlags：

```
// os/src/fs/inode.rs

bitflags! {
 pub struct OpenFlags: u32 {
 const RDONLY = 0;
 const WRONLY = 1 << 0;
 const RDWR = 1 << 1;
 const CREATE = 1 << 9;
 const TRUNC = 1 << 10;
 }
}

impl OpenFlags {
 /// Do not check validity for simplicity
 /// Return (readable, writable)
 pub fn read_write(&self) -> (bool, bool) {
```

(下页继续)

(续上页)

```
 if self.is_empty() {
 (true, false)
 } else if self.contains(Self::WRONLY) {
 (false, true)
 } else {
 (true, true)
 }
}
```

它的 `read_write` 方法可以根据标志的情况返回要打开的文件是否允许读写。简单起见，这里假设标志自身一定合法。

接着，我们实现 `open_file` 内核函数，可根据文件名打开一个根目录下的文件：

```
// os/src/fs/inode.rs

pub fn open_file(name: &str, flags: OpenFlags) -> Option<Arc<OSInode>> {
 let (readable, writable) = flags.read_write();
 if flags.contains(OpenFlags::CREATE) {
 if let Some(inode) = ROOT_INODE.find(name) {
 // clear size
 inode.clear();
 Some(Arc::new(OSInode::new(
 readable,
 writable,
 inode,
)))
 } else {
 // create file
 ROOT_INODE.create(name)
 .map(|inode| {
 Arc::new(OSInode::new(
 readable,
 writable,
 inode,
)))
 })
 }
 } else {
 ROOT_INODE.find(name)
 .map(|inode| {
 if flags.contains(OpenFlags::TRUNC) {
 inode.clear();
 }
 Arc::new(OSInode::new(
 readable,
 writable,
 inode
)))
 })
 }
}
```

这里主要是实现了 `OpenFlags` 各标志位的语义。例如只有 `flags` 参数包含 `CREATE` 标志位才允许创建文件；而如果文件已经存在，则清空文件的内容。另外我们将从 `OpenFlags` 解析得到的读写相关权限传入 `OSInode` 的创建过程中。

在其基础上, sys\_open 也就很容易实现了:

```
// os/src/syscall/fs.rs

pub fn sys_open(path: *const u8, flags: u32) -> isize {
 let task = current_task().unwrap();
 let token = current_user_token();
 let path = translated_str(token, path);
 if let Some(inode) = open_file(
 path.as_str(),
 OpenFlags::from_bits(flags).unwrap()
) {
 let mut inner = task.inner_exclusive_access();
 let fd = inner.alloc_fd();
 inner.fd_table[fd] = Some(inode);
 fd as isize
 } else {
 -1
 }
}
```

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。

```
// os/src/syscall/fs.rs

pub fn sys_close(fd: usize) -> isize {
 let task = current_task().unwrap();
 let mut inner = task.inner_exclusive_access();
 if fd >= inner.fd_table.len() {
 return -1;
 }
 if inner.fd_table[fd].is_none() {
 return -1;
 }
 inner.fd_table[fd].take();
 0
}
```

## 基于文件来加载并执行应用

在有了文件系统支持之后, 我们在 sys\_exec 所需的应用的 ELF 文件格式的数据就不再需要通过应用加载器从内核的数据段获取, 而是从文件系统中获取, 这样内核与应用的代码/数据就解耦了:

```
1 // os/src/syscall/process.rs
2
3 pub fn sys_exec(path: *const u8) -> isize {
4 let token = current_user_token();
5 let path = translated_str(token, path);
6 if let Some(app_inode) = open_file(path.as_str(), OpenFlags::RDONLY) {
7 let all_data = app_inode.read_all();
8 let task = current_task().unwrap();
9 task.exec(all_data.as_slice());
10 0
11 } else {
```

(下页继续)

(续上页)

```

12 -1
13 }
14 }
```

注意上面代码片段中的高亮部分。当执行获取应用的 ELF 数据的操作时，首先调用 `open_file` 函数，以只读的方式在内核中打开应用文件并获取它对应的 `OSInode`。接下来可以通过 `OSInode::read_all` 将该文件的数据全部读到一个向量 `all_data` 中：

```

// os/src/fs/inode.rs

impl OSInode {
 pub fn read_all(&self) -> Vec<u8> {
 let mut inner = self.inner.lock();
 let mut buffer = [0u8; 512];
 let mut v: Vec<u8> = Vec::new();
 loop {
 let len = inner.inode.read_at(inner.offset, &mut buffer);
 if len == 0 {
 break;
 }
 inner.offset += len;
 v.extend_from_slice(&buffer[..len]);
 }
 v
 }
}
```

之后，就可以从向量 `all_data` 中拿到应用中的 ELF 数据，当解析完毕并创建完应用地址空间后该向量将会被回收。

同样的，我们在内核中创建初始进程 `initproc` 也需要替换为基于文件系统的实现：

```

// os/src/task/mod.rs

lazy_static! {
 pub static ref INITPROC: Arc<TaskControlBlock> = Arc::new({
 let inode = open_file("initproc", OpenFlags::RDONLY).unwrap();
 let v = inode.read_all();
 TaskControlBlock::new(v.as_slice())
 });
}
```

## 读写文件

基于文件抽象接口和文件描述符表，我们可以按照无结构的字节流来处理基本的文件读写，这样可以让文件读写系统调用 `sys_read/write` 变得更加具有普适性，为后续支持把管道等抽象为文件打下了基础：

```

// os/src/syscall/fs.rs

pub fn sys_write(fd: usize, buf: *const u8, len: usize) -> isize {
 let token = current_user_token();
 let task = current_task().unwrap();
 let inner = task.inner_exclusive_access();
 if fd >= inner.fd_table.len() {
 return -1;
 }
```

(下页继续)

(续上页)

```

 }
 if let Some(file) = &inner.fd_table[fd] {
 let file = file.clone();
 // release current task TCB manually to avoid multi-borrow
 drop(inner);
 file.write(
 UserBuffer::new(translated_byte_buffer(token, buf, len))
) as isize
 } else {
 -1
 }
}

pub fn sys_read(fd: usize, buf: *const u8, len: usize) -> isize {
 let token = current_user_token();
 let task = current_task().unwrap();
 let inner = task.inner_exclusive_access();
 if fd >= inner.fd_table.len() {
 return -1;
 }
 if let Some(file) = &inner.fd_table[fd] {
 let file = file.clone();
 // release current task TCB manually to avoid multi-borrow
 drop(inner);
 file.read(
 UserBuffer::new(translated_byte_buffer(token, buf, len))
) as isize
 } else {
 -1
 }
}

```

操作系统都是通过文件描述符在当前进程的文件描述符表中找到某个文件，无需关心文件具体的类型，只要知道它一定实现了 `File Trait` 的 `read/write` 方法即可。`Trait` 对象提供的运行时多态能力会在运行的时候帮助我们定位到符合实际类型的 `read/write` 方法。

## 7.5 练习

### 7.5.1 课后练习

#### 编程题

1. \* 扩展 easy-fs 文件系统功能，扩大单个文件的大小，支持三重间接 inode。
2. \* 扩展内核功能，支持 `stat` 系统调用，能显示文件的 `inode` 元数据信息。
3. \*\* 扩展内核功能，支持 `mmap` 系统调用，支持对文件的映射，实现基于内存读写方式的文件读写功能。
4. \*\* 扩展 easy-fs 文件系统功能，支持二级目录结构。可扩展：支持 N 级目录结构。
5. \*\*\* 扩展 easy-fs 文件系统功能，通过日志机制支持 crash 一致性。

## 问答题

1. \* 文件系统的功能是什么？
2. \*\* 目前的文件系统只有单级目录，假设想要支持多级文件目录，请描述你设想的实现方式，描述合理即可。
3. \*\* 软链接和硬链接是干什么的？有什么区别？当删除一个软链接或硬链接时分别会发生什么？
4. \*\*\* 在有了多级目录之后，我们就可以为一个目录增加硬链接了。在这种情况下，文件树中是否可能出现环路（软硬链接都可以，鼓励多尝试）？你认为应该如何解决？请在你喜欢的系统上实现一个环路，描述你的实现方式以及系统提示、实际测试结果。
5. \* 目录是一类特殊的文件，存放的是什么内容？用户可以自己修改目录内容吗？
6. \*\* 在实际操作系统中，如 Linux，为什么会存在大量的文件系统类型？
7. \*\* 可以把文件控制块放到目录项中吗？这样做有什么优缺点？
8. \*\* 为什么要同时维护进程的打开文件表和操作系统的打开文件表？这两个打开文件表有什么区别和联系？
9. \*\* 文件分配的三种方式是如何组织文件数据块的？各有什么特征（存储、文件读写、可靠性）？
10. \*\* 如果一个程序打开了一个文件，写入了一些数据，但是没有及时关闭，可能会有什么后果？如果打开文件后，又进一步发出了读文件的系统调用，操作系统中各个组件是如何相互协作完成整个读文件的系统调用的？
11. \*\*\* 文件系统是一个操作系统必要的组件吗？是否可以将文件系统放到用户态？这样做有什么好处？操作系统需要提供哪些基本支持？

## 7.5.2 实验练习

实验练习包括实践作业和问答作业两部分。

**理解文件系统比较费事，编程难度适中**

### 实践作业

#### 硬链接

硬链接要求两个不同的目录项指向同一个文件，在我们的文件系统中也就是两个不同名称目录项指向同一个磁盘块。

本节要求实现三个系统调用 `sys_linkat`、`sys_unlinkat`、`sys_stat`。

**linkat:**

- syscall ID: 37
- 功能：创建一个文件的一个硬链接，`linkat` 标准接口。
- C 接口: `int linkat(int olddirfd, char* oldpath, int newdirfd, char* newpath, unsigned int flags)`
- Rust 接口: `fn linkat(olddirfd: i32, oldpath: *const u8, newdirfd: i32, newpath: *const u8, flags: u32) -> i32`
- **参数:**
  - `olddirfd`, `newdirfd`: 仅为了兼容性考虑，本次实验中始终为 `AT_FDCWD` (-100)，可以忽略。

- flags: 仅为了兼容性考虑, 本次实验中始终为 0, 可以忽略。
- oldpath: 原有文件路径
- newpath: 新的链接文件路径。

- 说明:

- 为了方便, 不考虑新文件路径已经存在的情况 (属于未定义行为), 除非链接同名文件。
- 返回值: 如果出现了错误则返回 -1, 否则返回 0。

- 可能的错误

- 链接同名文件。

**unlinkat:**

- syscall ID: 35
- 功能: 取消一个文件路径到文件的链接, `unlinkat` 标准接口。
- C 接口: `int unlinkat(int dirfd, char* path, unsigned int flags)`
- Rust 接口: `fn unlinkat(dirfd: i32, path: *const u8, flags: u32) -> i32`

- 参数:

- dirfd: 仅为了兼容性考虑, 本次实验中始终为 `AT_FDCWD` (-100), 可以忽略。
- flags: 仅为了兼容性考虑, 本次实验中始终为 0, 可以忽略。
- path: 文件路径。

- 说明:

- 为了方便, 不考虑使用 `unlink` 彻底删除文件的情况。

- 返回值: 如果出现了错误则返回 -1, 否则返回 0。

- 可能的错误

- 文件不存在。

**fstat:**

- syscall ID: 80
- 功能: 获取文件状态。
- C 接口: `int fstat(int fd, struct Stat* st)`
- Rust 接口: `fn fstat(fd: i32, st: *mut Stat) -> i32`
- 参数:

- fd: 文件描述符
- st: 文件状态结构体

```

#[repr(C)]
#[derive(Debug)]
pub struct Stat {
 /// 文件所在磁盘驱动器号, 该实验中写死为 0 即可
 pub dev: u64,
 /// inode 文件所在 inode 编号
 pub ino: u64,
 /// 文件类型
}

```

(下页继续)

(续上页)

```

pub mode: StatMode,
/// 硬链接数量, 初始为1
pub nlink: u32,
/// 无需考虑, 为了兼容性设计
pad: [u64; 7],
}

/// StatMode 定义:
bitflags! {
 pub struct StatMode: u32 {
 const NULL = 0;
 /// directory
 const DIR = 0o040000;
 /// ordinary regular file
 const FILE = 0o100000;
 }
}

```

## 实验要求

- 实现分支: ch7-lab
- 实验目录要求不变
- 通过所有测例

在 os 目录下 make run TEST=1 加载所有测例, test\_usertest 打包了所有你需要通过的测例, 你也可以通过修改这个文件调整本地测试的内容。

你的内核必须前向兼容, 能通过前一章的所有测例。

---

### 注解: 如何调试 easy-fs

如果你在第一章练习题中已经借助 log crate 实现了日志功能, 那么你可以直接在 easy-fs 中引入 log crate, 通过 log::info!/debug! 等宏即可进行调试并在内核中看到日志输出。具体来说, 在 easy-fs 中的修改是: 在 easy-fs/Cargo.toml 的依赖中加入一行 log = "0.4.0", 然后在 easy-fs/src/lib.rs 中加入一行 extern crate log。

你也可以完全在用户态进行调试。仿照 easy-fs-fuse 建立一个在当前操作系统中运行的应用程序, 将测试逻辑写在 main 函数中。这个时候就可以将它引用的 easy-fs 的 no\_std 去掉并使用 println! 进行调试。

---

## 问答作业

无

## 实验练习的提交报告要求

- 简单总结本次实验与上个实验相比你增加的东西。(控制在 5 行以内, 不要贴代码)
- 完成问答问题
- (optional) 你对本次实验设计及难度的看法。

## 7.6 练习参考答案

### 7.6.1 课后练习

#### 编程题

1. \* 扩展 easy-fs 文件系统功能, 扩大单个文件的大小, 支持三级间接 inode。

在修改之前, 先看看原始 inode 的结构:

```
/// The max number of direct inodes
const INODE_DIRECT_COUNT: usize = 28;

#[repr(C)]
pub struct DiskInode {
 pub size: u32,
 pub direct: [u32; INODE_DIRECT_COUNT],
 pub indirect1: u32,
 pub indirect2: u32,
 type_: DiskInodeType,
}

#[derive(PartialEq)]
pub enum DiskInodeType {
 File,
 Directory,
}
```

一个 DiskInode 在磁盘上占据 128 字节的空间。我们考虑加入 indirect3 字段并缩减 INODE\_DIRECT\_COUNT 为 27 以保持 DiskInode 的大小不变。此时直接索引可索引 13.5KiB 的内容, 一级间接索引和二级间接索引仍然能索引 64KiB 和 8MiB 的内容, 而三级间接索引能索引  $128 * 8\text{MiB} = 1\text{GiB}$  的内容。当文件大小大于  $13.5\text{KiB} + 64\text{KiB} + 8\text{MiB}$  时, 需要用到三级间接索引。

下面的改动都集中在 easy-fs/src/layout.rs 中。首先修改 DiskInode 和相关的常量定义。

```
pub struct DiskInode {
 pub size: u32,
 pub direct: [u32; INODE_DIRECT_COUNT],
 pub indirect1: u32,
 pub indirect2: u32,
 pub indirect3: u32,
 type_: DiskInodeType,
}
```

在计算给定文件大小对应的块总数时, 需要新增对三级间接索引的处理。三级间接索引的存在使得二级间接索引所需的块数不再计入所有的剩余数据块。

```

pub fn total_blocks(size: u32) -> u32 {
 let data_blocks = Self::_data_blocks(size) as usize;
 let mut total = data_blocks as usize;
 // indirect1
 if data_blocks > INODE_DIRECT_COUNT {
 total += 1;
 }
 // indirect2
 if data_blocks > INDIRECT1_BOUND {
 total += 1;
 // sub indirect1
 let level2_extra =
 (data_blocks - INDIRECT1_BOUND + INODE_INDIRECT1_COUNT - 1) / INODE_
 ↪INDIRECT1_COUNT;
 total += level2_extra.min(INODE_INDIRECT1_COUNT);
 }
 // indirect3
 if data_blocks > INDIRECT2_BOUND {
 let remaining = data_blocks - INDIRECT2_BOUND;
 let level2_extra = (remaining + INODE_INDIRECT2_COUNT - 1) / INODE_INDIRECT2_
 ↪COUNT;
 let level3_extra = (remaining + INODE_INDIRECT1_COUNT - 1) / INODE_INDIRECT1_
 ↪COUNT;
 total += 1 + level2_extra + level3_extra;
 }
 total as u32
}

```

DiskInode 的 get\_block\_id 方法中遇到三级间接索引要额外读取三次块缓存。

```

pub fn get_block_id(&self, inner_id: u32, block_device: &Arc<dyn BlockDevice>) -> u32
{
 let inner_id = inner_id as usize;
 if inner_id < INODE_DIRECT_COUNT {
 // ...
 } else if inner_id < INDIRECT1_BOUND {
 // ...
 } else if inner_id < INDIRECT2_BOUND {
 // ...
 } else { // 对三级间接索引的处理
 let last = inner_id - INDIRECT2_BOUND;
 let indirect1 = get_block_cache(self.indirect3 as usize, Arc::clone(block_
 ↪device));
 .lock()
 .read(0, |indirect3: &IndirectBlock| {
 indirect3[last / INODE_INDIRECT2_COUNT]
 });
 let indirect2 = get_block_cache(indirect1 as usize, Arc::clone(block_device))
 .lock()
 .read(0, |indirect2: &IndirectBlock| {
 indirect2[(last % INODE_INDIRECT2_COUNT) / INODE_INDIRECT1_COUNT]
 });
 get_block_cache(indirect2 as usize, Arc::clone(block_device))
 .lock()
 .read(0, |indirect1: &IndirectBlock| {
 indirect1[(last % INODE_INDIRECT2_COUNT) % INODE_INDIRECT1_COUNT]
 })
 }
}

```

(下页继续)

(续上页)

```

 }
}
```

方法 `increase_size` 的实现本身比较繁琐，如果按照原有的一级和二级间接索引的方式实现对三级间接索引的处理，代码会比较丑陋。实际上多重间接索引是树结构，变量 `current_blocks` 和 `total_blocks` 对应着当前树的叶子数量和目标叶子数量，我们可以用递归函数来实现树的生长。先实现以下的辅助方法：

```

/// Helper to build tree recursively
/// extend number of leaves from `src_leaf` to `dst_leaf`
fn build_tree(
 &self,
 blocks: &mut alloc::vec::IntoIter<u32>,
 block_id: u32,
 mut cur_leaf: usize,
 src_leaf: usize,
 dst_leaf: usize,
 cur_depth: usize,
 dst_depth: usize,
 block_device: &Arc<dyn BlockDevice>,
) -> usize {
 if cur_depth == dst_depth {
 return cur_leaf + 1;
 }
 get_block_cache(block_id as usize, Arc::clone(block_device))
 .lock()
 .modify(0, |indirect_block: &mut IndirectBlock| {
 let mut i = 0;
 while i < INODE_INDIRECT1_COUNT && cur_leaf < dst_leaf {
 if cur_leaf >= src_leaf {
 indirect_block[i] = blocks.next().unwrap();
 }
 cur_leaf = self.build_tree(
 blocks,
 indirect_block[i],
 cur_leaf,
 src_leaf,
 dst_leaf,
 cur_depth + 1,
 dst_depth,
 block_device,
);
 i += 1;
 }
 });
 cur_leaf
}
```

然后修改方法 `increase_size`。不要忘记在填充二级间接索引时维护 `current_blocks` 的变化，并限制目标索引 (a1, b1) 的范围。

```

/// Increase the size of current disk inode
pub fn increase_size(
 &mut self,
 new_size: u32,
 new_blocks: Vec<u32>,
 block_device: &Arc<dyn BlockDevice>,
```

(下页继续)

(续上页)

```

) {
 // ...
 // alloc indirect2
 // ...
 // fill indirect2 from (a0, b0) -> (a1, b1)
 // 不要忘记限制 (a1, b1) 的范围
 // ...
 // alloc indirect3
 if total_blocks > INODE_INDIRECT2_COUNT as u32 {
 if current_blocks == INODE_INDIRECT2_COUNT as u32 {
 self.indirect3 = new_blocks.next().unwrap();
 }
 current_blocks -= INODE_INDIRECT2_COUNT as u32;
 total_blocks -= INODE_INDIRECT2_COUNT as u32;
 } else {
 return;
 }
 // fill indirect3
 self.build_tree(
 &mut new_blocks,
 self.indirect3,
 0,
 current_blocks as usize,
 total_blocks as usize,
 0,
 3,
 block_device,
);
}

```

对方法 clear\_size 的修改与 increase\_size 类似。先实现辅助方法 collect\_tree\_blocks:

```

/// Helper to recycle blocks recursively
fn collect_tree_blocks(
 &self,
 collected: &mut Vec<u32>,
 block_id: u32,
 mut cur_leaf: usize,
 max_leaf: usize,
 cur_depth: usize,
 dst_depth: usize,
 block_device: &Arc<dyn BlockDevice>,
) -> usize {
 if cur_depth == dst_depth {
 return cur_leaf + 1;
 }
 get_block_cache(block_id as usize, Arc::clone(block_device))
 .lock()
 .read(0, |indirect_block: &IndirectBlock| {
 let mut i = 0;
 while i < INODE_INDIRECT1_COUNT && cur_leaf < max_leaf {
 collected.push(indirect_block[i]);
 cur_leaf = self.collect_tree_blocks(
 collected,
 indirect_block[i],
 cur_leaf,
 max_leaf,
);
 }
 });
}

```

(下页继续)

(续上页)

```

 cur_depth + 1,
 dst_depth,
 block_device,
);
 i += 1;
}
});
cur_leaf
}
}

```

然后修改方法 clear\_size。

```

/// Clear size to zero and return blocks that should be deallocated.
/// We will clear the block contents to zero later.
pub fn clear_size(&mut self, block_device: &Arc<dyn BlockDevice>) -> Vec<u32> {
 // ...
 // indirect2 block
 // ...
 // indirect2
 // 不要忘记限制 (a1, b1) 的范围
 self.indirect2 = 0;
 // indirect3 block
 assert!(data_blocks <= INODE_INDIRECT3_COUNT);
 if data_blocks > INODE_INDIRECT2_COUNT {
 v.push(self.indirect3);
 data_blocks -= INODE_INDIRECT2_COUNT;
 } else {
 return v;
 }
 // indirect3
 self.collect_tree_blocks(&mut v, self.indirect3, 0, data_blocks, 0, 3, block_
device);
 self.indirect3 = 0;
 v
}
}

```

接下来你可以在 easy-fs-fuse/src/main.rs 中测试 easy-fs 文件系统的修改，比如读写大小超过 10MiB 的文件。

2. \* 扩展内核功能，支持 stat 系统调用，能显示文件的 inode 元数据信息。

你将在本章的编程实验中实现这个功能。

3. \*\* 扩展内核功能，支持 mmap 系统调用，支持对文件的映射，实现基于内存读写方式的文件读写功能。

**注解：**这里只是给出了一种参考实现。mmap 本身行为比较复杂，使用你认为合理的方式实现即可。

在第四章的编程实验中你应该已经实现了 mmap 的匿名映射功能，这里我们要实现文件映射。mmap 的原型如下：

```

void *mmap(void *addr, size_t length, int prot, int flags,
 int fd, off_t offset);

```

其中 addr 是一个虚拟地址的 hint，在映射文件时我们不关心具体的虚拟地址（相当于传入 NULL），这里我们的系统调用忽略这个参数。prot 和 flags 指定了一些属性，为简单起见我们也不要这两个参数，映射的虚拟内存的属性直接继承自文件的读写属性。我们最终保留 length、fd 和 offset 三个参数。

考虑最简单的一种实现方式：mmap 调用时随便选择一段虚拟地址空间，将它映射到一些随机的物理页面上，之后再把文件的对应部分全部读到内存里。如果这段映射是可写的，那么内核还要在合适的时机（比如调用 msync、munmap、进程退出时）把内存里的东西回写到文件。

这样做的问题是被映射的文件可能很大，将映射的区域全部读入内存可能很慢，而且用户未必会访问所有的页面。这里可以应用按需分页的惰性加载策略：先不实际建立虚拟内存到物理内存的映射，当用户访问映射的区域时会触发缺页异常，我们在处理异常时分配实际的物理页面并将文件读入内存。

按照上述方式已经可以实现文件映射了，但让我们来考虑较为微妙的情况。比如以下的 Linux C 程序：

```
#include <unistd.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <stdio.h>

int main()
{
 char str[] = {"asdbasdq3423423\n"};
 int fd = open("2.txt", O_RDWR | O_CREAT | O_TRUNC, 0664);
 if (fd < 0) {
 printf("open failed\n");
 return -1;
 }

 if (write(fd, str, sizeof(str)) < 0) {
 printf("write failed\n");
 return -1;
 }

 char *p1 = mmap(NULL, sizeof(str), PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
 char *p2 = mmap(NULL, sizeof(str), PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
 printf("p1 = %p, p2 = %p\n", p1, p2);
 close(fd);

 p1[1] = '1';
 p2[2] = '2';
 p2[0] = '2';
 p1[0] = '1';
 printf("content1: %s", p1);
 printf("content2: %s", p2);
 return 0;
}
```

一个可能的输出结果如下：

```
p1 = 0x7f955a3cf000, p2 = 0x7f955a3a2000
content1: 112basdq3423423
content2: 112basdq3423423
```

可以看到文件的同一段区域被映射到了两个不同的虚拟地址，对这两段虚拟内存的修改全部生效（冲突的修改也是最后的可见），修改后再读出来的内容也相同。这样的结果是符合直觉的，因为底层的文件只有一个（也与 MAP\_SHARED 有关，由于设置 MAP\_PRIVATE 标志不会将修改真正写入文件，我们参考 MAP\_SHARED 的行为）。如果按照上面说的方式将两个虚拟内存区域映射到不同的物理页面，那么对两个区域的修改无法同时生效，我们也无法确定应该将哪个页面回写到文件。这个例子启示我们，**如果文件映射包含文件的相同部分，那么相应的虚拟页面应该映射到相同的物理页。**

不幸的是，现有的 MapArea 类型只含 Identical 和 Framed，不支持不同的虚拟页面共享物理页，所以我们需要手动管理一些资源。下面的 FileMapping 结构描述了一个文件的若干段映射：

```
pub struct FileMapping {
 file: Arc<Inode>,
 ranges: Vec<MapRange>,
 frames: Vec<FrameTracker>,
 dirty_parts: BTreeSet<usize>, // file segments that need writing back
 map: BTreeMap<usize, PhysPageNum>, // file offset -> ppn
}
```

其中 `file` 代表被映射的文件，你可能会好奇它的类型为什么不是一个文件描述符编号或者 `Arc<dyn File>`。首先 `mmap` 之后使用的文件描述符可以立即被关闭而不会对文件映射造成任何影响，所以不适合只存放 `fd` 编号；其次 `mmap` 通常要求映射的文件是常规文件（例：映射 `stdin` 和 `stdout` 毫无意义），这里用 `Inode` 来提醒我们这点。`ranges` 里面存放了若干 `MapRange`，每个都用于描述一段映射区域。`frames` 用于管理实际分配的物理页帧。`dirty_parts` 记录了需要回写的脏页，注意它实际上用文件内的偏移来表示。`map` 维护文件内偏移到物理页号的映射。需要注意的是这里记录脏页的方式比较简单，而且也完全没有考虑在进程间共享物理页，你可以使用引用计数等手段进行扩展。

```
#[derive(Clone)]
struct MapRange {
 start: VirtAddr,
 len: usize, // length in bytes
 offset: usize, // offset in file
 perm: MapPermission,
}
```

`MapRange` 描述了一段映射区域。`start` 是该区域的起始虚拟地址，`offset` 为其在文件中的偏移，`perm` 记录了该区域的属性。

前面提到过，我们的 `mmap` 忽略掉作为 `hint` 的 `addr` 参数，那这里的虚拟地址填什么呢？一般来说 64 位架构具有大到用不完的虚拟地址空间，用一个简单的线性分配器随便分配虚拟地址即可。

```
/// Base virtual address for mmap
pub const MMAP_AREA_BASE: usize = 0x0000_0001_0000_0000; // 随便选的基址，挑块没人用的

/// A naive linear virtual address space allocator
pub struct VirtualAddressAllocator {
 cur_va: VirtAddr,
}

impl VirtualAddressAllocator {
 /// Create a new allocator with given base virtual address
 pub fn new(base: usize) -> Self {
 Self {
 cur_va: base.into(),
 }
 }

 /// Allocate a virtual address area
 pub fn alloc(&mut self, len: usize) -> VirtAddr {
 let start = self.cur_va;
 let end: VirtAddr = (self.cur_va.0 + len).into();
 self.cur_va = end.ceil().into();
 start
 }

 // 不必释放
}
```

然后把 `VirtualAddressAllocator` 和 `FileMapping` 放进 `TaskControlBlockInner` 里。为简单起

见, fork 时不考虑这两个字段的复制和映射的共享。

列表 1: os/src/task/task.rs

```
pub struct TaskControlBlockInner {
 pub trap_cx_ppn: PhysPageNum,
 pub base_size: usize,
 pub task_cx: TaskContext,
 pub task_status: TaskStatus,
 pub memory_set: MemorySet,
 pub parent: Option<Weak<TaskControlBlock>>,
 pub children: Vec<Arc<TaskControlBlock>>,
 pub exit_code: i32,
 pub fd_table: Vec<Option<Arc<dyn File + Send + Sync>>>,
 pub mmap_va_allocator: VirtualAddressAllocator,
 pub file_mappings: Vec<FileMapping>,
}
```

下面来添加 mmap 系统调用:

```
/// This is a simplified version of mmap which only supports file-backed mapping
pub fn sys_mmap(fd: usize, len: usize, offset: usize) -> isize {
 if len == 0 {
 // invalid length
 return -1;
 }
 if (offset & (PAGE_SIZE - 1)) != 0 {
 // offset must be page size aligned
 return -1;
 }

 let task = current_task().unwrap();
 let mut tcb = task.inner_exclusive_access();
 if fd >= tcb.fd_table.len() {
 return -1;
 }
 if tcb.fd_table[fd].is_none() {
 return -1;
 }

 let fp = tcb.fd_table[fd].as_ref().unwrap();
 let opt_inode = fp.as_any().downcast_ref::<OSInode>();
 if opt_inode.is_none() {
 // must be a regular file
 return -1;
 }

 let inode = opt_inode.unwrap();
 let perm = parse_permission(inode);
 let file = inode.clone_inner_inode();
 if offset >= file.get_size() {
 // file offset exceeds size limit
 return -1;
 }

 let start = tcb.mmap_va_allocator.alloc(len);
 let mappings = &mut tcb.file_mappings;
 if let Some(m) = find_file_mapping(mappings, &file) {
```

(下页继续)

(续上页)

```

 m.push(start, len, offset, perm);
 } else {
 let mut m = FileMapping::new_empty(file);
 m.push(start, len, offset, perm);
 mappings.push(m);
 }
 start.0 as isize
}

```

这里面有不少无聊的参数检查和辅助函数，就不详细介绍了。总之这个系统调用实际做的事情只有维护对应的 FileMapping 结构，实际的工作被推迟到缺页异常处理例程中。

列表 2: os/src/trap/mod.rs

```

#[no_mangle]
/// handle an interrupt, exception, or system call from user space
pub fn trap_handler() -> ! {
 set_kernel_trap_entry();
 let scause = scause::read();
 let stval = stval::read();
 match scause.cause() {
 Trap::Exception(Exception::UserEnvCall) => {
 // ...
 }
 Trap::Exception(Exception::StoreFault)
 | Trap::Exception(Exception::StorePageFault)
 | Trap::Exception(Exception::InstructionFault)
 | Trap::Exception(Exception::InstructionPageFault)
 | Trap::Exception(Exception::LoadFault)
 | Trap::Exception(Exception::LoadPageFault) => {
 if !handle_page_fault(stval) {
 println!(
 "[kernel] {:?} in application, bad addr = {:#x}, bad instruction= {:#x}, kernel killed it.",
 scause.cause(),
 stval,
 current_trap_cx().sepc,
);
 // page fault exit code
 exit_current_and_run_next(-2);
 }
 }
 Trap::Exception(Exception::IllegalInstruction) => {
 // ...
 }
 Trap::Interrupt(Interrupt::SupervisorTimer) => {
 // ...
 }
 -=> {
 panic!(
 "Unsupported trap {:?}, stval = {:#x}!",
 scause.cause(),
 stval
);
 }
 }
}

```

(下页继续)

(续上页)

```

 trap_return();
}

```

我们在这里尝试处理缺页异常，如果 handle\_page\_fault 返回 true 表明异常已经被处理，否则内核仍然会杀死当前进程。

```

1 /// Try to handle page fault caused by demand paging
2 /// Returns whether this page fault is fixed
3 pub fn handle_page_fault(fault_addr: usize) -> bool {
4 let fault_va: VirtAddr = fault_addr.into();
5 let fault_vpn = fault_va.floor();
6 let task = current_task().unwrap();
7 let mut tcb = task.inner_exclusive_access();
8
9 if let Some(pte) = tcb.memory_set.translate(fault_vpn) {
10 if pte.is_valid() {
11 return false; // fault va already mapped, we cannot handle this
12 }
13 }
14
15 match tcb.file_mappings.iter_mut().find(|m| m.contains(fault_va)) {
16 Some(mapping) => {
17 let file = Arc::clone(&mapping.file);
18 // fix vm mapping
19 let (ppn, range, shared) = mapping.map(fault_va).unwrap();
20 tcb.memory_set.map(fault_vpn, ppn, range.perm);
21
22 if !shared {
23 // load file content
24 let file_size = file.get_size();
25 let file_offset = range.file_offset(fault_vpn);
26 assert!(file_offset < file_size);
27
28 // let va_offset = range.va_offset(fault_vpn);
29 // let va_len = range.len - va_offset;
30 // Note: we do not limit `read_len` with `va_len`
31 // consider two overlapping areas with different lengths
32
33 let read_len = PAGE_SIZE.min(file_size - file_offset);
34 file.read_at(file_offset, &mut ppn.get_bytes_array()[..read_len]);
35 }
36 true
37 }
38 None => false,
39 }
40 }

```

- handle\_page\_fault 的 9~13 行先检查触发异常的虚拟内存页是否已经映射到物理页面，如果是则说明此异常并非源自惰性按需分页（比如写入只读页），这个问题不归我们管，直接返回 false。
- 接下来的第 15 行检查出错的虚拟地址是否在映射区域内，如果是我们才上手来处理。

在实际的修复过程中： - 第 19 行先调用 FileMapping 的 map 方法建立目标虚拟地址到物理页面的映射； - 第 20 行将新的映射关系添加到页表； - 第 22~35 行处理文件读入。注意实际的文件读取只发生在物理页面的引用计数从 0 变为 1 的时候，存在共享的情况下再读取文件可能会覆盖掉用户对内存的修改。

FileMapping 的 map 方法实现如下：

```

1 impl FileMapping {
2 /// Create mapping for given virtual address
3 fn map(&mut self, va: VirtAddr) -> Option<(PhysPageNum, MapRange, bool)> {
4 // Note: currently virtual address ranges never intersect
5 let vpn = va.floor();
6 for range in &self.ranges {
7 if !range.contains(va) {
8 continue;
9 }
10 let offset = range.file_offset(vpn);
11 let (ppn, shared) = match self.map.get(&offset) {
12 Some(&ppn) => (ppn, true),
13 None => {
14 let frame = frame_alloc().unwrap();
15 let ppn = frame.ppn;
16 self.frames.push(frame);
17 self.map.insert(offset, ppn);
18 (ppn, false)
19 }
20 };
21 if range.perm.contains(MapPermission::W) {
22 self.dirty_parts.insert(offset);
23 }
24 return Some((ppn, range.clone(), shared));
25 }
26 None
27 }
28 }
```

- 第 6~9 行先找到包含目标虚拟地址的映射区域；
- 第 10 行计算虚拟地址对应的文件内偏移；
- 第 11~20 行先查询此文件偏移是否对应已分配的物理页，如果没有则分配一个物理页帧并记录映射关系；
- 第 21~23 行检查此映射区域是否有写入权限，如果有则将对应的物理页面标记为脏页。这个处理实际上比较粗糙，有些没有被真正写入的页面也被视为脏页，导致最后会有多余的文件回写。你也可以考虑不维护脏页信息，而是通过检查页表项中由硬件维护的 Dirty 位来确定哪些是真正的脏页。

修复后用户进程重新执行触发缺页异常的指令，此时物理页里存放了文件的内容，这样用户就实现了以读取内存的方式来读取文件。最后来处理被修改的脏页的同步，给 FileMapping 添加 sync 方法：

```

1 impl FileMapping {
2 /// Write back all dirty pages
3 pub fn sync(&self) {
4 let file_size = self.file.get_size();
5 for &offset in self.dirty_parts.iter() {
6 let ppn = self.map.get(&offset).unwrap();
7 if offset < file_size {
8 // WARNING: this can still cause garbage written
9 // to file when sharing physical page
10 let va_len = self
11 .ranges
12 .iter()
13 .map(|r| {
14 if r.offset <= offset && offset < r.offset + r.len {
15 PAGE_SIZE.min(r.offset + r.len - offset)
16 }
17 })
18 }
19 }
20 }
21 }
```

(下页继续)

(续上页)

```

16 } else {
17 0
18 }
19 })
20 .max()
21 .unwrap();
22 let write_len = va_len.min(file_size - offset);
23
24 self.file
25 .write_at(offset, &ppn.get_bytes_array()[..write_len]);
26 }
27 }
28 }
29 }
```

这个方法将所有潜在的脏物理页内容回写至文件。第 10~22 行的计算主要为了限制写入内容的长度，以避免垃圾被意外写入文件。

剩下的问题是何时调用 sync。正常来说 munmap、msync 是同步点，你可以自行实现这两个系统调用，这里我们把它放在进程退出之前：

列表 3: os/src/task/mod.rs

```

/// Exit the current 'Running' task and run the next task in task list.
pub fn exit_current_and_run_next(exit_code: i32) {
 let task = take_current_task().unwrap();
 // ...
 let mut inner = task.inner_exclusive_access();
 // ...
 inner.children.clear();
 // deallocate user space
 inner.memory_set.recycle_data_pages();
 // write back dirty pages
 for mapping in inner.file_mappings.iter() {
 mapping.sync();
 }
 drop(inner);
 // **** release current PCB
 // drop task manually to maintain rc correctly
 drop(task);
 // ...
}
```

这样我们就实现了基于内存读写方式的文件读写功能。可以看到 mmap 不是魔法，内核悄悄帮你完成了实际的文件读写。

4. \*\* 扩展 easy-fs 文件系统功能，支持二级目录结构。可扩展：支持 N 级目录结构。

实际上 easy-fs 现有的代码支持目录的存在，只不过整个文件系统只有根目录一个目录，我们考虑放宽现有代码的一些限制。

原本的 easy-fs/src/vfs.rs 中有一个用于在当前目录下创建常规文件的 create 方法，我们给它加个参数并包装一下：

列表 4: easy-fs/src/vfs.rs

```
impl Inode {
```

(下页继续)

(续上页)

```

 /// Create inode under current inode by name
 fn create_inode(&self, name: &str, inode_type: DiskInodeType) -> Option<Arc<Inode>
 -> {
 let mut fs = self.fs.lock();
 let op = |root_inode: &DiskInode| {
 // assert it is a directory
 assert!(root_inode.is_dir());
 // has the file been created?
 self.find_inode_id(name, root_inode)
 };
 if self.read_disk_inode(op).is_some() {
 return None;
 }
 // create a new file
 // alloc a inode with an indirect block
 let new_inode_id = fs.alloc_inode();
 // initialize inode
 let (new_inode_block_id, new_inode_block_offset) = fs.get_disk_inode_pos(new_
 ->inode_id);
 get_block_cache(new_inode_block_id as usize, Arc::clone(&self.block_device))
 .lock()
 .modify(new_inode_block_offset, |new_inode: &mut DiskInode| {
 new_inode.initialize(inode_type);
 });
 self.modify_disk_inode(|root_inode| {
 // append file in the dirent
 let file_count = (root_inode.size as usize) / DIRENT_SZ;
 let new_size = (file_count + 1) * DIRENT_SZ;
 // increase size
 self.increase_size(new_size as u32, root_inode, &mut fs);
 // write dirent
 let dirent = DirEntry::new(name, new_inode_id);
 root_inode.write_at(
 file_count * DIRENT_SZ,
 dirent.as_bytes(),
 &self.block_device,
);
 });
 });

 let (block_id, block_offset) = fs.get_disk_inode_pos(new_inode_id);
 block_cache_sync_all();
 // return inode
 Some(Arc::new(Self::new(
 block_id,
 block_offset,
 self.fs.clone(),
 self.block_device.clone(),
)));
 // release efs lock automatically by compiler
}

/// Create regular file under current inode
pub fn create(&self, name: &str) -> Option<Arc<Inode>> {
 self.create_inode(name, DiskInodeType::File)
}

```

(下页继续)

(续上页)

```
/// Create directory under current inode
pub fn create_dir(&self, name: &str) -> Option<Arc<Inode>> {
 self.create_inode(name, DiskInodeType::Directory)
}
```

这样我们就可以在一个目录底下调用 `create_dir` 创建新目录了 (笑)。本质上我们什么也没改，我们再改其它方法装装样子：

列表 5: easy-fs/src/vfs.rs

```
impl Inode {
 /// List inodes under current inode
 pub fn ls(&self) -> Vec<String> {
 let _fs = self.fs.lock();
 self.read_disk_inode(|disk_inode| {
 let mut v: Vec<String> = Vec::new();
 if disk_inode.is_file() {
 return v;
 }

 let file_count = (disk_inode.size as usize) / DIRENT_SZ;
 for i in 0..file_count {
 let mut dirent = DirEntry::empty();
 assert_eq!(
 disk_inode.read_at(i * DIRENT_SZ, dirent.as_bytes_mut(), &self.
 block_device),
 DIRENT_SZ,
);
 v.push(String::from(dirent.name()));
 }
 })
 }

 /// Write data to current inode
 pub fn write_at(&self, offset: usize, buf: &[u8]) -> usize {
 let mut fs = self.fs.lock();
 let size = self.modify_disk_inode(|disk_inode| {
 assert!(disk_inode.is_file());

 self.increase_size((offset + buf.len()) as u32, disk_inode, &mut fs);
 disk_inode.write_at(offset, buf, &self.block_device)
 });
 block_cache_sync_all();
 size
 }

 /// Clear the data in current inode
 pub fn clear(&self) {
 let mut fs = self.fs.lock();
 self.modify_disk_inode(|disk_inode| {
 assert!(disk_inode.is_file());

 let size = disk_inode.size;
 let data_blocks_dealloc = disk_inode.clear_size(&self.block_device);
 })
 }
}
```

(下页继续)

(续上页)

```

 assert!(data_blocks_dealloc.len() == DiskInode::total_blocks(size) as
 ↪usize);
 for data_block in data_blocks_dealloc.into_iter() {
 fs.dealloc_data(data_block);
 }
});
block_cache_sync_all();
}
}
}

```

对一个普通文件的 inode 调用 ls 方法毫无意义，但为了保持接口不变，我们返回一个空 Vec。随意地清空或写入目录文件都会损坏目录结构，这里直接在 write\_at 和 clear 方法中断言，你也可以改成其它的错误处理方式。

接下来是实际一点的修改（有，但不多）：我们让 find 方法支持简单的相对路径（不含“.” 和“..”）。

列表 6: easy-fs/src/vfs.rs

```

impl Inode {
 /// Find inode under current inode by **path**
 pub fn find(&self, path: &str) -> Option<Arc<Inode>> {
 let fs = self.fs.lock();
 let mut block_id = self.block_id as u32;
 let mut block_offset = self.block_offset;
 for name in path.split('/').filter(|s| !s.is_empty()) {
 let inode_id = get_block_cache(block_id as usize, self.block_device.
 ↪clone())
 .lock()
 .read(block_offset, |disk_inode: &DiskInode| {
 if disk_inode.is_file() {
 return None;
 }
 self.find_inode_id(name, disk_inode)
 });
 if inode_id.is_none() {
 return None;
 }
 (block_id, block_offset) = fs.get_disk_inode_pos(inode_id.unwrap());
 }
 Some(Arc::new(Self::new(
 block_id,
 block_offset,
 self.fs.clone(),
 self.block_device.clone(),
)))
 }
}

```

最后在 easy-fs-fuse/src/main.rs 里试试我们添加的新特性：

列表 7: easy-fs-fuse/src/main.rs

```

fn read_string(file: &Arc<Inode>) -> String {
 let mut read_buffer = [0u8; 512];
 let mut offset = 0usize;
 let mut read_str = String::new();
 loop {

```

(下页继续)

(续上页)

```

let len = file.read_at(offset, &mut read_buffer);
if len == 0 {
 break;
}
offset += len;
read_str.push_str(core::str::from_utf8(&read_buffer[..len]).unwrap());
}
read_str
}

fn tree(inode: &Arc<Inode>, name: &str, depth: usize) {
 for _ in 0..depth {
 print!(" ");
 }
 println!("{} {}", name);
 for name in inode.ls() {
 let child = inode.find(&name).unwrap();
 tree(&child, &name, depth + 1);
 }
}

#[test]
fn efs_dir_test() -> std::io::Result<()> {
 let block_file = Arc::new(BlockFile(Mutex::new({
 let f = OpenOptions::new()
 .read(true)
 .write(true)
 .create(true)
 .open("target/fs.img")?;
 f.set_len(8192 * 512).unwrap();
 f
 })));
 EasyFileSystem::create(block_file.clone(), 4096, 1);
 let efs = EasyFileSystem::open(block_file.clone());
 let root = Arc::new(EasyFileSystem::root_inode(&efs));
 root.create("f1");
 root.create("f2");

 let d1 = root.create_dir("d1").unwrap();

 let f3 = d1.create("f3").unwrap();
 let d2 = d1.create_dir("d2").unwrap();

 let f4 = d2.create("f4").unwrap();
 tree(&root, "/", 0);

 let f3_content = "3333333";
 let f4_content = "444444444444444444";
 f3.write_at(0, f3_content.as_bytes());
 f4.write_at(0, f4_content.as_bytes());

 assert_eq!(read_string(&d1.find("f3").unwrap()), f3_content);
 assert_eq!(read_string(&root.find("/d1/f3").unwrap()), f3_content);
 assert_eq!(read_string(&d2.find("f4").unwrap()), f4_content);
 assert_eq!(read_string(&d1.find("d2/f4").unwrap()), f4_content);
 assert_eq!(read_string(&root.find("/d1/d2/f4").unwrap()), f4_content);
}

```

(下页继续)

(续上页)

```

assert!(f3.find("whatever").is_none());
Ok(())
}

```

如果你觉得这个练习不够过瘾，可以试试下面的任务：

- 让 easy-fs 支持包含“.”和“..”的相对路径。你可以在目录文件里存放父目录的 inode。
- 在内核里给进程加上当前路径信息，然后实现 chdir 和 getcwd。当然，也可以顺便补上 openat 和 mkdir。
- 在 easy-fs 中实现 rename 和 mv 的功能。在目录文件中删掉一些目录项也许要实现 decrease\_size 或者类似删除的东西，但也可以考虑用删除标记这种常见的手段让一个目录项变得“不存在”。

## 问答题

1. \* 文件系统的功能是什么？

将数据以文件的形式持久化保存在存储设备上。

2. \*\* 目前的文件系统只有单级目录，假设想要支持多级文件目录，请描述你设想的实现方式，描述合理即可。

允许在目录项中存在目录（原本只能存在普通文件）即可。

3. \*\* 软链接和硬链接是干什么的？有什么区别？当删除一个软链接或硬链接时分别会发生什么？

软硬链接的作用都是给一个文件以“别名”，使得不同的多个路径可以指向同一个文件。当删除软链接时候，对文件没有任何影响，当删除硬链接时，文件的引用计数会被减一，若引用计数为0，则该文件所占据的磁盘空间将会被回收。

4. \*\*\* 在有了多级目录之后，我们就可以为一个目录增加硬链接了。在这种情况下，文件树中是否可能出现环路（软硬链接都可以，鼓励多尝试）？你认为应该如何解决？请在你喜欢的系统上实现一个环路，描述你的实现方式以及系统提示、实际测试结果。

是可以出现环路的，一种可能的解决方式是在访问文件的时候检查自己遍历的路径中是否有重复的 inode，并在发现环路时返回错误。

5. \* 目录是一类特殊的文件，存放的是什么内容？用户可以自己修改目录内容吗？

存放的是目录中的文件列表以及他们对应的 inode，通常而言用户不能自己修改目录的内容，但是可以通过操作目录（如 mv 里面的文件）的方式间接修改。

6. \*\* 在实际操作系统中，如 Linux，为什么会存在大量的文件系统类型？

因为不同的文件系统有着不同的特性，比如对于特定种类的存储设备的优化，或是快照和多设备管理等高级特性，适用于不同的使用场景。

7. \*\* 可以把文件控制块放到目录项中吗？这样做有什么优缺点？

可以，是对于小目录可以减少一次磁盘访问，提升性能，但是对大目录而言会使得在目录中查找文件的性能降低。

8. \*\* 为什么要同时维护进程的打开文件表和操作系统的打开文件表？这两个打开文件表有什么区别和联系？

多个进程可能会同时打开同一个文件，操作系统级的打开文件表可以加快后续的打开操作，但同时由于每个进程打开文件时使用的访问模式或是偏移量不同，所以还需要进程的打开文件表另外记录。

9. \*\* 文件分配的三种方式是如何组织文件数据块的？各有什么特征（存储、文件读写、可靠性）？

连续分配：实现简单、存取速度快，但是难以动态增加文件大小，长期使用后会产生大量无法使用（过小而无法放入大文件）碎片空间。

链接分配：可以处理文件大小的动态增长，也不会出现碎片，但是只能按顺序访问文件中的块，同时一旦有一个块损坏，后面的其他块也无法读取，可靠性差。

索引分配：可以随机访问文件中的偏移量，但是对于大文件需要实现多级索引，实现较为复杂。

10. \*\* 如果一个程序打开了一个文件，写入了一些数据，但是没有及时关闭，可能会有什么后果？如果打开文件后，又进一步发出了读文件的系统调用，操作系统中各个组件是如何相互协作完成整个读文件的系统调用的？

(若也没有 flush 的话) 假如此时操作系统崩溃，尚处于内存缓冲区中未写入磁盘的数据将会丢失，同时也会占用文件描述符，造成资源的浪费。首先是系统调用处理的部分，将这一请求转发给文件系统子系统，文件系统子系统再将其转发给块设备子系统，最后再由块设备子系统转发给实际的磁盘驱动程序读取数据，最终返回给程序。

11. \*\*\* 文件系统是一个操作系统必要的组件吗？是否可以将文件系统放到用户态？这样做有什么好处？操作系统需要提供哪些基本支持？

不是，如在本章之前的 rCore 就没有文件系统。可以，如在 Linux 下就有 FUSE 这样的框架可以实现这一点。这样可以使得文件系统的实现更为灵活，开发与调试更为简便。操作系统需要提供一个注册用户态文件系统实现的机制，以及将收到的文件系统相关系统调用转发给注册的用户态进程的支持。



---

## 第七章：进程间通信与 I/O 重定向

---

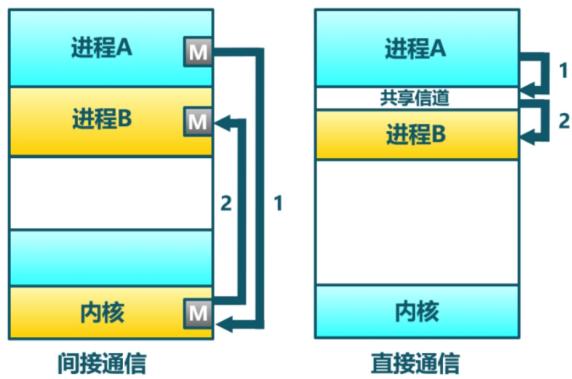
### 8.1 引言

#### 8.1.1 本章导读

在前面章节中，我们引入了非常重要的进程的概念，且实现的“伤齿龙”操作系统支持应用的动态创建进程和交互能力。这涉及到与进程管理相关的 `fork`、`exec` 等创建新进程相关的系统调用。同时，我们进一步引入了文件的抽象，使得进程能够通过一个统一的接口来读写内核管理的持久存储设备，这样数据可以方便地被长久保存。

截止到目前为止，进程在输入和输出方面，还有不少限制，特别是进程能够进行交互的 I/O 资源还非常有限，只能接受用户在键盘上的输入，并将字符输出到屏幕上。我们一般将它们分别称为 **标准输入** 和 **标准输出**。更让应用开发者觉得束手束脚的是：进程被操作系统彻底隔离了。进程间无法方便地“沟通”，导致进程不能一起协作干“大事”。如果能让不同进程实现数据共享与交互，就能把不同程序的功能组合在一起，实现更加强大和灵活的功能。为了让简单的应用程序能够组合在一起形成各种强大和复杂的功能，本章要完成的操作系统的核心目标是：**让不同应用通过进程间通信的方式组合在一起运行**。

其实在 **UNIX** 的早期发展历史中，也碰到了同样的问题，每个程序专注在完成一件事情上，但缺少把多个程序联合在一起完成复杂功能的机制。直到 1975 年 **UNIX v6** 中引入了让人眼前一亮的创新机制—**I/O 重定向与管道 (pipe)**。基于这两种机制，操作系统在不用改变应用程序的情况下，可以将一个程序的输出重新定向到另外一个程序的输入中，这样程序之间就可以进行任意的连接，并组合出各种灵活的复杂功能。



**注解:** 管道 (pipe) 可能是 UNIX 中最引人注目的发明

管道的概念来自贝尔实验室的 Douglas McIlroy，他在 1964 年写的一份内部文件中，提出了把多个程序“像花园水管一样”串连并拧在一起的想法，这样数据就可以在不同程序中流动。大约在 1972 年下半年，Ken Thompson 在听了 Douglas McIlroy 关于管道的唠叨后，灵机一动，迅速把管道机制实现在 UNIX 中。管道是一种由操作系统提供的机制，它将一个程序的输出连接到另一个程序的输入。可以通过在 shell 程序中使用“|”符号来轻松访问和操作管道。管道是 UNIX 最重要的贡献之一，通过管道可以把功能单一的小程序灵活地组合起来实现各种复杂的功能，从而让 UNIX 的简单功能哲学（一次把一件事情做好）和复杂系统能力（通过基于管道的排列组合形成复杂能力）有机地融合在一起。

本章我们将引入新操作系统概念—管道，并进行实现，以支持进程间的 I/O 重定向功能，即让一个进程的输出成为另外一个进程的输入，从而让进程间能够有效地合作起来。除了键盘和屏幕这样的 标准输入 和 标准输出 之外，管道其实也可以看成是一种特殊的输入和输出，而前面讲解的 文件系统 中的对持久化存储数据的 抽象文件 (file) 也是一种存储设备的输入和输出。所以，我们可以把这三种输入输出都统一在 文件 (file) 这个抽象之中。这也体现了在 UNIX 操作系统中“一切皆文件” (Everything is a file) 的重要设计哲学。

**注解: 一切皆文件**

在 UNIX 经典论文《The UNIX TimeSharing System》中，里奇和汤普森就提出了“一切皆文件”的朴素思想，这个思想或多或少地受到了 Multics 操作系统的影响。UNIX 将普通文件、设备和虚拟的资源（比如管道等）通过目录统一在了一个递归的树形结构中。形成了一个统一的命名空间。UNIX 文件系统是一个挂载在 ROOT 的树形目录结构，每一个目录节点都可以挂载一棵子树。“一切皆文件”意味着这棵树上可以挂载一切。少其实意味着多，对文件进行 read 和 write 操作，事实上可以完成“任意”操作。

文件这一抽象概念透明化了文件、I/O 设备之间的差异，因为在进程看来无论是标准输出还是常规文件都是一种文件，可以通过同样的接口来读写。这不但可以统一对设备的访问方式，且让应用与外设之间的交互进行了解耦，简化了应用的开发，更进一步支持进程间通信，实现了应用功能的组合扩展。这就是文件的强大之处。

为了让应用能够基于 文件 这个抽象接口对不同 I/O 设备或 I/O 资源进行操作，我们就需要对 进程 这个概念进行扩展，让它能够管理 文件 这种抽象资源和接口。为了实现 一切皆文件 (Everything is a file) 的 UNIX 设计思想，之前的进程控制块中有一个 文件描述符表，在表中保存着多个文件记录信息。每个文件描述符是一个非负的索引值，即对应文件记录信息的条目在文件描述符表中的索引。这个阶段的文件还仅仅是指磁盘上的数据存储。我们要对文件范围进行扩充，从数据存储扩大到外设、管道这样的物理和虚拟资源。

为了统一表示 标准输入、标准输出、管道 和 数据存储 等，我们把支持 `File trait` 定义的接口的结构都称为文件。这样只要 标准输入、标准输出、管道 也基于统一的 `File trait` 接口实现自己的打开、关闭和读写等文件操作，就可以让进程来对自己进行管理了。从用户的角度来看，应用访问文件将很简单，它不太需要关注文件的具体细节，只需通过文件描述符，就可以对 文件 进行读写，从而完成接收键盘输入，向屏幕输出，两个进程之间进行数据传输的操作，以及对存储设备上的数据进行读写。

仅仅实现文件的统一抽象和支持进程间通信的管道机制，还不够灵活。因为这需要两个进程之间相互“知道”它们要通信，即它们不能独立存在。我们可以进一步扩展进程动态管理的机制，来实现独立应用之间的 I/O 重定向，从而可以让独立的应用之间能够灵活组合完成复杂功能。具体而言，这需要扩展与应用程序执行相关的 exec 系统调用，加入对程序运行参数的支持，并进一步改进了对 shell 程序自身的实现，加入对重定向符号 >、< 的识别和处理。这样我们也可以像 UNIX 中的 shell 程序一样，基于文件机制和管道机制实现灵活的 I/O 重定位功能，更加灵活地把独立的应用程序组合在一起实现复杂功能。

仅仅有支持数据传递的管道机制还不够便捷，进程间也需要更快捷的通知机制。而且操作系统也不仅仅是被动地接受来自进程的系统调用，它也需要有主动让进程响应它发出的通知的需求。这些都推动了一种信号 (Signal) 的事件通知机制的诞生。

---

**注解：**信号 (Signal) 可能是 UNIX 中最早实现的内核通知进程的机制

信号从 UNIX 的第一版本就已经存在了，由于其设计实现的细节和边界条件较多，容易出错，所以经历了许多次设计开发的迭代。在早期设计中，UNIX 通过不同的系统调用来设置对不同类型信号的捕获；在 UNIX 第二版，出现了让用户给进程发信号的 kill 命令；在 UNIX 第四版，进一步简化了系统调用设计，可通过一个系统调用来设置对所有信号的捕获；发展到 UNIX 第七版，信号的设计模型还是不够可靠，会出现信号可能丢失的情况。在后续的 BSD UNIX 4.3 版和 UNIX SVR3 中，增加了可靠信号机制，而且 BSD UNIX 还扩展了 SIGUSR1 和 SIGUSR2 信号，目的是将其用于进程间传递特定事件，但 BSD 和 SVF3 二者并不兼容。直到 POSIX.1 标准的提出，才对可靠信号相关的系统调用和语义进行了标准化。

---

简而言之，本章我们首先建立基于文件的统一 I/O 抽象，将标准输入/标准输出的访问改造为基于文件描述符，然后同样基于文件描述符实现一种父子进程之间的通信机制——管道，从而实现灵活的进程间通信，并基于文件抽象和管道支持不同的独立进程之间的动态组合，来实现复杂功能。而且通过实现信号机制，进程和操作系统可以主动发出信号来异步地通知相关事件给其它进程。这样就构成了具有团队协作能力的白垩纪“迅猛龙”<sup>1</sup> 操作系统。

### 8.1.2 实践体验

获取本章代码：

```
$ git clone https://github.com/rcore-os/rCore-Tutorial-v3.git
$ cd rCore-Tutorial-v3
$ git checkout ch7
```

在 qemu 模拟器上运行本章代码：

```
$ cd os
$ make run
```

进入 shell 程序后，可以运行管道机制的简单测例 pipetest 和比较复杂的测例 pipe\_large\_test。pipetest 需要保证父进程通过管道传输给子进程的字符串不会发生变化；而 pipe\_large\_test 中，父进程将一个长随机字符串传给子进程，随后父子进程同时计算该字符串的某种 Hash 值（逐字节求和），子进程会将计算后的 Hash 值传回父进程，而父进程接受到之后，需要验证两个 Hash 值相同，才算通过测试。

运行两个测例的输出可能如下：

```
>> pipetest
Read OK, child process exited!
pipetest passed!
Shell: Process 2 exited with code 0
>> pipe_large_test
```

(下页继续)

<sup>1</sup> 迅猛龙是一种中型恐龙，生活于 8300 至 7000 万年前的晚白垩纪，它们是活跃的团队合作型捕食动物，可以组队捕食行动迅速的猎物。

(续上页)

```
sum = 369114(parent)
sum = 369114(child)
Child process exited!
pipe_large_test passed!
Shell: Process 2 exited with code 0
>>
```

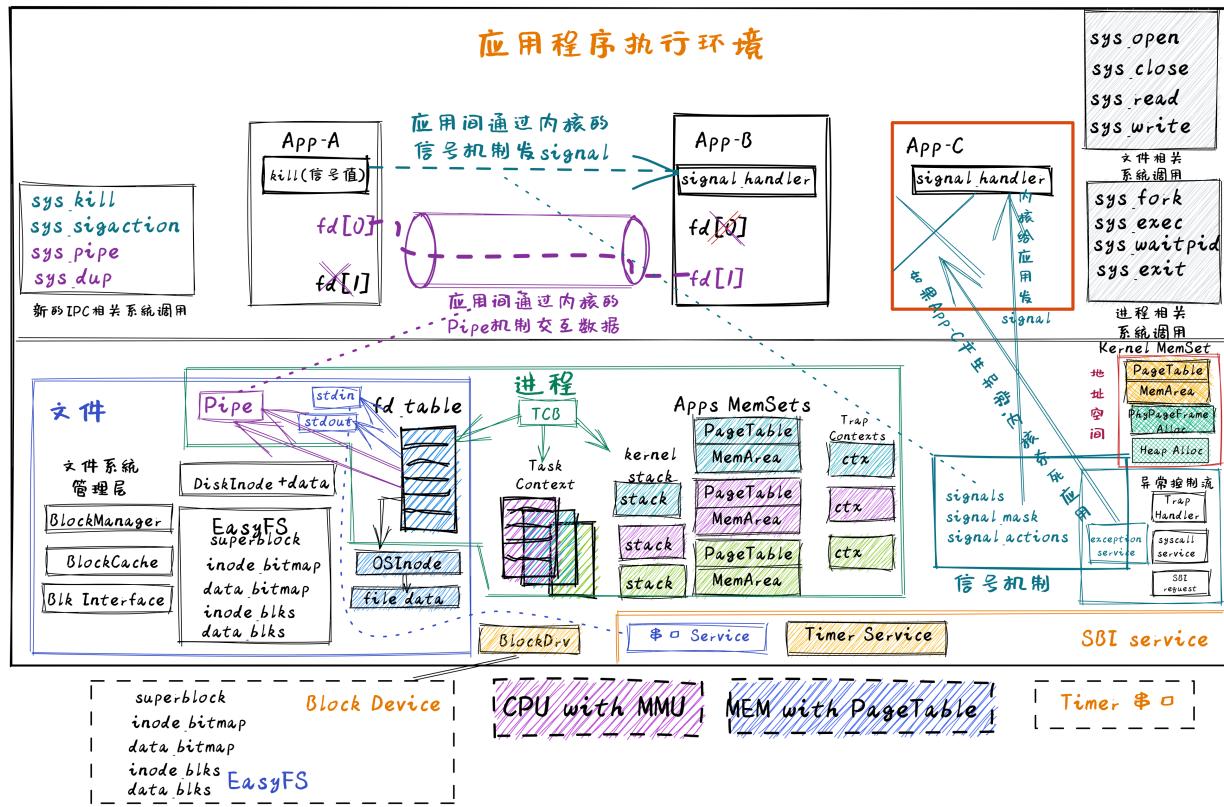
此外，在本章我们为 shell 程序支持了输入/输出重定向功能，可以将一个应用的输出保存到一个指定的文件。例如，下面的命令可以将 `yield` 应用的输出保存在文件 `fileb` 当中，并在应用执行完毕之后确认它的输出：

```
>> yield > fileb
Shell: Process 2 exited with code 0
>> cat fileb
Hello, I am process 2.
Back in process 2, iteration 0.
Back in process 2, iteration 1.
Back in process 2, iteration 2.
Back in process 2, iteration 3.
Back in process 2, iteration 4.
yield pass.

Shell: Process 2 exited with code 0
>>
```

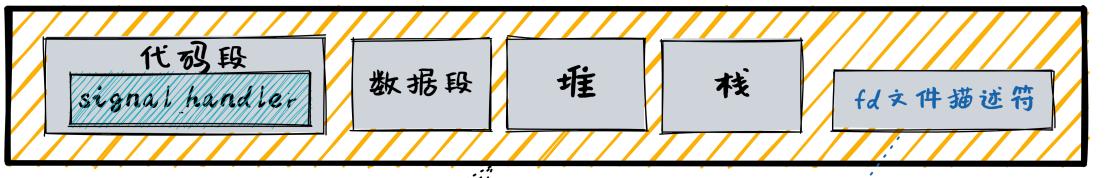
### 8.1.3 本章代码树

迅猛龙操作系统-IPCOS 的总体结构如下图所示：

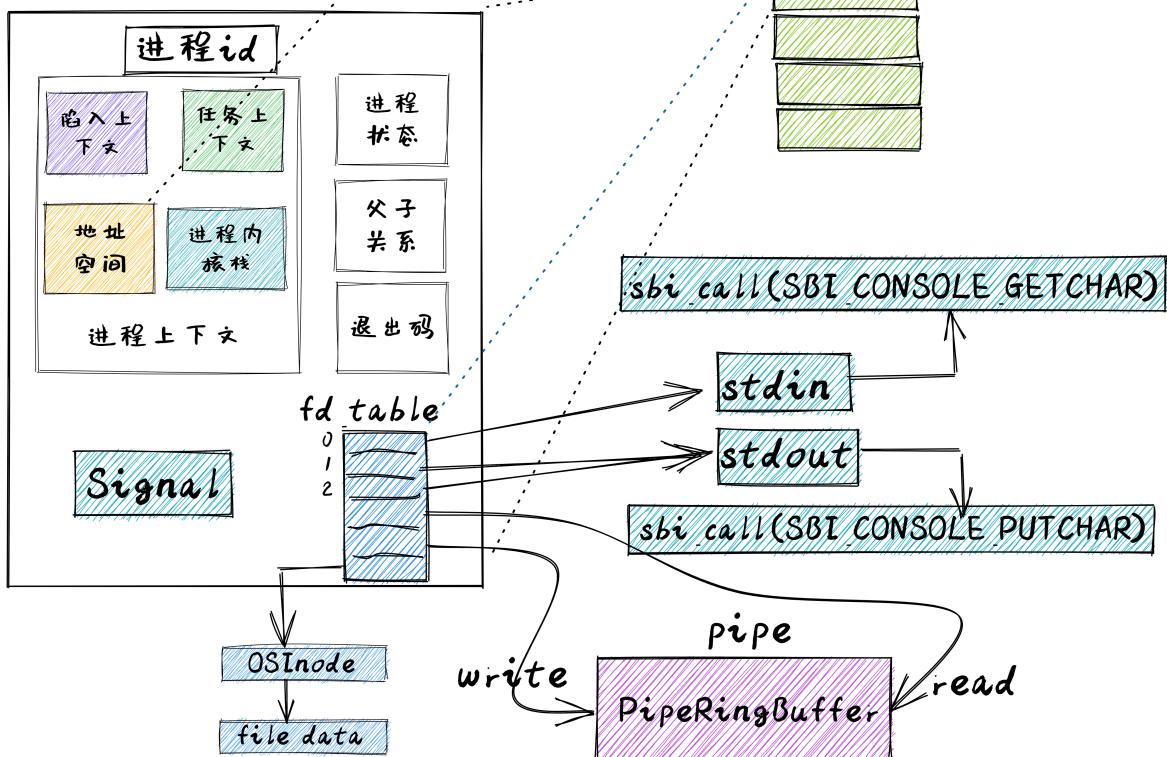


通过上图，大致可以看出迅猛龙操作系统-IPCOs 增加了两种通信机制，一种是交换数据的管道（Pipe）机制，另外一种是发送异步通知事件的信号（signal）机制，应用程序通过新增的管道和信号相关的系统调用可以完成进程间通信。这两种机制所对应的资源都被进程管理，如下图所示。

## 应用可访问的地址空间



## 进程控制块 (TCB) 数据结构



这里把管道看成是一种特殊的内存文件，并在进程的打开文件表 `fd_table` 中被管理，而且进程通过文件读写系统调用就可以很方便地基于管道实现进程间的数据交换。而信号是进程管理的一种资源，发送信号的进程可以通过系统调用给接收信号的目标进程控制块中的 `signal` 结构更新所发信号信息，操作系统再通过扩展 `trap_handler` 中从内核态返回到用户态的处理流程，改变了接收信号的目标进程的执行上下文，从而让接收信号的目标进程可以优先执行处理信号事件的预设函数 `signal_handler`，在处理完信号后，再继续执行之前暂停的工作。

位于 ch7 分支上的迅猛龙操作系统 - IPCOS 的源代码如下所示：

```
./os/src
Rust 28 Files 2061 Lines
Assembly 3 Files 88 Lines

├── bootloader
│ └── rustsbi-qemu.bin
├── LICENSE
└── os
 └── build.rs
```

(下页继续)

(续上页)

```

Cargo.lock
Cargo.toml
Makefile
src
 config.rs
 console.rs
 entry.asm
 fs(新增: 文件系统子模块 fs)
 mod.rs(包含已经打开且可以被进程读写的文件的抽象 File Trait)
 pipe.rs(实现了 File Trait 的第一个分支——可用来进程间通信的管道)
 stdio.rs(实现了 File Trait 的第二个分支——标准输入/输出)
 lang_items.rs
 link_app.S
 linker-qemu.1d
 loader.rs
 main.rs
 mm
 address.rs
 frame_allocator.rs
 heap_allocator.rs
 memory_set.rs
 mod.rs
 page_table.rs
 sbi.rs
 syscall
 fs.rs(修改: 调整 sys_read/write 的实现, 新增 sys_dup/pipe)
 mod.rs(修改: 调整 syscall 分发)
 process.rs
 task
 context.rs
 manager.rs
 mod.rs
 pid.rs
 processor.rs
 switch.rs
 switch.S
 task.rs(修改: 在任务控制块中加入文件描述符表相关机制)
 timer.rs
 trap
 context.rs
 mod.rs
 trap.S
README.md
rust-toolchain
user
 Cargo.lock
 Cargo.toml
 Makefile
 src
 bin
 exit.rs
 fantastic_text.rs
 forktest2.rs
 forktest.rs
 forktest_simple.rs
 forktree.rs

```

(下页继续)

(续上页)

```

├── hello_world.rs
├── initproc.rs
├── matrix.rs
├── pipe_large_test.rs (新增)
├── pipetest.rs (新增)
├── run_pipe_test.rs (新增)
├── sleep.rs
├── sleep_simple.rs
├── stack_overflow.rs
├── user_shell.rs
└── usertests.rs
└── yield.rs
├── console.rs
├── lang_items.rs
├── lib.rs (新增两个系统调用: sys_dup/sys_pipe)
├── linker.ld
└── syscall.rs (新增两个系统调用: sys_dup/sys_pipe)

```

### 8.1.4 本章代码导读

实现迅猛龙操作系统的过程就是对各种内核数据结构和相关操作的进一步扩展的过程。这里主要涉及到：

- 支持标准输入/输出文件
- 支持管道文件
- 支持对应用程序的命令行参数的解析和传递
- 实现标准 I/O 重定向功能
- 支持信号

#### 支持标准输入/输出文件

到本章为止我们将支持三种文件：标准输入输出、管道以及在存储设备上的常规文件和目录文件。不同于前面章节，我们将标准输入输出分别抽象成 `Stdin` 和 `Stdout` 两个类型，并为他们实现 `File Trait`。在 `TaskControlBlock::new` 创建初始进程的时候，就默认打开了标准输入输出，并分别绑定到文件描述符 0 和 1 上面。

#### 支持管道文件

管道 `Pipe` 是另一种文件，它可以用子进程间的单向进程间通信。我们还需要为它实现 `File Trait`。  
`os/src/syscall/fs.rs` 中的系统调用 `sys_pipe` 可以用来打开一个管道并返回读端/写端两个文件的文件描述符。管道的具体实现在 `os/src/fs/pipe.rs` 中，本章第二节 [管道](#) 中给出了详细的讲解。管道机制的测试用例可以参考 `user/src/bin` 目录下的 `pipetest.rs` 和 `pipe_large_test.rs` 两个文件。

#### 支持对应用程序的命令行参数的解析和传递

为支持独立进程间的 I/O 重定向，将在本章第三节 [命令行参数与标准 I/O 重定向](#) 中进一步支持对应用程序的命令行参数的解析和传递，这样可以让应用通过命令行参数来灵活地完成不同功能。这需要扩展对应的系统调用 `sys_exec`，主要的改动就是在创建新进程时，把命令行参数压入用户栈中，这样应用程序在执行时就可以从用户栈中获取到命令行的参数值了。

#### 实现标准 I/O 重定向功能

上面的工作都是为了支持 I/O 重定向，但还差一点。我们还需添加一条文件描述符相关的重要规则：即进程打开一个文件的时候，内核总是会将文件分配到该进程文件描述符表中编号最小的空闲位置。还需实现符合这个规则的新系统调用 `sys_dup`：复制文件描述符。这样就可以巧妙地实现标准 I/O 重定向功能了。

具体思路是，在某应用进程执行之前，父进程（比如 `user_shell` 进程）要对子进程的文件描述符表进行某种替换。以输出为例，父进程在创建子进程前，提前打开一个常规文件 A，然后 `fork` 子进程，在子进程的最初执行中，通过 `sys_close` 关闭 `Stdout` 文件描述符，用 `sys_dup` 复制常规文件 A 的文件描述符，这样 `Stdout` 文件描述符实际上指向的就是常规文件 A 了，这时再通过 `sys_close` 关闭常规文件 A 的文件描述符。至此，常规文件 A 替换掉了应用文件描述符表位置 1 处的标准输出文件，这就完成了所谓的 **重定向**，即完成了执行新应用前的准备工作。接下来，子进程调用 `sys_exec` 系统调用，创建并开始执行新应用。在重定向之后，新应用所在进程认为自己输出到 `fd=1` 的标准输出文件，但实际上输出到父进程（比如 `user_shell` 进程）指定的文件 A 中，从而实现了两个进程之间的信息传递。

### 支持信号

信号（Signals）是操作系统中实现进程间通信的一种异步通知机制，可以看成是一个应用发出某种信号，希望另外一个应用能及时响应。操作系统为支持这一目标，需要解决三个主要问题：如何向进程发送信号、进程如何接收信号、而信号如何被处理。

操作系统首先需要定义信号类型，表明不同含义的事件。接下来需要扩展进程控制块的内容，把与信号作为一种资源管理起来。发送信号的进程要做的事情比较简单，通过系统调用 `kill` 给接收信号的目标进程发信号，操作系统会在目标进程控制块中的 `signal` 结构中记录要接收信号。

这里比较复杂的是接收信号的进程要处理的事务。在进程控制块中，包含了接收到的信号集合 `signals`，以及要接收的信号对应的信号处理函数的地址 `SignalAction.handler`。当操作系统从内核态返回到目标进程的用户态继续执行时，具体的处理过程由 `trap_handler` 负责，`trap_handler` 分析目标进程控制块，如果该进程有带接收的信号，且提供了该信号对应的信号处理例程，则备份目标进程的用户态执行上下文，再修改目标进程的用户态执行上下文，让目标进程先执行信号处理函数。目标进程执行完毕信号处理函数后，需要执行一个系统调用 `sys_sigreturn` 回到内核态，这时内核再恢复刚才备份的目标进程的用户态执行上下文，这样目标进程就可以恢复之前的执行流程了。

## 8.2 基于文件的标准输入/输出

### 8.2.1 本节导读

本节我们介绍为何要把标准输入/输出用文件来进行抽象，以及如何以文件和文件描述符概念来重新定义标准输入/输出，并在进程中把各种文件描述符组织到文件描述符表中，同时将进程对于标准输入输出的访问修改为基于文件抽象的接口实现。这主要是为下一节支持进程间信息传递的管道实现打下基础。由于管道是基于文件抽象接口来实现的，所以我们将首先对一切皆是文件的设计思路进行介绍。

### 8.2.2 一切皆是文件

在 UNIX 操作系统之前，大多数的操作系统提供了各种复杂且不规则的设计实现来处理各种 I/O 设备（也可称为 I/O 资源），如键盘、显示器、以磁盘为代表的存储介质、以串口为代表的通信设备等，使得应用程序开发繁琐且很难统一表示和处理 I/O 设备。随着 UNIX 的诞生，一个简洁优雅的 I/O 设备抽象出现了，这就是 **文件**。在 UNIX 操作系统中，“一切皆文件”（*Everything is a file*）是一种重要的设计思想，这种设计思想继承于 Multics 操作系统的 **通用性**文件的设计理念，并进行了进一步的简化。在本章中，应用程序访问的 **文件**（File）就是一系列的字节组合。操作系统管理文件，但操作系统不关心文件内容，只关心如何对文件按字节流进行读写的机制，这就意味着任何程序可以读写任何文件（即字节流），对文件具体内容的解析是应用程序的任务，操作系统对此不做任何干涉。例如，一个 Rust 编译器可以读取一个 C 语言源程序并进行编译，操作系统并不会阻止这样的事情发生。

有了文件这样的抽象后，操作系统内核就可把能读写的 I/O 资源按文件来进行管理，并把文件分配给进程，让进程以统一的文件访问接口与 I/O 资源进行交互。在目前和后续可能涉及到的 I/O 硬件设备中，大致可以分成以下几种：

- **键盘设备**是程序获得字符输入的一种设备，也可抽象为一种只读性质的文件，可以从这个文件中读出一系列的字节序列；
- **屏幕设备**是展示程序的字符输出结果的一种字符显示设备，可抽象为一种只写性质的文件，可以向这个文件中写入一系列的字节序列，在显示屏上可以直接呈现出来；
- **串口设备**是获得字符输入和展示程序的字符输出结果的一种字符通信设备，可抽象为一种可读写性质的文件，可以向这个文件中写入一系列的字节序列传给程序，也可把程序要显示的字符传输出去；还可以把这个串口设备拆分成两个文件，一个用于获取输入字符的只读文件和一个传出输出字符的只写文件。

在 QEMU 模拟的 RISC-V 计算机和 K210 物理硬件上存在虚拟/物理串口设备，开发者可通过 QEMU 的串口命令行界面或特定串口通信工具软件来对虚拟/物理串口设备进行输入/输出操作。由于 RustSBI 直接管理了串口设备，并给操作系统提供了基于串口收发字符的两个 SBI 接口，从而使得操作系统可以很简单地通过这两个 SBI 接口，完成输出或输入字符串的工作。

### 8.2.3 标准输入/输出对 `File trait` 的实现

其实我们在第二章就对应用程序引入了基于 `文件` 的标准输出接口 `sys_write`，在第五章引入了基于 `文件` 的标准输入接口 `sys_read`；在第六章引入 `文件系统`，在进程控制块中添加了表示打开文件集合的文件描述符表。我们提前把标准输出设备在文件描述符表中的文件描述符的值规定为 1，用 `Stdout` 表示；把标准输入设备在文件描述符表中的文件描述符的值规定为 0，用 `Stdin` 表示。现在，我们可以重构操作系统，为标准输入和标准输出实现 `File Trait`，使得进程可以按文件接口与 I/O 外设进行交互：

```

1 // os/src/fs/stdio.rs
2
3 pub struct Stdin;
4
5 pub struct Stdout;
6
7 impl File for Stdin {
8 fn read(&self, mut user_buf: UserBuffer) -> usize {
9 assert_eq!(user_buf.len(), 1);
10 // busy loop
11 let mut c: usize;
12 loop {
13 c = console_getchar();
14 if c == 0 {
15 suspend_current_and_run_next();
16 continue;
17 } else {
18 break;
19 }
20 }
21 let ch = c as u8;
22 unsafe { user_buf.buffers[0].as_mut_ptr().write_volatile(ch); }
23 1
24 }
25 fn write(&self, _user_buf: UserBuffer) -> usize {
26 panic!("Cannot write to stdin!");
27 }
28}
29
30 impl File for Stdout {
31 fn read(&self, _user_buf: UserBuffer) -> usize {
32 panic!("Cannot read from stdout!");
33 }
34}
```

(下页继续)

(续上页)

```

33 }
34 fn write(&self, user_buf: UserBuffer) -> usize {
35 for buffer in user_buf.buffers.iter() {
36 print!("{} ", core::str::from_utf8(*buffer).unwrap());
37 }
38 user_buf.len()
39 }
40 }

```

可以看到，标准输入文件 `Stdin` 是只读文件，只允许进程通过 `read` 从里面读入，目前每次仅支持读入一个字符，其实现与之前的 `sys_read` 基本相同，只是需要通过 `UserBuffer` 来获取具体将字节写入的位置。相反，标准输出文件 `Stdout` 是只写文件，只允许进程通过 `write` 写入到里面，实现方法是遍历每个切片，将其转化为字符串通过 `print!` 宏来输出。

## 8.2.4 对标准输入/输出的管理

这样，应用程序如果要基于文件进行 I/O 访问，大致就会涉及如下几个操作：

- 打开 (open)：进程只有打开文件，操作系统才能返回一个可进行读写的文件描述符给进程，进程才能基于这个值来进行对应文件的读写；
- 关闭 (close)：进程基于文件描述符关闭文件后，就不能再对文件进行读写操作了，这样可以在一定程度上保证对文件的合法访问；
- 读 (read)：进程可以基于文件描述符来读文件内容到相应内存中；
- 写 (write)：进程可以基于文件描述符来把相应内存内容写到文件中；

在本节中，还不会涉及创建文件。当一个进程被创建的时候，内核会默认为其打开三个缺省就存在的文件：

- 文件描述符为 0 的标准输入
- 文件描述符为 1 的标准输出
- 文件描述符为 2 的标准错误输出

在我们的实现中并不区分标准输出和标准错误输出，而是会将文件描述符 1 和 2 均对应到标准输出。实际上，在本章中，标准输出文件就是串口输出，标准输入文件就是串口输入。

这里隐含着有关文件描述符的一条重要的规则：即进程打开一个文件的时候，内核总是会将文件分配到该进程文件描述符表中 **最小的空闲位置**。比如，当一个进程被创建以后立即打开一个文件，则内核总是会返回文件描述符 3 (0~2 号文件描述符已被缺省打开了)。当我们关闭一个打开的文件之后，它对应的文件描述符将会变得空闲并在后面可以被分配出去。

## 创建标准输入/输出文件

当新建一个进程的时候，我们需要按照先前的说明为进程打开标准输入文件和标准输出文件：

```

1 // os/src/task/task.rs
2
3 impl TaskControlBlock {
4 pub fn new(elf_data: &[u8]) -> Self {
5 ...
6 let task_control_block = Self {
7 pid: pid_handle,
8 kernel_stack,
9 inner: Mutex::new(TaskControlBlockInner {

```

(下页继续)

(续上页)

```

10 trap_cx_ppn,
11 base_size: user_sp,
12 task_cx_ptr: task_cx_ptr as usize,
13 task_status: TaskStatus::Ready,
14 memory_set,
15 parent: None,
16 children: Vec::new(),
17 exit_code: 0,
18 fd_table: vec![
19 // 0 -> stdin
20 Some(Arc::new(Stdin)),
21 // 1 -> stdout
22 Some(Arc::new(Stdout)),
23 // 2 -> stderr
24 Some(Arc::new(Stdout)),
25],
26 },
27 };
28 ...
29 }
30 }
```

## 继承标准输入/输出文件

此外，在 fork 的时候，子进程需要完全继承父进程的文件描述符表来和父进程共享所有文件：

```

1 // os/src/task/task.rs
2
3 impl TaskControlBlock {
4 pub fn fork(self: &Arc<TaskControlBlock>) -> Arc<TaskControlBlock> {
5 ...
6 // push a goto_trap_return task_cx on the top of kernel stack
7 let task_cx_ptr = kernel_stack.push_on_top(TaskContext::goto_trap_return());
8 // copy fd table
9 let mut new_fd_table: Vec<Option<Arc<dyn File + Send + Sync>>> = Vec::new();
10 for fd in parent_inner.fd_table.iter() {
11 if let Some(file) = fd {
12 new_fd_table.push(Some(file.clone()));
13 } else {
14 new_fd_table.push(None);
15 }
16 }
17 let task_control_block = Arc::new(TaskControlBlock {
18 pid: pid_handle,
19 kernel_stack,
20 inner: Mutex::new(TaskControlBlockInner {
21 trap_cx_ppn,
22 base_size: parent_inner.base_size,
23 task_cx_ptr: task_cx_ptr as usize,
24 task_status: TaskStatus::Ready,
25 memory_set,
26 parent: Some(Arc::downgrade(self)),
27 children: Vec::new(),
28 exit_code: 0,
29 fd_table: new_fd_table,
30 })
31 });
32 return task_control_block;
33 }
34 }
```

(下页继续)

(续上页)

```

30 } ,
31 });
32 // add child
33 ...
34 }
35 }
```

这样，即使我们仅手动为初始进程 `initproc` 打开了标准输入输出，所有进程也都可以访问它们。

## 8.2.5 读写标准输入/输出文件

由于有基于文件抽象接口和文件描述符表，之前实现的文件读写系统调用 `sys_read/write` 可以直接用于标准输入/输出文件，很好地达到了代码重用的目标。这样，操作系统通过文件描述符在当前进程的文件描述符表中找到某个文件，无需关心文件具体的类型，只要知道它一定实现了 `File Trait` 的 `read/write` 方法即可。Trait 对象提供的运行时多态能力会在运行的时候帮助我们定位到符合实际类型的 `read/write` 方法，完成不同类型文件各自的读写。

## 8.3 管道

### 8.3.1 本节导读

在上一节，我们实现了基于文件接口的标准输入和输出，这样一个进程可以根据不同的输入产生对应的输出。本节我们将基于上一节介绍的文件接口 `File` 来把不同进程的输入和输出连接起来，从而在不改变应用程序代码的情况下，让操作系统具有进程间信息交换和功能组合的能力。这需要我们实现一种父子进程间的单向进程间通信机制——管道，并为此实现两个新的系统调用 `sys_pipe` 和 `sys_close`。

### 8.3.2 管道机制简介

首先来介绍什么是 **管道 (Pipe)**。管道是一种进程间通信机制，由操作系统提供，并可通过直接编程或在 shell 程序的帮助下轻松地把不同进程（目前是父子进程之间或子子进程之间）的输入和输出对接起来。我们也可以将管道看成一个有一定缓冲区大小的字节队列，它分为读和写两端，需要通过不同的文件描述符来访问。读端只能用来从管道中读取，而写端只能用来将数据写入管道。由于管道是一个队列，读取数据的时候会从队头读取并弹出数据，而写入数据的时候则会把数据写入到队列的队尾。由于管道的缓冲区大小是有限的，一旦整个缓冲区都被填满就不能再继续写入，就需要等到读端读取并从队列中弹出一些数据之后才能继续写入。当缓冲区为空的时候，读端自然也不能继续从里面读取数据，需要等到写端写入了一些数据之后才能继续读取。

一般在 shell 程序中，“|”是管道符号，即两个命令之间的一道竖杠。我们通过管道符号组合的命令，就可以了解登录 Linux 的用户的各种情况：

```

who # 登录 Linux 的用户信息
who | grep chyyuu # 是否用户 ID 为 chyyuu 的用户登录了
who | grep chyyuu | wc # chyyuu 用户目前在线登录的个数
```

### 8.3.3 管道的系统调用原型及使用方法

接下来，我们将逐步尝试实现上面描述的管道的初步效果。我们新增一个系统调用来为当前进程打开一个代表管道的文件集（包含一个只读文件，一个只写文件）：

```
/// 功能：为当前进程打开一个管道。
/// 参数：pipe 表示应用地址空间中的一个长度为 2 的 usize_
//→ 数组的起始地址，内核需要按顺序将管道读端
/// 和写端的文件描述符写入到数组中。
/// 返回值：如果出现了错误则返回 -1，否则返回 0。可能的错误原因是：传入的地址不合法。
/// syscall ID: 59
pub fn sys_pipe(pipe: *mut usize) -> isize;
```

在用户库中会将其包装为 pipe 函数：

```
// user/src/syscall.rs

const SYSCALL_PIPE: usize = 59;

pub fn sys_pipe(pipe: &mut [usize]) -> isize {
 syscall(SYSCALL_PIPE, [pipe.as_mut_ptr() as usize, 0, 0])
}

// user/src/lib.rs

pub fn pipe(pipe_fd: &mut [usize]) -> isize { sys_pipe(pipe_fd) }
```

只有当一个管道的所有读端文件/写端文件都被关闭之后，管道占用的资源才会被回收，因此我们需要通过关闭文件的系统调用 sys\_close（它会在用户库中被包装为 close 函数。），来尽可能早的关闭之后不再用到的读端的文件和写端的文件。



我们来从简单的管道测例 pipetest 中介绍管道的使用方法：

```
1 // user/src/bin/pipetest.rs
2
3 #[no_std]
4 #[no_main]
5
6 #[macro_use]
7 extern crate user_lib;
8
9 use user_lib::{fork, close, pipe, read, write, wait};
10
11 static STR: &str = "Hello, world!";
12
13 #[no_mangle]
14 pub fn main() -> i32 {
15 // create pipe
16 let mut pipe_fd = [0usize; 2];
```

(下页继续)

(续上页)

```

17 pipe(&mut pipe_fd);
18 // read end
19 assert_eq!(pipe_fd[0], 3);
20 // write end
21 assert_eq!(pipe_fd[1], 4);
22 if fork() == 0 {
23 // child process, read from parent
24 // close write_end
25 close(pipe_fd[1]);
26 let mut buffer = [0u8; 32];
27 let len_read = read(pipe_fd[0], &mut buffer) as usize;
28 // close read_end
29 close(pipe_fd[0]);
30 assert_eq!(core::str::from_utf8(&buffer[..len_read]).unwrap(), STR);
31 println!("Read OK, child process exited!");
32 0
33 } else {
34 // parent process, write to child
35 // close read end
36 close(pipe_fd[0]);
37 assert_eq!(write(pipe_fd[1], STR.as_bytes()), STR.len() as isize);
38 // close write end
39 close(pipe_fd[1]);
40 let mut child_exit_code: i32 = 0;
41 wait(&mut child_exit_code);
42 assert_eq!(child_exit_code, 0);
43 println!("pipetest passed!");
44 0
45 }
46 }

```

在父进程中，我们通过 `pipe` 打开一个管道文件数组，其中 `pipe_fd[0]` 保存了管道读端的文件描述符，而 `pipe_fd[1]` 保存了管道写端的文件描述符。在 `fork` 之后，子进程会完全继承父进程的文件描述符表，于是子进程也可以通过同样的文件描述符来访问同一个管道的读端和写端。之前提到过管道是单向的，在这个测例中我们希望管道中的数据从父进程流向子进程，也即父进程仅通过管道的写端写入数据，而子进程仅通过管道的读端读取数据。

因此，在第 25 和第 36 行，分别第一时间在子进程中关闭管道的写端和在父进程中关闭管道的读端。父进程在第 37 行将字符串 `STR` 写入管道的写端，随后在第 39 行关闭管道的写端；子进程在第 27 行从管道的读端读取字符串，并在第 29 行关闭。

如果想在父子进程之间实现双向通信，我们就必须创建两个管道。有兴趣的同学可以参考测例 `pipe_large_test`。

### 8.3.4 基于文件的管道

我们将管道的一端（读端或写端）抽象为 `Pipe` 类型：

```

// os/src/fs/pipe.rs

pub struct Pipe {
 readable: bool,
 writable: bool,
 buffer: Arc<Mutex<PipeRingBuffer>>,
}

```

readable 和 writable 分别指出该管道端可否支持读取/写入，通过 buffer 字段还可以找到该管道端所在的管道自身。后续我们将为它实现 File Trait，之后它便可以通过文件描述符来访问。

而管道自身，也就是那个带有一定大小缓冲区的字节队列，我们抽象为 PipeRingBuffer 类型：

```
// os/src/fs/pipe.rs

const RING_BUFFER_SIZE: usize = 32;

#[derive(Copy, Clone, PartialEq)]
enum RingBufferStatus {
 FULL,
 EMPTY,
 NORMAL,
}

pub struct PipeRingBuffer {
 arr: [u8; RING_BUFFER_SIZE],
 head: usize,
 tail: usize,
 status: RingBufferStatus,
 write_end: Option<Weak<Pipe>>,
}
```

- RingBufferStatus 记录了缓冲区目前的状态：FULL 表示缓冲区已满不能再继续写入；EMPTY 表示缓冲区为空无法从里面读取；而 NORMAL 则表示除了 FULL 和 EMPTY 之外的其他状态。
- PipeRingBuffer 的 arr/head/tail 三个字段用来维护一个循环队列，其中 arr 为存放数据的数据，head 为循环队列队头的下标，tail 为循环队列队尾的下标。
- PipeRingBuffer 的 write\_end 字段还保存了它的写端的一个弱引用计数，这是由于在某些情况下需要确认该管道所有的写端是否都已经被关闭了，通过这个字段很容易确认这一点。

从内存管理的角度，每个读端或写端中都保存着所属管道自身的强引用计数，且我们确保这些引用计数只会出现在管道端口 Pipe 结构体中。于是，一旦一个管道所有的读端和写端均被关闭，便会导致它们所属管道的引用计数变为 0，循环队列缓冲区所占用的资源被自动回收。虽然 PipeRingBuffer 中保存了一个指向写端的引用计数，但是它是一个弱引用，也就不会出现循环引用的情况导致内存泄露。

## 管道创建

通过 PipeRingBuffer::new 可以创建一个新的管道：

```
// os/src/fs/pipe.rs

impl PipeRingBuffer {
 pub fn new() -> Self {
 Self {
 arr: [0; RING_BUFFER_SIZE],
 head: 0,
 tail: 0,
 status: RingBufferStatus::EMPTY,
 write_end: None,
 }
 }
}
```

Pipe 的 read/write\_end\_with\_buffer 方法可以分别从一个已有的管道创建它的读端和写端：

```
// os/src/fs/pipe.rs

impl Pipe {
 pub fn read_end_with_buffer(buffer: Arc<Mutex<PipeRingBuffer>>) -> Self {
 Self {
 readable: true,
 writable: false,
 buffer,
 }
 }
 pub fn write_end_with_buffer(buffer: Arc<Mutex<PipeRingBuffer>>) -> Self {
 Self {
 readable: false,
 writable: true,
 buffer,
 }
 }
}
```

可以看到，读端和写端的访问权限进行了相应设置：不允许向读端写入，也不允许从写端读取。

通过 make\_pipe 方法可以创建一个管道并返回它的读端和写端：

```
// os/src/fs/pipe.rs

impl PipeRingBuffer {
 pub fn set_write_end(&mut self, write_end: &Arc<Pipe>) {
 self.write_end = Some(Arc::downgrade(write_end));
 }
}

/// Return (read_end, write_end)
pub fn make_pipe() -> (Arc<Pipe>, Arc<Pipe>) {
 let buffer = Arc::new(Mutex::new(PipeRingBuffer::new()));
 let read_end = Arc::new(
 Pipe::read_end_with_buffer(buffer.clone())
);
 let write_end = Arc::new(
 Pipe::write_end_with_buffer(buffer.clone())
);
 buffer.lock().set_write_end(&write_end);
 (read_end, write_end)
}
```

注意，我们调用 PipeRingBuffer::set\_write\_end 在管道中保留它的写端的弱引用计数。

现在来实现创建管道的系统调用 sys\_pipe：

```
1 // os/src/task/task.rs
2
3 impl TaskControlBlockInner {
4 pub fn alloc_fd(&mut self) -> usize {
5 if let Some(fd) = (0..self.fd_table.len())
6 .find(|fd| self.fd_table[*fd].is_none()) {
7 fd
8 } else {
9 self.fd_table.push(None);
10 self.fd_table.len() - 1
11 }
12 }
13}
```

(下页继续)

(续上页)

```

11 }
12 }
13 }
14
15 // os/src/syscall/fs.rs
16
17 pub fn sys_pipe(pipe: *mut usize) -> isize {
18 let task = current_task().unwrap();
19 let token = current_user_token();
20 let mut inner = task.acquire_inner_lock();
21 let (pipe_read, pipe_write) = make_pipe();
22 let read_fd = inner.alloc_fd();
23 inner.fd_table[read_fd] = Some(pipe_read);
24 let write_fd = inner.alloc_fd();
25 inner.fd_table[write_fd] = Some(pipe_write);
26 *translated_refmut(token, pipe) = read_fd;
27 *translated_refmut(token, unsafe { pipe.add(1) }) = write_fd;
28 0
29 }

```

TaskControlBlockInner::alloc\_fd 可以在进程控制块中分配一个最小的空闲文件描述符来访问一个新打开的文件。它先从小到大遍历所有曾经被分配过的文件描述符尝试找到一个空闲的，如果没有的话就需要拓展文件描述符表的长度并新分配一个。

在 sys\_pipe 中，第 21 行我们调用 make\_pipe 创建一个管道并获取其读端和写端，第 22~25 行我们分别为读端和写端分配文件描述符并将它们放置在文件描述符表中的相应位置中。第 26~27 行我们则是将读端和写端的文件描述符写回到应用地址空间。

## 管道读写

首先来看如何为 Pipe 实现 File Trait 的 read 方法，即从管道的读端读取数据。在此之前，我们需要对于管道循环队列进行封装来让它更易于使用：

```

1 // os/src/fs/pipe.rs
2
3 impl PipeRingBuffer {
4 pub fn read_byte(&mut self) -> u8 {
5 self.status = RingBufferStatus::NORMAL;
6 let c = self.arr[self.head];
7 self.head = (self.head + 1) % RING_BUFFER_SIZE;
8 if self.head == self.tail {
9 self.status = RingBufferStatus::EMPTY;
10 }
11 c
12 }
13 pub fn available_read(&self) -> usize {
14 if self.status == RingBufferStatus::EMPTY {
15 0
16 } else {
17 if self.tail > self.head {
18 self.tail - self.head
19 } else {
20 self.tail + RING_BUFFER_SIZE - self.head
21 }
22 }
23 }

```

(下页继续)

(续上页)

```

23 }
24 pub fn all_write_ends_closed(&self) -> bool {
25 self.write_end.as_ref().unwrap().upgrade().is_none()
26 }
27 }
```

PipeRingBuffer::read\_byte 方法可以从管道中读取一个字节，注意在调用它之前需要确保管道缓冲区中不是空的。它会更新循环队列队头的位置，并比较队头和队尾是否相同，如果相同的话则说明管道的状态变为空 EMPTY。仅仅通过比较队头和队尾是否相同不能确定循环队列是否为空，因为它既有可能表示队列为空，也有可能表示队列已满。因此我们需要在 read\_byte 的同时进行状态更新。

PipeRingBuffer::available\_read 可以计算管道中还有多少个字符可以读取。我们首先需要判断队列是否为空，因为队头和队尾相等可能表示队列为空或队列已满，两种情况 available\_read 的返回值截然不同。如果队列为空的话直接返回 0，否则根据队头和队尾的相对位置进行计算。

PipeRingBuffer::all\_write\_ends\_closed 可以判断管道的所有写端是否都被关闭了，这是通过尝试将管道中保存的写端的弱引用计数升级为强引用计数来实现的。如果升级失败的话，说明管道写端的强引用计数为 0，也就意味着管道所有写端都被关闭了，从而管道中的数据不会再得到补充，待管道中仅剩的数据被读取完毕之后，管道就可以被销毁了。

下面是 Pipe 的 read 方法的实现：

```

// os/src/fs/pipe.rs

impl File for Pipe {
 fn read(&self, buf: UserBuffer) -> usize {
 assert!(&self.readable());
 let want_to_read = buf.len();
 let mut buf_iter = buf.into_iter();
 let mut already_read = 0;
 loop {
 let mut ring_buffer = self.buffer.exclusive_access();
 let loop_read = ring_buffer.available_read();
 if loop_read == 0 {
 if ring_buffer.all_write_ends_closed() {
 return already_read;
 }
 drop(ring_buffer);
 suspend_current_and_run_next();
 continue;
 }
 for _ in 0..loop_read {
 if let Some(byte_ref) = buf_iter.next() {
 unsafe {
 *byte_ref = ring_buffer.read_byte();
 }
 already_read += 1;
 if already_read == want_to_read {
 return want_to_read;
 }
 } else {
 return already_read;
 }
 }
 }
 }
}
```

- 第 7 行的 `buf_iter` 将传入的应用缓冲区 `buf` 转化为一个能够逐字节对于缓冲区进行访问的迭代器，每次调用 `buf_iter.next()` 即可按顺序取出用于访问缓冲区中一个字节的裸指针。
- 第 8 行的 `already_read` 用来维护实际有多少字节从管道读入应用的缓冲区。
- `File::read` 的语义是要从文件中最多读取应用缓冲区大小那么多字符。这可能超出了循环队列的大小，或者由于尚未有进程从管道的写端写入足够的字符，因此我们需要将整个读取的过程放在一个循环中，当循环队列中不存在足够字符的时候暂时进行任务切换，等待循环队列中的字符得到补充之后再继续读取。

这个循环从第 9 行开始，第 11 行我们用 `loop_read` 来表示循环这一轮次中可以从管道循环队列中读取多少字符。如果管道为空则会检查管道的所有写端是否都已经被关闭，如果是的话，说明我们已经没有任何字符可以读取了，这时可以直接返回；否则我们需要等管道的字符得到填充之后再继续读取，因此我们调用 `suspend_current_and_run_next` 切换到其他任务，等到切换回来之后回到循环开头再看一下管道中是否有字符了。在调用之前我们需要手动释放管道自身的锁，因为切换任务时候的 `__switch` 跨越了正常函数调用的边界。

如果 `loop_read` 不为 0，在这一轮次中管道中就有 `loop_read` 个字节可以读取。我们可以迭代应用缓冲区中的每个字节指针，并调用 `PipeRingBuffer::read_byte` 方法来从管道中进行读取。如果这 `loop_read` 个字节均被读取之后还没有填满应用缓冲区，就需要进入循环的下一个轮次，否则就可以直接返回了。在具体实现的时候注意边界条件的判断。

`Pipe` 的 `write` 方法即通过管道的写端向管道中写入数据的实现和 `read` 的原理类似，篇幅所限在这里不再赘述，感兴趣的同学可自行参考其实现。

这样我们就为管道 `Pipe` 实现了上节中的通用文件接口 `File trait`，并将其顺利整合到了文件子系统中。上一节我们提到子进程会继承父进程的所有文件描述符，管道自然也包括在内，使得父子进程可以使用共享的单向管道进行通信了。

### 8.3.5 小结

这一章讲述的重点是一种有趣的进程间通信的机制—管道。通过管道，能够把不同进程的输入和输出连接在一起，实现进程功能的组合。为了能够统一表示输入，输出，以及管道，我们给出了与 **地址空间**、**进程** 齐名的操作系统抽象 **文件**，并基于文件重构了操作系统的输入/输出机制。目前，仅仅实现了非常简单的基于父子进程的管道机制。在操作系统层面，还缺乏对命令行参数的支持，在应用层面，还缺少 I/O 重定向和 shell 程序中基于“|”管道符号的支持。但我们已经建立了基本的进程通信机制，实现了支持协作的白垩纪“迅猛龙”操作系统的大部分功能，使得应用程序之间可以合作完成更复杂的工作。但如果要让相互独立的应用程序之间也能合作，还需要对应用的执行参数进行一定的扩展，支持进程执行的命令行参数。这样才能在应用程序的层面，完善 I/O 重定向，并在 shell 中支持基于“|”管道符号，形成更加灵活的独立进程间的通信能力和 shell 命令行支持。

## 8.4 命令行参数与标准 I/O 重定向

### 8.4.1 本节导读

虽然我们已经支持从文件系统中加载应用，还实现了文件的创建和读写，但是目前我们在应用中只能硬编码要操作的文件，这就使得应用的功能大大受限，shell 程序对于文件的交互访问能力也很弱。为了解决这些问题，我们需要在 shell 程序和内核中支持命令行参数的解析和传递。而且我们可以把应用的命令行参数的扩展，管道以及标准 I/O 重定向功能综合在一起，来让两个甚至多个互不相干的应用也能合作。

## 8.4.2 命令行参数

在使用 C/C++ 语言开发 Linux 应用的时候，我们可以使用标准库提供的 `argc/argv` 来获取命令行参数，它们是直接被作为参数传给 `main` 函数的。下面来看一个打印命令行参数的例子：

```

1 // a.c
2
3 #include <stdio.h>
4
5 int main(int argc, char* argv[]) {
6 printf("argc = %d\n", argc);
7 for (int i = 0; i < argc; i++) {
8 printf("argv[%d] = %s\n", i, argv[i]);
9 }
10 return 0;
11 }
```

其中 `argc` 表示命令行参数的个数，而 `argv` 是一个长度为 `argc` 的字符串数组，数组中的每个字符串都是一个命令行参数。我们可以在 Linux 系统上运行这个程序：

```

$ gcc a.c -o a -g -Wall
$./a aa bb 11 22 cc
argc = 6
argv[0] = ./a
argv[1] = aa
argv[2] = bb
argv[3] = 11
argv[4] = 22
argv[5] = cc
```

为了支持后续的一些功能，我们希望在内核和 shell 程序上支持这个功能。为了对实现正确性进行测试，在本章中我们编写了一个名为 `cmdline_args` 的应用，它是用 Rust 编写的，并只能在我们的内核上执行，但是它的功能是和 `a.c` 保持一致的。我们可以在我们的内核上运行该应用来看看效果：

```

Rust user shell
>> cmdline_args aa bb 11 22 cc
argc = 6
argv[0] = cmdline_args
argv[1] = aa
argv[2] = bb
argv[3] = 11
argv[4] = 22
argv[5] = cc
Shell: Process 2 exited with code 0
>>
```

可以看到二者的输出是基本相同的。

但是，要实现这个看似简单的功能，需要内核和用户态应用的共同努力。为了支持命令行参数，`sys_exec` 的系统调用接口需要发生变化：

```

1 // user/src/syscall.rs
2
3 pub fn sys_exec(path: &str, args: &[*const u8]) -> isize;
```

可以看到，它的参数多出了一个 `args` 数组，数组中的每个元素都是一个命令行参数字符串的起始地址。由于我们是以引用的形式传递这个数组，实际传递给内核的是这个数组的起始地址：

```

1 // user/src/syscall.rs
2
3 pub fn sys_exec(path: &str, args: &[*const u8]) -> isize {
4 syscall(SYSCALL_EXEC, [path.as_ptr() as usize, args.as_ptr() as usize, 0])
5 }
6
7 // user/src/lib.rs
8
9 pub fn exec(path: &str, args: &[*const u8]) -> isize { sys_exec(path, args) }

```

接下来我们分析一下，一行带有命令行参数的命令从输入到它的命令行参数被打印出来中间经历了哪些过程。

### shell 程序的命令行参数分割

回忆一下，之前在 shell 程序 user\_shell 中，一旦接收到一个回车，我们就会将当前行的内容 line 作为一个名字并试图去执行同名的应用。但是现在 line 还可能包含一些命令行参数，只有最开头的一个才是要执行的应用名。因此我们要做的第一件事情就是将 line 用空格进行分割：

```

1 // user/src/bin/user_shell.rs
2
3 let args: Vec<_> = line.as_str().split(' ').collect();
4 let mut args_copy: Vec<String> = args
5 .iter()
6 .map(|&arg| {
7 let mut string = String::new();
8 string.push_str(arg);
9 string
10 })
11 .collect();
12
13 args_copy
14 .iter_mut()
15 .for_each(|string| {
16 string.push('\0');
17 });

```

经过分割，args 中的 &str 都是 line 中的一段子区间，它们的结尾并没有包含 \0，因为 line 是我们输入得到的，中间本来就没有 \0。由于在向内核传入字符串的时候，我们只能传入字符串的起始地址，因此我们必须保证其结尾为 \0。从而我们用 args\_copy 将 args 中的字符串拷贝一份到堆上并在末尾手动加入 \0。这样就可以安心的将 args\_copy 中的字符串传入内核了。我们用 args\_addr 来收集这些字符串的起始地址：

```

1 // user/src/bin/user_shell.rs
2
3 let mut args_addr: Vec<*const u8> = args_copy
4 .iter()
5 .map(|arg| arg.as_ptr())
6 .collect();
7 args_addr.push(0 as *const u8);

```

向量 args\_addr 中的每个元素都代表一个命令行参数字符串的起始地址。由于我们要传递给内核的是这个向量的起始地址，为了让内核能够获取到命令行参数的个数，我们需要在 args\_addr 的末尾放入一个 0，这样内核看到它的时候就能知道命令行参数已经获取完毕了。

在 fork 出来的子进程里面我们需要这样执行应用：

```

1 // user/src/bin/user_shell.rs
2
3 // child process
4 if exec(args_copy[0].as_str(), args_addr.as_slice()) == -1 {
5 println!("Error when executing!");
6 return -4;
7 }
```

## sys\_exec 将命令行参数压入用户栈

在 sys\_exec 中，首先需要将应用传进来的命令行参数取出来：

```

1 // os/src/syscall/process.rs
2
3 pub fn sys_exec(path: *const u8, mut args: *const usize) -> isize {
4 let token = current_user_token();
5 let path = translated_str(token, path);
6 let mut args_vec: Vec<String> = Vec::new();
7 loop {
8 let arg_str_ptr = *translated_ref(token, args);
9 if arg_str_ptr == 0 {
10 break;
11 }
12 args_vec.push(translated_str(token, arg_str_ptr as *const u8));
13 unsafe { args = args.add(1); }
14 }
15 if let Some(app_inode) = open_file(path.as_str(), OpenFlags::RDONLY) {
16 let all_data = app_inode.read_all();
17 let task = current_task().unwrap();
18 let argc = args_vec.len();
19 task.exec(all_data.as_slice(), args_vec);
20 // return argc because cx.x[10] will be covered with it later
21 argc as isize
22 } else {
23 -1
24 }
25 }
```

这里的 args 指向命令行参数字符串起始地址数组中的一个位置，每次我们都可以从一个起始地址通过 translated\_str 拿到一个字符串，直到 args 为 0 就说明没有更多命令行参数了。在第 19 行调用 TaskControlBlock::exec 的时候，我们需要将获取到的 args\_vec 传入进去并将里面的字符串压入到用户栈上。

```

1 // os/src/task/task.rs
2
3 impl TaskControlBlock {
4 pub fn exec(&self, elf_data: &[u8], args: Vec<String>) {
5 // memory_set with elf program headers/trampoline/trap context/user stack
6 let (memory_set, mut user_sp, entry_point) = MemorySet::from_elf(elf_data);
7 let trap_cx_ppn = memory_set
8 .translate(VirtAddr::from(TRAP_CONTEXT).into())
9 .unwrap()
10 .ppn();
11 // push arguments on user stack
12 user_sp -= (args.len() + 1) * core::mem::size_of::<usize>();
```

(下页继续)

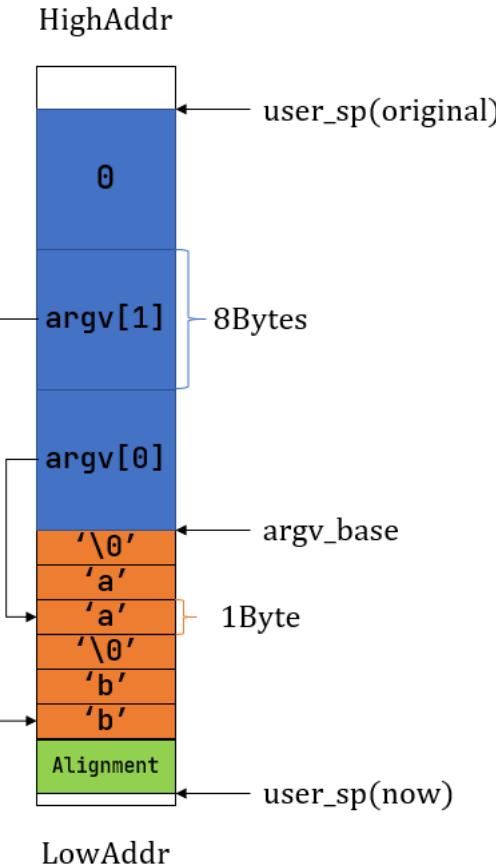
(续上页)

```

13 let argv_base = user_sp;
14 let mut argv: Vec<_> = (0..=args.len())
15 .map(|arg| {
16 translated_refmut(
17 memory_set.token(),
18 (argv_base + arg * core::mem::size_of::<u8>()) as *mut u8
19)
20 })
21 .collect();
22 *argv[args.len()] = 0;
23 for i in 0..args.len() {
24 user_sp -= args[i].len() + 1;
25 *argv[i] = user_sp;
26 let mut p = user_sp;
27 for c in args[i].as_bytes() {
28 *translated_refmut(memory_set.token(), p as *mut u8) = *c;
29 p += 1;
30 }
31 *translated_refmut(memory_set.token(), p as *mut u8) = 0;
32 }
33 // make the user_sp aligned to 8B for k210 platform
34 user_sp -= user_sp % core::mem::size_of::<u8>();
35
36 // **** hold current PCB lock
37 let mut inner = self.acquire_inner_lock();
38 // substitute memory_set
39 inner.memory_set = memory_set;
40 // update trap_cx_ppn
41 inner.trap_cx_ppn = trap_cx_ppn;
42 // initialize trap_cx
43 let mut trap_cx = TrapContext::app_init_context(
44 entry_point,
45 user_sp,
46 KERNEL_SPACE.lock().token(),
47 self.kernel_stack.get_top(),
48 trap_handler as u8,
49);
50 trap_cx.x[10] = args.len();
51 trap_cx.x[11] = argv_base;
52 *inner.get_trap_cx() = trap_cx;
53 // **** release current PCB lock
54 }
55

```

第 11-34 行所做的主要工作是将命令行参数以某种格式压入用户栈。具体的格式可以参考下图（比如应用传入了两个命令行参数 aa 和 bb）：



- 首先需要在用户栈上分配一个字符串指针数组，也就是蓝色区域。数组中的每个元素都指向一个用户栈更低处的命令行参数字符串的起始地址。在第 12~24 行可以看到，最开始我们只是分配空间，具体的值要等到字符串被放到用户栈上之后才能确定更新。
- 第 23~32 行，我们逐个将传入的 args 中的字符串压入到用户栈中，对应于图中的橙色区域。为了实现方便，我们在用户栈上预留空间之后逐字节进行复制。注意 args 中的字符串是通过 translated\_str 从应用地址空间取出的，它的末尾不包含 \0。为了应用能知道每个字符串的长度，我们需要手动在末尾加入 \0。
- 第 34 行将 user\_sp 以 8 字节对齐，即图中的绿色区域。这是因为命令行参数的长度不一，很有可能压入之后 user\_sp 没有对齐到 8 字节，那么在 K210 平台上在访问用户栈的时候就会触发访存不对齐的异常。在 Qemu 平台上则并不存在这个问题。

我们还需要对应修改 Trap 上下文。首先是第 45 行，我们的 user\_sp 相比之前已经发生了变化，它上面已经压入了命令行参数。同时，我们还需要修改 Trap 上下文中的 a0/a1 寄存器，让 a0 表示命令行参数的个数，而 a1 则表示图中 argv\_base 即蓝色区域的起始地址。这两个参数在第一次进入对应应用的用户态的时候会被接收并用于还原命令行参数。

## 用户库从用户栈上还原命令行参数

在应用第一次进入用户态的时候，我们放在 Trap 上下文 a0/a1 两个寄存器中的内容可以被用户库中的入口函数以参数的形式接收：

```

1 // user/src/lib.rs
2
3 #[no_mangle]
4 #[link_section = ".text.entry"]
5 pub extern "C" fn _start(argc: usize, argv: *const u8) -> ! {
6 unsafe {
7 HEAP.lock()
8 .init(HEAP_SPACE.as_ptr() as usize, USER_HEAP_SIZE);
9 }
10 let mut v: Vec<&'static str> = Vec::new();
11 for i in 0..argc {
12 let str_start = unsafe {
13 ((argv + i * core::mem::size_of::(<usize>()) as *const usize) .read_
14 volatile())
15 };
16 let len = (0..).find(|i| unsafe {
17 ((str_start + *i) as *const u8).read_volatile() == 0
18 }).unwrap();
19 v.push(
20 core::str::from_utf8(unsafe {
21 core::slice::from_raw_parts(str_start as *const u8, len)
22 }).unwrap()
23);
24 }
25 exit(main(argc, v.as_slice()));
}

```

可以看到，在入口 `_start` 中我们就接收到了命令行参数个数 `argc` 和字符串数组的起始地址 `argv`。但是这个起始地址不太好用，我们希望能够将其转化为编写应用的时候看到的 `&[&str]` 的形式。转化的主体在第 10~23 行，就是分别取出 `argc` 个字符串的起始地址（基于字符串数组的 `base` 地址 `argv`），从它向后找到第一个 `\0` 就可以得到一个完整的 `&str` 格式的命令行参数字符串并加入到向量 `v` 中。最后通过 `v.as_slice` 就得到了我们在 `main` 主函数中看到的 `&[&str]`。

## 通过命令行工具 `cat` 输出文件内容

有了之前的命令行参数支持，我们就可以编写命令行工具 `cat` 来输出指定文件的内容了。它的使用方法如下：

```

>> filetest_simple
file_test passed!
Shell: Process 2 exited with code 0
>> cat filea
Hello, world!
Shell: Process 2 exited with code 0
>>

```

`filetest_simple` 会将 `Hello, world!` 输出到文件 `filea` 中。之后我们就可以通过 `cat filea` 来打印文件 `filea` 中的内容。

`cat` 本身也是一个应用，且很容易实现：

```

1 // user/src/bin/cat.rs
2
3 #[no_std]
4 #[no_main]
5
6 #[macro_use]
7 extern crate user_lib;
8 extern crate alloc;
9
10 use user_lib::{
11 open,
12 OpenFlags,
13 close,
14 read,
15 };
16 use alloc::string::String;
17
18 #[no_mangle]
19 pub fn main(argc: usize, argv: &[&str]) -> i32 {
20 assert!(argc == 2);
21 let fd = open(argv[1], OpenFlags::RDONLY);
22 if fd == -1 {
23 panic!("Error occurred when opening file");
24 }
25 let fd = fd as usize;
26 let mut buf = [0u8; 16];
27 let mut s = String::new();
28 loop {
29 let size = read(fd, &mut buf) as usize;
30 if size == 0 { break; }
31 s.push_str(core::str::from_utf8(&buf[..size]).unwrap());
32 }
33 println!("{}", s);
34 close(fd);
35 0
36 }

```

### 8.4.3 标准输入输出重定向

为了进一步增强 shell 程序使用文件系统时的灵活性，我们需要新增标准输入输出重定向功能。这个功能在我们使用 Linux 内核的时候很常用，我们在自己的内核中举个例子：

```

>> yield > fileb
Shell: Process 2 exited with code 0
>> cat fileb
Hello, I am process 2.
Back in process 2, iteration 0.
Back in process 2, iteration 1.
Back in process 2, iteration 2.
Back in process 2, iteration 3.
Back in process 2, iteration 4.
yield pass.

Shell: Process 2 exited with code 0
>>

```

通过 `>` 我们可以将应用 `yield` 的输出重定向到文件 `fileb` 中。我们也可以注意到在屏幕上暂时看不到 `yield` 的输出了。在应用 `yield` 退出之后，我们可以使用 `cat` 工具来查看文件 `fileb` 的内容，可以看到里面的确是 `yield` 的输出。同理，通过 `<` 则可以将一个应用的输入重定向到某个指定文件而不是从键盘输入。

注意重定向功能对于应用来说是透明的。在应用中除非明确指出了数据要从指定的文件输入或者输出到指定的文件，否则数据默认都是输入自进程文件描述表位置 0（即 `fd=0`）处的标准输入，并输出到进程文件描述符表位置 1（即 `fd=1`）处的标准输出。这是由于内核在执行 `sys_exec` 系统调用创建基于新应用的进程时，会直接把文件描述符表位置 0 放置标准输入文件，位置 1 放置标准输出文件，位置 2 放置标准错误输出文件。标准输入/输出文件其实是把设备当成文件，标准输入文件就是串口的输入或键盘，而标准输出文件就是串口的输出或显示器。

因此，在应用执行之前，我们就要对应用进程的文件描述符表进行某种替换。以输出为例，我们需要提前打开文件并用这个文件来替换掉应用文件描述符表位置 1 处的标准输出文件，这就完成了所谓的重定向。在重定向之后，应用认为自己输出到 `fd=1` 的标准输出文件，但实际上输出到我们指定的文件中。我们能够做到这一点还是得益于文件的抽象，因为在进程看来无论是标准输出还是常规文件都是一种文件，可以通过同样的接口来读写。

为了实现重定向功能，我们需要引入一个新的系统调用 `sys_dup`：

```

1 // user/src/syscall.rs
2
3 /// 功能：将进程中一个已经打开的文件复制一份并分配到一个新的文件描述符中。
4 /// 参数：fd 表示进程中一个已经打开的文件的文件描述符。
5 /// 返回值：如果出现了错误则返回 -1，否则能够访问已打开文件的新文件描述符。
6 /// 可能的错误原因是：传入的 fd 并不对应一个合法的已打开文件。
7 /// syscall ID: 24
8 pub fn sys_dup(fd: usize) -> isize;

```

这个系统调用的实现非常简单：

```

1 // os/src/syscall/fs.rs
2
3 pub fn sys_dup(fd: usize) -> isize {
4 let task = current_task().unwrap();
5 let mut inner = task.acquire_inner_lock();
6 if fd >= inner.fd_table.len() {
7 return -1;
8 }
9 if inner.fd_table[fd].is_none() {
10 return -1;
11 }
12 let new_fd = inner.alloc_fd();
13 inner.fd_table[new_fd] = Some(Arc::clone(inner.fd_table[fd].as_ref().unwrap()));
14 new_fd as isize
15}

```

在 `sys_dup` 函数中，首先检查传入 `fd` 的合法性。然后在文件描述符表中分配一个新的文件描述符，并保存 `fd` 指向的已打开文件的一份拷贝即可。

那么我们应该在什么时候进行替换，又应该如何利用 `sys_dup` 进行替换呢？

答案是在 `shell` 程序 `user_shell` 中进行处理。在分割命令行参数的时候，我们要检查是否存在通过 `<` 或 `>` 进行输入输出重定向的情况，如果存在的话则需要将它们从命令行参数中移除，并记录匹配到的输入文件名或输出文件名到字符串 `input` 或 `output` 中。注意，为了实现方便，我们这里假设输入 `shell` 程序的命令一定合法：即 `<` 或 `>` 最多只会出现一次，且后面总是会有一个参数作为重定向到的文件。

```

1 // user/src/bin/user_shell.rs
2

```

(下页继续)

(续上页)

```

3 // redirect input
4 let mut input = String::new();
5 if let Some((idx, _)) = args_copy
6 .iter()
7 .enumerate()
8 .find(|(_, arg)| arg.as_str() == "<\0") {
9 input = args_copy[idx + 1].clone();
10 args_copy.drain(idx..=idx + 1);
11 }
12
13 // redirect output
14 let mut output = String::new();
15 if let Some((idx, _)) = args_copy
16 .iter()
17 .enumerate()
18 .find(|(_, arg)| arg.as_str() == ">\0") {
19 output = args_copy[idx + 1].clone();
20 args_copy.drain(idx..=idx + 1);
21 }

```

打开文件和替换的过程则发生在 fork 之后的子进程分支中：

```

1 // user/src/bin/user_shell.rs
2
3 let pid = fork();
4 if pid == 0 {
5 // input redirection
6 if !input.is_empty() {
7 let input_fd = open(input.as_str(), OpenFlags::RDONLY);
8 if input_fd == -1 {
9 println!("Error when opening file {}", input);
10 return -4;
11 }
12 let input_fd = input_fd as usize;
13 close(0);
14 assert_eq!(dup(input_fd), 0);
15 close(input_fd);
16 }
17 // output redirection
18 if !output.is_empty() {
19 let output_fd = open(
20 output.as_str(),
21 OpenFlags::CREATE | OpenFlags::WRONLY
22);
23 if output_fd == -1 {
24 println!("Error when opening file {}", output);
25 return -4;
26 }
27 let output_fd = output_fd as usize;
28 close(1);
29 assert_eq!(dup(output_fd), 1);
30 close(output_fd);
31 }
32 // child process
33 if exec(args_copy[0].as_str(), args_addr.as_slice()) == -1 {
34 println!("Error when executing!");

```

(下页继续)

(续上页)

```

35 return -4;
36 }
37 unreachable!();
38 } else {
39 let mut exit_code: i32 = 0;
40 let exit_pid = waitpid(pid as usize, &mut exit_code);
41 assert_eq!(pid, exit_pid);
42 println!("Shell: Process {} exited with code {}", pid, exit_code);
43 }

```

- 输入重定向发生在第 6~16 行。我们尝试打开输入文件 `input` 到 `input_fd` 中。之后，首先通过 `close` 关闭标准输入所在的文件描述符 0。之后通过 `dup` 来分配一个新的文件描述符来访问 `input_fd` 对应的输入文件。这里用到了文件描述符分配的重要性质：即必定分配可用描述符中编号最小的一个。由于我们刚刚关闭了描述符 0，那么在 `dup` 的时候一定会将它分配出去，于是现在应用进程的文件描述符 0 就对应到输入文件了。最后，因为应用进程的后续执行不会用到输入文件原来的描述符 `input_fd`，所以就将其关掉。
- 输出重定向则发生在 18~31 行。它的原理和输入重定向几乎完全一致，只是通过 `open` 打开文件的标志不太相同。

实现到这里，就可以通过 `exec` 来执行应用了。

## 8.4.4 小结

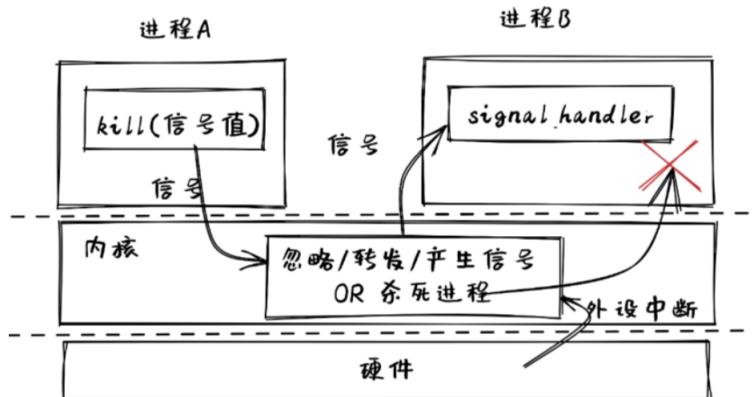
虽然 `fork/exec/waitpid` 三个经典的系统调用自它们于古老的 UNIX 时代诞生以来已经过去了太长时间，从某种程度上来讲已经不太适合新的内核环境了。人们也已经提出了若干种替代品并已经在进行实践，比如 POSIX 标准中的 `posix_spawn` 或者 Linux 上的 `clone` 系统调用。但是它们迄今为止仍然存在就证明在它们的设计中还能够找到可取之处。从本节介绍的重定向就可以看出它们的灵活性以及强大的功能性：我们能够进行重定向恰恰是因为创建新应用进程分为 `fork` 和 `exec` 两个系统调用，那么在这两个系统调用之间我们就能够进行一些类似重定向的处理。在实现的过程中，我们还用到了 `fork` 出来的子进程会和父进程共享文件描述符表的性质。

## 8.5 信号

### 8.5.1 本节导读

在本节之前的 IPC 机制主要集中在进程间的数据传输和数据交换方面，这需要两个进程之间相互合作，同步地来实现。比如，一个进程发出 `read` 系统调用，另外一个进程需要发出对应的 `write` 系统调用，这样两个进程才能协同完成基于 `pipe` 机制的数据传输。这种双向协作的方式不太适合单向的事件通知机制。

在进程间还存在“事件通知”的需求：操作系统或某进程希望能单方面通知另外一个正在忙其它事情的进程产生了某个事件，并让这个进程能迅速响应。如果采用之前同步的 IPC 机制，难以高效地应对这样的需求。比如，用户想中断当前正在运行的一个程序，于是他敲击 `Ctrl-C` 的组合键，正在运行的程序会迅速退出它正在做的任何事情，截止程序的执行。



我们需要有一种类似于硬件中断的软件级异步通知机制，使得进程在接收到特定事件的时候能够暂停当前的工作并及时响应事件，并在响应事件之后可以恢复当前工作继续执行。如果进程没有接收到任何事件，它可以执行自己的任务。这里的暂停与恢复的工作，都由操作系统来完成，应用程序只需设置好响应某事件的事件处理例程就够了。这在很大程度上简化了应用程序响应事件的开发工作。这些需求和想法推动了信号 (Signal) 机制的产生。

## 8.5.2 信号机制简介

信号 (Signals) 是类 UNIX 操作系统中实现进程间通信的一种异步通知机制，用来提醒某进程一个特定事件已经发生，需要及时处理。当一个信号发送给一个进程时，操作系统会中断接收到信号的进程的正常执行流程并对信号进行处理。如果该进程定义了信号的处理函数，那么这个处理函数会被调用，否则就执行默认的处理行为，比如让该进程退出。在处理完信号之后，如果进程还没有退出，则会恢复并继续进程的正常执行。

如果将信号与硬件中断进行比较，我们可以把信号描述为软件中断。当硬件发出中断后，中断响应的对象是操作系统，并由操作系统预设的中断处理例程来具体地进行中断的响应和处理；对于信号来说，当某进程或操作系统发出信号时，会指定信号响应的对象，即某个进程的 pid，并由该进程预设的信号处理例程来进行具体的信号响应。

进程间发送的信号是某种事件，为了简单起见，UNIX 采用了整数来对信号进行编号，这些整数编号都定义了对应的信号的宏名，宏名都是以 SIG 开头，比如 SIGABRT, SIGKILL, SIGSTOP, SIGCONT。

信号的发送方可以是进程或操作系统内核：进程可以通过系统调用 kill 给其它进程发信号；内核在碰到特定事件，比如用户对当前进程按下 Ctrl+C 按键时，内核会收到包含 Ctrl+C 按键的外设中断和按键信息，随后会向正在运行的当前进程发送 SIGINT 信号，将其终止。

信号的接收方是一个进程，接收到信号有多种处理方式，最常见的三种如下：

- 忽略：就像信号没有发生过一样。
- 捕获：进程会调用相应的处理函数进行处理。
- 终止：终止进程。

如果应用没有手动设置接收到某种信号之后如何处理，则操作系统内核会以默认方式处理该信号，一般是终止收到信号的进程或者忽略此信号。每种信号都有自己的默认处理方式。

### 注解：Linux 有哪些信号？

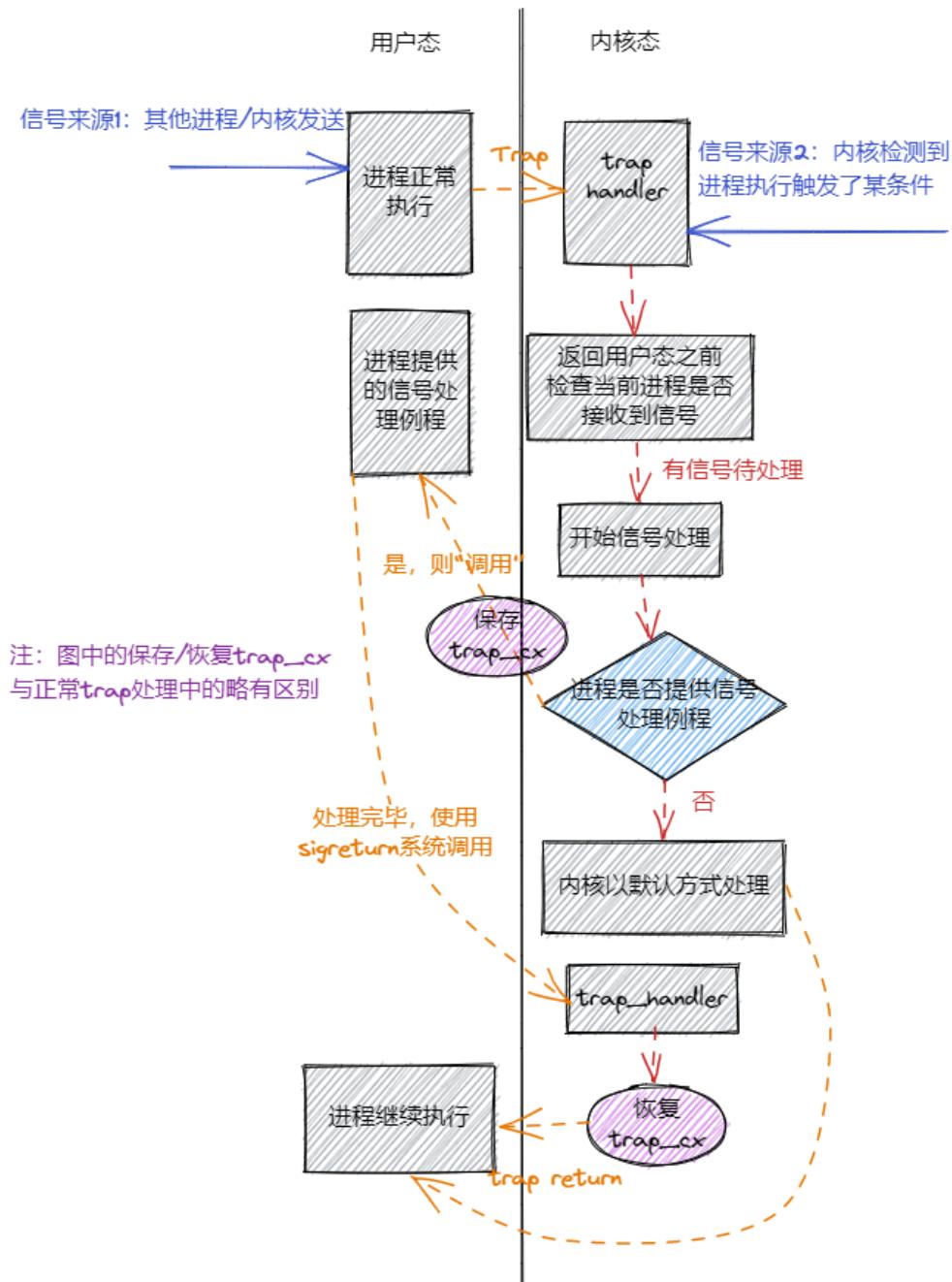
Linux 中有 62 个信号，每个信号代表着某种事件，一般情况下，当进程收到某个信号时，意味着该信号所代表的事件发生了。下面列出了一些常见的信号。

| 信号         | 含义                                                                                               |
|------------|--------------------------------------------------------------------------------------------------|
| SIGABRT    | 非正常的进程退出, 可能由调用 <code>abort</code> 函数产生                                                          |
| SIGCHLD    | 进程状态变更时 (通常是进程退出时), 由内核发送给它的父进程                                                                  |
| SIGINT     | 在终端界面按下 <code>CTRL+C</code> 组合键时, 由内核会发送给当前终端的前台进程                                               |
| SIGKILL    | 终止某个进程, 由内核或其他进程发送给被终止进程                                                                         |
| SIGSEGV    | 非法内存访问异常, 由内核发送给触发异常的进程                                                                          |
| SIGILL     | 非法指令异常, 由内核发送给触发异常的进程                                                                            |
| SIGTSTP    | 在终端界面按下 <code>CTRL+Z</code> 组合键时, 会发送给当前进程让它暂停                                                   |
| SIGSTOP    | 也用于暂停进程, 与 <code>SIGTSTP</code> 的区别在于 <code>SIGSTOP</code> 不能被忽略或捕获, 即 <code>SIGTSTP</code> 更加灵活 |
| SIGCONT    | 恢复暂停的进程继续执行                                                                                      |
| SI-GUSR1/2 | 用户自定义信号 1/2                                                                                      |

和之前介绍过的硬件中断一样, 信号作为软件中断也可以分成同步和异步两种, 这里的同步/异步指的是信号的触发同步/异步于接收到信号进程的执行。比如 `SIGILL` 和 `SIGSEGV` 就属于同步信号, 而 `SIGCHLD` 和 `SIGINT` 就属于异步信号。

### 8.5.3 信号处理流程

信号的处理流程如下图所示:



信号有两种来源：最开始的时候进程在正常执行，此时可能内核或者其他进程给它发送了一个信号，这些就属于异步信号，是信号的第一种来源；信号的第二种来源则是由进程自身的执行触发，在处理 Trap 的时候内核会将相应的信号直接附加到进程控制块中，这种属于同步信号。

内核会在 Trap 处理完成即将返回用户态之前检查要返回到的进程是否还有信号待处理。如果需要处理的话，取决于进程是否提供该种信号的处理函数，有两种处理方法：

- 如果进程通过下面介绍的 `sigaction` 系统调用提供了相应信号的处理函数，那么内核会将该进程 Trap 进来时留下的 Trap 上下文保存在另一个地方，并回到用户态执行进程提供的处理函数。内核要求处理函数的编写者在函数的末尾手动进行另一个 `sigreturn` 系统调用，表明处理结束并请求恢复进程原来的执行。内核将处理该系统调用并恢复之前保存的 Trap 上下文，等到再次回到用户态的时候，便会继续进程在处理信号之前的执行。

- 反之，如果进程未提供处理函数，这是一种比较简单的情况。此时，内核会直接默认的方式处理信号。之后便会回到用户态继续进程原先的执行。

## 8.5.4 信号机制系统调用原型

### 发送信号

为了与其他进程进行通信，一个进程可以使用 `kill` 系统调用发送信号给另一个进程：

```
// user/src/lib.rs

/// 功能：当前进程向另一个进程（可以是自身）发送一个信号。
/// 参数：pid 表示接受信号的进程的进程 ID, signum 表示要发送的信号的编号。
/// 返回值：如果传入参数不正确（比如指定进程或信号类型不存在）则返回 -1，否则返回 0。
/// syscall ID: 129
pub fn kill(pid: usize, signum: i32) -> isize;
```

我们的内核中各信号的编号定义如下：

```
// user/src/lib.rs

pub const SIGDEF: i32 = 0; // Default signal handling
pub const SIGHUP: i32 = 1;
pub const SIGINT: i32 = 2;
pub const SIGQUIT: i32 = 3;
pub const SIGILL: i32 = 4;
pub const SIGTRAP: i32 = 5;
pub const SIGABRT: i32 = 6;
pub const SIGBUS: i32 = 7;
pub const SIGFPE: i32 = 8;
pub const SIGKILL: i32 = 9;
...
```

从中可以看出，每次调用 `kill` 只能发送一个类型的信号。

### 处理信号

与信号处理相关的系统调用则有三个：

- `sys_sigaction`：设置信号处理例程
- `sys_procmask`：设置进程的信号屏蔽掩码
- `sys_sigreturn`：清除栈帧，从信号处理例程返回

下面依次对它们进行说明。

首先，进程可以通过 `sigaction` 系统调用捕获某种信号，即：当接收到某种信号的时候，暂停进程当前的执行，调用进程为该种信号提供的函数对信号进行处理，处理完成之后再恢复进程原先的执行。`sigaction` 的接口如下：

```
// os/src/syscall/process.rs

/// 功能：为当前进程设置某种信号的处理函数，同时保存设置之前的处理函数。
/// 参数：signum 表示信号的编号，action 表示要设置成的处理函数的指针
/// old_action 表示用于保存设置之前的处理函数的指针（SignalAction 结构稍后介绍）。
```

(下页继续)

(续上页)

```
/// 返回值: 如果传入参数错误 (比如传入的 action 或 old_action 为空指针或者)
/// 信号类型不存在返回 -1, 否则返回 0。
/// syscall ID: 134
pub fn sys_sigaction(
 signum: i32,
 action: *const SignalAction,
 old_action: *mut SignalAction,
) -> isize;
```

为了让编写应用更加方便, 用户库 user\_lib 中的接口略有不同:

```
// user/src/lib.rs

pub fn sigaction(
 signum: i32,
 action: Option<&SignalAction>,
 old_action: Option<&mut SignalAction>,
) -> isize;
```

注意这里参数 action 和 old\_action 使用引用而非裸指针, 且有一层 Option 包裹, 这样能减少对于不安全的裸指针的使用。在传参的时候, 如果传递实际存在的引用则使用 Some 包裹, 而用 None 来代替空指针, 这样可以提前对引用和空指针做出区分。在具体实现的时候, 再将 None 转换为空指针:

```
// user/src/lib.rs

pub fn sigaction(
 signum: i32,
 action: Option<&SignalAction>,
 old_action: Option<&mut SignalAction>,
) -> isize {
 sys_sigaction(
 signum,
 action.map_or(core::ptr::null(), |a| a),
 old_action.map_or(core::ptr::null_mut(), |a| a)
)
}
```

接下来介绍 SignalAction 数据结构。方便起见, 我们将其对齐到 16 字节使得它不会跨虚拟页面:

```
// user/src/lib.rs

/// Action for a signal
#[repr(C, align(16))]
#[derive(Debug, Clone, Copy)]
pub struct SignalAction {
 pub handler: usize,
 pub mask: SignalFlags,
}
```

可以看到它有两个字段: handler 表示信号处理例程的入口地址; mask 则表示执行该信号处理例程期间的信号掩码。这个信号掩码是用于在执行信号处理例程的期间屏蔽掉一些信号, 每个 handler 都可以设置它在执行期间屏蔽掉哪些信号。“屏蔽”的意思是指在执行该信号处理例程期间, 即使 Trap 到内核态发现当前进程又接收到一些信号, 只要这些信号被屏蔽, 内核就不会对这些信号进行处理而是直接回到用户态继续执行信号处理例程。但这不意味着这些被屏蔽的信号就此被忽略, 它们仍被记录在进程控制块中, 当信号处理例程执行结束之后它们便不再被屏蔽, 从而后续可能被处理。

mask 作为一个掩码可以代表屏蔽掉一组信号, 因此它的类型 SignalFlags 是一个信号集合:

```
// user/src/lib.rs

bitflags! {
 pub struct SignalFlags: i32 {
 const SIGDEF = 1; // Default signal handling
 const SIGHUP = 1 << 1;
 const SIGINT = 1 << 2;
 const SIGQUIT = 1 << 3;
 const SIGILL = 1 << 4;
 const SIGTRAP = 1 << 5;
 ...
 const SIGSYS = 1 << 31;
 }
}
```

需要注意的是，我们目前的实现比较简单，暂时不支持信号嵌套，也就是在执行一个信号处理例程期间再去执行另一个信号处理例程。

`sigaction` 可以设置某个信号处理例程的信号掩码，而 `sigprocmask` 是设置这个进程的全局信号掩码：

```
// user/src/lib.rs

/// 功能：设置当前进程的全局信号掩码。
/// 参数：mask 表示当前进程要设置成的全局信号掩码，代表一个信号集合，
/// 在集合中的信号始终被该进程屏蔽。
/// 返回值：如果传入参数错误返回 -1，否则返回之前的信号掩码。
/// syscall ID: 135
pub fn sigprocmask(mask: u32) -> isize;
```

最后一个系统调用是 `sigreturn`。介绍信号处理流程的时候提到过，在进程向内核提供的信号处理例程末尾，函数的编写者需要手动插入一个 `sigreturn` 系统调用来通知内核信号处理过程结束，可以恢复进程先前的执行。它的接口如下：

```
// user/src/lib.rs

/// 功能：进程通知内核信号处理例程退出，可以恢复原先的进程执行。
/// 返回值：如果出错返回 -1，否则返回 0。
/// syscall ID: 139
pub fn sigreturn() -> isize;
```

## 信号系统调用使用示例

我们来从简单的信号例子 `sig_simple` 中介绍如何使用信号机制：

```
1 // user/src/bin/sig_simple.rs
2
3 #![no_std]
4 #![no_main]
5
6 extern crate user_lib;
7
8 // use user_lib:::{sigaction, sigprocmask, SignalAction, SignalFlags, fork, exit, wait,
9 // → kill, getpid, sleep, sigreturn};
10 use user_lib::::*;
11
12 fn func() {
```

(下页继续)

(续上页)

```

12 println!("user_sig_test passed");
13 sigreturn();
14 }
15
16 #[no_mangle]
17 pub fn main() -> i32 {
18 let mut new = SignalAction::default();
19 let mut old = SignalAction::default();
20 new.handler = func as usize;
21
22 println!("signal_simple: sigaction");
23 if sigaction(SIGUSR1, Some(&new), Some(&mut old)) < 0 {
24 panic!("Sigaction failed!");
25 }
26 println!("signal_simple: kill");
27 if kill(getpid() as usize, SIGUSR1) < 0 {
28 println!("Kill failed!");
29 exit(1);
30 }
31 println!("signal_simple: Done");
32 0
33 }

```

在此进程中：

- 在第 18~20 行，首先建立了 new 和 old 两个 SignalAction 结构的变量，并设置 new.handler 为信号处理例程 func 的地址。
- 然后在第 23 行，调用 sigaction 函数，提取 new 结构中的信息设置当前进程收到 SIGUSR1 信号之后的处理方式，其效果是该进程在收到 SIGUSR1 信号后，会执行 func 函数来具体处理响应此信号。
- 接着在第 27 行，通过 getpid 函数获得自己的 pid，并以自己的 pid 和 SIGUSR1 为参数，调用 kill 函数，给自己发 SIGUSR1 信号。

执行这个应用，可以看到下面的输出：

```

>> sig_simple
signal_simple: sigaction
signal_simple: kill
user_sig_test passed
signal_simple: Done

```

可以看出，看到进程在收到自己给自己发送的 SIGUSR1 信号之后，内核调用它作为信号处理例程的 func 函数并打印出了标志性输出。在信号处理例程结束之后，还能够看到含有 Done 的输出，这意味着进程原先的执行被正确恢复。

## 8.5.5 设计与实现信号机制

我们将信号机制的实现划分为两部分：

- 一是进程通过 sigaction 系统调用设置信号处理例程和通过 sigprocmask 设置进程全局信号掩码。这些操作只需简单修改进程控制块中新增的相关数据结构即可，比较简单。
- 二是如何向进程发送信号、进程如何接收信号、而信号又如何被处理，这些操作需要结合到本书前面的章节介绍的对于 Trap 处理流程，因此会比较复杂。

## 设置信号处理例程和信号掩码

为了实现进程设置信号处理例程和信号掩码的功能，我们需要在进程控制块 TaskControlBlock 中新增以下数据结构（这些数据结构在进程创建之后可能被修改，因此将它们放置在内部可变的 inner 中）：

```
// os/src/task/task.rs

pub struct TaskControlBlockInner {
 ...
 pub signal_mask: SignalFlags,
 pub signal_actions: SignalActions,
 ...
}
```

其中，signal\_mask 表示进程的全局信号掩码，其类型 SignalFlags 与用户库 user\_lib 中的相同，表示一个信号集合。在 signal\_mask 这个信号集合内的信号将被该进程全局屏蔽。

进程可以通过 sigprocmask 系统调用直接设置自身的全局信号掩码：

```
// os/src/process.rs

pub fn sys_sigprocmask(mask: u32) -> isize {
 if let Some(task) = current_task() {
 let mut inner = task.inner_exclusive_access();
 let old_mask = inner.signal_mask;
 if let Some(flag) = SignalFlags::from_bits(mask) {
 inner.signal_mask = flag;
 old_mask.bits() as isize
 } else {
 -1
 }
 } else {
 -1
 }
}
```

进程控制块中的 signal\_actions 的类型是 SignalActions，是一个 SignalAction（同样与 user\_lib 中的定义相同）的定长数组，其中每一项都记录进程如何响应对应的信号：

```
// os/src/task/signal.rs

pub const MAX_SIG: usize = 31;

// os/src/task/action.rs

#[derive(Clone)]
pub struct SignalActions {
 pub table: [SignalAction; MAX_SIG + 1],
}
```

于是，在 sigaction 系统调用的时候我们只需要更新当前进程控制块的 signal\_actions 即可：

```
1 // os/src/syscall/process.rs
2
3 fn check_sigaction_error(signal: SignalFlags, action: usize, old_action: usize) ->_
4 bool {
5 if action == 0
6 || old_action == 0
```

(下页继续)

(续上页)

```

6 || signal == SignalFlags::SIGKILL
7 || signal == SignalFlags::SIGSTOP
8 {
9 true
10 } else {
11 false
12 }
13 }

14
15 pub fn sys_sigaction(
16 signum: i32,
17 action: *const SignalAction,
18 old_action: *mut SignalAction,
19) -> isize {
20 let token = current_user_token();
21 let task = current_task().unwrap();
22 let mut inner = task.inner_exclusive_access();
23 if signum as usize > MAX_SIG {
24 return -1;
25 }
26 if let Some(flag) = SignalFlags::from_bits(1 << signum) {
27 if check_sigaction_error(flag, action as usize, old_action as usize) {
28 return -1;
29 }
30 let prev_action = inner.signal_actions.table[signum as usize];
31 *translated_refmut(token, old_action) = prev_action;
32 inner.signal_actions.table[signum as usize] = *translated_ref(token, action);
33 0
34 } else {
35 -1
36 }
37 }

```

其中：

- `check_sigaction_error` 用来检查 `sigaction` 的参数是否有错误（有错误的话返回 `true`）。这里的检查比较简单，如果传入的 `action` 或者 `old_action` 为空指针则视为错误。另一种错误则是信号类型为 `SIGKILL` 或者 `SIGSTOP`，这是因为我们的内核参考 Linux 内核规定不允许进程对这两种信号设置信号处理例程，而只能由内核对它们进行处理。
- `sys_sigaction` 首先会调用 `check_sigaction_error` 进行检查，如果没有错误的话，则会使用 `translated_ref(mut)` 将进程提交的信号处理例程保存到进程控制块，随后将此前的处理例程保存到进程中的指定位置。注意使用 `translated_ref(mut)` 的前提是类型 `T` 不会跨页，我们通过设置 `SignalAction` 对齐到 16 字节来保证这一点。

## 信号的产生

信号的产生有以下几种方式：

1. 进程通过 `kill` 系统调用给自己或者其他进程发送信号。
2. 内核检测到某些事件给某个进程发送信号，但这个事件与接收信号的进程的执行无关。典型的例子如：`SIGCHLD` 当子进程的状态改变后由内核发送给父进程。可以看出这可以用来实现更加灵活的进程管理，但我们的内核为了简单目前并没有实现 `SIGCHLD` 这类信号。
3. 前两种属于异步信号，最后一种则属于同步信号：即进程执行的时候触发了某些条件，于是在 Trap 到内核处理的时候，内核给该进程发送相应的信号。比较常见的例子是进程执行的时候出错，比如段错

误 SIGSEGV 和非法指令异常 SIGILL。

首先来看 kill 系统调用的实现：

```
// os/src/task/task.rs

pub struct TaskControlBlockInner {
 ...
 pub signals: SignalFlags,
 ...
}

// os/src/syscall/process.rs

pub fn sys_kill(pid: usize, signum: i32) -> isize {
 if let Some(task) = pid2task(pid) {
 if let Some(flag) = SignalFlags::from_bits(1 << signum) {
 // insert the signal if legal
 let mut task_ref = task.inner_exclusive_access();
 if task_ref.signals.contains(flag) {
 return -1;
 }
 task_ref.signals.insert(flag);
 0
 } else {
 -1
 }
 } else {
 -1
 }
}
```

这需要在进程控制块的可变部分中新增一个 signals 字段记录对应进程目前已经收到了哪些信号尚未处理，它的类型同样是 SignalFlags 表示一个信号集合。sys\_kill 的实现也比较简单：就是调用 pid2task 得到传入进程 ID 对应的进程控制块，然后把要发送的信号插入到 signals 字段中。

然后是进程执行出错的情况（比如访存错误或非法指令异常），这会 Trap 到内核并在 trap\_handler 中由内核将对应信号发送到当前进程：

```
// os/src/trap/mod.rs

#[no_mangle]
pub fn trap_handler() -> ! {
 ...
 match scause.cause() {
 ...
 Trap::Exception(Exception::StoreFault)
 | Trap::Exception(Exception::StorePageFault)
 | Trap::Exception(Exception::InstructionFault)
 | Trap::Exception(Exception::InstructionPageFault)
 | Trap::Exception(Exception::LoadFault)
 | Trap::Exception(Exception::LoadPageFault) => {
 /*
 println!(
 "[kernel] {:?} in application, bad addr = {:#x}, bad instruction = {:#x}, kernel killed it.",
 scause.cause(),
 stval,
)
 }
 }
}
```

(下页继续)

(续上页)

```

 current_trap_cx().sepc,
);
/*
 current_add_signal(SignalFlags::SIGSEGV);
}
Trap::Exception(Exception::IllegalInstruction) => {
 current_add_signal(SignalFlags::SIGILL);
...
}
...
}

// os/src/task/mod.rs

pub fn current_add_signal(signal: SignalFlags) {
 let task = current_task().unwrap();
 let mut task_inner = task.inner_exclusive_access();
 task_inner.signals |= signal;
}

```

## 信号的处理

在 trap\_handler 完成 Trap 处理并返回用户态之前，会调用 handle\_signals 函数处理当前进程此前接收到的信号：

```

// os/src/task/task.rs

pub struct TaskControlBlockInner {
 ...
 pub killed: bool,
 pub frozen: bool,
 ...
}

// os/src/task/mod.rs

pub fn handle_signals() {
 loop {
 check_pending_signals();
 let (frozen, killed) = {
 let task = current_task().unwrap();
 let task_inner = task.inner_exclusive_access();
 (task_inner.frozen, task_inner.killed)
 };
 if !frozen || killed {
 break;
 }
 suspend_current_and_run_next();
 }
}

```

可以看到 handle\_signals 是一个无限循环，真正处理信号的逻辑在 check\_pending\_signals 函数中。这样做是为了处理 SIGSTOP 和 SIGCONT 这一对信号：当进程收到 SIGSTOP 信号之后，它的执行将被暂停，等到该进程收到 SIGCONT 信号之后再继续执行。我们在进程控制块中新增 frozen 字段表示进程目前是否已收到 SIGSTOP 信号被暂停，而 killed 字段表示进程是否已被杀死。这个循环的意义在于：只要进程还

处于暂停且未被杀死的状态就会停留在循环中等待 SIGCONT 信号的到来。如果 frozen 为真，证明还没有收到 SIGCONT 信号，进程仍处于暂停状态，循环的末尾我们调用 suspend\_current\_and\_run\_next 函数切换到其他进程期待其他进程将 SIGCONT 信号发过来。

check\_pending\_signals 会检查收到的信号并对它们进行处理，在这个过程中会更新上面的 frozen 和 killed 字段：

```

1 // os/src/task/mod.rs
2
3 fn check_pending_signals() {
4 for sig in 0..(MAX_SIG + 1) {
5 let task = current_task().unwrap();
6 let task_inner = task.inner_exclusive_access();
7 let signal = SignalFlags::from_bits(1 << sig).unwrap();
8 if task_inner.signals.contains(signal) && (!task_inner.signal_mask.
9 contains(signal)) {
10 let mut masked = true;
11 let handling_sig = task_inner.handling_sig;
12 if handling_sig == -1 {
13 masked = false;
14 } else {
15 let handling_sig = handling_sig as usize;
16 if !task_inner.signal_actions.table[handling_sig]
17 .mask
18 .contains(signal)
19 {
20 masked = false;
21 }
22 }
23 if !masked {
24 drop(task_inner);
25 drop(task);
26 if signal == SignalFlags::SIGKILL
27 || signal == SignalFlags::SIGSTOP
28 || signal == SignalFlags::SIGCONT
29 || signal == SignalFlags::SIGDEF
30 {
31 // signal is a kernel signal
32 call_kernel_signal_handler(signal);
33 } else {
34 // signal is a user signal
35 call_user_signal_handler(sig, signal);
36 return;
37 }
38 }
39 }
40 }
}

```

- 第 4 行的最外层循环遍历所有信号；
- 第 8 行检查当前进程是否接收到遍历到的信号（条件 1）以及该信号是否未被当前进程全局屏蔽（条件 2）；
- 第 9 ~ 21 行检查该信号是否未被当前正在执行的信号处理例程屏蔽（条件 3）；
- 当 3 个条件全部满足的时候，则在第 23 ~ 36 行开始处理该信号。目前的设计是：如果信号类型为 SIGKILL/SIGSTOP/SIGCONT/SIGDEF 四者之一，则该信号只能由内核来处理，调用 call\_kernel\_signal\_handler 函数来处理；否则调用 call\_user\_signal\_handler 函数尝试使用进程提供的信号处理例程来处理。

```

// os/src/task/task.rs

pub struct TaskControlBlockInner {
 ...
 pub handling_sig: isize,
 pub trap_ctx_backup: Option<TrapContext>,
 ...
}

// os/src/task/mod.rs

fn call_kernel_signal_handler(signal: SignalFlags) {
 let task = current_task().unwrap();
 let mut task_inner = task.inner_exclusive_access();
 match signal {
 SignalFlags::SIGSTOP => {
 task_inner.frozen = true;
 task_inner.signals ^= SignalFlags::SIGSTOP;
 }
 SignalFlags::SIGCONT => {
 if task_inner.signals.contains(SignalFlags::SIGCONT) {
 task_inner.signals ^= SignalFlags::SIGCONT;
 task_inner.frozen = false;
 }
 }
 _ => {
 // println!(
 // "[K] call_kernel_signal_handler:: current task sigflag {:?}",
 // task_inner.signals
 //);
 task_inner.killed = true;
 }
 }
}

fn call_user_signal_handler(sig: usize, signal: SignalFlags) {
 let task = current_task().unwrap();
 let mut task_inner = task.inner_exclusive_access();

 let handler = task_inner.signal_actions.table[sig].handler;
 if handler != 0 {
 // user handler

 // handle flag
 task_inner.handling_sig = sig as isize;
 task_inner.signals ^= signal;

 // backup trapframe
 let mut trap_ctx = task_inner.get_trap_cx();
 task_inner.trap_ctx_backup = Some(*trap_ctx);

 // modify trapframe
 trap_ctx.sepc = handler;

 // put args (a0)
 trap_ctx.x[10] = sig;
 } else {
}
}

```

(下页继续)

(续上页)

```

 // default action
 println!("[K] task/call_user_signal_handler: default action: ignore it or_
 ↵kill process");
}
}
}

```

- `call_kernel_signal_handler` 对于 `SIGSTOP` 和 `SIGCONT` 特殊处理：清除掉接收到的信号避免它们再次被处理，然后更新 `frozen` 字段；对于其他的信号都按照默认的处理方式即杀死当前进程，于是将 `killed` 字段设置为真，这样的进程会在 `Trap` 返回用户态之前就通过调度切换到其他进程。
- 在实现 `call_user_signal_handler` 之前，还需在进程控制块中新增两个字段：`handling_sig` 表示进程正在执行哪个信号的处理例程；`trap_ctx_backup` 则表示进程执行信号处理例程之前的 `Trap` 上下文。因为我们要 `Trap` 回到用户态执行信号处理例程，原来的 `Trap` 上下文会被覆盖，所以我们将其保存在进程控制块中。
- `call_user_signal_handler` 首先检查进程是否提供了该信号的处理例程，如果没有提供的话直接忽略该信号。否则就调用信号处理例程：除了更新 `handling_sig` 和 `signals` 之外，还将当前的 `Trap` 上下文保存在 `trap_ctx_backup` 中。然后修改 `Trap` 上下文的 `sepc` 到应用设置的例程地址使得 `Trap` 回到用户态之后就会跳转到例程入口并开始执行。注意我们并没有修改 `Trap` 上下文中的 `sp`，这意味着例程还会在原先的用户栈上执行。这是为了实现方便，在 Linux 的实现中，内核会为每次例程的执行重新分配一个用户栈。最后，我们修改 `Trap` 上下文的 `a0` 寄存器，使得信号类型能够作为参数被例程接收。

回到 `handle_signals`，从 `handle_signals` 退出之后会回到 `trap_handler` 中，这里在回到用户态之前会检查当前进程是否出错并可以退出：

```

// os/src/trap/mod.rs

#[no_mangle]
pub fn trap_handler() -> ! {
 ...
 handle_signals();

 // check error signals (if error then exit)
 if let Some((errno, msg)) = check_signals_error_of_current() {
 println!("[kernel] {}", msg);
 exit_current_and_run_next(errno);
 }
 trap_return();
}

```

这里的错误涵盖了 `SIGINT`/`SIGSEGV`/`SIGILL` 等，可以看 `check_signals_error_of_current` 的实现。出错之后会直接打印信息并调用 `exit_current_and_run_next` 退出当前进程并进行调度。

最后还需要补充 `sigreturn` 的实现。在信号处理例程的结尾需要插入这个系统调用来结束信号处理并继续进程原来的执行：

```

1 // os/src/syscall/process.rs
2
3 pub fn sys_sigreturn() -> isize {
4 if let Some(task) = current_task() {
5 let mut inner = task.inner_exclusive_access();
6 inner.handling_sig = -1;
7 // restore the trap context
8 let trap_ctx = inner.get_trap_cx();
9 *trap_ctx = inner.trap_ctx_backup.unwrap();
10 trap_ctx.x[10] as isize
}

```

(下页继续)

(续上页)

```

11 } else {
12 -1
13 }
14 }
```

这里只是将进程控制块中保存的记录了处理信号之前的进程上下文的 `trap_ctx_backup` 覆盖到当前的 Trap 上下文。这样接下来 Trap 回到用户态就会继续原来进程的执行了。注意在第 10 行，我们将 `trap_ctx` 中的 `a0` 的值作为系统调用返回值而不是使用 0 这种特定值，不然的话，在返回用户态恢复 Trap 上下文的时候，原来进程上下文中的 `a0` 寄存器将会被这些特定值覆盖，使得进程无法在信号处理完成后恢复正常执行。

## 8.5.6 小结

信号作为一种软件中断机制和硬件中断有很多相似之处：比如它们都可以用某种方式屏蔽，还细分为全局屏蔽和局部屏蔽；它们的处理都有一定延迟，硬件中断每个 CPU 周期仅在固定的阶段被检查，而信号只有在 Trap 陷入内核态之后才被检查并处理。这个延迟还与屏蔽、优先级和软件或硬件层面的调度策略有关。

这里仅仅给出了一个基本的信号机制的使用和实现的过程描述，在实际操作系统中，信号处理的过程要复杂很多，有兴趣的同学可以查找实际操作系统（如 Linux）在信号处理上的具体实现。

至此，我们基本上完成了“迅猛龙”操作系统，它具有 UNIX 的很多核心特征，比如进程管理、虚存管理、文件系统、管道、I/O 重定向、信号等，是一个典型的宏内核操作系统。虽然它还缺少很多优化的算法、机制和策略，但我们已经一步一步地建立了一个相对完整的操作系统框架和核心模块实现。在这个过程中，我们经历了从简单到复杂的 LibOS、批处理、多道程序、分时多任务、虚存支持、进程支持、文件系统支持等各种操作系统的设计过程，相信同学对操作系统的总体设计也有了一个连贯的多层次的理解。而且我们可以在这个操作系统的框架下，进一步扩展和改进它的设计实现，支持更多的功能并提高性能，这将是我们后续会进一步讲解的内容。

## 8.5.7 参考

- <https://venam.nixers.net/blog/unix/2016/10/21/unix-signals.html>
- <https://www.onitroad.com/jc/linux/man-pages/linux/man2/sigreturn.2.html>
- <http://web.stanford.edu/class/cs110/>

## 8.6 练习

### 8.6.1 课后练习

#### 编程题

1. \* 分别编写基于 UNIX System V IPC 的管道、共享内存、信号量和消息队列的 Linux 应用程序，实现进程间的数据交换。
2. \*\* 分别编写基于 UNIX 的 signal 机制的 Linux 应用程序，实现进程间异步通知。
3. \*\* 参考 rCore Tutorial 中的 shell 应用程序，在 Linux 环境下，编写一个简单的 shell 应用程序，通过管道相关的系统调用，能够支持管道功能。
4. \*\* 扩展内核，实现共享内存机制。
5. \*\*\* 扩展内核，实现 signal 机制。

## 问答题

1. \* 直接通信和间接通信的本质区别是什么？分别举一个例子。
2. \*\* 试说明基于 UNIX 的 signal 机制，如果在本章内核中实现，请描述其大致设计思路和运行过程。
3. \*\* 比较在 Linux 中的无名管道（普通管道）与有名管道（FIFO）的异同。
4. \*\* 请描述 Linux 中的无名管道机制的特征和适用场景。
5. \*\* 请描述 Linux 中的消息队列机制的特征和适用场景。
6. \*\* 请描述 Linux 中的共享内存机制的特征和适用场景。
7. \*\* 请描述 Linux 的 bash shell 中执行与一个程序时，用户敲击 *Ctrl+C* 后，会产生什么信号（signal），导致什么情况出现。
8. \*\* 请描述 Linux 的 bash shell 中执行与一个程序时，用户敲击 *Ctrl+Zombie* 后，会产生什么信号（signal），导致什么情况出现。
9. \*\* 请描述 Linux 的 bash shell 中执行 *kill -9 2022* 这个命令的含义是什么？导致什么情况出现。
10. \*\* 请指出一种跨计算机的主机间的进程间通信机制。

## 8.6.2 实验练习

实验练习包括实践作业和问答作业两部分。

本次难度也就和 lab3 一样吧

### 编程作业

#### 进程通信：邮箱

这一章我们实现了基于 pipe 的进程间通信，但是看测例就知道了，管道不太自由，我们来实现一套乍一看更靠谱的通信 syscall 吧！本节要求实现邮箱机制，以及对应的 syscall。

- 邮箱说明：每个进程拥有唯一一个邮箱，基于“数据报”收发字节信息，利用环形 buffer 存储，读写顺序为 FIFO，不记录来源进程。每次读写单位必须为一个报文，如果用于接收的缓冲区长度不够，舍弃超出的部分（截断报文）。为了简单，邮箱中最多拥有 16 条报文，每条报文最大长度 256 字节。当邮箱满时，发送邮件（也就是写邮箱）会失败。不考虑读写邮箱的权限，也就是所有进程都能够随意给其他进程的邮箱发报。

#### mailread:

- syscall ID: 401
- Rust 接口: fn mailread(buf: \*mut u8, len: usize)
- 功能: 读取一个报文，如果成功返回报文长度.
- 参数:
  - buf: 缓冲区头。
  - len: 缓冲区长度。
- 说明:
  - len > 256 按 256 处理，len < 队首报文长度且不为 0，则截断报文。

– `len = 0`, 则不进行读取, 如果没有报文读取, 返回-1, 否则返回 0, 这是用来测试是否有报文可读。

- 可能的错误:

- 邮箱空。
- `buf` 无效。

**mailwrite:**

- syscall ID: 402

- Rust 接口: `fn mailwrite(pid: usize, buf: *mut u8, len: usize)`

- 功能: 向对应进程邮箱插入一条报文.

- 参数:

- `pid`: 目标进程 id。
- `buf`: 缓冲区头。
- `len`: 缓冲区长度。

- 说明:

- `len > 256` 按 256 处理,
- `len = 0`, 则不进行写入, 如果邮箱满, 返回-1, 否则返回 0, 这是用来测试是否可以发报。
- 可以向自己的邮箱写入报文。

- 可能的错误:

- 邮箱满。
- `buf` 无效。

## 实验要求

- 实现分支: ch6-lab
- 实验目录要求不变
- 通过所有测例

在 `os` 目录下 `make run TEST=1` 加载所有测例, `test_usertest` 打包了所有你需要通过的测例, 你也可以通过修改这个文件调整本地测试的内容。

你的内核必须前向兼容, 能通过前一章的所有测例。

challenge: 支持多核。

## 问答作业

- (1) 举出使用 pipe 的一个实际应用的例子。
- (2) 共享内存的测例中有如下片段 (伪代码):

```

int main()
{
 uint64 *A = (void *)0x10000000;
 uint64 *B = (void *) (0x10000000 + 0x1000);
 uint64 len = 0x1000;
 make_shmem(A, B, len); // 将 [A, A + len) [B, B + len)
 ↵这两段虚存映射到同一段物理内存
 *A = 0xabab;
 __sync_synchronize(); // 这是什么?
 if (*B != 0xabab) {
 return ERROR;
 }
 printf("OK!");
 return 0;
}

```

请查阅相关资料, 回答 `__sync_synchronize` 这行代码的作用, 如果去掉它可能会导致什么错误? 为什么?

## 实验练习的提交报告要求

- 简单总结本次实验与上个实验相比你增加的东西。(控制在 5 行以内, 不要贴代码)
- 完成问答问题
- (optional) 你对本次实验设计及难度的看法。

## 8.7 练习参考答案

### 8.7.1 课后练习

#### 编程题

1. \* 分别编写基于 UNIX System V IPC 的管道、共享内存、信号量和消息队列的 Linux 应用程序, 实现进程间的数据交换。

管道

```

1 #include <unistd.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <string.h>
5
6 int main(void) {
7 int pipefd[2];
8 // pipe syscall creates a pipe with two ends
9 // pipefd[0] is the read end
10 // pipefd[1] is the write end

```

(下页继续)

(续上页)

```

11 // ref: https://man7.org/linux/man-pages/man2/pipe.2.html
12 if (pipe(pipefd) == -1) {
13 perror("failed to create pipe");
14 exit(EXIT_FAILURE);
15 }
16
17 int pid = fork();
18 if (pid == -1) {
19 perror("failed to fork");
20 exit(EXIT_FAILURE);
21 }
22
23 if (pid == 0) {
24 // child process reads from the pipe
25 close(pipefd[1]); // close the write end
26 // read a byte at a time
27 char buf;
28 while (read(pipefd[0], &buf, 1) > 0) {
29 printf("%s", &buf);
30 }
31 close(pipefd[0]); // close the read end
32 } else {
33 // parent process writes to the pipe
34 close(pipefd[0]); // close the read end
35 // parent writes
36 char* msg = "hello from pipe\n";
37 write(pipefd[1], msg, strlen(msg)); // omitting error handling
38 close(pipefd[1]); // close the write end
39 }
40
41 return EXIT_SUCCESS;
42 }
```

## 共享内存

```

1 #include <unistd.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <string.h>
5 #include <sys/shm.h>
6
7 int main(void) {
8 // create a new anonymous shared memory segment of page size, with a
9 // permission of 0600
10 // ref: https://man7.org/linux/man-pages/man2/shmget.2.html
11 int shmid = shmget(IPC_PRIVATE, sysconf(_SC_PAGESIZE), IPC_CREAT | 0600);
12 if (shmid == -1) {
13 perror("failed to create shared memory");
14 exit(EXIT_FAILURE);
15 }
16
17 int pid = fork();
18 if (pid == -1) {
19 perror("failed to fork");
20 exit(EXIT_FAILURE);
21 }
```

(下页继续)

(续上页)

```

20 }
21
22 if (pid == 0) {
23 // attach the shared memory into child process's address space
24 char* shm = shmat(shmid, NULL, 0);
25 while (!shm[0]) {
26 // wait until the parent signals that the data is ready
27 // WARNING: this is not the correct way to synchronize processes
28 // on SMP systems due to memory orders, but this implementation
29 // is chosen here specifically for ease of understanding
30 }
31 printf("%s", shm + 1);
32 } else {
33 // attach the shared memory into parent process's address space
34 char* shm = shmat(shmid, NULL, 0);
35 // copy message into shared memory
36 strcpy(shm + 1, "hello from shared memory\n");
37 // signal that the data is ready
38 shm[0] = 1;
39 }
40
41 return EXIT_SUCCESS;
42 }
```

## 信号量

```

1 #include <unistd.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <string.h>
5 #include <sys/sem.h>
6
7 int main(void) {
8 // create a new anonymous semaphore set, with permission 0600
9 // ref: https://man7.org/linux/man-pages/man2/semget.2.html
10 int semid = semget(IPC_PRIVATE, 1, IPC_CREAT | 0600);
11 if (semid == -1) {
12 perror("failed to create semaphore");
13 exit(EXIT_FAILURE);
14 }
15
16 struct sembuf sops[1];
17 sops[0].sem_num = 0; // operate on semaphore 0
18 sops[0].sem_op = 1; // increase the semaphore's value by 1
19 sops[0].sem_flg = 0;
20 if (semop(semid, sops, 1) == -1) {
21 perror("failed to increase semaphore");
22 exit(EXIT_FAILURE);
23 }
24
25 int pid = fork();
26 if (pid == -1) {
27 perror("failed to fork");
28 exit(EXIT_FAILURE);
29 }
```

(下页继续)

(续上页)

```

31 if (pid == 0) {
32 printf("hello from child, waiting for parent to release semaphore\n");
33 struct sembuf sops[1];
34 sops[0].sem_num = 0; // operate on semaphore 0
35 sops[0].sem_op = 0; // wait for the semaphore to become 0
36 sops[0].sem_flg = 0;
37 if (semop(semid, sops, 1) == -1) {
38 perror("failed to wait on semaphore");
39 exit(EXIT_FAILURE);
40 }
41 printf("hello from semaphore\n");
42 } else {
43 printf("hello from parent, waiting three seconds before release_"
44 "semaphore\n");
45 // sleep for three second
46 sleep(3);
47 struct sembuf sops[1];
48 sops[0].sem_num = 0; // operate on semaphore 0
49 sops[0].sem_op = -1; // decrease the semaphore's value by 1
50 sops[0].sem_flg = 0;
51 if (semop(semid, sops, 1) == -1) {
52 perror("failed to decrease semaphore");
53 exit(EXIT_FAILURE);
54 }
55
56 return EXIT_SUCCESS;
57 }

```

## 消息队列

```

1 #include <unistd.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <string.h>
5 #include <sys/msg.h>
6
7 struct msgbuf {
8 long mtype;
9 char mtext[1];
10 };
11
12 int main(void) {
13 // create a new anonymous message queue, with a permission of 0600
14 // ref: https://man7.org/linux/man-pages/man2/msgget.2.html
15 int msgid = msgget(IPC_PRIVATE, IPC_CREAT | 0600);
16 if (msgid == -1) {
17 perror("failed to create message queue");
18 exit(EXIT_FAILURE);
19 }
20
21 int pid = fork();
22 if (pid == -1) {
23 perror("failed to fork");
24 exit(EXIT_FAILURE);
25 }

```

(下页继续)

(续上页)

```

26
27 if (pid == 0) {
28 // child process receives message
29 struct msgbuf buf;
30 while (msgrcv(msgid, &buf, sizeof(buf.mtext), 1, 0) != -1) {
31 printf("%c", buf.mtext[0]);
32 }
33 } else {
34 // parent process sends message
35 char* msg = "hello from message queue\n";
36 struct msgbuf buf;
37 buf.mtype = 1;
38 for (int i = 0; i < strlen(msg); i++) {
39 buf.mtext[0] = msg[i];
40 msgsnd(msgid, &buf, sizeof(buf.mtext), 0);
41 }
42 struct msqid_ds info;
43 while (msgctl(msgid, IPC_STAT, &info), info.msg_qnum > 0) {
44 // wait for the message queue to be fully consumed
45 }
46 // close message queue
47 msgctl(msgid, IPC_RMID, NULL);
48 }
49
50 return EXIT_SUCCESS;
51 }
```

2. \*\* 分别编写基于 UNIX 的 signal 机制的 Linux 应用程序，实现进程间异步通知。

```

1 #include <unistd.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <signal.h>
5
6 static void sighandler(int sig) {
7 printf("received signal %d, exiting\n", sig);
8 exit(EXIT_SUCCESS);
9 }
10
11 int main(void) {
12 struct sigaction sa;
13 sa.sa_handler = sighandler;
14 sa.sa_flags = 0;
15 sigemptyset(&sa.sa_mask);
16 // register function sighandler as signal handler for SIGUSR1
17 if (sigaction(SIGUSR1, &sa, NULL) != 0) {
18 perror("failed to register signal handler");
19 exit(EXIT_FAILURE);
20 }
21
22 int pid = fork();
23 if (pid == -1) {
24 perror("failed to fork");
25 exit(EXIT_FAILURE);
26 }
27 }
```

(下页继续)

(续上页)

```

28 if (pid == 0) {
29 while (1) {
30 // loop and wait for signal
31 }
32 } else {
33 // send SIGUSR1 to child process
34 kill(pid, SIGUSR1);
35 }
36
37 return EXIT_SUCCESS;
38 }
```

3. \*\*参考 rCore Tutorial 中的 shell 应用程序，在 Linux 环境下，编写一个简单的 shell 应用程序，通过管道相关的系统调用，能够支持管道功能。

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <sys/wait.h>
5 #include <unistd.h>
6
7 int parse(char* line, char** argv) {
8 size_t len;
9 // read a line from stdin
10 if (getline(&line, &len, stdin) == -1)
11 return -1;
12 // remove trailing newline
13 line[strlen(line) - 1] = '\0';
14 // split line into tokens
15 int i = 0;
16 char* token = strtok(line, " ");
17 while (token != NULL) {
18 argv[i] = token;
19 token = strtok(NULL, " ");
20 i++;
21 }
22 return 0;
23 }
24
25 int concat(char** argv1, char** argv2) {
26 // create pipe
27 int pipefd[2];
28 if (pipe(pipefd) == -1)
29 return -1;
30
31 // run the first command
32 int pid1 = fork();
33 if (pid1 == -1)
34 return -1;
35 if (pid1 == 0) {
36 dup2(pipefd[1], STDOUT_FILENO);
37 close(pipefd[0]);
38 close(pipefd[1]);
39 execvp(argv1[0], argv1);
40 }
41 }
```

(下页继续)

(续上页)

```

42 // run the second command
43 int pid2 = fork();
44 if (pid2 == -1)
45 return -1;
46 if (pid2 == 0) {
47 dup2(pipefd[0], STDIN_FILENO);
48 close(pipefd[0]);
49 close(pipefd[1]);
50 execvp(argv2[0], argv2);
51 }
52
53 // wait for them to exit
54 close(pipefd[0]);
55 close(pipefd[1]);
56 wait(&pid1);
57 wait(&pid2);
58 return 0;
59 }
60
61 int main(void) {
62 printf("[command 1]$ ");
63 char* line1 = NULL;
64 char* argv1[16] = {NULL};
65 if (parse(line1, argv1) == -1) {
66 exit(EXIT_FAILURE);
67 }
68 printf("[command 2]$ ");
69 char* line2 = NULL;
70 char* argv2[16] = {NULL};
71 if (parse(line2, argv2) == -1) {
72 exit(EXIT_FAILURE);
73 }
74 concat(argv1, argv2);
75 free(line1);
76 free(line2);
77 }

```

4. \*\* 扩展内核，实现共享内存机制。

略

5. \*\*\* 扩展内核，实现 signal 机制。

略，设计思路可参见问答题 2。

## 问答题

1. \* 直接通信和间接通信的本质区别是什么？分别举一个例子。

本质区别是消息是否经过内核，如共享内存就是直接通信，消息队列则是间接通信。

2. \*\* 试说明基于 UNIX 的 signal 机制，如果在本章内核中实现，请描述其大致设计思路和运行过程。

首先需要添加两个 syscall，其一是注册 signal handler，其二是发送 signal。其次是添加对应的内核数据结构，对于每个进程需要维护两个表，其一是 signal 到 handler 地址的对应，其二是尚未处理的 signal。当进程注册 signal handler 时，将所注册的处理函数的地址填入表一。当进程发送 signal 时，找到目标进程，将 signal 写入表二的队列之中。随后修改从内核态返回用户态的入口点的代码，检查是否有待处理的 signal。若有，检查是否有对应的 signal handler

并跳转到该地址，如无则执行默认操作，如杀死进程。需要注意的是，此时需要记住原本的跳转地址，当进程从 signal handler 返回时将其还原。

3. \*\* 比较在 Linux 中的无名管道（普通管道）与有名管道（FIFO）的异同。

同：两者都是进程间信息单向传递的通路，可以在进程之间传递一个字节流。异：普通管道不存在文件系统上对应的文件，而是仅由读写两端两个 fd 表示，而 FIFO 则是由文件系统上的一个特殊文件表示，进程打开该文件后获得对应的 fd。

4. \*\* 请描述 Linux 中的无名管道机制的特征和适用场景。

无名管道用于创建在进程间传递的一个字节流，适合用于流式传递大量数据，但是进程需要自己处理消息间的分割。

5. \*\* 请描述 Linux 中的消息队列机制的特征和适用场景。

消息队列用于在进程之间发送一个由 type 和 data 两部分组成的短消息，接收消息的进程可以通过 type 过滤自己感兴趣的消息，适用于大量进程之间传递短小而多种类的消息。

6. \*\* 请描述 Linux 中的共享内存机制的特征和适用场景。

共享内存用于创建一个多个进程可以同时访问的内存区域，故而消息的传递无需经过内核的处理，适用在需要较高性能的场景，但是进程之间需要额外的同步机制处理读写的顺序与时机。

7. \*\* 请描述 Linux 的 bash shell 中执行与一个程序时，用户敲击 *Ctrl+C* 后，会产生什么信号（signal），导致什么情况出现。

会产生 SIGINT，如果该程序没有捕获该信号，它将会被杀死，若捕获了，通常会在处理完或是取消当前正在进行的操作后主动退出。

8. \*\* 请描述 Linux 的 bash shell 中执行与一个程序时，用户敲击 *Ctrl+Zombie* 后，会产生什么信号（signal），导致什么情况出现。

会产生 SIGTSTP，该进程将会暂停运行，将控制权重新转回 shell。

9. \*\* 请描述 Linux 的 bash shell 中执行 *kill -9 2022* 这个命令的含义是什么？导致什么情况出现。

向 pid 为 2022 的进程发送 SIGKILL，该信号无法被捕获，该进程将会被强制杀死。

10. \*\* 请指出一种跨计算机的主机间的进程间通信机制。

一个在过去较为常用的例子是 Sun RPC。



## 第八章：并发

### 9.1 引言

#### 9.1.1 本章导读

到本章开始之前，我们已经完成了组成应用程序执行环境的操作系统的三个重要抽象：进程、地址空间和文件，让应用程序开发、运行和存储数据更加方便和灵活。特别是操作系统通过硬件中断机制，支持分时多任务和抢占式调度机制。这样操作系统能强制打断进程的执行，及时处理 I/O 交互操作，从而提高整个系统的执行效率。有了进程以后，可以让操作系统从宏观层面实现多个应用的并发执行，而并发是通过操作系统不断地切换进程来达到的。

对于单核处理器而言，在任意一个时刻只会有一个进程被操作系统调度，从而在处理器上执行。到目前为止的并发，仅仅是进程间的并发，而对于一个进程内部，还没有并发性的体现。而这就是线程（Thread）出现的起因：提高一个进程内的并发性。

#### 注解：Dijkstra 教授团队设计解决并发问题的 THE 操作系统

早期的计算机硬件没有内存隔离保护机制，多个程序以任务（task）的形式进行执行，但各个任务之间是依次执行（批处理方式）或相互独立执行，基本没有数据共享的情况，所以还没有形成线程的概念。当多个任务需要共享数据和同步行为时，就需要扩展任务针对共享数据的执行特征，并建立相应的同步互斥机制。

在 1962 年，荷兰的 E.W.Dijkstra 教授和他的团队正在为 Electrologica X8 计算机设计开发了 **THE 操作系统**。他们观察到如果多个程序在执行中访问共享变量，可能会产生冲突和结果不确定。在 E.W.Dijkstra 教授在信号量机制的研究中，提出了多个“sequential processes”可以通过信号量机制合作访问共享变量，避免冲突导致结果不确定。这里的“sequential processes”的含义就是后续的线程。

#### Dijkstra 教授设计出信号量机制

Dijkstra 教授带领他的小团队在设计开发 THE 操作系统的过程中，异步中断触发的难以重现的并发错误，让他们在调试操作系统中碰到了困难。这种困难激发了 Dijkstra 团队的灵感，他们设计了操作系统的分层结构来避免操作系统的复杂性负担，同时还设计了信号量机制和对应的 P 和 V 操作，来确保线程对共享变量的灵活互斥访问，并支持线程之间的同步操作。P 和 V 是来自荷兰语单词“测试”和“增加”的首字母，是很罕见的非英语来源的操作系统术语。

### 贝尔实验室 Victor A. Vyssotsky 提出线程（thread）概念

1964 年开始设计的 Multics 操作系统已经有进程的概念，也有多处理器并行处理的 GE 645 硬件设计，甚至提出了线程（**thread**）的概念。1966 年，参与 Multics 开发的 MIT 博士生 Jerome Howard Saltzer 在其博士毕业论文的一个注脚提到贝尔实验室的 Victor A. Vyssotsky 用 **thread** 这个名称来表示处理器（processor）执行程序（program）代码序列这个过程的抽象概念，Saltzer 进一步把“进程（process）”描述为处理器执行程序代码的当前状态（即线程）和可访问的地址空间。但他们并没有建立类似信号量这样的有效机制来避免并发带来的同步互斥问题。

### Brinch Hansen、Tony Hoare 和 Dijkstra 提出管程机制

丹麦的 Brinch Hansen，英国的 Tony Hoare 和 Dijkstra 并不满足于信号量来解决操作系统和应用中的并发问题。因为对于复杂一些的同步互斥问题（如哲学家问题），如果使用信号量机制不小心，容易引起死锁等错误。在 1971 年的研讨会上，他们三人开始讨论管程（Monitor）的想法，希望设计一种更高级的并发管理语言结构，便于程序员开发并发程序。在 1972 年春天，Brinch Hansen 在他写的“操作系统原理”教科书中，提出了管程的概念，并把这一概念嵌入到了 Concurrent Pascal 编程语言中，然后他和他的学生再接再厉，在 PDP 11/45 计算机上编写了 Concurrent Pascal 编译器，并用 Concurrent Pascal 编写了 Solo 操作系统。Brinch Hansen 在操作系统和语言级并发处理方面的开创性工作影响了后续的操作系统并发处理机制（如条件变量等）和不少的编程语言并发方案。

Brinch Hansen 的两句名言：

- 写作是对简单性的严格测试：不可能令人信服地写出无法理解的想法。
  - 编程是用清晰的散文写文章并使它们可执行的艺术
- 

### 提示：并行与并发的区别

可回顾一下 [并行与并发的解释](#)。在单处理器情况下，多个进程或线程是并发执行的。

---

有了进程以后，为什么还会出现线程（Thread）呢？提高整个系统的并行/并发执行效率是主要的原因。考虑如下情况，对于很多应用（以单一进程的形式运行）而言，逻辑上由多个可并行执行的任务组成，如果其中一个任务被阻塞，将导致整个进程被阻塞，这意味着不依赖该任务的其他任务也被阻塞，然而它们实际上本不应该受到影响。这就降低了系统的并发执行效率。

举个具体的例子，我们平常用编辑器来编辑文本内容的时候，都会有一个定时自动保存的功能，即把当前文档内容保存到磁盘上。假设磁盘性能导致编辑器自动保存的过程较慢，并影响到整个进程被阻塞，这会影响到用户编辑文档的人机交互体验：即用户只有等到磁盘写入操作完成后，操作系统重新调度该进程运行，用户才可继续编辑文档。

如果我们把一个进程内的多个可并行执行的任务通过一种更细粒度的方式让操作系统进行调度，那么就可以在进程内实现并发执行。在上面的例子中，负责保存文档内容的任务与负责编辑文档的任务可以并发执行，不会出现一个被阻塞的任务导致其它任务都阻塞的情况。这种任务就是一种更细粒度的调度对象，也就是我们这里说的线程。

### 线程定义

简单地说，线程是进程的组成部分，进程可包含 1-n 个线程，属于同一个进程的线程共享进程的资源，比如地址空间，打开的文件等。线程基本上由线程 ID、执行状态、当前指令指针（PC）、寄存器集合和栈组成。线程是可以被操作系统或用户态调度器独立调度（Scheduling）和分派（Dispatch）的基本单位。

在本章之前，进程是程序的基本执行实体，是程序对某数据集合进行操作的一次执行过程，是系统进行资源（处理器，地址空间和文件等）分配和调度的基本单位。在有了线程后，对进程的定义也要调整了，进程是线程的资源容器，线程成为了程序的基本执行实体。

---

### 提示：线程与进程的区别

注：下面的比较是把以线程为调度对象的操作系统作为分析对象：

- 进程间相互独立（即资源隔离），同一进程的各线程间共享进程的资源（即资源共享）；
- 子进程和父进程有不同的地址空间和资源，而多个线程（没有父子关系）则共享同一所属进程的地址空间和资源；
- 每个线程有其自己的执行上下文（线程 ID、程序计数器、寄存器集合和执行栈），而进程的执行上下文包括其管理的所有线程的执行上下文和地址空间（故同一进程下的线程间上下文切换比进程间上下文切换要快）；
- 线程是一个可调度/分派/执行的实体（线程有就绪、阻塞和运行三种基本执行状态），进程不是可调度/分派/执行的的实体，而是线程的资源容器；
- 进程间通信需要通过 IPC 机制（如管道等），属于同一进程的线程间可以共享“即直接读写”进程的数据，但需要同步互斥机制的辅助，避免出现数据不一致性以及不确定计算结果的问题。

## 同步互斥

在上面提到了数据不一致性、不确定的计算结果，意味在操作系统的执行过程中，可能存在并发问题，并导致程序或操作系统执行失败。我们先给出 **线程的数据一致性的** 定义：在单处理器（即只有一个核的 CPU）下，如果某线程更新了一个可被其他线程读到的共享数据，那么后续其他线程都能读到这个最新被更新的共享数据。当多个线程共享同一进程的地址空间时，每个线程都可以访问属于这个进程的数据（全局变量）。如果每个线程使用到的变量都是其他线程不会读取或者修改的话，各个线程访问的变量与预期结果一样，那么就不存在一致性问题。如果变量是只读的，多个线程读取该变量与预期结果一致，也不会有一致性问题。

但是，当某些线程在修改变量，而其他线程在读取这个变量时，由于线程之间的执行顺序不能提前预知（取决于操作系统的调度），导致各个线程对同一变量的读写操作序列不确定，这就会导致不同线程可能会看到与预期结果不一样的值，这就出现了数据不一致性的问题，而且每次执行的结果不确定。我们把这种两个或多个线程在竞争访问同一资源时，执行结果取决于它们的不可预知的执行顺序的情况称为 **线程的竞争条件** (**race condition**)。竞态条件是一种常见的并发问题，可能导致应用程序或操作系统执行失败。

出现线程的数据不一致问题和竞态条件问题的根本原因是 **调度的不可控性**：即读写共享变量的代码片段会随时可能被操作系统调度和切换。先看看如下的伪代码例子：

```

1 //全局共享变量 NUM初始化为 0
2 static mut NUM : usize = 0;
3 ...
4
5 //主进程中的所有线程都会执行如下的核心代码
6 unsafe { NUM = NUM + 1; }
7 ...
8
9
10 //所有线程执行完毕后，主进程显示num的值
11 unsafe {
12 println!("NUM = {:?}", NUM);
13 }
```

如果线程的个数为  $n$ ，那么最后主进程会显示的数应该是多少呢？也许同学觉得应该也是  $n$ ，但现实并不是这样。为了了解事实真相，我们首先必须了解 Rust 编译器对  $num = num + 1;$  这一行源代码生成的汇编代码序列。

```

1 # 假设NUM的地址为 0x1000
2 # unsafe { NUM = NUM + 1; } 对应的汇编代码如下
3 addi x6, x0, 0x1000 # addr 100: 计算NUM的地址
4 # 由于时钟中断可能会发生线程切换
```

(下页继续)

(续上页)

```

5 ld x5, 0(x6) # addr 104: 把NUM的值加载到x5寄存器中
6 # 由于时钟中断可能会发生线程切换
7 addi x5, x5, 1 # addr 108: x5 <- x5 + 1
8 # 由于时钟中断可能会发生线程切换
9 sd x5, 0(x6) # addr 112: 把NUM+1的值写回到NUM地址中

```

在这个例子中，一行 Rust 源代码其实被 Rust 编译器生成了四行 RISC-V 汇编代码。如果多个线程在操作系统的管理和调度下都执行这段代码，那么在上述四行汇编代码之间（即第 4, 6, 8 行的地方）的时刻可能产生时钟中断，并导致线程调度和切换。

设有两个线程，线程 A 先进入上述汇编代码区，将要把 NUM 增加一，为此线程 A 将 NUM 的值（假设它这时是 0）加载到 x5 寄存器中，然后执行加一操作，此时  $x5 = 1$ 。这时时钟中断发生，操作系统将当前正在运行的线程 A 的上下文（它的程序计数器、寄存器，包括 x5 等）保存到线程控制块（在内存中）中。

再接下来，线程 B 被选中运行，并进入同一段代码。它也执行了前两条指令，获取 NUM 的值（此时仍为 0）并将其放入 x5 中，线程 B 继续执行接下来指令，将 x5 加一，然后将 x5 的内容保存到 NUM（地址 0x1000）中。因此，全局变量 NUM 现在的值是 1。

最后又发生一次线程上下文切换，线程 A 恢复运行，此时的  $x5=1$ ，现在线程 A 准备执行最后一条 sd 指令，将 x5 的内容保存到 NUM（地址 0x1000）中，NUM 再次被设置为 1。

简单总结，这两个线程执行的结果是：增加 NUM 的代码被执行两次，初始值为 0，但是结果为 1。而我们一般理解这两个线程执行的“正确”结果应该是全局变量 NUM 等于 2。

### 注解：并发相关术语

- 共享资源 (shared resource)：不同的线程/进程都能访问的变量或数据结构。
- 临界区 (critical section)：访问共享资源的一段代码。
- 竞态条件 (race condition)：多个线程/进程都进入临界区时，都试图更新共享的数据结构，导致产生了不期望的结果。
- 不确定性 (indeterminate)：多个线程/进程在执行过程中出现了竞态条件，导致执行结果取决于哪些线程在何时运行，即执行结果不确定，而开发者期望得到的是确定的结果。
- 原子性 (atomic)：一系列操作要么全部完成，要么一个都没执行，不会看到中间状态。在数据库领域，具有原子性的一系列操作称为事务 (transaction)。
- 互斥 (mutual exclusion)：一种原子性操作，能保证同一时间只有一个线程进入临界区，从而避免出现竞态条件，并产生确定的预期执行结果。
- 同步 (synchronization)：多个并发执行的进程/线程在一些关键点上需要互相等待，这种相互制约的等待称为进程/线程同步。
- 死锁 (dead lock)：一个线程/进程集合里面的每个线程/进程都在等待只能由这个集合中的其他一个线程/进程（包括他自身）才能引发的事件，这种情况就是死锁。
- 饥饿 (hungry)：指一个可运行的线程/进程尽管能继续执行，但由于操作系统的调度而被无限期地忽视，导致不能执行的情况。

在后续的章节中，会大量使用上述术语，如果现在还不够理解，没关系，随着后续的一步一步的分析和实验，相信大家能够掌握上述术语的实际含义。

为了解决数据不一致问题和竞态条件问题，操作系统需要提供一些保障机制（比如互斥、同步等），无论操作系统如何调度（当然需要是正常情况下的调度）这些对共享数据进行读写的线程，各个线程都能得到预期的共享数据的正确访问结果。操作系统中常见的同步互斥机制包括：互斥锁 (Mutex Lock)、信号量 (Semaphore)、条件变量 (Conditional Variable) 等。

## 互斥锁

互斥锁是操作系统中用于保护共享资源的机制。互斥锁能够确保在任何时候只有一个线程访问共享资源，从而避免资源竞争导致的数据不一致的问题。可以使用 Rust 标准库中的 `std::sync::Mutex` 类型来实现互斥锁。下面是一个使用互斥锁保护共享变量的示例：

```

1 use std::sync::Arc, Mutex;
2 use std::thread;
3
4 fn main() {
5 // 创建一个可变的整数并将其包装在 Mutex 中
6 let data = Arc::new(Mutex::new(0));
7
8 // 创建两个线程，并传递 `data` 的 Arc 实例给它们
9 let data_clone = data.clone();
10 let handle1 = thread::spawn(move || {
11 let mut data = data_clone.lock().unwrap();
12 *data += 1;
13 });
14
15 let data_clone = data.clone();
16 let handle2 = thread::spawn(move || {
17 let mut data = data_clone.lock().unwrap();
18 *data += 1;
19 });
20
21 // 等待两个线程结束
22 handle1.join().unwrap();
23 handle2.join().unwrap();
24
25 // 输出结果
26 println!("Result: {}", *data.lock().unwrap());
27 }
```

在上面的代码中，两个线程都会尝试访问 `data` 变量，但是因为它被包装在了 `Mutex` 中，所以只有一个线程能够获取锁并访问变量。在获取互斥锁的时候，线程会被挂起，直到另一个线程释放了锁。最终的输出结果是 2。

## 条件变量

条件变量是操作系统中的一种同步原语，可用于在多个线程之间进行协作，即允许一个线程在另一个线程完成某些操作之前等待。条件变量与互斥锁经常一起使用，以保证在同一时刻只有一个线程在访问共享资源。

在 Rust 中，条件变量是由 `std::sync::Condvar` 结构体表示的。条件变量需要配合互斥体（由 `std::sync::Mutex` 结构体表示）使用，因为条件变量用于在互斥体保护的条件下通知等待的线程。

```

1 fn main() {
2 use std::sync::Arc, Condvar, Mutex;
3 use std::thread;
4
5 let pair = Arc::new((Mutex::new(false), Condvar::new()));
6 let pair2 = Arc::clone(&pair);
7
8 // Inside of our lock, spawn a new thread, and then wait for it to start.
9 thread::spawn(move || {
10 let (lock, cvar) = &*pair2;
```

(下页继续)

(续上页)

```

11 let mut started = lock.lock().unwrap();
12 *started = true;
13 // We notify the condvar that the value has changed.
14 cvar.notify_one();
15);
16
17 // Wait for the thread to start up.
18 let (lock, cvar) = &*pair;
19 let mut started = lock.lock().unwrap();
20 while !*started {
21 started = cvar.wait(started).unwrap();
22 }
23 }
```

这是一个使用 Rust 中的条件变量 (Condvar) 和互斥锁 (Mutex) 来同步两个线程进行协作的示例。在这个示例中，新线程通过更改布尔值并通知条件变量来发送信号，而主线程则使用条件变量来等待信号。首先，它定义了一个元组 (Mutex<bool>, Condvar)，并使用 Arc (原子引用计数) 将其包装在一个可共享的指针中。这个指针有两个副本，因此两个线程都可以访问这个元组。然后，它启动了一个新的线程，并在这个线程内部使用互斥锁来更改共享的布尔值。最后，它使用条件变量来等待这个布尔值被更改，然后退出循环。

## 信号量

信号量是操作系统中的一种同步原语，用于在多个线程或进程之间共享资源时进行互斥访问。它通常是一个整数值，用于计数指定数量的资源可用。当一个线程需要使用资源时，它会执行信号量的 *acquire* 操作，如果信号量的值小于等于零，则线程将被挂起，(直到信号量的值变为正数，则会被唤醒)；否则将信号量的值减一，操作正常返回。另一方面，当一个线程完成使用资源后，它可以执行信号量的 *release* 操作，将信号量的值加一，并唤醒一个或所有挂起的线程。Rust 标准库中没有信号量类型，但我们可以用 Mutex 和 Condvar 来构造信号量类型。

```

1 use std::sync::{Condvar, Mutex};
2
3 pub struct Semaphore {
4 condvar: Condvar,
5 counter: Mutex<isize>,
6 }
7
8 impl Semaphore {
9 pub fn new(var: isize) -> Semaphore {
10 Semaphore {
11 condvar: Condvar::new(),
12 counter: Mutex::new(var),
13 }
14 }
15 pub fn acquire(&self) {
16 // gain access to the atomic integer
17 let mut count = self.counter.lock().unwrap();
18
19 // wait so long as the value of the integer <= 0
20 while *count <= 0 {
21 count = self.condvar.wait(count).unwrap();
22 }
23
24 // decrement our count to indicate that we acquired
25 // one of the resources
26 }
27 }
```

(下页继续)

(续上页)

```

26 *count -= 1;
27 }
28 pub fn release(&self) {
29 // gain access to the atomic integer
30 let mut count = self.counter.lock().unwrap();
31
32 // increment its value
33 *count += 1;
34
35 // notify one of the waiting threads
36 self.condvar.notify_one();
37 }
38 }
```

我们构造的 *Semaphore* 类型包含了三个方法：

- *new(var)* 方法创建一个信号量，并初始化信号量值 *counter* 的为 ‘var’；
- *acquire()* 方法将信号量值减一，如果信号量的值已经为零，则线程通过条件变量 *condvar* 的 *wait* 操作将自己挂起；
- *release()* 方法将信号量值加一，并通过条件变量 *condvar* 的 *notify\_one* 操作唤醒一个挂起线程。

有了信号量，我们就可以建立使用信号量的示例程序，该程序创建了三个线程，每个线程都会调用 *acquire* 方法获取信号量，然后输出一条消息，最后在信号量上调用 *release* 方法释放信号量。

```

1 use std::sync::Arc;
2 use std::thread;
3 fn main() {
4 //let sem = Semaphore::new(1);
5 // 创建信号量，并设置允许同时访问的线程数为 2。
6 let semaphore = Arc::new(Semaphore::new(2));
7
8 // 创建三个线程。
9 let threads = (0..3)
10 .map(|i| {
11 let semaphore = semaphore.clone();
12 thread::spawn(move || {
13 // 在信号量上调用 acquire 方法获取信号量。
14 semaphore.acquire();
15
16 // 输出消息。
17 println!("Thread {}: acquired semaphore", i);
18
19 // 模拟执行耗时操作。
20 thread::sleep(std::time::Duration::from_secs(1));
21
22 // 在信号量上调用 release 方法释放信号量。
23 println!("Thread {}: releasing semaphore", i);
24 semaphore.release();
25 })
26 })
27 .collect::<Vec<_>>();
28
29 // 等待所有线程完成。
30 for thread in threads {
31 thread.join().unwrap();
32 }
}
```

(下页继续)

(续上页)

33 }

这段代码创建了一个名为 *semaphore* 的信号量，并设置允许并发操作的线程数为 2。然后创建了三个线程，在每个线程中，首先调用信号量的 *acquire* 方法来尝试获取信号量。如果获取了信号量，就可以输出一条消息，并模拟执行一些耗时操作，最后调用信号量的 *release* 方法来释放信号量，从而让其他线程有机会获取信号量并继续执行。该示例运行的结果如下所示：

```
Thread 0: acquired semaphore
Thread 1: acquired semaphore
Thread 0: releasing semaphore
Thread 1: releasing semaphore
Thread 2: acquired semaphore
Thread 2: releasing semaphore
```

上述的示例都是在用户态实现的应用程序，其中的 Thread、Mutex 和 Condvar 需要应用程序所在的操作系统（这里就是 Linux）提供相应的支持。在本章中，我们会在自己写的操作系统中实现 Thread、Mutex、Condvar 和 Semaphore 机制，从而对同步互斥的原理有更加深入的了解，对应操作系统如何支持这些同步互斥底层机制有全面的掌握。

## 9.1.2 实践体验

获取本章代码：

```
$ git clone https://github.com/rcore-os/rCore-Tutorial-v3.git
$ cd rCore-Tutorial-v3
$ git checkout ch8
```

在 qemu 模拟器上运行本章代码：

```
$ cd os
$ make run
```

内核初始化完成之后就会进入 shell 程序，我们可以体会一下线程的创建和执行过程。在这里我们运行一下本章的测例 threads：

```
>> threads
aaa....bbb....ccc...
thread#1 exited with code 1
thread#2 exited with code 2
thread#3 exited with code 3
main thread exited.
Shell: Process 2 exited with code 0
>>
```

它会有 4 个线程在执行，等前 3 个线程执行完毕后，主线程退出，导致整个进程退出。

此外，在本章的操作系统支持通过互斥来执行“哲学家就餐问题”这个应用程序：

```
>> phil_din_mutex
time cost = 7273
'-' -> THINKING; 'x' -> EATING; ' ' -> WAITING
#0: ----- XXXXXXXX----- XXXX----- XXXXXX--XXX
#1: --XXXXXX-- XXXXXX----- X---XXXXXX
#2: ----- XX-----XX---XXXXXX----- XXXXX
#3: -----XXXXXXXXX-----XXXXX----- XXXXXX-- XXXXXXXXXX
```

(下页继续)

(续上页)

```
#4: ----- X----- XXXXXX-- XXXX----- XX
#0: ----- XXXXXXXX----- XXXX----- XXXX---XXX
>>
```

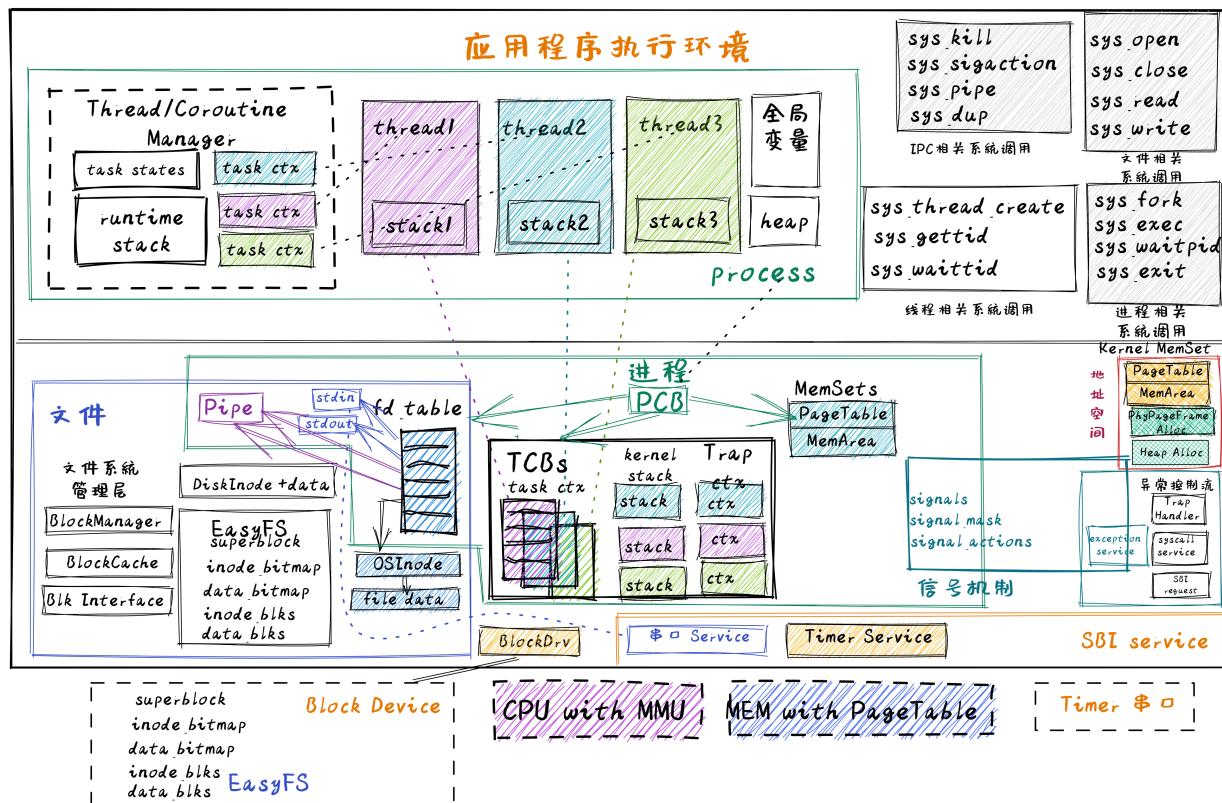
我们可以看到 5 个代表“哲学家”的线程通过操作系统的 **Mutex** 互斥机制在进行“THINKING”、“EATING”、“WAITING”的日常生活。没有哲学家由于拿不到筷子而饥饿，也没有相邻的两个哲学家同时拿到同一个筷子。

### 注解：哲学家就餐问题

计算机科学家 Dijkstra 提出并解决的哲学家就餐问题是经典的进程同步互斥问题。哲学家就餐问题描述如下：有 5 个哲学家共用一张圆桌，分别坐在周围的 5 张椅子上，在圆桌上有 5 个碗和 5 只筷子，他们的生活方式是交替地进行思考和进餐。平时，每个哲学家进行思考，饥饿时便试图拿起其左右最靠近他的筷子，只有在他拿到两只筷子时才能进餐。进餐完毕，放下筷子继续思考。

### 9.1.3 本章代码树

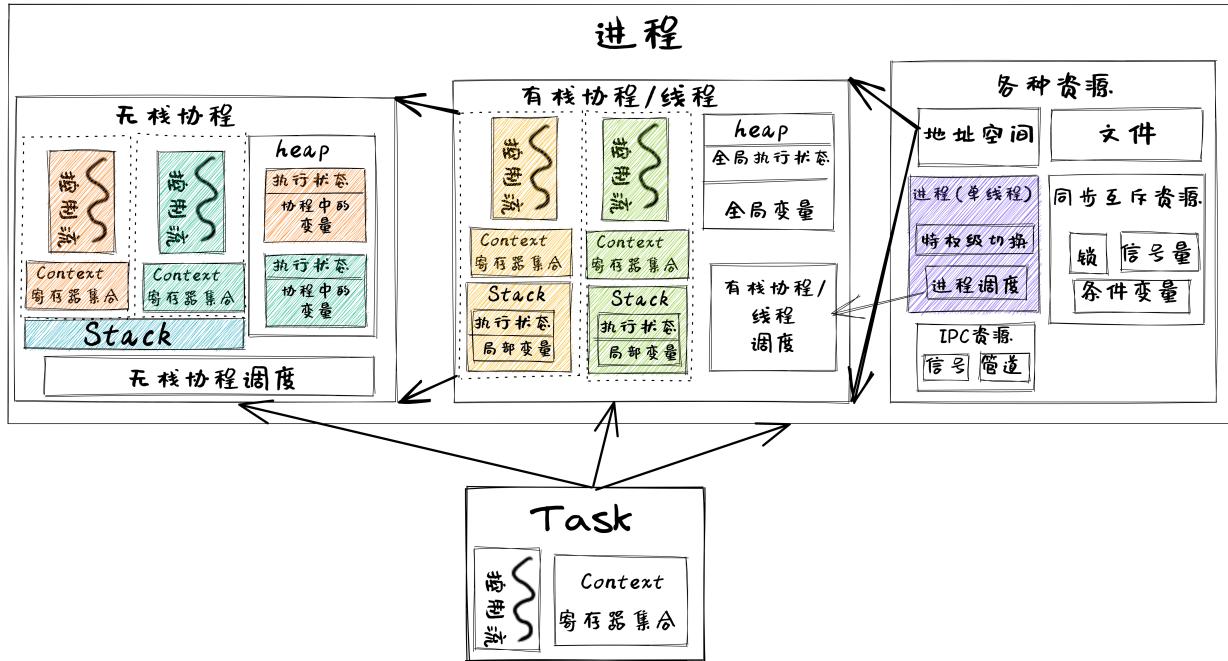
达科塔盗龙操作系统-Thread&Coroutine OS 的总体结构如下图所示：



从上图中可以看到，Thread&Coroutine OS 增加了在用户态管理的用户态线程/用户态协程，以及在内核态管理的用户态线程。对于用户态管理的用户态线程和协程，新增了一个运行在用户态的 *Thread/Coroutine Manager* 运行时库 (Runtime Lib)，这个不需要改动操作系统内核。而对于内核态管理的用户态线程，则需要新增线程控制块 (Thread Control Block, TCB) 结构，把之前进程控制块 (Process Control Block, PCB) 中与执行相关的内容剥离给了线程控制块。同时，进一步重构进程控制块，把线程控制块列表作为进程控制块中的一部分。

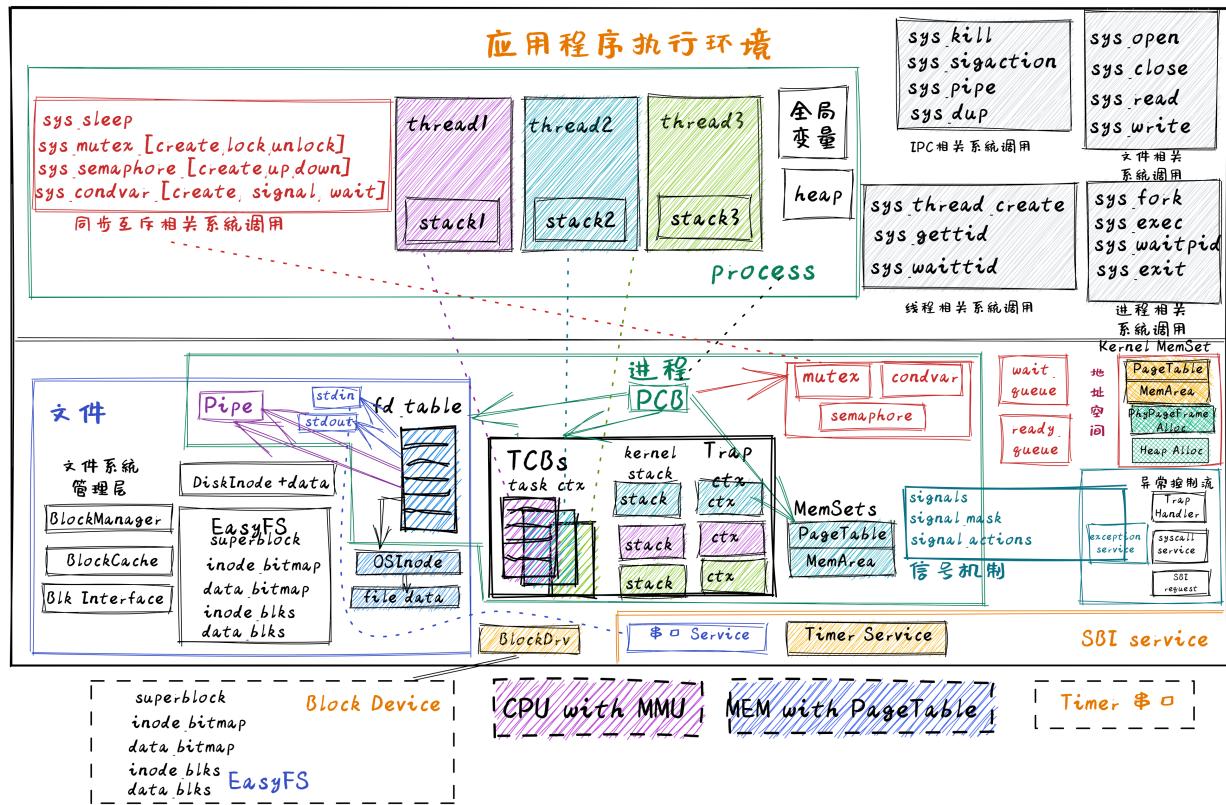
资源，这样一个进程控制块就可以管理多个线程了。最后还提供与线程相关的系统调用，如创建线程、等待线程结束等，以支持多线程应用的执行。

这里，我们可以把进程、线程和协程中的控制流执行看出是一种任务（Task）的执行过程，如下图所示：



在上图中，可以看出进程包含线程（即有栈协程），线程包含无栈协程，形成一个层次包含关系。而与它们执行相关的重点是切换控制流，即任务切换，关键就是保存于恢复任务上下文，任务上下文的核心部分就是每个任务所分时共享的硬件寄存器内容。对于无栈协程，切换这些寄存器就够了；对于拥有独立栈的线程而言，还需进一步切换线程栈；如果是拥有独立地址空间的进程而言，那还需进一步切换地址空间（即切换页表）。

进一步增加了同步互斥机制的慈母龙操作系统-SyncMutexOS 的总体结构如下图所示：



在上图中，可以看出在进程控制块中，增加了互斥锁（Mutex）、信号量（Semaphore）和条件变量（Condvar）这三种资源，并提供了与这三种同步互斥资源相关的系统调用。这样多线程应用就可以使用这三种同步互斥机制来解决各种同步互斥问题，如生产者消费者问题、哲学家问题、读者写者问题等。

位于 ch8 分支上的慈母龙操作系统-SyncMutexOS 的源代码如下所示：

```

1 ...
2 ...
3 os
4 ...
5 src
6 ...
7 sync (新增：同步互斥子模块 sync)
8 | mod.rs
9 | condvar.rs (条件变量实现)
10 | mutex.rs (互斥锁实现)
11 | semaphore.rs (信号量实现)
12 syscall
13 ...
14 | mod.rs (增加与线程/同步互斥相关的系统调用定义)
15 | sync.rs (增加与同步互斥相关的系统调用具体实现)
16 | thread.rs (增加与线程相关的系统调用具体实现)
17 task (重构进程管理子模块，以支持线程)
18 ...
19 | process.rs (包含线程控制块的进程控制块)
20 | task.rs (线程控制块)
21 | timer.rs (增加支持线程睡眠一段时间的功能)
22 trap
23 | context.rs

```

(下页继续)

(续上页)

```

24 ┌── mod.rs
25 └── trap.S
26
27 └── user
28 └── ...
29 └── src
30 └── bin (新增各种多线程/协程/同步互斥测试用例)
31 └── ...
32 └── early_exit2.rs (多线程测例)
33 └── early_exit.rs (多线程测例)
34 └── eisenberg.rs (面向n个线程的Eisenberg&McGuire)
35
36 ↳ 软件同步互斥示例)
37 └── mpsc_sem.rs (基于信号量的生产者消费者问题示例)
38 └── peterson.rs (面向2个线程的Peterson软件同步互斥示例)
39 └── phil_din_mutex.rs (基于互斥锁的哲学家就餐问题示例)
40 └── race_adder_arg.rs (具有竞态条件错误情况的多线程累加计算示例)
41 └── race_adder_atomic.rs (基于原子变量的多线程累加计算示例)
42 └── race_adder_loop.
43
44 ↳ rs (具有竞态条件错误情况的多线程累加计算示例)
45 └── race_adder_mutex_blocking.
46
47 ↳ rss (基于可睡眠互斥锁的多线程累加计算示例)
48 └── race_adder_mutex_spin.
49
50 ↳ rs (基于忙等互斥锁的多线程累加计算示例)
51 └── race_adder.rs (具有竞态条件错误情况的多线程累加计算示例)
52
53 ↳ rs (用户态多线程 (有栈协程) 管理运行时库和多线程示例)
54 └── stackful_coroutine.
55
56 ↳ rs (用户态无栈协程管理运行时库和多协程示例)
57 └── sync_sem.rs (基于信号量的多线程同步示例)
58 └── test_condvar.rs (基于条件变量和互斥锁的多线程同步示例)
59 └── threads_arg.rs (带参数的多线程示例)
60 └── threads.rs (无参数的多线程示例)
61 └── usertests.rs (运行所有应用的示例)
62
63 ...

```

## 9.1.4 本章代码导读

在本章实现支持多线程的达科塔盗龙操作系统—Thread&Coroutine OS 过程中，需要考虑如下一些关键点：线程的总体结构、管理线程执行的线程控制块数据结构、以及对线程管理相关的重要函数：线程创建和线程切换。这些关键点既可以在用户态实现，也可在内核态实现。

### 线程设计与实现

在用户态的线程管理一节中讲述了设计实现用户态线程管理运行时库的过程，这其实是第三章中任务切换的设计与实现 和 协作式调度 的一种更简单的用户态实现。首先是要构建多线程的基本执行环境，即定义线程控制块数据结构，包括线程执行状态、线程执行上下文（使用的通用寄存器集合）等。然后是要实现线程创建和线程切换这两个关键函数。这两个函数的关键就是构建线程的上下文和切换线程的上下文。当线程启动后，不会被抢占，所以需要线程通过 `yield_task` 函数主动放弃处理器，从而把处理器控制权交还给用户态线程管理运行时库，让其选择其他处于就绪态的线程执行。

在内核态的线程管理一节中讲述了在操作系统内部设计实现内核态线程管理的实现过程，这其实基于第三章中任务切换的设计与实现 和 抢占式调度 的进一步改进实现。这涉及到对进程的重构，把以前的线程管理相关数据结构转移到线程控制块中，并把线程作为一种资源，放在进程控制块中。这样与线程相关的关键部分包括：

- 任务控制块 TaskControlBlock：表示线程的核心数据结构
- 任务管理器 TaskManager：管理线程集合的核心数据结构
- 处理器管理结构 Processor：用于线程调度，维护线程的处理器状态
- 线程切换：涉及特权级切换和线程上下文切换

进程控制块和线程控制块的主要部分如下所示：

```

1 // os/src/task/tasks.rs
2 // 线程控制块
3 pub struct TaskControlBlock {
4 pub process: Weak<ProcessControlBlock>, //线程所属的进程控制块
5 pub kstack: KernelStack, //任务（线程）的内核栈
6 inner: UPSafeCell<TaskControlBlockInner>,
7 }
8 pub struct TaskControlBlockInner {
9 pub res: Option<TaskUserRes>, //任务（线程）用户态资源
10 pub trap_cx_ppn: PhysPageNum, //trap上下文地址
11 pub task_cx: TaskContext, //任务（线程）上下文
12 pub task_status: TaskStatus, //任务（线程）状态
13 pub exit_code: Option<i32>, //任务（线程）退出码
14 }
15 // os/src/task/process.rs
16 // 进程控制块
17 pub struct ProcessControlBlock {
18 pub pid: PidHandle, //进程ID
19 inner: UPSafeCell<ProcessControlBlockInner>,
20 }
21 pub struct ProcessControlBlockInner {
22 pub tasks: Vec<Option<Arc<TaskControlBlock>>>, //线程控制块列表
23 ...
24 }
```

接下来就是相关的线程管理功能的设计与实现了。首先是线程创建，即当一个进程执行中发出系统调用 `sys_thread_create` 后，操作系统就需要在当前进程控制块中创建一个线程控制块，并在线程控制块中初始化各个成员变量，建立好进程和线程的关系等，关键要素包括：

- 线程的用户态栈：确保在用户态的线程能正常执行函数调用
- 线程的内核态栈：确保线程陷入内核后能正常执行函数调用
- 线程的跳板页：确保线程能正确的进行用户态  $\leftrightarrow$  内核态切换
- 线程上下文：即线程用到的寄存器信息，用于线程切换

创建线程的主要代码如下所示：

```

1 pub fn sys_thread_create(entry: usize, arg: usize) -> isize {
2 // 创建新线程
3 let new_task = Arc::new(TaskControlBlock::new(...));
4 // 把线程加到就绪调度队列中
5 add_task(Arc::clone(&new_task));
6 // 把线程控制块加入到进程控制块中
7 let tasks = &mut process_inner.tasks;
8 tasks[new_task_tid] = Some(Arc::clone(&new_task));
9 //建立trap/task上下文
10 *new_task_trap_cx = TrapContext::app_init_context(
11 entry,
12 new_task_res.ustack_top(),
```

(下页继续)

(续上页)

```

13 kernel_token(),
14 ...

```

而关于线程切换和线程调度这两部分在之前已经介绍过。线程切换与第三章中介绍的特权级上下文切换和任务上下文切换的设计与实现是一致的，线程执行中的调度切换过程与第六章中介绍的进程调度机制是一致的。这里就不再进一步赘述了。

## 同步互斥机制的设计实现

在实现支持同步互斥机制的慈母龙操作系统-SyncMutexOS 中，包括三种同步互斥机制，在互斥锁一节中讲述了互斥锁的设计与实现，在信号量机制一节中讲述了信号量的设计与实现，在条件变量机制一节中讲述了条件变量的设计与实现。无论哪种同步互斥机制，都需要确保操作系统任意抢占线程，调度和切换线程的执行，都可以保证线程执行的互斥需求和同步需求，从而能够得到可预测和可重现的共享资源访问结果。这三种用于多线程的同步互斥机制所对应的内核数据结构都在进程控制块中，以进程资源的形式存在。

```

1 // 进程控制块内部结构
2 pub struct ProcessControlBlockInner {
3 ...
4 pub mutex_list: Vec<Option<Arc<dyn Mutex>>>, // 互斥锁列表
5 pub semaphore_list: Vec<Option<Arc<Semaphore>>>, // 信号量列表
6 pub condvar_list: Vec<Option<Arc<Condvar>>>, // 条件变量列表
7 }

```

在互斥锁的设计实现中，设计了一个更底层的 *UPSafeCell<T>* 类型，用于支持在单核处理器上安全地在线程间共享可变全局变量。这个类型大致结构如下所示：

```

1 pub struct UPSafeCell<T> { // 允许在单核上安全使用可变全局变量
2 inner: RefCell<T>, // 提供内部可变性和运行时借用检查
3 }
4 unsafe impl<T> Sync for UPSafeCell<T> {} // 声明支持全局变量安全地在线程间共享
5 impl<T> UPSafeCell<T> {
6 pub unsafe fn new(value: T) -> Self {
7 Self { inner: RefCell::new(value) }
8 }
9 pub fn exclusive_access(&self) -> RefMut<'_, T> {
10 self.inner.borrow_mut() // 得到它包裹的数据的独占访问权
11 }
12 }

```

并基于此设计了 *Mutex* 互斥锁类型，可进一步细化为忙等型互斥锁和睡眠型互斥锁，二者的大致结构如下所示：

```

1 pub struct MutexSpin {
2 locked: UPSafeCell<bool>, // locked 是被 UPSafeCell 包裹的布尔全局变量
3 }
4 pub struct MutexBlocking {
5 inner: UPSafeCell<MutexBlockingInner>,
6 }
7 pub struct MutexBlockingInner {
8 locked: bool,
9 wait_queue: VecDeque<Arc<TaskControlBlock>>, // 等待获取锁的线程等待队列
10 }

```

在上述代码片段的第 9 行，可以看到挂在睡眠型互斥锁上的线程，会被放入到互斥锁的等待队列 *wait\_queue* 中。*Mutex* 互斥锁类型实现了 *lock* 和 *unlock* 两个方法完成获取锁和释放锁操作。而系统调用 *sys\_mutex\_create*

、`sys_mutex_lock`、`sys_mutex_unlock`这几个系统调用，是提供给多线程应用程序实现互斥锁的创建、获取锁和释放锁的同步互斥操作。

信号量 `Semaphore` 类型的大致结构如下所示：

```

1 pub struct Semaphore {
2 pub inner: UPSafeCell<SemaphoreInner>, //UPSafeCell包裹的内部可变结构
3 }
4
5 pub struct SemaphoreInner {
6 pub count: isize, //信号量的计数值
7 pub wait_queue: VecDeque<Arc<TaskControlBlock>>, //信号量的等待队列
8 }

```

在上述代码片段的第 7 行，可以看到挂在信号量上的线程，会被放入到信号量的等待队列 `wait_queue` 中。信号量 `Semaphore` 类型实现了 `up` 和 `down` 两个方法完成获取获取信号量和释放信号量的操作。而系统调用 `sys_semaphore_create`、`sys_semaphore_up`、`sys_semaphore_down` 这几个系统调用，是提供给多线程应用程序实现信号量的创建、获取和释放的同步互斥操作。

条件变量 `Condvar` 类型的大致结构如下所示：

```

1 pub struct Condvar {
2 pub inner: UPSafeCell<CondvarInner>, //UPSafeCell包裹的内部可变结构
3 }
4
5 pub struct CondvarInner {
6 pub wait_queue: VecDeque<Arc<TaskControlBlock>>, //等待队列
7 }

```

在上述代码片段的第 6 行，可以看到挂在条件变量上的线程，会被放入到条件变量的等待队列 `wait_queue` 中。条件变量 `Condvar` 类型实现了 `wait` 和 `signal` 两个方法完成获取等待条件变量和通知信号量的操作。而系统调用 `sys_condvar_create`、`sys_condvar_wait`、`sys_condvar_signal` 这几个系统调用，是提供给多线程应用程序实现条件变量的创建、等待和通知的同步互斥操作。

同学可能会注意到，上述的睡眠型互斥锁、信号量和条件变量的数据结构几乎相同，都会把挂起的线程放到等待队列中。但是它们的具体实现还是有区别的，这需要同学了解这三种同步互斥机制的操作原理，再看看它们的方法对的设计与实现：互斥锁的 `lock` 和 `unlock`、信号量的 `up` 和 `down`、条件变量的 `wait` 和 `signal`，就可以看到它们的具体区别了。

## 9.2 用户态的线程管理

### 9.2.1 本节导读

在本章的起始介绍中，给出了线程的基本定义，但没有具体的实现，这可能让同学在理解线程上还有些不够深入。其实实现多线程不一定需要操作系统的支持，完全可以在用户态实现。本节的主要目标是理解线程的基本要素、多线程应用的执行方式以及如何在用户态构建一个多线程的基本执行环境(即线程管理运行时，Thread Manager Runtime)。

在这里，我们首先分析了一个简单的用户态多线程应用的执行过程，然后设计支持这种简单多线程应用的执行环境，包括线程的总体结构、管理线程执行的线程控制块数据结构、以及对线程管理相关的重要函数：线程创建和线程切换。安排本节的原因在于：它能帮助我们直接理解线程最核心的设计思想与具体实现，并对后续在有进程支持的操作系统内核中进一步实现线程机制打下一个基础。

## 9.2.2 用户态多线程应用

我们先看看一个简单的用户态多线程应用。

```

1 // 多线程基本执行环境的代码
2 ...
3 // 多线程应用的主体代码
4 fn main() {
5 let mut runtime = Runtime::new();
6 runtime.init();
7 runtime.spawn(|| {
8 println!("TASK 1 STARTING");
9 let id = 1;
10 for i in 0..10 {
11 println!("task: {} counter: {}", id, i);
12 yield_task();
13 }
14 println!("TASK 1 FINISHED");
15 });
16 runtime.spawn(|| {
17 println!("TASK 2 STARTING");
18 let id = 2;
19 for i in 0..15 {
20 println!("task: {} counter: {}", id, i);
21 yield_task();
22 }
23 println!("TASK 2 FINISHED");
24 });
25 runtime.run();
26 }
```

可以看出，多线程应用的结构很简单，大致含义如下：

- 第 5~6 行首先是多线程执行环境的创建和初始化，具体细节在后续小节会进一步展开讲解。
- 第 7~15 行创建了第一个线程；第 16~24 行创建了第二个线程。这两个线程都是用闭包的形式创建的。
- 第 25 行开始执行这两个线程。

这里面需要注意的是第 12 行和第 21 行的 `yield_task()` 函数。这个函数与我们在第二章讲的 `sys_yield` 系统调用在功能上是一样的，即当前线程主动交出 CPU 并切换到其它线程执行。

假定同学在一个 linux for RISC-V 64 的开发环境中，我们可以执行上述的程序：

注：可参看指导，建立 linux for RISC-V 64 的开发环境

```

$ git clone -b rv64 https://github.com/chyyuu/example-greenthreads.git
$ cd example-greenthreads
$ cargo run
...
 TASK 1 STARTING
 task: 1 counter: 0
 TASK 2 STARTING
 task: 2 counter: 0
 task: 1 counter: 1
 task: 2 counter: 1
 ...
 task: 1 counter: 9
 task: 2 counter: 9
 TASK 1 FINISHED
```

(下页继续)

(续上页)

```

...
task: 2 counter: 14
TASK 2 FINISHED

```

可以看到，在一个进程内的两个线程交替执行。这是如何实现的呢？

### 9.2.3 多线程的基本执行环境

线程的运行需要一个执行环境，这个执行环境可以是操作系统内核，也可以是更简单的用户态的一个线程管理运行时库。如果是基于用户态的线程管理运行时库来实现对线程的支持，那我们需要对线程的管理、调度和执行方式进行一些限定。由于是在用户态进行线程的创建，调度切换等，这就意味着我们不需要操作系统提供进一步的支持，即操作系统不需要感知到这种线程的存在。如果一个线程 A 想要运行，它只有等到目前正在运行的线程 B 主动交出处理器的使用权，从而让线程管理运行时库有机会得到处理器的使用权，且线程管理运行时库通过调度，选择了线程 A，再完成线程 B 和线程 A 的线程上下文切换后，线程 A 才能占用处理器并运行。这其实就是第三章讲到的[任务切换的设计与实现](#)和[协作式调度](#)的另外一种更简单的具体实现。

#### 线程的结构与执行状态

为了实现用户态的协作式线程管理，我们首先需要考虑这样的线程大致的结构应该是什么？在上一节的[线程的基本定义](#)中，已经给出了具体的答案：

- 线程 ID
- 执行状态
- 当前指令指针 (PC)
- 通用寄存器集合
- 栈

基于这个定义，就可以实现线程的结构了。把上述内容集中在一起管理，形成线程控制块：

```

1 //线程控制块
2 struct Task {
3 id: usize, // 线程 ID
4 stack: Vec<u8>, // 栈
5 ctx: TaskContext, // 当前指令指针 (PC) 和通用寄存器集合
6 state: State, // 执行状态
7 }
8
9 struct TaskContext {
10 // 15 u64
11 x1: u64, //ra: return address, 即目前正在执行线程的当前指令指针 (PC)
12 x2: u64, //sp
13 x8: u64, //s0, fp
14 x9: u64, //s1
15 x18: u64, //x18-27: s2-11
16 x19: u64,
17 ...
18 x27: u64,
19 nx1: u64, //new return address, 即下一个要执行线程的当前指令指针 (PC)
20 }

```

线程在执行过程中的状态与之前描述的进程执行状态类似，表明线程在执行过程中的动态执行特征：

```

1 enum State {
2 Available, // 初始态: 线程空闲, 可被分配一个任务去执行
3 Running, // 运行态: 线程正在执行
4 Ready, // 就绪态: 线程已准备好, 可恢复执行
5 }

```

下面的线程管理初始化过程中, 会创建一个线程控制块向量, 其中的每个线程控制块对应到一个已创建的线程 (其状态为 *Running* 或 *Ready* ) 或还没加入一个具体的线程 (此时其状态为 *Available* )。当创建线程并分配一个空闲的线程控制块给这个线程时, 管理此线程的线程控制块的状态将转为 *Ready* 状态。当线程管理运行时调度切换此线程占用处理器执行时, 会把此线程的线程控制块的状态设置为 *Running* 状态。

## 线程管理运行时初始化

线程管理运行时负责整个应用中的线程管理。当然, 它也需要完成自身的初始化工作。这里主要包括两个函数:

- *Runtime::new()* 主要有三个步骤:
  - 初始化应用主线程控制块 (其 TID 为 0 ), 并设置其状态为 *Running* 状态;
  - 初始化 *tasks* 线程控制块向量, 加入应用主线程控制块和空闲线程控制块, 为后续的线程创建做好准备;
  - 包含 *tasks* 线程控制块向量和 *current* 当前线程 id (初始值为 0, 表示当前正在运行的线程是应用主线程), 来建立 *Runtime* 变量;
- *Runtime::init()* , 把线程管理运行时的 *Runtime* 自身的地址指针赋值给全局可变变量 *RUNTIME*

```

1 impl Task {
2 fn new(id: usize) -> Self {
3 Task {
4 id,
5 stack: vec![0_u8; DEFAULT_STACK_SIZE],
6 ctx: TaskContext::default(),
7 state: State::Available,
8 }
9 }
10 }
11 impl Runtime {
12 pub fn new() -> Self {
13 // This will be our base task, which will be initialized in the `running` ↴
14 state
15 let base_task = Task {
16 id: 0,
17 stack: vec![0_u8; DEFAULT_STACK_SIZE],
18 ctx: TaskContext::default(),
19 state: State::Running,
20 };
21
22 // We initialize the rest of our tasks.
23 let mut tasks = vec![base_task];
24 let mut available_tasks: Vec<Task> = (1..MAX_TASKS).map(|i| Task::new(i)). ↴
25 collect();
26 tasks.append(&mut available_tasks);
27
28 Runtime {
29 tasks,
30 current: 0,
31 }
32 }
33 }

```

(下页继续)

(续上页)

```

29 }
30 }
31
32 pub fn init(&self) {
33 unsafe {
34 let r_ptr: *const Runtime = self;
35 RUNTIME = r_ptr as usize;
36 }
37 }
38
39 ...
40 fn main() {
41 let mut runtime = Runtime::new();
42 runtime.init();
43 ...
44 }
```

这样，在应用的 `main()` 函数中，首先会依次调用上述两个函数。这样线程管理运行时会附在 TID 为 0 的应用主线程上，处于运行正在运行的 *Running* 状态。而且，线程管理运行时也建立好了空闲线程控制块向量。后续创建线程时，会从此空闲线程控制块向量中找到一个空闲线程控制块，来绑定要创建的线程，并进行后续的管理。

## 线程创建

当应用要创建一个线程时，会调用 `runtime.spawn` 函数。这个函数主要完成的功能是：

- 第 4~12 行，在线程向量中查找一个状态为 *Available* 的空闲线程控制块；
- 第 14~20 行，初始化该空闲线程的线程控制块；
  - `x1` 寄存器：老的返回地址-`guard` 函数地址
  - `nx1` 寄存器：新的返回地址-输入参数 `f` 函数地址
  - `x2` 寄存器：新的栈地址-`available.stack+size`

```

1 impl Runtime {
2 pub fn spawn(&mut self, f: fn()) {
3 let available = self
4 .tasks
5 .iter_mut()
6 .find(|t| t.state == State::Available)
7 .expect("no available task.");
8
9 let size = available.stack.len();
10 unsafe {
11 let s_ptr = available.stack.as_mut_ptr().offset(size as isize);
12 let s_ptr = (s_ptr as usize & !7) as *mut u8;
13
14 available.ctx.x1 = guard as u64; //ctx.x1 is old return address
15 available.ctx.nx1 = f as u64; //ctx.nx1 is new return address
16 available.ctx.x2 = s_ptr.offset(-32) as u64; //ctx.x2 is sp
17
18 }
19 available.state = State::Ready;
20 }
21 }
```

(下页继续)

(续上页)

```

22
23 fn guard() {
24 unsafe {
25 let rt_ptr = RUNTIME as *mut Runtime;
26 (*rt_ptr).t_return();
27 };
28 }
29 ...
30 fn main() {
31 ...
32 runtime.spawn(|| {
33 println!("TASK 1 STARTING");
34 let id = 1;
35 for i in 0..10 {
36 println!("task: {} counter: {}", id, i);
37 yield_task();
38 }
39 println!("TASK 1 FINISHED");
40 });
41 ...
42 }

```

## 线程切换

当应用要切换线程时，会调用 `yield_task` 函数，通过 `runtime.t_yield` 函数来完成具体的切换过程。`runtime.t_yield` 这个函数主要完成的功能是：

- 第 4~12 行，在线程向量中查找一个状态为 *Ready* 的线程控制块；
- 第 14~20 行，把当前运行的线程的状态改为 *Ready*，把新就绪线程的状态改为 *Running*，把 `runtime` 的 `current` 设置为这个新线程控制块的 id；
- 第 23 行，调用汇编代码写的函数 `switch`，完成两个线程的栈和上下文的切换；

```

1 impl Runtime {
2 fn t_yield(&mut self) -> bool {
3 let mut pos = self.current;
4 while self.tasks[pos].state != State::Ready {
5 pos += 1;
6 if pos == self.tasks.len() {
7 pos = 0;
8 }
9 if pos == self.current {
10 return false;
11 }
12 }
13
14 if self.tasks[self.current].state != State::Available {
15 self.tasks[self.current].state = State::Ready;
16 }
17
18 self.tasks[pos].state = State::Running;
19 let old_pos = self.current;
20 self.current = pos;
21
22 unsafe {

```

(下页继续)

(续上页)

```

23 switch(&mut self.tasks[old_pos].ctx, &self.tasks[pos].ctx);
24 }
25 self.tasks.len() > 0
26 }
27 }
28
29 pub fn yield_task() {
30 unsafe {
31 let rt_ptr = RUNTIME as *mut Runtime;
32 (*rt_ptr).t_yield();
33 };
34 }
```

这里还需分析一下汇编函数 *switch* 的具体实现细节，才能完全掌握线程切换的完整过程。注意到切换线程控制块的函数 *t\_yield* 已经完成了当前运行线程的 *state*，*id* 这两个部分，还缺少：当前指令指针 (PC)、通用寄存器集合和栈。所以 *switch* 主要完成的就是完成这剩下的三部分的切换。

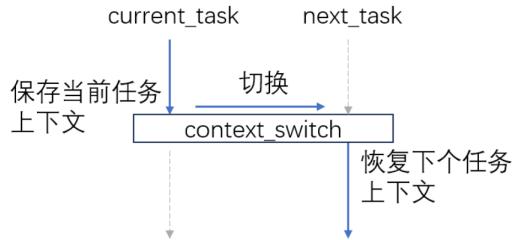
- 第 7, 14, 16, 23 行，完成当前指令指针 (PC) 的切换；
- 第 8, 17 行，完成栈指针的切换；
- 第 9-13, 18-22 行，完成通用寄存器集合的切换；

```

1 #[naked]
2 #[inline(never)]
3 unsafe fn switch(old: *mut TaskContext, new: *const TaskContext) {
4 // a0: old, a1: new
5 asm!(
6 //if comment below lines: sd x1..., ld x1..., TASK2 can not finish, and
7 //will segment fault
8 "sd x1, 0x00(a0)"
9 "sd x2, 0x08(a0)"
10 "sd x8, 0x10(a0)"
11 "sd x9, 0x18(a0)"
12 "sd x18, 0x20(a0) # sd x18..x27"
13 ...
14 "sd x27, 0x68(a0)"
15 "sd x1, 0x70(a0)"
16
16 "ld x1, 0x00(a1)"
17 "ld x2, 0x08(a1)"
18 "ld x8, 0x10(a1)"
19 "ld x9, 0x18(a1)"
20 "ld x18, 0x20(a1) #ld x18..x27"
21 ...
22 "ld x27, 0x68(a1)"
23 "ld t0, 0x70(a1)"
24
25 "jr t0"
26 ,
27 options(noreturn)
28);
29 }
```

这里需要注意两个细节。第一个是寄存器集合的保存数量。在保存通用寄存器集合时，并没有保存所有的通用寄存器，其原因是根据 RISC-V 的函数调用约定，有一部分寄存器是由调用函数 *Caller* 来保存的，所以就不需要被调用函数 *switch* 来保存了。第二个是当前指令指针 (PC) 的切换。在具体切换过程中，是基于函数返回地址来进行切换的。即首先把 *switch* 的函数返回地址 *ra* (即 *x1*) 寄存器保存在 *TaskContext* 中，在此函数的

倒数第二步，恢复切换后要执行线程的函数返回地址，即 *ra* 寄存器到 *t0* 寄存器，然后调用 *jr t0* 即完成了函数的返回。



## 开始执行

有了上述线程管理运行时的各种功能支持，就可以开始线程的正常执行了。假设完成了线程管理运行时初始化，并创建了几个线程。当执行 *runtime.run()* 函数时，通过 *t\_yield* 函数时，将切换线程管理运行时所在的应用主线程到另外一个处于 *Ready* 状态的线程，让那个线程开始执行。当所有的线程都执行完毕后，会回到 *runtime.run()* 函数，通过 *std::process::exit(0)* 来退出该应用进程，整个应用的运行就结束了。

```

1 impl Runtime {
2 pub fn run(&mut self) -> ! {
3 while self.t_yield() {}
4 std::process::exit(0);
5 }
6 }
7 ...
8 fn main() {
9 ...
10 runtime.run();
11 }

```

注：本节的内容参考了 Carl Fredrik Samson 设计实现的“Green Threads Example”<sup>12</sup>，并把代码移植到了 Linux for RISC-V64 上。

## 9.3 内核态的线程管理

### 9.3.1 本节导读

在上一节介绍了如何在用户态实现对多线程的管理，让同学对多线程的实际运行机制：创建线程、切换线程等有了一个比较全面的认识。由于在用户态进行线程管理，带了的一个潜在不足是没法让线程管理运行时直接切换线程，只能等当前运行的线程主动让出处理器使用权后，线程管理运行时才能切换检查。而我们的操作系统运行在处理器内核态，如果扩展一下对线程的管理，那就可以基于时钟中断来直接打断当前用户态线程的运行，实现对线程的调度和切换等。

本节参考上一节的用户态线程管理机制，结合之前的操作系统实现：具有 UNIX 操作系统基本功能的“迅猛龙”操作系统，进化为更加迅捷的“达科塔盗龙”<sup>1</sup> 操作系统，我们首先分析如何扩展现有的进程，以支持线程管理。然后设计线程的总体结构、管理线程执行的线程控制块数据结构、以及对线程管理相关的重要函

<sup>1</sup> <https://cfsamson.gitbook.io/green-threads-explained-in-200-lines-of-rust/>

<sup>2</sup> <https://github.com/cfsamson/example-greenthreads>

<sup>1</sup> 达科塔盗龙是一种生存于距今 6700 万-6500 万年前白垩纪晚期的兽脚类驰龙科恐龙，它主打的并不是霸王龙的力量路线，而是利用自己修长的后肢来提高敏捷度和奔跑速度。它全身几乎都长满了羽毛，可能会滑翔或者其他接近飞行行为的行动模式。

数：线程创建和线程切换。并最终合并到现有的进程管理机制中。本节的内容能帮助我们理解进程和线程的关系，二者在设计思想与具体实现上的差异，为后续支持各种并发机制打下基础。

#### 注解：为何要在这里才引入线程

学生对进程有一定了解后，再来理解线程也会更加容易。因为从对程序执行的控制流进行调度和切换上看，本章讲解的线程调度与切换操作是之前讲解的进程调度与切换的一个子集。另外，在这里引入线程的一个重要原因是便于引入并发。没在进程阶段引入并发这个专题的原因是，进程主要的目的是隔离，而线程的引入强调了共享，即属于同一进程的多个线程可共享进程的资源，这样就必须解决同步问题了。

### 9.3.2 线程概念

这里会结合与进程的比较来说明线程的概念。到本章之前，我们看到了进程这一抽象，操作系统让进程拥有相互隔离的虚拟的地址空间，让进程感到在独占一个虚拟的处理器。其实这只是操作系统通过时分复用和空分复用技术来让每个进程复用有限的物理内存和物理 CPU。而线程是在进程内的一个新的抽象。在没有线程之前，一个进程在一个时刻只有一个执行点（即程序计数器 PC 寄存器保存的要执行指令的指针以及栈的位置）。线程的引入把进程内的这个单一执行点给扩展为多个执行点，即在进程中存在多个线程，每个线程都有一个执行点。而且这些线程共享进程的地址空间，所以可以不必采用相对比较复杂的进程间通信机制（一般需要内核的介入）也可以很方便地直接访问进程内的数据进行协作。

在线程的具体运行过程中，需要有程序计数器寄存器来记录当前的执行位置，需要有一组通用寄存器记录当前的指令的操作数据，需要有一个栈作为线程执行过程的函数调用栈保存局部变量等内容，这就形成了线程上下文的主体部分。这样如果两个线程运行在一个处理器上，就需要采用类似两个进程运行在一个处理器上的调度/切换管理机制，即需要在一定时刻进行线程切换，并进行线程上下文的保存与恢复。这样在一个进程中的多线程可以独立运行，取代了进程，成为操作系统调度的基本单位。

由于把进程的结构进行了细化，通过线程来表示对处理器的虚拟化，使得进程成为了管理线程的容器。在进程中的线程没有父子关系，大家都是兄弟，但还是有个老大。这个代表老大的线程其实就是创建进程（比如通过 `fork` 系统调用创建进程）时建立的第一个线程，我们称之为为主线程。类似于进程标识符（PID），每个线程都有一个在所属进程内生效的线程标识符（TID），同进程下的两个线程有着不同的 TID，可以用来区分它们。主线程由于最先被创建，它的 TID 固定为 0。

### 9.3.3 通用操作系统多线程应用程序示例

当创建一个进程的时候，如果是基于 `fork` 系统调用的话，子进程会和父进程一样从系统调用返回后的下一条指令开始执行；如果是基于 `exec` 系统调用的话，子进程则会从新加载程序的入口点开始执行。而对于线程的话，除了主线程仍然从程序入口点（一般是 `main` 函数）开始执行之外，每个线程的生命周期都与程序中的一个函数的一次执行绑定。也就是说，线程从该函数入口点开始执行，当函数返回之后，线程也随之退出。因此，在创建线程的时候我们需要提供程序中的一个函数让线程来执行这个函数。

我们用 C 语言中的线程 API 来举例说明。在 C 语言中，常用的线程接口为 `pthread` 系列 API，这里的 `pthread` 意为 POSIX thread 即 POSIX 线程。这组 API 被很多不同的内核实现所支持，基于它实现的应用很容易在不同的平台间移植。`pthread` 创建线程的接口 `pthread_create` 如下：

```
#include <pthread.h>

int pthread_create(pthread_t *restrict thread,
 const pthread_attr_t *restrict attr,
 void *(*start_routine)(void *),
 void *restrict arg);
```

其中：

- 第一个参数为一个类型为 `pthread_t` 的线程结构体的指针。在实际创建线程之前我们首先要创建并初始化一个 `pthread_t` 的实例，它与线程一一对应，线程相关的操作都要通过它来进行。
- 通过第二个参数我们可以对要创建的线程进行一些配置，比如内核应该分配给这个线程多少栈空间。简单起见我们这里不展开。
- 第三个参数为一个函数指针，表示创建的线程要执行哪个函数。观察函数签名可以知道该函数的参数和返回值类型均被要求为一个 `void *`，这样是为了兼容各种不同的线程函数，因为 `void *` 可以和各种类型的指针相互转换。在声明函数的时候要遵循这个约定，但实现的时候我们常常需要首先将 `void *` 转化为具体类型的指针。
- 第四个参数为传给线程执行的函数的参数，类型为 `void *`，和函数签名中的约定一致。需要这个参数的原因是：方便区分，我们常常会让很多线程执行同一个函数，但可以传给它们不同的参数，以这种手段来对它们进行区分。

让我们来看一个例子：

```

1 #include <stdio.h>
2 #include <pthread.h>
3
4 typedef struct {
5 int x;
6 int y;
7 } FuncArguments;
8
9 void *func(void *arg) {
10 FuncArguments *args = (FuncArguments*)arg;
11 printf("x=%d, y=%d\n", args->x, args->y);
12 }
13
14 void main() {
15 pthread_t t0, t1, t2;
16 FuncArguments args[3] = {{1, 2}, {3, 4}, {5, 6}};
17 pthread_create(&t0, NULL, func, &args[0]);
18 pthread_create(&t1, NULL, func, &args[1]);
19 pthread_create(&t2, NULL, func, &args[2]);
20 pthread_join(t0, NULL);
21 pthread_join(t1, NULL);
22 pthread_join(t2, NULL);
23 return;
24 }
```

- 第 4~7 行我们声明线程函数接受的参数类型为一个名为 `FuncArguments` 的结构体，内含 `x` 和 `y` 两个字段。
- 第 15 行我们创建并默认初始化三个 `pthread_t` 实例 `t0`、`t1` 和 `t2`，分别代表我们接下来要创建的三个线程。
- 第 16 行在主线程的栈上给出三个线程接受的参数。
- 第 9~12 行实现线程运行的函数 `func`，可以看到它的函数签名符合要求。它实际接受的参数类型应该为我们之前定义的 `FuncArguments` 类型的指针，但是在函数签名中是一个 `void *`，所以在第 10 行我们首先进行类型转换得到 `FuncArguments*`，而后才能访问 `x` 和 `y` 两个字段并打印出来。
- 第 17~19 行使用 `pthread_create` 创建三个线程，分别绑定到 `t0~t2` 三个 `pthread_t` 实例上。它们均执行 `func` 函数，但接受的参数有所不同。

编译运行，一种可能的输出为：

```
x=1, y=2
x=5, y=6
x=3, y=4
```

从中可以看出，线程的实际执行顺序不一定和我们创建它们的顺序相同。在创建完三个线程之后，同时存在四个线程，即创建的三个线程和主线程，它们的执行顺序取决于操作系统如何调度它们。这可能导致主线程先于我们创建的线程结束，在一些内核实现中，这种情况下整个进程直接退出，于是我们创建的线程也直接被删除，未正常返回，没有达到我们期望的效果。为了解决这个问题，我们可以使用 `pthread_join` 函数来使主线程等待某个线程退出之后再继续向下执行。其函数签名为：

```
#include <pthread.h>

int pthread_join(pthread_t thread, void **retval);
```

我们需要传入线程对应的 `pthread_t` 实例来等待一个线程退出。另一个参数 `retval` 是用来捕获线程函数的返回值，我们这里不展开。上面代码片段的第 20~22 行我们便要求主线程依次等待我们创建的三个线程退出之后再退出，这样主线程就不会影响到其他线程的执行。

在开发 Rust 多线程应用的时候，我们也可以使用标准库 `std::thread` 中提供的 API 来创建、管理或结束线程。其中：

- `std::thread::spawn` 类似于 `pthread_create`，可以创建一个线程，它会返回一个 `JoinHandle` 代表创建的线程；
- `std::thread::JoinHandle::join` 类似于 `pthread_join`，用来等待调用 `join` 的 `JoinHandle` 对应的线程结束。

下面使用 Rust 重写上面基于 C 语言的多线程应用：

```
1 use std::thread;
2 struct FuncArguments {
3 x: i32,
4 y: i32,
5 }
6 fn func(args: FuncArguments) -> i32 {
7 println!("x={}, y={}", args.x, args.y);
8 args.x + args.y
9 }
10 fn main() {
11 let v = vec![
12 thread::spawn(|| func(FuncArguments {x: 1, y: 2})),
13 thread::spawn(|| func(FuncArguments {x: 3, y: 4})),
14 thread::spawn(|| func(FuncArguments {x: 5, y: 6})),
15];
16 for handle in v {
17 println!("result={}", handle.join().unwrap());
18 }
19 }
```

可以看到，相比 C 语言，在 Rust 实现中无需进行繁琐的类型转换，直接正常将参数传入 `thread::spawn` 所需的闭包中即可。同时使用 `handle.join` 即可接收线程函数的返回值。一种可能的运行结果如：

```
x=1, y=2
result=3
x=3, y=4
x=5, y=6
result=7
result=11
```

从中可以观察到主线程和我们创建的线程在操作系统的调度下交错运行。

为了在“达科塔盗龙”<sup>Page 10, 1</sup>操作系统中实现类似 Linux 操作系统的多线程支持，我们需要建立精简的线程模型和相关系统调用，并围绕这两点来改进操作系统。

### 9.3.4 线程模型与重要系统调用

目前，我们只介绍本章实现的内核中采用的一种非常简单的线程模型。这个线程模型有如下特征：

- 线程有三种状态：就绪态、运行态和阻塞态（阻塞态是本章后面并发部分的重点概念，到时会详细讲解）；
- 同进程下的所有线程共享所属进程的地址空间和其他共享资源（如文件等）；
- 线程可被操作系统调度来分时占用 CPU 执行；
- 线程可以动态创建和退出；
- 同进程下的多个线程不像进程一样存在父子关系，但有一个特殊的主线程在它所属进程被创建的时候产生，应用程序的 main 函数就运行在这个主线程上。当主线程退出后，整个进程立即退出，也就意味着不论进程下的其他线程处于何种状态也随之立即退出；
- 线程可通过系统调用获得操作系统的服务。注意线程和进程两个系列的系统调用不能混用。

我们实现的线程模型建立在进程的地址空间抽象之上：同进程下的所有线程共享该进程的地址空间，包括代码段和数据段。从逻辑上来说某些段是由所有线程共享的（比如包含代码中的全局变量的全局数据段），而某些段是由某个线程独占的（比如操作系统为每个线程分配的栈），通常情况下程序员会遵循这种约定。然而，线程之间并不能严格做到隔离。举例来说，一个线程访问另一个线程的栈这种行为并不会被操作系统和硬件禁止。这也体现了线程和进程的不同：线程的诞生是为了方便共享，而进程更强调隔离。

此外，线程模型还需要操作系统支持一些重要的系统调用：创建线程、等待线程结束等来支持灵活的多线程应用。接下来我们介绍这些系统调用的基本功能和设计思路。

#### 线程的创建

在我们的内核中，通过 `thread_create` 系统调用可以创建一个从属于当前进程的线程。类似于 C 标准库中的 `pthread_create`，我们也需要传入线程运行的函数的入口地址和函数接受的参数。不同之处在于：`pthread` 系列 API 中基于 `pthread_t` 实例对线程进行控制，而我们则是用线程的线程标识符（TID, Thread Identifier）来区分不同线程并对指定线程进行控制，这一点类似于进程的控制方式。因此，在没有出错的情况下，我们的 `thread_create` 系统调用会返回创建的线程的 TID。具体系统调用原型如下：

```

1 /// 功能：当前进程创建一个新的线程
2 /// 参数：entry 表示线程的入口函数地址，arg 表示传给线程入口函数参数
3 /// 返回值：创建的线程的 TID
4 /// syscall ID: 1000
5 pub fn sys_thread_create(entry: usize, arg: usize) -> isize;

```

内核会为每个线程分配一组专属于该线程的资源：用户栈、Trap上下文还有内核栈，前面两个在进程地址空间中，内核栈在内核地址空间中。这样这些线程才能相对独立地被调度和执行。相比于创建进程的 `fork` 系统调用，创建线程无需建立新的地址空间，这是二者之间最大的不同。另外属于同一进程中的线程之间没有父子关系，这一点也与进程不一样。

## 简单线程管理

类似 `getpid`，我们新增了一个 `gettid` 的系统调用可以获取当前线程的 TID，其 syscall ID 为 1001。由于比较简单，在这里不再赘述。

## 线程退出及资源回收

在 C/Rust 语言实现的多线程应用中，当线程执行的函数返回之后线程会自动退出，在编程的时候无需对函数做任何特殊处理。其实现原理是当函数返回之后，会自动跳转到用户态一段预先设置好的代码，在这段代码中通过系统调用实现线程退出操作。在这里，我们为了让实现更加简单，约定线程函数需要在返回之前通过 `exit` 系统调用退出。这里 `exit` 系统调用的含义发生了变化：从进程退出变成线程退出。

内核在收到线程发出的 `exit` 系统调用后，会回收线程占用的用户态资源，包括用户栈和 Trap 上下文等。线程占用的内核态资源（包括内核栈等）则需要在进程内使用 `waittid` 系统调用回来回收，这样该线程占用的资源才能被完全回收。`waittid` 的系统调用原型如下：

```

1 // 功能：等待当前进程内的一个指定线程退出
2 // 参数：tid 表示指定线程的 TID
3 // 返回值：如果线程不存在，返回 -1；如果线程还没退出，返回 -2；其他情况下，返回结束线程的退出码
4 // syscall ID: 1002
5 pub fn sys_waittid(tid: usize) -> i32;

```

`waittid` 基本上就是把我们比较熟悉的 `waitpid` 的操作对象从进程换成了线程，使用方法也和 `waitpid` 比较像。它像 `pthread_join` 一样能起到一定的同步作用，也能够彻底回收一个线程的资源。一般情况下进程/主线程要负责通过 `waittid` 来等待它创建出来的线程（不是主线程）结束并回收它们在内核中的资源（如线程的内核栈、线程控制块等）。如果进程/主线程先调用了 `exit` 系统调用退出，那么整个进程（包括所属的所有线程）都会退出，而对应父进程会通过 `waitpid` 回收子进程剩余还没被回收的资源。

## 进程相关的系统调用

在引入了线程机制后，进程相关的重要系统调用：`fork`、`exec`、`waitpid` 虽然在接口上没有变化，但在它要完成的功能上需要有一定的扩展。首先，需要注意到把以前进程中与处理器执行相关的部分拆分到线程中。这样，在通过 `fork` 创建进程其实也意味着要单独建立一个主线程来使用处理器，并为以后创建新的线程建立相应的线程控制块向量。相对而言，`exec` 和 `waitpid` 这两个系统调用要做的改动比较小，还是按照与之前进程的处理方式来进行。

而且，为了实现更加简单，我们要求每个应用对于 **线程和进程两个系列的系统调用只能使用其中之一**。比如说，使用了进程系列的 `fork` 就不能使用线程系列的 `thread_create`，这是因为很难定义如何 `fork` 一个多线程的进程。类似的，可以发现要将进程和线程模型融合起来需要做很多额外的工作。如果做了上述要求的话，我们就可以对进程-线程的融合模型进行简化。如果涉及到父子进程的交互，那么这些进程只会有一个主线程，基本等价于之前的进程模型；如果使用 `thread_create` 创建了新线程，那么我们只需考虑多个线程在同一个进程内的交互。因此，总体上看，进程相关的这三个系统调用还是保持了已有的进程操作的语义，并没有由于引入了线程，而带来大的变化。

### 9.3.5 应用程序示例

我们刚刚介绍了 `thread_create`/`waittid` 两个重要系统调用，我们可以借助它们和之前实现的系统调用开发出功能更为灵活的应用程序。下面我们通过描述一个多线程应用程序 `threads` 的开发过程，来展示这些系统调用的使用方法。

#### 系统调用封装

同学可以在 `user/src/syscall.rs` 中看到以 `sys_*` 开头的系统调用的函数原型，它们后续还会在 `user/src/lib.rs` 中被封装成方便应用程序使用的形式。如 `sys_thread_create` 被封装成 `thread_create`，而 `sys_waittid` 被封装成 `waittid`：

```

1 // user/src/lib.rs
2
3 pub fn thread_create(entry: usize, arg: usize) -> isize {
4 sys_thread_create(entry, arg)
5 }
6
7 pub fn waittid(tid: usize) -> isize {
8 loop {
9 match sys_waittid(tid) {
10 -2 => { yield_(); }
11 exit_code => return exit_code,
12 }
13 }
14 }
```

`waittid` 等待一个线程标识符的值为 `tid` 的线程结束。在具体实现方面，我们看到当 `sys_waittid` 返回值为 `-2`，即要等待的线程存在但它却尚未退出的时候，主线程调用 `yield_` 主动交出 CPU 使用权，待下次 CPU 使用权被内核交还给它的时候再次调用 `sys_waittid` 查看要等待的线程是否退出。这样做是为了减小 CPU 资源的浪费以及尽可能简化内核的实现。

#### 多线程应用程序

多线程应用程序 `threads` 开始执行后，先调用 `thread_create` 创建了三个线程，加上进程自带的主线程，其实一共有四个线程。每个线程在打印了 1000 个字符后，会执行 `exit` 退出。进程通过 `waittid` 等待这三个线程结束后，最终结束进程的执行。下面是多线程应用程序 `threads` 的源代码：

```

1 //usr/src/bin/threads.rs
2
3 #![no_std]
4 #![no_main]
5
6 #[macro_use]
7 extern crate user_lib;
8 extern crate alloc;
9
10 use user_lib::{thread_create, waittid, exit};
11 use alloc::vec::Vec;
12
13 pub fn thread_a() -> ! {
14 for _ in 0..1000 { print!("a"); }
15 exit(1)
16 }
```

(下页继续)

(续上页)

```

17
18 pub fn thread_b() -> ! {
19 for _ in 0..1000 { print!("b"); }
20 exit(2)
21 }
22
23 pub fn thread_c() -> ! {
24 for _ in 0..1000 { print!("c"); }
25 exit(3)
26 }
27
28 #[no_mangle]
29 pub fn main() -> i32 {
30 let mut v = Vec::new();
31 v.push(thread_create(thread_a as usize, 0));
32 v.push(thread_create(thread_b as usize, 0));
33 v.push(thread_create(thread_c as usize, 0));
34 for tid in v.iter() {
35 let exit_code = waittid(*tid as usize);
36 println!("thread#{} exited with code {}", tid, exit_code);
37 }
38 println!("main thread exited.");
39 0
40 }

```

另一个名为 threads\_arg 的应用和 threads 的功能相同，其不同在于利用 thread\_create 可以传参的特性，从而只需编写一个线程函数。

```

1 #[no_std]
2 #[no_main]
3
4 #[macro_use]
5 extern crate user_lib;
6 extern crate alloc;
7
8 use alloc::vec::Vec;
9 use user_lib::exit, thread_create, waittid;
10
11 struct Argument {
12 pub ch: char,
13 pub rc: i32,
14 }
15
16 fn thread_print(arg: *const Argument) -> ! {
17 let arg = unsafe { &*arg };
18 for _ in 0..1000 {
19 print!("{} ", arg.ch);
20 }
21 exit(arg.rc)
22 }
23
24 #[no_mangle]
25 pub fn main() -> i32 {
26 let mut v = Vec::new();
27 let args = [
28 Argument { ch: 'a', rc: 1 },

```

(下页继续)

(续上页)

```

29 Argument { ch: 'b', rc: 2 },
30 Argument { ch: 'c', rc: 3 },
31];
32 for arg in args.iter() {
33 v.push(thread_create(
34 thread_print as usize,
35 arg as *const _ as usize,
36));
37 }
38 for tid in v.iter() {
39 let exit_code = waittid(*tid as usize);
40 println!("thread#{} exited with code {}", tid, exit_code);
41 }
42 println!("main thread exited.");
43 0
44 }

```

这里传给创建的三个线程的参数放在主线程的栈上，在 `thread_create` 的时候提供的是对应参数的地址。参数会决定每个线程打印的字符和线程的返回码。

### 9.3.6 线程管理的核心数据结构

为了实现线程机制，我们需要将操作系统的 CPU 资源调度单位（也即“任务”）从之前的进程改为线程。这意味着调度器需要考虑更多的因素，比如当一个线程时间片用尽交出 CPU 使用权的时候，切换到同进程下还是不同进程下的线程的上下文切换开销往往有很大不同，可能影响到是否需要切换页表。不过我们为了实现更加简单，仍然采用 Round-Robin 调度算法，将所有线程一视同仁，不考虑它们属于哪个进程。

本章之前，进程管理的三种核心数据结构和一些软/硬件资源如下：

第一个数据结构是任务（进程）控制块 `TaskControlBlock`，可以在 `os/src/task/task.rs` 中找到。它除了记录当前任务的状态之外，还包含如下资源：

- 进程标识符 `pid`；
- 内核栈 `kernel_stack`；
- 进程地址空间 `memory_set`；
- 进程地址空间中的用户栈和 Trap 上下文（进程控制块中相关字段为 `trap_cx_ppn`）；
- 文件描述符表 `fd_table`；
- 信号相关的字段。

这些资源的生命周期基本上与进程的生命周期相同。但是在有了线程之后，我们需要将一些与代码执行相关的资源分离出来，让它们与相关线程的生命周期绑定。

第二个数据结构是任务管理器 `TaskManager`，可以在 `os/src/task/manager.rs` 中找到。它实质上是我们内核的调度器，可以决定一个任务时间片用尽或退出之后接下来执行哪个任务。

第三个数据结构是处理器管理结构 `Processor`，可以在 `os/src/task/processor.rs` 中找到。它维护了处理器当前在执行哪个任务，在处理系统调用的时候我们需要依据这里的记录来确定系统调用的发起者是哪个任务。

本章的变更如下：

- 进程控制块由之前的 `TaskControlBlock` 变成新增的 `ProcessControlBlock`（简称 PCB），我们在其中保留进程的一些信息以及由进程中所有线程共享的一些资源。PCB 可以在 `os/src/task/`

process.rs 中找到。任务控制块 TaskControlBlock 则变成用来描述线程的线程控制块，包含线程的信息以及线程独占的资源。

- 在资源管理方面，本章之前在 os/src/task/pid.rs 可以看到与进程相关的一些 RAII 风格的软/硬件资源，包括进程描述符 PidHandle 以及内核栈 KernelStack，其中内核栈被分配在内核地址空间中且其位置由所属进程的进程描述符决定。本章将 pid.rs 替换为 id.rs，仍然保留 PidHandle 和 KernelStack 两种资源，不过 KernelStack 变为一种线程独占的资源，我们可以在线程控制块 TaskControlBlock 中找到它。此外，我们还在 id.rs 中新增了 TaskUserRes 表示每个线程独占的用户态资源，还有一个各类资源的通用分配器 RecycleAllocator。
- CPU 资源调度单位仍为任务控制块 TaskControlBlock 不变。因此，基于任务控制块的任务控制器 TaskManager 和处理器管理结构 Processor 也基本不变，只有某些接口有小幅修改。

接下来依次对它们进行介绍。

### 通用资源分配器及线程相关的软硬件资源

在 os/src/task/id.rs 中，我们将之前的 PidAllocator 改造为通用的资源分配器 RecycleAllocator 用来分配多种不同的资源。这些资源均为 RAII 风格，可以在被 drop 掉之后自动进行资源回收：

- 进程描述符 PidHandle；
- 线程独占的线程资源组 TaskUserRes，其中包括线程描述符；
- 线程独占的内核栈 KernelStack。

通用资源分配器 RecycleAllocator 的实现如下：

```

1 // os/src/task/id.rs
2
3 pub struct RecycleAllocator {
4 current: usize,
5 recycled: Vec<usize>,
6 }
7
8 impl RecycleAllocator {
9 pub fn new() -> Self {
10 RecycleAllocator {
11 current: 0,
12 recycled: Vec::new(),
13 }
14 }
15 pub fn alloc(&mut self) -> usize {
16 if let Some(id) = self.recycled.pop() {
17 id
18 } else {
19 self.current += 1;
20 self.current - 1
21 }
22 }
23 pub fn deallocate(&mut self, id: usize) {
24 assert!(id < self.current);
25 assert!(
26 !self.recycled.iter().any(|i| *i == id),
27 "id {} has been deallocated!",
28 id
29);
30 }
31 }
```

(下页继续)

(续上页)

```

30 self.recycled.push(id);
31 }
32 }
```

分配与回收的算法与之前的 PidAllocator 一样，不过分配的内容从 PID 变为一个非负整数的通用标识符，可以用来表示多种不同资源。这个通用整数标识符可以直接用作进程的 PID 和进程内一个线程的 TID。下面是 PID 的全局分配器 PID\_ALLOCATOR：

```

// os/src/task/id.rs

lazy_static! {
 static ref PID_ALLOCATOR: UPSafeCell<RecycleAllocator> =
 unsafe { UPSafeCell::new(RecycleAllocator::new()) };
}

pub fn pid_alloc() -> PidHandle {
 PidHandle(PID_ALLOCATOR.exclusive_access().alloc())
}

impl Drop for PidHandle {
 fn drop(&mut self) {
 PID_ALLOCATOR.exclusive_access().dealloc(self.0);
 }
}
```

调用 pid\_alloc 可以从全局 PID 分配器中分配一个 PID 并构成一个 RAII 风格的 PidHandle。当 PidHandle 被回收的时候则会自动调用 drop 方法在全局 PID 分配器将对应的 PID 回收。

对于 TID 而言，每个进程控制块中都有一个给进程内的线程分配资源的通用分配器：

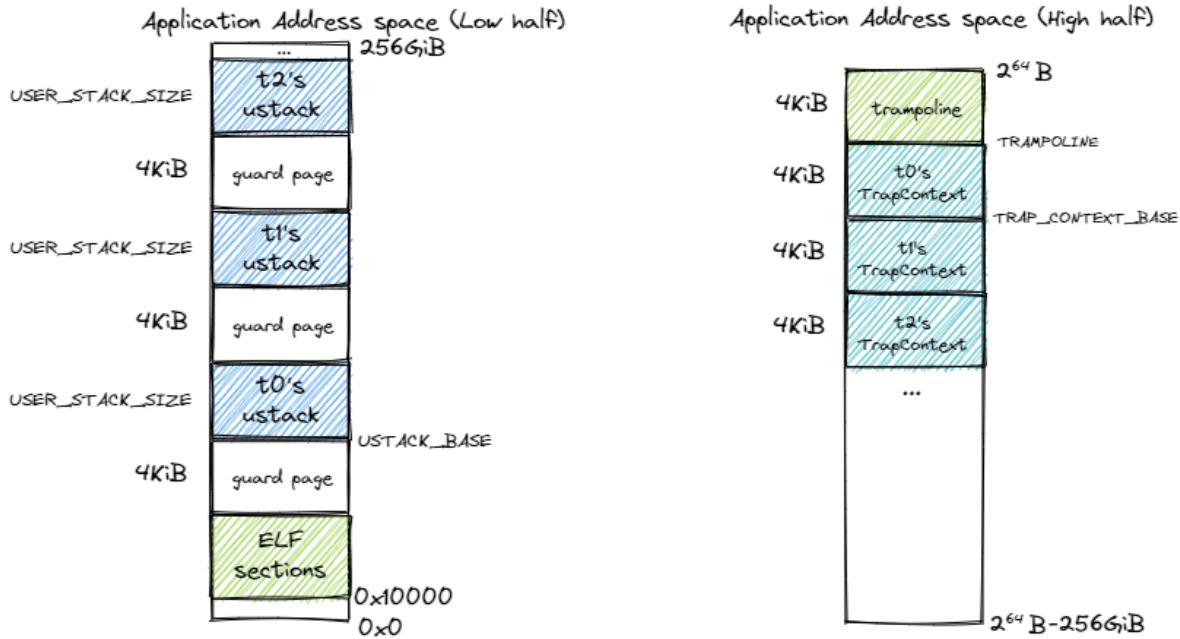
```

// os/src/task/process.rs

1 pub struct ProcessControlBlock {
2 // immutable
3 pub pid: PidHandle,
4 // mutable
5 inner: UPSafeCell<ProcessControlBlockInner>,
6 }
7
8 pub struct ProcessControlBlockInner {
9 ...
10 pub task_res_allocator: RecycleAllocator,
11 ...
12 }
13
14 impl ProcessControlBlockInner {
15 pub fn alloc_tid(&mut self) -> usize {
16 self.task_res_allocator.alloc()
17 }
18
19 pub fn deallocate_tid(&mut self, tid: usize) {
20 self.task_res_allocator.dealloc(tid)
21 }
22 }
23 }
```

可以看到进程控制块中有一个名为 task\_res\_allocator 的通用分配器，同时还提供了 alloc\_tid 和 deallocate\_tid 两个接口来分别在创建线程和销毁线程的时候分配和回收 TID。除了 TID 之外，每个线程都

有自己独立的用户栈和 Trap 上下文，且它们在所属进程的地址空间中的位置可由 TID 计算得到。参考新的进程地址空间如下图所示：



可以看到，在低地址空间中，在放置完应用 ELF 的所有段之后，会预留 4KiB 的空间作为保护页，得到地址 **ustack\_base**，这部分实现可以参考创建应用地址空间的 `MemorySet::from_elf`，**ustack\_base** 即为其第二个返回值。接下来从 **ustack\_base** 开始按照 TID 从小到大的顺序向高地址放置线程的用户栈，两两之间预留一个保护页放置栈溢出。在高地址空间中，最高的虚拟页仍然作为跳板页，跳板页中放置的是只读的代码，因此线程之间可以共享。然而，每个线程需要有自己的 Trap 上下文，于是我们在跳板页的下面向低地址按照 TID 从小到大的顺序放置线程的 Trap 上下文。也就是说，只要知道线程的 TID，我们就可以计算出线程在所属进程地址空间内的用户栈和 Trap 上下文的位置，计算方式由下面的代码给出：

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

pub const TRAMPOLINE: usize = usize::MAX - PAGE_SIZE + 1;
pub const TRAP_CONTEXT_BASE: usize = TRAMPOLINE - PAGE_SIZE;

// os/src/task/id.rs

fn trap_cx_bottom_from_tid(tid: usize) -> usize {
 TRAP_CONTEXT_BASE - tid * PAGE_SIZE
}

fn ustack_bottom_from_tid(ustack_base: usize, tid: usize) -> usize {
 ustack_base + tid * (PAGE_SIZE + USER_STACK_SIZE)
}
```

线程的 TID、用户栈和 Trap 上下文均和线程的生命周期相同，因此我们可以将它们打包到一起统一进行分配和回收。这就形成了名为 **TaskUserRes** 的线程资源集合，它可以在任务(线程)控制块 **TaskControlBlock** 中找到：

```
1 // os/src/task/id.rs
2
3 pub struct TaskUserRes {
4 pub tid: usize,
```

(下页继续)

(续上页)

```

5 pub ustack_base: usize,
6 pub process: Weak<ProcessControlBlock>,
7 }
8
9 impl TaskUserRes {
10 pub fn new(
11 process: Arc<ProcessControlBlock>,
12 ustack_base: usize,
13 alloc_user_res: bool,
14) -> Self {
15 let tid = process.inner_exclusive_access().alloc_tid();
16 let task_user_res = Self {
17 tid,
18 ustack_base,
19 process: Arc::downgrade(&process),
20 };
21 if alloc_user_res {
22 task_user_res.alloc_user_res();
23 }
24 task_user_res
25 }
26
27 /// 在进程地址空间中实际映射线程的用户栈和 Trap 上下文。
28 pub fn alloc_user_res(&self) {
29 let process = self.process.upgrade().unwrap();
30 let mut process_inner = process.inner_exclusive_access();
31 // alloc user stack
32 let ustack_bottom = ustack_bottom_from_tid(self.ustack_base, self.tid);
33 let ustack_top = ustack_bottom + USER_STACK_SIZE;
34 process_inner.memory_set.insert_framed_area(
35 ustack_bottom.into(),
36 ustack_top.into(),
37 MapPermission::R | MapPermission::W | MapPermission::U,
38);
39 // alloc trap_cx
40 let trap_cx_bottom = trap_cx_bottom_from_tid(self.tid);
41 let trap_cx_top = trap_cx_bottom + PAGE_SIZE;
42 process_inner.memory_set.insert_framed_area(
43 trap_cx_bottom.into(),
44 trap_cx_top.into(),
45 MapPermission::R | MapPermission::W,
46);
47 }
48 }

```

TaskUserRes 中记录了进程分配的 TID，用来计算线程用户栈位置的 `ustack_base`。我们还需要所属进程的弱引用，因为 TaskUserRes 中的资源都在进程控制块中，特别是其中的用户栈和 Trap 上下文需要在进程地址空间 `MemorySet` 中进行映射。因此我们需要进程控制块来完成实际的资源分配和回收。

在使用 `TaskUserRes::new` 新建的时候进程控制块会分配一个 TID 用于初始化，但并不一定调用 `TaskUserRes::alloc_user_res` 在进程地址空间中实际对用户栈和 Trap 上下文进行映射，这要取决于 `new` 参数中的 `alloc_user_res` 是否为真。举例来说，在 `fork` 子进程并创建子进程的主线程的时候，就不必再分配一次用户栈和 Trap 上下文，因为子进程拷贝了父进程的地址空间，这些内容已经被映射过了。因此这个时候 `alloc_user_res` 为假。其他情况下则需要进行映射。

当线程退出之后，`TaskUserRes` 会随着线程控制块一起被回收，意味着进程分配给它的资源也会被回收：

```

1 // os/src/task/id.rs
2
3 impl TaskUserRes {
4 fn deallocate_user_res(&self) {
5 // deallocate tid
6 let process = self.process.upgrade().unwrap();
7 let mut process_inner = process.inner_exclusive_access();
8 // deallocate ustack manually
9 let ustack_bottom_va: VirtAddr = ustack_bottom_from_tid(self.ustack_base, _

10 self.tid).into();
11 process_inner
12 .memory_set
13 .remove_area_with_start_vpn(ustack_bottom_va.into());
14 // deallocate trap_cx manually
15 let trap_cx_bottom_va: VirtAddr = trap_cx_bottom_from_tid(self.tid).into();
16 process_inner
17 .memory_set
18 .remove_area_with_start_vpn(trap_cx_bottom_va.into());
19 }
20 pub fn deallocate_tid(&self) {
21 let process = self.process.upgrade().unwrap();
22 let mut process_inner = process.inner_exclusive_access();
23 process_inner.deallocate_tid(self.tid);
24 }
25 }
26
27 impl Drop for TaskUserRes {
28 fn drop(&mut self) {
29 self.deallocate_tid();
30 self.deallocate_user_res();
31 }
32 }

```

可以看到我们依次调用 `deallocate_tid` 和 `deallocate_user_res` 使进程控制块回收掉当前 TID 并在进程地址空间中解映射线程用户栈和 Trap 上下文。

接下来是内核栈 `KernelStack`。与之前一样它是从内核高地址空间的跳板页下面开始分配，每两个内核栈中间用一个保护页隔开，因此总体地址空间布局和之前相同。不同的则是它的位置不再与 PID 或者 TID 挂钩，而是与一种新的内核栈标识符有关。我们需要新增一个名为 `KSTACK_ALLOCATOR` 的通用资源分配器来对内核栈标识符进行分配。

```

1 // os/src/task/id.rs
2
3 lazy_static! {
4 static ref KSTACK_ALLOCATOR: UPSafeCell<RecycleAllocator> =
5 unsafe { UPSafeCell::new(RecycleAllocator::new()) };
6 }
7
8 pub struct KernelStack(pub usize);
9
10 /// Return (bottom, top) of a kernel stack in kernel space.
11 pub fn kernel_stack_position(kstack_id: usize) -> (usize, usize) {
12 let top = TRAMPOLINE - kstack_id * (KERNEL_STACK_SIZE + PAGE_SIZE);
13 let bottom = top - KERNEL_STACK_SIZE;
14 (bottom, top)
15 }
16

```

(下页继续)

(续上页)

```

17 pub fn kstack_alloc() -> KernelStack {
18 let kstack_id = KSTACK_ALLOCATOR.exclusive_access().alloc();
19 let (kstack_bottom, kstack_top) = kernel_stack_position(kstack_id);
20 KERNEL_SPACE.exclusive_access().insert_framed_area(
21 kstack_bottom.into(),
22 kstack_top.into(),
23 MapPermission::R | MapPermission::W,
24);
25 KernelStack(kstack_id)
26 }
27
28 impl Drop for KernelStack {
29 fn drop(&mut self) {
30 let (kernel_stack_bottom, _) = kernel_stack_position(self.0);
31 let kernel_stack_bottom_va: VirtAddr = kernel_stack_bottom.into();
32 KERNEL_SPACE
33 .exclusive_access()
34 .remove_area_with_start_vpn(kernel_stack_bottom_va.into());
35 KSTACK_ALLOCATOR.exclusive_access().dealloc(self.0);
36 }
37 }

```

KSTACK\_ALLOCATOR 分配/回收的是内核栈标识符 kstack\_id，基于它可以用 kernel\_stack\_position 函数计算出内核栈在内核地址空间中的位置。进而，kstack\_alloc 和 KernelStack::drop 分别在内核地址空间中通过映射/解映射完成内核栈的分配和回收。

于是，我们就将通用资源分配器和三种软硬件资源的分配和回收机制介绍完了，这也是线程机制中最关键的一个部分。

## 进程和线程控制块

在引入线程机制之后，线程就代替进程成为了 CPU 资源的调度单位——任务。因此，代码执行有关的一些内容被分离到任务（线程）控制块中，其中包括线程状态、各类上下文和线程独占的一些资源等。线程控制块 TaskControlBlock 是内核对线程进行管理的核心数据结构。在内核看来，它就等价于一个线程。

```

1 // os/src/task/task.rs
2
3 pub struct TaskControlBlock {
4 // immutable
5 pub process: Weak<ProcessControlBlock>,
6 pub kstack: KernelStack,
7 // mutable
8 inner: UPSafeCell<TaskControlBlockInner>,
9 }
10
11 pub struct TaskControlBlockInner {
12 pub res: Option<TaskUserRes>,
13 pub trap_cx_ppn: PhysPageNum,
14 pub task_cx: TaskContext,
15 pub task_status: TaskStatus,
16 pub exit_code: Option<i32>,
17 }

```

线程控制块中的不变量有所属进程的弱引用和自身的内核栈。在可变的 inner 里面则保存了线程资源集合 TaskUserRes 和 Trap 上下文。任务上下文 TaskContext 仍然保留在线程控制块中，这样才能正常进行线程切换。此外，还有线程状态 TaskStatus 和线程退出码 exit\_code。

进程控制块中则保留进程内所有线程共享的资源：

```

1 // os/src/task/process.rs
2
3 pub struct ProcessControlBlock {
4 // immutable
5 pub pid: PidHandle,
6 // mutable
7 inner: UPSafeCell<ProcessControlBlockInner>,
8 }
9
10 pub struct ProcessControlBlockInner {
11 pub is_zombie: bool,
12 pub memory_set: MemorySet,
13 pub parent: Option<Weak<ProcessControlBlock>>,
14 pub children: Vec<Arc<ProcessControlBlock>>,
15 pub exit_code: i32,
16 pub fd_table: Vec<Option<Arc<dyn File + Send + Sync>>>,
17 pub signals: SignalFlags,
18 pub tasks: Vec<Option<Arc<TaskControlBlock>>>,
19 pub task_res_allocator: RecycleAllocator,
20 ... // 其他同步互斥相关资源
21 }
```

其中 pid 为进程标识符，它在进程创建后的整个生命周期中不再变化。可变的 inner 中的变化如下：

- 第 18 行在进程控制块里面设置一个向量保存进程中所有线程的任务控制块。其布局与文件描述符表比较相似，可以看成一组可以拓展的线程控制块插槽；
- 第 19 行是进程为进程内的线程分配资源的通用资源分配器 RecycleAllocator。

## 任务管理器与处理器管理结构

任务管理器 TaskManager 和处理器管理结构 Processor 分别在 task/manager.rs 和 task/processor.rs 中。它们的接口和功能和之前基本上一致，但是由于任务控制块 TaskControlBlock 和进程控制块 ProcessControlBlock 和之前章节的语义不同，部分接口略有改动。让我们再总体回顾一下它们对外提供的接口：

```

// os/src/task/manager.rs
/// 全局变量：
/// 1. 全局任务管理器 TASK_MANAGER
/// 2. 全局 PID-进程控制块映射 PID2TCB

/// 将线程加入就绪队列
pub fn add_task(task: Arc<TaskControlBlock>);
/// 将线程移除出就绪队列
pub fn remove_task(task: Arc<TaskControlBlock>);
/// 从就绪队列中选出一个线程分配 CPU 资源
pub fn fetch_task() -> Option<Arc<TaskControlBlock>>;
/// 根据 PID 查询进程控制块
pub fn pid2process(pid: usize) -> Option<Arc<ProcessControlBlock>>;
/// 增加一对 PID-进程控制块映射
pub fn insert_into_pid2process(pid: usize, process: Arc<ProcessControlBlock>);
/// 删除一对 PID-进程控制块映射
pub fn remove_from_pid2process(pid: usize);
```

(下页继续)

(续上页)

```
// os/src/task/processor.rs

/// 全局变量: 当前处理器管理结构 PROCESSOR

/// CPU 的调度主循环
pub fn run_tasks();
/// 取出当前处理器正在执行的线程
pub fn take_current_task() -> Option<Arc<TaskControlBlock>>;
/// 当前线程控制块/进程控制块/进程地址空间 satp/线程Trap上下文
pub fn current_task() -> Option<Arc<TaskControlBlock>>;
pub fn current_process() -> Arc<ProcessControlBlock>;
pub fn current_user_token() -> usize;
pub fn current_trap_cx() -> &'static mut TrapContext;
/// 当前线程Trap上下文在进程地址空间中的地址
pub fn current_trap_cx_user_va() -> usize;
/// 当前线程内核栈在内核地址空间中的地址
pub fn current_kstack_top() -> usize;
/// 将当前线程的内核态上下文保存指定位置, 并切换到调度主循环
pub fn schedule(swapped_task_cx_ptr: *mut TaskContext);
```

### 9.3.7 线程管理机制的设计与实现

在上述线程模型和内核数据结构的基础上, 我们还需完成线程管理的基本实现, 从而构造出一个完整的“达科塔盗龙”操作系统。这里将从如下几个角度分析如何实现线程管理:

- 线程生命周期管理
- 线程执行中的调度和特权级切换

#### 线程生命周期管理

线程生命周期管理包括线程从创建到退出的整个过程以及过程中的资源分配与回收。

#### 线程创建

线程创建有两种方式: 第一种是在创建进程的时候默认为这个进程创建一个主线程 (创建进程又分为若干种方式); 第二种是通过 `thread_create` 系统调用在当前进程内创建一个新的线程。

创建进程的第一种方式是调用 `ProcessControlBlock::new` 创建初始进程 `INITPROC`:

```
1 // os/src/task/process.rs
2
3 impl ProcessControlBlock {
4 pub fn new(elf_data: &[u8]) -> Arc<Self> {
5 // memory_set with elf program headers/trampoline/trap context/user stack
6 let (memory_set, ustack_base, entry_point) = MemorySet::from_elf(elf_data);
7 // allocate a pid
8 let pid_handle = pid_alloc();
9 // create PCB
10 let process = ...;
11 // create a main thread, we should allocate ustack and trap_cx here
12 let task = Arc::new(TaskControlBlock::new(
13 Arc::clone(&process),
```

(下页继续)

(续上页)

```

14 ustack_base,
15 true,
16);
17 // prepare trap_cx of main thread
18 let task_inner = task.inner_exclusive_access();
19 let trap_cx = task_inner.get_trap_cx();
20 let ustack_top = task_inner.res.as_ref().unwrap().ustack_top();
21 let kstack_top = task.kstack.get_top();
22 drop(task_inner);
23 *trap_cx = TrapContext::app_init_context(
24 entry_point,
25 ustack_top,
26 KERNEL_SPACE.exclusive_access().token(),
27 kstack_top,
28 trap_handler as usize,
29);
30 // add main thread to the process
31 let mut process_inner = process.inner_exclusive_access();
32 process_inner.tasks.push(Some(Arc::clone(&task)));
33 drop(process_inner);
34 insert_into_pid2process(process.getpid(), Arc::clone(&process));
35 // add main thread to scheduler
36 add_task(task);
37 process
38 }
39 }
40
41 // os/src/task/mod.rs
42
43 lazy_static! {
44 pub static ref INITPROC: Arc<ProcessControlBlock> = {
45 let inode = open_file("initproc", OpenFlags::RDONLY).unwrap();
46 let v = inode.read_all();
47 ProcessControlBlock::new(v.as_slice())
48 };
49 }

```

其中的要点在于：

- 第 10 和 12 行分别创建进程 PCB 和主线程的 TCB；
- 第 18~29 行获取所需的信息并填充主线程的 Trap 上下文；
- 第 32 行将主线程插入到进程的线程列表中。因为此时该列表为空，只需直接 push 即可；
- 第 34 行维护 PID-进程控制块映射。
- 第 36 行将主线程加入到任务管理器使得它可以被调度。

创建进程的第二种方式是 fork 出新进程：

```

1 // os/src/task/process.rs
2
3 impl ProcessControlBlock {
4 /// Only support processes with a single thread.
5 pub fn fork(self: &Arc<Self>) -> Arc<Self> {
6 let mut parent = self.inner_exclusive_access();
7 assert_eq!(parent.thread_count(), 1);
8 // clone parent's memory_set completely including trampoline/ustacks/trap_cxs

```

(下页继续)

(续上页)

```

9 let memory_set = MemorySet::from_existed_user(&parent.memory_set);
10 // alloc a pid
11 let pid = pid_alloc();
12 // copy fd table
13 let mut new_fd_table: Vec<Option<Arc<dyn File + Send + Sync>>> = Vec::new();
14 for fd in parent.fd_table.iter() {
15 ...
16 }
17 // create child process pcb
18 let child = ...;
19 // add child
20 parent.children.push(Arc::clone(&child));
21 // create main thread of child process
22 let task = Arc::new(TaskControlBlock::new(
23 Arc::clone(&child),
24 parent
25 .get_task(0)
26 .inner_exclusive_access()
27 .res
28 .as_ref()
29 .unwrap()
30 .ustack_base(),
31 // here we do not allocate trap_cx or ustack again
32 // but mention that we allocate a new kstack here
33 false,
34));
35 // attach task to child process
36 let mut child_inner = child.inner_exclusive_access();
37 child_inner.tasks.push(Some(Arc::clone(&task)));
38 drop(child_inner);
39 // modify kstack_top in trap_cx of this thread
40 let task_inner = task.inner_exclusive_access();
41 let trap_cx = task_inner.get_trap_cx();
42 trap_cx.kernel_sp = task.kstack.get_top();
43 drop(task_inner);
44 insert_into_pid2process(child.getpid(), Arc::clone(&child));
45 // add this thread to scheduler
46 add_task(task);
47 child
48 }
49 }
```

- 第 18 行创建子进程的 PCB，并在第 20 行将其加入到当前进程的子进程列表中。
- 第 22~34 行创建子进程的主线程控制块，注意它继承了父进程的 `ustack_base`，并且不用重新分配用户栈和 Trap 上下文。在第 37 行将主线程加入到子进程中。
- 子进程的主线程基本上继承父进程的主线程 Trap 上下文，但是其中的内核栈地址需要修改，见第 42 行。
- 第 44 行将子进程插入到 PID-进程控制块映射中。第 46 行将子进程的主线程加入到任务管理器中。

`exec` 也是进程模型中的重要操作，它虽然并不会创建新的进程但会替换进程的地址空间。在引入线程机制后，其实现也需要更新，但原理与前面介绍的类似，由于篇幅原因不再赘述，感兴趣的同学可自行了解。

第二种创建线程的方式是通过 `thread_create` 系统调用。重点是需要了解创建线程控制块，在线程控制块中初始化各个成员变量，建立好进程和线程的关系等。只有建立好这些成员变量，才能给线程建立一个灵活方便的执行环境。这里列出支持线程正确运行所必需的重要的执行环境要素：

- 线程的用户态栈：确保在用户态的线程能正常执行函数调用；
- 线程的内核态栈：确保线程陷入内核后能正常执行函数调用；
- 线程共享的跳板页和线程独占的 Trap 上下文：确保线程能正确的进行用户态与内核态间的切换；
- 线程的任务上下文：线程在内核态的寄存器信息，用于线程切换。

线程创建的具体实现如下：

```

1 // os/src/syscall/thread.rs
2
3 pub fn sys_thread_create(entry: usize, arg: usize) -> isize {
4 let task = current_task().unwrap();
5 let process = task.process.upgrade().unwrap();
6 // create a new thread
7 let new_task = Arc::new(TaskControlBlock::new(
8 Arc::clone(&process),
9 task.inner_exclusive_access().res.as_ref().unwrap().ustack_base,
10 true,
11));
12 // add new task to scheduler
13 add_task(Arc::clone(&new_task));
14 let new_task_inner = new_task.inner_exclusive_access();
15 let new_task_res = new_task_inner.res.as_ref().unwrap();
16 let new_task_tid = new_task_res.tid;
17 let mut process_inner = process.inner_exclusive_access();
18 // add new thread to current process
19 let tasks = &mut process_inner.tasks;
20 while tasks.len() < new_task_tid + 1 {
21 tasks.push(None);
22 }
23 tasks[new_task_tid] = Some(Arc::clone(&new_task));
24 let new_task_trap_cx = new_task_inner.get_trap_cx();
25 *new_task_trap_cx = TrapContext::app_init_context(
26 entry,
27 new_task_res.ustack_top(),
28 kernel_token(),
29 new_task.kstack.get_top(),
30 trap_handler as usize,
31);
32 (*new_task_trap_cx).x[10] = arg;
33 new_task_tid as isize
34}

```

上述代码主要完成了如下事务：

- 第 4~5 行，找到当前正在执行的线程 `task` 和此线程所属的进程 `process`。
- 第 7~11 行，调用 `TaskControlBlock::new` 方法，创建一个新的线程 `new_task`，在创建过程中，建立与进程 `process` 的所属关系，分配了线程资源组 `TaskUserRes` 和其他资源。
- 第 13 行，把线程挂到调度队列中。
- 第 19~22 行，把线程接入到所属进程的线程列表 `tasks` 中。
- 第 25~32 行，初始化位于该线程在用户态地址空间中的 Trap 上下文：设置线程的函数入口点和用户栈，使得第一次进入用户态时能从指定位置开始正确执行；设置好内核栈和陷入函数指针 `trap_handler`，保证在 Trap 的时候用户态的线程能正确进入内核态。

## 线程退出

线程可以通过 `sys_exit` 系统调用退出：

```
// os/src/syscall/process.rs

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

无论当前线程是否是主线程，都会调用 `exit_current_and_run_next`。如果是主线程，将会导致整个进程退出，从而其他线程也会退出；否则的话，只有当前线程会退出。下面是具体实现：

```
// 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 task_inner = task.inner_exclusive_access();
4 let process = task.process.upgrade().unwrap();
5 let tid = task_inner.res.as_ref().unwrap().tid;
6 // record exit code
7 task_inner.exit_code = Some(exit_code);
8 task_inner.res = None;
9 // here we do not remove the thread since we are still using the kstack
10 // it will be deallocated when sys_waittid is called
11 drop(task_inner);
12 drop(task);
13 // however, if this is the main thread of current process
14 // the process should terminate at once
15 if tid == 0 {
16 let pid = process.getpid();
17 ...
18 remove_from_pid2process(pid);
19 let mut process_inner = process.inner_exclusive_access();
20 // mark this process as a zombie process
21 process_inner.is_zombie = true;
22 // record exit code of main process
23 process_inner.exit_code = exit_code;
24
25 }
26 // move all child processes under init process
27 let mut initproc_inner = INITPROC.inner_exclusive_access();
28 for child in process_inner.children.iter() {
29 child.inner_exclusive_access().parent = Some(Arc::downgrade(&
30 INITPROC));
31 initproc_inner.children.push(child.clone());
32 }
33 }
34
35 // deallocate user res (including tid/trap_cx/ustack) of all threads
36 // it has to be done before we dealloc the whole memory_set
37 // otherwise they will be deallocated twice
38 let mut recycle_res = Vec::new();
39 for task in process_inner.tasks.iter().filter(|t| t.is_some()) {
40 let task = task.as_ref().unwrap();
41 // if other tasks are Ready in TaskManager or waiting for a timer to be
```

(下页继续)

(续上页)

```

43 // expired, we should remove them.
44 //
45 // Mention that we do not need to consider Mutex/Semaphore since they
46 // are limited in a single process. Therefore, the blocked tasks are
47 // removed when the PCB is deallocated.
48 remove_inactive_task(Arc::clone(&task));
49 let mut task_inner = task.inner_exclusive_access();
50 if let Some(res) = task_inner.res.take() {
51 recycle_res.push(res);
52 }
53 }
54 // deallocate_tid and deallocate_user_res require access to PCB inner, so we
55 // need to collect those user res first, then release process_inner
56 // for now to avoid deadlock/double borrow problem.
57 drop(process_inner);
58 recycle_res.clear();

59 let mut process_inner = process.inner_exclusive_access();
60 process_inner.children.clear();
61 // deallocate other data in user space i.e. program code/data section
62 process_inner.memory_set.recycle_data_pages();
63 // drop file descriptors
64 process_inner.fd_table.clear();
65 // Remove all tasks except for the main thread itself.
66 while process_inner.tasks.len() > 1 {
67 process_inner.tasks.pop();
68 }
69 }
70 drop(process);
71 // we do not have to save task context
72 let mut _unused = TaskContext::zero_init();
73 schedule(&mut _unused as *mut _);
74 }
75 }
```

- 第 4 行将当前线程从处理器管理结构 PROCESSOR 中移除，随后在第 9 行在线程控制块中记录退出码并在第 10 行回收当前线程的线程资源组 TaskUserRes。
- 第 17~68 行针对当前线程是所属进程主线程的情况退出整个进程和其他的所有线程（此时主线程已经在上一步中被移除）。其判断条件为第 17 行的当前线程 TID 是否为 0，这是主线程的特征。具体来说：
- 第 20~25 行更新 PID-进程控制块映射，将进程标记为僵尸进程然后记录进程退出码，**进程退出码即为其主线程退出码**。
- 第 29~33 行将子进程挂到初始进程 INITPROC 下面。
- 第 36~58 行回收所有线程的 TaskUserRes，为了保证进程控制块的独占访问，我们需要先将所有的线程的 TaskUserRes 收集到向量 recycle\_res 中。在第 57 行独占访问结束后，第 58 行通过清空 recycle\_res 自动回收所有的 TaskUserRes。
- 第 60~65 行依次清空子进程列表、回收进程地址空间中用于存放数据的物理页帧、清空文件描述符表。注意我们在回收物理页帧之前必须将 TaskUserRes 清空，不然相关物理页帧会被回收两次。目前这种回收顺序并不是最好的实现，同学可以想想看有没有更合适的实现。
- 第 66~69 行移除除了主线程之外的所有线程。目前线程控制块中，只有内核栈资源还未回收，但我们此时无法回收主线程的内核栈，因为当前的流程还是在这个内核栈上跑的，所以这里要绕过主线程。等到整个进程被父进程通过 waitpid 回收的时候，主线程的内核栈会被一起回收。

这里特别需要注意的是在第 48 行，主线程退出的时候可能有一些线程处于就绪状态等在任务管理器 TASK\_MANAGER 的队列中，我们需要及时调用 remove\_inactive\_task 函数将它们从队列中移除，不然

将导致它们的引用计数不能成功归零并回收资源，最终导致内存溢出。相关测例如 `early_exit.rs`，请同学思考我们的内核能否正确处理这种情况。

## 等待线程结束

如果调用 `sys_exit` 退出的不是进程的主线程，那么 `sys_exit` 之后该线程的资源并没有被完全回收，这一点和进程比较像。我们还需要另一个线程调用 `waittid` 系统调用才能收集该线程的退出码并彻底回收该线程的资源：

```

1 // os/src/syscall/thread.rs
2
3 /// thread does not exist, return -1
4 /// thread has not exited yet, return -2
5 /// otherwise, return thread's exit code
6 pub fn sys_waittid(tid: usize) -> i32 {
7 let task = current_task().unwrap();
8 let process = task.process.upgrade().unwrap();
9 let task_inner = task.inner_exclusive_access();
10 let mut process_inner = process.inner_exclusive_access();
11 // a thread cannot wait for itself
12 if task_inner.res.as_ref().unwrap().tid == tid {
13 return -1;
14 }
15 let mut exit_code: Option = None;
16 let waited_task = process_inner.tasks[tid].as_ref();
17 if let Some(waited_task) = waited_task {
18 if let Some(waited_exit_code) = waited_task.inner_exclusive_access().exit_
19 ↪code {
20 exit_code = Some(waited_exit_code);
21 }
22 } else {
23 // waited thread does not exist
24 return -1;
25 }
26 if let Some(exit_code) = exit_code {
27 // deallocate the exited thread
28 process_inner.tasks[tid] = None;
29 exit_code
30 } else {
31 // waited thread has not exited
32 -2
33 }
}

```

- 第 12~14 行，如果是线程等自己，返回错误。
- 第 17~24 行，如果找到 `tid` 对应的退出线程，则收集该退出线程的退出码 `exit_tid`，否则返回错误（退出线程不存在）。
- 第 25~32 行，如果退出码存在，则在第 27 行从进程的线程向量中将被等待的线程删除。这意味着该函数返回之后，被等待线程的 TCB 的引用计数将被归零从而相关资源被完全回收。否则，返回错误（线程还没退出）。

## 线程执行中的特权级切换和调度切换

在特权级切换方面，注意到在创建每个线程的时候，我们都正确设置了其用户态线程函数入口地址、用户栈、内核栈以及一些相关信息，于是第一次返回用户态之后能够按照我们的期望正确执行。后面在用户态和内核态间切换沿用的是前面的 Trap 上下文保存与恢复机制，本章并没有修改。有需要的同学可以回顾第二章和第四章的有关内容。

在线程切换方面，我们将任务上下文移至线程控制块中并依然沿用第三章的任务切换机制。同时，线程调度算法我们仍然采取第三章中时间片轮转的 Round-Robin 算法。

因此，这里我们就不再重复介绍这两种机制了。

## 9.4 互斥锁

### 9.4.1 本节导读

此前两节中，我们分别介绍了用户态和内核态的线程管理。相比进程模型，它们可以更加高效方便的进行协作。比如同进程下的线程共享进程的地址空间，它们可以直接通过读写内存中的共享变量来进行通信而不必进行繁琐且低效的进程间通信。然而，在方便的同时，这种做法也会产生一些问题。本节就让我们从一个简单的多线程计数的例子入手来看看我们会遇到什么问题以及如何解决。

### 9.4.2 引子：多线程计数器

我们知道，同进程下的线程共享进程的地址空间，因此它们均可以读写程序内的全局/静态数据。通过这种方式，线程可以非常方便的相互协作完成一项任务。下面是一个简单的例子，同学可以在 Linux/Windows 等系统上运行这段代码：

```

1 // adder.rs
2
3 static mut A: usize = 0;
4 const THREAD_COUNT: usize = 4;
5 const PER_THREAD: usize = 10000;
6 fn main() {
7 let mut v = Vec::new();
8 for _ in 0..THREAD_COUNT {
9 v.push(std::thread::spawn(|| {
10 unsafe {
11 for _ in 0..PER_THREAD {
12 A = A + 1;
13 }
14 }
15 }));
16 }
17 for handle in v {
18 handle.join().unwrap();
19 }
20 println!("{}", unsafe { A });
21 }
```

前一节中我们已经熟悉了多线程应用的编程方法。因此我们很容易看出这个程序开了 THREAD\_COUNT 个线程，每个线程都将一个全局变量 A 加 1，次数为 PER\_THREAD 次。从中可以看出多线程协作确实比较方便，因为我们只需将单线程上的代码（即第 11~13 行的主循环）提交给多个线程就从单线程得到了多线程版本。然而，这样确实能够达到我们预期的效果吗？

全局变量 A 的初始值为 0，而 THREAD\_COUNT 个线程每个将其加 1 重复 PER\_THREAD 次，因此当所有的线程均完成任务之后，我们预期 A 的值应该是二者的乘积即 40000。让我们尝试运行一下这个程序，可以看到类似下面的结果：

```
$ rustc adder.rs
$./adder
40000
$./adder
17444
$./adder
36364
$./adder
39552
$./adder
21397
```

可以看到只有其中一次的结果是正确的，其他的情况下结果都比较小且各不相同，这是为什么呢？我们可以尝试分析一下哪些因素会影响到代码的执行结果，使得结果与我们的预期不同。

1. 编译器在将源代码编译为汇编代码或者机器码的时候会进行一些优化。
2. 操作系统在执行程序的时候会进行调度。
3. CPU 在执行指令的时候会进行一些调度或优化。

那么按照顺序首先来检查第一步，即编译器生成的汇编代码是否正确。可以用如下命令反汇编可执行文件 adder 生成汇编代码 adder.asm：

```
$ rustup component add llvm-tools-preview
$ rust-objdump -D adder > adder.asm
```

在 adder.asm 中找到传给每个线程的闭包函数（这部分是我们自己写的，更容易出错）的汇编代码：

```
1 # adder.asm
2 000000000000bce0 <_ZN5adder4main28__u7b$$u7b$closure$u7d$$u7d$17hfcc06370a766a1c4E>:
3 bce0: subq $56, %rsp
4 bce4: movq $0, 8(%rsp)
5 bced: movq $10000, 16(%rsp) # imm = 0x2710
6 bcf6: movq 8(%rsp), %rdi
7 bcfb: movq 16(%rsp), %rsi
8 bd00: callq 0xb570 <_ZN63__LTIu20asu20$core..iter..traits..collect..
9 ↪IntoIteratorGT9into_iter17h0e9595229a318c79E>
10 bd05: movq %rax, 24(%rsp)
11 bd0a: movq %rdx, 32(%rsp)
12 bd0f: leaq 24(%rsp), %rdi
13 bd14: callq 0xb560 <_ZN4core4iter5range101__LT$impl$u20$core..iter..traits..
14 ↪iterator..Iterator$u20$for$u20$core..ops..range..RangeLTAGTGT
15 ↪$4next17h703752eeba5b7a01E>
16 bd19: movq %rdx, 48(%rsp)
17 bd1e: movq %rax, 40(%rsp)
18 bd23: cmpq $0, 40(%rsp)
19 bd29: jne 0xbd30 <_ZN5adder4main28__u7b$$u7b$closure$u7d$$u7d
20 ↪$17hfcc06370a766a1c4E+0x50>
21 bd2b: addq $56, %rsp
22 bd2f: retq
23 bd30: movq 328457(%rip), %rax # 0x5c040 <_ZN5adder1A17hce2f3c024bd1f707E>
24 bd37: addq $1, %rax
25 bd3b: movq %rax, (%rsp)
26 bd3f: setb %al
```

(下页继续)

(续上页)

```

23 bd42: testb $1, %al
24 bd44: jne 0xbd53 <_ZN5adder4main28__u7b$$u7b$closure$u7d$$u7d
25 ↳$17hfcc06370a766a1c4E+0x73>
26 bd46: movq (%rsp), %rax
27 bd4a: movq %rax, 328431(%rip) # 0x5c040 <_ZN5adder1A17hce2f3c024bd1f707E>
28 bd51: jmp 0xbd0f <_ZN5adder4main28__u7b$$u7b$closure$u7d$$u7d
29 ↳$17hfcc06370a766a1c4E+0x2f>
30 bd53: leaq 242854(%rip), %rdi # 0x47200 <str.0>
31 bd5a: leaq 315511(%rip), %rdx # 0x58dd8 <writev@GLIBC_2.2.5+0x58dd8>
32 bd61: leaq -15080(%rip), %rax # 0x8280 <_
33 ↳ZN4core9panicking5panic17h73f802489c27713bE>
34 bd68: movl $28, %rsi
35 bd6d: callq *%rax
36 bd6f: ud2
37 bd71: nopw %cs:(%rax,%rax)
38 bd7b: nopl (%rax,%rax)

```

虽然函数名经过了一些混淆，还是能看出这是程序 adder 的 main 函数中的一个闭包（Closure）。我们现在基于 x86\_64 而不是 RISC-V 架构，因此会有一些不同：

- 指令的目标寄存器后置而不是像 RISC-V 一样放在最前面；
- 使用 %rax, %rdx, %rsi, %rdi 作为 64 位通用寄存器，观察代码可以发现 %rsi 和 %rdi 用来传参，%rax 和 %rdx 用来保存返回值；
- %rsp 是 64 位栈指针，功能与 RISC-V 中的 sp 相同；
- %rip 是 64 位指令指针，指向当前指令的下一条指令的地址，等同于我们之前介绍的 PC 寄存器。
- callq 为函数调用，retq 则为函数返回。

在了解了这些知识之后，我们可以尝试读一读代码：

- 第 3 行是在分配栈帧；
- 第 4~8 行准备参数，并调用标准库实现的 `IntoIterator` trait 的 `into_iter` 方法将 `Range 0..10000` 转化为一个迭代器；
- 第 9 行的 24(%rsp) 应该保存的是生成的迭代器的地址；
- 第 11 行开始进入主循环。第 11 行加载 24(%rsp) 到 %rdi 作为参数并在第 12 行调用 `Iterator::next` 函数，返回值在 %rdx 和 %rax 中并被保存在栈上。我们知道 `Iterator::next` 返回的是一个 `Option<T>`。观察第 15~16 行，当 %rax 里面的值不为 0 的时候就跳转到 0xbd30，否则就向下执行到第 17~18 行回收栈帧并退出。这意味着 %rax 如果为 0 的话说明返回的是 `None`，这时迭代器已经用尽，就可以退出函数了。于是，主循环的次数为 10000 次就定下来了。
- 0xbd30（第 19 行）开始才真正进入 `A=A+1` 的部分。第 19 行从虚拟地址 0x5c040（这就是全局变量 A 的地址）加载一个 `usize` 到寄存器 %rax 中；第 20 行将 %rax 加一；第 26 行将寄存器 %rax 的值写回到虚拟地址 0x5c040 中。也就是说 `A=A+1` 是通过这三条指令达成。第 27 行无条件跳转到 0xbd0f 也就是第 11 行，进入下一轮循环。

#### 注解: Rust Tips: Rust 的无符号溢出是不可恢复错误

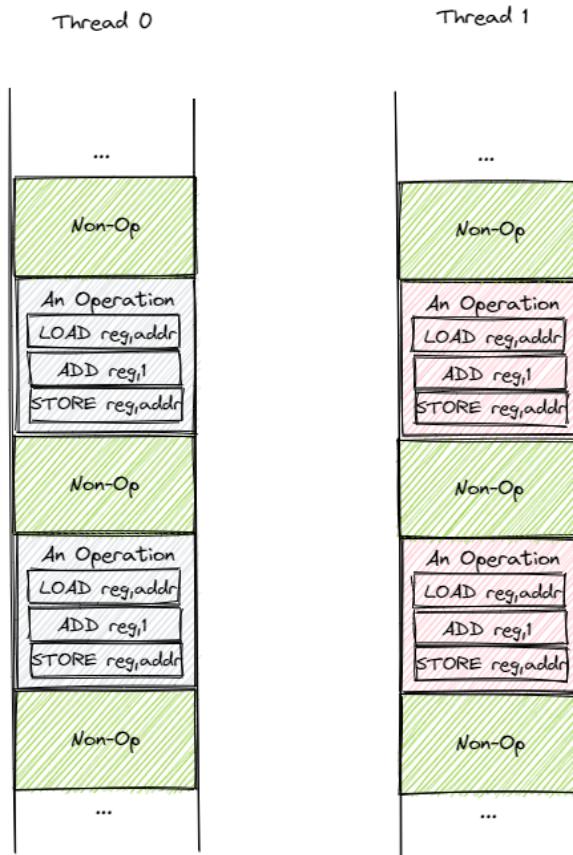
有兴趣的同学可以读一读第 21~24 行代码，它可以判断在将 %rax 加一的时候是否出现溢出（注意其中复用了 %rax，因此有一次额外的保存/恢复）。如果出现溢出的话则会跳转到 0xbd53（第 28 行）直接 panic。

从中我们可以看出，相比 C/C++ 来说 Rust 的确会生成更多的代码来针对算术溢出、数组越界的情况进行判断，但是这并不意味着在现代 CPU 上就会有很大的性能损失。如果可以确保不会出现溢出的情况，可以考虑使用 `unsafe` 的 `usize::unchecked_add` 来避免生成相关的判断代码并提高性能。

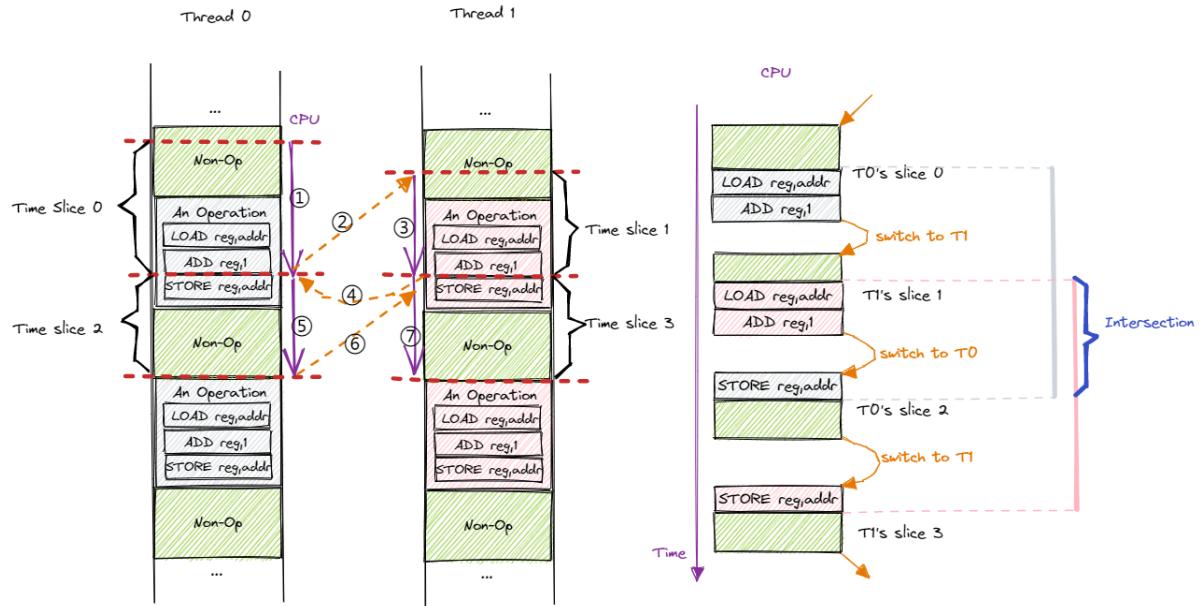
我们可以得出结论：编译器生成的汇编代码是符合我们的预期的。那么接下来进行第二步，操作系统的调度是否会影响结果的正确性呢？在具体分析之前，我们先对汇编代码进行简化，只保留直接与结果相关的部分。那么，可以看成每个线程进行 PER\_THREAD 次操作，每次操作按顺序进行下面三个步骤：

1. 使用访存指令，从全局变量 A 的地址 `addr` 加载 A 当前的值到寄存器 `reg`；
2. 使用算术指令将寄存器 `reg` 的值加一；
3. 使用访存指令，将 `reg` 的值写回到全局变量 A 的地址 `addr`，至此 A 的值成功加一。

这是一个可以认为与具体指令集架构无关的过程。因为对于传统的计算机架构而言，在 ALU 上进行的算术指令需要以寄存器为载体，而不能直接在 RAM 上进行操作。在此基础上，我们可以建立简化版的线程执行模型，如下图所示：



目前有两个线程 T0 和 T1，二者都是从上到下顺序执行。我们将  $A=A+1$  的操作打包成包含三条指令的一个块，剩下的绿色区域则表示与操作无关的那些指令。每个线程都会有一种幻觉就是它能够从头到尾独占 CPU 执行，但实际上操作系统会通过抢占式调度划分时间片使它们交错 (Interleave) 运行。注意时钟中断可能在执行任意一条指令之后触发，因此时间片之间的边界可能是任意一条指令。下图是一种可能的时间片划分方式：



我们暂时只考虑单 CPU 的简单情况。按照时间顺序，CPU 依次执行 T0 的时间片 0、T1 的时间片 1、T0 的时间片 2 和 T1 的时间片 3，在相邻两个时间片之间会进行一次线程切换。注意到在这种划分方式中，两个线程各有一个操作块被划分到多个时间片完成。图片的右侧展示了 CPU 视角的指令执行过程，我们仅关注操作块中的指令，并尝试模拟一下：

| 动作              | 所属线程 | 寄存器 reg 的值（动作后） | addr 处的值（动作后） |
|-----------------|------|-----------------|---------------|
| 切换到 T0          | T0   | •               | v             |
| LOAD reg, addr  | T0   | v               | v             |
| ADD reg, 1      | T0   | v+1             | v             |
| T0 切换到 T1       | T1   | •               | v             |
| LOAD reg, addr  | T1   | v               | v             |
| ADD reg, 1      | T1   | v+1             | v             |
| T1 切换到 T0       | T0   | v+1             | v             |
| STORE reg, addr | T0   | v+1             | v+1           |
| T0 切换到 T1       | T1   | v+1             | v+1           |
| STORE reg, addr | T1   | v+1             | v+1           |
| T1 切换出去         | •    | •               | v+1           |

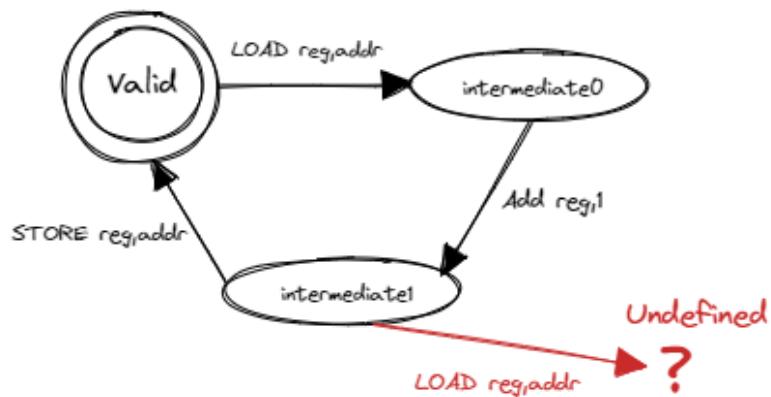
假设开始之前全局变量 A 的值为 v，而在这来自两个线程的四个时间片中包含了完整的两个  $A = A + 1$  的操作块，那么结束之后 A 的值应该变成  $v + 2$ 。然而我们模拟下来的结果却是  $v + 1$ ，这是为什么呢？首先需要说明的是，尽管两个线程都使用寄存器 reg 中转，但是它们之间并不会产生冲突，因为在线程切换的时候会对线程上下文进行保存与恢复，其中包括寄存器 reg。因此我们可以认为两个线程均有一份自己独占的寄存器。言归正传，我们从结果入手进行分析，A 最终的值来源于我们在这段时间对它进行的最后一次写入，这次写入由 T1 进行，但是为什么 T1 会写入  $v + 1$  而不是  $v + 2$  呢？从 T1 的视角来看，首先要读取 A 的值到 reg，发现是 v，这一点就很奇怪，好像此前 T0 什么都没做一样。而后 T1 将 reg 的值加一变成  $v + 1$ ，于是最后写入的也是这个值。所以，问题的关键在于 T0 将自己的 reg 更新为  $v + 1$  之后，还没来得及写回到 A，就被操作系统切换到 T1，因此 T1 会看到 v 而不是  $v + 1$ 。等再切换回 T0 将  $v + 1$  写入到 A 的时候已经为时已晚，因为已经过了关键的 T1 读取 A 的时间点，于是这次写入无法对 T1 产生任何影响，也无法影响到最终的结果了。因此，在这种情况下，由于操作系统的抢占式调度，可以看到 T0 的  $A = A + 1$  操作完全在做无用功，于是

最终结果比期望少 1。

从上个例子可以看出，操作系统的调度有可能使得两个线程上的操作块 交错出现，也就是说两个操作块从开始到结束的时间区间存在交集。一旦出现这种情况，便会导致结果出现偏差。最终的结果取决于这种交错的情况出现多少次，如果完全没有出现则结果正确；否则出现次数越多，结果偏差越大。这就能够解释为什么我们每次运行 adder.rs 会得到不同的结果。这种运行结果 不确定 (Indeterminate)，且取决于像是操作系统的调度顺序这种无法控制的外部事件的情况被称为 竞态条件 (Race Condition)。在 adder.rs 中，竞态条件导致了我们预料之外的结果，因此它应当被认为是一个 bug。

我们尝试更加形象的说明为什么操作块交错出现就会有问题。在写程序的时候，我们需要做的是通过软件控制一些资源，这些资源可能是软件资源或者硬件资源。软件资源可能包括保存在内存中的一些数据结构，硬件资源可能是内存的一部分或者某些 I/O 设备。在资源被初始化之后，资源处于一种合法 (Valid) 状态，这里的合法状态是指资源符合一些特定的约束条件从而具有该种资源所应该具有的特征。以我们耳熟能详的链表数据结构为例，一个合法的链表应该满足每个节点的 next 指针均为空指针或者指向合法的内存区域。同时，next 指针不能形成环。当然，实际上还有更多的约束条件，我们使用自然语言很难完全表述它们。总之，只有满足所有的约束条件，我们才说这是一个合法的链表。

每种资源可能都有多种不同的控制方式，每种控制方式称为对这种资源的一种操作。比如说，如果将链表看成一种资源，那么链表的插入和删除就是两种对链表的操作。每一种操作仅在资源处于合法状态时才能进行，且在操作完成之后保证资源仍旧处于合法状态。设想我们要实现链表的插入操作，这必须在待操作的数据结构是一个合法的链表这一前提下才能进行，不然我们的操作将完全没有意义。我们还需要保证插入之后链表依然合法，才称得上是正确的实现。但是资源并非任意时刻均处于合法状态。因为一般来说操作都比较复杂，会分成多个阶段多条指令完成。通常，处于合法状态的资源在操作时会变成不合法的 中间状态 (Intermediate State)，待操作结束之后再重回合法状态。以我们的多线程计数器 adder.rs 为例，状态转移过程如下：



这里我们将全局变量 A 视为一种资源，操作  $A = A + 1$  为一个三阶段操作。我们可以用有限状态自动机来描述资源 A 和操作  $A = A + 1$ ：状态机中一共有 3 种状态，一个合法状态和两个不合法的中间状态 0 和 1。对于每次操作，第一条指令 A 从合法状态转移到中间状态 0；第二条指令 A 从中间状态 0 转移到中间状态 1；第三条指令 A 从中间状态 1 转移回合法状态。将操作块交错的情况代入到状态机中，最开始切换到 T0 之前 A 处于合法状态，接下来切换到 T0 执行了第一、二条指令之后 A 转移到中间状态 1，而此时操作系统切换到 T1，T1 又开始执行第一条指令。问题来了：我们发现中间状态 1 并没有定义此时再执行第一条指令应该如何转移。如果去执行的话，就会产生未定义行为并可能永远无法使 A 回到合法状态。不过，由于 adder.rs 中 A 只是一个整数，我们会发现 A 仍能回到合法状态，只是结果不对。如果换成一种复杂的数据结构，就会产生极其微妙且难以调试的结果。

我们可以发现多线程对共享资源的访问天然需求某种互斥性：当一个线程在对共享资源进行操作的时候，共享资源处在不合法的中间状态，如果此时其他线程开始操作会产生未定义行为。只有当操作完成，共享资源重新回到合法状态之后，之前操作的线程或者其他线程才能开始下一次操作。只有满足这种互斥性，才能保证多线程对共享资源的访问符合我们的预期。下面，我们换用操作系统中的术语进行表述：

**共享资源** (Shared Resources) 是指多个线程均能够访问的资源。线程对于共享资源进行操作的那部分代码被称为 **临界区** (CriticalSection)。在多线程并发访问某种共享资源的时候，为了正确性，必须要满足 **互斥** (Mutual

Exclusion) 访问要求，即同一时间最多只能有一个线程在这种共享资源的临界区之内。这样才能保证当一个线程开始操作时，共享资源总是处于合法状态，这保证了操作是有意义的。如果能够做到互斥访问的话，我们 adder.rs 出现 bug 的根源——即对于 A 的操作可能交错出现的情况便能够被避免。

从 adder.rs 中可以看出，如果任由操作系统进行时间片切分和线程调度而不加任何特殊处理，是很难满足互斥访问要求的。那么应该如何实现互斥访问呢？接下来，我们将会尝试构建一组称之为 **互斥锁** (Mutex，源于 **Mut**ual **Ex** clusion，简称为 **锁** Lock) 的通用互斥原语来对临界区进行保护，从而在一般意义上保证互斥访问要求。这将是本节接下来的主要内容。

如果仅仅考虑 adder.rs 的话，其实不借助锁机制也能够解决问题。这是因为其中的共享资源为一个 64 位无符号整型，是一个十分简单的类型。对于这种原生类型，现代指令集架构额外提供一组 **原子指令** (Atomic Instruction)，在某些架构上只需一条原子指令就能完成包括访存、算术运算在内的一系列功能。这就是说 adder.rs 中的  $A = A + 1$  操作其实只需一条原子指令就能完成。如果这样做的话，我们相当于 **将临界区缩小为一条原子指令**，这已经是处理器执行指令和时间片切分的最小单位，因此我们不使用任何保护手段也能满足互斥要求。修改之后的代码如下：

```

1 // adder_fixed.rs
2
3 use std::sync::atomic::{AtomicUsize, Ordering};
4 static A: AtomicUsize = AtomicUsize::new(0);
5 const THREAD_COUNT: usize = 4;
6 const PER_THREAD: usize = 10000;
7 fn main() {
8 let mut v = Vec::new();
9 for _ in 0..THREAD_COUNT {
10 v.push(std::thread::spawn(|| {
11 for _ in 0..PER_THREAD {
12 A.fetch_add(1, Ordering::Relaxed);
13 }
14 }));
15 }
16 for handle in v {
17 handle.join().unwrap();
18 }
19 println!("{}", A.load(Ordering::Relaxed));
20 }
```

Rust 核心库在 `core::sync::atomic` 中提供了很多原子类型，比如我们这里可以使用 `usize` 对应的原子类型 `AtomicUsize`，它支持很多原子操作。比如，第 12 行 `fetch_add` 的功能是将 A 的值加一并返回 A 之前的值，这其中涉及到读取内存、算术运算和写回内存，但是却只需要这一个操作就能同时完成。这种原子操作基于硬件提供的原子指令，硬件可以保证其 **原子性** (Atomicity)，含义是该操作的一系列功能要么全部完成，要么都不完成，而不会出现有些完成有些未完成的情况。原子性中的“原子”是为了强调操作中的各种功能作为一个整体不可分割的属性。这种由硬件提供的 **原子指令是整个计算机系统中最根本的原子性和互斥性的来源**。无论软件执行了哪些指令，也无论 CPU 执行指令的时候出现了哪些中断/异常，又或者多个 CPU 同时访问内存中同一个位置这种情形，都不能破坏原子指令的原子性。

可惜的是，原子指令虽然强大，其应用范围却比较有限，通常它只能用来保护 **单内存位置** 上的简单操作，比如  $A = A + 1$  这种操作。当资源是比较复杂的数据结构的时候它就无能为力了。当然，我们也不会指望硬件提供一条“原子地完成红黑树插入/删除”这种指令，毕竟这样的数据结构有无数种，硬件总不可能对每种可能的数据结构和每种可能的操作都提供一条指令，这样的硬件是不存在的。即使如此我们也没有必要担心，只要我们能够灵活使用原子指令来根据实际需求限制多线程对共享资源的并发访问，比如基于原子指令实现通用的锁机制来保证互斥访问，所有的并发访问问题就一定能够迎刃而解。

需要注意的是，adder.rs 的错误结果是多种因素共同导致的，这里我们深入分析的操作系统调度带来的影响只不过是其中之一，其实 CPU 的指令执行也会有影响。

## 注解：原子性

原子性的说法最常见于数据库领域的原子 **事务** (Transaction)，表示对于数据库的一次修改要么全部完成，要么没有任何影响，而不存在任何中间状态。事务模型可以保证数据库能够在出错时顺利恢复以及高并发访问的正确性。但其实原子性这个概念在很多不同领域均有应用且有不同内涵。

### 注解: Rust Tips: static mut 和 unsafe 的消失

TODO:

## 9.4.3 锁的简介

### 锁机制的形态与功能

我们提到为了保证多线程能够正确并发访问共享资源，可以使用一种叫做 **锁** 的通用机制来对线程操作共享资源的 **临界区** 进行保护。这里的锁和现实生活中的含义很接近。回想一下我们如何使用常见于理发店或者游泳馆更衣室的公共储物柜：首先需要找到一个没有上锁的柜子并将物品存放进去。接着我们锁上柜子并拔出插在锁孔上的钥匙妥善保管。最后，当我们想取出物品时，我们使用钥匙打开存放物品的柜子并将钥匙留在锁孔上以便他人使用。至此，完整的使用流程结束。

那么，如何使用类似的思路用锁机制保护临界区呢？锁是附加在一种共享资源上的一种标记，最简单的情况下它只需有两种状态：上锁和空闲。上锁状态表示此时已经有某个线程在该种共享资源的临界区中，故而为了正确性其他线程不能进入临界区。相反的，空闲状态则表示线程可以进入临界区。显然，线程成功进入临界区之后锁也需要从空闲转为上锁状态。锁的两个基本操作是 **上锁** 和 **解锁**，在线程进入临界区之前和退出临界区之后分别需要成功上锁和解锁。通过这种方式，我们就可以保证临界区的互斥性。在引入锁机制之后，线程访问共享资源的流程如下：

- 第一步上锁：线程进入临界区之前检查共享资源是否已经上锁。如果已经上锁的话，则需要 **等待** 持有钥匙的线程归还钥匙并解锁。接下来，线程尝试“抢”到钥匙，如果成功的话，线程将资源上锁，此时我们说该线程 **获取到了锁**（或者说 **持有锁或拿到了锁**）。最后线程拿走钥匙并进入临界区。此时资源进入上锁状态，其他线程不能进入临界区。
- 第二步在临界区内访问共享资源。只有持有共享资源锁的线程能够进入临界区，这就能够保证临界区的互斥性。
- 第三步解锁：线程离开临界区之后将资源解锁并归还钥匙，我们说线程 **释放了锁**。此时资源回到空闲状态。

### 锁的使用方法

Rust 在标准库中提供了互斥锁 `std::sync::Mutex<T>`，它可以包裹一个类型为 `T` 的共享资源为它提供互斥访问。线程可以调用 `Mutex<T>::lock` 来获取锁，注意线程不一定立即就能拿到锁，所以它会等待持有锁的线程释放锁且自身抢到锁之后才会返回。其返回值为 `std::sync::MutexGuard<T>`（篇幅所限省略掉外层的 `Result`），可以理解为前面描述中的一把钥匙，拿到它的线程也就拿到了锁，于是有资格独占共享资源并进入临界区。`MutexGuard<T>` 提供内部可变性，可以看做可变引用 `&mut T`，用来修改共享资源。它的另一种功能是用来开锁，它也是 RAII 风格的，在它被 `drop` 之后会将锁自动释放。

让我们看看如何使用 `Mutex<T>` 来更正 `adder.rs`：

```

1 // adder_mutex0.rs
2
3 use std::sync::Mutex;
4 const THREAD_COUNT: usize = 4;
5 const PER_THREAD: usize = 10000;

```

(下页继续)

(续上页)

```

6 static A: Mutex<usize> = Mutex::new(0);
7 fn main() {
8 let mut v = Vec::new();
9 for _ in 0..THREAD_COUNT {
10 v.push(std::thread::spawn(|| {
11 for _ in 0..PER_THREAD {
12 let mut a_guard = A.lock().unwrap();
13 *a_guard = *a_guard + 1;
14 }
15 }));
16 }
17 for handle in v {
18 handle.join().unwrap();
19 }
20 println!("{}", *A.lock().unwrap());
21 }

```

第 6 行我们将共享资源用 A 使用 Mutex<T> 包裹。第 12~14 行构成一次完整的受锁保护的临界区访问：第 12 行获取锁；第 13 行是临界区；第 14 行循环的一次迭代结束，第 12 行的 MutexGuard<T> 退出作用域，于是它被 drop 之后自动解锁。

在上面的做法中，锁以及被锁保护的共享资源被整合到一个数据结构中，这也是最为常见的做法。但在某些情况下，它们之间可以相互分离，参考下面的代码：

```

1 // adder_mutex1.rs
2
3 use std::sync::Mutex;
4 static mut A: usize = 0;
5 static LOCK: Mutex<bool> = Mutex::new(true);
6 const THREAD_COUNT: usize = 4;
7 const PER_THREAD: usize = 10000;
8 fn main() {
9 let mut v = Vec::new();
10 for _ in 0..THREAD_COUNT {
11 v.push(std::thread::spawn(|| {
12 for _ in 0..PER_THREAD {
13 let _lock = LOCK.lock();
14 unsafe { A = A + 1; }
15 }
16 }));
17 }
18 for handle in v {
19 handle.join().unwrap();
20 }
21 println!("{}", unsafe { A });
22 }

```

其中锁 LOCK 用来保护共享资源 A。此处，LOCK 有用的仅有那个描述锁状态（可能是上锁或空闲）的标记，它内部包裹的值反而无关紧要，其类型 T 可以随意选择。可以看到在这种实现中，锁 LOCK 和共享资源 A 是分离开的，这样实现更加灵活，但是更容易由于编码错误而出现 bug。

## 评价锁实现的指标

锁机制有多种不同的实现。对于一种实现而言，我们常常用以下的指标来从多个维度评估这种实现是否能够正确、高效地达成锁这种互斥原语应有的功能：

- **忙则等待**：意思是当一个线程持有了共享资源的锁，此时资源处于繁忙状态，这个时候其他线程必须等待拿着锁的线程将锁释放后才有进入临界区的机会。这其实就是互斥访问的另一种说法。这种互斥性是锁实现中最重要的也是必须做到的目标，不然共享资源访问的正确性会受到影响。
- **空闲则入**（在《操作系统概念》一书中也被称为 **前进** Progress）：若资源处于空闲状态且有若干线程尝试进入临界区，那么一定能够在有限时间内从这些线程中选出一个进入临界区。如果不满足空闲则入的话，可能导致即使资源空闲也没有线程能够进入临界区，对于锁来说是不可接受的。
- **有界等待**（Bounded Waiting）：当线程获取锁失败的时候首先需要等待锁被释放，但这并不意味着此后它能够立即抢到被释放的锁，因为此时可能还有其他的线程也处于等待状态。于是它可能需要等待一轮、二轮、多轮才能拿到锁，甚至在极端情况下永远拿不到锁。有界等待要求每个线程在等待有限长时间后最终总能够拿到锁。相对的，线程可能永远无法拿到锁的情况被称之为 **饥饿** (Starvation)。这体现了锁实现分配共享资源给线程的 **公平性** (Fairness)。
- **让权等待**（可选）：线程如何进行等待实际上也大有学问。这里所说的让权等待是指需要等待的线程暂时主动或被动交出 CPU 使用权来让 CPU 做一些有意义的事情，这通常需要操作系统的支持。这样可以提升系统的总体效率。

总的来说，忙则等待、空闲则入和有界等待是一个合格的锁实现必须满足的要求，而让权等待则关系到锁机制的效率，是可选的。

这一小节我们介绍了锁的形态：一种附加在共享资源上的标记，需要区分当前是否有线程在该种资源的临界区中。它支持两种基本操作：上锁和解锁。接着我们还介绍了 Rust 标准库提供的互斥锁 Mutex<T> 并通过例子演示了它的用法。最后我们介绍了评价锁机制实现的一些指标，从中我们可以了解到怎样才可以称之为一个好的锁实现。接下来，我们将正式开始亲自动手根据上述需求尝试实现锁机制。

### 9.4.4 锁的纯用户态软件实现

在本节中，我们仅考虑在用户态使用锁机制保证多线程对共享资源的互斥访问的情形。此时在实现锁机制的时候，应用的执行环境的多个部分都有可能给我们带来一些帮助，从上到下它们分别是：

1. 用户态软件的一些函数库；
2. 操作系统内核的支持；
3. 相关硬件机制或特殊指令的支持。

为了简单起见，我们暂不考虑操作系统内核支持和硬件机制，看看能否仅基于用户态软件实现锁机制，如果无法实现或者实现得不好是由于哪些问题，应当如何改进。本节之前的例子 adder.rs 运行在我们的真实计算机上的 Linux/Windows 或其他操作系统上，不过当我们自己实现锁的时候，我们选择在一个比较简单的平台——即 Qemu 模拟出来的 **单核** 计算机上的我们自己实现支持多线程的“达科塔盗龙”操作系统上进行实现和测试。因为如果基于真实计算机和成熟的操作系统，多线程在 **多核** 上的执行模型比较复杂，为了正确实现需要了解很多相关知识。在 Qemu 上我们能最大限度简化问题，也同样能有机会讨论锁实现的一些核心问题。

基于 **内核态** 的线程管理，我们可以在自己的“达科塔盗龙”操作系统上实现 adder.rs。代码如下：

```

1 // user/src/bin/adder.rs
2
3 #![no_std]
4 #![no_main]
5
6 #[macro_use]
7 extern crate user_lib;

```

（下页继续）

(续上页)

```

8 extern crate alloc;
9
10 use alloc::vec::Vec;
11 use core::ptr::addr_of_mut;
12 use user_lib::{exit, get_time, thread_create, waittid};
13
14 static mut A: usize = 0;
15 const PER_THREAD_DEFAULT: usize = 10000;
16 const THREAD_COUNT_DEFAULT: usize = 16;
17 static mut PER_THREAD: usize = 0;
18
19 unsafe fn critical_section(t: &mut usize) {
20 let a = addr_of_mut!(A);
21 let cur = a.read_volatile();
22 for _ in 0..500 {
23 *t = (*t) * (*t) % 10007;
24 }
25 a.write_volatile(cur + 1);
26 }
27
28 unsafe fn f() -> ! {
29 let mut t = 2usize;
30 for _ in 0..PER_THREAD {
31 critical_section(&mut t);
32 }
33 exit(t as i32)
34 }
35
36 #[no_mangle]
37 pub fn main(argc: usize, argv: &[&str]) -> i32 {
38 let mut thread_count = THREAD_COUNT_DEFAULT;
39 let mut per_thread = PER_THREAD_DEFAULT;
40 if argc >= 2 {
41 thread_count = argv[1].parse().unwrap();
42 if argc >= 3 {
43 per_thread = argv[2].parse().unwrap();
44 }
45 }
46 unsafe { PER_THREAD = per_thread; }
47 let start = get_time();
48 let mut v = Vec::new();
49 for _ in 0..thread_count {
50 v.push(thread_create(f as usize, 0) as usize);
51 }
52 for tid in v.into_iter() {
53 waittid(tid);
54 }
55 println!("time cost is {}ms", get_time() - start);
56 assert_eq!(unsafe { A }, unsafe { PER_THREAD } * thread_count);
57 0
58 }

```

这里共享资源仍然为全局变量 A，具体操作为开 thread\_count 个线程，每个线程执行 A=A+1 操作 PER\_THREAD 次。线程数和每个线程上的操作次数默认值分别由 THREAD\_COUNT\_DEFAULT 和 PER\_THREAD\_DEFAULT 给出，也可以通过命令行参数设置。例如，命令 adder 4 1000 可以调整为 4 个线程，每个线程执行 1000 次操作，详情参考第 37~45 行的命令行参数逻辑。

每个线程在创建之后都会不带参数执行第 27 行的 `f` 函数，里面是一个循环。循环的每次迭代都会尝试进入临界区（第 18 行的 `critical_section` 函数），共迭代 `PER_THREAD` 次。临界区其实就是进行 `A=A+1` 的操作，但有两点不同：第一点是我们使用 `core::ptr::read/write_volatile` 而不是直接 `A=A+1`，这是为了生成的汇编代码严格遵循 `A=A+1` 的三阶段而不会被编译器误优化，其实有些类似于手写汇编；第二点是我们插入了一些关于 `t` 的冗余操作，这并不影响到共享资源的访问，目的在于加大阶段之间的间隔，使得一次 `A=A+1` 操作更容易横跨两个时间片从而出现一些有趣的现象。

### 注解：Rust Tips: 易失性读写 `read/write_volatile`

有些时候，编译器会对一些访存行为进行优化。举例来说，如果我们写入一个内存位置并立即读取该位置，并且在同段时间内其他线程不会访问该内存位置，这意味着我们写入的值能够在 RAM 上保持不变。那么，编译器可能会认为读取到的值必定是此前写入的值，于是在最终的汇编码中读取内存的操作可能被优化掉。然而，有些时候，特别是访问 I/O 外设以 MMIO 方式映射的设备寄存器时，即使是相同的内存位置，对它进行读取和写入的含义可能完全不同，于是读取到的值和我们之前写入的值可能没有任何关系。连续两次读取同个设备寄存器也可能得到不同的结果。这种情况下，编译器对访存行为的修改显然是一种误优化。

于是，在访问 I/O 设备寄存器或是与 RAM 特性不同的内存区域时，就要注意通过 `read/write_volatile` 来确保编译器完全按照我们的源代码生成汇编代码而不会自作主张进行删除或者重排访存操作等优化。若想更加深入了解 `volatile` 的含义，可以参考 [Rust 官方文档](#)。

在 `user/src/bin` 目录下我们还会看到很多 `adder.rs` 衍生出来的 `adder_*.rs` 测例，每个测例对应到一种锁实现。它们与 `adder.rs` 的主要不同在于临界区看起来是受保护的：在临界区 `critical_section` 的前后分别会调用上锁和解锁函数 `lock` 和 `unlock`，每个测例的 `lock` 和 `unlock` 的实现也是不同的。然而它们不全是正确的实现，效率也各不相同。

### 单标记的简单尝试

我们知道锁的本质上是一个标记，表明目前是否已经有线程进入共享资源的临界区了。于是，最简单的实现思路就是加入一个新的全局变量用作这个标记：

```

1 // user/src/bin/adder_simple_spin.rs
2
3 static mut OCCUPIED: bool = false;
4
5 unsafe fn lock() {
6 while vload!(OCCUPIED) {}
7 OCCUPIED = true;
8 }
9
10 unsafe fn unlock() {
11 OCCUPIED = false;
12 }
```

我们使用一个新的全局变量 `OCCUPIED` 作为标记，表示当前是否有线程在临界区内。在 `lock` 的时候，我们等待 `OCCUPIED` 变为 `false`（注意这里的 `vload!` 来自用户库 `user/lib`，基于临界区中用到的 `read_volatile` 实现，含义相同），这意味着没有线程在临界区内了，于是将标记修改为 `true` 并自己进入临界区。在退出临界区 `unlock` 的时候则只需将标记改成 `false`。

第 6 行不断 `while` 循环直到标记被改为 `false`，在循环体内则不做任何事情，这是一种典型的 **忙等待** (Busy Waiting) 策略，它也被形象地称为 **自旋** (Spinning)。我们目前基于单核 CPU，如果循环第一次迭代发现标记为 `true` 的话，在触发时钟中断切换到其他线程之前，无论多少次查看标记都必定为 `true`，因为当前线程不会修改标记。这就会造成 CPU 资源的严重浪费。针对这种场景，Rust 提供了 `spin_loop_hint` 函数，我们可以在循环体内调用该函数来通知 CPU 当前线程正处于忙等待状态，于是 CPU 可能会进行一些优化（比如降频减少功耗等），其在不同平台上有所不同。此外，如果我们有操作系统支持的话，便可以考虑锁实现评价指标中的“让权等待”，这个我们后面还会讲到。

可以看到，总体上这种实现是非常简单的，但是它能够保证最关键的互斥访问吗？我们可以尝试运行一下测例，很遗憾，最后的结果并不正确！那么是哪里出了问题呢？根据原先的思路我们还是先检查 lock 的汇编代码，我们将其简化为伪代码的形式（有兴趣的同学可以自行尝试），大致上分为三个阶段，每个阶段由一到多条指令组成：

1. 将标记的值加载到寄存器 reg
2. 条件跳转，如果 reg 为 1 则跳转回第一阶段开始新一轮循环；否则向下进行
3. 将标记赋值为 1

仿照最早的 adder.rs 的例子，我们很容易构造出一种时间片分割的方式使得互斥访问失效：假设某时刻标记 OCCUPIED 的值为 false，线程 T0 和 T1 都准备进入临界区。假设先切换到 T0，它经历 1、2 阶段，看到标记为 false，认为自己能够进入临界区，但是在执行关键的 3 阶段之前被操作系统切换到线程 T1。T1 也经历 1、2 阶段，由于 T0 并没有修改标记，它也认为自己能够进入临界区。接下来显然线程 T0 和 T1 能够同时进入临界区了，这就违背了互斥访问要求。

问题的本质是：在这个实现中，标记 OCCUPIED 也成为了多线程均可访问的 **共享资源**，那么 **它也需要互斥访问**。而我们并没有吸取 adder.rs 的教训，我们让操作为多阶段多指令的 OCCUPIED 无任何保护的暴露在操作系统调度面前，那么自然也会发生和 adder.rs 类似的问题。那么应该如何解决问题呢？参照 adder\_fixed.rs，在硬件的支持下，将对标记的操作替换为原子操作显然很靠谱。但我们不禁要问，如果不依赖硬件，是否有一种纯软件的解决方案呢？其实在某些限制条件下是可以的，下面就介绍一个例子。

## 多标记的组合

既然仅使用一个标记不行，我们尝试使用多标记的组合来表示锁的状态。比如下面的 Peterson 算法就适用于两个线程间的互斥访问：

```

1 // user/src/bin/adder_peterson_spin.rs
2
3 // FLAG[i]=true 表示线程 i 想要进入或已经进入临界区
4 static mut FLAG: [bool; 2] = [false; 2];
5 // TURN=i 表示轮到线程 i 进入临界区
6 static mut TURN: usize = 0;
7
8 // id 表示当前的线程 ID，为 0 或 1
9 unsafe fn lock(id: usize) {
10 FLAG[id] = true;
11 let j = 1 - id;
12 TURN = j;
13 // Tell the compiler not to reorder memory operations
14 // across this fence.
15 compiler_fence(Ordering::SeqCst);
16 // Why do we need to use volatile_read here?
17 // Otherwise the compiler will assume that they will never
18 // be changed on this thread. Thus, they will be accessed
19 // only once!
20 while vload!(FLAG[j]) && vload!(TURN) == j {} // while FLAG[j] && TURN == j {}
21
22 }
23
24 unsafe fn unlock(id: usize) {
25 FLAG[id] = false;
26 }
```

目前一共有三个标记：两个线程  $T_0$  和  $T_1$  各自有一个标记  $flag_i, i \in \{0, 1\}$ ，表示线程  $T_i$  想要或已经进入临界区。此外还有一个标记  $turn$ ，当  $turn$  为  $i$  的时候表示轮到线程  $T_i$  进入临界区。

我们来看看 `lock` 和 `unlock` 各自做了哪些事情（假设调用者为线程  $T_i$ ）。在 `lock` 中，首先我们将  $\text{flag}_i$  设置为 `true`，表明线程  $T_i$  想要进入临界区。接着得到另一个线程的编号  $j = 1 - i$ 。第 12 行非常有趣：线程  $T_i$  将标记 `turn` 设置为  $j$ ，由于 `turn` 的含义是有资格进入临界区的线程 ID，这相当于  $T_i$  将进入临界区的资格拱手让给  $T_j$ 。第 15 行我们可以先忽略。最后是第 20 行的忙等（为了防止编译器优化我们使用了 `vload!`，逻辑同第 21 行）：如果  $T_j$  想要或已经进入临界区而且轮到  $T_j$  进入临界区就一直等待下去。`unlock` 则只是将  $\text{flag}_i$  设置为 `false`。

运行一下这个测例发现总是可以得到正确的结果，说明在操作系统调度的影响下这种算法仍能够保证临界区的互斥访问，但这是如何做到的呢？这里并无必要给出一个严格证明，我们分成两种情况进行简单说明：

- 第一种情况考虑两个线程的 `lock` 操作不重叠的情况，这种情况比较简单。假设  $T_j$  已经成功 `lock` 并进入了临界区且尚未离开，此时  $T_i$  开始尝试进入临界区，它能成功吗？根据 `lock` 的流程， $T_i$  会将  $\text{flag}_i$  设置为 `true`，并将 `turn` 设置为  $j$ 。接下来来看忙等的两个条件。首先是 `turn`，它被  $T_i$  设置为  $j$ ，而  $T_j$  在退出临界区之前不会对它进行修改，所以此时 `turn` 的值一定为  $j$ ，条件成立；其次是 `flagj`，注意到  $T_j$  在进入临界区之前的 `lock` 中将其修改为 `true`，且在离开临界区之前不会将其改为 `false`，所以此时 `flagj` 一定为 `true`，条件也成立。因此， $T_i$  会陷入忙等而不会进入临界区。
- 第二种情况考虑两个线程的 `lock` 操作由于操作系统调度出现交错现象，也即两个线程在同段时间内尝试进入临界区，是否只有一个线程能成功进入呢？为了方便起见我们先排除  $\text{flag}_{i,j}$  的影响。因为两个线程在 `lock` 的第一步就是将自己的 `flag` 设置为 `true`，且在退出临界区之后才会修改为 `false`。从时间上，我们只考察从两个线程开始 `lock` 直到某个线程成功进入临界区且未离开的这段时间，我们可以认为在这段时间内始终有  $\text{flag}_i = \text{flag}_j = \text{true}$ 。

于是两个线程的忙等条件可以得到简化：即线程  $T_i$  只要  $\text{turn} = j$  就一直忙等，而线程  $T_j$  只要  $\text{turn} = i$  就一直忙等。每个线程都不能单靠自己进入临界区：以  $T_i$  为例，它将 `turn` 设置为  $j$ ，但它忙等的条件也是  $\text{turn} = j$ ，于是只有它自己的话就会进入忙等而不能进入临界区。必须要两个线程的协作：即当  $T_j$  将 `turn` 设置为  $i$ ， $T_i$  才能进入临界区。

可以看出问题的关键在于 `turn`。无论操作系统如何进行调度，在单核 CPU 上，线程  $T_i$  和线程  $T_j$  对于 `turn` 的修改总有一个在时间上靠后。不妨设  $T_i$  在  $T_j$  之后将 `turn` 修改为  $j$ ，那么这段时间中 `turn` 就会一直是  $j$  了，因为两个线程都已经完成了仅有一次的对于 `turn` 的修改。于是， $T_i$  不可能在这一轮进入临界区了。考虑  $T_j$ ，它已经完成了  $\text{turn} \leftarrow i$  的修改，此时可能已陷入忙等或者还没开始忙等。无论如何，只要接下来操作系统从  $T_i$  切换到  $T_j$ ， $T_j$  就能退出忙等并成功进入临界区。结论是，哪个线程最后修改了 `turn`，则另一个线程最终一定能进入临界区。

这样我们就说明了 Peterson 算法是如何在操作系统调度下仍保证互斥访问的。那么它能够满足空闲则入和有界等待这两个必须的要求吗？在这里需要说明的是，我们不必考虑操作系统调度导致的一些极端情况。比如说在考察是否能满足有界等待的时候，假如某个线程在进入等待状态之后，操作系统就再也不会调度到这个线程，看起来是出现了饥饿现象。但是这应当被归结于操作系统调度算法而非我们的锁实现。因此，我们需要合理地假设调度算法较为公平，各线程分到的 CPU 资源比较接近，这样调度算法就不会在很大程度上影响到锁的效果。现在回过头来看两个要求：

- 对于空闲则入，之前我们说明了当资源空闲的时候，如果只有单个线程要进入临界区，那么它一定能进去。如果两个线程要同时进入的话，只需要等到两个线程对于 `turn` 的修改均完成就能够确定哪个线程进入临界区，这一定能够在有限时间内做到。
- 对于有界等待，假设线程  $T_i$  由于没有竞争过  $T_j$  而在 `lock` 中陷入忙等待，相反  $T_j$  则成功进入临界区。在  $T_j$  离开临界区之后，可以发现它不可能在  $T_i$  进入临界区之前再次进入临界区了。可以使用反证法，假设  $T_j$  能够在  $T_i$  不进入的情况下再次进入，考虑  $T_j$  的忙等循环的两个条件，由于  $T_i$  一直处在忙等状态，有  $\text{flag}_i$  为 `true`，而  $\text{turn} = i$ （由  $T_j$  在进入忙等循环之前修改， $T_i$  已经在忙等没有机会修改为  $j$ ），可知  $T_j$  会卡在忙等中不能进入临界区，与假设矛盾。相反，从  $T_i$  的视角看来，在  $T_j$  离开临界区之后，只要操作系统通过调度切换到  $T_i$ ， $T_i$  立即就能进入临界区（除非在  $T_j$  完成  $\text{flag}_j \leftarrow \text{true}$  之后立即切换，但这种情况也只是让  $T_i$  晚一些进入临界区），有兴趣的同学可自行验证。总体看来， $T_i$  和  $T_j$  是在交替进入临界区，因此不存在某一个线程的无限等待——即饥饿现象。

这说明，Peterson 算法确实能够满足忙则等待、空闲则入和有界等待三个要求，是一种合格的锁实现。类似的算法还有适用于双线程的 Dekkers 算法 以及适用于多线程的 Eisenberg & McGuire 算法。这类算法仅依赖最基本的访存操作以及用户态权限，但思想上用到了多标记的组合以及精巧的构造，虽然代码比较简单，但

是不容易理解，要证明其正确性更是十分复杂。更为重要的是，这类算法存在着时代局限性，对 CPU 的内存访问有着比较严格的要求，因此这类算法是不能不加修改的运行在现代多核处理器上的。如果要保证其正确性，则要付出极大的性能开销，甚至得不偿失。因此，目前在实践中我们并不会用到这类算法实现锁机制。关于这种局限性更加深入的讨论超出了本书的范围，因此我们将其放在附录中，有兴趣的同学可以参考[附录 F: 类 Peterson 算法的局限性和内存顺序](#)。

#### 9.4.5 基于硬件机制及特殊指令的锁实现

上面我们介绍了锁的纯软件实现。可以看到，为了保证互斥访问要求，需要用到类似于 Peterson 的十分精密复杂的算法，它们不易理解，而且与现代处理器（特别是多核架构的处理器）的适配性不好。于是，我们可以考虑将这种通用的锁机制的需求下沉到硬件。如果硬件能提供相应的一些支持，那么我们在软件层面上就能比较简单地实现锁机制，且总体性能较高。下面依次介绍两种实现锁机制相关硬件支持：即关闭中断和原子指令。

##### 关闭中断

从前面的分析可以看出，站在用户态软件也就是应用的角度，并发问题的一个重要来源是操作系统随时有可能通过抢占式调度切换到其他线程，特别是当线程处于临界区被切换出去的时候，将导致不同线程的临界区交错运行使得共享资源访问出错。那么如何保证线程处于临界区的时候不会被操作系统切换出去呢？经过第三章的学习，我们知道操作系统进行的这种抢占式调度是由时钟中断触发的。如果应用程序能够控制中断的打开（使能）与关闭（屏蔽），那就能提供互斥解决方案了。代码如下：

```

1 fn lock() {
2 disable_interrupt(); //屏蔽中断的机器指令
3 }
4
5 fn unlock() {
6 enable_interrupt(); //使能中断的机器指令
7 }
```

在进入临界区之前，线程调用 `lock` 函数屏蔽整个 CPU 包括时钟中断在内的所有中断。这样在临界区中，线程就不用担心会被操作系统切换出去了，可以放心操作共享资源了。实际上此时它独占了整个 CPU。在离开临界区之后，线程再调用 `unlock` 函数使能所有中断，因为此时它再被切换出去也没有问题了。

这种方法的优点是简单，但是缺点则很多。首先，这种做法给了用户态程序使能/屏蔽中断这种特权，相当于相信应用并放权给它。这会面临和我们引入抢占式调度之前一样的问题：线程可以选择恶意永久关闭中断而独占所有 CPU 资源，这将会影响到整个系统的正常运行。因此，事实上至少在 RISC-V 这样含多个特权级的架构中，这甚至是完全做不到的。回顾第三章，可以看到中断使能和屏蔽的相关标志位分布在 S 特权级的 CSR `sstatus` 和 `sie`，甚至更高的 M 特权级的 CSR 中。用户态试图修改它们将会触发非法指令异常，操作系统会直接杀死该线程。其次，即使我们能够做到，它对于多处理器架构也是无效的。假设同一进程的多个线程运行在不同的 CPU 上，它们都尝试访问同种共享资源。一般来说，每个 CPU 都有自己的独立的中断相关的寄存器，它只能对自己的中断处理进行设置。对于一个线程来说，它可以关闭它所在 CPU 的中断，但是这无法影响到其他线程所在的 CPU，其他线程仍然可以在此时进入临界区，便不能满足互斥访问的要求了。所以，采用控制中断的方式仅对一些非常简单，且信任应用的单处理器场景有效，而对于更广泛的其他场景是不够的。

然而，需要注意的是，这种关闭中断的方法作为用户态锁机制的实现很不靠谱，在实现内核态锁机制的时候有些情况下却是必不可少的。我们将在第九章介绍位于内核态的一种基于关中断的锁实现。

## 原子指令

### 常用的 CAS 和 TAS 指令

在之前的 `adder_simple_spin.rs` 中，我们尝试用一个全局变量作为锁标记，但是最后却失败了。主要原因在于这个锁标记作为全局变量也是一种共享资源，对它进行的操作也是多阶段多条指令的，也会受到操作系统调度的影响，从而不能满足互斥访问要求，也就无法正确实现锁机制了。应该如何解决这个问题呢？回忆在 `adder_fixed.rs` 中我们使用 Rust 提供的原子操作将全局变量 A 的临界区缩小为一条原子指令使其免受操作系统调度的影响，这里我们也可以换用原子操作处理锁标记达到同样的效果，不过需要使用不同类型的原子操作：

```

1 // user/src/bin/adder_atomic.rs
2
3 static OCCUPIED: AtomicBool = AtomicBool::new(false);
4
5 fn lock() {
6 while OCCUPIED
7 .compare_exchange(false, true, Ordering::Relaxed, Ordering::Relaxed)
8 .is_err()
9 {
10 yield_();
11 }
12 }
13
14 fn unlock() {
15 OCCUPIED.store(false, Ordering::Relaxed);
16 }
```

这里我们将全局锁标记替换为原子 `bool` 类型 `AtomicBool`。它支持 `AtomicBool::compare_exchange` 操作，接口定义如下：

```

pub fn compare_exchange(
 &self,
 current: bool,
 new: bool,
 success: Ordering,
 failure: Ordering,
) -> Result<bool, bool>;
```

其功能为：如果原子变量当前的值与 `current` 相同，则将原子变量的值修改为 `new`，否则不进行修改。无论是否进行修改，都会返回原子变量在操作之前的值。可以看到返回值是一个 `Result`，如果修改成功的话这个值会用 `Ok` 包裹，否则则会用 `Err` 包裹。关于另外两个内存顺序参数 `success` 和 `failure` 不必深入了解，在单核环境下使用 `Ordering::Relaxed` 即可。注意 `compare_exchange` 作为一个基于硬件的原子操作，**它不会被操作系统的调度打断**。

那么 `lock` 和 `unlock` 是如何实现的呢？在 `lock` 中，主体仍然是一个 `while` 循环。每次迭代我们使用 `compare_exchange` 进行如下操作：如果 `OCCUPIED` 当前是 `false`，表明目前没有线程在临界区内，那么就将 `OCCUPIED` 改成 `true`，返回 `Ok` 退出 `while` 循环，随后线程进入临界区；否则，`OCCUPIED` 当前是 `true`，表明已经有线程在临界区之内了，那么 `compare_exchange` 修改失败并返回 `Err`，循环还需要继续下去。这种情况由于是在单核 CPU 上，进行下一轮迭代一定会继续失败，故而及时通过 `yield` 系统调用交出 CPU 资源。`unlock` 的实现则比较简单，离开临界区的线程同样通过原子存储操作 `store` 将 `OCCUPIED` 修改为 `false` 表示已经没有线程在临界区中了，此后线程可以进入临界区了。尝试运行一下 `adder_atomic.rs`，可以看到它能够满足互斥访问需求。

Rust 核心库为我们提供了原子类型以及 `compare_exchange` 这种方便的操作，那么它们是用硬件提供的哪些原子指令来实现的呢？硬件提供的原子指令一般是对于一个内存位置进行一系列操作，并保证整个过程的原子性。比如说最经典的 **比较并交换**（Compare-And-Swap, CAS）指令。它在不同平台上的具体表现存在细

微差异，我们这里按照 RISC-V 的写法描述它的核心功能：它有三个源寄存器和一个目标寄存器，于是应该写成 CAS rd, rs1, rs2, rs3。它的功能是将一个内存位置存放的值（其地址保存在一个源寄存器中，假设是 rs1）与一个期待值 expected（保存在源寄存器 rs2 中）进行比较，如果相同的话就将内存位置存放的值改为 new（保存在源寄存器 rs3 中）。无论是否相同，都将执行 CAS 指令之前这个内存位置存放的值写入到目标寄存器 rd 中。如果用 Rust 语言伪代码的形式描述 CAS 指令的功能应该是这样：

```

1 fn compare_and_swap(ptr: *mut i32, expected: i32, new: i32) -> i32 {
2 let original = unsafe { *ptr };
3 if original == expected {
4 unsafe { *ptr = new; }
5 }
6 original
7 }
```

所以“比较并交换”展开来说是如果比较结果相同，则将 new 交换到内存中，然后将内存中原来的值交换出来并返回。这段伪代码仅能用来描述 CAS 指令的功能，但 CAS 指令并不等价于这段伪代码，因为硬件能够保证 CAS 指令中的一系列操作是原子的，而不是按顺序执行若干访存和算术指令。同时，这里我们假定内存位置存放的值以及寄存器中有效的值是 32 位的。对于 CAS 指令，硬件通常会支持 16 位、32 位、64 位等通用寄存器常用的位宽，并在大多数情况下要求内存地址是对齐到对应位宽的。忽略这些细节，我们可以发现 CAS 指令和 Rust 中的 compare\_exchange 本质上是一回事，后者可以直接用前者来实现。CAS 是实现包括锁在内的同步机制最主要的方式，因此主流平台上基本都提供了 CAS 指令的支持。比如，在 x86 平台上我们有 CMPXCHG 指令，在 SPARC-V9 平台上我们则有 CASA/SWAP 等指令。

除了 CAS 之外，曾经还有另一类常用来实现同步机制的原子指令，它被称为 **测试并设置** (Test-And-Set, TAS)。相比 CAS，TAS 没有比较的步骤，它直接将 new 写入到内存并返回内存位置原先的值。用 Rust 语言伪代码描述 TAS 的功能如下：

```

1 fn test_and_set(ptr: *mut i32, new: i32) -> i32 {
2 let original = unsafe { *ptr };
3 unsafe { *ptr = new; }
4 original
5 }
```

在 TAS 中，内存中的值一定会被改成 new，于是更关键的在于返回值，也就是 TAS 之前内存中的值。比如，可以这样来用 TAS 指令实现锁机制：

```

1 static mut OCCUPIED: i32 = 0;
2
3 unsafe fn lock() {
4 while (test_and_set(&mut OCCUPIED, 1) == 1) {}
5 }
6
7 unsafe fn unlock() {
8 OCCUPIED = 0;
9 }
```

这里我们仍然使用 OCCUPIED 作为全局锁标记表示是否已经有线程在临界区中。当一个线程想要进入临界区的时候，它会调用 lock 函数，其中使用 TAS 指令原子地将锁标记设置为 1 并返回原先的值。假设有多个线程同时想要进入临界区，由于硬件保证 TAS 指令的原子性不受中断或多核同时访问的影响，只有 TAS 指令被最早执行的线程能够看到 TAS 的返回值为 0，意味着此时还没有线程在临界区中，那么这个线程可以结束忙等进入临界区。而后会执行其他线程上的 TAS 指令，它们的返回值就是 1 了，需要忙等待进入临界区的线程退出临界区。这样就保证了互斥访问。unlock 则是将全局锁标记改成 0 即可。

## RISC-V 架构上的原子指令

如果同学还有印象的话，我们曾经在第二章介绍 *Trap* 上下文保存与恢复的时候用到过一系列读写 RISC-V 控制状态寄存器 (CSR) 的特殊指令，比如 `csrr`, `csrw`, 特别是 `csrrw` 等。当时我们提到这些指令也都是原子指令。现在我们更加深入的理解了原子指令的含义，同学如有兴趣可以想想这些指令为什么必须保证其原子性。

除此之外，RISC-V 架构的原子拓展 (Atomic，简称 A 拓展) 提供了一些对于一个内存位置上的值进行原子操作的原子指令，分为两大类。其中第一类被称为原子内存操作 (Atomic Memory Operation, AMO)。这类原子指令首先根据寄存器 `rs1` 保存的内存地址将值从内存载入到寄存器 `rd` 中，然后将这个载入的值与寄存器 `rs2` 中保存的值进行某种运算，并将结果写回到 `rs1` 中的地址对应的内存区域中。整个过程可以被概括为一种 **read-modify-write** 的三阶段操作，硬件能够保证其原子性。AMO 支持多种不同的运算，包括交换、整数加法、按位与、按位或、按位异或以及有/无符号整数最大或最小值。容易看出，这类指令能够方便地实现 `adder_fixed.rs` 中 Rust 提供的 `fetch_add` 这类原子操作。

RISC-V 提供的另一类原子指令被称为加载保留/条件存储 (Load Reserved / Store Conditional，简称 LR/SC)，它们通常被配对使用。首先，LR 指令可以读取内存中的一个值 (其地址保存在寄存器 `rs1` 中) 到目标寄存器 `rd`。然后，可以使用 SC 指令，它的功能是将内存中的这个值 (其地址保存在寄存器 `rs1` 中且与 LR 指令中的相同) 改成寄存器 `rs2` 保存的值，但前提是：执行 LR 和 SC 这两条指令之间的这段时间内，内存中的这个值并未被修改。如果这个前提条件不满足，那么 SC 指令不会进行修改。SC 指令的目标寄存器 `rd` 指出 SC 指令是否进行了修改：如果进行了修改，`rd` 为 0；否则，`rd` 可能为一个非零的任意值。

那么 SC 指令是如何判断此前一段时间该内存中的值是否被修改呢？在 RISC-V 架构下，存在一个 **保留集** (Reservation Set) 的概念，这也是“加载保留”这种叫法的来源。保留集用来实现 LR/SC 的检查机制：当 CPU 执行 LR 指令的时候，硬件会记录下此时内存中的值是多少，此外还可能有一些附加信息，这些被记录下来的信息就被称为保留集。之后，当其他 CPU 或者外设对内存这个值进行修改的时候，硬件可以将这个值对应的保留集标记为非法或者删除。等到之前执行 LR 指令的 CPU 执行 SC 指令的时候，CPU 就可以检查保留集是否存在/合法或者保留集记录的值是否与内存中现在的值一致，以这种方式来决定是否进行写入以及目标寄存器 `rd` 的值。

RISC-V 并不原生支持 CAS/TAS 原子指令，但我们可以用 LR/SC 指令来实现它。比如下面是通过 LR/SC 指令对来模拟 CAS 指令。有兴趣的同学可以对照注释自行研究。

```

1 # 参数 a0 存放 内存中的值的所在地址
2 # 参数 a1 存放 expected
3 # 参数 a2 存放 new
4 # 返回值 a0 略有不同：这里若比较结果相同则返回 0，否则返回 1
5 # 而不是返回 CAS 之前内存中的值
6 cas:
7 lr.w t0, (a0) # LR 将值加载到 t0
8 bne t0, a1, fail # 如果值和 a1 中的 expected 不同，跳转到 fail
9 sc.w t0, a2, (a0) # SC 尝试将值修改为 a2 中的 new
10 bne t0, cas # 如果 SC 的目标寄存器 t0 不为 0，说明 LR/SC 中间值被修改，重试
11 li a0, 0 # 成功，返回值为 0
12 ret # 返回
13 fail:
14 li a0, 1 # 失败，返回值为 1
15 ret # 返回

```

## 原子指令小结

让我们来对原子指令部分进行一个小结。为了提供软件所需的包含互斥锁在内的各种同步机制，硬件对于内存中的一个字、双字、四字（位宽分别为 16、32、64 位，且通常要求是对齐的）这类通用的存储单位提供了一系列原子指令，这些原子指令能够对内存中的值进行加载、运算、修改等多种操作，且能够保证整个过程是原子的。也就是说，在硬件层面上，其原子性有着更高的优先级而不会被中断、多个 CPU 同时访问内存中同个位置或者指令执行中的更多情况破坏。

作为例子我们介绍了经典的 CAS/TAS 指令以及 RISC-V 上提供的 LR/SC 指令对，基于它们我们能够简单且高效的实现锁机制。重新回顾一下这些指令，可以发现它们从结果上都存在成功/失败之分。如果多个 CPU 同时用这些指令访问内存中同一个值，显然只有一个 CPU 能够成功。事实上，当多个 CPU 同时执行这些原子指令的时候，它们会将相关请求发送到 CPU 与 RAM 间总线上，总线会将这些请求进行排序。这就好像一群纷乱的游客在通过一个狭窄的隘口的时候必须单列排队通过，无论如何总会产生一种顺序。于是我们会看到，请求排在最前面的 CPU 能够成功，随后它便相当于独占了这一块被访问的内存区域。接下来，排在后面的 CPU 的请求都会失败了，这种状况会持续到之前独占的 CPU 将对应内存区域重置（相当于 unlock）。正因如此，我们才说：“原子指令是整个计算机系统中最根本的原子性和互斥性的来源。”这种最根本的互斥性来源于总线的仲裁，表现为原子指令，作用范围为基础存储单位。在原子指令的基础上，我们可以灵活地编写软件来延伸互斥性或其他同步需求的作用范围，使得对于各种丰富多彩的资源（如复杂数据结构和多种外设）我们都能将其管理得有条不紊。

虽然原子指令已经能够简单高效的解决问题了，但是在很多情况下，我们可以在此基础上再引入软件对资源进行灵活的调度管理，从而避免资源浪费并得到更高的性能。

### 9.4.6 在操作系统支持下实现让权等待

在编程时，常常遇到必须满足某些条件（或是遇到某些事件）才能进行接下来的流程的情况。如果一开始这些条件并不成立，那么必须通过某种方式暂时在原地 **等待** 这些条件满足之后才能继续前进。我们可以用多种不同的方式进行等待，最为常见的几种包括：忙等、通过 `yield` 暂时让权以及后面重点介绍的阻塞。

#### 忙等

应用 `adder_peterson_spin.rs` 展示了如何使用 Peterson 算法实现锁机制，在这里我们关注它如何进行等待。在其中的 `lock` 函数中，需要满足  $flag_j$  为 `false` 和  $turn \neq j$  两个条件至少满足其一才能继续前进并成功进入临界区。于是在第 20 行，我们通过一个 `while` 循环进行 **忙等待** 直到条件成立。

运行这个测例会发现一种现象：虽然其结果是正确的，但是每个线程尝试进入临界区的次数 `PER_THREAD` 比较大（比如大于 5000）的时候有时总运行时间会非常长，对于这种数据规模来说是完全不可接受的。如何解释这种现象呢？对于一个线程来说，在 `while` 循环的第一次迭代中如果发现条件不成立，那么就可以知道在这个分配给它的时间片中，接下来的所有迭代条件都不成立。因为我们是单核，此时 CPU 由这个线程使用，它只是不断做判断而不会做任何修改。另一个线程可以做修改，但是暂时没有 CPU 使用权。所以，在这种忙等待的做法中，只要一开始条件不成立，那么剩余时间片都会被浪费在 `while` 循环中，这也就导致了某些情况下测例运行时间很长。要解决这个问题，需要线程一旦注意到条件不成立，就立即主动或者被动交出 CPU 使用权，反正它暂时无法做任何有意义的事情了。我们后面再介绍具体如何实现。

总体上说，若要以忙等方式进行等待，首先要保证忙等是有意义的，不然就只是单纯的在浪费 CPU 资源。怎样才算是有意义的忙等呢？那就是 **在忙等的时候被等待的条件有可能从不满足变为满足**。比如说，一个线程占据 CPU 资源进行忙等的同时，另一个线程可以在另一个 CPU 上执行，外设也在工作，它们都可以修改内存使得条件得到满足。这种情况才有等待的价值。于是可以知道，在单核环境下且等待条件不涉及外设的时候，一个线程的忙等是没有意义的，因为被等待的条件的状态不可能发生变化。

在忙等有意义的前提下，忙等的优势是在条件成立的第一时间就能够进行响应，对于事件的响应延迟更低，实时性更好，而且不涉及开销很大的上下文切换。它的缺点则是不可避免的会浪费一部分 CPU 资源在忙等上。因此，如果我们能够预测到条件将很快得到满足，在这种情况下使用忙等是一个好主意。如果条件成立的时间无法预测或者所需时间比较长，那还是及时交出 CPU 资源更好。

## 通过 `yield` 暂时让权

在应用 `adder_peterson_spin.rs` 的基础上，线程可以在发现条件暂时不成立的情况下通过 `yield` 系统调用主动交出 CPU 使用权：

```

1 // user/src/bin/adder_peterson_yield.rs
2
3 unsafe fn lock(id: usize) {
4 FLAG[id] = true;
5 let j = 1 - id;
6 TURN = j;
7 // Tell the compiler not to reorder memory operations
8 // across this fence.
9 compiler_fence(Ordering::SeqCst);
10 while FLAG[j] && TURN == j {
11 yield_();
12 }
13 }
```

事实上，这并不是我们第一次利用 `yield` 在等待条件不满足的时候让权了，此前我们就做过类似的事情。考虑 `sys_waitpid` 系统调用在 `user_lib` 中的封装 `waitpid`：

```

1 // user/src/lib.rs
2
3 pub fn waitpid(pid: usize, exit_code: &mut i32) -> isize {
4 loop {
5 match sys_waitpid(pid as isize, exit_code as *mut _) {
6 -2 => {
7 yield_();
8 }
9 // -1 or a real pid
10 exit_pid => return exit_pid,
11 }
12 }
13 }
```

这里是被等待的子进程尚未退出，于是我们先 `yield`，等下一次获得 CPU 使用权的时候再检查该子进程是否退出。

另一个例子是在本章之前，`sleep` 由用户库 `user_lib` 提供而不是一个系统调用：

```

1 // user/src/lib.rs
2
3 pub fn sleep(period_ms: usize) {
4 let start = sys_get_time();
5 while sys_get_time() < start + period_ms as isize {
6 sys_yield();
7 }
8 }
```

这里是发现睡眠的时间还不够，于是 `yield` 并在下一次获得时间片的时候再检查。

使用 `yield` 实现通常情况下不会出错。但是它的实际表现却很大程度上受到操作系统调度器的影响。如果在条件满足之前就多次调度到等待的线程，虽然看起来线程很快就会再次通过 `yield` 主动让权从而没什么开销，但是实际上却增加了上下文切换的次数。上下文切换的开销是很大的，除了要保存和恢复寄存器之外，更重要的一点是会破坏程序的时间和空间局部性使得我们无法高效利用 CPU 上的各类缓存。比如说，我们的实现中在 `Trap` 的时候需要切换地址空间，有可能需要清空 TLB；由于用户态和内核态使用不同的栈，在应用 `Trap` 到内核态的时候，缓存中原本保存着用户栈的内容，在执行内核态代码的时候可能由于缓存容量不足而

需要逐步替换成内核栈的内容，而在返回用户态之后又需要逐步替换回来。整个过程中的缓存命中率将会很低。所以说，即使线程只是短暂停留也有可能对整体性能产生影响。相反，如果在条件满足很久之后才调度到等待的线程，这则会造成事件的响应延迟不可接受。使用 `yield` 就有可能出现这些极端情况，而且我们完全无法控制或预测其效果究竟如何。因此，我们需要一种更加确定、可控的等待方案。

## 阻塞

在操作系统的协助下，我们可以对于等待进行更加精细的控制。为了避免等待事件的线程在事件到来之前被调度到而产生大量上下文切换开销，我们可以新增一种 **阻塞** (Blocking) 机制。当线程需要等待事件到来的时候，操作系统可以将该线程标记为阻塞状态 (Blocked) 并将其从调度器的就绪队列中移除。由于操作系统每次只会从就绪队列中选择一个线程分配 CPU 资源，被阻塞的线程就不再会获得 CPU 使用权，也就避免了上下文切换。相对的，在线程要等待的事件到来之后，我们需要解除线程的阻塞状态，将线程状态改成就绪状态，并将线程重新加入到就绪队列，使其有资格得到 CPU 资源。这就是与阻塞机制配套的唤醒机制。在线程被唤醒之后，由于它所等待的事件已经出现，在操作系统调度到它之后它就可以继续向下运行了。

阻塞与唤醒机制相配合就可以实现精确且高效的等待。阻塞机制保证在线程等待的事件到来之前，线程不会参与调度，因此不会浪费任何时间片或产生上下文切换。唤醒机制则在事件到来之后允许线程正常继续执行。注意到，操作系统能够感知到事件以及等待该事件的线程，因此根据事件的实时性要求以及线程上任务的重要程度，操作系统可以在对于调度策略进行调整。比如，当事件为键盘或鼠标输入时，操作系统可以在唤醒之后将对应线程的优先级调高，让其能够被尽量早的调度到，这样就能够降低响应延迟并提升用户体验。也就是说，相比 `yield`，这种做法的可控性更好。

阻塞机制的缺点在于会不可避免的产生两次上下文切换。站在等待的线程的视角，它会被切换出去再切换回来然后再继续执行。在事件产生频率较低、事件到来速度比较慢的情况下这不是问题，但当事件产生频率很高的时候直接忙等也许是更好的选择。此外，阻塞机制相对比较复杂，需要操作系统的支持。

下面介绍我们的操作系统如何实现阻塞机制以及阻塞机制的若干应用。

## 实现阻塞与唤醒机制

在任务管理方面，此前我们已经有 `suspend`/`exit_current_and_run_next` 两种接口。现在我们新增第三种接口 `block_current_and_run_next`：

```

1 // os/src/task/mod.rs
2
3 pub fn block_current_and_run_next() {
4 let task = take_current_task().unwrap();
5 let mut task_inner = task.inner_exclusive_access();
6 let task_cx_ptr = &mut task_inner.task_cx as *mut TaskContext;
7 task_inner.task_status = TaskStatus::Blocked;
8 drop(task_inner);
9 schedule(task_cx_ptr);
10 }
11
12 pub fn suspend_current_and_run_next() {
13 let task = take_current_task().unwrap();
14 let mut task_inner = task.inner_exclusive_access();
15 let task_cx_ptr = &mut task_inner.task_cx as *mut TaskContext;
16 task_inner.task_status = TaskStatus::Ready;
17 drop(task_inner);
18 add_task(task);
19 schedule(task_cx_ptr);
20 }
```

当一个线程陷入内核态之后，如果内核发现这个线程需要等待某个暂未到来的事件或暂未满足的条件，就需要调用这个函数阻塞这个线程。从实现来看，第 4 行我们将线程从当前 CPU 的处理器管理结构中移除；第

5~8 行我们将线程状态修改为阻塞状态 Blocked；最后在第 9 行我们保存当前的任务上下文到线程控制块中并触发调度切换到其他线程。

作为比较，上面还给出了我们非常熟悉的会在时钟中断时被调用的 `suspend_current_and_run_next` 函数。我们的 `block` 版本和它的区别仅仅在于我们会将线程状态修改为 Blocked 以及我们此处 **不会将被阻塞的线程重新加回到就绪队列中**。这样才能保证被阻塞的线程在事件到来之前不会被调度到。

上面就是阻塞机制的实现，那么唤醒机制如何实现呢？当一个事件到来或是条件被满足的时候，首先我们要找到有哪些线程在等待这个事件或条件，这样才能够唤醒它们。因此，在内核中我们会 **将被阻塞的线程的控制块按照它们等待的具体事件或条件分类存储**。通常情况，对于每种事件，我们将所有等待该事件的线程而被阻塞的线程保存在这个事件的阻塞队列（或称等待队列）中。这样我们在事件到来的时候就知道要唤醒哪些线程了。

在事件到来的时候，我们要从事件的等待队列中取出线程，并调用唤醒它们的函数 `wakeup_task`：

```

1 // os/src/task/manager.rs
2
3 pub fn wakeup_task(task: Arc<TaskControlBlock>) {
4 let mut task_inner = task.inner_exclusive_access();
5 task_inner.task_status = TaskStatus::Ready;
6 drop(task_inner);
7 add_task(task);
8 }
```

这里只是简单的将线程状态修改为就绪状态 Ready 并将线程加回到就绪队列。

在引入阻塞机制后，还需要注意它跟线程机制的结合。我们知道，当进程的主线程退出之后，进程的所有其他线程都会被强制退出。此时，这些线程不光有可能处于就绪队列中，还有可能正被阻塞等待某些事件，因而在这些事件的阻塞队列中。所以，在线程退出时，我们需要 **检查线程所有可能出现的位置并将线程控制块移除，不然就会造成内存泄漏**。有兴趣的同学可以参考 `os/src/task/mod.rs` 中的 `remove_inactive_task` 的实现。

这样，我们就成功实现了阻塞-唤醒机制。

## 基于阻塞机制实现 sleep 系统调用

现在我们尝试基于阻塞机制将 sleep 功能作为一个系统调用而不是在用户库 `user_lib` 中通过 `yield` 来实现。首先来看该系统调用的接口定义：

```

/// 功能：当前线程睡眠一段时间。
/// 参数：sleep_ms 表示线程睡眠的时间，单位为毫秒。
/// 返回值：0
/// syscall ID : 101
pub fn sys_sleep(sleep_ms: usize) -> isize;
```

当线程使用这个系统调用之后，它将在陷入内核态之后被阻塞。线程等待的事件则是时钟计数器的值超过当前时间再加上线程睡眠的时长的总和，也就是超时之后就可以唤醒线程了。因此，我们要等待的事件可以用一个超时时间表示。然后，我们用一个数据结构 `TimerCondVar` 将这个超时时间以及等待它的线程放在一起，这是唤醒机制的关键数据结构：

```

1 pub struct TimerCondVar {
2 pub expire_ms: usize,
3 pub task: Arc<TaskControlBlock>,
4 }
```

那么如何保证在超时的时候内核能够接收到这个事件并做相应处理呢？我们这里选择一种比较简单做法：即在每次时钟中断的时候检查在上个时间片中是否有一些线程的睡眠超时了，如果有的话我们就唤醒它们。

这样的做法可能使得线程实际睡眠的时间不太精确，但是其误差也不会超过一个时间片，还算可以接受。同学有兴趣的话可以想想看有没有什么更好的做法。

当时钟中断的时候我们可以扫描所有的 TimerCondVar，将其中已经超时的移除并唤醒相应的线程。不过这里我们可以用学过的数据结构知识做一点小优化：可以以超时时间为键值将所有的 TimerCondVar 组织成一个小根堆（另一种叫法是优先级队列），这样每次时钟中断的时候只需不断弹出堆顶直到堆顶还没有超时。

为了能将 TimerCondVar 放入堆中，需要为其定义偏序、全序比较方法和相等运算，也就是要实现 PartialEq, Eq, PartialOrd, Ord 这些 Trait：

```

1 // os/src/timer.rs
2
3 impl PartialEq for TimerCondVar {
4 fn eq(&self, other: &Self) -> bool {
5 self.expire_ms == other.expire_ms
6 }
7 }
8 impl Eq for TimerCondVar {}
9 impl PartialOrd for TimerCondVar {
10 fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
11 let a = -(self.expire_ms as isize);
12 let b = -(other.expire_ms as isize);
13 Some(a.cmp(&b))
14 }
15 }
16 impl Ord for TimerCondVar {
17 fn cmp(&self, other: &Self) -> Ordering {
18 self.partial_cmp(other).unwrap()
19 }
20 }
```

由于标准库提供的二叉堆 BinaryHeap 是一个大根堆，因此在实现 PartialOrd Trait 的时候我们需要将超时时间取反。这样的话，就可以把所有的 TimerCondVar 放到一个全局二叉堆 TIMERS 中：

```

// os/src/timer.rs

use alloc::collections::BinaryHeap;

lazy_static! {
 static ref TIMERS: UPSafeCell<BinaryHeap<TimerCondVar>> =
 unsafe { UPSafeCell::new(BinaryHeap::new()) };
}
```

设计好数据结构之后，首先来看 sleep 系统调用如何实现：

```

// os/src/syscall/sync.rs

pub fn sys_sleep(ms: usize) -> isize {
 let expire_ms = get_time_ms() + ms;
 let task = current_task().unwrap();
 add_timer(expire_ms, task);
 block_current_and_run_next();
 0
}

// os/src/timer.rs
```

(下页继续)

(续上页)

```
1 pub fn add_timer(expire_ms: usize, task: Arc<TaskControlBlock>) {
2 let mut timers = TIMERS.exclusive_access();
3 timers.push(TimerCondVar { expire_ms, task });
4 }
```

在 `sys_sleep` 中，我们首先计算当前线程睡眠超时时间 `expire_ms`，然后调用 `adder_timer` 生成一个 `TimerCondVar` 并将其加入到全局堆 `TIMERS` 中。注意这个过程中线程控制块是如何流动的：它被复制了一份并移动到 `TimerCondVar` 中，此时在处理器管理结构 `PROCESSOR` 中还有一份。而在调用 `block_current_and_run_next` 阻塞当前线程之后，`PROCESSOR` 中的那一份就被移除了。此后直到线程被唤醒之前，线程控制块都只存在于 `TimerCondVar` 中。

在时钟中断的时候则会调用 `check_timer` 尝试唤醒睡眠超时的线程：

```
1 // os/src/timer.rs
2
3 pub fn check_timer() {
4 let current_ms = get_time_ms();
5 let mut timers = TIMERS.exclusive_access();
6 while let Some(timer) = timers.peek() {
7 if timer.expire_ms <= current_ms {
8 // 调用 wakeup_task 唤醒超时线程
9 wakeup_task(Arc::clone(&timer.task));
10 timers.pop();
11 } else {
12 break;
13 }
14 }
15 }
```

当线程睡眠的时候退出，`timer.rs` 中的 `remove_timer` 可以移除掉线程所在的 `TimerCondVar`，在此不再赘述。

## 基于阻塞机制实现锁机制

为了在操作系统的支持下实现锁机制，我们可以将锁看成进程内的一种资源（类似文件描述符表和地址空间），一个进程可以有多把锁，这些锁可以用它们的 ID 来区分，每把锁可以用来保护不同的共享资源。进程内的所有线程均可以访问锁，但是只能通过系统调用这种间接的方式进行访问。因此，需要新增若干锁机制相关的系统调用：

```
1 // 功能：为当前进程新增一把互斥锁。
2 // 参数：blocking 为 true 表示互斥锁基于阻塞机制实现，
3 // 否则表示互斥锁基于类似 yield 的方法实现。
4 // 返回值：假设该操作必定成功，返回创建的锁的 ID 。
5 // syscall ID: 1010
6 pub fn sys_mutex_create(blocking: bool) -> isize;
7
8 // 功能：当前线程尝试获取所属进程的一把互斥锁。
9 // 参数：mutex_id 表示要获取的锁的 ID 。
10 // 返回值：0
11 // syscall ID: 1011
12 pub fn sys_mutex_lock(mutex_id: usize) -> isize;
13
14 // 功能：当前线程释放所属进程的一把互斥锁。
15 // 参数：mutex_id 表示要释放的锁的 ID 。
16 // 返回值：0
```

(下页继续)

(续上页)

```

17 /// syscall ID: 1012
18 pub fn sys_mutex_unlock(mutex_id: usize) -> isize;

```

下面来看我们如何基于这些系统调用实现多线程计数器：

```

1 // user/src/bin/adder_mutex_blocking.rs
2
3 unsafe fn f() -> ! {
4 let mut t = 2usize;
5 for _ in 0..PER_THREAD {
6 mutex_lock(0);
7 critical_section(&mut t);
8 mutex_unlock(0);
9 }
10 exit(t as i32)
11 }
12
13 #[no_mangle]
14 pub fn main(argc: usize, argv: &[&str]) -> i32 {
15 ...
16 assert_eq!(mutex_blocking_create(), 0);
17 let mut v = Vec::new();
18 for _ in 0..thread_count {
19 v.push(thread_create(f as usize, 0) as usize);
20 }
21 ...
22 }

```

在第 16 行我们调用用户库提供的 `mutex_blocking_create` 在进程内创建一把基于阻塞实现的互斥锁，由于这是进程内创建的第一把锁，其 ID 一定为 0。接下来，每个线程在执行的函数 `f` 中，在进入临界区前后分别获取（第 6 行调用 `mutex_lock`）和释放（第 8 行调用 `mutex_unlock`）在主线程中创建的 ID 为 0 的锁。这里用到的三个函数都是由用户库直接封装相关系统调用得到的。

那么在内核态这些系统调用是如何实现的呢？在我们的设计中，允许同个进程内的多把锁有着不同的底层实现，这样更加灵活。因此，我们用 `Mutex Trait` 来规定描述一把互斥锁应该有哪些功能：

```

// os/src/sync/mutex.rs

pub trait Mutex: Sync + Send {
 fn lock(&self);
 fn unlock(&self);
}

```

其实只包含 `lock` 和 `unlock` 两个方法。然后，我们可以在进程控制块中新增互斥锁这类资源：

```

1 // os/src/task/process.rs
2
3 pub struct ProcessControlBlockInner {
4 ...
5 pub fd_table: Vec<Option<Arc<dyn File + Send + Sync>>>,
6 pub mutex_list: Vec<Option<Arc<dyn Mutex>>>,
7 ...
8 }

```

和文件描述符表 `fd_table` 一样，`mutex_list` 使用 `Vec<Option<T>>` 构建一个含有多个可空槽位且槽位数可以拓展的互斥锁表，表中的每个元素都实现了 `Mutex Trait`，是一种互斥锁实现。`sys_mutex_create` 会找到第一个空闲的槽位（如果没有则新增一个）并将指定类型的锁的实例插入进来：

```

1 // os/src/syscall/sync.rs
2
3 pub fn sys_mutex_create(blocking: bool) -> isize {
4 let process = current_process();
5 let mutex: Option<Arc<dyn Mutex>> = if !blocking {
6 Some(Arc::new(MutexSpin::new()))
7 } else {
8 Some(Arc::new(MutexBlocking::new()))
9 };
10 let mut process_inner = process.inner_exclusive_access();
11 if let Some(id) = process_inner
12 .mutex_list
13 .iter()
14 .enumerate()
15 .find(||(_, item)| item.is_none())
16 .map(||(id, _)| id)
17 {
18 process_inner.mutex_list[id] = mutex;
19 id as isize
20 } else {
21 process_inner.mutex_list.push(mutex);
22 process_inner.mutex_list.len() as isize - 1
23 }
24}

```

然后是 sys\_mutex\_lock 和 sys\_mutex\_unlock 的实现，它们都只是从进程控制块中的互斥锁表的指定槽位中复制一份 Mutex 实现，并调用其 lock 和 unlock 方法。以 sys\_mutex\_lock 为例：

```

1 // os/src/syscall/sync.rs
2
3 pub fn sys_mutex_lock(mutex_id: usize) -> isize {
4 let process = current_process();
5 let process_inner = process.inner_exclusive_access();
6 let mutex = Arc::clone(process_inner.mutex_list[mutex_id].as_ref().unwrap());
7 drop(process_inner);
8 drop(process);
9 mutex.lock();
10 0
11}

```

目前，我们仅提供了两种互斥锁实现：基于阻塞机制的 MutexBlocking 和基于类似 yield 机制的 MutexSpin，篇幅原因我们只介绍前者。首先看 MutexBlocking 包含哪些内容：

```

1 // os/src/sync/mutex.rs
2
3 pub struct MutexBlocking {
4 inner: UPSafeCell<MutexBlockingInner>,
5 }
6
7 pub struct MutexBlockingInner {
8 locked: bool,
9 wait_queue: VecDeque<Arc<TaskControlBlock>>,
10}

```

可以看到，最外层是一个 UPSafeCell，然后 inner 里面有两个成员：

- locked 作用和之前介绍的单标记软件实现相同，表示目前是否有线程进入临界区。在线程通过 sys\_mutex\_lock 系统调用尝试获取锁的时候，如果发现 locked 为 true，那么就需要等待 locked

变为 false，在此之前都需要被阻塞。

- `wait_queue` 作为阻塞队列记录所有等待 `locked` 变为 false 而被阻塞的线程控制块。

于是，获取和释放锁的实现方式如下：

```

1 // os/src/sync/mutex.rs
2
3 impl Mutex for MutexBlocking {
4 fn lock(&self) {
5 let mut mutex_inner = self.inner.exclusive_access();
6 if mutex_inner.locked {
7 mutex_inner.wait_queue.push_back(current_task().unwrap());
8 drop(mutex_inner);
9 block_current_and_run_next();
10 } else {
11 mutex_inner.locked = true;
12 }
13 }
14
15 fn unlock(&self) {
16 let mut mutex_inner = self.inner.exclusive_access();
17 assert!(!mutex_inner.locked);
18 if let Some(waking_task) = mutex_inner.wait_queue.pop_front() {
19 wakeup_task(waking_task);
20 } else {
21 mutex_inner.locked = false;
22 }
23 }
24}

```

- 对于 `lock` 来说，首先检查是否已经有线程在临界区中。如果 `locked` 为 true，则将当前线程复制一份到阻塞队列中，然后调用 `block_current_and_run_next` 阻塞当前线程；否则当前线程可以进入临界区，将 `locked` 修改为 true。
- 对于 `unlock` 来说，简单起见我们假定当前线程一定持有锁（也就是所有的线程一定将 `lock` 和 `unlock` 配对使用），因此断言 `locked` 为 true。接下来尝试从阻塞队列中取出一个线程，如果存在的話就将这个线程唤醒。被唤醒的线程将继续执行 `lock` 并返回，进而回到用户态进入临界区。在此期间 `locked` 始终为 true，相当于 释放锁的线程将锁直接移交给这次唤醒的线程。反之，如果阻塞队列中没有线程的话，我们则将 `locked` 改成 false，让后来的线程能够进入临界区。

观察 `lock` 中调用 `block_current_and_run_next` 的位置，可以发现阻塞在 `lock` 中的线程一经唤醒便能够立刻进入临界区，因此在 `unlock` 的时候我们最多只能唤醒一个线程，不然便无法满足互斥访问要求。我们这种实现能够满足一定的公平性，因为我们每次都是唤醒阻塞队列头的线程，于是每个线程在被阻塞有限时间之后一定能进入临界区。但还有一些其他的互斥锁实现，比如在 `unlock` 的时候将阻塞队列中的所有线程同时唤醒，感兴趣的同學可自行想想如何实现以及其优缺点。

---

#### 注解：为什么同样使用单标记，这里却无需用到原子操作？

这里我们仅用到单标记 `locked`，为什么无需使用原子指令来保证对于 `locked` 本身访问的互斥性呢？这其实是因为，RISC-V 架构规定从用户态陷入内核态之后所有（内核态）中断默认被自动屏蔽，也就是说与应用的执行不同，目前系统调用的执行是不会被中断打断的。同时，目前我们是在单核上，也不会有多个 CPU 同时执行系统调用的情况。在这种情况下，内核态的共享数据访问就仍在 `UPSafeCell` 的框架之内，只要使用它就能保证互斥访问。

---

## 9.4.7 小结

本节我们从一个多线程计数器的小例子入手，它的结果与我们的预期不同。于是我们以一种通用的方式：即从编译结果，到操作系统调度，再到处理器执行这多个阶段排查出错的原因。我们着重分析由于操作系统调度所导致的不同线程对全局变量 A 的操作出现的 **交错** 现象，进而引入了 **共享资源**、**临界区** 以及 **互斥** 等重要概念。我们介绍了 **锁** 这种通用的互斥原语的功能和使用方式，还提到了如何评价一种锁的实现：有忙则等待、空闲则入和有界等待这些必须的功能性要求，还有让权等待这种性能层面上的可选要求。

接下来，我们尝试以多种方式实现锁机制。首先我们尝试锁的纯软件实现。第一种方法是设置单变量作为锁标记，但是由于它自身无法支持互斥访问而失败。第二种方式是使用多变量联合表示锁的状态，我们以经典的 Peterson 算法为例说明了它的正确性，但它以及类似的其他算法尤其是在现代 CPU 上存在局限性。

于是我们考虑在硬件的帮助下实现锁机制。一种简单的方法是屏蔽中断保证临界区的原子性，但这种做法需要过度放权给用户态而且在多核环境下无效。另一种做法则是依赖原子指令，我们深入理解了其原子性内涵及其使用范围（仅支持通用存储单位）。我们介绍了经典的 CAS/TAS 型原子指令以及 RISC-V 架构提供的 AMO 和 LR/SC 原子指令，并最终用它们成功简单且高效的实现了锁机制。

随后我们介绍如何基于操作系统支持实现让权等待。一般地，我们列举出了三种不同的等待方式：**忙等**、**yield 轮询** 和 **阻塞** 并分析了它们的优劣势和各自的适用场景。我们着重在我们内核中实现了关键的阻塞机制，并实现了 sleep 和互斥锁系统调用作为其应用。在实际中实现锁的时候，我们可以将软件算法、硬件支持和操作系统支持融合起来，这样才能得到更好的锁实现。

最后，让我们再从整体上回顾一下互斥这一关键概念。可以思考一下为何会存在互斥访问需求？是因为不同线程（或其他执行流，比如说系统调用执行）在执行期间出现了 **时间与空间的交集**。空间交集体现为访问同种共享资源，而时间交集体现在对于共享资源的操作或访问在时间上存在交错。只要满足了这两个条件，那就必须考虑使用锁之类的互斥原语对共享资源进行保护。另一种思路则是想办法规避时间或空间交集，这样也能够避免并发问题。

## 9.4.8 参考文献

- Database Transaction, Wikipedia
- 竞态条件，来自维基百科
- Peterson 算法，来自维基百科
- Dekkers 算法，来自维基百科
- Eisenberg & McGuire 算法，来自维基百科
- 《操作系统概念》第七版，第六章
- 《Operating Systems: Three Easy Pieces》 Chapter 28 Locks
- 《The SPARC Architecture Manual》 Version 9 Page 153
- RISC-V A Standard Extension for Atomic Instructions, Version 2.1
- LL/SC，来自维基百科

## 9.5 信号量机制

### 9.5.1 本节导读

在上一节中，我们介绍了互斥锁的起因、使用和实现过程。通过互斥锁，可以让线程在临界区执行时，独占共享资源。然而，当我们需要一种线程间更灵活的同步访问需求，如要求同时最多只允许 N 个线程在临界区中访问共享资源，或让某个线程等待另外一个线程执行到某一阶段后再继续执行的同步过程等，互斥锁这种方式就有点力不从心了。

在本节中，将介绍功能更加强大和灵活的同步互斥机制—信号量 (Semaphore) 的设计思路，使用方式和在操作系统中的具体实现。可以看到，在实现信号量的时候可能会用到互斥锁和处理器提供的原子指令，因此它是一种更高级的同步互斥机制。

---

#### 注解：同步互斥

**同步 (Synchronization)** 和 **互斥 (Mutual Exclusion)** 事实上是在多线程并发访问过程中出现的两种不同需求。同步指的是线程执行顺序上的一些约束，比如一个线程必须等待另一个线程执行到某个阶段之后才能继续向下执行；而互斥指的是多线程在访问共享资源的时候，同一时间最多只有一个线程能够在共享资源的临界区中。

同步和互斥的界限其实比较模糊。比如，互斥也可以看成是一种同步需求，即一个线程在进入临界区之前必须等待当前在临界区中的线程（如果存在）退出临界区；而对于一种特定的同步需求，线程间也往往需要某些共享状态，线程需要通过查看或修改这些共享状态来进入等待或唤醒等待的线程。为了能够正确访问这些共享状态，就需要互斥。所以，二者之间是一种“你中有我，我中有你”的关系，我们就将这两种需求统称为 **同步互斥**，而将针对于这种需求比较通用的解决方案称之为 **同步互斥原语**（简称为 **同步原语**，英文 Synchronization Primitives）。

---

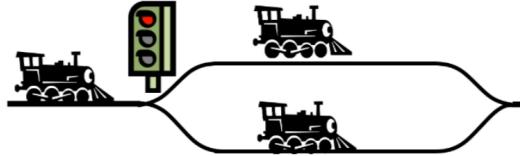
### 9.5.2 信号量的起源和基本思路

1963 年前后，当时的数学家兼计算机科学家 Edsger Dijkstra 和他的团队为 Electrologica X8 开发一款名为 THE 多道程序的操作系统的时候，提出了 **信号量 (Semaphore)** 的概念。信号量是一种同步原语，用一个变量或一种抽象数据类型实现，用于控制多个线程对共享资源的访问。1965 年，Dijkstra 发表了论文手稿“Cooperating sequential processes”，详细论述了在多个顺序代码执行流（在论文中被称为 Sequential Processes）的并发执行过程中，如果没有约束机制则会产生不确定的执行结果。为此，他提出了信号量的设计思路，能够让这些松耦合的顺序代码执行流能够在同步需求的约束下正确并发执行。

信号量相比互斥锁是一种更为强大和灵活的同步原语。它用来描述这样一种同步需求：初始状态下，某种资源的可用数量为一个非负整数  $N$ 。线程可以在某个时刻占用一个该种资源，并在使用过后将资源返还。然而，如果此时已经没有可用的资源了，也就是说所有这类资源都已经被其他线程占用了，那么当前线程就需要暂时进入等待，等待其他线程至少返回一个资源之后再重新尝试占用资源或是直接向下执行。可以结合现实中这样一个例子来形象理解信号量：考虑一个火车站只有两个站台，最多可以同时停放两列火车。那么就可以用一个  $N = 2$  的信号量来描述站台资源。设想，当一列火车尝试进站时，它会尝试占用一个站台来停车。如果此时仍有空闲的站台，那么它会看到绿色的信号灯而顺利进站；否则，它会看到信号灯为红色，此时它需要在站外等待空出一个站台再进站。

## ■ 信号量与铁路的类比

- 2个站台的车站
- 2个资源的信号量



信号量支持两种操作：P 操作（来自荷兰语中的 Proberen，意为尝试）和 V 操作（来自荷兰语中的 Verhogen，意为增加），其中 P 操作表示线程尝试占用一个资源，而与之匹配的 V 操作表示线程将占用的资源归还。P 操作和 V 操作也是基于阻塞-唤醒机制实现的。当进行 P 操作的时候，如果此时没有可用的资源，则当前线程会被阻塞；而进行 V 操作的时候，如果返还之后有了可用的资源，且此时有线程被阻塞，那么就可以考虑唤醒它们。从数据结构层面，信号量中存在一个整数变量表示当前资源的状态，同时还有一个阻塞队列保存所有被阻塞的线程。在信号量不同的实现中，整数变量的具体含义以及阻塞队列中的内容都是不同的。它们其实也属于所有线程都能访问到的共享资源，但是不用担心其互斥性。因为线程不会直接访问它们，而是只能通过 P 操作和 V 操作，操作系统保证这两个操作中包括整数变量的修改和阻塞/唤醒线程的整个流程是原子的。

一种信号量实现的伪代码如下所示：

```

1 fn P(S) {
2 if S >= 1
3 // 如果还有可用资源，更新资源剩余数量 S
4 S = S - 1;
5 // 使用资源
6 else
7 // 已经没有可用资源
8 // 阻塞当前线程并将其加入阻塞队列
9 <block and enqueue the thread>;
10 }
11
12 fn V(S) {
13 if <some threads are blocked on the queue>
14 // 如果已经有线程在阻塞队列中
15 // 则唤醒这个线程
16 <unblock a thread>;
17 else
18 // 否则只需恢复 1 资源可用数量
19 S = S + 1;
20 }
```

在上述实现中信号量中的整数变量  $S$  为非负整数。当  $S > 0$  时，表示还有  $S$  个可用资源；当  $S = 0$  时，表示没有可用资源且可能已经有线程被阻塞。显然  $S$  应该被初始化为  $N$ 。这种情形下 P 操作和 V 操作的具体实现可以参考注释。注意，阻塞在 P 操作中的线程一被唤醒就会立即进入临界区而不会检查此时是否有可用资源。这是因为 **进行 V 操作的线程直接将资源移交给它唤醒的线程**，于是此时并没有更新资源可用数量。

下面是另外一种信号量实现的伪代码：

```

1 fn P(S) {
2 S = S - 1;
3 if 0 > S then
4 // 阻塞当前线程并将其加入阻塞队列
5 <block and enqueue the thread>;
6 }
```

(下页继续)

(续上页)

```

8 fn V(S) {
9 S = S + 1;
10 if <some threads are blocked on the queue>
11 // 如果已经有线程在阻塞队列中
12 // 则唤醒这个线程
13 <unblock a thread>;
14 }

```

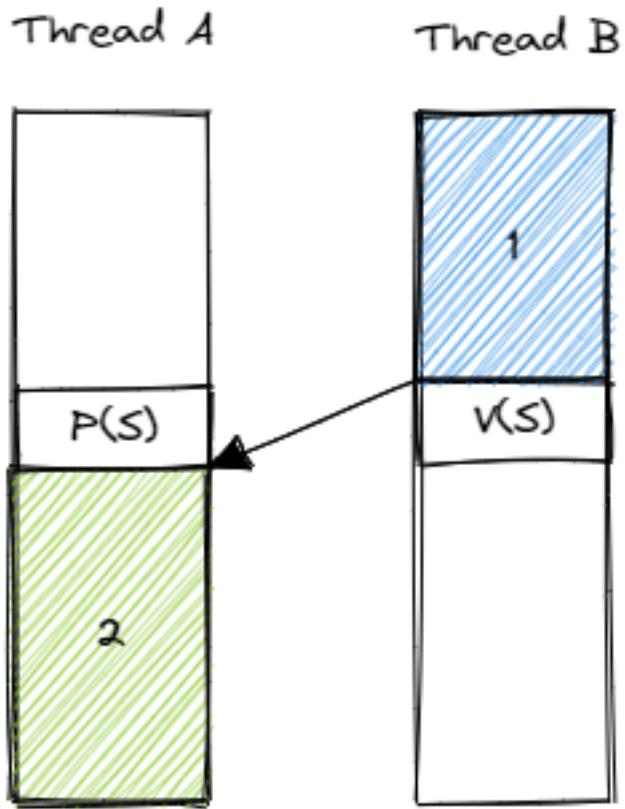
上述实现中，整数变量  $S$  的含义如下：当  $S > 0$  时，表示还有  $S$  个可用资源；当  $S = 0$  时，表示所有可用资源恰好耗尽；当  $S < 0$  时，表示此时有  $-S$  个线程被阻塞。显然  $S$  也应该被初始化为  $N$ 。对于 P 操作，我们首先将  $S$  减一，如果发现  $S < 0$ ，说明之前  $S \leq 0$ ，一定没有可用资源了，于是需要阻塞当前线程；对于 V 操作，这里将  $S$  加一可以这样理解：如果此时没有线程被阻塞则恢复 1 个可用资源；否则将阻塞线程数减少 1，因为当前线程将资源移交给了其中一个被阻塞的线程并唤醒了它。

这只是信号量的两种不同实现，本质上是相同的。

### 9.5.3 信号量的使用方法

信号量的初始资源可用数量  $N$  是一个非负整数，它决定了信号量的用途。如果  $N$  为大于 0 的任意整数，我们称之为计数信号量 (Counting Semaphore) 或者一般信号量 (General Semaphore)，它可以用来进行上面描述过的那种资源管理。特别地，当  $N = 1$  的时候，我们称其为二值信号量 (Binary Semaphore)。从定义上容易看出它和互斥锁是等价的。因此，互斥锁可以看成信号量的一种特例。

然而，当  $N = 0$  的时候，信号量就与资源管理无关了，而是可以用作一种比较通用的同步原语。比如，现在的需求是：线程 A 需要等待线程 B 执行到了某一阶段之后再向下执行。假设有一个  $N = 0$  的信号量。那么，在线程 A 需要等待的时候可以对该信号量进行 P 操作，于是线程会被阻塞。在线程 B 执行完指定阶段之后再对该信号量进行 V 操作就能够唤醒线程 A 向下执行。如下图所示：



在线程 A 和 B 上分别对一个  $N = 0$  的信号量使用 P 操作和 V 操作即可保证：在线程 B 的代码块 1 执行完毕之后才会开始执行线程 A 的代码块 2。换句话说，在线程 A 的代码块 2 的开头就可以假定此前线程 B 的代码块 1 已经执行完毕了。这在很多情况下是一种很有用的同步约束关系。

#### 注解：唤醒丢失问题

在上面线程 A 和 B 的同步问题中，其实未必总是线程 A 执行到 P 操作被阻塞住，然后线程 B 执行 V 操作唤醒线程 A。也有另一种可能是线程 B 先执行 V 操作，随后线程 A 再执行 P 操作。那么在线程 B 执行 V 操作的时候，信号量的阻塞队列中是没有任何线程的，此时 B 无法唤醒 A。但如果此时 B 什么都不做，那么之后 A 执行 P 操作陷入阻塞的时候就没有任何线程能够唤醒 A 了，这将导致 A 无法顺利执行。

这种问题被我们称为 **唤醒丢失 (Lost Wakeup)** 问题。为了解决这个问题，我们需要 B 在进行 V 操作的时候即使没有线程需要唤醒，也需要一种方法将这次可能的唤醒记录下来。请同学思考我们在上面的信号量实现中是如何解决这个问题的。

#### 9.5.4 信号量的系统调用接口

和互斥锁一样，我们将信号量也视为进程内的一种由操作系统管理并由进程内所有线程共享的资源。同个进程内可以有多个不同信号量，它们之间通过信号量 ID（与互斥锁或其他资源的 ID 独立）来区分。相关系统调用接口如下：

```

1 // 功能：为当前进程新增一个信号量。
2 // 参数：res_count 表示该信号量的初始资源可用数量，即 N，为一个非负整数。
3 // 返回值：假定该操作必定成功，返回创建的信号量的 ID。
4 // syscall ID : 1020

```

(下页继续)

(续上页)

```

5 pub fn sys_semaphore_create(res_count: usize) -> isize;
6 // 功能：对当前进程内的指定信号量进行 V 操作。
7 // 参数：sem_id 表示要进行 V 操作的信号量的 ID 。
8 // 返回值：假定该操作必定成功，返回 0 。
9 pub fn sys_semaphore_up(sem_id: usize) -> isize;
10 // 功能：对当前进程内的指定信号量进行 P 操作。
11 // 参数：sem_id 表示要进行 P 操作的信号量的 ID 。
12 // 返回值：假定该操作必定成功，返回 0 。
13 pub fn sys_semaphore_down(sem_id: usize) -> isize;

```

可以看到，这里我们分别用 `down` 和 `up` 这样比较形象的名字作为 P 操作和 V 操作的别名，因为 P 操作和 V 操作通常分别导致整数变量  $S$  的降低和增加。这几个系统调用也会在用户库 `user_lib` 被直接封装为 `semaphore_create/down/up`。

## 9.5.5 信号量的应用

这里给出两个应用：第一个是信号量作为同步原语来解决条件同步问题；第二个则是生产者和消费者基于一个有限缓冲进行协作的复杂问题。

### 条件同步问题

来看这样一个例子：

```

1 // user/src/bin/sync_sem.rs
2
3 const SEM_SYNC: usize = 0;
4
5 unsafe fn first() -> ! {
6 sleep(10);
7 println!("First work and wakeup Second");
8 semaphore_up(SEM_SYNC);
9 exit(0)
10 }
11
12 unsafe fn second() -> ! {
13 println!("Second want to continue, but need to wait first");
14 semaphore_down(SEM_SYNC);
15 println!("Second can work now");
16 exit(0)
17 }
18
19 #[no_mangle]
20 pub fn main() -> i32 {
21 // create semaphores
22 assert_eq!(semaphore_create(0) as usize, SEM_SYNC);
23 // create threads
24 let threads = vec![
25 thread_create(first as usize, 0),
26 thread_create(second as usize, 0),
27];
28 // wait for all threads to complete
29 for thread in threads.iter() {
30 waittid(*thread as usize);
31 }

```

(下页继续)

(续上页)

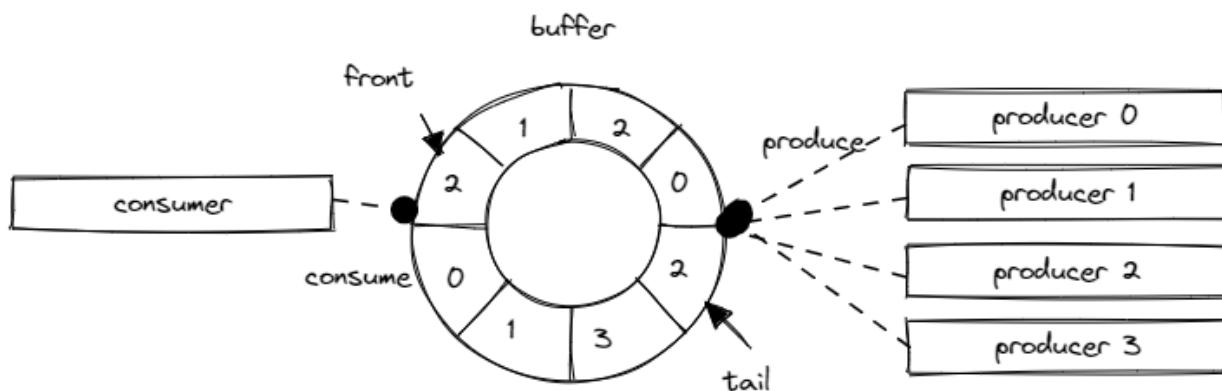
```

32 println!("sync_sem passed!");
33 0
34 }

```

其中，两个线程并发执行，第一个线程执行 `first` 函数而第二个线程执行 `second` 函数。我们想要达到的同步约束是：在第二个线程执行 `second` 的后一行打印之前，第一个线程必须完成 `first` 中的打印。于是，根据上面关于信号量用法的介绍，第 22 行我们调用 `semaphore_create` 函数创建一个用于同步的  $N = 0$  的信号量，其 ID 应为 `SEM_SYNC`。为了实现同步约束，则只需第一个线程在 `first` 中的打印结束后进行信号量的 V (也就是 up) 操作，而第二个线程在 `second` 后一次打印之前进行信号量的 P (也就是 down) 操作，这样即可解决这个问题。

## 生产者和消费者问题



生产者-消费者问题（也称为有限缓冲问题）是 Dijkstra 自 1965 年以来描述的一系列同步互斥问题中的一个。如图所示，一共有 5 个线程在同进程中进行协作，其中有 4 个生产者（Producer，图中右侧）和 1 个消费者（Consumer，图中左侧），它们共享一个容量有限的环形缓冲区（图中间）。生产者的职责是将输入放入缓冲区，而消费者则从缓冲区中取出数据进行处理。然而，这两种操作并不总是能够立即成功的。比如，当缓冲区已满的情况下，生产者就无法将数据放入缓冲区，需要等消费者取出数据空出缓冲区槽位；而当缓冲区为空没有数据的时候，消费者也无法从中取出数据，需要等生产者将数据填充到缓冲区。考虑使用信号量来实现上述同步需求，可以看成管理以下资源：

- 空闲槽位资源，初始数量  $N$  等于缓冲区容量。生产者每次写入需要占用 1 个，消费者每次读取恢复 1 个；
- 可用数据资源，初始数量  $N = 0$ （最开始缓冲区为空）。消费者每次读取占用 1 个，生产者每次写入恢复 1 个；
- 将缓冲区以及相应指针（即 `front` 和 `tail`）整体上视作一种共享资源，那么生产者和消费者的写入和读取都会对这个共享资源进行修改。注意 信号量只保证无可用资源时进行阻塞，但并不保证访问共享资源的互斥性，甚至这可能是两种不同资源。因此，我们还需要引入互斥锁对缓冲区进行保护，这里使用一个  $N = 1$  的二值信号量来实现。

代码如下：

```

1 // user/src/bin/mpsc_sem.rs
2
3 const SEM_MUTEX: usize = 0;
4 const SEM_EMPTY: usize = 1;
5 const SEM_AVAIL: usize = 2;
6 const BUFFER_SIZE: usize = 8;

```

(下页继续)

(续上页)

```

7 static mut BUFFER: [u8; BUFFER_SIZE] = [0; BUFFER_SIZE];
8 static mut FRONT: u8 = 0;
9 static mut TAIL: u8 = 0;
10 const PRODUCER_COUNT: u8 = 4;
11 const NUMBER_PER_PRODUCER: u8 = 100;
12
13 unsafe fn producer(id: *const u8) -> ! {
14 let id = *id;
15 for _ in 0..NUMBER_PER_PRODUCER {
16 semaphore_down(SEM_EMPTY);
17 semaphore_down(SEM_MUTEX);
18 BUFFER[TAIL] = id;
19 TAIL = (TAIL + 1) % BUFFER_SIZE;
20 semaphore_up(SEM_MUTEX);
21 semaphore_up(SEM_AVAIL);
22 }
23 exit(0)
24 }
25
26 unsafe fn consumer() -> ! {
27 for _ in 0..PRODUCER_COUNT * NUMBER_PER_PRODUCER {
28 semaphore_down(SEM_AVAIL);
29 semaphore_down(SEM_MUTEX);
30 print!("{} ", BUFFER[FRONT]);
31 FRONT = (FRONT + 1) % BUFFER_SIZE;
32 semaphore_up(SEM_MUTEX);
33 semaphore_up(SEM_EMPTY);
34 }
35 println!("");
36 exit(0)
37 }
38
39 #[no_mangle]
40 pub fn main() -> i32 {
41 // create semaphores
42 assert_eq!(semaphore_create(1) as u8, SEM_MUTEX);
43 assert_eq!(semaphore_create(BUFFER_SIZE) as u8, SEM_EMPTY);
44 assert_eq!(semaphore_create(0) as u8, SEM_AVAIL);
45 // create threads
46 let ids: Vec<_> = (0..PRODUCER_COUNT).collect();
47 let mut threads = Vec::new();
48 for i in 0..PRODUCER_COUNT {
49 threads.push(thread_create(
50 producer as u8,
51 &ids.as_slice()[i] as *const _ as u8,
52));
53 }
54 threads.push(thread_create(consumer as u8, 0));
55 // wait for all threads to complete
56 for thread in threads.iter() {
57 waittid(*thread as u8);
58 }
59 println!("mpsc_sem passed!");
60 0
61 }

```

第 42-44 行分别创建了二值信号量 SEM\_MUTEX，描述空闲槽位资源的信号量 SEM\_EMPTY 以及描述可用数

据资源的信号量 SEM\_AVAIL。生产者线程会执行 producer 函数，循环的每次迭代向共享缓冲区写入数据。于是在写入之前需要进行信号量 SEM\_EMPTY 的 down 操作尝试占用一个空闲槽位资源，而在写入之后进行信号量 SEM\_AVAIL 的 up 操作释放一个可用数据资源。相对的，消费者线程会执行 consumer 函数，循环的每次迭代从共享缓冲区读取数据。于是在读入之前需要进行信号量 SEM\_AVAIL 的 down 操作尝试占用一个可用数据资源，而在读取之后进行信号量 SEM\_EMPTY 的 up 操作释放一个空闲槽位资源。两个线程对共享缓冲区的操作都需要用二值信号量 SEM\_MUTEX 来保护。

从这个例子可以看出，信号量的使用可以是非常灵活的。同一个信号量的 P 操作和 V 操作不一定是连续的，甚至可以不在一个线程上。

**提示：**请同学们思考：能否将二值信号量的 down 和 up 操作放在循环每次迭代的最外层？为什么？

## 9.5.6 实现信号量

现在我们来看操作系统如何实现信号量相关的系统调用。首先，和互斥锁一样，信号量也是进程内的一种资源，而且同一个进程中也可以有多个不同的信号量，我们可以通过它们的组合来实现更加复杂的同步需求。因此，从数据结构角度看，需要将一个信号量表加入到进程控制块 PCB 中：

```

1 // os/src/task/process.rs
2
3 pub struct ProcessControlBlockInner {
4 ...
5 pub mutex_list: Vec<Option<Arc<dyn Mutex>>>,
6 pub semaphore_list: Vec<Option<Arc<Semaphore>>>,
7 ...
8 }
```

在内核中，信号量是一个叫做 Semaphore 内核数据结构，定义如下：

```

1 // os/src/sync/semaphore.rs
2
3 pub struct Semaphore {
4 pub inner: UPSafeCell<SemaphoreInner>,
5 }
6
7 pub struct SemaphoreInner {
8 pub count: isize,
9 pub wait_queue: VecDeque<Arc<TaskControlBlock>>,
10 }
11
12 impl Semaphore {
13 pub fn new(res_count: usize) -> Self { ... }
14 pub fn up(&self) { ... }
15 pub fn down(&self) { ... }
16 }
```

其中包括一个整数变量 count（也就是上面信号量基本思路中的整数变量 S）以及保存阻塞在该信号量中的所有线程的阻塞队列 wait\_queue。信号量 Semaphore 支持三种操作：创建 new（带有一个参数 res\_count，也即信号量初始可用资源数量 N）以及 up（也即 V 操作）和 down（也即 P 操作）。相关的系统调用主要是找到当前进程中的指定信号量实例，随后再调用它的这几种方法来实现的，在此不再赘述。于是我们主要看这几种方法是如何实现的：

```

1 // os/src/sync/semaphore.rs
2
```

(下页继续)

(续上页)

```

3 impl Semaphore {
4 pub fn new(res_count: usize) -> Self {
5 Self {
6 inner: unsafe {
7 UPSafeCell::new(SemaphoreInner {
8 count: res_count as isize,
9 wait_queue: VecDeque::new(),
10 })
11 },
12 }
13 }
14
15 pub fn up(&self) {
16 let mut inner = self.inner.exclusive_access();
17 inner.count += 1;
18 if inner.count <= 0 {
19 if let Some(task) = inner.wait_queue.pop_front() {
20 wakeup_task(task);
21 }
22 }
23 }
24
25 pub fn down(&self) {
26 let mut inner = self.inner.exclusive_access();
27 inner.count -= 1;
28 if inner.count < 0 {
29 inner.wait_queue.push_back(current_task().unwrap());
30 drop(inner);
31 block_current_and_run_next();
32 }
33 }
34}

```

new 方法比较简单。而 up 和 down 方法和我们在信号量基本思路中介绍的信号量的第二种实现一致。只需要注意如何使用阻塞-唤醒机制的核心接口 block\_current\_and\_run\_next 和 wakeup\_task 即可。

## 9.5.7 小结

本节我们介绍了相比互斥锁更加灵活强大的同步原语——信号量，并用它解决了条件同步和经典的生产者-消费者问题。但是要看到的是，信号量还是比较复杂的。对于程序员来说开发和阅读代码比较困难，且比较容易出错，对程序员的要求比较高。

## 9.5.8 参考文献

- Dijkstra, Edsger W. Cooperating sequential processes (EWD-123) (PDF). E.W. Dijkstra Archive. Center for American History, University of Texas at Austin. (transcription) (September 1965) <https://www.cs.utexas.edu/users/EWD/transcriptions/EWD01xx/EWD123.html>
- Downey, Allen B. (2016) [2005]. “The Little Book of Semaphores” (2nd ed.). Green Tea Press.
- Leppäjärvi, Jouni (May 11, 2008). “A pragmatic, historically oriented survey on the universality of synchronization primitives” (pdf). University of Oulu, Finland.
- Producer-consumer problem, Wikipedia

## 9.6 条件变量机制

### 9.6.1 本节导读

到目前为止，我们已经了解了操作系统提供的互斥锁和信号量两种同步原语。它们可以用来实现各种同步互斥需求，但是它们比较复杂（特别是信号量），对于程序员的要求较高。如果使用不当，就有可能导致效率低下或者产生竞态条件、死锁或一些不可预测的情况。为了简化编程，避免错误，计算机科学家针对某些情况设计了一种抽象层级较高、更易于使用的同步原语，这就是本节要介绍的条件变量机制。

### 9.6.2 条件变量的背景

首先来看我们需要解决的一类一种同步互斥问题。在信号量一节中提到的条件同步问题的基础上，有的时候我们还需要基于共享资源的状态进行同步。如下面的例子所示：

```

1 static mut A: usize = 0;
2 unsafe fn first() -> ! {
3 A = 1;
4 ...
5 }
6
7 unsafe fn second() -> ! {
8 while A == 0 {
9 // 忙等直到 A==1
10 };
11 //继续执行相关事务
12 }
```

其中，全局变量 A 初始值为 0。假设两个线程并发运行，分别执行 first 和 second 函数，那么这里的同步需求是第二个线程必须等待第一个线程将 A 修改成 1 之后再继续执行。

如何实现这种同步需求呢？首先需要注意到全局变量 A 是一种共享资源，需要用互斥锁保护它的并发访问：

```

1 unsafe fn first() -> ! {
2 mutex_lock(MUTEX_ID);
3 A = 1;
4 mutex_unlock(MUTEX_ID);
5 ...
6 }
7
8 unsafe fn second() -> ! {
9 mutex_lock(MUTEX_ID);
10 while A == 0 { }
11 mutex_unlock(MUTEX_ID);
12 //继续执行相关事务
13 }
```

然而，这种实现并不正确。假设执行 second 的线程先拿到锁，那么它需要等到执行 first 的线程将 A 改成 1 之后才能退出忙等并释放锁。然而，由于线程 second 一开始就拿着锁也不会释放，线程 first 无法拿到锁并修改 A。这样，实际上构成了死锁，线程 first 可能被阻塞，而线程 second 一直在忙等，两个线程无法做任何有意义的事情。

为了解决这个问题，我们需要修改 second 中忙等时锁的使用方式：

```

unsafe fn second() -> ! {
 loop {
```

(下页继续)

(续上页)

```

mutex_lock(MUTEX_ID);
if A == 0 {
 mutex_unlock(MUTEX_ID);
} else {
 mutex_unlock(MUTEX_ID);
 break;
}
}
//继续执行相关事务
}

```

在这种实现中，我们对忙等循环中的每一次对 A 的读取独立加锁。这样的话，当 second 线程发现 first 还没有对 A 进行修改的时候，就可以先将锁释放让 first 可以进行修改。这种实现是正确的，但是基于忙等会浪费大量 CPU 资源和产生不必要的上下文切换。于是，我们可以利用基于阻塞机制的信号量进一步进行改造：

```

1 // user/src/bin/condsync_sem.rs
2
3 unsafe fn first() -> ! {
4 mutex_lock(MUTEX_ID);
5 A = 1;
6 semaphore_up(SEM_ID);
7 mutex_unlock(MUTEX_ID);
8 ...
9 }
10
11 unsafe fn second() -> ! {
12 loop {
13 mutex_lock(MUTEX_ID);
14 if A == 0 {
15 mutex_unlock(MUTEX_ID);
16 semaphore_down(SEM_ID);
17 } else {
18 mutex_unlock(MUTEX_ID);
19 break;
20 }
21 }
22 //继续执行相关事务
23 }

```

按照使用信号量解决条件同步问题的通用做法，我们创建一个  $N = 0$  的信号量，其 ID 为 SEM\_ID。在线程 first 成功修改 A 之后，进行 SEM\_ID 的 up 操作唤醒线程 second；而在线程 second 发现 A 为 0，也即线程 first 还没有完成修改的时候，会进行 SEM\_ID 的 down 操作进入阻塞状态。这样的话，在线程 first 唤醒它之前，操作系统都不会调度到它。

上面的实现中有一个非常重要的细节：请同学思考，second 函数中第 15 行解锁和第 16 行信号量的 down 操作可以交换顺序吗？显然是不能的。如果这样做的话，假设 second 先拿到锁，它发现 A 为 0 就会进行信号量的 down 操作在拿着锁的情况下进入阻塞。这将会导致什么问题？如果想要线程 second 被唤醒，就需要线程 first 修改 A 并进行信号量 up 操作，然而前提条件是线程 first 能拿到锁。这是做不到的，因为线程 second 已经拿着锁进入阻塞状态了，在被唤醒之前都不会将锁释放。于是两个线程都会进入阻塞状态，再一次构成了死锁。可见，这种 **带着锁进入阻塞的情形是我们需要特别小心的**。

从上面的例子可以看出，互斥锁和信号量能实现很多功能，但是它们对于程序员的要求较高，一旦使用不当就很容易出现难以调试的死锁问题。对于这种比较复杂的同步互斥问题，就可以用本节介绍的条件变量来解决。

### 9.6.3 管程与条件变量

我们再回顾一下我们需要解决的一类同步互斥问题：首先，线程间共享一些资源，于是必须使用互斥锁对这些资源进行保护，确保同一时间最多只有一个线程在资源的临界区内；其次，我们还希望能够高效且灵活地支持线程间的条件同步。这应该基于阻塞机制实现：即线程在条件未满足时将自身阻塞，之后另一个线程执行到了某阶段之后，发现条件已经满足，于是将之前阻塞的线程唤醒。刚刚，我们用信号量与互斥锁的组合解决了这一问题，但是这并不是一种通用的解决方案，而是有局限性的：

- 信号量本质上是一个整数，它不足以描述所有类型的等待条件/事件；
- 在使用信号量的时候需要特别小心。比如，up 和 down 操作必须配对使用。而且在和互斥锁组合使用的时候需要注意操作顺序，不然容易导致死锁。

针对这种情况，Brinch Hansen (1973) 和 Hoare (1974) 结合操作系统和 Concurrent Pascal 编程语言，提出了一种高级同步原语，称为 **管程** (Monitor)。管程是一个由过程 (Procedures，是 Pascal 语言中的术语，等同于我们今天所说的函数)、共享变量及数据结构等组成的一个集合，体现了面向对象思想。编程语言负责提供管程的底层机制，程序员则可以根据需求设计自己的管程，包括自定义管程中的过程和共享资源。在管程帮助下，线程可以更加方便、安全、高效地进行协作：线程只需调用管程中的过程即可，过程会对管程中线程间的共享资源进行操作。需要注意的是，管程中的共享资源不允许直接访问，而是只能通过管程中的过程间接访问，这是在编程语言层面对共享资源的一种保护，与 C++/Java 等语言中类的私有成员类似。

下面这段代码是 使用 Concurrent Pascal 语言编写的管程示例的一部分：

```

1 type
2 buffer = Monitor
3 { 管程数据成员定义 }
4 var
5 { 共享资源 }
6 saved: Integer;
7 full : Boolean;
8 { 条件变量 }
9 fullq, emptyq: Queue;
10
11 { 管程过程定义 }
12 procedure entry put(item: Integer);
13 begin
14 if full then
15 { 条件不满足，阻塞当前线程 }
16 delay(fullq);
17 saved := item;
18 full := true;
19 { 条件已经满足，唤醒其他线程 }
20 continue(emptyq);
21 end;

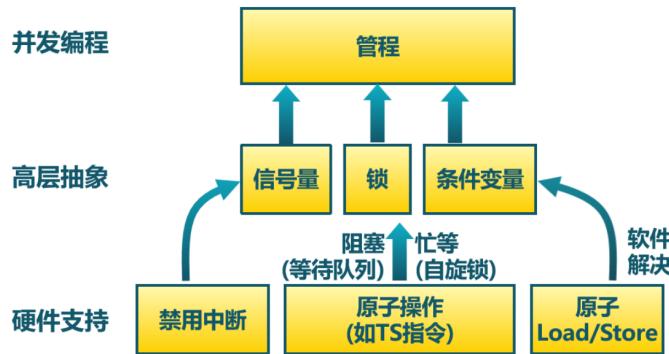
```

那么，管程是如何满足互斥访问和条件同步这两个要求的呢？

- **互斥访问**：区别于 Pascal 语言中的一般过程，管程中的过程使用 entry 关键字（见第 12 行）描述。编程语言保证同一时刻最多只有一个活跃线程在执行管程中的过程，这保证了线程并发调用管程过程的时候能保证管程中共享资源的互斥访问。管程是编程语言的组成部分，编译器知道其特殊性，因此可以采用与其他过程调用不同的方法来处理对管程的调用，比如编译器可以在管程中的每个过程的入口/出口处自动加上互斥锁的获取/释放操作。这一过程对程序员是透明的，降低了程序员的心智负担，也避免了程序员误用互斥锁而出错。
- **条件同步**：管程还支持线程间的条件同步机制，它也是基于阻塞等待的，因而也分成阻塞和唤醒两部分。对于阻塞而言，第 14 行发现条件不满足，当前线程需要等待，于是在第 16 行阻塞当前线程；对于唤醒而言，第 17~18 行的执行满足了某些条件，随后在第 20 行唤醒等待该条件的线程（如果存在）。

在上面的代码片段中，阻塞和唤醒操作分别叫做 delay 和 continue（分别在第 16 和 20 行），它们都是在

数据类型 Queue 上进行的。这里的 Queue 本质上是一个阻塞队列：delay 会将当前线程阻塞并加入到该阻塞队列中；而 continue 会从该阻塞队列中移除一个线程并将其唤醒。今天我们通常将这个 Queue 称为 **条件变量** (Condition Variable)，而将条件变量的阻塞和唤醒操作分别叫做 wait 和 signal。



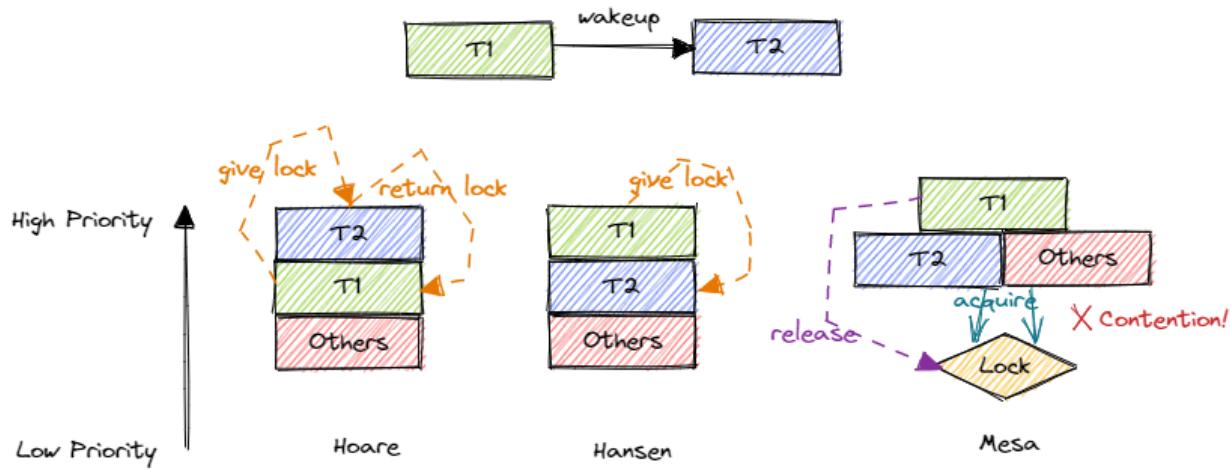
一个管程中可以有多个不同的条件变量，每个条件变量代表多线程并发执行中需要等待的一种特定的条件，并保存所有阻塞等待该条件的线程。注意条件变量与管程过程自带的互斥锁是如何交互的：当调用条件变量的 wait 操作阻塞当前线程的时候，注意到该操作是在管程过程中，因此此时当前线程是持有锁的。经验告诉我们 **不要在持有锁的情况下陷入阻塞**，因此在陷入阻塞状态之前当前线程必须先释放锁；当被阻塞的线程被其他线程使用 signal 操作唤醒之后，需要重新获取到锁才能继续执行，不然的话就无法保证管程过程的互斥访问。因此，站在线程的视角，必须持有锁才能调用条件变量的 wait 操作阻塞自身，且 wait 的功能按顺序分成下述多个阶段，由编程语言保证其原子性：

- 释放锁；
- 阻塞当前线程；
- 当前线程被唤醒之后，重新获取到锁。
- wait 返回，当前线程成功向下执行。

由于互斥锁的存在，signal 操作也不只是简单的唤醒操作。当线程  $T_1$  在执行过程（位于管程过程中）中发现某条件满足准备唤醒线程  $T_2$  的时候，如果直接让线程  $T_2$  继续执行（也位于管程过程中），就会违背管程过程的互斥访问要求。因此，问题的关键是，在  $T_1$  唤醒  $T_2$  的时候， $T_1$  如何处理它正持有的锁。具体来说，根据相关线程的优先级顺序，唤醒操作有这几种语义：

- Hoare 语义：优先级  $T_2 > T_1 > \text{other processes}$ 。也就是说，当  $T_1$  发现条件满足之后，立即通过 signal 唤醒  $T_2$  并 **将锁转交给  $T_2$** ，这样  $T_2$  就能立即继续执行，而  $T_1$  则暂停执行并进入一个 **紧急等待队列**。当  $T_2$  退出管程过程后会将锁交回给紧急等待队列中的  $T_1$ ，从而  $T_1$  可以继续执行。
- Hansen 语义：优先级  $T_1 > T_2 > \text{other processes}$ 。即  $T_1$  发现条件满足之后，先继续执行，直到退出管程之前再使用 signal 唤醒并 **将锁转交给  $T_2$** ，于是  $T_2$  可以继续执行。注意在 Hansen 语义下，signal 必须位于管程过程末尾。
- Mesa 语义：优先级  $T_1 > T_2 = \text{other processes}$ 。即  $T_1$  发现条件满足之后，就可以使用 signal 唤醒  $T_2$ ，但是并 **不会将锁转交给  $T_2$** 。这意味着在  $T_1$  退出管程过程释放锁之后， $T_2$  还需要和其他线程竞争，直到抢到锁之后才能继续执行。

这些优先级顺序如下图所示：



可以看出, Hoare 和 Hansen 语义的区别在于  $T_1$  和  $T_2$  的优先级顺序不同。Hoare 语义认为被唤醒的线程应当立即执行, 而 Hansen 语义则认为应该优先继续执行当前线程。二者的相同之处在于它们都将锁直接转交给唤醒的线程, 也就保证了  $T_2$  一定紧跟着  $T_1$  回到管程过程中, 于是在  $T_2$  被唤醒之后其等待的条件一定是成立的 (因为  $T_1$  和  $T_2$  中间没有其他线程), 因此没有必要重复检查条件是否成立就可以向下执行。相对的, Mesa 语义中  $T_1$  就不会将锁转交给  $T_2$ , 而是将锁释放让  $T_2$  和其他同优先级的线程竞争。这样,  $T_1$  和  $T_2$  之间可能存在其他线程, 这些线程的执行会影响到共享资源, 以至于  $T_2$  抢到锁继续执行的时候, 它所等待的条件又已经不成立了。所以, 在 Mesa 语义下, `wait` 操作返回之时不见得线程等待的条件一定成立, 有必要重复检查确认之后再继续执行。

#### 注解: 条件等待应该使用 `if/else` 还是 `while`?

在使用 `wait` 操作进行条件等待的时候, 通常有以下两种方式:

```
// 第一种方法, 基于 if/else
if (!condition) {
 wait();
} else {
 ...
}

// 第二种方法, 基于 while
while (!condition) {
 wait();
}
```

如果基于 `if/else` 的话, 其假定了 `wait` 返回之后条件一定已经成立, 于是不再做检查直接向下执行。而基于 `while` 循环的话, 则是无法确定 `wait` 返回之后条件是否成立, 于是将 `wait` 包裹在一个 `while` 循环中重复检查直到条件成立。

根据上面的分析可以, 如果条件变量是 Mesa 语义, 则必须将 `wait` 操作放在 `while` 循环中; 如果是 Hoare/Hansen 语义, 则使用 `if/else` 或者 `while` 均可。在不能确定条件变量为何种语义的情况下, 应使用 `while` 循环, 这样保证不会出错。

一般情况下条件变量会使用 Hansen 语义, 因为它在概念上更简单, 并且更容易实现。其实除了条件变量之外, 这几种语义也作用于其他基于阻塞-唤醒机制的同步原语。例如, 前两节的互斥锁和信号量就是基于 Hansen 语义实现的, 有兴趣的同学可以回顾一下。在操作系统中 Mesa 语义也比较常用。

早期提出的管程是基于 Concurrent Pascal 语言来设计的, 其他语言, 如 C 和 Rust 等, 并没有在语言上支持这种机制。对此, 我们的做法是从管程中将比较通用的同步原语——条件变量抽取出来, 然后再将其和互斥锁

组合使用（手动加入加锁/解锁操作代替编译器），以这种方式模拟原始的管程机制。在目前的 C 语言应用开发中，实际上也是这样做的。

## 9.6.4 条件变量系统调用

于是，我们新增条件变量相关系统调用如下：

```
1 /// 功能：为当前进程新增一个条件变量。
2 /// 返回值：假定该操作必定成功，返回创建的条件变量的 ID 。
3 /// syscall ID : 1030
4 pub fn sys_condvar_create() -> isize;
5
6 /// 功能：对当前进程的指定条件变量进行 signal 操作，即
7 /// 唤醒一个在该条件变量上阻塞的线程（如果存在）。
8 /// 参数：condvar_id 表示要操作的条件变量的 ID 。
9 /// 返回值：假定该操作必定成功，返回 0 。
10 /// syscall ID : 1031
11 pub fn sys_condvar_signal(condvar_id: usize) -> isize;
12
13 /// 功能：对当前进程的指定条件变量进行 wait 操作，分为多个阶段：
14 /// 1. 释放当前线程持有的一把互斥锁；
15 /// 2. 阻塞当前线程并将其加入指定条件变量的阻塞队列；
16 /// 3. 直到当前线程被其他线程通过 signal 操作唤醒；
17 /// 4. 重新获取当前线程之前持有的锁。
18 /// 参数：mutex_id 表示当前线程持有的互斥锁的 ID，而
19 /// condvar_id 表示要操作的条件变量的 ID 。
20 /// 返回值：假定该操作必定成功，返回 0 。
21 /// syscall ID : 1032
22 pub fn sys_condvar_wait(condvar_id: usize, mutex_id: usize) -> isize;
```

这里，条件变量也被视作进程内的一种资源，进程内的不同条件变量使用条件变量 ID 区分。注意 wait 操作不仅需要提供条件变量的 ID，还需要提供线程目前持有的锁的 ID。需要注意的是，**我们内核中实现的条件变量是 Mesa 语义的**。

## 9.6.5 条件变量的使用方法

### 条件同步问题

下面展示了如何使用条件变量解决本节开头提到的条件同步问题：

```
1 // user/src/bin/condsync_condvar.rs
2
3 const CONDVAR_ID: usize = 0;
4 const MUTEX_ID: usize = 0;
5
6 unsafe fn first() -> ! {
7 sleep(10);
8 println!("First work, Change A --> 1 and wakeup Second");
9 mutex_lock(MUTEX_ID);
10 A = 1;
11 condvar_signal(CONDVAR_ID);
12 mutex_unlock(MUTEX_ID);
13 exit(0)
14 }
```

(下页继续)

(续上页)

```

15
16 unsafe fn second() -> ! {
17 println!("Second want to continue, but need to wait A=1");
18 mutex_lock(MUTEX_ID);
19 while A == 0 {
20 println!("Second: A is {}", A);
21 condvar_wait(CONDVAR_ID, MUTEX_ID);
22 }
23 println!("A is {}, Second can work now", A);
24 mutex_unlock(MUTEX_ID);
25 exit(0)
26 }
27
28 #[no_mangle]
29 pub fn main() -> i32 {
30 // create condvar & mutex
31 assert_eq!(condvar_create() as usize, CONDVAR_ID);
32 assert_eq!(mutex_blocking_create() as usize, MUTEX_ID);
33 ...
34 }

```

第 31 和 32 行我们分别创建要用到的条件变量和互斥锁。在 second 中，首先有一层互斥锁保护，然后由于条件变量是 Mesa 语义的，所以我们需要使用 while 循环进行等待，不符合条件调用 condvar\_wait 阻塞自身的时候还要给出当前持有的互斥锁的 ID；在 first 中，最外层同样有互斥锁保护。在修改完成之后只需调用 condvar\_signal 即可唤醒执行 second 的线程。

在使用条件变量的时候需要特别注意 [唤醒丢失](#) 问题。也就是说和信号量不同，如果调用 signal 的时候没有任何线程在条件变量的阻塞队列中，那么这次 signal 不会有任何效果，这次唤醒也不会被记录下来。对于这个例子来说，我们在 first 中还会修改 A，因此如果 first 先执行，即使其中的 signal 没有任何效果，之后执行 second 的时候也会发现条件已经满足而不必进入阻塞。

## 同步屏障问题

接下来我们看一个有趣的问题。假设有 3 个线程，每个线程都执行如下 thread\_fn 函数：

```

// user/src/bin/barrier_fail.rs

fn thread_fn() {
 for _ in 0..300 { print!("a"); }
 for _ in 0..300 { print!("b"); }
 for _ in 0..300 { print!("c"); }
 exit(0)
}

```

可以将 thread\_fn 分成打印字符 a、打印字符 b 和打印字符 c 这三个阶段。考虑这样一种同步需求：即在阶段间设置 **同步屏障**，只有所有的线程都完成上一阶段之后，这些线程才能够进入下一阶段。也就是说，如果有线程更早完成了一个阶段，那么它需要等待其他较慢的线程也完成这一阶段才能进入下一阶段。最后的执行结果应该是所有的 a 被打印出来，然后是所有的 b，最后是所有的 c。同学们在向下阅读之前可以思考如何用我们学过的同步原语来实现这种同步需求。

这里给出基于互斥锁和条件变量的一种参考实现：

```

1 // user/src/bin/barrier_condvar.rs
2
3 const THREAD_NUM: usize = 3;

```

(下页继续)

(续上页)

```

4
5 struct Barrier {
6 mutex_id: usize,
7 condvar_id: usize,
8 count: UnsafeCell<usize>,
9 }
10
11 impl Barrier {
12 pub fn new() -> Self {
13 Self {
14 mutex_id: mutex_create() as usize,
15 condvar_id: condvar_create() as usize,
16 count: UnsafeCell::new(0),
17 }
18 }
19 pub fn block(&self) {
20 mutex_lock(self.mutex_id);
21 let count = self.count.get();
22 // SAFETY: Here, the accesses of the count is in the
23 // critical section protected by the mutex.
24 unsafe { *count = *count + 1; }
25 if unsafe { *count } == THREAD_NUM {
26 condvar_signal(self.condvar_id);
27 } else {
28 condvar_wait(self.condvar_id, self.mutex_id);
29 condvar_signal(self.condvar_id);
30 }
31 mutex_unlock(self.mutex_id);
32 }
33 }
34
35 unsafe impl Sync for Barrier {}
36
37 lazy_static! {
38 static ref BARRIER_AB: Barrier = Barrier::new();
39 static ref BARRIER_BC: Barrier = Barrier::new();
40 }
41
42 fn thread_fn() {
43 for _ in 0..300 { print!("a"); }
44 BARRIER_AB.block();
45 for _ in 0..300 { print!("b"); }
46 BARRIER_BC.block();
47 for _ in 0..300 { print!("c"); }
48 exit(0)
49 }

```

我们自定义一种 Barrier 类型，类似于前面讲到的管程。这里的关键在于 Barrier::block 方法。在拿到锁之后，首先检查 count 变量。count 变量是一种共享资源，记录目前有多少线程阻塞在同步屏障中。如果所有的线程都已经到了，那么当前线程就可以唤醒其中一个；否则就需要先阻塞，在被唤醒之后再去唤醒一个其他的。最终来看会形成一条唤醒链。

有兴趣的同学可以思考如何用其他同步原语来解决这个问题。

## 9.6.6 实现条件变量

最后我们来看在我们的内核中条件变量是如何实现的。首先还是将条件变量作为一种资源加入到进程控制块中：

```
1 // os/src/task/process.rs
2
3 pub struct ProcessControlBlockInner {
4 ...
5 pub mutex_list: Vec<Option<Arc<dyn Mutex>>>,
6 pub semaphore_list: Vec<Option<Arc<Semaphore>>>,
7 pub condvar_list: Vec<Option<Arc<Condvar>>>,
8 }
```

条件变量 Condvar 在数据结构层面上比信号量还简单，只有一个阻塞队列 `wait_queue`（因此再次强调小心唤醒丢失问题）：

```
1 // os/src/sync/condvar.rs
2
3 pub struct Condvar {
4 pub inner: UPSafeCell<CondvarInner>,
5 }
6
7 pub struct CondvarInner {
8 pub wait_queue: VecDeque<Arc<TaskControlBlock>>,
9 }
```

条件变量相关的系统调用也是直接调用 Condvar 的同名方法实现的，因此这里我们主要看 Condvar 的方法：

```
1 // os/src/sync/condvar.rs
2
3 impl Condvar {
4 pub fn new() -> Self {
5 Self {
6 inner: unsafe {
7 UPSafeCell::new(CondvarInner {
8 wait_queue: VecDeque::new(),
9 })
10 },
11 }
12 }
13
14 pub fn signal(&self) {
15 let mut inner = self.inner.exclusive_access();
16 if let Some(task) = inner.wait_queue.pop_front() {
17 wakeup_task(task);
18 }
19 }
20
21 pub fn wait(&self, mutex: Arc<dyn Mutex>) {
22 mutex.unlock();
23 let mut inner = self.inner.exclusive_access();
24 inner.wait_queue.push_back(current_task().unwrap());
25 drop(inner);
26 block_current_and_run_next();
27 mutex.lock();
28 }
29}
```

(下页继续)

(续上页)

```
28 }
29 }
```

- 第 4 行的 new 创建一个空的阻塞队列；
- 第 14 行的 signal 从阻塞队列中移除一个线程并调用唤醒原语 wakeup\_task 将其唤醒。注意如果此时阻塞队列为空则此操作不会有任何影响；
- 第 21 行的 wait 接收一个当前线程持有的锁作为参数。首先将锁释放，然后将当前线程挂在条件变量阻塞队列中，之后调用阻塞原语 block\_current\_and\_run\_next 阻塞当前线程。在被唤醒之后还需要重新获取锁，这样 wait 才能返回。

## 9.6.7 参考文献

- Hansen, Per Brinch (1993). “Monitors and concurrent Pascal: a personal history” . HOPL-II: The second ACM SIGPLAN conference on History of programming languages. History of Programming Languages. New York, NY, USA: ACM. pp. 1–35. doi:10.1145/155360.155361. ISBN 0-89791-570-4.
- Monitor, Wikipedia
- Concurrent Pascal, Wikipedia

## 9.7 并发中的问题

### 9.7.1 本节导读

应用程序员在开发并发应用的过程中，经常由于各种 不小心编写出各种并发缺陷。并发缺陷有很多种，典型的主要有三类：互斥缺陷、同步缺陷和死锁缺陷。了解这些缺陷的模式是写出健壮、正确并发程序的关键。

### 9.7.2 互斥缺陷

互斥缺陷也称为违反原子性缺陷，在并发应用程序中对共享变量没进行合理的保护是导致出现这类缺陷的一个重要原因。下面是一个简单的例子：

```
1 static mut A: usize = 0;
2 //... other code
3 unsafe fn thr1() -> ! {
4 if (A == 0) {
5 println!("thr1: A is Zero --> {}", A);
6 }
7 //... other code
8 }
9 unsafe fn thr2() -> ! {
10 A = A+1;
11 println!("thr2: A is One --> {}", A);
12 }
```

A 是共享变量。粗略地看，可以估计执行流程为：第一个线程 thr1 检查 A 的值，如果为 0，则显示 “`thr1: A is Zero -> 0`”；第二个线程 thr2 将 A 的值由 0 设置为 1（因为除了初始化之外没有其他地方修改了 A 的值），并显示 “`thr2: A is One -> 1`”。但如果线程 thr1 执行完第 4 行代码，准备执行第 5 行代码前发生了线程切换，开始执行线程 th2；当线程 thr2 完成第 10 行后，操作系统又切换回线程 thr1 继续执行，那么线程 thr1 就会输出 “`thr1: A is Zero -> 1`” 这样的奇怪结果。

这里出现问题的根源是线程在对共享变量进行访问时，违反了临界区的互斥性（原子性）原则。解决这样的问题需要给共享变量的访问加锁，确保每个线程访问共享变量时，都持有锁，修改后的代码如下：

```

1 static mut A: usize = 0;
2 //... other code
3 unsafe fn thr1() -> ! {
4 mutex.lock();
5 if (A == 0) {
6 println!("thr1: A is Zero --> {}", A);
7 }
8 mutex.unlock();
9 //... other code
10 }
11 unsafe fn thr2() -> ! {
12 mutex.lock();
13 A = A+1;
14 println!("thr2: A is One --> {}", A);
15 mutex.unlock();
16 }
```

这种问题如果能发现，那么修复相对比较简单，即在对共享变量进行访问的代码区域前后加上请求锁和释放锁的操作。但主要的问题是发现缺陷比较难，特别是代码量比较大，代码的控制逻辑比较复杂的情况。

### 9.7.3 同步缺陷

同步缺陷也称为违反顺序缺陷，在并发应用程序中对共享变量访问的先后顺序的可能性没有充分分析是导致出现这类缺陷的一个重要原因。下面是一个简单的例子：

```

1 static mut A: usize = 0;
2 ...
3 unsafe fn thr1() -> ! {
4 ... //在某种情况下会休眠
5 A = 1;
6 ...
7 }
8 unsafe fn thr2() -> ! {
9 if A==1 {
10 println!("Correct");
11 }else{
12 println!("Panic");
13 }
14 }
15 pub fn main() -> i32 {
16 let mut v = Vec::new();
17 v.push(thread_create(thr1 as usize, 0));
18 sleep(10);
19 ...
20 v.push(thread_create(thr2 as usize, 0));
21 ...
22 }
```

A是共享变量。粗略地看，可以估计执行流程为：线程 thr1 先被创建，等了 10ms 后，线程 thr2 再被创建。一般情况下，这就导致了 thr1 先于 thr2 执行，即第 5 行会先于第 9 行执行，得到预期的结果。但可能出现一种执行情况：线程 thr1 在执行第 5 句前，由于某种原因进入了休眠，导致线程 thr2 执行第 9 行在前，线程 th1 执行第 5 行在后，导致获得非预期的错误结果。

这里出现问题的根源是线程在对共享变量进行访问时，违反了临界区的预期顺序原则。解决这样的问题需要

给线程的相关代码位置加上同步操作（如通过信号量或条件变量等），确保线程间的执行顺序符合预期，修改后的代码如下：

```

1 static mut A: usize = 0;
2 semaphore.value = 0; //信号量初值为0
3 unsafe fn thr1() -> ! {
4 ... //在某种情况下会休眠
5 A = 1;
6 semaphore.up();
7 ...
8 }
9 unsafe fn thr2() -> ! {
10 semaphore.down(); // 需要等待 semaphore.up() 的唤醒
11 if A==1 {
12 println!("Correct");
13 }else{
14 println!("Panic");
15 }
16 }
17 pub fn main() -> i32 {
18 let mut v = Vec::new();
19 v.push(thread_create(thr1 as usize, 0));
20 sleep(10);
21 ...
22 v.push(thread_create(thr2 as usize, 0));
23 ...
24 }
```

这种问题如果能发现，那么修复相对也比较简单，即在线程的代码区域设置合理的同步操作，让线程间的执行顺序符合预期。但主要的问题还是发现缺陷比较难，特别是代码量比较大，代码的控制逻辑比较复杂的情况。

也许有同学说，这样的错误缺陷很容易发现呀，只要开发者在编写时注意一下，就可以了。但其实不尽然，因为我们这里给出的是一个刻意简化的例子，在实际的并发应用程序中，由于代码量远大于这个例子，控制逻辑会有循环、跳转、函数调用等，涉及到的共享变量的数量、访问操作，以及与互斥/同步操作的关系等会错综复杂，难以一下子就能一目了然地分析清楚，导致很容易出现互斥和同步缺陷。

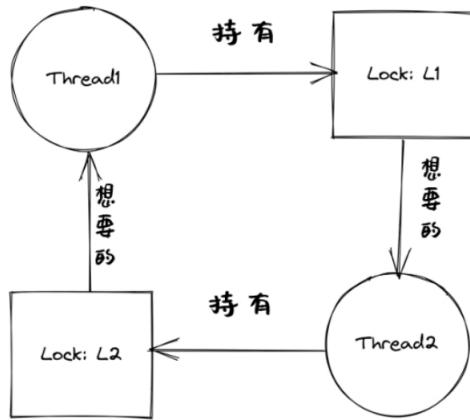
#### 9.7.4 死锁缺陷

除了上面的两类并发缺陷，还有一类导致程序无法正常执行的并发缺陷—死锁（Dead lock）。在并发应用中，经常需要线程排他性地访问若干种资源。大部分死锁都和不可抢占的资源相关，这里把线程需要申请获取、排他性使用和释放的对象称为资源（resource）。需要互斥访问的共享变量就是一种资源。操作系统通过互斥锁、信号量或条件变量等同步互斥机制，能授权一个线程（临时）具有排他地访问某一种资源的能力。下面是一个死锁的例子：

```

1 unsafe fn thr1() -> ! {
2 mutex1.lock();
3 mutex2.lock();
4 ...
5 }
6 unsafe fn thr2() -> ! {
7 mutex2.lock();
8 mutex1.lock();
9 ...
10 }
```

当线程 thr1 持有锁 mutex1, 正在等待另外一个锁 mutex2, 而线程 thr2 持有锁 mutex2, 正在等待另外一个锁 mutex1 时, 死锁就产生了。



对于这个代码, 可以很容易避免死锁:

```

1 unsafe fn thr1() -> ! {
2 mutex1.lock();
3 mutex2.lock();
4 ...
5 }
6 unsafe fn thr2() -> ! {
7 mutex1.lock();
8 mutex2.lock();
9 ...
10 }

```

只要线程 thr1 和线程 thr2 都用相同的请求锁顺序, 就不会发生死锁了。但这与上面的分析一样, 对于实际的复杂程序, 发现死锁就是一个很费劲的事情。目前计算机科学家对死锁的研究比较深入, 指出了死锁产生的四个必要条件:

- 互斥: 线程互斥地访问资源。
- 持有并等待: 线程已持有了部分资源, 同时又在等待其他资源。
- 非抢占: 线程已持有的资源不能被抢占。
- 循环等待: 线程之间存在一个资源持有/等待的环, 环上每个线程都持有部分资源, 而这部分资源又是下一个线程在等待申请的资源。

## 死锁预防

如果线程间产生了死锁, 那么上面四个条件一定会发生。换个角度来看, 如果这四个条件中的任意一个没有满足, 死锁就不会产生。

一个比较实用的预防死锁的方法是打破循环等待, 具体做法就是给锁/访问的资源进行排序, 要求每个线程都按照排好的顺序依次申请锁和访问资源。这种顺序性避免了循环等待, 也就不会产生死锁。

## 死锁避免

计算机科学家 Dijkstra 在 1965 年为 THE 操作系统设计提出的一种死锁避免（avoidance）的调度算法，称为银行家算法（banker's algorithm）算法的核心是判断满足线程的资源请求是否会导致整个系统进入不安全状态。如果是，就拒绝线程的资源请求；如果满足请求后系统状态仍然是安全的，就分配资源给线程。

状态是安全的，是指存在一个资源分配/线程执行序列使得所有的线程都能获取其所需资源并完成线程的工作。如果找不到这样的资源分配/线程执行序列，那么状态是不安全的。这里把线程的执行过程简化为：申请资源、释放资源的一系列资源操作。这意味着线程执行完毕后，会释放其占用的所有资源。

我们需要知道，不安全状态并不等于死锁，而是指有死锁的可能性。安全状态和不安全状态的区别是：从安全状态出发，操作系统通过调度线程执行序列，能够保证所有线程都能完成，一定不会出现死锁；而从不安全状态出发，就没有这样的保证，可能出现死锁。



## 银行家算法的数据结构

为了描述操作系统中可利用的资源、所有线程对资源的最大需求、系统中的资源分配，以及所有线程还需要多少资源的情况，需要定义对应的四个数据结构：

- 可利用资源向量 Available：含有  $m$  个元素的一维数组，每个元素代表可利用的某一类资源的数目，其初值是该类资源的全部可用数目，其值随该类资源的分配和回收而动态地改变。Available[j] =  $k$ ，表示第  $j$  类资源的可用数量为  $k$ 。
- 最大需求矩阵 Max： $n * m$  矩阵，表示  $n$  个线程中，每个线程对  $m$  类资源的最大需求量。Max[i,j] =  $h$ ，表示线程  $i$  需要第  $j$  类资源的最大数量为  $h$ 。
- 分配矩阵 Allocation： $n * m$  矩阵，表示每类资源已分配给每个线程的资源数。Allocation[i,j] =  $g$ ，则表示线程  $i$  当前已分得第  $j$  类资源的数量为  $g$ 。
- 需求矩阵 Need： $n * m$  的矩阵，表示每个线程还需要的各类资源数量。Need[i,j] =  $d$ ，则表示线程  $i$  还需要第  $j$  类资源的数量为  $d$ 。

上述三个矩阵间存在如下关系：

$$\text{Need}[i,j] = \text{Max}[i,j] - \text{Allocation}[i,j]$$

## 银行家算法的步骤

设 Request 是线程的请求资源矩阵, 如果  $Request[i,j] = t$ , 表示线程  $thr[i]$  需要  $t$  个第  $j$  类型的资源。当线程  $thr[i]$  发出资源请求后, 操作系统的银行家算法按下述步骤执行:

1. 如果  $Request[i,j] \leq Need[i,j]$ , 则转步骤 2; 否则出错, 因为线程所需的资源数已超过它所宣布的最大值。
2. 如果  $Request[i,j] \leq Available[j]$ , 则转步骤 3; 否则, 表示尚无足够资源, 线程  $thr[i]$  进入等待状态。
3. 操作系统试着把资源分配给线程  $thr[i]$ , 并修改下面数据结构中的值:

```

1 Available[j] = Available[j] - Request[i,j];
2 Allocation[i,j] = Allocation[i,j] + Request[i,j];
3 Need[i,j] = Need[i,j] - Request[i,j];

```

4. 操作系统执行安全性检查算法, 检查此次资源分配后系统是否处于安全状态。若安全, 则实际将资源分配给线程  $thr[i]$ ; 否则不进行资源分配, 让线程  $thr[i]$  等待。

## 安全性检查算法

安全性检查算法如下:

1. 设置两个向量: 工作向量 Work, 表示操作系统可提供给线程继续运行所需的各类资源数目, 它含有  $m$  个元素, 初始时,  $Work = Available$ ; 结束向量 Finish, 表示系统是否有足够的资源分配给线程, 使之运行完成。初始时  $Finish[0..n-1] = false$ , 表示所有线程都没结束; 当有足够资源分配给线程时, 设置  $Finish[i] = true$ 。
2. 从线程集合中找到一个能满足下述条件的线程

```

1 Finish[i] == false;
2 Need[i,j] <= Work[j];

```

若找到, 执行步骤 3, 否则, 执行步骤 4。

3. 当线程  $thr[i]$  获得资源后, 可顺利执行, 直至完成, 并释放出分配给它的资源, 故应执行:

```

1 Work[j] = Work[j] + Allocation[i,j];
2 Finish[i] = true;

```

跳转回步骤 2

4. 如果  $Finish[0..=n-1]$  都为  $true$ , 则表示系统处于安全状态; 否则表示系统处于不安全状态。

通过操作系统调度, 如银行家算法来避免死锁不是广泛使用的通用方案。因为从线程执行的一般情况上看, 银行家算法需要提前获知线程总的资源申请量, 以及未来的每一次请求, 而这些请求对于一般线程而言在运行前是不可知或随机的。另外, 即使在某些特殊情况下, 可以提前知道线程的资源申请量等信息, 多重循环的银行家算法开销也是很大的, 不适合于对性能要求很高的操作系统中。

## 9.8 练习

### 9.8.1 课后练习

#### 编程题

在 Linux 环境中，基于 pthread 线程，有一系列的系统调用实现对应用程序的线程间同步互斥的支持。

信号量是一种特殊的变量，可用于线程同步。它只取自然数值，并且只支持两种操作：P(SV)：如果信号量 SV 大于 0，将它减一；如果 SV 值为 0，则挂起该线程。V(SV)：如果有其他进程因为等待 SV 而挂起，则唤醒，然后将 SV+1；否则直接将 SV+1。其系统调用为：

- `sem_wait(sem_t *sem)`：以原子操作的方式将信号量减 1，如果信号量值为 0，则 `sem_wait` 将被阻塞，直到这个信号量具有非 0 值。
- `sem_post(sem_t *sem)`：以原子操作将信号量值 +1。当信号量大于 0 时，其他正在调用 `sem_wait` 等待信号量的线程将被唤醒。

互斥量：互斥量又称互斥锁，主要用于线程互斥，不能保证按序访问，可以和条件锁一起实现同步。当进入临界区时，需要获得互斥锁并且加锁；当离开临界区时，需要对互斥锁解锁，以唤醒其他等待该互斥锁的线程。其主要的系统调用如下：

- `pthread_mutex_init`：初始化互斥锁
- `pthread_mutex_destroy`：销毁互斥锁
- `pthread_mutex_lock`：以原子操作的方式给一个互斥锁加锁，如果目标互斥锁已经被上锁，`pthread_mutex_lock` 调用将阻塞，直到该互斥锁的占有者将其解锁。
- `pthread_mutex_unlock`：以一个原子操作的方式给一个互斥锁解锁。

条件变量：条件变量，又称条件锁，用于在线程之间同步共享数据的值。条件变量提供一种线程间通信机制：当某个共享数据达到某个值时，唤醒等待这个共享数据的一个/多个线程。即，当某个共享变量等于某个值时，调用 `signal/broadcast`。此时操作共享变量时需要加锁。其主要的系统调用如下：

- `pthread_cond_init`：初始化条件变量
- `pthread_cond_destroy`：销毁条件变量
- `pthread_cond_signal`：唤醒一个等待目标条件变量的线程。哪个线程被唤醒取决于调度策略和优先级。
- `pthread_cond_wait`：等待目标条件变量。需要一个加锁的互斥锁确保操作的原子性。该函数中在进入 `wait` 状态前首先进行解锁，然后接收到信号后会再加锁，保证该线程对共享资源正确访问。

1. \*\* 在 Linux 环境下，请用信号量实现哲学家就餐的多线程应用程序。
2. \*\* 在 Linux 环境下，请用互斥锁和条件变量实现哲学家就餐的多线程应用程序。
3. \*\* 在 Linux 环境下，请建立一个多线程的模拟资源分配管理库，可通过银行家算法来避免死锁。
4. \*\* 扩展内核功能，实现读者优先的读写信号量。
5. \*\* 扩展内核功能，实现写者优先的读写信号量。
6. \*\*\* 扩展内核功能，在内核中支持内核线程。
7. \*\*\* 进一步扩展内核功能，在内核线程中支持同步互斥机制，实现内核线程用的 mutex, semaphore, cond-var。
8. \*\*\* 扩展内核功能，实现多核支持下的同步互斥机制。
9. \*\*\* 解决优先级反转问题：实现 RM 实时调度算法，设计优先级反转的实例，实现优先级天花板和优先级继承方法。

## 问答题

1. \* 什么是并行？什么是并发？
2. \* 为了创造临界区，单核处理器上可以【关中断】，多核处理器上需要使用【自旋锁】。请回答下列问题：
  - 多核上可不可以只用【关中断】？
  - 单核上可不可以只用【自旋锁】？
  - 多核上的【自旋锁】是否需要同时【关中断】？
  - [进阶] 假如某个锁不会在中断处理函数中被访问，是否还需要【关中断】？
3. \*\*Linux 的多线程应用程序使用的锁（例如 `pthread_mutex_t`）不是自旋锁，当上锁失败时会切换到其它进程执行。分析它和自旋锁的优劣，并说明为什么它不用自旋锁？
4. \*\*\* 程序在运行时具有两种性质：safety: something bad will never happen; liveness: something good will eventually occur. 分析并证明 Peterson 算法的 safety 和 liveness 性质。
5. \* 信号量结构中的整数分别为 +n、0、-n 的时候，各自代表什么状态或含义？
6. \*\* 考虑如下信号量实现代码：

```
class Semaphore {
 int sem;
 WaitQueue q;
}
Semaphore:::P() {
 sem--;
 if(sem < 0) {
 Add this thread to q.
 block.
 }
}
Semaphore:::V() {
 sem++;
 if(sem <= 0) {
 t = Remove a thread from q;
 wakeup(t);
 }
}
```

假如 P 操作或 V 操作不是原子操作，会出现什么问题？举一个例子说明。上述代码能否运行在用户态？上面代码的原子性是如何保证的？

7. \*\* 条件变量的 Wait 操作为什么必须关联一个锁？
8. \*\* 下面是条件变量的 wait 操作实现伪代码：

```
Condvar:::wait(lock) {
 Add this thread to q.
 lock.unlock();
 schedule();
 lock.lock();
}
```

如果改成下面这样：

```
Condvar:::wait() {
 Add this thread to q.
```

(下页继续)

(续上页)

```

 schedule();
}
lock.unlock();
condvar.wait();
lock.lock();

```

会出现什么问题？举一个例子说明。

9. \* 死锁的必要条件是什么？
10. \* 什么是死锁预防，举例并分析。
11. \*\* 描述银行家算法如何判断安全性。

## 9.8.2 实验练习

实验练习包括实践作业和问答作业两部分。

### 编程作业

#### 银行家算法——分数更新

---

**注解：**本实验为用户态实验，请在 Linux 环境下完成。

---

**背景：**在智能体大赛平台 [Saiblo](#) 网站上每打完一场双人天梯比赛后需要用 ELO 算法更新双方比分。由于 [Saiblo](#) 的评测机并发性很高，且 ELO 算法中的分值变动与双方变动前的分数有关，因此更新比分前时必须先为两位选手加锁。

**作业：**请模拟一下上述分数更新过程，简便起见我们简化为有  $p$  位选手参赛（编号  $[0, p)$  或  $[1, p]$ ），初始分值为 1000 分，有  $m$  个评测机线程（生产者）给出随机的评测结果（两位不同选手的编号以及胜负结果，结果可能为平局），有  $n$  个 worker 线程（消费者）获取结果队列并更新数据库（全局变量等共享数据）记录的分数。 $m$  个评测机各自模拟  $k$  场对局结果后结束线程，全部对局比分更新完成后主线程打印每位选手最终成绩以及所有选手分数之和。

上述参数  $p$ 、 $m$ 、 $n$ 、 $k$  均为可配置参数（命令行传参或程序启动时从 `stdin` 输入）。

简便起见不使用 ELO 算法，简化更新规则为：若不为平局，当胜者分数  $\geq$  败者分数时胜者 +20，败者 -20，否则胜者 +30，败者 -30；若为平局，分高者 -10，分低者 +10（若本就同分保持则不变）。

**消费者核心部分可参考如下伪码：** 获取选手 A 的锁获取选手 B 的锁更新 A、B 分数睡眠 1ms（模拟数据库更新延时）释放选手 B 的锁释放选手 A 的锁

**tips:**

- 由于 ELO 以及本题中给出的简化更新算法均为零和算法，因此出现冲突后可以从所有选手分数之和明显看出来，正确处理时它应该永远为  $1000p$
- 将一个 worker 线程看作哲学家，将 worker 正在处理的一场对局的两位选手看作两根筷子，则得到了经典的哲学家就餐问题

## 实现 eventfd

在 Linux 中有一种用于事件通知的文件描述符，称为 eventfd。其核心是一个 64 位无符号整数的计数器，在非信号量模式下，若计数器值不为零，则 `read` 函数会从中读出计数值并将其清零，否则读取失败；`write` 函数将缓冲区中的数值加入到计数器中。在信号量模式下，若计数器值非零，则 `read` 操作将计数值减一，并返回 1；`write` 将计数值加一。我们将实现一个新的系统调用：`sys_eventfd2`。

### eventfd：

- syscall ID: 290
- 功能：创建一个 eventfd，`eventfd` 标准接口。
- C 接口: `int eventfd(unsigned int initval, int flags)`
- Rust 接口: `fn eventfd(initval: u32, flags: i32) -> i32`
- 参数：
  - `initval`: 计数器的初值。
  - **flags**: 可以设置为 0 或以下两个 flag 的任意组合 (按位或):
    - \* `EFD_SEMAPHORE` (1)：设置该 flag 时，将以信号量模式创建 eventfd。
    - \* `EFD_NONBLOCK` (2048)：若设置该 flag，对 eventfd 读写失败时会返回 -2，否则将阻塞等待直至读或写操作可执行为止。
- 说明：
  - 通过 `write` 写入 eventfd 时，缓冲区大小必须为 8 字节。
  - 进程 `fork` 时，子进程会继承父进程创建的 eventfd，且指向同一个计数器。
- 返回值：如果出现了错误则返回 -1，否则返回创建成功的 eventfd 编号。
- 可能的错误
  - flag 不合法。
  - 创建的文件描述符数量超过进程限制

---

**注解：**还有一个 `sys_eventfd` 系统调用 (调用号 284)，与 `sys_eventfd2` 的区别在于前者不支持传入 `flags`。

Linux 中的原生异步 IO 接口 libaio 就使用了 eventfd 作为内核完成 IO 操作之后通知应用程序的机制。

---

## 实验要求

- 完成分支: ch8-lab
- 实验目录要求不变。
- 通过所有测例。

## 问答作业

无

## 实验练习的提交报告要求

- 简单总结本次实验与上个实验相比你增加的东西。(控制在 5 行以内, 不要贴代码)
- 完成问答问题
- (optional) 你对本次实验设计及难度的看法。

## 9.9 练习参考答案

### 9.9.1 课后练习

#### 编程题

在 Linux 环境中, 基于 pthread 线程, 有一系列的系统调用实现对应用程序的线程间同步互斥的支持。

信号量是一种特殊的变量, 可用于线程同步。它只取自然数值, 并且只支持两种操作: P(SV): 如果信号量 SV 大于 0, 将它减一; 如果 SV 值为 0, 则挂起该线程。V(SV): 如果有其他进程因为等待 SV 而挂起, 则唤醒, 然后将 SV+1; 否则直接将 SV+1。其系统调用为:

- `sem_wait(sem_t *sem)`: 以原子操作的方式将信号量减 1, 如果信号量值为 0, 则 `sem_wait` 将被阻塞, 直到这个信号量具有非 0 值。
- `sem_post(sem_t *sem)`: 以原子操作将信号量值 +1。当信号量大于 0 时, 其他正在调用 `sem_wait` 等待信号量的线程将被唤醒。

互斥量: 互斥量又称互斥锁, 主要用于线程互斥, 不能保证按序访问, 可以和条件锁一起实现同步。当进入临界区时, 需要获得互斥锁并且加锁; 当离开临界区时, 需要对互斥锁解锁, 以唤醒其他等待该互斥锁的线程。其主要的系统调用如下:

- `pthread_mutex_init`: 初始化互斥锁
- `pthread_mutex_destroy`: 销毁互斥锁
- `pthread_mutex_lock`: 以原子操作的方式给一个互斥锁加锁, 如果目标互斥锁已经被上锁, `pthread_mutex_lock` 调用将阻塞, 直到该互斥锁的占有者将其解锁。
- `pthread_mutex_unlock`: 以一个原子操作的方式给一个互斥锁解锁。

条件变量: 条件变量, 又称条件锁, 用于在线程之间同步共享数据的值。条件变量提供一种线程间通信机制: 当某个共享数据达到某个值时, 唤醒等待这个共享数据的一个/多个线程。即, 当某个共享变量等于某个值时, 调用 `signal/broadcast`。此时操作共享变量时需要加锁。其主要的系统调用如下:

- `pthread_cond_init`: 初始化条件变量
  - `pthread_cond_destroy`: 销毁条件变量
  - `pthread_cond_signal`: 唤醒一个等待目标条件变量的线程。哪个线程被唤醒取决于调度策略和优先级。
  - `pthread_cond_wait`: 等待目标条件变量。需要一个加锁的互斥锁确保操作的原子性。该函数中在进入 `wait` 状态前首先进行解锁, 然后接收到信号后会再加锁, 保证该线程对共享资源正确访问。
1. \*\* 在 Linux 环境下, 请用信号量实现哲学家就餐的多线程应用程序。
  2. \*\* 在 Linux 环境下, 请用互斥锁和条件变量实现哲学家就餐的多线程应用程序。

3. \*\* 在 Linux 环境下, 请建立一个多线程的模拟资源分配管理库, 可通过银行家算法来避免死锁。
4. \*\* 扩展内核功能, 实现读者优先的读写信号量。
5. \*\* 扩展内核功能, 实现写者优先的读写信号量。
6. \*\*\* 扩展内核功能, 在内核中支持内核线程。
7. \*\*\* 进一步扩展内核功能, 在内核线程中支持同步互斥机制, 实现内核线程用的 mutex, semaphore, cond-var。
8. \*\*\* 扩展内核功能, 实现多核支持下的同步互斥机制。
9. \*\*\* 解决优先级反转问题: 实现 RM 实时调度算法, 设计优先级反转的实例, 实现优先级天花板和优先级继承方法。

## 问答题

1. \* 什么是并行? 什么是并发?
  - “并行”指的是同时进行多个任务。在多 CPU 环境中, 计算机具有多个独立的 CPU, 可以同时执行多个任务。例如, 如果你有两个 CPU, 那么它们可以同时运行两个不同的程序, 这样它们就是并行的。
  - “并发”指的是多个任务的同时发生, 但它们不一定是同时执行的。在单 CPU 环境中, 并发和并行是通过 CPU 快速地在多个任务之间切换来模拟同时发生的效果。例如, 如果你在同时运行多个程序, 那么 CPU 可以快速地在这些程序之间切换, 从而模拟它们同时发生的效果。这种情况下, 这些程序是并发的, 但不是并行的。
2. \* 为了创造临界区, 单核处理器上可以【关中断】, 多核处理器上需要使用【自旋锁】。请回答下列问题:
  - 多核上可不可以只用【关中断】?
    - 在多核处理器上仅使用关中断 (disable interrupt) 来实现临界区是不可行的, 因为关中断只能保证当前核上的代码不会被中断, 但不能保证其他核上的代码不会进入临界区, 对共享数据进行修改。
  - 单核上可不可以只用【自旋锁】?
    - 单核处理器上可以使用自旋锁来实现临界区, 但是并不是必须使用自旋锁, 如上述的关中断。
  - 多核上的【自旋锁】是否需要同时【关中断】?
    - 对于多核处理器上的自旋锁, 通常不需要关中断来创建临界区。相反, 自旋锁的实现会使用处理器提供的硬件特性来确保原子性, 例如原子操作、内存屏障等。这种方式能够避免全局中断, 从而提高系统的性能。
  - [进阶] 假如某个锁不会在中断处理函数中被访问, 是否还需要【关中断】?
    - 在单核处理器上, 如果所有的代码都是在同一个上下文中运行, 也就是没有中断或者线程切换的情况下, 如果在代码中使用锁来保护共享资源, 那么可以使用简单的互斥锁来实现临界区的保护, 而不需要关中断。
    - 在多核处理器上, 不同的核心可以独立运行不同的线程, 彼此之间不会互相干扰。在这种情况下, 可以使用自旋锁等更高效的同步机制来实现临界区的保护。如果代码中使用的锁需要在中断处理函数中被访问, 那么在多核处理器上需要关中断来保护临界区。在中断处理函数中, 由于上下文的切换, 可能会发生竞争条件, 因此需要通过关中断的方式来避免这种竞争。这样可以保证在中断处理函数执行期间, 不会有其他线程在访问共享资源, 从而保证临界区的安全性。
3. \*\* Linux 的多线程应用程序使用的锁 (例如 pthread\_mutex\_t) 不是自旋锁, 当上锁失败时会切换到其它进程执行。分析它和自旋锁的优劣, 并说明为什么它不用自旋锁?

**互斥锁和自旋锁的优劣：**互斥锁和自旋锁的本质区别在于加锁失败时，是否会释放 CPU。互斥锁在加锁失败时，会释放 CPU

- 当线程加锁失败时，内核会把线程的状态从「运行」状态设置为「睡眠」状态，然后把 CPU 切换给其他线程运行；
- 接着，当锁被释放时，之前「睡眠」状态的线程会变为「就绪」状态，然后内核会在合适的时间，把 CPU 切换给该线程运行。

**不使用自旋锁的原因是：**

- 可移植性：`pthread_mutex_t` 是 POSIX 标准中定义的一种互斥锁，不仅可以在 Linux 系统上使用，还可以在其他的 POSIX 兼容系统上使用，提高了应用程序的可移植性。
  - 性能：自旋锁在多核处理器上可以提高并发性能，但是在单核处理器上可能会降低性能，因为自旋锁需要不断地检查锁的状态，如果锁一直处于被占用的状态，就会一直占用处理器时间。而 `pthread_mutex_t` 是一种阻塞锁，在锁被占用时，会将线程挂起，让出处理器时间，从而避免了空转浪费处理器资源的情况。
  - 死锁：使用自旋锁需要非常小心，否则容易出现死锁的情况。例如，当一个线程持有一个自旋锁并等待另一个自旋锁时，如果另一个线程持有了这个自旋锁并等待第一个自旋锁，就会出现死锁。而 `pthread_mutex_t` 是一种阻塞锁，在锁的等待队列中维护了线程的等待关系，可以避免死锁的情况。
4. \*\*\* 程序在运行时具有两种性质：safety: something bad will never happen; liveness: something good will eventually occur. 分析并证明 Peterson 算法的 safety 和 liveness 性质。

下面是这两个性质的证明：

- **Safety 性质：**假设同时有两个线程 `$P_0$` 和 `$P_1$`，它们都试图进入其临界区，即执行关键代码段。如果两个线程同时进入关键代码段，就会发生竞态条件，可能导致不正确的结果。因此，我们希望确保只有一个线程能够进入其临界区。

Peterson 算法确保了只有一个线程可以进入其临界区。这是因为，在进入临界区之前，线程必须首先尝试获取锁。如果另一个线程已经获得了锁，则当前线程将被阻塞，直到另一个线程释放锁。因此，只有一个线程可以进入其临界区，这证明了 Peterson 算法的 safety 性质。

- **Liveness 性质：**我们需要证明，如果一个线程尝试进入其临界区，则它最终将能够进入。假设线程 `$P_0$` 和 `$P_1$` 都试图进入其临界区。如果线程 `$P_0$` 先尝试进入其临界区，则线程 `$P_1$` 会被阻塞，直到线程 `$P_0$` 退出其临界区并释放锁。反之亦然。

假设线程 `$P_0$` 试图进入其临界区，但是线程 `$P_1$` 已经占用了锁并且正在执行其临界区。线程 `$P_0$` 将被阻塞，并等待线程 `$P_1$` 释放锁。线程 `$P_1$` 将在其临界区内执行，并最终退出其临界区并释放锁。此时，线程 `$P_0$` 将获得锁，并能够进入其临界区。同样，如果线程 `$P_1$` 试图进入其临界区，那么也将发生类似的过程。

因此，Peterson 算法保证了线程能够最终进入其临界区，这证明了 Peterson 算法的 liveness 性质。

1. \* 信号量结构中的整数分别为 `+n`、`0`、`-n` 的时候，各自代表什么状态或含义？

- `+n`: 还有  $n$  个可用资源
- `0`: 所有可用资源恰好耗尽
- `-n`: 有  $n$  个进程申请了资源但无资源可用，被阻塞。

6. \*\* 考虑如下信号量实现代码：

```
class Semaphore {
 int sem;
 WaitQueue q;
}
Semaphore::P() {
```

(下页继续)

(续上页)

```

sem --;
if(sem < 0) {
 Add this thread to q.
 block.
}
}
Semaphore::V() {
 sem++;
 if(sem <= 0) {
 t = Remove a thread from q;
 wakeup(t);
 }
}

```

假如 P 操作或 V 操作不是原子操作，会出现什么问题？举一个例子说明。上述代码能否运行在用户态？上面代码的原子性是如何保证的？

如果 P 操作或 V 操作不是原子操作，将无法实现资源的互斥访问。P 操作和 V 操作都是通过关中断来实现的（可以再确认一下这点）。上述代码不能运行在用户态，因为这将带给用户态程序使能/屏蔽中断这种特权，相当于相信应用并放权给它。这会面临和我们引入抢占式调度之前一样的问题：线程可以选择恶意永久关闭中断而独占所有 CPU 资源，这将会影响到整个系统的正常运行。因此，事实上至少在 RISC-V 这样含多个特权级的架构中，这甚至是完全做不到的。

#### 7. \*\* 条件变量的 Wait 操作为什么必须关联一个锁？

当调用条件变量的 wait 操作阻塞当前线程的时候，该操作是在管程过程中，因此此时当前线程持有锁。在持有锁的情况下不能陷入阻塞，因此在陷入阻塞状态之前当前线程必须先释放锁；当被阻塞的线程被其他线程使用 signal 操作唤醒之后，需要重新获取到锁才能继续执行，不然的话就无法保证管程过程的互斥访问。

因此，站在线程的视角，必须持有锁才能调用条件变量的 wait 操作阻塞自身。

#### 8. \*\* 下面是条件变量的 wait 操作实现伪代码：

```

Condvar::wait(lock) {
 Add this thread to q.
 lock.unlock();
 schedule();
 lock.lock();
}

```

如果改成下面这样：

```

Condvar::wait() {
 Add this thread to q.
 schedule();
}
lock.unlock();
condvar.wait();
lock.lock();

```

会出现什么问题？举一个例子说明。

这种情况就是第 7 题提到的条件变量的 wait 操作没有关联一个锁。会造成被阻塞的线程被其他线程使用 signal 操作唤醒之后，无法获取锁，从而无法保证管程过程的互斥访问，导致管程失效。

#### 9. \* 死锁的必要条件是什么？

死锁的四个必要条件：

- 互斥条件：一个资源每次只能被一个进程使用。

- 请求与保持条件: 一个进程因请求资源而阻塞时, 对已获得的资源保持不放。
- 不剥夺条件: 进程已获得的资源, 在未使用完之前, 不能被其他进程强行剥夺。
- 循环等待条件: 若干进程之间形成一种头尾相接的循环等待资源关系。

这四个条件是死锁的必要条件, 只要系统发生死锁, 这些条件必然成立, 而只要上述条件之一不满足, 就不会发生死锁。

#### 10. \* 什么是死锁预防, 举例并分析。

预防死锁只需要破坏死锁的四个必要条件之一即可, 例如:

- 破坏互斥条件
- 破坏不可剥夺条件: 当进程的新资源不可取得时, 释放自己已有的资源, 待以后需要时重新申请。
- 破坏请求并保持条件: 进程在运行前一次申请完它所需要的全部资源, 在它的资源为满足前, 不把它投入运行。一旦投入运行, 这些资源都归它所有, 不能被剥夺。
- 破坏循环等待条件: 给锁/访问的资源进行排序, 要求每个线程都按照排好的顺序依次申请锁和访问资源

#### 11. \*\* 描述银行家算法如何判断安全性。

- 设置两个向量: 工作向量 Work, 表示操作系统可提供给线程继续运行所需的各类资源数目, 它含有  $m$  个元素, 初始时,  $Work = Available$ ; 结束向量 Finish, 表示系统是否有足够的资源分配给线程, 使之运行完成。初始时  $Finish[0..n-1] = false$ , 表示所有线程都没结束; 当有足够资源分配给线程时, 设置  $Finish[i] = true$ 。
- 从线程集合中找到一个能满足下述条件的线程

```
Finish[i] == false;
Need[i,j] <= Work[j];
```

若找到, 执行步骤 3, 否则, 执行步骤 4。

- 当线程  $thr[i]$  获得资源后, 可顺利执行, 直至完成, 并释放出分配给它的资源, 故应执行:

```
Work[j] = Work[j] + Allocation[i,j];
Finish[i] = true;
```

跳转回步骤 2

- 如果  $Finish[0..n-1]$  都为  $true$ , 则表示系统处于安全状态; 否则表示系统处于不安全状态。

通过操作系统调度, 如银行家算法来避免死锁不是广泛使用的通用方案。因为从线程执行的一般情况上看, 银行家算法需要提前获知线程总的资源申请量, 以及未来的每一次请求, 而这些请求对于一般线程而言在运行前是不可知或随机的。另外, 即使在某些特殊情况下, 可以提前知道线程的资源申请量等信息, 多重循环的银行家算法开销也是很大的, 不适合于对性能要求很高的操作系统中。

## 9.9.2 实验练习

实验练习包括实践作业和问答作业两部分。

## 编程作业

### 银行家算法——分数更新

**注解：**本实验为用户态实验，请在 Linux 环境下完成。

**背景：**在智能体大赛平台 [Saiblo](#) 网站上每打完一场双人天梯比赛后需要用 ELO 算法更新双方比分。由于 [Saiblo](#) 的评测机并发性很高，且 ELO 算法中的分值变动与双方变动前的分数有关，因此更新比分前时必须先为两位选手加锁。

**作业：**请模拟一下上述分数更新过程，简便起见我们简化为有  $p$  位选手参赛（编号  $[0, p)$  或  $[1, p]$ ），初始分值为 1000 分，有  $m$  个评测机线程（生产者）给出随机的评测结果（两位不同选手的编号以及胜负结果，结果可能为平局），有  $n$  个 worker 线程（消费者）获取结果队列并更新数据库（全局变量等共享数据）记录的分数。 $m$  个评测机各自模拟  $k$  场对局结果后结束线程，全部对局比分更新完成后主线程打印每位选手最终成绩以及所有选手分数之和。

上述参数  $p$ 、 $m$ 、 $n$ 、 $k$  均为可配置参数（命令行传参或程序启动时从 `stdin` 输入）。

简便起见不使用 ELO 算法，简化更新规则为：若不为平局，当胜者分数  $\geq$  败者分数时胜者 +20，败者 -20，否则胜者 +30，败者 -30；若为平局，分高者 -10，分低者 +10（若本就同分保持则不变）。

**消费者核心部分可参考如下伪码：** 获取选手 A 的锁获取选手 B 的锁更新 A、B 分数睡眠 1ms（模拟数据库更新延时）释放选手 B 的锁释放选手 A 的锁

**tips:**

- 由于 ELO 以及本题中给出的简化更新算法均为零和算法，因此出现冲突后可以从所有选手分数之和明显看出来，正确处理时它应该永远为 1000p
- 将一个 worker 线程看作哲学家，将 worker 正在处理的一场对局的两位选手看作两根筷子，则得到了经典的哲学家就餐问题

## 实现 eventfd

在 Linux 中有一种用于事件通知的文件描述符，称为 `eventfd`。其核心是一个 64 位无符号整数的计数器，在非信号量模式下，若计数器值不为零，则 `read` 函数会从中读出计数值并将其清零，否则读取失败；`write` 函数将缓冲区中的数值加入到计数器中。在信号量模式下，若计数器值非零，则 `read` 操作将计数值减一，并返回 1；`write` 将计数值加一。我们将实现一个新的系统调用：`sys_eventfd2`。

**eventfd:**

- syscall ID: 290
- 功能：创建一个 `eventfd`，`eventfd` 标准接口。
- C 接口：`int eventfd(unsigned int initval, int flags)`
- Rust 接口：`fn eventfd(initval: u32, flags: i32) -> i32`
- 参数：
  - `initval`: 计数器的初值。
  - **flags**: 可以设置为 0 或以下两个 flag 的任意组合（按位或）：
    - \* `EFD_SEMAPHORE` (1)：设置该 flag 时，将以信号量模式创建 `eventfd`。
    - \* `EFD_NONBLOCK` (2048)：若设置该 flag，对 `eventfd` 读写失败时会返回 -2，否则将阻塞等待直至读或写操作可执行为止。

- 说明:

- 通过 `write` 写入 `eventfd` 时，缓冲区大小必须为 8 字节。
- 进程 `fork` 时，子进程会继承父进程创建的 `eventfd`，且指向同一个计数器。

- 返回值：如果出现了错误则返回 -1，否则返回创建成功的 `eventfd` 编号。

- 可能的错误

- `flag` 不合法。
- 创建的文件描述符数量超过进程限制

---

**注解：**还有一个 `sys_eventfd` 系统调用（调用号 284），与 `sys_eventfd2` 的区别在于前者不支持传入 `flags`。

Linux 中的原生异步 IO 接口 `libaio` 就使用了 `eventfd` 作为内核完成 IO 操作之后通知应用程序的机制。

---

## 实验要求

- 完成分支: ch8-lab
- 实验目录要求不变。
- 通过所有测例。

## 问答作业

无

## 实验练习的提交报告要求

- 简单总结本次实验与上个实验相比你增加的东西。（控制在 5 行以内，不要贴代码）
- 完成问答问题
- (optional) 你对本次实验设计及难度的看法。



## 第九章：I/O 设备管理

---

### 10.1 引言

#### 10.1.1 本章导读

上一章的“达科塔盗龙”操作系统和“慈母龙”操作系统已经具备了传统操作系统中的内在重要因素，如进程、文件、地址空间、进程间通信、线程并发执行、支持线程安全访问共享资源的同步互斥机制等，应用程序也能通过操作系统输入输出字符，读写在磁盘上的数据。不过与我们常见的操作系统（如 Linux, Windows 等）比起来，好像感知与交互的 I/O 能力还比较弱。

终于到了 I/O 设备管理这一章了。人靠衣裳马靠鞍，如果操作系统不能把计算机的外设功能给发挥出来，那应用程序感知外在环境的能力和展示内在计算的能力都会大打折扣。比如基于中断和 DMA 机制的高性能 I/O 处理，图形化显示，键盘与鼠标输入等，这些操作系统新技能将在本章展现出来。所以本章要完成的操作系统的核心目标是：**让应用能便捷地访问外设。**

---

#### 注解：DMA 机制

DMA (Direct Memory Access) 是一种用于在计算机系统中传输 I/O 数据的技术。它允许 I/O 设备通过 DMA 控制器直接将设备中的 I/O 数据读入内存或把内存中的数据写入 I/O 设备，而整个数据传输过程无需处理器的介入。这意味着处理器可以在 DMA 传输期间执行其他任务，从而提高系统的性能和效率。I/O 设备通过 DMA 控制器访问的内存称为 DMA 内存。

---

## 以往操作系统对设备的访问

其实在第一章就非常简单介绍了 QEMU 模拟的 RISC-V 64 计算机中存在的外设：UART、时钟、virtio-net/block/console/gpu 等。并且 LibOS 模式的操作系统就已通过 RustSBI 间接地接触过串口设备了，即通过 RustSBI 提供的一个 SBI 调用 SBI\_CONSOLE\_PUTCHAR 来完成字符输出功能的。

在第三章，为了能实现抢占式调度，引入了时钟这个外设，结合硬件中断机制，并通过 SBI 调用 SBI\_SET\_TIMER 来帮助操作系统在固定时间间隔内获得控制权。而到了第五章，我们通过另外一个 SBI 调用 SBI\_CONSOLE\_GETCHAR 来获得输入的字符能力。这时的操作系统就拥有了与使用者进行简单字符交互的能力了。

后来在第六章又引入了另外一个外设 virtio-block 设备，即一个虚拟的磁盘设备。还通过这个存储设备完成了对数据的持久存储，并在其上实现了管理存储设备上持久性数据的文件系统。对 virtio-block 设备的 I/O 访问没有通过 RustSBI 来完成，而是直接调用了 `virtio_drivers` crate 中的 `virtio-blk` 设备驱动程序来实现。但我们并没有深入分析这个设备驱动程序的具体实现。

可以说在操作系统中，I/O 设备管理无处不在，且与 I/O 设备相关的操作系统代码—设备驱动程序在整个操作系统中的代码量比例是最高的（Linux/Windows 等都达到了 75% 以上），也是出错概率最大的地方。虽然前面章节的操作系统已经涉及了很多 I/O 设备访问的相关处理，但我们并没有对 I/O 设备进行比较全面的分析和讲解。这主要是由于各种 I/O 设备差异性比较大，操作系统很难像进程/地址空间/文件那样，对各种 I/O 设备建立一个一致通用的抽象和对应的解决方案。

但 I/O 设备非常重要，由于各种 I/O(输入输出)设备的存在才使得计算机的强大功能得以展现在大众面前，事实上对于各种 I/O 设备的高效管理是计算机系统操作能够在大众中普及的重要因素。比如对于手机而言，大众关注的不是 CPU 有多快，内存有多大，而是关注显示是否流畅，触摸是否敏捷这些外设带来的人机交互体验。而这些体验在很大程度上取决于操作系统对外设的管理与访问效率。

另外，对 I/O 设备的管理体现了操作系统最底层的设计机制，如中断，并发，异步，缓冲，同步互斥等。这对上层的进程，地址空间，文件等有着深刻的影响。所以在设计和实现了进程，地址空间，文件这些经典的操作系统抽象概念后，我们需要再重新思考一下，具备多种 I/O 设备管理能力的操作系统应该如何设计，特别是是否能给 I/O 设备也建立一个操作系统抽象。如果同学带着这些问题来思考和实践，将会对操作系统有更全面的体会。

---

### 注解：UNIX 诞生是从磁盘驱动程序开始的

回顾 UNIX 诞生的历史，你会发现一个有趣的故事：贝尔实验室的 Ken Thompson 在退出 Multics 操作系统开发后，还是想做继续操作系统方面的探索。他先是给一台闲置的 PDP-7 计算机的磁盘驱动器写了一个包含磁盘调度算法的磁盘驱动程序，希望提高磁盘 I/O 读写速度。为了测试磁盘访问性能，Ken Thompson 花了三周时间写了一个操作系统，这就是 Unix 的诞生。这说明是磁盘驱动程序促使了 UNIX 的诞生。

---

### 注解：设备驱动程序是操作系统的一部分？

我们都知道计算机是由 CPU、内存和 I/O 设备组成的。即使是图灵创造的图灵机这一理论模型，也有其必须存在的 I/O 设备：笔和纸。1946 年出现的远古计算机 ENIAC，都具有读卡器和打卡器来读入和输出穿孔卡片中的数据。当然，这些外设不需要额外编写软件，直接通过硬件电路就可以完成 I/O 操作了。但后续磁带和磁盘等外设的出现，使得需要通过软件来管理越来越复杂的外设功能了，这样设备驱动程序（Device Driver）就出现了，它甚至出现在操作系统之前，以子程序库的形式存在，以便于应用程序来访问硬件。

随着计算机外部设备越来越多，越来越复杂，设备驱动程序在操作系统中的代码比重也越来越大。甚至某些操作系统的名称直接加入了外设名，如微软在 1981 年至 1995 年间主导了个人计算机市场的 DOS 操作系统的全称是“Disk Operating System”。1973 年，施乐 PARC 开发了 Alto 个人电脑，它是第一台具有图形用户界面（GUI）的计算机，直接影响了苹果公司和微软公司设计的带图形界面的操作系统。微软后续开发的操作系统名称“Windows”也直接体现了图形显示设备（显卡）能够展示的抽象概念，显卡驱动和基于显卡驱动的图形界面上系统在 Windows 操作系统中始终处于非常重要的位置。

目前评价操作系统被产业界接受的程度有一个公认的量化指标，该操作系统的设备驱动程序支持的外设种类

和数量。量越大说明它在市场上的接受度就越高。正是由于操作系统能够访问和管理各种外设，才给了应用程序丰富多彩的功能。

本章的目标是深入理解 I/O 设备管理，并将站在 I/O 设备管理的角度来分析 I/O 设备的特征，操作系统与 I/O 设备的交互方式。接着会进一步通过串口，磁盘，图形显示等各种外设的具体实现来展现操作系统是如何管理 I/O 设备的，并展现设备驱动与操作系统内核其它重要部分的交互，通过扩展操作系统的 I/O 能力，形成具有灵活感知和捕猎能力的侏罗猎龙<sup>1</sup> 操作系统。

## 10.1.2 实践体验

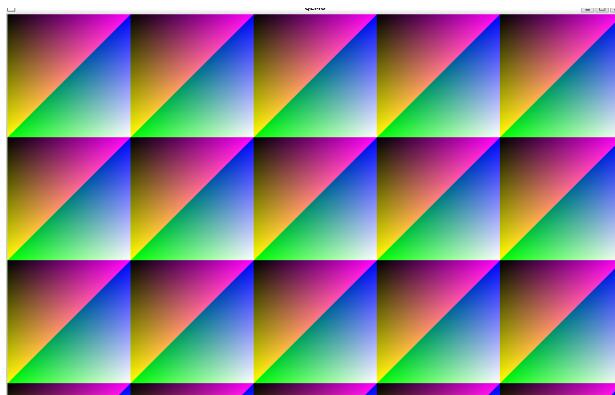
### 裸机设备驱动程序

获取代码：

```
$ git clone https://github.com/rcore-os/virtio-drivers.git
$ cd virtio-drivers
$ cd examples/riscv
```

在 qemu 模拟器上运行：

```
$ make qemu
...
[INFO] Detected virtio MMIO device with vendor id 0x554D4551, device type Block, ↵
↳ version Modern
[INFO] Detected virtio MMIO device with vendor id 0x554D4551, device type GPU, ↵
↳ version Modern
[INFO] Detected virtio MMIO device with vendor id 0x554D4551, device type Input, ↵
↳ version Modern
[INFO] Detected virtio MMIO device with vendor id 0x554D4551, device type Network, ↵
↳ version Modern
...
```



在这个测例中，可以看到对块设备（virtio-blk）、网络设备（virtio-net）、键盘鼠标类设备（virtio-input）、显示设备（virtio-gpu）的识别、初始化和初步的操作。

<sup>1</sup> 侏罗猎龙是一种小型恐龙，生活在1亿5千万年前的侏罗纪，它有独特的鳞片状的皮肤感觉器官，具有类似鳄鱼的触觉、冷热以及pH等综合感知能力，可能对狩猎有很大帮助。

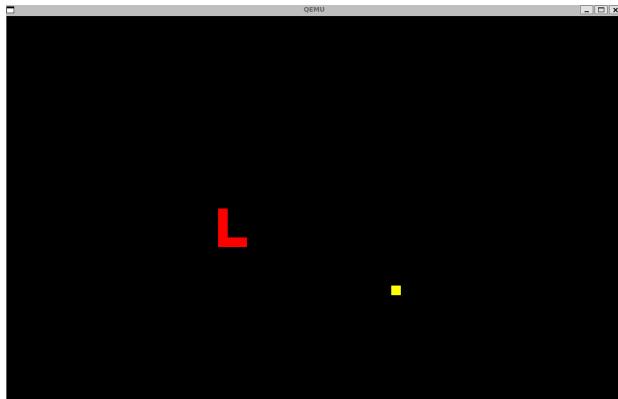
## 侏罗猎龙操作系统

```
$ git clone https://github.com/rcore-os/rCore-Tutorial-v3.git
$ cd rCore-Tutorial-v3
$ git checkout ch9
```

在 qemu 模拟器上运行:

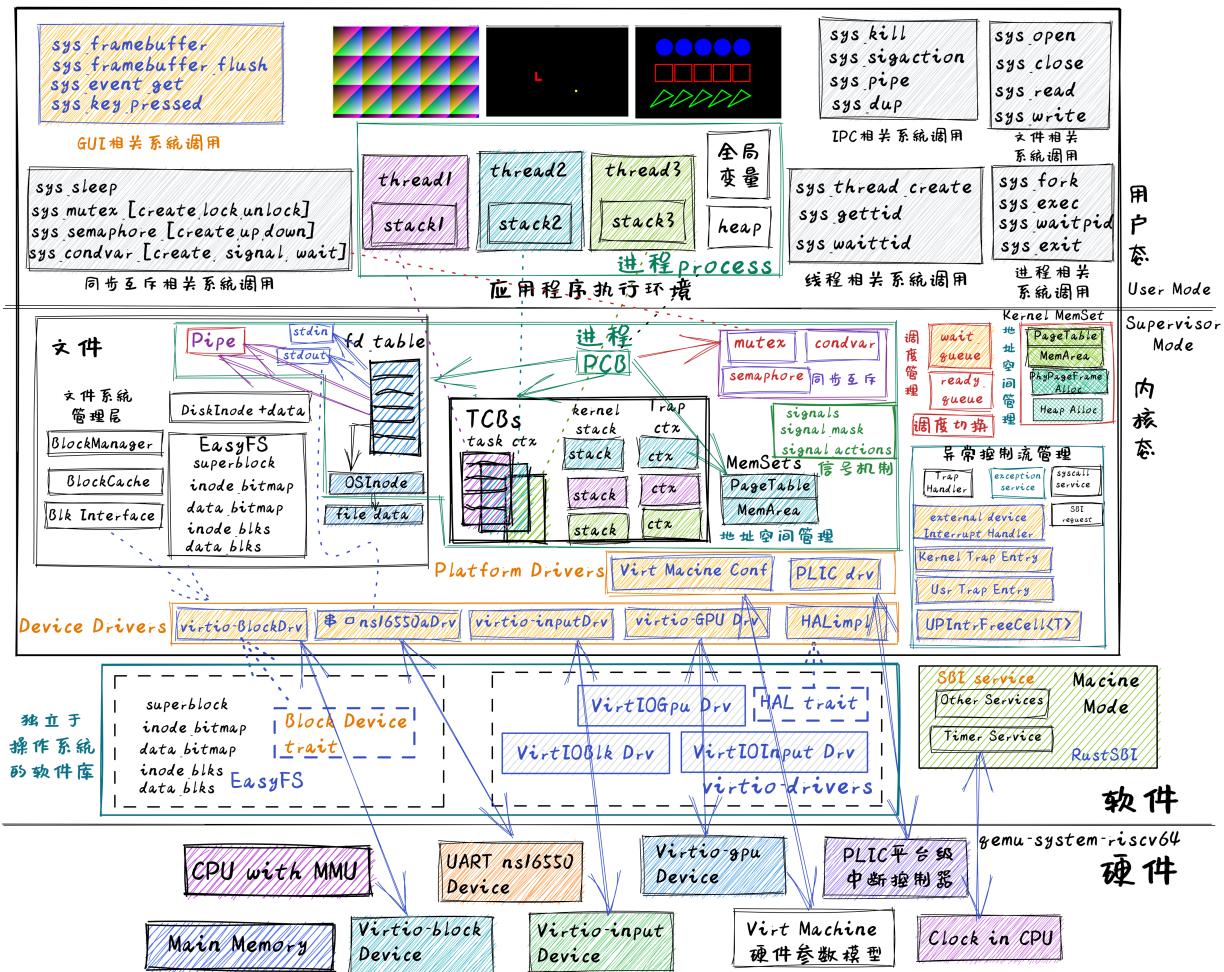
```
$ cd os
$ make run GUI=on
>> gui_snake #在OS启动后的shell界面中执行gui——snake游戏应用
```

在这个应用中, 可以看到 `gui_snake` 图形应用通过操作系统提供的 UART 串口驱动和 `virtio-gui` 显示驱动提供的服务来实现的一个贪吃蛇交互式小游戏。下面是该应用的演示图: 其中红色为贪吃蛇, 黄色方块为食物。玩家可以使用 `wasd` (分别表示上左下右) 控制贪吃蛇的行进方向。由于控制是基于和前面章节一样的命令行标准输入实现的, 在游玩的时候需要让焦点位于 `user shell` 命令行界面, 才能成功将控制传递给应用程序。应用画面可以在另一个图形显示窗口看到。



### 10.1.3 本章代码树

进一步增加了多种设备驱动程序的侏罗盗龙操作系统-DeviceOS 的总体结构如下图所示:



我们先分析一下图的上下两部分。从上图的左上角可以看到为应用程序增加了 GUI 相关的新系统调用。应用程序可以通过 sys\_FRAMEBUFFER 和 sys\_FRAMEBUFFER\_FLUSH 来显示图形界面, 通过 sys\_EVENT\_GET 和 sys\_KEY\_PRESSED 来接收来自串口/键盘/鼠标的输入事件。这其实就形成了基本的 GUI 应用支持框架。在上图的中上部, 添加了三个 GUI 应用的图形显示, 从左到右分别是: gui\_simple、gui\_snake 和 gui\_rect。

在上图的最下面展示的硬件组成中, 可以看到由 Qemu 模拟器仿真的 Virt Machine, 它包含了我们要管理的各种硬件组件, 包括在前面章节中重点涉及的 CPU 和 Main Memory, 还包括新引入的外设, ns16500 UART 串口外设、virtio-gpu 图形显示外设、virtio-input 键盘鼠标外设、virtio-blk 硬盘存储设备。为了与这些硬件交互, 系统软件还需了解有关这些外设的硬件参数模型, 如各个外设的控制寄存器的内存起始地址和范围等, 这就是 Qemu 模拟器中的 Virt Machine 硬件参数模型。硬件参数的具体内容可以在 Qemu 源码 qemu/include/hw/riscv/virt.h 和 qemu/hw/riscv/virt.c 中找到。

```

1 // qemu/hw/riscv/virt.c
2 static const MemMapEntry virt_memmap[] = {
3 [VIRT_PLIC] = { 0xc000000, VIRT_PLIC_SIZE(VIRT_CPUS_MAX * 2) },
4 [VIRT_UART0] = { 0x10000000, 0x1000 },
5 [VIRT_VIRTIO] = { 0x10001000, 0x1000 },
6 [VIRT_DRAM] = { 0x80000000, 0x0 },
7 ...
8 };
9 // qemu/include/hw/riscv/virt.h
10 enum {

```

(下页继续)

(续上页)

```

11 UART0_IRQ = 10,
12 VIRTIO_IRQ = 1, /* 1 to 8 */
13 ...
14 };

```

在上面的代码片段中，可以看到 UART 串口外设的控制寄存器的 MMIO 内存起始地址和空间大小为：{ 0x10000000, 0x100 }，而其它 virtio 外设的控制寄存器的 MMIO 内存起始地址和空间大小为 { 0x10001000, 0x1000 }。当操作系统知道这些外设的控制寄存器的 MMIO 内存地址后，就可以通过读写这些寄存器来访问和管理这些外设了。

同时，我们也看到了各种外设的中断号，如串口中断号 UART0\_IRQ 为 10，而“virtio”外设的中断号有 8 个，编号为 1~8。而对各种外设的中断的管理、检测发送给 CPU 等事务都在一个特殊的设备中完成，即 PLIC 平台级中断控制器 (Platform Level InterruptController)，它的控制寄存器内存起始地址和空间大小为 { 0xc000000, VIRT\_PLIC\_SIZE(VIRT\_CPUS\_MAX \* 2) }，它的空间大小与 CPU 个数相关。

现在看看上图中部的操作系统，蓝边橙底方块的部分是主要增加的内容，包括了外设驱动和与外设相关的中断处理。根据与各种外设的连线可以看到两类驱动：外设驱动和平台驱动。

- virtio-GPU Drv: 图形显示驱动
- ns16550a Drv: 串口驱动
- virtio-input Drv: 键盘鼠标驱动
- virtio-Block Drv: 块设备驱动
- PLIC drv: 平台级中断控制器驱动
- Virt Machine Conf: virt 计算机系统配置信息（可以理解为平台级配置驱动）

在与外设相关的中断处理方面，主要增加了对外设中断的处理，并被功能扩展的 异常控制流管理内核模块进行统一管理。异常控制流管理内核模块主要的扩展包括两方面，一方面是支持在内核态响应各种中断，这样就能在内核态中处理外设的中断事件。为此需要扩展在内核态下的中断上下文保存/恢复操作，并根据外设中断号来调用相应外设驱动中的外设中断处理函数。

另一方面是提供了 UPIntrFreeCell<T> 接口，代替了之前的 UPSafeCell<T>。在 Device OS 中把 UPSafeCell<T> 改为 UPIntrFreeCell<T>。这是因为在第九章前，系统设置在 S-Mode 中屏蔽中断，所以在 S-Mode 中运行的内核代码不会被各种外设中断打断，这样在单处理器的前提下，采用 UPSafeCell 来实现对可写数据的独占访问支持是够用的。但在第九章中，系统配置改为在 S-Mode 中使能中断，所以内核代码在内核执行过程中会被中断打断，无法实现可靠的独占访问。本章引入了新的 UPIntrFreeCell 机制，使得在通过 UPIntrFreeCell 对可写数据进行独占访问前，先屏蔽中断；而对可写数据独占访问结束后，再使能中断。从而确保线程对可写数据的独占访问时，不会被中断打断或引入可能的线程切换，而避免了竞态条件的产生。

在内核层，为了支持 Qemu 模拟的 Virt 计算机中不同外设，增加了 3 个外设级设备驱动程序，分别是 virtio-gpu 显示驱动、virtio-input 输入驱动和 ns16650 串口设备驱动，改进了 virtio-blk 块设备驱动，以支持高效的中断响应。而各种外设需要计算机中的支持。这 4 个外设级设备驱动程序需要计算机平台级的配置与管理，所以还增加了 Virt Machine Conf 和 PLIC 两个平台级设备驱动程序。在独立于操作系统的软件库中，增加了 virtio-drivers 库，实现了各种 virtio 外设的裸机设备驱动的主要功能。这样在实现操作系统中的设备驱动程序时，就可以直接封装 virtio 裸机设备驱动中的功能，简化了设备驱动程序的编写难度。

本章的代码主要包括两部分内容。一部分是 virtio-drivers 仓库中的驱动代码和裸机示例代码：

```

1 └── examples
2 └── riscv
3 └── src
4 ├── main.rs (各种virtio设备的测试用例)
5 └── virtio_impl.rs (用于I/O数据的物理内存空间管理的简单实现)

```

(下页继续)

(续上页)

```

6 └── src
7 ├── blk.rs (virtio-blk 驱动)
8 ├── gpu.rs (virtio-gpu 驱动)
9 ├── hal.rs (用于I/O数据的物理内存空间管理接口)
10 ├── header.rs (VirtIOHeader: MMIO设备寄存器接口)
11 ├── input.rs (virtio-input 驱动)
12 ├── net.rs (virtio-net 驱动)
13 └── queue.rs (virtqueues: 批量I/O数据传输的机制)

```

另外一部分是侏罗猎龙操作系统—Device OS 代码：

```

1 └── ...
2 └── easy-fs
3 └── src
4 ├── ...
5 └── block_dev.rs (BlockDevice trait中增加handle_irq接口)
6 └── os
7 └── src
8 ├── ...
9 └── main.rs (扩展blk/gpu/input等外设初始化调用) w
10 └── config.rs (修改KERNEL_HEAP_SIZE和MEMORY_END, 扩展可用内存空间)
11 └── boards
12 └── qemu.rs (扩展blk/gpu/input等外设地址设定/初始化/中断处理等操作)
13 └── drivers
14 ├── block
15 | └── virtio_blk.rs (增加非阻塞读写块/中断响应等I/O操作)
16 ├── bus
17 | └── virtio.rs (增加virtio-drivers需要的Hal trait接口)
18 ├── chardev
19 | └── ns16550a.rs (增加s-mode下的串口驱动)
20 ├── gpu
21 | └── mod.rs (增加基于virtio-gpu基本驱动的OS驱动)
22 ├── input
23 | └── mod.rs (增加基于virtio-input基本驱动的OS驱动)
24 └── plic.rs (增加RISC-V的PLIC中断控制器驱动)
25 └── fs
26 └── stdio.rs (改用s-mode下的串口驱动进行输入输出字符)
27 └── mm
28 └── memory_set.rs (扩展Linear内存映射方式, 用于显示内存存映射)
29 └── sync
30 ├── condvar.rs (扩展条件变量的wait方式, 用于外设驱动)
31 └── up.rs (扩展UPIIntrFreeCell<T> 支持内核态屏蔽中断的独占访问)
32 └── syscall
33 ├── gui.rs (增加图形显示相关的系统调用)
34 └── input.rs (增加键盘/鼠标/串口相关事件的系统调用)
35 └── trap
36 ├── mod.rs (扩展在用户态和内核态响应外设中断)
37 └── trap.S (扩展内核态响应中断的保存与恢复寄存器操作)
38
39 └── user
40 └── src
41 ├── bin
42 | ├── gui_rect.rs (显示不同大小正方形)
43 | ├── gui_simple.rs (彩色显示屏幕)
44 | ├── gui_snake.rs (用'a'/'s'/'d'/'w'控制的贪吃蛇图形游戏)
45 | ├── gui_uart.rs (用串口输入字符来控制显示正方形)
46 | └── huge_write_mt.rs (写磁盘文件性能测试例子)

```

(下页继续)

(续上页)

```

46 | └── inputdev_event.rs (接收键盘鼠标输入事件)
47 | └── random_num.rs (产生随机数)
48 | └── ...
49 | └── file.rs (文件系统相关的调用)
50 | └── io.rs (图形显示与交互相关的系统调用)
51 | └── sync.rs (同步互斥相关的系统调用)
52 | └── syscall.rs (扩展图形显示与交互的系统调用号和系统调用接口)
53 | └── task.rs (进程线程相关的系统调用)

```

## 10.1.4 本章代码导读

### 设计设备驱动程序的总体思路

这里简要介绍一下在内核中添加设备驱动的大致开发过程。本章涉及的代码主要与设备驱动相关，需要了解硬件，还需要了解如何与操作系统内核的其他部分进行对接，包括其他内核模块可以给驱动提供的内核服务，如内存分配等，以及需要驱动提供的支持功能，如外设中断响应等。在 Rust 软件工程开发中，推荐代码重用的 Crate 设计。所以在实际开发中，可以先在没有操作系统的裸机环境下 (no-std) 实现具备基本功能的裸机设备驱动 Crate，再实现一个最小执行环境（类似我们在第一章完成的三叶虫操作系统 -- LibOS），并在此最小执行环境中测试裸机设备驱动 Crate 的基本功能能正常运行。然后再在操作系统内核中设计实现设备驱动程序。操作系统中的设备驱动程序可以通过一层封装来使用裸机设备驱动 Crate 的各种功能，并对接操作系统的而其他内核模块。这样，操作系统中的设备驱动的开发和测试相对会简化不少。

设计设备驱动程序前，需要了解应用程序或操作系统中的其他子系统需要设备驱动程序完成哪些功能，再根据所需提供的功能完成如下基本操作：

- 1. 设备扫描/发现
- 2. 设备初始化
- 3. 准备发给设备的命令
- 4. 通知设备
- 5. 接收设备通知
- 6. (可选) 卸载设备驱动时回收设备驱动资源

对于设计实现裸机设备驱动，首先需要大致了解对应设备的硬件规范。在本章中，主要有两类设备，一类是实际的物理设备--UART (QEMU 模拟了这种 NS16550a UART 芯片规范)；另外一类是虚拟设备（如各种 Virtio 设备）。

然后需要了解外设是如何与 CPU 连接的。首先是 CPU 访问外设的方式，在 RISC-V 环境中，把外设相关的控制寄存器映射为某特定的内存区域（即 MMIO 映射方式），然后 CPU 通过读写这些特殊区域来访问外设（即 PIO 访问方式）。外设可以通过 DMA 来读写主机内存中的数据，并可通过中断来通知 CPU。外设并不直接连接 CPU，这就需要了解 RISC-V 中的平台级中断控制器（Platform-Level Interrupt Controller，PLIC），它管理并收集各种外设中断信息，并传递给 CPU。

## 裸机设备驱动程序

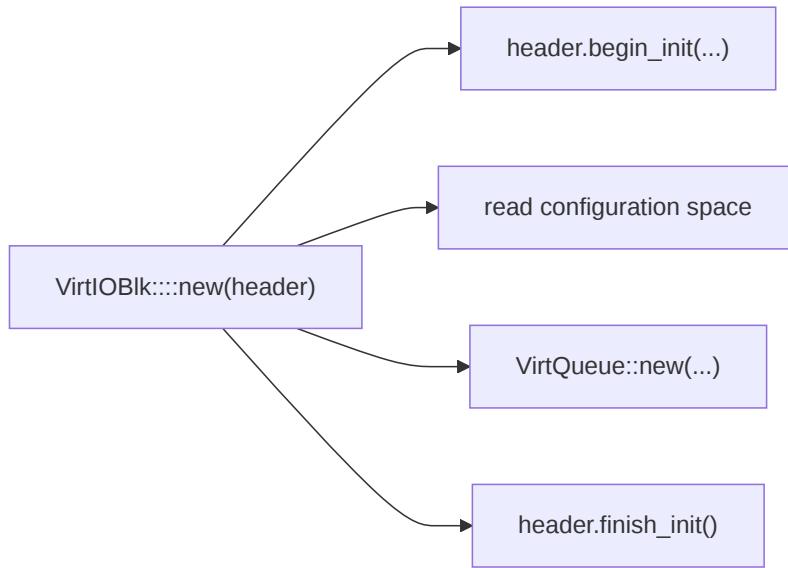
对于裸机设备驱动程序对外设的具体管理过程，大致会有发现外设、初始化外设和 I/O 读写与控制等操作。理解这些操作和对应的关键数据结构，就大致理解外设驱动要完成的功能包含哪些内容。每个设备驱动的关键数据结构和处理过程有共性部分和特定的部分。同学们可以从 `virtio-drivers` crate 中的 `examples/riscv/src/main.rs` 和 `src\blk.rs` 有关 `virtio` 设备的功能测试例子来分析。以 `virtio-blk` 存储设备为例，可以看到需要完成的工作包括：

1. 设备扫描/发现：首先是访问 OpenSBI (这里没有用 RustSBI，用的是 QEMU 内置的 SBI 实现) 提供的设备树信息，了解 QEMU 硬件中存在的各种外设，根据外设 ID 来找到 `virtio-blk` 存储设备。



2. 设备初始化：找到 virtio-blk 外设后，就进行外设的初始化，如果学习了 virtio 规范（需要关注的是 virtqueue、virtio-mmio device，virtio-blk device 的描述内容），那就可以看出代码实现的初始化过程和 virtio 规范中的 virtio 设备初始化步骤基本上是一致的，但也有与具体设备相关的特定初始化内容，比

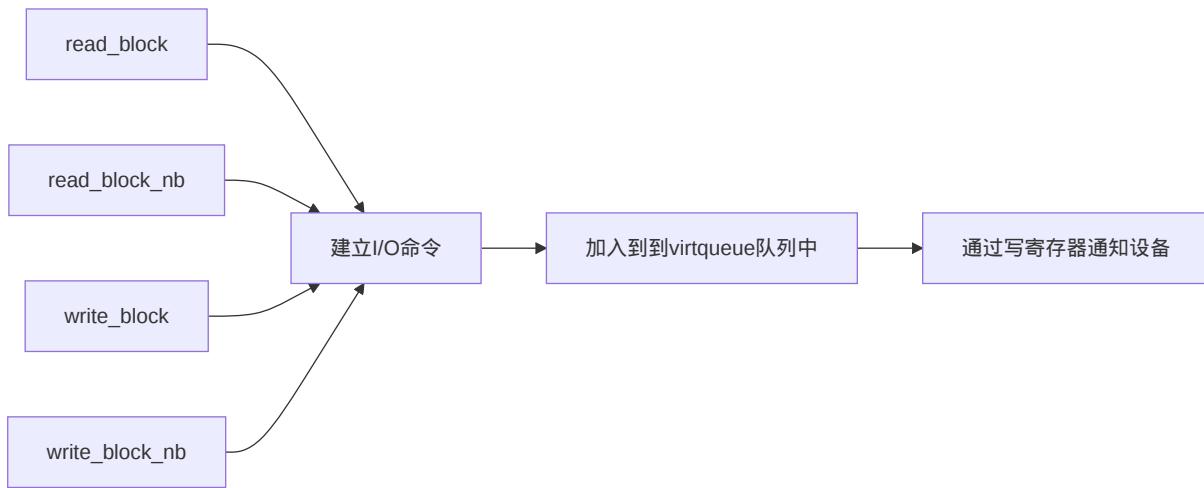
如分配 I/O buffer 等。



3. 准备发给设备的命令：初始化完毕后，设备驱动在收到上层内核发出的读写扇区/磁盘块的请求后，就

能通过 virtqueue 传输通道发出 virtio-blk 设备能接收的 I/O 命令和 I/O buffer 的区域信息。

4. 通知设备：驱动程序通过 *kick* 机制（即写 virtio 设备中特定的通知控制寄存器）来通知设备有新请求。



- 接收设备通知: virtio-blk 设备收到信息后, 会通过 DMA 操作完成磁盘数据的读写, 然后通过中断或其他方式让设备驱动知道命令完成或命令执行失败。由于中断处理例程的完整操作与操作系统内核相关性较大, 所以在裸机设备驱动中, 没有实现这部分的完整功能, 而只是提供了表示收到中断的操作。

## 操作系统设备驱动程序

由于有了裸机设备驱动程序的实现, 对于操作系统设备程序, 可以直接访问裸机设备驱动程序的各种功能。这样操作系统设备驱动程序的复杂性和代码量大大降低, 整个代码量不到 100 行。不过还需解决如下一些关键问题:

- 发现具体计算机 (如 virt machine ) 中的设备 (即与设备交互的设备控制寄存器的 MMIO 内存地址);
- 与其它操作系统内核模块 (如文件系统、同步互斥、进程管理等) 的对接;
- 封装裸机设备驱动程序, 提供操作系统层面的 I/O 设备访问能力 (初始化、读写、控制等操作)。

另外, 操作系统还需满足裸机设备驱动程序对操作系统的需求, 并能对各种外设进行统一的管理, 这主要集中在硬件平台级的支持。主要的服务能力包括:

- 在硬件平台层面发现具体计算机 (如 virt machine ) 中的各种外设的能力;
- 在硬件平台层面的外设中断总控能力, 即在外设中断产生后, 能分析出具体是哪个外设产生的中断, 并进行相应的处理;
- 给裸机设备驱动程序提供操作系统级别的服务能力, 以 virtio-drivers 为例, OS 需要提供 HAL Trait 的具体实现, 即驱动进行 I/O 操作所需的内存动态分配。

以面向 virtio-blk 外设的操作系统驱动为例, 我们可以看看上述过程的具体实现。在硬件平台的总体支持方面, 为简化操作, 通过对 Qemu 的分析, 在操作系统中直接给出 virt machine 中各个外设的控制寄存器地址 (代码位置: os/src/boards/qemu.rs)。为了完成外设中断总控, 操作系统提供了 PLIC 驱动, 支持对 virt machine 中各种外设的中断响应 (代码位置: os/src/drivers/plic.rs)。

在具体设备驱动实现上, 首先是发现设备, 操作系统建立了表示 virtio-blk 设备驱动的全局变量 BLOCK\_DEVICE (代码位置: os/src/drivers/block/mod.rs)。为简化发现设备的过程, 操作系统直接指定了 virtio-blk 设备在 virt machine 中的设备控制寄存器地址 VIRTIO0。

然后是驱动程序初始化、读写块和中断处理的实现 (代码位置: os/src/drivers/block/virtio\_blk.rs)。在操作系统的第一次访问 BLOCK\_DEVICE 时, 会执行 VirtIOBlock::new() 方法, 通过调用 virtio-blk 裸机设备驱动库中的功能, 完成了块设备驱动的初始化工作, 并初始化条件变量, 用于后续块读写过程中与进程的交互 (即让等待 I/O 访问结果的进程先挂起)。

块设备驱动的服务对象是文件系统, 它们之间需要有一个交互的接口, 这就是在 easy-fs 文件系统模块定义的 BlockDevice trait:

```

1 pub trait BlockDevice: Send + Sync + Any {
2 fn read_block(&self, block_id: usize, buf: &mut [u8]);
3 fn write_block(&self, block_id: usize, buf: &[u8]);
4 fn handle_irq(&self);
5 }
```

操作系统块设备驱动程序通过调用裸机块设备驱动程序库, 可以很简洁地实现上述功能。在具体实现上, 在调用了裸机块设备驱动程序库的读写块方法后, 通过条件变量让等待 I/O 访问结果的进程先挂起)。在中断处理的方法中, 在得到 I/O 读写块完成的中断信息后, 通过条件变量唤醒等待的挂起进程。

至此, 就分析完毕操作系统设备驱动程序的所有功能了。接下来, 我们就可以深入分析到 I/O 设备管理的级别概念、抽象描述和侏罗猎龙操作系统的具体实现。

## 10.2 I/O 设备

### 10.2.1 本节导读

本节简要介绍了 I/O 设备的发展，可以看到设备越来越复杂，越来越多样化；然后进一步介绍了主要的 I/O 传输方式，这在后续的驱动程序中都会用上；最后介绍了历史上出现的几种 I/O 设备抽象，从而可以理解设备驱动程序的编程基础所在。

### 10.2.2 I/O 设备概述

I/O 设备从早期相对比较简单的串口、键盘和磁盘等，逐步发展壮大，已经形成种类繁多，不同领域的各种类型的设备大家庭。而各种设备之间功能不一，性能差异巨大，难以统一地进行管理，这使得对应 I/O 设备的设备驱动程序成为了操作系统中最繁杂的部分。

站在不同的角度会对 I/O 设备有不同的理解。在硬件工程师看来，I/O 设备就是一堆芯片、电源和其他电路的组合；而软件程序员则主要关注 I/O 设备为软件提供的接口（interface），即硬件能够接收的命令、能够完成的功能以及能产生的各种响应或错误等。操作系统重点关注的是如何对 I/O 设备进行管理，而不是其内部的硬件工作原理。然而，对许多 I/O 设备进行编程还是不可避免地涉及到其内部的硬件细节。如果对 I/O 设备的发展过程进行深入分析，是可以找到 I/O 设备的共性特点，从而可以更好地通过操作系统来管理 I/O 设备。

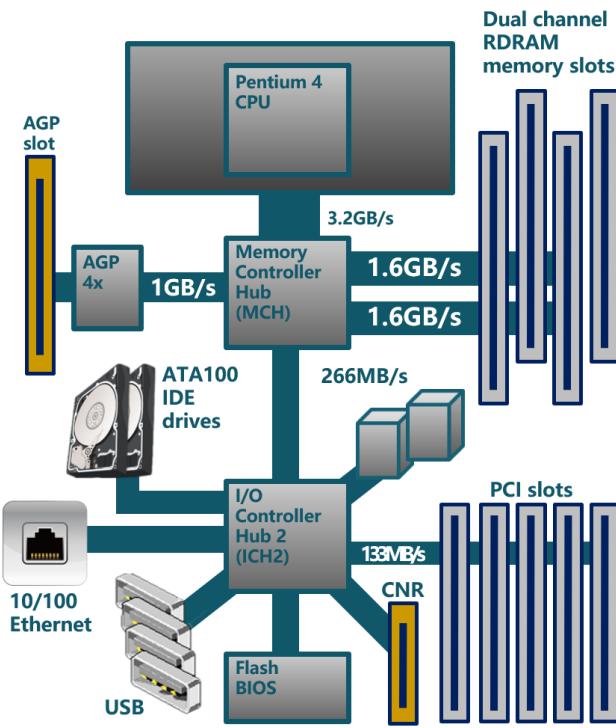
#### I/O 设备的发展

计算机的发展历史可体现为计算机硬件中各个部件的复杂度和集成度的变化发展过程。而在 I/O 设备变化过程，除了外设硬件的多样性越来越广和集成度越来越高以外，与 CPU 进行交互的能力也越来越强。在计算机发展过程中，I/O 设备先后出现了很多，也消亡了不少。

现在 I/O 设备的种类繁多，我们可以从数据传输的特点来给 I/O 设备进行分类。早期的 UNIX 把 I/O 设备分为两类：块设备（block device）和字符设备（character device）。块设备（比如磁盘）把信息存储在固定大小的块中，每个块有独立的地址。块的大小一般在 0.5KB 至 32KB 不等。块设备的 I/O 传输以一个或多个完整的（连续的）块为单位。另一类 I/O 设备是字符设备，字符设备（如串口，键盘等）以单个字符为单位发送或接收一个字符流。字符设备不需要寻址，只需访问 I/O 设备提供的相关接口即可获得/发出字符信息流。

后来随着网络的普及，又出现了一类设备：网络设备。网络面向报文而不是面向字符流或数据块，还具有数据丢失等可靠性问题，因此将网络设备映射为常见的文件比较困难。为此 UNIX 的早期继承者 BSD (Berkeley Software Distribution) 提出了 socket 接口和对应操作，形成了事实上的 TCP/IP 网络 API 标准。

再后来随着个人电脑的普及，计算机已经进入寻常百姓家中，计算机的功能和控制范围进一步放大，各种类型的 I/O 设备层出不穷。GPU、声卡、显卡等外设已经很难归类到上述的三种分类中，所以它们也就形成了各自独立的设备类型。各种设备出现时间有早晚，功能特点各异，这也使得现有的操作系统在设备驱动的设计和实现上面比较繁杂。当前典型的桌面计算机的 I/O 总体架构如下图所示：



## 北桥

- ▶ 内存
- ▶ AGP/PCI-Express
- ▶ Built-in display

## 南桥

- ▶ ATA/IDE
- ▶ PCI总线
- ▶ USB/Firewire总线
- ▶ Serial/Parallel接口
- ▶ DMA 控制器
- ▶ Interrupt控制器
- ▶ RTC, ACPI, BIOS, ...

从 CPU 与外设的交互方式的发展过程来看，CPU 可管理的设备数量越来越多，CPU 与设备之间的数据传递性能（延迟和吞吐量）也越来越强。总体上看，CPU 连接的外设有如下的发展过程：

### 简单设备

计算机发展早期，CPU 连接的设备不多，设备性能较低，所以 CPU 可通过 I/O 接口（如嵌入式系统中的通用输入输出 GPIO 接口）直接控制 I/O 设备（如简单的发光二极管等），这在简单的单片机和微处理器控制设备中经常见到。其特点是 CPU 发出 I/O 命令或数据，可立刻驱动 I/O 设备并产生相应的效果。

#### 注解：GPIO

GPIO (General-Purpose Input/Output) 是一种输入/输出接口，可以在微控制器或嵌入式系统上使用。它由一组可编程的引脚组成，可以作为输入或输出使用。GPIO 引脚可以控制或感测电平，并且可以用于连接各种类型的传感器和输出设备，如 LED、按钮、马达和各种类型的传感器。

GPIO 引脚可以配置为输入或输出，并且可以通过软件控制其电平。在输入模式下，GPIO 引脚可以感测外部电平变化，如按钮按下或传感器发出的信号。在输出模式下，GPIO 引脚可以控制外部设备的状态，如 LED 亮或灭或电机转动。GPIO 可用于实现软件定义的功能，这意味着可以在设备驱动程序中来控制 GPIO 引脚的状态。

## 基于总线的多设备

随着计算机技术的发展, CPU 连接的设备越来越多, 需要在 CPU 与 I/O 设备之间增加了一层 I/O 控制器 (如串口控制器等)。CPU 可通过对 I/O 控制器进行编程来控制各种设备。其特点是 CPU 给 I/O 控制器发出 I/O 命令或读写数据, 由 I/O 控制器来直接控制 I/O 设备和传达 I/O 设备的信息给 CPU。CPU 还需通过访问 I/O 控制器相关寄存器获取 I/O 设备的当前状态。其特点是 CPU 需要轮询检查设备情况, 对于低速设备 (如串口等) 而言, 高速 CPU 和低速设备之间是一种串行执行的过程, 导致 CPU 利用率低。随着设备的增多, I/O 控制器也逐渐通用化 (如各种总线接口等), 把不同设备连接在一起, 并能把设备间共性的部分进行集中管理。

同时, 为了简化 CPU 与各种设备的连接, 出现了 **总线 (bus)**。总线定义了连接在一起的设备需要共同遵守连接方式和 I/O 时序等, 不同总线 (如 SPI 总线、I2C 总线、USB 总线、PCI 总线等) 的连接方式和 I/O 时序是不同的。

---

### 注解: 总线

计算机中的总线是一种用于在计算机中连接不同设备的电气传输路径。它可用于在计算机的主板和外部设备之间传输数据。SPI (Serial Peripheral Interface)、I2C (Inter-Integrated Circuit)、USB (Universal Serial Bus) 和 PCI (Peripheral Component Interconnect) 总线都是用于在电脑和外部设备之间传输数据技术, 但它们之间有一些显著的区别。在操作系统的眼里, 总线也是一种设备, 需要设备驱动程序对其进行管理控制。

连接方式:

- SPI 总线使用四根线来连接设备。它通常使用四条线路, 一个用于传输数据 (MOSI), 一个用于接收数据 (MISO), 一个用于时钟 (SCK), 另一个用于选择设备 (SS)。
- I2C 总线使用两根线来连接设备。它通常使用两个线路, 一个用于传输数据 (SDA), 另一个用于时钟 (SCL)。
- USB 总线是一种通用外部总线, 具有即插即用和热插拔的功能, 处理器通过 USB 控制器与连接在 USB 上的设备交互。
- PCI 总线使用一条板载总线来连接计算机中处理器 (CPU) 和周边设备。这个总线通常是一条板载在主板上的总线, 它使用一组插座和插头来连接设备。

在速度上, I2C 总线的传输速度较慢, 通常在几 KB/s~ 几百 KB/s。SPI 总线的传输速度较快, 通常在几 MB/s。USB 总线设备的传输速度更快, 通常在几 MB/s~ 几十 GB/s。PCI 总线的传输速度最快, 可达到几 GB/s~ 几十 GB/s。

在应用领域方面, SPI 总线和 I2C 总线通常用于嵌入式系统中, 连接传感器、显示器和存储设备等外设。USB 总线和 PCI 总线通常用于桌面和服务器计算机中, 连接打印机、键盘、鼠标、硬盘、网卡等外设。

---

## 支持中断的设备

随着处理器技术的高速发展, CPU 与外设的性能差距在加大, 为了不让 CPU 把时间浪费在等待外设上, 即为了解决 CPU 利用率低的问题, I/O 控制器扩展了中断机制 (如 Intel 推出的 8259 可编程中断控制器)。CPU 发出 I/O 命令后, 无需轮询忙等, 可以干其他事情。但外设完成 I/O 操作后, 会通过 I/O 控制器产生外部中断, 让 CPU 来响应这个外部中断。由于 CPU 无需一直等待外设执行 I/O 操作, 这样就能让 CPU 和外设并行执行, 提高整个系统的执行效率。

---

### 注解: 中断并非总是比 PIO 好

尽管中断可以做到计算与 I/O 的重叠, 但这仅在快速 CPU 和慢速设备之间的数据交换速率差异大的情况下上有意义。否则, 如果设备的处理速度也很快 (比如高速网卡的速率可以达到 1000Gbps), 那么额外的中断处理和中断上下文切换、进程上下文切换等的代价反而会超过其提高 CPU 利用率的收益。如果一个或多个

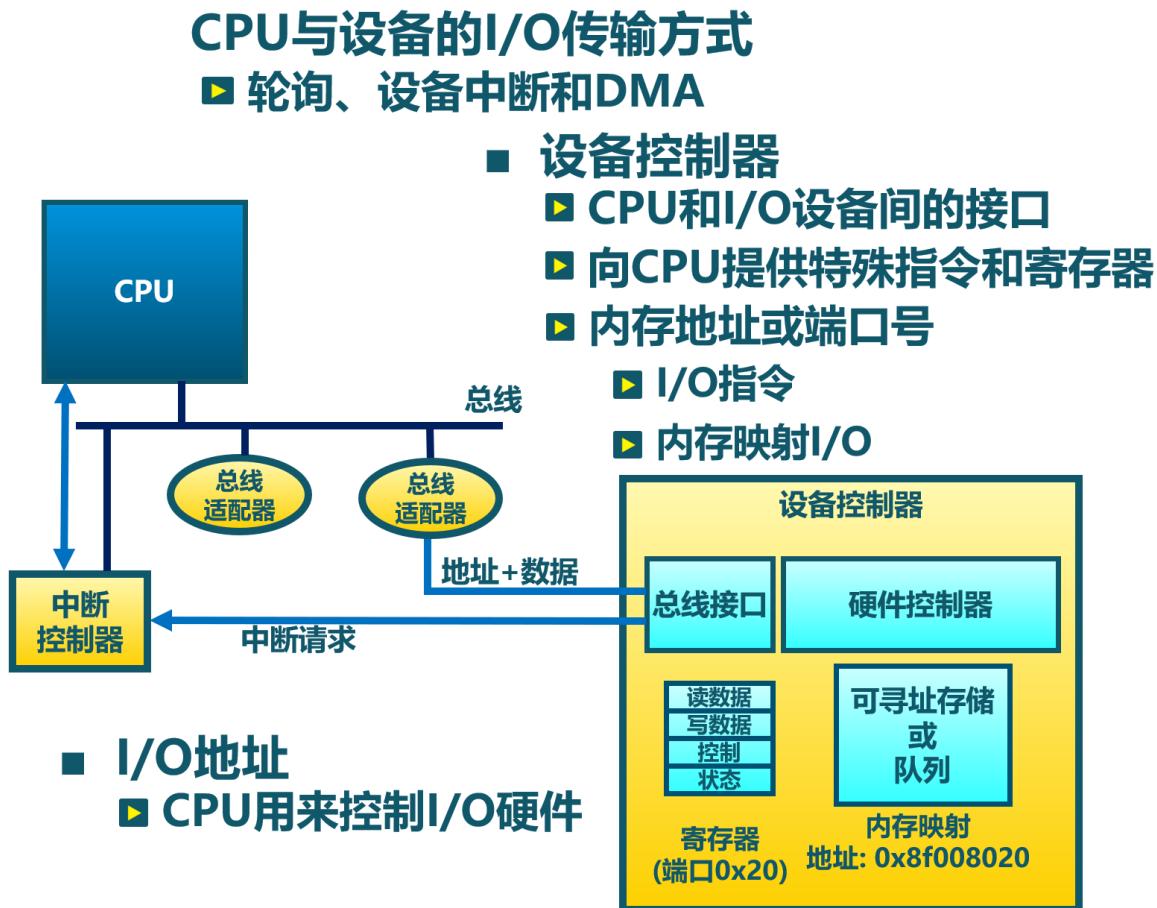
设备在短时间内产生大量的外设中断，可能会使得系统过载，并且让操作系统应付不过来，从而引发活锁<sup>1</sup>。在上述比较特殊的情况下，采用轮询的方式反而更有效，可以在操作系统自身的调度上提供更多的控制，甚至绕过操作系统直接让应用管理和控制外设。

## 高吞吐量设备

外设技术的发展也在加速，某些高性能外设（SSD，网卡等）的性能在逐步提高，如果每次中断产生的 I/O 数据传输量少，那么 I/O 设备（如硬盘/SSD 等）要在短期内传输大量数据就会频繁中断 CPU，导致中断处理的总体开销很大，系统效率会降低。通过 DMA（Direct Memory Access，直接内存访问）控制器（如 Intel 推出 8237DMA 控制器等），可以让外设在 CPU 没有访问内存的时间段中，以数据块的方式进行外设和内存之间的数据传输，且不需要 CPU 的干预。这样 I/O 设备的传输效率就大大提高了。CPU 只需在开始传送前发出 DMA 指令，并在外设结束 DMA 操作后响应其发出的中断信息即可。

### 10.2.3 I/O 传输方式

在上述的 I/O 设备发展过程可以看到，CPU 主要有三种方式可以与外设进行数据传输：Programmed I/O（简称 PIO）、Interrupt、Direct Memory Access（简称 DMA），如下图所示：



<sup>1</sup> Jeffrey Mogul and K. K. Ramakrishnan, Eliminating Receive Livelock in an Interrupt-driven Kernel, USENIX ATC 1996, San Diego, CA, January 1996

## Programmed I/O

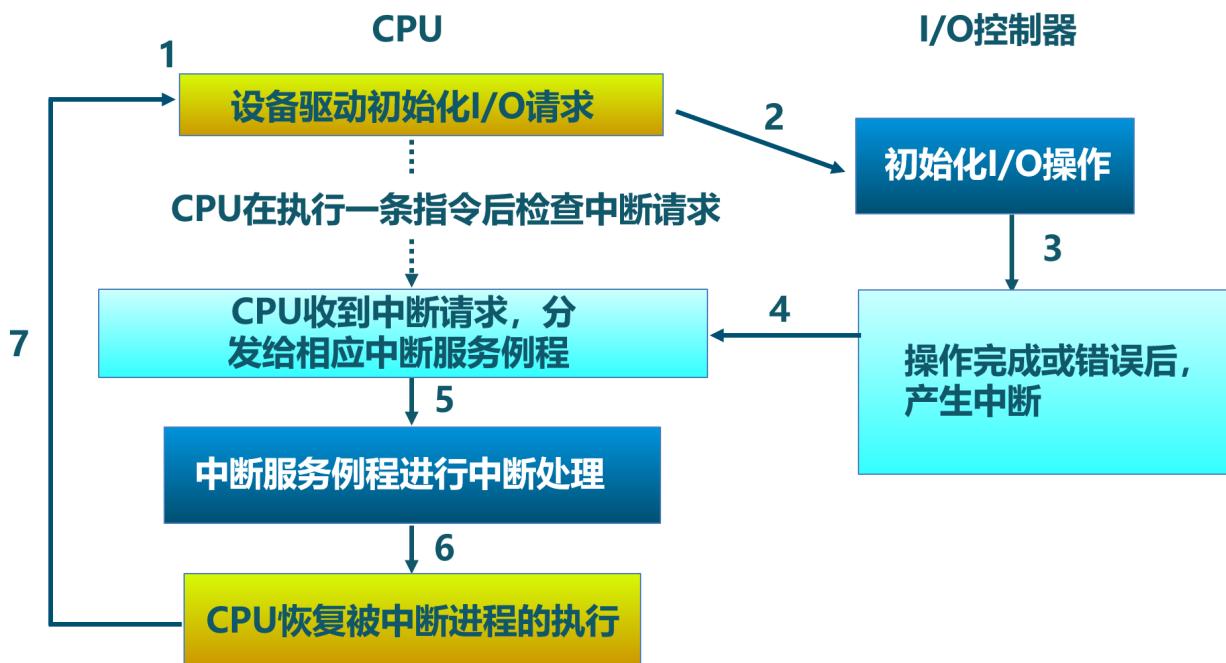
PIO 指 CPU 通过发出 I/O 指令的方式来进行数据传输。PIO 方式可以进一步细分为基于 Memory-mapped 的 PIO (简称 MMIO) 和 Port-mapped 的 PIO (简称 PMIO)，MMIO 是将 I/O 设备物理地址映射到内存地址空间，这样 CPU 就可以通过普通访存指令将数据送到 I/O 设备在主存上的位置，从而完成数据传输。

对于采用 PMIO 方式的 I/O 设备，它们具有自己独立的地址空间，与内存地址空间分离。CPU 若要访问 I/O 设备，则需要使用特殊的 I/O 指令，如 x86 处理器中的 IN 、 OUT 指令，这样 CPU 直接使用 I/O 指令，就可以通过 PMIO 方式访问设备。

## Interrupt based I/O

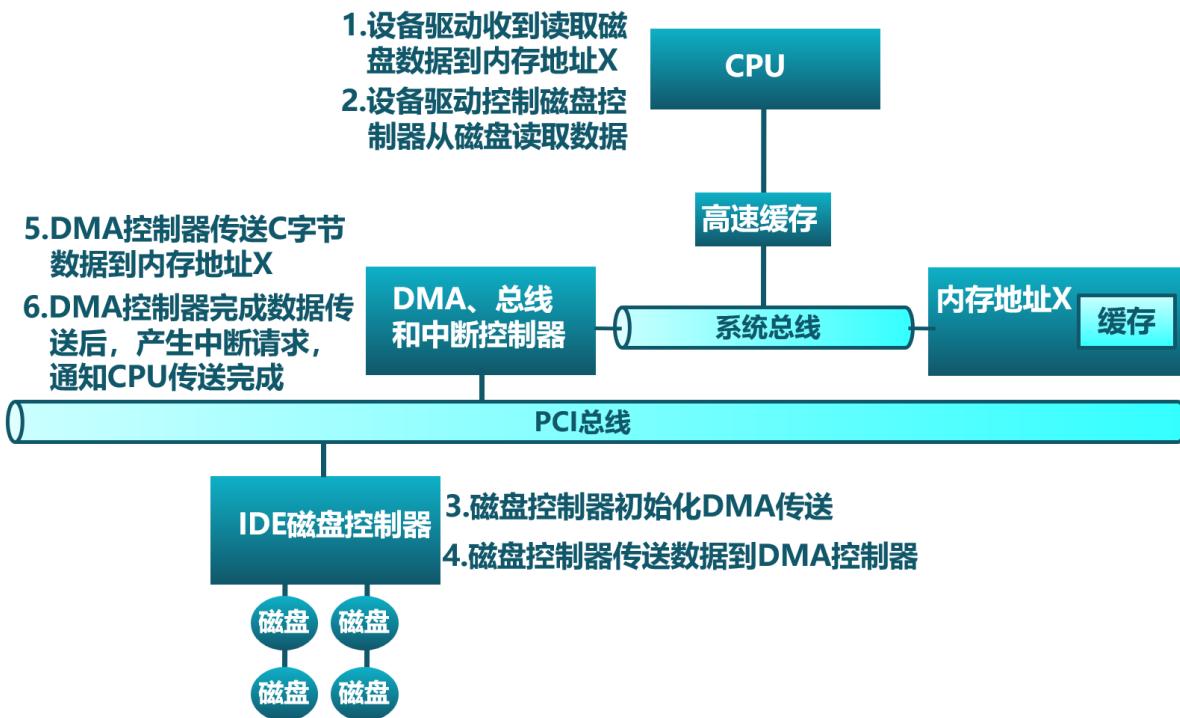
如果采用 PIO 方式让 CPU 来获取外设的执行结果，那么这样的 I/O 软件中有一个 CPU 读外设相关寄存器的循环，直到 CPU 收到可继续执行 I/O 操作的外设信息后，CPU 才能进一步做其它事情。当外设 (如串口) 的处理速度远低于 CPU 的时候，将使 CPU 处于忙等的低效状态中。

中断机制的出现，极大地缓解了 CPU 的负担。CPU 可通过 PIO 方式来通知外设，只要 I/O 设备有了 CPU 需要的数据，便会发出中断请求信号。CPU 发完通知后，就可以继续执行与 I/O 设备无关的其它事情。中断控制器会检查 I/O 设备是否准备好进行传输数据，并发出中断请求信号给 CPU。当 CPU 检测到中断信号，CPU 会打断当前执行，并处理 I/O 传输。下图显示了设备中断的 I/O 处理流程：



## Direct Memory Access

如果外设每传一个字节都要产生一次中断，那系统执行效率还是很低。DMA (Direct Memory Access) 是一种用于在计算机系统中进行快速数据传输的技术。它允许设备直接将数据传输到内存中，而不必通过 CPU 来直接处理。这样使得 CPU 从 I/O 任务中解脱出来，从而提高了系统的整体性能。DMA 操作通常由 DMA 控制器来完成。当 CPU 需要从内存中读取或写入设备数据时，它会提前向 DMA 控制器发出准备请求，然后 DMA 控制器会在后续阶段直接将数据传输到目标位置。下图显示了磁盘数据的 DMA 传输的图示例子：



在后面的小节中，我们会进一步介绍基于 I/O 控制器的轮询，中断等方式的设备驱动的设计与实现。

#### 注解: DMA 技术工作流程

当 CPU 想与外设交换一块数据时，它会向 DMA 控制器发出一条命令。命令的基本内容包括：读或写 I/O 设备的操作标记，I/O 设备的地址，DMA 内存的起始地址和传输长度。然后 CPU 继续其它工作。DMA 控制器收到命令后，会直接从内存中或向内存传送整块数据，这个传输过程不再需要通过 CPU 进行操作。传送结束后，DMA 控制器会通过 I/O 控制器给 CPU 发送一个表示 DMA 操作结束的中断。CPU 在收到中断后，知道这次 I/O 操作完成，可进行后续相关事务的处理。

在后续讲解的 virtio-blk, virtio-gpu 等模拟设备中，就是通过 DMA 来传输数据的。

### 10.2.4 I/O 设备抽象

#### I/O 接口的交互协议

对于一个外设而言，它包含了两部分重要组成部分。第一部分是对外向系统其他部分展现的设备 I/O 接口 (hardware I/O interface)，这样操作系统才能通过接口来管理控制外设。所有设备都有自己的特定接口以及典型交互的协议。第二部分是对内的内部结构，包含了设备相关物理实现。由于外在接口的多样性，使得操作系统难以统一管理外设。

如果我们不考虑具体的设备，而是站在一个高度抽象的角度来让软件管理设备，那么我们就不用太关注设备的内部结构，而重点考虑设备的接口如何进行简化。其实一个简化的抽象设备接口需要包括三部分：状态、命令、数据。软件可以读取并查看设备的当前状态，从而根据设备当前状态决定下一步的 I/O 访问请求；而软件是通过一系列的命令来要求设备完成某个具体的 I/O 访问请求；在完成一个 I/O 访问请求中，会涉及到将数据传给设备或从设备接收数据。CPU 与设备间的 I/O 接口的交互协议如下所示：

```

1 while STATUS == BUSY {}; // 等待设备执行完毕
2 DATA = data; // 把数据传给设备
3 COMMAND = command; // 发命令给设备
4 while STATUS == BUSY {}; // 等待设备执行完毕

```

引入中断机制后，这个简化的抽象设备接口需要包括四部分：状态、命令、数据、中断。CPU 与设备间的 I/O 接口的交互协议如下所示：

```

1 DATA = data; // 把数据传给设备
2 COMMAND = command; // 发命令给设备
3 do_otherwork(); // 做其它事情
4 ... // I/O 设备完成 I/O 操作，并产生中断
5 ... // CPU 执行被打断以响应中断
6 trap_handler(); // 执行中断处理例程中的相关 I/O 中断处理
7 restore_do_otherwork(); // 恢复 CPU 之前被打断的执行
8 ... // 可继续进行 I/O 操作

```

中断机制允许 CPU 的高速计算与外设的慢速 I/O 操作可以重叠 (overlap)，CPU 不用花费时间等待外设执行的完成，这样就形成 CPU 与外设的并行执行，这是提高 CPU 利用率和系统效率的关键。

站在软件的角度来看，为提高一大块数据传输效率引入的 DMA 机制并没有改变抽象设备接口的四个部分。仅仅是上面协议伪码中的 data 变成了 data block。这样传输单个数据产生的中断频度会大大降低，从而进一步提高 CPU 利用率和系统效率。

这里描述了站在软件角度上的抽象设备接口的交互协议。如果站在操作系统的角度，还需把这种设备抽象稍微再具体一点，从而能够在操作系统中实现对设备的管理。

## 基于文件的 I/O 设备抽象

在二十世纪七十到八十年代，计算机专家为此进行了诸多的探索，希望能给 I/O 设备提供一个统一的抽象。首先是把本来专门针对存储类 I/O 设备的文件进行扩展，认为所有的 I/O 设备都是文件，这就是传统 UNIX 中常见的设备文件。所有的 I/O 设备按照文件的方式进行处理。你可以在 Linux 下执行如下命令，看到各种各样的设备文件：

```
$ ls /dev
i2c-0 gpiochip0 nvme0 tty0 rtc0 ...
```

这些设备按照文件的访问接口（即 open/close/read/write）来进行处理。但由于各种设备的功能繁多，仅仅靠 read/write 这样的方式很难有效地与设备交互。于是 UNIX 的后续设计者提出了一个非常特别的系统调用 ioctl，即 input/output control 的含义。它是一个专用于设备输入输出操作的系统调用，该调用传入一个跟设备有关的请求码，系统调用的功能完全取决于设备驱动程序对请求码的解读和处理。比如，CD-ROM 驱动程序可以弹出光驱，于是操作系统就可以设定一个 ioctl 的请求码来对应这种操作。当应用程序发出带有 CD-ROM 设备文件描述符和 弹出光驱请求码这两个参数的 ioctl 系统调用请求后，操作系统中的 CD-ROM 驱动程序会识别出这个请求码，并进行弹出光驱的 I/O 操作。

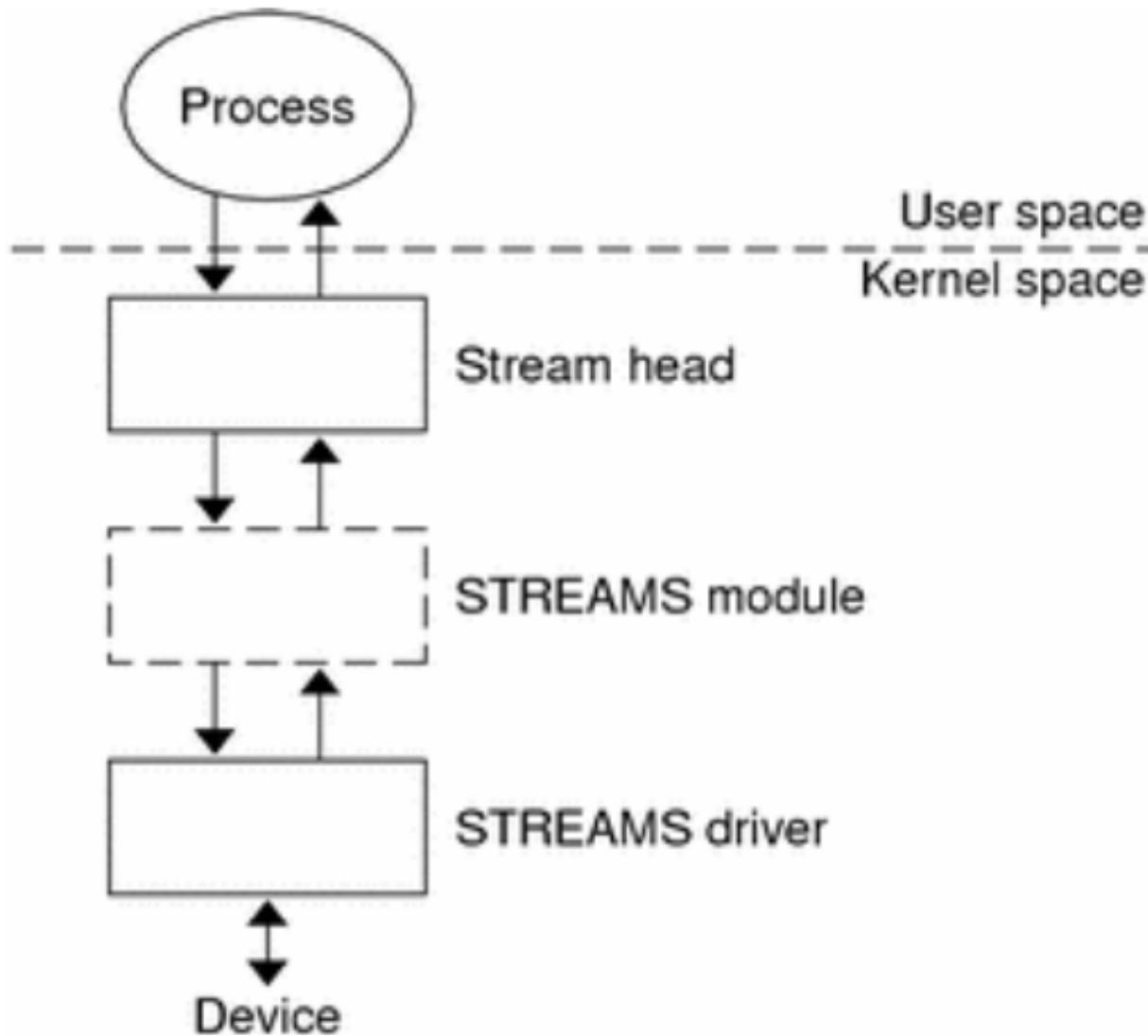
ioctl 这名字第一次出现在 Unix 第七版中，他在很多类 unix 系统（比如 Linux、Mac OSX 等）都有提供，不过不同系统的请求码对应的设备有所不同。Microsoft Windows 在 Win32 API 里提供了相似的函数，叫做 DeviceIoControl。

表面上看，基于设备文件的设备管理得到了大部分通用操作系统的支持，且这种 ioctl 系统调用很灵活，但它的问题是太灵活了，请求码的定义无规律可循，文件的接口太面向用户应用，并没有挖掘出操作系统在进行 I/O 设备处理过程中的共性特征。所以文件这个抽象还不足覆盖到操作系统对设备进行管理的整个执行过程中。

## 基于流的 I/O 设备抽象

在二十世纪八十年代的 UNIX 操作系统的发展过程中，出现了网络等更加复杂的设备，也随之出现了流 stream 这样的面向 I/O 设备管理的抽象。Dennis M. Ritchie 在 1984 年写了一个技术报告 “A Stream Input-Output System”，详细介绍了基于流的 I/O 设备的抽象设计。现在看起来，是希望把 UNIX 中的管道 (pipe) 机制拓展到内核的设备驱动中。

流是用户进程和设备或伪设备之间的全双工连接。它由几个线性连接的处理模块 (module) 组成，类似于一个 shell 程序中的管道 (pipe)，用于数据双向流动。流中的模块通过向邻居模块传递消息来进行通信。除了一些用于流量控制的常规变量，模块不需要访问其邻居模块的其他数据。此外，一个模块只为每个邻居提供一个入口点，即一个接受消息的例程。

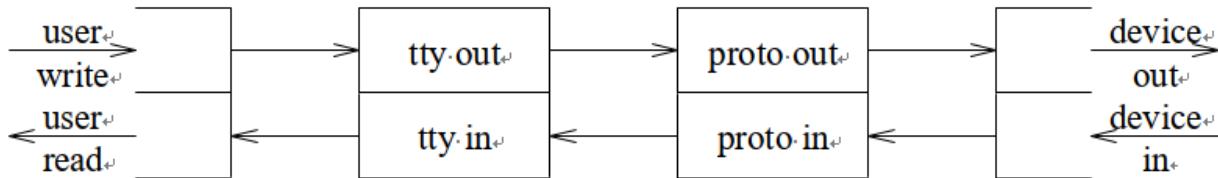


在最接近进程的流的末端是一组例程，它们为操作系统的其余部分提供接口。用户进程的写操作请求和输入/输出控制请求被转换成发送到流的消息，而读请求将从流中获取数据并将其传递给用户进程。流的另一端是设备驱动程序模块。对字符或网络传输而言，从用户进程以流的方式传递数据将被发送到设备；设备检测到的字符、网络包和状态转换被合成为消息，并被发送到流向用户进程的流中。整个过程会经过多个中间模块，这些模块会以各种方式处理或过滤消息。

在具体实现上，当设备打开时，流中的两个末端管理的内核模块自动连接；中间模块是根据用户程序的请求

动态附加的。为了能够方便动态地插入不同的流处理模块，这些中间模块的读写接口遵从相同的语义约束并互相兼容。

每个流处理模块由一对队列（queue）组成，每个方向一个队列。队列不仅包括数据队列本身，还包括两个例程和一些状态信息。一个是 put 例程，它由邻居模块调用以将消息放入数据队列中。另一个是服务（service）例程，被安排在有工作要做的时候执行。状态信息包括指向下游下一个队列的指针、各种标志以及指向队列实例化所需的附加状态信息的指针。



虽然基于流的 I/O 设备抽象看起来很不错，但并没有在其它操作系统中推广开来。其中的一个原因是 UNIX 在当时还是一个曲高和寡的高端软件系统，运行在高端的工作站和服务器上，支持的外设有限。而 Windows 这样的操作系统与 Intel 的 x86 形成了 wintel 联盟，在个人计算机市场被广泛使用，并带动了多媒体、GUI 等相关外设的广泛发展，Windows 操作系统并没有采用流的 I/O 设备抽象，而是针对每类设备定义了一套 Device Driver API 接口，提交给外设厂商，让外设厂商写好相关的驱动程序，并加入到 Windows 操作系统中。这种相对实用的做法再加上微软的号召力让各种外设得到了 Windows 操作系统的支持，但也埋下了标准不统一，容易包含 bug 的隐患。

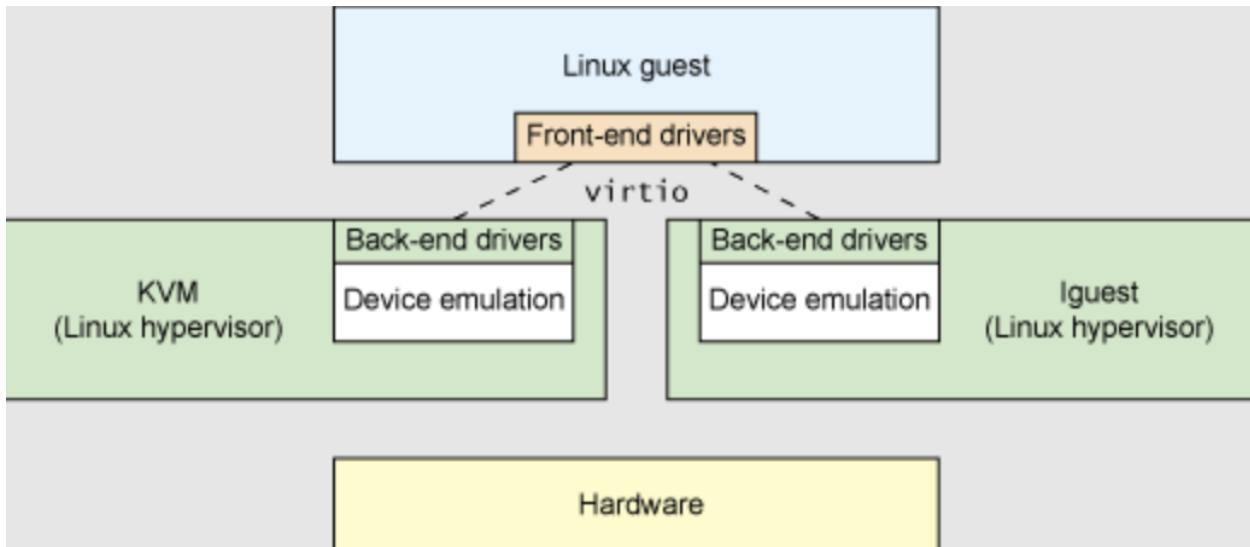
## 基于 virtio 的 I/O 设备抽象

到了二十一世纪，对于操作系统如何有效管理 I/O 设备的相关探索还在继续，但环境已经有所变化。随着互联网和云计算的兴起，在数据中心的物理服务器上通过虚拟机技术（Virtual Machine Monitor, Hypervisor 等），运行多个虚拟机（Virtual Machine），并在虚拟机中运行 guest 操作系统的模式成为一种主流。但当时存在多种虚拟机技术，如 Xen、VMware、KVM 等，要支持虚拟化 x86、Power 等不同的处理器和各种具体的外设，并都要求让以 Linux 为代表的 guest OS 能在其上高效的运行。这对于虚拟机和操作系统来说，实在是太繁琐和困难了。

IBM 资深工程师 Rusty Russell 在开发 Lguest (Linux 内核中的一个 hypervisor (一种高效的虚拟计算机的系统软件)) 时，深感写模拟计算机中的高效虚拟 I/O 设备的困难，且编写 I/O 设备的驱动程序繁杂且很难形成一种统一的表示。于是他经过仔细琢磨，提出了一组通用 I/O 设备的抽象—virtio 规范。虚拟机（VMM 或 Hypervisor）提供 virtio 设备的实现，virtio 设备有着统一的 virtio 接口，guest 操作系统只要能够实现这些通用的接口，就可以管理和控制各种 virtio 设备。而虚拟机与 guest 操作系统的 virtio 设备驱动程序间的通道是基于共享内存的异步访问方式来实现的，效率很高。虚拟机会进一步把相关的 virtio 设备的 I/O 操作转换成物理机上的物理外设的 I/O 操作。这就完成了整个 I/O 处理过程。

由于 virtio 设备的设计，使得虚拟机不用模拟真实的外设，从而可以设计一种统一和高效的 I/O 操作规范来让 guest 操作系统处理各种 I/O 操作。这种 I/O 操作规范其实就形成了基于 virtio 的 I/O 设备抽象，并逐渐形成了事实上的虚拟 I/O 设备的标准。

外部设备为 CPU 提供存储、网络等多种服务，是计算机系统中除运算功能之外最为重要的功能载体。CPU 与外设之间通过某种协议传递命令和执行结果；virtio 协议最初是为虚拟机外设而设计的 IO 协议，但是随着应用范围逐步扩展到物理机外设，virtio 协议正朝着更适合物理机使用的方向而演进。



由于 virtio 具有相对的通用性和代表性，本章将进一步分析 virtio 规范，以及针对多种 virtio 设备的设备驱动程序，从而对设备驱动程序和操作系统其他部分的关系有一个更全面的了解。

**注解：**Rusty Russell 工程师在 2008 年在“ACM SIGOPS Operating Systems Review”期刊上发表了一篇论文“virtio: towards a de-facto standard for virtual I/O devices”，提出了给虚拟环境（Virtual Machine）中的操作系统提供一套统一的设备抽象，这样操作系统针对每类设备只需写一种驱动程序就可以了，这极大降低了系统虚拟机（Virtual Machine Monitor）和 Hypervisor，以及运行在它们提供的虚拟环境中的操作系统的开发成本，且可以显著提高 I/O 的执行效率。目前 virtio 已经有相应的规范，最新的 virtio spec 版本是 v1.1。

### 10.2.5 I/O 执行模型

从用户进程的角度看，用户进程是通过 I/O 相关的系统调用（简称 I/O 系统调用）来进行 I/O 操作的。在 UNIX 环境中，I/O 系统调用有多种不同类型的执行模型。根据 Richard Stevens 的经典书籍“UNIX Network Programming Volume 1: The Sockets Networking”的 6.2 节“I/O Models”的介绍，大致可以分为五种 I/O 执行模型（I/O Execution Model，简称 IO Model, IO 模型）：

- blocking IO
- nonblocking IO
- IO multiplexing
- signal driven IO
- asynchronous IO

当一个用户进程发出一个 `read` I/O 系统调用时，主要经历两个阶段：

1. 等待数据准备好 (Waiting for the data to be ready)
2. 把数据从内核拷贝到用户进程中 (Copying the data from the kernel to the process)

上述五种 IO 模型在这两个阶段有不同的处理方式。需要注意，阻塞与非阻塞关注的是进程的执行状态：

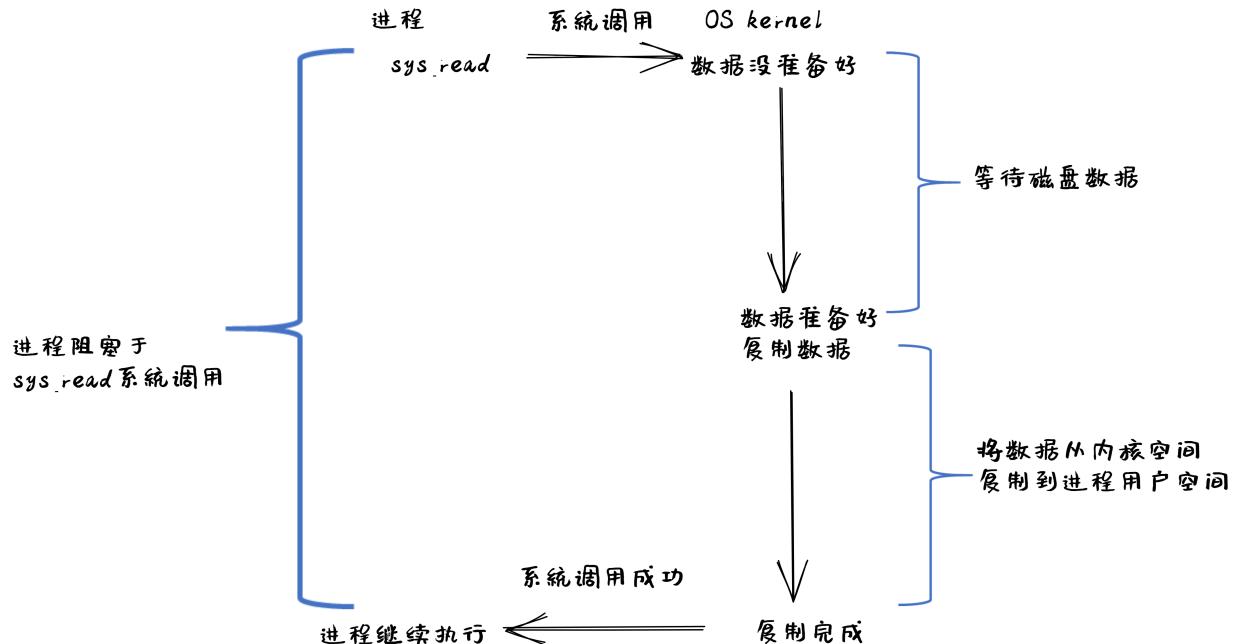
- 阻塞：进程执行系统调用后会被阻塞
- 非阻塞：进程执行系统调用后不会被阻塞

同步和异步关注的是消息通信机制：

- 同步：用户进程与操作系统（设备驱动）之间的操作是经过双方协调的，步调一致的
- 异步：用户进程与操作系统（设备驱动）之间并不需要协调，都可以随意进行各自的操作

## 阻塞 IO (blocking IO)

基于阻塞 IO 模型的文件读系统调用 `read` 的执行过程如下图所示：



从上图可以看出执行过程包含如下步骤：

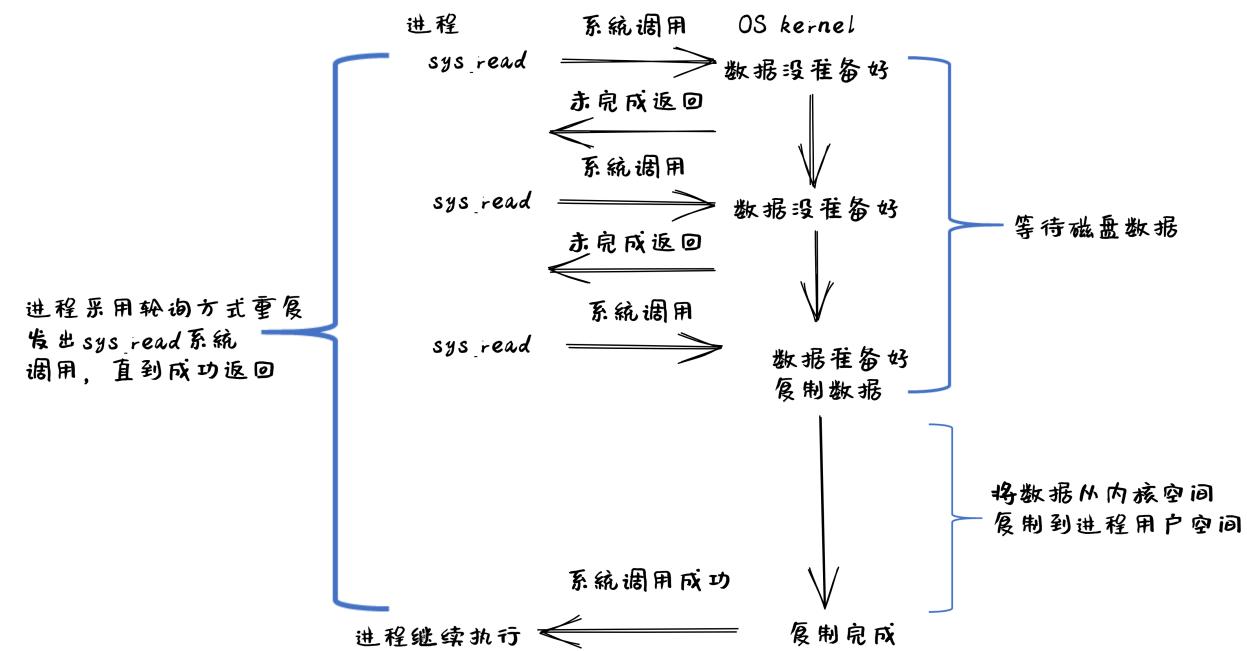
1. 用户进程发出 `read` 系统调用；
2. 内核发现所需数据没在 I/O 缓冲区中，需要向磁盘驱动程序发出 I/O 操作，并让用户进程处于阻塞状态；
3. 磁盘驱动程序把数据从磁盘传到 I/O 缓冲区后，通知内核（一般通过中断机制），内核会把数据从 I/O 缓冲区拷贝到用户进程的 buffer 中，并唤醒用户进程（即用户进程处于就绪态）；
4. 内核从内核态返回到用户态的进程，此时 `read` 系统调用完成。

所以阻塞 IO (blocking IO) 的特点就是用户进程在 I/O 执行的两个阶段（等待数据和拷贝数据两个阶段）都是阻塞的。

当然，如果正好用户进程所需数据位于内存中，那么内核会把数据从 I/O 缓冲区拷贝到用户进程的 buffer 中，并从内核态返回到用户态的进程，`read` 系统调用完成。这个由于 I/O 缓冲带了的优化结果不会让用户进程处于阻塞状态。

## 非阻塞 IO (non-blocking IO)

基于非阻塞 IO 模型的文件读系统调用-read 的执行过程如下图所示：



从上图可以看出执行过程包含如下步骤：

1. 用户进程发出 read 系统调用；
2. 内核发现所需数据没在 I/O 缓冲区中，需要向磁盘驱动程序发出 I/O 操作，并不会让用户进程处于阻塞状态，而是立刻返回一个 error；
3. 用户进程判断结果是一个 error 时，它就知道数据还没有准备好，于是它可以再次发送 read 操作（这一步操作可以重复多次）；
4. 磁盘驱动程序把数据从磁盘传到 I/O 缓冲区后，通知内核（一般通过中断机制），内核在收到通知且再次收到了用户进程的 system call 后，会马上把数据从 I/O 缓冲区拷贝到用户进程的 buffer 中；
5. 内核从内核态返回到用户态的进程，此时 read 系统调用完成。

所以，在非阻塞式 IO 的特点是用户进程不会被内核阻塞，而是需要用户进程不断的主动询问内核所需数据准备好了没有。非阻塞系统调用相比于阻塞系统调用的差异在于在被调用之后会立即返回。

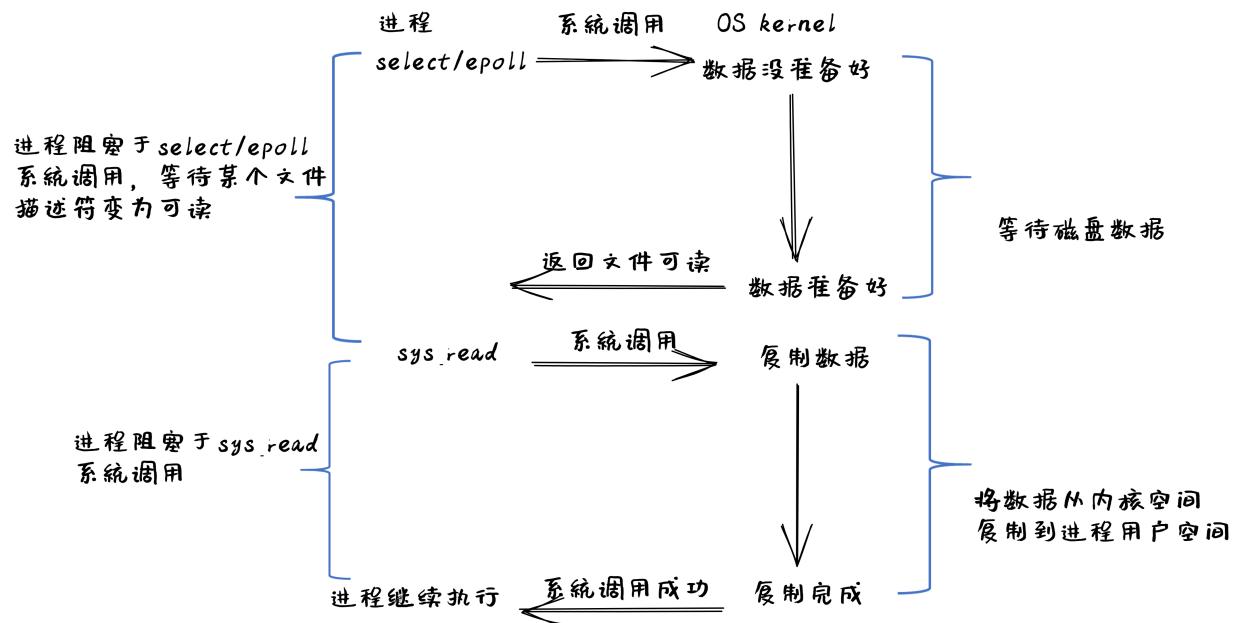
使用系统调用 `fcntl(fd, F_SETFL, O_NONBLOCK)` 可以将对某文件句柄 `fd` 进行的读写访问设为非阻塞 IO 模型的读写访问。

## 多路复用 IO (IO multiplexing)

IO multiplexing 对应的 I/O 系统调用是 `select` 和 `epoll` 等，也称这种 IO 方式为事件驱动 IO(event driven IO)。`select` 和 `epoll` 的优势在于，采用单进程方式就可以同时处理多个文件或网络连接的 I/O 操作。其基本工作机制就是通过 `select` 或 `epoll` 系统调用不断的轮询用户进程关注的所有文件句柄或 socket，当某个文件句柄或 socket 有数据到达了，`select` 或 `epoll` 系统调用就会返回到用户进程，用户进程再调用 `read` 系统调用，让内核将数据从内核的 I/O 缓冲区拷贝到用户进程的 buffer 中。

在多路复用 IO 模型中，对于用户进程关注的每一个文件句柄或 socket，一般都设置成为 non-blocking，只是用户进程是被 `select` 或 `epoll` 系统调用阻塞住了。`select`/`epoll` 的优势并不会导致单个文件或 socket 的 I/O 访问性能更好，而是在有很多个文件或 socket 的 I/O 访问情况下，其总体效率会高。

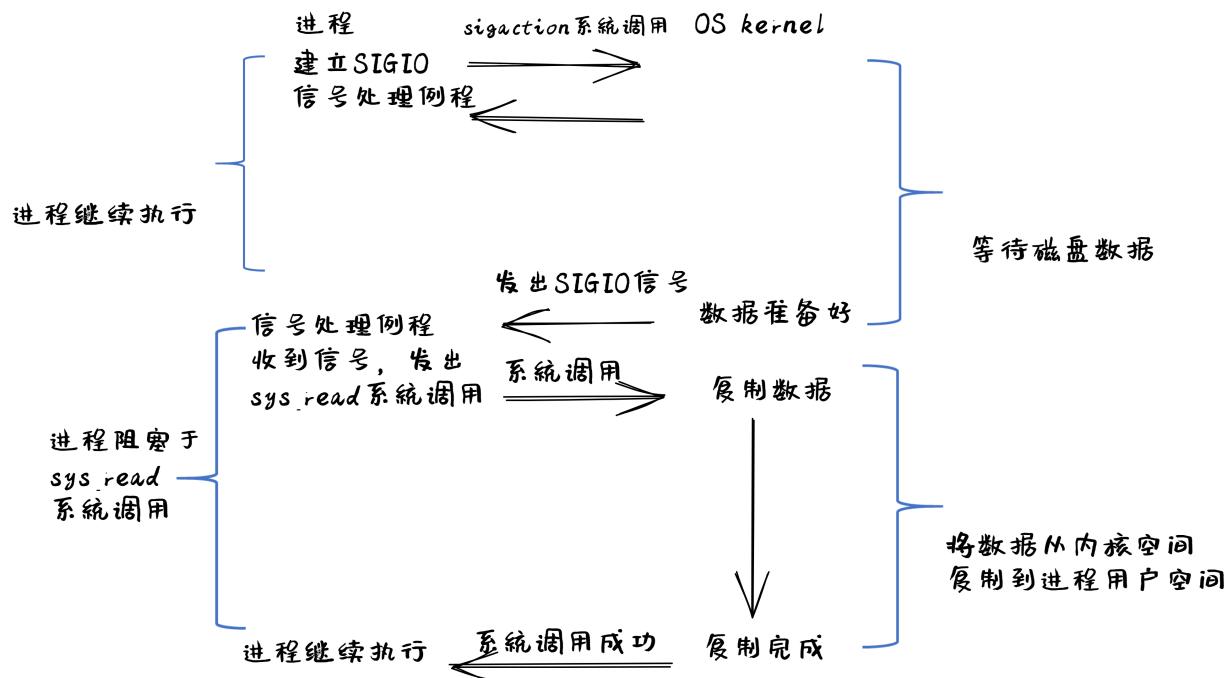
基于多路复用 IO 模型的文件读的执行过程如下图所示：



## 信号驱动 IO (signal driven I/O)

当进程发出一个 `read` 系统调用时，会向内核注册一个信号处理函数，然后系统调用返回。进程不会被阻塞，而是继续执行。当内核中的 IO 数据就绪时，会发送一个信号给进程，进程便在信号处理函数中调用 `IO 读取数据`。此模型的特点是，采用了回调机制，这样开发和调试应用的难度加大。

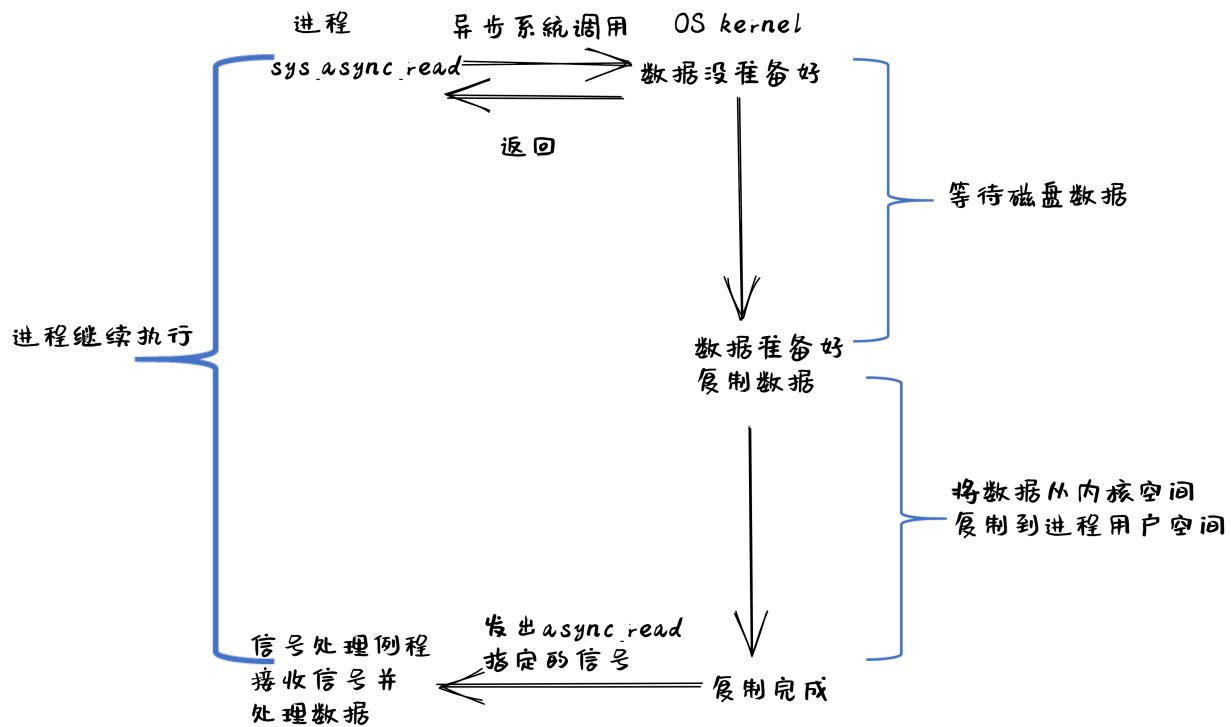
基于信号驱动 IO 模型的文件读的执行过程如下图所示：



## 异步 IO (Asynchronous I/O)

用户进程发起 `async_read` 异步系统调用之后，立刻就可以开始去做其它的事。而另一方面，从内核的角度看，当它收到一个 `async_read` 异步系统调用之后，首先它会立刻返回，所以不会对用户进程产生任何阻塞情况。然后，kernel 会等待数据准备完成，然后将数据拷贝到用户内存，当这一切都完成之后，kernel 会通知用户进程，告诉它 read 操作完成了。

基于异步 IO 模型的文件读的执行过程如下图所示：



### 注解: Linux 异步 IO 的历史

2003 年，Suparna Bhattacharya 提出了 Async I/O 在 Linux kernel 的设计方案，里面谈到了用 Full async state machine 模型来避免阻塞，把一系列的阻塞点用状态机来驱动，把用户态的 buffer 映射到内核来驱动，这个模型被应用到 Linux kernel 2.4 中。在出现 `io_uring` 之前，虽然还出现了一系列的异步 IO 的探索 (`syslet`、`LCA`、`FSAIO`、`AIO-epoll` 等)，但性能一般，实现和使用复杂，应该说 Linux 没有提供完善的异步 IO(网络 IO、磁盘 IO) 机制。`io_uring` 是由 Jens Axboe 提供的异步 I/O 接口，`io_uring` 围绕高效进行设计，采用一对共享内存 `ringbuffer` 用于应用和内核间通信，避免内存拷贝和系统调用。`io_uring` 的实现于 2019 年 5 月合并到了 Linux kernel 5.1 中，现在已经在多个项目中被使用。

## 五种 IO 执行模型对比

这里总结一下阻塞 IO、非阻塞 IO、同步 IO、异步 IO 的特点：

- 阻塞 IO：在用户进程发出 IO 系统调用后，进程会等待该 IO 操作完成，而使得进程的其他操作无法执行。
- 非阻塞 IO：在用户进程发出 IO 系统调用后，如果数据没准备好，该 IO 操作会立即返回，之后进程可以进行其他操作；如果数据准备好了，用户进程会通过系统调用完成数据拷贝并接着进行数据处理。
- 同步 IO：导致请求进程阻塞/等待，直到 I/O 操作完成。
- 异步 IO：不会导致请求进程阻塞。

从上述分析可以得知，阻塞和非阻塞的区别在于内核数据还没准备好时，用户进程是否会阻塞（第一阶段是否阻塞）；同步与异步的区别在于当数据从内核 copy 到用户空间时，用户进程是否会阻塞/参与（第二阶段是否阻塞）。

所以前述的阻塞 IO (blocking IO)，非阻塞 IO (non-blocking IO)，多路复用 IO (IO multiplexing)，信号驱动 IO 都属于同步 IO (synchronous IO)。这四种模型都有一个共同点：在第二阶段阻塞/参与，也就是在真正 IO 操作 `read` 的时候需要用户进程参与，因此以上四种模型均称为同步 IO 模型。

有人可能会说，执行非阻塞 IO 系统调用的用户进程并没有被阻塞。其实这里定义中所指的 **IO 操作**是指实际的 **IO 操作**。比如，非阻塞 IO 在执行 `read` 系统调用的时候，如果内核中的 IO 数据没有准备好，这时候不会 block 进程。但是当内核中的 IO 数据准备好且收到用户进程发出的 `read` 系统调用时（处于第二阶段），内核中的 `read` 系统调用的实现会将数据从 kernel 拷贝到用户内存中，这个时候进程是可以被阻塞的。

而异步 IO 则不一样，当用户进程发起 IO 操作之后，就直接返回做其它事情去了，直到内核发送一个通知，告诉用户进程说 IO 完成。在这整个过程中，用户进程完全没有被阻塞。

## 10.3 外设平台

### 10.3.1 本节导读

现在我们有了对设备的基本了解，接下来就要考虑如何编写驱动程序来控制各种外设了。本节首先讲述了驱动程序要完成的主要功能，包括初始化设备，接收用户进程的 I/O 请求并给设备发出 I/O 命令，响应设备发出的通知，完成用户进程的 I/O 请求。然后介绍了计算机硬件系统中除了 CPU/内存之外的其他重要的外设和相关 I/O 控制器，以及如何通过编程来获取外设相关信息。

### 10.3.2 驱动程序概述

很难为驱动程序提供一个精确的定义。基本而言，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：

1. 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
2. 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
3. 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）
4. 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

## 5. 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。注：这部分可以通过协程技术来实现，但目前我们不会就此展开讨论。

编写驱动程序代码其实需要的知识储备还是比较多的，需要注意如下的一些内容：

1. 了解硬件规范：从而能够正确地与硬件交互，并能处理访问硬件出错的情况；
2. 了解操作系统，由于驱动程序与它所管理的设备会同时执行，也可能与操作系统其他模块并行/并发访问相关共享资源，所以需要考虑同步互斥的问题（后续会深入讲解操作系统同步互斥机制），并考虑到申请资源失败后的处理；
3. 理解驱动程序执行中所在的可能的上下文环境：如果是在进行中断处理（如在执行 `trap_handler` 函数），那是在中断上下文中执行；如果是在代表进程的内核线程中执行后续的 I/O 操作（如收发 TCP 包），那是在内核线程上下文执行。这样才能写出正确的驱动程序。

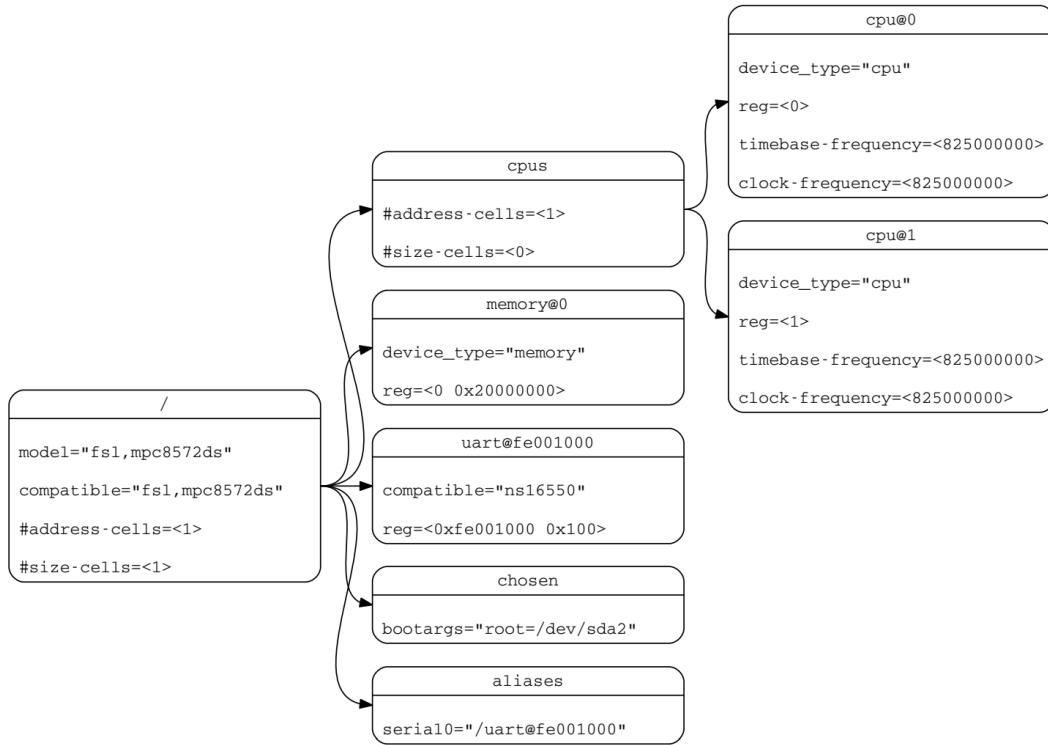
### 10.3.3 硬件系统架构

#### 设备树

首先，我们需要了解 OS 管理的计算机硬件系统—QEMU riscv-64 virt machine，特别是其中的各种外部设备。`virt` 表示了一台虚拟的 RISC-V 64 计算机，CPU 的个数是可以通过参数 `-cpu num` 配置的，内存也是可通过参数 `-m numM/G` 来配置。这台虚拟计算机还有很多外设信息，每个设备在物理上连接到了父设备上最后再通过总线等连接起来构成一整个设备树。QEMU 可以把它模拟的机器细节信息全都导出到 `dtb` 格式的二进制文件中，并可通过 `dtc`（Device Tree Compiler）工具转成可理解的文本文件。如想详细了解这个文件的格式说明可以参考 [Devicetree Specification](#)。

```
$ qemu-system-riscv64 -machine virt -machine dumpdtb=riscv64-virt.dtb -bios default
qemu-system-riscv64: info: dtb dumped to riscv64-virt.dtb. Exiting.
$ dtc -I dtb -O dts -o riscv64-virt.dts riscv64-virt.dtb
$ less riscv64-virt.dts
就可以看到 QEMU RV64_
↳ virt 计算机的详细硬件（包括各种外设）细节，包括 CPU，内存，串口，时钟和各种 virtio 设备的信息。
```

一个典型的设备树如下图所示：



### 注解：设备树与设备节点属性

设备树（Device Tree）是一种数据结构，用于表示硬件系统的结构和功能。它是一个文本文件，描述了硬件系统的结构和功能，并将这些信息提供给操作系统。设备树包含了关于硬件系统的信息，如：

- 处理器的类型和数量
- 板载设备（如存储器、网卡、显卡等）的类型和数量
- 硬件接口（如 I2C、SPI、UART 等）的类型和地址信息

设备树中的节点是用来描述硬件设备的信息的。一个设备树节点包含了一个或多个属性，每个属性都是一个键-值对，用来描述设备的某一特定信息。而操作系统就是通过这些节点上的信息来实现对设备的识别和初始化。具体而言，一个设备节点上会有一些常见的属性：

- **compatible**：表示设备的类型，可以是设备的厂商名、产品名等，如“virtio,mmio”指的是这个设备通过 virtio 协议、MMIO（内存映射 I/O）方式来驱动
- **reg**：表示设备在系统中的地址空间位置
- **interrupts**：表示设备支持的中断信号

设备树在很多嵌入式系统中都得到了广泛应用，它是一种常用的方法，用于将硬件（特别是外设）信息传递给操作系统。在桌面和服务器系统中，PCI 总线可以起到设备树的作用，通过访问 PCI 总线上特定地址空间，也可以遍历出具有挂在 PCI 总线上的各种 PCI 设备。

我们可以运行 `virtio_drivers` crate 中的一个在裸机环境下的测试用例，来动态查看 `qemu-system-riscv64` 模拟的 `virt` 计算机的设备树信息：

```

获取virtio_driver git仓库源码
$ git clone https://github.com/rcore-os/virtio-drivers.git
在qemu模拟器上运行测试用例:
$ cd virtio-drivers/examples/riscv
$ make qemu
qemu命令行参数
qemu-system-riscv64 \
 -machine virt \
 -serial mon:stdio \
 -bios default \
 -kernel target/riscv64imac-unknown-none-elf/release/riscv \
 -global virtio-mmio.force-legacy=false \
 -drive file=target/riscv64imac-unknown-none-elf/release/img,if=none,format=raw,
 ↪id=x0 \
 -device virtio-blk-device,drive=x0 \
 -device virtio-gpu-device \
 -device virtio-mouse-device \
 -device virtio-net-device
...

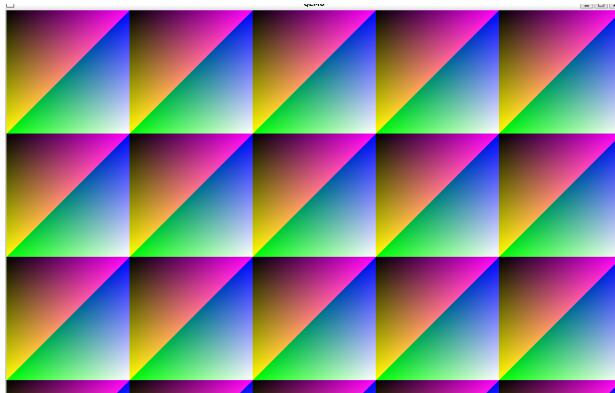
```

在上面的 `qemu` 命令行参数中，可以看到 `virt` 计算机中配置了基于 `virtio` 协议的存储块设备 `virtio-blk-device`、图形显示设备 `virtio-gpu-device`、鼠标设备 `virtio-mouse-device` 和网卡设备 `virtio-net-device`。通过看到测试用例扫描出的设备树信息，且可以看到通过 `virtio_gpu` 显示的漂亮的图形：

```

[INFO] device tree @ 0x87000000
[INFO] walk dt addr=0x10008000, size=0x1000
[INFO] Device tree node virtio_mmio@10008000: Some("virtio,mmio")
[INFO] Detected virtio MMIO device with vendor id 0x554D4551, device type Block, ↪
 ↪version Modern
[INFO] device features: SEG_MAX | GEOMETRY | BLK_SIZE | FLUSH | TOPOLOGY | CONFIG_
 ↪WCE | DISCARD | WRITE_ZEROES | RING_INDIRECT_DESC | RING_EVENT_IDX | VERSION_1
[INFO] config: 0x10008100
[INFO] found a block device of size 16KB
[INFO] virtio-blk test finished
[INFO] walk dt addr=0x10007000, size=0x1000
[INFO] Device tree node virtio_mmio@10007000: Some("virtio,mmio")
[INFO] Detected virtio MMIO device with vendor id 0x554D4551, device type GPU, ↪
 ↪version Modern
[INFO] Device features EDID | RING_INDIRECT_DESC | RING_EVENT_IDX | VERSION_1
[INFO] events_read: 0x0, num_scanouts: 0x1
[INFO] GPU resolution is 1280x800
[INFO] => RespDisplayInfo { header: CtrlHeader { hdr_type: OkDisplayInfo, flags: 0, ↪
 ↪fence_id: 0, ctx_id: 0, _padding: 0 }, rect: Rect { x: 0, y: 0, width: 1280, ↪
 ↪height: 800 }, enabled: 1, flags: 0 }
[INFO] virtio-gpu test finished
[INFO] walk dt addr=0x10006000, size=0x1000
[INFO] Device tree node virtio_mmio@10006000: Some("virtio,mmio")
[INFO] Detected virtio MMIO device with vendor id 0x554D4551, device type Input, ↪
 ↪version Modern
[INFO] Device features: RING_INDIRECT_DESC | RING_EVENT_IDX | VERSION_1
[INFO] walk dt addr=0x10005000, size=0x1000
[INFO] Device tree node virtio_mmio@10005000: Some("virtio,mmio")
[INFO] Detected virtio MMIO device with vendor id 0x554D4551, device type Network, ↪
 ↪version Modern
[INFO] Device features CTRL_GUEST_OFFLOADS | MAC | MRG_RXBUF | STATUS | CTRL_VQ | ↪
 ↪CTRL_RX | CTRL_VLAN | CTRL_RX_EXTRA | GUEST_ANNOUNCE | CTL_MAC_ADDR | RING_INDIRECT_
 ↪DESC | RING_EVENT_IDX | VERSION_1

```



在上述输出中，我们看到了 *type* 为 *Block*、*GPU*、*Input* 和 *Network* 的设备，所以我们的测例确实通过发现了这些设备，还通过 *GPU* 设备进行操作，让我们终于可以看到图形了。

### 传递设备树信息

操作系统在启动后需要了解计算机系统中所有接入的设备，这就要有一个读取全部已接入设备信息的能力，而设备信息放在哪里，又是谁帮我们来做的呢？在 RISC-V 中，这个一般是由 bootloader，即 OpenSBI or RustSBI 固件完成的。它来完成对于包括物理内存在内的各外设的探测，将探测结果以 **设备树二进制对象 (DTB, Device Tree Blob)** 的格式保存在物理内存中的某个地方。然后 bootloader 会启动操作系统，即把放置 DTB 的物理地址将放在 a1 寄存器中，而将会把 HART ID (HART, Hardware Thread, 硬件线程，可以理解为执行的 CPU 核) 放在 a0 寄存器上，然后跳转到操作系统的入口地址处继续执行。

在 `virtio_drivers/examples/riscv` 目录下，我们可以看到 `main.rs` 文件，它是一个裸机环境下的测试用例，它会在启动后打印出设备树信息：

```

1 //virtio_drivers/examples/riscv/src/main.rs
2 #[no_mangle]
3 extern "C" fn main(_hartid: usize, device_tree_paddr: usize) {
4 ...
5 init_dt(device_tree_paddr);
6 ...
7 }
8
9 fn init_dt(dtb: usize) {
10 info!("device tree @ {:#x}", dtb);
11 // Safe because the pointer is a valid pointer to unaliased memory.
12 let fdt = unsafe { Fdt::from_ptr(dtb as *const u8).unwrap() };
13 walk_dt(fdt);
14 }
15
16 fn walk_dt(fdt: Fdt) {
17 for node in fdt.all_nodes() {
18 if let Some(compatible) = node.compatible() {
19 if compatible.all().any(|s| s == "virtio,mmio") {
20 virtio_probe(node);
21 }
22 }
23 }
24 }
```

我们只需要给 `main` 函数增加两个参数（即 a0 和 a1 寄存器中的值）即可，这样测试用例就获得了 bootloader 传来的放置 DTB 的物理地址。然后 `init_dt` 函数会将这个地址转换为 `Fdt` 类型，然后遍历整个设备树，找

到所有的 virtio, mmio 设备 (其实就是 QEMU 模拟的各种 virtio 设备), 然后调用 virtio\_probe 函数来显示设备信息并初始化这些设备。

### 解析设备树信息

virtio\_probe 函数会进一步查找 virtio 设备节点中的 ‘reg’ 属性, 从而可以找到 virtio 设备的具体类型 (如 *DeviceType::Block* 块设备类型) 等参数。这样我们就可以对具体的 virtio 设备进行初始化和进行具体 I/O 操作了。virtio\_probe 函数的主体部分如下所示:

```

1 fn virtio_probe(node: FdtNode) {
2 //分析 reg 信息
3 if let Some(reg) = node.reg().and_then(|mut reg| reg.next()) {
4 let paddr = reg.starting_address as usize;
5 let size = reg.size.unwrap();
6 let vaddr = paddr;
7 info!("walk dt addr={:#x}, size={:#x}", paddr, size);
8 info!({
9 "Device tree node {}: {:?}", node.name,
10 node.compatible().map(Compatible::first),
11 });
12 let header = NonNull::new(vaddr as *mut VirtIOHeader).unwrap();
13 //判断virtio设备类型
14 match unsafe { MmioTransport::new(header) } {
15 Err(e) => warn!("Error creating VirtIO MMIO transport: {}", e),
16 Ok(transport) => {
17 info!({
18 "Detected virtio MMIO device with vendor id {:#X}, device type {:?},",
19 & version {:?},
20 transport.vendor_id(),
21 transport.device_type(),
22 transport.version(),
23 });
24 virtio_device(transport);
25 }
26 }
27 }
28 }
29 // 对不同的virtio设备进行进一步的初始化工作
30 fn virtio_device(transport: impl Transport) {
31 match transport.device_type() {
32 DeviceType::Block => virtio_blk(transport),
33 DeviceType::GPU => virtio_gpu(transport),
34 DeviceType::Input => virtio_input(transport),
35 DeviceType::Network => virtio_net(transport),
36 t => warn!("Unrecognized virtio device: {:?}", t),
37 }
38 }
```

显示图形的操作其实很简单, 都在 virtio\_gpu 函数中:

```

1 fn virtio_gpu<T: Transport>(transport: T) {
2 let mut gpu = VirtIOGpu::<HalImpl, T>::new(transport).expect("failed to create gpu-");
3 // 获得显示设备的长宽信息
4 let (width, height) = gpu.resolution().expect("failed to get resolution");

```

(下页继续)

(续上页)

```

5 let width = width as usize;
6 let height = height as usize;
7 info!("GPU resolution is {}x{}", width, height);
8 // 设置显示缓冲区
9 let fb = gpu.setup_FRAMEBUFFER().expect("failed to get fb");
10 // 设置显示设备中的每个显示点的红、绿、蓝分量值, 形成丰富色彩的图形
11 for y in 0..height {
12 for x in 0..width {
13 let idx = (y * width + x) * 4;
14 fb[idx] = x as u8;
15 fb[idx + 1] = y as u8;
16 fb[idx + 2] = (x + y) as u8;
17 }
18 }
19 gpu.flush().expect("failed to flush");
20 info!("virtio-gpu test finished");
21 }

```

可以发现, 对各种设备的控制, 大部分都是基于对特定内存地址的读写来完成的, 这就是 MMIO 的 I/O 访问方式。看到这, 也许你会觉得查找、初始化和控制计算机中的设备其实没有特别复杂, 前提是你对外设的硬件规范有比较深入的了解。不过当与操作系统结合在一起后, 还需要和操作系统内部的其他内核模块 (如文件系统等) 进行交互, 复杂性就会增加。我们会逐步展开这方面的讲解。

## 平台级中断控制器

在之前的操作系统中, 已经涉及到中断处理, 但还没有处理外设 (时钟中断时 RISC-V 处理器产生的) 产生的中断。如果要让操作系统处理外设中断, 就需要对中断控制器进行初始化设置。在 RISC-V 中, 与外设连接的 I/O 控制器的一个重要组成是平台级中断控制器 (Platform-Level Interrupt Controller, PLIC), 它的一端汇聚了各种外设的中断信号, 另一端连接到 CPU 的外部中断引脚上。当一个外部设备发出中断请求时, PLIC 会将其转发给 RISC-V CPU, CPU 会执行对应的中断处理程序来响应中断。通过 RISC-V 的 mie 寄存器中的 meie 位, 可以控制这个引脚是否接收外部中断信号。当然, 通过 RISC-V 中 M Mode 的中断委托机制, 也可以在 RISC-V 的 S Mode 下, 通过 sie 寄存器中的 seie 位, 对中断信号是否接收进行控制。

### 注解: 中断控制器 (Interrupt Controller)

计算机中的中断控制器是一种硬件, 可帮助处理器处理来自多个不同 I/O 设备的中断请求 (Interrupt Request, 简称 IRQ)。这些中断请求可能同时发生, 并首先经过中断控制器的处理, 即中断控制器根据 IRQ 的优先级对同时发生的中断进行排序, 然后把优先级最高的 IRQ 传给处理器, 让操作系统执行相应的中断处理例程 (Interrupt Service Routine, 简称 ISR)。

CPU 可以通过 MMIO 方式来对 PLIC 进行管理, 下面是一些与 PLIC 相关的寄存器:

| 寄存器       | 地址          | 功能       | 描述               |
|-----------|-------------|----------|------------------|
| Priority  | 0x0c00_0000 | 设置       | 特定中断源的优先级        |
| Pending   | 0x0c00_1000 | 包含       | 已触发 (正在处理) 的中断列表 |
| Enable    | 0x0c00_2000 | 启用 / 禁用  | 某些中断源            |
| Threshold | 0x0c20_0000 | 设置       | 中断能够触发的阈值        |
| Claim     | 0x0c20_0004 | 按优先级顺序返回 | 下一个中断            |
| Complete  | 0x0c20_0004 | 写操作表示    | 完成对特定中断的处理       |

在 QEMU qemu/include/hw/riscv/virt.h 的源码中, 可以看到

```

1 enum {
2 UART0_IRQ = 10,
3 RTC_IRQ = 11,
4 VIRTIO_IRQ = 1, /* 1 to 8 */
5 VIRTIO_COUNT = 8,
6 PCIE_IRQ = 0x20, /* 32 to 35 */
7 VIRTIO_NDEV = 0x35 /* Arbitrary maximum number of interrupts */
8 };

```

可以看到串口 UART0 的中断号是 10, virtio 设备的中断号是 1~8。通过 dtc (Device Tree Compiler) 工具生成的文本文件, 我们也可以发现上述中断信号信息, 以及基于 MMIO 的外设寄存器信息。在后续的驱动程序中, 这些信息我们可以用到。

操作系统如要响应外设的中断, 需要做两方面的初始化工作。首先是完成第三章讲解的中断初始化过程, 并需要把 sie 寄存器中的 seie 位设置为 1, 让 CPU 能够接收通过 PLIC 传来的外部设备中断信号。然后还需要通过 MMIO 方式对 PLIC 的寄存器进行初始设置, 才能让外设产生的中断传到 CPU 处。其主要操作包括:

- 设置外设中断的优先级
- 设置外设中断的阈值, 优先级小于等于阈值的中断会被屏蔽
- 激活外设中断, 即把 Enable 寄存器的外设中断编号为索引的位设置为 1

上述操作的具体实现, 可以参考 ch9 分支中的内核开发板初始化代码 qemu.rs 中的 device\_init() 函数:

```

1 // os/src/boards/qemu.rs
2 pub fn device_init() {
3 use riscv::register::sie;
4 let mut plic = unsafe { PLIC::new(VIRT_PLIC) };
5 let hart_id: usize = 0;
6 let supervisor = IntrTargetPriority::Supervisor;
7 let machine = IntrTargetPriority::Machine;
8 // 设置PLIC中外设中断的阈值
9 plic.set_threshold(hart_id, supervisor, 0);
10 plic.set_threshold(hart_id, machine, 1);
11 // 使能PLIC在CPU处于S-Mode下传递键盘/鼠标/块设备/串口外设中断
12 // irq_nums: 5 keyboard, 6 mouse, 8 block, 10 uart
13 for intr_src_id in [5usize, 6, 8, 10] {
14 plic.enable(hart_id, supervisor, intr_src_id);
15 plic.set_priority(intr_src_id, 1);
16 }
17 // 设置S-Mode CPU使能中断
18 unsafe {
19 sie::set_sext();
20 }
21 }

```

但外设产生中断后, CPU 并不知道具体是哪个设备传来的中断, 这可以通过读 PLIC 的 claim 寄存器来了解。Claim 寄存器会返回 PLIC 接收到的优先级最高的中断; 如果没有外设中断产生, 读 Claim 寄存器会返回 0。

操作系统在收到中断并完成中断处理后, 还需通知 PLIC 中断处理完毕。CPU 需要在 PLIC 的 complete 寄存器中写入对应中断号为索引的位, 来通知 PLIC 中断已处理完毕。

上述操作的具体实现, 可以参考 ch9 分支的开发板初始化代码 qemu.rs 中的 irq\_handler() 函数:

```

1 // os/src/boards/qemu.rs
2 pub fn irq_handler() {
3 let mut plic = unsafe { PLIC::new(VIRT_PLIC) };

```

(下页继续)

(续上页)

```

4 // 读PLIC的 ``Claim`` 寄存器获得外设中断号
5 let intr_src_id = plic.claim(0, IntrTargetPriority::Supervisor);
6 match intr_src_id {
7 5 => KEYBOARD_DEVICE.handle_irq(),
8 6 => MOUSE_DEVICE.handle_irq(),
9 8 => BLOCK_DEVICE.handle_irq(),
10 10 => UART.handle_irq(),
11 _ => panic!("unsupported IRQ {}", intr_src_id),
12 }
13 // 通知PLIC中断已处理完毕
14 plic.complete(0, IntrTargetPriority::Supervisor, intr_src_id);
15 }

```

这样同学们就大致了解了计算机中外设的发现、初始化、I/O 处理和中断响应的基本过程。不过大家还没有在操作系统中实现面向具体外设的设备驱动程序。接下来，我们就会分析串口设备驱动、块设备设备驱动和显示设备驱动的设计与实现。

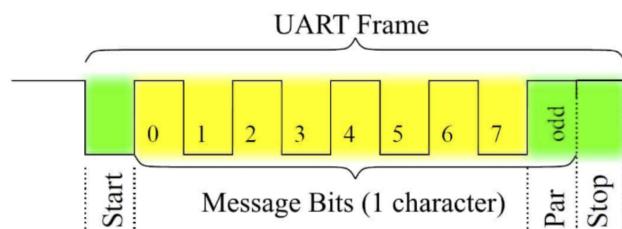
## 10.4 串口驱动程序

### 10.4.1 本节导读

现在我们对如何了解计算机系统中给的外设信息，以及如何初始化或与外设进行交互有了基本的了解。接下来，我们需要看看如何在完整的操作系统中通过添加设备驱动程序来扩展应用访问外设的 I/O 能力。本节将通过介绍一个具体的物理设备串口的驱动程序的设计与实现，来分析如何在操作系统中添加设备驱动程序。

### 10.4.2 串口驱动程序

我们要管理是串口 (UART) 物理设备。我们在第一章其实就接触了串口，但当时是通过 RustSBI 来帮 OS 完成对串口的访问，即 OS 只需发出两种 SBI 调用请求就可以输出和获取字符了。但这种便捷性是有代价的。比如 OS 在调用获取字符的 SBI 调用请求后，RustSBI 如果没收到串口字符，会返回 -1，这样 OS 只能采用类似轮询的方式来继续查询。到第七章为止的串口驱动不支持中断是导致在多进程情况下，系统效率低下的主要原因之一。大家也不要遗憾，我们的第一阶段的目标是 **Just do it**，先把 OS 做出来。在本节，我们需要逐步改进优化对串口的操作了。



串口 (Universal Asynchronous Receiver-Transmitter, 简称 UART) 是一种在嵌入式系统中常用的用于传输、接收系列数据的外部设备。串行数据传输是逐位 (bit) 顺序发送数据的过程。了解 QEMU 模拟的兼容 NS16550A 硬件规范<sup>12</sup> 是写驱动程序的准备工作，建议同学们先阅读一下。每个 UART 使用 8 个 I/O 字节来访问其寄存器。下表<sup>Page 10, 1</sup> 显示了 UART 中每个寄存器的地址和基本含义。表中使用的 *base* 是串口设备的起始地址。在 QEMU 模拟的 virt 计算机中串口设备寄存器的 MMIO 起始地址为 0x10000000。

<sup>1</sup> Serial UART information, <https://www.lammertbies.nl/comm/info/serial-uart>

<sup>2</sup> TECHNICAL DATA ON 16550, <http://www.byterunner.com/16550.html>

| I/O port | Read (DLAB=0)                       | Write (DLAB=0)                 | Read (DLAB=1)                       | Write (DLAB=1)               |
|----------|-------------------------------------|--------------------------------|-------------------------------------|------------------------------|
| base     | <b>RBR</b> receiver buffer          | <b>THR</b> transmitter holding | <b>DLL</b> divisor latch LSB        | <b>DLL</b> divisor latch LSB |
| base+1   | <b>IER</b> interrupt enable         | <b>IER</b> interrupt enable    | <b>DLM</b> divisor latch MSB        | <b>DLM</b> divisor latch MSB |
| base+2   | <b>IIR</b> interrupt identification | <b>FCR</b> FIFO control        | <b>IIR</b> interrupt identification | <b>FCR</b> FIFO control      |
| base+3   | <b>LCR</b> line control             | <b>LCR</b> line control        | <b>LCR</b> line control             | <b>LCR</b> line control      |
| base+4   | <b>MCR</b> modem control            | <b>MCR</b> modem control       | <b>MCR</b> modem control            | <b>MCR</b> modem control     |
| base+5   | <b>LSR</b> line status              | <i>factory test</i>            | <b>LSR</b> line status              | <i>factory test</i>          |
| base+6   | <b>MSR</b> modem status             | <i>not used</i>                | <b>MSR</b> modem status             | <i>not used</i>              |
| base+7   | <b>SCR</b> scratch                  | <b>SCR</b> scratch             | <b>SCR</b> scratch                  | <b>SCR</b> scratch           |

注：LCR 寄存器中的 DLAB 位设置为 0 或 1，会导致 CPU 访问的是不同的寄存器。比如，DLAB 位为 0 时，读位于 *base* 处的串口寄存器是 *RBR*，DLAB 位为 1 时，读位于 *base* 处的串口寄存器是 *DLL*。

大致猜测完上述寄存器的含义后，我们就算是完成前期准备工作，在接下来的驱动程序设计中，会用到上述的部分寄存器。我们先尝试脱离 RustSBI 的帮助，在操作系统中完成支持中断机制的串口驱动。

通过查找 dtc (Device Tree Compiler) 工具生成的 *riscv64-virt.dts* 文件，我们可以看到串口设备相关的 MMIO 模式的寄存器信息和中断相关信息。

```
...
chosen {
 bootargs = [00];
 stdout-path = "/uart@10000000";
};

uart@10000000 {
 interrupts = <0x0a>;
 interrupt-parent = <0x02>;
 clock-frequency = <0x384000>;
 reg = <0x00 0x10000000 0x00 0x100>;
 compatible = "ns16550a";
};
```

*chosen* 节点的内容表明字符输出会通过串口设备打印出来。*uart@10000000* 节点表明串口设备中寄存器的 MMIO 起始地址为 0x10000000，范围在 0x00~0x100 区间内，中断号为 0x0a。*clock-frequency* 表示时钟频率，其值为 0x38400，即 3.6864 MHz。*compatible = "ns16550a"* 表示串口的硬件规范兼容 NS16550A。

在如下情况下，串口会产生中断：

- 有新的输入数据进入串口的接收缓存
- 串口完成了缓存中数据的发送
- 串口发送出现错误

这里我们仅关注有输入数据时串口产生的中断。

在 UART 中，可访问的 I/O 寄存器一共有 8 个。访问 I/O 寄存器的方法把串口寄存器的 MMIO 起始地址加上偏移量，就是各个寄存器的 MMIO 地址了。

## 串口设备初始化

对串口进行初始化设置的相关代码如下所示：

```

1 // os/src/drivers/chardev/mod.rs
2 ...
3 lazy_static! {
4 pub static ref UART: Arc<CharDeviceImpl> = Arc::new(CharDeviceImpl::new());
5 }
6 // os/src/boards/qemu.rs
7 pub type CharDeviceImpl = crate::drivers::chardev::NS16550a<VIRT_UART>;
8 // os/src/drivers/chardev/ns16550a.rs
9 impl<const BASE_ADDR: usize> NS16550a<BASE_ADDR> {
10 pub fn new() -> Self {
11 let mut inner = NS16550aInner {
12 ns16550a: NS16550aRaw::new(BASE_ADDR),
13 read_buffer: VecDeque::new(),
14 };
15 inner.ns16550a.init();
16 Self {
17 inner: unsafe { UPIntrFreeCell::new(inner) },
18 condvar: Condvar::new(),
19 }
20 }
21 }
22 ...
23 impl NS16550aRaw {
24 pub fn init(&mut self) {
25 let read_end = self.read_end();
26 let mut mcr = MCR::empty();
27 mcr |= MCR::DATA_TERMINAL_READY;
28 mcr |= MCR::REQUEST_TO_SEND;
29 mcr |= MCR::AUX_OUTPUT2;
30 read_end.mcr.write(mcr);
31 let ier = IER::RX_AVAILABLE;
32 read_end.ier.write(ier);
33 }
34 }
```

上述代码完成的主要工作包括：

## 串口设备输入输出操作

先看串口输出，由于不设置和处理输出后产生中断的情况，使得整个输出操作比较简单。即向偏移量为 0 的串口控制寄存器的 MMIO 地址写 8 位字符即可。

```

1 // os/src/drivers/chardev/ns16550a.rs
2
3 impl<const BASE_ADDR: usize> CharDevice for NS16550a<BASE_ADDR> {
4 fn write(&self, ch: u8) {
5 let mut inner = self.inner.exclusive_access();
6 inner.ns16550a.write(ch);
7 }
8 impl NS16550aRaw {
9 pub fn write(&mut self, ch: u8) {
10 let write_end = self.write_end();
```

(下页继续)

(续上页)

```

11 loop {
12 if write_end.lsr.read().contains(LSR::THR_EMPTY) {
13 write_end.thr.write(ch);
14 break;
15 }
16 }
17 }
```

在以往的操作系统实现中，当一个进程通过 `sys_read` 系统调用来获取串口字符时，并没有用上中断机制。但一个进程读不到字符的时候，将会被操作系统调度到就绪队列的尾部，等待下一次执行的时刻。这其实也就是一种变相的轮询方式来获取串口的输入字符。这里其实是可以对进程管理做一个改进，来避免进程通过轮询的方式检查串口字符输入。既然我们已经在上一章设计实现了让用户态线程挂起的同步互斥机制，我们就可以把这种机制也用在内核中，在外设不能及时提供资源的情况下，让想获取资源的线程或进程挂起，直到外设提供了资源，再唤醒线程或进程继续执行。

目前，支持中断的驱动可有效地支持等待的进程唤醒的操作。以串口为例，如果一个进程通过系统调用想获取串口输入，但此时串口还没有输入的字符，那么操作系统就设置一个进程等待串口输入的条件变量（条件变量包含一个等待队列），然后把当前进程设置等待状态，并挂在这个等待队列上，再把 CPU 让给其它就绪进程执行。对于串口输入的处理，由于要考虑中断，相对就要复杂一些。读字符串的代码如下所示：

```

1 //os/src/fs/stdio.rs
2 impl File for Stdin {
3 ...
4 fn read(&self, mut user_buf: UserBuffer) -> usize {
5 assert_eq!(user_buf.len(), 1);
6 //println!("before UART.read() in Stdin::read()");
7 let ch = UART.read();
8 unsafe {
9 user_buf.buffers[0].as_mut_ptr().write_volatile(ch);
10 }
11 1
12 }
13 // os/src/drivers/chardev/ns16550a.rs
14 impl<const BASE_ADDR: usize> CharDevice for NS16550a<BASE_ADDR> {
15 fn read(&self) -> u8 {
16 loop {
17 let mut inner = self.inner.exclusive_access();
18 if let Some(ch) = inner.read_buffer.pop_front() {
19 return ch;
20 } else {
21 let task_cx_ptr = self.condvar.wait_no_sched();
22 drop(inner);
23 schedule(task_cx_ptr);
24 }
25 }
26 }
27 }
```

响应串口输入中断的代码如下所示：

```

1 // os/src/boards/qemu.rs
2 pub fn irq_handler() {
3 let mut plic = unsafe { PLIC::new(VIRT_PLIC) };
4 let intr_src_id = plic.claim(0, IntrTargetPriority::Supervisor);
5 match intr_src_id {
6 ...
7 10 => UART.handle_irq(),
```

(下页继续)

(续上页)

```

8
9 plic.complete(0, IntrTargetPriority::Supervisor, intr_src_id);
10
11 // os/src/drivers/chardev/ns16550a.rs
12 impl<const BASE_ADDR: usize> CharDevice for NS16550a<BASE_ADDR> {
13 fn handle_irq(&self) {
14 let mut count = 0;
15 self.inner.exclusive_session(|inner| {
16 while let Some(ch) = inner.ns16550a.read() {
17 count += 1;
18 inner.read_buffer.push_back(ch);
19 }
20 });
21 if count > 0 {
22 self.condvar.signal();
23 }
24 }

```

对于操作系统的一般处理过程是，首先是能接收中断，即在 `trap_handler` 中通过访问 `scause` 寄存器，能够识别出有外部中断产生。然后再进一步通过读 PLIC 的 `Claim` 寄存器来了解是否是收到了串口发来的输入中断。如果 PLIC 识别出是串口，就会调用串口的中断处理例程。当产生串口有输入并产生中断后，操作系统通过对偏移量为 0 的串口寄存器的进行读操作，从而获得通过串口输入的字符，并存入 `NS16550aInner::read_buffer` 中。然后操作系统将查找等待串口输入的等待队列上的进程，把它唤醒并加入到就绪队列中。这样但这个进程再次执行时，就可以获取到串口数据了。

## 10.5 virtio 设备驱动程序

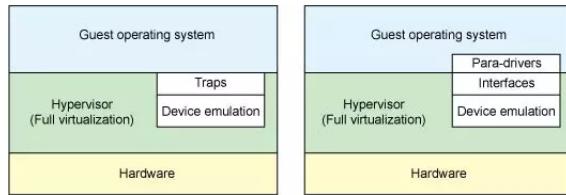
### 10.5.1 本节导读

本节主要介绍了 QEMU 模拟的 RISC-V 计算机中的 virtio 设备的架构和重要组成部分，以及面向 virtio 设备的驱动程序主要功能；并对 virtio-blk 设备及其驱动程序，virtio-gpu 设备及其驱动程序进行了比较深入的分析。这里选择 virtio 设备来进行介绍，主要考虑基于两点考虑，首先这些设备就是 QEMU 模拟的高性能物理外设，操作系统可以面向这些设备编写出合理的驱动程序（如 Linux 等操作系统中都有 virtio 设备的驱动程序，并被广泛应用于云计算虚拟化场景中。）；其次，各种类型的 virtio 设备，如块设备（virtio-blk）、网络设备（virtio-net）、键盘鼠标类设备（virtio-input）、显示设备（virtio-gpu）具有对应外设类型的共性特征、专有特征和与具体处理器无关的设备抽象性。通过对这些设备的分析和比较，能够比较快速地掌握各类设备的核心特点，并掌握编写裸机或操作系统驱动程序的关键技术。

### 10.5.2 virtio 设备

#### virtio 概述

Rusty Russell 在 2008 年左右设计了 virtio 协议，并开发了相应的虚拟化解决方案 lguest，形成了 VirtIO 规范（Virtual I/O Device Specification）。其主要目的是为了简化和统一虚拟机（Hypervisor）的设备模拟，并提高虚拟机环境下的 I/O 性能。virtio 协议是对 hypervisor 中的一组通用模拟设备的抽象，即 virtio 协议定义了虚拟设备的输入/输出接口。而基于 virtio 协议的 I/O 设备称为 virtio 设备。下图列出了两种在虚拟机中模拟外设的总体框架。



在上图左侧的虚拟机模拟外设的传统方案中，如果 guest VM 要使用底层 host 主机的资源，需要 Hypervisor 截获所有的 I/O 请求指令，然后模拟出这些 I/O 指令的行为，这会带来较大的性能开销。

### 注解：虚拟机（Virtual Machine, VM）

虚拟机是物理计算机的虚拟表示形式或仿真环境。虚拟机通常被称为访客机（Guest Machine，简称 Guest）或访客虚拟机（Guest VM），而它们运行所在的物理计算机被称为主机（Host Machine，简称 Host）。

### 虚拟机监视器 Hypervisor

虚拟机监视器（Hypervisor 或 Virtual Machine Monitor，简称 VMM）是创造并且运行虚拟机的软件、固件、或者硬件。这样主机硬件上能同时运行一个至多个虚拟机，这些虚拟机能高效地分享主机硬件资源。

在上图右侧的虚拟机模拟外设的 virtio 方案中，模拟的外设实现了功能最小化，即虚拟外设的数据面接口主要是与 guest VM 共享的内存、控制面接口主要基于内存映射的寄存器和中断机制。这样 guest VM 通过访问虚拟外设来使用底层 host 主机的资源时，Hypervisor 只需对少数寄存器访问和中断机制进行处理，实现了高效的 I/O 虚拟化过程。

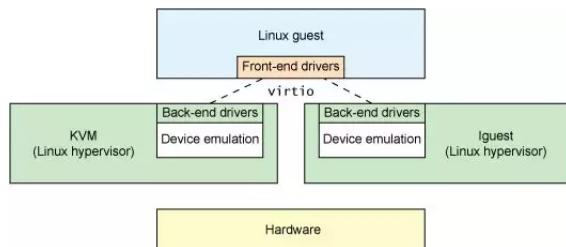
### 注解：数据面（Data Plane）

设备与处理器之间的 I/O 数据传输相关的数据设定（地址、布局等）与传输方式（基于内存或寄存器等）

### 控制平面（Control Plane）

处理器发现设备、配置设备、管理设备等相关的操作，以及处理器和设备之间的相互通知机制。

另外，各种类型的 virtio 设备，如块设备（virtio-blk）、网络设备（virtio-net）、键盘鼠标类设备（virtio-input）、显示设备（virtio-gpu）具有共性特征和独有特征。对于共性特征，virtio 设计了各种类型设备的统一抽象接口，而对于独有特征，virtio 尽量最小化各种类型设备的独有抽象接口。这样，virtio 就形成了一套通用框架和标准接口（协议）来屏蔽各种 hypervisor 的差异性，实现了 guest VM 和不同 hypervisor 之间的交互过程。



上图意味着什么呢？它意味着在 guest VM 上看到的虚拟设备具有简洁通用的优势，这对运行在 guest VM 上的操作系统而言，可以设计出轻量高效的设备驱动程序（即上图的 Front-end drivers）。

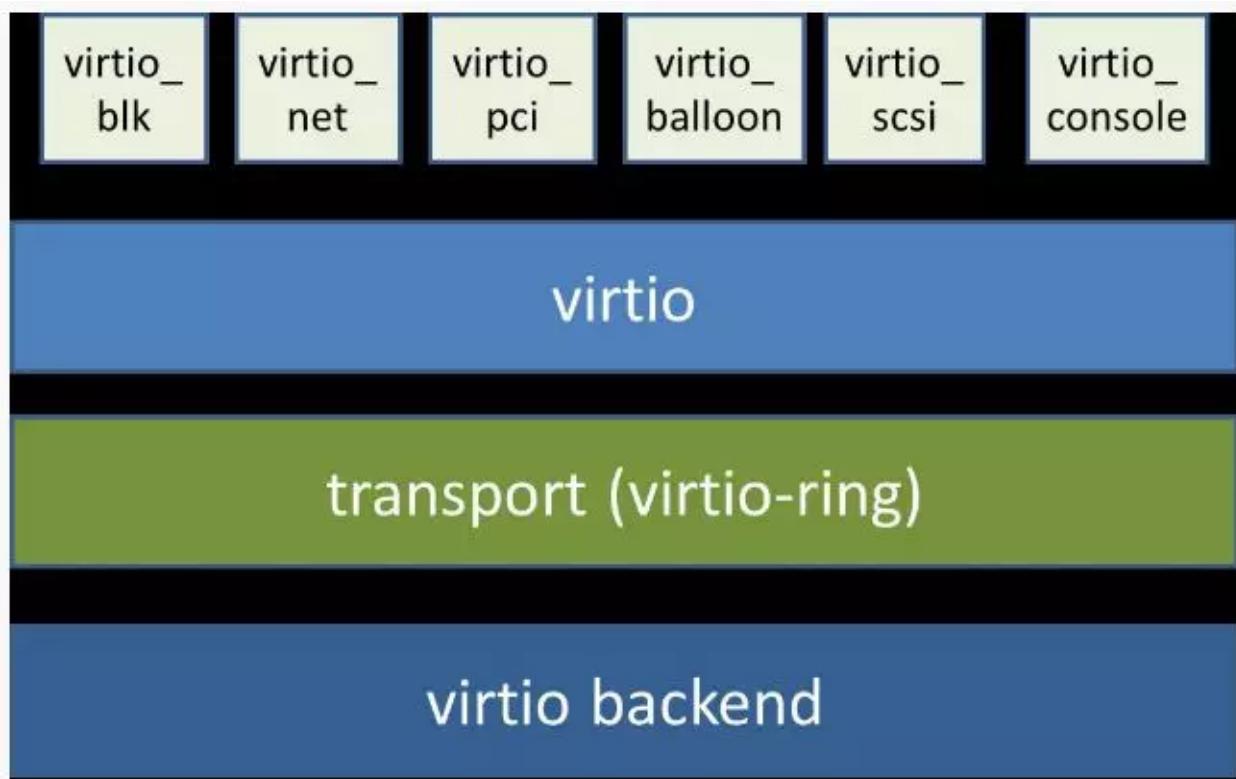
从本质上讲，virtio 是一个接口，允许运行在虚拟机上的操作系统和应用软件通过访问 virtio 设备使用其主机的设备。这些 virtio 设备具备功能最小化的特征，Guest VM 中的设备驱动程序（Front-end drivers）只需实现基本的发送和接收 I/O 数据即可，而位于 Hypervisor 中的 Back-end drivers 和设备模拟部分让主机处理其实际物理硬件设备上的大部分设置、维护和处理。这种设计方案极大减轻了 virtio 驱动程序的复杂性。

virtio 设备是虚拟外设，存在于 QEMU 模拟的 RISC-V 64 virt 计算机中。而我们要在操作系统中实现 virtio 驱动程序，来管理和控制这些 virtio 虚拟设备。每一类 virtio 设备都有自己的 virtio 接口，virtio 接口包括了数据

结构和相关 API 的定义。这些定义中，有共性内容，也有属于设备特定类型的非共性内容。

### virtio 架构

总体上看，virtio 架构可以分为上中下三层，上层包括运行在 QEMU 模拟器上的前端操作系统中各种驱动程序（Front-end drivers）；下层是在 QEMU 中模拟的各种虚拟设备 Device；中间层是传输（transport）层，就是驱动程序与虚拟设备之间的交互接口，包含两部分：上半部是 virtio 接口定义，即 I/O 数据传输机制的定义：virtio 虚拟队列（virtqueue）；下半部是 virtio 接口实现，即 I/O 数据传输机制的具体实现：virtio-ring，主要由环形缓冲区和相关操作组成，用于保存驱动程序和虚拟设备之间进行命令和数据交互的信息。



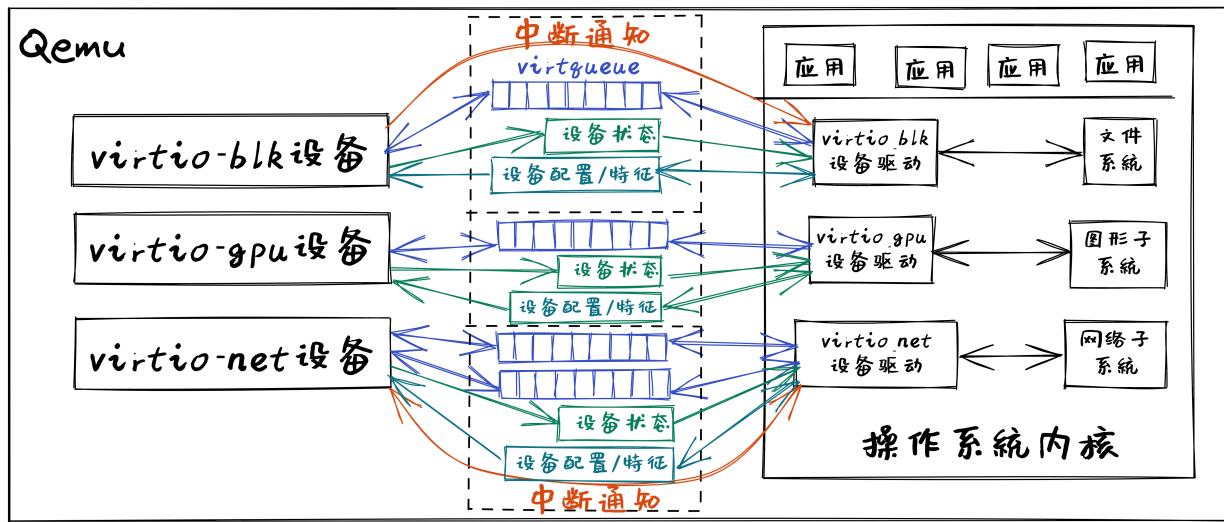
操作系统中 virtio 驱动程序的主要功能包括：

- 接受来自用户进程或操作系统其它组件发出的 I/O 请求
- 将这些 I/O 请求通过 virqueue 发送到相应的 virtio 设备
- 通过中断或轮询等方式查找并处理相应设备完成的 I/O 请求

Qemu 或 Hypervisor 中 virtio 设备的主要功能包括：

- 通过 virqueue 接受来自相应 virtio 驱动程序的 I/O 请求
- 通过设备仿真模拟或将 I/O 操作卸载到主机的物理硬件来处理 I/O 请求，使处理后的 I/O 数据可供 virtio 驱动程序使用
- 通过寄存器、内存映射或中断等方式通知 virtio 驱动程序处理已完成的 I/O 请求

运行在 Qemu 中的操作系统中的 virtio 驱动程序和 Qemu 模拟的 virtio 设备驱动的关系如下图所示：



## I/O 设备基本组成结构

virtio 设备代表了一类 I/O 通用设备，为了让设备驱动能够管理和使用设备。在程序员的眼里，I/O 设备基本组成结构包括如下恩利：

- 呈现模式：设备一般通过寄存器、内存或特定 I/O 指令等方式让设备驱动能看到和访问到设备；
- 特征描述：让设备驱动能够了解设备的静态特性（可通过软件修改），从而决定是否或如何使用该设备；
- 状态表示：让设备驱动能够了解设备的当前动态状态，从而确定如何进行设备管理或 I/O 数据传输；
- 交互机制：交互包括事件通知和数据传输；对于事件通知，让设备驱动及时获知设备的状态变化的机制（可基于中断等机制），以及让设备及时获得设备驱动发出的 I/O 请求（可基于寄存器读写等机制）；对于数据传输，让设备驱动能处理设备给出的数据，以及让设备能处理设备驱动给出的数据，如（可基于 DMA 或 virtqueue 等机制）。

virtio 设备具体定义了设备驱动和设备之间的接口，包括设备呈现模式、设备状态域、特征位、通知、设备配置空间、虚拟队列等，覆盖了上述的基本接口描述。

## virtio 设备基本组成要素

virtio 设备的基本组成要素如下：

- 设备状态域（Device status field）
- 特征位（Feature bits）
- 通知（Notifications）
- 设备配置空间（Device Configuration space）
- 一个或多个虚拟队列（virtqueue）

其中的设备特征位和设备配置空间属于 virtio 设备的特征描述；设备状态域属于 virtio 设备初始化时的状态表示；通知和虚拟队列属于 virtio 设备的交互机制，也包含 virtio 设备运行时的状态表示。

## **virtio 设备呈现模式**

virtio 设备支持三种设备呈现模式：

- Virtio Over MMIO，虚拟设备直接挂载到系统总线上，我们实验中的虚拟计算机就是这种呈现模式；
- Virtio Over PCI BUS，遵循 PCI 规范，挂在到 PCI 总线上，作为 virtio-pci 设备呈现，在 QEMU 虚拟的 x86 计算机上采用的是这种模式；
- Virtio Over Channel I/O：主要用在虚拟 IBM s390 计算机上，virtio-ccw 使用这种基于 channel I/O 的机制。

在 Qemu 模拟的 RISC-V 计算机–virt 上，采用的是 Virtio Over MMIO 的呈现模式。这样在实现设备驱动时，我们只需要找到相应 virtio 设备的 I/O 寄存器等以内存形式呈现的地址空间，就可以对 I/O 设备进行初始化和管理了。

## **virtio 设备特征描述**

virtio 设备特征描述包括设备特征位和设备配置空间。

### **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。

### **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的 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。

## **virtio 设备状态表示**

virtio 设备状态表示包括在设备初始化过程中用到的设备状态域，以及在设备进行 I/O 传输过程中用到的 I/O 数据访问状态信息和 I/O 完成情况等。

### **设备状态域**

设备状态域包含对设备初始化过程中 virtio 设备的 6 种状态：

- **ACKNOWLEDGE** (1)：驱动程序发现了这个设备，并且认为这是一个有效的 virtio 设备；
- **DRIVER** (2)：驱动程序知道该如何驱动这个设备；
- **FAILED** (128)：由于某种错误原因，驱动程序无法正常驱动这个设备；
- **FEATURES\_OK** (8)：驱动程序认识设备的特征，并且与设备就设备特征协商达成一致；
- **DRIVER\_OK** (4)：驱动程序加载完成，设备可以正常工作了；
- **DEVICE\_NEEDS\_RESET** (64)：设备触发了错误，需要重置才能继续工作。

在设备驱动程序对 virtio 设备初始化的过程中，需要经历一系列的初始化阶段，这些阶段对应着设备状态域的不同状态。

### **I/O 传输状态**

设备驱动程序控制 virtio 设备进行 I/O 传输过程中，会经历一系列过程和执行状态，包括 *I/O* 请求状态、*I/O* 处理状态、*I/O* 完成状态、*I/O* 错误状态、*I/O* 后续处理状态等。设备驱动程序在执行过程中，需要对上述状态进行不同的处理。

virtio 设备进行 I/O 传输过程中，设备驱动会指出 *I/O* 请求队列的当前位置状态信息，这样设备能查到 I/O 请求的信息，并根据 *I/O* 请求进行 I/O 传输；而设备会指出 *I/O* 完成队列的当前位置状态信息，这样设备驱动通过读取 *I/O* 完成数据结构中的状态信息，就知道设备是否完成 I/O 请求的相应操作，并进行后续事务处理。

比如，virtio\_blk 设备驱动发出一个读设备块的 I/O 请求，并在某确定位置给出这个 I/O 请求的地址，然后给设备发出‘kick’通知（读或写相关 I/O 寄存器映射的内存地址），此时处于 I/O 请求状态；设备在得到通知后，此时处于 I/O 处理状态，它解析这个 I/O 请求，完成这个 I/O 请求的处理，即把磁盘块内容读入到内存中，并给出读出的块数据的内存地址，再通过中断通知设备驱动，此时处于 I/O 完成状态；如果磁盘块读取发生错误，此时处于 I/O 错误状态；设备驱动通过中断处理例程，此时处于 I/O 后续处理状态，设备驱动知道设备已经完成读磁盘块操作，会根据磁盘块数据所在内存地址，把数据传递给文件系统进行进一步处理；如果设备驱动发现磁盘块读错误，则会进行错误恢复相关的后续处理。

## virtio 设备交互机制

virtio 设备交互机制包括基于 Notifications 的事件通知和基于 virtqueue 虚拟队列的数据传输。事件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。数据传输是指设备和驱动程序之间进行 I/O 数据（如磁盘块数据、网络包）传输。

### Notification 通知

驱动程序和设备在交互过程中需要相互通知对方：驱动程序组织好相关命令/信息要通知设备去处理 I/O 事务，设备处理完 I/O 事务后，要通知驱动程序进行后续事务，如回收内存，向用户进程反馈 I/O 事务的处理结果等。

驱动程序通知设备可用“门铃 doorbell”机制，即采用 PIO 或 MMIO 方式访问设备特定寄存器，QEMU 进行拦截再通知其模拟的设备。设备通知驱动程序一般用中断机制，即在 QEMU 中进行中断注入，让 CPU 响应并执行中断处理例程，来完成对 I/O 执行结果的处理。

### 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），virtio 设备的虚拟队列（virtqueue）可以由各种数据结构（如数组、环形队列等）来具体实现。每个 virtio 设备可以拥有零个或多个 virtqueue，每个 virtqueue 占用多个物理页，可用于设备驱动程序给设备发 I/O 请求命令和相关数据（如磁盘块读写请求和读写缓冲区），也可用于设备给设备驱动程序发 I/O 数据（如接收的网络包）。

## virtqueue 虚拟队列

virtio 协议中一个关键部分是 virtqueue，在 virtio 规范中，virtqueue 是 virtio 设备上进行批量数据传输的机制和抽象表示。在设备驱动实现和 Qemu 中 virtio 设备的模拟实现中，virtqueue 是一种数据结构，用于设备和驱动程序中执行各种数据传输操作。

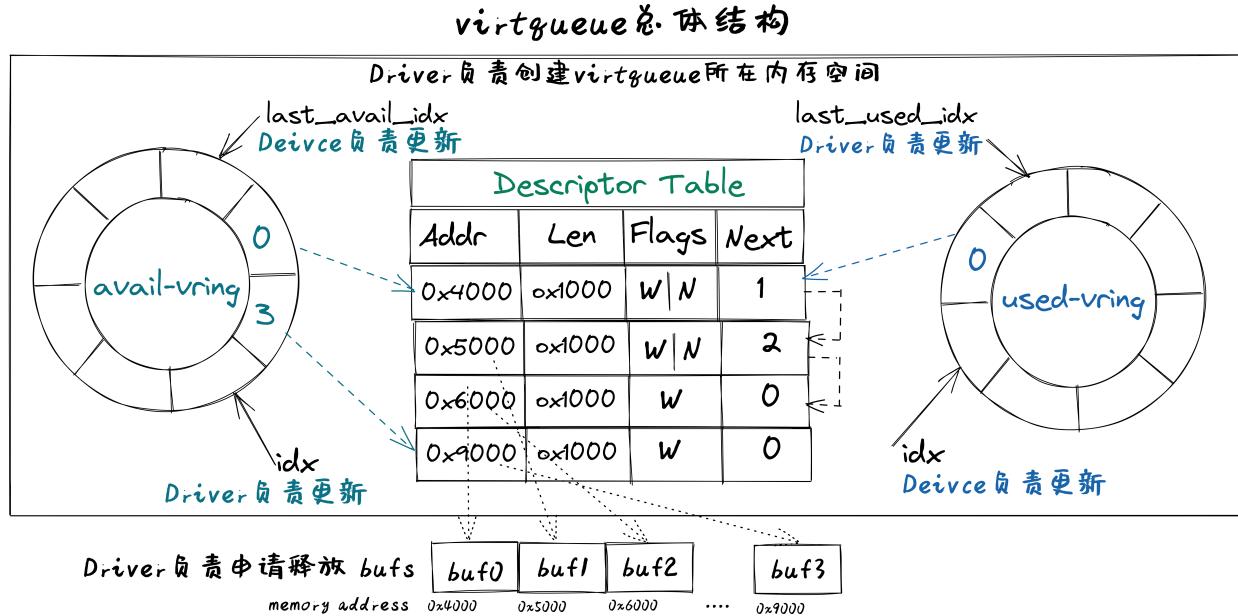
操作系统在 Qemu 上运行时，virtqueue 是 virtio 驱动程序和 virtio 设备访问的同一块内存区域。

当涉及到 virtqueue 的描述时，有很多不一致的地方。有将其与 vring（virtio-rings 或 VRings）等同表示，也有将二者分别单独描述为不同的对象。我们将在这里单独描述它们，因为 vring 是 virtqueues 的主要组成部分，是达成 virtio 设备和驱动程序之间数据传输的数据结构，vring 本质是 virtio 设备和驱动程序之间的共享内存，但 virtqueue 不仅仅只有 vring。

virtqueue 由三部分组成（如下图所示）：

- 描述符表 Descriptor Table：描述符表是描述符为组成元素的数组，每个描述符描述了一个内存 buffer 的 address/length。而内存 buffer 中包含 I/O 请求的命令/数据（由 virtio 设备驱动填写），也可包含 I/O 完成的返回结果（由 virtio 设备填写）等。
- 可用环 Available Ring：一种 vring，记录了 virtio 设备驱动程序发出的 I/O 请求索引，即被 virtio 设备驱动程序更新的描述符索引的集合，需要 virtio 设备进行读取并完成相关 I/O 操作；

- 已用环 Used Ring: 另一种 vring, 记录了 virtio 设备发出的 I/O 完成索引, 即被 virtio 设备更新的描述符索引的集合, 需要 virtio 设备驱动程序进行读取并对 I/O 操作结果进行进一步处理。



### 描述符表 Descriptor Table

描述符表用来指向 virtio 设备 I/O 传输请求的缓冲区 (buffer) 信息, 由 Queue Size 个 Descriptor (描述符) 组成。描述符中包括 buffer 的物理地址-addr 字段, buffer 的长度-len 字段, 可以链接到 next Descriptor 的 next 指针 (用于把多个描述符链接成描述符链)。buffer 所在物理地址空间需要设备驱动程序在初始化时分配好, 并在后续由设备驱动程序在其中填写 I/O 传输相关的命令/数据, 或者是设备返回 I/O 操作的结果。多个描述符 (I/O 操作命令, I/O 操作数据块, I/O 操作的返回结果) 形成的描述符链可以表示一个完整的 I/O 操作请求。

### 可用环 Available Ring

可用环在结构上是一个环形队列, 其中的条目 (item) 仅由驱动程序写入, 并由设备读出。可用环中的条目包含了一个描述符链的头部描述符的索引值。可用环用头指针 (idx) 和尾指针 (last\_avail\_idx) 表示其可用条目范围。virtio 设备通过读取可用环中的条目可获取驱动程序发出的 I/O 操作请求对应的描述符链, 然后 virtio 设备就可以进行进一步的 I/O 处理了。描述符指向的缓冲区具有可读写属性, 可读的缓冲区用于 Driver 发送数据, 可写的缓冲区用于接收数据。

比如, 对于 virtio-blk 设备驱动发出的一个读 I/O 操作请求包含了三部分内容, 由三个 buffer 承载, 需要用到三个描述符: (1) “读磁盘块”, (2) I/O 操作数据块-“数据缓冲区”, (3) I/O 操作的返回结果-“结果缓冲区”。这三个描述符形成的一个完成的 I/O 请求链, virtio-blk 从设备可通过读取第一个描述符指向的缓冲区了解到是“读磁盘块”操作, 这样就可把磁盘块数据通过 DMA 操作放到第二个描述符指向的“数据缓冲区”中, 然后把“OK”写入到第三个描述符指向的“结果缓冲区”中。

### 已用环 Used Ring

已用环在结构上是一个环形队列, 其中的条目仅由 virtio 设备写入, 并由驱动程序读出。已用环中的条目也是一个描述符链的头部描述符的索引值。已用环也有头指针 (idx) 和尾指针 (last\_avail\_idx) 表示其已用条目的范围。

比如, 对于 virtio-blk 设备驱动发出的一个读 I/O 操作请求 (由三个描述符形成的请求链) 后, virtio 设备完成相应 I/O 处理, 即把磁盘块数据写入第二个描述符指向的“数据缓冲区”中, 可用环中对应的 I/O 请求条目“I/O 操作的返回结果”的描述符索引值移入到已用环中, 把“OK”写入到第三个描述符指向的“结果缓冲区”中, 再在已用环中添加一个已用条目, 即 I/O 操作完成信息; 然后 virtio 设备通过中断机制来通知 virtio 驱动程序, 并让 virtio 驱动程序读取已用环中的描述符, 获得 I/O 操作完成信息, 即磁盘块内容。

上面主要说明了 virqueue 中的各个部分的作用。对如何基于 virqueue 进行 I/O 操作的过程还缺乏一个比较完整的描述。我们把上述基于 virqueue 进行 I/O 操作的过程小结一下，大致需要如下步骤：

### 1. 初始化过程：（驱动程序执行）

- 1.1 virtio 设备驱动在对设备进行初始化时，会申请 virqueue（包括描述符表、可用环、已用环）的内存空间；
- 1.2 并把 virqueue 中的描述符、可用环、已用环三部分的物理地址分别写入到 virtio 设备中对应的控制寄存器（即设备绑定的特定内存地址）中。至此，设备驱动和设备就共享了整个 virqueue 的内存空间。

### 2. I/O 请求过程：（驱动程序执行）

- 2.1 设备驱动在发出 I/O 请求时，首先把 I/O 请求的命令/数据等放到一个或多个 buffer 中；
- 2.2 然后在描述符表中分配新的描述符（或描述符链）来指向这些 buffer；
- 2.3 再把描述符（或描述符链的首描述符）的索引值写入到可用环中，更新可用环的 idx 指针；
- 2.4 驱动程序通过 kick 机制（即写 virtio 设备中特定的通知控制寄存器）来通知设备有新请求；

### 3. I/O 完成过程：（设备执行）

- 3.1 virtio 设备通过 kick 机制（知道有新的 I/O 请求，通过访问可用环的 idx 指针，解析出 I/O 请求；
- 3.2 根据 I/O 请求内容完成 I/O 请求，并把 I/O 操作的结果放到 I/O 请求中相应的 buffer 中；
- 3.3 再把描述符（或描述符链的首描述符）的索引值写入到已用环中，更新已用环的 idx 指针；
- 3.4 设备通过再通过中断机制来通知设备驱动程序有 I/O 操作完成；

### 4. I/O 后处理过程：（驱动程序执行）

- 4.1 设备驱动程序读取已用环的 idx 信息，读取已用环中的描述符索引，获得 I/O 操作完成信息。

## 基于 MMIO 方式的 virtio 设备

基于 MMIO 方式的 virtio 设备没有基于总线的设备探测机制。所以操作系统采用 Device Tree 的方式来探测各种基于 MMIO 方式的 virtio 设备，从而操作系统能知道与设备相关的寄存器和所用的中断。基于 MMIO 方式的 virtio 设备提供了一组内存映射的控制寄存器，后跟一个设备特定的配置空间，在形式上是位于一个特定地址上的内存区域。一旦操作系统找到了这个内存区域，就可以获得与这个设备相关的各种寄存器信息。比如，我们在 `virtio-drivers` crate 中就定义了基于 MMIO 方式的 virtio 设备的寄存器区域：

```

1 //virtio-drivers/src/header.rs
2 pub struct VirtIOHeader {
3 magic: ReadOnly<u32>, //魔数 Magic value
4 ...
5 //设备初始化相关的特征/状态/配置空间对应的寄存器
6 device_features: ReadOnly<u32>, //设备支持的功能
7 device_features_sel: WriteOnly<u32>, //设备选择的功能
8 driver_features: WriteOnly<u32>, //驱动程序理解的设备功能
9 driver_features_sel: WriteOnly<u32>, //驱动程序选择的设备功能
10 config_generation: ReadOnly<u32>, //配置空间
11 status: Volatile<DeviceStatus>, //设备状态
12
13 //virtqueue虚拟队列对应的寄存器
14 queue_sel: WriteOnly<u32>, //虚拟队列索引号
15 queue_num_max: ReadOnly<u32>, //虚拟队列最大容量值
16 queue_num: WriteOnly<u32>, //虚拟队列当前容量值
17 queue_notify: WriteOnly<u32>, //虚拟队列通知
18 queue_desc_low: WriteOnly<u32>, //设备描述符表的低32位地址
19 queue_desc_high: WriteOnly<u32>, //设备描述符表的高32位地址
20 queue_avail_low: WriteOnly<u32>, //可用环的低32位地址

```

(下页继续)

(续上页)

```

21 queue_avail_high: WriteOnly<u32>, //可用环的高32位地址
22 queue_used_low: WriteOnly<u32>, //已用环的低32位地址
23 queue_used_high: WriteOnly<u32>, //已用环的高32位地址
24
25 //中断相关的寄存器
26 interrupt_status: ReadOnly<u32>, //中断状态
27 interrupt_ack: WriteOnly<u32>, //中断确认
28 }
```

这里列出了部分关键寄存器和它的基本功能描述。在后续的设备初始化以及设备 I/O 操作中，会访问这里列出的寄存器。

在有了上述 virtio 设备的理解后，接下来，我们将进一步分析 virtio 驱动程序如何管理 virtio 设备来完成初始化和 I/O 操作。

### 10.5.3 virtio 驱动程序

这部分内容是各种 virtio 驱动程序的共性部分，主要包括初始化设备，驱动程序与设备的交互步骤，以及驱动程序执行过程中的一些实现细节。

#### 设备的初始化

操作系统通过某种方式（设备发现，基于设备树的查找等）找到 virtio 设备后，驱动程序进行设备初始化的常规步骤如下所示：

1. 重启设备状态，设置设备状态域为 0
2. 设置设备状态域为 ACKNOWLEDGE，表明当前已经识别到了设备
3. 设置设备状态域为 DRIVER，表明驱动程序知道如何驱动当前设备
4. 进行设备特定的安装和配置，包括协商特征位，建立 virtqueue，访问设备配置空间等，设置设备状态域为 FEATURES\_OK
5. 设置设备状态域为 DRIVER\_OK 或者 FAILED（如果中途出现错误）

注意，上述的步骤不是必须都要做到的，但最终需要设置设备状态域为 DRIVER\_OK，这样驱动程序才能正常访问设备。

在 *virtio\_driver* 模块中，我们实现了通用的 virtio 驱动程序框架，各种 virtio 设备驱动程序的共同的初始化过程为：

1. 确定协商特征位，调用 *VirtIOHeader* 的 *begin\_init* 方法进行 virtio 设备初始化的第 1-4 步骤；
2. 读取配置空间，确定设备的配置情况；
3. 建立虚拟队列 1~n 个 virtqueue；
4. 调用 *VirtIOHeader* *finish\_init* 方法进行 virtio 设备初始化的第 5 步骤。

比如，对于 *virtio\_blk* 设备初始化的过程如下所示：

```

1 // virtio_drivers/src/blk.rs
2 //virtio_blk驱动初始化：调用 header.begin_init方法
3 impl<H: Hal> VirtIOBlk<'_, H> {
4 /// Create a new VirtIO-Blk driver.
5 pub fn new(header: &'static mut VirtIOHeader) -> Result<Self> {
6 header.begin_init(|features| {
```

(下页继续)

(续上页)

```

7 ...
8 (features & supported_features).bits()
9 };
10 // 读取 virtio_blk 设备的配置空间
11 let config = unsafe { &mut * (header.config_space() ...) };
12 // 建立 1 个虚拟队列
13 let queue = VirtQueue::new(header, 0, 16)?;
14 // 结束设备初始化
15 header.finish_init();
16 ...
17 }
18 // virtio_drivers/src/header.rs
19 // virtio 设备初始化的第 1~4 步骤
20 impl VirtIOHeader {
21 pub fn begin_init(&mut self, negotiate_features: impl FnOnce(u64) -> u64) {
22 self.status.write(DeviceStatus::ACKNOWLEDGE);
23 self.status.write(DeviceStatus::DRIVER);
24 let features = self.read_device_features();
25 self.write_driver_features(negotiate_features(features));
26 self.status.write(DeviceStatus::FEATURES_OK);
27 self.guest_page_size.write(PAGE_SIZE as u32);
28 }
29
30 // virtio 设备初始化的第 5 步骤
31 pub fn finish_init(&mut self) {
32 self.status.write(DeviceStatus::DRIVER_OK);
33 }

```

## 驱动程序与设备之间的交互

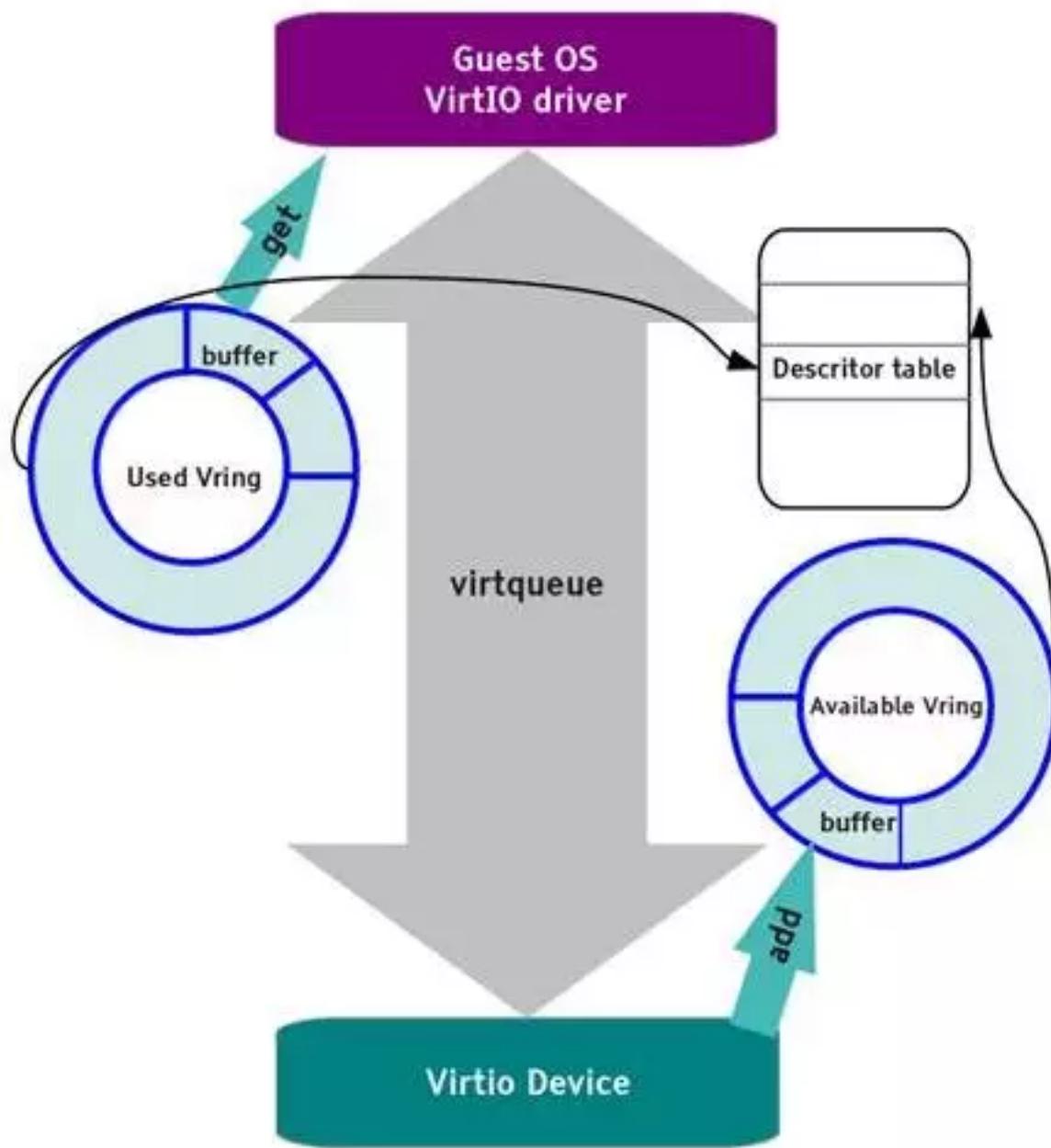
驱动程序与外设可以共同访问约定的 virtqueue，virtqueue 将保存设备驱动的 I/O 请求信息和设备的 I/O 响应信息。virtqueue 由描述符表（Descriptor Table）、可用环（Available Ring）和已用环（Used Ring）组成。在上述的设备驱动初始化过程中已经看到了虚拟队列的创建过程。

当驱动程序向设备发送 I/O 请求（由命令/数据组成）时，它会在 buffer（设备驱动申请的内存空间）中填充命令/数据，各个 buffer 所在的起始地址和大小信息放在描述符表的描述符中，再把这些描述符链接在一起，形成描述符链。

而描述符链的起始描述符的索引信息会放入一个称为环形队列的数据结构中。该队列有两类，一类是包含由设备驱动发出的 I/O 请求所对应的描述符索引信息，即可用环。另一类由包含由设备发出的 I/O 响应所对应的描述符索引信息，即已用环。

一个用户进程发起的 I/O 操作的处理过程大致可以分成如下四步：

1. 用户进程发出 I/O 请求，经过层层下传给到驱动程序，驱动程序将 I/O 请求信息放入虚拟队列 virtqueue 的可用环中，并通过某种通知机制（如写某个设备寄存器）通知设备；
2. 设备收到通知后，解析可用环和描述符表，取出 I/O 请求并在内部进行实际 I/O 处理；
3. 设备完成 I/O 处理或出错后，将结果作为 I/O 响应放入已用环中，并以某种通知机制（如外部中断）通知 CPU；
4. 驱动程序解析已用环，获得 I/O 响应的结果，在进一步处理后，最终返回给用户进程。



### 发出 I/O 请求的过程

虚拟队列的相关操作包括两个部分：向设备提供新的 I/O 请求信息（可用环→描述符→缓冲区），以及处理设备使用的 I/O 响应（已用环→描述符→缓冲区）。比如，virtio-blk 块设备具有一个虚拟队列来支持 I/O 请求和 I/O 响应。在驱动程序进行 I/O 请求和 I/O 响应的具体操作过程中，需要注意如下一些细节。

驱动程序给设备发出 I/O 请求信息的具体步骤如下所示：

1. 将包含一个 I/O 请求内容的缓冲区的地址和长度信息放入描述符表中的空闲描述符中，并根据需要把多个描述符进行链接，形成一个描述符链（表示一个 I/O 操作请求）；
2. 驱动程序将描述符链头的索引放入可用环的下一个环条目中；

3. 如果可以进行批处理 (batching)，则可以重复执行步骤 1 和 2，这样通过 (可用环-> 描述符-> 缓冲区) 来添加多个 I/O 请求；
4. 根据添加到可用环中的描述符链头的数量，更新可用环；
5. 将“有可用的缓冲区”的通知发送给设备。

注：在第 3 和第 4 步中，都需要指向适当的内存屏障操作 (Memory Barrier)，以确保设备能看到更新的描述符表和可用环。

---

#### 注解：内存屏障 (Memory Barrier)

大多数现代计算机为了提高性能而采取乱序执行，这使得内存屏障在某些情况下成为必须要执行的操作。内存屏障是一类同步屏障指令，它使得 CPU 或编译器在对内存进行操作的时候，严格按照一定的顺序来执行，也就是说在内存屏障之前的指令和内存屏障之后的指令不会由于系统优化等原因而导致乱序。内存屏障分为写屏障 (Store Barrier)、读屏障 (Load Barrier) 和全屏障 (Full Barrier)，其作用是：

- 防止指令之间的重排序
  - 保证数据的可见性
- 

#### 将缓冲区信息放入描述符表的操作

缓冲区用于表示一个 I/O 请求的具体内容，由零个或多个设备可读/可写的物理地址连续的内存块组成（一般前面是可读的内存块，后续跟着可写的内存块）。我们把构成缓冲区的内存块称为缓冲区元素，把缓冲区映射到描述符表中以形成描述符链的具体步骤：

对于每个缓冲区元素 `b` 执行如下操作：

1. 获取下一个空闲描述符表条目 `d`；
2. 将 `d.addr` 设置为 `b` 的起始物理地址；
3. 将 `d.len` 设置为 `b` 的长度；
4. 如果 `b` 是设备可写的，则将 `d.flags` 设置为 `VIRTQ_DESC_F_WRITE`，否则设置为 0；
5. 如果 `b` 之后还有一个缓冲元素 `c`：
  - 5.1 将 `d.next` 设置为下一个空闲描述符元素的索引；
  - 5.2 将 `d.flags` 中的 `VIRTQ_DESC_F_NEXT` 位置 1；

#### 更新可用环的操作

描述符链头是上述步骤中的第一个条目 `d`，即描述符表条目的索引，指向缓冲区的第一部分。一个驱动程序实现可以执行以下的伪码操作（假定在与小端字节序之间进行适当的转换）来更新可用环：

```
avail.ring[avail.idx % qsz] = head; //qsz表示可用环的大小
```

但是，通常驱动程序可以在更新 `idx` 之前添加许多描述符链（这时它们对于设备是可见的），因此通常要对驱动程序已添加的数目 `added` 进行计数：

```
avail.ring[(avail.idx + added++) % qsz] = head;
```

`idx` 总是递增。由于上一行的取模操作，我们不必担心 `idx > qsz` 时会产生溢出。

```
avail.idx += added;
```

一旦驱动程序更新了可用环的 `idx` 指针，这表示描述符及其它指向的缓冲区能够被设备看到。这样设备就可以访问驱动程序创建的描述符链和它们指向的内存。驱动程序必须在 `idx` 更新之前执行合适的内存屏障操作，以确保设备看到最新描述符和 `buffer` 内容。

## 通知设备的操作

在包含 virtio 设备的 Qemu virt 虚拟计算机中，驱动程序一般通过对代表通知“门铃”的特定寄存器进行写操作来发出通知。

```

1 // virtio_drivers/src/header.rs
2 pub struct VirtIOHeader {
3 // Queue notifier 用户虚拟队列通知的寄存器
4 queue_notify: WriteOnly<u32>,
5 ...
6 impl VirtIOHeader {
7 // Notify device.
8 pub fn notify(&mut self, queue: u32) {
9 self.queue_notify.write(queue);
10 }

```

## 接收设备 I/O 响应的操作

一旦设备完成了 I/O 请求，形成 I/O 响应，就会更新描述符所指向的缓冲区，并向驱动程序发送已用缓冲区通知（used buffer notification）。一般会采用中断这种更加高效的通知机制。设备驱动程序在收到中断后，就会对 I/O 响应信息进行后续处理。相关的伪代码如下所示：

```

1 // virtio_drivers/src/blk.rs
2 impl<H: Hal> VirtIOBlk<'_, H> {
3 pub fn ack_interrupt(&mut self) -> bool {
4 self.header.ack_interrupt()
5 }
6
7 // virtio_drivers/src/header.rs
8 pub struct VirtIOHeader {
9 // 中断状态寄存器 Interrupt status
10 interrupt_status: ReadOnly<u32>,
11 // 中断响应寄存器 Interrupt acknowledge
12 interrupt_ack: WriteOnly<u32>,
13 impl VirtIOHeader {
14 pub fn ack_interrupt(&mut self) -> bool {
15 let interrupt = self.interrupt_status.read();
16 if interrupt != 0 {
17 self.interrupt_ack.write(interrupt);
18 true
19 }
20 ...

```

这里给出了 virtio 设备驱动通过中断来接收设备 I/O 响应的共性操作过程。如果结合具体的操作系统，还需与操作系统的总体中断处理、同步互斥、进程/线程调度进行结合。

## 10.6 virtio\_blk 块设备驱动程序

### 10.6.1 本节导读

本节主要介绍了与操作系统无关的基本 virtio\_blk 设备驱动程序的设计与实现，以及如何在操作系统中封装 virtio\_blk 设备驱动程序，实现基于中断机制的 I/O 操作，提升计算机系统的整体性能。

### 10.6.2 virtio-blk 驱动程序

virtio-blk 设备是一种 virtio 存储设备，在 QEMU 模拟的 RISC-V 64 计算机中，以 MMIO 和中断等方式方式来与驱动程序进行交互。这里我们以 Rust 语言为例，给出 virtio-blk 设备驱动程序的设计与实现。主要包括如下内容：

- virtio-blk 设备的关键数据结构
- 初始化 virtio-blk 设备
- 操作系统对接 virtio-blk 设备初始化
- virtio-blk 设备的 I/O 操作
- 操作系统对接 virtio-blk 设备 I/O 操作

### 10.6.3 virtio-blk 设备的关键数据结构

这里我们首先需要定义 virtio-blk 设备的结构：

```
1 // virtio-drivers/src/blk.rs
2 pub struct VirtIOBlk<'a, H: Hal> {
3 header: &'static mut VirtIOHeader,
4 queue: VirtQueue<'a, H>,
5 capacity: usize,
6 }
```

header 成员对应的 VirtIOHeader 数据结构是 virtio 设备的共有属性，包括版本号、设备 id、设备特征等信息，其内存布局和成员变量的含义与上一节描述 [virt-mmio](#) 设备的寄存器内存布局是一致的。而 VirtQueue 数据结构与上一节描述的 [virtqueue](#) 在表达的含义上基本一致的。

```
1 #[repr(C)]
2 pub struct VirtQueue<'a, H: Hal> {
3 dma: DMA<H>, // DMA guard
4 desc: &'a mut [Descriptor], // 描述符表
5 avail: &'a mut AvailRing, // 可用环 Available ring
6 used: &'a mut UsedRing, // 已用环 Used ring
7 queue_idx: u32, // 虚拟队列索引值
8 queue_size: u16, // 虚拟队列长度
9 num_used: u16, // 已经使用的队列项目数
10 free_head: u16, // 空闲队列项目头的索引值
11 avail_idx: u16, // 可用环的索引值
12 last_used_idx: u16, // 上次已用环的索引值
13 }
```

其中成员变量 free\_head 指空闲描述符链表头，初始时所有描述符通过 next 指针依次相连形成空闲链表，成员变量 last\_used\_idx 是指设备上次已取的已用环元素位置。成员变量 avail\_idx 是可用环的索引值。

这里出现的 Hal trait 是 *virtio\_drivers* 库中定义的一个 trait，用于抽象出与具体操作系统相关的操作，主要与内存分配和虚实地址转换相关。这里我们只给出 trait 的定义，对应操作系统的具体实现会在后续的章节中会给出。

```

1 pub trait Hal {
2 /// Allocates the given number of contiguous physical pages of DMA memory for
3 /// virtio use.
4 fn dma_alloc(pages: usize) -> PhysAddr;
5 /// Deallocates the given contiguous physical DMA memory pages.
6 fn dma_dealloc(paddr: PhysAddr, pages: usize) -> i32;
7 /// Converts a physical address used for virtio to a virtual address which the
8 /// program can
9 /// access.
10 fn phys_to_virt(paddr: PhysAddr) -> VirtAddr;
11 /// Converts a virtual address which the program can access to the corresponding
12 /// physical
13 /// address to use for virtio.
14 fn virt_to_phys(vaddr: VirtAddr) -> PhysAddr;
15 }
```

## 10.6.4 初始化 virtio-blk 设备

virtio-blk 设备的初始化过程与 virtio 规范中描述的一般 virtio 设备的初始化过程大致一样，对其实现的初步分析在 *virtio-blk* 初始化代码中。在设备初始化过程中读取了 virtio-blk 设备的配置空间的设备信息：

```

1 capacity: Volatile<u64> = 32 //32个扇区，即 16KB
2 blk_size: Volatile<u32> = 512 //扇区大小为 512字节
```

为何能看到扇区大小为 512 字节<sup>11</sup>，容量为 16KB 大小的 virtio-blk 设备？这当然是我们让 Qemu 模拟器建立的一个虚拟硬盘。下面的命令可以看到虚拟硬盘创建和识别过程：

```

1 # 在 virtio-drivers 仓库的 example/riscv 目录下执行如下命令
2 make run
3 # 可以看到与虚拟硬盘创建相关的具体命令
4 ## 通过 dd 工具创建了扇区大小为 ``512`` 字节11，容量为 ``16KB`` 大小的硬盘镜像 (disk
5 ## img)
6 dd if=/dev/zero of=target/riscv64imac-unknown-none-elf/release/img bs=512 count=32
7 记录了 32+0 的读入
8 记录了 32+0 的写出
9 16384 字节 (16 kB, 16 KiB) 已复制, 0.000439258 s, 37.3 MB/s
10 ## 通过 qemu-system-riscv64 命令启动 Qemu 模拟器，创建 virtio-blk 设备
11 qemu-system-riscv64 \
12 -drive file=target/riscv64imac-unknown-none-elf/release/img,if=none,format=raw,
13 id=x0 \
14 -device virtio-blk-device,drive=x0 ...
15 ## 可以看到设备驱动查找到的 virtio-blk 设备信息
16 ...
17 [INFO] Detected virtio MMIO device with vendor id 0x554D4551, device type Block,
18 version Modern
19 [INFO] device features: SEG_MAX | GEOMETRY | BLK_SIZE | FLUSH | TOPOLOGY | CONFIG_
20 WCE | DISCARD | WRITE_ZEROES | RING_INDIRECT_DESC | RING_EVENT_IDX | VERSION_1
21 [INFO] config: 0x10008100
22 [INFO] found a block device of size 16KB
23 ...
```

virtio-blk 设备驱动程序了解了 virtio-blk 设备的扇区个数，扇区大小和总体容量后，还需调用 “VirtQueue::new”

成员函数来创建虚拟队列 VirtQueue 数据结构的实例，这样才能进行后续的磁盘读写操作。这个函数主要完成的事情是分配虚拟队列的内存空间，并进行初始化：

- 设定 queue\_size (即虚拟队列的描述符条目数) 为 16;
- 计算满足 queue\_size 的描述符表 desc，可用环 avail 和已用环 used 所需的物理空间的大小 size；
- 基于上面计算的 size 分配物理空间；//VirtQueue.new()
- VirtIOHeader.queue\_set 函数把虚拟队列的相关信息 (内存地址等) 写到 virtio-blk 设备的 MMIO 寄存器中；
- 初始化 VirtQueue 实例中各个成员变量 (主要是 dma, desc, avail, used) 的值。

做完这一步后，virtio-blk 设备和设备驱动之间的虚拟队列接口就打通了，可以进行 I/O 数据读写了。下面简单代码完成了对虚拟硬盘的读写操作和读写正确性检查：

```

1 // virtio-drivers/examples/riscv/src/main.rs
2 fn virtio_blk(header: &'static mut VirtIOHeader) { {
3 // 创建blk结构
4 let mut blk = VirtIOBlk::<HalImpl, T>::new(header).expect("failed to create blk
5 ↪driver");
6 // 读写缓冲区
7 let mut input = vec![0xffu8; 512];
8 let mut output = vec![0; 512];
9 ...
10 // 把input数组内容写入virtio-blk设备
11 blk.write_block(i, &input).expect("failed to write");
12 // 从virtio-blk设备读取内容到output数组
13 blk.read_block(i, &mut output).expect("failed to read");
14 // 检查virtio-blk设备读写的正确性
15 assert_eq!(input, output);
...
}

```

## 10.6.5 操作系统对接 virtio-blk 设备初始化过程

但 virtio\_drivers 模块还没有与操作系统内核进行对接。我们还需在操作系统中封装 virtio-blk 设备，让操作系统内核能够识别并使用 virtio-blk 设备。首先分析一下操作系统需要建立的表示 virtio\_blk 设备的全局变量 BLOCK\_DEVICE：

```

1 // os/src/drivers/block/virtio_blk.rs
2 pub struct VirtIOBlock {
3 virtio_blk: UPIntrFreeCell<VirtIOBlk<'static, VirtioHal>>,
4 condvars: BTreeMap<u16, Condvar>,
5 }
6 // os/easy-fs/src/block_dev.rs
7 pub trait BlockDevice: Send + Sync + Any {
8 fn read_block(&self, block_id: usize, buf: &mut [u8]);
9 fn write_block(&self, block_id: usize, buf: &[u8]);
10 fn handle_irq(&self);
11 }
12 // os/src/boards/qemu.rs
13 pub type BlockDeviceImpl = crate::drivers::block::VirtIOBlock;
14 // os/src/drivers/block/mod.rs
15 lazy_static! {

```

(下页继续)

(续上页)

```

16 pub static ref BLOCK_DEVICE: Arc<dyn BlockDevice> =_
17 ↪Arc::new(BlockDeviceImpl::new());
}

```

从上面的代码可以看到，操作系统中表示 virtio\_blk 设备的全局变量 BLOCK\_DEVICE 的类型是 VirtIOBlock，封装了来自 virtio\_drivers 模块的 VirtIOBlk 类型。这样，操作系统内核就可以通过 BLOCK\_DEVICE 全局变量来访问 virtio\_blk 设备了。而 VirtIOBlock 中的 condvars: BTreeMap<u16, Condvar> 条件变量结构，是用于进程在等待 I/O 读或写操作完全前，通过条件变量让进程处于挂起状态。当 virtio\_blk 设备完成 I/O 操作后，会通过中断唤醒等待的进程。而操作系统对 virtio\_blk 设备的初始化除了封装 VirtIOBlk 类型并调用 VirtIOBlk::<VirtioHal>::new() 外，还需要初始化 condvars 条件变量结构，而每个条件变量对应着一个虚拟队列条目的编号，这意味着每次 I/O 请求都绑定了一个条件变量，让发出请求的线程/进程可以被挂起。

```

1 impl VirtIOBlock {
2 pub fn new() -> Self {
3 let virtio_blk = unsafe {
4 UPIntrFreeCell::new(
5 VirtIOBlk::<VirtioHal>::new(&mut *(VIRTIO0 as *mut VirtIOHeader)).
6 unwrap(),
7)
8 };
9 let mut condvars = BTreeMap::new();
10 let channels = virtio_blk.exclusive_access().virt_queue_size();
11 for i in 0..channels {
12 let condvar = Condvar::new();
13 condvars.insert(i, condvar);
14 }
15 Self {
16 virtio_blk,
17 condvars,
18 }
19 }
}

```

在上述初始化代码中，我们先看到 VIRTIO0，这是 Qemu 模拟的 virtio\_blk 设备中 I/O 寄存器的物理内存地址，VirtIOBlk 需要这个地址来对 VirtIOHeader 数据结构所表示的 virtio-blk I/O 控制寄存器进行读写操作，从而完成对某个具体的 virtio-blk 设备的初始化过程。而且我们还看到了 VirtioHal 结构，它实现 virtio-drivers 模块定义 Hal trait 约定的方法，提供 DMA 内存分配和虚实地址映射操作，从而让 virtio-drivers 模块中 VirtIOBlk 类型能够得到操作系统的服务。

```

1 // os/src/drivers/bus/virtio.rs
2 impl Hal for VirtioHal {
3 fn dma_alloc(pages: usize) -> usize {
4 // 分配页帧 page-frames
5 }
6 let pa: PhysAddr = ppn_base.into();
7 pa.0
8 }
9
10 fn dma_dealloc(pa: usize, pages: usize) -> i32 {
11 // 释放页帧 page-frames
12 0
13 }
14
15 fn phys_to_virt(addr: usize) -> usize {
}

```

(下页继续)

(续上页)

```

16 addr
17 }
18
19 fn virt_to_phys(vaddr: usize) -> usize {
20 //把虚地址转为物理地址
21 }
22

```

## 10.6.6 virtio-blk 设备的 I/O 操作

操作系统的 virtio-blk 驱动的主要功能是给操作系统中的文件系统内核模块提供读写磁盘块的服务，并在对进程管理有一定的影响，但不用直接给应用程序提供服务。在操作系统与 virtio-drivers crate 中 virtio-blk 裸机驱动对接的过程中，需要注意的关键问题是操作系统的 virtio-blk 驱动如何封装 virtio-blk 裸机驱动的基本功能，完成如下服务：

1. 读磁盘块，挂起发起请求的进程/线程；
2. 写磁盘块，挂起发起请求的进程/线程；
3. 对 virtio-blk 设备发出的中断进行处理，唤醒相关等待的进程/线程。

virtio-blk 驱动程序发起的 I/O 请求包含操作类型（读或写）、起始扇区（块设备的最小访问单位的一个扇区的长度 512 字节）、内存地址、访问长度；请求处理完成后返回的 I/O 响应仅包含结果状态（成功或失败，读操作请求的读出扇区内容）。系统产生的一个 I/O 请求在内存中的数据结构分为三个部分：Header（请求头部，包含操作类型和起始扇区）；Data（数据区，包含地址和长度）；Status（结果状态），这些信息分别放在三个 buffer，所以需要三个描述符。

virtio-blk 设备使用 VirtQueue 数据结构来表示虚拟队列进行数据传输，此数据结构主要由三段连续内存组成：描述符表 Descriptor[]、环形队列结构的 AvailRing 和 UsedRing。驱动程序和 virtio-blk 设备都能访问到此数据结构。

描述符表由固定长度（16 字节）的描述符 Descriptor 组成，其个数等于环形队列长度，其中每个 Descriptor 的结构为：

```

1 struct Descriptor {
2 addr: Volatile<u64>,
3 len: Volatile<u32>,
4 flags: Volatile<DescFlags>,
5 next: Volatile<u16>,
6 }

```

包含四个域：addr 代表某段内存的起始地址，长度为 8 个字节；len 代表某段内存的长度，本身占用 4 个字节（因此代表的内存段最大为 4GB）；flags 代表内存段读写属性等，长度为 2 个字节；next 代表下一个内存段对应的 Descriptor 在描述符表中的索引，因此通过 next 字段可以将一个请求对应的多个内存段连接成链表。

可用环 AvailRing 的结构为：

```

1 struct AvailRing {
2 flags: Volatile<u16>,
3 // A driver MUST NOT decrement the idx.
4 idx: Volatile<u16>,
5 ring: [Volatile<u16>; 32], // actual size: queue_size
6 used_event: Volatile<u16>, // unused
7 }

```

可用环由头部的 flags 和 idx 域及 ring 数组组成：flags 与通知机制相关；idx 代表最新放入 IO 请求的编号，从零开始单调递增，将其对队列长度取余即可得该 I/O 请求在可用环数组中的索引；可用环数组元

素用来存放 I/O 请求占用的首个描述符在描述符表中的索引，数组长度等于可用环的长度 (不开启 event\_idx 特性)。

已用环 UsedRing 的结构为：

```

1 struct UsedRing {
2 flags: Volatile<u16>,
3 idx: Volatile<u16>,
4 ring: [UsedElem; 32], // actual size: queue_size
5 avail_event: Volatile<u16>, // unused
6 }
```

已用环由头部的 flags 和 idx 域及 ring 数组组成：flags 与通知机制相关；idx 代表最新放入 I/O 响应的编号，从零开始单调递增，将其对队列长度取余即可得该 I/O 响应在已用环数组中的索引；已用环数组元素主要用来存放 I/O 响应占用的首个描述符在描述符表中的索引，数组长度等于已用环的长度 (不开启 event\_idx 特性)。

针对用户进程发出的 I/O 请求，经过系统调用，文件系统等一系列处理后，最终会形成对 virtio-blk 驱动程序的调用。对于写操作，具体实现如下：

```

1 //virtio-drivers/src/blk.rs
2 pub fn write_block(&mut self, block_id: usize, buf: &[u8]) -> Result {
3 assert_eq!(buf.len(), BLK_SIZE);
4 let req = BlkReq {
5 type_: ReqType::Out,
6 reserved: 0,
7 sector: block_id as u64,
8 };
9 let mut resp = BlkResp::default();
10 self.queue.add(&[req.as_buf(), buf], &[resp.as_buf_mut()])?;
11 self.header.notify(0);
12 while !self.queue.can_pop() {
13 spin_loop();
14 }
15 self.queue.pop_used()?;
16 match resp.status {
17 RespStatus::Ok => Ok(()),
18 - => Err(Error::IoError),
19 }
20 }
```

基本流程如下：

1. 一个完整的 virtio-blk 的 I/O 写请求由三部分组成，包括表示 I/O 写请求信息的结构 BlkReq，要传输的数据块 buf，一个表示设备响应信息的结构 BlkResp。这三部分需要三个描述符来表示；
2. (驱动程序处理) 接着调用 VirtQueue.add 函数，从描述符表中申请三个空闲描述符，每项指向一个内存块，填写上述三部分的信息，并将三个描述符连接成一个描述符链表；
3. (驱动程序处理) 接着调用 VirtQueue.notify 函数，写 MMIO 模式的 queue\_notify 寄存器，即向 virtio-blk 设备发出通知；
4. (设备处理) virtio-blk 设备收到通知后，通过比较 last\_avail (初始为 0) 和 AvailRing 中的 idx 判断是否有新的请求待处理 (如果 last\_vail 小于 AvailRing 中的 idx，则表示有新请求)。如果有，则 last\_avail 加 1，并以 last\_avail 为索引从描述符表中找到这个 I/O 请求对应的描述符链来获知完整的请求信息，并完成存储块的 I/O 写操作；
5. (设备处理) 设备完成 I/O 写操作后 (包括更新包含 BlkResp 的 Descriptor)，将已完成 I/O 的描述符放入 UsedRing 对应的 ring 项中，并更新 idx，代表放入一个响应；如果设置了中断机制，还会产生中断来通知操作系统响应中断；

6. (驱动程序处理) 驱动程序可用轮询机制查看设备是否有响应 (持续调用 `VirtQueue.can_pop` 函数), 通过比较内部的 `VirtQueue.last_used_idx` 和 `VirtQueue.used_idx` 判断是否有新的响应。如果有, 则取出响应 (并更新 `last_used_idx`), 将完成响应对应的三项 `Descriptor` 回收, 最后将结果返回给用户进程。当然, 也可通过中断机制来响应。

I/O 读请求的处理过程与 I/O 写请求的处理过程几乎一样, 仅仅是 `BlkReq` 的内容不同, 写操作中的 `req.type_` 是 `ReqType::Out`, 而读操作中的 `req.type_` 是 `ReqType::In`。具体可以看看 `virtio-drivers/src/blk.rs` 文件中的 `VirtIOBlk.read_block` 函数的实现。

这种基于轮询的 I/O 访问方式效率比较差, 为此, 我们需要实现基于中断的 I/O 访问方式。为此在支持中断的 `write_block_nb` 方法:

```

1 pub unsafe fn write_block_nb(
2 &mut self,
3 block_id: usize,
4 buf: &[u8],
5 resp: &mut BlkResp,
6) -> Result<u16> {
7 assert_eq!(buf.len(), BLK_SIZE);
8 let req = BlkReq {
9 type_: ReqType::Out,
10 reserved: 0,
11 sector: block_id as u64,
12 };
13 let token = self.queue.add(&[req.as_buf(), buf], &[resp.as_buf_mut()])?;
14 self.header.notify(0);
15 Ok(token)
16 }
17
18 // Acknowledge interrupt.
19 pub fn ack_interrupt(&mut self) -> bool {
20 self.header.ack_interrupt()
21 }
```

与不支持中的 `write_block` 函数比起来, `write_block_nb` 函数更简单了, 在发出 I/O 请求后, 就直接返回了。`read_block_nb` 函数的处理流程与此一致。而响应中断的 `ack_interrupt` 函数只是完成了非常基本的 virtio 设备的中断响应操作。在 `virtio-drivers` 中实现的 virtio 设备驱动是看不到进程、条件变量等操作系统的各种关键要素, 只有与操作系统内核对接, 才能完整实现基于中断的 I/O 访问方式。

## 10.6.7 操作系统对接 virtio-blk 设备 I/O 处理

操作系统中的文件系统模块与操作系统中的块设备驱动程序 `VirtIOBlock` 直接交互, 而操作系统中的块设备驱动程序 `VirtIOBlock` 封装了 `virtio-drivers` 中实现的 `virtio_blk` 设备驱动。在文件系统的介绍中, 我们并没有深入分析 `virtio_blk` 设备。这里我们将介绍操作系统对接 `virtio_blk` 设备驱动并完成基于中断机制的 I/O 处理过程。

接下来需要扩展文件系统对块设备驱动的 I/O 访问要求, 这体现在 `BlockDevice` trait 的新定义中增加了 `handle_irq` 方法, 而操作系统的 `virtio_blk` 设备驱动程序中的 `VirtIOBlock` 实现了这个方法, 并且实现了既支持轮询方式, 也支持中断方式的块读写操作。

```

1 // easy-fs/src/block_dev.rs
2 pub trait BlockDevice: Send + Sync + Any {
3 fn read_block(&self, block_id: usize, buf: &mut [u8]);
4 fn write_block(&self, block_id: usize, buf: &[u8]);
5 // 更新的部分: 增加对块设备中断的处理
6 fn handle_irq(&self);
```

(下页继续)

(续上页)

```

7 }
8 // os/src/drivers/block/virtio_blk.rs
9 impl BlockDevice for VirtIOBlock {
10 fn handle_irq(&self) {
11 self.virtio_blk.exclusive_session(|blk| {
12 while let Ok(token) = blk.pop_used() {
13 // 唤醒等待该块设备I/O完成的线程/进程
14 self.condvars.get(&token).unwrap().signal();
15 }
16 });
17 }
18
19 fn read_block(&self, block_id: usize, buf: &mut [u8]) {
20 // 获取轮询或中断的配置标记
21 let nb = *DEV_NON_BLOCKING_ACCESS.exclusive_access();
22 if nb { // 如果是中断方式
23 let mut resp = BlkResp::default();
24 let task_cx_ptr = self.virtio_blk.exclusive_session(|blk| {
25 // 基于中断方式的块读请求
26 let token = unsafe { blk.read_block_nb(block_id, buf, &mut resp).
27 unwrap() };
28 // 将当前线程/进程加入条件变量的等待队列
29 self.condvars.get(&token).unwrap().wait_no_sched();
30 });
31 // 切换线程/进程
32 schedule(task_cx_ptr);
33 assert_eq!(
34 resp.status(),
35 RespStatus::Ok,
36 "Error when reading VirtIOBlk"
37);
38 } else { // 如果是轮询方式，则进行轮询式的块读请求
39 self.virtio_blk
40 .exclusive_access()
41 .read_block(block_id, buf)
42 .expect("Error when reading VirtIOBlk");
43 }
44 }
}

```

write\_block 写操作与 read\_block 读操作的处理过程一致，这里不再赘述。

然后需要对操作系统整体的中断处理过程进行调整，以支持对基于中断方式的块读写操作：

```

1 // os/src/trap/mode.rs
2 // 在用户态接收到外设中断
3 pub fn trap_handler() -> ! {
4 ...
5 crate::board::irq_handler();
6 // 在内核态接收到外设中断
7 pub fn trap_from_kernel(_trap_cx: &TrapContext) {
8 ...
9 crate::board::irq_handler();
10 // os/src/boards/qemu.rs
11 pub fn irq_handler() {
12 let mut plic = unsafe { PLIC::new(VIRT_PLIC) };
13 // 获得外设中断号
14 let intr_src_id = plic.claim(0, IntrTargetPriority::Supervisor);
}

```

(下页继续)

(续上页)

```

15 match intr_src_id {
16 ...
17 // 处理 virtio_blk 设备产生的中断
18 8 => BLOCK_DEVICE.handle_irq(),
19 }
20 // 完成中断响应
21 plic.complete(0, IntrTargetPriority::Supervisor, intr_src_id);
22 }

```

BLOCK\_DEVICE.handle\_irq() 执行的就是 VirtIOBlock 实现的中断处理方法 handle\_irq()，从而让等待在块读写的进程/线程得以继续执行。

有了基于中断方式的块读写操作后，当某个线程/进程由于块读写操作无法继续执行时，操作系统可以切换到其它处于就绪态的线程/进程执行，从而让计算机系统的整体执行效率得到提升。

## 10.7 virtio\_gpu 设备驱动程序

### 10.7.1 本节导读

本节主要介绍了与操作系统无关的基本 virtio\_gpu 设备驱动程序的设计与实现，以及如何在操作系统中封装 virtio\_gpu 设备驱动程序，实现对丰富多彩的 GUI app 的支持。

### 10.7.2 virtio-gpu 驱动程序

让操作系统能够显示图形是一个有趣的目标。这可以通过在 QEMU 或带显示屏的开发板上写显示驱动程序来完成。这里我们主要介绍如何驱动基于 QEMU 的 virtio-gpu 虚拟显示设备。大家不用担心这个驱动实现很困难，其实它主要完成的事情就是对显示内存进行写操作而已。我们看到的图形显示屏幕其实是由一个一个的像素点来组成的，显示驱动程序的主要目标就是把每个像素点用内存单元来表示，并把代表所有这些像素点的内存区域（也称显示内存，显存，frame buffer）“通知”显示 I/O 控制器（也称图形适配器，graphics adapter），然后显示 I/O 控制器会根据内存内容渲染到图形显示屏上。这里我们以 Rust 语言为例，给出 virtio-gpu 设备驱动程序的设计与实现。主要包括如下内容：

- virtio-gpu 设备的关键数据结构
- 初始化 virtio-gpu 设备
- 操作系统对接 virtio-gpu 设备初始化
- virtio-gpu 设备的 I/O 操作
- 操作系统对接 virtio-gpu 设备 I/O 操作

### 10.7.3 virtio-gpu 设备的关键数据结构

```

1 // virtio-drivers/src/gpu.rs
2 pub struct VirtIOGpu<'a, H: Hal> {
3 header: &'static mut VirtIOHeader,
4 /// 显示区域的分辨率
5 rect: Rect,
6 /// 显示内存 frame buffer
7 frame_buffer_dma: Option<DMA<H>>,
8 /// 光标图像内存 cursor image buffer.

```

(下页继续)

(续上页)

```

9 cursor_buffer_dma: Option<DMA<H>>,
10 /// Queue for sending control commands.
11 control_queue: VirtQueue<'a, H>,
12 /// Queue for sending cursor commands.
13 cursor_queue: VirtQueue<'a, H>,
14 /// Queue buffer
15 queue_buf_dma: DMA<H>,
16 /// Send buffer for queue.
17 queue_buf_send: &'a mut [u8],
18 /// Recv buffer for queue.
19 queue_buf_recv: &'a mut [u8],
20 }

```

header 成员对应的 VirtIOHeader 数据结构是 virtio 设备的共有属性，包括版本号、设备 id、设备特征等信息，其内存布局和成员变量的含义与本章前述的 virt-mmio 设备的 [寄存器内存布局](#) 是一致的。而 [VirtQueue](#) 数据结构的 [内存布局](#) 和 [virtqueue](#) 的含义与本章前述内容一致。与具体操作系统相关的服务函数接口 [Hal](#) 在上一节已经介绍过，这里不再赘述。

显示内存区域 `frame_buffer_dma` 是一块要由操作系统或运行时分配的显示内存，当把表示像素点的值就写入这个区域后，virtio-gpu 设备会在 Qemu 虚拟的显示器上显示出图形。光标图像内存区域 `cursor_buffer_dma` 用于存放光标图像的数据，当光标图像数据更新后，virtio-gpu 设备会在 Qemu 虚拟的显示器上显示出光标图像。这两块区域与 `queue_buf_dma` 区域都是用于与 I/O 设备进行数据传输的 [DMA 内存](#)，都由操作系统进行分配等管理。所以在 `virtio_drivers` 中建立了对应的 DMA 结构，用于操作系统管理这些 DMA 内存。

```

1 // virtio-drivers/src/gpu.rs
2 pub struct DMA<H: Hal> {
3 paddr: usize, // DMA 内存起始物理地址
4 pages: usize, // DMA 内存所占物理页数量
5 ...
6 }
7 impl<H: Hal> DMA<H> {
8 pub fn new(pages: usize) -> Result<Self> {
9 // 操作系统分配 pages*页大小的 DMA 内存
10 let paddr = H::dma_alloc(pages);
11 ...
12 }
13 // DMA 内存的物理地址
14 pub fn paddr(&self) -> usize {
15 self.paddr
16 }
17 // DMA 内存的虚拟地址
18 pub fn vaddr(&self) -> usize {
19 H::phys_to_virt(self.paddr)
20 }
21 // DMA 内存的物理页帧号
22 pub fn pfn(&self) -> u32 {
23 (self.paddr >> 12) as u32
24 }
25 // 把 DMA 内存转为便于 Rust 处理的可变一维数组切片
26 pub unsafe fn as_buf(&self) -> &'static mut [u8] {
27 core::slice::from_raw_parts_mut(self.vaddr() as _, PAGE_SIZE * self.pages as_
28 -> usize)
29 ...
30 }
31 impl<H: Hal> Drop for DMA<H> {
32 // 操作系统释放 DMA 内存
33 }

```

(下页继续)

(续上页)

```

31 fn drop(&mut self) {
32 let err = H::dma_dealloc(self.paddr as usize, self.pages as usize);
33 ...

```

virtio-gpu 驱动程序与 virtio-gpu 设备之间通过两个 virtqueue 来进行交互访问, control\_queue 用于驱动程序发送显示相关控制命令 (如设置显示内存的起始地址等) 给 virtio-gpu 设备, cursor\_queue 用于驱动程序给给 virtio-gpu 设备发送显示鼠标更新的相关控制命令。queue\_buf\_dma 是存放控制命令和返回结果的内存, queue\_buf\_send 和 queue\_buf\_recv 是 queue\_buf\_dma DMA 内存的可变一维数组切片形式, 分别用于虚拟队列的接收与发送。

#### 10.7.4 初始化 virtio-gpu 设备

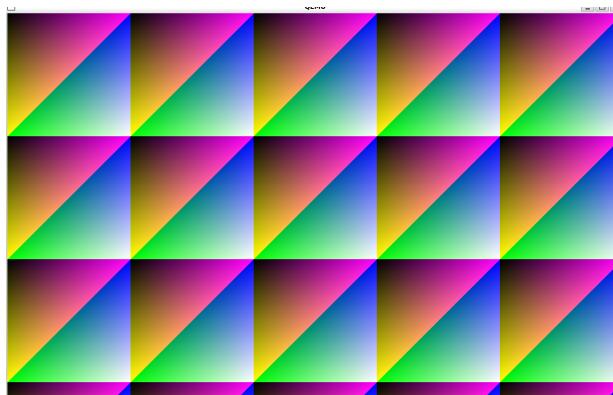
在 virtio-drivers crate 的 examples/riscv/src/main.rs 文件中的 virtio\_probe 函数识别出 virtio-gpu 设备后, 会调用 virtio\_gpu(header) 函数来完成对 virtio-gpu 设备的初始化过程。virtio-gpu 设备初始化的工作主要是查询显示设备的信息 (如分辨率等), 并将该信息用于初始显示扫描 (scanout) 设置。下面的命令可以看到虚拟 GPU 的创建和识别过程:

```

1 # 在 virtio-drivers 仓库的 example/riscv 目录下执行如下命令
2 make run
3 ## 通过 qemu-system-riscv64 命令启动 Qemu 模拟器, 创建 virtio-gpu 设备
4 qemu-system-riscv64 \
5 -device virtio-gpu-device ...
6 ## 可以看到设备驱动查找到的 virtio-gpu 设备色信息
7 ...
8 [INFO] Detected virtio MMIO device with vendor id 0x554D4551, device type GPU, \
9 ↳ version Modern
10 [INFO] Device features EDID | RING_INDIRECT_DESC | RING_EVENT_IDX | VERSION_1
11 [INFO] events_read: 0x0, num_scanouts: 0x1
12 [INFO] GPU resolution is 1280x800
13 [INFO] => RespDisplayInfo { header: CtrlHeader { hdr_type: OkDisplayInfo, flags: 0, \
14 ↳ fence_id: 0, ctx_id: 0, _padding: 0 }, rect: Rect { x: 0, y: 0, width: 1280, \
15 ↳ height: 800 }, enabled: 1, flags: 0 }

```

并看到 Qemu 输出的图形显示:



接下来我们看看 virtio-gpu 设备初始化的具体过程如:

```

1 // virtio-drivers/examples/riscv/src/main.rs
2 fn virtio_gpu(header: &'static mut VirtIOHeader) {
3 let mut gpu = VirtIOGpu::<HalImpl>::new(header).expect("failed to create gpu \
4 ↳ driver");

```

(下页继续)

(续上页)

```

4 let (width, height) = gpu.resolution().expect("failed to get resolution");
5 info!("GPU resolution is {}x{}", width, height);
6 let fb = gpu.setup_framebuffer().expect("failed to get fb");
7 ...

```

在 `virtio_gpu` 函数调用创建了 `VirtIOGpu::<HalImpl>::new(header)` 函数，获得关于 virtio-gpu 设备的重要信息：显示分辨率 1280x800；而且会建立 virtio 虚拟队列，并基于这些信息来创建表示 virtio-gpu 的 gpu 结构。然后会进一步调用 `gpu.setup_framebuffer()` 函数来建立和配置显示内存缓冲区，打通设备驱动与 virtio-gpu 设备间的显示数据传输通道。

`VirtIOGpu::<HalImpl>::new(header)` 函数主要完成了 virtio-gpu 设备的初始化工作：

```

1 // virtio-drivers/src/gpu.rs
2 impl VirtIOGpu<'_, H> {
3 pub fn new(header: &'static mut VirtIOHeader) -> Result<Self> {
4 header.begin_init(|features| {
5 let features = Features::from_bits_truncate(features);
6 let supported_features = Features::empty();
7 (features & supported_features).bits()
8 });
9
10 // read configuration space
11 let config = unsafe { &mut * (header.config_space() as *mut Config) };
12
13 let control_queue = VirtQueue::new(header, QUEUE_TRANSMIT, 2)?;
14 let cursor_queue = VirtQueue::new(header, QUEUE_CURSOR, 2)?;
15
16 let queue_buf_dma = DMA::new(2)?;
17 let queue_buf_send = unsafe { &mut queue_buf_dma.as_buf() [..PAGE_SIZE] };
18 let queue_buf_recv = unsafe { &mut queue_buf_dma.as_buf() [PAGE_SIZE..] };
19
20 header.finish_init();
21
22 Ok(VirtIOGpu {
23 header,
24 frame_buffer_dma: None,
25 rect: Rect::default(),
26 control_queue,
27 cursor_queue,
28 queue_buf_dma,
29 queue_buf_send,
30 queue_buf_recv,
31 })
32 }
}

```

首先是 `header.begin_init` 函数完成了对 virtio 设备的共性初始化的常规步骤的前六步；第七步在这里被忽略；第八步读取 virtio-gpu 设备的配置空间（config space）信息；紧接着是创建两个虚拟队列：控制命令队列、光标管理队列。并为控制命令队列分配两个 page（8KB）的内存空间用于放置虚拟队列中的控制命令和返回结果；最后的第九步，调用 `header.finish_init` 函数，将 virtio-gpu 设备设置为活跃可用状态。

虽然 virtio-gpu 初始化完毕，但它目前还不能进行显示。为了能够进行正常的显示，我们还需建立显存区域 frame buffer，并绑定在 virtio-gpu 设备上。这主要是通过 `VirtIOGpu.setup_framebuffer` 函数来完成的。

```

1 // virtio-drivers/src/gpu.rs
2 pub fn setup_framebuffer(&mut self) -> Result<&mut [u8]> {
3 // get display info
4 let display_info: RespDisplayInfo =

```

(下页继续)

(续上页)

```

5 self.request(CtrlHeader::with_type(Command::GetDisplayInfo))?;
6 display_info.header.check_type(Command::OkDisplayInfo)?;
7 self.rect = display_info.rect;
8
9 // create resource 2d
10 let rsp: CtrlHeader = self.request(ResourceCreate2D {
11 header: CtrlHeader::with_type(Command::ResourceCreate2d),
12 resource_id: RESOURCE_ID,
13 format: Format::B8G8R8A8UNORM,
14 width: display_info.rect.width,
15 height: display_info.rect.height,
16 })?;
17 rsp.check_type(Command::OkNodata)?;
18
19 // alloc continuous pages for the frame buffer
20 let size = display_info.rect.width * display_info.rect.height * 4;
21 let frame_buffer_dma = DMA::new(pages(size as usize))?;
22
23 // resource_attach_backing
24 let rsp: CtrlHeader = self.request(ResourceAttachBacking {
25 header: CtrlHeader::with_type(Command::ResourceAttachBacking),
26 resource_id: RESOURCE_ID,
27 nr_entries: 1,
28 addr: frame_buffer_dma.paddr() as u64,
29 length: size,
30 padding: 0,
31 })?;
32 rsp.check_type(Command::OkNodata)?;
33
34 // map frame buffer to screen
35 let rsp: CtrlHeader = self.request(SetScanout {
36 header: CtrlHeader::with_type(Command::SetScanout),
37 rect: display_info.rect,
38 scanout_id: 0,
39 resource_id: RESOURCE_ID,
40 })?;
41 rsp.check_type(Command::OkNodata)?;
42
43 let buf = unsafe { frame_buffer_dma.as_buf() };
44 self.frame_buffer_dma = Some(frame_buffer_dma);
45 Ok(buf)
46 }

```

上面的函数主要完成的工作有如下几个步骤，其实就是驱动程序给 virtio-gpu 设备发控制命令，建立好显存区域：

1. 发出 GetDisplayInfo 命令，获得 virtio-gpu 设备的显示分辨率；
2. 发出 ResourceCreate2D 命令，让设备以分辨率大小 (width \*height)，像素信息 (Red/Green/Blue/Alpha 各占 1 字节大小，即一个像素占 4 字节)，来配置设备显示资源；
3. 分配 width \*height \* 4 字节的连续物理内存空间作为显存，发出 ResourceAttachBacking 命令，让设备把显存附加到设备显示资源上；
4. 发出 SetScanout 命令，把设备显示资源链接到显示扫描输出上，这样才能让显存的像素信息显示出来；

到这一步，才算是把 virtio-gpu 设备初始化完成了。做完这一步后，virtio-gpu 设备和设备驱动之间的虚拟队列接口就打通了，显示缓冲区也建立好了，就可以进行显存数据读写了。

## 10.7.5 virtio-gpu 设备的 I/O 操作

对初始化好的 virtio-gpu 设备进行图形显示其实很简单，主要就是两个步骤：

1. 把要显示的像素数据写入到显存中；
2. 发出刷新命令，让 virtio-gpu 在 Qemu 模拟的显示区上显示图形。

下面简单代码完成了对虚拟 GPU 的图形显示：

```

1 // virtio-drivers/src/gpu.rs
2 fn virtio_gpu(header: &'static mut VirtIOHeader) {
3 ...
4 // 把像素数据写入显存
5 for y in 0..height { //height=800
6 for x in 0..width { //width=1280
7 let idx = (y * width + x) * 4;
8 fb[idx] = x as u8;
9 fb[idx + 1] = y as u8;
10 fb[idx + 2] = (x + y) as u8;
11 }
12 }
13 // 发出刷新命令
14 gpu.flush().expect("failed to flush");

```

这里需要注意的是对 virtio-gpu 进行刷新操作比较耗时，所以我们尽量先把显示的图形像素值都写入显存中，再发出刷新命令，减少刷新命令的执行次数。

## 10.7.6 操作系统对接 virtio-gpu 设备初始化

虽然 virtio-gpu 设备驱动程序已经完成了，但是还需要操作系统对接 virtio-gpu 设备，才能真正的把 virtio-gpu 设备驱动程序和操作系统对接起来。这里以侏罗猎龙操作系统—Device OS 为例，来介绍 virtio-gpu 设备在操作系统中的初始化过程。其初始化过程主要包括：

1. 调用 virtio-drivers/gpu.rs 中提供 VirtIOGpu::new() 方法，初始化 virtio\_gpu 设备；
2. 建立显存缓冲区的可变一维数组引用，便于后续写显存来显示图形；
3. 建立显示窗口中的光标图形；
4. 设定表示 virtio\_gpu 设备的全局变量。

先看看操作系统需要建立的表示 virtio\_gpu 设备的全局变量 GPU\_DEVICE：

```

1 // os/src/drivers/gpu/mod.rs
2 pub trait GpuDevice: Send + Sync + Any {
3 fn update_cursor(&self); //更新光标，目前暂时没用
4 fn get_framebuffer(&self) -> &mut [u8];
5 fn flush(&self);
6 }
7 pub struct VirtIOGpuWrapper {
8 gpu: UPIntrFreeCell<VirtIOGpu<'static, VirtioHal>>,
9 fb: &'static [u8],
10 }
11 lazy_static::lazy_static!{
12 pub static ref GPU_DEVICE: Arc<dyn GpuDevice> = Arc::new(VirtIOGpuWrapper::new());
13 }

```

从上面的代码可以看到，操作系统中表示表示 virtio\_gpu 设备的全局变量 GPU\_DEVICE 的类型是 VirtIOGpuWrapper，封装了来自 virtio\_drivers 模块的 VirtIOGpu 类型，以及一维字节数组引用表示的显存缓冲区 fb。这样，操作系统内核就可以通过 GPU\_DEVICE 全局变量来访问 gpu\_blk 设备了。而操作系统对 virtio\_blk 设备的初始化就是调用 VirtIOGpuWrapper::<VirtioHal>::new()。

当用户态应用程序要进行图形显示时，至少需要得到操作系统的两个基本图形显示服务。一个是得到显存在用户态可访问的内存地址，这样应用程序可以在用户态把表示图形的像素值写入显存中；第二个是给 virtio-gpu 设备发出 flush 刷新指令，这样 virtio-gpu 设备能够更新显示器中的图形显示内容。

为此，操作系统在 VirtIOGpuWrapper 结构类型中需要实现 GpuDevice trait，该 trait 需要实现两个函数来支持应用程序所需要的基本显示服务：

```

1 impl GpuDevice for VirtIOGpuWrapper {
2 // 通知virtio-gpu设备更新图形显示内容
3 fn flush(&self) {
4 self.gpu.exclusive_access().flush().unwrap();
5 }
6 // 得到显存的基于内核态虚地址的一维字节数组引用
7 fn get_framebuffer(&self) -> &mut [u8] {
8 unsafe {
9 let ptr = self.fb.as_ptr() as *const _ as *mut u8;
10 core::slice::from_raw_parts_mut(ptr, self.fb.len())
11 }
12 }
13 }

```

接下来，看一下操作系统对 virtio-gpu 设备的初始化过程：

```

1 // os/src/drivers/gpu/mod.rs
2 impl VirtIOGpuWrapper {
3 pub fn new() -> Self {
4 unsafe {
5 // 1. 执行virtio-drivers的gpu.rs中virtio-gpu基本初始化
6 let mut virtio =
7 VirtIOGpu::<VirtioHal>::new(&mut *(VIRTIO7 as *mut VirtIOHeader)).
8 unwrap();
9 // 2. 设置virtio-gpu设备的显存，初始化显存的一维字节数组引用
10 let framebuffer = virtio.setup_framebuffer().unwrap();
11 let len = framebuffer.len();
12 let ptr = framebuffer.as_mut_ptr();
13 let fb = core::slice::from_raw_parts_mut(ptr, len);
14 // 3. 初始化光标图像的像素值
15 let bmp = Bmp::<Rgb888>::from_slice(BMP_DATA).unwrap();
16 let raw = bmp.as_raw();
17 let mut b = Vec::new();
18 for i in raw.image_data().chunks(3) {
19 let mut v = i.to_vec();
20 b.append(&mut v);
21 if i == [255, 255, 255] {
22 b.push(0x0)
23 } else {
24 b.push(0xff)
25 }
26 }
27 // 4. 设置virtio-gpu设备的光标图像
28 virtio.setup_cursor(b.as_slice(), 50, 50, 50, 50).unwrap();
29 // 5. 返回VirtIOGpuWrapper结构类型
30 Self {

```

(下页继续)

(续上页)

```

30 gpu: UPIntrFreeCell::new(virtio),
31 fb,
32 }
33 ...

```

在上述初始化过程中，我们先看到 VIRTIO7，这是 Qemu 模拟的 virtio\_gpu 设备中 I/O 寄存器的物理内存地址，VirtIOGpu 需要这个地址来对 VirtIOHeader 数据结构所表示的 virtio-gpu I/O 控制寄存器进行读写操作，从而完成对某个具体的 virtio-gpu 设备的初始化过程。整个初始化过程的步骤如下：

1. 执行 virtio-drivers 的 gpu.rs 中 virtio-gpu 基本初始化
2. 设置 virtio-gpu 设备的显存，初始化显存的一维字节数组引用
3. (可选) 初始化光标图像的像素值
4. (可选) 设置 virtio-gpu 设备的光标图像
5. 返回 VirtIOGpuWrapper 结构类型

上述步骤的第一步 “*virtio-gpu 基本初始化*” 和第二步 *设置显存* 是核心内容，都由 virtio-drivers 中与具体操作系统无关的 virtio-gpu 裸机驱动实现，极大降低本章从操作系统的代码复杂性。至此，我们已经完成了操作系统对 virtio-gpu 设备的初始化过程，接下来，我们看一下操作系统对 virtio-gpu 设备的 I/O 处理过程。

### 10.7.7 操作系统对接 virtio-gpu 设备 I/O 处理

操作系统的 virtio-gpu 驱动的主要功能是给操作系统提供支持，让运行在用户态应用能够显示图形。为此，应用程序需要知道可读写的显存在哪里，并能把更新的像素值写入显存。另外还需要能够通知 virtio-gpu 设备更新显示内容。可以看出，这主要与操作系统的进程管理和虚存管理有直接的关系。

在操作系统与 virtio-drivers crate 中 virtio-gpu 裸机驱动对接的过程中，需要注意的关键问题是操作系统的 virtio-gpu 驱动如何封装 virtio-blk 裸机驱动的基本功能，完成如下服务：

1. 根据 virtio-gpu 裸机驱动提供的显存信息，建立应用程序访问的用户态显存地址空间；
2. 实现系统调用，把用户态显存地址空间的基址和范围发给应用程序；
3. 实现系统调用，把更新显存的命令发给 virtio-gpu 设备。

这里我们还是做了一些简化，即应用程序预先知道了 virtio-blk 的显示分辨率为 1280x800，采用的是 R/G/B/Alpha 像素显示，即一个像素点占用 4 个字节。这样整个显存大小为  $1280 \times 800 \times 4 = 4096000$  字节，即大约 4000KB，近 4MB。

我们先看看图形应用程序所需要的两个系统调用：

```

1 // os/src/syscall/mod.rs
2 const SYSCALL_FRAMEBUFFER: usize = 2000;
3 const SYSCALL_FRAMEBUFFER_FLUSH: usize = 2001;
4 // os/src/syscall/gui.rs
5 // 显存的用户态起始虚拟地址
6 const FB_VADDR: usize = 0x10000000;
7 pub fn sys_framebuffer() -> isize {
8 // 获得显存的起始物理页帧和结束物理页帧
9 let gpu = GPU_DEVICE.clone();
10 let fb = gpu.get_framebuffer();
11 let len = fb.len();
12 let fb_ppn = PhysAddr::from(fb.as_ptr() as usize).floor();
13 let fb_end_ppn = PhysAddr::from(fb.as_ptr() as usize + len).ceil();
14 // 获取当前进程的地址空间结构 mem_set

```

(下页继续)

(续上页)

```

15 let current_process = current_process();
16 let mut inner = current_process.inner_exclusive_access();
17 let mem_set = &mut inner.memory_set;
18 // 把显存的物理页帧映射到起始地址为FB_VADDR的用户态虚拟地址空间
19 mem_set.push_noalloc(
20 MapArea::new(
21 (FB_VADDR as usize).into(),
22 (FB_VADDR + len as usize).into(),
23 MapType::Framed,
24 MapPermission::R | MapPermission::W | MapPermission::U,
25),
26 PPNRange::new(fb_ppn, fb_end_ppn),
27);
28 // 返回起始地址为FB_VADDR
29 FB_VADDR as isize
30 }
31 // 要求virtio-gpu设备刷新图形显示
32 pub fn sys_FRAMEBUFFER_FLUSH() -> isize {
33 let gpu = GPU_DEVICE.clone();
34 gpu.flush();
35 0
36 }
```

有了这两个系统调用，就可以很容易建立图形应用程序了。下面这个应用程序，可以在 Qemu 模拟的屏幕上显示一个彩色的矩形。

```

1 // user/src/bin/gui_simple.rs
2 pub const VIRTGPU_XRES: usize = 1280; // 显示分辨率的宽度
3 pub const VIRTGPU_YRES: usize = 800; // 显示分辨率的高度
4 pub fn main() -> i32 {
5 // 访问sys_FRAMEBUFFER系统调用，获得显存基址
6 let fb_ptr = framebuffer() as *mut u8;
7 // 把显存转换为一维字节数组
8 let fb = unsafe { core::slice::from_raw_parts_mut(fb_ptr as *mut u8, VIRTGPU_
9 ↪ XRES * VIRTGPU_YRES * 4 as usize) };
10 // 更新显存的像素值
11 for y in 0..800 {
12 for x in 0..1280 {
13 let idx = (y * 1280 + x) * 4;
14 fb[idx] = x as u8;
15 fb[idx + 1] = y as u8;
16 fb[idx + 2] = (x + y) as u8;
17 }
18 }
19 // 访问sys_FRAMEBUFFER_FLUSH系统调用，要求virtio-gpu设备刷新图形显示
20 framebuffer_flush();
21 0
22 }
```

到目前为止，看到的操作系统支持工作还是比较简单的，但其实我们还没分析如何给应用程序提供显存虚拟地址空间的。以前章节的操作系统支持应用程序的用户态地址空间，都是在创建应用程序对应进程的初始化过程中建立，涉及不少工作，具体包括：

- 分配空闲物理页帧
- 建立进程地址空间 (Address Space) 中的逻辑段 (MemArea)
- 建立映射物理页帧和虚拟页的页表

目前这些工作不能直接支持建立用户态显存地址空间。主要原因在于，用户态显存的物理页帧分配和物理虚地址页表映射，都是由 virtio-gpu 裸机设备驱动程序在设备初始化时完成。在图形应用进程的创建过程中，不需要再分配显存的物理页帧了，只需建立显存的用户态虚拟地址空间。

为了支持操作系统把用户态显存地址空间的基址发给应用程序，需要对操作系统的虚拟内存管理进行一定的扩展，即实现 sys\_FRAMEBUFFER 系统调用中访问的 mem\_set.push\_noalloc 新函数和其它相关函数。

```

1 // os/src/mm/memory_set.rs
2 impl MemorySet {
3 pub fn push_noalloc(&mut self, mut map_area: MapArea, ppn_range: PPNRange) {
4 map_area.map_noalloc(&mut self.page_table, ppn_range);
5 self.areas.push(map_area);
6 }
7 impl MapArea {
8 pub fn map_noalloc(&mut self, page_table: &mut PageTable, ppn_range: PPNRange) {
9 for (vpn, ppn) in core::iter::zip(self.vpn_range, ppn_range) {
10 self.data_frames.insert(vpn, FrameTracker::new_noalloc(ppn));
11 let pte_flags = PTEFlags::from_bits(self.map_perm.bits).unwrap();
12 page_table.map(vpn, ppn, pte_flags);
13 }
14 }
15 // os/src/mm/frame_allocator.rs
16 pub struct FrameTracker {
17 pub ppn: PhysPageNum,
18 pub nodrop: bool,
19 }
20 impl FrameTracker {
21 pub fn new_nowrite(ppn: PhysPageNum) -> Self {
22 Self { ppn, nodrop: true }
23 }
24 impl Drop for FrameTracker {
25 fn drop(&mut self) {
26 if self.nodrop {
27 return;
28 }
29 frame_dealloc(self.ppn);
30 }
31 }
}

```

这样，就可以实现把某一块已分配的物理页帧映射到进程的用户态虚拟地址空间，并且在进程退出是否地址空间的物理页帧时，不会把显存的物理页帧给释放掉。

### 10.7.8 图形化应用程序设计

现在操作系统有了显示的彩色图形显示功能，也有通过串口接收输入的功能，我们就可以设计更加丰富多彩的应用了。这里简单介绍一个贪吃蛇图形小游戏的设计。

**注解：**“贪吃蛇”游戏简介

游戏中的元素主要有蛇和食物组成，蛇的身体是由若干个格子组成的，初始化时蛇的身体只有一格，吃了食物后会增长。食物也是一个格子，代表食物的格子位置随机产生。游戏的主要运行逻辑是，蛇可以移动，通过用户输入的字母 asdw 的控制蛇的上下左右移动的方向。用户通过移动贪吃蛇，并与食物格子位置重合，来增加蛇的身体长度。用户输入回车键时，游戏结束。

为了简化设计，我们移植了 `embedded-graphics` 嵌入式图形库<sup>1</sup> 到侏罗猎龙操作系统中，并修改了一个基于此图形库的 Linux 图形应用`embedded-snake-rs`<sup>2</sup>，让它在侏罗猎龙操作系统中能够运行。

### 移植 `embedded-graphics` 嵌入式图形库

`embedded-graphics` 嵌入式图形库给出了很详细的移植说明，主要是实现 `embedded_graphics_core::draw_target::DrawTarget` trait 中的函数接口 `fn draw_iter<I>(&mut self, pixels: I)`。为此需要为图形应用建立一个能够表示显存、像素点特征和显示区域的数据结构 `Display` 和创建函数 `new()`：

```

1 pub struct Display {
2 pub size: Size,
3 pub point: Point,
4 pub fb: &'static mut [u8],
5 }
6 impl Display {
7 pub fn new(size: Size, point: Point) -> Self {
8 let fb_ptr = framebuffer() as *mut u8;
9 println!(
10 "Hello world from user mode program! 0x{:X} , len {}",
11 fb_ptr as usize, VIRTGPU_LEN
12);
13 let fb =
14 unsafe { core::slice::from_raw_parts_mut(fb_ptr as *mut u8, VIRTGPU_LEN_
15 as usize) };
16 Self { size, point, fb }
17 }
18 }

```

在这个 `Display` 结构的基础上，我们就可以实现 `DrawTarget` trait 要求的函数：

```

1 impl OriginDimensions for Display {
2 fn size(&self) -> Size {
3 self.size
4 }
5 }
6
7 impl DrawTarget for Display {
8 type Color = Rgb888;
9 type Error = core::convert::Infallible;
10
11 fn draw_iter<I>(&mut self, pixels: I) -> Result<(), Self::Error>
12 where
13 I: IntoIterator<Item = embedded_graphics::Pixel<Self::Color>>,
14 {
15 pixels.into_iter().for_each(|px| {
16 let idx = ((self.point.y + px.0.y) * VIRTGPU_XRES as i32 + self.point.x +_
17 px.0.x)
18 as usize
19 * 4;
20 if idx + 2 >= self.fb.len() {
21 return;
22 }
23 self.fb[idx] = px.1.b();
24 });
25 }
26 }

```

(下页继续)

<sup>1</sup> <https://github.com/embedded-graphics/embedded-graphics>

<sup>2</sup> <https://github.com/libesz/embedded-snake-rs>

(续上页)

```

23 self.fb[idx + 1] = px.1.g();
24 self.fb[idx + 2] = px.1.r();
25 });
26 framebuffer_flush();
27 Ok(())
28 }
29 }
```

上述的 `draw_iter()` 函数实现了对一个由像素元素组成的显示区域的绘制迭代器，将迭代器中的像素元素绘制到 `Display` 结构中的显存中，并调用 `framebuffer_flush()` 函数将显存中的内容刷新到屏幕上。这样，`embedded-graphics` 嵌入式图形库在侏罗猎龙操作系统的移植任务就完成了。

## 实现贪吃蛇游戏图形应用

`embedded-snake-rs` 的具体实现大约有 200 多行代码，提供了一系列的数据结构，主要的数据结构（包含相关方法实现）包括：

- `ScaledDisplay`：封装了 `Display` 并支持显示大小可缩放的方块
- `Food`：会在随机位置产生并定期消失的“食物”方块
- `Snake`：“贪吃蛇”方块，长度由一系列的方块组成，可以通过键盘控制方向，碰到食物会增长
- `SnakeGame`：食物和贪吃蛇的游戏显示配置和游戏状态

有了上述事先准备的数据结构，我们就可以实现贪吃蛇游戏的主体逻辑了。

```

1 pub fn main() -> i32 {
2 // 创建具有virtio-gpu设备显示内存虚地址的Display结构
3 let mut disp = Display::new(Size::new(1280, 800), Point::new(0, 0));
4 // 初始化游戏显示元素的配置：红色的蛇、黄色的食物，方格大小为20个像素点
5 let mut game = SnakeGame::<20, Rgb888>::new(1280, 800, 20, 20, Rgb888::RED, ↴
6 Rgb888::YELLOW, 50);
7 // 清屏
8 let _ = disp.clear(Rgb888::BLACK).unwrap();
9 // 启动游戏循环
10 loop {
11 if key_pressed() {
12 let c = getchar();
13 match c {
14 LF => break,
15 CR => break,
16 // 调整蛇行进方向
17 b'w' => game.set_direction(Direction::Up),
18 b's' => game.set_direction(Direction::Down),
19 b'a' => game.set_direction(Direction::Left),
20 b'd' => game.set_direction(Direction::Right),
21 _ => (),
22 }
23 }
24 //绘制游戏界面
25 let _ = disp.clear(Rgb888::BLACK).unwrap();
26 game.draw(&mut disp);
27 //暂停一小会
28 sleep(10);
29 }
30 }
```

(下页继续)

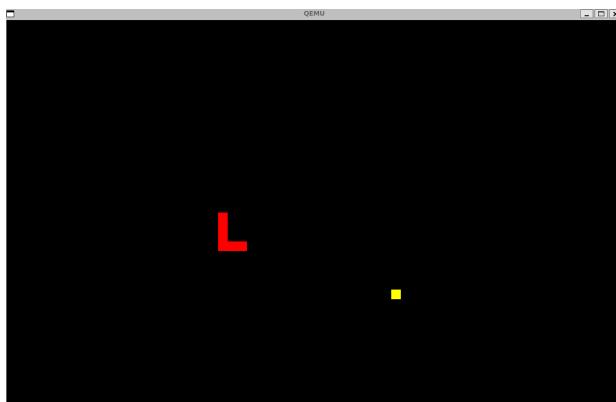
(续上页)

```
29 0
30 }
```

这里看到，为了判断通过串口输入的用户是否按键，我们扩展了一个系统调用 `sys_key_pressed`：

```
1 // os/src/syscall/input.rs
2 pub fn sys_key_pressed() -> isize {
3 let res = !UART.read_buffer_is_empty();
4 if res {
5 1
6 } else {
7 0
8 }
9 }
```

这样，我们结合串口和 `virtio-gpu` 两种外设，并充分利用已有的 Rust 库，设计实现了一个贪吃蛇小游戏（如下图所示）。至此，基于侏罗猎龙操作系统的图形应用开发任务就完成了。



## 10.8 练习

### 10.8.1 课后练习

#### 编程题

1. \*\*\* 在 Linux 的字符（命令行）模式下，编写贪吃蛇小游戏应用程序。

#### 问答题

1. \* 字符设备的特点是什么？
2. \* 块设备的特点是什么？
3. \* 网络设备的特点是什么？
4. \* 阻塞 I/O、非阻塞 I/O、多路复用 I/O、信号驱动 I/O、异步 I/O 这几种 I/O 方式的特点和区别是？
5. \* IO 数据传输有哪几种？各自的特征是什么？
6. \* 描述磁盘 I/O 操作时间组成。其中的瓶颈是哪部分？

7. \*\*RISC-V 中的异常, 中断的区别是啥? 有几类中断? 每类中断有哪些具体的常见中断实例? PLIC/CLINT 的具体功能是啥? 中断可否从 M 态响应委托给 S 态响应? S 态响应可否委托给 U 态响应? 与中断相关的 M 态/S 态寄存器有哪些, 这些寄存器的功能是啥? 外设产生一个中断后, PLIC/CPU/OS 如何协同进行响应处理的?
8. \*\*是否可以把设备抽象为文件? 如果可以, 那用户进程对设备发出 IO 控制命令, 如何通过系统调用实现?
9. \*\*GPU 是外设吗? GPU 与 CPU 交互和数据传输的方式是什么? (需要查看一下相关 GPU 工作过程的信息)

## 10.8.2 实验练习

实验练习包括实践作业和问答作业两部分。本次难度: 中

### 实践作业

#### 支持图形显示的应用

本章虽然讲述了 `virtio-gpu` 设备驱动, 且可以直接进行图形显示, 但这个设备驱动并没有加入到操作系统中, 使得应用程序无法进行图形显示。`lab8` 的练习要求操作系统支持有彩色图形显示的应用, 使得我们可以从单调的字符交互界跳入到多彩的图形界面中。

#### 实验要求

- 实现分支: `ch9-lab`
- 实验目录要求不变
- 在裸机上让操作系统支持“贪吃蛇”游戏应用

需要在操作系统中加入 `virtio-gpu` 设备驱动程序; 需要实现设备文件 `/dev/fb0` 和相关操作, 用于应用访问显存。

可以正确执行“贪吃蛇”游戏应用。

### 问答作业

1. 通过阅读和运行试验等分析, 你认为在目前的操作系统中, 如果运行在用户态, 可以响应哪些中断? 如果运行在内核态, 可以响应哪些中断? 请简要描述分析经过。
2. 对于串口驱动程序, 在 RustSBI 中有具体的实现, 请问它与本章讲的串口驱动有何异同之处?
3. 对于目前操作系统中的 `virtio-blk` 设备驱动程序, 存在哪些可以改进的地方来提升性能?

## 实验练习的提交报告要求

- 简单总结本次实验你编程的内容。(控制在 5 行以内, 不要贴代码)
- 完成问答问题。
- (optional) 你对本次实验设计及难度/工作量的看法, 以及有哪些需要改进的地方, 欢迎畅所欲言。

## 10.9 练习参考答案

### 10.9.1 课后练习

编程题

问答题



## 综合练习

---

- 本节难度：对 OS 的全局理解要求较高。
- 实验分为基础作业实验和扩展作业实验（二选一）。

### 11.1 基础作业

在保持 syscall 数量和基本含义不变的情况下，通过对 OS 内部的改进，提升 OS 的质量。

同学们通过独立完成前面的实验后，应该对于操作系统的根本核心机制有了较好的了解，并知道如何形成一个有进程 / 地址空间 / 文件核心概念的基本功能 OS。但同学自制的 OS 可能还需进一步完善，才能在功能 / 性能 / 可靠性上进一步进化，以使得测试用例的正常运行。

综合实验的目的是希望同学们能够在完成前面实验的基础上，站在全局视角，分析之前的测试用例（没增加新的 syscall 访问，只是更加全面和深入地测试操作系统的质量和能力）的运行情况，分析和理解自己写的 OS 是否能比较好地满足应用需求？如果不满足应用需求，或者应用导致系统缓慢甚至崩溃，那原因出在哪里？应该如何修改？修改后的 OS 是否更加完善还是缺陷更多？

#### 11.1.1 实验要求

- 实现分支：final。
- 运行 [final 测例](#)，观察并分析部分测试用例对 OS 造成的影响。
- 结合你学到的操作系统课程知识和你的操作系统具体实践情况，分析你写的 OS 对测试用例中的 app 支持不好的原因，比如：为何没运行通过，为何死在某处了，为何系统崩溃，为何系统非常缓慢。分析可能的解决方法。（2~4 个，4 个合理的分析就可得到满分，超过 4 个不额外得分）。
- 更进一步完成编程实现，使其可以通过一些原本 fail 的测例。（1~2 个，超过 2 个不额外得分）。

### 11.1.2 报告要求

- 对于失败测例的现象观察，原因分析，并提出可能的解决思路 (2~4 个)。
- 编程实现的具体内容，不需要贴出全部代码，重要的是描述清楚自己的思路和做法 (1~2 个)。
- (optional) 你对本次实验的其他看法。

### 11.1.3 其他说明

- 注意：编程实现部分的底线是 OS 不崩溃，如果你解决不了问题，就解决出问题的进程。可以通过简单杀死进程方式保证 OS 不会死掉。比如不支持某种 corner case，就把触发该 case 的进程杀掉，如果是这样，至少完成两个。会根据报告综合给分。
- 有些测例属于非法程序，比如申请过量内存，对于这些程序，杀死进程其实就是正确的做法。参考：[OOM killer](#)。
- 不一定所有的测例都会导致自己实现的 OS 崩溃，与语言和实现都有关系，选择出问题的测例分析即可。对于没有出错的测例，可以选择性分析自己的 OS 是如何预防这些“刁钻”测例的。对于测例没有测到的，也可以分析自己觉得安全 / 高效的实现，只要分析合理及给分。
- 鼓励针对有趣的测例进行分析！开放思考！

---

注解：

1. 本次实验的分值与之前 lab 相同，截至是时间为 15 周周末，基础实验属于必做实验 (除非你选择做扩展作业来代替基础作业)。
  2. 在测例中有简明描述：想测试 OS 哪方面的质量。同学请量力而行，推荐不要超过上述上限。咱们不要卷。
  3. 对于有特殊要求的同学 (比如你觉得上面的实验太难)，可单独找助教或老师说出你感兴趣或力所能及的实验内容，得到老师和助教同意后，做你提出的实验。
  4. 欢迎同学们贡献新测例，有意义测例经过助教检查可以写进报告充当工作量，欢迎打掉框架代码 OS，也欢迎打掉其他同学的 OS。
- 

### 11.1.4 实验检查

- 实验目录要求

目录要求不变 (参考 lab1 目录或者示例代码目录结构)。同样在 os 目录下 `make run` 之后可以正确加载用户程序并执行。

加载的用户测例位置：`../user/build/elf`。

- 检查

可以正确 `make run` 执行，可以正确执行目标用户测例，并得到预期输出 (详见测例注释)。

### 11.1.5 问答作业

无

## 11.2 拓展作业 (可选)

给部分同学不同的 OS 设计与实现的实验选择。扩展作业选项 (1-14) 基于之前的 OS 来实现，扩展作业选项 (15) 是发现目标内核 (ucore / rcore os) 漏洞。可选内容 (有大致难度估计) 如下：

1. 实现多核支持，设计多核相关测试用例，并通过已有和新的测试用例 (难度：8)
  - 某学长的有 bug 的 rcore tutorial 参考实现 <https://github.com/xy-plus/rCore-Tutorial-v3/tree/ch7>
2. 实现 slab 内存分配算法，通过相关测试用例 (难度：7)
  - <https://github.com/tokio-rs/slab>
3. 实现新的调度算法，如 CFS、BFS 等，通过相关测试用例 (难度：7)
  - [https://en.wikipedia.org/wiki/Completely\\_Fair\\_Scheduler](https://en.wikipedia.org/wiki/Completely_Fair_Scheduler)
  - <https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html>
4. 实现某种 IO buffer 缓存替换算法，如 2Q, LRU-K, LIRS 等，通过相关测试用例 (难度：6)
  - LIRS: <http://web.cse.ohio-state.edu/~zhang.574/lirs-sigmetrics-02.html>
  - 2Q: <https://nyuscholars.nyu.edu/en/publications/2q-a-low-overhead-high-performance-buffer-replacement-algorithm>
  - LRU-K: <https://dl.acm.org/doi/10.1145/170036.170081>
5. 实现某种页替换算法，如 Clock, 二次机会算法等，通过相关测试用例 (难度：6)
6. 实现支持日志机制的可靠文件系统，可参考 OSTEP 教材中对日志文件系统的描述 (难度：7)
7. 支持 virtio disk 的中断机制，提高 IO 性能 (难度：4)
  - chapter8 <https://github.com/rcore-os/rCore-Tutorial-Book-v3/tree/ch8>
  - <https://github.com/rcore-os/virtio-drivers>
  - <https://github.com/belowthetree/TisuOS>
8. 支持 virtio framebuffer / 键盘/鼠标处理，给出 demo(推荐类似 pong 的 graphic game) 的测试用例 (难度：6)
  - code: <https://github.com/sgmarz/osblog/tree/pong>
  - code: <https://github.com/belowthetree/TisuOS>
  - tutorial doc: Talking with our new Operating System by Handling Input Events and Devices
  - tutorial doc: Getting Graphical Output from our Custom RISC-V Operating System in Rust
  - tutorial doc: Writing Pong Game in Rust for my OS Written in Rust
9. 支持 virtio NIC，给出测试用例 (难度：7)
  - <https://github.com/rcore-os/virtio-drivers>
10. 支持 virtio fs or 其他 virtio 虚拟外设，通过测试用例 (难度：5)
  - <https://docs.oasis-open.org/virtio/virtio/v1.1/csprd01/virtio-v1.1-csprd01.html>
11. 支持 testsuits for kernel 中 15 个以上的 syscall，通过相关测试用例 (难度：6)

- 大部分与我们实验涉及的 syscall 类似
  - <https://gitee.com/oscomp/testsuits-for-oskernel#testsuits-for-os-kernel>
12. 支持新文件系统，比如 fat32 或 ext2 等，通过相关测试用例 (难度: 7)
- <https://github.com/rafalh/rust-fatfs>
  - <https://github.com/pi-pi3/ext2-rs>
13. 支持物理硬件 (如全志哪吒开发板，K210 开发板等)。(难度: 7)
- 可找老师要物理硬件开发板和相关开发资料
14. 支持其他处理器 (如鲲鹏 ARM64、x64 架构等)。(难度: 7)
- 可基于 QEMU 来开发
  - 可找老师要基于其他处理器的物理硬件开发板 (如树莓派等) 和相关开发资料
15. 对 fork/exec/spawn 等进行扩展，并改进 shell 程序，实现 “!” 这种经典的管道机制。(难度: 4)
- 参考 rcore tutorial 文档中 chapter7 中内容
16. 向实验用操作系统发起 fuzzing 攻击 (难度: 6)
- 其实助教或老师写出的 OS kernel 也是漏洞百出，不堪一击。我们缺少的仅仅是一个可以方便发现 bug 的工具。也许同学们能写出或改造出一个 os kernel fuzzing 工具来发现并 crash 它/它们。下面的仅仅是参考，应该还不能直接用，也许能给你一些启发。
  - [gustave fuzzer for os kernel tutorial](#)
  - [gustave fuzzer project](#)
  - [paper: GUSTAVE: Fuzzing OS kernels like simple applications](#)
17. 学生自己的想法，但需要告知老师或助教，并得到同意。

---

**注解:**

1. 支持 1~3 人组队，如果确定并组队完成，请在截止期前通过电子邮件告知助教。成员的具体得分可能会通过与老师和助教的当面交流综合判断给出。尽量减少划水与抱大腿。
  2. 根据老师和助教的评价，可获得额外得分，但不会超过实验的满分 (30 分)。也就是如果前面实验有失分，可以通过一个简单扩展把这部分分数拿回来。
- 

### 11.2.1 其他说明

- 不能抄袭其他上课同学的作业，查出后，**所有实验成绩清零**。
- final 扩展作业可代替 final 基础作业。拓展实验给分要求会远低于大实验，简单的拓展也可以的得到较高的评价。在完成代码的同时，也要求写出有关设计思路，问题及解决方法，实验分析等内容的实验报告。
- 完成之前的编程作业也可得满分。这个扩展作业不是必须要做的，是给有兴趣但不想选择大实验的同学一个选择。

### 11.2.2 实验检查

完成后当面交流。

### 11.2.3 问答作业

无



## 附录 A: Rust 系统编程入门

---

### 12.1 Rust 编程相关

- OS Tutorial Summer of Code 2020: Rust 系统编程入门指导
- Stanford 新开的一门很值得学习的 Rust 入门课程
- 一份简单的 Rust 入门介绍
- 《RustOS Guide》中的 Rust 介绍部分
- 一份简单的 Rust 宏编程新手指南

### 12.2 Rust 系统编程 pattern

- Arc<Mutex<\_> in Rust
- Understanding Closures in Rust
- Closures in Rust



---

## 附录 B：常见工具的使用方法

---

### 13.1 调试工具的使用

#### 13.1.1 下载或编译 GDB

可以在[实验环境配置](#)中下载编译好的二进制（版本为 8.3.0，由于包括整个哦那工具链，解压后大小约为 1G），也可以编译最新版本（仅 gdb，大小约为 300M）

```
wget https://github.com/riscv/riscv-binutils-gdb/archive/refs/heads/riscv-binutils-2.36.1.zip
unzip riscv-binutils-2.36.1.zip
mkdir build
cd build
../riscv-binutils-2.36.1/configure --target=riscv64-unknown-elf
make
```

如果是编译好的二进制，gdb 在 ./bin/riscv64-unknown-elf-gdb 中。如果是自己编译的最新版本，gdb 在 build/bin/gdb 中。你可以移动到一个你喜欢的位置。

首先修改 Makefile，下以 ch1 分支的为例：

1. 第三行 release 改为 debug
2. 第 46 行去掉 --release
3. 第 66 行的 qemu 的选项中增加 -s -S

这时，运行 make run 应该会停在系统开始前，等待 gdb 客户端连接。

### 13.1.2 在命令行中直接使用 gdb

```
启动gdb, 传入二进制文件作为参数。
记得修改路径
./bin/riscv64-unknown-elf-gdb /Volumes/Code/rCore-Tutorial-v3/os/target/riscv64gc-
↳unknown-none-elf/debug/os
导入源码路径
(gdb) directory /Volumes/Code/rCore-Tutorial-v3/os/
Source directories searched: /Volumes/Code/rCore-Tutorial-v3/os:$cdir:$cwd
连接到qemu中的gdb-server
(gdb) target remote localhost:1234
Remote debugging using localhost:1234
0x00000000000001000 in ?? ()
现在可以开始调试了, 下面给出一些示例指令:
(gdb) b rust_main
Breakpoint 1 at 0x802005aa: file /Volumes/Code/rCore-Tutorial-v3/os/src/main.rs, line
↳36.
(gdb) continue
Continuing.

Breakpoint 1, os::rust_main () at /Volumes/Code/rCore-Tutorial-v3/os/src/main.rs:36
36 clear_bss();
(gdb) l
31 fn sbss();
32 fn ebss();
33 fn boot_stack();
34 fn boot_stack_top();
35 }
36 clear_bss();
```

### 13.1.3 在 IDE 中直接使用 gdb

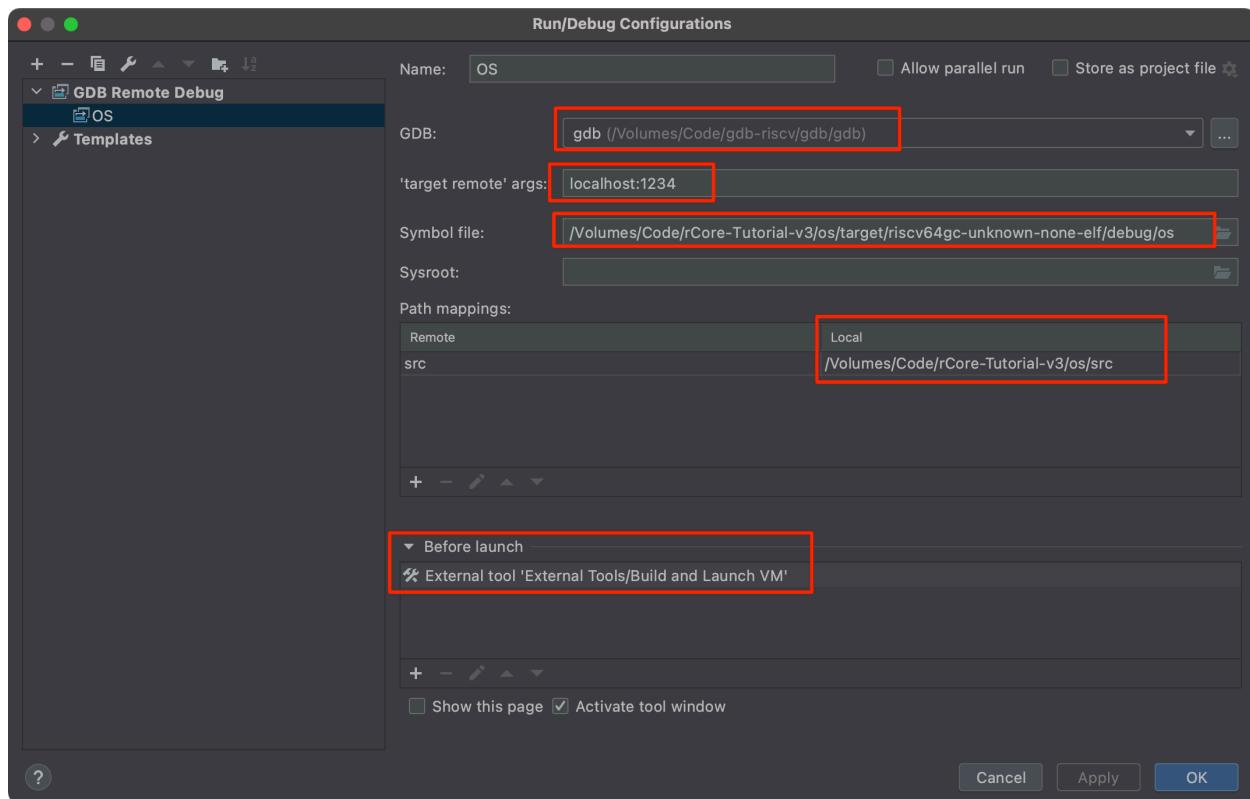
下面以 [CLion](<https://www.jetbrains.com/clion/>) 中 [Rust 插件](<https://plugins.jetbrains.com/plugin/8182-rust>) 为例。其他 IDE 的配置大同小异。

注意：上面提供的 GDB 二进制版本过低，需要使用自己编译的最新版本的 GDB。

1. 在 CLion 中打开项目 (os 文件夹)，选择 cargo project。
2. 在项目中新建一个 sh 文件，输入以下内容并给予可执行权限：

```
#!/usr/bin/env bash
killall qemu-system-riscv64 #_
↳由于无法在debug结束时关闭虚拟机，我们在debug开始时关闭上一次开启的虚拟机。
nohup bash -c "make run > run.log 2>&1" & # 后台启动qemu
echo "Done!"
```

3. 在右上角点击 Edit Configurations，新增一个 GDB Remote Debug，并如图配置：



第 1 个红框中选择你的自己编译的 gdb 路径第 3, 4 个红框中根据你的代码路径做适当修改第 5 个红框中，点击下面加号，选择‘External Tools’，并选择上面新建的‘sh’脚本。

## 13.2 分析可执行文件

对于 Rust 编译器生成的执行程序，可通过各种有效工具进行分析。如果掌握了对这些工具的使用，那么在后续的开发工作中，对碰到的各种奇怪问题就进行灵活处理和解决了。我们以 Rust 编译生成的一个简单的“Hello, world”应用执行程序为分析对象，看看如何进行分析。

让我们先来通过 file 工具看看最终生成的可执行文件的格式：

```
$ cargo new os
$ cd os; cargo build
Compiling os v0.1.0 (/tmp/os)
Finished dev [unoptimized + debuginfo] target(s) in 0.26s

$ file target/debug/os
target/debug/os: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2,
$
```

从中可以看出可执行文件的格式为 **可执行和链接格式** (Executable and Linkable Format, ELF)，硬件平台是 x86-64。在 ELF 文件中，除了程序必要的代码、数据段（它们本身都只是一些二进制的数据）之外，还有一些 **元数据** (Metadata) 描述这些段在地址空间中的位置和在文件中的位置以及一些权限控制信息，这些元数据只能放在代码、数据段的外面。

### 13.2.1 rust-readobj

我们可以通过二进制工具 `rust-readobj` 来看看 ELF 文件中究竟包含什么内容，输入命令：

```
$ rust-readobj -all target/debug/os
```

首先可以看到一个 ELF header，它位于 ELF 文件的开头：

```

1 File: target/debug/os
2 Format: elf64-x86-64
3 Arch: x86_64
4 AddressSize: 64bit
5 LoadName:
6 ElfHeader {
7 Ident {
8 Magic: (7F 45 4C 46)
9 Class: 64-bit (0x2)
10 DataEncoding: LittleEndian (0x1)
11 FileVersion: 1
12 OS/ABI: SystemV (0x0)
13 ABIVersion: 0
14 Unused: (00 00 00 00 00 00 00 00)
15 }
16 Type: SharedObject (0x3)
17 Machine: EM_X86_64 (0x3E)
18 Version: 1
19 Entry: 0x5070
20 ProgramHeaderOffset: 0x40
21 SectionHeaderOffset: 0x32D8D0
22 Flags [(0x0)
23]
24 HeaderSize: 64
25 ProgramHeaderEntrySize: 56
26 ProgramHeaderCount: 12
27 SectionHeaderEntrySize: 64
28 SectionHeaderCount: 42
29 StringTableSectionIndex: 41
30 }
31

```

- 第 8 行是一个称之为 **魔数 (Magic)** 独特的常数，存放在 ELF header 的一个固定位置。当加载器将 ELF 文件加载到内存之前，通常会查看该位置的值是否正确，来快速确认被加载的文件是不是一个 ELF。
- 第 19 行给出了可执行文件的入口点为 `0x5070`。
- 从 20-21 行中，我们可以知道除了 ELF header 之外，还有另外两种不同的 header，分别称为 **program header** 和 **section header**，它们都有多个。ELF header 中给出了其他两种 header 的大小、在文件中的位置以及数目。
- 从 24-27 行中，可以看到有 12 个不同的 **program header**，它们从文件的 `0x40` 字节偏移处开始，每个 `56` 字节；有 64 个 **section header**，它们从文件的 `0x2D8D0` 字节偏移处开始，每个 `64` 字节；

有多个不同的 **section header**，下面是个具体的例子：

```

.....
Section {
 Index: 14
 Name: .text (157)
 Type: SHT_PROGBITS (0x1)

```

(下页继续)

(续上页)

```

Flags [(0x6)
 SHF_ALLOC (0x2)
 SHF_EXECINSTR (0x4)
]
Address: 0x5070
Offset: 0x5070
Size: 208067
Link: 0
Info: 0
AddressAlignment: 16
EntrySize: 0
}

```

每个 section header 则描述一个段的元数据。

其中，我们看到了代码段 .text 需要被加载到地址 0x5070，大小 208067 字节。它们分别由元数据的字段 Offset、Size 和 Address 给出。

我们还能够看到程序中的符号表：

```

Symbol {
 Name: _start (37994)
 Value: 0x5070
 Size: 47
 Binding: Global (0x1)
 Type: Function (0x2)
 Other: 0
 Section: .text (0xE)
}
Symbol {
 Name: main (38021)
 Value: 0x51A0
 Size: 47
 Binding: Global (0x1)
 Type: Function (0x2)
 Other: 0
 Section: .text (0xE)
}

```

里面包括了我们写的 main 函数的地址以及用户态执行环境的起始地址 \_start 函数的地址。

因此，从 ELF header 中可以看出，ELF 中的内容按顺序应该是：

- ELF header
- 若干个 program header
- 程序各个段的实际数据
- 若干的 section header

### 13.2.2 rust-objdump

如果想了解正常的 ELF 文件的具体指令内容, 可以通过 `rust-objdump` 工具反汇编 ELF 文件得到:

```
$ rust-objdump -all target/debug/os
```

具体结果如下:

```
505b: e9 c0 ff ff ff jmp 0x5020 <.plt>

Disassembly of section .plt.got:
0000000000005060 <.plt.got>:
5060: ff 25 5a 3f 04 00 jmpq *278362(%rip) # 48fc0 <_GLOBAL_OFFSET_
→TABLE_+0x628>
5066: 66 90 nop

Disassembly of section .text:
0000000000005070 <_start>:
5070: f3 0f 1e fa endbr64
5074: 31 ed
5076: 49 89 d1
5079: 5e
507a: 48 89 e2
507d: 48 83 e4 f0
5081: 50
5082: 54
5083: 4c 8d 05 86 2c 03 00
→fini>
508a: 48 8d 0d 0f 2c 03 00
→csu_init>
5091: 48 8d 3d 08 01 00 00
5098: ff 15 d2 3b 04 00
→TABLE_+0x2d8>
.....
00000000000051a0 <main>:
51a0: 48 83 ec 18
51a4: 8a 05 db 7a 03 00
→debug_gdb_scripts_section_>
51aa: 48 63 cf
51ad: 48 8d 3d ac ff ff ff
→ZN2os4main17h717a6a6e05a70248E>
51b4: 48 89 74 24 10
51b9: 48 89 ce
51bc: 48 8b 54 24 10
51c1: 88 44 24 0f
51c5: e8 f6 00 00 00
→start17hc258028f546a93a1E>
51ca: 48 83 c4 18
51ce: c3
51cf: 90
.....
```

从上面的反汇编结果, 我们可以看到用户态执行环境的入口函数 `_start` 以及应用程序的主函数 `main` 的地址和具体汇编代码内容。

### 13.2.3 rust-objcopy

当前的 ELF 执行程序有许多与执行无直接关系的信息（如调试信息等），可以通过 `rust-objcopy` 工具来清除。

```
$ rust-objcopy --strip-all target/debug/os target/debug/os.bin
$ ls -l target/debug/os*
-rwxrwxr-x 2 chyyuu chyyuu 3334992 1月 19 22:26 target/debug/os
-rwxrwxr-x 1 chyyuu chyyuu 297200 1月 19 22:59 target/debug/os.bin

$./target/debug/os.bin
Hello, world!
```

可以看到，经过处理的 ELF 文件 `os.bin` 在文件长度上大大减少了，但也能正常执行。

另外，当将程序加载到内存的时候，对于每个 `program header` 所指向的区域，我们需要将对应的数据从文件复制到内存中。这就需要解析 ELF 的元数据才能知道数据在文件中的位置以及即将被加载到内存中的位置。但如果不需要从 ELF 中解析元数据就知道程序的内存布局（这个内存布局是我们按照需求自己指定的），我们可以手动完成加载任务。

具体的做法是利用 `rust-objcopy` 工具删除掉 ELF 文件中的所有 `header` 只保留各个段的实际数据得到一个没有任何符号的纯二进制镜像文件：

```
$ rust-objcopy --strip-all target/debug/os -O binary target/debug/os.bin
```

这样就生成了一个没有任何符号的纯二进制镜像文件。由于缺少了必要的元数据，我们的 `file` 工具也没有办法对它完成解析了。而后，我们可直接将这个二进制镜像文件手动载入到内存中合适位置即可。

## 13.3 qemu 平台上可执行文件和二进制镜像的生成流程

### 13.3.1 make & Makefile

首先我们还原一下可执行文件和二进制镜像的生成流程：

```
os/Makefile
TARGET := riscv64gc-unknown-none-elf
MODE := release
KERNEL_ELF := target/$(TARGET)/$(MODE)/os
KERNEL_BIN := $(KERNEL_ELF).bin

$(KERNEL_BIN): kernel
 @$(OBJCOPY) $(KERNEL_ELF) --strip-all -O binary $@

kernel:
 @cargo build --release
```

这里可以看出 `KERNEL_ELF` 保存最终可执行文件 `os` 的路径，而 `KERNEL_BIN` 保存只保留各个段数据的二进制镜像文件 `os.bin` 的路径。目标 `kernel` 直接通过 `cargo build` 以 `release` 模式最终可执行文件，目标 `KERNEL_BIN` 依赖于目标 `kernel`，将可执行文件通过 `rust-objcopy` 工具加上适当的配置移除所有的 `header` 和符号得到二进制镜像。

我们可以通过 `make run` 直接在 `qemu` 上运行我们的应用程序，`qemu` 是一个虚拟机，它完整的模拟了一整套硬件平台，就像是一台真正的计算机一样，我们来看运行 `qemu` 的具体命令：

```

1 KERNEL_ENTRY_PA := 0x80020000
2
3 BOARD ?= qemu
4 SBI ?= rustsbi
5 BOOTLOADER := ./bootloader/${SBI}-$(BOARD).bin
6
7 run: run-inner
8
9 run-inner: build
10 ifeq ($(BOARD), qemu)
11 @qemu-system-riscv64 \
12 -machine virt \
13 -nographic \
14 -bios $(BOOTLOADER) \
15 -device loader,file=$(KERNEL_BIN),addr=$(KERNEL_ENTRY_PA)
16 else
17 @cp $(BOOTLOADER) $(BOOTLOADER).copy
18 @dd if=$(KERNEL_BIN) of=$(BOOTLOADER).copy bs=128K seek=1
19 @mv $(BOOTLOADER).copy $(KERNEL_BIN)
20 @sudo chmod 777 $(K210-SERIALPORT)
21 python3 $(K210-BURNER) -p $(K210-SERIALPORT) -b 1500000 $(KERNEL_BIN)
22 miniterm --eol LF --dtr 0 --rts 0 --filter direct $(K210-SERIALPORT) 115200
23 endif

```

### 13.3.2 qemu

注意其中高亮部分给出了传给 qemu 的参数。

- `-machine` 告诉 qemu 使用预设的硬件配置。在整个项目中我们将一直沿用该配置。
- `-bios` 告诉 qemu 使用我们放在 `bootloader` 目录下的预编译版本作为 bootloader。
- `-device` 则告诉 qemu 将二进制镜像加载到内存指定的位置。

可以先输入 `Ctrl+A`，再输入 `X` 来退出 qemu 终端。

**警告: FIXME: 使用 GDB 跟踪 qemu 的运行状态**

## 13.4 k210 平台上可执行文件和二进制镜像的生成流程

对于 k210 平台来说，只需要将 maix 系列开发板通过数据线连接到 PC，然后 `make run BOARD=k210` 即可。从 Makefile 中来看：

```

1 K210-SERIALPORT = /dev/ttyUSB0
2 K210-BURNER = ./tools/kflash.py
3
4 run-inner: build
5 ifeq ($(BOARD), qemu)
6 @qemu-system-riscv64 \
7 -machine virt \
8 -nographic \
9 -bios $(BOOTLOADER) \
10 -device loader,file=$(KERNEL_BIN),addr=$(KERNEL_ENTRY_PA)

```

(下页继续)

(续上页)

```

11 else
12 @cp ${BOOTLOADER} ${BOOTLOADER}.copy
13 @dd if=${KERNEL_BIN} of=${BOOTLOADER}.copy bs=128K seek=1
14 @mv ${BOOTLOADER}.copy ${KERNEL_BIN}
15 @sudo chmod 777 ${K210-SERIALPORT}
16 python3 ${K210-BURNER} -p ${K210-SERIALPORT} -b 1500000 ${KERNEL_BIN}
17 miniterm --eol LF --dtr 0 --rts 0 --filter direct ${K210-SERIALPORT} 115200
18 endif

```

在构建目标 run-inner 的时候，根据平台 BOARD 的不同，启动运行的指令也不同。当我们传入命令行参数 BOARD=k210 时，就会进入下面的分支。

- 第 13 行我们使用 dd 工具将 bootloader 和二进制镜像拼接到一起，这是因为 k210 平台的写入工具每次只支持写入一个文件，所以我们只能将二者合并到一起一并写入 k210 的内存上。这样的参数设置可以保证 bootloader 在合并后文件的开头，而二进制镜像在文件偏移量 0x20000 的位置处。有兴趣的同学可以输入命令 man dd 查看关于工具 dd 的更多信息。
- 第 16 行我们使用烧写工具 K210-BURNER 将合并后的镜像烧写到 k210 开发板的内存的 0x80000000 地址上。参数 K210-SERIALPORT 表示当前 OS 识别到的 k210 开发板的串口设备名。在 Ubuntu 平台上一般为 /dev/ttyUSB0。
- 第 17 行我们打开串口终端和 k210 开发板进行通信，可以通过键盘向 k210 开发板发送字符并在屏幕上看到 k210 开发板的字符输出。

可以输入 **Ctrl+]** 退出 miniterm。

## 13.5 其他工具和文件格式说明的参考

- 链接脚本 (Linker Scripts) 语法和规则解析 (翻译自官方手册)
- Make 命令教程



## 附录 C: 深入机器模式: RustSBI

---

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 项目发起于鹏城实验室的“rCore 代码之夏-2020”活动, 它是完全由 Rust 语言开发的 SBI 实现。现在它能够在支持的 RISC-V 设备上运行 rCore 教程和其它操作系统内核。

RustSBI 项目的目标是, 制作一个从固件启动的最小 Rust 语言 SBI 实现, 为可能的复杂实现提供参考和支持。RustSBI 也可以作为一个库使用, 帮助更多的 SBI 开发者适配自己的平台, 以支持更多处理器核和片上系统。

当前项目实现源码: <https://github.com/rustsbi/rustsbi>



## 附录 D: RISC-V 相关信息

### 15.1 RISCV 汇编相关

#### 15.1.1 如何生成汇编代码

```
通常办法，生成的汇编代码有比较冗余的信息
生成缺省debug模式的汇编
$cargo rustc -- --emit asm
$ls target/debug/deps/<crate_name>-<hash>.s
生成release模式的汇编
$cargo rustc --release -- --emit asm
$ls target/release/deps/<crate_name>-<hash>.s
在rcore-tutorial-v3中的应用的汇编代码生成举例
$cd user
$cargo rustc --release --bin hello_world -- --emit asm
$find ./target -name "hello_world*.s"

生成更加干净的汇编代码
基于 cargo-show-asm(https://github.com/pacak/cargo-show-asm) 的办法
如果没用安装这个cargo asm子命令，就安装它
$cargo install cargo-show-asm
在rcore-tutorial-v3中的应用的汇编代码生成举例
$cd user
$cargo asm --release --bin hello_world

Compiling user_lib v0.1.0 (/home/chyyuu/the.codes/rCore-Tutorial-v3/user)
Finished release [optimized + debuginfo] target(s) in 0.10s

.section .text.main,"ax",@progbits
.globl main
.p2align 1
.type main,@function
main:
```

(下页继续)

(续上页)

```
.cfi_sections .debug_frame
.cfi_startproc
addi sp, sp, -64
.cfi_def_cfa_offset 64

sd ra, 56(sp)
sd s0, 48(sp)
.cfi_offset ra, -8
.cfi_offset s0, -16
addi s0, sp, 64
.cfi_def_cfa s0, 0

auipc a0, %pcrel_hi(.L__unnamed_1)
addi a0, a0, %pcrel_lo(.LBB0_1)

sd a0, -64(s0)
li a0, 1

sd a0, -56(s0)
sd zero, -48(s0)

auipc a0, %pcrel_hi(.L__unnamed_2)
addi a0, a0, %pcrel_lo(.LBB0_2)

sd a0, -32(s0)
sd zero, -24(s0)

addi a0, s0, -64

call user_lib::console::print
li a0, 0
ld ra, 56(sp)
ld s0, 48(sp)
addi sp, sp, 64
ret
```

### 15.1.2 参考信息

- RISC-V Assembly Programmer’s Manual
- RISC-V Low-level Test Suits
- CoreMark®-PRO comprehensive, advanced processor benchmark
- riscv-tests 的使用

## 15.2 RISCV 硬件相关

### 15.2.1 硬件信息

- Registers & ABI
- Interrupt
- ISA & Extensions
- Toolchain
- Control and Status Registers (CSRs)
- Accessing CSRs
- Assembler & Instructions

### 15.2.2 指令集规范

- User-Level ISA, Version 1.12
- 4 Supervisor-Level ISA, Version 1.12
- Vector Extension
- RISC-V Bitmanip Extension
- External Debug
- ISA Resources



## 附录 E：操作系统进一步介绍

---

### 16.1 openEuler 操作系统

国产操作系统始于九十年代，大多数基于 Fedora/CentOS/Debian/Ubuntu 进行二次开发，直到 2019 年，华为公司开源了 openEuler 操作系统，openEuler 是自主演进的根操作系统，不基于其他任何 OS 二次开发，这是与其他国产 OS 的主要差异。

openEuler 是一个开源、免费的 Linux 发行版平台，其致力于打造中国原生开源、可自主演进操作系统根社区。openEuler 前身是华为公司发展近 10 年的服务器操作系统 EulerOS，2019 年开源，更名为 openEuler，当前有多个国产 OSV 厂商基于 openEuler 发布商用版本。openEuler 最新定位是面向数字基础设施的开源操作系统，支持多样性计算，支持服务器、云计算、边缘计算、嵌入式等应用场景，支持 OT (Operational Technology) 领域应用及 OT 与 ICT 的融合。

openEuler 在具有通用的 Linux 系统架构，包括内存管理子系统、进程管理子系统、进程调度子系统、进程间通讯 (IPC)、文件系统、网络子系统、设备管理子系统和虚拟化与容器子系统等。同时，openEuler 又不同于其他通用操作系统，openEuler 从 OS 内核、可靠性、安全性和生态使能等方面做了特性增强。openEuler 的关键特性如下表所示：

| 名称          | 特性说明                                               | 网页链接                                                                                                                                                                      |
|-------------|----------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Stra-toVirt | 轻量级虚拟机引擎                                           | <a href="https://docs.openeuler.org/zh/docs/21.03/docs/StratoVirt/StratoVirtGuide.html">https://docs.openeuler.org/zh/docs/21.03/docs/StratoVirt/StratoVirtGuide.html</a> |
| iSula       | 轻量级容器引擎                                            | <a href="https://docs.openeuler.org/zh/docs/21.03/docs/Container/iSula容器引擎.html">https://docs.openeuler.org/zh/docs/21.03/docs/Container/iSula容器引擎.html</a>               |
| A-Tune      | AI 智能调优引擎                                          | <a href="https://docs.openeuler.org/zh/docs/21.03/docs/A-Tune/A-Tune.html">https://docs.openeuler.org/zh/docs/21.03/docs/A-Tune/A-Tune.html</a>                           |
| secGear     | 跨平台机密计算框架使                                         | <a href="https://docs.openeuler.org/zh/docs/21.03/docs/secGear/secGear.html">https://docs.openeuler.org/zh/docs/21.03/docs/secGear/secGear.html</a>                       |
| 可信计算        | 安全可信计算                                             | <a href="https://docs.openeuler.org/zh/docs/21.03/docs/Administration/可信计算.html">https://docs.openeuler.org/zh/docs/21.03/docs/Administration/可信计算.html</a>               |
| KAE         | 鲲鹏加速引擎 (Kunpeng Accelerator Engine)                | <a href="https://docs.openeuler.org/zh/docs/21.03/docs/Administration/使用 KAE 加速引擎.html">https://docs.openeuler.org/zh/docs/21.03/docs/Administration/使用 KAE 加速引擎.html</a> |
| MPAM        | Memory System Resource Partitioning and Monitoring | <a href="https://mp.weixin.qq.com/s/0TgrFjFtobmk-h1HwJskqq">https://mp.weixin.qq.com/s/0TgrFjFtobmk-h1HwJskqq</a>                                                         |
| 毕 F<br>JDK  | Huawei 开源 JDK                                      | <a href="https://gitee.com/openeuler/bishengjdk-8/wikis/Home">https://gitee.com/openeuler/bishengjdk-8/wikis/Home</a>                                                     |

## 附录 F: 类 Peterson 算法的局限性和内存顺序

---

### 17.1 参考文献

- Rust Atomics and Locks



## 术语中英文对照表

## 18.1 第一章：RV64 裸机应用

| 中文       | 英文                                | 出现章节          |
|----------|-----------------------------------|---------------|
| 执行环境     | Execution Environment             | 应用程序运行环境与平台支持 |
| 系统调用     | System Call                       | 应用程序运行环境与平台支持 |
| 指令集体体系结构 | ISA, Instruction Set Architecture | 应用程序运行环境与平台支持 |
| 抽象       | Abstraction                       | 应用程序运行环境与平台支持 |
| 平台       | Platform                          | 应用程序运行环境与平台支持 |
| 目标三元组    | Target Triplet                    | 应用程序运行环境与平台支持 |
| 裸机平台     | Bare-Metal                        | 应用程序运行环境与平台支持 |
| 交叉编译     | Cross Compile                     | 移除标准库依赖       |
| 物理地址     | Physical Address                  | 内核第一条指令(原理篇)  |
| 物理内存     | Physical Memory                   | 内核第一条指令(原理篇)  |
| 引导加载程序   | Bootloader                        | 内核第一条指令(原理篇)  |
| 控制流      | Control Flow                      | 为内核支持函数调用     |
| 函数调用     | Function Call                     | 为内核支持函数调用     |
| 源寄存器     | Source Register                   | 为内核支持函数调用     |
| 立即数      | Immediate                         | 为内核支持函数调用     |
| 目标寄存器    | Destination Register              | 为内核支持函数调用     |

下页继续

表 1 - 续上页

| 中文       | 英文                                  | 出现章节        |
|----------|-------------------------------------|-------------|
| 伪指令      | Pseudo Instruction                  | 为内核支持函数调用   |
| 上下文      | Context                             | 为内核支持函数调用   |
| 活动记录     | Activation Record                   | 为内核支持函数调用   |
| 保存/恢复    | Save/Restore                        | 为内核支持函数调用   |
| 被调用者保存   | Callee-Saved                        | 为内核支持函数调用   |
| 调用者保存    | Caller-Saved                        | 为内核支持函数调用   |
| 开场白      | Prologue                            | 为内核支持函数调用   |
| 收场白      | Epilogue                            | 为内核支持函数调用   |
| 调用规范     | Calling Convention                  | 为内核支持函数调用   |
| 栈/栈指针/栈帧 | Stack/Stack Pointer/Stackframe      | 为内核支持函数调用   |
| 后入先出     | LIFO, Last In First Out             | 为内核支持函数调用   |
| 段        | Section                             | 为内核支持函数调用   |
| 内存布局     | Memory Layout                       | 为内核支持函数调用   |
| 堆        | Heap                                | 为内核支持函数调用   |
| 编译器      | Compiler                            | 为内核支持函数调用   |
| 汇编器      | Assembler                           | 为内核支持函数调用   |
| 链接器      | Linker                              | 为内核支持函数调用   |
| 目标文件     | Object File                         | 为内核支持函数调用   |
| 链接脚本     | Linker Script                       | 为内核支持函数调用   |
| 可执行和链接格式 | ELF, Executable and Linkable Format | 手动加载、运行应用程序 |
| 元数据      | Metadata                            | 手动加载、运行应用程序 |
| 魔数       | Magic                               | 手动加载、运行应用程序 |
| 裸指针      | Raw Pointer                         | 手动加载、运行应用程序 |
| 解引用      | Dereference                         | 手动加载、运行应用程序 |

## 18.2 第二章：批处理系统

| 中文        | 英文                                    | 出现章节         |
|-----------|---------------------------------------|--------------|
| 批处理系统     | Batch System                          | 引言           |
| 特权级       | Privilege                             | 引言           |
| 监督模式执行环境  | SEE, Supervisor Execution Environment | RISC-V 特权级架构 |
| 异常控制流     | ECF, Exception Control Flow           | RISC-V 特权级架构 |
| 陷入        | Trap                                  | RISC-V 特权级架构 |
| 异常        | Exception                             | RISC-V 特权级架构 |
| 执行环境调用    | Environment Call                      | RISC-V 特权级架构 |
| 监督模式二进制接口 | SBI, Supervisor Binary Interface      | RISC-V 特权级架构 |
| 应用程序二进制接口 | ABI, Application Binary Interface     | RISC-V 特权级架构 |
| 控制状态寄存器   | CSR, Control and Status Register      | RISC-V 特权级架构 |
| 胖指针       | Fat Pointer                           | 实现应用程序       |
| 内部可变性     | Interior Mutability                   | 实现应用程序       |
| 指令缓存      | i-cache, Instruction Cache            | 实现批处理系统      |
| 数据缓存      | d-cache, Data Cache                   | 实现批处理系统      |
| 原子指令      | Atomic Instruction                    | 处理 Trap      |

## 18.3 第三章：多道程序与分时多任务

| 中文      | 英文                        | 出现章节          |
|---------|---------------------------|---------------|
| 多道程序    | Multiprogramming          | 引言            |
| 分时多任务系统 | Time-Sharing Multitasking | 引言            |
| 任务上下文   | Task Context              | 任务切换          |
| 输入/输出   | I/O, Input/Output         | 多道程序与协作式调度    |
| 任务控制块   | Task Control Block        | 多道程序与协作式调度    |
| 吞吐量     | Throughput                | 分时多任务系统与抢占式调度 |
| 后台应用    | Background Application    | 分时多任务系统与抢占式调度 |
| 交互式应用   | Interactive Application   | 分时多任务系统与抢占式调度 |
| 协作式调度   | Cooperative Scheduling    | 分时多任务系统与抢占式调度 |
| 时间片     | Time Slice                | 分时多任务系统与抢占式调度 |
| 公平性     | Fairness                  | 分时多任务系统与抢占式调度 |
| 时间片轮转算法 | RR, Round-Robin           | 分时多任务系统与抢占式调度 |
| 中断      | Interrupt                 | 分时多任务系统与抢占式调度 |
| 同步      | Synchronous               | 分时多任务系统与抢占式调度 |
| 异步      | Asynchronous              | 分时多任务系统与抢占式调度 |
| 并行      | Parallel                  | 分时多任务系统与抢占式调度 |
| 软件中断    | Software Interrupt        | 分时多任务系统与抢占式调度 |
| 时钟中断    | Timer Interrupt           | 分时多任务系统与抢占式调度 |
| 外部中断    | External Interrupt        | 分时多任务系统与抢占式调度 |
| 嵌套中断    | Nested Interrupt          | 分时多任务系统与抢占式调度 |
| 轮询      | Busy Loop                 | 分时多任务系统与抢占式调度 |

## 18.4 第四章：地址空间

| 中文       | 英文                                           | 出现章节               |
|----------|----------------------------------------------|--------------------|
| 幻象       | Illusion                                     | 引言                 |
| 时分复用     | TDM, Time-Division Multiplexing              | 引言                 |
| 地址空间     | Address Space                                | 地址空间               |
| 虚拟地址     | Virtual Address                              | 地址空间               |
| 内存管理单元   | MMU, Memory Management Unit                  | 地址空间               |
| 地址转换     | Address Translation                          | 地址空间               |
| 插槽       | Slot                                         | 地址空间               |
| 位图       | Bitmap                                       | 地址空间               |
| 内碎片      | Internal Fragment                            | 地址空间               |
| 外碎片      | External Fragment                            | 地址空间               |
| 页面       | Page                                         | 地址空间               |
| 虚拟页号     | VPN, Virtual Page Number                     | 地址空间               |
| 物理页号     | PPN, Physical Page Number                    | 地址空间               |
| 页表       | Page Table                                   | 地址空间               |
| 静态分配     | Static Allocation                            | Rust 中的动态内存分配      |
| 动态分配     | Dynamic Allocation                           | Rust 中的动态内存分配      |
| 智能指针     | Smart Pointer                                | Rust 中的动态内存分配      |
| 集合       | Collection                                   | Rust 中的动态内存分配      |
| 容器       | Container                                    | Rust 中的动态内存分配      |
| 借用检查     | Borrow Check                                 | Rust 中的动态内存分配      |
| 引用计数     | Reference Counting                           | Rust 中的动态内存分配      |
| 垃圾回收     | GC, Garbage Collection                       | Rust 中的动态内存分配      |
| 资源获取即初始化 | RAII, Resource Acquisition Is Initialization | Rust 中的动态内存分配      |
| 页内偏移     | Page Offset                                  | 实现 SV39 多级页表机制 (上) |
| 类型转换     | Type Conversion                              | 实现 SV39 多级页表机制 (上) |
| 字典树      | Trie                                         | 实现 SV39 多级页表机制 (上) |
| 多级页表     | Multi-Level Page Table                       | 实现 SV39 多级页表机制 (上) |
| 页索引      | Page Index                                   | 实现 SV39 多级页表机制 (上) |
| 大页       | Huge Page                                    | 实现 SV39 多级页表机制 (上) |
| 恒等映射     | Identical Mapping                            | 实现 SV39 多级页表机制 (下) |

下页继续

表 2 - 续上页

| 中文    | 英文                           | 出现章节               |
|-------|------------------------------|--------------------|
| 页表自映射 | Recursive Mapping            | 实现 SV39 多级页表机制 (下) |
| 跳板    | Trampoline                   | 内核与应用的地址空间         |
| 隔离    | Isolation                    | 内核与应用的地址空间         |
| 保护页面  | Guard Page                   | 内核与应用的地址空间         |
| 快表    | Translation Lookaside Buffer | 基于地址空间的分时多任务       |
| 熔断    | Meltdown                     | 基于地址空间的分时多任务       |



## 修改和构建本项目

---

1. 参考 [这里](#) 安装 Sphinx。
2. 切换到仓库目录下, `pip install -r requirements.txt` 安装各种 python 库依赖。
3. *reStructuredText* 基本语法 是 ReST 的一些基本语法, 也可以参考已完成的文档。
4. 修改之后, 在项目根目录下 `make clean && make html` 即可在 `build/html/index.html` 查看本地构建的主页。请注意在修改章节目录结构或者更新各种配置文件/python 脚本之后需要 `make clean` 一下, 不然可能无法正常更新。
5. 如想对项目做贡献的话, 直接提交 pull request 即可。

---

### 注解: 实时显示修改 rst 文件后的 html 文档的方法

1. `pip install autoloadd` 安装 Sphinx 自动加载插件。
  2. 在项目根目录下 `sphinx-autobuild source build/html` 即可在浏览器中访问 `http://127.0.0.1:8000/` 查看本地构建的主页。
- 

---

### 注解: 如何生成教程 pdf 电子版

注意: 经过尝试在 wsl 环境下无法生成 pdf, 请使用原生的 Ubuntu Desktop 或者虚拟机。

1. 首先 `sudo apt update`, 然后通过 `sudo apt install` 安装如下软件包: `latexmk texlive-latex-recommended texlive-latex-extra texlive-xetex fonts-freefont-otf texlive-fonts-recommended texlive-lang-chinese tex-gyre`。
2. 从 Node.js 官方网站下载最新版的 Node.js, 配置好环境变量并通过 `npm --version` 确认配置正确。然后通过 `npm install -g @mermaid-js/mermaid-cli` 安装 mermaid 命令行工具。
3. 确认 Python 环境配置正确, 也即 `make html` 可以正常生成 html。
4. 打上必要的补丁: 在根目录下执行 `git apply --reject scripts/latexpdf.patch`。
5. 构建: 在根目录下执行 `make latexpdf`, 过程中会有很多 latex 的警告, 但可以忽略。
6. 构建结束后, 电子版 pdf 可以在 `build/latex/rcore-tutorial-book-v3.pdf` 找到。

---

### 注解: 如何生成 epub 格式

1. 配置好 Sphinx Python 环境。
  2. `make epub` 构建 epub 格式输出, 产物可以在 `build epub rCore-Tutorial-Book-v3.epub` 中找到。
-

# CHAPTER 20

## reStructuredText 基本语法

**注解:** 下面是一个注记。

这里 给出了在 Sphinx 中外部链接的引入方法。注意，链接的名字和用一对尖括号包裹起来的链接地址之间必须有一个空格。链接最后的下划线和片段的后续内容之间也需要有一个空格。

接下来是一个文档内部引用的例子。比如，戳[实验环境配置](#)可以进入快速上手环节。

文档间互引用：比如在关于进程的退出部分：

```
可以使用 .. _process-exit: 记录文档的一个位置。
然后在文档中使用 :ref:`链接名 <process-exit>` 创建指向上述位置的一个链接。
```

**警告:** 下面是一个警告。

列表 1: 一段示例 Rust 代码

```
1 // 我们甚至可以插入一段 Rust 代码！
2 fn add(a: i32, b: i32) -> i32 { a + b }
```

下面继续我们的警告。

**注意:** Here is an attention.

**小心:** please be cautious!

**错误:** 下面是一个错误。

**危险:** it is dangerous!

---

**小技巧:** here is a tip

---

---

**重要:** this is important!

---

---

**提示:** this is a hint.

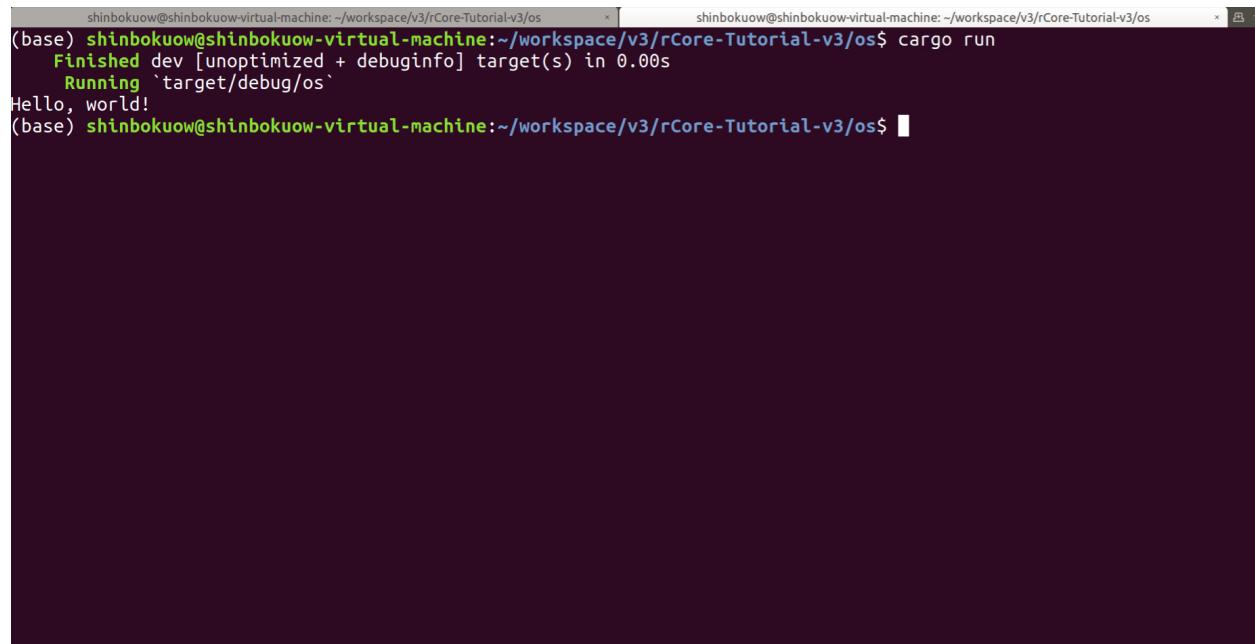
---

这里是一行数学公式  $\sin(\alpha + \beta) = \sin \alpha \cos \beta + \cos \alpha \sin \beta$ 。

基本的文本样式: 这是 斜体, 这是 **加粗**, 接下来的则是行间公式 a0。它们的前后都需要有一个空格隔开其他内容, 这个让人挺不爽的…

这是一个全面展示章节分布的例子, 来自于 ReadTheDocs 的官方文档。事实上, 现在我们也采用 ReadTheDocs 主题了, 它非常美观大方。

下面是一个测试的截图。



```
shinbokuow@shinbokuow-virtual-machine:~/workspace/v3/rCore-Tutorial-v3/os$ cargo run
 Finished dev [unoptimized + debuginfo] target(s) in 0.00s
 Running `target/debug/os`
Hello, world!
(base) shinbokuow@shinbokuow-virtual-machine:~/workspace/v3/rCore-Tutorial-v3/os$
```

接下来是一个表格的例子。

表 1: RISC-V 函数调用跳转指令

| 指令                       | 指令功能                                               |
|--------------------------|----------------------------------------------------|
| jal rd, imm[20 : 1]      | $rd \leftarrow pc + 4$<br>$pc \leftarrow pc + imm$ |
| jalr rd, (imm[11 : 0])rs | $rd \leftarrow pc + 4$<br>$pc \leftarrow rs + imm$ |



# CHAPTER 21

---

## 更新日志

---

### 21.1 2022-12-14

- 移除各分支上的 K210 开发板支持，仅保留 k210 分支作为原先 ch8 分支的镜像支持 K210。

### 21.2 2022-10-08

- 将各分支的 rustsbi-qemu 的版本更新到 701e891，支持 QEMU 7.1.0 版本。

### 21.3 2022-10-02

- 更新 VMware 虚拟机镜像至 Ubuntu22.04，内置 QEMU7.0.0 版本。同时，还更新了各章节的 Docker 相关文件。

### 21.4 2022-01-02

- 第一章更新完成，Rust 版本升级至 nightly-2022-01-01，asm 和 global\_asm 特性已稳定，相关的宏可在 `core::arch` 中找到。更新了作者和版权信息，版本暂定 3.6.0-alpha.1。

## 21.5 2021-11-20

- 更新 1~9 章，添加第八章（同步互斥），原第八章（外设）改为第九章。

## 21.6 2021-10-20

- 旧版的 3.5.0 文档及代码（全七章）已经发布在 [这里](#)。目前开始在主分支上更新新版的文档和代码。

## 21.7 2021-03-15

- 增加了在做实验的时候打补丁继承上一章节修改的教程。

## 21.8 2021-03-09

- 将所有分支的 RustSBI 版本更新为 [81d53d8] 的 0.2.0-alpha.1，主要是在 Qemu 平台上支持非法指令的转发，目前可以正确处理带有非法指令的应用程序了。参考 ch2 分支上的测例 00hello\_world.rs。

## 21.9 2021-03-07

- 在各章分支的链接脚本中加入了 .srodata/.sbss/.sdata。

## 21.10 2021-03-06

- 文档第一版初稿（全七章）完成！
- 修复了框架中基于 Qemu 平台运行却仍需要下载 kflash.py 工具的问题。

## 21.11 2021-03-05

- 第三章练习中增加了对于 sys\_gettime 语义在教程和测例中差异的相关说明。
- 修正了第四章练习中 mmap 系统调用语义中的一处错误。

## 21.12 2021-03-03

- 更新了第四章练习题。
- 为方便调试，提供了 riscv64 gcc 工具链的下载链接。
- 将文档渲染改为宽屏模式。

## 21.13 2021-02-28

修复了 ch3-coop 分支在 Rust 版本更新后无法成功运行的问题。

## 21.14 2021-02-27

完善了 easy-fs :

- 订正了 easy-fs 块缓存层的实现，移除了 dirty 子模块。
- 支持二级间接块索引，使得支持的单个文件最大容量从 94KiB 变为超过 8MiB。调整了单个 DiskInode 大小为 128 字节。
- 在新建一个索引节点的时候不再直接分配一二级间接索引块，而是完全按需分配。
- 将 easy-fs 的测试和应用程序打包的函数分离到另一个名为 easy-fs-fuse 的 crate 中。

从 ch7 开始：

- 出于后续的一些需求，sys\_exec 需要支持命令行参数，为此 shell 程序 user\_shell 中需要相应增加一些解析功能，内核中 sys\_exec 的实现也需要进行修改。新增了应用 cmdline\_args 来打印传入的命令行参数。
- 新增了应用 cat 工具可以读取一个文件的全部内容。
- 在 shell 程序中支持通过 < 和 > 进行简单的输入/输出重定向，为此在内核中新增了一个 sys\_dup 系统调用。

另外，在所有章节分支新增了 docker 支持来尽可能降低环境配置的时间成本，详见[使用 Docker 环境](#)。

## 21.15 2021-02-20

第六章文档完成。

## 21.16 2021-02-16

第五章文档完成。

## 21.17 2021-02-13

将 ch2-ch6 的 build.rs 中的对齐需求修改为刚好合适。

## 21.18 2021-02-09

在每一章的引言处加入了本章的代码树改动概况。

## 21.19 2021-02-08

将 K210 开发板的烧写工具 `kflash.py` 从项目中移除。

## 21.20 2021-02-07

将所有分支的 RustSBI 更新为最新的 0.1.1 版本 [3257d899]，**不加任何改动**直接放在项目中。这导致 `qemu` 和 `k210` 两个平台的内核入口点变得不同，目前根据 RustSBI 的默认配置，`qemu` 平台上的内核入口点为 `0x80200000`，而 `k210` 平台上为了提高烧写速度则为 `0x80020000`。

前几个章节应用放置在内存中的位置也需要对应进行修改：

- 第二章应用的起始地址变为 `0x80400000`；
- 第三章应用的起始地址变为 `0x80400000`。

文档稍后更新。

欢迎来到 rCore-Tutorial-Book 第三版！

欢迎参加 2022 年开源操作系统训练营！

---

注解: [更新日志](#)

---

# CHAPTER 22

---

## 项目简介

---

这本教程旨在一步一步展示如何 **从零开始** 用 **Rust** 语言写一个基于 **RISC-V** 架构的 **类 Unix 内核**。值得注意的是，本项目不仅支持模拟器环境（如 Qemu/terminus 等），还支持在真实硬件平台 Kendryte K210 上运行（目前主要在 rCore-Tutorial-v3 仓库的 [k210](#) 分支上维护）。



# CHAPTER 23

---

## 导读

---

请大家先阅读第零章，对于项目的开发背景和操作系统的概念有一个整体把控。

在正式进行实验之前，请先按照第零章章末的[实验环境配置](#)中的说明完成环境配置，再从第一章开始阅读正文。



## 项目协作

---

- [修改和构建本项目](#) 介绍了如何基于 Sphinx 框架配置文档开发环境，之后可以本地构建并渲染 html 或其他格式的文档；
- [reStructuredText 基本语法](#) 给出了目前编写文档才用的 ReStructuredText 标记语言的一些基础语法及用例；
- [项目的源代码仓库 && 文档仓库](#)
- 时间仓促，本项目还有很多不完善之处，欢迎大家积极在每一个章节的评论区留言，或者提交 Issues 或 Pull Requests，让我们一起努力让这本书变得更好！
- 欢迎大家加入项目交流 QQ 群，群号：735045051



## 本项目与其他系列项目的联系

---

随着 rcore-os 开源社区的不断发展，目前已经有诸多基于 Rust 语言的操作系统项目，这里介绍一下这些项目之间的区别和联系，让同学们对它们能有一个整体了解并避免混淆。

rcore-os 开源社区大致上可以分为两类项目：即探索使用 Rust 语言构建 OS 的主干项目，以及面向初学者的从零开始写 OS 项目。它们都面向教学用途，但前一类项目参与的开发者更多、更为复杂、功能也更为完善，也会用到更多新的技术；而另一类项目则作为教程项目，尽可能保持简单易懂，目的为向初学者演示如何从头开始写一个 OS。

主干项目按照时间顺序有这些：最早的是用 Rust 语言实现 linux syscall 的 [rCore](#)，这也是 rcore-os 开源社区的第一个项目。接着，紧跟 Rust 异步编程的浪潮，诞生了使用 Rust 语言重写 Google Fuchsia 操作系统的 Zircon 内核的 [zCore](#) 项目，其中利用了大量 Rust 异步原语实现了超时取消等机制。最新的主干项目则是探索 OS 模块化架构的 [arceos](#)。

教程项目则分布在 rcore-os 和 LearningOS 两个开源社区中。最早的第一版教程是 [rcore\\_step\\_by\\_step](#)，第二版教程是 [rCore\\_tutorial](#)，第三版教程是 [rCore-Tutorial](#)，最新的教程(暂定 v3.6 版本)就是本项目 [rCore-Tutorial-v3](#) 仍在持续更新中。

教程项目均以 rCore 为前缀，是因为它们都是主干项目 [rCore](#) 的简化版。“rCore”这个词在不同的语境中指代的具体项目也不一样：如果在讨论教程项目的语境，比如以 xv6 和 ucore 以及 ChCore 等项目类比的时候，那么往往指的是最新的教程项目；相反如果讨论的是大规模项目的话，应该指代 [rCore](#) 或者其他主干项目。由于教程项目是由 [rCore](#) 简化来的，所以“大 rCore”指的是 [rCore](#) 主干项目，相对的“小 rCore/rCore 教程”则指的是最新版的教程项目。



## 项目进度

---

- 2020-11-03: 环境搭建完成，开始着手编写文档。
- 2020-11-13: 第一章完成。
- 2020-11-27: 第二章完成。
- 2020-12-20: 前七章代码完成。
- 2021-01-10: 第三章完成。
- 2021-01-18: 加入第零章。
- 2021-01-30: 第四章完成。
- 2021-02-16: 第五章完成。
- 2021-02-20: 第六章完成。
- 2021-03-06: 第七章完成。到这里为止第一版初稿就已经完成了。
- 2021-10-20: 第八章代码于前段时间完成。开始更新前面章节文档及完成第八章文档。
- 2021-11-20: 更新 1~9 章，添加第八章（同步互斥），原第八章（外设）改为第九章。
- 2022-01-02: 第一章文档更新完成。
- 2022-01-05: 第二章文档更新完成。
- 2022-01-06: 第三章文档更新完成。
- 2022-01-07: 第四章文档更新完成。
- 2022-01-09: 第五章文档更新完成。