基于 Rust 语言的内存管理设计与实现

摘要

操作系统是计算机最重要的系统程序，由不同较为独立的子系统组成。内存管理系统作为其中的重要组成部分，提供了物理内存管理，地址空间构建等重要功能，内核与应用的正确运行都离不开内存管理系统的支持。经历长年发展，内存管理技术诞生了多种相关的算法与数据结构，经典的管理策略往往是计算机专业学生学习的重点。

然而，出于安全性与功能性的考虑，现实操作系统中的内存管理实现往往复杂、庞大。为了辅助教师教学，帮助学习者理解，本论文设计、开发了一个教学内存管理系统。系统在 QEMU 平台模拟了一台基于 RISC-V 硬件指令集的裸机，并使用 Rust 语言开发，能够在兼顾安全性的同时，发挥教学作用。

内存管理系统定义了相关功能与接口，可以相对独立地运行，也可以嵌入在功能完善的内核中，其功能包括分配物理资源、生成地址空间、提供内存管理器的可视化展示等。通过构建底层的数据结构、两个基于不同物理资源抽象的内存管理器、直观的配置文件，内存管理系统定义了内存管理器的基本行为，展示了多种数据结构与算法的实现。学习者可以低成本地了解内存管理策略的具体存在形式，并快速参与实际系统的开发与实现中，充分达到理论与实践结合的教学目标。

关键词： Rust　RISC-V　操作系统　内存管理　分页机制

Design And Implementation Of Memory Management Based On Rust Language

ABSTRACT

The operating system is the most important system program of a computer and consists of different and independent subsystems. As a vital proportion, the memory management system provides essential functions such as management of physical memory and address space of applications. During a long development, various related algorithms and data structures have emerged in memory management. Some classical strategies are usually necessary for students who study computer science.

However, considering of security and functionality, the implementation of memory management in real operating systems is usually complex. In order to assist teachers and learners, this thesis develops a memory management system for teaching. The system simulates a bare metal computer using RISC-V hardware instruction on QEMU platform, and is developed by Rust language, catering to both teaching's purpose and safety concern.

The memory management system defines some functions and interfaces, and is able to run independently or be embedded in a big system. Its functions include allocating physical memory, generating address space, and providing a visual representation. By fundamental data structures, two memory managers based on different physical resources abstraction, and a configuration file, the system defines the basic behaviors of a memory manager, and shows the implementation of several data structures and algorithms. Learners can comprehend practical management strategies easily, and participate in the implementation based on a real system, fully combining theoretical knowledge with practice.

Key words: Rust RISC-V Operating system Memory management Paging mechanism

目录

第 1 章 引言

1.1 课题背景及意义

操作系统作为计算机中最基本、最重要的系统软件，经历了批处理操作系统，多任务管理系统，分时操作系统[1]，实时操作系统[2]，面向网络的操作系统[3]等形态。作为软、硬件沟通的桥梁，操作系统管理内存、输入与输出设备、文件系统等硬件，并为上层应用程序提供执行环境与服务。

熟悉操作系统是计算机领域从业者的必修课程。在课堂、教科书中，初学者一般能了解操作系统中原理性的知识，但面对子系统繁多、编码复杂的真实操作系统时，初学者往往难以理解与掌握。无法参与对真实操作系统的开发、调试，实现理论与实践相结合，是操作系统教学领域面临的难题[4]。

基于上述背景，本文设计与实现了一个内存管理系统，在教学中有以下意义：

（1）学习者可以通过直观的方式了解系统的软硬件管理方式，如硬件资源的接口，系统调用的实现等。

（2）内存管理中关键的算法与数据结构，例如最佳适应算法、页表数据结构等，学习者可以在掌握理论知识的同时，从教学系统中了解它们的真实存在形式。

（3）系统能够定义出简洁、直观的接口，对各部分功能进行模块化管理，学习者能充分掌握各部分功能的抽象与实现。

内存管理系统能够为内核与应用分配内存资源，生成地址空间，开启虚拟地址访存方式。同时，系统使用多种类型的数据结构与算法，学习者能根据已定义的通用接口，实现多种定制化的内存管理方式，测试不同的内存管理器。通过简单、直接、易于上手的方式，学习者能了解并掌握不同内存管理方式的原理与设计，在较低的学习成本下进行实践。

内存管理系统使用 Rust 语言，基于 RISC-V 指令集架构下实现。Rust 编程语言的能兼顾可靠性与高性能[5]，它克服 C 语言中开发者对内存进行直接的读写操作[6]所带来的安全隐患[7]，同时也不需要使用虚拟机牺牲性能来保证安全性[8]。使用 RISC-V 而没有使用基于 Intel 处理器的 x86 指令集[9]，原因在于经历长时间的迭代与发展，x86 指令集庞大而复杂，学习成本较大，相比下 RISC-V 指令集架构是一种新兴、开源、精简的指令

集架构[10]，具备更开放、学习成本更低的优点[11]。

内存管理系统将基于 RISC-V 架构，使用 Rust 语言开发。利用 Rust 的特性，保证了系统的安全性，满足了教学操作系统的需求，能够帮助学习者对内存管理的学习与实践。

1.2 国内外研究现状

1.2.1 Rust 安全性研究

Rust 语言特性能为内存管理提供安全保证，对 Rust 安全性的研究帮助开发者了解 Rust 所有权机制的本质，避免产生不安全的 Rust 代码。

Rust 语言诞生之初，R Jung，JH Jourdan 等人认为其声称的安全性没有得到正式证明，有理由质疑它们是否成立。Rust 采用了一个强大的、基于所有权的类型系统，但通过内部使用不安全特性的库，扩展了类型系统的表达能力。研究为代表 Rust 的一个真实子集的语言提供了第一个正式的安全证明，并指出不安全特性的新 Rust 库必须满足什么验证条件，才能被视为对 Rust 的安全扩展，随后在 Rust 生态一些重要的库中进行验证[12]。

Aaron Weiss, Olek Gierczak 通过着手 Rust 借用检查器的类型系统账户，捕捉 Rust 所有权模型的本质。他们推出了一种接近 Rust 源码级别的形式化编程语言 Oxide，采用了生命周期的新观点，作为引用来源的近似值，类型系统能够通过子结构类型判断自动计算此信息，为使用进度和保存的借用检查提供了第一个类型安全的语法证明[13]。

Boqin Qin 等人对实践中的 Rust 程序的安全性做实证性的研究，最终给出了若干编写 Rust 安全代码的启示，以及提出了对 Rust 开发者的建议。他们调研了开发者在实际编写 Rust 程序中存在的安全问题，以及 Rust 自身特性对开发者在实际编程中的影响。文章重点研究了以下几个问题：Rust 开发者编写不安全代码的原因；现实中的 Rust 程序存在的内存安全隐患；容易被开发者触发的 Rust 程序并发漏洞。通过对实际程序中不安全代码、内存漏洞、并发漏洞的研究、分析和统计，文章提出了对 Rust 设计者和开发者的数条启示与建议[14]。

1.2.2 Rust 内核与 RISC-V 相关研究与实践

基于 RISC-V，李晓英、刘敏设计、实现了一个微处理机。该系统分为 CPU 内核部分和外围设备部分。内核部分采用 RISC-V 指令集系统并使用五级流水线，同时分散了控制单元，并拥有 8 个中断。内核部分和外设部分通过总线连接，包括地址总线、控制总线、数据总线。经测试整个系统最终能够在最快 50 MHz 的时钟输入频率下正常运行

[15]。Yunsup Lee、Andrew Waterman 等人在基于敏捷开发的流程下搭建了 RISC-V 微处理器。文章介绍了一种灵活的硬件开发方法，将其用于现代的 RISC-V 微处理器中，并讨论了这种方法如何使小型团队能够在几个月内构建出节能、经济且具有行业竞争力的高性能微处理器[16]。

A Levy, B Campbell 等人使用 Rust 进行内核开发过程中，得出了与此前研究者不一样的结论。他们认为用 Rust 语言编写的操作系统内核将具有极其细粒度的隔离边界，没有内存泄漏，并且可以免受各种安全威胁和内存错误的影响，且不需要对 Rust 进行任何修改，可以使用非常少量的不安全代码来实现内核[17]。A Levy，MP Andersen 等人使用 Rust 语言在嵌入式系统的开发实践中，发现在事件驱动较常见的嵌入式平台，Rust 所有权模型阻止了常见的资源共享，与硬件资源的现实相冲突。团队总结了这些经验以及它们与内存安全的关系，并给出了最大程度地保留安全保证的变通方法[18]。

针对高校计算机系统各课程实验衔接不紧密、成本较高的问题，孙卫真、刘雪松等人设计了一套计算机系统综合实验平台。实验分为硬件实验与软件实验两部分，硬件实验以在低成本 FPGA 芯片上搭建 picorv32 RISC-V CPU 为核心，软件实验在硬件实验基础上，以编写 Rust 操作系统内核为核心。实验以低成本在同一平台完成计算机组成原理与操作系统的课程实验，两者软硬件结合，构建了一个完整的实验体系，可以满足高校计算机系统教学的要求[19]。

1.2.3 内存管理相关研究

Mads Tofte，Jean-Pierre Talpin 描述了用于执行动态内存分配和回收的程序的内存管理规则。在运行时，所有值都被放入区域中，使用基于类型和效果的程序分析自动推断所有区域分配和回收点，这种方案不需要垃圾收集器的存在[20]。

Gay D, Aiken A 认为，基于区域的内存管理能作为垃圾回收或显式分配、释放的内存的一种替代方案。在基于区域的内存分配与回收系统中，每个分配都指定一个区域，并且通过销毁一个区域来回收内存，释放其中分配的所有存储空间。研究指出，在一套分配密集的 C 语言程序中，基于区域的系统与 malloc/free 性能相当，有时甚至更快，研究还展示了区域内存管理系统支持低开销的安全内存管理，而基准测试经验表明，修改许多现有程序以使用区域内存管理系统并不困难[21]。

基于区域的类型系统通过使内存管理显式，但可安全验证，成为垃圾收集的替代方案。然而在低级、类型安全的代码中使用区域类型系统尚未澄清。Crary K, Walker D 等人提出了一种编译器中间语言，它支持基于区域的内存管理，有可证明安全的类型系统，

并且可以直接编译为类型化的汇编语言。源语言可以使用已知的区域推理算法编译成中间语言。此外，区域生命周期不需要在中间语言中进行词法范围，但可以在没有复杂分析的情况下检查语言的安全性[22]。

魏海涛, 姜昱明等基于Windows虚拟内存管理，提出一种高效的动态内存管理机制，用于高效管理内存，防止内存泄漏、越界访问等问题的出现。研究团队通过创建内存管理器维护虚拟地址空间，处理内存申请和释放请求，同时改进了传统动态内存分配与回收算法。测试结果表明，该方法降低了申请与释放内存的时间耗费，减少了内存碎片的产生，提高了动态内存管理效率[23]。

杨峰基于 Linux 内核实现一种动态内存管理机制。软件中的共享内存有时会遇到一个进程占用过大，导致其他进程无法得到需要内存的潜在问题。作者推出的动态内存管理机制能够限制每个进程所能申请的最大内存数，避免进程内存泄露造成的系统崩溃，效率高，易用性好[24]。

内存管理教学相关研究与实践中，孙温稳通过开发相关接口，引导学生实现内存管理中的多种常见行为，包括初始化内存、使用最先适应、最优适应等策略分配内存、释放内存、显示内存等函数[25]。吴敬仙、缪行外通过内存分区算法与内核演示系统，展示了内存管理中多种算法的动态模拟实现过程，运用多媒体教学方法帮助学生理解内存管理的分配算法，提高教学质量[26]。动态内存分配是编程语言中的教学难点之一，徐艳艳、陈志泊等人从创建动态对象、创建动态数组、包含动态内存分配的类、异常与内存管理四方面介绍了编程语言中动态内存管理相关知识，提高学生使用指针操作动态内存的编程能力[27]。

1.3 论文主要研究内容

本课题基于 Rust 语言设计与开发了操作系统中的内存管理系统。主要工作如下：

（1）基于 RISC-V 指令集的 SV39 多级页表机制，设计地址、页号、逻辑段等基本数据结构,构建页表存储虚实页号映射。

（2）设计分页、分段内存管理方式，抽象分配器的基本行为。分页模式下，设计基于不同数据结构的也管理器；分段模式下，以不同算法实现段分配。学习者可根据分配器基本行为定义实现多种不同的内存管理器。

（3）设计内存管理系统的功能与接口，包括物理资源管理、地址空间生成、可视化接口等。接口可被其他模块调用以取得底层资源。

（4）设计外部配置文件。使用者通过对配置文件的修改，无需更改源码，系统可按

指定模式进行资源分配与回收。

1.4 论文组织结构

本论文分为五章：

（1）第 1 章介绍课题背景与课题意义，对介绍相关技术选型的原因及其国内外研究现状。

（2）第 2 章介绍课题相关核心知识，包括使用 Rust 核心的语言特性，引导加载程序 Rust-SBI、内存管理关键概念、SV39 多级页表机制原理等。

（3）第 3 章介绍内存管理系统整体设计与具体实现，包括基本数据结构的定义、分页、分段内存管理机制的实现、接口定义等。

（4）第 4 章从不同层次与角度对内存管理系统进行测试，验证各分配器的正确性与配置文件的有效性等。

（5）第 5 章总结本课题达到的目标与具备的价值，对未来的系统的扩展与完善做出展望。

第 2 章 相关技术介绍

本章介绍基于 Rust 的内存管理的相关知识与技术，主要内容包括 Rust 语言及其特性、操作系统执行环境、SV39 分页机制等。

2.1 Rust 编程语言特性

Rust 兼顾性能与安全性，是一种面向系统的编程语言。同时，Rust 也是一门编译型的语言，编译器在编译期进行一系列安全性检查，确保程序不会出现内存泄漏、内存重复释放等错误。了解 Rust 语言特性，编写内核时可更加专注代码逻辑，将人为可能导致的安全错误交由编译器检查，确保内存管理系统的安全性。

2.1.1 所有权

所有权机制是 Rust 最为显著的特性。Rust 通过所有权系统在编译时明确变量资源被分配及回收的时机，使得资源的管理不会在运行时带来性能上的损失。在编程语言的设计中，有的语言会触发 stop-the-world 机制挂起所有进程来进行垃圾回收，降低运行效率。有的语言则允许开发者显式地分配或释放内存，引起一些不安全的问题，例如 Double-Free、Use-After-Free 等错误[28]。而 Rust 所有权系统能同时发挥出安全、高效的特性。

开发者在编写 Rust 程序时须清楚每个变量的所有权归属，才能通过编译器的安全检查。所有权的规则包括：

（1）所有变量都是其值（资源）的所有者；

（2）一个值只有一个所有者；

（3）当变量（所有者）离开了作用域，其拥有的资源则会被自动回收；

对于编译期可明确其占用空间的类型（如整型），它们是“可复制的”。当变量之间进行赋值操作，值会先被复制，后进行赋值。对于编译期无法确定大小的类型（如 String），变量间的赋值会导致值所有权的移交，移交后则无法再通过原变量访问值。

所有权的移交不仅发生在变量赋值之间，函数调用者和被调用之间的值传递，例如入参、返回，同样会发生值的所有权移交。当调用一个函数，而不希望将入参所有权移交给函数时，则需要“借用”，用符号“&”表示借用某个值的所有权。从而在不移交所有权下给其他变量访问值的权限。值的引用分为了可变引用和不可变引用，规则如下：

a. 当一个值有一个可变引用时，不允许其拥有其他任何类型的引用；

b. 否则，一个值可以有多个不可变引用。

借用规则可以避免数据争用[29]的情况出现。

所有权机制是 Rust 独特的机制，在开发 Rust 程序时明确各值的所有权，及变量的生命周期，以通过编译器的安全检查，从而使 Rust 的安全性应用到内存管理系统的开发中。

2.1.2 Rust 的空类型

很多编程语言会定义空类型，但空值的定义使得一些人为错误无法被编译器发现，开发者遗漏对于潜在空值的判断，可能会在运行时带来巨大的安全问题与损失[30]。Rust 的空值被一个名为 Option 的枚举类型包裹，从语言设计层面强制开发者进行空值检查，否则无法通过编译。Option 类型从根源上杜绝了由于开发者的粗心所带来的空指针、空类型异常，在分配、回收操作较为常见的内存中应用广泛。

```
enum Option<T> {
    Some(T),
    None,
}
```

图 2-1 Rust Option 类型定义

Option 枚举类型包含了 Some(T)与 None 两个变量，如果一个变量为 Option 类型，它必须为 Some(T)类型与 None 中的一种。其中，T 代表任意类型，一般为非空时希望返回的类型。当底层返回值可能为空或非空时，开发者必须将返回值包裹在 Option 中。

当调用者取得 Option 类型的返回值，意味着该值可能为空。此时必须对 Option 进行空值判断。如果试图直接取 Option 中的 T 实例则无法通过编译。这种封装从源头上避免了人为错误。

内存管理中，常出现“分配成功”或“分配失败”的语境。此时 Option 类型的存在使得在系统运行前就可以发现一些错误，变得更加可靠。

2.2 操作系统执行环境

执行环境是计算机体系结构中重要的概念，其主要作用是为上层的软件提供的功能与资源。计算机中自底向上按逐层划分，因此在不同的层次中运行的软件就对应着不同的执行环境。早期应用程序在硬件上运行，应用程序执行环境是硬件。后来出现了一些较为通用的函数库，应用程序可以通过函数库来进行开发，此时应用程序执行环境是函

数库，而函数库执行环境是硬件。

本项目应用程序在操作系统提供的服务下运行，其执行环境是 Rust 编写的操作系统。而 Rust 内核并非直接面对裸机开发，而是基于 Rust SBI。SBI 是计算机的引导加载程序。在 CPU 加电后会执行 ROM 中的指令并加载引导程序及数据进内存中，随后引导程序执行一些初始化操作，完成后转交计算机控制权给操作系统。Rust 内核的部分功能需要通过对 SBI 的调用来访问硬件，因此内核实在 Rust SBI 提供的执行环境下运行的。

CPU 在运行过程中，可以会由于系统调用、应用程序异常等原因，发生执行环境之间的切换，例如从应用程序切换至内核进行系统调用的分发处理，处理完毕后从内核返回应用程序。不同执行环境之间的切换主要依靠层之间的接口完成，在保证层与层能彼此交互的同时实现了不同层的封装与隔离。

2.3 地址空间与分页机制

地址空间是对物理内存的抽象，也称虚拟内存。它是操作系统在 CPU 中的内存管理单元（MMU, Memory Management Unit）的硬件支持的基础上给应用程序和用户提供一片大的（可以远远大于物理内存大小）、连续的、隔离的存储空间。当操作系统将内存与硬盘空间一起管理，就可以为用户提供一个容量比实际内存大的虚拟存储空间。在虚拟内存的管理方式下，用户存放在各自虚存中的数据与其他用户的空间隔离。在启动虚存机制后，软件通过 CPU 访问的每个虚拟地址会通过 MMU 的自动转换，变为一个真正在物理内存中的物理地址。

SV39 是 RISC-V 提供的一种分页机制[31]。基于 SV39 机制，可以开启虚拟内存的管理方式，为应用程序生成隔离的地址空间。在内存管理中，了解 SV39 分页机制的原理与规范是开发各分配器、开启地址空间的基础。

2.3.1 satp 寄存器

RISC-V 默认不使能分页机制，CPU 的访存地址视为一个直接的物理地址。根据 SV39 机制，修改 satp 的寄存器后，分页模式则会启用。使能分页机制后的访存地址都会被作为当前地址空间的虚拟地址，自动被 MMU 根据多级页表中的映射，转换为一个物理地址，再通过物理地址访存。

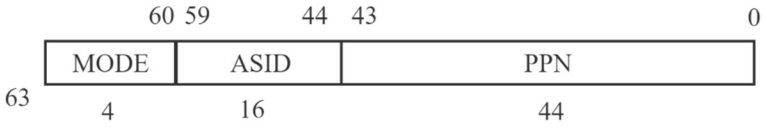

图 2-2 satp 寄存器

上图展示了 satp 寄存器的构造及各部分的含义，其中：

MODE：当被设置为 0 时（默认），代表所有访存都被视为物理地址；设置为 8 时，SV39 分页机制被启用。

ASID：暂时不涉及。

PPN：存放当前地址空间根页表的物理页号。在给定一个虚拟页号后，MMU 就会自动从根页表所对应的多级页表中逐级根据虚拟页号检索其对应的物理页号。

2.3.2 虚拟地址与物理地址格式

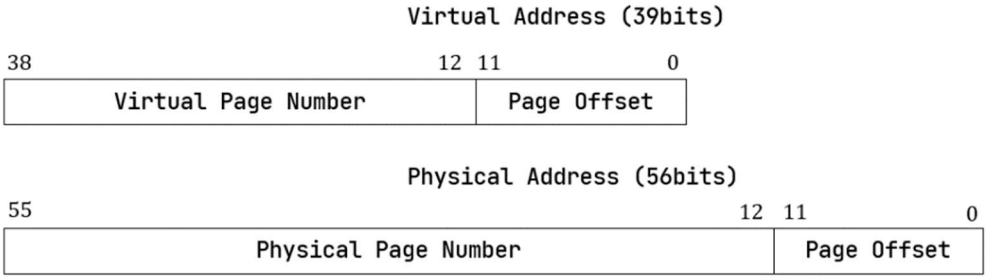

图 2-3 SV39 下的虚拟地址与物理地址

SV39 页表中，存储的是（虚拟）页与（物理）页之间的映射。虚拟页与物理页都有各自的页号。按照规定，一个页的大小为 4KB，用 12 位地址表示。因此，无论是虚拟页还是物理页，页号都可以用地址的低 12 位以外的位表示，而低 12 位则作为页内偏移。一个虚拟地址通过其虚拟页号与页表检索到物理页号后，拼接上页内偏移，就是虚拟地址所对应的物理地址。

2.3.3 页表项（Page Table Entry，PTE）格式

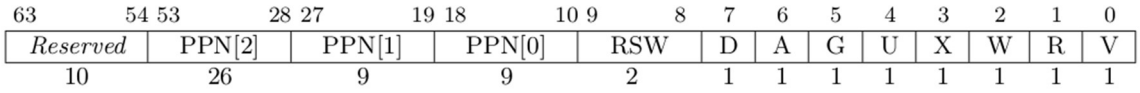

图 2-4 SV39 多级页表中的页表项

页表项是 MMU 通过虚拟页号在页表中检索到的内容。页表项中除了物理页号，还包含一组标志位，它控制了应用对地址空间中该虚拟页的访问权限，其中：

V(Valid)：当位 V 为 1 时，页表项合法的；

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处于 U 特权级（用户特权级）下被允许访问；

A(Accessed)：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)：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。

2.3.4 三级页表的地址转换

虚拟页从根页表开始，从三级页表中逐层检索到对应的物理页号，检索过程如图所示。

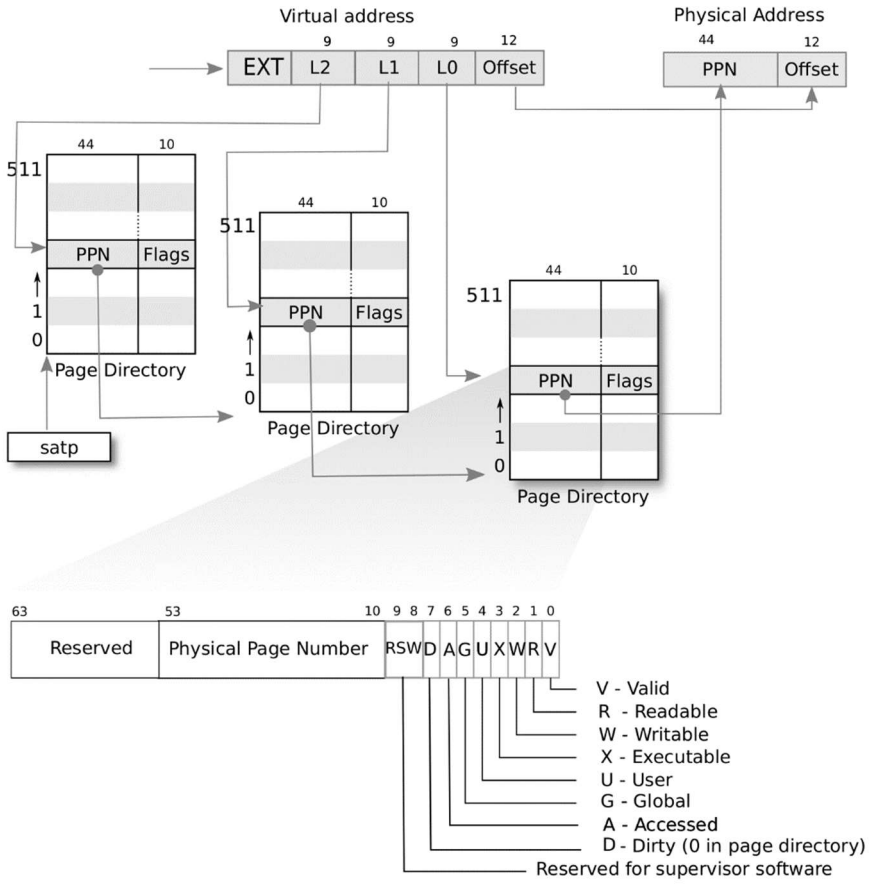

图 2-5 三级页表的地址转换[32]

每一张页表都有 512 个页表项，占用 4KB 的存储空间，正好是一个页帧的大小。同时，人为地将虚拟页号的 27 位分为三部分，每部分长 9 位，9 位地址正好对应 2^9=512 个页表项，即一张页表的大小。

当切换到一个地址空间时，装载当前的地址空间的根页表所在的物理页号到 satp 寄存器后。MMU 取得一个虚拟页号后，首先根据 satp 寄存器找到根页表所在地址，把最高 9 位的虚拟页号（VPN2）作为在第三级页表（根页表）中的页内偏移量，找到根页表中对应的页表项，此时页表项中中的 PPN（物理页号）字段为第二级页表的物理页号。

以此类推，在找到第二级页表的物理页号后，把 VPN1 作为第二级页表的偏移量，找到对应页表项，该页表项中的 PPN 字段为为第一级页表的物理页号；同样，VPN0 作为第一级页表的偏移量，找到一级页表中的页表项。此时，页表项中的物理页号就是最终虚拟页号所映射的物理页的页号了。

最后，目标物理页号拼接上虚拟地址中的页内偏移（低 12 位），就是虚拟地址所对应的物理地址。

2.4 本章小结

本章介绍了与内存管理系统相关的知识、概念与技术。

（1）介绍了开发语言 Rust 与其他语言不同的特性。这些特性为 Rust 安全性与性能提供保证，了解与掌握这些特性是使用 Rust 语言进行系统编程的基础。

（2）介绍了操作系统中执行环境的基本概念与层次。Rust-SBI 是 Rust 内核启动前的引导程序，同时为内核提供了一系列硬件接口，是内核的执行环境。操作系统内核则是应用程序的执行环境。应用程序因系统调用产生陷入，会导致执行环境的切换，从而需要进行一系列对上下文的保存等操作。这是内存管理系统的设计中需要考虑的。

（3）介绍了地址空间基本概念与 SV39 多级页表机制。地址空间为不同的应用提供了一个隔离的虚拟内存空间。SV39 多级页表机制是 RISC-V 一种分页机制，使能 SV39 开启虚拟内存的访存方式，需要熟悉了解分页机制中页的大小、物理/虚拟地址的组成、页表项的格式，以及多级页表的转换过程。开启分页机制后，MMU 可在访存时根据页表自动完成地址的转换。

第 3 章　内存管理系统设计与实现

3.1 整体设计思路与框架

本节整体介绍内存管理系统提供的功能及其内部的实现框架。

3.1.1 内存管理系统整体设计

在系统内部，底层数据结构定义了符合 SV39 规范的类型，是上层多种内存分配方式的基础。基于底层定义的数据结构，定义了两类内存管理器。基于页的内存管理器以页为单位进行地址空间的划分，使用不同的数据结构和算法实现不同的分配与回收策略。基于段的内存管理器以段位单位进行地址空间的划分。内存管理系统的最上层，配置文件提供了内存管理方式的选择，可选择不同的策略的页分配器或段分配器。

从外部视角，内存管理系统主要为内核与应用程序生成隔离的地址空间。外界通过调用相关接口，内存管理系统完成物理资源的分配并返回地址空间。当任务执行完毕，内存管理系统自动对其占用的资源执行回收操作。

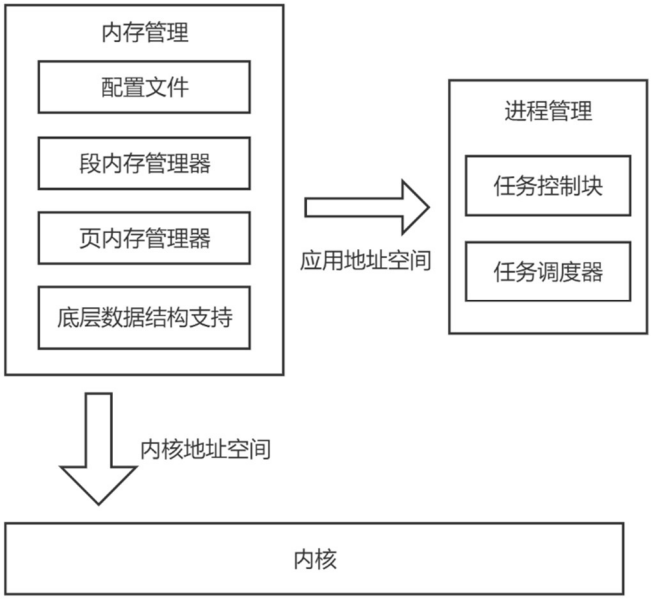

图 3-1 内存管理系统整体设计

3.1.2 内存管理系统内部框架

在 SV39 分页机制上进行内存管理，物理内存、虚拟内存被分成大小固定的页，同时需要构建页表保存物理页与虚拟页映射关系。如图展现了内存管理系统的内部框架：

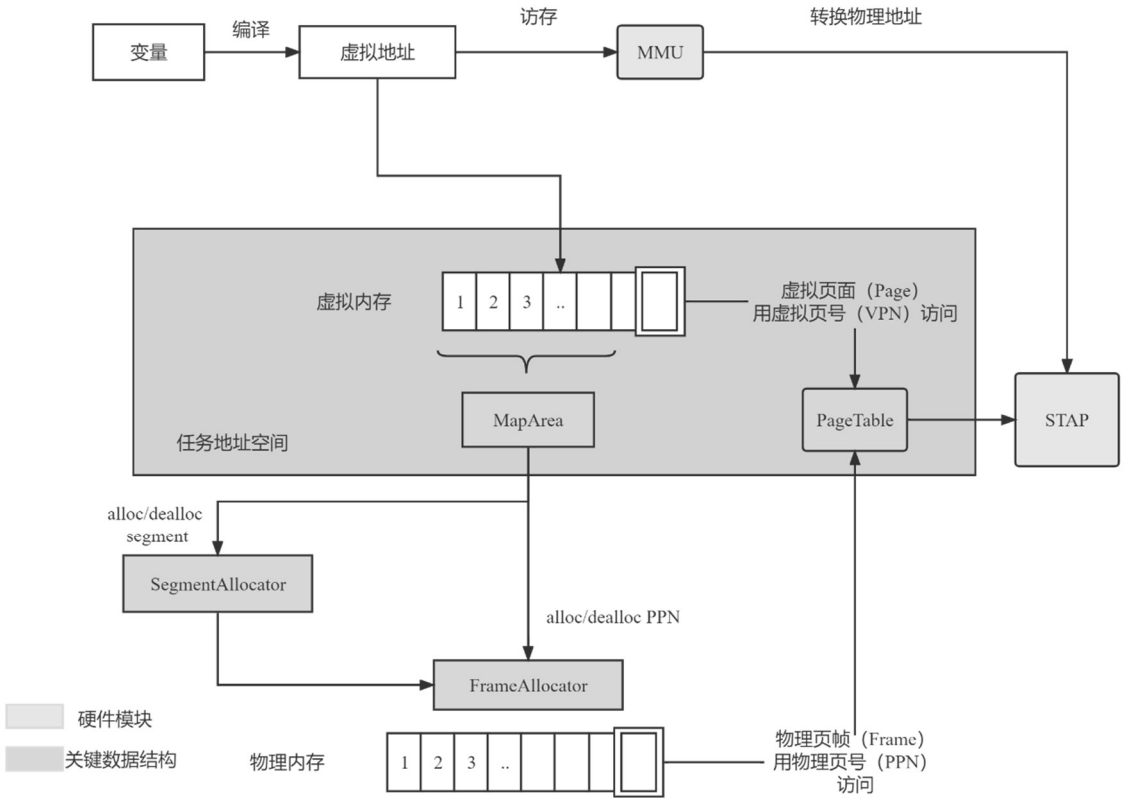

图 3-2 内存管理系统内部框架

底层的页帧分配器 FrameAllocator 以物理页号的方式管理所有物理页帧，对外暴露出 alloc/dealloc 接口，调用者通过接口获取、释放页帧资源。页帧管理器内部选用适当的数据结构进行页帧的维护。

应用程序在编译完成后形成若干逻辑段，逻辑段对应一段连续的虚拟页。不同逻辑段有不同的读写权限。一个应用分配内存资源的过程为：获取应用各逻辑段的虚拟地址范围，分别对不同逻辑段进行内存资源分配。最终，逻辑段内的所有虚拟页都被分配了对应的物理页帧，并将虚拟页号与物理页号的映射记录在应用程序的页表 PageTable 中。一个应用程序所有逻辑段中的虚拟地址范围及其页表共同组成了应用的地址空间。

当切换到其他应用程序时，地址空间也需要被切换。根据 SV39 分页机制，内核把切换的应用程序的页表所在物理页号写入 satp 寄存器中完成地址空间的切换。

从访存的角度看，一个变量在编译完后转换为该应用程序地址空间中的一个虚拟地址，在访问该变量时，硬件中的内存管理单元（Memory Management Unit）自动把虚拟地址根据 STAP 寄存器所指向的页表转换为物理地址，完成访存。

在管理物理页资源的页帧管理器与地址空间间有一层段分配器（SegmentAllocator）。段分配器搭建在底层的页帧分配器基础之上。此处段（Segment）是物理段，与地址空间

中的逻辑段（MapArea）有区别。加入段分配器可以以段为单位进行内存资源的分配与回收，应用首次适应、最佳适应的算法分配逻辑段的资源请求，进而在教学操作系统中展现到更多样的内存管理方式。

3.2 基本数据结构设计

3.2.1 物理资源数据结构

物理地址、物理页号、虚拟地址、虚拟页号，都是一个非负整数，用 Rust 中的无符号整型 usize 表示。规定一个物理页帧的大小为 4KB，因此页帧内地址的访问需要 12 位。所有在同一页内的物理地址，除去其低 12 位，其余部分是相同的，故将物理地址低 12 位以外的部分作为物理页号。基于以上设计，地址和页号之间的转换能通过简单移位完成。在内存管理系统中，会频繁使用这两种地址和页号及它们间的转换，因此将地址和页号定义为一个类型。如图所示：

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

/// Virtual Address
#[derive(Copy, Clone, Ord, PartialOrd, Eq, PartialEq)]
pub struct VirtAddr(pub usize);

/// Physical Page Number
#[derive(Copy, Clone, Ord, PartialOrd, Eq, PartialEq)]
pub struct PhysPageNum(pub usize);

/// Virtual Page Number
#[derive(Copy, Clone, Ord, PartialOrd, Eq, PartialEq)]
pub struct VirtPageNum(pub usize);
```

图 3-3 地址与页号类型定义

与页号、地址有关的 4 种类型是对基本类型 usize 的简单封装，目的如下：

（1）后续使用时，传递相应类型所对应的实例，从类型就可以得知当前使用的是物理/虚拟地址/页号的实例。如果传递的是仅仅是整型，在使用当前变量时可能会对是 4 种类型中的某一种产生混淆；

（2）可为类型常用的行为定义方法，当需要地址、页号相互转换时能通过调用较为方便地实现。

四个类型都实现了 Copy、Clone 特性，在进行实例的值传递时不会产生所有权的移交。Ord、Eq 等特性则使能了实例之间的大小、等值判断等操作。

物理页帧虽然可用其物理页号唯一表示，但为了利用 Rust 中变量的生命周期特性更方便地进行页帧的分配与回收，封装物理页号为物理页帧类型。

```
/// manage a frame
pub struct FrameTracker {
    pub ppn: PhysPageNum,
}
```

图 3-4 物理页帧类型

运用 RAII[33]的思想，针对页帧类型可以定义一系列与之相关的行为。外界对于获取、控制、释放页帧资源以 FrameTracker 的实例来进行的。例如，在构建新页帧实例时即完成初始化、清空页内数据操作。而为 FrameTracker 实现 Drop 特性，可以在页帧的生命周期结束后自动调用释放方法，免去手动调用释放资源的接口，其作用在后面有明显的体现。

物理段是一段连续的物理内存，为在 RISC-V 架构上开启虚拟内存管理，定义页帧以符合 SV39 规范是必须的。因此将段定义为一组连续的页帧，一个段实例维护一个以页帧实例为元素的向量。当一个段实例生成后，该段控制了其占有的页帧的生命周期。

```
pub struct Segment {
    frames: Vec<FrameTracker>,
}
```

图 3-5 段（Segment）类型

实际开发中， Rust 的所有权机制的存在，导致仅使用段类型会出现实现的困难：初始化后段分配器拥有段实例的所有权。在进行段分配时，如果把将要分配的段所有权移交给使用者，使用者生命周期结束时，段实例被释放。之后段分配器再试图分配该段不得不重新生成，不符合段分配的设计初衷。而如果把分配的段克隆再转交所有权给使用者，则造成效率更低下、占用空间更大。因此定义一个“段号”类型，将段号（整型）封装为一个类。

```
pub struct SegmentTracker {
    seg_index: usize,
}
```

图 3-6 段号类型

段号类型是一个轻量的、无符号整数的封装类，其创建与移交花费代价小。在此基础上，进行段分配时可生成段号实例并移交使用者，与使用者生命周期绑定。当使用者被释放时，段号实例也被释放。此时自动调用段号类型的 Drop 特性中的段回收函数，可完成段的回收。

3.2.2 页表项与页表

基于 SV39 机制开启虚拟内存模式，设计记录虚实地址映射的页表是必须的。对页表项及页表进行开发时，需要遵守相应规范与格式，才能在使能分页模式后 MMU 自动、正确地完成地址转换、获取相应页的访问权限。

页表项（Page Table Entry，PTE）是页表中的记录，可以用一个无符号整数来表示，可以定义为 usize 的封装类型。

```
pub struct PageTableEntry {
    pub bits: usize,
}
```

图 3-7 页表项类型

一个页表项包含了其保存的物理页号，与该物理页号的标志位。标志位用于控制该页的读写访问权限。生成一个页表项实例，就是将传入的物理页号与标志位进行拼接。

一张完整的多级页表（以下简称“多级页表”）是一棵三层的树，树的每个节点都是一张由 512 个页表项组成的子页表（以下简称“页表”），每张页表占用一个页帧（4KB）的存储空间。因此用页帧类型来表示一张页表。一张多级页表是需要控制多张页表，即多个页帧资源。

如图，一张多级页表由根页表所处的物理页号，与一个页帧类型的向量组成。在使能多级页表的分页机制时，只需把根页表物理页号写入 STAP 寄存器即可，各级页表之间的联系体现在页表项中。此处定义页帧元素类型向量的原因是让多级页表实例获得所有子页表的所有权，完成生命周期的绑定。当多级页表生命周期结束，页表树中的所有页表会被自动回收。

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

图 3-8 多级页表类型定义

3.2.3 逻辑数据结构

逻辑数据结构面向使用者，初始化时不占用物理资源。逻辑数据结构通过调用物理资源分配器的接口完成的内存分配。

一个程序编译、连接完成后会形成若干逻辑段。逻辑段与物理段不同，逻辑段是程序必然存在的几个用于保存不同数据的虚拟地址范围，物理段则是一种用于内存管理的模型或结构。

不同逻辑段对应不同的权限，因此对每一个单独的逻辑段做统一的内存分配。为了后续提供自由选择选择不同类型内存分配器的接口接口，定义逻辑段如下：

```
pub struct MapArea {
    vpn_range: VPNRange,
    data_frames: BTreeMap<VirtPageNum, FrameTracker>,
    segment: SegmentTracker,
    map_type: MapType,
    map_perm: MapPermission,
}
```

图 3-9 逻辑段类型定义

如图，逻辑段类型包含其虚拟地址的范围、控制资源、映射类型、读写访问权限。一个逻辑段的映射方式有恒等映射与页帧映射两种。恒等映射方式下，虚拟页号与物理页号完全相等，主要在内核生成，使能分页机制前使用，目的是能够从物理访存方式平滑地过渡到使用虚拟访存方式。页帧映射方式下，虚拟页号与物理页号没有关系，为虚拟页面分配资源而调用页帧分配接口，所得到的物理页号不是使用者决定的，此时是页帧映射的方式。

使用页帧内存管理方式时，一个逻辑段将会控制到多个物理页帧。data_frames 字段生效，被分配的页帧保存在 Map 中，逻辑段生命周期与分配到的物理页帧绑定。当逻辑段生命周期结束，相应页帧的回收方法被自动调用；使用段内存管理方式时，一个逻辑段只需要控制一个段实例，段号类型的 segment 对象生效，逻辑段所占用的物理段段号

实例与自身生命周期绑定。逻辑段生命周期结束时，物理段也自动调用回收方法被段分配器回收。

逻辑段的设计与程序编译、连接后形成的段结构相契合。然而程序在编译时无法知道自己将在内存中的确切位置，需要在分配资源后，才能通过页表获取虚实地址映射。因此，地址空间应该包含若干逻辑段与一张多级页表，如图所示：

```
pub struct MemorySet {
    page_table: PageTable,
    areas: Vec<MapArea>,
}
```

图 3-10 地址空间结构

构建一个应用程序的地址空间，核心步骤是为各逻辑段分配物理资源，并将虚拟页号与物理页号映射到地址空间的页表中，如图所示。

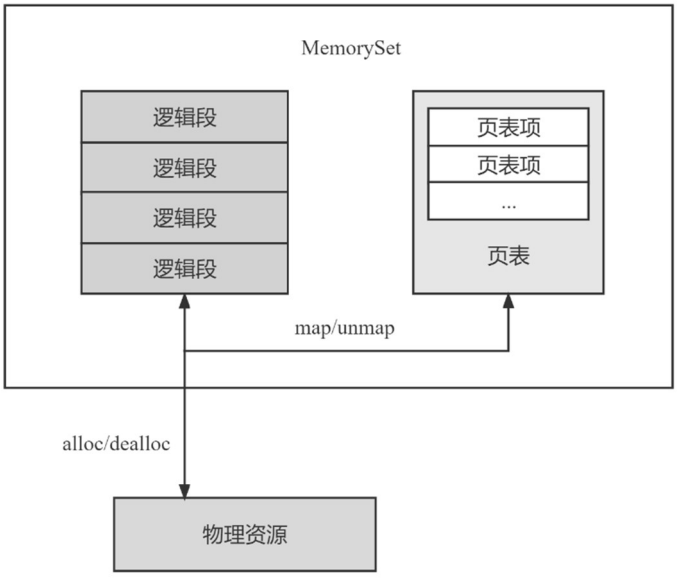

图 3-11 地址空间、逻辑段与页表的关系

当应用地址空间形成后，用地址空间中多级页表根页表的页号使能分页机制，则进入隔离的地址空间进行虚拟内存访问模式。

3.3 页帧分配与回收模块

页帧分配与回收模块使用了多种方式实现分页内存管理。

为了在发挥内存管理系统的教学作用，首先抽象出页帧分配器的基本行为。学习者可根据页帧分配器行为的抽象，进一步采用不同的基本数据结构和算法实现多种页帧管

理器。定义如图所示：

```
trait FrameAllocator {
    fn new() -> Self;
    fn alloc(&mut self) -> Option<PhysPageNum>;
    fn dealloc(&mut self, ppn: PhysPageNum);
}
```

图 3-12 页帧管理器行为

用 Rust Trait 定义页帧管理器的基本行为，类似于 JAVA 中的 Interface。FrameAllocator Trait 定义了页帧管理器的需要实现的方法，包括创建、分配、回收三个方法及其函数签名。

（1）new：构造方法，返回一个页帧管理器实例；

（2）alloc：页帧分配方法，规定了返回值是一个 Option 包裹的物理页号类型，因为页帧的分配可能成功，也可能失败；

（3）dealloc：回收方法，传入一个物理页号，在方法体中完成对页帧回收，没有返回值；

对 FrameAllocator Trait 的不同实现可以构建多种多样的页帧管理器。本课题中，一共实现了 3 种不同的页帧管理器，分别是栈页帧管理器、位视图页帧管理器、链表页帧管理器。

栈页帧管理器。使用了三个内部变量来完成管理页帧。

```
pub struct StackFrameAllocator {
    current: usize,
    end: usize,
    recycled: Vec<usize>,
}
```

图 3-13 栈页帧管理器

current、end 两个无符号整数共同标记了从未分配过的一段物理内存的起始页号和结束页号。recycled 向量保存了曾被分配过，而当前已回收的物理页号，是一个栈结构。初始化时，传入能被分配的物理内存的起始与结束物理地址，赋值 current 与 end。此时所有物理页帧尚未被分配，因此初始化的 recycled 向量为空向量。

分配策略：当分配物理页帧时，首先检查 recycled 栈是否为空。若不为空，表明有被分配过，而目前已回收的页帧，recycled 弹栈，将最近被回收的页帧分配出去。若

recycled 为空，表明当前没有已经被回收了的页帧，则只能从被分配过的页帧中分配，因此返回 current 当前的值，随后 current 加 1。

回收策略：传入被回收的物理页号。首先需要对传入的物理页号做检查，如果页号存在于 recycled 栈中，或该页号大于 current 的值，说明是一个错误的页号，因为无法回收一个已经被回收或从未被分配过的页帧，说明上层调用的时出现逻辑错误。如果通过检查，把传入的页号压入 recycled 栈，完成页帧的回收。

位视图页帧管理器。使用一张位视图来标记页帧的工作状况。

```
pub struct BitMapFrameAllocator {
    bitmap: Vec<bool>,
    start_ppn: usize,
    end_ppn: usize,
}
```

图 3-14 位视图页帧管理器

位视图分配器定义了一个数组，数组中的每个元素都是一个布尔值，用于标记所有页帧的工作状态。同时保存了所有物理页帧的起始物理页号与结束物理页号，可以通过起始物理页号+位视图中的索引，得到位视图中某一位对应物理页帧的物理页号。初始时，传入起始与结束物理页号，转换得到物理页帧数量，然后对位视图设置物理页帧数的初始位。

分配策略：找到位视图中第一个值为 0 的元素，获得其下标。返回下标+起始物理页号，即成功被分配的页号。

回收策略：把要回收的物理页号转换为位视图中的索引，检查该索引所在位是否为 0。如果是，表明尝试回收一个空闲的页框，报错退出。否则将该索引中指向的值由 1 改为 0，完成页帧的回收。

链表页帧管理器。链表数据结构的优势在于添加、删除节点只需 O(1)的时间复杂度，可用于页帧分配算法中。Rust 提供的链表在标准库 std::collections 中，而搭建在裸机上的 Rust 内核无法使用基于操作系统服务的 Rust 标准库，因此需要先自定义链表数据结构 List，如图所示。

```rust
type NextNode<T> = Option<Box<Node<T>>>;

#[derive(Clone, Debug)]
struct Node<T> {
    value: T,
    next: NextNode<T>,
}

impl<T> Node<T> {...}

#[derive(Clone, Debug)]
pub struct List<T> {
    len: usize,
    next: NextNode<T>,
}

impl<T> List<T> {...}
```

图 3-15 链表数据结构

在此基础上，链表页帧分配器只需要维护一个节点值为物理页号的链表，如图所示。

```rust
pub struct LinkedListAllocator {
    list: List<usize>,
}
```

图 3-16 链表页帧分配器

初始化时，传入可分配物理内存的起始页号与结束页号，根据起始与结束页号将所有能分配的页号生成链表节点，依次插入链表中，完成分配器的初始化。

分配策略：取出链表最后一个节点，返回节点保存的物理页号。

回收策略：传入的物理页号，做合法性检查后，为该页号生成新节点，添加新节点至链表末尾。

最终，无论选用上述何种具体的页帧管理器，页帧分配与回收子模块暴露出两个接口：frame_alloc 与 frame_dealloc，如图所示：

```
/// allocate a frame
pub fn frame_alloc() -> Option<FrameTracker> {...}

/// deallocate a frame
fn frame_dealloc(ppn: PhysPageNum) {...}
```

图 3-17 页帧管理子模块对外接口

这两个接口页帧分配与回收子模块的对外接口，与各分配器内部的分配与回收方法不一样。对于分配页帧，使用者调用 frame_alloc()，接口内部通过调用当前被使能的页帧分配器的分配接口，返回页帧实例。外界不需要取得当前工作的页帧分配器，再调用其方法进行资源的分配，降低模块间的耦合度。

页帧的回收不是一个公共方法。原因在于页帧类型 FrameTracker 已经实现了 Drop 特性。当页帧类型实例生命周期结束，drop 方法被自动调用，进而调用页帧回收接口 frame_dealloc，回收相应页号所对应的物理页帧。因此，外界或资源使用方不需要显式调用页帧回收接口，当资源使用者获得了页帧资源，只需要将页帧资源与使用者生命周期绑定，在使用者生命周期结束时，其占用的页帧资源也会随之自动释放，无需担心使用者未释放而产生的内存泄漏。

3.4 段分配与回收模块

为了增加内存管理系统的资源分配方式，在教学中给学习者更多种选择以及策略的实现，在页帧管理器基础上抽象出段（Segment）分配器。整体设计如图所示。

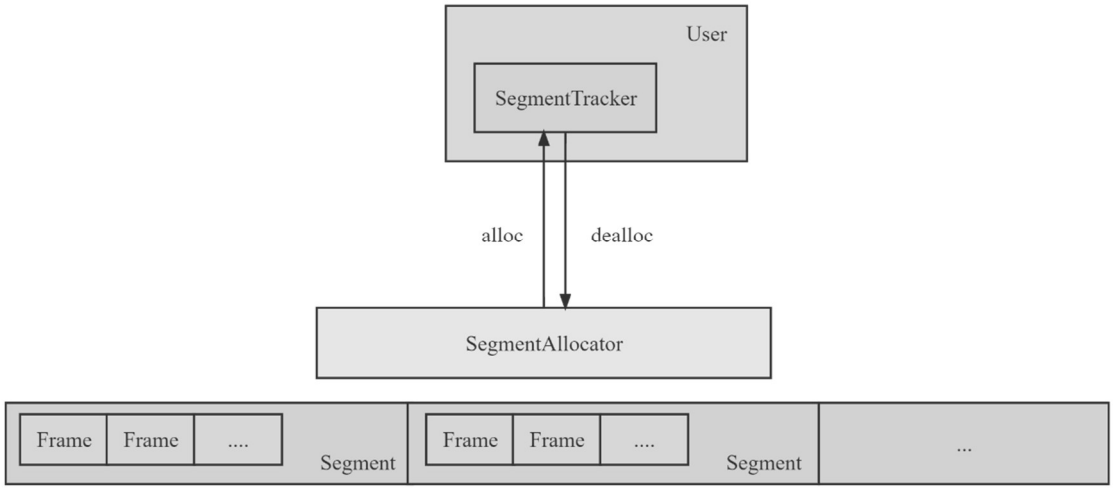

图 3-18 段分配器整体设计

段是一块连续的物理内存空间，控制着一组连续的页帧。段管理器以段为单位进行

资源的分配和回收。段管理器对外暴露出分配接口，返回段号实例。回收段号的接口则在段实例生命周期结束后自动被调用。当不开启段内存管理时，逻辑层对申请内存空间前需要逐页拆分，依次调用页帧分配接口。当开启了段内存管理后，可以调用段分配器的接口进行整段划分。

下面给出段分配器类型的定义。进行段的分配和回收操作时，需要用到段的大小、段号、段是否被占用三个信息。因此段分配器包含一个以段对象为元素的向量，一个标志各段工作状态的位视图。

```
pub struct SegmentAllocator{
    segments: Vec<Segment>,
    bitmap: Vec<bool>,
}
```

图 3-19 段分配器类型

段管理器同页帧管理器一样定义了基本的行为，包括创建，分配，回收。学习者可根据段管理基本行为，使用不同数据结构实现段管理器。

段管理器初始化时控制了所有页帧的生命周期，把以页为单位的物理资源从页帧模块移交至段管理器。主要原因在于段的分配种，各段的大小固定，只需在初始化时生成好所有段的实例。

段分配与回收时，段分配器不需要反复调用页帧分配器的接口生成和释放段实例，而只需要维护自己内部的生成好的段实例即可。这样设计可以提高段分配器的效率。与页帧分配器不同，页帧分配器每次分配新的页帧都会生成新的页帧实例。由于它只是物理页号的一个简单的封装类，构建一个页帧实例的代价较小。但是段实例内部需要维护一个页帧数组，不适合反复地创建与释放。

抽象出段类型结构，可以采用不同的分配策略进行段内存分配。使用首次适应算法，从空闲段的第一个开始查找，把最先能分配需求的段分配出去。该策略倾向于分配低地址部分的空闲分区，保留了高地址部分的空闲分区。

3.5 接口设计

本小节从外部角度介绍内存管理系统所提供的功能及其相应的实现接口。

（1）内核在启动时通过 new_kernel()接口获得内核地址空间，使内核能进入虚存访问模式。

（2）应用的地址空间获取，则通过调用内存管理系统的 from_elf()接口，接口内通

过传入应用编译完成后形成的 elf 文件，为应用生成地址空间并返回。对于地址空间具体的设计，例如各逻辑段、用户栈、内核栈、任务上下文等在虚存中的位置，需要为进程在运行时的陷入、切换等操作做考虑。

（3）从物理访存模式到开启虚拟访存模式，以及在应用与应用，应用与内核进行切换时，需要为当前地址空间的根页表构建一个符合 satp 寄存器的格式。内存管理系统中的 token()接口能够为当前地址空间返回一个写入 satp 寄存器的值，以让 MMU 自动完成虚实地址的转换。

（4）内存管理系统的可视化接口 visible()。通过对它的调用可以获得当前所选择的内存分配器的工作状态，以简单、直接的方式观察到在运行时内存分配器各数据结构的变化。

从外部的视角观察内存管理系统提供的功能，以及外界对于这些接口的具体使用方式，如图所示。

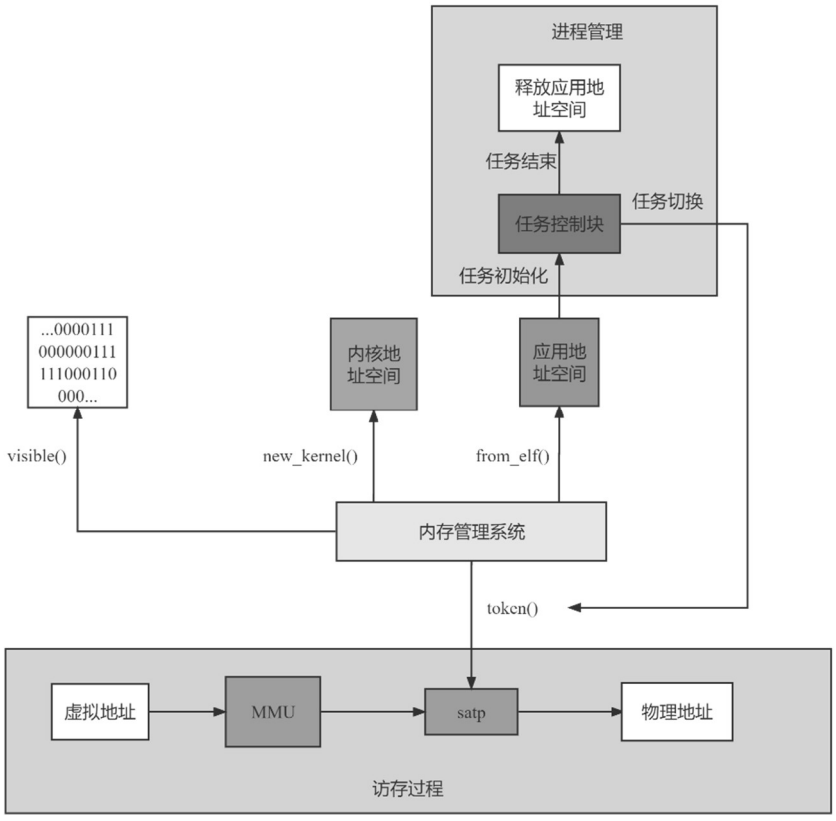

图 3-20 内存管理系统接口设计及应用

内核地址空间生成接口 new_kernel()在内核运行之初被调用并使能。应用地址空间生成接口 from_elf()在应用被进程管理系统初始化时调用，在应用执行完毕时被释放。token()接口提供了使能当前地址空间、写入 satp 寄存器的值。在进程管理模块进行任务

切换时也需要进行地址空间切换，此时也需要调用 token()完成对 satp 寄存器的更改。可视化接口 visible()提供了当前内存分配器的工作状况的可视化输出，外界能调用此接口对内存分配器进行直观的观察。

3.5.1 内核地址生成接口

CPU 控制权刚转交给 Rust 内核时，使用实存访问方式。通过调用内存管理系统提供内核的地址空间生成接口 new_kernel()，获得内核的地址空间实例，将其中的根页表地址写入 stap 寄存器后，内核进入虚拟访存模式。对于内核的地址空间设计如下所示。

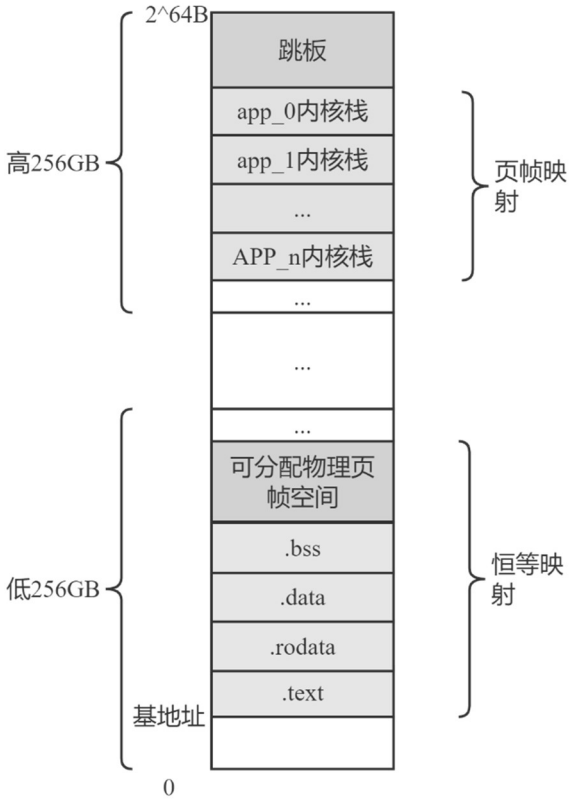

图 3-21 内核地址空间设计

根据页表的定义，虚拟地址空间一共有 2^64B 的大小，而实际物理资源为 8MB。

在内核地址空间的低 256GB 中的有效部分，逻辑段使用恒等映射，在页表中该部分的虚拟页号与物理页号一致。从物理资源的层面看，这部分是计算机所拥有的内存大小。其中.bss、.data、.nodata、.text 部分是内核程序所占用的空间，“可分配物理页帧空间”部分实际则是内核占用的空间以外的内存区域，是可被内存管理系统分配与回收的区域。

高 256GB 的地址空间使用的是页帧映射，这部分地址空间中的段最终需要调用物理资源分配接口，从蓝色部分的内存资源中为分配物理资源。因此他们的虚拟地址图中

高地址，物理地址是蓝色部分中的某一物理页所在地址。

高 256GB 中主要保存了 app 的内核栈，这些栈会在对应 app 这会在陷入内核时使用。而跳板页面则是保存了内核与应用程序之间陷入与返回的代码。这部分代码实际的物理位置在内核的.text 段中。

接口 new_kernel()返回上述所示的内核地址空间，它是地址空间类型的实例，包含了内核的各逻辑段的集合，以及内核地址空间所使用的多级页表。

3.5.2 应用地址空间生成接口

应用地址空间生成接口返回应用地址空间，进程管理模块在装载应用时调用。为了满足应用运行时的陷入、切换等操作，应用的地址空间设计如图所示。

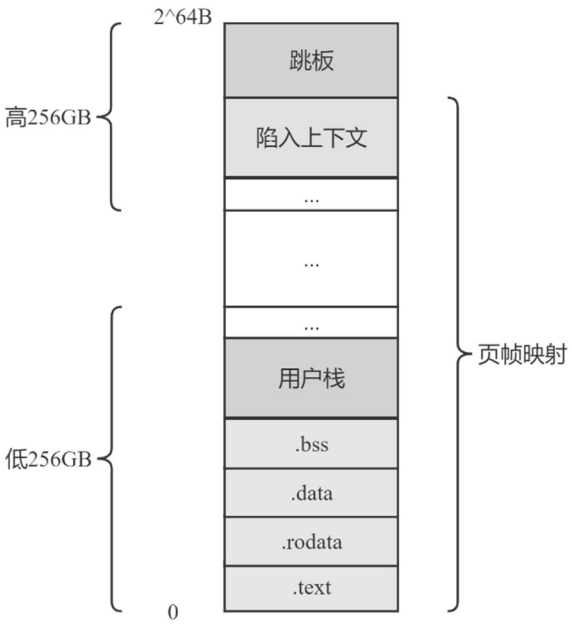

图 3-22 应用地址空间

应用所占用的空间，在物理层面，都会被分配在内核地址空间“可分配的物理空间”部分中，因此应用地址空间中均采用页帧映射。因为使用的是虚拟地址，低 256GB 中可以从 0 开始放置各逻辑段。在高 256GB 中，除了跳板区域，另一个存放的是应用陷入时的上下文。

进程模块初始化后，其将会调用内存系统的此接口，返回上述的应用地址空间，并将其作为任务控制块的一部分。当切换应用执行时，则需要从地址空间中获取相应的根页表页号，写入 stap 寄存器中完成地址空间的切换；当应用执行完毕，随着任务控制块的释放，地址空间也会被释放，此时内存管理系统中的回收接口将在页帧、段等实例生

命周期结束时被自动调用。

3.5.3 地址空间指定接口

当构建了包含根页表的物理地址和分页模式标志的值，写入 satp 寄存器，即完成地址空间的切换与分页模式的使能。底层 MMU 在进行地址访问时能根据 satp 寄存器中所指向的页表自动完成地址转换。

token()接口提供了这样的功能。外部对它的调用将获得一个能够写入 satp 的值，写入后，当前地址空间将会被切换为该值中的根页表地址所处的地址空间。

3.5.4 内存系统可视化接口

为了方便测试与验证内存管理系统的正确运行方式，设计了可视化接口 visible()提供给外部调用。

对于所有物理资源分配器，都必须实现一个可视化的方法。而内存管理系统暴露给外界的函数中，将从配置文件中判断当前使用分配器，并调用其可视化接口，从而提供一种直接、清晰的方式令外界了解分配器各数据结构的状态。

3.6 本章小结

本章自底向上详细介绍了内存管理系统各部分的设计与实现。

（1）内存管理系统提供的功能与其他模块的联系，以及其内部的整体框架设计。

（2）内存管理系统中关键的数据结构，包括物理资源数据结构、页表结构、逻辑数据结构。

（3）两种内存资源管理器，页帧管理器与段管理器。两种管理器内部基于策略与算法实现对物理资源的分配与回收。

（4）内存管理系统对外的接口设计，包括内核、应用地址空间的生成接口，与底层 satp 寄存器交互接口、可视化接口。

第 4 章　内存管理系统测试与分析

4.1 测试配置文件

内存管理系统定义了若干种物理资源的分配方式，交由用户使用时，可以通过内核配置文件中的选项来决定当前应用程序使用何种内存管理方式。在配置文件中，对于页帧的分配方式，可以从栈、位视图、链表在三种页帧管理器类型中选择。而另一个配置则是用于指定上层使用者的内存分配器，有页与段两种。

```
// Choose A specific Frame Allocator
#[cfg(feature = "stackframe_allocator")]
pub type FrameAllocatorImpl = StackFrameAllocator;
#[cfg(feature = "bitmap_allocator")]
pub type FrameAllocatorImpl = BitMapFrameAllocator;
#[cfg(feature = "linkedlist_allocator")]
pub type FrameAllocatorImpl = LinkedListAllocator;


// Choose A specific Memory Allocator for MapArea
#[cfg(feature = "frame_allocator")]
pub const MEMORY_ALLOCATOR:MemoryAllocator = MemoryAllocator::Frame;
#[cfg(feature = "segment_allocator")]
pub const MEMORY_ALLOCATOR:MemoryAllocator = MemoryAllocator::Segment;
```

图 4-1 内存分配器配置文件

4.2 页帧、段分配器接口功能测试

对段与页管理器的分配与回收接口进行测试，确保当分配接口被调用时，分配器中的维护的数据结构能正常标记出被分配的资源，而资源生命周期结束时，回收接口能被自动调用，分配器内部也能将相应的资源标记为回收。

页帧分配器测试以位视图分配器为例。在测试程序中，多次调用分配接口，并将返回的页帧实例压入向量中，查看页帧分配器内部数据结构状态。随后将向量清空，倘若页帧生命周期与向量绑定，此使页帧实例生命周期也会结束，回收接口会被自动调用，页帧分配器则完成回收。测试过程的中间信息输出到控制台，如图所示。

对于段分配器的分配与回收接口测试也使用类似方法。对外界而言，使用者通过调用接口获得页号/段号的封装实例，而这些页号/段号实例都是在使用者被编译器回收时，被分配器回收。

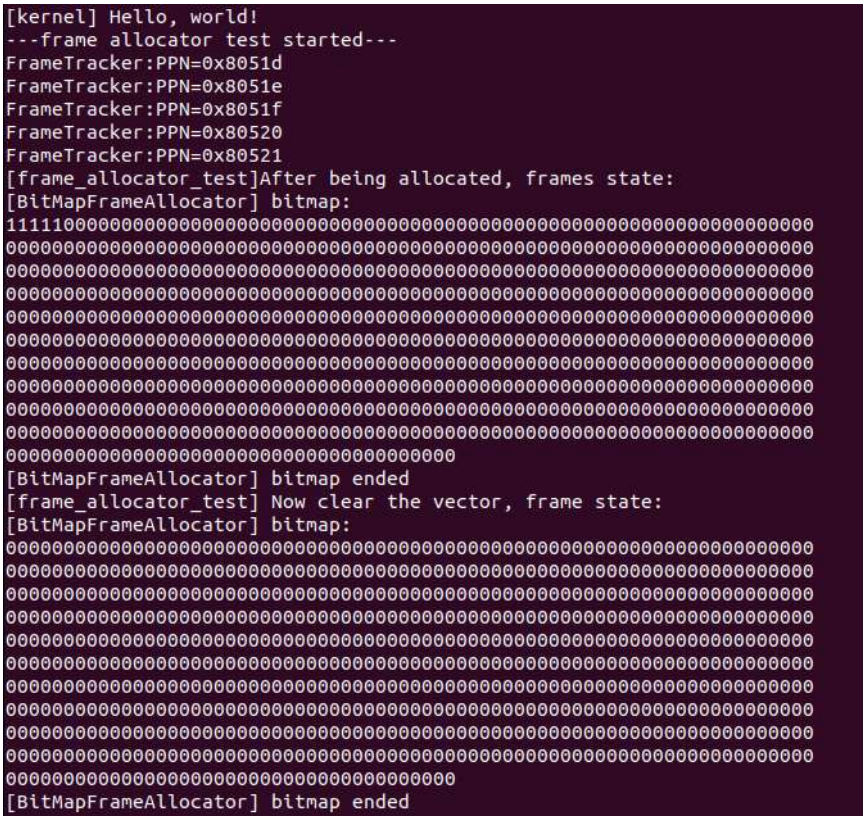

图 4-2 页帧分配器接口测试

对段分配器进行可视化时，每个段以三元组的方式输出，分别是段号、段大小、段分配情况。从结果可以看出，使用首次适应的段分配算法，连续调用 5 次段分配接口，需求分配段大小为 3 的段，通过图中展示的被分配的段，符合首次适应算法的特征。在向量被清空后，所有被分配出去的段被自动回收，表明生命周期按预期绑定到了使用方。

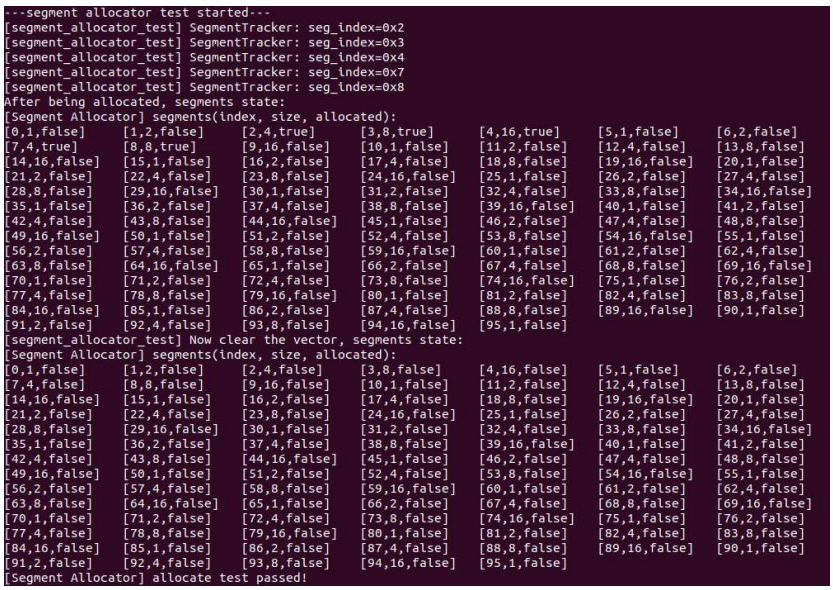

图 4-3 段分配器结果测试

通过对两种分配器分配、回收接口的测试，验证内存管理系统内部的各分配器的所

提供的分配、回收接口正常工作，以预期的策略和算法处理资源的分配请求。同时内存管理系统中的 RAII 思想的实现也是成功的，资源实例的生命周期被成功绑定至调用者中，当调用者生命周期结束，内存管理系统中的回收接口也被自动调用，资源随之释放。

4.3 内存管理系统功能测试

本节对内存管理系统提供的接口进行测试与验证。

4.3.1 内核地址空间测试

对内核空间的测试中，测试程序输出内核中被映射的各逻辑段，在地址空间生成完毕后，检查内核地址空间中根页表的变化。然后选择逻辑段中的一个虚拟页号，通过手动查页表的方式检索多级页表，检查最后一级页表是否存在该虚拟页号映射的物理页号。最后，测试获取不同段中虚拟页对应的物理页帧，检查物理页帧的访问权限是否与逻辑段定义的一致，若不一致，则调用 Rust 中的 assert_eq!宏直接报错退出。通过以上方式能较为全面地验证内核地址空间是否与预期一致完成资源分配，以及页表是否已记录虚实映射。

```
----------Generating kernel space----------
.text [0x80200000, 0x80207000)
mapping .text section
.rodata [0x80207000, 0x8020a000)
mapping .rodata section
.data [0x8020a000, 0x8020b000)
mapping .data section
.bss [0x8020b000, 0x8051c000)
mapping .bss section
physical memory [0x8051c000, 0x80800000)
mapping physical memory
[BitMapFrameAllocator] bitmap:
11111110000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
[BitMapFrameAllocator] bitmap ended
----------Kernel space has been generated----------
---------- Printing Root page ----------
[PTE in root] Flags:1, PPN:525599, Offset:2
[PTE in root] Flags:1, PPN:525597, Offset:511
---------- Root page ended ----------
Testing vpn mapping
Successfully access the physical frame, given VPN = 524803
[PTE] Flags:75, PPN:524803
Test finished
Testing privilege flags of map area in kernel space
Test passed!
```

图 4-4 内核地址空间测试输出

测试结果显示，内核空间被分配到了若干页帧中。随后内核空间的根页表出现了两条页表项。当时根据一个预期中存在的虚拟页号手动查询多级页表中的页表项时，能够

检索到存在的页表项，并取出其中的物理页号。随后对内核空间各逻辑段的权限检查中没有出现报错现象，说明内核空间逻辑段的映射符合预期。

4.3.2 应用地址空间测试

编写若干测试应用程序，在应用程序编译完成后由进程管理模块通过调用内存管理系统的接口进行装载。应用程序内部对于自身被分配到的物理地址是透明的。通过地址空间的构造和分页机制的使能，被编译后的应用程序所有访存都会被 CPU 中的内存管理单元自动转换。测试将会展示应用程序从被分配到运行结束退出被回收的过程。

当选用页帧分配器时，以位视图分配器为例。

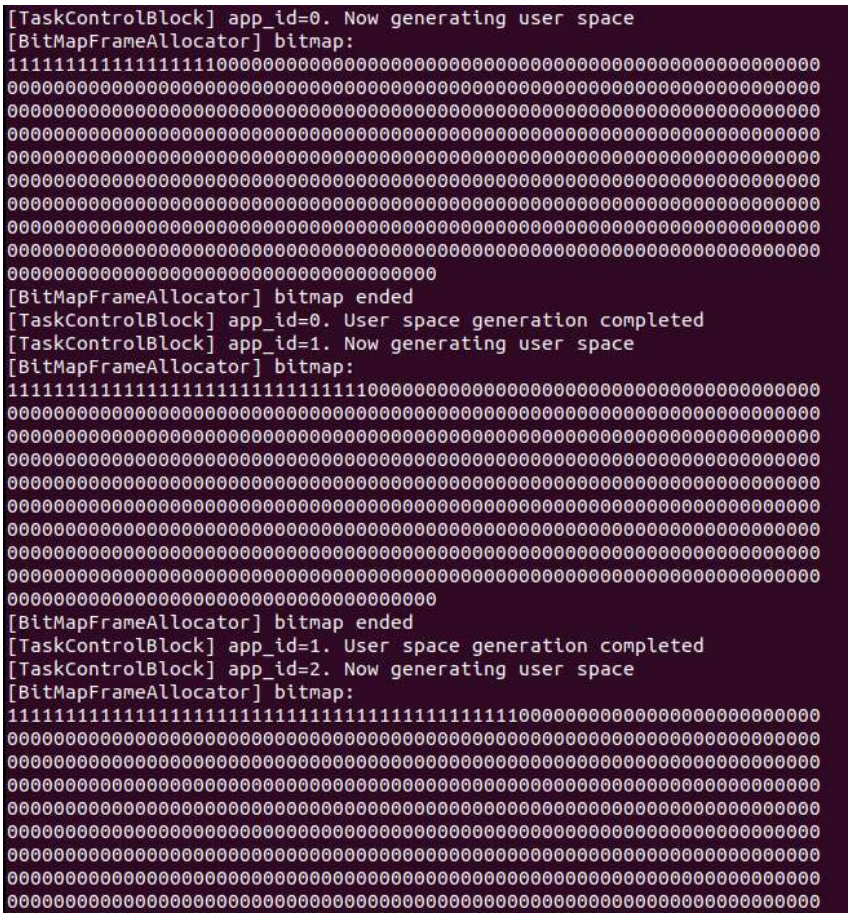

图 4-5 应用地址空间分配测试

从结果可以看出，当调用分配应用空间接口后，内存管理系统内部发生了变化，若干页帧按需求被分配。

应用程序在执行完毕后依次退出，其所占用的物理资源也被逐一回收。当最后一个应用程序执行完毕，页帧位视图显式只剩下内核的页表占有部分页帧，其余页帧均已空闲。

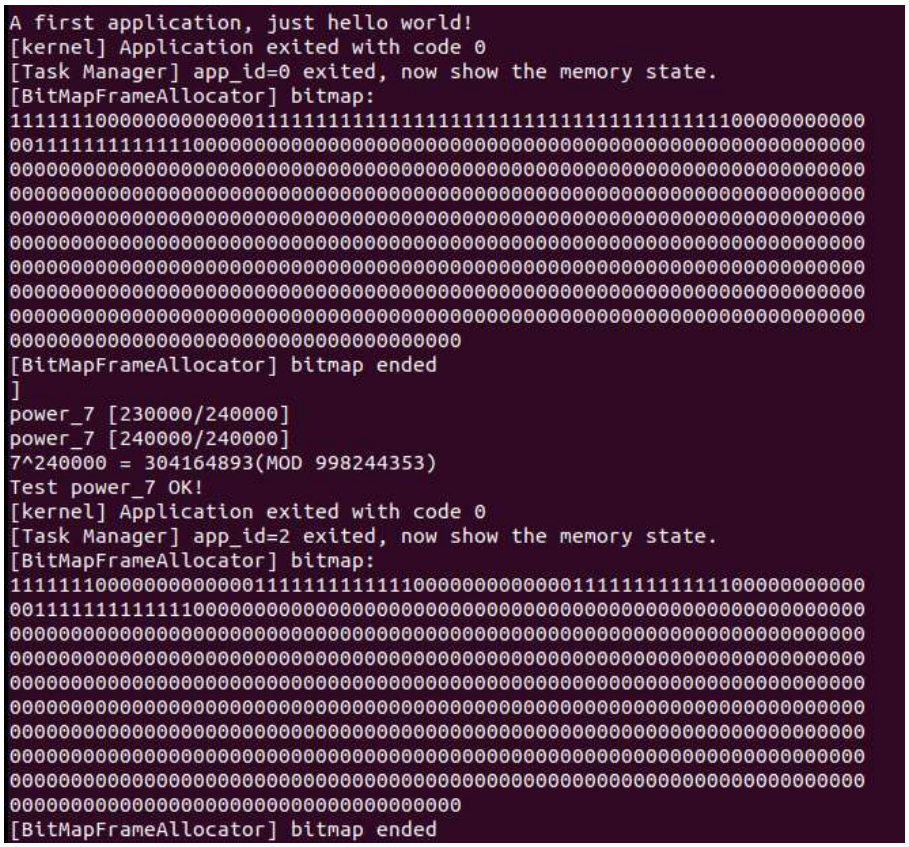

图 4-6 部分应用程序执行完毕，其页帧资源被回收

图 4-7 最后一个应用程序执行完毕后的页帧位视图

使用段分配器对应用程序的段分配与回收进行测试。

段分配器选用首次适应方法进行段的分配与回收，从测试中可以看出应用程序被分配的段所在位置，以及应用程序退出后段分配器中的状态。

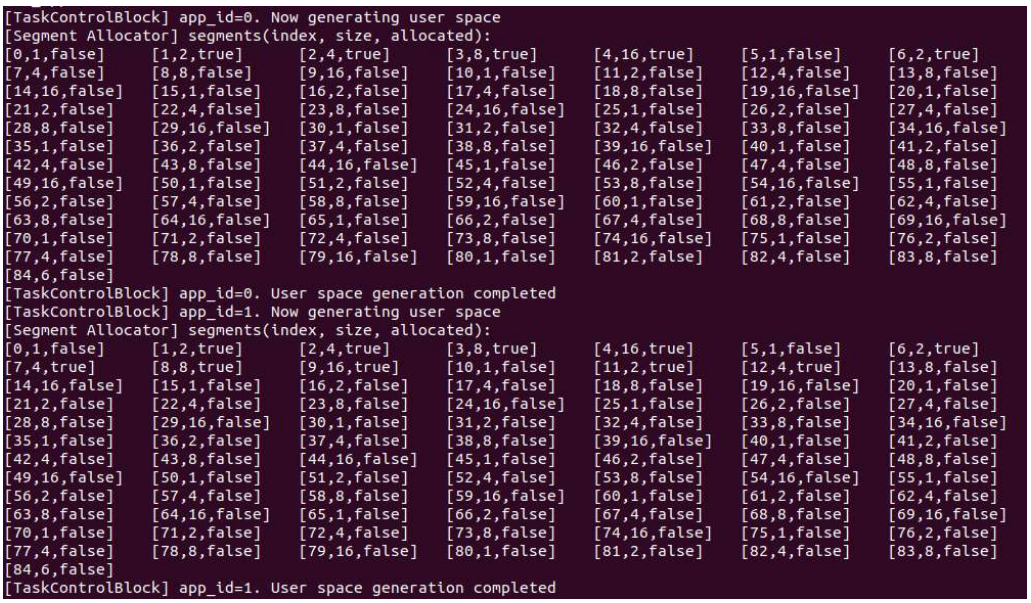

图 4-8 应用程序段分配测试

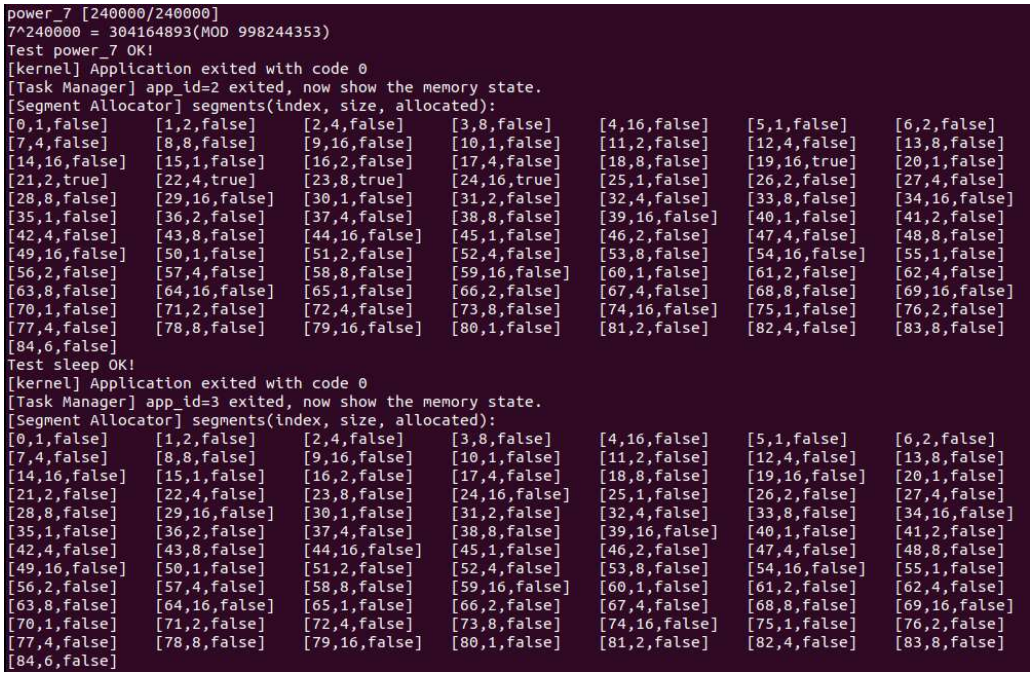

图 4-9 应用程序段回收测试

4.4 内存管理模式切换测试

在内核中使用 Rust 中条件编译的方式[34]包裹着内存管理系统中不同的配置变量，在编译运行内核前，将会读取当前环境变量的参数进行编译。最终，外界可以通过对环境变量的赋值完成对内核中不同内存管理方式的选用。

```
aaron@aaron-virtual-machine:~/CLionProjects/graduate_os/os$ export ALLOCATOR=frame_allocator
aaron@aaron-virtual-machine:~/CLionProjects/graduate_os/os$ export FRAME_ALLOCATOR=stackframe_allocator
aaron@aaron-virtual-machine:~/CLionProjects/graduate_os/os$ make run
(rustup target list | grep "riscv64gc-unknown-none-elf (installed)") || rustup target add riscv64gc-unknown-none-elf
riscv64gc-unknown-none-elf (installed)
cargo install cargo-binutils --vers =0.3.3
     Ignored package `cargo-binutils v0.3.3` is already installed, use --force to override
rustup component add rust-src
info: component 'rust-src' is up to date
rustup component add llvm-tools-preview
info: component 'llvm-tools-preview' for target 'x86_64-unknown-linux-gnu' is up to date
(which last-qemu) || (rm -f last-k210 && touch last-qemu && make clean)
make[1]: 进入目录“/home/aaron/CLionProjects/graduate_os/os”
make[1]: 离开目录“/home/aaron/CLionProjects/graduate_os/os”
make[1]: 进入目录“/home/aaron/CLionProjects/graduate_os/user”
```

图 4-10 赋值环境变量后编译内核

```
[TaskControlBlock] app_id=0. Now generating user space
[StackFrameAllocator] current=525687, end=526336
[StackFrameAllocator] recycled stack:
[StackFrameAllocator] recycled stack ended
[TaskControlBlock] app_id=0. User space generation completed
[TaskControlBlock] app_id=1. Now generating user space
[StackFrameAllocator] current=525700, end=526336
[StackFrameAllocator] recycled stack:
[StackFrameAllocator] recycled stack ended
[TaskControlBlock] app_id=1. User space generation completed
[TaskControlBlock] app_id=2. Now generating user space
[StackFrameAllocator] current=525713, end=526336
[StackFrameAllocator] recycled stack:
[StackFrameAllocator] recycled stack ended
[TaskControlBlock] app_id=2. User space generation completed
[TaskControlBlock] app_id=3. Now generating user space
[StackFrameAllocator] current=525726, end=526336
[StackFrameAllocator] recycled stack:
[StackFrameAllocator] recycled stack ended
[TaskControlBlock] app_id=3. User space generation completed
```

图 4-11 栈式页帧分配器生效

上图的测试中，将环境变量定义 ALLOCATOR 与 FRAME_ALLOCATOR 分别赋值为 frame_allocator 与 stackframe_allocator，即使用页帧分配器器对逻辑段进行内存分配，页帧分配器内部使用栈页帧管理器实现。在内核编译后，装载应用程序的过程中可见栈页帧分配器已被成功使能。对于段分配器以及其他页帧分配器也可以用相同的方式，在 shell 中指定。

4.5 性能分析

本小节对不同内存管理器进行性能测试与分析。

Rust std 提供时间相关的库，但由于内核在裸机上运行，std 库被禁用。RISC-V 架构中有一个计数器用来统计处理器上电以来经过的的时钟周期，该计数器保存在一个 mtime 中，可以通过读该寄存器获得当前经历的时钟周期。

4.5.1 页帧分配器器性能分析

本节分析不同页帧管理器的分配、回收接口效率。

页帧分配器使用不同数据结构与策略，随着分配接口被不断调用，内部数据结构持

续改变，其工作性能也可能发生变化。下图展示了不同页帧分配器的分配接口随着被调用次数的增加，分配页帧的时间开销变化。

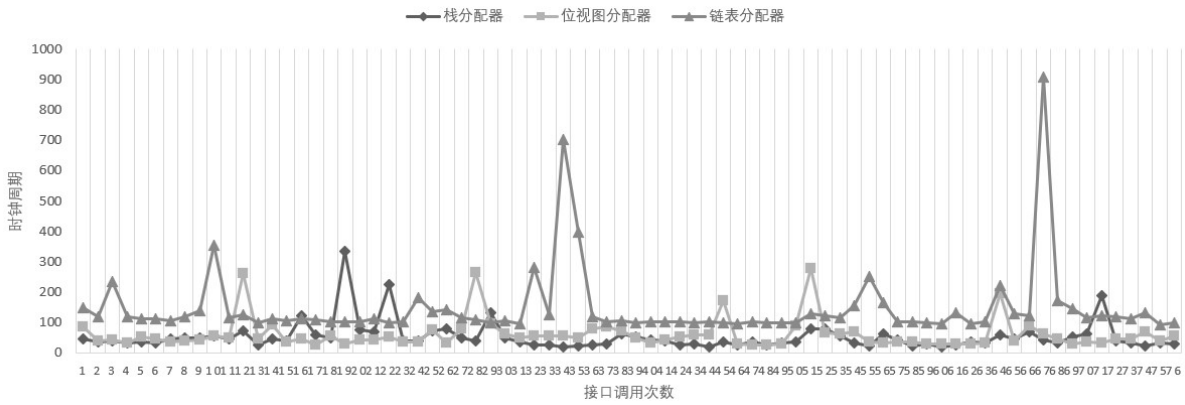

图 4-12 页帧分配器分配接口性能

如图所示，栈分配器与位视图分配器性能表现接近，偶尔产生波动。链表分配器的时间开销则比栈、位视图分配器整体更大。分析主要原因在于链表分配器使用了自定义的链表数据结构，创建、删除链表节点通过自定义的链表类进行，导致开销比基于原生 Vector 类型的栈与位视图分配器更大。

页帧分配器的回收接口性能如图所示。

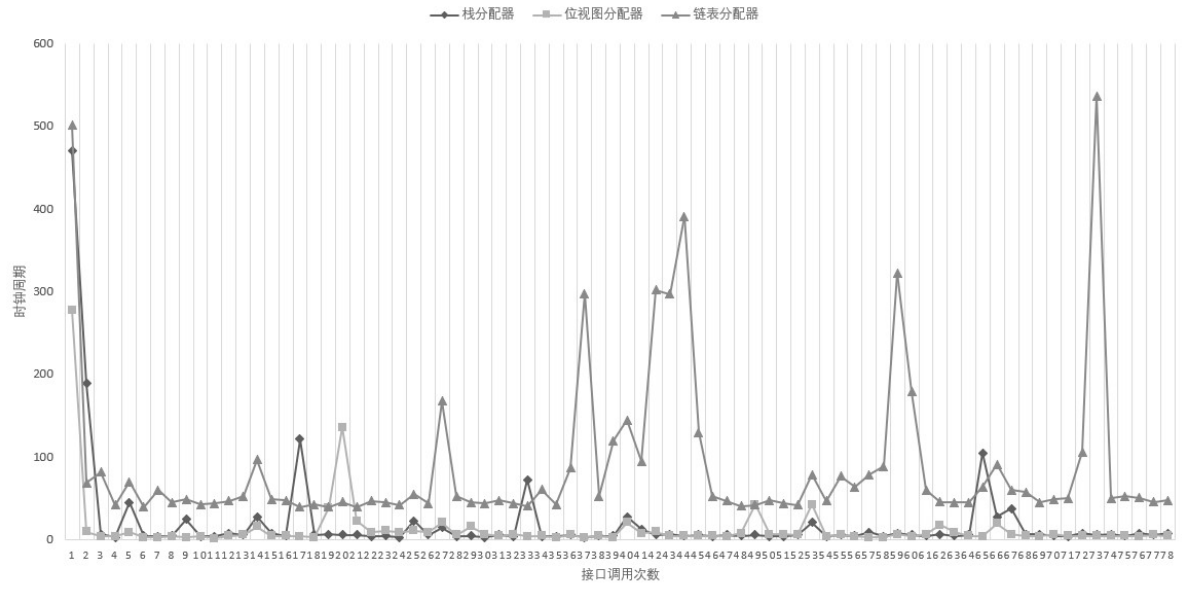

图 4-13 页帧分配器回收接口性能

各分配器的回收接口性能对比与分配接口相近。链表分配器开销较高。位视图与栈分配器回收效率相近。而在多次调用的开销统计中可观察到位视图分配器的性能相比栈分配器更为稳定。原因在于位视图回收时以位进行读写，相比栈分配器将页号压栈的操

作更为轻量。

分析各分配器的空间开销，从初始化与执行策略的角度分析，栈分配器初始阶段所需空间最小，占用两个整形与一个空 Vector。其次是位视图分配器，起始阶段维护了两个正 i 选哪个与一个位视图，位视图元素个数为页帧数量，元素类型为布尔值。链表类型初始阶段即维护了元素个数为页帧数量，元素类型为物理页号的链表，占用空间最大。

随着程序运行，被回收的页号逐渐压入栈分配器的 Vector 中，其所占用的空间逐渐与链表分配器接近。而位视图分配器不会随着分配与回收的运行占用更多空间，其空间开销是稳定且最小的。

4.5.2 地址空间生成效率分析

本节使用不同分配器或分配策略对比生成应用地址空间的时间开销。

使用不同的分配器，为 6 个测试程序生成地址空间，时间开销如图所示。

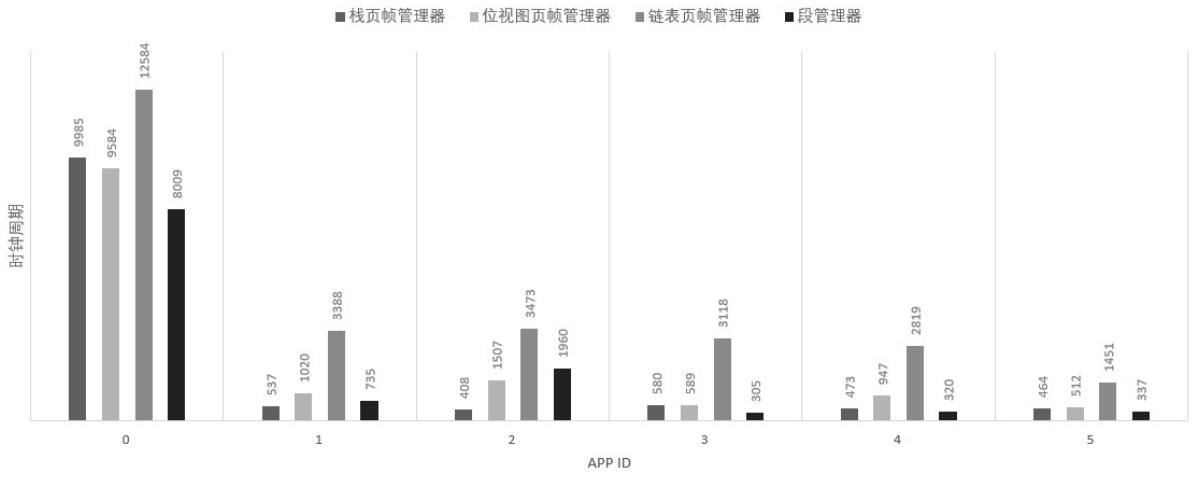

图 4-14 应用地址空间生成时间开销

对比了栈页帧管理器、位视图页帧管理器、链表页帧管理器、段管理器生成 6 个应用地址空间的时间花费。链表页帧管理器由于使用自定义类型，效率最低。位视图分配器比栈分配器开销略大，开销原因在于位视图分配器在分配时额外需要检索空闲页帧的操作。

段分配器在多次测试中均有超越各页帧分配器的表现，表明在为逻辑段进行资源分配时，物理段的抽象能为逻辑段带来更小开销，绑定在逻辑段中的物理资源从一组页帧转变为一个段号，不仅节省了存储空间，也在分配时减少了堆内存读写的操作。

4.6 本章小结

在本章中自底向上测试了内存管理系统的各个功能。

（1）定义了各管理方式的配置文件。

（2）编写了测试函数验证物理资源管理器层面的接口是否能以符合预期的方式管理资源。

（3）调用内存管理系统接口为内核与应用程序生成地址空间，以用多种方式测试、验证分配与回收的有效性和正确性。

（4）通过条件编译的方式，用户可以在外部配置不同的内存管理方式的参数，并编译运行使用了指定内存分配器的内核。

（5）测试与分析了各内存管理器的接口性能。

第 5 章 总结与展望

5.1 总结

本课题基于 Rust 语言设计与开发了操作系统中的内存管理系统。能够在教学上帮助学习者在理解繁多的操作系统理论、概念、算法的同时，利用真正的操作系统进行开发与实践。

（1）定义了内存管理的功能、在内核中的作用，以及与其他模块的交互方式。设计了内存管理系统内部的实现框架。

（2）底层设计了地址、页号、逻辑段等基本数据结构，支持内存管理系统上层不同策略的应用。设计了存放虚实地址映射的页表，支持在 SV39 机制基础上开启虚拟内存访问模式。

（3）设计了基于页与基于段的两种内存管理方式，在页与段两种内存管理器中均实现多种内存管理策略。抽象出了分配器的基本行为，能引导学习者使用不同策略实现更多种内存分配与回收方式。

（4）设计了内存管理系统的对外接口。内核、应用地址空间生成接口获得隔离的地址空间，并完成所需资源的分配；构建 satp 寄存器接口能使 CPU 开启虚拟内存访问模式，或进行地址空间的转换；可视化接口提供了清晰的输出方式，能观察工作中的内存管理器各数据结构的变化，学习者能直观地通过输出信息验证内存管理策略的有效性。

（5）设计了外部配置文件。通过对配置文件的设置，使用者无需更改系统内部源码也可从外部指定内存管理系统的资源分配与回收模式，提高系统的易用性与便利性。

（6）完成对内存管理系统及其各子模块接口的测试与验证。已实现的多种内存分配与回收方式能生效并符合预期工作。学习者在实现不同内存管理策略后，同样能进行直观、方便的测试。

5.2 展望

本课题基于 Rust 实现了一个能在教学中有指导作用的内存管理系统，而在后续的开发中，依然有扩展与改进的空间。

（1）定义了比物理内存空间更大的虚存空间，而当前暂时没有将内存与外存联系起来，真正占用空间比物理内存大的应用程序未做到无感知地运行。现代操作系统中可

以通过交换等虚存管理手段实现，这是当前内存管理系统需要扩展的功能。

（2）在使用者选择了不同的内存管理方式后，内核需要重新编译以让内存管理系统应用新的管理器或策略。目前无法做到在应用程序运行时动态选择不同的内存分配器。达到这个目标需要更多的知识与技术支持，同时需要重新设计内存管理系统的内部架构。而这样的目标也可以让学习者更为生动地感受不同内存管理方式带来的变化，是当前内存管理系统值得探索与扩展的方向。

参考文献

[1] Ritchie D M, Thompson K. The UNIX time‐sharing system[J]. Bell System Technical Journal, 1978, 57(6):1905-1929.

[2] Shimojima T, Teramoto M. V60 real-time operating system[J]. Microprocessing and Microprogramming, 1987, 21(1):197-204.

[3] 梅宏, 郭耀. 面向网络的操作系统——现状和挑战[J]. 中国科学(信息科学), 2013, 43(3):303-321.

[4] Nisan N, Schocken S. The elements of computing systems: building a modern computer from first principles[M]. Cambridge, Massachusetts: MIT press, 2021.

[5] Klabnik S, Nichols C. The Rust Programming Language (Covers Rust 2018)[M]. San Francisco: No Starch Press, 2019.

[6] 刘素娇. C语言的内存漏洞分析与研究[J]. 电脑编程技巧与维护, 2019(6):152-153.

[7] 柳崧轶. 基于C语言的程序安全性分析[D].吉林大学, 2007.

[8] Fong P W L. Reasoning about safety properties in a JVM-like environment[J]. Science of Computer Programming, 2007, 67(2):278-300.

[9] Pryce, Dave. 80486 32-bit CPU breaks new ground in chip density and operating performance[R]. California: Intel Corporation, 1989.

[10] Asanović K, Patterson D A. Instruction sets should be free: The case for risc-v[R]. California: EECS Department, University of California, Berkeley, 2014.

[11] Greengard S. Will RISC-V revolutionize computing?[J]. Communications of the ACM, 2020, 63(5):30-32.

[12] Jung R, Jourdan J H, Krebbers R, et al. RustBelt: Securing the foundations of the Rust programming language[J]. Proceedings of the ACM on Programming Languages, 2017, 2(POPL):1-34.

[13] Weiss A, Gierczak O, Patterson D, et al. Oxide: The essence of rust[EB/OL]. https://arxiv.org/abs/1903.00982, 2019-3-3.

[14] Qin B, Chen Y, Yu Z, et al. Understanding memory and thread safety practices and

issues in real-world Rust programs[C].Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. New York: ACM, 2020: 763-779.

[15] 李晓英, 刘敏. 基于RISCV-32I的微处理器的设计与实现[J]. 内江科技, 2018, 39(11):49-50.

[16] Lee Y, Waterman A, Cook H, et al. An agile approach to building RISC-V microprocessors[J]. IEEE Micro, 2016, 36(2):8-20.

[17] Levy A, Campbell B, Ghena B, et al. The case for writing a kernel in rust[C]. Proceedings of the 8th Asia-Pacific Workshop on Systems. New York: Association for Computing Machinery, 2017:1-7.

[18] Levy A, Andersen M P, Campbell B, et al. Ownership is theft: Experiences building an embedded OS in Rust[C]. Proceedings of the 8th Workshop on Programming Languages and Operating Systems. New York: Association for Computing Machinery, 2015:21-26.

[19] 孙卫真, 刘雪松, 朱威浦, 等. 基于RISC-V的计算机系统综合实验设计[J]. 计算机工程与设计. 2021,42(4):1159-1165.

[20] Tofte M, Talpin J P. Region-based memory management[J]. Information and computation, 1997, 132(2):109-176.

[21] Gay D, Aiken A. Memory management with explicit regions[C]. Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation. New York: ACM, 1998: 313-323.

[22] Crary K, Walker D, Morrisett G. Typed memory management in a calculus of capabilities[C].Conference Record of POPL'99. New York: ACM, 1999:262-275.

[23] 魏海涛, 姜昱明, 李建武, 等. 内存管理机制的高效实现研究[J]. 计算机工程与设计, 2009, 30(16):3708-3712.

[24] 杨峰. 基于 Linux 内核的动态内存管理机制的实现[J]. 计算机工程, 2010, 36(9):85-86.

[25] 孙温稳. 操作系统内存管理的实现[J]. 河南科技, 2016(3): 62-63.

[26] 吴敬仙, 缪行外. 操作系统存储管理方法与教学演示系统[J]. 计算机教育, 2009(14):131-132.

[27] 徐艳艳,陈志泊,田萱.《C++程序设计》的动态内存管理教学探讨[J].现代计算机(专业版),2012(23):34-38.

[28] Caballero J, Grieco G, Marron M, et al. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities[C]. Proceedings of the 2012 International Symposium on Software Testing and Analysis. New York: Association for Computing Machinery, 2012:133-143.

[29] Chandrasekhar Boyapati, Robert Lee, Martin Rinard. Ownership Types for Safe Programming: Preventing data races and Deadlocks[J]. ACM SIGPLAN Notices: A Monthly Publication of the Special Interest Group on Programming Languages, 2002, 37(11):211-230.

[30] Meyer B, Kogtenkov A, Stapf E. Avoid a void: The eradication of null dereferencing[M]. London: Reflections on the Work of CAR Hoare, 2010.

[31] Waterman A, Lee Y, Avizienis R, et al. The risc-v instruction set manual volume 2: Privileged architecture version 1.7[R]. California: University of California, Berkeley, 2015.

[32] Cox R, Kaashoek M F, Morris R T. Xv6, a simple Unix-like teaching operating system[EB/OL]. http://pdos.csail.mit.edu/6.828/2014/xv6.html, 2016.

[33] Stroustrup B. Exception safety: concepts and techniques[M]. Advances in exception handling techniques. Springer, Berlin, Heidelberg, 2001.

[34] Aparicio Rivera J. Real time Rust on multi-core microcontrollers[EB/OL]. https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1391552&dswid=-6351, 2020.